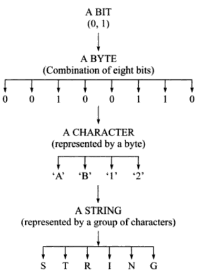
**UNIT – I**

**Introduction to the Theory of Data Structures**

* The study of computer science encompasses the study of organization and flow of data in a computer.
* 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 you to the data structures, but also helps you to understand and use the concept of abstraction, analyse problems step by step and develop algorithms to solve real world problems. It enables you to design and implement various data structures like stacks, queues, lists, etc.

**Data Representation**

* Various methods are used to represent data in computers.
* The basic unit of data representation is a bit. The value of a bit is either **0** or **1**.
* Various combinations of two values of a bit are used to represent data in a different manner in different systems.
* Eight bits together form one byte which represents a character and one or more than one characters are used to form a string.
* The representation of a string can be made easier by wrapping it into another data structure which takes care of such details and supports a set of operations that allows us to perform various string related operations like storing and fetching a string, joining two strings, finding the length of strings, etc.



1. **Integer Representation**

* An integer is the basic data type which is commonly used for storing negative as well as non-negative integer numbers. The non-negative data is represented using binary number system.
* In this, each bit position represents the power of 2.
* The rightmost bit position represents 20 which is 1, the next represents 21 which is 2, then 22 which is 4 and so on.

**For example**, 110 represents the integer as 1x 22 + 1x21 + 0 x 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.
* In two’s complement method, 1 is added to one’s complement representation of the negative number.

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

1 1 0 1 1 0 0 1

+ 1

-------------------

1 1 0 1 1 0 1 0 which represents -38 in two’s complement notation.

1. **Real Number Representation**

* The method used to represent real numbers in computers is floating point notation. In this notation, the real number is represented by a number called a mantissa, times a base raised to integer power can an exponent.

(mantissa)exponent

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

1. **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’.

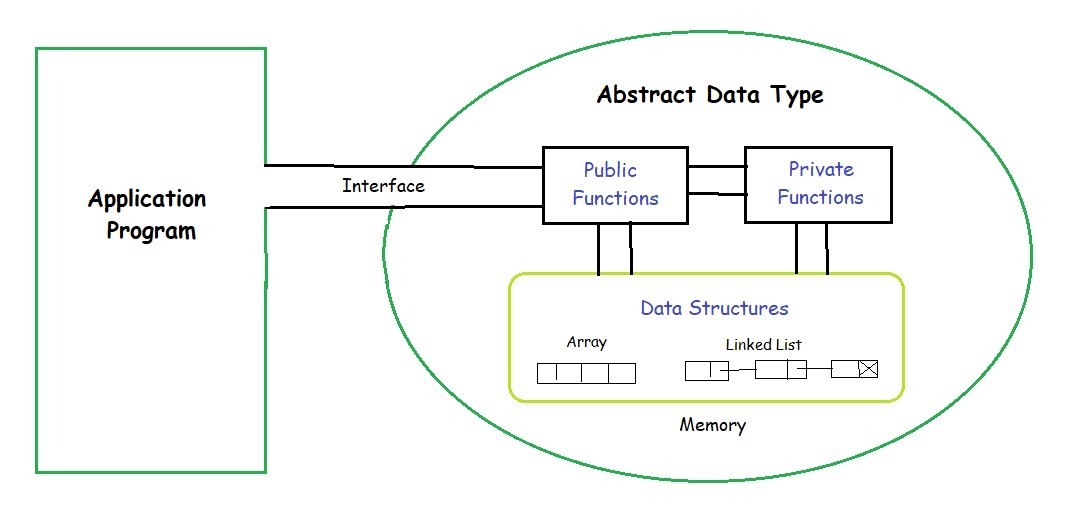
1. **Boolean**

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

**ADT**

* ADT stands for **Abstract Data Type.**
* It is an abstraction of a data structure.
* An ADT is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.
* ADT is the specification of logical and mathematical properties of a data type or structure.
* It describes a container which holds a finite number of objects where the objects may be associated through a given binary relationship.
* It is a logical description of how we view the data and the operations allowed without regard to how they will be implemented.
* ADT concerns only with what the data is representing and not with how it will eventually be constructed.
* It is a set of objects and operations. For example, List, Insert, Delete, Search, Sort.
* **It consists of following three parts:**

1. Data
2. Operation
3. Error
4. **Data** describes the structure of the data used in the ADT.
5. **Operation** describes valid operations for the ADT. It describes its interface.
6. **Error** describes how to deal with the errors that can occur.

[](https://media.geeksforgeeks.org/wp-content/uploads/20190828194629/ADT.jpg)

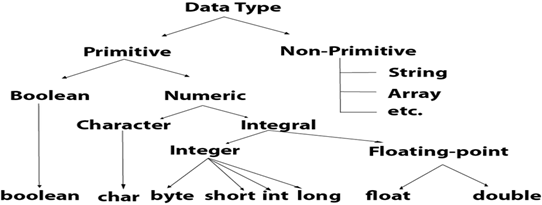
**Advantages of ADT**

* ADT is reusable and ensures robust data structure.
* It reduces coding efforts.
* Encapsulation ensures that data cannot be corrupted.
* ADT is based on principles of Object Oriented Programming (OOP) and Software Engineering (SE).
* It specifies error conditions associated with operations.

**DATA TYPE**

A data type can be defined as the type of values to be stored in a memory**.** There are two types of data types in C. They are -

* 1. Primitive Data type
  2. Non-primitive Data type



1. **Primitive Data type:**

C supports different types primitive data types: byte, short, int, long, float, double, char and Boolean. These data types are classified into four groups.

* 1. Integer
  2. Rational Numbers (Floating point)
  3. Characters
  4. Boolean

**Integer Type:**

Integer is the whole number without any fractional point. Integers may be either positive or negative. It can hold whole numbers such as 196, -52, 4036, etc. C supports four different types of integers, they are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Types** | **Range** | **Memory** |
| Integer | Signed int | -32,768 to 32767 | 2 bytes |
| Unsigned it | 0 to 65,535 | 2 bytes |
| Signed Long Int | -2147,483,648  to  2147,483,647 | 4 byte |
| Unsigned long int | 0 to 4,294,967,295 | 8 bytes |

**Rational Numbers:**

It is used to hold whole numbers containing fractional part such as 36.74, or -23.95 (which are known as floating point constants). Floating point number contains integer, decimal point (.) and exponent notation (e).

**Examples:** 20.05, 99.9, −50.12, 6.02 e23

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Types** | **Range** | **Memory** |
| Rational | Float | 3.4\*10-38 to 3.4\*10+37 | 4 bytes |
| Double | 1.7 \* 10-308 to 1.7 \* 10+307 | 8 bytes |

**Characters**

It is used to store character constants in memory. C provides a character data type called char whose type consumes a size of two bytes but can hold only a single character.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Types** | **Range** | **Memory** |
| Character | Signed Char | -128 0 to +127 | 1 byte |
| Unsigned char | 0 to 255 | 1 byte |

### Non-primitive data type

### Non-primitive data types are created by programmer. It is also called as 'Reference Variables' or 'Object reference' because it refers a memory location where data is stored.

### Non Primitive data types - Data types in Java - Edureka

1. **Strings:**

String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The *java.lang.String* class is used to create a string object.

1. **Arrays:**

Arrays are homogeneous data structures implemented in programming languages. Arrays store one or more values of a specific data type and provide indexed access to store the same. A specific element in an array is accessed by its index.

1. **Classes:**

A [class](https://www.edureka.co/blog/java-tutorial/#obj) is a blueprint which includes all your data.  A class contains fields(variables) and methods to describe the behavior of an object.

1. **Interface:**

Like a class, an *interface* can have methods and variables, but the methods declared in [*interface*](https://www.edureka.co/blog/java-collections/#interface) are by default abstract

**Data Structure**

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

(or)

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

**Structured type**

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**

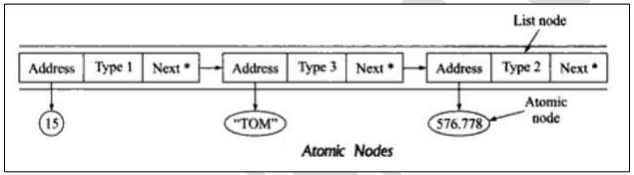
**Ans:** Generally, a data structure is represented by a memory block, which has two parts:

* 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

**Diff. between Abstract Data Types, Data Types, and Data Structures**

|  |  |  |
| --- | --- | --- |
| **ADT** | **DATA TYPES** | **DATA STRUCTURES** |
| 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
5. **Conceptual level**

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

1. **Algorithmic or Data structure Level**

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

1. **Programming or Implementation Level**

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

1. **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.

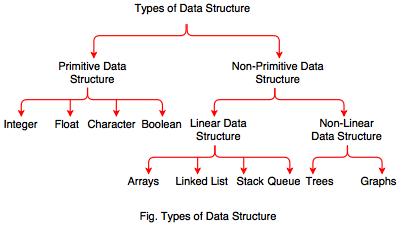
**Data Structure**

* Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently.
* Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.
* Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.
* It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible
* Data structure is an arrangement of data in computer's memory. It makes the data quickly available to the processor for required operations.
* It is a structure program used to store ordered data, so that various operations can be performed on it easily.
* Data structure is about providing data elements in terms of some relationship for better organization and storage.
* It is a specialized format for organizing and storing data that can be accessed within appropriate ways.
* In programming, the term structure refers to a scheme for organizing related pieces of information.
* The basic types of data structures include arrays, structures, stacks, queues, linked lists, trees, graphs, etc. Each of these basic structures has many variations and allows different operations to be performed on the data

**Types of Data Structure:**

**Data structure is classified into two types** based on their nature and construction.

1. Primitive Data Structures
2. Non-Primitive Data Structures



**Primitive Data Structures**

A primitive Data structures can be defined as the collection of pre-defined types of a language **such as integer, float, character, byte, Boolean** etc is also called as **Primitive Data Structure**. It is also called as **Fundamental**/**Elementary Data structure**. Primitive Data structures can be used to build array, structures etc. Fundamental or elementary data structures are primitive in nature and are the building blocks used for complex structures constitution.

**Non-Primitive Data Structures**

Non-primitive data structures are the data structures which are complex in nature and are constructed using the elementary data structures. For some operations on data, the elementary structures are not useful as well as efficient and in such cases, complex data structures can be preferred. It is also called as **complex data structure**. The Non-primitive data structures are classified into two types. They are

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

**Linear Data Structures**

In Linear Data Structures the data can be arranged in sequence. The Linear Data structures can be used **to build stacks, queues, lists, etc**.

**Non-Linear Data Structures**

In non-linear data structures the data can be arranged in random order. The non-linear data structures can be used to build **trees and graphs.**

**Meaning of Data Structure**

**Arrays:**

Array is elementary data structure which holds the same data values in the contiguous memory locations the elements of the array can be accessed sequentially / randomly.

**List:**

List is a data structure which supports the logical adjacency of elements rather than physical adjacency. This logical adjacency of elements is maintained using pointers.

**Stacks:**

A stack is a linear data structure, in which all insertions of elements takes place at one end and all deletions of elements takes place at other end.

**Queue:**

A queue is another special kind of list, where items are inserted at one end called the read and deleted at the other end called the front. Another name for a queue is a FIFO or first-in-first out List.

**Graphs:**

Graph is a data structure mainly used in the application such as finding shortest routes/paths, analysis of electrical circuits, chemical compounds etc. This data structure consists of a finite number of non-empty set of vertices as well as edges.

**Trees:**

Tree is a data structure which is helpful in variety of computer applications. A tree is a data structure consisting of one or more nodes such that there is a single root node and other child nodes.

**Questions**

**10 Marks**

* 1. Write about Data Representation
  2. Write about Data Structure and structured Type
  3. Write about Refinement Stages
  4. Write about ADT

**5 MARKS**

1. Write about Primitive Data types
2. Write the difference b/w data types, ADT and data structures.
3. Write about Atomic type.

**UNIT – II**

**Linear Data Structures**

* It is a non-primitive Data Structure
* In Linear data structures the data can be arranged in sequence.
* A linear data structure contains only one starting point and one ending point
* There are two ways to represent a linear data structure in memory. They are

1. Static memory allocation
2. Dynamic memory allocation

* The Linear Data structures can be used **to build stacks, queues, lists, etc**.

**Non-Linear Data Structures**

* It is a non-primitive Data Structure
* In non-linear data structures the data can be arranged in random order.
* A non-linear data structure contains several starting and ending points or no definite starting and ending points.
* The non-linear data structures can be used to build **trees and graphs.**

**Arrays**

A collection of values of similar types which can be stored under one variable or heading it is called as an array. It is also called as homogenous type.

**TYPES OF ARRAY:**

Arrays are of different types. They are –

1. Single Dimension Array
2. Double Dimension Array
3. Multi Dimension Array

**SINGLE DIMENSION ARRAY:**

An array which contains only one row and it can have set of columns. It is called as a single dimension array. It is also called as **one dimensional array** or **single subscripted value array**. Array position always starts from **“zero”**. Each section of an array is called as an **“element”**.

## Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [ arraySize ];

## eg: int a[4];

char s[10];

## Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows –

**Initialization of values in an array** –

Eg: - 1. int a[4]= { 10,20,30,40 };

a[0] = 10

a[1] = 20

a[2] = 30

a[3] = 40

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

2. int x[] ={10,20,30,40 };

x[0] = 10

x[1] = 20

x[2] = 30

x[3] = 40

**Accessing array elements:**

Once array is defined, its elements can be accessed by using an index.

**Syntax:** array\_name [index];

**Example:** scanf(“%d”, &arr[1]);

printf(“%d”, arr[1]);

**DOUBLE DIMENSION ARRAY:**

An array which contains number of rows and number of columns, it is called as a double dimension array. It is also called as Double subscripted value array.

***Syntax:*** Datatype var[rs][cs];

**Eg:- 1**. int a[3][3];

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

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

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

**Initialization of values in array –**

Int a[][]= {{10,20,30}, {40,50,60}, {70,80,90}}

a[0][0]=10 a[0][1]=20 a[0][2]=30

a[1][0]=40 a[1][1]=50 a[1][2]=60

a[2][0]=70 a[2][1]=80 a[2][2]=90

**Multi-Dimensional array:**

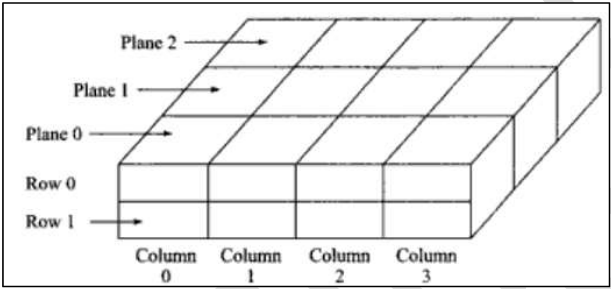
The array can also have more than two dimensions. For example, a three-dimensional array may be declared as

int arr[3][2][4]

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

**For example**, arr contains 3\*2\*4=24 elements.

The arr[3][2][4] can be represented as shown below

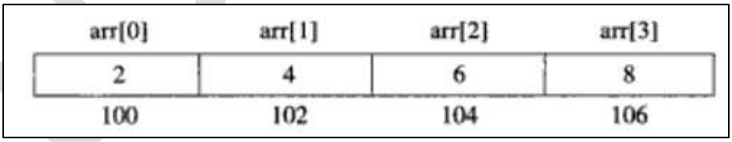


**Pointers and Arrays**

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

**For example**, int arr[4]={2,4,6,8}

The element of an array is declared is as follows



The name arr acts as a constant pointer pointing to the first element,i.e. arr[0] of the array. Therefore, the value of arr is equal to the address where arr[0] is stored,i.e. arr =&arr[0]=100

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

ip=arr

ip=&arr[0]

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

ip =&arr[0] (=100)

ip +1 =&arr[1] (=102)

ip+2 =&arr[2] (=104)

so, the address of many element can be calculated by using its index and the size of the data type.

**For example,**

Address of arr[2] = base address + (2\* size of int) =100 + (2\*2) =104

**Overview of Pointers**

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte

**POINTER OPERATORS**

1. **\* : Indirection Operator / Value at address operator**
2. **&: Address operator**

**Declaring a pointer**

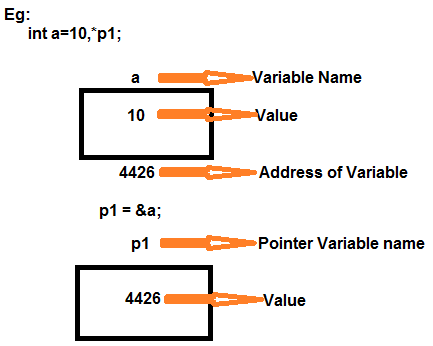
The pointer in c language can be declared using \* (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

**Syn: datatype \*var\_name;**

**Eg: int \*p1;**

**float \*p2;**

**char \*p3;**

****

**Note:** By the help of \* (**indirection operator**), we can print the value of pointer variable p.

# Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

* Increment
* Decrement
* Addition
* Subtraction

eg-1: int a,\*p1;

p1 = &a

= 100 + 1 => 102

= 100 + 2 => 104

= 100 - 1 => 98

## eg-2: float x,\*p2;

## p2= &x;

## = 200 + 1 => 204

## = 200 + 2 => 208

## = 200 - 1 => 196

## eg -3: char c,\*p3;

## p3= &c;

## = 300 + 1=>301

## = 300 + 2=>302

## = 300 - 1=>299

## Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

**The Rule to increment the pointer is given below:**

**new\_address= current\_address + i \* size\_of(data type)**

**Where i is the number by which the pointer get increased.**

### 32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

### 64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

**Pointer using structure**

A collection of values of different types which can store under one variable or heading, it is called as a structure. The members of structure can be accessed by defining either simple structure variable or pointer structure variable. If we want to access the members of structure with pointer structure variable we should also make use of "->" symbol

**syn:**

struct struct\_name

{

member1;

member2;

member3;

member-n;

};

struct struct\_name \*var;

var->member1=value (\*var).member1=value

var->member2=value (\*var).member2=value

var->member3=value or (\*var).member3=value

var->member-n=value (\*var).member-n=value

**Call by reference:**

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types

**List**

**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 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.

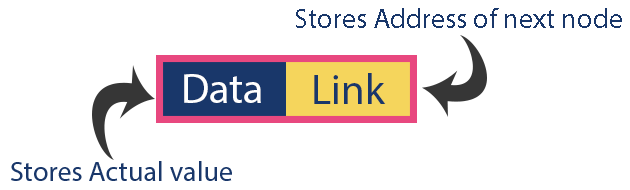
**List implementation**

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;

}

**Types of Linked List:**

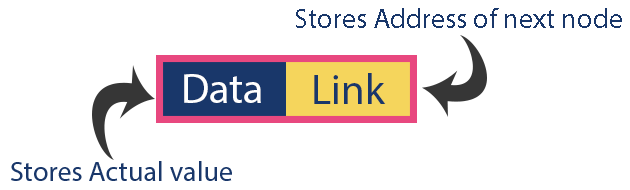
Linked List are of different types. They are

* 1. Singly Linked List
  2. Doubly Linked List
  3. Circular Linked List
  4. Doubly circular Linked List

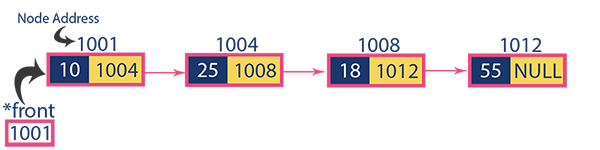
**Singly Linked List**

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

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



**Example**



The last node in the list contains a **NULL pointer** to indicate that it is the end or tail of the list. NULL pointer is commonly point to an invalid address. Linked list also contains a pointer variable, called Head, which represent the list is simply a pointer to the first node of the list. The HEAD variable is not exactly part of list but it is useful as a handle for the list. As items are added to a list, memory for a node is dynamically allocated.

**Operations**

In a single linked list we perform the following operations.

1. Insertion
2. Deletion
3. Display
4. **Insertion**

In a single linked list, the insertion operation can be performed in three ways. They are as follows.

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list
4. **Inserting At Beginning of the list**

We can use the following steps to insert a new node at the beginning of list

**Steps:**

1. Create a newNode with given value.
2. Check whether list is Empty (head == NULL)
3. If it is Empty then, set newNode→next = NULL and head = newNode.
4. If it is Not Empty then, set newNode→next = head and head = newNode.

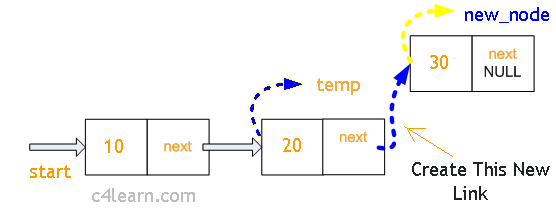


1. **Inserting at the End of the list**

We can use the following steps to insert a new node at end of the single linked list.

**Steps**

1. Create a newNode with given value and newNode → next as NULL.
2. Check whether list is Empty (head == NULL).
3. If it is Empty then, set head = newNode.
4. If it is Not Empty then, define a node pointer temp and initialize with head.
5. Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
6. Set temp → next = newNode.

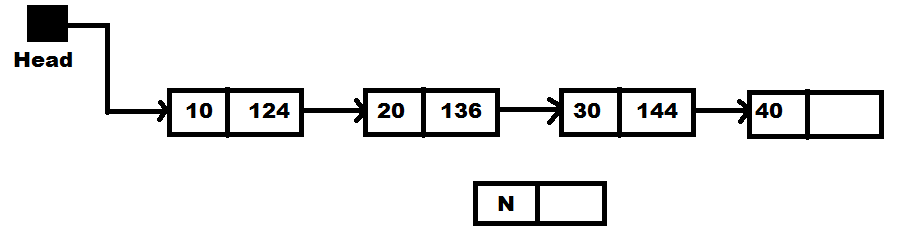


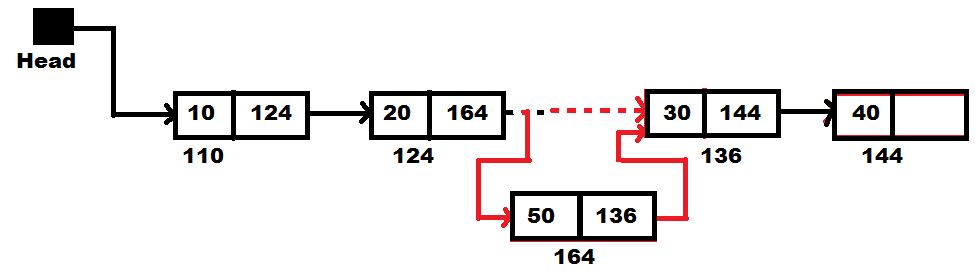
1. **Inserting at Specific location in the list**

We can use the following steps to insert a new node after a node in the single linked list.

