**SRI RACHAPUDY NAGABHUSHANAM DEGREE & PG COLLEGE, BADVEL**

**FIRST YEAR – II SEM**

**B.SC (MPCs & MSCs)**

**DATA STRUCTURES**

**MATERIAL**



**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.

# 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.**

# 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 22 + 1x21 + 0x 20 = 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.

EX:- 11011001

When applying the one’s complement for above code its change 00100110

In **two’s complement** method,there are two steps. They are:

1. First chage the given code into one’s complement.
2. 1 is added to one’s complement representation of the negative number.

For example,

0 0 1 0 0 1 1 0

step-1: 1 1 0 1 1 0 0 1

step-2: + 1

1 1 0 1 1 0 1 0 .

# 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)exponent

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

# Character Representation

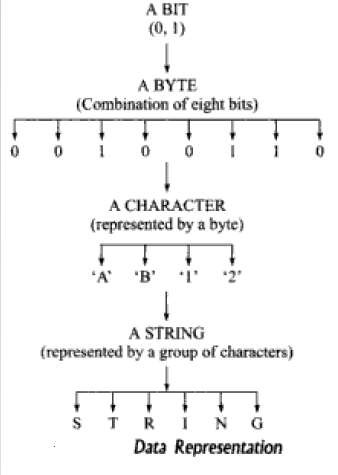
* + In computer, character representation is used to represent the characters
  + There are different codes available to store data in character form such as BCD, EBCDIC and ASCII. (the ASCII codes for A-z are 65-90,a-z are 97-122)

# For example

* + 01000001 is used to represent the character ‘A’
  + 01100001 is used to represent character ‘a’.

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





# 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

# Abstract data types (ADT):

Abstract data type is a theoretical construct that consist of data as well as the operations to be performed on data to implementation.

(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 List‖ 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

1. Declaration of data
2. 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.

# 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;

};

# 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:**

**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‘, "lakshmi"

# 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 is a technique of storing and organizing the data in such a way that the data can be utilized in an efficient manner. A data structure is designed in such a way that it can work with various algorithms.

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.

**ATOMIC TYPE**

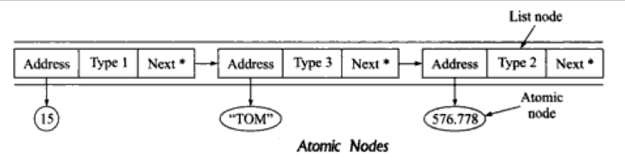
Generally, a data structure is represented by a memory block, which has two parts:

* 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.



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.

# 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.



# 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

# Conceptual level

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

# Algorithmic or Data structure Level

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

# Programming or Implementation Level

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

# 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.

**Q) What is Data Structure?**

A data structure is a technique of storing and organizing the data in such a way that the data can be utilized in an efficient manner. A data structure is designed in such a way that it can work with various algorithms.

**Advantages of Data Structures**

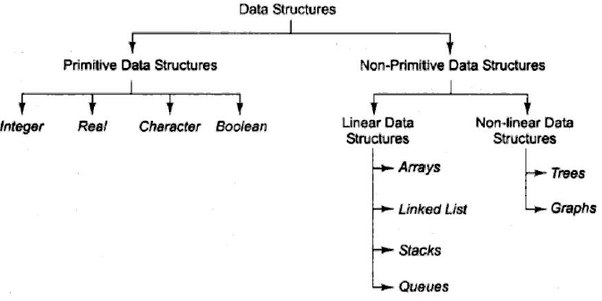
1. **Efficiency**: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.
2. **Reusability**: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.
3. **Abstraction**: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

**Operations on data structure**

1. **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.
2. **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.
3. **Deletion:** The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location. If we try to delete an element from an empty data structure, then underflow occurs.
4. **Searching**: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.
5. **Sorting**: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.
6. **Merging**: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

**Q) Primitive and Non-primitive Data Structures?**

A data structure is a technique of storing and organizing the data in such a way that the data can be utilized in an efficient manner. A data structure is designed in such a way that it can work with various algorithms. Data Structures are classified into two types:



* 1. **Primitive Data Structures:**

Primitive data structures are data structures where it can store single value in one specific location. Primitive data structures are the predefined data types that are supported by any programming language. The primitive data structures contain values that are provided by the programmer.

There are four main data structures in every language.

**Integer** – The integers are used to represent the numeric data. The integer generally stores whole numbers which can be positive and negative. Long can be used in cases where the range of integer data type is not large enough.

**Float** – The float data type is used to represent the fractional numbers or numbers with decimal figures in the languages. Double can be used to increase the range and precision of decimal figures that float data type holds.

**Boolean** – The Boolean data type can only take up to two values that are TRUE or FALSE. Mostly, the boolean values are used for conditional testing.

**Character** – The character data type is used to store single word

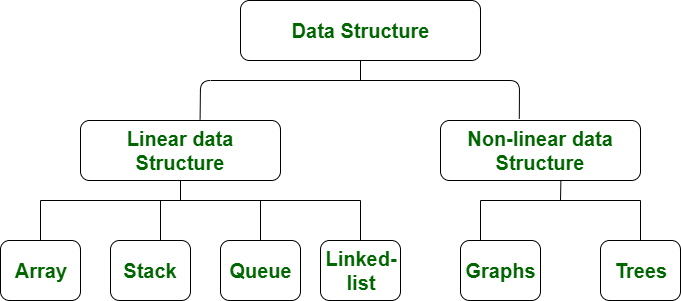
characters both upper and lower case such as ‘Z’ or ‘z’.

1. **Non-Primitive Data Structures:**

The Non-Primitive Data Structures are created with the help of the primitive data structures. The Non-primitive data structures are more complicated than primitive data structures but highly useful.

The non-primitive data types are the types that are defined by the programmer. Non-Primitive Data structures are classified into two types

* 1. Linear Data Structures
  2. Non-Linear Data Structures



**Q) Difference between Primitive Data Structures and Non-Primitive Data Structures:**

|  |  |
| --- | --- |
| Primitive Data Structure | Non-Primitive Data Structure |
| Primitive Data Structure are predefined in the language | Non-Primitive Data structure are not defined in language and  created by the programmer |
| Primitive Data structures will have  a certain value | Non-Primitive Data structure can  have NULL value |
| The size depends upon the type of  data structure | The size of non-primitive data  structure is not fixed |
| The primitive data structure starts  with lowercase | The non-primitive data type starts  with an uppercase |
| Can be used to call methods to  perform operations | Cannot be used. |

1. **Linear and Non-Linear data structure**

A Non-Primitive Data Structure is classified into two categories:

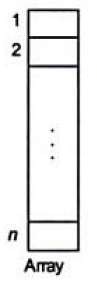
* 1. Linear data structure
  2. Non-linear data structure

**1. Linear data structure:**

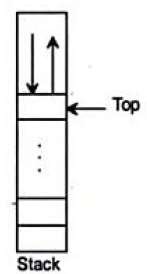
A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

The types of linear data structures are Array, Queue, Stack, Linked List.

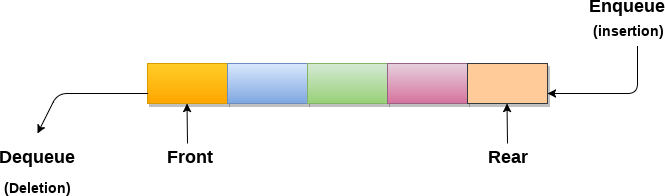
**Array**: An array consists of data elements of a same data type. For example, if we want to store the roll numbers of 10 students, so instead of creating 10 integer type variables, we will create an array having size10. Therefore, we can say that an array saves a lot of memory and reduces the length of the code.



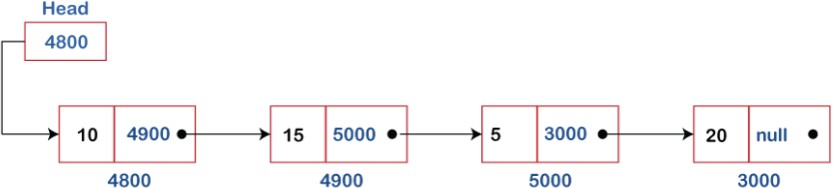
**Stack:** It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.



**Queue:** It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue front and rear. The insertion operation performed at the back end is known ad enqueue, and the deletion operation performed at the front end is known as dequeue. Queue can be represented as below



**Linked list:** Linked list is a collection of nodes in which one node is connected with another node. A node consists of two parts, i.e., one is the data element and the second one is address part as shown on below.



In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the NULL value in the address part.

**2. Non-linear data structure:**

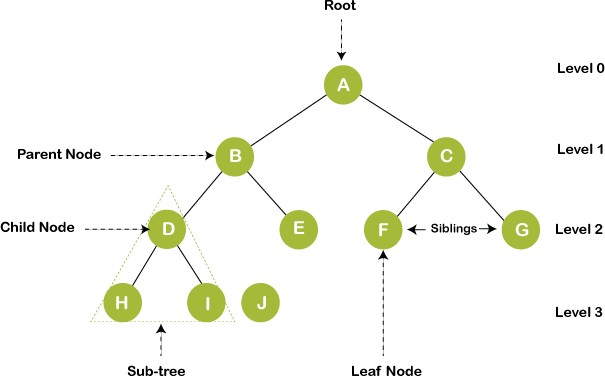
A non-linear data structure is also a data structure in which the data elements are not stored sequentially. As the arrangement is nonsequential, so the data elements cannot be traversed or accessed in a single run. A non-linear data structure, an element can be connected to more than two elements.

Trees and Graphs are the types of non-linear data structure.

**Tree:** Tree is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a tree data structure is shown below:

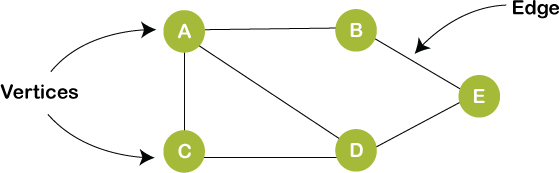
For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the below

diagram, A represents a manager, B and C represent the officers, and other nodes represent the clerks.



**Graph**: A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices.

The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Facebook. In the case of Facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.



Q) Difference Between Linear Vs Non-Linear Data Structures:

|  |  |
| --- | --- |
| Linear Data structure | Non-Linear Data structure |
| In this structure, the elements are arranged sequentially or linearly and attached to one another. | In this structure, the elements are arranged hierarchically or non-linear manner. |
| Arrays, linked list, stack, queue are the types of a linear data structure. | Trees and graphs are the types of a non-linear data structure. |
| Due to the linear organization, they are easy to implement. | Due to the non-linear organization, they are difficult to implement. |
| As linear data structure is a single level, so it requires a single run to traverse each data item. | The data items in a non-linear data structure cannot be accessed in a single run. It requires multiple runs to be traversed. |
| Each data item is attached to the previous and next items. | Each item is attached to many other items. |
| This data structure does not contain any hierarchy, and all the data elements are organized in a single level. | In this, the data elements are arranged in multiple levels. |
| In this, the memory utilization is not efficient. | In this, memory is utilized in a very efficient manner. |
| The time complexity of linear data structure increases with the increase in the input size. | The time complexity of non- linear data structure often remains same with the increase in the input size. |
| Linear data structures are mainly used for developing the software. | Non-linear data structures are used in image  processing and Artificial Intelligence. |

**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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

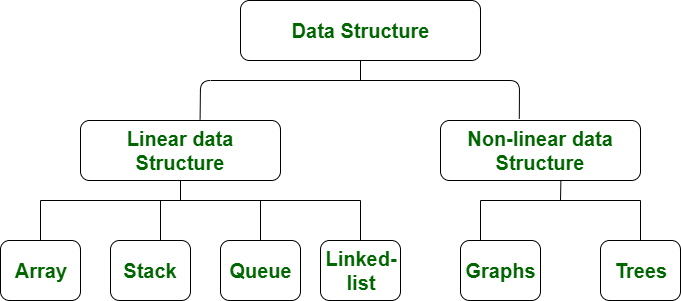
**Arrays:**

**Introduction to Linear and Non- Linear Data Structures**

1. **Linear and Non-Linear data structure**

A Non-Primitive Data Structure is classified into two categories:

* 1. Linear data structure
  2. Non-linear data structure



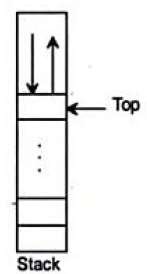
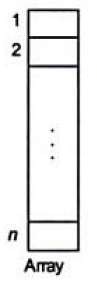
**1. Linear data structure:**

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

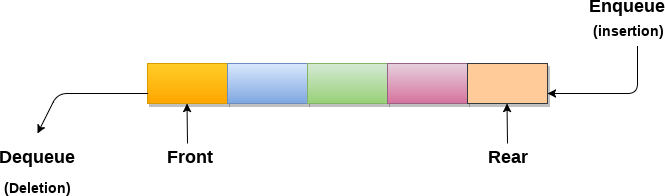
The types of linear data structures are Array, Queue, Stack, Linked List.

**Array**: An array consists of data elements of a same data type. For example, if we want to store the roll numbers of 10 students, so instead of creating 10 integer type variables, we will create an array having size10. Therefore, we can say that an array saves a lot of memory and reduces the length of the code.

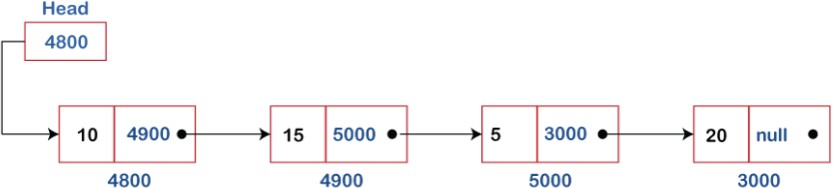
**Stack:** It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.



**Queue:** It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue front and rear. The insertion operation performed at the back end is known ad enqueue, and the deletion operation performed at the front end is known as dequeue. Queue can be represented as below



**Linked list:** Linked list is a collection of nodes in which one node is connected with another node. A node consists of two parts, i.e., one is the data element and the second one is address part as shown on below.



In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the NULL value in the address part.

