Kavayitri Bahinabai Chaudhari
North Maharashtra University

SEM. IV ▪ BCA 402

BCA

Bachelor in Computer Application

Data Structure

- Sanjay E. Pate
- Dr. Gauri M. Patil
- Prashant M. Savdekar

Prashant

As per U.G.C. Guidelines and also on the basis of revised syllabus of
Kavayitri Bahinabai Chaudhari North Maharashtra University
with effect from Academic Year 2023-24. Also useful for all Universities.

SEM IV ▪ BCA-402

Bachelor in Computer Application

DATA STRUCTURE

Text Book Development Committee

- C O O R D I N A T O R -

Dr. K. C. Deshmukh
Head, Department of Computer Science,
Nanasaheb Y. N. Chavan Arts, Science and Commerce College, Chalisgaon.

- A U T H O R S -

Sanjay E. Pate
Department of Computer Science,
Nanasaheb Y. N. Chavan Arts, Science and Commerce College, Chalisgaon.

Dr. Gauri M. Patil
Department of Computer Science,
Bhusawal Arts, Commerce and P. O. Nahta Science College, Bhusawal.

Prashant M. Savdekar
Head, Department of BCA,
DNCVPS Shirish Madhukarrao Chaudhari College, Jalgaon.

Prashant Publications

DATA STRUCTURE

SEM IV ▪ BCA-402

Publisher | Printer
Prashant Publications
3, Pratap Nagar, Sant Dnyaneshwar Mandir Rd,
Near Nutan Maratha College, Jalgaon 425001.

Phone | Website | E-mail
0257-2235520, 0257-2232800, 9665626717
www.prashantpublications.com
prashantpublication.jal@gmail.com

Distributor
Prashant Book House
17, Stadium Shopping Centre, Opp. SBI, Jalgaon.
Ph. 0257-2227231

Available at all leading booksellers
Jalgaon | Dhule | Nadurbar

First Edition, January, 2024

ISBN 978-81-19120-36-9

Price ₹ 115/-

PREFACE

DATA STRUCTURE is a Simple version for S.Y.B.C.A. students, published by Prashant Publication.

This text is in accordance with the new syllabus CBCS-2023 recommended by the Kavayitri Bahainabai Chaudhari North Maharashtra University, Jalgaon, which has been serving the need of S.Y.B.C.A. Computer Science students from various colleges. This text is also useful for the student of Engineering, B.Sc. (Information Technology and Computer Science), M.Sc, M.C.A. B.B.M., M.B.M. other different Computer courses.

We are extremely grateful to Prof. Dr. S.R.Kolhe, Professor, Director, School of Computer Sciences and Chairman, Board of Studies, Kavayitri Bahinabai Chaudhari North Maharashtra University, Jalgaon for his valuable guidance.

We are obligated to Principals Dr. S. R. Jadhav, Nanasaheb Y. N. Chavan Arts, Science and Commerce College, Chalisgaon, Dr. B. H. Barhate Vice-principal, P.O. Nahta College, Bhusawal, Dr. P. R. Chaudhari, Shirish Madhukarrao Chaudhari Arts, Science and Commerce College, Jalgaon and Librarians and staff of respective colleges for their encouragement.

We are very much thankful to Shri. Rangrao Patil, Shri. Pradeep Patil of Prashant Publications, who has shown extreme co-operation during the preparation of this book, for getting the book published in time and providing an opportunity to be a part of this book.

We welcome any suggestions aimed at enhancing the content of this book, and we look forward to constructive feedback from our readers.

- Authors

SYLLABUS

Bachelor in Computer Applications (BCA)

Data Structure | Sem. IV | BCA-402

w.e.f. Academic Year 2023-24 (Semester System 60 + 40 Pattern)

Unit 1 : Introduction **(10 L, 15 M)**

Meaning of Data, Data item, Elementary and Group Data items, Meaning of Data Structure, Linear and Non, Linear Data Structure, Meaning of Algorithm, Algorithm development.

Unit 2 : Array **(10 L, 15 M)**

Introduction to Arrays, Definition, One Dimensional Array and Multidimensional Arrays, Representation of linear array in memory, Traversing linear array, Inserting and Deleting, Sorting (Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort), Searching (Linear Search, Binary Search).

Unit 3 : Stack **(10 L, 15 M)**

Introduction to Stack, Definition, Stack Implementation, Operations of Stack, Applications of Stack, Polish notation, Arithmetic expression, Recursion.

Unit 4 : Queues **(10 L, 15 M)**

Introduction to Queue, Definition, Queue Implementation, Operations of Queue, Circular Queue, De-queue and Priority Queue, Queue Applications.

Unit 5 : Linked List **(10 L, 15 M)**

Introduction, Representation and Operations of Linked Lists- Traversing, Searching, Insert and Delete, Singly Linked List, Doubly Linked List, Circular Linked List, And Circular Doubly Linked List.

Unit 6 : Trees and Graphs **(10 L, 15 M)**

Introduction to Tree, Binary tree, representing binary trees in memory, traversing binary trees, Threaded Binary Tree. Graph: - Types, representation in memory.

CONTENTS

1. Chapter

Introduction

Contents:

(10 L, 15 M)

1.1 Meaning of Data :

Data is a collection of facts and figures or a set of values or values of a specific format that refers to a single set of item values.

Since the invention of computers, people have been using the term "**Data**" to refer to Computer Information, either transmitted or stored. However, there is data that exists in order types as well.

Data can be numbers or texts written on a piece of paper, in the form of bits and bytes stored inside the memory of electronic devices, or facts stored within a person's mind.

For example, the Employee's name and ID are the data related to the Employee.

1.2 Data Item :

A data item represents the piece of information you find in one cell of a data table, representing one trait of one observation.

1.3 Elementary Items :

Data Items that are unable to divide into sub-items are known as Elementary Items. For example, the ID of an Employee.

Entity and Attribute :

Data items that cannot be divided into sub-items are called 'Elementary Items'. E.g. the roll number of a student is an elementary item because it cannot be divided further into sub-items. Other examples are: Marks, Age etc.

Group Items : Data items that can be divided into sub-items are called 'Group Items'. E.g. the name of a person is a group item because it can be divided into three sub-items: First Name, Middle Initial and Last Name. Other examples are: Address, Date, Qualification, Salary etc.

1.4 Meaning of Data Structure :

Data Structure is a branch of Computer Science. The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program. Data Structure is a particular way of storing and organizing data in the memory of the computer so that these data can easily be retrieved and efficiently utilized in the future when required.

The data can be managed in various ways, like the logical or mathematical model for a specific organization of data is known as a data structure.

Some examples of Data Structures are Arrays, Linked Lists, Stack, Queue, Trees, etc. Data Structures are widely used in almost every aspect of Computer Science, i.e., Compiler Design, Operating Systems, Graphics, Artificial Intelligence, and many more.

Data Structures are the main part of many Computer Science Algorithms as they allow the programmers to manage the data in an effective way. It plays a crucial role in improving the performance of a program or software, as the main objective of the software is to store and retrieve the user's data as fast as possible.

Understanding the Need for Data Structures :

As applications are becoming more complex and the amount of data is increasing every day, which may lead to problems with data searching, processing speed, multiple requests handling, and many more. Data Structures support different methods to organize, manage, and store data efficiently. With the help of Data Structures, we can easily traverse the data items. Data Structures provide Efficiency, Reusability, and Abstraction.

Why should we learn Data Structures?

1. Algorithms and data structures are two of the fundamental concepts in computer science.
2. While algorithms enable us to process data meaningfully, data structures let us arrange and store it.

3. Gaining knowledge about algorithms and data structures will make us stronger programmers.
4. We'll be able to build more dependable and efficient code.
5. Additionally, we will be able to resolve issues more rapidly and effectively.

Some examples of how data structures are used include the following :

- **Storing data :** Data structures are used for efficient data persistence, such as specifying the collection of attributes and corresponding structures used to store records in a database management system.
- **Managing resources and services :** Core operating system (OS) resources and services are enabled through the use of data structures such as linked lists for memory allocation, file directory management and file structure trees, as well as process scheduling queues.
- **Data exchange :** Data structures define the organization of information shared between applications, such as TCP/IP packets.
- **Ordering and sorting :** Data structures such as binary search trees -- also known as an ordered or sorted binary tree - provide efficient methods of sorting objects, such as character strings used as tags. With data structures such as priority queues, programmers can manage items organized according to a specific priority.
- **Indexing :** Even more sophisticated data structures such as B-trees are used to index objects, such as those stored in a database.
- **Searching :** Indexes created using binary search trees, B-trees or hash tables speed the ability to find a specific sought-after item.
- **Scalability :** Big data applications use data structures for allocating and managing data storage across distributed storage locations, ensuring scalability and performance. Certain big data programming environments -- such as Apache Spark -- provide data structures that mirror the underlying structure of database records to simplify querying.

Characteristics of Data Structures :

Data structures are often classified by their characteristics. The following three characteristics are examples :

1. **Linear or non-linear** : This characteristic describes whether the data items are arranged in sequential order, such as with an array, or in an unordered sequence, such as with a graph.
2. **Homogeneous or heterogeneous :** This characteristic describes whether all data items in a given repository are of the same type. One example is a collection of elements in an array, or of various types, such as an abstract data type defined as a structure in C or a class specification in Java.
3. **Static or dynamic** : This characteristic describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time.

Dynamic data structures have sizes, structures and memory locations that can shrink or expand, depending on the use.

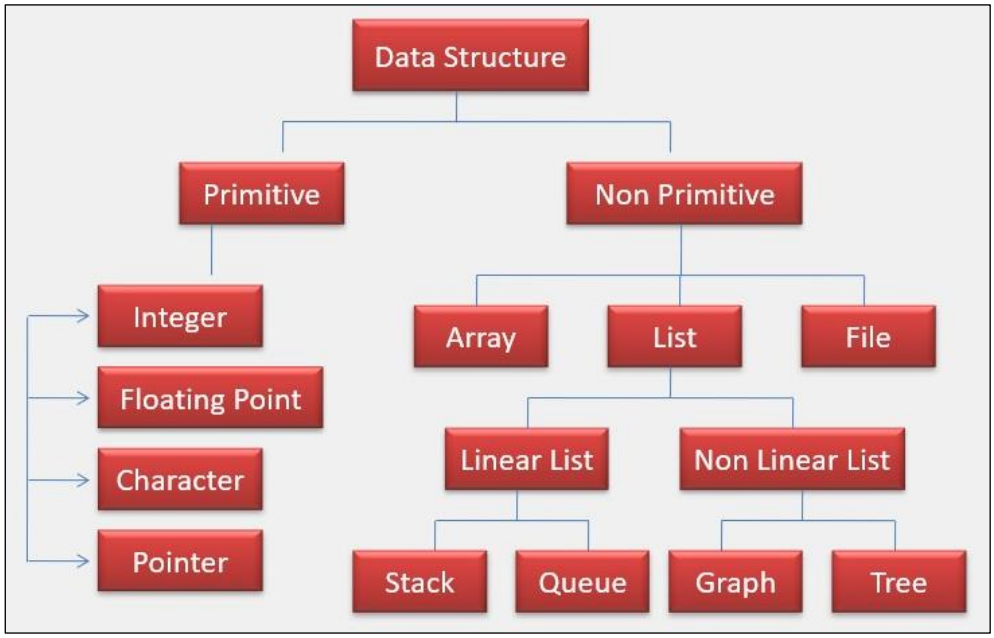
Classification of Data Structures :

A Data Structure delivers a structured set of variables related to each other in various ways. It serves as the foundation for a programming tool that helps programmers handle data effectively by indicating the relationships between data pieces.

We can classify Data Structures into two categories :

1. Primitive Data Structure
2. Non-Primitive Data Structure

The following figure shows the different classifications of Data Structures.



Primitive Data Structure :

The primitive data structure is the basic data structure that directly operates upon the machine instruction. In programming language, it is used as built in data type. It has a different representation of different computer machine.

Integer :

It represents some range of mathematical integers.

Float :

Float stores double precision floating point number.

Character :

The character represents a sequence of character data.

Pointer :

Pointer holds the memory address of another variable.

Non-Primitive Data Structures :

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).

1.5 Linear and Non-Linear Data Structure :

Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -

a) Linear Data Structures

b) Non-Linear Data Structures

a. **Linear Data Structures** : A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element.

   e.g. Arrays
   Linked Lists
   Stacks
   Queues

b. **Non-Linear Data Structures** : Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order.

   e. g. Trees
   Graphs

Basic Operations of Data Structures :

In the following section, we will discuss the different types of operations that we can perform to manipulate data in every data structure :

1. **Traversal :** Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.

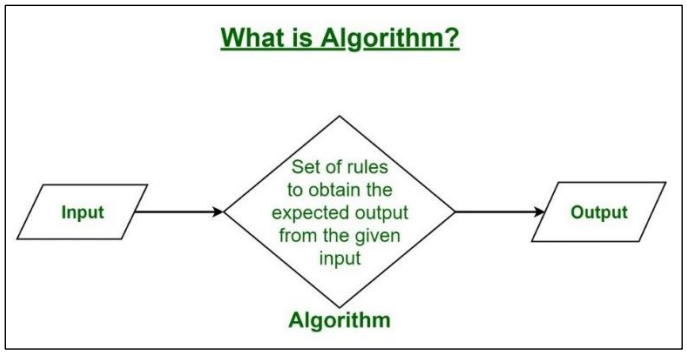
2. **Search :** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.
3. **Insertion :** Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.
4. **Deletion :** Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.
5. **Sorting :** Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.
6. **Merge :** Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.
7. **Create :** Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:
   a) Compile-time b) Run-time
   For example, the **malloc()** function is used in C Language to create data structure.
8. **Selection :** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.
9. **Update :** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.
10. **Splitting :** The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

1.6 Meaning of Algorithm in Data Structure :

An algorithm is a sequence of steps executed by a computer that takes an input and transforms it into a target output.

Together, data structures and algorithms combine and allow programmers to build whatever computer programs they would like.

Therefore Algorithm refers to a sequence of finite steps to solve a particular problem.



Use of the Algorithms :

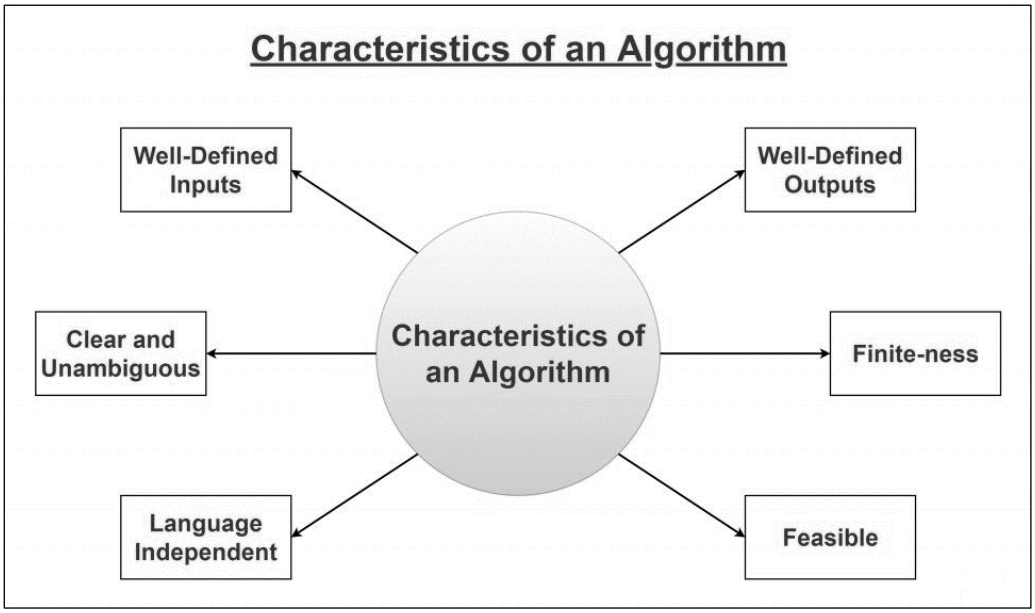
Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include :

1. **Computer Science :** Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.
2. **Mathematics :** Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.
3. **Operations Research** : Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.
4. **Artificial Intelligence :** Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.
5. **Data Science :** Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

What is the need for algorithms?

1. For complicated issues to be solved successfully and efficiently, algorithms are required.
2. They help in the automation of procedures, improving their dependability, speed, and simplicity of use.
3. Additionally, algorithms provide computers the ability to carry out tasks that would be difficult or impossible for people to carry out by hand.
4. They are used to improve procedures, analyse data, make predictions, and offer answers to issues in a variety of industries, including mathematics, computer science, engineering, finance, and many more.

What are the Characteristics of an Algorithm?



1. **Clear and Unambiguous** : The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
2. **Well-Defined Inputs** : If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
3. **Well-Defined Outputs :** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
4. **Finite-ness :** The algorithm must be finite, i.e. it should terminate after a finite time.
5. **Feasible :** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
6. **Language Independent :** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
7. **Input** : An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs.
8. **Output** : An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs.
9. **Definiteness :** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly

understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.

10. **Finiteness :** An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.
11. **Effectiveness :** An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Properties of Algorithm :

1. It should terminate after a finite time.
2. It should produce at least one output.
3. It should take zero or more input.
4. It should be deterministic means giving the same output for the same input case.
5. Every step in the algorithm must be effective i.e. every step should do some work.

Advantages of Algorithms :

1. It is easy to understand.
2. An algorithm is a step-wise representation of a solution to a given problem.
3. In an Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms :

1. Writing an algorithm takes a long time so it is time-consuming.
2. Understanding complex logic through algorithms can be very difficult.
3. Branching and Looping statements are difficult to show in Algorithms.

1.7 Algorithm Development :

Problem 1 : write an algorithm to add two numbers entered by the user

Step 1 : Start
Step 2 : Declare three variables a, b, and sum.
Step 3 : Enter the values of a and b.
Step 4 : Add the values of a and b and store the result in the sum variable, i.e., sum=a+b.
Step 5 : Print sum
Step 6 : Stop

Problem 2 : Write an algorithm to find the largest of 2 numbers?

Step 1 : Start
Step 2 : Input the values of A, B Compare A and B.

Step 3 : If A > B then go to step 5
Step 4 : Print “B is largest” go to Step 6
Step 5 : Print “A is largest”
Step 6 : Stop

Problem 3 : write an algorithm to find the factorial of a number

Step 1 : Start
Step 2 : Read a number n
Step 2 : Initialize variables:
i = 1, fact = 1
Step 3 : if i <= n go to step 4 otherwise go to step 7
Step 4 : Calculate
fact = fact * i
Step 5 : Increment the i by 1 (i=i+1) and go to step 3
Step 6 : Print fact
Step 7 : Stop

: QUESTIONS :

Short Answer Type Questions :

1. What is data structure? Explain various types of data structure.
2. Differentiate between linear and non-linear data structures.
3. Why we need data structure?
4. What is an algorithm? Discuss the different steps in the development of an algorithm?
5. Explain Linear Data structure with examples.
6. Explain Non Linear Data structure with examples.

Long Answers Type Questions :

1. What do you mean by Array? Describe the storage structure of array. Also explain various types of array in detail.
2. Write an algorithm for binary search and discuss its speed compared with linear search.
3. Write an algorithm for binary search and discuss its speed compared with Binary search.

***

2. Chapter

Array

Contents:

(10 L, 15 M)

2.1 Definition:

What is an Array?

- An **array** is a collection of items of the same variable type that are stored at contiguous memory locations.
- It's one of the most popular and simple data structures and is often used to implement other data structures.
- Each item in an array is indexed starting with 0.
- It falls under the category of linear data structures.
- Arrays have a fixed size where the size of the array is defined when the array is initialized.

Terminologies :

Two essential terminologies that are used in the array -

- **Element** - each item stored in an array is called an element.
- **Index** - each memory location of an element in an array is identified by a numerical index.

In the above image, [1, 2, 5, 3, 4, 8] are the elements of the array and 0, 1, 2, 3, 4, and 5 are the indexes of the elements of the array.

2.2 Representation of Array :

Arrays are represented in various ways in different languages. Generally, we need three things to declare an array.

1. **Data Type** : Data type of element you want to store in an array.
2. **Name :** Name of the array.
3. **Size :** Length of the array i.e. how many elements you want to store.

In C language, the array is declared as

```
data_type name_of_array[size_of_array];
```

For Example : To store 6 integers in an array we declared it as

```
int A[6]; // here A is the name of array
```

In most of the languages, the indexing of array elements starts from 0.

As arrays store elements in contiguous memory locations. This means that any element can be accessed by adding an offset to the base value of the array or the location of the first element. We can access elements as A[i] where "i" is the index of that element.

Basic Operations :

There are some specific operations that can be performed or those that are supported by the array. These are :

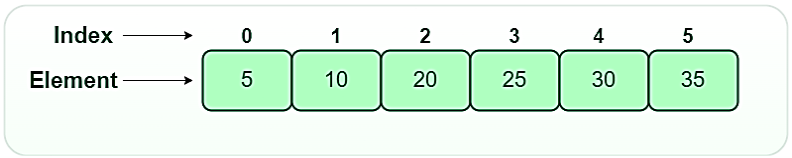
- **Traversal :** To traverse all the array elements one after another.
- **Insertion :** To add an element at the given position.
- **Deletion :** To delete an element at the given position.
- **Searching :** To search an element(s) using the given index or by value.

- **Updating :** To update an element at the given index.
- **Sorting :** To arrange elements in the array in a specific order.
- **Merging :** To merge two arrays into one.

2.3 Types of arrays :

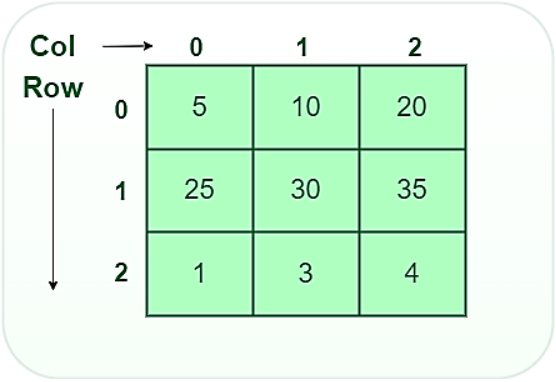
There are majorly two types of arrays :

- **One-dimensional array (1-D arrays) :** You can imagine a 1d array as a row, where elements are stored one after another.



1D array

- **Two-dimensional array :** 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.



2D array

2.4 What is traversal operation in array?

Traversal operation in array or simply traversing an array means, Accessing or printing each element of an array exactly once so that the data items (values) of the array can be checked or used as part of some other operation or process (This accessing and processing is sometimes called "visiting" the array).

Note : Accessing or printing of elements always takes place one by one.

Algorithm for Traversing an Array :

- **Step 01 :** Start
- **Step 02 :** [Initialize counter variable.] Set i = LB.
- **Step 03 :** Repeat for i = LB to UB.
- **Step 04 :** Apply process to arr[i].
- **Step 05 :** [End of loop.]
- **Step 06 :** Stop

Variables used :

1. **i :** Loop counter or counter variable for the for loop.
2. **arr :** Array name.
3. **LB :** Lower bound. [The index value or simply index of the first element of an array is called its **lower bound**]
4. **UB :** Upper bound. [The index of the last element is called its **upper bound**]

// C++ program to traverse the array

```
#include <iostream.h>
// Function to traverse and print the array
void printArray(int* arr, int n)
{
   int i;
   cout << "Array: ";
   for (i = 0; i < n; i++)
   {
      cout << arr[i] << "  ";
   }
}
void main()
{
   int arr[] = {2, -1, 5, 6, 0, -3};
   int n = sizeof(arr) / sizeof(arr[0]);

   printArray(arr, n);
  getch();
}
```

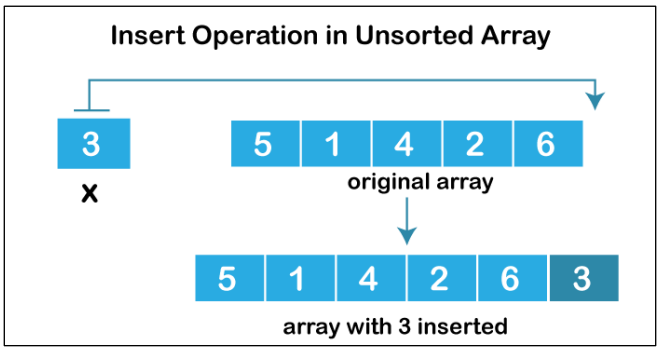
Output :

```
Array: 2  -1  5  6  0  -3
```

2.5 Insert Operation :

1. Insert at the end :

In an unsorted array, the insert operation is quicker than in a sorted array as we do not need to worry about the position at which the element is to be inserted.



C++ Program:

```
#include <iostream.h>
using namespace std;
int insertSorted(int array[], int n, int key, int capacity)
{
   if (n >= capacity)
      return n;
    array[n] = key;
    return (n + 1);
}
 void main()
{
   int array[20] = { 22, 26, 30, 50, 60, 80 };
   int capacity = sizeof(array) / sizeof(array[0]);
   int n = 6;
   int i, key = 34;
    cout << "\n Before Insertion: ";
   for (i = 0; i < n; i++)
      cout << array[i] << " ";
    n = insertSorted(array, n, key, capacity);
    cout << "\n After Insertion: ";
   for (i = 0; i < n; i++)
```

```
    cout << array[i] << " ";
     getch();
}
```

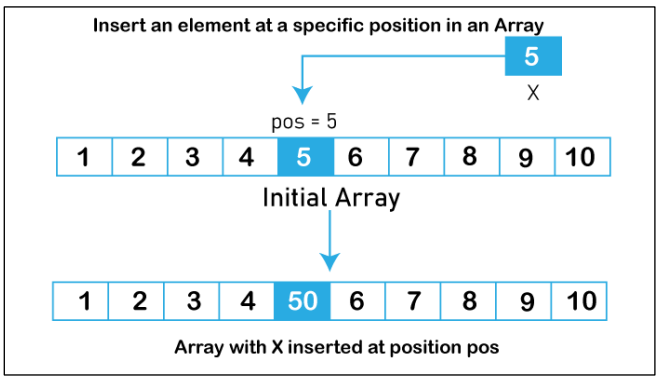
Output :

Before Insertion: 22 26 30 50 60 80

After Insertion: 22 26 30 50 60 80 34

Insert at any point :

Insert operations in an array can be performed at any point by moving elements to the right that are on the right side of the desired position.



C++ Program :

```
    #include <iostream.h>
using namespace std;
 void insertElement(int array[], int n, int x, int pos)
{
   for (int i = n - 1; i >= pos; i--)
      array[i + 1] = array[i];
    array[pos] = x;
}
 void main()
{
   int array[15] = { 1, 3, 5, 7, 4 };
   int n = 5;
    cout << "Before insertion : ";
   for (int i = 0; i < n; i++)
      cout << array[i] << " ";
```

```
    cout << endl;
    int x = 9, pos = 3;
    insertElement(array, n, x, pos);
  n++;
    cout << "After insertion : ";
  for (int i = 0; i < n; i++)
     cout << array[i] << " ";
   getch();
}
```

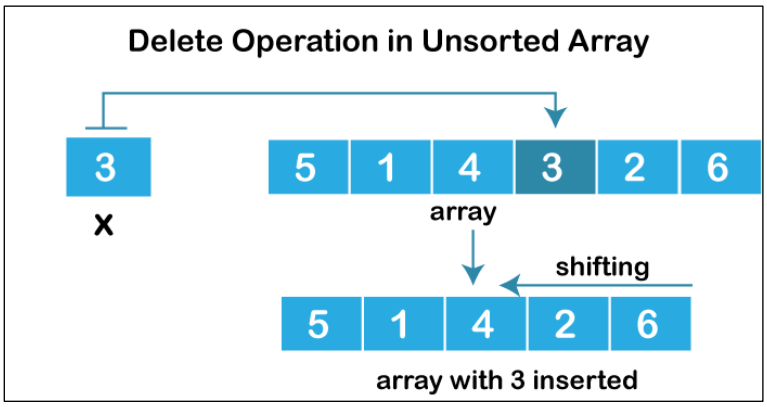
Output :

Before insertion: 1 3 5 7 4

After insertion: 1 3 5 9 7 4

2.6 Delete Operation :

The element to be deleted is found using a linear search, and then the delete operation is executed, followed by relocating the elements.



C++ Program :

```
#include <iostream.h>
using namespace std;
  int findElement(int array[], int n, int key);
  int deleteElement(int array[], int n, int key)
{
    int pos = findElement(array, n, key);
     if (pos == -1) {
       cout << "Element not found";
```

```
        return n;
    }
      int i;
    for (i = pos; i < n - 1; i++)
        array[i] = array[i + 1];
      return n - 1;
}
  int findElement(int array[], int n, int key)
{
    int i;
    for (i = 0; i < n; i++)
        if (array[i] == key)
            return i;
      return -1;
}
void main()
{
    int i;
    int array[] = { 20, 60, 40, 50, 30 };
      int n = sizeof(array) / sizeof(array[0]);
    int key = 40;
      cout << "Array before deletion\n";
    for (i = 0; i < n; i++)
        cout << array[i] << " ";
         n = deleteElement(array, n, key);
      cout << "\n\nArray after deletion\n";
    for (i = 0; i < n; i++)
        cout << array[i] << " ";
     getch();
}
```

Output :

Array before deletion
20 60 40 50 30
Array after deletion
20 60 50 30

Program to Insert & Delete an element:

```
#include<iostream.h >
int a[20],b[20],c[40];
int m,n,p,val,i,j,key,pos,temp;
/*Function Prototype*/
void display();
void insert();
void del();
void main()
{
int choice;
cout<<"\nEnter the size of the array elements:\t";
cin>>n;
cout<<"\nEnter the elements for the array:\n";
for (i=0;i<n;i++)
{
cin>>a[i];
}
do {
cout<<"\n\n--------Menu----------\n";
cout<<"1.Insert\n";
cout<<"2.Delete\n";
cout<<"3.Exit\n";
cout<<"----------------------";
cout<<"\nEnter your choice:\t";
cin>>choice;
switch (choice)
{
case 1: insert();
break;
case 2: del();
break;
case 3:break;
default :cout<<"\nInvalid choice:\n";
}
} while (choice!=3);
getch();
```

```
}
void display()    //displaying an array elements
{
int i;
cout<<"\n The array elements are:\n";
for(i=0;i<n;i++)
{
cout<<a[i]<<" ";
}
}       //end of display()
void insert()       //inserting an element in to an array
{
cout<<"\nEnter the position for the new element:\t";
cin>>pos;
cout<<"\nEnter the element to be inserted :\t";
cin>>val;
for (i=n; i>=pos-1; i--)
{
a[i+1]=a[i];
}
a[pos-1]=val;
 n=n+1;
display();
}       //end of insert()
void del()          //deleting an array element
{
cout<<"\n Enter the position of the element to be deleted:\t";
cin>> pos;
val= a [pos];
for (i= pos;i<n-1;i++)
{
a[i]=a[i+1];
}
n=n-1;
cout<<"\nThe deleted element is = "<<val;
display();
}       //end of delete()
```

2.7 Sorting :

What is Sorting :

Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. For example, consider an array A = {A1, A2, A3, A4, ?? An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > ? > An .

