AIKMAN SERIES

DATA STRUCTURES in C++

MUHAMMAD TAUQUEER

AIKMAN BOOK COMPANY

AIKMAN SERIES

DATA STRUCTURES in C + +

Written by:

C M Aslam
M.Sc. (Comp. Sc.), M.Sc. (PU)
Former Faculty Member, Petroman Computer Training Institute, Ministry of Science & Technology, Lahore Campus

T A Qureshi
B.Sc. (Hons) UET, Lahore
M.Sc. (Engg) ASU, USA

Can be had from:

Aikman Book Company
Qazafi Market, Urdu Bazaar, Lahore

بسم اللہ الرحمن الرحیم

Aikman Book Corporation
URDU BAZAR, LAHORE.

Published by:

Aikman Book Corporation, Urdu Bazar, Lahore.

First Published in 2002 as Data Structures in C++

Edition 2013-14
Copies 1000

Typeset in Monotype®
Times New Roman, Courier New and Arial Typography

Printed at NB Printers, Lahore.

Rs. 187

PREFACE

Computer Science, in essence, is the study of data structures and their manipulation in the computer memory. An understanding of basic concepts of the subject is, therefore, essential to achieving excellence in the field of computer science. This book presents the fundamentals of the subject and intends to provide basic concepts involved in manipulating various data structures to our students in simple language with the help of small and simple algorithms. The book is suitable for use in self-study, directed study and as a class text.

C++ language has been used to implement the algorithms. The syntax and rules of Turbo C++ (Version 3.0) and Borland C++ (Version 5.0) for MS DOS environment have been used in the book. All example programs have been tested using Turbo C++. With very few exceptions, no changes should be necessary if other C++ compilers are used.

Throughout this book a number of program examples have been given. The source code of program examples are available for downloading at Aikman Book Corporation's website **http://www.aikman.com.pk**. Readers are encouraged to download the source code of programming examples to save the time that will be required for writing and debugging the source code of a program example.

I am also indebted to Professor N. A. Qureshi for his assistance in reading and improving the language of the manuscript.

Author

TABLE OF CONTENTS

1 INTRODUCTION

DATA & INFORMATION

Data consists of values or a set of values collected for a specific purpose. These are the raw facts. The facts may be about human beings, natural phenomena, ideas, etc. These may consist of numeric values, characters, pictures, sounds or any combination of these. A single unit of values in the data is called the data item. The processed data that gives useful meaning is called information.

The collected data is organized into a hierarchy of fields, records and files. The basic terminologies that are used in data structure are defined below.

Entity: An entity is something that has certain attributes or properties which may be assigned values. For example, variables used in a programming language are entities used to assign values.

Field: A single unit of information is called field. For example, name, address and date of birth represent the fields.

Record: Collection of related fields is called record.

File: Collection of related records is called file.

Primary Key: A field that may uniquely determine the record in a database is called primary key. The values in the key field are called keys or key values.

DATA STRUCTURES

All programming languages provide some basic data types. The basic data types are usually sufficient to solve simple problems. The complex problems require more complex data objects. Most programming languages provide facilities for combining basic data types into complex data types.

A data structure is a collection of basic data types combined according to the rules of the programming language to create a new user-defined data types. Thus by creating a data structure, a programmer has the facility to create a data type according to the specific needs.

There are many ways to organize data. The possible ways in which data items are logically related and organized is called **data structure**. To understand the logical relationships between data items you must understand the data itself. Data consists of elementary items. A data item consists of a single element such as integers, characters, combination of bits, etc. To solve a specific problem the data is organized in such a manner that its elements can be accessed easily, It is a key step in defining the relationship between data items and then solving the problem

The representation of a particular data structure in the computer memory is called the storage structures. There are many storage structures corresponding to particular data structures. These are divided into the following categories

Primitive or nonlinear data Structure: The data structures whose elements are arranged in non-linear or non-sequence form are called nonlinear data structures. Examples of these data structures include trees and graphs.

Non-Primitive or linear data Structure: The data structure whose elements are arranged in a sequence is called linear data structure. Arrays, Linked Lists, Queues, Stacks, etc. are examples of linear data structures.

OPERATIONS ON DATA STRUCTURES

A data structure is created to define and process complex data. The data in a data structure is processed using certain operations. Some commonly used operations performed on data structures are:

Inserting: adding new data items into a data structure is called inserting.

Deleting: removing data items from a data structure is called deleting.

Searching: finding specific data items in a data structure is called searching.

Traversing: accessing each record or item in a data structure exactly once for processing is called traversing. It is also called visiting.

Sorting: arranging data items in a data structure into a specific order is called sorting.

Merging: combing two lists of data items into a single data list is called merging.

The operation through which data structure is created is called the **Creation** operation. Similarly, the operation through which the created data structure is destroyed is called the **Destruction** operation.

For example, to create a variable of integer type in C++, the statement is written as:

```
int a, b;
```

The above statement creates a memory space of two bytes for each variable a & b.

The operations on data structures are represented through algorithms or by directly using the programming language instructions. The data is processed by the computer through the program instructions.

ALGORITHM & ALGORITHMIC NOTATION

The step-by-step procedure to solve a particular problem is called algorithm. The algorithmic notations are used to represent different steps in solving a problem. It is the general solution of the problem. After solving the general solution, the program is written in a programming language by following the given algorithm.

An algorithm consists of several parts and different types of statements. These are explained below:

Name of algorithm Every algorithm is given a name that represents the purpose of the algorithm. For example:

- an algorithm to find the sum of two numbers is given the name "sum".
- an algorithm to find the maximum number in an array is given the name "max".

Introductory Comment The algorithm name is followed by a brief description of the tasks that the algorithm performs.

Steps The algorithm consists of a sequence of numbered steps. Each step describes one task to be performed. The instructions in a computer program are written according to these steps.

Comments

Comments are used to explain purpose of a step or statement. These may be given at the end of each statement or at the beginning of the step. In C++, the comments are given starting with double slash (//). In algorithms, these are written between square brackets [ ].

Variable name

A variable is an entity that possesses a value. The name of the variable is chosen according to the nature of value it is to hold. For example, a variable that is used to hold the maximum value is named as Max. Usually, variable names in algorithm are written in capital letters. A variable name consists of letters, numeric digits and some special characters. It always begins with a letter. Blank spaces are not allowed within variable names.

Operators

Arithmetic (+, –, *, /), Relational (<, >, >=, <=, etc.) & Logical (AND, OR, NOT) operators are used to describe various mathematical operations.

Assignment Statement

The assignment statement is used to evaluate an expression and assign the calculated value to the variable. The assignment operator "=" sign is used for this purpose. For example, to evaluate the expression N+M and assign its value to a variable S, the assignment statement is written as:

```
S = N+M
```

Algorithm – Sum

Write an algorithm to get two numbers and to calculate their sum.

```
1.  INPUT first number in N
2.  INPUT second number in M
3.  Add N to M and store result in S, i.e. S = N + M
4.  Print S
5.  End
```

Program

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

class sum
   {
   private:
    int n, m,s;
   public:
    input()
        {
         cout<<"Enter first number ? ";
         cin>>n;
         cout<<"Enter second number ? ";
         cin>>m;
        }
   print()
        {
         s= n + m;
         cout<<"Sum = "<<s;
        }
  };

main()
 {
  sum obj;
  clrscr();
  obj.input();
  obj.print();
  getch();
 }
```

Algorithm – Exchange Values

Write an algorithm to exchange values of two variables.

```
1. N = 10      [assign value 10 to variable N]
2. M = 20      [assign value 20 to variable M]
3. Temp = N    [assign value of N to variable Temp]
4. N = M       [assign the value of M to N]
5. M = Temp    [assign the value of Temp, i.e.
                previous value of N to M]
```

Program

```
#include<iostream.h>
#include<conio.h>
class sum
   {
   private:
    int n, m, t;
   public:
    input()
        {
         cout<<"Enter first number in n ? ";
         cin>>n;
         cout<<"Enter second number in m ? ";
         cin>>m;
        }
     exchange()
        {
         t= n;
         n=m;
         m=t;
         cout<<"\nValues after Exchange\n\n";
         cout<<"Values in m ="<<m<<endl;
         cout<<"Values in n ="<<n<<endl;
        }
  };

main()
{
sum obj;
clrscr();
obj.input();
obj.exchange();
getch();
}
```

INPUT & OUTPUT STATEMENTS

Input or Read statement is used in algorithms to enter data into a variable. The statement is written as:

```
Input variable-name
```

For example, to input a value into variable N, the Input statement is written as:

```
Input N
```

Similarly, to print a message or contents of a variable, PRINT statement is used with the following format.

```
PRINT message or variable name
```

The message is written within double quotes. To print the contents of a variable, it is written without using double quotes. In C++, usually the "cin" object is used to get input from the keyboard and "cout" object is used to print the output on the screen.

SELECTION STATEMENTS

Selection statements are used for making decisions. The decision is made by testing a given condition. After testing a condition, a statement or a set of statements are executed or ignored. The structure that implements this logic in a programming language is called conditional structure. A conditional statement is represented by the IF structure. It has one of the following two forms:

```
1-   IF condition THEN
        statement(s)
     END IF

2-   IF condition THEN
        Block A
     ELSE
        Block B
     END IF
```

In the first "IF structure", the statements following the "IF structure" are executed when the given condition is true. Otherwise, these statement(s) are ignored.

In the second "IF structure", one of the two blocks of statements are executed. If the given condition is true, then the block of statements under "IF statement" is executed otherwise the block following the "ELSE statement" will be executed.

Algorithm – Greater Number

Write an algorithm to find the greater of the two given numbers.

```
1.   Start
2.   Input two numbers in A & B
3.   IF A > B THEN
        PRINT "A is greater"
     ELSE
      PRINT "B is greater"
     END IF
4.   Exit
```

Program

```
#include<iostream.h>
#include<conio.h>
class ab
   {
   private:
    int n, m;
   public:
    input()
        {
         cout<<"Enter first number ? ";
         cin>>m;
         cout<<"Enter second number ? ";
         cin>>n;
        }
    test()
        {
         if(m>n)
           cout<<"First number is greater";
         else
           cout<<"Second number is greater";
        }
  };
main()
{
ab obj;
clrscr();
obj.input();
obj.test();
getch();
}
```

Algorithm — Greater Number

Write an algorithm to find out the largest number from a given set of three numbers by using nested IF Structure.

```
1.   Start
2.   Input Three Numbers A, B & C.
3.   IF A > B THEN
       IF A > C THEN        [Nested If Structure]
          PRINT "A is greater"
       ELSE
          PRINT "C is greater"
       END IF
     ELSE
       IF B > C THEN
          PRINT "B is greater"
       ELSE
          PRINT "C is greater"
       END IF
     END IF
4.   Exit
```

Program

```
#include<iostream.h>
#include<conio.h>
class three
 {
 private:
  int a, b, c;
 public:
  input()
     {
      cout<<"Enter first number ? ";
      cin>>a;
      cout<<"Enter second number ? ";
      cin>>b;
```

```
        cout<<"Enter third number ? ";
        cin>>c;
       }

    test()
        {
         if(a>b)
           if(a>c)
              cout<<"First value is greater";
           else
            cout<<"Third value is greater";
         else
           if(b>c)
             cout<<"Second value is greater";
           else
            cout<<"Third value is greater";
         }
  };

main()
{
three obj;
clrscr();
obj.input();
obj.test();
getch();
}
```

LOOPING STATEMENTS

The looping statements are used to execute certain statement(s) repeatedly. The statements that are executed repeatedly are called body of loop. In algorithm notation, "Repeat" statement is used to execute the statements repeatedly. The "Repeat" statement has two different forms:

- Repeat For Loop
- Repeat While Loop

Repeat FOR Loop

"Repeat For" loop is used to execute statement(s) for a specified number of times. This loop is also called the counter loop. Its general format is:

```
REPEAT FOR Index = I-value TO F-value BY S-value
```

where

I-value	represents the initial value for index variable.
F-value	represents the final value.
S-value	represents the step value, i.e. increment/decrement value. Its use is optional. If it is omitted, the value of the index variable is incremented by 1 after each iteration.

The REPEAT FOR loop structure uses an index variable to control the number of iterations of the loop. In the above syntax 'Index' is the control variable.

Algorithm – Natural Numbers

Write an algorithm to print first ten natural numbers using REPEAT FOR Structure.

```
1.   Start
2.   REPEAT Step-3 FOR C = 1 To 10 BY 1
3.   PRINT C
     [End of Step-2 Loop]
4.   Exit
```

Program

```
#include<iostream.h>
#include<conio.h>
class abc
   {
   public:
    print()
        {
         for( int c = 1; c<=10; c++)
          {
            cout<<c<<endl;
          }
        }
  };
```

```
main()
{
abc obj;
clrscr();
obj.print();
getch();
}
```

Repeat WHILE Structure

This loop structure is used to execute a statement or a set of statements repeatedly until the given condition remains true. It is also referred to as conditional loop. Its general syntax is:

```
REPEAT WHILE (condition)
     Body of loop
[end of loop]
```

The condition is tested at the beginning of the loop. There must be a statement in the body of the loop that will change the condition during the execution of the program to bring it near the specified condition.

Algorithm – Natural Numbers

Write an algorithm to print first ten natural numbers using REPEAT WHILE loop Structure.

```
1.   C = 1
2.   Repeat Steps 3 TO 4 WHILE (C <=10)
3.   PRINT C
4.   C = C + 1
     [End of Step-2 Loop]
5.   Exit
```

Program

```
#include<iostream.h>
#include<conio.h>
class abc
   {
   public:
    print()
        {
         int c=1;
         while( c<=10 )
          {
           cout<<c<<endl;
           c=c+1;
          }
        }
   };

main()
{
abc obj;
clrscr();
obj.print();
getch();
}
```

Sub-Algorithms

A sub-algorithm is a complete and independently defined algorithmic module. It is called by the main algorithm or by some other sub-algorithm. It can receive values from the calling algorithm. The general form of a sub-algorithm is:

```
Name (p1, p2, ----, pn)
 {
   body of sub-algorithm
 }
```

where

Name: represents the name of sub-algorithm

p1, p2, ----, pn: represent the parameters or arguments

The RETURN statement is used as the last statement of the sub-algorithm to return the control back to the calling program.

Type of Sub-Algorithm

The sub-algorithm is divided into two categories:

1- **Function sub-algorithm**

2- **Procedure sub-algorithm**

Both sub-algorithms are independent modules and are written to perform specific tasks. The difference between the two is that:

- the function sub-algorithm returns a single value to the calling algorithm through RETURN statement.
- procedure sub-algorithm may return more than one values. It returns values through parameters.
- the function sub-algorithm is called as an expression or as a variable. The returned value is used in an expression.
- the procedure sub-algorithm is called by the CALL statement in algorithms.

Function: Mean (A, B, C)

Write a function sub-algorithm to find the average of three numbers.

```
      MEAN(A, B, C)
1     AVG = (A+B+C)/3
2     Return(AVG)
```

Program

```
#include<iostream.h>
#include<conio.h>
class abc
   {
   public:
```

```
        float mean (float a, float b)
            {
              return (a+b)/2;
            }
    };

main()
{
abc obj;
float m,n;
clrscr();
cout<<"Enter first number ? "; cin>>m;
cout<<"Enter second number ? "; cin>>n;
cout<<"Mean of "<<m<<" & "<<n<<" = "<< obj.mean(m,n);
getch();
}
```

Procedure Algorithm – Exchange Values

Write a procedure sub-algorithm to interchange values of two variables.

```
Exchange(A, B)
1     TEMP = A
2     A = B
3     B = TEMP
4     RETURN
```

Program

```
#include<iostream.h>
#include<conio.h>
class abc
   {
    public:
    exchange(int &a, int &b)
        {
         int t;
         t = a;
         a = b;
         b = t;
        }
  };

main()
```

```
{
abc obj;
int m, n;
clrscr();
cout<<"Enter first number in n ? ";
cin>> n;
cout<<"Enter second number in m ? ";
cin>> m;
obj.exchange(n, m);
cout<<"\nValues after Exchange\n\n";
cout<<"Values in m ="<<m<<endl;
cout<<"Values in n ="<<n<<endl;
getch();
}
```

EXERCISES

Q.1 Fill in the blanks.

1. The raw facts collected for specific purpose are called _____.
2. The processed data is called _____.
3. A single unit of information is known as _____.
4. The collection of related records is called _____.
5. The ways in which a data item is logically related and organized is called _____.
6. The data structure whose elements are arranged in a sequence is called _____ data structure.
7. The non-linear data structure is also referred as _____.
8. The non-primitive data structure is also referred as _____.
9. The step by step procedure to solve a problem is called _____.
10. A computer program is developed in any programming language by following the given _____.
11. An algorithm is a sequence of _____.
12. In an algorithm _____ statement is used as a general input statement and _____ statement is used to print data.
13. In an algorithm Repeat _____ statement is also called the counter loop statement.
14. A _____ is a complete and independently defined algorithmic module.
15. The two types of sub-algorithm are: _____ and _____.

Q.2 Mark True or False.

1. A collection of data is called an entity.
2. A sub-algorithm is also an independent algorithm.

3. The way in which the data elements are logically related and organized is called data structure.
4. The process of finding a specific data items from a data structure is called the Traversing operation.
5. The process of removing a specific data item from a data structure is called the deleting operation.
6. Arranging the data items into an ordered form is called sorting operation.
7. The traversing operation is also known as traveling operation.
8. The Searching operation is also referred as visiting operation.
9. The program developed in any programming language is also called algorithm.
10. The procedure sub-algorithm returns only a single value after execution.

Q.3 Select a correct answer from multiple choices.

1. a. A set of raw facts collected for a specific purpose is called information.
   b. A set of raw facts collected for a specific purpose is called a record.
   c. A set of raw facts collected for a specific purpose is called data.
   d. A set of raw facts collected for a specific purpose is called a database.
2. a. The processed data is called Database.
   b. The processed data is called Information.
   c. The processed data is called Data Structure.
3. a. A single unit of information is known as Data.
   b. A single unit of information is known as a Record.
   c. A single unit of information is known as a Data File.
   d. A single unit of information is known as a Field.

4. a. The way in which the data items are logically related & organized is called data file.
   b. The way in which the data items are logically related & organized is called data collection.
   c. The way in which the data items are logically related & organized is called data structure.

5. a. The data structure whose elements are arranged in a sequence is called non-primitive data structure.
   b. The data structure whose elements are arranged in a sequence is called primitive data structure.
   c. The data structure whose elements are arranged in a sequence is called simple data structure.
   d. The data structure whose elements are arranged in a sequence is called ordered data structure.

6. a. The step by step procedure to solve a problem is called computer program.
   b. The step by step procedure to solve a problem is called daily life program.
   c. The step by step procedure to solve a problem is called algorithm.
   d. The step by step procedure to solve a problem is sub-routine program.

7. a. In an algorithm "cin" statement is used as a general input statement.
   b. In an algorithm "Get" statement is used as a general input statement.
   c. In an algorithm "scan" statement is used as a general input statement.
   d. In an algorithm "Input" statement is used as a general input statement.

8.
a. In an algorithm Repeat While statement is also called the counter loop statement.
b. In an algorithm Repeat statement is also called the counter loop statement.
c. In an algorithm Repeat For statement is also called the counter loop statement.
d. In an algorithm Repeat Counter statement is also called the counter loop statement.

9.
a. A function is a complete and independently defined algorithmic module.
b. A sub-algorithm is a complete and independently defined algorithmic module.
c. A procedure is a complete and independently defined algorithmic module.

Q.4 **Write an algorithm to print the prime numbers of first N natural numbers.**

Q.5 **Write an algorithm to print the odd numbers of first N natural numbers.**

Q.6 **Write an algorithm to print the sum of even numbers of first N natural numbers.**

Q.7 **Write a program to find the largest number from four numbers.**

Q.8 **Write a function sub-algorithm to find the exponential power if b represents base and p its power.**

Q.9 **Differentiate between the following:**

a. **Primitive & Non-primitive Data Structures**
b. **Counter loop & Conditional loop**
c. **Function sub-algorithm & Procedure sub-algorithm**
d. **Algorithm & Computer program**

2 ARRAYS

ARRAYS

An array is the simplest type of data structure. It consists of an ordered collection of elements which are all of the same type. It can be thought of as a box with multiple compartments in which each compartment is capable of storing only one data item.

Each array is given a name and the elements of the array are accessed with reference to their position. Each element in the array has a unique position. This is called its index. The array name and index of an element are used to directly reference the element within the array. The following notation is used:

```
array-name[element-index]
```

For example, the elements of an array A are denoted as:

$A_1, A_2, A_3, \ldots, A_n$

or

```
A(1), A(2), A(3),...A(n)
```

or

```
A[1], A[2],...A[n]
```

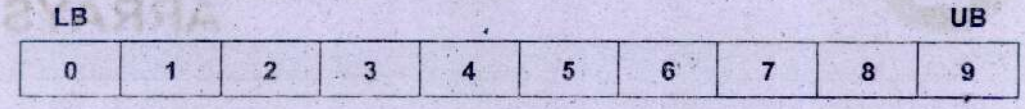
In the above notations, '`A`' is the name of the array and numbers `1, 2, 3,...n` are indexes of individual elements.

The maximum number of elements in the array is called the size of the array. The index of first element of array is called its lower bound (LB) and the index of the last element is called its upper bound (UB). The index can begin at 0 or 1. Maximum number of elements in a one-dimensional array is computed as under:

```
Maximum number of elements = UB - LB + 1
```

In C++, the index of the first element is 0. Thus, an array "abc" of integer type having 10 elements is declared as:

```
int abc[10];
```



In the above example, index of the first element is 0 and that of the last element is 9. Thus,

```
Number of elements in the array    = UB - LB + 1
                                   = 9 - 0 + 1
                                   = 10
```

The size of array can be declared either **statically** or **dynamically**. In static declaration, the number of elements of the array is specified at the time of array declaration in the program and it remains fixed during program execution. But in dynamic declaration, the size of the array can change during program execution.

Representation of Linear Array In Memory

An array occupies a continuous block of memory. This memory block is divided into equal parts. Each part of the block represents one element of the array. In C++, these elements are numbered from **0** to **n–1**, where **n** is the total number of elements of the array.

Each element of the array also has a unique memory address. The starting address of array is called the **base address** of the array. Since the elements of an array occupy a continuous block of memory, if base address of the array is known, the address of any element of the array can be easily computed.

The address of an element is determined arithmetically by adding a suitable offset to the base address of the array. Suppose `X` is an array and the index for its first element is `1`. The base address of array is `L`, and `C` represents the memory size of each element in bytes. The memory address of an element with index `k` is computed as:

$$L(X[k]) = L_o + C * (k - 1)$$

For example, if an array X has 5 elements, each of 2 bytes length, then to find out the address of third element of X when the base address of the array is $L_o = 200$:

```
L(X[3]) = 200 + 2 * ( 3-1)
        = 200 + 4
        = 204
```

Thus the address of the third element of array X is 204. The memory address of an element of the array is used to access its data directly.

Operations on Linear Arrays

Several operations can be performed on linear arrays. The most important of these operations include traversing, inserting, deleting and searching & sorting.

Traversing Operation

In traversing operation, each element of an array is accessed exactly once for processing. This is also called visiting of the array.

An array is traversed to process its data. For example, to compute the sum of values of each element of an array, all elements of the array are accessed exactly once and their values are added. Similarly, to print values of each element of an array, each element of the array is accessed exactly once and the values are printed.

Arrays are linear data structures and simple steps are required to traverse them. The following algorithm explains traversing of arrays.

Algorithm – Traversing Linear Arrays

Write an algorithm to compute the sum of values of elements of a linear array having ten elements.

```
1.    Input Values in array ABC
```

```
        [Initialize 0 to variable SUM]
2.        SUM = 0
3.      [Compute sum of Array ABC]
          REPEAT FOR C = 0 to 9
          SUM = SUM + ABC[C]
        [End of Loop]
        [Print the calculated sum]
4.        PRINT SUM
5.      EXIT
```

C=N to 0 by -1

Program

```
#include<iostream.h>
#include<conio.h>
class temp
 {
 private:
 int abc[10], sum;
 public:
 temp()
 {
 sum=0; // constructor to initilize 0
 }

// member function to input data into array
 input()
 {
 cout<<"Enter 10 values & Press Enter after typing each value\n";
 for(int i=0; i<=9; i++)
 cin>>abc[i];
 }

// member function to compute and print sum of array
 print()
 {
 for(int i=0; i<=9; i++)
 sum = sum+abc[i];
 cout<<"Sum = "<<sum;
 }
 };

main()
{
```

```
temp obj;
clrscr();
obj.input();
obj.print();
getch();
}
```

Algorithm – Traversing Linear Arrays

Write an algorithm to print out even values stored in a linear array ABC having 10 elements.

```
1.  REPEAT Step-2 FOR C = 0 to 9
2.  IF ABC[C]/2 = 0 THEN
     PRINT ABC[C]
    END IF
    [End of loop]
3.  EXIT
```

Program

```
#include<iostream.h>
#include<conio.h>
class temp
 {
 private:
 int abc[10];
 public:

// member function to input data into array
 input()
 {
 cout<<"Enter 10 values & Press Enter after typing each value\n";
 for(int i=0; i<=9; i++)
 cin>>abc[i];
 }

// member function to print even numbers
 print()
 {
 for(int i=0; i<=9; i++)
 if(abc[i]%2 == 0)
 cout<<abc[i]<<endl;
```

```
};

main()
{
temp obj;
clrscr();
obj.input();
obj.print();
getch();
}
```

Inserting Operation

In inserting operation, new items are added into an array. A new item can be added:

1. at the end of an array without disturbing other elements of the array if there is an empty element.
2. at some specified location within the array.

Inserting at the End of Array

Consider an array 'Temp' having 10 elements with its first three elements having values as shown below:

Temp

value	66	72	36							
index	0	1	2	3	4	5	6	7	8	9

To insert a value 78 at end of the array, the assignment statement is written as:

```
Temp[3] = 78
```

New values can be inserted in the above array from fourth to tenth element as these elements are empty.

Algorithm – Insertion at End

Write an algorithm to add 6 values into elements at the end of array 'Temp' that has only four items stored in its first four elements.

```
1.    REPEAT Step-2 TO 3 FOR I = 4 to 9
2.    INPUT value in N
3.    Temp [I] = N
      [End of Step-2 Loop]
4.    EXIT
```

Program

```
#include<iostream.h>
#include<conio.h>
class temp
 {
 private:
 int abc[10];
 public:
 assign()
        {
        abc[0]=66;
        abc[1]=72;
        abc[2]=36;
        abc[3]=78;
        }

// member function to insert values
 input()
 {
  cout<<"Enter 6 values & Press Enter after typing each value\n";
  for(int i=4; i<=9; i++)
 cin>>abc[i];
 }

// member function to print values
 print()
  {
  for(int i=0; i<=9; i++)
   cout<<abc[i]<<endl;
  }
 };
```

```
main()
{
temp obj;
clrscr();
obj.assign();
obj.input();
cout<<"Values in Array "<<endl;
obj.print();
getch();
}
```

Inserting Value at Specified Location in an Array

A new value can also be inserted at a specified location in an array. To insert a new value, the values of all the existing elements are moved one step forward or backward to make space for the new value. The following algorithm explains this process.

Algorithm – Inserting value at a specified location

Write an algorithm to insert a value *M* at location *pos* in array *ABC* having *N* elements.

```
1.  Input value in M
2.  Input position in POS
    [Move elements one step forward]
3.  REPEAT step - 4 TO 5 WHILE N >= POS
4.  ABC[N+1] = ABC[N]
5.  N = N - 1
    [End of step-3 Loop]
6.  [Insert value M at Position POS]
    ABC[POS] = M
7.  EXIT
```

Program

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

```
class temp
 {
 private:
 int abc[5];
 public:

// member function to assign values to array
 assign(int p[])
        {
        for(int i=0;i<=3;i++)
        abc[i]=p[i];
        }

// member function to insert a value into array
 insert(int loc, int val)
        {
        for(int i=4;i>=loc;i--)
        abc[i+1]=abc[i];
        abc[loc]= val;
        }

// member function to print all values
 print(int n)
        {
        for(int i=0; i<=n;i++)
        cout<<abc[i]<<endl;
        }
};
main()
{
temp obj;
int pos, n, a[4]={44,55,6,3};
clrscr();
obj.assign(a);
cout<<"Values before insertion"<<endl;
obj.print(3);
cout<<"Enter value to insert ? ";
cin>>n;
cout<<"Enter position to insert ? ";
cin>>pos;
if(pos >=5)
        {
        cout<<"Invalid Location ";
        return 0;
        }
obj.insert(pos,n);
cout<<"Values after insertion"<<endl;
obj.print(4);
getch();
}
```

DELETING OPERATIONS

In deleting operation, the element(s) of an array are removed. Like insertion, deletion can be performed at the end or at any position in the array.

Deleting Items at the End

Deleting or removing a data item at the end of the array is simple and easy. To remove the last element from the array, the last element is accessed and deleted by placing null value or by decreasing the size of array by one element if it is the last element of the array.

Consider an array of names of 5 countries as shown below:

COUNTRY

Saudi Arabia	England	America	Pakistan	India
1	2	3	4	5

To remove the last element of the array (i.e. India), move to the last element and then delete it by placing the null value. It can also be deleted by decreasing the size of array by one element as it is the last element of the array.

Deleting Item from a Specified Location

When an item is removed from a specified location within an array, the items from that location up to the end of array are moved one location towards the beginning of the array. The size of the array also decreases.

Consider an array of names of 5 countries as shown below:

COUNTRY

Saudi Arabia	England	America	Pakistan	China
1	2	3	4	5

To delete America at position 3, the contents of locations 4 and 5 are moved one step towards the beginning of the array. Thus "Pakistan" will be shifted to location 3 and "China" to location 4. After shifting the values, the size of the array also decreases.

Algorithm – Deleting item from array

Write an algorithm to delete the value at location K of an array "country" having N elements.