**2. Non-linear data structure:**

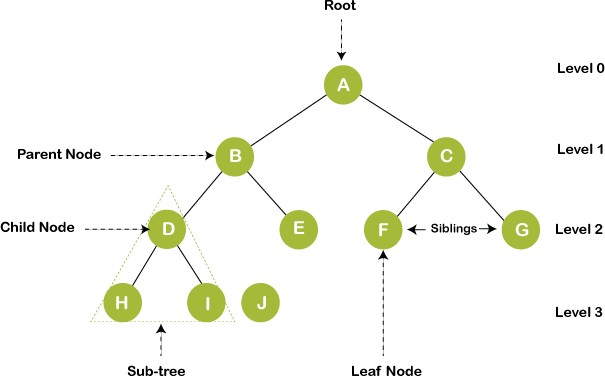
A non-linear data structure is also a data structure in which the data elements are not stored sequentially. As the arrangement is nonsequential, so the data elements cannot be traversed or accessed in a single run. A non-linear data structure, an element can be connected to more than two elements.

Trees and Graphs are the types of non-linear data structure.

**Tree:** Tree is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a tree data structure is shown below:

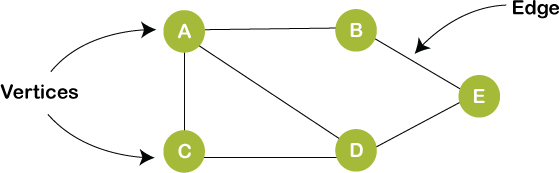
For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the below

diagram, A represents a manager, B and C represent the officers, and other nodes represent the clerks.



**Graph**: A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices.

The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Facebook. In the case of Facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.



**Q) What is an Array?**

An array is a collection of similar type of data items stored at contiguous memory locations which share a similar name. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc.

**Examples of array.**

1. IfI want to store salaries of a group of employees.

2. List of employees in an organization.

3. List of products and their cost sold by a store.

4. List of customers and their telephone numbers. etc

**Advantage of Arrays:**

1) **Code Optimization:** Less code to the access the data.

2) **Ease of traversing:** By using the for loop, we can retrieve the elements of an array easily.

3) **Ease of sorting:** To sort the elements of the array, we need a few lines of code only.

4) **Random Access:** We can access any element randomly using the array.

**Disadvantage of Array:**

1. **Fixed Size:** Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically.

==========================================================

**Q) Declaration of an Array?**

We can declare an array in the c language using one of the following ways.

data\_type array\_name[array\_size]; OR

data\_type [array\_size] array\_name;

Here, data\_type is the type of data you want to store in the array.

array\_name is the name of the array.

array\_size is the size of the array.

**Example:** int Salary[5];

==========================================================

**Q) Storing Values in Array? OR Initialization of Array?**

After declaration we need to put some values to an array. Assigning values to an is called initialization. The simplest way to initialize an array is by using the index/subscript of each element. We can initialize each element of the array by using the index. The syntax for the same is

array\_name[subscript/index] = value; OR

array\_name[array\_size] = { values based on size};

In C the subscript is always starts with Zero and ends with a value one less than the size of the array. Examples are,

Salary[0] = 2000;

Salary[1] = 3000;

Salary[2] = 1000;

Salary[3] = 5000;

Salary[4] = 4000;

These all values can be assigned using a single line

Salary[5] = {2000,3000,1000,5000,4000};

We can also initialize arrays at the time of declaration.

data\_type array\_name[array\_size] = { value…};

Example:

int Salary[5] = {2000,3000,1000,5000,4000};

In the memory it can be initialized as

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2000 | 30000 | 1000 | 5000 | 4000 |

Salary[0] Salary[1] Salary[2] Salary[3] Salary[4]

==========================================================

**Q) Accessing Array Elements:**

Once an array is declared and initialized, we can access the individual element of array by using an integer index or subscript. Array index starts with 0 and goes till size of array minus 1. The array elements are stored at contiguous memory locations.

**Example:** int marks[6] = {50,65,35,29,50,67};

Here,marks is an array storing the marks of 6 subjects. I can access the each subject marks by using the index or subscript.

marks[0] value is 50

marks[1] value is 65

marks[2] value is 35

marks[3] value is 29

marks[4] value is 50

marks[5] value is 67

==========================================================

**Q) Operations on Arrays?**

There are a number of operations that can be performed on an array which are:

1. Traversal
2. Copy
3. Reverse
4. Sort
5. Insertion
6. Deletion
7. Search
8. Merge
9. Update

**1. Traversal:**Traversing an Array means going through each element of an Array exactly once. We start from the first element and go to the last element. Traversal means accessing each array element for a specific purpose, either to perform an operation on them, counting the total number of elements or else using those values to calculate some other result.

**2. Copy:** Copying array elements to another array will create another array of the same length and elements as the original one.In order to so, we need to know the length of original array in advance. The destination array should also be of the same or greater size as that of original array in order to hold the array contents. The copying of elements would be done on index by index basis.

**3. Reverse:** Reversing an array means that the sequence of elements of array will be reversed.

For Examle, array ‘A’ has two elements : A[0] = 1; A[1] = 2; then after reversal A[0] = 2 and A[1] = 1.

There are two ways to perform reversal of array. The first one is Using another array to store the reverse values of the array and the second one is without using another aray to store the reverse values of an array.

**4. Sorting:** Sorting elements of array means to order the elements either in ascending or descending order. By default the order is ascending order. There are a number of algorithms or techniques available for sorting arrays in C. The basic approach to sorting is Bubble sort method where in nested loop is used to sort elements of array.

Insertion Operation

**5. Insertion:** Insert operation is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, at the end, or any given index of array.

**6. Deletion Operation:** Deletion is used to removing an existing element from the array and re-organizing all elements of an array.

**7. Search Operation:** Search operation is used to perform a search operation of an array element based on its value or its index.

**8. Merge:** Merge is used to combine two or more arrays into one array.

**9. Update Operation:** Update operation is used to perform a update an existing element of the array at a given index.

==========================================================

**Q) Types of Arrays?**

We can use arrays to represent not only lists of values but also tables of data in two, three or more dimensions. Arrays are classified into three types

1. One-dimensional Arrays
2. Two-dimensional Arrays
3. Multidimensional Arrays

**Q) One-dimensional Array(1D Array)**

A list of items can be given one variable name and the individual elements can be accessed by the specification of their positions with respect to the start of the list. These positions are referred as index of an element. The list of elements can be access by a variable name assigned to it and its subscript is called one-dimensional array.

**Declaration of 1D array:**

Arrays must be declared before they are used. The general syntax for array declaration is

data\_type array\_name[array\_size];

data\_type [array\_size] array\_name;

Here, data\_type specifies the built-in data types like int, char, float, double etc.

For example, we want to represent a set of five numbers 30,20,10,50,40 by an array variable number then we can declare number as follows

int number[5];

number[5]={30,20,10,50,40};

The computer reserves five storage locations as below

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Number[0] number[1] number[2] number[3] number[4]

The values to array elements can be assigned as follows

number[0]=30

number[1]=20

number[2]=10

number[3]=50

number[4]=40

In the computer memory these values can be assigned as below

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 30 | 20 | 10 | 50 | 40 |

number[0] number[1] number[2] number[3] number[4]

**Initializing values:**

The values can be assigned to an array in the same way as the any variable. The general syntax is

data\_type array\_name[size]={values};

**Example:**

int a[4]={25,35,45,55};

In computer memory it can be represented as

|  |  |  |  |
| --- | --- | --- | --- |
| 25 | 35 | 45 | 55 |

a[0] a[1] a[2] a[3]

==========================================================

**Q) Two-Dimensional Array( 2D Array)**

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as tables which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database.

**Declaration of two-dimensional Array:**

The syntax to declare the 2D array is given below.

data\_type array\_name[rows][columns];

Consider the following **example**.

int twodimen[4][3];

Here, 4 is the number of rows, and 3 is the number of columns.

**Initialization of 2D Array**

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};

**Two-dimensional** array example in C

#include<stdio.h>

int main(){

int i=0,j=0;

int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};

//traversing 2D array

for(i=0;i<4;i++){

for(j=0;j<3;j++){

printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);

}//end of j

}//end of i

return 0;

}

**Output:**arr[0][0] = 1

arr[0][1] = 2

arr[0][2] = 3

arr[1][0] = 2

arr[1][1] = 3

arr[1][2] = 4

arr[2][0] = 3

arr[2][1] = 4

arr[2][2] = 5

arr[3][0] = 4

arr[3][1] = 5

arr[3][2] = 6

==========================================================

**Q) Multi-dimensional array (ND array):**

The group of values that are stored under a single name by using 'n' subscript values. C language allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general syntax of multi-dimensional array is

data\_type array\_name[s1][s2][s3]....[sn];

where sn is the size of the nth dimension.

Some examples are

int a[2][4][5];

char ch[3][1][2][3];

Here a is a three-dimensional array with the name a and store 40 values.

ch is another array and it stores 18 values into an array.

In mathematical it is represented by using 3D axis. It is represented using A[iji]. Here A is 3D array and i,j,k is the value stores at x-axis,y-axis and z-axis.

**Initialization of values:**

In multi-dimensional array the values can be assigned as follows

data\_type array\_name[size1][size2][size3] ....[sizen] ={ values of size1\*size2\*size3\*...\*sizen};

int b[2][1][3]={{2,4,0},{1,5,2}};

we can access the values using the subscript or index as follows

b[0][0][0]=2

b[0][0][1]=4

b[0][0][2]=0

b[1][0][0]=1

b[1][0][1]=5

b[1][0][2]=2

**Overview of POINTERS**

A pointer is a variable which stores the memory address of another variable. Pointers are used to store the address of variables.

**Declaring a pointer:**

Like other variables, the pointer variable must be declared before using in the program

**Syntax:**

datatype \*ptr\_var;

**Ex:** int \*ptr;

char \*c;

Here ‘ptr’ is a pointer variable that contains address of a variable which is of integer type and c is a pointer variable that contains address of a variable which is character type.

**Initializing a pointer:**

We can assign the address of a variable to a pointer variable by using address (&) operator.

**Syntax:** pointer\_variable= &variable;

**Ex:** int a,\*p;

a=10;

p=&a;

**Accessing Pointer variable:**

We can access the value of a pointer variable by using the indirection (\*) operator.

**Syntax:** \*pointer\_name;

**POINTER AND ARRAYS**

It is a collection of addresses (or) collection of pointers.

**Declaration:- Datatype \*pointer-name[size];**

**Ex:- Int \*x[5];**

It represents an array of pointers with 5 elements. That are x[0],x[1],x[2],x[3] & x[4].

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in memory locations. The base address is the location of the first array element (index 0) of the array. Suppose we declare an array x as follows:

**int x[5]={1,2,3,4,5};**

if we declare **p** as an integer pointer, then we can make the pointer **p** to point to the array x by the following assignment:

**int \*p;**

**p=&x[0];**

**1.Array of integer pointers:** pointer may be arrayed like any other data type. An array of pointers can be declared as:

int \*ptr[5];

The above statement declares an array of 5 pointers where each of the pointer points to an integer variable. For example, look at the following code:

#include<stdio.h>

void main()

{

int \*ptr[3],i;

int a=10,b=20,c=30;

ptr[0]=&a;

ptr[1]=&b;

ptr[2]=&c;

printf(“a,b and c values are \n”);

for(i=0;i<3;i++)

printf(“%d”,\*ptr[i]);

getch();

}

**2.Array of character pointers:** now consider an array of character pointers that is pointed to the strings.

**char \*ptr[3];**

In the ptr array, each element is a character pointer. Initialize aan array of characters with three strings can be given as:

Char \*ptr[3]={“Rohan”,”Raj”,”Sree”};

**Ex:-**

#include<stdio.h>

void main()

{

Char \*ptr[3]={“Rohan”,”Raj”,”Sree”};

int i;

for(i=0;i<3;i++)

{

printf(“%s”,ptr[i]);

}

getch();

}

## Q) Pointers Vs Arrays

|  |  |
| --- | --- |
| **Arrays** | **Pointers** |
| An array is a collection of elements  of similar data type. | The pointer is a variable that stores  the address of another variable. |
| An array size decides the number  of variables it can store. | A pointer variable can store the  address of only one variable in it. |
| Arrays can be initialized at the  definition. | Pointers can not be initialized at  the definition. |
| Arrays are static | Pointers are dynamic |
| Memory allocated for arrays at  compile time. | Memory allocated for pointer at  run time. |

**CHAPTER-2**

**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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Dynamic memory allocation**

* The process of allocating memory to the variable during execution of the program (or) at run time is known as dynamic memory allocation.
* In dynamic memory allocation we can be increased memory size dynamically.
* In Linked List we are using dynamic memory allocation
* Stdlib.h is the header file of the dynamic memory allocation.

C provides four library functions to automatically allocate memory at the runtime . they are:

* 1. malloc()
  2. calloc()
  3. realloc()
  4. free()

**malloc():-**

**“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

**Syntax:**

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

**For Example:**

***ptr = (int\*) malloc(100 \* sizeof(int));***

**calloc():-**

**“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

**Syntax:**

ptr = (cast-type\*)calloc(n, element-size);

**For Example:**

***ptr = (float\*) calloc(25, sizeof(float));***

**realloc();-**

**“realloc”** or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

**Syntax:**

**ptr = realloc(ptr, newSize);**

where ptr is reallocated with new size 'newSize'

### free() method

**“free”** method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

**Syntax:**

free(ptr);

# Q) 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.**

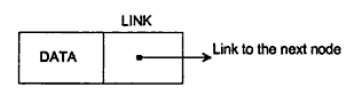
***Q) DISCUSS ABOUT THE LINKED LIST IMPLEMENTATION CONCEPTS/* Introduction to Linked list:**

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) TYPES OF LINKED LISTS

There are 3 different implementations of Linked List available, they are:

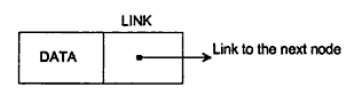
* 1. Single Linked List
  2. Double Linked List
  3. Circular Linked List

1. **LINKED LIST (OR) SINGLE LINKED LIST**

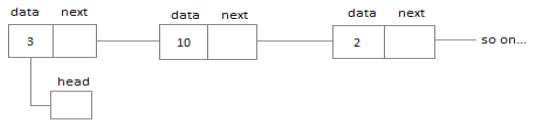
**A single Linked list is a sequence of nodes in which every node has link to its next node in the sequence.**

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:**

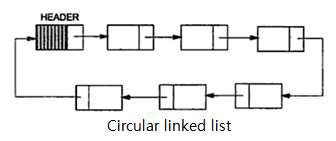
* + In a single linked list, the address of the first node always stored in a reference node known as

“HEAD” or “FRONT”.

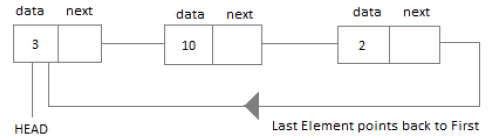
* + Always LINK part of the last node must be NULL.