Consider an array ;

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)

The Array sorted in ascending order will be given as;

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

There are many techniques by using which, sorting can be performed. In this section of the tutorial, we will discuss each method in detail.

Sorting Algorithms

Sorting algorithms are described in the following table along with the description.

SN	Sorting Algorithms	Description
1	Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replacing them accordingly.
2	Selection Sort	Selection sort finds the smallest element in the array and places it on the first place on the list, then it finds the second smallest element in the array and places it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worse than insertion sort.
3	Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method that is used to arrange the deck of cards while playing bridge.
4	Quick Sort	Quick sort is the most optimized sort algorithm that performs sorting in O(n log n) comparisons. Like Merge sort, quick sort also works by using the divide and conquer approach.
5	Merge Sort	Merge sort follows the divide and conquer approach in which, the list is first divided into sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.

Bubble Sort :

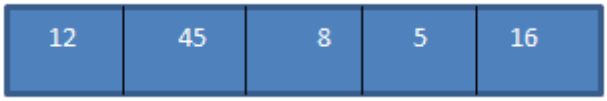
Bubble sort is the simplest technique in which we compare every element with its adjacent element and swap the elements if they are not in order. This way at the end of every iteration (called a pass), the heaviest element gets bubbled up at the end of the list.

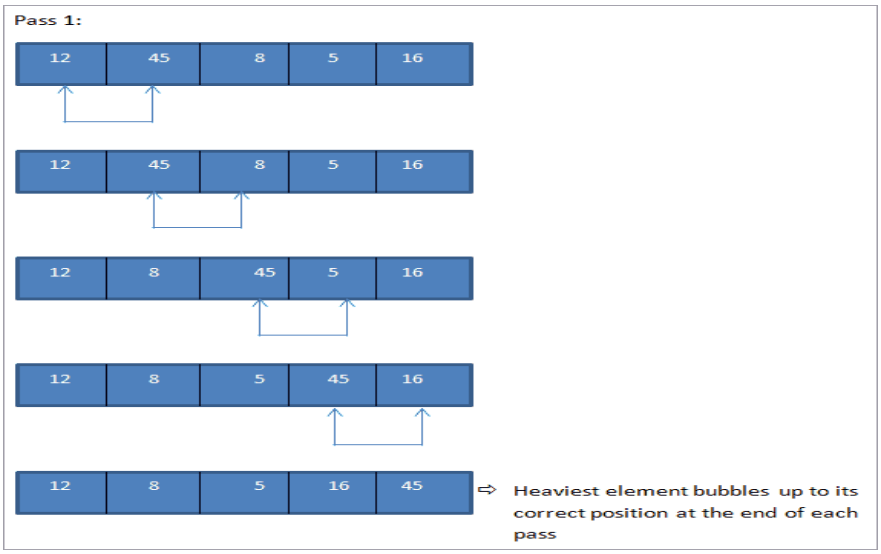
Algorithm Bubble sort (Data, N.) → 12)

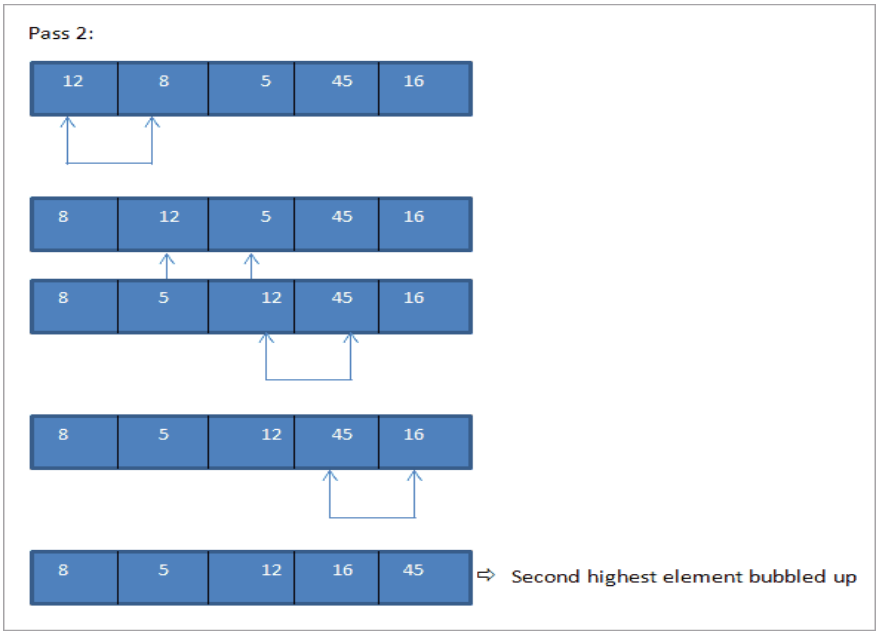
1. start
2. Accept n number of elements & store them in array A from A [0] to A [n-1].
3. Perform steps 4 to step 8 for i = 0 to n-1
4. Perform step 5 to step 7 for j=0 to n-1 pass.
5. Is A[j] > A [j + 1]? If yes then go to the next step else Step 7.
6. Interchange, values of A[j] & A [j+1] as
   temp= A[j]
   A[j] = A [j+1]
   A [j+1] = temp
7. Increment the value of j by 1 & go to step 4.
8. Increment the value of i by 1 & go to step 3.

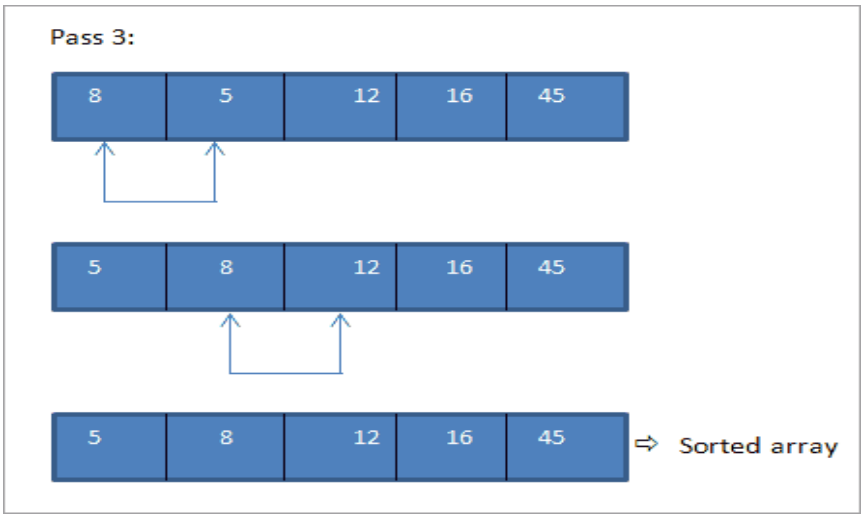
Given below is an Example of Bubble Sort :

Array to be sorted :









As seen above since it's a small array and was almost sorted, we managed to get a completely sorted array in a few passes.

Program : Write a program to implement bubble sort in C++ language.

```
#include<iostream.h>
using namespace std;
  void print(int a[], int n) //function to print array elements
  {
  int i;
```

```
    for(i = 0; i < n; i++)
    {
      cout<<a[i]<<" ";
    }
    }
 void bubble(int a[], int n) // function to implement bubble sort
 {
 int i, j, temp;
  for(i = 0; i < n; i++)
    {
     for(j = i+1; j < n; j++)
       {
          if(a[j] < a[i])
          {
             temp = a[i];
             a[i] = a[j];
             a[j] = temp;
          }
       }
    }
   }
 void main()
 {
    int i, j,temp;
    int a[5] = {45, 1, 32, 13, 26};
    int n = sizeof(a)/sizeof(a[0]);
    cout<<"Before sorting array elements are - \n";
    print(a, n);
    bubble(a, n);
    cout<<"\nAfter sorting array elements are - \n";
    print(a, n);
 return 0;
 }
```

Output :

```
Before sorting array elements are -
45 1 32 13 26
After sorting array elements are -
1 13 26 32 45
```

Selection Sort :

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted into its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm.

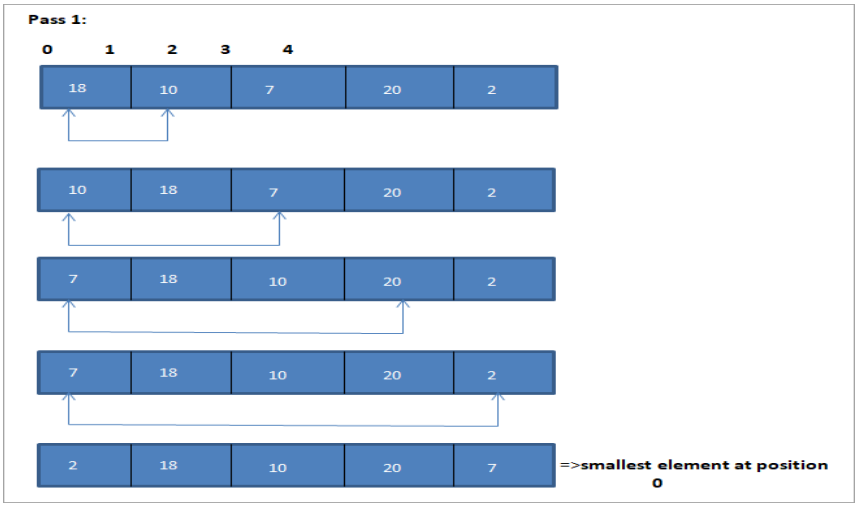
In this algorithm, the array is divided into two parts, the first is the sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and the unsorted part is the given array. The sorted part is placed at the left, while the unsorted part is placed at the right.

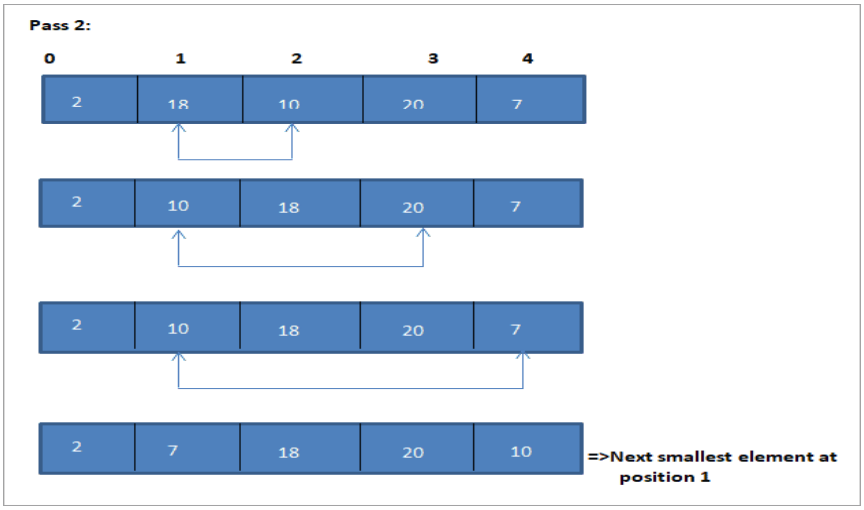
We compare each element with all its previous elements and put or insert the element in its proper position. The insertion sort technique is more feasible for arrays with a smaller number of elements. It is also useful for sorting linked lists.

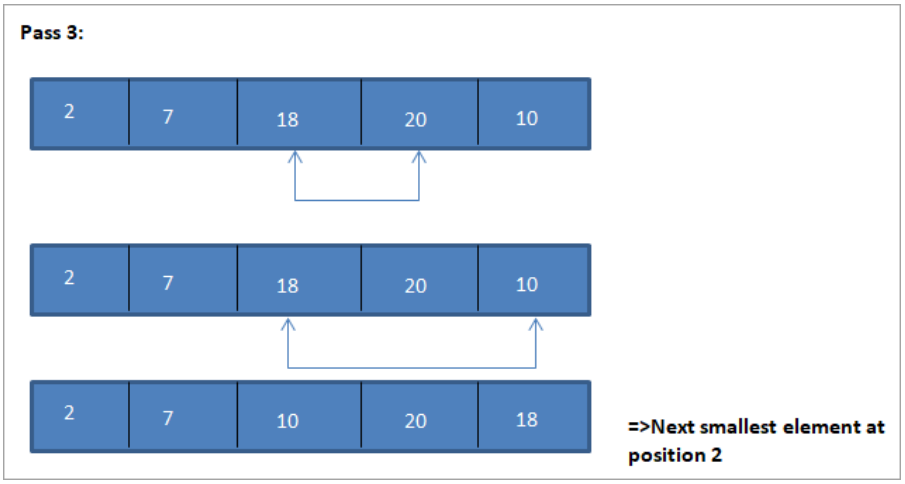
*** Algorithm for selection sort :**

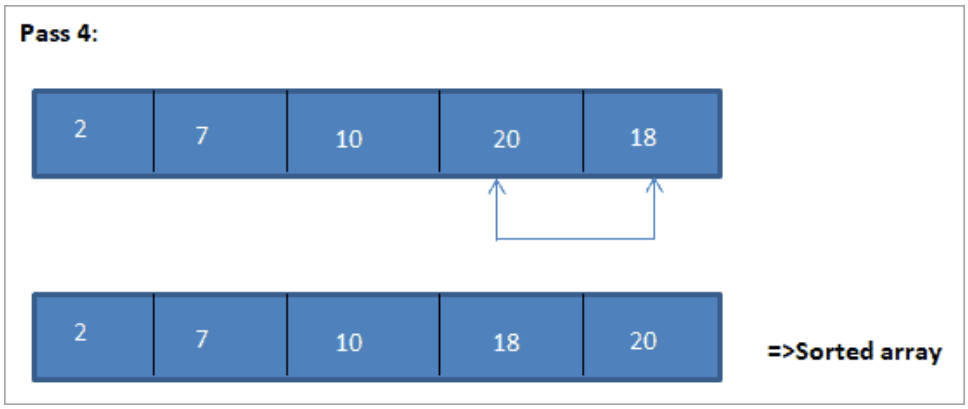
1. Start
2. Create an array & with n elements.
3. Let i=0
4. Repeat steps 5 to 10 as long as i<n-1
5. Let smallest = i
6. Let j=i+1
   Repeat steps 7 to 8 as long as j<n.
7. If x [j] < x [smallest] then smallest = j.
   Increment j by 1.
8. Interchange x [i] & x [smallest]
9. Increment I by 1.
10. Stop

An example to illustrate this selection sort algorithm is shown below.




Pass 2:
0
1
2
3
4
2
18
10
20
7
2
10
18
20
7
2
10
18
20
7
2
7
18
20
10
=>Next smallest element at position 1


Pass 3:
2
7
18
20
10
2
7
18
20
10
2
7
10
20
18
=>Next smallest element at position 2


Pass 4:
2
7
10
20
18
2
7
10
18
20
=>Sorted array

The tabular representation for this illustration is shown below :

Unsorted list	Least element	Sorted list
{18,10,7,20,2}	2	{}
{18,10,7,20}	7	{2}
{18,10,20}	10	{2,7}
{18,20}	18	{2,7,10)
{20}	20	{2,7,10,18}
{}		{2,7,10,18,20}

From the illustration, we see that with every pass the next smallest element is put in its correct position in the sorted array. From the above illustration, we see that in order to sort an array of 5 elements, four passes were required. This means in general, to sort an array of N elements, we need N-1 passes in total.

Program : Write a program to implement selection sort in C++ language.

```
#include <iostream.h>
  using namespace std;
  void selection(int arr[], int n)
{
   int i, j, small;

   for (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
   {
      small = i; //minimum element in unsorted array
            for (j = i+1; j < n; j++)
      if (arr[j] < arr[small])
         small = j;
// Swap the minimum element with the first element
   int temp = arr[small];
   arr[small] = arr[i];
   arr[i] = temp;
   }
}
  void printArr(int a[], int n) /* function to print the array */
{
   int i;
```

```
    for (i = 0; i < n; i++)
        cout<< a[i] <<" ";
}
 void main()
{
    int a[] = { 80, 10, 29, 11, 8, 30, 15 };
    int n = sizeof(a) / sizeof(a[0]);
    cout<< "Before sorting array elements are - "<<endl;
    printArr(a, n);
    selection(a, n);
    cout<< "\nAfter sorting array elements are - "<<endl;
    printArr(a, n);

  getch();
}
```

Output :

```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

Insertion Sort :

Suppose array 'A' with 'n' elements A[1], A[2].... A[N]. The Insertion sort algorithm scans 'A' from A [1] to A [N]. Inserting each element A[K] into its proper position is the previously sorted array A[1], A[2]

Pass 1: A[I] is itself trivially sorted.

Pass2: A [2] is inserted either before or after A[1], so that A [1], A [2] is sorted.

Pass 3: A [3] is inserted into its proper position is A [1], A [2], that is before A [1], between A[1] & A[2] or after A [2], so that A[1], A[2], A[3] is sorted.

Pass 4: A[4] is inserted into its proper position in A[1], A[2], A [3], so that A [1], A[2], A[3], A[4] is sorted.

……

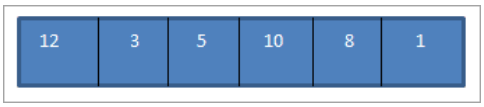
Pass N : A[N] is inserted into its proper place in A[1], A[2]...A[N] so that A[1], A[2],... A [N] is sorted.

Algorithm :

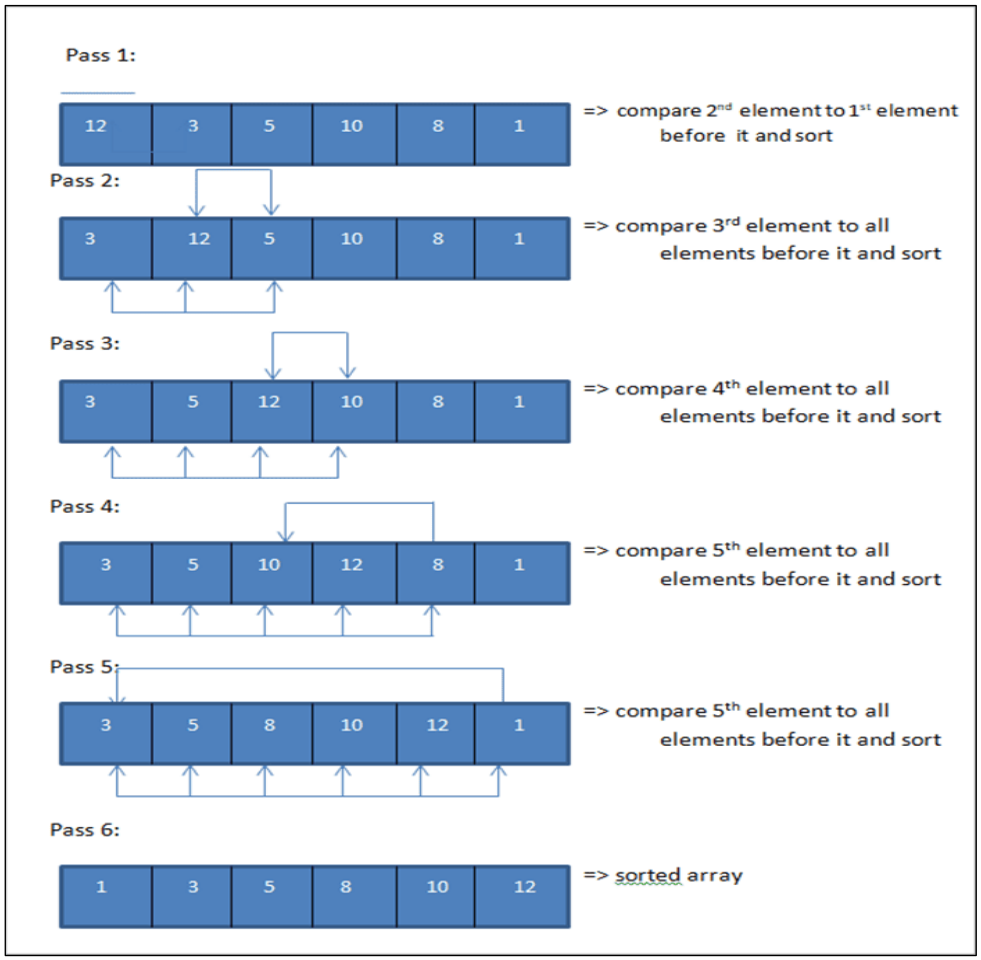
1. Start
2. Accept 'n' no. of elements to be sorted & store in array ‘A'.

3. Repeat step 4 to step 9, for j=1 to n-1
4. KEY = A [j].
5. i=j-1
6. Repeat step 7 & 8, while I >= 0 & A [i] > Key
7. A [i+1] = A [i]
8. i=i-1
9. A [i+1] = KEY
10. Stop

The array to be sorted is as follows :



Now for each pass, we compare the current element to all its previous elements. So in the first pass, we start with the second element.



Thus we require N number of passes to completely sort an array containing N number of elements.

The above illustration can be summarized in a tabular form :

Pass	Unsorted list	Comparison	Sorted list
1	{12,3,5,10,8,1}	{12,3}	{3,12,5,10,8,1}
2	{3,12,5,10,8,1}	{3,12,5}	{3,5,12,10,8,1}
3	{3,5,12,10,8,1}	{3,5,12,10}	{3,5,10,12,8,1}
4	{3,5,10,12,8,1}	{3,5,10,12,8}	{3,5,8,10,12,1}
5	{3,5,8,10,12,1}	{3,5,8,10,12,1}	{1,3,5,8,10,12}
6	{}	{}	{1,3,5,8,10,12}

As shown in the above illustration, we begin with the 2nd element as we assume that the first element is always sorted. So we begin with comparing the second element with the first one and swap the position if the second element is less than the first.

This comparison and swapping process positions two elements in their proper places. Next, we compare the third element to its previous (first and second) elements and perform the same procedure to place the third element in the proper place.

In this manner, for each pass, we place one element in its place. For the first pass, we place the second element in its place. Thus in general, to place N elements in their proper place, we need N-1 passes.

Write a program to implement insertion sort in C++ language.

```
#include <iostream.h>
using namespace std;
  void insert(int a[], int n) /* function to sort an aay with insertion sort */
{
   int i, j, temp;
   for (i = 1; i < n; i++) {
      temp = a[i];
      j = i - 1;
        while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position a
head from their current position*/
      {
         a[j+1] = a[j];
         j = j-1;
      }
      a[j+1] = temp;
```

```
    }
}

void printArr(int a[], int n) /* function to print the array */
{
    int i;
    for (i = 0; i < n; i++)
        cout << a[i] <<" ";
}
  void main()
{
    int a[] = { 89, 45, 35, 8, 12, 2 };
    int n = sizeof(a) / sizeof(a[0]);
    cout<<"Before sorting array elements are - "<<endl;
    printArr(a, n);
    insert(a, n);
    cout<<"\nAfter sorting array elements are - "<<endl;
    printArr(a, n);
  getch();
}
```

Output :

```
Before sorting array elements are -
89 45 35 8 12 2
After sorting array elements are -
2 8 12 35 45 89
```

Quick Sort :

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm.

This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into sub problems, then solving the sub problems, and combining the results back together to solve the original problem.

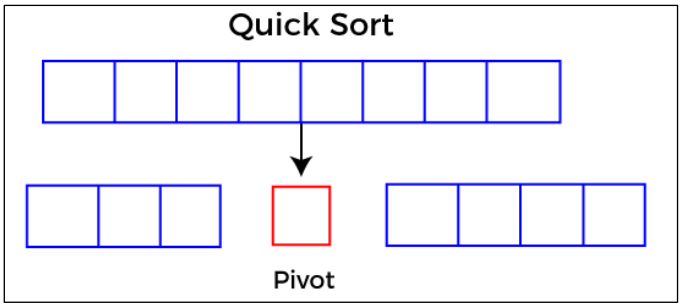
Divide : In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer : Recursively, sort two sub arrays with Quicksort.

Combine : Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot :

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows :

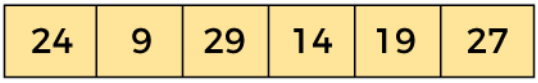
- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Working of Quick Sort Algorithm :

Now, let's see the working of the Quicksort Algorithm.

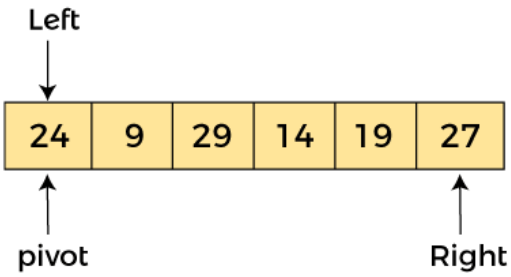
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

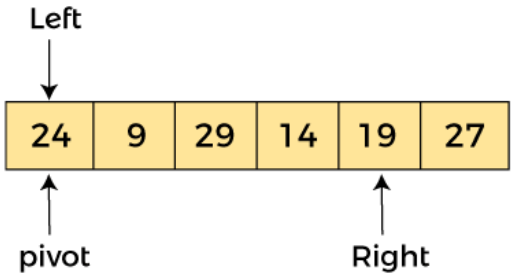


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

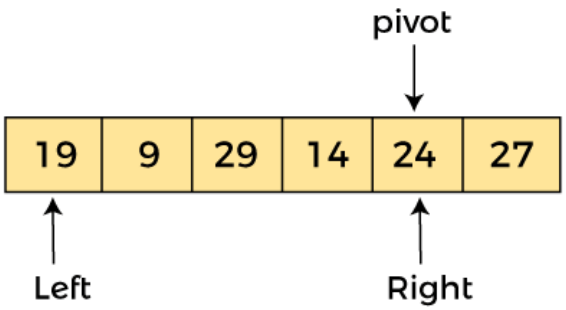


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



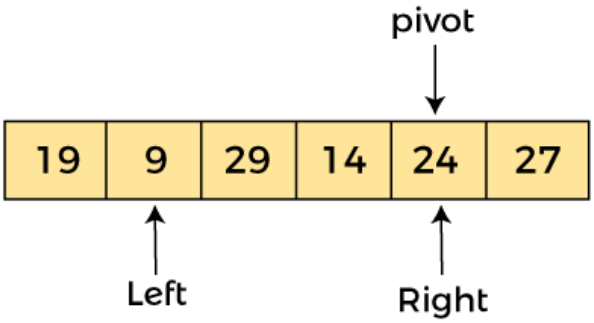
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

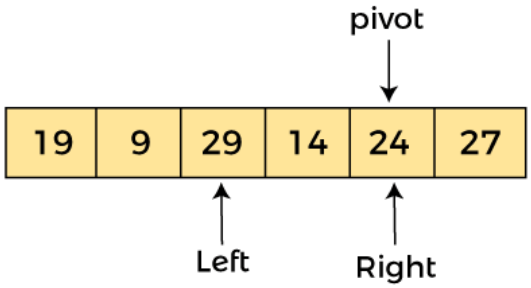


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

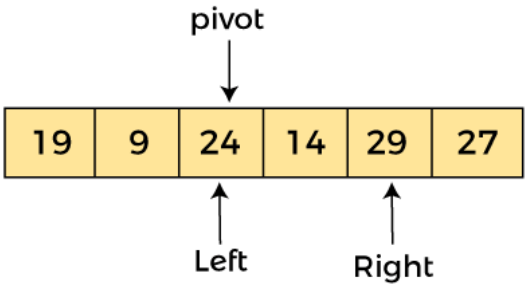
As a[pivot] > a[left], so algorithm moves one position to right as -



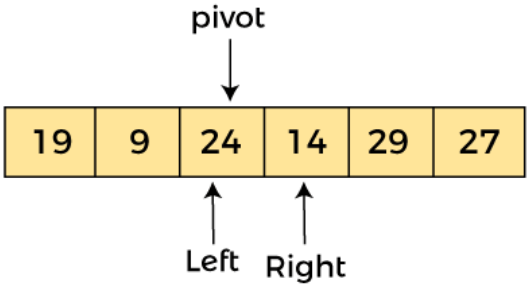
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



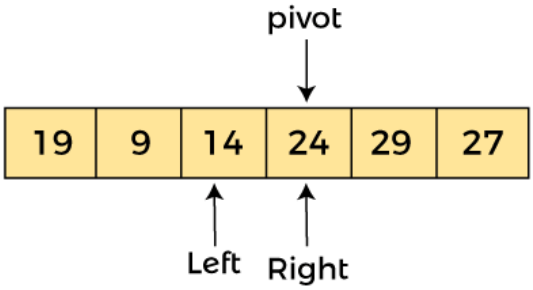
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



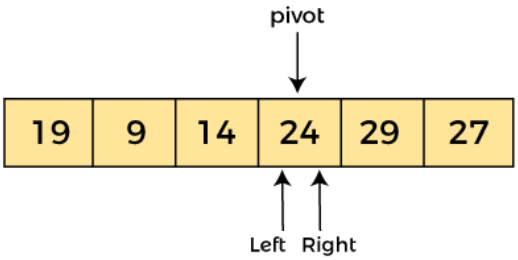
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



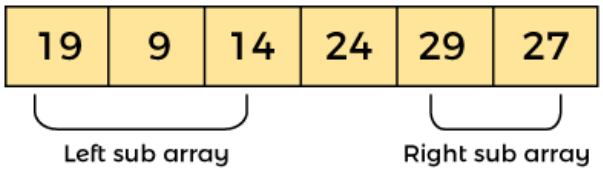
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



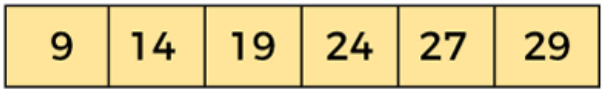
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Algorithm of Quick Sort :

1. Initialize dn = lb, up = ub
2. Initialize pivot =A [lb]
3. Repeat step 4 to 7 till dn < up
4. While (A [dn] <= pivot && dn <up)
   dn++
5. while (A [up]> pivot)
   up--
6. If (dn <up)
   Interchange A [dn] & A [up]
7. Else interchange A [up] & pivot
   j=up that is pivot position = up.

Program : Write a program to implement quick sort in C++ language.

