I B.SC - SECOND SEMESTER

COMPUTER SCIENCE

Data Structures Using C

Question Bank

Prepared By

L.Pulla Reddy
M.Sc Computer Science

K.Sreenivasulu
M.Sc Computer Science, B.Ed

I B.Sc-Sem 2 - DATA STRUCTURES USING C
(2021 Batch)

UNIT – I:

Introduction to Data Structures:

1. Introduction to the Theory of Data Structures
2. Data Representation
3. Abstract Data Types
4. Data Types
5. Primitive Data Types
6. Data Structure and Structured Type
7. Atomic Type
8. Difference between Abstract Data Types Data Types and Data Structures
9. Refinement Stages

UNIT – II:

Arrays:

1. Introduction to Linear and Non- Linear Data Structures
2. One- Dimensional Arrays
3. Two- Dimensional arrays
4. Multidimensional Arrays
5. Array Operations
6. Pointers and Arrays
7. Overview of Pointers

Linked Lists:

1. Introduction to Lists and Linked Lists
2. Dynamic Memory Allocation
3. Single Linked List
4. Doubly Linked List
5. Circular Linked List
6. Atomic Linked List
7. Basic Linked List Operations
8. Linked List in Arrays
9. Linked List versus Arrays

UNIT – III:

Stacks:

1. Introduction to Stacks
2. Stack as an Abstract Data Type
3. Representation of Stacks through Arrays
4. Representation of Stacks through Linked Lists
5. Applications of Stacks
6. Stacks and Recursion

Queues:

1. Introduction
2. Queue as an Abstract data Type
3. Representation of Queues
4. Circular Queues
5. Double Ended Queues (or Dequeue)
6. Priority Queues
7. Application of Queues

UNIT – IV:

Binary Trees:

1. Introduction to Non- Linear Data Structures
2. Introduction Binary Trees
3. Types of Trees
4. Basic Definition of Binary Trees
5. Properties of Binary Trees
6. Representation of Binary Trees
7. Operations on a Binary Search Tree
8. Binary Tree Traversal
9. Counting Number of Binary Trees
10. Applications of Binary Tree

UNIT – V:

Searching and sorting:

1. Introduction to Sorting
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Introduction to Searching
6. Linear or Sequential Search
7. Binary Search
8. Indexed Sequential Search

Graphs:

1. Introduction to Graphs
2. Terms Associated with Graphs
3. Sequential Representation of Graphs
4. Linked Representation of Graphs
5. Traversal of Graphs
6. Spanning Trees
7. Shortest Path
8. Application of Graphs

DATA STRUCTURES USING C LAB PROGRAMS

1. Write a program to read 'N' numbers of elements into an array and also perform the following operation on an array

    a. Add an element at the begging of an array

    b. Insert an element at given index of array

    c. Update a element using a values and index

    d. Delete an existing element

2. Write a program using stacks to convert a given

    a. postfix expression to prefix

    b. prefix expression to postfix

    c. infix expression to postfix

3. Write Programs to implement the Stack operations using an array

4. Write Programs to implement the Stack operations using Liked List.

5. Write Programs to implement the Queue operations using an array.

6. Write Programs to implement the Queue operations using Liked List.

7. Write a program for arithmetic expression evaluation.

8. Write a program for Binary Search Tree Traversals

9. Write a program to implement dequeue using a doubly linked list.

10. Write a program to search an item in a given list using the following Searching Algorithms

    a. Linear Search

    b. Binary Search.

11. Write a program for implementation of the following Sorting Algorithms

    a. Bubble Sort

    b. Insertion Sort

    c. Quick Sort

UNIT – I
INTRODUCTION TO DATA STRUCTURES

Introduction to the Theory of Data Structures, Data Representation, Abstract Data Types, Data Types, Primitive Data Types, Data Structure and Structured Type, Atomic Type, Difference between Abstract Data Types, Data Types, and Data Structures, Refinement Stages

INTRODUCTION TO THE THEORY OF DATA STRUCTURES

- Computer is an electronic device which is used for data processing and manipulation
- When programmer collects this type of data for processing, he would require storing all of them in computer's main memory.
- In order to make computer work we need to know
  - Representation of data in computer
  - Accessing the data
  - How to solve problems step-by-step
- For doing the above tasks, we use data structures
- Data structure is the branch of computer science that unleashes the knowledge of how the data should be organized, how the flow of data should be controlled and how a data structure should be designed and implemented to reduce the complexity and increase the efficiency of the algorithm.
- The theory of structures not only introduces to the data structures, but also helps to understand and use the concept of abstraction, analyses problems step by step and develop algorithms to solve real world problems.
- It enables us to design and implement various data structures, for example, the stacks. Queues, linked lists, trees and graphs.

Q. DATA REPRESENTATION

In computers, various methods are used to represent data. The basic unit of data representation is a **bit.** The value of a bit is either 0 or 1. Eight bits together form one **byte** which represents a **character** and one or more than one characters are used to form a **string.**

1. Integer Representation

- An integer representation is used for storing integer values without fractional parts
- The non-negative data is represented using **binary number system**.
- In this, each bit position represents the power of 2.
- For example, 110 represents the integer as $1x\ 2^2 + 1x2^1 + 0x\ 2^0 + = 4+ 2 + 0 = 6$

For negative binary numbers, we use **one's complement** and **two's complement.**

→In **one's complement** method the number is represented by complementing each bit, i.e. changing each bit in its value to the opposite bit setting.

→In **two's complement** method, 1 is added to one's complement representation of the negative number.

For example, -38 is represented by 1 1 0 1 1 0 0 1 which on adding 1 to it will become

```
  1 1 0 1 1 0 0 1
          + 1
  ----------------
  1 1 0 1 1 0 1 0    which represents -38 in two's complement notation.
```

2. Real Number Representation

- In computers, Real number representation (or) **floating-point notation** is used to represent the real numbers (i.e., numbers with fractional parts)
- Floating point number contains integer, decimal point(.) and exponent notation('e').

  (mantissa)$^{\text{exponent}}$

  For example, 20.05, 99.9, −50.12, 6.02e23

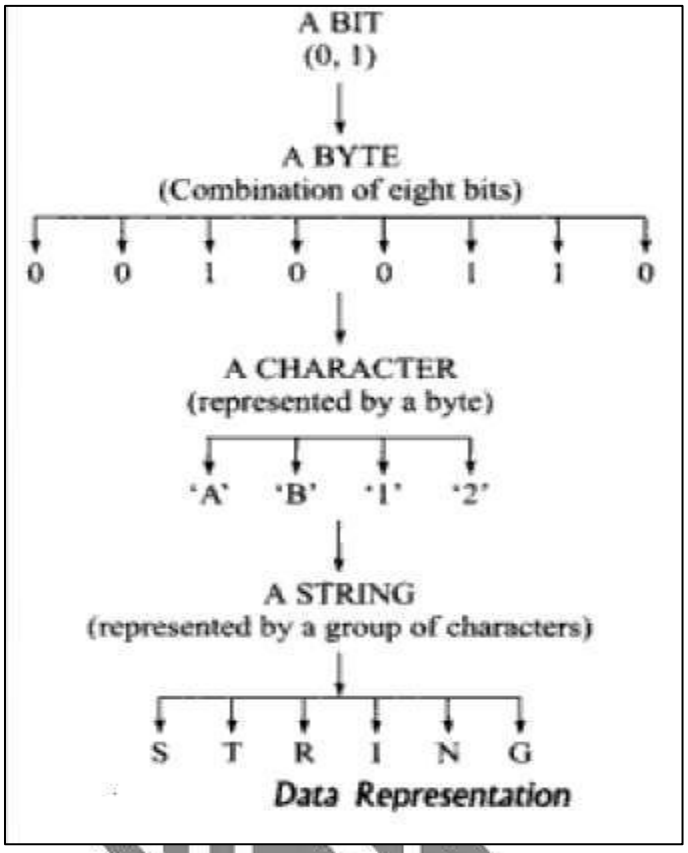
3. Character Representation

- In computer, character representation is used to represent the characters
- There are different codes available to store data in character form such as BCD, EBCDIC and ASCII. (the ASCII codes for A-z are 65-90,a-z are 97-122)

For example

- 01000001 is used to represent the character 'A'
- 01100001 is used to represent character 'a'.

4. Boolean: A Boolean type is a single-bit type that can be either true (1) or false (0).



Q. Write a detail note on 1. ADT 2.Composite types 3. Primitive types

Data Type: Data type defines what type of value to store on a variable. (or) A data type defines a set of values and the allowable operations on those values.

Different data types include:

1. Abstract data types (ADT)
2. Composite types
3. Primitive types

1. Abstract data types (ADT):

Abstract data types are mathematical models of a set of data values or information that share similar behaviors. Abstract data types are used in algorithms and also data items.

(OR)

ADT is a user defined data type which encapsulates a range of data values and their functions.

(OR)

An abstract data type is a definition of new type, describes its properties and operations.

Ex:
—Stack is a ADT which contains push( ), pop( ) operations.
—Linked Listll is a ADT which contains insert( ), delete( ) operations etc.

In an ADT, we encapsulate the data and the operations on data and we hide them from the user.
An abstract data type consists of
i) Declaration of data
ii) Declaration of operations

Advantages:

- Code is easier to understand.
- Implementations of ADTs can be changed without requiring changes to the program that uses the ADTs.
- ADTs can be reused in future programs.

2. Composite types:
Composite data type is any data type which can be constructed in a program using the programming languages like primitive data types and other data types.
Example: Structures, unions in c

```
struct Account
{
int account_number;
char first_name;
char last_name;
float balance;
};
```

3. **Primitive types:**
Primitive types are data types provided by a programming language. Primitive types are also known as built-in types or basic types. Primitive types may include:
1. Character (character, char);
2. Integer (integer, int, short, long, byte) with a variety of precisions;
3. Floating-point number (float, double, real, double precision);
4. Boolean having the values true and false.

Basic primitive types:

SIZE	NAME	SIGNED INTEGER RANGE	UNSIGNED INTEGER RANGE
8 bits	Byte	-128 to + 127	0 to 255
16 bits	Integer	-32,768 to +32,767	0 to 65,535
32 bits	Double Long integer	-2,147,483,648 to +2,147,483,648	0 to 4,294,967,295
64 bits	Long(JAVA)	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,808	0 to 18,446,744,073,709,551,615

→**Integer number type:**
An integer number type is used to store integer values without fractional part. Integers may be either signed (allowing Negative values) or unsigned (non-negative values only).
Examples: 10, -20

→**Floating-point number type:**
A floating-point number type is used to store integer values with fractional part. Floating point number contains integer, decimal point (.) and exponent notation (e).
Examples: 20.05, 99.9, –50.12, 6.02 e23

→**Boolean:** A Boolean type is a single-bit type that can be either true (1) or false (0).

→**Characters and strings:** A character type ("char") is used to store character values. It may contain a single letter, digit, punctuation mark, or control character placed in between single quotation marks. The group of Characters is called as strings which are placed in between double quotes.
Examples: 'L', 'P', "pullareddy"

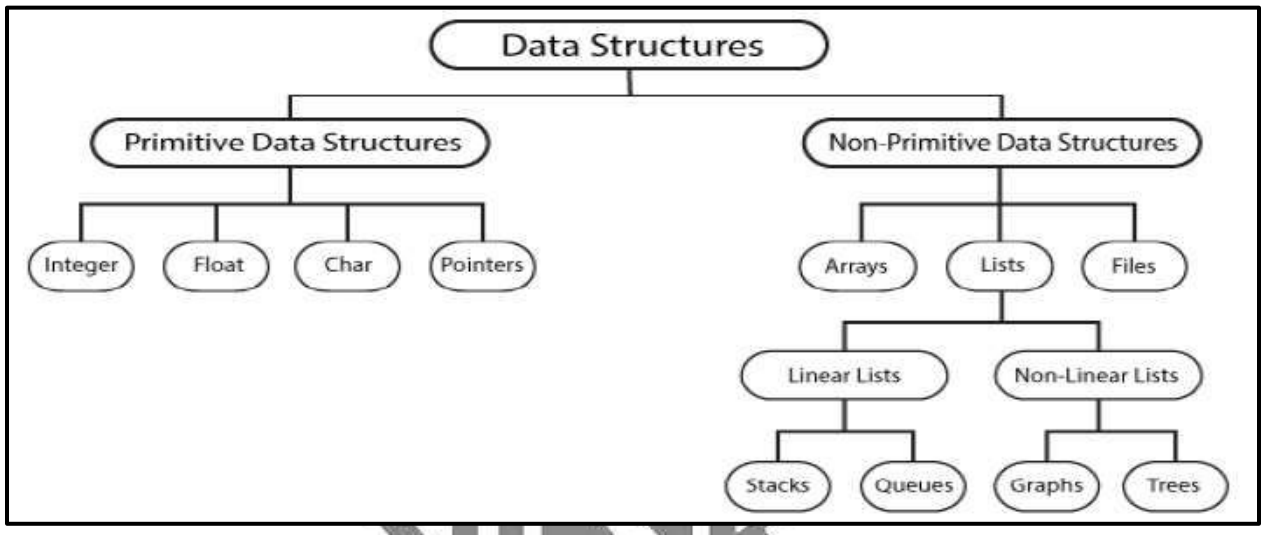
Q. Define data structures. List various types of data structures.

(OR)

Write short notes on Primitive and Non-primitive data structures.

Data Structure:

Data is organized in many different ways. The logical (or) mathematical model of a particular organization of a data is called **data structure**.



1. Primitive Data Structures

- Primitive Data Structures are the basic data structures that are used to store the data values that directly operate upon the machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, Floating point numbers, Character constants, String constants and Pointers come under this category.
    - **Integer:** It is a data type used to store numbers without fractional part.
    - **Float:** It is a data type used to store integer with fractional part.
    - **Character:** It is a data type used to store character values.
    - **Pointer:** Pointer is a variable which holds memory address of another variable.

2. Non – Primitive data structures:

- The data structures which are not primitive, are called as Non-primitive data structures
- These are derived from primitive data structures.
- Arrays, Lists and Files come under this category.

- **Array:** An array is a collection of same datatype of elements that share a common name.
- **List:** An ordered group of elements is called as lists.

- A Non- primitive data type is further divided into two types
  - **Linear data structure:**
    A data structure is said to be *linear* if its elements form a sequence or a linear list.
    Example: array, stacks, queues and linked lists
  - **Non- Linear data structures**:
    A data structure is said to be *non-linear* if its elements not form a sequence or linear list.
    Examples: Trees and Graphs

Q. Discuss about linear and Non- linear data structures

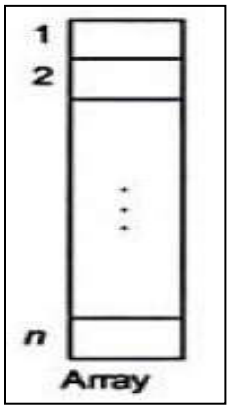
Data Structure:

Data is organized in many different ways. The logical (or) mathematical model of a particular organization of a data is called **data structure**.

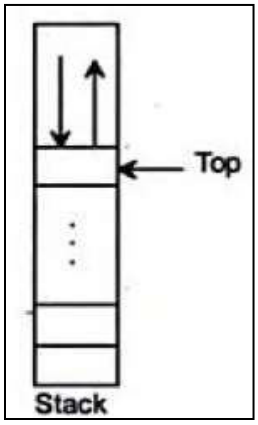
1. Linear data structures:

- A data structures is said to be linear, if its elements form a sequence or a linear list order.
- A Linear data structure contains only one starting point and only one ending point.
- There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation
- The possible operations on the linear data structure are Traversal, Insertion, Deletion, Sorting and Merging.
- **Examples**: Array, Linear Linked List, Stack, Queue.

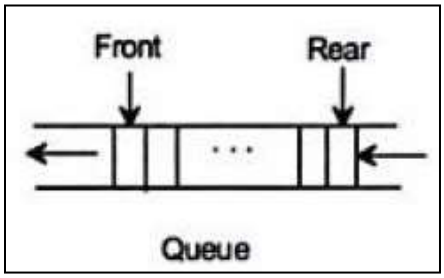
→**Array**: An array is a collection of same type of data items that share a common name.



→**Stack**: Stack is a data structure in which insertion and deletion operations are performed at one end only called "**top**". The Insertion operation is called as PUSH and deletion operation is called as POP. Stack is also called as Last in Fist Out (LIFO) data structure.



→**Queue**: Queue is a data structure in which the insertion is performed at one end (called **REAR**) and deletion is performed at another end (called **FRONT**). Queue is also called as First in First out (**FIFO**) data structure.

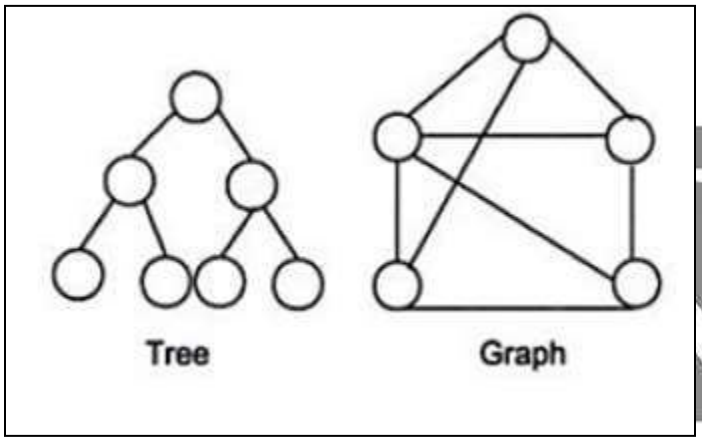


2. Non-linear data structures:

- A data structure is said to be *non-linear* if its elements does not arranged in a sequence order.
- A non-linear data structure contains several starting and ending points or no definite starting and ending points.
- **Examples**: tree and Graph.

→**Tree**: A tree is defined as finite set of data items (or nodes) in which data items are arranged in branches and sub branches according to requirement.

→**Graph**: Graph is a collection of nodes and edges between the nodes.



Diference between linear and Non linear data structures:

Linear Data structure	Non Linear data structure
Every item is related to the previous and time	Every item is attached with many other items.
Data is arrange is linear sequence	Data is not arranged in sequence
Data items can be traversed in a single run.	Data cannot be traversed in single run.
Eg. Array, stacks, linked list, queue	Eg.tree, graph
Implementation is easy	Implementation is difficult.

Q. DATA STRUCTURE AND STRUCTURED TYPE

The logical (or) mathematical model of a particular organization of a data is called **data structure**.

(or)

A data structure can be defined as the structural relationship present within the data set and it can be viewed as 2 tuple (N, R) where 'N' is the finite set of nodes representing the data structure and 'R' is the set of relationship among those nodes.

A **structured type** refers to a data structure which is made up of one or more elements known as **components.** These elements are simpler data structures that exist in the language. The components of structured data type are grouped together according to a set of rules, for example, the representation of **polynomials** requires at least two components:

- Coefficient
- Exponent

The two components together form a **composite type** structure to represent a polynomial.

Q. ATOMIC TYPE

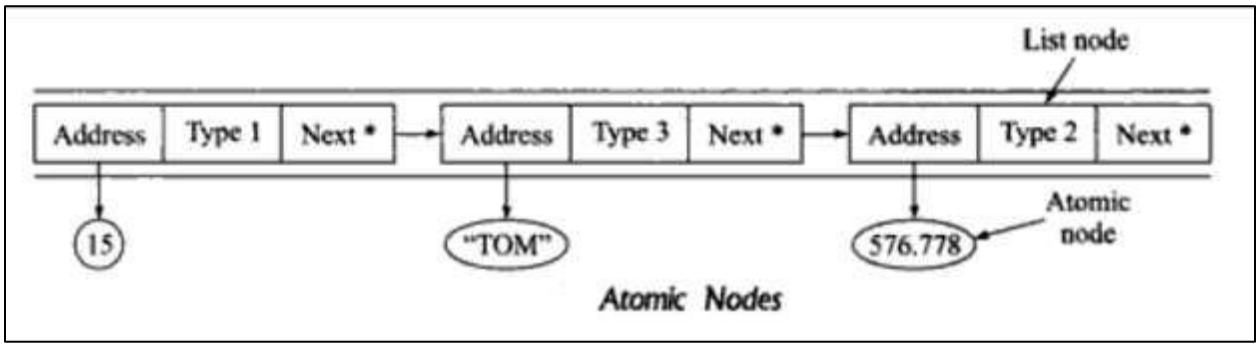
Generally, a data structure is represented by a memory block, which has two parts:

- Data storage
- Address storage

This facility in storing the data and relating it to some other data by means of storing pointers in the address part.

An atomic type data is a data structure that contains only the data items and not the pointers. Thus, for a list of data items, several atomic type nodes may exist each with a single data item corresponding to one of the legal data types.

The list is maintained using a list node which contains pointers to these atomic nodes and a type indicator indicating the type of atomic node to which it points. Whenever a test node is inserted in the list, its address is stored in the next free element of the list of pointers.



Atomic Nodes

The above figure shows a list of atomic nodes maintained using list of nodes.
In each node,

- **Type** represents the type of data stored in the atomic node to which the list node points.
- **1** stands for integer type, **2** for real number and 3 for character type or any different data types.

Q. DIFFERENCE BETWEEN ABSTRACT DATA TYPES, DATA TYPES AND DATA STRUCTURES

To avoid the confusion between abstract data types, data types, and data structures, it is relevant to understand the relationship between the three.

- An abstract data type is the specification of the data type which specifies the logical and mathematical model of the data type.
- A data type is the implementation of an abstract data type.
- Data structure refers to the collection of computer variables that are connected in some specific manner.

Q. REFINEMENT STAGES

The best approach to solve a complex problem is to divide it into smaller parts such that each part becomes an independent module which is easy to manage.

An example of this approach is the **System Development Life Cycle (SDLC)** methodology. This helps in understanding the problem, analyzing solutions, and handling the problems efficiently. The principle underlying writing large programs is the **top-down refinement**.

The application or the nature of problem determines the number of refinement stages required in the specification process.

Different problems have different number of refinement stages, but in general, there are **four** levels of refinement processes:

1. Conceptual (or) abstract level
2. Algorithmic (or) data structures
3. Programming (or) implementation
4. Applications

1. Conceptual level

At this level we decide how the data is related to each other, and what operations are needed.

2. Algorithmic or Data structure Level

At data structure level we decide about the operations on the data as needed by our problem.

3. Programming or Implementation Level

At implementation level, we decide the details of how the data structures will be represented in the computer memory.

4. Application Level

This level settles all details required for particular application such as names for variables or special requirements for the operations imposed by applications.

UNIT – II

Arrays: Introduction to Linear and Non- Linear Data Structures, One- Dimensional Arrays, Array Operations, Two- Dimensional arrays, Multidimensional Arrays, Pointers and Arrays, an Overview of Pointers

Linked Lists: Introduction to Lists and Linked Lists, Dynamic Memory Allocation, Basic Linked List Operations, Doubly Linked List, Circular Linked List, Atomic Linked List, Linked List in Arrays, Linked List versus Arrays

UNIT – II

Chapter – I: Arrays

Introduction to Linear and Non- Linear Data Structures, One- Dimensional Arrays, Array Operations, Two- Dimensional arrays, Multidimensional Arrays, Pointers and Arrays, an Overview of Pointers

---

Q. Discuss about linear and Non- linear data structures

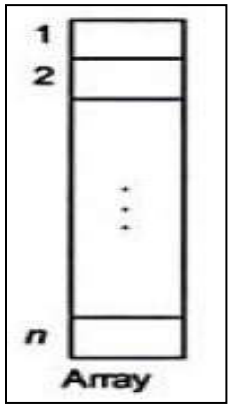
Data Structure:

Data is organized in many different ways. The logical (or) mathematical model of a particular organization of a data is called **data structure**.

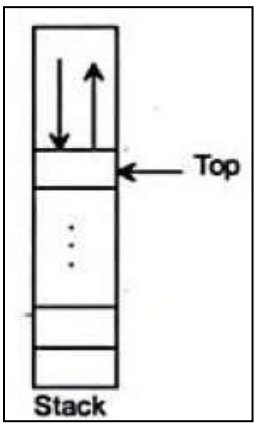
1. Linear data structures:

- A data structures is said to be linear, if its elements form a sequence or a linear list order.
- A Linear data structure contains only one starting point and only one ending point.
- There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation
- The possible operations on the linear data structure are Traversal, Insertion, Deletion, Sorting and Merging.
- **Examples**: Array, Linear Linked List, Stack, Queue.

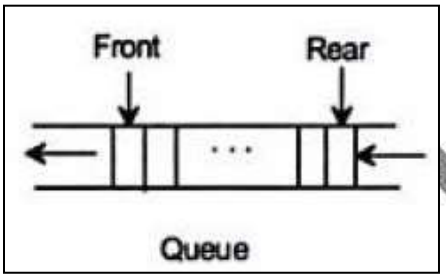
→**Array**: An array is a collection of same type of data items that share a common name.



→**Stack**: Stack is a data structure in which insertion and deletion operations are performed at one end only called "**top**". The Insertion operation is called as PUSH and deletion operation is called as POP. Stack is also called as Last in Fist Out (LIFO) data structure.



→**Queue**: Queue is a data structure in which the insertion is performed at one end (called **REAR**) and deletion is performed at another end (called **FRONT**). Queue is also called as First in First out (**FIFO**) data structure.

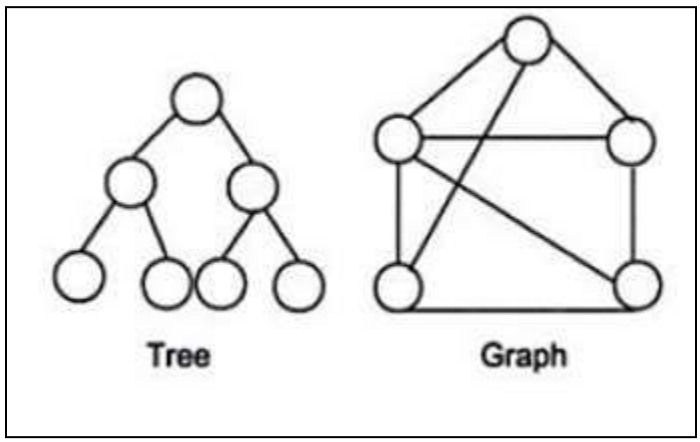


2. Non-linear data structures:

- A data structure is said to be *non-linear* if its elements does not arranged in a sequence order.
- A non-linear data structure contains several starting and ending points or no definite starting and ending points.
- **Examples**: tree and Graph.

→**Tree**: A tree is defined as finite set of data items (or nodes) in which data items are arranged in branches and sub branches according to requirement.

→**Graph**: Graph is a collection of nodes and edges between the nodes.



ARRAYS

Array Definition: An array is a finite, ordered and collection of homogeneous (i.e., same data type) data elements. An array is finite because it contains only a limited number of elements, ordered as all the elements are stored one by one in contiguous locations of the computer memory in a linear order.

Array Declaration:

Like other variables, the array variable must be defined before its use.

Syntax: datatype array_name [size];

Example: int age[20]; float sal[20]; char grade[10];

Initialization of array:

An array can be initialized at the time of declaration as follows

Syntax: datatype array_name [size]={list of values};

Example: int age[5]={8,10,5,15,20};

Age	[0]	[1]	[2]	[3]	[4]
Value →	8	10	5	15	20
Address →	100	102	104	106	108

Accessing array elements:

Once array is defined, it's elements can be accessed by using an index.

Syntax: array_name [index];

Example: scanf("%d", &arr[1];

printf("%d", arr[1]);

Q. BASIC TERMINOLOGY:

1. **Size:** The number of elements in an array is called the size or length of the array.
2. **Type:** The type of an array represents the kind of data type.
3. **Base:** The base of an array is the address of the memory location where the first element of the array is located.
4. **Element** – Each item stored in an array is called an element.
5. **Index:** All the elements in an array can be referenced by a subscript like A[i], this subscript is known as index. The index value is always an integer value.
6. **Range of Indices:** Indices of arrays elements may range from a lower bound to an upper bound called boundaries of an array.
7. **Word:** Word '*w*' denotes the size of an element i.e., the amount memory that is required to store an element of the array.

Q. One- Dimensional arrays and their address mapping and operations

One-Dimensional Array:

If only one subtype/index is required to refer or represent an element in an array, then the array is called as —One-Dimensional Array.

Memory Allocation:

All the indexed variables of an array are allocated contiguous memory locations in computer's memory. So memory representation of an array is simple.

Suppose, we have an array with n indexed variables, let the memory location of the first element is M. If each element requires one word, then the location of any element in the array can be calculated by using the following formula.

Address (A[i]) =M+(i x w) or Address (A[i]) = *M* + (i- 1)

Generally, an array is written as A *[L … U],* where *L* and *U* represents the lower and upper bounds for index. If it is stored starting from memory location M and for each element, then the address for A[i] is

Address (A[i]) = M + (i - L) x w

The above formula is known as "***indexing formula***", used to map the logical presentation of an array to physical presentation.

Physical Representation:

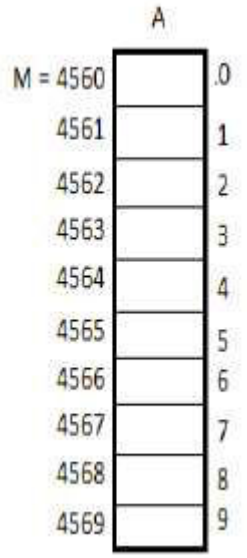
Eg: Let M=4560 and w=1, calculate the address of A[0], A[2], A[7]

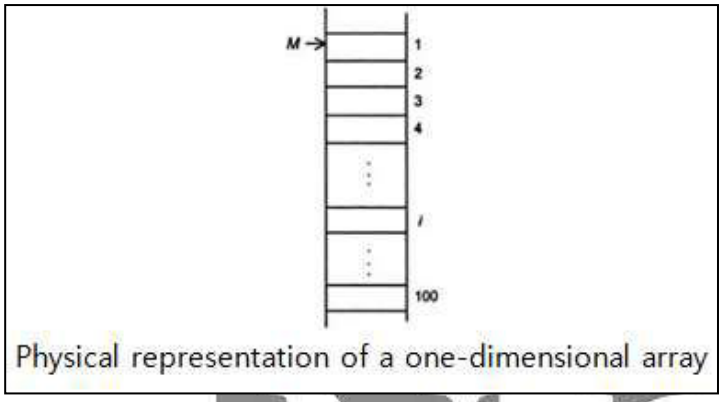
Formula: **Address (A[i]) =M+(i x w)**

Address (A[0]) = 4560+(0)x1 = 4560

Address (A[2]) = 4560 +(2)x1 = 4562

Address (A[7]) = 4560 +(7)x1 = 4567





Physical representation of a one-dimensional array

Q. ARRAY OPERATIONS:

We can perform various operations on an array are like Traversing, Sorting, Searching, Insertion, Deletion and Merging. Following operations can be performed on arrays:

1. Traversing: It is used to access each data item exactly once so that it can be processed.
(i.e., this operation is used to visit all the elements in an array.)