**Steps:**

1. Create a newNode with given value.
2. Check whether list is Empty (head == NULL)
3. If it is Empty then, set newNode → next = NULL and head = newNode.
4. If it is Not Empty then, define a node pointer temp and initialize with head.
5. Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode
6. Every time check whether temp is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!!** Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.
7. Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'





1. **Deletion**

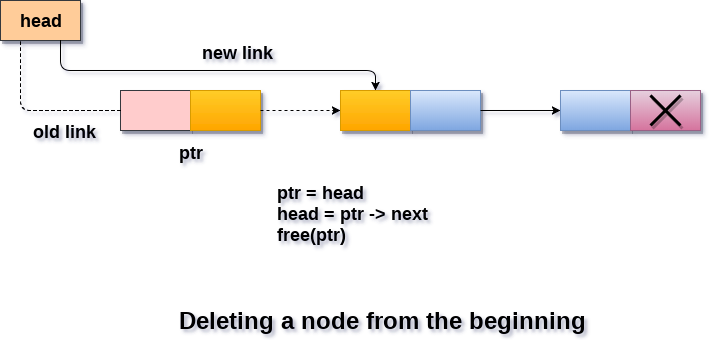
In a single linked list, the deletion operation can be performed in three ways. They are as follows.

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node
4. **Delete a node from beginning of the list:**

We can use the following steps to delete a node from beginning of the single linked list.

**Steps:**

1. Check whether list is Empty (head == NULL)
2. If it is Empty then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
3. If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
4. Check whether list is having only one node (temp → next == NULL)
5. If it is TRUE then set head = NULL and delete temp If it is FALSE then set head = temp → next, and delete temp.

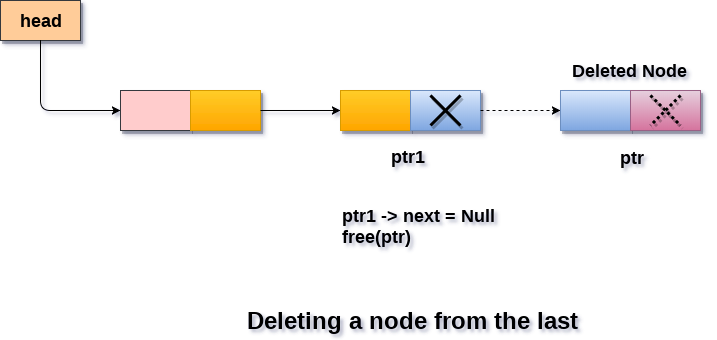


1. **Deleting from the End of the list**

We can use the following steps to delete a node from end of the single linked list.

**Steps:**

1. Check whether list is Empty (head == NULL)
2. If it is Empty then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
3. If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
4. Check whether list has only one Node (temp1 → next == NULL)
5. If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function.
6. If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)
7. Finally, Set temp2 → next = NULL and delete temp1.

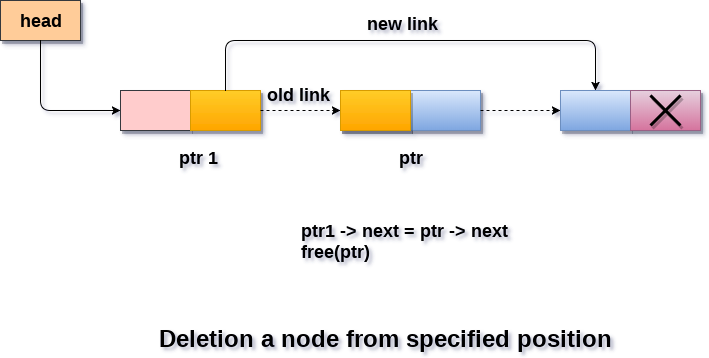


1. **Deleting a Specific Node from the list**

We can use the following steps to delete a specific node from the singly linked list.

**Steps:**

1. Check whether list is Empty (head == NULL)
2. If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
3. If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
4. Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
5. If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
6. If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
7. If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
8. If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
9. If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
10. If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).
11. If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
12. If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).



1. **Displaying a Singly Linked List**

We can use the following steps to display the elements of a singly linked list.

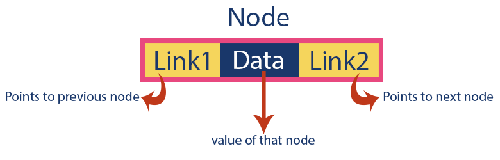
**Steps:**

1. Check whether list is Empty (head == NULL)
2. If it is Empty then, display 'List is Empty!!!' and terminate the function.
3. If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
4. Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
5. Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

**Doubly Linked List**

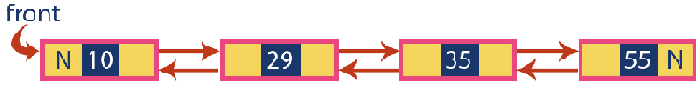
Doubly linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In doubly linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure.



Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

**Example**



**Operations:**

In a doubly linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display
4. **Insertion**

In a doubly linked list, the insertion operation can be performed in three ways as follows.

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list
4. **Inserting At Beginning of the list**

We can use the following steps to insert a new node at beginning of the double linked list.

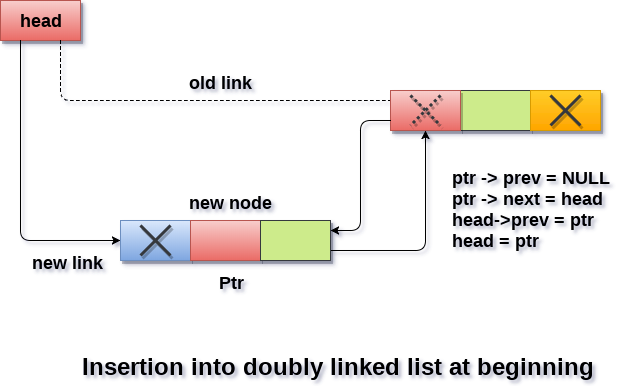
**Steps:**

1. IF ptr = NULL

Write OVERFLOW

Go to Step 9  
[END OF IF]

1. SET NEW\_NODE = ptr
2. SET ptr = ptr -> NEXT
3. SET NEW\_NODE -> DATA = VAL
4. SET NEW\_NODE -> PREV = NULL
5. SET NEW\_NODE -> NEXT = START
6. SET head -> PREV = NEW\_NODE
7. SET head = NEW\_NODE
8. EXIT



1. **Inserting At End of the list**

We can use the following steps to insert a new node at end of the double linked list.

**Steps:**

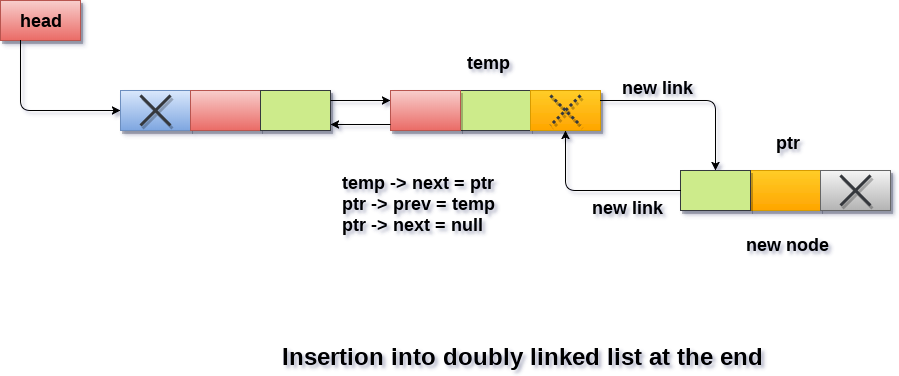
1. IF PTR = NULL

 Write OVERFLOW  
   Go to Step 11  
  [END OF IF]

1. SET NEW\_NODE = PTR
2. SET PTR = PTR -> NEXT
3. SET NEW\_NODE -> DATA = VAL
4. SET NEW\_NODE -> NEXT = NULL
5. SET TEMP = START
6. Repeat Step 8 while TEMP -> NEXT != NULL
7. SET TEMP = TEMP -> NEXT

[END OF LOOP]

1. SET TEMP -> NEXT = NEW\_NODE
2. SET NEW\_NODE -> PREV = TEMP
3. EXIT



1. **Inserting At Specific location in the list**

We can use the following steps to insert a new node after a node in the double linked list.

**Steps:**

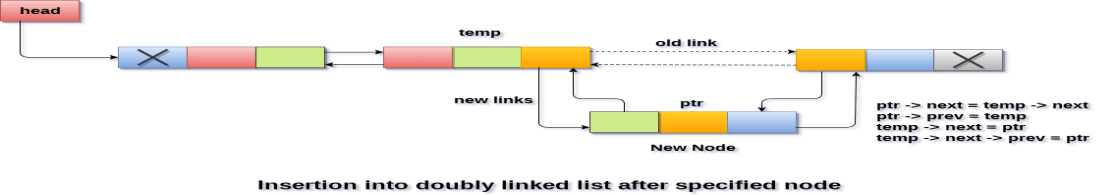
1. IF PTR = NULL

Write OVERFLOW  
   Go to Step 15  
  [END OF IF]

1. SET NEW\_NODE = PTR
2. SET PTR = PTR -> NEXT
3. SET NEW\_NODE -> DATA = VAL
4. SET TEMP = START
5. SET I = 0
6. REPEAT 8 to 10 until I
7. SET TEMP = TEMP -> NEXT
8. IF TEMP = NULL
9. WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

GOTO STEP 15  
    [END OF IF]   
  [END OF LOOP]

1. SET NEW\_NODE -> NEXT = TEMP -> NEXT
2. SET NEW\_NODE -> PREV = TEMP
3. SET TEMP -> NEXT = NEW\_NODE
4. SET TEMP -> NEXT -> PREV = NEW\_NODE
5. EXIT



1. **Deletion**

In a double linked list, the deletion operation can be performed in three ways as follows.

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node
4. **Deleting from Beginning of the list**

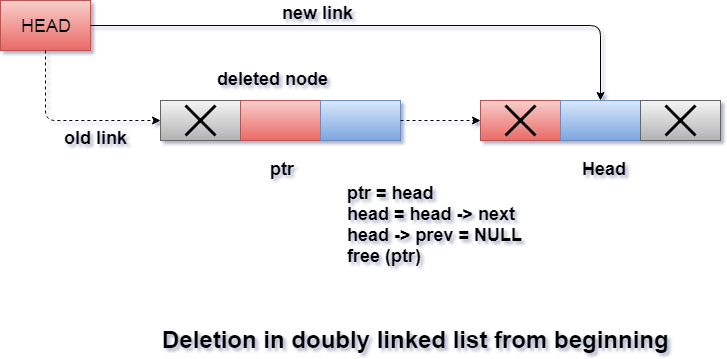
We can use the following steps to delete a node from beginning of the double linked list.

**Steps:**

1. IF HEAD = NULL

WRITE UNDERFLOW  
GOTO STEP 6

1. SET PTR = HEAD
2. SET HEAD = HEAD → NEXT
3. SET HEAD → PREV = NULL
4. FREE PTR
5. EXIT



1. **Deleting from End of the list**

We can use the following steps to delete a node from end of the double linked list.

**Steps:**

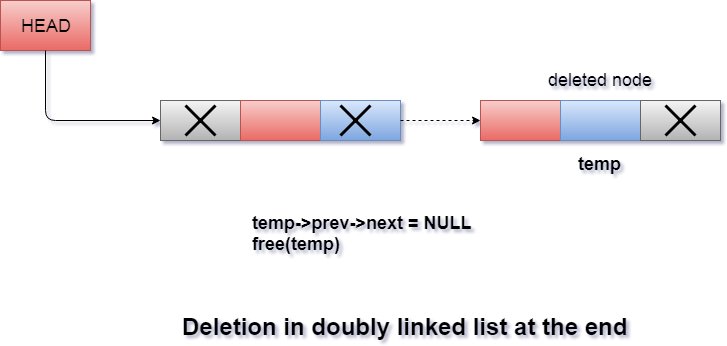
1. IF HEAD = NULL

Write UNDERFLOW  
 Go to Step 7  
 [END OF IF]

1. SET TEMP = HEAD
2. REPEAT STEP 4 WHILE TEMP->NEXT != NULL
3. SET TEMP = TEMP->NEXT

[END OF LOOP]

1. SET TEMP ->PREV-> NEXT = NULL
2. FREE TEMP
3. EXIT



1. **Deleting a Specific Node from the list**

We can use the following steps to delete a specific node from the double linked list.

**Steps:**

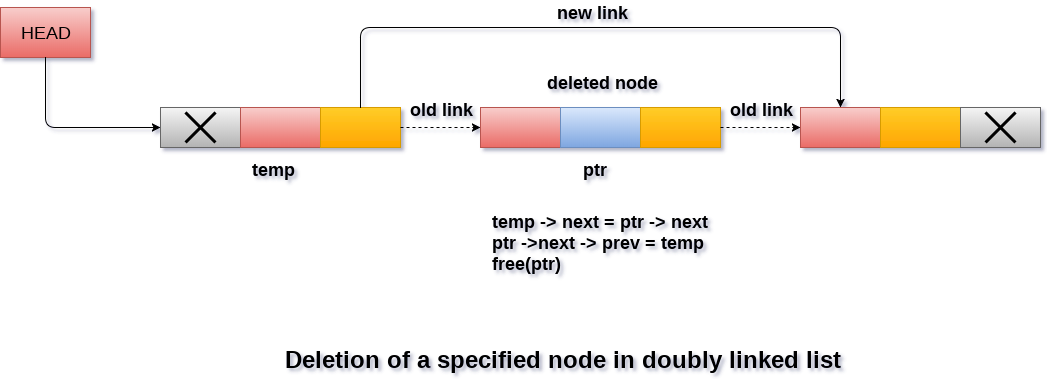
1. IF HEAD = NULL

Write UNDERFLOW  
   Go to Step 9  
  [END OF IF]

1. SET TEMP = HEAD
2. Repeat Step 4 while TEMP -> DATA != ITEM
3. SET TEMP = TEMP -> NEXT

[END OF LOOP]

1. SET PTR = TEMP -> NEXT
2. SET TEMP -> NEXT = PTR -> NEXT
3. SET PTR -> NEXT -> PREV = TEMP
4. FREE PTR
5. EXIT



1. **Displaying a Double Linked List**

We can use the following steps to display the elements of a double linked list.

**Steps:**

1. IF HEAD == NULL

WRITE "UNDERFLOW"  
 GOTO STEP 6  
 [END OF IF]

1. Set PTR = HEAD
2. Repeat step 4 and 5 while PTR != NULL
3. Write PTR → data
4. PTR = PTR → next
5. Exit

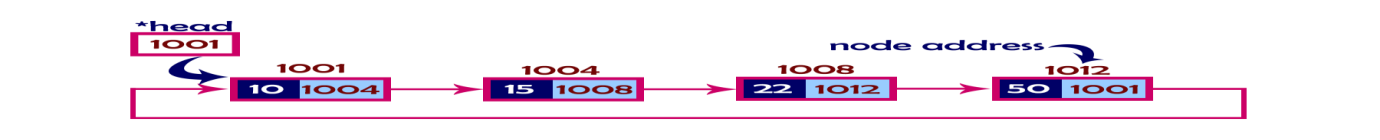
**Circular Singly Linked List**

In circular linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

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

That means circular linked list is similar to the singly linked list except that the last node points to the first node in the list

**Example**



**Operations:**

In a circular linked list, we perform the following operations.

1. Insertion
2. Deletion
3. Display
4. **INSERTION**

In a circular linked list, the insertion operation can be performed in three ways. They are as follows.

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list
4. **Inserting At Beginning of the list**

We can use the following steps to insert a new node at the beginning of the circular linked list.

**Steps:**

1. IF PTR = NULL

Write OVERFLOW

Go to Step 11

[END OF IF]

1. SET NEW\_NODE = PTR
2. SET PTR = PTR -> NEXT
3. SET NEW\_NODE -> DATA = VAL
4. SET TEMP = HEAD
5. Repeat Step 8 while TEMP -> NEXT != HEAD
6. SET TEMP = TEMP -> NEXT

[END OF LOOP]

1. SET NEW\_NODE -> NEXT = HEAD
2. SET TEMP → NEXT = NEW\_NODE
3. SET HEAD = NEW\_NODE
4. EXIT
5. **Inserting At End of the list**

We can use the following steps to insert a new node at end of the circular linked list.

**Steps:**

1. IF PTR = NULL

Write OVERFLOW

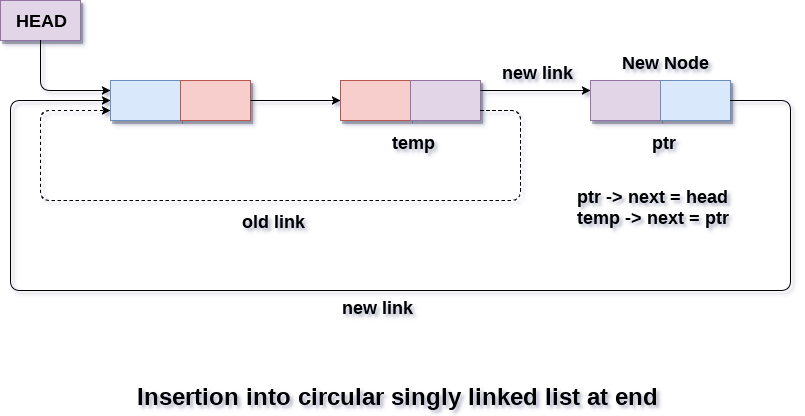
Go to Step 1

[END OF IF]

1. SET NEW\_NODE = PTR
2. SET PTR = PTR -> NEXT
3. SET NEW\_NODE -> DATA = VAL
4. SET NEW\_NODE -> NEXT = HEAD
5. SET TEMP = HEAD
6. Repeat Step 8 while TEMP -> NEXT != HEAD
7. SET TEMP = TEMP -> NEXT

[END OF LOOP]

1. SET TEMP -> NEXT = NEW\_NODE
2. EXIT



1. **Inserting At Specific location in the list**

We can use the following steps to insert a new node after a node in the circular linked list.

**Steps:**

1. START
2. Store the data to create linked list.
3. Store the element to be at any position of list.
4. Store the position and start the counter.
5. Check if head==null not possible else goto step 6.
6. Store data at stored position.
7. Create new link for data stored and point towards next position.
8. Change the link address of next and previous node where new node is entered.
9. STOP
10. **Deletion**

In a circular linked list, the deletion operation can be performed in three ways those are as follows.

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node
4. **Deleting from Beginning of the list**

We can use the following steps to delete a node from beginning of the circular linked list.

**Steps:**

1. IF HEAD = NULL

Write UNDERFLOW

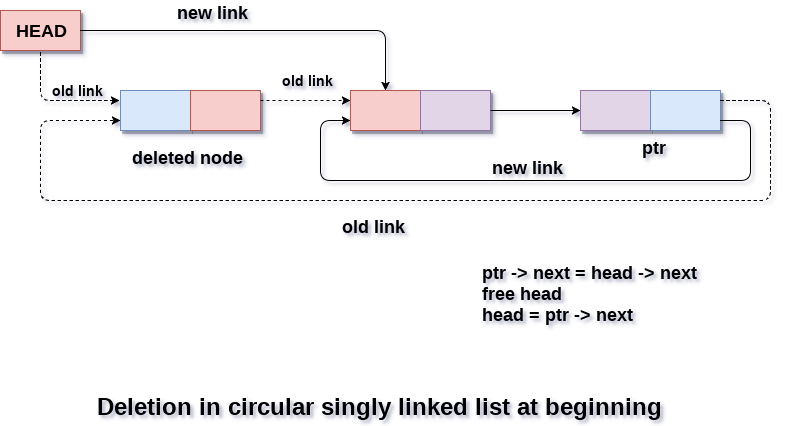
Go to Step 8

[END OF IF]

1. SET PTR = HEAD
2. Repeat Step 4 while PTR → NEXT != HEAD
3. SET PTR = PTR → next

[END OF LOOP]

1. SET PTR → NEXT = HEAD → NEXT
2. FREE HEAD
3. SET HEAD = PTR → NEXT
4. EXIT



1. **Deleting from End of the list**

We can use the following steps to delete a node from end of the circular linked list.

**Steps:**

1. IF HEAD = NULL

Write UNDERFLOW

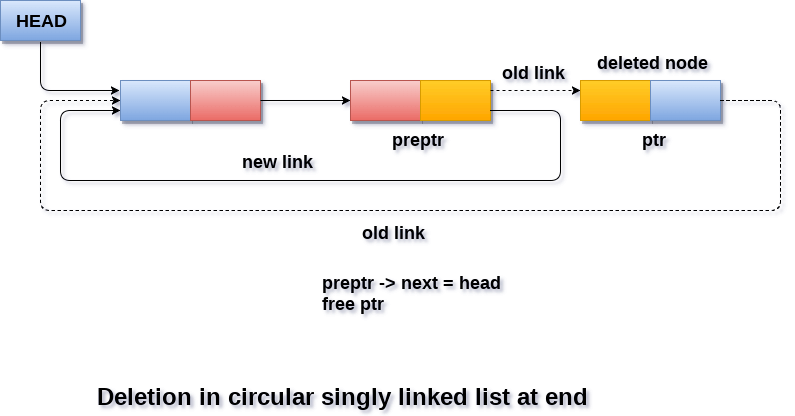
Go to Step 8

   [END OF IF]

1. SET PTR = HEAD
2. Repeat Steps 4 and 5 while PTR -> NEXT != HEAD
3. SET PREPTR = PTR
4. SET PTR = PTR -> NEXT

[END OF LOOP]

1. SET PREPTR -> NEXT = HEAD
2. FREE PTR
3. EXIT



1. **Deleting a Specific Node from the list**

We can use the following steps to delete a specific node from the circular linked list.

**Steps:**

1. START
2. Store the element to be deleted.
3. Store the position.
4. Initialize counter c=1
5. Check **while (c<pos)**
6. Count the position of data to be deleted **y = x; x = x->link; c++;**
7. Use **free()** to delete y->link = x->link; free(x);
8. STOP
9. **Displaying a circular Linked List**

Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable **temp** to head pointer and run the while loop until the next pointer of temp becomes **head**.