# 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

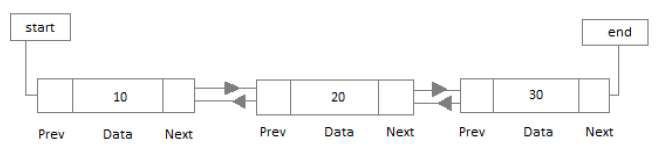
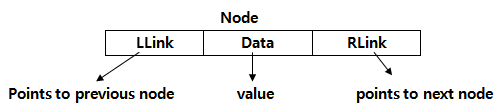


# Example:



**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.

**Advantages of Linked Lists**

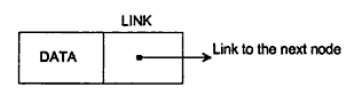
* + They are a dynamic in nature which allocates the memory when required.
  + Insertion and deletion operations can be easily implemented.
  + Stacks and queues can be easily executed.
  + Linked List reduces the access time

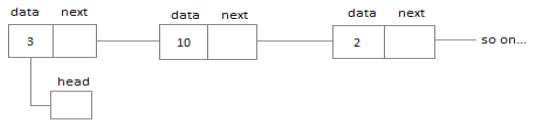
**Disadvantages of Linked Lists**

* + The memory is wasted as pointers require extra memory for storage.
  + No element can be accessed randomly; it has to access each node sequentially.
  + Reverse Traversing is difficult in linked list.

**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.

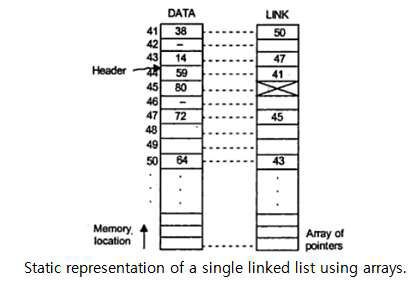
# Q) Representation of a Linked List in Memory:

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

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

# 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.



# 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.

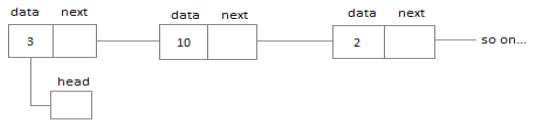
**Single Linked List:**

**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.

**Operations/ Implementation of Singly Linked list:**

There are various operations which can be performed on singly linked list. Here are the most basic operations on Singly Linked list.

* + Insertion
  + Deletion
  + Searching
  + Traversing

**Insertion:**

Insertion operation is used to insert a node to the list. If the list is empty, then that is the first and last node of a list. The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

* 1. Inserting a node at beginning
  2. Inserting a node at end of the list
  3. Inserting a node after specified node

1. **Inserting a node at beginning:** It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.

We will use below algorithm to insert an element at the beginning of the linked list:

Step 1 : BEGIN

Step 2: IF HEAD/ STATRT = NULL

Write OVERFLOW Go to Step 7 [END OF IF]

Step 3: SET NEW\_NODE = PTR

Step 4: SET PTR = PTR → NEXT

Step 5: SET NEW\_NODE → DATA = VAL Step

6: SET NEW\_NODE → NEXT = HEAD

Step 7: SET HEAD = NEW\_NODE

Step 8: EXIT

1. **Inserting a node at end of the list:** It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list, or it can be inserted as the last one. Different logics are implemented in each scenario. We will use the below algorithm to insert an element at the end of the Linked list.

Step 1: BEGIN

Step 2: IF HEAD/ STATRT = NULL

Write OVERFLOW Go to Step 11

[END OF IF]

Step 3: SET NEW\_NODE = PTR

Step 4: SET PTR = PTR - > NEXT

Step 5: SET NEW\_NODE - > DATA = VAL

Step 6: SET NEW\_NODE - > NEXT = NULL

Step 7: SET PTR = HEAD

Step 8: Repeat Step 9 while PTR - > NEXT != NULL

Step 9: SET PTR = PTR - > NEXT

[END OF LOOP]

Step 10: SET PTR - > NEXT = NEW\_NODE Step 11: EXIT

1. **Inserting a node after specified node :** It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted.

We will use the below algorithm to insert a node after the specified node.

STEP 1: IF HEAD/ STATRT = NULL

WRITE OVERFLOW GOTO STEP 12

END OF IF

STEP 2: SET NEW\_NODE = PTR STEP 3: NEW\_NODE → DATA = VAL STEP 4: SET TEMP = HEAD

STEP 5: SET i = 0

STEP 6: REPEAT STEP 7 AND 8 UNTIL i<loc

STEP 7: TEMP = TEMP → NEXT

STEP 8: IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT" GOTO STEP 12

END OF IF END OF LOOP

STEP 9: PTR → NEXT = TEMP → NEXT STEP 10: TEMP → NEXT = PTR

STEP 11: SET PTR = NEW\_NODE

STEP 12: EXIT

**Deletion:**

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

1.Deletion at beginning

2.Deletion at the end of the list:

3.Deletion after specified node:

1. **Deletion at beginning:** It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just needs a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head. This will be done by using the following statements. We will use the below algorithm to delete the node at the beginning of a linked list.

Step 1: IF HEAD/ STATRT = NULL

Write UNDERFLOW

Go to Step 5 [END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET HEAD = HEAD -> NEXT Step 4: FREE PTR

Step 5: EXIT

1. **Deletion at the end of the Singly Linked list:** It involves deleting the last node of the list. The list can be either empty or full. And also, the situations are there is a only one node that needs to be deleted and one more nodes and the last node to be deleted.

Step 1: IF HEAD/STATRT = NULL

Write UNDERFLOW

Go to Step 8 [END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT!= NULL

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT [END OF LOOP]

Step 6: SET PREPTR -> NEXT = NULL

Step 7: FREE PTR

Step 8: EXIT

1. **Deletion after specified node:** It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list. We will use the below algorithm

STEP 1: IF HEAD/START = NULL

WRITE UNDERFLOW GOTO STEP 11

END OF IF

STEP 2: SET TEMP = HEAD STEP 3: SET I = 0

STEP 4: REPEAT STEP 5 TO 8 UNTIL I STEP 5: TEMP1 = TEMP

STEP 6: TEMP = TEMP → NEXT

STEP 7: IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT" GOTO STEP 12

END OF IF STEP 8: I = I+1

END OF LOOP

STEP 9: TEMP1 → NEXT = TEMP → NEXT

STEP 10: FREE TEMP STEP 11: EXIT

1. **Traversing in a Singly Linked list:** Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following algorithm.

STEP 1: SET PTR = HEAD/START STEP 2: IF PTR = NULL

WRITE "EMPTY LIST" GOTO STEP 7

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL

STEP 5: PRINT PTR→ DATA STEP 6: PTR = PTR → NEXT [END OF LOOP]

STEP 7: EXIT

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

Step 1: SET PTR = HEAD/START

Step 2: Set I = 0

STEP 3: IF PTR = NULL WRITE "EMPTY LIST" GOTO STEP 8

END OF IF

STEP 4: REPEAT STEP 5 TO 7 UNTIL PTR != NULL

STEP 5: if ptr → data = item

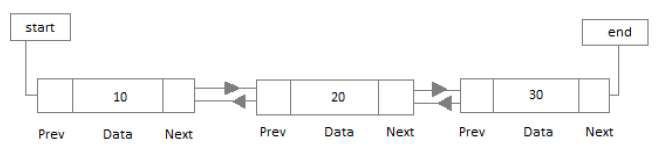
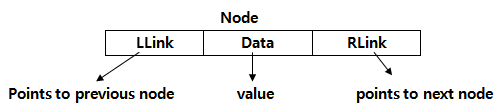
write i+1 End of IF

STEP 6: I = I + 1

STEP 7: PTR = PTR → NEXT [END OF LOOP] STEP 8: EXIT

**2. Doubly 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.

**Operations on doubly linked list:**

Below are the operations that can be performed on Doubly linked list.

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

1.**Inserting a NODE in DOUBLY linked list:** Insertion operation is used to insert a given Node to the linked list. Insertion can be done with the following ways

* + Insertion at beginning
  + Insertion at end
  + Insertion after specified node

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

* Allocate the space for the new node in the memory. This will be done by using the following statement.
* Check whether the list is empty or not. The list is empty if the condition head/START == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL, and the head pointer will point to this node.
* In the second scenario, the condition head == NULL becomes false, and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

We will use the below algorithm to insert node at the beginning of the linked list.

Step 1: IF ptr = NULL

Write OVERFLOW Go to Step 9

[END OF IF]

Step 2: SET NEW\_NODE = ptr

Step 3: SET ptr = ptr -> NEXT

Step 4: SET NEW\_NODE -> DATA = VAL

Step 5: SET NEW\_NODE -> PREV = NULL

Step 6: SET NEW\_NODE -> NEXT = START

Step 7: SET head -> PREV = NEW\_NODE

Step 8: SET head = NEW\_NODE

Step 9: EXIT

**2.Inserting in doubly linked list at the end of linked list:** In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty, or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

* + Allocate the memory for the new node. Make the pointer ptr point to the new node being inserted.
  + Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL, and the head pointer will point to this node.
  + In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer temp to head and traverse the list by using this pointer.

We will use below algorithm to insert an element. Step 1: IF PTR = NULL

Write OVERFLOW Go to Step 11

[END OF IF]

Step 2: SET NEW\_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW\_NODE -> DATA = VAL

Step 5: SET NEW\_NODE -> NEXT = NULL

Step 6: SET TEMP = START

Step 7: Repeat Step 8 while TEMP -> NEXT != NULL

Step 8: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW\_NODE Step 10C: SET NEW\_NODE -> PREV = TEMP

Step 11: EXIT

3.**Insertion in doubly linked list after specified node:** In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

* + Allocate the memory for the new node.
  + Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.
  + The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node therefore we need to make a ptr pointer adjustments here. Make the next pointer of ptr point to the next node of temp.

Step 1: IF PTR = NULL

Write OVERFLOW Go to Step 15

[END OF IF]

Step 2: SET NEW\_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW\_NODE -> DATA = VAL

Step 5: SET TEMP = START

Step 6: SET I = 0

Step 7: REPEAT 8 to 10 until I

Step 8: SET TEMP = TEMP -> NEXT

STEP 9: IF TEMP = NULL

STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

GOTO STEP 15 [END OF IF]

[END OF LOOP]

Step 11: SET NEW\_NODE -> NEXT = TEMP -> NEXT

Step 12: SET NEW\_NODE -> PREV = TEMP

Step 13: SET TEMP -> NEXT = NEW\_NODE

Step 14: SET TEMP -> NEXT -> PREV = NEW\_NODE

Step 15: EXIT

1. **Deletion in DOUBLY linked list:** Deletion operation is used to delete the given Node from the linked list. Deletion can be done with the following ways
2. Deletion at beginning
3. Deletion at end
4. Deleting a node after specified node
5. **Deletion in doubly linked at the beginning:** Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW GOTO STEP 6

STEP 2: SET PTR = HEAD

STEP 3: SET HEAD = HEAD → NEXT

STEP 4: SET HEAD → PREV = NULL

STEP 5: FREE PTR

STEP 6: EXIT

1. **Deletion in doubly linked list at the end:** Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position. In order to delete the last node of the list, we need to follow the following steps.

* If the list is already empty, then the condition head == NULL will

become true and therefore the operation cannot be carried on.

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

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 7 [END OF IF]

Step 2: SET TEMP = HEAD

Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL

Step 4: SET TEMP = TEMP->NEXT

[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT

1. **Deletion in doubly linked list after the specified node:** In order to delete the node after the specified data, we need to perform the following steps.

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

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 9 [END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> DATA != ITEM

Step 4: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 5: SET PTR = TEMP -> NEXT

Step 6: SET TEMP -> NEXT = PTR -> NEXT

Step 7: SET PTR -> NEXT -> PREV = TEMP Step 8: FREE PTR Step 9: EXIT

**3) Searching for a specific node in Doubly Linked List:** We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

* Copy head pointer into a temporary pointer variable ptr.
* declare a local variable I and assign it to 0.
* Traverse the list until the pointer ptr becomes null. Keep shifting

pointer to its next and increasing i by +1.

* Compare each element of the list with the item which is to be

searched.

* If the item matched with any node value, then the location of that value I will be returned from the function else NULL is returned.

We will use below algorithm for this

Step 1: IF HEAD == NULL

WRITE "UNDERFLOW" GOTO STEP 8

[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Set i = 0

Step 4: Repeat step 5 to 7 while PTR != NULL

Step 5: IF PTR → data = item

return item [END OF IF]

Step 6: i = i + 1

Step 7: PTR = PTR → next Step 8: Exit

1. **Traversing in doubly linked list:** Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr. Then, traverse through the list by using while loop. Keep shifting value of pointer variable ptr until we find the last node. The last node contains null in its next part. We will use the below algorithm

Step 1: IF HEAD == NULL

WRITE "UNDERFLOW" GOTO STEP 6

[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Repeat step 4 and 5 while PTR != NULL

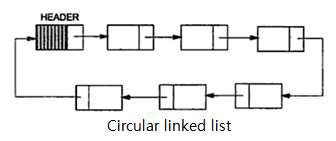
Step 4: Write PTR → data

Step 5: PTR = PTR → next

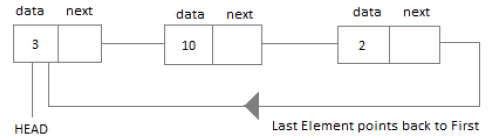
Step 6: Exit

1. **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:



**Operations on Circular linked list:** There are many operations that can be performed on Circular linked lists. Here are some of them

* 1. Insertion
  2. Deletion
  3. Traversing
  4. Searching

1. **Insertion:** Insertion operation is used to insert an element into circular linked list. In circular linked list Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

* Allocate the memory space for the new node.
* If the List is empty, then the condition head == NULL will be true. Since, the list in which, we are inserting the node is a circular linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node.
* If the List has some nodes, then the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list.

Step 1: IF PTR = NULL

Write OVERFLOW Go to Step 11

[END OF IF]

Step 2: SET NEW\_NODE = PTR

Step 3: SET PTR = PTR -> NEXT

Step 4: SET NEW\_NODE -> DATA = VAL

Step 5: SET TEMP = HEAD

Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 7: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 8: SET NEW\_NODE -> NEXT = HEAD

Step 9: SET TEMP → NEXT = NEW\_NODE

Step 10: SET HEAD = NEW\_NODE

Step 11: EXIT

1. **Deletion:** Deletion operation is used to delete the node from Circular linked list. In order to delete a node in circular linked list, we need to make a few pointer adjustments. There are three ways of deleting a node in circular linked list. First one is, If the list is empty then the condition head == NULL will become true, then just print underflow and exit. Second one is, If the list contains single node then, the condition head → next == head will become true. In this case, we need to delete the entire list and make the head pointer free. And last, If the list contains more than one node then, in that case, we need to traverse the list by using the pointer ptr to reach the last node of the list.