Algorithm:

Steps:

1. Initialize Counter
   Set counter: = LB
2. Repeat Steps 3 and 4 while counter <= UB
3. Visit element
   Apply PROCESS to arr[counter]
4. Increase counter
5. Set counter = counter + 1
6. Exit

2. Searching: It is used to find out the location of the data item if it exists in the given array. The search is said to successful if the item is found otherwise it is unsuccessful.

Algorithm for linear search:

1. Insert ITEM at the end of the arr
   Set arr{N+1]:=ITEM
2. Initialize counter
   Set LOC:=1
3. Search for ITEM
   Repeat while arr[LOC]≠ITEM
   Set LOC:=LOC+1
   End of loop
4. If LOC=N+1 then Set LOC:=0
5. Exit

3. Insertion: It is used to add a new data item in the given array.

Algorithm for Insertion

Let A be a linear array-the function used is INSERT(A,N,K,ITEM). N is the number of items, K is the positive integer such that K<=N. The following algorithm inserts an element ITEM into the K^{th} position of array A.

1. Initialize Counter
   Set J:-N

2. Repeat Steps 3 and 4 while J>=K
3. Move J^{th} element downward
   Set A[J+1] := A[J]
4. Decrease Counter
   Set J:=J-1
   End of step 2 loop
5. Insert element
   Set A[K]:= ITEM
6. Reset N
   Set N:=N+1
7. Exit

4. Deletion: It is used to delete an existing data item from an array.

Algorithm for Deletion

Let A be a linear array. The function used to delete from the array is DELETE(A,N,K,ITEM), Where N is the number of elements. K is the positive integer such that **K<=N**. The algorithm deletes K^{th} element from the array.

1. Set ITEM := A[K]
2. Repeat for J=K to N-1:
   [Move J+1 element upward]
   Set A[J] := A[J+1]
   End of loop
3. Reset the number N of elements in A
   Set N :=N-1
4. Exit

5. Sorting: It is used to arrange the data items either in ascending or descending order. There are many different sorting algorithms. A very simple sorting algorithm known as **bubble sort.**

Algorithm for Bubble Sort

Let **arr** be an array of N elements.

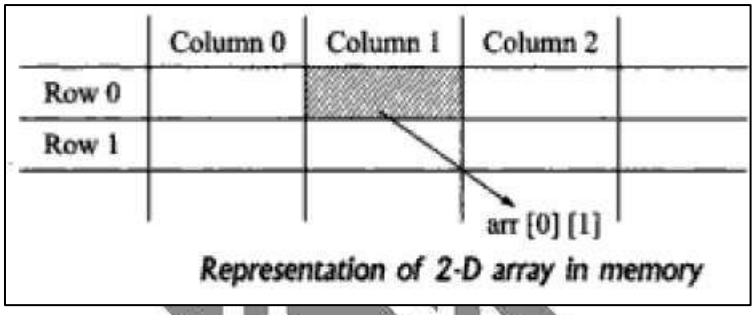
1. Repeat steps 2 and 3 for K=1 to N-1
2. Set PTR :=1 [Initializes pass pointer PTR]
3. Repeat while PTR <= N-K : [Execute Pass]
4. If arr[PTR] > arr[PTR+1], then :
5. Interchange arr[PTR] and arr[PTR+1]
6. [End of if structure]
7. Set PTR := PTR+1
8. [End of inner loop]

9. [End of step 1 outer loop]
10. Exit.

Q. Two-Dimensional arrays and their address mapping and operations

Two-Dimensional Array:

- If the elements are represented by using two indices/subscripts, then the array is called as Two-Dimensional Array.
- Two dimensional arrays are a collection of homogeneous elements placed in rows and columns.
- The 2-D array is also called a matrix. It is a collection of elements aij for all i and j's such that 0<=i<m and 0<j<=n. A matrix is said to be order of mxn.
- The various operations can be performed on a matrix are: addition, multiplication, transposition and finding determinant of the matrix.



Representation of 2-D array in memory

Eg: Consider an array of size mxn, here m is the no.of rows and n is the no.of columns as follows.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \end{pmatrix}_{mXn}$$

Memory Representation:

Like one-dimensional arrays, two-dimensional arrays are also stored in continuous memory locations. There are two ways of storing two-dimensional array elements in the memory.

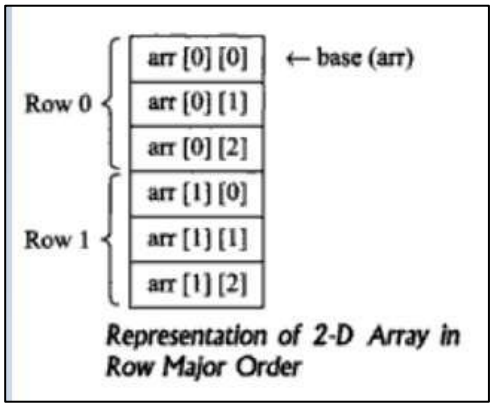
1. Row-Major Order
2. Column-Major Order

1. Row-Major Order:

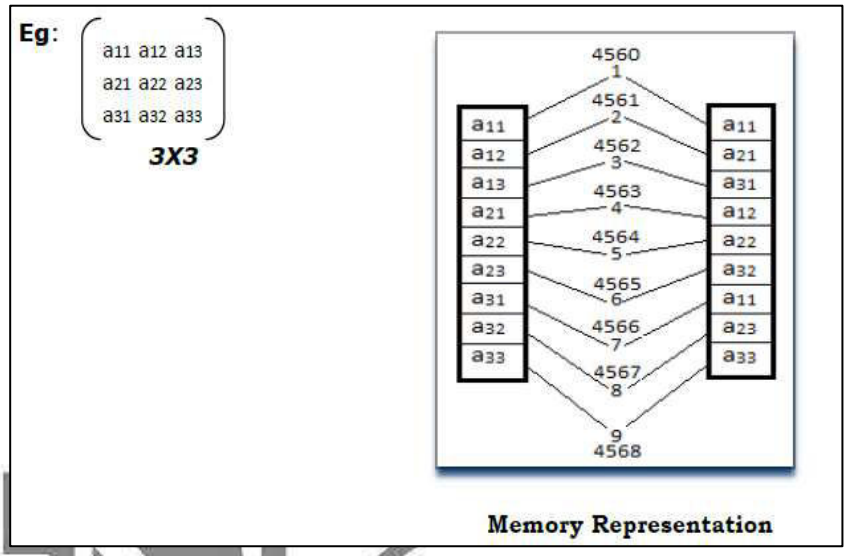
The elements of a two-dimensional array are stored in row-by-row basis that is all the elements in the first row, then in the second row and so on.

2. Column-Major Order:

The elements of a two-dimensional array are stored in column-by-column basis that is all the elements in the first column are stored in their order of rows, then in the second column and so on.



Representation of 2-D Array in Row Major Order



Referencing an element:

To refer an element in a two-dimensional structure, we need two index values - one for row another for column. The indexing formula of row-major order is different from column major order.

Let us assume that we have an array with mxn elements.

→Row-Major Order:

Address (aij) = (i-1) x n + j

Eg: Address (a32) = ((3-1)x3)+2 = 2x3+2 = 8

If the base address is M, then the above formula can be modified as follows.

Address (aij) = M+ (i-1) x n + j-1

Eg: Address (a32) = 4560+((3-1)x3)+2-1 = 4560+2x3+2-1 =4560+6+2-1 = 4567

→Column-Major Order:

Address (aij) = (j-1) x m + i

Eg: Address (a32) = ((2-1)x3)+2 = 2x3+2 = 6

If the base address is M, then the above formula can be modified as follows.
Address (aij) = M+ (j-1) x m +i-1

Eg: Address (a32) = 4560+((2-1)x3)+3-1 = 4560+3+3-1 =4560+3+2 = 4565

Q. MULTI DIMENSIONAL ARRAYS

Matrix (2-Dimensional array) and 3-Dimensional arrays are two examples of multi-Dimensional arrays.

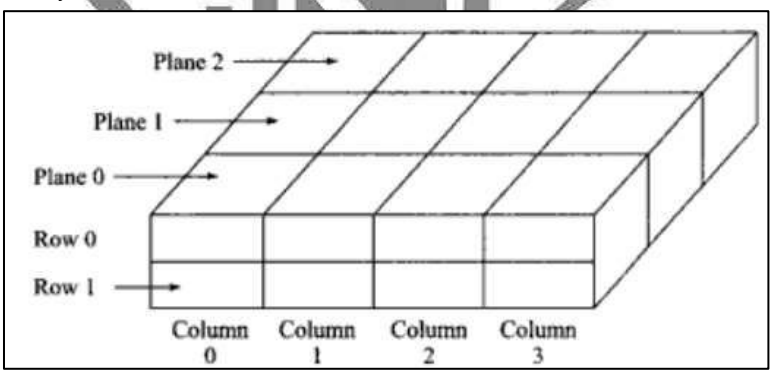
Multi-Dimensional array:

If elements are represented using more than one index/subscript, then the array is called as Multi-Dimensional Array.
For example, a three-dimensional array may be declared as

int arr[3][2][4]

The element of this array is referenced by three subscripts. The first specifies the plane number, the second specifies the row number and the third the column number.

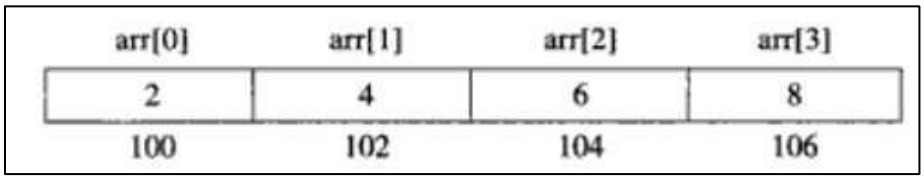
For example, **arr** contains 3*2*4=24 elements.
The arr[3][2][4] can be represented as shown below



Q. POINTERS AND ARRAYS

- Pointers are special variables which contain the address of another memory location. Pointers are useful in accessing any memory location directly.
- Pointers also allow arithmetic operations except subtraction, division and multiplication between two operands of pointer type.
- On array declaration, sufficient amount of storage is allocated by the compiler to store all the elements of the array.

For example, int arr[4]={2,4,6,8}
The element of an array is declared is as follows



The name **arr** acts as a constant pointer pointing to the first element,i.e. arr[0] of the array. Therefore, the value of **arr** is equal to the address where arr[0] is stored,i.e.

arr =&arr[0]=100

An integer pointer can be made to point to the array **arr** by the following assignment:

ip=arr
ip=&arr[0]

when the pointer variable is increased by 1 then it points to the next address which is equal to the base address +2 because pointer is of type int. Now the value of each element of **arr** can be accessed using ip++ as shown below:

ip =&arr[0] (=100)
ip +1 =&arr[1] (=102)
ip+2 =&arr[2] (=104)

so, the address of many element can be calculated by using its index and the size of the data type.
Syntax: Address of arr[index]=base address + (2*size of data type)
For example,
Address of arr[2] = base address + (2* size of int)
=100 + (2*2)
=104

AN OVERVIEW OF POINTERS

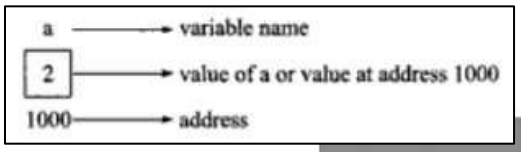
A pointer is a variable which contains the address of some memory location.

Points to be remembered while working with pointers -

- The **'&'** is the address operator, it represents the address of the variable.
- The **%u** is used for obtaining the address.

  For example,

```
main()
{
        int a=2;
        printf("value of a = %d\n" . a);
        printf("address of  a = %u\n" . &a);
}
```



The '*' operator is the value at address operator. It gives the value at specified address. Considering the above example we can use it as:

printf("value at address %u = %d\n, &a, (*&a));

So we can say that the address of variable 'a' preceded by *, gives the value at that address.

- When a pointer variable is declared, an '*' symbol should precede the variable name.

  For example, int *b; char *p; float *q;

- The address of a variable can be assigned to another variable.

  For example,

  int *b;

  b=&a; here, b is the variable which contains the address of variable a as its value.

- Another pointer variable can store the address of a pointer variable.

  For example,

  - Int a = 2:
  - Int *b:
  - Int **c:
  - b = &a:
  - c =&b:

In the example given above, c has been declared as a pointer to pointer variable which contains the address of pointer variable b.

- Various arithmetic operations can be performed on pointers-postfix, prefix, increment, decrement. In pointer arithmetic, all pointers increase or decrease by the length of the data type they point to.
- Pointers can also be used in handling functions. The arguments are passed to the functions in two ways:

  --->call by value

  ---->call by reference

UNIT - 2

Chapter – II: LINKED LISTS

Introduction to Lists and Linked Lists, Dynamic Memory Allocation, Basic Linked List Operations, Doubly Linked List, Circular Linked List, Atomic Linked List, Linked List in Arrays, Linked List versus Arrays

---

INTRODUCTION TO LISTS AND LINKED LISTS

'List' is a term used to refer to a linear collection of data items. A list is implemented either by using arrays or linked lists.

In arrays there is a linear relationship between the data elements which is evident or detailed from the physical relationship of data in the memory. The address of any element in the array can easily be computed but, it is very difficult to insert and delete any element in an array. Usually, a large block of memory is occupied by an array which may not be in use and it is difficult to increase the size of an array, if required.

Another way of storing a list is to have each element in a list contain a field called a **link or pointer,** which contains the address of the next element in the list. The successive elements in the list need not occupy adjacent space in memory. This type of data structure is called a **linked list.**

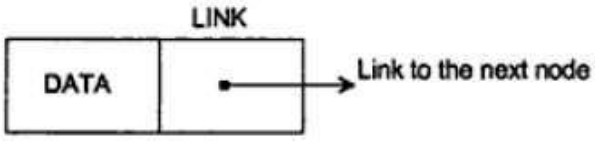
Q. DISCUSS ABOUT THE LIST (or) LINKED LIST

A linked list is an ordered collection of finite, homogeneous data elements called nodes where the linear order is maintained by links or pointers. Lists are used to create trees and graphs.

A Linked list refers to a linear collection of data items. A linked list is called as **Dynamic data structure** because its size is variable length.

An element in a linked list is called as node. A node contains two fields:

- DATA (to store the actual information)
- LINK (to points to the next node).



The structure defined for single linked list is implemented as follows:

```
struct node
{
        int data;
```

```
        struct node *next;
}
```

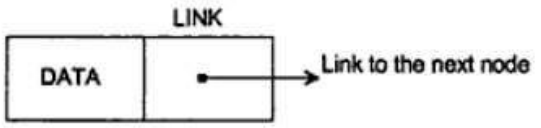
Q. WRITE ABOUT TYPES OF LINKED LISTS

1. LINKED LIST (OR) SINGLE LINKED LIST

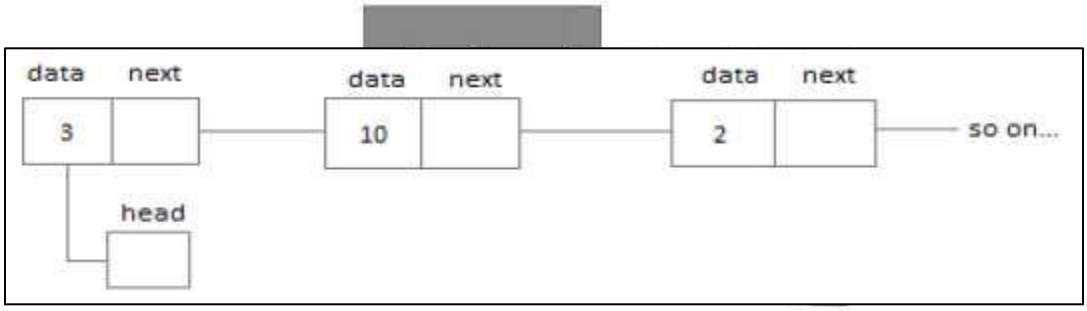
A single Linked list is a sequence of nodes in which every node has link to its next node in the sequence.

In any single linked list, the individual element is called as node. Every node contains two fields, data and link. The data field is used to store actual value of that node and next (or link) field is used to store the address of the next node in the sequence.

The graphical representation of a node in the single linked list is as follows



Example:



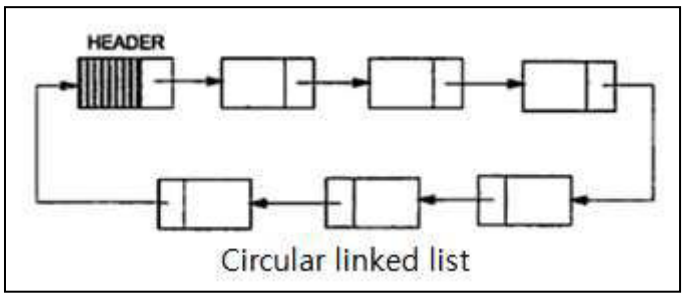
Note:

- ✓ In a single linked list, the address of the first node always stored in a reference node known as "HEAD" or "FRONT".
- ✓ Always LINK part of the last node must be NULL.

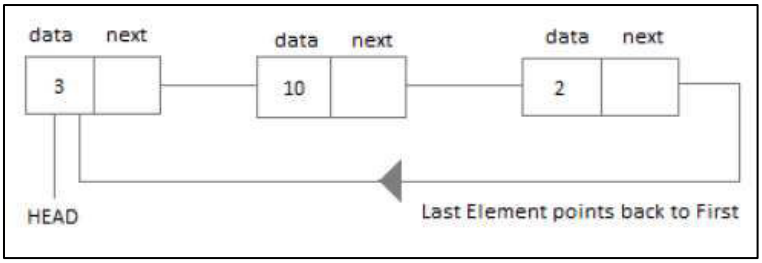
2. CIRCULAR LINKED LIST

Circular linked list is a sequence of elements in which every element has link to its next element and last element link to the first element in the sequence.

The graphical representation of a circular linked list is as follows

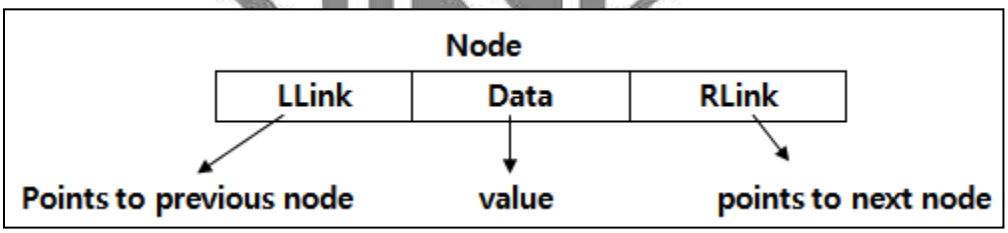


Example:

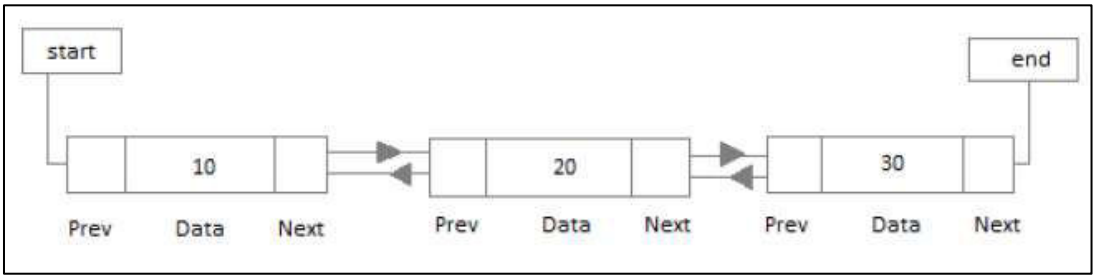


3. DOUBLE LINKED LIST

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence. Every node in a double linked list contains 3 fields: Data, LLINK and RLINK.



Example:



Note:

- ✓ In double linked list, the first node must be always pointed by head.
- ✓ Always the previous field of the first node must ne NULL.
- ✓ Always the next field of the last node must be NULL.

Q. WRITE ABOUT DYNAMIC MEMORY ALLOCATION

C language requires us to specify the number of elements in array at compile time. This may cause wastage of memory space. Such situations can be taken care of by using **dynamic data structures.**

Dynamic memory management techniques allow us to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space. The memory management functions that can be used for allocating and free memory during program execution are.

- malloc - Allocates requested size of bytes
- calloc - Allocates space for an array of elements
- free - Frees previously allocated space
- realloc - Modifies the size of previously allocated space.

Allocating a Block of Memory:

The malloc function can be used to allocate a block of memory. It reserves a specified size of memory and returns a pointer of type void and takes the following from:

ptr = (cast-type *) malloc (byte-size)

The malloc returns a pointer (of cast type) to an area of memory with size byte-size. For example,

y=(int *) malloc (100 * sizeof(int)):

In the statement give above, space *100 times the size of an int bytes is reserved and the pointer x is assigned the address of the first byte of the allocated memory .

Allocating Multiple Blocks of Memory:

The calloc function is used for seeking memory space for storing derived data types at run time. Multiple blocks of storage are allocated using the calloc function . The calloc takes the following form:

ptr= (cast –type *) calloc (n, elem- size)

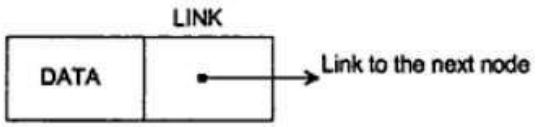
Contiguous space for n blocks, each of size **elem-size** bytes is allocated.

1. EXPLAIN ABOUT LINKED LIST (OR) SINGLE LINKED LIST

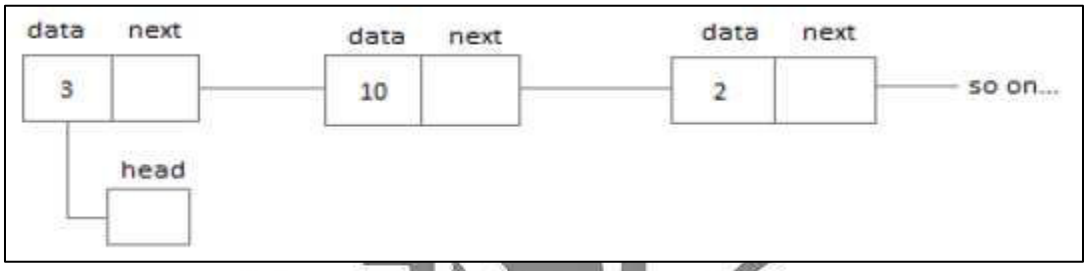
A single Linked list is a sequence of nodes in which every node has link to its next node in the sequence.

In any single linked list, the individual element is called as node. Every node contains two fields, data and link. The data field is used to store actual value of that node and next (or link) field is used to store the address of the next node in the sequence.

The graphical representation of a node in the single linked list is as follows



Example:



Note:

- ✓ In a single linked list, the address of the first node always stored in a reference node known as "HEAD" or "FRONT".
- ✓ Always LINK part of the last node must be NULL.

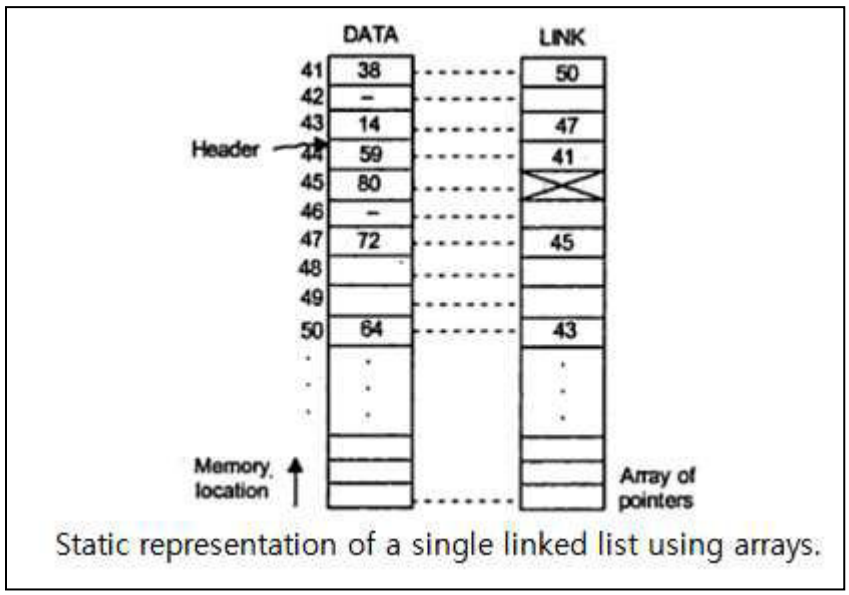
REPRESENTATION OF A LINKED LIST IN MEMORY:

There are two ways to represent a linked list in memory:

1. Static representation using array
2. Dynamic representation using free pool of storage

1) Static representation

Static representation of a single linked list maintains two arrays: one array for data and other for links. The Static representation for the linked list is shown below.



Static representation of a single linked list using arrays.

2) Dynamic representation

- ✓ The efficient way of representing a linked list is using free pool of storage. In this method, there is a **memory bank** (which is a collection of free memory spaces), and a **memory manager** (a program).
- ✓ During the creation of linked list, whenever a node is required the request is placed in **the memory manager**. Now, it will search the **memory bank** for the requested block and if found grants a desired block.
- ✓ There is also another program called **garbage collector**, it returns the unused node to the memory bank, whenever a node is not used.
- ✓ Memory bank is also a list of memory space to a programmer. Such a memory management is known as dynamic memory management.

Q. WRITE ABOUT OPERATIONS ON A SINGLE LINKED LIST

Creating a Linked List:

Linked list is used to avoid any reference to specific number of items in the list so that insertion and deletion is easily done. This can be achieved by using unnamed locations to store nodes. These can be created by using pointers and dynamic memory allocation functions such as **malloc.** The **head** pointer is used to create and access unnamed nodes.

```
struct linked_list
{
        int no :
        struct linked_list *next :
}:
```

typedef struct linked_list node :
node *head :
head = (node*) malloc (size of (node)) :

The above statement obtains memory to store a node and assigns its address to **head** which is a pointer variable.

To store values in the member fields use the following statements:

Head→ no = 10 ;
Head→ next = NULL;

The second node can be added as follows:

Head→ next = (node*) malloc (size_of (node));
Head→ next→number = 20;
Head→next→next =NULL;

Q. Write about OPERATIONS ON A SINGLE LINKED LIST

We can perform the following operations in Single linked list

1. Insertion of a node
2. Deletion of a node
3. Traversing
4. Merging two linked lists
5. Searching for an element

1. Traversing a single linked list

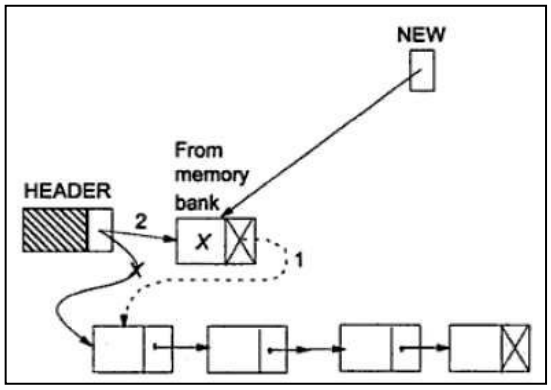
Traversing means to visit every node in the list starting from the first node to the last node.

```
ptr = HEADER→LINK
While (ptr ≠ NULL) do
    PROCESS(ptr)
    ptr = ptr→LINK
End While
Stop
```

2. Insertion of a node into a single linked list

This operation is used to insert an element in a single linked list. There are various positions to insert nodes:

a) Insert at front (as a first element)
b) Insert at end (as a last element)
c) Insert at any position.



a) Insertion of a node at the front

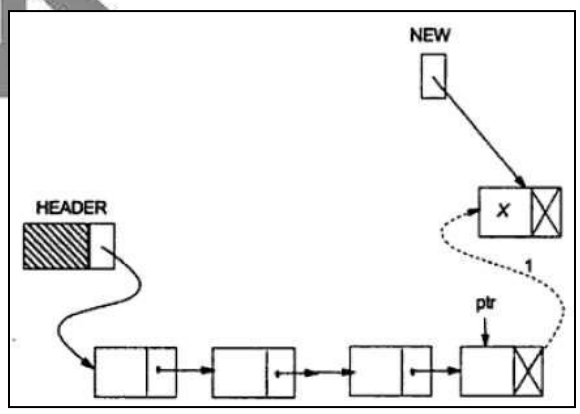
Steps:

```
new = GETNODE (NODE)
If (new = NULL) then
  Print "Memory underflow: No insertion"
  Exit
Else
  new→LINK = HEADER→LINK
  new→DATA = X
  HEADER→.LINK = new
End If
Stop
```

b) Insertion of a node at the end.