**Steps:**

1. SET PTR = HEAD
2. IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 8

END IF

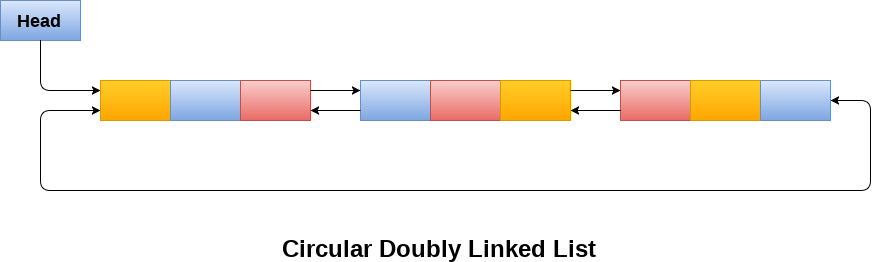
1. REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD
2. PRINT PTR → DATA
3. PTR = PTR → NEXT

[END OF LOOP]

1. PRINT PTR→ DATA
2. EXIT

**Doubly Circular LinkedList**

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



Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

## Operations

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

1. Insertion
2. Deletion
3. Display
4. **INSERTION**

In a circular linked list, the insertion operation can be performed in three ways. They are as follows.

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list
   1. **Inserting At Beginning of the list**

We can use the following steps to insert a new node at the beginning of the circular linked list.

**Steps**

1. IF PTR = NULL

Write OVERFLOW

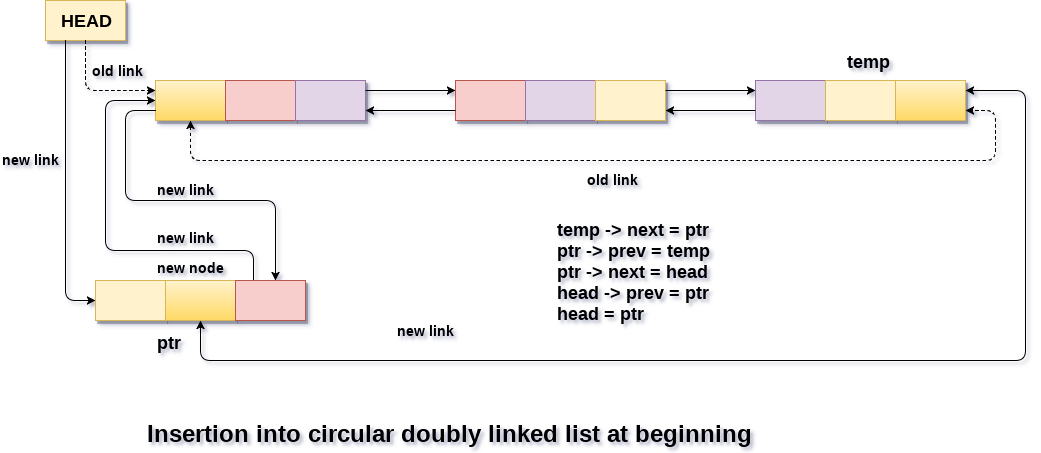
Go to Step 13

[END OF IF]

1. SET NEW\_NODE = PTR
2. SET PTR = PTR -> NEXT
3. SET NEW\_NODE -> DATA = VAL
4. SET TEMP = HEAD
5. Repeat Step 7 while TEMP -> NEXT != HEAD
6. SET TEMP = TEMP -> NEXT

[END OF LOOP]

1. SET TEMP -> NEXT = NEW\_NODE
2. SET NEW\_NODE -> PREV = TEMP
3. SET NEW\_NODE -> NEXT = HEAD
4. SET HEAD -> PREV = NEW\_NODE
5. SET HEAD = NEW\_NODE
6. EXIT



* 1. **Inserting At End of the list**

We can use the following steps to insert a new node at end of the circular linked list.

**Steps:**

1. IF PTR = NULL

Write OVERFLOW

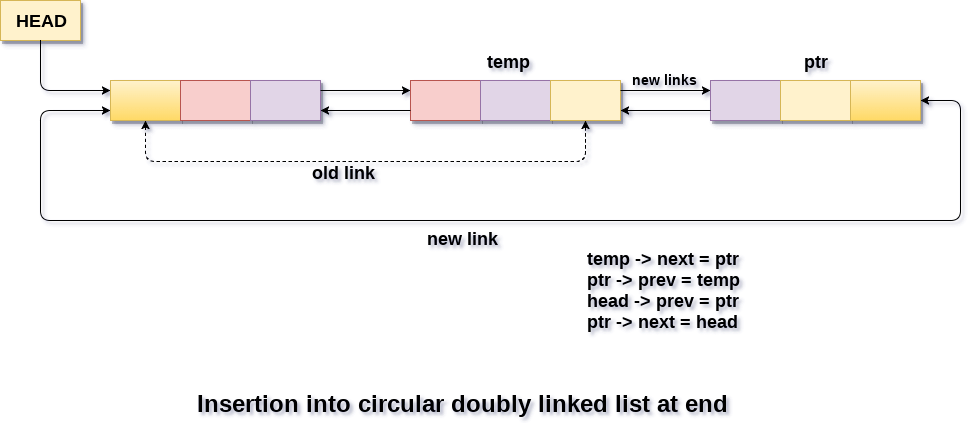
Go to Step 12

[END OF IF]

1. SET NEW\_NODE = PTR
2. SET PTR = PTR -> NEXT
3. SET NEW\_NODE -> DATA = VAL
4. SET NEW\_NODE -> NEXT = HEAD
5. SET TEMP = HEAD
6. Repeat Step 8 while TEMP -> NEXT != HEAD
7. SET TEMP = TEMP -> NEXT

[END OF LOOP]

1. SET TEMP -> NEXT = NEW\_NODE
2. SET NEW\_NODE -> PREV = TEMP
3. SET HEAD -> PREV = NEW\_NODE
4. EXIT



1. **Deletion**

In a circular linked list, the deletion operation can be performed in three ways those are as follows.

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node
4. **Deleting from Beginning of the list**

We can use the following steps to delete a node from beginning of the doubly circular linked list.

**Steps:**

1. IF HEAD = NULL

Write UNDERFLOW

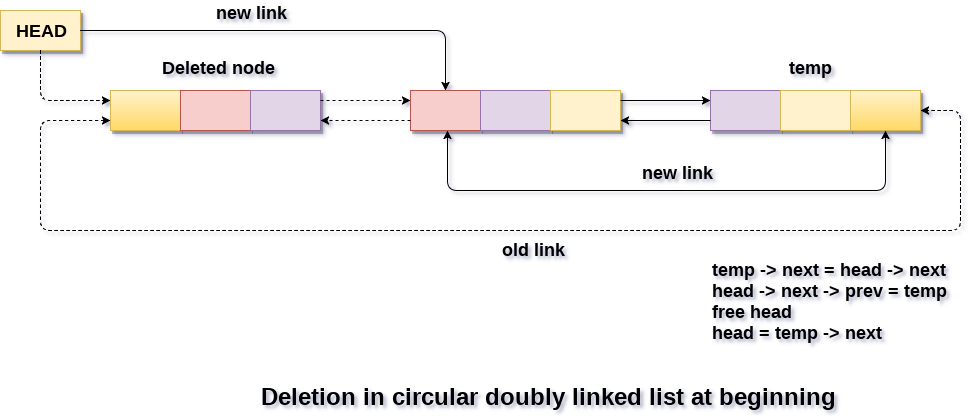
Go to Step 8

[END OF IF]

1. SET TEMP = HEAD
2. Repeat Step 4 while TEMP -> NEXT != HEAD
3. SET TEMP = TEMP -> NEXT

[END OF LOOP]

1. SET TEMP -> NEXT = HEAD -> NEXT
2. SET HEAD -> NEXT -> PREV = TEMP
3. FREE HEAD
4. SET HEAD = TEMP -> NEXT



1. **Deleting from Beginning of the list**

We can use the following steps to delete a node from beginning of the doubly circular linked list.

**Steps:**

1. IF HEAD = NULL

Write UNDERFLOW

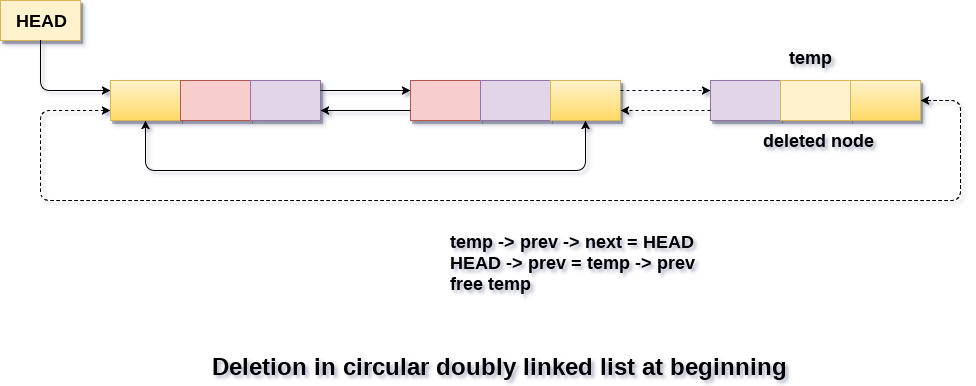
Go to Step 8

[END OF IF]

1. SET TEMP = HEAD
2. Repeat Step 4 while TEMP -> NEXT != HEAD
3. SET TEMP = TEMP -> NEXT

[END OF LOOP]

1. SET TEMP -> PREV -> NEXT = HEAD
2. SET HEAD -> PREV = TEMP -> PREV
3. FREE TEMP
4. EXIT



**Dynamic memory allocation**

As pointer variables are used to hold the address of a variable, which is declared with the pointer variable during the variable declaration statement. So assigning of address of variables to a pointer variable is one of the legal way of the initialization of pointer. This process is always dynamic.

In the dynamic memory allocation of the memory, the declaration of variable is no longer required. The job of memory allocation and deallocation can be done by using the following functions. They are

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

* The malloc() function allocates single block of requested memory.
* It doesn't initialize memory at execution time, so it has garbage value initially.
* It returns NULL if memory is not sufficient.

**Syn:** void \*malloc(unsigned size)

**Eg:**

#include<stdio.h>

#include<stdlib.h>

main()

{

int n,i,\*ptr,sum=0;

clrscr();

printf("Enter number of elements: ");

scanf("%d",&n);

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

if(ptr==NULL)

{ printf("Sorry! unable to allocate memory");

exit(0);

}

printf("Enter elements of array: ");

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

{

scanf("%d",ptr+i);

sum+=\*(ptr+i);

}

printf("Sum=%d",sum);

free(ptr);

}

**Output:**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

* + - 1. **Calloc():**
* The calloc() function allocates multiple block of requested memory.
* It initially initialize all bytes to zero.
* It returns NULL if memory is not sufficient.

**Syn:** void \*calloc(xyz, unsigned size)

**Eg:**

#include<stdio.h>

#include<stdlib.h>

main()

{

int n,i,\*ptr,sum=0;

clrscr();

printf("Enter number of elements: ");

scanf("%d",&n);

ptr=(int\*)calloc(n,sizeof(int)); //memory allocated using calloc

if(ptr==NULL)

{

printf("Sorry! unable to allocate memory");

exit(0);

}

printf("Enter elements of array: ");

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

{

scanf("%d",ptr+i);

sum+=\*(ptr+i);

}

printf("Sum=%d",sum);

free(ptr);

getch();

}

**Output**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

* + - 1. **Realloc():**

It adjusts the previously allocated block to the size bytes. It will allocate the increasing or decreasing the block to the new size. Here block refers to the original block already allocated by malloc() or calloc() function. It returns the address of the new reallocated block. Note that if it will unable to reallocate the block, then it will display and return the NULL value.

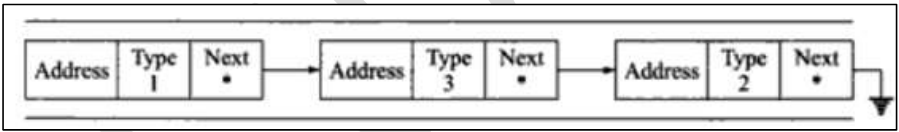
**Syn:** void \*recalloc(xyz, unsigned size)

* + - 1. **free():**
* The **free() function** in C library allows you to release or deallocate the memory blocks which are previously allocated by calloc(), malloc() or realloc() functions.
* It frees up the memory blocks and returns the memory to heap.
* It helps freeing the memory in your program which will be available for later use.
* In C, the memory for variables is automatically deallocated at compile time. For dynamic memory allocation in C, you have to deallocate the memory explicitly. If not done, you may encounter out of memory error.

**Syntax:** void free(void \*ptr)

**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.



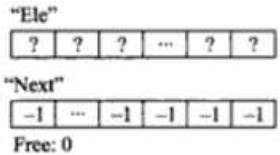
**Linked List In Arrays**

Linked list is 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 guaranteed to be stored in the ith index of an array “Ele”.

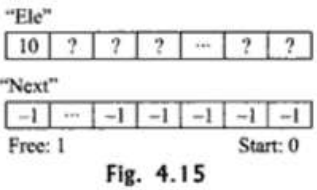
The node in a linked list contains two parts - “data” and “next”. These two parts of node are split and stored in two arrays “Ele” and “Next”. If Ele[i] represents the data part of the node then Next[i] denotes the next part of that node.

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.

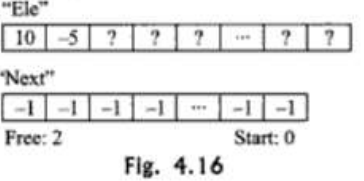
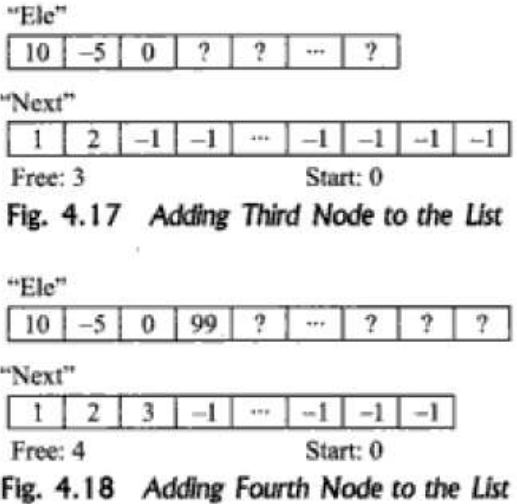
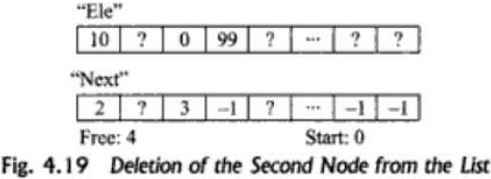


Figure 4.16 illustrates adding second element in the lis. Note that Ele[1] is set to -5, Next[0] is set to 1 and Next[1] is set to -1. Another addition will need to traverse the list from “start”, start = 0. As Next[0] = 1 the next node can be found in index 1 of the array “Ele” and “Next”. The Next[1]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

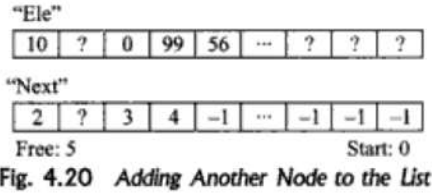
As before, Next[1] is set to the current value of “free”. Ele\*free+ is set to -1, and “free” is incremented. Figure 4.17 and 4.18 illustrate the addition of third and fourth nodes to the list.



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.

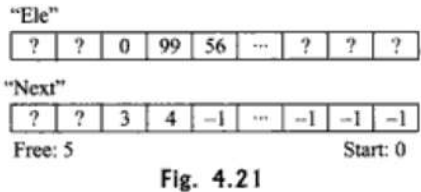


Therefore, it can be seen that no shifting of elements is done. Only the “next” array is updated. Say, if we want to again add an element with value 56 to the list after deletion (fig. 4.20)



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.



**LINKED LISTS VERSUS ARRAYS**

We can store similar data in memory with the use of either an array or a linked list. Arrays are very simple data structures that are easy to understand but they have the following disadvantages:

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

Linked list can be used to overcome all these disadvantages.

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

**UNIT – III**

**Stack**

A stack is a non-primitive linear data structure. A stack is an ordered collection of data in which data can be inserted from the top end and the removing of elements can be done from the top end itself, so this type of structure is called as **“Last in First Out”**

A Stack is defined as a data structure which operates on a Last in First Out basis. It is generally used to provide temporary storage space for values. Stack is most commonly used as a place to store local variables, parameters and return addresses when a function is called. The use of stacks is critical where recursive functions that is, functions that call themselves, are defined.

Stack uses a single pointer to keep track of the information in the stack. The basic operations associated with a stack are

* 1. Push
  2. Pop

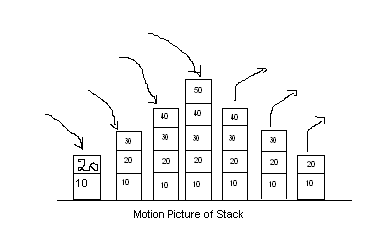
The general terminology associated with stacks is as follows: A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be pushed on the stack. When an object is removed from the stack, it is said to be popped off the stack. Two additional terms almost always used with stacks are overflow, which occurs when we try to push more information on a stack that it can hold, and underflow, which occurs when we try to pop an item off a stack which empty.

**Push:**

While inserting values in a stack first verify that is there any space available in a stack or not. If available then we can insert from the top end otherwise it raises a message as stack is not empty.

**Pop:**

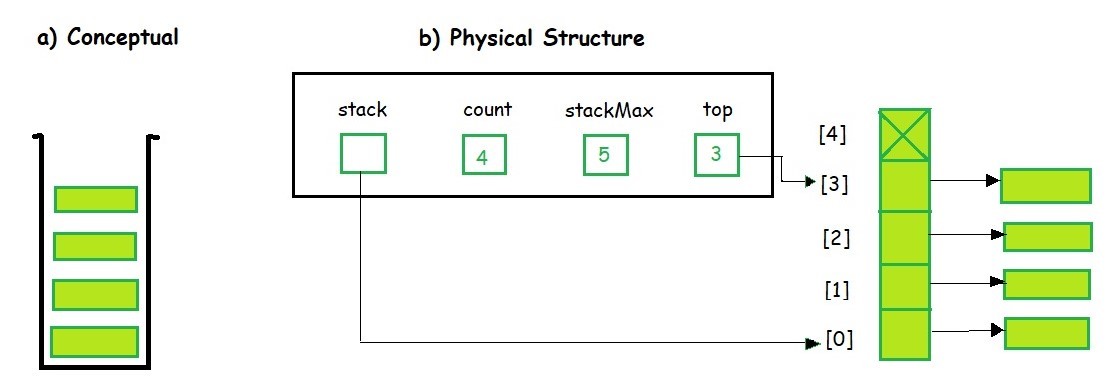
While removing values from the stack first verify that is there any values available in a stack or not. If available then removing of values can be done from the end itself otherwise it raises a message as stack is empty.



Stack operations can be implemented either by using arrays or by dynamic memory allocations.

**Stack as an Abstract Data Type**

* In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
* The program allocates memory for the *data* and *address* is passed to the stack ADT.



* The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
* The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.

**//Stack ADT Type Definitions**

typedef struct node

{

void \*DataPtr;

struct node \*link;

} StackNode;

typedef struct

{

int count;

StackNode \*top;

} STACK;

A Stack contains elements of the same type arranged in sequential order. All operations take place at a single end that is top of the stack and following operations can be performed:

* push() – Insert an element at one end of the stack called top.
* pop() – Remove and return the element at the top of the stack, if it is not empty.
* peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
* size() – Return the number of elements in the stack.
* isEmpty() – Return true if the stack is empty, otherwise return false.
* isFull() – Return true if the stack is full, otherwise return false.

**Representation of Stack**

The two standard methods of representing a stack are in terms of an **array** and **Linked List**.

1. **Array implementation of a stack:**

The stack contains 5 elements. The stack pointer pointing to the top of the stack. The size of array is fixed at the time of its declaration itself. But, as the definition shows, stacks are dynamic data structures. They can grow and shrink during their operation. However, stacks can be implemented using arrays by specifying a maximum size.

We required two things two represent stacks. First one is an array to hold the items and second is an integer variable to hold the index of top element.

a[4]=50

a[3]=40

a[2]=30

a[1]=20

a[0]=10

**// Program to demonstrate stack operations**

#include<stdio.h>

#include<conio.h>

#define m=5

int stk[m],top=-1,f=0;

main()

{

int v,ch;

clrscr();

do

{

printf("\n 1. Push \n 2. Pop \n 3.List");

printf("Select the choice");

scanf(%d”,&ch);

if(ch==1)

{

printf("Enter the value to push");

scanf(“%d”,&v);

push(stk,v);

if(f==1)

{

if(top==(m-1))

printf("Stack is full");

}

else

printf("Stack is overflow");

}

else

if(ch==2)

{

v=pop(stk);

if(f==1)

printf("Deleted value from stack ="+v);

else

printf("Stack is empty");

}

else

if(ch==3)

{

for(int i=top;i>=0;i--)

printf("Stk["+i+"]="+stk[i]);

}

}while(ch<=3);

getch();

}

push(int stk[],int v)

{

if(top==(m-1))

f=0;

else

{

top++;

stk[top]=v;

f=1;

}

}

int pop(int stk[])

{

int x=0;

if(top==-1)

f=0;

else

{

x=stk[top];

top--;

f=1;

}

return x;

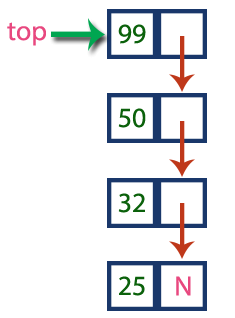
}

1. **Using Linked List**

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL

**Example**



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

**// Program to Stack operations using Linked**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int a;

struct node \*next;

};

struct node \*top;

void push()

{

struct node \*temp;

int i;

printf("Enter value to push");

scanf("%d",&i);

temp=(struct node\*) malloc(sizeof(struct node));

temp->a=i;

temp->next=NULL;

if(top!=NULL)

temp->next=top;

top=temp;

}

void show()

{

struct node \*x=top;

printf("top->");

while(x!=NULL)

{

printf("%d->",x->a);

x=x->next;

}

}

void pop()

{

struct node \*temp;

temp=top->next;

free(top);

top=temp;

}

void main()

{

int ch;

clrscr();

do

{ printf("\n 1. Push \n 2. pop \n 3. list");

printf("\n Select the choice");

scanf("%d",&ch);

if(ch==1)

push();

else

if(ch==2)

pop();

else

if(ch==3)

show();

}while(ch<=3);

getch();

}

**STACK APPLICATIONS**

There are several applications where the data structure STACK can be put to use. The applications includes

1. Stack is used by compilers to check for balancing of parantheses, brackets and braces.
2. Stack is used to convert an infix expression into postfix or prefix form.
3. Stack is used to evaluate a postfix expression
4. In recursion, all intermediate arguments and return values are stored on the processors stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off
6. Depth first search uses a stack data structure to find an element from a graph.