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8 [END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Step 4 while PTR → NEXT != HEAD

Step 4: SET PTR = PTR → next

[END OF LOOP]

Step 5: SET PTR → NEXT = HEAD → NEXT

Step 6: FREE HEAD

Step 7: SET HEAD = PTR → NEXT

Step 8: EXIT

1. **Traversing in Circular linked list:** Traversing in circular linked list can be done through a loop. Initialize the temporary pointer variable temp to head pointer and run the while loop until the next pointer of temp becomes head. The algorithm is given below

STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

WRITE "EMPTY LIST" GOTO STEP 7

END OF IF

STEP 3: REPEAT STEP 4 AND 5 UNTIL PTR → NEXT != HEAD

STEP 4: PRINT PTR → DATA

STEP 5: PTR = PTR → NEXT

[END OF LOOP]

STEP 6: PRINT PTR→ DATA

STEP 7: EXIT

1. **Searching in circular linked list:** Searching in circular linked list needs traversing across the list. The item which is to be searched in the list is matched with each node data of the list once and if the match found then the location of that item is returned otherwise -1 is returned.

The algorithm for this is given below

Step 1: SET PTR = HEAD

Step 2: Set I = 0

STEP 3: IF PTR = NULL

WRITE "EMPTY LIST" GOTO STEP 8

END OF IF

STEP 4: IF HEAD → DATA = ITEM

WRITE i+1 RETURN [END OF IF]

STEP 5: REPEAT STEP 5 TO 7 UNTIL PTR->next != head

STEP 6: if ptr → data = item

write i+1 RETURN

End of IF STEP 7: I = I + 1

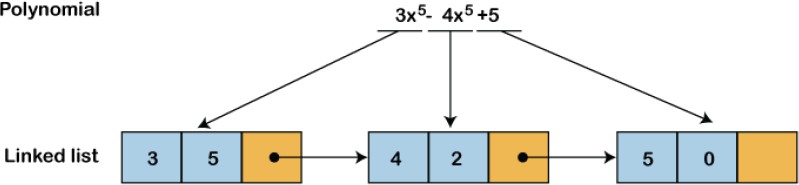
STEP 8: PTR = PTR → NEXT

[END OF LOOP] STEP 9: EXIT

# Q) Applications of Linked List?

The applications of the linked list are given below:

* Circular linked lists are mostly used in task maintenance in operating systems.
* With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial. We know that polynomial is a collection of terms in which each term contains coefficient and power. The coefficients and power of each term are stored as node and link pointer points to the next element in a linked list, so linked list can be used to create, delete and display the polynomial.



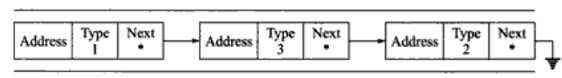
* + A sparse matrix is used in scientific computation and numerical analysis. So, a linked list is used to represent the sparse matrix.
* The various operations like student's details, employee's details or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
* Stack, Queue, tree and various other data structures can be implemented using a linked list.
* The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
* To implement hashing, we require hash tables. The hash table

contains entries that are implemented using linked list.

* A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

## Q) 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.



Atomic node list can be represented in C using unions and structures and it can be defined as below

typedef struct nodeList {

int type;

union atomicNode \*data; struct nodeList\* next;

}nodeList;

nodeList \* start = NULL;

typedef union atomicNode { char data;

int data;

}AtomNode;

The data structure for atomic node has been chosen as a union instead of structure type because only one member of data structure will be used for each data item.

Then union is the best data structure for C implementation of atomic node.

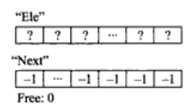
# Q) LINKED LIST IN ARRAYS

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 ith element of “L” is stored in the ith index of an array “Ele”.

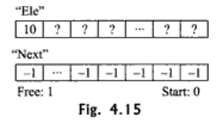
The node in a linked list contains two parts -“data” and “next”. These two parts of node are split and stored in two arrays “Ele” and “Next”. If Ele\*i+ represents the data part of the node then Next\*i+ denotes the next part of that node.

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 ith index of “Ele” and the “Next” is the node represented by jth 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.



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.

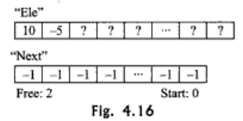
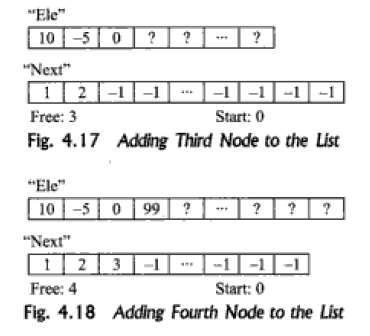


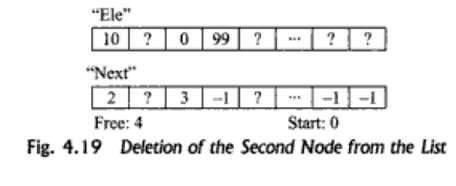
Figure 4.16 illustrates adding second element in the lis. Note that Ele[1] is set to -5, Next[0] is set to 1 and Next[1] is set to -1. Another addition will need to traverse the list from “start”, start = 0. As Next\*0+ = 1 the next node can be found in index 1 of the array “Ele” and “Next”. The Next\*1+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

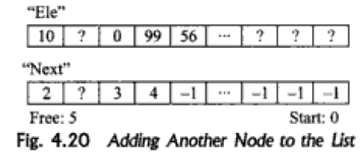


Now if we want to delete the second node in the linked list, the next field of the first node is to be changed, in respect of the next field of second node. The array will be shown as given in fig. 4.19.



Therefore, it can be seen that no shifting of elements is done. Only the “next” array is updated. Say, if we

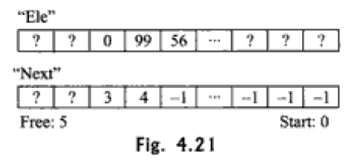
want to again add an element with value 56 to the list after deletion (fig. 4.20)



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.

**Q) Linked List vs Arrays**



Array of Linked list is an important data structure used in many applications. It is an interesting structure to form a useful data structure. It combines static and dynamic structures. Static means array and dynamic means Linked list used to form a useful data structure. This array of Linked list structure is most appropriate for applications.

let's understand how array is different from Linked list.

|  |  |
| --- | --- |
| **ARRAY** | **LINKED LIST** |
| Array is a collection of elements of similar data type. | Linked List is an ordered collection of elements of same type, which are connected to each other using pointers. |
| Array supports **Random Access**, which means elements can be accessed directly using their index, like arr[0] for 1st element, arr[6] for 7th element etc.  Hence, accessing elements in an array is **fast** with a constant time complexity of O(1). | Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element.  To access **nth** element of a linked list, time complexity is O(n). |
| In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory. | In a linked list, new elements can be stored anywhere in the memory.  Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements. |
| In array, **Insertion and Deletion** operation takes more time, as the memory locations are consecutive and fixed. | In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list.  Insertion and Deletion operations are **fast** in linked list. |
| Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**. | Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**. |
| In array, each element is independent and can be accessed using it's index value. | In case of a linked list, each node/element points to the next, previous, or maybe both nodes. |
| Array can be **single dimensional**, **two dimensional** or **multidimensional** | Linked list can be [**Linear(Singly) linked list**](https://www.studytonight.com/data-structures/linear-linked-list), [**Doubly linked list**](https://www.studytonight.com/data-structures/doubly-linked-list) or [**Circular linked list**](https://www.studytonight.com/data-structures/circular-linked-list) linked list. |
| Size of the array must be specified at time of array declaration. | Size of a Linked list is variable. It grows at runtime, as more nodes are added to it. |
| Array gets memory allocated in the **Stack** section. | Whereas, linked list gets memory allocated in **Heap** section. |

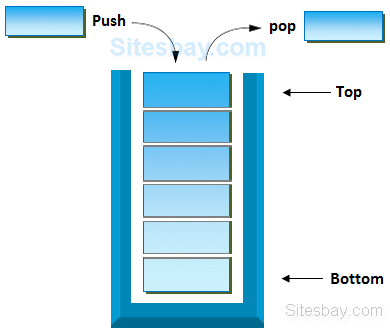
**UNIT III**

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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Q) Introduction to Stacks

**Definition:** A stack is a Linear Data Structure which stores data elements in a sequential manner. In Stack the elements are inserted and deleted from only one end called TOP. Stack follows LIFO(Last-In-First-Out) principle.



If Top = -1 then the stack is empty. If TOP = 0 then there is only one element in stack. If Top = MAX-1 then Stack is full. If TOP = MAX then Stack id OVERFLOW.

When a Stack is completely full then the STACK is OVEFLOW state and if the stack is empty then the stack is UNDERFLOW state. Stack performs the insertions and deletions at only one end that is TOP end.

**A stack is defined as follows:**

Struct stack

{

int a[arr];

int top;

}

**Example of Stack is:**

1.Collection of Plates in a stand as shown below. When you want to remove the plate, you remove the topmost plate from the queue.

2.Collection of Books in draw

3.Stack of Discs/rings

4.Stack of Chairs

# Q) Stack as an ADT

Stack can also be defined as ADT(Abstract Data Type). The stack ADT operates on a collection of elements like most ADTs, inserts, removes, and manipulates data items from the collection.

If a stack is an ADT it performs below operations

* 1. An empty stack is initialized
  2. Identify whether the stack is empty or not
  3. Identify if a stack is full or not
  4. If a stack is not full then the elements can be inserted at one end of stack. This operation is called PUSH
  5. If the stack is not empty, then delete the element at the TOP. This operation is called POP.
  6. If the stack is not empty then retrieve the TOP node element.

**Q) Operations on Stacks**

There are many operations that can be performed on Stacks. Below are the some of them:

1. push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
2. pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
3. isEmpty(): It determines whether the stack is empty or not.
4. isFull(): It determines whether the stack is full or not.'
5. Status(): it tells the status of the Stack
6. peek(): It returns the element at the given position.
7. count(): It returns the total number of elements available in a stack.
8. change(): It changes the element at the given position.
9. display(): It prints all the elements available in the stack

# Q)Representation of Stacks OR Implementation of Stacks?

Stacks can be represented in the memory in different ways. Mainly there are two ways:

* 1. Array Representation of Stacks
  2. Linked List Representation of Stacks

**1.Array Representation of Stacks:** In the computer’s memory stacks can be represented as an Array. Every Stack has a variable called TOP associated with the index. TOP stores the address of the topmost element of the stacks. This is the place where the elements can be added and deleted from stack.

There is a variable called MAX, which is used to store the maximum number of elements that the stack can store. ( MAX = Array Size)

If TOP=NULL, then the stack is empty and if TOP = MAX-1 then stack is full. (Here we are writing MAX-1, because Array Index always starts from 0 (ZERO)).

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 50 | 40 | 30 | 20 | 10 |  |  |  |  |  |  |  |

0 1 2 3 4 5 6 7 8 9 10 11

Here is an Array of stack with size 12. But the value of TOP is 4 and still we can insert 7 more elements to this stack.

**Operation of stack using Array:** We can perform all stack operations using Arrays. But mainly there are two operations those are Push and POP.

**PUSH Operation:**

Push operation is used to insert the element to the topmost position of the stack. Before inserting the element, we first check if TOP=MAX-1, then stack is full, then no more insertions can be done. If you try to insert an element to the stack which is full, then it will display an OVERFLOW message.

We will be using below algorithm to insert an element to stack

**Algorithm for PUSH:**

Step 1: IF TOP = MAX-1

PRINT OVERFLOW (Stack is FULL)

GOTO Step 4 [END OF IF)

Step 2: SET TOP = TOP +1

Step 3: SET STACK[TOP]=VALUE

Step 4: END

In the above algorithm in step 1 we first check for the OVERFLOW condition. In Step 2 TOP is incremented by 1 so that it points to the next location in the array. In Step 3 the value is inserted to the stack at the location pointed by TOP.

**POP Operation**:

POP is used to remove the element from the top of the stack. However before deleting an element we must check if TOP=NULL then the stack is empty, and no more deletions can be done. If you try to delete an element from the stack which is empty, then and UNDERFLOW message will be printed.

We will be using below algorithm to remove an element from stack .

**Algorithm for POP operation:**

Step 1: IF TOP = NULL

PRINT UNDERFLOW (Stack is empty)

GOTO Step 4 [END OF IF)

Step 2: SET VAL= STACK[TOP] Step 3: SET TOP = TOP - 1

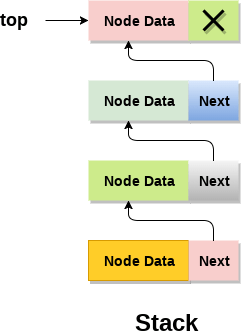
Step 4: END

In the above algorithm in step 1 we are checking for the UNDERFLOW condition. In Step 2, the value of the location in the stack pointed by TOP is stored in VAL. In Step 3, TOP is decremented by 1 so that it points to the previous location in the array.

**Linked Representation of Stacks:**

Creating a stack using Array is very easy and quick but the disadvantage is that the array must be declared with fixed size. In several applications size of the stack may differ during the program execution. During this time, we can represent stack using Linked lists.

In a linked stack every node has two parts – one part stores the data and the other part stores the address to the next node. The START pointer of linked list is used as TOP.

All insertions and deletions can be done at the node pointed by TOP. If TOP=NULL, then the stack is empty.The topmost node in the stack always contains null in its address field.

**Operation of stack using Linked List:**

We can perform all stack operations using linked list. But mainly Push and POP operations.

**PUSH Operation:**

Push operation is used to insert the element to the topmost position of the stack. The below algorithm is used to push the element into a linked stack.

**Algorithm:**

Step 1: Allocate a memory for the new node and name it as NEW\_NODE Step 2: set NEW\_NODEDATA=VALUE

Step 3: IF TOP= NULL THEN

SET NEW\_NODENEXT=NULL SET TOP=NEW\_NODE

ELSE

SET NEW\_NODENEXT=TOP SET TOP=NEW\_NODE

END IF Step 4: END.

