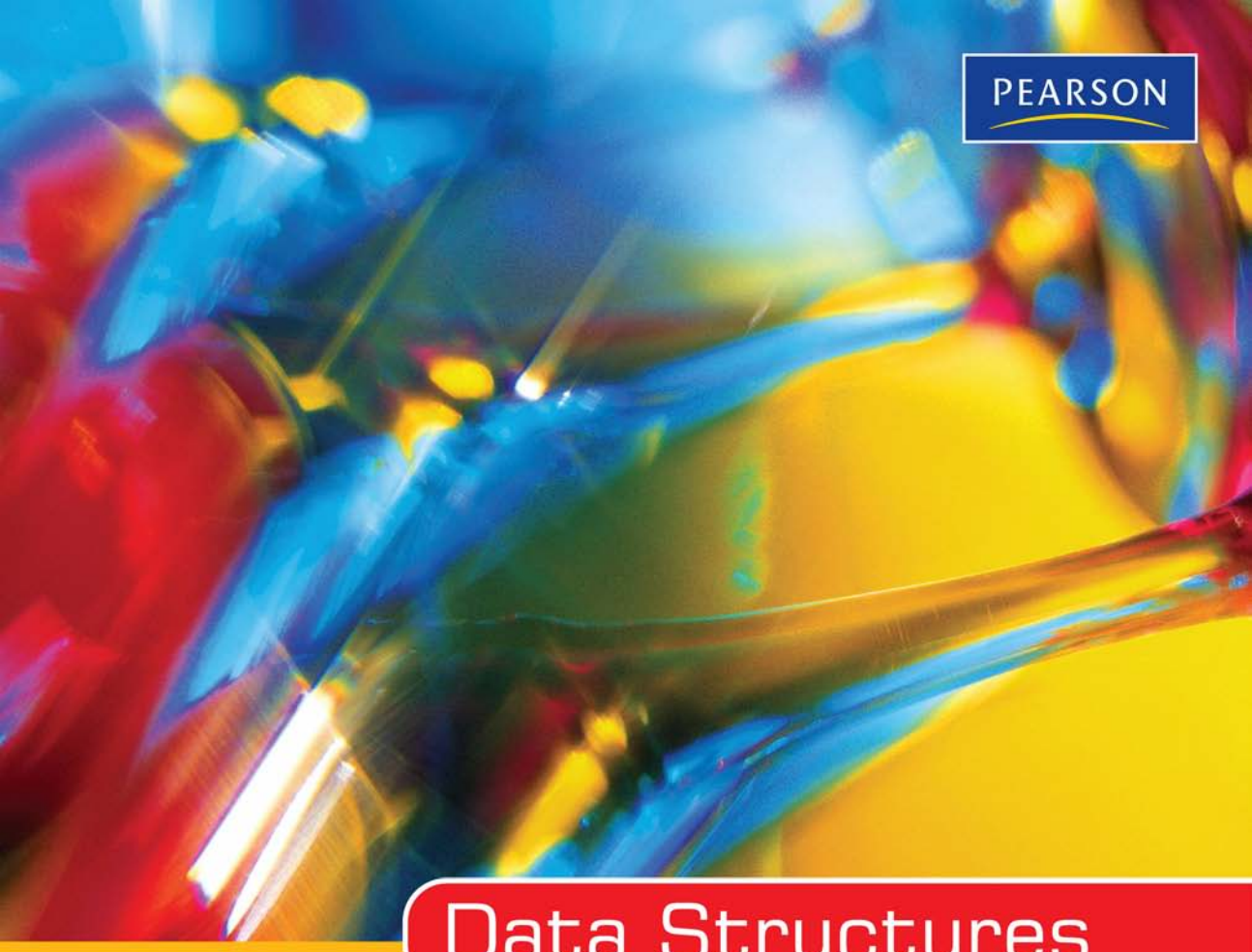


PEARSON

Data Structures and Algorithms Using C++

Ananda Rao Akepogu

Radhika Raju Palagiri

Data Structures and Algorithms Using C++

Ananda Rao Akepogu
Professor
Computer Science and Engineering Department
JNTUA, Anantapur, Andhra Pradesh

Radhika Raju Palagiri
Lecturer
Computer Science and Engineering Department
JNTUA, Anantapur, Andhra Pradesh

PEARSON

Chennai • Delhi • Chandigarh

Assistant Acquisitions Editor: S. Shankari
Associate Production Editor: Jennifer Sargunar
Composition: Sigma Publishing Services, Chennai
Printer: Chennai Micro Print, Chennai

ISBN 978-81-317-5567-9

10 9 8 7 6 5 4 3 2 1

Published by Dorling Kindersley (India) Pvt. Ltd, licensees of Pearson Education in South Asia.

Head Office: 7th Floor, Knowledge Boulevard, A-8(A), Sector-62, Noida 201 309, UP, India.
Registered Office: 11 Community Centre, Panchsheel Park, New Delhi 110 017, India.

About the Authors

Ananda Rao Akepogu is Professor and Head of the Computer Science and Engineering Department and Vice Principal of Jawaharlal Nehru Technological University Anantapur College of Engineering Anantapur. He is an alumnus of IIT Madras, Chennai. He received his B.Tech. (CSE) and M.Tech. (AI and Robotics) degrees from the University of Hyderabad, Andhra Pradesh, and his Ph.D. degree from IIT Madras, Chennai. He has 18 years of teaching experience in JNT University. He has published around 50 papers in various international and national journals and conferences. He has also worked on *Programming in C and Data Structures* by J. R. Hanly, E. B. Koffman and A. Kamthane and *Database Management Systems* by Peter Rob and Carlos Coronel. His research interests include Artificial Intelligence, Software Engineering, Cloud Computing and Object-oriented Databases.

Radhika Raju Palagiri is presently working as an ad hoc lecturer in the Computer Science and Engineering Department, Jawaharlal Nehru Technological University Anantapur College of Engineering Anantapur. She received her M.Tech. in Computer Science from JNTU Anantapur. She has 13 years of teaching experience at the undergraduate and postgraduate levels. Her areas of interest lie in Software Engineering and Expert Systems.

Contents

Preface

The study of data structures is a basic component of Computer Science and Engineering. People who work in any phase of a software system require a bare minimum knowledge of data structures. So, all Computer Science, Engineering and other allied disciplines offer courses on this subject. The hand-in-hand relationship between data structures and their algorithms and their complexity analysis is also a point of discussion.

Data Structures and Algorithms Using C++ helps one to master data structures, their algorithms and analysis of complexities of these algorithms.

This book meets the requirements of the course curricula of almost all the universities. Every chapter of the book includes the Abstract Data Type (ADT) and applications along with a detailed explanation of the topics. We present an implementation vehicle for these data structure concepts using C++ programming language known for its power, reusability and portability.

The first two chapters of the book are dedicated to reviewing C++ programming concepts and object-oriented concepts. Those who are familiar with programming in C can cope with C++ by going through Chapters 1 and 2.

Chapter 1 discusses about the class, defining class members and object declaration. It covers I/O streams, access control, class scope and `static` class members. It also includes functions, `this` pointer, dynamic memory allocations and exception handling.

Chapter 2 explains object-oriented goals and principles, constructors, destructors, overloading of constructors and operators. It also covers inheritance, polymorphism, abstract classes, generic programming with function, class templates and recursion.

Chapter 3 provides an extensive study of algorithms. Starting with basic notations, types of algorithms and performance analysis of algorithms are covered. A very detailed explanation of space complexity—the instruction space, data space and stack space—is discussed in depth. It also includes apriori analysis, asymptotic notations (Big oh, omega, theta and little oh) and time complexity.

Basic and linear data structures are covered in Chapters 4—7. Chapter 4 defines and classifies the data structures into linear and non-linear data structures. It provides a detailed study of a linear data structure—array, its types and representation. Array initialization, accessing values of an array, array operations, arrays passed as parameters and character sequences are also explained.

Chapter 5 discusses and exemplifies the linked list. The static and dynamic representations of a linked list are explained. It covers singly linked lists, circular linked lists and doubly linked lists along with representations and operations.

Stacks and queues are exclusively dealt with in Chapters 6 and 7. The representation and implementation of these data structures are explained along with examples. Various operations of stacks are discussed in Chapter 6. Circular queues and doubly ended queues are explained in Chapter 7.

Chapter 8 covers dictionaries and linear lists, skip lists and hash table representations of dictionaries. It includes collision occurrences, techniques to overcome the collisions and comparisons of chaining and open addressing.

The study of non-linear data structures is considered from Chapter 9 onwards. Chapter 9 defines the tree, its related terminology and representations. It covers binary tree—its representation, operations, traversals and threaded binary trees. It also includes the conversion of a general tree into a binary tree.

Chapter 10 discusses graphs, the basic terminology of graphs, its representations and operations. Breadth and depth first traversals of a graph are also exemplified. It also includes Prim's Algorithm in finding the minimal cost spanning tree of a graph.

Chapter 11 defines priority queues, implementations of priority queues and operations on min and max heaps. It includes external sorting using multiway merge and polyphase merge. Chapter 12 deals with the definitions of binary search trees and AVL trees along with their operations. It lists the drawbacks of binary search trees and states the need of AVL trees as a remedy. Chapter 13 is dedicated to the discussion of multiway trees and B trees. It includes the node structure of an m-way tree, its operations, drawbacks and the need for B trees. It also covers B tree operations and other variations of B trees.

Chapter 14 introduces red–black trees and splay trees. It also discusses the popular operations on red–black trees. Splay trees, splay rotations and amortized analysis with respect to splay trees are also explained. Pattern matching and tries are introduced along with terminology in Chapter 15. It includes the Brute Force Algorithm, the Boyer–Moore Algorithm and the Knuth–Morris–Pratt Algorithm in detail. Categories of tries namely standard tries, compressed tries and suffix tries and their operations are also explained in detail.

The last chapter, Chapter 16, is about sorting and searching. It covers bubble sort, insertion sort, selection sort, quick sort, merge sort, shell sort, radix sort and heap sort. It also includes popular and frequently used searching techniques like linear search, binary search and Fibonacci search.

Every chapter discusses the topics clearly in depth illustrated with examples, appropriate diagrams and tables. From Chapter 4 onwards, each chapter is provided with Abstract Data Type (ADT) of the corresponding data structures and several real-world applications explained clearly.

The concepts and techniques regarding each data structure, their representation, implementation and applications are clearly exemplified. Each chapter includes clear and detailed codes of programs with sample data and outputs, which help a student with minimum knowledge of programming in understanding and extending them to other operations also. The technical content of the chapter is summarized at the end of each chapter. The exercise part of the chapter helps the students in self-testing their skills theoretically and practically (laboratory sessions).

An overall emphasis of concepts, programs, applications and exercises makes a student know, understand, implement and feel comfortable to work with data structures. This book can help a student to start with the introduction of data structures and to end by mastering even the advanced concepts of data structures.

This book can be used as a text book for undergraduate, graduate and research programmes that offer data structures as a course. The book aims to serve as a course material for use in the classroom. It can also be used in laboratory sessions as a companion guide.

This book is one among very few books that include widely used data structures, pattern matching and tries explained simply and clearly along with examples and programs.

Utmost care has been taken in writing this book to make it free from errors. However, should you come across any error, please do not hesitate to contact us. Your suggestions and feedback may be sent to akepogu@gmail.com.

Ananda Rao Akepogu
Radhika Raju Palagiri

Acknowledgements

I would like to thank my family members—my mother Janakamma (in memory of father), my wife Prasanna, my daughter Anusha and my son Joel. I would also like to thank the ROSE (Research on Software Engineering) laboratory students, particularly Kalpana, who helped me and encouraged me through all the phases of the manuscript.

It has been a pleasure to work with the Pearson team towards this endeavor.

Ananda Rao Akepogu

I would like to extend my gratitude towards my husband Venkat, my beloved children Janvitha and Hitesh Phalgun, my father Sri. P. Krishnam Raju, my mother Smt. V. Sarojamma, my sister Latha and my brother Sathish.

Radhika Raju Palagiri

Chapter 1

Introduction to C++

The study of structures and algorithms using programs for solving day-to-day problems requires a very effective method of representing both data and algorithm. Before moving towards the data structures and algorithms, it is essential to have efficient C++ programming skills.

Chapter 1 provides a review of required skills that are well known to the readers. This chapter describes about some of the features of C++ language. It gives an introduction to C++, class overview that covers how a class and the class member are defined, and its objects declared. Also I/O streams, access control, class scope, and static class members are covered. This provides a detailed discussion about functions, `this` pointer, and dynamic memory allocations—the new and delete operators along with exception handling.

1.1 INTRODUCTION

C++ was first invented by Bjarne Stroustrup in 1979 at Bell Laboratories, Murray Hill, New Jersey (USA). Features of SIMULA 67 and C are incorporated into C++, which could support the features of object-oriented programming. Initially the new language was called C with classes, later in 1983, this was named C++ by Dick Mascitti. The name C++ originated from the increment operator ++ of C, so that C++ was called an extension of C. The name of C++ can be read as "one more than C," "next C," or "successor to C". It is pronounced as "see plus plus."

The standardization of C++ began in 1990 under the American National Standards Institute (ANSI) and the International Standard Organization (ISO) and underwent various revisions. Finally in 1998, C++ was standardized and was referred to as "Standard C++."

1.2 CLASS OVERVIEW

Class overview deals in detail with class, members of a class, i.e. both member variables and member functions. It also discusses the objects of a class.

1.2.1 Class

Class is one of the most important features of C++. A class is a collection of variables of different data types and functions. The variables in a class are called member variables, and the functions in a class are called member functions or methods. A class is created with the keyword `class`. Class is similar to structure the only difference being that all the members of a class are by default private, whereas all the members of structure are public by default. Syntactically, class is similar to structure.

The general form of class is as follows:

```
class Class-name
{
   private:
      members
         acess-specifier1
      members
         acess-specifier2
      members
   - - - - - - -
   - - - - - -
};
```

The body of the class may contain members that are either data or a function declaration. The class - name is the name of the class, the access - specifier can be one of the three keywords `public`, `private` or `protected`, which will be discussed later. By convention the class name should begin with an uppercase letter.

```
class Shape
{
   int l,b;
   public:
   void area()
   {  cout<<"area of rectangle is:"<<l*b;
   }
};
```

In the above program `Shape` is class. The variables `l`, `b` are private and the function `area()` is public. Every class must terminate with a semicolon.

1.2.2 Objects

Class defines a new data type that combines both the code and the data, and this new type can be used to create an object of that class. So, an object is an instance of a class. A class has a logical appearance, whereas an object has physical existence.

The declaration of an object is similar to the declaration of variables.

```
float x,y;                          //declaration of variables
Shape s1,s2                         //declaration of objects with type Shape.
```

For the above class, `Shape s1, s2` are the objects. `s1` will hold the variables `l,b` values and function `area()`. Similarly, `s2` will also have the copy of members of the `Shape` class.

An object holds data as well as the methods that operate on data.

1.2.3 Class Members

Class members include member variables or data variables and member functions or methods. Class is a combination of member variables and member functions. [The difference between method and function is that the function defined inside a class is called a method.] Class can control access to members from outside the class using access specifiers.

The members of a class can be accessed by using the object of that class and dot (.) operator.

```
class Shape
{
   public:
   int a,b;
      void set()
   {
      - - - - - -
   }
};
```

Suppose `s` is an object of class `Shape` then the class members are accessed through the object based on access control such as

```
 s.a=10, s.b=5
 s.set();                          //calling the member-function using object s
```

Program 1.1

```
#include<iostream.h>
#include<conio.h>
class Shape
{
   int a,b;   //by default private
   public:
   void area()
   {
      cout<<"Enter a&b values:";
      cin>>a>>b;
      cout<<"a="<<a<<"b="<<b;
   }
};
void main()
{
   Shape s1;
   s1.area();
}
```

Output

```
Enter a&b values:  5  6
a=5    b=6
```

In Program 1.1, class `Shape` is defined with the variables `a, b` and a method `area()` operates on variables. The object `s1` is created and it has a copy of variables `a` and `b`.

1.3 I/O STREAMS

Stream is a logical entity of an Input/Output (I/O) device. A stream can produce or consume information, and it is directly linked with the I/O subsystem. A stream is directly connected to the physical device. Whenever read or write operations are performed on the streams, they are reflected on the physical I/O device. C++ provides a rich set of I/O library.

Two different template classes are available in C++. One is for 8-bit streams and the other is for 16-bit streams (wide characters). The class `ios` is the base class for I/O streams. When a C++ program is loaded into the memory the standard input stream (`cin`), the standard output stream (`cout`), the standard error output stream (`cerr`) and the buffered version of standard error output streams will be automatically opened. When anything is typed through the keyboard, the input stream can be gathered by using the `cin` stream. When anything is written on to the standard output stream using `cout`, the character stream will be displayed on the standard output (screen). If any error is identified at the time of execution or at the compile time, then those messages will be passed on to the screen using `cerr`.

Formatting Input/Output: C++ provides two ways to define the input or output:

1. By formatting the `ios` class members (flags)
2. By making the use of I/O manipulators

The `ios` class consists of different status flags for the formatted I/O. The `ios` flags can be set or unset. When the flag is set, the following operations will be performed during the I/O transaction.

Different flags in the `ios` class:

adjustfield: left, right and internal fields are collectively called adjustfield.
basefield: oct, hex and dec fields are collectively called basefield.
boolalpha: boolean values will be accepted by the keywords true or false.
dec: I/O numeric system will be set to decimal.
fixed: Floating point numbers will be displayed in the normal notation.
floatfield: Scientific and fixed fields are collectively called floatfield.
hex: I/O numeric system will be set to hexadecimal.
internal: The numeric values will be padded by inserting spaces between any sign and the base character.
left: Ouput is left justified.
oct: I/O numeric system will be set to octal.
right: Output is right justified.
scientific: Floating point numbers will be displayed in the scientific notation.
showbase: Numeric base system will be displayed while showing the output.
showpoint: The decimal point will be displayed for the floating point values.
showpos: Displays the positive or negative sign before the value.
skipws: is used to skip the white spaces while performing the input operations on a stream.
unitbuf: I/O buffer is flushed after the insertion operation.
uppercase: Characters will be displayed in the uppercase.

Setting the `ios` format flags:

The `ios` class member function `setf()` is used to set the flag. The general usage of `setf` is as follows :

```
fmtflags setf(fmtflags flags);
```

Program 1.2

```
#include<iostream.h>
void main()
{
   int a=10;
   cout.setf(ios::showpos);
   cout <<"\n a="<<a;
}
```

Output

```
a=+10
```

In Program 1.2 for the `cout` stream, `showpos` flag is set. So the sign of the variable `a` is displayed.

Program 1.3

```
#include<iostream.h>
void main()
{
   float a=10.2;
   cout.setf(ios::showpos|ios::showpoint);
   cout <<"\n A="<<a;
}
```

Output

```
 A=+10.200000
```

If the `showpoint` flag is not set, then the value +10.2 will be displayed. Because `showpoint` and `showpos` flags are set, the value 10.2 is shown as +10.200000. Program 1.3 demonstrates that two or more flags can be set by making use of the pipe symbol (|).

Unset the `ios` flags: As `setf()` is used to set the flags of `ios`, the `unsetf()` member function of `ios` class is used to unset the set flags. The general form is as follows:

```
void unsetf(fmtflags flags);
```

Program 1.4 unsets the flags that are set in Program 1.3.

Program 1.4

```
#include<iostream.h>
#include<conio.h>
void main()
```

```
{
   float a=10.2;
   clrscr();
      cout.setf(ios::showpos|ios::showpoint);
      cout<<"\nAfter Setting the flags the 'a'="<<a;
      cout.unsetf(ios::showpos|ios::showpoint);
      cout<<"\nAfter Clearing the flags the 'a'="<<a;
}
```

Output

```
After Setting the flags the 'a'=+10.200000
After Clearing the flags the 'a'=10.2
```

Additional `ios` class methods: The class `ios` provides the additional member functions `width()`, `precision()` and `fill()`. The width function can be used to set the minimum field size. The method precision can be used to set the floating point precisions. In addition, the fill method allows the developer to modify the default fill value with the user-specified character. The general forms of width, precision and fill methods are as follows:

```
streamsize width(streamsize w);
streamsize precision(streamsize p);
char fill(char ch);
```

Program 1.5

```
#include<iostream.h>
#include<conio.h>
void main()
{
   float a=10.123456;
   clrscr();
   cout.precision(2);
   cout.width(10);
   cout.fill('#');
   cout<<a;
}
```

Output

```
#####10.12
```

It can be observed from Program 1.5 that because the width is set to 10 characters, the output is displayed as 10 characters wide. Because the precision is set to 2 characters, 10.12 is displayed instead of 10.123456. Because the default justification is right, five empty characters will be filled by the # character.

Input and output manipulators: The Input/Output manipulators are the members of `ios`. The input and the output streams can be manipulated using the manipulator member functions. These member functions do not take an input argument. The `setiosflags` manipulator is similar to the `setf` member function of the `ios` class. Both the functions do the same.

Program 1.6

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
void main()
{
   float a=10.2;
   clrscr();
   cout<<endl<<"\nAfterSetting the flags the 'a'="<<setiosflags(ios::showpos|
ios::showpoint)<<a;
}
```

Output

```
After Setting the flags the 'a'=+10.200000
```

In Program 1.6 the `endl` and the `setiosflags` manipulators are used. The `endl` manipulator is used to print in a new line.

Program 1.7

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
void main()
{
   clrscr();
      cout<<setiosflags(ios::showbase);
      cout<<hex<<10<<endl;
      cout<<oct<<10<<endl;
}
```

Output

```
0xa
012
```

Here `hex` and `setiosflags` manipulators are used to manipulate the `cout` output stream (Program 1.7). When the showbase flag is set, numerical values are prefixed with a base format such as hexadecimal values are prefixed with '0x', octal values with '0' and no prefix for decimal values. The first output is the hexadecimal value of 10 prefixed with '0x' and the second output is the octal value of 10 prefixed with '0'.

1.4 ACCESS CONTROL

The access specifier provides access rights for the class members from outside the class. An access specifier is one of the following keywords:

`public`, `private`, and `protected`.

- By default, all the members of a class are private and they cannot be accessible outside the class. The `private` members of a class can be accessible through the `public` members of the same class.

- The member variables and functions that are declared as `public` can be accessed by the members outside the class.
- The access specifier `protected` works similar to the keyword `private`. It is frequently used in inheritance, which is discussed in the Chapter 2.

Syntax of access specifiers: The keyword `private`, `public` and `protected` are followed by a colon (:).

```
Class   Class-name
{
   private or public or protected:
   members;
}
```

Program 1.8

```
/*Program for private and public access specifier*/
#include<iostream.h>
#include<conio.h>
class Shape
{
   private:
   int a,b;         // private section
   void display()
   {
      cout<<"a="<<a<<"b="<<b;
   }
   public:         // public section
   void sum()
   {
      a=10, b=20;    //using  private numbers.
      cout<<"sum="<<a+b;
   }
};
   void main()
   {
      clrscr();
      Shape  s;
         //s.a=10;             //private variables not accessible.
         //s.b=20;             //private variables not accessible.
         //s.display();        //private member function not accessible.
      s.sum();                 //calling public member function.
   }
```

Output

```
Sum : 30
```

In Program 1.8, the variables `a`, `b` and the method `display()` are defined in the private section and the method `sum()` is defined in the public section. The object `s` is created for the class shape. The object `s`, cannot

be used to initialize private variables directly. When it is used, the compiler gives an error message as `shape::a` is not accessible, i.e. the private members are not accessible outside the class, and the private member function `display()` also cannot be accessed directly by the object. Using the public member function `sum()`, the private data members are accessed and the addition operation is performed on them.

The private member function can be accessed by the public member functions. The member function can also be defined outside the class, and then their prototype must be declared inside the class.

Program 1.9

```
#include<iostream.h>
class Shape
{
   int a,b;
   void set_val()    //private function.
   {
      a=10,b=20;
   }
   void display()
   {
      cout<<"a="<<a<<"\nb="<<b;
   }
   public:
   void sum()  //public section
   {
      set_val();
      display();    //call to private Functions..
      cout<<"\nSum="<<a+b;
   }
   void sub(void);
};
void Shape::sub()
{
   a=30;
   b=10;
   cout<<"\nSubtraction="<<a-b;
}
void main()
{
   Shape s;
   //s.set_val();                //not accessible
   s.sum();
   s.sub();
}
```

Output

```
a=10
b=20
```

```
Sum=30
Subtraction=20
```

In Program 1.9, the method `set _ val()` declared in the private section, and methods `sum()` and `sub()` are declared in the public section. The method `sub()` is defined outside the class that performs subtraction on `a` and `b`. The private method `set _ val()` is accessed by making a call to it in the public member function `sum()`.

1.5 CLASS SCOPE

Identifiers such as the variable name, function name, `typedef` in C++ can be used only in certain areas of a program. This area is called the "scope" of the identifier. Scope determines the lifetime and visibility of an identifier. It also determines whether the identifier clashes with another identifier in any other parts of the program. There are five kinds of scope.

a. Local scope: Variables declared within a function or blocks have local scope, i.e. they are accessed only within that block. If the same variable is used in more than one function then each function, defines a new scope for it. Blocks can also be nested. A name declared in a block is local to that and to all the blocks contained in it. The formal arguments to a function will have local scope and are treated as if they belong to the outermost block.

```
{
   int i;
}
```

Here `i` has local scope, because it is declared inside a block enclosed by braces and it is never accessible since no code accesses it before closing the braces.

b. Function scope: Labels are those that have function scope and they can be accessed only within the function but not outside of that function.

c. File scope: When an identifier is not part of a function or class definition, then it is said to have file scope. File scope is the outermost scope in the program and it encloses both local and class scope. Variables defined at the file scope are called global variables.

d. Class scope: Every C++ class defines a unique scope for its members called the class scope, i.e. data members and member functions have class scope.

```
class A
{
     int a;
     public:
     viod b();
}
```

Here the variable `a` and function `b()` must be accessed, so that the compiler determines its class. Accessing a class member is done in three ways:

i. *Using .(dot) operator*:
`obj.a,`
`obj.b()`

ii. *Using -> operator:*

```
ptr->a
ptr->b()
```

iii. Using :: (scope access) operator:

```
A::a
a::b()
```

e. Prototype scope: Variables declared in a function prototype are visible only to the end of the prototype.

```
void val(int a, float b, char c)
```

The above prototype declares three names and they go out of scope at the end of the prototype.

1.6 STATIC CLASS MEMBERS

Every variable in C++ program is associated with a unit of storage in memory. C++ has two kinds of storage classes: `automatic` and `static`. Static members will be accessed using . (dot) or -> operators. Unlike normal class members, static members may be directly accessed by applying :: scope access operator to the class name. It is not necessary to create objects for accessing the static members.

1.6.1 Static Member Variables

When objects are created for a class, each object will have its own copy of members of class. The member functions are created ones, and all the objects will share them. There is no separate copy of every function of each object. Like member functions, the member variables can also be made common by using the keyword `static`. It makes the variables persist from the point of execution in which they are created until the program is terminated.

By declaring the member variables as static, only one copy of the variables is created and it is shared by all the objects of that class. Every static variable must be initialized to zero, the scope of the static variables is within the class. A static variable may be global to a particular transaction unit or global to a class. The general form is as follows:

```
static variable-declaration;
static function-definition;
```

Program 1.10

```
#include<iostream.h>
#include<conio.h>
class Sum
{
   static  int a;
   int b;
   public:
   void add()
   {
      b=0;
      a=a++;
      b=b++;
```

```
        cout<<"a="<<a<<"\tb="<<b;
    }
};
int sum::a=9;   //initialization of static variable
void main()
{
    Sum s1,s2;
       cout<<"\nvalues of a&b in object s1";
    s1.add();
       cout<<"\nvalues of a&b in object s2";
    s2.add();
}
```

Output

```
values of a&b in object  s1   a=10   b=0
values of a&b in object  s2   a=11   b=0
```

In Program 1.10, the class `sum` has two variables—one is static and the other is non-static. The static variables are accessed by the public member functions of the class. Only one copy of data variable `a` is created. Because it is declared to be static, and objects `s1` and `s2` share it, any change made to the static variable reflects in other objects.

1.6.2 Static Member Function

The member function in a class can also be defined as static, when the functions are declared as static, they can only access the static member variables and functions static member functions do not have access to nonstatic functions.

Static member functions are invoked by using the class name and they can also be invoked by using objects. The scope of the static method is global within the class.

Program 1.11

```
#include<iostream.h>
class Sum
{
    static int a;
    public:
    static void set()
    {
       a=10;
    }
    static void display()
    {
       cout<<"\nA="<<a;
    }
};
int Sum::a=0;
```

```
void main()
{
   sum::set();
   sum::display();
}
```

Output

```
A=10
```

In Program 1.11, the class `sum` is declared with one static data member and two static member functions. The static data members are accessed through static member functions only. The static functions are called by using the class name and scope access operators such as `sum::set()` and `sum::display()`.

Static member functions can also be declared as private and they are accessed through public static functions. Static variables declared as public can be initialized in the `main()` function like a normal variable by using the class name and the scope access operator.

Program 1.12

```
#include<iostream.h>
#include<conio.h>
int a=20;
class Sum
{
   private:
   static void set()
   {
      b++;
   }
   public:
   static int c,b;
   static void display()
   {
      set();
      cout<<"\nb value is:="<<b;
   }
};
int Sum::b=70;
void main()
   {
      int b=30;
      cout<<"\nClass variable"<<Sum::b;
      cout<<"\nGlobal variable"<<::a;
      cout<<"\nLocal variable"<<b;
      Sum::display();
   }
```

Output

```
Class variable 70
Global variable 20
Local variable 30
b value is:=71
```

In Program 1.12, the class `Sum` is declared with the private static function `set()`, public static function `display()` and public static variables `a`, `b`. The `set()` function is called by the public static function `display()` which increments the `b` value. The public variable `b` is initialized in the `main()` function.

1.6.3 Static Object

Static variables are generally initialized to zero. An object is a collection of members. Using constructors, objects are initialized to desired values. Constructors will be discussed in Chapter 2. The class members can also be initialized to zero using the static keyword. By declaring an object as static, all the members belonging to objects get initialized to zero.

Program 1.13

```
#include<iostream.h>
#include<conio.h>
class Sum
{
   private:
   int a,b;
   public:
   void add()
   {
        a=a+2;
        b=b+2;
   }
   void display()
   {
      cout<<"A and B values are:\na="<<a<<"\nb="<<b;
   }
};
void main()
   {
      clrscr();
         static Sum s;
         s.add();
         s.diaplay();
   }
```

Output

```
A and B values are:
  a=2;
  b=2;
```

In Program 1.13, the class sum has two member variables a, b and the member functions add() and display(). The object of the class sum s is declared as static. So, automatically all the data members of object s get initialized to zero. The variables a, b are not initialized to any value in the program. When the add() method is called, it adds 2 to each of the variable.

1.7 FUNCTIONS

A function is a subroutine or a subprogram. A large program can be divided in to smaller programs, and each small program can be written in the form of functions. Functions can be called repeatedly and the size of the program can be reduced. The general form is as follows:

```
return_type function-name(parameter_list)
{
   Body of the format;
}
```

The return-type specifies the type of the data that the function returns, which includes basic data types such as int, float, double, etc. and parameter list is the list of variable names with the data types. A function can have the parameter list as empty. If the function does not return anything then its return type is declared as void. Every function including main() returns an integer value .

```
/* main in c++*/
int main()
{
   Statements;
   -------;
   return 0;
}
```

The main() function in C++ always returns an integer value to the operating system by default. If it returns zero, the program is said to be successful, otherwise it is unsuccessful. The function main() can be declared as void without writing the return statement.

A function consists of the function declaration and the function definition. The function declaration involves the function return type, name and parameter list. The function definition must match the prototype of the function.

Example

```
void main()
{
    float area(float,int)                  //function prototype
    flaot x=2.5,y;
    int z=5;
    y=area(x,z);                           //function call;
}
float area(float i, int j)                 //function declarator
{
  return (i*j);                            //function body
}
```

In the above example, x, y are the actual arguments defined in the function call, and i, j are formal arguments declared in the called function. The function returns a float type value, then the return type of area() is float.

A function can be called anywhere in the program and can be called any number of times. A function must be called by using its name followed by parenthesis and terminated by a semicolon.

Advantages of a function:

- Supports modularity
- Duplication of code can be avoided
- Provides reusability

1.7.1 Parameter Passing Methods

Parameters can be passed to methods in three ways, which are as follows.

- Call by value
- Call by address
- Call by reference

The arguments in the caller function are actual arguments, and the arguments in the callee function are formal arguments. While passing values to the methods, the data type and the number of formal arguments and actual arguments must be the same.

Call by value: Using the pass by value method, the value of the actual arguments is passed to the formal arguments and operations are performed on the formal arguments. In this method any change made in the formal argument does not affect the actual arguments. Changes made to the formal arguments are local to the block of the function called.

Program 1.14

```
/* Program to explain call by value*/
#include<iostream.h>
#include<conio.h>
void main()
{
   int x,y;
   void input(int,int);  //prototype
      cout<<"Enter x&y values";
   cin>>x>>y;
      cout<<"values of the actual parameters are"<<"a="<<x<<"b="<<y;
   input(x,y); //function call.
}
void input(int a, int b)
{
   a=a+b;
   b=b+10;
```

```
        cout<<"\nvalues of formal parameter";
        cout<<"a="<<a<<"b="<<b;
}
```

Output

```
Enter x&y values 4 5
values of the actual parameters are a=4 b=5
values of formal parameter a=9 b=15
```

In Program 1.14, we pass values of actual parameters to method `input()`. The formal parameters `a`, `b` will receive these values, and the values `a` and `b` are changed. When the control goes back to the `main()` function, this change does not effect the actual parameters `x` and `y` values because the values of `a` and `b` are local to the function `input()`. Finally, the changes made in the formal arguments do not affect the actual arguments.

Call by address: In this method, the address of the actual arguments are passed to the formal arguments instead of values. The operations will be performed on the values at their address. So, any change made in the formal parameters will affect the actual parameters.

Program 1.15

```
/*Program to explain the call by address*/
#include<iostream.h>
#include<conio.h>
void input (int*x, int*y)
{
   int t;
   t=*x;
   *x=*y;
   *y=t;
}
void main()
{
   int a, b;
      cout<<"\nEnter the values of a, b:";
   cin>>a>>b;
      cout<<"\nA,B values Before swapping:\na="<<x<<"\nb="<<y;
   input(&a,&b);
      cout<<"\nAfter Swapping";
      cout<<"\nA="<<x<<"\nB="<<y;
}
```

Output

```
Enter the values of a, b:10 20
a, b values Before swapping:
a=10
b=20
After Swapping
a=20
b=10
```

In Program 1.15, we pass the address of the actual parameters x and y to the formal arguments *a and *b, which are pointers to the actual arguments. The function input() operates on the address of actual arguments through pointers, the address of the actual and formal parameters will be the same so any changes made in the formal parameters will reflect in the actual parameters.

Call by reference: In this method the reference of actual arguments can be passed to the formal arguments. The reference is an alias of the actual argument. The reference types declared with '&' operator are identical to the pointer variables but are not the same. Consider the following example:

```
float pi=3.14
float pi=&p
```

where &p is an reference of pi.
p=3.14, same as pi=3.14 any operation on p results the same as the operation on pi.

Program 1.16

```
/* Program for call by reference*/
#include<iostream.h>
void main()
{
   int a;
   void input(int &);
      cout<<"enter a value";
   cin>>a;
   input(a);
}
void input(int&r)
{
    r++;
      cout<<"Value in r:"<<r;
}
```

Output

```
Enter a value 7
Value in r:8
```

In Program 1.16, the function input(), receives the value of a by reference. The value of a is received by value of r. Since it is a reference of a, both will have the same memory location. Any change made to the variable r reflects the actual variable a. Thus, the value of actual variable is also changed using the call by reference method.

1.7.2 Inline Functions

Inline functions act like macros and the function allows multiple lines of code. When a function is preceded by the keyword inline, then the function is called an inline function. Whenever a normal function is called, the control transfers from the caller function to the callee function and after executing the control comes back to the caller function.

The mechanism involved in the inline function is that whenever a function is called, the control does not move to the definition of the function, rather the definition of the function is copied to the point where the function is called.

If the function is large, making it as an inline function reduces the execution speed of the program. Therefore, it is recommended that one use an inline for small functions. The CPU stores the memory address of the instruction at each function call because the function is stored in only one place and is executed there itself, and the register and the other process must be saved before the function call, which is time consuming. The general form of inline functions is as follows:

```
inline data type function-name(arguments)
{
   Body of the function.
}
inline int area(int i)
{
   return (i*i);
}
```

Program 1.17

```
#include<iostream.h>
#include<conio.h>
inline int rectangle(int y)
{
   return 5*y;
}
void main ()
{
   int x;
   clrscr();
      cout<<"enter the input value:";
   cin>>x;
   rectangle(x);
      cout<<"\nthe output is:"<<(x);
}
```

Output

```
enter the input value:5
the output is:25
```

In Program 1.17, the method `rectangle()` is an inline function. The inline specifies and intimates the compiler that `rectangle()` is an inline function, so copy the definition to the point where it is called.

1.7.3 The `friend` Function

In a class, the private data can be accessed only by the public member function of that class. C++ provides a mechanism in which a non-member can have access to the private member of a class. This is achieved by the

non-member function friend to the class, whose private data are to be accessed. Using the keyword friend a non-member function can be made the friend function.

```
class Class-name
{
   private data;
   public:
   function name();
   friend function-name();
};
```

The function name will be preceded by the keyword friend. A function can be made friend to any number of class. The declaration of the function proceeds with the keyword friend but not the definition.

Properties of a friend function:

- A friend function takes objects as arguments.
- They cannot access members directly like normal member functions. By using the object and dot operator, they can access the member of a class.
- The function can be declared anywhere in the class either in the private or public section.
- More than one function in a class can be friend or the entire class can also be declared as the friend class; here friendship is not exchanged, i.e. making class A friend to class B does not mean that class B is friend to class A.

Program 1.18

```
/* Program for friend function*/
#include<iostream.h>
#include<conio.h>
class Shape
{
   int age;
   char name[10];
   public:
   void set_val()
   {
        cout<<"\nEnter values\n";
        cout<<"\nEnter the name";
      cin>>name;
        cout<<"\nEnter age";
      cin>>age;
   }
   friend void display (Shape);
};
void display(Shape s)
{
   cout<<"\nName="<<s.name;
   cout<<"\nAge="<<s.age;
}
```

```
void main()
{
   Shape sh;
   sh.set_val();
   display(sh);
}
```

Output

```
Enter values:
Enter the name:Rahul
Enter age:24
Name=Rahul
Age=24
```

In Program 1.18, the class `shape` is declared with two variables and the function `set-val()`. It also has `display()` which is declared as `friend` of class `shape`. When an outside function is declared as `friend` to a class, it can access private data of that class. The function `set _ val()` reads values for variables and the `friend` function `display()` displays `name` and `age`.

A class can be made `friend` to another class. The `friend` class can access the members of class to which it is a `friend`.

Program 1.19

```
/*Program for friend class*/
#include<iostream.h>
#include<conio.h>
class Shape
class Rectangle
{
   private:
   int x;
   public:
   void set_val_x()
   {
      x=10;
   }
void display(Shape);
};
class Shape
{
   private:
   int y;
   public:
   void set_val_y()
   {
      y=20;
   }
```

```
friend void rectangle::display(Shape s);
};
void rectangle::display(Shape sh)
{
   cout<<"x="<<x<<"\ny="<<sh.y;
}
void main()
{
   clrscr();
   Rectangle r;
   r.set_val_x();
   Shape s1;
   s1.set_val_y();
   r.display(s1);
}
```

Output

```
x=10
y=20
```

In Program 1.19, two classes `Rectangle` and `Shape` are declared. Here, `rectangle` is the `friend` of `shape`. The member function of class `Rectangle` can access the member of class `Shape`. The `display()` method displays the values of members of both classes.

1.7.4 Function Overloading

C++ provides a facility called function overloading to serve the different functionalities with a common function name. Function overloading allows the developers to define two or more functions with a common name and with different arguments or with a different number of arguments within a class or within a program.

Example

```
 int area(int,int)
 int area(int,int,int)
double area(double)   //here the function area() has been
                      //overloaded and differentiated
                      //based upon data type and number of arguments.
```

Properties of overloaded functions:

- All the overloaded functions should share the common function name.
- They should differ with the number of arguments or data types of the arguments.

a. `area(int,int)`
`area(int,int,int)`

The above two functions can be overloaded though the data types of the arguments are the same, because they differ in the number of arguments.

b.
```
area(int,int)
area(float,float)
```

Here, the two functions can be overloaded, the number of arguments are the same but they differ in data types.

- No two functions should have the same number of arguments and the same type of argument sequence.
- An overloaded function can have default arguments.

```
area(int,int)
area(int l=10,int b=15,int h=20)
```

Here, the second version of `area()` has default values for variables `l`, `b`, `h`.

- The compiler invokes the function that matches with either the data types or the number of arguments.
- The compiler will do the automatic type casting while passing input arguments. If any matching function exists then that function will be called at runtime.
- The overloaded functions should not differ only in the return type.

Example

```
int area(int)
void area(int)              //these two functions cannot be overloaded
                              though they differ with data type.
```

Program 1.20 finds the area of different shapes (circle, rectangle and cube) using function overloading with the same function name and different input arguments.

Program 1.20

```
#include<iostream.h>

//function to calculate area of a circle
float area(float radius)
{
   return(3.141*radius*radius);
}

//function to calculate area of a rectangle
int area(int length,int breadth)
{
   return(length*breadth);
}

//function to calculate area of a cube
int area(int length,int breadth,int height)
{
   return(length*breadth*height);
}
void main()
```

```
{
   cout<<"\nArea of a circle:"<<area(2.1);
   cout<<"\nArea of a rectangle:"<<area(2,3);
   cout<<"\nArea of a cube:"<<area(2,3,4);
}
```

Output

```
Area of a circle:13.851809
Area of a rectangle:6
Area of a cube:24
```

In Program 1.20, the function `area(float radius)` is called when a one float value is passed as an input argument. The function `area(int length,int breadth)` is called when two integer values are passed as input arguments. The function `area(int length,int breadth,int height)` is called when three integer values are passed as input arguments.

Program 1.21 finds the area of different shapes (circle and square) using function overloading with the same function name and the same number of input arguments.

Program 1.21

```
#include<iostream.h>

//function to calculate area of a circle
double area(double radius)
{
   return(3.141*radius*radius);
}

//function to calculate area of a square
int area(int length)
{
   return(length*length);
}
void main()
{
   cout<<"\nArea of a circle:"<<area(2.1);
   cout<<"\nArea of a square:"<<area(2);
}
```

Output

```
Area of a circle:13.85181
Area of a square:4
```

In Program 1.21, `area()` is overloaded. One version of `area()` is called to calculate the area of a circle and another version of `area()` to calculate the area of a square.

Ambiguity in function overloading: The compiler performs implicit conversion when it is unable to match the function. For example, `char` is converted to `int` and `float` is converted to `double`. Therefore, no two functions are overloaded when they seem to be the same after typecasting is done.

Ambiguity is a situation where the compiler is unable to select the appropriate function when two or more functions are overloaded based on the arguments passed to the function. This situation occurs because of automatic typecasting done at the runtime when one data type is compatible with two or more data types. For example consider the following functions:

```
long area(long)
double area(double)
```

and if this function is called with an integer value as `area(20)`, then the compiler will be in confusion to decide which version of `area()` is to be invoked.

Ambiguity may also occur when constant values are passed to the functions directly instead of variables. When `int area(int,int)` and `float area(float,float)` are overloaded, calling function `area()` with constant values such as `area(2,4)` and `area(4.5,5.4)` leaves the compiler in confusion to decide which version of `area()` is to be invoked. After internal conversion the integer version of `area()` will be invoked for both of the above calls.

Finding the address of an overloaded function: To know the address of overloaded functions, they can be called through function pointers. If the method is not overloaded, by simply assigning the method address to the function pointer the method can be called.
If the method is not overloaded, the method can be called with the following generic form:

```
Function-pointer=method-name;
Ex: p=myfunction;
```

If the method is overloaded, the compiler does not know which function address should be considered. To avoid this ambiguity `p` should be declared with the method arguments.

```
Ex: p=myfuntion(arguments)
```

Program 1.22

```
#include<iostream.h>

//function to calculate area of a square
int area(int length)
{
   return(length*length);
}

//function to calculate area of a rectangle
int area(int length,int breadth)
{
   return(length*breadth);
}
void main()
{
   int(*fp1)(int a);        //pointer to int function(int)
   fp1=area;                //points to the function area(int a)
```

```
    int(*fp2)(int a,int b);
    fp2=area;
       cout<<"\nArea of a square:"<<fp1(2);
       cout<<"\nArea of a rectangle:"<<fp2(2,3);
}
```

Output

```
Area of a square:4
Area of a rectangle:6
```

In Program 1.22, the method `area` is overloaded, the function pointer `fp1` is declared as `int(*fp1)(int a)` and `fp1` is assigned with the method `area`. So that `fp1` will point to the method that is declared with one integer as an argument. The other function pointer `fp2` will be pointing to the `area(int a,int b)`, because `fp2` is declared as `int(*fp2)(int a,int b)`.

1.8 THE `this` POINTER

When a member function is called, an implicit argument is passed to it automatically and this argument is a pointer to the invoking object, this pointer is called `this`. The keyword `this` is a local variable that is always present in any of the non-static member functions, the declaration of the keyword `this` is not necessary. Suppose if `a` is an object of class A and has a non-static member function `fun()`, using the object this `fun()` is called as `a.fun()`. The keyword `this` in the body of `fun()` stores the address of `a`. A static member function does not have a `this` pointer.

Program 1.23

```
#include<iostream.h>
#include<conio.h>
class A
{
   int a;
   public:
      void set()
   {
         this->a=11;
         cout<<"\na="<<this->a;
         cout<<"\na="<<a;
   }
};
void main()
{
   A obj;
   obj.set();
}
```

Output

```
a=11
a=11
```

In Program 1.23, the statement `this->a=11` uses the `this` pointer to initialize the variable `a` with value 11, which is same as `a=11`. For a constant member function class A, the `this` pointer is of type `a*` to prevent modification of the object by itself. In the first `cout` statement, the value `a` is accessed by implicitly using the `this` pointer and in the second `cout` statement the `this` pointer is explicitly used with (->) arrow operator to access the variable `a`.

Program 1.24

```
#include<iostream.h>
#include<conio.h>
class Address
{
   private:
   char  c[10];
   public:
   void show()
   {     cout<<"\n the object address is"<<this;
   } };
void main()
{
   Address a1,a2,a3;
   a1.show();
   a2.show();
   a3.show();
}
```

Output

```
the object address is:0x8fc3ffec
the object address is:0x8fc3ffe2
the object address is:0x8fc3ffd8
```

In Program 1.24, the function is called using object `a1`, `a2`, `a3` and implicitly the function will have `this` pointer variable with its value as address of object, such as `this=&a1`, `this=&a2` and `this=&a3`.

1.9 DYNAMIC MEMORY ALLOCATION AND DEALLOCATION

C++ provides dynamic memory allocation and deallocation using two operators, `new` and `delete`. The operator `new` is used to allocate memory at runtime, and the `delete` operator is used to deallocate the memory.

1.9.1 The `new` Operator

The `new` operator works similar to the `malloc()` function, the only difference is that it can create objects, and returns the pointer of appropriate types. But `malloc()` just allocates the memory and returns the void pointer. The `new` operator allocates the required amount of memory from heap. When the object is created and the memory allocated, it must be deleted after its use, otherwise it may lead to some mismatch operations that may crash the system. The `new` operator can be overloaded. Using the `new` operator, memory for arrays and objects can be allocated dynamically. The general form of the `new` operator is as follows:

```
p-val=new type.
pointer-variable=new array-type [size]
```

Here pointer-variable is that creates a pointer to memory that hold the array type.

Example

```
p=new int[10];
```

This statement creates an integer array of size10 dynamically.

1.9.2 The delete Operator

The delete operator is used to delete the objects. The objects created using the new operator remains in the memory until they are released. The delete operator deletes the objects and releases the memory. The general form of the delete operator is as follows:

```
Delete point-val;
Delete[ ] p-val;
```

The new and delete operators can be overloaded like other operators.

Program 1.25

```
#include<iostream.h>
#include<conio.h>
void main()
{
   int x,*y,i ;
   y=&x;
   y=new int[10];
   for(i=0;i<=5;i++)
      y[i]=i;
   for (i=0;i<=5;i++)
      cout<<y[i];
   delete []y;
}
```

Output

```
y[1]=0        y[2]=1
y[3]=2        y[4]=3
y[5]=4        y[6]=5
```

In Program 1.25, integer variables x and *y are declared, and the pointer variable y is initialized with the address of x. Using the new operator, an integer array with 10 elements is allocated. A for loop is used to display the contents of *y. The delete operator releases the memory. Objects are also created dynamically

Program 1.26

```
#include<iostream.h>
#include<conio.h>
```

```
class Shape
{
   int a,b;
   public:
   void set_val()
   {
      a=10;
      b=20;
         cout<<"a="<<a<<"\nb="<<b;
   }
};
void main()
{
   Shape*s;
   s=new Shape();
   s->set_val();
   delete s;
}
```

Output

```
a=10   b=20
```

In Program 1.26, the statement `s=new Shape()` creates an object for the class Shape dynamically using the `new` operator.

Program 1.27

```
#include<iostream.h>
#include<conio.h>
#include<new.h>
#include<stdlib.h>
void main()
{
   clrscr();
   int i;
   void message();
      void*operator new(size _ t,int);
      void operator delete(void*);
         char*c=new('$')char[10];
         cout<<"first time allocation of memory:p="<<(unsigned)long(c)<<endl;
      while(i<10)
            cout<<c[i];
         delete c;
            c=new('*')char[64000u];
         delete t;
}
void  message()
```

```
{
   cout<<"\n memory insufficient";
      exit(1);
}
void*operator new(size _ t s, int v)
{  void*c;
   c=malloc(s);
   if(c==null)
      message();
      memset(c,v,s);
   return c;
}
void operator delete(void*dd)
{  free(dd);
}
```

Output

```
first time memory allocation:p=3267
$$$$$$$$$$
Memory insufficient.
```

In Program 1.27, the `new` and `delete` operators are overloaded. The `size _ t` is used to define the size of the object. The `message()` is called by the `new` operator when the `malloc()` function returns null. The `new` operator fails to allocate memory and calls the function `message()`, when the statement `c=new(' *') char[64000u]` requests memory allocation. The `delete` operator releases memory when called by using `free()`.

1.10 EXCEPTION HANDLING

Any mistake in writing the program results in an error, errors are of two types—syntax errors and semantic errors. The exception handling mechanism is used to reduce such errors. Exception is an error that occurs at runtime and the program terminates due to the exception. Exceptions are of two types—synchronous and asynchronous exceptions.

The exception handling mechanism detects the exception, intimates that the exception occurred, receives the error message and takes necessary actions without disturbing the users. The exception handling mechanism involves three keywords, `try`, `catch`, `throw`.

Whenever an exception occurs, the control passes from the point of exception to the exception handler, which is linked with the `try` block and is called only by the `throw` statement. The general form of `try`, `catch` and `throw` is as follows.

```
try
{
   Statement 1:
                          //try block
   Statement 2;
}
```

```
throw(excep);
throw excep;
throw;                          //re-throwing of an exception;
```

here `excep` can be any data type or it may a constant.

```
try
{
   S-1;
   S-2;
}
Catch(argument)
{
   Statement 3;                 //catch block
}
```

When an exception occurs, the `throw` statement inside the `try` block intimates the `catch` block that an error has been raised by throwing the exception. The errors that occurred only in the `try` block are used to throw the exception.

The `catch` block receives the exception thrown by the `throw` statement. It is executed when an exception is found. Arguments to the `catch` block are of exception type and they are optional. If no exception is caught just the `catch` block is ignored.

The types of arguments used in `throw` and `catch` statements must be same. Otherwise, the program will be aborted using the `abort()` function.

```
Statements;
Statements;                         //start of the exception handler
try
{
   Statement1;
      Statement;                    //try block
   throw(object);                   //finds the exception and throws it
   catch(object)
   {
      Statement;      //catch block statements which takes action on excep-
tion.
   }
}
```

Program 1.28

```
#include<iostream.h>
#include<conio.h>
void main()
{
   int a,b,c;
      cout<<"Enter a,b values";
   cin>>a>>b;
   c=a>b?0:1;
   try
```

```
    {
       if(c==0)
          cout<<"subtraction of a,b"<<a-b;
       else
          throw(c);
    }
 catch (int c)
    {
       cout <<"Caught exception c="<<c;
    }
 }
```

Output

```
Enter  values of a,b:2  7
Exeption caught:1
Enter values of a,b:7  4
Subtraction of a,b:3
```

In Program 1.28, a, b, c are integer variables. If a>b then c gets a value 0 else it is 1. If c=0 subtraction of a, b is displayed. Otherwise, exception is caught.

Multiple `catch` Statement: A `try` block can associate with more than one `catch` block; in such cases it may also have multiple `throw` statements.

```
try
{
   //try block
}
catch(argument)
   {
      catch block 1
   }
catch(arguments)
   {
      catch block 2
   }
- - -   - - -   - - -
- - -   - - -   - - -
- - -   - - -   - - -
catch(arguments)
{
   catch block n
}
```

When an exception is thrown, the compiler searches for the `catch` block that matches with the exception type thrown, then the `catch` block gets executed. If no match is found, the program terminates.

If the exception type matches with more than one `catch` block argument, then the first occurrence of `catch` that matches will be executed.

Program 1.29

```
#include<iostream.h>
#include<conio.h>
void sum(int s)
{
   try
   {
      if(s==0)throw k;
         else
      if(s>0)
      throw 'p';
         else if(s <0)throw .3;
      catch(char c)
      {
         cout<<"caught a positive value";
      }
      catch(int i)
      {
         cout<<"caught a null value";
      }
      catch(float f)
      {
         cout<"caught a negative value";
      } }
void main()
{
   cout<<"multiple catches:";
   sum(-5)
   sum(0);
   sum(10);
}
```

Output

```
Multiple catches:
Caught a Negative value
Caught a null value
Caught a positive value
```

In Program 1.29, the function `sum()` has a `try` block and there are `catch` blocks. The first `catch` block takes the `char` type object, the second `catch` block takes integer arguments and the third `catch` statement takes float type as arguments. The function `sum()` is called with positive, negative and null values, the argument `s` takes their values. When `s` receives a positive value, the `throw` statement receives a `char` type exception. When `s` receives an integer, throws integer type exception similarly when `s` is negative throws float type. The appropriate `catch` block is executed when the argument type in it matches with the argument of the throw statement.

A single `catch` block can be used to catch exceptions thrown by multiple `throw` statements with different types. The following form of the `catch` statement catches all exceptions:

```
catch(...)
{
catch block;
}
```

Program 1.30

```
#include<iostream.h>
#include<conio.h>
void sum()
{
   try
{
   if(s==0)throw k;
      else
   if(s>0)throw p;
      else if(s<0)throw 0.0;
}
catch(...)//catch all exception
   {     cout<<"caught all exception";
   } }
void main ()
{
   sum(-5);
   sum(0);
   sum(10);
}
```

Output

```
Caught all exeptions
Caught all exeptions
Caught all exeptions
```

In Program 1.30, all the exceptions are caught by the single `catch` block. `catch(...)` catches all the exceptions and processes them. It is also possible to throw the exception from the `catch` block. This is known as the re-throwing exception.

throw: The `throw` statement does not take any arguments. When re-throwing an exception, it will not be caught by the same `catch` statement but it is passed on to the next `catch` statement.

Program 1.31

```
#include<iostream.h>
#include<conio.h>
void sum(int i, int  j)
{
   cout<<"in function sum()";
   try
   {
      if(i==0)
         throw i;
      else
         cout<<"subtraction of I&j:"<<i-j;
   }
   catch(int)
   {
      cout<<"caught null values";
      throw;
   }
      cout<<"End of sum()";
}
void main()
   {
      cout<<"in main() function";
      try
      {
         sum(10,6);
         sum(0,5);
      }
   catch(int)
   {
      cout<<"caught null value inside main()";
   }
   cout<<"End of main()";
}
```

Output

```
In main() function
In function sum()
Subtraction of i&j:4
End of sum().
In function sum()
Caught null values
Caught null value inside  main()
End of main()
```

In Program 1.31, when the function `sum()` is called with null values, the `catch` block in it catches the exception, the `throw` statement associated with this `catch` block re-throws the exception and is caught by the same `catch` block. The `catch` block inside the main function processes the re-thrown exception.

SUMMARY

- A class is a collection of member variables and member functions.
- An object is an instance of a class. It holds data as well as the methods that operate on data.
- Stream is a logical entity of an I/O device. A stream can produce or consume information, and is directly linked with the I/O subsystem. The class `ios` is the base class for I/O streams.
- The access specifier provides access rights for the class members from outside the class. The `private` members of a class can be accessible through the `public` members of the same class. The member variables and functions that are declared as `public` can be accessed by the members outside the class. The access specifier `protected` works similar to the keyword `private` and are very useful in inheritance.
- Scope determines the lifetime and visibility of an identifier. Local scope, function scope, file scope, class scope and prototype scope are the different types of scopes.
- All the objects of the class share a single copy of the static members. Every static variable must be initialized to zero.
- Static functions can only access static member variables and the functions, its scope is global within the class.
- Inline function copies the function definition when it is accessed.
- The `friend` function allows access to private members of a class.
- Function overloading defines more than one function with the same name but with different operations.
- The `this` pointer refers to current invoking object.
- The `new` operator allocates memory dynamically, and the `delete` operator deallocates memory at runtime.
- Exceptions are runtime errors. `try`, `catch` and `throw` are used to handle exceptions.
- Re-throwing exception is throwing an exception from the `catch` block.

EXERCISES

Fill in the Blanks

1. An object is _____ of a class.
2. By default all the members of a class are _____ .
3. Static members are invoked by using _____ .
4. The difference between the `new` operator and `malloc()` functions is _____ .
5. Exception is _____ , which occurs _____ .
6. Exception handling involves _____ keywords.

Multiple-choice Questions

1. The `iostream` file contains _____ .
   a. Declaration of standard Input/ Output library
   b. Streams of include and output of program effects
   c. Both (a) and (b).
   d. None of the above

2. Which of the following is true?
   a. `cout` represents the standard output stream
   b. `cout` is not a member of `iostream`
   c. The default console for `cout` is keyboard
   d. None of the above

3. Static members can be accessed using _____ operators.
   a. . (dot)
   b. → (pointer)
   c. `new`
   d. Both a and b

4. When member variables are declared as static then _____.
   a. Each object will have a separate copy
   b. All objects will have a common copy of member variables
   c. They can be accessed by the objects
   d. None of the above

5. The following keyword is used to handle exceptions _____.
   a. `try`
   b. `throw`
   c. `catch`
   d. `rethrow`

6. In function overloading the functions differ by _____.
   a. Number of arguments
   b. Type of the arguments
   c. Return type
   d. Both a and b

7. Static functions are invoked directly by using _____.
   a. Objects
   b. Member functions
   c. Class name
   d. Scope access operator (::)

Short-answer Questions

1. Define class and objects.
2. Give the difference between an object and a variable.
3. Give the usage of access specifier.
4. What is a scope? Define various types of scopes.
5. What are static members and functions?
6. Give the properties of overloaded functions.
7. Define the `friend` class.
8. Discuss inline functions and `friend` functions.
9. Explain the `this` pointer.

Essay Questions

1. Discuss I/O streams in detail.
2. Explain the different ways of passing parameters to methods.
3. Illustrate in detail dynamic memory allocations.
4. What is an exception? Discuss the way of handling exceptions.
5. Explain the concept of function overloading with a suitable example.

Chapter 2

Object Oriented Concepts

In Chapter 1, a review of some important features of C++ have been covered. This chapter starts with Object- oriented Goals and Principles. It proceeds with the discussion about constructors, destructors, overloading of constructors and operators. It also concentrates on other important concepts of C++ such as inheritance, polymorphism, abstract classes, generic programming with function and class templates. Finally it covers recursion.

2.1 GOALS AND PRINCIPLES

Object - Oriented design has some goals that are to be achieved in developing a modern software. The object-oriented principles are to be followed in the design to facilitate the goals.

2.1.1 Object Oriented Design Goals

The software implementation should achieve the following important design goals:

- Robustness
- Adaptability
- Reusability

2.1.1.1 Robustness

Developers want to produce software, which gives right output for all the inputs, and the software must be robust, which is capable of handling unexpected inputs.

Robustness of software not only handles unexpected inputs but also yields accurate solutions. For example, if a user tries to store more elements in a data structure than expected then the software should have the capacity to expand this structure to take more elements. This feature is available in C++. The vector class of C++ standard template library defines a dynamically growing array.

Software should also achieve correctness for all its possible inputs including boundary cases such as when an integer value is 0 or 1 or the maximum or minimum possible values. However, robustness and correctness do not come automatically, they must be designed right from the beginning.

2.1.1.2 Adaptability

Adaptability is also called evoluability or portability. This is the ability that the software can work with minimal changes on all different hardware and open system platforms.

2.1.1.3 Reusability

Reusability is a process of reusing the existing code wherever it is required. As components of different systems are used in multiple applications, the software is reused. Developing quality software is very expensive. By reusing, the software cost can be reduced. If proper care is not taken when reusing the software, it leads to several software errors. Reusability of software saves cost and time.

2.1.2 Object Oriented Design Principles

The important principles of the object-oriented approach that serves the above goals are as follows:

- Abstraction
- Encapsulation
- Modularity

2.1.2.1 Abstraction

Abstraction is a mechanism to present only the essential features without including their background details. Consider Yahoo mail; when a user submits the username and password it opens the mail, when the user clicks on the compose button, it opens the compose box. However, the user does not know what are the actions performed internally to open the compose box. Just opening the compose box is an essential feature and internal actions performed to open the compose box is not an essential feature for the user. Thus, using abstraction what is to be performed is known but not how.

Abstract data types (ADT) have evolved by applying abstraction in the design of data structure. An ADT of data structure specifies the data that are stored, the fundamental operations preformed on them along with arguments and their types.

An ADT is an interface that deals with the performance of operations. However, not how they are performed. An ADT is represented as a class in C++.

2.1.2.2 Encapsulation

Wrapping of data and code into one single unit is known as encapsulation. It is also known as data hiding. The data are not accessible to the outside world. Combining code and data creates an object. Data hiding allows the users to use the object without knowing the internal working of it.

With encapsulation, the programmers can freely implement the details of a system but the only thing is that an abstract interface is to be maintained. Encapsulation supports adaptability by allowing only part of the program to be changed without affecting other parts of the program.

2.1.2.3 Modularity

A large software system can be divided into a number of components or modules that can be implemented independent of each other. Every module implements a set of related data structures and functions. Modern programming languages including C++ supports modularity.

Modular software design makes software maintenance easier and provides reusability. Modularity makes Object-oriented Programming (OOP) an effective programming method.

2.2 CONSTRUCTORS AND DESTRUCTORS

Constructors and Destructors are the special member functions that have the same name as that of the class through which they differ from other functions; the only difference with the destructors is they are preceded by ~ (tilde) operator. Whenever an object of a class is created, the constructor is invoked automatically. The destructor is used to destroy the objects when they are of no use. When constructors and destructors are not defined, the compiler executes the default constructor and destructor.

2.2.1 Constructors

Constructors are used to create and initialize the objects. For example,

```
class Rectangle
{
   Rectangle(){---} //constructor
   -----
}
```

Here the class `Rectangle` has the constructor `Rectangle()`. Whenever an object is created for this class, the constructor is invoked automatically and initializes the object; no need to call it explicitly.

2.2.1.1 Properties of Constructors

1. Constructor name must be the same as the class name.
2. It should not have a return type not even `void`.
3. Constructors can be overloaded.
4. They can also be invoked explicitly.
5. Constructors can have default arguments.

2.2.1.2 Types of Constructors

Constructors are of three types:

- Default constructors
- Parameterized constructors
- Copy constructors

Default constructors: The constructor without arguments is called a default constructor.

Parameterized constructors: Constructors that are created with arguments are called parameterized constructors. When an object is created for such constructors, values must be passed to the constructors.

```
class Area
{
   int l,b,h;
   public:
   Area(int x,int y,int z);      //declaration of parameterized constructor
   ------
   Area(){---} //default constructor
};
Area::Area(int i,int j,int k)    //definition of parameterized constructor
```

```
{
    a=i;
    b=j;
    c=k;
}
void main()
{
  Area a=Area();
  Area b=Area(1,2,3);
}
```

Here the class `Area` has a default and parameterized constructor. The declaration of the constructor with arguments is done inside the class and the definition is outside the class. Constructors can also be defined inside the class itself. The statements `Area a=Area()` and `Area b=Area(1,2,3)` create object and pass values to the constructor. While creating the objects, the values to the parameterized constructor must be passed, there is no other way.

Program 2.1

```
#include<iostream.h>
class  area
{
   int l,b,h;
   public:
   area()
   {
      cout<<"default constructor";
      l=0;b=0;h=0;
   }
area(int i,int j)
   {
      cout<<"\n constructor with two arguments";
      l=i;
      b=j;
   }
area(int x,int y,int z)
   {
      cout<<"\n constructor with three arguments";
      l=x;
      b=y;
      h=z;
   }
void display()
   {
      cout<<"\n\t l="<<l<<"b="<<b<<"h="<<h;
   }
};
```

```
void main()
   {
      area a=area();
      a.display();
      area b=area(10,20);
      b.display();
      area c=area(10,20,30);
      c.display();
   }
```

Output

```
Default constructor:
l=0 b=0 h=0
Constructor with two arguments
l=10 b=20
Constructor with three arguments
l=10 b=20 h=30
```

In Program 2.1, the first version of `display()` displays the contents of the default constructor, the second version of `display()` displays the contents of the constructor with two arguments and the third displays contents of the constructor with three arguments.

Copy constructors: When the reference of an object is passed as an argument to the constructors, such types of constructors are called copy constructors.

```
class area
{
   private:
   - - - - -
   - - - - -
   public:
      area(area);                //here the argument of constructor
                                 //is same as its  class which is invalid
      area(area&);               //copy constructor; here the reference of
                                 //the  area class object is passed,
};
```

Using copy constructors, objects can be declared and initialized by the reference of another objects. So, whenever a constructor is called a copy of an object is created.

Program 2.2

```
class area
{
   int a;
   public:
   area(int b)
   {
      a=b;
```

```
    }
    area(area&a1)  //copy constructor
    {
       a=a1.a;
    }
    void display()
    {
       cout<<a;
    }
};
void main()
{
    area a1(10);
    area a2(a1);
       cout<<"value of a in object a1";
    a1.display();
       cout<<"value of a in object a2";
    a2.display();
}
```

Output

```
value of a in object a1:10
value of a in object a2:10
```

In Program 2.2, class `area` is declared with one integer variable and two constructors; one is parameterized and other is a copy constructor. Object `a1` is created and with the reference of object `a1` object `a2` is created. The argument in copy constructor `&a1` receives the object `a2`.

2.2.2 Constructor Overloading

A class having more than one constructor is called constructor overloading. The constructors will be distinguished by the number of arguments. The compiler invokes the constructor that matches with the number of arguments. Constructor overloading allows both initialized and uninitialized objects to be created.

```
class Sum
{
   int a,b;
   public:
      Sum(int ,float b, float c){….}
      Sum(int a, int b){….}
      Sum(){…}
};
```

Suppose `S1(1,2.5,3)`, `S2(2,5,)` and `S3()` are the objects for this class. When object `S1` is created, a constructor with three arguments is called, for object `S2` a constructor with two arguments is called and for object `S3` a constructor with no arguments is called.

Program 2.3

A program for overloading the constructor with initialized input arguments:

```
#include<iostream.h>
class area
   {
                /*function to calculate area of a circle*/
   public:
      area(float radius);
      area(int length, int breadth);
      area(int length, int breadth, int height);
   };
area::area(float radius)
   {
      cout<<"\nArea of a circle:"<<(3.141*radius*radius);
   }
   /*function to calculate area of a rectangle*/
area::area(int length,int breadth)
   {
      cout<<"\nArea of a rectangle:"<<(length*breadth);
   }
/*function to calculate area of a cube*/
area::area(int length,int breadth,int height)
   {
      cout<<"\nArea of a cube:"<<(length*breadth*height);
   }
void main()
   {
      area circle(2.1);
      area rectangle(2,3);
      area cube(2,3,4);
   }
```

Output

```
Area of a circle:13.851809
Area of a rectangle:6
Area of a cube:24
```

In Program 2.3, constructors of the class `area` are overloaded and are called based on the input arguments.

2.2.3 Destructors

Destructors are also member functions that destroy the objects created by the constructors. Whenever the object goes out of its scope the destructor is executed and releases the memory space occupied by the object. The destructor name is also the same as the class name but it is preceded by (~) tilde.

Properties of destructors:

1. Destructors cannot be overloaded.
2. Destructors do not require any arguments, not even the return type.
3. Destructors are the only way to destroy objects.
4. Whenever the program is terminated by either return or exit statements, the destructor is executed.

Program 2.4

```
#include<iostream.h>
#include<conio.h>
class area
{
   area()
   {
      cout<<"object("<<c<<")created";
   }
   ~area()
   {
      cout<<"object("<<c<<")released";
   }};
void main()
   {
      Area a1,a2;
   };
```

Output

```
Object 1 created
Object 2 created
Object 2 released
Object 1 released
```

In Program 2.4, the class `area` declares two objects. The constructor invokes and initializes the objects. When the destructor is invoked, the object first created will be the last one to be released and the last one is released first.

2.3 OPERATOR OVERLOADING

Operator overloading is a procedure to change the meaning of the predefined operators to perform certain operations on the objects. The plus (+) operator is a predefined operator and can be used to add two or more values or variables. However, the same + operator cannot be used to add the variables of two or more similar objects. C++ provides the facility to modify the meaning of the operator (allows to redefine the predefined operators). After overloading the operator, it can be directly applied on the objects to perform the newly defined procedure.

```
x=y+z;
x=y-x;        //here x,y,z are basic data types.
```

The operators `+,=,-` can be used on these variables but when we use them on objects of a class, the compiler gives an error.

Program 2.5

```
/*Program for to  add two objects and store result in another object*/
#include<iostream.h>
#include<conio.h>
class vector
   {
      int x,y,z;
      public:
      vector(){}
      vector(int a,int b,int c)
      {
         x=a;
         y=b;
         z=c;
      }
   vector operator+(vector D)
   {
      vector T;
      T.x=x+D.x;
      T.y=y+D.y;
      T.z=z+D.z;
      return T;
   }
   void show()
   {
      cout<<x<<"i+"<<y<<"j+"<<z<<"k";
   } };
void main()
   {
      vector a(10,20,30),b(1,2,3),c;
      clrscr();
         cout<<"\nVector A:";
      a.show();
         cout<<"\nVector B:";
      b.show();
      c=a+b;
         cout<<"\nAddition of Vector A+Vector B:";
         cout<<"\nVector C:";
      c.show();
   }
```

Output

```
Vector A:10i+20j+30k
Vector B:1i+2j+3k
Addition of Vector A+Vector B:
Vector C:11i+22j+33k
```

In Program 2.5, a,b,c are the objects of vector class. Each object will have the copy of variables x,y,z. Addition of objects a and b means adding member variables of a and member variables of b and stored in object c. In Program 2.5, each variable is accessed individually, that is x of a and x of b are added and stored in x of c, similarly for other variables. We cannot directly perform as c=a+b; this can be overcome by using operator overloading.

The general form of operator overloading is as follows:

```
Return-type class-name::operator<symbol of the operator>(arguments)
{
      //Redefinition of the operator
}
```

2.3.1 Overloading the Plus (+) Operator

Program 2.6

```
/*Program to overload+operator*/
#include<iostream.h>
#include<conio.h>
class vector
{
   int x,y,z;
   public:
   vector(){}
   vector(int a,int b,int c)
   {
      x=a;
      y=b;
      z=c;
   }
   void show()
   {
      cout<<x<<"i+"<<y<<"j+"<<z<<"k";
   }
   vector operator+(vector v)
   {
      vector temp;
      temp.x=x+v.x;
      temp.y=y+v.y;
      temp.z=z+v.z;
      return temp;
   }
};
void main()
   {
      vector a(10,20,30),b(1,2,3),c;
         clrscr();
         cout<<"\nVector A:";
```

```
      a.show();
         cout<<"\nVector B:";
      b.show();
         c=a+b;
         cout<<"\nAddition of Vector A+Vector B:";
         cout<<"\nVector C:";
      c.show();
   }
```

Output:

```
Vector A:10i+20j+30k
Vector B:1i+2j+3k
Addition of Vector A+Vector B:
Vector C:11i+22j+33k
```

In Program 2.6, the `vector` is defined as a class with `x,y,z` as local variables. To solve the `vector` addition problem, the plus (+) operator cannot be applied on the objects of the `vector` class. By overloading the plus operator in the `vector operator+(vector v)` form, the addition problem of `vector` can be solved. The return type of the operator + here is a `vector` type.

2.3.2 Overloading the Minus (–) Operator

Program 2.7

```
/*Program to overload-operator*/
#include<iostream.h>
#include<conio.h>
class vector{
int x,y,z;
public:
vector(){}
vector(int a,int b,int c)
   {
      x=a;
      y=b;
      z=c;
   }
   void show()
   {
      cout<<x<<"i+"<<y<<"j+"<<z<<"k";
   }
   vector operator-(vector v)
   {
      vector temp;
      temp.x=x-v.x;
      temp.y=y-v.y;
      temp.z=z-v.z;
      return temp;
```

```
    }
};
   void main()
   {
      vector a(10,20,30),b(1,2,3),c;
      clrscr();
         cout<<"\nVector A:";
      a.show();
         cout<<"\nVector B:";
      b.show();
         c=a-b;
         cout<<"\n A-B:";
      c.show();
   }
```

Output

```
Vector A:10i+20j+30k
Vector B:1i+2j+3k
   A-B:9i+18j+27k
```

In Program 2.7, the `vector` is defined as a class with `x,y,z` as local variables. To solve the `vector` subtraction problem, the minus (–) operator cannot be applied on the objects of the `vector` class. By overloading the minus operator in the `vector operator-(vector v)` form, the subtraction problem of vector can be solved. The return type of the operator–here is a `vector` type.

2.3.3 Overloading Unary Operators

The general form of the prefix form of the unary ++ operator

```
return-type operator++()
```

Program 2.8

```
/*Program to overload++operator as prefix form*/
#include<iostream.h>
#include<conio.h>
class vector{
int x,y,z;
public:
vector(){}
vector(int a,int b,int c)
   {
      x=a;
      y=b;
      z=c;
   }
void show()
   {
```

```
      cout<<x<<"i+"<<y<<"j+"<<z<<"k";
   }
vector operator++()
   {
      ++x;
      ++y;
      ++z;
      return*this;
   }
};
void main()
   {
      vector a(10,20,30);
         clrscr();
         cout<<"\n         Vector A:";
      a.show();
         ++a;
         cout<<"\nAfter Increment A:";
      a.show();
   }
```

Output

```
Vector A:10i+20j+30k
After Increment A:11i+21j+31k
```

In Program 2.8, the operator ++ is overloaded to perform a preincrement on `vector` class objects. Operator overloading on the unary operators is similar to the plus and the minus operators. But the plus and the minus operators require arguments, while the prefix form the unary operators like ++ does not require the arguments. In the above example the ++ unary operator is called in the prefix form as ++a.

2.3.4 Postfix Form of Overloading the Unary Operator ++

The general form of the post form of the unary ++ operator is as follows:

```
return-type operator++(int x)
```

Program 2.9

```
/*Program to overload++operator as postfix form*/
#include<iostream.h>
#include<conio.h>
class vector{
int x,y,z;
public:
vector(){}
vector(int a,int b,int c)
   {
      x=a;
      y=b;
```

```
      z=c;
   }
void show()
   {
      cout<<x<<"i+"<<y<<"j+"<<z<<"k";
   }
vector operator++(int i)
   {
      i=1;
      x+=i;
      y+=i;
      z+=i;
      return*this;
   }
};
void main()
{
   vector a(10,20,30);
   clrscr();
   cout<<"\n         Vector A:";
   a.show();
   a++;
   cout<<"\nAfter Increment A:";
   a.show();
}
```

Output

```
Vector A:10i+20j+30k
After Increment A:11i+21j+31k
```

In Program 2.9, the operator ++ is overloaded to perform a post increment on the `vector` class objects. In the postfix form ++ operator, an integer argument is required, and the method is called when ++ operator follows the operand.

2.3.5 Prefix Form of Overloading the Unary Operator --

The general form of the prefix form of a unary -- operator is as follows:

```
return-type operator--()
```

In Program 2.10, operator -- is overloaded to perform a predecrement on the `vector` class object. The method operator--() is called when -- operator follows the operand.

Program 2.10

```
/*Program to overload--operator as prefix form*/
#include<iostream.h>
#include<conio.h>
class vector
```

```
{
   int x,y,z;
   public:
   vector(){}
   vector(int a,int b,int c)
   {
      x=a;
      y=b;
      z=c;
   }
void show()
   {
      cout<<x<<"i+"<<y<<"j+"<<z<<"k";
   }
vector operator--()
   {
      --x;
      --y;
      --z;
      return*this;
 } };
void main()
   {
      vector a(10,20,30);
         clrscr();
      cout<<"\n         Vector A:";
         a.show();
         --a;
      cout<<"\nAfter Decrement A:";
         a.show();
   }
```

Output

```
Vector A:10i+20j+30k
After Increment A:9i+19j+29k
```

2.3.6 Postfix Form of Overloading the Unary Operator --

The general form of the post form of the unary -- operator is as follows:

```
return-type operator--(int x)
```

Program 2.11

```
/*Program to overload--operator as postfix form*/
#include<iostream.h>
#include<conio.h>
class vector{
int x,y,z;
public:
```

```
vector(){}
vector(int a,int b,int c)
   {
      x=a;
      y=b;
      z=c;
   }
   void show()
   {
      cout<<x<<"i+"<<y<<"j+"<<z<<"k";
   }
   vector operator--(int i)
   {
      i=1;
      x-=i;
      y-=i;
      z-=i;
      return*this;
   }
};
   void main()
   {
      vector a(10,20,30);
         clrscr();
         cout<<"\n         Vector A:";
         a.show();
         a--;
         cout<<"\nAfter Decrement A:";
         a.show();
   }
```

Output

```
Vector A:10i+20j+30k
After Decrement A:9i+19j+29k
```

In Program 2.11, the operator -- is overloaded in the `vector operator--(int i)` to perform post decrement on the vector class object `a`.

Shorthand operators like `+=,-=,*=,/=` are also can be overloaded using the operator overloading.

Program 2.12

```
/*Program to overload+=operator*/
#include<iostream.h>
#include<conio.h>
class vector
{
   int x,y,z;
   public:
```

```
    vector(){}
    vector(int a,int b,int c)
    {
       x=a;
       y=b;
       z=c;
    }
    void show()
    {
       cout<<x<<"i+"<<y<<"j+"<<z<<"k";
    }
    vector operator+=(int i)
    {
       x+=i;
       y+=i;
       z+=i;
       return*this;
    }
};
void main()
{
   vector a(10,20,30);
   clrscr();
      cout<<"\n         Vector A:";
   a.show();
   a+=10;
      cout<<"\nAfter Increment A:";
   a.show();
}
```

Output

```
Vector A:10i+20j+30k
After Increment A:20i+30j+40k
```

In Program 2.12, the operator += is overloaded in the `vector operator+=(int i)` to increment the vector class object values by 10. The statement `a+=10` is the short form of `a=a+10`.

Program 2.13

```
/*Program to overload-=operator*/
#include<iostream.h>
#include<conio.h>
class vector
{
   int x,y,z;
   public:
   vector(){}
   vector(int a,int b,int c)
```

```
    {
       x=a;
       y=b;
       z=c;
    }
void show()
    {
       cout<<x<<"i+"<<y<<"j+"<<z<<"k";
    }
vector operator-=(int i)
    {
       x-=i;
       y-=i;
       z-=i;
       return*this;
    }
};
void main()
    {
       vector a(10,20,30);
       clrscr();
       cout<<"\n         Vector A:";
       a.show();
       a-=3;
          cout<<"\nAfter Decrement A:";
       a.show();
    }
```

Output

```
Vector A:10i+20j+30k
After Decrement A:7i+17j+27k
```

In Program 2.13, the operator `-=` is overloaded in the vector `operator-=(int i)` to decrement the vector class object values by 3. The statement `a-=3` is the short form of `a=a-3`.

2.4 INHERITANCE

Inheritance is a process of acquiring the properties of one object to the other object. The class that is inherited is called a base class, and the inheriting class is called the derived class. After inheriting the base class, members of the base class become members of the derived class. Inheritance allows the reuse of code.

The general form of inheritance is as follows:

```
Class derived-class:access-specifier base-class{
// members of the derived class
};
```

Derived class is the new class that is inherited from the base class and can also have its own members. Access specifier can be public, protected or private. The members of the base class will be acquired by the way

the base class is specified with the access specifier. If no access specifier is specified, then the default specifier will be private if the derived class is a `Class`. If the derived class is a structure, then the default specifier will be public.

2.4.1 Base Class Access Control

The derived class will have the members of the base class based on the declaration type such as private, public or protected and the access specifier at the time of inheriting the base class.

The members of the base class will be protected from the derived class based on the access specifier. All the private members of the base class will not be inherited by the derived class irrespective of the access specifier.

In case the derived class is a class, the following cases will be based on the access specifier:

Case 1: Access specifier is public:

a. All the public members of the base class will become the public members of the derived class.
b. All the protected members of the base class will become the protected members of the derived class.
c. All the private members of the base class will not be inherited to the derived class.

Case 2: Access specifier is protected:

a. All the public members of the base class will become the protected members of the derived class.
b. All the protected members of the base class will become the protected members of the derived class.
c. All the private members of the base class will not be inherited to the derived class.

Case 3: Access-Specifier is private

a. All the public members of the base class will become the private members of the derived class.
b. All the protected members of the base class will become the private members of the derived class.
c. All the private members of the base class will not be inherited to the derived class.

2.4.2 Types of Inheritance

There are two basic classifications of inheritance

- Single inheritance
- Multiple inheritance

Based on the above existing classifications, inheritance can be further divided into various types that are as follows:

1. Hierarchical inheritance
2. Single-level inheritance
3. Multi-level inheritance
4. Hybrid inheritance
5. Multipath inheritance

2.4.2.1 Single Inheritance

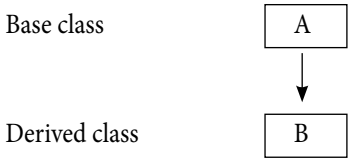
If a class is derived from a single base class, this type of inheritance is called single inheritance. The new class is called the derived class, and the old class is called the base class.

Example

Class B is derived from a single Class A; here Class A is the base class and Class B is the derived class.

The general form of single inheritance is as follows:

```
Class  <Derived-Class-Name> : <Access-Specifier>  <Base-Class-Name>
```



Program 2.14

```
#include<iostream.h>
#include<conio.h>
/*Single Inheritance*/
class A
{
   public:
   int a,b;
   void set(int x,int y)
   {
      a=x;
      b=y;
   }
};
   class B:public A{
   int c;
   public:
   B(int z)
   {
      c=z;
   }
void show()
   {
      cout<<"A ="<<a<<"\nB="<<b<<"\nC="<<c;
   }
};
void main()
   {
      B obj(30);
      obj.set(10,20);
      clrscr();
      obj.show();
   }
```

Output

```
A=10
B=20
C=30
```

In Program 2.14, Class B is inherited from Class A. Because Class B is inherited from Class A, Class B will have all the public members of Class A. `obj` is an object created for Class B. In the definition of Class B, the *show* method calls the members of Class A, which are not part of the Class B definition. Because a single derived Class B is derived from a single base Class A, the inheritance type is a single-level inheritance.

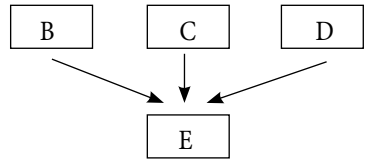
2.4.2.2 Multiple Inheritance

If a class is derived from multiple base classes, then the inheritance type is called multiple inheritance.

Example

Class E is derived from three different base classes B, C, D.

```
Class  <Derived-Class-NameE> : <Access-Specifier>  <Base-Class-NameB>,
                               <Access-Specifier>  <Base-Class-NameC>,
                               <Access-Specifier>  <Base-Class-NameD>
```



Program 2.15

```
#include<iostream.h>
#include<conio.h>
/*Multiple Inheritance*/
class A
{
   public:
   int a;
   void setA(int x)
   {
      a=x;
   }
void showA()
   {
      cout<<"\nA="<<a;
   }
};
class B
{
   public:
   int b;
   void setB(int x)
   {
      b=x;
   }
void showB()
   {  cout<<"\nB="<<b;
   }
```

```
};
class C: public A,public B{
public:
int c;
void setC(int x)
   {
      c=x;
   }
void showC()
   {
      cout<<"\nC="<<c;
   }
};
void main()
   {
      A objA;
      B objB;
      C objC;
      clrscr();

//Calling Class A methods
   cout<<"\nCalling Class A methods";
      objA.setA(10);
      objA.showA();

//Calling Class B methods
   cout<<"\nCalling Class B methods";
      objB.setB(20);
      objB.showB();

//Calling Class C methods
   cout<<"\nCalling Class C methods";
      objC.setA(10);
      objC.setB(20);
      objC.setC(30);
      objC.showA();
      objC.showB();
      objC.showC();
   }
```

Output

```
Calling Class A methods
   A=10
Calling Class B methods
   B=20
Calling Class C methods
   A=10
   B=20
   C=30
```

A class can be derived from two or more base classes. In Program 2.15, Class C is inherited from Class A and Class B. Therefore, Class C will have the members of Class A, Class B and the members of itself.

2.4.2.3 Hierarchical Inheritance

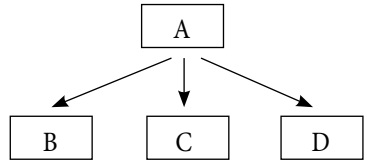
When two or more classes are derived from the same base class, the inheritance type is called hierarchical inheritance.

Example

Class B, Class C and Class D are derived from base Class A.

The general form of hierarchical inheritance is as follows:

```
Class<Derived-Class-NameB>:<Access-Specifier><Base-Class-NameA>
Class<Derived-Class-NameC>:<Access-Specifier><Base-Class-NameA>
Class<Derived-Class-NameD>:<Access-Specifier><Base-Class-NameA>
```



Program 2.16

```
#include<iostream.h>
#include<conio.h>
/*Hierarchical Inheritance*/
class A
{
public:
int a;
void setA(int x)
   {
      a=x;
   }
void showA()
   {
      cout<<"\nA="<<a;
   }
};
class B:public A
{
public:
int b;
void setB(int x)
   {
      b=x;
   }
void showB()
   {
```

```
      cout<<"\nB="<<b;}
};
class C:public A
{
   public:
   int c;
   void setC(int x)
   {
      c=x;
   }
void showC()
   {
      cout<<"\nC="<<c;
   }
};
void main()
{  A objA;
   B objB;
   C objC;
   clrscr();

//Calling Class A methods
cout<<"\nCalling Class A methods";
   objA.setA(10);
   objA.showA();

//Calling Class B methods
cout<<"\nCalling Class B methods";
   objB.setA(10);
   objB.setB(20);
   objB.showA();
   objB.showB();

//Calling Class C methods
cout<<"\nCalling Class C methods";
   objC.setA(10);
   objC.setC(30);
   objC.showA();
   objC.showC();
}
```

Output

```
Calling Class A methods
   A=10
Calling Class B methods
   A=10
   B=20
Calling Class C methods
   A=10
   C=30
```

Program 2.16 is an example for hierarchical inheritance. In this example, Class B and Class C are inherited from the same base Class A. So, Class B will have the members of Class B and the members of Class A and Class C will have the members of Class C and the members of Class A.

2.4.2.4 Single-level Inheritance

This inheritance is the same as the single inheritance, the only difference is that in single-level inheritance the derived class cannot be a base class.

2.4.2.5 Multi-level Inheritance

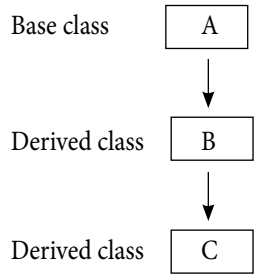
If a class is derived from a derived class, then the inheritance type is called multi-level inheritance.

Example

Class B is derived from Class A and Class C is derived from class B; now Class B is base class to C and derived class for Class A.

The general form of multi-level inheritance is as follows:

```
Class<Derived-Class-NameB>:<Access-Specifier><Base-Class-NameA>
Class<Derived-Class-NameC>:<Access-Specifier><Derived-Class-NameB>
```



Program 2.17

```
#include<iostream.h>
#include<conio.h>
/*Multi-Level Inheritance*/
class A{
public:
int a;
void setA(int x)
   {
      a=x;
   }
void showA()
   {
      cout<<"\nA="<<a;
   }
};
class B: public A{
public:
int b;
```

```
void setB(int x)
   {
      b=x;
   }
void showB()
   {
      cout<<"\nB="<<b;
   }
};
class C:public B{
public:
int c;
void setC(int x)
   {
      c=x;
   }
void showC()
   {
      cout<<"\nC"<<c;
   }
};
void main()
   {
      A objA;
      B objB;
      C objC;
      clrscr();

//Calling Class A methods
   cout<<"\nCalling Class A methods";
   objA.setA(10);
   objA.showA();

//Calling Class B methods
   cout<<"\nCalling Class B methods";
   objB.setA(10);
   objB.setB(20);
   objB.showA();
   objB.showB();

//Calling Class C methods
   cout<<"\nCalling Class C methods";
   objC.setB(20);
   objC.setC(30);
   objC.showB();
   objC.showC();
}
```

Output

```
Calling Class A methods
   A=10
Calling Class B methods
   A=10
   B=20
Calling Class C methods
   B=20
   C=30
```

This is an example for multi-level inheritance. In Program 2.17 Class C is derived from the derived Class B and Class B is derived from the base Class A. When an object is created for Class B, the object will have all the public members of Class A, and the object of Class C will have all the public members of Class B.

2.4.2.6 Hybrid Inheritance

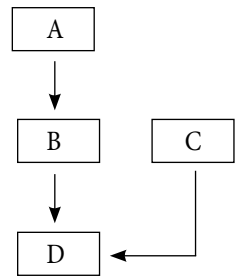
If a class is inherited with the combinations of one or more basic inheritance types, then it is called hybrid inheritance.

Example

Class B is derived from base Class A, and Class D is derived from derived Class B and base Class C.

The general form of multi-level inheritance is as follows:

```
Class<Derived-Class-NameB>:<Access-Specifier><Base-Class-NameA>,
Class<Derived-Class-NameD>:<Access-Specifier><Derived-Class-NameB>,
                           <Access-Specifier><Base-Class-NameC>,
```



Program 2.18

```
#include<iostream.h>
#include<conio.h>
/*Hybrid  Inheritance*/
class A
{
   public:
   int a;
   void setA(int x)
   {
      a=x;
   }
   void showA()
```

```
{
   cout<<"\nA="<<a;
}
};
class C
{
   public:
   int c;
   void setC(int x)
   {
      c=x;
   }
   void showC()
   {
      cout<<"\nC="<<c;
   }
};
class B: public A
{
   public:
   int b;
   void setB(int x)
   {
      b=x;
   }
void showB()
   {
      cout<<"\nB="<<b;
   }
};
class D: public B,public C
{
   public:
   int b;
   void setB(int x)
   {
      b=x;
   }
void showB()
   {
      cout<<"\nB="<<b;
   }
};
void main()
{
   A objA;
   B objB;
   C objC;
   D objD;
```

```
   clrscr();
//Calling Class A methods
   cout<<"\nCalling Class A methods";
   objA.setA(10);
   objA.showA();
//Calling Class B methods
   cout<<"\nCalling Class B methods";
   objB.setA(10);
   objB.setB(20);
   objB.showA();
   objB.showB();
//Calling Class C methods
   cout<<"\nCalling Class C methods";
   objC.setC(30);
   objC.showC();
   cout<<"\nCalling Class D methods";
   objD.setB(10);
   objD.setC(20);
   objD.showB();
   objD.showC();
}
```

Output

```
Calling Class A methods
   A=10
Calling Class B methods
   A=10
   B=20
Calling Class C methods
   C=30
Calling Class D methods
   B=10
   C=20
```

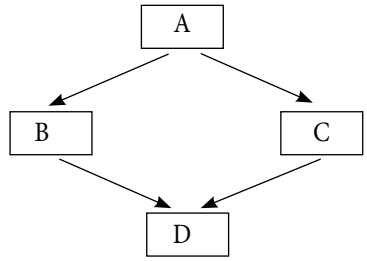
In Program 2.18, hybrid inheritance is a combination of multiple and multi-level inheritances. Class A and Class C are base classes. Class B is derived from Class A, and Class D is derived from Class B and Class C.

2.4.2.7 Multipath Inheritance

If a class is derived from one or more derived classes that are derived from the same base class, such type of inheritance is called multipath inheritance.

Example

Class B and Class C are derived from the same base Class A, and Class D is derived from Class B and Class C.




```
class A
{
   public:
   int a;
};
class B:public A
{
   public:int b;
};
Class C:public A
{
   public:int c;
};
Class D:public B,public C
{
    public:int d;
};
```

In the above program, Classes B and C are derived classes from the same base Class A. Classes B and C can access variable a of Class A. Class D is derived from Class B and Class C which are derived from the same base Class A. When Class D accesses variables of Class A, the compiler gives an error message because the compiler is in an ambiguous state in selecting variable a, whether to select it from Class B or Class C since both of them have the copy of a. This ambiguity can be resolved by using *virtual* keyword.

Access to base class: Because the base class members are inherited by the derived class, the base class members should have controlled access to the derived class. When the access specifier of the base class is public, then all the public members of the base class are accessible by the derived class. All the private members of the base class cannot be accessed by the derived class.

Access specifier of the base class is public: When the access specifier of the base class is public, all the public members of the base class are accessible to the derived class.

Program 2.19

```
/*Program to demonstrate when the access specifier of the base class
   is public*/
#include<iostream.h>
#include<conio.h>
```

```
      /*Base Class Access Control in inheritance when Access Specifier is
      PUBLIC*/
class Base
{
   int a;                //Private Member Data Variable, Not Accessible by
                         //the derived Class
   void setA(int x)      //Private Member Function, Not Accessible by
                         //the derived Class
   {
      a=x;
   }
   void showA()          //Private Member Function,Not Accessible by
                         //the derived Class
   {
      cout<<"\nA:"<<a;
   }
   public:
   int b;
   void setB(int y)
   {
      b=y;
   }
   void showB()
   {
      cout<<"\nB="<<b;
   }
};
class Derived:public Base{
int c;
public:
void setC(int z)
   {
      c=z;
   }
void showC()
   {
      cout<<"\nC="<<c;
   }
};
void main()
{
   Base obj1;
   Derived obj2;
   clrscr();
      //Calling Base Class  methods
         cout<<"\nCalling Base Class methods";
      //obj1.a=1;             /*Not Accessible Because of private member of
                                Base Class*/
```

```
    //obj1.showA();          /*Not Accessible Because of private member of
                               Base Class*/
    //obj1.setA(10);         /*Not Accessible Because of private member of
                               Base Class*/
    //obj1.showA();          /*Not Accessible Because of private member of
                               Base Class*/
    //Calling Class B methods
       cout<<"\nCalling Class B methods";
    //obj2.setA(10);         /*Not Accessible Because of private member of
                               Base Class*/
    obj2.setB(20);
    //obj2.showA();          /*Not Accessible Because of private member of
                               Base Class*/
    obj2.showB();
    obj2.setC(30);
    obj2.showC();
  }
```

Output

```
Calling Base Class methods
Calling Class B methods
B=20
C=30
```

In Program 2.19, the private members of the base class are not accessible by the derived class. The base Class's `setA()`, `showA()` are not accessible by the derived class because they are private members. The public members of the base Class `setB()`, `showB()` are accessible by the derived class. It shows that all the public members of the base class become public members of the derived class. All the private members of the base class cannot be accessible by the derived class.

Access specifier of the base class is private: When the access specifier of the base class is private, then all the public and protected members of the base class become private members of the derived class.

Program 2.20

```
#include<iostream.h>
#include<conio.h>
class Base
{
   int a;
   public:
   void setA(int x)
   {
      a=x;
   }
void showA()
   {
      cout<<"\nA="<<a;
   }
```

```
};
class Derived:private Base
{
   int b;
   public:
   void setB(int y)
   {
      b=y;
   }
void showB()
   {
      cout<<"\nB="<<b;
   }
};
void main()
   {
      Derived obj;
      clrscr();
      cout<<"\nDerived Class Members";
      //obj.setA(10);                  /*Not Accessible*/
      //obj.showA();                   /*Not Accessible*/
      obj.setB(20);
      obj.showB();
   }
```

Output

```
Derived Class Members
B=20
```

In Program 2.20, the private members of the base class are not accessible by the derived class. The base class's `setA()`, `showA()` are not accessible by the derived class because they are public members and the access specifier of the base class is private. It shows that all the public members of the base class become private members of the derived class when the access specifier of the base class is private. All the private members of the base class cannot be accessible by the derived class.

Protected members become protected members of the derived class when the access specifier of the base class is public.

Program 2.21

```
#include<iostream.h>
#include<conio.h>
class Base
{
   protected:
   int a;
   public:
void setA(int x)
```

```
   {
      a=x;
   }
void showA()
   {
      cout<<"\nA="<<a;
   }
};

class Derived:public Base
   {
      int b;
      public:
      void setB(int y)
      {
         b=y;
      }
void showAB()
   {
      cout<<"\nA="<<a<<"B="<<b;
   }
};

void main()
   {
      Derived obj;
      clrscr();
      cout<<"\nDerived Class Members";
      obj.setA(10); /*Accessible*/
      obj.showA(); /*Accessible*/
      obj.setB(20);
      obj.showAB();/*The method showAB can access protected variable 'a'*/
   }
```

Output

```
Derived Class Members
A=10
A=10 B=20
```

In Program 2.21, the derived class `showAB()` method calls the protected member of the base class because the access specifier of the derived class is public. It shows that the member functions of the derived class can access the protected members of the base class when the access specifier of the base class is public.

Access specifier of the base class is protected: All the public and protected members of the base class become protected members of the derived class when the access specifier of the base class is protected. All the private members of the base class are not accessible by the derived class.

Program 2.22

```
#include<iostream.h>
#include<conio.h>
class Base
{
   protected:
   int a;
   public:
void setA(int x)
   {
      a=x;
   }
void showA()
   {
      cout<<"\nA=”<<a;
   }
};
class Derived:protected Base
{
   int b;
   public:
   void setB(int y)
   {
      setA(10);
      b=y+a;
   }
void showAB()
   {
      cout<<"\nA=”<<a<<“B=”<<b;
   }
};
void main()
   {
      Derived obj;
      clrscr();
      cout<<"\nDerived Class Members”;
      // obj.setA(10);           /*Illegal because of setA is protected member
                                    of derived class*/
      obj.setB(20);
      obj.showAB();
   }
```

Output

```
Derived Class Members
A=10 B=30
```

In Program 2.22, the derived class member function `showAB()` calls the protected member `a`, because the protected members of the base class become protected members of the derived class when the access specifier is protected.

The execution order of the constructor and destructor when inheritance is applied: Inheritance allows the base and derived classes to have constructors and destructors. The compiler knows the order how the classes are derived; in the same order constructors are called. The destructors are called exactly opposite to that of the constructor.

Program 2.23

```
#include<iostream.h>
class Base
{
   public:
   Base()
   {
      cout<<"\nCalling Base Class Constructor";
   }
      ~Base()
   {
      cout<<"\nCalling Base Class Destructor";
   }
};
class Derived:public Base
   {
      public:
      Derived()
      {
         cout<<"\nCalling Derived Class Constructor";
      }
      ~Derived()
      {
         cout<<"\nCalling Derived Class Destructor";
      }
   };
void main()
   {
      Derived obj;
   }
```

Output

```
Calling Base Class Constructor
Calling Derived Class Constructor
Calling Derived Class Destructor
Calling Base Class Destructor
```

The main function of Program 2.23 is creating an object to the derived class. Because the derived class is derived from the base class, the compiler calls the base class first and then the derived class constructor is called. Exactly reverse to the order of the constructor the destructors are called.

Program 2.24

```
#include<iostream.h>
#include<conio.h>
class Base
{
   public:
   Base()
   {
      cout<<"\nCalling Base Class Constructor";
   }
      ~Base()
   {
      cout<<"\nCalling Base Class Destructor";
   }
};
class DerivedB:public Base{
public:
DerivedB()
   {
      cout<<"\nCalling DerivedB Class Constructor";
   }
~DerivedB()
   {
      cout<<"\nCalling DerivedB Class Destructor";
   }
};
class DerivedC:public DerivedB{
public:
DerivedC()
   {
      cout<<"\nCalling DerivedC Class Constructor";
   }
~DerivedC()
   {
      cout<<"\nCalling DerivedC Class Destructor";
   }
};
void main()
{
   clrscr();
   DerivedC obj;
}
```

Output

```
Calling Base Class Constructor
Calling DerivedB Class Constructor
Calling DerivedC Class Constructor
```

```
Calling DerivedC Class Destructor
Calling DerivedB Class Destructor
Calling Base Class Destructor
```

In case of multi-level inheritance, three or more constructors are to be called. In Program 2.24, Class C is derived from derived Class B. In this case, the constructors of base Class A, Class B and Class C are called, respectively. In the reverse order the destructors are called.

Arguments passing to the base class constructor: Inheritance allows the classes to have constructors, the constructors can be with and without arguments. The problem here is that an object can be created to a derived class. If that is the case "*how to pass the arguments to the base class constructor while creating an object to the derived class*?". To solve this problem, C++ allows the developers to pass the arguments to the base class constructor by using colon (:) operator in the declaration of the derived class, and the arguments can be passed in the order of derived class object arguments, base class argument list.

Suppose the derived class constructors have one integer argument and two float arguments to the base class constructor. Then the developer can pass the derived class argument followed with the base class arguments.

The general form of declaration of the derived class constructors is as follows:

```
Derived-class-constructor( args) : base-class1-constructor (args) ,
   base-class2-constructor(args)
    . . . . . .
   base-classN-constructor(args)
   {
      // definition of the derived class
   }
```

The general form of passing the arguments to the derived class constructors is as follows:

```
Derived-class object-name(list of derived-class-args,list-of-base-class-args);
```

Program to demonstrate the arguments passing to the base class constructors:

Program 2.25

```
#include<iostream.h>
#include<conio.h>
class Base{
public:
Base(int x)
{
   cout<<"\nCalling Base Class Constructor";
   cout<<"\nArgument Value Received by the Base Class:"<<x;
}
~Base()
   {
      cout<<"\nCalling Base Class Destructor";
   }
};
```

```
class Derived:public Base
{
   public:
   /*Derived Class uses only p and the variable q will be passed to the base
     class contructor*/
   Derived(int p,int q):Base(q)
   {
      cout<<"\nCalling Derived Class Constructor";
      cout<<"\nArgument Value Received by the Derived Class:"<<p;
   }
   ~Derived()
   {
      cout<<"\nCalling Derived Class Destructor";
   }
};
void main()
{
   clrscr();
   Derived obj(10,20);
}
```

Output

```
Calling Base Class Constructor
Argument Value Received by the Base Class:20
Calling Derived Class Constructor
Argument Value Received by the Derived Class:10
Calling Derived Class Destructor
Calling Base Class Destructor
```

In Program 2.25, the derived Class B has only one integer input argument and the constructor of the base class also has only one integer input argument. To allow the arguments to be passed on to the base class constructor, the derived class is defined as `Derived(int p,int q):Base(q)`. Initially the derived class object accepts input values to the variables `p` and `q`. Only `p` is required by the derived class and `q` is passed on to the base class constructor. At the time of creating the object to the derived class the derived class input argument value 10 is placed and the input argument value 20 is passed on to the base class constructor.

Passing the input arguments to the base class constructor when multi-level inheritance is used:

Program 2.26

```
#include<iostream.h>
#include<conio.h>
class Base
{
   public:
   Base(int x)
   {
      cout<<"\nCalling Base Class Constructor";
```

```
      cout<<"\nArgument Value Received by the Base Class:"<<x;
   }
~Base()
   {
      cout<<"\nCalling Base Class Destructor";
   }
};
class DerivedA : public Base{
public:
                /*DerivedA Class uses only p and the variable q will be
                  passed to the base class contructor*/
DerivedA(int p,int q):Base(q)
   {
      cout<<"\nCalling DerivedA Class Constructor";
      cout<<"\nArgument Value Received by the DerivedA Class:"<<p;
   }
~DerivedA()
   {
      cout<<"\nCalling DerivedA Class Destructor";
   }
};
   class DerivedB:public DerivedA{
   public:
           /* DerivedB Class uses only p and the variables q,r will be passed
              to the DerivedA class contructor*/
DerivedB(int p,int q,int r): DerivedA(q,r)
   {
      cout<<"\nCalling DerivedB Class Constructor";
      cout<<"\nArgument Value Received by the DerivedB Class:"<<p;
   }
~DerivedB()
   {
      cout<<"\nCalling DerivedB Class Destructor";
   }
};
void main()
{
   clrscr();
   DerivedB obj(10,20,30);
}
```

Output

```
Calling Base Class Constructor
Argument Value Received by the Base Class:30
Calling DerivedA Class Constructor
Argument Value Received by the DerivedA Class:20
Calling DerivedB Class Constructor
Argument Value Received by the DerivedB Class:10
```

```
Calling DerivedB Class Destructor
Calling DerivedA Class Destructor
Calling Base Class Destructor
```

In Program 2.26, Class B is derived from the derived Class A, and Class A is derived from the base class. All the constructors of all the classes accept one integer argument. The base class constructor is defined as `Base(int x)`, so that it accepts only one input argument. Class A is derived from the base class. To pass the arguments to the base class, the Class A constructor is defined as `DerivedA(int p,int q):Base(q)`. The constructor of Class `DerivedA` accepts `p` and `q` as input arguments at the time of object creation and uses only `p` for its purpose and passes `q` to the base class constructor. The Class `DerivedB` is also inherited from the Class `DerivedA`. To pass the arguments to the constructor of ClassA, the constructor Class `DerivedB` should accept the arguments at its constructor declaration. Arguments to all the constructors are passed as "`DerivedB obj(10,20,30);`". The value 10 is used by the constructor of Class `DerivedB` and the argument values 20, 30 are passed to the constructor of Class DerivedA. The constructor of Class `DerivedA` accepts 20, 30 as input arguments. The constructor of Class `DerivedA` only uses the value 20, and the value 30 will be passed on to the base class constructor. In this way the arguments can be passed on to the base class constructors, while creating the objects for the derived class.

Making the inherited private members as its original access specification in the inheritance: There may be a situation where some of the inherited private members have to be with its original access specification. In this case C++ allows the developers to restore its original access specification in the derived class.

When the access specifier of the base class is *private*, then all the public and protected members of the base class become private members of the derived class. The said mechanism allows one to restore some of the inherited public and protected members, which become private in the derived class and can restore their original access specification as public or protected.

The general form of access specification is as follows:

```
Base-class::member; //by this statement the original access specification of
the member will be restored.
```

Program 2.27

```
#include<iostream.h>
#include<conio.h>
class Base
{
   public:
   int a;
   Base()
   {
      a=10;
   }
};
class DerivedA:private Base
{
   public:
   Base::a;          //Making the private variable
                     //to again public access specification
```

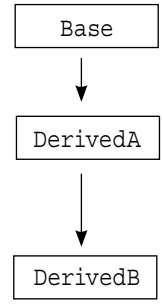
```
};
class DerivedB:private DerivedA
{
   public:
   void showa()
   {
      cout<<"\nA="<<a;
   }
};
void main()
{
   clrscr();
   DerivedB obj;
   obj.showa();
}
```

Output

```
A=10
```

In Program 2.27, the Class `DerivedA` is inherited from the base class, and the base class access specifier is private. So, all the public and protected members of the base class become private. The class `DerivedB` is inherited from the Class `DerivedA`. So, the inherited members of the `DerivedA` class will not be present in the class `DerivedB`. In the base class the variable `a` is declared as public, it became private in the class `DerivedA`. The statement `Base::a;` makes the private variable of class `DerivedA` to be restored as public, and the variable can be further inherited to the other derived class.

```
                        public:int a;
            Class DerivedA:private Base
                        public:int a;
                        become
                        private:int a;
After the statement Base::a;
the access specifier became public:int a;
                        public:int a;
                        become
                        private:int a;
```



2.4.3 Reasons for the Usage of Inheritance

Specialization: A new class is derived from the base class and this class may just add something more to the base class or it may slightly differ from the base class.

Generalization: When two or more classes have similar properties, then this similarity can be combined together into a common base class.

Interface: Used by the base class to define interface, but doesn't implement some or all parts of it.

The derived class Is –A type of the base class. This means that anywhere the base class works, the derived class can also be used. There may be a confusion between IS-KIND-OF-LIKE and IS-A when derived class is a kind of already existing class.

2.4.4 Advantages

- In hierarchical relationships inheritance makes the development model clones to real-life object model.
- Reusability: The public members of base class can be reused by derived class without rewriting them.
- Data hiding: The base class can control the access of some data so that they cannot be modified by the derived class.
- Extensibility: Extending the base class properties by the derived class to generate more dominate object.

2.4.5 Disadvantages

- Objects, when used for invoking member functions create more compiler overheads.
- Memory allocated to unused data elements will not be utilized, which wants the memory.
- Inheritance if not used correctly leads a program to be more complicated.
- As Vtables grow the performance of the program decreases.

2.4.6 Delegation

In object-oriented programming, two classes can be joined either by using inheritance or delegation. Delegation is a mechanism in which the object of one class is used as a member of another class. The kind of relationship in inheritance is IS KIND OF whereas in delegation HAS-A relationship.

```
class x
   {};
class y
   {};
class z
{
  x a; //object of class x
  y b; //object of class y
}
```

Here class `z` has two data members `a, b` which are objects of other classes `x` and `y`; such a kind of relationship is known as HAS-A relationship.

2.5 POLYMORPHISM

Polymorphism can be defined as "many forms". Poly means "many", morph means "form" and "ism" is a theory or a process. In general, polymorphism means "one can have many forms". Operator overloading and function overloading are the types of polymorphism.

C++ supports two types of binding—static and dynamic. Choosing a function call at the compile time is called static binding. Choosing a function call at the run-time is called dynamic binding. Static binding is implemented by using function and operator overloading, and dynamic binding is implemented using virtual functions.

2.5.1 Virtual Functions

Virtual keyword is used to avoid ambiguous situations in the multipath inheritance. In the multipath inheritance, an object can inherit the properties of the derived objects, which are inherited from the same base class objects. This kind of situations leads to ambiguity. To avoid ambiguity while calling the inherited members, virtual functions are introduced. Using the keyword *virtual* ambiguous situation can be resolved. Ambiguity is a compiler error.

Program with ambiguity:

```
#include<iostream.h>
#include<conio.h>
class Base
   {
     public:
        int base_var;
   };
class derived1:public Base
   {
     public:
        int derived1_var;
   };
class derived2:public Base
   {
     public:
        int derived2_var;
   };
class derived3:public derived1, public derived2
   {
     public:
        int derived3_var;
   };
void main()
   {
     derived3 obj;
     obj.base_var=20;      /*derived1 and derived2 instances are inherited
                              to the obj, compiler cannot address which
                              member has to be called*/
}
```

In the above program when the object `obj` is created, the object `obj` will inherit the public properties of `derived1` and `derived2` classes. Because both the `derived1` and `derived2` classes are inherited from the base class, both the instances of the `derived1` and `derived2` classes will have a `base _ var` member. Because `derived3` is inherited from `derived1` and `derived2` classes, two copies of the `base _ var` member will be present in the `obj` object. The compiler will be in ambiguous state to decide which member has to be addressed when `obj.base _ var` is called.

If the access specifier of the base class is defined as virtual, then compiler avoids multiple instance copies of the same base class and keeps only one base class instance in the multipath inheritance. The below example avoids ambiguity by using the virtual keyword:

```
#include<iostream.h>
#include<conio.h>
class Base
{
   public:
   int base_var;
};
class derived1:virtual public Base
{
   public:
      int derived1_var;
};
class derived2:virtual public Base
{
   public:
      int derived2_var;
};
class derived3:public derived1, public derived2
{
   public:
      int derived3_var;
};
void main()
{
   derived3 obj;
   obj.base_var=20;         /*derived1 and derived2 instances are inherited
                              to the obj, compiler copies only one instance
                              of base class's base_var member*/
}
```

Virtual function is a function that is defined in the base class and can be redefined by the derived classes. Virtual function allows one to implement the concept of polymorphism. Because virtual functions are defined in one class and may be redefined by the derived classes, the virtual function concept facilitates one interface and can have more number of forms.

Program 2.28

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class base
{
    public:
    virtual void date(int dd,int mm,int yy)
    {
           cout<<"\nBase Class date function:"<<dd<<"\\"<<mm<<"\\"<<yy;
    }
};
```

```
class derived:public base
{
public:
char month[5];
void date(int dd,int mm,int yy)
{
     switch(mm)
     {
     case 1:strcpy(month,"JAN");
            break;
     case 2:strcpy(month,"FEB");
            break;
     case 3:strcpy(month,"MAR");
            break;
     case 4:strcpy(month,"APR");
            break;
     case 5:strcpy(month,"MAY");
            break;
     case 6:strcpy(month,"JUN");
            break;
     case 7:strcpy(month,"JULY");
            break;
     case 8:strcpy(month,"AUG");
            break;
     case 9:strcpy(month,"SEP");
            break;
     case 10:strcpy(month,"OCT");
            break;
     case 11:strcpy(month,"NOV");
            break;
     case 12:strcpy(month,"DEC");
            break;
     }
            cout<<"\nDerived Class date function:"<<dd<<"\\"<<month<<"\\"<<yy;
     }
};
void main()
{
     clrscr();
     base obj1;
     obj1.date(1,1,2010);
     derived obj2;
     obj2.date(1,1,2010);
}
```

Output

```
Base Class date function:1\1\2010
Derived Class date function:1\JAN\2010
```

In Program 2.28, the `date()` method is declared as a virtual function and is defined in the base class to display the date in numeric format and date\month\year in the integer format. For derived class convenience the virtual function `date()` redefines the day as integer, month in three characters and year as integer.

Runtime polymorphic nature of the virtual function through base class reference: If any function is declared as virtual in the base class, that function can be redefined by the derived classes. When a base class reference pointer is created, then that pointer can be used for the derived classes.

/*Virtual Functions*/

Program 2.29

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class base
{
   public:
   virtual void date(int dd,int mm,int yy)
   {
      cout<<"\nBase Class date function:"<<dd<<"\\"<<mm<<"\\"<<yy;
   }
};
class derived: public base{
public:
char month[5];
void date(int dd,int mm,int yy)
{
   switch(mm)
   {
   case 1: strcpy(month,"JAN");
           break;
   case 2: strcpy(month,"FEB");
           break;
   case 3: strcpy(month,"MAR");
           break;
   case 4: strcpy(month,"APR");
           break;
   case 5: strcpy(month,"MAY");
           break;
   case 6: strcpy(month,"JUN");
           break;
   case 7: strcpy(month,"JULY");
           break;
```

```
    case 8: strcpy(month,"AUG");
            break;
    case 9: strcpy(month,"SEP");
            break;
    case 10:strcpy(month,"OCT");
            break;
    case 11:strcpy(month,"NOV");
            break;
    case 12:strcpy(month,"DEC");
            break;
       };
    cout<<"\nDerived Class date function:"<<dd<<"\\"<<month<<"\\"<<yy;
    }
};
void main()
{
    clrscr();
    base*p,obj1;
    p=&obj1;
    p->date(1,1,2010);
    derived obj2;
    p=&obj2;
    p->date(1,1,2010);
}
```

Output

```
Base Class date function:1\1\2010
Derived Class date function:1\JAN\2010
```

In Program 2.29, the pointer `p` is created as a base class pointer by the definition `base *p`. This is because the pointer `p` is declared as a base class reference pointer and should not be used for the other class objects. But the virtual function concept allows the developer to implement the runtime polymorphism. So the base class pointer can be used to refer to the derived class objects.

By the statements `p=&obj1;` and `p->date(1,1,2010);` the pointer `p` is used to refer the base class members. With the statements `p=&obj2;` and `p->date(1,1,2010);` the base class pointer `p` is used for the derived class. The output clearly says that the statement `p->date(1,1,2010);` is executed when the derived class date function is executed. The result becomes `Derived Class date function:1\JAN\2010`.

Base class virtual functions are called when the derived class fails to overwrite the base class's virtual functions: When the virtual functions are defined in the base class, the virtual function may or may not be overwritten by the derived classes. If the virtual function is overwritten, then the derived class function will be called. If the virtual function is not overwritten or redefined by the derived class, then the base class virtual function is called.

Program 2.30

```
/*Virtual Functions are Hierarchical*/
#include<iostream.h>
```

```
#include<string.h>
#include<conio.h>
class base
{
   public:
   virtual void date(int dd,int mm,int yy)
   {
      cout<<"\nBase Class date function:"<<dd<<"\\"<<mm<<"\\"<<yy;
   }
};
class derived:public base
{
   public:
   void show()
   {
      cout<<"\n I am in the derived class";
   }
};
void main()
{
   clrscr();
   base obj1;
   obj1.date(1,1,2010);
   derived obj2;
   obj2.date(1,1,2010);
   obj2.show();
}
```

Output

```
Base Class date function:1\1\2010
Base Class date function:1\1\2010
I am in the derived class
```

In Program 2.30, the virtual function is not redefined by the derived class. When `obj2.date(1,1,2010)` is called the base class definition of the date function will be called because the date function is not redefined by the derived class.

2.5.2 Pure Virtual Functions

A pure virtual function is a virtual function declared in the base class, but there is no definition for the virtual function, so that it can be overridden by the derived class. The pure virtual function must be defined by the derived class. When a function is declared as pure virtual function in a class, then that class cannot be used to declare any object because it becomes an abstract class.

A pure virtual function can be declared as

```
virtual return-type function-name(args)=0;
```

Example

```
virtual void greet()=0;
```

Here greet is a pure virtual function and the assignment operation is not used to assign zero, but it intimates the compiler that it is a pure virtual function and it does not have definition.

Program 2.31

```
/*Program to demonstrate Pure Virtual Function*/
#include<iostream.h>
#include<conio.h>

class Base
{
   public:
   virtual void greet()=0;  /*Pure virtual function, memory will not
                              be created for this method, But the derived
                              classes can work with runtime polymorphism*/
};
class Derived:public Base
{
   public:
   void greet()
   {
      cout<<"\nHi! i am Derived Class";
   }
};
void main()
   {
      Derived obj;
      clrscr();
      obj.greet();
   }
```

Output

```
Hi! i am Derived Class
```

In the base class in Program 2.31, the virtual method `greet()` is declared as a pure virtual function, but there is no definition for the method in the base class. When the derived class defines the pure virtual function, then the version of derived class will be called.

2.6 ABSTRACT CLASSES

An abstract class is a class where there exists at least one pure virtual function defined in the class. Creation of objects for the abstract class is not possible. Abstract classes can have pointers. The pure virtual functions of the abstract classes can be referred by the pointers. The implementation of runtime polymorphism is possible by the abstract classes. A class is made as abstract by making one or more of its functions as pure.

Program 2.32

```
/*Program to Demonstrate ABSTRACT Class*/
#include<iostream.h>
#include<conio.h>
class Base
{
   public:
   virtual void greet()=0;
   virtual void showBye()=0;
};
class Derived:public Base
{
   public:
   void greet()
   {
      cout<<"\nHi! i am Derived Class";
   }
void showBye()
   {
      cout<<"\nBye, Have a good day";
   }
};
void main()
{
   Derived obj;
   clrscr();
   obj.greet();
   obj.showBye();
}
```

Output

```
Hi! i am Derived Class
Bye, Have a good day
```

In the example in Program2.32, the methods `greet()` and `showBye()` are defined as pure virtual functions. So, the base class is said to be an abstract class. The derived class derived the virtual functions of the abstract class and is defined in the derived class. When the derived class object calls the methods `greet()` and `showBye()`, then the derived class definitions will be called by the methods.

2.7 GENERIC PROGRAMMING WITH TEMPLATES

Template is one of the most useful software reuse feature of C++. Template defines a range of related functions called the function template and a range of related classes called the class template. This technique is called generic programming. A function that works for all functions is called a generic function. The data type will be passed as an argument to the generic function. By creating a template for the function `sum()`, it can be used to perform addition on any data type.

When there is a need to perform operations on different data types, normally function overloading is used, which increases the program size and also requires one to create more local variables, which also increase the memory requirement. Template overcomes the limitations of function overloading. The mechanism provided by the template can also be implemented on classes.

A generic function can be used for different data types. A generic function or a class can be defined by using *template* keyword.

2.7.1 Function Templates

A function can be defined as a template if its operations are similar on all the data types instead of overloading the function.

The general form of a template function is as follows:

```
template<class Template-type>return-type function-name(args)
{
   //definition of the template function.
}
```

Here `template` is a keyword to define the template function and `template-type` is a data type. `return-type` is the data type of the return value of the template function. The `function-name` is the name of the template function and `args` is the list of input arguments to the template function. The template type can be used as data type in the body of the generic function or a class.

Example

```
template<class T>void echo_value()
{
   T value;
   Cout<<"Enter a value:";
   Cin>>value;
   Cout<<"\nYou have entered:"<<value;
}
```

In the example, `T` is the data type of the template function `echo _ value()`. This function can be applied on any input argument type.

A template class or a function can have any number of template types.

Program 2.33

```
/*A template function with single generic data type*/
#include<iostream.h>
#include<conio.h>
template<class X> void add(X a,X b)
{
   X sum;
   sum=a+b;
   cout<<"\n"<<a<<"+"<<b<<"="<<sum;
}
void main()
{
```

```
    clrscr();
    add(10,20);
    add(0.1,0.2);
}
```

Output

```
10+20=30
0.1+0.2=0.3
```

In Program 2.33, the addition of two numbers is considered. A function written with integer arguments will operate for the other data types. It is also difficult to write a separate add function for all the data types. With the function overloading process, the `add` function can be defined for any of the data types, but the logic of the `add` function is the same for all the methods, which are defined for different data types. Generic function allows the developers to write a function, which has common logic for different data types. Function overloading is best suited for the functions with different logics.

The above generic function add is defined with a template class `X`. Here `X` can be any data type in the runtime. In the main function `add(10,20)` is invoked because `(int,int)` are passed to the generic function, the function works with the data type integer, i.e. `X` becomes an integer, and in the second statement `add(0.1,0.2)` is called, because the `add` function is called as `add(float,float)`, at the time of execution `X` becomes float and the add function will work with float data types. Instead of writing two different functions for solving addition of integers and addition of float values, a single generic function can be used for any of the data types. The statement is generalized, a single generic function can be used for the different data types where the logic is the same for all the data types.

A template function can have more than one template type. The template types can be used as normal data types in the body of the generic function or a class.

Program 2.34

```
/*A template function with more than one generic data type*/
#include<iostream.h>
#include<conio.h>
template<class X,class Y>void generic_fun(X a,Y b)
{
cout<<"\nArgument values Recieved by the function:"<<a<<" "<<b;
}
void main()
{
   clrscr();
   generic_fun(10,20.2);
   generic_fun(0.1,100);
   generic_fun(10,"Hello World");
   generic_fun(10.2,'H');
}
```

Output

```
Argument values Recieved by the function:10 20.2
Argument values Recieved by the function:0.1 100
```

```
Argument values Recieved by the function:10 Hello World
Argument values Recieved by the function:10.2 H
```

In Program 2.34, X and Y are the two different template types. These template types are used to define a and b input arguments of generic _ fun. When generic _ fun(10,20.2) is called X becomes integer data type, and the Y becomes float data type. When a,b are called by the generic _ fun the values of a and b will be echoed on to the screen.

2.7.1.1 Generic Functions can be Overloaded

Overloading a template function with a normal function: As methods are overloaded generic function can also be overloaded, whenever new logic is required to perform the same functionality for any of the data type.

Program 2.35

```
/*Explicitly Overloading Generic Functions*/
#include<iostream.h>
#include<conio.h>
template<class X,class Y>void generic_fun(X a,Y b)
{
   cout<<"\nA="<<a<<"B="<< b;
}
void generic_fun(int a,int b)
{
   int sum=a+b;
   cout<<"\nSum="<<sum;
}
void main()
{
   clrscr();
   generic_fun(10,20); /*int generic_fun will be called*/
   generic_fun(10,20.2);/*generic generic_fun will be called*/
   generic_fun("Hello"," World");
}
```

Output

```
Sum=30
A=10 B=20.2
A=Hello B=World
```

In Program 2.35, the template generic _ fun will be called for all the data types except when generic _ fun is called with two integer arguments. Here the template function generic _ fun is overloaded with generic _ fun(int,int) function. Whenever generic _ fun(int,int) is called void generic _ fun(int a,int b) will be called. For all the other data types, the template function generic _ fun will be called.

Overloading a template function with another template function: As template functions are overloaded with the normal functions, a generic function can be overloaded with another template function. Depending upon the number of arguments, a generic function will be called. In Program 2.36, generic _ fun(X a) template function is overloaded with another template function generic _ fun(X a,Y b). When generic _ fun(10); is called,

the template function `generic _ fun(X a)` will be called. When `generic _ fun(10,20.2)` is called, the second overloaded template function `generic _ fun(X a, Y b)` will be called.

Program 2.36

```
/*Explicitly Overloading Generic Functions*/
#include<iostream.h>
#include<conio.h>
template<class X>void generic_fun(X a)
{
   cout<<"\nA="<<a;
}
template<class X,class Y>void generic_fun(X a,Y b)
{
   cout<<"\nA="<<a<<"B="<<b;
}
void main()
{
   clrscr();
   generic_fun(10); /*first template generic_fun will be called*/
   generic_fun(10,20.2);/*Second template generic_fun will be called*/
   generic_fun("Hello","World");/*Second template generic_fun will be called*/
}
```

Output

```
A=10
A=10 B=20.2
A=Hello B=World
```

2.7.1.2 Standard Parameters can be Used with the Template Function

A template function can be defined with the constants and predefined standard parameters. In Program 2.37, height is a constant parameter, the template function `volume _ of _ cube(X l, X b)` can use the standard parameter. The program is written with height as constant value 8, and the volume of a cube whose height is 8 can be computed with the generic function `volume _ of _ cube(X l,X b)`.