```
#include <iostream.h.h>
/* function that consider last element as pivot, place the pivot at its exact position, and place  s
maller elements to left of pivot and greater elements to right of pivot.  */
int partition (int a[], int start, int end)
{
   int pivot = a[end]; // pivot element
   int i = (start - 1);
    for (int j = start; j <= end - 1; j++)
   {
      // If current element is smaller than the pivot
      if (a[j] < pivot)
      {
         i++; // increment index of smaller element
         int t = a[i];
         a[i] = a[j];
         a[j] = t;
      }
   }
   int t = a[i+1];
   a[i+1] = a[end];
   a[end] = t;
   return (i + 1);
}
  /* function to implement quick sort */
void quick(int a[], int start, int end)
/* a[] = array to be sorted, start = Starting index, end = Ending index */
{
   if (start < end)
   {
      int p = partition(a, start, end);  //p is the partitioning index
      quick(a, start, p - 1);
      quick(a, p + 1, end);
   }
}
```

```
/* function to print an array */
void printArr(int a[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      cout<<a[i]<< " ";
}
void main()
{
   int a[] = { 23, 8, 28, 13, 18, 26 };
   int n = sizeof(a) / sizeof(a[0]);
   cout<<"Before sorting array elements are - \n";
   printArr(a, n);
   quick(a, 0, n - 1);
   cout<<"\nAfter sorting array elements are - \n";
   printArr(a, n);

  getch();
}
```

Output :

```
Before sorting array elements are -
23 8 28 13 18 26
After sorting array elements are -
8 13 18 23 26 28
```

Merge Sort :

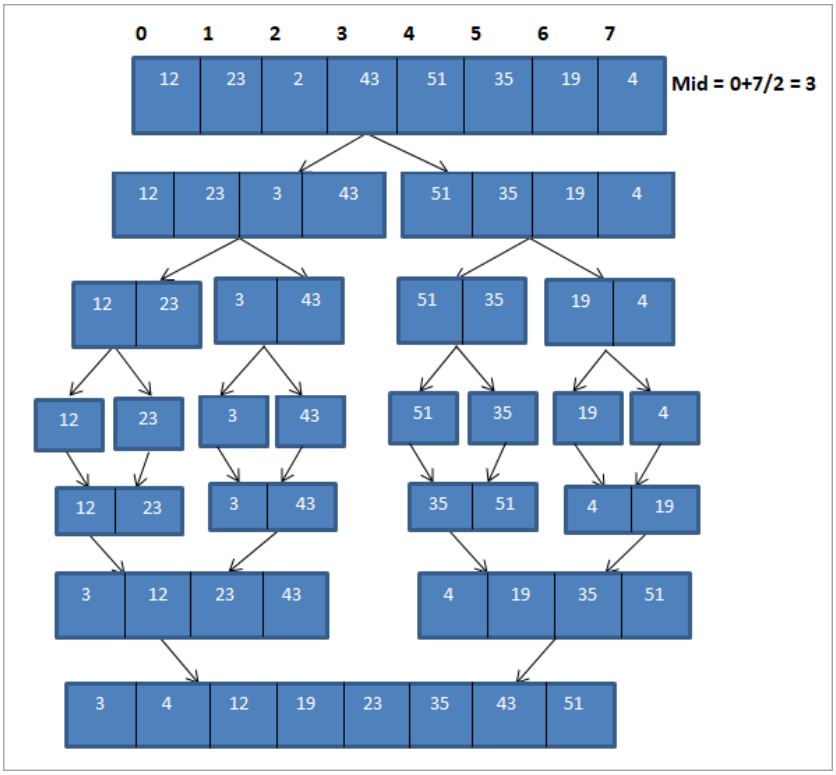
Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm.

The provided list is split in half equally, it calls for the two sections. And it then combines the two sorted parts.

The sub-lists are divided again and again into sections until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process.

The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Let us now illustrate the merge sort technique with an example.



The above illustration can be shown in a tabular form below :

Pass	Unsorted list	divide	Sorted list
1	{12, 23,2,43,51,35,19,4 }	{12,23,2,43} {51,35,19,4}	{}
2	{12,23,2,43} {51,35,19,4}	{12,23}{2,43} {51,35}{19,4}	{}
3	{12,23}{2,43} {51,35}{19,4}	{12,23} {2,43} {35,51}{4,19}	{12,23} {2,43} {35,51}{4,19}
4	{12,23} {2,43} {35,51}{4,19}	{2,12,23,43} {4,19,35,51}	{2,12,23,43} {4,19,35,51}
5	{2,12,23,43} {4,19,35,51}	{2,4,12,19,23,35,43,51}	{2,4,12,19,23,35,43,51}
6	{}	{}	{2,4,12,19,23,35,43,51}

As shown in the above representation, first the array is divided into two sub-arrays of length 4. Each sub-array is further divided into two more sub arrays of length 2.

Each sub-array is then further divided into a sub-array of one element each. This entire process is the “Divide” process.

Once we have divided the array into sub-arrays of single element each, we now have to merge these arrays in sorted order.

As shown in the illustration above, we consider each subarray of a single element and first combine the elements to form sub-arrays of two elements in sorted order. Next, the sorted subarrays of length two are sorted and combined to form two sub-arrays of length four each. Then we combine these two sub-arrays to form a complete sorted array.

Program : Write a program to implement merge sort in C++ language.

```
#include <iostream.h>
using namespace std;
void merge(int *,int, int , int );
void merge_sort(int *arr, int low, int high)
{
   int mid;
   if (low < high){
      //divide the array at mid and sort independently using merge sort
      mid=(low+high)/2;
      merge_sort(arr,low,mid);
      merge_sort(arr,mid+1,high);
      //merge or conquer sorted arrays
      merge(arr,low,high,mid);
   }
}
// Merge sort
void merge(int *arr, int low, int high, int mid)
{
   int i, j, k, c[50];
   i = low;
   k = low;
   j = mid + 1;
   while (i <= mid && j <= high) {
      if (arr[i] < arr[j]) {
         c[k] = arr[i];
         k++;
         i++;
      }
      else  {
         c[k] = arr[j];
```

```
            k++;
            j++;
        }
    }
    while (i <= mid) {
        c[k] = arr[i];
        k++;
        i++;
    }
    while (j <= high) {
        c[k] = arr[j];
        k++;
        j++;
    }
    for (i = low; i < k; i++)  {
        arr[i] = c[i];
    }
}
// read input array and call mergesort
void main()
{
    int myarray[30], num;
    cout<<"Enter number of elements to be sorted:";
    cin>>num;
    cout<<"Enter "<<num<<" elements to be sorted:";
    for (int i = 0; i < num; i++) { cin>>myarray[i];
    }
    merge_sort(myarray, 0, num-1);
    cout<<"Sorted array\n";
    for (int i = 0; i < num; i++)
    {
        cout<<myarray[i]<<"\t";
    }
}
```

Output :

```
Enter the number of elements to be sorted:10
Enter 10 elements to be sorted:101 10 2 43 12 54 34 64 89 76
Sorted array
2       10      12      34      43      54      64      76      89      101
```

2.8 Searching (Linear Search and Binary Search)

Linear Search :

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

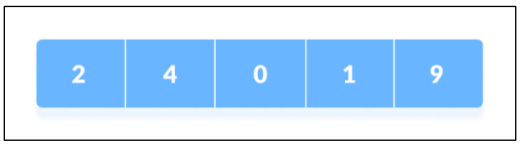
Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

Algorithm to implement linear search in C++ :

1. Read the item to be searched by the user.
2. Compare the search element with the first element in the list.
3. If they both matches, terminate the function.
4. Else compare the search element with the next element in the list.
5. Repeat steps 3 and 4 until the element to be search is found.

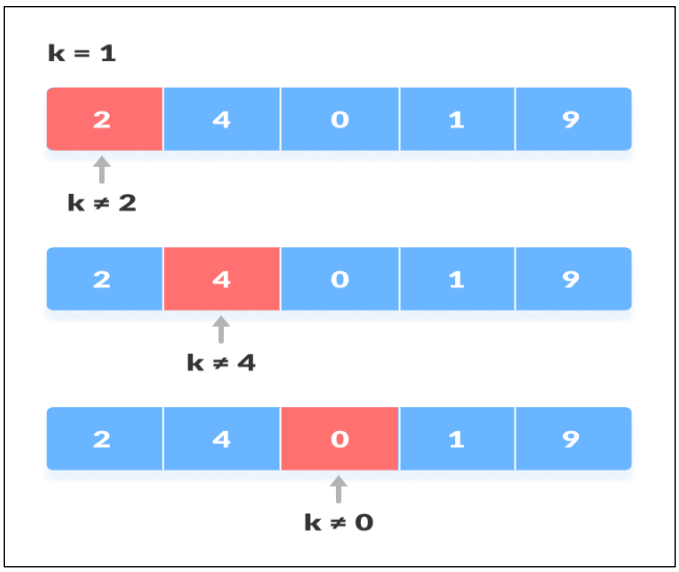
How Linear Search Works?

The following steps are followed to search for an element k = 1 in the list below.



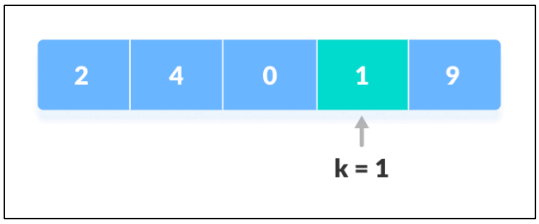
Array to be searched for

1. Start from the first element, compare k with each element x.



Compare with each element

2. If x == k, return the index



Element found

3. Else, return *not found*.

Program to implement linear search algorithm in C++

```
#include<iostream.h>
using namespace std;
void LinearSearch(int arr[], int len, int item){
   for(int i=0;i<len;i++){
      if(arr[i] == item){
         cout << item << " Found at index : " << i;
         return;
      }
   }
   cout << "Not found";
   }
void main() {
   int arr[] = {10, 5, 15, 21, -3, 7};
      // calculating length of array
   int len = sizeof(arr)/sizeof(arr[0]);
      // item to be searched
   int item = 21;
   LinearSearch(arr, len, item);
  getch();
}
```

Output :

21 Found at index : 3

Binary Search :

Binary search follows the divide and conquer approach in which the list is divided into two sections (halves), and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Conditions for when to apply Binary Search in a Data Structure :

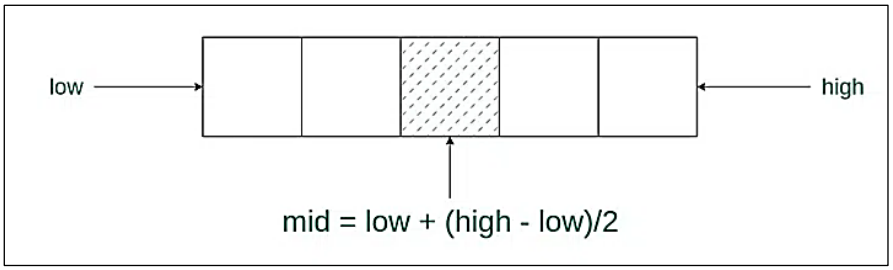
To apply Binary Search algorithm :

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

Binary Search Algorithm :

In this algorithm,

- Divide the search space into two halves (Sections) by **finding the middle index "mid"**.



Finding the middle index "mid" in Binary Search Algorithm

- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
- If the key is smaller than the middle element, then the left side is used for next search.
- If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

// Program: Binary Search in C++

```
#include <iostream.h>

int binarySearch(int array[], int x, int low, int high) {
   // Repeat until the pointers low and high meet each other
   while (low <= high) {
      int mid = low + (high - low) / 2;
```

```
        if (array[mid] == x)
            return mid;
        if (array[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