Steps:

```
new = GETNODE(NODE)
If (new = NULL) then
  Print "Memory is insufficient: Insertion is not
possible"
  Exit
Else
  ptr = HEADER
   While (ptr→LINK≠ NULL) do
     ptr = ptr→LINK
   EndWhile
ptr→LINK = new
new→DATA = X
End If
Stop
```



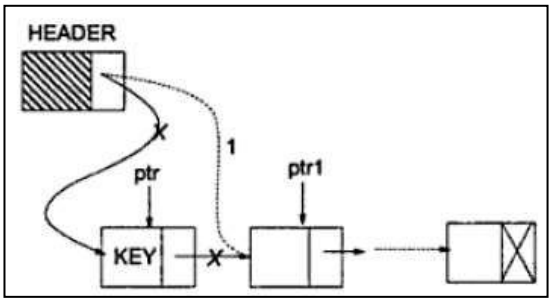
3. Deletion of a node from a single linked list

This operation is used to delete a node from the single linked list. There are various positions to delete nodes:

a) Deletion at the front
b) Deletion at the end
c) Deletion at any position

a) Deletion of a node at the front

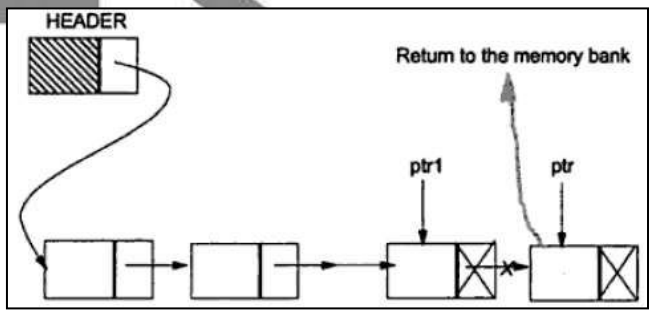
Steps:




```
ptr = HEADER→LINK
If (ptr = NULL)
   Print "The list is empty: No deletion"
   Exit
Else
   Ptr1 = ptr→LINK
   HEADER→LINK = ptr1
   RETURNNODE(ptr)
End If
Stop
```

b) Deletion of a node at the end

Steps:




```
ptr = HEADER
If (ptr→LINK = NULL) then
   Print "The list is empty: No deletion
possible"
   Exit
Else
   While (ptr→LINK ≠ NULL) do
      Ptr1 = ptr
      ptr = ptr→LINK
   End While
ptrl→LINK = NULL
RETURN NODE(ptr)
End If
Stop
```

4. Merging two Single linked Lists

Merging means combining the two single linked lists in to one single linked list

Steps:

```
ptr = HEADER1
While (ptr→LINK ≠ NULL) do
  ptr = ptr→LINK
End While
ptr→LINK = HEADER2→LINK
RETURN NODE(HEADER2)
HEADER = HEADER1
Stop
```

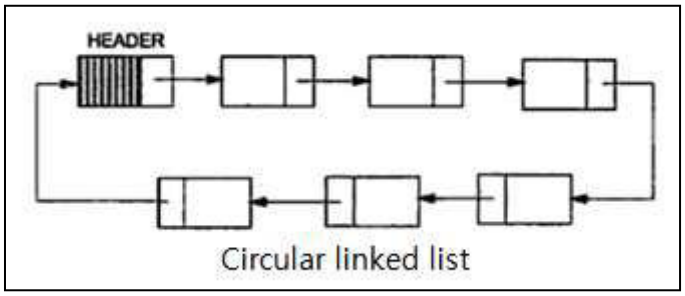
2. EXPLAIN ABOUT CIRCULAR LINKED LIST

Circular linked list is a sequence of elements in which every element has link to its next element and last element link to the first element rather than the NULL pointer in the sequence. A circular linked list can be used to represent a stack and a queue.

Therefore, the structure defined for circular linked list is same as for the linear linked list as given below:

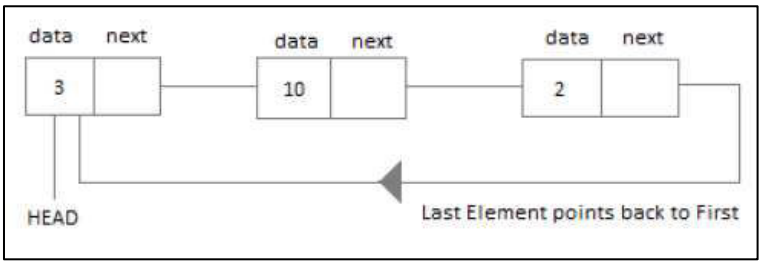
```
struct node
{
        int data :
        struct node * next :
}
```

The graphical representation of a circular linked list is as follows



Circular linked list

Example:



Creating a Circular Linked List using Queue:

Function Ciradd( ):

This function accepts **three parameters**. The **first** parameter receives the address of the pointer to the first node (i.e., address of first node - front), the **second** parameter receives the address of the pointer to the last node (i.e., address of last node - rear). The **third** parameter stores the data items we need to add in the list.

The memory is allocated for the new node whose address is stored in pointer **q**. Then, the data which is preset in item is stored in the data part of the new node. If the new node is added the empty list then the address of the new node is stored in front *f =q;

Then *r = q; is executed, which stores the address of the new node into rear. Thus, both front and rear point to the same node.

The statement (*r)→ link = *f; is executed to store the address of the front node in the next part of the rear node (As the link part of the last node should contain the address of the first node).

Next, if the new node is not added in the first node then the address present in the next part of the last node is overwritten with the address of new node, (*r)→ next = q ;

Then the address of the new node is stored in the pointer rear *r=q; and the address of the first node is stored in the next part of the new node. This is done by:

(*r)→ link = *f;

Deleting from Circular Linked List:

Function Delcirq( ):

This function receives **two parameters**. The **first** parameter is the pointer to the front and the **second** is the pointer to the rear. The condition is checked for the empty list.

If the list is not empty, then it is checked whether the front and rear point to the same node or not. If they point to the same node, then the memory occupied by the node is released and front and rear are both assigned a NULL value.

If the front and rear are pointing to different nodes then the address of the first node is stored in a →pointer q. then the front pointer is made to point to the next node in the list, i.e. the node pointed by (*f) → link;. Now the address of front is stored in the next part of the last node. Then the memory occupied by the node being deleted is released.

Displaying the Circular Linked List:

Function Cirq-disp( ):

This function receives the pointer to the first node in the list as a parameter. Then **q** is also made to point to the first node in the list. This is done because the entire list is traversed using q. another pointer **p** is set to NULL initiality. The circular list is traversed through a loop till the time we reach the first node again. We should reach the first node when **q** equals **p.**

Q. Write about OPERATIONS ON CIRCULAR LINKED LIST

We can perform the following operations in a circular linked list

1. Searching of an element:

This operation is used to search particular element in circular linked list.

Steps:

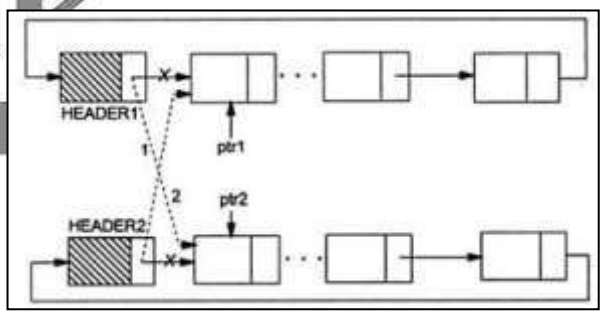
ptr = HEADER→LINK
While (ptr→DATA≠ KEY) and (ptr ≠ HEADER) do
ptr = ptr→LINK
End While
If (ptr→DATA = KEY)
Return (ptr)
Else
Print "Entire list is searched: KEY node is not found"
End If
Stop

2. Merging:

This operation is used to combine two circular linked lists into one circular linked list.

Steps:

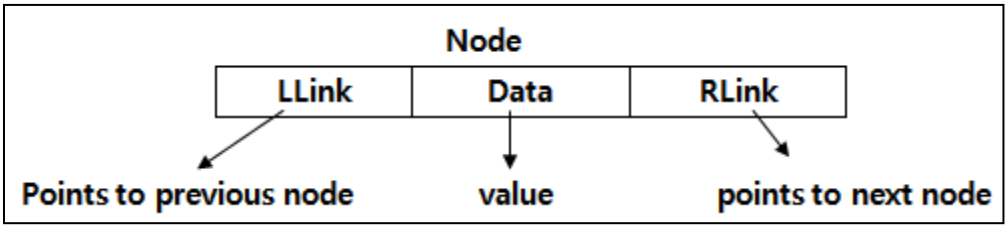
ptrl = HEADER 1→LINK
ptr2 = HEADER2→LINK
HEADER1→LINK = ptr2
HEADER2→LINK = ptr1



3. EXPLAIN ABOUT DOUBLE LINKED LIST

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next filed and can traverse backward by using previous field. Every node in a double linked list contains 3 fields: Data, LLINK and RLINK.



Note:

- ✓ In double linked list, the first node must be always pointed by head.
- ✓ Always the previous field of the first node must ne NULL.
- ✓ Always the next field of the last node must be NULL.

The structure defined for doubly linked list is:

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

Q. Write about OPERATIONS ON DOUBLE LINKED LIST

We can perform the following operations in a double linked list

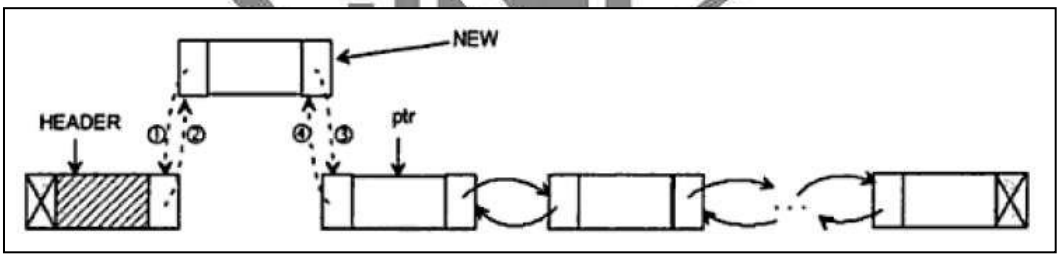
1. Inserting:

This operation is used to insert an element into a Double linked List

a) Inserting a node at the Front

Steps:

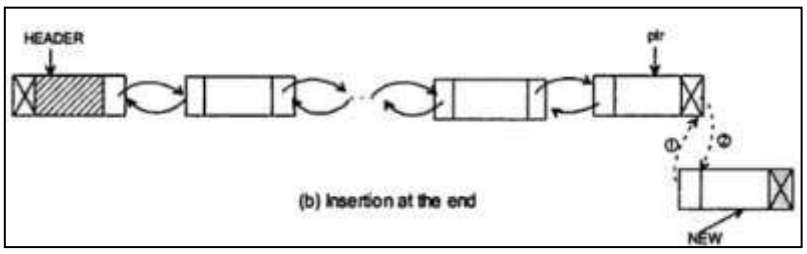
```
ptr = HEADER→RLINK
new = GETNODE(NODE)
if (new ≠ NULL) then
  new→LLINK = HEADER
  HEADER→RLINK = new
  new→RLINK = ptr
  ptr→LLINK = new
  new→DATA = X
Else
  Print "Unable to allocate memory: Insertion is not possible"
End If
Stop
```



b) Insertion of a node at the end

Steps:

```
ptr = HEADER
While (ptr→RLINK≠ NULL) do
  ptr = ptr→RLINK
End While
new = GETNODE(NODE)
If (new ≠  NULL) then
  New→LLINK = ptr
  Ptr→RLINK = new
  New→RLINK = NULL
  New→DATA = X
Else
```



Print "Unable to allocate memory
End If
Stop

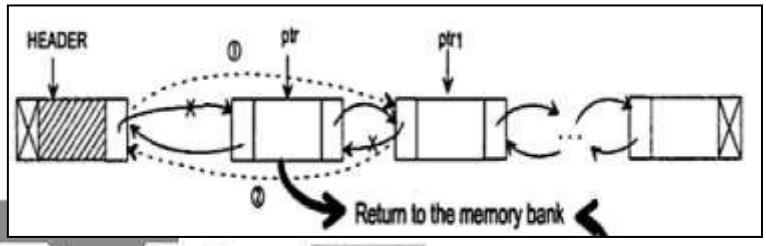
2. Deletion:

This operation is used to delete a node from double linked.

a) Deletion of a node at the front:

Steps:

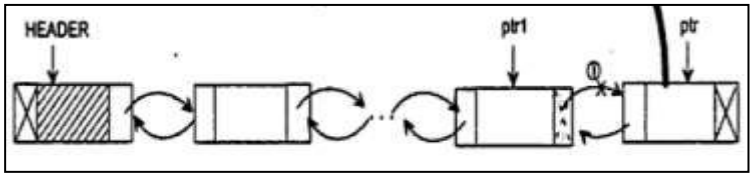
ptr = HEADER→RLINK
If (ptr = NULL) then
Print "List is empty: No deletion is made"
Exit
Else
Ptr1 = ptr→RLINK



HEADER→RLINK = ptr1
If (ptrl ≠ NULL)
Ptr1→LLINK = HEADER
End If
RETURNNODE (ptr)
End If
Stop

b) Deletion of a node at the end:

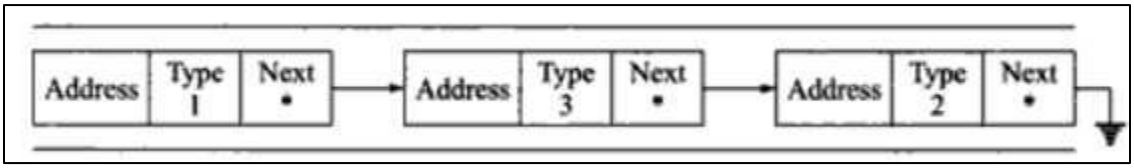
Steps:



ptr = HEADER
While (ptr→RLINK ≠ NULL) do
ptr = ptr→RLINK
End While
If (ptr = HEADER) then
Print "List is empty: No deletion is made"
Exit
Else
Ptr1 = ptr→LLINK
Ptr1→RLINK = NULL
RETURNNODE(ptr)
End If
Stop

Q. Write about ATOMIC NODE LINKED LIST

An atomic data type contains only the data items and not the pointers. Thus, for a list of data items several atomic type nodes may exist, each with a single data item corresponding to one of the legal data types. Their list is maintained using a list node which contains pointers to these atomic nodes and a type indicator indicating the type of atomic node to which it points. Whenever a list node is inserted in a list, its address is stored in the next free element of the list of pointers.



Q. Write about LINKED LIST IN ARRAYS

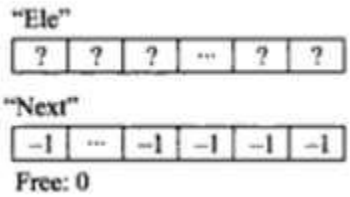
Linked lists are implemented without using pointers.

For example, consider an ordered list of integers given by L = (10,-5, 0, 99). This list can be stored in an array, say "Ele". The concept of link can be implemented by using another array, "Next". The i^{th} element of "L" is stored in the i^{th} index of an array "Ele".

The node in a linked list contains two parts -"data" and "next". These two parts of node are split and stored in two arrays "Ele" and "Next". If Ele[i] represents the data part of the node then Next[i] denotes the next part of that node.

In this case the actual physical address is not denoted by next. Rather Next[i] is an integer and if Next[i] is j then the node next to the one represented by i^{th} index of "Ele" and the "Next" is the node represented by j^{th} index of "Ele" and "Next".

If Next[i] = -1 then, the node under consideration is assumed to be the last node, initially these arrays are unused and so a variable "free" is set to 0. The "free" function keeps track of the available parts in the arrays.



When the first element of our list L is added to the array, Ele[0] contains 10, Next[0] remains -1, "free" becomes 1 and new variable "start" is required to remember which index in these arrays represent first node in the list. Figure 4.15 illustrates the addition of first node in the list.

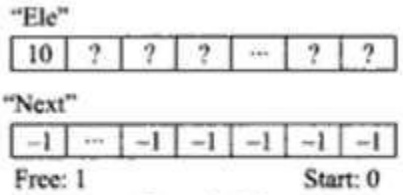


Fig. 4.15

When an attempt is made to add the second element of the list, the existing element is traversed from "start". As in our case, the value is 0. Then the Next[0] is -1, this is the end of the list. So the new node is added after this node. The index where the new node is to be stored in "Ele" and "Next" is found by inspecting "free". Next [0] is updated by current value of "free" and "free" is also incremented.

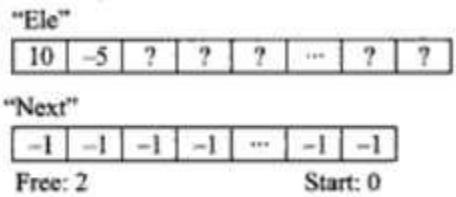


Fig. 4.16

Figure 4.16 illustrates adding second element in the lis. Note that Ele[1] is set to -5, Next[0] is set to 1 and Next[1] is set to -1. Another addition will need to traverse the list from "start", start = 0. As Next[0] = 1 the next node can be found in index 1 of the array "Ele" and "Next". The Next[1] is found the value is -1, i.e.the node is the last node.

As before, Next[1] is set to the current value of "free". Ele[free] is set to -1, and "free" is incremented. Figure 4.17 and 4.18 illustrate the addition of third and fourth nodes to the list.

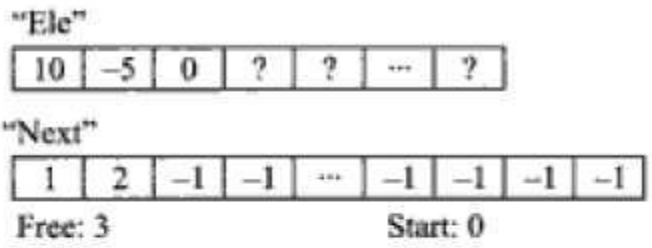


Fig. 4.17 *Adding Third Node to the List*

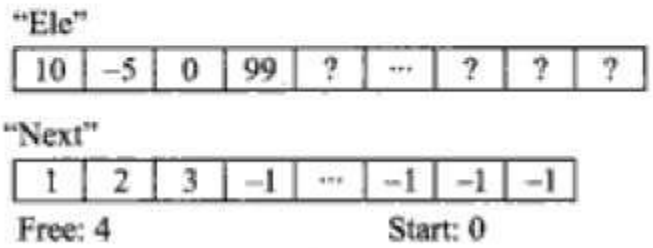


Fig. 4.18 *Adding Fourth Node to the List*

Now if we want to delete the second node in the linked list, the next field of the first node is to be changed, in respect of the next field of second node. The array will be shown as given in fig. 4.19.

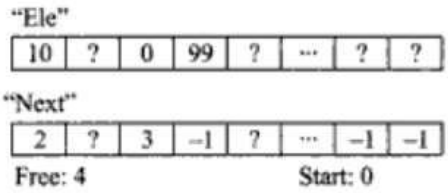


Fig. 4.19 *Deletion of the Second Node from the List*

Therefore, it can be seen that no shifting of elements is done. Only the "next" array is updated. Say, if we want to again add an element with value 56 to the list after deletion (fig. 4.20)

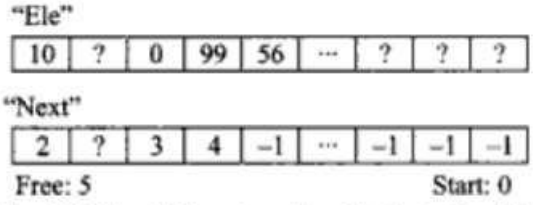


Fig. 4.20 *Adding Another Node to the List*

It can be observed that the unused node 2 (i.e. the element with index 1 in the arrays "Ele" and "Next") could not be reused. The variable "free" keeps on increasing without paying any attention to the unused nodes.

Figure 4.21 illustrates the deletion of the first node in the list. The value of the "Next" array with index "start" has to be made the new value of "start". In this example start =0 and Next[0] =2 before deletion, so the new value of "start" become 2 and Next [0] becomes undefined after deletion.

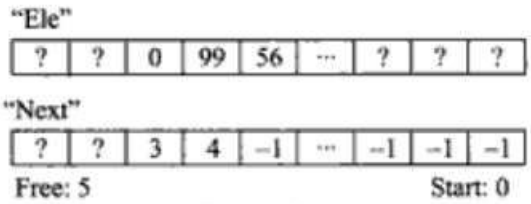


Fig. 4.21

LINKED LISTS VERSUS ARRAYS

We can store similar data in memory with the use of either an array or a linked list.

Arrays are very simple data structures that are easy to understand but they have the following disadvantages:

- The size of arrays cannot be increased or decreased during execution. They have a fixed dimension.
  For example, If we have allocated space for 10 elements and try to add more than 10 elements we are not able to.
- The elements in an array are stored in contiguous memory locations, but in many cases it may be possible that the contiguous memory space is not available.
- The operations like insertion of a new element in an array or deletion of an existing element after the specified position may be tedious as insertion or deletion requires each element after the specified position to be shifted one position to the right (insertion) or one position to the left (deletion).

Linked list can be used to overcome all these disadvantages.

- A linked list can grow or shrink during the execution of program.
- There is no problem of shortage of memory as the nodes are stored in different memory locations.
- In various operations like insertion and deletion no shifting of nodes is required

UNIT – III
CHAPTER – 1: STACKS

Introduction to Stacks, Stack as an Abstract Data Type, Representation of Stacks through Arrays, Representation of Stacks through Linked Lists, Applications of Stacks, Stacks and Recursion

---

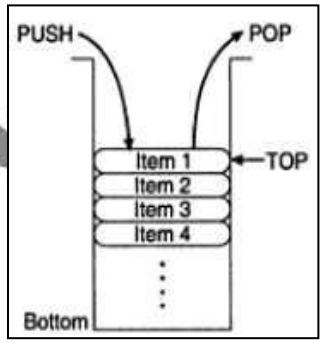
Q. STACK

Stack is a non-linear data structure where insertion and deletion of elements at one end of the stack called as "Top". In this, the elements are inserted or deleted at one end. Therefore stack is also called as LIFO (Last in First Out) lists.

(Or)

A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only, called "top".

The insertion operation is known as **PUSH** and deletion operation is known as **POP**, and the position of the stack where these operations are performed is known as "**Top of the stack**". An element in a stack is termed as **Item** and the maximum no. of elements in a stack is termed as **Size**. For example, a stack of plates, a stack of coins etc



A stack is defined as follows

```
struct stack
{
        int a[arr];
       int top;
}
```

Q. STACK ADT

Stacks are defined as Abstract Data Types (ADT). We can implement the stack ADT either with **array or linked list.**

- **Initialize** the stack to be empty
- Determine whether stack is **empty** or **not**
- Determine if the stack is **full** or **not**
- If stack is not full, then add or insert a new item at one end of the stack called **top**. This operation is known as **PUSH**.
- If the stack is not empty then delete the item at its top. This operation is known as **POP**.

- **Stack overflow** happens when we try to push one more item onto our stack than it can actually hold.
- **Stack underflow** happens when we try to pop (remove) an item from the stack, when nothing is actually there to remove.
- To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –
    - peek() – get the top data element of the stack, without removing it.
    - isFull() – check if stack is full.
    - isEmpty() – check if stack is empty.

Q. What is stack? Discuss about array and linked list representation of a stack

(Or)

Representation of Stack

Stack:

A Stack is a linear list in which insertions and deletions take place at the same end called **"top"**. The other end of the stack is called the **Bottom**.

Representation of Stack:

A stack may be represented in the memory in various ways. Mainly there are two ways. They are:

1. Using one dimensional arrays (Static Implementation)
2. Using linked lists (Dynamic Implementation)

1. Stack Representation using Arrays:

A Stack data structure can be represented using arrays. We perform stack operations *Push* and *POP* by maintaining the top value to point to the topmost element position of the stack.

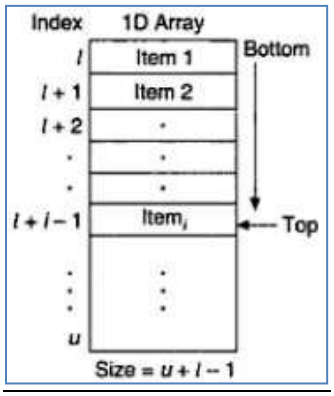
First we have to allocate a memory block of sufficient SIZE to accommodate the full capacity of the stack. Then, starting from the first location items of the stack can be stored in sequential order.

Here, $item_i$ denotes the i^{th} item in the stack. l and u denotes the index ranges with the values 1 and **SIZE.TOP** is a pointer point to position of the array.

With this representation we stated that:

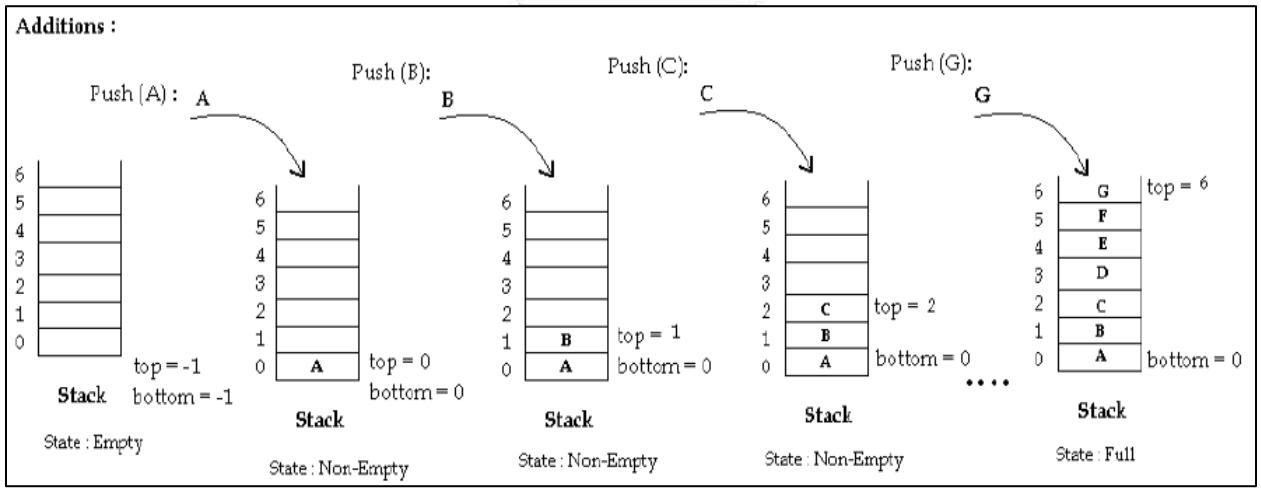
Empty: top<1

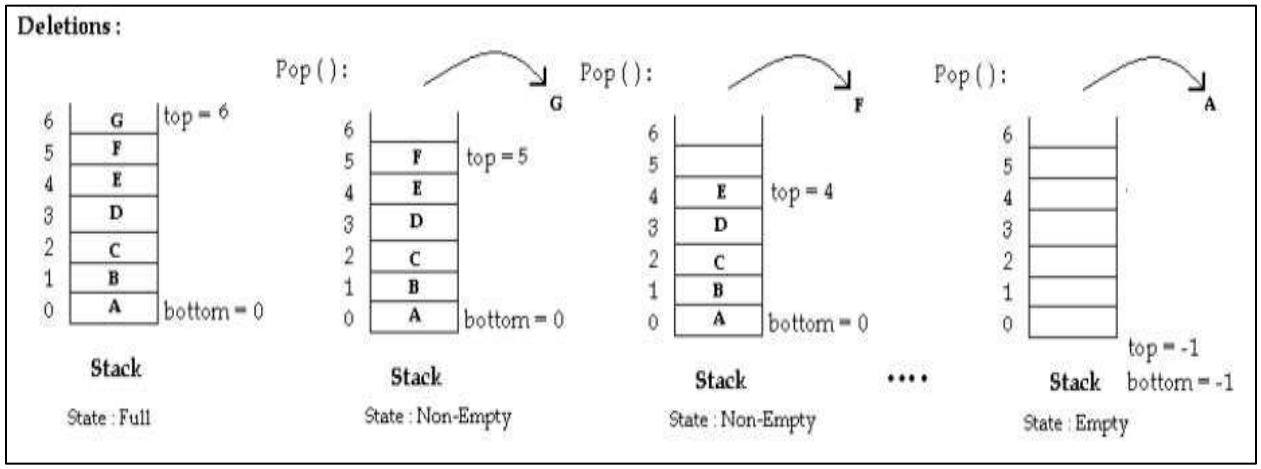
Full: TOP>=u



Operations on stack (Using Array)

- Create stack - To create an empty stack
- push - To add or insert a new element
- pop - to remove the top element from stack
- isFull - returns true if the stack is full
- isEmpty - returns true if the stack is empty





2. Linked List Representation

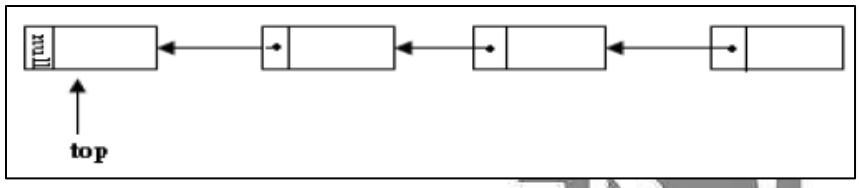
The stack also is implemented using linked lists. The array representation of stack suffers from the drawbacks of the array's size that cannot be increased or decreased once it is created. Therefore either most of the space is wasted if not used or storage of space is needed.

The stack as linked list is represented as a single list. Each node contains data and a pointer to the next node.