Program 2.37

```
/*Using Standard Parameters with Template Functions, For demonstrating constants can be mixed with the generic data type*/
#include<iostream.h>
#include<conio.h>
const int height=8; /*Here Height of the volume is fixed*/
template<class X>void volume_of_cube(X l,X b)
{
   X volume=l*b*height;
   cout<<"\nVolume of Cube where has constant height as '8' is:"<< volume;
}
void main()
```

```
{
   clrscr();
   volume_of_cube(10,20);
}
```

Output

```
Volume of Cube where has constant height as '8' is:1600
```

2.7.2 Class Templates

As generic functions are possible, defining the generic classes is also possible with templates. Generic classes are very useful for writing a program with the same logic for more number of data types.

The general form of a template class is as follows:

```
template<class template-type>class class-name
{
      //definition of the template class
}
```

The general form of a class template with more parameters is as follows:

```
template<class template-type1,class template-type2,…>class class-name
{
   //definition of the template class
}
```

Here `template-type` can be of any data type, and any number of class template-types can be defined in the template class definition.

An object for the template class can be created using the general form below:

```
Class-name<data type>object;
```

Operations of the stack can be generalized for all the data types. So, instead of writing different classes with different data types, a single generic class stack program can be applied for all the data types (Program 2.38).

Program 2.38

```
/*Generic Class*/
#include<iostream.h>
#include<conio.h>
template<class T>class A
{
   T a,b,sum;
   public:
   void setAB(T x,T y)
   {
      a=x;
      b=y;
      sum=a+b;
   }
void showAB()
```

```
   {
      cout<<"\nA="<<a<<"B="<<b<<"SUM="<<sum;
   }
};
void main()
{
   A<int>obj_int;
   A<float>obj_float;
   clrscr();
   obj_int.setAB(10,20);
   obj_int.showAB();
   obj_float.setAB(0.1,0.2);
   obj_float.showAB();
}
```

Output

```
A=10 B=20 SUM=30
A=0.1 B=0.2 SUM=0.3
```

/*Generic Class*/

Program 2.39

```
#include<iostream.h>
#include<conio.h>
template<class T1,class T2>class SUM
{
   T1 a;
   T2 b;
   public:
   SUM(T1 x,T2 y)
   {
      a=x;
      b=y;
   }
void show()
   {
      cout<<"\nSum of"<<a<<","<<b<<"="<<(a+b);
   }
};
void main()
{
   clrscr();
   SUM<int>obj1(10,20);
   SUM<float>obj2(0.1,0.2);
   obj1.show();
   obj2.show();
}
```

Output

```
Sum of 10,20=30
Sum of 0.1,0.2=0.3
```

In Program 2.39, `SUM` is a class to find the addition of two numbers. The generic class helps the developer to use the same generic template class for the sum of two integers and the sum of two float values. The same `SUM` class is used to create `obj1` with integers and the `obj2` with floats to find the generic problem of the addition of two numbers.

A generic class can have default argument types and with the default values.

2.8 RECURSION

Recursion is a technique of calling a function by itself. Sometimes a function calling itself may produce bugs, but when this technique is correctly used it is very powerful. Recursion can be easily understand with a `fact()` function whose operation is to find the factorial of a number. The non recursive technique of `fact()` appears as follows:

Non-recursive or iterative way of `fact()`:

```
int fact(int n)
{
   int i,result;
   result=1;
   for(i=1;i<=n;i++)
   result=result*i;
   return(result);
}
```

In the above program, the loop in the non-recursive `fact()` function executes from 1 to `n` and multiplies each number.

Recursive way of `fact()`:

Program 2.40

```
#include<iostream.h>
int factorial(int n )
{
   int n;
   int result ;
   cout<<"enter an integer";
   cin>>n
   result=fact(n);
   cout<<"factorial of"<<n<"is:"<<result;
   return 0;
}
   int  fact(int n)
{
   if(n==1) return1
      else
```

```
    return n*fact(n-1);        //call by itself.
}
```

Output

```
Enter an integer
5
Factorial of 5 is:120
```

In Program 2.40, the operation of the recursive `fact()` is somewhat complex, when `fact()` is called with an argument as 1 the function returns 1, otherwise it returns the value of `n*fact(n-1)`. If `n` value is 5, fact is called for the first time with 5 as an argument and second time it is called with an argument of 4 and third time with an argument 3 and so on until `n` is equal to 1. Each call of `fact()` stores the value of `n` while calling itself with another value of `n`.

When a function calls itself, a new set of local variables and arguments is allocated, and the function code executes with these variables. A recursive call does not create a new copy of function but a new a copy of values being operated are created. Every recursive function must be ended, otherwise it will call itself forever. The statement `if(n==1)` in the program does this, when the `n` value becomes 1 the program terminates.

The main advantage of the recursive function is that they can be used to implement several algorithms in the simplest way. The implementation of the quick sort is difficult using the iteration way. Thus, some may think that recursion is easier to iteration.

SUMMARY

- Robustness, adaptability and reusability are goals of an object-oriented design.
- Abstraction, encapsulation and modularity are principles of an object-oriented design.
- Constructors are used for initializing objects. Destructors are used to destroy objects.
- Default constructors, parameterized constructors and copy constructors are the types of constructors.
- Constructor overloading is a technique of defining more than one constructor with the same name.
- Operator overloading changes the meaning of the predefined operators.
- Inheritance derives a new class from the already existing class. Access to base class members is controlled through public, private and protected access specifiers.
- The derived class will have the members of the base class based on the declaration type such as private, public or protected and the access specifier at the time of inheriting the base class.
- Single inheritance and multiple inheritance are the basic classifications of inheritance.
- Inheritance can be further divided into various types: hierarchical inheritance, single-level inheritance, multi-level inheritance, hybrid inheritance and multipath inheritance.
- When the access specifier of the base class is public, all the public members of the base class are accessible to the derived class.
- When the access specifier of the base class is private, then all the public, protected members of the base class become private members of the derived class.
- All the public and protected members of the base class become protected members of the derived class when the access specifier of the base class is protected. All the private members of the base class are not accessible by the derived class.
- Delegation is a mechanism in which the object of one class is used as a member of another class.
- Polymorphism defines one interface with many forms.

- *Virtual* keyword is used to avoid ambiguity in multipath inheritance.
- A virtual function defined in the base class can be redefined by the derived classes.
- A pure virtual function is declared in the base class and is defined by the derived class.
- Generic programming defines a range of related functions called function template and a range of related classes called class template.
- Recursion is a technique of calling a function by itself.

EXERCISES

Fill in the Blanks

1. Reusability is a process of _____ .
2. Abstraction refers to _____ .
3. _____ is used for creating objects.
4. Constructors' and destructors' name is same as _____ .
5. A class having pure virtual functions is called _____ .
6. Polymorphism is a technique of _____ .
7. _____ is the process of combining data and data members into a single unit.
8. Generic programming is a process of _____ .
9. A function is called _____ when it works for all operations.
10. A generic function or class is defined by _____ .
11. _____ are used for writing programs for all data types.
12. Recursion is the process of _____ .

Multiple-choice Questions

1. Modular software design provides _____ .
   a. Adaptability
   b. Portability
   c. Reusability
   d. None of the above
2. Constructors are invoked when _____ .
   a. Objects are created
   b. Variables are initialized
   c. Member functions are called
   d. Objects are deleted
3. Destructors are used to _____ .
   a. Allocate memory for objects
   b. Destroy objects
   c. Create classes
   d. Create objects
4. When constructors are overloaded, they differ by _____ .
   a. Type of arguments
   b. Number of arguments
   c. Return type
   d. Class name
5. The access specifier protected allows _____ .
   a. Access of base class members directly by derived class
   b. Direct access of public member is prevented
   c. Object to access private members
   d. All of the above

6. A class is said to be abstract class _____ .
   a. When at least one pure virtual function exists
   b. When static functions exist
   c. When virtual functions exist
   d. None of the above
7. Ambiguity occurs in _____ type of inheritance.
   a. Single inheritance
   b. Multipath inheritance
   c. Multi-level inheritance
   d. Multiple inheritance

Short-answer Questions

1. Define constructors and destructors.
2. Give the properties of constructors.
3. Discuss the need for virtual functions.
4. Explain pure virtual functions.
5. Give the reasons for the usage of inheritance.
6. Write the advantages and disadvantages of inheritance.
7. What is delegation? Differentiate between delegation and inheritance.

Essay Questions

1. Discuss the goals and principles of object-oriented programming.
2. What is a constructor? Explain various types of constructors.
3. Discuss in detail about operator overloading.
4. What is inheritance? Explain.
5. Give an illustration on generic programming.
6. Can generic function be overloaded. Explain with an example.

Chapter 3

Algorithms

Chapter 3 provides an exclusive study of algorithms. It discusses the basic notations used in the algorithms, various types of algorithms that exist and the performance analysis of algorithms. A detailed explanation regarding space complexity, apriori analysis and asymptotic notations is given. This chapter also elucidates the time complexity and worst, average and best case time complexities.

3.1 INTRODUCTION

An algorithm is a step-by-step process of solving a problem. It is a mathematical procedure or notation to represent the solution of a problem. It should be clear and unambiguous. Algorithm specifies the behaviour of the program to be implemented. It includes data structures used, comments on the usage of the data structures, process statements and subroutines used by the algorithm if any.

Many algorithms may be designed as a solution for a given problem. Among them, the best is selected based on the amount of space and execution time it requires.

3.2 BASIC NOTATIONS

- An algorithm starts with the string "Algorithm" followed by the algorithm name
  - List of input values
  - List of output values
- Assignment of values to a variable will be represented as '←' or '='
  - *Example*: Set radius ← 3 (here the value '3' will be assigned to the variable radius)
- All the variables in the algorithms are scalars (any value can fit into the variable, irrespective of their data types)
  - *Example*: SET radius←3
  - *Example*: SET pi ← 3.141
  - *Example*: SET str ← "Welcome to Algorithms"
  - *Example*: SET array ← {0,0,0}

- Method invocation will be used with `call`
  - *Example*: `call calculat _ area(raidus)`
- Condition will be represented with if–then–else.
- Execution control flows (Example: loops) will be used with simple notations like
  - For loop-variable ←initial value *to* final value
    - `Begin or do`
    - ---
    - ---
    - `End or done`
  - `while` (condition)
    - `begin or do`
    - ---
    - ---
    - `End or done`
  - `Loop`
    - `Begin`
    - ---
    - ---
    - `End or done`
- Array elements can be referenced as
  - `Array[0], Array[1],…Array[n]`
- If the algorithm is for subroutine or a function or a method
  - Starts with algorithm string followed with the string subroutine or procedure strings
- An un-conditional execution control flow jumps can be used in the algorithms
  - *Example*: Goto Step 4
- An algorithm can return a value as an output using the string `return`.

3.2.1 Pseudo Code

Algorithm shows the detailed instructional procedure. When detailed instruction is not needed, then the algorithms can be represented with partial code segments. These segments of the code can be called pseudo code.

3.3 TYPES OF ALGORITHMS

All the algorithms try to solve a given problem. Based on approaches of solving the problem, the algorithms can be classified into the following categories:

- Brute Force Algorithms
- Divide and Conquer Algorithms
- Dynamic Programming Algorithms
- Greedy Algorithms
- Branch and Bound Algorithms
- Recursive Algorithms
- Back Tracking Algorithms

- Randomized Algorithms
- Hill Climbing Algorithms

3.3.1 Brute Force Algorithms

The brute force algorithm tries to find all the possible cases until the satisfactory solution is found.
Example: Finding the best path for the travelling salesman problem.

3.3.2 Divide and Conquer Algorithms

The divide and conquer algorithm tries to divide the solution of a problem into smaller sub-problems of the same type and tries to solve the problems recursively. Finally, all the smaller sub-problem solutions will be combined to get the full solution. These algorithms will always contain recursive calls to the sub-algorithms or sub-routines.
Example: Quick sort, merge sort.

3.3.3 Dynamic Programming Algorithms

The dynamic programming algorithm remembers the past results and uses them to find the results. These algorithms are used for optimization problems. Here, programming refers to finding the new results in combination with the old results.

3.3.4 Greedy Algorithms

The greedy algorithms are also used to solve the optimization problems. At each step, the greedy algorithm tries to get the best solution without considering the future consequences. Finally, all the best local solutions will be combined to get the global optimal solution.

Example: Suppose a bus ticket collector wants to take 5 rupees from his bag:

- First he searches for the 5 rupees coins.
- If it is not found, he tries to take 2 two rupee coins and 1 one rupee coin to make 5 rupees.
- If this is not found, he tries to take 1 two rupee coin and 3 one rupee coins to make 5 rupees.
- If this is also not found, he collects 5 one rupee coins to make 5 rupees.

3.3.5 Branch and Bound Algorithms

The branch and bound algorithms are used to solve the optimization problems in miss-classified problems such as decision trees. As the algorithm process proceeds, it tries to form sub-problems. At each new sub-problem, apply upper and lower bounds. If the bound matches, that will be the feasible solution or else a new partition will be created. This process continues until the best solution is found.

3.3.6 Recursive Algorithms

The simple recursive algorithm uses recursive procedures or repetitive steps to get the solution.
Example: Reversing an 'n-digit' number.

3.3.7 Back Tracking Algorithms

The back tracking algorithms are completely based on the depth-first-recursive search approach. At each step, the algorithm tries to verify whether the solution is found or not. If the solution is found, it exits, otherwise it moves to the previous step (back) and tries to select the move to the possible case. If no more choices are found and the solution is not found, then it returns a failure.
Example: 8-Queen Problem.

3.3.8 Randomized Algorithms

The randomized algorithms make a random choice for at least once to make the decision.
Example: Quick sort uses a random number to select the pivot element.

3.3.9 Hill Climbing Algorithms

The complete solution of the problem starts from the poor solution. By repeatedly applying optimizations, an optimal solution will finally be obtained.

3.4 PERFORMANCE ANALYSIS

An algorithm is a step-by-step process of solving a problem. However, a problem can be solved in many ways. Among all the feasible solutions or algorithms, very few algorithms will give a good approach to the solution with minimum amount of time and hardware requirements. These feasible solutions or algorithms are best suited for the implementations.

3.4.1 Properties of the Best Algorithms

Suppose there are a set of algorithms to a problem. Among all the algorithms, the best algorithm will have the following properties:

- Takes less time for the complete execution (time complexity)
- Takes less memory space (space complexity)
- Correctness of the solution
- Modularity
- Maintainability
- Robustness
- Reliability
- Scalability
- Functionality
- Extensibility
- Security
- Simplicity
- User-friendliness
- Programmer time

The algorithms will give a blueprint of software. The algorithms will allow the analysts to find the time it requires to develop the software and it tells the execution time and memory space it requires.

3.5 SPACE COMPLEXITY

The number of units of space the algorithm requires is called a space complexity of the algorithm. Each vector variable is allocated with one unit of space in the space complexity of the algorithm. But in the programs, space complexity depends on the data types used and number of data items used.

Space complexity of a program is the total memory space required at the execution time. Each and every executable program will have the following sections and occupies memory:

- Text section
- Data section
- Bss section
- Rodata section
- Stack section
- Heap section

3.5.1 Instruction Space

The total memory space occupied by all the instructions at the runtime is called as **instruction space**. The instruction space can be calculated based on the static or dynamic libraries that the program uses. During execution, dynamic libraries (DLLs) can be loaded into the memory based on the runtime and on the conditional requirements. If any dynamic libraries are not introduced in the program, then the total space occupied by all the instructions will be called as **instruction space**.

3.5.2 Text Section of a Program

Text section of an executable program contains the processor instructions of the program. Memory size of each instruction depends on the word length and is platform dependant (processor and the operating system). Total memory space occupied by all the instructions will be called **Instruction space**.

3.5.3 Data Space

Total memory space required by all the initialized variables, uninitialized variables, static and constant variables is called data space of a program.

i. *Data section of the executable program*
   Data section contains all the initialized global variables and static variables.
ii. *Rodata section*
   Rodata is nothing but read only data. Rodata section consists of all the constants of a program. The user cannot modify the data of this section.
iii. *Bss section*
   Bss section contains all the uninitialized global variables and static variables.
iv. *Heap section*
   Heap section contains all the dynamically created data of the program. This section will occupy memory only when the program is in execution.
   The data space is nothing but the total memory space occupied by all the data, Bss, rodata and heap sections.

3.5.4 Stack Space

The stack space is the memory space occupied by all the uninitialized local variables, formal parameters, frame pointer and function return pointer. When the main function or a user-defined function calls another sub-function, the function return pointer will be stored in the stack.

Example 1

A simple C- program that prints the PI value on to the screen.

Test.c program file

```
#include<stdio.h>
#define PI 3.141   /*rodata-8bytes*/
int main()
{
   printf("PI:%f",PI); /*PI:%f will be stored in rodata*/
}
```

Disassembly of all the sections of the executable program using `objdump` utility

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0.text          00000030  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1.data          00000000  00000000  00000000  00000064  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2.bss           00000000  00000000  00000000  00000064  2**2
                  ALLOC
  3.rodata        00000010  00000000  00000000  00000068  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4.comment       0000002d  00000000  00000000  00000078  2**0
                  CONTENTS, READONLY
  5.note.GNU-stack 00000000  00000000  00000000  000000a5  2**0
                  CONTENTS, READONLY
```

Disassembly of section .text

```
00000000<main>:
   0:  8d 4c 24 04             lea    0x4(%esp),%ecx
   4:  83 e4 f0                and    $0xfffffff0,%esp
   7:  ff 71 fc                pushl  0xfffffffc(%ecx)
   a:  55              push    %ebp
   b:  89 e5                   mov    %esp,%ebp
   d:  51              push    %ecx
   e:  83 ec 14                sub    $0x14,%esp
  11: dd 05 08 00 00 00        fldl   0x8
  17: dd 5c 24 04              fstpl  0x4(%esp)
  1b: c7 04 24 00 00 00 00     movl   $0x0,(%esp)
  22: e8 fc ff ff ff           call   23<main+0x23>
  27: 83 c4 14                 add    $0x14,%esp
  2a: 59              pop     %ecx
  2b: 5d              pop     %ebp
```

```
   2c: 8d 61 fc                lea    0xfffffffc(%ecx),%esp
   2f: c3              ret
```

Disassembly of section .rodata

```
00000000<.rodata>:
   0: 50            push   %eax
   1: 49            dec    %ecx
   2: 3a 25 66 00 00 00     cmp    0x66,%ah
   8: 54            push   %esp
   9: e3 a5                 jecxz  ffffffb0 <main+0xffffffb0>
   b: 9b            fwait
   c: c4 20                 les    (%eax),%esp
   e: 09            .byte 0x9
   f: 40            inc    %eax
```

Disassembly of section .comment

```
00000000<.comment>:
   0: 00 47 43              add    %al,0x43(%edi)
   3: 43            inc    %ebx
   4: 3a 20                 cmp    (%eax),%ah
   6: 28 47 4e              sub    %al,0x4e(%edi)
   9: 55            push   %ebp
   a: 29 20                 sub    %esp,(%eax)
   c: 34 2e                 xor    $0x2e,%al
   e: 31 2e                 xor    %ebp,(%esi)
  10: 30 20                 xor    %ah,(%eax)
  12: 32 30                 xor    (%eax),%dh
  14: 30 36                 xor    %dh,(%esi)
  16: 30 33                 xor    %dh,(%ebx)
  18: 30 34 20              xor    %dh,(%eax)
  1b: 28 52 65              sub    %dl,0x65(%edx)
  1e: 64 20 48 61           and    %cl,%fs:0x61(%eax)
  22: 74 20                 je     44 <main+0x44>
  24: 34 2e                 xor    $0x2e,%al
  26: 31 2e                 xor    %ebp,(%esi)
  28: 30            .byte   0x30
  29: 2d            .byte   0x2d
  2a: 33 29                 xor    (%ecx),%ebp
   ...
```

3.5.5 Calculating the Instruction Space

The instruction space is the total space occupied by the text section of the executable program. For the above program, the total text section size is $(30)_{16}$.

Example

```
Idx   Name       Size
0     .text      00000030     the decimal equivalence of (30)16 is 48 bytes.
```

So, the instruction space occupied by the above program is 48 bytes.

In the disassembled program, the instructions of the main function started from 0 to 2f hexadecimal value. So, the total memory occupied by the above main function is $(30)_{16} = (48)_{10}$. Hence, the total instruction space is 48 bytes.

3.5.6 Calculating the Data Space

The data space is the total space occupied by the data, rodata, bss and heap sections of the executable program. So, let the space occupied by each of the said section is found. Because there is no dynamic memory allocation, the heap section will not be created at the runtime.

Table 3.1 Size occupied by the program sections

Idx	Name	Size
1.	.data	00000000
2 .	.bss	00000000
3.	.rodata	00000010

From the disassembled program object dump, Table 3.1 shows the size occupied by each of the program sections in hexadecimal.

3.5.7 Size of Data Section

This section holds all the initialized global variables. In the program, there are no initialized variables. So, the size will be zero. In Table 3.1, the first row shows the size occupied by the data section 00000000 is nothing but zero.

3.5.8 Size of Rodata Section

Rodata section consists of constant values, because the section itself is a read-only section. In the above program, a constant is declared with 3.141. The constant value has taken 00000010 of hexadecimal or 16 bytes in decimal.

3.5.9 Size of bss Section

This section consists of uninitialized global data because there are no uninitialized variables. So, the size of the bss section should be zero. As shown in Table 3.1, the bss section taking 00000000 is nothing but zero.

So, the total data space is

Data space = size (data section) + size (rodata) +size (bss section)
Data space = 0 + 16 + 0= 16 bytes

The total space complexity of the program is

Space complexity = Instruction space + data space + stack space
Space complexity = 48 + 16 + 0 = 64 bytes

So, the space complexity of the given program is 64 bytes.

Example

```
#include<stdio.h>
#include<stdlib.h>
#define PI 3.141 /*rodata*/
int a; /*bss section*/
int b=10; /*data section*/
static int c=20;/*data section*/
int main()
{
   char name[10];/*stack*/
   cout<<"%d %d"<<b<<c;/*rodata*/
}
```

Sections

```
Idx  Name     Size       VMA        LMA        File off   Algn
  0  .text    0000003e   00000000   00000000   00000034   2**2
               CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1  .data    00000008   00000000   00000000   00000074   2**2
               CONTENTS, ALLOC, LOAD, DATA
  2  .bss     00000004   00000000   00000000   0000007c   2**2
               ALLOC
  3  .rodata  00000006   00000000   00000000   0000007c   2**0
               CONTENTS, ALLOC, LOAD, READONLY, DATA
```

Data space

size (data) + size (rodata) + size (bss) = 8 + 6 + 4 = 18 bytes

Instruction space

$(3e)_{16}$ = 62 bytes

Stack space

4 Bytes Frame Pointer + 4 Bytes Stack Pointer + 12 Bytes of array = 20 bytes

Here the 10 bytes of array will be placed onto the word blocks of stack. When one word = 4 bytes, 10 bytes will be placed into 12 bytes (i.e 3 words).

Space complexity = 18 + 62 + 20 = 100 bytes

3.6 APRIORI ANALYSIS

The *empirical* or *theoretical* approach may be used for computing the time complexities of algorithms.

In the *empirical* or *posterior* testing approach, algorithms are completely implemented and are executed for various instances of the problem on a computer. The time taken to do this is considered and compared. Among the various algorithmic solutions, the algorithm that takes less time to implement it can be considered as the best one.

In the *theoretical* or *apriori* approach, the resources that are required by the algorithm are determined mathematically.

When a problem is considered, the apriori approach is performed as a function of a parameter that is related to instances of the problem. The size of input instances is often used as a parameter. For different classes of problems there are algorithms that consider the number of basic operations or element comparisons or multiplications as a parameter in order to find out the efficiency.

Apriori analysis is totally machine, language and program independent. In apriori analysis, algorithm efficiency can be studied over any input instances of any size.

For a given program, apriori analysis computes its efficiency by considering the frequency count of the statement as a function of the total frequency count of the statements in the program. Here, the frequency count is the number of times the statement is getting executed in the program.

Consider the statement `c=a+b` that is appearing in the following three program segments. The frequency count is estimated as

Program Segment 1	**Frequency Count**
. . .	. . .
c=a+b;	1
. . .	. . .

Total Frequency Count is 1

Program Segment 2	**Frequency Count**
. . .	. . .
for i=1 to n do	n+1
c=a+b;	n
end	n
. . .	. . .

Total Frequency Count is 3n+1

Program Segment 3	**Frequency Count**
. . .	. . .
for j=1 to n do	n+1
	$\sum_{j=1}^{n} (n+1) = (n+1)n$
for i=1 to n do	
c=a+b;	n^2
	$\sum_{j=1}^{n} n = n^2$
end	

Total Frequency Count is $3n^2+3n+1$

In Program segment 1, the frequency count of the statement is 1. In Program segment 2 it is **n** as its `for` loop executes it for **n** times. In Program segment 3 it is $\mathbf{n}^2$ because it is embedded in a nested `for` loop where each of them are executed for **n** times.

In the three program segments, the total frequency counts are 1, (3n+1) and $3n^2+3n+1$, respectively. The order of magnitude of the total frequency counts is proportional to 1, n and n^2, respectively, in the three program segments and they can be expressed as O(1), O(n) and $O(n^2)$. The notation O is explained in the Section 3.7.

3.7 ASYMPTOTIC NOTATION

Some of the algorithms may be short in the number of instructions but when input varies, the total number of execution steps grows gradually. The asymptotic notations will help to predict the program behavior at the runtime. It provides a measurable value. Using the asymptotic notations, the best algorithm can be found, which can perform better than the other possible algorithms of a problem. The asymptotic notations consider the operation count or step count of an algorithm.

Example

Let `A` and `B` be the two algorithms for a problem where

$A=3n^2+2n$
$B=5n$

The computational growth of two algorithms varies with the `n` value as shown in Figure 3.1. In the graph, the *x*-axis represents the input variable values, i.e. `n`, and the *y*-axis represents the time complexity.

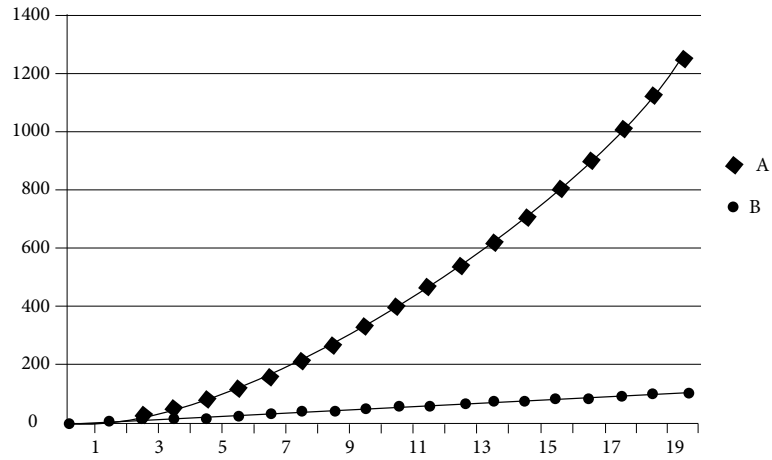


Figure 3.1 Computational growth changes

Algorithms `A` and `B` work similarly with respect to the time complexities when `n=1`.

Time complexity of `A`=5 and
Time complexity of `B`=5
when `n=2` Time complexity of `A`=16 and
Time complexity of `B`=10

Therefore, for `n=2`, `B` is the optimal algorithm.
Similar changes can be observed for other values of n as shown in the above graph.

3.7.1 Big oh Notation (O)

Definition: Let $f(x)$ and $g(x)$ be two non-negative functions defined on a set of real numbers. $f(x) = O(g(x))$ iff there exists positive constants 'c' and n_0 such that $f(n) \leq c*g(n)$ for all $n \geq n_0$.

The big oh notation considers the upper bound of the function.

Apriori analysis of Big oh notation: In this analysis, the maximum frequent items will be selected. In this approach, the maximum possible upper limit of the function constant value will be selected to represent the function as a Big oh notation.

For example:

$f(x)$	$g(x)$	
$10n^3+3n+2$	n^3	$f(n) = O(11n^3)$
$4n+2$	n	$f(n) = O(5n)$
5	1	$f(n) = O(1)$

3.7.1.1 Finding g(x), c and n_0 Values

Example 1

Let $f(n) = 3n+2$ for an algorithm. Computation of c, n_0 and $g(n)$ can be performed as $f(n) = 3n+2$.
The function can be written in terms of $g(n)$ as

$f(n) \le c \star g(n)$
$3n+2 \le c \star g(n)$ because the function value is more than $3n$, consider $c=4$, then
$3n+2 \le 4 \star g(n)$ define $g(x)=n$
$3n+2 \le 4 \star n$ find n_0.

Put $n=1$ $3 \star 1+2 \le 4 \star 1$ (condition failed)
So put $n=2$ $3 \star 2+2 \le 4 \star 2$ (true)
Put $n=3$ $3 \star 3+2 \le 4 \star 3$ (true)
Put $n=4$ $3 \star 4+2 \le 4 \star 4$ (true)

So, as per the mathematical induction, the function $f(x)$ can be written in terms of $g(x)$, and is true from $n_0=2$ to all positive integers.
The function $f(x)$ can be defined as $f(x) \le 4n$ where $n \ge 2$

Example 2

Let $f(n) = 3n^4 + 2n^2$ for an algorithm. Computation of c, n_0, and $g(n)$ can be performed as $f(n) = 3n^4 + 2n^2$

f(n) ≤ c⋆g(n) (Big oh notation)
$3n^4 + 2n^2 \le c \star g(n)$
$3n^4 + 2n^2 \le 4 \star g(n)$
$3n^4 + 2n^2 \le 4 \star g(n)$ Let $g(n) = n^4$
$3n^4 + 2n^2 \le 4 \star n^4$

for $n=1$ $5 \le 4 \star 1$ (Failed)
for $n=2$ $56 \le 64$ (True)
for $n=3$ $261 \le 324$ (True)
So, for all $n \ge 2$ function $f(x)$ can be written as $4 \star n^4$.

3.7.2 Omega Notation (Ω)

Definition: $f(n) = \Omega(g(n))$ iff there exists positive constants c and n_0 such that $f(n) \ge c \star g(n)$ or all $n \ge n_0$. In this case, the maximum possible lower bound will be selected.

For example

$f(x)$	$g(x)$	
$10n^3+3n+2$	n^3	$f(n) = O(n^3)$
$4n+2$	n	$f(n) = O(n)$
5	1	$f(n) = O(1)$

3.7.2.1 Finding g(x), c and n_0 Values

Example 1

Let $f(n) = 3n+2$ for an algorithm. Computation of c, n_0, and $g(n)$ can be performed as $f(n) = 3n+2$.
The function can be written in terms of $g(n)$ as

$f(n) \geq c \star g(n)$
$3n+2 \geq c \star g(n)$, let function value nearest minimum be $3n$, consider $c=3$, then
$3n+2 \geq 3 \star g(n)$ define $g(x)=n$
$3n+2 \geq 3 \star n$ find n_0.

put $n=1$ $5 \geq 3$ (True)
put $n=2$ $3*2+2 \geq 3 \star 2$ (true)
put $n=3$ $3*3+2 \geq 3 \star 3$ (true)
put $n=4$ $3*4+2 \geq 3 \star 4$ (true)

So, as per the mathematical induction, the function $f(x)$ can be written in terms of $g(x)$, and is true for all n_0 positive integers.

The function $f(x)$ can be defined as $f(x) \geq 3n$ where $n \geq 1$.

Example 2

Let $f(n) = 3n^4 + 2n^2$ for an algorithm. Computation of c, n_0, and $g(n)$ can be performed as $f(n) = 3n^4 + 2n^2$.

$f(n) \geq c \star g(n)$ (Omega notation)
$3n^4 + 2n^2 \geq c \star g(n)$
$3n^4 + 2n^2 \geq 3 \star g(n)$
$3n^4 + 2n^2 \geq 3 \star g(n)$ Let $g(n) = n^4$
$3n^4 + 2n^2 \geq 3 \star n^4$

for $n=1$ $5 \geq 3$ (True)
for $n=2$ $56 \geq 48$ (True)
for $n=3$ $261 \geq 243$ (True)

So, for all $n \geq 1$ function $f(x)$ can be written as $3 \star n^4$.

3.7.3 Theta Notation (θ)

Definition: $f(n)=\theta(g(n))$ iff there exists positive constants $c1$ and $c2$ and n_0 such that $c_1 \star g(n) \leq f(n) \leq c_2 \star g(n)$ for all $n \geq n_0$.

In this notation, the value that lies in between the upper bound and maximum possible lower bounds will be selected.

Example

$f(x)$	$g(x)$		
$10n^3+3n+2$	n^3	$f(n)=\theta(n^3)$	since $f(n)>10n^3$ and $f(n)<15n^3$ for all $n \geq 1$
$4n+2$	n	$f(n)=\theta(n)$	since $f(n)>4n$ and $f(n)<6n$ for all $n \geq 1$

In the first example, $f(x)=10n^3+3n+2$. The value of the function is always greater than $g(x)=10n^3$ because some other expression and constant is available after the $g(x)$ in the $f(x)$. Consider $n=1$, in this case $f(x)=15\times 1=15$. So, to find the upper bound for this function, it needs to consider multiples of 15, then $g(x)=15n^3$ will always greater than $f(x)$.

In the second example, $f(x)=4n+2$. The value of the function is always greater than $g(x)=4n$ because some other expression and constant is available after $g(x)$ in $f(x)$. Consider $n=1$, in this case $f(x)=4\times 1+1=5$. So, to find the upper bound for this function, it needs to consider multiples of 5, then $g(x)=5n$ will always greater than $f(x)$.

3.7.4 Little oh Notation(o)

Definition: $f(n)=o(g(n))$ if $f(n)=O(g(n))$ and $f(n)\neq\Omega(g(n))$.

Example

$f(n)$	$g(n)$	
$10n+2$	n^2	$f(n)=o(n^2)$ since $f(n)$ and $f(n)\neq\Omega(g(n))$; however $f(n)=O(n^2)$

3.8 TIME COMPLEXITY

Time complexity of a program can be defined as the total time taken for compilation and execution. The program has to be compiled before running it. Once the compilation process creates the executable binary file, to run the program, compilation is not required as long as there are no changes in the code. An executable binary file is required to run a program. Hence, consider only the execution time for calculating the time complexity.

The time complexity depends on the space complexity that the program uses, time taken to load the program, total number of executable instructions in the binary file, number of control loops present, number of function calls made, number of arithmetic operations used, number of console input/output operations, number of read/write locks and finally the input values of the program.

Consider a program to display natural numbers from 1 to n. If $n=2$, then the print function will be called only 2 times. If the n value is very high, then the print function will be called for n number of times.

If a program requires more memory than the available physical memory, then the processor makes use of virtual memory, which is placed in secondary storage devices such as a hard disk. Otherwise, the processor waits till sufficient physical memory is available. So, the runtime of a program depends on the space complexity. When the space complexity of the program is more than the available physical memory, then the runtime will be more. So, the time complexity of a simple program is very less than the complex huge program.

As the time complexity depends on so many factors, there is no exact procedure to find the time complexity of a program. But, by considering the basic operations and the number of times the operations are called, the time complexity can be estimated. The time complexity can be calculated by estimating the total number of instructions executed.

3.8.1 Time Complexity Analysis of Bubble Sort

Program 3.1

```
//Program for Bubble Sort
#include<iostream.h>
#define MAX 256
void bubble(int a[],int n)
{
int i,j,temp;
//Start of Bubble Sort
for(i=0;i<n;i++)
   {
   for(j=0;j<(n-1);j++)
      {
      if(a[j]>a[j+1])
         {
         temp=a[j];
         a[j]=a[j+1];
         a[j+1]=temp;
         }
      }
   }
// End of Bubble Sort

cout<<"\nSorted Array:";
for(i=0;i<n;i++)
   {
   cout<<"\n"<<a[i];
   }
}
void main()
{
int i,n,a[256];
cout<<"\nEnter size:";
cin>>n;
for(i=0;i<n;i++)
   {
   cout<<"\nEnter Element:";
   cin>>a[i];
   }
bubble(a,n);
}
```

In Program 3.1, the bubble sort program accepts an array and size of the array as input arguments. The first `for` loop continues to execute for `n` times, the inner `for` loop executes for `n–1` times and each time, three statements will be executed. So, the total number of executable statements in the program is $n*(n-1)*3=3n^2-3n$. Therefore, time complexity of the program is $O(n^2)$.

3.8.2 Time Complexity Analysis of Selection Sort

Program 3.2

```
//Program for Selection Sort
#include<iostream.h>
//Function returns the maximum index of the array.
int maxindex(int a[],int n)
{
int i,temp,max_index=0;
for(i=1;i<n;i++)
   {
   if(a[max_index]<a[i])
      {
      max_index=i;
      }
   }
return(max_index);
}
void main()
{
int i,n,temp,a[256];
cout<<"\nEnter size:";
cin>>n;
for(i=0;i<n;i++)
   {
   cout<<"\nEnter Element:";
   cin>>a[i];
   }

/*Selection Sort Procedure*/
for(int s=n;s>1;s--)
   {
   int j=maxindex(a,s);
   temp=a[j];
   a[j]=a[s-1];
   a[s-1]=temp;
   }

/*Selection Sort Procedure*/
cout<<"\nSorted";
for(i=0;i<n;i++)
   {
   cout<<"\nElement:"<<a[i];
   }
}
```

In Program 3.2, the `maxindex` function returns the index of the maximum number. In the worst case it runs for `n` times. The selection sort procedure has one loop, executes `n` times. For each `n` value, `maxindex` will

be called and three statements of swapping procedure will be called for n times. Therefore, time complexity will be $f(x)=n^2+3n = O(n^2)$.

3.9 WORST CASE, AVERAGE CASE AND BEST CASE COMPLEXITY

The number of steps an algorithm takes can be known when it is executed on the given input instance. To check the correctness of the algorithm, execute it for all instances. An algorithm when executed on all possible instances of data, the best, worst and average case complexities can be estimated. Suppose for the problem of sorting, the possible instances are the possible arrangements of all the possible keys. There is a need for differentiating at least three cases for which the efficiency of algorithm has to be determined.

3.9.1 Worst Case

The worst case complexity of an algorithm is when the algorithm takes maximum number of steps on any instance of size *n*. The Big oh notation tells the maximum upper bounds of the running time complexity. The Big oh time complexity can be called as worst case, because at any cost the maximum boundary will be O (Big oh) notation expression.

3.9.2 Average Case

The average case complexity of an algorithm is when the algorithm takes the average number of steps on any instance of size *n*. The Theta notation tells the time complexity that lies in between the maximum lower bounds and minimum upper bounds of the running time complexity. The Theta time complexity can be called as average case because at any cost the time complexity will not exceed Big O and Omega minimum boundary.

3.9.3 Best Case

The best case complexity of an algorithm is when the algorithm takes the minimum number of steps on any instance of size *n*. The Omega notation tells the time complexity, i.e., maximum possible lower bounds of the given function.

Consider the linear search performed on an unsorted array to find an element. If the element is found in the first cell then it is a best case. If the element is found in the last cell or not found in the array, then it is a worst case. In the worst case, all the elements will be checked. Then what is the average case? It is assumed that the chance of finding the element is equal all through the array. In this case, the probability that the element is found at the first cell is $1/n$, the probability of finding the element in the second cell is $1/n$ and so on. Therefore, averaging all these probabilities gives the average number of steps taken to find the element. But, if the probability differs, then the average case complexity differs. The probability of finding an element in a particular cell has no effect on the best and worst cases.

SUMMARY

- Algorithm is step-by-step process to solve a problem.
- The algorithms are the prior written steps to the implementation of software. The algorithm shows the clear path to get the required software solution.
- The segments of code used to represent an algorithm can be called as a pseudo code.
- Space complexity is the number of units of space the algorithm requires.

- Space complexity of a program is the total memory space required at execution time.
- The total memory space occupied by all the instructions at the runtime will be called as instruction space.
- Text section of an executable program contains the processor instructions of the program.
- Total memory space required by all the initialized variables, uninitialized variables, static and constant variables is called data space.
- Time complexity of a program can be defined as the total time taken for compilation and execution time.
- The time complexity can be estimated by considering the basic operations and the number of times the operations are called.
- The time complexity can be calculated by estimating the total number of instructions executed.

EXERCISES

Fill in the Blanks

1. An algorithm with partial code segments is called _____.
2. _____ is a property of a best algorithm.
3. The asymptotic notations consider the _____ of an algorithm.
4. The Big oh notation considers the _____ of the function.
5. The greedy algorithms are used to solve _____ problems.

Multiple-choice Questions

1. Space complexity depends on the _____ and _____ used.
   a. Data types, number of data items
   b. Data types, functions
   c. Functions, number of data items
   d. Number of function calls, number of data items
2. At the runtime the total memory space occupied by all the instructions is called _____ .
   a. Data section
   b. Instruction space
   c. Stack space
   d. None
3. _____ is the property of the best algorithm.
   a. Portability
   b. Modularity
   c. Reliability
   d. Both b and c
4. The algorithm takes _____ number of steps in the worst case.
   a. Maximum
   b. Minimum
   c. Average
   d. None
5. The _____ notation tells maximum possible lower bounds of the given function.
   a. Omega
   b. Theta
   c. Big O
   d. None

Short-answer Questions

1. What is an algorithm?
2. Discuss various types of algorithms.

3. Give the properties of best algorithms.
4. Define space complexity.
5. Define time complexity.
6. What are the different cases of measuring the efficiency of algorithms?
7. Define Big oh-O, Theta-Θ, Omega-Ω notations.

Essay Questions

1. Explain space complexity.
2. Discuss in detail about apriori analysis.
3. Write short notes on asymptotic notation.
4. What is a time complexity? Explain it with an example.
5. Write short notes on worst, average and best cases complexities.

Chapter 4

Arrays

From this chapter the study of data structure starts with the definition of data structure and classifying them as linear and non-linear data structures. Array, a linear data structure, is considered in this chapter. Here, a clear explanation about what an array is, array types and array representation is given. This chapter also discusses the initialization of arrays, accessing values of an array, array operations and how arrays are passed as parameters. The character sequences, applications and ADT of arrays are also covered in depth.

4.1 INTRODUCTION

Data structure is the logical or mathematical model of representing data. The data which comprise the data structure and its fundamental operations is known as Abstract Data Type (ADT).

Abstract data type is a conceptual specification of a data type which includes description of objects and operations but not implementation in order to fulfill the following:

- Maximize reuse
- Promote object-oriented techniques
- Make the code easier to maintain

Abstract Data Type:

An ADT contains:

- Various attributes
- May have maximum size
- Operations

In order to create an actual instance of the ADT in a program, the real data that needs to be manipulated is added to it, but the ADT manipulation need not change.

Data structures are classified in two ways:

- Linear data structures
- Non-linear data structures

In linear data structures, all the elements are arranged in sequence. Examples for linear data structures are arrays, linked lists, stacks and queues.

In contrast to the linear data structures, in non-linear data structures the elements do not have a sequence. Examples for such structures are trees, graphs and tables.

4.1.1 Array

An array is a finite set of homogenous elements stored in contiguous memory locations which are referenced, respectively, by adding an index as a unique identifier. The number of data elements in an array can be obtained by the index using the following formula:

$$\text{Size of array} = UB - LB + 1$$

where UB is the upper boundary value and LB is the lower boundary value. The size of array is UB when LB = 1.

The elements of an array can be denoted by using subscripts or brackets such as `A[1], A[2], A[3], … , A[N]` or `A(1), A(2), A(3), …, A(N)`. Let A be an array of 5 integers such that `A[0]=3, A[1]=4, A[2]=5, A[3]=2` and `A[4]=8`.

4.2 ARRAY TYPES

The number of subscripts of an array decides the dimension of an array. Based on the dimensions, arrays can be classified as single-dimensional arrays and multi-dimensional arrays.

4.2.1 Single-dimensional Array

When the elements in an array are referenced by a single subscript, then the array is called a one-dimensional array.

4.2.2 Multi-dimensional Array

Multi-dimensional arrays can be defined as "array of arrays". Elements of multi-dimensional arrays are represented using more than one subscript. For example, a two-dimensional array can be imagined as a table with elements, which are of the same data type (Figure 4.1).

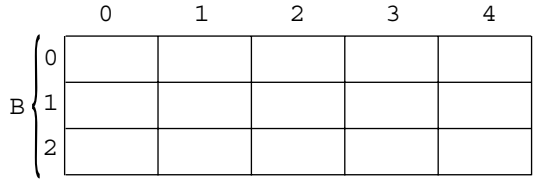


Figure 4.1 Two-dimensional array

`B` represents a two-dimensional array of 3 elements where each element is a one-dimensional array consisting of 5 elements of type integer. It can be declared as

```
int B[3][5];
```

where `B[1][3]` refers to the element horizontally fourth and vertically second (Figure 4.2).

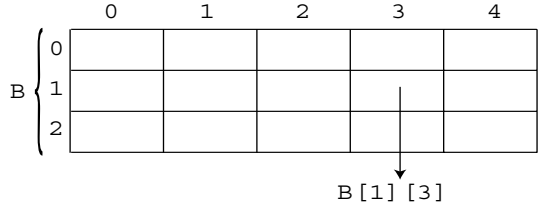


Figure 4.2 Referencing `B[1][3]`

Let A[1:R, 1:C] be a two-dimensional array with R and C data elements where R is the number of rows and C is the number of columns in the array A. Each element in array is represented by a pair of integers called subscripts such as A[i][j], where i range from 1 to R and j range from 1 to C.

Two-dimensional arrays are also called matrix arrays. Figure 4.3 illustrates a two-dimensional array.

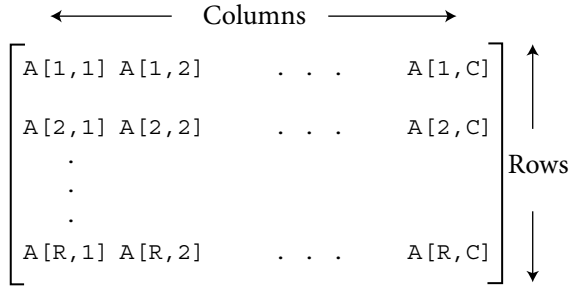


Figure 4.3 Two-dimensional R×C array A

4.2.3 N-dimensional Array

An N-dimensional m1×m2×…×mn array A is a collection of m1, m2, m3 … mn data elements, where each element is referred by a list of n integer such as j1, j2, …, jn called subscripts where 1<=j<m1, 1<=j2<=m2, 1<=j3<=m3, … , 1<=jn<mn.

4.3 ARRAY REPRESENTATION

Accommodating arrays of various dimensions in the memory is explained as follows:

Representation of single-dimensional arrays: Representation of array in memory is very simple (Figure 4.4). Suppose an array A[10] is to be stored in memory and the memory location of the first element in an array can be obtained from the formula:

Address of A[K]=R+(K–1).

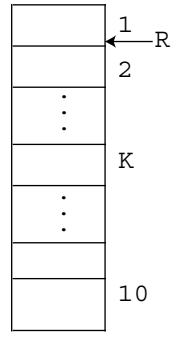


Figure 4.4 Array representation

Generalizing the formula gives Address of `A[K]=R+(K-L)*W`, where `R` is the location of the first element, `L` is the lower bound and `W` is the number of words (size) required by each element. This is known as the indexing formula which is used for mapping the logical array to the physical array. Using this formula the K^{th} element address can be found easily if the first element's location is known, i.e. `R` (Figure 4.5).

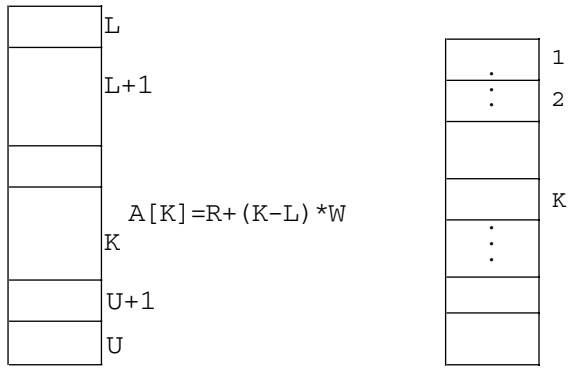


Figure 4.5 General indexing formula

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
data type Array-name[elements];
```

where data type is a valid type like `int`, `float`, etc. Array-name is a valid identifier and the elements field specifies the maximum number of elements the array can hold. Therefore, an array `A` of integer type can be declared as

```
int A[5];
```

Representation of two-dimensional arrays: A two-dimensional array can be represented in two different ways in the memory:

- Row major order
- Column major order

The elements are stored on a row-by-row order, that is, the first row elements followed by the second row elements and so on in the row major order. Elements are stored column by column, that is, the first column elements followed by the second column and so on in the column major order. Consider the array `A[R,C]` in Figure 4.6.

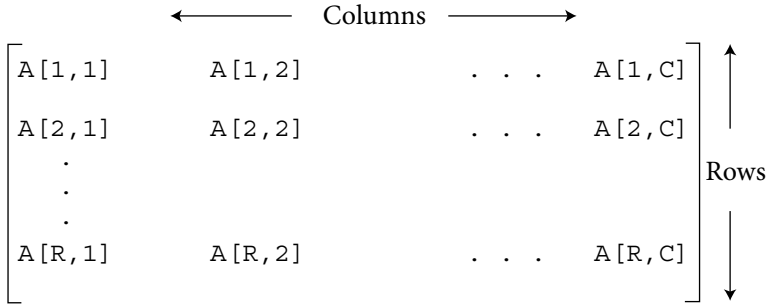


Figure 4.6 An array `A[R,C]`

The representation of A[R,C] using row major order is as shown in Figure 4.7.

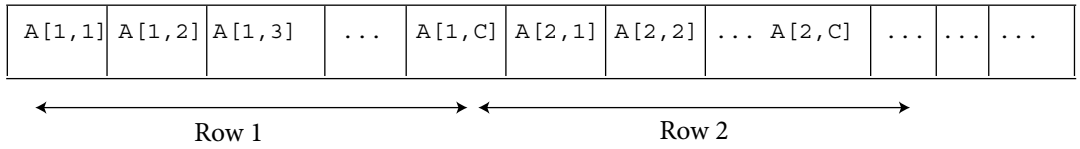


Figure 4.7 Array representation using row major order

Column major order representation is as shown in Figure 4.8.

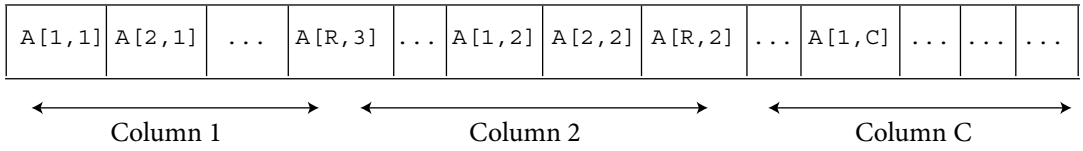


Figure 4.8 Array representation using column major order

The indexing formula for arrays with different orders is as follows:

Row major order: The address of A[i,j] is

A[i,j]= all the elements in the first $(i-1)^{th}$ rows + number of elements from the i^{th} to j^{th} column

$=(i-1) * c+j$

Column major order:

A[i,j]= all the elements in the first $(j-1)^{th}$ column + number of element from the j^{th} column to i^{th} row

$= (j-1) * r + i$

4.4 INITIALIZING ARRAYS

When declaring a regular array of local scope (within a function, for example), no default initialization is done to the elements. So they should be initialized, whereas the elements of global and static arrays are automatically initialized with that default value, that is zero. In both the cases of local and global, while declaring an array, there is a possibility of assigning initial values to each one of its elements by enclosing the values in braces { }. For example:

```
int A[5]={3,4,5,2,8};
```

It creates an array as shown in Figure 4.9.

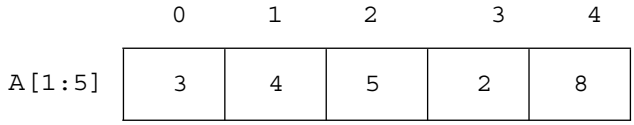


Figure 4.9 Initialization of array

The number of values between braces { } must not exceed the array size. In the example in Figure 4.9, array A is declared with 5 elements, and the list of initial values within braces { } is assigned to each of the 5 elements.

During initialization of an array, it is not compulsory to specify the size, and the square brackets may be left empty [ ] as

```
int A[]={3,4,5,2,8};
```

In this case, the compiler will assume the size of the array that matches the number of values in the braces { }.

4.5 ACCESSING VALUES OF AN ARRAY

The elements of an array can be accessed individually like normal variables, thus being able to read and modify their values. The format is as follows:

```
Array-name[index]
```

Consider the previous example in which A had 5 elements, and each of those elements was of type int the name that is used to refer each element is given in Figure 4.10.

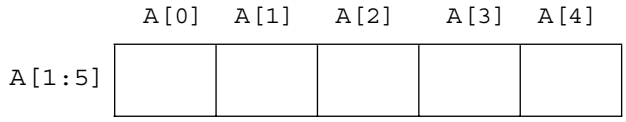


Figure 4.10 Referencing elements

To store the value 75 in the third element of A, the statement is written as A[2]=75; to pass the value of the third element of A to any variable then the statement is a = A[2]. Notice that the third element of A is specified as A[2], since the array index starts from zero, i.e. the first one is A[0], the second one is A[1], so the third one is A[2] and the last element is A[4]. Therefore, A[5] refers to the sixth element of A and this exceeds the size of the array.

In C++ it is syntactically correct to exceed the valid range of indices for an array. This will create problems, since accessing out-of-range elements do not cause compilation errors but can cause run-time errors. At this point it is important to clearly distinguish between the two uses of subscripts [ ] related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared, and the second one is to specify indices for concrete array elements.

```
int A[5];        //declaration of a new array
A[2];            //access to an element of the array
```

The data type always precedes a variable or array declaration, while it never precedes an access.

4.6 ARRAY OPERATIONS

Various operations that can be performed on array are

- Traversing
- Insertion
- Deletion
- Sorting
- Searching

4.6.1 Traversing

Visiting each element of an array exactly once is called traversing. This operation is used to display the content of each element of the array to count number of elements in an array easily.

Algorithm 4.1

```
1  Set i=LB          //Initializing the counter variable
2. Repeat step3 and 4 while i<=UB
3. VISIT (A[i])     //process the element
4. i=i+1           //Moving to next location, increment the counter
5. End
```

Or using `for` loop

```
1. Repeat for i=LB to UB
2. VISIT (A[i])
3. End
```

In Algorithm 4.1 `VISIT()` is a procedure which performs operations on the array, for example, displaying the elements of an array, etc.

4.6.2 Insertion

Insertion is an operation that is used to add an element into the array. Inserting an element at the end of the array is easy when memory is available. When insertion is to be done at the middle of the array, then most of the elements must be moved downwards to maintain the order of the elements.

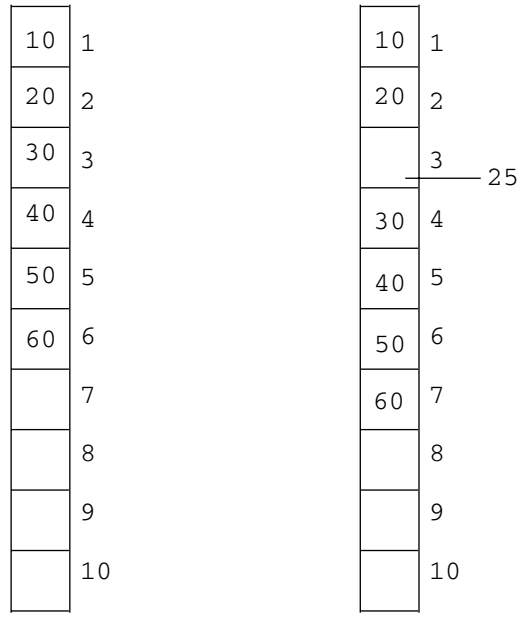
Algorithm 4.2 illustrates the insertion of an element `ITEM` into the array `ARR` at the given index position `POS`.

Algorithm 4.2

```
INSERT(ARR,N,POS,ITEM)
ARR is an array of size N. POS is the index to insert the element.
1. Set i=N          //initialize the counter variable
2. Repeat steps 3 and 4 while i>=POS
3. Move ith element downwards
   Set ARR[i+1] = ARR[i].
4. Set i=i-1        //decrement the counter variable.
5. Insert element at 'POS'
   Set ARR[POS]=ITEM
6. Set N=N+1        //increment N value
7. End
```

Algorithm 4.2 creates memory space by moving each element downwards from the POS^{th} location. Move the element in reverse order, that is move `ARR[N]` first, then `ARR[N-1]` and `ARR[POS]`, otherwise the data might be overlapped. After creating space, insert the element `ITEM` in the POS^{th} location.

Example: `ARR` is an array of size 10 containing 6 elements. All the items in the array are in increasing order Figure 4.11(a). Let `25` be added to the array, then the data elements `30,40,50,60` must be moved downwards one location each as shown in Figure 4.11(b).



(a) Array with 6 elements (b) Moving elements downwards

Figure 4.11 Insertion operation

First the `ITEM ARR[6]=60` is moved to `ARR[7]` and `A[5]` is moved to `ARR[6]` and so on.

4.6.3 Deletion

Deletion is an operation that is used to remove an element from the array. The deletion of an element at the end of the array can easily be done, whereas the deletion of the element in the middle of the array or at specified locations makes the data elements to move one location upwards to fill the array.

Algorithm 4.3 deletes an element from array `ARR` with `N` elements at a specified location `POS` where `POS<=N` and assigns it to the variable `ITEM`.

Algorithm 4.3

```
DELETE(ARR,N,POS,ITEM)
1. Set ITEM=ARR[POS]
2. Repeat for i=POS to N-1
3. Set ARR[i]=ARR[i+1] //move elements upward
4. Set N=N-1
5. End
```

Example: Consider an array ARR of size 10 and with 6 data elements. Deleting the element at A[2] makes the other element to move one location upward as shown in Figure 4.12.

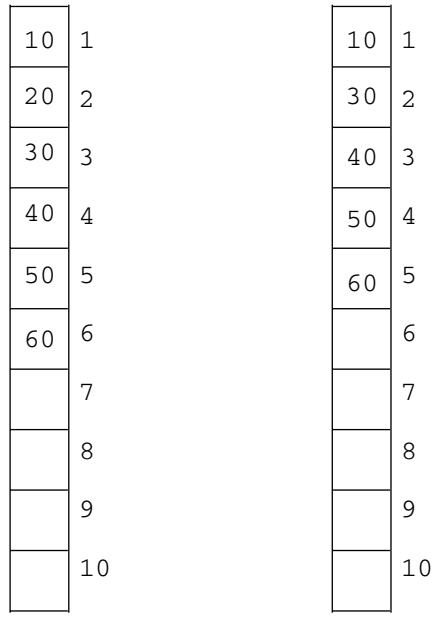


Figure 4.12 Deletion operation

4.6.4 Sorting

Sorting is an operation that is used to arrange the elements of an array either in ascending or descending order. Algorithm 4.4 illustrates sorting of the elements in an ascending order .

Algorithm 4.4

```
SORT :
1. Set N=UB
2. while i>=LB do
3. Set q=LB    //comparing the first
4. while j<i do
5. check whether A[j]&A[j+1] are in order, if so swap the elements
   SWAP(A[j], A[j+1])
6. Set j=j+1
7. Set i=i-1
8. End
```

4.6.5 Searching

Searching is an operation that is used to find the location of a particular element from the list of elements in an array. When the item is found, then search is said to be successful. There are different types of searching algorithms. Usage of these algorithms depends upon the way the data elements are arranged in the data structures. A detailed discussion of searching techniques can be found in Chapter 16.

Program 4.1

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
template<class t>
class array
{
   t a[50];
   int n,size;
   public:
   array(int m)
   {
      size=0;
      n=m;
   }
   void insert();
   void deletee();
   void display();
};
template<class t>
void array<t>::insert()
{
   t ele;
   if(size>=n)
   {
      cout<<"array is overflow\n";
      return;
   }
   size++;
      cout<<"enter the element\n";
   cin>>ele;
   a[size]=ele;
}
template<class t>
void array<t>::deletee()
   {
      t ele;
      if(size==0)
      {
         cout<<"array is underflow\n";
         return;
      }
   ele=a[size];
   size--;
   cout<<"the deleted element is"<<ele<<"\n";
}
template<class t>
void array<t>::display()
```

```
    {
       int i;
       if(size==0)
       {
          cout<<"array is underflow\n";
          return;
       }
    cout<<"the elements are\n";
    for(i=1;i<=size;i++)
       cout<<a[i]<<" ";
       cout<<"\n";
    }
void main()
{
    int ch,n;
    clrscr();
       cout<<"enter the size of the array\n";
    cin>>n;
    array<float>a(n);
    while(1)
    {
       cout<<"menu\n";
       cout<<"1.add\n";
       cout<<"2.delete\n";
       cout<<"3.display\n";
       cout<<"4.exit\n";
       cout<<"enter your choice\n";
       cin>>ch;
       switch(ch)
       {
          case 1:
             a.insert();
             break;
          case 2:
             a.deletee();
             break;
          case 3:
             a.display();
             break;
          case 4:
             exit(0);
          break;
       default:
          cout<<"invalid option\n";
       break;
       }
    }
}
```

Output

```
enter the size of the array
10
menu
1.add
2.delete
3.display
4.exit
enter your choice
1
enter the element
2
menu
1.add
2.delete
3.display
4.exit
enter your choice
1
enter the element
3
menu
1.add
2.delete
3.display
4.exit
enter your choice
3
the elements are
2 3
menu
1.add
2.delete
3.display
4.exit
enter your choice
2
the deleted element is 3
menu
1.add
2.delete
3.display
4.exit
enter your choice
4
```

4.7 ARRAYS AS PARAMETERS

Arrays can be passed as parameters to functions. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but is allowed to pass its address. In practice, this has almost the same effect and it is much faster and is a more efficient operation.

In order to accept arrays as parameters the only thing that is to be done while declaring a function is to specify the type and name of the array as its parameter, an identifier and a pair of brackets[]. For example, the following function:

```
void procedure(int arg[])
```

accepts a parameter which is an array of type `int`, called `arg`. In order to pass to this function, an array is declared as:

```
int myarray[40];
```

it would be enough to write a call like

```
procedure(myarray);
```

Program 4.2

```
/*arrays as parameters*/
#include<iostream>
using namespace std;
void printarray(int arg[],int length)
   {
      for (int n=0;n<length;n++)
         cout<<arg[n]<<"  ";
         cout<<"\n";}
int main()
   {
       int firstarray[]={5, 10, 15};
       int secondarray[]={2, 4, 6, 8, 10};
       printarray(firstarray,3);
       printarray(secondarray,5);
       return 0;
   }
```

Output

```
5 10 15
2 4 6 8 10
```

In Program 4.2, the first parameter (`int arg[]`) accepts any array whose elements are of type `int` of any length. So, a second parameter which specifies the length of each array is passed to the function as the first parameter. This allows the `for` loop that prints out the array to know the range for iterations in the passed array without going out of range.

In a function declaration it is also possible to include multi-dimensional arrays. The format for a tridimensional array parameter is as follows:

```
base _ type[][depth][depth]
```

A function with a multidimensional array as the argument could be

```
void procedure(int myarray[][3][4])
```

Notice that the first brackets `[]` are left blank while the other two are with values. This is so because the compiler must be able to determine the depth of each additional dimension within the function.

4.8 CHARACTER SEQUENCES

The C++ Standard Library possesses a powerful string class, which is very useful to handle and manipulate strings. However, because strings are in fact a sequence of characters that can be represented as arrays of `char` elements. The array

```
char C[20];
```

can store up to 20 elements of type `char`. It can be represented as given in Figure 4.13.

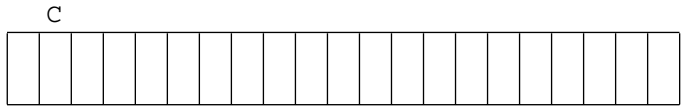


Figure 4.13 Character array

This array can store a sequence of characters up to 20 characters long. But, it can also store shorter sequences. For example, `char` array `C` could store at some point in a program either the sequence `Hello` or the sequence `Merry Christmas`, as both are shorter than 20 characters. Since, the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence, i.e. the *null character*, whose literal constant can be written as '`\0`' (backslash, zero). An array `C` of 20 elements of type `char`, can be represented by storing the character sequences "`Hello`" and "`Merry Christmas`" as shown in Figure 4.14.

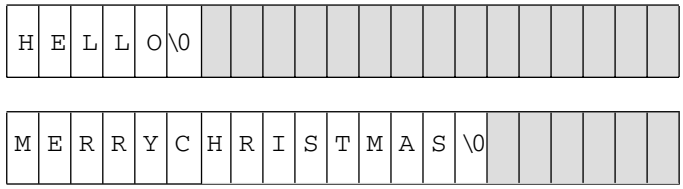


Figure 4.14 Null character at the end of the string

Notice how after the valid content a null character ('`\0`') has been included in order to indicate the end of the sequence. The panels in gray color represent `char` elements with undetermined values.

Initialization of null-terminated character sequences: Because arrays of characters are ordinary arrays, they follow all their same rules. For example, to initialize an array of characters with some predetermined sequence of characters, it can be done just like any other array:

```
char myword[]={'H', 'e', 'l', 'l', 'o', '\0'};
```

In this case an array is declared with 6 elements of type `char` initialized with the characters that form the word "`Hello`" plus a null character '`\0`' at the end. But arrays of `char` elements have an additional method to initialize their values using string literals.

In the expressions, entire strings of characters are specified enclosing the text to become a string literal between double quotes (`"`). For example,

```
"the result is:"
```

is a constant string literal.

Double-quoted strings (`"`) are literal constants whose type is in fact a null-terminated array of characters. So, string literals enclosed between double quotes always have a null character ('`\0`') automatically appended at the end.

Therefore, the array of `char` elements called `myword` is initialized with a null-terminated sequence of characters by either one of these two methods.

```
char myword[]={'H', 'e', 'l', 'l', 'o', '\0'};

char myword[]="Hello";
```

In both cases the array of characters `myword` is declared with a size of 6 elements of type `char` the 5 characters that compose the word "`Hello`" plus a final null character ('`\0`'), which specifies the end of the sequence and that, in the second case, when using double quotes (`"`) it is appended automatically.

Assuming `mystext` is a `char[]` variable, expressions within a source code like

```
mystext="Hello";
mystext[]="Hello";
```

would *not* be valid. Similarly, neither would be

```
mystext={'H', 'e', 'l', 'l', 'o', '\0'};
```

The reason for this may become more comprehensible once a bit more is known about pointers. Therefore, it will be clarified that an array is in fact a constant pointer pointing to a block of memory.

Using null-terminated sequences of characters: Null-terminated sequences of characters are the natural way of treating strings in C++. For example, `cin` and `cout` support null-terminated sequences as valid containers for a sequence of characters. So, they can be used directly to extract strings of characters from `cin` or to insert them into `cout`.

Program 4.3

```
/* null-terminated sequences of characters*/
#include<iostream>
using namespace std;
int main()
   {
      char question[]="Please, enter your first name:";
      char greeting[]="Hello,";
      char yourname[80];
      cout<<question;
      cin>>yourname;
```

```
        cout<<greeting<<yourname<<"!";
    return 0;
}
```

Output

```
Please, enter your first name:John
Hello, John!
```

In Program 4.3, three arrays are declared of `char` type. The first two were initialized with string literal constants, while the third one was left uninitialized. The size was implicitly defined by the length of the literal constant they were initialized to. While for `yourname` it is explicitly specified that it has a size of 80 `chars`. Finally, sequences of characters stored in `char` arrays can easily be converted into string objects just by using the assignment operator.

```
String mystring;
  char myntcs[]="some text";
  mystring=myntcs
```

4.9 APPLICATIONS

Arrays have a wide range of applications. Some of them are listed here and the sparse matrix application is detailed.

- Matrices
- Sparse matrix
- Records
- Ordered list

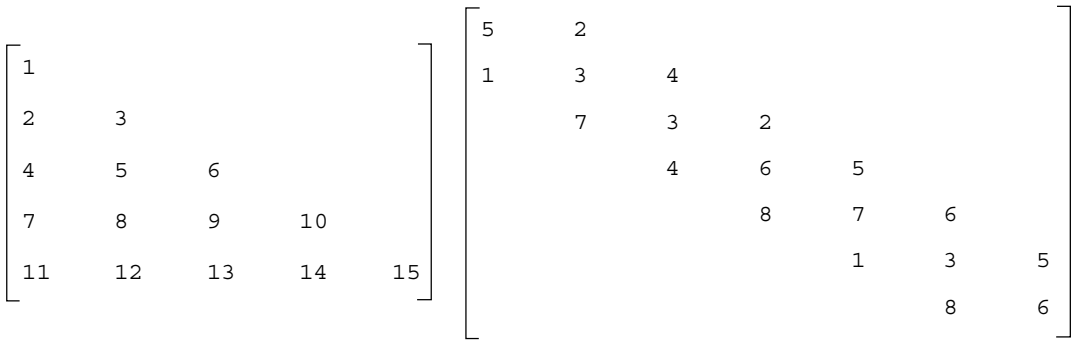
Sparse matrix: A matrix is a representation of elements in rows and columns. Sparse matrix is a matrix with a high proportion of zero entries. Figure 4.15 illustrates matrix and sparse matrix.

$$\begin{bmatrix} 3 & 2 & 1 & 5 \\ 1 & 0 & 4 & 2 \\ 0 & 6 & 5 & 6 \\ 2 & 1 & 0 & 5 \end{bmatrix} \qquad \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

(a) Matrix (b) Sparse matrix

Figure 4.15 Examples of matrix and sparse matrix

The most commonly used *n* square sparse matrices in various applications are shown in the Figure 4.16.



(a) Triangular matrix (b) Tridiagonal matrix

Figure 4.16 Sparse matrices

In Figure 4.16(a), all the entries above the main diagonal are zero or equivalent and all non-zero entries can occur only on or below the diagonal. This is known as the lower matrix or triangular matrix.

In Figure 4.16(b), all the non-zero entries occur on the diagonal or on the elements above the diagonal. This is known as the tridiagonal matrix.

A matrix requires lot of space in memory. A two-dimensional array is used to represent matrices. Thus, a 1000×1000 matrix requires huge storage locations in memory. If it is a sparse matrix, to store non-zero elements a huge amount of memory will be wasted. A two-dimensional array may not be suitable for sparse matrices.

Suppose to store a triangular matrix in memory, it is clear that storing all entries above the diagonal wastes the memory since they are all zeros. In such a case using a triple representation (`i,j,value`) can save the memory to represent non-zero elements of sparse matrix.

The ADT array: An array is a particular method of storing elements of indexed data. Elements of data are stored sequentially in blocks within the array. Each element is referenced by an index or subscript.

An array supports two operations. They are store and retrieve:

Store: Writing values into an array

Retrieve: Reading from an array

SUMMARY

- An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.
- An array can either be one dimensional or multi-dimensional.
- An ADT array performs mainly two operations—store and retrieve—and other operations include deletion, sort and search of elements.

EXERCISES

Fill in the Blanks

1. An array is a collection of _____ .
2. The character array ends with _____ .

3. Arrays are _____ type of structures.
4. _____ are the operations on arrays
5. _____ is an application of array.

Multiple-choice Questions

1. Array is of _____.
   a. Scalar type b. Aggregate type
   c. Union type d. Enumerated type
2. Array name is a _____.
   a. Variable b. Pointer to constant
   c. Constant pointer d. Constant
3. The only operator that can act on the array as a whole is _____.
   a. Indirection operator b. Size of operator
   c. { } d. [ ]
4. Which of the following is true about an array?
   a. Array elements are stored in contiguous memory locations
   b. In C, arrays are stored in row major order
   c. An array can be declared as local as well as global
   d. All the above

Short-answer Questions

1. What is the highest legal index for the following arrays?
   a. int arr1[4]; b. double means[12];
2. Which of the following identifiers are declared as arrays?
   a. int i, j[4], k; b. float r[3],s[2],t,u[7];
3. What is the highest legal index for the following arrays?
   a. int arr1[4]; b. double means[12];
4. Define character sequences.
5. Explain the representation of arrays.

Essay Questions

1. Write a C++ program to find the transpose of an array.
2. Write ADT operations for array implementation of polynomial addition.
3. Map elements of a two-dimensional array beginning with the right column and within a column from top to bottom.
   a. List the indexes of `a[4][4]` in this order.
   b. Develop the mapping function for `a[m][n]`.
4. Define an array. With an example, describe how you declare, insert an element, delete an element and display the array. Write a C++ program demonstrating the array.
5. Define ADT. Explain array ADT. Describe multi-dimensional arrays with an example and program in C++.

Chapter 5

Linked List

Chapter 5 discusses and exemplifies the data structure called linked list. The static and dynamic representations of linked list are clearly explained. A rigorous discussion of singly linked lists, circular linked lists and doubly linked lists along with their representation and operations is given. The chapter also compares various types of linked list along with arrays. This chapter continues with advantages, disadvantages, applications and ADT, i.e. Abstract Data Type of Linked Lists.

5.1 INTRODUCTION

Arrays are linear data structure for which memory should be allocated in advance to maintain the adjacency of the data elements. Once the memory is allocated, it cannot be extended as per the need. So, arrays are static data structures.

The implementation of insertion and deletion operations and storage of memory are not efficiently performed using arrays.

Linked list overcomes the pitfalls of arrays; linked lists are also known as dynamic data structures because allocation of memory will be done at the runtime, which can be extended as per the need. A linked list is also a linear collection of data elements called nodes, and the adjacency of elements is provided by the links or pointers. The general structure of the node is shown in Figure 5.1.

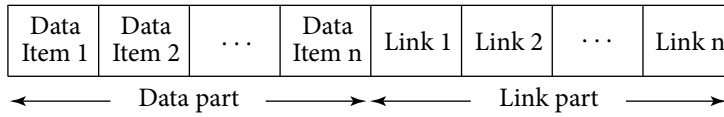


Figure 5.1 General structure of a node

Linked list does not store the nodes in contiguous locations as do the arrays, whereas the link field in the node maintains the address of its neighbour.

5.2 REPRESENTATION OF LINKED LIST IN MEMORY

Linked lists can be represented in two ways:

1. Static representation using arrays
2. Dynamic representation using free storage list

5.2.1 Static Representation

The linked list is maintained by two linear arrays—one is used for data and the other for links. Let *DATA* and *LINK* be the two arrays, DATA contains the information part, and their corresponding pointers to the next node are stored in the array LINK. A pointer variable *HEAD* is used to store the first location of the linked list, and a Null pointer is used to denote the end of the list. Since array subscripts are positive integers, Null is represented by '0'. Figure 5.2 illustrates the static representation of a linked list.

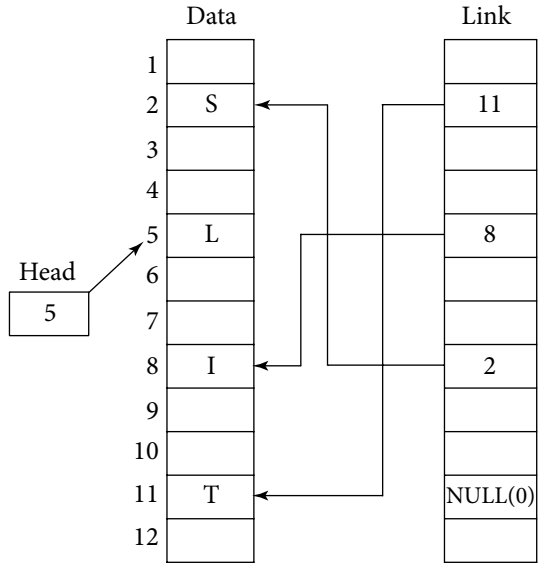


Figure 5.2 Static representation

In Figure 5.2, each node of the list contains a single character. The required string can be obtained as follows:

HEAD = 5, so DATA[5] = L is the first character. The corresponding link of DATA[5] is stored in LINK[5].

LINK[5] = 8 shows that 8th element of DATA that holds the next element, I, i.e. DATA[8] = I, the second character of the string.

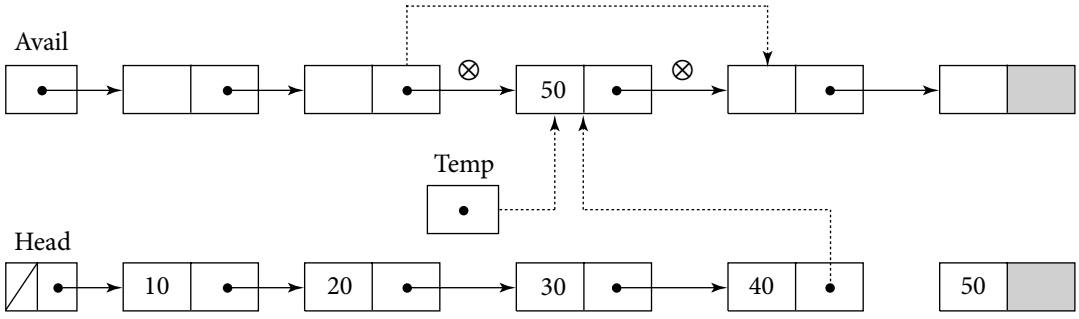
LINK[8] = 2 shows that 2nd element of DATA that holds element S, i.e. DATA[2] = S, the third character of the string.

LINK[2] = 1 shows that 11th element of DATA will hold element T, i.e. DATA[11] = T, the fourth character of the string.

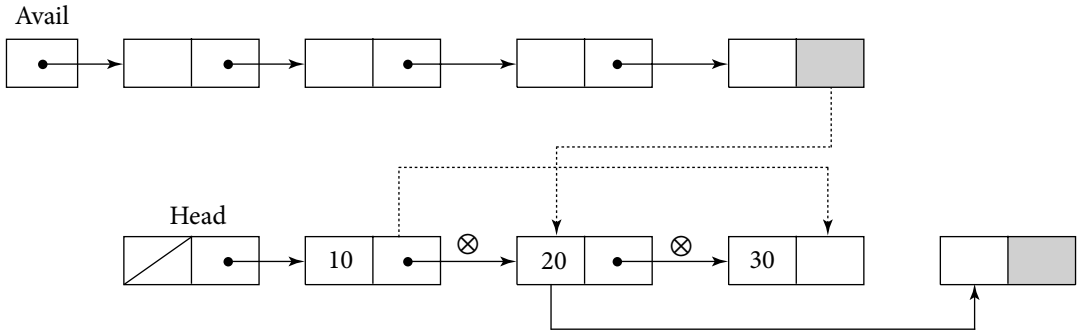
LIST is the required string.

5.2.2 Dynamic Representation

In this representation, a memory bank that is a collection of free memory space and memory manager program is used. When a new node is to be created, the memory management searches the memory bank for the required memory and if it is found, allocates memory to the new node and whenever a node is of no use, a program called garbage collector is invoked to return the unused node to the memory bank. This type of memory management is called Dynamic Memory Management. To represent linked list dynamically, a special list called AVAIL, a list of available memory spaces that has its own pointer, is maintained. Figures 5.3(a) and (b) illustrate the dynamic representation of a linked list.



(a) Allocating a new node to the list



(b) Node returned to the memory bank

Figure 5.3 Dynamic representation

In Figure 5.3(a), a new node is taken from the AVAIL and temporarily holds the address of the new node in the variable. The new node is then added to the existing list. The dotted arrow shows the insertion of the new node to the list and the symbol '⊕' is used to represent the deletion of links. The head node in the list does not contain any data. In Figure 5.3(b), deletion of the new node from the existing list is done and it is returned to the memory bank, i.e. AVAIL. The LINK field of the last node in the AVAIL will be pointing to the deleted node.

Linked lists are classified into three types regardless of the number of data fields:

- Singly linked list
- Doubly linked list
- Circular linked list

5.3 SINGLY LINKED LIST

A singly linked list is a linear data structure in which each node contains only one link field. Figure 5.4 illustrates the singly linked list.

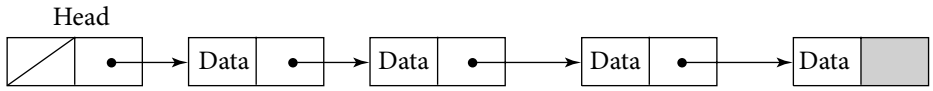


Figure 5.4 Singly linked list

HEAD pointer contains the address of the first node. The representation of the singly linked list is similar to the representation explained in Section 5.2.

5.3.1 Operations

The operations performed on a singly linked list are as follows:

- Traversing
- Searching
- Insertion
- Deletion

To perform insertion and deletion, the functions GETNODE(N) and RETURN(N) are used. GETNODE(N) is used to allocate memory for node N and RETURN (N) is used to return the unused node to memory bank.

5.3.1.1 Traversing

Traversing the list implies visiting each and every node in the list from the first node to the last node only once.

Algorithm 5.1: SL-TRAVERSE

```
1. Set PTR with the current pointer node.
2. Repeat steps 3 and 4.
3. VISIT the current node.
4. Set PTR with pointer to next node.
5. End.
```

5.3.1.2 Searching

Searching operation is used to find the value of DATA field in the singly linked list, while inserting an item into the list at a specified location and deleting an item at the specified location from list. The search operation is performed to locate the node with the DATA item, is to be deleted and to insert a node after that node and to display that particular node.

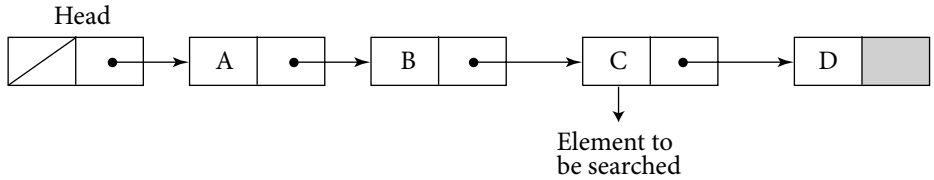


Figure 5.5 Searching for an element

In Figure 5.5, the node with DATA as C is to be searched. Let VALUE be assigned with C and starting from the head node search for the VALUE by comparing each node's DATA with it. When it is found, display it else display the message as VALUE is not found.

Algorithm 5.2: SL-SEARCH

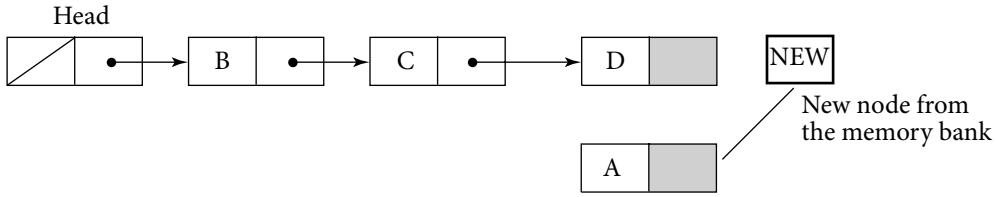
```
1. Set PTR to HEAD.                        //HEAD is a pointer to head node
2. Repeat Steps 3 to 5 until PTR is NULL.
3. Check whether DATA of PTR is VALUE then
     Write "Search is Successful".
4. Return (PTR).
5. Else Write "VALUE not found".
7. End.
```

5.3.1.3 Insertion

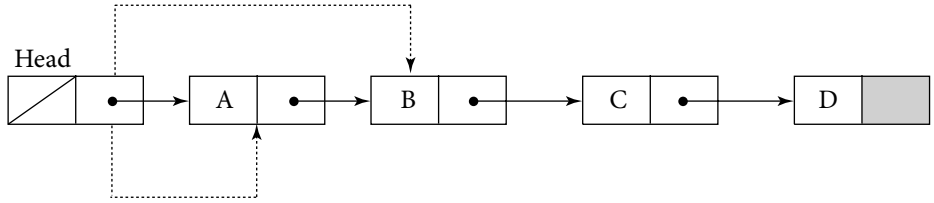
Insertion is an operation to add a node into the list. In a singly linked list, a node can be inserted at three different locations.

- Insertion as first node
- Insertion as last node
- Insertion at the specified position

Insertion as first node:



(a) Before insertion



(b) After insertion

Figure 5.6 Insertion as first node

In Figure 5.6(b), the node with data A is inserted as first node of the list and now the HEAD pointer points to the node A, previously which was pointing to the node B and the LINK of node A points to the node B.

Algorithm 5.3: SL-INSERT-F

```
1. call GETNODE(N) to allocate memory for node N and return the pointer of N.
2. Set DATA of N to ITEM.
```

```
3. Set LINK of N to LINK of HEAD.
4. Set LINK of HEAD to N.
5. End.
```

Insertion as last node:

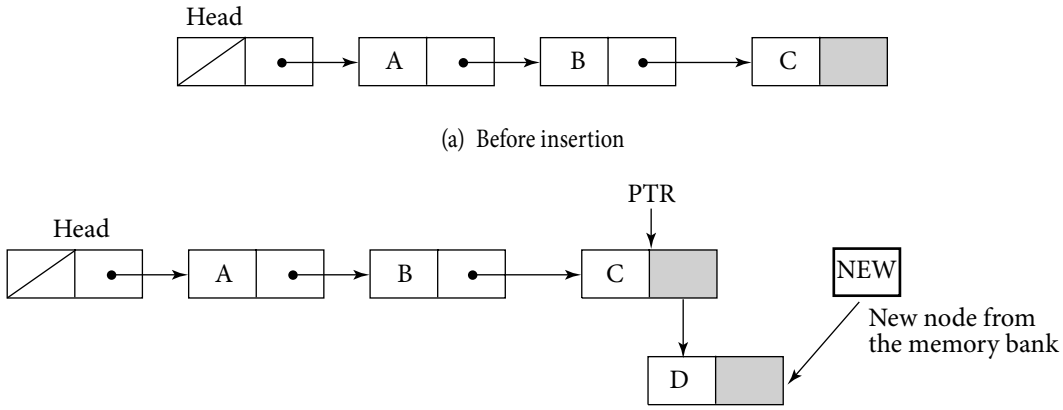


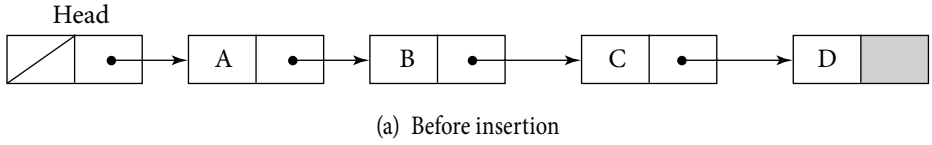
Figure: 5.7 Insertion as last node

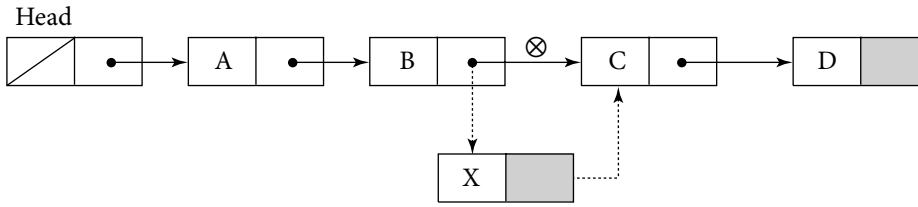
In Figure 5.7(b), the new node with DATA as D is inserted as last node of the list. The previous last node's LINK field, which was NULL, now points to the new node.

Algorithm 5.4: SL-INSERT–E

```
1. call GETNODE(N) to allocate memory for N and return the points of N.
2. Set PTR to HEAD.                          //HEAD is pointer to head node.
3. Loop to move to the end of the list and insert.
4. Set PTR to LINK of PTR.
5. End loop.
6. Set LINK of PTR to N.
7. Set DATA of N to ITEM.
8. End.
```

Insertion at the specified location: Search for the node whose DATA is VALUE; if it is found insert the node after that node.





(b) After insertion

Figure 5.8 Inspection at specified location

In Figure 5.8(b), the new node with DATA as X is inserted between the nodes B and C. The LINK field of B is pointing to the new node and the link of the new node points to the node C. Let VALUE be the DATA of the node after which the node is inserted. Here B is the key node after which the new node has to the inserted.

Algorithm 5.5: SL-INSERT-SP

```
1. Call GETNODE(N) to allocate memory for node N and return the pointer of N.
2. Start from the HEAD node i.e., PTR=HEAD. //HEAD is the pointer to the head
   node
3. Loop to find the node with DATA field as VALUE or move to end of list if VALUE
   is not found.
4. Set PTR to LINK of PTR.
5. End Loop.
6. Check whether LINK of PTR is NULL then write "VALUE is not found".
7. Else
   Set LINK of N to LINK of PTR.
   Set DATA of N to ITEM.
   Set LINK of PTR to N.
8. End.
```

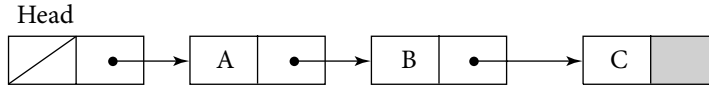
5.3.1.4 Deletion

The deletion operation is used to delete a node from the list, and it can be performed at three different locations.

- Deletion of the first node
- Deletion of the last node
- Deletion at the specified position

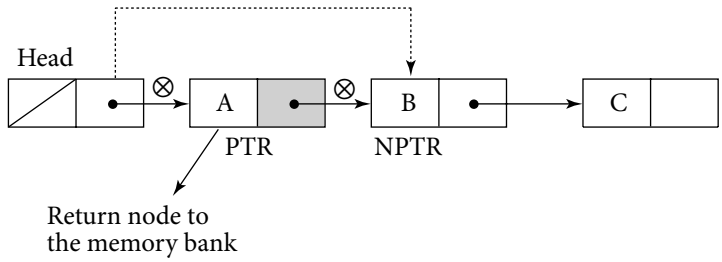
When a NODE is deleted then its memory must be returned to the memory bank. The function RETURN(N) returns the deleted node to the free pool storage space.

Deletion of the first node:



(a) Before deletion

Figure 5.9 Deletion of the first node



(b) After deletion

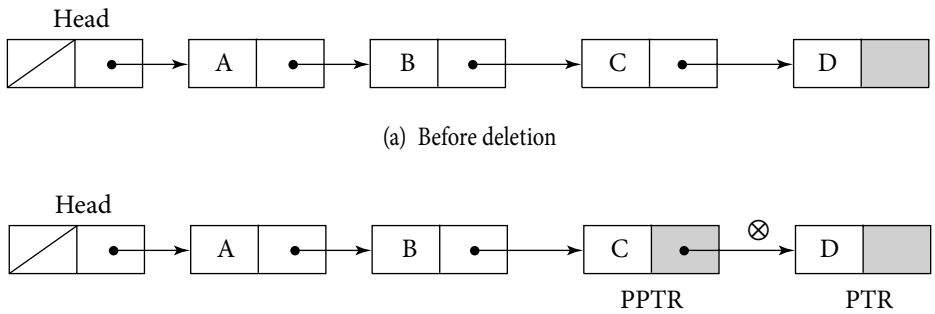
Figure 5.9 *Continued*

In Figure 5.9(b), the first node of the list is deleted and the head node's LINK field pointing to the deleted node now points to the next node. The LINK field of the node to be deleted is set to NULL.

Algorithm 5.6: SL-DELETION–F

```
1. Set PTR to LLINK of HEAD
2. Check whether list is empty else
3. Set NPTR to LINK of PTR              //NPTR is next node pointer
4. Set LINK of HEAD to PTR              //making next node first node now
5. RETURN (PTR)
6. End.
```

Deletion of the last node:



(a) Before deletion

(b) After deletion

Figure 5.10 Deletion of the last node

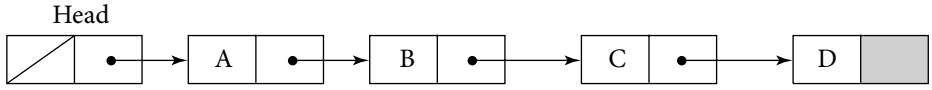
In Figure 5.10(b), the node that is at end of the list is deleted and the previous node's LINK, which was pointing to the last node is set to NULL, that is, deleting the link to the next node and return the unused node to the memory bank.

Algorithm 5.7: SL-DELETION-E

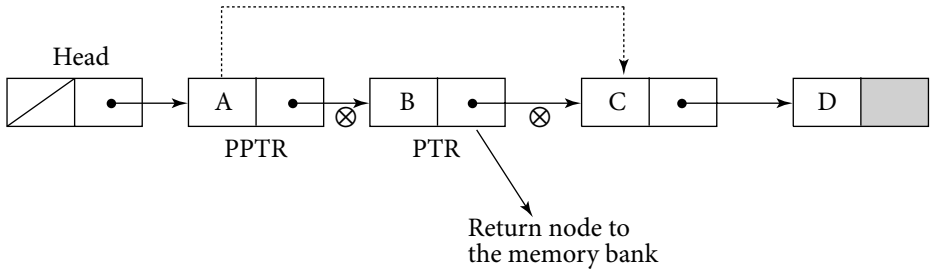
```
1. PTR=HEAD                          //HEAD is pointer to the head node
2. Check whether list is empty else
```

```
3. Loop to move to the end of list
    Set PPTR to PTR              //PPTR is the previous node pointer, and
                                 //PTR is last node pointer
4. End loop
5. Set LINK of PPTR to NULL      //making previous node pointer NULL
6. RETURN(PTR)
7. End.
```

Deletion at the specified position: Let VALUE be the DATA of the node, which is to be deleted. In Figure 5.11(b), the VALUE=B is searched in the list, if it matches with any one of the DATA of the nodes in the list then it will be deleted. Here, the node with DATA as B is deleted and LINK of B pointing to the next node is deleted and node A, which was earlier pointing to B now points to node C.



(a) Before deletion



(b) After deletion

Figure 5.11 Deletion at the specified location

Algorithm 5.8: SL-DELETION-SP

```
1. Start from head node that is PPTR=Head
2. Set PTR with link of Head that is pointing to the first node.
3.  Loop till PTR is NULL
4. Check whether DATA of PTR is equal to VALUE then
5. Set LINK of PPTR with LINK of PTR
6. Else Set PPTR with PTR
   Set LINK of PTR with PTR
7. Return(PTR)
8. End loop
9. Check whether PTR is NULL then Write "VALUE not found and deletion not pos-
   sible"
10. End.
```

Program 5.1

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>

template<class t>
class llist
{
   struct node
   {
      t data;
      node*link;
   }*next,*head,*temp,*temp1;
public:
llist();
void insertf();
void inserte();
void inserta();
void deletef();
void deletee();
void deletea();
void display();
};

template<class t>
llist<t>::llist()
{
   head->link=NULL;
}
template<class t>
void llist<t>::insertf()
{
   t ele;
   cout<<"enter the element\n";
   cin>>ele;
   temp=(struct node*)malloc(sizeof(struct node));
   if(temp==NULL)
   {
      cout<<"memory allocation error\n";
      return;
   }
   if(head->link==NULL)
   {
      temp->data=ele;
      head->link=temp;
      temp->link=NULL;
   }
```

```
   else
   {
      temp->link=head->link;
      head->link=temp;
      temp->data=ele;
   }
}
template<class t>
void llist<t>::inserte()
{
   t ele;
   cout<<"enter the element\n";
   cin>>ele;
   temp=(struct node*)malloc(sizeof(struct node));
   if(temp==NULL)
   {
      cout<<"memory allocation error\n";
      return;
   }
   if(head->link==NULL)
   {
      temp->data=ele;
      head->link=temp;
      temp->link=NULL;
   }
   else
   {
      for(temp1=head->link;temp1->link!=NULL;temp1=temp1->link)
      {
      }
   temp1->link=temp;
   temp->data=ele;
   temp->link=NULL;
   }
}
template<class t>
void llist<t>::inserta()
{
   t ele;
   int pos,i;
   cout<<"enter the element\n";
   cin>>ele;
   cout<<"enter the position\n";
   cin>>pos;
   temp=(struct node*)malloc(sizeof(struct node));
   if(temp==NULL)
   {
```

```
      cout<<"memory allocation error\n";
      return;
   }
   if(head->link==NULL)
   {
      temp->data=ele;
      head->link=temp;
      temp->link=NULL;
   }
   else
   {
      for(temp1=head,i=1;i<pos;i++)
      {
         temp1=temp1->link;
      }
   temp->link=temp1->link;
   temp1->link=temp;
   temp->data=ele;
   }
}
template<class t>
void llist<t>::deletef()
{
   t ele;
   if(head->link==NULL)
   {
      cout<<"linked list is empty\n";
      return;
   }
   temp=head->link;
   temp1=temp->link;
   head->link=temp1;
   ele=temp->data;
   free(temp);
   cout<<"the deleted element is"<<ele<<endl;
}
template<class t>
void llist<t>::deletee()
{
   t ele;
   if(head->link==NULL)
   {
      cout<<"linked list is empty\n";
      return;
   }
   for(temp1=head;temp1->link!=NULL;temp1=temp1->link)
   {
```

```
      temp=temp1;
   }
   temp->link=NULL;
   ele=temp1->data;
   free(temp1);
   cout<<"the deleted element is"<<ele<<endl;
}
template<class t>
void llist<t>::deletea()
{
   t ele;
   int pos,i;
   if(head->link==NULL)
   {
      cout<<"linked list is empty\n";
      return;
   }
   cout<<"enter the position\n";
   cin>>pos;
   for(temp1=head,i=0;i<pos;i++,temp1=temp1->link)
   {
      temp=temp1;
   }
   temp->link=temp1->link;
   ele=temp1->data;
   cout<<"the deleted element is"<<ele<<endl;
   free(temp1);
}
template<class t>
void llist<t>::display()
   {
      if(head->link==NULL)
      {
         cout<<"linked list is empty\n";
         return;
      }
   for(temp=head->link;temp!=NULL;temp=temp->link)
   {
      cout<<temp->data<<" ";
   }
}
void main()
{
   int ch;
   llist <int> l;
   clrscr();
   while(1)
   {
```

```
        cout<<"\nmenu\n";
        cout<<"1.insertion at front\n";
        cout<<"2.insertion at end\n";
        cout<<"3.insertion at any position\n";
        cout<<"4.deletion at front\n";
        cout<<"5.deletion at end\n";
        cout<<"6.deletion at any position\n";
        cout<<"7.display\n";
        cout<<"8.exit\n";
        cout<<"enter your choice\n";
        cin>>ch;
        switch(ch)
        {
           case 1:
              l.insertf();
              break;
           case 2:
              l.inserte();
              break;
           case 3:
              l.inserta();
              break;
           case 4:
              l.deletef();
              break;
           case 5:
              l.deletee();
              break;
           case 6:
              l.deletea();
              break;
           case 7:
              l.display();
              break;
           case 8:
              exit(0);
              break;
        }
     }
  }
```

Output

```
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
```

```
6.deletion at any position
7.display
8.exit
enter your choice
1
enter the element
10
menu
1. insertion at front
2. insertion at end
3. insertion at any position
4. deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
1
enter the element
20
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
20 10
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
2
enter the element
30
menu
1.insertion at front
2.insertion at end
```

```
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
2
enter the element
40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
20 10 30 40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
3
enter the element
50
enter the position
3
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
```

```
7
20 10 50 30 40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
4
the deleted element is 20
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
10 50 30 40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
5
the deleted element is 40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
```

```
7
10 50 30
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
6
enter the  position
2
the deleted element is 50
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
10 30
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice 8
```

5.4 CIRCULAR LINKED LIST

A singly linked list in which the link field of the last node is assigned with the address of the first node is known as a circular linked list or a circular list. Figure 5.12 illustrates the representation of a circular linked list.

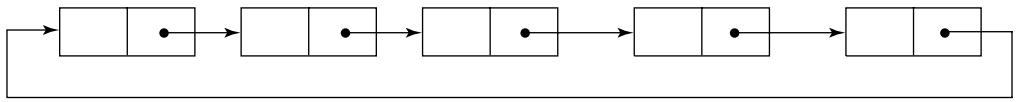
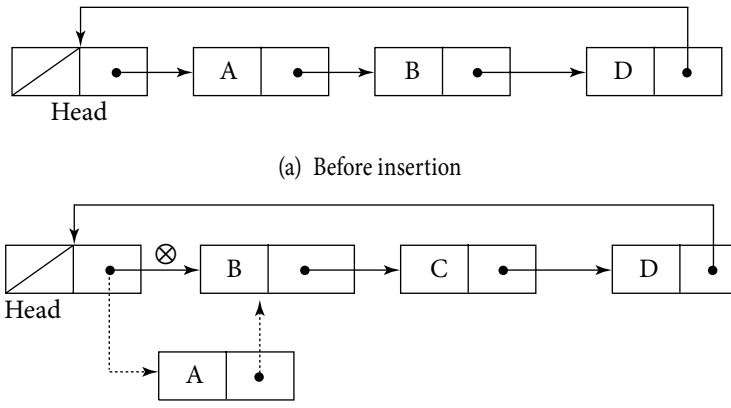


Figure 5.12 Circular linked list

Here, the LINK field of the last node is pointing to the first node. Using the circular linked list any node from any position can be accessed without the need of starting from the first node. List based implementations such as concatenation of lists, splitting of lists and so on can be efficiently performed with circular lists.

One disadvantage with circular lists is that while processing, the list makes sure that it does not form an infinite loop since the direction of pointers in the list is circular. This can be solved with the HEAD node, which is the Header node in the list. It helps us to point out the end of the list and terminate the loop.

The operations insertion into and deletion from the circular linked lists are the same as in a singly linked list except that the last node points to the first node. So, when inserting the first node, update the head node pointer and LINK field of it to point to the first node. When deleting the last node, update the previous node of the last node to point to the head node.

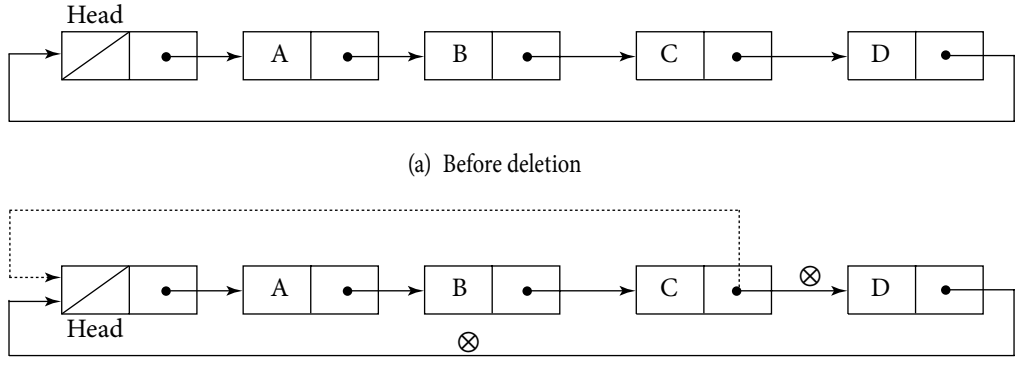


(a) Before insertion

(b) After insertion

Figure 5.13 Insertion in a circular linked list

In Figure 5.13(b), the node A is inserted between the head node and node B. First obtain the value of the head node LINK field and make the new node pointer point to the node B. Now assign the head node pointer with the address of the new node, i.e. head node pointing to the new node.



(a) Before deletion

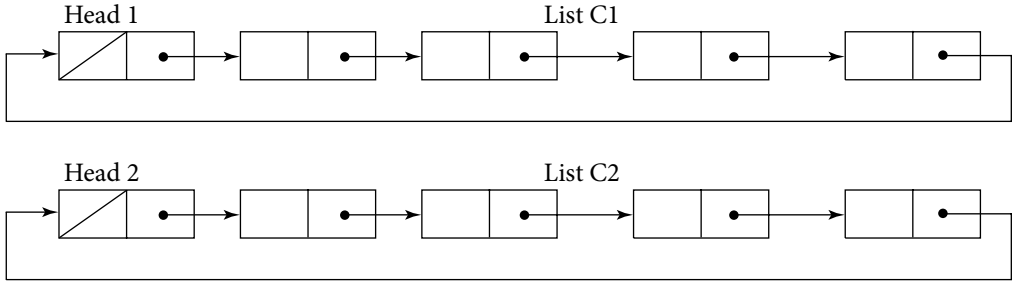
(b) After deletion

Figure 5.14 Deletion of the node in a circular linked list

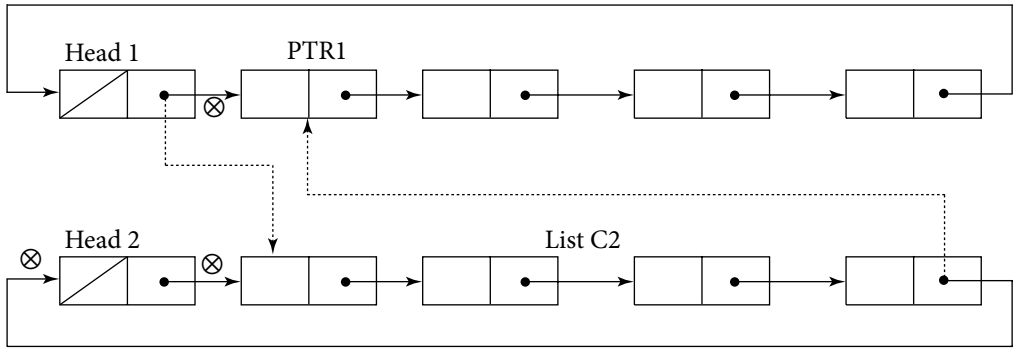
In Figure 5.14(b), the last node D is deleted first getting the address of the node containing D and its predecessor node C. Now, LINK of node C points to node D which is deleted and makes it to point to the head node since it is a last node now, which is shown by a dashed line in Figure 5.14. Return the unused node to the memory bank.

5.4.1 Merging of Two Circular Lists

Let C1 and C2 be the two circular linked lists to be merged and HEAD1 and HEAD2 be the header nodes of C1 and C2, respectively. The LINK field of HEAD1 node pointing to the first node of C1 is deleted and it is pointed to the first node of list C2, and the last node of list C2 pointing to HEAD2 is deleted and it is pointed to the first node of C1; this is illustrated in Figure 5.15.



(a) Before merging two lists, C1 and C2



(b) After merging C1 and C2

Figure 5.15 Merging two circular linked lists

Algorithm 5.9: CL-CONC

```
1. Set PTR1 to LINK of HEAD1
2. Set  PTR2 to LINK of HEAD2
3. Set LINK of HEAD1 to PTR2
4. Loop till LINK of PTR2 is equal to HEAD2
5. Set PTR2 to LINK of PTR2
6. End loop
```

```
7. Set LINK of PTR2 to PTR1
8. Return (HEAD2)                    //return HEAD2 to the memory bank.
9. End.
```

5.5 DOUBLY LINKED LISTS

In a singly linked list, moving from one node to another node is done in only one direction and it is known as a one-way list.

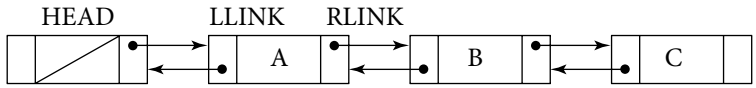
A doubly linked list is the one in which each node of it contains two links pointing to either side of the nodes. The movement in the doubly linked list is either of the direction i.e. either from left to right or right to left. Figure 5.16 shows the structure of a node in a doubly linked list.

Figure 5.16 Structure of the node

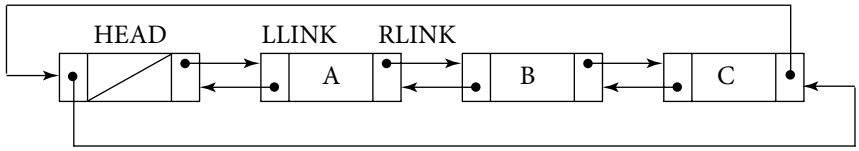
5.5.1 Representation of Doubly Linked List

A doubly linked list is a linear data structure in which each node has two links called the left link (LLINK) and the right link (RLINK).

The LLINK field of a node points to the node to its left, and the RLINK points to the node to its right. A doubly linked list can also be a circular list and it is known as a circular doubly linked list.



(a) Doubly linked list



(b) Circular doubly linked list

Figure 5.17 Types of a doubly linked list

In Figure 5.17(a), each node has two links, except the first and the last node. The RLINK of the last node is NULL and LLINK of the head node is NULL.

Figure 5.17(b) shows a circular doubly linked list. Here, the RLINK of the last node points to the head node and LLINK of the head node points to the last node.

5.5.2 Operations

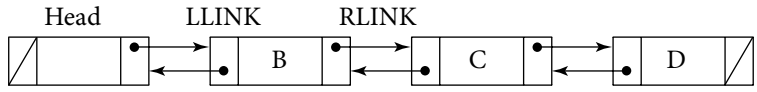
All the operations performed on a singly linked list can be implemented efficiently using a doubly linked list.

5.5.2.1 Insertion

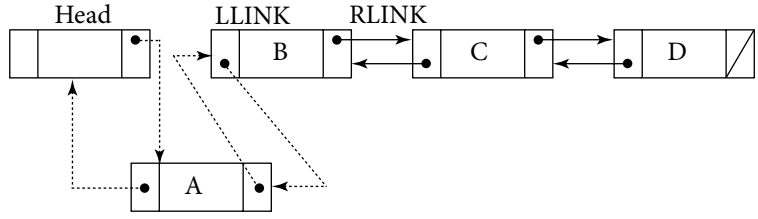
This operation can be performed in three ways:

1. Insertion as first node
2. Insertion as last node
3. Insertion at a specified position

Insertion as first node:



(a) Before insertion



(b) After insertion

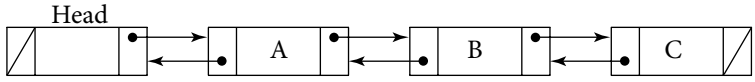
Figure 5.18 Insertion operation in a doubly linked list

Algorithm 5.10: DL_INSERTION_F

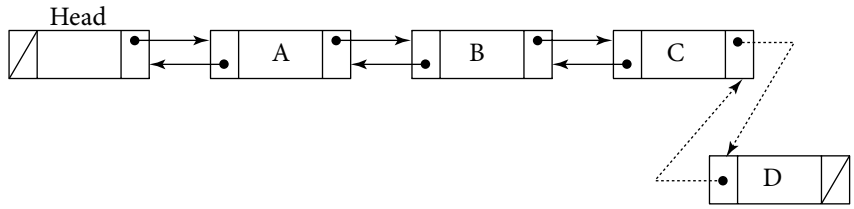
```
1. Set  PTR to point to first node
2. Call GETNODE(N) to allocate memory for new node N.
3. Set LLINK of N to point to HEAD
4. Set RLINK of HEAD to point to N
5. Set RLINK of N with PTR
6. Set LLINK of PTR with N
7. Assign ITEM to DATA of N
8. End
```

Algorithm 5.10 inserts the node as the first node of the list. The RLINK of the head node points to the new node and LLINK of the new node points to the head node, the RLINK of the new node points to the next node and LLINK of the next node points to the new node (Figure 5.18).

Insertion at the last node: To insert the node as the last node of the list, the last node in the list is traced and Set the RLINK of the last node to point to the new node, and LLINK of the new node to point to the last node. RLINK of the new node is set to NULL. So, the new node will be the last node (Figure 5.19; Algorithm 5.11).



(a) Before insertion



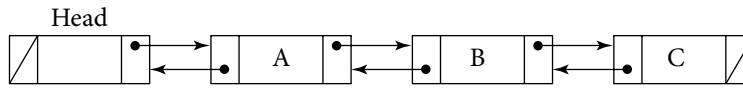
(b) After insertion

Figure 5.19 Insertion as the last node

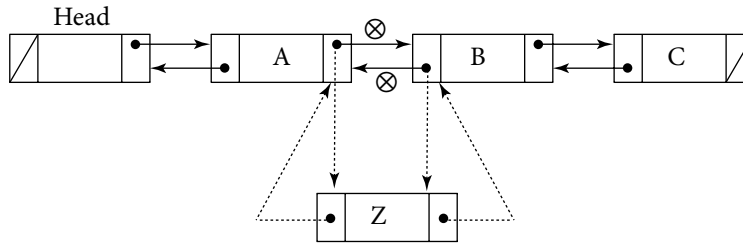
Algorithm 5.11: DL_INSERT_E

```
1. Set PTR to Head node pointer
2. Loop to move to last node
3. Set PTR to RLINK of PTR
4. End LOOP
5. Call GETNODE (N)                          //get new node
6. Set LLINK of N to PTR
7. Set RLINK of PTR to N
7. Set RLINK of N to NULL                   //make new node (N) last node
9. Assign ITEM to DATA of N
10. End
```

Insertion at a specified position: To insert the given node at a specified position, let a node with DATA field as ITEM be inserted after the node with DATA field as VALUE. So, search for the node whose DATA field is VALUE; if it is found, then insert the new node after that node (Figure 5.20; Algorithm 5.12).



(a) Before insertion



(b) After insertion

Figure 5.20 Insertion at a specified position

Algorithm 5.12: DL_INSERTION_SP

```
1. Set PTR to HEAD
2. Loop to find the node with VALUE
```

```
2. Set PTR to RLINK of PTR
3. End Loop
4. Call GETNODE(N)                  //get the new node
5. Set N PTR to RLINK of PTR        //N PTR is the pointer to next node after
                                    //node with VALUE
6. Set LLINK of N to PTR
7. Set RLINK of N to NPTR
8. Set RLINK of  PTR to N
9. Set LLINK of NPTR to N
10. Set PTR to N
11. Assign ITEM to DATA of N
12. End
```

5.5.2.2 Deletion

Like insertion, the deletion operation can also be performed in three ways:

1. Deletion of the first node
2. Deletion of the last node
3. Deletion at a specified position

Deletion of the first node: Deleting the first node in the list updates RLINK of HEAD to point to the node after the first node, that is, deleting the link between the HEAD node and the first node and LLINK of the node after the deleted node is updated to point to the HEAD node and then it becomes the first node (Figure 5.21; Algorithm 5.13).

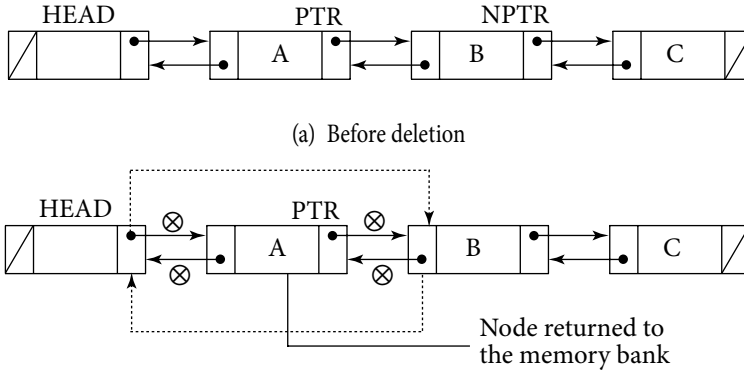
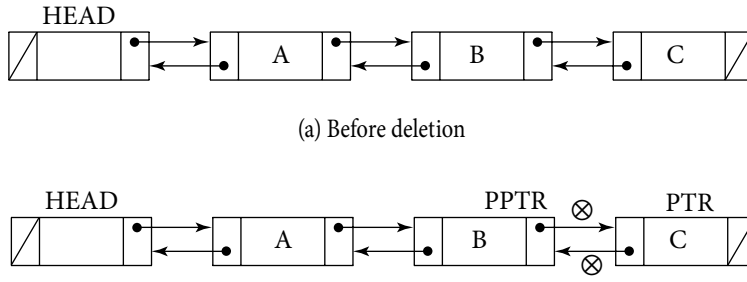


Figure 5.21 Deletion of the first node

Algorithm 5.13: DL_DELETION_F

```
1. Set PTR to point to the first node
2. Check whether list is empty then write "deletion cannot be done"
3. Set NPTR to RLINK of PTR                    //NPTR is point to next node
4. Set RLINK of HEAD to NPTR
5. Set LLINK of NPTR to HEAD
6. Return(PTR)
7. End
```

Deletion of the last node: To delete the last node in the list, let PTR point to the last node, and PPTR is the pointer to the previous node of the last node. Deleting the last node is deleting links between nodes pointed by the pointers PPTR and PTR and then making the RLINK of PPTR node to NULL. Now PPTR becomes the last node in the list. Figure 5.22 illustrates the deletion of the last node (Algorithm 5.14).



(a) Before deletion

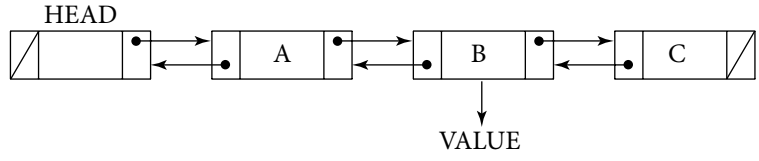
(b) After deletion

Figure 5.22 Deletion of the last node

Algorithm 5.14: DL_DELETION_E

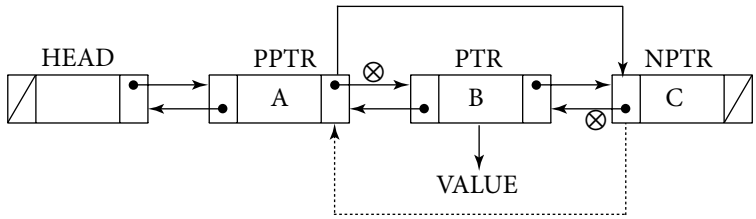
```
1. Set PTR to point head node
2. Loop to move to the last node
3. Set PTR to RLINK of PTR
4. End loop
5. Check whether list is empty then write" Deletion not possible"
6. Set PPTR to LLINK of PTR
7. Set RLINK of PPTR to NULL
8. Return(PTR)
9. End
```

Deletion at a specified position: To delete the node at any specified location, suppose VALUE is the DATA of the node to be deleted. Search for the VALUE in the list. Let PTR point to the node to be deleted and NPTR point to the next node to PTR, and PPTR points to the previous node of PTR. To delete PTR, update the LLINK of NPTR to point to the PPTR and RLINK of PPTR to point to NPTR. Figure 5.23 illustrates the deletion of the node at a specified position (Figure 5.23, Algorithm 5.15).



(a) Before deletion

Figure 5.23 Deletion at a specified position



(b) After deletion

Figure 5.23 *Continued*

Algorithm 5.15: DL_DELETION_SP

```
1. Set PTR to point to first node
2. Check whether list is empty then write" Deletion not possible"
3. Loop to move to the required node
4. Checking for the VALUE
5. Set PTR to RLINK  of PTR
6. End loop
7. When DATA of PTR is equal to VALUE
8. Set PPTR to LLINK of PTR
9. Set NPTR to RLINK of PTR
10. Set RLINK of PPTR to NPTR
11. Check whether deleted node is the last node then
12. Set LLINK of NPTR to NULL
13. Return(PTR)
14. End.
```

Program 5.2

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
template<class t>
class llist
{
   struct node
   {
      t data;
      node*flink,*blink;
      }*next,*head,*temp,*temp1;
   public:
   llist();
   void insertf();
   void inserte();
   void inserta();
   void deletef();
```

```
   void deletee();
   void deletea();
   void fdisplay();
   void bdisplay();
   };
   template<class t>
   llist<t>::llist()
   {
      head->flink=NULL;
      head->blink=NULL;
   }
   template<class t>
   void llist<t>::insertf()
   {
      t ele;
      cout<<"enter the element\n";
      cin>>ele;
      temp=(struct node*)malloc(sizeof(struct node));
      if(temp==NULL)
      {
         cout<<"memory allocation error\n";
         return;
      }
   if(head->flink==NULL)
   {
      temp->data=ele;
      head->flink=temp;
      temp->flink=NULL;
      temp->blink=head;
   }
   else
   {
      temp->flink=head->flink;
      temp->blink=head;
      head->flink=temp;
      temp->data=ele;
      temp1=temp->flink;
      temp1->blink=temp;
   }
}
template<class t>
void llist<t>::inserte()
{
   t ele;
   cout<<"enter the element\n";
   cin>>ele;
   temp=(struct node*)malloc(sizeof(struct node));
   if(temp==NULL)
```

```
    {
       cout<<"memory allocation error\n";
       return;
    }
    if(head->flink==NULL)
    {
       temp->data=ele;
       temp->blink=head;
       head->flink=temp;
       temp->flink=NULL;
    }
else
    {
       for(temp1=head->flink;temp1->flink!=NULL;temp1=temp1->flink)
    {
    temp1->flink=temp;
    temp->data=ele;
    temp->flink=NULL;
    temp->blink=temp1;
    }
}
template<class t>
void llist<t>::inserta()
{
    t ele;
    int pos,i;
    cout<<"enter the element\n";
    cin>>ele;
    cout<<"enter the position\n";
    cin>>pos;
    temp=(struct node*)malloc(sizeof(struct node));
    if(temp==NULL)
    {
       cout<<"memory allocation error\n";
       return;
    }
    if(head->flink==NULL)
    {
       temp->data=ele;
       head->flink=temp;
       temp->blink=head;
       temp->flink=NULL;
    }
    else
    {
       for(temp1=head,i=1;i<pos;i++)
       {
          temp1=temp1->flink;
```

```
      }
      temp->flink=temp1->flink;
      temp1->flink=temp;
      temp->data=ele;
      temp->blink=temp1;
      temp=temp->flink;
      temp->blink=temp;
   }
}
template<class t>
void llist<t>::deletef()
{
   t ele;
   if(head->flink==NULL)
   {
      cout<<"flinked list is empty\n";
      return;
   }
   temp=head->flink;
   temp1=temp->flink;
   head->flink=temp1;
   temp1->blink=head;
   ele=temp->data;
   free(temp);
   cout<<"the deleted element is"<<ele<<endl;
}
template<class t>
void llist<t>::deletee()
{
   t ele;
   if(head->flink==NULL)
   {
      cout<<"flinked list is empty\n";
      return;
   }
   for(temp1=head;temp1->flink!=NULL;temp1=temp1->flink)
   {
      temp=temp1;
   }
   temp->flink=NULL;
   ele=temp1->data;
   free(temp1);
   cout<<"the deleted element is"<<ele<<endl;
}
template<class t>
void llist<t>::deletea()
{
   t ele;
```

```
   int pos,i;
   if(head->flink==NULL)
   {
      cout<<"flinked list is empty\n";
      return;
   }
   cout<<"enter the position\n";
   cin>>pos;
   for(temp1=head,i=0;i<pos;i++,temp1=temp1->flink)
   {
      temp=temp1;
   }
   temp->flink=temp1->flink;
   temp=temp1->flink;
   temp->blink=temp1->blink;
   ele=temp1->data;
   cout<<"the deleted element is"<<ele<<endl;
   free(temp1);
}
template<class t>
void llist<t>::fdisplay()
{
   if(head->flink==NULL)
   {
      cout<<"flinked list is empty\n";
      return;
   }
   for(temp=head->flink;temp!=NULL;temp=temp->flink)
   {
      cout<<temp->data<< " ";
   }
}
   template<class t>
   void llist<t>::bdisplay()
   {
      if(head->flink==NULL)
      {
         cout<<"flinked list is empty\n";
         return;
}
for(temp=head;temp->flink!=NULL;temp=temp->flink)
{
   //cout<<temp->data<< "";
}
for(temp1=temp;temp1->blink!=NULL;temp1=temp1->blink)
   {
      cout<<temp1->data<<" ";
   }
```

```
}
void main()
{
   int ch;
   llist<char> l;
   clrscr();
   while(1)
   {
      cout<<"\nmenu\n";
      cout<<"1.insertion at front\n";
      cout<<"2.insertion at end\n";
      cout<<"3.insertion at any position\n";
      cout<<"4.deletion at front\n";
      cout<<"5.deletion at end\n";
      cout<<"6.deletion at any position\n";
      cout<<"7.fdisplay\n";
      cout<<"8.bdisplay\n";
      cout<<"9.exit\n";
      cout<<"enter your choice\n";
      cin>>ch;
      switch(ch)
   {
      case 1:
         l.insertf();
         break;
      case 2:
         l.inserte();
         break;
      case 3:
         l.inserta();
         break;
      case 4:
         l.deletef();
         break;
      case 5:
         l.deletee();
         break;
      case 6:
         l.deletea();
         break;
      case 7:
         l.fdisplay();
         break;
      case 8:
         l.bdisplay();
         break;
      case 9:
      exit(0);
```

```
        break;
        }
    }
}
```

Output

```
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
1
enter the element
2
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
1
enter the element
3
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
7
3 2
```

```
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
2
enter the element
5
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
2
enter the element
6
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
8
6 5 2 3
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
```

```
7.fdisplay
8.bdisplay
9.exit
enter your choice
3
enter the element
8
enter the position
2
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
7
3 8 2 5 6
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
4
the deleted element is 3
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
7
8 2 5 6
```

```
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
5
the deleted element is 6
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
7
8 2 5
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.fdisplay
8.bdisplay
9.exit
enter your choice
6
enter the position
2
the deleted element is 2
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
```

```
7.fdisplay
8.bdisplay
9.exit
enter your choice
9
```

5.6 COMPARISON OF VARIOUS LINKED LISTS

This section deals with a comparison of various types of linked lists.

5.6.1 Linked Lists Versus Arrays

Linked lists have several advantages over arrays. The insertion of an element at a specific point of a list is a constant time operation, whereas the insertion in an array requires half of the elements or more to be moved downwards. While deletion of an element from an array requires constant time by somehow marking its slot as "vacant", an algorithm that iterates over the elements may have to skip a large number of vacant slots.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while an array will eventually fill up, and then have to be resized—an expensive operation that may not even be possible if the memory is fragmented. Similarly, an array from which many elements are removed may have to be resized in order to avoid wasting too much space.

On the other hand, arrays allow random access, while linked lists allow only sequential access to elements. Singly linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it is useful to look up an element by its index quickly, such as heap sort. Sequential access on arrays is also faster than linked lists because they have greater locality of reference and thus profit more from processor caching.

Another disadvantage of linked lists is the additional storage needed for references, especially for lists of small data items such as characters or Boolean values. It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

Some hybrid solutions try to combine the advantages of the two representations. Unrolled linked lists store several elements in each list node, increasing cache performance while decreasing memory overhead for references.

A good example that highlights the pros and cons of using arrays versus linked lists is by implementing a program that resolves the Josephus problem. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle n times. Once the n^{th} person is reached, take them out of the circle and have the members close the circle. Then count around the circle the same n times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list over an array, because people are in a circular linked list; it shows how easily the linked list is able to delete nodes as it only has to rearrange the links to the different nodes. However, the linked list will be poor at finding the next person to remove and will need to recur through the list until it finds that person. An array, on the other hand, will be poor at deleting nodes (or elements) as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the nth person in the circle by directly referencing them by their position in the array.

The list ranking problem concerns the efficient conversion of a linked list representation into an array. Although trivial for a conventional computer, solving this problem by a parallel algorithm is complicated.

5.6.1.1 Singly Linked Lists Versus Other Lists

While doubly linked and/or circular lists have advantages over singly linked linear lists, the linear lists offer some advantages that make them preferable in some situations.

Many operations on singly linked linear lists (such as merging two lists or enumerating the elements in reverse order) often have very simple recursive algorithms, much simpler than any solution using iterative commands. While one can adapt those recursive solutions for doubly linked and circularly linked lists, the procedures generally need extra arguments and more complicated base cases.

Linear singly linked lists also allow tail-sharing, the use of a common final portion of sub-list as the terminal portion of two different lists. In particular, if a new node is added at the beginning of a list, the former list remains available as the tail of the new one—a simple example of a persistent data structure. Again, this is not true with the other variants: a node may never belong to two different linked lists.

In particular, end-sentinel nodes can be shared among singly linked non-circular lists. One may even use the same end-sentinel node for every such list. Indeed, the advantages of the fancy variants are often limited to the complexity of the algorithms, not in their efficiency. A circular list, in particular, can usually be emulated by a linear list together with two variables that point to the first and last nodes at no extra cost.

5.6.1.2 Doubly Linked Versus Singly Linked Lists

Doubly linked lists require more space per node (unless one uses xor-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node`s address. To do the same in a singly linked list, one must have the *previous* node`s address. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

5.6.1.3 Circular-linked Versus Linear l-linked Lists

A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. for the corners of a polygon, for a pool of buffers that are used and released in FIFO (FIRST-IN-FIRST-OUT) order, or for a set of processes that should be time-shared in round-robin order. In these applications, a pointer to any node serves as a handle to the whole list.

With a circular list, a pointer to the last node gives easy access also to the first node by following one link. Thus, in applications that require access to both ends of the list (e.g. in the implementation of a queue), a circular structure allows one to handle the structure by a single pointer instead of two.

A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes, nodes in two distinct lists, joins the two lists into one. This property greatly simplifies some algorithms and data structures, such as the quad-edge and face-edge.

5.7 APPLICATIONS

1. Sparse matrix representation
2. Polynomial manipulation
3. Dynamic memory storage
4. In symbol table

5.7.1 Polynomial Manipulation

A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term. However, for any polynomial operation, such as addition or multiplication of polynomials, the

linked list representation is easier to deal with. The structure of a node to represent a polynomial is shown in Figure 5.24.

Figure 5.24 Node structure of polynomial

In a polynomial all the terms may not be present, especially if it is going to be a very high-order polynomial. Consider $5x^{12} + 2x^9 + 4x^7 + 6x^5 + x^2 + 12x$, now this 12th order polynomial does not have all the 13 terms (including the constant term). It would be very easy to represent the polynomial using a linked list structure, where each node can hold information pertaining to a single term of the polynomial. Each node will need to store the variable x, the exponent and the coefficient for each term. It often does not matter whether the polynomial is in x or y. This information may not be very crucial for the intended operations on the polynomial. Thus, there is a need to define a node structure to hold two integers, viz. exponent and coefficient. Compare this representation with storing the same polynomial using an array structure.

In the array a slot for each exponent of x should be maintained; thus, if a polynomial of order 50 is to be arranged but containing just 6 terms, then a large number of entries will be 0 in the array. It is also easy to manipulate a pair of polynomials if they are represented using linked lists.

5.7.1.1 Addition of Two Polynomials

Consider the addition of the following polynomials:

$5x^{12} + 2x^9 + 4x^7 + 6x^6 + x^3$

$7x^8 + 2x^7 + 8x^6 + 6x^4 + 2x^2 + 3x + 40$

The resulting polynomial is going to be:

$5x^{12} + 2x^9 + 7x^8 + 6x^7 + 14x^6 + 6x^4 + x^3 + 2x^2 + 3x + 40$

Now notice how the addition is carried out. Let the result of addition be stored in a third list. Start with the highest power in any polynomial. If there is no item having the same exponent, simply append the term to the new list and continue with the process. Wherever matching of the exponents is found, then simply add the coefficients and then store the term in the new list. If one list gets exhausted earlier and the other list still contains some lower-order terms, then simply append the remaining terms to the new list. Algorithm 5.16 show how to add two polynomials.

Algorithm 5.16: ADD-POLY

Let phead1, phead2 and phead3 represent the pointers of the three lists and each node contains two integer's exponent and coefficient. Assume that the two linked lists already contain relevant data about the two polynomials. Also assume that a function *append()* to insert a new node at the end of the given list.