```
1.   Input location K
2.   IF K > N then
       PRINT "invalid location"
       Return
     END IF
     [Move each element from the specified location
     one step towards the beginning. The item at the
     specified location is automatically deleted]
3.   REPEAT FOR C = K TO N-1
           Country[C] = Country[C+1]
     [End of Loop]
4.   EXIT
```

Program

```
#include<iostream.h>
#include<conio.h>
class temp
{
private:
int abc[5];
public:

// member function to assign values
assign(int p[])
      {
      for(int i=0;i<=4;i++)
      abc[i]=p[i];
      }
```

```
// member function to delete a value
 del(int loc)
        {
        for(int i=loc;i<=4;i++)
        abc[i]=abc[i+1];
        abc[4]=0;
        }

// member function to print values of array
 print()
        {
        for(int i=0; i<=4;i++)
        cout<<abc[i]<<endl;
        }
};

main()
{
temp obj;
int pos, n, a[5]={44,55,6,3,66};
clrscr();
obj.assign(a);
cout<<"Values before deletion"<<endl;
obj.print();
cout<<"Enter position to delete ? ";
cin>>pos;
if(pos >=5)
        {
        cout<<"Invalid Location ";
        return 0;
        }

obj.del(pos-1);
cout<<"Values after deletion"<<endl;
obj.print();
getch();
}
```

Searching and Sorting Arrays

The process of finding a specific data item and its location is called searching. If the data item is found then the search is said to be successful. Otherwise, it is called unsuccessful.

Searching is an important operation in data management. It is used for data modification together with inserting, deleting and updating processes. For example, when a data item is to be deleted, it is first searched and then deleted.

Similarly, the process of rearranging items of an array in a specific order is called sorting. The items in an array are usually sorted into ascending or descending order. The data in an array is sorted to expedite searching process.

The details on searching and sorting are given in the chapter on "Sorting and Searching".

Two-Dimensional Arrays

The two-dimensional array consists of rows and columns. It is also called table or matrix. The elements of a two-dimensional array are referenced by two subscripts or index values. A matrix with the same number of rows and columns is called square matrix.

The elements of the array are denoted as:

```
A[r, c]
r - represents the row number
c - represents the column number
```

A two-dimensional array **abc** having 2 rows and 3 columns is shown below.

abc

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)

The total number of elements of a two-dimensional array having m rows and n columns is m x n. For example, an array having 2 rows and 4 columns has 2 x 4 = 8 elements.

Each programming language has different rules to declare multi-dimensional arrays. In C++, a two-dimensional array "temp" of integer data type having 6 rows and 4 columns is declared as:

```
int temp[6][4];
```

Representation of Two-Dimensional Arrays In Memory

Two-dimensional arrays are represented in memory in two ways. These

are Row-major order and Column-major order.

In row-major order, a two-dimensional array is represented in memory by row order. For example, a two-dimensional array having 2 rows and 3 columns is stored in row-major order as:

```
{X[1,1], X[1,2], X[1,3]}, {X[2,1], X[2,2], X[2,3]}
```

In column-major order, a two-dimensional array is represented in memory by column order. For example, a two-dimensional array having 2 rows and 3 columns is stored in column-major order as:

```
{X[1,1], X[2,1]}, {X[1,2], X[2,2]}, {X[1,3], X[2,3]}
```

Algebraic Operations on Matrices

The algebraic operations like addition, subtraction, multiplication, etc. can be easily performed on matrices. The matrices must obey certain rules for performing these operations. These rules are:

- To add/subtract two matrices, the number of rows & columns of the first matrix must be equal to the number of rows & columns of the second matrix.
- To multiply two matrices, the number of columns of first matrix must be equal to the number of rows of the second matrix.

For example, to add two matrices A and B, the corresponding elements of these matrices are added:

$$A = \begin{bmatrix} 1 & 6 \\ 2 & 3 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 5 \\ 1 & 2 \end{bmatrix}$$

$$A + B = \begin{bmatrix} 1+2 & 6+5 \\ 2+1 & 3+2 \end{bmatrix}$$

$$A + B = \begin{bmatrix} 3 & 11 \\ 3 & 5 \end{bmatrix}$$

EXERCISES

Q.1 Fill in the blanks

1. A group of elements of same data type is called _____.
2. Each element of an array is referenced by its position in the array called _____.
3. In C++, the index of first element is _____.
4. The index of the first element of array is knows as _____ of the array.
5. The index of the last element of array is knows as _____ of the array.
6. The maximum number of elements in the array is called _____ of array.
7. The maximum number of elements in the array is computed by the formula _____.
8. The address of first element of an array is called _____ of the array.
9. Two-dimensional array is also known as _____.
10. Each element of a table is referenced by _____ index values.

Q.2 Mark True or False

1. Different elements of an array may be of different sizes.
2. Different elements of an array may be of different types.
3. Each element of an array is accessed by the name of array.
4. The size of array can be declared either statically or dynamically.
5. In static declaration, the size of array can change during program execution.

6. In dynamic declaration, the size of array remains the same during program execution.
7. Accessing each element of an array exactly once is called as traversing the array.
8. The index of first element of array is called its upper bound.
9. The maximum number of elements in the array is called large array.
10. The address of first element of an array is called standard address of the array.
11. The deletion operation is not possible in arrays.
12. A matrix with same number of rows and columns is called square matrix.

Q.3 **Write an algorithm and program in C++ to traverse an array in reverse order.**

Q.4 **How is a linear array represented in the computer memory?**

Q.5 **Explain with example the insertion operation in linear arrays.**

Q.6 **Explain with example the deletion operation in linear arrays.**

Q.7 **Write a C++ program to input the values into a table and print these values on computer screen in tabular form.**

Q.8 **Write an algorithm to calculate total number of odd and even values in an array.**

Q.9 **Write an algorithm to calculate and print the factorial of each element of integer array of N elements.**

Q.10 **Write an algorithm to traverse an integer array with N elements and find whether or not the accessed value is a prime number.**

STRINGS

STRINGS

A string is a sequence of symbols. These symbols may be a collection of alphabetic letters (a-z), numeric digits (0-9) and special characters (e.g. #, $ etc.).

The total number of symbols in a string is called string length. For example, the length of "abcde" is five and the length of "xyz" is three.

The string with no symbol in it has zero length and is called empty or null string. In C++, the end of the string is indicated by a special character called the **null character**. The null character is indicated by '\0' (back slash and zero). A constant string is specified by enclosing the string in single or double quotes, e.g.

"Computer is an Electronic Machine."

String is an important data structure in various applications like word processing, graphics, etc. Therefore, efficient representation and handling of strings plays an important role in data structures.

Representation of Strings into Memory

A string is actually an array of characters. It is represented in the memory as a sequence of storage location. Each storage location stores one character of the string. Each character of string takes one byte.

There are three types of strings. These are:

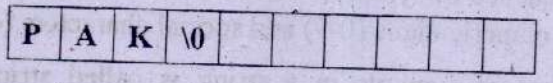
1) **Fixed-length strings**
2) **Variable-length strings**
3) **Linked strings**

Fixed-Length Strings

In fixed-length strings, the amount of memory required for storing a string is fixed at the time the string variable is declared. Its length remains fixed throughout the execution of the program. For example, to declare a string variable "str" with size of 10 and store "Pak" in it, C++ statement is written as:

```
char str[10] = "Pak";
```

In memory, the above string is represented as:



Four locations are occupied by the string in the memory. Three locations for "Pak" and one location for the null character. Six locations remain empty.

This way of storing and processing string usually results in wastage of a large amount of memory. It also creates problems while processing the data and slows down the data accessing speed.

Variable-Length Strings

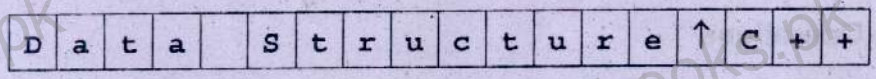
The variable length string is more dynamic and powerful type of string. It is declared without specifying the string size. The size of a variable-length string can be changed during the program execution.

The storage of variable-length strings in memory is handled in different ways. The most commonly used methods are:

- Boundary Marker
- String Descriptors

A boundary marker is a character used to specify the end of the string. Usually the character ↑ (or $$ double dollar signs) is used as boundary marker. In this method strings are stored in memory one after the other separated by boundary markers ↑ (or $$).

For example, two strings "Data Structure" and "C++" are stored in the memory as:



In string descriptor method the pointer array is used to represent the strings having variable lengths. In this method two fields are used to represent a single string. These are:

Length Field It contains the length of the string.

Pointer Field It contains the memory address of the string i.e. the address of first character of the string.

The pointer field points to the memory area where the actual information of the string is stored. The string is accessed through the pointer field. An example is given below to represent the string "I Love Pakistan" by using string descriptor method.

Linked Strings

In linked storage of strings, a string is stored in a sequence of memory cells called nodes. Each node stores a fixed number of characters of the string. Each node also contains an element called the link. The link points to the next node that contains the next part of the string.

The following diagram shows storage of string "I Love Pakistan" in linked storage with four characters per node:

String Operations

Although a string is like an array, the string operations are different from that of the arrays that store numeric data. In numeric data arrays, individual elements of the array are accessed. Whereas in strings, a group of consecutive elements, called sub-string, are usually accessed for processing.

Several operations that can be applied on a string include:

1. **Computing length of string**
2. **Copying a string into another**
3. **Concatenating strings**
4. **Extracting a sub-string from a string**
5. **Pattern matching**
6. **Inserting a string into another string**
7. **Deleting a sub-string from a string**
8. **Searching and replacing a sub-string in a string**

Computing Length of String

Total number of symbols or characters in a string is known as string length. Computing the length of a string is an important string operation. It is usually performed before applying several other string operations. For example, to insert characters in a string, the string length is first computed and then the insertion operation is performed.

To compute the length of the string, the string is traversed and its characters are counted starting from the beginning to the end of the string, i.e. up to the null character.

Algorithm – computing length of string

Write an algorithm to compute the length of a string entered by user during program execution.

```
1.  Input string in variable str
2.  C = 0
```

```
3.    Repeat Step-4 while str[C] != NULL
4.    C = C + 1
      [End of Step-3 Loop]
5.    PRINT "Length of string = ", C
6.    Exit
```

Program

```
#include<iostream.h>
#include<conio.h>
class temp
{
public:

// member function to compute length of string
 int len( char str[])
        {
        int i;
        for(i=0; str[i]!='\0'; i++);
        return i;
        }
};

main()
{
temp obj;
char s[80];
clrscr();
cout<<"Enter string ? ";
cin>>s;
cout<<"Length of string is = "<<obj.len(s);
getch();
}
```

Copying String

The process of copying the contents of one string to another is called the string copying. The string whose contents are copied is called the source string. Similarly, the string into which the contents are copied is called the destination or target string. The entire source string or a part of the source string can be copied to the target string.

Algorithm – Copying Strings

Write an algorithm to copy the contents of string STR1 into an empty string STR2.

```
1.    INPUT string in STR1
2.    C = 0
3.    REPEAT Steps-4 to 5 WHILE STR1[C] != NULL
4.    STR2[C] = STR1[C]
5.    STR2[C+1] = NULL
      [End of Step-3 Loop]
6.    Exit
```

Program

```
#include<iostream.h>
#include<conio.h>
class temp
{
public:

// member function to copy string
      cpystr( char st1[], char st2[])
       {
       for(int i=0;st1[i]!='\0';i++)
       {
       st2[i]=st1[i];
       st2[i+1]='\0';
       }
      }
};
main()
{
temp obj;
char s1[80],s2[80];
clrscr();
cout<<"Enter string ? ";
cin>>s1;
obj.cpystr(s1,s2);
cout<<"Second string = "<<s2;
getch();
}
```

String Concatenation

The process of combining the contents of two or more strings into a single string is called string concatenation. It is an important string operation. The operator 'O' or '||' is used to denote concatenation of two strings. For example, if A = "ABC" and B = "XYZ" then

```
A || B = ABCXYZ
```

The strings are joined without a blank space between them. If two concatenated strings are to be separated by a space, a blank space character must be inserted as shown below:

```
A || "□" || B = ABC XYZ
```

The symbol "□" represents the blank space character. The space character is inserted at the end of first string.

Similarly, more than two strings can also be concatenated into a single string. Suppose, if A = "Aikman C++", B = "Aikman Visual Basic" and C = "Aikman Data Structures" then to concatenate these strings in an empty string X:

```
X = A || "□" || B || "□" || C
```

In the above example, more than one "||" operators have been used to combine the strings A, B and C and blank space characters are inserted at end of each string.

Algorithm – Concatenation

Write an algorithm to combine three strings into a single string:

1. INPUT three strings in ST1, ST2 and ST3
2. Combine strings into a single string using Concatenation
3. Exit

Program

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

```
{
public:

// member function to concatenate strings
        combine(char st1[],char st2[],char st3[], char res[])
        {
        int i=0, r=0;
        for(i=0; st1[i]!='\0'; i++, r++)
        res[r]=st1[i];
        for(i=0; st2[i]!='\0'; i++, r++)
        res[r] = st2[i];
        for(i=0; st3[i]!='\0'; i++, r++)
        res[r] = st3[i];
        res[r]='\0';
        }
};

main()
{
temp obj;
char s1[20],s2[20],s3[20], resultant[80];
clrscr();
cout<<"Enter first string ? ";
cin>>s1;
cout<<"Enter second string ? ";
cin>>s2;
cout<<"Enter third string ? ";
cin>>s3;
obj.combine(s1,s2,s3,resultant);
cout<<"combined string ="<<resultant;
getch();
}
```

Extracting Sub-string from String

The process of taking out a sub-string from a string is called string extracting. It is normally used when a part of the string is to be processed. Three types of information are required to extract a sub-string from a string:

- Name of the string
- Starting position of the sub-string
- Length of the sub-string or position of the last character of the sub-string

This information is used in a function for extracting the substring. The

procedure written for this purpose is usually named as "substring". The general syntax of the procedure is:

```
substring(string, initial, length)
```

where

string specifies the string from which the sub-string is to be extracted

initial specifies the starting position of the substring within the string

length specifies the length of substring

For example, to extract 2 letters from string "Pakistan" starting from the 4th position, the procedure is specified as:

```
substring("Pakistan", 4, 2)
```

Algorithm — extracting Sub-string

Extract the part of string "Pakistan" and print the output as:

```
P
PA
PAK
.....
```

```
1.  STR = "PAKISTAN"
2.  C = 0
3.  REPEAT Step-4 WHILE C <= 7
4.  PRINT substr(STR, 1; C)
    [End of Step-3 Loop]
5.  Exit
```

Program

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

```
class temp
 {
 private:
 char str[20];
 public:
// member function to extract string
        substr(char st1[], int loc, int s)
         {
         for(int i=loc-1; i<=s; i++)
         {
         str[i]=st1[i];
         str[i+1]='\0';
         }
         cout<<str<<endl;
         }
};

main()
{
temp obj;
clrscr();
for(int c=0; c<=7; c++)
obj.substr("PAKISTAN", 1, c);
getch();
}
```

Pattern Matching

The process of finding a sub-string within a string is called pattern matching. The length of sub-string must be less than or equal to the length of the string.

The function used for pattern matching is usually named as "index". Its general syntax is:

```
index(string, pattern)
```

where

string represents the string within which the sub-string is to be searched.

pattern represents the sub-string.

Both the parameters may be string constants or string variables. If the pattern is not found in the string, then a zero value is returned.

The starting position of the sub-string in the string is called cursor

position. The cursor position indicates the position of the left-most character of the sub-string in the string.

For example, if S = "ABCBXCBY" and the sub-string CB is to be searched, then the function is specified as:

```
index(S, "CB")
```

It will return cursor position 3. Similarly `index(S, "CBD")` will return 0 as cursor position because the sub-string "CBD" does not exist in the string S.

The above pattern (i.e. "CB") is compared with each sub-string of S, starting from left to right until the given pattern is matched. The comparison is made as explained below:

1) Compare "AB" to "CB"
2) Compare "BC" to "CB"
3) Compare "CB" to "CB"
4) Compare "BX" to "CB"
5) Compare "XC" to "CB"
6) Compare "CB" to "CB"
7) Compare "BY" to "CB"

In the above example, the pattern is matched at Step-3. The cursor position is noted and the searching process is terminated.

For pattern matching, each sub-string of S is compared with pattern "CB " character by character. Total number of sub-strings in a string to compare a pattern is computed as:

```
Number of Sub-Strings = String Length - Pattern Length + 1
```

In the above example, length of the string S is 8 and length of the pattern is 2. Thus, total number of sub-strings to be compared with are:

```
Number of Sub-Strings  =  8 - 2 +1
                       =  7
```

Nested loops are used to search a pattern within a string. The outer loop is used to divide the string into sub-strings. The inner loop is used to compare each sub-string of the string with the pattern character by character.

Algorithm – Pattern Matching

STR and SUBSTR are two strings with length **len** and **sublen** respectively. Write an algorithm to search SUBSTR within STR.

```
1.   INPUT String in STR
2.   INPUT Substring in SUBSTR
3.   FIND length of STR in LEN
4.   FIND length of SUBSTR in SUBLEN
5.   COMPUTE TOTAL sub-strings within STR
6.   TOTAL = LEN - SUBLEN + 1
7.   C=0
8.   REPEAT Step-9 TO 10 WHILE C <= TOTAL
9.   REPEAT Step-10 FOR I= 0 TO SUBLEN
10.  IF I=SUBLEN THEN
     PRINT "PATTERN MATCHED AT POSITION = ", C+1
     EXIT
     END IF
     IF SUBSTR[I]!=STR[C+I] THEN
      C = C+1
      GO TO STEP-8
     END IF
     [End of step-9 inner loop]
     [End of step-8 upper loop]
11.  PRINT "PATTERN NOT MATCHED"
12.  EXIT
```

Program

```
#include<iostream.h>
#include<conio.h>
class temp
{
private:
char str[30],substr[30];
int len, sublen,total;
public:

// member function to input string & sub-string
      input()
       {
       cout<<"Enter a string "; cin>>str;
       cout<<"Enter a sub-string "; cin>>substr;
       }

// member function to calculate string & sub-string lengths
      length()
       {
       for(int i=0; str[i]!='\0';i++)
       len=i;
       for(int i=0;substr[i]!='\0';i++)
       sublen=i;
       total=len-sublen+1;
       }

// member function to match pattern
      match()
       {
       int i,c;
       for(c=0;c<=total;c++)
       {
       for(i=0;i<=sublen;i++)
       {
       if(i==sublen)
             {
             cout<<"Pattern matched at position = "<<c+1;
             return 0;
             }
       if(substr[i]!=str[c+i])
             break;
       }
       }
       cout<<"Pattern not matched ";

      }
      };
```

```
main()
{
temp obj;
clrscr();
obj.input();
obj.length();
obj.match();
getch();
}
```

Insertion Operation

The process of adding a sub-string at a specified location in a string is called insertion operation. The procedure that is used to insert a sub-string is usually named as "insert". Its syntax is:

```
insert(string, position, sub-string)
```

where

string specifies the given string into which text is to be inserted.

position specifies the position at which the sub-string is to be inserted.

sub-string specifies the sub-string that is to be inserted into the string.

For example, to insert "is" in string "Paktan" at position 4, the insert procedure is specified as:

```
insert("Paktan", 4, "is")
```

The procedure returns the string as "Pakistan".

In insertion operation, the operations of extracting and concatenation are used. In extracting, the given string into which the sub-string is to be inserted is divided into two parts, i.e:

- First part consists of the sub-string starting from the first character up to the specified position in the given string.

- Second part consists of the remaining part of the string, i.e. from the specified position up to the last character of the string.

The text that is to be inserted is concatenated with the first part of the string. The second part of the original string is concatenated to it.

Algorithm – Insertion

Write an algorithm to insert a substring "**substr**" into the string "**str**" at position "**p**".

```
1.  INPUT string into STR
2.  INPUT substring into SUBSTR
3.  INPUT position to insert SUBSTR into STR in P
    [Extract first part of string in ST1]
4.  ST1 = SUBSTRING[STR, 1, P-1]
    [Extract second part of string in ST2]
5.  ST2 = SUBSTRING(STR, P, LENGTH(STR) - P+1)
    [Combine the strings, ST1, SUBSTR and ST2]
6.  RESULT = ST1 + SUBSTR + ST2
7.  PRINT RESULT
8.  EXIT
```

Program

```
#include<iostream.h>
#include<conio.h>
class temp
{
private:
char str[75], st1[25], st2[25], substr[25];
int p, len;
public:

// member function to input strings
      input()
       {
```

```
            cout<<"Enter string ? "; cin>>str;
            cout<<"Enter sub-string to insert ? "; cin>>substr;
            cout<<"Enter position in string to insert?"; cin>>p;
        }

// member function to find string length
        length()
        {
        for(int i=0;str[i]!='\0';i++)
        len=i;
        }

// member function to divide string into two substrings
extract()
        {
        for(int i=0;i<=p-1;i++)
        {
        st1[i]=str[i]; // copy string from position 1 to p into st1
        st1[i+1]='\0';
        }
        for(int i=p; i<=len; i++)
        {
        st2[i-p]=str[i]; // copy p to end to st2 from str
        st2[i-p+1]='\0';
        }
}
// member function to combine three strings
insert()
{
int cc, ccc;
for(int i=0, c=0; st1[i]!='\0'; i++, c++)
{
      str[c]=st1[i]; // copy st1 to str
      cc = c + 1;
}

for(int i=0, c=cc; substr[i]!='\0'; i++, c++)
{
      str[c]=substr[i]; // copy substr to str
      ccc = c + 1;
}

      for(int i=0, c=ccc; st2[i]!='\0';i++,c++)
      {
      str[c]=st2[i]; // copy st2 to str
      str[c+1]='\0';
}
      cout<<" Resultant string = "<<str;
}
};
```

```
main()
{
temp obj;
clrscr();
obj.input();
obj.length();
obj.extract();
obj.insert();
getch();
}
```

Deletion Operation

The process of removing a substring from a specified position in a given string is called deletion. The function used to remove a substring from a string is usually named "delete". Its syntax is:

```
delete(string, position, size)
```

where

string	specifies the string from which the part of the text is to be deleted.
position	specifies the starting position of sub-string within the string from which sub-string is to be removed.
size	specifies the size of the sub-string to be removed.

For example, if S = "ABCXYZ", then to delete the sub-string "CX" from the string S, the starting position is 3 and length is 2. The procedure is written as:

```
delete(S, 3, 2)
```

The delete operation is also similar to the insertion operation. The only difference is that the extracted sub-string is omitted when the strings are concatenated.

Program

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

```
{
private:
char str[100], st1[50], st2[50];
int loc, len, s;
public:

// member function to input data
        input()
         {
         cout<<"Enter string ? "; cin>>str;
         cout<<"Enter location to delete sub-string ? "; cin>>loc;
         cout<<"Enter length of substring to delete ? "; cin>>s;
         }

// member function to find string length
        length()
         {
         for(int i=0; str[i]!='\0'; i++)
         len=i;
         }

// member function to extract string
        extract()
         {
         int j;
         for(int i=0; i<=loc-1; i++)
        {
         st1[i]=str[i]; // copy string from 1 to loc to st1
         st1[i+1]='\0';
         j = i+1;
         }

         for(int i=j; str[i]!='\0'; i++)
         {
         st2[i]=str[loc+i]; // copy remaining part to st2
         st2[i+1]='\0';
         }
         }

// member function to delete substring
        del()
         {
         int cc;
         for(int i=0, c=0; st1[i]!='\0'; i++, c++)
{
         str[c]=st1[i]; // copy st1 to str
         str[c+1]='\0';
         cc = c+1;
         }
```

```
        for(int i=s, c = cc; st2[i]!='\0';i++, c++)
        {
        str[c]=st2[i]; // copy st2 to str
        str[c+1]='\0';
        }
        cout<<"Resultant string = "<<str;
}
};
main()
{
temp obj;
clrscr();
obj.input();
obj.length();
obj.extract();
obj.del();
getch();
}
```

Replacement Operation

The process of finding a sub-string within a string and replacing it with new sub-string is known as replacement operation. The procedure used to replace sub-string is usually named as "replace". Its general syntax is:

```
replace(string, substring1, substring2)
```

where

string	specifies the string in which the sub-string is to be searched and replaced.
substring1	specifies the sub-string that is to be searched in the string.
substring2	specifies new value or text that is to be replaced with the substring1 in the string.

For example, if S = "XYZABC" then to replace "AB" with "XXYY" in S, the Replace function is written as:

```
Replace (S, "AB", "XXYY")
```

The contents of S after replacement will be "XYZXXYYC". The replacement operation consists of:

- Indexing operation to search substring1.
- Deletion operation to delete the substring1 from the given string.
- Insertion operation to insert the new value in place of substring1.

Suppose if S = "Programing" then to replace "mm" in place of "m" the following steps have to be performed.

1. `C = index (S, "m")`
   Search "m" within string S. Return its position and store the position into C. The value 7 will be stored in C.

2. `S = Delete (S, C, length ("m"))`
   Delete "m" from S. The resulting string is stored in the string variable S, i.e. S = "prograing".

3. `Insert (string, C, "mm")`
   Insert sub-string "mm" at position C, i.e. 7 in string S. The contents of string after insertion are S = "Programming".

Algorithm – Replacing

Write an algorithm to input three strings into variables STR, SUBSTR1, and SUBSTR2. Search the substring SUBSTR1 in string STR and, if found, replace it with new value SUBSTR2.

```
1.  Input string in STR
2.  Input substrings in SUBSTR1 and SUBSTR2
    [Search substr1 in str]
3.  C = index(STR, SUBSTR1)
4.  IF C > 0 THEN
        STR = DELETE(STR, C, length(SUBSTR1))
        INSERT (STR, C, SUBSTR2)
    ELSE
        PRINT "SUBSTRING NOT FOUND"
    [End of IF Structure]
5.  Exit
```

Program

```
#include<iostream.h>
#include<conio.h>
class temp
 {
 private:
char str[50], st1[25], st2[25], substr1[25], substr2[25];
int loc, len,s, sublen, total;
 public:

// member function to input strings
        input()
        {
        cout<<"Enter string ? "; cin>>str;
        cout<<"Enter sub-string to search ? "; cin>>substr1;
        cout<<"Enter sub-string to replace ? "; cin>>substr2;
        }

// member function to find string length
        length()
        {
        for(int i=0; str[i]!='\0'; i++)
        len=i;

        for(int i=0; substr1[i]!='\0'; i++)
        sublen=i;
        total=len-sublen+1;
        }

// member function to match strings
        match()
        {
        int i,c;
        for(c=0;c<=total;c++)
        {
        for(i=0;i<=sublen;i++)
        {
        if(i==sublen)
        {
        loc = c; // get the location of substring
        return 0;
        }
        if(substr1[i]!=str[c+i])
        break;
        }
        }
cout<<"Pattern not matched ";
return 0;
```

```
}

// member function to extract strings
        extract()
        {
        for(int i=0;i<=loc-1;i++)
        {
        st1[i]=str[i]; // copy 1 to loc to st1 from str
        st1[i]='\0';
        sublen=sublen+i;
}
        for(int j=sublen;str[j]!='\0';j++)
        {
        st2[j-sublen]=str[j]; //copy remaining to st2
        st2[j-sublen]='\0';
}
}

// member function to replace text
        replace()
        {
  int i, c;
        for(i=0,c=0;st1[i]!='\0';i++)
        str[c]=st1[i]; // copy st1 to str

        for(i=0;substr2[i]!='\0';i++,c++)
        str[c]=substr2[i]; // copy substr1 to str

        for(i=0;st2[i]!='\0';i++,c++)
        str[c]=st2[i]; // copy st2 to str
        str[c]='\0';
        cout<<"Resultant string = "<<str;
        }
};

main()
{
temp obj;
clrscr();
obj.input();
obj.length();
obj.match();
obj.extract();
obj.replace();
getch();
}
```

EXERCISES

Q.1 Fill in the blanks.

1. A sequence of symbols is called _____.
2. The non-numerical data is also called _____.
3. Total number of symbols in a string specifies _____ of the string.
4. End of a string is indicated by a special character called _____ character.
5. The string with zero length is also known as _____.
6. In C++, the null character in a string is represented by _____.
7. The string in which the size of string variable remains fixed during program execution is called _____ string.
8. The string in which the size of string variable can be changed during program execution is called _____ string.
9. The process of combining the contents of two or more strings into a single string is called _____.
10. The process of copying the contents of one string into another is called _____.
11. The process of taking out a part of string is called _____.
12. The process of finding a sub-string within a string is called _____.
13. The process of adding a sub-string at a specified location in a string is called _____.

Q.2 Mark True or False

1. The string with zero length is called null string.
2. The string with zero length is also called empty string.
3. When the length of string is computed, the null character at the end of the string is also counted.

4. The process of combining contents of two or more strings into a single string is called combination operation.
5. The process of copying the contents of one string into another is known as string copying.
6. A sequence of symbols is called array.
7. The non-numerical data is also called special data.
8. In C++, the null character is indicated by '\0'.
9. The process of finding a sub-string within a string is called string searching.
10. The strings can only be represented inside memory as linked lists.

Q.3 Write an algorithm to count the total number of alphabets and numeric digit (if any) in a string.

Q.4 Explain different methods of representing strings in the memory.

Q.5 What is a sub-string? Give examples.

Q.6 Write a program to copy one string into another.

Q.7 Write a program to convert a string from lower case to uppercase characters.

Q.8 Write a program to count the total number of capital letters in a string.

4 STACKS

STACKS

A stack is a special kind of linear list in which only two operations, insertion and deletion, can be performed. These operations may occur only at its top end. The items in it are stored and retrieved in **Last In First Out (LIFO)** manner.

A common example of a stack is a dish rack. A dish rack is a spring-loaded device that holds dishes in such a manner that only the top dish is visible. It is used such that:

- A clean dish is placed on the top of the stack. This forces the spring down and only the new dish is available.
- When a clean dish is needed, the top one is removed. This causes the spring to release so that the second dish becomes available.
- The last dish cleaned is the first one used.

The stacks have very restricted use. However, they are very efficient and easier to implement. They are used for programming simple card games and maintaining the order of operations in complex programs. They are also used in management programs where the newest tasks must be executed first.

Representation of Stacks

A stack is usually represented by a linear array in the computer memory. The following figure shows a stack represented as a linear array in the memory. Four items are stored in the above stack. It can store a maximum of 8 items. The position of the top item is four.

The most accessible item in the stack is called the **Top** of the stack and the least accessible item is called **Bottom** of the stack.

Stack Operations

The only two operations that can be performed on a stack are insertion and deletion.

The process of inserting or adding new items into stack is called **Pushing**. Before pushing an item into a stack, it is tested whether or not there is any room in the stack to store the item. If there is no room in the stack, then the stack condition is known as overflow.

Similarly, the process of removing or deleting items from a stack is called **Popping**. Before removing an item from a stack it is tested whether or not at least one element exists in the stack. If no element exists in a stack, i.e. the stack is empty, the condition is known as underflow.

Push Procedure — Push (S, X)

Write an algorithm for Push procedure to add an item 'X' into a stack 'S'.

Suppose the stack has N elements and pointer Top represents the top element in the stack. The following algorithm shows the Push procedure used to add an item into the stack.

```
1.    [Check for Overflow Condition]
      IF Top >= N Then
      PRINT "Stack Overflow"
      RETURN
      [End of IF Structure]
2.    [Increment 1 Into Top]
```

```
        TOP = TOP + 1
3.      [Add Element X Into Stack "S"]
            S[TOP] = X
        [End Of Procedure]
4.      RETURN
```

Procedure Pop — Pop (S, Top)

Write an algorithm to pop the top element from stack S.