In this algorithm in step 1 allocation a memory for the new node. In step 2, the DATA part of the nee node is assigned with the value to be stored in the node. In step 3, checking if TOP=NULL then the node that we are inserting is the first node and assigning NULL address to the Address part and this node to TOP. If the stack has some nodes, then the new node address will be TOP and new node will be TOP.

**POP Operation:**

POP is used to remove the element from the top of the stack. However, before deleting an element we must check if TOP=NULL then the stack is empty, and no more deletions can be done. If you try to delete an element from the stack which is empty, then and UNDERFLOW message will be printed.We will be using below algorithm to remove an element from stack.

**Algorithm for POP operation:**

Step 1: IF TOP = NULL

PRINT UNDERFLOW (Stack is empty)

GOTO Step 5 [END OF IF)

Step 2: SET PTR = TOP

Step 3: SET TOP= TOPNEXT Step 4: FREE PTR

Step 5: END

In the algorithm in step 1 we first check for the UNDERFLOW condition. In step 2, we use a pointer PTR that points to TOP. In step 3, TOP is pointing to next node in the Stack. In step 4 the memory occupied by the PTR will be removed.

# Q) Applications of Stack:

The following are the applications of the stack:

**1.String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "YOGIVEMANA" string, so we can achieve this with the help of a stack.

First, we push all the characters of the string in a stack until we reach the null character.

After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

**2.Implementing parentheses Checker:** Stacks can be used to check the validity of parentheses in an algebra expression.

**Example:** An algebra expression is valid if for every open bracket there is a corresponding close bracket.

(3+2-(9-5) \*2) is a valid expression.

{3+2-(9-5) \*2) is invalid expression

**3.Recursion:** The recursion means that the function is calling itself again. Since the recursive function calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.

Every recursive solution has two cases. They are

* 1. Base case: in which the problem is simple, and it will be solved directly with the same function.
  2. Recursive case: In which the problem is divided into smaller parts and the smaller parts performs the task and finally the solution is combination of these all sub parts.

The recursion is used in large and complex problems. to understand the recursion, we will calculate the factorial of a number n. To calculate the n! .., we multiply the number with the factorial of a number that is 1 less than the number. n!=n\*(n-1)!

Example is Calculate the factorial of a number 4. 4!=4\*3\*2\*1=24

This can be written as

4!=4\*3!, where 3!=3\*2!, Where 2!=2\*1! We know 1!=1 Finally we can write as 4!=4\*3\*2\*1=24

**4.Expression Evolution:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:

Infix to prefix Infix to postfix Prefix to infix

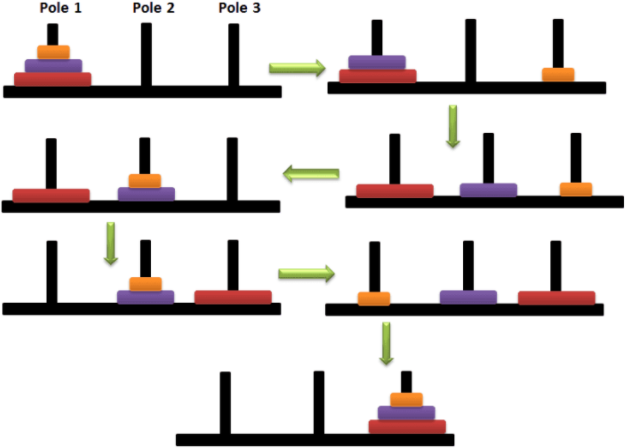
Prefix to postfix Postfix to infix

**5.Tower of Hanoi:** Another complex application of stack is Towers of Hanoi. Towers of Hanoi is one of the mathematical puzzles.

Suppose there are three poles A, B and C. there are n disks of decreasing size so that no two disks are of the same size. Initially all the disks are stacked on one pillar in their decreasing size and other two pillars are empty.

The problem is to move all the disks from one pillar to other using third pillar, so that

* Only one disk may be moved at a time.
* A disk may be moved from any pillar to another.
* At no time can a larger disk be placed on a smaller disk.



# Q) Stacks and Recursion

A function which calls itself is called a Recursion. There are some statements in the function calls itself to execute the same function.

While using a Recursive function, it is very important to define the exit condition from the function or then it may result into an infinite loop.

The recursive functions are very useful for solving mathematical problems. The recursive functions are used for calculating factorial of a number and fibonacci series etc

"Recursion" is technique of solving any problem by calling same function again and again until some breaking (base) condition where recursion stops, and it starts calculating the solution from there on. For eg. calculating factorial of a given number

Thus, in recursion last function called needs to be completed first.

Now Stack is a LIFO data structure i.e. (Last In First Out) and hence it is used to implement recursion.

The High-level Programming languages, such as Pascal, C etc. that provides support for recursion use stack for bookkeeping.

Also, as a function calls to another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.

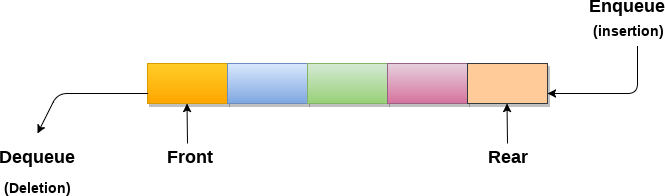
Recursion is extremely useful and extensively used because many problems are elegantly specified or solved in a recursive way.

The example of recursion as an application of stack is keeping books inside the drawer and the removing each book recursively.

**CHAPTER-2**

**QUEUES**

**Q) QUEUE Introduction**

 A Queue is a linear data structure, and the elements are stored in a sequential order. Queue follows FIFO (First in First Out) principle. In which the element which is inserted first will be removed first from the queue.

The elements in a queue are added at one end called the REAR and elements removed from the other end is called FRONT.

**Examples of a Queues:**

* + People moving on an Escalator.
  + People waiting outside the ticket window.
  + People waiting for a bus.
  + Cars lined at a toll bridge.

**Operations on Queue**

There are some fundamental operations performed on a Queue:

* Enqueue: The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
* Dequeue: The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
* Peek: This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
* Queue overflow (isfull): When the Queue is completely full, then it shows the overflow condition.
* Queue underflow (isempty): When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

# Q) QUEUE ADT

The queue ADT operates on a collection of elements of any proper type T and, like most ADTs, inserts, removes, and manipulates data items from the collection.

1. Initialize a queue to be empty

2. Determine if a queue is empty or not..

3. Determine if a queue is full or not.

4. Insert new element after the last element in a queue, if it is not null.

5. Delete the first element ina queue, if it is not empty

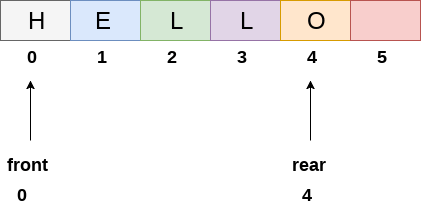
# Representation of Queues (OR) Implementation of Queues?

A queue can be implemented using two ways. Those are

* 1. Array Representation of Queue
  2. Linked list representation of Queue

**1.Array Representation of Queues:**

Queues can be represented using arrays. There are two variables i.e. FRONT and REAR, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of FRONT and REAR is -1 which represents the queue is empty.

Let’s, consider the below queue with 5 elements, where FRONT is 0 and REAR is 4. The size of the queue MAX is 6.

**Operations on Queues using Arrays:** We can perform all basic operations on queues using Arrays. Mainly there are two operations Enqueue(insertion) and Dequeue(deletion).

**Inserting an Element into Queue (Enqueue):** This operation can be used to insert an element into queue. Before inserting an element into queue, we must check for OVERFLOW condition, that is REAR=MAX-1. Here MAX is the size of the queue.

We will use the below algorithm to insert an element into queue using Array:

Step 1: IF REAR=MAX-1

WRITE OVERFLOW

Goto Step 4 END IF

Step 2: IF FRONT=-1 and REAR=-1

SET FRONT=REAR=0

ELSE

SET REAR=REAR+1

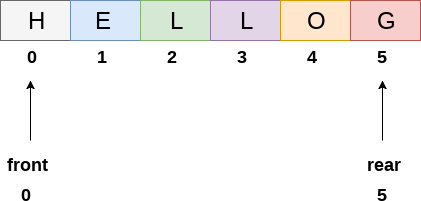
END IF

Step 3: SET QUEUE[REAR]=VALUE

Step 4: END

In step 1 we are checking for the OVERFLOW condition. In step 2, we are checking if the queue is empty, then both FRONT and REAR are set to ZERO(0). So that the new value can be stored at the Oth location. If the queue has some values, then increment the REAR by 1. In Step 3, we are assigning the Value at the REAR postion.

The queue after inserting an element for the above queue.

**Deleting an Element from Queue (Dequeue):** This operation is used to delete an element from queue. Before deleting an element from queue, we must check for UNDERFLOW condition, that is FRONT=-1.

We will use the below algorithm to delete an element from queue.

Step 1: IF FRONT = -1 or FRONT > REAR

Write UNDERFLOW

ELSE

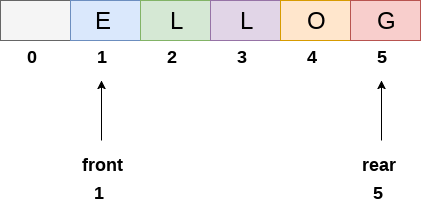
SET VAL = QUEUE[FRONT] SET FRONT = FRONT + 1

Remove VAL

[END OF IF] Step 2: EXIT

In step1, we check for underflow condition. It this is false then the queue has some values then FRONT is incremented by 1 so the FRONT is point to the next location and Remove the Value.

After deleting an element from Queue is shown below:

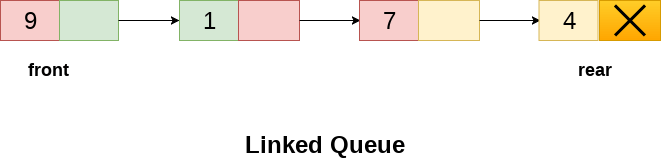


**2.Linked Representation of Queues:**

Array representation of Queue is very easy and convenient, but it allows the representation of queue with fixed size. In several applications the size of the queue may differ during program execution. The solution for this is representation of queue using Linked list.

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.



Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

**Operations on Linked Queue:**

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

**Insert operation:** The insert operation appends the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

We will use the below algorithm to insert an element into the queue. Step 1: Allocate the space for the new node PTR

Step 2: SET PTR -> DATA = VAL Step 3: IF FRONT = NULL

SET FRONT = REAR = PTR

SET FRONT -> NEXT = REAR -> NEXT = NULL

ELSE

SET REAR -> NEXT = PTR SET REAR = PTR

SET REAR -> NEXT = NULL

[END OF IF]

Step 4: END

In this algorithm first we are allocating some memory for the new node. In Step 2, the DATA part of the node is initialized with the Value to be inserted. In step 3, we check if the node is the first node of the linked queue then tag the new node as FRONT and REAR. Also, NULL is also stored into the same node NEXT part. If new node is not the first node in the list, then it is added at the REAR end of the linked queue.

**Deletion Operation:** Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case, we simply write underflow on the console and make exit.

We will use the below algorithm to delete an element from queue. Step 1: IF FRONT = NULL

Write " Underflow "

Go to Step 5 [END OF IF]

Step 2: SET PTR = FRONT

Step 3: SET FRONT = FRONT -> NEXT Step 4: FREE PTR

Step 5: END

In this algorithm first we check for the underflow condition. If this is true, then simply display underflow and exit. In step 2, we use a PTR which is assigned by FRONT. In step3, FRONT is made to point to next node in the queue. In step 4, the memory occupied by PTR is removed.

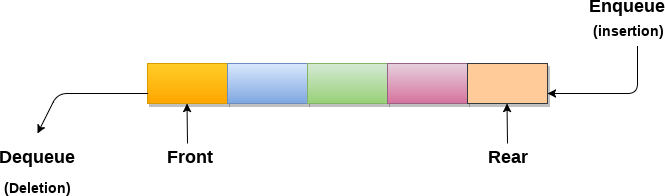
# Q)TYPES OF QUEUES:

A Queue data structure has been classified into the following types:

* + Simple Queue
  + Circular Queue
  + Priority Queue and
  + DeQueue( Double-Ended Queue)

**Simple Queue:-**

A Queue is a linear data structure, and the elements are stored in a sequential order. Queue follows FIFO (First in First Out) principle. In which the element which is inserted first will be removed first from the queue.



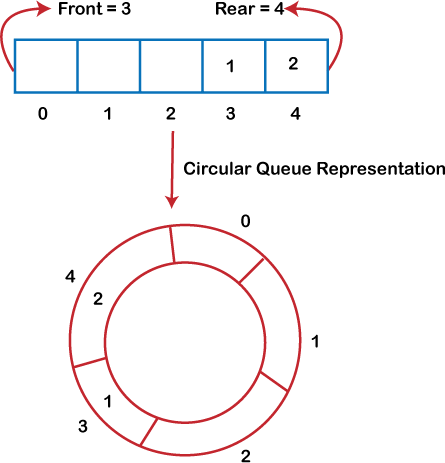
The elements in a queue are added at one end called the REAR and elements removed from the other end is called FRONT.

**Circular Queue**

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue, then there can be some spaces are left at the beginning of the queue which cannot be utilized. This can be resolved in two ways.

First one is, shift all the elements to left so that the vacant space can be available at rear end and elements can be inserted. But this this will take more time if we have large size of queues.

The second one is, to use a Circular Queue. In Circular queue, the first index comes right after last index.



A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a *Ring Buffer*.

**Operations on Circular Queue:**

The following are the operations that can be performed on a circular queue:

**Front:** It is used to get the front element from the Queue.

**Rear:** It is used to get the rear element from the Queue.

**enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end. **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

The circular queue will be full only when Front=0 and Rear=MAX-1. Circular queue can also be implemented in the same manner as linear queue. But the code can differ.

**Enqueue operation on Circular Queue:** Before inserting an element into circular queue we have to check the below three conditions:

* + If front=0 and rear=MAX-1, then circular queue is full, and no more insertions can be done.
  + If rear!=MAX-1, then rear will be incremented and the value will be inserted at the rear position.
  + If front!=0 and rear=MAX-1, then it means the queue is not full, so set rear=0 and insert the new element at the rear position.

We will use the below algorithm to insert an element into circular queue: Step 1: IF front=0 and rear=MAX-1 then

Write " OVERFLOW "

Goto step 4 [End OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE

SET REAR = (REAR + 1) % MAX

[END OF IF]

Step 3: SET QUEUE[REAR] = VAL Step 4: EXIT

In step 1, we check for the OVERFLOW condition. In step 2, we make two checks. First to see if the queue is empty and second to see if the REAR end has already reached maximum storage while there are certain locations before the FRONT end. In step 3, the value is stored at the rear position of the queue.