```
1. Set p1=phead1;
2. Set p2=phead2;
3. Call malloc to create a new node p3 to build the third list
4. Set p3=phead3;
5. Traverse the lists till one list gets exhausted
6. Loop if the exponent of p1 is higher than that of p2 then the next term
   in final list is going to be the node of p1
7. Set p3->exp=p1->exp;
8. Set p3->coff=p1->coff;
9. Append(p3, phead3);
```

```
11.Set p1=p1->next;   /*now move to the next term in list 1*/
12.End loop
13.Loop if p2 exponent turns out to be higher then make p3 same as p2 and
   append to final list
14.Set p3->exp=p2->exp;
15.Set p3->coff=p2->coff;
16.Append(p3, phead3);
17.Set p2=p2->next;
18.End loop
19.Loop if  both exponents are same and  then add the coefficients to get
    the term for the final
20.Setp3->exp=p1->exp;
21.Set p3->coff=p1->coff+p2->coff;
22.Append(p3, phead3);
23.Set p1=p1->next;
24.Set p2=p2->next;
25.End loop
26.End loop
27.Check whether p1 is exhausted then
28.Append(p1, phead3);
29.Append(p2, phead3);
30.End
```

5.7.1.2 The List ADT

A linked list is fundamental data type, which can be considered almost as basic as the array.
A **list** is a finite sequence of storage cells, for which the following operations are defined:

create(l) creates an empty list l;
insert(item, i, list) changes the list from the form (a1,..., ai - 1, ai,...an) by inserting an item between a_{i-1} and a_i; it is an error to call this unless the list has at least i – 1 items before the call.
delete(i, list) deletes the item at position *i* from the list, returning list containing one fewer item; and
read(i, list) returns the item at position *i* in the list, without changing the list.

Other operations on the list are search(), traverse(), merge(), search() is used to search a node in a list; traverse() is used to visit each node of the list and merge() is to combine two lists. It is often convenient to speak of the first item as the *head* of the list.

SUMMARY

- Linked list is a data structure, which is a collection of zero or more nodes, where each node contains two fields. The first field contains data, and the second field is a pointer field containing the address of the next node.
- Linked list basically supports insert and delete operations.
- Types of linked lists are singly linked list, doubly linked list, circular singly linked list and circular doubly linked list.
- A singly linked list is a linear data structure in which each node contains only one link field.
- A singly linked list in which the link field of the last node is assigned with the address of the first node is known as a circular linked list or circular list

- Doubly linked list is the one in which each node of it contains two links pointing to either side of the nodes.
- Applications of linked list are sparse matrix representation, polynomial manipulation, dynamic memory storage, in symbol table, etc.

EXERCISES

Fill in the Blanks

1. Linked lists are _____.
2. A node in a linked list consists of _____ and _____.
3. _____ is a pointer to a list of available memory spaces.
4. _____ are the types of linked list.
5. A node in a doubly linked list contains _____ and _____ pointers.
6. _____ are the applications of linked list.
7. Circular linked lists are used to overcome _____ disadvantage of a singly linked list.

Multiple-choice Questions

1. Linked lists are _____ structures.
   a. Static data
   b. Dynamic data
   c. Non-linear data
   d. None
2. Which of the following statements is true?
   a. Insertion in a linked list can be done only at front
   b. Traversing the list in only one direction
   c. Operations on a linked list can be performed anywhere in the list
   d. Both a and b
3. The situation when in a linked list START=NULL is
   a. Underflow
   b. Overflow
   c. Houseful
   d. Saturated
4. Which of the following is a two-way list?
   a. Grounded header list
   b. Circular header list
   c. Linked list with header and trailer nodes
   d. None
5. _____ is an application of a linked list.
   a. Towers of Hanoi
   b. Recursion
   c. Sparse matrices
   d. CPU scheduling

Short-answer Questions

1. Define a linked list.
2. Define a singly linked list.
3. Define a doubly linked list.
4. Give one disadvantage and one advantage of a singly linked list.
5. Give one disadvantage and one advantage of a doubly linked list.

6. List out four applications of linked lists.
7. Differentiate between a singly linked list and a doubly linked list.
8. Define a circular singly linked list with an example.
9. Briefly explain with diagrams how a node can be added to the linked list.
10. Differentiate between a doubly linked list and a circular singly linked list.

Essay Questions

1. With an example discuss how the data structure linked list is represented in memory.
2. Explain the following operations in a doubly linked list:
   (i) Create (ii) Insert (iii) Delete
   (iv) Display (v) Insert at a given position
3. Implement concatenation of two circular singly linked lists List1 and List2. Use header nodes to implement the list.
4. Write an algorithm to implement the following on a singly linked list.
   (i) To find the average of a set of values
   (ii) To replace all the occurrences at a given value by another value from the list.
   (iii) To insert a given value in to its proper position.
5. Write an algorithm to implement insertion and deletion on a singly linked list.
6. Write the algorithms for implementing various operations on a circular doubly linked list.
7. Write a program to perform polynomial multiplication using a linked list.

Chapter 6

Stacks

Stack data structure is exclusively discussed in this chapter. Representation and implementation of stacks are explained along with examples. Four of the common applications are stated and illustrated in detail.

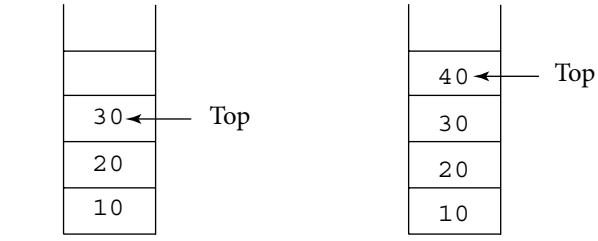
6.1 DEFINITION

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is termed as *Top* of stack. The last element that is inserted will be the first element to be deleted. Therefore, this is referred to as a Last In First Out (LIFO) structure.

Examples

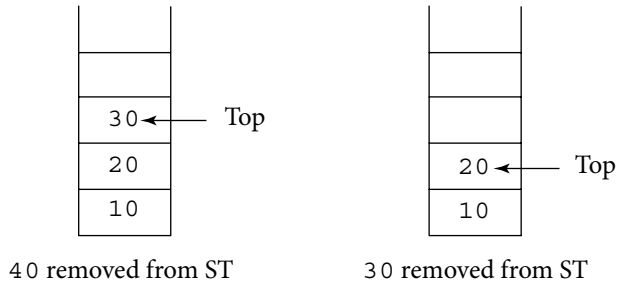
1. The stack of trays in a cafeteria
2. A stack of plates in a cupboard

Let ST be a stack with elements `10,20,30`. The element `30` is pointed by the top. When the new element `40` is to be added, it is inserted at the top. Now Top is pointing to the element `40` as in Figure 6.1(a). When deletion of an element is to be performed, the element pointed by the top is removed first, i.e. `40` is removed first as shown in Figure 6.1(b).



(a) Insertion

Figure 6.1 Stack and its working



(b) Deletion

Figure 6.1 *Continued*

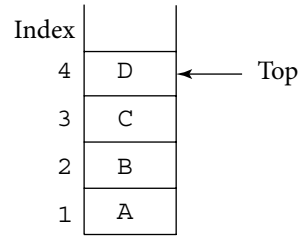
6.2 REPRESENTATION OF A STACK

The representation of a stack in the memory can be done in two ways. They are as follows:

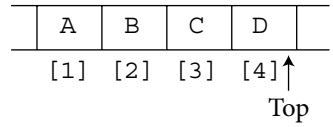
- Static representation using arrays
- Dynamic representation using linked lists

6.2.1 Array Representation of a Stack

Stack can be represented using a one-dimensional array. Allocate a block of memory required to accommodate the full capacity of the stack, and the items of a stack are stored in a sequential manner from the first location of the memory block.



(a) Stack



(b) Array representation of a stack

Figure 6.2 Stack and its array representation

In Figure 6.2(a), stack is a one-dimensional array. *Top* is a pointer that points to the top element in the stack.

6.2.2 Linked Representation of a Stack

Representing a stack using arrays is easy and convenient. However, it is useful for fixed sized stacks. Sometimes in a program, the size of a stack may be required to increase during execution, i.e. dynamic creation of a stack. Using a linked list, a stack can be dynamically created. A singly linked list is used to represent a stack. Figure 6.3 shows the linked representation of the stack. Here, the node consists of data and link fields. Data field is to store the items or data and link field is to point to the next field.

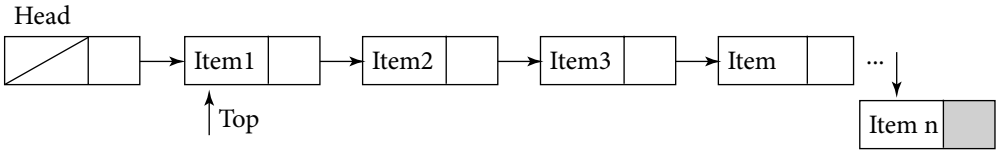


Figure 6.3 Linked representation of a stack

The item in the first node is the top of the stack and the item in the last node is the bottom of the stack.

6.3 OPERATIONS ON STACK

The basic operations on stacks are:

(i) PUSH—insertion of the elements into a stack
(ii) POP—deletion of the elements from the stack

Push operation: Inserting an element into the stack is called push operation. Check for Stack Overflow condition i.e. whether the stack is full of elements or not before performing Push operation. Every time an element is pushed into the stack, the top must be incremented. In Figure 6.4, the stack has 2 elements 10, 5.

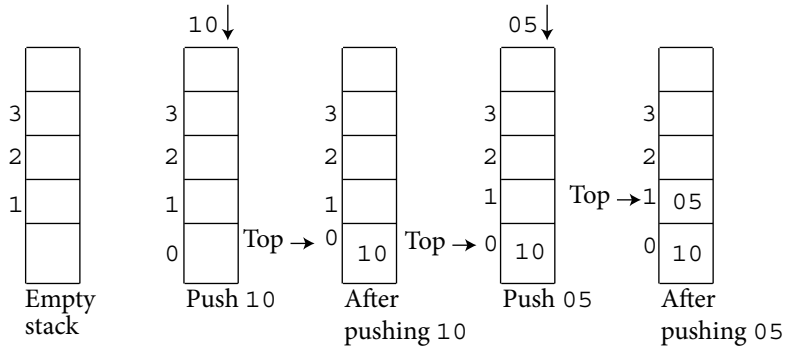


Figure 6.4 Push operation

Pop operation: Deleting an element from the stack is called Pop operation. Check for the Stack Underflow condition before performing Pop operation, i.e. whether the stack is empty or not. Every time an element from the stack is deleted, the top must be decremented by one. Now, the top points to the current top element of the stack. In Figure 6.5, the stack is empty after popping all the elements.

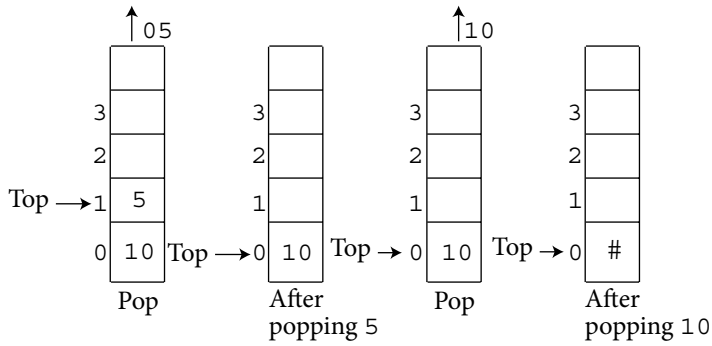


Figure 6.5 Pop operation

Push and *Pop* are the two important operations that are performed on the stack. In addition, *Show* and *Size* operations can also be performed on a stack. *Show* operation displays all the elements of the stack if it already has some elements. If not (i.e., if the stack is empty) it displays a message that the stack is empty. *Size* operation returns the number of elements in the stack.

6.3.1 Array Implementation of a Stack

Suppose *STACK [1: N]* is a linear array that represents a stack. *Top* is a pointer variable that holds the location of the top element of the stack and `N` is the number of elements that can be stored in a stack. Top = 0 or NULL represents that the stack is empty, and Top = `N` represents that the stack is full. The following are the algorithms for Push and Pop operations using arrays.

Algorithm 6.1: PUSH (STACK, N, TOP, ITEM)

This algorithm pushes an ITEM into a STACK; N is the maximum size of a stack.

```
1. [Check for stack overflow]
   If Top=N then write:"stack overflow" and return
2. Set Top=Top+1                  //increment Top by 1
3. Set STACK[Top]=ITEM           //insert item in new Top position
4. End
```

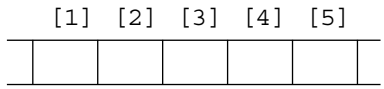
Algorithm 6.2: POP (STACK, TOP, ITEM)

This algorithm deletes the top element of the STACK and assigns it to the variable ITEM.

```
1. [check for stack underflow]
   If Top=0 then write:stack underflow and return
2. Set ITEM=STACK[Top]         //initializing Item with Top element
3. Set Top=Top-1                //Decrement top by 1
4. End
```

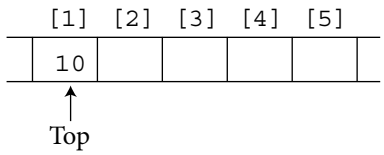
Consider a stack with five elements called `NUMBERS [1:5]` of numerical values. The insertion of values `10,20, 30,40,50` and deletion are shown in Figure 6.6.

- Initially stack NUMBER[1:5] is



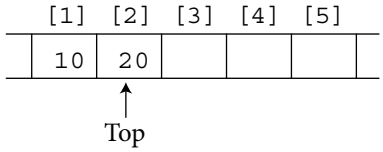
Top=0

- Push 10 into NUMBER[1:5]



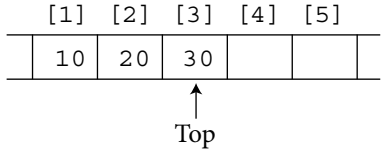
Top=1

- Push 20 into NUMBER[1:5]



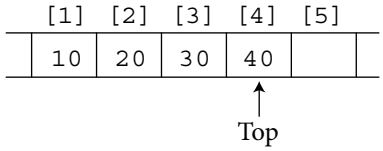
Top=2

- Push 30 into NUMBER[1:5]



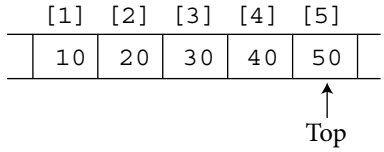
Top=3

- Push 40 into NUMBER[1:5]



Top=4

- Push 50 into NUMBER[1:5]



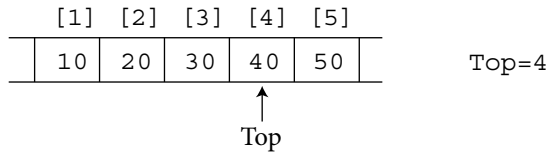
Top=5

- PUSH 60 into NUMBER[1:5]

Here TOP=N and cannot insert 60 into the stack since *stack overflow* condition is invoked.

The following steps illustrate the pop operation over the same stack:

- Pop an element from NUMBER[1:5]



- POP an element from NUMBER[1:5]

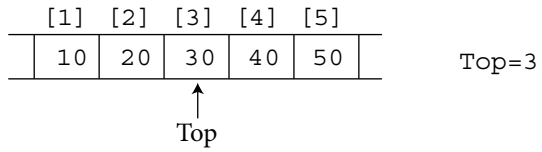


Figure 6.6 Array implementation of a stack

Here in the Pop operation the Top value is decremented, but the deletion of the element is not done. Program 6.1 is the program for array implementation of stack.

Program 6.1

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
template<class t>
class stack
{
   int top,n;
   t a[50];
   public:
   stack(int m)
   {
      top=-1;
      n=m;
   }
   void push();
   void pop();
   void display();
};
template<class t>
void stack<t>::push()
   {
      t ele;
      if(top>=n-1)
      {
         cout<<"stack is over flow\n";
         return;
      }
```

```
   top++;
   cout<<"enter the element \n";
   cin>>ele;
   a[top]=ele;
   }
template<class t>
void stack<t>::pop()
   {
      t ele;
      if(top==-1)
      {
         cout<<"stack is underflow\n";
         return;
      }
      ele=a[top];
      top--;
      cout<<"the deleted element is"<<ele;
   }
template<class t>
void stack<t>::display()
   {
      int i;
      if(top==-1)
      {
         cout<<"stack is underflow\n";
         return;
      }
      cout<<"the elements are\n";
      for(i=0;i<=top;i++)
      cout<<a[i]<<" ";
   }
void main()
   {
      int n,ch;
      clrscr();
      cout<<"enter the size of the stack\n";
      cin>>n;
      stack<char>s(n);
      while(1)
      {
         cout<<"\nmenu\n";
         cout<<"1.push\n";
         cout<<"2.pop\n";
         cout<<"3.display\n";
         cout<<"4.exit\n";
         cout<<"enter your choice\n";
         cin>>ch;
         switch(ch)
```

```
        {
           case 1:
              s.push();
              break;
           case 2:
              s.pop();
              break;
           case 3:
              s.display();
              break;
           case 4:
              exit(0);
              break;
        default:
              cout<<"invalid option\n";
              break;
     }
   }
}
```

Output

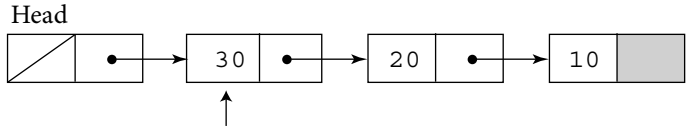
```
enter the size of the queue
10
menu
1.push
2.pop
3.display
4.exit
enter your choice
1
enter the element
2
menu
1.push
2.pop
3.display
4.exit
enter your choice
1
enter the element
3
menu
1.push
2.pop
3.display
4.exit
```

```
enter your choice
3
2 3
menu
1.push
2.pop
3.display
4.exit
enter your choice
2
the deleted element is 2
menu
1.push
2.pop
3.display
4.exit
enter your choice
4
```

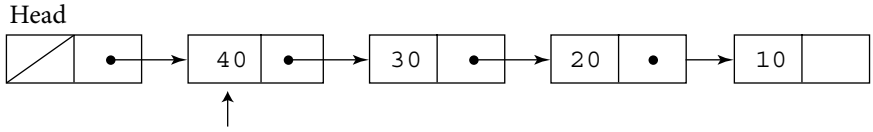
6.3.2 Linked Implementation of a Stack

To insert an item into the linked stack, insert the new node with that item as the first node in the singly linked list. The Top pointer that was previously pointing to the first element in the list now points to the new node. To delete an item from the linked stack, the node pointing by the top is deleted and Top points to the next node. Figure 6.7 illustrates the operations on the linked stack.

Initially stack is

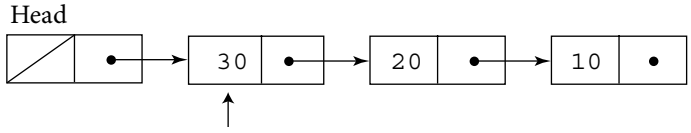


Insert 40 into the list



(a) Push operation

Delete 40 from the linked stack



(b) Pop operation

Figure 6.7 Linked implementation of a stack

The following are the algorithms for Push and Pop operations using linked list.

Algorithms 6.3: PUSH –LSTACK(TOP, ITEM)

Insert item ITEM into the linked stack LSTACK with pointer TOP

```
1. Insert ITEM into LSTACK
     NODE=call GETNODE (N)
2. Set DATA of NODE to ITEM       //node for item
3. Set LINK of NODE to Top       //insert node NODE into stack
4. Top=NODE                      //update top pointer
5. END.
```

Algorithm 6.4: POP–LSTACK (TOP, ITEM)

Delete the top element from LSTACK and assign to the ITEM.

```
1. Check stack is empty
     if(Top=0) then write "stack is empty"
2. Exit
3. Else
4. Temp=Top
5. Set ITEM to Top of DATA
6. Set Top to Top of LINK
7. End
```

Program 6.2 is the program for linked representation of stack.

Program 6.2

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
template<class t>
class llist
{
   struct node
   {
      t data;
      node*link;
   }
   *top,*head,*temp,*temp1;
   public:
   llist();
   void push();
   void pop();
   void display();
};
template<class t>
llist<t>::llist()
   {
      top=NULL;
```

```
      head->link=NULL;
   }
template<class t>
void llist<t>::push()
   {
      t ele;
      cout<<"enter the element\n";
      cin>>ele;
      temp=(struct node*)malloc(sizeof(struct node));
      if(temp==NULL)
      {
         cout<<"memory allocation error\n";
         return;
      }
   temp->data=ele;
   if(top==NULL)
   {
      head->link=temp;
      temp->link=NULL;
   }
else
   {
      top->link=temp;
      temp->link=NULL;
   }
   top=temp;
}
template<class t>
void llist<t>::pop()
   {
      t ele;
      if(top==NULL)
      {
         cout<<"linked list is empty\n";
         return;
      }
   for(temp1=head;temp1!=top;temp1=temp1->link)
   {
      temp=temp1;
   }
   temp->link=NULL;
   ele=temp1->data;
   free(temp1);
   top=temp;
   cout<<"the deleted element is"<<ele<<endl;
}
template<class t>
void llist<t>::display()
```

```
    {
       if(top==NULL)
       {
          cout<<"linked list is empty\n";
          return;
       }
    for(temp=head->link;temp!=top->link;temp=temp->link)
    {
       cout<<temp->data<<" ";
    }
}
void main()
{
int ch;
llist<int> l;
clrscr();
while(1)
   {
      cout<<"\nmenu\n";
      cout<<"1.insertion\n";
      cout<<"2.deletion\n";
      cout<<"3.display\n";
      cout<<"4.exit\n";
      cout<<"enter your choice\n";
      cin>>ch;
      switch(ch)
      {
         case 1:
            l.push();
            break;
         case 2:
            l.pop();
            break;
         case 3:
            l.display();
            break;
         case 4:
            exit(0);
            break;
      }
   }
}
```

Output

```
menu
1. insertion
2. deletion
```

```
3. display
4. exit
enter your choice
1
enter the element
10
menu
1.insertion
2.deletion
3.display
4.exit
enter your choice
1
enter the element
20
menu
1.insertion
2.deletion
3.display
4.exit
enter your choice
3
10 20
menu
1.insertion
2.deletion
3.display
4.exit
enter your choice
2
the deleted element is 20
menu
1.insertion
2.deletion
3.display
4.exit
enter your choice
3
10
menu
1.insertion
2.deletion
3.display
4.exit
enter your choice
4
```

The implementation of stack using arrays and linked list are discussed. In the following section a detailed explanation is given on the areas where stack is applied.

6.4 APPLICATIONS OF STACKS

Stacks have a wide range of applications. The following are some of the applications of stack:

- Expression evaluation
- Recursion
- Balancing of the matching parenthesis

6.4.1 Expression Evaluation

An arithmetic expression can be represented in three ways:

(1) *Infix*: An operator between two operands is an infix expression.

```
<operand><operator><operand>
```

Ex: `a+b`

(2) *Postfix*: An operator that follows two operands is a postfix expression.

```
<operand><operand><operator>
```

Ex: `a b+`

(3) *Prefix*: An operator that is followed by two operands is a prefix expression.

```
<operator><operand><operand>
```

Ex: `+ab`

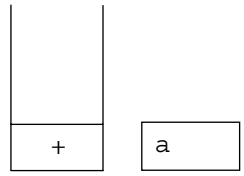
Stacks can be used to evaluate expressions and also to convert expressions from one form to another.

Infix to postfix conversion: In algebra the infix notation is like `a+b.c`. The corresponding postfix notation is `abc.+`. Algorithm 6.5 shows the conversion.

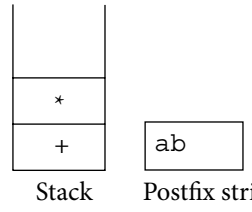
Algorithm 6.5:

1. Scan the Infix string from left to right.
2. Create an empty stack.
3. If the scanned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty, Push the character to stack.
4. If the scanned character is an operator and the stack is not empty, compare the precedence of the character with the element on top of the stack (here after top of the stack is referred as topStack). If topStack has higher precedence over the scanned character then pop the stack, else push the scanned character into the stack. Repeat this step as long as the stack is not empty and topStack has precedence over the scanned character. Continue this step till all the characters are scanned.
5. Return the postfix string.

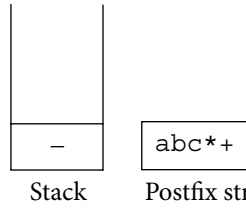
Consider the infix string: `a+b*c-d`. Initially, the stack is empty and the postfix string has no characters. Now, the first character scanned is 'a'. The character '`a`' is added to the postfix string. The next character scanned is '+', being an operator it is pushed into the stack.



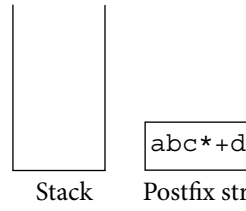
Next, the character scanned is 'b' which will be placed in the postfix string, since it is an operand. The next character is '*', which is an operator. Now, the top element of the stack is '+', which has a lower precedence than '*', so '*' will be pushed on to the stack.



The next character is 'c', which is placed in the postfix string. The next character scanned is '–'. The topmost character in the stack is '*', which has a higher precedence than '–'. Thus '*', will be popped out from the stack and added to the postfix string. Even now the stack is not empty. Now, the topmost element of the stack is '+', which has equal priority to '–'. So, pop the '+' from the stack and add it to the postfix string. The '–' will be pushed to the stack.



The next character is 'd', which is added to the postfix string. Now, all characters have been scanned so pop the remaining elements from the stack and add it to the postfix string. At this stage, only '–' is in the stack. It is popped out and added to the postfix string. So, after all characters are scanned, the stack and the postfix string will be as follows:



The end result for a given infix string : `a+b*c-d` is postfix string : `abc*+d-`

Program 6.3

```
/*infix to postfix conversion*/
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
class expression
{
   private:
       char infix[100];
       char stack[200];
       int top;
       int r;
       char postfix[100];
   public:
       void convert();
       int input_p(char);
       int stack_p(char);
       int rank(char);
};
int expression::input_p(char c)
{
 if(c=='+'||c=='-')
  return 1;
 else if(c=='*'||c=='/')
  return 3;
 else if(c=='^')
  return 6;
 else if(isalpha(c)!=0)
  return 7;
 else if(c=='(')
  return 9;
 else if(c==')')
  return 0;
 else
   {
     cout<<"Invalid expression::input error\n";
     exit(0);
   }
}
int expression::stack_p(char c)
{
 if(c=='+'||c=='-')
  return 2;
 else if(c=='*'||c=='/')
  return 4;
 else if(c=='^')
  return 5;
```

```
 else if(isalpha(c)!=0)
  return 8;
 else if(c=='(')
  return 0;
 else
    {
        cout<<"Invalid expression::stack error\n";
        exit(0);
    }
}
int expression::rank(char c)
{
 if(c=='+'||c=='-')
  return -1;
  else if(c=='*'||c=='/')
  return -1;
  else if(c=='^')
  return -1;
  else if(isalpha(c)!=0)
  return 1;
  else
 {
  cout<<"Invalid expression::in rank\n";
  exit(0);
 }
}
void expression::convert()
{
 cout<<"infix expression to postfix form"
 cout<<"Enter an infix expression::\n";
 cin>>infix;
 int l=strlen(infix);
 infix[l]=')';
 infix[l+1]='';

 //Convertion starts
 top=1;
 stack[top]='(';
 r=0;
 int x=-1;
 int i=0;
 char next=infix[i];
 while(next!='')
 {
  //Pop all the elements to outputin stack which have higher precedence
  while(input_p(next)<stack_p(stack[top]))
  {
   if(top<1)
   {
```

```
     cout<<"invalid expression::stack error\n";
     exit(0);
    }
    postfix[++x]=stack[top];
    top-;
    r=r+rank(postfix[x]);

    if(r<1)
    {
     cout<<"Invalid expression::r<1\n";
     exit(0);
    }
   }
   if(input_p( next )!=stack_p(stack[top]))
    stack[++top]=next;
   else
    top-;
   i++;
   next=infix[i];
  }
  postfix[++x]=";
  if(r!=1 || top!=0)
  {
   cout<<"Invalid expression::error in rank or stack\n";
   exit(0);
  }
  cout<<"\n\nThe corresponding postfix expression is::\n";
  cout<<postfix<<endl;
}
int main()
{
 expression obj;
 obj.convert();
 return 0;
}
```

Output

```
infix expression to postfix form
Enter an infix expression::
(a+b^c^d)*(c+d)
The corresponding postfix expression is::
abcd^^+cd+*
Press any key to continue
```

Similarly, other conversions can be implemented using stacks.

6.4.2 Postfix Evaluation

The postfix expression can be easily evaluated using a stack. Scan the expression from left to right and if any operand is found, push them into the stack. If any operator is found, pop the required number of operands from the stack and then perform the operation. Push the result again into the stack. Finally, the evaluated expression is placed into the stack.

In postfix notation the order of the operations is determined by the precedence hierarchy and associativity of the operators and operands in the expressions. Algorithm 6.6 explains the postfix evaluation.

Algorithm 6.6: POST-EVAL (EXP)

```
1. Scan EXP from left to right and repeat step 2 and step 3 for each element
of EXP.
2. If operand is found, push it on to the stack.
   1. If operator is found then
      1. Pop the required number of operands from the stack
      2. Evaluate the operator
      3. PUSH the result back into the stack.
         End if
3. end if
4. End of step1 loop
5. Pop out the result from stack
6. End
```

Consider the following expression in postfix form and evaluate the expression:

```
Postfix    2, 3, 4,+,*,21,7,/,-.
```

Scan	Operation	Stack
2	PUSH(2)	2
3	PUSH(3)	2,3
4	PUSH(4)	2,3,4
+	POP(4),POP(3),PUSH(7)	2,7
*	POP(7),POP(2),PUSH(14)	14
21	PUSH(21)	14,21
7	PUSH(7)	14,21,7
/	POP(7),POP(14),PUSH(2)	14,3
–	POP(2),POP(14),PUSH(12)	11

Finally the value of the expression is 11.

6.4.3 Recursion

A function calling itself or a call to another function, which in turn calls the first function is called a recursive function. These recursive functions are executed using stacks. Return address and all local variables and formal parameters of the called method will be stored into the stack. Whenever any method is called all the elements stored in the stack will be restored after a return is executed.

Recursion is implemented using the data structure stack. Each recursive call to a method requires that the following information to be pushed onto the system stack:

- Formal parameters
- Local variables
- Return address

This information, collectively, is referred to as a stack frame. The stack frame for each method call will be different. For example, the stack frame for a parameterless or no local variable method will contain just a return address. The compiler will take care of the entire above required stack maintenance when a call to a recursive method is made.

The following strategy can be used to remove the recursion from a recursive routine, although not elegantly. There might be a far more pleasing method for a particular routine, but this technique is very instructional. It allows one to simulate the stack by declaring user's own stack structure and manage the recursion. This is accomplished as follows:

i. Each time a recursive call is made in the algorithm, push the necessary information onto the stack.
ii. When the process is completed at this deeper level, pop the simulated stack frame and continue processing in the higher level at the point indictated by the return address popped from the stack.
iii. Use an iterative control structure that loops until there are no return addresses left to pop from the stack.

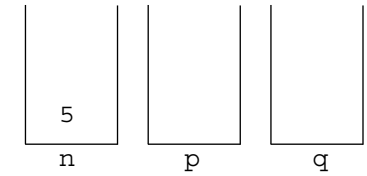
Algorithm 6.7: FACT (n)

```
1. If(n==0) then set fact=1 and return
2. else p=n-1
3. q=fact(p)
4. fact=n*q;
5. End
```

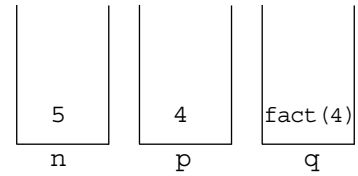
The use of the stack in performing recursion is explained through a simple recursive program. Consider the recursive program for a factorial as shown in Figure 6.8.

Examine the recursive call in Step 3. To avoid an endless series of calls to `fact()`, a base case `n==0` is used in Step 1. The function `fact()a` calling itself is based on the value of `n`. The function `fact()` has normal termination only when the base case is executed or recursive calls have successfully ended.

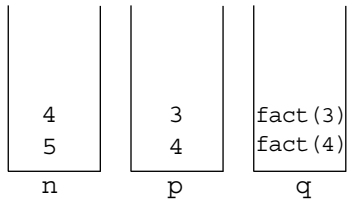
While executing, to maintain the record of values of parameters and to keep track of calls made by itself, the stack is used.



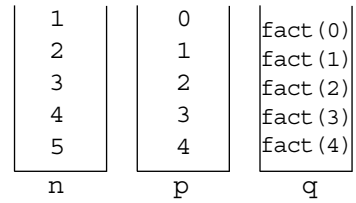
(a) Call to `fact(5)`



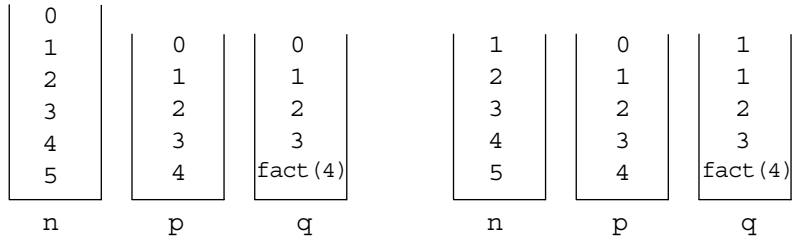
(b) Calling `fact(4)` during execution of `fact(5)`



(c) Calling `fact(3)` during execution of `fact(4)`



(d) Contents of stack after making subsequent calls and during call to `fact(0)`



(e) Call to `fact(0)`

(f) `fact(0)` has normal termination since `0!=1` and returns to the called position

(g) `fact(1)` terminates returning the value `1!=1` to the variable `q`

(h) Stack contents for calls to `fact()` except for `fact(5)`

Figure 6.8 Recursive calls to function `fact()`

When the function `fact(5)` is called and executed, the variable `p` gets the value `4` and calls the function `fact(4)` as shown in Figure 6.8(b), still the call to `fact(5)` is not yet finished. Every new function call pushes the current values of the parameter into the stack. Figure 6.8(d) illustrates the stack contents during call to `fact(4)` and subsequent calls. Call to `fact(0)` in Figure 6.8(f) satisfies the condition in step 1 and gets terminated yielding the value for `0!` as `1` and returns it to the variable `q`. In Step 4 of the algorithm the calculating `fact = n * q = 1 * 1 = 1`. Then, the function fact (1) terminates yielding the value `1` to Step 3 for the previous call to `fact(2)` and computes `fact = n * q = 2 * 1 = 2` and terminates by returning the result to `q` and so on.

Stack structure increases in size due to push operation during the function calls and decreases itself in size by pop operation until the first call to `fact()` is reached, i.e. `fact(5)`. During the execution of `fact(5)`, the first and the oldest call in Step 3 computes the `q` value as 24 and proceeds to get the `fact = n * q = 5 * 2 4 = 1 2 0`, which is the required value.

Generally in the implementation of recursive calls using stack, all the local variables are pushed on to the stack when the call is made and poped out from the stack when the recursive call is terminated and their original values are restored.

6.4.4 Balancing of the Matching Parenthesis

Given an expression of '(' and ')' which are left and right parenthesis. A '(' must match with a ')', or else it is invalid. For example, ( )( ), ((( ))), ((( )) ( )) are valid sets of parenthesis, while (( ) (,) ( ) (are all illegal. Obviously, counting the numbers of '(' and ')' in the expression is not enough. By convention, the expression is reading from left to right. A ')' is matched with the closest, unmatched '(' on its left. For example, ((( )) ( )).

Suppose to find which is closest and unmatched for ')', read the expression from left to right; the most recently unmatched '(' is cancelled with ')'. To keep track, the most recently read '(' assuming that there are many pending unmatched '(', stack is useful.

- When a '(' is processed, *push* it into a stack.
- When a ')' is processed, *pop* it from the stack. This '(' matches with current ')'.
- If reading the expression is finished, but the **stack** is not empty, it means there are more '(' than ')' in the expression, then the input is invalid.

6.4.4.1 The ADT Stack

A **stack** is a collection of elements or items, for which the following operations are defined:

- **create(S)** creates an empty stack S.
- **isEmpty(S)** is a predicate that returns "true" if S exists and is empty, and "false" otherwise.
- **push(S,item)** adds the given item to the stack S.
- **pop(S)** removes the most recently added item from the stack S and returns it as the value of the function.

The primitive **isempty** is needed to avoid calling pop on an empty stack, which may cause an error. An additional operation called top(S) is sometimes defined, which simply returns the last item to be added to the stack, without removing it.

SUMMARY

- Stack is a non-primitive linear data structure into which elements are inserted and deleted from the same end called Top.
- Stack supports two basic operations, Push and Pop.
- There are many applications of stacks like recursion elimination, evaluation of expressions, etc.
- Stack can be implemented using arrays or linked lists.

EXCERCISES

Fill in the Blanks

1. The insertion and deletion are done at _____ of the stack.
2. _____ and _____ are the operations of stack.
3. _____ is an application of stack.
4. When rear= =size of stack then stack is _____.
5. The postfix evaluation is done based on _____.

Multiple-choice Questions

1. The stack is a _____ data structure.
   a. FIFO b. LIFO
   c. Both a and b d. FILO
2. The pointer Top increases when an _____.
   a. Element is inserted b. Element is deleted
   c. Element is displayed d. None
3. Which of the following is not stack application?
   a. Recursion b. Templates
   c. String reversion d. Process scheduling

4. Stacks are dynamically represented using _____.
   a. Pointers
   b. Structures
   c. Linked lists
   d. None
5. When is a stack said to be empty?
   a. top==n
   b. top==0
   c. front==0
   d. front=rear=0

Short-answer Questions

1. Define a stack.
2. Define a postfix expression.
3. Give examples for infix, postfix and prefix expressions.
4. Convert the following postfix expression to infix expression ABCDG/*++.
5. Define prefix expression.
6. List out any four applications of stack.
7. Write a C function to insert an element on the top of the stack.
8. Evaluate the following postfix expression: 123+*321-+*.
9. Describe the operations performed over the stack.

Essay Questions

1. Write an algorithm for converting postfix expression; further trace the algorithm clearly indicating the contents of the stack for the following expression:
   ((A-(B+C)) * D)$(E+F)
2. Discuss the various exceptional conditions that should be handled while using stacks and convert the following expression to infix and prefix expressions: ABCDE/*-F/G++
3. How do you define a data structure? Give a C++ program to construct a stack of integers and to perform all necessary operations on it.
4. Write a C++ program to implement multiple stacks using a single array.
5. Give a complete specification of DS for stack and explain how a stack of N-integers is represented in C++.

Chapter 7

Queues

Chapter 7 discusses about queues and its variants such as circular queues and doubly ended queues. The static, dynamic representation and operations of all the types of queues are explained and exemplified. Various applications of the queues are listed, and one of the applications called Simulation of Time-Sharing System is explained in detail.

7.1 INTRODUCTION

Queue is a linear data structure in which insertions are made at one end called *rear*, and deletions are made at the other end called *front*. A queue is also known as a structure that models the *first-come first-serve order* or equivalently the *first-in first-out* (FIFO) order. That is, the element that is inserted first into the queue will be the element to be deleted first, and the element that is inserted last is deleted last. The insertion operation in the queue is also known as *enqueue,* and deletion is also referred as *deque*. The main difference between a queue and a stack is that elements in a queue are put at the bottom and taken off from the top whereas in a stack, elements put at the top and taken off from the top.

Examples

- An electronic mailbox
- A waiting line in a store, at a service counter, on a one-lane road, at a bank, at a bus stop, playlist for jukebox, etc.
- Equal-priority processes waiting to run on a processor in a computer system
- Printer jobs
- CPU processes

Queues are more difficult to implement than stacks, because operations are performed at both the ends. The simplest implementation uses an array, adds elements at one end and *moves* all the elements when one of the elements is taken out of the queue.

7.2 REPRESENTATION OF A QUEUE

Queues can be represented in two ways:

- Static representation using arrays
- Dynamic representation using linked list

7.2.1 Array Representation

A one-dimensional array is used to represent a queue. Consider a queue `QUE[1:N]`; two pointers `FRONT` and `REAR` represent two ends of the `QUE`. To insert an element, the `REAR` pointer is used and to delete an element, the `FRONT` pointer is used (Figure 7.1).

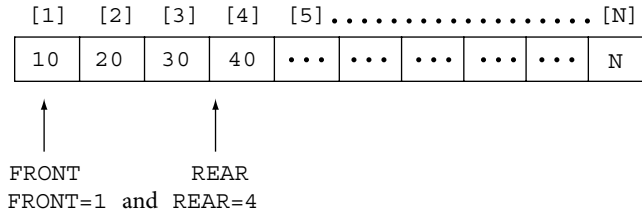


Figure 7.1 Array representation of queue `QUE[1:N]`

A queue `QUE[1:N]` cannot hold more than `N` elements, and every insertion of an element must check the condition whether the `QUE` is full, and deletion of an element must ensure that `QUE` is not empty.

7.2.2 Linked Representation

Using linked list, queue is created dynamically by which the size of the queue can be increased as per the necessity in the application or program, whereas using arrays the size of the queue is fixed.

Figure 7.2 shows the single linked list representation of a queue. The pointer `FRONT` points to the first node, and `REAR` points to the last node in the list.

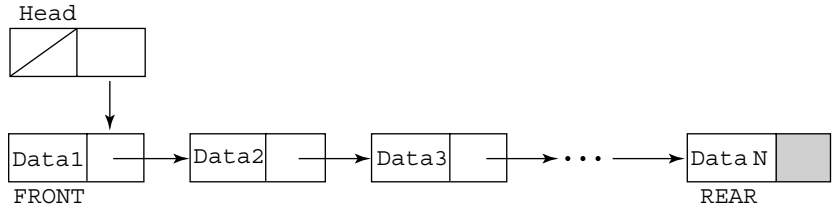


Figure 7.2 Linked representation of a queue

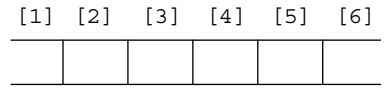
When `FRONT=REAR` the queue is said to be empty.

7.3 OPERATIONS ON A QUEUE

The two basic operations on queues are:

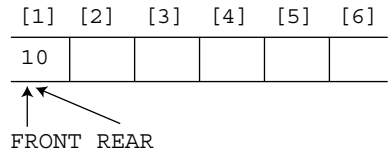
1. Insertion (Enqueue)—adds an element into the queue
2. Deletion (Deque)—removes an element from the queue

Enqueue: Inserting an element into a queue will be done at the rear end. When the queue is empty both rear and front will be initialized to 0. Before inserting any element, check for queue overflow condition, i.e. whether the queue is full or not. If it is full display the appropriate message otherwise insert an element. While inserting, check whether front is 0 or not. If it is so, this is the first element to be inserted, so both front and rear must be incremented. If not, only the rear end is incremented. Figure 7.3 illustrates the insertion operation.

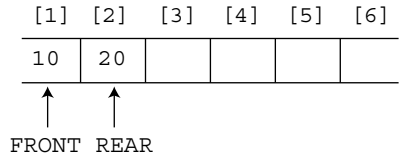


FRONT=0 and REAR=0

(a) Initially queue is empty



(b) After inserting 10

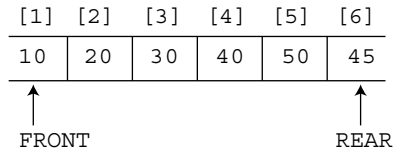


(c) After inserting 20

Figure 7.3 Insertion operation

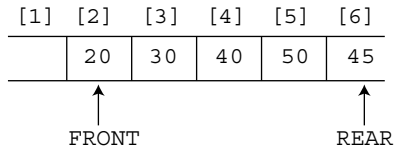
In Figure 7.3(a), initially the queue is empty and `FRONT` and `REAR` pointers are initialized to zero. When element `10` is inserted, the `FRONT` and `REAR` pointers are incremented by 1 and now both point to the first position; this is shown in Figure 7.3(b). When element `20` is inserted, the `REAR` pointer is incremented by 1 and now points to the second position as in Figure 7.3(c).

Deque: Deleting an element from the queue is done at the front end. The element that is inserted first will be the first element to be deleted. Initially, check for queue underflow condition, i.e. if both front and rear are pointing to 0, that means (there are no elements in queue) the queue is empty. While deleting an element, front will be incremented by 1.

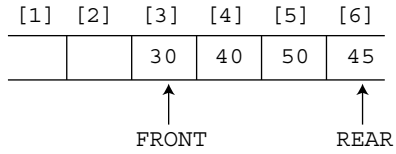


(a) Initial queue

Figure 7.4 Deletion operation



(b) After deleting 10



(c) After deleting 20

Figure 7.4 *Continued*

In Figure 7.4(a), the queue is with 6 elements. FRONT points to the first element and REAR points to the sixth element. After deleting element 10 the FRONT pointer is incremented by 1, as shown in Figure 7.4(b). Similarly when element 20 is deleted, FRONT is updated.

After deleting the entire elements, front will be greater than rear. So, after every deletion, check for front greater than rear condition. In Figure 7.4(c), suppose all the elements 30, 40, 50, 45 are deleted then the FRONT value will be 7 (because after deleting element 45 FRONT is incremented by 1) and the REAR value will be 6. So, if FRONT is greater than REAR again initialize both FRONT and REAR to 0, which indicates that the queue is empty.

Thus, inserting an element and deleting an element are the two important operations of a queue. In addition, two other operations are also there. They are size and display.

- Size gives us the number of elements in the queue
- Display prints all the elements in the queue

7.3.1 Enqueue and Deque Using Arrays

The following algorithms describe the insertion and deletion operations using arrays. Let QUE[1:N] be an array representation of a queue. Suppose FRONT and REAR are the two pointers pointing to the *front* and *rear* positions of the queue, respectively. Initially FRONT and REAR pointers are initialized with 0.

Algorithm 7.1: INSERT (QUE, N, ITEM, REAR)

```
1. Check for "Queue is full"
   if(REAR==N)then Write "Queue is full"
2. Else
3. Increment FRONT by 1
3. Increment REAR by 1
4. SET QUE[REAR] to ITEM
5. Endif
6. End
```

In Algorithm 7.1, insertion of a new element is done by incrementing the REAR variable. Before insertion, check whether the condition QUE is full to ensure that there is no overflow in the queue.

Algorithms 7.2: DELETE (QUE, FRONT, REAR, ITEM)

```
1. Check for "Queue is empty"
   If(FRONT==REAR) then Write "Queue is empty"
2. Else
3. Set ITEM to FRONT of QUE
4. Increment FRONT by 1
5. Endif
6. End
```

In Algorithm 7.2, deletion of element is done by just incrementing the FRONT variable so that it points to the next element.

In the array implementation of queues while inserting an element, the REAR moves away from FRONT, and while deleting an element the FRONT moves towards the REAR in the array. When the queue is empty FRONT=REAR and when the queue is full REAR=N, where N is the size of the queue.

Example

Let QUE[1:4] is a linear queue. The operations insertion and deletion on QUE are as shown below:

- Initially QUE[1:4] is

[1]	[2]	[3]	[4]

FRONT=0 and REAR=0

- Insert 10 by calling INSERT(QUE,4,10,0), then QUE is as follows:

[1]	[2]	[3]	[4]
10			

FRONT=1 and REAR=1

- Insert 20 by calling INSERT(QUE,4,20,1), then QUE becomes:

[1]	[2]	[3]	[4]
10	20		

FRONT=1 and REAR=2

- Insert 30 by calling INSERT(QUE,4,30,2), then QUE becomes:

[1]	[2]	[3]	[4]
10	20	30	

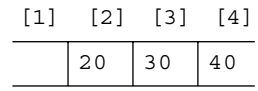
FRONT=1 and REAR=3

- Insert 40 by calling INSERT(QUE,4,40,3), then QUE becomes:

[1]	[2]	[3]	[4]
10	20	30	40

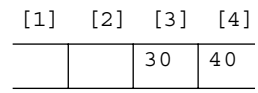
FRONT=1 and REAR=4

- Deleting element 10 from QUE by calling DELETE(QUE,1,4,10):



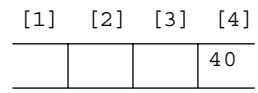
FRONT=1, REAR=4
and ITEM=10

- Deleting element 20 from QUE by calling DELETE(QUE,2,4,20):



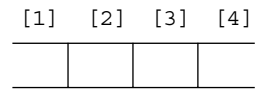
FRONT=2, REAR=4
and ITEM=20

- Deleting element 30 from QUE by calling DELETE(QUE,3,4,30):



FRONT=3, REAR=4
and ITEM=30

- Insert element 50 by making call as INSERT(QUE,4,50,4). The condition *Queue Overflow* is invoked and insertion fails because REAR=N that is REAR=4. So, QUE is full.
- Delete element 40 from QUE, then it becomes:



FRONT=4, REAR=4
and ITEM=40

- Again performing deletion by calling DELETE(QUE,4,4,ITEM) invokes the condition *Queue Underflow* which reports an error because FRONT=REAR that shows that QUE is empty.

Program 7.1 shows the implementation of a queue using arrays.

Program 7.1

```
// Array implementation of queue.
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
template<class t>
class queue
{
   int rear,front,n;
   t a[50];
   public:
```

```
   queue(int m)
   {
      rear=front=-1;
      n=m;
   }
   void push();
   void pop();
   void display();
};
template<class t>
void queue<t>::push()
{
   t ele;
   if(rear>=n-1)
   {
      cout<<"queue is overflow\n";
      return;
   }
   cout<<"enter the element \n";
   cin>>ele;
   if(front==-1)
      rear=front=0;
   else
      rear++;
   a[rear]=ele;
}
template<class t>
void queue<t>::pop()
{
   t ele;
   if(rear==-1)
   {
      cout<<"queue is underflow\n";
      return;
   }
   ele=a[front];
   if(rear==front)
      rear=front=-1;
   else
      front++;
   cout<<"the deleted element is"<<ele;
}
template<class t>
void queue<t>::display()
{
   int i;
   if(rear==-1)
   {
```

```
      cout<<"queue is underflow\n";
      return;
   }
   for(i=front;i<=rear;i++)
   cout<<a[i]<<" ";
   cout<<"\n";
}
void main()
{
   int ch,n;
   clrscr();
   cout<<"enter the size of the queue\n";
   cin>>n;
      queue<int>q(n);
   while(1)
   {
      cout<<"\nmenu\n";
      cout<<"1.push\n";
      cout<<"2.pop\n";
      cout<<"3.display\n";
      cout<<"4.exit\n";
      cout<<"enter your choice\n";
      cin>>ch;
      switch(ch)
      {
         case 1:
            q.push();
            break;
         case 2:
            q.pop();
            break;
         case 3:
            q.display();
            break;
         case 4:
            exit(0);
            break;
      default:
         cout<<"invalid option\n";
         break;
      }
   }
}
```

Output

```
enter the size of the queue
10
```

```
menu
1.push
2.pop
3.display
4.exit
enter your choice
1
enter the element
2
menu
1.push
2.pop
3.display
4.exit
enter your choice
1
enter the element
3
menu
1.push
2.pop
3.display
4.exit
enter your choice
3
2 3
menu
1.push
2.pop
3.display
4.exit
enter your choice
2
the deleted element is 2
menu
1.push
2.pop
3.display
4.exit
enter your choice
4
```

7.3.2 Enqueue and Deque Using Linked List

The following algorithm describes the insertion and deletion operations using the linked list.

Algorithm 7.3: INSERT-LINK(FRONT, REAR, ITEM)

Insert ITEM into a QUE with FRONT and REAR as the *front* and *rear* pointers to the Queue

```
1. Call GETNODE(N)
   //Create new node
2. Set ITEM to new node DATA
3. Set new node LINK to NULL
4. If(FRONT=0) then
5. Set FRONT, REAR to N //If QUE is empty ITEM is the first element in Que
6. Else
7. Set  LINK of REAR  to N
8. Set N to REAR
9. End
```

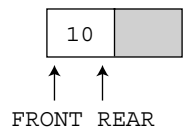
Algorithm7.4: DELETE-LINK(FRONT,ITEM)

```
// Check for the condition Queue empty
1. If(FRONT=0)then write "Queue is empty"
2. Else
3. Set FRONT to PTR
4. Set ITEM with DATA of PTR
5. Set FRONT with LINK of  PTR
6. Return PTR
7. End
```

Example

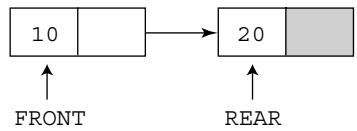
Consider the queue QUE, three insertions and two deletions are shown below. **For the sake of an explanation, the HEAD node is not considered**.

- Initially QUE is empty. Insert 10 into the list.



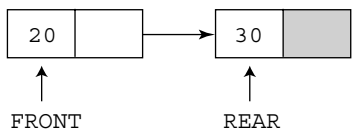
  Element 10 is the first node and both FRONT and REAR point to the node.

- Insert 20 into QUE:

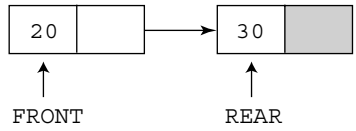


  Now REAR points to the new node.

- Insert 30 into the QUE and update REAR:

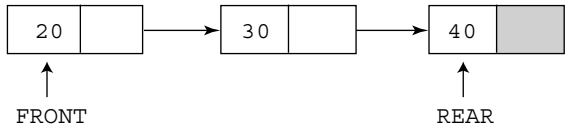


- Now delete the node pointed by FRONT, that is 10 and update FRONT:



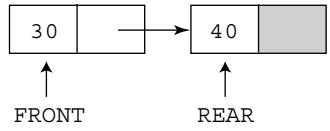
Now FRONT points to the node with element 20.

- Insert 40 to the QUE and update REAR.



FRONT points to node 20 and REAR points to node 40.

- Delete the node pointed by FRONT and update FRONT.



The deleted ITEM is 20. Previously front points to node 20, now it is updated to point to node 30.

Program 7.2 shows the linked implementation of a queue.

Program 7.2

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
template<class t>
class cqueue
{
   struct node
   {
      t data;
      node*link;
      }*head,*temp,*front,*rear;
   public:
   cqueue();
   void push();
   void pop();
   void display();
};
template<class t>
cqueue<t>::cqueue()
{
   head->link=NULL;
   rear=NULL;
```

```
   front=NULL;
}
template<class t>
void cqueue<t>::push()
{
   t ele;
   cout<<"enter the element\n";
   cin>>ele;
   temp=(struct node*)malloc(sizeof(struct node));
   if(temp==NULL)
   {
      cout<<"memory allocation error\n";
      return;
   }
if(front==NULL)
{
   temp->data=ele;
   head->link=temp;
   temp->link=NULL;
   front=rear=temp;
}
else
   {
   temp->data=ele;
   rear->link=temp;
   temp->link=NULL;
   rear=temp;
   }
}
template<class t>
void cqueue<t>::pop()
{
   t ele;
   if(front==NULL)
   {
      cout<<"linked list is empty\n";
      return;
   }
if(front==rear)
{
   ele=front->data;
   front=rear=NULL;
}
else
   {
   ele=front->data;
   temp=front;
```

```
   front=front->link;
   free(temp);
   }
   cout<<"the deleted element is"<<ele<<endl;
}
template<class t>
void cqueue<t>::display()
{
   if(rear==NULL)
{
   cout<<"linked list is empty\n";
   return;
}
for(temp=front;temp!=rear->link;temp=temp->link)
{
   cout<<temp->data<<" ";
   }
}
void main()
{
   int ch;
   cqueue<int>q;
   clrscr();
   while(1)
   {
      cout<<"\nmenu\n";
      cout<<"1.push\n";
      cout<<"2.pop\n";
      cout<<"3.display\n";
      cout<<"4.exit\n";
      cout<<"enter your choice\n";
      cin>>ch;
      switch(ch)
      {
         case 1:
            q.push();
            break;
         case 2:
            q.pop();
            break;
         case 3:
            q.display();
            break;
         case 4:
            exit(0);
            break;
         default:
      cout<<"invalid option\n";
```

```
        break;
        }
    }
}
```

Output

```
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
1
enter the element
10
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
1
enter the element
20
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
20 10
menu
1.insertion at front
2.insertion at end
```

```
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
2
enter the element
30
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
2
enter the element
40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
20 10 30 40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
3
enter the element
50
```

```
enter the position
3
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
20 10 50 30 40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
4
the deleted element is 20
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
10 50 30 40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
```

```
5
the deleted element is 40
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
10 50 30
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
6
enter the  position
2
the deleted element is 50
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
8.exit
enter your choice
7
10 30
menu
1.insertion at front
2.insertion at end
3.insertion at any position
4.deletion at front
5.deletion at end
6.deletion at any position
7.display
```

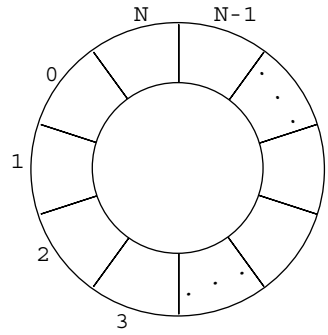
```
8.exit
enter your choice
8
```

7.3.2.1 Disadvantages of a Linear Queue

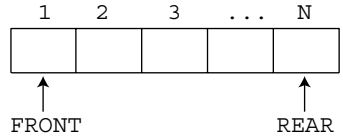
Consider the example discussed in Section 7.3.1, enqueue and deque using arrays. The insertion of element 50 fails because the REAR value is equal to the maximum size of the queue, i.e. REAR=4, but positions [1], [2] and [3] are still empty. This leads to the limitation of linear queue. When REAR=N the queue overflow condition is true but this doesn't mean that the queue is full, i.e. locations [1],[2] and [3] are empty. Therefore, some space is available to insert a new element in the place of deleted elements. This limitation can be prevailed over by a circular queue.

7.4 CIRCULAR QUEUES

In a circular queue the insertion of an element can be done even when *rear* is equal to the maximum size of a queue when the space is available at *front* in the queue, which is not possible in a linear queue. Both FRONT and REAR pointers move in a clockwise direction over the queue. The array representation of a circular queue is the same as a normal queue physically. But, logically in a circular queue the front element comes after the last element. Figure 7.5 shows the physical and logical view of a circular queue.



(a) Logical view of a circular queue



(b) Physical view of a circular queue

Figure 7.5 Circular queue

The representation of a circular queue is the same as a normal queue with the only difference being that the REAR pointer is not the end of the queue and will again be pointing towards the starting position.

7.4.1 Operations on Circular Queue

All the operations performed on the queue can also be performed on a circular queue. Algorithms INSERTION-CQ and DELETION-CQ describe the insertion and deletion operations in the circular queue. The array implementation of a circular queue is declared as CIR-QUE[0, N-1] to provide circular movement of FRONT and REAR pointers.

Algorithm7.5: INSERT-CQ(CIR-QUE ,FRONT,REAR,N,ITEM)

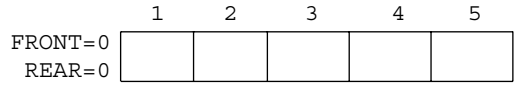
```
//ITEM is the ITEM to be inserted. CIR-QUE is array representation of
//circular queue. FRONT and //REAR are pointers and N is the size of array.
1. If (FRONT=0) then
2. Set FRONT to 1
3. Set REAR to 1
4. Set ITEM to FRONT of CIR-QUE
5. Else
6. Set REAR=(REAR+1)mod n
7. If(FRONT=REAR) then    //circular queue is  physically empty
   Write "circular queue is logically full but physically empty"
8. Set ITEM to REAR of CIR-QUE
9. End
```

Algorithm 7.6: DELETE-CQ(CIR-QUE,FRONT,REAR,N,ITEM)

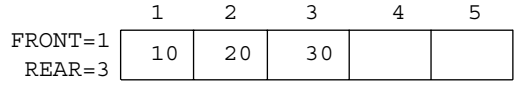
```
1. If(FRONT=0)then
   Write "Queue is empty"
2. Else
3. Set FRONT=(FRONT+1)mod n          //UPDATE FRONT
4. Set ITEM the FRONT ELEMENT
5. End.
```

The following example explains the working of a circular queue

- Initially the circular queue is empty:



- Insert 10, 20, 30 elements:



- Delete element 10:

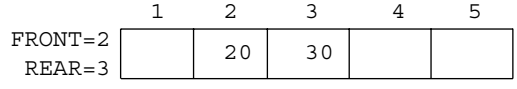
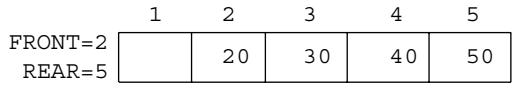
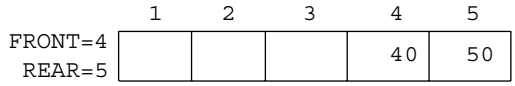


Figure 7.6 Operations on circular queue

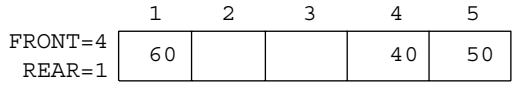
- Insert 40, 50 elements:



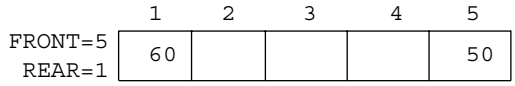
- Delete 20, 30 elements:



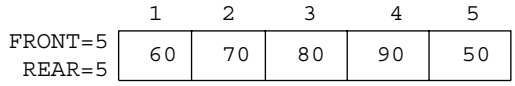
- Insert element 60:



- Delete element 40:

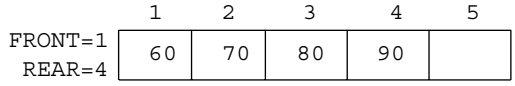


- Insert 70, 80, 90 elements:

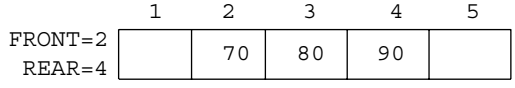


Inserting any element at this stage fails because the circular queue is full.

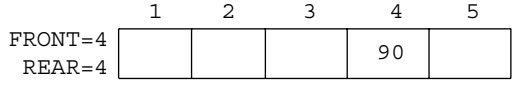
- Delete element 50:



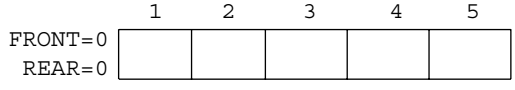
- Delete element 60:



- Delete 70, 80 elements:



- Delete element 90:



Circular queue is empty

Figure 7.6 *Continued*

Programs 7.3 and 7.4 give the implementations of a circular queue using arrays and linked list.

Program 7.3

```
// Array implementation of Circular Queue
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
template<class t>
class queue
{
   int rear,front,n;
   t a[50];
   public:
   queue(int m)
   {
      rear=front=-1;
      n=m;
   }
   void push();
   void pop();
   void display();
};
template<class t>
void queue<t>::push()
{
   t ele;
   if(front==(rear+1)%n)
   {
      cout<<"queue is overflow\n";
      return;
   }
   cout<<"enter the element \n";
   cin>>ele;
   if(front==-1)
      rear=front=0;
   else
      rear=(rear+1)%n;
   a[rear]=ele;
}
template<class t>
void queue<t>::pop()
{
   t ele;
   if(rear==-1)
   {
      cout<<"queue is underflow\n";
      return;
```

```
    }
    ele=a[front];
    if(rear==front)
       rear=front=-1;
    else
       front=(front+1)%n;
    cout<<"the deleted element is"<<ele;
}
template<class t>
void queue<t>::display()
{
    int i;
    if(rear==-1)
    {
       cout<<"queue is underflow\n";
       return;
    }
if(front<=rear)
{
    for(i=front;i<=rear;i++)
    cout<<a[i]<<" ";
}
else
{
    for(i=front;i<n;i++)
    cout<<a[i]<<" ";
    for(i=0;i<=rear;i++)
    cout<<a[i]<<" ";
    }
    cout<<"\n";
}
void main()
{
    int ch,n;
    clrscr();
    cout<<"enter the size of the queue\n";
    cin>>n;
    queue<char>q(n);
    while(1)
    {
       cout<<"\nmenu\n";
       cout<<"1.push\n";
       cout<<"2.pop\n";
       cout<<"3.display\n";
       cout<<"4.exit\n";
       cout<<"enter your choice\n";
       cin>>ch;
       switch(ch)
```

```
      {
         case 1:
            q.push();
            break;
         case 2:
            q.pop();
            break;
         case 3:
            q.display();
            break;
         case 4:
            exit(0);
            break;
         default:
            cout<<"invalid option\n";
            break;
      }
   }
}
```

Output

```
enter the size of the queue
10
menu
1.push
2.pop
3.display
4.exit
enter your choice
1
enter the element
2
menu
1.push
2.pop
3.display
4.exit
enter your choice
1
enter the element
3
menu
1.push
2.pop
3.display
4.exit
```

```
enter your choice
3
2 3
menu
1.push
2.pop
3.display
4.exit
enter your choice
2
the deleted element is 2
menu
1.push
2.pop
3.display
4.exit
enter your choice
4
```

Program 7.4

```
// Linked list implementation of Circular queue
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
template<class t>
class cqueue
{
   struct node
   {
      t data;
      node*link;
   }*head,*temp,*front,*rear;
   public:
   cqueue();
   void push();
   void pop();
   void display();
};
template<class t>
cqueue<t>::cqueue()
{
   head->link=NULL;
   rear=NULL;
   front=NULL;
}
template<class t>
```

```
void cqueue<t>::push()
   {
      t ele;
      cout<<"enter the element\n";
      cin>>ele;
      temp=(struct node*)malloc(sizeof(struct node));
      if(temp==NULL)
      {
         cout<<"memory allocation error\n";
         return;
      }
   if(front==NULL)
   {
      temp->data=ele;
      head->link=temp;
      temp->link=NULL;
      front=rear=temp;
   }
else
   {
      temp->data=ele;
      rear->link=temp;
      temp->link=front;
      rear=temp;
   }
}
template<class t>
void cqueue<t>::pop()
{
   t ele;
   if(front==NULL)
   {
      cout<<"linked list is empty\n";
      return;
   }
   if(front==rear)
   {
      ele=front->data;
      front=rear=NULL;
   }
   else
   {
      ele=front->data;
      rear->link=front->link;
      temp=front;
      front=front->link;
      free(temp);
   }
   cout<<"the deleted element is"<<ele<<endl;
```

```
}
template<class t>
void cqueue<t>::display()
{
   if(rear==NULL)
   {
      cout<<"linked list is empty\n";
      return;
   }
   for(temp=front;temp!=rear;temp=temp->link)
   {
      cout<<temp->data<<" ";
   }
   cout<<temp->data<<" ";
}
void main()
{
   int ch;
   cqueue<int>q;
   clrscr();
   while(1)
   {
      cout<<"\nmenu\n";
      cout<<"1.push\n";
      cout<<"2.pop\n";
      cout<<"3.display\n";
      cout<<"4.exit\n";
      cout<<"enter your choice\n";
      cin>>ch;
      switch(ch)
      {
         case 1:
            q.push();
            break;
         case 2:
            q.pop();
            break;
         case 3:
            q.display();
            break;
         case 4:
            exit(0);
            break;
      }
   }
}
```

Output

```
menu
1.push
2.pop
3.display
4.exit
enter your choice
1
enter the element
2
menu
1.push
2.pop
3.display
4.exit
enter your choice
1
enter the element
3
menu
1.push
2.pop
3.display
4.exit
enter your choice
3
2 3
menu
1.push
2.pop
3.display
4.exit
enter your choice
2
the deleted element is 2
menu
1.push
2.pop
3.display
4.exit
enter your choice4
```

7.5 DEQUE

Another type of queue is deque, which is pronounced as "deck" or "deque". In deque, insertion and deletion operations can be performed on both sides of the queue. The term deque has emerged from double-ended queue. Figure 7.7 resembles the deque.

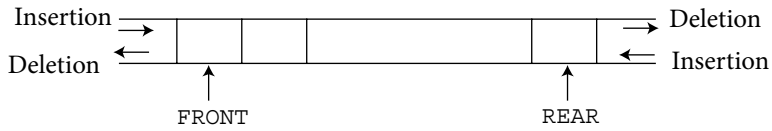
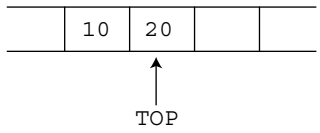
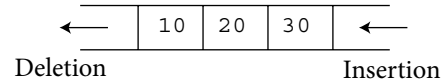


Figure 7.7 Deque

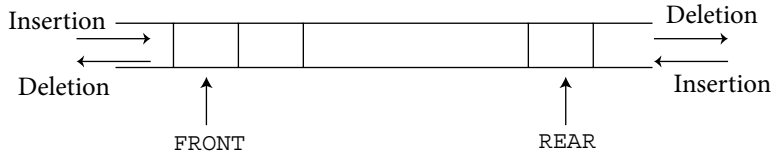
Figure 7.8 illustrates that a deque is the general form of both stack and queue representation, i.e. the deque can be used as a stack or a queue.



(a) Stack



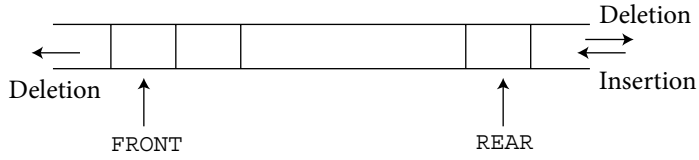
(b) Queue



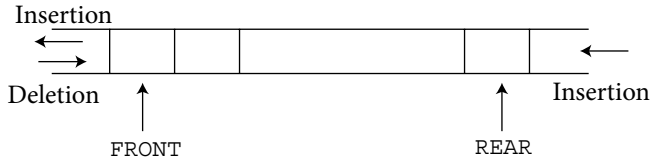
(c) Deque

Figure 7.8 Comparison of stack, queue, deque

Deque's are of two types *input restricted* deque and *output restricted* deque. When insertion of elements into the deque is restricted to only one end and deletion can be done in both directions, such a deque is called input restricted deque. In an output restricted deque, insertion of elements can be done in both directions, but deletions can be done at only one end. Commonly deques are implemented using circular arrays. Figure 7.9 shows the different types of deques.



(a) Input restricted deque



(b) Output restricted deque

Figure 7.9 Types of deques

7.5.1 Operations on a Deque

Four operations can be performed on deque. They are

- PUSH-DEQ(ITEM) inserts ITEM at front end of Deque.
- POP-DEQ() deletes item at front of Deque.
- INSERT-DEQ(ITEM) inserts ITEM at REAR end of Deque.
- DELETE-DEQ() deletes ITEM from REAR end of Deque.

The algorithm for POP-DEQ() and INSERT-DEQ(ITEM) operations are same as DELETE-CQ() and INSERT-CQ(), respectively.

Consider DEQ[1:N] as a deque represented as circular array of size N. The following are the algorithms for insertion at the *front* end and deletion at the *rear* end.

Algorithm 7.7: PUSH-DEQ(DEQ,FRONT,REAR,ITEM)

```
      //Check whether FRONT is at left extreme
1.  If(FRONT==1) then
2.  Set Temp=N
3.  ELSE   //check whether FRONT is at extreme right or deque empty
4.  If(FRONT=N) OR (FRONT=0) then
5.  Set Temp=1
6.  Else
7.  Set Temp=FRONT-1   //FRONT at middle position
8.  If(Temp=REAR) then
9.  Write  "deque is full"
10. Else
11. FRONT=Temp
12. DEQ[FRONT]=ITEM
13. END.
```

Algorithm 7.8: DELETE-DEQ(DEQ,FRONT,REAR,ITEM)

The element is deleted from the rear end and is assigned to ITEM

```
1.  If(FRONT=0) then
2.  Write "deque is empty"
3.  Else
4.  If(FRONT=REAR) then          //DEQ contains right element
5.  Set ITEM=DEQ(REAR)
6.  Set FRONT=REAR=0            //Deque becomes empty
7.  Else
8.  If(REAR=1)then               //check whether REAR is at extreme left
9.  Set ITEM=DEQ[REAR]
10. Set REAR=N
11. Else
12. If(REAR=N)then               //check for REAR at extreme right
    Set ITEM=DEQ[REAR]
    Set REAR=1
13. Else                         //REAR at middle position
```

```
    Set ITEM=DEQ[REAR]
    REAR=REAR-1
14. END
```

Figure 7.10 explains the working of a deque. Suppose `DEQ[1,6]` is Deque. The operations on the deque are as follows.

- Initially deque is with three elements:

	[1]	[2]	[3]	[4]	[5]	[6]
FRONT=3 REAR=5			10	20	30	

- Insert `40` at front and `50` at rear:

	[1]	[2]	[3]	[4]	[5]	[6]
FRONT=2 REAR=6		40	10	20	30	50

- Delete `30, 50` from the rear end:

	[1]	[2]	[3]	[4]	[5]	[6]
FRONT=2 REAR=4		40	10	20		

- Insert `60, 70, 80` at the front end:

	[1]	[2]	[3]	[4]	[5]	[6]
FRONT=5 REAR=4	60	40	10	20	80	70

- Insert `90` at the rear end, insertion fails because the deque is full: `FRONT=REAR+1`
- Delete two elements from the front end:

	[1]	[2]	[3]	[4]	[5]	[6]
FRONT=1 REAR=4	60	40	10	20		

Figure 7.10 Operations on deque

In the example in Figure 7.10, observe that insertion at the front end decrements `FRONT` by `1(mod)N` and insertion at the rear end increments `REAR` by `1(mod)N`. Deletion at the front end increments `FRONT` by `1(mod)N` and deletion at the rear end decrements `REAR` by `1(mod)N`, where `N` is the size of the deque.

7.6 APPLICATIONS OF QUEUES

In general, queues are often used as "waiting lines." A few examples where queues would be used are as follows:

(1) In operating systems for controlling access to shared system resources such as printers, files, communication lines, disks and tapes.

(2) Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

(3) Computer systems must often provide a "holding area" for messages between two processes, two programs or even two systems. This holding area is usually called a "buffer" and is often implemented as a queue.
(4) In the situation where there are multiple users or in a networked computer system, a printer is shared among all users. When a file is requested to be printed, the request is added to the print queue. When the request reaches the front of the print queue, the file is printed. This ensures that only one person at a time has access to the printer and it is given on a first-come-first-served basis.
(5) A file server in a computer network handles file access request from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.
(6) Most computers have only one processor, so only one user at a time can be served. Entries for the other users are placed in a queue. Each entry gradually advances to the front of the queues as the users receive service. The entry at the front of the queue is the next to receive service.
For example:
   i. Modeling computer network
   ii. Each device is connected to the network by a 'controller'.
   iii. Before a controller sends a message it checks whether the network is busy or not. If busy, it waits otherwise it sends a message.
   iv. When two controllers attempt to send a message at the same time the messages collide and the receiving controllers ignore it. Similarly, the sending controllers recognize the problem and resend their messages.
(7) For simulation of real-world situations. For instance, a new bank may want to know how many tellers to install. The goal is to service each customer within a reasonable wait time, but not have too many tellers for the number of customers. To find out a good number of tellers, they can run a computer simulation of typical customer transactions using queues to represent the waiting customers.
(8) When placed on hold for telephone operators. For example, when a toll-free number for a bank is dialed, a recording that says, Thank you for calling A-1 Bank. Your call will be answered by the next available operator. Please wait, may be heard. This is a queuing system.

7.6.1 Simulation of Time-sharing System

In time-sharing system, various users will share a single computer simultaneously. Since, there is only one CPU it is shared among *n* users. The processor is shared by allowing one user program to execute for a shorter period of time and then another user program is allowed to execute and followed by another, etc. until it is returned to the point of execution in the initial user program. This cycle is continued repeatedly on all active user programs. This technique of sharing the processor among various users is known as **time sharing**. The following example illustrates the working of a time-sharing system.

Suppose three user programs U_1, U_2, U_3 start their online session at their terminals, which are connected to single system. The table below shows the time periods of the three user programs:

Session start	Program ID	Requested CPU time period
0	U_1	3, 7, 2
1	U_2	1, 2, 1, 1
2	U_3	3, 5, 1

After logging on the system at time 0, initially user program U_1 is allotted 3 sec of CPU time before it receives a message at its terminal. While receiving the message, the user thinks for some time and then gives a new input for the program. Further 7 secs of CPU time is required before another message is printed on the user U_1 terminal. Again user thinks for some time and gives a new input for the program. Let the time period for thinking and giving input be the *delay time*. Finally, user program U_1 uses 2 secs of CPU time and then logs off the session.

In the above discussions, ignore the fact that the other two user programs U_2, and U_3 logged on at time 1 and 2, respectively, and they need the processor immediately. Since there is only one CPU, the user programs U_2 and U_3 will be waiting until U_1 finishes its first requested CPU time period of 2 secs. To indicate that the user programs U_2 and U_3 are waiting for the processor, place the program ID of U_2 and U_3 in a queue after U_1. Hence the queue at time 2 will look as in Figure 7.11.

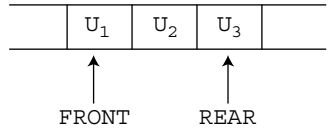


Figure 7.11 Queue at time 2

When U_1 completes its first requested CPU time period, a scheduling strategy must be used to allocate U_2 to CPU while U_1 is thinking during *delay time*. When U_2 finishes its CPU time period U_3 should be allocated to execute. This type of scheduling technique is often called as first-come-first-serve (FCFS) scheduling.
The rules for FCFS scheduling for this applications are as follows:

1. When a program requests the CPU time period, it is placed into the queue.
2. The program pointed by the front pointer is the one that is currently executing. It remains at the front of the queue for its entire CPU time period.
3. When a program completes its requested CPU time, it is removed from the queue and is not placed again into the queue until further request is made.

Following the above rules three user programs can be scheduled. The problem here is to simulate the activity of such a time-sharing system. The simulation must place users in the queue according to rule 1 and remove them as in rule 3.

The purpose of such a simulation is to obtain the information about the efficiency of the time-sharing system. The data related to this is as below:

$$\text{CPU utilization (for each user)} = \frac{\text{Total CPU time}}{\text{Total session time}} * 100$$

Total user waiting time (for each user) = total time – total CPU time – total delay time
Total delay time (for each user) = time delay period*(number of CPU request – 1)

7.6.2 Queue ADT

A **queue** is a collection of elements, or items, for which the following operations are defined:

1. `Create(Q)` creates an empty queue `Q`.
2. `IsEmpty(Q)` returns whether the queue is empty or not.
3. `IsFull(Q)` returns whether the queue is full or not.
4. `Enqueue(Element X,Q)` inserts an element `X` on the rear side of the queue.
5. `Dequeue(Q)` removes the element pointed to by the front end of the queue.

SUMMARY

- A queue is a non-primitive linear data structure in which elements are inserted from one end and deleted from another end. It is also called as first in first out, i.e. the element is inserted first will be deleted first. The end into which elements are inserted is called the rear end and the end from which elements are deleted is called the front end.
- A queue basically supports two operations, insert and delete.
- A queue has many applications, like it is used in operating systems for job scheduling, in networks, to check whether a given string is a palindrome or not, etc.
- Types of queues are circular queue and doubly ended queue.
- Insertion of elements can also be done at the rear when its value is the maximum size in circular queue.
- In deque insertion and deletion operations can perform on both sides of the queue, so it is called as double-ended queue.
- Input-restricted deque and output-restricted deque are the two types of deque.

EXERCISES

Fill in the Blanks

1. Queues are also knows as _____ structures.
2. _____ are dynamic queues.
3. _____ are the types of queues.
4. Queue is said to be full when _____ .
5. _____ is the drawback of linear queues.
6. _____ are knows as deques.
7. The advantage of circular queue _____ .

Multiple-choice Questions

1. The data structure that allows deleting data elements from front and inserting at rear is _____.
   a. Stacks
   b. Queues
   c. Deques
   d. Binary search tree

2. Identify the data structure that allows deletions at both ends of the list but insertion at only one end.
   a. Input-restricted deque
   b. Output-restricted deque
   c. Priority queues
   d. None of the above

3. A difference between a queue and a stack is
   a. Queues require dynamic memory, but stacks do not.

b. Stacks require dynamic memory, but queues do not.
c. Queues use two ends of the structure, stacks use only one.
d. Stacks use two ends of the structure, queues use only one.

4. If the characters 'D', 'C', 'B', 'A' are placed in a queue (in the same order), and then removed one at a time, in what order will they be removed?
   a. ABCD b. ABDC
   c. DCAB d. DCBA
5. Which of the following is an application of queue?
   a. Recursion b. CPU scheduling
   c. Sorting d. Matrices

Short-answer Questions

1. Define queue with an example.
2. Define circular queue with an example.
3. What do you mean by a priority in a queue.
4. Give two disadvantages of queues.
5. Give two advantages of queues.
6. List any four applications of linear queue.
7. List any four applications of circular queue.
8. Mention two differences between queues and stacks.
9. Give the specification of data structure queue.
10. A circular queue has a size of 5 has 3 elements `20, 40, 60` where `FRONT=0` and `REAR=2`. Show the necessary diagrams. What is the value of `FRONT` and `REAR` after each of these operations?
    (i) Insert an item 10 (ii) Delete an item

Essay Questions

1. Define an input-restricted queue. Show with a suitable C++ program the implementation of a queue using the operations of a input-restricted queue.
2. Explain the working of ordinary queue with primitive operations and show the underflow and overflow conditions using an example.
3. What is the disadvantage of an ordinary queue and how it is overcome? Explain with an example.
4. What is a circular queue? Explain the implementation of a circular queue using linked list. Write an algorithm for the same.
5. How are a queue of stacks, a stack of queues and a queue of queues implemented? Write routines to implement the appropriate operations for each of these data structures.
   (i) insertion (ii) deletion (iii) display
6. Write a program for implementing deque.

Chapter 8

Dictionaries

This chapter discusses dictionaries and their representations in various methods such as linear list representation, skip list representation and hash table representation. In this chapter the reasons and situations of collision occurrences and techniques to overcome the collisions are explained. Comparisons of chaining and open addressing, applications and ADT of dictionary are also explained in detail.

8.1 DICTIONARIES

Dictionary contains data elements as pairs of the form (k,v), where k is the key and v is the value. The field key must be unique and is used to identify the data elements uniquely. No two pairs are allowed to have the same key.

A dictionary with duplicates is a dictionary which allows two or more (key, value) pairs with the same key. To solve the ambiguity while performing operations on elements with duplicate keys, dictionaries should frame some rules.

A dictionary can either be ordered or unordered. In ordered dictionaries, all the data elements are placed either in ascending or in descending order of the keys. Here sequential access of (key, value) pairs is allowed.

In an unordered dictionary, no particular order of data elements is maintained. Here, data elements or (key, value) pairs are accessed randomly.

Examples

- A dictionary that stores student records
- A general dictionary of words
- A computer dictionary similar to dictionary of words
- Symbol table is a dictionary with duplicates which is used by a compiler

8.2 LINEAR LIST REPRESENTATION

An ordered dictionary can be represented as an ordered linear list. In this every element or node is a (key, value) dictionary pair stored generally in ascending order of the keys. Sorted array list and sorted chain are used for this representation.

This representation uses binary search method and hence the time complexity of search operation is O(log n). Insertion and deletion operations perform search operation internally before either inserting an element or deleting an element. Insertion requires additional O(n) time to shift O(n) pairs to make room for the new (key, value) pair. Similarly, the deletion also requires additional O(n) time to shift O(n) pairs forward to fill the gap left by the deleted pair. Sequential access allows O(1) time to access every (key, value) pair, which are stored in ascending order.

8.3 SKIP LISTS REPRESENTATION

William Pugh developed skip lists in 1990. Skip lists are probabilistic alternatives to balanced trees like AVL trees or splay trees. Skip lists are balanced by considering a random number generator.

The balanced trees perform very efficiently, O(log n) for the operations such as search, insert and delete over random input data. But, for the ordered input data they are less efficient, O(n). To balance the tree and to assure good performance, the trees are rearranged by the balanced tree algorithms. Random input data is impractical in many cases where queries are to be answered on-line.

When a sorted list is considered, the performance of search, insert and delete operations is poor, O(n). Because, to find a relevant node the list must be scanned node-by-node starting from the head. The basic idea behind the skip lists is to allow the list to be scanned in bigger steps. Skip lists are sorted linked lists but differ in two points. They are

- In a skip list, nodes have many 'next' references known as forward references, whereas in an ordinary list, nodes have only one 'next' reference.
- For a given node the number of forward references is probabilistically determined.

In a linked list shown in Figure 8.1 to find an element every node might be examined. In a sorted ordered list, every second node of the list also points to a node two positions ahead to it in the list as shown in Figure 8.2. A maximum of n/2+1 nodes are to be searched where n is the length of the list. When every fourth node also points to a node four position ahead to it as shown in Figure 8.3. Then it requires a maximum of n/4+2 nodes to be searched. Similarly, if every $(2^i)^{th}$ node also points to 2^i nodes ahead to it as shown in Figure 8.4. Then the number of nodes to be searched can be reduced to $\log_2 n$. Skip lists are very efficient for search operations, but for insertion and deletion this would be inefficient. Maximum level (M level) will be $\log_{1/p} n$ where p is the probability and p=1/2.

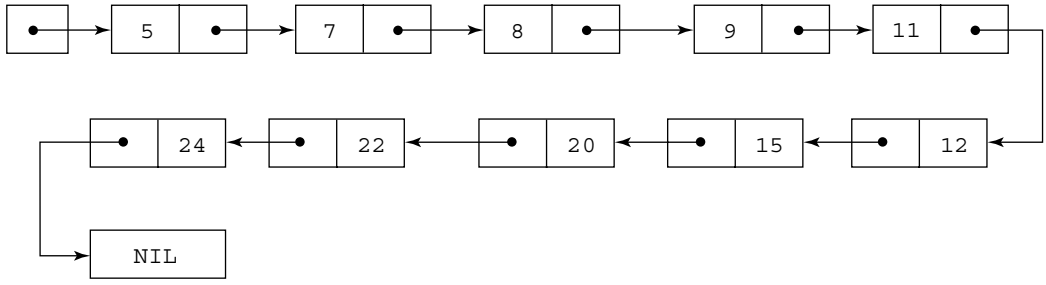


Figure 8.1 Linked list

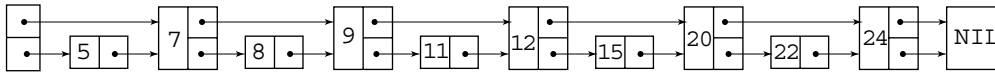


Figure 8.2 Skip list with level 2

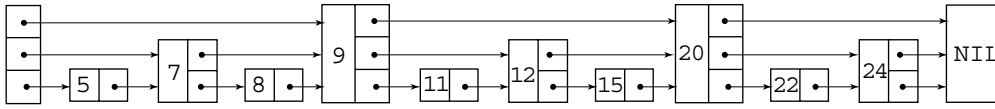


Figure 8.3 Skip list with level 3

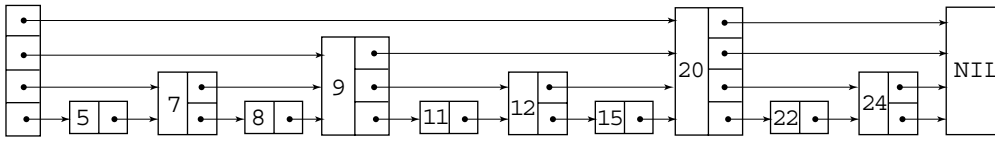


Figure 8.4 Skip list with level 2^i

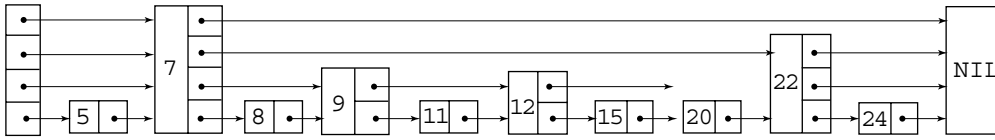


Figure 8.5 Skip list with level 2^i

The level of the node is known by the number of forward pointers it has. A node with `k` forward pointers is called a level `k` node. The levels of nodes can be distributed when every $(2^i)^{th}$ node points to 2^i nodes ahead in the list, as 50% of the nodes are at level 1, 25% of the nodes are at level 2, 12.5% of the nodes are at level 3 and so on (Figure 8.4). Levels of the nodes may be chosen randomly but with same proportions as shown in Figure 8.5. That is, the number of nodes at different levels must be considered irrespective of their positions. The i^{th} forward pointer of a node points to the next node of level `i` or higher instead of pointing to 2^{i-1} nodes ahead.

Now insertion or deletion does not require restructuring but requires only the local modifications. So, the level of the node chosen during its insertion need not be changed.

8.3.1 Operations

A dictionary or a symbol table can be efficiently represented as a skip list. Basic operations performed on them are searching, insertion and deletion. Other additional operations like finding the minimum key, finding the next key, etc. can also be easily supported by the skip list.

A node at level `i` has `i` forward pointers, indexed from 1 to `i`. Level of the list is the maximum level existing in the list. When the list is empty the maximum level is 1. The header of the list holds forward pointers at levels 1 through the maximum level.

Initially, an element NIL is allocated with a value, i.e. greater than any legal value. All the levels of all the skip lists terminate at NIL. A new list is initialized with level as 1, and all the forward pointers of the corresponding list header node point to NIL.

8.3.2 Searching

Searching for an element begins by following the highest level pointer. If the element is found then the search is successful. If the end of the list is found or when an element that is greater than the element being searched is found then no more progress can be made at the present level. Now, the search moves down to the next lower level and continues till the element is found or level 1 pointers are followed to find the end of the list or to find an element greater than the searching element.
Algorithm 8.1 explains the search procedure

ALGORITHM 8.1: SEARCH(list, key)

```
1. Set p=list[header]
2. Loop for i=list[level] to 1 do
   1. Loop while p[forward[i] of key]<key do
   2. Set p=p[forward[i]]
3. Set p=p[forward[1]]
4. Check whether p[key]=key then
       Return p[value]
5. Else
       Return failure
6. End
```

A vector `i.p` is used to hold the pointer to the rightmost node of level `i` or higher, i.e. to the left of the location where insertion or deletion takes place. This can also be referred to as the insertion point.

8.3.3 Insertion

Dictionaries work with key–value pairs. During insertion the key is searched; if it is found then the value is updated with the new value, otherwise a node is created with random level and the value is inserted. Algorithm 8.2 explains the insertion process. Figures 8.6 and 8.7 show the insertion procedure.

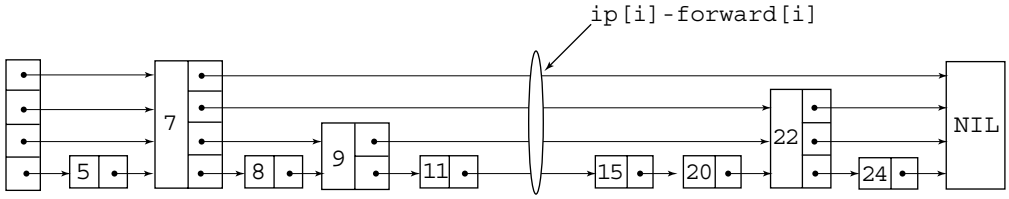


Figure 8.6 Original skip list before insertion

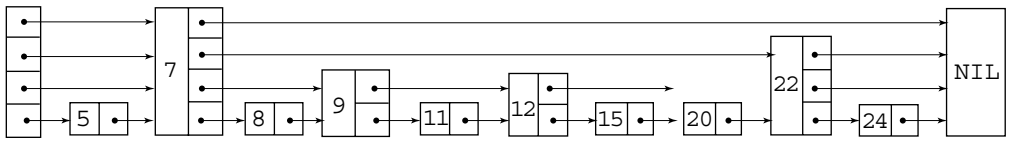


Figure 8.7 Skip list after insertion

Algorithm 8.2: SINSERT(list, key, Item)

```
1. Set p=list[head]
2. Loop for i=list[level] to 1 do
   1. Loop while p=key[forward[i]]<key do
   2. Set p=p[forward[i]]
   3. Set ip[i]=p
3. Set p=p[forward[1]
4. Check whether p[key]=key then
   Set p[value]=Item
5. Else
  1. Set |v|=rdLevel()
   2. Check whether |v|>list[level] then
      1. Loop for i=list[level+] to |v| do
         Set ip[i]=list[head]
      2. Set list[level]=|v|
      3. Set p=MakeNode(|v|, key, value)
      4. Loop for i=1 to level do
         Set p[forward[i]]=ip[i] of forward[i]
         Set ip[i] of forward[i]=p
6. End
```

8.3.4 Deletion

Deletion process also requires to search for the key to delete the corresponding key–value pair. After identifying it the vector `i.p` points to the node that is before (left of) the deleting node. The link of the following node which was with the deleted node should be assigned to the node that was left to the deleted node. Algorithm 8.3 explains the deletion process.

Algorithm 8.3: DELETE (list, key)

```
1. Set p=list[header]
2. Loop for i=list[level] to 1 do
   Loop while p[forward[i]] of key<key do
   Set p=p[forward[i]]
   Set ip[i]=p
3. Set p=p[forward[i]]
4. Check whether p[key]=key then
   Loop for i=1 to list[level] do
      i. Check whether ip[i] of forward[i]≠p then
     ii.Break
        free(p)
3. Loop while list[level]>1 and list[header] of forward[list[level]]=NIL do
4. Set list[level]=list[level-1]
5. End
```

Algorithm 8.4 explains the random level generation.

Algorithm 8.4: rdLevel()

```
1. Set |v|=1
2. Loop while random( )<r and |v|<MLevel do
```

```
3. Set |v|=|v|+1
4. return |v|
```

Many applications are naturally represented with skip lists than trees. Skip lists have better speed by a constant factor when compared with balanced trees and self-adjusting trees. With every node no balance or priority information is to be stored. Skip lists can be configured easily so that for every element it requires on an average only $1^1/_3$ pointers or even less. Hence, skip lists are also efficient in terms of space.

8.4 HASH TABLE

Hash table is an efficient method of implementing dictionaries. A hash function is used to map dictionary pairs into the positions of a hash table. A bucket array is used for a hash table. It is an array of size n. Each and every cell in the array is treated as a bucket and it is used to hold the key–value pairs. When an item is looked up, its key is hashed to find the appropriate bucket. Then, the bucket is searched for the right key–value pair. If a bucket can store only one element and when two keys are mapped to the same bucket then collision occurs. Efficiency of a hash function lies in avoiding collisions. Using the hash function all the operations on the hash table such as searching for a key–value pair, inserting a key–value pair or deleting a key–value pair are performed.

8.4.1 Hash Functions

When a table with n elements and m positions, where $n \leq m$ is considered then to assign positions to the elements n number of hash functions can be used.
Some of the methods that are used for creating a hash function are as follows:

1. The division method
2. The multiplication method
3. Truncation
4. Folding
5. Extraction

1. The division method: Map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \mod m.$$

Example: If table size m = 12, key k = 100 then

$$h(100) = 100 \mod 12$$
$$= 4$$

Values of m such as $m = 2^p$ should be avoided. Good values for m are primes, and are not too close to exact powers of 2.

2. Truncation: Ignore some digits of the key and use the rest as the array index. When the keys are alphabets, then their numerical equivalents can be considered. The problem with this approach is that there may not always be an even distribution throughout the table.

Example: If student ID's are the key 928324312 then from the 9-digit number select just the 4^{th} and 8^{th} position digits as the index, i.e. 31 as the index.

3. Folding: Partition the key into several pieces, i.e. two or three or more parts. Each of the individual parts is combined using any of the basic arithmetic operations such as addition or multiplication. The resultant

number could be conveniently manipulated, for example, truncated, to get the index of the key that is to be stored. Folding gives better spread of keys across the hash table. This process is fast and simple specially when bit patterns are used in place of numerical values. In the bit-oriented versions exclusive–or operations are applied.

Example: Consider a number 123456. Partition the number into three parts, 12 | 34 | 56. Add the three parts 12 + 34 + 56 = 112. Truncating the result gives 12, which is the index for the hash table to store the number 123456.

4. Extraction: In this method computing the address uses only a part of the key. From a student id 928324312 the method may use first four digits, 9283, or the last four digits, 4312, or the combination of the first two and last two digits, 9212, or any other combination. Every time only some part of the key is used. It is sufficient for hashing if this part is carefully choosen, only when the omitted part distinguishes the keys in an insignificant manner. For example, the starting digits of the ISBN code are the same for all the books of the same publisher. So, they must be excluded when computing the address if the data table has only the books from a single publisher.

8.5 COLLISIONS

When a hash function maps two different keys to same location then collision occurs. Thus the corresponding records cannot be stored in the same location. An array with four records, each with two fields, one for the key and one to hold data is considered. Let the hashing function be a simple modulus operation, i.e. array index is computed by finding the remainder of dividing the key by 4:

Array Index: = key MOD 4

Then key values 9, 13, 17 will all hash to the same index. When two (or more) keys hash to the same value, a **collision** is said to occur (Figure 8.8).

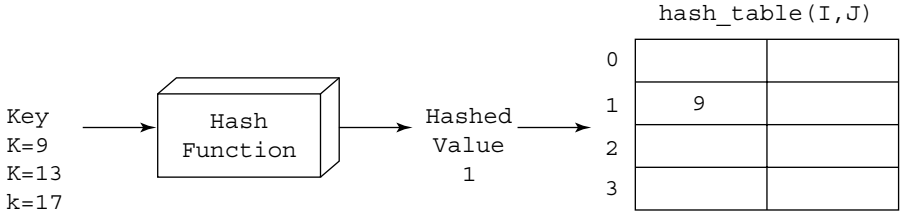


Figure 8.8 Collision

Collision resolution techniques: The two most popular techniques for collision resolution are:

1. Separate chaining—An array of linked list implementation
2. Open addressing (or closed hashing)—Uses an array based implementation

8.5.1 Separate Chaining

The hash table is implemented as an array of linked lists. To insert an item into the table, it is appended to the corresponding linked lists. The linked list to which it is to be appended is determined by hashing the inserting item. This technique is known as **chaining** or **separate chaining** or **open hashing** because each hash table element is a separate chain (linked list). Its data field contains either keys or references to keys. This technique allows one to perform all operations easily without any collision. Separate chaining is depicted in Figure 8.9:

- Inserting an item `e` that hashes at index `i` is simply inserted into the linked list at position `i`. Synonyms are chained in the same linked list.
- Retrieval of an item `e` with hash address `i` is simply retrieving it from position `i` of the linked list.
- Deletion of an item `e` with hash address `i` is simply retrieving it from position `i` of the linked list and deleting or disconnecting it from the linked list (Figure 8.9).

Insert: A_5, A_2, A_7, B_5, A_9, B_2, B_9, C_2

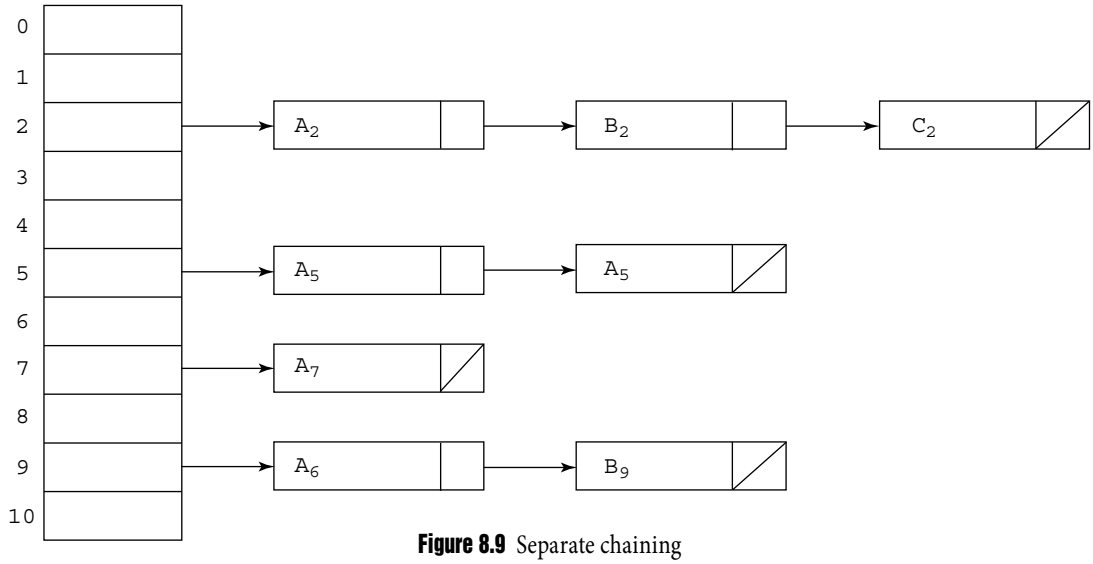


Figure 8.9 Separate chaining

Example: Load the keys 23, 13, 21, 14, 7, 8 and 15, in the same order, in a hash table of size 7 using separate chaining (Figure 8.10) with the hash function **h(key)=key%7**.

h(23)=23%7=2
h(13)=13%7=6
h(21)=21%7=0
h(14)=14%7=0 (collision)
h(7)=7%7=0 (collision)
h(8)=8%7=1 (collision)
h(15)=15%7=1 (collision)

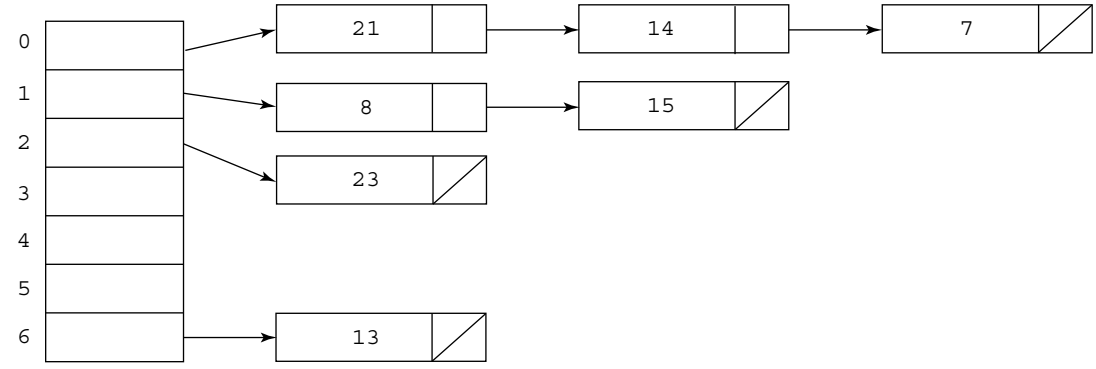


Figure 8.10 Example of separate chaining

Operations on separate chained hash table: The very frequently used search, insert and delete operations on separate chained hash table are discussed here.

Searching: Hash function of the element that is to be searched should be computed. Access the bucket with corresponding hash value and proceed to search the chain of nodes sequentially. If the element is found, then search is successful, else it is an unsuccessful search.

Algorithm 8.5: Separatechain-hash-search(ht, b, e)

```
/* Search an element e in separate chained hash table*/
/* ht is the hash table, an array of pointer to buckets, b is the bucket
   number and e is the element to be searched*/
hf=h(e)   /*h(e) is the hash function on e*/
t=ht(hf)  /*t is the pointer to the first node in the chain*/
While (DATA(t)≠e and t≠NULL) do
   t=LINK(t);
end while
If(DATA(t)==e) then
 print("Search is Successful");
If(t==NULL) then
print("Search is Unsuccessful");
End
```

Insertion: Inserting an element `e` into the hash table also requires computing hash function on the element to determine the bucket. After finding the corresponding bucket it is similar to inserting an element into a singly linked list.

If the elements in each chain are maintained in either ascending or descending order, then it will be less expensive to perform all the operations.

Deletion: Deleting an element `e` from the hash table also requires computing hash function on the element to determine the bucket. After finding the corresponding bucket it is similar to deleting an element from a singly linked list.

Performance analysis: The length of the chain of nodes corresponding to a bucket decides the complexity of separate chained hash tables.

Its best case complexity of search operation is O(1). The worst case complexity is O(`n`). This occurs when all the `n` elements are mapped to the same bucket and the searched element is the last element in the chain of `n` nodes.

8.5.2 Open Addressing

In open addressing all the record entries are stored in the bucket array. The locations of the hash table are termed as buckets. Each bucket is portioned into **slots** as shown in Figure 8.11. The buckets are examined whenever a new entry has to be inserted, starting with hashed-to slot using some probe sequence until an empty slot is found.

To search for an entry all the buckets are scanned in the same sequence until the entry is found or an empty slot is found which indicates that there is no such entry in the table.

The location or address of an item is not determined by the hash value with open addressing. Each location in an array will be in the EMPTY, DELETED or OCCUPIED state. If the state of a location is OCCUPIED then

it contains key and data otherwise it does not have any value. Initially all the locations in the array are in the EMPTY state. When an item at a particular location is deleted then its state will be DELETED, not EMPTY (the item may be deleted but it is not removed until some other element is inserted in that place). The need of these three states is explained by Algorithm 8.6, which inserts an item into a table.

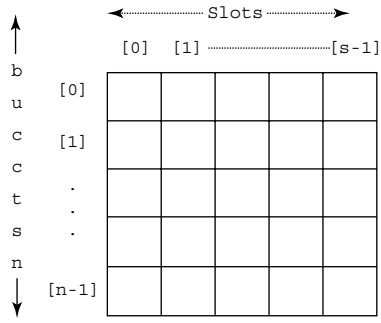


Figure 8.11 Hash table

Advantages of separate chaining: Separate chaining has several advantages over open addressing:

- Collision resolution is simple and efficient.
- The hash table can hold more elements without the large performance deterioration of open addressing (the load factor can be 1 or greater).
- The performance of chaining declines much more slowly than open addressing.
- Deletion is easy—no special flag values are necessary.
- Table size need not be a prime number.
- The keys of the objects to be hashed need not be unique.

Disadvantages of separate chaining:

- It requires the implementation of a separate data structure for chains, and a code to manage it.
- The main cost of chaining is the extra space required for the linked lists.
- For some languages, creating new nodes (for linked lists) is expensive and slows down the system.

ALGORITHM 8.6:

```
1. Find the location at which item is to be stored, L=H(key(Item))
2. If location L is not OCCUPIED then insert Item.
3. If location L is OCCUPIED find another location using some probe sequence.
   Set location L to  ITEM  and repeat steps 2 and 3
4. End
```

The following are the well-known probe sequences:
Linear probing: in which the interval between probes will be fixed usually to 1.
Quadratic probing: in which interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the hash function.
Double hashing: in which the interval between probes is obtained by using another hash function.

Linear probing: Linear probing is a technique for resolving hash collisions of values of hash functions by searching the hash table sequentially for a free location. This is done by making use of two values where one is the starting value and another is the interval between successive values. The second value which is the same for all keys repeatedly adds to the starting value until free location is found or entire table is traversed.

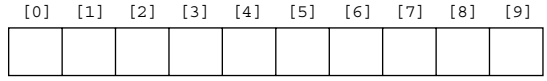
For a given ordinary hash function `h(x)`, the linear probing function will be

```
h(x)=(h(x)+i)%mod n
```

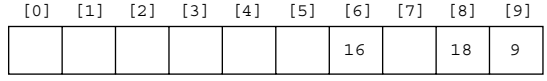
where `h(x)` is the starting value, `i` is the value added repeatedly to the starting value and `n` is the size of the hash table. When the `i` value is 1, the linear probing gives good memory catching through good locality of reference but also results in clustering. Consider the following example.

Let the table size be 10, keys are 2-digit integers and `h(x)=x mod 10`.

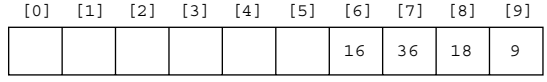
- Initially all the locations are empty



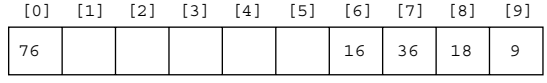
- While inserting 16, 18 and 9, there will be no collisions
  `h(x)= x` mod10
  h(16)=16 mod 10=6, so insert 16 at the 6th location.
  h(18)= 18 mod 10=8, so insert 18 at the 8th location.
  h(9)= 9 mod 10=9, so insert 9 at the 9th location.



- Now insert 36, h(36)=36 mod 10= 6, collision occurs because the state of location 6 is OCCUPIED. Using linear probing, the next free location will be 6+1 mod 10=7 where the value 36 is to be stored. The state of location 7 is EMPTY, so 36 can be inserted into 7th location.



- Now insert 76. h(76)=76 mod 10=6, collision occurs because the state of location 6 is OCCUPIED. Then find another location, i.e. (6+1 mod 10) =7 which is in an OCCUPIED state. Then try next location as (7+1 mod 10) = 8, OCCUPIED state. Finally try the location 0 as (9+1 mod 10)=0 which is in EMPTY state, so insert 75 at the 0th location.



The disadvantage of linear probing is primary clustering. Observe the above example when further insertion of 86, 96 and 56 is made, it causes more number of collisions, i.e. each one would probe exactly the same partitions as its predecessors. This is known as primary clustering. Clustering leads to very inefficient operations because it causes more number of collisions. To eliminate clustering each different key should probe the table in a different order. Algorithm 8.7 denotes insertion.

Algorithm 8.7: LINSERT(key, p)

```
1. Set i=h(key)
2. Set last=(i+m-1)%m;
3. Repeat  while(i!=last&&!empty(p[i])&&!deleted(p[i])&&p[i].k!=key)
4. Set i=(i+1)%m;
5. Check whether empty p[i] or deleted p[i] then
6. Set p[i].k=key                          //insert here
7. Set n=n+1
8. Else write "table overflow or key already in table"
```

Search: To search an item in the hash table that uses linear probing, start at `h(k)` and probe consecutive locations until one of the following occurs:

1. Item is found
2. An empty cell is found or
3. Entire table has been scanned

Algorithm 8.8 searches an item in the hash table using linear probing.

Algorithm 8.8: LP-SEARCH(k)

```
1. Set i=h(k), p=0
2. Repeat until P=N
3. Set c=A[i]
4. Check whether c=NULL then write "key not found"
5. Else check whether c.key()=k then write "key found"
6. Else
7. Set i=(i+1)mod N
8. Set p=p+1
9. End step 2 loop
```

Example: Perform the operations given below, in the given order, on an initially empty hash table of size 10 using linear probing with `i=1` and the hash function: `h(x) = x` mod10:

insert(18), insert(26), insert(35), insert(8), find(15), find(48), delete(35), delete(40), find(8), insert(64), insert(47), find(35).

The required probe sequences are given by `h(key) = (h(key) + i)` mod10 where `i=1`.

The following table shows the result of the above operations:

Operation	Probe sequence	Result
Insert(18)	`h(18)=18` mod `10=8`	SUCCESS
Insert(26)	`h(26)=26` mod `10=6`	SUCCESS
Insert(35)	`h(35)=35` mod `10=5`	SUCCESS
Insert(8)	`h(8)=8` mod `10=8` `h(8)=(8+1)` mod `10`	COLLISION SUCCESS

Find(15)	`h(15)=15` mod `10=5`
	`h(15)=15+1` mod `10=6`
	`h(16)=16+1` mod `10=7`
		COLLISION
		COLLISION
		FAIL because location 7 does not contain 15, i.e. EMPTY status

Operation	Probe sequence	Result
Find(48)	h(48)=48 mod 10=8 h(48)=48+1 mod 10=9 h(48)=49+1mod 10=0	COLLISION COLLISION FAIL because location 0 does not contain 48
Delete(35)	h(35)=35 mod 10=5	SUCCESS because location 5 contains 35 and the status is OCCUPIED. The status is changed to deleted but key 35 is not removed
Delete(40)	h(40)=40 mod 10=4	FAIL no such key exists in location 4 and status is EMPTY
Find(18)	h(18)=18 mod 10=8	The search success location 8 contains key 18
Insert(64)	h(64)=64 mod 10=4	SUCCESS
Insert(47)	h(47)=47 mod 10=7	SUCCESS
Find(35)	h(35)=35 mod 10=5	Fail because location 5 contains 35 but its status is DELETED

The following table shows index values and their status:

Index	Status	Value
0	E	
1	E	
2	E	
3	E	
4	O	64
5	D	35
6	E	26
7	O	47
8	O	18
9	E	8
10	E	
11	E	
12	E	

Quadratic probing: Quadratic probing is another method for resolving collisions in hash tables. It operates by taking the original hash value and adding successive values of a quadratic polynomial to the starting value. Quadratic probing avoids the clustering problem that occurs with linear probing and may also result in probing the same set of alternate cells.

In this method when a collision occurs at location l, probes bucket l+1, l+4, …, l+9 whereas in linear probing probes bucket at locations l+1, l+2, l+3, etc. In general this method probes buckets at locations $(l+i^2)$ mod n, i=1, 2, …. where l is the location of home buckets and n is the number of buckets.

Quadratic probing may also result in probing the same set of alternative cells and this is known as secondary clustering, which occurs when hash table size is not prime. If n is a prime number then quadratic probing probes exactly half of the number of locations in the hash table. Algorithms 8.9 and 8.10 denote insertion and search operations on a table using quadratic probing.

Algorithm 8.9: QINSERTION (key, r)

```
1. Set i=h(key)
2. Set c=0
3. Loop  while(c<m) and (not empty(r[i])) and (not deleted(r[i])) and
   (r[i].k<>key) do
4. Set i=(i+c+1) mod m
5. Set c=c+2
6. End loop
7. check  whether empty(r[i]) or deleted(r[i]) then
8. Set r[i].k=key
9. Set n=n+1
10.else write "table full, or key already in table"
11.End
```

Algorithm 8.10: QSEARCH (key, r)

```
1. Set i=h (key)
2. c=increment(key)
3. Set last=(i+(n-1)*c) mod m
4. Loop while (i<last) and (not empty(r[i])) and (r[i].k<key) do
5. Set i=(i+c) mod m
6. Check whether r[i].k=key then
7. Value=i                    //found(r[i])
8. else
9. Value=-1                   //notfound(key)
10.End
```

Double hashing or rehashing: Double hashing is a technique used in hash table to resolve hash collisions. It uses one hash value as a starting value and continual evolution of an interval until the desired value is found, or an empty location is found or the entire table is searched.

The interval is decided using another independent hash function. So it is named as double hashing. Here the interval depends on the data so that even values mapping the same location will have different sequences that minimize the repeated collisions and the effect of clustering. For a given independent hash functions h_1 and h_2, the i^{th} location in the bucket sequence for value x in a hash table of size n is given as

$$h(x,i)=h_1(x)+i*h_2(x) \mod n$$

Consider the same example as in linear probing. Suppose let hash function h_2 be defined as $h_2(x)= 1+(x \mod 6)$. Adding 1 to h_2 ensures that the increment is not equal to 0. The values 16, 18, 9 are inserted.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
						16		18	9

- Now insert 36, $h_1(36)=36 \mod 10= 6$, collision occurs because the state of location 6 is OCCUPIED. So find the next free location using double hashing. The next location is determined by $h_2(x)$ function,

i.e. $h_2(36)$ =1+(36 mod 6)=1. Adding this to the current location leads to probe location 7, which is in an EMPTY state, so 36 can be inserted into 7th location.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
						16	36	18	9

- Now insert 26 with 26. With linear probing 26 follows the same probing sequence as 16 and 36, but using double hashing it will not. 26 is initially hashed to location 6 which is OCCUPIED. The next free location is determined by h_2 function. So $h_2(26)$=1+(26 mod 6)=2. Therefore, the next location is at (6+ 2) mod 6, i.e. location 2. The state of location 2 is EMPTY so insertion of 26 at 2nd location can be done.

Observe that each key probes the array locations in a different order. Hence, there is no clustering. If two keys probe the same location the next location they probe is different.
Algorithms 8.11 and 8.12 denote the insertion of a value into a table using double hashing.

Algorithm 8.11: DINSERT (key, r)

```
1. Set i=h( key )
2. Set c=increment(key)
3. last=(i+(m-1)*c) % m
4. Loop while(i!=last&&!empty(r[i])&&!deleted(r[i])&&r[i].k!=key)
5. Set i=(i+c)%m
6. check whether(empty(r[i])||deleted(r[i])) then
   1. Set r[i].k=key
   2. Set n=n+1
7. Else write "table full, or key already in table"
8. End
```

Algorithm 8.12: SEARCH (key, r)

```
1. Set i=h(key)
2. Set c=increment(key)
3. Set last=(i+(n-1)*c)%m
4. Loop while(i!=last&&!empty(r[i])&&r[i].k!=key)
5. Set i=(i+c)%m;
6. check whether r[i].k==key then
     return(i)
7. Else return(-1)
8. End
```

Another example of double hashing: Consider a hash table storing integer keys that handle collision with double hashing where N = 13, $h_1(x)$ = x mod 13 and $h_2(x)$ = 7 − x mod 7. Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

The following table shows the number probes of each key:

x	$h_1(x)$	$h_2(x)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

Then the array after insertions is as shown below:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
31		41			18	32	59	73	22	44		

Extendible hashing: Extendible hashing makes hashing dynamic. With this technique, access to data from the buckets is done indirectly through an index which is adjusted dynamically in order to reflect the changes in a table. The main feature of extendible is to organize the index which is an expandable table.

Whenever a hash function is applied for a key, the value returned by it gives the position in the index—not the table of keys and these values are known as pseudo keys. Using extendible hashing there is no need to reorganize the table when insertion and deletion take place.

A single hash function `h` is used but depending on the size of the index a portion of address of `h(k)` is used. The address of `h(k)` is represented as a string of bits and only the `i` left-most bits are used. Here `i` is called the depth of the directory.

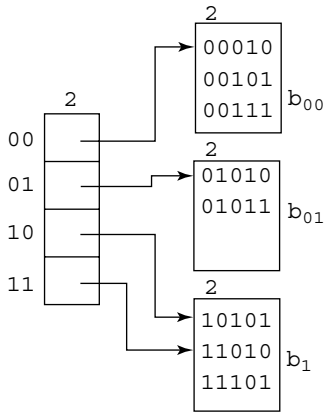
Consider an example where the hash function `h` returns patterns of five bits. Suppose a pattern is a string 01101 and the depth is two, then the left-most bits 01 are used to represent the location in the directory which contains the pointer to a bucket where the required key can be `S` found or the one where the required key can be inserted.

In Figure 8.12 the values of hash functions `h` are shown in buckets and these values represent the keys that are actually stored in the buckets.

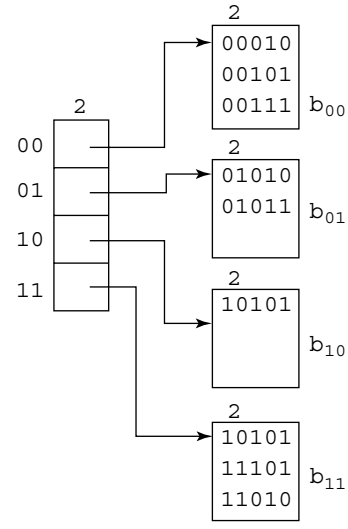
Each bucket will have its own depth called local depth which indicates the number of leftmost bits of `h(k)`. Each bucket holds all the keys whose local depth is same. In Figure 8.12, the local depth of a bucket is shown on top of it. The bucket b_{00} contains the keys for which `h(k)` starts with 00 bits. The use of local depth is to determine whether the bucket can be accessed from only one location or at least two.

When local depth of a bucket is smaller than the depth of directory then overflow occurs. Now split the bucket by changing half of the pointers pointing to the bucket to point to newly created bucket. In Figure 8.12(a) a key with `h` value as 11010 is to be inserted then it is first moved to 4^{th} location of the directory through which is sent to bucket b_1 which holds the keys whose left-most bit is 1. Here overflow occurs because there is no slot in bucket b_1 so split the b_1 into b_{10} (new name for b_1) and b_{11} (newly created one). The keys of bucket b_1 are shared between b_{10} and b_{11} as in Figure 8.12(b).

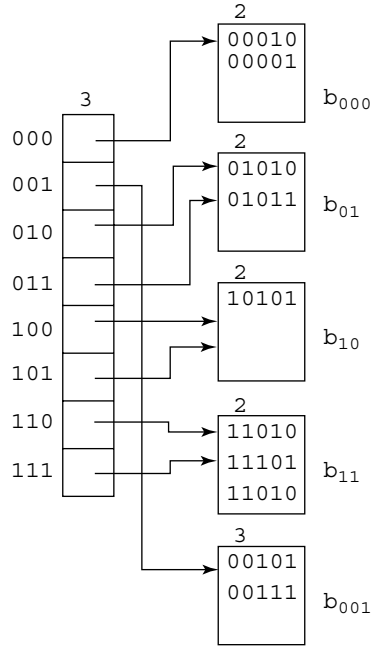
When the local depth is equal to the depth of directory and if overflow occurs then change the size of the directory after splitting the buckets. In Figure 8.12(b) a key with `h` value as 00001 is to be inserted into the directory. Since its left-most bits are 00 it is hashed to the first location of the directory through which it is moved to b_{00}. Here overflow occurs because no slot is empty in b_{00}. The directory size is doubled since the local depth is equal to the depth of the directory. Now the depth directory is changed from 2 to 3, i.e. b_{00} becomes



(a) Insert `h(k)` = 11010



(b) b_1 is split into b_{10} and b_{11}



(c) Insertion of h_k =00010 b_{00} is split into $b_{000}$and b_{001}

Figure 8.12 Insertions using linear probing

b_{000} and the newly created bucket is b_{001}. All the keys of whose left-most bits are 000, becomes elements of b_{000}, and the keys whose h value starts with 001 become elements of b_{001}, as in Figure 8.12(c).
Algorithm 8.13 denotes the insertion of a a value into a table using extendible hashing.