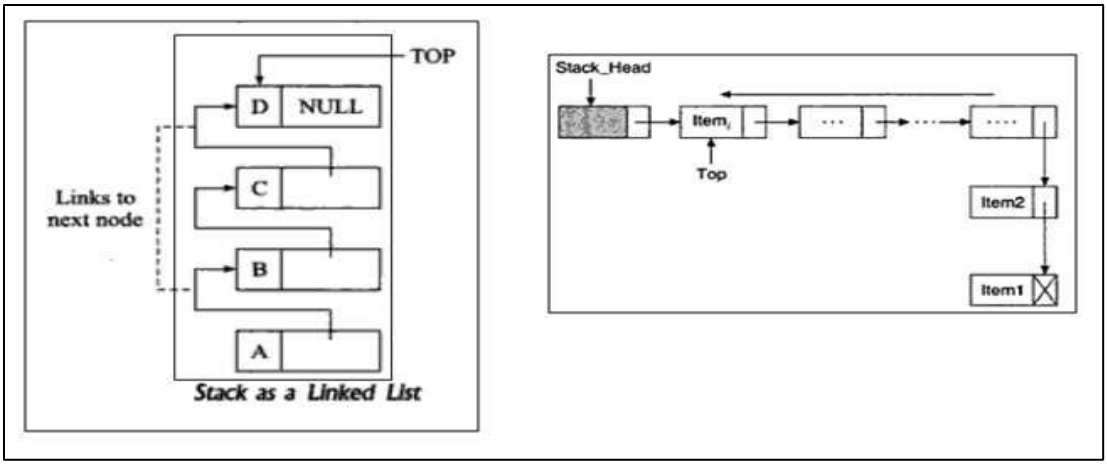
The structure is defined as follows:

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

Example:

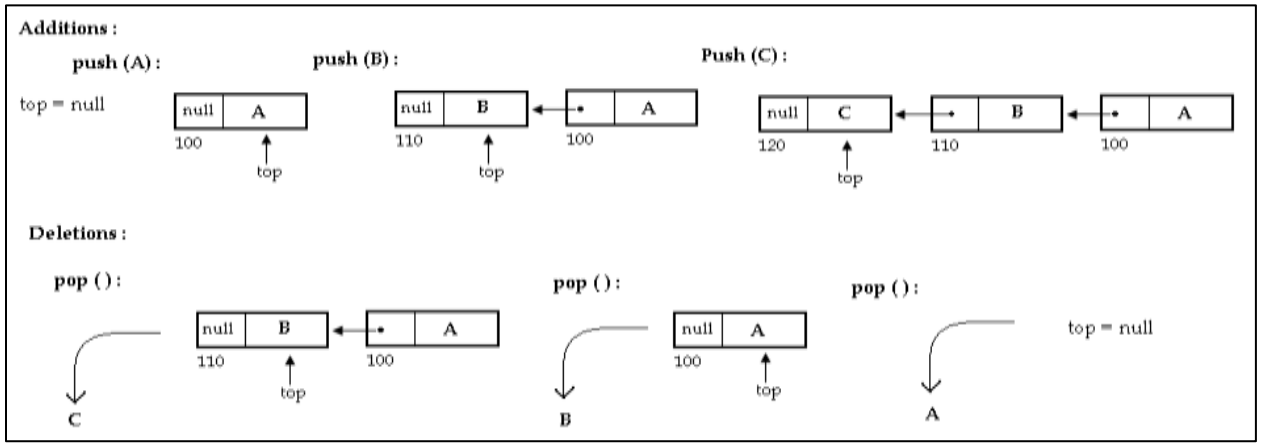


In the linked list representation, first node on the list is the current item that is the item at the top of the stack and the last node is the node containing bottom most item. So PUSH operation will perform at the front and POP operation will perform at front. SIZE of the stack is not important here because dynamic representation.



Operations on stack (Using Linked List)

- Create stack: To create an empty stack
- Push: To add or insert a new element
- Pop: to remove the top element



Q. OPERATIONS ON STACK USING ARRAY

The basic operations on stack are

a) Push
b) Pop
c) Status

a) PUSH

This operation is used to add an element in to the stack. Whenever we add an element into the stack the pointer TOP incremented as TOP+1 or TOP++.

Algorithm:

1. If(TOP>=SIZE) then
2. Print "Stack is full/overflow"
3. Else
4. TOP=TOP+1
5. A[TOP]=item
6. End if
7. Stop

b) POP

This operation is used to delete an element in the stack. Whenever we delete an element in to the stack, the pointer TOP decremented as TOP-1 or TOP--.

Algorithm:

1. If TOP<1
2. Print "Stack is empty/underflow"
3. Else
4. Item=A[TOP]
5. TOP=TOP-1
6. stop

c) STATUS

This operation is used to know the present state of the stack.

Algorithm:

1. If TOP<1
2. Print "Stack is empty/underflow"
3. Else
4. If TOP>=SIZE
5. Print "Stack is Full"
6. Else
7. Print "The element at TOP is", A[TOP]
8. Free=(SIZE-TOP)/SIZE*100
9. Print "percentage of free stack is", free
10. End if
11. End if
12. Stop

Q. OPERATIONS ON STACK USING LINKED LIST

The basic operations on stack are

a) Push
b) Pop
c) Status

a) PUSH

This operation is used to add an element in to the stack.

Algorithm:

1. new=getnode(node)
2. new→data=item
3. new→link=TOP
4. TOP=new
5. STACK_HEAD→LINK=TOP
6. Stop

b) POP

This operation is used to delete an element in the stack. Whenever we delete an element in to the stack, the pointer TOP decremented as TOP-1 or TOP--.

Algorithm:

1. If TOP=NULL
2. Print "Stack is empty/underflow"
3. exit
4. Else
5. ptr=TOP→LINK
6. item=TOP→DATA
7. STACK_HEAD→LINK=ptr
8. TOP=ptr
9. stop

c) STATUS

This operation is used to know the present state of the stack.

Algorithm:

1. ptr=stack_head→link
2. if(ptr=null)
3. print "stack is empty/underflow"
4. else
5. nodecount=0
6. while(ptr!=null) do
7. nodecount=nodecount+1
8. ptr=ptr→link
9. end while
10. print "item found at the front is " top→data,"stack contains",nodecount,"no of items"
11. end if
12. stop

Q. APPLICATIONS OF STACK

Various applications of stacks are

1. Evaluation of Arithmetic Expressions
2. *Conversion of an Infix expression to postfix expression*
3. *Evaluation of a postfix expression*
4. Towers of Hanoi

1. Evaluation of Arithmetic Expressions

One of the important applications of stack is Evaluation of Arithmetic expression. Stack can be used to check whether braces, brackets and parenthesis are balanced in a program. There are notations to evaluate an arithmetic expression.

An arithmetic expression contains operands and operators. Operands are variables or constants and operators are of various types such as arithmetic unary and binary operators (for example, - (unary). + (addition), - (subtraction), * (multiplication), / (division), ^ (exponentiation), % (remainder modulo), **etc.,** relational operators (for example, <, >, <=, < >, >=, etc.), and Boolean operators (such as, AND, OR, NOT, XOR, etc.). In addition to these, parentheses such as '(' and ')' are also used.

A simple arithmetic expression is as follows:

A + B * C / D - E ^ F * G

There are two ways to fix it. First, we can assign to each operator precedence and associativity.

Precedence and associativity of operators

Operators	*Precedence*	*Associativity*
- (unary), +(unary), NOT	6	-
^ (exponentiation)	6	Right to left
* (multiplication), / (division)	5	Left to right
+ (addition), - (subtraction)	4	Left to right
<, <=, +, < >. >=	3	Left to right
AND	2	Left to right
OR, XOR	1	Left to right

These problems can be solved in the following two steps:

1. Conversion of a given expression into a special notation
2. Evaluation/production of an object code using a slack.

Notations for Arithmetic Expressions

There are three notations to represent an arithmetic expression; they are infix, prefix and postfix (or suffix). For example, A + B, C – D, E * F, G/H

Infix: In this type of expression, the operator comes in between the operands
Here, the notation is
<Operand> <operator> <operand>.
The following are simple expressions in prefix notation:
A + B, C – D, E * F, G/H

Prefix: In this type of expression, the operator comes before the operands.
<Operator> <operand> <operand>
The following are simple expressions in prefix notation:
+AB, -CD. *EF, /GH

Postfix: in this type of expression, the operator comes after the operands.
<Operand> <operand> <operator>
The following expressions are in postfix notation:
AB+, CD-, EF*, *GW*

2. Conversion of an Infix Expression to Postfix Expression

To convert infix expression into postfix expression using a stack data structure, we can use the following steps:

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is **operand**, then directly print it to the result (Output).
3. If the reading symbol is **left parenthesis** '(', then Push it on to the Stack.
4. If the reading symbol is **right parenthesis** ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is **operator** (+, - , *, / etc.,), then Push it on to the Stack.

Example:

Consider the following arithmetic expression:
(A - B) *(D / E)

Step 1- add the ‘(‘at start and ‘)’ at end of the expression.
((A - B) *(D / E))

Step 2-scan the expression form left to right.

	(	(	A	-	B	)	*	(	D	/	E	)	)
Symbol reading:	1	2	3	4	5	6	7	8	9	10	11	12	13

Read symbol	*Stack*	*Output*
1	(	-----
2	((	----
3	((	A
4	((-	A
5	((-	AB
6	(	AB -
7	(*	AB -
8	(*(	AB -
9	(*(	AB - D
10	(*(/	AB - D
11	(*(/	AB - DE
12	(*	AB – DE /
13	--	AB – DE /*

3. Evaluation of a Postfix Expression

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression Evaluation using Stack Data Structure

To evaluate a postfix expression using stack data structure we can use the following steps:

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is **operand**, then push it on to the Stack.
3. If the reading symbol is **operator** (+, - , *, / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally, perform a pop operation and display the popped value as final result.

Example:

Consider the following expression: *infix:* A + (B * C) / D *Postfix:* A B C * D / + (i.e. 2 3 4 * 6 /)
Input: A B C * D / + with A = 2. B = 3, C = 4 and D = 6

Read symbol	*Stack*	
A	2	PUSH(A = 2)
B	2 3	PUSH(B = 3)
C	2 3 4	PUSH(C = 4)
*	2 12	POP(4), POP(3), PUSH(T = 12)
D	2 12 6	PUSH(D = 6)
/	2 2	POP(6), POP(12), PUSH(T = 2)
+	4	POP(2), POP(2), PUSH(T = 4)
		value = POP()

Example:

*Consider the postfix expression: 5 4 – 4 2 / **

Step 1*- add ')' at the end of the expression*

	5	*4*	*-*	*4*	*2*	*/*	***	*)*
Symbol	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Step 2*- Scan the expression from left to right*

Reading symbol	*stack*	*output*
1	*5*	*5*
2	*4*	*5, 4*
3	*-*	*1*
4	*4*	*1, 4*
5	*2*	*1, 4, 2*
6	*/*	*1, 2*
7	***	*1, 2*
8	*)*	*2*

Polish Notations

The process of writing the operators of an expression either before their operands or after them is called the Polish Notation. This notation was introduced by Jan Lukasiewicz. The main property of Polish Notation is that the order in which operations are to be performed is ascertained by the position of the operators and operands in the expression.

The computer system can understand and work only on binary paradigm, it assumes that an arithmetic operation can take place between two operands only. For example, A+B, C×D, D/A, etc. Usually an arithmetic expression may consist of more than one operator and two operands, for example, (A+B)×C(D/(J+D)). These complex arithmetic expressions can be converted into polish strings using stacks which can then be executed in two operands and an operator form.

The notation refers to these complex arithmetic expressions in three forms:

- If the operator symbols are placed before its operands, then the expression is in **prefix notation**.
- If the operator symbols are placed after its operands, then the expression is in **postfix notation**.
- If the operator symbols are placed between the operands then the expression is in **infix notation**.

For example,

Table 6.1 Notations

Infix Notation	*Prefix Notation*	*Postfix Notation*
A + B	+ AB	AB +
(A – C) * B	* – ACB	AC – B *
A + (B * C)	+ A * BC	ABC * +
(A + B)/(C – D)	/ + AB – CD	AB + CD – /
(A + (B * C))/(C – (D * B))	/ + A * BC – C * DB	ABC * + CDB * –/

STACKS AND RECURSION

Suppose a procedure contains either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure. Then such a procedure is called a **recursive procedure**. For example, the problem of factorial can be solved using a recursive procedure.

Recursion may be useful in developing algorithms for specific problems. The stacks may be used to implement recursive procedures.

Before going to stack implementation of recursive procedures, let us discuss about subprograms. A subprogram can contain both parameters and local variables. The parameters are the variables which receive values from objects in the calling program, called arguments and which transmit values back to the calling program. The subprogram, besides the parameters and local variables, also keeps track of return address in the calling program. This return address is essential, since control must be transferred back to its proper place in the calling program. Once the subprogram is finished executing and control is transferred back to its calling program, the values of local variables and return address are no longer needed.

Suppose the subprogram is a recursive program. Then each level of execution of the subprogram may contain different values for parameters and local variables and for the return address. Now, if the recursive subprogram call itself, then these current values must be saved, since they will be used again when program is reactivated.

The translation of recursive procedure into a non-recursive procedure using stack is as follows:

UNIT – III

CHAPTER – II: Queues

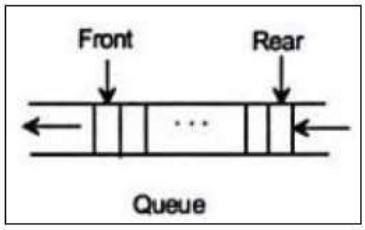
Introduction, Queue as an Abstract data Type, Representation of Queues, Circular Queues, Double Ended Queues- Dequeues, Priority Queues, Application of Queues

---

Q. Define QUEUE:

A queue is a linear data structure in which insertion can take place only at one end, called as REAR, and deletions can take place only at other end, called as FRONT. i.e., In Queue we can add elements at one end called **REAR**(also called tail) and delete the elements at another end called **FRONT**(also called head).

It is also called as **FIFO** (First in First Out) list. The insertion and deletion operations are called as **Enqueue** and **Dequeue.** We can implement the queue by using Arrays and linked list.



Q. What is Queue ADT

1. Initialize a queue to be empty
2. Determine if a queue is empty or not
3. Determine a queue is full or not
4. Insert a new element after the last element in a queue, if it is not full.
5. Delete the first element in a queue, if it is not empty

Q. REPRESENTATION OF QUEUES

Queue data structure can be implemented in two ways. They are as follows

1. Using Arrays
2. Using Linked List

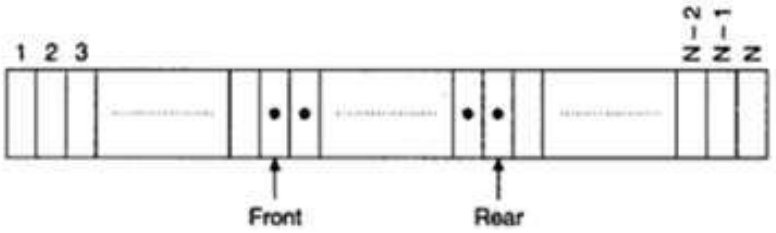
Here, first kind of representation uses one-dimensional array and other representation uses the double linked list.

When queue is implemented using array that can organize only limited no. of elements. When a queue is implemented using linked list that can organize unlimited no. of elements.

1. Array Representation of Queue

A one dimensional array is used to represent a queue. But, queue implemented by using array can store only fixed no. of elements. The implementation of queue using array is very simple. In this representation, two pointers front and rear are used to indicate the two ends of the queue.

Whenever, we want to insert a new element, increment rear value by one and insert at that position. Whenever we want to delete an element from queue, then increment front value by one and then display the value at front position as deleted element.



Three states of a queue are

- Queue is Empty: Front=0, Rear=0
- Queue is Full: Rear=N, Front=1
- Queue contains elements: Front ≤ Rear, No. of elements=Rear-Front+1

Algorithm for ENQUEUE

```
Steps:
1.  If (REAR = N) then                                  // Queue is full
2.      Print "Queue is full"
3.      Exit
4.  Else
5.      If (REAR = 0) and (FRONT = 0) then              // Queue is empty
6.          FRONT = 1
7.      EndIf
8.      REAR = REAR + 1                 // Insert the item into the queue at REAR
9.      Q[REAR] = ITEM
10. EndIf
11. Stop
```

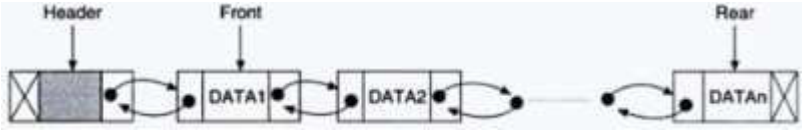
Algorithm for DEQUEUE

```
Steps:
1.  If (FRONT = 0) then
2.      Print "Queue is empty"
3.      Exit
4.  Else
5.      ITEM = Q[FRONT]                                 // Get the element
6.      If (FRONT = REAR)          // When the queue contains a single element
7.          REAR = 0                                    // The queue becomes empty
8.          FRONT = 0
9.      Else
10.         FRONT = FRONT + 1
11.     EndIf
12. EndIf
13. Stop
```

2. Representation of Queue using a Linked List

The major problem with the queue using array is, it will work only for fixed no. of elements. Queue using array is not suitable when we don't know the size of the data. To overcome this problem, a queue data structure can be implemented using linked list representation. In this, we use double linked list which allows us to move both ways. The pointers FRONT and REAR point the first node and last node in the list. A queue represented using a linked list is also known as a linked queue

Note: A queue represented using a linked list is also known as a linked queue



Two states of the queue are:

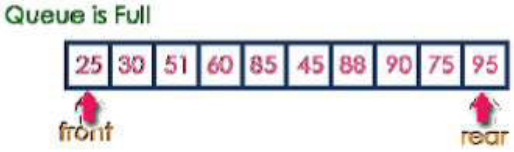
- Queue is Empty: FRONT=REAR=HEADER
- Queue contains atleast one element: HEADER→LINK≠ NULL

Q. Write about VARIOUS QUEUE STRUCTURES (OR) TYPES OF QUEUES

1. Circular Queue

In a normal queue data structure, we can inset elements until queue becomes full. But once if queue becomes full, we cannot insert next element until all the elements are deleted from the queue. For example,

After inserting all elements into the queue

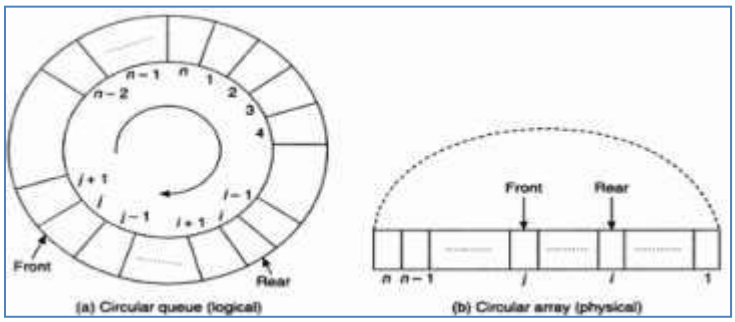


Now consider the following situation after deleting 3 elements from the queue

In above situation,even though we have empty positions in the queue we cannot make use of them to insert new element. Because **REAR** is still at last position.This is the major problem in normal queue.To overcome this problem we use circular queue data structure.

CIRCULAR QUEUE:
Circular Queue is a linear data structure in which the operations are performed on FIFO principle and the last position is connected back to the first position to make a circle.

The graphical representation of circular queue is as follows



Two states of the circular queue are:

- Circular queue is empty
  FRONT=REAR=0
- Circular queue if full
  Front= (REAR mod Length) +1

Algorithm Enqueue_CQ

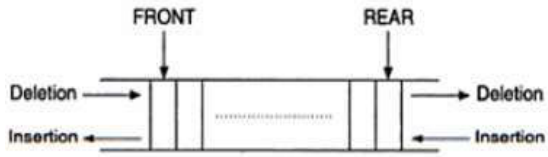
```
Steps:
1.  If (FRONT = 0) then                          // When the queue is empty
2.      FRONT = 1
3.      REAR = 1
4.      CQ[FRONT] = ITEM
5.  Else                                         // Queue is not empty
6.      next = (REAR MOD LENGTH) + 1
7.      If (next ≠ FRONT) then                   // If the queue is not full
8.          REAR = next
9.          CQ[REAR] = ITEM
10.     Else
11.         Print "Queue is full"
12.     EndIf
13. EndIf
14. Stop
```

Algorithm Dequeue_CQ

```
Steps:
1.  If (FRONT = 0) then
2.      Print "Queue is empty"
3.      Exit
4.  Else
5.      ITEM = CQ[FRONT]
6.      If (FRONT = REAR) then              // If the queue contains a single element
7.          FRONT = 0
8.          REAR = 0
9.      Else
10.         FRONT = (FRONT MOD LENGTH) + 1
11.     EndIf
12. EndIf
13. Stop
```

2. DEQUEUE (Double Ended Queue)

A Dequeue is an ordered collection of elements in which elements can be inserted and deleted from both ends. It is also called as "Double Ended Queue".



There are different ways of representing a dequeue.One simple way to represent is by using a double linked list. Another representation is using a circular array (as used in a circular queue)

Operations:

The following four operations are performed on a Dequeue

1. Push_DQ(item) – to insert item at the FRONT end of a Dequeue
2. Pop_DQ() – to remove the FRONT item from a Dequeue
3. Inject(item) – to insert item at the REAR end of a Dequeue
4. Eject() – to remove the REAR item from a Dequeue

Algorithm for Push_DQ

```
Steps:
1.  If (FRONT = 1) then                                  // If FRONT is at extreme left
2.      ahead = LENGTH
3.  Else                                  // If FRONT is at extreme right or the deque is empty
4.      If (FRONT = LENGTH) or (FRONT = 0) then
5.          ahead = 1
6.      Else
7.          ahead = FRONT – 1                  // FRONT is at an intermediate position
8.      EndIf
9.      If (ahead = REAR) then
10.         Print "Deque is full"
11.         Exit
12.     Else
13.         FRONT = ahead                                          // Push the ITEM
14.         DQ[FRONT] = ITEM
15.     EndIf
16. EndIf
17. Stop
```

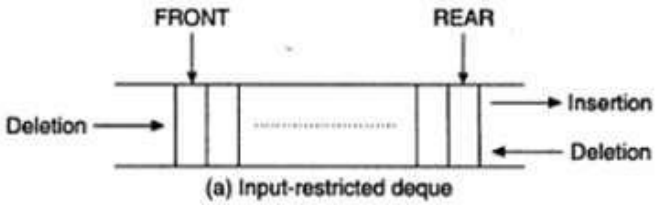
Algorithm Pop_DQ: This algorithm is the same as the algorithm Dequeue_CQ

There are two variations of Dequeue

a) Input-restricted Dequeue
b) Output restricted Dequeue

a. Input-restricted Dequeue

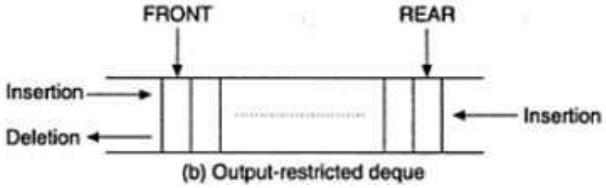
It is a Dequeue which allows insertions at one end (say REAR end) only, but allows deletions at both ends (say FRONT end).



(a) Input-restricted deque

b. Output-restricted Dequeue

It is a Dequeue which allows deletions at one end only (say FRONT end), but allows insertions at both ends (say REAR end).

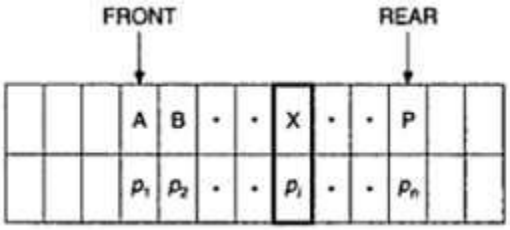


(b) Output-restricted deque

3. PRIORITY QUEUE

Priority queue is a another variation of queue, in which element has been assigned a value, called the priority of the element, and each element can be inserted or deleted not only at the ends but also at any position on the queue.

Priority queue is a variant of queue data structure in which insertion is Performed in the order of arrival and deletion is performed based on the priority



With this structure, an element X of priority pi may be deleted before an element which is at FRONT. Similarly, insertion of an element is based on its priority.

Priority Queue has the following properties.

- Every item has a priority associated with it.
- An element with high priority is Dequeue before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

There are different ways of implementing the structure of Priority Queue. These are:

a) Using a simple/circular array
b) Multi-Queue implementation
c) Using a Double Linked List
d) Using Heap tree

APPLICATIONS OF QUEUE:

UNIT – IV
BINARY TREES

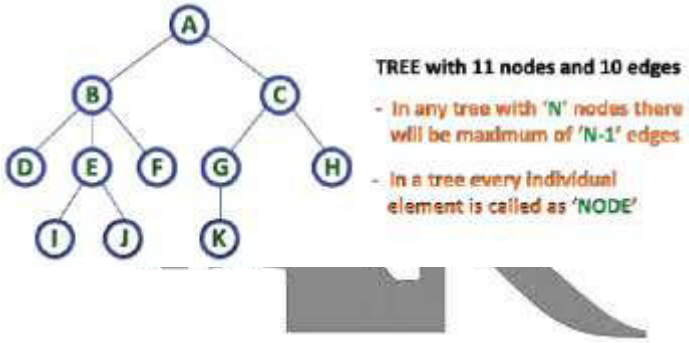
Introduction to Non- Linear Data Structures, Introduction Binary Trees, Types of Trees, Basic Definition of Binary Trees, Properties of Binary Trees, Representation of Binary Trees,
Binary Search Tree, Operations on a Binary Search Tree, Binary Tree Traversal, Counting Number of Binary Trees, Applications of Binary Tree

Q. TREE:

Tree is a nonlinear (or two dimensional) data structure that contains data items or elements arranged in hierarchical format.

A tree contains a finite non empty set of elements, called Nodes which are connected to each other using a finite set of directed lines called Branches.

In a tree data structure, if we have N no. of nodes then we can have a maximum of N-1 number of links.



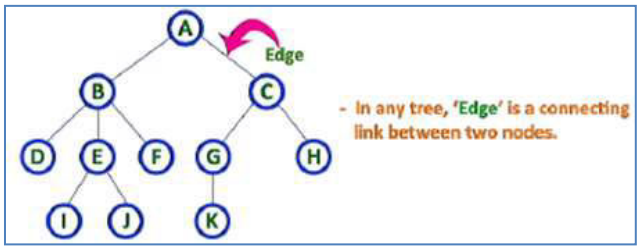
BASIC TERMINOLOGY

1. Root

In a tree data structure, the first node is called as Root node. Every tree must have root node. It is the origin of tree data structure. In any tree, there must be only one root node.

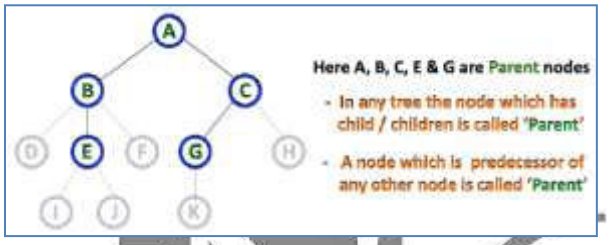
2. Edge

In a tree data structure, the connecting link between any two nodes is called as Edge. In a tree with N no. of nodes there will be a maximum of N-1 no. of edges.



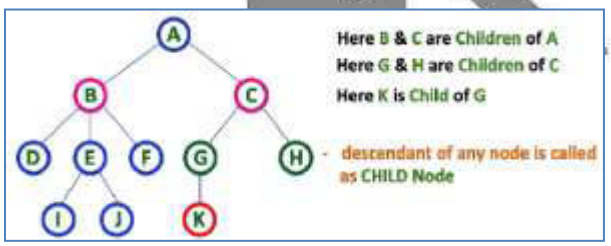
3. Parent

In a tree data structure, the node which is predecessor of any node is called as parent node. In other words, the node which has branch (or child) from it to any other node is called as parent node



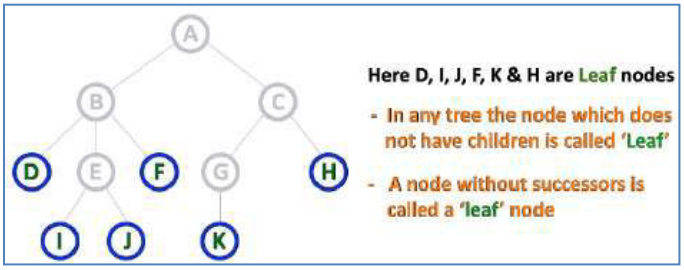
4. Child

In a tree data structure, the node which is descendant of any node is called as child node. In other words, the node which has a link from its parent node is called as child node. In a tree, a parent node has any no. of child nodes. In a tree, all nodes except root re child nodes



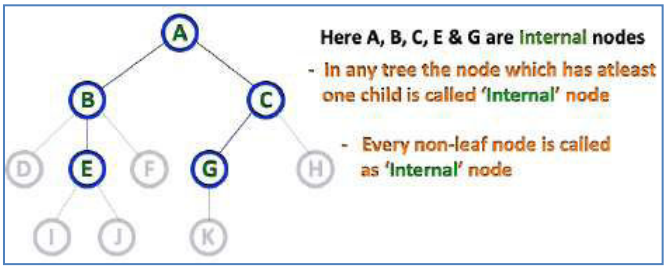
5. Leaf

In a tree data structure, the node which does not have a child is called as Leaf node. In other words, a leaf is a node with no child. In a tree, the leaf nodes are also called as External nodes. External node is also a node with no child. In a tree, leaf node is also called as Terminal mode.



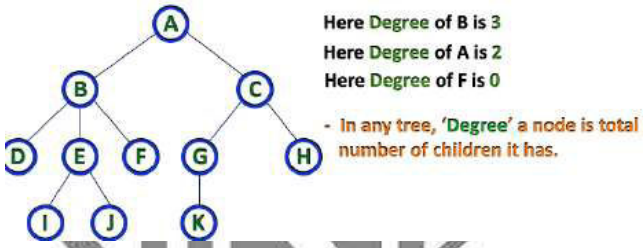
6. Internal nodes

In a tree data structure, the node which has atleast one child is called as internal node. In a tree, nodes other than leaf nodes are called as internal nodes. The root node is also said to be internal node if tree has more than one node. Internal nodes are also called as Non-terminal nodes.