```
1.      [Check For Underflow Condition]
        IF Top = 0 Then
        PRINT "Stack Underflow"
        RETURN
        [End of IF Structure]
2.      [Decrement Pointer to Delete The Top Item]
        Top = Top - 1
        [End Of Procedure]
3.      RETURN
```

Program

```
// program to push and pop items in a stack
#include <iostream.h>
#include <conio.h>
class stack
   {
    private:
    int top;
    int S[10];
    public:

// constructor to initialize stack
    stack()
        {
         top=-1;
```

```
// member function to push item into stack
    push(int n)
        {
        if(top==9)
         {
         cout<<"Stack overflow";
         getch();
         return 0;
         }
         top++;
         S[top]=n;
        }

// member function to delete item from stack
    int pop()
        {
        int data;
        if(top==-1)
         {
         cout<<"Stack is empty";
         getch();
         return NULL;
         }
         data=S[top];
         top--;
         return data;
        }

// member function to print items of stack
        print()
         {
         if(top==-1)
          {
          cout<<"Stack is empty";
          return NULL;
          }
          for(int i=top; i>=0; i--)
           cout<<S[i]<<endl;
           getch();
         }
 };

main()
{
stack obj;
int opt, val;
        while(opt!=3)
        {
```

```
        clrscr();
        cout<<"1: Push\n";
        cout<<"2: Pop\n";
        cout<<"3: Exit\n";
        cout<<"Enter the choice:";
        cin>>opt;

        switch(opt)
        {
        case 1:
            cout<<"Enter Value to insert ? ";
            cin>>val;
            obj.push(val);
            cout<<"\nStack after insertion\n";
            obj.print();
            break;
        case 2:
            cout<<"\nValue "<<obj.pop()<<" is popped\n";
            cout<<"Stack after deletion\n";
            obj.print();
            break;
        }
    }
}
```

RECURSION

A procedure that calls itself, either directly or indirectly, is called **recursive procedure** and the process is called **recursion**. The process in which a procedure calls itself is called **direct recursion**. Similarly, the process in which a procedure A calls another procedure B, which in turn calls the procedure A is called **indirect recursion**. The number of times a function is called recursively is called depth of recursion.

Recursion is a fundamental concept in computer science. The feature of calling a function from itself was not supported in older programming languages like Fortran and Basic. But almost all modern programming languages support recursion.

The recursion is used because it is easier to define algorithms recursively. However, recursive algorithms do not always provide the most efficient solution. In fact, all recursive algorithms can be converted into equivalent non-recursive algorithms and often non-recursive algorithms are preferred because of their efficiency.

Conditions for Recursive Procedures

There are two important conditions that must be satisfied by any recursive procedure. These conditions are required to prevent the recursive procedure from running indefinitely. These conditions are that:

- There must be a criteria for exiting from the recursive procedure. The condition at which the control exits from the procedure is known as the Base Criteria.
- Each time the procedure is called directly or indirectly, the condition must get closer to the Base Criteria.

A recursive function that fulfills these two conditions is called a well-defined function.

Implementation of Recursive Procedure by Stacks

A recursive procedure can be executed several times. It receives values from the calling procedure and transmits results back to it. For its proper functioning, it must save the parameters and variables upon execution and restore these parameters and variables at completion. While returning values, the procedure must keep track of the return address in the calling procedure. This return address is used to transfer the control back to its proper place in the calling procedure. It saves the return address of each calling procedure in the same order in which it is called and returns the control to proper place in the calling procedure each time the control is returned.

The recursive function terminates execution when the base criteria is reached. It then returns parameters and variables to the calling procedures. It returns these values such that the values from the last execution are returned first. This last-in and first-out characteristics of a recursive procedure shows that a stack is the most suitable data structure to implement it. At each procedure call, the stack is pushed to save the necessary values. Upon exit from the procedure, the stack is popped to retrieve the values.

The general algorithm for a recursive procedure contains the following three steps:

Prologue — It is the opening part of the recursive procedure. Its purpose is to save the formal parameters, local variables and return address.

Body — It contains the definition of the procedure, including the test criteria, in which the procedure calls itself. Each time the procedure calls itself, the prologue of the procedure saves all necessary information required for its functioning.

Epilogue — It is the last part of the recursive procedure. It restores the most recently saved parameters, local variables and return addresses.

Example: Describe steps to calculate the factorial of 3 using recursive definition.

```
If n = 0 then n! = 1
If n > 0 then n! = n.(n-1)
    Step 1        3! = 3.2!
    Step 2           2! = 2.1!
    Step 3              1! = 1.0!
    Step 4                 0! = 1
    Step 5              1! = 1.1 = 1
    Step 6           2! = 2.1 = 2
    Step 7        3! = 3.2 = 6
```

- From steps 1 to 3, the factorial is evaluated in terms of factorial of another integer.
- Step-4 explicitly evaluates factorial of 0! as '0!' is base value of the recursive definition.
- After reaching the base criteria, the evaluation is performed in the reverse order.
- The factorial of 0 is used to find out the factorial of 1. Similarly, 1! is used to find out 2!, and so on.

Algorithm — Factorial

Write the algorithm to find factorial of integer N using loop process.

```
1.    IF N = 0 Then
         Factorial = 1
         PRINT "Factorial = ", Factorial
         RETURN
      End IF

2.    Factorial = 1
      REPEAT FOR C = 1 To N By 1
         Factorial = Factorial * C
      [End Of Loop]

3.    PRINT "Factorial =", Factorial
4.    EXIT
```

Program

```
// factorial of given number using loop

#include <iostream.h>
#include <conio.h>
class factorial
   {
   public:

// member function to compute factorial
    int fact(int n)
        {
        int f=1;
        if(n ==0)
          return f;
        else
          for(;n>1;n--)
          f = f * n;
          return f;
        }
};
```

```
main()
{
factorial obj;
int val, res;
clrscr();
cout<<"Enter Value to find factorial ? ";
cin>>val;
res = obj.fact(val);
cout<<"Factorial of "<<val<<" is = "<<res;
getch();
}
```

Procedure — Factorial (F, N)

Write an algorithm to find factorial of integer N using recursive procedure.

```
      Factorial (F, N)
1.    IF N = 0 Then
        F = 1
        RETURN
      End IF
2.    Call Factorial (F, N-1)
      F = F * N
      Return
```

In the above algorithm, the procedure Factorial calls itself to calculate the factorial of the given integer N. It returns the value in variable F.

Program

```
// factorial of given number using recursion

#include <iostream.h>
#include <conio.h>
class factorial
   {
    public:

// member function to compute factorial
     int fact(int n)
        {
```

```
        if(n ==0)
          return 1;
        else
          return n*fact(n-1);
       }
 };
main()
{
factorial obj;
int val, res;
clrscr();
cout<<"Enter Value to find factorial ? ";
cin>>val;
res = obj.fact(val);
cout<<"Factorial of "<<val<<" is = "<<res;
getch();
}
```

Program

```
// program to find exponential of an integer using recursion
#include <iostream.h>
#include <conio.h>
class exponent
    {
     public:
// member function to find exponential
     int power(int b, int n)
          {
          if(n == 0)
            return 1;
          else
            return b*power(b, n-1);
          }
 };
main()
{
exponent obj;
int val, p, res;
clrscr();
cout<<"Enter Value ? ";
cin>>val;
cout<<"Enter power ? ";
cin>>p;
res = obj.power(val,p);
cout<<"Result is = "<<res;
getch();
}
```

Evaluation of Expressions

An arithmetic expression is made up of operands, arithmetic operators and parentheses. The operands may be numeric variables or numeric constants. Following are some examples of arithmetic expressions:

```
A+B, X*Y, A*(B-C)/2, (A+B)^2
```

The arithmetic operators +, –, * and / are the same that are used in ordinary algebra. The operator ^ is used as to compute the exponential. This operator is not available in C++. Instead 'pow' function is used to calculate the exponential. The expression is always evaluated from left to right. The order in which the expression is evaluated is:

- If the expression has parenthesis, then they are evaluated first.
- Exponential (^) is given highest priority.
- Multiplication (*) and division (/) have the next highest priority.
- Addition (+) and subtraction (–) have the lowest priority.

Example: Describe the steps to evaluate the following expression.

```
2^3 + 6*2 - 9/3
```

Step-1: The expression is evaluated from left to right. Exponential is evaluated first. After evaluating the exponential, the expression becomes:

```
= 8 + 6*2 - 9/3
```

Step-2: Multiplication and division are performed next and the expression becomes:

```
= 8 + 12 - 3
```

Step-3: Addition and subtraction are performed last, from left to right, and the final result is:

```
= 17
```

Polish Notation

In most arithmetic expressions, the arithmetic operator is placed between two operands, e.g. `x+y`, `x/y`. This type of notation is called **infix notation**. In polish notation, however, the arithmetic operator is placed before its two operands. Following are some examples of polish notation.

	Infix notation	Polish notation
i)	`X+Y`	`+XY`
ii)	`X/Y`	`/XY`
iii)	`(X+Y)*Z`	`*+XYZ`
iv)	`X+(Y*Z)`	`+X*YZ`

Since the operators are used before the operands in polish notation, it is also called **prefix notation**. The polish notation is named in honour of Polish mathematician Jan Lukasiewiez. Parentheses are not used in polish notation.

Reverse Polish Notation or Postfix Notation

In this notation the arithmetic operator is placed after the operands. This notation is also known as **suffix notation**. The following table shows expressions in infix notation and equivalent prefix and postfix notations.

Infix	Prefix (Polish notation)	Postfix (Reverse Polish notation)
`A+B`	`+AB`	`AB*`
`A*B`	`*AB`	`AB*`
`A/B`	`/AB`	`AB/`
`A-B`	`-AB`	`AB-`

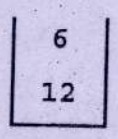
The computer evaluates an expression given in infix notation by converting it into postfix notation. The stack is used to perform this operation. The following steps are taken to evaluate a postfix expression:

1. The expression is scanned from left to right until the end of the expression.
2. When an operand is encountered, it is pushed into stack.
3. When an operator is encountered, then:
   - The top two operands of stack are removed.
   - The arithmetic operation is performed.
   - The computed result is pushed back to the stack.
4. When end of the expression is reached, the top value from the stack is picked. It is the computed value of the expression.

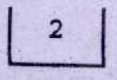
Example Evaluate expression `AB/C-DE*+` when `A=12, B=6, C=3, D=4 & E=5`.

The total number of items in the expression "AB/C – DE*+" are 9. So the loop will be executed 9 times.

1. In first and second iteration, the values of A and B are pushed into the stack as shown below:

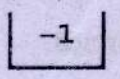


2. In third iteration, the operator '/' is encountered. So the two values 6 and 12 are removed from stack and division is performed, i.e. 12/6=2. The computed value is pushed back to the stack. The values in the stack will be:

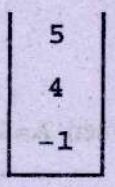


3. In fourth iteration, value of C is pushed into stack as shown below:

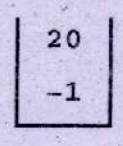
4. In fifth iteration, operator '–' is encountered. The two top values from the stack are popped and arithmetic operation is performed, i.e. 2-3 = –1. The result is pushed back to stack.



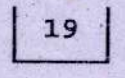
5. In sixth and seventh iterations, values of D and E are pushed into the stack as shown below:



6. In eighth iteration, multiplication operator * is encountered, Thus 4 & 5 are popped. Multiplication operation is performed and calculated result 20 is pushed into the stack. The stack will be as shown below:



7. In ninth repetition the operator '+' is encountered. Two values 20 and –1 are popped. The addition of these numbers is performed and the result 19 is pushed back to stack.



8. The top value in stack is 19. It is the final value after evaluating the expression:

```
12/6 -3 + 4 * 5 =
                = 2 - 3 + 20
                = -1 + 20
                = 19
```

Infix to Postfix Conversion

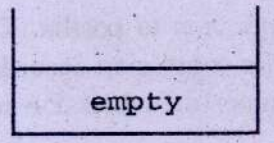
The stack is used to convert an infix expression to postfix. The stack is used to store operands and then pass to the postfix expression according to their precedence. The infix expression is converted into postfix expression according to the following rules:

1. The infix expression is scanned from left to right until end of the expression. The operands are passed directly to the output. Whereas, the operators are first passed to the stacks.
2. Whenever an operand is encountered, it is added to the output, i.e. to the postfix expression.
3. Each time an operator is read, the stack is repeatedly popped and operands are passed to the output, until an operator is reached that has a lower precedence than the most recently read operator. The most recently read operator is then pushed onto the stack.
4. When end of the infix expression is reached, all operators remaining in the stack are popped and passed to the output in the same sequence.
5. Parentheses can be used in the infix expression but these are not used in the postfix expression. During conversion process, parentheses are treated as operators that have higher precedence than any other operator. The left parenthesis is pushed into the stack when encountered.
6. The right parenthesis is never pushed to the stack. The left parenthesis is popped only when right parenthesis is encountered. The parentheses are not passed to the output postfix expressions; they are discarded.
7. When end of expression is reached, then all operators from stack are popped and added to the output.

Example Convert the following expression in infix notation into postfix notation using stack:

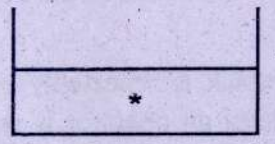
```
x * (y + z) + a/b
```

1. The infix expression is scanned from left to right. The first operand read is x and it is passed to the output.



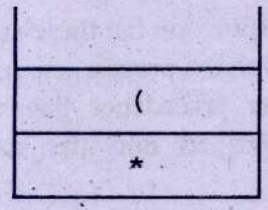
Output: x

2. Next the '*' operator is read. At this stage, the stack is empty. Therefore no operators are popped and '*' is pushed onto the stack. Thus the stack and the output will be:



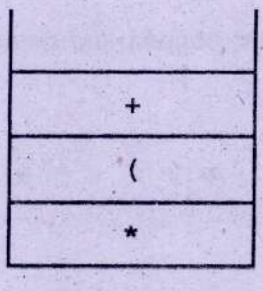
Output: x

3. The left parenthesis '(' is read. Since all operators have lower precedence than the left parenthesis, it is pushed onto the stack. Then the operand 'y' is read and is passed to the output. Thus the stack and the output will be:



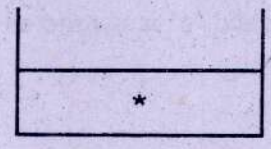
Output: x y

4. The next symbol read is the '+' operator. Now, left parenthesis '(' has higher precedence than '+'; it cannot be popped from the stack until a right parenthesis has been read. Thus the stack in not popped and the '+' operator is pushed into the stack. Next the operand 'z' is read and passed to the output. Thus the stack and the output will be:



Output: x y z

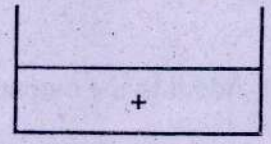
5. Now, the right parenthesis is read. All operators upto left parenthesis are popped and passed to the output. The left parenthesis is also popped and discarded. Thus the stack and the output will be:



Output: x y z +

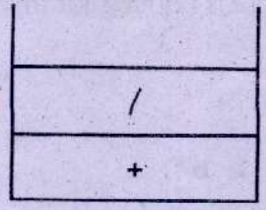
6. Next symbol read is the '+' operator. The stack has '*' operator which has higher precedence than the '+' operator. The stack is popped and the '*' operator is passed to the output and '+' is pushed into the stack.

   Next the operand 'a' is read and passed to the output. Thus the stack and the output will be:



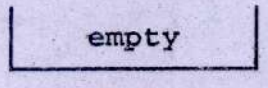
Output: x y z + * a

7. Now, symbol read is '/' operator. The '+' operator in the stack has lower precedence, therefore, it is not popped and the recently read symbol '/' is pushed into the stack. The final symbol read is the operand 'b' and it is passed to the output. Thus the stack and the output will be:



Output: x y z + * a b

8. The remaining operators stored in the stack are popped and passed to the output. The stack and the final output will be:

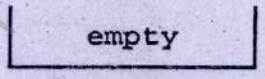


Output: x y z + * a b / +

Example Convert a+b*c into postfix using the stack method.

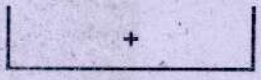
Follow these steps to convert the above expression into postfix notation. It is noted that output shows the output result for postfix.

1. Scan expression from left to right. The first operand 'a' is countered. So it is not pushed to stack. It is only added to output.



Output: a

2. The next item of expression is operator (+). Push it into stack but it is not added to output.



Output: a

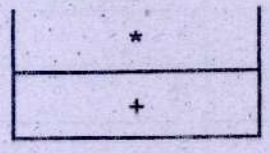
3. The third item of the expression is an operand (b). Add it to the output but not pushed to stack.

Output: a b

4. The fourth item is operator (*). This operator has higher priority than the top operator in the stack. So push it into stack but output is nothing for this step.

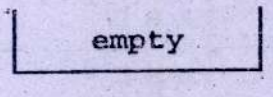
Output: a b

5. The fifth item is an operand (c). Add it to output.



Output: a b c

6. The end of expression is encountered. The operators are removed from the stack and added to the output in the same sequence in which these are popped.



Output: a b c * +

Example Convert the following infix expressions into postfix expressions:

1. (A+B)*C
2. A+(B*C)
3. A+B*C+(D*E+F)*G

(i) (A+B)*C

Steps	Symbol Scanned	Stack	Output
1	(	(	Empty
2	A	(	A
3	+	(+	A
4	B	(+	A B
5	)	Empty	A B +
6	*	*	A B +
7	C	*	A B + C
8	End	Empty	A B + C *

(ii) A+(B*C)

Steps	Symbol Scanned	Stack	Output
1	A	Empty	A
2	+	+	A
3	(	+(	A
4	B	+(	A B
5	*	+(*	A B
6	C	+(*	A B C
7	)	+	A B C *
8	End	Empty	A B C * +

(iii) A+B*C + (D*E+F) * G

Steps	Symbol Scanned	Stack	Output
1	A	Empty	A
2	+	+	A
3	B	+	A B
4	*	+*	A B
5	C	+*	A B C
6	+	+	A B C * +
7	(	+(	A B C * +
8	D	+(	A B C * + D
9	*	+(*	A B C * + D
10	E	+(*	A B C * + D E
11	+	+(+	A B C * + D E *
12	F	+(+	A B C * + D E * F
13	)	+	A B C * + D E * F +

14	*	+*	A B C * + D E * F +
15	G	+*	A B C * + D E * F + G
16	End	Empty	A B C * + D E * F + G * +

Program

```
// Program to convert the infix expression into postfix expression

#include <iostream.h>
#include <conio.h>
class exp
{
  private:
        char st[100];
        char stack[15], ch;
        int top;
  public:
    exp() { top=-1;}
    input()
       {
        cout<<"Enter Expression without space ? ";
        cin>>st;
       }

     void scan(void);
     void push(void);
     void pop(void);
};

main()
{
clrscr();
exp s1;
s1.input();
s1.scan();
getch();
}

// member function to read an expression
void exp::scan(void)
{
  for(int i=0;st[i]!='\0';i++)
   {
    ch=st[i];
```

```
        if( ch=='('|| ch=='*'|| ch=='/'|| ch=='+'
                                || ch=='-'||ch==')')
        {
            if(ch=='(')
                    push();
            else if(stack[top]=='(')
                    push();
            else if(ch==')')
                    pop();
            else if((stack[top]=='+' ||stack[top]=='-' )
                                    && (ch=='*'||ch =='/'))
                    push();
            else if(top==-1)
                    push();
            else
                {   pop(); push();}
        }
        else
            cout<<ch<<" ";
    }
    pop();
}

// member function to push operators into stack
void exp::push(void)
    {
     top++; stack[top]=ch;
    }

// member function to remove operators from stack
void exp::pop(void)
{
 while(top!=-1)
  {
    if(ch==')')
       {
        cout<<stack[top]<<" ";
        top--;
        if(stack[top]=='(' )
         {top--; break;}
        }
      else
       {
        cout<<stack[top]<<" ";
        top--;
        if(stack[top]=='(') break;
       }
  }
}
```

EXERCISES

Q.1 Fill in the blanks

1. _____ is a data structure in which only two operations, insertion & deletion can be performed.
2. LIFO stands for _____.
3. A common example of stacks is _____.
4. The insertion operation in stacks is called _____.
5. The deletion operation in stacks is called _____.
6. The most accessible item in a stack is called the _____ of stack and the least accessible item is called _____ of the stack.
7. A stack is represented inside the computer as _____.
8. A procedure that calls itself, either directly or indirectly, is called _____ procedure and this process is called _____.
9. The process in which a procedure calls itself is called _____.
10. The criteria used in a recursive function is called _____.
11. The number of times a function is called recursively is called _____ of recursion.
12. The _____ is used to implement the recursion.
13. The arithmetic notation, in which the arithmetic operator is placed between two operands is called _____.
14. The Polish notation, in which the arithmetic operator is placed before its two operands is called _____.
15. The Polish notation, in which the arithmetic operator is placed after its two operands is called _____.
16. The postfix notation is also referred as _____.
17. The stack is used to convert an infix expression to _____ notation.

Q.2 Mark True or False

1. The stack is used to convert infix expression to postfix notation.
2. The infix expression is scanned from right to left to convert it to postfix notation.
3. The right parenthesis is also pushed to stack.
4. The prefix notation is also referred as infix notation.
5. The postfix notation is also referred as suffix notation.
6. The Polish notation is named in honour of Polish mathematician Jan Lukasiewiez.
7. In C++, operator ^ is used to compute exponential of an integer.
8. The insertion operation performed in stack is referred as popping.
9. The deletion operation performed in stack is referred as pushing.
10. The items in a stack are stored and retrieved in Last In First Out.
11. If no element exists in a stack, the condition is known as overflow.
12. If there is no room in a stack, the condition is known as underflow.
13. LIFO stands for Last In First Out.
14. A stack is represented inside the computer by a linked list.
15. The stack is used to compute the factorial of a given number.
16. The criteria used in recursive function is known as standard criteria.
17. The number of times a function is called recursively is called height of recursion.

Q.3 What is a stack? Briefly describe operation of a stack.

Q.4 What is pushing & popping in stack? Explain with examples.

Q.5 What is recursion? Also write a program in C++ using a recursive function to print first 10 odd numbers.

Q.6 Convert the following infix expressions into postfix notations.

1. `A+2AB+B^2`
2. `X+6*(Y+Z)^3`

Q.7 Write a program to input an infix expression into a string variable. Convert the expression into postfix notation by using a stack.

Q.8 Write a program to input and push 10 values into a stack. Pop the values from the stack and print only even values on the screen.

Q.9 Write a program to input and push 5 values into a stack. Pop values from the stack one by one and calculate factorial of each using a recursive function and print the results on the screen.

5 QUEUES

QUEUES

A queue is a special type of linear structure. It holds a series of items for processing on a first in first out basis. Elements can only be inserted to the back of a queue, and only the front element can be accessed and modified. The end at which the elements are inserted is called the Rear and the end at which the elements can be deleted is called Front of the queue.

The structure of a queue is the same as that of a line of people. A person who wishes to stand in line must go to the back, and the person in front of the line is served. Thus, a queue is a "First In, First Out" (FIFO) structure.

The purpose of a queue is to provide some form of buffering. In computer systems, queues are used for:

- **process management:** For example, in a timesharing system in a computer, programs are added to a queue and are executed one after the other.
- **buffer between the fast computer and a slow printer:** Documents sent to the printer for printing are added to a queue. The document sent first is printed first and document sent last is printed last.

The above figure shows a queue with 5 elements. The element 1 is the first (or front) element and element 5 is the rear (or last) element. A new element can be added after 5. Similarly, the elements can be deleted starting at the front

of the queue. For example, the first element to be deleted will be ❶ and to delete the element ❺, all elements in front of ❺, i.e. ❶ ❷ ❸ and ❹ must be deleted first.

Representation of Queues

Queues may be represented in computer in various ways. The best and the easiest way to represent a queue is by using a linear array.

Suppose a linear array 'Q' represents a queue and pointer variables F and R represent the addresses of Front and Rear of the queue. If the array has N elements, then whenever a new item is added, the value of R is increased by 1, i.e.

```
R = R+1
```

Similarly, when an item is deleted from the queue, the value of F is increased by 1, i.e.

```
F = F+1
```

Algorithm — insertion...

Algorithm — Insertion

Write an algorithm to add an item "Item" in a queue 'Q'.

```
1.    IF R >= N THEN
         PRINT "Overflow "
         RETURN
      ELSE
         R = R+1
      [End of If Structure]
2.    [Insert item]
         Q[R] = "Item"
      IF F = -1 THEN
         F = 0
      END IF
```

Algorithm — Deletion

Write an algorithm to delete an item "Item" from a queue 'Q'.

```
1.    IF F = 0 THEN
         PRINT "Underflow"
      Return
      [End of If Structure]
2.    [Delete item]
      Del = Q[F]
      IF F = R THEN
         F = R = -1
      ELSE
          F = F+1
      END IF
```

In the above algorithm, "Del" is a temporary variable and has Null value.

Program

```
// Program to insert and/or delete items from a queue

#include <iostream.h>
#include <conio.h>
class que
   {
    private:
    int F, R;
    int QA[10];
    public:
    que() // constructor to initilize F & R
        {
         F = -1; R = -1;
        }

// member function to inset an item into queue
    insert(int n)
        {
        if(R>=9)
```

```
            {
            cout<<"Queue is full";
            getch();
            return 0;
            }
           else
            {
           R++;
           QA[R]=n;
           if(F==-1) F=0;
          }
      }

// member function to delete an item from queue
    int del()
          {
          int data;
          if(F==-1)
           {
           cout<<"Queue is empty";
           getch();
           return NULL;
           }
          data=QA[F];
          if(F == R) F = R = -1;
          else F++;
          return data;
         }

// member function to print data in queue
        print()
         {
         if(F==-1)
          {
          cout<<"Queue is empty";
          return NULL;
          }
         for(int i=F;i<=R;i++)
           cout<<QA[i]<<"\t";
           getch();
        }
 };

main()
{
que obj;
int opt, val;
        while(opt!=3)
        {
        clrscr();
```

```
            cout<<"1: Insert Item\n";
            cout<<"2: Delete Item\n";
            cout<<"3: Exit\n";
            cout<<"Enter the choice:";
            cin>>opt;

            switch(opt)
            {
            case 1:
                cout<<"Enter Value to insert ? ";
                cin>>val;
                obj.insert(val);
                cout<<"\nQueue after insertion\n";
                obj.print();
                break;
            case 2:
                cout<<"\nValue "<<obj.del()<<" is deleted\n";
                cout<<"\nQueue after deletion\n";
                obj.print();
            }
        }
    }
```

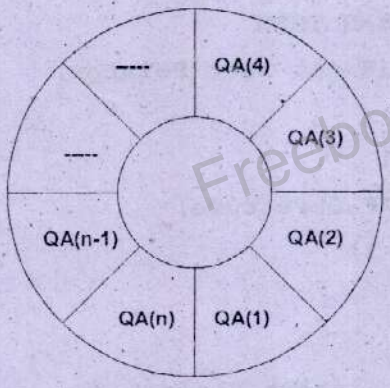
In the above algorithms, a large amount of memory is required to store and process the queue elements. For example, consider a queue 'Q' that can hold a maximum of four elements. Initially the queue contains three elements A, B and C as shown below:

When elements A and B are deleted, the queue will be as shown below:

In the above queue, the values of R and F are same. Only one element can be added into the queue in this state. For example, when an element D is added, the queue will be as shown below:

In this state, the value of F is 'C' and the value of R is 'D'. If an item E is to be added, it would have to be added after D. Although there are still two empty spaces in the queue, but adding another element into the queue would result in overflow.

When elements are deleted from the front, their spaces cannot be used to store new elements. To overcome this problem, a more suitable method is used by arranging the elements of queue in a circular fashion as shown in the figure below:



Circular Queue

By arranging the queue in the circular form, new elements can be inserted or deleted starting at any location by resetting the values of R and F.

Procedure — QINSERT

The procedure used to insert a value into a queue is usually called "QINSERT". This procedure inserts an element X into queue 'Q' having N elements. F and R represent the pointers to the Front and Rear elements of the queue.

The algorithm of the procedure is given below:

```
1.    [Check overflow]
      IF F = 1 and R = N THEN
         PRINT "Overflow"
         RETURN
      [End of If Structure]

2.    [Check for empty Space in Queue]
      IF F=NULL THEN [Queue Is Empty]
        F = 1
        R = 1
      ELSE IF R=N THEN
        R = 1 [Reset Rear Pointer]
      ELSE
        R = R+1
      [End of IF Structure]
      IF (F = -1)
        F = 0
      END IF
      Q[R] = X [Insert Element X]

3.    RETURN
```

Procedure — QDEL

The procedure used to delete a value from a queue is usually called "QDEL". This procedure deletes an element X from queue 'Q' having N elements. F and R represent the pointers to the Front and Rear elements of the queue. The value of the deleted element is assigned it to variable S.

The algorithm of the procedure is given below:

```
1.    [Check whether or not the Queue is empty]
```

```
        IF F = NULL THEN
          PRINT "Underflow"
        RETURN
        [End of IF Structure]

2.      [Delete Element]
        S = Q[F]

3.      [Check New Value of F]
        IF F = R THEN
          F = R = 0
        ELSE
        IF F = N THEN
          F = 1
        ELSE
        F = F+1
        [End of IF Structure]

4.      RETURN S
```

Program

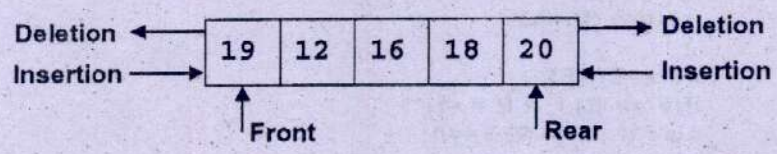
```
// Program to insert and delete items from a circular queue
#include <iostream.h>
#include <conio.h>
class que
   {
    private:
    int F, R;
    int CQ[10];
    public:
    que() // constructor to initilize F & R
        {
         F = -1; R = -1;
        }

    void insert(int);
```

```
    int  del(void);
    void print(void);
 };

main()
{
que obj;
int opt, val;
       while(opt!=3)
       {
       clrscr();
       cout<<"1: Insert Item\n";
       cout<<"2: Delete Item\n";
       cout<<"3: Exit\n";
       cout<<"Enter the choice:";
       cin>>opt;

       switch(opt)
       {
       case 1:
          cout<<"Enter Value to insert ? ";
          cin>>val;
          obj.insert(val);
          cout<<"\nQueue after insertion\n";
          obj.print();
          break;
       case 2:
          cout<<"\nValue "<<obj.del()<<" is deleted\n";
          cout<<"\nQueue after deletion\n";
          obj.print();
       }
   }
}

// member function to add items
void que::insert(int n)
        {
        if(F == 0 && R == 8)
         {
         cout<<"Circular Queue is full";
         getch();
         }

        if(F == -1)
          F = R = 0;
        else if(R == 9)
          R = 0;
        else
          R++;
```

```
        CQ[R]=n;
        if(F==-1) F=0;
    }

// member function to delete items
int que::del()
        {
        int data;
        if(F==-1)
         {
         cout<<"Circular Queue is empty";
         getch();
         return NULL;
         }
         data=CQ[F];
         if(F == R) F = R = -1;
         else if (F == 9) F==0;
          F++;
          return data;
        }

// member function to print items
 void que::print()
        {
        if(F==-1)
         {
         cout<<"Queue is empty";
         return;
         }
         for(int i=F;i<=R;i++)
          cout<<CQ[i]<<"\t";
          getch();
        }
```

DEQUES

The deque is pronounced as ***deck*** or ***de-queue***. It is a linear structure in which items can be added or removed at either end. However, an item cannot be added or deleted in the middle of the deque. The term deque stands for *double-ended queue*. Since items can be added or deleted at both ends of the deque, it is called double-ended queue. The deque is represented as shown in the following figure:



There are two variations of deques. These are:

1. Input – Restricted Deque

The input restricted deque allows insertions only at one end but allows deletions at both ends.

2. Output – Restricted Deque

The output restricted deque allows deletions only at one end but allows insertions at both ends.

Representation of Deque

The deque may be represented in the computer in several ways. Usually, it is represented by a circular array with two pointer variables pointing to its two ends.

Consider an array representing a circular deque 'DEQ'. F and R are pointer variables pointing to the addresses of Front and Rear of the deque. The DEQ can have a maximum of six elements as shown below:

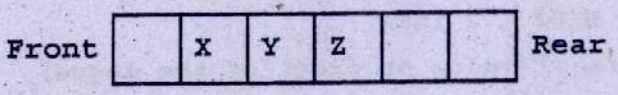
DEQ

When an item is inserted at the Front of DEQ, then F is decreased by 1.

However, when an item is inserted at the Rear, then R is increased by 1. Similarly, when an item is deleted at front, then value of F is increased by 1 and when an item is deleted from Rear, then the value R is decreased by 1.

When the values of F and R are equal, then it indicates that the deque has only one item stored in it. After deleting the last item, the values of both F and R are assigned NULL value. This indicates that the deque is empty.

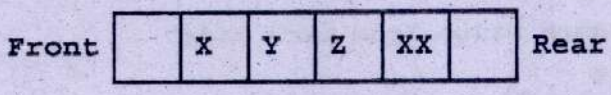
In the following figure, the DEQ has some items stored in it.



The first element at the Front of the deque is empty and two elements at the Rear are also empty. Thus the values of F and R are:

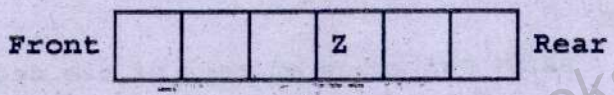
F = 2 & R = 4

If another item XX is added at the Rear, then the DEQ and values of F and R will be:



F = 2 & R = 5

If two items are deleted at the Front and one item is deleted at the Rear, then the DEQ and values of F and R will be:



F = 4 & R = 4

Algorithm — Insertion in Deque

Write an algorithm to insert values into a deque 'DQ' having ten elements.