Algorithm 8.13: EXH- INSERT (k)

```
1. Set pattern=h(k)
2. Set p=directory(depth(directory) local depth of  pattern)
3. Check whether slots available in bucket b_d pointed by p then
   insert k into the bucket
4. Else
   1. Split bucket b_d into b_d0 and b_d1
   2. Set local depth[b_d0] and b_d1=depth[b_d]+1
   3. Share keys of b_d between b_d0 and b_d1
   4. check whether local depth[b_d]<depth of directory then
       Update half of the pointer of b_d to point to b_d1
5. Else
6. Double the size of directory and increment its local depth
7. Set proper pointers to directory entries
5. End
```

8.6 COMPARISON OF CHAINING AND OPEN ADDRESSING

- Chained hash tables are simple to implement and require only basic data structures.
- Chained hash tables are insensitive to clustering while writing suitable hash functions.
- Open addressing depends upon better hash functions to avoid clustering.
- Chains grow longer as the table fills, and a chained hash table cannot fill up and does not exhibit the sudden increases in lookup times that occur in a near-full table with open addressing.
- When hash tables are used to store large records, chaining uses less memory than open addressing.
- For small record sizes, open addressing is more space-efficient than chaining since they do not need to store any pointers or allocate any additional space outside the hash table.
- Open addressing avoids the extra indirection required for changing external storage because it uses internal storage. It also has better locality of reference, particularly with linear probing. They can be easier to serialize because they don't use pointers.
- If the open addressing table only stores references to elements (external storage), it uses space comparable to chaining even for large records but loses its speed advantage. On the other hand, normal open addressing is a poor choice for large elements, since these elements fill entire cache lines, and a large amount of space is wasted on large empty table slots.
- Generally open addressing is better used for hash tables with small records that can be stored within the table and fit in a cache line. They are particularly suitable for elements of one word or less. In cases where the tables have high load factors, the records are large, or the data are variable-sized, chained hash tables often perform better as well.

8.7 APPLICATIONS

Hash tables have their applications in various fields. Some of them are listed below:

- Finding duplicate records

- Geometric hashing
- Finding similar substrings
- Finding similar records
- Evaluation of join operation on relational databases
- Direct file organization
- Implementing associative arrays and dynamic sets

8.8 DICTIONARY ADT

A dictionary `D` supports the following operations:

`size()`: It gives the number of elements in `D`
`isEmpty()`: Returns true if `D` is empty
`elements()`: Returns the elements of `D`
`keys()`: Returns the keys of `D`
`search(k)`: Returns the position of the item to be searched if found. If not return null position
`searchAll(k)`: Returns the position of all items whose key matches with `k`
`insert(k,v)`: Inserts an item v with the key `k` into `D`
`delete(k)`: Deletes an item whose key matches with `k` from `D`
`deleteAll(k)`: Deletes all items whose key matches with `k` from `D`

SUMMARY

- Dictionary contains data elements as pairs of the form (k, v), where `k` is the key and `v` is the value.
- A dictionary with duplicates is a dictionary which allows two or more (key, value) pairs with the same key.
- An ordered dictionary can be represented as an ordered linear list.
- Skip lists are sorted linked lists. In a skip list, nodes have many 'next' references known as forward references, whereas in an ordinary list nodes have only one 'next' reference.
- Linear probing is a technique for resolving hash collisions of values of hash functions by searching the hash table sequentially for a free location. The disadvantage of linear probing is primary clustering.
- Quadratic probing is another method for resolving collisions in hash tables. It operates by taking the original hash value and adding successive values of a quadratic polynomial to the starting value. Quadratic probing avoids the clustering problem that occurs with linear probing.
- Double hashing is a technique used in hash table to resolve hash collisions. It uses one hash value as a starting value and continual evolution of an interval until the desired value is found, an empty location is found or the entire table is searched.

EXERCISES

Fill in the Blanks

1. Sequential access takes _____ time to access every (key, value) pair.
2. _____ are the collision resolution techniques.
3. The process of mapping the keys to their respective positions in the hash table is called _____ .
4. In _____ collision technique the interval is decided using another independent hash function.
5. _____ hashing performs hashing dynamic.

Multiple-choice Questions

1. The locations in the hash table are called as _____ .
   a. Slots b. Sets
   c. Buckets d. Entries
2. Hash tables are ideal data structures for _____ .
   a. Dictionaries b. Graphs
   c. Trees d. None
3. Keeping subsequent items within the table and computing possible locations is known as _____ .
   a. Separate chaining b. Open addressing
   c. Direct chaining d. None
4. For a given ordinary hash function `h(x)`, the linear probing function will be
   a. `h(x)=h(x)+i) % mod n` b. `h(x)=(h(x)+i) % mod n`
   c. `h(x)=h(x)% mod n` d. None
5. Local depth of a bucket gives _____ .
   a. Number of left-most bits b. Number of most significant bits
   c. Number of bits d. None
6. The double hashing function is defined as _____ .
   a. $h(x,i)=h_1(x)+h_2(x) \bmod n$ b. $h(x,i)=h_1(x)+i*h_2(x) \bmod n$
   c. $h(x,i)= i*h_1(x)+ h_2(x) \bmod n$ d. $h(x,i)=h_1(x)-i*h_2(x) \bmod n$

Short-answer Questions

1. Explain dictionary as an ADT.
2. What are skip lists? Explain with an example.
3. Define hash table.
4. Explain methods of defining hashing function.
5. What are collisions?
6. Give the advantages and disadvantages of separate chaining.

Essay Questions

1. Give the applications of dictionary or dictionary with duplicates in which sequential access is desired.
2. Explain the operations that can be performed on linear list and skip lists.
3. How are insertions and deletions handled in a chained hash table? Explain.
4. Illustrate the linear probing method in hashing. Explain its performance analysis.
5. What is rehashing? How does it overcome the drawbacks of linear probing?
6. How are collisions handled in linear probing?
7. Differentiate between separate chaining and open addressing.

Chapter 9

Trees and Binary Trees

In the previous chapters, linear data structures such as arrays, stacks and queues have been studied. Trees and graphs are two dimensional in structure and are referred to as non-linear data structures. The study of non-linear data structures is considered in this and forth-coming chapters. This chapter particularly deals with defining the tree data structure, the terminology related to them and their representation in different ways. A basic type of tree, i.e. binary tree is discussed in detail with its representation, operations and traversals. This chapter also includes threaded binary trees, ADT of binary trees, changing a general tree into a binary tree along with a detailed explanation about expression trees as an application.

9.1 INTRODUCTION

A tree is a finite set of one or more nodes. In this set of nodes a node is referred to as a root. The other nodes could be partitioned into $T_1, T_2, T_3, \ldots, T_t$ disjoint sets of trees where $t \geq 0$, and are referred to as subtrees of the tree.

Every node in a tree represents a unit of information. The links/arcs between the nodes are known as branches and specify the association between the units of information. A tree is depicted in Figure 9.1.

All the nodes are reachable from the root through a unique sequence of branches called a path, and no node is left isolated in the tree. The number of arcs in a path is called the length of the path.

By definition (1) connected less and (2) acyclicness or no closed loops are ensured in a tree. The tree structure does not allow any set of nodes to link together to from a closed loop or to be acyclic.

9.2 TERMINOLOGIES

The number of branches associated with a node is called the *degree* of a node. A branch that comes to the node is the *indegree* branch and the branch that emerges from the node is the *outdegree* branch. *Degree* of a node is the sum of indegree and outdegree branches. A node whose indegree is zero is called the *root*. Nodes whose outdegree is zero is the *leaf nodes* or *terminal nodes*. The *degree of a tree* is the maximum degree of the node in the tree. All the nodes other than root and leaf nodes are the *internal nodes* or *non-terminal nodes*. All the

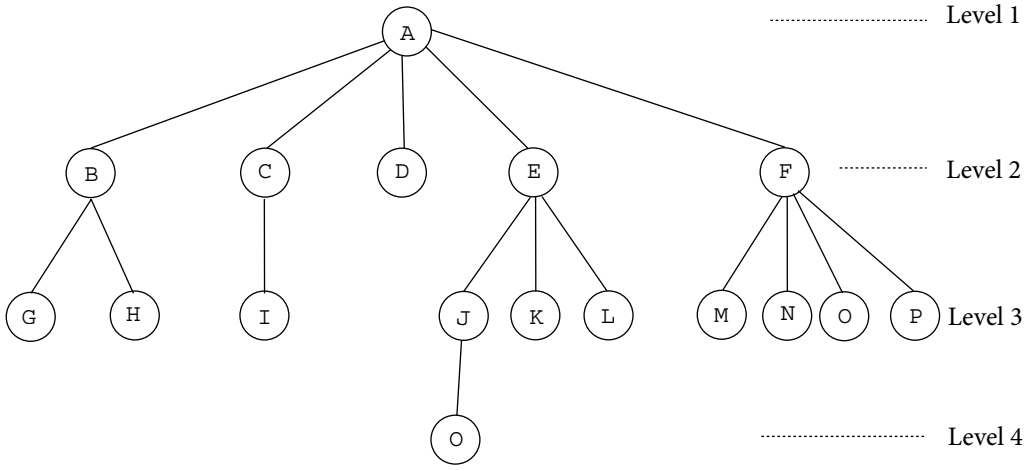


Figure 9.1 A tree

nodes except the root must have a single indegree. The nodes that emerge from the branches of a node are the *children,* and the node from which the branches emerge is the *parent* node. Two or more nodes of the same parent are referred to as *siblings*. The nodes that appear in the path from the root to a given node are the *ancestors* of the given node, and the nodes that appear in all the paths from a given node to a leaf are the *descendents* of the node. *Level* of a node is defined starting with the root given as Level 1 and the other nodes are given with higher level numbers based on their association. If a parent is at Level `i` then its children must be at Level `i+1`. The *height* of a tree is the maximum level of a node in the tree. A set of zero or more disjoint trees is a *forest*. If a root node is deleted from a tree then the set of all disjoint trees i.e. subtrees leads to a forest. Hierarchical structures such as family, university, etc. can be very effectively depicted using trees. In a tree a node can have any number of children and every child node is again considered a tree, which may have either zero or more children.

In Figure 9.1, `A` is the root and `D, G, H, I, K, L, M, N, O, P, Q` are the leaf nodes and the remaining are the internal nodes or non-terminal or non-leaf nodes. Nodes `G, H` are the children of `B` and `B` is a parent node. Nodes `B, C, D, E, F` are siblings. Nodes `J, K, L` are also siblings. Nodes `A, E, J` are ancestors to `Q`. Nodes `J, K, L, Q` are descendents of `E`. The root `A` is at level 1, its children `B, C, D, E, F` are at level 2 and so on. The height of a tree is the maximum level, i.e. 4 . When `A` is removed from the tree, it results in a forest with five disjoint trees such as `{B,G,H}, {C,I}, {D}, {E,J,K,L,Q}, {F,M,N,O,P}`.

A tree does not limit the number of children for a given node. A University with two constituent colleges and each having many departments and each department having professors, undergraduate (UG) students and post-graduate (PG) students can be clearly represented using a tree (Figure 9.2).

9.3 REPRESENTATION OF A TREE

A tree can be very effectively represented using linked lists. In this the root node appears first and it is followed by the list of subtrees of the node. The same is repeated for every subtree in the tree. The general node structure of a linked list is shown in Figure 9.3(a). The list form of the tree depicted in Figure 9.1 is as `(A,(B,(G,H),C (I),D,E(J,(Q),K,L),F(M,N,O,P)))`. Figure 9.3(b) shows the linked representation of the tree depicted in Figure 9.1.

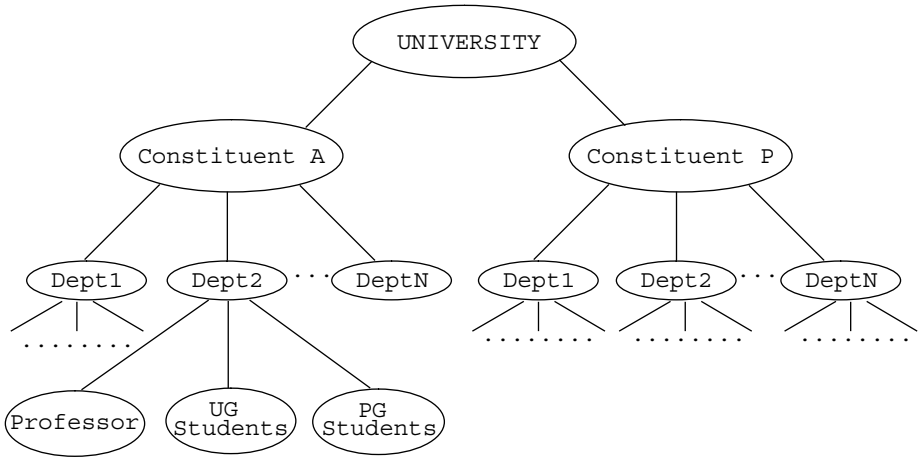
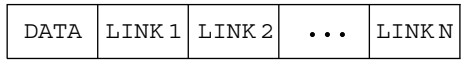
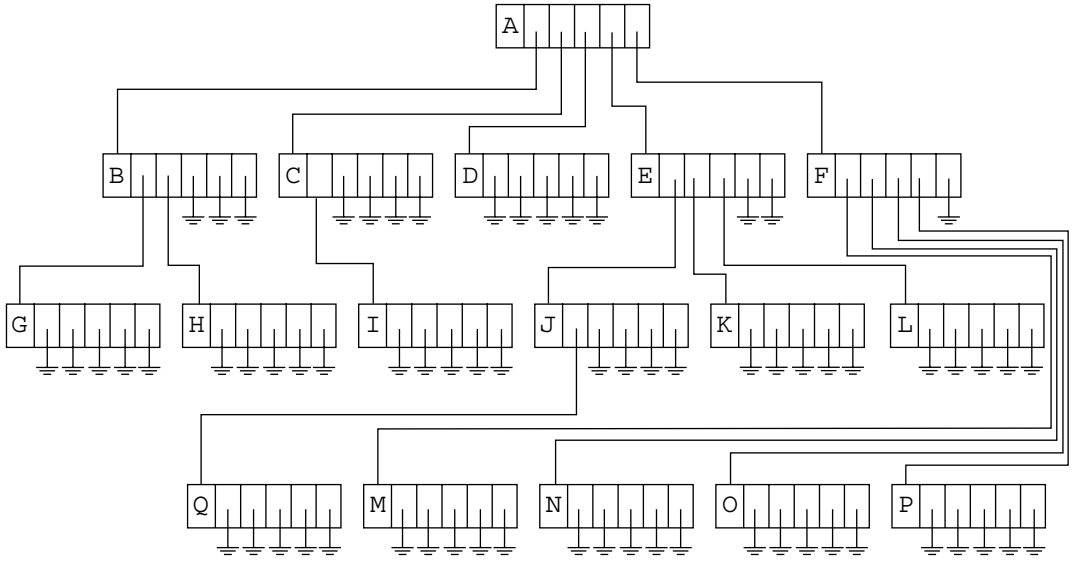


Figure 9.2 A tree representing a hierarchical structure of a university



(a) General node structure



(b) Linked list representation of trees shown in Figure 9.1

Figure 9.3 Representation of a tree

Information part of the tree node is stored in the `DATA` field of the node. Pointers to the child nodes are stored in the `LINK` field of the node.

9.4 BINARY TREES

A binary tree T is a finite set of elements. An empty tree is a binary tree. If non-empty, a tree in which all nodes can have at most two children and are treated as two binary trees, are referred to as left and right subtrees of T (Figure 9.4).

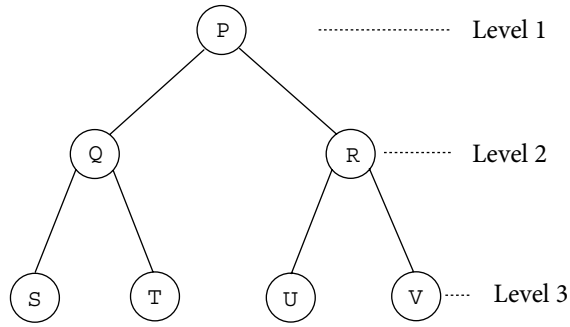


Figure 9.4 A binary tree

The major differences between a tree and a binary tree are:

- A tree can never be empty whereas a binary tree can be empty with no nodes.
- There may be any number of subtrees for a node in a tree. There must be either zero or one or two subtrees for a node in a binary tree.
- There is no ordering of subtrees in a tree, but the ordering of left and right subtrees of all nodes is clearly maintained in a binary tree.

A binary tree is very similar to a tree in its structure and terminology. The additional features with respect to binary tree are:

(1) A binary tree with n elements will have $n-1$ number of edges, where $n>0$.
(2) At level i a binary tree can have at most 2^{i-1} number of nodes, where $i \geq 1$.
(3) A binary tree of height h can have minimum h and maximum 2^h-1 number of nodes, here $h \geq 0$.
(4) For a binary tree with n elements, height is minimum $\lceil \log_2(n+1) \rceil$ and maximum n.
(5) In a binary tree the number of non-terminal nodes will be one less than that of the number of terminal nodes.

A binary tree of height h that consists of maximum permissible number of children to all nodes, i.e. 2^h-1 nodes is known as a **full binary tree**. Figure 9.5 illustrates a full binary tree of height 4.The nodes of a full binary tree are numbered 1 through 2^h-1 starting from root at level 1 to nodes at level h and from left to right.

A binary tree that has maximum permissible number of nodes at all the levels except at the last level and if all the nodes at the last level are as far left as possible is known as a **complete binary tree**.

The nodes of a complete binary tree are also numbered in the same way as a full binary tree. Using this numbering, it is easy to find the parent, the left and the right children of a node.

A complete binary tree with n nodes is considered and if it is numbered such that $1 \leq i \leq n$, where i is the number of the nodes then it satisfies the following properties:

1. If $i=1$ then it is the root of the binary tree.
2. If $i>1$ then,

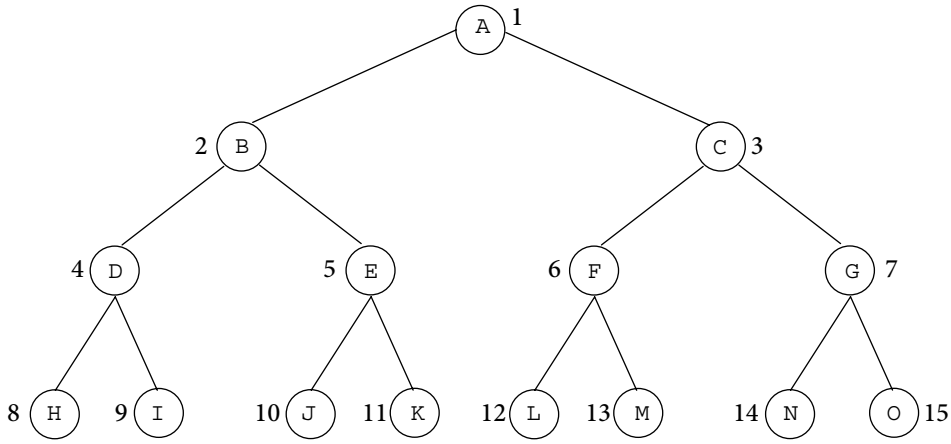


Figure 9.5 Full binary tree of height 4

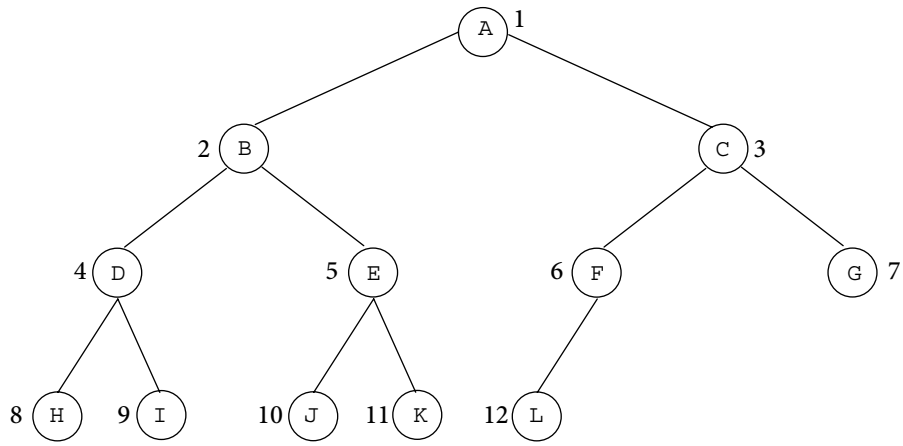
(i) the parent of the node is a node with the number $\lfloor i/2 \rfloor$.

(ii) the left child of the node is a node with the number $2i$ where $2i \leq n$. If $2i > n$ then i has no left child.

(iii) the right child of the node is a node with the number $2i+1$ where $2i+1 \leq n$. If $2i+1 > n$ then i has no right child.

The height of a complete binary tree with n nodes is $h = \lceil \log_2(n+1) \rceil$. A full binary tree is a special case of a complete binary tree. Figure 9.6. illustrates a complete binary tree.

The symbol '$\lfloor\ \rfloor$' refers to floor function. The floor function takes a floating point number and returns the first integer, which is less than or equal to that number. The symbol '$\lceil\ \rceil$' refers to the ceiling function. The ceiling function takes a floating point number and returns the first integer, which is greater than and equal to that number.



(a)

Figure 9.6 Complete binary trees

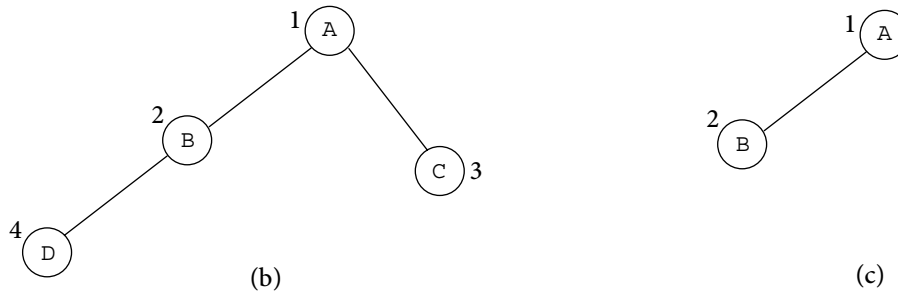


Figure 9.6 *Continued*

A binary tree that has only left or right children is known as a skewed binary tree in general and specifically left-skewed binary tree or right-skewed binary tree, respectively. Figure 9.7 illustrates the skewed binary trees.

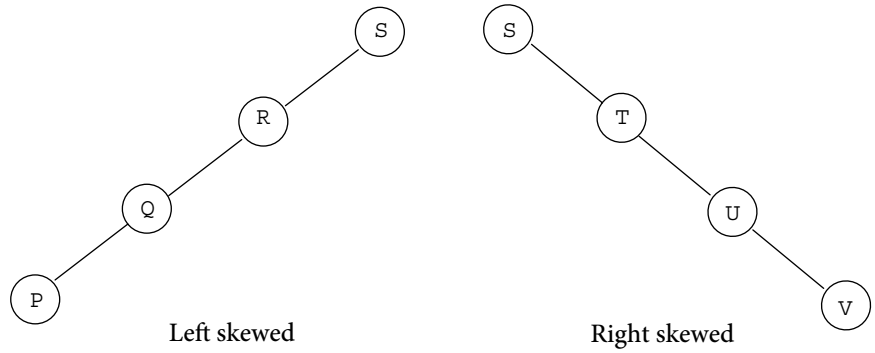


Figure 9.7 Skewed binary tree

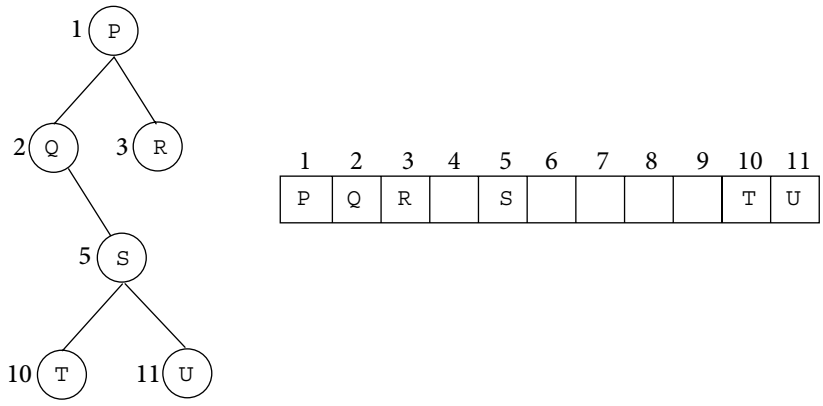
9.5 REPRESENTATION OF BINARY TREES

Binary trees can be represented using both sequential (arrays) and linked data structure.

9.5.1 Array Representation of a Binary Tree

Numbering of nodes and the properties that are discussed in the earlier section are used in representing the binary trees as arrays. The elements in the nodes that are numbered are placed into the corresponding positions of the arrays. The array positions of missing nodes in the binary tree are left empty in the array. Hence, it leads to wastage of a space. Figure 9.8 illustrates the array representation of a binary tree.

For any element in array position `i`, the left child will be in position `2i`, the right child will be in position `(2i+1)`, and the parent is in position $\lfloor i/2 \rfloor$. The array representation of a binary tree depicted in Figure 9.8(a) is shown in Figure 9.8(b). Here, the root element `P` is placed in the first position of the array, i.e. `i=1` position. Its left child `Q` is placed in `2i=2*1=2`nd position of the array. Its right child `R` is placed in `2i+1=2*1+1=2+1=3`, 3rd position of the array. Now consider the element `Q`, as it is in the 2nd position of array, `i=2`, since it does not have a left child, the 4th position of the array is left empty. Its right child `S` is placed in `2i+1=2*2+1=5`th position



(a) Binary tree (b) Array representation of binary tree

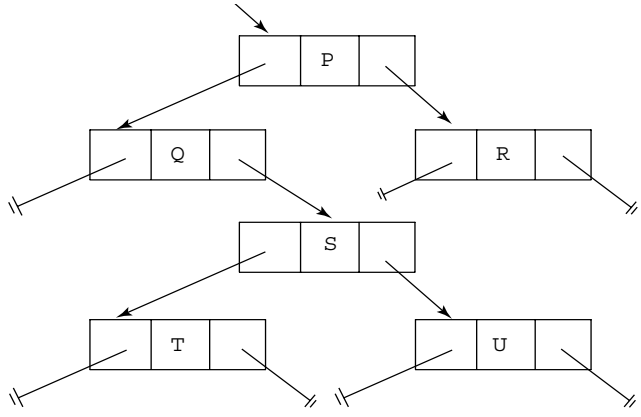
Figure 9.8 Array representation of a binary tree

of the array. Similarly, when `S` is considered, `i=5`, its left child is placed in 10^{th} position and its right child is placed in 11^{th} position. For the elements that are missing in the binary tree, their corresponding positions in the array are left empty.

9.5.2 Linked Representation of Binary Trees

The linked representation is an efficient and advantageous method to represent a binary tree over arrays. The node structure contains three fields, a `DATA` field and two pointers, `LCHILD` and `RCHILD` to point to the left and right children of nodes, respectively. By using a pointer to the root node the tree can be accessed. Figure 9.9 illustrates the linked representation of a binary tree.

(a) Structure of a node



(b) Linked representation of a binary tree of Figure 9.8(a)

Figure 9.9 Linked representation of a binary tree

For a given node in a binary tree the left and the right children can be determined using `LCHILD` and `RCHILD` pointers. But its parent node cannot be determined. In any application, if the parent node reference is needed then one more pointer field may be included in all the nodes.

The linked representation of a binary tree with `n` nodes uses `2*n` number of pointers out of which `n+1` will be the null pointers.

9.6 BINARY TREE OPERATIONS

Various operations can be performed on binary trees. A few commonly performed operations are listed below.

- Search: Search is an operation that is performed on a binary tree to find a given key in the tree. If it is found then the search is successful and if it is not found it is an unsuccessful search.
- Insert: Insertion is an operation that is performed on a binary tree to include an element in an existing binary tree. A new element can be inserted at any position in a binary tree. To insert an element along with the element the node or element to which the new element is inserted a child must be specified. The new element either to be inserted as a left child or as a right child can also be mentioned.
- Delete: Deletion is an operation that is performed on a binary tree to remove an element from a non-empty binary tree.
- Merge: Merge is an operation that is performed on a binary tree to merge or combine two binary trees into one.
- Display: Display is an operation that is performed on a binary tree to output or show the elements of a binary tree.

The implementation of most frequently used operations search, insert, delete and display is given in Program 9.1.

Program 9.1 Binary tree operations

```
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
int item,k;
struct BTREE
{
   int data;
   struct BTREE*lchild,*rchild;
   }*root=NULL,*n,*ptr,*ptr1,*ptr2,*p;
   struct BTREE*search(struct BTREE*ptr,int key)
   {
      if(ptr!=NULL){
      if((ptr->lchild->data==key)||(ptr->rchild->data==key))
      p=ptr;
      if(ptr->data==key)
         ptr1=ptr;
      else {
      search(ptr->lchild,key);
      search(ptr->rchild,key);
```

```
   }  }
 return ptr1;
}
void create()
{
 if(root==NULL)
 {
  cout<<"\nEnter Element To Be Insert";
  cin>>item;
  n=(struct BTREE*)malloc(sizeof(struct BTREE));
  n->data=item;n->lchild=NULL;n->rchild=NULL;
  cout<<"\nItem"<<item<<"Is Successfully Inserted\n";
  root=n;
 }
 else
  cout<<"\nBinary Tree Is Already Created\n";}
void insert()
{
 int key;
 char c;
 cout<<"\nEnter Element To Be Insert";
 cin>>item;
 cout<<"\nEnter Key Element";
 cin>>key;
 ptr=search(root,key);
 if(ptr==NULL)
 {
  cout<<"\nKey element Is Not Found..Insertion Is Not Possible";
  return;
 }
 n=(struct BTREE*)malloc(sizeof(struct BTREE));
 if(ptr->lchild==NULL||ptr->rchild==NULL)
 {
  cout<<"\nWhere Do You Want To Insert Press Left(L) Right(R)";
  c=getch();
  if(c=='l'||c=='L')
  {
   if(ptr->lchild==NULL){
    n->data=item;n->lchild=n->rchild=NULL;
    ptr->lchild=n;}
   else {
    cout<<"\nInsertion Is Not Posible As A Left Child";
    return;
   }
  }
  else {if(ptr->rchild==NULL){
    n->data=item;n->lchild=n->rchild=NULL;
    ptr->rchild=n;
```

```
                }
    else{
     cout<<"\nInsertion Is Not Possible As A Right Child";
     return;
     }
    }
  }
  else
  {
   cout<<"\nThe Key Node Already Has Child";
   return;
  }
  cout<<"\nItem"<<item<<"Is Successfully Inserted";
}
void delet()
{
 struct BTREE*ptr3;
 if(root==NULL)
 {
  cout<<"\nTree Is Empty";
  return;
 }
 cout<<"\nWhich Element You Want To Delete";
 cin>>item;
 ptr=search(root,item);
 if(ptr==NULL)
 {
  cout<<"\nKey Element Is Not Found";
  return;
 }
 if(ptr->lchild!=NULL&&ptr->rchild!=NULL)
 {
  cout<<"\nKey Element Node Has Two Childs...,Deletion Is Not Possible";
  return;
 }
 if(ptr==root)
 {
  if(root->lchild!=NULL)
   root=ptr->lchild;
  else
   root=ptr->rchild;
 }
 else
 {
  if(p->lchild==ptr)
  {
   if(ptr->lchild==NULL&&ptr->rchild==NULL)
    {p->lchild=NULL;ptr=NULL;}
   else if(ptr->lchild!=NULL)
```

```
    {p->lchild=ptr->lchild;ptr=NULL;}
   else{p->lchild=ptr->rchild;ptr=NULL;}
  }
  else
  {
   if(ptr->lchild==NULL&&ptr->rchild==NULL)
    {p->rchild=NULL;ptr=NULL;}
   else if(ptr->lchild!=NULL)
    {p->rchild=ptr->lchild;ptr=NULL;}
   else{p->rchild=ptr->rchild;ptr=NULL;}
  }
 }
 cout<<"\nItem"<<item<<"Is Successfully Deleted";
}
void traverse(struct BTREE *ptr)
{
 if(ptr!=NULL)
 {
  cout<<ptr->data<<" ";
  traverse(ptr->lchild);
  traverse(ptr->rchild);
 }
}
int searchele(struct BTREE*p,int key)
{
 if(p!=NULL)
  {
  if(p->data==key)
   k=1;
  else
  {
   searchele(p->lchild,key);
   searchele(p->rchild,key);
  }
 }
 return k;
}
void main()
{
 int ch,s;
 clrscr();
 while(1)
 {
  cout<<"\nBINARY TREE OPERATIONS\n1.CREATION\n2.INSERTION\n3.DELETION\n";
  cout<<"4.TRAVERSE\n5.SEARCHING\n6.EXIT";
  cout<<"\nEnter Your Choice";
  cin>>ch;
  switch(ch)
```

```
  {
   case 1:create();break;
   case 2:insert();break;
   case 3:delet();break;
   case 4:if(root==NULL){cout<<"Treee Is Empty";break;}
         cout<<"\nTree Elements Are";traverse(root);break;
   case 5:if(root==NULL){cout<<"Treee Is Empty";break;}
     s=k=0;cout<<"Enter Search Element";
     cin>>item;
     s=searchele(root,item);
     if(s==1)
      cout<<"Item"<<item<<"Is Found";
     else
      cout<<"Item"<<item<<"Is Not Found";
     break;
   case 6:exit(0);
   default:cout<<"\nInvalid Choice";
  }
 }
}
```

Output

```
BINARY TREE OPERATIONS
1.CREATION
2.INSERTION
3.DELETION
4.TRAVERSE
5.SEARCHING
6.EXIT
Enter Your Choice1

Enter Element To Be Insert10

Item 10 Is Successfully Inserted

BINARY TREE OPERATIONS
1.CREATION
2.INSERTION
3.DELETION
4.TRAVERSE
5.SEARCHING
6.EXIT
Enter Your Choice2

Enter Element To Be Insert20
```

```
Enter Key Element10

Where Do You Want To Insert Press Left(L) Right(R)
Item 20 Is Successfully Inserted
BINARY TREE OPERATIONS
1.CREATION
2.INSERTION
3.DELETION
4.TRAVERSE
5.SEARCHING
6.EXIT
Enter Your Choice2

Enter Element To Be Insert30

Enter Key Element20

Where Do You Want To Insert Press Left(L) Right(R)
Item 30 Is Successfully Inserted

BINARY TREE OPERATIONS
1.CREATION
2.INSERTION
3.DELETION
4.TRAVERSE
5.SEARCHING
6.EXIT
Enter Your Choice4

Tree Elements Are10 20 30
BINARY TREE OPERATIONS
1.CREATION
2.INSERTION
3.DELETION
4.TRAVERSE
5.SEARCHING
6.EXIT
Enter Your Choice5
Enter Search Element20
Item 20 Is Found
BINARY TREE OPERATIONS
1.CREATION
2.INSERTION
3.DELETION
4.TRAVERSE
5.SEARCHING
6.EXIT
Enter Your Choice3
```

```
Which Element You Want To Delete10
Item 10 Is Successfully Deleted
BINARY TREE OPERATIONS
1.CREATION
2.INSERTION
3.DELETION
4.TRAVERSE
5.SEARCHING
6.EXIT
Enter Your Choice 6
```

9.7 BINARY TREE TRAVERSALS

To perform an operation on the element of a node it needs to be visited only once, and visiting a node of a binary tree is known as *traversal*.

During traversal the three tasks that are performed are visiting a node (V), traversing the left subtree (L) and traversing the right subtree (R). Various combinations of these three tasks lead to six different traversals. They are VLR, VRL, LVR, RVL, LRV and RLV. Among these the traversals that move from left to right are very significant. The following are familiar ways of traversals:

1. Inorder traversal
2. Preorder traversal
3. Postorder traversal
4. Level-order traversal

9.7.1 INORDER TRAVERSAL (LVR)

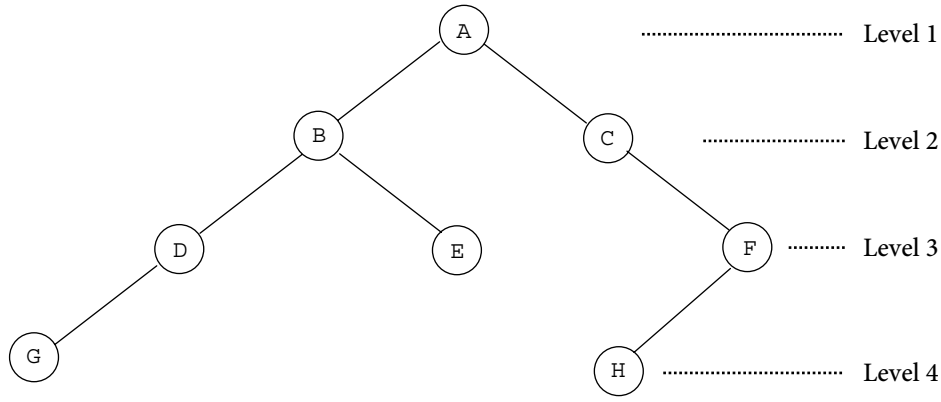
The inorder traversal starts with the root node and traverses the left subtree. In the left subtree if a node has a left child then it is traversed if it does not have any left child then it is visited and traverses its right child in the same way. If it does not have any right child then it traces back by one node and proceeds the traversal in the same manner till all the nodes of the binary tree are visited.

Algorithm 9.1. explains the recursive procedure to perform the inorder traversal of a binary tree.

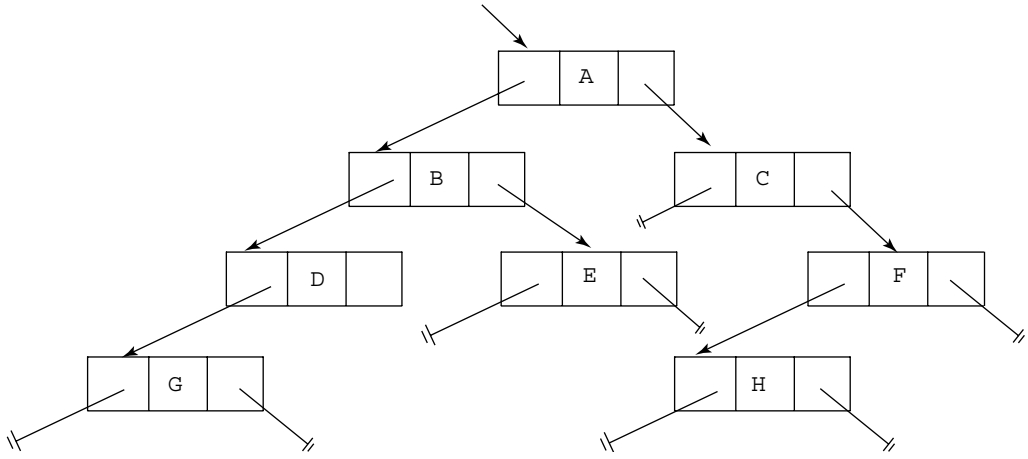
Algorithm 9.1

```
inorder_traversal (NODE)
 if NODE!=NILL then
{
   inorder_traversal(LCHILD(NODE));   //traverse the left subtree(L)
   visit(DATA (NODE));                //visit the node
  inorder_traversal(RCHILD(NODE)); //traversal the (Right subtree(R))
}
end if
End inorder_traversal.
```

The inorder traversal of the binary tree starts with the root node traversing the left subtree, visiting the node and traversing the right subtree.



(a) Structure of a node



(b) Linked representation of a binary tree of Figure 9.10(a)

Figure 9.10 Binary tree to demonstrate traversal

The inorder traversal of a binary tree in Figure 9.10 starts with the root node A and traverses the left subtree, i.e. B, as node B has the left child move to D and D has its left subtree so move to G. Now node G does not have any left subtree, i.e. LCHILD(G)=NIL so visit G. Now as node G also does not have any right subtree, trace back by one node, i.e. to D, visit D. Node D also does not have any right subtree to traverse so again trace back by one node to B, visit B. Node B has the right subtree so traverse right subtree E. As it does not have any left child visit E and it also does not have any right child so trace back to B, and B is already visited again, so trace back to A. All the nodes in the left subtree of A are visited, i.e. left subtree is traversed. So visit A. Node A also has the right subtree. So traverse right subtree. Node C, the right child of A, does not have any left child so visit C and move to its right child F as it has the left child move to H and H does not have any left child so visit H. Node H also does not have any right child so trace back by one node, i.e. to F, visit F. Node F also does not have any right child so trace back by one node to C. Now the traversal of the right subtree is also completed. Finally, the output of inorder traversal is GDBEACHF.

9.7.2 Preorder Traversal (VLR)

The preorder traversal visits a node first after which it traverses its left subtree and then traverses its right subtree.

The preorder traversal starts with the root node visiting it. If the root node has a left subtree then it is preorderly traversed and then proceeds to traverse the right subtree in the same manner.

The recursive procedure to perform the preorder traversal is explained by Algorithm 9.2.

Algorithm 9.2

```
preorder_traversal(NODE)
If NODE!=NIL then
{
   Visit(DATA (NODE));                    //visit the node
   preorder_traversal(LCHILD(NODE));      //preorder traverse of the left
                                             subtree(L)
   preorder_traversal(RCHILD(NODE));      //preorder traverse of the right
                                             sub tree(R)
}
End preorder_traversal
```

The preorder traversal of the binary tree depicted in Figure 9.10 starts with the root node, A is visited. It has both the left and right subtrees. According to the preorder traversal, the left subtree should be traversed, so visit the left child of A, i.e. B. Now visit D, the left subtree of B and visit G, the left subtree of D. Node G does not have a left subtree, i.e. LCHILD(G)=NIL and G also does not have a right subtree, i.e. RCHILD(G)=NIL. So trace back by one node, i.e. to D. Node D is already visited, its left child is also visited now it is time to traverse its right child but it does not have right child. So trace back by one node, i.e. to B. Here B is visited, its left child is already visited; now traverse E, the right child of B. Node E does not have both the children, so trace back by one node, i.e. B and as its traversal is completed, trace back to A and proceed to the right subtree of A. Visit C the right child of A. It do not have any left child so traverse its right subtree F. Visit F it has a left child so move to H and visit H. All the nodes are visited; hence, the traversal is completed. The final output of the preorder traversal of the binary tree is ABDGECFH.

9.7.3 Postorder Traversal (LRV)

The postorder traversal of a binary tree traverses the left subtree of a node first, and traverses its right subtree and finally visits the node.

The postorder traversal starts with the root node and checks whether it has a left subtree or not. If it has the left subtree then the postorder traversal of the left subtree takes place. Again it checks for the right subtree of the root node. If it has the right subtree, then the postorder traversal of the right subtree takes place. After completing the traversal of the left and right subtrees the root node is visited.

The recursive procedure to perform the postorder traversal is explained by Algorithm 9.3.

Algorithm 9.3

```
postorder_traversal(NODE)
if NODE!=NIL then
{
   postorder_traversal(LCHILD(NODE)); // postorder traverse of the left
```

```
                                        subtree(L)
   postorder_traversal(RCHILD(NODE));    //postorder traverse of the right
                                           sub tree(R)
   Visit(DATA(NODE));                    //visit the node
  }
End of postorder_traversal.
```

The postorder traversal of the binary tree depicted in Figure 9.10 starts with the root A, A has the left subtree, LCHILD(A)=B, move to B, B also has a left subtree D, move to D and D also has the left subtree G, move to G. G does not have the left child, LCHILD(G)=NIL and it also does not have the right child, RCHILD(G)=NIL so visit G. Trace back by one node to D; the left subtree of D is visited and it does not have any right child, i.e. RCHILD(D)=NIL, so visit D, trace back by one node to B. The nodes in the left subtree of B are visited, now move to its right subtree E. E does not have both the subtrees so visit E, trace back by one node to B. Now, the left and right subtrees are traversed, so visit B; trace back by one node to A. Traversal of the left subtree of A is completed; as A has the right subtree it must be traversed. Move to C, the right subtree of A. Node C does not have any left subtree but it has the right subtree so move to F. Node F has the left subtree so move to H. Node H does not have both left and right subtrees so visit H. Trace back by one node to F. Node F does not have any right subtree, so visit F, trace back by one node to C. As its right subtree is traversed, now visit C and trace back by one node to A, the root node. Now, traversal of left and right subtrees is completed so visit A. Finally, the output of the postorder traversal of the binary tree is GDEBHFCA.

9.7.4 Level-order Traversal

The level-order traversal of a binary tree traverses the nodes in a level-by-level manner from top to bottom, and among the nodes of the same level they are traversed from left to right. A data structure called queue is used to keep track of the elements yet to be traversed.

The recursive procedure to perform the level-order traversal is explained by Algorithm 9.4.

Algorithm 9.4

```
levelorder-traversal(NODE)
While(NODE!=NULL)
{
   VISIT(DATA(NODE))           //visit the node
   If(LCHILD(NODE)!=NULL)
      Insert(LCHILD(NODE))     //insert the left child of NODE into queue
   If(RCHILD(NODE)!=NULL)
      Insert(RCHILD(NODE))     //insert the right child of NODE into queue
      levelorder_traversal(front element of queue)
      Delete( ); //delete the front element of queue
}
End levelorder-traversal.
```

The level-order traversal of the binary tree depicted in Figure 9.10 begins with the root node A which is at the first level, i.e. level 1. In a binary tree only one element will be there at the first level, so the elements at level 2 must be traversed now. If more than one element is found in the same level, they should be traversed from left to right. Now, in level 2 traverse the leftmost child B and move towards the right to traverse C. Similarly, in level 3 traverse the leftmost child D and move towards the right to traverse E and F. In the same way, traverse

the nodes G followed by H that are in level 4. All the nodes in the binary tree are traversed. Hence, the output of the level-order traversal is ABCDEFGH.

The space and time complexity of all the four traversals is O(n). Program 9.2 shows the implementation of preorder, inorder and postorder traversals of a binary tree.

Program 9.2

```
// 1. preorder 2.inorder 3.post order traversals
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int item,k;
struct TRAVERSAL
{
 int data;
  struct TRAVERSAL*lchild,*rchild;
}*root=NULL,*n,*ptr,*ptr1;
struct TRAVERSAL*search(struct TRAVERSAL*ptr,int key)
{
 if(ptr!=NULL){
  if(ptr->data==key)
   ptr1=ptr;
  else{
   search(ptr->lchild,key);
   search(ptr->rchild,key);
       }      }
 return ptr1;
}
void insert()
{
 int key;
 char c;
 printf("\nEnter Element To Be Insert");
 scanf("%d",&item);
 if(root==NULL)
 {
  n=(struct TRAVERSAL*)malloc(sizeof(struct TRAVERSAL));
  n->data=item;n->lchild=NULL;n->rchild=NULL;
  printf("\nItem %d Is Successfully Inserted",item);
  root=n;return;
 }
 printf("\nEnter Key Element");
 scanf("%d",&key);
 ptr=search(root,key);
 if(ptr==NULL)
 {
  printf("\nKey element Is Not Found..Insertion Is Not Possible");
  return;
```

```
 }
 n=(struct TRAVERSAL*)malloc(sizeof(struct TRAVERSAL));
 if(ptr->lchild==NULL||ptr->rchild==NULL)
 {
  printf("\nWhere Do You Want To Insert Press Left(L) Right(R)");
  c=getch();
  if(c=='l'||c=='L')
  {
   if(ptr->lchild==NULL){
    n->data=item;n->lchild=n->rchild=NULL;
    ptr->lchild=n;}
   else{
    printf("\nInsertion Is Not Posible As A Left Child");
    return;
   }
  }
  else {if(ptr->rchild==NULL){
    n->data=item;n->lchild=n->rchild=NULL;
    ptr->rchild=n;
              }
   else{
    printf("\nInsertion Is Not Possible As A Right Child");
    return;
    }
   }
 }
 else
 {
  printf("\nThe Key Node Already Has Child");
  return;
 }
 printf("\nItem %d Is Successfully Inserted",item);
}
void preorder(struct TRAVERSAL*ptr)
{
 if(ptr!=NULL)
 {
  printf("%3d",ptr->data);
  preorder(ptr->lchild);
  preorder(ptr->rchild);
 }
}
void inorder(struct TRAVERSAL*ptr)
{
 if(ptr!=NULL)
 {
```

```
   inorder(ptr->lchild);
   printf("%3d",ptr->data);
   inorder(ptr->rchild);
  } }
void postorder(struct TRAVERSAL*ptr)
{
 if(ptr!=NULL)
 {
  postorder(ptr->lchild);
  postorder(ptr->rchild);
  printf("%3d",ptr->data);
 }
}
main()
{
 int ch,s;
 clrscr();
 while(1)
 {
  printf("\nTREE TRAVERSALS\n1.INSERTION\n2.INORDER\n");
  printf("3.PREORDER\n4.POSTORDER\n5.EXIT");
  printf("\nEnter Your Choice");
  scanf("%d",&ch);
  switch(ch)
  {
   case 1:insert();break;
   case 2:if(root==NULL){printf("Treee Is Empty");break;}
     printf("\nTree Elements Are");inorder(root);break;
   case 3:if(root==NULL){printf("Treee Is Empty");break;}
     printf("\nTree Elements Are");preorder(root);break;
   case 4:if(root==NULL){printf("Treee Is Empty");break;}
     printf("\nTree Elements Are");postorder(root);break;
   case 5:exit(0);
   default:printf("\nInvalid Choice");
  }
 }
}
```

Output

```
TREE TRAVERSALS
1.INSERTION
2.INORDER
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice1
Enter Element To Be Insert92
```

```
Item 92 Is Successfully Inserted
TREE TRAVERSALS
1.INSERTION
2.INORDER
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice1
Enter Element To Be Insert45
Enter Key Element92
Where Do You Want To Insert Press Left(L) Right(R)
Item 45 Is Successfully Inserted
TREE TRAVERSALS
1.INSERTION
2.INORDER
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice1
Enter Element To Be Insert67
Enter Key Element92
Where Do You Want To Insert Press Left(L) Right(R)
Item 67 Is Successfully Inserted
TREE TRAVERSALS
1.INSERTION
2.INORDER
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice1
Enter Element To Be Insert38
Enter Key Element45
Where Do You Want To Insert Press Left(L) Right(R)
Item 38 Is Successfully Inserted
TREE TRAVERSALS
1.INSERTION
2.INORDER
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice1
Enter Element To Be Insert59
Enter Key Element45
Where Do You Want To Insert Press Left(L) Right(R)
Item 59 Is Successfully Inserted
TREE TRAVERSALS
1.INSERTION
2.INORDER
```

```
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice2
Tree Elements Are 38 45 59 92 67
TREE TRAVERSALS
1.INSERTION
2.INORDER
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice3
Tree Elements Are 92 45 38 59 67
TREE TRAVERSALS
1.INSERTION
2.INORDER
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice4
Tree Elements Are 38 59 45 67 92
TREE TRAVERSALS
1.INSERTION
2.INORDER
3.PREORDER
4.POSTORDER
5.EXIT
Enter Your Choice5
```

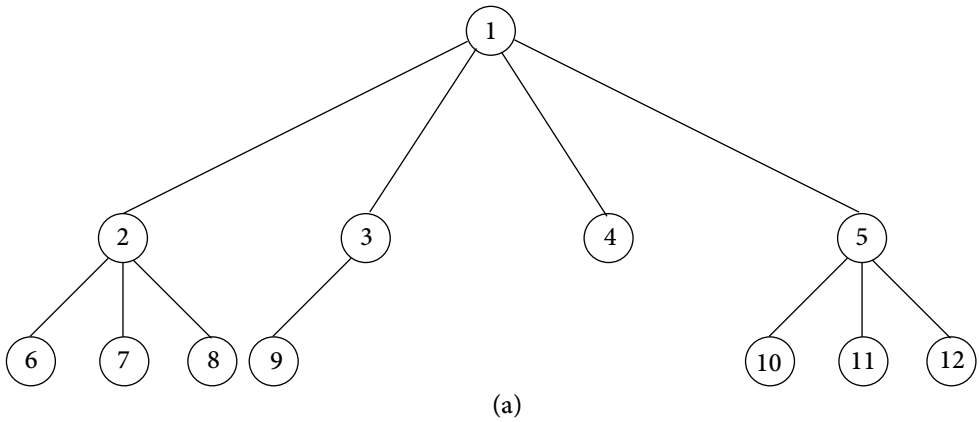
9.8 CONVERSION OF A TREE INTO A BINARY TREE

Representation of a binary tree in programs is easier to that of a tree. Therefore, a general tree should be converted into a binary tree. This can be done using the following procedure:

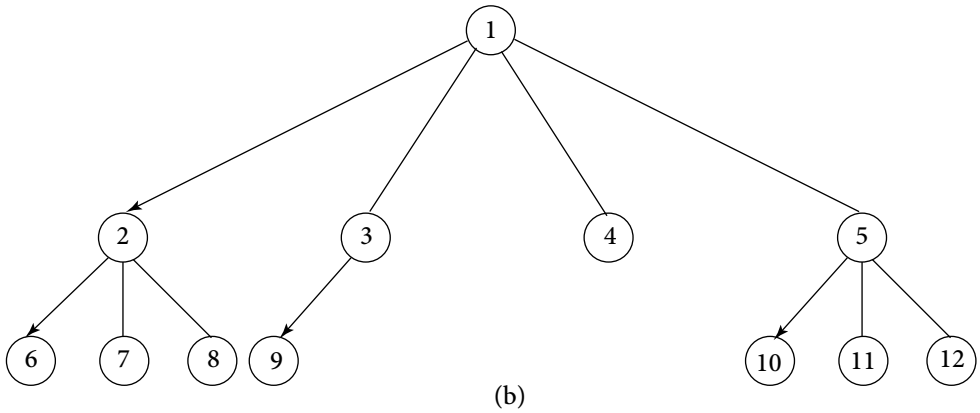
1. Identify the branch from the parent to its first child and these branches become left branch or subtree in the binary tree.
2. From the leftmost child use a branch to connect the node with a node to its right of the same parent if any, which are at the same level. These become the right branch or subtree in the binary tree.
3. Remove all branches that are not required from the parent to its children.

Consider a tree shown in Figure.9.11(a). According to the first step of the procedure, node 1 is the root node, its first child is node 2. Similarly, node 6 is the first child of 2, node 9 is the first child of node 3 and node 10 is the first child of node 5. So, 2, 6, 9 and 10 become the left branches to their parents. These are shown by the pointed links in Figure 9.11(b).

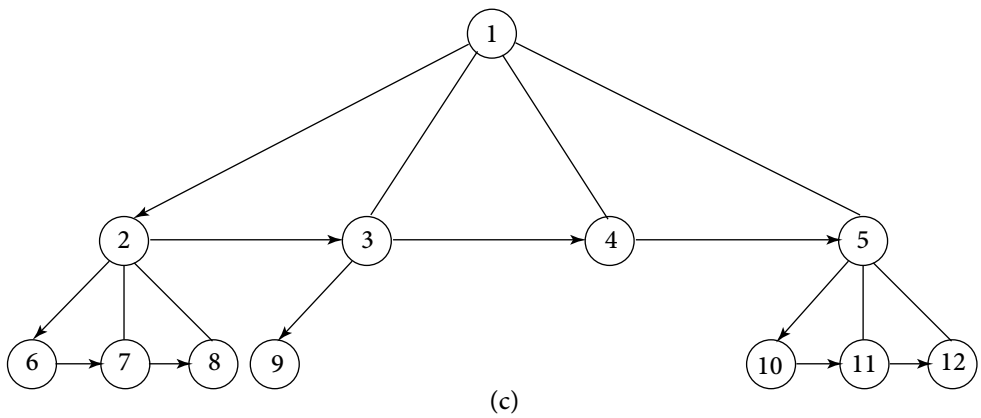
According to the second step of the procedure from the leftmost child, node 2 connects nodes 3, 4, 5 that are right to it, children of the same parent and at the same level. Similarly, from node 6 connect nodes 7, 8 and from node 10 connect nodes 11 and 12. These are shown by the pointed horizontal links in Figure 9.11(c). These become the right branches of the binary tree.



(a)

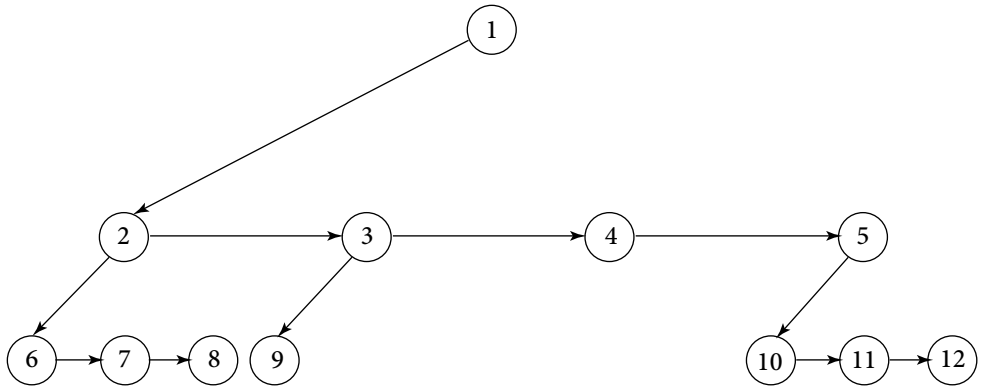


(b)

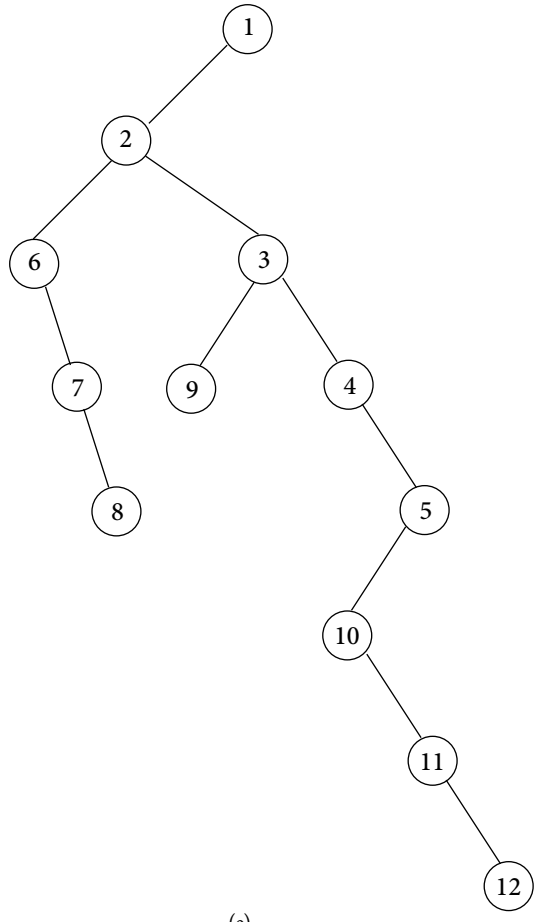


(c)

Figure 9.11 Conversion of tree into a binary tree



(d)



(e)

Figure 9.11 *Continued*

Remove the branches that are not required from parent to its children, such as 1 to 3, 1 to 4, 1 to 5, 2 to 7, 2 to 8, 5 to 11 and 5 to 12. Therefore, the resulting binary tree of a general tree shown in Figure 9.11(a) is shown in Figure 9.11(d). This is not in a regular binary tree format. So, redrawing the same gives a perfect shape of a binary tree as depicted in Figure 9.11(e).

9.9 THREADED BINARY TREES

The linked representation of a binary tree of Section 9.5 says that a binary tree with `n` nodes uses `2n` pointers of which `(n+1)` are null pointers. These `(n+1)` null pointers can be used to point some valuable information of the binary tree. To do this A. J. Perlis and C. Thornton introduced the concept of *threads*. Threads are also pointers or links. For a node `NODE` if `RCHILD(NODE)` is `NIL`, then the null pointer is replaced by a thread that is replaced by a thread that points to a node that occurs often, `NODE`, when the binary tree is traversed in inorder. If `LCHILD(NODE)` is `NIL` then this null pointer is replaced by a thread that points to a node that precedes `NODE` when the binary tree is traversed in inorder.

Thus, threads are pointers to the predecessor and successor of the node corresponding to an inorder traversal, and trees whose nodes use threads are called *threaded trees*. Threaded trees can be used for all the traversals of a binary tree. A threaded binary tree is illustrated in Figure 9.12. The threads are indicated by dotted lines.

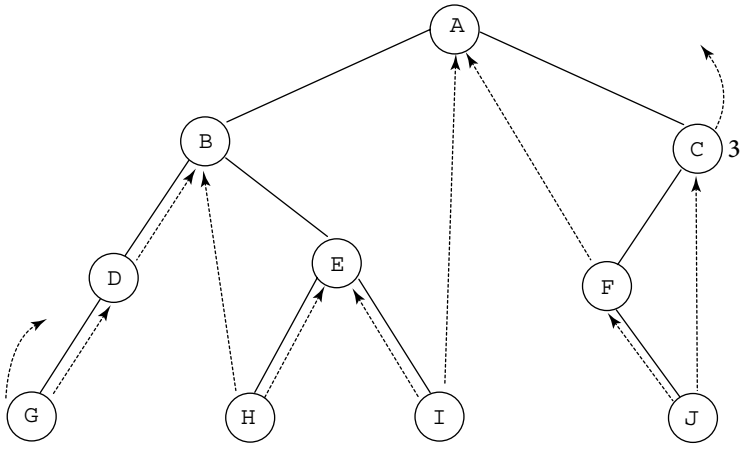


Figure 9.12 Threaded binary tree

As nodes `G` and `C` have no predecessor and successor, respectively, the left child of `G` and the right child of `C` have threads that are left dangling. Threading can be represented in two ways. They are one-way threading and two-way threading.

One-way threading: It is a representation in which a thread appears only on the `RCHID` field of the node, when it is null, pointing to the inorder successor of a node.

Two-way threading: It is a representation in which threads appear on both `LCHILD` and `RCHILD` fields of the node, when they are null, pointing to the inorder predecessor and successor of the node. The first and last nodes of the inorder traversal hold dangling threads. All the traversals of a binary tree, either recursive or non-recursive, use a stack to keep track of the nodes whose processing is yet to be completed. This requires additional time and space. It is still worse in the case of skewed binary trees where the stack needs to hold the information regarding almost all the nodes of a tree, a severe concern for very large trees. It would be very efficient if a stack can be incorporated as a part of a tree; this can be done by using threads.

To know that whether LCHILD and RCHILD fields are referring to the children of the node or to its predecessor and successor, there should be a node structure that clarifies this ambiguity.

9.9.1 Linked Representation of a Threaded Binary Tree

A linked representation of a threaded binary tree has a node structure as depicted in Figure 9.13.

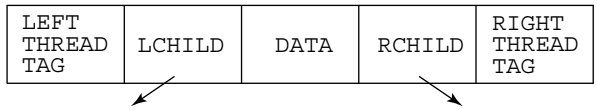
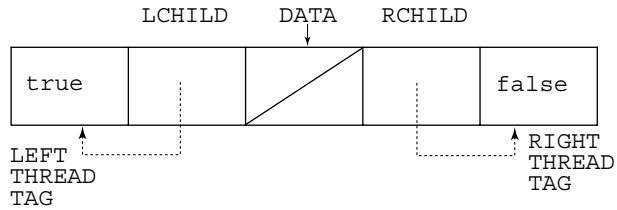
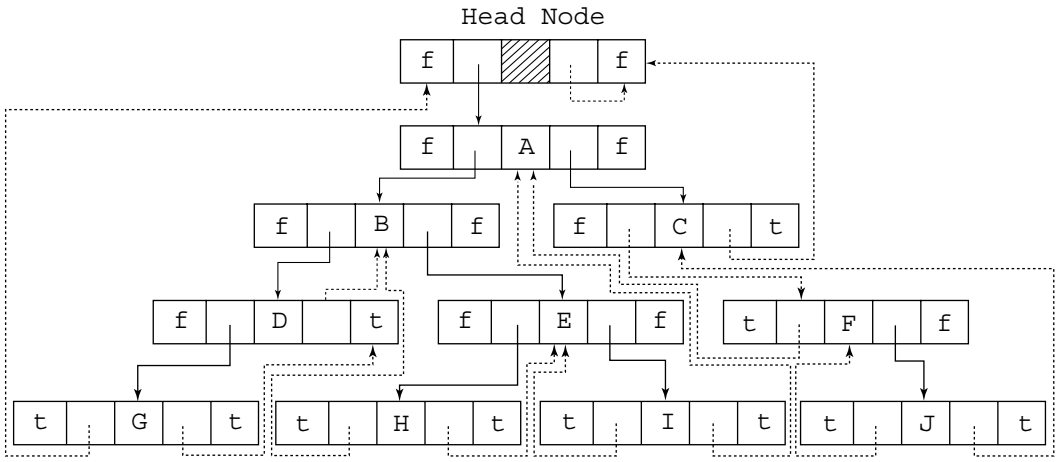


Figure 9.13 Node structure of a linked representation of a threaded binary tree

This node structure representation includes two additional fields to the linked representation of a binary tree. These two fields act as flags that indicate whether the LCHILD and RCHILD fields represent a thread or a link. If the LEFT THREAD TAG or RIGHT THREAD TAG fields are *true* then the corresponding LCHILD or RCHILD fields represent threads, else they represent links to the childs. The problem dangling threads can also be handled in this representation, because it has a head node to which dangling threads point to. By convention, the head node has its LCHILD pointing to the root node of the threaded binary tree. So the LEFT THREAD TAG is set to false but the RCHILD link points to the head node itself. Figure 9.14(a) shows the linked representation of an empty threaded binary tree and Figure 9.14(b) shows a non-empty threaded binary tree.



(a) Empty threaded binary tree



(b) Non-empty threaded binary tree

Figure 9.14 Linked representation of a threaded binary tree

9.10 APPLICATIONS OF BINARY TREES

Binary trees have a number of applications. Representing arithmetic, logical or relational expressions without any ambiguity is one among important applications. In an expression tree operands appear as leaf or external or terminal nodes and the operators appear as internal or non-terminal nodes. The operand may be either an identifier or a number.

A Polish logician's, Jan Lukasiewicz, invention in early 1920s regarding a special notation for propositional logic allows the removal of all parentheses from formulas. This is known as the Polish notation. This leads to less readability of formulas, but is very useful in computers, specially in writing compilers and interpreters. In order to provide both readability and unambiguity in the formulas, additional symbols like parentheses must be used. But, if avoiding ambiguity is the only goal then these symbols can be eliminated and this results in change in the order of symbols in the formulas. A complier does the same, i.e. it rejects the things that are not important in getting the proper meaning of formulas.

9.10.1 Traversal of an Expression Tree

Traversing an expression tree being a binary tree is the same as the traversal of a binary tree. Inorder traversal of an expression tree gives infix notation, preorder traversal gives prefix notation and postorder traversal gives postfix notation of an expression.

The expressions are represented in three ways, they are infix, prefix and postfix. A commonly used representation of an expression is infix, it appears as

`<operand1><operator><operand2>`

for instance `A+R, P*Q`

The prefix expression appears as `<operator><operand1>operand2>`

for instance `+AR, *PQ`

The postfix expression appears as `<operand1><operasnd2><operator>`

for instance `AR+, PQ+`

Generally, infix notation is used but some programming languages use Polish notation. For instance, `LISP` and to a large extent `LOGO` use prefix notation. `Forth` and `Postscript` use postfix notation.

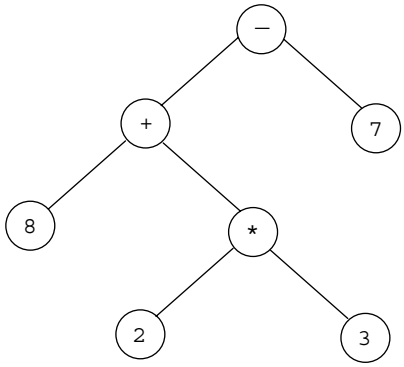
9.10.2 Operations on Expression Trees

Binary trees can be created either in top–down or bottom–up manner. Top–down approach is used for implementing insertions. Bottom–up approach is used in creating expression trees and scanning infix expressions from left to right.

While constructing an expression tree, it is a must to retain the precedence of operators and their associativity as they are in the expression for which an expression tree is constructed.

Let an algebraic expression 8+2*3–7 be considered. Its result is based on the order of evaluating the operations. If multiplication is done first followed by addition and subtraction, the result will be 7. If addition is done first followed by subtraction and multiplication as (8+2)*(3–7) the result will be –40. If addition is done after multiplication and subtraction as 8+ (2*3–7) the result will be 7.

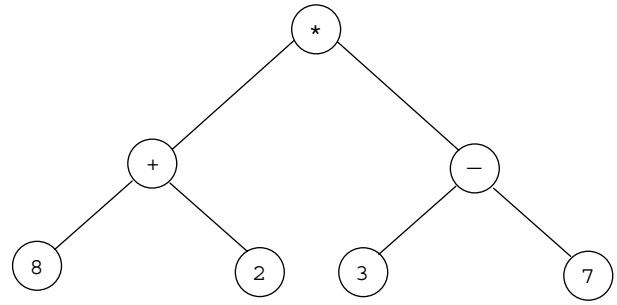
The complier generates an assembly code where only one operation is preformed at once and its result is retained for further operations. So, all the expressions must be broken down unambiguously into individual operations and kept in their proper order. Here Polish notation is useful. This allows one to create an expression tree which imposes an order in executing the operations. For example, the expression 8+2*3–7, which is the same as 8+ (2*3)–7 is represented by the tree in Figure 9.15(a). The second and third expressions are represented by the trees shown in Figures 9.15(b) and (c), respectively.



8+2*3−7

Preorder −+8*237
Inorder 8+2*3−7
Postorder 823*+7−

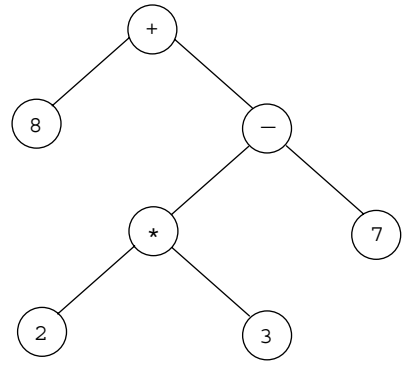
(a)



(8+2)*(3−7)

Preorder *+82−37
Inorder 8+2*3−7
Postorder 82+37−*

(b)



8+(2*3−7)

Preorder +8−*237
Inorder 8+2*3−7
Postorder 823*7−+

(c)

Figure 9.15 Examples of three expression trees and results of their transversals

There is no ambiguity in this tree representation. To compute the final result, the intermediate results are to be calculated first. Even with no parentheses no ambiguity is found. Expression trees are very convenient for performing symbolic operations such as differentiation.

9.11 ADT OF BINARY TREE

A binary tree of nodes; each node contains data fields `DATA` and two link fields `LCHILD` and `RCHILD`. `ROOT` is pointed to the root node.

Operations

- `isEmpty(ROOT)`
  Checks whether binary tree is empty.
- `TRAVERSE _ LCHILD(N)`
  Traversing the left subtree of node `N` by moving down its `LCHILD` pointer.
- `TRAVERSE _ RCHILD(N)`
  Traversing the right subtree of node `N` by moving down the `RCHILD` pointer.
- `INSERT(ROOT,ITEM)`
  Insert a node with `ITEM` as `DATA` as the root node.
- `LEFT _ INSERT(N,ITEM)`
  Insert a node with `ITEM` as `DATA` as left child of a node `N`.
- `RIGHT _ INSERT(N,ITEM)`
  Insert a node with `ITEM` as `DATA` as right child of a node `N`.
- `DELETE(ROOT)`
  Delete root node of a binary tree.
- `DELETE _ LCHILD(N)`
  Delete the left child of a node `N`.
- `DELETE _ RCHILD(N)`
  Delete the right child of a node `N`.
- `ADD _ DATA(LOC,ITEM)`
  Add `ITEM` into a node whose address is in `LOC`.
- `RETRIEVE _ DATA(LOC,ITEM)`
  Retrieve data of a node whose address is `LOC`.
- `INORDER(T)`
  Perform inorder traversal on binary tree `T`.
- `PREORDER(T)`
  Perform preorder traversal on binary tree `T`.
- `POSTORDER(T)`
  Perform postorder traversal on binary tree `T`.

SUMMARY

- A tree is a finite set of one or more nodes. Every node in a tree represents a unit of information. The links/arcs between the nodes are known as branches.
- A binary tree `T` is a finite set of elements. An empty tree is a binary tree.
- A binary tree of height `h` that consists of maximum permissible number of children to all nodes, i.e. 2^h-1 nodes is known as a full binary tree.
- A binary tree that has maximum permissible number of nodes at all the levels except at the last level is known as a complete binary tree. A binary tree that has only left or right children is known as a skewed binary tree.
- Binary trees can be represented using both sequential and linked data structures.
- Various operations can be performed on binary trees such as searching, insertion, deletion, copy and display.

- Visiting a node of a binary tree is known as traversal. Various traversal techniques are inoreder, preorder, postorder and level order.
- The inorder traversal starts with root node and traverses the left subtree and then the right subtree.
- The preorder traversal visits a node first then traverses its left subtree and then traverses its right subtree.
- The postorder traversal of a binary tree traverses the left subtree of a node first and traverses its right subtree and finally visits the node.
- The level-order traversal of a binary tree traverses the nodes in a level-by-level manner from top to bottom and among the nodes of the same level they are traversed from left to right.
- Threads are pointers or links. Threaded trees are the trees whose nodes use threads.

EXERCISES

Fill in the blanks

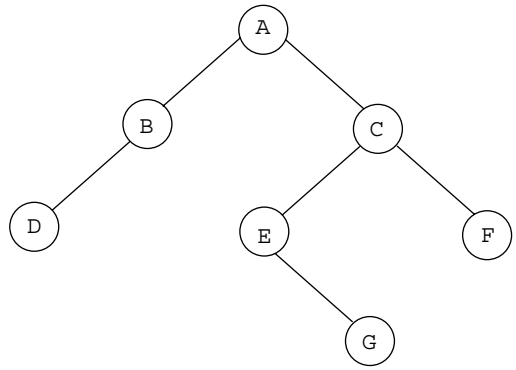
1. The number of branches associated with a node is called the _____ of a node.
2. A binary tree with n elements will have _____ number of edges.
3. _____ cannot be a tree.
4. The height of a tree is _____ in the tree.
4. Visiting each node in a tree exactly once is known as _____ .
5. Trees whose nodes use threads are called _____ .

Multiple-choice Questions

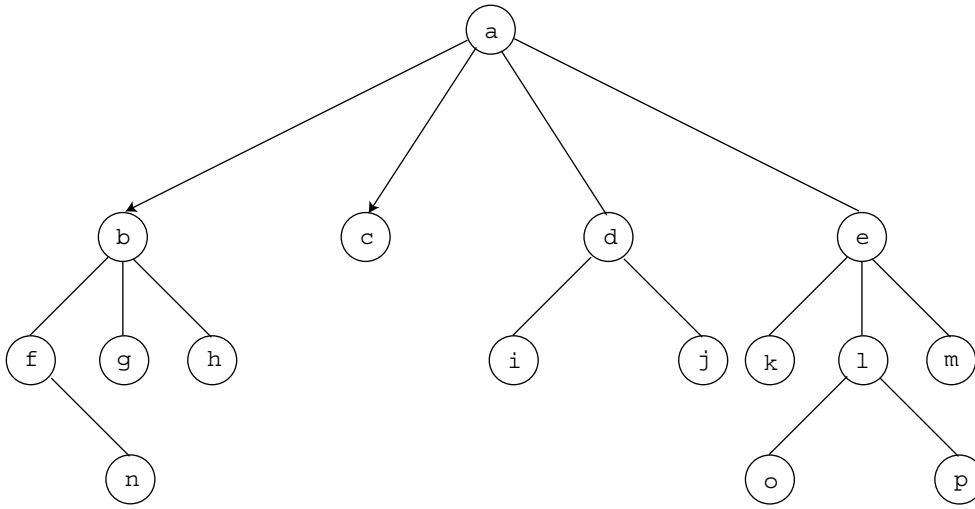
1. A binary tree that has only left or right children is known as _____ .
   a. Complete binary tree b. Full binary tree
   c. Skewed binary d. None.
2. A binary tree with n elements will have _____ number of edges.
   a. n-1 b. 2n c. n+1 d. 2n-1
3. At level i a binary tree can have at most _____ number of nodes.
   a. 2^i b. 2^{i+1} c. 2^{i-1} d. 2^i-1
4. The height of a complete binary tree with n nodes in h is _____ .
   a. $\log_2(n+1)$ b. $\log_2 n$ c. $\log_2(n-1)$ d. $\log 2^{n+1}$
5. In an expression tree, operands appear as _____ .
   a. Leaf node b. Child node c. Parent node d. None

Short-answer Questions

1. Define trees.
2. Give the basic terminology of trees.
3. Discuss the representation of trees.
4. For the below binary tree, find
   i. ROOT ii. Children of C
   iii. Leaf nodes iv. Siblings
   v. Height of the tree



5. Convert the following into a binary tree:



Essay Questions

1. Explain the binary trees and their representation with an example.
2. Discuss various operations on binary trees.
3. Illustrate various traversal techniques on binary trees.
4. Write a program to implement level-order traversal algorithm.
5. Write a program to convert a tree into a binary tree.

Chapter 10

Graphs

This chapter provides a brief introduction to the concept of graphs and discusses the basic terminology of graphs. Various popular and widely used representations of graphs are explained along with examples. Different operations that are frequently applied on graphs are explained with algorithms, examples and their implementations. Breadth and Depth first traversals of a graph are also exemplified. The chapter also includes the detailed discussion of the Prim's algorithm regarding the minimal cost spanning tree of a graph among the listed applications, followed by the Abstract Data Type of graph.

10.1 INTRODUCTION

Graph theory is applied in diverse areas such as social sciences, physical sciences, communication engineering and others. Graph theory also plays an important role in several areas of computer science such as Artificial Intelligence, Formal Languages, Computer Graphics, Operating Systems and Compiler Design. Graphs are widely used in the representation of data structures.

10.2 BASIC TERMINOLOGY

Graphs have various terms that one has to know before discussing other related aspects.

Graph: A graph `G=(V,E)` consists of two sets. A set `V` called the set of all vertices or nodes and a set `E` called the set of all edges or links or arcs. The set `E` is the set of pair of elements from `V`.

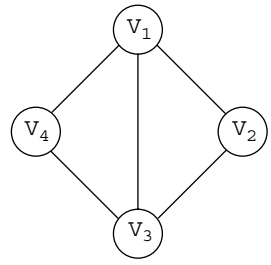


Figure 10.1 Graph G_1

In Figure 10.1, $V=\{v_1,v_2,v_3,v_4\}$ and $E=\{(v_1,v_2),(v_1,v_3),(v_1,v_4),(v_2,v_3),(v_3,v_4)\}$.

A pair of vertices which are connected by an edge in a graph are called adjacent vertices.

Labeled graph: If the vertices and/or edges of a graph G are given with some data, then it is known as a labeled graph. It is easier to refer edges with labels. This is depicted in Figure 10.2.

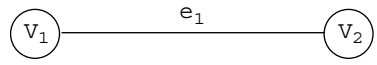


Figure 10.2 Graph G_2

In Figure 10.2, v_1 and v_2 are adjacent vertices. The edge (v_1,v_2) can also be represented as e_1.

Directed graph: A graph in which every edge is directed is called a directed graph or simply a digraph. Edges are ordered pairs of vertices. An edge e_{ij} is referred to as $<v_i, v_j>$ where v_i,v_j are distinct vertices, v_i is tail and v_j is the head of the edge.

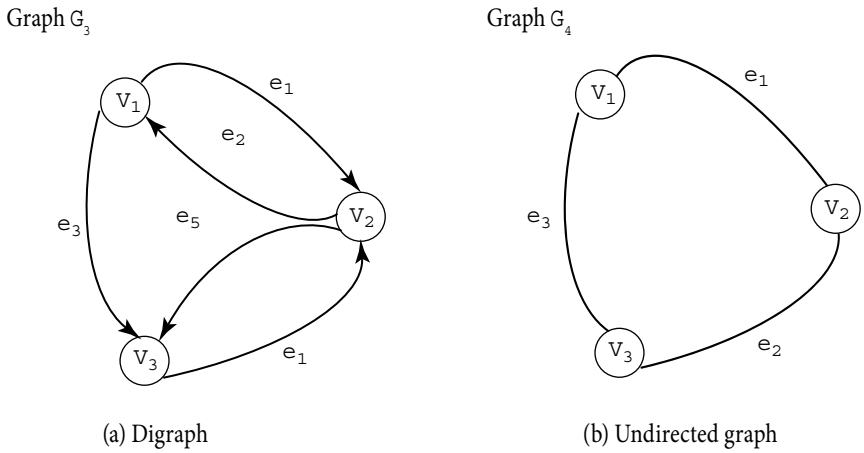


Figure 10.3 Graphs G_3 and G_4

Undirected graph: A graph in which every edge has no direction (undirected) is called an undirected graph. Edges are unordered pairs of vertices. An edge e_{ij} can be referred as (v_i,v_j) and is same as (v_j,v_i) where v_i, v_j are distinct vertices.

In Figure 10.3(a), e_1 is a directed edge between nodes v_1 and v_2, i.e. $e_1=<v_1,v_2>$. Here v_1 is said to be adjacent to v_2, and v_2 is said to be adjacent from v_1. In Figure 10.3(b) e_1 is an undirected edge between v_1 and v_2, i.e. $e_1=(v_1,v_2)$ and is the same as (v_2,v_1). Here v_1,v_2 are said to be adjacent. The lists of vertices and edges of graphs G_3 and G_4 are

Vertices $(G_3) = \{v_1,v_2,v_3\}$
Vertices $(G_4) = \{v_1,v_2,v_3\}$
Edges $(G_3) = \{<v_1,v_2>,<v_2,v_1>,<v_1,v_3>,<v_3,v_2>,<v_2,v_3>\}$ or $\{e_1,e_2,e_3,e_4\}$
Edges $(G_4) = \{(v_1,v_2),(v_1,v_3),(v_2,v_3)\}$ or $\{e_1,e_2,e_3\}$

Mixed graph: In a graph if some edges are directed and some edges are undirected then it is said to be a mixed graph. This is shown in Figure 10.4.

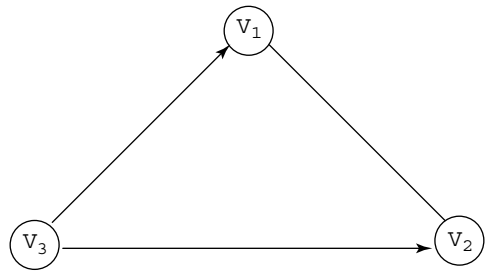


Figure 10.4 Graphs G_5

Loop: An edge of a graph which joins a vertex to itself is called a loop. In Figure 10.5, e_2 is a loop.

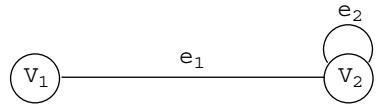


Figure 10.5 Graphs G_6

Parallel edges: In a graph, if a pair of vertices are joined by more than one edge, such edges are called parallel edges. This is shown in Figure 10.6.

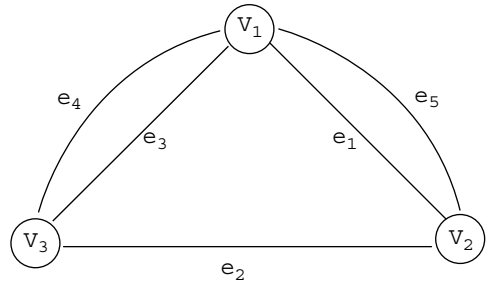


Figure 10.6 Graphs G_7

Here in Figure 10.6, edges e_4, e_3 and e_1, e_5 are parallel edges.

Simple graph: Any graph without parallel edges and loops is known as a simple graph as depicted in Figure 10.7.

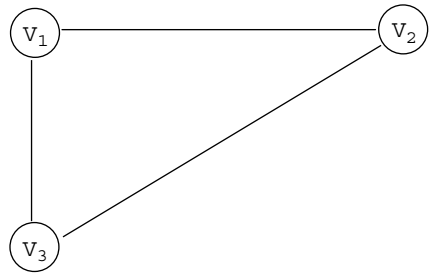


Figure 10.7 Graphs G_8

Multigraph: Any graph which contains some parallel edges and loops is called a multigraph.

Trivial graph: A finite graph with one vertex and no edges, i.e. single vertex, is called a trivial graph.

Isolated vertex: In any graph, a vertex which is not adjacent to any other vertex is called an isolated vertex.

Null graph: A graph with no vertices or with only isolated vertices is called a null graph. The set of edges in a null graph is empty.

Weighted graph: A graph in which weights are assigned to every edge is called a weighted graph.

Finite graph and infinite graph: If the number of vertices and edges in a graph are finite, then it is called a finite graph otherwise it is called an infinite graph.

Order and size of a graph: Let G be a graph, then V(G) or V denotes the set of vertices, and E(G) or E denotes the set of edges in graph G.

|V(G)| or |V| denotes the number of vertices in G and is called order of G. Similarly, |E(G)| or |E| denotes the number of edges in G and is called size of G.

Incidence: If a vertex v_i is an end vertex of an edge e_j, then v_i and e_j are said to be incident with one another.

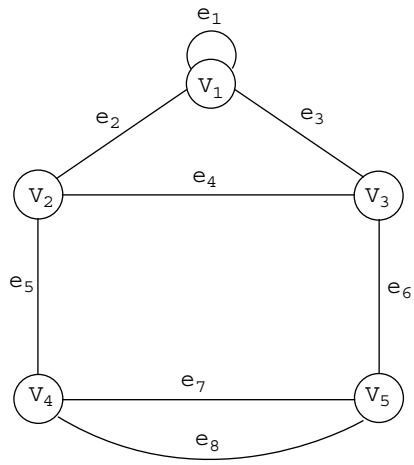


Figure 10.8 Graphs G_9

In Figure 10.8, edges e_3, e_4 and e_6 are incident on vertex v_3. Edges e_6, e_7 and e_8 are incident on vertex v_5 and so on.

Degree of a vertex: The number of edges incident on vertex v_i, where self-loop is counted twice, is called the degree of a vertex v_i. It is denoted by deg(v_i). In Figure 10.8, deg(v_1)=4; deg(v_2)=3.

Indegree of a directed graph: The number of edges incident to a vertex v_i is called the indegree of the vertex v_i.
Outdegree of a directed graph: The number of edges incident from a vertex v_i is called the outdegree of the vertex v_i. If the degree of a vertex is one then the vertex is called a *pendant* vertex.

minimum degree $\delta(G)=0$

maximum degree $\Delta(G)=4$

Subgraph: A graph $G^1=(V^1,E^1)$ is a subgraph of a graph G=(V,E) when $V^1 \subseteq V$ and $E^1 \subseteq E$. Figure 10.9 illustrates a subgraph.

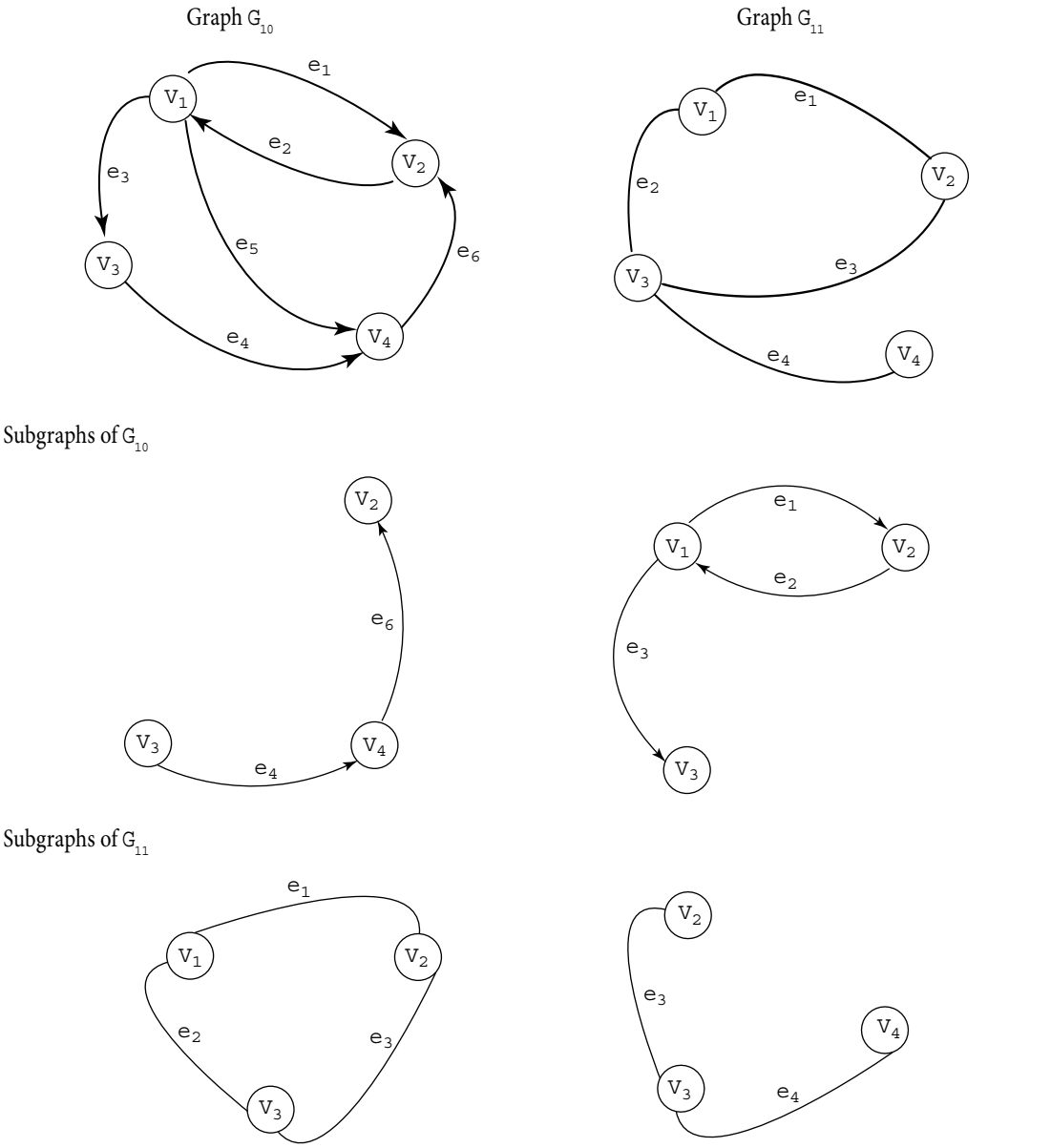
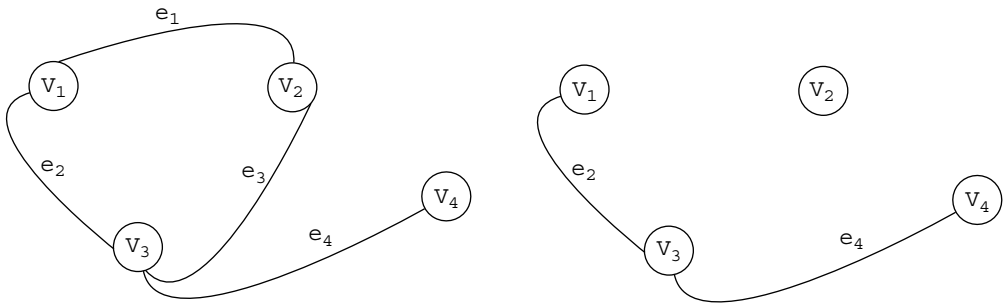


Figure 10.9 Subgraphs

Cut set: Removal of a set of edges from a connected graph G makes G disconnected, then this set of edges is called a cut set. No proper subset of these edges should disconnect the graph G. Cut sets are also referred to as a proper cut set or minimal cut set or cocycle.



(a) Graph G_{12} (b) Cut set of the graph G_{12}

Figure 10.10 Cut set of a graph

The cut set $\{e_1, e_3\}$ disconnects the graph G_{23} as shown in Figure 10.10. Similarly, $\{e_4\}$,$\{e_2, e_3\}$ are other cut sets of the graph.

Connected graph: In a graph G, two vertices v_i and v_j are said to be connected if there is a path in G from v_i to v_j. A graph is said to be connected if there is path between every pair of distinct vertices v_i, v_j in G.

If a graph is not connected, the maximal connected subgraph is known as a **connected component** or simply a **component**. This is shown in Figure 10.11.

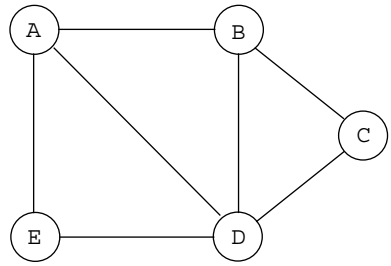
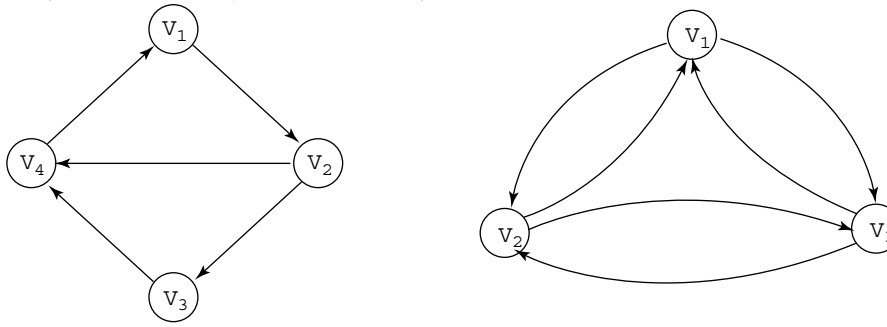


Figure 10.11 Connected graph G_{13}

A directed graph that holds the connectedness property is known as a **strongly connected graph**. A digraph G is said to be strongly connected if for every pair of distinct vertices v_i, v_j in G, there is a directed path from v_i to v_j and also from v_j to v_i. If a digraph is not strongly connected, then the graph is said to be **weakly connected**. Figure 10.12 shows types of connected graphs.



G_{14}: Weakly connected graph G_{15}: Strongly connected graph

Figure 10.12 Connected digraphs

Path: In an undirected graph G, a sequence P of zero or more edges of the form $\{v_0,v_1\}$, $\{v_1,v_2\}$, …, $\{v_{n-1},v_n\}$ or v_0–v_1– … v_n is called a path from v_0 to v_n, where v_0 is the initial vertex and v_n is the terminal vertex of the path P.

In the definition of path, vertices and edges may be repeated.

If $v_0=v_n$, then P is called a **closed path**.

If $v_0 != v_n$, then P is called an **open path**.

Circuit: A path of length ≥1 with no other repeated vertices or edges except its starting and ending vertices is called a circuit.

Length of the path: The number of edges appearing in the sequence of the path P is called the length of the path.

If the length of the path P is zero, i.e. path P has no edges at all, it contains only a single vertex and is called a **trivial path**.

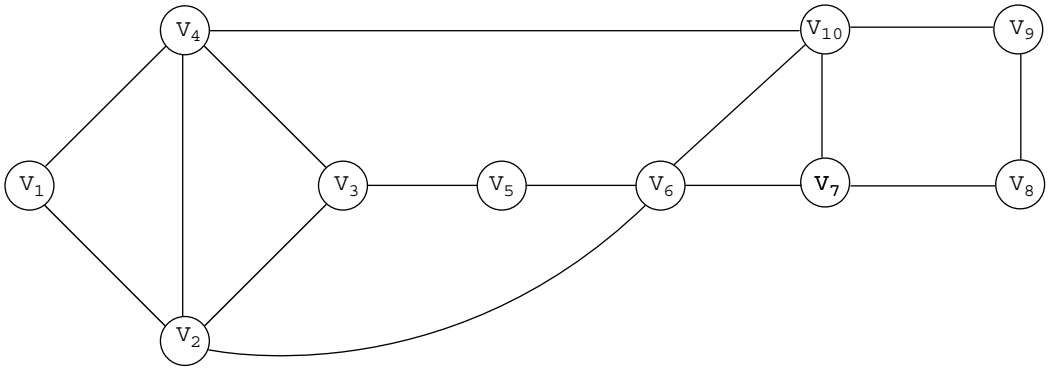


Figure 10.13 Graph G_{16}

From the graph G_{16} shown in Figure 10.13, the following observations can be made.

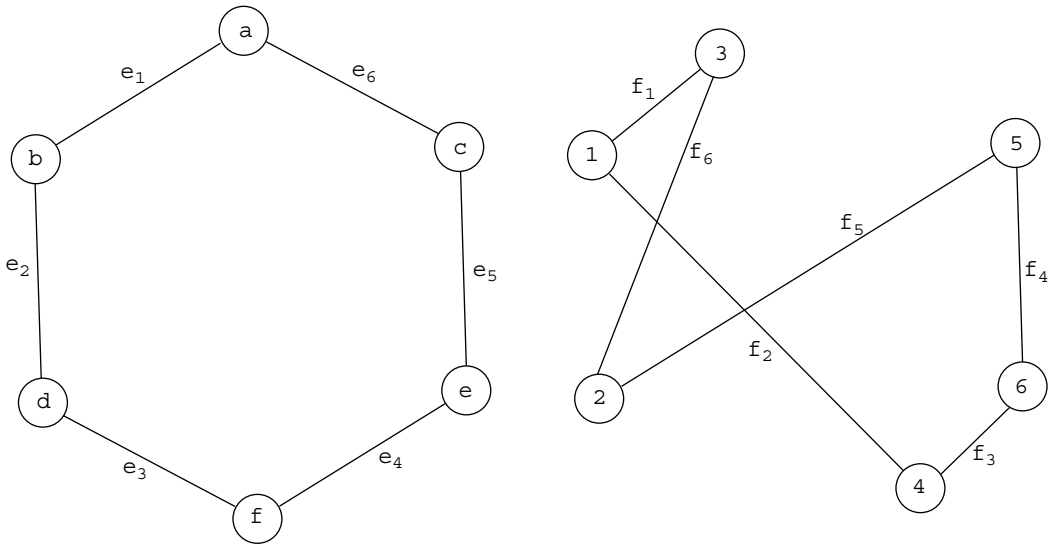
Consider the path v_2–v_3–v_5–v_6–v_7–v_{10}–v_6–v_2. Here, the length of the path is 7 as it has 7 edges involved in the path. Though the length of the path is more then 1, as v_6 is repeated it is not a circuit. As the end vertices are the same it is a closed path.

Isomorphism: Two graphs G and G^1 are said to be isomorphic, there is a one-to-one correspondence between their vertices and between their edges such that the incident relationship is preserved.

In other words, if v_1 and v_2 are the vertices in graph G, v_1 and v_2 are the corresponding vertices in graph G′, then for an edge having v_1 and v_2 as the end vertices in G, there is a corresponding edge in G^1 with v_1', and v_2', as the end vertices. Then G and G^1 are said to be isomorphic.

Conditions for isomorphism:

1. Both graphs should have the same number of vertices.
2. Both graphs should have the same number of edges.
3. Both graphs should have an equal number of vertices with the same degree.
4. Both graphs should have the same cycle vector $(c_1,c_2, \ldots,c_n)$, where c_i= number of cycles of length i.



(a) Graph G_{17} (b) Graph G_{18}

Figure 10.14 Graphs G_{17} and G_{18} demonstrate isomorphism

Consider the graphs G_{17} and G_{18} shown in Figure 10.14. Both the graphs have 6 vertices and 6 edges. Also there exists one to one correspondences among them, which can be expressed as

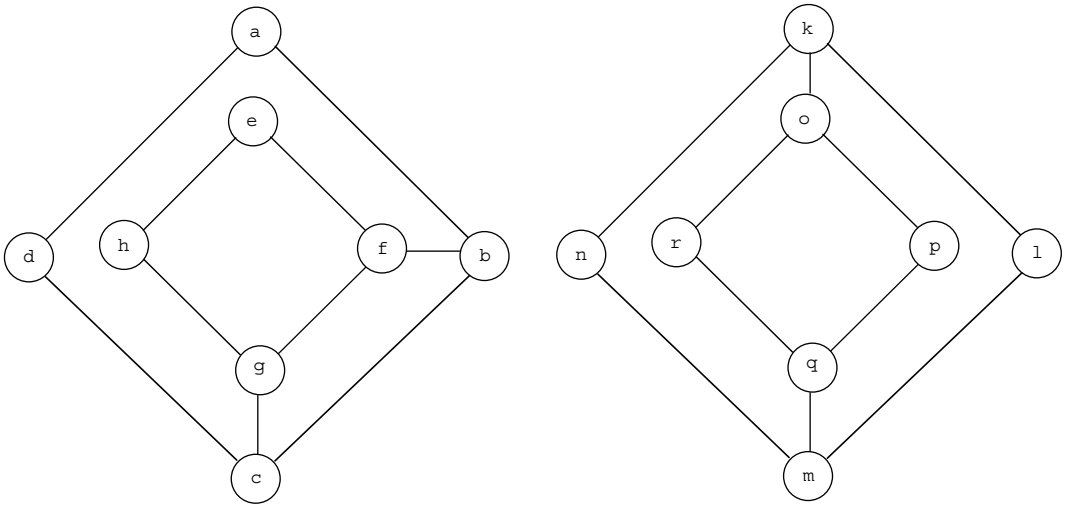
Vertices a–3, b–1, c–2, d–4, e–5, f–6

Edges e_1–f_1, e_2–f_2, e_3–f_3, e_4–f_4, e_5–f_5, e_6–f_6

Degree of all the vertices in G_{17} a, b, c, d, e, f is 2 and

Degree of all the vertices in G_{18} 1, 2, 3, 4, 5, 6 is also 2

Therefore, the above graphs G_{17} and G_{18} are isomorphic. Consider the graphs G_{19} and G_{20} shown in Figure 10.15.



(a) Graph G_{19} (b) Graph G_{20}

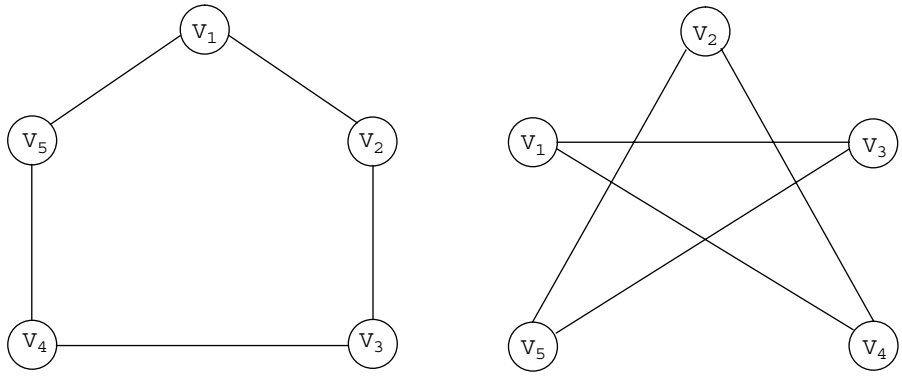
Figure 10.15 Graphs G_{19} and G_{20}: non-isomorphic graphs

The following observations can be made:

1. $|V_{19}|=|V_{20}|=8$
2. $|E_{19}|=|E_{20}|=10$
3. Degrees are the same
4. In G_{19} the length of the cycle is 8 and in G_{20} the length of the cycle is 6, cycle vector is not the same. Therefore, the two graphs G_{19} and G_{20} are not isomorphic.

Complement of a graph: The complement of a graph G is denoted by G^1 on the same vertices if any only if two vertices that are adjacent in G are not adjacent in G^1.

Figure 10.16 shows the two graphs G_{21} and G_{22}; the above said characteristic of complement graph is satisfied. Therefore, graph G_{22} is a complement of the graph G_{21}.



(a) Graph G_{21} (b) Graph G_{22}

Figure 10.16 Graph G_{21} and its complement G_{22}

Note: If two graphs are isomorphic then their complements will also be isomorphic.

Eulerian circuit: If every edge of the graph G appears exactly once in the path which starts and ends at the same vertex, it is called an Eulerian circuit.

A graph which has an Eulerian circuit is called an Euler graph.

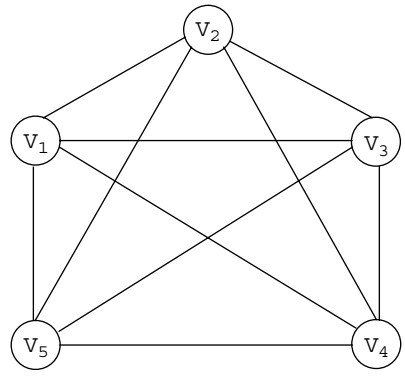


Figure 10.17 Graph G_{23}: Euler graph

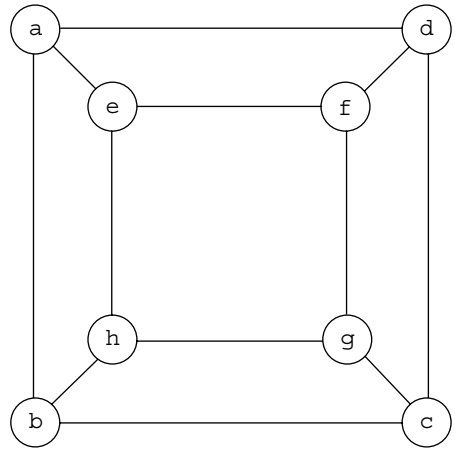
In graph G_{23} shown in Figure 10.17.

Euler circuit: $v_1-v_2-v_4-v_5-v_1-v_3-v_5-v_2-v_3-v_4-v_1$

As the graph G_{23} has an Euler circuit it is an Euler graph.

Hamiltonian path: Hamiltonian path in a connected graph is defined as a path that traverses each vertex of the graph exactly once, i.e. the terminal vertices of the Hamiltonian path are distinct.

Hamiltonian circuit: Hamiltonian circuit is a circuit that passes through all the vertices in a graph exactly once except the starting vertex because the same will also be the terminating vertex. A graph which has a Hamiltonian circuit is called a Hamiltonian graph. Figure 10.18 illustrates Hamiltonian cycle and Hamiltonian circuit.



Hamiltonian circuit is: a-b-c-d-f-g-h-e-a

Figure 10.18 Graph G_{23}: Hamiltonian graph

10.3 REPRESENTATION OF GRAPHS

Graphs can be represented in two ways:

1. Sequential representation
2. Linked representation

All the methods of sequential representation use matrices. So, arrays are used for implementing them. In the linked representation, a singly linked list is used as a basic data structure.

10.3.1 Sequential Representation of Graphs

Some of the popular sequential representation methods are:

- Adjacency matrix representation
- Incidence matrix representation

Adjacency matrix representation: For a graph G=(V,E) of n vertices the adjacency matrix is an n×n symmetric binary matrix, and every entry of this matrix is defined as

a_{ij}=1, if there is an edge between v_i and v_j

a_{ij} = 0, if there is no edge between v_i and v_j.

i.e. it only shows the direct paths between the vertices.

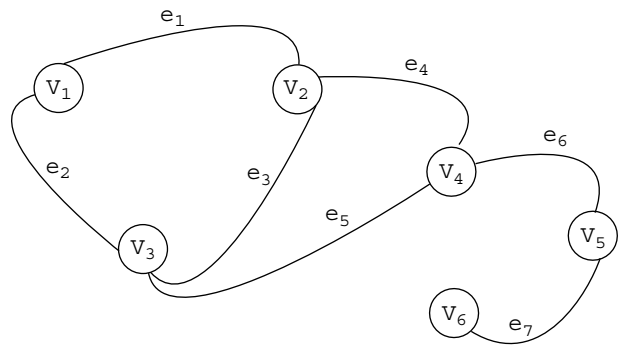


Figure 10.19 Graph G_{25}

The adjacency matrix representation of graph G_{25} in Figure 10.19 is

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	1	0	0	0
v_2	1	0	1	1	0	0
v_3	1	1	0	1	0	0
v_4	0	1	1	0	1	0
v_5	0	0	0	1	0	1
v_6	0	0	0	0	1	0

This representation is the best and widely used representation. Its time complexity is $O(n^2)$ because minimum n^2-n entries, excluding diagonal elements, must be examined.

Incidence matrix representation: For a graph $G=(V,E)$ of n vertices and e edges the incidence matrix, is an $n \times e$ matrix, and every entry of this matrix is defined as

a_{ij}=1, if an edge e_j is incident on vertex v_i
a_{ij}=0, otherwise

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	1	1	0	0	0	0	0
v_2	1	0	1	1	0	0	0
v_3	0	1	1	0	1	0	0
v_4	0	0	0	1	1	1	0
v_5	0	0	0	0	0	1	1
v_6	0	0	0	0	0	0	1

Incidence matrix of graph G_{25}

10.3.2 Linked Representation of Graphs

The linked representation of a graph G=(V,E) with n vertices and e edges creates n head nodes with respect to n vertices. Each of these head nodes refers to a singly linked list that holds the adjacent vertices of the vertex corresponding to the head node.

This representation is also known as an adjacent list representation. This is also referred to as a star representation when implemented as a table.

The linked representation is considered to be efficient over its counterpart of sequential representation with a time complexity of O(n+e) for directed and O(n+2e) for undirected graphs. Figure 10.20 shows the adjacency list representation of graph G_{25}.

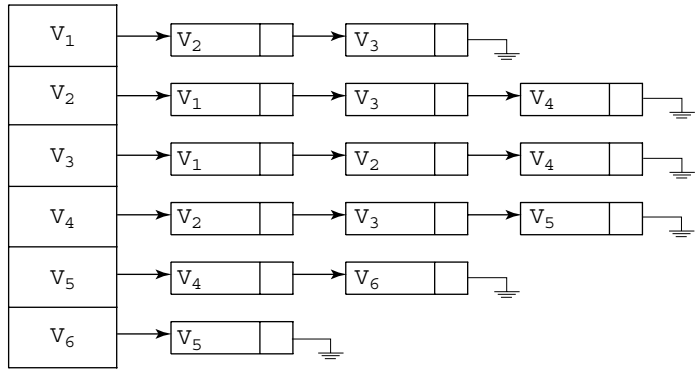


Figure 10.20 The adjacency list representation of graph G_{25}

10.4 OPERATIONS ON GRAPHS

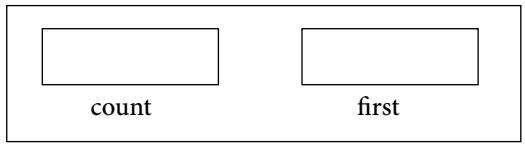
Several operations can be performed on graphs. The primitive operations that are required to maintain a graph are:

1. Create a graph
2. Insertion of a vertex
3. Deletion of a vertex
4. Insertion of an edge
5. Deletion of an edge
6. Search for a vertex
7. Traversal of a graph
8. Deletion of or destroy a graph

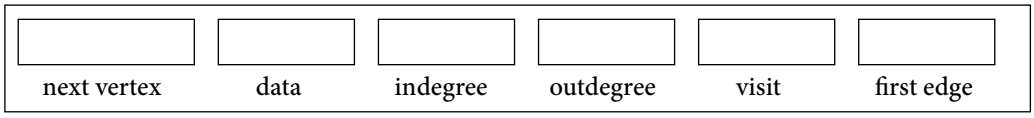
Other operations such as vertex count, vertex's indegree, vertex's outdegree, etc. are application dependent.

Data structure of a graph: Adjacency list representation can be used to store the graph. A head structure can be added to hold the metadata regarding the list. Here only count of vertex in a graph is stored along with the pointer to the first vertex. Vertex node holds the data about the vertex. The node structure of the vertex is chosen to contain six fields. In addition to the data, pointers to next vertex and first edge, it also contains indegree,

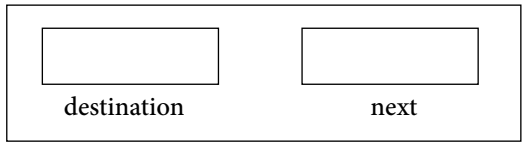
outdegree and visit fields. The indegree field is very helpful regarding deletion. Deletion of a vertex is allowed only when its indegree is zero, i.e. no edge should point to it. Otherwise deletion is not allowed and leads to failure of program. The visit field is a flag and is useful during traversal to know whether it is already visited or not. If it is true then the vertex is traversed else it is not traversed. The edge details are stored in its structure that holds two fields, destination and next edge. The graph data structure is shown in Figure 10.21.



(a) Graph head



(b) Graph vertex



(c) Graph edge

Figure 10.21 Graph data structure

Create a graph: Initially a graph should be created; this can be done by initializing the metadata elements of a graph head structure. Algorithm 10.1 gives the procedure of creating a graph.

Algorithm 10.1

```
creategraph
 1. Set count to 0.
 2. Set first to NULL
 3. Return graph head.
End creategraph.
```

Insertion of a vertex: Insert vertex operation is used to add a vertex to a graph. After insertion, the new vertex will be an isolated or a disjoint vertex which is not connected with any other vertices. This is the first step of the insertion process. To connect this vertex with another vertex, an edge associated with this must be inserted separately. The insertion of a vertex into a graph is shown in Figure 10.22.

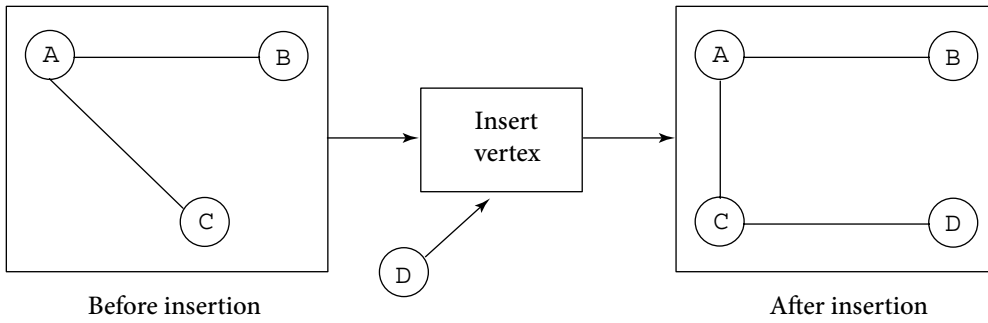


Figure 10.22 Insertion of a vertex

Algorithm 10.2 explains the procedure of inserting a vertex into a graph.

Algorithm 10.2

```
insertvertex(graph, data)
1. GETNODE(AVAIL)                  //memory allocation for new vertex.
2. Set data to new vertex
3. Initialize metadata elements in new node
4. Increment graph count           //finding the insertion point or where to
                                   //insert the new vertex
5. If graph is empty
     Set graphfirst to new node.
6. Else
   (i)  Search for position of insertion
   (ii) If  insertion as a first vertex set  graphfirst to new vertex
   (iii)Else
        Insert new vertex (sequentially) as last vertex
   (iv)  End if
7. End if
End insertvertex.
```

Deletion of a vertex: Delete vertex operation is used to remove a vertex from a graph. To delete a vertex it should be searched. If found, it should be ensured that no edges are coming to or going out of the vertex, then delete the vertex else deletion is not allowed. Figure 10.23 shows the deletion of a vertex.

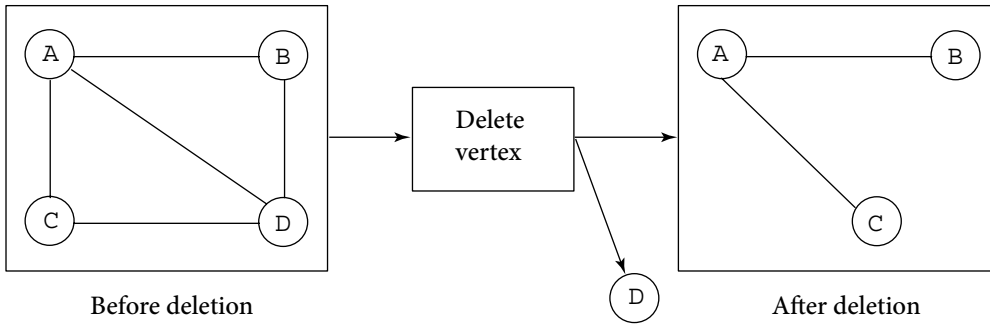


Figure 10.23 Deletion of a vertex

Algorithm 10.3 explains the procedure of deleting a vertex from a graph.

Algorithm 10.3

```
deletevertex(graph, element)
1. If graph is empty
      Print "element not found".
2. End If
3. Search for the vertex to be deleted
4. If not found
       Print "element not found"
5. Else
     1. If indegree(vertex)>0 or outdegree(vertex)>0
          Print "deletion is not allowed"
     2.  Else
          deletevertex
     3. Decrement graph count
     4. Print "vertex deleted successfully"
6. End deletevertex.
```

Insertion of an edge: Insert edge operation is used to add an edge that connects a vertex to a destination vertex. To insert an edge, two vertices must be specified. In case of digraph one vertex must be specified as the source and the other as a destination. If a vertex requires multiple edges, this operation must be invoked once to connect with each of its adjacent vertices. Figure 10.24 shows the insertion of an edge {B, D} in a graph.

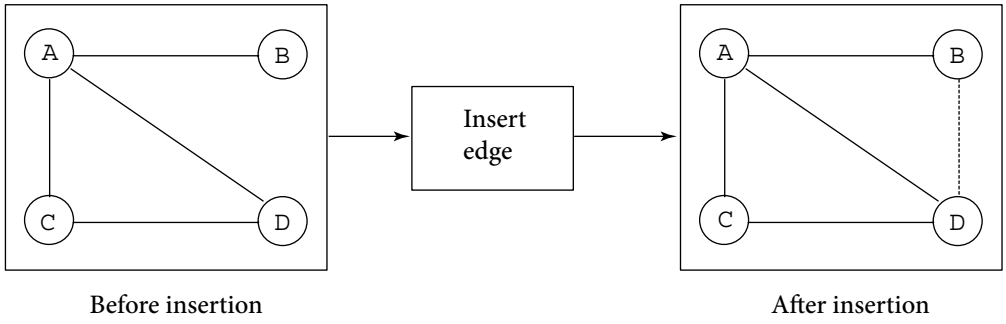


Figure 10.24 Insertion of an edge

Algorithm 10.4 details how an edge is inserted into a graph.

Algorithm 10.4

```
insertedge(graph, fromvertex, tovertex)
1. GETNODE(AVAIL)   // memory allocation for new edge
2. Find and set fromvertex //identify the source vertex.
3. If fromvertex not found
             Print "fromvertex not found"
4. End if
5. Find and set tovertex //identify destination vertex.
6. If tovertex not found.
           Print "tovertex not found".
```

```
7. End if
8. Increment outdegree of fromvertex.
9. Increment indegree of tovertex.
10. Set edge destination to tovertex.
11. If edge list of fromvertex is empty.
              1. Set first edge of fromvertex to new edge.
              2. set next edge of new edge to null
              3. Print "insertion is successful"
12. End if
13. Search for insertion point in adjacency list
14. If insertion as a first edge
               Set first edge of fromvertex to new edge
15. Else
         Insert in edge list.
16. End if
17. Print "insertion successful"
18. End insertedge.
```

Deletion of an edge: Delete edge operation is used to remove an edge from a graph. To delete an edge the two vertices source and destination must be identified. So, search the vertex list to find the source vertex. Identify the destination vertex from the adjacency list of the source vertex. Now, the edge is located and removed from the adjacency list. Adjust the degree of from and to vertices and recycle the memory. Deletion of an edge {B, D} is shown in Figure 10.25.

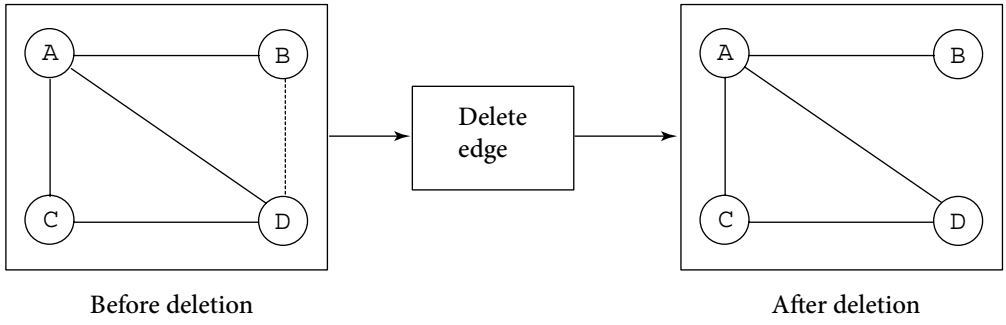


Figure 10.25 Deletion of an edge

Algorithm 10.5 gives the process of deleting an edge from a graph.

Algorithm 10.5

```
delete edge(graph, fromvertex, tovertex)
1. If graph is empty
            Print "source vertex not found".
2. End if
3. Find and set fromvertex    //identify source vertex.
4. If fromvertex not found.
          Print "from vertex not found".
5. End if
//identify the destination vertex in adjacency list.
```

```
6. If edge list of fromvertex is null.
              Print "tovertex not found".
7. End if
8. Find the tovertex and set the edge.
9. If tovertex not found
           Print "tovertex not found".
10. End if
    //fromvertex, tovertex and edge are located. Now delete edge.
11. Set tovertex to edge destination.
12. Delete edge.
13. Decrement the outdegree of fromvertex.
14. Decrement the indegree of tovertex.
15. Print "edge successfully deleted"
16. End deleteedge.
```

Search for a vertex: Search vertex operation is used to return the data that is stored in a vertex. This is very similar to the search operation of a linked list. Algorithm 10.6 explains the procedure of searching for a vertex in a graph.

Algorithm 10.6

```
searchvertex(graph, key, item)
1. If graph is empty
        Print "search is not successful as the graph is empty".
2. End if
3. Search for vertex.
4. If vertex is found
                1. Set data(vertex) to item.
                2. Print "search is successful".
5. Else
        Print "search is not successful"
6. End if
7.End searchvertex.
```

Destroy a graph: When a delete vertex operation is performed it should be isolated first by deleting its edges. Destroy graph operation initially deletes all the edges of a vertex before it deletes a vertex. Deleting all vertices including the corresponding edges is destroying a graph. Algorithm 10.7 discusses the method of destroying a graph.

Algorithm 10.7

```
destroygraph(graph)
1. If graph is empty
        Return
2. If graph head count>0  loop
      1. If outdegree of vertex>0loop
                1. Delete first edge of vertex
                2. Decrement the outdegree of vertex by 1
      2. End loop
3. End loop
4. End destroygraph.
```

Implementation of operations: The implementation of above-discussed operations are explained in Program 10.1.

Program 10.1

```
// To PERFORM GRAPH OPERATIONS USING LINKED LIST REPRESENTATION
#include<iostream.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>
typedef struct graph
{
 int vertex,weight;
 struct graph*next;
}GRAPH;
GRAPH*adj[100],*ptr;
int n,count=0,a,b,i;
void insertvt()//To Insert Vertex
{
 cout<<"Enter Vertex To Be Insert";
 cin>>a;
 cout<<"Enter Reference Vertex";
 cin>>b;
 if(a>n||a<1||b>n||b<1)
 {
  cout<<"Wrong Input";
  return;
 }
 if(count!=0)
 {
  ptr=adj[b];
  while(ptr->vertex==b)
   ptr=ptr->next;
  if(ptr==NULL)
  {
   cout<<"Vertex Not Found";
   return;
  }
 }
 ptr=(graph*)malloc(sizeof(graph));
 ptr->vertex=b;
 ptr->weight=1;
 ptr->next=adj[a];
 adj[a]=ptr;
 if(count==0)
  count+=2;
 else
  count++;
 cout<<"Vertex"<<a<<"Is Inserted And Edge Is Established B/w"<<a<<"To"<<b;
}
void insertedg()//To Insert Edge
```

```
{
 int wt;
 cout<<"Enter First Vertex";
 cin>>a;
 cout<<"Enter Second Vertex";
 cin>>b;
 if(a>n||a<1||b>n||b<1)
 {
  cout<<"Wrong Input";
  return;
 }
 ptr=adj[a];
 while(ptr!=NULL)
 {
  if(ptr->vertex==b)
  {
   cout<<"Vertex Already Exists";
   return;
  }
  ptr=ptr->next;
 }
 ptr=(graph*)malloc(sizeof(graph));
 ptr->vertex=b;
 ptr->weight=1;
 ptr->next=adj[a];
 adj[a]=ptr;
 cout<<"Edge Is Inserted";
}
void deletevt()//To Delete Vertex
{
 if(count==0)
 {
  cout<<"Graph Is Empty";
  return;
 }
 cout<<"Enter Vertex To Be Delete";
 cin>>a;
 if(a>n||a<1)
 {
  cout<<"Vertex Not Found";
  return;
 }
 for(i=1;i<=n;i++)
 {
  ptr=adj[i];
  while(ptr!=NULL)
  {
   if(ptr==adj[a])
```

```
    adj[i]=NULL;
    ptr=ptr->next;
   }
  }
  count--;
  cout<<"Vertex"<<a<<"Is Deleted";
}
void deletedg()//To Delete Edge
{
  GRAPH*prev;
  if(count==0)
  {
   cout<<"Graph Is Empty";
   return;
  }
  cout<<"Enter First Vertex";
  cin>>a;
  cout<<"Enter Second Vertex";
  cin>>b;
  if(a>n||a<1||b>n||b<1)
  {
   cout<<"Wrong Input";
   return;
  }
  ptr=adj[a];
  while(ptr!=NULL&&ptr->vertex!=b)
   ptr=ptr->next;
  if(ptr==NULL)
  {
   cout<<"Wrong Input";
   return;
  }
  if(ptr==adj[a])
  {
   adj[a]=ptr->next;
   free(ptr);
   return;
  }
  prev=adj[a];
  while(prev->next!=ptr)
   prev=prev->next;
  prev->next=ptr->next;
  free(ptr);
  cout<<"Edge Is Deleted";
}
void display()//To Display Edges
{
  GRAPH *ptr;
```

```
 int i;
 cout<<"Edges In The Given Graph\n";
 for(i=1;i<=n;i++)
 {
  ptr=adj[i];
  while(ptr!=NULL)
  {
   cout<<i<<"->"<<ptr->vertex<<" "<<endl;
   ptr=ptr->next;
  }
 }
}
void main()
{
 int op,x,i;
 clrscr();
 cout<<"Enter Maximum No Of Nodes";
 cin>>n;
 for(i=1;i<=n;i++)
 adj[i]=NULL;
 while(1)
 {
  cout<<"\nGraph Operations\n1.Insert Vertex\n2.Insert Edge";
  cout<<"\n3.Delete Vertex\n4.Delete Edge\n5.Display\n";
  cout<<"6.Exit\nEnter Your Option";
  cin>>op;
  switch(op)
  {
   case 1:insertvt();break;
   case 2:insertedg();break;
   case 3:deletevt();break;
   case 4:deletedg();break;
   case 5:display();break;
   case 6:exit(0);
   default:cout<<"Invalid Choice";
  }
 }
}
```

Output

```
//OUTPUT FOR GRAPH OPERATIONS USING LINKED LIST REPRESENTATION
Enter Maximum No Of Nodes5
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
```

```
6.Exit
Enter Your Option1
Enter Vertex To Be Insert1
Enter Reference Vertex2
Vertex 1 Is Inserted And Edge Is Established B/w 1 To 2
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option1
Enter Vertex To Be Insert3
Enter Reference Vertex1
Vertex 3 Is Inserted And Edge Is Established B/w 3 To 1
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option2
Enter First Vertex2
Enter Second Vertex3
Edge Is Inserted
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option5
Edges In The Given Graph
1->2
2->3
3->1
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option1
Enter Vertex To Be Insert5
```

```
Enter Reference Vertex1
Vertex 5 Is Inserted And Edge Is Established B/w 5 To 1
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option1
Enter Vertex To Be Insert4
Enter Reference Vertex3
Vertex 4 Is Inserted And Edge Is Established B/w 4 To 3
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option5
Edges In The Given Graph
1->2
2->3
3->1
4->3
5->1
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option4
Enter First Vertex5
Enter Second Vertex1
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option5
Edges In The Given Graph
1->2
```

```
2->3
3->1
4->3
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option3
Enter Vertex To Be Delete4
Vertex 4 Is Deleted
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option5
Edges In The Given Graph
1->2
2->3
3->1
Graph Operations
1.Insert Vertex
2.Insert Edge
3.Delete Vertex
4.Delete Edge
5.Display
6.Exit
Enter Your Option6
```

10.5 GRAPH TRAVERSALS

Similar to the case of trees, traversal visits elements of the nodes in a particular manner. Graphs can be traversed in two ways. They are:

- Breadth first traversal
- Depth first traversal

10.5.1 Breadth First Traversal

An arbitrary vertex `V` is selected to begin the traversal, visit the vertex. All the adjacent vertices of `V`, a_i are visited. Now, the unvisited adjacent vertices of each a_i, b_{ij} are visited. This process continues till all the vertices of the graph are visited. The vertices whose adjacent vertices are to be visited are kept in queue.