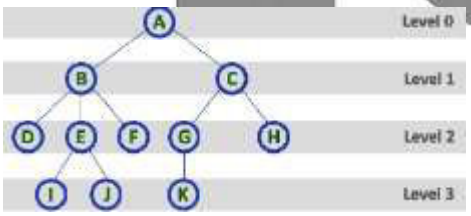
7. Degree

In a tree data structure, the total number of children of a node is called as Degree of that node. In other words, the degree of an element is the number of children it has. The degree of a leaf is zero



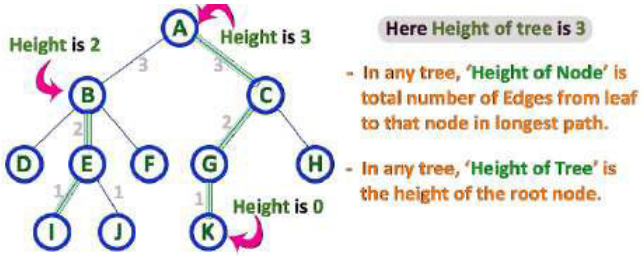
8. Level

In a tree data structure, each step from top to bottom is called as a level and the level count starts from 0 and incremented by one at each level. Root node is at level zero.



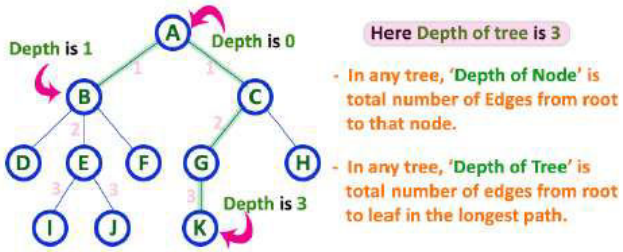
9. Height

In a tree data structure, the total no. of edges from leaf node to a particular node in the longest path is called as Height of that node. In a tree, height of all leaf nodes is '0'.



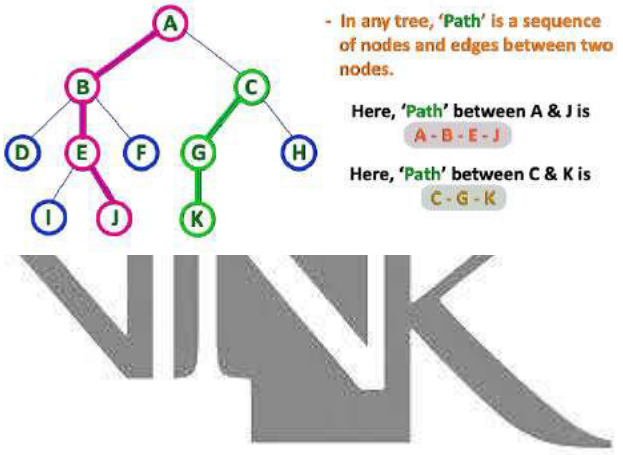
10. Depth

In a tree data structure, the total no. of edges from root node to a particular node in longest path is called as Depth of that node. In a tree, depth of the root node is '0'.



11. Path

In a tree data structure, the sequence of nodes and edges from one node to another node is called as path between those two nodes. Length of a path is total no. of nodes in that path.

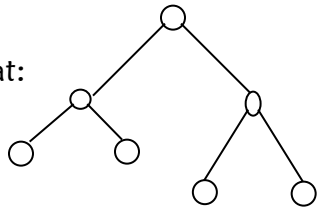


Q. WHAT IS BINARY TREE? WRITE THE PROPERTIES OF BINARY TREES?

BINARY TREE:

A binary tree is a special form of tree. A tree in which every node can have a maximum of two children is called as Binary Tree.

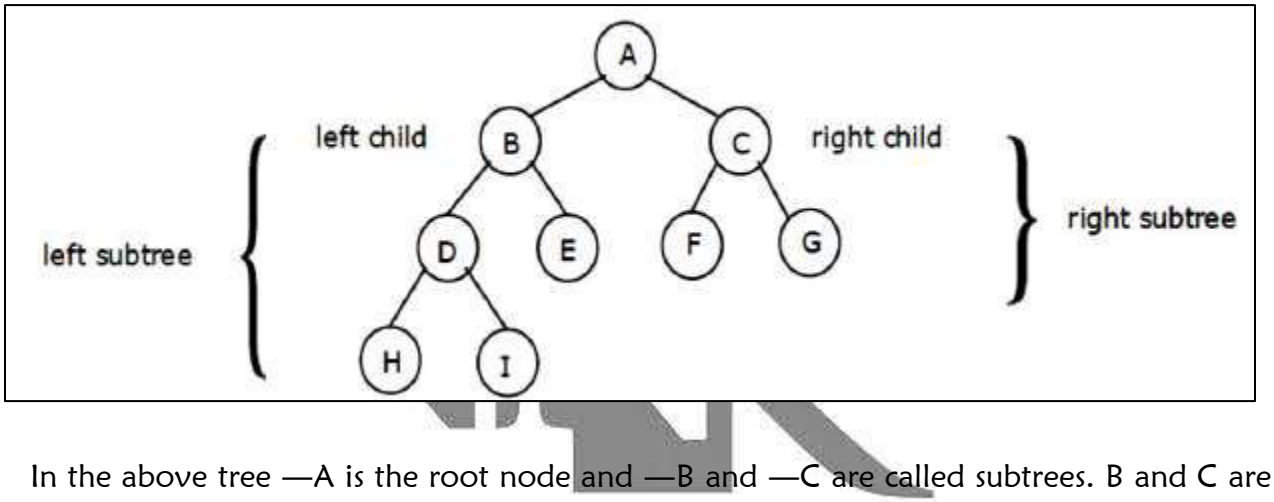
(or)



A binary tree is defined as a finite set of elements, called nodes, such that:

1) Tree is empty (called the null tree or empty tree) or
2) Tree contains a node called root node together with two binary Trees called left sub tree and right sub tree of the root.

Example:



In the above tree —A is the root node and —B and —C are called subtrees. B and C are left and right successor of A. The node A is called parent node and B and C are called children.

All lower level nodes are called descendants and upper level nodes are called ancestors of their descendants.

The line drawn between parent and child is called an edge or arc whereas the line(s) between and ancestor and descendant is called path.

A node without any children is called a terminal or leaf node and all others are called nonterminal or non-leaf node. A path ending with a leaf is called a branch. Nodes of same parent are called ***siblings***.

<u>Note:</u>

- A Binary tree with n nodes has exactly n-1 edges.
- A Binary tree of depth d, has atleast d and at most 2d – 1 node.
- If a binary tree contains n nodes at level L, then it contains at most 2n nodes at level L+1

Properties of binary trees

- The depth of a binary tree containing N nodes is at most N-1.
- The depth of a binary tree containing N leaves is at least [log2(N)].
- A binary tree of depth d has at most 2d leaves.
- The number of nodes in a binary tree of depth d is at most 2d-1-1.
- The average depth of a binary tree containing N nodes is O(N)
- The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$
- The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$
- If a complete binary tree with n nodes (depth = $\lfloor \log_2 n + 1 \rfloor$) is represented sequentially, then for any node with index i, $1 \leq i \leq n$, we have:

1. parent (i) is at $\lfloor i / 2 \rfloor$ if i ≠ 1 if i = 1, i is at the root and has no parent
2. left_child(i) is at 2i if 2i ≤ n. If 2i > n, then i has no left child
3. rightt_child(i) is at 2i + 1 if 2i + 1 ≤ n. If 2i + 1 > n, then i has no right child

Properties of binary trees

1. In any Binary tree, the maximum of nodes on level L is 2^L, where **L ≥ 0**
2. The maximum no. of nodes possible in a binary tree of height h is $2^h - 1$.
3. The minimum no. of nodes possible in a binary tree of height h is h.
4. For any non-empty binary tree, if n is the no. of nodes and e is the no. of edges, then n=e – 1.
5. For any non-empty binary tree T, if n_0 is the no. of leaf nodes (degree=0) and $n_2$is the no. of internal nodes (degree=2), then$n_0 = n_2 + 1$.
6. The height of a complete binary tree with n number of nodes is $[\log_2(n+1)]$
7. The total no. of binary tree possible with n nodes is $\frac{1}{n+1}\ {}^{2n}C_n$

Q. REPRESENTATIONS OF BINARY TREE

A (Binary) tree is represented using two methods. Those methods are

1. Array or Linear Representation
2. Linked List Representation

1. Array or Linear Representation

In Array (or linear) representation of binary tree, we use a 1-D array to represent a binary tree. In this representation, the nodes are stored level by level, starting from the level 0. The size of one dimensional array can be calculated by using following formula.

Formula:

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

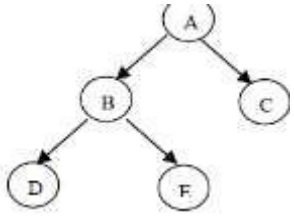
Where d is the depth of the tree. (Depth means maximum levels of the tree)

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

1. Root node is stored at position 1.
2. 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.

Example:

Consider the following incomplete binary tree:



Array size = $2^{d+1} + 1$

$= 2^{2+1} - 1$

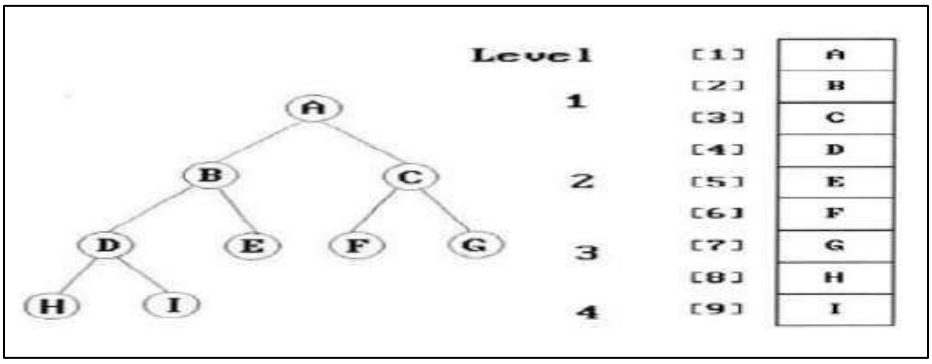
$= 2^3 - 1$

$= 8 - 1$

$= 7$

All the elements are stored in array as follow:

1	2	3	4	5	6	7
A	B	C	D	E	-	-



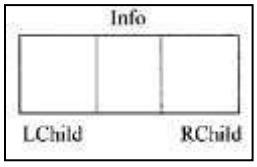
2. Linked Representation of a Binary Tree

We can easily overcome the problems arise due to array representation by using a linked representation. Each node has three fields, left_child, data, and right_child.

We use double linked list to represent a binary tree. In linked representation, every element is represented as node. A node contains three fields such as:

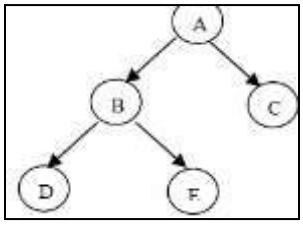
a) Left Child (LChild)
b) Information of the Node (Info)
c) Right Child (RChild)

The LChild contains address of left child node and RChild contains address of right child node of the parent node. An info field contains actual data about node. When a node has no child then the corresponding pointer fields are NULL.

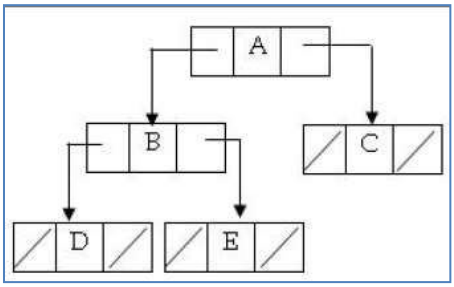


Example:

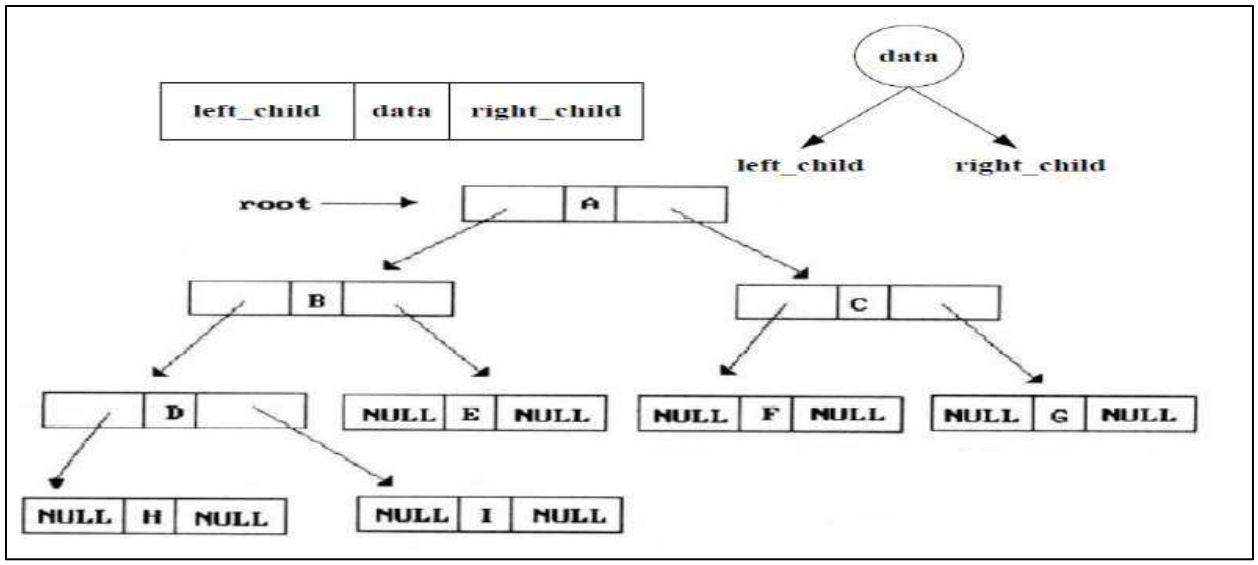
Consider the following incomplete binary tree



In linked representation, above tree can be represented as



Example:



Q. APPLICATIONS OF BINARY TREES

Binary Tree Applications:

- Binary Search Tree Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
- Binary Space Partition Used in almost every 3D video game to determine what objects need to be rendered.
- Binary Tries Used in almost every high-bandwidth router for storing router-tables.
- Hash Trees Used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
- Heaps Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, also used in heap-sort.
- Huffman Coding Tree (Chip Uni) Used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats
- Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
- B-Tree and B+ Tree : They are used to implement indexing in databases.
- Syntax Tree: Used in Compilers.
- K-D Tree: A space partitioning tree used to organize points in K dimensional space.
- Trie : Used to implement dictionaries with prefix lookup.
- Suffix Tree : For quick pattern searching in a fixed text.

Applications of Binary Trees

Binary trees are used to represent a non-linear data structure. There are various forms of Binary trees. Binary trees play a vital role in software applications. One of the most important applications of Binary trees is in the searching algorithms. Most efficient and commonly used search known as Binary search uses a special form of binary tree known as Binary Search tree. A binary search tree always has two children and the left child node is always lighter than the root node, right child node is always heavier than the root node. Binary search tree brings down the time complexity of algorithm to less than 50%.

Another application of binary trees may be seen in populating a voluminous, relational and hierarchical data, into the memory. This increases the efficiency of the algorithm which manages this data. This is because Binary trees allow the algorithms to access a particular node at low cost and Binary trees also help to insert a new node easily.

Similarly, Binary trees are used in decision making, artificial intelligence, compilers, expression evaluation, etc.

APPLICATIONS OF BINARY TREE:

1) Symbol Table Construction:

The notion of symbol table arises frequently in computer science while building compilers, loaders, linkers, assemblers etc. A symbol table is a set of name-value pairs. Associated with each name in the table is an attribute, a collection of attributes, or some directions about what further processing is needed. One of the criteria that a symbol table routine must meet is that the table searching must be performed efficiently. This requirement originates in the compilation phase while handling many lexemes and tokens of the program. The three required operation of the symbol table are:

a) Insertion of new entry
b) Deletion of existing entry
c) Looking up information of an existing entry.

Each of above operation requires searching.

Generally, a tree is used to construct a symbol table because

a) if the symbol table entries as encountered are uniformly distributed according to lexicographic order, then table searching becomes approximately equivalent to a binary search, as long as the tree is maintained in lexicographic order.
b) A binary tree is easily maintained in lexicographic order.

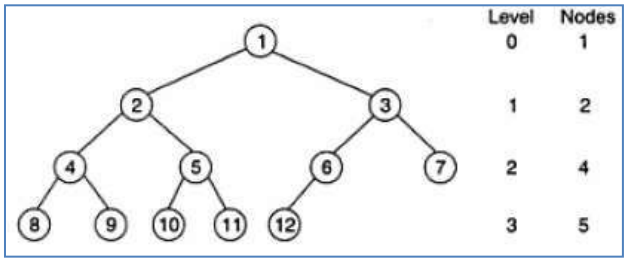
2) Manipulation of the Arithmetic Expressions:

We observed that the formulas in Reverse polish notation are very useful in the compilation process. There is a close relationship between binary trees and formulas in prefix or suffix notations. Let us write the infix formula as a binary tree where a node has an operator as a value and where the left and right sub trees are the left and right operands of that operator. The leaves of the tree are the variables and constants of the expression. We represent the expression in binary tree due to similarities of infix to inorder and postfix to postorder traversal of tree. The tree used for expression is called parse tree.

Q. EXPLAIN ABOUT TYPES OF BINARY TREES

1. Complete Binary Tree:

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.



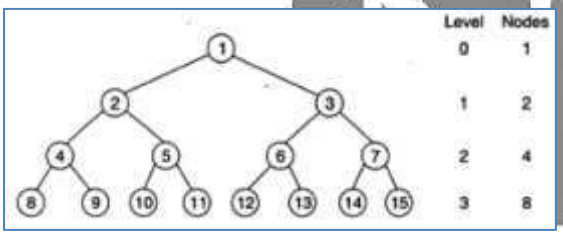
A complete binary tree of height 4

2. Full binary tree:

A binary tree is said to be complete binary tree if all the leaf nodes are at same level. It is also known as full binary tree.

(Or)

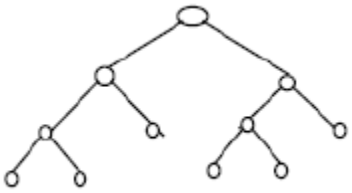
A full binary tree is a tree in which there is one node at root level 0 ($2^0 = 1$), two nodes at level 1($2^1 = 2$), four nodes at level 2 ($2^2 = 4$) and so on.

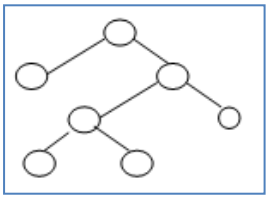


A full binary tree of height 4

3. Extended Binary Tree (Strictly Binary Tree or 2-tree):

A binary tree is said to be Extended binary tree if each node has either 0 or 2 children. In this case the leaf nodes are called external nodes and the node with two children are called internal nodes.

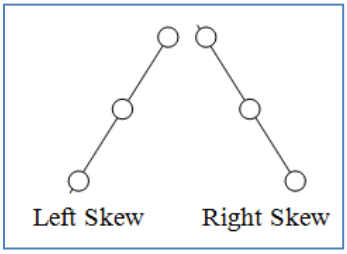




4. SKEWED TREE:

A tree is called Skew if all the nodes of a tree are attached to one side only. i.e. A left skew will not have any right children in its each node and right skew will not have any left child in its each node.

Note: If a binary tree has only left subtree, then it is called left skewed binary tree. If a binary tree has only right sub trees, then it is called as right skewed binary tree.



Q. EXPLAIN VARIOUS METHODS OF TRAVERSING OF A BINARY TREE

(or)

TREE TRAVERSAL TECHNIQUES

The traversal operation is used to visit each node in the tree exactly once. A tree can be traversed in different ways. The most commonly used traversals are in-order, pre-order and post-order traversal.

a)Pre-order traversal

In this traversal, the root is visited first, then left sub-tree and then right sub-tree is visited in pre-order method.

The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

Algorithm for preorder traversal

```
void preorder(node root)
{
if( root != NULL )
{
print root . data;
preorder (root . lchild);
preorder (root . rchild);
}
}
```

b) In-order traversal

In this traversal, before visiting the root node, the root node of the left sub-tree is visited, and then the root node and then the root node of the right sub-tree are visited in in-order method.

The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node root)
{
if(root != NULL)
{
inorder(root . lchild);
print root . data;
inorder(root . rchild);
}
}
```

c) Post-order traversal

In this traversal, first visit the left sub-tree, then the right sub-tree, and then lastly the root is visited in pre-order method.

The steps for traversing a binary tree in postorder traversal are:

1.Visit the left subtree, using postorder.
2.Visit the right subtree, using postorder
3.Visit the root.

Algorithm for postorder traversal

```
void postorder(node root)
{
if( root != NULL )
{
postorder (root . lchild);
postorder (root . rchild);
print (root . data);
}
}
```

Pre-order	1. visit the root 2. Traverse the left sub-tree in pre-order 3. Traverse the right sub-tree in pre-order	Traversing a tree in the order Root→Left→Right
In-order	1. Traverse the left sub-tree in pre-order 2. visit the root 3. Traverse the right sub-tree in pre-order	Traversing a tree in the order Left→Root→Right
Post-order	1. Traverse the left sub-tree in pre-order 2. Traverse the right sub-tree in pre-order 3. visit the root	Traversing a tree in the order Left→Right→Root

Level order Traversal:

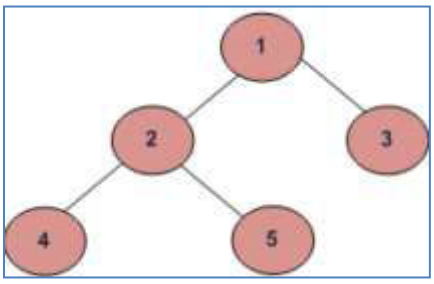
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique.

Algorithm for level order traversal

```
void levelorder( )
{
int j;
for(j = 0; j < ctr; j++)
{
if(tree[j] != NULL)
print tree[j] . data;
}
}
```

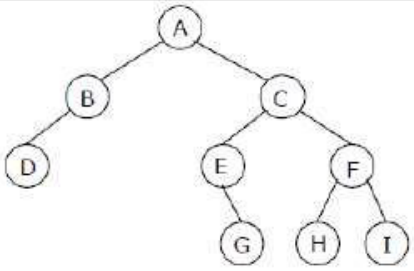
Example

Consider the following tree

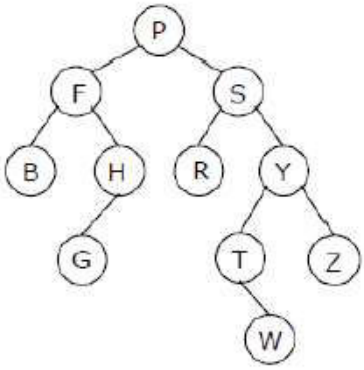


(a) Inorder (Left, Root, Right): 4 2 5 1 3
(b) Preorder (Root, Left, Right): 1 2 4 5 3
(c) Postorder (Left, Right, Root): 4 5 2 3 1

Example:



- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I



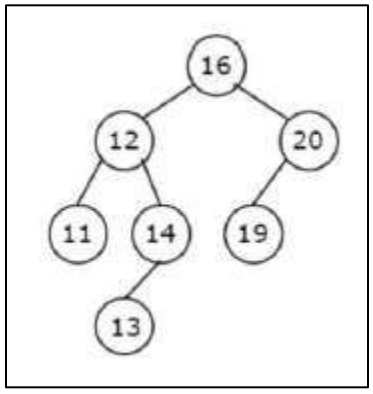
- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
  P, F, S, B, H, R, Y, G, T, Z, W

Q. EXPLAIN VARIOUS OPERATIONS ON BST

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

- Every element has a key and no two elements have the same key.
- The keys in the left subtree are smaller than the key in the root.
- The keys in the right subtree are larger than the key in the root.
- The left and right subtrees are also binary search trees.

Example:



BST Operations:

The basic operations that can be performed on a binary search tree data structure are the following

1. Insert node
2. Searching
3. Delete a node
4. Traversal

1. Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm:

Step1: Compare item with the root(N) of the tree as

If (item<N)proceed to the left child of N.

If(item>N) proceed to the right child of N

Step2: Repeat step1 until one of the following occurs.

We meet a node N such that item = N. in this case the search is successful.

We meet an empty sub tree, which indicates the search is unsuccessful. Insert item in place of the empty sub tree.

Step 3: exit

2. Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, searches for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
        return root
else
        while data not found
        If data is greater than node.data
                goto right subtree
        else
                goto left subtree
If data found
        return node
end while
        return data not found
end if
```

3.Binary Search Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree

a) **In-order Traversal**
b) **Pre-order Traversal**
c) **Post-order Traversal**

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

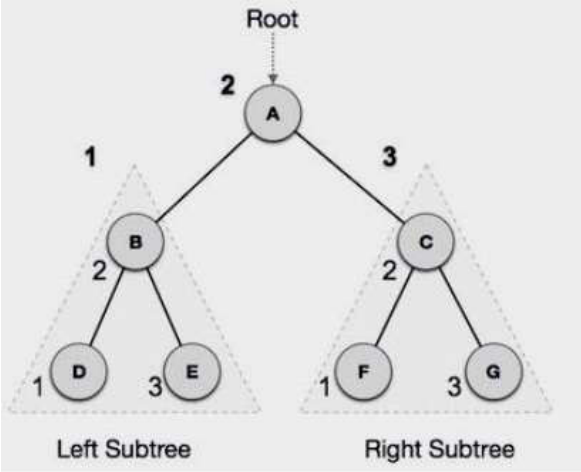
a) In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order. We start from **A**, and following in-order traversal, we move to its left

subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

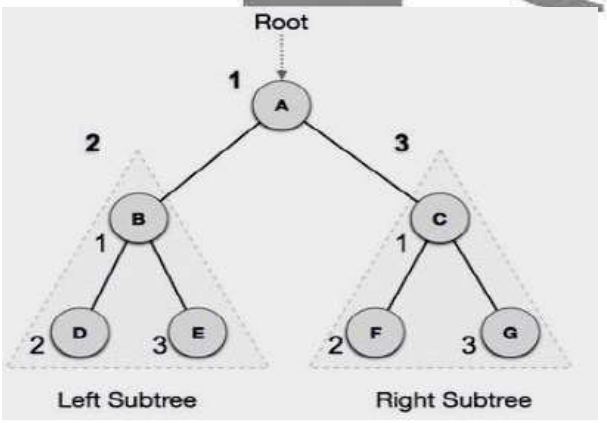
$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G.$



b) Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree. We start from **A**, and following pre-order traversal, we first visit **A**itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

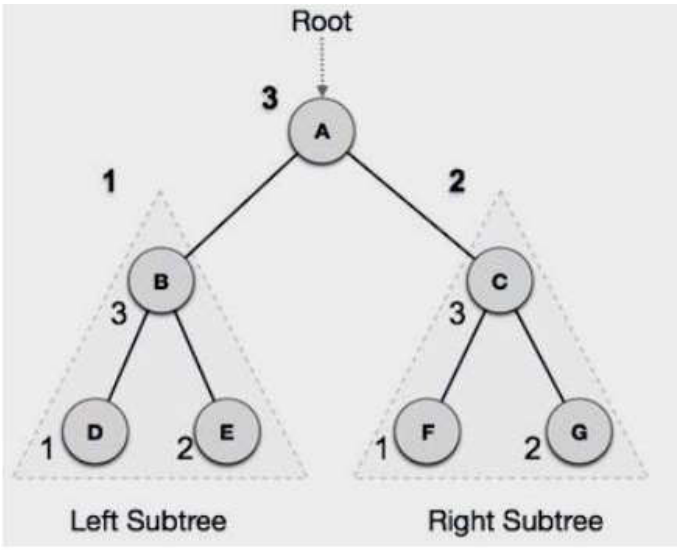


c) Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$



4. DELETING A NODE

Removing is the most complicated operation from the basic binary search Tree operations. After it the tree must keep its order. The first step before we remove an element from the tree is to find it. We have three cases.

- Deleting a node with no children
- Deleting a node with one sub tree
- Deleting a node with two sub trees

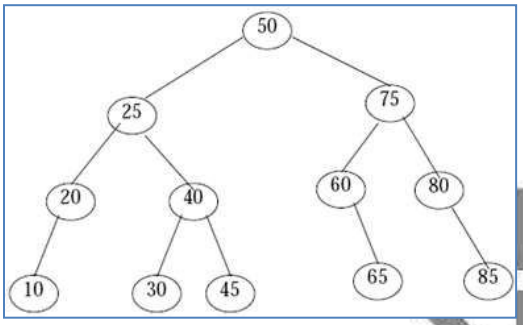
1. if the tree "null" print the message accordingly.
2. First, search for that starting from the root. i.e. move to left the element is less than parent or right till the element is encountered or empty node is encountered.
3. if empty node is encountered, print no element.
4. otherwise, if the node has no children, delete it directly.
5. if the node has only left child, parent node must be linked to its left before deleting the node.
6. if the node has only right child, parent node must be linked to its right before deleting the node.
7. if a node to be deleted has two children, find minimum value in its right sub tree. Replace the element to be deleted with the minimum element and repeat the same process to delete the node containing minimum element.

Q. BINARY SEARCH TREE(BST)

A Binary Search Tree is a binary tree, which is either empty or satisfies the following properties:

1. Every node has a value and no two nodes have the same value (i.e., all the values are unique).
2. If there exists a left child or left sub tree then its value is less than the value of the root.
3. The value(s) in the right child or right sub tree is larger than the value of the root node.

All the nodes or sub trees of the left and right children follows above rules. A typical binary search tree is as follows. Here the root node information is 50. Note that the right sub tree node's value is greater than 50, and the left sub tree nodes value is less than 50. Again right child node of 25 has large values than 25 and left child node has small values than 25. Similarly right child node of 75 has large values than 75 and left child node has small values that 75 and so on.



Q. OPERATIONS ON Binary Search Tree

The operations performed on binary tree can also be applied to Binary Search Tree (BST).

1. Inserting a node
2. Searching a node
3. Deleting a node
4. Traversal

1. INSERTING A NODE

A BST is constructed by the repeatedly insertion of new nodes to the tree structure. Inserting a node in to a tree is achieved by performing two separate operations.

a) The tree must be searched to determine where the node is to be inserted.
b) Then the node is inserted into the tree.

Suppose a "DATA" is the information to be inserted in a BST.