```
1.   SET F = R = -1
2.   [Enter value to insert]
     Input VAL
3.   [Specify front or rear side to insert value]
     Input SIDE
4.   IF SIDE = 1 THEN
     [Insert value at front of the deque]
     IF F = -1 AND R = -1 THEN
          R=F=0
          DQ[F]=VAL
     ELSE IF F >0 THEN
               F= F-1
               DQ[F]=VAL
          ELSE
               PRINT "No space at front of the deque"
          END IF
     END IF
     [Insert value from back side]
     ELSE
     IF F = -1 AND R = -1 THEN
          R=F=0
          DQ[R]=VAL
     ELSE IF R < 9 THEN
          R= R+1
          DQ[R]=VAL
     ELSE
          PRINT "No space at rear of the deque"
     END IF
     EXIT
```

Algorithm — Deletion in Deque

Write an algorithm to delete values from a deque 'DQ' consisting of ten elements.

```
1.    SET F = R = -1
2.    [Specify front or rear side to insert value]
      INPUT SIDE
3.    IF SIDE = 1 THEN
      [Delete value from front side]
          IF F = R THEN
                F = R = -1
           ELSE IF F = 9 THEN F=-1
                ELSE
                   F=F+1
           END IF
      END IF

      [Delete value from back side]
         ELSE
                IF F = R THEN
                      F = R = -1
                ELSE
                      R= R-1
                END IF
4.    EXIT
```

Program

```
// Program to insert and delete items in a circular deque
#include <iostream.h>
#include <conio.h>
class deque
   {
    private:
```

```
    int F, R;
    int DQ[10];
    public:
    deque() // constructor to initilize F & R
        {
         F = -1; R = -1;
        }

    void add_front(int);
    void add_rear(int);
    void del_front(void);
    void del_rear(void);
    void print(void);
 };

main()
{
deque obj;
int opt, val;
  while(opt!=5)
   {
    clrscr();
    cout<<"1: Insert Item from Front\n";
    cout<<"2: Insert Item from Rear\n";
    cout<<"3: Delete Item from Front\n";
    cout<<"4: Delete Item from Rear\n";
    cout<<"5: Exit\n";
    cout<<"Enter the choice:[1-5] ?";
    cin>>opt;
    switch(opt)
        {
         case 1:
         cout<<"\nEnter Value to insert ? ";
         cin>>val;
         obj.add_front(val); break;

         case 2:
         cout<<"\nEnter Value to insert ? ";
         cin>>val;
         obj.add_rear(val); break;

         case 3: obj.del_front(); break;
         case 4: obj.del_rear();
        }
        cout<<"\nDeque after operation\n";
        obj.print();
   }
}
```

```
// member function to insert item from front side
void deque::add_front(int n)
 {
 if(F == 0 && R == 9)
  {
  cout<<"Deque is full";
  return;
  getch();
  }
 if(F== -1 && R==-1)
  {
   R=F=0;
   DQ[F]=n;
  }
 else if(F >0)
       {
       F--;
       DQ[F]=n;
       }
     else
       {
       cout<<"No space from front side";
       getch();
       }
}

// member function to insert item from back side
void deque::add_rear(int n)
 {
 if(F == 0 && R == 9)
  {
  cout<<"Deque is full";
  return;
  getch();
  }
 if(F== -1 && R==-1)
  {
   R=F=0;
   DQ[R]=n;
  }
 else if(R < 9)
       {
       R++;
       DQ[R]=n;
       }
     else
       {
       cout<<"No space from rear side";
       getch();
       }
```

```
}

// member function to delete item from front side
void deque::del_front()
 {
 if(F==-1 && R == -1)
  {
  cout<<"Deque is empty";
  getch();
  return;
  }
 if(F == R) F = R = -1;
 else if (F == 9) F==-1;
      else
      F++;
}

// member function to delete item from back side
void deque::del_rear()
 {
 if(F==-1 && R == -1)
  {
  cout<<"Deque is empty";
  getch();
  return;
  }
 if(R == F) F = R = -1;
 else   R--;
}

// member function to print data from deque
void  deque::print()
  {
 if(F==-1)
   {
   cout<<"Queue is empty";
   return;
   getch();
   }
  for(int i=F;i<=R;i++)
    cout<<DQ[i]<<"\t";
 getch();
 }
```

PRIORITY QUEUES

The items in queues are removed in the order in which they are added. For example, when jobs are sent to the printer, they are placed in a queue. The job sent first is printed first and the job sent in the end is printed at the end. It may not always be the requirement. Some jobs may be important than others and the user may like to print the important jobs as soon as possible. These types of jobs should have preference in the queue over other jobs. For this purpose, a special type of queue, known as priority queue, is used.

A priority queue is a collection of items in which each item is assigned a priority. The items in the priority queue are processed such that:

- The item having higher priority is processed prior to the item having lower priority.
- If more than one item have the same priority then they are processed in the order in which they are inserted in the queue.

The following figure shows a priority queue. It represents jobs waiting to be performed on the computer. The priority 1, 2, 3 and so on are attached to each job.

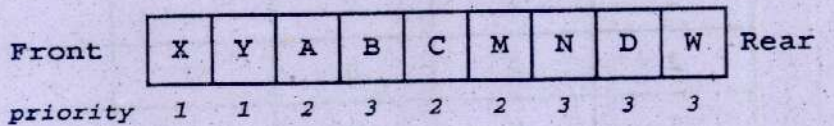


Figure: Priority Queue

The priority 1 is given to jobs X and Y, priority 2 is given to jobs A, C and M and priority 3 is given to the jobs B, N, D & W.

The insertion and deletion in a priority queue is based on the priority of the items. These can take place at any position within the queue. The queue shown in the above figure is a single queue in which insertion can be done at any position. This priority queue can be thought of as a combination of three queues, based upon the priorities of the items, as shown below:

Priority-1	X	Y				Queue-1
Priority-2	A	C	M			Queue-2
Priority-3	B	N	D	W		Queue-3

The items in queue-2 can be removed only when the queue-1 is empty. Similarly, to remove items from queue-3, both the queue-1 and queue-2 must be empty.

Representation of Priority Queue

The priority queues can be represented in the memory in several ways. The best way to represent it is to use a separate queue for each level of priority. Each such queue is represented in circular fashion and has its own front and rear pointers.

Usually, an array of arrays, i.e. a two-dimensional array is used to represent a priority queue. The figure below shows the representation for priority queue.

1	X	Y		
2	A	B	C	
3	M	N	O	P
4				

Algorithm — Insertion in Priority Queue

Write an algorithm to insert items into priority queue represented by a two-dimensional array ABC with 3 rows and 10 columns.

```
1.    [initialize Rear & Front for first row of abc]
```

```
        SET R1 & F1 to -1
2.      [initialize Rear & Front for 2nd row of abc]
        SET R2 & F2 to -1
3.      [initialize Rear & Front for 3rd row of abc]
        SET R3 & F3 to -1
4.      [Enter Value to insert in N]
        INPUT N
5.      [Enter Priority in P]
        INPUT P
6.      IF P = 1 THEN CALL ADD(N, 0, R1, F1)
7.      IF P = 2 THEN CALL ADD(N, 1, R2, F2)
8.      IF P = 3 THEN CALL ADD(N, 2, R3, F3)
9.      EXIT

        [Procedure ADD]
        ADD(X, I, R, F)
        IF R >= 9 THEN
        PRINT "QUEUE IS FULL"
        RETURN
        ELSE
        R++;
        ABC[I][R]=X
        END IF
        IF F=-1 THEN F=0
```

Algorithm — Deletion in Priority Queue

Write an algorithm to delete items from a priority queue consisting of a two-dimensional array ABC with 3 rows and 10 columns.

```
1.      [Initialize Rear & Front for first row of abc]
        SET R1 & F1 to -1
2.      [Initialize Rear & Front for 2nd row of abc]
```

```
        SET R2 & F2 to -1
3.      [Initialize Rear & Front for 3rd row of abc]
        SET R3 & F3 to -1
4.      [Enter Priority in P]
        INPUT P
5.      IF P = 1 THEN X = REMOVE(0, R1, F1)
6.      IF P = 2 THEN X = REMOVE(1, R2, F2)
7.      IF P = 3 THEN X = REMOVE(2, R3, F3)
8.      PRINT X, " deleted from the queue"
9.      EXIT

        [Procedure REMOVE]
        REMOVE(R, F, I)
        IF F=-1 THEN
                PRINT "QUEUE IS EMPTY"
                RETURN NULL
            END IF
            DATA=ABC[I][F]
            IF F = R THEN
          F = R = -1
            ELSE
               F=F+1
            END IF
        RETURN DATA
```

Program

```
// Program to insert and/or delete items from priority queue
//having 3 arrays with 10 elements each

#include <iostream.h>
#include <conio.h>
class que
   {
```

```
    private:
    int F1,R1, F2,R2, F3,R3;
    int QA[3][10];
    public:
    que() // constructor to initilize Fronts & Rears
        {
         F1 = -1; R1 = -1;
         F2 = -1; R2 = -1;
         F3 = -1; R3 = -1;
        }
    void insert(int, int);
    void add (int, int&, int&, int);
         del(int);
    int  remove (int&, int&, int);
    void print(int);
    void  ppp (int&, int&, int);


};

main()
{
que obj;
int opt, val, pro;
       while(opt!=3)
       {
       clrscr();
       cout<<"1: Insert Item\n";
       cout<<"2: Delete Item\n";
       cout<<"3: Exit\n";
       cout<<"Enter the choice:";
       cin>>opt;

       switch(opt)
       {
       case 1:
           cout<<"Enter Value to insert ? ";
           cin>>val;
           cout<<"Enter Priority ? ";
           cin>>pro;
           obj.insert(val, pro);
           cout<<"\nQueue after insertion\n";
           obj.print(pro);
           break;
       case 2:
           cout<<"Enter Priority ? ";
           cin>>pro;
           cout<<"\nValue "<<obj.del(pro)<<" is deleted\n";
           cout<<"\nQueue after deletion\n";
           obj.print(pro);
```

```
            getch();
          }
      }
}

void que::insert(int n, int p)
   {
   switch(p)
    {
     case 1: add(n,R1,F1,0); break;
     case 2: add(n,R2,F2,1); break;
     case 3: add(n,R2,F2,2); break;
    }
   }

void que::add(int x, int& R, int& F, int i)
{
 if(R>=9)
   {
    cout<<"Queue is full";
    getch();
    return;
   }
  else
  {
   R++;
   QA[i][R]=x;
   if(F==-1) F=0;
  }
}

que::del(int p)
 {
 switch(p)
    {
     case 1: remove(R1,F1,0); break;
     case 2: remove(R2,F2,1); break;
     case 3: remove(R2,F2,2); break;
    }
 }

int que::remove(int& R, int& F, int i)
{
  int data;
  if(F==-1)
   {
        cout<<"Queue is empty";
        getch();
        return NULL;
        }
```

```
        data=QA[i][F];
        if(F == R) F = R = -1;
        else F++;
        return data;
       }

void que::print(int p)
  {
  switch(p)
    {
     case 1: ppp(R1,F1,0); break;
     case 2: ppp(R2,F2,1); break;
     case 3: ppp(R2,F2,2); break;
    }
  }
void que::ppp(int& R, int& F, int i)
  {
   if(F==-1)
    {
      cout<<"Queue is empty";
      return;
      }
    for(int c=F;c<=R;c++)
    cout<<QA[i][c]<<"\t";
  }
```

EXERCISES

Q.1 Fill in the blanks.

1. In _____data structures, items are processed on first in first out basis.
2. Items can only be inserted at the back and removed from the front in _____ data structures.
3. The end at which elements are inserted into the queue is called the _____.
4. The end at which the elements are removed from the queue is called the _____.
5. FIFO stands for _____.
6. The queue data structure is represented inside the computer by using _____.
7. The linear data structure in which items can be inserted or removed at either end is called _____.
8. The term deque stands for _____.
9. The deque that allows insertions at only one end but allows deletion at both ends is called _____.
10. The deque that allows deletion at only one end but allows insertions at both ends is called _____.
11. The priority queue is usually represented inside the memory by using _____.

Q.2 Mark True or False.

1. The end at which the elements are removed from the queue is called rear.
2. FIFO stands for First Insertion First Operation.

3. The queue data structure is represented inside the computer by using a linear array.
4. The deque is pronounced as de-queue.
5. A queue is a special type of circular structure.
6. In deque data structures, the items are accessed on first in first out basis.
7. The items can only be inserted at the back and removed from the front of the queue.
8. The end at which the elements are inserted into the queue is called front.
9. The linear data structure in which items can be inserted or removed at either end is called priority queue.
10. The items in a deque can also be added or deleted in its middle.
11. The last job send to the computer is printed first.
12. In priority queue, the item that has higher priority is processed in the last.
13. The items in a queue are removed in the sequence in which they are added.
14. The jobs sent to the printer are added to a deque.

Q.3 **Differentiate between the following:**

1. Queue & Deque 2. Priority Queue & Queue

Q.4 **Explain representation of the priority queue inside the computer.**

Q.5 **What is a circular queue? Give examples.**

Q.6 **Write an algorithm to insert items into a deque.**

Q.7 **Write a program to remove and insert items into a deque.**

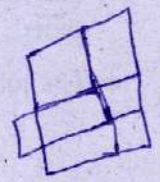
Q.8 **Write a program to insert five integer values into a queue. Remove**

the values from queue one by one, calculate factorial of each integer and print it on the screen.

Q.9 Write a program to insert ten real values into a queue. Remove the values from the queue one by one, find maximum & minimum values and print results on the screen.

Q.10 Write a program to insert five integer values into a deque. Remove all the values from the deque, calculate their average and print the results on the screen.

Q.11 Write a program to insert ten names into a priority queue. Print all the names from the priority queue that have the top priority.



SEARCHING & SORTING

SEARCHING & SORTING

Searching & sorting are two most important operations that are frequently performed on data structures. These are fundamental operations in computer science. These are mostly performed on data structures like arrays, linked lists and trees. This chapter presents techniques for implementing these operations on array data structures.

Searching

Computer systems are often used to store large amounts of data from which individual records are retrieved according to some search criterion. The process of finding a specific data item or record from a list is called searching. The efficient storage of data to facilitate fast searching is an important task. All other operations like inserting, deletion, etc. are dependent on this operation. For example, to delete a data item from a list, its position is first located in the list and then the deletion operation is performed.

The search is successful if the specified data item or record is found during searching process. Search operation terminates when it is successful. If the specified data is not found then the search is unsuccessful. The records in a database are searched based on their key field values. The database records can be searched randomly as well as sequentially. Different techniques are used to carry out search operations. The commonly used searching methods are:

- **Sequential Search**
- **Binary Search**

Sequential Search

The sequential search is a simple and straightforward technique to search a specified item in an unordered list. The specified value is searched in the list sequentially, i.e. starting from the first element to the last element in the list in a sequence. When the required value is found, search operation stops.

The sequential search is a slow process. It is used for small amounts of data. This method is not recommended for large amount of data.

Algorithm — Sequential Search

Write an algorithm to find a value ITEM from an array ABC consisting of N elements.

In the following algorithm, variable LOC has been used to hold the position of the item in the array.

```
1.    SET LOC = -1
2.    [Enter value that is to be searched]
      INPUT ITEM
3.    REPEAT STEP-4 FOR I = 1 TO N
4.    IF ITEM = ABC[I] THEN
       LOC = I
       PRINT "Data found at location ", LOC
       EXIT
      END IF
      [End of step-3 loop]
5.    IF LOC = -1 THEN
       PRINT "Item not Found"
       EXIT
      END IF
```

Program

```
// program to search in an array using sequential search
#include<conio.h>
#include<iostream.h>
```

```
class seq
{
private:
int a[10];
public:
void input(void);
void search(int);
};

main()
{
seq obj;
int item;
clrscr();
obj.input();
cout<<"Enter required value to search ? ";
cin>>item;
obj.search(item);
getch();
}

// member function to input data into array
void seq::input(void)
{
cout<<"Enter 10 values "<<endl;
for( int i=0;i<=9;i++)
cin>>a[i];
}

// member function to search data from array
void seq::search(int n)
{
int i, loc=-1;
i=0;
while(i<=9)
{
if(n ==a[i])
{
        loc=i+1;
        cout<<" Data found in location = "<<loc;
        break;
        }
i++;
}
if(loc ==-1)
cout<< " Data not found ";
}
```

Algorithm — Sequential Search

Write an algorithm to find an item V1 in an array ABC consisting of N elements. Replace the searched item with new item V2. Use variable LOC to hold the position of element that contains the required value in the array. Use sequential method to search the array.

```
1.    SET LOC = -1
2.    [Enter item to be searched in V1]
      INPUT V1
3.    [Enter item to be replaced with V1]
      INPUT V2
4.    REPEAT STEP-5 FOR I = 1 TO N
5.    IF V1 = ABC[I] THEN
       ABC [I] = V2
       LOC = I
       PRINT "Item modified at location ", LOC
       EXIT
      END IF
      [End of step-4 loop]
6.    IF LOC = -1 THEN
       PRINT "Item not Found"
       EXIT
      END IF
```

Algorithm – Largest Value in an Array

Write an algorithm to find and print the largest value in an array ABC consisting of ten elements.

```
1.    C = 1, MAX = ABC[0]

      [Assign value of first element to MAX]
2.    REPEAT Step 3 WHILE C<=10
3.    IF MAX < ABC[C] Then
```

```
        [Assign value to MAX if condition is true]
        MAX = ABC[C]
        [End IF]
        C = C + 1
        [End of Loop]
4.      PRINT MAX
5.      EXIT
```

Program

```
// program to find maximum number from array with sequential
search method

#include<conio.h>
#include<iostream.h>
class seq
 {
 private:
 int a[10];
 public:
 void input(void);
 int find(int &);
 };

main()
{
 seq obj;
 int p, m;
 clrscr();
 obj.input();
 m = obj.find(p);
 cout<<" Maximum No. = "<<m<< " at location = "<<p;
 getch();
}

 // member function to input data into array
void seq::input(void)
 {
 cout<<"Enter 10 values "<<endl;
 for( int i=0;i<=9;i++)
 cin>>a[i];
 }
```

```
// member function to find maximum number and its location from
array
int seq::find(int &loc)
{
int max, i;
i=0;
max=a[0];
while(i<=9)
{
if(max < a[i])
{
        max=a[i];
        loc=i+1;
        }
i++;
}
return max;
}
```

BINARY SEARCH

It is a more efficient technique to search a specific item from list of items. It is mostly used for relatively large lists or table of records that are sorted in ascending or descending order.

A binary search begins by searching the required value from the middle of the list. If the required value is in the middle of the list then search process terminates at that point. If the list is sorted in ascending order and the required value is greater than the value at the middle, the control goes to the higher value to search the required value. Similarly, if the list is sorted in ascending order and the required value is less than the value at the middle, the control goes to the lesser values to search the required value. In both cases, half of the list is searched to find the required value.

Algorithm — Binary Search

Write an algorithm to find a value in an array ABC consisting of N elements sorted in ascending order. Use variable LOC to hold the position of the element that has the required value. Also use MID variable to hold the position of the middle of the array.

```
1.    SET LOC = -1
2.    [Enter value to be searched]
      INPUT ITEM
3.    MID = (1 + N)/2
4.    IF ITEM = ABC[MID] THEN
       PRINT "Value found at middle"
       EXIT
      END IF
5.    [Search towards right]
      IF ITEM > ABC[MID] THEN
       REPEAT FOR I = MID+1 TO N BY 1
       IF ITEM = ABC[I] THEN
       PRINT "Value at position ", I
       EXIT
      END IF
      ELSE  [Check towards left]
      REPEAT FOR I = MID-1 TO 1 BY -1
      IF ITEM = ABC[I] THEN
       PRINT "Value at position ", I
       EXIT
      END IF
      IF LOC = -1 THEN
       PRINT "Data not found"
       EXIT
      END IF
```

Program

```
// program to search a required value from array with Binary search
method

#include<conio.h>
#include<iostream.h>
class bin
 {
 private:
 int a[10];
```

```
 public:
 void input(void);
 void sort(void);
 void bsearch(int);
 };

main()
{
 bin obj;
 int val;
 clrscr();
 obj.input();
 obj.sort();
 cout<<"Enter value to search ? ";
 cin>>val;
 obj.bsearch(val);
 getch();
}

// member function to input data into array
void bin::input(void)
 {
 cout<<"Enter 10 values "<<endl;
 for( int i=0;i<=9;i++)
 cin>>a[i];
 }

// member function to sort array in ascending order
void bin::sort(void)
 {
 int t, i, c;
 for(c=0;c<=9;c++)
 {
        t=a[c];
        for(i=c;i>0 && t < a[i-1];i--)
         a[i]=a[i-1];
        a[i]=t;
}
cout<<"Sorted array \n";
for(i=0;i<=9;i++)
cout<<a[i]<<endl;
}

// member function find item from array
void bin::bsearch(int item)
{
int mid, loc;
loc=-1;
mid = (0+9)/2;
if (item == a[mid])
{
        cout<<"Required value found at mid ";
        return;
       }
if(item > a[mid])
```

```
{
        for(int i=mid+1;i<=9;i++)
         if(item==a[i])
         { loc=i+1; break; }
        }
else
        {
        for(int i=mid;i>=0;i--)
         if(item==a[i])
         { loc=i+1; break; }
        }
if(loc==-1)
        cout<<" Data not found ";
else
        cout<<" Data found at location = "<<loc;
}
```

SORTING

The process of arranging data in a specified order according to a given criteria is called sorting. The numeric type data may be arranged either in ascending (increasing) or in descending (decreasing) order. Similarly, character type data may be arranged in alphabetical order.

Sorting is one of the most important operations performed by computers. It is mostly used to arrange records in databases. A record in a database consists of several fields. It contains one (or more) field whose value uniquely determines the record in the database file. Such a field is called the **key field** or the **primary key**. The value that a key field holds is called the **key value**. Sorting operation is performed on the key values of records.

For example, in a database consisting of records of students, the key field may be "roll numbers" of students. The value in this field is always unique as no two students can have the same roll number. The records of student database can be sorted according to the class roll numbers.

There are several methods for sorting data. The common sorting methods can be divided into two types based upon the complexity of their algorithms. One type of sorting algorithms includes bubble sort, insertion sort and selection sort. The other type consists of the merge sort. All these algorithms are very simple. The choice of a method depends upon the type and size of data to be sorted.

BUBBLE SORT

Bubble sort is the oldest and simplest method. It is also the slowest. It works by comparing each item in the list with the item next to it, and swapping them if required. The process is repeated until no item is swapped in the pass all the way through the list. This causes larger values to "bubble" to the end while smaller values "sink" towards the beginning of the list.

The bubble sort is generally considered to be the most inefficient sorting algorithm. However, it is simple and can be easily implemented. It is used for sorting only a small amount of data.

To sort data in an array of n elements, **n–1** iterations are required. The following steps explain sorting of data in an array in ascending order using bubble sort method:

In first iteration, the largest value moves to the last position in the array.

- In the second iteration, the above process is repeated and the second largest value moves to the second last position in the array, and so on.
- In n–1 iteration, the data is arranged in ascending order.

To understand bubble sort, a numerical example is given below. Suppose the name of the array is A and it has four elements with the following values:

4	19	1	3

To sort this array in ascending order, n-1, i.e. three iterations will be required. These iterations are:

Iteration -1: A[0] is compared with element A[1]. Since 4 is not greater than 19, there will be no change in the list, i.e. elements are not interchanged.

A[1] is compared with element A[2]. Since 19 is greater than 1, the values are interchanged.

A[2] is compared with element A[3]. Since 19 is greater than 3, the values are interchanged.

Thus at the end of the first iteration, the largest values moves to the last position in the array.

4 ↔ 19	1	3
4	19 ↔ 1	3
4	1	19 ↔ 3
4	1	3	19

Iteration – 2: A[0] is compared with element A[1]. Since 4 is greater than 1, the values are interchanged.

A[1] is compared with third element A[2]. Since 4 is greater than 3, the values are interchanged.

At the end of the second iteration, the second largest value moves to the second last position in the array.

4 ↔	1	3	19
1	4 ↔	3	19
1	3	4	19

Iteration – 3: A[0] is compared with element A[1]. Since 1 is not greater than 3, the values are not interchanged.

In this iteration, no values are interchanged. The array is already sorted in ascending order.

1	3	4	19

Algorithm — Bubble Sort

Write an algorithm to sort an array A consisting of N elements in ascending order using bubble sort.

Suppose variable U represents the control variable for upper loop to control the number of iterations. Similarly, variable I represents the control variable for inner loop that scans the array starting from element A[1] to element A[U].

```
1.    SET U = N
2.    [Upper loop]
      REPEAT STEP-3 TO 7 WHILE (U>=1)
3.    SET I = 1
4.    [Inner loop]
      REPEAT STEP-5 TO 6 WHILE (I<=U)
5.    IF A[I] > A[I+1] THEN [Interchange value]
        T = A[I]
        A[I] = A[I+1]
        A[I+1] = T
      END IF
6.    I = I + 1  [End of step-4 loop]
7.    U = U - 1  [End of step-2 loop]
8.    EXIT
```

Progra

```
// program to sort array with 5 elements using bubble sort

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

class bubble
 {
 private:
 int a[5];
 public:
 input()                    // Input data into array
 {
 cout<<"Enter 5 values "<<endl;
 for( int i=0;i<=4;i++)
 cin>>a[i];
 }

 sort()          // Sort array
 {
 int t;
 for(int u=4;u>=1;u--)
 for(int i=0;i<u;i++)
        if(a[i]>a[i+1])
            {
             t=a[i];
             a[i]=a[i+1];
             a[i+1]=t;
             }
 }

 print() // Member function to print sorted data
 {
 cout<<"Sorted array "<<endl;
 for( int i=0;i<=4;i++)
 cout<<a[i]<<endl;
 }
};

main()

{
 bubble obj;
 clrscr();
 obj.input();
 obj.sort();
 obj.print();
 getch();
}
```

SELECTION SORT

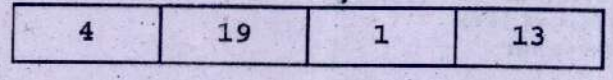
Selection sort works by selecting the smallest unsorted item remaining in the list and then swapping it with the item in the next position to be filled.

Selection sort is simple and easy to implement. It is, however, inefficient for large lists. It is usually used to sort lists of no more than 1000 items. In an array of n elements, n–1 iterations are required to sort the array.

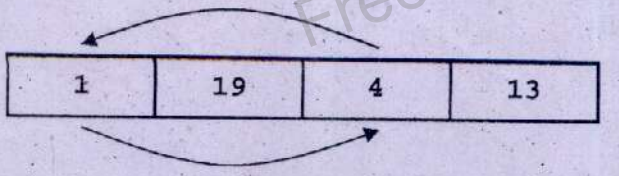
The selection sort process to sort an array in ascending order consists of the following iterations:

- In first iteration, the array is scanned from the first to the last element and the element that has the smallest value is selected. The value of the selected smallest element is interchanged with the first element of the array.
- In second iteration, the array is scanned from second to the last element and the element that has smallest value is selected. The value of smallest element is interchanged with the second element of the array.
- This process is repeated until the entire array is sorted.

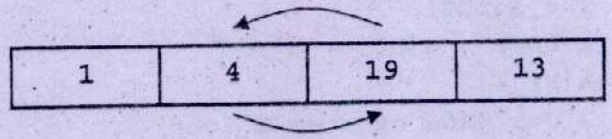
In the following example, an array consisting of 4 elements is sorted using selection sort. There will be three iterations to sort the list in ascending order. The array and the selection sort iterations are:



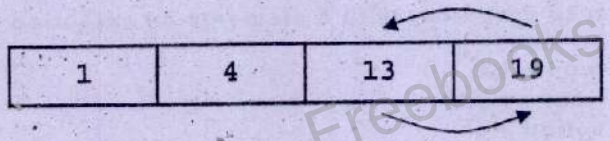
Iteration – 1: The array is scanned starting from the first to the last element and the element that has the smallest value is selected. The smallest value is 1 at location 3. The address of element that has the smallest value is noted and the selected value is interchanged with the first element, i.e. values of A[0] and A[2] are swapped.



Iteration- 2: The array is scanned starting from the second to the last element and the element that has the smallest value is selected. The smallest value is 4 at location 3. The address of selected element is noted. The value of the second element is interchanged with the third element, i.e. values of A[1] and A[2] are swapped.



Iteration – 3: The array is scanned starting from the third to the last element and the element that has the smallest value is selected. The smallest value is 13 at location 4. The address of selected element is noted. The values of the fourth element is interchanged with the third element, i.e. values of A[2] and A[3] are swapped.



Algorithm — Selection Sorting

Write an algorithm to sort an array A consisting of N elements in ascending order using selection sort.

Suppose, variable U represents the control variable for upper loop that controls the number of iterations and variable I represents the control variable for inner loop that scans the array starting from element A[U] to the last element. The variable LOC is used to hold the location of the element that has the smallest value.