**Expression**

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

**“An expression is a collection of operators and operands that represents a specific value”.**

In above definition, operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

**Expression Types**

Based on the operator position, expressions are divided into THREE types. They are as follows.

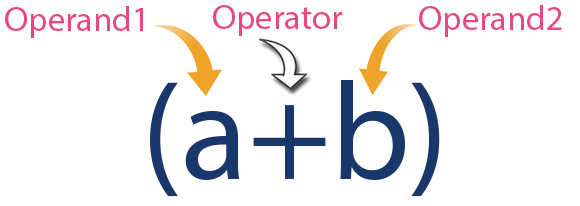
1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

**Infix Expression:**

In infix expression, operator is used in between operands.

**Syn: Operand1 Operator Operand2**

**Example**

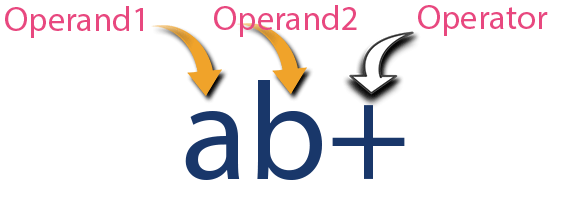


**Postfix Expression**

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".

**Syn: Operand1 Operand2 Operator**

**EXAMPLE**

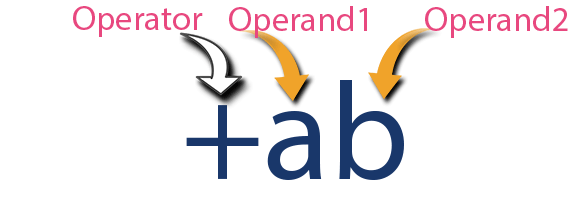


**Prefix Expression**

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

**Syn: Operator Operand1 Operand2**

**EXAMPLE**



Any expression can be represented using the above three different types of expressions. And we can convert an expression from one form to another form like Infix to Postfix, Infix to Prefix, Prefix to Postfix and vice versa.

**Expression Conversion**

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

1. **Convert Infix expression into Postfix** 
   * 1. Find all the operators in the given Infix Expression.
     2. Find the order of operators evaluated according to their Operator precedence.
     3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

**Example**

**Consider the following Infix Expression to be converted into Postfix Expression.**

#### D = A + B \* C

**Step 1:** The Operators in the given Infix Expression : = , + , \*

**Step 2:** The Order of Operators according to their preference : \* , + , =

**Step 3:** Now, convert the first operator \* ----- D = A + B C \*

**Step 4:** Convert the next operator + ----- D = A BC\* +

**Step 5:** Convert the next operator = ----- D ABC\*+ =

**Finally, given Infix Expression is converted into Postfix Expression as follows**

#### D A B C \* + =

**Infix to Postfix Conversion using Stack Data Structure**

To convert Infix Expression into Postfix Expression using a stack data structure, we can use the following steps.

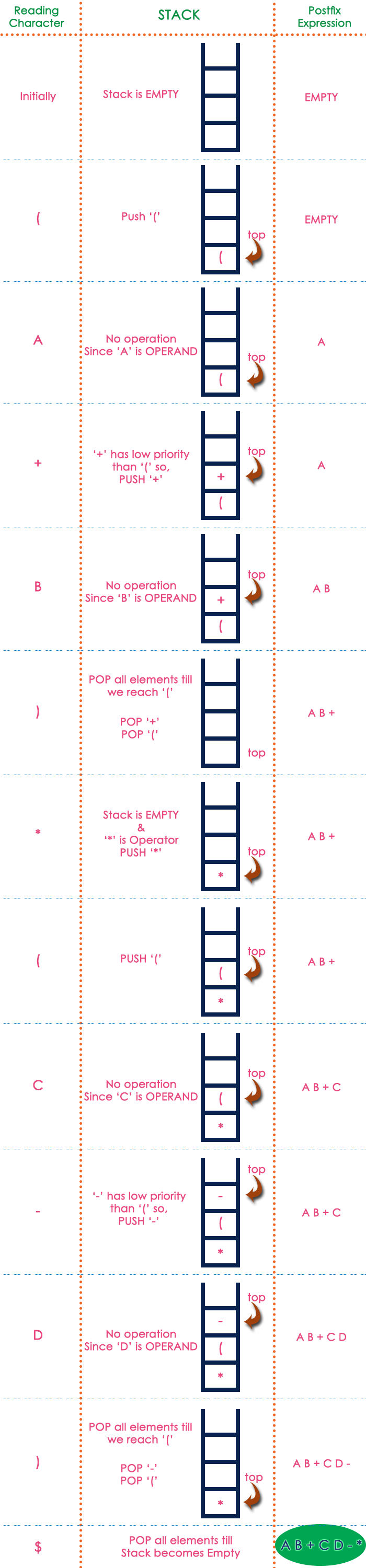
1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is operator (+ , - , \* , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

**Example**

**Consider the following Infix Expression.**

#### ( A + B ) \* ( C - D )

The given infix expression can be converted into postfix expression using Stack data Structure as follows.



**The final Postfix Expression is as follows**

#### A B + C D - \*

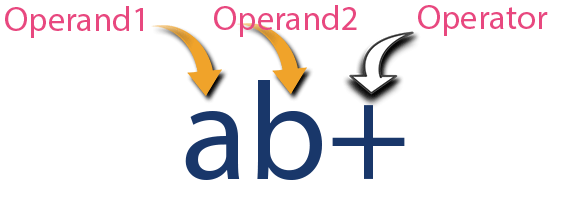
**Postfix Expression Evaluation**

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

**Postfix Expression has following general structure.**

#### Operand1 Operand2 Operator

**Example**



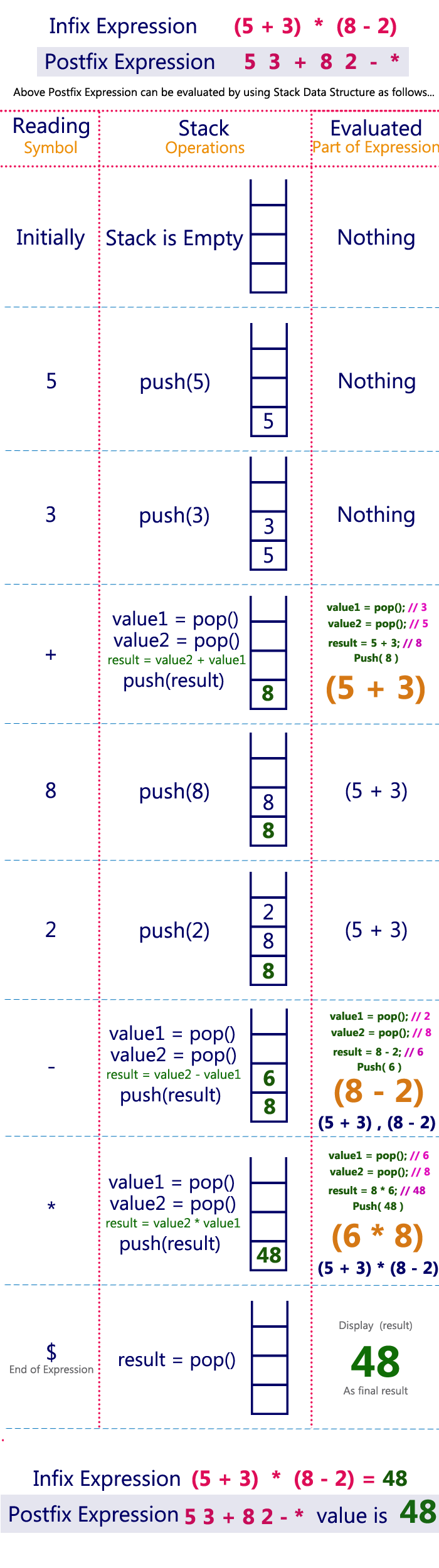
**Postfix Expression Evaluation using Stack Data Structure**

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps.

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

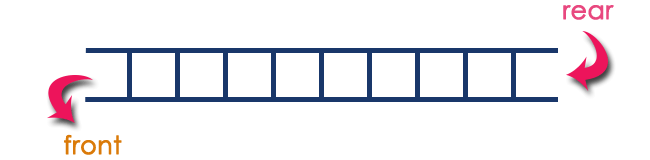
**Example**

Consider the following Expression.



**Queues**

Queue is a non-primitive linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as **'rear'** and the deletion operation is performed at a position which is known as **'front'**. In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle.



Typical uses of queues in computers are **process management**, **buffer between the fast computer and a slow printer**. In a queue the following operations can be performed. They are -

* 1. Push()
  2. Pop()
  3. Display()

**Push():**

While inserting values in a queue first verify that is there any space available in a queue or not. If available then we can insert from the top end otherwise it raises a message as queue is not empty.

**Algorithm:**

1. Check whether queue is FULL. (rear == SIZE-1)
2. If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
3. If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

**Pop():**

While removing values from the queue first verify that is there any values available in a queue or not. If available then removing of values can be done from the front end itself otherwise it raises a message as queue is empty.

**Algorithm:**

1. Check whether queue is EMPTY. (front == rear)
2. If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
3. If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

**Display():** We can use the following steps to display the elements of a queue.

**Algorithm:**

1. Check whether queue is EMPTY. (front == rear)
2. If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
3. If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.
4. Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value is equal to rear (i <= rear)

**Real life examples of Queues:**

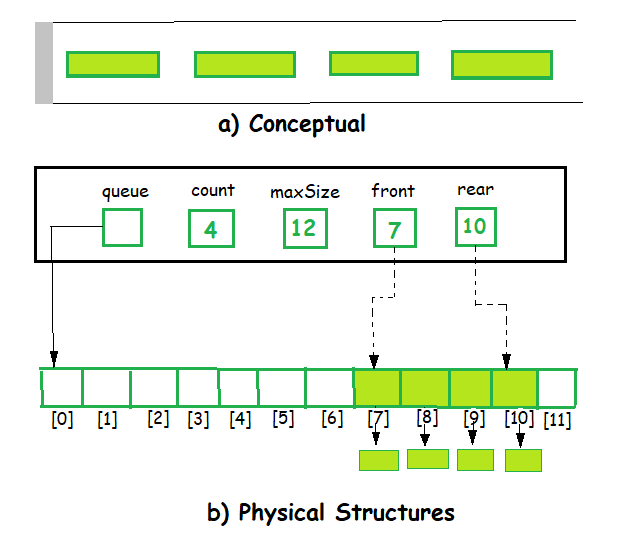
* 1. Queue lines waiting for buses
  2. Queue lines wait to be served at post office
  3. Queue lines wait to be served at banks
  4. Queue at cinema theatre
  5. Queue at ration shops

**Applications of Queues:**

Numerous applications of queue structures are known in computer science. One major application of queue is in simulation. Another important application of queue is observed in implementation of various aspects of operating system. Multiprogramming environment uses several queues to control various programs.

**Queue ADT**

* The queue abstract data type (ADT) follows the basic design of the stack abstract data type.



* Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program’s responsibility is to allocate memory for storing the data.

**//Queue ADT Type Definitions**

typedef struct node

{

void \*DataPtr;

struct node \*next;

} QueueNode;

typedef struct

{ QueueNode \*front;

QueueNode \*rear;

int count;

} QUEUE;

A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at the end and deletion is done at the front. Following operations can be performed:

* enqueue() – Insert an element at the end of the queue.
* dequeue() – Remove and return the first element of the queue, if the queue is not empty.
* peek() – Return the element of the queue without removing it, if the queue is not empty.
* size() – Return the number of elements in the queue.
* isEmpty() – Return true if the queue is empty, otherwise return false.
* isFull() – Return true if the queue is full, otherwise return false.

**Different ways of representing Queues**

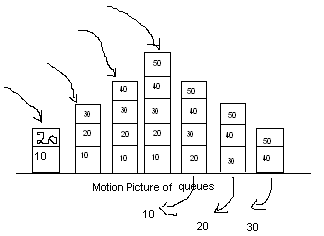
Queue data structure can be implemented in two ways. They are as follows.

1. **Using Array**
2. **Using Linked List**

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

1. **Queue Using Array**

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.



**//Program to perform queue operations**

#include<stdio.h>

#include<conio.h>

#define m 5

int que[m],top=1,f=0,c=-1,pos=0;

void push(int que[],int v)

{

if(top==(m-1))

f=0;

else

{

top++;

que[top]=v;

f=1;

c++;

}

}

int pop(int que[])

{ int x=0;

if(top==-1)

f=0;

else

{ x=que[c-top];

top--;

f=1;

pos++;

if(pos==(m-1))

{ pos=0;

c=-1;

}

}

return x;

}

main()

{ int v,ch;

clrscr();

do

{ printf("\n 1. Push \n 2. Pop \n 3.List");

printf("Select the choice");

scanf(“%d”,&ch);

if(ch==1)

{ printf("Enter the value to push");

scanf(“%d”,&v);

push(que,v);

if(f==1)

{

if(top==(m-1))

printf("queueis full");

}

else

printf("queue is overflow");

}

else

if(ch==2)

{ v=pop(que);

if(f==1)

print("Deleted value from queue="+v);

else

System.out.println("Queue is empty");

}

else

if(ch==3)

{ for(int i=pos;i<=c;i++)

System.out.println("Que["+i+"]="+que[i]);

}

}while(ch<=3);

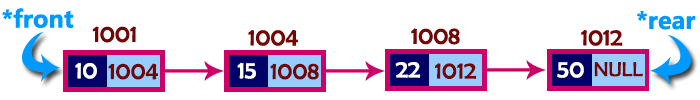
}

}

1. **Queue using Linked List**

The major problem with the queue implemented using array is, it will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data. The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.



In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

**// Program to perform Queue Operations using Linked list**

#include<stdio.h>

#include<conio.h>

#include <stdlib.h>

struct que

{ int a;

struct que \*next;

}\*s,\*t1,\*t2,\*t3;

void main()

{ int ch;

clrscr();

do

{ printf("\n 1. push \n 2. pop \n 3. list");

printf("\n Select the choice");

scanf("%d",&ch);

if(ch==1)

{

if(t1==NULL)

{ t1=(struct que \*) malloc(sizeof(struct que));

printf("\n Enter the value to push");

scanf("%d",&t1->a);

t1->next=NULL;

s=t1;

}

else

{ t2=(struct que \*) malloc(sizeof(struct que));

printf("\n Enter the value to push");

scanf("%d",&t2->a);

t1->next=t2;

t2->next=NULL;

t1=t2;

} }

else

if(ch==2)

{

if(s==NULL)

prinf("\n Que is empty");

else

{ t3=s->next;

printf("\n Deleted value of que = %d",s->a);

free(s);

s=t3;

}

}

else

if(ch==3)

{ t3=s;

while(t3!=NULL)

{ printf(" %d ->",t3->a);

t3=t3->next;

}

}

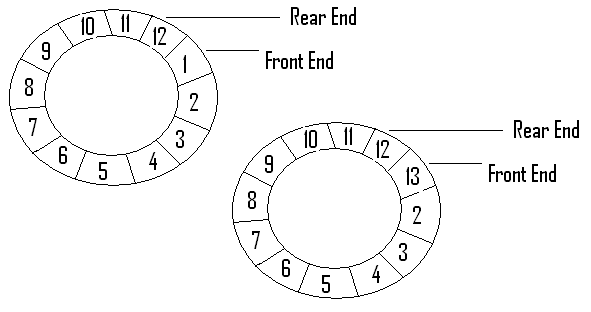
}while(ch<=3);

getch();

}

**CIRCULAR QUEUES**

It is a non-primitive linear data structure. A circular queue, is a queue in which all locations are treated as circular such that the first location CQ[1] follows the last location CQ[max] and is a conceptual view of a circular queue. The normal queue, that suffers from one limitation. In that implementation when the Rear pointer reaches at the end, insertion will be denied even if room is available at the front. To overcome this limitation we can implement the queue as a circular queue.



**ALGORITHM OF CIRCULAR QUEUE**

**Cqueinsertion(q, max, front ,rear, item)**

* + 1. If rear=max then

Rear = 1

Else

Rear = rear + 1

1. If front = rear then

Print “Que is Overflow

Q[rear]=item

1. If front = 0 then

Front=1

**Cquedeletion(q,max,front,rear)**

* 1. if front = 0 then

print “Que is empty”

k=q[front]

* 1. if front = rear then

begin

front =0

rear=0

end

else

* 1. if front = max then

front =1

else

font = front +1

return k

**/\* Program to perform circular queue operations using arrays \*/**

#include<stdio.h>

#include<conio.h>

#define m 5

int front=-1,rear=-1,cque[m],ch,v,i;

main()

{

clrscr();

do

{ printf("\n 1. Insert \n 2. Delete \n 3. Display");

printf("\nSelect the choice");

scanf("%d",&ch);

if(ch==1)

insert();

else

if(ch==2)

delete();

else

if(ch==3)

display();

}while(ch<=3);

getch();

}

**insert()**

{

if((rear+1)%m==front)

printf("que is overflow");

else

{ rear=(rear+1)%m;

printf("Enter Data to insert into cque");

scanf("%d",&v);

cque[rear]=v;

if(front==-1)

front=0;

} }

**delete()**

{

if(front==-1)

printf("Que is underflow");

else

{

printf("Deleted element= %d",cque[front]);

if(front%m!=rear)

front=(front+1)%m;

else

rear=front-1;

}

}

**display()**

{

if(front==-1)

printf("Que is empty");

else

if(front<rear)

{

printf("Front->");

for(i=front;i<=rear;i++)

printf("%d->",cque[i]);

printf("Rear");

}

else

{

printf("Front->");

for(i=front;i<m;i++)

printf("%d->",cque[i]);

for(i=front;i<=rear;i++)

printf("%d->",cque[i]);

}

}

**Priority Queues**

Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue – highest priority items are removed first.

A priority queue is a queue that contains items that have some predefined ordering. Unlike the usual removal of the first item, when an item is removed from a priority queue, the item with the highest priority is removed. For instance, assume that an integer priority queue has small integers with the highest priority. Assume that the items 4,3,1,5,2 are added sequentially to the queue. Unlike the usual queue, this priority queue will order the items in ascending order.

**Front:** 1 2 3 4 5 : **back**

The first item to be returned from this priority queue is the 1. If the number 0 is added to the queue after the removal of 1, then the 1 will be in the front of the queue, even though it is the last item to be added to the queue:

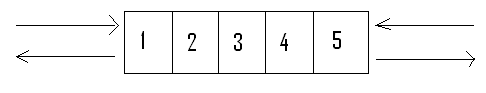
**Front:** 0 2 3 4 5 : **back**

If a new item is added to the queue which has the same value as an existing member of the queue, then this new item has a lower precedence then the item already existing. For instance, if a new number 3 is added to the above priority queue, then the queue would look like this.

**Front:** 0 2 3 **3** 4 5 : **back**

**DEQUEUES**

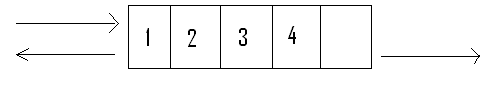
A dequeue, or simple dequeue, is a queue in which the elements can be inserted and deleted at both ends of a queue. This is a very flexible structure when compared to stacks and queues.



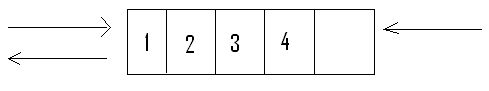
Dequeues can also be maintained in circular fashion. We can have several other types of queues by limiting or relaxing some constraints. Two out of them are frequently used.

* 1. Input restricted Deque
  2. Output restricted Deque

The input restricted dequeue is a dequeue in which deletions are allowed at both ends but insertions are allowed only at one end.



The output restricted dequeue is a dequeue in which insertions are allowed at both ends but deletions are allowed only at one end.



**10 Mark Questions**

* + 1. **Write about stacks? Write about the different ways of representing stack?**
    2. **Write the algorithm of stack operations?**
    3. **Write a program to perform stack operations?**
    4. **What is Expression? Write different types of expressions?**
    5. **Write about Queue?** Write about different ways of representing Queues?
    6. **Write the algorithm of queue operations?**
    7. **Write a program to perform queue operations?**

**5. Mark Questions**

* + - 1. **Write about Stack ADT?**
      2. **Write the applications of Stack?**
      3. **Write about Queue ADT?**
      4. Write about Circular Queues and its algorithm?
      5. **Write about Priority Queues?**
      6. **Write about Dequeues?**

**Unit - IV**

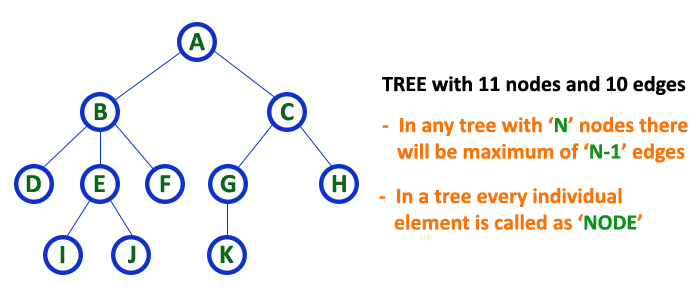
**Tree**

In linear data structure, data is organized in sequential order and in non-linear data structure data is organized in random order. Tree is a very popular data structure used in wide range of applications. **“Tree is a non-linear data structure which organizes data (Node) in hierarchical structure and this is a recursive definition”**.