***Step* 1:** Compare DATA with root node information of the tree

(*i*) If (DATA < ROOT →Info)
Proceed to the left child of ROOT

(*ii*) If (DATA > ROOT →Info)
Proceed to the right child of ROOT

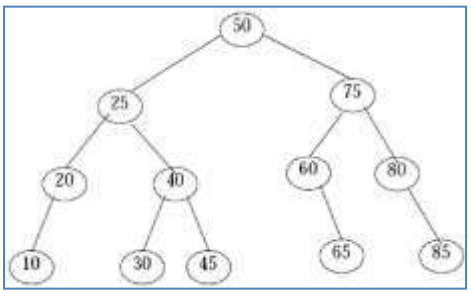
***Step* 2:** Repeat the Step 1 until we meet an empty sub tree, where we can insert the DATA in place of the

empty sub tree by creating a new node.

***Step* 3:** Exit

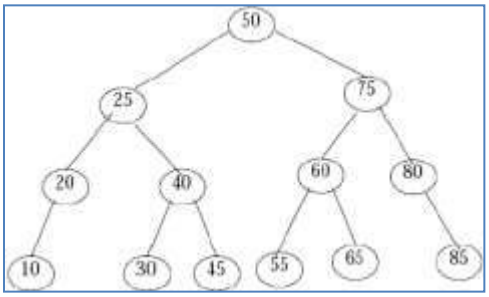
For example:

Consider a binary search tree.



Suppose we want to insert a DATA = 55 in to the tree, then following steps one obtained:

1. Compare 55 with root node info (*i.e.*, 50) since 55 > 50 proceed to the right sub tree of 50.
2. The root node of the right sub tree contains 75. Compare 55 with 75. Since 55 <75 proceed to the left sub tree of 75.
3. The root node of the left sub tree contains 60. Compare 55 with 60. Since 55 < 60 proceed to the right sub tree of 60.
4. Since left sub tree is NULL place 55 as the left child of 60



Binary tree after inserting node 55

ALGORITHM

NEWNODE is a pointer variable to hold the address of the newly created node. DATA is the information to be pushed.

1. Input the DATA to be pushed and ROOT node of the tree.
2. NEWNODE = Create a New Node.
3. If (ROOT == NULL)

   (*a*) ROOT=NEW NODE

4. Else If (DATA < ROOT→Info)
   (*a*) ROOT = ROOT → Lchild
   (*b*) GoTo Step 6
5. Else If (DATA > ROOT → Info)
   (*a*) ROOT = ROOT → Rchild
   (*b*) GoTo Step 7
6. If (DATA < ROOT → Info)
   (*a*) ROOT → LChild = NEWNODE
7. Else If (DATA > ROOT → Info)
   (*a*) ROOT → RChild = NEWNODE
8. NEW NODE → Info = DATA
9. NEW NODE → LChild = NULL
10. NEW NODE → RChild = NULL
11. EXIT

2. SEARCHING A NODE

Searching algorithm is used to search an element from a binary search tree.

ALGORITHM

1. Input the DATA to be searched and assign the address of the root node to ROOT.
2. If (DATA == ROOT → Info)
   (*a*) Display "The DATA exist in the tree"
   (*b*) GoTo Step 6
3. If (ROOT == NULL)
   (*a*) Display "The DATA does not exist"
   (*b*) GoTo Step 6
4. If(DATA > ROOT→Info)
   (*a*) ROOT = ROOT→RChild
   (*b*) GoTo Step 2
5. If(DATA < ROOT→Info)
   (*a*) ROOT = ROOT→Lchild
   (*b*) GoTo Step 2
6. Exit

3. DELETING A NODE

In this algorithm, first search and locate the node to be deleted. Then any one of the following conditions arises:

1. The node to be deleted has no children
2. The node has exactly one child (or sub tress, left or right sub tree)
3. The node has two children (or two sub tress, left and right sub tree)

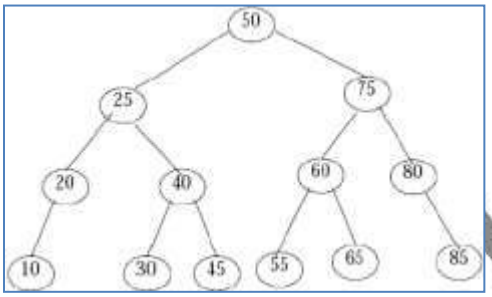
- Suppose the node to be deleted is N. If N has no children then simply delete the node and place its parent node by the NULL pointer.

- If N has one child, check whether it is a right or left child. If it is a right child, then find the smallest element from the corresponding right sub tree. Then replace the smallest node information with the deleted node. If N has a left child, find the largest element from the corresponding left sub tree. Then replace the largest node information with the deleted node.

- The same process is repeated if N has two children, *i.e.*, left and right child. Randomly select a child and find the small/large node and replace it with deleted node.

Example:

Consider a binary search tree.

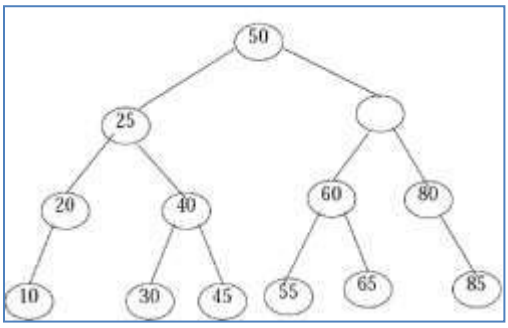


If we want to delete 75 from the tree, following steps are obtained:

***Step* 1:** Assign the data to be deleted in DATA and NODE = ROOT.

***Step* 2:** Compare the DATA with ROOT node, *i.e.*, NODE, information of the tree. Since (50 < 75)
NODE = NODE → RChild

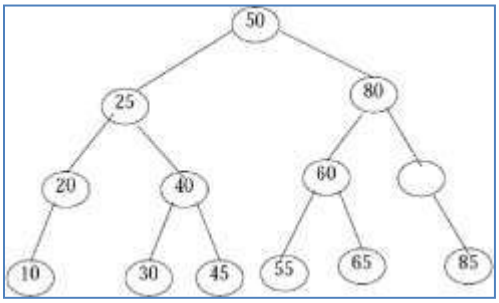
***Step* 3:** Compare DATA with NODE. Since (75 = 75) searching successful. Now we have located the data to
be deleted, and delete the DATA.



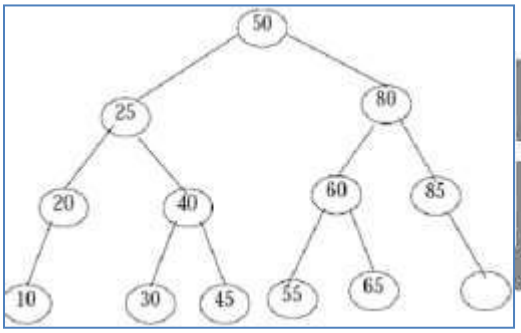
***Step* 4:** Since NODE (*i.e.*, node where value was 75) has both left and right child choose one. (Say Right Sub

Tree) - If right sub tree is opted then we have to find the smallest node. But if left sub tree is opted then we have to find the largest node.

***Step* 5**: Find the smallest element from the right sub tree (*i.e.*, 80) and replace the node with deleted node.

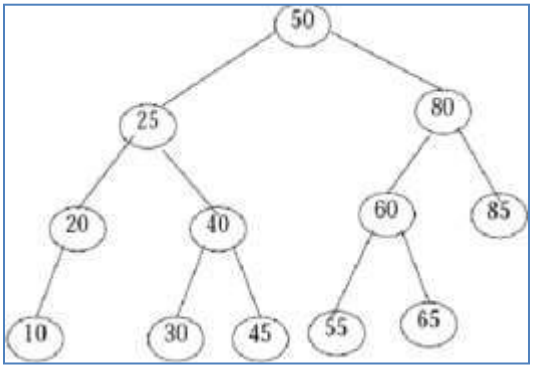


Step 6: Again the (NODE→Rchild is not equal to NULL) find the smallest element from the right Subtree (i.e. 85) and replace it with empty node.



Step 7: Since (NODE→Rchild=NODE→Lchild=NULL) delete the NODE and place NULL in the parent

node



***Step* 8:** Exit.

ALGORITHM

NODE is the current position of the tree, which is in under consideration. LOC is the place where node is to be replaced. DATA is the information of node to be deleted.

1. Find the location NODE of the DATA to be deleted.
2. If (NODE = NULL)
    (a) Display "DATA is not in tree"
    (b) Exit
3. If(NODE → Lchild = NULL)
    (a) LOC = NODE
    (b) NODE = NODE → RChild
4. If(NODE → RChild= =NULL)
    (a) LOC = NODE
    (b) NODE = NODE → LChild
5. If((NODE → Lchild not equal to NULL) && (NODE → Rchild not equal to NULL))
    (a) LOC = NODE → RChild
6. While(LOC → Lchild not equal to NULL)
    (a) LOC = LOC → Lchild
7. LOC → Lchild = NODE → Lchild
8. LOC → RChild= NODE → RChild
9. Exit

UNIT- 5
CHAPTER – 1: SEARCHING AND SORTING

Sorting–An Introduction, Bubble Sort, Insertion Sort, Merge Sort, Searching – An Introduction, Linear or Sequential Search, Binary Search, Indexed Sequential Search

Q. SORTING:

Sorting is one of the most important and fundamental operations performed in computer science. Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Here the data may be of any type like numerical, alphabetical or alphanumeric.

Basic Terminology

1. Internal Sorting:

If sorting is performed on the list of elements that are stored in main memory, then it is called Internal Sorting. This technique is applied to lists that have small no. of elements, due to memory constraints

2. External Sort:

If sorting is performed on the list of elements that are stored in secondary memory, then it is called External Sorting. This technique is applied to lists that have large no. of elements.

3. Ascending Order:

An arrangement of data is in increasing order of elements is called ascending order. If *Ai* and *Aj* are two data items and *Ai* proceeds *Aj* then *Ai<= Aj.*

Eg: {1, 2, 3, 4, 5, 6, 7, 8}

4. Descending Order:

An arrangement of data is in decreasing order of elements is called descending order. If *Ai* and *Aj* are two data items and *Ai* proceeds *Aj* then *Ai>= Aj.*

Eg: {8, 7, 6, 5, 4, 3, 2, 1}

5. Lexicographic Order:

Arranging character or string data values into dictionary order is known lexicographic order.

Eg: {Ant, Bat, Cat, Doll, Egg}

6. Swap:

Swap between two data storages implies the interchange of their contents.

Eg: Before Swap a[1] = 10 a[2] = 20 After Swap a[1] = 20 a[2] = 10.

Q. EXPLAIN ABOUT DIFFERENT TYPES OF SORTING TECHNIQUES

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, or any user-defined order. Sorting can be classified in two types;

The sorting techniques are divided into two categories. These are:

1. **Internal Sorting**

   If sorting is performed on the list of elements that are stored in main memory, then it is called Internal Sorting. This technique is applied to lists that have small no. of elements, due to memory constraints. There are 3 types of internal sorting techniques.

   - SELECTION SORT
     - Selection sort algorithm, Heap Sort algorithm
   - INSERTION SORT
     - Insertion sort algorithm, Shell Sort algorithm
   - EXCHANGE SORT
     - Bubble Sort Algorithm, Quick sort algorithm

2. **External Sorting**

   If sorting is performed on the list of elements that are stored in secondary memory, then it is called External Sorting. This technique is applied to lists that have large no. of elements. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.

   Ex: - Merge Sort

DIFFERENT TYPES OF SORTING METHODS

There are several sorting methods or strategies available to sort data. Each method follows a different strategy/algorithm to sort data.

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort

Q. Explain the algorithm for bubble sort and give a suitable example.

(OR)

Explain the algorithm for exchange sort with a suitable example.

BUBBLE SORTING:

Bubble Sort is a simple sorting algorithm. It is also known as **Exchange sort.** In this, each pair of adjacent elements is compared and the elements are swapped if they are not in proper order. This process continues until the entire list is sorted.

Process:

In bubble sort method the list is divided into two sub-lists sorted and unsorted. The smallest element is bubbled from unsorted sub-list. After moving the smallest element the imaginary wall moves one element ahead. The bubble sort was originally written to bubble up the highest element in the list. But there is no difference whether highest / lowest element is bubbled. This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.

Algorithm

```
for i ← 1 to n do
        for j ← 1 to n-i do
                If Array[j] > Array[j+1] then        /* For decreasing order use < */
                        temp ← Array[j]
                        Array[j] ← A [j+1]
                        Array[j+1] ← temp
```

Example:

Consider the unsorted elements 50 40 30 20 10

Pass 1:

Step 1- compare first two values (i.e. 50 & 40)	**50 40** 30 20 10
As 50>40, swap these values	
Then new sequence is	40 50 30 20 10
Step 2- take Next two vales (i.e. 50 & 30)	40 **50 30** 20 10
As 50>30, swap these values	
Then new sequence is	40 30 50 20 10
Step 3- take Next two vales (i.e. 50 & 20)	40 30 **50 20** 10
As 50>20, swap these values	
Then new sequence is	40 30 20 50 10

Step 4- take Next two vales (i.e. 50 & 10) 40 30 20 **50 10**

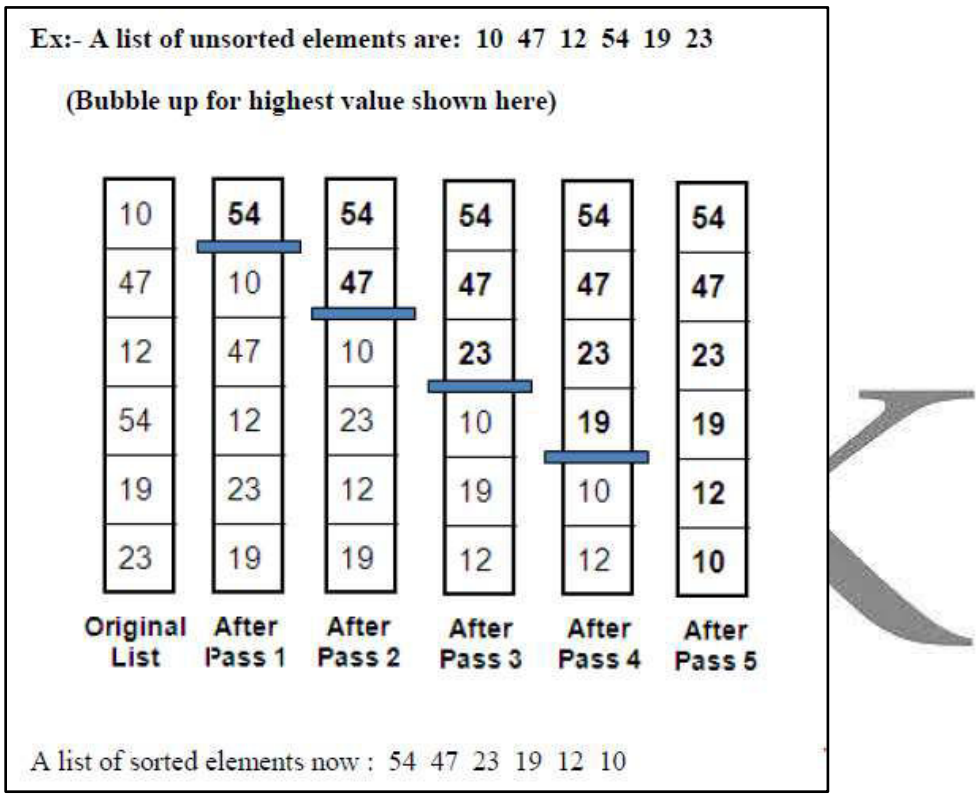
As 50>20, swap these values

Then new sequence is 40 30 20 10 **50**

Continue these steps until all the elements are in proper order. Finally we get

10 20 30 40 50

Example:



Example

Consider an array A of 5 element

A[0]	45
A[1]	34
A[2]	56
A[3]	23
A[4]	12

Pass-1: The comparisons for pass-1 are as follows.

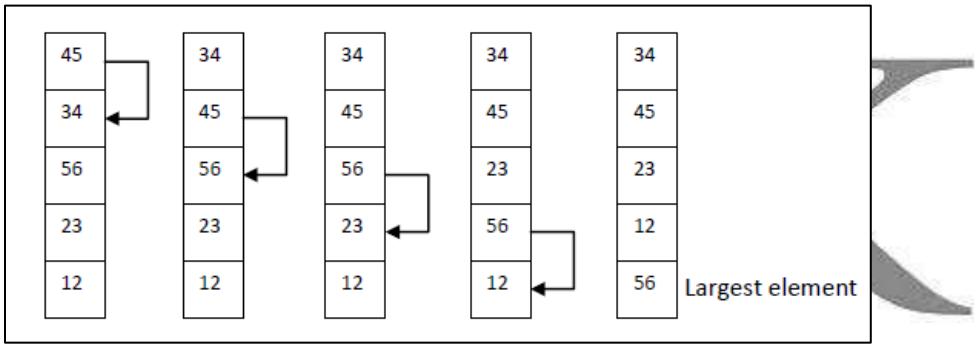
Compare A[0] and A[1]. Since 45>34, interchange them.

Compare A[1] and A[2]. Since 45<56, no interchange.

Compare A[2] and A[3]. Since 56>23, interchange them.

Compare A[3] and A[4]. Since 56>12 interchange them.

At the end of first pass the largest element of the array, 56, is bubbled up to the last position in the array as shown.

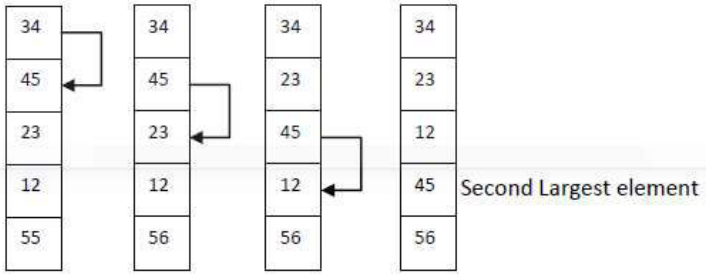


Pass-2: The comparisons for pass-2 are as follows.

Compare A[0] and A[1]. Since 34<45, no interchange.

Compare A[1] and A[2]. Since 45>23, interchange them.

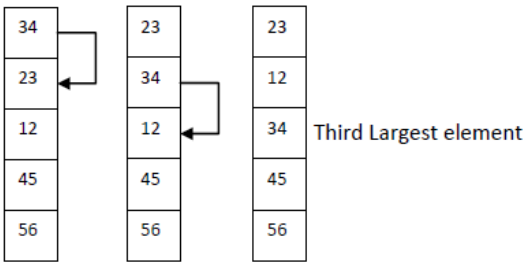
Compare A[2] and A[3]. Since 45>12, interchange them.



Pass-3: The comparisons for pass-3 are as follows.

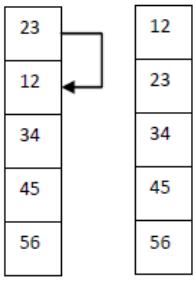
Compare A[0] and A[1]. Since 34>23, interchange them.

Compare A[1] and A[2]. Since 34>12, interchange them.



Pass-4: The comparisons for pass-4 are as follows.

Compare A[0] and A[1]. Since 23>12, interchange them.



2. INSERTION SORTING

In insertion sort, the list of elements is sorted by shifting the elements one by one. Suppose we have 'n' elements, we need n-1 passes to sort the elements.

Process:

The insertion sort is performed using following steps

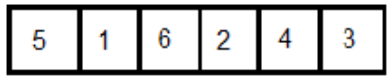
1. Assume the second element (i.e. element at index 1) as **key**
2. Compare the **key** element with the element(s) before it
   - If the **key** element is less than the first element, then place the key element before the first element
   - If the **key** element is greater than the first element, then place the key element after the first element
3. Then, take the third element as **key** and compare it with elements to its left and place it at the proper position.
4. Continue this process until the list is sorted

Algorithm

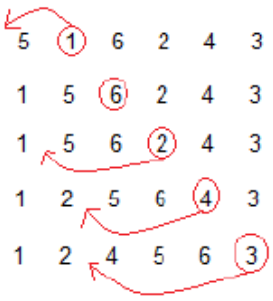
INSERTION_SORT (*A*)
1. **FOR** *j* ← 2 **TO** length[*A*]
2. **DO** key ← *A*[*j*]
3. {Put *A*[*j*] into the sorted sequence *A*[1 . . *j* − 1]}
4. *i* ← *j* − 1
5. **WHILE** *i* > 0 and *A*[*i*] > key
6. **DO** *A*[*i* +1] ← *A*[*i*]
7. *i* ← *i* − 1
8. *A*[*i* + 1] ← key

Example:

In this array, first we pick 1 as key, compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and are compared with, 6 and 5, and then '2' is placed after 1. And this process continues, until complete array will sort.



Lets take this Array.



(Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

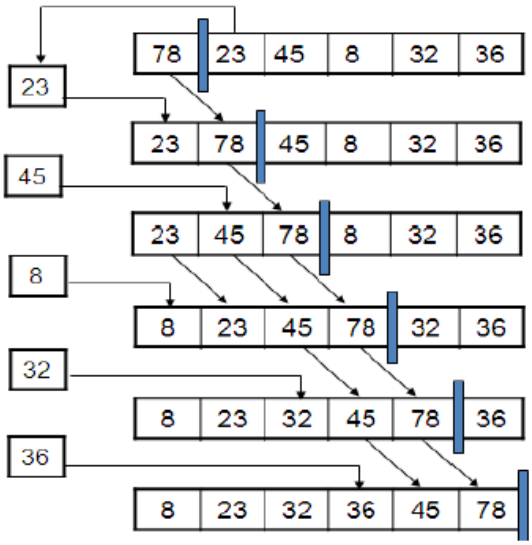
6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

Example:

Ex:- A list of unsorted elements are: 78 23 45 8 32 36 . The results of insertion sort for each pass is as follows:-



A list of sorted elements now : 8 23 32 36 45 78

3. MERGE SORT

Merge sort is based on the **divide-and-conquer** method. The basic concept of merge sort is divides the list into two smaller sub-lists of equal size. Recursively repeat this procedure till only one element is left in the sub-list. After this, various sorted sub-lists are merged to form sorted list. This process goes on recursively till the original sorted list arrived.

It operates as follows:

- **DIVIDE:** Partition the n-element sequence to be sorted into two subsequences of n/2 elements each.
- **CONQUER:** Sort the two subsequences recursively using the merge sort.
- **COMBINE:** Merge the two sorted subsequences of size n/2 each to produce the sorted sequence contains n elements.

Algorithm:

To sort $A[p .. r]$:

1. **Divide Step**

   If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

2. **Conquer Step**

   Conquer by recursively sorting the two sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$.

3. **Combine Step**

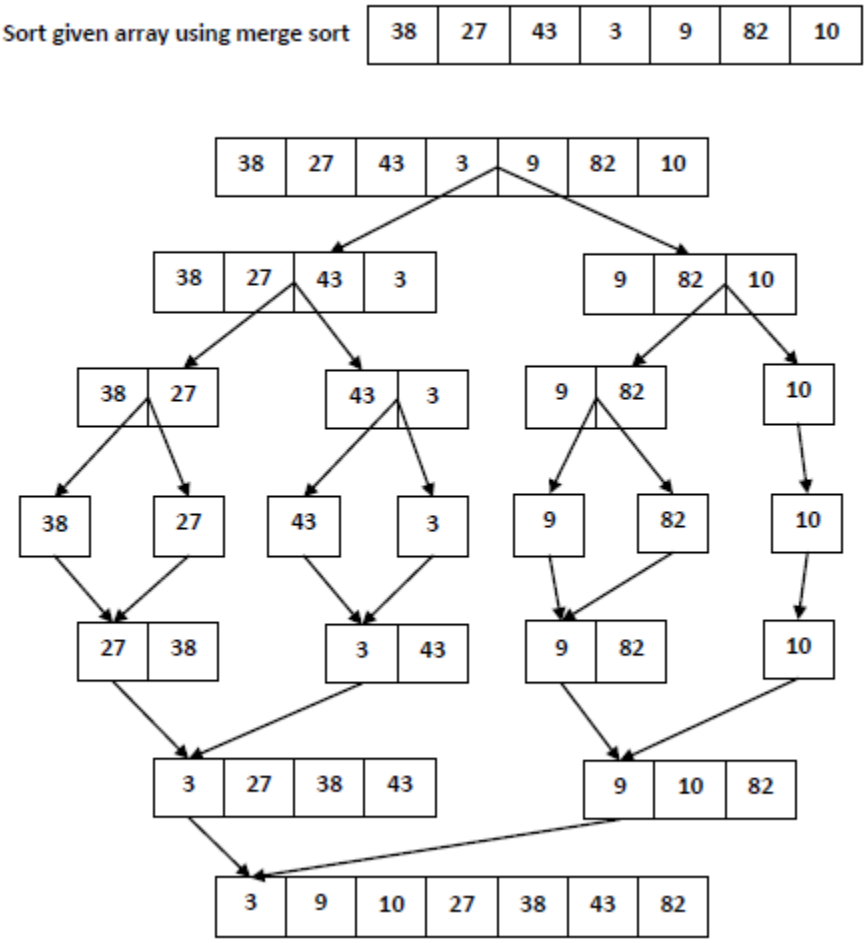
   Combine the elements back in $A[p .. r]$ by merging the two sorted sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

MERGE-SORT (A, p, r)

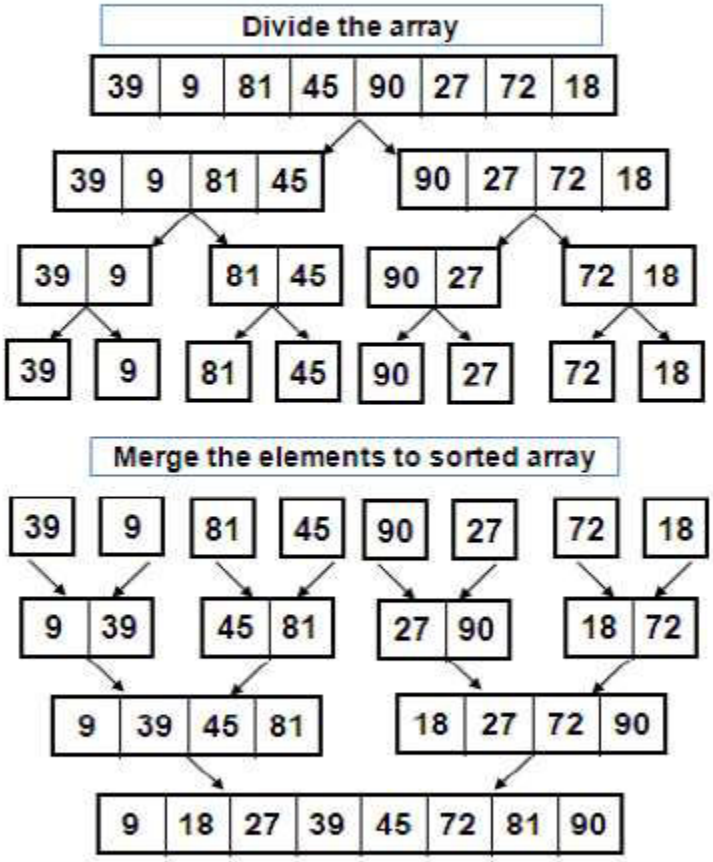
```
1. IF p < r                          // Check for base case
2. THEN q = FLOOR[(p + r)/2]         // Divide step
3. MERGE (A, p, q)                   // Conquer step.
4. MERGE (A, q + 1, r)               // Conquer step.
5. MERGE (A, p, q, r)                // Conquer step.
```

Example

Sort given array using merge sort



Ex:- A list of unsorted elements are: 39 9 81 45 90 27 72 18



Sorted elements are: 9 18 27 39 45 72 81 90

Q. Write a short note on SEARCHING

Searching is a process of finding a value in a list of values (or) Searching refers to the operation of finding the location of a given item in list of items.

Consider an array is given with n elements. A specific element item is given to search. Now, we want to find whether the item is available in the list of n elements or not. If the search item is exist, then it refers to successful search; otherwise, it refers to unsuccessful search.

Most important techniques used for search operation are:
1. Linear search
2. Binary search

Q. LINEAR (OR) SEQUENTIAL SEARCH

Linear search is the basic and simple search algorithm. It is also known as **sequential search** technique. In linear search algorithm, an element or value is searched in a sequential order. Linear Search is applied on the unsorted or unordered list when there are few elements in a list.

Linear Search process starts comparing of **search element** with the first element in the given list. If both are matching then it returns index value otherwise **search element** is compared with next element in the list. Repeat the same process with the next element in the list until the search element compared with the last element. If the last element is also does not match then it returns -1 (i.e., Element is not found in the List). Linear search algorithm finds given elements with O (n) time complexity.

Steps:

Linear search is implemented using following steps

Step 1: Read the search element

Step 2: Compare the search element with the first element in the list

Step 3: If both are matching, then display "Given element found at index position" and terminate the function

Step 4: If both are not matching, then compare the search element with the next element in the list.

Step 5: Repeat the step 3 and 4 until the search element is compared with last element in the list.

Step 6: If the last element is also doesn't match, then display "Element not found" and terminate the function.

Algorithm NonRecLSearch(K, n, item):

Suppose K is an array that contains n elements. Search element is given in the variable 'item'. This function returns an index position 'i' if the element is found; otherwise, return -1.