```
1.    Set U = 0
2.    REPEAT STEP-3 TO 9 WHILE(U < N)
3.    TEMP=A[U]
4.    I = U
5.    REPEAT STEP-6 TO 7 WHILE(I<=N)
6.    IF TEMP > A[I] THEN
```

```
        Temp = A[I]
        LOC = I
      END IF
7.    I = I+1 [End of step-5 loop]
8.    [Interchange values]
        T = A[LOC]
        A[LOC] = A[U]
        A[U] = T
9.    U = U+1 [End of step-2 loop]
10.   EXIT
```

Program

```
// program to sort array with 5 elements by selection sort

#include<conio.h>
#include<iostream.h>
class selection
{

private:
int a[5];
public:
input()        // Input data into array

{
cout<<"Enter 5 values "<<endl;
for( int i=0;i<=4;i++)
cin>>a[i];
}

sort() // Sort array
{
int min, t, loc;
for(int u=0;u<4;u++)
{
min=a[u];
loc=u;
for(int i=u;i<=4;i++)
if(min>a[i])
      {
      min=a[i];
      loc=i;
      }
// swap values
```

```
            t=a[loc];
            a[loc]=a[u];
            a[u]=t;
            cout<<endl;
        }
        }

        print()
        {
        cout<<"Sorted array "<<endl;
        for( int i=0;i<=4;i++)
        cout<<a[i]<<endl;
        }

};

main()
{
 selection obj;
 clrscr();
 obj.input();
 obj.sort();
 obj.print();
 getch();
}
```

INSERTION SORT

The insertion sort works by inserting each item into its proper place in the final list. The simplest implementation of this requires two list structures — the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort.

The insertion sort is as simple as the bubble sort but it is almost twice as efficient as the bubble sort. It is relatively simple and easy to implement. However, it is inefficient for large lists.

In insertion sorting, the list or array is scanned from the beginning to the end. In each iteration, one element is inserted into its correct position relative to the previously sorted elements of the list. The array elements are not swapped or interchanged. They are shifted towards the right of the list to make room for the new element to be inserted.

Suppose A is an array with 6 elements A[1], A[2], A[3], A[4], A[5], A[6] having values 16, 17, 2, 8, 18, 1 respectively as shown below:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
16	17	2	8	18	1

To sort this array in ascending order 6 iterations are required. These are explained below:

Iteration – 1: A[1] is compared with itself and it is not shifted. The array A remains the same.

16	17	2	8	18	1

Iteration – 2: All data of elements on left of A[2] that are greater than A[2] are shifted one position to the right to make room for A[2] to insert its data into the correct location.

There is only one element with value 16 to the left of A[2]. Thus

no shifting takes place because 16 is less than 17. So A[1] and A[2] are in correct position relative to each other. The array A remains same.

16	17	2	8	18	1

Iteration – 3: All data of elements on left of A[3] that are greater than A[3] are shifted one position to the right to make room for A[3] to insert its data into the correct location.

There are two elements on left side of A[3] and both are greater than A[3]. Thus shift data of A[1] & A[2] one position to right and insert the data of A[3] at A[1]. The array A after shifting and inserting value is:

2	16	17	8	18	1

Iteration – 4: All data of elements on left of A[4] that are greater than A[4] are shifted one position to the right to make room for A[4] to insert its data into the correct location.

There are three elements on the left of A[4] and A[2] & A[3] are greater than A[4]. Thus shift data of A[2] & A[3] one position to right and insert the value of A[4] at A[2]. The array A after shifting and inserting the value is:

2	8	16	17	18	1

Iteration – 5: All data of elements on left of A[5] that are greater than A[5] are shifted one position to the right to make room for A[5] to insert its data into the correct location.

There are four elements on the left of A[5] and all are less than A[5]. Thus no shifting & insertion takes place. The array A remains same:

2	8	16	17	18	1

Iteration – 6: All data of elements on left of A[6] that are greater than A[6] are shifted one position to the right to make room for A[6] to insert its data into the correct location.

There are five elements on the left of A[6] and all are greater than A[6]. Thus shift data of each element from A[1] to A[5] one position to right and insert the value of A[6] at A[1]. The array A after shifting and inserting value is:

1	2	8	16	17	18

Algorithm: — Insertion Sorting

Write an algorithm to sort an array A having N elements in ascending order using insertion sort method.

Suppose variable C represents the control variable for upper loop that controls the number of iterations and variable L is used to control the inner loop to shift the values to the right.

```
1.  REPEAT STEPS-2 TO 6 FOR C = 1 TO N
2.  TEMP = A[C]
3.  L = C
4.  REPEAT STEP-5 WHILE(L>0 AND TEMP<A[L-1])
5.  A[L] = A[L-1]     [Shift to right]
    L = L - 1
    [End of step-4 inner loop]
6.  A[L] = TEMP       [Insert value]
    [End of step-1 upper loop]
7.  EXIT
```

Program

```
// program to sort array with 5 elements by insertion sort method

#include<conio.h>
#include<iostream.h>
class insertion

{
private:
int a[5];
public:
input() // Input data into array
{
cout<<"Enter 5 values "<<endi;
for( int i=0;i<=4;i++)
        cin>>a[i];
}

sort() // Sort array
{
int temp, i, c;
for(c=0;c<=4;c++)
{
        temp=a[c];
        for(i=c;i>0 && temp < a[i-1];i--)
         a[i]=a[i-1];
        a[i]=temp;
}
}

print()
{
cout<<"Sorted array "<<endl;
for( int i=0;i<=4;i++)
cout<<a[i]<<endl;
}

};

main()
{
insertion obj;
clrscr();
obj.input();
obj.sort();
obj.print();
getch();
}
```

MERGE SORTING

The merge sort splits the list to be sorted into two equal halves and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list.

Elementary implementations of merge sort make use of three arrays — one for each half of the data and one to store the sorted list. For in place merging of arrays, only two arrays are required. There are non-recursive versions of the merge sort, but they do not give any significant performance enhancement over the recursive algorithm on most computers.

The merge sort requires more memory than other sorting methods because it requires use of an additional array. This additional memory requirement makes it unattractive for most purposes.

To sort an array using merge sort:

- **Split the array into two arrays**
- **Sort the left half**
- **Sort the right half**
- **Merge the two halves into the final sorted array**

The array is split into two halves by finding the midpoint of the array. If the lower bound of the array is LB and upper bound is UB, then the midpoint of array is:

```
MID = (LB+UB)/2
```

First sub-array has lower bound LB and upper bound MID. Second sub-array has lower bound MID+1 and upper bound UB.

Suppose A is an array with 6 elements A[1], A[2], A[3], A[4], A[5], A[6] having values 16, 17, 2, 8, 18 and 1, respectively as shown below:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
16	17	2	8	18	1

The array A is divided into two sub-arrays. First sub-array has lower bound 1 and upper bound 3, i.e. [(1+6)/2 = 3]. Similarly, the second sub-array has lower bound 4 and upper bound 6.

Algorithm — Merging Sorting

Write an algorithm to sort an array AB consisting of N elements using merge sort.

To sort the array using merge sort, the array AB is divided into two sub-arrays A & B. The array A has lower bound & upper bound as 1 and R respectively. Similarly, array B has lower bound & upper bound as 1 and S, respectively. The arrays A & B are sorted and combined. The combined stored array is sorted back into the same array AB.

```
1.   Divide array AB into two sub-arrays A & B.
2.   Sort A & B
3.   Set L1 = 1 [Access elements of A]
4.   Set L2 = 1  [Access elements of B]
5.   Set L = 1   [Store data into elements of AB]
6.   REPEAT WHILE L1 <= R AND L2 <=S
          IF A[L1]< B[L2] THEN
                AB[L] = A[L1]
                L = L+1
                L1= L1+1
          ELSE
                AB[L] = B[L2]
                L = L+1
                L2 = L2+1
          END IF
     [End of step-6 loop]
7.   [Assigning remaining elements into AB]
     IF L1 > R THEN
          REPEAT FOR I = 0 TO S-L2
          AB[L+I] = B[L2+I]
     ELSE
          REPEAT FOR I = 0 TO R-L1
          AB[L+I] = B[L1+I]
     END IF
8.   EXIT
```

When the array with an odd number of elements is divided into two sub-arrays then the second sub-array will have one more element than the first sub-array. For example, the array with elements 8 is divided into two sub-arrays of

equal lengths, i.e. each having 4 elements. However, if length of an array is 9, then first sub-array will have 4 elements and the second array will have 5 elements.

Program

```
// program to sort array with 10 elements by merging sort method
#include<conio.h>
#include<iostream.h>
class merging
 {
 private:
 int ab[10], a[5], b[5],r,s;
 public:
 void sort(void);
 void input(void);
 void print(void);
};

main()
{
 merging obj;
 clrscr();
 obj.input();
 obj.sort();
 obj.print();
 getch();
}

void merging::input(void) // Input data into array ab
 {
 cout<<"Enter 10 values "<<endl;
 for( int i=0;i<=9;i++)
 cin>>ab[i];
 }
void merging::sort()
 {
 // cut array ab into two sub-arrays
 r=(9+0)/2;
 s=r+1;
 int i;
 for(i=0;i<=r;i++)
 a[i]=ab[i];
 for(i=s;i<=9;i++)
 b[i-s]=ab[i];

 // sort sub_arrays
 int temp, c;
 for(c=0;c<=r;c++) // sort first sub_array
 {
 temp=a[c];
```

```
    for(i=c;i>0 && temp < a[i-1];i--)
            a[i]=a[i-1];
            a[i]=temp;
    }

    for(c=0;c<s;c++)     // sort second sub_array
    {
            temp=b[c];
            for(i=c;i>0 && temp < b[i-1];i--)
            b[i]=b[i-1];
            b[i]=temp;
            }
    // merge two arrays into sorted form
    int L1, L2, L;
    L1=0;
    L2=0;
    L =0;
    while(L1<=r && L2 <s)
            {
             if (a[L1] < b[L2])
             {
             ab[L]=a[L1];
             L++;
             L1++;
             }
             else
             {
             ab[L]=b[L2];
             L++;
             L2++;
             }
            }

    if(L1 >r)
            for(i=0;i<s-L2;i++)
            ab[L+i]=b[L2+i];
    else
            for(i=0;i<=r-L1;i++)
            ab[L+i]=a[L1+i];
    }
    void merging::print()
    {
    cout<<"Sorted array "<<endl;
    for( int i=0;i<=9;i++)
    cout<<ab[i]<<endl;
    }
```

EXERCISES

Q.1 Fill in the blanks

1. The process of finding a specific data item in a list is called _____.
2. The search is said to be _____ if the required item is found.
3. If the specified data is not found then the search is _____.
4. The record of a database is searched based on its _____.
5. In _____ search, the list of data is searched from first element to the required data item one after the other in a sequence.
6. The major drawback of _____ search method is that it is very slow.
7. The _____ search method begins by searching the required value in the middle of the list.
8. In _____ search method, only half of the list is checked to find the required item in the list.
9. The process of arranging data in a specified order according to a given criteria is called _____.
10. In _____ sorting, the list of data is sorted in ascending order by comparing two neighboring elements and if the first is larger than the second, then they are interchanged.
11. In _____ sort, the larger value slowly floats to the top.
12. In _____ sort, the array is scanned from first element to last element and the element that has the smallest value is shifted to the first position of array in first iteration.
13. In _____ sort, the data of elements is not swapped or interchanged. Only the data of elements is shifted towards the right of list to make room for the new element to be inserted.

Q.2 Mark True or False.

1. The sequential search begins by searching the required value in the middle of the list.
2. In binary search, only half of the list is checked to find the required number from the list.
3. The process of arranging data in a specified order according to a given criteria is called searching.
4. In ascending order the data is arranged in decreasing order.
5. In bubble sort, a list of data is sorted in ascending order by comparing two neighboring elements and if first is larger than second then they are interchanged.
6. In bubble sort, the larger value slowly floats to the top.
7. In insert sort, the array is scanned from first element to last element and the element that has smallest value is selected and inserted at its correct position.
8. In insertion sort, data of elements are not swapped or interchanged but data of elements is shifted towards right to make room for the new element to be inserted.
9. The process of combining two sorted sub-arrays (or arrays) into a single sorted array is called Merge sorting.
10. The process of finding a specific data item in a list is called searching.
11. The search is said to be unsuccessful if the specified item is found.
12. If the specified data is not found then the search is successful.
13. The record of a database is searched based on data in the key field.
14. The records in a database can be searched randomly as well as sequentially.
15. The specified value is searched from a list sequentially, i.e. from first element to last element.
16. In binary search, the entire list is searched from the first element to the end.

17. The binary search is a more efficient technique than the sequential search.

18. If the numeric list of items is sorted in ascending order then the first element of list has the lowest value.

Q.3 **Write an algorithm to find the smallest and the largest values in an array.**

Q.4 **Write an algorithm to search the position of the largest value in an array.**

Q.5 **Write a program to find the largest value in an array using binary search.**

Q.6 **Write a program to sort a list of names in descending order using insertion sort.**

Q.7 **Differentiate between the following:**

a. **Insertion sort & Bubble sort**

b. **Selection sort & Merge sort**

c. **Sequential search & Binary search**

LINKED LISTS

LINKED LISTS

A data structure can be arranged in memory in several ways. Each method has advantages and disadvantages. For example, elements of an array are stored sequentially in the memory. The sequential storage is not either possible or efficient in larger computer systems because:

- Where a number of users share main memory, there may not be enough adjacent memory locations left to hold an array. But there could be enough memory in the shape of small free blocks.
- Array is a static data structure. The size of array cannot be changed during the program execution.
- The data access speed becomes slow as the size of the array increases.

To overcome these limitations, linked lists are used. In linked lists, data is arranged into records. Each record is called a node. A data item may be stored anywhere in the memory. A special item, called pointer, is added to each record. It is used to contain a link to the next record. Thus an item in the list is linked logically with the next item by assigning to it the memory address of next item.

There are two types of linked lists. These are:

- **Single-linked lists *or* One-way lists**
- **Double-linked lists *or* Two-way lists**

Pointer

The concept of pointers is fundamental to understanding the concept of linked lists. A pointer is a variable that is used to store memory address of another variable. It is also simply called pointer variable. The data type of the pointer and of the variable whose address it stores must be the same. Pointers always have the

same memory size. This property of pointers is used to perform operations on data in the memory in a uniform manner.

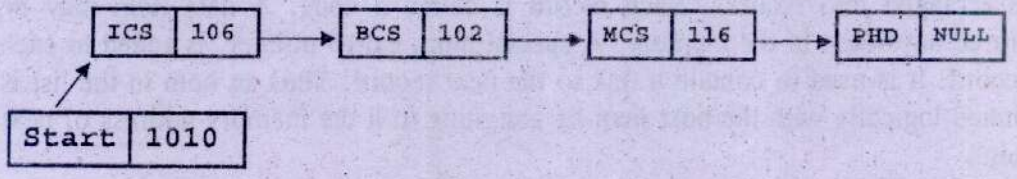
The use of pointers to link elements of a data structure implies that the elements need not be physically adjacent in the memory. They can be stored anywhere in the memory but they logically remain adjacent. This type of storage is called linked storage.

Single Linked List

A single linked list is a linear collection of data elements. The elements in a single linked list can be visited starting from beginning and towards its end, i.e. only in one direction. Therefore, the single linked list is also called one-way list or one-way chain.

Each node in a single linked list consists of at least two fields. The first field contains the data or value called data field (more than one data fields can be used in a node to store information). The second field contains the pointer or link to the next node in the list. This field is called the linked field or pointer field. The linked field of the last node contains a NULL value. A NULL value in the pointer field indicates that pointer does not point to any other node. The single linked list that has no node is called empty list. It has a value NULL.

An example of single linked list is given below.



The variable Start (called list pointer variable) contains the memory address of the first node of the list. It points to the starting memory location of the linked list. The linked list is accessed with reference to this pointer variable. The first part of the node contains the actual data of the node and the second part contains the memory address of the next node in the list. The last node of the list contains a NULL value in its pointer field. If a new item is to be added at the end of the linked list, then a memory address is assigned to the pointer field of last node and a new item is added.

In the single-linked list, each node has only the address of the next node in the link. It does not contain the address of the previous node. Thus it can only be visited in one direction, i.e. starting from the beginning toward the end of list.

Representation of Linked List in Memory

The storage technique of linked lists is different from that of linear arrays. The items of linked list are stored in memory in scattered form but these are linked together through pointers. Each item of list is called an object. One object may contain one or more fields. Each item or object of linked list contains at least two fields. One field is used to store data and the other to store address of memory location of the next object. In this way the objects of list are linked together in the form of a chain. The following example explains this concept using a C++ structure:

```
struct test
    {
      int data;
      test * link;
    };
test *start;
```

In the above example, structure "test" is defined with two fields:

- "data" is of **int** type and is used to store integer values in the node.
- "link" as a pointer to the object "test". It is used to store the memory addresses. It contains the memory address of the next object in the chain.

Another variable "start" of pointer type that points to the structure type object "test" has also been declared. It is used to store the memory address of the first or starting object in the linked list.

IN C++, nodes of the linked list are created by using the "new" operator during program execution. The "new" operator allocates memory for the specified object and returns its memory address to the pointer variable. Its syntax is:

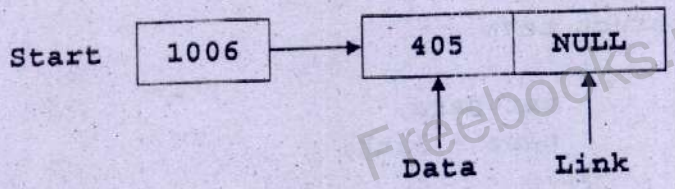
```
pointer-variable = new object-name;
```

The operator "->" (hyphen and greater than sign) is used to access the data of pointer type objects.

To allocate memory to variable "start" of 'test' type and store values in it, the statements are written as:

```
start = new test;
start -> data = 405;
start -> link = NULL;
cout<<"Value in starting node = "<<start -> data;
```

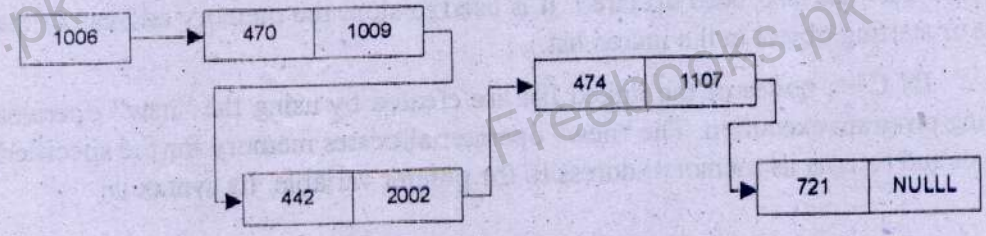
Suppose the memory address of variable "start" is 1006. The storage in the memory will be:



The memory allocation for an item of the list exists until the program is terminated or the created item is destroyed with the "delete" operator.

A new node can also be created by the "new" operator and values can be stored into it during program execution. While creating a new node, the address of the previous node is stored in a pointer variable. The memory address of newly created item is stored in the pointer field of the previous node.

The representation of linked list having nodes in memory is given below:



The above list is a linear linked list. It does not occupy consecutive memory locations like an array. Each item of the list is at a different location. These are at memory addresses 1006, 1009, 2002 and 1107 and have values 470, 442, 474 and 721, respectively. Each node has a pointer variable that holds the address of next node. The pointer of the last object does not point to any object. It contains NULL value. The whole list is accessed with reference of the pointer object that holds the starting address of the linked list.

Algorithm — Creating Linked List

Write an algorithm to create a linked list consisting of ten nodes and enter values into the list items.

```
1.  DEFINE structure of node
2.  DECLARE three pointer objects start, current &
    next of type node
3.  [Create first node and assign its address to
    "start". Also assign value to the node]
    start = new-node
    node  = value
4.  [Assign address of start node to the current node
    to proceed next]
    current = start
5.  Repeat Step-6 to 7 FOR C = 1 to 9
6.  Create new-node
    [Assign value and address of new node to current
    node and address of new-node to current]
7.  new-node-data = value
    current pointer = new-node
    current = new-node
    [End of step-5 loop]
8.  Exit
```

Program

```
// Program to create a new linked list & store values into its nodes
#include<iostream.h>
#include<conio.h>
struct node
    {
     int data;
     node *link;
     };

class list
    {
    private:
     node *start, *cur, *temp;
    public:
     list()
        {
         start = NULL;
        }

// member function to create and add data into list
     append(int n)
        {
        // create first node & assign data
        if(start == NULL)
         {
          start = new node;
          start->data = n;
          start->link=NULL;
         }
        else
         {
          cur=start;

          // go to end of list
          while(cur->link!=NULL)
              cur=cur->link;

          // create and add new nodes at end
              temp=new node;
              temp->data = n;
              temp->link=NULL;
              cur->link=temp;
         }
       }

// member function to print data from nodes
     print()
```

```
        {
        cout<<"\nData in link list\n\n" ;
        cur=start;
        while(cur->link!=NULL)
         {
         cout<<cur->data<<endl;
         cur=cur->link;
         }

        // print last node value
        cout<<cur->data<<endl;
       }
};

main()
{
list obj;
int val,p;
clrscr();
cout<<"Enter five values \n" ;
for(int i=1; i<=5;i++)
{
cin>>val;
obj.append(val);
}
obj.print();
getch();
}
```

Singe-Linked List Operations

Like other data structures, different operations can also be performed on single linked lists. The important operations are:

- Traversing
- Insertion
- Deletion
- Searching
- Sorting

Traversing Single Linked List

The single linked list is visited from the beginning to the end of the list, until NULL occurs in the pointer object of the list element. A linked list is usually visited to display its items or to count the number of items or nodes in it.

Algorithm — Displaying Linked Lists

Write an algorithm to display all nodes of a linked list.

Suppose "temp" is a linked list created and values have been entered into it. The following algorithm displays the data of all nodes of the linked list. Each node of a list has two fields. The first is data field for storing data and the other is link field for storing address of the next node.

```
1.  REPEAT step-2 to 3 WHILE temp->link != NULL
2.  PRINT temp->data
3.  temp=temp->link [Move to next node]
    [End of step-1 loop]
4.  EXIT
```

Algorithm — Counting

Write an algorithm to count total number of items or nodes in a linked list.

```
1.  C = 0 [Assign value zero to the counter]
2.  Repeat step-3 to 4 WHILE TEMP->LINK != NULL
3.  C = C + 1
4.  [Move to next mode]
    TEMP = TEMP -> LINK
    [End of step-2 loop]
5.  PRINT "Total items are = ", C
6.  EXIT
```

Program

```
// Program to count total number of nodes in a linked list

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

```
struct node
    {
     int data;
     node *link;
     };

class list
    {
    private:
     node *start, *cur, *temp;
    public:
     list()
        {
         start = NULL;
        }

// member function to add items into list
     add_item(int n)
         {
         // create first node & assign data
         if(start == NULL)
          {
           start = new node;
           start->data = n;
           start->link=NULL;
           }
          else
           {
            cur=start;
            // go to end of list
            while(cur->link!=NULL)
                cur=cur->link;
            // create and add new nodes at end
                temp=new node;
                temp->data = n;
                temp->link=NULL;
                cur->link=temp;
           }
         }

// member function to count nodes
     int count()
         {
         int c=0;
         cur=start;
         while(cur->link!=NULL)
          {
          c++;
          cur=cur->link;
          }
```

```
        return c+1;
        }
    };

main()
{
list obj;
int val;
clrscr();
cout<<"Enter five values \n" ;
for(int i=1; i<=5;i++)
{
cin>>val;
obj.add_item(val);
}
cout<<"Total items = "<<obj.count();
getch();
}
```

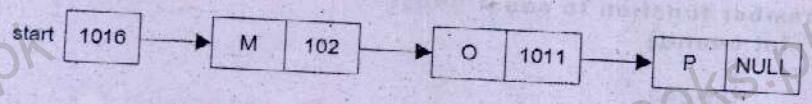
Insertion in Linked List

The insertion operation in linked list is simple and easy. A new node can be inserted in a linked list at any of these locations:

- at the end of list
- at the beginning of list
- at any specified location in the list

Algorithm — Insertion at the End of Linked List

The new node is inserted in the list by simply interchanging the pointer fields. Suppose a list having three nodes is given below:

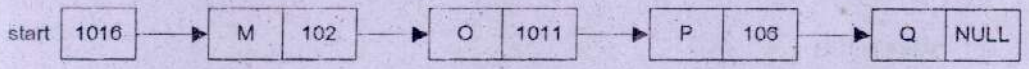


To add new node with value Q at end of the above list.

1. Create the list
2. Create a new node using the "new" operator

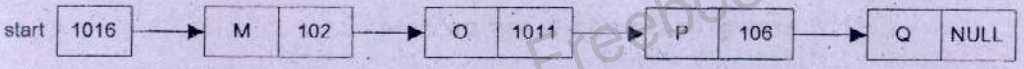
3. Go to the end of list and interchange the values of pointer fields of the newly created node and last node of the list

4. Assign NULL value to the pointer field of the newly created field and assign value Q to its data field. The new created node becomes the last node of list. Suppose the new created node has address 106, the list will be as shown below:



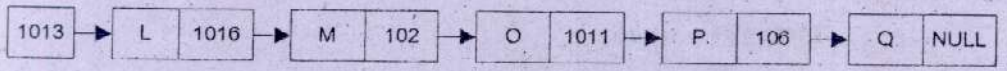
Algorithm — Insertion at the Beginning of Linked List

Write an algorithm to add a new node having value L at the beginning of the linked list shown in the figure below.



1. Create the list
2. Create a new node
3. Go to the beginning of the list, i.e. to the first node of the list
4. Interchange the starting address of list with new created node. The address of the newly created node becomes the starting address of the list and previous starting address is assigned to the pointer field of newly created node
5. Assign value L to the data field of newly created field

The figure given below shows the linked list after inserting a node at its beginning. The address of the newly created node is 1013.



Program

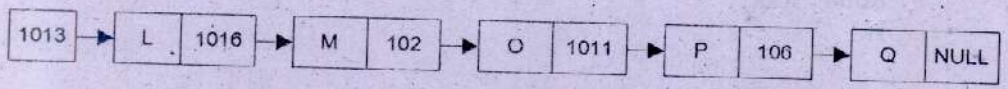
```
// Program to insert a node at the Beginning of Linked List
#include<iostream.h>
#include<conio.h>
struct node
   {
    int data;
    node *link;
   };

class list
   {
   private:
    node *start, *cur, *temp;
   public:
    list()
      {
       start = NULL;
      }

    add_item(int n)
      {
      // create first node & assign data
      if(start == NULL)
       {
        start = new node;
        start->data = n;
        start->link=NULL;
       }
      else
       {
        cur=start;
        // go to end of list
        while(cur->link!=NULL)
            cur=cur->link;
        // create and add new nodes at end
            temp=new node;
            temp->data = n;
            temp->link=NULL;
            cur->link=temp;
       }
    }

// member function to insert node
    insert(int x)
     {

      temp=new node;
```

```
            temp->data = x;
            temp->link=start;
            start=temp;
           }

        print()
           {
           cur=start;
           cout<<"\nValue of List \n\n";
           while(cur->link!=NULL)
            {
            cout<<cur->data<<endl;
            cur=cur->link;
            }
           // print value of last node
           cout<<cur->data;
           }
    };
 main()
 {
 list obj;
 int val, n;
 clrscr();
 cout<<"Enter five values \n" ;
 for(int i=1; i<=5;i++)
 {
 cin>>val;
 obj.add_item(val);
 }
 cout<<"Enter value to insert ? " ;
 cin>>n;
 obj.insert(n);
 obj.print();
 getch();
 }
```

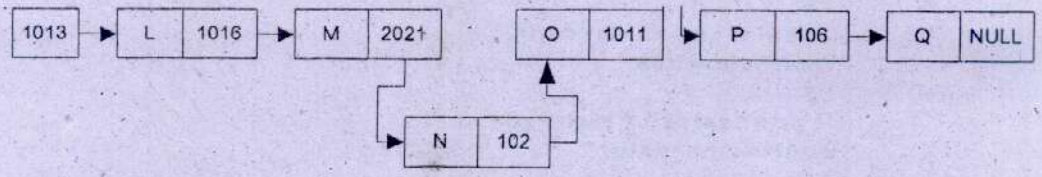
Algorithm — Insertion at a Specified Location in Linked List

Write an algorithm to add a new node at a specified location within a linked list.

Suppose a node having value N is to be inserted between nodes having values M and O of the linked list shown in the following figure:



A new node is created in the memory and value N is assigned to it. Suppose address of the newly created node is 2021. To insert the node, the pointer values of M and N are interchanged. The pointer of M will contain the memory address of the newly created node, i.e. 2021. Similarly, the pointer of N will contain the memory address of the node M, i.e. 102. After inserting the new node, the list will be as shown below:



The steps of the algorithm to insert a new node at any specified location in the linked list are:

1. Create list
2. Create a new node and assign value N to its data field
3. Go to the node after which the new node is to be inserted
4. Assign the address of newly create node to the pointer field of node that is to precede it. Assign the value of pointer field of the preceding node to the pointer field of the newly created node

Program

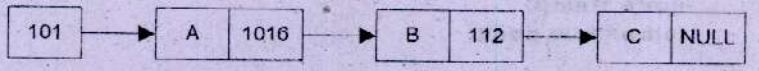
```
// Program to insert a node at any specified location in Linked List
#include<iostream.h>
#include<conio.h>
struct node
   {
    int data;
    node *link;
    };
```

```
class list
    {
    private:
     node *start, *cur;
    public:
     list()
        {
         start = NULL;
        }

     add_item(int n)
         {
         node *temp;
         // create first node & assign data
         if(start == NULL)
          {
           start = new node;
           start->data = n;
           start->link=NULL;
          }
         else
          {
            cur=start;
            // go to end of list
            while(cur->link!=NULL)
                cur=cur->link;
              // create and add new nodes at end
               temp=new node;
               temp->data = n;
               temp->link=NULL;
               cur->link=temp;
          }
        }
// member function to insert node
    insert(int n, int pos)
        {
        // go to specified node
        cur = start;
        for(int i=1;i<pos-1;i++)
           {
             cur=cur->link;
             if(cur == NULL)
              {
               cout<<"Invalid position";
               return 0;
              }
            }
        // create and insert node
        node *temp;
        temp=new node;
```

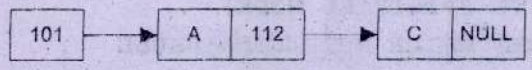
```
        temp->data=n;
        temp->link=cur->link;
        cur->link=temp;
      }

    print()
      {
      cout<<"Data in link list\n" ;
      cur=start;
      while(cur->link!=NULL)
       {
       cout<<cur->data<<endl;
       cur=cur->link;
       }
      // print last node value
      cout<<cur->data<<endl;
      }
  };

main()
{
list obj;
int val,p;
clrscr();
cout<<"Enter five values \n" ;
for(int i=1; i<=5;i++)
{
cin>>val;
obj.add_item(val);
}
cout<<"Enter number to insert ? " ;
cin>>val;
cout<<"Enter position in list ? " ;
cin>>p;
obj.insert(val,p);
obj.print();
getch();
}
```

Deletion Operation

The deletion operation in linked lists is also simple and straightforward. To delete a node from the list, its pointer field value is assigned to the pointer field of its preceding node.



For example, to delete second node from the above list, its pointer field value is assigned to the pointer field of its preceding node, i.e. the first node. After deleting the second node, the linked list will be as shown below:



Algorithm — Deletion

Write an algorithm to delete a specified node in a linked list..