In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

**Example**

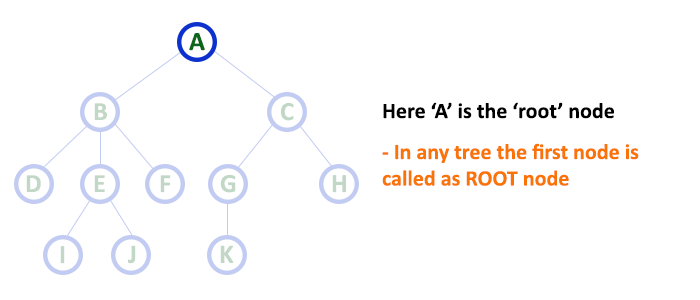


**Tree Terminology**

In a tree data structure, we use the following terminology

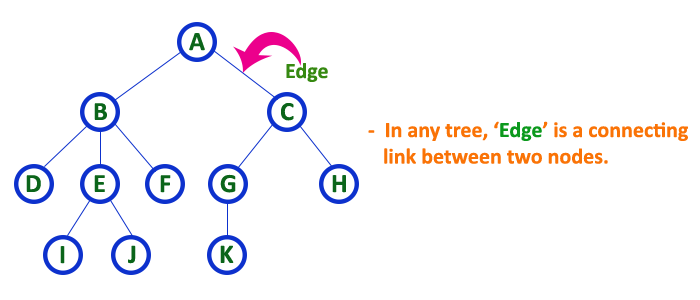
1. **Root**

In a tree data structure, the first node **“A”** is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



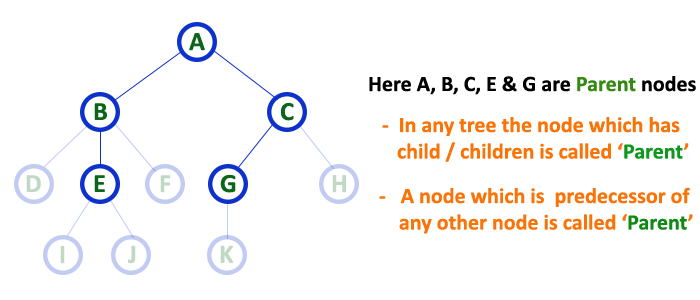
1. **Edge**

In a tree data structure, **the connecting link between any two nodes** is called as **EDGE.** In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



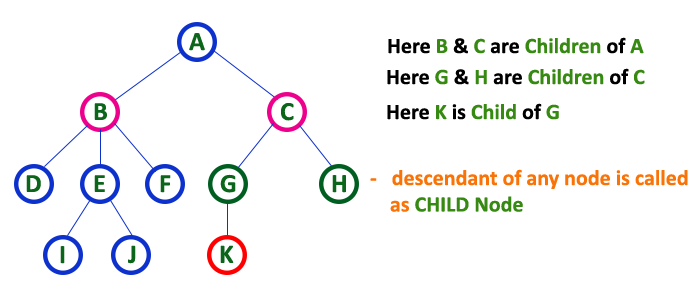
1. **Parent**

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children".



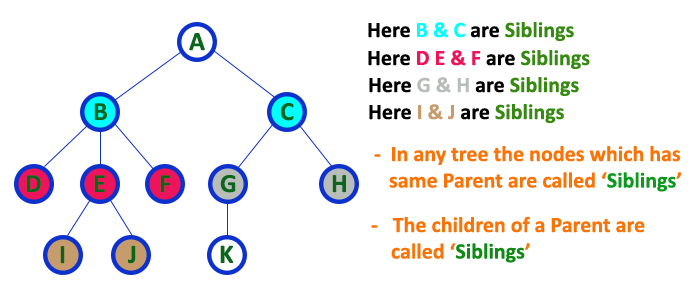
1. **Child**

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



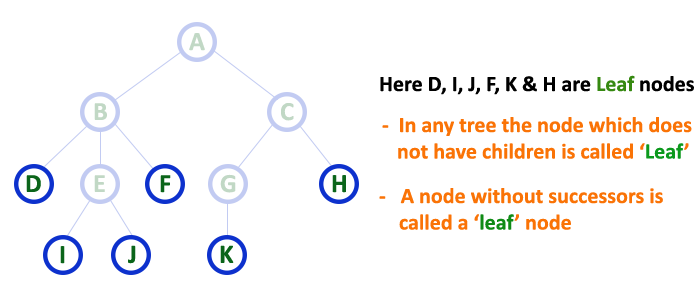
1. **Siblings**

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes



1. **Leaf**

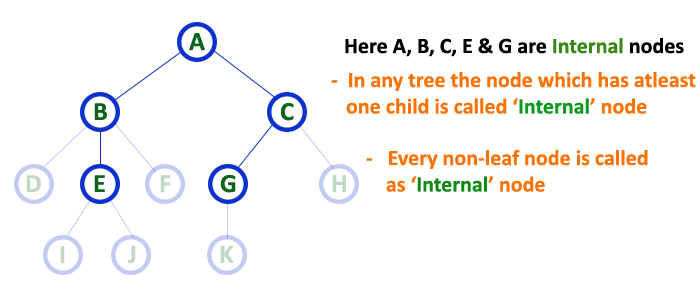
In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.



1. **Internal Nodes**

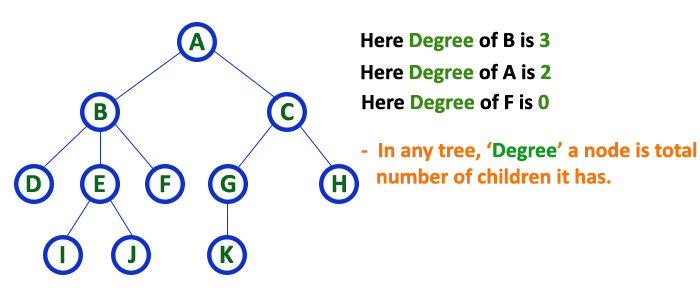
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node.** In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



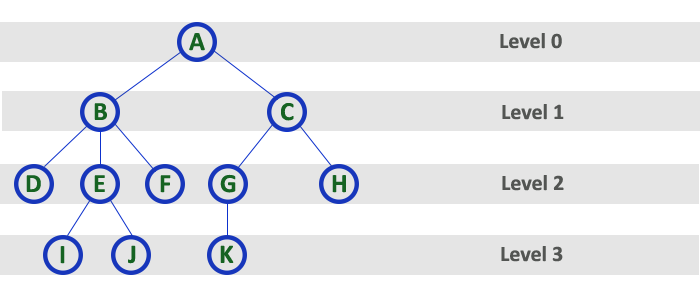
1. **Degree**

In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



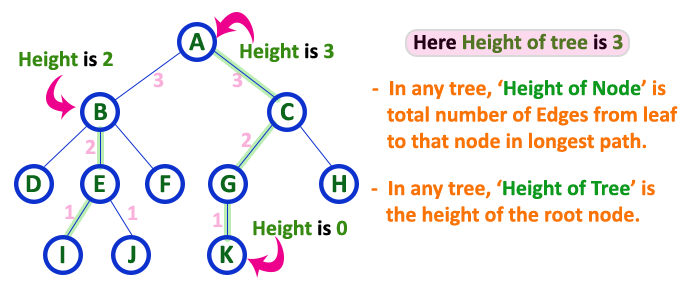
1. **Level**

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step)



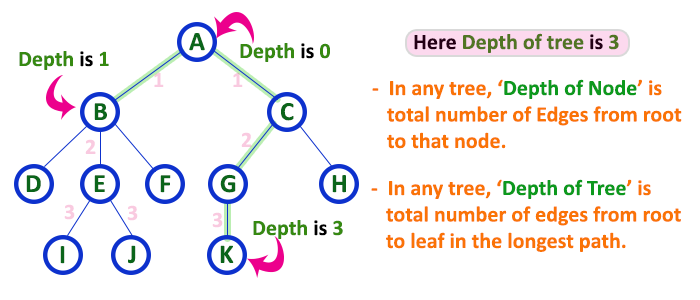
1. **Height**

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



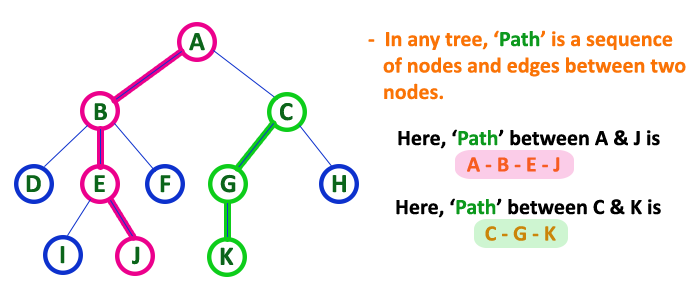
1. **Depth**

In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.



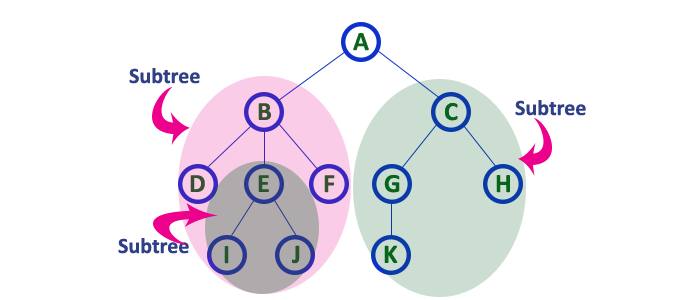
1. **Path**

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



1. **Sub Tree**

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

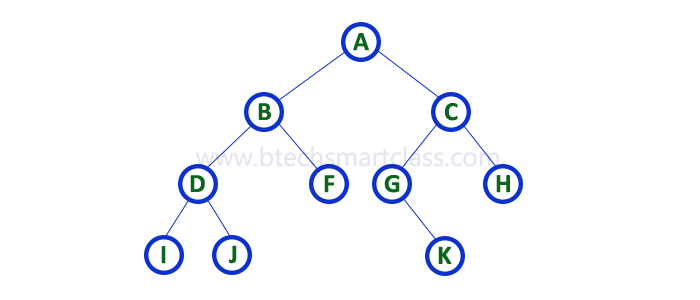


**Binary Tree**

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree. In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

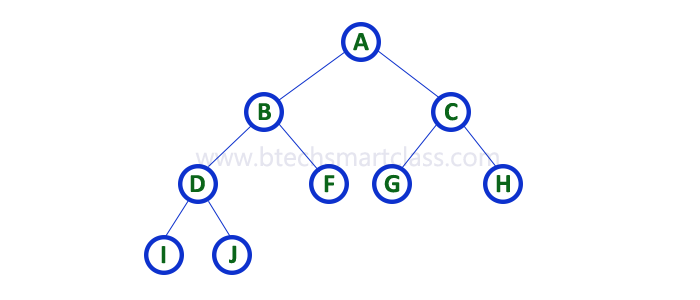
**Example**



There are different types of binary trees and they are.

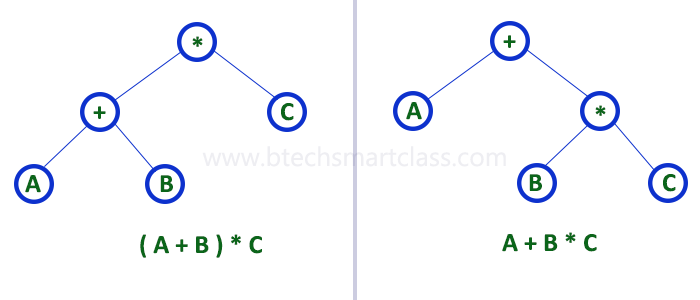
1. **Strictly Binary Tree**

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as **“A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree”**. **Strictly binary tree is also called as *Full Binary Tree* or *Proper Binary Tree* or *2-Tree***



Strictly binary tree data structure is used to represent mathematical expressions.

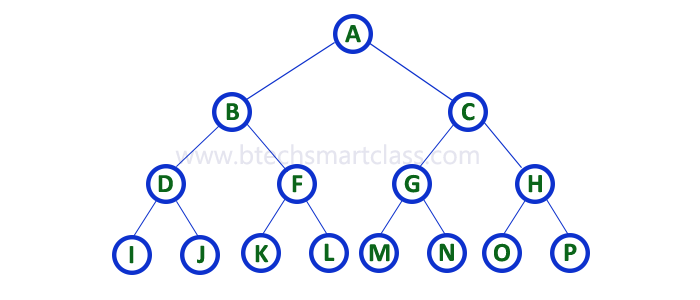
**Example**



1. **Complete Binary Tree**

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2level number of nodes. For example at level 2 there must be 22 = 4 nodes and at level 3 there must be 23 = 8 nodes.

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.Complete binary tree is also called as Perfect Binary Tree



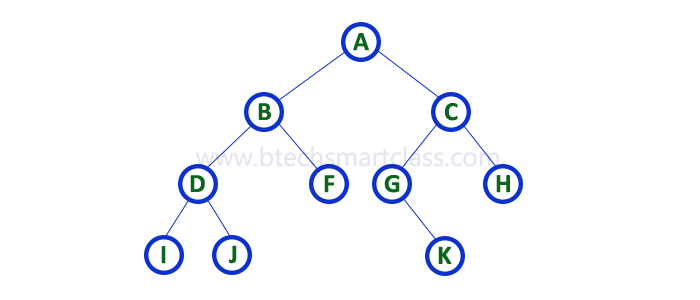
**Binary Tree Representations**

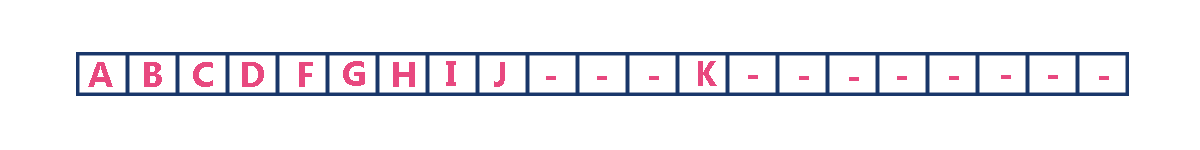
A binary tree data structure is represented using two methods. They are

1. **Array Representation**
2. **Linked List Representation**
3. **Array Representation**

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

**Consider the following binary tree.**





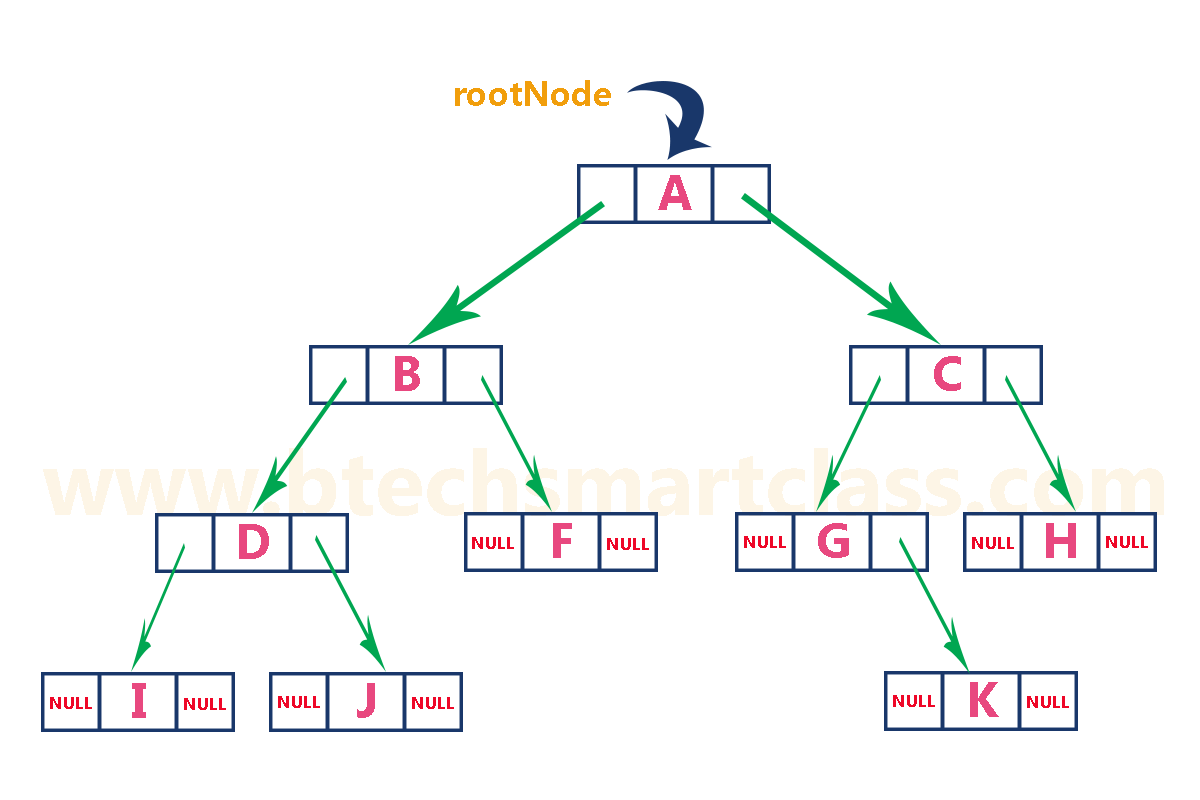
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n+1 - 1.

1. **Linked List Representation**

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.In this linked list representation, a node has the following structure.



The above example of binary tree represented using Linked list representation is shown as follows.

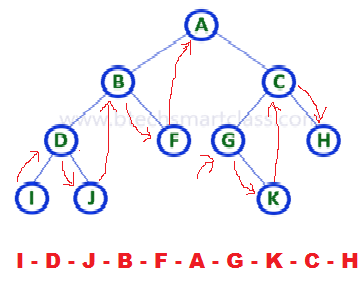


**Binary Tree Traversals**

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method. Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal. There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal
4. **In - Order Traversal ( leftChild - root - rightChild )**

* In In-Order traversal, the root node is visited between left child and right child.
* In this traversal, the left child node is visited first.
* The root node is visited
* Later we go for visiting right child node.
* This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.



**In the above example of binary tree**, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. So we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

**Algorithm**

**Procedure inorder(T)**

1. If T = NULL then

print “Tree is empty”

2. if leftPtr (T) # NULL then

call inorder(LeftPtr)

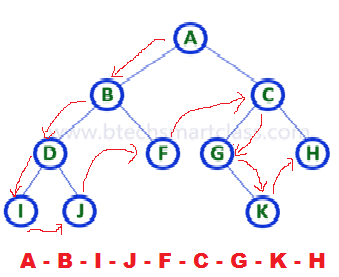
3. print info(T)

4. if RightPtr(T) # NULL then

call inorder(RightPtr)

1. **Pre - Order Traversal ( root - leftChild - rightChild )**

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.



In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

**Algorithm**

**Procedure Preorder(T)**

1. If T = NULL then

begin

print “Tree is empty”

return

end

2. print info(T)

3. if leftPtr (T) # NULL then

call preorder(LeftPtr)

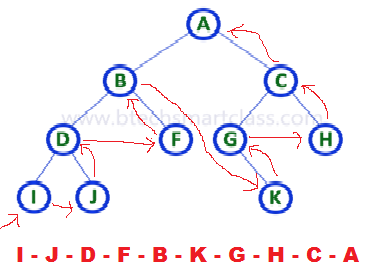
4. if RightPtr(T) # NULL then

call preorder(RightPtr)

5. Return

1. **Post - Order Traversal ( leftChild - rightChild - root )**

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.



Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

#### I - J - D - F - B - K - G - H - C - A

**Algorithm**

**Procedure postorder(T)**

1. If T = NULL then

print “Tree is empty”

2. if leftPtr (T) # NULL then

call postorder(LeftPtr)

3. if RightPtr(T) # NULL then

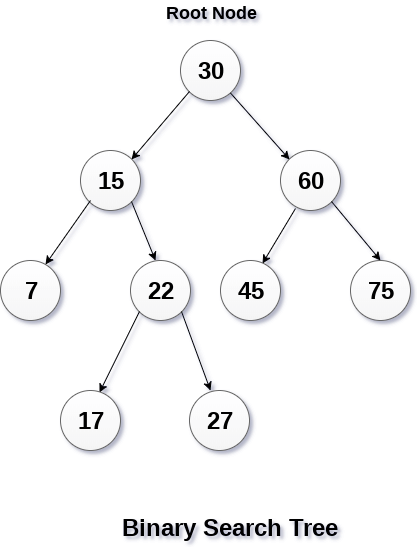
call postorder(RightPtr)

4. print info(T)

5. Return

**BST (BINARY SEARCH TREE/BINARY SORT TREE)**

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

## Advantages of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o(log2n) time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

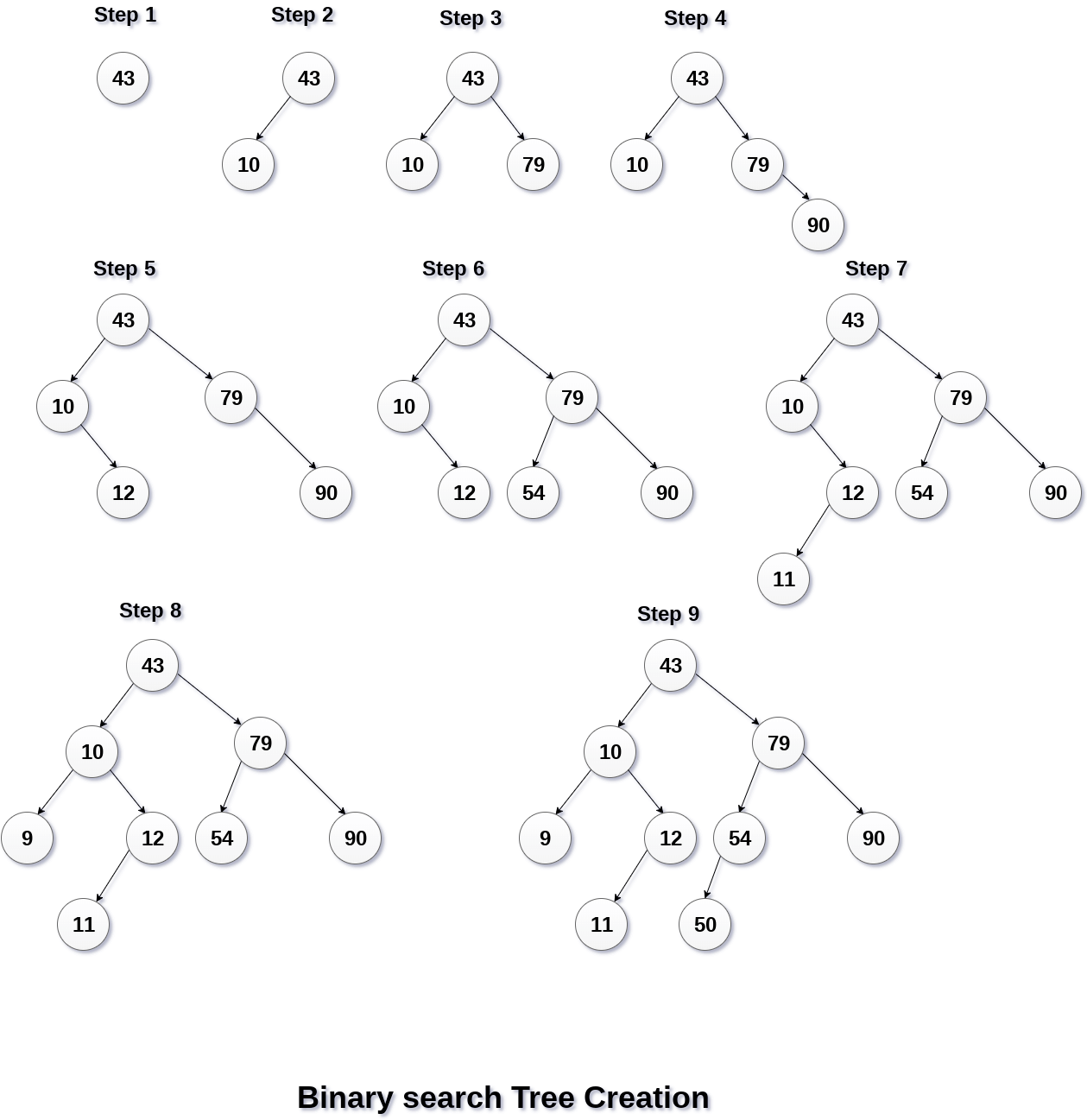
### Eg: Create the binary search tree using the following data elements.