Step 1: Repeat for i ← 1 to n

```
    If K[i]=item Then Return i
        End If
    End Repeat
```

Step 2: Return -1

Algorithm RecLSearch(K, n,item):

```
Step 1: If n = 0 Then
        Return -1
    ElseIf K[n]=item Then
        Return n
    Else
    Return RecLSearch(K, n-1, item)
    End If
```

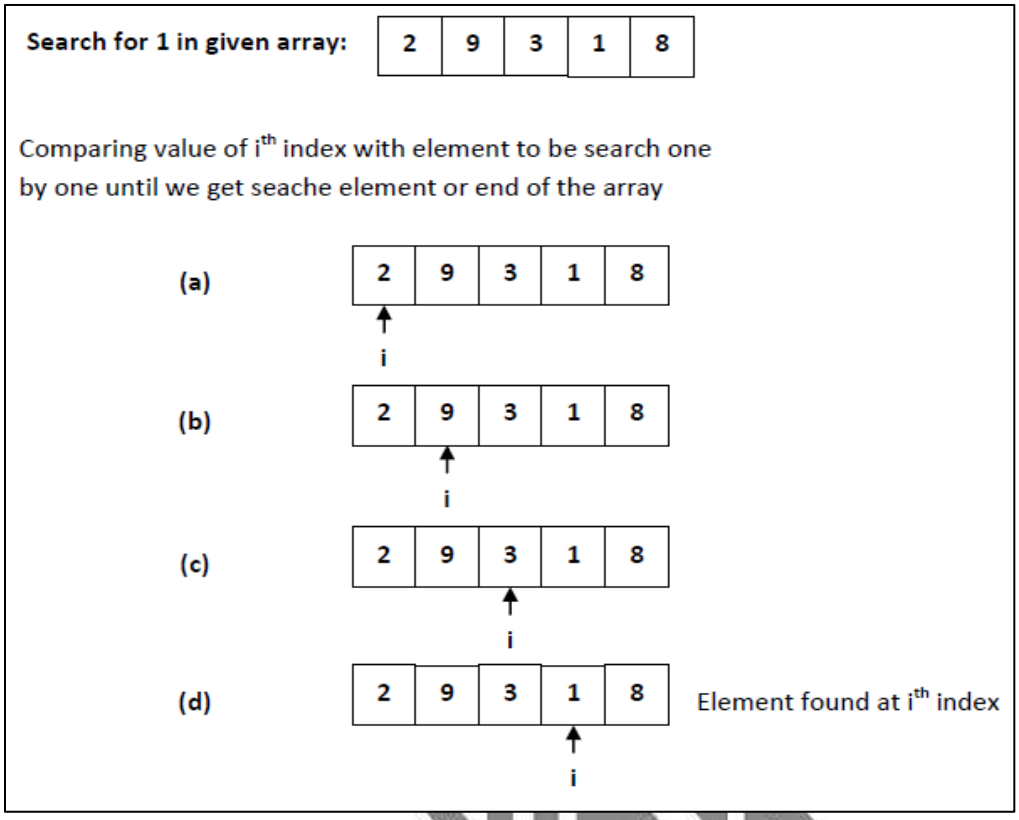
Example: Search an element 19 from the list of elements: 18 35 78 23 19 709

item = 19

i=1	K[1]=item	→	18 = 19 FALSE
i=2	K[2]=item	→	35 = 19 FALSE
i=3	K[3]=item	→	78 = 19 FALSE
i=4	K[4]=item	→	23 = 19 FALSE
i=5	K[5]=item	→	19 = 19 TRUE

ELEMENT FOUND

Example:


Search for 1 in given array:
2 9 3 1 8
Comparing value of i^th index with element to be search one by one until we get seache element or end of the array
(a)
2 9 3 1 8
i
(b)
2 9 3 1 8
i
(c)
2 9 3 1 8
i
(d)
2 9 3 1 8
i
Element found at i^th index

2. BINARY SEARCH

Binary search is quicker than the linear search. It cannot be applied on unsorted data structure. It is useful when there are large numbers of elements in the list. The binary search is based on the approach **divide-and-conquer**.

The binary search starts by testing the data in the middle element (Mid = (Low + High) / 2) of the array. Each time we divide the list into two equal parts and compare the search element with middle element. If both are matched, then it returns the index values. Otherwise, we check whether search element is smaller or larger than the middle element. If the search element is less than to the middle element, then we repeat the same process for left sublist of the middle element. If the search element is larger than to the middle element, then we repeat the same process for right sublist. This process continues until we find the search element in the list or until sublist contains only one element.

Steps:

Binary search is implemented using following steps

Step 1- Read the search element

Step 2- Find the middle element in the sorted list.

Mid = (Low + High)/2 , Where, Low refers to the first index and High refers to last index

Step 3- Compare the search element with the middle element

Step 4- If both are matching then display "Given element found at index position" and terminate the function

Step 5- If both are not matching, then checks whether the search element is smaller or larger than middle element.

Step 6- If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

If ITEM < K[Mid];

Step 7- If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element

If ITEM > K[Mid];

Step 8- Repeat the same process until we find the search element in the list or until sublist contains only one element.

Step 9- If that element also does not match with the search element then display "Element not found" and terminate the function.

Algorithm Non-Recursive BSearch(K, Low, High, ITEM):

```
Step 1:          Repeat while Low ≤ High
                 Mid ← (Low+High)/2
                 If ITEM < K[Mid] Then
                        High ← Mid-1
                 ElseIf ITEM > K[Mid] Then
                        Low ← Mid+1
                 Else
                        Return Mid
                 EndIf
           EndRepeat
Step 2:          Return -1
```

Algorithm Recursive BSearch (K, Low, High, ITEM):

```
Step 1:          If Low ≤ High Then
                 Mid ← (Low+High)/2
                 If ITEM = K[Mid] Then
                        Return Mid
                 ElseIf ITEM < K[Mid] Then
                        Return RBsearch(K, Low, Mid-1, ITEM)
                 Else
                        Return RBsearch(K,Mid+1,High,ITEM)
                 EndIf
           EndIf
Step 2:          Return -1
```

Example 1: Search an element 44 from the list

11 22 30 33 41 44 55

Low = 1 High = 7 Mid = (1+7) / 2 = 4
ITEM = K[Mid] 44 = 33 FALSE
ITEM > K[Mid] 44 > 33 TRUE
Reset Low = 4+1 = 5

Low = 5 High = 7 Mid = (5+7)/2 = 6
ITEM = K[Mid] 44 = 44 TRUE

SUCCESSFUL SEARCH, ITEM FOUND

Example

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 --> (middle element is 19 > 6): Search in left part

-1 5 6 18 **19** 25 46 78 102 114

Step 2 --> (middle element is 5 < 6): Search in Right part

-1 **5** 6 18

Step 3 --> (middle element is 6 == 6): Element Found

6 18

UNIT – 5
CHAPTER – 2: GRAPHS

Introduction to Graphs, Terms Associated with Graphs, Sequential Representation of Graphs, Linked Representation of Graphs, Traversal of Graphs, Spanning Trees, Shortest Path, and Application of Graphs.

Q. What is Graph? Explain terminology of Graphs

GRAPH:

Graph is a non-linear data structure which contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

Generally, a graph G is represented as G = (V, E), where V is set of vertices and E is set of edges.

G=(V,E)

V(G)={v0,v1,v2…..vn-1}

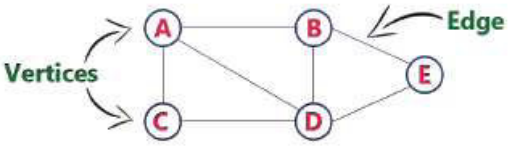
E(G)={e1,e2….en)

Example:

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as

G = (V, E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}



Graph ADTs

The ADT Graph is a graph treated as abstract data type, consisting of

- a set V of items (nodes), and
- a set E of edges, each linking 2 nodes.

Q. GRAPH TERMINOLOGY

1. **Graph:**

   Graph is a non-linear data structure that contains a finite set of vertices (called as nodes) and ordered pair of the elements (called as edge). This can be represented as

   G= (V, E)

   V (G) $= v_1, v_2, v_3 \dots \dots v_{n-1}$

   E (G) $= e_1, e_2, e_3 \dots \dots e_n$

2. **Vertex**

   An individual data element of a graph is called as Vertex. Vertex is also known as node. In the above example graph, A, B, C, D & E are known as vertices.

3. **Edge**

   A connecting link between two vertices is called as Edge. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).

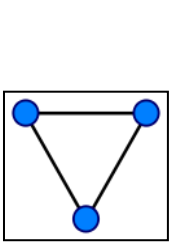
   For example, in above graph, the link between vertices A and B is represented as (A, B). In above example graph, there are 7 edges (i.e., (A, B), (A, C), (A, D), (B, D), (B, E), (C, D), (D, E)).

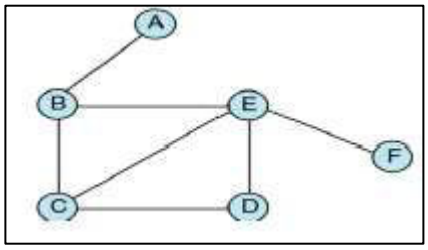
   Edges are three types.

   1. **Undirected Edge** - An undirected edge is a bidirectional edge.
   2. **Directed Edge** - A directed edge is a unidirectional edge.
   3. **Weighted Edge** - A weighted edge is an edge with cost on it.

4. **Undirected Graph**

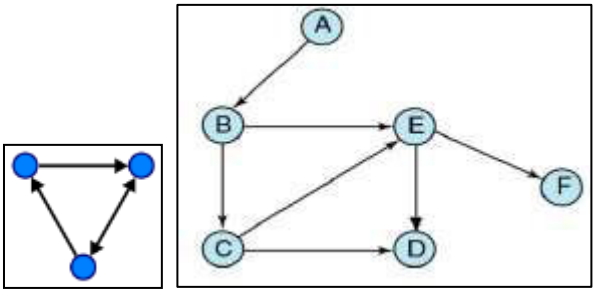
   A graph with only undirected edges is said to be undirected graph.





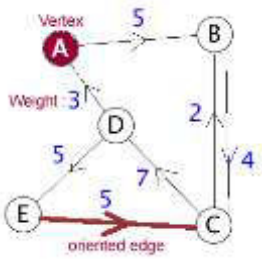
5. **Directed Graph**

   A graph with only directed edges is said to be directed graph. It is also called as Digraph.



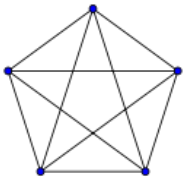
6. **Weighted Graph**

   A graph is said to be weighted graph if every edge and/or vertices in the graph is assigned with some weight or value.



7. **Complete graph**

   A complete graphs have the feature that each pair of vertices has an edge connecting them.



   A complete graph with 5 vertices. Each vertex has an edge to every other vertex.

8. **End vertices or Endpoints**

   The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

9. **Origin**

   If an edge is directed, its first endpoint is said to be origin of it.

10. **Destination**

    If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

11. **Adjacent**

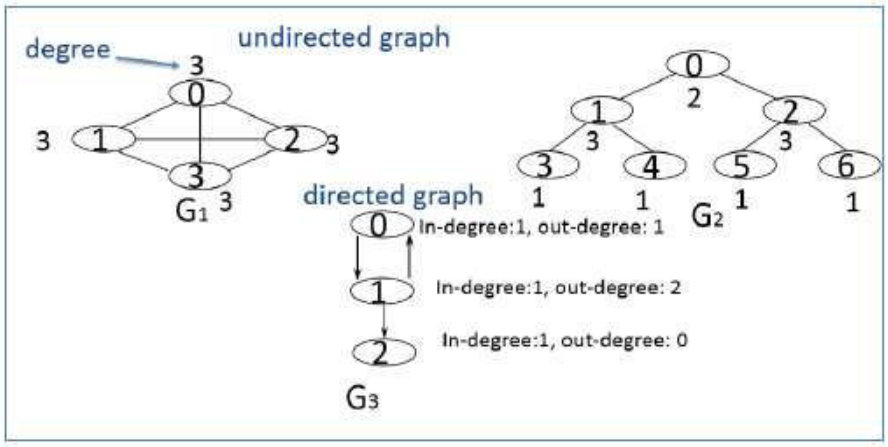
    If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

12. Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

13. Degree

Total number of edges connected to a vertex is said to be degree of that vertex.



14. In-degree

Total number of incoming edges connected to a vertex is said to be in-degree of that vertex.

15. Out degree

Total number of outgoing edges connected to a vertex is said to be out-degree of that vertex.

16. Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

17. Simple Graph (Digraph)

A graph is said to be simple if there are no parallel and self-loop edges.

18. Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

19. Complete Graph (Digraph)

A graph is said to complete (or fully connected or strongly connected) if there is a path from every vertex to every other vertex

20. Isolated Graph

A vertex is isolated if there is no edge connected from any other vertex to the vertex.

Q. REPRESENTATION OF GRAPHS

A graph data structure is represented using following representations

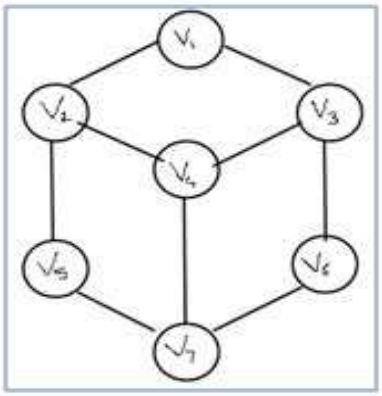
1. Set representation
2. Linked List representation / Adjacency List
3. Matrix representation / Adjacency Matrix / **Sequential Representation**

1. Set Representation:

In this representation, two sets are maintained:

- V, the set of vertices
- E, the set of edges, which is subset of V × V.

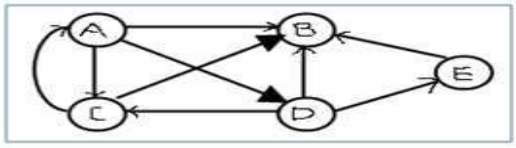
If the graph is weighted, the set E is the ordered collection of three tuples, that is, E= W × V × V, where W is the set of weights.



Undirected Graph:

V(G)= { v1,v2,v3,v4,v5,v6,v7}

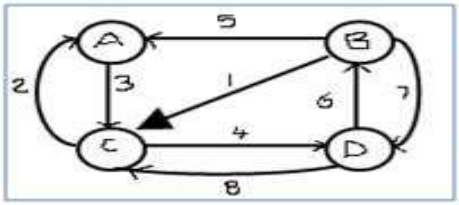
E (G) = {(v1, v2), (v1, v3), (v2, v4), (v2, v5), (v3, v4), (v3, v6), (v4, v7), (v5, v7), (v6, v7)}



Directed Graph:

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

E (G) = {(A, B), (A, C), (C, B), (C, A), (D, A), (D, B), (D, C), (D, E), (E, B)}



Weighted Graph:

V (G) = {A, B, C, D}

E(G)= {(3,A,C),(5,B,A),(1,B,C),(7,B,D), (2,C,A),(4,C,D),(6,D,B),(8,D,C)}

2. Linked list Representation/ Adjacency List

In this representation, we use two types of node structures. In each node, **data** field contains information and **link** field contains the address of the next adjacent node.

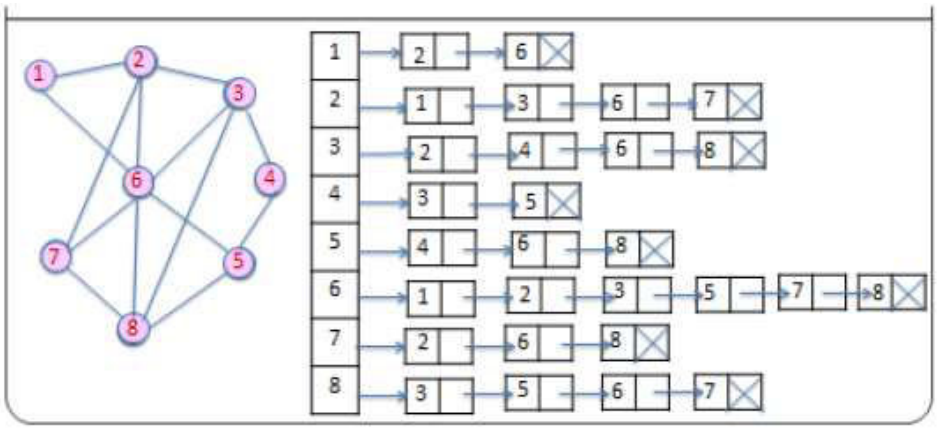
Node_label	Adj_list

Node structure for directed/undirected graph

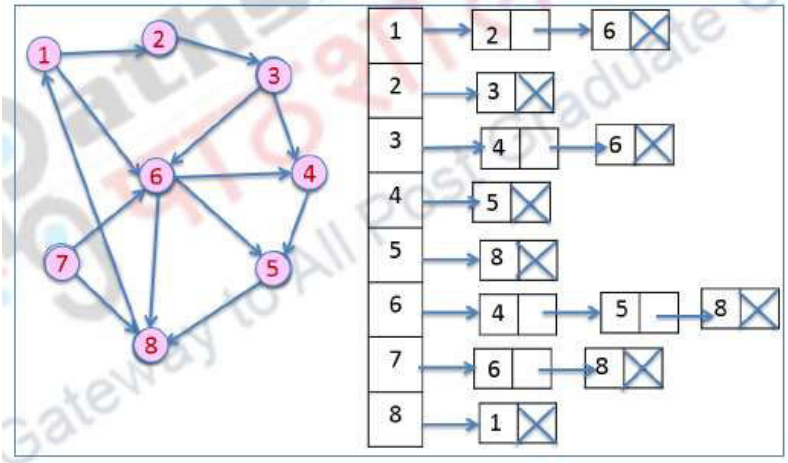
Weight	Node_label	Adj_list

Node structure for weighted graph

In this representation, we store a graph as a linked structure. First we store all the vertices of the graph in a list and then each adjacent vertex will be represented using linked list node.

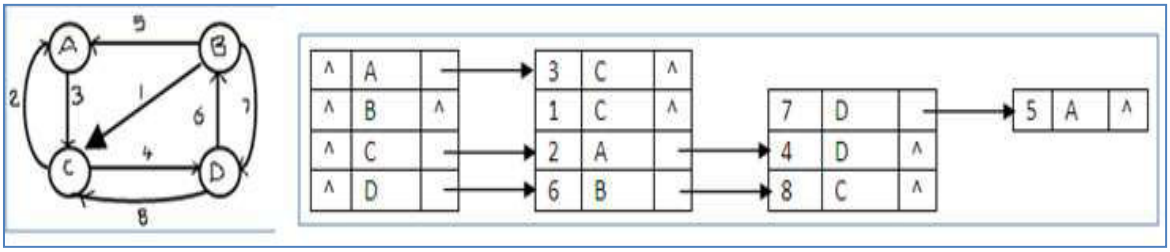


Undirected graph



Directed graph

Weighted graph can be represented using linked list by storing the corresponding weight along with the terminal vertex of the edge.



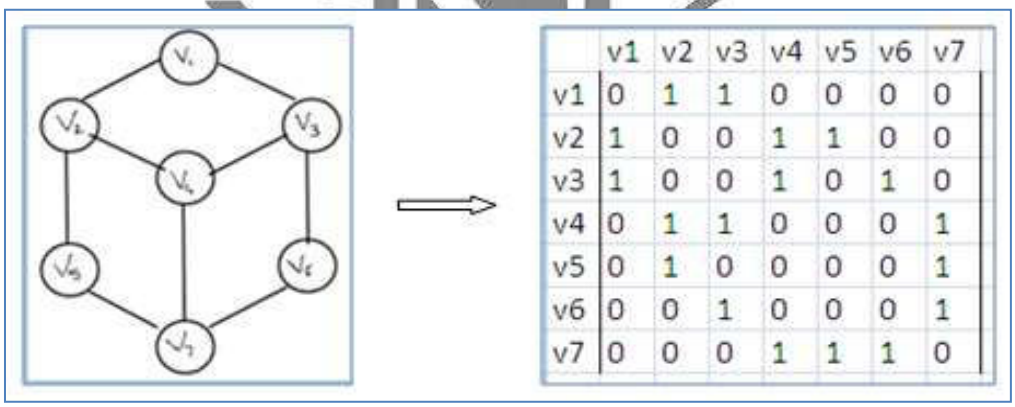
Weighted graph

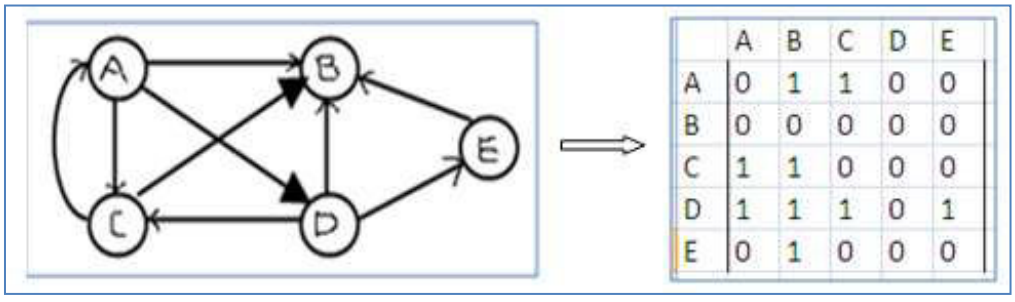
3. Matrix Representation/ Adjacency Matrix / Sequential Representation:

Matrix representation is also called as **Adjacency Matrix representation (or) Sequential Representation**. In this representation, graph is represented by using a matrix of size total number of vertices by total number of vertices. Adjacency matrix is the matrix, which keeps the information of adjacency nodes.

a) The Adjacency Matrix representation of a directed and undirected graph is represented as:

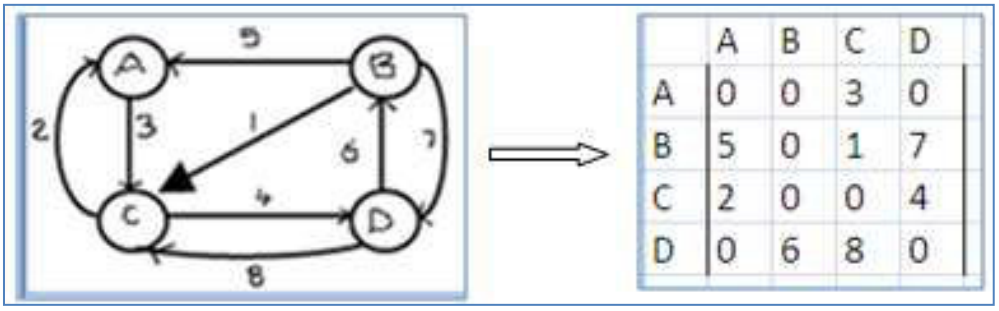
$a_{ij} =$ { 1, if there is an edge from v_i to v_j
0, if there is no edge from v_i to v_j





b) The Adjacency Matrix representation of a weighted graph is represented as:

$a_{ij} =$ { w_{ij} , if there is an edge from v_i to v_j . Where w_{ij} is weight
0, otherwise



	A	B	C	D
A	0	0	3	0
B	5	0	1	7
C	2	0	0	4
D	0	6	8	0

Q. EXPLAIN VARIOUS GRAPH TRAVERSALS WITH EXAMPLES.

GRAPH TRAVERSALS:

Graph traversal or graph search means, the process of visiting each vertex exactly once in a graph. Traversals are classified based on the order in which the vertices are visited.

There are two types of traversal methods:

1. **Breadth First Search (BFS)**
2. **Depth First Search (DFS)**

1. DFS (Depth First Search)

DFS traversal of a graph produces a spanning tree as a final result. Spanning Tree is a graph without any loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

Depth Fist search (DFS) traversal is similar to the inorder traversal of a binary tree. Starting from a given node, the DFS traversal visits all the nodes up to the deepest level and so on.

Steps to implement DFS traversal:

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

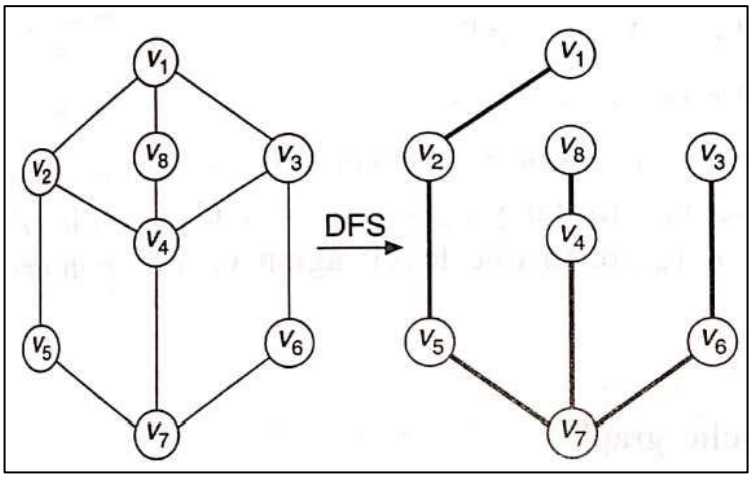
Step 3: Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

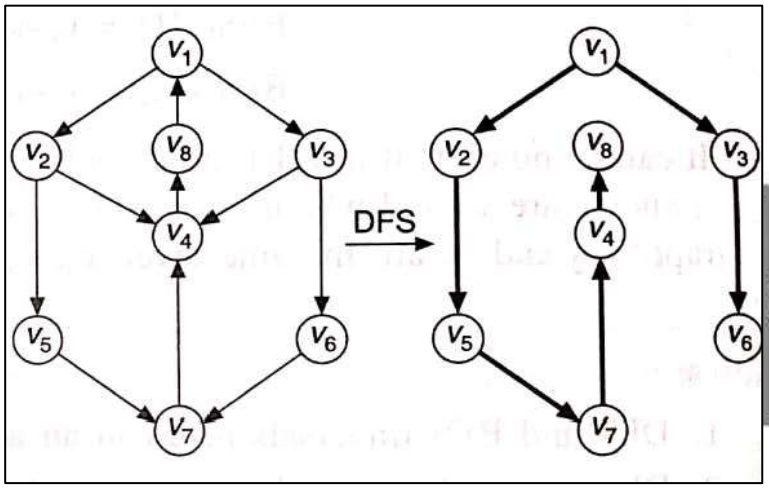
Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph



DFS: v1→v2→v5→v7→v4→v8→v6→v3



DFS: v1→v2→v5→v7→v4→v8→v3→v6

2. BFS (Breadth First Search)

BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph. Breadth Fist Search (BFS) traversal is similar to level by level (pre-order) traversal of a tree.

Steps to implement BFS traversal:

Step 1: Define a Queue of size total number of vertices in the graph.

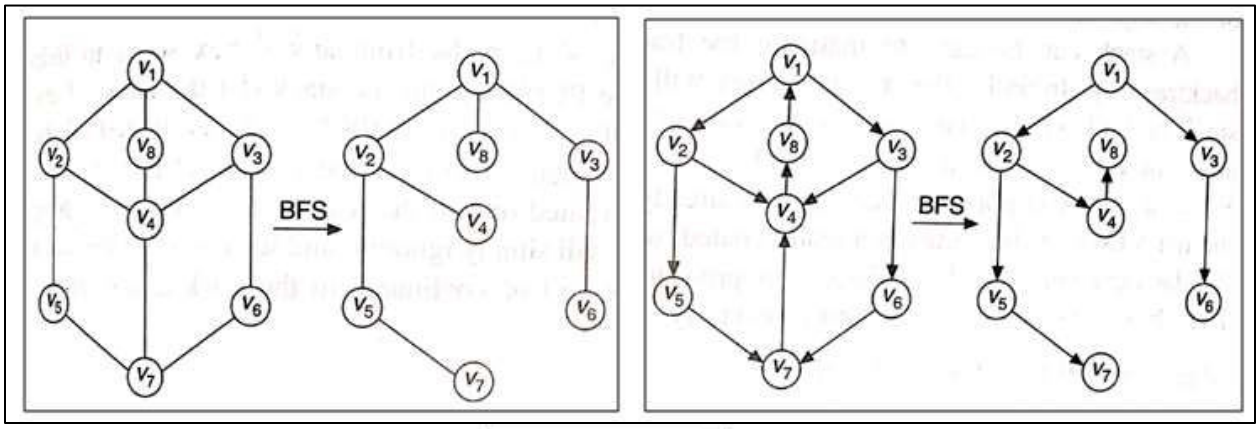
Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

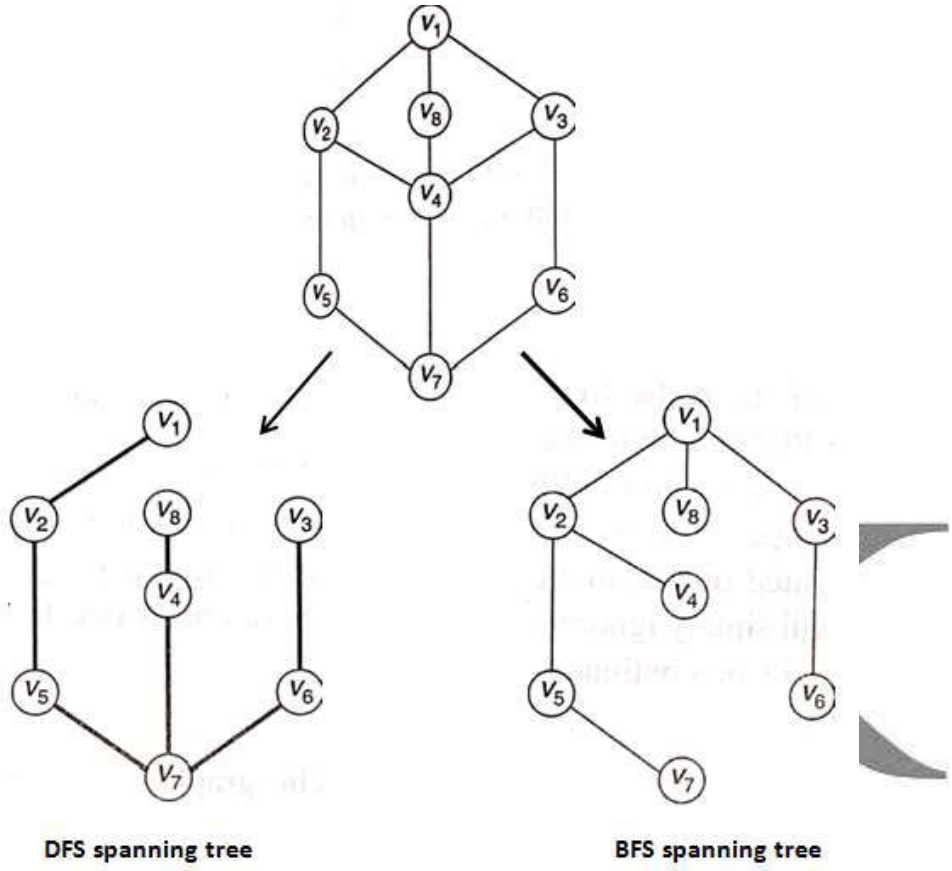
Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph



BFS: v1→v2→v8→v3→v5→v4→v6→v7 BFS: v1→v2→v3→v5→v4→v6→v7→v8

Q. DISCUSS ABOUT SPANNING TREES

A spanning tree of a Graph G is defined as a tree which includes all the vertices of G. A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected. Every connected and undirected Graph G has at least one spanning tree.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where **n** is the number of nodes. In the above addressed example, $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Q. MINIMUM SPANNING TREE

A Spanning tree of a graph is an undirected tree contains only edges to connect all the nodes in the original graph. A graph has many different spanning trees. The spanning tree is related to the weighted graph, where we can find a spanning tree so that the sum of all the weights of all the edges in the tree is minimum.