Algorithm 10.8

```
breadthfirsttraversal(G)
This algorithm performs breadth first traversal starting from the vertex V of
graph G
/*QUE is a queue, keeping track of vertices whose adjacency nodes are to be
visited.
 Vertices which are visited are set to 1
 Initially all the vertices in a graph are set to 0*/
1. Initialize queue QUE
2. Set Visit(V) to 1
3. Insert the vertex V into QUE
4. Loop till QUE is empty then perform deletion from QUE
5. Display the visited vertex
6. Loop for all adjacent vertices A of V
7. Check whether visit(A) is zero then insert vertices into QUE
8. Set VISIT(A) to 1
9. End loop
10. End loop
11. End
```

Let the graph G_{26} in Figure 10.26 be used to discuss the traversal.

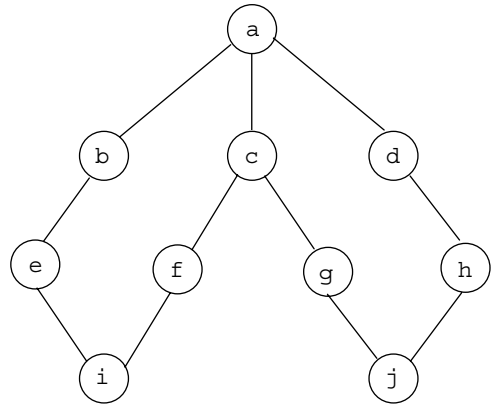


Figure 10.26 Graph G_{26} BST

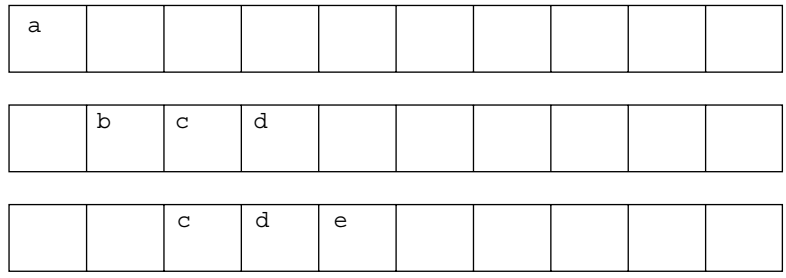


Figure 10.27 Queue notation of BST

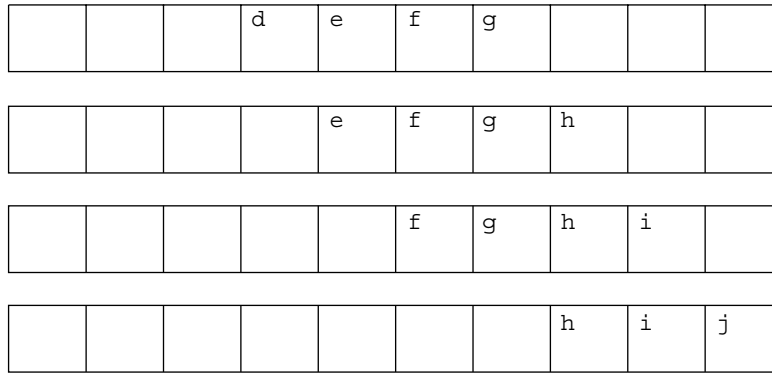


Figure 10.27 *Continued*

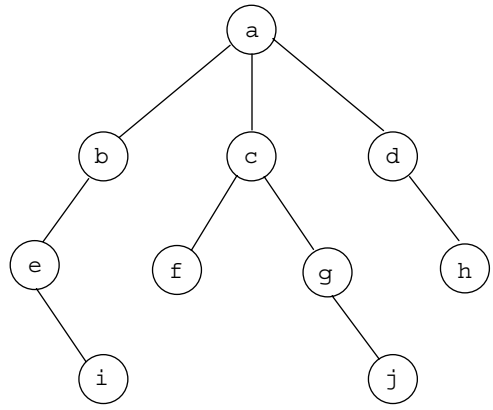


Figure 10.28 Result of BST

The vertex 'a' is chosen to begin the traversal and 'a' is visited, enque 'a'. Deque 'a' to visit. The vertices 'b', 'c,' 'd' that are adjacent to 'a', enque 'b', 'c', 'd'. The traversal output at this stage is a. Deque 'b' to visit its only adjacent vertex 'e', enque 'e'. The traversal output at this stage is ab. Deque 'c' to visit its two adjacent vertices 'f', 'g' and then enque the traversal output at this stage is abc. Now, deque 'd' to visit its adjacent vertex 'h' and enque it. The traversal output at this stage is abcd. Deque 'e' to visit its adjacent vertex i, enque 'i'. The traversal output at this stage is abcde. Deque 'f', its only adjacent vertex 'i' is already visited. The traversal output at this stage is abcdef. So, proceed to deque 'g' to visit its adjacent vertex 'j', enque it. Now the traversal output is abcdefg. Deque 'h', its only adjacent node 'j' is also already visited. So, no enque is required. The traversal at this stage is abcdefgh. Deque 'i', it does not have any unvisited vertices, so no enque is required. The traversal at this stage is abcdefghi. Deque 'j' the queue is empty as all its adjacent vertices are already visited, no further enquening is required. The final traversal output is abcdefghij.

The queue notation of the breadth first traversal of the graph G_{26} is shown in Figure 10.27 and the result of this is depicted in Figure 10.28.

Program 10.2

```
//PROGRAM FOR BREADTH FIRST TRAVERSAL OF A GRAPH
#include<iostream.h>
#include<conio.h>
char queue[20],g[20],x,delet();
int a[20][20],m,n,i,j,state[20],front=0,rear=-1;
void insert(char),bfs();
void main()
{
 clrscr();
 cout<<"Enter No Of Nodes Of Graph:";
 cin>>n;
 cout<<"Enter The Nodes Of Graph (In Alphabets):";
 for(i=1;i<=n;i++)
 cin>>g[i];
 cout<<"Instructions\nPress 1 If Edge Exists Otherwise Press 0\n";
 for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
 {
  cout<<g[i]<<"To"<<g[j]<<":";
  cin>>a[i][j];
 }
 cout<<"The Order Of Visiting The Nodes:";
 // Make State Of Each Node To Ready
 for(i=1;i<=n;i++)
 state[i]=1;
 state[1]=2;
 insert(g[1]);
 bfs();
 getch();
}
void bfs()
{
 x=delete();
 for(i=1;i<=n;i++)
 if(g[i]==x)
 break;
 state[i]=3;
 cout<<g[i];
 // Make State Of Each Traversed Node As Visisted
 for(j=1;j<=n;j++)
 {
  if(a[i][j]==1&&state[j]==1)
  {
   state[j]=2;
   insert(g[j]);
  }
 }
```

```
 if(front<=rear)
 bfs();
}
void insert(char x)
{
 rear++;
 queue[rear]=x;
}
char delete()
{
 char x;
 x=queue[front];
 front++;
 return(x);
}
```

Output

```
/*BFS OUTPUT*/
Enter No Of Nodes Of Graph:8
Enter The Nodes Of Graph (In Alphabets):A
B
C
D
E
F
G
H
Instructions
Press 1 If Edge Exists Otherwise Press 0
A To B : 1   A To C : 1   A To D : 1
A To E : 0   A To F : 0   A To G : 0
A To H : 0   B To A : 1   B To C : 0
B To D : 0   B To E : 1   B To F : 0
B To G : 0   B To H : 0   C To A : 1
C To B : 0   C To D : 0   C To E : 0
C To F : 0   C To G : 1   C To H : 1
D To A : 1   D To B : 0   D To C : 0
D To E : 0   D To F : 0   D To G : 1
D To H : 0   E To A : 0   E To B : 1
E To C : 0   E To D : 0   E To F : 1
E To G : 0   E To H : 0   F To A : 0
F To B : 0   F To C : 0   F To D : 0
F To E : 1   F To G : 0   F To H : 1
G To A : 0   G To B : 0   G To C : 1
G To D : 1   G To E : 0   G To F : 0
G To H : 1   H To A : 0   H To B : 0
H To C : 1   H To D : 0   H To E : 0
H To F : 1   H To G : 1
The Order Of Visiting The Nodes: ABCDEGHF
```

10.5.2 Depth First Traversal

The depth first traversal arbitrarly chooses vertex V to begin the traversal and visits the vertex. All the adjacent vertices of V, a_1 are considered but only the first adjacent vertex a_1 is visited leaving the other vertices. The difference between the breadth first and depth first traversal can be majorly observed from this point onwards. Unlike breadth first traversal only one of the adjacent vertices is picked, traversed and continues to traverse from its adjacent vertex till all the vertices are visited. Now among the adjacent vertices of a_1, b_{1j} only the first adjacent vertex b_{11} is visited and proceeds in the same manner till no more vertices remain unvisited. Stack is used to keep track the vertices whose adjacent vertices are to be visited.

Algorithm 10.9

```
depthfirsttraversal(G)
This algorithm performs depth first traversal. V is the starting vertex  of
graph G.
1. Set VISIT(V) to 1
2. Display the visited vertex V
3. Loop for each adjacent vertex A of V
4. Check whether VISIT(A) is zero then call DFT(A)
5. End Loop
6. End
```

The graph G_{26} in Figure 10.29 is used to discuss the traversal.

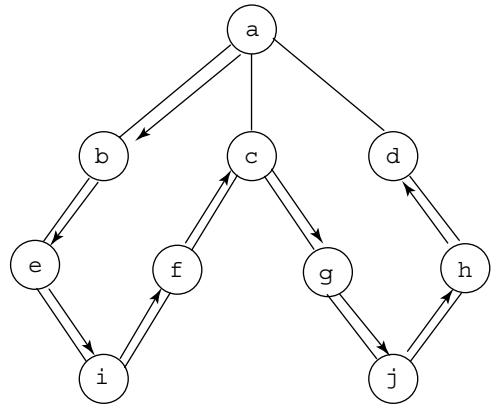


Figure 10.29 Result of DST

The vertex 'a' is chosen arbitrarily to begin the traversal and it is visited. Its three adjacent vertices 'b', 'c' and 'd' are considered, but only 'b', the first adjacent vertex, is selected and visited. The output of the traversal is ab. The adjacent vertex of b, 'e' is visited and the output of the traversal is abe. Vertex 'i' adjacent to 'e' is visited and the output is abei. The adjacent vertex of 'i', 'f' is visited and the output of the traversal is abeif. After visiting 'c', the adjacent vertex of 'f', the output of the traversal is abeifc. Now, visit 'g', the adjacent vertex of 'c' to obtain the output of the traversal as abeifcg. The adjacent vertex 'j' of 'g' is visited and the output of the traversal is abeifcgj. The vertex 'h' adjacent to 'j' is visited and the output is abeifcgjh. The vertex 'd' adjacent to 'h' is visited; at this stage all the vertices are visited. So, the final output of the traversal is abeifcgjhd.

The time complexity of depth first traversal() in case of adjacency matrix implementation is O(n2) and in case of adjacency list implementation is O(e). The result of the depth first traversal is shown in Figure 10.29.

The two traversals breadth first and depth first produce the traversal of all the connected vertices of a graph. If any vertex is left unvisited in either of the traversal then that would be an unconnected vertex and the graph also will be an unconnected graph. So, both these traversals help in finding whether a graph is connected or not. If the final output includes all the vertices then the graph is said to be connected else it is an unconnected graph.

Program 10.3

```
//PROGRAM FOR DEPTH FIRST TRAVERSAL OF A GRAPH
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
char stack[20],x,pop(),g[20];
int top=-1,n,a[20][20],i,j,state[20],count=0;
void push(char),dfs();
void main()
{
 clrscr();
 cout<<"Enter No Of Nodes Of Graph:";
 cin>>n;
 cout<<"Enter The Nodes Of Graph(In Alphabets):";
 for(i=1;i<=n;i++)
 cin>>g[i];
 cout<<"Instructions\nPress 1 If Edge Exists Otherwise Press 0\n";
 for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
 {
  cout<<g[i]<<"To"<<g[j]<<":";
  cin>>a[i][j];
 }
 cout<<"The Order Of Visiting The Nodes:";
 // Make State Of Each Node To Ready
 for(i=1;i<=n;i++)
 state[i]=1;
 state[1]=2;
 dfs();
 getch();
}
void dfs()
{
 if(count==0)
 push(g[1]);
 x=pop();
 for(i=1;i<=n;i++)
 if(g[i]==x)
 break;
 state[i]=3;
```

```
 cout<<g[i];
 count++;
 if(count==n)
 {
  getch();
  exit(0);
 }
 //Make State Of Each Traversed Node As Visisted
 for(j=n;j>=1;j--)
 {
  if(a[i][j]==1&&state[j]!=3)
   push(g[j]);
 }
 if(top!=-1)
 dfs();
}
void push(char x)
{
 top++;
 stack[top]=x;
}
char pop()
{
 char x;
 x=stack[top];
 top--;
 return(x);
}
```

Output

```
Enter No Of Nodes Of Graph:10
Enter The Nodes Of Graph (In Alphabets):A
B
C
D
E
F
G
H
I
J
Instructions
Press 1 If Edge Exists Otherwise Press 0
A To A : 0    A To B : 1    A To C : 1
A To D : 1    A To E : 0    A To F : 0
A To G : 0    A To H : 0    A To I : 0
A To J : 0    B To A : 1    B To B : 0
B To C : 0    B To D : 0    B To E : 1
```

```
B To F : 0    B To G : 0    B To H : 0
B To I : 0    B To J : 0    C To A : 1
C To B : 0    C To C : 0    C To D : 0
C To E : 0    C To F : 1    C To G : 1
C To H : 0    C To I : 0    C To J : 0
D To A : 1    D To B : 0    D To C : 0
D To D : 0    D To E : 0    D To F : 0
D To G : 0    D To H : 1    D To I : 0
D To J : 0    E To A : 0    E To B : 1
E To C : 0    E To D : 0    E To E : 0
E To F : 0    E To G : 0    E To H : 0
E To I : 1    E To J : 0    F To A : 0
F To B : 0    F To C : 1    F To D : 0
F To E : 0    F To F : 0    F To G : 0
F To H : 0    F To I : 1    F To J : 0
G To A : 0    G To B : 0    G To C : 1
G To D : 0    G To E : 0    G To F : 0
G To G : 0    G To H : 0    G To I : 0
G To J : 1    H To A : 0    H To B : 0
H To C : 0    H To D : 1    H To E : 0
H To F : 0    H To G : 0    H To H : 0
H To I : 0    H To J : 1    I To A : 0
I To B : 0    I To C : 0    I To D : 0
I To E : 1    I To F : 1 I To G : 0
I To H : 0    I To I : 0    I To J : 0
J To A : 0    J To B : 0    J To C : 0
J To D : 0    J To E : 0    J To F : 0
J To G : 1    J To H : 1    J To I : 0
J To J : 0
The Order Of Visiting the Nodes: ABEIFCGJHD
```

10.6 APPLICATIONS

The following are the some of the applications of graphs:

- Minimum cost spanning tree
- Single-source shortest path
- Union-find problem
- Connectivity of components

Minimum cost spanning tree can be found using either Prim's or Krushkal's algorithms. The following is the illustration of Prim's algorithm to find the minimum cost spanning tree.

A graph G of n vertices is considered, its spanning tree with minimum cost is to be found. Using Prim's algorithm an arbitrary vertex is considered as a root node. Among various incident edges, a minimum cost edge is selected to connect the root node with other nodes b, similarly this node is connected with another node. This process is continued till all the nodes are connected. Finally a minimum cost spanning tree of a given graph is obtained.

Algorithm 10.10

```
prim(G)
//G(V,E) is a weighted connected undirected graph and E¹ is the set of edges
//of the obtained minimum cost spanning tree
1. Set E¹ to NULL
2. Select minimum cost edge(U,V) from E
3. Assign u to V¹
4. Loop till V¹ is not equal to V
   1. Suppose (U,V) is the lowest cost edge such that V¹={u} and V is in
      V- V¹
   2. Add (U,V) edge to E¹
   3. Add V to V¹
   4. End loop
5. End
```

Graph G_{27} depicted in Figure 10.30 is used to illustrate this.

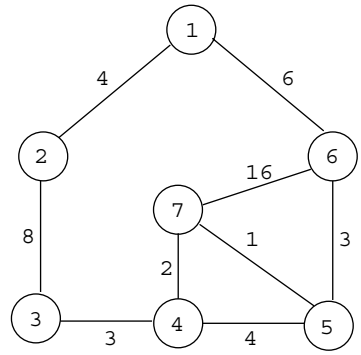


Figure 10.30 Graph G_{27}

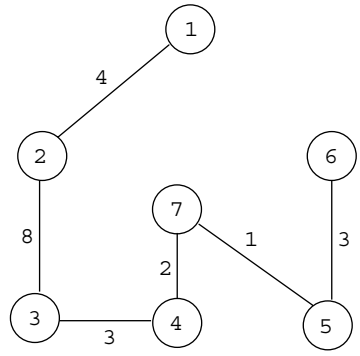


Figure 10.31 Spanning tree T of Graph G_{27}

Consider node 1 as the root node, it has two incident edges with 4 and 6 as their costs. The edge with minimum cost 4 is selected and used to connect 1 with 2. Node 2 has only one edge with cost 8; this is used to connect nodes 2 and 3. Node 3 also has only one edge with cost 3 and is used to connect nodes 3 and 4. Node 4 has two incident edges with 4 and 2 as their cost. Edge with cost 2 is selected to connect node 7 with 4. Node 7 has two incident edges with costs 1 and 16, the edge with cost 1 is used to connect node 5 with 7. Node 5 has only one edge with cost 3 to proceed to an unvisited node 6. So, it is used to connect 6 with 5. Now, all the nodes are connected using minimum possible cost edges resulting in a minimum cost spanning tree T of graph G_{27} such that $T \subseteq G$ is a subgraph where $G=(V,E)$ and $T=(V^1,E^1)$ (Figure 10.31). The time complexity of Prim's algorithm is $O(n^2)$.

10.7 GRAPH ADT

A graph G consisting of vertices and edges:

- is empty (G)
  Check whether graph G is empty
- INSERT (G,V)
  Insert vertex V into graph G
- INSERT(G,U,V)
  Insert an edge connecting vertices U and V
- DELETE(G,V)
  Delete vertex V from graph G and all the edges belonging to V
- DELETE(G,U,V)
  Delete an edge from the graph G connecting vertices u and V
- ADD-DATA(G,V,ITEM)
  Add ITEM into vertex V of graph G
- RETRIVE (G,V,ITEM)
  Retrieve data of a vertex V in the graph G and store it in ITEM
- BREADTH-FIRST(G)
  Implements breadth first traversal of a graph G
- DEPTH-FIRST(G).
  Implement depth first traversal of a graph G

SUMMARY

- A graph $G=(V,E)$ consists of a set of vertices G and a set of edges E.
- A graph in which every edge is directed is called a directed graph or simply a digraph.
- A graph in which every edge has no direction (undirected) is called an undirected graph.
- Two graphs G and G_1 are said to be isomorphic, if there is a one-to-one correspondence between their vertices and between their edges.
- A circuit is called an Eulerian circuit; if every edge of Graph G appears exactly once, it is called an Eulerian circuit.
- Hamiltonian path in a connected graph is defined as a path that traverses each vertex of the graph exactly once.
- Hamiltonian circuit is a circuit that passes through all the vertices in a graph exactly once except the starting vertex.

- Graphs can be represented using adjacency matrix representation and incidence matrix representation.
- The primitive operations of a graph are Create a Graph, Insertion of a Vertex, Deletion of a Vertex, Insertion of an Edge, Deletion of an Edge, Search for a Vertex, Traversal of a Graph and Deletion of or destroy a graph.
- Insert vertex operation is used to add a vertex to a graph
- Delete vertex operation is used to remove a vertex from a graph.
- Insert edge operation is used to add an edge that connects a vertex to a destination vertex.
- Delete edge operation is used to remove an edge from a graph. To delete an edge the two vertices source and destination must be identified.
- Graphs can be traversed in two ways. They are Breadth first traversal and Depth first traversal.
- The applications of graphs are minimum cost spanning tree, single-source shortest path, union-find problem and connectivity of components.

EXERCISES

Fill in the Blanks

1. A graph in which every edge is directed is called _____ .
2. _____ is a path that traverses each vertex of the graph exactly once.
3. Graphs can be represented sequentially using _____ and _____ .
4. Linked representation is also called _____.
5. _____ and _____ are the traversal of graphs.

Multiple-choice Questions

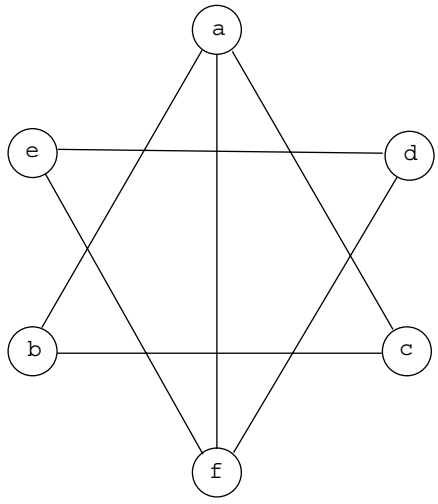
1. _____ passes through all the vertices in a graph exactly once except the starting vertex.
   a. Eulerian circuit b. Hamiltonian circuit
   c. Cutset d. None
2. A graph with only isolated vertices is called _____ .
   a. Null graph b. Weighted graph
   c. Finite graph d. Infinite graph
3. A graph is said to be _____ if there is a path between every pair of distinct vertices v_i, v_j in G.
   a. Directed b. Undirected
   c. Connected d. None
4. A path of length ≥1 with no repeated edges and whose end vertices are the same is called _____ .
   a. Cycle b. Length of path
   c. Circuit d. Degree
5. If every edge of graph G appears exactly once in the path, it is called_____ .
   a. Hamiltonian path b. Eulerian path
   c. Planarity d. Cutset

Short-answer Questions

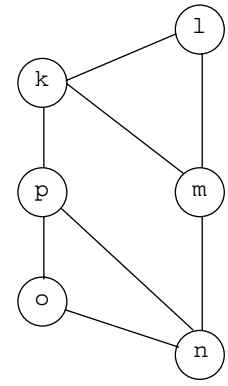
1. Define a graph.
2. Give the basic terminologies of a graph.
3. Discuss different types of graphs.
4. Explain the linked representation of graphs.
5. Write short notes on isomorphism.

Essay Questions

1. Define graphs and explain their representation.
2. Find whether the following graphs G_1 and G_2 are isomorphic?



G_1



G_2

3. Discuss in detail about the operations of graphs.
4. Explain the traversal of graphs.
5. Write a program to implement the Prim algorithm.

Chapter 11

Priority Queues

The chapter defines priority queues and its abstract data type. It discusses about various implementations of priority queues in detail regarding heaps, insertions and deletions of min and max heaps. This chapter details external sorting using multiway merge and polyphase merge.

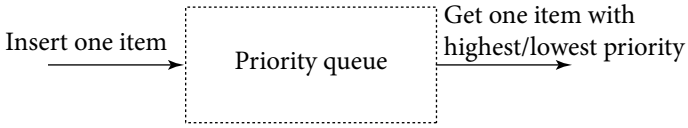
11.1 INTRODUCTION

Queues are FIFO structures in which the elements are deleted in the order in which they arrive in the queue. The order of insertion and deletion from a priority queue is determined by the element priority. The elements are removed in the decreasing or increasing order of priority of elements. A priority queue which is a data structure allows at least two operations: `insert` and `delete`. The operation `insert` inserts an element into the priority queue and the operation `delete` deletes an element from the priority queue based on priority. It is possible to add other operations to this data structure.

Basically, priority queues are of two types. They are

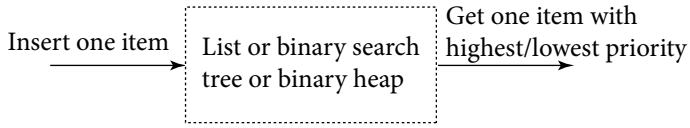
1. Ascending priority queue (Min Priority Queue)
   - Removal of minimum-priority element
2. Descending priority queue (Max Priority Queue)
   - Removal of maximum-priority element

The specification view of priority queue is given in Figure 11.1(a). This view elides the implementation strategy of a priority queue.



(a) Specification view

Figure 11.1 Abstract model of a priority queue



(b) Implementation view

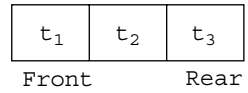
Figure 11.1 *Continued*

The implementation view of a priority queue is given in Figure 11.1(b). In this view the implementation strategy of a priority queue is given.

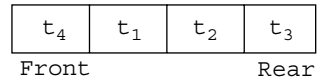
Assume *Pqueue* is a priority queue with elements *A, B, C, D* whose priorities are given as 3, 2, 1, 1, respectively. Here the element A has the highest priority, i.e. 3. When the new element with the highest priority as 4 is to be inserted, it is added at the front of the queue. When elements with the same priority are to be inserted then the priority queue acts like a normal queue with FIFO principle.

Let *task* be the queue of tasks (`t`). Assume that 0, 1, 2… `n` are the priorities given to `t` tasks. Let $t_1(n)$ be the i^{th} task with priority `n`. The implementation of priority queue to process these tasks is discussed for a set of tasks, insertion and task completion which is as follows:

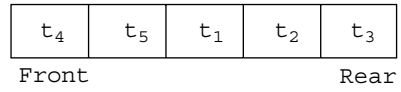
- Initially *task* is with t_1, t_2, t_3 tasks with priorities 1,1,0, respectively.



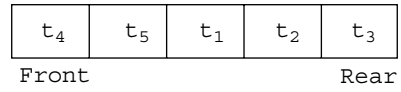
  Highest priority task $t_1(1)$ is at the front, and the lowest priority $t_3(0)$ is at the rear.
- The task $t_4(2)$ with priority 2 comes, it is inserted at front



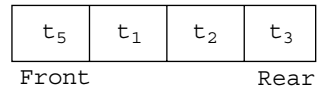
- When task $t_5(2)$ is to be inserted, t_4 and t_5 both have the same priority based on the FIFO property t_5 that is inserted.



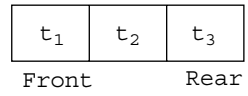
The elements are removed in the decreasing or increasing order of priority of elements.



When the decreasing order of elements is considered, task t_4 with highest priority is deleted first.



Task t_5 is deleted and then t_1, t_2, t_3 are deleted, respectively, based on the priority.



Priority queue can be implemented using:

1. *Unsorted list*: Store all elements in an unordered list. The element is added at the end of the list. An element with min/max priority is removed by searching the list. The problem with this approach is that it is expensive to search. The time required for inserting an element is O(1), whereas the time required to delete an element with minimum/maximum priority is O(N) where N is the number of elements.
2. *Sorted list*: Keep the list always sorted. The cost for inserting an element is O(N) and for deleting an element with min/max priority is O(1).
3. *Binary search tree*: A third alternative is the binary search tree. This gives an O(log N) average running time for insertion and deletion operations. The problem is that the tree could be unbalanced on non-random input.
4. *Heap*: A heap is a complete binary tree, where the element at each node is greater than or less than or equal to the element in its children. To add an element to a heap, place the new element at the next available spot and perform a reheapification. To remove the element, move the last node onto the root and perform a reheapification. The time for insertion could be as much as O(log N) and for deletion an average running time is O(log N), where N is the number of elements (nodes) in the tree.

11.2 PRIORITY QUEUE ADT

The abstract data type for a min priority queue is given in Figure 11.2.

```
    empty()    : return true iff the queue is empty
    full()     : return true iff the queue is full
    minEleRet(): returns element with minimum priority, but element is not
                 deleted
 insertEle()   : insert an element into the queue
deleteMinEle(): remove/delete the element with min priority from the queue
```

Figure 11.2 Abstract data type specification of a min priority queue with basic operations

The abstract data type for a max priority queue is given in Figure 11.3.

```
    empty()       : return true iff the queue is empty
    full()        : return true iff the queue is full
    maxEleRet()   : returns element with maximum priority,but element is
                    not deleted
    insertEle()   : insert an element into the queue
    deleteMaxEle(): remove/delete the element with max priority from
                    the queue
```

Figure 11.3 Abstract data type specification of a max priority queue with basic operations

11.3 PRIORITY QUEUE IMPLEMENTATION USING HEAPS

A priority queue is effectively implemented using heap data structure, which is a complete binary tree that is most efficiently stored by using the array representation. A heap is a binary tree that is completely filled with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a complete binary tree. A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. This implies that the height

of a complete binary tree is O(log N), where N is the number of elements. A complete binary tree is shown in Figure 11.4, whereas the tree in Figure 11.5 is not a complete binary tree.

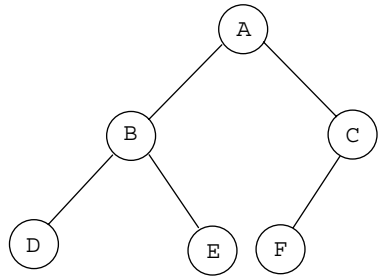


Figure 11.4 A complete binary tree

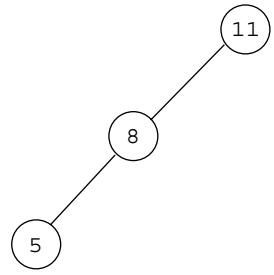


Figure 11.5 A binary tree which is not complete

The tree in Figure 11.5 doesn't have a right sub-tree and the internal node in the left sub-tree is also not completely filled. Hence, it is not a complete binary tree.

Max heap: A max heap is a complete binary tree in which the value in each node is greater than or equal to those in its children (if any). The tree in Figure 11.6 is a max heap.

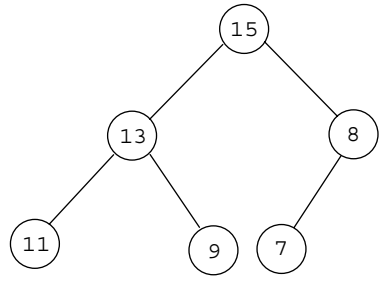


Figure 11.6 A max heap

Min heap: A min heap is a complete binary tree in which the value in each node is less than or equal to those in its children (if any). The tree in Figure 11.7 is a min heap.

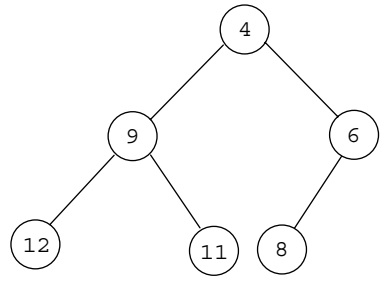


Figure 11.7 A min heap

Since a heap is a complete binary tree, a heap can be efficiently represented using a one-dimensional array. The array in Figure 11.9 corresponds to the min heap in Figure 11.8.

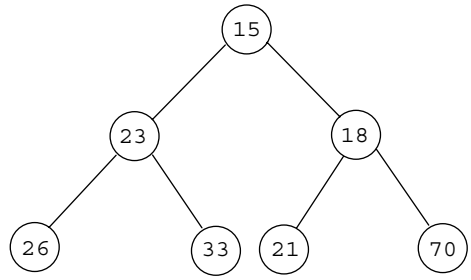


Figure 11.8 A min heap

	15	23	18	26	33	21	70		
0	1	2	3	4	5	6	7	8	9

Figure 11.9 Array implementation of a complete binary tree given in Figure 11.8

For any element in array position i, the left child is in position 2i, the right child is in the cell after the left child (2i+1), and the parent is in position $\lfloor i/2 \rfloor$. Thus, using array links are not required and operations required to traverse the tree are extremely simple and are likely to be fast.

11.3.1 Priority Queue Interface

It is assumed that when two elements of type T are compared using relational operators such as < and <= the element priorities are compared. Unlike ADT specification for priority queue, the interface specification in Figure 11.10(a) is programming language dependent (C++).

```
Template<class T>
class MinHeap
{
   public:
      explicit MinHeap(int capacity=150);
      explicit MinHeap(const vector<T>&items);

      bool empty() const;
      bool full() const;
      void insertEle(const T&x);
      void deleteMinEle(T&minEle);

  private:
      int cSize;            //Number of elements in heap
      vector<T>array;      //The heap array

    void buildMinHeap();
       void moveDownHeap(int newnode);
};
```

(a) Class interface for min priority queue

```
Template<class T>
class MaxHeap
{
   public:
      explicit MaxHeap(int capacity=150);
      explicit MaxHeap(const vector<T>&items);

      bool empty() const;
      bool full() const;
      void insertEle(const T&x);
      void deleteMaxEle(T&maxEle);
  private:
      int   cSize;             //Number of elements in heap
      vector<T>array;          //The heap array

      void buildMaxHeap();
      void moveDownHeap(int newnode);
};
```

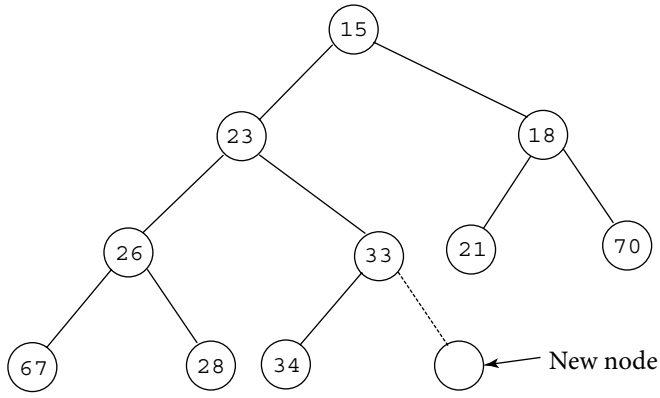
(b) Class interface for max priority queue

Figure 11.10 Class interface

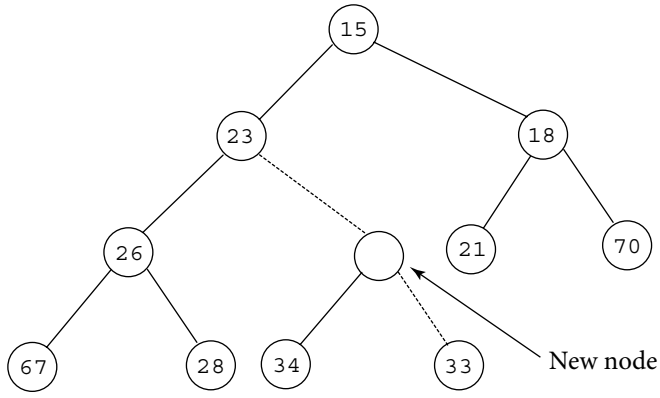
11.3.2 Min Heap–Insertion

To insert an element x into the min heap, a new node is created in the next available location to keep the tree complete (preserving the property of the complete binary tree). Bubble up the new node to maintain the min heap property. To insert the element 16, a new node is created in the next available heap location

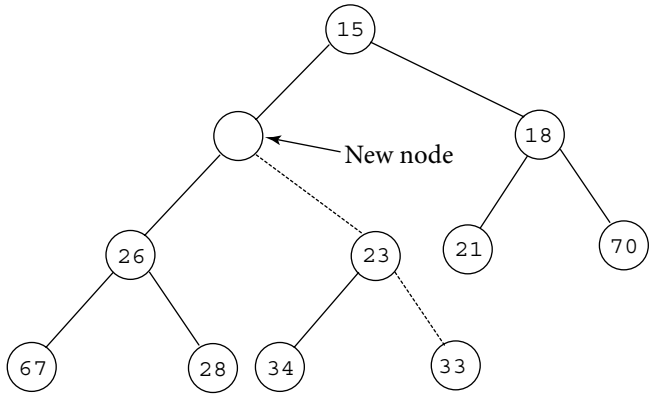
(Figure 11.11(a)). Inserting 16 in the new node would violate the min heap property. Since 33 parents of the new node is greater than 16 (the element to be inserted), slide 33 into the new node (that means the new node moved up: Figure 11.11(b)).



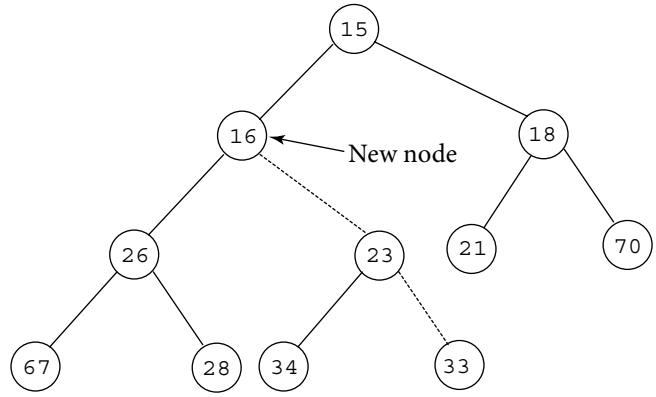
(a) Finding the location to insert 16, new node is created



(b) Finding the location to insert 16, new node is moved one level up



(c) Finding the location to insert 16, new node is moved one more level up, it is the location for 16



(d) Element 16 inserted in the new node

Figure 11.11 Insertion of an element

At this stage, placing 16 in the new node does not achieve min heap property. Hence, a new node needs to be moved up further. Since 23 parents of the new node is greater than 16 (the element to be inserted), slide 23 into the new node and the new node is moved up which is shown by the dashed line, (Figure 11.11(b)).

Now placing element 16 in the new node, shown by the dashed circle (Figure 11.11(b)), preserves the min heap property. A min heap is a complete binary tree in which the value in each node is less than or equal to those in its children (if any). Since the parent of 16 is less than or equal to 16; hence, it has reached its proper location. Element 16 in this location preserves the min heap property. Hence, element 16 is inserted in this location (Figure 11.11(d)). The C++ code for method `insertEle` is given in Figure 11.12(a).

Insert element x into min priority queue maintaining heap order.

```
Template<class T>
void MinHeap<T>::insertEle(const T&x)
{
   //         if(full())
   //       throw Overflow();
   if(cSize==array.size()-1)
   array.resize(array.size()*2);    //if the array is full,it is resized

   // Move up
   int newnode=++cSize;
   for(;newnode>1&&x<array[newnode/2];newnode/=2)
      array[newnode]=array[newnode/2];
   array[newnode]=x;                    //element x inserted
}
```

(a) Method to insert an element into min heap

```
Template<class T>
void MinHeap<T>::insertEle(const T&x)
{
      if(full())
      throw Overflow();

      //Move up
   int newnode=++cSize;
   int temp;
   array[newnode]=x;
   for(;newnode>1&&array[newnode]<array[newnode/2];newnode/=2)
      {
         temp=array[newnode];
         array[newnode]=array[newnode/2];
         array[newnode/2]=temp;
      }
      //Ultimately element x is reached to array index of array[newnode].
}
```

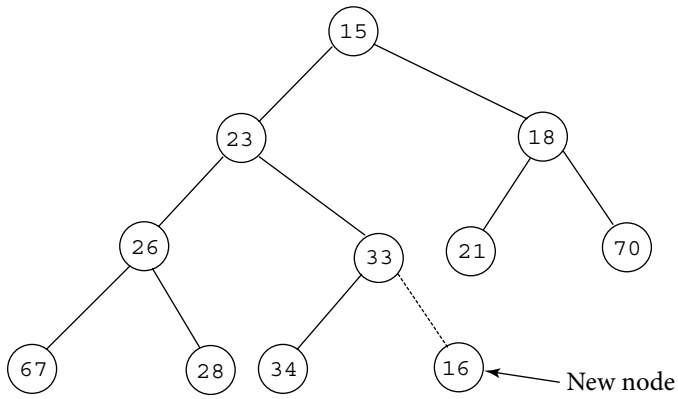
(b) Method to insert an element into min heap using swapping

Figure 11.12 Method to insert an element

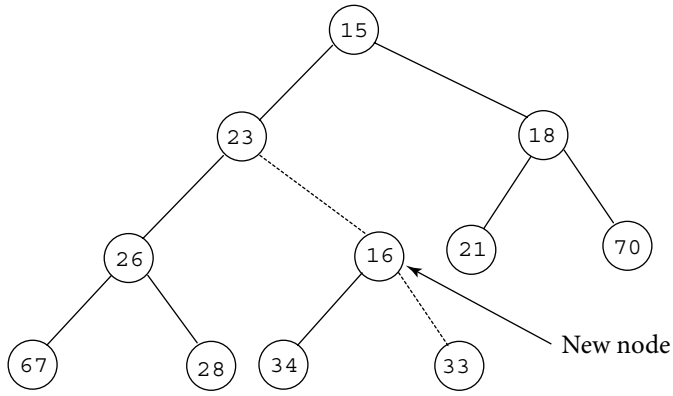
Inserting the element could have been implemented using swap operation. Initially the element should be inserted in the new node, and swapping should be done between the element and its parent until the min property is achieved, but a swap requires three assignment statements. If an element is moved up m levels, the number of assignments performed by the swaps would be 3m. The above method (without swaps) uses m +1 assignments. The modified method which uses swapping is given in Figure 11.12(b).

The modified function which uses swapping: Insert element x into the priority queue, maintaining heap order.

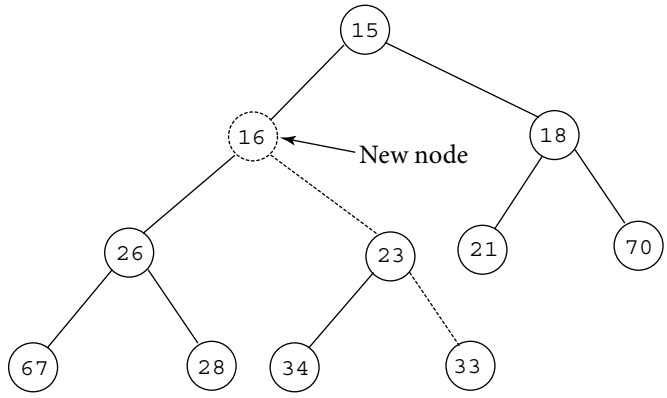
Insertion into min heap using swapping: Moving the element to be inserted into its appropriate position uses swapping. Initially, the element to be inserted is placed by creating a new node in the tree such that the tree is a complete binary tree. After insertion if the tree is min heap then the element is not moved up. Otherwise, it is moved up by swapping with its parent until the min heap property is achieved.



(a) Finding the location to insert 16, temporarily insert 16 into the new node



(b) Element 16 is moved one level up by swapping 33 with 16



(c) Element 16 is moved one level up by swapping 23 with 16

Figure 11.13 Insertion using swapping

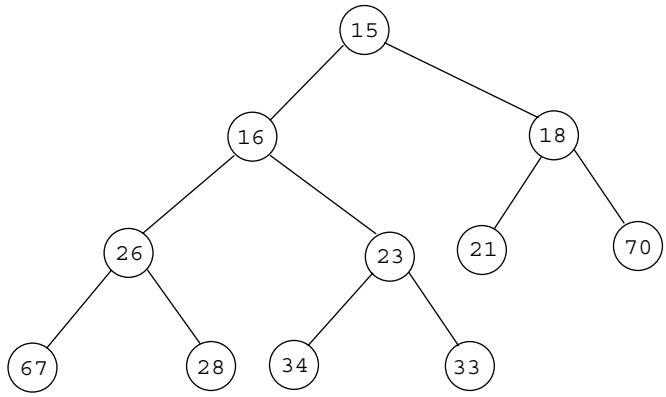
To insert element 16, a new node is created in the next available heap location (Figure 11.13(a)). Insert 16 in the new node. Element 16 in this location would violate min heap property. Hence, move 16 up by swapping with its parent. Since 33 parents of the new node is greater than 16, swap 33 and 16; 16 is moved up and is indicated by a dashed line: Figure 11.13(b).

At this stage (Figure 11.13(b)), heap does not have min heap property. Hence, 16 needs to be moved up further. Since 23 which is a parent of 16 is greater, swap 23 with 16 (Figure 11.13(c)) which is shown by a dashed line.

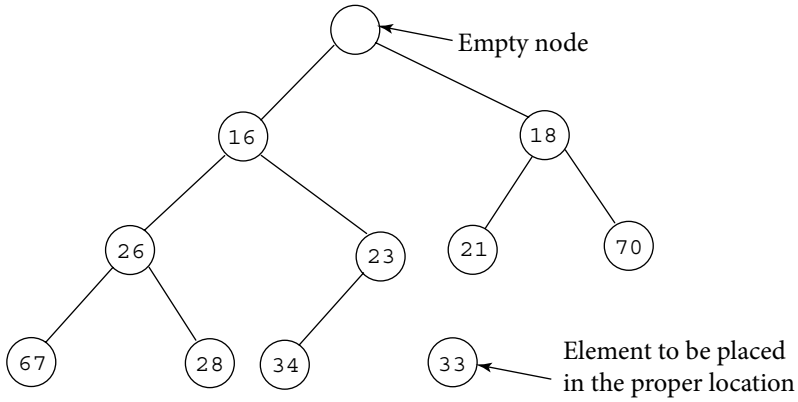
In Figure 11.13(c), further swapping is not required since min heap property is achieved. The time for inserting an element into priority queue could be as much as O(log N) if the element is moved all the way to the root.

11.3.3 Min Heap–Deletion

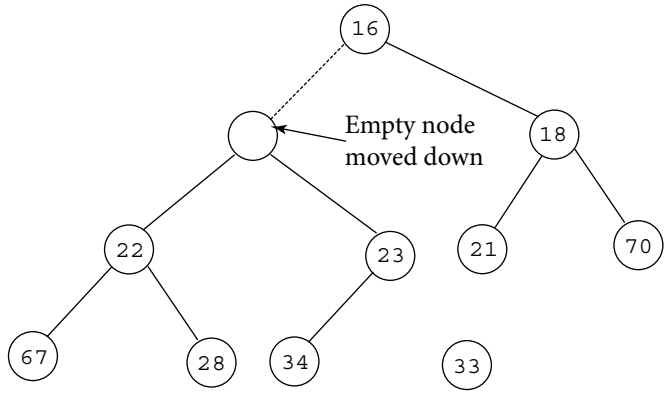
When the min element is removed an empty node is created at the root. Since the heap tree is now with one node less than the earlier number of nodes, the last element in the heap must move somewhere in the heap.



(a) Delete minimum element 15

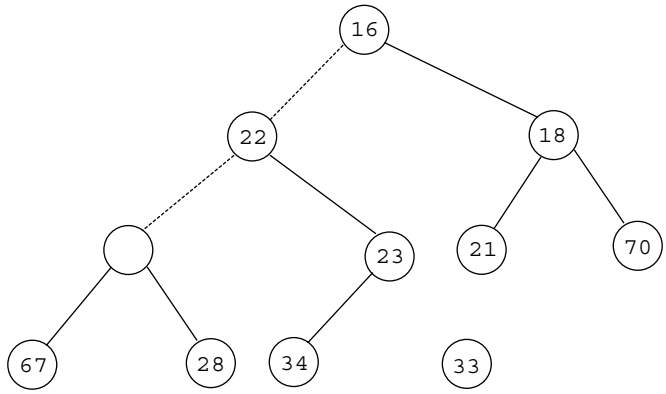


(b) Deletetion of minimum element 15 created an empty node

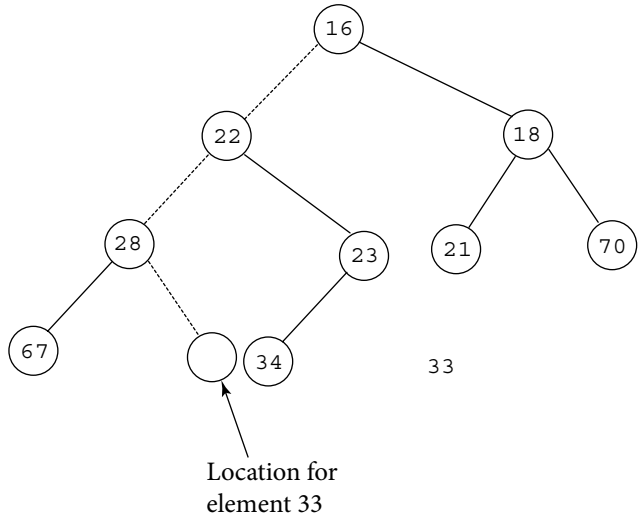


(c) Empty node moved one level down

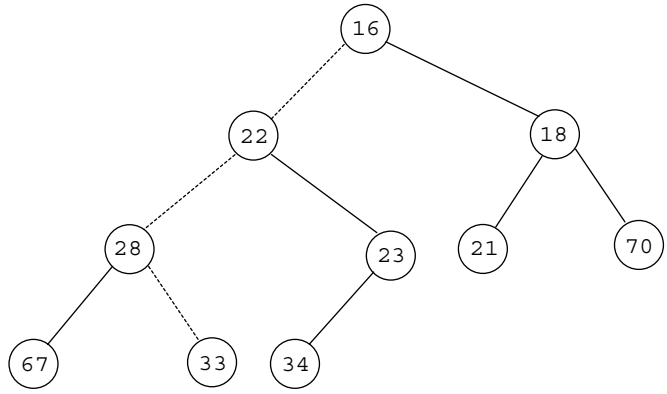
Figure 11.14 Deletion of element



(d) Empty node moved one more level down



(e) Location found to insert the last element 33



(f) Last element 33 is placed in its location

Figure 11.14 *Continued*

Deletion of minimum element created an empty node as shown in Figure 11.14(b). The last element in the tree which is 33 should be moved somewhere into the tree. Moving 33 into a proper location in the tree means ensuring min heap property after placing element 33 in its proper place.

Assume that the last element 33 is placed in an empty node. Now compare 33 with its children 16 and 18 (Figure 11.14(b)). If both the children are greater than or equal to its parent, then moving down stops. Otherwise, move down the empty node towards the smaller child. In this case, it is moved towards the left. Hence 16 is moved up by one level (Figure 11.14(c)).

Assume that element 33 is placed in the empty node (Figure 11.14(c)). In this position it does not maintain the min heap property. Now compare 33 with its children 22 and 23 (Figure 11.14(c)). The empty node is moved towards the left which is represented by a dashed line (Figure 11.14(d)).

Assume that element 33 is placed in an empty node (Figure 11.14(d)). In this position it does not maintain the min heap property. Now compare 33 with its children 67 and 28 (Figure 11.14(d)). The empty node is moved towards the right as shown with the dashed line (Figure 11.14(e)).

Assume that the element 33 is placed in an empty node (Figure 11.14(e)). In this position it does maintain the min heap property. Hence, place the element in this empty node (Figure 11.14(f)).

C++ implementation of the method to delete an element with minimum priority from the heap is given in Figure 11.15.

Remove the smallest element from the priority queue and reheap.

```
Template<class T>
Void MinHeap<T>::deleteMinEle(T&minEle)
{
   if(empty())
      throw Underflow();
   minEle=array[1];
   array[1]=array[cSize--];
   moveDownHeap(1);
}

// Method to move down the heap
// emptynode is the index at which moving down begins

Template<class T>
void MinHeap<T>::moveDownHeap(int emptynode)
{
   int child;
   T temp=array[emptynode];
for(;emptynode*2<= Size;emptynode=child )
   {
   child=emptynode*2;
   if(child!=cSize&&array[child+1]<array[child])
      child++;
   if(array[child]<temp)
      array[emptynode]=array[child];
   else
```

Figure 11.15 Method to delete an element with minimum priority and reheaping

```
        break;
    }
aray[emptynode]=temp;
}
```

Figure 11.15 *Continued*

Given N items by successive insertions, a min heap can be built. A method `buildMinHeap()` is defined which builds a heap of N elements; this method is given in Figure 11.16.

```
explicit MinHeap(const vector<T>&items)
    :array(items.size()+11), cSize(items.size())
    {
        for(int i=0;i<items.size();i++)
            array[i+1]=items[i];
        buildMinHeap();
    }
Template<class T>
void MinHeap<T>::buildMinHeap()
 {
     for(int i=cSize/2;i>0;i--)
        moveDownHeap(i);
}
```

Figure 11.16 Method to build min heap from N elements

Program 11.1

```
#include<iostream.h>
#include<process.h>
#include<conio.h>
int n,front=0,rear=0,i=0;
class minheap
{
   int a[50],b[50],item,p;
   public:
   void minheapinsert();
   void minheapdeletel();
   void minheapdisplay();
};
void minheap::minheapinsert()
{
   if(front==((rear%n)+1))
   {
      cout<<"queue is overflow";
      return;
   }
   cout<<"enter element&prioriry";
```

```
      cin>>item>>p;
   if(front==0)
      front=rear=1;
   else
      rear=(rear%n)+1;
   a[rear]=item;
   b[++i]=p;
   for(int j=i-1;j>=1;j--)
   {
      for(int k=1;k<=j;k++)
      {
         if(b[k]>b[k+1])
         {
            b[k]=(b[k]+b[k+1])-(b[k+1]=b[k]);
            a[k]=(a[k]+a[k+1])-(a[k+1]=a[k]);
         }
      }
   }
   cout<<"\n:element is  inserted successfully:";
}
void minheap::minheapdeletel()
{
   if(front==0)
   {
      cout<<"queue is empty";
      return;
   }
else
   {
      cout<<endl<<a[front]<<"successfully deleted";
      if(front==rear)
         front=rear=0;
      else
         front=(front%n)+1;
   }
}
void minheap::minheapdisplay()
{
   int j;
   if(front==0)
   {
      cout<<endl<<"queue is underflow";
      return;
   }
cout<<endl<<"elements";
if(front<=rear)
{
   for(j=front;j<=rear;j++)
```

```
    cout<<a[j]<<" ";
}
else
{
    for(j=front;j<=n;j++)
        cout<<a[j]<<" ";
    for(j=1;j<=rear;j++)
        cout<<a[j]<<" ";
    }
}
void main()
{
    int op;
    clrscr();
    minheap ob;
        cout<<"enter queue size";
        cin>>n;
    while(1)
    {
        cout<<"\nmenu\n1.insert\n2.delete\n3.display\n4.exit\n";
        cout<<"enter option";
        cin>>op;
    switch(op)
   {
        case 1:
            ob.minheapinsert();
            break;
        case 2:
            ob.minheapdeletel();
            break;
        case 3:
            ob.minheapdisplay();break;
        case 4:
            exit(0);break;
        default:
        cout<<"you are entered invalid choice";
        }
    }
  }
```

Output

```
enter queue size5
menu
1.insert
2.delete
3.display
4.exit
```

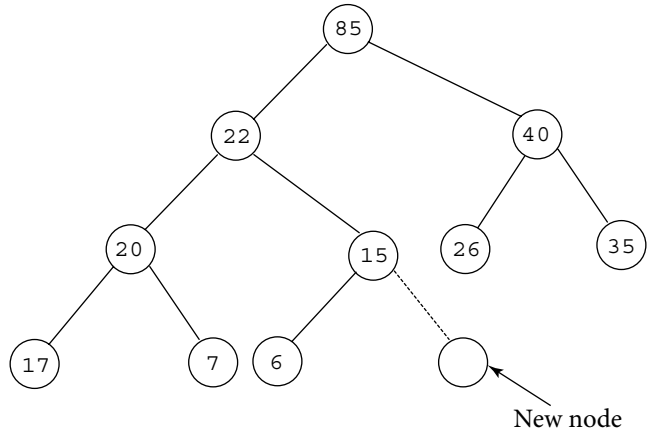
```
enter option1
enter element & prioriry12
5
:element is  inserted succesfully:
menu
1.insert
2.delete
3.display
4.exit
enter option1
enter element & prioriry11
8
:element is  inserted succesfully:
menu
1.insert
2.delete
3.display
4.exit
enter option1
enter element & prioriry30
2
:element is  inserted succesfully:
menu
1.insert
2.delete
3.display
4.exit
enter option3
elements30 12 10
menu
1.insert
2.delete
3.display
4.exit
enter option2
30 successfully deleted
menu
1.insert
2.delete
3.display
4.exit
enter option3
elements12 10
menu
1.insert
2.delete
3.display
4.exit
enter option4
```

In Program 11.1 the function `minheapinsert()` is to insert an element based on the priority into the min heap, and `minheapdelete()` is to delete an element with highest priority from the min heap, and `minheapdisplay()` is used to display the heap tree.

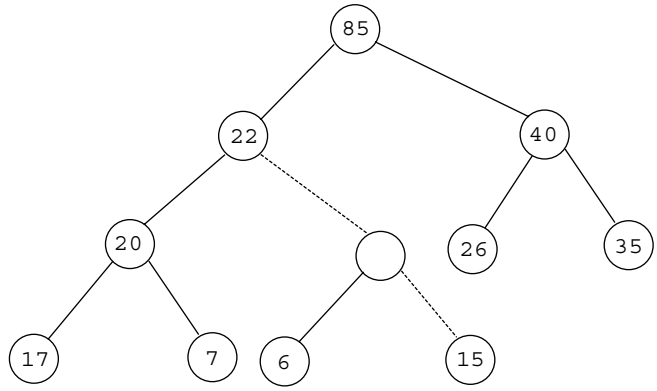
11.3.4 Max Heap–Insertion

To insert an element x into the max heap, a new node is created in the next available location to keep the tree complete (preserving the property of complete binary tree). Bubble up the new node to maintain the max heap property.

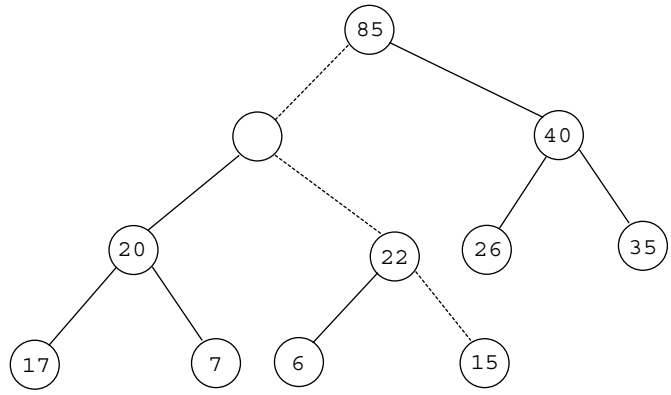
To insert element 90, a new node is created in the next available heap location (Figure 11.17(a)). Inserting 90 in the new node would violate the max heap property. Since 15 parent of the new node is smaller than 90 (the element to be inserted), slide 15 into the new node (that means the new node moved up, indicated by a dashed line: Figure 11.17(b)).



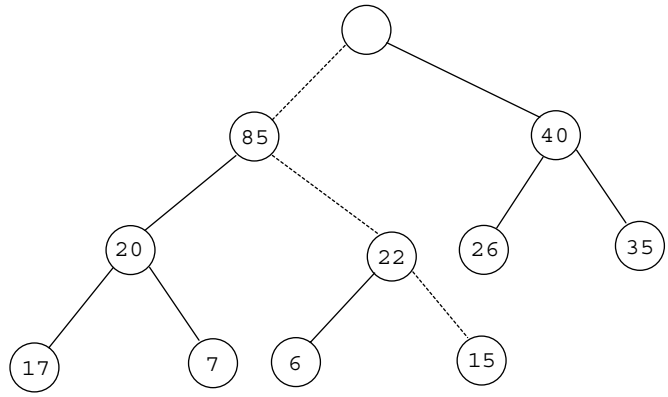
(a) Finding the location to insert 90, showing the created new node



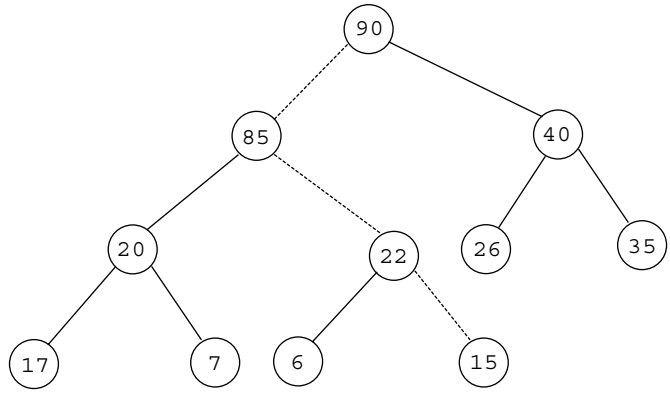
(b) New node moved one level up



(c) New node moved one more level



(d) Location found to insert element 90



(e) Element 90 inserted

Figure 11.17 Insertion of an element into max heap

At this stage, placing the element 90 in the new node does not satisfy max heap property. Hence, the new node needs to be moved up further. Here, 22 parents of the new node is smaller than 90 (the element to be inserted), slide 22 into the new node and the new node is moved up, which are shown by dashed lines (Figure 11.17(c)).

Now placing element 90 in the new node (Figure 11.17(c)) does not preserve the max heap property. Here, 85 parents of the new node are smaller than 90 (the element to be inserted), slide 85 into the new node and the new node is moved up as shown by the dashed line (Figure 11.17(d)).

Now placing element 90 in the new node (Figure 11.17(d)), preserves the max heap property. Since there are no nodes to be tested, element 90 in this location preserves the max heap property and it is the root node (Figure 11.17(e)).

C++ implementation of the method to insert an element into max heap is given in Figure 11.18.

```
Template<class T>
void MaxHeap<T>::insertEle(const T&x)
   {
      if(full())
      throw Overflow();

      //Move up
      int newnode=++cSize;
      for(;newnode>1&&x>array[newnode/2];newnode/=2)
         array[newnode]=array[newnode/2];
      array[newnode]=x;                    //element x inserted
   }
```

Figure 11.18 Method to insert an element into max heap

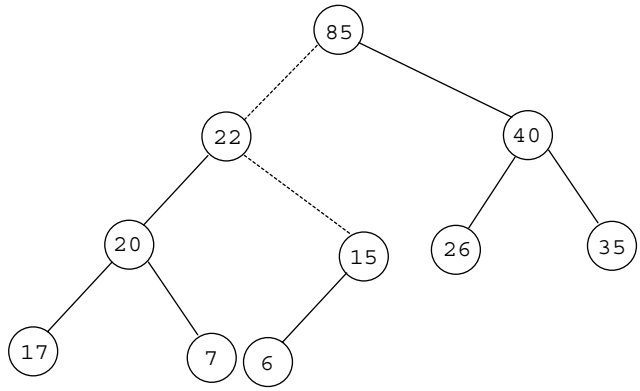
11.3.5 Max Heap–Deletion

When the max element is deleted, an empty node is created at the root. Since the heap tree is now with one node less than the earlier number of nodes, the last element in the heap must move somewhere in the heap.

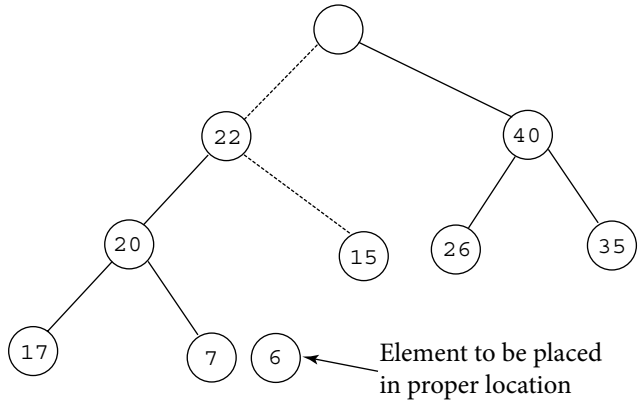
Deletion of max element 85 created an empty node as shown in Figure 11.19(b). The last element in the tree which is 6 should be moved somewhere into the tree. Moving 6 into proper location in the tree means ensuring max heap property after placing element 6 in its proper place.

Assume that the last element 6 is placed in an empty node. Now compare 6 with its children 22 and 40 (Figure 11.19(c)). If the parent is greater than or equal to both the children, then stop moving downwards. Otherwise, move down the empty node towards the larger child. In this case, it is moved towards the right. Hence, 40 is moved up by one level (Figure 11.19(c)).

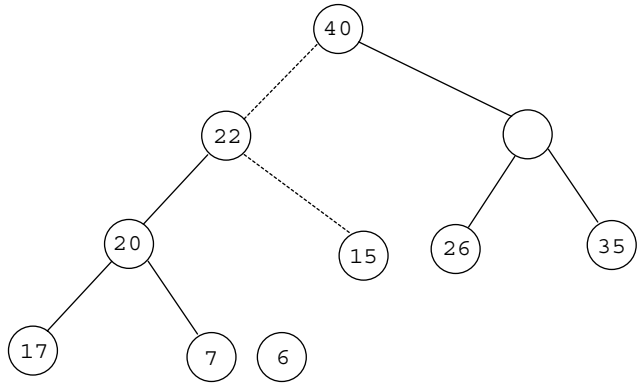
Assume that element 6 is placed in an empty node (Figure 11.19(c)). In this position it does not maintain the max heap property. Now compare 6 with its children 26 and 35 (Figure 11.19(c)). Move down the empty node towards the larger child. The empty node is moved down towards the right (Figure 11.19(d)). This is the place for element 6 to be inserted. After inserting element 6 at this location, it maintains max heap property. Max heap after inserting element 6 is shown in Figure 11.19(e).



(a) Delete max element 85

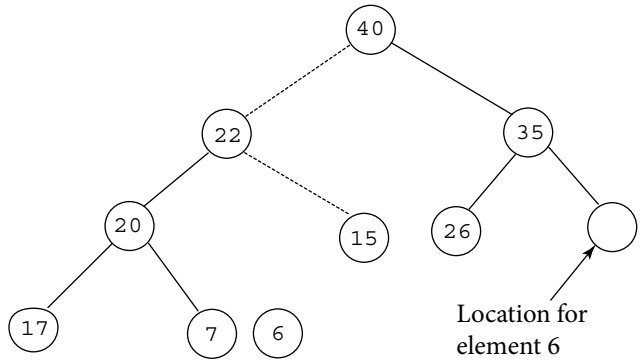


(b) Empty node created after deleting max element 85

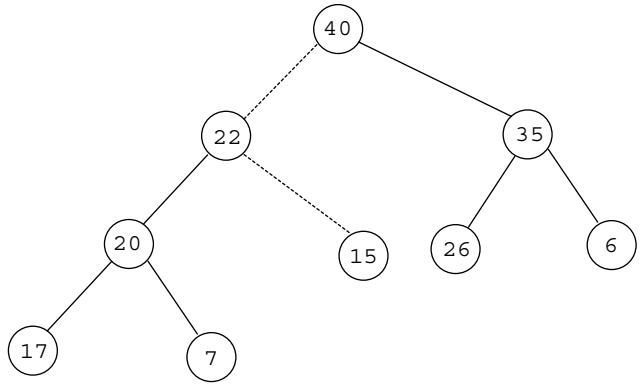


(c) Empty node moved one level down

Figure 11.19 Deletion of the max element



(d) Location found to insert last element 6



(e) Element 6 inserted

Figure 11.19 *Continued*

C++ implementation of method to delete an element from max heap is given in Figure 11.20. Remove the largest element from the priority queue and reheap.

```
Template<class T>
void MaxHeap<T>::deleteMaxEle(T&maxEle)
{
   if(empty())
      throw Underflow();

   maxEle=array{1];
   array[1]=array[cSize--];
   moveDownHeap(1);
}

//Method to move down the heap
//empty node is the index at which moving down begins
```

```
Template<class T>
void MaxHeap<T>::moveDownHeap(int emptynode)
{
   int child;
   T temp=array[emptynode];
for(;emptynode*2<=cSize;emptynode=child)
{
   child=emptynode*2;
   if(child!=cSize&&array[child+1]>array[child])
      child++;
   if(array[child]>temp)
      array[emptynode]=array[child];
   else
      break;
   }
   array[emptynode]=temp;
}
```

Figure 11.20 Method to delete an element with maximum priority and reheaping

Program 11.2

```
#include<iostream.h>
#include<process.h>
#include<conio.h>
int n,front=0,rear=0,i=0;
class maxheap
{
   int a[50],b[50],item,p;
      public:
   void maxheapinsert();
   void maxheapdeletel();
   void maxheaptraverse();
};
   void maxheap::maxheapinsert()
{
   if(front==((rear%n)+1))
   {
      cout<<"queue is overflow";
      return;
   }
   cout<<"enter element&prioriry";
   cin>>item>>p;
   if(front==0)
   front=rear=1;
   else
   rear=(rear%n)+1;
```

```
    a[rear]=item;
    b[++i]=p;
    for(int j=i-1;j>=1;j--)
    {
       for(int k=1;k<=j;k++)
       {
          if(b[k]<b[k+1])
          {
             b[k]=(b[k]+b[k+1])-(b[k+1]=b[k]);
             a[k]=(a[k]+a[k+1])-(a[k+1]=a[k]);
          }
    }
}
    cout<<"\n:element is inserted succesfully:";
}
void maxheap::maxheapdeletel()
    {
       if(front==0)
       {
          cout<<"queue is empty";
          return;
       }
    else
    {
       cout<<endl<<a[front]<<"successfully deleted";
       if(front==rear)
          front=rear=0;
       else
          front=(front%n)+1;
    }
}
void maxheap::maxheaptraverse()
    {
       int j;
       if(front==0)
       {
          cout<<endl<<"queue is underflow";
          return;
       }
    cout<<endl<<"elements";
    if(front<=rear)
    {
       for(j=front;j<=rear;j++)
       cout<<a[j]<<" ";
    }
    else
    {
       for(j=front;j<=n;j++)
```

```
        cout<<a[j]<<" " ;
      for(j=1;j<=rear;j++)
        cout<<a[j]<<" ";
   }
}
void main()
{
   int op;
   clrscr();
   maxheap ob;
   cout<<"enter queue size";
   cin>>n;
 while(1)
 {
     cout<<"\nmenu\n1.insert\n2.delete\n3.traverse\n4.exit\n";
     cout<<"enter option";
     cin>>op;
     switch(op)
     {
        case 1:
          ob.maxheapinsert();
          break;
        case 2:
          ob.maxheapdeletel();
          break;
        case 3:
          ob.maxheaptraverse();break;
        case 4:
        exit(0);break;
        default:
        cout<<"you entered invalid choice";
     }
   }
 }
```

Output

```
enter queue size5
menu
1.insert
2.delete
3.traverse
4.exit
enter option1
enter element & prioriry11
3
:element is  inserted succesfully:
menu
```

```
1.insert
2.delete
3.traverse
4.exit
enter option1
enter element & prioriry21
1
menu
1.insert
2.delete
3.traverse
4.exit
enter option1
enter element & prioriry20
5
:element is  inserted succesfully:
menu
1.insert
2.delete
3.traverse
4.exit
enter option3
elements20  11  21
menu
1.insert
2.delete
3.traverse
4.exit
enter option2
20 successfully deleted
menu
1.insert
2.delete
3.traverse
4.exit
enter option3
elements11  21
menu
1.insert
2.delete
3.traverse
4.exit
enter option4
```

In Program 11.2, the function `maxheapinsert()` is to insert an element based on the priority into the max heap, and `maxheapdelete` is to delete the element with highest priority from the max heap, and `maxheaptraverse()` is used to traverse the heap tree.

In summary, a max/min heap is a complete binary tree, where the element at each node is greater/smaller than or equal to the element in its children. To add an element to a heap, place the new element at the next available spot and perform a reheapification upward. To remove the element, delete the root element and find the proper place for the last element using reheapification downward starting from the root index. The worst case running time for the insertion (which includes reheapification) is O(log N). The worst case and average running time for deletion (which includes reheapification) is O(log N).

11.4 APPLICATIONS

The main applications of priority queue are:

- Job scheduling
- Allocating CPU resources to the most urgent task
- Communications
- Send the most urgent message first
- Event-driven simulation
- Printer queue
- Sorting—heap sort
- Shortest path graph algorithms (Greedy algorithms)

11.4.1 Job Scheduling

Suppose there are n jobs and m machines. Processing these n jobs on m machines and processing the time of each job is represented by t_i. The jobs are processed on machines in such a way that

- Each machine can process only one job at any time
- Each job must be processed by only one machine at any time

Each machine is available at time 1 and the total elapsed time is the total time taken to process all the jobs.

Machines: In Figure 11.21, there are three machines m_1, m_2, m_3 and six jobs j_1, j_2, j_3, j_4, j_5 and j_6 with processing times as (1,8,3,9,5,2). Job j_4 is processed on machine m_1 from 1 to 9 time units, job j_2 is processed on machine m_2 from 1 to 8 and then do job j_6 from 8 to 10 time units. Machine m_3 processes job j_5 from 1 to 5 and then job j_3 from 5 to 9 and job j_1 from 9 to 10 time units. The total time required to process all the jobs is 10 units. In Figure 11.21, the jobs are assigned based on the minimum finish time of the jobs. The task is to assign the jobs based on the minimum finish time of m machines for a set of n jobs. This is done by considering the Highest Processing Time strategy. Using this strategy, jobs are assigned to machines in the descending order of their processing times t_i. When a job is assigned to a machine, it is given to the machine that becomes idle first.

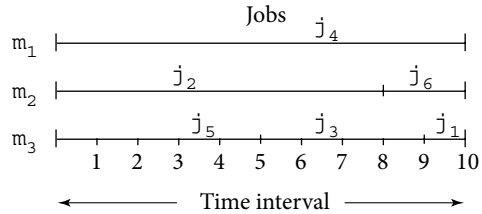


Figure 11.21 Scheduling jobs over machines

For the above, scheduling jobs are sorted in descending order of processing times, i.e. (9, 8, 5, 3, 2, 1). First job j_4 is to be processed. Since all the machines are available initially j_4 can be given to any of the three machines. Assume that j_4 is assigned to machine m_1 so m_1 is not available to other jobs until 9 units of time. Job j_2 can be assigned to any of the two machines that is either m_2 or m_3 as both are still available at time 1. Suppose job j_2 is assigned to machine m_2 then m_2 is not available up to 8 units of time. Job j_5 is processed on machine m_3 from time 1 to 6 units. Now job 3 is to be assigned, the first machine that becomes idle is m_3 at time 6 units and j_3 is assigned to it from 6 to 9 units and the next machine that becomes idle is machine m_2 at time 8 units. In this way jobs are assigned to the machines that become idle first.

11.5 EXTERNAL SORTING

External sorting is required if the data to be sorted are large enough, which cannot be loaded at once in memory, and common sorting algorithms are not applicable. Larger files may be too large to fit in memory simultaneously and require external sorting. External sorting uses secondary devices (magnetic tapes or magnetic disks). The criteria for evaluating external sorting algorithms are different from internal sorts. Internal sort comparison criteria are: number of comparisons required, number of swaps made and memory requirement, whereas external sort comparison criteria are dominated by I/O time (transfers between secondary storage and main memory). External sorting which uses magnetic tapes is known as tape sorting.

Runlists: External sorting is the process of sorting small groups of records from a file in memory. These groups are called runlists. The size of these runlists depends upon the internal memory used for internal sorting. These runlists are stored in a target file from which they are retrieved and merged together to form large runlists. This process is continued until a single runlist is generated, which is the required sorted file.

Assume that there is a buffer enough to hold p records and a file with q records; always p is less than q. Sorting a file involves generation of runlists which are used to read q records into an internal buffer, sort them using internal sorting and write them onto a target file of size q. If the value of q is smaller than 15 then selection sort can be used and for larger values of q heap sort can be used. This process of generating runlists is continued until all q records are performed.

11.5.1 Polyphase Merge

Magnetic tape is a sequential storage medium used for data collection and back up. The main disadvantage of tape sorting is that huge amounts of tape rewinding takes place. Polyphase sort is one of the best method for performing external sorting on tapes.

A process of distributing ordered runlists of predetermined size on to the tapes and continuously merging these runlists in multiple phases is the polyphase merge. Each phase has a fixed number of merges before a new tape is selected. The initial distribution of runlists on the working of tapes affects the performance of sorting. However, it is found that the Fibonacci distribution of initial runlists produces better performance. Polyphase merge is also known Polyphase sort.

Consider sorting of 9 records using 4 tapes. Originally all the records are on tape 4 as shown in Figure 11.22(a). Record is represented by its key. The n^{th} order Fibonacci series is used to determine the number of runs on each tape where n=T–1, where T is the number of tapes. This series is defined as follows:

$$F^n_s = F^n_{s-1} + F^n_{s-2} + \ldots + F^n_{s-n}$$

$$F^n_s = 0 \text{ for } 0 \le s \le n-2$$

$F^n_{n-1}=1$

for the s-level perfect distribution where s is the size of the runlist. In general, the k^{th} tape should be allotted $F^n_{s-1}+F^n_{s-2}+\ldots+ F^n_{s-k}$ initial runs in the s-level perfect distribution.

Therefore, in the example with 9 records, when n=3, tape 1 receives 2 runs, tape 2 receives 3 runs and tape 3 receives 4 runs. Based on the recurrence formulae, the initial runs are distributed on the tapes (Figure 11.22(b)).

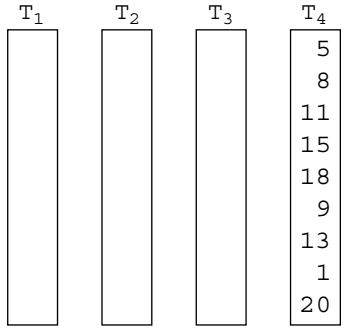
Algorithm 11.1

Step 1: A loop designed to distribute the runs on the tapes. The distribution is a Fibonacci distribution.

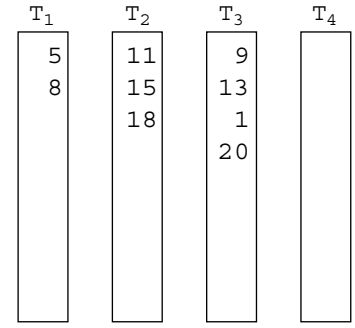
Step 2: Rewound tapes to allow them to be read.

Step 3: Merge and sort: During merging runs from the source tapes are compared and sorted and written on object (output tape). When the source tape becomes empty it takes the role of an object tape. The earlier object tape becomes the source tape.

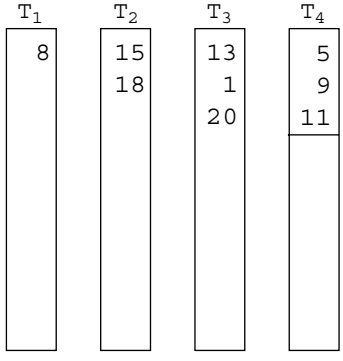
To keep the example simple all initial runs are assumed to be of length 1 (one record in each run). In practice these runs should be made as large as possible. Nine initial runs are available on tape 4. The initial run distribution on tapes 1, 2 and 3 is a Fibonacci distribution (Figure 11.22(b)).



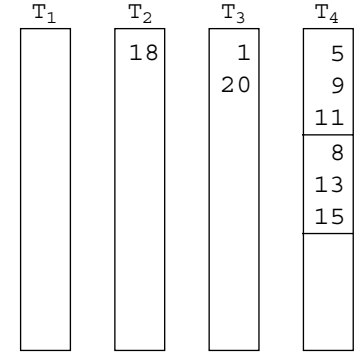
(a) Initial runs on tape 4



(b) Fibonacci distribution of runs on tapes

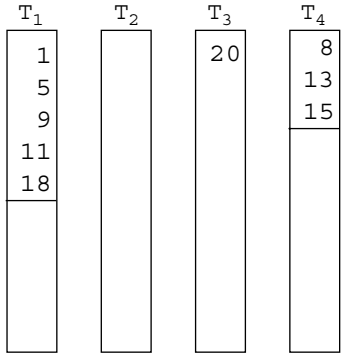


(c) After merge 1, the merged run is placed on tape 4

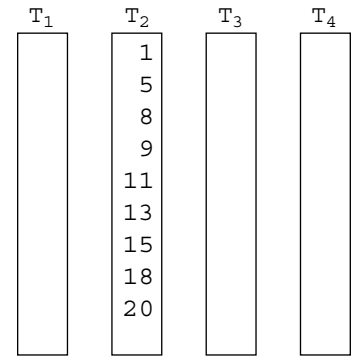


(d) Records on tapes after merge 2

Figure 11.22 Initial run distribution



(e) Records on tapes after merge 3



(f) Sorted records on tape 2

Figure 11.22 *Continued*

Tapes 1, 2 and 3 are source tapes, whereas tape 4 is an object tape (output tape).

On the first merge pass the initial runs on each of the source tapes 1, 2 and 3 are merged (i.e. records 5, 11, 9), and the run <5, 9, 11> is placed on the initially empty tape 4 (Figure 11.22(c)).

The second merge places the run <8,13,15> on the tape 4 (Figure 11.21(d)). This now leaves tape 1 with no more runs left to be read. It is therefore rewound in preparation for its new role as the object tape (output tape) in the next merge phases. Tape 4 is also rewound so that it can play as the source tape (input tape).

The next phase begins by merging runs from tapes 2, 3 and 4 which results in run <1,5,9,11,18> and placed on tape 1.

At this stage, tape 2 is empty. Now this acts as the object tape, whereas tapes 1, 3 and 4 act as source tapes. After merging runs from tapes 1, 3 and 4 result in run <1,5,8,9,11,13,15,18,20> and placed on tape 2.

At this stage, all the other three tapes 1, 3 and 4 are empty. Hence, the run on tape 2 is in sorted form (Figure 11.22(f)).

The number of records to sort may not always agree exactly with a perfect Fibonacci distribution. In this situation null or dummy records need to be assumed on the tapes. Enough dummy records should be included to bring the total number of runs up to the next perfect Fibonacci distribution. It is worth noting that a major portion of time in the sort is used in the rewinding of tapes so that they can be read. In fact, at the completion of every pass two of the tapes must be rewound. This time can be reduced if tapes are readable backwards also.

11.5.2 Multiway Merge

The capability to access a particular record directly on a magnetic disk is very important when considering the sorting of records in external memory. With direct record access the problems of setting up initial runs according to certain merge patterns and having to rewind working tapes can be ignored. Specifically, it is simple to use a k-way merge strategy that allows ignoring these two problems and thereby reducing the sort time significantly. When data do not fit in memory (RAM), external (or secondary) memory is used. Magnetic disks are the most commonly used type of external memory. Because of access to disk drive is much slower than access to RAM, analysis of external memory algorithms usually focuses on the number of disk accesses (I/O operations).

11.5.2.1 2-Way Merge Sort

Algorithm 1.2

Pass 0:

- Read each of the N pages page-by-page
- Sort the records on each page individually
- Write the sorted pages to disk (the sorted page is referred to as a run) (Pass 0 writes $N=2^s$ sorted runs to disk, only one page of buffer space is used)

Pass 1:

- Select and read two runs written in Pass 0,
- Merge their records
- Write the new two-page run to disk (page-by-page) (Pass 1 writes $2^s/2=2^{s-1}$ runs to disk, three pages of buffer space are used)

Pass n:

- Select and read two runs written in Pass n–1,
- Merge their records
- Write the new 2^n-page run to disk (page-by-page)
- Pass n writes 2^{s-2} runs to disk, three pages of buffer space is used)

Pass s:

- Pass s writes a single sorted run (i.e. the complete sorted file) of size $2^s=N$ to disk.

The storage requirements to perform 2-way merge sort are two input buffers and one output buffer (Figure 11.23). More generally, for a k-way merge `k + 1` buffers are required.

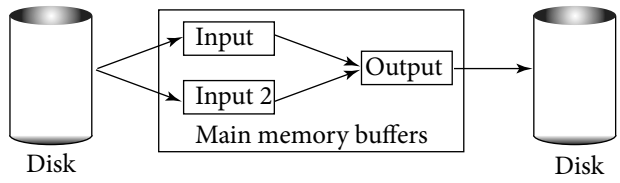


Figure 11.23 Required buffers for 2-way merge sort

During pass 0 only one buffer is used. In pass 0 the pages are sorted using some sorting algorithm. In subsequent passes merging is done and during these passes two buffers for input and one buffer for output are used. A typical tree view that emerges for sorting 8 pages using the 2-way merge sort is shown in Figure 11.24.

A file with 8 pages using 2-way merge sort is given in Figure 11.25. It took 4 passes to sort the file. In pass 0 the initial single pages are sorted. The sorted pages are the output of pass 0. They are known as single page runs. In pass 1 two single page runs are merged. The output of this pass 1 is a 2-page run. Similarly, the output of pass 2 is a 4-page run and the output of pass 3 is an 8-page run. It took total four passes by 2-way merge sort using 3 buffers.

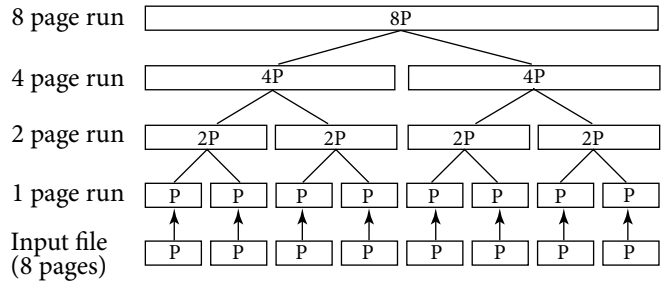


Figure 11.24 A typical 2-way merge sort tree

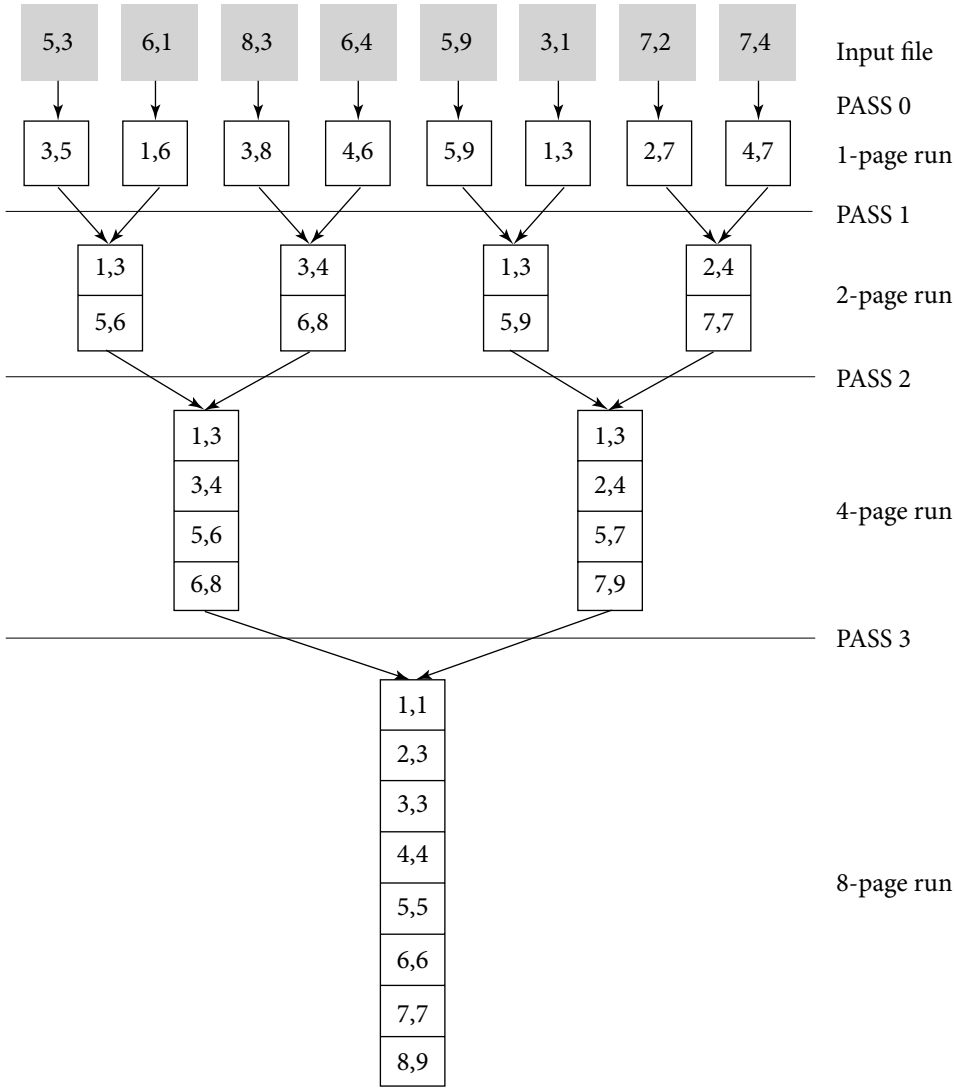


Figure 11.25 Showing runs during merging passes in 2-way merge sort

Assuming r initial runs (pages), it can be shown that the number of passes required to sort a file using a k-way merge is $O(\lceil \log_k r+1 \rceil)$. The number of passes required for a 2-way merge for sorting 8 initial runs is $(\lceil \log 28 \rceil +1) = 4$.

The total cost (cost in terms of I/O) of 2-way merge sort is : $2N(\lceil \log_2 N \rceil +1)$, where N is the number of pages in the file.

If more main memory buffers are available, by using k-way merge sort, the total I/O cost is reduced when compared to a 2-way merge sort (where k>2).

Program 11.3

```
#include<iostream.h>
#include<conio.h>
void merge(int[],int,int);
void mergesort(int[],int,int,int);
void main()
{
   int n,a[50],low,up,i;
   clrscr();
   cout<<"==================";
   cout<<"\n2-Way Merge Sort";
   cout<<"\n==================";
   cout<<"\nHow Many Elements You Want To Sort:";
   cin>>n;
   cout<<"Enter numbers:\n";
   for(i=0;i<n;i++)
      cin>>a[i];
      cout<<"\nBefore Sorting Record Elements Are\n";
   for(i=0;i<n;i++)
      cout<<a[i]<<" ";
      merge(a,0,n-1);
      cout<<"\nAfter Sorting Record Elements Are:\n";
   for(i=0;i<n;i++)
      cout<<a[i]<<" ";
      getch();
}
void merge(int a[],int low,int high)
{
   int mid;
   if(low<high)
   {
      mid=(low+high)/2;
      merge(a,low,mid);
      merge(a,mid+1,high);
      mergesort(a,low,mid,high);
   }
}
void mergesort(int a[],int l,int mid,int h)
{
```

```
    int i,j,k=0,b[50];
    i=l;
    j=mid+1;
    while(i<=mid&&j<=h)
    {
       if(a[i]<a[j])
          b[k++]=a[i++];
       else
          b[k++]=a[j++];
    }
    while(i<=mid)
       b[k++]=a[i++];
    while(j<=h)
       b[k++]=a[j++];
    for(k=0;k<=h-l;k++)
       a[k+l]=b[k];
}
```

Output

```
==================
2-Way Merge Sort
==================
How Many Elements You Want To Sort:6
Enter numbers:
7
3
0
1
4
9
Before Sorting Record Elements Are
7 3 0 1 4 9
After Sorting Record Elements Are:
0 1 3 4 7 9
```

Program 11.3 sorts the given list in ascending order using the 2-way merge sort technique. The method `merge()` divides the list and calls `mergesort()` method which sorts the given list and also makes a call to itself.

11.5.2.2 k-Way merge sort

Algorithm 11.3

To sort a file with N pages using B buffer pages (k+1):

- Pass 0: Produce sorted 1-page run
- Pass 1,2, ..., etc.: merge k runs.

During merging, tournament tree concept is used to find the first winner and second winner and so on… (Uses k main memory buffers for input and one buffer for output).

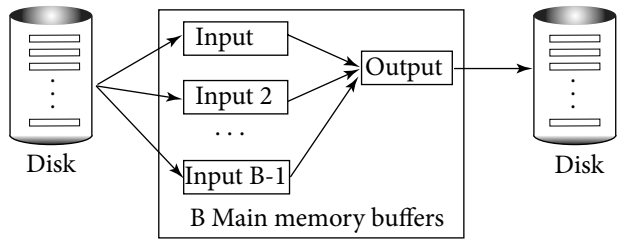


Figure 11.26 Buffers required for k-way merge sort

k-Way merge sort requires k+1 buffers (Figure 11.26). A 2-way merge algorithm involves comparison strategy for two values which is easy to perform. One of the main problems in a general k-way merge is to establish an efficient comparison strategy that involves more than two values.

Consider the records in the k runs that are participating in the merge as players in the tournament. A competition tree can then be formed in which the winner of a particular competition is the record with the smallest key (Figure 11.27).

One typical tournament tree view during the merging of k runs in a k-way merge is shown in Figure 11.27(a). The smallest key (first winner) for the above example is 2 and is placed in the output buffer. Similarly, the next smallest key (second winner) is to be found. In this case, it is 4. It should be noted that the record 2 is not considered for comparison (Figure 11.27(b)).

The third winner would be 5. In this case the buffer size is assumed to be of size 3 records. Once the buffer is full it is written to the disk. Once again the buffer is ready to receive the next 3 records. Since the contents of buffer are written to the disk once it is full, it requires only one buffer for the output where as, k buffers are required for input.

A file consisting of 8 pages is sorted using 8-way merge sort (Figure 11.28). It took two passes to complete the sorting. During pass 1, tournament tree is used for merging. The 8-page run sorted output is shown in Figure 11.28. This output is on the disk.

The total cost for k-way merge sort is:

$$\begin{aligned} & 2N * (\#\text{ of passes}) \\ = \; & 2N * \lceil \log_k N \rceil + 1. \end{aligned}$$

where N, is the number of pages in a file. If a file contains 8 pages, the total cost for a 2-way merge is = 2*8*4 = 64. If an 8-way merge is used the total cost is = 2*8*2 = 32. But the main memory buffer required by a 2-way merge is 3, whereas for 8-way merge is 9.

Program 11.4

```
#include<iostream.h>
#include<conio.h>
void merge(int[],int,int);
void mergesort(int[],int,int);
int n,a[50],low,up,i,k;
void main()
{
   clrscr();
   cout<<"=================";
   cout<<"\nk-Way Merge Sort";
   cout<<"\n=================";
```

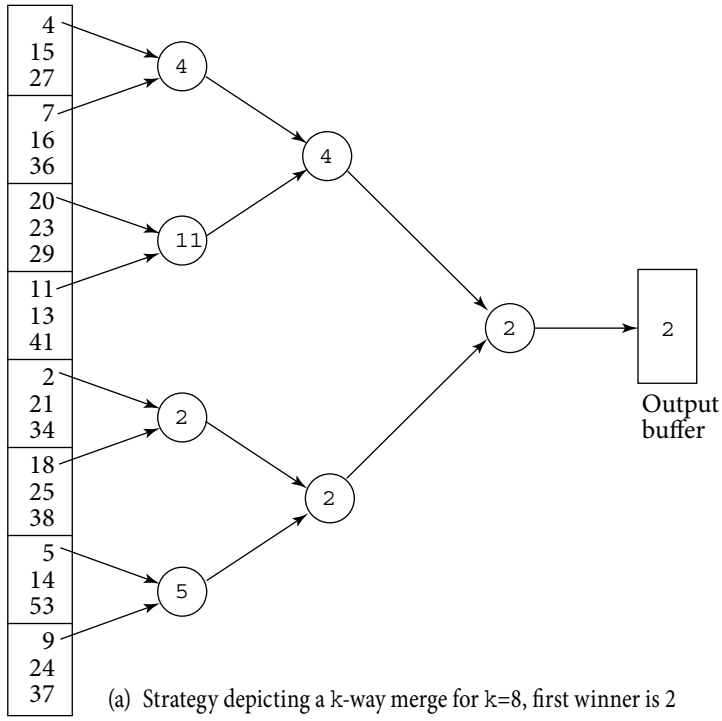
```
   cout<<"\nEnter K Value";
   cin>>k;
   cout<<"How Many Numbers You Want To Enter";
   cin>>n;
   cout<<"\nEnter"<<n<<"Elements";
   for(i=0;i<n;i++)
   cin>>a[i];
   cout<<s"\nBefore Sorting Record Elements Are\n";
   for(i=0;i<n;i++)
   cin>>a[i];
   merge(a,0,k-1);
   cout<<"\nAfter Sorting Record Elements Are:\n";
   for(i=0;i<n;i++)
   cin>>a[i];
   getch();
}
void merge(int a[],int l,int h)
{
   int count=1;
   while(count<=n)
   {
      mergesort(a,l,h);
      count=count+k;
      l=l+k;
      h=h+k;
   }
}
void mergesort(int a[],int l,int h)
{
   int i,j=0,b[50],k,temp;
   if(l==0)
   {
      for(i=l;i<=h;i++)
      b[j++]=a[i];k=j;
   }
 else
   {
      for(i=l-1;i>=0;i--)
         b[j++]=a[i];
      for(i=l;j<=h;i++)
         b[j++]=a[i];k=j;
   }
 for(i=j-1;i>0;i--)
   {
      for(j=0;j<i;j++)
         if(b[j]>b[j+1])
         b[j]=(b[j+1]+b[j])-(b[j+1]=b[j]);
   }
```

```
    j=0;
       for(i=0;i<k;i++)
          a[j++]=b[i];
       for(i=h+1;i<=n;i++)
          a[j++]=a[i];
}
```

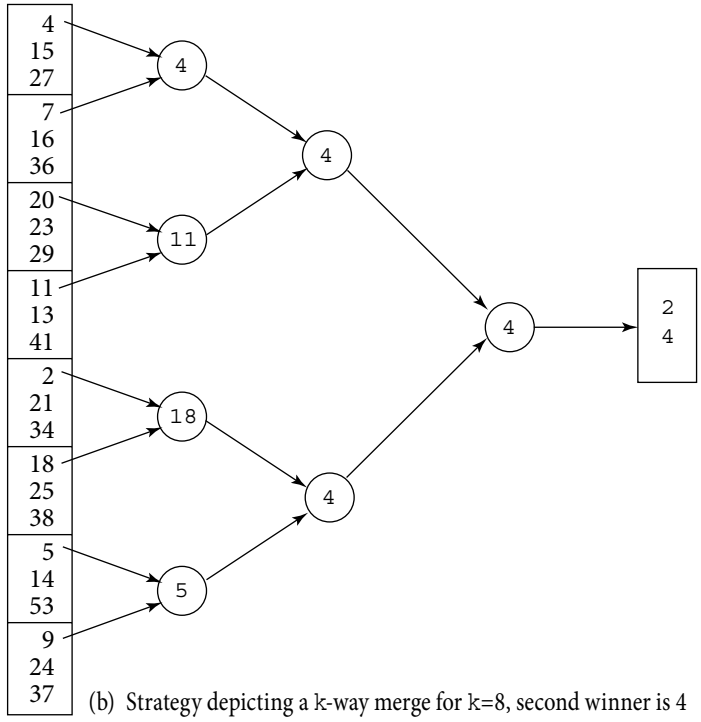
Output

```
==================
k-Way Merge Sort
==================
Enter K Value4
How Many Numbers You Want To Enter13
Enter Elements8
1
0
2
5
3
7
10
33
70
90
11
55
Before Sorting Record Elements Are
8 1 0 2 5 3 7 10 33 70 90 11 55
After Sorting Record Elements Are:
0 1 2 3 5 7 8 10 11 33 55 70 90
```

Program 11.4 sorts the given list of elements in ascending order using the k-way merge sort technique. The method `merge()` performs merging based on k-value and calls the `mergesort()` method which sorts the given list.



(a) Strategy depicting a k-way merge for k=8, first winner is 2



(b) Strategy depicting a k-way merge for k=8, second winner is 4

Figure 11.27 k-Way merge sort

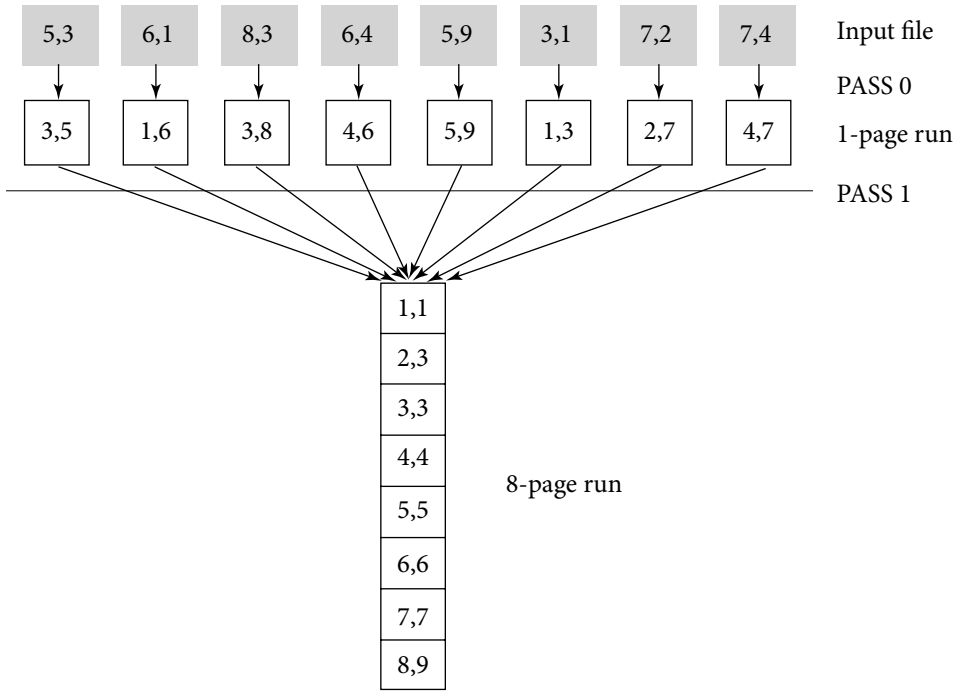


Figure 11.28 Example for k-way merge for k=8

SUMMARY

- A priority queue is a queue in which insertion and deletion of items are performed based on priority.
- Priority queue can be implemented using unsorted list, sorted list, binary search tree and heap.
- A min heap is a complete binary tree in which the value in each node is less than or equal to those in its children.
- A max heap is a complete binary tree in which the value in each node is greater than or equal to those in its children.
- External sorting is used for sorting files or lists that are large enough to store into the internal memory. So, external storage devices like tapes and disks are used for storing the files.
- Groups of records from files in memory are called runlists.
- A process of distributing ordered runlists of predetermined size on to the tapes and continuously merging these runlists in multiple phases is called the polyphase merge.

EXERCISES

Fill in the Blanks

1. Elements in the priority queue are associated with _____ value.
2. The average running time for deletion of an item from heap is _____.
3. Magnetic tape is a sequential storage medium used for _____ and _____.

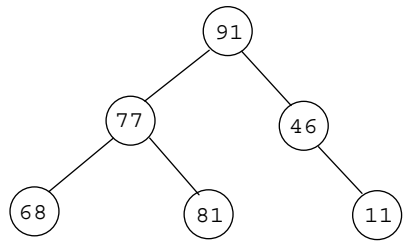
4. The technique involved in external sorting is _____.
5. Polyphase sort is used for sorting _____.

Multiple-choice Questions

1. The difference between priority queue and queue is
   a. Only insertion is done based on priority of the element
   b. Only deletion is done based on priority of the element
   c. Both insertion and deletion are done based on priority of the element
   d. None of the above
2. Priority trees can be implemented using
   a. AVL trees
   b. Stacks
   c. Heap trees
   d. Btrees
3. Which of the following statements is true?
   a. Heaps are binary search trees
   b. Heaps are complete binary trees
   c. Heaps are full binary trees
   d. Heaps contain only integer data
4. Sorting of disks can be implemented using the following algorithm.
   a. Heapsort
   b. Selection sort
   c. 2-way or k-way merge sort
   d. Insertion sort
5. The total cost of k-way merge sort is _____.
   a. $2N*(\log_2 N)+1$
   b. $2N*(\log_2 N)-1$
   c. $2\log_2 N+2N$
   d. $N*\log_2 N$

Short-answer Questions

1. Consider the array of elements = [10, 20, 15, 12, 17, 25, 19, 11, 5, 3, 22, 24].
   a. Build min heap
   b. Build max heap
   c. Insert an element 2 into min heap
   d. Insert an element 55 into max heap
   e. Delete min element from min heap and reheap
   f. Delete max element from max heap and reheap
2. Give two different reasons to explain why the following binary tree is not a heap:



3. Write a method to read the min/max element from min/max heap without deleting the element.
4. Write a method to check whether a min/max priority queue is empty or not.

5. Write a method to check whether a min/max priority queue is full or not.
6. Write a short notes on polyphase sorting with an example.

Essay Questions

1. Explain priority queues with the help of a suitable example.
2. Discuss in detail about implementation of priority queues using heaps.
3. Consider the keys for the records in a file = [2, 5, 3, 60, 34, 22, 45, 56, 43, 54, 23, 32, 5, 8, 6, 9, 44].
   a. Sort the records using polyphase merge using 4 tapes.
   b. Sort the records using polyphase merge using 5 tapes.
   c. Sort the records using 2-way merge sort.
   d. Sort the records using k-way merge sort (assume k value).
   e. Find the total cost in terms of I/O for the above three cases a, b, c, d.
4. Write a C++ program implementing polyphase sort algorithm.
5. Write a C++ program implementing k-way merge sort algorithm.

Chapter 12

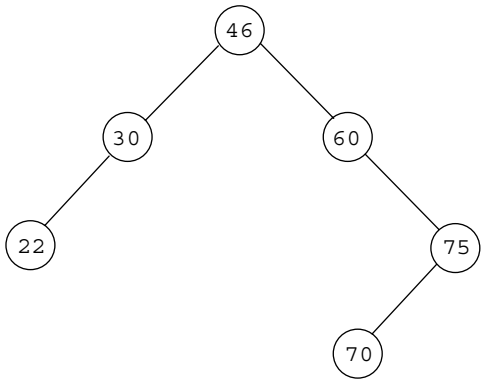
Binary Search Trees and AVL Trees

Chapter 9 discussed about trees and binary trees. Binary search trees and AVL trees which belong to the category of binary trees provide retrievals in an efficient manner. This chapter elaborates operations on binary search trees along with the definition. Listing the drawbacks of binary search trees, AVL trees which provide the remedy, are defined. Operations on AVL trees are explained. This chapter also includes applications and implementation of both the trees.

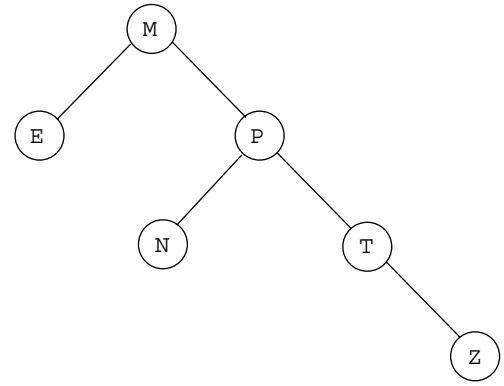
12.1 BINARY SEARCH TREES

A binary search tree (BST) is a binary tree. An empty binary tree is a binary search tree. A non-empty binary search tree should satisfy the following properties.

1. All the elements must have a key and it must be distinct.
2. All the keys (if any) in the left subtree of the tree must be less than the root element.
3. All the keys (if any) in the right subtree of the tree must be greater than the root element.
4. The left and right subtrees of the tree must also be the binary search trees.



(a) A non-empty binary search tree for {46, 60, 75, 30, 70, 22}



(b) A non-empty binary search tree for {M, P, E, T, N, Z}

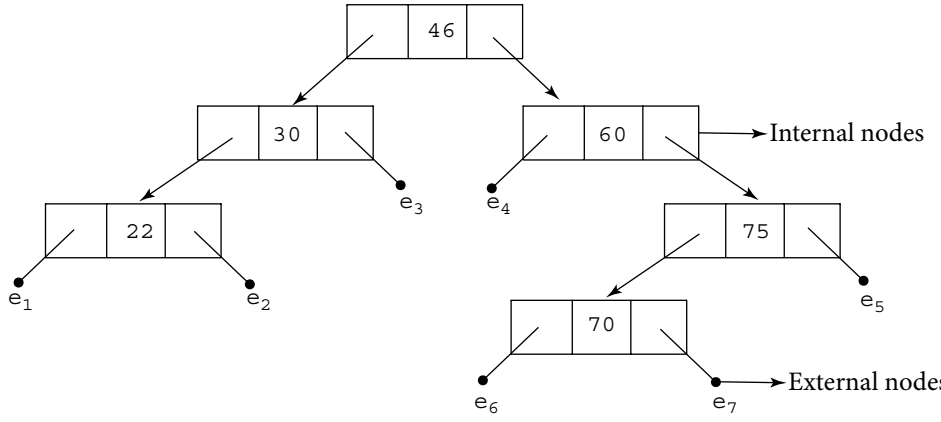
Figure 12.1 Binary search trees

The inorder traversal of a binary search tree produces the sorted list of the given elements in ascending order. The inorder traversal of the binary search trees shown in Figure 12.1(a) produces the sorted list as {22,30,46,60,70,75}, the ascending order of the given list {46,60,75,30,70,22}, and Figure 12.1(b) produces the sorted list as {E,M,N,P,T,Z}, the ascending order of the given list {M,P,E,T,N,Z}.

12.2 REPRESENTATION OF A BINARY SEARCH TREE

Similar to the representation of a binary tree, a binary search tree is also represented in a linked representation manner. To illustrate the representation, consider the binary search tree depicted in Figure 12.1(a); its node structure and linked representation are given in Figure 12.2.

(a) Node structure of a binary search tree



(b) Linked representation of a binary search tree of Figure 12.1(a)

Figure 12.2 Linked representation of a binary search tree

The null pointers of the nodes are not represented physically but are only fictitious and are known as external nodes. The external nodes are depicted with solid circles and are specified as e_i. The path from the root to an external node is known as an external path. Thus, the linked representation of a binary search tree is a collection of internal nodes that represent the keys and external nodes that represent null pointers. This kind of binary tree is known as an extended binary tree.

12.3 OPERATIONS ON BINARY SEARCH TREES

Various operations can be performed on binary search trees but the most frequently used are search, insert and delete operations.

12.3.1 Searching

In a binary search tree T searching for a key u is initiated at the root. If the key u is less than the key at the root then consider only the left subtree of T to search the key u. If the key u is greater than the key at the root then

consider only the right subtree of T to search the key u. If the key u is equal to the key at the root then the search is successful. If the root is null then it is an unsuccessful search. In the same way the search is continued in the subtrees also till the key u is either found or not. If the key u is found then it is a successful search, else it is an unsuccessful search.

All successful searches end at corresponding internal nodes, whereas all unsuccessful searches end at corresponding external nodes of a binary search tree. External nodes can also be termed as failure nodes. As specified in Section 11.1 the inorder traversal of a binary search tree produces the elements in ascending order such as $x_1<x_2<x_3<\ldots<x_n$. Same is the case with external nodes $e_0, e_1, e_2,\ldots,e_n$. Here e_0 catches all the unsuccessful searches of the keys that are less than x_1, e_1 catches all unsuccessful searches of keys between x_1 and x_2 and so on, i.e. e_i catches all the unsuccessful searches of keys that are greater than x_i and less than x_{i+1}.

While searching for a key, in binary search tree by comparing the key with the root element, search operation proceeds in only one of its subtrees. Hence, binary search tree is advantageous over sequential list and efficient in performing searching in a faster manner.

In case of successful search if the key is found at the root itself then it is considered to be the best case and its time complexity is O(1). If the key is found as one of the leaf nodes at a level equal to the height h of the binary search tree then the time complexity is O(n). In still worst cases the height h may be equal to the number of elements n, then the time complexity is O(n). In an average case where frequent insertions and/or deletions are assumed, the height h of the binary search tree is O(log n) and the time complexity is O(log n).

Algorithm 12.1 BST-SEARCH

This algorithm is used to search an ITEM in a binary search tree. Let PTR be the pointer to the node containing ITEM.

```
1. Check whether tree is empty.
2. Set PTR to root node pointer.
3. Loop until PTR is not equal to NULL.
4. CASE 1: check whether ITEM is less than the root node value, then proceed
           to left subtree.
   CASE 2: check whether ITEM is equal to root node value, then write "ITEM
           found, at", PTR.
   CASE 3: check whether ITEM is greater than the root node value,then proceed
           to right subtree.
5. End case
6. End loop
7. Write "ITEM not found", when PTR=NULL.
```

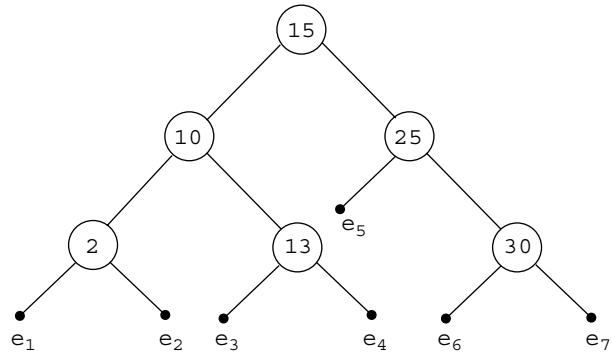


Figure 12.3 A binary search tree

In the BST shown in Figure 12.3, to search for the key 13, it is initially compared with the root 15 as 13 is less than 15 consider only the left subtree of node 15. Now, the root of the left subtree 10 is compared with the key 13 and as 13 is greater than 10, consider its right subtree. Proceed to the root of the right subtree 13 and it is compared with the key 13, as they are equal the search is successful.

Similarly to search for a key 28, it is compared with the root 15 as 28 is greater than 15 consider the right subtree to continue the search. Now, the root of the right subtree 25 is compared with 28 and as 28 is greater than 25 consider the right subtree of 25. Now, the root of the right subtree 30 is compared with the key 28 and as 28 is less than 30 its left subtree is considered to proceed the search but the node in the left subtree is an external node e_6, i.e. a failure node or NULL. So, the search is unsuccessful and the key 28 is not found in the BST.

12.3.2 Insertion

To perform an insertion operation, initially the search operation is to be performed. The place where a null pointer or failure node is found, a new node with key is inserted. So, insertion is an extension of search operation. Its time complexity is O(log n).

Algorithm 12.2 BST-INSERT

This algorithm inserts ITEM into the binary search tree if it doesn't exist before. Let PTR and NPTR be the two pointer variables.

```
1.  Start from the root node
    Set PTR to root node pointer
2.  Loop until PTR is not equal to NULL.
3.  CASE 1:Check whether ITEM is less than DATA of PTR, then proceed to left
           subtree.
           Set NPTR to PTR.
    CASE 2:Check whether ITEM is equal to DATA of PTR, then write"ITEM already
           exists".
    CASE 3:Check whether ITEM is greater than DATA of PTR, then
           Set NPTR to PTR
           Proceed to right subtree.
4.  End case
5.  End loop
6.  When PTR is equal to NULL then
7.  Call GETNODE(N)                        /*to create a new node*/
8.  SET  DATA of N to ITEM
9.  SET LCHILD of N to NULL.
10. SET RCHILD of N to NULL.
11. Check for DATA of NPTR to NULL, then
12. Set RCHILD of NPTR to N                /*insert as  right child*/
13. Else.
    Set LCHILD of NPTR                     /*insert as left child*/
14. END
```

To insert a key 20 into the binary search tree shown in the Figure 12.4, it is compared with the root 15. As 20 is greater than 15 its right subtree is chosen to find the correct place to insert 20. In the right subtree the key 20 is compared with its root 25 and as 20 is less than 25 its left subtree is considered to proceed with the insertion. The left subtree is NULL and is represented by the external or failure node e_5 and this is the exact position to insert the new node with the key 20.

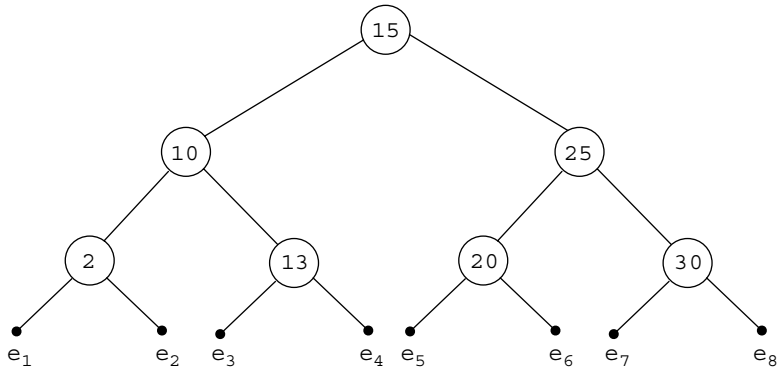


Figure 12.4 Insertion of an element 20 in the BST shown in Figure 12.3

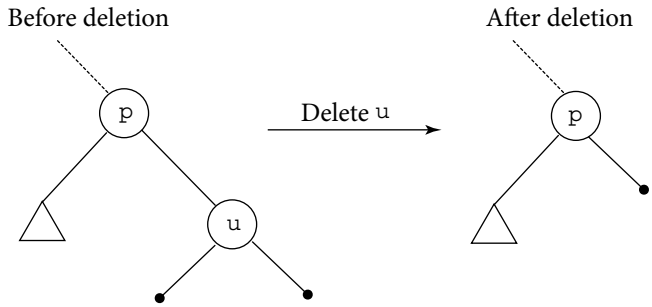
12.3.3 Deletion

The three different cases in which a node is deleted from a BST are when it

i. is a leaf node
ii. has only one subtree (either left subtree or right subtree)
iii. has both left and right subtrees

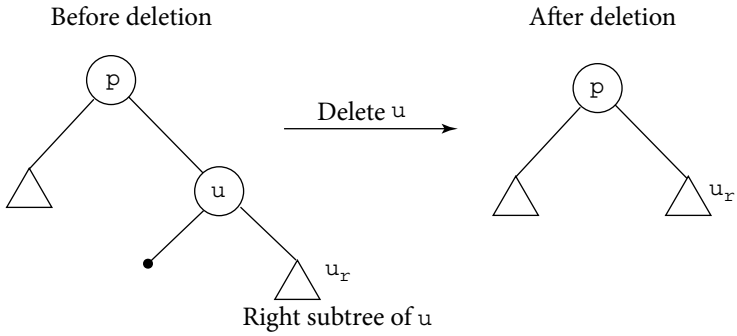
Case i: When the node to be deleted is a leaf node then the deletion is simple because only the link field of its parent node should be set to NIL. This is illustrated in Figure 12.5(a).
Case ii: When the node to be deleted has either a left or right subtree then the link field of its parent node should be set to point the corresponding subtree. This is illustrated in Figure 12.5(b).
Case iii: When the node to be deleted has both left and right subtrees then it is replaced either with the largest element in the left subtree or with the smallest element in the right subtree. After the replacement, the replacing element is deleted from its original position in the tree by using the appropriate deletion methods discussed in Case i or Case ii. It is assured that the replacing element has either empty subtrees or any one non-empty subtree. The right subtree holds the elements that are greater than u.

In the case where the link of the parent node of node (u) that is to be deleted is set to point the right subtree of u. Then to accommodate the left subtree of u move (to the farthest position left) as far left as possible in the right subtree of u till an empty left subtree is found, place or link the left subtree of u at this pointer. This is illustrated in the Figure 12.5(c).

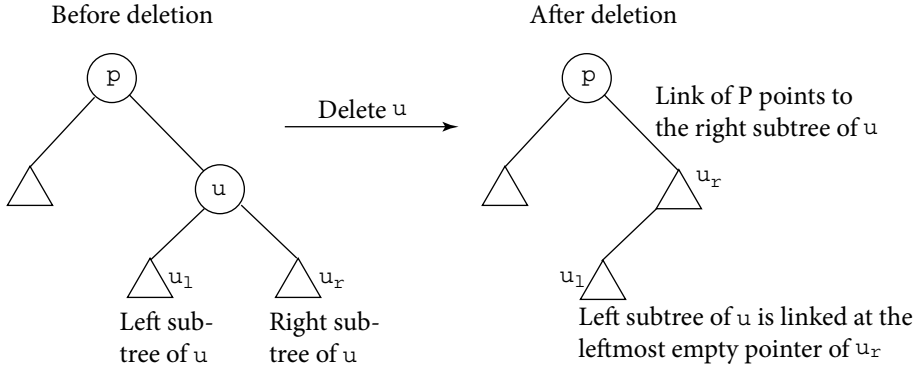


(a) Deletion of a leaf node u

Figure 12.5 Deletion of a node from a BST



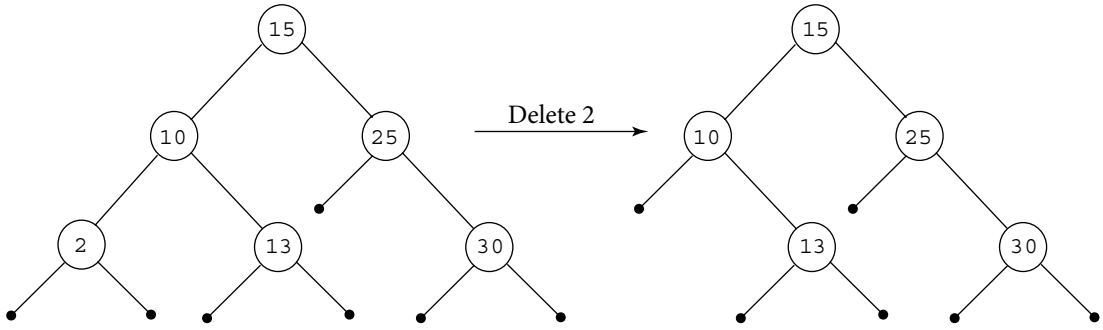
(b) Deletion of a node with one subtree



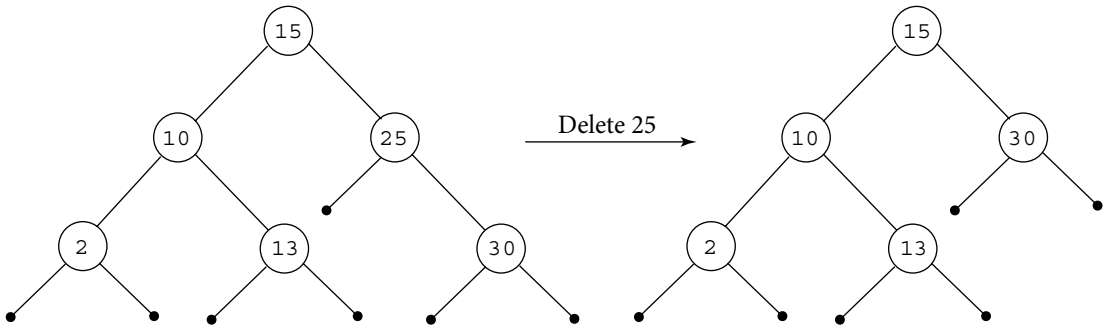
(c) Deletion of a node with both left and right subtrees

Figure 12.5 *Continued*

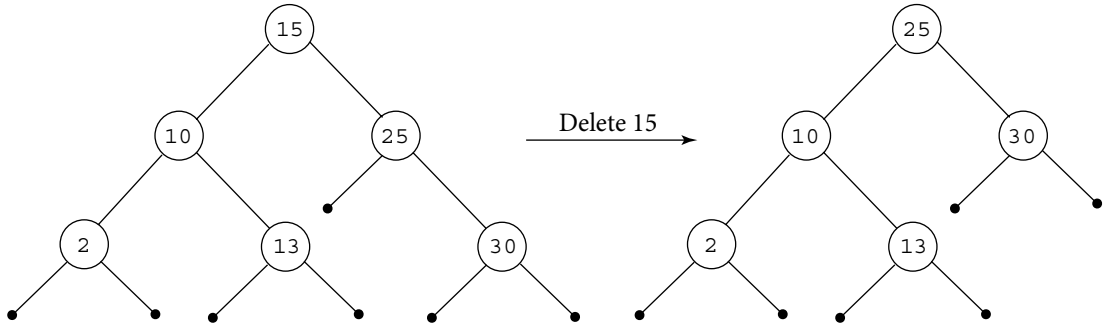
From the BST shown in Figure 12.3, Case i of deletion is illustrated by deleting a node 2, the leaf node. The LCHILD link of its parent node 10 is set to NULL, Figure 12.6(a) depicts this deletion. Deleting a node with one subtree. Case ii of deletion can be illustrated by deleting 25 that has only a single right subtree, i.e. 30. Now, the RCHILD link of its parent node 15 is set to point the node 30, Figure 12.6(b) depicts the deletion. Deletion of a node with both left and right subtrees. Case iii of deletion can be illustrated by deleting the root node 15. Now, node 25 the right subtree of 15 is the root and the left subtree with node 10 as the root is connected as the left subtree of the node 25. This deletion is depicted in Figure 12.6(c).