**43, 10, 79, 90, 12, 54, 11, 9, 50**

**Steps:**

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.

  
**Operations on Binary Search Tree**

The basic operations which can be performed on binary search tree are:

* + - 1. **Insertion** of a node in Binary Search Tree
      2. **Deletion** of a node from BST
      3. **Searching** for a particular node in BST.

# Insertion of a node in Binary Search Tree

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

## Algorithm

## Insert (TREE, ITEM)

* **Step 1:** IF TREE = NULL

    Allocate memory for TREE

SET TREE -> DATA = ITEM

SET TREE -> LEFT = TREE -> RIGHT = NULL

ELSE  
   IF ITEM < TREE -> DATA

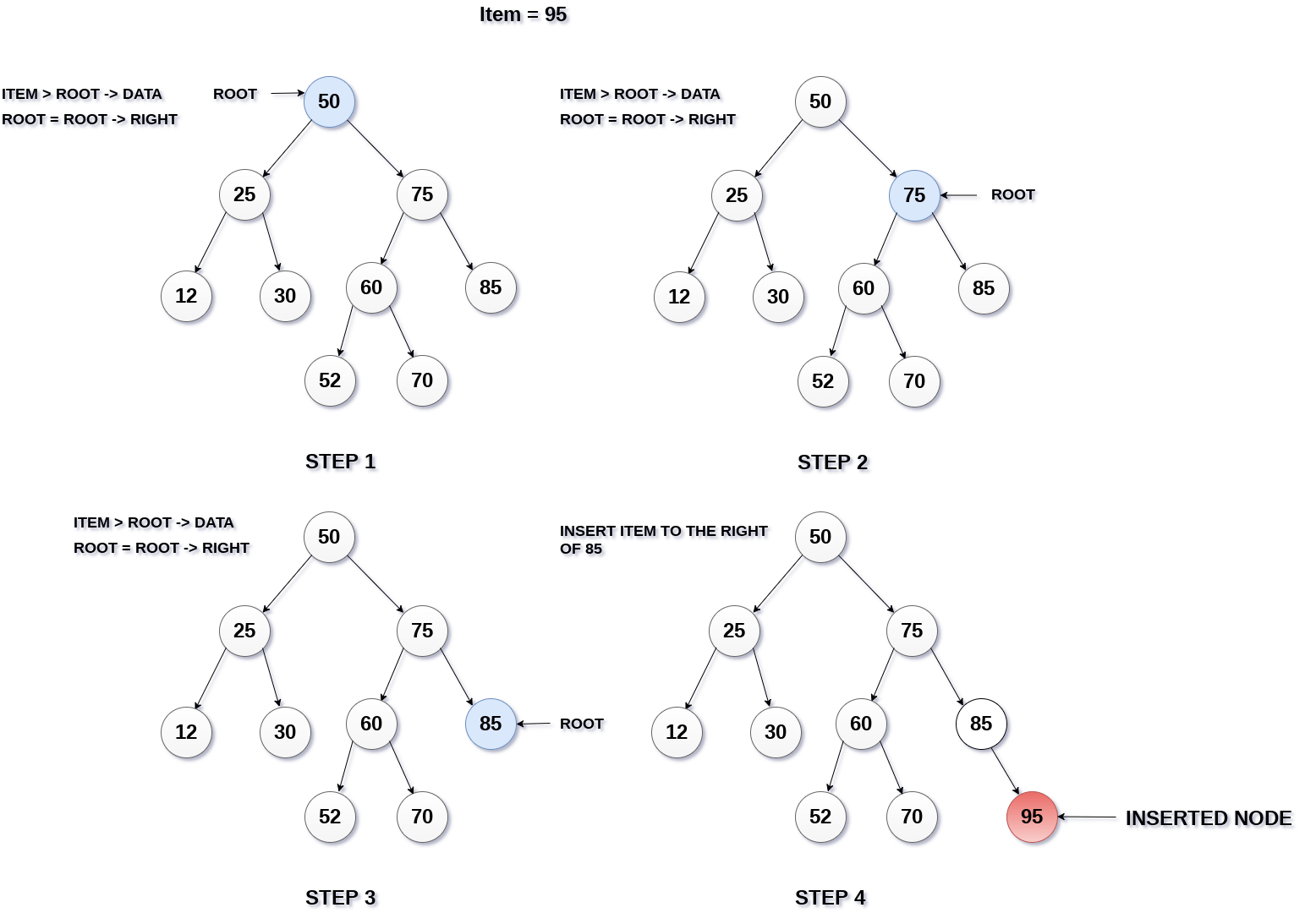
    Insert(TREE -> LEFT, ITEM)

ELSE  
   Insert(TREE -> RIGHT, ITEM)

[END IF]

  [END IF]

* **Step 2:** END

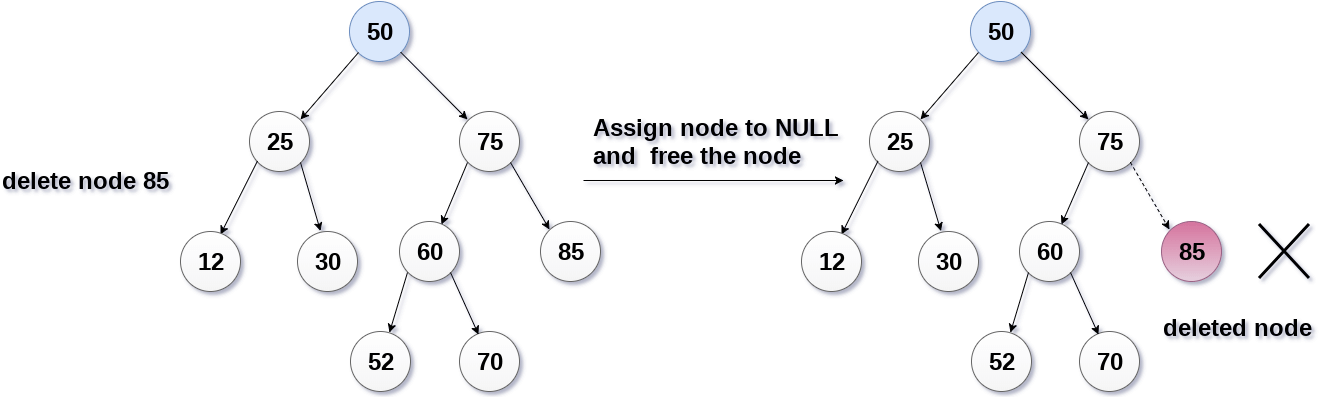


# Deletion of a node in BST

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three cases of deleting a node from binary search tree.

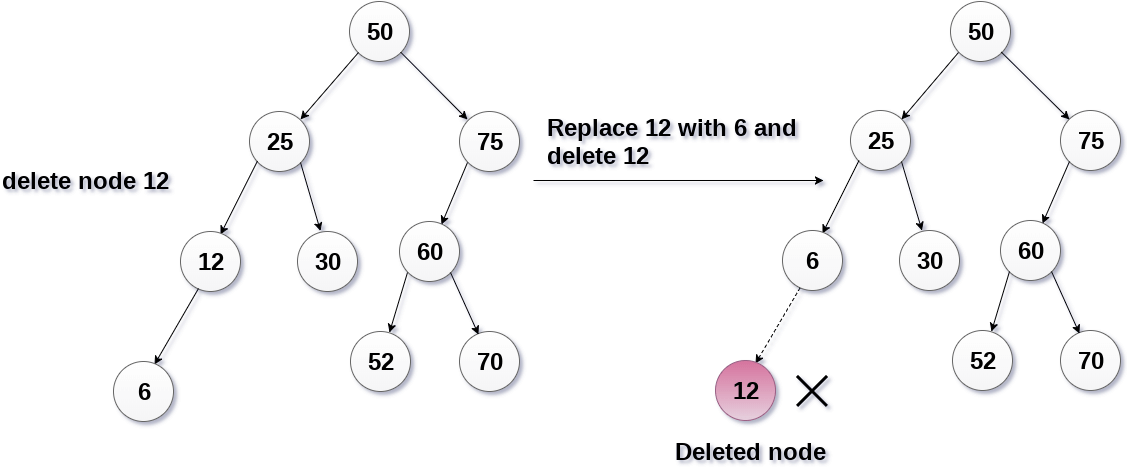
## Case – 1: The node to be deleted is a leaf node

* It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.
* In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



## Case -2: The node to be deleted has only one child.

* In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.
* In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.

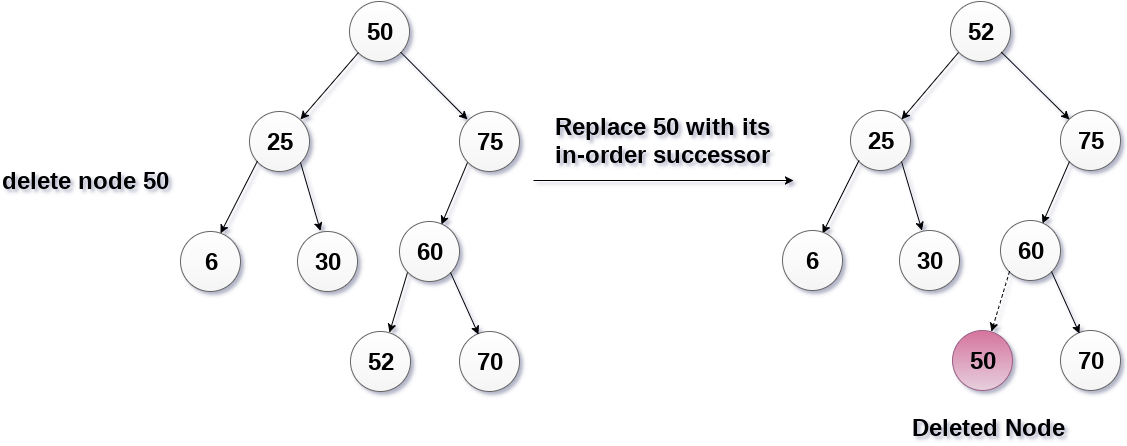


## Case – 3: The node to be deleted has two children.

* It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.
* In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



**Algorithm**

**Delete (TREE, ITEM)**

* **Step 1:** IF TREE = NULL

   Write "item not found in the tree"

ELSE IF ITEM < TREE -> DATA

Delete(TREE->LEFT, ITEM)

ELSE IF ITEM > TREE -> DATA

Delete(TREE -> RIGHT, ITEM)

ELSE IF TREE -> LEFT AND TREE -> RIGHT

SET TEMP = findLargestNode(TREE -> LEFT)

  SET TREE -> DATA = TEMP -> DATA

   Delete(TREE -> LEFT, TEMP -> DATA)

ELSE

SET TEMP = TREE

IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL

SET TREE = NULL

ELSE IF TREE -> LEFT != NULL

SET TREE = TREE -> LEFT

ELSE  
    SET TREE = TREE -> RIGHT

[END IF]

FREE TEMP

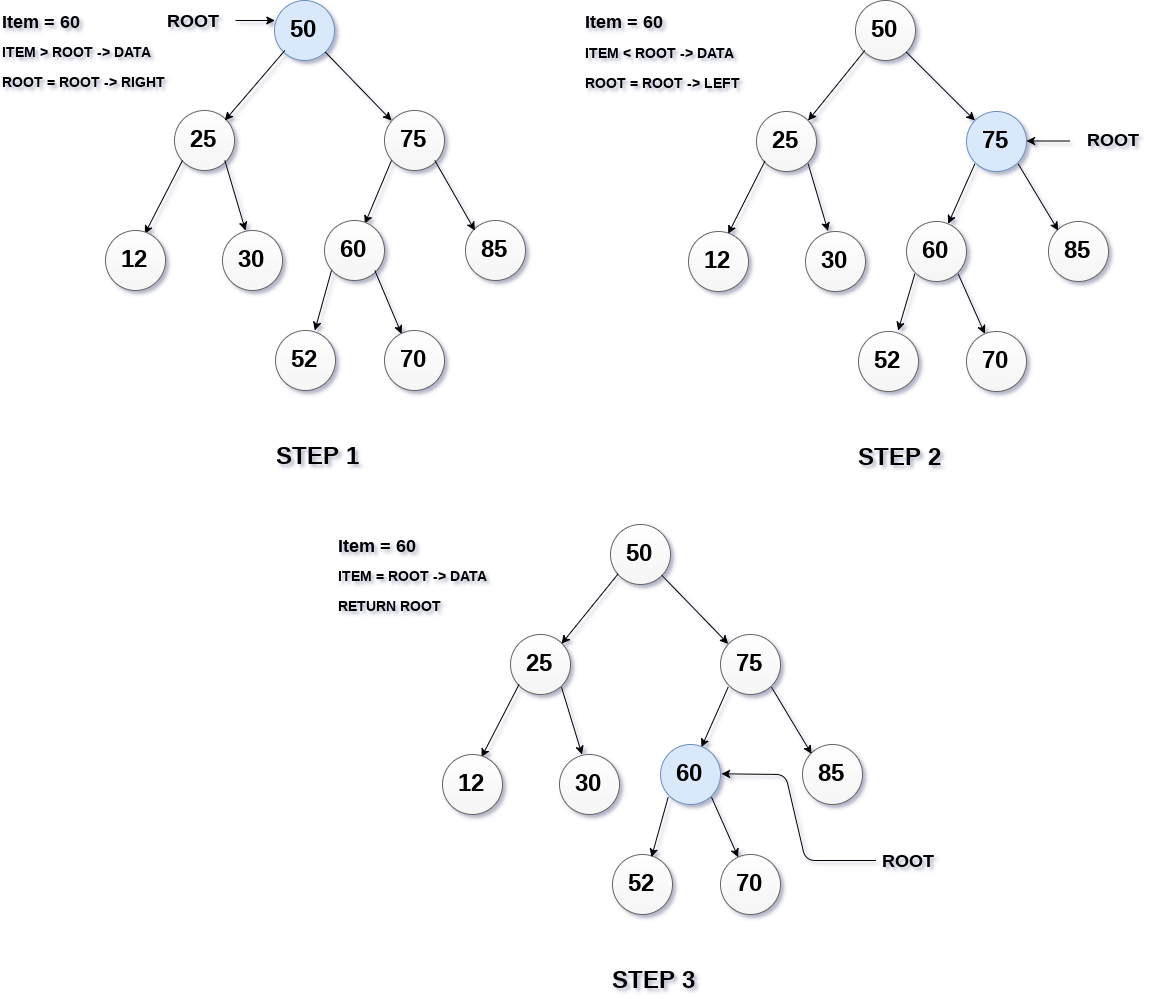
[END OF IF]

* **Step 2:** END

1. **Searching**

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.



**Algorithm:**

**Search (ROOT, ITEM)**

* **Step 1:** IF ROOT -> DATA = ITEM OR ROOT = NULL

Return ROOT

ELSE  
   IF ROOT < ROOT -> DATA

Return search(ROOT -> LEFT, ITEM)

ELSE  
   Return search(ROOT -> RIGHT,ITEM)

[END IF]

[END IF]

* **Step 2:** END

## Program to implement BST operations

#include <stdio.h>

#include <stdlib.h>

struct Node

{

**int** data;

Node \*left, \*right;

};

Node\* create(**int** item)

{

Node\* node = **new** Node;

node->data = item;

node->left = node->right = NULL;

**return** node;

}

**void** inorder(Node \*root)

{

**if** (root == NULL)

**return**;

inorder(root->left);

cout<< root->data << "   ";

inorder(root->right);

}

Node\* findMinimum(Node\* cur)

{

**while**(cur->left != NULL) {

cur = cur->left;

}

**return** cur;

}

Node\* insertion(Node\* root, **int** item)

{

**if** (root == NULL)

**return** create(item);

**if** (item < root->data)

root->left = insertion(root->left, item);

**else**

root->right = insertion(root->right, item);

**return** root;

}

**void** search(Node\* &cur, **int** item, Node\* &parent)

{

**while** (cur != NULL && cur->data != item)

{

parent = cur;

**if** (item < cur->data)

cur = cur->left;

**else**

cur = cur->right;

}  }

**void** deletion(Node\*& root, **int** item)

{  Node\* parent = NULL;

Node\* cur = root;

search(cur, item, parent);

**if** (cur == NULL)

**return**;

**if** (cur->left == NULL && cur->right == NULL)

{

**if** (cur != root)  {

**if** (parent->left == cur)

parent->left = NULL;

**else**

parent->right = NULL;

}

**else**

root = NULL;

free(cur);

}

**else** **if** (cur->left && cur->right)  {

Node\* succ  = findMinimum(cur- >right);

**int** val = succ->data;

deletion(root, succ->data);

cur->data = val;

}

**else**

{

Node\* child = (cur->left)? Cur- >left: cur->right;

**if** (cur != root)

{

**if** (cur == parent->left)

parent->left = child;

**else**

parent->right = child;

}

**else**

root = child;

free(cur);  }  }

main()

{  Node\* root = NULL;

**int** keys[8];

**for**(**int** i=0;i<8;i++)

{

cout << "Enter value to be inserted";

cin>>keys[i];

root = insertion(root, keys[i]);

}

inorder(root);

cout<<"\n";

deletion(root, 10);

inorder(root);

**return** 0;  }

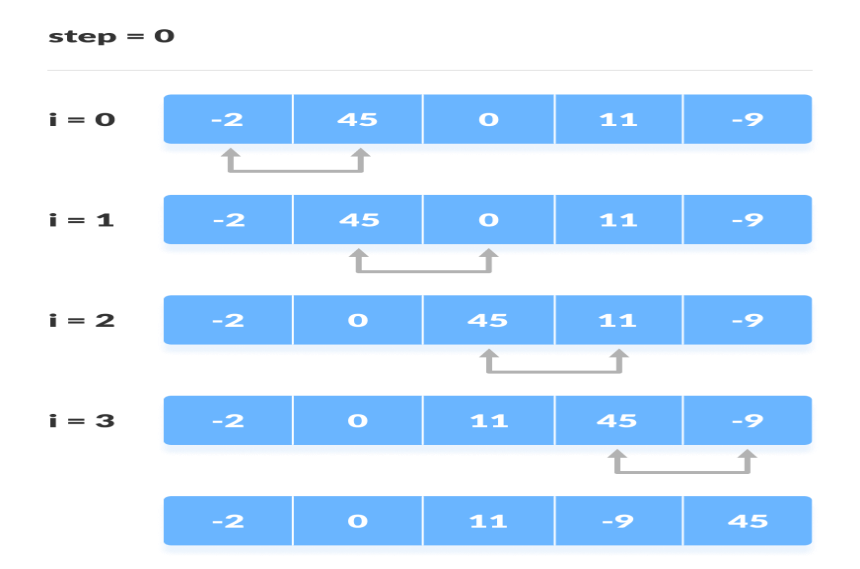
**UNIT – V**

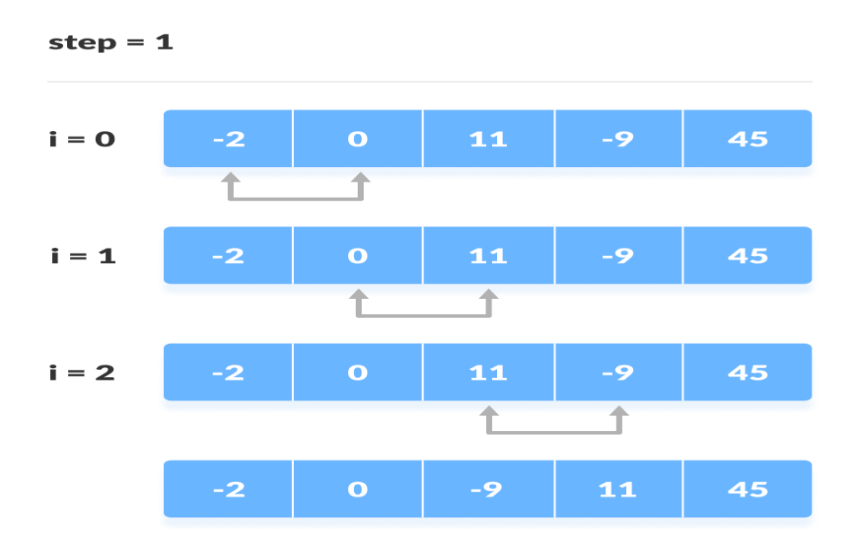
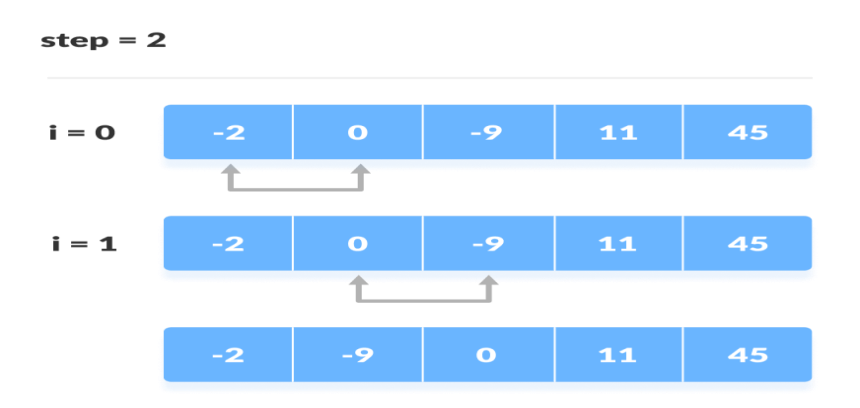
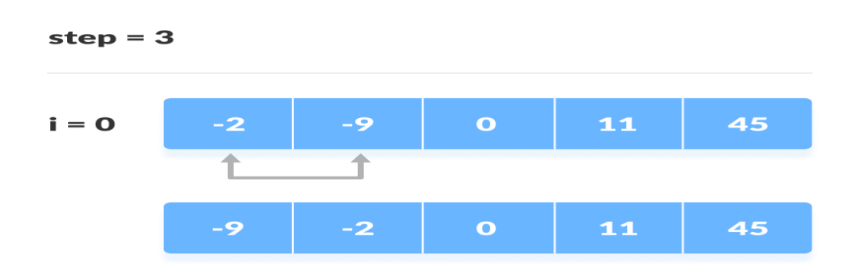
**Bubble Sort**

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.

## How Bubble Sort Works?

1. Starting from the first index, compare the first and the second elements.If the first element is greater than the second element, they are swapped.

Now, compare the second and the third elements. Swap them if they are not in order. The above process goes on until the last element.  


The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.  
  
In each iteration, the comparison takes place up to the last unsorted element.  
The array is sorted when all the unsorted elements are placed at their correct positions.  
  