Suppose the starting address of the list is "start" and "data" & "link" are two fields of nodes.

```
1.    DEFINE STRUCTURE OF NODE
2.    SET START = NULL
3.    [Add 5 items into the list]
      REPEAT STEP-4 FOR I= 1 TO 5
4.    INPUT VALUE IN N
      [create nodes and Add items into the link list]
       IF START = NULL THEN
          START = NEW NODE
          START -> DATA = N
          START -> LINK=NULL
       ELSE
          CUR=START
          REPEAT WHILE(CUR->LINK!=NULL)
          [GoTo End Of List]
          CUR=CUR->LINK
          [End of loop]
          [Create And Add New Nodes At End]
                TEMP=NEW NODE;
                TEMP -> DATA = N
                TEMP -> LINK=NULL
                CUR  -> LINK=TEMP
       END IF
```

```
        [End of step-3 loop]

5.      INPUT Value to delete in N
6.      Set CUR = TEMP= START
7.      [GO TO SPECIFIED NODE]
        REPEAT WHILE CUR->LINK!=NULL
        IF CUR -> DATA = N THEN
              PRINT "NUMBER FOUND & DELETED"
              TEMP -> LINK = CUR -> LINK
              DELETE CUR
              RETURN
        END IF
              TEMP=CUR
              CUR = CUR -> LINK
        [End of Step-7 loop]

        IF CUR= NULL THEN
              PRINT "DATA NOT FOUND"
        END IF
8.      EXIT
```

Program

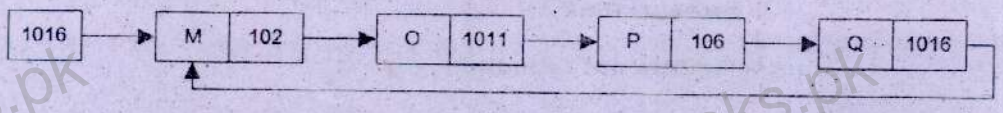
```
// Program to delete a specified node from a Linked List
#include<iostream.h>
#include<conio.h>
struct node
   {
    int data;
    node *link;
    };

class list
   {
   private:
    node *start, *cur,*temp;
   public:
    list()
       {
        start = NULL;
       }
```

```
add_item(int n)
    {
    node *temp;
    // create first node & assign data
    if(start == NULL)
     {
      start = new node;
      start->data = n;
      start->link=NULL;
      }
     else
      {
        cur=start;
        // go to end of list
        while(cur->link!=NULL)
            cur=cur->link;
        // create and add new nodes at end
            temp=new node;
            temp->data = n;
            temp->link=NULL;
            cur->link=temp;
       }
    }

// member function to delete node
    del(int n)
    {
     // go to specified node
     cur = temp= start;
      while(cur->link!=NULL)
        {

          if(cur->data == n)
           {
            cout<<"\nNumber found & deleted \n";
            temp->link=cur->link;
            delete cur;
            break;
            }
           temp=cur;
           cur=cur->link;
          }
      cout<<"\nData not found\n";
    }

// member function to print values
    print()
    {
    cout<<"Data in link list\n" ;
    cur=start;
```

```
            while(cur->link!=NULL)
             {
             cout<<cur->data<<endl;
             cur=cur->link;
             }
            // print last node value
            cout<<cur->data<<endl;
           }
   };
main()
{
list obj;
int val;
clrscr();
cout<<"Enter five values \n" ;
for(int i=1; i<=5;i++)
{
cin>>val;
obj.add_item(val);
}
cout<<"Enter number to search & delete ? " ;
cin>>val;
obj.del(val);
obj.print();
getch();
}
```

CIRCULAR LINKED LISTS

A linked list in which the last node of the list points back to the first node of the list is called circular linked list. It is difficult to detect the end of a circular linked list. To overcome this problem, the end of list is detected with reference to the pointer field of the nodes. If the pointer field value of any node in the list is equal to the starting address of the field, that node is the last node of the list. A circular linked list having four nodes is given below:



The starting address of the list is 1016 and it points to the first node. The node 4 has a value 1016 in its pointer field. It points to the first node. It is the last node of the circular linked list.

The operations on circular linked lists are similar to those of linear linked lists.

Algorithm —Circular linked list

Write an algorithm to create a circular linked list consisting of ten nodes.

```
1.  Define Structure of list nodes
2.  Define pointer variables S, P and C of the list
3.  [Create first node and assign its address to S]
    S = P
4.  REPEAT step-5 to 7 FOR I = 1 to 9
5.  Create new node in C
6.  [Assign address of C in previous node]
    p -> link = C

7.  [Assign address of C to P]
    P = C
    [End of step-5 loop]
8.  [Assign the address of starting list in the
    pointer field of last node, i.e. C -> link]
    C -> link = S
9.  EXIT
```

Double Linked List (Two-Way List)

A linear list in which each node has the addresses both of the next node and of the previous node is called double-linked list. A double-linked list can be visited in both directions, i.e. in forward and backward directions. It is also called two-way list.

The nodes of a linear two-way list consist of at least three fields. These fields are:

- First field to store the information
- Second field to store address of the next node in the list
- Third field to store address of the previous node in the list

A single linked list can be visited in one direction only, i.e. starting from the beginning towards the end of list. A previous node cannot be accessed from the current node. A double-linked list, however, can be visited in both directions. This is the major difference between these two types of linked lists.

Representation of Double-Linked List

The double-linked list is represented in the memory as a data structure having at least three fields. In C++, it is defined as:

```
struct node
    {
        node * previous;
        int item;
        node * next;
    } * start, * pn, * temp;
```

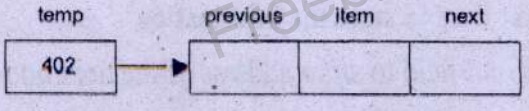
The variables 'start', 'pn' and 'temp' are pointer variables. The purpose of these pointer variables is that:

- 'start' is used to hold the starting address of the list.
- 'pn' is used to hold the address of previous node of the list when a new node is created.
- 'temp' is used to temporarily hold address of the newly created node.

Suppose the first node of list is created by using the new operator as:

```
temp = new node;
```

The node is created in the memory and its address is stored in TEMP. Suppose its memory address is 402.

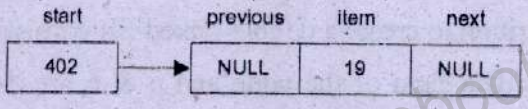


Now assign the value of temp to start:

```
start = temp;
```

The contents of 'start' remains unchanged during program execution. To store values in the above node, statements are written as:

```
start -> previous = NULL;
start -> item = 19;
start -> Next = Null;
pn = temp; //Store 402 in pn, i.e. previous address
```



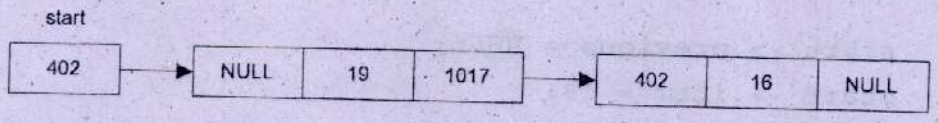
The values in previous and next fields of the first node are NULL. It indicates that the list has only one node. The previous pointer field of first node is always NULL for double-linked lists.

To create a new node and link it to the list as second node, the statements are written as:

```
// Create new node
Temp = new node;

// Assign address of newly created node to its preceding node
pn -> next = temp;

// Store previous node address
Temp -> previous = pn;
```

```
// Assign value 16 to item field
Temp -> item = 16;

Temp -> next = NULL;
```

Suppose the second node with value 16 is created at address 1017. The list having two nodes will be as shown below:



Algorithm — Creating Double-Linked List

Write an algorithm to create a double-linked list consisting of ten nodes.

Suppose X is data field of the node and p & n are its pointer fields for previous and next nodes, respectively.

```
1.   REPEAT Step-2 FOR C = 1 to 10
2.   Temp = new node
          IF C = 1 Then
          Start = temp
          Start -> p = NULL
          Start -> X = Value
          Start -> n = NULL
          pn = Temp
     Else
          pn   -> n = temp
          Temp -> p = pn
          Temp -> X = Value
          Temp -> n = NULL
          pn = Temp
     [End of If structure]
     [End of step-1 loop]

3.   EXIT
```

Program

```
// Program to create a Double-Linked List
#include<iostream.h>
#include<conio.h>
struct node
   {
    node *p;
    int data;
    node *n;
    };

class list
   {
   private:
    node *start, *cur, *temp;
   public:
    list()
       {
        start = NULL;
       }

    add_item(int x)
       {
       // create first node & assign data
       if(start == NULL)
        {
         start = new node;
         start->p = NULL;
         start->data = x;
         start->n=NULL;
        }
       else
        {
          node *prev;
          cur=start;
          // go to end of list
          while(cur->n!=NULL)
              cur=cur->n;
         // create and add new nodes at end
              prev=temp;
              temp=new node;
              temp->p=prev;
              temp->data = x;
              temp->n=NULL;
              cur->n=temp;
        }
      }
```

```
        print()
            {
            cout<<"Data in link list\n";
            cur=start;
            while(cur->n!=NULL)
             {
             cout<<cur->data<<endl;
             cur=cur->n;
             }
 // print last node value
            cout<<cur->data<<endl;
           }
    };

 main()
 {
 list obj;
 int val;
 clrscr();
 cout<<"Enter five values \n" ;
 for(int i=1; i<=5;i++)
 {
 cin>>val;
 obj.add_item(val);
 }
 obj.print();
 getch();
 }
```

Double-Linked List Operations

Like single-linked lists, the following operations can be performed on the double linked list.

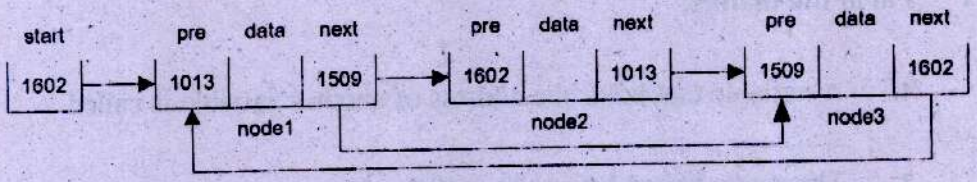
- Traversing
- Deletion
- Sorting
- Insertion
- Searching

Circular Double-Linked Lists

A circular double-linked list is similar to a circular single-linked list. In it, the next pointer field of the last node is linked to the first node of the list and the previous pointer field of first node contains the address of last node.

Consider a three node circular double-linked list created with nodes *node1*, *node2*, and *node3* having address 1602, 1509 and 1013, respectively. The

pointer field *pre* points to the previous node, *next* points to the next node and *data* field stores value of the item. This list is shown graphically in the following figure:



EXERCISES

Q.1 Fill in the blanks.

1. A variable that holds the address of another variable is called _____.
2. The single linked list is also called _____.
3. The linked list that has no node is called _____.
4. The field in a node of linked list that holds the address of next node is usually called _____.
5. The value in the link field of the last node is _____.
6. In C++, the operator _____ is used to create a new node in linked list.
7. The linked list in which the last node points back to the first node of list is called _____.
8. A linked list in which each node of the list contains the address of next node as well as the address of previous node is called _____.
9. The linked list that can be visited in both directions is called _____.

Q.2 Mark True or False.

1. Linear arrays are used to store data of linked lists.
2. The linked list in which items of the list are visited only in one direction is called one-way linked list.
3. The linked list that has only one item is called single linked list.
4. The linked list that has two items is called double-linked list.
5. The linked list that has no node is called empty list.

6. Only numeric data can be stored in the data field of nodes in linked lists.
7. The link field of the last node of single linked list contains NULL value.
8. The linked list in which items of the list are visited in both directions is called two-way linked list.
9. The items of the linked list are stored in consecutive memory locations.
10. No new items into the linked list can be inserted.
11. The searching operation cannot be performed on linked lists.
12. In circular single linked list, the first and the last linked fields have the same value.
13. A large amount of consecutive memory is required to store same type of data by using linked list technique.
14. The nodes of a linked list may be of different size and data type.
15. The double linked list contains two linked fields for each node.
16. The double linked list may be visited in both directions.
17. The circular double linked list contains only one link field for each node.
18. The double linked list is also called two-way linked list.
19. The circular single linked list may be visited in both directions.

Q.4 **Write an algorithm to print odd numbers stored in a linked list consisting of ten nodes.**

Q.5 **Write an algorithm to visit the double linked list consisting of ten nodes in reverse order.**

Q.6 **Write a program to insert a new value in a two-way list at a specified location.**

Q.7 **Write a program to delete a node from a two-way list at a specified location.**

Q.8 Write a program to modify a value from a single linked list after searching it.

Q.9 Write a program to visit a single linked list consisting of five nodes and having a data field of integer type. Calculate the factorial of each value stored in it. Create a new linked list and store the factorials in the newly created linked list. Also print both the linked lists.

Q.10 Write a program to enter 10 records of a student using linked list. Use the following structure of the nodes of the linked list:

```
struct node
{
  char name[15];
  char address[25];
   int age;
   int phone;
   node *next;
};
```

Q.11 Differentiate between the following:

a. Two-way list & Circular Two-way list

b. Two-way list & one-way list

TREES

Trees

A tree is a non-linear data structure. Each object of a tree starts with a trunk or root and extends into several branches. Each branch may extend into other branches until finally terminated by a leaf. Trees are common structures in everyday life. A common example of trees is the lineage of a person that is referred to as the family tree. Organizational chart of a company is another example of trees.

In computer applications, a tree is an hierarchical collection of finite elements. Each element is called a node. A typical tree is shown below:

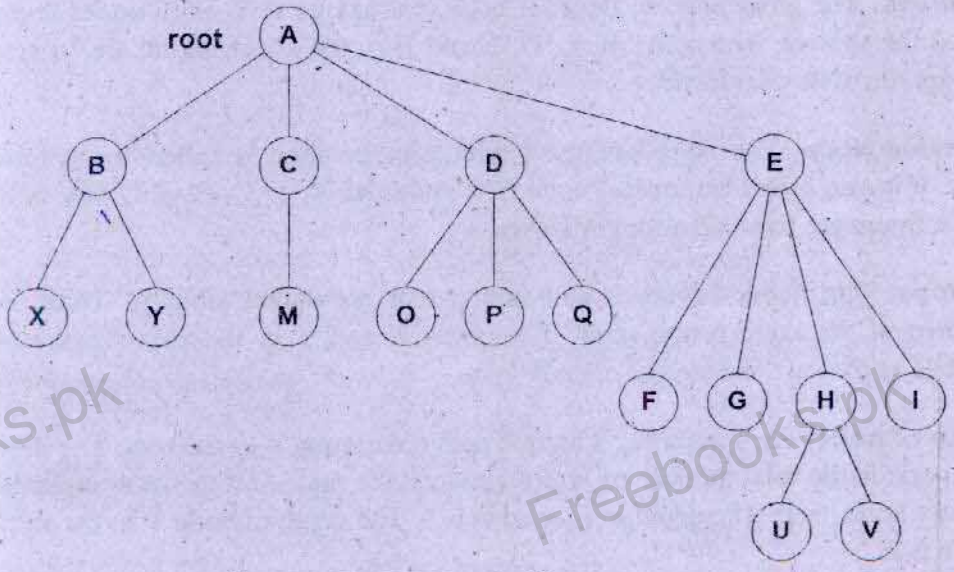


Fig 8.1: General Tree

The trees have the following properties:

- Each tree has one node designated as the root of the tree.
- A unique path exists from the root node to all other nodes in the tree.
- Each node, other that the root, has a unique predecessor. It is called the parent.
- Each node may have no, one or several successor nodes. The successor node is called the child.

Some basic terms about tree structures are defined below:

Root node: The unique node in the tree that has no parent node is called the root node or root of the tree. It represents the beginning of the tree. Element A in above figure is the root node of the tree.

Parent node: The node, which generates links for other nodes, is known as the parent node. The parent node is always above its child nodes. The node A in the above figure is the root. It is also the parent node of nodes B, C, D and E.

Child node: The node that is directly connected to a parent node is called the child node. The nodes B, C, D and E are the Child nodes of A.

Subtrees: The child node of the root node that has its own child nodes is also called the subtree. The nodes B, C, D, E and H in the above figure are subtrees and are the roots of subtrees.

Terminal Node: The node having no successor or child is called the terminal node. It is also called leaf or leaf node. The nodes M, X, Y, O, P, Q, U & V in the above figure are terminal nodes or leaves.

Siblings: The nodes having a common parent are called siblings. These are children of the same parent node. The nodes X and Y in the above figure are siblings as they are children of node B.

Depth of node: Each node has a unique path connecting it to the root. The depth of a node is the total number of nodes between the node and the root, including the root node itself. The depth of root node is 0. The depth of node Y in the above figure is 2.

Height of Tree: The maximum depth among all of the nodes of a tree is called the

height of tree. The depth of each node U & V in the above figure is 3. It is the maximum height of any node in the tree. Thus, the height of tree in the above figure is 3.

Empty Tree: A tree that has no node is called empty Tree. Its height is –1.

Degree of node: The number of children of a node is called degree of the node. In the above figure, the degree of node C is one and the degree of node D is three.

Full Tree: A tree is called the full tree if all its internal nodes have the same degree and all the leaves are at the same level. The tree in the figure 8.2 is an example of a full tree.

Singleton Tree: The tree whose root is its only node is called the singleton tree.

A tree in which a node may have no child, one child or any number of children is called general tree. It is difficult to implement and handle a general tree in the computer memory. Therefore, a special kind of tree known as the binary tree is used which can be easily maintained in the computer.

BINARY TREE

A binary tree is a hierarchical collection of a finite number of nodes in which each node can have a maximum of two children. One child is on the left side and the other is on the right side of the tree or subtree. A binary tree may be empty or non-empty but a general tree cannot be empty.

A non-empty binary tree may have the following three parts:

Root node: The node of the binary tree that has no parent is called the root node. It is the starting node of any tree.

Left Child node: The node that lies on the left side of the root of binary tree (or root of subtree) is referred to as left child node. The node B in figure 8.2 is left child node of A.

Right Child node: The node that lies on the right side of root of binary tree (or root of subtree) is referred to as right child node. The node D in figure 8.2 is the right child node of A.

Both the child nodes of the root of a binary tree (or subtree) may be empty or either one of the child nodes may be empty.

An example of Binary Tree is given in the following figure 8.2.

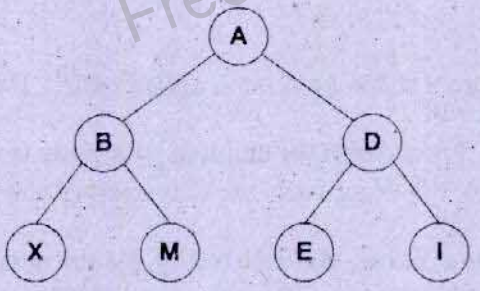


Figure 8.2: Binary Tree

In the above figure, A is root of the binary tree and B & D are left and right children of the root A, respectively. The left child B and right child D also have their own child nodes.

Binary trees are a very useful data structure. They are used to keep data that needs to be searched quickly, or that needs to maintain a sorted order with lots of insertions or deletions.

Full Binary Tree

A binary tree is said to be a full binary tree when:

- its all leaves are at same level.
- every node has two children, i.e. left child & right child.

A typical full binary tree is shown in figure 8.3.

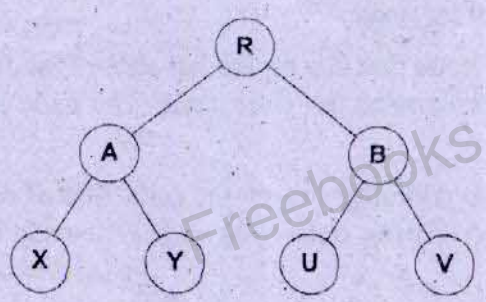


Figure 8.3: Full Binary Tree

Extended Binary Tree (2–Tree)

A binary tree in which each node has either no child or two children is called an extended binary tree. It is also called 2–Tree. The node that has two children is called internal node and the node that has no child is called external node. A typical extended binary tree is shown in the following figure 8.4.

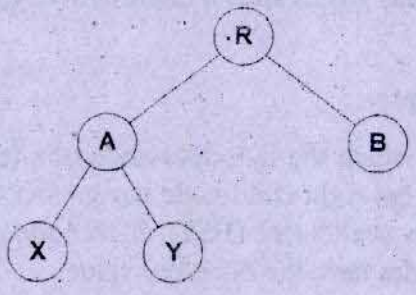


Figure 8.4: Extended Binary Tree

In the above figure of an extended binary tree, the root node R and node A both have two children each but the nodes, B, X & Y have no children.

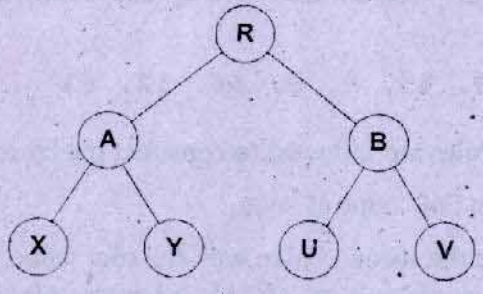


Figure 8.5: Complete Binary Tree

Complete Binary Tree

A binary tree is said to be a complete binary tree of height H if:

- all nodes at level H–2 and above have two children each.
- a node at level H–1 has two children and all its left siblings have two children each.
- a node at level H–1 has one child, it is a left child and all left siblings of the node have two children each.

The complete binary tree is shown in the above figure 8.5.

Binary Search Tree (BST)

A binary tree in which the left child node of a tree or subtree has lesser value than its root node but right child node has greater (or equal) value than its root node is called binary search tree (BST). Each node in the BST is assigned a key value and no two nodes have the same key value.

When a binary search tree is traversed inorder (i.e., left, root, right) and the number of each node is printed then they are printed in ascending order. The main advantage of binary search trees is that it is easy to perform operations like creating new tree, insertion, searching and deletion on a binary search.

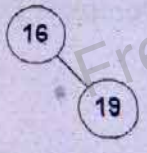
Constructing Binary Search Tree

Suppose a binary search tree having the following values is to be constructed:

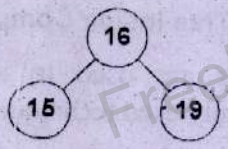
```
16, 19, 15, 9, 8, 66, 12, 61
```

The following rules are followed to construct the binary search tree:

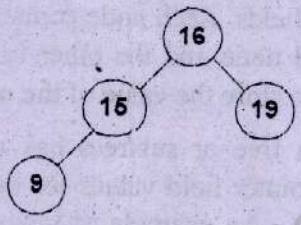
1. place the first value as root.
2. compare the second value with the root value, if it is less than root value then place it on left side otherwise place it on the right side. Since 19 is greater than 16, place it on the right side.



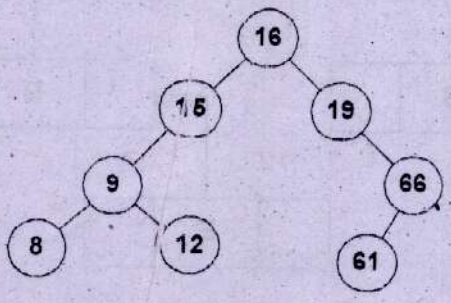
3. Repeat step–2 for all values starting from root. Thus, the third value is 15. It is less than 16. So place it on left side of the root.



4. Compare fourth value with the root value, if it is less than the root value then compare it with the left subtree otherwise compare it with right subtree and follow the same rule as mentioned at step–2. Since 9 is less than 16, compare it with the left subtree. It is also less than the left subtree (i.e. 9 is less than 15), so it will be placed on its left side.



5. By following the above procedure, the binary tree for the given values is created as shown below.



It must be noted that dup lication of values in binary search trees is not allowed.

Representation of Binary Tree Inside Computer

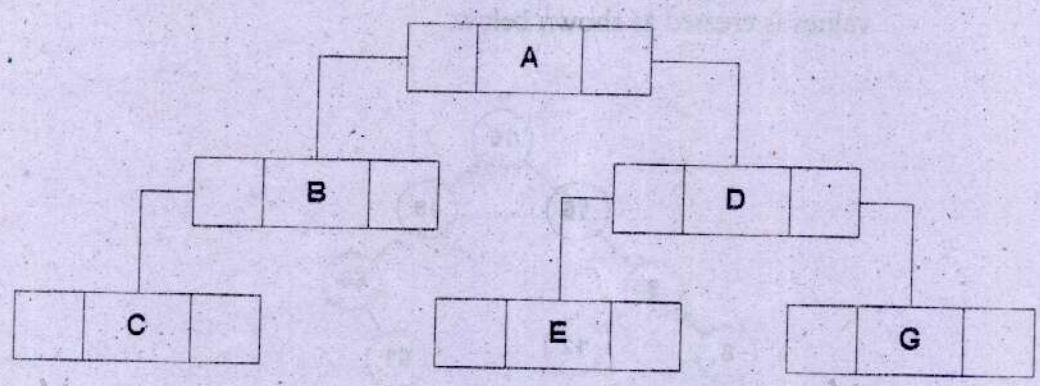
Different techniques are used to represent binary trees inside the computer. The most commonly used techniques are:

- **Linked storage technique.**
- **Sequential storage technique.**

Linked Storage Technique

It is the most commonly used method for representing binary trees inside the computer memory. In this technique, each node of the binary tree is represented by a set of three fields. Each node contains two pointer fields, one to hold the address of left child node and the other to hold address of right child node. The third field is used to store the value of the node.

If the root node of a tree or subtrees has no child then pointer fields contain NULL values. The pointer field values for both left and right children of leave nodes are always NULL. An example of linked representation of a binary tree is given in the following figure.



The value in the root is "A". Similarly, the left and right children nodes of the root have values B & D, respectively.

To represent a binary tree in the memory, a node structure of the tree is defined as given below:

```
struct node
    {
        int data;
        node * left;
        node * right;
    };
```

This representation is similar to a two-way linked list. Many of the rules that apply to the linked lists also apply to represent binary trees inside the computer. For example, a node of the tree is created in a similar way as that of a linked list by using the "new" operator. Similarly, a node is removed by using the "delete" operator.

Sequential Storage Technique

The sequential allocation is a simple technique to store the tree inside the computer. In this technique a linear array is used to represent the tree inside the computer. This technique is efficient if structure of the tree does not change very much during its existence. It is best used to represent full binary trees. The following rules are used to represent the binary tree using sequential storage:

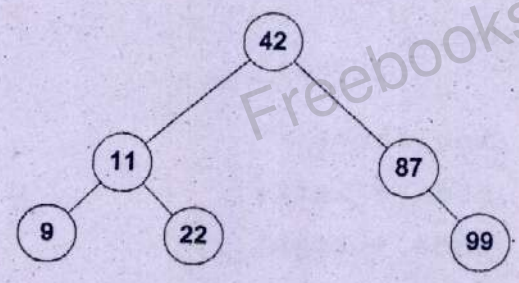
1. Root R is stored in first element of the array.
2. The left child of the node K is stored in element 2*K of the array.
3. The right child of the node K is stored in element 2*K+1 of the array.

K represents the number of node of the tree. A NULL value indicates that the tree or subtree is empty.

The following figure shows sequential representation in array of a binary tree. The tree has the values:

```
42, 11, 87, 9, 22, 99
```

The binary tree and its sequential representation is shown below:



Position	1	2	3	4	5	6	7
Elements	42	11	87	9	22		99

Fig: Sequential Representation of Binary Tree

The linked storage technique is recommended to represent the complete or full binary trees. If the tree is not full, a lot of elements in the array remain empty. Approximately, sequential representation of a binary tree of depth h needs an array of approximately 2^{h+1} elements. Thus for a tree that has only 11 nodes and depth 5, the array size required to represent the tree is approximately $2^6 = 64$ elements.

Operations on Binary Search Trees

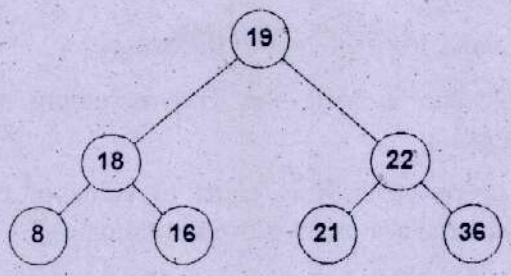
Like other data structures, different operations can be performed on the binary trees also. The most common of these operations are:

- **Inserting**
- **Searching**
- **Traversing**
- **Removing**

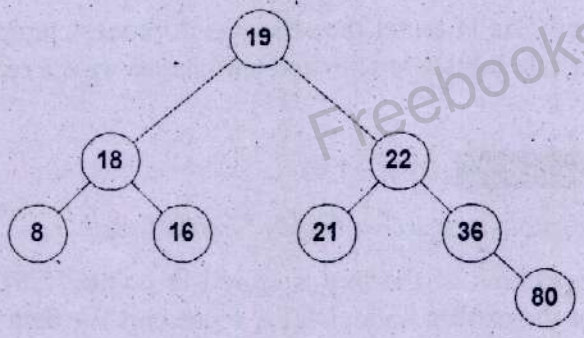
Insertion Operation

A new node is inserted into the binary search tree after searching other nodes. If the new value already exists in any node then the insertion operation fails otherwise the new node is inserted at the point the search terminates.

Suppose a new value N is to be inserted into the BST shown in the figure below. To insert the value 80, first this value is searched in the tree. If the value is not found then it is inserted at the point the search terminates.



The BST after insertion will be as shown below:



Algorithm — Insertion

Write an algorithm to insert a new node in a Binary Search Tree.

```
1.  INPUT value to insert in N
2.  Search Value N in the Tree
3.  IF value N exists then print "Number already
    exists" and return. Otherwise insert the value at
    the point the search stops.
```

Searching Operation

Recursive search method is used to search the binary search trees. It is the easiest way to search a binary tree. The following steps are taken to search a given value N in the binary search tree using recursive search:

1. Search starts from the root of the tree.
2. If root value is Null, the Tree is empty and search process terminates.
3. If the given value N is equal to value of root then search is successful and searching process terminates.
4. If given value N is less than the value of root, only the left subtree is searched recursively.
5. If given value N is greater than the value of root, only the right subtree is searched recursively.
6. If the value is found then the search process terminates otherwise the whole subtree is searched until its leaves are reached.

Algorithm — Searching BST

Write an algorithm to search a Binary Search Tree.