(a) Delete 2



(b) Delete 25



(b) Delete 15

Figure 12.6 Deletion of keys 2, 25 and 15 from the BST shown in Figure 12.3

Algorithm 12.3 BST-DELETION

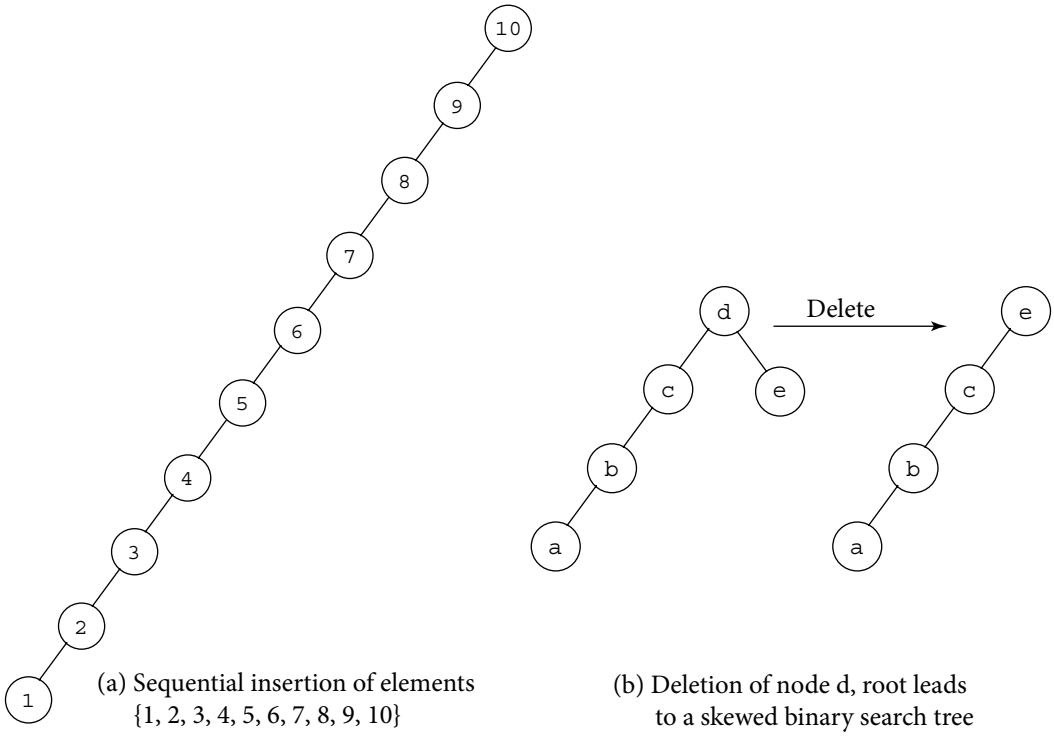
This algorithm is used to delete the node with ITEM as data. Let NODE be the node to be deleted.

```
1. Start from the root node
2. Search for the node with data as ITEM.
3. CASE 1: When node is a leaf node
           LCHILD of NODE==RCHILD of NODE=NULL
1. Set RCHILD of parent of NODE to NULL or LCHILD of parent of NODE to NULL
   based on whether NODE is left child or right child of its parent.
2. Call RETURN (NODE).
   CASE 2: When NODE contains both left and right subtrees
           LCHILD of NODE≠NULL and RCHILD of NODE≠NULL
1. Set RCHILD of parent of NODE to NULL or LCHILD of parent of NODE to RCHILD
   of NODE based on whether NODE is right child or left child of its parent
   respectively.
2. Set TEMP to RCHILD of NODE
3. Loop until LCHILD of TEMP not equal to NULL
   Set TEMP to LCHILD of TEMP
   End loop
4. Set LCHILD of temp to LCHILD of NODE
```

```
5. Call RETURN(NODE).
   CASE 3: When NODE has only left subtree
           LCHILD of NODE≠NULL and
           RCHILD of NODE=NULL
1. Set TEMP to LCHILD of NODE
2. Set RCHILD of parent of NODE or LCHILD of parent of NODE to TEMP based
   on whether NODE is right child or left child of its parent respectively.
3. Call RETURN (NODE).
   CASE 4: When NODE has only right subtree
           LCHILD of NODE=NULL and  RCHILD of NODE≠ NULL
1. Set TEMP to RCHILD of NODE
2. Set RCHILD of parent of NODE or LCHILD of parent of NODE to TEMP based
   on whether NODE is right child or left child of its parent respectively.
3. Call RETURN (NODE).
4. End case
5. End
```

12.3.4 Disadvantages of Binary Search Tree

A BST is highly efficient in performing all the operations with a time complexity of O(logn) over a sequential list. But, in the worst case where height h of the binary search tree is equal to the number of nodes n, its performance deteriorates. This happens because of its sequence of insertion and/or deletion operations which results in skewed BSTs with the time complexity of O(n). The binary search trees in Figure 12.7 show the skewed BSTs.



(a) Sequential insertion of elements {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

(b) Deletion of node d, root leads to a skewed binary search tree

Figure 12.7 Skewed binary search tree

This kind of skewed BST deteriorates its performances. This also destroys the basic purpose of the BST that in a binary search tree only one half is considered for searching. At the root node itself which reduces half the searching time, it can be decided that which half of the binary search tree is considered and which half is deducted.

Hence, to retain the purpose of the BST its height should be checked and kept balanced. Even in worst case the trees whose height is O(log n) are termed as balanced trees.

Program 12.1

```
/*BINARY SEARCH TREE OPERATIONS
1. INSERTION 2.DELETION 3.TRAVERSING 4.SEARCHING*/

#include<stdio.h>
#include<stdlib.h>
#include<iostream.h>
#include<conio.h>
struct BST
{
   int data;
   struct BST*lchild,*rchild;
}*root=NULL,*ptr,*ptr1,*n;
 int item,k=0,s;
 void insert()
{
   cout<<"Enter Element To Be Insert";
   cin>>item;
   ptr=root;
   if(ptr==NULL)
 {
   n=(struct BST*)malloc(sizeof(struct BST));
   n->data=item;n->lchild=n->rchild=NULL;
   cout<<"Element Is Successfully Inserted";
   root=n;return;
 }
   while(ptr!=NULL)
 {
  if(item<ptr->data)
  {
     ptr1=ptr;
     ptr=ptr->lchild;
  }
  else if(item>ptr->data)
  {
     ptr1=ptr;
     ptr=ptr->rchild;
  }
  else
  {
     if(item==ptr->data)
```

```
      {
         cout<<"Item Already Exists";
         return;
       }
     }
 }
     if(ptr==NULL)
 {
     n=(struct BST*)malloc(sizeof(struct BST));
     n->data=item;n->lchild=n->rchild=NULL;
     if(ptr1->data<item)
     ptr1->rchild=n;
     else
     ptr1->lchild=n;
 }
}
void case1()
{
   if(ptr1->lchild==ptr)
   ptr1->lchild=NULL;
   else
   ptr1->rchild=NULL;
   cout<<"Element Is Successfully Deleted";
   ptr=NULL;
}
void case2()
{
   struct BST*ptr2,*y;
   int x=0;
   ptr2=ptr->rchild;
   if(ptr2->lchild==NULL&&ptr2->rchild==NULL)
 {
   ptr1->lchild=ptr2;
   ptr1->lchild->lchild=ptr->lchild;
 }
  else if(ptr2->lchild==NULL&&ptr2->rchild!=NULL)
 {
   ptr1->rchild=ptr2;
   ptr1->rchild->lchild=ptr->lchild;
   ptr=NULL;
   free(ptr);
 }
 else
 {
   while(ptr2->lchild!=NULL)
    ptr2=ptr2->lchild;
   if(ptr2->rchild!=NULL)
   {x=1;y=ptr2->rchild;}
```

```
    ptr1->lchild=ptr2;
    ptr2->lchild=ptr->lchild;
    ptr2->rchild=ptr->rchild;
    if(x==1)
    ptr2->rchild->lchild=y;
   else
     ptr2->rchild->lchild=NULL;
     ptr=NULL;
   free(ptr);
  }
 }
void case3()
{
 if(ptr1->lchild==ptr)
 {
  if(ptr->lchild==NULL)
   ptr1->lchild=ptr->rchild;
  else
   ptr1->lchild=ptr->lchild;
 }
 else
 {
  if(ptr1->rchild==ptr)
  {
   if(ptr->lchild==NULL)
    ptr1->rchild=ptr->rchild;
   else
    ptr1->rchild=ptr->lchild;
  }
 }
 cout<<"Element Is Successfully Deleted";
 ptr=NULL;
}
void delet(int item)
{
 int c=0;
 struct BST*pt;
 ptr=root;
 if(ptr==NULL)
 {
  cout<<"Tree Is Empty";
  return;
 }
 if(ptr->data==item)
 {
  cout<<item<<"Is A Root Node Deletion Is Not Possible";
  return;
 }
```

```
 if(ptr->data==item&&ptr->lchild==NULL&&ptr->rchild==NULL)
 {
  root=NULL;
  cout<<"Item"<<item<<"Is Successfully Deleted";
  return;
 }
 while((ptr!=NULL)&&(c==0))
 {
  if(item<ptr->data)
  {
   ptr1=ptr;
   ptr=ptr->lchild;
  }
  else if(item>ptr->data)
  {
   ptr1=ptr;
   ptr=ptr->rchild;
  }
  else
   c=1;
 }
 if(c==0)
 {
  cout<<"Item Does Not Exist..,Deletion Is Not Possible";
  return;
 }
 if(ptr->lchild==NULL&&ptr->rchild==NULL)
 case1();
  else if(ptr->lchild!=NULL&&ptr->rchild!=NULL)
 case2();
  else
 case3();
}
void traverse(struct BST*ptr)
{
 if(ptr!=NULL)
 {
  traverse(ptr->lchild);
  cout<<ptr->data<<" ";
  traverse(ptr->rchild);
 }
}
int searching(struct BST*p,int key)
{
 if(p!=NULL)
 {
  if(p->data==key)
   k=1;
```

```
    else
    {
     searching(p->lchild,key);
     searching(p->rchild,key);
    }
  }
  return k;
}
main()
{
 int ch;
 clrscr();
 while(1)
 {
  cout<<"\nBINARY SEARCH TREE OPERATIONS\n1.INSERTION\n2.DELETION\n";
  cout<<"3.TRAVRSE\n4.SEARCHING\n5.EXIT";cout<<"\nEnter Your Choice";
  cin>>ch;
  switch(ch)
  {
   case 1:insert();break;
   case 2:cout<<"Which Item You Want To Delete";cin>>item;
     delet(item);break;
   case 3:if(root==NULL){cout<<"Treee Is Empty";break;}
         cout<<"Tree Elements Are";traverse(root);break;
   case 4:if(root==NULL){cout<<"Tree Is Empty";break;}
     s=k=0;cout<<"Which Item You Want To Search";
     cin>>item;
     s=searching(root,item);
     if(s==1)
      cout<<"Item"<<item<<"Is Found";
     else
      cout<<"Item"<<item<<"Is Not Found";
     break;
   case 5:exit(0);
   default:cout<<"Invalid Choice";
  }
 }
}
```

Output

```
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert35
```

```
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert20
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert40
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert10
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert24
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert22
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert21
```

```
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert23
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert25
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice1
Enter Element To Be Insert30
Enter Your Choice3
Tree Elements Are10 20 21 22 23 24 25 30 35 40
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice2
Which Item You Want To Delete24
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice3
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice3
```

```
Tree Elements Are 10 20 21 22 23 25 30 35 40
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice2
Which Item You Want To Delete20

BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice3
Tree Elements Are 10 21 22 23 25 30 35 40
BINARY SEARCH TREE OPERATIONS
1.INSERTION
2.DELETION
3.TRAVRSE
4.SEARCHING
5.EXIT
Enter Your Choice5
```

12.4 AVL TREES

In 1962, Adelson-Velskii and Landis proposed balanced or height balanced trees known as AVL trees. An AVL tree is one among the balanced trees and the height of the tree will be O(log n) in the worst case. AVL trees are height balanced versions of binary search trees.

Definition: An empty binary tree is an AVL tree. A non-empty binary tree T is an AVL tree if and only if

(i) the left subtree T_L and right subtree T_R of T are also AVL trees

(ii) $|h(T_L) - h(T_R)| \leq 1$ where $h(T_L)$ and $h(T_R)$ are the heights of left and right subtrees of T, respectively.

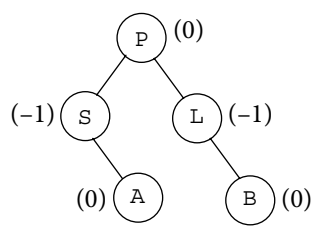
The balancing factor of a node u, denoted as bf(u), can be defined as $bf(u) = h(u_L) - h(u_R)$ where $h(u_L)$ is the height of the left subtree of u and $h(u_R)$ is the height of the right subtree of u.

Every node u of an AVL tree is allowed to have a balancing factor bf(u) of –1 or 0 or +1.

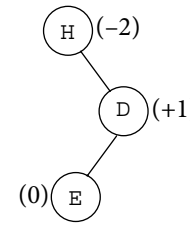
A binary search tree which is (also possesses the quality of) an AVL tree is known as an AVL search tree.

Figure 12.8 gives a clear view of the above-discussed terms. The balance factors of the corresponding nodes are shown within parentheses adjacent to the nodes. Observe how the balance factors are either –1 or 0 or 1.

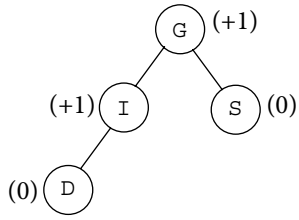
Representation of AVL trees and AVL search trees are very similar to that of binary trees and BTSs using a linked representation. In order to facilitate insertion and deletion operation efficiently an additional field called balance factor bf can be added with the node structure to hold the balance factor of a particular node.



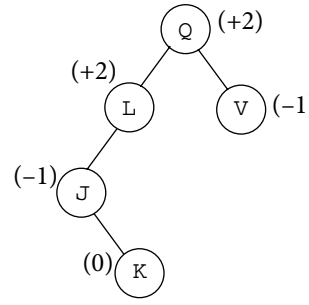
(a) An AVL tree



(b) A non-AVL tree



(c) An AVL search tree



(d) A non-AVL search tree

Figure 12.8 Instances of AVL and non-AVL trees

12.5 OPERATION OF AVL SEARCH TREES

The following are the common operations that are performed on AVL search trees.

12.5.1 Searching

Searching for an element in an AVL search tree is quite similar to the search operation on a BST. Algorithm 12.1 and its implementation in Program 12.1 can also be used to search an element from an AVL search tree also. The height of an AVL search tree of `n` elements is O(log `n`) and, hence, the time complexity of a search operation on it is also O(log `n`).

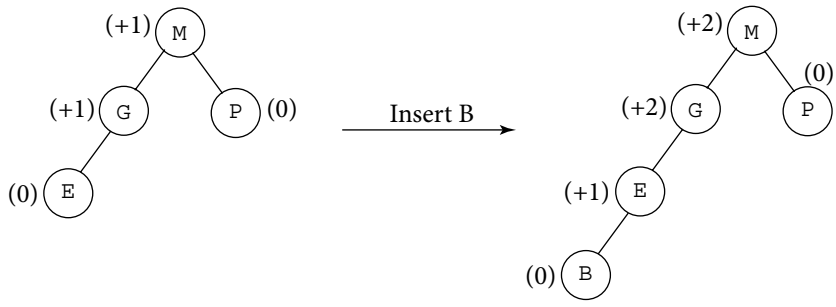
12.5.2 Insertion

Inserting an element into an AVL search tree follows the same procedure as the insertion of an element into a binary search tree. But, the insertion may lead to a situation where the balance factors of any of the nodes may be other than –1, 0 and +1 and the tree is unbalanced.

In order to rebalance the tree, i.e. to acquire all balance factors as –1 or 0 or 1, some of the subtree of the unbalanced trees are to be shifted which can also be referred to as rotations. This can be shown in Figure 12.9.

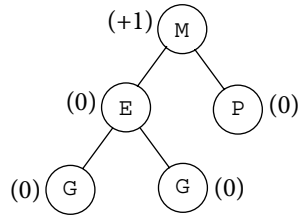
The following observations can be made regarding an unbalanced tree due to insertion:

1. The balance factor of an unbalanced tree can only be among –2, –1, 0, 1 and +2.
2. After insertion, for the nodes whose balance factor is either +2 or –2 their corresponding balance factors before insertion would have been +1 or –1, respectively.



(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after inserting B



(c) Rebalanced AVL search tree

Figure 12.9 Insertion into an AVL search tree

3. Insertion may affect the balance factors of only those nodes that appear on the path from the root to the new node that is inserted.
4. The nearest ancestor A of the newly inserted node whose balance factor is either +2 or –2 initiates the rotation.

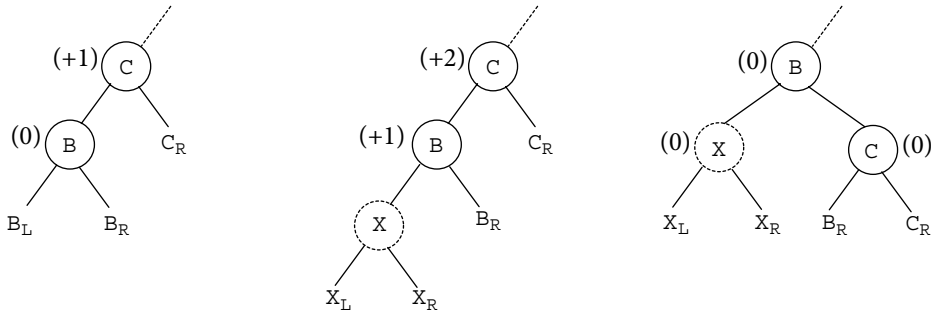
In an AVL search tree, prior to the insertion, if the balance factor of all the nodes on the path from the root node to the currently inserted node was 0, then the tree will not become unbalanced even after insertion. Because, the insertion will make the balance factor to change by –1, 0 or 1.

If the insertion makes an AVL search tree unbalanced, the height of the subtree must be adjusted by rotating them approximately with respect to the closest ancestor so that it is balanced.

Based on the place where the new node is inserted, that causes the imbalance rotations are classified into four types such as

1. LL rotation—new node inserted as the left subtree (L) of the left subtree (L) of A.
2. LR rotation—new node inserted as the right subtree (R) of the left subtree (L) of A.
3. RR rotation—new node inserted as the right subtree (R) of the right subtree (R) of A.
4. RL rotation—new node inserted as the left subtree (L) of the right subtree (R) of A.

LL rotation: General representation of LL rotation is illustrated in Figure 12.10. This figure shows how insertion leads to LL type imbalance and how the rotation is made to balance the tree again. Figure 12.10(a) shows the balanced AVL search tree before insertion in which B,C are with balance factors 0, +1, respectively. B_L, B_R and C_R are the subtrees of B and C.



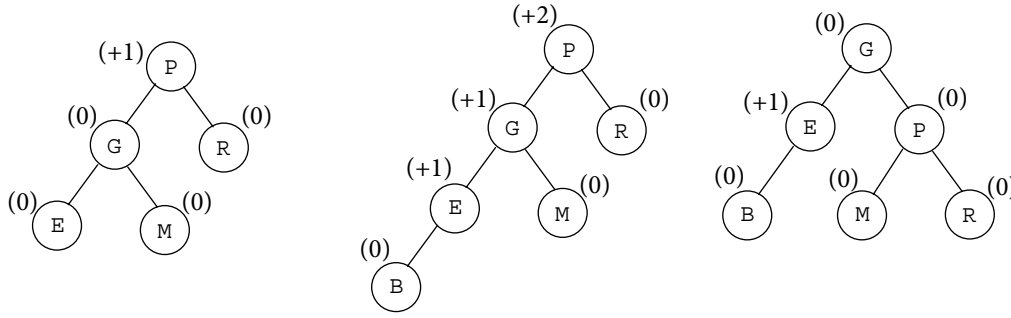
(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after insertion of X into B_L

(c) Balanced AVL search tree after LL rotation

Figure 12.10 General representation of LL rotation

In this a new node called X is inserted as a left subtree (L) of left subtree (L) of C, the ancestor node. Now the balance factors of B change from 0 to +1 and C changes from +1 to +2 and the tree is unbalanced; this is shown in Figure 12.10(b). To rebalance the tree LL rotation must be invoked. This rotation pushes B up into place of C and C is moved into the place of its right subtree earlier. The new node remains as the left subtree of B and B_R earlier right subtree of B is now the left subtree of C along with its right subtree, C_R remained as it was. This is shown in Figure12.10(c) where the height of the subtree is balanced and the balance factor of all nodes is 0. Hence, the AVL search tree is balanced after LL rotation.



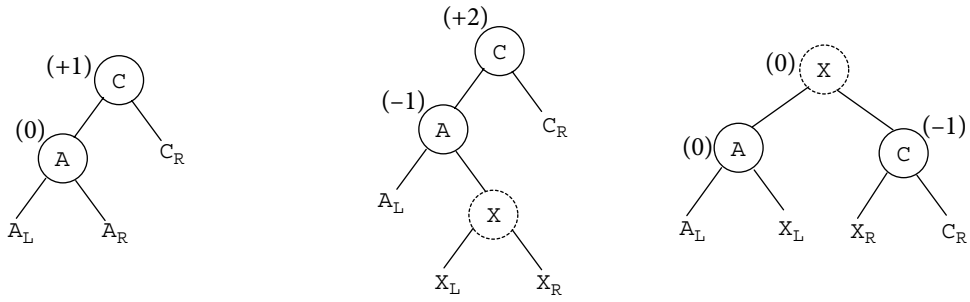
(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after insertion of B

(c) Balanced AVL search tree after LL rotation

Figure 12.11 Example for LL rotation

Consider Figure 12.11 which is an instance of LL rotation. Figure 12.11(a) shows an AVL search tree prior to the insertion. Insertion of an element B as a left subtree of left subtree of ancestor node P, whose balance factor with insertion has changed from +1 to +2 disturbed the balance of the tree. This is shown in Figure 12.11(b). LL rotation shifts G up into the place of P and P is moved down along with its right subtree R. Node M, earlier right subtree of G, now becomes the left subtree of P. It can be observed from Figure 12.11(c) that the height of the tree is balanced and all the balance factors are either –1 or 0 or +1. General representation of LR rotation is depicted in Figure 12.12. It illustrates how an insertion leads to LR type imbalance and rebalancing the tree using corresponding rotation. Figure 12.12(a) shows a balanced AVL search tree prior to insertion. In this tree



(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree tree after insertion into A_R

(c) Balanced AVL search tree after LR rotation

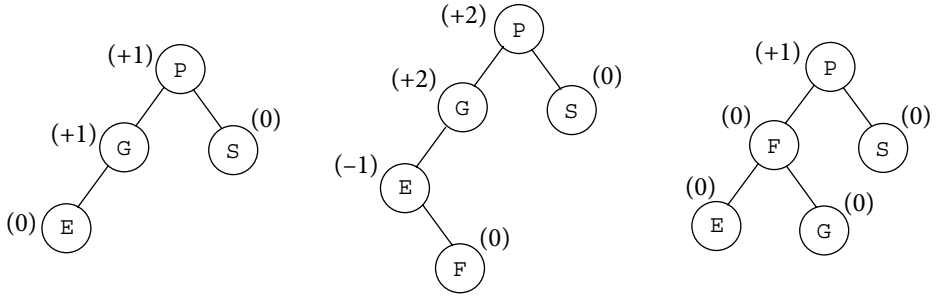
Figure 12.12 General representation of LR rotation

C and A are nodes with balance factors as +1 and 0, respectively. A_L and A_R are subtrees of A and C_R is the right subtree of C.

Inserting some node X as a right subtree of A, i.e. in A_R, changes the balance factor of A from 0 to –1 and C from +1 to +2. A new node is inserted as a right subtree (R) of left subtree (L) of the closest ancestor node C and this leads to an imbalance of tree. This is shown in Figure 12.12(b). In order to rebalance the tree LR rotation is invoked. This rotation pushes the new node X up to the place of C, the ancestor node where the imbalance is observed. C is shifted into the position of its right subtree C_R. Figure 12.12(c) shows the balanced AVL search tree after LR rotation in which X is the root and A is the left subtree with A_L as its left subtree and X_L the earlier left subtree of X as its right subtree. C is the right subtree with X_R the earlier right subtree of X as its left subtree and C_R as its right subtree.

Here, balance factors of nodes A and C after LR rotation are based on the balance factor of the new node after insertion and before rotation.

if bf(X)=0 then bf(A) and bf(C) will also be 0 after rotation
if bf(X)=1 then bf(A)=0 and bf(C)=–1 after rotation
if bf(X)=–1 then bf(A)=1 and bf(C)=0 after rotation.



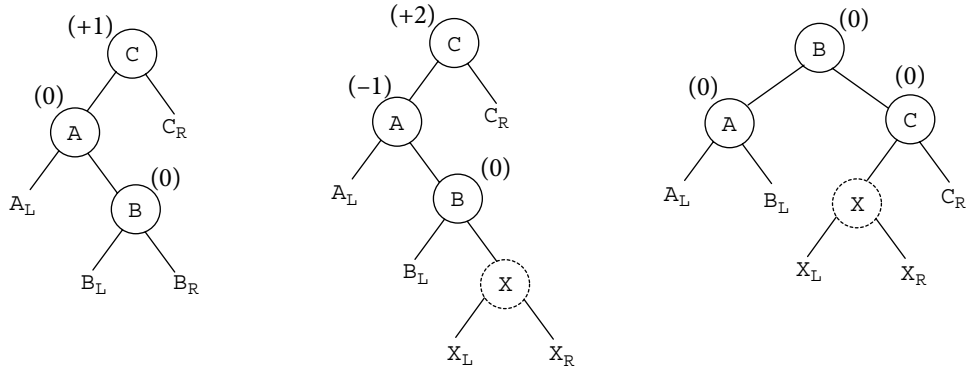
(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after inserting F

(c) Balanced AVL search tree after LR rotation

Figure 12.13 Example of LR rotation

Consider Figure 12.13, an instance of LR rotation. Figure 12.13(a) shows the balanced AVL search tree prior to insertion. After inserting F as a right subtree of left subtree of the closest ancestor G, the balance factor of the nodes in the path from the root to the newly inserted node is changed. Balance factor of node E has been changed from 0 to –1 and that of node G and node P has been changed from +1 to +2. The tree becomes imbalanced as shown in Figure 12.13(b). The LR rotation shifts F, the new node into the place of G and G is pushed down as the right subtree of F. Now, the tree is once again a balanced AVL search tree as shown in Figure 12.13(c) with the balance factors of all the nodes as +1 or 0.



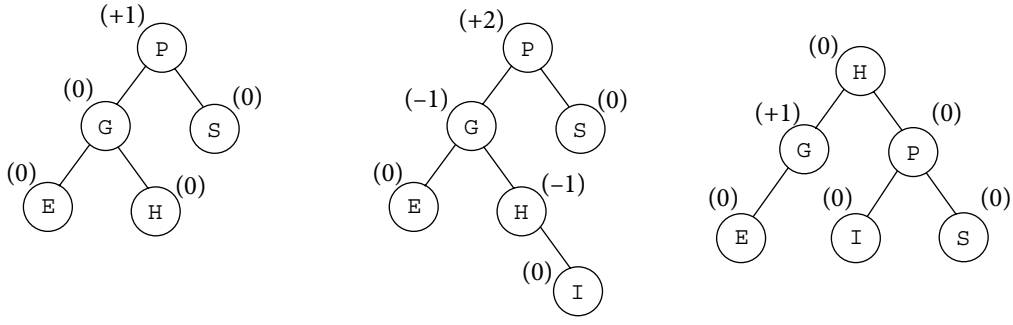
(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after insertion into B_R

(c) Balanced AVL search tree after LR rotation

Figure 12.14 General representation of complex LR rotation

Figure 12.14 shows the general representation of complex LR rotation. Figure 12.14(a) shows balanced AVL search tree prior to insertion. Here C is with the balance factor +1, nodes A and B are with 0 balance factor. After inserting some node X as the right subtree (R) of left subtree (L) into B_R then the balance factor of C changes to +2 from +1, nodes A and B change to –1 from 0. Now, the tree is unbalanced as shown in Figure 12.14(b). LR rotation is invoked to rebalance the tree. After rotation the tree is balanced as shown in Figure 12.14(c).



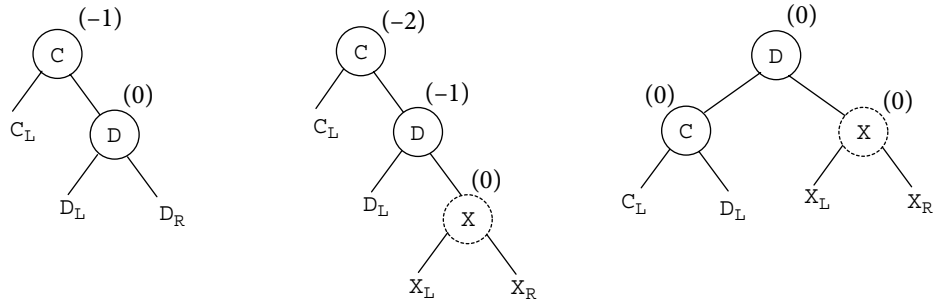
(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after inserting I

(c) Balanced AVL search tree after LR rotation

Figure 12.15 Example of LR rotation

Consider Figure 12.15, an instance of LR rotation. Figure 12.15(a) shows the balanced AVL search tree prior to insertion. After inserting I as the right subtree of the left subtree of the closest ancestor G, the balance factor of the nodes in the path from the root to the newly inserted node is changed. The balance factor of nodes G and H has been changed from 0 to –1 and node P has been changed from +1 to +2. This leads to the imbalance of the tree as shown in Figure 12.15(b). Invoke LR rotation, this shifts H up into the place of P and P is pushed down as its right subtree along with S. Now the node I becomes the left subtree of P. The rebalanced AVL search tree is shown in Figure 12.15(c).



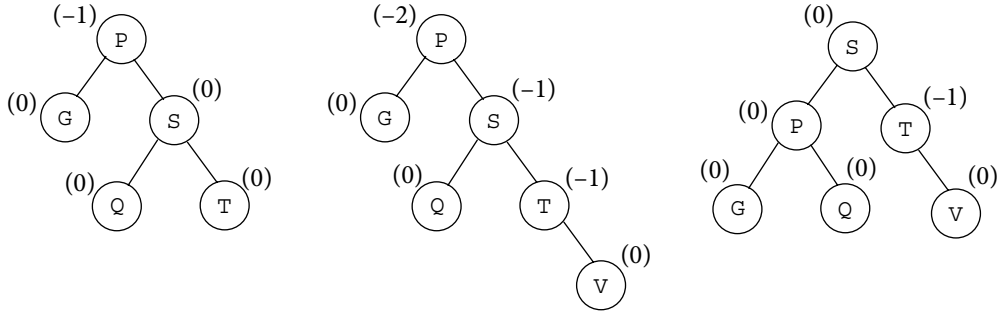
(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after insertion into D_R

(c) Balanced AVL search tree after RR rotation

Figure12.16 General representation of RR rotation

RR rotation: The general representation of RR rotation is shown in Figure 12.16. The balanced AVL search tree prior to the insertion is shown in Figure 12.16(a). Here C and D are with the balance factors –1 and 0, respectively, D_L and D_R are the subtrees of D and C_L is the left subtree of C. When some element X is inserted as a right subtree (R) of the right subtree (R), i.e. in D_R. The balance factor of D changes from 0 to –1 and that of C, the closest ancestor changes from –1 to –2. This indicates that the tree got RR type imbalanced after insertion, and is shown in Figure 12.16(b). Invoking the RR rotation shifts node D up into the place of C and C is moved to be the left subtree of D. The new node X remained as the right subtree of D with X_L and X_R as its subtrees. Node C, along with its left subtree C_L as it was, has D_L, earlier the left subtree of D, as its right subtree. Now the tree is a balanced AVL search tree after RR rotations with the balance factor as 0 for all the nodes. This is shown in Figure 12.16(c).



(a) Balanced AVL search tree before insertion

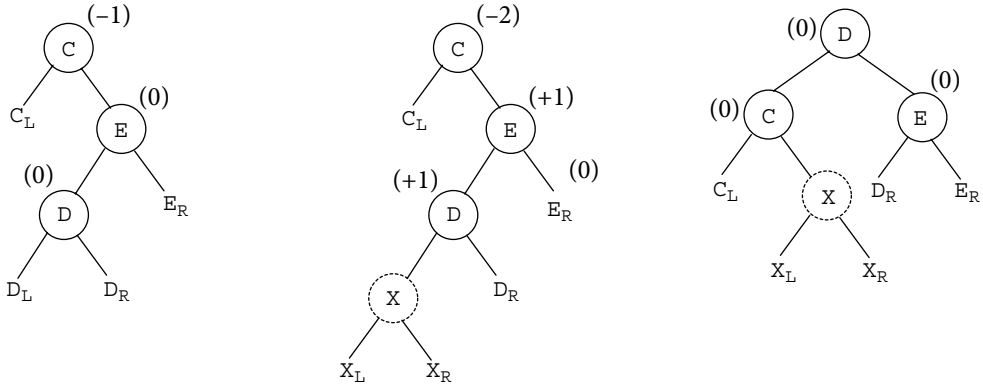
(b) Unbalanced AVL search tree after inserting V

(c) Balanced AVL search tree after RR rotation

Figure 12.17 Example of RR rotation

Consider an instance of RR rotation depicted in Figure 12.17. The balanced AVL search tree before insertion is shown in Figure 12.17(a). After inserting node V as right subtree of right subtree, i.e. as the right subtree of T the balance factor of all the nodes in the path from root P to the newly inserted node V changes and makes the tree imbalanced as shown in Figure 12.17(b). The balance factor of T and S changed from 0 to –1 and that of P changed from –1 to –2. RR rotation is invoked to rebalance the tree. This pushes S up into the place of P and P now becomes the left subtree of S. Node Q earlier the left subtree of S now becomes the right subtree of P. The balanced AVL search tree after RR rotation is shown in Figure 12.17(c) with the balance factor of all the nodes as either 0 or –1.

RL Rotation: The general representation of RL rotation is depicted in Figure 12.16.



(a) Balanced AVL search tree before insertion

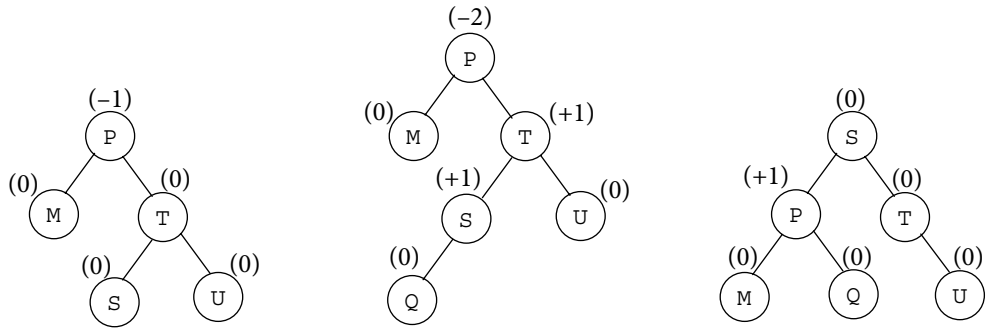
(b) Unbalanced AVL search tree after insertion into D_L

(c) Balanced AVL search tree after RL rotation

Figure 12.18 General representation of RL rotation

The balanced AVL search tree prior to insertion with nodes C and E having balance factors as –1 and 0, respectively, is shown in Figure 12.18(a). E_L and E_R are subtrees of E and C_L is the left subtree of C. Insertion of some element X has a left subtree (L) of a right subtree (R) leads to R_L type imbalance of that tree by changing the balance factor of the closest ancestor node C from –1 to –2. The balance factor of E is also changed from 0 to +1. This is shown in Figure 12.18(b). The RL rotation is called to rebalance the tree. This rotation shifts the new node X up into the place of the ancestor node C and C is pushed into the place of its left subtree. Now X is with the left subtree C along with its left subtree C_L and X_L as its right subtree which was earlier the left subtree of X. Node E is now the right subtree of X with E_R as its right subtree and X_R as its left subtree which was earlier the right subtree of X. Now the tree is a balanced AVL search tree after R_L rotation. This is shown in Figure 12.18(c).

Consider the instance of RL rotation shown in Figure 12.19. The balanced AVL search tree prior to the insertion is shown in Figure 12.19(a). After the new node Q is inserted as a left subtree of a right subtree, the balance factors in the path from root P to Q are changed. The balance factors of S and T changed from 0 to +1 and that of P changed from –1 to –2 which made the tree the RL type imbalance. This is shown in Figure 12.19(b). The RL rotation is invoked to rebalance the tree. This rotation shifts the parent of the new node S into the place of the ancestor node P and P is moved down into the place of its left subtree along with node M. The new node Q becomes the right subtree of P which was earlier the left subtree of S. Now tree is with balanced heights of subtrees with all the nodes having the balance factors as either 0 or +1. Hence, the tree is a balanced AVL search tree after RL rotation.



(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after inserting Q

(c) Balanced AVL search tree after RL rotation

Figure 12.19 Example of RL rotation

The transformations that are performed to solve the LL and RR type imbalances are known as single rotations, and that of LR and RL type imbalances are known as double rotations. The transformation of LR type imbalance can be considered as the sequence of an RR rotation followed by an LL rotation, and that of RL type imbalance can be considered as an LL rotation followed by RR rotation.

Algorithm 12.4 Insertion into an AVL tree

```
Insert(T, element)
1. GETNODE(X)
2. DATA(X)=element
3. LCHILD(X)=RCHILD(X)=NULL and bf(x)=0
4. If the tree T is empty then
   (a) Set T to X.
   (b) Exit.
   //when AVL search tree T is non empty.
5. Starting from the root search the place to insert the new element.
6. Identify the most recently seen node with balance factor of either -1 or
   +1 as the ancestor node A.
7. If the element already exists
   (i)  Print "Insertion not possible as the element already exists".
   (ii) Exit.
8. If no ancestor node A exists then update the balance factors in the path
   from root and exit.
9. If ancestor node A is found then
   (a)  If(bfA)=+1 and the new node is inserted in the right subtree of A)
        or (bf(A)=-1 and the node is inserted in the left subtree of A)then
        (i)   bf(A)=0.
        (ii)  Update the balance factors of all the nodes on the path from A
              to the newly inserted node.
        (iii) Exit.
   (b)  Else
        (i)   Recognize the type of imbalance at A and execute the appropriate
              rotation.
```

```
        (ii) Update the balance factors of nodes in the path from new subtree
             root to the newly inserted node as needed by the rotation.
        (iii) Reset the left and right subtrees of the corresponding nodes.
   (c)  Exit
10.End.
```

The time complexity of insertion operation into an AVL search tree is O(height)=O(log n).

12.5.3 Deletion

Deletion of an element from an AVL search tree may also cause imbalance similar to insertion, rotations are invoked to rebalance the tree after deletion. Major classification of rotations are as L and R which can be further categorized as L0, L1, L-1 regarding L rotations and R0, R1, R-1 regarding R rotations.

Let p be the parent of the node that was deleted. Due to deletion the balance factors of some or all of the nodes on the path from the root to p might be changed. It is required to update the balance factors from node p to the root in the upward direction. All the deletions may not call the rotations. They are referred to as rotation free deletions.

The following observations can be made regarding deletion:

1) If the node was deleted from the right subtree of p then bf(p) increases by 1 and if it was from the left subtree of p then bf(p) decreases by 1.
2) The height of the tree is decreased by 1 when the new bf(p)= 0. It is required to update the balance factor of its parent and/or its ancestors.
3) The height of the tree remains the same when the new bf(p)=+1 or –1 and there will be no change in the balance factors of its ancestor nodes.
4) The tree is imbalanced at node p when the new bf(p)= +2 or –2, and hence corresponding rotations are to be invoked.

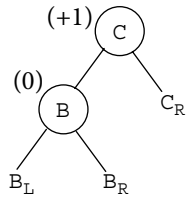
In this process the balance factors of some nodes may become +2 or –2. In the path from p to the root the first node among them is A, the ancestor node. At node A the balance has to be restored, so the imbalance can be classified as type L when the deletion is from the left subtree of A and it is of type R when the deletion is from the right subtree of A.

Regarding type L, the bf(A) would have been –1 prior to deletion when it is now –2; bf(A)=–2. Ancestor node A might have a right subtree with root B. With respect to the bf(B) the L type rotations are further classified as L0 if bf(B) =0, L1 if bf(B) =+1 and L–1 if bf(B)=–1.

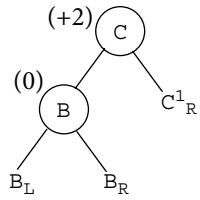
Regarding type R, the bf(A) would have been +1 prior to the deletion when it is now +2; bf(A)=+2. Ancestor node A might have a left subtree with root B. With respect to the bf(B) the R type rotations are further classified as R0 if bf(B)=0, R1 if bf(B)=+1 and R–1 if bf(B)=–1.

R category rotations: The following are the three types of R category rotations. They are explained along with representation and example.

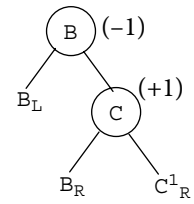
R0 rotation: General representation of R0 rotation is illustrated in Figure 12.20. Some node X is to be deleted from right subtree of C, i.e. C_R. After deleting it, the balance factor of C changed from +1 to +2. Node B the left subtree of C has a balance factor of 0. So, R0 rotation is to be used to rebalance the tree. It pushes B into the place of C along with its left subtree B_L and C is shifted down into the place of its right subtree along with the C^1_R and B_R as its left subtree, the earlier right subtree of B. Here C^1_R is the right subtree of C after deletion.



(a) Balanced AVL search tree before deletion

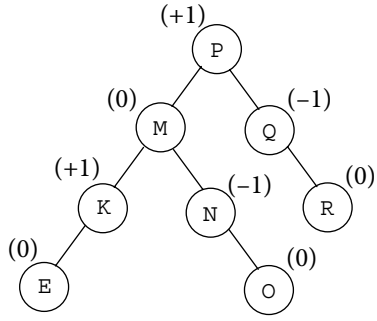


(b) Unbalanced AVL search tree after deletion

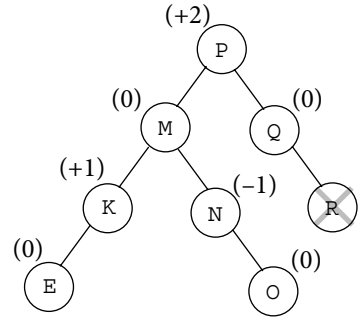


(c) Balanced AVL search tree after deletion

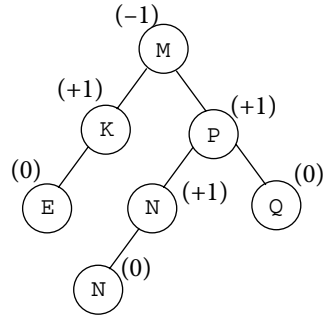
Figure 12.20 General representation of R0 rotation



(a) Balanced AVL search tree before deletion



(b) Unbalanced AVL search tree after deletion

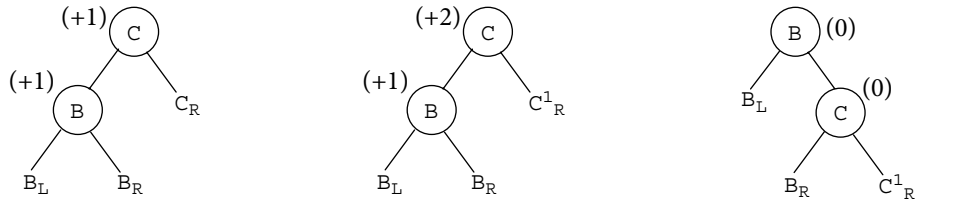


(c) Balanced AVL search tree after R0 rotation

Figure 12.21 Example of R0 rotation

Consider Figure 12.21 an instance of R0 rotation. From the balanced AVL search tree shown in Figure 12.21(a) the node R is deleted and bf(Q) is changed from 0 to 1. Also the bf(P) the ancestor node is also changed to +2 from +1 which calls R0 rotation as its left subtree, root M has the 0 balance factor. The unbalanced tree is shown in Figure 12.21(b). The balanced AVL search tree after R0 rotation and node M as the root is shown in Figure 12.21(c).

R1 rotation: The general representation of R1 rotation is depicted in Figure 12.22. Deleting a node from the right subtree C_R of an ancestor node C causes its balance factor to become +2. If the root of its left subtree has a balance factor as 1 then to rebalance the tree R1 rotation is to be invoked.

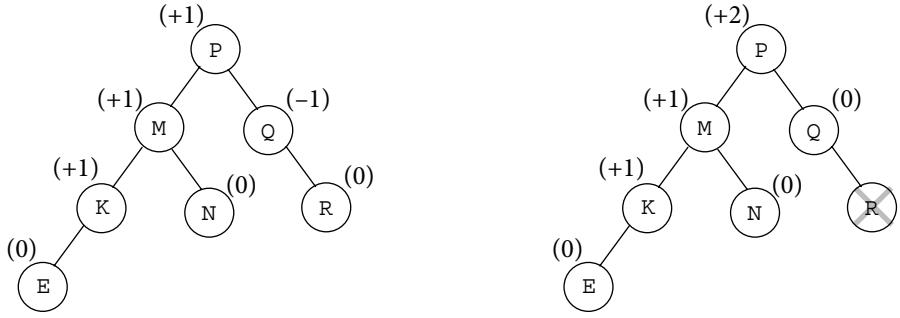


(a) Balanced AVL search tree before deletion

(b) Unbalanced AVL search tree after deletion

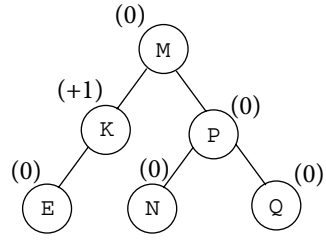
(c) Balanced AVL search tree after `R1` rotation

Figure 12.22 General representation of R1 rotation



(a) Balanced AVL search tree before deletion

(b) Unbalanced AVL search tree after deletion

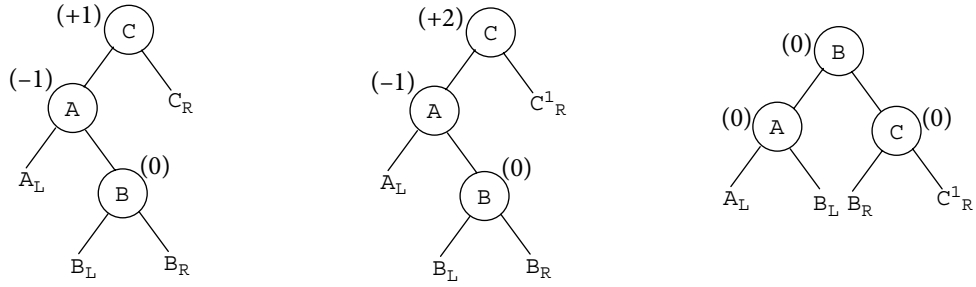


(c) Balanced AVL search tree after `R1` rotation

Figure 12.23 Example of R1 rotation

Consider the instance of R1 rotation depicted in Figure 12.23. After deleting node R, bf(Q) is updated to 0 and p to +2. Imbalance occurred and bf(M)=+1. So invoke R1 rotation. This balances the tree with M as the root node as depicted in Figure 12.23(c).

R-1 rotations: General representation of R-1 rotation is shown in Figure 12.24. The general representation of R-1 rotation is depicted in Figure 12.24. Deleting a node from the right subtree C_R of an ancestor node C causes its balance factor to become +2. If the root of its left subtree has a balance factor of –1 then to rebalance the tree R-1 rotation is to be invoked.

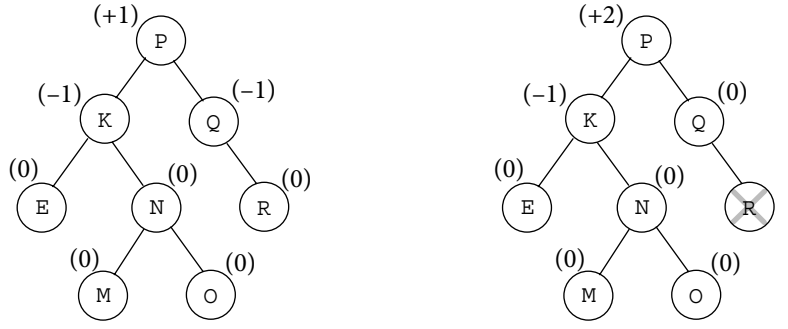


(a) Balanced AVL search tree before deletion

(b) Unbalanced AVL search tree after deletion

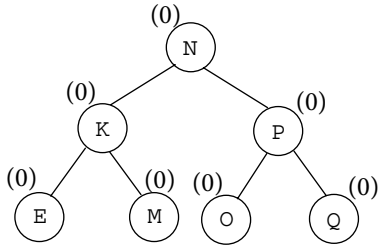
(c) Balanced AVL search tree after R-1 rotation

Figure 12.24 General representation of R–1 rotation



(a) Balanced AVL search tree before deletion

(b) Unbalanced AVL search tree after deletion



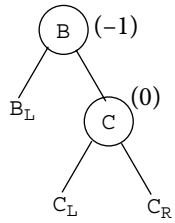
(c) Balanced AVL search tree after R-1 rotation

Figure 12.25 Example of R-1 rotation

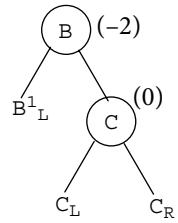
Consider Figure 12.25 instance of R-1 rotation. Here node R is marked for deletion. After deleting it, bf(Q) the parent node is updated to 0 and that of also its ancestor node P is bf(P) = +2. Now the tree is imbalanced as shown in Figure 12.25(b). As K the left subtree of P has its balance factor as –1 invoke R-1 rotation. After the rotation the AVL search tree is balanced with N as the root node as shown in Figure 12.25(c).

L category rotations: When a node is deleted from the left subtree of A, the first ancestor node in the path to the root in upward direction leads bf(A) to –2. Then to rebalance an unbalanced AVL search tree, L category rotations are used. With respect to the balance factor of the root of the right subtree of A these rotations can be further classified as L0 when its balance factor is 0, L1 when its balance factor is +1 and L–1 when its balance factor is –1. General representations of L0, L1 and L–1 rotations are depicted in Figure 12.26. Here B^1_L is the left subtree of B after deletion.

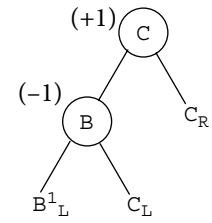
An AVL search tree may require more than one rotation to rebalance it after deletion, whereas insertion requires a single rotation.



(a) Balanced AVL search tree before deletion

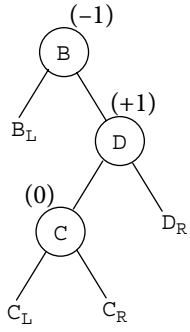


(b) Unbalanced AVL search tree after deletion

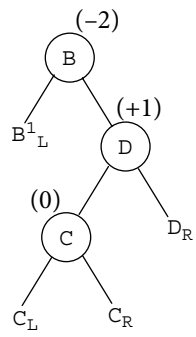


(c) Balanced AVL search tree after L0 rotation

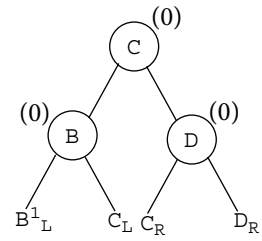
(i) L0 rotation



(a) Balanced AVL search tree before deletion

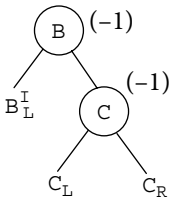


(b) Unbalanced AVL search tree after deletion

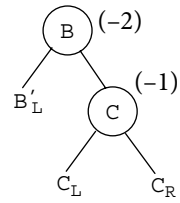


(c) Balanced AVL search tree after L1 rotation

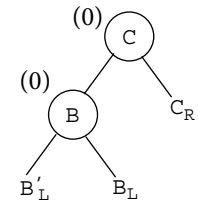
(ii) L1 rotation



(a) Balanced AVL search tree before deletion



(b) Unbalanced AVL search tree after deletion



(c) Balanced AVL search tree after L1 rotation

(iii) L-1 rotation

Figure 12.26 L Category rotations

Program 12.2 gives the implementation of operations on AVL search trees.

Program 12.2

```
/*AVL SEARCH TREE OPERATIONS
  1. INSERTION 2.DELETION 3.TRAVERSING 4.SEARCHING*/

#include<stdlib.h>
#include<stdlib.h>
#include<conio.h>
#include<iostream.h>
struct AVLTREE
{
 struct AVLTREE*lchild,*rchild,*auxchild;
 int height,data;
};
typedef struct AVLTREE tree;
tree*root=NULL,*ptr,*ptr1;
int s,k=0,item;
int findheight(tree*ptr)
{
 if(ptr==NULL)
  return -1;
 else
  return ptr->height;
}
int max(int item1,int item2)
{
   if(item1>item2)
      return item1;
   else
      return item2;
}
void adheight(tree*loc)
{
```

```
   while(loc!=NULL)
   {
      loc=loc->auxchild;
      loc->height=max(findheight(loc->lchild),findheight(loc->rchild))+1;
   }
}
void rcrotate(tree*ptr,tree*prev)
{
   tree*hold,*temp;
   temp=ptr->lchild;
   hold=temp->rchild;
   temp->rchild=ptr;
   ptr->lchild=hold;
   if(ptr==root)
      root=ptr;
   else if(ptr==prev->lchild)
      prev->lchild=temp;
   else
      prev->rchild=temp;
      ptr->height=max(findheight(ptr->lchild),findheight(ptr->rchild))+1;
   temp->height=max(findheight(temp->lchild),findheight(temp->rchild))+1;
   adheight(temp);
   temp->auxchild=prev;
   if(hold!=NULL)
      hold->auxchild=ptr;
   ptr->auxchild=temp;
   }
   void lcrotate(tree*ptr,tree*prev)
   {
      tree*hold,*temp;
      temp=ptr->rchild;
      hold=temp->lchild;
      temp->lchild=ptr;
      ptr->rchild=hold;
      if(ptr==root)
         root=temp;
      else if(ptr==prev->rchild)
         prev->rchild=temp;
      else
         prev->lchild=temp;
      ptr->height=max(findheight(ptr->lchild),findheight(ptr->rchild))+1;
      temp->height=max(findheight(temp->lchild),findheight(temp->rchild))+1;
      adheight(temp);
      temp->auxchild=prev;
      if(hold!=NULL)
         hold->auxchild=ptr;
      ptr->auxchild=temp;
   }
```

```
void inorder(tree*ptr)
   {
      if(ptr!=NULL)
      {
         inorder(ptr->lchild);
         cout<<ptr->data<<" ";
         inorder(ptr->rchild);
      }
   }
void checkbal(tree*ptr,int item)
   {
      int hlchild,hrchild,bal;
      while(ptr->auxchild!=NULL)
      {
         ptr=ptr->auxchild;
         hlchild=findheight(ptr->lchild);
         hrchild=findheight(ptr->rchild);
         bal=hlchild-hrchild;
         if(bal>1)
         {
            if(item>ptr->lchild->data)
            {
               lcrotate(ptr->lchild,ptr);
               rcrotate(ptr,ptr->auxchild);
            }
         else
         rcrotate(ptr,ptr->auxchild);
      }
      else if(bal<-1)
      {
         if(item<ptr->rchild->data)
         {
            rcrotate(ptr->rchild,ptr);
            lcrotate(ptr,ptr->auxchild);
         }
      else
      lcrotate(ptr,ptr->auxchild);
      }
   }
}
void case1()
{
   if(ptr1->lchild==ptr)
      ptr1->lchild=NULL;
   else ptr1->rchild=NULL;
      adheight(ptr1);
      checkbal(ptr1,item);
   cout<<"Element Is Successfully deleted";
```

```
}
void case2()
{
   if(ptr1->lchild==ptr)
   {
      if(ptr->lchild==NULL)
         ptr1->lchild=ptr->rchild;
      else
         ptr1->lchild=ptr->lchild;
   }
   else
   {
      if(ptr1->rchild==ptr){if(ptr->lchild==NULL)ptr1->rchild=ptr->rchild;
         else ptr1->rchild=ptr->lchild;
   }
}
   adheight(ptr1);
   checkbal(ptr1,item);
   cout<<"Element Is Successfully deleted";
}
void delet()
{
   int c=0;
   ptr=root;
   if(ptr==NULL)
   {
      cout<<"Tree Is Empty";
      return;
   }
   cout<<"Which Item You Want To delete";
   cin>>item;
      if(ptr->data==item)
      {
         cout<<item<<"Is A Root Node Deletion Is Not Possible";
         return;
      }
   if(ptr->data==item&&ptr->lchild==NULL&&ptr->rchild==NULL)
   {
      root=NULL;
      cout<<item<<"Is Successfully deleted";
      return;
   }
   while((ptr!=NULL)&&(c==0))
   {
      if(item<ptr->data)
      {
         ptr1=ptr;
         ptr=ptr->lchild;
```

```
      }
   else if(item>ptr->data)
   {
      ptr1=ptr;
      ptr=ptr->rchild;
   }
   else
   c=1;
}
if(c==0)
   {
      cout<<"Item Does Not Exist..,Deletion Is Not Possible";
      return;
   }
}
void delleaf()
{
   delet();
      if(ptr->lchild==NULL&&ptr->rchild==NULL)
   case1();
   else
      cout<<"Your Input Data Is Invlid";
}
void delone()
{
   delet();
      if((ptr->lchild!=NULL&&ptr->rchild==NULL)||(ptr->lchild==NULL&&ptr
         ->rchild!=NULL))
   case2();
   else
      cout<<"Your Input Data Is Invalid";
}
void traverse(struct AVLTREE*ptr)
{
   if(ptr!=NULL)
   {
      traverse(ptr->lchild);
      cout<<ptr->data<<" ";
      traverse(ptr->rchild);
   }
}
void insert()
{
   tree*loc,*prev;
   cout<<"Enter Item To Be Insert";
   cin>>item;
   ptr=(tree*)malloc(sizeof(tree));
   ptr->data=item;
```

```
   ptr->auxchild=ptr->lchild=ptr->rchild=NULL;
   ptr->height=0;
   if(root==NULL)
   {
      root=ptr;
      cout<<"Item"<<item<<"Is Successfully Inserted";
      return;
   }
 loc=root;
 while(loc!=NULL)
 {
   prev=loc;
   if(item<loc->data)
      loc=loc->lchild;
  else
      loc=loc->rchild;
 }
   ptr->auxchild=prev;
   if(item<prev->data)
      prev->lchild=ptr;
   else
      prev->rchild=ptr;
   adheight(ptr);
   checkbal(ptr,item);
   cout<<"Item"<<item<<"Is Successfully Inserted";
   }
int searching(tree *p,int key)
{
   if(p!=NULL)
   {
      if(p->data==key)
         k=1;
      else
      {
         searching(p->lchild,key);
         searching(p->rchild,key);
      }
   }
   return k;
}
void main()
{
   int n,height,op,x;
   clrscr();
   x:
   {
      while(1)
      {
```

```
        cout<<"\nAVLTREE Operations\n1.Insert\n2.Delete\n3.Travese\n";
        cout<<"4.Height Of Tree\n5.Searching\n6.Exit";
        cout<<"\nEnter Your Option";
        cin>>n;
        switch(n)
        {
           case 1:insert();break;
           case 2:
           if(root==NULL){cout<<"Treee Is Empty";break;}
           cout<<"\n1.Delete Leaf Node\n2.Delete Single Child Node";
           cout<<"\nEnter Your Option";
           cin>>op;
           switch(op)
           {
              case 1:delleaf();break;
              case 2:delone();break;
              case 3:goto x;
                 default:cout<<"Invalid Choice";
           }
        break;
           case 3:cout<<"Tree Elemnts Are:";inorder(root);break;
           case 4:height=findheight(root);
              cout<<"Height Of The Tree:"<<height;break;
           case 5:if(root==NULL){cout<<"Tree Is Empty";break;}
              s=k=0;cout<<"Which Item You Want To Search";
              cin>>item;
              s=searching(root,item);
      if(s==1)
        cout<<"Item"<<item<<"Is Found";
      else
        cout<<"Item"<<item<<"Is Not Found";
      break;
      case 6:exit(0);
      default:cout<<"Invalid Choice";
      }
   }
}
}
```

Output

```
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
```

```
Enter Your Option1
Enter Item To Be Insert35
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option1
Enter Item To Be Insert20
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option1
Enter Item To Be Insert40
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option1
Enter Item To Be Insert10
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option1
Enter Item To Be Insert22
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option1
Enter Item To Be Insert37
AVLTREE Operations
```

```
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option1
Enter Item To Be Insert15
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option1
Enter Item To Be Insert25
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option1
Enter Item To Be Insert36
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option3
Tree Elemnts Are:10 15 20 22 25 35 36 37 40
1.Delete Leaf Node
2.Delete Single Child Node
Enter Your Option1
Which Item You Want To delete15
Element Is Successfully deleted
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option3
Tree Elemnts Are:10 20 22 25 35 36 37 40
```

```
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option2
1.Delete Leaf Node
2.Delete Single Child Node
Enter Your Option2
Which Item You Want To delete22
Element Is Successfully deleted
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option3
Tree Elemnts Are:10 20 25 35 36 37 40
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option4
Height Of The Tree:3
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option5
Which Item You Want To Search37
Item 37 Is Found
AVLTREE Operations
1.Insert
2.Delete
3.Travese
4.Height Of Tree
5.Searching
6.Exit
Enter Your Option6
```

12.6 APPLICATIONS

A self-balancing binary search tree has numerous applications. Its basic characteristic is to maintain the elements in order. So, it is applied in all the environment where the order of elements is required:

1. Priority queues
2. Associative array based on the order of elements the key–value pairs are inserted
3. Binary tree sort
4. Many algorithms in computational geometry such as line segment, intersection problem and point location problem
5. Internet routers

SUMMARY

- A non-empty binary search tree should satisfy the following properties:
  1. All the elements must have a key and it must be distinct.
  2. All the keys (if any) in the left subtree of the tree.
  3. All the keys (if any) in the right subtree of the tree must be greater than the root element.
  4. The left and right subtrees of the tree must also be the binary search trees.
- The inorder traversal of a binary search tree produces the sorted list of the given elements in ascending order.
- The operations on binary search tree are searching, insertion and deletion.
- BST is highly efficient in performing all the operations with a time complexity of O(log n) over a sequential list.
- Binary search trees suffers from the drawback of becoming skewed.
- An empty binary tree is an AVL tree. A non-empty binary tree T is an AVL tree if and only if
  1. the left subtree T_L and the right subtree T_R of T are also AVL trees and
  2. $|h(T_L) - h(T_R)| \leq 1$ where $h(T_L)$ and $h(T_R)$ are the heights of left and right subtrees of T, respectively.
- The balancing factor of a node u, bf(u) can be defined as $bf(u)=h(u_L)-h(u_R)$, where, $h(u_L)$ is the height of the left subtree of u and $h(u_R)$ is the height of the right subtree of u.
- Representation of AVL tree and AVL search trees are very similar to that of binary trees and binary search trees using a linked representation.
- Searching, insertion and deletion are the common operations on AVL trees.
- The insertion operation on AVL trees is similar to BST but leads to imbalance. The rotations LL, LR, RL, RR are used to rebalance the tree.
- A deletion operation on AVL trees also performs some rotations.

EXERCISES

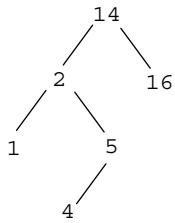
Fill in the Blanks

1. The path from the root to an external node is known as an _____.
2. Linked representation of a binary search tree with a collection of internal nodes that represent the keys and external nodes that represent null pointers is known as _____.
3. The skewed binary search trees have the time complexity of _____.

4. Major classification of rotations while performing deletion in AVL trees is _____ .
5. _____ are the rotations performed while inserting an element into an AVL tree.

Multiple-choice Questions

1. _____ the time complexity in an average case where frequent insertions and/or deletions are assumed.
   a. O(log n) b. O(n)
   c. O(nlog n) d. None
2. Which of the following traversal technique lists the nodes of a binary tree in ascending order?
   a. Postorder b. Inorder
   c. Preorder d. None
3. Consider this binary search tree:

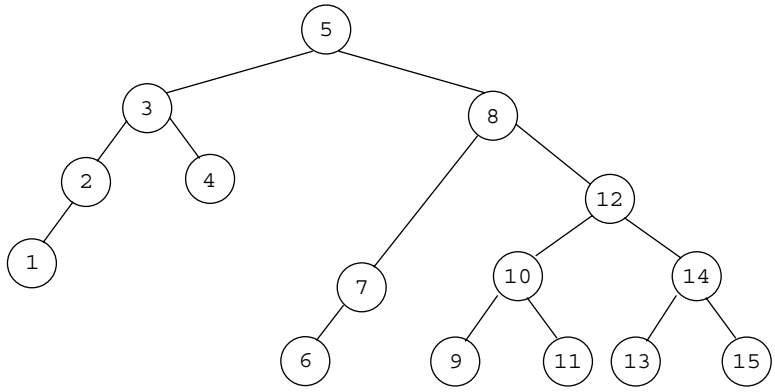


   Suppose we remove the root, replacing it with something from the left subtree, what will be the new root?
   a. 1 b. 2 c. 4 d. 5 e. 16
4. The balancing factor of a node u, bf(u) can be defined as _____.
   a. bf(u)-h(u_L)-h(u_R) b. bf(u)-h(u_L)-h(u_R)
   c. bf(u)+h(u_L)-h(u_R) d. bf(u)+h(u_L)+h(u_R)
5. Which of the following statement is true?
   a. An AVL search tree may require one rotation to rebalance it after deletion.
   b. An AVL search tree may require more than one rotation, insertion requires single rotation.
   c. An AVL search tree may require more than one rotation to rebalance it after deletion
   d. None.

Short-answer Questions

1. Define binary search trees.
2. Define AVL trees.
3. Suppose that we want to create a binary search tree where each node contains information on some data type called Item (which has a default constructor and a correct value semantics), what additional factor is required for the Item data type?
4. Suppose that a binary search tree contains the number 42 at a node with two children. Write two or three clear sentences to describe the process required to delete 42 from the tree.

5. Consider the following AVL tree:



Show the modified tree under each of the following operations:
Deletion of the key 4
Insertion of the key 16.

Essay Questions

1. Explain about binary search trees and its operations in detail.
2. Discuss in detail about AVL trees.
3. Give the representations of various rotations involved while inserting an element into an AVL tree.
4. Write an algorithm for performing deletion in AVL trees.
5. Explain various rotations involved while deleting an element from an AVL tree.

Chapter 13

Multiway Trees and B Trees

This chapter is dedicated to the discussion of multiway trees and B trees. The structure of a node in m-way trees and operations on them are explained in detail. Stating the drawbacks of m-way search trees, the need for B trees is explained along with its definition. Regarding B trees operation performed on them and its height are discussed. This also includes variations of B trees and database as an application of B trees.

13.1 INTRODUCTION

The tree data structures in which a node has more than two branches are termed as *multiway trees*. Data structures m-way search trees and B trees come under this category. In some applications like Database very large index entries may be found. Such index can be maintained as m-way search trees than AVL search trees which are just balanced binary search trees. A binary search tree is a two-way search tree. An m-way search tree is an extended binary search tree and supports more efficient retrievals. B trees are the height balanced versions of m-way search trees.

Definition: An m-way search tree T may be an empty tree. If T is a non-empty m-way search tree then the following properties can be observed from it:

1. Every node can have a maximum of m subtrees, where m is some integer and known as the order of the tree.
2. A node with k subtrees will have $k-1$ elements where $k<m$.
3. All the elements of a node will be in ascending order when a node with k subtrees is considered then $k_1, k_2, k_3, \ldots, k_{k-1}$ are the elements of the node in the order $k_1<k_2<k_3<\ldots,<k_{k-1}$. The subtrees will be $C_0, C_1, C_2 \ldots, C_{K-1}$.
4. The subtree with root C_0 will have the elements that are less than k_1 and that of C_{k-1} will have elements that are greater than k_{K-1}. The subtree with root C_i will have the elements that are greater than k_i and less than k_{i+1}; $1 \le i \le k$.
5. All the subtrees are once again m-way trees.

13.2 REPRESENTATION OF A NODE STRUCTURE

An m-way search tree is an extended tree where the null pointers are represented by external nodes. In defining an m-way search tree and while discussing its operations external nodes are very helpful. Similar to the case of a tree external nodes are not physically represented but are only fictitious.

The general node structure in an m-way search tree is shown in Figure 13.1. The node is with *m–1* elements and *m* child pointers pointing to the roots of *m* subtrees. Pointers to empty subtrees are shown by external nodes represented with filled circles as shown in Figure 13.2.

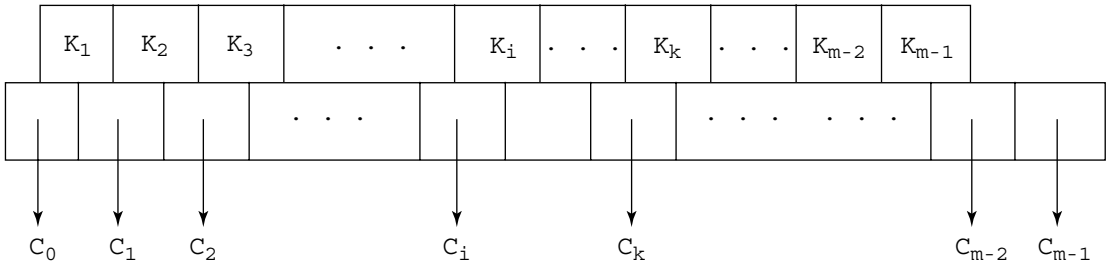


Figure 13.1 Node structure in an m-way search tree

A 4-way search tree is depicted in Figure 13.2. Nodes have a maximum of 4 child nodes out of which some are external nodes.

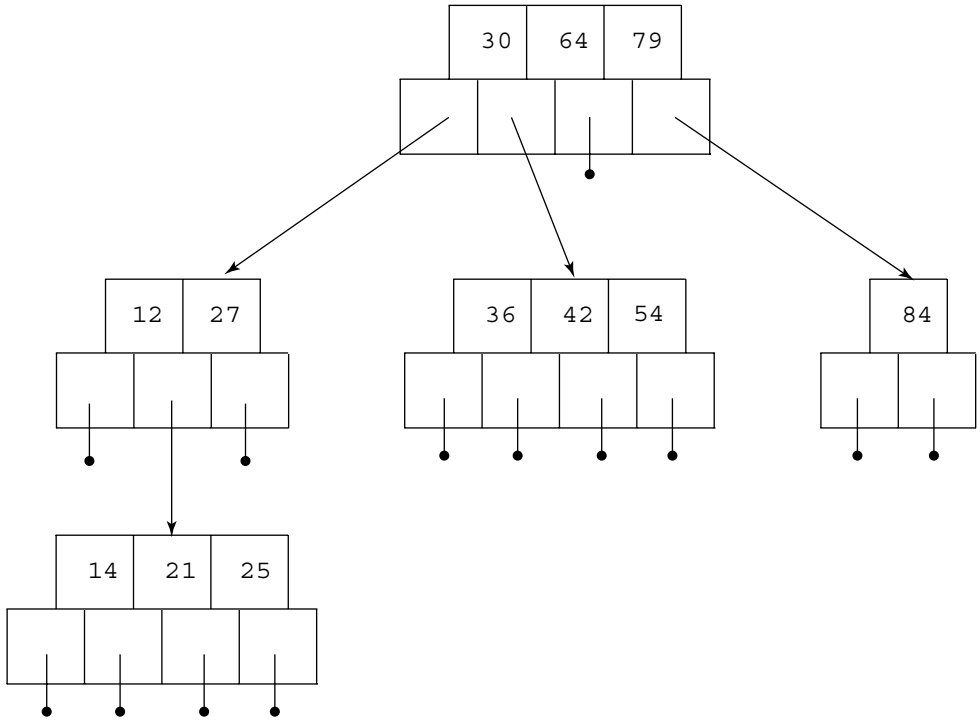


Figure 13.2 A 4-way search tree

The root node is with three elements [30,64,79] and four subtrees. The first subtree is with the elements that are less than 30, the second subtree has the elements that are greater than 30 and less than 64. The third subtree that should contain the elements that are greater than 64 and less than 79 is empty. The fourth subtree is with the element 84 that is greater than 79. The same procedure is followed for all nodes in the subtrees.

13.3 OPERATIONS ON m-WAY SEARCH TREES

Binary search tree is an m-way search tree of order two. Operations on m-way search tree are the extensions of operations on binary search trees. Some of the very frequently used operations are discussed.

13.3.1 Searching

Searching begins with a root node. The element k for which the search is carried out is compared with each element of the root node sequentially. If the required element matches with any of the root elements then the search is successful. If $k>k_i$ and $k<k_{i+1}$ for some value of i then the search continues with the respective subtree T_i. Similarly, the search proceeds till the required element is found and the search is successful or an external node or null pointer is found where the search is unsuccessful.

The 4-way search tree depicted in Figure 13.2 is considered. Searching for an element 15 compares it with the first element of the root 30. As 15 is less than (<) 30 search moves to the first subtree of the root [12,27]. As 15 is greater than (>) 12, the first element of the node, it again moves down to the subtree with the elements [14,21,25]. As 15>14 and 15<21 search moves to the next subtree which is an external node. So, searching is terminated as an unsuccessful search. From the same tree if 42 is the element that is to be searched then 42 is compared with the root node [30,64,79]. As 42>30 and 42<64 search moves to the second subtree [36,42,54]. Now, 42 is compared with 36 which does not match, proceeds to 42 the next element of the node where match is found. So, the search is successful.

13.3.2 Insertion

Inserting an element into the m-way search tree begins with the search operation. The place where the search operation terminates is where the insertion takes place. The new element may be inserted into an existing node if it is able to accommodate, otherwise it may be inserted as a new node in the next level.

In the 4-way search tree from Figure 13.2 if an element 29 is to be inserted then search for 29 falls off at the node [12,27]. This node is able to accommodate one more element. So, insert 29 as the last element in the node then the node is [12,27,29]. The corresponding pointer field can also be observed.

If another element 50 is to be inserted then the search falls off at the third external node of the node [36,42,54]. As the node is full of its capacity it cannot accommodate 50 in it. So, the new node with the element 50 and its pointer field appears as the third child of [36,42,54]. This is illustrated in Figure 13.3.

13.3.3 Deletion

Deleting an element from an m-way search tree also follows the search operation. When an element k with its left subtree pointer C_l and its right subtree pointer C_r is to be deleted it can be done in any of the following four cases.

Case 1: $C_l = C_r$ = Nil, i.e. if the left and right subtrees of k are Nil then k is deleted and the number of pointer fields of the node is adjusted.

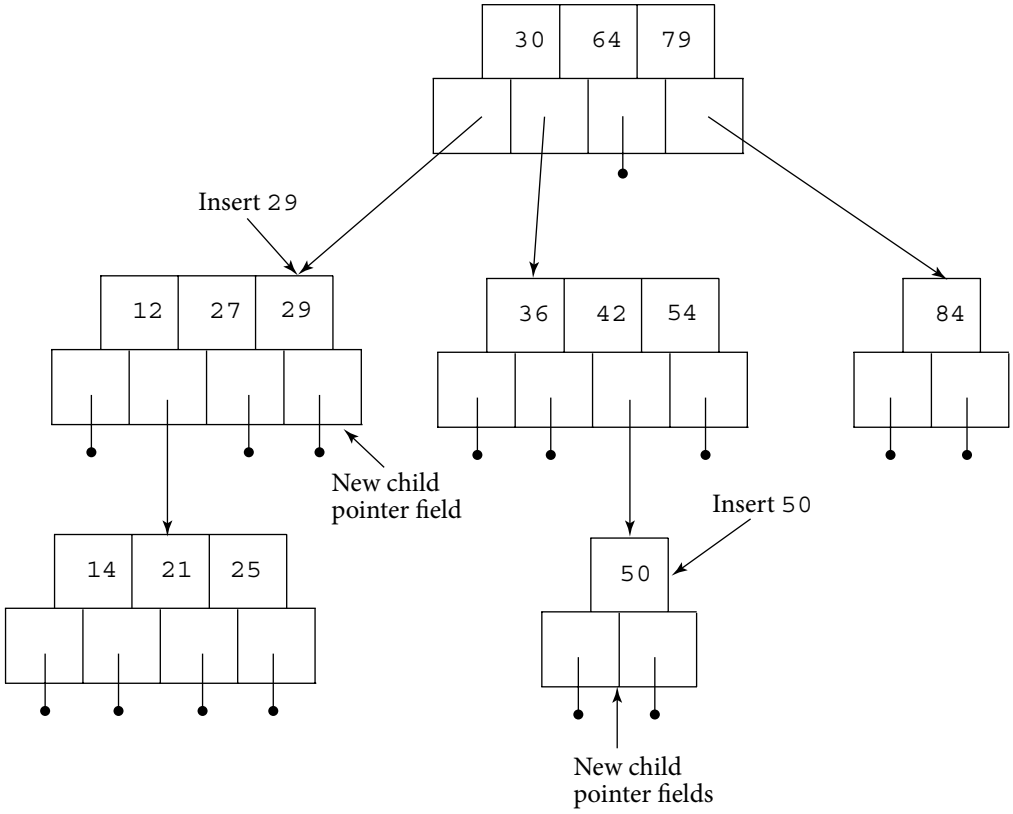


Figure 13.3 Insertion of elements 29 and 50 into 4-way search tree in Figure 11.2

Case 2: $C_1 \neq$ Nil and $C_r =$ Nil, i.e. if the left subtree is not empty and the right subtree is empty then select the largest element from the left subtree and k is replaced with it. This will recursively invoke the deletion of the largest element by following one or more of the four cases.
Case 3: $C_1 =$ Nil and $C_1 \neq$ Nil, i.e. if the left subtree of k is empty and the right subtree is not empty. Select the smallest element from the right subtree and replace k with it. This will recursively invoke the deletion of the smallest element by following one or more of the four cases.
Case 4: $C_1 \neq$ Nil and $C_r \neq$ Nil, i.e. if both left and right subtrees of k are not empty. Select either the largest element from the left subtree or the smallest element from the right subtree and replace k with it. This follows the appropriate steps to delete the replaced element from the tree.

Consider the 4-way search tree shown in Figure 13.2. To delete element 14 which does not have both left and right subtrees, Case 1 can be applied. So, simply 14 and its corresponding pointer are deleted from the node. Deletion of 27 which has the left subtree but no right subtree follows Case 2. From the left subtree the largest element 25 is replaced with 27. Case 1 is applied again in deleting 25. To delete 30 which has both left and right subtrees, Case 4 is applied. Either the largest element 27 from the left subtree or the smallest element 36 from the right subtree may be selected to replace 30. Here 36 is replaced with 30. Case 1 is applied again to delete 36 from the tree. These instances may be shown in Figure 13.4.

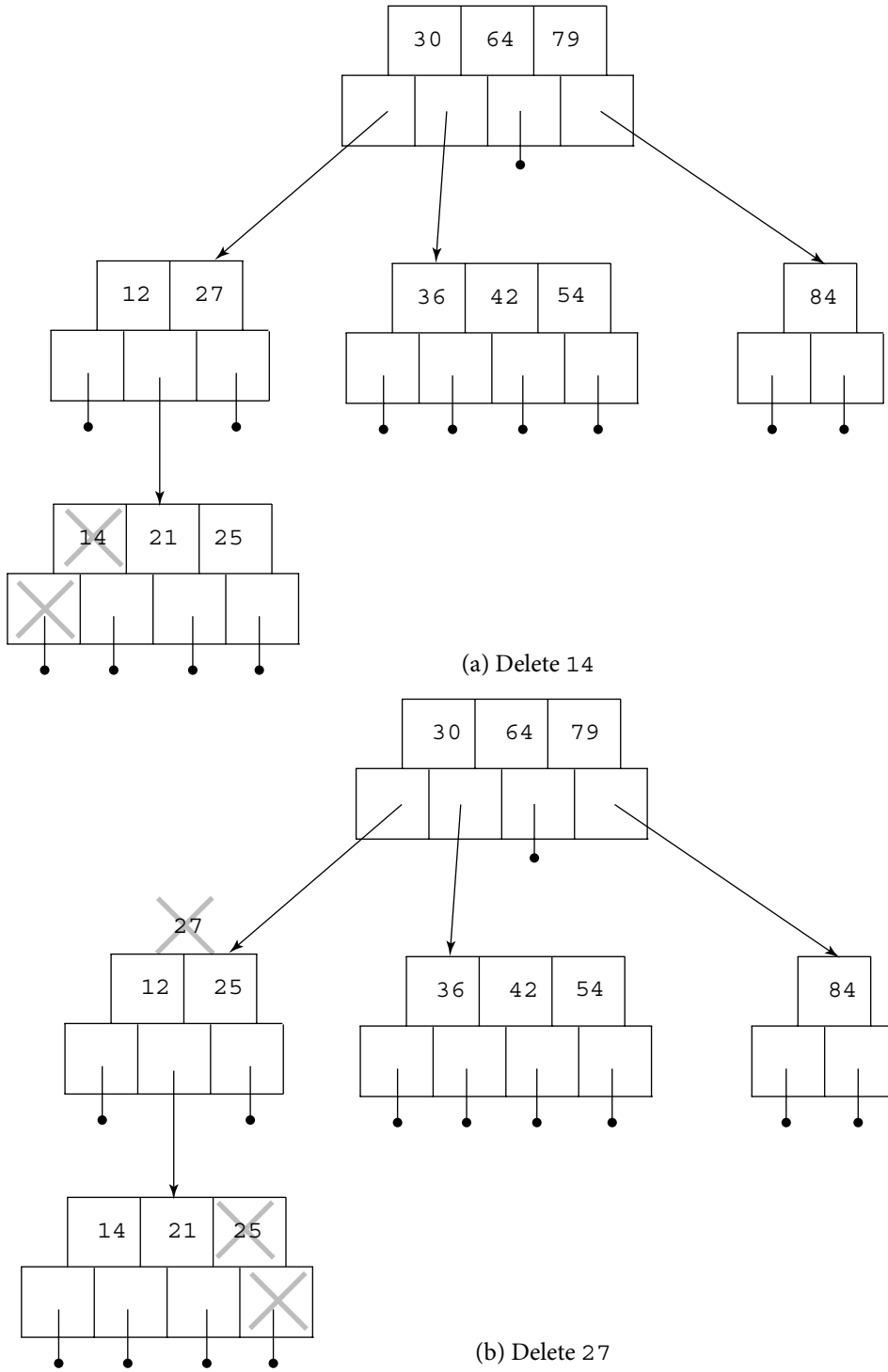
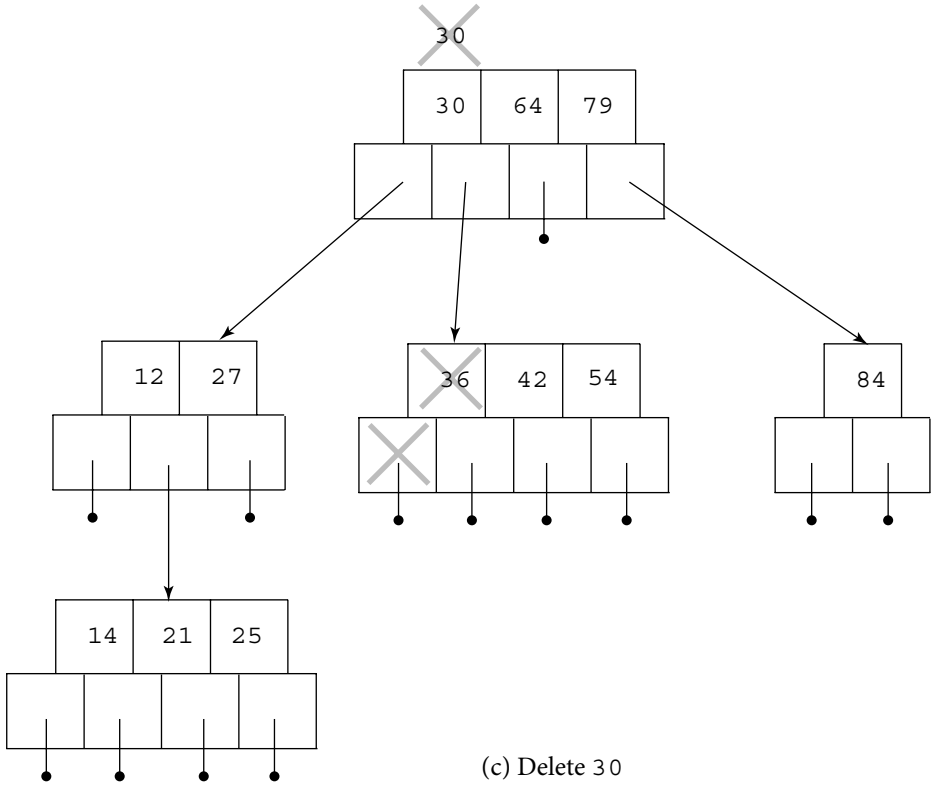


Figure 13.4 Deletion of 14, 27 and 30 individually from 4-way search tree shown in Figure 13.2



(c) Delete 30

Figure 13.4 *Continued*

13.3.4 Drawbacks of m-Way Search Trees

The search, insert and delete operations on an m-way search tree of height h(without external nodes) have the complexity as O(h). An m-way search tree of height h will have a minimum of h and a maximum of m^h-1 number of elements. Minimum of h is as one element per node at every level and maximum as at every level; each node has m child nodes and m–1 elements. This says that the maximum number of nodes at level i is given by m^{i-1}. As in an m-way search tree of height h, the total number of nodes is

$$\sum_{i=1}^{h} m^{i-1} = \frac{(m^h - 1)}{(m-1)}$$

and each of these nodes will have m–1 elements, the maximum number of elements is

$$\frac{(m^h - 1)}{(m-1)}(m-1) = (m^h - 1)$$

Because the number of elements in an m-way search tree with a height h will be from h to m^h–1, when a tree with n elements is considered its height will be a minimum of $\log_m(n+1)$ and a maximum of n. In the worst case the height of an m-way search tree with n elements may be O(n) which yields a poor performance. So, it is required to maintain the balanced heights in m-way search trees.

13.4 B TREES

A data structure which is a height balanced version of m-way search tree is known as a B tree of order m. When the growth of an m-way search tree is left uncontrolled then in the worst case it yields a complexity of O(n). This shows deterioration in performance. So, there is a need to have balanced m-way search trees which guarantees a complexity of O(log n) for search, insert and delete operations.

Definition: A B tree of order m is an m-way search tree and it may be empty. If not empty then the following properties are to be satisfied by the extended trees:

(i) The root node should have a minimum of two children and a maximum of m children.
(ii) All the internal nodes except the root node should have a minimum of $\lceil m/2 \rceil$ non-empty children and a maximum of m non-empty children.
(iii) All the external nodes are at the same level.
(iv) A leaf node must have minimum $\lceil m/2-1 \rceil$ and maximum m–1 elements.

The 4-way search tree shown in Figure 13.2 is not a B tree because all the external nodes are not at the same level. All the internal nodes except the root must be with a minimum $\lceil 4/2 \rceil = 2$ children. But, the node [12,27] is a non-root internal node with only one child.

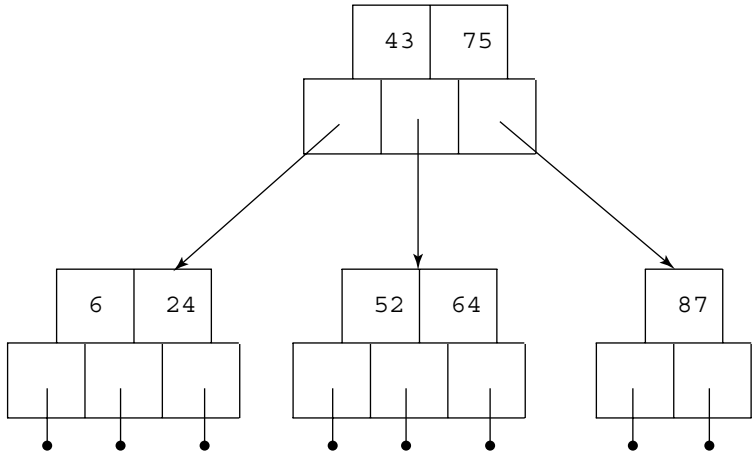


Figure 13.5 B tree of order 3 (2–3 trees)

B trees with order 3 are also referred to as 2–3 trees as their internal nodes can have only two or three children. Figure 13.5 shows a B tree of order 3. B trees of order 4 are also referred to as 2–3–4 or 2–4 trees.

A B tree of order 5 is shown in Figure 13.6. In this tree the root [40,82] has minimum two children. All the internal nodes except the root have minimum $\lceil m/2 \rceil = \lceil 5/2 \rceil = 3$ children and hence two elements in corresponding nodes. All the external nodes are at the same level. As all the properties are satisfied, it is a B tree of order 5.

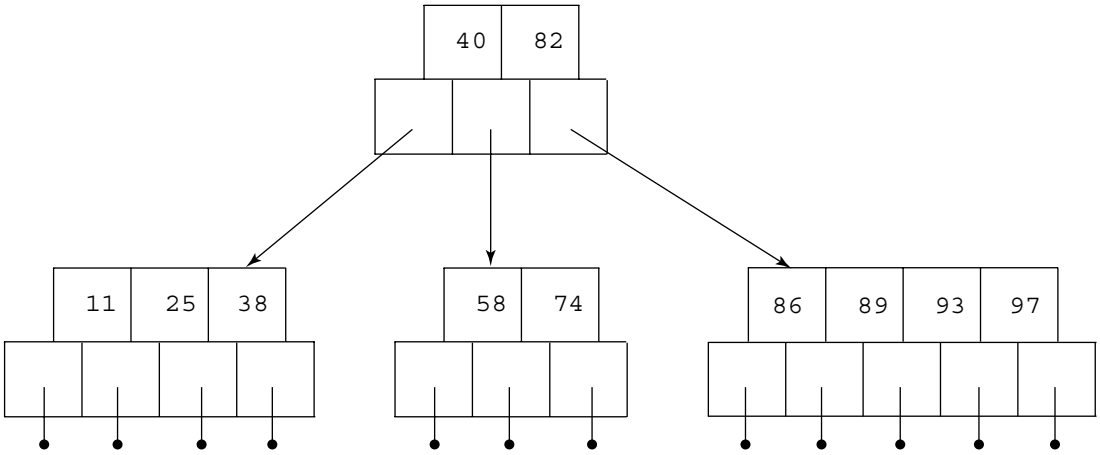


Figure 13.6 B tree of order 5

13.5 OPERATIONS ON B TREES

B tree grows in a bottom up manner. Various operations can be performed on B trees of order `m` but the most frequently used are search, insert and delete operations.

13.5.1 Searching

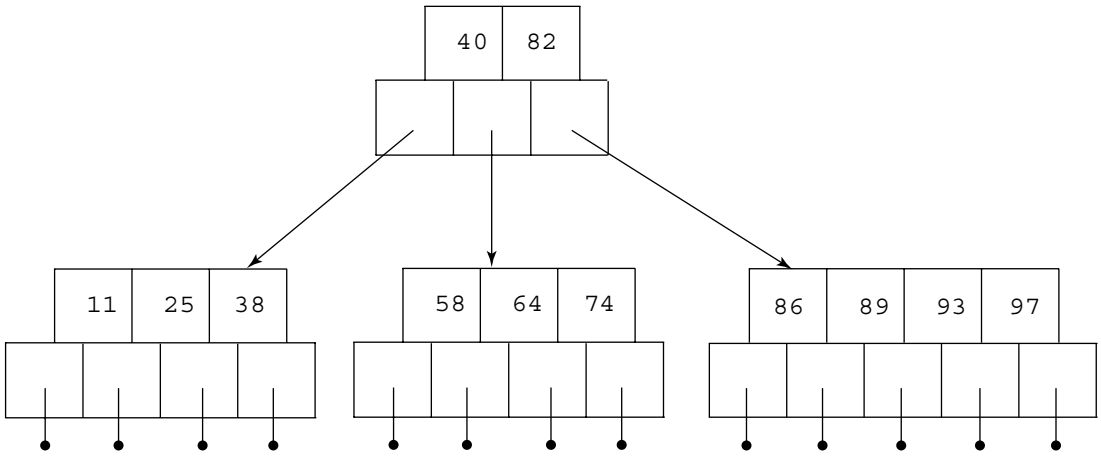
The search operation on a B tree of order `m` is exactly the same as that performed on m-way search trees. If `h` is the height of the B tree of order `m` then the search operation yields a complexity of O(`h`).

13.5.2 Insertion

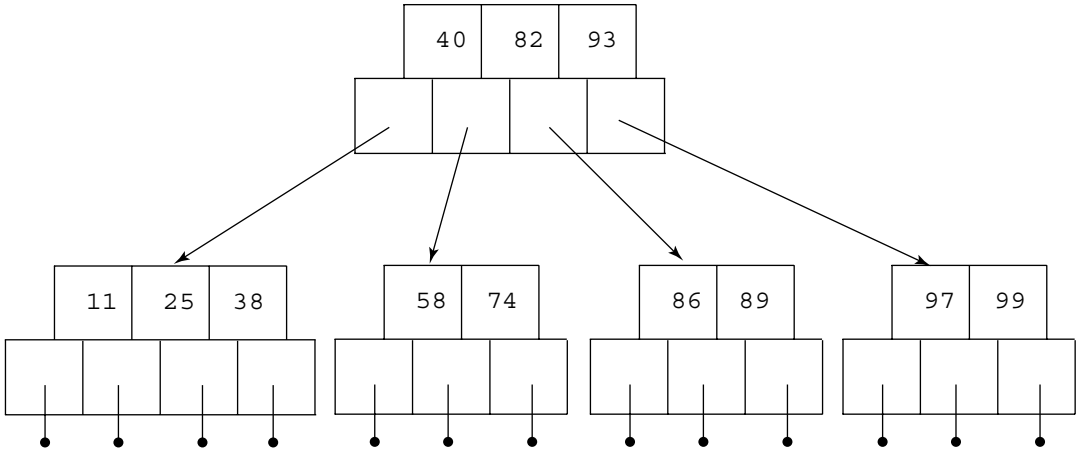
Inserting an element into a B tree of order m is followed by the search operation for the same element. The place where search falls off, the element is inserted by following either of the cases.
Case 1: In a B tree of order `m`, if node `X` in which the element is to be inserted can accommodate the element then it is inserted in the same node and the number of child pointer fields are adjusted accordingly.
Case 2: In a B tree of order `m` if a node `X` in which the element is to be inserted is full, then first insert the element into its list of elements. This list is split into two at the median. The elements that are less than the median becomes the left node and that are greater than the median becomes the right node. The median element is shifted up into the parent node of `X`. Accommodating the median element into the parent node may once again invoke either of the insertion cases.

Insertion of element 64 into the B tree of order 5 shown in Figure 13.6 initiates search operation for 64. This falls off at node [58, 74]. As it is not full the node is able to accommodate 64 and the node now appears as [58, 64, 74]. This is illustrated in Figure 13.7(a).

To insert element 99 into the B tree the search operation for 99 falls off at node [86, 89, 93, 97]. This node is full and cannot accommodate 99. So, insert 99 into the list of elements, the list appears as 86, 89, 93, 97, 99. Here the median element is 93. So, split the node at the median into two as [86, 89] and [97, 99]. Now the median element 93 is shifted up into the parent, i.e. root. Again Case 1 is invoked to insert 93 into the root. This is illustrated in Figure 13.7(b).



(a) Insert 64



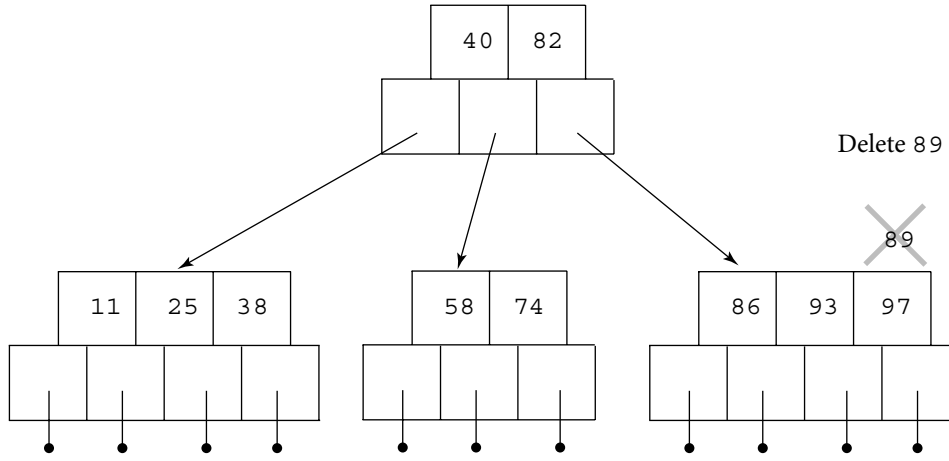
(b) Insert 99

Figure 13.7 Inserting 64 and 99 individually into the B tree of order 5 shown in Figure 13.6

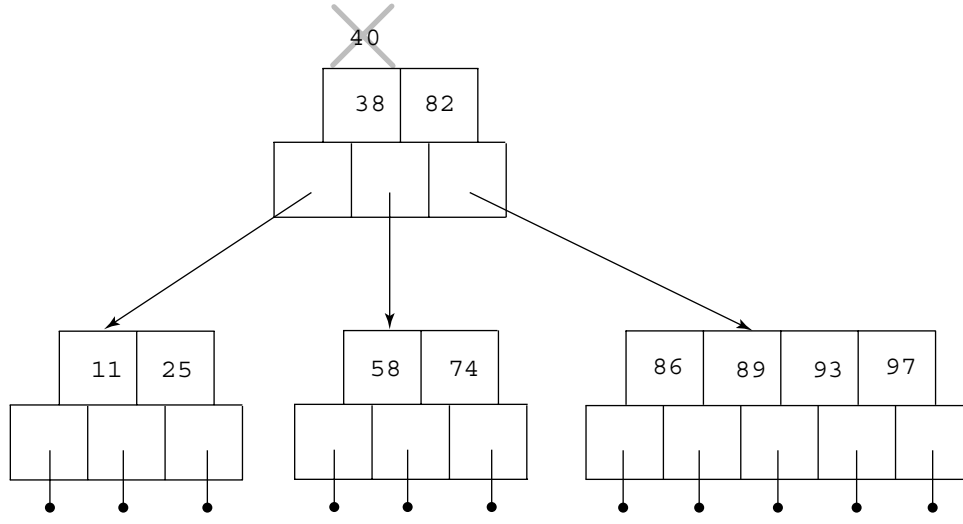
13.5.3 Deletion

Deleting an element from a B Tree of an order m may be performed according to one of the four cases.
Case 1: This deletion is the simplest of all the cases. An element belongs to a leaf node and deleting it does not make the node with the number of elements less than its minimum number of elements. In this case the element is deleted from the leaf node and the child pointers are adjusted accordingly.
Case 2: When an element k from a non-leaf node is to be deleted, replace k with the largest element from its left subtree or the smallest element from its right subtree. The replaced element must be deleted from the node and as it happens to be from a leaf node, Case 1 is followed to delete it.

The B tree of order 5 shown in Figure 13.6 is considered. To delete 89 from the tree as it is a leaf node element, Case 1 can be followed. Delete 89 and adjust its child pointer fields. To delete 40, a non-leaf node element, Case 2 can be followed. Replace 40 with 38 the largest element from its left subtree. Deleting 38 again follows Case 1. This is illustrated in Figure 13.8



(a) After deleting 89



(b) Delete 40

Figure 13.8 Deletion of 89 and 40 individually from B tree of order 5 shown in Figure 13.6

If the deletion of an element leaves less than the minimum number of elements in the corresponding nodes, then deletion becomes complicated. Cases 3 and 4 deal with such cases.

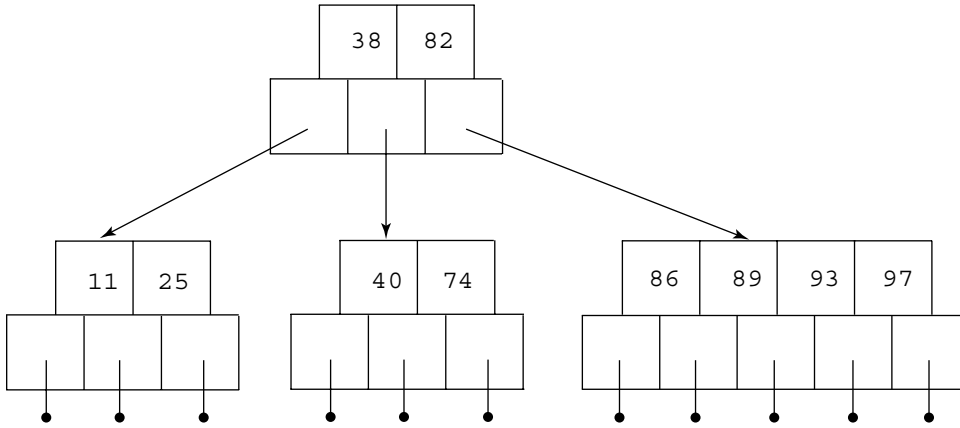
Case 3: If deleting an element k from a node leaves it with less than its minimum number of elements, then elements can be borrowed from either of its sibling nodes. When the left subtree node is capable to spare the

element then its largest element is shifted into the parent node. If the right subtree node is capable to do so, its smallest element is shifted into the parent node. From the parent node the intervening element is shifted down to fill the vacancy created by the deleted element.

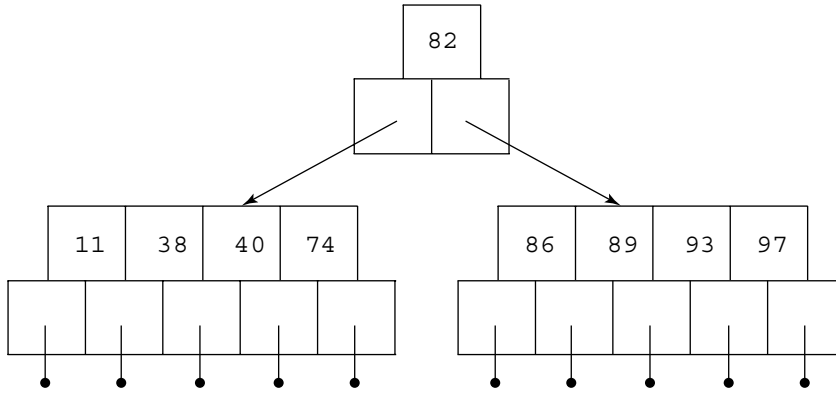
Case 4: In a situation where the deletion of an element is making the elements of the node to be less than its minimum number and if either or both of its sibling nodes have no chance of sparing an element, then this node is merged with either of the sibling nodes including the intervening element from the parent node. This once again invokes one of the cases to delete the intervening element from the parent node.

Consider the B tree of order 5 shown in Figure 13.6. Deleting 58 leaves the node with one element, i.e. less than the minimum number of elements. From its left sibling node 38 can be borrowed. So, 38 replaces the intervening parent node 40 and is pushed down to fill the vacancy created by deleting 58. Figure 13.9(a) depicts this situation. This in turn calls Case 1 to delete 38 from the leaf node. This is an illustration of Case 3.

To illustrate Case 4 consider the B tree of order 5 shown in Figure 13.9(a), the resulting tree after deleting 58. Deletion of 25 makes the nodes with elements less than the minimum number of elements. Its sibling right subtree is unable to spare the element. Now, merge 11 with [40, 74] along with the intervening element from the parent node, i.e. 38. So, the new node is [11, 38, 40, 74]. This is depicted in Figure 13.9(b).



(a) After deleting 58



(b) After deleting 25

Figure 13.9 Delete 58 and 25 from B tree of order 5 shown in Figure 13.6

Program 13.1

```
//Implementation of  BTREE Operations
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 50
struct BTREE
{
   int n;
   int keys[MAX-1];
   struct BTREE*p[MAX];
}*root=NULL;
typedef struct BTREE node1;
int count=0,ele;
enum KeyStatus{Duplicate,SearchNodeFailure,Success,InsertIt,LessKeys};
enum KeyStatus ins(node1*r,int x,int*y,node1**u);
enum KeyStatus del(node1*r,int x);
void InsertNode()
{
   node1*newnode;
   int upKey;
   enum KeyStatus value;
   cout<<"Enter Element To Insert";
   cin>>ele;
   value=ins(root,ele,&upKey,&newnode);
   if(value==Duplicate)
   {
      cout<<"Element Already Exists\n";
      return;
   }
   if(value==InsertIt)
  {
      node1*uproot=root;
      root=(node1*)malloc(sizeof(node1));
      root->n=1;
      root->keys[0]=upKey;
      root->p[0]=uproot;
      root->p[1]=newnode;
   }
   count++;
   cout<<"Element Is Successfully Inserted";
}
int SearchNodePos(int key,int*key_arr,int n)
{
   int pos=0;
   while(pos<n&&key>key_arr[pos])
   pos++;
   return pos;
}
```

```
enum KeyStatus ins(node1*ptr,int key,int*upKey,node1**newnode)
{
   node1*newPtr,*lastPtr;
   int pos,i,n,splitPos;
   int newKey,lastKey;
   enum KeyStatus value;
   if(ptr==NULL)
   {
      *newnode=NULL;
      *upKey=key;
      return InsertIt;
   }
 n=ptr->n;
 pos=SearchNodePos(key,ptr->keys,n);
 if(pos<n&&key==ptr->keys[pos])
 return Duplicate;
 value=ins(ptr->p[pos],key,&newKey,&newPtr);
 if(value!=InsertIt)
 return value;
 if(n<MAX-1)
   {
      pos=SearchNodePos(newKey,ptr->keys,n);
      for(i=n;i>pos;i--)
      {
         ptr->keys[i]=ptr->keys[i-1];
         ptr->p[i+1]=ptr->p[i];
      }
   ptr->keys[pos]=newKey;
   ptr->p[pos+1]=newPtr;
   ++ptr->n;
   return Success;
   }
   if(pos==MAX-1)
   {
      lastKey=newKey;
      lastPtr=newPtr;
   }
   else
   {
      lastKey=ptr->keys[MAX-2];
      lastPtr=ptr->p[MAX-1];
      for(i=MAX-2;i>pos;i--)
      {
         ptr->keys[i]=ptr->keys[i-1];
         ptr->p[i+1]=ptr->p[i];
      }
      ptr->keys[pos]=newKey;
      ptr->p[pos+1]=newPtr;
 }
```

```
 splitPos=(MAX-1)/2;
 (*upKey)=ptr->keys[splitPos];
 (*newnode)=(node1*)malloc(sizeof(node1));
 ptr->n=splitPos;
 (*newnode)->n=MAX-1-splitPos;
 for(i=0;i<(*newnode)->n;i++)
 {
   (*newnode)->p[i]=ptr->p[i+splitPos+1];
   if(i<(*newnode)->n-1)
      (*newnode)->keys[i]=ptr->keys[i+splitPos+1];
   else
      (*newnode)->keys[i]=lastKey;
   }
   (*newnode)->p[(*newnode)->n]=lastPtr;
   return InsertIt;
}
void Display(node1 *ptr,int blanks)
{
   if(count==0)
   {
      cout<<"BTREE Is Empty";
      return;
   }
   if(ptr)
   {
      int i;
      for(i=0;i<ptr->n;i++)
      cout<<ptr->keys[i]<<" ";
      cout<<endl;
      for(i=0;i<=ptr->n;i++)
      Display(ptr->p[i],blanks+10);
   }
}
void search()
{
   int pos,i,n;
   node1*ptr=root;
   if(count==0)
   {
      cout<<"BTREE Is Empty";
      return;
   }
 cout<<"Enter Element To Search Node";
 cin>>ele;
 while(ptr)
 {
  n=ptr->n;
  pos=SearchNodePos(ele,ptr->keys,n);
```

```
  if(pos<n&&ele==ptr->keys[pos])
  {
   cout<<"Element"<<ele<<"Is Found";
   return;
  }
  ptr=ptr->p[pos];
 }
 cout<<"Element"<<ele<<"Is Not Found";
}
void DeleteNode()
{
 int flag=1;
 node1*uproot;
 enum KeyStatus value;
 if(count==0)
 {
  cout<<"BTREE Is Empty";
  return;
 }
 cout<<"Enter Element To Delete";
 cin>>ele;
 value=del(root,ele);
 switch(value)
 {
   case SearchNodeFailure:cout<<"Element"<<ele<<"Is Not Available";
      flag=0; break;
   case LessKeys:uproot=root;root=root->p[0];free(uproot);break;
 }
 if(flag==1)
 {
  cout<<"Element"<<ele<<"Is Deleted";
  count--;
 }
}
enum KeyStatus del(node1*ptr,int key)
{
 int pos,i,pivot,n,MAXin;
 int*key_arr;
 enum KeyStatus value;
 node1**p,*lchild,*rchild;
 if(ptr==NULL)
 return SearchNodeFailure;
 n=ptr->n;
 key_arr=ptr->keys;
 p=ptr->p;
 MAXin=(MAX-1)/2;
 pos=SearchNodePos(key,key_arr,n);
 if(p[0]==NULL)
```

```
{
 if(pos==n||key<key_arr[pos])
 return SearchNodeFailure;
 for(i=pos+1;i<n;i++)
 {
  key_arr[i-1]=key_arr[i];
  p[i]=p[i+1];
 }
 return --ptr->n>=(ptr==root?1:MAXin)?Success:LessKeys;
}
if(pos<n&&key==key_arr[pos])
{
 node1*qp=p[pos],*qp1;
 int nkey;
 while(1)
 {
  nkey=qp->n;
  qp1=qp->p[nkey];
  if(qp1==NULL) break;
  qp=qp1;
 }
 key_arr[pos]=qp->keys[nkey-1];
 qp->keys[nkey-1]=key;
}
value=del(p[pos],key);
if(value!=LessKeys)
return value;
if(pos>0&&p[pos-1]->n>MAXin)
{
 pivot=pos-1;
 lchild=p[pivot];
 rchild=p[pos];
 rchild->p[rchild->n+1]=rchild->p[rchild->n];
 for(i=rchild->n;i>0;i--)
 {
  rchild->keys[i]=rchild->keys[i-1];
  rchild->p[i]=rchild->p[i-1];
 }
 rchild->n++;
 rchild->keys[0]=key_arr[pivot];
 rchild->p[0]=lchild->p[lchild->n];
 key_arr[pivot]=lchild->keys[--lchild->n];
 return Success;
}
if(pos>MAXin)
{
 pivot=pos;
 lchild=p[pivot];
```

```
 rchild=p[pivot+1];
 lchild->keys[lchild->n]=key_arr[pivot];
 lchild->p[lchild->n+1]=rchild->p[0];
 key_arr[pivot]=rchild->keys[0];
 lchild->n++;
 rchild->n--;
 for(i=0;i<rchild->n;i++)
 {
  rchild->keys[i]=rchild->keys[i+1];
  rchild->p[i]=rchild->p[i+1];
 }
 rchild->p[rchild->n]=rchild->p[rchild->n+1];
 return Success;
}
if(pos==n)
 pivot=pos-1;
else
 pivot=pos;
lchild=p[pivot];
rchild=p[pivot+1];
lchild->keys[lchild->n]=key_arr[pivot];
lchild->p[lchild->n+1]=rchild->p[0];
for(i=0;i<rchild->n;i++)
{
 lchild->keys[lchild->n+1+i]=rchild->keys[i];
 lchild->p[lchild->n+2+i]=rchild->p[i+1];
}
lchild->n=lchild->n+rchild->n+1;
free(rchild);
for(i=pos+1;i<n;i++)
{
 key_arr[i-1]=key_arr[i];
 p[i]=p[i+1];
}
return--ptr->n>=(ptr==root?1:MAXin)?Success:LessKeys;
}
int main()
{
 int op;
 clrscr();
 while(1)
 {
  cout<<"\nBTREE Opeartions\n1.Insertion\n2.Deletion\n3.Searching\n";
  cout<<"4.Traversing\n5.Exit\n";
  cout<<"Enter Your Option:";
  cin>>op;
  switch(op)
  {
```

```
    case 1:
      InsertNode();
      break;
    case 2:
      DeleteNode();
      break;
    case 3:
      search();
      break;
    case 4:
      cout<<"BTREE Elements Are:\n";
      Display(root,0);
      break;
    case 5:
      exit(0);
    default:
       cout<<"Invalid Choice\n";
   }
  }
}
```

Output

```
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:1
Enter Element To Insert14
Element Is Successfully Inserted
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:1
Enter Element To Insert20
Element Is Successfully Inserted
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:1
Enter Element To Insert6
Element Is Successfully Inserted
```

```
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:1
Enter Element To Insert25
Element Is Successfully Inserted
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:1
Enter Element To Insert3
Element Is Successfully Inserted
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:1
Enter Element To Insert30
Element Is Successfully Inserted
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:4
BTREE Elements Are:
3 6 14 20 25 30
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:2
Enter Element To Delete20
Element 20 Is Deleted
BTREE Opeartions
1.Insertion
2.Deletion
```

```
3.Searching
4.Traversing
5.Exit
Enter Your Option:4
BTREE Elements Are:
3 6 14 25 30
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:1
Enter Element To Insert10
Element Is Successfully Inserted
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:4
BTREE Elements Are:
3 6 10 14 25 30
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:3
Enter Element To Search Node25
Element 25 Is Found
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:3
Enter Element To Search Node15
Element 15 Is Not Found
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
```

```
Enter Your Option:1
Enter Element To Insert10
Element Already Exists
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:1
Enter Element To Insert20
Element Is Successfully Inserted
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:4
BTREE Elements Are:
3 6 10 14 20 25 30
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:2
Enter Element To Delete10
Element 10 Is Deleted
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:4
BTREE Elements Are:
3 6 14 20 25 30
BTREE Opeartions
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter Your Option:5
```

13.6 HEIGHT OF B TREES

Let T be a B tree of order m and height h with n elements. Since a B tree with order m is an m-way tree, then m satisfies $n \le m^h - 1$. Now the upper bound of n is known, then what is its lower bound? That is what is the minimum number of elements that a B tree of order m and height h can hold. First find the minimum number of nodes in levels 1, 2, 3. . . (h+1), where h+1 is the level at which external nodes exist, let t= [m/2]. The minimum number of nodes on the levels 1, 2, 3 . . ., (h+1) is given by 1, 2, 2t, $2t^2$, . . ., $2t^{(h-1)}$, so the lower bound of n is given by $n \ge 2t^{(h-1)}$.

Since the number of external nodes is 1 more than the number of inernal nodes, the lower bound of n is $n \ge 2t^{(h-1)} - 1$. Hence, the lower and upper bound of n is $2t^{(h-1)} - 1 \le n \le m^h - 1$. From this it can be represented in logarithms very easily as $\log_m(n+1) \le h \le \log t((n+1)/2) + 1$ which determines the best and worst case complexities of operations on B trees, which is given by O(h), the height of the B tree.

13.7 VARIATIONS OF B TREE

B* tree and B+ tree are the two popular variations of the B trees.

13.7.1 B* Tree

In a B tree with huge number of elements, space requirements may become excessive. This is because nearly half of the elements will be empty. The first variation, B* tree, discusses the space usage of large trees. Instead of each node containing a minimum of half of the maximum entries, the minimum is reduced to two-thirds.

In case of node overflows, instead of immediately splitting, the data are redistributed among the siblings of the node. This delays the creation of a new node. Splitting occurs only in the case when all of its siblings are full. When the nodes are split, data from the two full siblings are distributed among the two full nodes and the new node resulting in two-thirds full in all the three nodes. Figure 13.10 shows the redistribution when 99 is inserted into the B tree of order 5 shown in Figure 13.6. Its counterpart B tree with splitting is shown in Figure 13.7(b).

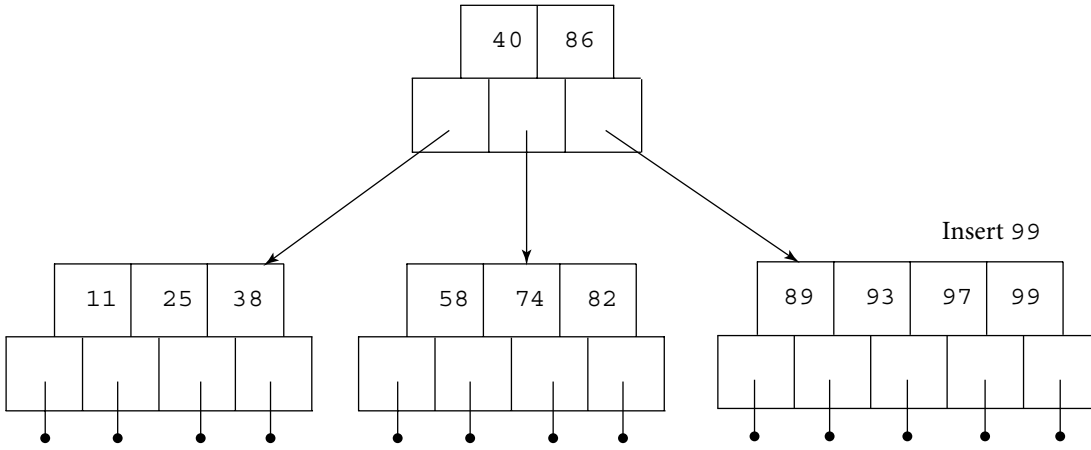


Figure 13.10 Redistribution in B* tree

13.7.2 B+ Tree

In large file systems, both random and sequential processing of data are required. For such systems most of the popular file organization methods prefer and use the B tree to process the data randomly. But, when the data are to be processed sequentially the B tree takes much processing time in moving up and down the tree structure. This inefficiency leads to the second variant of the B tree, B+ tree.

B+ tree differs from B tree in two points. They are:

1. Though elements are found in the internal nodes their corresponding data entry must be represented at the leaf level. As the internal nodes are used only for searching, they do not hold data.
2. Every leaf node is with an additional pointer and is used to proceed to the next leaf node in sequence.

A B+ tree is illustrated in Figure 13.11.

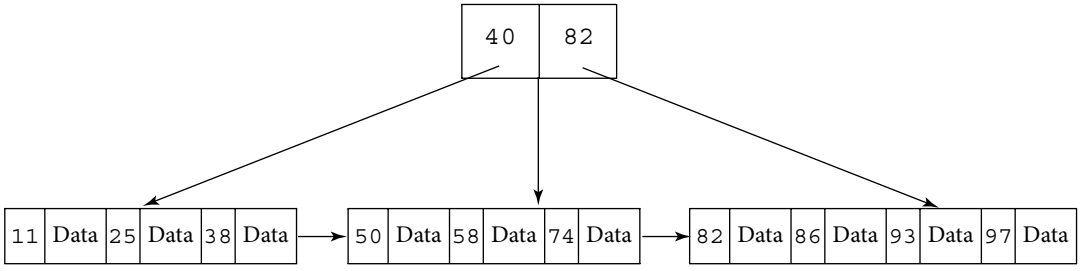


Figure 13.11 B^+ tree

To find the required data, search is performed only at the leaf level when the data are processed randomly. Thus, the search time is slightly increased. To process the data sequentially the far left entry is identified and then proceed to process the data similar to the way of processing a linked list in which every node is an array.

13.8 APPLICATIONS

Various data structures along with search trees like binary search trees and AVL search trees support search and retrievals on small amounts of data that can be stored in the internal memory of the computer. The large amount of data that are to be stored in external storage devices must also have efficient techniques for storage and retrievals. Efficient retrievals are based on indexes. Though indexes are just lookup tables, they are required to be represented using efficient data structures to provide the essential services in an efficient manner. B trees provide solutions for this.

13.8.1 Databases

A database is a collection of data organized in an order and allows updating, retrieving and managing the data. Data are not restricted to a particular type. Database may contain voluminous data with millions of records that require many gigabytes of storage. A database to be useful and usable should allow one to perform some desired operations such as storage and retrieval at a faster rate. These databases cannot be stored entirely in internal memory. To index the data and for faster access B trees are useful. The worst case runtime to search an unsorted and unindexed database with n elements is O(n). It will be O(log n) when the same data are indexed using a B tree. Indexing huge amounts of data significantly increases search performance. B trees ideally suit to store indexes, its internal node stores <key, address> pair. During retrieval fewer accesses are called because of

its balanced heights. A B tree along with indexing also optimizes costly disk accesses which are to be considered when dealing with large data sets.

SUMMARY

- A tree whose nodes have more than two branches are termed as *multiway* trees.
- Binary search tree is a two-way search tree. An m-way search tree is an extended binary search tree.
- B trees are the height balanced versions of m-way search trees.
- Operations on an m-way search tree are the extensions of operations on binary search trees.
- The search, insert and deletion operations on an m-way search tree of height h have the complexity as O(h). An m-way search tree of height h will have a minimum of h and a maximum of m^h–1.
- B* tree and B+ tree are the two popular variations of the B trees.
- B* tree discusses the space usage of large trees. Instead of each node containing a minimum of half of the maximum entries, the minimum is reduced to two-thirds.
- In B+ tree though elements are found in the internal nodes their corresponding data entry is found at the leaf level. As the internal nodes are used only for searching, they do not hold data.
- In B+ tree every leaf node is with an additional pointer and is used to proceed to the next leaf node in sequence.

EXERCISES

Fill in the Blanks

1. Nodes with k subtrees will have _____ elements.
2. The search, insert and delete operations on an m-way search tree of height h have the complexity as _____.
3. A leaf node in B tree must have minimum _____ and maximum _____ elements.
4. The lower and upper bound of n in B tree of order n is _____ .
5. _____ and _____ are the two popular variations of the B trees.

Multiple-choice Questions

1. A B tree with order m is an m-way tree, then n satisfies _____.
   a. $n \leq m^h - 1$ b. $n \leq m^h$
   c. $n \leq m^{h+1}$ d. $n \leq m^{h-1}$
2. The worst case runtime to search an unsorted and unindexed database with n elements is _____.
   a. O(n log n) b. O(log n)
   c. O(n). d. None
3. An m-way search tree of height h will have a minimum of _____ and a maximum of _____ elements.
   a. h, m^h+1 b. h+1, m^h-1
   c. h, m^h-1 d. h–1, m^h-1
4. The maximum number of nodes in m-way trees at a level i is given by _____.
   a. m^i+1 b. m^{i-1}
   c. m^{i+1} d. m^i-1

5. The height of an m-way search tree with n elements may be _____.
   a. O(n–1)
   b. O(n)
   c. O(n+1)
   d. None

Short-answer Questions

1. Define m-way trees.
2. Explain the representation of node in an m-way tree.
3. Explain various operations on m-way trees.
4. Define B trees and explain the operations on it.
5. Define different types of B trees.

Essay Questions

1. Discuss in detail about m-way trees and give the algorithms for their operations.
2. Write an algorithm for insertion and deletion operations on B trees.
3. Write notes on the height of B trees.
4. Explain various forms of B trees.
5. Discuss the applications of B trees.

Chapter 14

Red–Black Trees and Splay Trees

This chapter introduces the data structures, red–black trees and splay trees. It describes how red–black trees exist with the reason behind them along with its definition and representation. Most popular operations on red–black trees are exemplified. Splay trees, splay rotations and amortized analysis with respect to splay trees are also explained in the chapter. This chapter also includes the applications of red–black trees and splay trees.

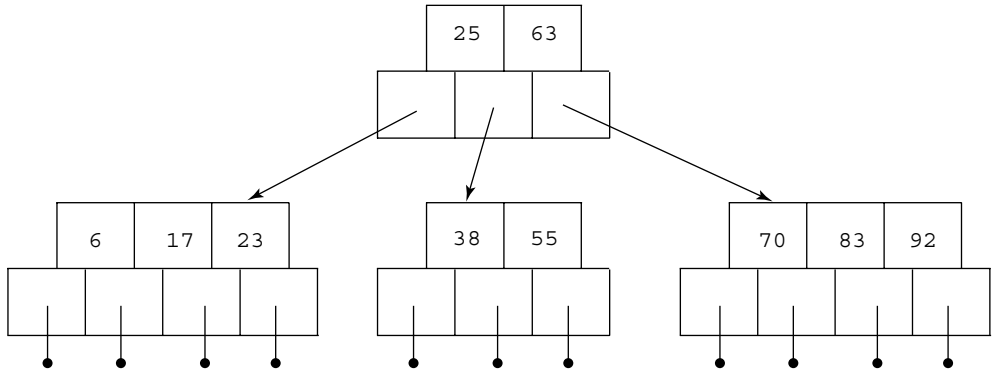
14.1 INTRODUCTION

B trees of order m discussed in the earlier chapter have a number of applications as they minimize disk accesses. To implement its node, sequential data structures like arrays may be invoked. So, in a B-tree of order m every node requires two arrays of maximum size $m-1$ and m for elements and child pointers, respectively. This leads to the wastage of space in the worst case. The elements of each of the nodes in the B tree can be maintained as a binary search tree and this would be an effective solution when the order of the B tree is small. But, branches that link the nodes of binary search tree must be different from the branches that link the nodes of a B tree. This yields a special type of binary search tree called red–black tree. A 2–4 tree shown in Figure 14.1(a) is considered. Each node is represented as a binary search tree as shown in Figure 14.1(b). Here, the thin lines represent branches that link the nodes of binary search tree, whereas the thick lines show the same in between the nodes of the original B-tree. The resulting red–black tree shown in Figure 14.1(c) is formed when a light shade (red) is given to the thin lines and nodes hanging from them and grey shade (black) is given to the thick lines and nodes hanging from them. The root node is always shaded black.