  


**// Bubble sort Technique.**

#include<stdio.h>

#include<conio.h>

main()

{

int a[100],n,i,j,t;

clrscr();

printf("Enter n value");

scanf("%d",&n);

printf("Enter %d values in array",n);

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

{

scanf("%d",&a[i]);

}

for(i=0;i<n-1;i++)

{

for(j=0;j<n-i-1;j++)

{

if(a[j]>a[j+1])

{

t=a[j];

a[j]=a[j+1];

a[j+1]=t;

}

}

}

printf("After sorting");

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

printf("\n %d",a[i]);

getch();

}

**Insertion Sort**

* This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name **insertion sort**
* The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of items

## How Insertion Sort Works?

* We take an unsorted array for our example.

Unsorted Array

* Insertion sort compares the first two elements.

Insertion Sort

* It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

* Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

* And finds that 33 is not in the correct position.

Insertion Sort

* It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

* By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

* These values are not in a sorted order.

Insertion Sort

* So we swap them.

Insertion Sort

* However, swapping makes 27 and 10 unsorted.

Insertion Sort

* Hence, we swap them too.

Insertion Sort

* Again we find 14 and 10 in an unsorted order.

Insertion Sort

* We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

* This process goes on until all the unsorted values are covered in a sorted sub-list.

**// program to perform insertion sort**

#include<stdio.h>

#include<conio.h>

main()

{

int i=1,n,j,t,a[100];

clrscr();

printf("Enter value to insert in array");

scanf("%d",&n);

while(n!=-99)

{

t=i-1;

while(a[t]>n && t>=0)

{

a[t+1]=a[t];

t--;

}

a[t+1]=n;

i++;

scanf("%d",&n);

}

printf("After Sorting");

for(j=1;j<i;j++)

printf("\n %d",a[j]);

getch();

}

**Merge Sort**

**Ans: In this sorting technique,** the entire list is divided into **two sub lists** of sizes as nearly equal as possible and then the two sub lists are sorted separately. After separate sorting, the two sub lists were merged together into a single list. Thus Division of entire list into sub lists, sorting the sub lists separately, finally merging the sub lists into single list are the steps involved in the technique of merge sorting.

**Eg:** Let the following list of seven numbers be sorted

**26 33 35 29 19 12 22**

**Step-1:**

Divide the list into two. As these are seven elements in the list, we divide as 4 elements to one list and 3 elements to other list.

26 33 35 29 **Sub list 1**

19 12 22 **Sub list 2**

**Step-2**

Consider the sub list 1 and again divide the sub list 1 as:

26 33 Sub list 1(1)

35 29 Sub list 2(2)

**Step-3:**

For each of the above sub lists again the division is done and now the sub lists contain only one element as under

**26** Sub list 1(1(1))

**33** Sub list 1(1(2))

**35** Sub list 2(1(1))

**29** Sub list 2(1(2))

**Step-4:**

Now the single element sub lists are merged together to get sorted list. Sub lists 26 and 33 are merged to give sorted list 26,33 and the sub lists 35 and 29 are merged to give sorted list 29 35

**Step-5:**

Now the sorted sub lists of length 2 are merged together to give a four element sorted sub list as under.

22 29 33 35 🡪 (1)

**Step-6:** Steps 2 to 5 are repeated for the sub list 2 to get sorted sub list as:

12 19 22 🡪 (2)

**Step-7:**

Finally, the sorted sub lists (1) and(2) of lengths 4 and 3 respectively merged to give sorted lists as:

12 19 22 26 29 33 35

**// Program to perform merge sort**

#include<stdio.h>

#include<conio.h>

main()

{

int n, i,a[100];

clrscr();

printf("Enter n value") ;

scanf("%d",&n);

printf("\n Enter %d integer elements into array",n);

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

{

scanf("%d",&a[i]);

}

sort(a, 0, n);

printf("\n Elements after sorting ");

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

printf("%d ",a[i]);

getch();

}

sort(int a[100], int low, int high)

{

int mid,N,temp[100],i,j,k;

N = high - low;

if (N <= 1)

return;

mid = low + N/2;

sort(a, low, mid);

sort(a, mid, high);

i = low;

j = mid;

for (k = 0; k < N; k++)

{

if (i == mid)

temp[k] = a[j++];

else if (j == high)

temp[k] = a[i++];

else if (a[j]<a[i])

temp[k] = a[j++];

else

temp[k] = a[i++];

}

for (int k = 0; k < N; k++)

a[low + k] = temp[k];

}

**Output:**

**Enter n value 20**

**Enter 20 values in array**

415 440 845 535 420 555 984 509 11 561 900 413 195 963 566 305 2 169 547 607

**Elements after sorting**

2 11 169 195 305 413 415 420 440 509 535 547 555 561 566 607 845 900 963 984

**Linear Search**

Linear search is also called as **“Brute Force” method.** This is a simplest search technique. Assuming the search list is an array, the approach is to iterate through all the elements until a match is found. Although linear search is highly inefficient, it is also the basis of virtually all search techniques when the search list is not ordered.

We begin search by comparing the first element of the list with the target element. If it matches, the search ends. Otherwise, we will move to next element and compare. In this, the target element is compared with all the elements until a match occurs or when there are no elements left to be compared. This method requires no ordering of elements in the list.

**For example, consider the following list of elements:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **8** | **5** | **9** | **15** | **43** | **2** | **17** | **65** | **23** | **11** |

Suppose we want to search the element 17 (i.e key=17), we first compare the key with the first element which is 8. Since both are not matching, we have to move to next element and compare. By repeating this process we will find the key after 7 comparisons. That means, the desired key is at 7th position in the list. On the other hand, suppose the key is 31, the comparison test fails from first element to last element resulting in an unsuccessful search. This is because the key is not present in the given list.

**Algorithm**

**Function LinearSearch(A,N,Key)**

1. i=1
2. while(i<=N) do

begin

1. if(A[i]=key) then

begin

1. print “Target element found at position”,i
2. return i

end

1. i=i+1

end

1. print “Target element not found – Unsuccessful search”
2. return (-1)

**// program to perform Linear search**

#include<stdio.h>

#include<conio.h>

main()

{ int a[100],n,i,s,f=0;

clrscr();

printf("Enter n value");

scanf("%d",&n);

printf("Enter %d Values in array",n);

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

scanf("%d",&a[i]);

printf("Enter the value to search");

scanf("%d",&s);

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

{

if(a[i]==s)

{ printf(" %d Element Found at %d location",s,i);

f=1;

break;

}

}

if(f==0)

printf("Element not found- Search is unsuccessful");

getch();

}

**Binary Search**

The pre-requisite for Binary Search method is that the input element list is in the sorted order. The method starts with looking ag the middle element of the list. If it matches with the target element, then the search is complete. Otherwise, the target element may be in the upper half or lower half. The search progresses with the upper half if the target element is less than the middle element or with the lower half if the target element is greater than the middle. The process is continued until the target element is found or when the portion of the sub-list to be searched is empty.

**Method:**

In binary search, we start by taking look at the approximate middle element. If the number of elements in the table is odd, then the middle element is considered. In case, the table has even number of elements then the last element of the first half is taken into consideration. See the figure for deciding the middle element in both the cases.

We compare the middle element with the one we wish to search in the table. If value of middle element is greater than the element to be searched then we discard the first half of the table and start all over again by taking the second half of the table. If the value of middle element is smaller than element to be searched, then we discard the second half of the table. We keep on looking for the middle element of the table and keep on discarding either the first or the second half of the table depending on the result of the comparison of the middle element and the element to be searched, till we find the required element when the table cannot be further subdivided.

**Algorithm**

**Function BinarySearch(A,N,key)**

1. low=1
2. high=N
3. **while low<=high do**

**begin**

1. mid=trunc((low+high)/2)
2. **if key=A[mid] then**

**begin**

1. print “Target element found at position”, mid
2. return (mid)

**end**

**else**

1. **if key<a[mid] then**

high=mid-1

**else**

low=mid+1

**end**

1. **print “Target element not found – Unsuccessful search”**
2. **return -1**

**// Program to perform binary search**

#include<stdio.h>

#include<conio.h>

main()

{ int a[100],n,i,s,f=0,low,mid,high;

clrscr();

printf("\n Enter n value");

scanf("%d",&n);

printf("Enter %d Values in array",n);

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

scanf("%d",&a[i]);

printf("Enter the value to search");

scanf("%d",&s);

low=0;

high=n-1;

while(low<=high)

{

mid=(low+high)/2;

if(a[mid]>s)

high =mid-1;

else

if(a[mid]<s)

low = mid + 1;

else

if(a[mid]==s)

{

printf("Element Found at =%d"+ mid);

f=1;

break;

}

}

if(f==0)

printf("Element Not found - Unsuccessful Search");

getch();

}

**Indexed Sequential Search**

In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time because it is located in a specified group.

**Note:** When the user makes a request for specific records it will find that index group first where that specific record is recorded.

### Characteristics of Indexed Sequential Search:

* In Indexed Sequential Search a sorted index is set aside in addition to the array.
* Each element in the index points to a block of elements in the array or another expanded index.
* The index is searched 1st then the array and guides the search in the array.

**// C program for Indexed Sequential Search**

#include <stdio.h>

#include <stdlib.h>

void indexedSequentialSearch(int arr[], int n, int k)

{

int elements[20], indices[20], temp, i, set = 0;

int j = 0, ind = 0, start, end;

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

{

elements[ind] = arr[i];

indices[ind] = i;

ind++;

}

if (k < elements[0])

{

printf("Not found");

exit(0);

}

else

{

for (i = 1; i <= ind; i++)

if (k <= elements[i])

{ start = indices[i - 1];

end = indices[i];

set = 1;

break;

}

}

if (set == 0)

{

start = indices[i - 1];

end = n;

}

for (i = start; i <= end; i++)

{

if (k == arr[i])

{

j = 1;

break;

}

}

if (j == 1)

printf("Found at index %d", i);

else

printf("Not found");

}

void main()

{

int arr[] = { 6, 7, 8, 9, 10 };

int n = sizeof(arr) / sizeof(arr[0]);

int k = 8;

clrscr();

indexedSequentialSearch(arr, n, k);

getch();

}

**GRAPHS**

Graph is an important data structure mostly used in application like analysis of project planning, finding the shortest routes, analysis of electrical circuits, identification of chemical compounds, statistical machines etc., This structure was first introduced by the mathematician ***L.Euler* in 1736.**

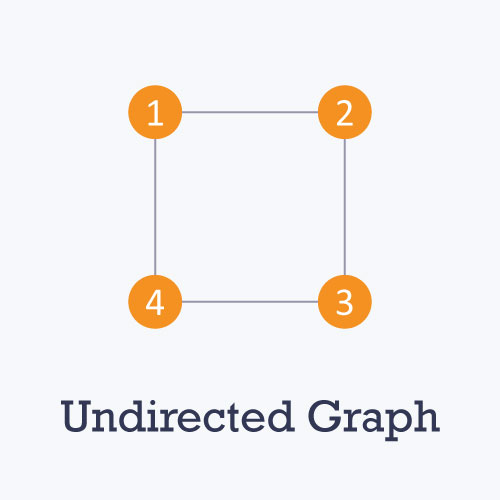
Graphs are mathematical structures and are found very useful in problem solving. Unlike trees, which have a strict hierarchical structure, graphs are more flexible. Graphs can have loops and parts may be disconnected.

A graph G consists of two sets called the nodes N and the edges E. N is a finite non-empty set of nodes and E is a finite set of pairs of nodes. Graphs can be classified as Directed graphs and undirected graphs. A graph can be represented as

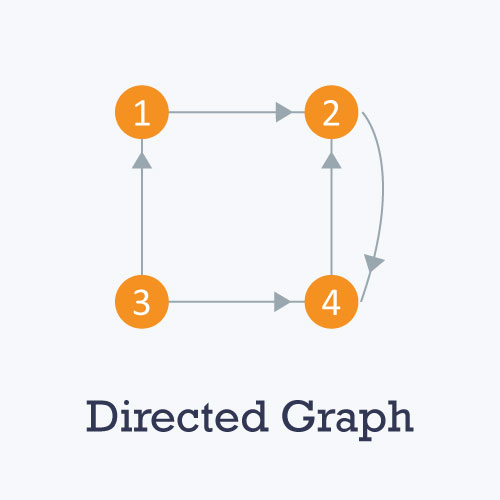
G=(V,E)

**Types of graphs**

* **Undirected**: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.



* **Directed:** A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



* **Weighted:** In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:
  + 1 -> 2 -> 3
  + 1 -> 3
  + 1 -> 4 -> 3

Therefore the total cost of each path will be as follows: - The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units



* **Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

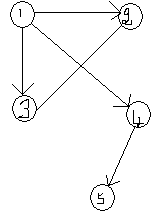
# Graph Representations

Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

# Adjacency Matrix

Graphs can be represented using sequential representation method. This method uses a square table with n rows and n columns, where n is the number of nodes of the graph. This table is called as the adjacency matrix.



For the above graph,

Set of nodes N = {1,2,3,4,5}

Set of edges E ={<1,2>,<1,3>,<1,4>,<2,3>, <4,5>}

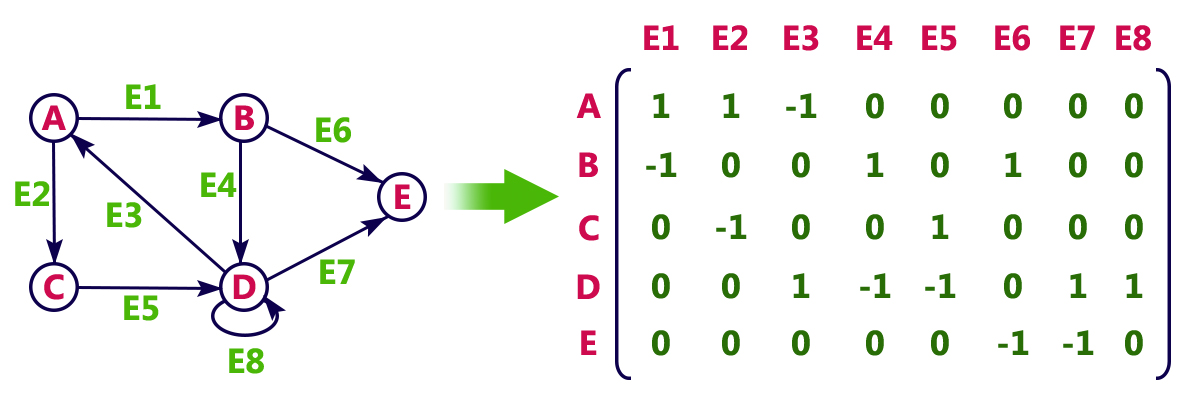
Let us write the adjacency matrix for the directed graph. There are 5 nodes in the graph and hence, the matrix size in 5 x 5. In the filling of adjacency matrix elements, the element (i, j) = 1 if <i, j> is an edge, otherwise the element value is 0. Thus, the elements of adjacency matrix are either 1 or 0 and the total number of 1’s in the matrix is equal to the total number of edges of the graph. Applying the above method to the example graph, the adjacency matrix will be

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | V1 | V2 | V3 | V4 | V5 |
| V1 | 0 | 1 | 1 | 1 | 0 |
| V2 | 0 | 0 | 1 | 0 | 0 |
| V3 | 0 | 0 | 0 | 0 | 0 |
| V4 | 0 | 0 | 0 | 0 | 1 |
| V5 | 0 | 0 | 0 | 0 | 0 |

# Incidence Matrix

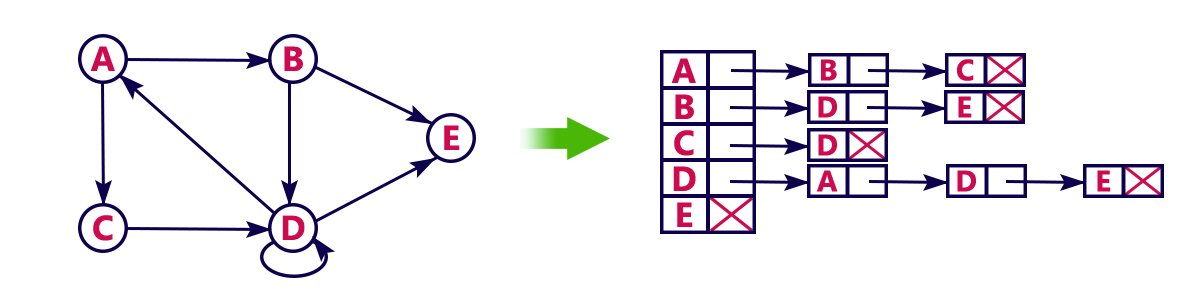
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation...

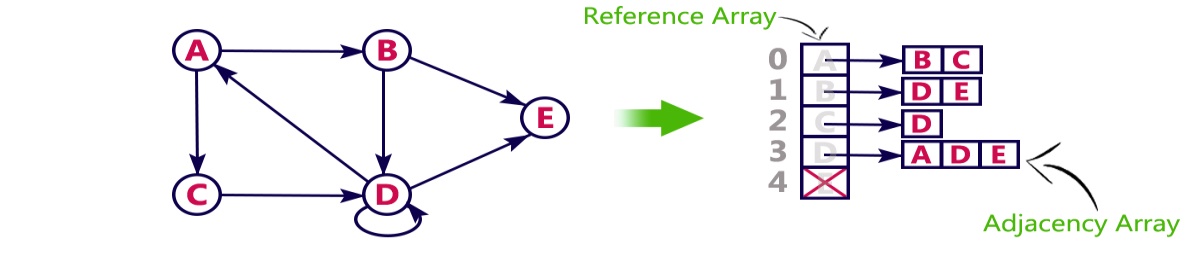


# Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.  
For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



**TRAVERSING OF A GRAPH**

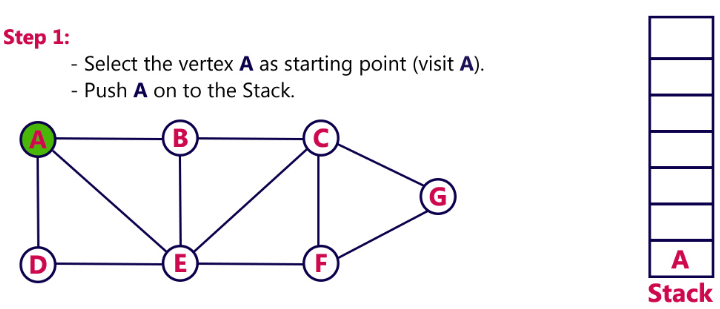
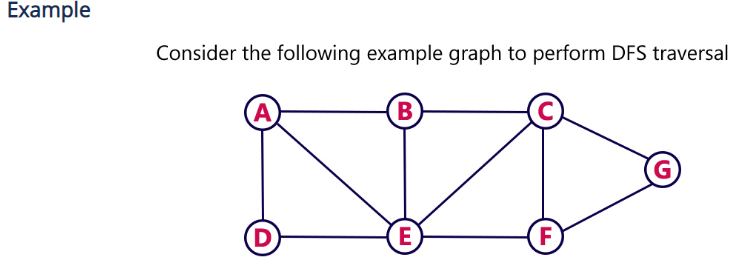
Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.There are two graph traversal techniques and they are as follows.

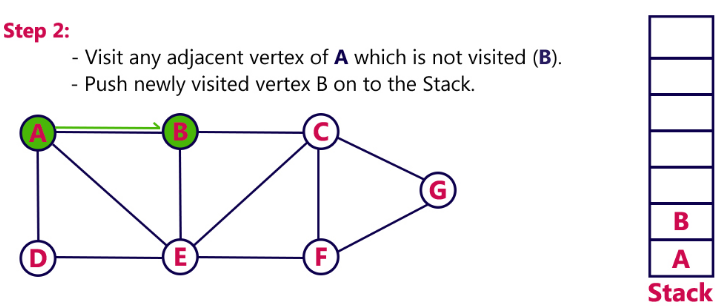
1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

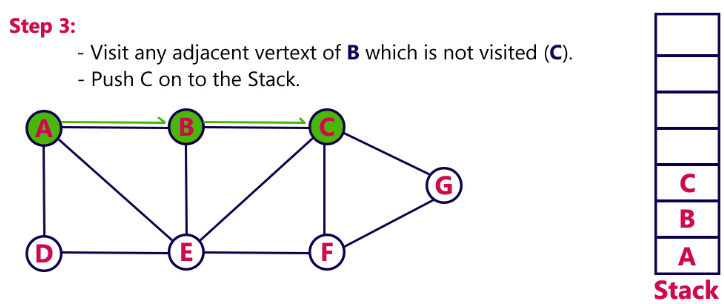
# DFS (Depth First Search)

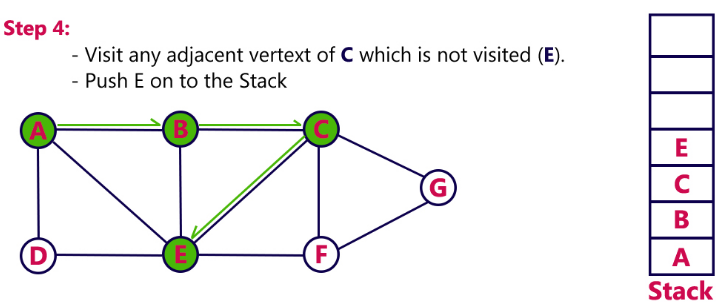
DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.We use the following steps to implement DFS traversal.