**Dequeue Operation on Circular Queue:** Before deleting an element from circular queue we have to check the below three conditions:

* + If front = -1 then, there are no elements in the queue. So, the underflow will be reported.
  + If front! =-1 and front=rear, then after deleting the element at the front, the becomes empty and front and rear are set to -1.
  + If front! =-1 and front = MAX-1, then after deleting the element at the front, front is set to 0 (ZERO).

We will use the below algorithm to delete an element from circular queue.

Step 1: IF FRONT = -1

Write " UNDERFLOW "

Goto Step 4 [END of IF]

Step 2: SET VAL = QUEUE[FRONT] Step 3: IF FRONT = REAR

SET FRONT = REAR = -1

ELSE

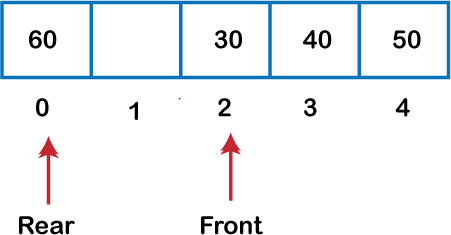
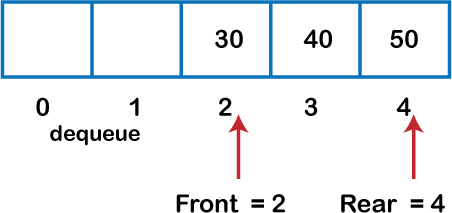
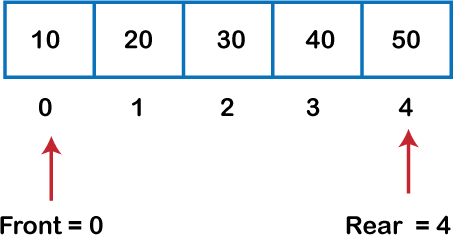
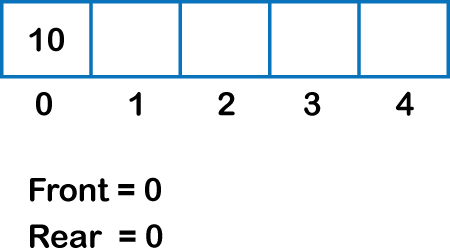
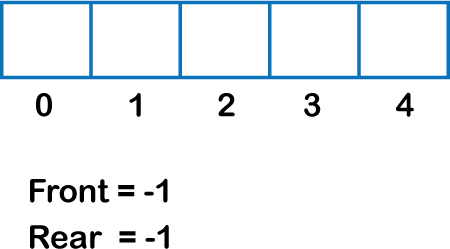
IF FRONT = MAX -1 SET FRONT = 0

ELSE

SET FRONT = FRONT + 1

[END of IF] [END OF IF]

Step 4: EXIT

Let's understand the enqueue and dequeue operation through the

diagrammatic representation.

**Q)Applications of Circular Queue:**

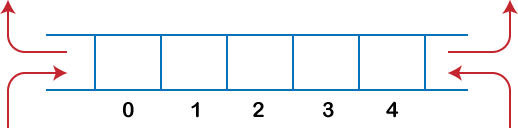
The circular Queue can be used in the following Applications:

**Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.

**CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.

**Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every interval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

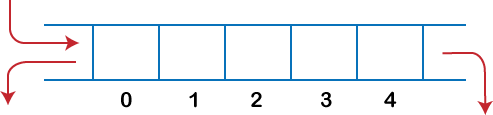
**Deque ( Double Ended Queues)**

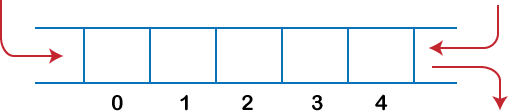
 A deque pronounced as deck or dequeue. The dequeue stands for Double Ended Queue. Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. There are two types of Queues:

* Input-restricted dequeue.
* Output-restricted dequeue.

**Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



**Deletion Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.

# 4.Priority Queue:

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority. In which the highest priority element will come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

**Rules for the Priority Queue:**

* An element with higher priority is processed before an element with a lower priority.
* Two elements with same priority are processed on a First-Come-First Serve (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest priority will be removed first.

The priority of the element can be set based on various factors. Priority queues are used in operating system to execute the highest priority processes first.

**Types of Priority Queue:**

* **Min Priority Queue:** In min priority Queue minimum number of values gets the highest priority and lowest number of elements gets the highest priority.
* **Max Priority Queue:** Max priority Queue is the opposite of min priority Queue in it, maximum number value gets the highest priority and minimum number of value gets the minimum priority.

**Applications of Queue:**

Due to the fact, that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues as below.

* + Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
  + Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
  + Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
  + Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
  + Queues are used in operating systems for handling interrupts.

# Q) Stacks Vs Queues

|  |  |
| --- | --- |
| Stacks | Queues |
| Stack is a Linear Data structure it  follows LIFO mechanism | Queue is a Linear Data Structure it  follows FIFO mechanism |
| Stack allows insertions and  deletions only at one end. | Queue allows insertions at one end  and deletions at the other end. |
| Since insertions and deletions are only at one end of the Stack, so the elements can be removed in the  opposite order of Insertion. | Since insertion and deletions are performed both ends of queue, so the elements can be removed in  the same order of insertion. |
| Insertion operation is called PUSH  and deletion operation is called POP. | Insertion operation is called as  ENQUEUE and deletion operation is called DEQUEUE. |
| The most accessible is called TOP. | Insertion is possible at REAR end and Deletion is possible at FRONT  end |

**UNIT – IV:**

**BINARY TREES:**

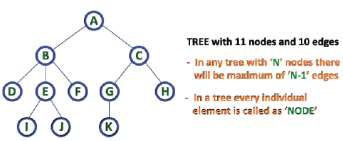
**Introduction to Non- Linear Data Structures, Introduction Binary Trees, Types of Trees, Basic Definition of Binary Trees, Properties of Binary Trees, Representation of Binary Trees, Operations on a Binary Search Tree, Binary Tree Traversal, Counting Number of Binary Trees, Applications of Binary Tree**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Q) WRITE A SHORT NOTE ON TREES:

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.

**Q) BASIC TERMINOLOGY**

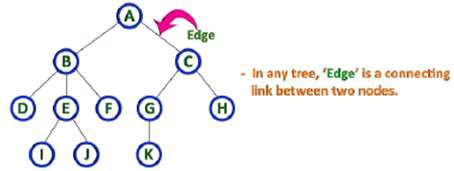
1. **Root**

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.



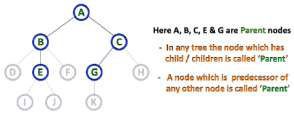
# Edge

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.



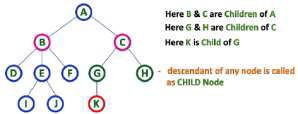
# Parent

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



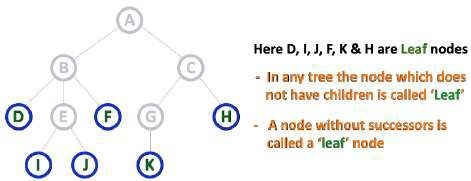
# Child

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



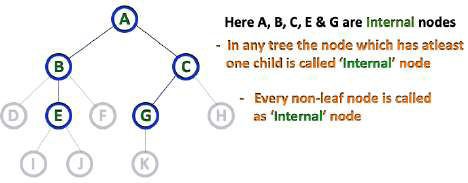
# Leaf

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.



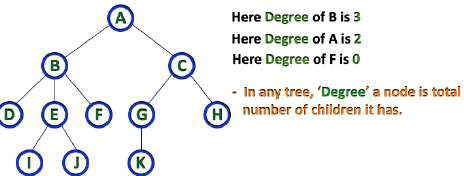
# Internal nodes

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.



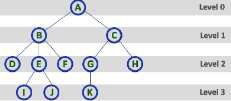
# Degree

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



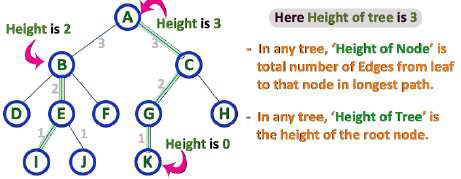
# Level

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.



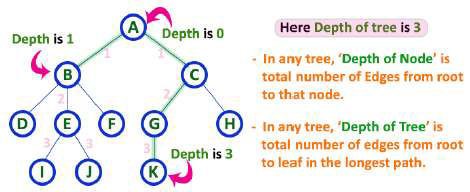
# Height

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’.

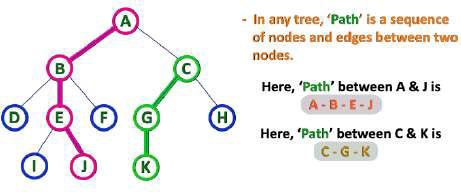


# Depth

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’.

# 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 AND APPLICATIONS 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.

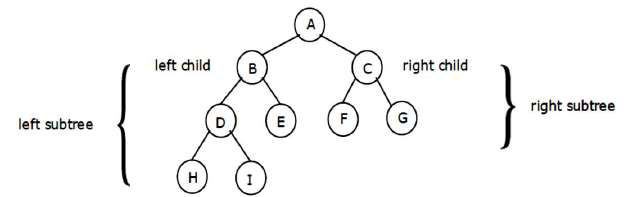


(or)

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*.**

**Note:**

* A Binary tree with n nodes has exactly n-1 edges.
* A Binary tree of depth d, has at least 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

# Q) Properties of binary trees

1. In any Binary tree, the maximum of nodes on level L is **2L**, where **L ≥ 0**
2. The maximum no. of nodes possible in a binary tree of height h is 2h+1 – 1.
3. The minimum no. of nodes possible in a binary tree of height h is h+1.
4. In any binary tree, it has n nodes and n-1 edges.
5. Every node except the root node has exactly one parent node.
6. There is exactly one path connecting any two nodes in tree.
7. The total number of leaf nodes in a binary tree is equal to the total no. of nodes with 2 child+1.
8. For any non-empty binary tree, if n is the no. of nodes and e is the no. of edges, then n=e – 1.
9. For any non-empty binary tree T, if is the no. of leaf nodes (degree=0) and is the no. of internal nodes (degree=2), then .
10. The height of a complete binary tree with n number of nodes is( )

**Q) 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.
* **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.

**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

# Array or Linear Representation

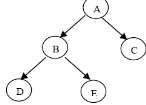
In Array (or linear) representation of binary tree, we use a 1-D array to represent a binary tree. In this representation, the nodes are stored level by level, starting from the level 0. The size of one dimensional array can be calculated by using following formula.

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

* 1. Root node is stored at position 0.
  2. If the node is stored at position “ i ” then its left child node is stored at position 2\*i+1 and its right child node is at 2\*i+2.

Example:

Consider the following incomplete binary tree:



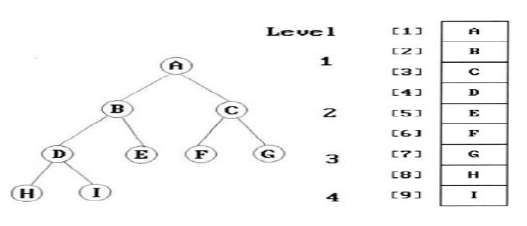
Here, in the above tree root node position is i=0, so A is store in 0 position in array.

All the elements are stored in array as follow:

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | -- | -- |

**2.Linked Representation of a Binary Tree**

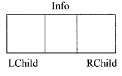


We can easily overcome the problems arise due to array representation by using a linked representation. Each node has three fields, left\_child, data, and right\_child.

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:

1. Left Child (LChild)
2. Information of the Node (Info)
3. Right Child (RChild)

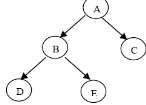
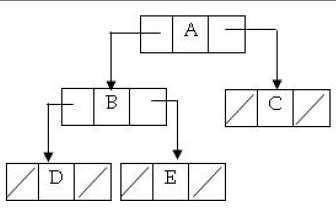
The LChild contains address of left child node and RChild contains address of right child node of the parent node. An info field contains actual data about node. When a node has no child then the corresponding pointer fields are NULL.



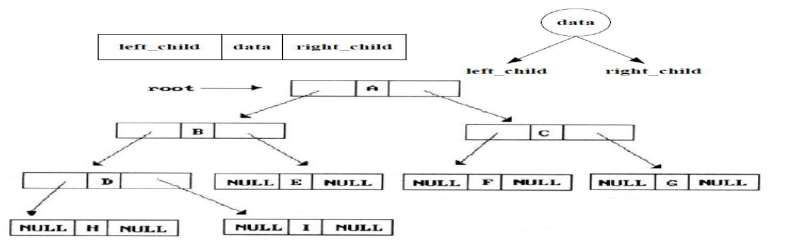
# Example:

Consider the following incomplete binary tree In linked representation, BELOW tree can be

represented as



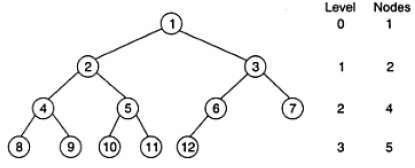
Example:



**Q) EXPLAIN BINARY TREE TYPES**

**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

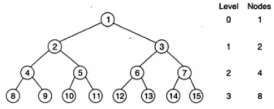
# Full binary tree:

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)

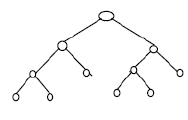
A full binary tree is a tree in which there is one node at root level 0 (20 = 1), two nodes at level 1(21

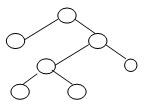
= 2), four nodes at level 2 (22 = 4) and so on.A full binary tree of height 4



# 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.



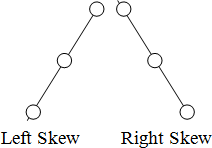


# Skewed Tree:

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

This 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.

# 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 sub tree, using preorder.
3. Visit the right sub tree, using preorder.