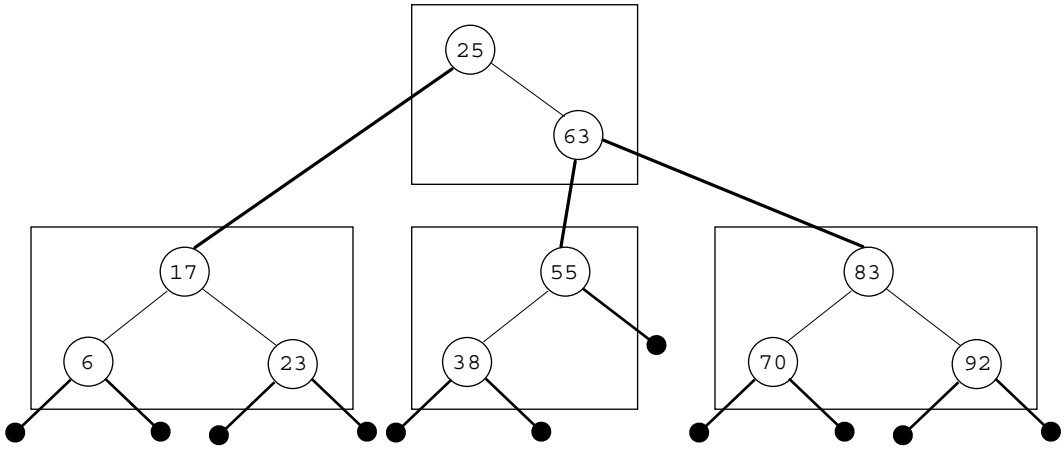
Definition: A red–black tree is an extended binary search tree in which all the nodes and edges are coloured either red or black. It should satisfy the following properties:

1. The root node and all the external nodes are always coloured black.
2. In every root node to an external node path no two consecutive red nodes should appear.
3. The number of black nodes in all root node to the external node path must be same.

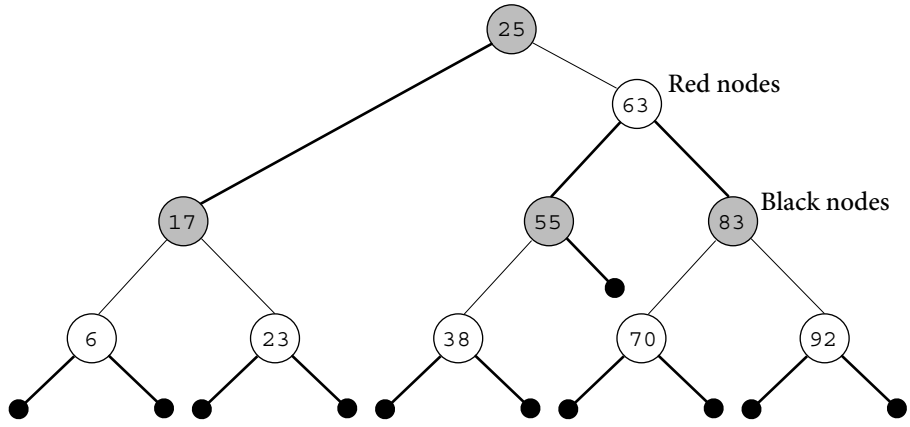
From a parent, a black child is connected with the black edge and a red child is connected with the red edge. Based on the colours of the pointers or edges the above definition can have the following additional properties:



(a) A 2–4 tree



(b) Binary search tree representation of the nodes of the 2–4 tree



(c) Red–black tree

Figure 14.1 Red–black tree from a 2–4 tree

1. The edges that connect an internal node and an external node must be black.
2. In every root node to an external node path no two consecutive red edges or pointers (or red nodes) should appear.
3. The number of black edges or pointers (or black nodes) in all root node to the external node path must be same.

On the path from a node to an external node, the number of black edges gives the rank of a node. Therefore, the rank of all external nodes is 0.

All through the chapter, red nodes are represented by normal circles, red edges are represented by thin lines, black nodes are represented by shaded circles and black edges are represented by thick lines.

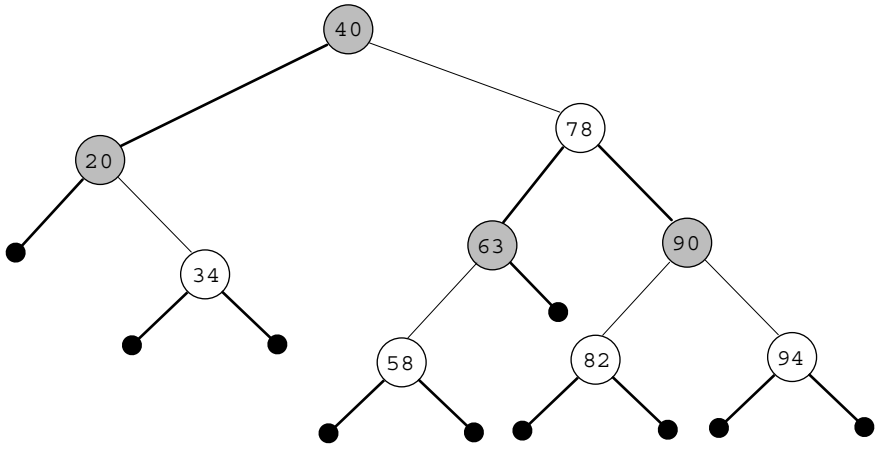


Figure 14.2 A red–black tree

The red–black tree shown in Figure 14.2 has the root node and all the external nodes are with black colour. In any root node to the external node path no two consecutive red nodes exist. It has three black nodes and two black pointers in every root node to external node path.

14.2 REPRESENTATION OF A RED–BLACK TREE

The node representation of a red–black tree is the same as that of a binary search tree because it is an extended binary search tree. As colour plays a vital role it is necessary to record the colour of either the node or two pointers that emerge from the node. Based on the method that is followed either one field or two fields for colour are added to the node structure.

The insert or delete operations may lead to the imbalance of the red–black tree. In rebalancing the tree, nodes may move up and down. In order to facilitate this movement, a field that holds a pointer to its parent node should be incorporated in the node structure of a red–black tree.

14.3 OPERATIONS

Among the numerous operations, the most frequently used search, insert and delete operations are explained.

14.3.1 Searching

The process of searching for an element in a red–black tree is exactly the same as an element that is searched in a binary search tree. So, Algorithm 12.1 serves the purpose of search operation in a red–black tree.

14.3.2 Insertion

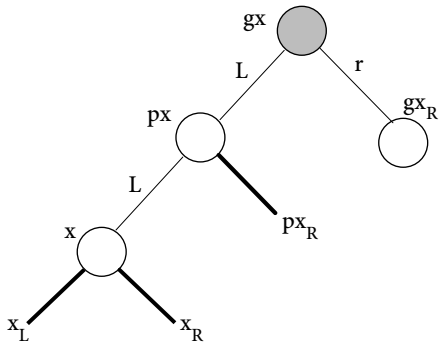
Insertion of an element into a red–black tree is the same as that of into a binary search tree. To the new node either black or red colour must be set. Prior to the insertion if the tree was empty then black colour may be set to the new node as it becomes the root. If the tree is non-empty and the new node is set to black then it violates the third property of the definition as an additional black node appears in the path from a root node to an external node. If the new node is set to red colour it may violate the second property of the definition. This leads to the imbalance of the tree and needs to be rebalanced. So, it is obvious to set the new node with red colour because this may or may not violate the definition. But, if the new node is set to black then it definitely violates the definition.

When it is a case of imbalance, along with the new node x its parent node px must be red in colour, the two consecutive red nodes. Node px cannot be the root so, the grandparent of x, gx must exist and must be black in colour. The imbalance may be classified based on the position of the new node x with respect to its parent (px) and grandparent (gx), and colour of the other child of the grandparent (gx).

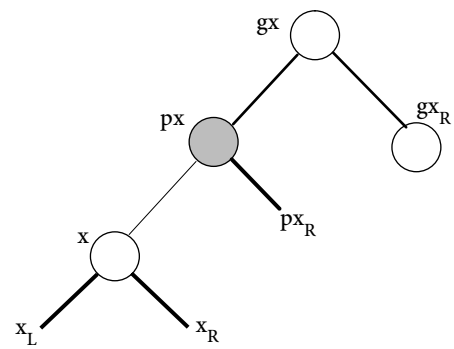
If the new node x is inserted as a left child of its parent px, which is again a left child of the grandparent of x, gx and colour of other child of the grandparent gx is black then it is an LLb type imbalance. If x is left child of px, px is a left child of gx and the other child of gx is red then it is an LLr type imbalance. If x is a right child of px, px is a left child of gx and the other child of gx is red then it is an LRr type imbalance. If x is a right child of px, px is a left child of gx and another child of gx is black then it is an LRb type imbalance. Similarly other imbalances are RLb, RLr, RRb and RRr.

The tree with imbalances, which have red as the colour of other children of the grandparent gx, LLr, RLr, LRr and RRr are rebalanced by changing colours of nodes. The other type imbalances, which have black as the colour of other children of the grandparent gx, LLb, RLb, LRb and RRb of the tree are rebalanced by rotation mechanism.

LLr, LRr, RRr and RLr imbalances: The general representation of LLr, LRr, RRr and RLr imbalances, the changes in node colour to rebalance the tree are illustrated in Figure 14.3. The classification of imbalance is depicted by L, R and r notations on the edges of the red–black trees.



LLr imbalance



After LLr colour change

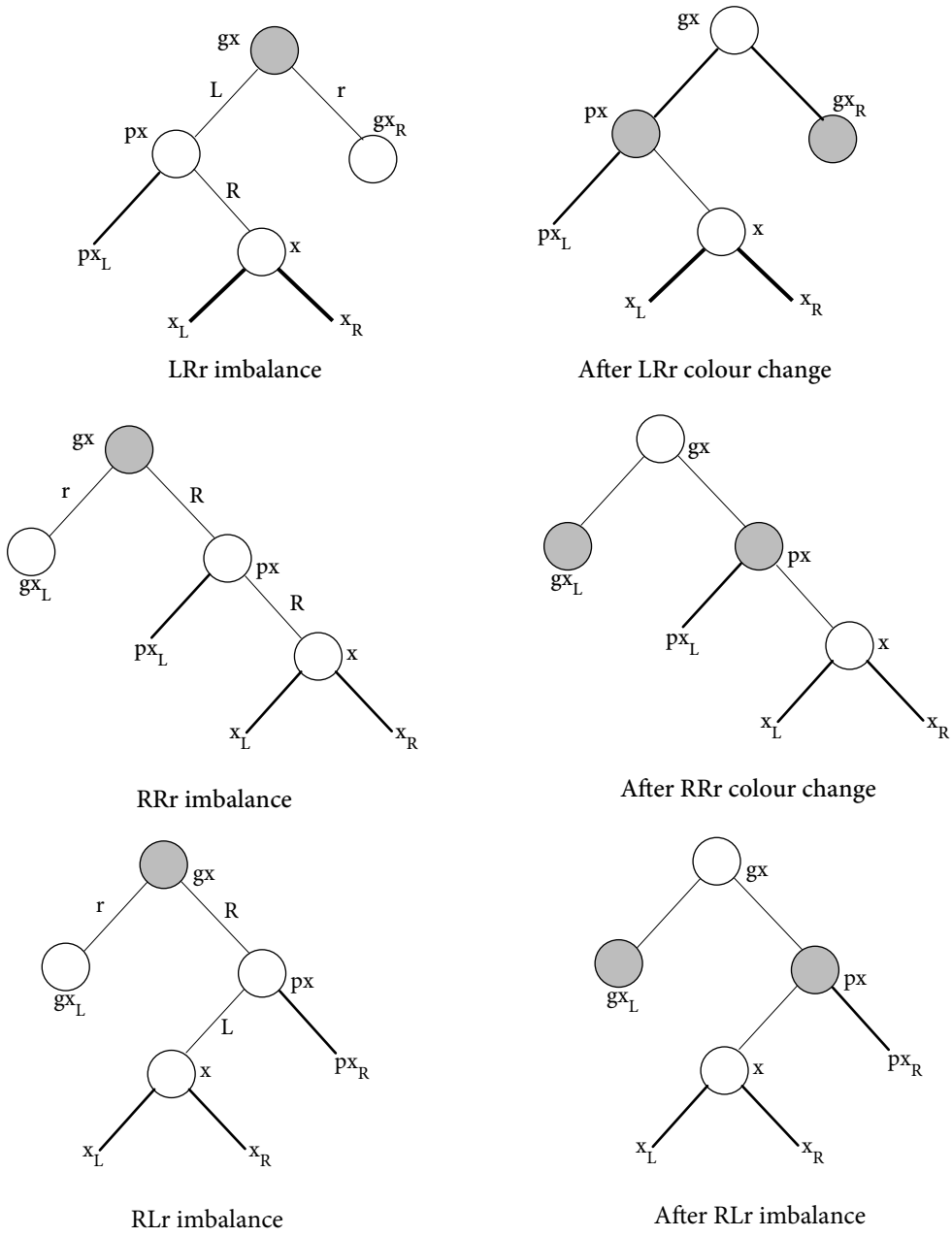
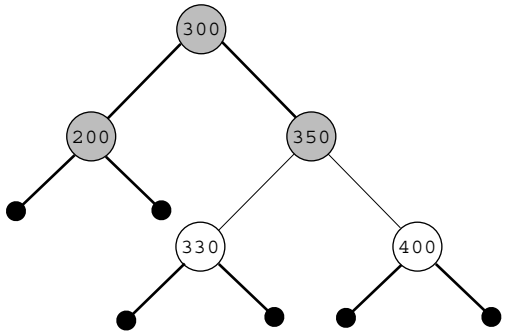
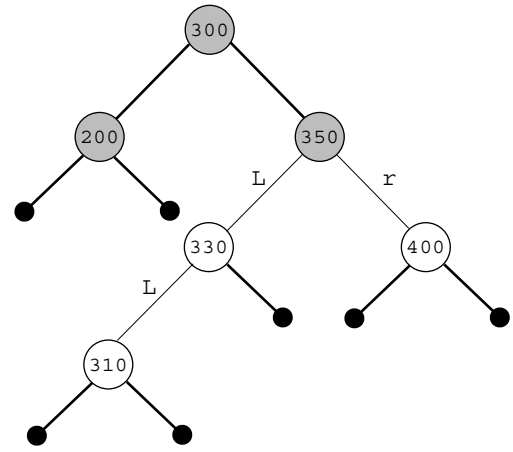


Figure 14.3 General representations of LLr, LRr, RRr and RLr imbalances and colour changes

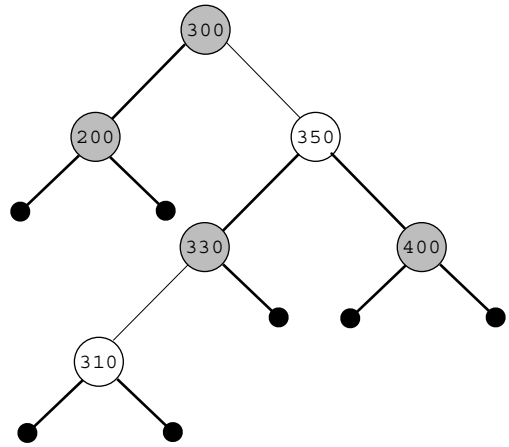
Consider the red–black tree shown in Figure 14.4(a). After inserting 310, LLr imbalance is observed as shown in Figure 14.4(b). LLr colour change is applied on this imbalanced tree. Now, change in colour of the node 350 from black to red and nodes 330 and 400 from red to black can be observed in the rebalanced red–black tree shown in Figure 14.4(c).



(a) Before insertion

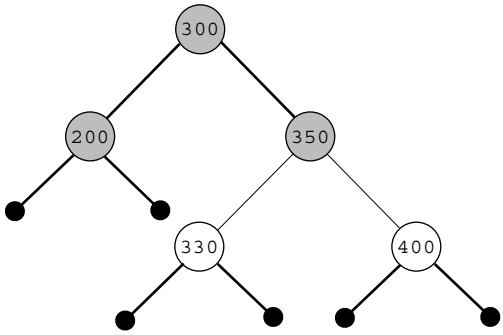


(b) LLr imbalance after inserting 310

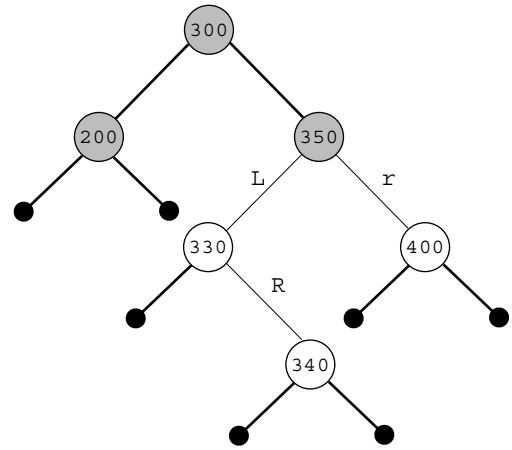


(c) LLr colour change to rebalance the tree

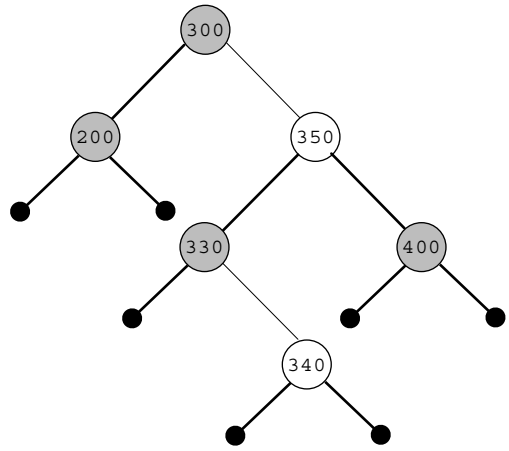
Figure 14.4 Example of LLr imbalance and colour change



(a) Before insertion

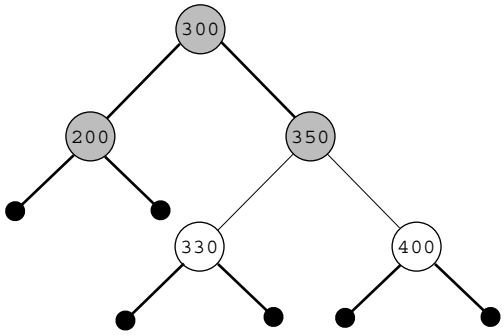


(b) LRr imbalance after inserting 340

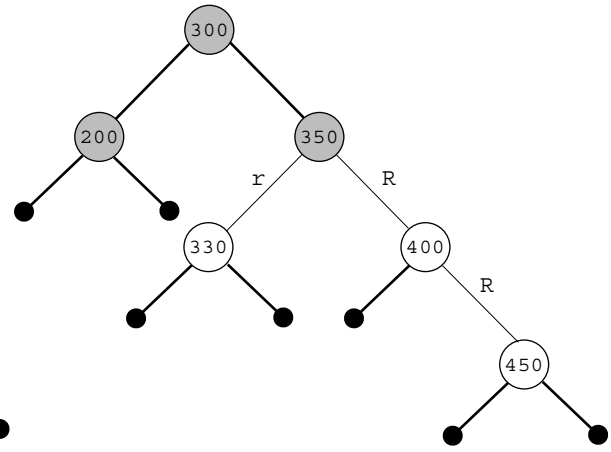


(c) LRr colour change to rebalance the tree

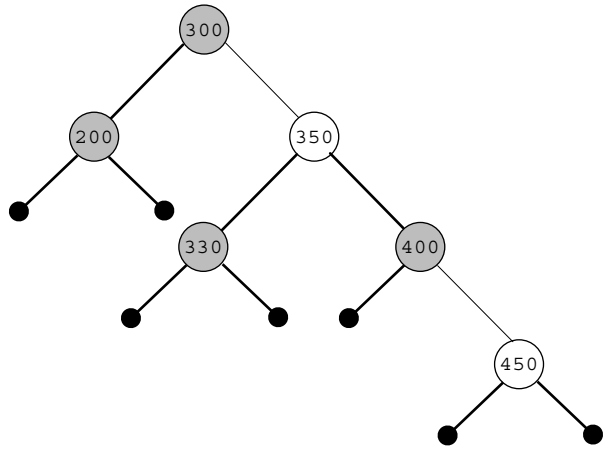
Figure 14.5 Example of LRr imbalance and colour change



(a) Before insertion

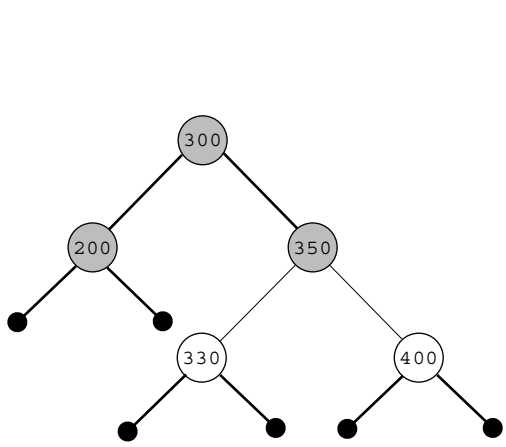


(b) RRr imbalance after inserting 450

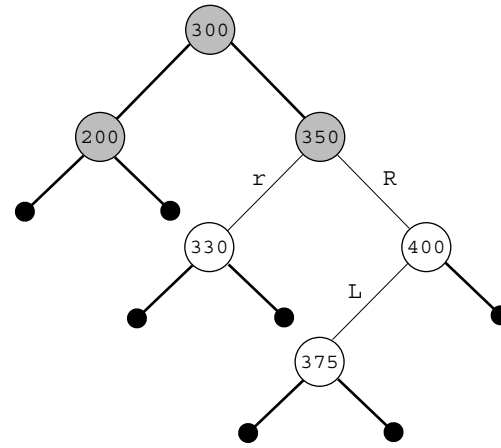


(c) RRr colour change to rebalance the tree

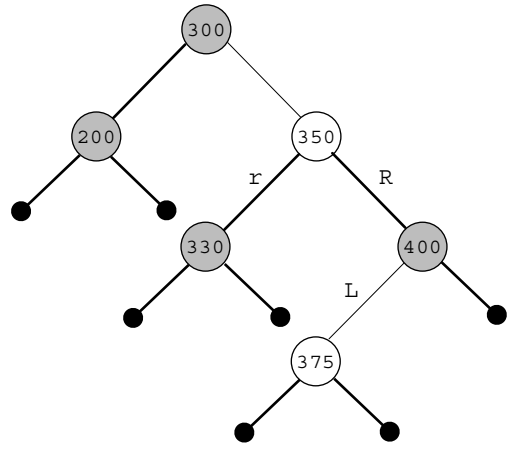
Figure 14.6 Example of RRr imbalance and colour change



(a) Before insertion



(b) RLr imbalance after inserting 375



(c) RLr colour change to rebalance the tree

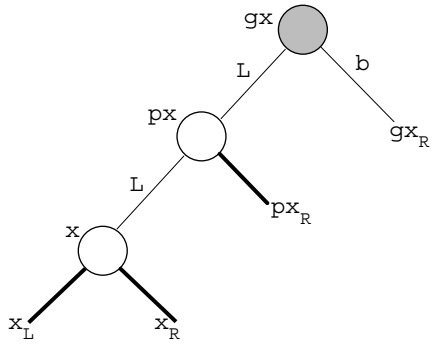
Figure 14.7 Example of RLr imbalance and colour change

The LRr type imbalance after inserting 340 and its corresponding colour changes are shown in Figure 14.5. Similarly, Figures 14.6 and 14.7 show the RRr type imbalance after inserting 450 and RLr type imbalance after inserting 375 and their corresponding colour changes, respectively.

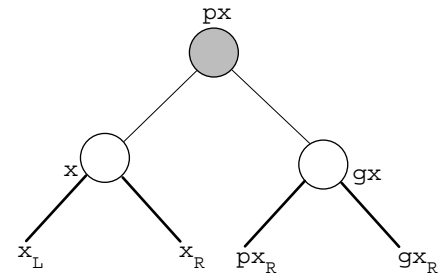
In all the above examples satisfying the number of black nodes and no two consecutive red nodes, properties in the definition of red–black trees can be observed both before insertion and after rebalancing the tree.

Change in colour is possible only when gx is a non-root node. If it is a root node then no colour change is possible. The number of black nodes on all the paths from root node (gx) to the external nodes is increased by 1. In case, change in colour of gx from black to red leads to further imbalance up the tree, then it is identified as one of the imbalance type and corresponding rebalancing method is applied.

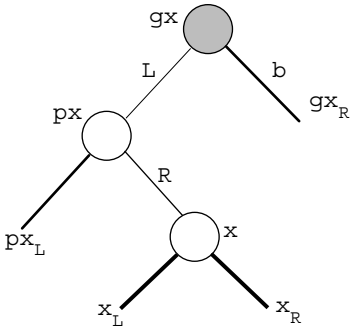
LLb, LRb, RRb, and RLb imbalances: The general representation of LLb, LRb, RRb and RLb imbalances and their corresponding rotations in rebalancing the red–black tree are shown in Figure 14.8. The classification of imbalance is depicted by L, R and b notations on the edges of the red–black trees.



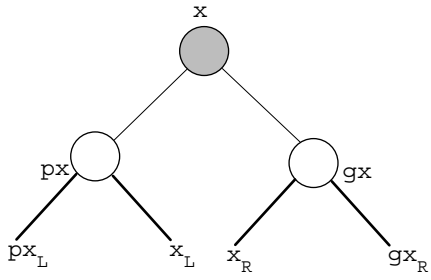
LLb imbalance



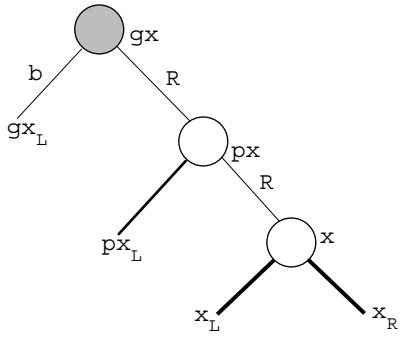
After LLb rotation



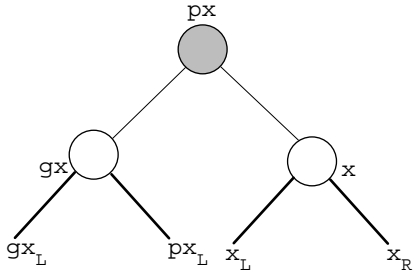
LRb imbalance



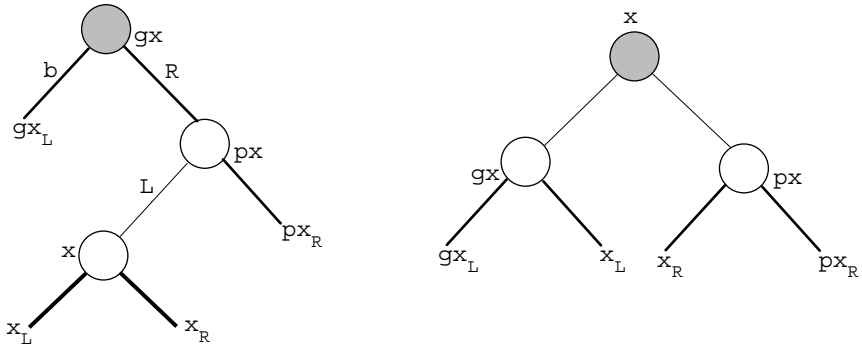
After LRb rotation



RRb imbalance

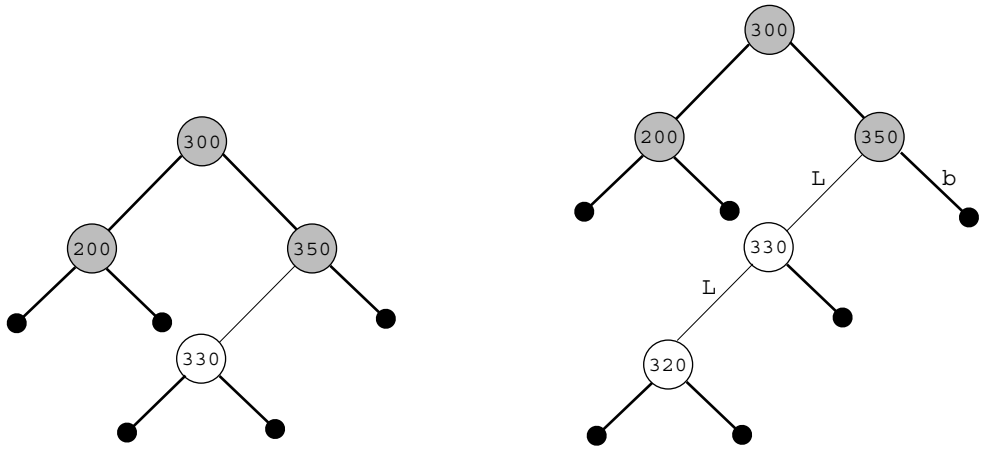


After RRb rotation

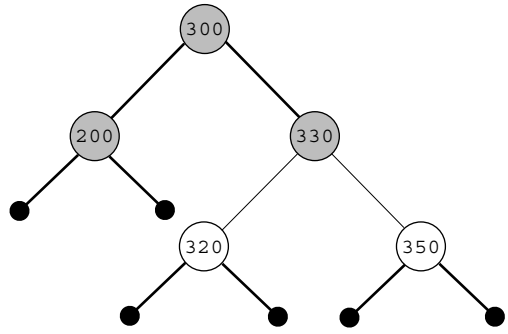


RLb imbalance After RLb rotation

Figure 14.8 General representations of LLb, LRb, RRb and RLb imbalances and rotations

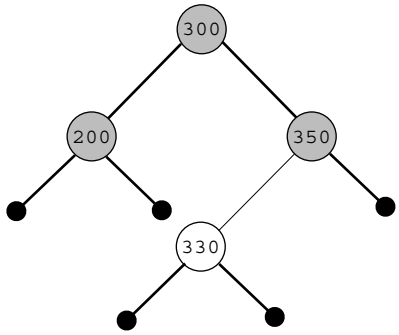


(a) Before insertion (b) LLb imbalance after inserting 320

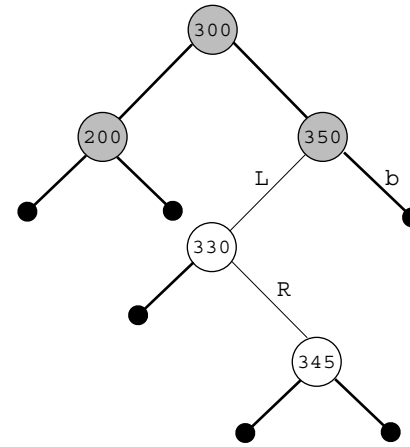


(c) LLb rotation to rebalance the tree

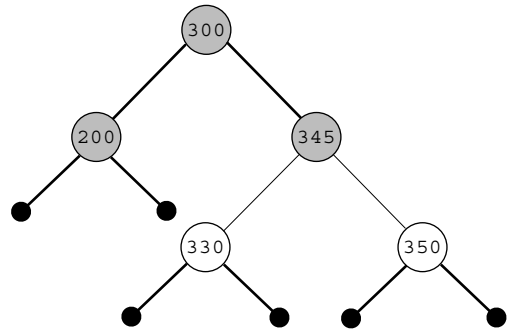
Figure 14.9 Example of LLb imbalance and rotation



(a) Before insertion

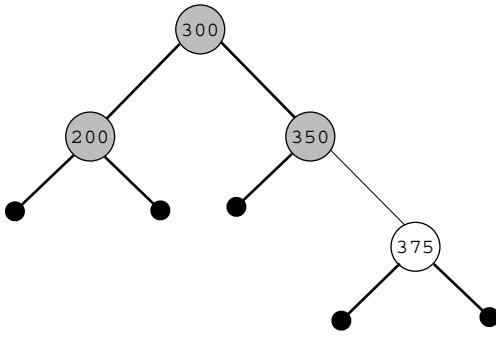


(b) LRb imbalance after inserting 345

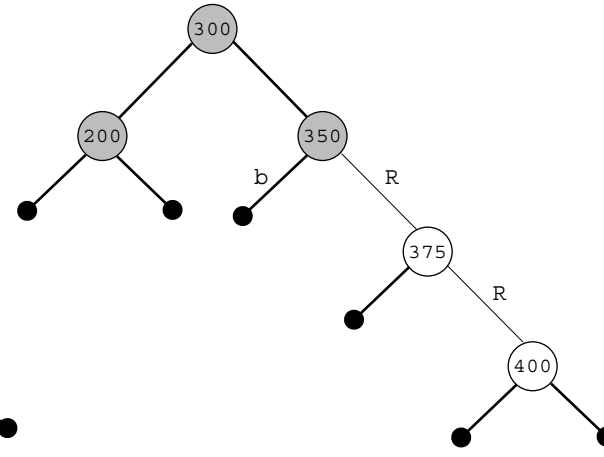


(c) LRb rotation to rebalance the tree

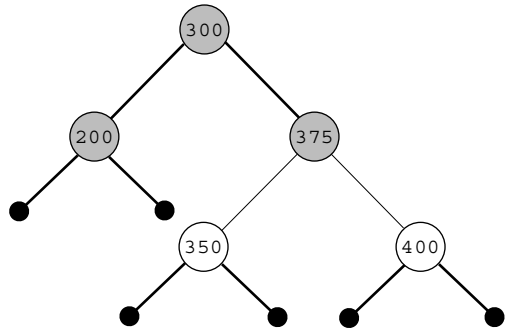
Figure 14.10 Example of LRb imbalance and rotation



(a) Before insertion

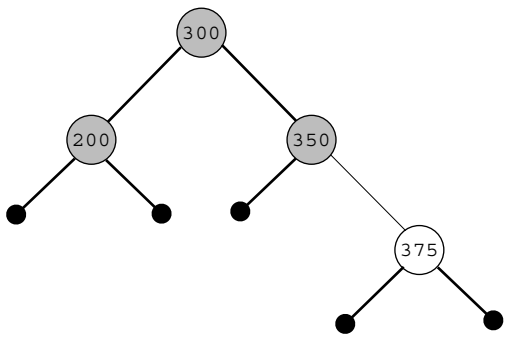


(b) RRb imbalance after inserting 400

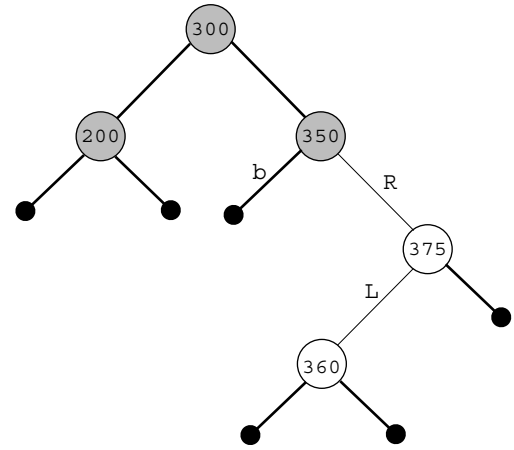


(c) RRb rotation to rebalance the tree

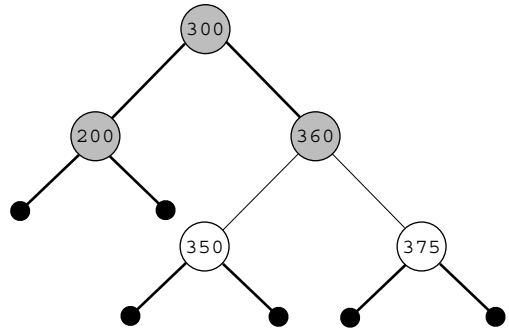
Figure 14.11 Example of RRb imbalance and rotation



(a) Before insertion



(b) RLb imbalance after inserting 360



(c) RLb rotation to rebalance the tree

Figure 14.12 Example of RLb imbalance and rotation

Consider Figure 14.9(a) a red–black tree before insertion. Insertion of 320 causes LLb imbalance it is shown in Figure 14.9(b). To rebalance the tree LLb rotation is invoked, this is shown in Figure 14.9(c). Here 330 is the new root of the subtree and 320, 350 are its left and right children, respectively. Node 330 changed its colour from red to black and 350 from black to red, and accordingly there is change in its pointers colour. After rotation the number of black nodes on all the paths from root node to external nodes is the same as before insertion. No two consecutive red nodes are also found. Hence, the resultant is a rebalanced red–black tree.

Similarly, Figure 14.10 illustrates LRb imbalance due to insertion of 345 and its corresponding rotation. Figures 14.11 and 14.12 depict the cases of RRb imbalance and RLb imbalances after inserting 400 and 360 and their corresponding rotations, respectively.

In all the above examples the rebalanced red–black trees after rotations satisfy the properties of the red–black trees. Except colour change LLb, LRb, RRb and RLb rotations are the same as LL, LR, RR and RL rotations of AVL search trees.

14.3.3 Deletion

Deletion of an element from a red–black tree follows the same procedure as that of a binary search tree. The three cases of deleting an element from a binary search tree, the element as a leaf node, element with one subtree (either left subtree or right subtree only) and elements with two subtrees.

The deletion may lead to the imbalance of the red–black tree that may propagate toward the root node and it may be solved either by colour change or by appropriate rotations.

If the deleted node is red it will not violate the properties of a red–black tree. So, no imbalance occurs and the tree is still a red–black tree. If the deleted node is black then it violates the third property of the red–black trees as the number of black nodes in the path from the root node to the external node passing through the deleted node is decreased by 1. So, the tree is imbalanced.

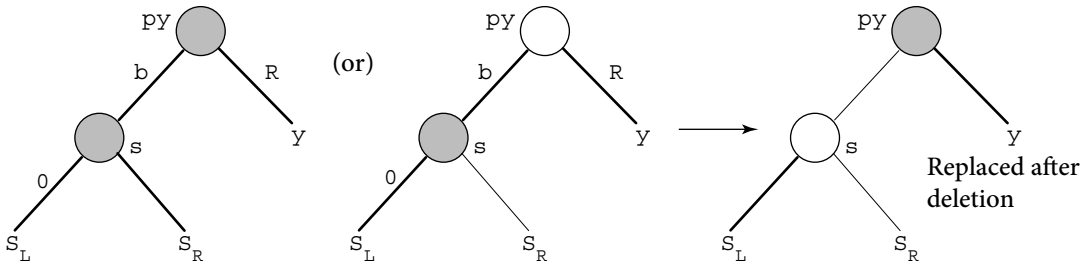
The type of imbalance left (L) or right (R) is determined by the location of the deleted node with respect to its parent (py). Here, y refers to the node that replaces the deleted node. If the sibling of the deleted node s is a black node then the imbalance type is further categorized as LB or RB. If s is a red node then the imbalances are Lr or Rr.

If s is black then Lb or Rb is further classified as Lb0, Lb1 and Lb2 and Rb0, Rb1 and Rb2, respectively, based on whether s has 0 or 1 or 2 red children. If s is red then Lr or Rr is further classified as Lr0, Lr1 and Lr2 and Rr0, Rr1 and Rr2 based on 0 or 1 or 2 grandchildren of s.

These imbalances are handled by corresponding rotations. During rotations swap only the elements but not the colours. The imbalances are explained based on their category along with the general representations and examples. In the representation the hatched node indicates either red or black colour of the node.

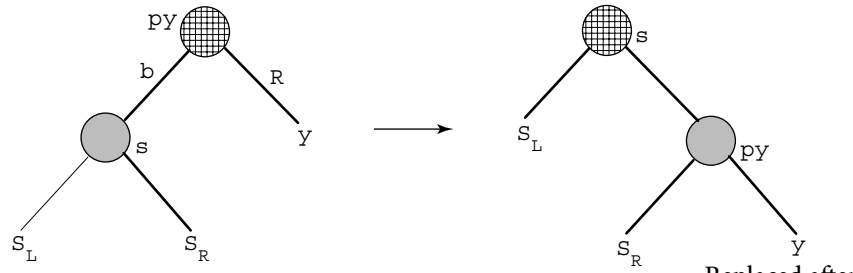
Rb0, Rb1 and Rb2 imbalances: The general representation of Rb0, Rb1 and Rb2 imbalances and their corresponding rotations are depicted in Figure 14.13. The notations R, b and 0 or 1 or 2 specified on the edges of the red–black tree indicate the classification of the imbalance.

The two possible cases of Rb0 are shown in Figure 14.13. The rebalancing merely calls for colour change of nodes. This may send the imbalance up to the root of the subtree where appropriate rebalancing steps must be taken. Rb1 imbalance is again of two types and specified as Rb1 (type 1) and Rb1 (type 2). In these two cases, s has a single red child either in s_L or w, respectively. The Rb2 imbalance occurs when s has two red children in s_L and w.



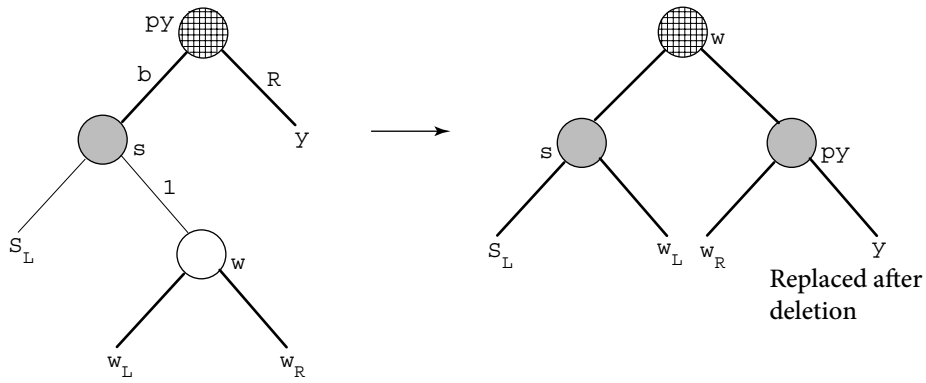
Rb0 imbalance

Rb0 colour change, sending the imbalance up to the root of the subtree



Rb1(type 1) imbalance

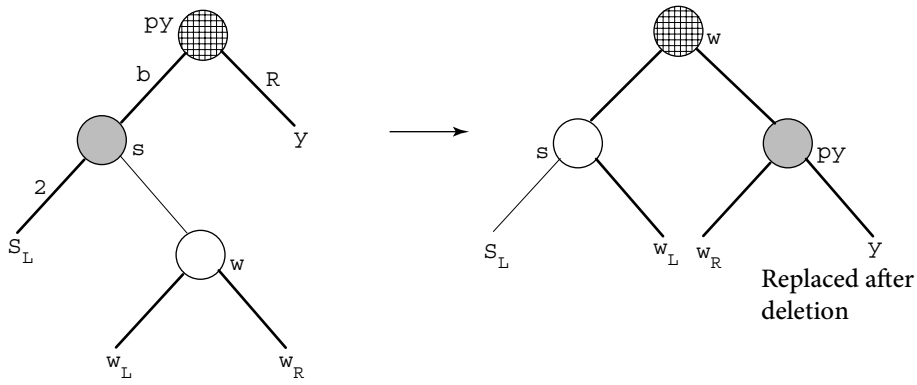
After Rb1(type 1) rotation



Rb1(type 2) imbalance

After Rb1(type 2) rotation

Figure 14.13 General representation of Rb0, Rb1 and Rb2 imbalances and corresponding rebalancing mechanism for red–black trees

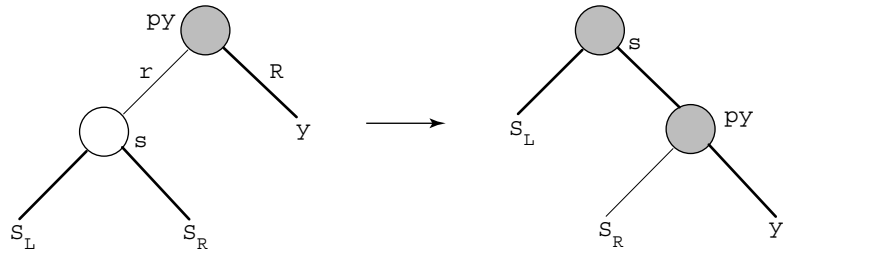


Rb2 imbalance

After Rb2 rotation

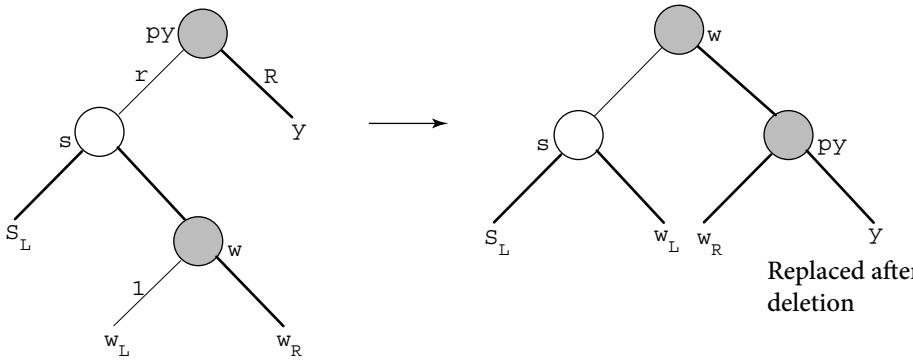
Figure 14.13 *Continued*

Rr0, Rr1 and Rr2 imbalances: General representations of Rr0, Rr1 and Rr2 imbalances and their corresponding rotations are shown in Figure 14.14. The notations R, r and 0 or 1 or 2 specified on the edges of the red–black tree indicate the classification of the imbalance. Rebalancing requires rotations in all the three cases of imbalances. Rr1 is again of two types and specified as Rr1(type1) and Rr1(type 2).



Rr0 imbalance

After Rr0 rotation, it might result in LLb imbalance of both parent and child having the red colour



Rr1(type 1) imbalance

After Rr1(type 1) rotation

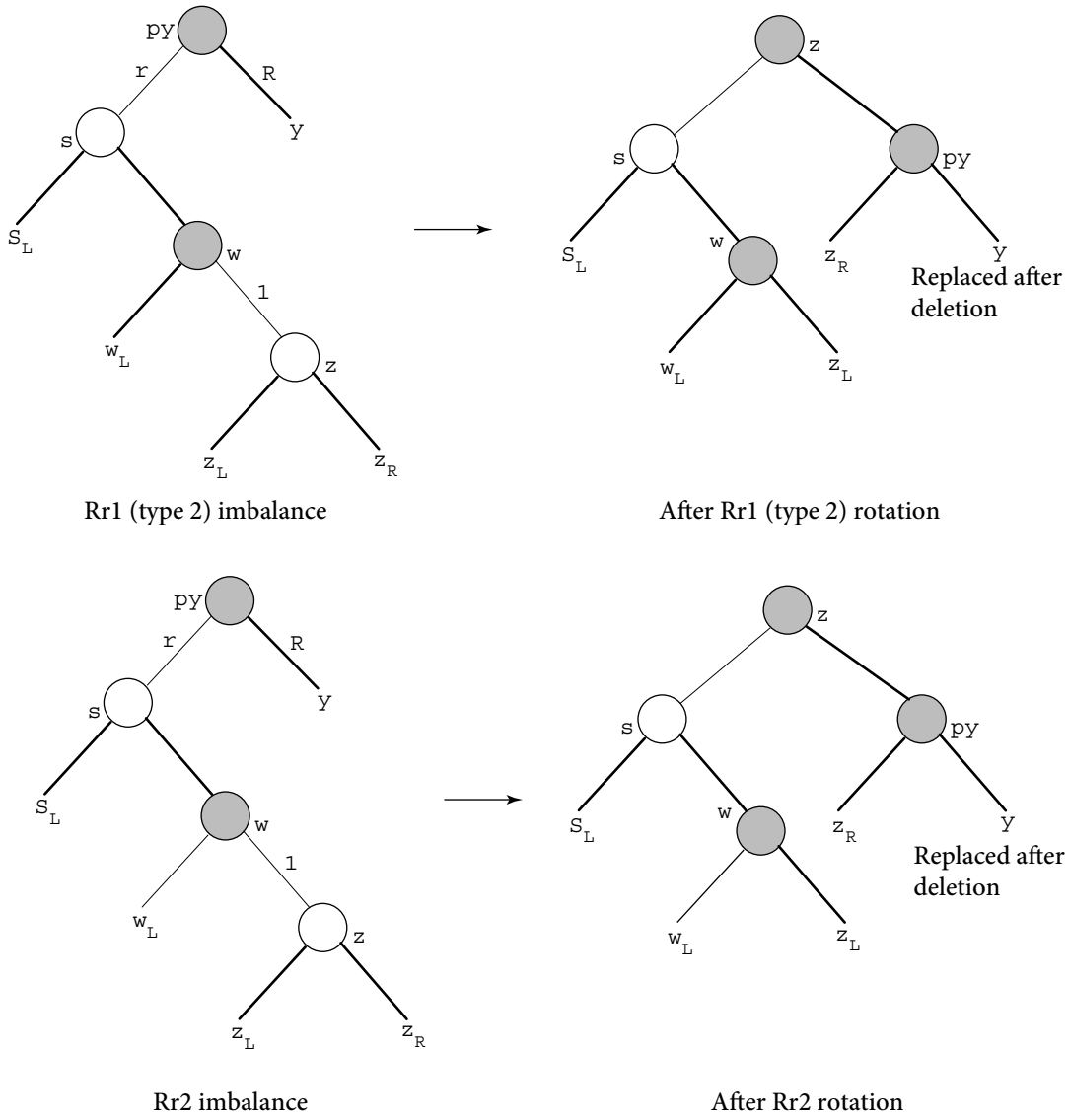
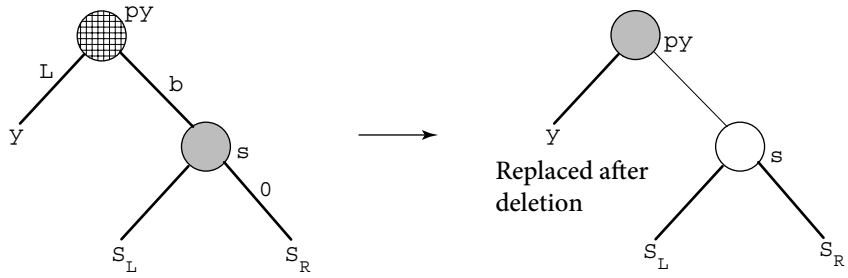


Figure 14.14 General representation of Rr0, Rr1 and Rr2 imbalances and corresponding rotation for red–black trees

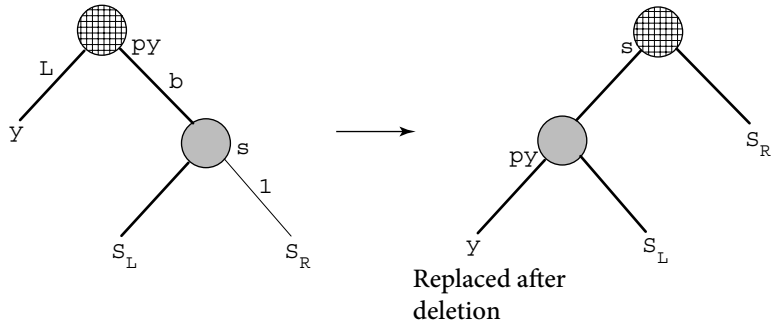
Lb0, Lb1 and Lb2 imbalances: The general representation of Lb0, Lb1 and Lb2 imbalances and their corresponding rotations are depicted in Figure 14.15. The notations L, b and 0 or 1 or 2 specified on the edges of the red–black tree indicate the classification of the imbalance.

The rebalancing Lb0 imbalance merely calls for colour change of nodes. Lb1 imbalance is again of two types and specified as Lb1 (type 1) and Lb1 (type 2).



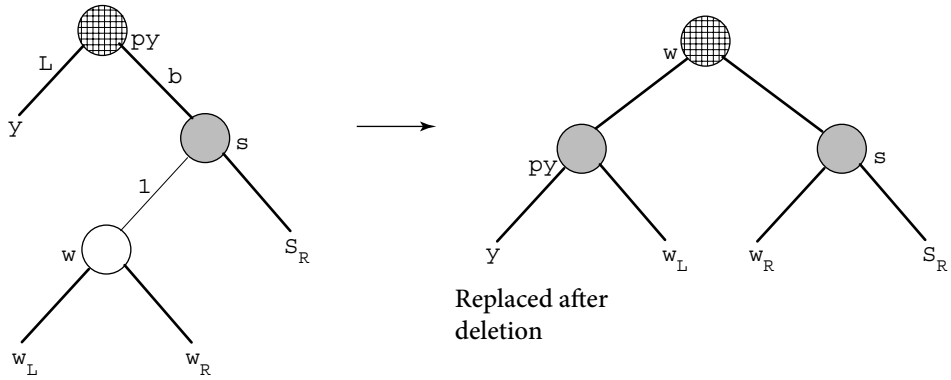
Lb0 imbalance

After Lb0 colour change



Lb1(type 1) imbalance

After Lb1(type 1) rotation



Lb1(type 2) imbalance

After Lb1(type 2) rotation

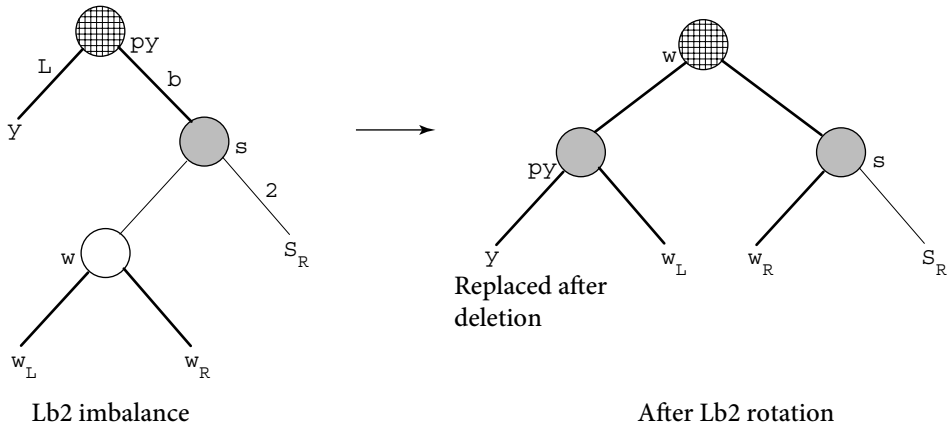


Figure 14.15 General representation of Lb0, Lb1 and Lb2 imbalances and corresponding rebalancing mechanism for red–black trees

Lr0, Lr1 and Lr2 imbalances: The general representation of Lr0, Lr1 and Lr2 imbalances and their corresponding rotations are shown in Figure 14.16. The notations L, r and 0 or 1 or 2 specified on the edges of the red–black tree indicate the classification of the imbalance. Rebalancing requires rotations in all the three cases of imbalances. Lr1 is again of two types and specified as Lr1(type 1) and Lr1(type 2).

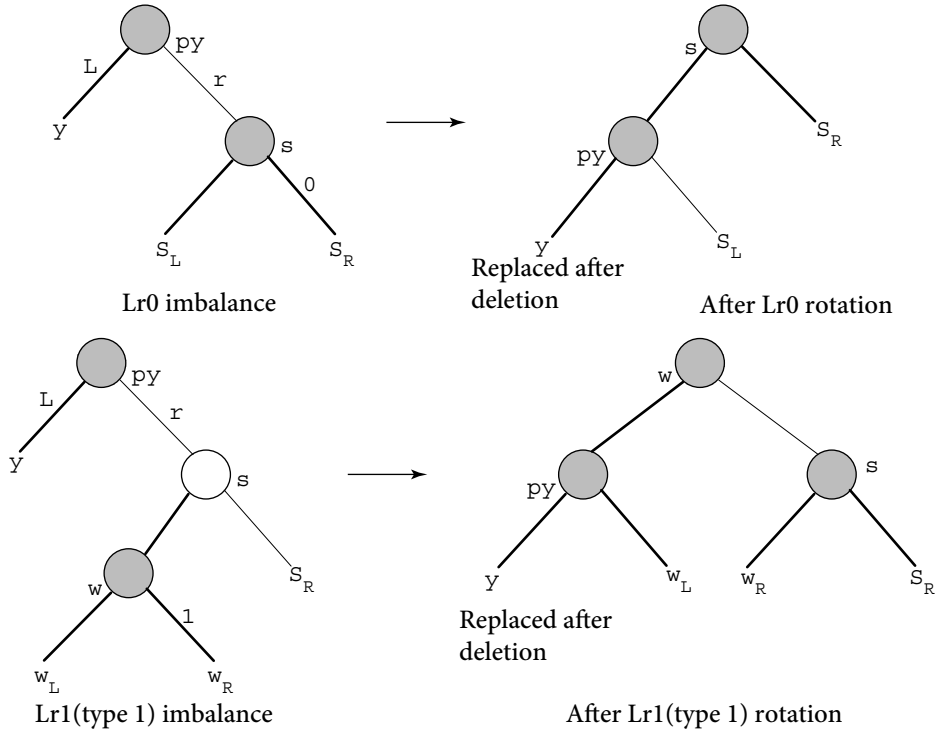
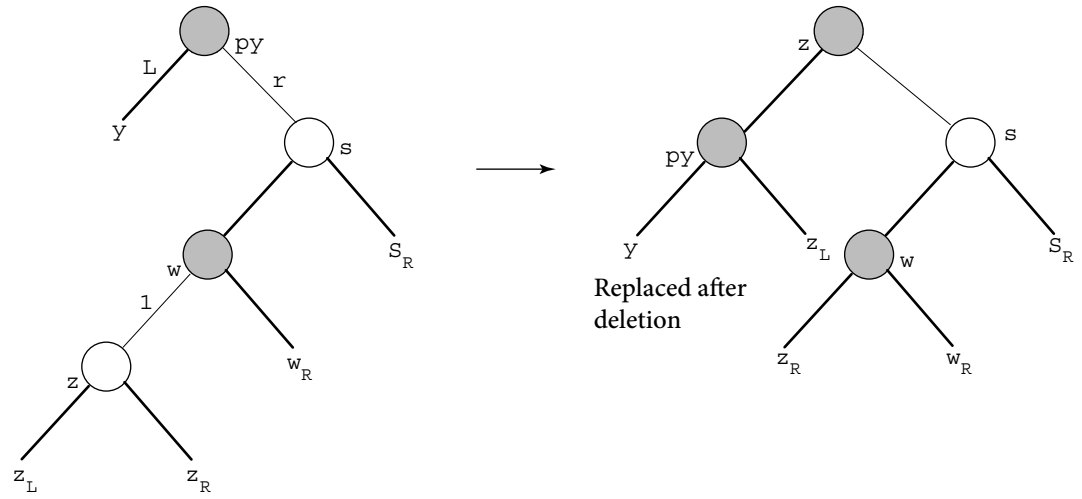
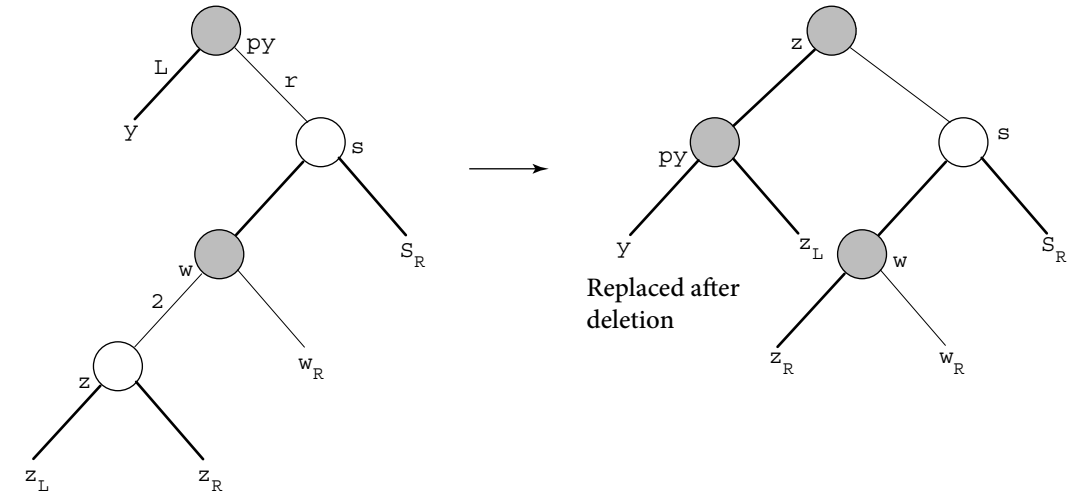


Figure 14.16 General representation of Lr0, Lr1, and Lr2 imbalances and corresponding rotations for red–black trees



Lr1(type 2) imbalance

After Lr1(type 2) rotation

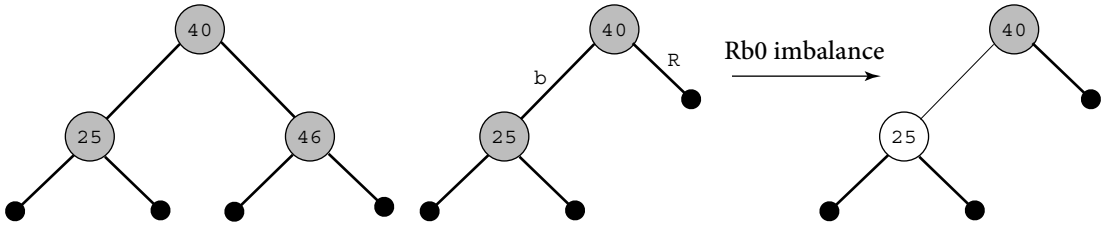


Lr2 imbalance

After Lr2 rotation

Figure 14.16 *Continued*

Delete 46



Before deletion

After deletion

(a)

After Rb0 colour change

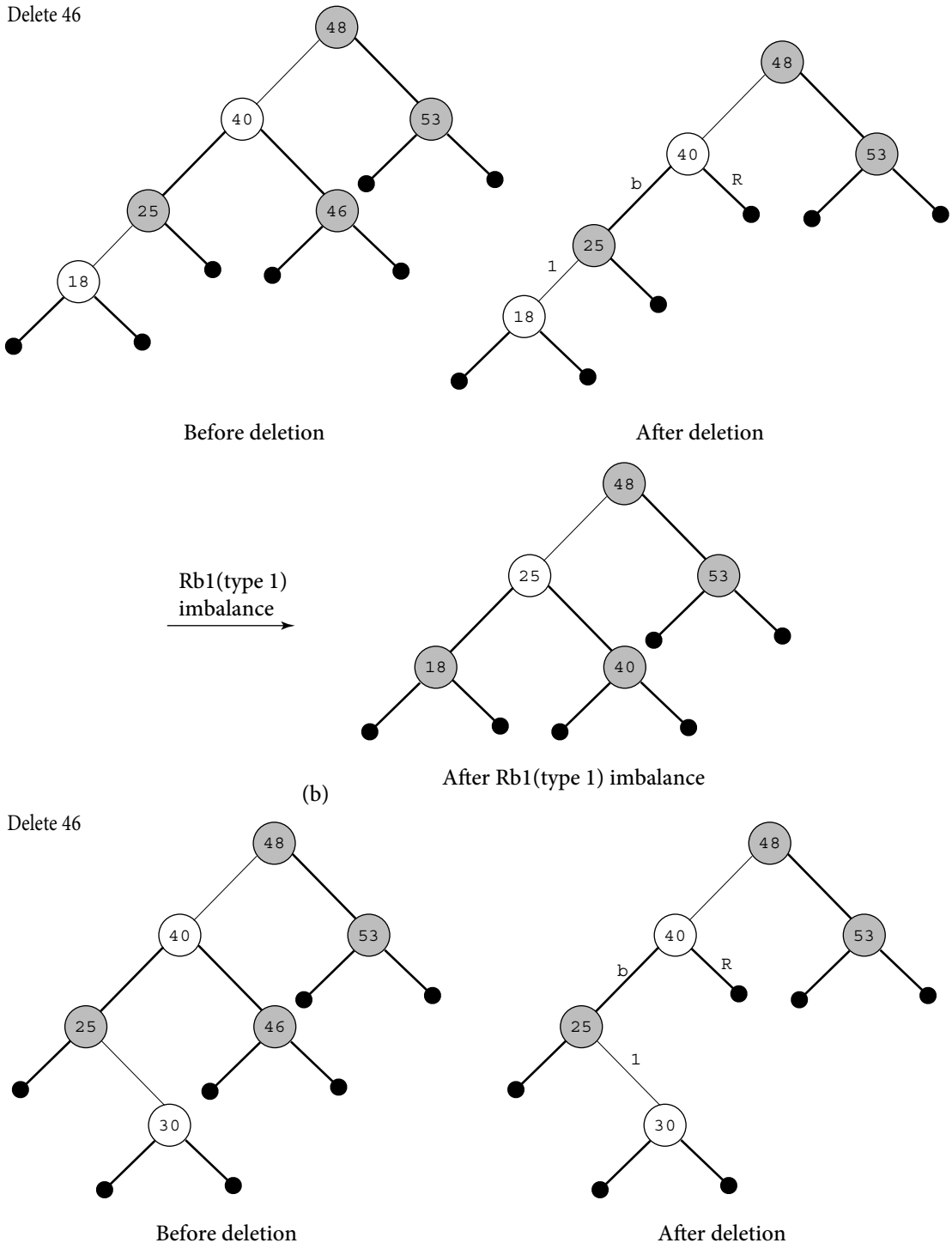
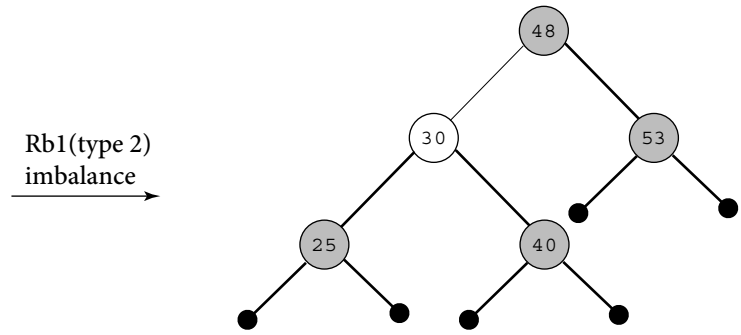


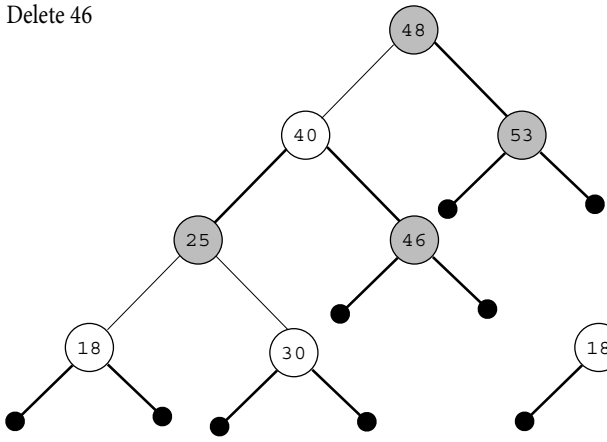
Figure 14.17 Example Rb0, Rb1 and Rb2 imbalances



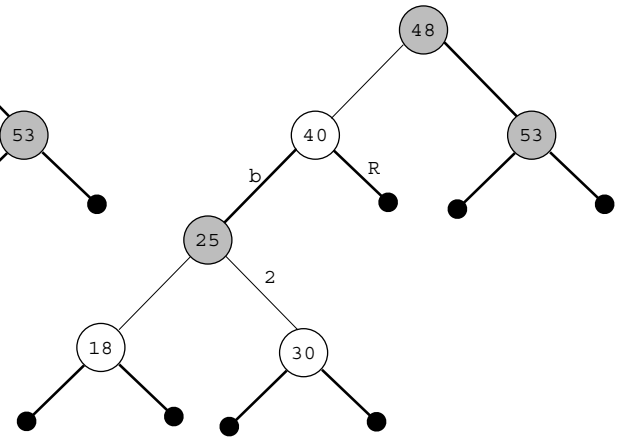
After Rb1(type 2) imbalance

(c)

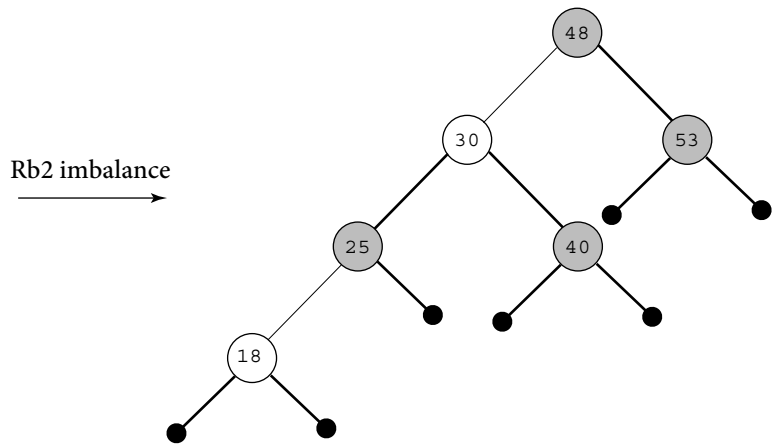
Delete 46



Before deletion



After deletion



(d) After Rb2 rotation

Figure 14.17 *Continued*

An example for the imbalances regarding Rb0, Rb1 and Rb2 and their rebalancing mechanisms like colour change and corresponding rotations are shown in Figure 14.17. The snapshots are self-explanatory.

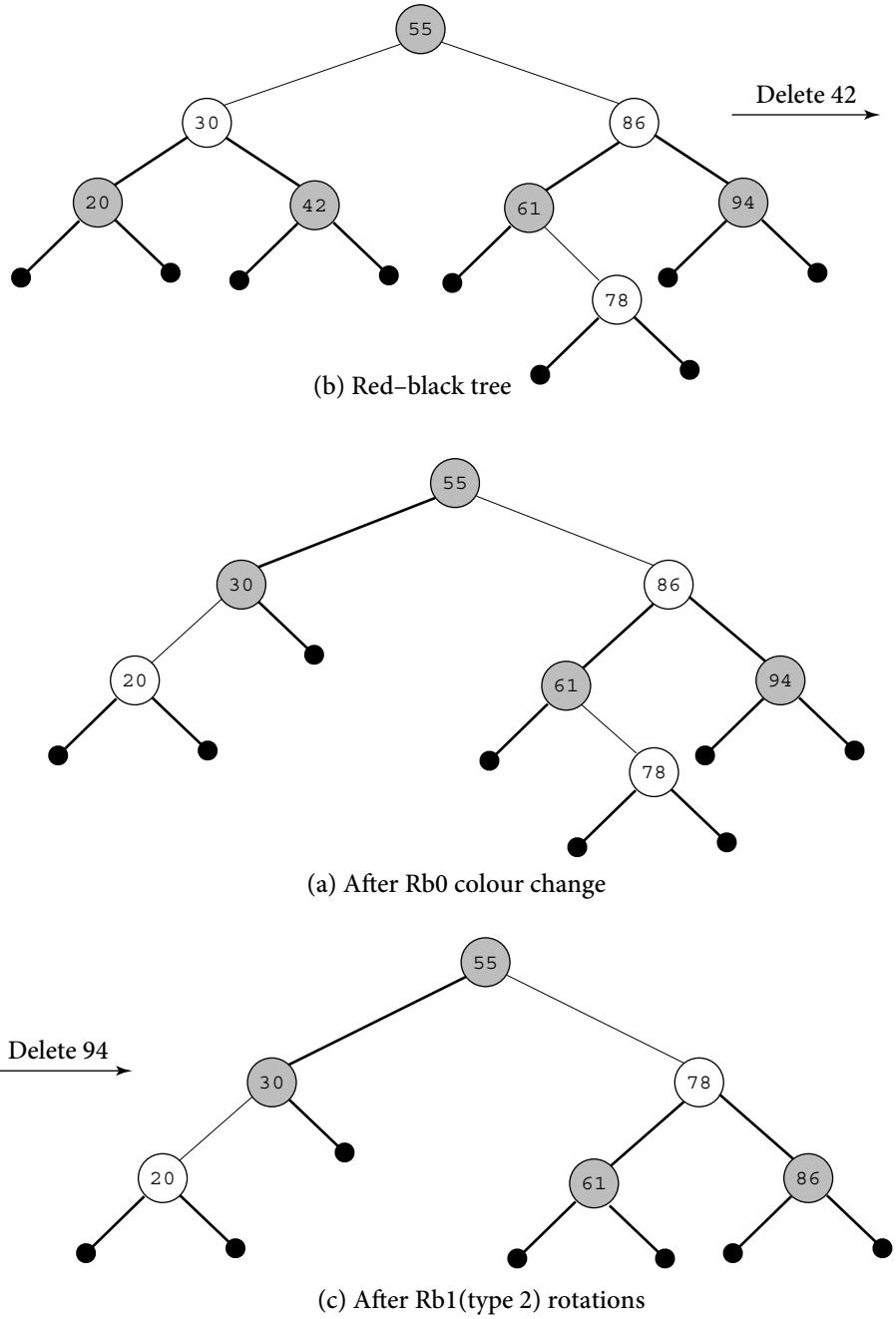
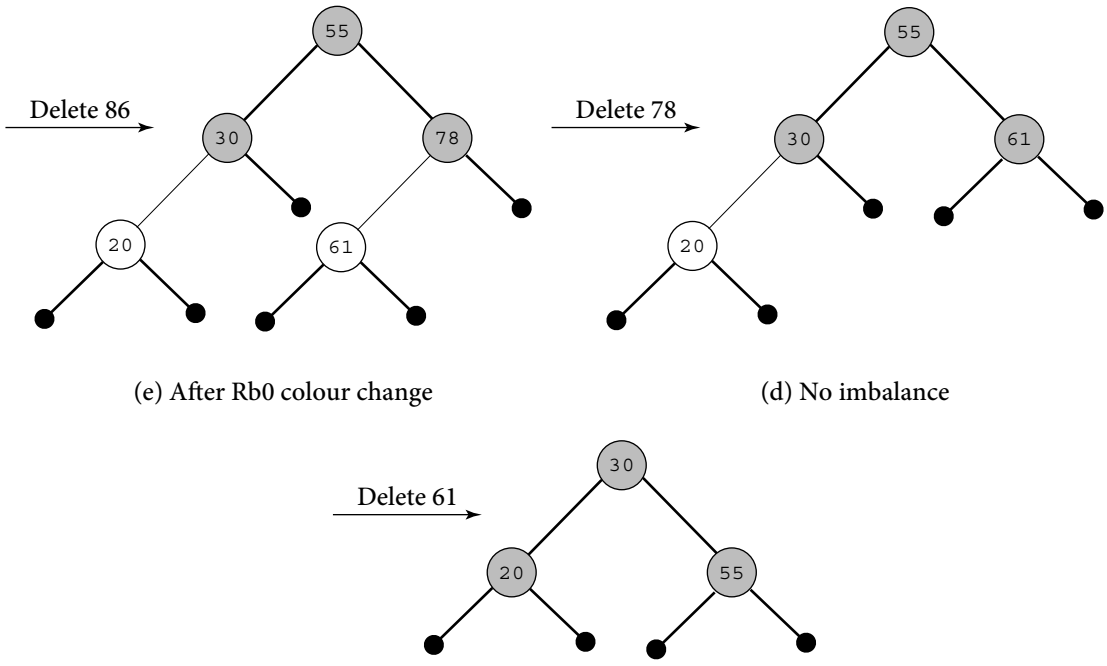


Figure 14.18 Example of deletion operations on a red–black tree



(e) After Rb0 colour change

(d) No imbalance

(f) After Rb1(type 1) rotation

Figure 14.18 *Continued*

Consider a red–black tree shown in Figure 14.18(a) Deleting the elements 42, 94, 86, 78 and 61 from the tree calls corresponding rotations and colour changes. Their snapshots are shown in Figures 14.18(b)–14.18 (f). They are all self-explanatory.

Program 14.1

```
/*To Implement the following RED BLACK TREE Operations
  1.Insertion 2.Deletion 3.Searching 4.Traversing */
#include<stdlib.h>
#include<conio.h>
#include<iostream.h>
struct RBT
{
 int data,color;
 struct RBT*lchild,*rchild,*parent;
}*root;
typedef struct RBT node;
int s,k;
node*minimum(node*ptr)
{
 while(ptr->lchild!=NULL)
 ptr=ptr->lchild;
```

```
 return ptr;
}
node*maximum(node*ptr)
{
 while(ptr->rchild!=NULL)
 ptr=ptr->rchild;
 return ptr;
}
void setparent(node*r,node*r1)
{
 if(r!=NULL) r->parent=r1;
}
void rrrotate(node*x)
{
 node*y;
 y=x->lchild;
 x->lchild=y->rchild;
 setparent(y->rchild,x);
 setparent(y,x->parent);
 if(x->parent==NULL)
 root=y;
 else if(x==x->parent->rchild)
 x->parent->rchild=y;
 else
 x->parent->lchild=y;
 y->rchild=x;
 setparent(x,y);
}
void llrotate(node*x)
{
 node*y;
 y=x->rchild;
 x->rchild=y->lchild;
 setparent(y->lchild,x);
 setparent(y,x->parent);
 if(x->parent==NULL)
 root=y;
 else if(x==x->parent->lchild)
 x->parent->lchild=y;
 else
 x->parent->rchild=y;
 y->lchild=x;
 setparent(x,y);
}
node*successor(node*ptr)
{
 node*y;
 if(ptr->rchild!=NULL)
```

```
 return(minimum(ptr->rchild));
 y=ptr->parent;
 while(y!=NULL&&ptr==y->rchild)
 {
  ptr=y;
  y=y->parent;
 }
 if(y==NULL)
  return ptr;
 else
  return y;
}
node*predecessor(node*ptr)
{
 if(ptr->parent!=NULL)
 if(ptr->parent->rchild==ptr)
  return ptr->parent;
 if(ptr->lchild!=NULL)
  return maximum(ptr->lchild);
 return ptr;
}
node*binarysearch(int i,node*r)
{
 if(r==NULL||r->value==i) return r;
 if(r->value>i)
  r=r->lchild;
 else
  r=r->rchild;
 return binarysearch(i,r);
}
void RBDeleteFixup(node*x)
{
 node*w;
 while(x!=NULL&&x->color==1)
 {
  if(x==x->parent->lchild)
  {
   w=x->parent->rchild;
   if(w!=NULL)
   {
    if(w->color==0)
    {
     w->color=1;
     x->parent->color=0;
     llrotate(x->parent);
     w=x->parent->rchild;
    }
    if(w->lchild->color==1&&w->rchild->color==1)
```

```
    {
     w->color=0;
     x=x->parent;
    }
    else if(w->rchild->color==1)
    {
     w->lchild->color=1;
     w->color=0;
     rrrotate(w);
     w=x->parent->rchild;
    }
    w->color=x->parent->color;
    x->parent->color=1;
    w->rchild->color=1;
   }
   llrotate(x->parent);
   x=root;
  }
  else  {
   w=x->parent->lchild;
   if(w!=NULL)
   {
    if(w->color==0)
    {
     w->color=1;
     x->parent->color=0;
     llrotate((x->parent));
     w=x->parent->lchild;
    }
    if(w->rchild->color==1&&w->lchild->color==1)
    {
     w->color=0;
     x=x->parent;
    }
    else if(w->lchild->color==1)
    {
     w->rchild->color=1;
     w->color=0;
     rrrotate(w);
     w=x->parent->lchild;
    }
    w->color=x->parent->color;
    x->parent->color=1;
    w->lchild->color=1;
   }
   llrotate(x->parent);
   x=root;
  }
```

```
  x->color=1;
 }
}
void delet(node*z)
{
 node*y,*x;
 y=z;
 if(z==NULL)
 cout<<"No Node to Delete";
 else
 {
  if(z->lchild==NULL||z->rchild==NULL)
   y=z;
  else
   y=successor(z);
  if(y->lchild==NULL)
   x=y->lchild;
  else
   x=y->rchild;
  setparent(x,y->parent);
  if(y->parent==NULL)
   root=x;
  else if(y==y->parent->lchild)
   y->parent->lchild=x;
  else
   y->parent->rchild=x;
  if(y!=z)
   z->data=y->data;
  if(y->color==1)
   RBDeleteFixup(x);
 }
}
void RBInsertFixup(node*x)
{
 node*y;
 while(x->parent!=NULL&&x->parent->color==0)
 {
  if(x->parent==x->parent->parent->lchild)
  {
   y=x->parent->parent->rchild;
   if(y!=NULL)
   {
    if(y->color==0)
    {
     x->parent->color=1;
     y->color=1;
     x->parent->parent->color=0;
     x=x->parent->parent;
```

```
     }
    }
    else
    {
     if(x==x->parent->rchild)
     {
      x=x->parent;
      llrotate(x);
     }
     else
     {
      x->parent->color=1;
      x->parent->parent->color=0;
      rrrotate(x->parent->parent);
     }
    }
   }
   else
   {
    y=x->parent->parent->lchild;
    if(y!=NULL)
    {
     if(y->color==0)
     {
      x->parent->color=1;
      y->color=1;
      x->parent->parent->color=0;
      x=x->parent->parent;
     }
    }
    else
    {
     if(x==x->parent->lchild)
     {
      x=x->parent;
      llrotate(x);
     }
     else
     {
      x->parent->color=1;
      x->parent->parent->color=0;
      rrrotate(x->parent->parent);
     }
    }
   }
  }
  root->color=1;
}
```

```
void insert(node*z)
{
 node*x,*y;
 y=NULL;
 x=root;
 while(x!=NULL)
 {
  y=x;
  if(z->data<x->data)
   x=x->lchild;
  else
   x=x->rchild;
 }
 z->parent=y;
 if(y==NULL)
 root=z;
 else
 {
  z->parent=y;
  if(z->data<y->data)
  y->lchild=z;
  else
  y->rchild=z;
 }
 RBInsertFixup(z);
}
int height(node*ptr)
{
 if(ptr!=NULL)
   return((height(ptr->rchild)>height(ptr->lchild)?height(ptr->rchild)+1:
height(ptr->lchild)+1));
 else return -1;
}
void inorder(node*ptr)
{
 node*z;
 if(ptr->lchild!=NULL)
 inorder(ptr->lchild);
 cout<<endl<<ptr->data<<" ";
 cout<<"Height:"<<height(ptr)<<" ";
 z=successor(ptr);
 cout<<"Successor:"<<z->data<<" ";
 z=predecessor(ptr);
 cout<<"Predecessor:"<<z->data<<" "<<"Color:";
 if(ptr->color==0)cout<<"Red";else cout<<"Black";
 if(ptr->parent!=NULL) cout<<"Parent:"<<ptr->parent->data;
 if(ptr->rchild!=NULL)
 inorder(ptr->rchild);
```

```
}
int search(int key,node*ptr)
{
 if(ptr!=NULL)
 {
  if(ptr->data==key)
   k=1;
   else
   {
    search(key,ptr->lchild);
    search(key,ptr->rchild);
   }
 }
 return k;
}
void main()
{
 node*z;
 int op,item;
 clrscr();
 root=NULL;
 while(1)
 {
  cout<<"\nRed Black Tree Operations";
  cout<<"\n1.Insertion\n2.Deletion\n3.Searching\n4.Traversing\n5.Exit";
  cout<<"\nEnter your option:";
 cin>>op;
 switch(op)
 {
  case 1:z=(node*)malloc(sizeof(node));
         cout<<"Enter Item To Be Insert";
         cin>>item;
    z->data=item;
    z->color=0;
    z->rchild=NULL;
    z->lchild=NULL;
    z->parent=NULL;
    insert(z);
    break;
  case 2:if(root==NULL)
      cout<<"Tree Is Empty";
    else
    {
     cout<<"Enter Item To Be Delete";
     cin>>item;
     delet(binarysearch(item,root));
    }
    break;
```

```
    case 3:s=k=0;
      cout<<"Enter Element To Search:";
      cin>>item;
      s=search(item,root);
      if(s==1)
      cout<<"Search Is Found";
      else
      cout<<"Search Is Not Found";
      break;
    case 4:cout<<"Inorder Traversal:\n";
      inorder(root);
      break;
    case 5:exit(0);
   }
  }
}
```

Output

```
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:1
Enter Item To Be Insert11
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:1
Enter Item To Be Insert2
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:1
Enter Item To Be Insert14
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
```

```
Enter your option:1
Enter Item To Be Insert1
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:1
Enter Item To Be Insert7
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:1
Enter Item To Be Insert15
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:1
Enter Item To Be Insert5
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:1
Enter Item To Be Insert8
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:4
Enter your option:4
Inorder Traversal:
1    Height:0   Successor:2    Predecessor:1    Color:Black   Parent:2
2    Height:2   Successor:5    Predecessor:1    Color:Red     Parent:11
5    Height:0   Successor:7    Predecessor:5    Color:Red     Parent:7
7    Height:1   Successor:8    Predecessor:2    Color:Black   Parent:2
8    Height:0   Successor:11   Predecessor:7    Color:Red     Parent:7
```

```
11   Height:3   Successor:14   Predecessor:8    Color:Black
14   Height:1   Successor:15   Predecessor:11   Color:Black   Parent:11
15   Height:0   Successor:11   Predecessor:14   Color:Red     Parent:14
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:2
Enter Item To Be Delete7
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:4
Traversal:
1    Height:0   Successor:2    Predecessor:1    Color:Black   Parent:2
2    Height:2   Successor:5    Predecessor:1    Color:Red     Parent:11
5    Height:0   Successor:8    Predecessor:5    Color:Red     Parent:8
8    Height:1   Successor:11   Predecessor:2    Color:Black   Parent:2
11   Height:3   Successor:14   Predecessor:8    Color:Black
14   Height:1   Successor:15   Predecessor:11   Color:Black   Parent:11
15   Height:0   Successor:11   Predecessor:14   Color:Red     Parent:14
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:3
Enter Element To Search:8
Search Is Found
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:3
Enter Element To Search:10
Search Is Not Found
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
```

```
4.Traversing
5.Exit
Enter your option:2
Enter Item To Be Delete2
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:4
Inorder Traversal:
1    Height:0   Successor:5    Predecessor:1    Color:Black   Parent:5
5    Height:1   Successor:8    Predecessor:1    Color:Red     Parent:11
8    Height:0   Successor:11   Predecessor:5    Color:Black   Parent:5
11   Height:2   Successor:14   Predecessor:8    Color:Black
14   Height:1   Successor:15   Predecessor:11   Color:Black   Parent:11
15   Height:0   Successor:11   Predecessor:14   Color:Red     Parent:14
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:2
Enter Item To Be Delete11
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:4
Inorder Traversal:
1    Height:0   Successor:5    Predecessor:1   Color:Black   Parent:5
5    Height:1   Successor:8    Predecessor:1   Color:Red     Parent:14
8    Height:0   Successor:14   Predecessor:5   Color:Black   Parent:5
14   Height:2   Successor:14   Predecessor:8   Color:Black
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:1
Enter Item To Be Insert13
```

```
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:4
Inorder Traversal:
1    Height:0   Successor:5    Predecessor:1    Color:Black   Parent:5
5    Height:2   Successor:8    Predecessor:1    Color:Red     Parent:14
8    Height:1   Successor:13   Predecessor:5    Color:Black   Parent:5
13   Height:0   Successor:14   Predecessor:8    Color:Red     Parent:8
14   Height:3   Successor:14   Predecessor:13   Color:Black
Red Black Tree Operations
1.Insertion
2.Deletion
3.Searching
4.Traversing
5.Exit
Enter your option:5
```

Time complexity of operations: The search operation on a red–black tree is exactly the same as is on a binary search tree with the time complexity of O(log n). Insertion and/or deletion operations may invoke colour change which may spread in the upward direction till the root node and may also invoke the rotations to rebalance the tree. The rotations and colour change require only a constant time O(1). In the worst case the insertion or deletion operation will have the time complexity of O(log n). The maximum height of the red–black tree is $2\log_2(n+1)$. So, the operations search, insert and delete that require O(h) will have a time complexity of O(log n).

14.4 Splay Trees

The worst case time complexity of either binary search trees or AVL trees or red–black tree is O(n). A binary search tree is used to store a set of records or elements. If one of them is retrieved for m number of times then the worst case time complexity of this operation sequence will be O(m·n). In many day-to-day applications such as dictionary, once an element is accessed, the likeliness to access the same element frequently is more than to an unaccessed element.

If the accessed element is in a leaf node then its time complexity is O(n). As the same element may be accessed for many times in the near future, if it is available in the root node for further accesses its time complexity will be O(1). Though adjusting the tree in this manner is expensive, it makes further accesses faster and cheaper. In this process of adjustment other elements which are deep down in the tree may come up and allows their accesses to be relatively cheaper.

Daniel Dominic Sleator and Robert Endre Tarjan invented a data structure called a splay tree, which provides the above-discussed mechanism. Splay trees are binary search trees and they are with a wonderful property that can be adapted optimally to a sequence of tree operations. When any operation (such as search, insert and delete) performed on a node, it is moved towards the root node. In due course, the inactive elements are moved far away from the root node.

Splay trees do not require balance or height information and hence are easier to implement. Splay trees are cheaper to AVL trees over long sequences of retrievals. This is shown using the techniques of amortized analysis. It is an analysis over a sequence of operations, among them the expensive operations are averaged over less expensive operations. If O(n) is the time complexity of one access then for the sequence of m operations the amortized analysis of access in a splay tree is O(m·log n).

14.4.1 Splay Rotations

All the operations performed on a splay tree are the same as that of on a binary search tree. But after performing the operation the tree is splayed with respect to the referred node. That is the node is moved up towards the root node using the mechanism called splay rotations. They are similar to AVL tree rotations and follow the bottom up method from the node to the root node.

Splaying tree is aimed to make the referred or accessed node as the root node. In this regard the accessed node x and its parent node px and its grandparent node gx are affected. To perform spalying x should be moved two levels up in every step. When the parent node px became the root node then splaying remains with the last step, i.e. x is one step down the root and single rotation is required for x to become the root. To do so the path from the root to the referred node x is to be tracked. When the path turns left it is termed as **zig** and when it turns right it is termed as **zag**.

So, in the case when x is a single step down the tree the path would be either a **zig** or **zag**. When two steps are to be considered, the path would be four rotations, one among **zig-zig, zig-zag, zag-zig** or **zag-zag**. In the case of even length of the path from root node to the referred node x, corresponding two step series of rotations are used. In case of odd length, the final rotation will be one of the corresponding single step series. The single step and the two step series are as shown in Figure 14.19.

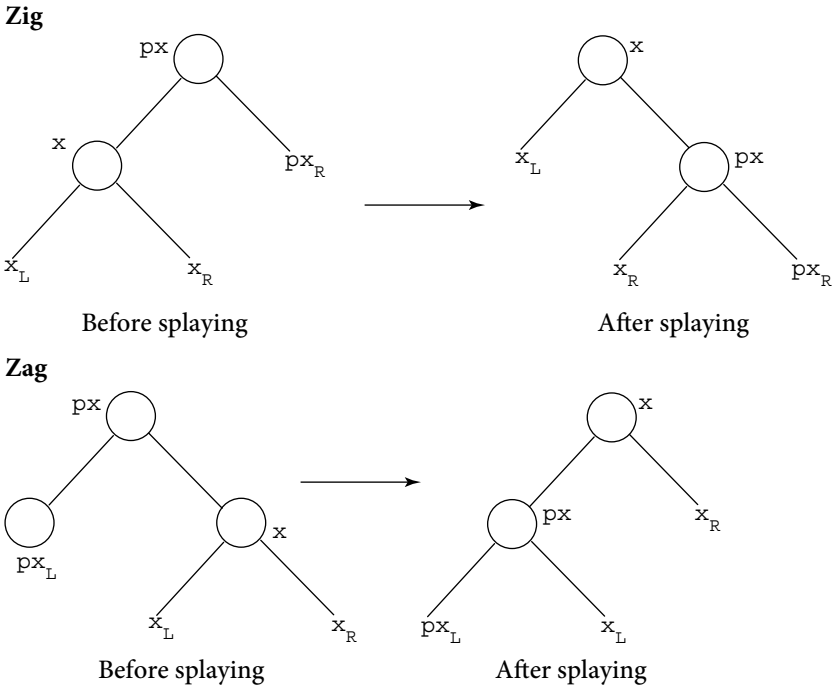
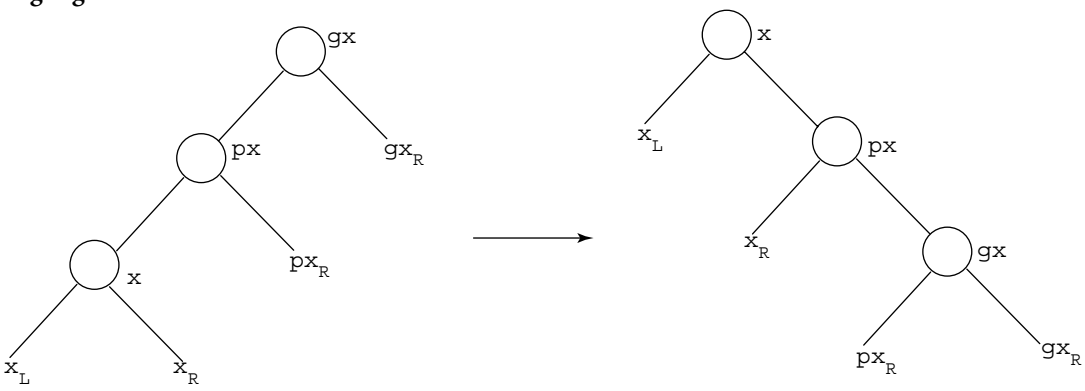


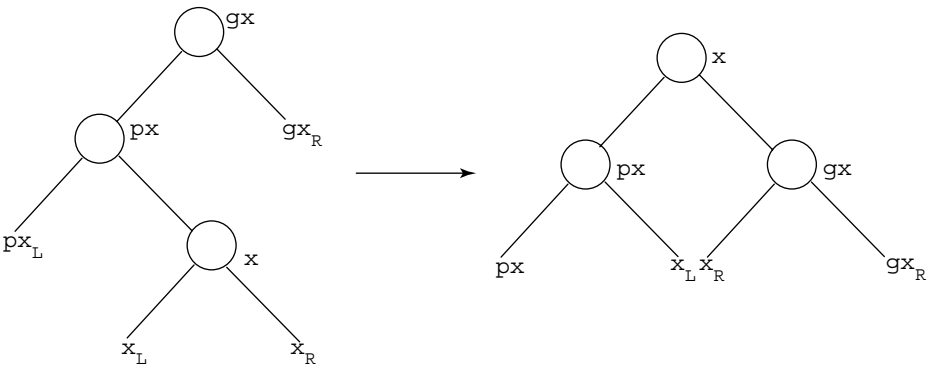
Figure 14.19 Splay rotations

Zig-zig



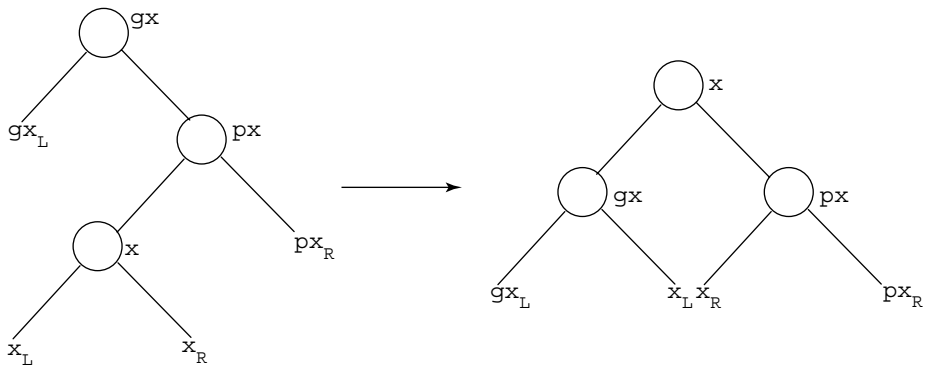
Before splaying After splaying

Zig-zag



Before splaying After splaying

Zag-zig



Before splaying After splaying

Figure 14.19 *Continued*

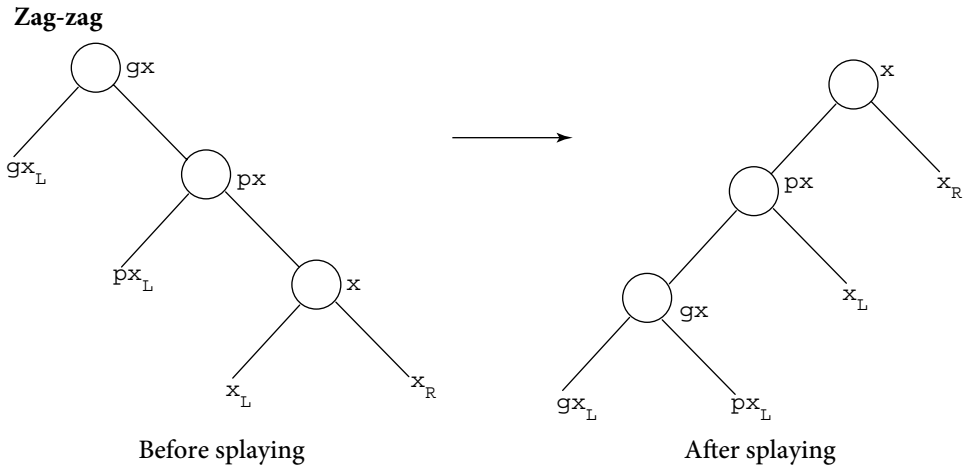
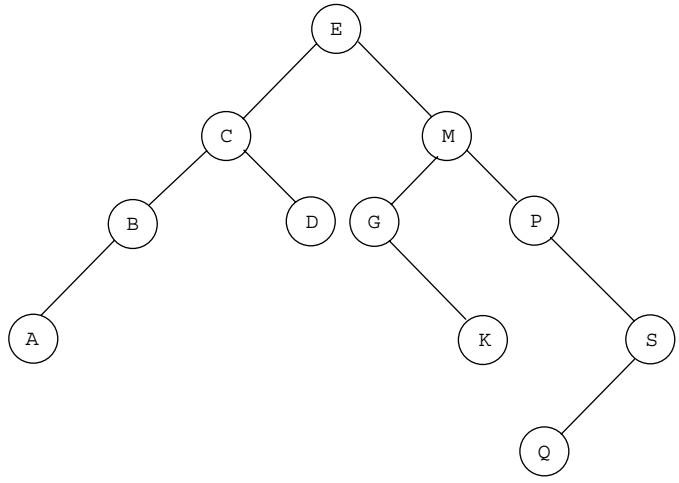


Figure 14.19 *Continued*

On performing a deletion operation, as the deleted node does not appear in the resulting tree, splaying is applied on the parent of the deleted node.

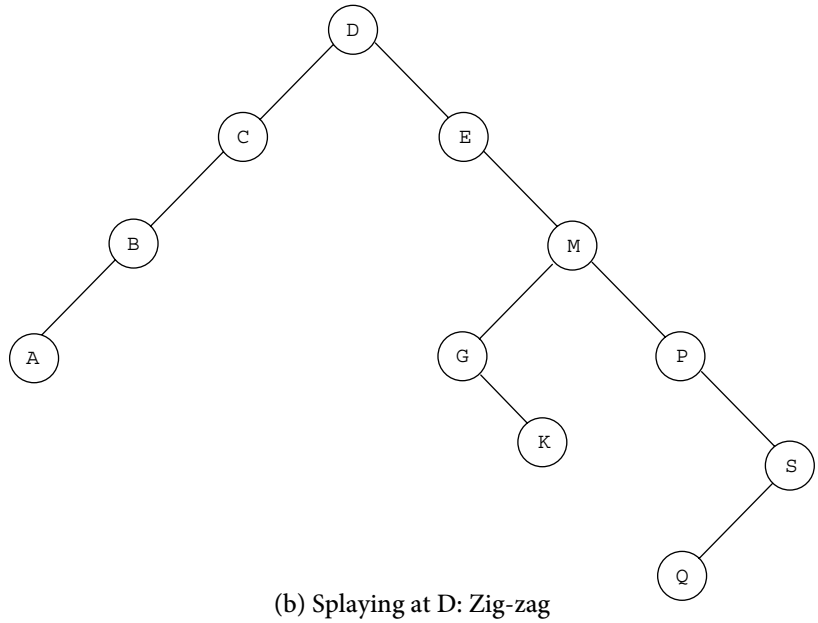
Consider a binary search tree shown in Figure 14.20(a). To splay the tree at node D, the path from the root to node D involves E–C–D. In the bottom-up procedure a **zig-zag** rotation is performed and the resultant tree after splay is shown in Figure 14.20(b).

Consider the tree shown in Figure 14.20(b); to splay at G the path involves D–E–M–G from the root to node G. In the bottom-up procedure the first step involves **zag-zig** rotation. It is shown in Figure 14.20(c) where G is one step down the root, so now apply **zag** rotation to get G in the root node. This is shown in Figure 14.20(d).

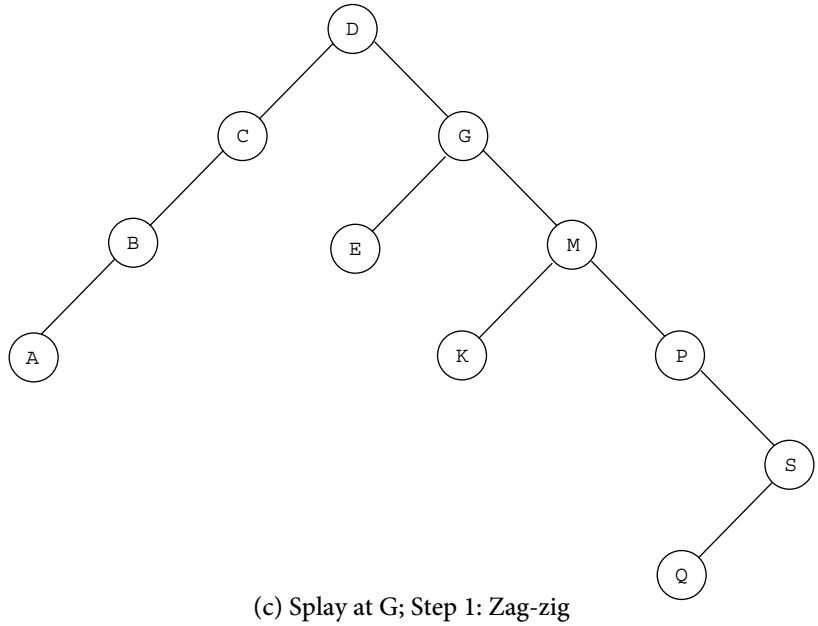


(a) Binary search tree

Figure 14.20 Splaying a binary search tree

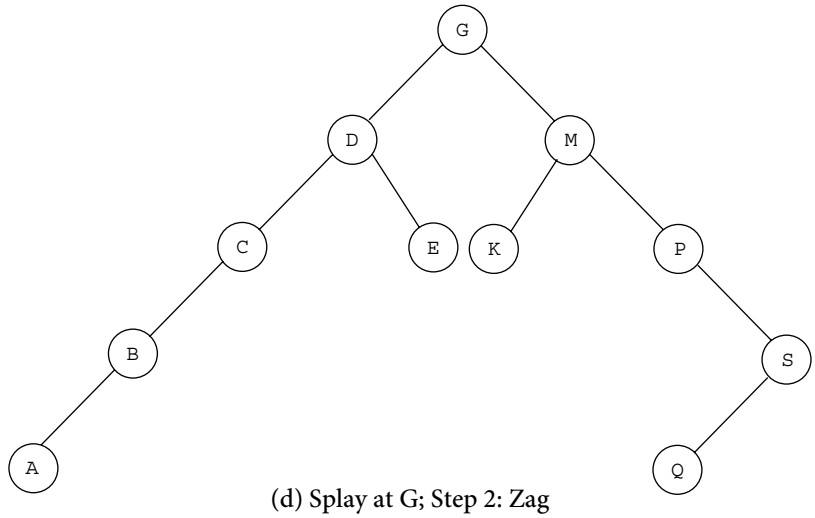


(b) Splaying at D: Zig-zag



(c) Splay at G; Step 1: Zag-zig

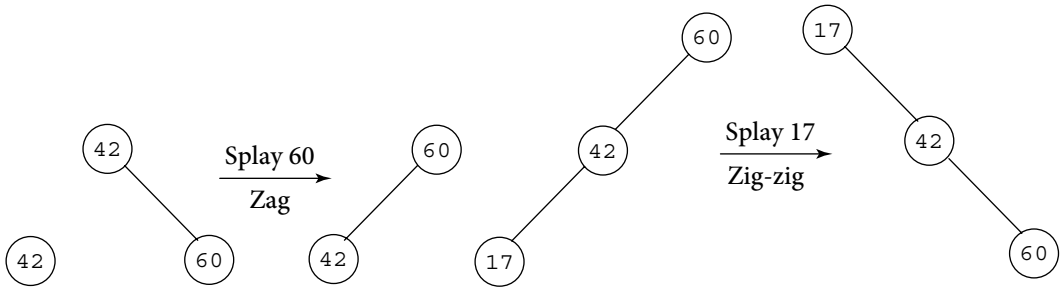
Figure 14.20 *Continued*



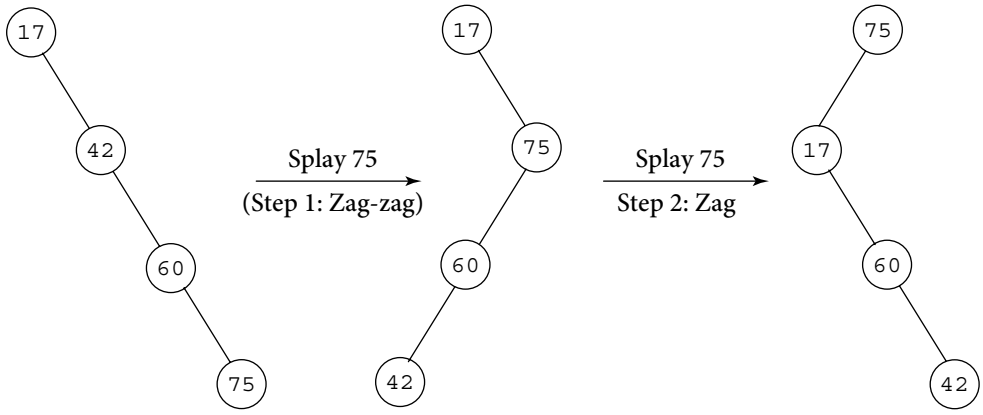
(d) Splay at G; Step 2: Zag

Figure 14.20 *Continued*

Constructing a splay tree with 42, 60, 17, 75. Elements in the sequence can be depicted in Figure 14.21.



(a) Insert 42 (b) Insert 60 (c) Insert 17



(d) Insert 75

Figure 14.21 Steps of insertion operations in a splay tree

The steps of the splay tree while inserting 42 as the root of the splay tree are shown in Figure 14.20(a). Inserting 60 leads to zag rotation and is shown in Figure 14.20(b). Insertion of 17 leads to zig-zig rotation and is shown in Figure 14.20(c). Insertion of 75 calls two splay rotations; their corresponding steps are shown in Figure 14.20(d).

14.4.2 Amortized Analysis

Based on the input instances given to the algorithm the best, average and worst case time complexities are computed and this is known as algorithm analysis.

An amortized analysis analyzes a sequence of operations to show that though a single operation within the sequence may be expensive the average cost per operation is less. Even when averages are taken no probability is involved. The average performance of each operation in the worst case is guaranteed by an amortized analysis. However, an amortized analysis is not an average case analysis. Over a set of independent input instances the work done by the algorithm is handled by the average case analysis. But, the amortized analysis does the same over related or associated instances.

Splay trees give an amortized time complexity as O(log n). The individual operation may not have O(log n) but the amortized complexity of m operations in a splay tree will be O(m·log n).

An amortized analysis is used to establish efficiency in retrieving a node in a binary search tree which uses the splaying technique. For a binary search tree t, let nodes(x) be the number of nodes in the subtree with x as its root node:

rank(x) = log(nodes(x))

rank(root(t)) = log(n) where n is the number of nodes in the tree.

potential(t) = $\Sigma_{x \text{ is a node of } t}$ rank(x).

Always nodes(x) + 1≤ nodes (parent(x)) and rank (x) < rank (parent(x))

To access a node x the amortized cost can be defined as the function

amCost(x) = cost(x) + potential$_s$(t) – potential$_o$(t)

where potential$_s$(t) and potential$_o$(t) refer to the potentials of the tree before and after accessing. It should be noted that a rotation affects only the ranks of node x, which is accessed, its parent px and grandparent gx.

Program 14.2

```
/*To implement the following SPLAY TREE Operations
   1. Insertion  2.Deletion  3.Searching  4.Traversing*/
#define N 50
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
struct SPLAY
{
 int data;
 struct SPLAY*lchild,*rchild;
}*root=NULL,*r=NULL,*ptr,*ptr1,*ne,*ptr2;
int item,k=0,s,a[N],i=0,count=0,ele;
void splay1(struct SPLAY*ptr)
{
 if(ptr!=NULL)
```

```
 {
  splay1(ptr->lchild);
  a[i++]=ptr->data;
  splay1(ptr->rchild);
 }
}
void splay2()
{
 int j;
 if(count>1)
 {
  splay1(r);
  for(j=0;j<i;j++)
  {
   ptr=root;
   if(ptr->data!=a[j]&&a[j]!=ele){
   item=a[j];
   while(ptr!=NULL)
   {
    if(item<ptr->data)
    {
     ptr1=ptr;
     ptr=ptr->lchild;
    }
    else if(item>ptr->data)
    {
     ptr1=ptr;
     ptr=ptr->rchild;
    }
   }
   if(ptr==NULL)
   {
    ne=(struct SPLAY*)malloc(sizeof(struct SPLAY));
    ne->data=item;ne->lchild=ne->rchild=NULL;
    if(ptr1->data<item)
    ptr1->rchild=ne;
    else
    ptr1->lchild=ne;
   }
  } }
 }
 r=root;
 if(count>1)
 i=0;
}
void insert()
{
 int j;
```

```
 ne=(struct SPLAY*)malloc(sizeof(struct SPLAY));
 ne->data=item;ne->lchild=ne->rchild=NULL;
 root=ne;
 count++;
 splay2();
}
void splay3()
{
 int c=0;
 ptr=r;
 while((ptr!=NULL)&&(c==0))
 {
  if(item<ptr->data)
   ptr=ptr->lchild;
  else if(item>ptr->data)
   ptr=ptr->rchild;
  else
   c=1;
 }
 ne=(struct SPLAY*)malloc(sizeof(struct SPLAY));
 ne->data=ptr->data;ne->lchild=ne->rchild=NULL;
 root=ne;
 splay2();
}
void delet(int item)
{
 int c=0;
 struct SPLAY*pt;
 ptr=root;
 if(ptr==NULL)
 {
  cout<<"Tree Is Empty";
  return;
 }
 if((ptr->data==item)&&(ptr->lchild==NULL&&ptr->rchild==NULL))
 {
  free(root);
  root=NULL;
  return;
 }
 else if(ptr->data==item)
 {
  cout<<"Deletion Is Not Possible(It Is Root Node)";
  return;
 }
 else
 {
  while((ptr!=NULL)&&(c==0))
```

```
   {
    if(item<ptr->data)
    {
     ptr1=ptr;
     ptr=ptr->lchild;
    }
    else if(item>ptr->data)
    {
     ptr1=ptr;
     ptr=ptr->rchild;
    }
    else
     c=1;
   }
   if(c==0)
   {
    cout<<"Item Does Not Exist..,Deletion Is Not Possible";
    return;
   }
   ne=(struct SPLAY*)malloc(sizeof(struct SPLAY));
   ne->data=ptr1->data;ne->lchild=ne->rchild=NULL;
   root=ne;
   splay2();
   count--;
  }
}
void traverse(struct SPLAY*ptr)
{
 if(ptr!=NULL)
 {
  cout<<ptr->data<<" ";
  traverse(ptr->lchild);
  traverse(ptr->rchild);
 }
}
int searching(struct SPLAY*p,int key)
{
 if(p!=NULL)
 {
  if(p->data==key)
   k=1;
   else
   {
    searching(p->lchild,key);
    searching(p->rchild,key);
   }
 }
 return k;
```

```
}
main()
{
 int ch,a;
 clrscr();
 while(1)
 {
  cout<<"\nSPLAY TREE OPERATIONS\n1.INSERTION\n2.DELETION\n";
  cout<<"3.SEARCHING\n4.TRAVERSING\n5.EXIT";
  cout<<"\nEnter Your Choice:";
  cin>>ch;
  switch(ch)
  {
   case 1:k=0;cout<<"Enter Element To Be Insert:";
     cin>>item;
     s=searching(root,item);
     if(s!=1)insert();
     else cout<<"Item Already Exists";
     break;
   case 2:cout<<"Which Item You Want To Delete:";cin>>item;
     ele=item;delet(item);break;
   case 3:if(root==NULL){cout<<"Tree Is Empty";break;}
     s=k=0;cout<<"Which Item You Want To Search:";
     cin>>item;ele=item;
     s=searching(root,item);
     if(s==1)
     {
       cout<<"Item"<<item<<"Is Found";
       if(root->lchild!=NULL||root->rchild!=NULL)
       splay3();
     }
     else
     cout<<"Item"<<item<<"Is Not Found";
     break;
   case 4:if(root==NULL){cout<<"Treee Is Empty";break;}
     cout<<"Tree Elements Are:";traverse(root);break;
   case 5:exit(0);
   default:cout<<"Invalid Choice";
  }
 }
}
```

Output

```
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
```

```
4.TRAVERSING
5.EXIT
Enter Your Choice:1
Enter Element To Be Insert:10
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:1
Enter Element To Be Insert:5
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:1
Enter Element To Be Insert:24
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:1
Enter Element To Be Insert:36
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:1
Enter Element To Be Insert:40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:1
Enter Element To Be Insert:15
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
```

```
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:15 5 10 24 36 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:3
Which Item You Want To Search:10
Item 10 Is Found
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:10 5 15 24 36 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:2
Which Item You Want To Delete:24
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:15 5 10 36 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:2
Which Item You Want To Delete:10
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
```

```
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:5 15 36 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:1
Enter Element To Be Insert:20
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:20 5 15 36 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:3
Which Item You Want To Search:50
Item 50 Is Not Found
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:20 5 15 36 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:3
Which Item You Want To Search:36
Item 36 Is Found
SPLAY TREE OPERATIONS
```

```
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:36 5 15 20 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:3
Which Item You Want To Search:20
Item 20 Is Found
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:20 5 15 36 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:2
Which Item You Want To Delete:15
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:4
Tree Elements Are:5 20 36 40
SPLAY TREE OPERATIONS
1.INSERTION
2.DELETION
3.SEARCHING
4.TRAVERSING
5.EXIT
Enter Your Choice:5
```

14.5 APPLICATIONS

In all the applications in which binary search trees are used, red–black trees can also be used because they are derived from B trees of order 4, a variant of binary search trees. In Java and C++, the library map structures are implemented with a red–black tree.

Splay trees are the self-adjusting binary search trees. Due to caching effect of the splay trees the recently accessed elements have better access times than that of the less frequently accessed elements. They are best suitable for the applications in which the recently accessed information is very likely to be accessed in the near future. The slay trees are typically used in caches and memory allocators. Other applications include a hospital management system, university information system, dynamic Hoffman coding and data compression.

SUMMARY

- A red–black tree is an extended binary search tree in which all the nodes and edges are coloured either red or black.
- A red–black tree should satisfy the following properties:
  The root node and all the external nodes are always coloured black.
  In every root node to an external node path no two consecutive red nodes should appear.
  The number of black nodes in all root node to the external node path must be the same.
- Insertion of an element into a red–black tree is the same as that of into a binary search tree. Only difference is that to the new node either black or red colour must be set.
- If the tree was empty then black colour may be set to the new node as it becomes the root.
- Deletion of an element from a red–black tree follows the same procedure as that of from a binary search tree. If the deleted node is red it will not violate the properties of red–black tree. So, no imbalance occurs and the tree is still a red–black tree.
- If the deleted node is black then it violates the third property of the red–black trees. So, the tree is unbalanced.
- The height of the red–black tree is maximum $2\log_2 (n+1)$. So, the operations search, insert and deletion will have a time complexity of O(log n).
- Splay trees are binary search trees with a wonderful property that can be adapted optimally to a sequence of tree operations.
- When any operation is performed on a node, it is moved towards the root node. In the due course the inactive elements are moved far away from the root node.
- Splay trees do not require balance or height information and hence are easier to implement and are cheaper than AVL trees over long sequences of retrievals.
- All the operations performed on splay trees are the same as that of on a binary search tree. But after performing the operation the tree is splayed with respect to the referred node.
- Splaying tree is aimed to make the referred or accessed node as the root node
- An amortized analysis analyzes a sequence of operations to show that though a single operation within the sequence may be expensive the average cost per operation is small.

EXERCISES

Fill in the Blanks

1. A red–black tree is an _____ tree.
2. If x is a right child of px, px is a left child of gx and the other child of gx is black, then it is an _____ type imbalance.
3. The root node and all the external nodes in a red–black tree are always coloured _____.
4. _____ and _____ invented data structures called a splay a tree.
5. Splay tree gives an amortized time complexity as _____.

Multiple-choice Questions

1. The height of the red–black tree is maximum _____.
   a. $2\log_2(n-1)$ b. $2\log_2(n+1)$ c. $\log_2(n+1)$ d. $2\log_2(n)$.
2. If x is a right child of px, px is a left child of gx and other child of gx is red then it is an _____ type imbalance.
   a. LLr b. LRr c. LRb d. None
3. The operations search, insert and delete on red–black trees have a time complexity of_____.
   a. O(log n) b. O(n) c. O(n log n) d. None
4. For sequence of m operations the amortized analysis of access in a splay tree is _____ when the time complexity of one access is O(n).
   a. O(m·log n) b. O(log m) c. O(log n) d. O(m·n)
5. The mechanism used to move a node up towards the root node is called _____ rotations.
   a. RLr b. RRb c. RRr d. Splay

Short-answer Questions

1. Define red–black trees.
2. Define splay trees.
3. Explain various rotations involved while inserting a node into the red–black tree.
4. What is an amortized analysis? Explain.
5. Explain about various splay rotations.

Essay Questions

1. Explain in detail about red–black trees.
2. Give the algorithms for various operations on red–black trees.
3. Discuss about splay trees.
4. Give the algorithms for insertion and deletion operations on splay trees.
5. Discuss an application where red–black trees are useful.

Chapter 15

Pattern Matching and Tries

Pattern matching and Tries are introduced along with terminology. The chapter discusses about Brute Force Algorithm, Boyer–Moore Algorithm and Knuth–Morris–Pratt Algorithm in detail along with examples and their applications. Categories of Tries namely standard tries, compressed tries and suffix tries and operations on them are explained clearly with applications.

15.1 INTRODUCTION

In the day-to-day life of a computer user string matching has greater importance. During text editing, the user processes the text, organizes the text into sections and paragraphs and reorganizes the text very frequently. The text is searched to find a subtext or pattern and replace it with some other text. Efficiency of the search algorithm is very important when searching for larger text unlike in the case of a dictionary, the algorithm does not depend only on the alphabetical order of words. For instance, in molecular biology the importance of string search algorithms is increasing day-by-day. Its usage is in locating some pattern in the string, comparing the sequences with usual subsequences and extracting information from DNA. With the assumption that a perfect match cannot be expected, is sort of processing should be done very often. Stringology handles this type of problems and its major area of interest is *pattern matching*.

A tree-based data structure that supports keys with varying sizes is known as a *Trie*. In search operation tries use some parts of the key. Trie belongs to the category of multi-way trees.

15.2 TERMINOLOGY

A pattern is a set of elements or recurring events. Elements can be a template or model which can be used to generate things or parts of a thing. A pattern is a uniquely identifiable event.

A *word* in the given text document is a pattern, and the pattern (word) is framed by means of putting alphabets in an order.

A *sentence* can be called as a pattern, and the pattern (sentence) is framed by means of putting words in an order.

An image of a human face from a photo can be called as a pattern, and the pattern (face of a human) is framed by means of putting pixels in an order.

So, the pattern is an ordered set of events like alphabets, words, pixels, etc. in the natural word.

Pattern matching: Pattern matching is a problem of checking for the presence of the given pattern in the given set of patterns. Suppose, to search for a pattern (word '*the*') in the given document by looking at all the words in the document, it can be concluded that the pattern '*the*' is present in the document or not. To state that the pattern '*the*' is found or not, the words which starts with an alphabet't' followed with 'h' and followed with 'e' must be observed. If such a structure is found in the document, then it can be said that the pattern '*the*' exists in the document.

Pattern recognition: Pattern recognition is the detected pattern over the set of patterns. If the pattern which is being searched for is found in the given document, then the pattern is recognized or else pattern is not recognized. If a human photo is given and asked to locate '*head*' of the human from the photo, as the pattern '*head*' is known and how the head will be, it is possible to locate that shape from the photo. If head is found in the photo, then the pattern head is recognized from the photo else the pattern is not recognized from the photo.

15.3 PATTERN MATCHING ALGORITHMS

Pattern matching algorithms are the procedures for matching or finding the given pattern on a set of patterns. Pattern matching algorithms are of two types:

- Fixed pattern matching algorithms
- Regular expression pattern matching algorithms

15.3.1 Fixed Pattern Matching Algorithms

If the given pattern for searching is constant until the search is complete, then the given search pattern is called a fixed pattern, and the algorithms used for pattern matching are called fixed pattern matching algorithms.

Example: Search for the number "21" over the set of patterns {12, 21, 23, 45, 56, 546, 58}. The pattern that is to be searched is 21 and is fixed.

15.3.2 Regular Expression Pattern Matching

Consider a problem of searching for the number which starts with the digit "1 or 2" over the set of patterns {12, 21, 23, 45, 56, 546, 58}. In this problem the search pattern varies, the recognized patterns in the set are 12, 21 and 23. In this problem the first character of the substring can be 1 or 2, which means that the first pattern is varying. In the fixed pattern search, the pattern characters will not vary in the entire search.

The regular expressions can be written with some special symbols. Every symbol in the regular expression will have a special meaning. A few important regular expression symbols are as follows:

Symbol	Description
\d	Matches with a single digit
\w	Matches with a single character
\W	Matches with a single non-word character
\s	Matches with a single space character
\S	Matches with a single non-space character
r*	Regular expression occurrence for zero or more number of times
r+	Regular expression occurrence for one or more number of times

For example, to search for the pattern 'a word followed with a space and followed with a number', then the regular expression becomes "\w+\s\d+".

Here, the regular expression \w matches a word character, because the regular expression is with + symbol. So, \w+ matches with one or more number characters. Then \s matches with a space character, and the \d+ matches with one or more number of digits.

15.4 FIXED PATTERN MATCHING ALGORITHMS

Various algorithms that are used to match a given pattern with the available patterns are discussed. Here, the pattern that is to be matched is stable all through the search process.

15.4.1 Brute Force Pattern Matching Algorithm

Brute force pattern matching algorithm is a good and simple algorithm for pattern matching. This algorithm finds all the possible substrings over the given large string, and checks for the given pattern on all the possible substrings.

If the pattern is with length 5, then the first substring from the text will be from the first character to the 5th character. The first substring will be cross checked with the pattern. If the substring and the pattern match, then the brute force algorithm returns with TRUE. If the substring and the pattern do not match, then the second substring will be from second character of the text to the 6th character of the text. The second substring will be cross checked with the pattern, if matches return TRUE or else the same process will be continued until the start index of the substring reaches n-length(P).

Algorithm 15.1: BruteForceMatch(Text, Pattern)

```
//n = length of Text and m = length of Pattern
Step 1: for i=0 to n-m do /*For each character of text, loop continues*/
Step 2: for j=0 to m do
   /*For each character of pattern, loop continues, used to frame text sub
      string with the length of the pattern*/
   /*If any character of text substring is not equal to the character
      sequence of pattern, Second for loop exits and first for loop continues
      with the next possible text substring*/
Step 3: if(Pattern[j]!=Text[i+j]) break;
   done
Step 4: if(j==m) then return TRUE
   /*If any text substring pattern characters fully matches with the pattern,
      function returns true as matched*/
   done
Step 5: Return FALSE
   /*If all the text substring pattern characters fully match with the
      pattern, function returns false as not-matched*/
```

Here, n is length of the text string (T) and m is the length of the pattern string (P). In the first step, the algorithm estimates the possible number substrings with the length of pattern string over the text string, i.e. (n–m). The second step or loop identifies all the possible strings. If all the characters of the pattern string match with the substring characters, then the algorithm returns TRUE and exits else continues to check the next substring and is cross checked against the pattern string. This process continues for all the substrings. If the given pattern matches with any of the substrings the algorithm exits by returning TRUE, i.e. pattern match message.

Consider a text T = "abcdefgh", to search for the pattern P = "cde" the Brute Force Algorithm works as follows:

T ="abcdefgh"
P ="cde"

1. Finds all the possible substrings over the text (T).
   a. "abc"
   b. "bcd"
   c. "cde"
   d. "def"
   e. "efg"
   f. "fgh"
2. Checks for the pattern P = "cde" on all the possible patterns
   a. "abc" compares with "cde", and the pattern is not matched
   b. "bcd" compares with "cde", and the pattern is not matched
   c. "cde" compares with "cde", and the pattern is matched, exits from this point saying that the pattern is matched.
   d. "def"
   e. "efg"
   f. "fgh"

This algorithm searches character by character. It will not search the entire string at a time.