The starting address of the tree is stored in pointer BST. The pointer P stores the address of the current node. DATA represents the data field and L & R represent left and right pointers of a node of the tree. Value N represents the value that is to be searched.

```
1.    Set P = BST, Set Check =0
2.    IF N = NULL THEN
         PRINT "Tree is Empty "
         RETURN
      END IF
3.    IF N = P->DATA THEN
            PRINT "Number is found "
            Check =1
            RETURN
      ELSE
```

```
        PERFORM STEP-4
    END IF

4.  REPEAT STEP-5 WHILE P!= NULL
5.  IF N < P->DATA THEN
        P = P -> L
        ELSE
        P = P -> R
    END IF
        RETURN TO STEP-3
    [END OF STEP-4 LOOP]

    If Check =0 THEN PRINT "Number not found"
6.  RETURN
```

Program

```
// program to create a binary tree and search a given value in it

#include<iostream.h>
#include<conio.h>
struct node
   {
       int data;
       node *left;
       node *right;
    };

class tree
   {
    private:
      node *start, *cur, *temp;
    public:
      tree()
        {
          start = NULL;
        }
    void create(int x);
    void search(int x);

};

main()
```

```
{
tree obj;
int val,s, opt, c=1;
while(c)
{
 clrscr();
 cout<<"1: Create new Binary Tree \n" ;
 cout<<"2: Search Value from Binary Tree \n" ;
 cout<<"3: Exit \n\n" ;
 cout<<"Select your Choice [1-3] ?" ; cin>>opt;
 switch(opt)
  {
   case 1:
        cout<<"Enter ten  values \n" ;
        for(int i=1; i<=10;i++)
        {
        cin>>val;
        obj.create(val);
        }
        break;
   case 2:
        cout<<"Enter value to search ?" ;
        cin>>s;
        obj.search(s);
        break;
   case 3: c=0; break;
  }
 }
}

void tree::create(int x)
 {

// create root node & assign data
 if(start == NULL)
   {
    start = new node;
    start->data = x;
    start->left  = NULL;
    start->right = NULL;
    }
  else
   { node *parent;
      cur=start;

// go to proper position to insert node
     while(cur!=NULL)
          {
           if(cur->data == x)
            {
```

```
            cout<<"Data already exist";
            return;
            }
           if(cur->data < x )
            { parent=cur; cur=cur->right;}
          else
            { parent =cur; cur=cur->left;}
         }

// create and add new node
        temp=new node;
        temp->data = x;
        temp->left  = NULL;
        temp->right = NULL;
        if (parent == NULL)
           parent=temp;
        else if(parent->data>x)
              parent->left=temp;
            else
              parent->right = temp;
    }
}

void tree::search(int x)
 {
  if(start == NULL)
  {
   cout <<"Tree is empty ";
   getch();
   return;
  }
 else
  {   cur=start;

// search value
    while(cur!=NULL)
        {
         if(cur->data == x)
          {
          cout <<"Data found ";
          getch();
          return;
          }
         if(cur->data < x )
          cur=cur->right;
        else
          cur=cur->left;
        }

       if(cur == NULL)
```

```
            cout <<"Value not found ";
        getch();
    }
}
```

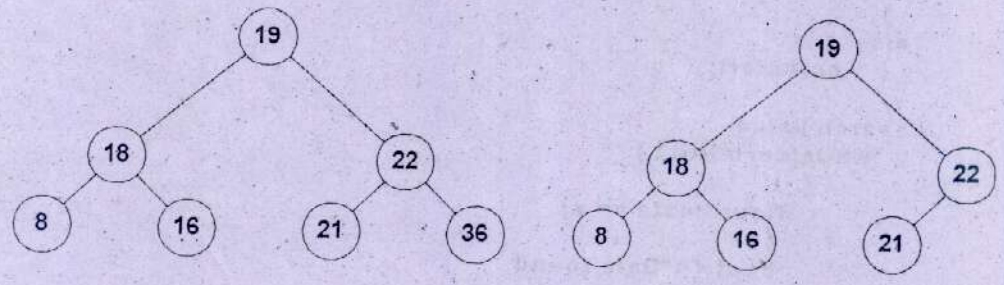
Deletion Operation

The nodes of a binary search tree can also be easily deleted. The value of the node that is to be deleted is first searched in the binary search tree and if found, it is deleted. The deletion operation of leaf nodes and non-leaf nodes that have only one child is easy. But if the node that is to be deleted is a non-leaf node and has two children, the procedure becomes complicated.

Deleting Leaf Nodes

The leaf node in a tree is deleted by simply assigning NULL value to the pointer of its parent node. For example, if the leaf node with value 36 is to be deleted from the tree shown below, NULL value is assigned to the right pointer of its parent node. The leaf node is disconnected from the tree and is automatically deleted.

The binary search trees before and after deletion are shown below:



Before Deletion

After Deletion

Algorithm – Deletion Leaf Node in BST

Write an algorithm for deleting a leaf node from a binary search tree.

```
1.  INPUT value to delete in N
2.  Find the Value N in the tree
3.  IF value N is found and is leaf node then remove
    it otherwise return and stop the operation
```

Traversing Operation

In traversing operation each node of a binary tree is accessed for processing exactly once in a systematic manner. It is also called visiting. Different methods are used to visit a binary tree. The important methods include:

- **Preorder Traversal**
- **Inorder Traversal**
- **Postorder Traversal**

Preorder Traversal

Root, Left, Right

In preorder traversing, first the root is visited and then its left & right children are visited. The order of traversing is Root–Left–Right. A stack is used to implement the preorder traversal and a pointer variable is used to hold location of the node being currently visited.

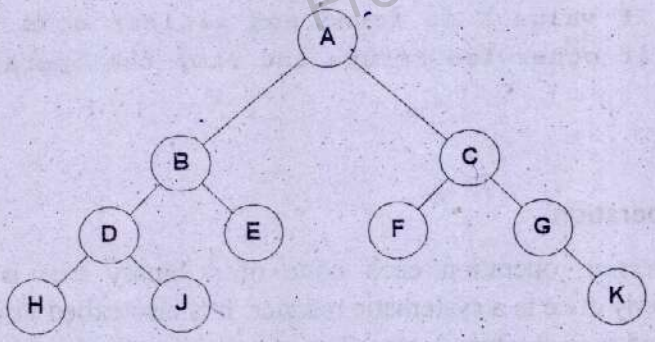
To traverse a binary tree in preorder sequence:

1. Root node is visited. The address of its right child, if any, is pushed onto the stack.
2. The left subtree of the node is visited. The address of its right child, if any, is pushed onto the stack.

   Similarly, all subsequent left subtrees are visited up to the last leftmost child.
3. After visiting the last leftmost child, the top value from the stack is popped and visited in preorder sequence as explained above.

The preorder traversal of the following binary tree gives the following sequence:

A B D H J E C F G K



Algorithm – Preorder Traversal of BST

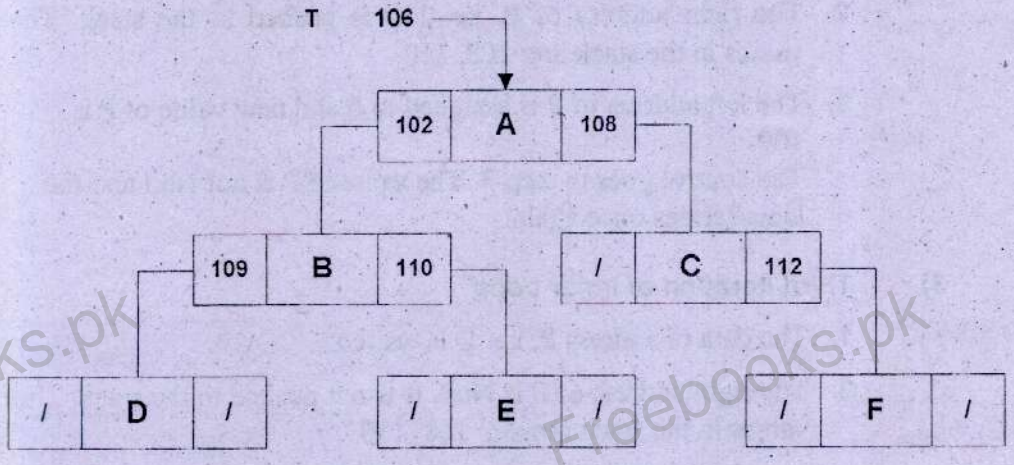
Write an algorithm for preorder traversal of binary search tree.

Consider a binary tree whose root node address is stored in pointer variable T and pointer variable P holds the address of current node during processing. STK denotes the stack and Top denotes the top position of the stack STK. The function PUSH is used to insert the value in the stack STK and POP is used to remove the value from top of the stack. L and R represent pointer variables that hold addresses of left & right children.

```
1.   [Read starting address of BST]
     IF T = NULL THEN
          PRINT "Binary Tree is Empty"
          RETURN
     ELSE [Place pointer to root of Tree on top of stack STK]
          TOP = 0
          CALL PUSH (STK, TOP, T)
     [END IF]
2.   REPEAT STEP-3 TO 4 WHILE TOP>0
```

```
3.   [Pop top value of stack STK into P to visit Tree in preorder]
     P = POP(STK, TOP)
4.   [Visit the left most nodes of the tree or subtree and
     assign addresses of each right node to top of the stack]
     REPEAT WHILE P!=NULL
     PRINT P->DATA [Print data of node]
          IF P → R != NULL THEN
          CALL PUSH (STK, TOP, P->R)
     END IF [Insert the right node address to stack]
     P = P->L   [Point to left branch]
     [End of Step-4 Loop]
     [End of Step-2 Loop]
     RETURN
```

Example: Explain the preorder traversal using a stack of the binary tree shown in the following figure. The addresses of its left & right children are given along with node values. T represents starting address of the tree with value 106.



Suppose, P represents the current position of pointer through which tree is visited.

Step-1: Starting Address of the Tree (i.e. 106) is pushed to the stack. The top value of stack =106.

Step-2: Execute Outer Loop WHILE Top >0

OUTER Loop Iteration 1

Top value of the stack is popped and value is assigned to P. Thus, P = 106.

Step-3: Inner Loop While P != Null

1) First Iteration of Inner Loop

1. Data of address P, i.e. A is visited.
2. The right address of P, i.e. 108 is pushed to the stack. The top value of the stack is 108.
3. The left address of P is assigned to P and new value of P is 102. The control goes to step-3. The value of P is not Null and the loop iterates once again.

2) Second Iteration of Inner Loop

1. Data of address P, i.e. B is visited.
2. The right address of P, i.e. 110 is pushed to the stack. The values in the stack are: 108, 110.
3. The left address of P is assigned to P and new value of P is 109.
   The control goes to step-3. The value of P is not Null and the loop iterates once again.

3) Third Iteration of Inner Loop

1. The data of address P, i.e. D is visited.
2. The right address of P is Null. It is not pushed to the stack. The values in the stack remain: 108, 110.
3. The left address of P is assigned to P and new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

OUTER Loop Iteration 2:

The control shifts back to step-2. As the value in the stack is not NULL, the loop continues and the value of the stack, i.e. 110 is popped and assigned to P. Loop at step-3 is again executed as:

1) First Iteration of Inner Loop

1. The data of address P, i.e. E is visited.
2. The right address of P is Null. It is not pushed to the stack. The value of the stack remains: 108.
3. The left address of P is assigned to P and new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

OUTER Loop Iteration 3:

The control shifts back to step-2. As the value in the stack is not NULL, the loop continues and the value of the stack, i.e. 108 is popped and assigned to P. Loop at step-3 is again executed as:

1) First Iteration of Inner Loop

1. The data of address P, i.e. C is visited.
2. The right address of P, i.e. 112 is pushed to the stack. The top value of the stack is 112.
3. The left address of P is assigned to P and new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

OUTER Loop Iteration 4:

The control shifts back to step-2 as the value of stack is not NULL. The loop continues and the value of the stack, i.e. 112 is popped and assigned to P. Loop at step-3 is again executed as:

1) First Iteration of Inner Loop

1. The data of address P, i.e. F is visited.
2. The right address of P, i.e. NULL is pushed to the stack. The top value of the stack is NULL.

3. The left address of P is assigned to P and new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

The control shifts back to step-2. As the value of stack is NULL, the loop condition is satisfied. The loop terminates.

Program

```
// Program to create and visit binary search tree inorder recursively
#include<iostream.h>
#include<conio.h>
struct node
    {
     int data;
     node *left;
     node *right;
     };

class tree
    {
    private:
     node *start, *cur, *temp;
     int top;
    public:
     tree()
        {
          start = NULL;
        }

     void create(int);
     void preorder(void);
     void preord(node *s);
  };

main()
{
tree obj;
int val,p;
clrscr();
cout<<"Enter ten values \n" ;
for(int i=1; i<=10;i++)
{
cin>>val;
obj.create(val);
}
```

```
 obj.preorder();
 getch();
 }

 // member function to call the recursive function preord
 void tree::preorder()
 {
  preord(start);

 }
 // preord recursive function to visit tree in preorder traversal
 void tree::preord( node *s)
  {
   if(s!=NULL)
    {
    cout<<s->data<<"\t";
    preord(s->left);
    preord(s->right);
    }
 }
 // member function to create and assign data into tree nodes
 void tree::create(int x)
 {
  if(start==NULL)
    {
    // create root node & assign data
    start = new node;
    start->data = x;
    start->left = NULL;
    start->right = NULL;
    }
  else
   {
     node *parent;
     cur = start;
     // search value in tree & go to proper position to insert
     while(cur!=NULL)
      {
          if(cur->data == x)
           { cout<<"Data already exist\n"; return;}
          if(x > cur->data)
           { parent=cur; cur=cur->right;}
          else
           { parent = cur; cur=cur->left;}
      }

     // create and insert new node
     temp = new node;
     temp->data = x;
     temp->left = NULL;
```

```
        temp->right = NULL;
        if(parent == NULL)
            parent = temp;
        else if( parent->data > x)
                parent->left=temp;
              else
                parent->right=temp;
    }
   }
```

Program

```
//Program to create & visit binary search tree in preorder using stack

#include<iostream.h>
#include<conio.h>
struct node
    {
     int data;
     node *left;
     node *right;
     };

class tree
    {
    private:
     node *start, *cur, *temp;
    public:
     tree()
        {
          start = NULL;
        }

     void create(int x);
     void preorder(void);
   };

main()
{
tree obj;
int val,p;
clrscr();
cout<<"Enter ten values \n" ;
for(int i=1; i<=10;i++)
{
cin>>val;
```

```
obj.create(val);
}
obj.preorder();
getch();
}

// member function to print the data of tree in preorder
void tree::preorder()
{
 int top = 0;
 cout<<"\nPrint data in Preorder\n\n";
 node *stack[10];       // stack to store right node address
 stack[top]=start;
 while(top>=0)
  {
   cur = stack[top];  // pop right node address
   top--;
   while(cur!=NULL)
    {
     // print data in preorder
     cout<<cur->data<<"\t";
     if(cur->right!=NULL)
      {
       top++;
       stack[top]= cur->right;// push right node address
      }
     cur = cur->left;
    }
  }
}

// member function to create and assign data into tree nodes
void tree::create(int x)
{
 if(start==NULL)
  {

// create root node & assign data
   start = new node;
   start->data = x;
   start->left = NULL;
   start->right = NULL;
  }
 else
  {
   node *parent;
   cur = start;

// search value in tree & go to proper position to insert
   while(cur!=NULL)
```

```
        {
            if(cur->data == x)
            { cout<<"Data already exist\n"; return;}
            if(x > cur->data)
            { parent=cur; cur=cur->right;}
            else
            { parent = cur; cur=cur->left;}
        }

// create and insert new node
        temp = new node;
        temp->data = x;
        temp->left = NULL;
        temp->right = NULL;
        if(parent == NULL)
            parent = temp;
        else if( parent->data > x)
                parent->left=temp;
             else
                parent->right=temp;
    }
}
```

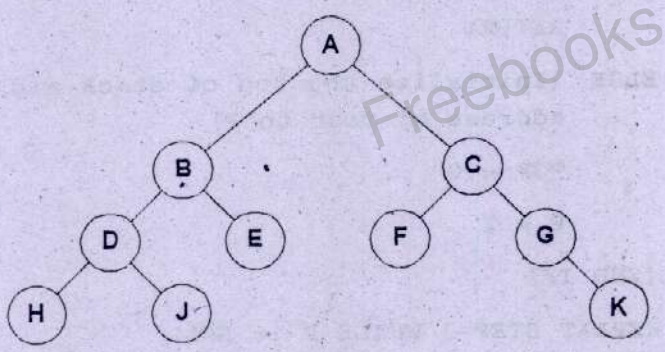
Inorder Traversal

Left Root Right

In inorder traversing, first the left child is visited and then root & right child are visited. The sequence of traversing is Left–Root–Right. When a binary search tree is visited inorder, data of the tree is displayed in ascending order.

A stack is used to implement inorder traversal of binary trees. A pointer variable is also required to hold the location of the node being currently traversed. To visit a binary search tree in inorder:

1. Proceed down the left-most path from the root node to the last left node. Push addresses of each left node into the stack.
2. Pop the top address from the stack. It will be the address of the leftmost node. Visit the node.
3. If the node being visited has a right child, then repeat steps 1 & 2 (i.e. the addresses of each left node are pushed into the stack up to its left most node).
4. The above procedure is repeated till all the tree is traversed.



Inorder traversal of the above binary tree gives the following sequence:

```
H D J B E A F C G K
```

Algorithm – Inorder Traversal of Binary Tree

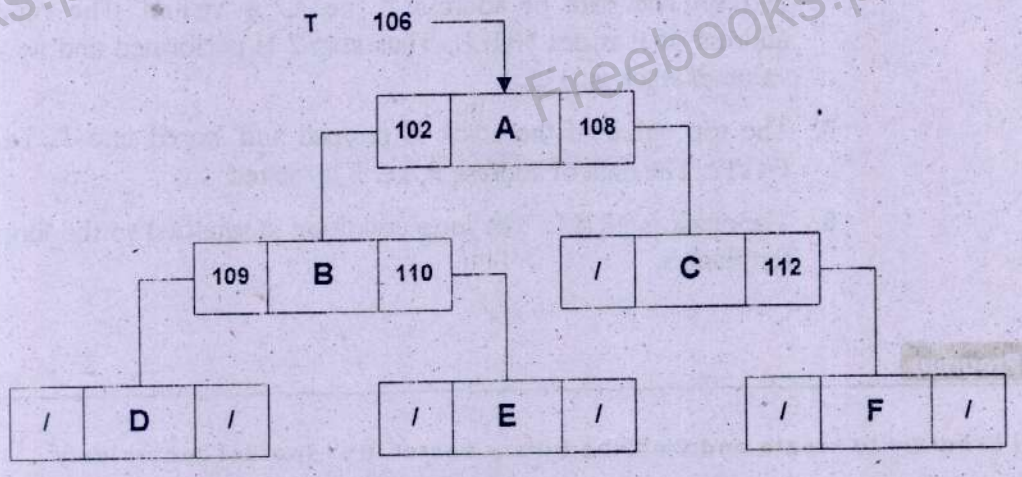
Write an algorithm for inorder traversing of a binary tree.

- Suppose the root node address of the binary tree is stored in pointer variable T.
- Another pointer variable P holds address of the current node during processing.
- STK denotes the stack.
- TOP denotes the top position of stack STK.
- PUSH function is used to insert a value into the stack.
- POP function is used to get or remove the value from top of the stack.
- L and R represent pointer variables to hold addresses of left & right children.

```
1.    [Read starting address of Tree]
      IF T = NULL THEN
            PRINT "Binary Tree is Empty"
```

```
        RETURN
    ELSE [Initialize the top of stack and assign the
         address of root to P]
        TOP = 0
        P = T
    [END IF]
2.  REPEAT STEP-3 WHILE P != NULL
    [Proceed to left most node by pushing left node to
    stack]
3.  CALL PUSH(STK, TOP, P)
    P = P -> L
    [End of Step-2 loop]
4.  REPEAT STEP-5 to 7 WHILE TOP > 0
    [Pop top value of stack STK into P to visit the
    Tree Inorder]
5.  P = POP(STK, TOP)
6.  PRINT P -> Data
7.  IF P -> R != NULL THEN
        P=P -> R
        PERFORM STEP-2
    END IF
    [End of Step-4 Loop]
8.  RETURN
```

Example: A Binary Tree is shown below. The values and addresses of its left & right children are also shown. T represents the starting address of the tree and has a value of 106. P represents the current position of pointer through which the tree is traversed. Explain the steps taken in inorder traversal of the tree using a stack.



Step-1: Starting address of the tree is assigned to P, thus P = 106.

Step-2: Loop While P != NULL

Starting from the root node, proceed down the left-most path to the last left node while pushing addresses of each left node into the stack.

At the end of the loop, the value in the stack will be: 106,102;109 and P = NULL

Step-3: Loop While TOP != NULL

1. The top value of stack is popped and stored into P, i.e. P = 109. The data of address P, i.e. D is visited.
2. The top value of the stack is popped and stored into P, i.e. P=102. The data of address P, i.e. B is visited.
3. The right address of P is not NULL. Thus loop at step-2 is executed. The values in the stack will be: 106, 110.
4. The top value of stack is popped and stored into P, i.e. P=110. The data of address P, i.e. E is visited.
5. Top value of stack is popped and stored into P, i.e. P=106. The data of address P, i.e. A is visited. The right address of P is not NULL. Thus step-2 is performed and value in the stack is: 108.
6. The top value of the stack is popped and stored into P, i.e.

P=108. The data of address P, i.e. C is visited. The right address of P is not NULL. Thus step-2 is performed and new value of stack is: 112.

7. The top value of the stack is popped and stored into P, i.e. P=112. The data of address P, i.e. F is visited.
8. The stack is NULL. The loop condition is satisfied so the loop terminates.

Program

```
// Program to create and visit the binary search tree inorder recursively

#include<iostream.h>
#include<conio.h>
struct node
   {
    int data;
    node *left;
    node *right;
    };

class tree
   {
   private:
    node *start, *cur, *temp;
    int top;
   public:
    tree()
       {
        start = NULL;
       }

    void create(int);
    void inorder(void);
    void inord(node *s);
  };

main()
{
tree obj;
int val,p;
clrscr();
cout<<"Enter ten values \n" ;
for(int i=1; i<=10;i++)
{
```

```
cin>>val;
obj.create(val);
}
obj.inorder();
getch();
}

// member function to call the recursive function inord
void tree::inorder()
{
 inord(start);

}
// Inord recursive function
void tree::inord( node *s)
 {
  if(s!=NULL)
   {
   inord(s->left);
   cout<<s->data<<"\t";
   inord(s->right);
   }
}
// member function to create and assign data into tree nodes
void tree::create(int x)
{
 if(start==NULL)
  {
  // create root node & assign data
  start = new node;
  start->data = x;
  start->left = NULL;
  start->right = NULL;
  }
 else
  {
    node *parent;
    cur = start;
    // search value in tree & go to proper position to insert
    while(cur!=NULL)
     {
        if(cur->data == x)
         { cout<<"Data already exist\n"; return;}
        if(x > cur->data)
         { parent=cur; cur=cur->right;}
        else
         { parent = cur; cur=cur->left;}
     }

   // create and insert new node
```

```
        temp = new node;
        temp->data = x;
        temp->left = NULL;
        temp->right = NULL;
        if(parent == NULL)
            parent = temp;
        else if( parent->data > x)
                parent->left=temp;
              else
                parent->right=temp;
    }
}
```

Program

```
// Program to create and visit binary search tree inorder using stack

#include<iostream.h>
#include<conio.h>
struct node
    {
     int data;
     node *left;
     node *right;
     };

class tree
    {
    private:
     node *start, *cur, *temp, *stack[10];
     int top;
    public:
     tree()
        {
         start = NULL;
         top=-1;
        }

     void create(int);
     void inorder(void);
     void push(node *s);
    },

main()
{
tree obj;
```

```
int val,p;
clrscr();
cout<<"Enter ten values \n" ;
for(int i=1; i<=10;i++)
{
cin>>val;
obj.create(val);
}
obj.inorder();
getch();
}

// member function to print the data of tree in order
void tree::inorder()
{
 cur = start;
 push(cur);
 while(top>=0)
  {
   cur = stack[top];  // pop address
   top--;
   // print data in inorder
     cout<<cur->data<<"\t";
      if(cur->right!=NULL)
          push(cur->right);
  }
}
// member function to push the left most node of parent to stack
void tree::push( node *s)
{
while(s!=NULL)    // go to left most node
   {
   top++;
   stack[top]=s;  // push parent node address
   s=s->left;
   }
}

// member function to create and assign data into tree nodes
void tree::create(int x)
{
if(start==NULL)
  {
  // create root node & assign data
  start = new node;
  start->data = x;
  start->left = NULL;
  start->right = NULL;
  }
else
```

```
{
 node *parent;
 cur = start;
 // search value in tree & go to proper position to insert
 while(cur!=NULL)
 {
    if(cur->data == x)
    { cout<<"Data already exist\n"; return;}
    if(x > cur->data)
    { parent=cur; cur=cur->right;}
    else
    { parent = cur; cur=cur->left;}
 }

 // create and insert new node
 temp = new node;
 temp->data = x;
 temp->left = NULL;
 temp->right = NULL;
 if(parent == NULL)
    parent = temp;
 else if( parent->data > x)
        parent->left=temp;
      else
        parent->right=temp;
}
}
```

Postorder Traversal

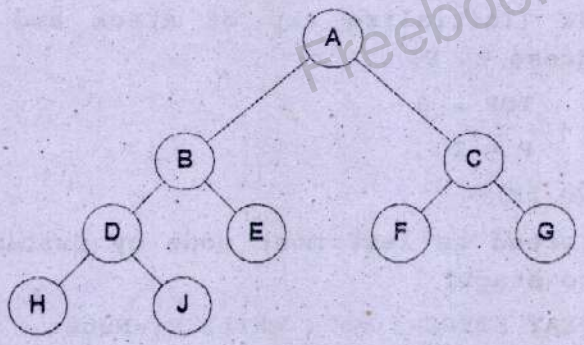
Left, Right, root.

The sequence of traversing for postorder traversing is Left, Right and then Root. A stack is used to implement the traversal. A pointer variable is also used to hold the location of node currently being scanned.

Following steps are taken to visit the binary tree in postorder traversing:

1. Proceed down the left-most path from the root node to the last left node. Push address of each left node into the stack.
2. Pop the top address from the stack and visit the node. If a sibling right node exists, proceed to the right node and repeat step-1.
3. The root is visited in the end.

Consider the following binary tree:



Postorder traversal of the tree gives the following sequence:

H J D E B F G C A

Algorithm – Postorder Traversal of Binary Tree

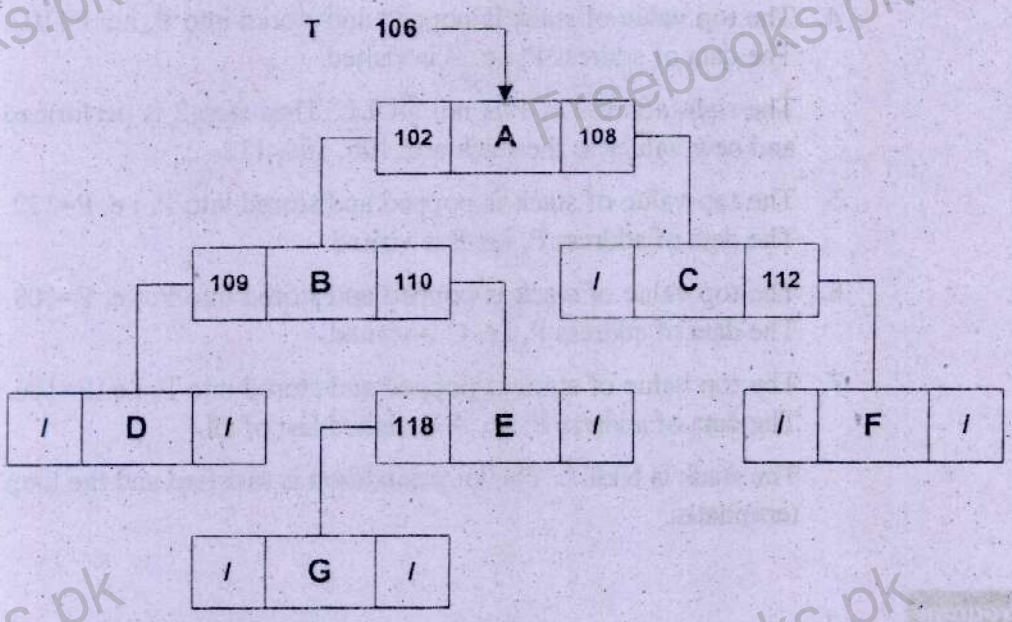
Write an algorithm for postorder traversing of a binary tree.

- Suppose the root node address of the binary tree is stored in pointer variable T.
- Another pointer variable P holds address of the current node during processing.
- STK denotes the stack.
- TOP denotes the top position of stack STK.
- PUSH function is used to insert a value into the stack.
- POP function is used to get or remove the value from top of the stack.
- L and R represent pointer variables to hold addresses of left & right children.

```
1.    [Read starting address of Tree]
      IF T = NULL THEN
            PRINT "Binary Tree is Empty"
```

```
        RETURN
    ELSE [Initialize top of stack and assign root
    address to P]
        TOP = 0
        P = T
    [END IF]
2.  [Proceed to left most node by pushing left node
    onto stack]
    REPEAT STEPS-3 to 4 WHILE P!=NULL
3.  CALL PUSH(STK, TOP, P)
4.  IF P -> L = NULL THEN
        P = P -> R
    ELSE
        P = P -> L
    END IF
    [End of step-2 loop]
5.  [Pop top value of stack into P to visit the tree
    in postorder sequence]
    REPEAT STEPS-6 to 8 WHILE TOP>0
6.  P = POP(STK, TOP)
7.  PRINT P -> Data
8.  IF P -> R != NULL THEN
        P = P -> R
        PERFORM STEP-2
    END IF
    [End of Step-5 Loop]
9.  RETURN
```

Example: A Binary Tree is given below. The values and addresses of its left & right children are also shown. T represents the starting address of the tree with value 106. P represents the current position of pointer through which the tree is traversed. Explain the steps to traverse the tree in postorder sequence using a stack.



Step-1: Starting address of the tree is assigned to P, i.e. P = 106.

Step-2: Loop While P != NULL

Starting from the root node, proceed down the left-most path to the last left node while pushing addresses of each left node into the stack.

At the end of the loop, the value in the stack will be:
106,102,109 and P = NULL

Step-3: Loop While TOP != Null

1. The top value of stack is popped and stored into P, i.e. P=109. The data of address P, i.e. D is visited.

   The right address of P is not NULL. Thus step-2 is executed and the values in the stack at the end of the loop will be: 106, 102, 110, 118.

2. The top value of stack is popped and stored into P, i.e. P=118. The data of address P, i.e. G is visited.

3. The top value of stack is popped and stored into P, i.e. P=110. The data of address P, i.e. E is visited.

4. The top value of stack is popped and stored into P, i.e. P=102. The data of address P, i.e. B is visited.

   The right address of P is not NULL. Thus step-2 is performed and new values in the stack are: 106, 108, 112.

5. The top value of stack is popped and stored into P, i.e. P=112. The data of address P, i.e. F is visited.

6. The top value of stack is popped and stored into P, i.e. P=108. The data of address P, i.e. C is visited.

7. The top value of stack is popped and stored into P, i.e. P=106. The data of address P, i.e. A is visited last of all.

   The stack is NULL. The loop condition is satisfied and the loop terminates.