There are two methods or algorithms to find the minimal spanning tree for a graph

1. Prim's algorithm
2. Kruskal's algorithm

1. Prim's Algorithm

Prim's algorithm constructs the minimum cost spanning tree, edge by edge. It is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. Prim's algorithm shares a similarity with the **shortest path first** algorithms

The Prim's algorithm can easily implement using the adjacency matrix representation of a graph. This algorithm starts by selecting a vertex and then in each stage, we can add an edge to the tree.

Algorithm

1) One node is picked as a root node (u) from the given connected graph.
2) At each stage choose a new vertex v from u, by considering an edge (u, v) with minimum cost among all the edges from u, where u is already in the tree and v is not in the tree.
3) The Prim's algorithm table is constructed with three parameters. They are:
Known – Vertex is added in the tree or not.
d_v -Weight of the shortest arc connecting v to a known vertex.
P_v - last vertex which causes a change in d_v
4) After selecting the vertex v, update rule is applied for each unknown w adjacent to v. The rule is d_w=Min (d_w, $C_{w,v}$) that is if more than one path exist between v to w, then d_w is updated with minimum cost.

Example:

2. Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm in graph that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

Algorithm:

1) Start by selecting the two nodes with the minimal costing link.
2) Select any two nodes with the minimal costing link. Your selection is not bound by any requirement to
select nodes connected to previously selected nodes. Any two nodes we can select, as long as it the minimal cost.
3) Repeat steps 1 and 2 until all nodes have been selected / connected.

Example:

Q. WRITE THE APPLICATIONS OF GRAPHS

Since they are powerful abstractions, graphs can be very important in modeling data.

1. **Social network graphs:**
   Graphs that represent who knows whom, who communicates with whom or other relationships in social structures
2. **Transportation networks:**
   Graph networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps to find the best routes between locations.
3. **Utility graphs:**
   The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them.
4. **Document link graphs:**
   The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge.
5. **Network packet traffic graphs:**
   Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them.
6. **Robot planning:**
   Vertices represent states the robot can be in and the edges the possible transitions between the states.
7. **Neural networks:**
   Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn.
8. **Semantic networks:**
   Vertices represent words or concepts and edges represent the relationships among the words or concepts.
9. **Graphs in compilers:**
   Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes.

C2-P DATA STRUCTURES USING C LAB MANUAL

(For I B.Sc- Sem 2 – 2021 admitted Batch)

1. Write a program to read 'N' numbers of elements into an array and also perform the following operation on an array
    a. Add an element at the begging of an array
    b. Insert an element at given index of array
    c. Update an element using a values and index
    d. Delete an existing element
2. Write a program using stacks to convert a given a. postfix expression to prefix
b. prefix expression to postfix
c. infix expression to postfix
3. Write Programs to implement the Stack operations using an array
4. Write Programs to implement the Stack operations using Liked List.
5. Write Programs to implement the Queue operations using an array.
6. Write Programs to implement the Queue operations using Liked List.
7. Write a program for arithmetic expression evaluation.
8. Write a program for Binary Search Tree Traversals
9. Write a program to implement dequeue using a doubly linked list.
10. Write a program to search an item in a given list using the following Searching Algorithms
    a. Linear Search
    b. Binary Search.
11. Write a program for implementation of the following Sorting Algorithms
    a. Bubble Sort
    b. Insertion Sort
    c. Quick Sort

1. Write a program to read 'N' numbers of elements into an array and also perform the following operation on an array

a. Add an element at the begging of an array

b. Insert an element at given index of array

c. Update an element using a values and index

d. Delete an existing element

Program Statement:

Write a program to read 'N' numbers of elements into an array and also perform the operations on an array

Aim:

To write a program to read 'N' numbers of elements into an array and also perform the operations on an array

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
int a[10], pos, elem;
int n = 0;
void create();
void display();
void insert();
void del();
void main()
{
int choice;
while(1)
{
printf("\n\nMENU");
printf("\n=>1. Create an array of N integers");
printf("\n=>2. Display of array elements");
printf("\n=>3. Insert ELEM at a given POS");
printf("\n=>4. Delete an element at a given POS");
printf("\n=>5. Exit");
printf("\nEnter your choice: ");
scanf("%d", &choice);
switch(choice)
{
case 1: create();
break;
```

```
case 2: display();
break;
case 3: insert();
break;
case 4:del();
break;
case 5:exit(1);
break;
default:printf("\nPlease enter a valid choice:");
}
}
}

void create()
{
int i;
printf("\nEnter the number of elements: ");
scanf("%d", &n);
printf("\nEnter the elements: ");
for(i=0; i<n; i++)
{
scanf("%d", &a[i]);
}
}

void display()
{
int i;
if(n == 0)
{
printf("\nNo elements to display");
return;
}
printf("\nArray elements are: ");
for(i=0; i<n;i++)
printf("%d\t ", a[i]);
}

void insert()
{
```

```
int i;
if(n == 5)
{
printf("\nArray is full. Insertion is not possible");
return;
}
do
{
printf("\nEnter a valid position where element to be inserted: ");
scanf("%d", &pos);
}while(pos > n);
printf("\nEnter the value to be inserted: ");
scanf("%d", &elem);
for(i=n-1; i>=pos ; i--)
{
a[i+1] = a[i];
}
a[pos] = elem;
n = n+1;
display();
}

void del()
{
int i;
if(n == 0)
{
printf("\nArray is empty and no elements to delete");
return;
}
do
{
printf("\nEnter a valid position from where element to be deleted: ");
scanf("%d", &pos);
}while(pos>=n);
elem = a[pos];
printf("\nDeleted element is : %d \n", elem);
for( i = pos; i< n-1; i++)
{
a[i] = a[i+1];
```

```
}
n = n-1;
display();
}
```

2. Write Programs to implement the Single Linked list using an array

Program Statement:

Write Programs to implement the Single Linked list using an array

Aim:

To Write Programs to implement the Single Linked list using an array

Source Code:

```
#include<stdio.h>
#include<conio.h>
#define TRUE 1
#define SIZE 10
struct link
{
int info;
int next;
};
struct link node[SIZE];
int Getnode();
void Createlist();
void Freenode(int);
void Display();
void Insert(int,int);
void Delete(int);
int p, avail=0;
void main()
{
int ch=1,i,n,x;
clrscr();
/*Creation of available list*/
for(i=0;i<SIZE-1;i++)
node[i].next=i+1;
node[SIZE-1].next=-1;
printf("\n Create a List:");
Createlist();
```

```
while(ch!=4)
{
printf("\n1-DISPLAY");
printf("\n2-INSERT");
printf("\n3-DELETE");
printf("\n4-QUIT");
printf("\n Enter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1 :
Display();
break;
case 2:
printf("\n Node insertion:after which node:");
scanf("%d",&n);
p=n;
printf("\n Enter the item for insertion:");
scanf("%d",&x);
Insert(p,x);
break;
case 3:
printf("\n Enter the node after which the node will be deleted:");
scanf("%d",&n);
p=n;
Delete(p);
break;
case 4:
break;
default:
printf("\n Wrong choice!Try again:");
}
}
}
int Getnode()
{
if (avail==-1)
{
printf("\n Overflow:");
exit(0);
```

```
}
p=avail;
avail=node[avail].next;
return p;
}
void Freenode(int q)
{
node[q].next=avail;
avail=q;
return;
}
void Createlist()
{
int x;
char c;
p=Getnode();
printf("\n Enter an item to be inserted:");
scanf("%d", &x);
node[p].info=x ;
node[p].next=-1;
while(TRUE)
{
printf("\n Enter the choice(y/n):");
fflush(stdin);
c=getchar();
if(c=='y'||c=='Y')
{
printf("\n Enter an item to be inserted:");
scanf("%d",&x);
Insert(p,x);
node[p].next= -1;
}
else
return;
}
}
void Display()
{
p=0;
while(node[p].next!=-1)
```

```
{
printf("\n%d\t%d\t%d:",p,node[p].info,node[p].next);
p=node[p].next;
}
printf("\n%d\t%d\t%d:",p,node[p].info,node[p].next);
}
void Insert(int r,int x)
{
int q;
if(r==-1)
{
printf("\n void insertion:");
return;
}
q=Getnode();
node[q].info=x;
node[q].next=node[r].next;
node[r].next=q;
return;
}
void Delete(int r)
{
int q;
if(r==-1||node[r].next==-1)
{
printf("\n void deletion:");
return;
}
q=node[r].next;
node[r].next=node[q].next;
Freenode(q);
return;
}
```

3. Write Programs to implement the Double linked list operations

Program Statement:

Write Programs to implement the Double linked list operations

Aim:

To Write Programs to implement the Double linked list operations

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
struct node *prev;
struct node *next;
int data;
};
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
while(choice != 9)
{
printf("\n*********Main Menu*********\n");
printf("\n Choose one option from the following list ...\n");
printf("\n==========================\n");
printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete
from
5.Delete from last\n6.Delete the node after the given
data\n7.Search\n8.Show\n9.Exit\n");
printf("\nEnter your choice?\n");
scanf("\n%d",&choice);
switch(choice)
{
```

```
case 1:
insertion_beginning();
break;
case 2:
insertion_last();
break;
case 3:
insertion_specified();
break;
case 4:
deletion_beginning();
break;
case 5:
deletion_last();
break;
case 6:
deletion_specified();
break;
case 7:
search();
break;
case 8:
display();
break;
case 9:
exit(0);
break;
default:
printf("Please enter valid choice..");
}
}
}
void insertion_beginning()
{
struct node *ptr;
int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW");
```

```
}
else
{
printf("\nEnter Item value");
scanf("%d",&item);
if(head==NULL)
{
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
head=ptr;
}
else
{
ptr->data=item;
ptr->prev=NULL;
ptr->next = head;
head->prev=ptr;
head=ptr;
}
printf("\nNode inserted\n");
}
}
void insertion_last()
{
struct node *ptr,*temp;
int item;
ptr = (struct node *) malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW");
}
else
{
printf("\nEnter value");
scanf("%d",&item);
ptr->data=item;
if(head == NULL)
{
ptr->next = NULL;
```

```
ptr->prev = NULL;
head = ptr;
}
else
{
temp = head;
while(temp->next!=NULL)
{
temp = temp->next;
}
temp->next = ptr;
ptr ->prev=temp;
ptr->next = NULL;
}
}
printf("\nnode inserted\n");
}
void insertion_specified()
{
struct node *ptr,*temp;
int item,loc,i;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\n OVERFLOW");
}
else
{
temp=head;
printf("Enter the location");
scanf("%d",&loc);
for(i=0;i<loc;i++)
{
temp = temp->next;
if(temp == NULL)
{
printf("\n There are less than %d elements", loc);
return;
}
}
```

```
printf("Enter value");
scanf("%d",&item);
ptr->data = item;
ptr->next = temp->next;
ptr -> prev = temp;
temp->next = ptr;
temp->next->prev=ptr;
printf("\nnode inserted\n");
}
}
void deletion_beginning()
{
struct node *ptr;
if(head == NULL)
{
printf("\n UNDERFLOW");
}
else if(head->next == NULL)
{
head = NULL;
free(head);
printf("\n node deleted\n");
}
else
{
ptr = head;
head = head -> next;
head -> prev = NULL;
free(ptr);
printf("\n node deleted\n");
}
}
void deletion_last()
{
struct node *ptr;
if(head == NULL)
{
printf("\n UNDERFLOW");
}
else if(head->next == NULL)
```

```
{
head = NULL;
free(head);
printf("\n node deleted\n");
}
else
{
ptr = head;
if(ptr->next != NULL)
{
ptr = ptr -> next;
}
ptr -> prev -> next = NULL;
free(ptr);
printf("\n node deleted\n");
}
}
void deletion_specified()
{
struct node *ptr, *temp;
int val;
printf("\n Enter the data after which the node is to be deleted : ");
scanf("%d", &val);
ptr = head;
while(ptr -> data != val)
ptr = ptr -> next;
if(ptr -> next == NULL)
{
printf("\nCan't delete\n");
}
else if(ptr -> next -> next == NULL)
{
ptr ->next = NULL;
}
else
{
temp = ptr -> next;
ptr -> next = temp -> next;
temp -> next -> prev = ptr;
free(temp);
```

```
printf("\n node deleted\n");
}
}
void display()
{
struct node *ptr;
printf("\n printing values...\n");
ptr = head;
while(ptr != NULL)
{
printf("%d\n",ptr->data);
}
}
void search()
{
struct node *ptr;
int item,i=0,flag;
ptr = head;
if(ptr == NULL)
{
printf("\n Empty List\n");
}
else
{
printf("\n Enter item which you want to search?\n");
scanf("%d",&item);
while (ptr!=NULL)
{
if(ptr->data == item)
{
printf("\n item found at location %d ",i+1);
flag=0;
break;
}
else
{
flag=1;
}
i++;
ptr = ptr -> next;
```

```
}
if(flag==1)
{
printf("\n Item not found\n");
}
}
```

4. Write Programs to implement the Stack operations using an array

Program Statement:

Write Programs to implement the Stack operations using an array

Aim:

To Write Programs to implement the Stack operations using an array

Source Code:

```
//Write a program to perform Push, Pop, and Peek operations on a stack.
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3
int st[MAX], top=-1;
void push(int st[],int val);
int pop(int st[]);
int peek(int st[]);
void display(int st[]);
int main()
{
int val, option;
do
{
printf("\n *****MAIN MENU*****");
printf("\n 1. PUSH");
printf("\n 2. POP");
printf("\n 3. PEEK");
printf("\n 4. DISPLAY");
printf("\n 5. EXIT");
printf("\n Enter your option: ");
scanf("%d", &option);
switch(option)
```

```
{
case 1:
printf("\n Enter the number to be pushed on stack: ");
scanf("%d", &val);
push(st, val);
break;
case 2:
val = pop(st);
if(val != -1)
printf("\n The value deleted from stack is: %d", val);
break;
case 3:
val = peek(st);
if(val != -1)
printf("\n The value stored at top of stack is: %d", val);
break;
case 4:
display(st);
break;
}
}
while(option != 5);
return 0;
}
void push(int st[],int val)
{
if(top == MAX-1)
{
printf("\n STACK OVERFLOW");
}
else
{
top++;
st[top] = val;
}
}
int pop(int st[])
{
int val;
if(top == -1)
```

```
{
printf("\n STACK UNDERFLOW");
return -1;
}
else
{
val = st[top];
top--;
return val;
}
}
void display(int st[])
{
int i;
if(top == -1)
printf("\n STACK IS EMPTY");
else
{
for(i=top;i>=0;i--)
printf("\n %d",st[i]);
printf("\n");
// Added for formatting purposes
}
}
int peek(int st[])
{
if(top == -1)
{
printf("\n STACK IS EMPTY");
return -1;
}
else
return (st[top]);
}
```

5. Write Programs to implement the Stack operations using Liked List.

Program Statement:

Write Programs to implement the Stack operations using Liked List.

Aim:

To Write Programs to implement the Stack operations using Liked List.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
};
struct node *head;
void main ()
{
int choice=0;
printf("\n*********Stack operations using linked list*********\n");
printf("\n----------------------------------------------\n");
while(choice != 4)
{
printf("\n\nChose one from the below options...\n");
printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
printf("\n Enter your choice \n");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
push();
break;
}
case 2:
{
pop();
break;
}
```

```
case 3:
{
display();
break;
}
case 4:
{
printf("Exiting....");
break;
}
default:
{
printf("Please Enter valid choice ");
}
};
}
}
void push ()
{
int val;
struct node *ptr = (struct node*)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("not able to push the element");
}
else
{
printf("Enter the value");
scanf("%d",&val);
if(head==NULL)
{
ptr->val = val;
ptr -> next = NULL;
head=ptr;
}
else
{
ptr->val = val;
ptr->next = head;
head=ptr;
```

```
}
printf("Item pushed");
}
}
void pop()
{
int item;
struct node *ptr;
if (head == NULL)
{
printf("Underflow");
}
else
{
item = head->val;
ptr = head;
head = head->next;
free(ptr);
printf("Item popped");
}
}
void display()
{
int i;
struct node *ptr;
ptr=head;
if(ptr == NULL)
{
printf("Stack is empty\n");
}
else
{
printf("Printing Stack elements \n");
while(ptr!=NULL)
{
printf("%d\n",ptr->val);
ptr = ptr->next;
}
}
}
```

6. Write Programs to implement the Queue operations using an array

Program Statement:

Write Programs to implement the Queue operations using an array

Aim:

To Write Programs to implement the Queue operations using an array

Source Code:

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int queue[MAX];
int front = -1;
int rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()
{
int option, val;
do
{
printf("\n\n ***** MAIN MENU *****");
printf("\n 1. Insert an element");
printf("\n 2. Delete an element");
printf("\n 3. Peek");
printf("\n 4. Display the queue");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option)
{
case 1:
insert();
break;
case 2:
val = delete_element();
if(val != -1)
printf("\n The number deleted is : %d", val);
```

```
break;
case 3:
val = peek();
if(val != -1)
printf("\n The first value in queue is : %d", val);
break;
case 4:
display();
break;
}
}
while(option != 5);
getch();
return 0;
}

void insert()
{
int num;
printf("\n Enter the number to be inserted in the queue : ");
scanf("%d", &num);
if(rear == MAX-1)
printf("\n OVERFLOW");
else if(front == -1 && rear == -1)
front = rear = 0;
else
rear++;
queue[rear] = num;
}

int delete_element()
{
int val;
if(front == -1 || front>rear)
{
printf("\n UNDERFLOW");
return -1;
}
else
{
```

```
val = queue[front];
front++;
if(front > rear)
front = rear = -1;
return val;
}
}

int peek()
{
if(front==-1 || front>rear)
{
printf("\n QUEUE IS EMPTY");
return -1;
}
else
{
return queue[front];
}
}

void display()
{
int i;
printf("\n");
if(front == -1 || front > rear)
printf("\n QUEUE IS EMPTY");
else
{
for(i = front;i <= rear;i++)
printf("\t %d", queue[i]);
}
}
```

7. Write a Program for Implementation of circular queue using array

Program Statement:

Write a Program for Implementation of circular queue using array

Aim:

To Write a Program for Implementation of circular queue using array

Source Code:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXQ 100
int front=-1,rear=-1;
int items[MAXQ];
int Isempty();
int Isfull();
void Insert(int);
int Delete();
void Display();
void main()
{
int x;
char ch='1';
clrscr();
while(ch!='4')
{
printf("\n 1-INSERT");
printf("\n 2-DELETE");
printf("\n 3-DISPLAY");
printf("\n 4-QUIT");
printf("\n Enter your choice:");
fflush(stdin);
ch=getchar();
switch(ch)
{
case '1':
printf("\n Enter the nos of element to be inserted:");
scanf("%d",&x);
Insert(x);
break;
```

```
case '2':
x=Delete();
printf("\n Deleted element is %d\n:",x);
break;
case '3':
Display();
break;
case '4':
break;
default:
printf("\n Wrong choice!Try again:");
}
}
getch();
}
int Isempty()
{
if(front==-1)
return 1;
else
return 0;
}
int Isfull()
{
if(front==(rear+1)%MAXQ)
return 1;
else
return 0;
}
void Insert(int x)
{
if(Isfull())
{
printf("\n Queue full");
return;
}
if (front==-1)
{
front=0;
rear=0;
```

```
}
else
rear=(rear+1)%MAXQ;
items[rear]=x;
}
int Delete()
{
int x;
if(Isempty())
{
printf("\n Queue is empty");
exit(0);
}
x=items[front];
if (front==rear)
{
front=-1;
rear=-1;
}
else
front=(front+1)%MAXQ;
return x;
}
void Display()
{
int i,n;
if(Isempty())
{
printf("\n Queue is empty");
return;
}
printf("\n Elements in the Queue are :\n");
if(front<=rear)
{
for(i=front;i<=rear;i++)
printf("%d\n",items[i]);
}
else
{
for(i=front;i<=MAXQ-1;i++)
```

```
printf("%d\n",items[i]);
for(i=0;i<=rear;i++)
printf("%d\n",items[i]);
}
}
```

8. Write a program for Implementation of binary search tree using array

Program Statement:

Write a program for Implementation of binary search tree using array

Aim:

To Write a program for Implementation of binary search tree using array

Source Code:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define TRUE 1
#define TREENODES 100
#define FALSE 0
struct tree
{
int info;
int used;
};
struct tree node[TREENODES];
void Createtree();
void Insert(int);
void Display();
void Setleft(int,int);
void Setright(int,int);
void main()
{
int x;
char ch='1';
clrscr();
printf("\n Enter root node value:");
scanf("%d", &x);
Createtree(x);
while(ch!='3')
```

```
{
printf("\n1-INSERT");
printf("\n2-DISPLAY");
printf("\n3-QUIT");
printf("\n Enter your choice:");
fflush(stdin);
ch=getchar();
switch(ch)
{
case '1' :
printf("\n Enter the element to be inserted:");
scanf("%d",&x);
Insert(x);
break;
case '2':
Display();
break;
case '3':
break;
default:
printf("\n Wrong choice!Try again:");
}
}
}
void Createtree(int x)
{
int i;
node[0].info=x;
node[0].used=TRUE;
for(i=1;i<TREENODES;i++)
node[i].used=FALSE;
}
void Insert(int x)
{
int p,q;
p=q=0;
while(q<TREENODES && node[q].used && x!=node[p].info)
{
p=q;
if(x<node[p].info)
```

```
q=2*p+1;
else
q=2*p+2;
}
if(x==node[p].info)
printf("\n %d is a duplicate number\n",x);
else
if(x<node[p].info)
Setleft(p,x);
else
Setright(p,x);
}
void Setleft(int pos,int x)
{
int q;
q=2*pos+1;
if(q>TREENODES)
printf("\n Array overflow.");
else
if(node[q].used==TRUE)
printf("\n Invalid insertion.");
else
{
node[q].info=x;
node[q].used=TRUE;
}
}
void Setright(int pos,int x)
{
int q;
q=2*pos+2;
if(q>TREENODES)
printf("\n Array overflow.");
else
if(node[q].used==TRUE)
printf("\n Invalid insertion.\n");
else
{
node[q].info=x;
node[q].used=TRUE;
```

```
}
}
void Display()
{
int i;
for(i=0;i<TREENODES;i++)
if(node[i].used==TRUE)
printf("%d ",node[i].info);
printf("\n");
}
```

9. Write a program for Binary Search Tree Traversals

Program Statement:

Write a program for Binary Search Tree Traversals

Aim:

To Write a program for Binary Search Tree Traversals

Source Code:

```
//Binary tree traversal:
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
struct node *left;
int data;
struct node *right;
};
void main()
{
void insert(struct node **,int);
void inorder(struct node *);
void postorder(struct node *);
void preorder(struct node *);
struct node *ptr;
int no,i,num;
ptr = NULL;
ptr->data=NULL;
```

```
clrscr();
printf("\nProgram for Tree Traversal\n");
printf("Enter the number of nodes to add to the tree.<BR>\n");
scanf("%d",&no);
for(i=0;i<no;i++)
{
printf("Enter the item\n");
scanf("%d",&num);
insert(&ptr,num);
}
//getch();
printf("\nINORDER TRAVERSAL\n");
inorder(ptr);
printf("\nPREORDER TRAVERSAL\n");
preorder(ptr);
printf("\nPOSTORDER TRAVERSAL\n");
postorder(ptr);
getch();
}
void insert(struct node **p,int num)
{
if((*p)==NULL)
{
printf("Leaf node created.");
(*p)=malloc(sizeof(struct node));
(*p)->left = NULL;
(*p)->right = NULL;
(*p)->data = num;
return;
}
else
{
if(num==(*p)->data)
{
printf("\nREPEATED ENTRY ERROR VALUE REJECTED\n");
return;
}
if(num<(*p)->data)
{
printf("\nDirected to left link.\n");
```

```
insert(&((*p)->left),num);
}
else
{
printf("Directed to right link.\n");
insert(&((*p)->right),num);
}
}
return;
}
void inorder(struct node *p)
{
if(p!=NULL)
{
inorder(p->left);
printf("\nData :%d",p->data);
inorder(p->right);
}
else
return;
}
void preorder(struct node *p)
{
if(p!=NULL)
{
printf("\nData :%d",p->data);
preorder(p->left);
preorder(p->right);
}
else
return;
}
void postorder(struct node *p)
{
if(p!=NULL)
{
postorder(p->left);
postorder(p->right);
printf("\nData :%d",p->data);
}
```

```
else
return;
}
```

10. Write a program To Search an element using sequential search

Program Statement:

Write a program To Search an element using sequential search

Aim:

To Write a program To Search an element using sequential search

Source Code:

```
#include<stdio.h>
#include<conio.h>
int Sequentialsearch(int[],int,int);
void main()
{
int x[20],i,n,p,key;
clrscr();
printf("\n Enter the no of element:");
scanf("%d",&n);
printf("\n Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&x[i]);
printf("\n Enter the element to be search:");
scanf("%d",&key);
p=Sequentialsearch(x,n,key);
if(p==-1)
printf("\n The searchis unsuccessful:\n");
else
printf("\n%d is found at location %d",key,p);
getch();
}
int Sequentialsearch(int a[],int n ,int k)
{
int i;
for(i=0;i<n;i++)
{
if(k==a[i])
return(i);
```

```
}
return(-1);
}
```

11. Write a program To Search an element using binary search.

Program Statement:

Write a program To Search an element using binary search.

Aim:

To Write a program To Search an element using binary search.

Source Code:

```
#include<stdio.h>
#include<conio.h>
int Binarysearch(int[],int,int);
void main()
{
int x[20],i,n,p,key;
clrscr();
printf("\n Enter the no of element:");
scanf("%d",&n);
printf("\n Enter %d elements in assending order:",n);
for(i=0;i<n;i++)
scanf("%d",&x[i]);
printf("\n Enter the element to be search:");
scanf("%d",&key);
p=Binarysearch(x,n,key);
if(p==-1)
printf("\n The searchis unsuccessful:\n");
else
printf("\n%d is found at location %d",key,p);
getch();
}
int Binarysearch(int a[],int n ,int k)
{
int lo,hi,mid;
lo=0;
hi=n-1;
while(lo<=hi)
```

```
{
mid=(lo+hi)/2;
if(k==a[mid])
return(mid);
if(k<a[mid])
hi=mid-1;
else
lo=mid+1;
}
return(-1);
}
```

12. Write a program to Arrange the list of numbers in ascending order using Bubble Sort.

Program Statement:

Write a program to Arrange the list of numbers in ascending order using Bubble Sort.

Aim:

To Write a program to Arrange the list of numbers in ascending order using Bubble Sort.

Source Code:

```
#include<stdio.h>
#include<conio.h>
void Bubblesort(int[],int);
void main()
{
int x[20],i,n;
clrscr();
printf("\n Enter the no of element to be sorted:");
scanf("%d",&n);
printf("\n Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&x[i]);
Bubblesort(x,n);
printf("\n The sorted array is:\n");
for(i=0;i<n;i++)
printf("%4d",x[i]);
getch();
}
void Bubblesort(int a[],int n)
```

```
{
int temp,pass,i;
for(pass=0;pass<n-1;pass++)
{
for(i=0;i<n-pass-1;i++)
{
if(a[i]>a[i+1])
{
temp=a[i];
a[i]=a[i+1];
a[i+1]=temp;
}
}
}
}
```

13. Write a program to Arrange the list of numbers in ascending order using Insertion Sort.

Program Statement:

Write a program to Arrange the list of numbers in ascending order using Insertion Sort.

Aim:

To Write a program to Arrange the list of numbers in ascending order using Insertion Sort.

Source Code:

```
#include<stdio.h>
#include<conio.h>
void Insertionsort(int[],int);
void main()
{
int x[20],i,n;
clrscr();
printf("\n Enter the no of element to be sorted:");
scanf("%d",&n);
printf("\n Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&x[i]);
Insertionsort(x,n);
printf("\n The sorted array is:\n");
for(i=0;i<n;i++)
printf("%4d",x[i]);
```

```
getch();
}
void Insertionsort(int a[],int n)
{
int i,j,key;
for(j=1;j<n;j++)
{
key=a[j];
i=j-1;
while((i>-1)&&(a[i]>key))
{
a[i+1]=a[i];
i=i-1;
}
a[i+1]=key;
}
}
```

14. Write a program to Arrange the list of numbers in ascending order using Quick Sort.

Program Statement:

Write a program to Arrange the list of numbers in ascending order using Quick Sort.

Aim:

To Write a program to Arrange the list of numbers in ascending order using Quick Sort.

Source Code:

```
#include<stdio.h>
#include<conio.h>
void Quicksort(int[],int,int);
int partition(int[],int,int);
void main()
{
int x[20],i,n;
clrscr();
printf("\n Enter the no of element to be sorted:");
scanf("%d",&n);
printf("\n Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&x[i]);
Quicksort(x,0,n-1);
printf("\n The sorted array is:\n");
for(i=0;i<n;i++)
```

```
printf("%4d",x[i]);
getch();
}
void Quicksort(int a[],int p,int r)
{
int q;
if(p<r)
{
q=Partition(a,p,r);
Quicksort(a,p,q);
Quicksort(a,q+1,r);
}
}
int Partition(int a[], int p,int r)
{
int k,i,j,temp;
k=a[p];
i=p-1;
j=r+1;
while(1)
{
do
{
j=j-1;
}while(a[j]>k);
do
{
i=i+1;
}while(a[i]<k);
if(i<j)
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
else
return(j);
}
}
```