# Algorithm for preorder traversal

void preorder(node root)

{

if( root != NULL )

{

print root . data; preorder (root . lchild); preorder (root . rchild);

}

}

# 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);

}

}

# 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);

}

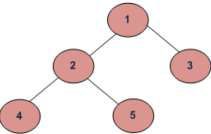
}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 order  RootLeftRight | in | the |
| 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 order  LeftRootRight | in | the |
| 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 order  LeftRightRoot | in | the |



**Example**

Consider the following tree



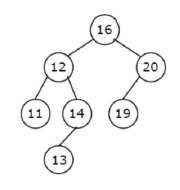
1. Inorder (Left, Root, Right): 4 2 5 1 3
2. Preorder (Root, Left, Right): 1 2 4 5 3
3. Postorder (Left, Right, Root): 4 5 2 3 1

**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 sub tree are smaller than the key in the root.
  + The keys in the right sub tree are larger than the key in the root.
  + The left and right sub trees 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

# 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

# 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

If data found

goto left subtree

return node end while

return data not found

end if



# 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

# In-order Traversal

* 1. **Pre-order Traversal**
  2. **Post-order Traversal**

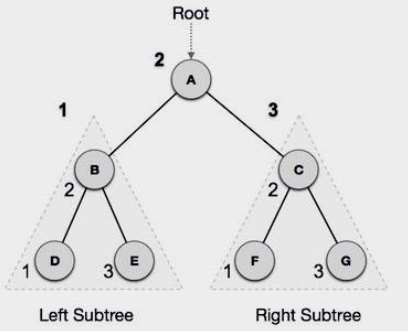
Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

# 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 –

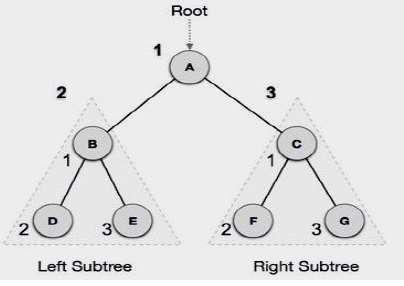
## D → B → E → A → F → C → G.

******

1. **Pre-order Traversal**

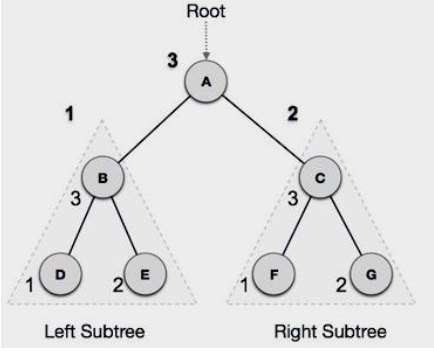
In this traversal method, the root node is visited first, then the left subtree and finally the right subtree. We start from **A**, and following pre-order traversal, we first visit **A**itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the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

## A → B → D → E → C → F → G



1. **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 → E → B → F → G → C → A

1. **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

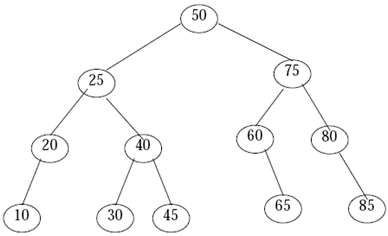
1. if the tree “null” print the message accordingly.
2. First, search for that starting from the root. i.e. move to left the element is less than parent or right till the element is encountered or empty node is encountered.
3. if empty node is encountered, print no element.
4. otherwise, if the node 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.



**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

# 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.

* 1. The tree must be searched to determine where the node is to be inserted.
  2. 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

* + 1. If (DATA < ROOT →Info)

Proceed to the left child of ROOT

* + 1. If (DATA > ROOT →Info)

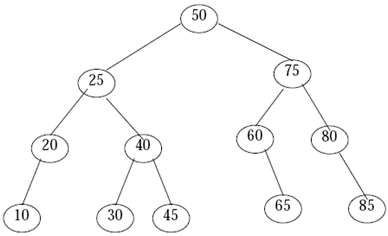
Proceed to the right child of ROOT

***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

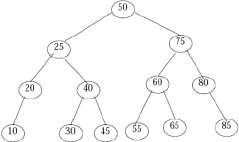
# For example:

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)
   1. ROOT=NEW NODE

 4. Else If (DATA < ROOTInfo)

* 1. ROOT = ROOT → Lchild
  2. GoTo Step 6

1. Else If (DATA > ROOT → Info)
   1. ROOT = ROOT → Rchild
   2. GoTo Step 7
2. If (DATA < ROOT → Info)
   1. ROOT → LChild = NEWNODE
3. Else If (DATA > ROOT → Info)
   1. ROOT → RChild = NEWNODE
4. NEW NODE → Info = DATA
5. NEW NODE → LChild = NULL
6. NEW NODE → RChild = NULL
7. EXIT

# SEARCHING A NODE

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

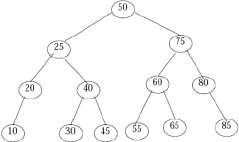
# ALGORITHM

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

# 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.

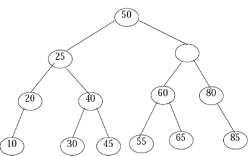
If we want to delete 75 from the tree, following steps are obtained:

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

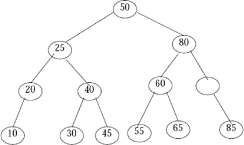
***Step* 2:** Compare the DATA with ROOT node, *i.e*., NODE, information of the tree. Since (50 < 75)

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.

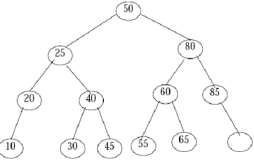


***Step* 4:** Since NODE (*i.e.,* node where value was 75) has both left and right child choose one. (Say Right Sub Tree) - If right sub tree is opted then we have to find the smallest node. But if left sub tree is opted then we have to find the largest node.

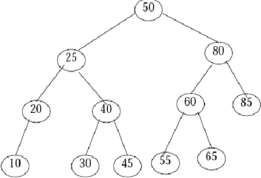


***Step* 5**: Find the smallest element from the right sub tree (*i.e.,* 80) and replace the node with deleted node.

***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.

**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

# 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

# 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.

# 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}

# 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}

# Lexicographic Order:

Arranging character or string data values into dictionary order is known lexicographic order. Eg: {Ant, Bat, Cat, Doll, Egg}

# 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.

# EXPLAIN IN DETAIL ABOUT SORTING AND 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:

# 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

# 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

# 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* ← 0 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

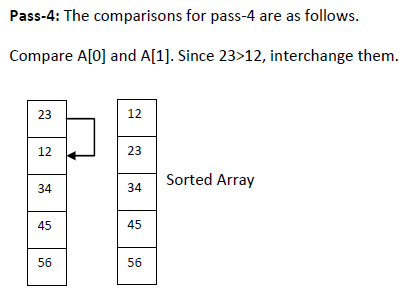
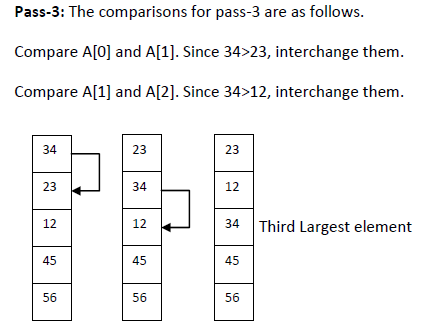
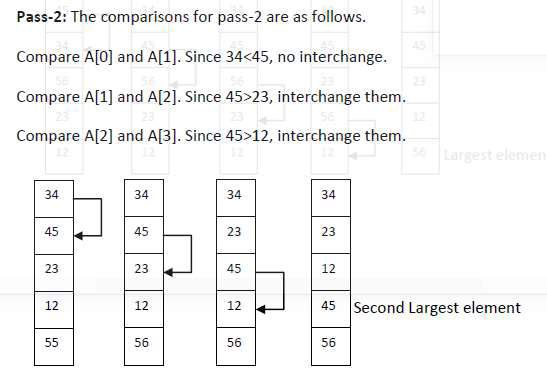
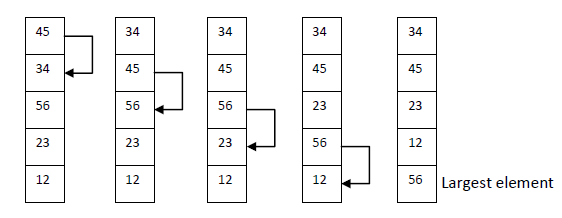
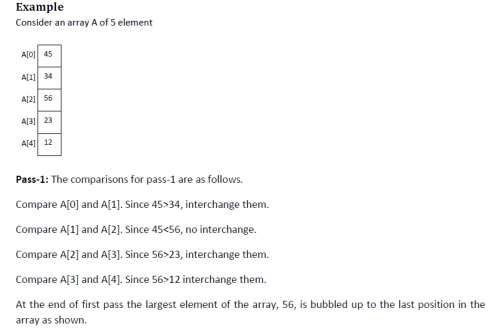
# Example:

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



**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

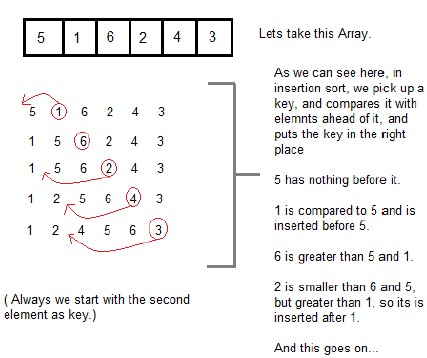
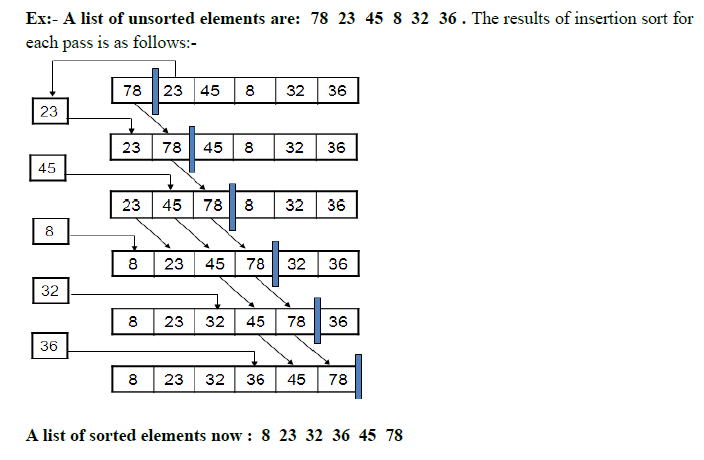
6. **DO** *A*[*i* +1+ ← *A*[*i*]

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.



# Example:

1. **MERGE SORT**

**M**erge 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*]:**

# 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*].

# Conquer Step

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

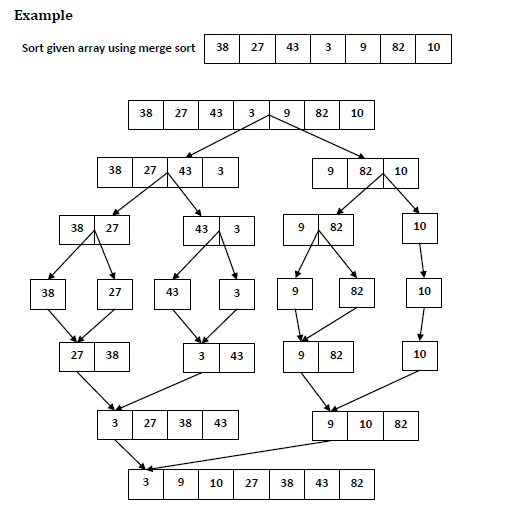
# Combine Step

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*).

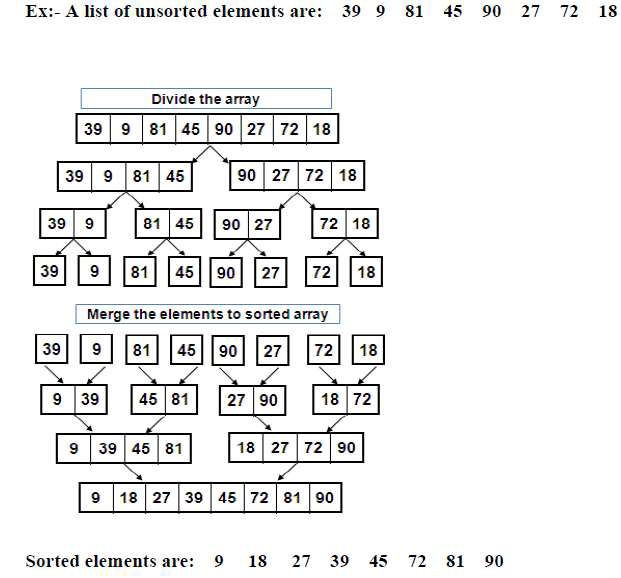
# MERGE-SORT (*A*, *p*, *r*)

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.









# 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

# 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 LINEAR Search:

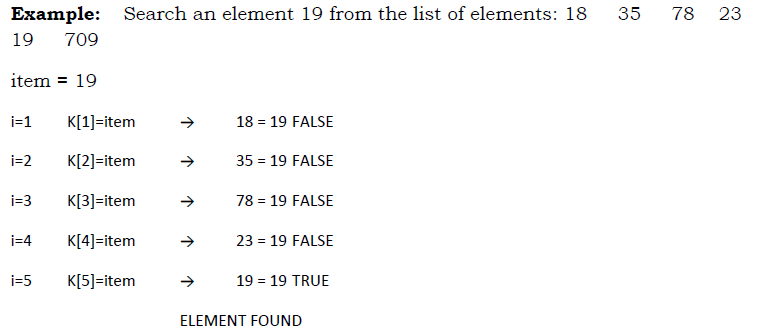
Suppose K is an array that contains n elements. Search element is given in the variable ‘item’. This function returns an index position ‘i' if the element is found; otherwise, return -1.

**Step 1:** Repeat for i ← 0 to n

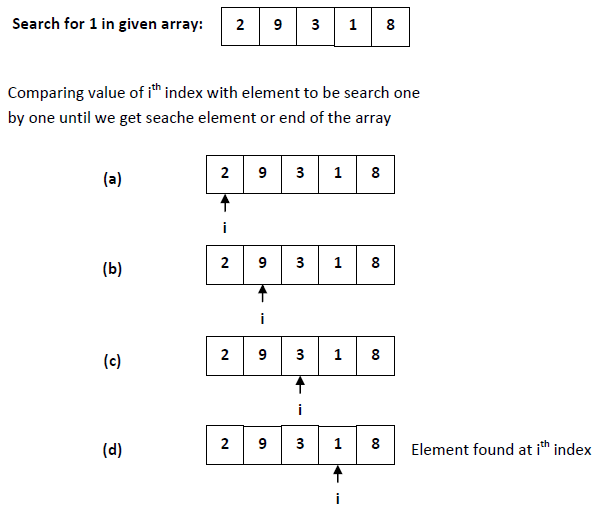
If K[i]=KEY Then Return i End If

End Repeat

**Step 2:** Return -1



# Example:





**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.

## How Binary Search Works?

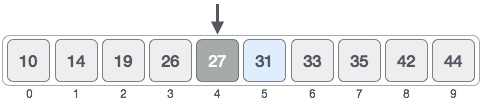
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula −

mid = low + high / 2

Here it is, 0 + 9 / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.

low = mid + 1

mid = low + high / 2

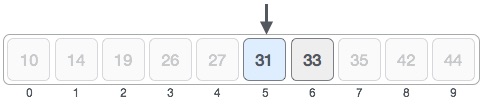
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



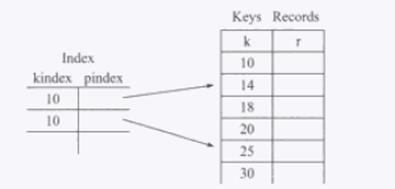
We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

**INDEXED SEQUENTIAL SEARCHING:-**

Another technique to improve search efficiency for a sorted file is indexed sequential search. But it involves an increase in the amount of memory space required. An auxiliary table called an **index**, is set aside in addition to a sorted file. Each element in the index table consists of a **key Kindex**and pointer to the record in the field that corresponds to the kindex. The elements in the index, as well as elements in the file, must be sorted on the file.