void main() {
    int array[] = {3, 4, 5, 6, 7, 8, 9};
    int x = 4;
    int n = sizeof(array) / sizeof(array[0]);
    int result = binarySearch(array, x, 0, n - 1);
    if (result == -1)
        std::cout << "Not found";
    else
        std::cout << "Element is found at index " << result;
   getch();
}
```

: QUESTIONS :

Short and Long Answer Questions:

1. Write algorithm for insertion sort. Explain with the help of example.
2. Write an algorithm to implement Bubble sort with suitable example.
3. Write an algorithm to implement Quick sort with suitable example.
4. Write an algorithm to implement insertion sort with suitable example.
5. Write an algorithm to implement selection sort with suitable example.
6. Write an algorithm for Linear search with suitable example.
7. Write an algorithm for binary search with suitable example.
8. Discuss the Algorithm of merge sort with an example.

***

3. Chapter

Stack

Contents:

(10 L, 15 M)

3.1. Definition and Concepts :

The linear lists and linear array permits one to insert and delete element at anywhere inside the list that is at the beginning, at the middle, at the end. There are certain things user needs to restrict insertions and deletions in such situation one of the data structures that is useful is stack.

A stack is linear data structure in which items may be added or removed only at one end figure 3.1. Shows example of such structure : A stack of rings, books, chairs, cups.

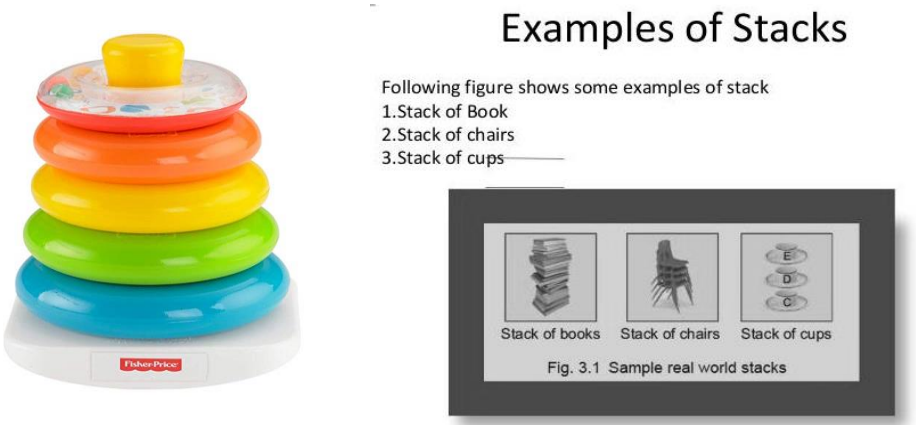


Fig. 3.1

3.2 Definition of Stack :

"A Stack is a linear data structure (dynamic data structure) in which is insertion and deletions of item are allowed. A stack is an ordered collection of items into which is new items may be inserted and from which items may be deleted at one end, called **top** of stack."

A stack data structure is base on the **Last - in –first – Out** (LIFO) i.e. insertion and deletion operation are performed at only one end of stack. The foremost accessible data in an exceedingly stack is at the top of the stack and least accessible data is at the bottom of the stack.

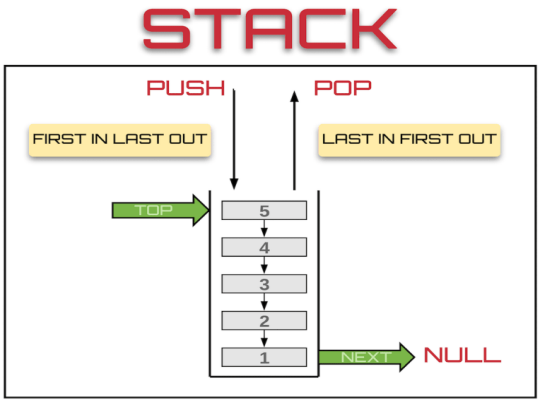


Fig. 3.2

New items may be written (insert) on the top of stack this is known as ***push*** operation. After pushing an element the size of stack increments. Top of stack points to newly inserted item.

Also item from top of the stack can be removed. This is known as ***pop*** operation. After removing top elements from stack, stack decrements and top of stack moves downwards to correspond to new highest elements. Figure 3.3 (a) and (b) show push and pop operation of stack.

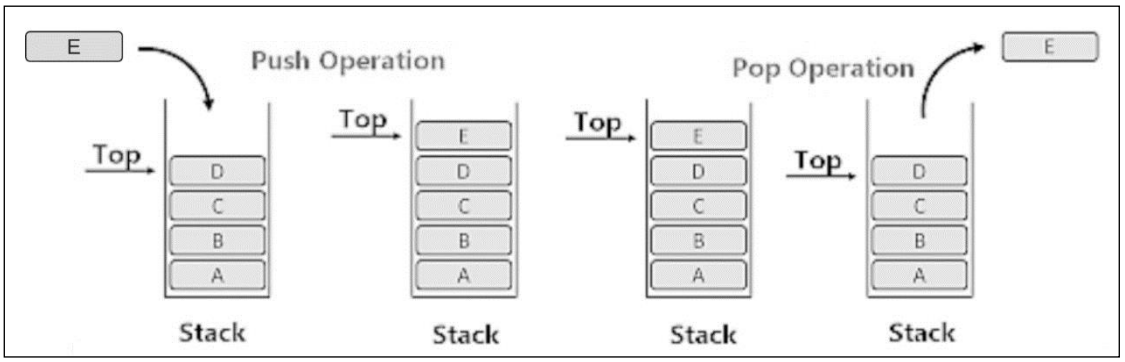


Fig. 3.3 (a) **(b)**

3.3 Representation of Stack - Static :

Stacks may be represented in the computer memory in various ways, by suggests that of a unidirectional list or a linear array. Unless otherwise declared, stacks are maintained by a array STACK; a pointer variable TOP, which contains the area of the top component of the stack; and a variable MAXSTK which gives the greatest number of components that can be held by the stack. The condition TOP = 0 or TOP = NULL will demonstrate that the stack is void.

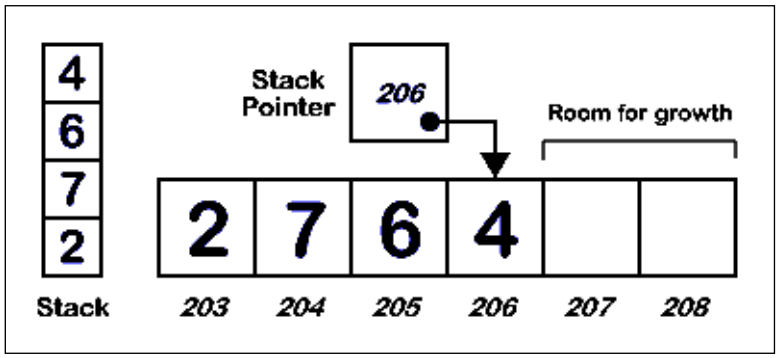


Fig. 3.5 Memory Representation of Stacks

3.4 Operations of Stack :

Basic operation required to manipulate a stack are: - Push, Pop, Peep, Change.

1. PUSH Operation :

The process of inserting an item into the stack is known as **PUSH** operation. In order to insert an item into stack first we have to check whether free space is available in the stack or not. If the stack is full then we cannot insert an item into stack. If value of TOP variable is greater than or equal to SIZE-1 then Stack is full. This condition is known as **"Overflow condition".**

If stack is not overflow then we can insert an item into stack. First we have to increment the value of variable TOP by one and then insert an item into stack.

Following algorithm shows push operation :

Step 1 : [Check For Stack Overflow]
If TOP >=SIZE- 1 then
Write "Stack is Overflow"

Step 2 : [Increment TOP Pointer]
TOP=TOP+1

Step 3 : [Insert Element in the Stack]
STACK [TOP] =X

Step 4 : [Exit]

The first step of this algorithm checks an overflow condition. If such a condition exists, then the insertion cannot be performed and an appropriate error message results.

2. POP Operation :

The process of deleting an item from stack is known as POP operation. In order to delete an item from stack first we need to check weather stack is empty or not. If stack is empty then we cannot delete an element from stack. If stack isn't underflow then we are able to delete topmost item from stack. After deleting topmost item from stack, we need to decrement the value of TOP by one, so that it can point to the next top most item in the stack.

Following algorithm shows pop operation :

Step 1 : [Check for Underflow on Stack]
If TOP= -1 then
Write "Stack is Underflow"
Exit

Step 2 : [Decrement Stack Pointer]
TOP=TOP - 1

Step 3 : [Return former top element of Stack]
Return (S [TOP+1])

Step 4 : [Exit]

3. PEEP Operation :

This process returns the value of the i^{th} element from the top of the stack. Without removing from the stack

Following algorithm shows Peep Operation :

Step 1 : [Check for stack underflow]
Write "Stack is Underflow on Peep"

Step 2 : [Return i^{th} element from top of stack]
Return (S [TOP – I +1])

Step 3 : [Exit]

4. CHANGE Operation :

This process changes the value of the i^{th} element from the top of the stack to the value contained in X.

Following algorithm shows Change operation

Step 1 : [Check for stack underflow]
If TOP- I +1 ≤ 0
Write "Stack Underflow on Change"

Step 2 : [Change i^{th} element value from top of stack]
S [TOP-I +1] =X

Step 3 : [Exit]

3.5 Applications of Stack :

Stack is generally used in many applications of computer some of the real use of stack are :

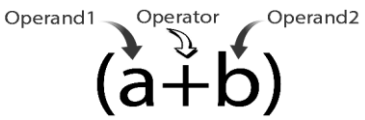
1. Compiler design is the evaluation of arithmetic expressions. Here the compiler uses a stack to translate an input arithmetic expression into its corresponding object code. Stack is used to convert polish notation from one form to another form.
2. Reversing a string
3. Checking correctness of Nested parenthesis
4. Simulating recursion
5. Parsing of computer program
6. Backtracking of algorithm
7. Processing of subprogram calls

In general there are 3 kinds of expressions available on the position of the operators & operands.

1) Infix Expression : It is the common notation used for representing expressions.

"In this expression operator is in between operands, the expression is known as Infix expression"

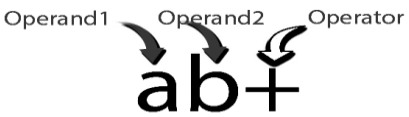
Example :



2) Post Fix Expression : (Reverse Polish Notation)

"In this expression the operator is put after the operands".

Example :



3) Prefix Expression : (Polish Notation)

"In this expression the operators is before operands is called Prefix expression"

Example :

All the infix expression will be converted into post fix expression with the help of stack. The stack will be helpful in solving the postfix expressions also.

Expression Representation :

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
(A + B) * C	* + A B C	A B + C *
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +

3.5.1 Infix to Postfix Conversion :

To formalize the conversion method, we will assume simple arithmetic expressions containing the +,-,*, / and ^ (exponentiation) operators only (i.e. without unary operators, Boolean operators and relation operators). The expression may be parenthesizedornot parenthesized.

First we have to append the symbol ‘)’as the delimiter at the end of given infix expression and initialize the stack with ‘(‘. These symbols ensure that either the input or the stack is exhausted.

Our next step is iterative: read one input symbol at a time and decide whether it has to be pushed onto the stack or not. This decision will be governed by following table.

Table : In-Stack and In – Coming Priorities of symbols

Symbol	In-Stack PriorityValue	In-Coming PriorityValue
+ -	2	1
*/	4	3
^	5	6
Operand	8	7
(	0	9
)	-	0

From the table, it can be noted that for a symbol we have considered two priority values, viz. in -stack priority and in-coming priority values. A symbol will be pushed onto the stack if its in-coming priority value is greater than the in-stack priority value of the top most elements. Similarly a symbol will be popped from the stack if its in-stack priority value is greater than or equal to the in-coming priority value of the incoming elements.

The rules to be remembered during infix to postfix conversion are

1. Parenthesize the expression starting from left to right.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parent he sized. For example in expression A+B*C, B*Cis parenthesized first before A+B

3. The sub-expression (part of expression) which has been converted into postfix expression is to be treated as single operand.
4. Once the expression is converted to postfix form remove the parenthesis.

Let us consider some example for converting infix expression to postfix, we will follow the above said four rules.

Algorithm for Converting Infix Expression to Postfix Form :

Assume that Q is an arithmetic expression written in infix expression.

This algorithm finds the equivalent postfix expression P.

1. Push "(" on to STACK, and add ")" to the end Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each elements of Q until the STACK is empty.
3. If an operand is encountered, add it to P.
4. If a left Parenthesis is encountered, push it into STACK.
5. If an any operator is encountered then:
   a) Repeatedly pop from stack and circle the * as operator to P each operator (on the top of STACK)
      Which has the same precedence as or higher precedence than operator?
   b) Add operator to STACK
      [End of the If Structure]
6. If a right parenthesis is encountered, then :
   a) Repeatedly pop from STACK and add to P each operator (on the top of STACK until a left parenthesis is encountered.)
   b) Eliminate the left parenthesis .[Don't add the left parenthesis to P]
   c) [End of If Structure]
   d) [End of Step 2 loop]
7. Exit.

Example 1) Convert the given expression in to postfix form for A+B*C

Solution : Given By A+B*C is Infix form

A+B*C	**Infix form**
A+ (B*C)	Parenthesized expression
A+ (BC*)	Convert the multiplication
A (BC*) +	Convert the addition
ABC*+	**Postfix form**

Example 2) Convert the expression (A+B)/(C-D) to postfix form

Solution : In this expression the brackets are already specified .

(AB+) / (CD-)

T/S	Where T= (AB+) and S=(CD-)
TS/	Put original value
AB+CD- /	Postfix Expression

Example 3) Give the post form for (A+B)*C/D

Solution : (AB+)*C/D

T * C/D	Where T=(AB+)
(T* C) /D	
(TC*) / D	
S /D	Where S=TC*
SD/	
TC * D /	
AB+C*D/	**Postfix Expression**

Example 4) Convert A * (B + C) * D to postfix notation.

Move	Current token	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	*	*	A B C + *
9	D	*	A B C + * D
10		Empty	

Example 5) Convert the following expression A* B + C / D to postfix form

Solution : In this particular example the * operator is encountered during left to right evolution, which shares an equal precedence to operator. When this happen (two operators with same precedence comes in same expression), the operator that is encountered first while evaluating from left is first parenthesized. Hence the above expression will be interpreted as:-

A * B + C / D	Given expression
(A* B) + C / D	Parenthesized

(AB *) + C / D

(T) + C / D	T=AB*
(T) + (C / D)	Parenthesized
(T) + (CD /)	
T + S	S=CD /
TS +	
Put original value of T & S	
AB* CD /	**Postfix Expression**

Example 6) Convert the following expression A+[(B+C) + (D+E) * F] / G to postfix form

Solution : A+ [(BC +) + (DE+) * F / G

A + [(BC +) + (DE + F *) / G

A + [(BC + (DE + F * +) / G

A + [BC + DE + F * + G /]

ABC + DE + F * + G / +	**Postfix Expression**

Example 7) Convert the following expression A+ B / C – D to postfix form

Solution :

A + (B / C) – D

A+ (BC /) – D

A + T – D	T= BC /
(A+ T) – D	
(AT+) – D	
S – D	S= (AT+)
SD-	Put original value
AT + D -	
ABC / + D-	**Postfix Expression**

Example 8) Convert the following expression (A+ B) / (C – D) to postfix form

Solution : In this expression the brackets are already specified. Therefore the conversions look like

(AB +) / (CD-)

T/ S	T= (AB+) and S= (CD-)
TS /	
AB + CD- /	**Postfix Expression**

Example 9) Convert the following expression (A+ B) *C / D to postfix form

Solution : In this expression brackets are already specified there for

(AB+) * C / D

T * C/ D T= AB+

(T * C) / D

(TC *) / D

S / D S=(TC*)

SD/

Put original value

TC* D /

AB + C * D/ Postfix Expression

Example 10) Convert the following expression (A+ B) *C / D+ E ^ F / G to postfix form

Solution :

(AB+) * C / D + E ^ F / G

T * C /D + (E ^ F) / G T= AB+, (^has the highest priority)

T * C / D + (EF ^) / G

T * C / D + S / G S= (EF ^)

(T * C) / D + S / G

(TC *) / D + S / G

Q/ D + S/ G Q=(TC*)

(Q / D) + S / G

(QD/) + S / G

P + S/ G P= (QD/)

P+ (S/ G)

P + (SG /)

P + O O= (SG/)

PO+

Now we will expand the expression PO+

PO+

QD/ O+

TC * D / O+

AB+ C* D/SG/+

AB+ C * D/ EF^ G / + Postfix expression

Example 11) Convert the following expression A+ [(B +C) + (D+ E) * F]/G to postfix form

Solution : A+ [(BC+) + (DE+) * F]/ G

A + [T + S * F] / G where T=(BC+) and S=(DE+)

A+ [T +(S * F)] / G

A+ [T + (SF*)]/ G

A+ [T + Q] / G where Q= (SF*)

A+ (TQ+) / G

A+ P / G where P= (TQ+)

A+ (PG/)

A + N where N= (PG/)

AN +

Expanding the expression AN+

APG/+

ATQ + G / +

ABC + Q + G / +

ABC + SF * + G / +

ABC + DE + F * G / +

3.5.2. Conversion of infix expression into Prefix expression :

Suppose Q is an arithmetic expression written in infix form. The algorithm finds equivalent prefix expression P.

Step 1. Push ")" onto STACK, and add "(" to begin of the Q

Step 2. Scan Q from right to left and repeat step 3 to 6 for each element of Q until the STACK is empty

Step 3. If an operand is encountered add it to P

Step 4. If a right parenthesis is encountered push it onto STACK

Step 5. If an operator is encountered then:

- **a.** Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has same or higher precedence than the operator.
- **b.** Add operator to STACK

Step 6. If left parenthesis is encountered then

- **a.** Repeatedly pop from the STACK and add to P (each operator on top of stack until a left parenthesis is encountered)
- **b.** Remove the left parenthesis

Step 7. Exit

Consider the following arithmetic expression written in infix notation

Q : (5 + 2) * 3

First push ")" onto STACK and add "(" at the beginning of Q.

Following table demonstrate the elements of STACK and generate string P at each step when expression in Q scanned from right to left.

Q : ((5 + 2) * 3

Symbol Scanned	STACK	Expression B
1) 3	)	3
2) *	) *	3
3))	) *)	3
4) 2	) *)	3 2
5) +	) *) +	3 2
6) 5	) *) +	3 2 5
7) (	) *	3 2 5 +
8) (		3 2 5 + *

P = 3 2 5 + *

Reverse the string the prefix expression is P = * + 5 2 3

3.6 Polish Notation or Evaluation of Postfix expression :

Suppose P is an arithmetic expression written in postfix notation.

The following algorithm evaluates P which uses STACK to hold operands.

Algorithm POSTFIX_EVALUATE.

This algorithm finds the value of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P.
2. Scan P from left to right and repeat step 3 and 4 for each element of P until sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator "x" is encountered, then:
   a. Remove the two top elements of STACK, where A is the top element and B is the next-to-
   b. top element.
   c. Evaluate B "x" A
   d. Place the result of (b) back on STACK

   [End of If structure]
5. Set value equal to top element of STACK
6. Exit

Symbol Scanned	Stack
(1) 2	2
(2) 5	2 5
(3) 3	2 5 3
(1) 2	2
(2) 5	2 5
(3) 3	2 5 3
(4) *	2 15
(5) 12	2 15 12
(6) 2	2 15 12 2
(7) 2	2 15 12 2 2
(8) ^	2 15 12 4
(9) /	2 15 3
(10) 5	2 15 3 5
(11) *	2 15 15
(12) -	2 0
(13) +	2
(14))	

Example :

Consider the following arithmetic expression P written i

n postfix notation :

P : 2 5 3 * 12 2 2 ^ / 5 * - +

First we add a sentinel right parenthesis at the end of P to obtain

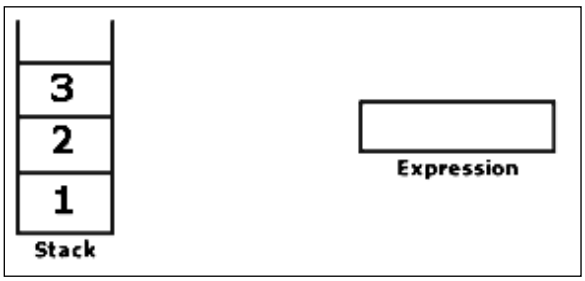
P : 2 5 3 * 12 2 2 ^ / 5 * - +)

The final number in STACK is 2, which is assigned to "value" when sentinel ")" is scanned, is the value of P.

Postfix Evaluation :

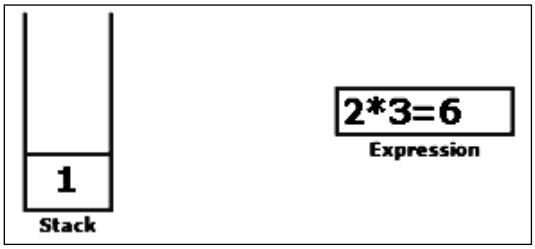
Example 1) Evaluate the Postfix expression 123*+4 –

Solution : Postfix String 123* + 4 –

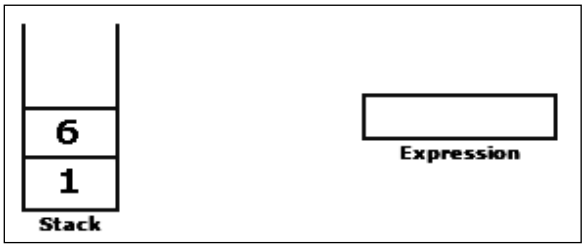


Initially the stack is empty. Now the first three Operand is scanned are 1, 2 and 3. Now they will pushed into the stack in FIFO manner.

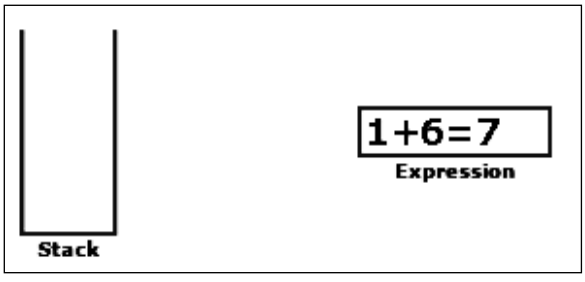
Next character scanned is "* ", which is operator. Thus we pop the top the two elements from the stack and perform the multiplication "* "operation with the two operands. The second number will be the first element that is popped.



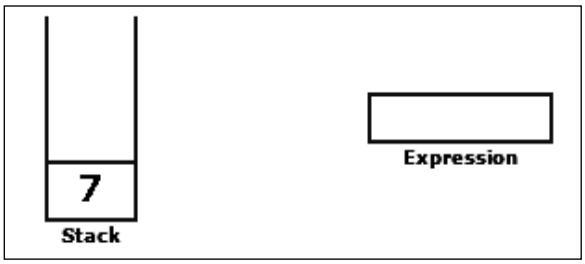
Now the value of the multiplication expression (2* 3) that has been evaluated (6) is pushed into the stack.



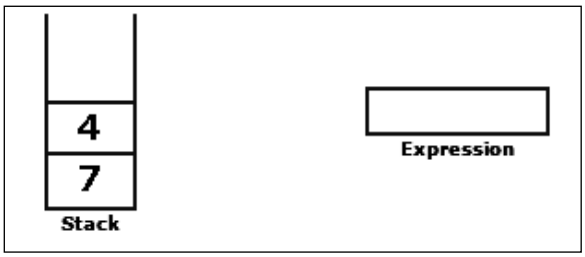
Now, next character scanned is "+ ", which is an also one of the operator so, we pop the top two elements once again from the stack and perform the addition "+ "operation.



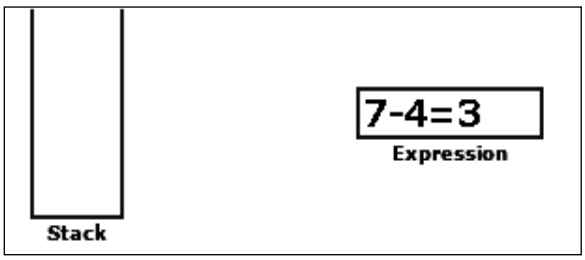
Now, the value of the addition expression (1 + 6) that has been evaluated (7) is pushed into the stack.



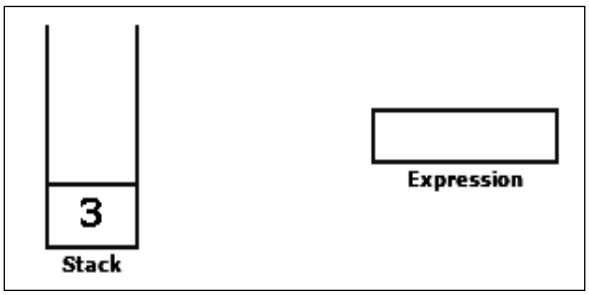
Now, next character is scanned which operand“ 4” ,which is added to the stack



Now, next character scanned is “– “, which is also one of the operator. Now , we pop the top two elements from the stack and perform the minus “– “operation. The second operand will be the first element that is popped.



The value of the minus expression is evaluated (7- 4=3) and push (3) in stack



Now, since all the character are scanned, the remaining element in the stack (there will be only one element remain in the stack) will be returned. Then output

Given Postfix String : 123*+4-

Result : 3

Example 2) Evaluate the Postfix expression 10 2 8 * + 3 –

Solution : Given Postfix string: 10 2 8 * + 3 –

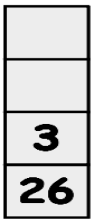
Initially the stack is empty. Now the first three Operand is scanned are 10, 2 and 8. Now they will pushed into the stack in FIFO manner.

Next character scanned is multiplication "* " then pop of top of two element and multiplication will be done.

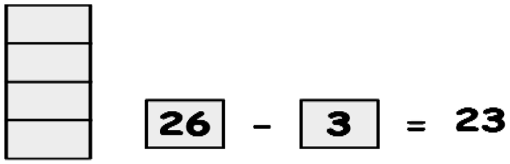
Now the value of the multiplication expression (2* 8) that has been evaluated (16) is pushed into the stack.

Now, next character scanned is "+ ", which is an also one of the operator so, we pop the top two elements once again from the stack and perform the addition "+ "operation. The second operand will be the first element that is popped.

We see the next number 3 Push (3) into the stack



Now, next character scanned is "– ", which is also one of the operator. Now, we pop the top two elements from the stack and perform the minus "– "operation. The second operand will be the first element that is popped.



Since all the character are scanned, there is no remaining element in the stack

Given Postfix String: 10 2 8 * + 3 –

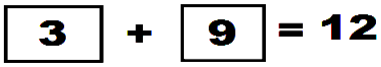
Result: 23

Example 3) Evaluate the Postfix expression 5 9 3 + 4 2 * * 7 + *

Solution: Given Postfix expression 5 9 3 + 4 2 * * 7 + *

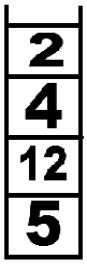
Initially the stack is empty. Now the first three Operand is scanned are 5, 9 and 3. Now they will pushed into the stack in FIFO manner.

Now, next character scanned is addition "+ ", which is operator. Thus we pop the top the two elements from the stack and perform the addition "+ "operation with the two operands. Then second operand will be the first number that is popped.



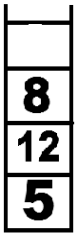
Now the value of the addition expression (3 + 9) that has been evaluated (12) is pushed into the stack.

We see the next number 4, 2 then Push (4) and Push (2) in the stack,



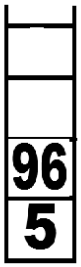
Now, next character scanned is multiplication "* ", which is operator. Thus we pop the top the two elements from the stack and perform the multiplication "* "operation with the two operands. Then second operand will be the first number that is popped.

Now the value of the multiplication expression (2 * 4) that has been evaluated (8) is pushed into the stack. Then stack will be look like



Now, next character scanned is " * ", which is also one of the operator. Perform the multiplication "*"operation.

Next value of the multiplication expression (12 * 8=96) push (96) in the stack



We see the next number 7 then Push (7) into the stack,

Now, next character scanned is addition "+ ", which is operator. Thus we pop the top the two elements from the stack and perform the addition "+ "operation with the two operands. Then second operand will be the first number that is popped.

96+7=103

Now the value of the addition expression (96 + 7) that has been evaluated (103) is pushed into the stack. Then stack

Next character scanned is multiplication "* ", then

103* 5=515

Then finally

Given Postfix String: 5 9 3 + 4 2 * * 7 + *

Result: 515

Example 4) Evaluate the Postfix expression 2 3 4 * 6 / +

Solution :- Given Postfix expression 2 3* 4 *

Read Symbol	**Stack**	**Action**
2	2	Push(2)
3	3	Push(3)
4	4	Push(4)
*	2 12	**Pop(4), Pop(3) (4*3=12) Push (12)**
6	2 12 6	Push(6)
/	2 2	**Pop(6), Pop(12) (12/6=2) Push(2)**
+	4	Push(4)
Final Result	**4**	

Example 5) Evaluate the Postfix expression 345 * +

Read Symbol	**Stack**	**Action**
3	3	Push(3)
4	4	Push(4)
5	5	**Push(5)**
*	3 20	**Pop(5), Pop(4), (5*4=20) Push(20)**
+	23	**Pop(3), Pop(20), (3 +20 =23) Push(23)**
Final Result	**23**	

Example 6) Evaluate the Postfix expression 72+3 *

Read Symbol	**Stack**	**Action**
2	2	Push(2)
7	7	Push(7)
+	9	**Pop(7),Pop(2),(7+2=9),Push(9)**
3	9 3	Push(3)
*	27	**Pop(3) , Pop (9), (9*3= 27) Push(27)**
Final Result	**27**	

Example 7) Evaluate the Postfix expression 2 3 * 4 *

Read Symbol	Stack	Action
2	2	Push(2)
3	3	Push(3)
*	6	**Pop(3),Pop(2),(3*2=6),Push(6)**
4	6 4	Push(4)
*	24	**Pop(4) , Pop (6), (6*4= 24) Push(24)**
Final Result	**24**	

Example 8) Evaluate the expression 2 3 4 + * 5 *

Move	Current Token	Stack (grows toward left)	
1	2		2
2	3		3 2
3	4		4 3 2
4	+		7 2
5	*		14
6	5		5 14
7	*		70

Evaluation of Prefix expression :

Suppose P is an arithmetic expression written in prefix notation.

The following algorithm evaluates P which uses STACK to hold operands.

Algorithm PREFIX_EVALUATE.

This algorithm finds the value of an arithmetic expression P written in prefix notation.

1. Reverse the given prefix expression.
2. Add ")" at end of P.
3. Scan P from left to right and repeat step 3 and 4 for each element of P until sentinel ")" is encountered.
4. If an operand is encountered, put it on STACK.
5. If an operator "x" is encountered, then:
   a. Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   b. Evaluate A "x" B
   c. Place the result of (b) back on STACK
   [End of If structure]

6. Set value equal to top element of STACK
7. Exit

3.7 Recursion using Stack :

Recursion is one of the applications of stack. It is an important facility in many programming language.

There are many problems whose algorithmic description is best described in a recursive manner.

Recursion is a function invoking an instance of itself, either directly or indirectly. Recursion is a programming technique and some programming tasks are naturally solved with the use of recursion, which can be considered an advanced form of flow of control. Recursion is an alternate to iteration. Recursion defines a problem in terms of itself. A recursive solution repeatedly divides a problem into smaller sub problems until a directly solvable sub problem is reached. Once we obtain the solution of to solvable sub problem, we feed it back into the next large sub problem for its solution. This process continues until we solve the original problem.

Let the original problem be P.P is redefined in terms of sub problem P1 and P1 is again redefined in terms of P2 and so on. Let the last sub problem be Pn can be used to solve the sub problem Pn-1 . This solution is fed back into Pn-2,….., P2, P1 until we finally have solved original problem P.

As an the most common example is a factorial in which n!= n(n-1)(n-2)…… 2*1= n(n-1) is calculated. Here we obtain the value of n! by taking the number n and multiplying it by(n-1)!..

: QUESTIONS :

Short and Long Answer Questions:

1. Define Stack.
2. Give real world example of stack
3. List the operations on stack.
4. List the application on stack.
5. Define push operation on stack.
6. Define pop operation on stack.
7. Define peep operation on stack.
8. When stack is said to be overflow?
9. Give definition of infix, prefix and postfix notation.
10. Identify the types of expression whether it is infix, prefix or postfix.
    a. 4,2$3*3-8,4/1,1+/+ b. PQ+R+-S↑UV+*

11. Explain Stack with its example.
12. Explain the operation performed on Stack.
13. Explain Push operation with algorithm.
14. Explain Pop operation with algorithm.
15. Explain Peep operation with algorithm.
16. Explain application of Stack.
17. Explain Evaluation of expressions on stack.
18. Evaluate following expression.
    a. 10+3-2-8/2*6-7 b. (12-(2-3)+10/2+4*2)
19. Convert following infix expression to postfix expression :
    a. ((a+b)/d-((e-f)+g) b. 12/3*6+6-6+8/2 11.
20. Convert following infix expression to prefix expression :
    a. ((a+b)/d-((e-f)+g) b. 12/3*6+6-6+8/2 12.
21. Convert the following postfix expression to an infix expression :
    a. 4,2$3*3-8,4/1,1+/+ b. PQ+R+-S↑UV+*

***

4. Chapter

Queues

Contents:

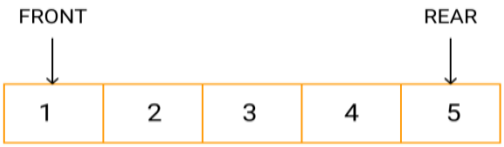
(10 L, 15 M)

4.1 Introduction :

Queue is a non-primitive linear data structure. It is an ordered list of elements in which the addition of new elements can take place from one end called the rear and the deletion of existing elements is done from another end, called as front. This means that elements are removed from a queue in the same order as that in which they were inserted into the queue. As the element inserted first in the queue is deleted first from the queue queues are also called first in first out (FIFO) lists or first come first served (FCFS).

4.2 Definition :

Queues may be represented by a linear array with two pointer variables. The front is the pointer containing the location of the front element of the queue and the Rear contains the rear element of the queue. Whenever a new element is added to the queue rear pointer is increased by one and when an existing element is deleted from the queue, the front pointer is increased by one.



- A good real-life example of a queue is the queue of people waiting at the railway station to get a ticket. Each new person who will come joins at the end of the queue. The person who comes first gets the ticket first.
- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular process to occur.

4.3 Implementation / Static Representation of Queue :

Static representation of queue can be obtained using arrays. A dimensional array is used to represent a queue. Suppose the size of the queue is N and two pointers FRONT and REAR contain integer values to represent two ends of the queue. Each time data is added REAR is incremented and when data is removed FRONT is incremented. Condition FRONT=0 will indicate that the queue is empty and REAR=N represents that queue is full. An empty queue is indicated by setting FRONT and REAR values to 0.

The array has a few limitations. Once the size of an array is declared, its size cannot be modified during runtime or at the time of executing the program. If we store less number of elements than the declared size of the array then memory is wasted and if we want to store more elements than the declared size then the array cannot be expanded. Array is only suitable when prior we know exact number of elements to be stored.

Queue can be declared as array of variable by following statement

int queue[max_queue_size]; or char stack[max_queue_size];

max_queue_size is maximum number of elements inserted into queue.

4.4 Operations Performed on Queue :

a) **Insert :** This is the process of addition of new element onto the queue called as insertion operation. When an element is added it will add to the rear end of queue. At that time

rear value is increased by one and then the element is inserted. Before inserting the element one must test whether there is room in the queue for the new item. If there is no space to store element in queue means queue is full or overflow.

b) **Delete :** This is the process of deletion of an existing element from the front end of queue called as delete operation. When as element is deleted from front end of queue at that time element is deleted and then front value is increased by one. Before deleting the element one must test whether there is any element in the queue or not. If there is no element in queue means queue is underflow or empty.

Algorithms on different operation performed on queue:

Algorithm to insert an element in queue : In first step overflow condition is checked. If queue becomes overflow then insertion cannot be performed and appropriated error message is return.

Procedure QINSERT(Q, N, F, R, ITEM) :

This procedure inserts an ITEM onto rear position of queue. Queue is represented by vector Q containing N elements. F and R are the pointers pointing to the front and rear elements of Q. Initially F&R are set to zero.

1. [Is queue already full?]
   If R >= N
   Then write("Queue Overflow")
   Return
2. [Increase rear by 1]
   R ←R + 1
3. [Insert ITEM at rear position]
   Q[R] ← ITEM
4. [Is front pointer property 1]
   If F = 0
   Then F ← 1
   Return

Algorithm to delete an element from queue – Initially underflow condition is checked. If queue becomes empty then deletion of element cannot be performed and appropriated error message is return.

Procedure QDELETE (Q, F, R)

This procedure removes an ITEM from front pointer of queue Q and returns the element. F&R are the pointers pointing to front and rear position of queue resp.

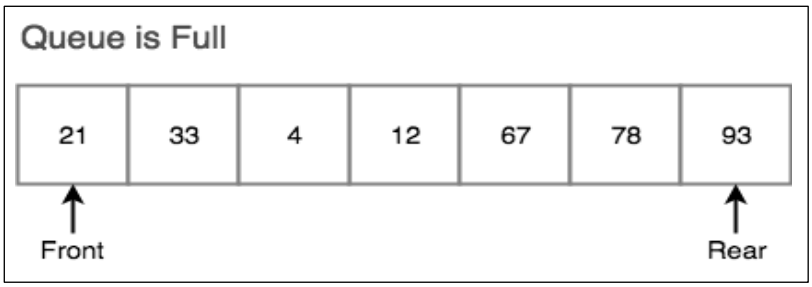
1. [Is queue empty?]
   If F = 0
   Then write("Queue Underflow")

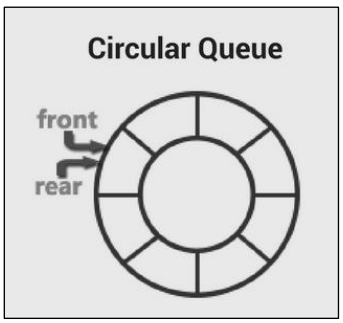
Return

2. [Assign front element to ITEM]
   ITEM ← Q[F]
3. [Increase F by 1]
   F ←F + 1
4. [Finished]
   Return

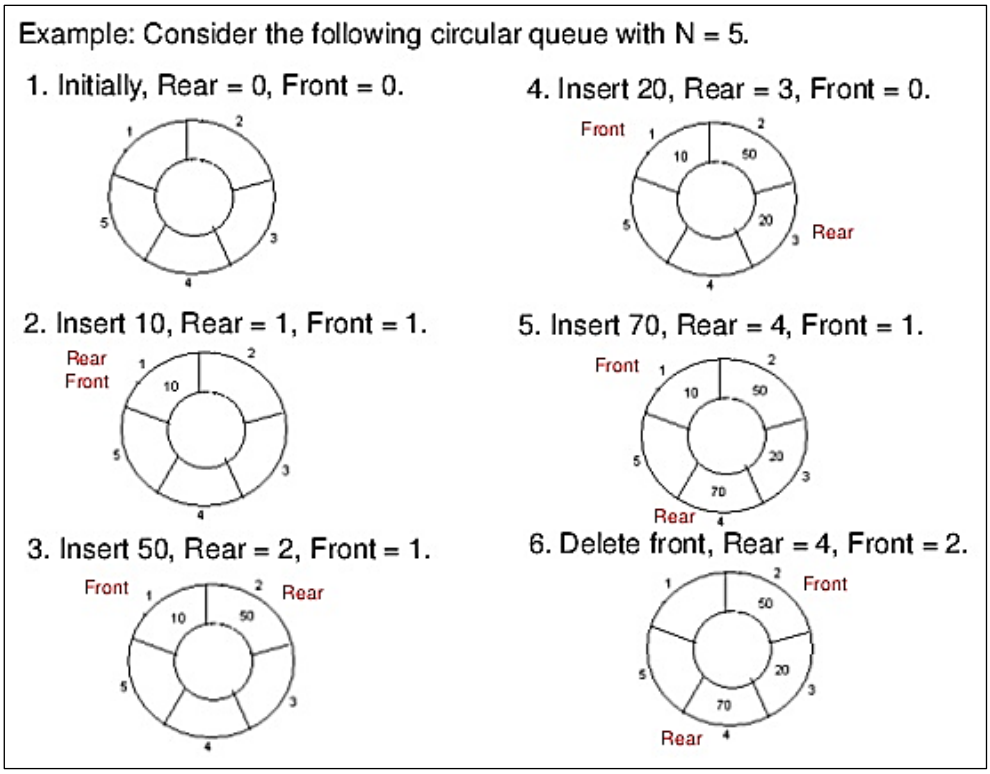
4.5 Circular Queue :

In linear queue, insertion take place from rear end and deletion take place from front end. When some elements are deleted from queue then empty space is created there and even if the queue has empty cells we cannot insert any new element there. Means space is not utilize for further storage. This problem is solved in case of circular queue. Even if the rear is full but there is space at front end, then the element can be inserted in the beginning nodes until queue overflows. In circular queue after pointing the last element of queue, rear points to the first element to make circle.





Algorithms to perform different operation on circular queue :



Algorithm to insert an element in circular queue : In first step overflow condition is checked. If queue becomes overflow then insertion cannot be performed and appropriated error message is return.

Procedure CQINSERT(Q,N,F,R,ITEM) :

This procedure inserts an ITEM onto rear position of queue. Queue is represented by vector Q containing N elements. F and R are the pointers pointing to the front and rear elements of Q. Initially F&R are set to zero.

1. [Is queue already full?]
   If F=1 and R=N or if F=R+1
   Then write ("Circular Queue Overflow")
   Return
2. [Find new value of R]
   If F=0
   Then F=1 and R=1
   Else If R=N

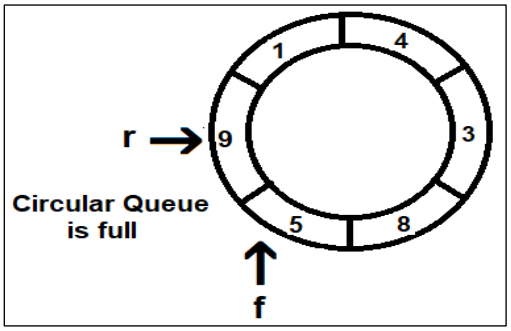
Then R ← 1

Else R ←R + 1

3. [Insert ITEM at rear position]

   Q[R] ← ITEM

4. Return



Algorithm to delete an element from circular queue : Initially underflow condition is checked. If circular queue becomes empty then deletion of element cannot be performed and appropriated error message is return.

Procedure CQDELETE (Q, F, R) :

This procedure removes an ITEM from front pointer of queue Q and returns the element. F&R are the pointers pointing to front and rear position of queue resp.

1. [Is queue empty?]

   If F = 0

   Then write("Circular Queue Underflow")

   Return

2. [Assign front element to ITEM]

   ITEM ← Q[F]

3. [Find new value of front]

   If F = R

   Then F =0 and R= 0

   Else If F = N

   Then F← 1

   Else

   F ←F + 1

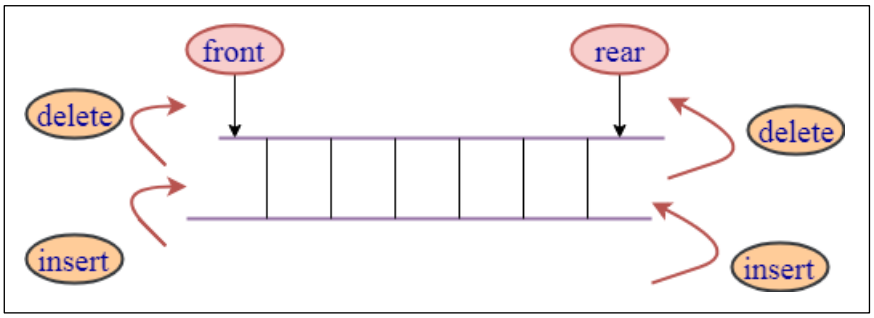
4. [Finished]

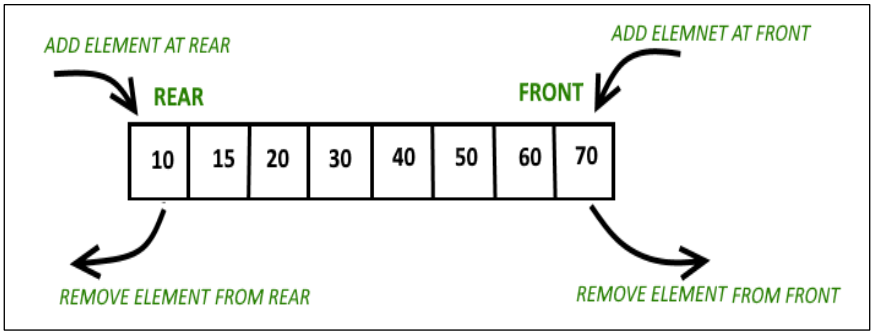
   Return

4.6 De-queue and Priority Queue :

DeQue :

Deque is a special type of data structure in which insertions and deletions of elements will be done either at the front end or at the rear end of the queue called deque or double ended queue. This differs from linear queue where elements can added to one end and remove from the other end. Following figure shows the representation of deque.





Basic operations that can be performed on dequesare :

a) Insert an item from front end
b) Insert an item from rear end
c) Delete an item from front end
d) Delete an item from rear end

The DeQue can be constructed by using array or link list.

There are two variations in deque,

1. **Input restricted deque** : Input restricted deque allows insertions at only one end of array or list but deletions allowed at both ends.
2. **Output restricted deque** : Output restricted deque allows deletions at only one end of the array or list but insertions allowed at both ends.

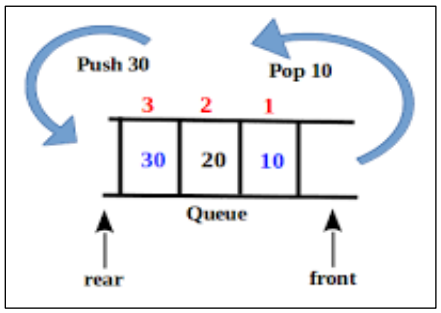
Priority Queue :

A special type of queue in which each element is assigned a priority and the order in which elements are deleted and processed follow following rules

a) An element with highest priority is processed first before any element of lower priority.

b) Two or more elements with same priority are processed according to the order in which they were added to the queue.

Such type of queue is called priority queue.



There are two types of priority queues :

1. Ascending Priority Queue
2. Descending Priority Queue

1. Ascending priority queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed first.
2. Descending priority queue allows deletion of only the largest item.

Stack may be viewed as descending priority queue.

Queue may be viewed as ascending priority queue.

Examples of priority queues are :

a) In CPU scheduling shortest job is given the highest priority to process over the longer jobs.

b) An important job is given the highest priority over a routine type job.

4.7 Applications of queues :

i) Used in time sharing system in which programs with the same priority form a queue while waiting to be executed.

ii) Used in simulation related problem.

iii) When jobs are submitted to a networked printer, they are arranged in order of arrival, ie. Jobs are placed in a queue.

iv) Checking string of language

v) Simulation (Queuing Theory)
vi) Input output buffers.
vii) Graph Searching.

: QUESTIONS :

Short and Long Answer Questions:

1. How is the queue different from stack?
2. What are the limitations of a queue?
3. What is circular queue? Explain with suitable example.
4. What is queue? How it is implemented in memory of computer?
5. Differentiate queue and circular queue.
6. What is priority queue?
7. Write an algorithm for insertion and deletion operations on circular queue.
8. What are the advantages of circular queue over linear queue?
9. Define Queue.
10. Give real world example of Queue.
11. List the operations on Queue.
12. List the application on Queue.
13. Define Insertion operation on Queue.
14. Define Deletion operation on Queue.
15. Define Circular Queue.
16. List out limitation of linear queue.
17. Write algorithm to insert element into circular queue.
18. What is Deque? Explain with example.
19. Define priority queue.
20. Explain the operation performed on Queue.
21 Explain insertion operation for queue with algorithm.
22. Explain Deletion operation for queue with algorithm.
23. Explain application of Queue.
24. Write short note on Deque.
25. Write short note on Priority Queue.
26. What are the difference between stack and queue.
27. How to overcome limitation of linear queue? Explain in detail.

***

5. Chapter

Linked List

Contents:

(10 L, 15 M)

5.1 Introduction :

Linked lists were developed in 1955-1956, by Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation and Carnegie Mellon University as the primary data structure for their Information Processing Language (IPL).

A linked list is a linear collection of data elements in which each element points to the next element.

A linked list is a linear data structure consisting of a collection of nodes which together represent a sequence.

A linked list is a sequence of nodes that contain two fields - **data** and **a link (**a **pointer to the next node)** to the next node. The last node is linked to a terminator used to indicate the end of the list more often denoted by NULL.

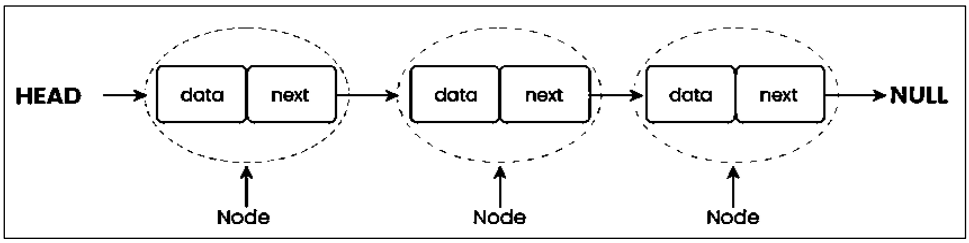


Fig. 5.1 : Structure of Linked list

Example 1 : Linked list containing character data

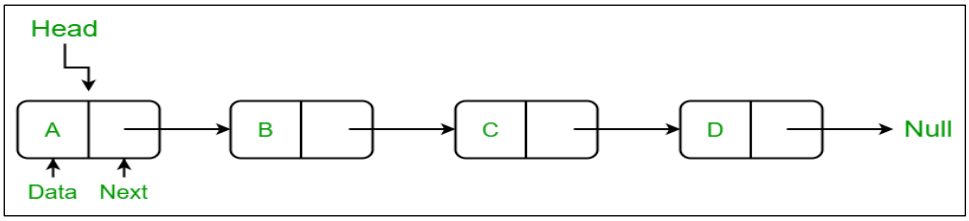


Fig. 5.2 : Linked list containing character data

Example 2 : Linked list containing integer data

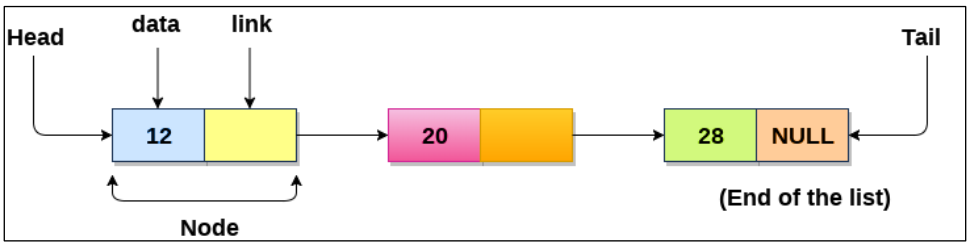


Fig. 5.3 : Linked list containing integer data

Some important points to be considered in linked list :

- The entry point of a linked list is known as the head which is link element.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.
- The consecutive elements are connected by pointers.
- The size of a linked list is not fixed.
- The last node of the linked list points to null.

- Memory is not wasted but extra memory is consumed as it also uses pointers to keep track of the next successive node.
- The entry point of a linked list is known as the head.

Basically each node contains two fields as data and a reference/link to the next node in sequential manner. This structure allows insertion or removal of elements efficiently from any position in the sequence. As in linked lists the nodes are serially linked the data access time is linear in respect to the number of nodes present in the list. To access any node requires that the prior node be accessed beforehand.

Advantages of a linked list :

- **Dynamic Data structure** : The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- **Ease of Insertion/Deletion** : The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- **Efficient Memory Utilization** : As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- **Implementation** : Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Disadvantages of Linked Lists :

- **Memory usage** : Linked lists require additional memory for storing the pointers, compared to arrays. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- **Random Access** : Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
- **Traversal** : Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing** : Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked list :

The applications of the Linked list are given as follows -

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.

- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.
- A linked list can be used in undo functionality of software's

Major differences between array and linked-list are listed below :

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

5.2 Representation of Linked List :

Each node of the linked list is represented as follows.

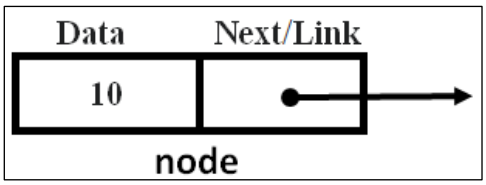


Fig. 5.4 : Node Representation

Each node consists :

- A data item
- An address of another node

Both the data item and the next node reference are wrapped in a structure as:

```
struct node
{
 int data;
 struct node *next;
};
```

Each struct node has a data item and a pointer to another struct node.

Representation of linked list in memory :

1. Linked list can be represented in memory by using two array respectively known as INFO/DATA and LINK. INFO[K] contains information of element and LINK[K] contains next node address respectively.
2. The list also requires a variable "Name" or "Start" or "Head", which contains address of the first node.
3. Pointer field of last node denoted by NULL which indicates the end of list.
   The linked list can be represented in memory as shown in following figure.

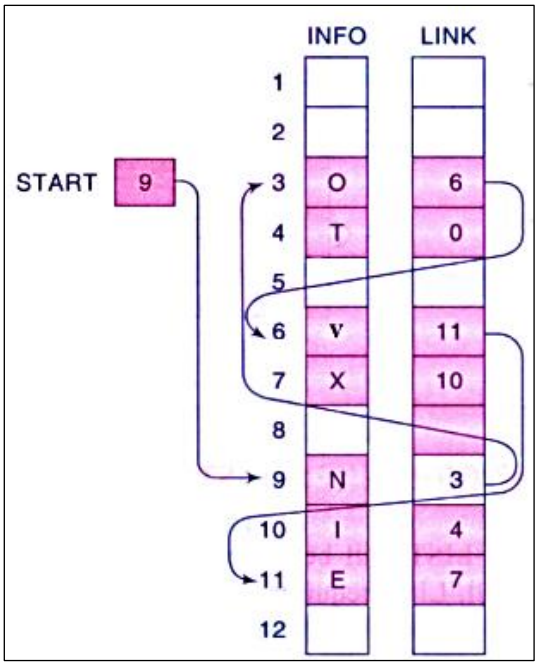


Fig. 5.5 : Representation of linked list in memory

The Figure shows a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters as follows:

START=9, so INFO [9] =N is the first character.

LINK [9] =3, so INFO [3] =O is the second character.

LINK [3] =6, so INFO [6] =\n (blank) is the third character.

LINK [6] =11 so INFO [11] =E is the fourth character.

LINK [11] =7, so INFO [7] =X is the fifth character.

LINK [7] =10, so INFO [10] =I is the sixth character.

LINK [10] -4, so INFO [4] = T is the seventh character.

LINK [4] =0, the NULL value, so the list is ended.

In other words NO EXIT is the character string.

5.3 Operations of Linked Lists :

The basic operations in a linked list are :

1. **Creation :** Creating a singly linked list process starts with creating a new node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the dynamic memory allocation malloc() function.
2. **Insertion :** Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list
3. **Deletion :** Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.
4. **Searching :** Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

5.4 Types of linked lists :

There are various types of linked list :

1. **Singly Linked Lists** :

   The nodes only point to the address of the next node in the list :

   Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.

Fig. 5.6 : Singly linked lists

2. **Doubly Linked Lists** :

The nodes point to the addresses of both previous and next nodes :

Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.

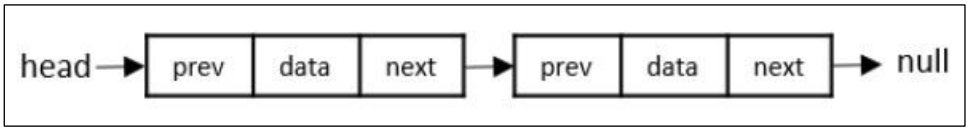


Fig. 5.7 : Doubly linked lists

3. **Circular Linked Lists** :

The last node in the list will point to the first node in the list. It can either be singly linked or doubly linked :

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.

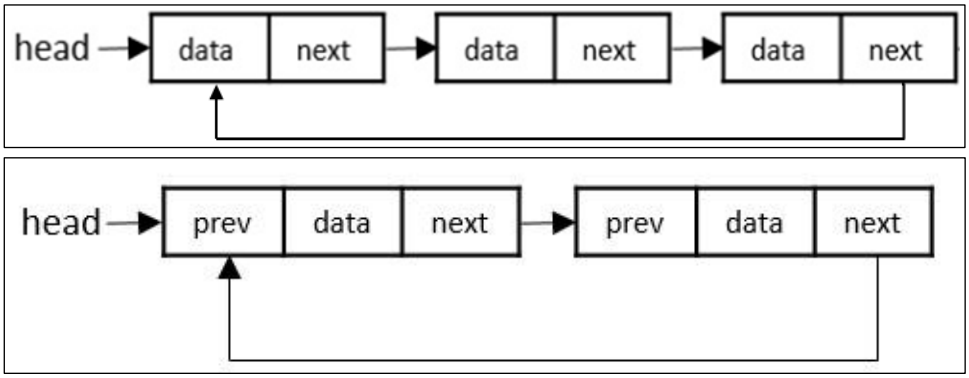


Fig. 5.8 : Circular linked lists

4. **Circular Doubly Linked List** :

The last node in the list will point to the first node in the list and the nodes point to the addresses of both previous and next nodes :

Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the

address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

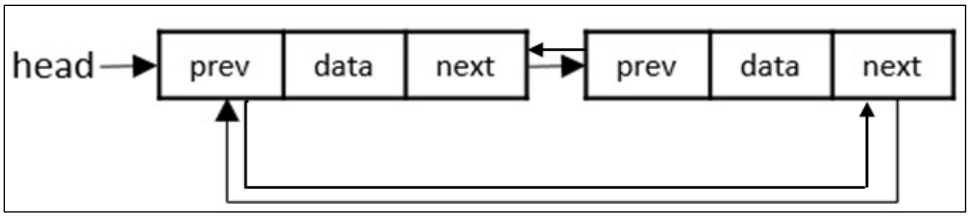


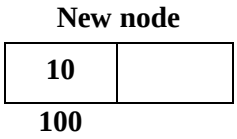
Fig. 5.9 : Circular Doubly linked lists

5.5 Singly Linked Lists :

a. Creating a node for Single Linked List :

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 5 illustrates the creation of a node for single linked list.

```
node* get node( )
{
        node* new node;
        new node =  ( node  * )  malloc( sizeof( node);
        printf( "\ n Enter data:  ");
        scanf( " % d",  & new node  - >  data);
        new node  - >  next =  NUL;
        return new nod e;
}
```



Creating a Singly

Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes :

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty, assign new node as start.
   start = newnode;
3. If the list is not empty, follow the steps given below:
   - The next field of the new node is made to point the first node (1.e. start node) in the list by assigning the address of the first node.
   - The start pointer is made to point new node by assigning address of new node times.

4. Repeat the above steps 'n' times.

Following Figure 5.10 shows 4 elements (data items) in a single linked list stored at different locations in memory.

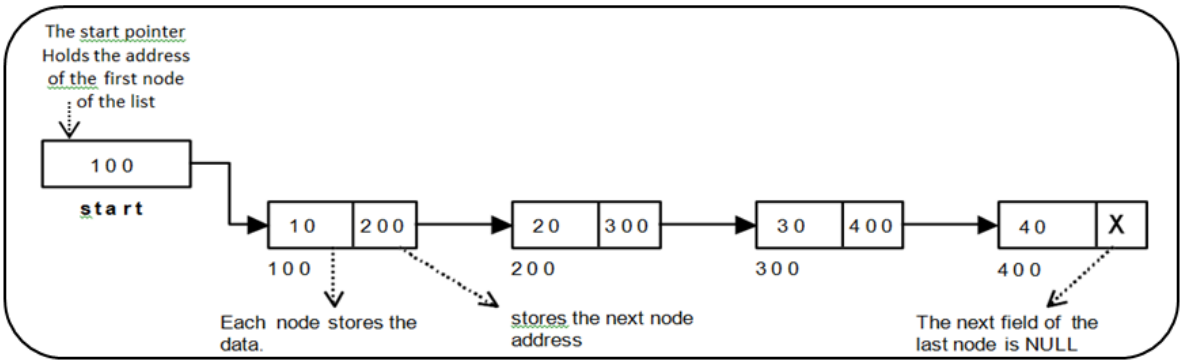


Fig. 5.10 : Single Linked List with four nodes

We can observe in the above Figure 5.10 that there are four different nodes having address 100, 200, 300 and 400 respectively. The first node contains the address of the next node, which is 200, the second node contains the address of the next node which is 300, the third node contains the address of the last node, which is 400, and the fourth node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a head pointer named as start and it contains address of first node which is 100.

b. Insertion of a node in Single Linked List :

The insertion of a node in a singly linked list is one of the most basic operations. Memory is to be allocated for the new node as allocated for creating a list before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL.

The new node can then be inserted at three different places as :

i) Inserting a node at the beginning.

ii) Inserting a node at the end.

iii) Inserting a node at intermediate position.

The process of inserting the node at particular place is discussed below.

i) Inserting a node at the beginning of the list :

Approach : To insert a node at the start/beginning/front of a Linked List, we need to :

1. Make the first node of Linked List linked to the new node
2. Remove the head from the original first node of Linked List
3. Make the new node as the Head of the Linked List.

The following are the **steps** to insert a new node at the beginning of the list.

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty, assign new node as start.
   start = newnode;
3. If the list is not empty, follow the steps given below
   - newnode->next = start;
   - start=newnode;

The Figure 5.11 shows inserting a node into the single linked list at the beginning.

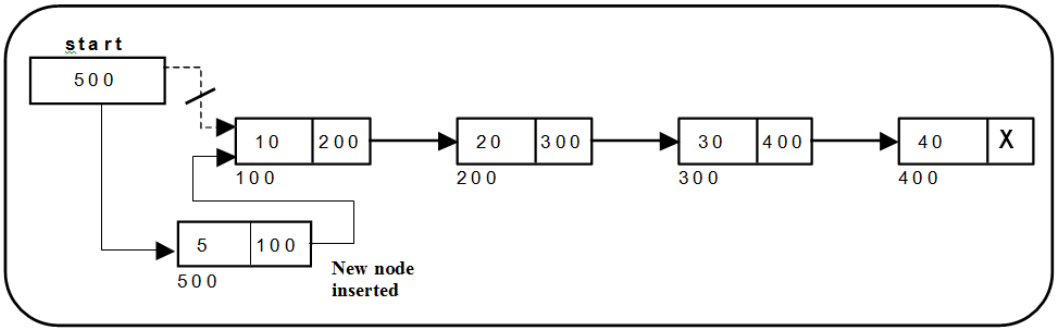


Fig. 5.11 : inserting a node at the beginning of the list.

ii) Inserting a node at the end of the list :

Approach : To insert a node at the end of a Linked List, we need to :

1. Go to the last node of the Linked List
2. Change the next pointer of last node from NULL to the new node
3. Make the next pointer of new node as NULL to show the end of Linked List

The **steps** to insert a new node at the end of the list are given below.

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty, assign new node as start.
   start = newnode;
3. If the list is not empty follow the steps given below:
   temp = start;
   while(temp->next != NULL)
   temp temp->next;
   temp->next = newnode;

Following Figure 5.12 shows how a node is inserted at the end into the single linked.

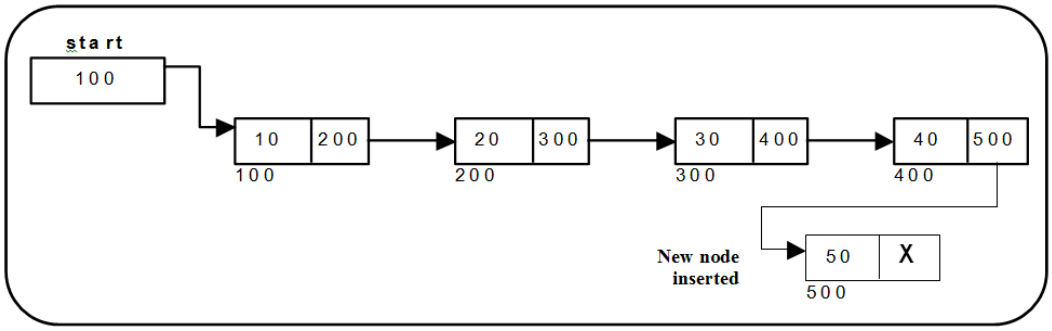


Fig. 5.12 : Inserting a node at the end of the list

iii) Inserting a node at intermediate position of list :

Approach : To insert a node after a given node in a Linked List, we need to :

Check if the given node exists or not.

The following are the steps to insert a new node in an intermediate position in the list :

1. Get the new node using getnode().
   newnode = getnode();
2. Ensure that the specified position is in between first node and last node.
   If not, specified position is invalid.
   This can be done by other countnode() function.
3. Store the starting address (which is in start pointer) in temp and prev pointers.
   Then traverse the temp pointer upto specified position followed by prev pointer.
4. After reaching the specified position, follow the steps given below:
   prev -> next = newnode;
   newnode -> next = temp;

Figure 5.13 shows inserting a node at a specified intermediate position other than beginning and end into the single linked list. Let the intermediate position be 3.

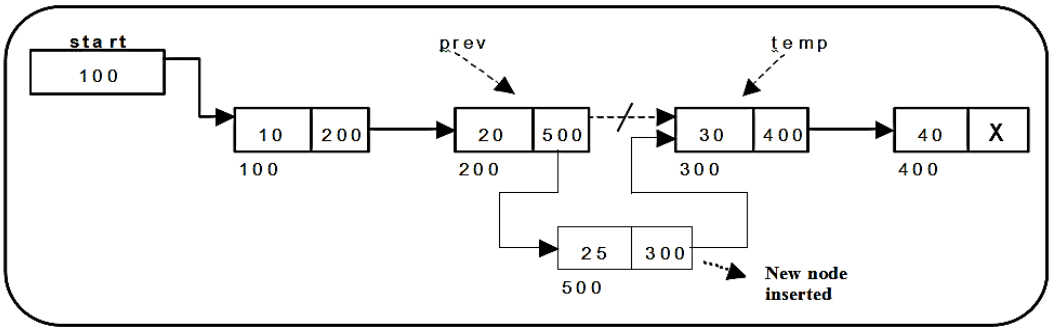


Fig. 5.13 : inserting a node at a specified intermediate position into single linked list.

c) Deletion of a node from Single Linked List :

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

i) Delete node from Beginning of the list

ii) Delete node from End of the list

iii) Delete a Specific Node (intermediate) of the list

Memory is always release out after each delete operation on linked list at any position.

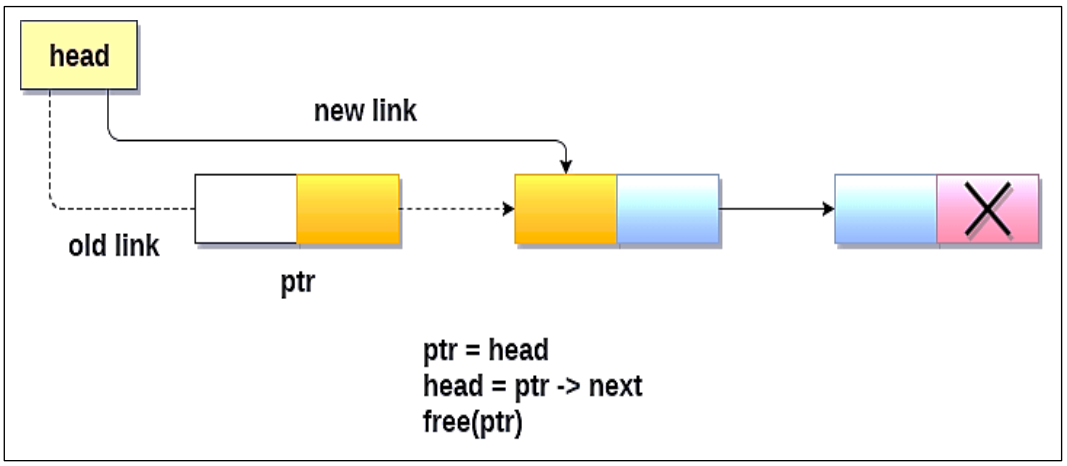


Fig. 5.14 : Deletion of a node from Single Linked List.

i) Delete node from Beginning of the list :

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the start/head, point to the next of the start/head. This will be done by using the following statements.

The following steps are followed, to delete a node from the beginning of the list :

1. Check If list is empty then
   Display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
   temp = start;
   start start->next;
3. free(temp);

The following Figure 5.15 shows deleting a node at the beginning of a single linked list.

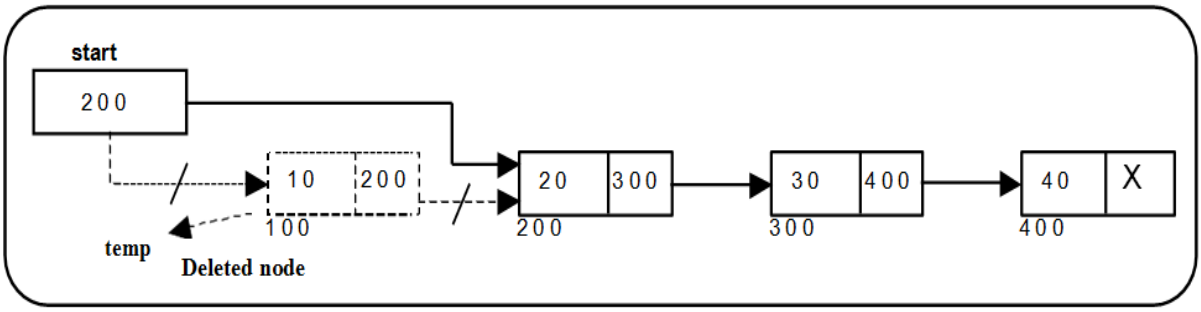


Fig. 5.15 : deleting a node at the beginning of a single linked list

Memory is then release out by free operation with the pointer temp which was pointing to the start/head node of the list. This will be done by using the following statement.

free(temp);

ii) Delete node from End of the list :

It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.

There are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario, the condition start -> next = NULL will survive and therefore, the only node start/head of the list will be assigned to null.

This will be done by using the following statements.

1. temp = start
2. start = NULL
3. free(temp)

In the second scenario, the condition head -> next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers **temp** and **prev** will be used where **temp** will point to the last node and **prev** will point to the second last node of the list.

The following **steps** are followed to delete a node at the end of the list :

1. If list is empty then
   Display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
   temp= prev= start;

```
while(temp-> next != NULL)
{
prev = temp;
temp= temp-> next;
}
Prev->next=NULL,
free(temp);
```

The following Figure 5.16 shows deleting a node at the end of the list.

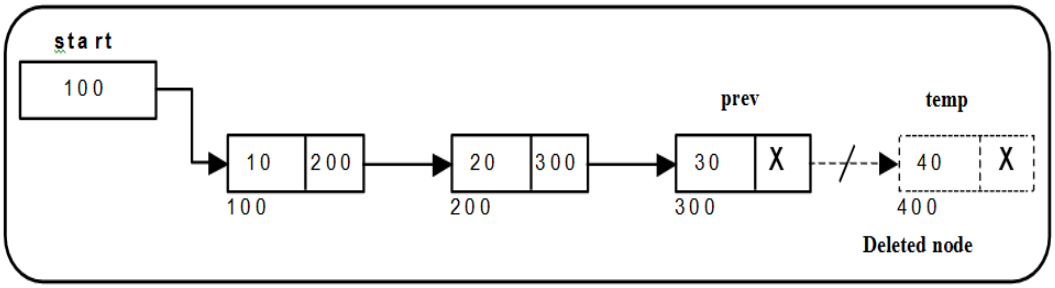


Fig. 5.16 : deleting a node at the end of the list.

iii) Delete a Specific Node (intermediate) of the list :

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used and then just need to make a few pointer adjustments.

The **step**s to delete a node from an intermediate position from the list having more than two nodes :

1. If list is empty then
   Display 'Empty List' message.
2. If the list is not empty, follow the steps given below.

```
if(pos> 1 && pos<nodectr)
{
    temp= prev= start,
    ctr=1;
    while(ctr<pos)
    {
        prev=temp;
```

```
            temp=temp->next;
            ctr++;
        }
        prev-> next= temp -> next;
        free(temp);
        printf("\n node deleted..");
    }
```

Figure 5.17 shows deleting a Specific node (intermediate) from a single linked list.

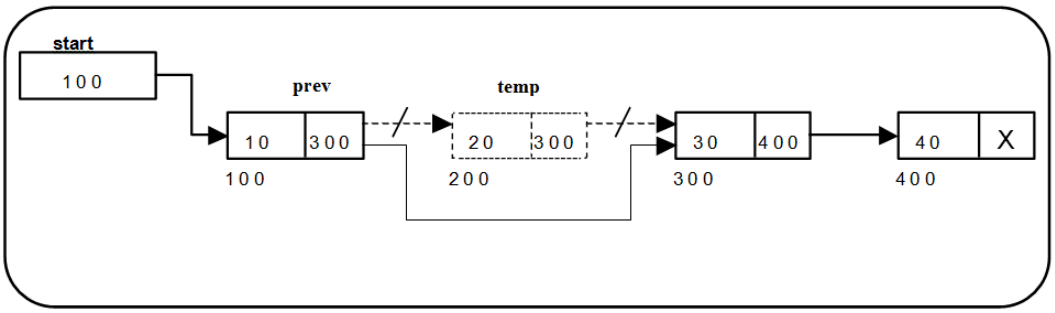


Fig. 5.17 : deleting a Specific node (intermediate) of the list.

d) Traversing and Displaying Single Linked List :

The data or information contained in a list can be display by traversing a list. Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing is the process of visiting each node of the list once in order to perform some operation on that. Traverse a linked list is done node by node from the first node until the end of the list is reached.

Approach :

1. Assign the address of start pointer to a temp pointer.
2. Display the information from the data field of each node.

Traversing a list involves the following **steps.**

1. temp = start;
2. IF temp = NULL then
   Display 'Empty List';
3. If list not empty then do following steps
   While (temp != NULL)
   {
   Display temp->DATA;
   Temp=temp -> NEXT;
   }

e) Searching Single Linked List :

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and makes the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Steps for Searching Single Linked List are given below :

1. Assign temp= start;
2. i = 0;
3. IF temp = NULL then
   Display 'Empty List';
4. If list not empty then do following steps
   while (temp!=NULL)
   {
   If temp->data=ITEM then
   Display i+1;
   i=i+1;
   temp=temp->next;
   }

5.6 Doubly Linked List:

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

A doubly linked list is a two-way list in which all nodes will have two links. It is a complex type of linked list in which a node contains a pointer to the previous/left as well as the next/right node in the sequence.

In a doubly linked list, a node consists of three parts :

- Data
- Pointer to the next/right node in sequence (next/right pointer)
- Pointer to the previous/left node (previous/left pointer).

The structure definition of doubly Linked List:

```
struct  dlinklist
{
        struct dlinklist *left ;
```

```
        int data;
        struct  dlinklist *right ;
} ;
Typedef struct  dlinklist node;
node *start = NULL;
```

A sample node in a doubly linked list is shown in the Figure 5.18.

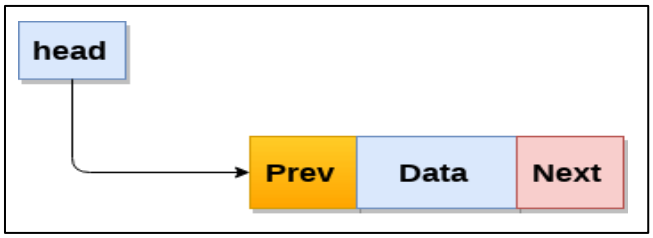


Fig. 5.18 : Node in a doubly linked list.

Example : A doubly linked list containing three nodes having numbers from 1 to 3 in their data part is shown in the following Figure 5.19.

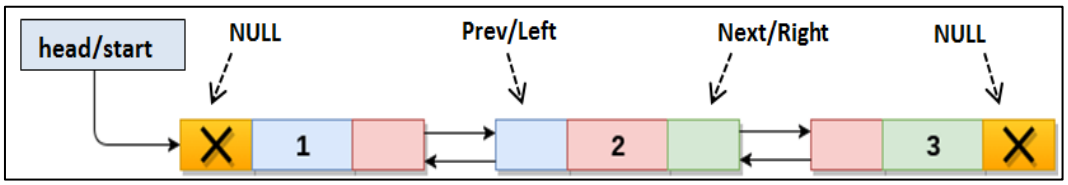


Fig. 5.19 : Example of a doubly linked list

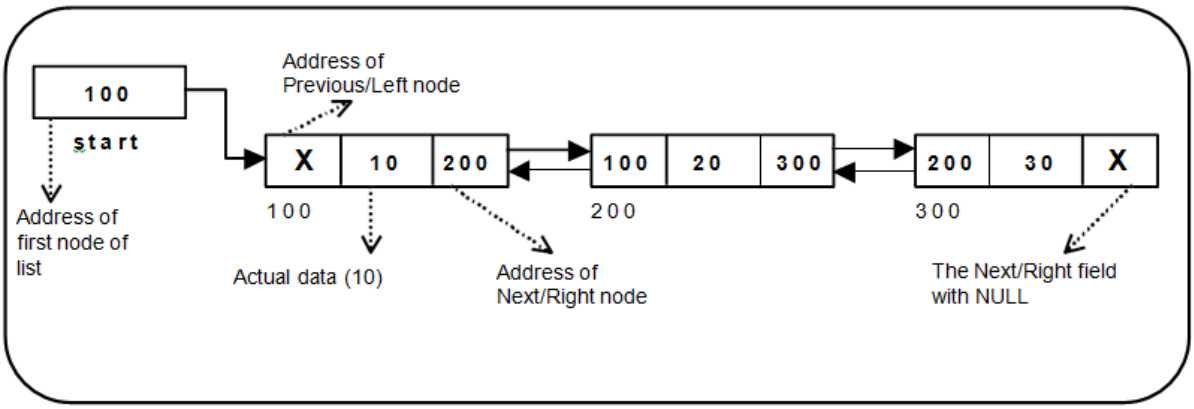


Fig. 5.20 : Example of a doubly linked list

The basic operations in a doubly linked list are :

1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.

a) Creating a node in doubly Linked List :

For creating a double linked list sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode() is used for creating a node. After allocating memory for the structure of type node the information for the item/data has to be read from the user and set both left field and right field to NULL.

The code to create a new node in double linked list :

```
node* getnode( )
{
        nod e* newnode;
        newnode = ( nod e * ) malloc( sizeo f ( node) ) ;
        printf ( " \n Enter data: " ) ;
        sca nf( " %d " , & newnode -> data) ;
        newnode - > left = NULL;
        newnode  - > right =   NULL;
         return newnode;
}
```

After creating a new node with data value 10 the node can be visualize as shown in Figure 5.21.

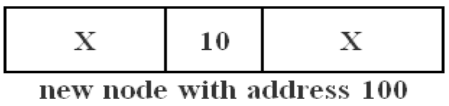


Fig. 5.21 : creation of new node in doubly link list

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list. There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element.

a) Creating a Double Linked List with 'n' number of nodes:

The following **steps** are to be followed to create 'n' number of nodes:

1. Get the new node using getnode().

newnode =getnode();

2. If the list is empty then
   start = newnode.
3. If the list is not empty, follow the steps given below :
   - The left field of the new node is made to point the previous node.
   - The previous nodes right field must be assigned with address of the new node.
4. Repeat the above steps 'n' times.

Figure 5.22 shows 3 items in a double linked list stored at different locations using createlist() function.

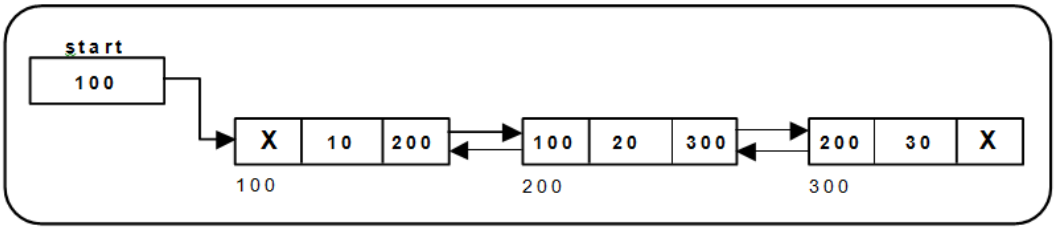


Fig. 5.22 : creation of new node with 3 nodes in doubly link list.

To create 'n' number of nodes in Double Linked List the function createlist () is given below:

```
void createlist( int n)
{
  int i;
  node *newnode;
  node *temp;
  for(i=0; i<n; i++)
  {
    newnode = getnode();
    if(start = = NULL)
    {
            start = newnode;
    }
    else
    {
        temp = start;
        while(temp -> right)
        temp=temp-> right;
        temp-> right= newnode;
```

```
            newnode-> left = temp;
        }
    }
}
```

b) Inserting a node in doubly linked list :

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

Node can be inserted at various position in doubly linked list as :

i) Inserting a node at the beginning in doubly linked list

ii) Inserting a node at the end in doubly linked list

iii) Inserting a node at the intermediate position in doubly linked list

i) Inserting a node at the beginning in doubly linked list :

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

The following **steps** are to be followed to insert a new node at the beginning of the list :

1. Get the new node using getnode().
   ```
   newnode=getnode();
   ```
2. If the list is empty then
   ```
   start = newnode.
   ```
3. If the list is not empty, follow the steps given below:
   ```
   newnode-> right= start;
   start->left= newnode;
   start = newnode;
   ```

Figure 5.23 shows inserting a node into the double linked list at the beginning.

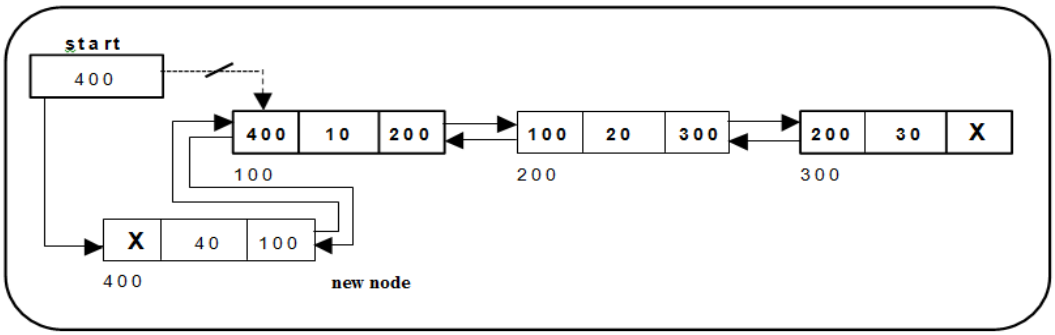


Fig. 5.23 : inserting a node at the beginning in doubly linked list.

ii) Inserting a node at the end in doubly linked list :

Figure 5.24 shows inserting a node into the double linked list at the end.

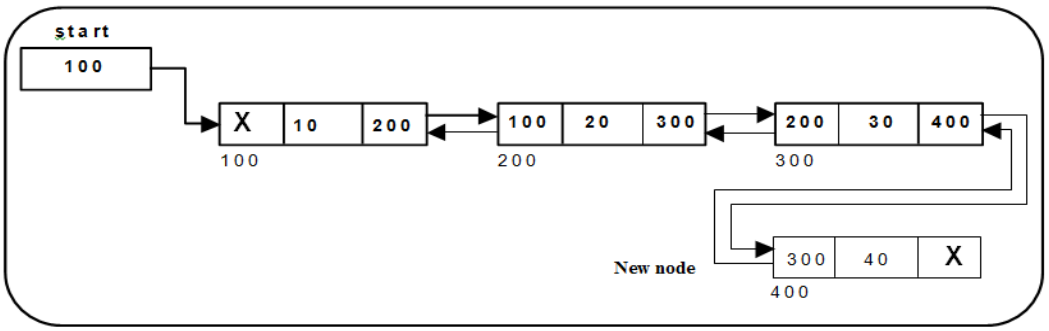


Fig. 5.24 : Inserting a node at the end in doubly linked list

The **steps** to insert a new node at the end of the list :

1. Get the new node using
   getnode()
2. If the list is empty then
   start = newnode.
3. If the list is not empty follow the steps given below:

```
temp = start;
while (temp -> right!= NULL)
temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;
```

iii) Inserting a node at the intermediate position in doubly linked list :

Figure 5.25 shows inserting a node into the doubly linked list at intermediate position.

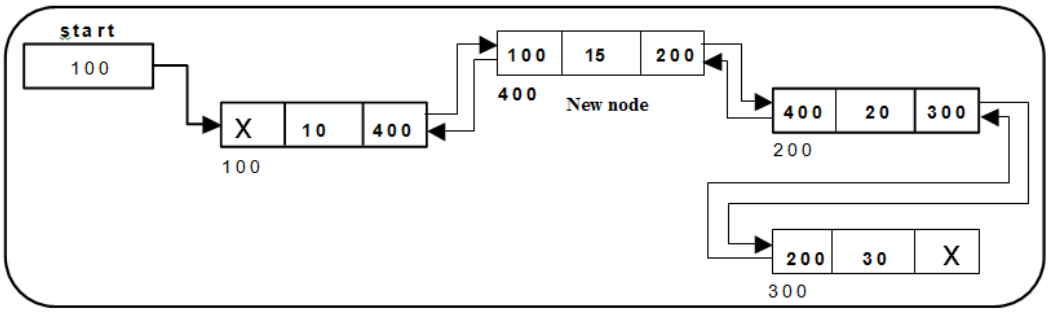


Fig. 5.25 : Inserting a node at intermediate position in doubly linked list.

The **steps** to insert a new node in an intermediate position in the doubly linked list :

1. Get the new node using getnode().
   newnode=getnode();
2. Ensure that the specified position is in between first node and last node.
   If not, specified position is invalid. (done by countnode() function)
3. Store the starting address (which is in start pointer) in temp and prev pointers.
4. Then traverse the temp pointer upto the specified position followed by prev pointer.
5. After reaching the specified position, follow the steps given below:
   newnode->left = temp;
   newnode-> right = temp -> right;
   temp -> right -> left = newnode;
   temp -> right = newnode;

c) Deleting a node from doubly linked list

Node can be deleted from various position in doubly linked list as:

i) Deleting a node from the beginning in doubly linked list
ii) Deleting a node from the end in doubly linked list
iii) Deleting a node from the intermediate position in doubly linked list

i) Deleting a node from the beginning in doubly linked list :

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the start/head pointer to pointer ptr and shift the head/start pointer to its next.

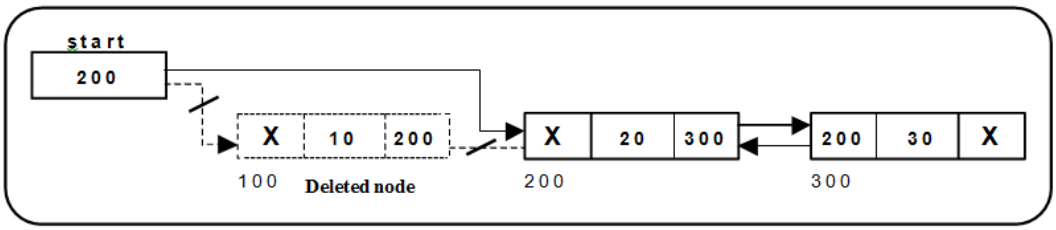


Fig. 5.26 : Deleting a node at beginning in doubly linked list.

ii) Deleting a node from the end in doubly linked list :

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, follow the given steps as below :

- If the list is already empty then the condition start == NULL will become true and therefore the operation can not be carried on.

- If there is only one node in the list then the condition start → next == NULL become true. In this case, we just need to assign the start of the list to NULL and free start in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list.

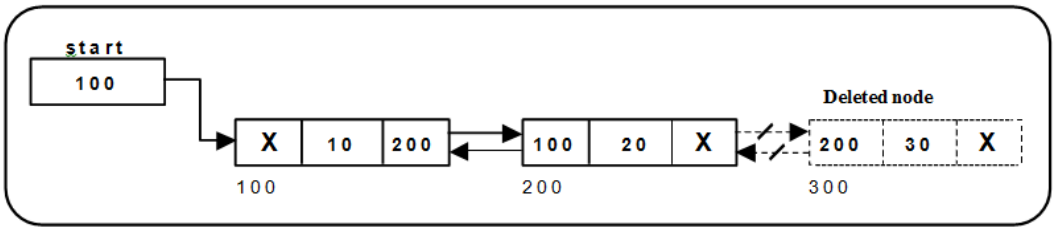


Fig. 5.27 : Deleting a node at end in doubly linked list

The following steps are followed to delete a node at the end of the list :

1. If list is empty then display 'Empty List' message

2. If the list is not empty, follow the steps given below:

```
temp start;
while(temp-> right != NULL)
{
temp = temp-> right;
}
temp -> left -> right = NULL;
```

3. free(temp);

iii) Deleting a node at an Intermediate position in doubly linked list :

In order to delete the node after the specified data, we need to perform the following **steps**.

- Copy the head pointer into a temporary pointer temp.
- Traverse the list until we find the desired data value.
- Check if this is the last node of the list. If it is so then we can't perform deletion.
- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.
- Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

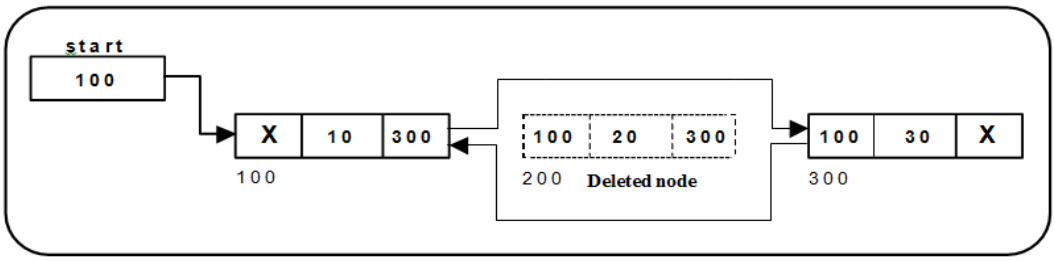


Fig. 5.28 : Deleting a node at Intermediate position in doubly linked list.

The following **steps** are followed, to delete a node from an intermediate position in the list (List must have more than two nodes).

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
   - Get the position of the node to delete.
   - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
   - Then perform the following steps:

```
if(pos> 1 && pos<nodectr)
{ temp=start;
  i=1:
  while(i<pos)
  { temp=temp-> right;
    i++;
  }
  temp-> right-> left = temp-> left;
  temp-> left-> right = temp -> right;
  free(temp);
  printf("\n node deleted..");
  }
```

d) Traversal and displaying a doubly linked list (Left to Right) :

Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc. To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached.

The following steps are followed, to traverse a list from left to right :

1. If list is empty then display 'Empty List'

2. If the list is not empty, follow the steps given below :

```
temp = start;
while(temp != NULL)
{
Display temp-> data;
temp = temp-> right;
}
```

5.7 Circular Linked List:

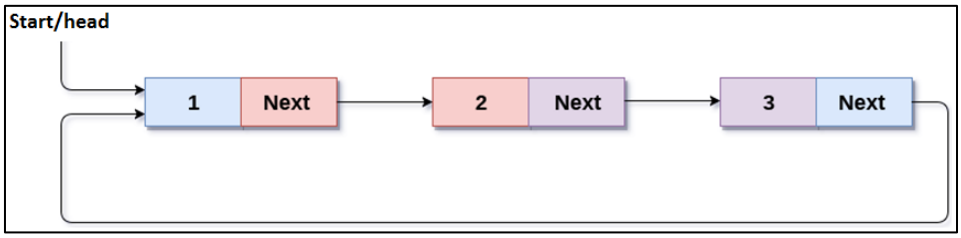
In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list. That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following figure 5.29 shows a circular singly linked list.

Example : 1



Example : 2

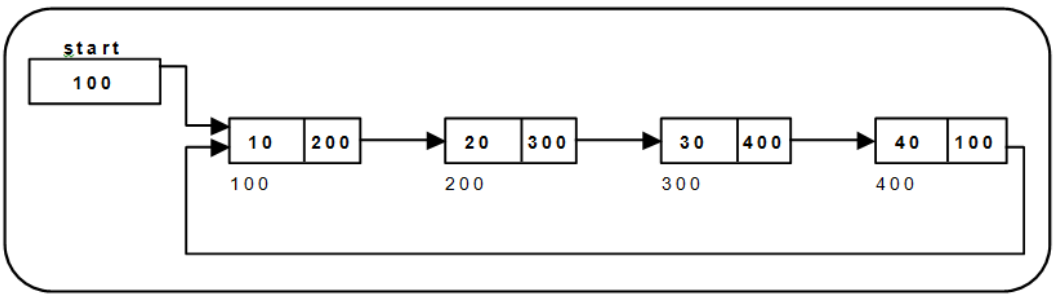


Fig. 5.29 : Circular singly linked list

The basic operations in a circular single linked list are :

1. Creation
2. Insertion
3. Deletion
4. Traversing

a) Creating a circular single Linked List with 'n' number of nodes

The **steps** to create n number of nodes :

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty, assign new node as start.
   start = newnode;
3. If the list is not empty then
   temp = start;
   while(temp->next != NULL)
   temp = temp-> next;
   temp->next = newnode;
4. Repeat the above steps n times.
5. newnode->next = start;

b) Inserting a node in circular linear linked list :

A node can be inserted at various positions (start, end, intermediate) in the circular linear list as :

i) Inserting a node at the beginning in circular linear linked list

ii) Inserting a node at the end in circular linear linked list

i) Inserting a node at the beginning in circular linear linked list :

There are two scenarios in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Figure 5.30 shows inserting a node into the circular single linked list at the beginning.

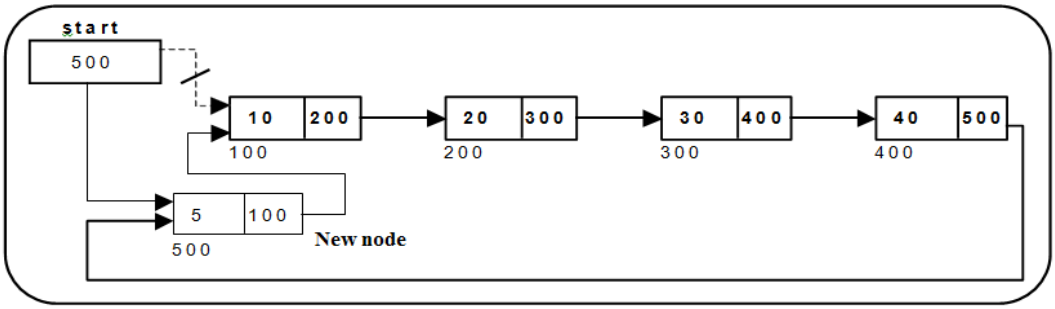


Fig. 5.30 : inserting a node at the beginning in circular single linked list

The following **steps** are to be followed to insert a new node at the beginning of the circular list:

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty then assign new node as start.
   Start= newnode;
   newnode->next = start;
3. If the list is not empty then
   last = start;
   while (last-> next != start)
   last=last->next;
   newnode->next = start;
   start = newnode;
   last-> next= start;

ii) Inserting a node at the end in circular linear linked list :

There are two scenarios in which a node can be inserted in circular singly linked list at end. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Fig. 5.31 shows inserting a node into the circular single linked list at the end.

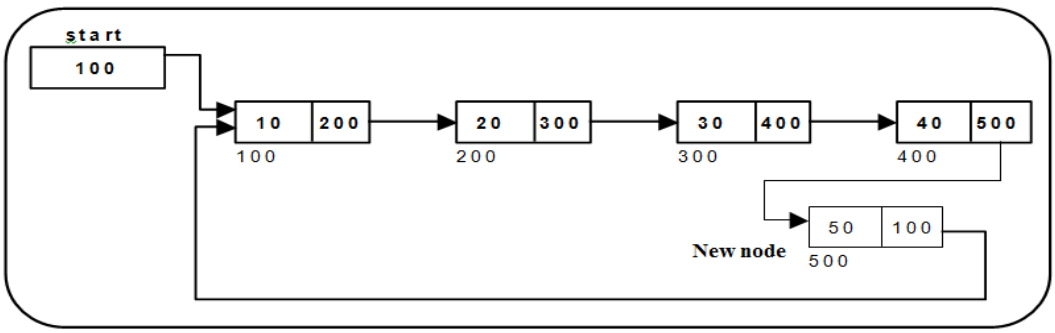


Fig. 5.31 : inserting a node at the end in circular single linked list

The following steps are to be followed to insert a new node at the end of the circular list :

1. Get the new node using getnode().
   newnode = getnode();
2. If the list is empty, assign new node as start.
   start = newnode;
   newnode->next = start;

3. If the list is not empty then
   temp = start;
   while(temp-> next != start)
   temp= temp-> next;
   temp->next = newnode;
   newnode->next = start;

c) Deleting a node from circular linear linked list

A node can be deleted from various positions (start, end, intermediate) in the circular linear list as :

i) Deleting a node at the beginning in circular linear linked list
ii) Deleting a node at the end in circular linear linked list

i) Deleting a node at the beginning in circular linear linked list :

Removing the node from circular singly linked list at the beginning.

Figure 5.32 shows deleting a node from the circular single linked list at the beginning.

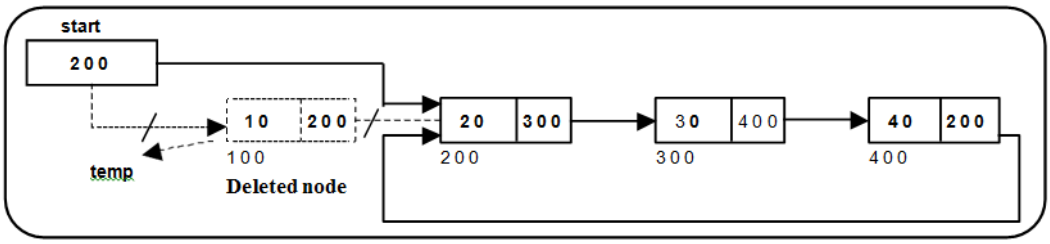


Fig. 5.32 : deleting a node at the beginning in circular single linked list

The **steps** to delete a node at the beginning of the circular single linked list:

1. If the list is empty
   Display 'Empty List"
2. If the list is not empty then
   Last= temp= start;
   while(last-> next != start)
   last= last-> next;
   start= start->next;
   last-> next= start,
3. After deleting the node, if the list is empty then
   start = NULL

ii) Deleting a node at the end in circular linear linked list :

There are three scenarios of deleting a node in circular singly linked list at the end: the list is empty, the list contains single element and the list contains more than one element.

The following **steps** are followed to delete a node at the end of the circular linear linked list :

1. If the list is empty
   Display a message 'Empty List"
2. If the list is not empty then
   temp= start;
   prev= start;
   while(temp-> next != start)
   {prev=temp;
   temp = temp-> next;
   }prev-> next= start;
3. After deleting the node, if the list is empty then
   start = NULL.

Figure 5.33 shows deleting a node at the end of a circular single linked list.

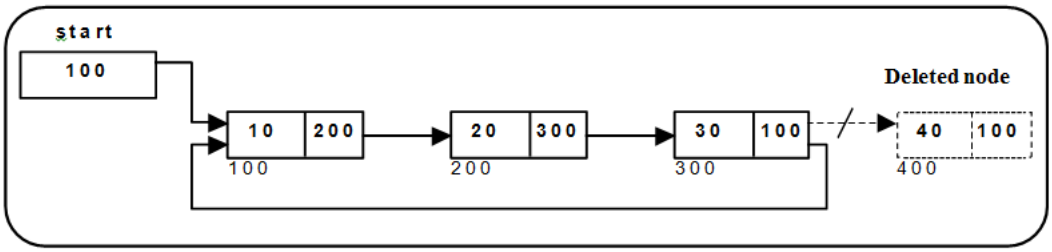


Fig. 5.33 : deleting a node at the end in circular single linked list

d) Traversing circular linear linked list from Left to Right :

Traversing circular linear linked list means visiting each element of the list at least once in order to perform some specific operation. Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable **temp** to head pointer and run the while loop until the next pointer of temp becomes **start/head**.

Steps to traverse a circular single linked list from left to right :

1. If list is empty then
   Display 'Empty List' message
2. If the list is not empty then
   temp = start;
   do

```
{
Display ("%d", temp- > data);
temp = temp->next;
} while(temp!= start);
```

5.8 Circular Doubly Linked List:

Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contains address of the last node in its previous pointer.

A circular doubly linked list is shown in the following Figure 5.34.

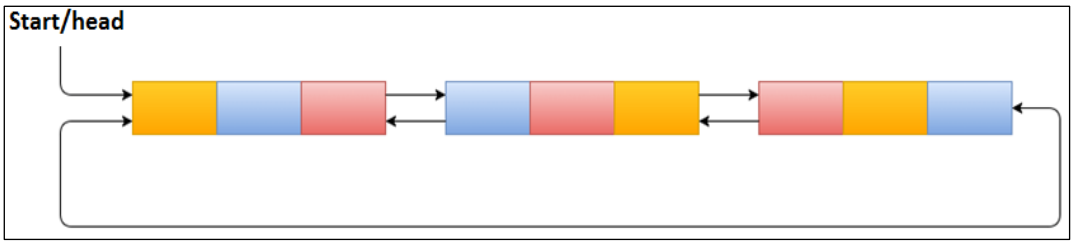


Fig. 5.34 : Circular doubly linked list.

Operations on circular doubly linked list :

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list.

The basic operations in a circular doubly linked list are :

1. Insertion
2. Deletion

a) Inserting a node in circular doubly Linked List :

A node in circular doubly Linked List can be inserted in various ways as :

i) Inserting a node at beginning in circular doubly linked list

ii) Inserting a node at end in circular doubly linked list

i) Inserting a node at beginning in circular doubly linked list :

The following are **steps** to insert a new node at the beginning of the circular doubly linked list:

1. Get the new node using getnode().
   ptr = getnode();
2. If the list is empty (head == NULL)then
   head = ptr;

ptr -> next = head;
ptr -> prev = head;

3. If the list is not empty (head != NULL)then
   temp = head;
   while(temp -> next != head)
   {
      temp = temp -> next;
   }
4. All the pointer adjustments done
   temp -> next = ptr;
      ptr -> prev = temp;
      head -> prev = ptr;
      ptr -> next = head;
      head = ptr;

Figure 5.34 shows how the new node is inserted at beginning in circular doubly linked list

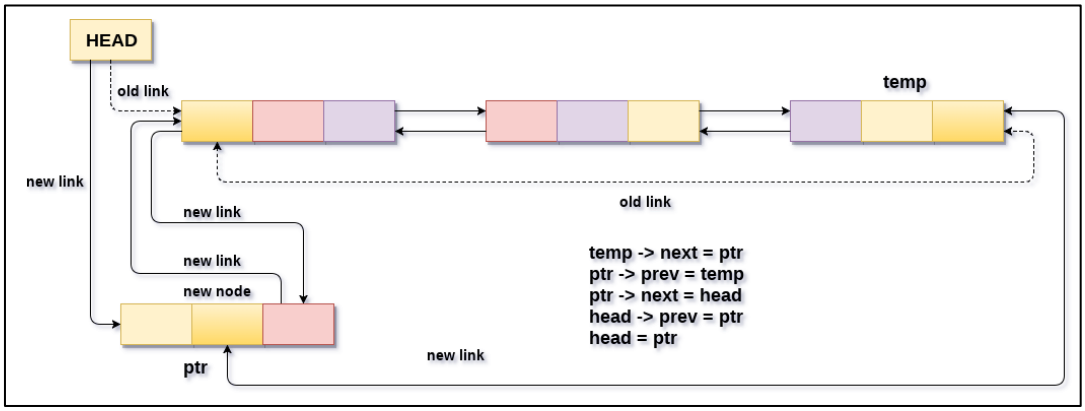


Fig. 5.34 : A node is inserted at beginning in circular doubly linked list

ii) Inserting a node at end in circular doubly linked list :

The following are **steps** to insert a new node at the end of the circular doubly linked list:

1. Get the new node using getnode().
   ptr = getnode();
2. If the list is empty (head == NULL)then
   head = ptr;
   ptr -> next = head;

ptr -> prev = head;

3. If the list is not empty (head != NULL) then
   head -> prev = ptr;
   ptr -> next = head;
4. All the pointer adjustments done
   temp->next = ptr;
   ptr ->prev=temp;

Figure 5.35 shows how the new node is inserted at end in circular doubly linked list

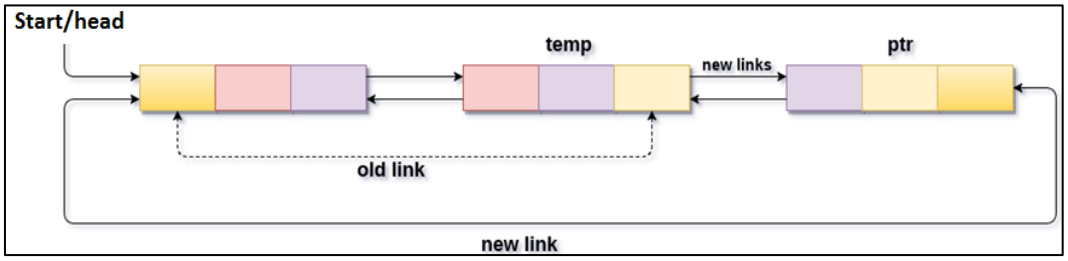


Fig. 5.35 : A node is inserted at end in circular doubly linked list.

b) Deleting a node in circular doubly Linked List

A node in circular doubly Linked List can be deleted in various ways as:

i) Deleting a node at beginning in circular doubly linked list

ii) Deleting a node at end in circular doubly linked list

i) Deleting a node at beginning in circular doubly linked list

There can be two scenario of deleting the beginning node in a circular doubly linked list. The node which is to be deleted can be the only node present in the linked list and the list contains more than one element in the list

The following are **steps** to delete node at the beginning of the circular doubly linked list:

1. If only one node (head → next == head)
   head = NULL;
   free(head);
2. Otherwise (head → next == head is false)
   temp = head;
   while(temp -> next != head)
   {
   temp = temp -> next;
   }

3. All the pointer adjustments done
   temp -> next = head -> next;
   head -> next -> prev = temp;
4. free the head and make new head node of the list
   free(head);
   head = temp -> next;

Figure 5.36 shows how the beginning node is deleted in circular doubly linked list

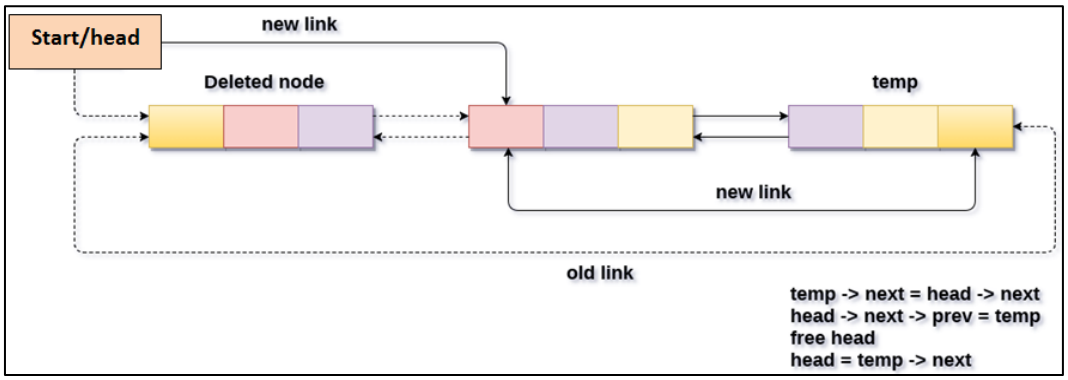


Fig. 5.36 : deleted beginning node in circular doubly linked list.

ii) Deleting a node at end in circular doubly linked list :

There can be two scenario of deleting the end node in a circular doubly linked list. The node which is to be deleted can be the only node present in the linked list and the list contains more than one element in the list.

The following are **steps** to delete node at the end of the circular doubly linked list :

1. If only one node (head → next == head)
   head = NULL;
   free(head);
2. Otherwise (head → next == head is false)
   temp = head;
   while(temp -> next != head)
   {
   temp = temp -> next;
   }
3. All the pointer adjustments done
   temp -> prev -> next = head;

head -> prev = ptr -> prev;

4. free the head
   free(head);

Figure 5.36 shows how the end node is deleted in circular doubly linked list.

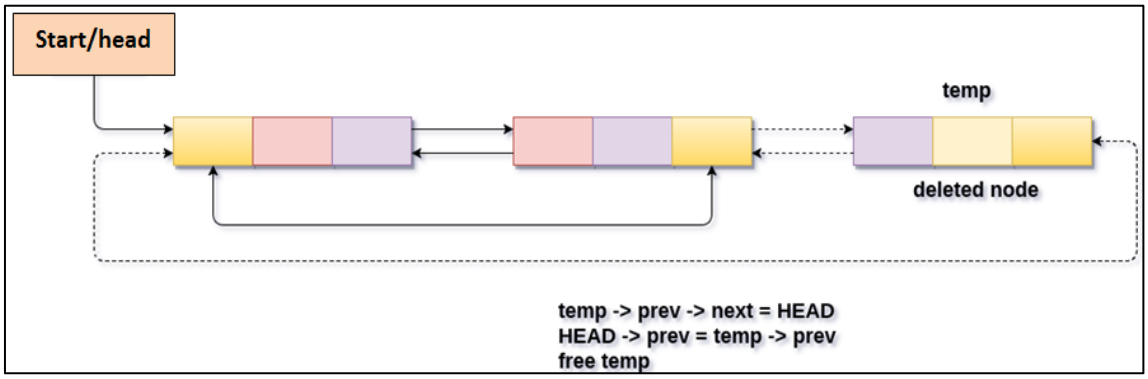


Fig. 5.36 : deleted end node in a circular doubly linked list.

: QUESTIONS :

Short and Long Answer Questions :

1. What is the limitation of sequential data structures?
2. What is linked list?
3. Give real world example of linked list.
4. Explain logical representation of linked list.
5. What are the advantages of singly linked list?
6. What are the disadvantages of singly linked list?
7. Which are the operations performed in singly linked list?
8. What is the need for linked representation of lists?
9. Define circular linked list.
10. What are the advantages of circular linked list?
11. What are the disadvantages of circular linked list?
12. What is the node structure for circular linked list?
13. Define doubly linked list.
14. What are the advantages of doubly linked list?
15. What are the disadvantages of doubly linked list?
16. List out operations performed in doubly linked list.

17. List application of linked list.
18. What is the difference between circular linked list and linear linked list?
19. What is the difference between array and stack? 2
20. What do you mean by polynomials?
21. Give node structure for the term of polynomial having single variable.
22. How singly linked list representation of polynomials?
23. Write short note on linked list.
24. Explain operation of singly linked list with algorithm.
25. Explain circular linked list.
26. What are the advantages of circular linked list over singly linked list?
27. Explain application of linked list.

***

6. Chapter

Trees and Graphs

Contents:

(10 L, 15 M)

6.1 Introduction to Tree :

A Tree is a finite set of data items or elements refer as nodes such that -

1. There is a special data item or element (node) called the root of the tree.
2. The remaining data items or elements (nodes) are partitioned into number of disjoint subsets each of which itself and are called Sub-trees of the root.

A tree is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes. Some data organizations require categorizing data into groups or sub-groups. A data structure is said to be non- linear if its elements form a hierarchical classification in which data items appear at various levels.

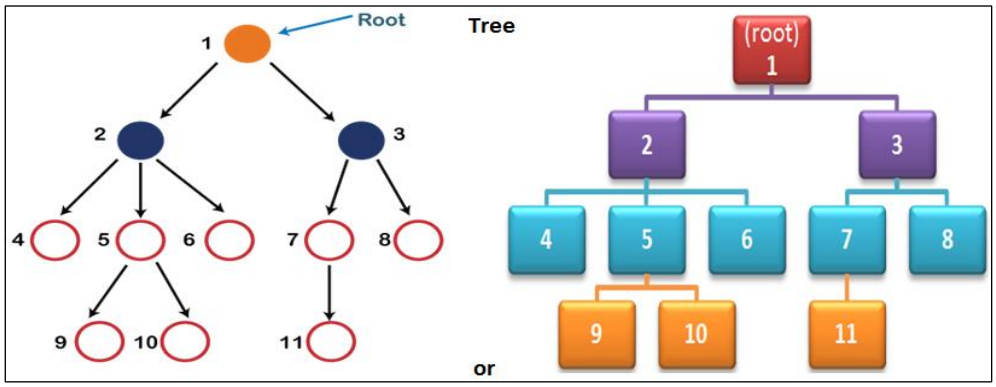


Fig. 6.1 : Tree

The hierarchical structure of tree is shown in Figure-1. Each node is labelled with some **data/value** (char/number). Each arrow shown in this figure is known as a ***link*** between the two nodes.

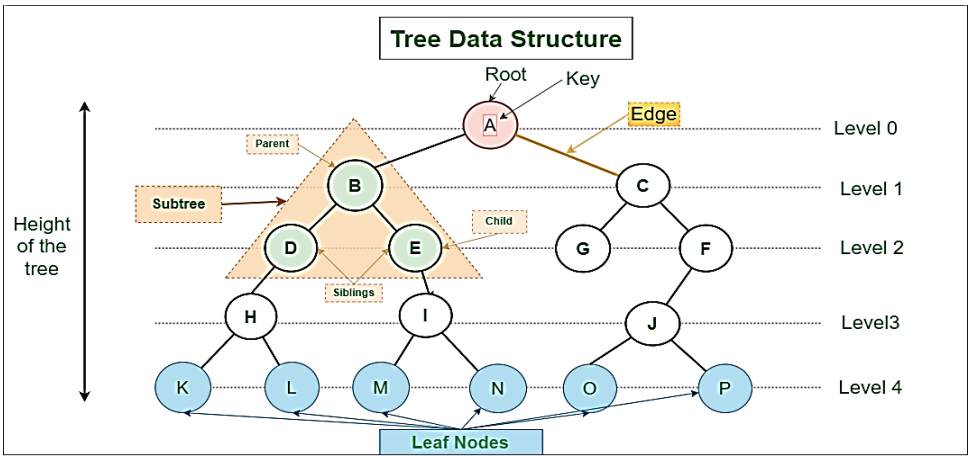


Fig. 6.2 : Tree Terminology

Basic Terminologies In Tree Data Structure :

The basic terminologies in tree data structure is discussed below and shown in Figure-2.

1. **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
2. **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
3. **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must

contain exactly one root node and exactly one path from the root to all other nodes of the tree.

4. **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{K, L, M, N, O, P, G}** are the leaf nodes of the tree.
5. **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
6. **Descendant:** Any successor node on the path from the leaf node to that node. **{E,I}** are the descendants of the node **{B}.**
7. **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
8. **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
9. **Internal node:** A node with at least one child is called Internal Node.
10. **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
11. **Subtree**: Any node of the tree along with its descendant.
12. **Height or Depth**: The height or depth of a tree is defined to be the maximum level of any node in the tree. This is equal to the longest path from root to any leaf node. The depth or height of tree in Figure-1 is 5.
13. **Forest:** A forest is set of n>=0 disjoint tree. If we remove root of a tree we get a forest. In the Figure-1, if we remove the root node A we get a forest with two trees.

Applications of trees :

The following are the applications of trees :

1. **Storing naturally hierarchical data :** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
2. **Organize data :** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
3. **Trie :** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
4. **Heap :** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
5. **B-Tree and B+ Tree :** B-Tree and B+ Tree are the tree data structures used to implement indexing in databases.
6. **Routing table :** The tree data structure is also used to store the data in routing tables in the routers.

6.2 Binary Tree :

A binary tree is a tree in which each node can have at most two children such that :

1. a binary tree can be empty called the NULL or
2. consist of root node and two distjoint binary trees termed as left subtree and right subtree.

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

A binary tree is either empty or consists of a node called the root having two binary subtrees called the left subtree and the right subtree.

A tree with no nodes is called as a null tree.

The structure of binary tree is shown in Figure-6.3.

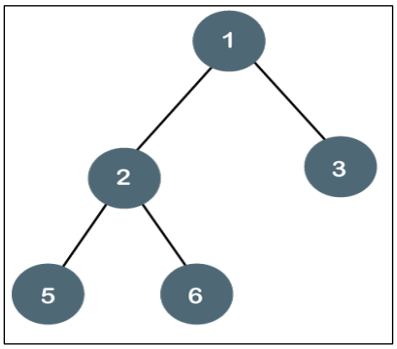


Fig.6.3 : Binary Tree

In programming, the structure of a node can be defined as :