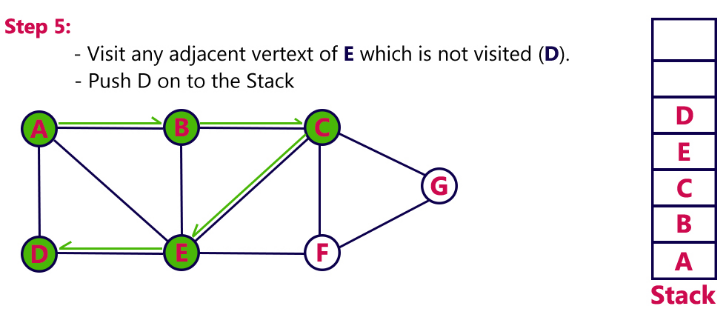
* **Step 1 -**Define a Stack of size total number of vertices in the graph.
* **Step 2 -**Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
* **Step 3 -**Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
* **Step 4 -**Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
* **Step 5 -**When there is no new vertex to 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

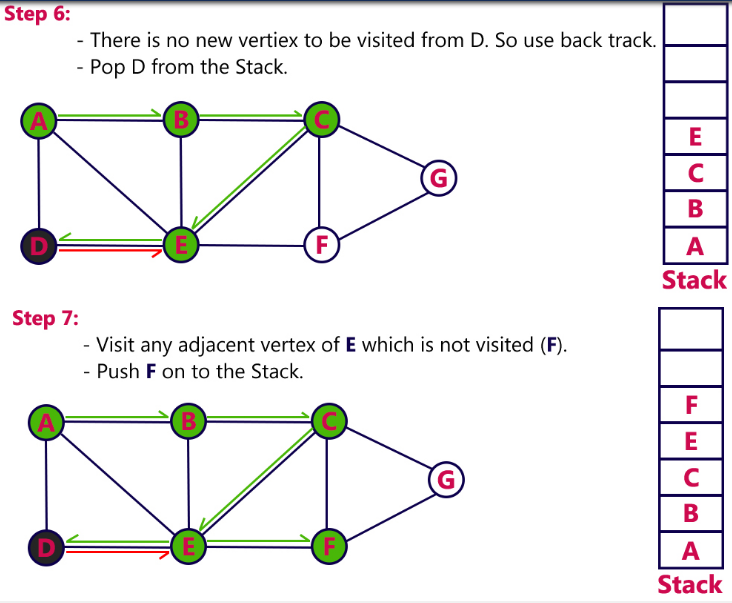


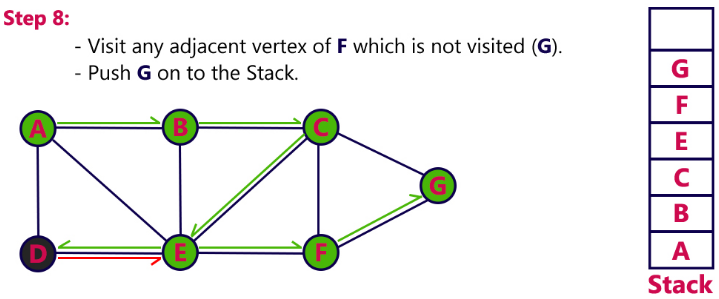


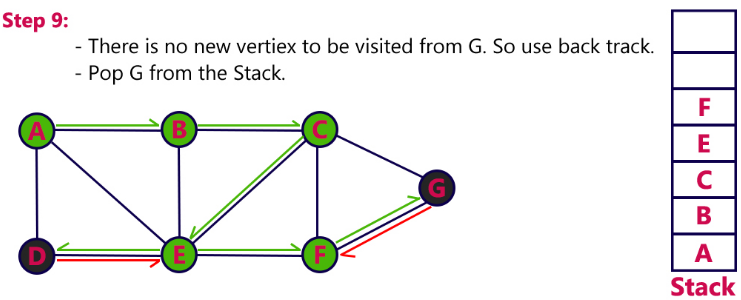


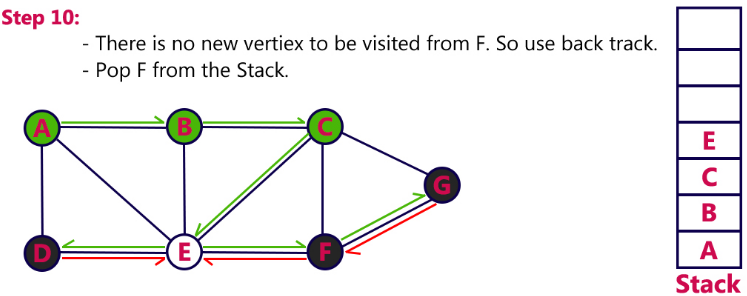


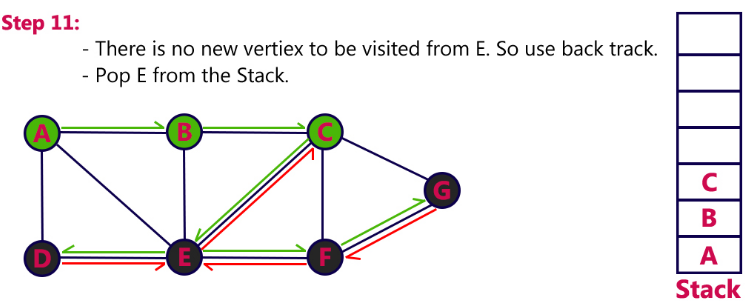


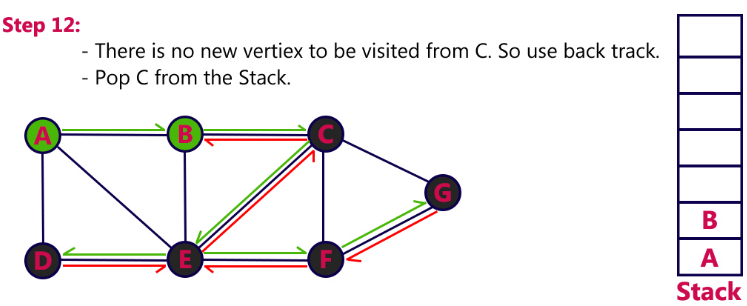


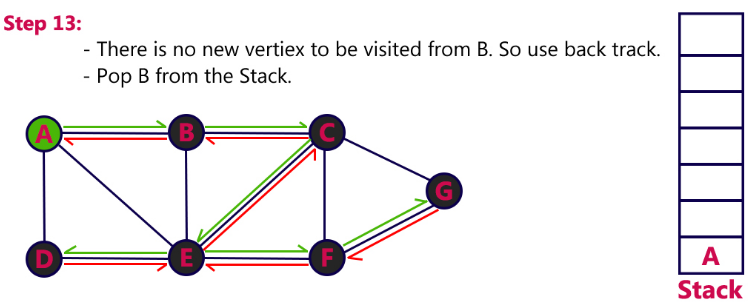


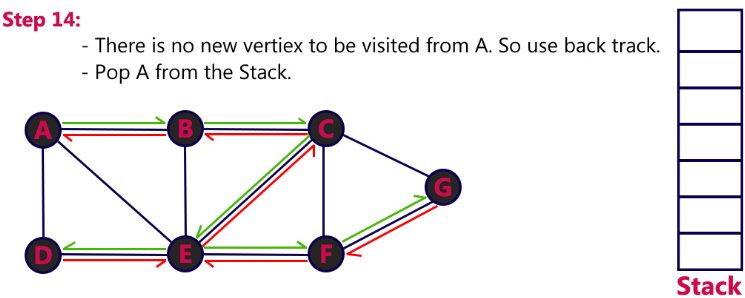


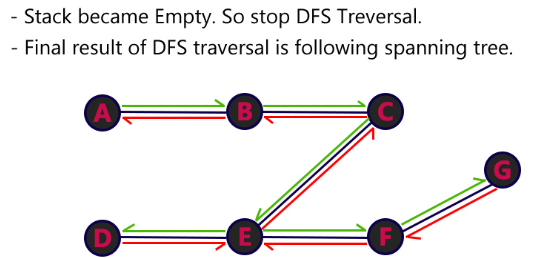










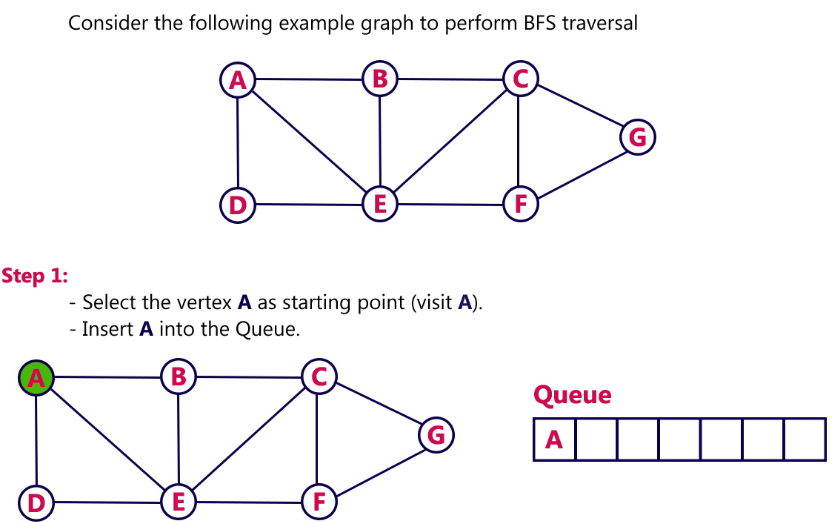


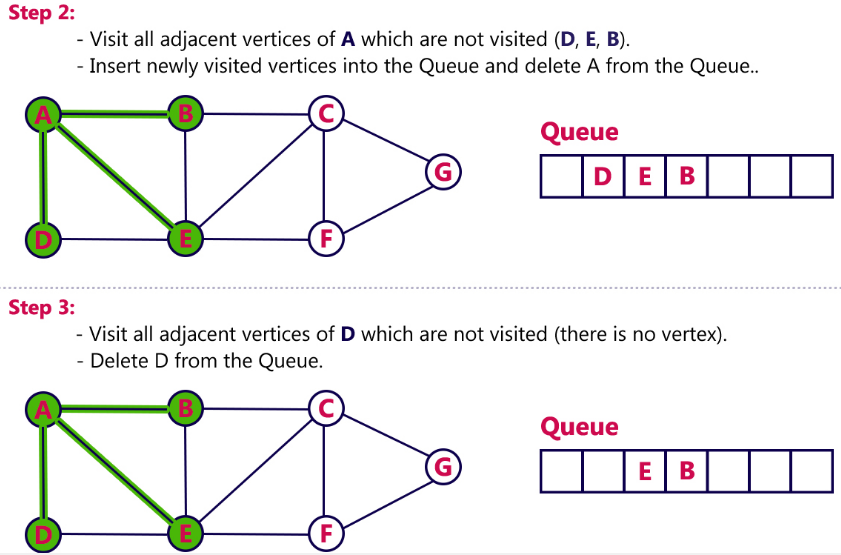
# BFS (Breadth First Search)

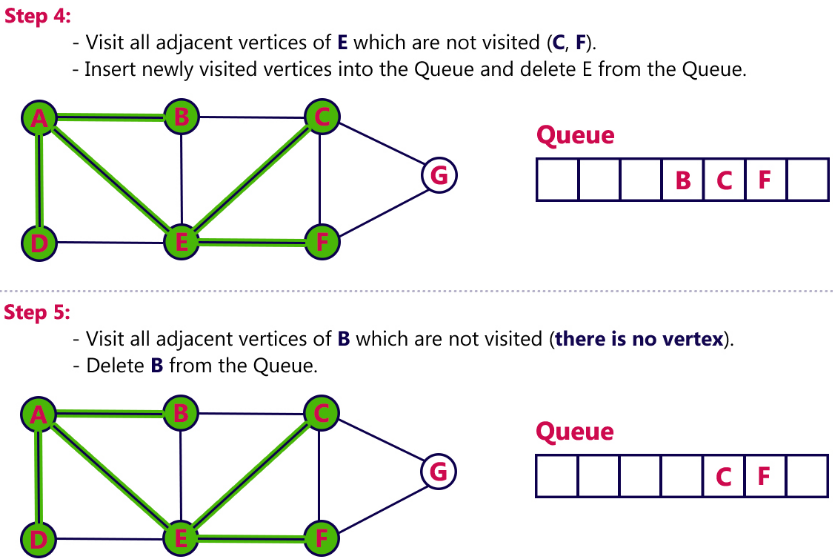
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.We use the following steps to implement BFS traversal.

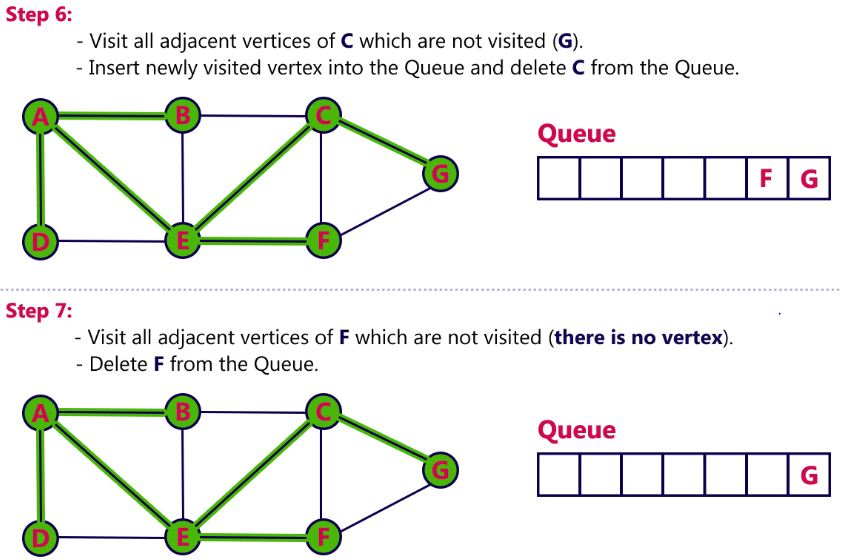
* **Step 1 -**Define a Queue of size total number of vertices in the graph.
* **Step 2 -**Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
* **Step 3 -**Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
* **Step 4 -**When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
* **Step 5 -**Repeat steps 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

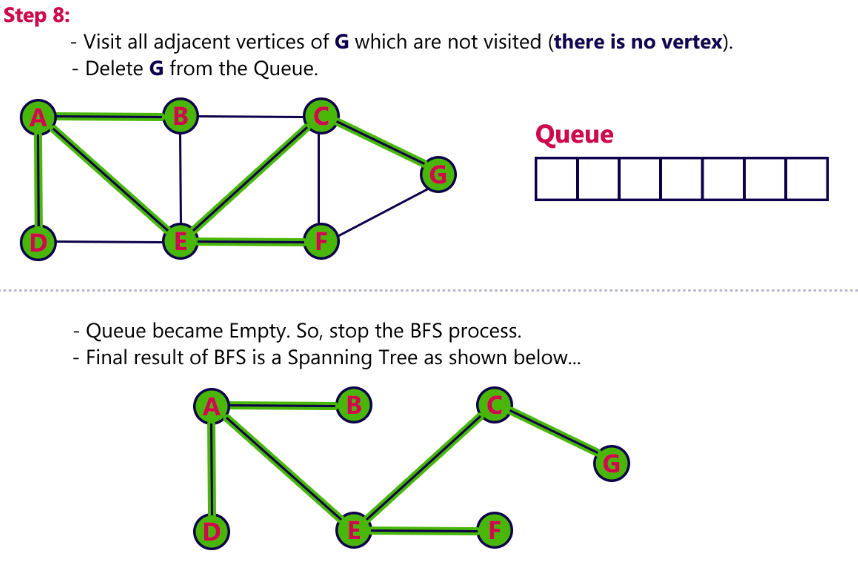
##### **Example**

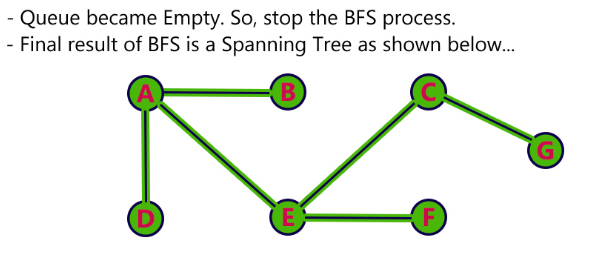












**Spanning Tree**

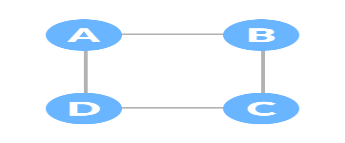
* A graph which contains all vertices with minimum number of edges

**or**

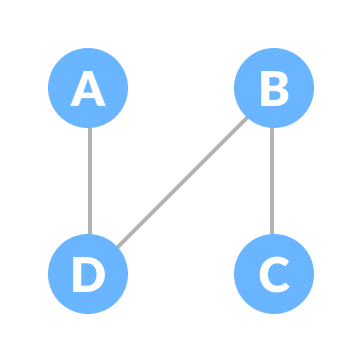
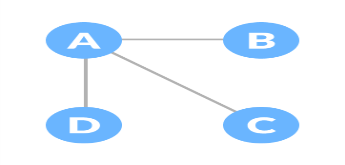
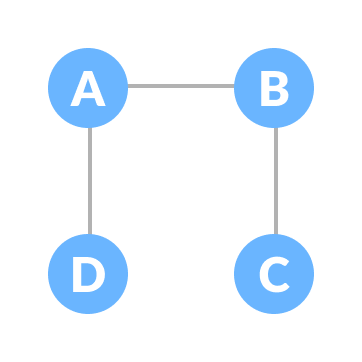
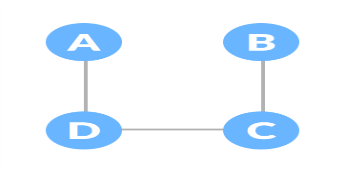
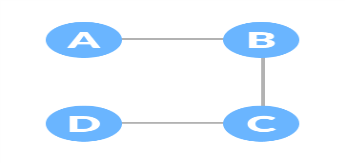
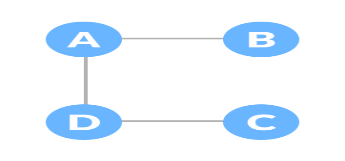
* A spanning tree is a sub-graph of an undirected and a connected graph, which includes all the vertices of the graph having a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.
* The edges may or may not have weights assigned to them.
* The total number of spanning trees with n vertices that can be created from a complete graph is equal to n(n-2).

## Example:

Let the original graph be:



Some of the possible spanning trees that can be created from the above graph are:



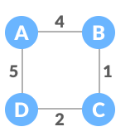
We have n = 4, thus the maximum number of possible spanning trees is equal to 44-2 = 16. Thus, 16 spanning trees can be formed from the above graph.

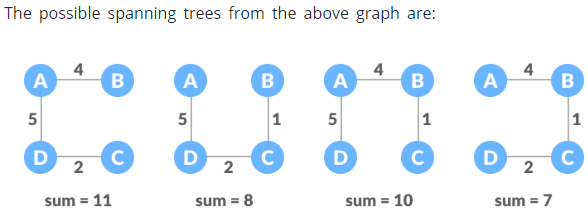
**Minimum/Minimal Spanning Tree:**

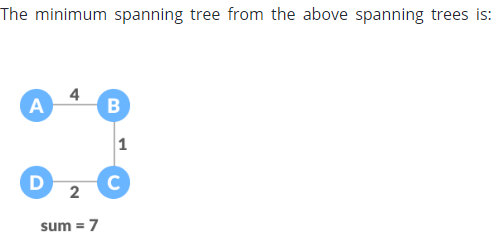
The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

**Eg-1:**

The initial Graph is:







Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

There are two famous algorithms for finding the Minimum Spanning Tree:

1. **Kruskal’s Algorithm**
2. **Prim’s Algorithm**

**Kruskal’s Algorithm**

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

## Algorithm

* **Step 1:** Create a forest in such a way that each graph is a separate tree.
* **Step 2:** Create a priority queue Q that contains all the edges of the graph.
* **Step 3:** Repeat Steps 4 and 5 while Q is NOT EMPTY
* **Step 4:** Remove an edge from Q
* **Step 5:** IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).  
  ELSE  
  Discard the edge
* **Step 6:** END

### Example:

**Apply the Kruskal's algorithm on the graph given as follows.**



**Solution:**

the weight of the edges given as :

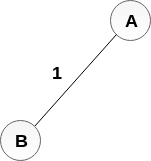
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Edge** | **AE** | **AD** | **AC** | **AB** | **BC** | **CD** | **DE** |
| Weight | 5 | 10 | 7 | 1 | 3 | 4 | 2 |

Sort the edges according to their weights.

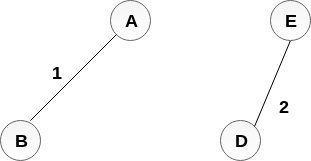
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Edge** | **AB** | **DE** | **BC** | **CD** | **AE** | **AC** | **AD** |
| Weight | 1 | 2 | 3 | 4 | 5 | 7 | 10 |

Start constructing the tree;

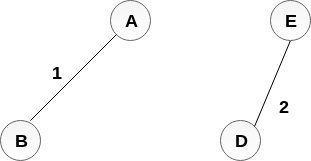
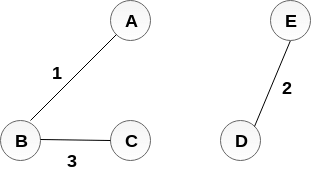
**Add AB to the MST;**



**Add DE to the MST;**



**Add BC to the MST;**

  
  
The next step is to add AE, but we can't add that as it will cause a cycle.

The next edge to be added is AC, but it can't be added as it will cause a cycle.

The next edge to be added is AD, but it can't be added as it will contain a cycle.

the cost of MST = 1 + 2 + 3 + 4 = 10.

# Prim's Algorithm

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

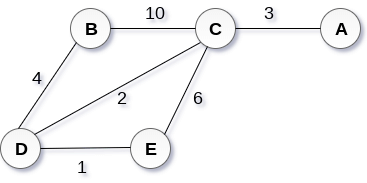
Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

## Algorithm

* **Step 1:** Select a starting vertex
* **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
* **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
* **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T  
  [END OF LOOP]
* **Step 5:** EXIT

### Example:

Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.



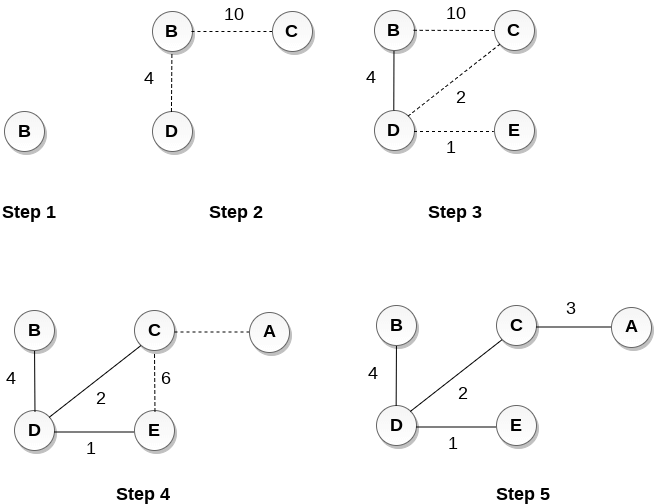
## Solution

* **Step 1 :** Choose a starting vertex B.
* **Step 2:** Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.
* **Step 3:** Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
* **Step 3:** Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.
* **Step 4:** Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

The cost of MST will be calculated as;

cost(MST) = 4 + 2 + 1 + 3 = 10 units.



**Applications of Graph**

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

* + - 1. **Social network graphs:**

Graphs that represent who knows whom, who communicates with whom or other relationships in social structures

* + - 1. **Transportation networks:**

Graph networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps to find the best routes between locations.

* + - 1. **Utility graphs:**

The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them.

1. **Document link graphs:**

The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge.

1. **Network packet traffic graphs:**

Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them.

1. **Scene graphs.**

In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

1. **Robot planning**

Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles

1. **Semantic networks.**

Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

1. **Graphs in compilers.**

Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.