Program 15.1

```
#include<iostream.h>
#include<string.h>
#include<iomanip.h>
void main()
{
char Text[256],Pattern[50];
int m,n;
cout <<"\nEnter Text, At end enter'.'\n";
for(int i=0;i<256;i++)
{
char ch;
cin.get(ch);
if(ch!='.')
   {
   Text[i]=ch;
   }
else break;
}
n=strlen(Text);
cout<<"\nEnter Pattern For Search:";
cin>>Pattern;
m=strlen(Pattern);
for(i=0;i<=(n-m);i++)
   {
```

```
    for(int j=0;j<m;j++)
       {
       if(Pattern[j]!=Text[i+j])break;
       }
    If (j==m)
       {
       cout<<"\nThe Pattern\'"<<Pattern<<"\'Found at"<<i<<"on the given Text\""
<<Text<<"\"";
       }
    }
}
```

Output

```
Enter Text, At end enter '.'
DOGS DO NOT SPOT HOT POTS OR CATS.

Enter Pattern For Search: SPOT
The Pattern 'SPOT' Found at 12 on the given Text "DOGS DO NOT SPOT HOT POTS OR
CATS"
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
D	**O**	**G**	**S**		**D**	**O**		**N**	**O**	**T**		**S**	**P**	**O**	**T**		H	O	T		P	O	T	S		O	R		C	A	T	S
S	P	O	T																													
	S	P	O	T																												
		S	P	O	T																											
			S	**P**	O	T																										
				S	P	O	T																									
					S	P	O	T																								
						S	P	O	T																							
							S	P	O	T																						
								S	P	O	T																					
									S	P	O	T																				
										S	P	O	T																			
											S	P	O	T																		
												S	P	O	T																	

Figure 15.1 Execution trace of Brute Force Algorithm

In Figure 15.1, the bold characters are cross checked for matching. In the first iteration the character of the Text "D" and the first character of the pattern "S" are cross checked, because the match is failed, the algorithm went for the second iteration. In the second iteration the characters "O" and "S" are cross checked. In this iteration also match failed so that the algorithm went for the third, fourth and so on iterations. In the fourth iteration the match of the first character "S" and the first character of the pattern "S" are matched, and "P" and " " (space) do not match. In the 12th iteration all the characters are matched. So, the Brute Force Algorithm returns the offset 12 as the return value.

Performance analysis of Brute Force Algorithm: Brute Force Algorithm performs well except for the worst case. In the worst case, the algorithm searches for O(mn) times because the algorithm performs 'm' character comparisons to confirm that the pattern does not match.

15.4.2 The Boyer–Moore Algorithm

The Brute Force Algorithm finds all the possible substrings that can be matched with the search pattern, and searches all the characters of the given text. The Boyer–Moore (B–M) Algorithm is a faster algorithm when the search string is large. It is one of the best suitable algorithms for searching the words in the text document. Unlike the Brute Force algorithm, the Boyer–Moore Algorithm skips the unnecessary checks.

The B–M Algorithm works with a 'backward' approach, the target string is aligned with the start of the check string, and the last character of the target string is checked against the corresponding character in the check string. In the case of a match, then the second-to-last character of the target string is compared to the corresponding check string character. In the case of a mismatch, the algorithm computes a new alignment for the target string based on the mismatch. In the case of mismatch the algorithms skip checking some of the characters.

Boyer–Moore Algorithm steps: Character comparison is done from right of the pattern to the left:

I. Constructing the SHIFT Table
II. GOOD SUFFIX SHIFT or BAD CHARACTER SHIFT

Constructing the SHIFT table: An ASCII table with all the ASCII 256 characters is constructed and an ASCII character associated with the number of shifts. Initially the table is filled with the length of the pattern to be matched. Later the number of shifts of the characters of the pattern will be modified with the descending count from the original length of the pattern in each character that occurs.

For example, the search pattern is "ALGORITHM"; in the initial step the snapshot of the ASCII table will be as below:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9

The characters of the search pattern will be filled with the descending count from the length of the pattern.

A	L	G	O	R	I	T	H	M
8	7	6	5	4	3	2	1	

So, the SHIFT table becomes as below:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
8	9	9	9	9	9	6	1	3	9	9	7	9	9	5	9	9	4	9	2	9	9	9	9	9	9

Shifting of the pattern is done when the mismatch occurs with the character being compared.

Good suffix shift: The character of the pattern to be searched and the character that is being compared is not equal and the character being compared is there in the characters' set of the pattern; the pattern is moved forward by the number of shifts present in the SHIFT table.

Bad character shift: The character of the pattern to be searched and the character is being compared is not equal and the character being compared is not there in the characters' set of the pattern; the pattern is moved forward by the number of shifts present in the SHIFT table (i.e. length of the pattern). This shift avoids unnecessary character comparisons.

Boyer–Moore Algorithm

```
/*    T is a Text string
      P is a pattern string
```

```
        ∑ is the set of pattern symbols or characters.
*/
```

Algorithm 15.2: BM (T, P, Σ)

```
last←lastFunction(P,Σ) /*last function prepares the shift table*/
i←m-1
j←m-1
repeat
if T[i]=P[j]
if j=0
return i /*pattern match found at i*/
else
i←i-1
j←j-1
else    /*character-jump*/
l←last[T[i]]
i←i+m-min(j,1+l)
j←m-1
until i>n-1
return-1 /*pattern not found*/
```

Boyer–Moore Algorithm with an example:

```
Text    : "THIS IS A TEST OF BOYER MOORE ALGORITHM"
Pattern : "ALGORITHM"
```

In the first step SHIFT table is filled with all the ASCII characters associated with the length of the pattern "ALGORITHM" (i.e. 9). In the second step, the SHIFT table is modified with the descending count from the length of the pattern.

The SHIFT table consists of the ASCII characters associated with the number of shifts. The matching of the characters starts from the rightmost character of the pattern.

Pass 1: In this pass, the **text** character 'A' and the **pattern** character 'M' are compared. In this case the comparison fails. But the character 'A' is present in the set of characters of the pattern. This case is referred to as the Good Suffix Shift. In the shift table, the number of shifts for the character 'A' is 8. So, the pattern 'ALGORITHM' is moved 8 characters forward.

A	L	G	O	R	I	T	H	M
8	7	6	5	4	3	2	1	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	
T	H	I	S		I	S		A		T	E	S	T		O	F		B	O	Y	E	R		M	O	O	R	E		A	L	G	O	R	I	T	H	M	
A	L	G	O	R	I	T	H	M																																PASS 1
								A	L	G	O	R	I	T	H	M																								PASS 2
																A	L	G	O	R	I	T	H	M															PASS 3
																								A	L	G	O	R	I	T	H	M						PASS 4
																										A	L	G	O	R	I	T	H	M				PASS 5

Figure 15.2 Execution trace of Boyer–Moore Algorithm

Pass 2: In this pass, the **text** character 'F' and the **pattern** character 'M' are compared. In this case the comparison fails. The character 'F' is not present in the set of characters of the pattern. This case is referred to as the Bad Character Shift. In the shift table, the number of shifts for the character 'F' is 9. So, the pattern 'ALGORITHM' is moved 9 characters forward.

Pass 3: In this pass, the **text** character 'O' and the **pattern** character 'M' are compared. In this case the comparison fails. But the character 'O' is present in the set of characters of the pattern. In the shift table, the number of shifts for the character 'O' is 5. So, the pattern 'ALGORITHM' is moved 5 characters forward.

Pass 4: In this pass, the **text** character 'A' and the **pattern** character 'M' are compared. In this case the comparison fails. But the character 'A' is present in the set of characters of the pattern. This case is referred to as the Good Suffix Shift. In the shift table, the number of shifts for the character 'A' is 8. So, the pattern 'ALGORITHM' is moved 8 characters forward.

Pass 5: In this pass, the **text** character 'M' and the **pattern** character 'M' are compared. In this case the comparison succeeded. So, the algorithm compares the character of the Text 'H' and the character of the pattern 'H' is compared. So, again the immediate previous characters of the pattern and the text are compared until all the characters are compared. Because all the characters are matched, the index of the last character match will be returned.

Program 15.2

```
//Boyer-Moore
#include<conio.h>
#include<string.h>
#include<iostream.h>
#include<stdio.h>
int*build_table(char*p)
{
int last[128];
int i;
for(i=0;i<128;i++)last[i]=-1;
for(i=0;i<strlen(p);i++)
   {
      last[p[i]]=i;
   }
return(last);
}

int min(int a,int b)
{
if(a>b)return(b);
else return(a);
}
int search(char*t,char*p)
{
int*last=build_table(p);
int n=strlen(t);
int m=strlen(p);
```

```
int i=m-1;
if(i>n-1)return(-1);
int j=m-1;

      do{
   if(p[j]==t[i])
      if(j==0)return i;
      else   {
         i--;
         j--;
         }
   else  {
      i+=m-min(j,1+last[t[i]]);
      j=m-1;
      }
   }while(i<=n-1);
   return(-1);
}

void main()
{
char T[128],P[128];
int index;
clrscr();
cout << "\nEnter Text:";
gets(T);
cout <<"\nEnter Pattern:";cin>>P;
index=search(T,P);
if(index!=-1)
   {
   cout<<"\nMatch Found At:"<<index;
   }
else  {
   cout<<"\nMatch Not Found";
   }
}
```

Output

```
Enter Text: Sample Text
Enter Pattern: ple
Match Found At: 3
```

15.4.3 Knuth–Morris–Pratt Algorithm (KMP)

This algorithm is best suitable when the pattern is in the redundant characters. This algorithm reduces the duplicate comparisons in the inner loops of the process. If the B–M Algorithm discovers a pattern character that does not match in the text, then the algorithm throws the matched information or knowledge away. KMP Algorithm uses those matched or knowledge about the pattern to avoid unnecessary checks.

This algorithm works with a *failure function*. This failure function is used to find the exact shift of P. So, the algorithm can reuse the previous performed comparisons.

Failure function: The failure function f(j) can be defined as the length of the longest prefix of P that occurs as a suffix of P[1..j]. The failure function starts by assigning f(0)=0. The failure function calculates the exact shift location of P.

KMP ALGORITHM

```
/*
T is a Text string
P is a pattern string.
*/
```

ALGORITHM 15.3: KMP(T[1 .. n], P[1 ..m])

```
j←1
for i←1 to n
while j>0 and T[i]≠P[j] do
j←fail[j]
if j=m   //Pattern Found
return (i-m+1)
else
j← j+1
return 'There is no substring of T matching P.'
```

FAILURE_FUNCTION(P[1 ..m])

```
j← 0
f(0)←0
for i←1 to m
if P[i]=P[j]
fail[i]←fail[j]
else
fail[i]←j
while j>0 and P[i]≠P[j]
j←fail[j]
j←j+1
```

In the KMP matching algorithm, at the same time the 'i' and 'j' are incremented, or 'j' is decremented and the value of 'i' is kept unchanged.

Program 15.3

```
// Knuth-Morris-Pratt
#include<iostream.h>
#include<string.h>
#include<conio.h>
int n,m,fail[256];
char T[512],P[256];
void FailureFunction()
```

```
{
int i,j;
fail[0]=0;
m=strlen(P); //Length of Pattern String
j=0;
i=1;
while(i<m)
   {
   if(P[j]==P[i])
      {
      fail[i]=j+1;
      i++;
      j++;
      }
   else   if(j>0)j=fail[j-1];
          else {
          fail[i]=0;
          i++;
        }
   }
}
void main()
{
char ch;
int i,j,flag=0;
clrscr();

//Reading of Text
cout<<"\nEnter Text (At End press dot(.)):";
i=0;
while((ch=getche())!='.')
   {
   T[i]=ch;
   i++;
   }
T[i]='\0';
cout<<"\nText:\n" << T;

//Reading of Pattern
cout<<"\nEnter Pattern (At End press dot(.):";
i=0;
while((ch=getche())!='.')
   {
   P[i]=ch;
   i++;
   }
P[i]='\0';
cout<<"\nPattern:\n" << P;
```

```
n=strlen(T);
m=strlen(P);
FailureFunction();
i=j=0;
while(i<n)
   {
   if(P[j]==T[i])
      {
      if(j==(m-1))
         {
         cout<<"\nPattern Found AT: "<<(i-m+1);
         flag=1;
         }
      i++;
      j++;
      }
   else if(j>0) j=fail[j-1];
      else i++;
   }
if(flag==0)
   {
   cout<<"\nPattern Not Found";
   }
}
```

Output

```
Enter Text (At End press dot(.)):ababacab.
Text:
ababacab
Enter Pattern (At End press dot(.):abacab.
Pattern:
abacab
Patten Found AT:2
```

The above-discussed algorithms are best suitable for comparing the single time pattern search over a short length of text string.

15.5 APPLICATIONS OF PATTERN MATCHING ALGORITHMS

Various applications of pattern matching algorithms are discussed below.

Search engines: Most efficient pattern matching algorithms are needed for the information retrieval from the internet. A search engine examines all the domains, extracts the Web pages from the Web sites and prepares possible patterns (indexes) that the users of the search engine expect.

For example, if the pattern "pattern matching algorithms" is to be searched, the search engines have to give the reply within seconds. But, the whole Web data will be in millions of tera bytes. Searching for the given pattern on bulk data within the short span of time is really a challenging thing. The search engine service providers do the offline processing for finding the possible sequence of patterns and keep the index terms for the

possible Web pages. For searching the index words or patterns from the bulk data, the search engine service providers need highly sophisticated pattern matching algorithms. The pattern matching algorithms are also needed for providing the relevant answers to the search queries.

Database management systems: The database management systems contain data and the interface for the data such as query language interfaces. If any query has to be processed on the data, the DBMS should have an efficient pattern matching algorithm. Because database systems organize bulk of data, the query processing system of the database management system should have an efficient pattern matching algorithm.

Signature-based intrusion detection systems: Intrusion Detection Systems (IDS) are the systems that provide security for the corporate network. Most of the IDS work with signature logic. Here signatures are nothing but the attack patterns. The IDS maintain a large number of signatures to detect various kinds of attacks. If any packet arrives to the IDS, the system has to cross check with available signatures. If any signature matches against the arrived packet, then the IDS alerts the administrator. For cross checking the signatures (attack patterns) against the arrived packet, pattern matching algorithms are required.

Text editors: The text editors like notepad provide facilities like adding text, modifying text, deleting the text and an interface for searching patterns on the text file to the users. For searching text (pattern) on the text file, the pattern matching algorithms are required.

15.6 TRIES

A **trie** pronounced as 'try' is a tree-based data structure, stores the large text string as a tree for fast pattern matching. The word "trie" came from the word "retrieval". When multiple patterns are to be searched over a large text string, the tries are best suitable.

Unlike binary trees, the tries can have any number of children. The pattern matching starts by matching the prefix of the text string to be compared. Trie is a tree-like structure, the shape of the trie depends on the data nodes that the trie holds.

Different types of tries:

- Standard Tries
- Compressed Tries
- Suffix Tries

15.6.1 Standard Tries

In the standard tries, each character is represented as a node in the tree. Standard trie starts with a node called root node. If a set of strings (say 'S') is to be processed for the pattern matching with multiple patterns, then the standard trie preprocesses (stores the strings) as a tree. If all the sets of strings of 'S' do not have the different prefix characters, then the tree becomes an ordered tree.

This approach avoids the multitime character matching. The procedure returns the index occurrences of the pattern by a single time match. It uses less storage space and a fast pattern search. Standard tries are suitable for the application like search for a word in a document.

Let a set of strings S = {ant, animal, blew, blue} is to be searched for the specified pattern, in the preprocessing step the set 'S' is arranged as a tree as shown in Figure 15.3.

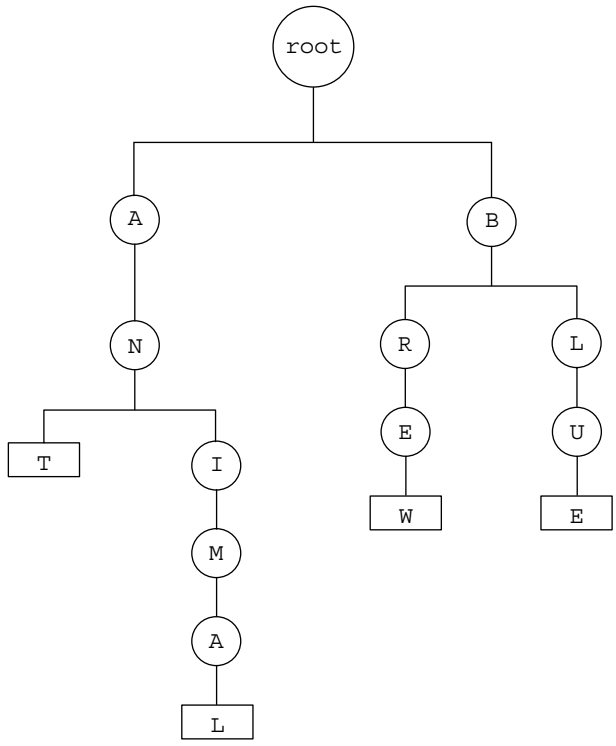


Figure 15.3 Example of standard tries

If the pattern "BLUE" is to be searched the search starts from the **root** node, passes through the node "B", "L", "U", and finally "E". Because of the pattern found in the tree the standard trie returns the offset of the "B".

The standard tries can also be called digital search trees. These tries operate at the character level or bit-level.

If any node is to be inserted then find whether the root node is empty or not. If the root node is empty, then create a node and treat that node as the root node. This is done by creating a node, storing the node value in the node and inserting the node at the first level. If the node that is to be inserted is already present in the tree then insert the next node.

Disadvantages of standard tries:

- Uses more number of nodes than other tries (space complexity is more than that of other tries)
- Not suitable for matching non-word patterns, such as string with white space
- Can compare a single character at a time

Node structure of a standard trie: The structure of the node of the standard trie will have 256 symbol elements array and 256 links or branches. Each node will be capable of handling any character, printable or non-printable characters.

If the trie is limited to only upper case alphabets, the number of symbol elements will be 26 and branches or links will be 26.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Link	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Node structure:

```
struct trie{
   struct trie *link[26];
   };
```

In the initial stage, the node will have null values at all the link locations. If any subtrie needs to be added to the trie node, then an appropriate link location will be filled with the address of the subtrie.

In the trie trees, each node of the trie will act as the node of a state machine. On input symbol, the path or the link will be selected. On any input symbol, branch index or the link location can be easily identified by referring:
Node. Link [Decimal Equivalence of ASCII Index of Input Symbol].

Algorithm 15.4: Insert (Trie T, String S)

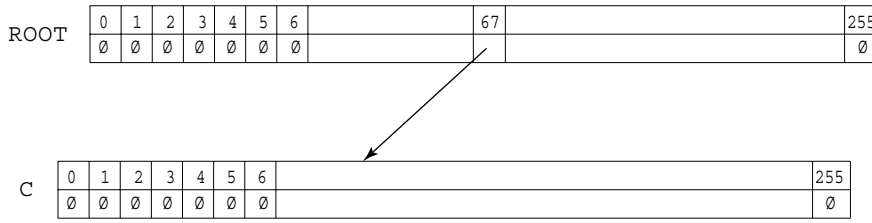
```
1. Index ← s[0];
2. Check the string 'S' is empty or not.
   a. If the string is empty then no more characters are to be inserted.
   b. Else
      1. If T.link[index]==NULL then "The Character is new character, and
         it is to be inserted".
         a. Create and empty node 'temp'
         b. T.link[index] ← Address of the node 'temp'
         c. T.symbol[index] ←s[0]
         d. S++ /*Move the address pointer of string 'S'to the next location.*/
         e. T ← T.link[index] /*Move to Sub Trie location*/
         f. Call to insert(T,S) /*Insert remaining characters of the string
            at Sub Tries T.*/
      2. Else  /* Similar Node Exists */
         a. S++  /*Move the address pointer of string'S'to the next location.*/
         b. T ← T.link[index]  /* Move to Sub Trie location*/
         c. Call to insert(T,S)  /* Insert remaining characters of the string
            at Sub Tries T.*/
```

Insertion with an example: To insert the strings "CAR" and "CAB" into the trie,

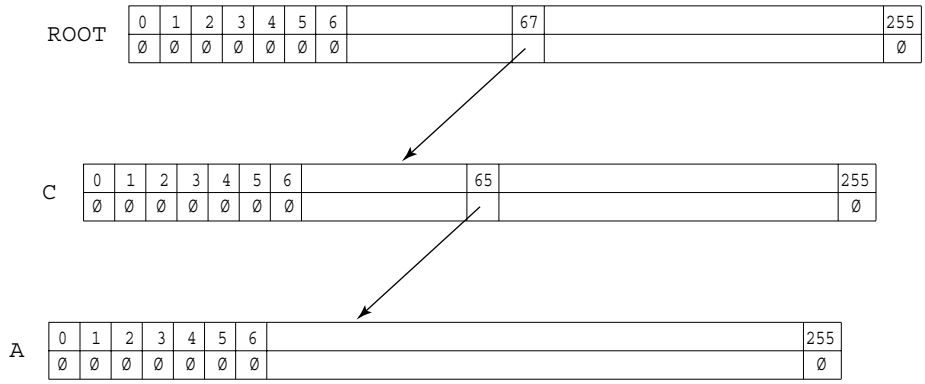
1. String S="CAR".
2. Initially the root node is empty the node will be created with null values.

ROOT	0	1	2	3	4	5	6		255
	Ø	Ø	Ø	Ø	Ø	Ø	Ø		Ø

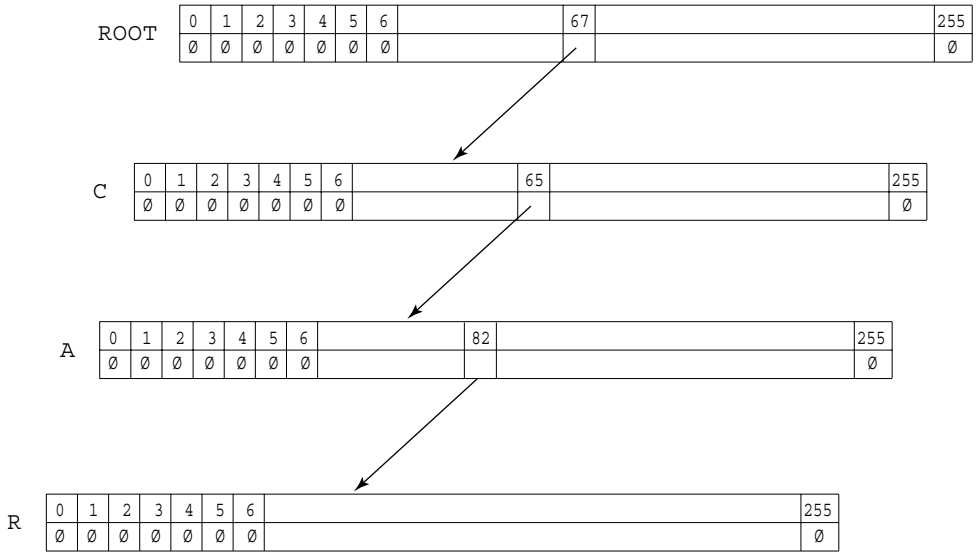
3. Start by inserting the first character "C".
   a. Index=ASCII Code of Character 'C', i.e index=67.
   b. Because root->link[67]== NULL, the character 'C' is not there at the root level.
   c. Create a new empty trie node, fill the link at the root node as the address of the newly created empty trie node. So, the insertion of "C" is completed.



d. Consider the second character, "A". Because the "C" node does not have any branches, create a new node and again fill the new node with the symbol "A" and fill the C branch or link [65] as node address "A".



e. Consider the third character, "R". Because the "A" node does not have any branches, create a new node and again fill the new node with the "A" branch or link [82] as node address "R".



4. While inserting the string "CAB", at first "C" will be searched at the root node.
   a. Because "C" is available, move to the node "C".
   b. At node "C", check for the node "A", because link to the node at "A" is not null, move in that path.
   c. Check for the node "B", because the node, "B" is not available, the node "B" will be treated as a new node, and inserted under "A" as a branch node.

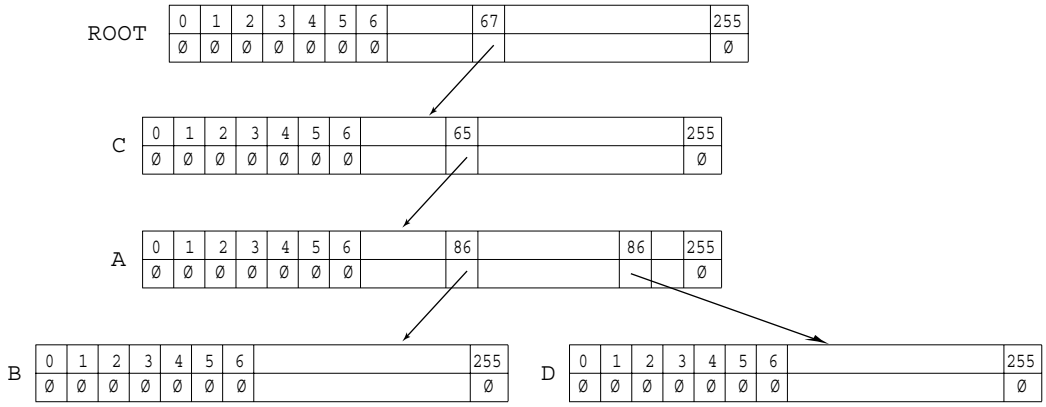


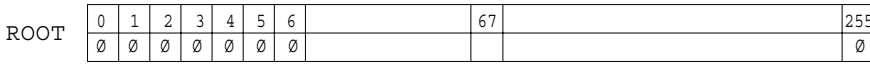
Figure 15.4 Insertions of CAB and CAR

Algorithm 15.5: Search (Trie T, String S)

```
1. Find the index of the node.
   a. Index←ASCII Code S[0]
2. If length(S) and S[0]=='\0'
   a. Return "STRING FOUND"
3. Else
   a. If T->link[index]==NULL
      i. Return "String Not Found".
   b. Else  /*Path Found, Follow the link*/
      i.    S++
      ii.   T=T->link[index]
      iii.  Search(T,S);
```

Search algorithm with an example: The search procedure takes the address of a trie (T) and a search string (S) as input arguments. Let a search string S="CAB" and trie (T) as the root node of Figure 15.4 be considered, then the search procedure will be as below:

Pass 1: Checking for the node 'C'
S = "CAB" and
Trie T=



1. Find the index of the node.
   Index = ASCII[S[0]]
   = ASCII['C']
   = 67

2. The length of (S) and S[0] are not null goto Step 3
3. Check whether the T->link[67] is 'NULL'
   a. Because it is not null goto 'Step b'
   b. Set S = 'AB'
   c. Trie(T) = 'C' Sub Trie

Pass 2: Checking for the node 'A'
String S = "AB" and
Trie (T) =

C

0	1	2	3	4	5	6		65		255
Ø	Ø	Ø	Ø	Ø	Ø	Ø				Ø

1. Find the index of the node.
   Index = ASCII[S[0]]
   = ASCII['A']
   = 65
2. The length of (S) and S[0] are not null goto Step 3
3. Check whether the T->link[65] is 'NULL'
   a. Because it is not null goto 'Step b'
   b. Set S ='B'
   c. Trie(T) = 'A' Sub Trie

Pass 3: Checking for the node 'B'
String S = "B"
Trie (T) =

A

0	1	2	3	4	5	6		66		82		255
Ø	Ø	Ø	Ø	Ø	Ø	Ø						Ø

1. Find the index of the node.
   Index = ASCII[S[0]]
   = ASCII['B']
   = 66
2. The length of (S) and S[0] are not null goto Step 3
3. Check whether the T->link[66] is 'NULL'
   a. Because it is not null goto 'Step b'
   b. Set S = '\0' (Null)
   c. Trie(T) = 'B' Sub Trie

Pass 4: Final pass
String S = "\0" [i.e. null string]
Trie T =

B

0	1	2	3	4	5	6		255
Ø	Ø	Ø	Ø	Ø	Ø	Ø		Ø

1. Find the index of the node.
   Index = ASCII[S[0]]
   = ASCII['\0']
   = 0

2. The length of (S) and S[0] is null. So, all the nodes found in the trie, print the "String found" message and exit.

Algorithm 15.6: Deletion (Trie T, String S)

1. Check for the existence of the string 'S' over the Trie 'T'
2. ch ←S[0];
3. Push(Find_the_node_address(ch)); /*insert the trie node address into the stack*/
4. For each element of the stack
   a. Check for the number of branches
      i. If no_branches > 1
         1. Delete the branch[node index];
         2. Return;
      ii. Else if(no_branches ==1)
         1. Delete the element.
            Else
               Delete (element);

Deletion algorithm with an example: The deletion algorithm takes the Trie (T) and the String (S) as input arguments. Let us delete "CAR" from the Trie as shown in Figure 15.4.

Step 1: Check for the String S="CAR" present in the trie=ROOT by calling prepare_stack procedure. If not found, deletion is not possible else goto Step 2. The prepare_stack procedure also creates a stack as in Figure 15.5.

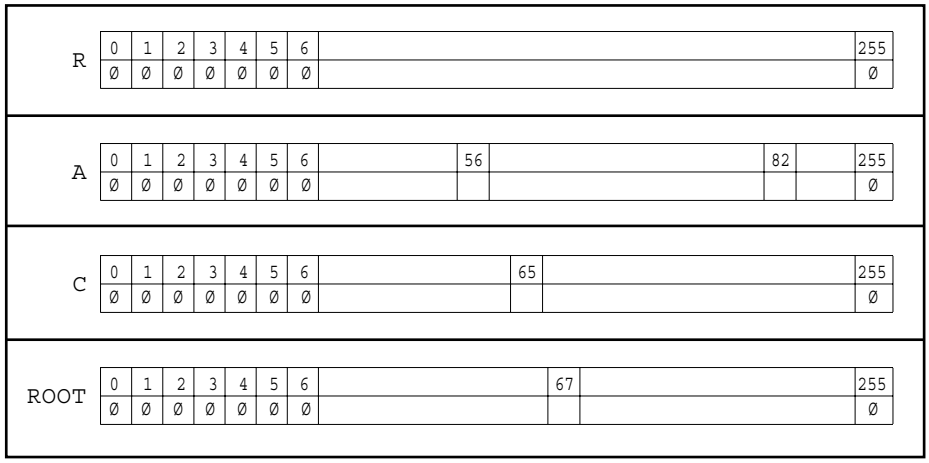


Figure 15.5 Stack of nodes to be deleted

Step 2: For each node or element of the stack, check for the number of branches:

1. The top of the stack is "R" and has all the null branches. So, delete the branch.
2. The top of the stack is now "A" and has two branches. So, fill the "R" link information in the node "A" with null. Complete deletion of this node is not possible, because other than the deleted branch another branch exists in the node "A".

Then trie becomes as shown in Figure 15.6.

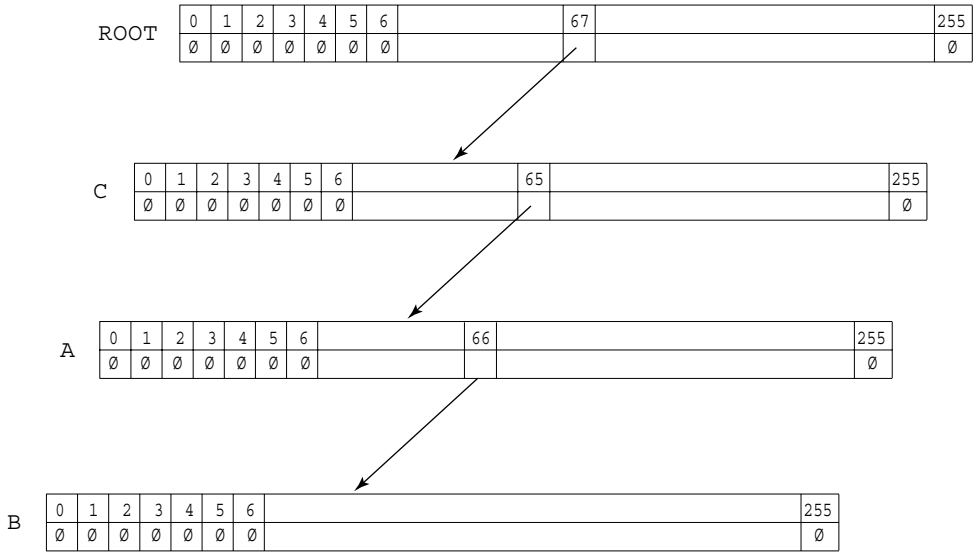


Figure 15.6 Trie after deletion of CAR

Program 15.4

```
/ / Standard Trie Operations
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#include<alloc.h>
#define ASCII 256
struct trie{
struct trie*link[ASCII];
};
struct stack{
struct trie*element;
struct stack*next;
};

class stries{
public:
struct trie*create_node();
int search(struct trie*,char*);
int prepare_stack(struct trie*,char*);
int push(struct trie*);
int insert(struct trie*,char*);
int deletion(struct trie*,char*);
```

```
void display();
struct trie*root;
struct stack*front;

   stries()   //Constructor to initialise root
   {
   root=create_node();
   front=NULL;
   }
};

struct trie*stries::create_node()
{
int i;
struct trie*t=(struct trie*)malloc(sizeof(struct trie));
for(i=0;i<ASCII;i++)
   {
   t->link[i]=NULL;
   }

return(t);
}

int  stries::insert(struct trie*T,char*s)
{
int index=s[0];

if(s[0]=='\0')
{
cout<<"\nInsertion Completed.\n\n";
return(1);
}
cout<<"\nTrying to Insert"<<s[0];
if(T->link[index]==NULL)
   {
   cout<<"Node Not Found, Creating Branch";
   T->link[index]=create_node();
   s++;
   T=T->link[index];
   insert(T,s);
   }
else {
   cout<<"Node Found,Moving in the path";
   s++;
   T=T->link[index];
   insert(T,s);
   }
return(1);
}
```

```
void stries::display()
{
cout<<"\n1.Insert";
cout<<"\n2.Delete";
cout<<"\n3.Search";
cout<<"\n4.Exit";
cout<<"\nEnter Your Choice:";
}

int stries::search(struct trie*T,char*s)
{
int index;
   if(s[0]=='\0') //Check for the empty search string.
      {
      cout<<"\nString Found";
      return(1);
      }
   index=s[0];

   if(T->link[index]==NULL)
      {
      cout<<"\nString Not Found";
      return(0);
      }
   else {
      cout<<"\n"<<s[0]<<"=Node Found,Moving in the path";
      s++;
      T=T->link[index];
      search(T,s);
      }
return(1);
}

int stries::deletion(struct trie*T,char*s)
{
struct stack*i,*t;
int k,flag,index;
char ch;
if(prepare_stack(T,s)==1)
   {
   cout<<"\n"<<s<<"String Found in the Trie";
   }
else {
   cout<<"\nDeletion not possible, because string does not exists";
   return(0);
   }
cout<<"\nInverted Trie\n";
index=strlen(s)-1;
```

```
for(i=front;i!=NULL;i=i->next)
   {
   flag=0;
   for(k=0;k<ASCII;k++)
      {
      if(i->element->link[k]!=NULL)
         {
         cout <<(char)k<<"-";
         flag++;
         }
      }
   cout<<endl;
   if(flag>1)
      {
      cout<<"\nFound Other nodes in the trie, Deleting the element"<<endl;
      free(i->element->link[(int)s[index]]);
      i->element->link[(int)s[index]]=NULL;
      return(1);
      }
   else if(flag==1)
      {
      cout<<"\nFound Exact Node";
      free(i->element->link[(int)s[index]]);
      i->element->link[(int)s[index]]=NULL;
      index--;
      }
      else{
         free(i->element);
         }
   }
front=NULL;
return(1);
}

int stries::prepare_stack(struct trie*T,char*s)
{
int index;
   if(s[0]=='\0')
   {
   cout<<"\nString Found";
   push(T);
   return(1);
   }
index=s[0];
if(T->link[index]==NULL)
   {
   cout<<"\nString Not Found";
   return(0);
   }
```

```
else {
   cout<<"\n"<<s[0]<<"=Node Found,Moving in the path";
   push(T);
   s++;
   T=T->link[index];
   prepare_stack(T,s);
   }
return(1);
}

int stries::push(struct trie*T)
{
struct stack*t;
if(front==NULL)
   {
   front=(struct stack*)malloc(sizeof(struct stack));
   front->element=T;
   front->next=NULL;
   }
else{
   t=(struct stack*)malloc(sizeof(struct stack));
   t->element=T;
   t->next=front;
   front=t;
   }
return(1);
}

void main()
{
clrscr();
char s[ASCII];
stries obj;
int ch;
while(1)
{
obj.display();
cin >>ch;
switch(ch)
   {
case 1:cout<<"\nEnter Word to Insert:";
   cin >>s;
   obj.insert(obj.root,s);
   break;
case 2:cout<<"\nEnter Word to Delete:";
   cin >>s;
   obj.deletion(obj.root,s);
   break;
```

```
case 3:cout<<"\nEnter Word to Search:";
   cin>>s;
   obj.search(obj.root,s);
   break;
case 4:return;
default:return;
   };
 }
}
```

Output

```
1. Insert
2. Delete
3. Search
4. Exit
Enter Your Choice: 1
Enter Word to Insert: CAR
Trying to Insert C Node Not Found, Creating Branch
Trying to Insert A Node Not Found, Creating Branch
Trying to Insert R Node Not Found, Creating Branch
Insertion Completed.
1. Insert
2. Delete
3. Search
4. Exit
Enter Your Choice: 1
Enter Word to Insert: CAT
Trying to Insert C Node Found, Moving in the path
Trying to Insert A Node Found, Moving in the path
Trying to Insert T Node Not Found, Creating Branch
Insertion Completed.
1. Insert
2. Delete
3. Search
4. Exit
Enter Your Choice: 3
Enter Word to Search: CAR
C = Node Found, Moving in the path
A = Node Found, Moving in the path
R = Node Found, Moving in the path
String Found
1. Insert
2. Delete
3. Search
4. Exit
Enter Your Choice: 2
Enter Word to Delete: CAR
C = Node Found, Moving in the path
```

```
A = Node Found, Moving in the path
R = Node Found, Moving in the path
String Found
CAR String Found in the Trie
Inverted Trie
R-T-
Found other nodes in the trie, deleting the element
1. Insert
2. Delete
3. Search
4. Exit
Enter Your Choice: 3
Enter Word to Search: CAR

C = Node Found, Moving in the path
A = Node Found, Moving in the path
String Not Found
```

15.6.2 Compressed Tries

Compressed tries or Patricia tries are similar to the standard tries and are like the tree structure, but the shape of the node depends on the data it holds. But a node in the standard trie will have only one character, but a compressed trie can have multiple characters. Compressed tries will have at least two children at each of its internal nodes.

Let a standard trie 'T' have an internal node 'v' and 'v' is not a root node, and 'v' has not only one branch. Then all the child nodes of the node 'v' are transformed or compressed as a single node.

Let 'v' have a single branch with 'k' number of child nodes and each child node does not have more than one child, then all the child nodes will be compressed as a single node.

If V = (v0,v1) (v1,v2)(v2,v3)(v3,v4)….(vk-1, vk), where v1 has a child v2, v2 has a child v3, v3 has a child v4 ... vk–1 has a child vk then vi is redundant for i=1,2,3,4 ... k–1, v0 and vk are not redundant. The node vkk is compressed with the concatenation of labels of v1…vk (Figure 15.7).

Properties of compressed tries: Let S be a set of strings with an alphabet set of size d, which will have the following properties:

- All the internal nodes in the compressed trie should have at least two children and at most d children.
- If all the strings of the set 'S' do not have the same start suffix character then the compressed trie should have 's' number of external nodes where 's' is the number of strings in the 'S'.
- The number of nodes of 'T' is O(s).

Disadvantages of compressed tries:

- Inefficient when too many duplicates are present in the patterns.
- Each node occupies at least one byte
- Updating the node is difficult

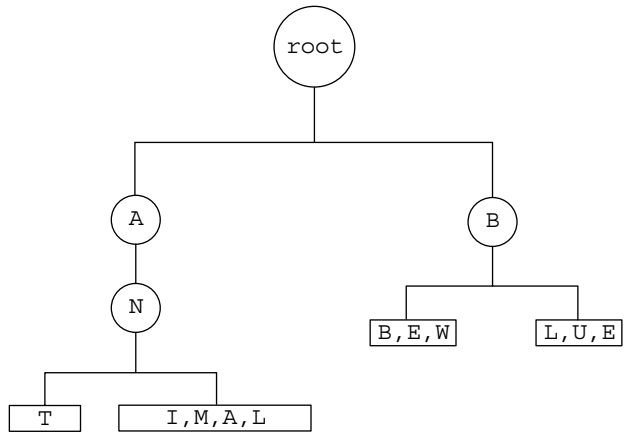


Figure 15.7 Example of a compressed trie

15.6.3 Suffix Tries

Suffix trie is a tree which has the suffix symbols of each string from a branch. Suffix trie can be called a position tree or a suffix tree. This procedure is better suited for finding the largest repetitive substring from the given large pattern string. This procedure completely avoids the unnecessary comparisons.

Example 1: Construction of suffix trie with an example
Let the Pattern P = "banana"

Step 1: Find the all possible substrings
T1 = banana
T2 = anana
T3 = nana
T4 = ana
T5 = na
T6 = a

Step 2: Sort all the substrings alphabetically.
T6 = a
T4 = ana
T2 = anana
T1 = banana
T5 = na
T3 = nana
Step 3: Find the prefixes shared by the substrings.
Here the prefix of T2 is shared by T6 and T4. T1 is the only node that starts with the prefix, and finally the prefix of T3 is shared by T5.

So, the found major possible strings are T2, T1 and T3. So, the tree will be as shown in Figure 15.8.

Step 4: Construct the tree

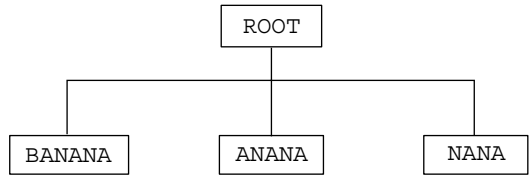


Figure 15.8 Example of a suffix trie

Example 2:

Let the Pattern P = "abracadabra"

Step 1: Find all the possible substrings

T1 = abracadabra
T2 = bracadabra
T3 = racadabra
T4 = acadabra
T5 = cadabra
T6 = adabra
T7 = dabra
T8 = abra
T9 = bra
T10 = ra
T11 = a

Step 2: Sort all the substrings alphabetically

T11 = a
T8 = abra
T1 = abracadabra
T4 = acadabra
T6 = adabra
T9 = bra
T2 = bracadabra
T5 = cadabra
T7 = dabra
T10 = ra
T3 = racadabra

Step 3: Find the prefixes shared by the substrings

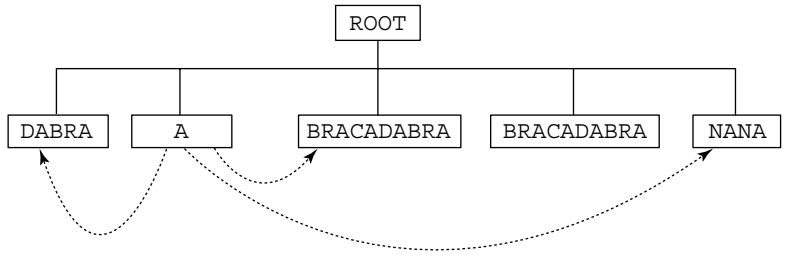


Figure 15.9 Example of a suffix trie

15.7 Applications of Tries

Tries find numerous applications. Some of them are explained as follows:

1. Tries in the Web search engines: Tries can be used to store and maintain the index words and the relevant Web site address. In the Web search process the users query the index words, the search engine has to fetch the appropriate links as results. In this process the search engines maintain the table of index words to the Web link or Web site address. To maintain a sequence of words as the index, the search engines follow the tries procedure for quick retrieval of appropriate information.

2. Symbol table maintenance: A symbol table is a data structure used in the language translators such as compilers or interpreters. A symbol table consists of the address location, syntax and semantic information of the identifier, type and scope of the identifier. To maintain all the identifiers' information, the compilers or interpreters use tries procedure in the form of hash tables.

3. Domain Name Servers: The Domain Name Servers (DNS) provide a facility to map the IP address to the domain name. The users of the DNS query the DNS with IP address or domain name, the DNS servers respond with its corresponding domain name or IP address, respectively. The DNS servers maintain a trie with each bit of the IP address as a node.

4. Indexing a book: The index of a book contains the index words associated with page numbers. To prepare the index word associated with its page, numbers can be achieved by using the tries.

5. Document similarity check: If two or more documents are given, to find the similarity between the documents tries can be used.

6. Tries in the Internet routers: In the Internet, the routers will have many incoming lines and many outgoing lines. If the router receives any packet, the router has to send the packet through the appropriate outgoing link. The router should work with good speed to find the appropriate link for the destination IP address, the router maintains the IP domain with its corresponding link as a trie.

SUMMARY

- Pattern matching algorithms are of two types, one is fixed pattern matching where the pattern is fixed and the second is regular expression pattern matching where the pattern is framed with a regular expression.
- The fixed pattern matching algorithms are Brute Force Algorithm, Boyer–Moore Algorithm, Knuth–Morris–Pratt Algorithm and Tries.
- The Brute Force Algorithm searches for the pattern in the linear fashion. It finds the possible patterns over the given large string and checks for the existence of the pattern.
- The Boyer–Moore Algorithm and the Knuth–Morris–Pratt Algorithm avoid the unnecessary pattern searches.
- The Boyer–Moore Algorithm works with two stages, it prepares the shift table. Using the shift table the search string will be moved forward to avoid the unnecessary checks. But the algorithm does not guarantee the pattern match. This is the drawback of this algorithm.
- The Knuth–Morris–Pratt Algorithm works with the failure function, this algorithm is somewhat better than that of the Boyer–Moore Algorithm.
- Tries are the most common techniques where quick information retrieval is needed.
- Simple tries or standard tries will have more number of nodes than other tries, such as compressed tries and suffix tries.

EXERCISES

Fill in the Blanks

1. _____ is the time complexity of Brute Force Algorithm.
2. _____ algorithm is suitable for searching a word in the text editors.
3. _____ is the worst case time complexity of Knuth–Morris–Pratt Algorithm.
4. _____ are the best suitable techniques for information retrieval on the Web.
5. _____ tries take less space complexity.

Multiple-choice Questions

1. _____ algorithm checks all the possible substrings of the given text for finding the pattern.
   a. KMP b. BM
   c. Brute Force d. Standard tries
2. When m=n/2 the time complexity of Brute Force algorithm is
   a. O(mn) b. $O(m^2)$
   c. $O(n^2)$ d. O(n+m)
3. The Worst case time complexity of BM algorithm is _____ A.
   a. O(mn) b. $O(m^2)$
   c. $O(n^2)$ d. O(n+m)
4. _____ Algorithm is the best algorithm if more number of patterns are to be searched on the same text string.
   a. BM b. KMP
   c. Brute Force d. Both a and b
5. _____ trie can be used for finding the repetitive patterns of a search string.
   a. Standard b. Compressed
   c. Suffix d. Both a and b

Short-answer Questions

1. Write an algorithm for Brute Force Algorithm and discuss the performance analysis.
2. Write an algorithm for Boyer–Moore Algorithm and discuss the performance analysis.
3. Write an algorithm for Knuth–Morris–Pratt Algorithm and discuss the performance analysis.
4. You are given 4 MB of text file; you are supposed to search for 10 bytes of pattern. Which algorithm will you prefer? Give reasons.

Essay Questions

1. Explain an algorithm with an example for Brute–Force pattern matching, and write a C++ program.
2. Write an algorithm for Boyer–Moore pattern matching, and write a C++ program.
3. Write an algorithm for Knuth–Morris–Pratt algorithm and write a C++ program.
4. Explain standard tries with an example.
5. Explain compressed tries with an example.
6. Explain suffix tries with an example.

7. Write short notes on space and time complexities of Brute Force, BM and KMP pattern matching algorithms.
8. Write short notes on space and time complexities of standard, compressed and suffix tries.
9. Explain failure functionality of KMP when P= “abacab” and P=”abacaabaccabacabaabb”.
10. Explain detailed procedure of Boyer–Moore algorithm when P= “abacab” and P=”abacaabaccabacabaabb”.
11. Write and describe algorithms for insert, search and delete procedures, C++ code for standard tries.
12. Write and describe algorithms for insert, search and delete procedures, and C++ code for compressed tries.

Chapter 16

Sorting and Searching

This chapter provides an exclusive discussion regarding sorting and searching. In this chapter, a number of sorting techniques namely bubble sort, insertion sort, selection sort, quick sort, merge sort, shell sort, radix sort and heap sort are detailed. The chapter also gives a clear explanation about the very popular and frequently used searching techniques namely linear search, binary search and Fibonacci search.

16.1 SORTING

Sorting is a technique to arrange a set of items in some order. One reason that it is so useful is that it is much easier to search for something in a sorted list than an unsorted one. Among the sorting algorithms, bubble sort, insertion sort and selection sort are good enough for most of the small tasks. To process a large amount of data, choose one among the sorting algorithms such as quick sort, merge sort, shell sort, radix sort, heap sort, etc.

16.1.1 Bubble Sort

The simplest sorting algorithm is bubble sort. The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and swapping their positions if necessary. This process is repeated as many times as necessary until the array is sorted. Let L be an unordered list with elements A1,A2,A3,…,An. Bubble sort ordered these elements in their increasing order, that is L={A1,A2,A3,…, An}, where A1≤A2≤A3≤…An.

The following are the steps involved in Bubble sort.

Step 1: A1 and A2 are compared and if they are out of order, then interchange so that A1≤A2. Then compare A2 and A3 and interchange so that A2 ≤ A3. This process is repeated for all n elements. When this step is completed observe that the largest element is "bubbled up" to the nth position, i.e. An contains the largest element. Here n-1 comparisons take place.

Step 2: Repeating Step 1 with one fewer comparison, i.e. up to n–2 elements. After this step the second largest element is stored in the n–1th location.

.....

.....

Step n–1: Finally at the n–1th comparison, compare A1 and A2 and interchange so that A1<A2. After the n–1 comparisons the list will be in the sorted order.
Algorithm 16.1 is the algorithm for bubble sort.

Algorithm 16.1: BUBBLE SORT (DATA,N)

```
//Here DATA is an array with N elements
1. Repeat 2 and 3 steps for K=1 to N-1
2. Set PTR=1 // initializing pass pointer PTR
3. Loop until PTR≤N-K
1. Check whether DATA[PTR]>DATA[PTR+1] then
2. Swap(DATA[PTR],DATA[PTR+1])
3. Set PTR=PTR+1
4. End loop
5. End
```

Example 16.1

Let A be an array with the elements 23 15 72 58 66 32 31 70 to be sorted. Figure 16.1 shows various steps in bubble sort.

Pass 1: Compare A1 andA2:

23 > 15, so interchange as follows:

(15) (23) 72 58 66 32 31 70

Compare A2 and A3:

23 < 72, the elements are in order.

15 23 72 58 66 32 31 70

Compare A3 and A4:

72 > 58, so interchange as follows:

15 23 (58) (72) 66 32 31 70

Compare A4 and A5:

72 > 66, so interchange as follows:

15 23 58 (66) (72) 32 31 70

Compare A5 and A6::

72 >32, so interchange as follows:

15 23 58 66 (32) (72) 31 70

Compare A6 and A7:

72 > 31, so interchange as follows:

15 23 58 66 32 (31) (72) 70

Compare A7 and A8:

72 > 70, so interchange as follows:

15 23 58 66 32 31 (70) (72)

At the first pass, the largest elements in the list is placed in the nth position and it takes (n–1) comparisons.

Pass 2: (15) (23) 58 66 32 31 70 72 [No interchange]

15 (23) (58) 66 32 31 70 72 [No interchange]

15 23 (58) (66) 32 31 70 72 [No interchange]

15 23 58 (66) (32) 31 70 72 [Interchange]

32 66

15 23 58 32 (66) (31) 70 72 [Interchange]

15 23 58 32 31 (66) (70) 72 [No interchange]

15 23 58 32 31 66 70 72

At this pass the second largest element is placed in the n–1th location and takes n–2 comparisons.

Pass 3: (15) (23) 58 32 31 66 70 72 [No interchange]

15 (23) (58) 32 31 66 70 72 [No interchange]

15 23 (32) (58) 31 66 70 72 [Interchange]

15 23 32 (31) (58) 66 70 72 [Interchange]

15 23 32 31 (58) (66) 70 72 [No interchange]

At this pass the third largest element is placed in the n–2nd location and takes n–3 comparisons.

Pass 4: (15) (23) 32 31 58 66 70 72 [No interchange]

15 (23) (32) 31 58 66 70 72 [No interchange]

15 23 (31) (32) 58 66 70 72 [Interchange]

15 23 31 32 58 66 70 72

At this pass the fourth largest element is placed in the n–3rd location and takes n–4 comparisons.

Pass 5: (15) (23) 31 32 58 66 70 72 [No interchange]

15 (23) (31) 32 58 66 70 72 [No interchange]

15 23 (31) (32) 58 66 70 72 [No interchange]

At this pass the fifth largest element is placed in the n–4th location and takes n–5 comparisons.

Pass 6: 15 23 31 32 58 66 70 72 [No interchange]

15 23 31 32 58 66 70 72

At this pass the sixth largest element is placed in the n–5th location and takes n–6 comparisons.

Pass 7: A1 is compared with A2, since 15 < 23 no interchange takes place.

Finally the sorted array is {15 23 31 32 58 66 70 72}

Figure 16.1 Trace of bubble sort

Complexity: The number of comparisons in bubble sort can be easily calculated. In the first pass, n–1 comparisons are there and in the second pass n–2 comparisons and so on. The number of comparisons can be calculated as follows:

$$f(n) = f(n-1) + f(n-2) + \cdots + 2 + 1$$

$$= \frac{n(n-1)}{2} = O(n^2)$$

The time complexity of the bubble sort algorithm is $O(n^2)$.

Program 16.1

```
#include<iostream.h>
#include<conio.h>
void bubblesort(int a[],int n)
{
 int i,j;
 for(i=n-1;i>0;i--)
 for(j=0;j<i;j++)
 if(a[j]>a[j+1])
 a[j]=(a[j]+a[j+1])-(a[j+1]=a[j]);//Single Line Swapping
}
void main()
{
 int i,a[30],n;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=0;i<n;i++)
 cin>>a[i];
 cout<<"Before Sorting List Is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 bubblesort(a,n);
 cout<<"\nAfter Sorting List Is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 getch();
}
```

Output

```
Enter Size Of Array:8
Enter 8 Elements:14
21
10
30
45
87
55
46
```

```
Before Sorting List Is:
14 21 10 30 45 87 55 46
After Sorting List Is:
10 14 21 30 45 46 55 87
```

16.1.2 Insertion Sort

This sorting algorithm sorts the array by inserting each element in their appropriate positions. Let L be the unordered list with elements A1,A2,…,An. The insertion sort algorithm scans A1 to An, inserting each element Ak into its proper position in the sorted sublist of predecessors {A1,A2,…,Ak-1} where k is the key element position that is to be inserted. The following are the steps involved in insertion sort:

Step 1: A2 is compared with its sorted sublist of its predecessors, i.e. {A1} and A2 is inserted either after or before A1 so that {A1,A2} will be in the sorted order.

Step 2: A3 is compared with its sorted sublist of its predecessors, i.e. {A1,A2}, and A3 is inserted at its proper position so that {A1,A2,A3} is in sorted order.

……
……
……

Step n: An is compared with its sorted sublist of predecessor, i.e. {A1,A2,…,An-1}, and An is inserted at its proper position so that {A1,A2,…,An} is in sorted order.

This algorithm takes n comparisons and is useful only when n is small. Algorithm 16.2 is the algorithm for insertion sort.

Algorithm 16.2: INSERTION SORT (DATA,N)

```
 //DATA[1:N] is an array of N elements to be sorted.
1.  Loop for k=2 to n
    1. Set temp=DATA[k]
    2. Set PTR=k
    3. Repeat while (PTR>1) and (DATA[PTR-1])>k  then
        i.   Set DATA[PTR]=DATA[PTR-1]
        ii.  Set PTR=PTR-1
        iii. Set DATA[PTR]=temp
     4. End loop
2. End loop
```

Example 16.2

Let DATA be an array with elements {88,44,55,22,99,33,77,66}. Figure 16.2 shows the procedure of insertion sort algorithm.

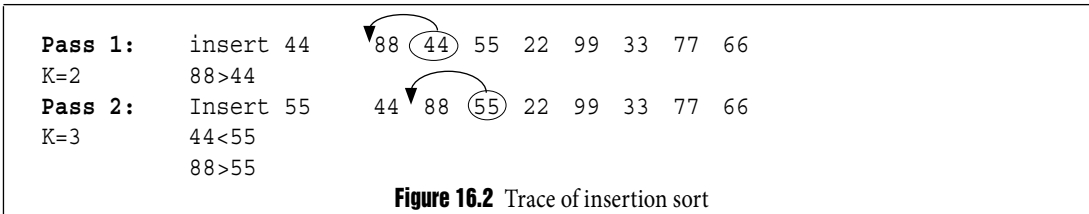


Figure 16.2 Trace of insertion sort

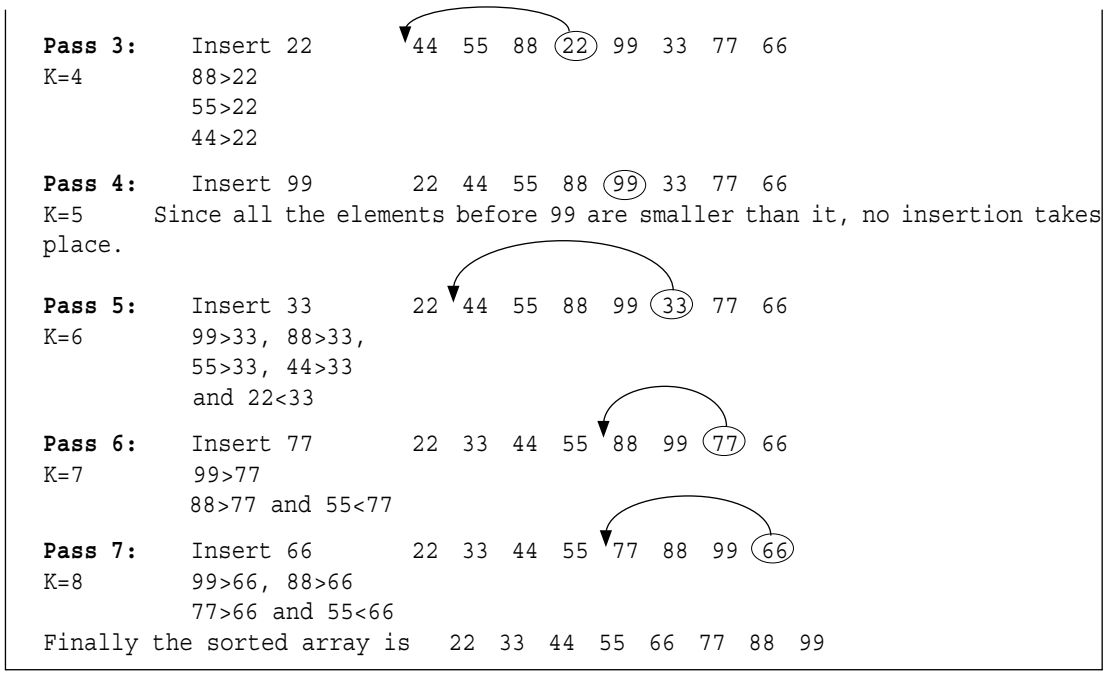


Figure 16.2 *Continued*

In Figure 16.2 each element is compared with its previous element and inserted in its appropriate position. Sorting starts with the second element because there is no element previous to the first element to compare it. Arrows show the position of the element to be inserted.

Complexity: The time complexity of insertion sort is $O(n^2)$.

Program 16.2

```
/*To Sort the Given List Using INSERTION SORT*/
#include<iostream.h
#include<conio.h>
void main()
{
 int i,a[30],n,j,temp,item;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=0;i<n;i++)
 cin>>a[i];
 cout<<"Before Sorting List Is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 for(i=0;i<n;i++)
```

```
 {
  item=a[i];
  j=i-1;
  while(j>=0&&item<a[j])
  {
   a[j+1]=a[j];
   j--;
  }
  a[j+1]=item;
 }
 cout<<"\nAfter Sorting List Is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 getch();
}
```

Output

```
Enter Size Of Array:9
Enter 9 Elements:4
6
0
1
5
7
12
10
2
Before Sorting List Is:
4 6 0 1 5 7 12 10 2
After Sorting List Is:
0 1 2 4 5 6 7 10 12
```

16.1.3 Selection Sort

Selection sort is the most conceptually simple of all the sorting algorithms. It works by selecting the smallest (or largest, if it is to sort in descending order) element of the array and placing it at the head of the array. Then the process is repeated for the remainder of the array; the next largest element is selected and put into the next slot, and so on down the array. Selection sort technique selects an element and places it into appropriate position. Because a selection sort looks at gradually smaller parts of the array each time, a selection sort is slightly faster than the bubble sort. The following are the steps involved in the sorting process. Let L be an unordered list of elements A1, A2, …, An.

Step 1: In the first pass find the position POS of the smallest element in the list of n elements. Then swap POS with the first element of the list so that the first smallest element acquires its right position, i.e. first position in the list.

Step 2: In the second pass find the position POS of the second smallest element in the list of n–1 elements. Swap POS with second element of the list.

.....

…….

Step n: In the (n–1)th pass excluding the first two elements in the list repeat the sorting process by finding the next smallest element and swap it with an appropriate element until the entire list is sorted. The entire sorting takes (n–1) passes.

To find the smallest element among all the elements in the list, first the variable SMALL is set to the first element of the list and the variable POS is set to the position of the first element and then traversing the list comparing SMALL with every other element of DATA[i] as follows:

1. If SMALL < DATA[i], then move to the next element
2. If SMALL>DATA[i], then update SMALL and POS as SMALL=DATA[i] and POS=i

After comparing all the elements SMALL contains the smallest among all the elements and POS contains its location.

Selection sort can also sort the elements in descending order by selecting the largest element instead of the smallest element and swapping it with the element at the last position in the list. Algorithm 16.3 is the algorithm for selection sort. The algorithm FINDMIN finds the smallest element in the list and returns it to the algorithm SELECTION-SORT, which places the smallest element in its appropriate position.

Algorithm 16.3: SELECTION-SORT (DATA, N)

```
1. Repeat for k=1 to N-1
2. Call FINDMIN(DATA, K, N, POS)
3. Set temp=DATA[K]
4. DATA[K]=DATA[POS]
5. DATA[POS]=temp
6. End loop
7. End
```

Algorithm 16.3.1: FINDMIN(DATA, K, N, POS)

```
/*This algorithm is used to find the smallest element among DATA[K], DATA[K+1],…,
DATA[N]*/
1. Set SMALL=DATA[K] and POS=K
2. Loop for i=K+1 to N
(i)   Check whether SMALL is greater than DATA[i] then
(ii)  Set SMALL=DATA[i] and POS=i
3. End loop
4. End
```

Example 16.3

Let DATA be an array with elements 88 44 55 22 99 33 77 66.

Figure16.3 shows the selection sort procedure on array DATA.

Pass	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
K=1, POS=4	(88)	44	55	(22)	99	33	77	66
K=2, POS=6	22	(44)	55	88	99	(33)	77	66
K=3, POS=6	22	33	(55)	88	99	(44)	77	66
K=4, POS=6	22	33	44	(88)	99	(55)	77	66
K=5, POS=8	22	33	44	55	(99)	88	77	(66)
K=6, POS=7	22	33	44	55	66	(88)	(77)	99
K=7, POS=7	22	33	44	55	66	77	(88)	99
Sorted array is	22	33	44	55	66	77	88	99

Figure 16.3 Trace of selection sort

Observe that in Figure 16.3 POS gives the smallest among DATA[K], DATA[K+1], ..., DATA[N] during pass K. The circled elements indicate the elements which are to be swapped. In a similar way all the elements will be sorted.

Complexity: The time complexity of selection sort is $O(n^2)$.

In pass 1 it takes n–1 comparison, in pass 2 it takes n–2 comparison and in pass n it takes i comparison. So, the recurrence relation is given as

$$F(n) = (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2 = O(n^2)$$

16.1.4 Quick Sort

Quick sort belongs to the family of sorting by exchange where the elements which are out of order are exchanged themselves in order to obtain a sorted list.

Quick sort performs sorting based on the principle of partitioning the list into two sublists taking any one of the elements in the list as the key element which is called the pivot element. Two sublists occur left and right of the pivot element, i.e. the pivot element will be placed in its appropriate position. Again the sublists are further partitioned against their respective pivot elements until there is no possibility to partition. At this stage all the elements in the list will be in sorted order. The following is the procedure for quick sort:

Let L be an unordered list with elements {A1, A2, …, An}

Step 1: Suppose A1 is chosen as the pivot element; now compare A1 with all the elements in the list moving A1 left to right until the first occurrence of a greatest number Ai that is greater than or equal to A1, i.e. Ai>=A1.

Step 2: Again A1 is compared with the elements moving right to left until the first occurrence of smallest number Aj, Aj<=A1 is found.

Step 3: If i<j then Ai and Aj are exchanged.

If i>j, then A1 is exchanged with Aj, i.e. A1 will be smaller than Aj and greater than Ai. At this stage partition takes place. The list will be divided into two sublists based on the pivot element. Algorithm 16.4 is the algorithm for quick sort.

Algorithm 16.4: QUICK SORT (DATA, LOW, HIGH)

```
1. Check whether LOW is less than HIGH then
   PQUICK(DATA, LOW, HIGH, POS)
   QUICKSORT(DATA, LOW, POS-1)
   QUICKSORT(DATA, POS+1, HIGH)
2. End
```

Algorithm16.4.1: PQUICK (DATA, LOW, HIGH, POS)

```
/*This algorithm is used to partition the list into sublists. POS is the posi-
tion of the pivot element at the end of the partition. LEFT and RIGHT variables
hold the first and last positions of the array DATA. The variable PIVOT holds
the pivot element.*/
1. Set  LEFT=LOW, RIGHT=HIGH and PIVOT=DATA[LOW]
2. Loop while LEFT<RIGHT
   1. Repeat until DATA[LEFT]>=PIVOT
      Set LEFT=LEFT+1  //pivot elements moves form left to right
   2. Repeat until DATA[RIGHT]<=PIVOT
      Set RIGHT=RIGHT+1 //pivot elements moves form right to left
   3. Check whether LEFT is less than RIGHT then
   4. Swap(DATA[LEFT], DATA[RIGHT])
3. End loop
4. Set POS=RIGHT
5. Swap(DATA[LEFT],DATA[RIGHT])
6. End
```

Example 16.4

Let DATA be an array of elements 6 2 25 10 75 33 10

Figure 16.4 shows various stages of quick sort

Phase 1:

- Choose the first element as the pivot element:

  [6] 2 25 10 75 33 10

- Scan from left to right and compare the pivot element with the rest of the elements until the first occurrence of the greatest number:

  6 2 (25) 10 75 33 10
  → LEFT

- Scan from right to left until the first occurrence of the lowest number:

  6 (2) (25) 10 75 33 10
  ← RIGHT → LEFT

  Since LEFT>RIGHT swap pivot element with RIGHT:

  2 [6] 25 10 75 33 10
  Sublist 1 Sublist 2

- Now the pivot element is placed in its position and the list is partitioned into two sublists. Sort the two sublists with the same procedure as above.

Figure 16.4 Trace of quick sort

Phase 2:

- There is no need to sort sublist 1 since only one element exists.
- Perform quick sort on sublist 2 {25,10,75,33,10}.
- Choose the pivot element as [25]
- Scan from left to right [25] 10 (75) 33 10

  25<75

  Scan from right to left [25] 10 (75) 33 (10)

  LEFT RIGHT

Since LEFT<RIGHT, swap LEFT and RIGHT:

10 [25] 10 33 75

- Scan from left to right 10 [25] 10 (33) 75

  Scan from right to left 10 [25] (10) (33) 75

  RIGHT LEFT

Since LEFT > RIGHT, exchange pivot element and RIGHT:

10 10 [25] 33 75

- At this stage the list after partition is:

[2] [6] 10 10 [25] 33 75

Sublist 1 Sublist 2

Phase 3:

- Quick sort the sublist {10,10}:

  pivot [10] [10]

  LEFT=RIGHT

  Exchange the pivot element with RIGHT.

  At this stage the list after partition [2] [6] 10 [10] [25] 33 75

Phase 4:

- There is no need to sort sublist {10} since only one element exists.
- Quick sort of sublist {33,75}
- Choose the pivot element as [33]
- Scan from left to right [33] 75

  Scan from right to left (33) (75)

  RIGHT LEFT

  No need to exchange.

Figure 16.4 *Continued*

At this stage sorted array is [2] [6] [10] [10] [25] [33] [75]

Finally the sorted array is 2 6 10 10 25 33 75

Figure 16.4 *Continued*

Program 16.4

```
/*To Sort The Given List Using QUICK SORT*/
#include<iostream.h>
#include<conio.h>
void quicksort(int a[],int lb,int ub)
{
 int key=lb,i=lb,j=ub,temp;
 while(i<j)
 {
  while(a[i]<=a[key])
   i++;
  while(a[j]>a[key])
   j--;
  if(i<j)
  a[i]=(a[i]+a[j])-(a[j]=a[i]); //Single Line Swapping
  a[key]=(a[key]+a[j])-(a[j]=a[key]);
  quicksort(a,lb,j-1);
  quicksort(a,j+1,ub);
 }
}
void main()
{
 int i,a[30],n;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=0;i<n;i++)
 cin>>a[i];
 cout<<"Before Sorting List Is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 quicksort(a,0,n-1);
 cout<<"\nAfter Sorting List Is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 getch();
}
```

Output

```
Enter Size Of Array:8
Enter 8 Elements:7
```

```
0
4
9
54
26
35
11
Before Sorting List Is:
7 0 4 9 54 26 35 11
After Sorting List Is:
0 4 7 9 11 26 35 54
```

16.1.5 Merge Sort

Before describing about merge sort, let us discuss about merging. Merging is the process of combining two ordered lists of elements and merged into a single ordered list. Merge sort can be formatted in two ways. The first approach is recursive merge sort which is more complex.

Let the two ordered lists to be merged be list [l: m] and list [m+1; n] and the resultant list is `mergelist` [1: n]. Algorithm 16.5 is the algorithm for merging two ordered lists.

Algorithm 16.5: MERGE (list, merge list, l, m, x)

```
1. Set i=l, j=l, k= m+1                          //i,j,k are list partitions
2. Loop for i ≤m, k≤n:j++
   1. Check whether  list[i]≤list[k] then
   2.  Set mergelist[j]=list[i]
   3.  Set i=i+1
   4.  Else
   5.  Set mergelist[j]=list[k]
   6.  Set k=k+1
   7.    Copy(list+i,list+k,mergelist+j)         /*copy the remaining elements,
                                                    if any,of first list*/
   8.    Copy(list+k,list+n+1,mergelist+j)       /*copy the reaming elements,
                                                    if any of second list*/
3. End loop
4. End
```

j will be incremented by 1 at each iteration of the loop. The total increment in j is at most n-l+1. Hence the total number of iterations in the loop is at n–l+1 times. The statement copy copies at most n-l+1 elements. So, the total time complexity is given by O (n–l+1).

Non-recursive merge sort: The non-recursive version of merge sort begins by treating the given input list of n elements as n independent ordered sublist of size 1. The first pass merges all these sublists in order to obtain n/2 sublists each with two sublists again. If n is odd then one sublist will be of size 1. In the second pass, all these n/2 sublists are then merged to obtain n/4 sublists. Each pass reduces the number of sublists to half. Continue the process of merge until only one sublist exists.

Example 16.5

The given list is {32, 4, 55, 1, 16, 11, 95, 51, 84, 19} which is to be sorted. The tree of Figure 16.5 illustrates the merging of sublists at each pass.

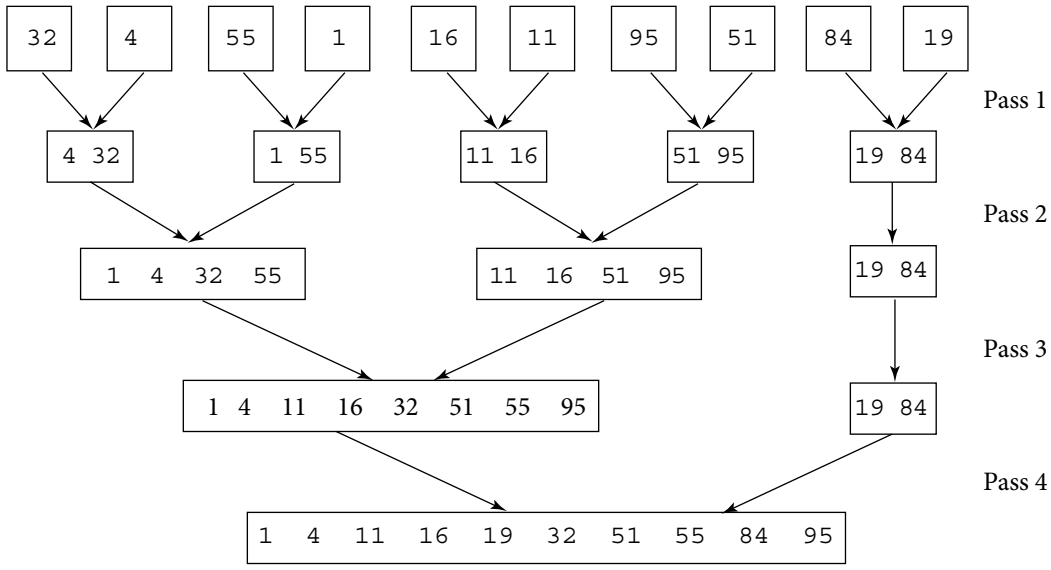


Figure 16.5 Merge tree

Algorithm 16.5.2 contains several passes over the elements to be sorted. Now sorting can be done by repeatedly invoking the algorithm called MERGEPASS. In the first pass, lists with size 1 are merged and in the second pass lists with size 2 are merged and so on. On the i^{th} pass, the size of the lists to be merged will be 2^{i-1}. Therefore a total of $\lfloor \log_2 n \rfloor$ passes are required to merge the sublists. Since two lists are merged with linear time and the complexity of merge sort is O(n), the total time complexity is O(n log n)

Algorithm 16.5.1: MERGEPASS (list, FinalList, n,s)

```
/*s is the size of  sublists. n is the number of elements in the list. Sublists
of size s are merged from list to  FinalList.*/
1. Set i=1    //i is the first position in the first sub list being merged
2. loop for i<=n-2*s+1 do
   1. Set i=i+2*s
   2. MERGE(list,FinalList,i,i+s-1,i+2*s-1);
   3. MERGE (list,FinalList,i,i+s-1,n)      // Merging the remaining list
   4. Check whether(i+s-1<n)then
   5. Else
   6. Copy (list+i,list+n+1,FinalList+i)
3. End
```

Algorithm 16.5.2: MERGESORT(A, n)

```
/* sorting Array A[1:n] into ascending order. Let temp be an array which holds
the sorted elements. l is the length of the sublist that is to the merged.*/
1. Set l=1
2. loop for (l<n; l*=2)
   1. MergePass(A,temp,n,l)
   2. Set l=l*2
```

```
   3. MergePass (temp,A,n,l);    //interchanging the arrays A and temp
   4. End loop
3. End
```

Recursive merge sort: The recursive version of merge sort uses the principle of divide and conquer. Here the given list to be sorted is divided into two sublists of approximately equal in size. These sublists are sorted recursively and all the sorted sublists are merged into a single list which is the required sorted list. Algorithm 16.5.3 is the algorithm for recursive merge sort.

Algorithm 16.5.3: MERGE SORT (A, BEG, END)

```
// A [BEG: END] is an array to the sorted
1. Check whether BEG<END then    //if more than one element lists
   1. Set MID=[(BEG+END)/2]      //find where to divide the list
   2. Mergesort(BEG, MID)
   3. Mergesort(MID+1,END)       //sort the sublist
   4. RMerge(BEG,MID,END)         //combine the sublists
2. end
```

Algorithm 16.5.4: RMERGE (BEG, MID, END)

```
// A[beg: end]is an array which contain two sorted sub arrays in A[beg:mid]
// and [mid+ 1:end]
// This algorithm merges these two sub arrays into a single list.
// temp[] is an temporary array.
1. Set b=BEG, i=BEG, j=MID+1;
2. Loop for b≤MID and j≤END do
   1. Check whether A[b]<=A[i] then
      1. Set temp[i]=A[b];
      2. Set b=b+1
   2. else
      1. Set temp[j]=A[j]
      2. Set j=j+1
      3. Set i=i+1
   3. End loop
   4. Check whether b is greater than mid then
      1. Loop for k=j to END do
          i. Set temp[i]=A[k]
         ii. Set i=i+1
      2. End loop
   5. Else
      1. Loop for k=b to mid do
          i.  Set temp[i]=A[k]
         ii.  Set i=i+1
      2. End loop
   6. Loop for k=beg to end do              //copy array temp to array A
      Set A[k]=temp[k]
   7. End
```

Example 16.6

Let A be an array of elements {21,65,1,43,98,87,34,10} to be sorted. Using merge sort algorithm, the array A is divided into two sublists to be sorted before they are merged to get sorted array A.

Figure 16.6 shows a tree of recursive calls illustrating the working of merge sort on array A. Each node in the tree shows a call to merge sort and shows the output sublist at the end of the execution of the call to mergesort.

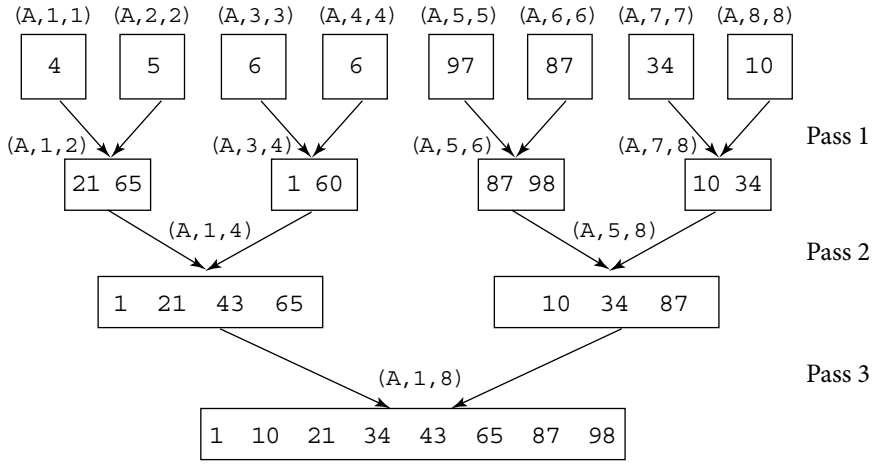


Figure 16.6 Recursive merge sort

Program 16.5

```
/*To Sort The Given List Using MERGE SORT*/
#include<iostream.h>
#include<conio.h>
void mergesort(int a[],int lb,int mid,int ub)
{
 int i=lb,j=mid+1,k=0,b[50];
 while(i<=mid&&j<=ub)
 {
  if(a[i]<a[j])
   b[k++]=a[i++];
  else
   b[k++]=a[j++];
 }
 while(i<=mid)
  b[k++]=a[i++];
 while(j<=ub)
  b[k++]=a[j++];
 for(k=0;k<=ub-lb;k++)
 a[k+lb]=b[k];
}
```

```
void merge(int a[],int low,int high)
{
 int mid;
 if(low<high)
 {
  mid=(low+high)/2;
  merge(a,low,mid);
  merge(a,mid+1,high);
  mergesort(a,low,mid,high);
 }
}
void main()
{
 int i,a[30],n;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=0;i<n;i++)
 cin>>a[i];
 cout<<"Before Sorting List Is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 merge(a,0,n-1);
 cout<<"\nAfter Sorting List Is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 getch();
}
```

Output

```
Enter Size Of Array:10
Enter 10 Elements:9
0
4
6
7
20
15
25
5
2
Before Sorting List Is:
9 0 4 6 7 20 15 25 5 2
After Sorting List Is:
0 2 4 5 6 7 9 15 20 25
```

16.1.6 Shell Sort

Shell sort is a sorting algorithm that is a generalization of insertion sort with two observations:

- Insertion sort is efficient if the input is "almost sorted".
- Insertion sort is typically inefficient because it moves values just one position at a time.

The shell sort is named after its inventor D.L. Shell in 1959. It is fast, easy to understand and easy to implement. However, its complexity analysis is more complicated. The idea of shell sort is to arrange the data sequence in a two-dimensional array and sort the columns of the array. The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column. In each step, the sortedness of the sequence is increased until the last step is completely sorted. However, the number of sorting operations necessary in each step is limited due to the presortedness of the sequence obtained in the preceding steps.

Consider a small value that is initially stored in the incorrect position of the array which makes the array out of order. Using an $O(n^2)$ sort such as bubble sort or insertion sort, it will take roughly n comparisons and exchanges to move this value all the way to the other end of the array. Shell sort first moves values using huge step sizes, so a small value will move a long way towards its final position with just a few comparisons and exchanges

Actually, the data sequence is not arranged in a two-dimensional array, but held in a one-dimensional array that is indexed appropriately. For instance, data elements at positions 0, 5, 10, 15, etc. would form the first column of an array with five columns. The "columns" obtained by indexing in this way are sorted with insertion sort since this method has a good performance with presorted sequences.
Algorithm 16.6 is the algorithm for shell sort.

Algorithm 16.6: SHELL –SORT

```
// An array DATA of length n with array elements numbered 0 to n-1
1. Set  inc=round(n/2)
2. Repeat while inc>0 do
   1. Repeat for i=inc to n-1 do
   2. temp=DATA[i]
   3. j=i
   4. Repeat while j≥inc and DATA[j-inc]>temp do
   5. DATA[j]=DATA[j-inc]
   6. j=j-inc
   7. DATA[j]=temp
   8. End step 4 loop
   9. inc=round(inc/2.2)
  10. End step2 loop
3. End loop
4. End
```

Example 16.7

Let 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2 be the data sequence to be sorted. First, it is arranged in an array with seven columns (left) and then the columns are sorted (right) (Figure 16.7)

3	7	9	0	5	1	6		3	3	2	0	5	1	5
8	4	2	0	6	1	5	→	7	4	4	0	6	1	6
7	3	4	9	8	2			8	7	9	9	8	2	
(a) Array as 7 columns								(b) Sorted columns						

Figure 16.7 Sorted array with seven columns

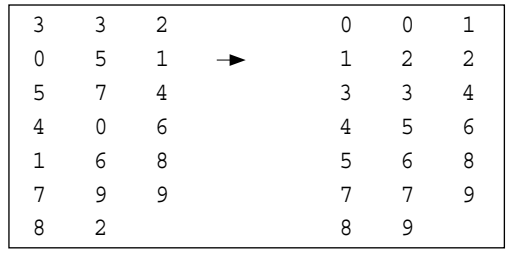


Figure 16.8 Sorting elements by representing them as three columns

Data elements 8 and 9 have now already come to the end of the sequence, but a small element (2) is still there as shown in Figure 16.7(b). In the next step, the sequence is arranged in three columns, which are again sorted.

Now the sequence is almost completely sorted as shown in Figure 16.8. When arranging it in one column in the last step, it is only the elements 6, 8 and 9 that have to move a little bit to their correct positions.

Program 16.6

```
/*To Sort The Given List Using SHELL SORT*/
#include<iostream.h>
#include<conio.h>
#include<math.h>
int i,a[30],num,b[30],n;
void build(int i)
{
 if(b[i]<n)
 {
  b[i+1]=2*b[i];
  build(i+1);
 }
 else
 num=i-1;
}
void shellsort()
{
 int temp,i,j,s;
 for(;num>=1;num--)
 {
  s=b[num];
  for(i=s+1;i<=n;i++)
  {
   temp=a[i];
   j=i-s;
   while((j>=1)&&(a[j]>temp))
   {
    a[j+s]=a[j];
    j-=s;
```

```
    }
    a[j+s]=temp;
   }
  }
 }
void main()
{
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<< Enter"<<n<<"Elements:";
 for(i=1;i<=n;i++)
 cin>>a[i];
 cout<<"Before Sorting List Is:\n";
 for(i=1;i<=n;i++)
 cout<<a[i]<<" ";
 b[1]=1,build(1);
 shellsort();
 cout<<"\nAfter Sorting List Is:\n";
 for(i=1;i<=n;i++)
 cout<<a[i]<<" ";
 getch();
}
```

Output

```
Enter Size Of Array:10
Enter 10 Elements:87
64
2
8
44
66
87
12
14
10
Before Sorting List Is:
87 64 2 8 44 66 87 12 14 10
After Sorting List Is:
2 8 10 12 14 44 64 66 87 87
```

16.1.7 Radix Sort

Radix sort was originally used to sort punched cards in several passes. A computer algorithm was invented for radix sort in 1954 by Harold H. Seward. This sort is performed on a mechanical card sorter. Each card contains 80 columns, and each column may contain a character of alphabet. When sorting is performed on this type of card sorter, only one column is examined at a time.

A metal pointer is used to select only one of the 80 columns. For sorting numerical data, there are 10 packets corresponding to 10 decimal digits, and the sort places all the cards for a given data in an appropriate pocket.

The operator of the sorter combines all the decks of cards in the order from the 10 pockets, i.e. the resulting deck contains pocket 0 on the bottom and pocket 9 on the top. Generally numbers with more than one digit are sorted; in such a case the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit. Each column in the card is sorted by starting from the lowest-order column and moves towards the other column from right to left.

Each pass through the array takes O(n) time. If the maximum magnitude of a number in the array is 'p', and if entries are treated as base 'q' numbers, then 1+floor($\log_q(p)$) passes are needed. If 'p' is a constant, radix sort takes linear time, O(n). However, if all the numbers in the array are different then m is at least O(n), so O(log(n)) passes are needed, O(n.log(n)) is the total time complexity.

Algorithm 16.7: FIND (DATA)

```
//This algorithm finds the single digit places in the array of elements
1. Set max=0
2. Loop for i=1 to n
3. Check whether DATA[i]>max then
   i. Set max=DATA[i]
   ii.Set  max1=0
4. Loop while max!=0 do
   i.  max=max/10
   ii. max1++
5. End loop
6. RADIX SORT(DATA,max1,n)
```

Algorithm 16.8: RADIX SORT (DATA, m,n)

```
//Places the elements in appropriate pockets and recollects them in order.
1. Set y=0
2. Loop for i=1 to m
3. Set q=1
4. Set p= pow(10,y)
5. Set y=y+1
6. Loop for j=0 to 9
7. Loop for k=1 to n
8. Set digit=(DATA[k]/p)%10
9. Check whether digit is equal to j  then
10.Set b[q++]=DATA[k]
11.End step6 loop
12.End step5 loop
13.Loop for temp=1 to n
14.Set DATA[temp]=b[temp]
15.End step13 loop
16.End step2 loop
```

Example 16.8 explains the radix sort.

Example 16.8

Let DATA be an array of elements such as {82, 43, 64, 11, 56, 75, 49, 63, 99, 78, 70, 18, 61}.

First pass:

Pocket:	0	1	2	3	4	5	6	7	8	9
	70	11	82	43	64	75	56	78	49	99
		61		63						

First unit digits in the sequence are sorted. Now the array at this stage will be

70 11 61 82 43 63 64 75 56 78 18 49 99

Second pass: Now comparing the 10-digit values of numbers in the sequence.

Pocket: 0	1	2	3	4	5	6	7	8	9
11			43	56	61	70	82	99	
18			49		63	75			
					64	78			

Combining all the 10 pockets gives the resultant sequence:

11 18 43 49 56 61 63 64 70 75 78 82 99

Program 16.7

```
/*To Sort The Given List Using RADIX SORT*/
#include<iostream.h>
#include<conio.h>
#include<math.h>
void radixsort(int a[],int m,int n)
{
 int temp,i,j,k,y,digit,b[50],p,q;
 y=-1;
 for(i=1;i<=m;i++)
 {
  q=1,y++,p=pow(10,y);
  for(j=0;j<=9;j++)
  {
   for(k=1;k<=n;k++)
   {
    digit=(a[k]/p)%10;
    if(digit==j)
    b[q++]=a[k];
   }
  }
  for(temp=1;temp<=n;temp++)
  a[temp]=b[temp];
 }
```

```
}
void main()
{
 int i,a[30],n,max,max1;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=1;i<=n;i++)
 cin>>a[i];
 cout<<"Before Sorting List Is:\n";
 for(i=1;i<=n;i++)
 cout<<a[i]<<" ";
 max=0;
 for(i=1;i<=n;i++)
 if(a[i]>max)
 max=a[i];
 max1=0;
 while(max!=0)
 {
  max=max/10;
  max1++;
 }
 radixsort(a,max1,n);
 cout<<"\nAfter Sorting List Is:\n";
 for(i=1;i<=n;i++)
 cout<<a[i]<<" ";
 getch();
}
```

Output

```
Enter Size Of Array:10
Enter 10 Elements:87
64
23
54
10
2
9
452
100
234
Before Sorting List Is:
87 64 23 54 10 2 9 452 100 234
After Sorting List Is:
2 9 10 23 54 64 87 100 234 452
```

16.1.8 Heap Sort

The heap sort combines the best of both merge sort and insertion sort. Like merge sort, the worst case time of heap sort is O(n log n) and like insertion sort, heap sort sorts by inserting the element in its appropriate position [in-place]. The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1:n]. Since the maximum element of the array stored at the root is A[1], it can be put into its correct final position by exchanging it with A[n] (the last element in A). Now discard node n from the heap then the remaining elements can be made into a heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

Algorithm 16.8: HEAPSORT (A)

```
1. BUILD-HEAP (A)
2. Loop for i=length (A) down to 2 do
   exchange A[1] and A[i]
   Set heap-size[A] to heap-size[A]-1
   HEAP(A,1)
3. End
```

The HEAPSORT procedure takes time O(n log n) since the call to BUILD-HEAP takes time O(n) and each of the n–1 calls to Heap takes time O(log n).

Building a heap: Using the algorithm HEAP in bottom-up order the array A[1:n] is converted into a heap. Since the elements in the subarray A[n/2 +1: n] are all leaves, the algorithm BUILD-HEAP goes through the remaining nodes of the tree and runs the algorithm HEAP on each one. The bottom-up order of processing node guarantees that the subtree rooted at the children are heap before HEAP is run at their parent.

Algorithm 16.8.1: BUILD-HEAP (A)

```
1. Set heap-size(A)=length[A]
2. Loop for i=floor(length[A]/2) to 1 do
3. HEAP(A,i)
4. End
```

Heap: The algorithm HEAP picks the largest child node and compares it to the parent node. If the parent node is larger than the child node then the heap process terminates, otherwise it swaps the parent node with the largest child node so that the parent now becomes larger than its children. It is important to note that swap may destroy the heap property of the subtree rooted at the largest child node. If this is the case, HEAP calls itself again using the largest child node as the new root.

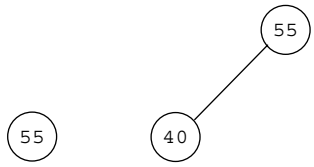
Algorithm 16.8.2: HEAP (A, i)

```
1. Set l=left[i]
2. Set r=right[i]
3. Check whether (l≤heap-size[A] and A[l]>A[i]) then
4. Set largest=l
5. else
6. Set largest=i
7. Check whether (r≤heap-size [A] and A[i]>A[largest]) then
8. Set largest=r
9. Check whether (largest≠i)  then
```

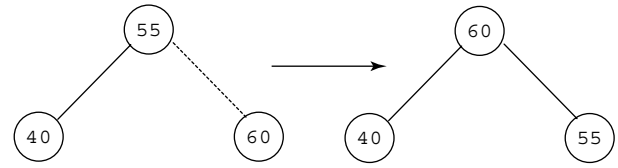
```
10.   Swap(A[i],A[largest])
11.   HEAP(A,largest)
```

Example 16.9

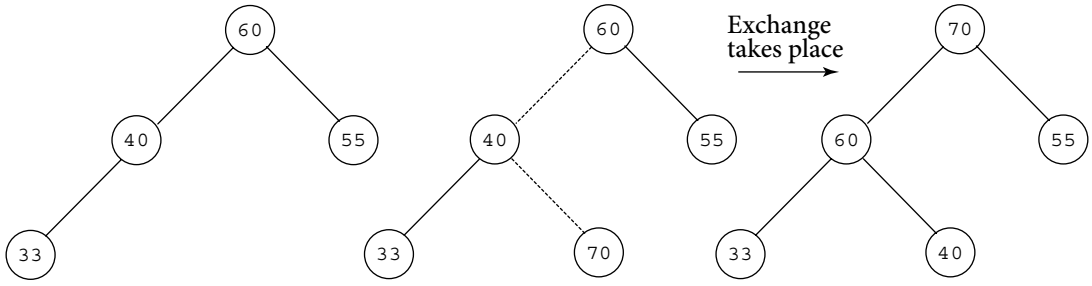
Suppose a heap is built from the following list of elements 55, 40, 60, 33, 70, 66, 88, 66. Figure 16.9 shows the stages of building a heap. Consider that a Max heap is to be created. First node 55 is created and when node 40 is to be added as a child to node 55, check whether the parent node is greater than the child node. If so no exchange takes place otherwise interchange parent and child nodes, here in this case 55 > 40 so no exchange takes place. In Figure 16.9(c) when adding a node 66 as the right child to node 55 exchange takes place, since 66 > 55, i.e. child node is greater than the parent node. The dotted line indicates the exchange of nodes. Continuing in this process, one can create a Maxheap.



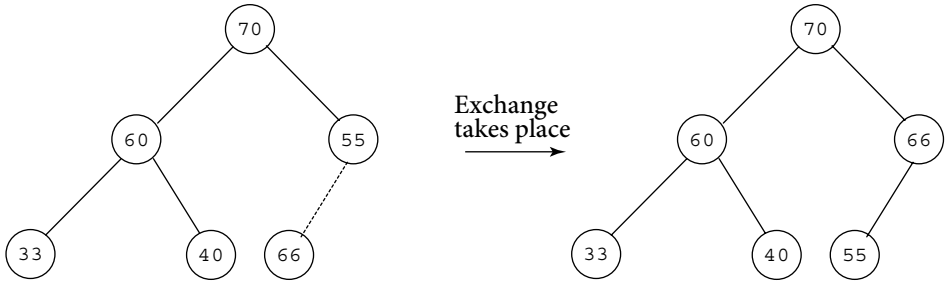
(a) ITEM = 55 (b) ITEM = 40



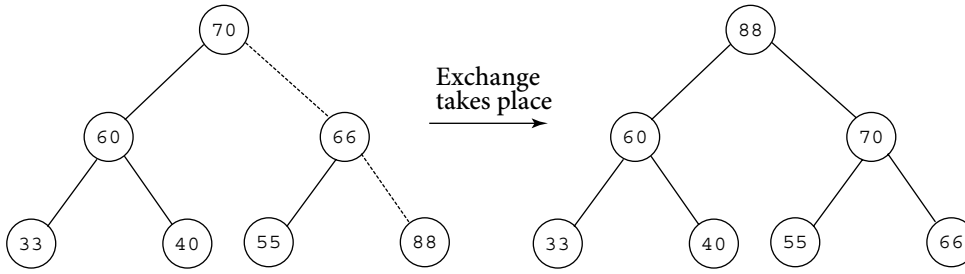
(c) ITEM = 60



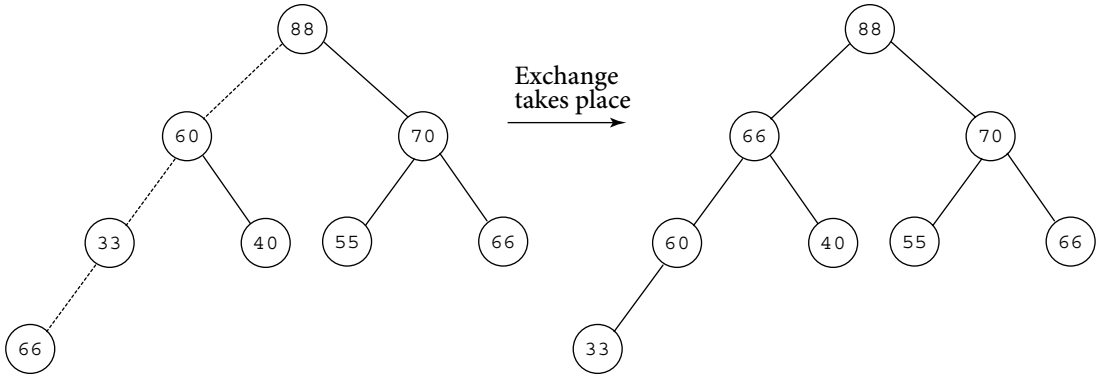
(d) ITEM = 33 (e) ITEM = 70



(f) ITEM = 66



(g) ITEM = 88

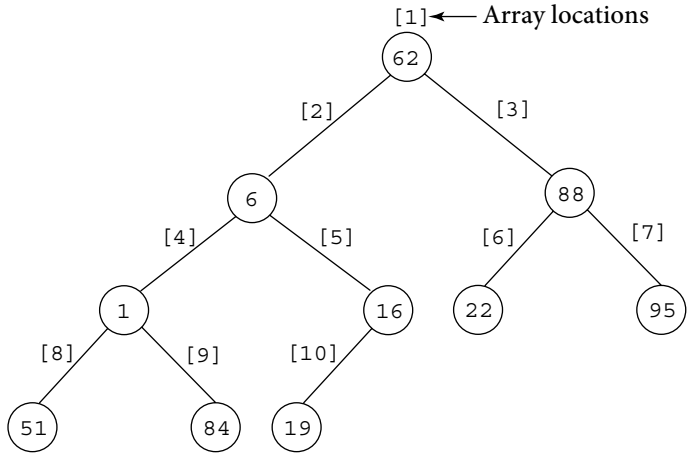


(g) ITEM = 66

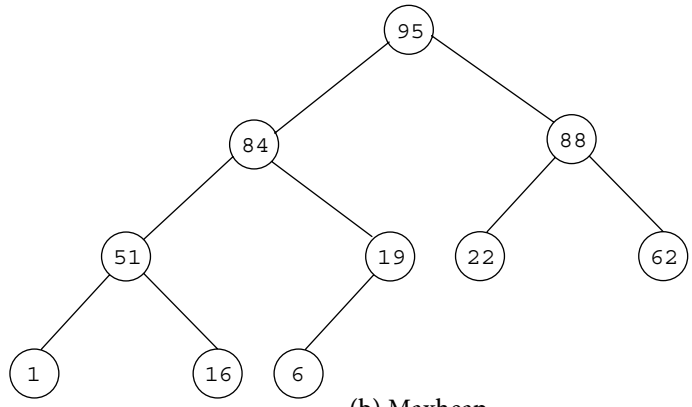
Figure 16.9 Building a heap

Example 16.10

Let L be the given array of {62, 6, 88, 1, 16, 22, 95, 51, 84, 19} on which heap sort is to be performed. Figure 16.10, shows the tree representation of the given list and Figure 16.10(b) shows the initial heap.

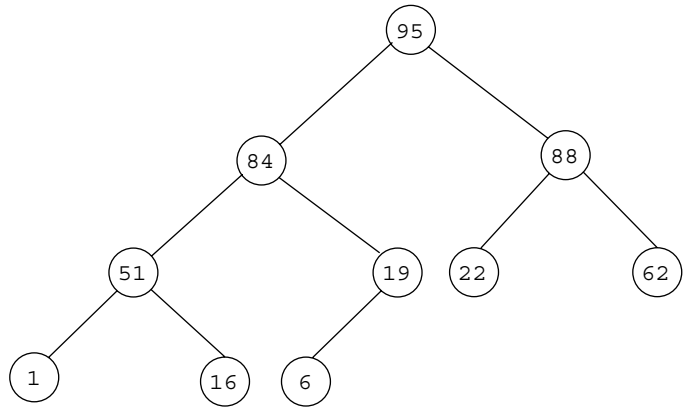


(a) Binary tree representation of array

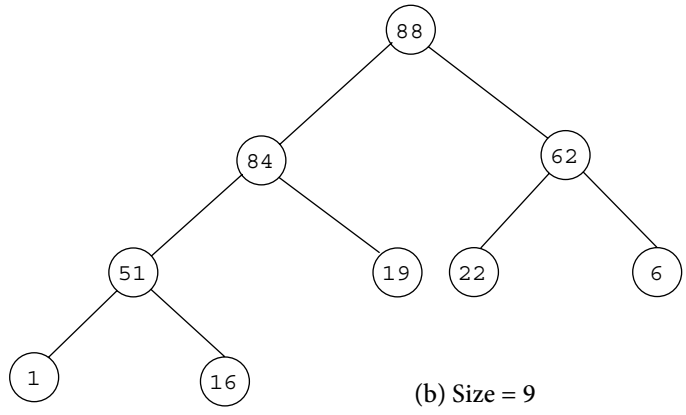


(b) Maxheap

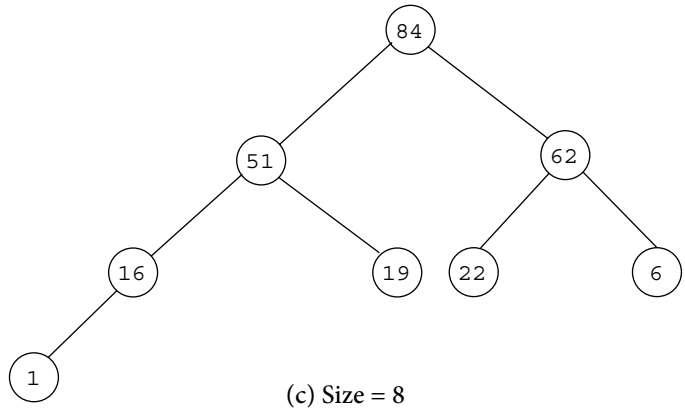
Figure 16.10 Heap representation



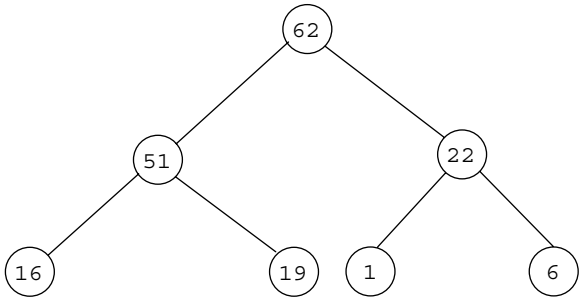
(a) Size = 10



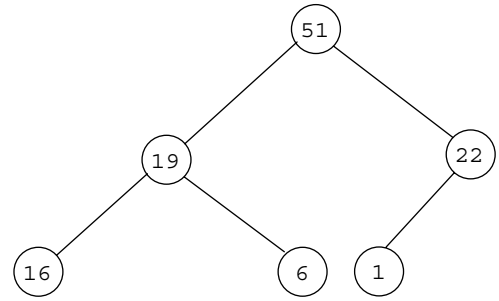
(b) Size = 9
Sorted list {95}



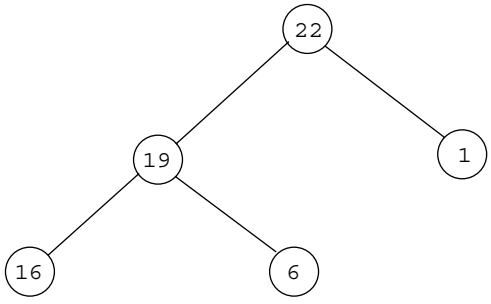
(c) Size = 8
Sorted list {95, 88}



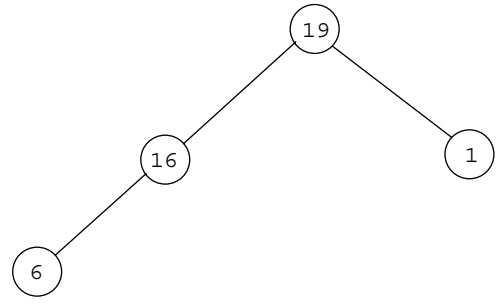
(d) Size = 7
Sorted list {95, 88, 84}



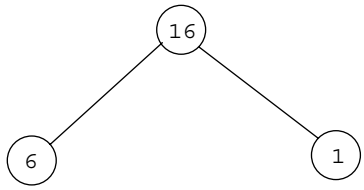
(e) Size = 6
Sorted list {95, 88, 84, 62}



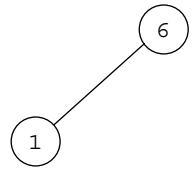
(f) Size = 5
Sorted list {95, 88, 84, 62, 51}



(g) Size = 4
Sorted list {95, 88, 84, 62, 51, 22}



(h) Size = 3
Sorted list {95, 88, 84, 62, 51, 22, 19}



(i) Size = 2
Sorted list {95, 88, 84, 62, 51, 22, 19, 16}

(j) Size = 1
Sorted list {95, 88, 84, 62, 51, 22, 19, 16, 6, 1}

Figure 16.11 Trace of heap sort

Figure 16.11 shows various stages in heap sort. The root element is deleted and again reheap is performed. In Figure 16.11(b), 95 is deleted and reheap is performed to make the next highest element as root, i.e. 88. In Figure 16.11(c) 88 is deleted and reheap is performed and the remaining element to make the next highest element is root, i.e. 84. This is continued in this way until only one element exists in the tree.

Program 16.8

```
#include <iostream.h>
#include<conio.h>
int i,j,k,flag;
void swap(int*x,int*y)
{
 int temp;
 temp=*x;
 *x=*y;
 *y=temp;
}
void adjust(int a[],int i,int n)
{
 k=a[i];
 flag=1;
 j=2*i;
 while(j<=n&&flag)
 {
  if(j<n&&a[j]<a[j+1])
   j++;
  if(k>=a[j])
   flag=0;
  else
```

```
   {
    a[j/2]=a[j];
    j=j*2;
   }
  }
 a[j/2]=k;
}
void buildheap(int a[],int n)
{
 for(i=(n/2);i>=0;i--)
 adjust(a,i,n-1);
}
void heapsort(int a[],int n)
{
 buildheap(a,n);
 for(i=(n-2); i>=0;i--)
 {
  swap(&a[0],&a[i+1]);
  adjust(a,0,i);
 }
}
void main()
{
 int a[50],n;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=0;i<n;i++)
 cin>>a[i];
 cout<<"The List Before Sorting is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 heapsort(a,n);
 cout<<"\nThe List After Sorting is:\n";
 for(i=0;i<n;i++)
 cout<<a[i]<<" ";
 getch();
}
```

Output

```
Enter Size Of Array:10
Enter 10 Elements:46
23
84
74
96
56
```

```
25
88
62
90
Before Sorting List Is:
46 23 84 74 96 56 25 88 62 90
After Sorting List Is:
 23 25 46 56 62 74 84 88 90 96
```

16.2 SEARCHING

Searching is the process of finding the location of a given element in a set of elements. The search is said to be successful if the given element is found, i.e. the element does exist in the collection (such as an array); otherwise it is unsuccessful. In this section various classical algorithms for searching are discussed in detail.

16.2.1 Linear Search or Sequential Search

Linear search is the technique of searching a given item in the list by comparing each and every element in the list sequentially. If the item is found then the search is said to be successful, if not, it is considered to be unsuccessful.

Suppose DATA is an array with N elements, let ITEM be the element to be searched in DATA. Traverse the array DATA sequentially and search for the ITEM by comparing each element of it with ITEM, i.e. first compare DATA[1] with ITEM and then DATA[2] with ITEM and so on until ITEM is found. If item is not, found the search is said to be unsuccessful.

Algorithm 16.9 is the algorithm for Linear Search.

Algorithm 16.9: LINEAR_SEARCH (DATA, N, ITEM, LOC)

```
1. Set LOC=1                        //initialize the counter variable
2. Repeat while(LOC<N) and DATA[LOC]≠TEM
3. Set LOC=LOC+1
4. End loop
5. Check whether DATA[LOC]=ITEM hen
   Print "unsuccessful search"
6. Else
   Print "successful search at location LOC"
7. End
```

Example 16.11

Let DATA be an array of elements {35, 20, 33, 45, 60, 55, 60}. Suppose ITEM=60 is to be searched. Search proceeds down the array comparing each element with ITEM until it finds the last element in the array. Initially assign DATA[N+1]=60.

First it checks with DATA[1], i.e. 35=60 which is false, then it checks with DATA[2], i.e. 65=60 again false, then it checks with DATA[3], i.e. 20=60 is also false. Continuing in this way at LOC=6, the element 60 matches with the ITEM. So, the search is said to be successful. If the ITEM is not found, then the search is said to be unsuccessful.

The time complexity of linear search in best case is O(1), worst case is O(n) and in average case is also O(n).

Program 16.9

```
/*To search an element in the given list using Linear search*/
#include<iostream.h>
#include<conio.h>
void main()
{
 int a[50],search,n,i;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=0;i<n;i++)
 cin>>a[i];
 cout<<"Enter Search Element:";
 cin>>search;
 for(i=0;i<n;i++)
 if(a[i]==search)
 {
  cout<<"Search Element"<<search<<"Is Found At"<<i+1<<"Location";
  getch();
  return;
 }
 cout<<"Search Element"<<search<<"Is Not Found";
 getch();
}
```

Output

```
Enter Size Of Array:6
Enter 6 Elements:4
8
9
1
2
3
Enter Search Element:2
Search Element 2 Is Found At 5 Location
```

16.2.2 Binary Search

Linear search so far discussed is simple and it is efficient if the list is small. Suppose to find a word in a dictionary performing linear search takes more time, whereas binary search instead of searching the dictionary from first to last, it opens the middle of the dictionary and searches that half of the dictionary which contains the word. Then again opening that half in the middle to find which quarter of it contains the word. Continuing like this, one can find the location of the word in the dictionary quickly since the number of possible locations to find it in the dictionary is reduced. This policy of searching is known as binary search and is better than linear search.

Binary search is a technique of searching an item in the ordered list by dividing the list into two halves for each comparison until the item is found or not.

Suppose DATA is an array of ordered elements. Let ITEM be the element to be searched in the DATA. First the ITEM is compared with the middle element of DATA, i.e. DATA[MID] where MID value is obtained by MID=(LB+UB)/2 where LB is the lower bound and UB is the upper bound. If ITEM matches with the DATA[MID] then search is said to be successful and the variable LOC contains the location of the MID. Otherwise a new segment of DATA is designed as follows.

1. If ITEM<DATA[MID], then ITEM may appear in the left sublist of DATA[MID] that is DATA[LB], DATA[LB+1], …, DATA[MID–1]. Update UB to MID–1 value and then start searching again.
2. If ITEM > DATA[MID] then ITEM may appear on the right sublist of DATA[MID] that is DATA[MID+1], DATA[MID+2], …, DATA[UB]. Then update LB value to MID+1 and then start searching again.

Algorithm 16.10 is the algorithm for binary search.

Algorithm 16.10: BINARY_SEARCH (DATA, LB, UB, ITEM, LOC)

```
/*DATA is the ordered list of elements, LB and UB are lower and upper bounds,
ITEM is the element to be searched. This algorithm finds the location LOC of
the ITEM in DATA. Initially assign LB, UB values to the variables B, E respec-
tively, i.e. B=LB and E=UB*/
1. Find MID value
   Set MID=(B+E)/2
2. Repeat step 3 and 4 for checking whether (B ≤ E and DATA[MID]≠ITEM).
3. Check whether (ITEM<DATA[MID]) then
   1. Set E=MID-1
   2. Else
   3. Set B=MID+1
4. update MID as MID=(B+E)/2
5. End step2 loop
6. check whether DATA[MID] is equal to ITEM then
7. Set LOC=MID and  print "search is successful"
8. End
```

Example 16.12

Let DATA be an array of ordered elements as 22, 33, 40, 50, 55, 65, 70, 88, 99. Suppose ITEM=40 is the element to be searched. The search process is as shown below:

1. Initially B=1 and E=9, then
   MID=(B+E)/2=(1+9)/2=5
   So DATA [MID]=55.
   First ITEM is checked with DATA [MID]

   22 33 40 50 (55) 65 70 88 99

   Here the list is divided into two halves and since the ITEM<DATA[MID] ignore the second half of the list without searching.
2. Since 40<55 update the value of E as E=MID–1, i.e. E=4 then MID=(B+E)/4=(1+4)/2=3 so DATA [MID]= 40. Here again the sublist is divided into two halves.

   22 30 (40) 50 55
3. Since 40=40, i.e. ITEM=DATA[MID] the search is said to be successful and the location of ITEM=3.

Program 16.10

```
/*To search an element in the given list using Binary search*/
#include<iostream.h>
#include<conio.h>
void sort(int a[],int n)
{
 int i,j;
 for(i=n-1;i>0;i--)
  for(j=0;j<i;j++)
 if(a[j]>a[j+1])
 a[j]=(a[j]+a[j+1])-(a[j+1]=a[j]);
}
void main()
{
 int i,a[30],n,low,up,search,mid;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=0;i<n;i++)
 cin>>a[i];
 cout<<"Enter Search Element:";
 cin>>search;
 sort(a,n);
 low=0,up=n-1;
 while(low<=up)
 {
  mid=(low+up)/2;
  if(a[mid]==search)
  {
   cout<<"Search Element"<<search<<"Is Found At"<<mid+1<<"Location";
   getch();
   return;
  }
  else if(a[mid]>search)
   up=mid-1;
  else
   low=mid+1;
 }
 cout<<"Search Element"<<search<<"Is Not Found";
 getch();
}
```

Output

```
Enter Size Of Array:7
Enter 7 Elements:8
4
```

```
0
6
9
2
1
Enter Search Element:6
Search Element 6 Is Found At 5 Location
```

16.2.3 Fibonacci Search

The Fibonacci search is a technique of searching a sorted array using the divide and conquer principle that narrows down possible locations with the use of Fibonacci numbers. Compared to binary search, Fibonacci search finds the locations whose addresses have lower distribution. The advantage of Fibonacci search over binary search is that it slightly reduces the average time needed to access a storage location.

Fibonacci search has a complexity of O(log(n)). Instead of splitting the array in the middle, this implementation splits the array corresponding to the Fibonacci numbers, which are defined by the following recurrence relations:

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1}+F_{n-2}$ for n>=2

Fibonacci sequence when used as a search technique is termed as Fibonacci search. As the binary search determines the median in the new list as the next element for comparison, the Fibonacci search also determines the next element for comparison, which is indictated by the sequence of Fibonacci number. Fibonacci search performs search technique only on the ordered list. For the convenience of explanation, consider that the number of lists is one less than the Fibonacci number, i.e. $n=F_{k-1}$.

Consider an ancestor, parent and its child nodes to perform Fibonacci search on a decision tree that satisfies the following properties. If the difference in index between parent and ancestor is F_k, then

i. If the parent is a left child node, then the difference between parent and its child in an index is F_{k-1}.
ii. If the parent is a right child node, then the difference between parent and its child in the index is F_{k-2}.

Suppose consider an order list L = $\{E_1, E_2, E_3, \ldots, E_n\}$, where n is F_{k-1}. The Fibonacci search decision tree is shown in Figure 16.12. For n=20 where $20=F_{s-1}$, the root of the decision tree which is the first element to be compared with the element ITEM is E_i, whose index i is the closest Fibonacci number to n. Here for n=20, 13 is the closest Fibonacci sequence number. So, E_{13} is the root.

If ITEM<E_{13} then the next element to be compared is E_8. If ITEM<E_8, then the next element to be compared is E_5 and so on. By the above properties, the other decision nodes can be easily determined. The right child of E_{13} should be E_{18} and for E_8 the right child should be E_{11} and so on.

Consider the nodes E_8 and E_{11}, since E_{11} is the right child of E_8, the difference between them is 3, i.e. F_4 (because $F_4=F_3+F_2=3$). The difference between E_{11} and its two children is F_2 which is 1. Hence the two child nodes of E_{11} are E_{10} and E_{12}. Similarly consider the nodes E_{18} and E_{16}. E_{16} is the left child of E_{18} and the difference between them is given by F_3, the two child nodes of E_{16} are given by E_{15} and E_{17}, their difference is F_2.

Algorithm 16.11 is the algorithm for Fibonacci search. Here DATA is an ordered list and n is the number of elements such that $F_{K+1}>(n+1)$ and $F_{K+m}=(n+1)$. Item is the one to be searched in the list.

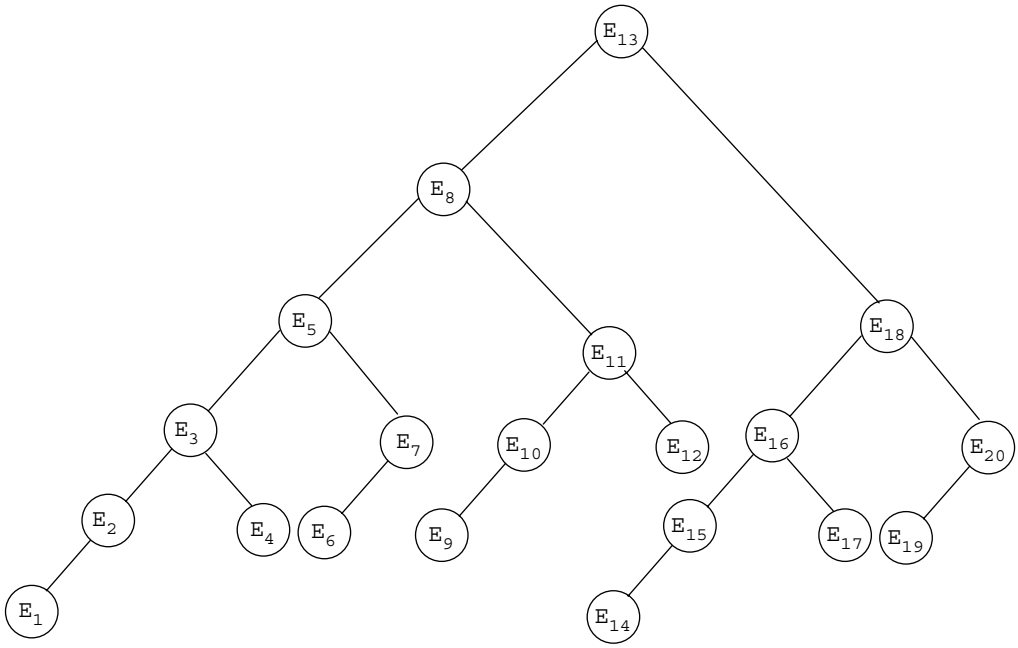


Figure 16.12 Fibonacci search decision tree

Algorithm 16.11: FIBONACCI SEARCH (DATA, n, ITEM)

```
1. Set p=F_{K-1}, q=F_{K-2}, r=F_{K-3} and
   M=(n+1)-(p+q); //Obtain the largest Fibonacci number FK close to n+1.
2. Check whether ITEM is greater than DATA[p]
   then
   Set p=p+m;
   Set flag=false;
3. loop while(p≠0 and flag=true)
   1. Check whether ITEM=DATA[p] then
      Print "successful search"
   2. Check whether ITEM<DATA[p] then
      If(r==0) then set p to 0
         Else
      Set p=p-r, t=q, q=r and r=t-r;
   3. Check whether ITEM>DATA[p] then
      If(q==1) then Set p=0
         Else
      Set p=p+r, q=q-r and r=r-q
   4. End
4. End loop
5. Check whether flag=true then
    Print("search is unsuccessful");
6. End.
```

Example 16.13

The Fibonacci search algorithm can be easily understood by tracing the algorithm. The trace of Fibonacci search is shown in Figure 16.13. Here, the array DATA contains 20 elements such that F_9>(n+1) and F_8+m=(n+1) where m=0 and n=20. Let search for the element K=100 in the list {10,15,20,25,30,35,40,45,50,55,60,65, 70,75,80,85,90,95,100,105}.

The algorithm for Fibonacci search first gets the largest number closer to n+1, i.e. in this case it is F_8. Compare K=100 with the data element with index F_7 that is DATA[13]=70. Since K>DATA[13], the list is reduced to DATA[14:20]={75,80,85,90,95,100,105}. Now compare K with DATA[18]=95, since K>DATA[18], the list is further reduced to DATA[19,20]={100,105}. Again compare K with DATA[20], since K<DATA[20] reduce the list to DATA[20,20]={100} which when searched again contains the search element. The element is searched successfully.

Searching element	Check K,DATA[P]	p,q,r,t values	
K=100		p=13,q=8,r=5	n=0, m=0 since F_8+0=n+1
	K>DATA[13]=70	p=13,q=8,r=5	p=p+m since K>DATA[p]
		p=18,q=3,r=2	
	K>DATA[18]=95	p=20,q=1,r=1,t=1	
	K<DATA[20]=105	p=19,q=1,r=0	
	K=DATA[19]=100		Element found

Figure 16.13 Trace of Fibonacci search

Complexity: The time complexity is measured in terms of the number of comparisons taken to locate ITEM in DATA which contains n elements. Since each comparison reduces the array size in half, f(n) comparisons are required to locate ITEM where

$$2f(n)>n \text{ or } f(n)=\log n+1$$

Limitations:

1. List must be already sorted.
2. Direct access to the middle element in any sublist is done that requires a sorted array to hold the data, which will be very expensive when there are many insertions and deletions.

Program 16.11

```
#include<iostream.h>
#include<conio.h>
void sort(int a[],int n)
{
 int i,j;
 for(i=n-1;i>0;i--)
  for(j=0;j<i;j++)
 if(a[j]>a[j+1])
 a[j]=(a[j]+a[j+1])-(a[j+1]=a[j]);
}
```

```
int fib(int c)
{
 int f1=-1,f2=1,f3;
 while(f2<=c)
 {
  f3=f1+f2,f1=f2,f2=f3;
  if(f3==c)
   return f3;
  if(f3>c)
   return f1;
 }
 return 0;
}
void fsearch(int a[],int x,int y,int s)
{
 while(x<y)
 {
  int count=0,c1=1,flag=0,i,mid;
  for(i=x;i<y;i++)
  count++;
  mid=fib(count);
  for(i=x;i<y&&(c1==1);i++)
  {
   flag++;
   if(flag==mid)
    mid=x,c1=0;
  }
  if(a[mid]==s)
  {
   cout<<"Search Element"<<s<<"Is Found";
   getch();
   return;
  }
  if(a[mid]>s)
   y=mid-1;
  else
   x=mid+1;
 }
 cout<<"Search Element"<<s<<"Is Not Found";
}
void main()
{
 int i,a[30],n,search;
 clrscr();
 cout<<"Enter Size Of Array:";
 cin>>n;
 cout<<"Enter"<<n<<"Elements:";
 for(i=0;i<n;i++)
```

```
 cin>>a[i];
 cout<<"Enter Search Element:";
 cin>>search;
 sort(a,n);
 fsearch(a,0,n,search);
}
```

Output

```
Enter Size Of Array:10
Enter 10 Elements:4 2 6 3 1 5 7 8 10 21
Enter Search Element:8
Search Element 8 Is Found

Enter Size Of Array:5
Enter 5 Elements:6
2
4
3
7
Enter Search Element:10
Search Element 10 Is Not Found
```

SUMMARY

- The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary.
- Insertion sort inserts each element of the array into its proper position leaving progressively larger stretches of the array sorted.
- Quicksort is a relatively simple sorting technique using the divide-and-conquer recursive procedure.
- Merge sort is a fast, stable sorting routine with guaranteed O(n log n) efficiency.
- The idea of Shell sort is to arrange the data sequence in a two-dimensional array and sort the columns of the array.
- Radix sort keeps the elements in order by comparing the digits of the numbers.
- Heap sort is a relatively simple technique built upon the heap data structure.
- Searching is the process of finding the location of a given element in a set of elements.
- Linear is the technique of searching a given item in the list by comparing each and every element in the list sequentially.
- Binary search is a technique of searching an item in the ordered list by dividing the list into two halves with each comparison until the item is found or not.
- The Fibonacci search is a technique of searching a sorted array using the divide and conquer principle.
- Fibonacci search determines the next element for comparison based on the sequence of the Fibonacci number and it performs a search technique only on the ordered list.

EXERCISES

Fill in the Blanks

1. _____ algorithms have the same complexity as $O(n^2)$
2. Merge sort and quick sort follow _____ principle.
3. The time complexity of merge sort is _____.
4. Shell sort performs sorting on an array by _____.
5. Radix sort uses _____ to sort the elements.

Multiple-choice Questions

1. Which of the following sorting algorithm is of divide-and-conquer type?
   a. Bubble sort
   b. Insertion sort
   c. Quick sort
   d. All of the above
2. The complexity of Bubble sort algorithm is
   a. $O(n)$
   b. $O(\log n)$
   c. $O(n^2)$
   d. $O(n \log n)$
3. The complexity of the merge sort algorithm is _____.
   a. $O(n)$
   b. $O(\log n)$
   c. $O(n^2)$
   d. $O(n \log n)$
4. In a heap tree
   a. Values in a node are greater than every value in the left subtree and smaller than the right subtree
   b. Values in a node are greater than every value of its child
   c. Both a and b
   d. None
5. Finding the location of the element with a given value is _____.
   a. Traversal
   b. Search
   c. Sort
   d. None
6. The worst case occurs in a linear search algorithm when _____.
   a. Item is somewhere in the middle of the array
   b. Item is not in the array at all
   c. Item is the last element in the array
   d. Item is the last element in the array or is not there at all
7. The complexity of linear search algorithm is
   a. O(n)
   b. O(log n)
   c. $O(n^2)$
   d. O(n log n)
8. The complexity of binary search algorithm is
   a. O(n)
   b. O(log n)
   c. $O(n^2)$
   d. O(n log n)

9. Which of the following is not the required condition for a binary search algorithm?
   a. List must be sorted
   b. Need of direct access to the middle element in any sublist
   c. Need to insert and/or delete elements in list
   d. None

Short-answer Questions

1. Define various sorting algorithms.
2. Prove that any comparison-based algorithm to sort 4 elements requires at least 5 comparisons.
3. In how many ways can two sorted arrays of combined size N be merged?
4. What are the properties of a heap?
5. What is the running time of heap sort on an array A of length n that is already sorted in increasing order? What about the same in decreasing order?
6. Define linear search and give its complexity.
7. Explain the time complexity of binary search and Fibonacci search.

Essay Questions

1. Give illustrations on:
   a. Bubble sort
   b. Insertion sort
   c. Selection sort
2. Explain in detail about
   a. Quick sort
   b. Merge sort
3. Write short notes on
   a. Shell sort
   b. Radix sort
4. Trace out the heap sort algorithm for the following list:
   {25, 44, 55, 99, 30, 37, 15, 10, 2, 4}.
5. Modify the BUILD-HEAP algorithm to create a Min Heap and explain it with an example.
6. Compare various sorting algorithms in their complexities.
7. Discuss about (i) linear search and (ii) binary search.
8. Give an illustration on Fibonacci search.

Index

Data Structures and Algorithms Using C++

Ananda Rao Akepogu | **Radhika Raju Palagiri**

Data Structures and Algorithms Using C++ helps students to master data structures, their algorithms and the analysis of complexities of these algorithms. Each chapter includes an Abstract Data Type (ADT) and applications along with a detailed explanation of the topics. This book meets the requirements of the course curricula of all Indian universities.

Salient Features

- Pattern matching and tries, tournament trees and red–black trees are extensively discussed by providing suitable examples.
- Extensive visual figures help in understanding data structures and their operations.
- The book has a large number of programs, multiple-choice questions, fill in the blanks and other exercises to help students in their course and also in competitive exams.

Ananda Rao Akepogu is Professor and Head of the Computer Science and Engineering Department, and Vice Principal of Jawaharlal Nehru Technological University, Anantapur College of Engineering Anantapur, Andhra Pradesh.

Radhika Raju Palagiri is Ad Hoc Lecturer in the Computer Science and Engineering Department, Jawaharlal Nehru Technological University, Anantapur College of Engineering Anantapur, Andhra Pradesh.

ISBN 978-81-317-5567-9

9 788131 755679

www.pearsoned.co.in

PEARSON