```
struct node
{
 int data;
struct node *left;
 struct node *right;
}
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Properties of binary trees :

Some of the important properties of a binary tree are as follows:

- If h = height of a binary tree, then
  Maximum number of leaves = 2h

Maximum number of nodes = 2h + 1 - 1

- If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.
- Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2l node at level l.
- The total number of edges in a full binary tree with n node is n - 1.

Types of Binary tree :

There are various types of Binary tree on the basis of the number of children and the completion of levels.

1. Full/ Proper/ Strict Binary tree :

Full Binary tree is based on the number of children nodes have :

A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree or Strict Binary tree.

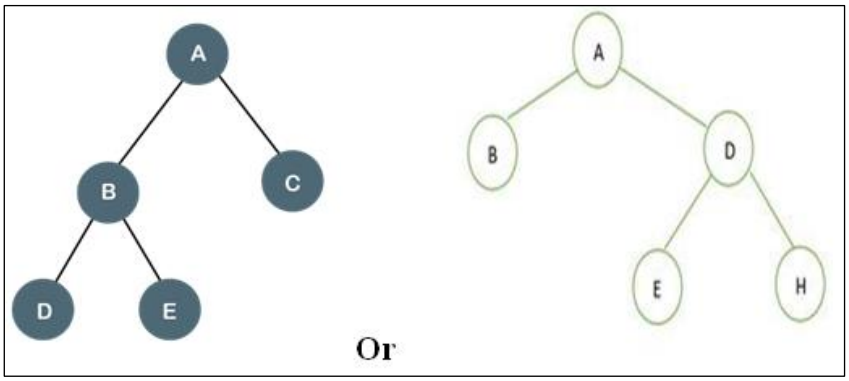


Fig.6.4 : Full Binary Tree

In the tree shown in Figure-6.4 we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

2. Complete Binary tree :

A complete binary tree is a binary tree type where all the leaf nodes must be on the same level. However, root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes.

A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

A complete binary tree is just like a full binary tree, but with three major differences :

1. Every level except the last level must be completely filled.
2. All the leaf elements must lean towards the left.

3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

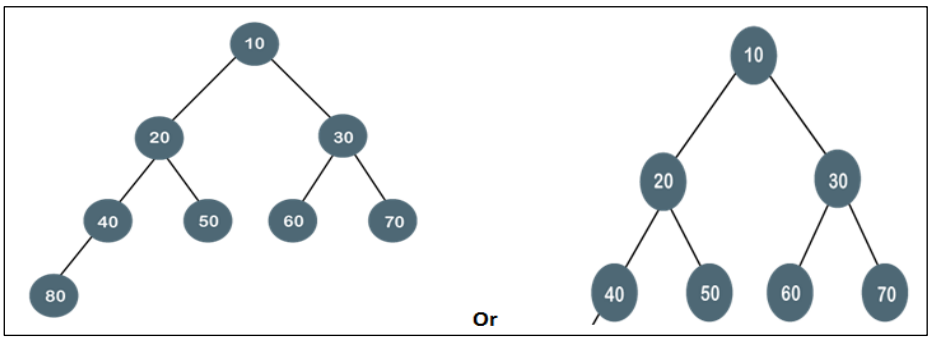


Fig.6.5 : Complete Binary Tree

The tree in Figure-6.5 is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

3. Perfect Binary Tree :

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level. That means a perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.

The following Figure-6.6 is an example of perfect binary tree.

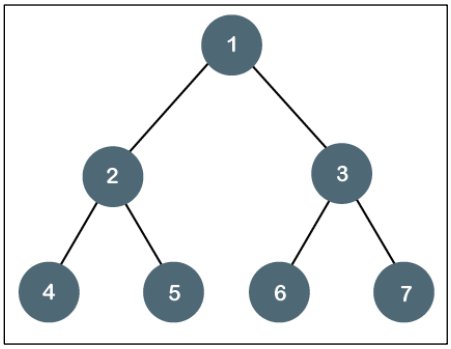


Fig. 6.6 : Perfect Binary Tree

4. Degenerate Binary tree :

Degenerate Binary tree is based on the number of children nodes have.

Degenerate Binary tree is a tree where every internal node has one child. A degenerate or pathological tree is a tree having a single child either left or right.

The following Figure-6.7 is three examples of degenerate binary trees.

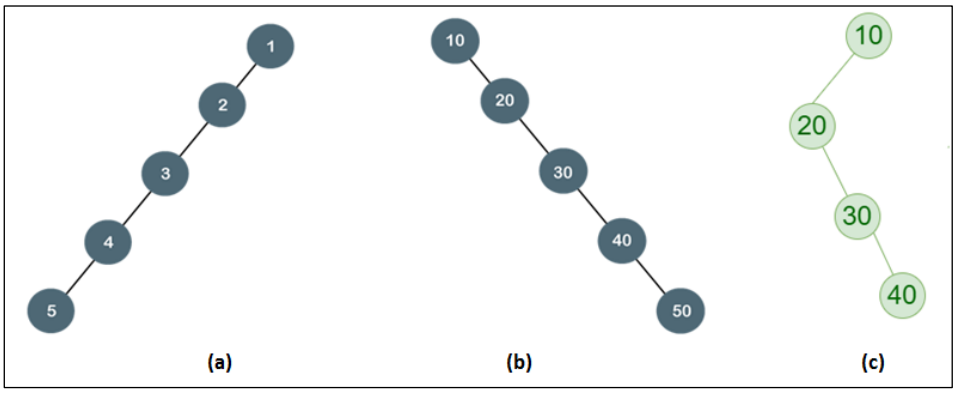


Fig. 6.7 : Degenerate binary trees.

5. Skewed binary tree :

Skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.

The following Figure-6.8 is example of Skewed binary trees.

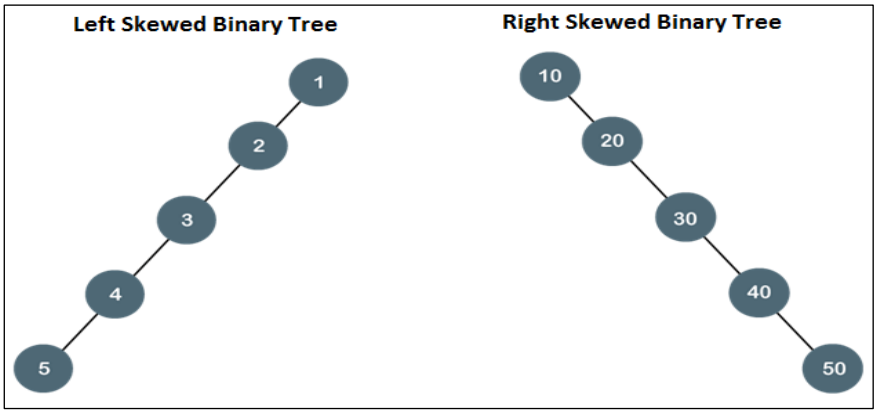


Fig. 6.8 : Skewed binary trees.

6. Extended Binary Tree :

A **binary tree** is called extended binary tree if every node of tree has zero or two children. It is also called 2-Tree.

Extended binary tree is a type of binary tree in which all the null sub tree of the original tree are replaced with special nodes called **external nodes** represented by rectangle (□) symbol whereas other nodes are called **internal nodes**

Any binary tree can be converted into an extended binary tree by replacing each empty sub-tree by **external nodes (**failure node).

The following Figure-6.9 is example of extended binary trees.

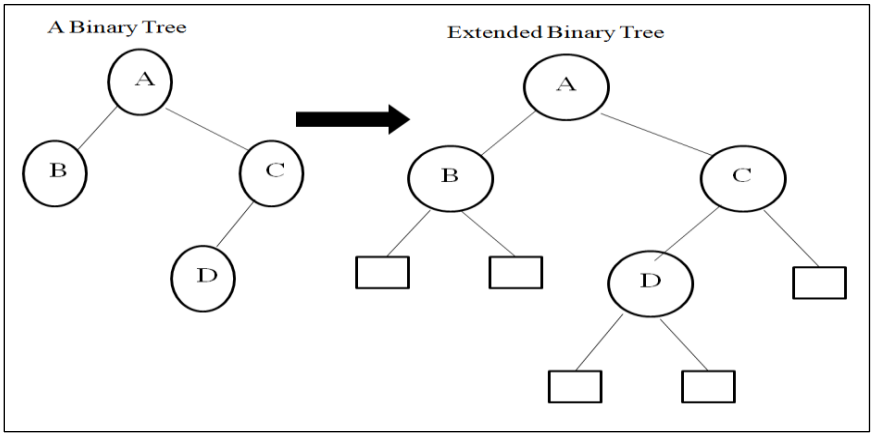


Fig. 6.9 : Extended binary trees.

7. Balanced Binary tree :

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

6.3 Representing binary trees in memory :

There are three types of storage representation of binary tree.

1. Sequential storage representation (Array)
2. Linked storage representation (Linked List)
3. Threaded storage representation (Linked List)

1. Sequential Storage Representation Of Binary Tree :

Sequential Storage Representation is also called as linear storage representation of binary tree. A Sequential storage representation can be done by one dimensional array. The size of one dimensional array can be calculated by using following formula.

Formula :

Array size = $\mathbf{2^{d+1} - 1}$

Where **d** is the depth (maximum levels of the tree) of the tree.

All the elements of binary tree can be stored into one dimensional array by following rules.

1. Position **0** indicates the size of an array.
2. Root node is stored at position **1**.
3. If the node is stored at position **n** then its left child node is stored at position **2*n** and its right child node is at **2*n + 1**.

In order to present a tree in sequential storage representation, the nodes are numbered sequentially level by left to right. Even empty nodes are numbered. When the data of the tree is stored in an array then the number appearing against the node will work as indices of the node in an array. Location number zero of the array can be used to store the size of the tree in terms of total number of nodes (existing or not existing).

Consider the following Figure-6.10, example of binary tree with numbering node.

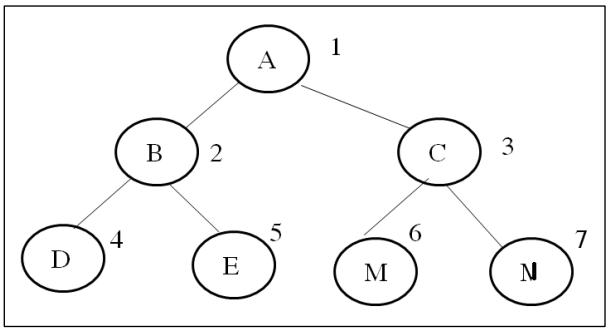


Fig. 6.10 : binary trees with 7 nodes

Array size = $2^{d+1} - 1$ Where d is the depth of the tree. (d=2 depth of above tree)

Array size = $2^{2+1} - 1 = 2^3 - 1 = 8 - 1 = 7$

Array size = 7

The array/sequential representation of tree in Figure-6.11 is as follows.

7	A	B	C	D	E	M	N
0	1	2	3	4	5	6	7

Fig. 6.11 : array/sequential representation of tree

Advantages :

- It is simple and easy to implement.
- It is good method for complete binary tree.
- It possible to find parent node for any child node.

(Position of a parent node = Position of child node / 2).

Disadvantages :

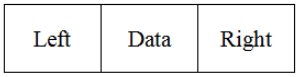
- There is wastage of memory (some of the memory locations are left empty).
- This method is wasteful for other binary tree.
- Insertion and deletion of a node required a lots of data movement.

2. Linked Storage representation of Binary Tree :

Linked representation of binary tree is more efficient than array representation. A linked storage representation can be done by doubly linked list.

A node of a binary tree consists of three fields i.e. Data, Address of the left child, and Address of the right child.

The node can be represented in linked format as:



Left and Right are pointer type fields. Left field holds the address of left child and Right field holds the address of right child.

Linked representation of a node can be defined as :

```
struct node
{
  int data;
 struct node *left-child;
  struct node *right-child;
}
```

The binary tree and its linked representation are shown in Fig. 6.12.

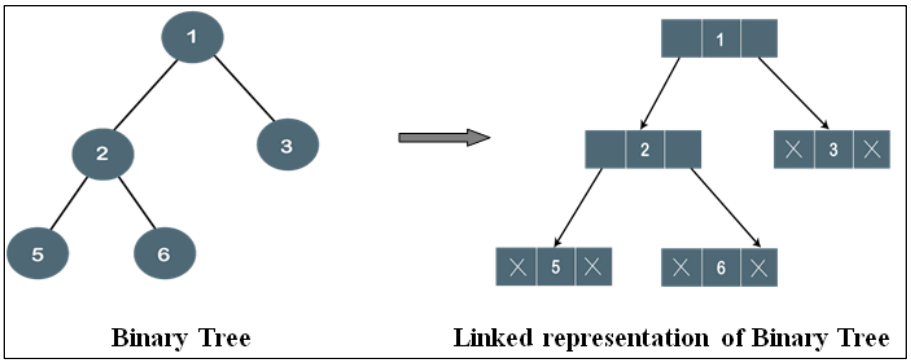


Fig. 6.12 : The binary tree and its linked representation.

In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain NULL pointer on both left and right parts.

Advantages :

- Linked representation is most efficient.

Disadvantages :

- There is wastage of memory space in null pointers.
- It is difficult to find parent node from the given child node.

- It is difficult to implement in old languages that do not support dynamic storage techniques.

3. Threaded storage representation of Binary Tree :

The linked representation of a binary tree T contains most of the entries in the left pointer field and right pointer field with NULL values. This space occupied by NULL entries can be efficiently utilized to store some kind of valuable information. These special pointers are called threads and the binary tree having such pointers is called a threaded binary tree. Threads in a binary tree are represented by a dotted line. Left or right link of a node can denote either a structural link or thread. Structural link can be represented as positive pointer value address While threads on other hand represented by negative address.

There are many ways to thread a binary tree :

A binary tree is threaded according to particular traversal order, such tree with a threads is called Threaded Binary tree -

1. A right threaded binary tree :

The right NULL pointer of each leaf node can be replaced by a thread to the successor of that node under **in-order** traversal called a right thread.

If thread appears in the right link field of a node then it will point to the next node that will appear on performing **in-order** traversal. Such trees are called **Right threaded binary trees**.

The Figure-6.13 shows Linked representation of right threaded binary tree.

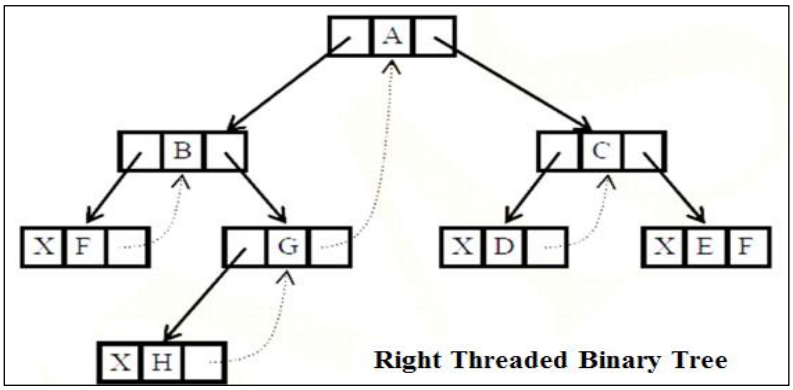


Fig. 6.13 : Right threaded binary

2. A left threaded binary tree :

The left NULL pointer of each node can be replaced by a thread to the predecessor of that node under **in-order** traversal called left thread

If thread appears in the left field of a node then it will point to the nodes **in-order** predecessor. Such trees are called **Left threaded binary trees.**

The Figure-6.14 shows Linked representation of left threaded binary tree.

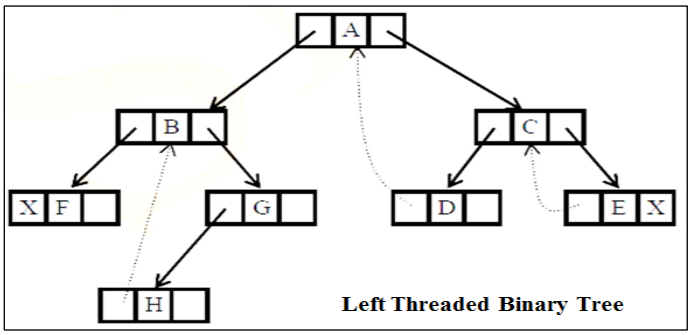


Fig. 6.14 : Left threaded binary.

3. A fully threaded tree :

Both left and right NULL pointers can be used to point to predecessor and successor of that node respectively under **in-order** traversal.

A threaded binary tree where only one thread is used is also known as one way threaded tree and where both threads are used is also known as two way threaded tree.

The Figure-6.15 shows Linked representation of full threaded binary tree.

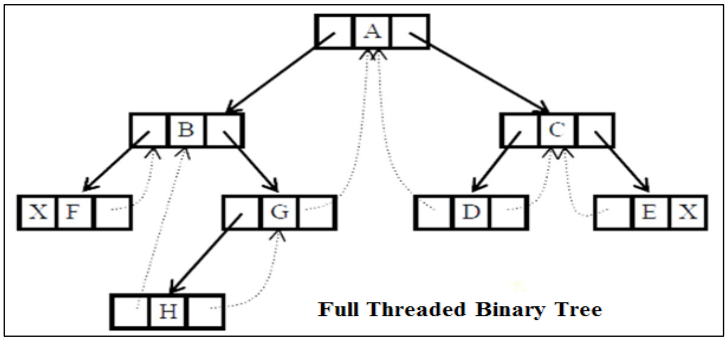


Fig. 6.15 : Full Threaded Binary Tree

6.4 Traversing binary trees :

The term 'tree traversal' means traversing or visiting each node of a tree :

Traversal is a process to visit all the nodes of a tree and may also print their values. As all nodes are connected via edges (links) we always start from the root (head) node. We cannot randomly access a node in a tree.

There are multiple ways to traverse a tree :

a. Preorder traversal (root, Left, Right)
b. Inorder traversal (Left, root, Right)
c. Postorder traversal (Left, Right, root)

The Figure 6.16: shows the binary tree and all inorder, preorder and postorder traversal techniques used to print the data stored in tree data structure.

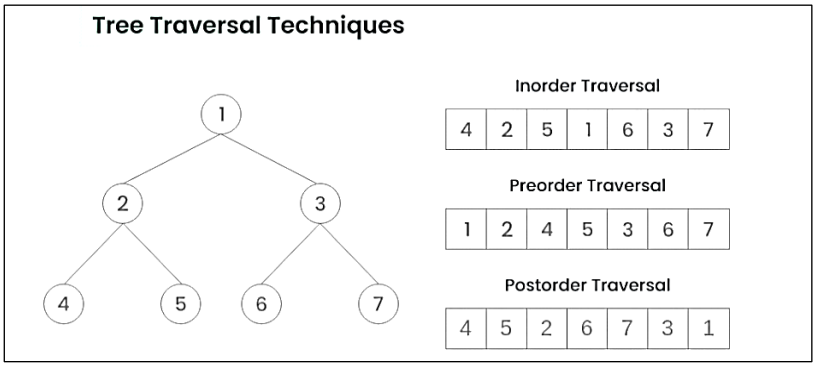


Fig. 6.16 : preorder traversal

A) Preorder traversal

Algorithm of Preorder traversal

Until all nodes of the tree are not visited

1. Visit the **root** node.
2. Traverse the **left** subtree recursively.
3. Traverse the **right** subtree recursively.

The example of the preorder traversal technique is shown in Figure 6.17

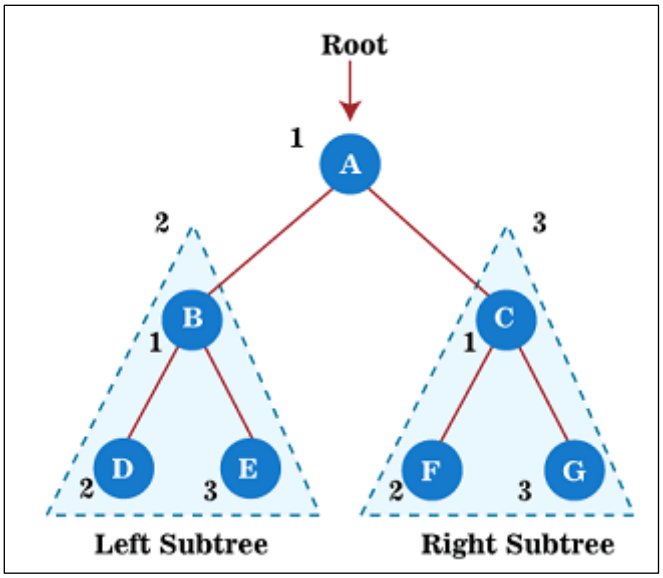


Fig. 6.17 : preorder traversal.

First, we traverse the root node **A;** then traverse its left subtree **B**, which will also be traversed in preorder.

So, for left subtree B, first, the root node **B** is traversed itself; after that, its left subtree **D** is traversed. Since node **D** does not have any children, move to right subtree **E**. As node E also does not have any children, the traversal of the left subtree of root node A is completed.

Now, move towards the right subtree of root node A that is C. So, for right subtree C, first the root node **C** has traversed itself; after that, its left subtree **F** is traversed. Since node **F** does not have any children, move to the right subtree **G**. As node G also does not have any children, traversal of the right subtree of root node A is completed.

The output of the preorder traversal of the above tree –

A → B → D → E → C → F → G

Algorithm of Preorder Traversal of Binary Tree :

PREORDER (T): Given Binary tree 'T' whose root node address given by pointer variable LPTR is left child and RPTR is right child. DATA is actual data value stored. This algorithm traverses a binary tree preorder in recursive manner.

1. [Check for empty tree]
   If (T==NULL) Then
   Write ('Empty Tree');
   Return
2. [Process the root node]
   Write (DATA (T));
3. [Process the Left node]
   If (LPTR (T) ! = NULL) Then
   Call PREORDER (LPTR (T));
4. [Process the Right node]
   If (RPTR (T) ! = NULL) Then
   Call PREORDER (RPTR (T));
5. [Finished]
   Exist.

Function of Preorder Traversal of Binary Tree :

```
Void BTree :: Preorder (Node *h)
    {
        if (h!=0)
        {
            cout<<h→data<<"/t";
```

```
            Preorder(h→leftchild);
            Preorder(h→rightchild);
        }
    }
```

B) Inorder traversal:

Inorder traversal technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal. So, in the inorder traversal, each node is visited in between of its subtrees.

Algorithm of In traversal :

Until all nodes of the tree are not visited

1. Traverse the **left** subtree recursively.
2. Visit the **root** node.
3. Traverse the **right** subtree recursively.

The example of the preorder traversal technique is shown in Figure 6.18

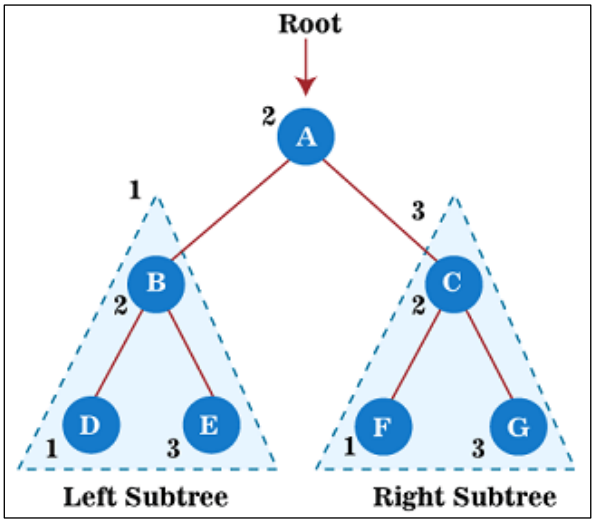


Fig. 6.18 : Inorder traversal

In inorder traversal technique first traverse the left subtree **B** that will be traversed in inorder. After that, we will traverse the root node **A**. And finally, the right subtree **C** is traversed in inorder.

So, for left subtree **B**, first, its left subtree **D** is traversed. Since node **D** does not have any children, so after traversing it, node **B** will be traversed, and at last, right subtree of node B, that is **E**, is traversed. Node E also does not have any children; therefore, the traversal of the left subtree of root node A is completed.