Program

```
// Program to create & visit binary search tree postorder recursively
#include<iostream.h>
#include<conio.h>
struct node
    {
     int data;
     node *left;
     node *right;
    };
class tree
    {
    private:
     node *start, *cur, *temp;
    public:
     tree()
        {
         start = NULL;
        }
     void create(int);
     void postorder(void);
     void post(node *s);
  }
main()
{
tree obj;
int val,p;
clrscr();
```

```
cout<<"Enter ten values \n" ;
for(int i=1; i<=10;i++)
{
cin>>val;
obj.create(val);
}
obj.postorder();
getch();
}
// postorder recursive function to call recursive function
void tree::postorder()
 {
  post(start);
 }
// member to visit tree recursively
void tree::post( node *s)
 {
 if(s!=NULL)
  {
  post(s->left);
  post(s->right);
  cout<<s->data<<"\t";
  }
 }
// member function to create and assign data into tree nodes
void tree::create(int x)
{
 if(start==NULL)
  {
  // create root node & assign data
  start = new node;
  start->data = x;
  start->left = NULL;
  start->right = NULL;
  }
else
  {
   node *parent;
   cur = start;
   // search value in tree & go to proper position to insert
   while(cur!=NULL)
    {
       if(cur->data == x)
        { cout<<"Data already exist\n"; return;}
       if(x > cur->data)
        { parent=cur; cur=cur->right;}
       else
        { parent = cur; cur=cur->left;}
    }
```

```
        // create and insert new node
        temp = new node;
        temp->data = x;
        temp->left = NULL;
        temp->right = NULL;
        if(parent == NULL)
            parent = temp;
        else if( parent->data > x)
                parent->left=temp;
              else
                parent->right=temp;
    }
}
```

EXPRESSION BINARY TREES

The binary trees that use arithmetic operators and operands as node values are called expression binary trees. The internal nodes represent the operators of the expression. The leaves of an expression tree represent the operands of the expression which may be either constants or variables. Most operators represent binary operations and thus most internal nodes have two children. However, this is not always the case. Unary operators, such as unary minus, will have only a single child node representing the single operation of such an operator.

Expression trees are useful for representing and evaluating expressions. They are used to convert infix expressions into prefix or postfix expressions.

Constructing an Expression Binary Tree

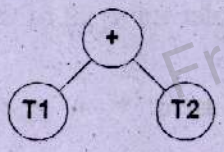
Consider the following arithmetic expression:

```
(a+(b*c))+(((d*e)+f)*g)
```

To construct the expression binary tree of the arithmetic expression, the expressions within parentheses are divided into subtrees. The left part will be the left subtree and right part will be the right subtree of the expression binary tree. Thus, the given expression is rewritten as:

`(a+(b*c))`	as left subtree say T1
`(((d*e)+f)*g)`	as right subtree say T2
`+`	root of T1 and T2

The binary tree is constructed as:

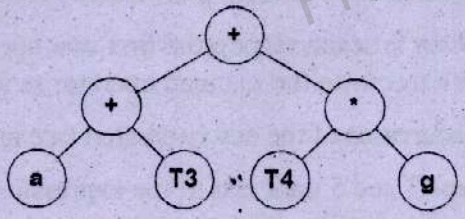


The subtrees T1 & T2 are further divided as:

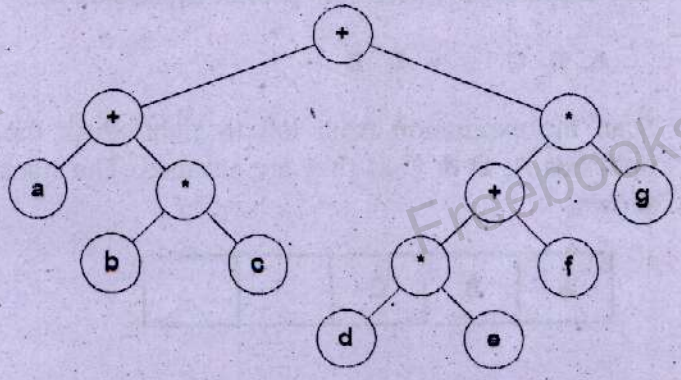
(a+(b*c)) is divided as: "a" as the left subtree
"b*c" as the left subtree T3
+ as root of these subtrees

(((d*e)+f)*g) is divided as: ((d*e)+f) as left subtree T4
g as right child, and
* as root of these subtrees.

The above tree becomes as:



The T3 & T4 are further expanded and the final expression binary tree is as given below:



To convert the expression to prefix Polish notation, the preorder traversal is performed. By using the preorder traversal of the above tree, the expression is obtained in prefix notation as:

```
+ + a * b c * + * d e f g
```

To convert the expression to postfix notation, the postorder traversal is performed. By using the postorder traversal of the above tree, the expression is written in postfix Polish notation as:

```
a b c * + d e* f + g * +
```

Constructing Binary Expression Tree Using Stack

A binary expression tree can also be constructed by using a stack. To construct a tree:

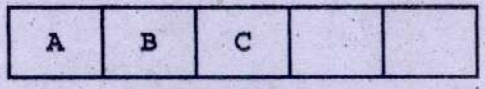
1. Convert the infix expression into postfix form.
2. Scan the postfix expression from left to right.
3. If an operand is encountered, push it into the stack.
4. If an operator is scanned, pop the first two operands from the stack and create a tree with the scanned operator as its root node.
5. Store the addresses of the newly created tree into the stack.
6. Repeat steps 3 and 5 until end of the expression.

Consider conversion of the expression `[A+(B-C)]*[(D-E)]`.

1. The expression is written in postfix expression as:

   ```
   A B C - + D E - *
   ```

2. Scan the expression from left to right. Push the first three operands A, B & C as they are scanned. The stack will be as shown:



3. Next, the operator "–" is scanned. Pop the first two operands C & B from the stack. The first tree T1 is created with the operator at its node and the two operands as its leaves.

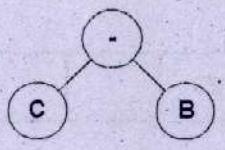
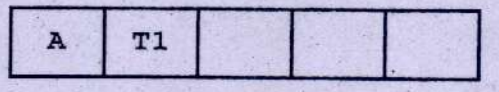


Fig: Tree T1

4. The address of the tree is then pushed into the stack. The stack will be as shown below:



5. When the operator "+" is scanned, T1 and A are popped from the stack and the tree T2 is created say with "+" operator as root and T1 and A as its leaves. The address of T2 is pushed into the stack. The stack and the expression tree will look like this:

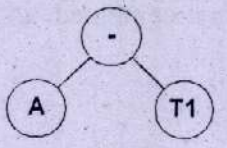
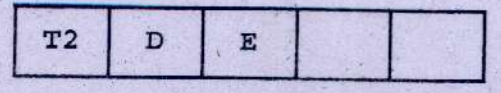
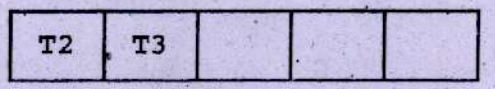


Fig: Tree T2

6. Next two operands D & E are pushed into the stack as shown below:



7. When the operator "–" is scanned, both D & E are popped. Tree T3 is created with "–"operator as root and D & E as its leaves. The address of the tree T3 is pushed into the stack. The stack and the expression tree T3 will be as shown below:



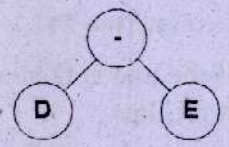


Fig: Tree T3

8. When next operator "*" is scanned, both T2 & T3 are popped and tree T4 is created with "*"operator as its root and its address is pushed into the stack.

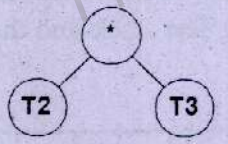
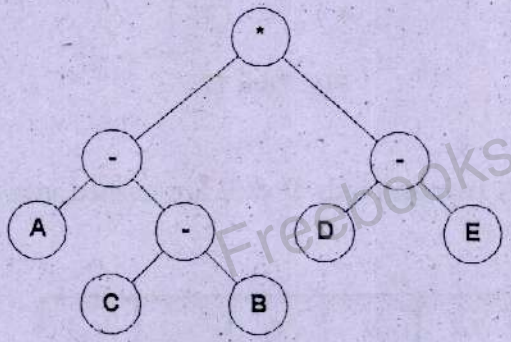


Fig: Tree T4

9. All subtrees are combined and final tree, shown below, is created.



EXERCISES

Q.1 Fill in the blanks.

1. The organization chart is best represented by _____ data structure.
2. Each node of a tree, other than the root node, has a unique predecessor or parent but may have many or no _____.
3. The node of the tree that has no parent is called _____ node.
4. The _____ node represents the beginning of tree.
5. The node which generates links to other nodes is known as _____ node.
6. The node having no successor or child is called the _____ node.
7. The nodes having a common parent are called _____.
8. The depth of a node is measured as the total number of nodes between the _____ node and the specified node.
9. The greatest depth among all the nodes in a tree is called _____ of the tree.
10. The tree with no node is called _____ tree.
11. The height of _____ tree is –1.
12. The number of children of a node is called _____ of the node.
13. If all internal nodes have same degree and all leaves are at the same level, then the tree is called _____ tree.
14. The tree whose root is its only node is called _____ tree.
15. Each node in a _____ tree can have maximum of two children.
16. A tree is said to be a _____ tree or 2-tree if each node has either 0 or 2 children.
17. In binary search trees, the _____ child node has greater or equal value than its root value.

18. In binary search trees, the _____ child node has less than its root value.

19. In _____ traversing, the binary tree is visited as: root, left child & then right child.

20. In _____ traversing, the binary tree is visited as: left child, root & then right child.

21. In _____ traversing, the binary tree is visited as: left child, right child and then the root.

22. When a binary search tree is traversed inorder and number of each node is printed then the numbers are printed in _____ order.

23. The binary tree that uses arithmetic operators and operands as node values is called _____.

Q.2 Mark True or False

1. The unique node of the tree that has no parent is called main node.
2. The node which generates links to other nodes is called parent node.
3. The nodes that are below the child node are called the parent nodes.
4. The node having no children is called leaf or leaf node.
5. The nodes having a common parent are called common nodes.
6. The tree whose root is its only node is called singleton tree.
7. In a binary tree each node can have a maximum of two children.
8. The basic component of a tree that has no parent is called the root node.
9. A tree is said to be a full binary tree if all its leaves are at same level and every interior node has two children.
10. A tree is said to be an extended binary tree or 3-Tree if each node has either 0 or 3 children.
11. The pointer field values for left and right child of leave nodes are always NULL.
12. Trees are a linear data structures.

13. In inorder traversing, the binary tree is visited as: root, left child & then right child in preorder.
14. In preorder traversing, the binary tree is visited as: left child, root & then right child in preorder.
15. In postorder traversing, the binary tree is visited as: left child, right child and then root at last in postorder.
16. The binary tree that uses arithmetic operators and operands as node values is called expression binary tree.
17. The leaves of a binary tree contain operands and other parent nodes contain operators.
18. The binary expression tree can be constructed by using a stack.
19. The organization chart is an example of a general tree.
20. Each node of a Tree has a unique predecessor or parent but may have many or no successors.
21. The greatest depth among all its nodes is called the depth of the tree.
22. The tree with no node is called NULL tree.
23. The number of children of a node is called the height of the node.
24. If all the internal nodes have the same degree and all leaves are at the same level, then the tree is called full tree.
25. In binary search trees, the left child node has greater or equal value than its root value.
26. In binary search trees, the right child node has less than its root value.
27. Each node of a binary tree has two pointer fields, one to hold the address of left child node and the other to hold the right child node.

9 GRAPHS

Many to many Representation

GRAPHS

Tree data structures represent one to many relationships. In real life we frequently come across problems that can be best described by many to many relationships. Such problems cannot be solved using trees or other data structures. To deal with such problems, graph data structures are used.

Graphs are non-linear data structures. A graph is made up of sets of nodes and lines. Nodes are called vertices or points. Lines are called edges or arcs. Lines are used to connect nodes. An edge of the graph is represented as:

```
e = [u,v]
```

'u' and 'v' denote the start and end nodes of edge 'e', respectively. They are also known as the tail and head nodes of edge 'e'.

Graphs can be used to represent essentially *any* relationship. They are used to study problems in a wide variety of areas including computer science, electrical engineering, chemistry, etc. For example, they are commonly used to represent and study transport networks, communication networks and electrical circuits.

In transportation networks, graph vertices correspond to locations between which people or goods are moved. The locations may represent cities, warehouses, airports, terminals, and so on. The edges correspond to the connections between the vertices. The edges may represent roads, railway tracks, air routes, etc. through which communication between cities takes place.

The following graph represents transportation links between five cities. The vertices represent cities. The edges correspond to the roads that link the cities.

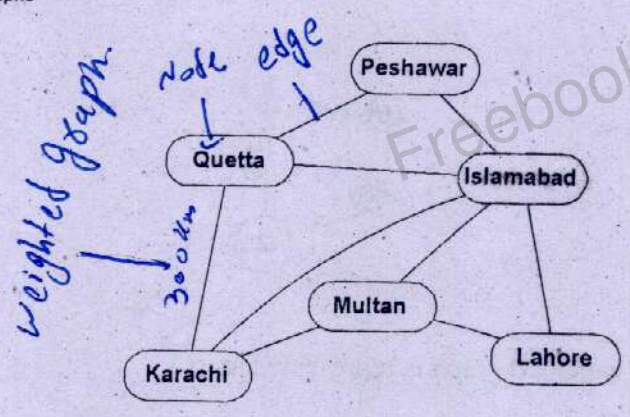


Figure: 9.1

The above graph has 6 vertices and 9 edges. The vertices represent the cities and the edges represent the links or roads between the cities.

Undirected Graphs

An *undirected graph* has edges that have no direction. An undirected graph is also called an *undigraph*.

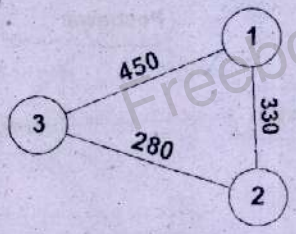
Directed Graphs

A *directed graph* has edges that are unidirectional. A directed graph is also called a *digraph*.

Weighted Graph

A graph which has a weight, or number, associated with each edge is called weighted graph. Weight of an edge is sometimes also called its cost.

The weight of an edge usually represents certain conditions or situations associated with the edge. For example, in the following weighted graph, weights of the graph denote distances between the cities.



An edge of a weighted graph is represented as:

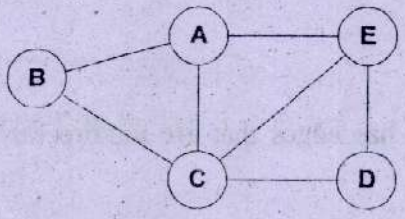
```
e = [u,v,w]
```

'u' and 'v' denote the start and end nodes of edge 'e', respectively. 'w' represents the weight of the corresponding edge.

Degree of Node

No of nodes.

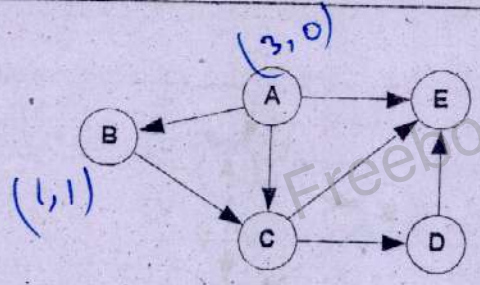
The number of edges that a node contains is called degree of the node. In graph shown below, node B has a degree of 2, node C has a degree of 4 and node A has a degree of 3.



A node that has degree 0 is called **isolated node**. A graph that has only one isolated node is called the **null graph**.

Out-degree & In-degree

The number of edges beginning from a node is called the out-degree of the node. In graph shown below, the out-degree of node A is 3. Similarly, node B has an out-degree of 1. The node that has an out-degree of 0 is called terminal node or leaf and other nodes are called the branch nodes.



The number of edges ending at a node is called the **in-degree** of the node. In the graph shown above, the in-degree of node A is 0 and node B has an in-degree of 1.

The sum of the out-degree & in-degree is the total degree of the node. The total degree of a loop node is 2 and that of an isolated node is 0.

Source & Sink Nodes

The node that has a positive out-degree but zero in-degree is called source node. In the graph shown above, node A is the source node. This node has a positive out-degree of 3 and its in-degree is zero.

The node that has zero out-degree but a positive in-degree is called sink node. In the graph shown above, node E is the sink node. This node has a positive in-degree of 3 and its out-degree is zero.

Path & Length of Graph

A list of nodes of a graph where each node has an edge from it to the next node is called the path. It is written as a sequence of **nodes** $u_1, u_2, u_3, \ldots, u_n$.

A path which repeats no node is known as the simple path. A path is usually assumed to be a simple path, unless otherwise defined.

The number of edges in a path of a graph is called length of the graph. For a path consisting of 'n' nodes, the path length is 'n–1'.

Loop Edge

An edge 'e' is said to be a loop edge if it has the same node at its tail and head. A loop edge is shown in the following figure:

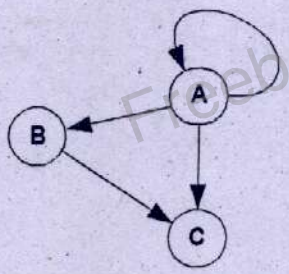


Figure: Loop Edge

Multiple Edges

A graph is said to have multiple edges if its more than one edges have the same tail and head nodes. An example of multiple edges is in the graph shown below:

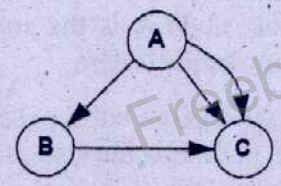


Figure: Multiple Edges

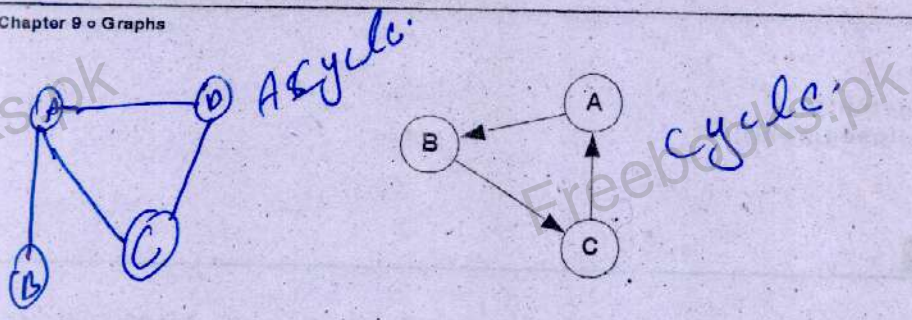
The edges that have the same tail and head nodes are known as parallel edges.

Complete Graph

A graph is said to be a complete graph if there is an edge between every pair of vertices.

Cycle & Acycle

A path which originates or begins and ends at the same node is called cycle. In other words, a path from a node to itself is called a cycle. It is also known as the circuit. The length of a cycle must be at least 1. In the figure below, the example of cycle is given.



The directed graph that has no cycles is called acyclic graph. A directed acyclic graph is also referred to as DAG.

Program

```
// Program to draw edges between vertices
// A(200,100) to B(100,300)
// A(200,100) to C(300,300)
// B(100,300) to C(300,300)
#include <graphics.h>
#include <conio.h>
class graph
 {
   public:
    draw_line(int x1, int y1, int x2, int y2)
     {
     line(x1,y1, x2,y2);
     }
 };

main(void)
{

  int gdriver = DETECT, gmode;

  // initialize graphics. The path for bgi files must
  // be correct. In my computer the path is c:\tc\bgi
  initgraph(&gdriver, &gmode, "c:\\tc\\bgi");
  cleardevice();
  outtextxy(200,90,"A");
  outtextxy(100,310,"B");
  outtextxy(300,310,"C");
// Draw line between A(200,100) to B(100,300)
  line(200,100,100,300);
// Draw line between A(200,100) to C(300,300)
  line(200,100,300,300);
// Draw line between B(100,300) to C(300,300)
  line(100,300,300,300);
```

```
    getch();
    closegraph();
}
```

Program

```
// Program to draw edges between vertices:// A(100,100) to
B(300,100)// B(300,100) to C(300,400)
//C(300,400) to D(100,400)
// D(100,400) to A(100,100)
// A(100,100) to C(300,400)
// B(300,100) to D(100,400)

#include <graphics.h>
#includa <conio.h>
class graph
 {
 public:
  draw_line(int x1, int y1, int x2, int y2)
   {
   line(x1,y1, x2,y2);
   }
 };

main(void)
{
  int gdriver = DETECT, gmode;

  // initialize graphics. The path for bgi files must
  // be correct. In my computer the path is c:\tc\bgi
  initgraph(&gdriver, &gmode, "c:\\tc\\bgi");
  cleardevice();
  outtextxy(100,90,"A");
  outtextxy(300,90,"B");
  outtextxy(300,410,"C");
  outtextxy(100,410,"D");

// Draw line between A(100,100) to B(300,100)
  line(100,100,300,100);
// Draw line between B(300,100) to C(300,400)
  line(300,100,300,400);
// Draw line between C(300,400) to D(100,400)
  line(300,400,100,400);
// Draw line between D(100,400) to A(100,100)
  line(100,400,100,100);
// Draw line between A(100,100) to C(300,400)
  line(100,100,300,400);
// Draw line between B(300,100) to D(100,400)
```

```
    line(300,100,100,400);
    getch();
    closegraph();
}
```

Representation of Graphs

Graphs are unstructured. One vertex in a graph might be adjacent to every other vertex. Similarly, a vertex might only be adjacent to just one vertex. This property of adjacency is used to represent graphs in the computer memory. There are two common ways of representing graphs. One is called using adjacency list and the other is using adjacency matrix.

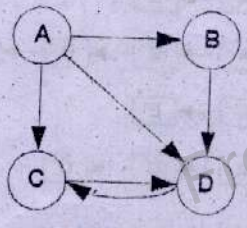
Adjacency Matrix

An adjacency matrix is a table, which represents the edges between vertices of the graph. In case of a weighted graph, the adjacency matrix specifies weights of the edges.

If an edge (u, v) exists between two vertices u and v, it is represented in the matrix as 1. Absence of an edge is represented as 0. The adjacency matrix consists of only 0s and 1s. Such a matrix, which contains only 0s and 1s is called Bit matrix or Boolean matrix.

If there are N vertices in the graph, the adjacency matrix will be an N x N matrix. The columns of the matrix represent the "from" end of the edge, and the rows represent the "to" end of the edge. The entry at (i, j) contains the weight of the edge from vertex i to vertex j. It contains 0 if no such edge exists.

The graph shown in the following figure has vertices A, B, C and D. The graph has 4 vertices, so the adjacency matrix of G is a two dimensional 4 x 4 array:



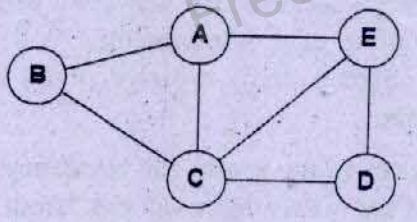
	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	0	0	1
D	0	0	1	0

In the case of a bi-directional graph, if there is an edge from vertex i to vertex j, there is also an edge from vertex j to vertex i, so the adjacency matrix will be symmetric.

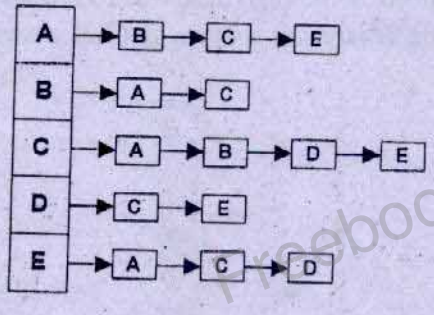
Adjacency List

An adjacency list is an array of linked lists, one for each vertex. Each linked list contains all of the vertices that are adjacent to a given vertex.

Suppose we have the following graph:



Then the adjacency list would be:



If this were a weighted graph, the nodes of the list would include both

number and cost of the vertex. If this were a directed graph, then an edge from A to B would not necessarily mean that there would be an edge from B to A.

Lists *vs* Matrices

The representation of a graph is selected depending upon the purpose of the graph.

If the graph is sparse, i.e. there are not many edges, then the matrix will take up a lot of space, but the adjacency list does not have that problem, because it only keeps track of what edges are actually in the graph. On the other hand, if there are a lot of edges in the graph, or if it is fully connected, then the list requires more space because of all of the references.

If it is to be found out whether or not an edge exists between two vertices, it can be found out directly in an adjacency matrix. Whereas, a long linked list has to be traversed to before it is found out that it is not in the graph.

On the other hand, if all of a vertex's neighbors are to be looked at, then the entire matrix will have to be scanned. Whereas, in the list, only the linked-list of neighbors is scanned.

One disadvantage of matrices is that it is difficult when a new node is inserted or an existing node is deleted. For insertion and deletion, the size of matrix has to be changed and all the nodes may have to be reordered. Thus a lot of work has to be performed to carry out these operations. These operations are better managed with adjacency lists.

Traversing of Graph

There are two standard ways to traverse a graph. These are:

- **Breadth-First Search**
- **Depth-First Search**

When one of the above methods is used then each of the vertices of graph should be in one of three states during visiting process. These states are also known as status of vertices. These states are:

Ready State: It is the initial state of the vertex. Its value is 1.

Waiting State: It is the state of the vertex when it is in the queue or stack and is waiting to be processed. Its value is 2.

Processed State: It is the state of the vertex when it has been processed. Its value is 3.

Breadth–First Search (BFS)

Breadth First Search (BFS) searches the graph one edge away from the starting vertex at a time.

The search starts at some arbitrarily chosen vertex. Suppose the search starts at some vertex v. Then all of v's neighbors are visited and processed. Then, all of v's neighbors' neighbors are visited and processed. For this, all neighbors of the first neighbor are visited, then all the neighbors of the second neighbor are visited, and so on. This process is continued until all vertices in the graph are visited.

Recursion is not used in BFS because traversing is done one level at a time. To perform a BFS, a queue is used. Every time a vertex v's neighbors are visited, v's neighbors are dequeued and enqueued. In this way, the record is kept of which neighbors belong to which vertex.

The algorithm for Breadth-First Search given below explain the process:

Algorithm — Breadth–First Search

1. Visiting first node and its neighbors of a graph.
2. Set the status of all nodes to ready state, i.e. STATUS = 1.
3. Put the starting node A into the queue and set its status to the waiting state.
4. Repeat step-5 to 6 until queue is empty.
5. Pick the front node of queue, process it and set its status to processed state.
6. Add all the neighbors of A into the queue that have status of ready state and set their status to the waiting state.

```
        [End of loop at step-4]
7.      Exit
```

Depth-First Search

In Depth First Search (DFS), the search can be started from any vertex in the graph. For example, a vertex 'v' is selected. It is processed and marked. Then all unmarked vertices adjacent to 'v' are recursively traversed. The vertex 'v' will be different with every new method call.

When a vertex is reached in which all of its neighbors have been visited, the control returns to its calling vertex, and one of its unvisited neighbors is visited. The recursion is repeated in the same manner until all of the starting vertex's neighbors are visited.

In Depth-First Search, the stack is used to hold vertices that are waiting to be processed. The following algorithm explains the Depth-First Search :

Algorithm — Depth–First Search

```
1.      Visiting first node and its neighbors along the
        path P of a graph.
2.      Set the status of all nodes to ready state, i.e.
        STATUS = 1.
3.      Push the starting node A onto the stack and set
        its status to the waiting state.
4.      Repeat step-5 to 6 until stack is empty.
5.      Pop the top node of stack, process it and set its
        status to processed state.
6.      Push all the neighbors of A onto the stack that
        have status of ready state and set their status
        to the waiting state.
        [End of loop at step-4]
8.      Exit
```

EXERCISES

Q.1 Fill in the blanks

1. Graph is _____ data structure.
2. A node of a graph is also called _____ and the set of lines connecting the nodes are called _____.
3. A graph is said to be _____ if its each edge is directed from one node to another.
4. The graph that has no direction on its edges is said to be _____ graph.
5. A graph is said to be a _____ graph if each edge is assigned a value.
6. The number of edges that a node contains is called the _____ of the node.
7. The node that has degree 0 is called _____ node.
8. A graph that has only one isolated node is called the _____ graph.
9. The number of edges beginning or outing from any node is called _____ of the node.
10. The node that has an out-degree of 0 is called _____ node.
11. The number of edges ending at any node is called the _____ degree of the node.
12. The sum of out-degree and in-degree is the _____ of a node.
13. The _____ degree of a loop node is 2 and that of an isolated node is 0.
14. The node that has a positive out-degree but zero in-degree is called the _____ node.
15. The node that has zero out-degree but positive in-degree is called _____ node.

16. A graph is said to be a _____ graph if there is an edge between its every pair of vertices.
17. A path which begins and ends in the same node is called _____.
18. The matrix which contains entries of only 0 and 1 is called a _____ or _____.
19. In _____ search in graphs, the nodes of the graph are visited in layers. The nodes closest to the start are visited first and the most distant nodes are visited last.
20. In _____ search in graphs, the queue is used to hold nodes that are waiting to be processed.
21. In _____ search in graphs, the nodes of the graph are visited by going down a branch to its deepest point and moving up.
22. In _____ search in graphs, the stack is used to hold nodes that are waiting to be processed.

Q.2 Mark True or False.

1. The node that has zero out-degree but a positive in-degree is called source node.
2. A sequence of vertices or nodes u_1, u_2, u_3,..., u_n of a graph that connects them is known as the path of the graph.
3. The number of edges appearing in the sequence of nodes u_1, u_2, u_3, ..., u_n of a graph is called length of the graph.
4. The length of graph that has n nodes is n–1.
5. A graph is said to be an incomplete graph if there is an edge between every pair of vertices.
6. The node that has an out-degree of 0 is called terminal node or leaf and other nodes are called the branch nodes.
7. The number of edges ending at any node is called the out-degree of the node.
8. The sum of out-degree and in-degree is the total degree of a node.
9. The degree of a loop node is 2 and that of an isolated node is 0.

10. The node that has a positive out-degree but zero in-degree is called sink node.
11. A path which begins and ends at the same node is called the acycle or circuit.
12. The length of a cycle must be at least 1.
13. The matrix which contains entries of only 0 and 1, is called Boolean matrix.
14. In Depth-First Search in graphs, the nodes of the graph are visited in layers. The nodes closest to the start are visited first and the most distant nodes are visited last.
15. In Breadth-First Search in graphs, the queue is used to hold nodes that are waiting to be processed.
16. In Breadth-First Search in graphs, the nodes of the graph are visited by going down a branch to its deepest point and moving up.
17. In Depth-First Search in graphs, the stack is used to hold nodes that are waiting to be processed.
18. Graph is a special type of linear data structure.
19. The nodes of a graph are also called edges or arcs.
20. The graph that has no direction of its edges is called digraph or directed graph.
21. The number of edges a node contains is called the degree of the node.
22. The node that has a zero degree is called the leaf node.
23. A graph that has only one isolated node is called the Null graph.

AIKMAN SERIES BOOKS

Aikman
Simple Steps Series

HTML IN SIMPLE STEPS
ASP IN SIMPLE STEPS
PHP IN SIMPLE STEPS
SQL IN SIMPLE STEPS
JAVASCRIPT IN SIMPLE STEPS
CASCADING STYLE SHEETS IN SIMPLE STEPS

Aikman Series

C++
VISUAL BASIC
DATA STRUCTURES
FUNDAMENTALS OF COMPUTER SYSTEM

A PRIMER ON
PROGRAMMING WITH JAVA

A PRIMER ON
DATABASE MANAGEMENT

AIKMAN BOOK COMPANY