 The algorithm used for searching an indexed sequential file is simple and straight. Let **r, k** and **key**be defined as before. Let **kindex** be an array of the keys in the index, and let **pindex** be the array of pointers within the index to the actual records in the file, and the size of index is also taken in a variable. The indexed sequential search can be explained as the following example in the figure.



The advantage of the indexed sequential method is that items in the table can be examined sequentially if all records in the file have to be accessed, yet the search time for particular items is reduced. A sequential search is performed on the smaller index rather than on the large table. Once the index position is found, the search is made on the record table itself.

 Deletion from an indexed sequential table can be most easily by flagging deleted entries. When sequential searching is done deleted items are ignored. The item is deleted from the original table.

 Insertion into an indexed sequential table may be difficult as there may not be any place between two table entries which may lead to a shift to a large number of elements.

 However, the deleted items can be overwritten.

**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 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.

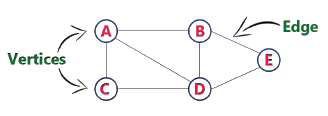
G=(V,E)

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.

# 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)

# 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.

# 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.

# End vertices or Endpoints

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

# Origin

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

# 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.

# Adjacent

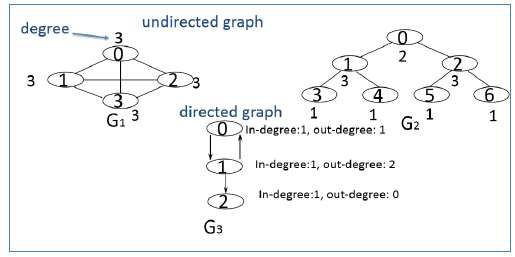
If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

# Incident

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

# Degree

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



# In-degree

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

# Out degree

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

# Self-loop

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

# Simple Graph (Digraph)

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

# 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.

# 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

# Isolated Graph

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

**Types OF GRAPHS:-**

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)

types of graphs:

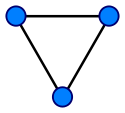
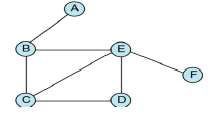
1. undirected graph

2.directed graph

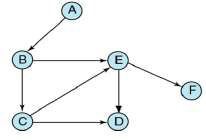
3.weighted graph

4.complete graph

**1. Undirected Graph:** A graph with only undirected edges is said to be undirected graph.



**2.Directed Graph:-**

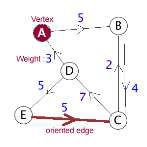


A graph with only directed edges is said to be directed graph. It is also called as Digraph.



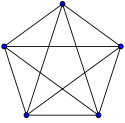
**3.Weighted Graph**

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



**4.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.

# REPRESENTATION OF GRAPHS

A graph data structure is represented using following representations

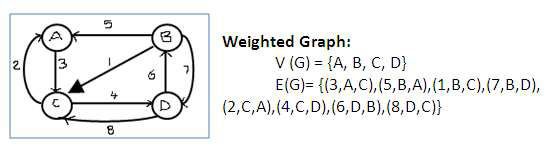
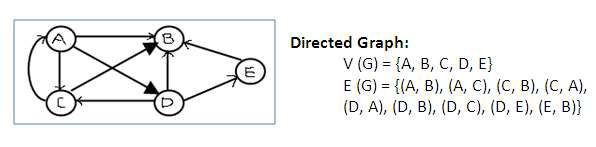
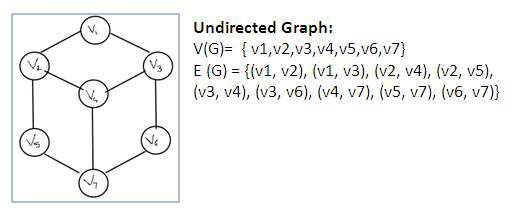
1. Set representation
2. Linked List representation / Adjacency List
3. Matrix representation / Adjacency Matrix / **Sequential Representation**

# 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.

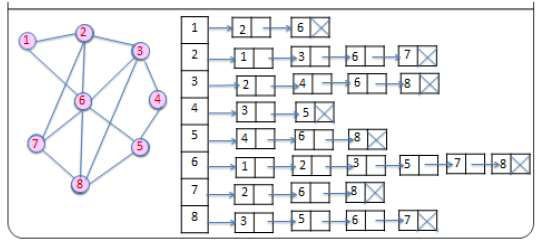


# Linked list Representation/ Adjacency List

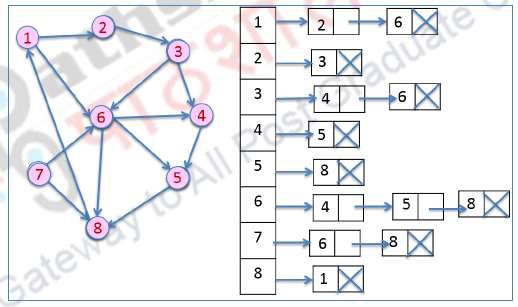
In this representation, we use two types of node structures. In each node, **data** field contains information and **link** field contains the address of the next adjacent node.



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.

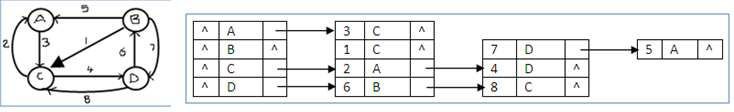


# Undirected graph



**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

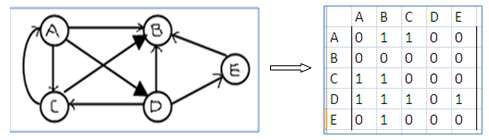
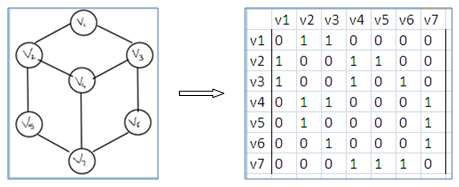
1. **Matrix Representation/ Adjacency Matrix / Sequential Representation:**

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.

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

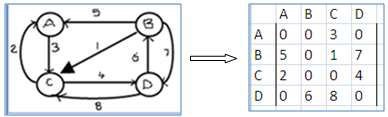
= 1, if there is an edge from to

0, if there is no edge from to



1. The Adjacency Matrix representation of a weighted graph is represented as:

=



, if there is an edge from to  . Where is weight 0, otherwise

# 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:

# Breadth First Search (BFS) 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.

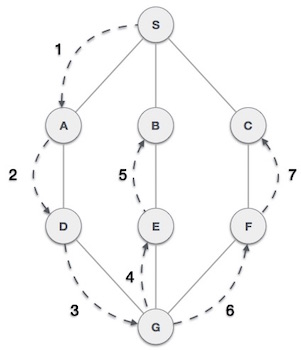
**Step 3:** Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

**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

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
* **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
* **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

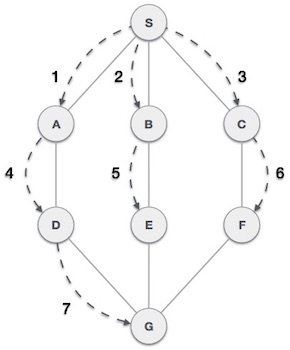
|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | Depth First Search Step One | Initialize the stack. |
| 2 | Depth First Search Step Two | Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. |
| 3 | Depth First Search Step Three | Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only. |
| 4 | Depth First Search Step Four | Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order. |
| 5 | Depth First Search Step Five | We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack. |
| 6 | Depth First Search Step Six | We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack. |
| 7 | Depth First Search Step Seven | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

# 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.

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
* **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
* **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | Breadth First Search Step One | Initialize the queue. |
| 2 | Breadth First Search Step Two | We start from visiting **S** (starting node), and mark it as visited. |
| 3 | Breadth First Search Step Three | We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it. |
| 4 | Breadth First Search Step Four | Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it. |
| 5 | Breadth First Search Step Five | Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it. |
| 6 | Breadth First Search Step Six | Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**. |
| 7 | Breadth First Search Step Seven | From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it. |

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

# 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.

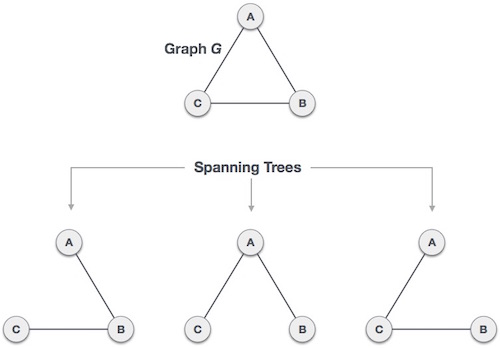
**Step 5:** Repeat step 3 and 4 until queue becomes empty.

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

# Q) DISCUSS ABOUT SPANNING TREES

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..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum **nn-2** number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence **33−2 = 3** spanning trees are possible.

## General Properties of Spanning Tree

We now understand that one graph can have more than one 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**.

## Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

## Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here −

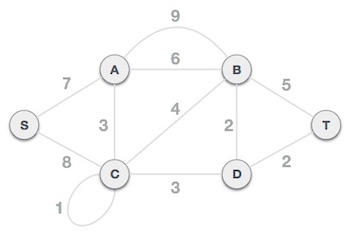
* [Kruskal's Algorithm](https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm)
* [Prim's Algorithm](https://www.tutorialspoint.com/data_structures_algorithms/prims_spanning_tree_algorithm.htm)

Both are greedy algorithms.

[**Kruskal's Algorithm**](https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm)

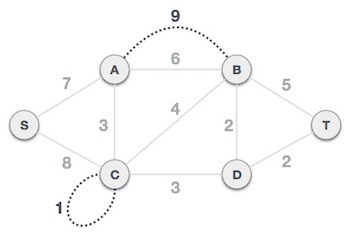
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −

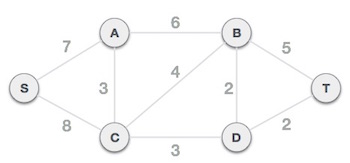


## Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.

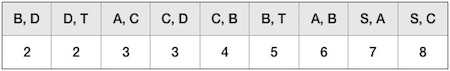


In case of parallel edges, keep the one which has the least cost associated and remove all others.



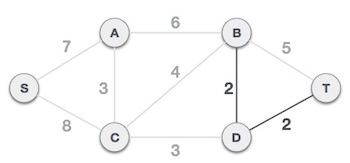
## Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).



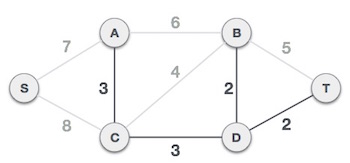
## Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

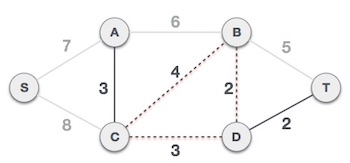


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

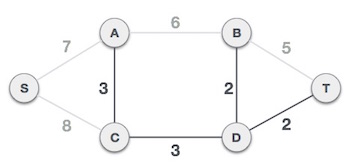
Next cost is 3, and associated edges are A,C and C,D. We add them again −



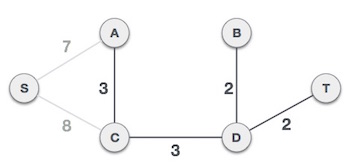
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



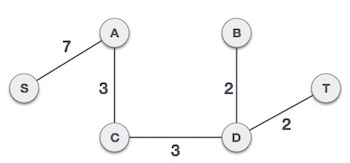
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



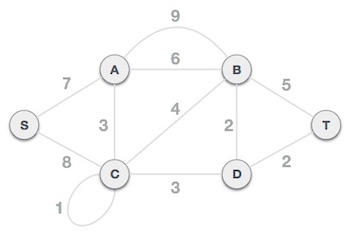
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

**Prim's algorithm**

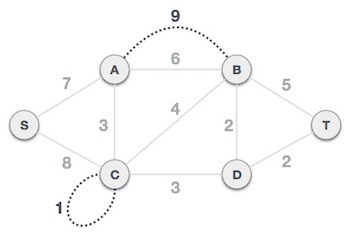
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

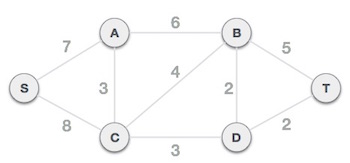
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



## Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

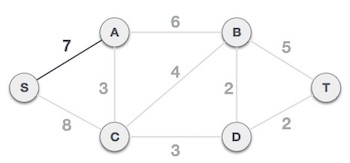


## Step 2 - Choose any arbitrary node as root node

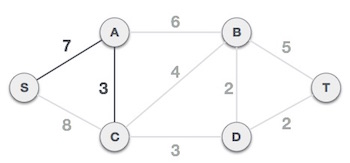
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

## Step 3 - Check outgoing edges and select the one with less cost

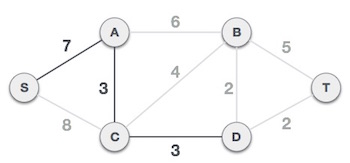
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



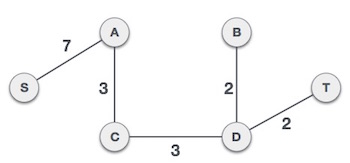
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

**Q) DISCUSS ABOUT SHORTEST PATH**:-

**SEE THE TEST PAPER PAGE NO: 196 & 208**

**Q) WRITE THE APPLICATIONS OF GRAPHS**

**SEE THE TEST PAPER PAGE NO: 197 &198**