After that, traverse the root node of a given tree, i.e., **A**.

At last, move towards the right subtree of root node A that is C. So, for right subtree C; first, its left subtree **F** is traversed. Since node **F** does not have any children, node **C** will be traversed, and at last, a right subtree of node C, that is, **G**, is traversed. Node G also does not have any children; therefore, the traversal of the right subtree of root node A is completed.

As all the nodes of the tree are traversed, the inorder traversal of the given tree is completed. The output of the inorder traversal of the above tree is -

D → B → E → A → F → C → G

Algorithm of Inorder Traversal of Binary Tree :

INORDER (T) : Given Binary tree 'T' whose root node address given by pointer variable LPTR is leftchild and RPTR is rightchild. DATA is actual data value stored. This algorithm traverses a binary tree inorder in recursive manner.

1. [Check for empty tree]
   If (T==NULL) Then
   Write ('Empty Tree');
   Return
2. [Process the left node]
   If (LPTR (T) ! = NULL) Then
   Call INORDER (LPTR (T));
3. [Process the root node]
   Write (DATA(T));
4. [Process the right node]
   If (RPTR (T) ! = NULL) Then
   Call INORDER (RPTR (T));
5. [Finished]
   Exist.

Function of Inorder Traversal of Binary Tree :

```
Void BTree :: Inorder (Node *h)
{
        If (h! =0)
        {
          Inorder (h→leftchild);
          cout<<h→data<<"/t";
          Inorder (h→rightchild);
        }
}
```

C) Postorder traversal :

Postorder traversal technique follows the '**left-right-root**' policy. The first left subtree of the root node is traversed, after that recursively traverses the right subtree, and then the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

In postorder traversal, each node is visited after both of its subtrees.

Algorithm of **postorder traversal :**

Until all nodes of the tree are not visited

1. Traverse the **left** subtree recursively.
2. Traverse the **right** subtree recursively.
3. Visit the **root** node.

Example **of the postorder traversal technique is given in Figure 6.19**

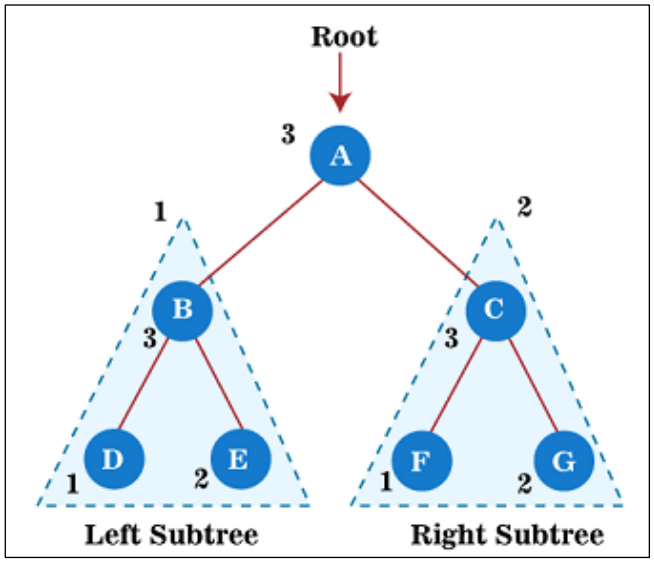


Fig. 6.19 : Postorder traversal technique.

In the postorder traversal first traverse the left subtree **B** that will be traversed in postorder. Then traverse the right subtree **C** in postorder. And finally, the root node of the above tree, i.e., **A**, is traversed.

Now for left subtree **B**, first, its left subtree **D** is traversed. Since node **D** does not have any children, traverse the right subtree **E**. As node **E** also does not have any children, move to the root node **B.** After traversing node **B,** the traversal of the left subtree of root node **A** is completed.

Now, move towards the right subtree of root node **A** that is **C**. So, for right subtree **C**, first its left subtree **F** is traversed. Since node **F** does not have any children, traverse the right subtree **G**.

As node G also does not have any children, therefore, finally, the root node of the right subtree, i.e., **C,** is traversed. The traversal of the right subtree of root node A is completed.

At last, traverse the root node of a given tree, i.e., **A**. After traversing the root node, the postorder traversal of the given tree is completed and all the nodes of the tree are traversed. The output of the postorder traversal of the above tree is -

D → E → B → F → G → C → A

Algorithm of Postorder Traversal of Binary Tree :

POSTORDER (T): Given Binary tree 'T' whose root node address given by pointer variable LPTR is leftchild and RPTR is rightchild. DATA is actual data value stored. This algorithm traverses a binary tree preorder in recursive manner.

1. [Check for empty tree]
   If (T==NULL) Then
   Write ('Empty Tree');
   Return
2. [Process the left node]
   If (LPTR (T) ! = NULL) Then
   Call POSTORDER (LPTR (T));
3. [Process the right node]
   If (RPTR (T) ! = NULL) Then
   Call POSTORDER (RPTR (T));
4. [Process the root node]
   Write (DATA (T));
5. [Finished]
   Exist.

Function of Postorder Traversal of Binary Tree :

```
Void BTree :: Postorder (Node *h)
{
        if (h!=0)
        {
            Postorder (h→leftchild);
            Postorder (h→rightchild);
            cout<<h→data<<"/t";
        }
}
```

6.5 Threaded Binary Tree :

The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. In a Threaded Binary Tree, the nodes will store the in-order predecessor / successor instead of storing NULL in the left/right child pointers.

So the basic idea of a threaded binary tree is that for the nodes whose right pointer is null, we store the in-order successor of the node (if-exists), and for the nodes whose left pointer is null, we store the in-order predecessor of the node(if-exists).

One thing to note is that the leftmost and the rightmost child pointer of a tree always points to null as their in-order predecessor and successor do not exist.

The Figure 6.20 shows the example of threaded binary tree.

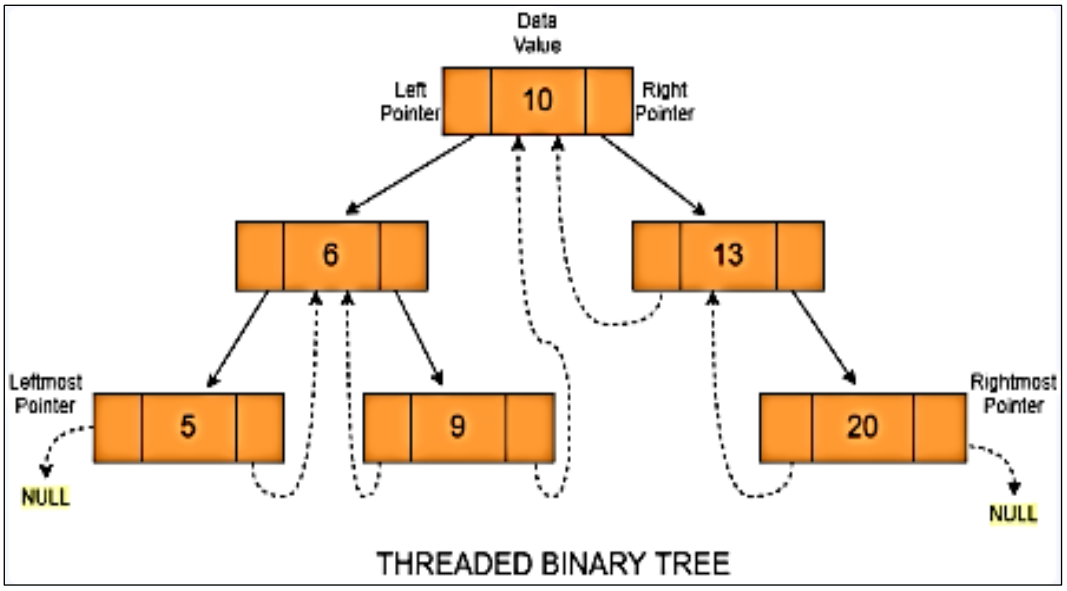


Figure 6.20 : Threaded binary tree.

There are two types of threaded binary trees :

1. **Single (One-way) Threaded Binary Tree** : Where a NULL right pointers is made to point to the inorder successor (if successor exists)
2. **Double (Two-way) Threaded Binary Tree** : Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

These techniques are discussed earlier in storage representation of binary tree.

6.6 Graph : Types, representation in memory :

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph.

A graph G is a set of vertices (V) and set of edges (E). The set V is a finite, nonempty set of vertices. The set E is a set of pair of vertices representing edges.

G = (V, E) is a graph and

V (G) = Vertices of graph G,

E (G) = Edges of graph G

Figure 6.21 shows a graph G1, G2, and G3.

The set of Vertices V (G):

V (G1) = {A, B, C, D, E, F}

V (G2) = {A, B, C, D, E, F}

V (G3) = {A, B, C}

The set of Edges E (G):

E (G1) = {(A,B),(A,C),(B,C),(B,D),(D,E),(D,F), (E, F)}

E (G2) = {(A,B),(A,C),(B,D),(C,E),(C,F)}

E (G3) = {(A, B), (A, C), (C, B)}

The Figure 6.21 shows an example of various graph in which the vertices are represented by circles and the edges by lines.

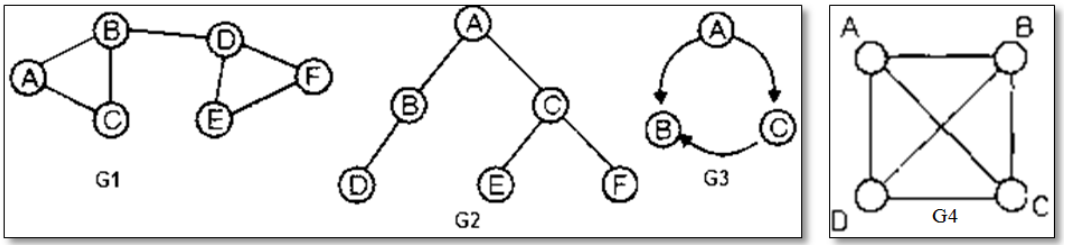


Fig. 6.21 : Various graphs.

- **Directed / Undirected Edge :** An edge with an orientation (arrow head) is a directed edge, while an edge with no orientation is our undirected edge.
- **Adjacent Vertex :**
  Two vertices V_1 and V_2 are said to be adjacent if there is an edge between V_1 and V_2. In Figure 6.21(G1), adjacent vertex of A is B and C.
- **Path :** A path from vertex V_0 to V_n is a sequence of vertices V_0, V_1, V_2....., V_{n-1}, and Vn. Here, V_0 is adjacent to V_1, V_1 is adjacent to V_2 and V_{n-1} is adjacent to V_n. The length of a

path is the number of edges on the path. A path with n vertices has a length of n - 1. A path is simple if all vertices on the path except possibly the first and last, are distinct. In Figure 6.21(G1) there is path between A and F is A-B-D-F.

- **Self Edges or Self Loops :**
  An edge of the form (V, V) is known as self edge or self loop. For self edge/loop starting and ending vertex of edge is same.
- **Sub-graph :** A sub-graph of G is a graph G, such that V (G1) is a subset of V (G) and E (G1) is a subset of E (G).
  The sub-graphs of Figure 6.21(G4) is shown in following Figure 6.22.

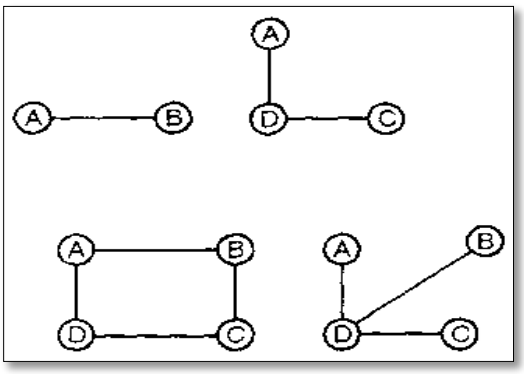


Fig. 6.22: Sub-Graphs

- **Degree of Vertex :**
  The total number of edges linked to a vertex is called its degree of vertex. The degree of a vertex is the number of edges incident to that vertex. When the degree of a vertex is 0, it is an isolated vertex. In Figure 6.21(G1) node degree of node B and D is 3.
- **In-degree of Vertex :**
  The in-degree of a vertex is the total number of edges coming to that node. A vertex having only incoming edges and no outgoing edges is called a sink. When in-degree of a vertex is one and out-degree is zero then such a vertex is called a pendant vertex. In Figure 6.21(G3) the in-degree of node B is 2 and out-degree is 0.
- **Out-degree of Vertex :**
  The out-degree of a node is the total number of edges going out from that node. A vertex, which has only outgoing edges and no incoming edges, is called a source. In Figure 6.21(G3) the out-degree of node B is 0.

Applications of graph data structure in various fields are :

- **Computer Science :** Graphs are used to model many problems and solutions in computer science, such as representing networks, web pages, and social media

connections. Graph algorithms are used in path finding, data compression, and scheduling.

- **Social Networks :** Graphs represent and analyze social networks, such as the connections between individuals and groups.
- **Transportation :** Graphs can be used to model transportation systems, such as roads and flights, and to find the shortest or quickest routes between locations.
- **Computer Vision :** Graphs represent and analyze images and videos, such as tracking objects and detecting edges.
- **Natural Language Processing :** Graphs can represent and analyze text, such as in syntactic and semantic dependency graphs.
- **Telecommunication :** Graphs are used to model telecommunication networks, such as telephone and computer networks, and to analyze traffic and routing.
- **Circuit Design :** Graphs are used in the design of electronic circuits, such as logic circuits and circuit diagrams.
- **Bioinformatics :** Graphs model and analyze biological data, such as protein-protein interaction and genetic networks.
- **Operations research :** Graphs are used to model and analyze complex systems in operations research, such as transportation systems, logistics networks, and supply chain management.
- **Artificial Intelligence :** Graphs are used to model and analyze data in many AI applications, such as machine learning, Artificial Intelligence, and natural language processing.
- **etc**. not limited to this list.

6.7 Graph : Types in Memory :

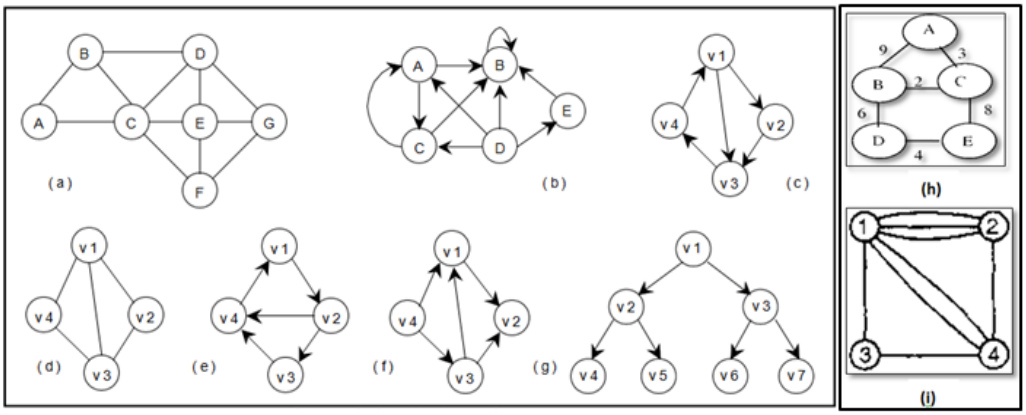


Fig. 6.23 : Various graphs.

Undirected Graph : If all the edges in a graph are undirected, then the graph is an undirected graph. The graphs in Figure 6.23(a), (d) are undirected graphs.

Directed Graph/ Digraph : If all the edges are directed; then the graph is a directed graph. The graph of Figure 6.23 (b) is one of the directed graph. A directed graph is also called as digraph.

Connected Graph : A graph **G** is connected if and only if, there is the simple paths between any two nodes in graph **G**.

Complete Graph : A graph **G** is said to be complete if every node **a** in **G** is adjacent to every other node **v** in **G**. A complete graph with n nodes will have n(n- 1)/2 edges. For example, Figure 6.21(G4) is complete graph.

Cyclic Graph : A graph in which there is path that begins and ends at the same vertex. In Figure 6.21 Graph G1 and G4 are examples of a cyclic graph means graph with cycle.

A B C A is a cycle of length 3

D E F is a cycle of length 3

Regular Graph :

A graph 'G' is said to be regular graph of degree 'R', If all vertices in graph 'G' are of same degree 'R'. In regular graph every vertices has same degree. A complete graph is regular graph. In Figure 6.21 graph G4 is example of a regular graph with same degree of every vertices is 3.

Weighted Graph :

A weighted graph is a graph in which edges are assigned some value. Most of the physical situations are shown using weighted graph. An edge may represent a highway link between two cities. The graph of Figure 6.23 (h) is example of weighted graph with labled edges indicated weight of edge.

Multigraph :

A graph with multiple occurrences of the same edge is known as a multigraph. The graph of Figure 6.23 (i) is example of multigraph graph with more than one edges between vertices 1 and 2 as well as vertices 1 and 4.

Tree :

A tree is a connected graph without any cycle (acyclic graph). The graphs of Fig. 7(a) are not trees as they contain cycles. The graph of Figure 6.23 (g) is example of tree type graph.

Spanning Trees :

A spanning tree of a graph G = (V, E) is a connected sub-graph of G having all vertices of G and no cycles in it. If the graph G is not connected then there is no spanning tree of G. A graph may have multiple spanning trees.

The Figure 6.24 shows graph and some of its spanning trees in (b) respectively.

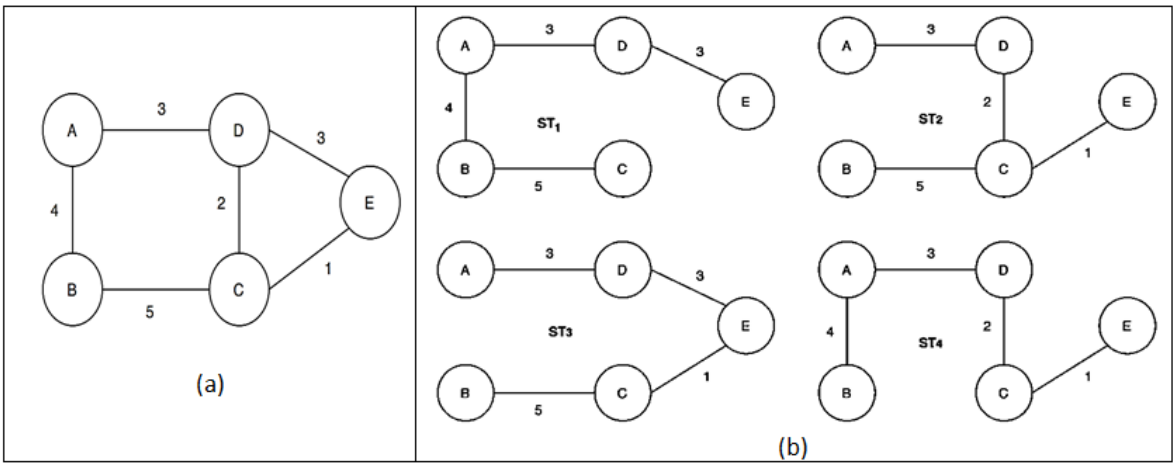


Fig. 6.24 : Shows the various spanning trees.

6.8 Graph : Representation in Memory :

There are three popular data structures used to represent graph :

1. Matrix representation of Graph
2. List representation of Graph
3. Multi list representation of Graph

Depending upon the application, we can use adjacency matrix representation, adjacency list representation or Multi list representation of Graph.

1. **Matrix representation of graph :**

   An adjacency matrix is a V×V array of graph G(V,E) having V number of vertices. The entry in the matrix will be either 0 or 1. If there is an edge between vertices A and B, we set the value of the corresponding cell to 1 otherwise we simply put 0.

   Figure 6.25 and Figure 6.26 shows the adjacency matrix representation of directed and undirected graph respectively.

2. **List representation of graph :**

   Adjacency lists, in simple words, are the array of linked lists. We create an array of vertices and each entry in the array has a corresponding linked list containing the neighbors. In other words, if a vertex 1 has neighbors 2, 3, 4, the array position corresponding the vertex 1 has a linked list of 2, 3, and 4.

   Figure 6.25 and Figure 6.26 shows the adjacency List representation of directed and undirected graph respectively.

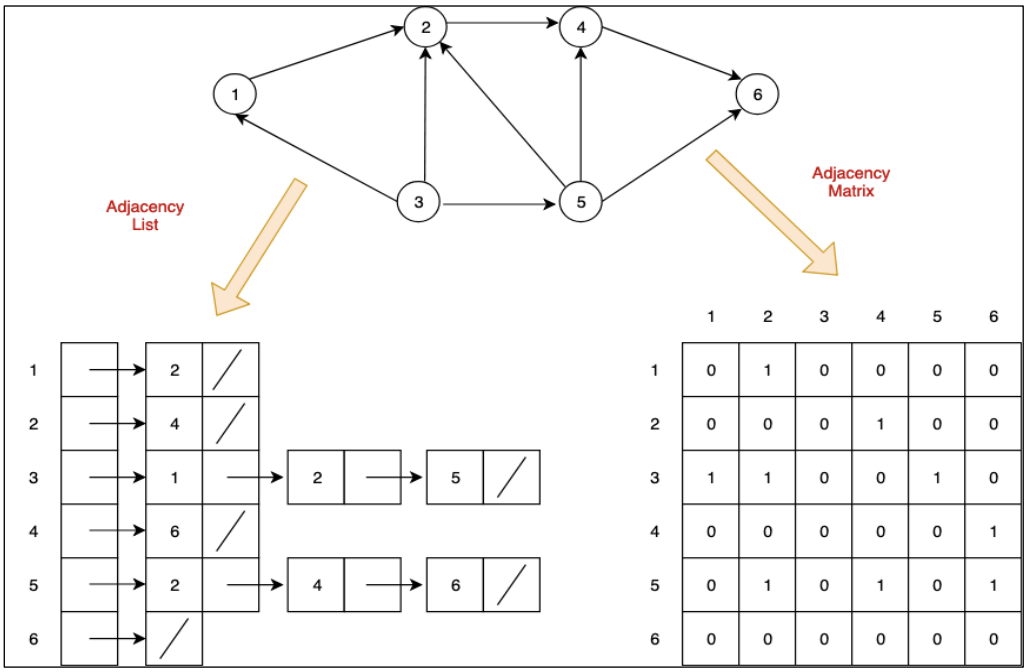


Fig. 6.25 : The Adjacency Matrix representation and List representation of directed graph.

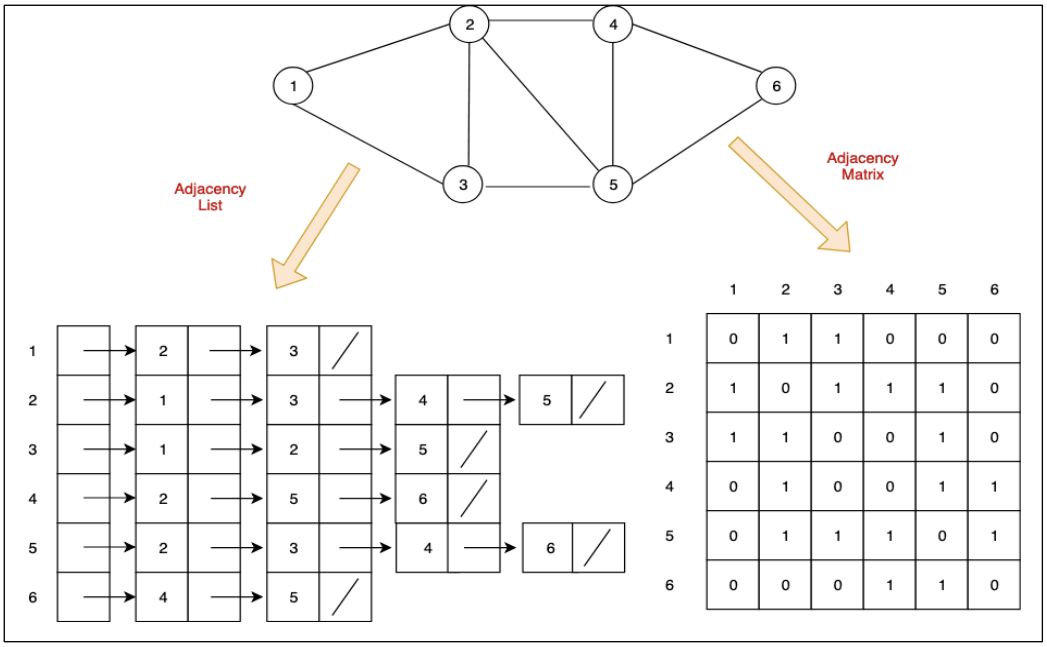


Fig. 6.26 : The Adjacency Matrix representation and List representation of undirected graph.

3. **Multi list representation of Graph** : In the adjacency list representation of an undirected graph each edge (Vi, Vj) is represented by two entries, one on the list for vertex-Vi and the other on the list for vertex-Vj. In some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as already having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (that means, lists in which nodes may be shared among several lists). For each edge there will be exactly one node, but this node will be in two lists, that is the adjacency lists for each of the two nodes it is incident too.

The node structure for Multilist now becomes as shown below.

M	V_1	V_2	LINK (I) For V_1	LINK (j) For V_2

M is a one bit **mark** field that may be used to indicate whether or not the edge has been examined. The storage requirements are the same as for normal adjacency lists except for the addition of the mark bit M.

Figure 6.27 shows the Multi list representation of graph.

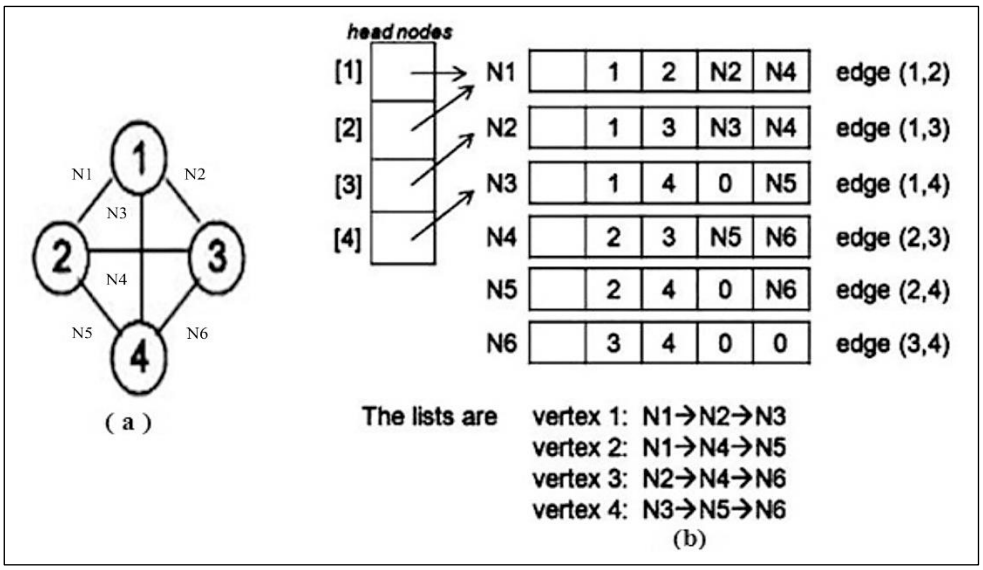


Fig. 6.27 : The Adjacency Matrix representation and List representation of undirected graph.

Difference between Graph and Tree :

The basis of Comparison	**Tree**	**Graph**
Definition	Tree is a non-linear data structure.	Graph is a non-linear data structure.
Structure	It is a collection of nodes and edges.	It is a collection of vertices/nodes and edges.
Structure cycle	A tree is a type of graph that is connected, acyclic (meaning it has no cycles or loops), and has a single root node.	A graph can be connected or disconnected, can have cycles or loops, and does not necessarily have a root node.
Edges	If there is n nodes then there would be n-1 number of edges.	Each node can have any number of edges.
Types of Edges	They are always directed.	They can be directed or undirected.
Root node	There is a unique node called root(parent) node in trees.	There is no unique node called root in graph.
Loop Formation	There will not be any cycle.	A cycle can be formed.
Traversal	We traverse a tree using in-order, pre-order, or post-order traversal methods.	For graph traversal, we use Breadth-First Search (BFS), and Depth-First Search (DFS).
Applications	For game trees, decision trees, the tree is used.	For finding shortest path in networking graph is used.
Node relationships	In a tree, each node (except the root node) has a parent node and zero or more child nodes.	In a graph, nodes can have any number of connections to other nodes, and there is no strict parent-child relationship.
Commonly used for	Trees are commonly used to represent data that has a hierarchical structure, such as file systems, organization charts, and family trees.	Graphs are commonly used to model complex systems or relationships, such as social networks, transportation networks, and computer networks.
Connectivity	In a tree, each node can have at most one parent, except for the root node, which has no parent.	In a graph, nodes can have any number of connections to other nodes.

***

As Per New Syllabus of Kavayitri Bahinabai Chaudhari North Maharashtra University, Jalgaon

Bachelor of Computer Application (BCA) (w.e.f. 2023-24)

SEMESTER I

BCA 101	Fundamentals of Accounting
BCA 102	Fundamentals of Computer
BCA 103	Programming in C – I
BCA 104	Web Design – I

SEMESTER II

BCA 201	Professional Communication Skill
BCA 202	Database Management System
BCA 203	Programming in C – II
BCA 204	Web Design - II

SEMESTER III

BCA 301	Fundamental Mathematics and Statistics
BCA 302	Operating System
BCA 303	Programming in C++
BCA 304A	Web Development Technology – I
BCA 304C	Python Programming

SEMESTER IV

BCA 401	Software Engineering
BCA 402	Data Structure
BCA 403	Java Programming
BCA 404A	Web Development Technology - II
BCA 404C	Artificial Intelligence

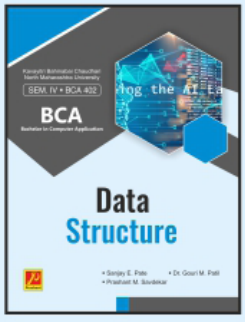
SEMESTER V

BCA 501	Employability Skill
BCA 502	E-Commerce and M-Commerce
BCA 503	Cloud Computing Application
BCA 504A	Web Development Technology – III
BCA 504C	Machine Learning

SEMESTER VI

BCA 601	Entrepreneurship Development
BCA 602	Cyber Security
BCA 603	Android Application Development
BCA 604A	Web Development Technology – IV
BCA 604C	Data Mining

SEMESTER IV



Also Available in **e-Book**

BCA ₹ 115
ISBN 978-81-19120-36-9
Prashant
www.prashantpublications.com
prashantpublication.jal@gmail.com