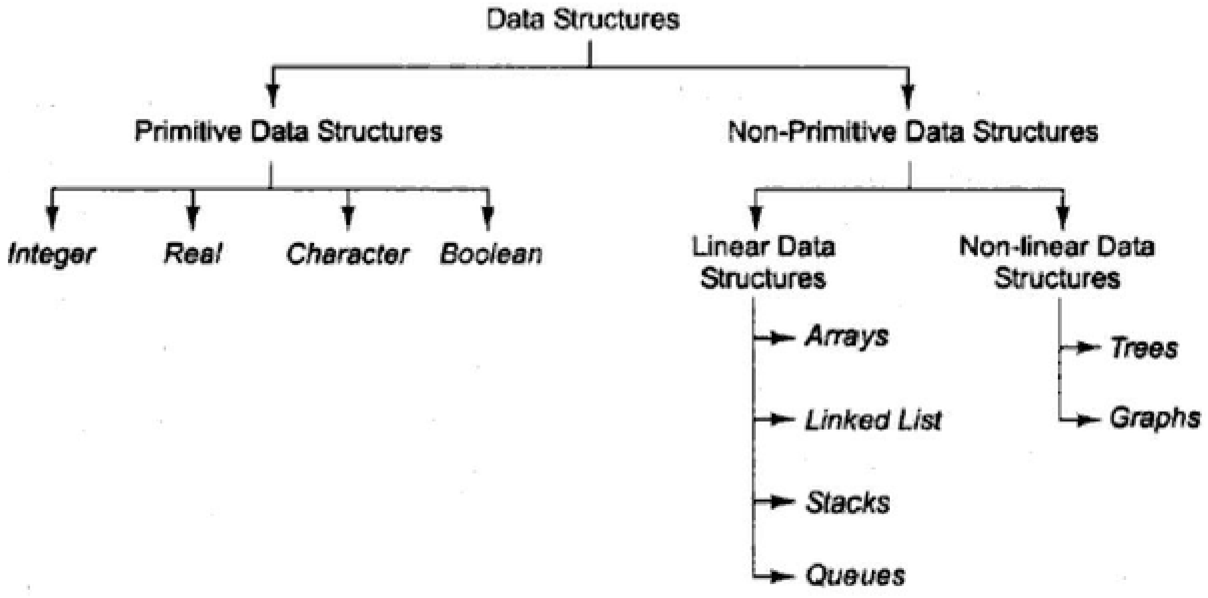
**Syllabus**:

# UNIT-I

**Abstract Data Types and the C++ Class:** An Introduction to C++ Class, Data Abstraction and Encapsulation in C++, Declaring Class Objects and Invoking Member Functions, Special Class Operations (Constructor, Destructor and Operator Overloading), Miscellaneous Topics (Struct, Union, Static), ADTs and C++ Classes**, The Array as an Abstract Data Type**, **The Polynomial Abstract Data type:** Introduction, Polynomial Representation, Polynomial Addition. **Spares Matrices:** Introduction, Sparse Matrix Representation, Transposing a Matrix, Matrix Multiplication, **Representation of Arrays**

## DATA STRUCTURE:-

A data structure is the way of organizing and storing the data into the computer, for efficient accessing or retrieval of data.



**Primitive Data Structures:** The data structures that are not defined by any other data structures, The basic data types are int, real, char, Boolean.

**Non-Primitive Data Structures:** The data structures that are defined by any other data structures. Again it is categorized into 2 types.

1. **Linear Data Structures**: A data structure is said to be linear if its elements form a sequence or a linear list.

   Examples: Array, Linked List, Stacks, Queues
2. **Non-Linear Data:** A data structure is said to be Non-linear if its elements form a not sequence or non linear list.

   Examples: Trees, Graphs

**Operations on Linear and Non Linear Data Structures**

- **Traversal** : Visit every part of the data structure
- **Search** : Traversal through the data structure for a given element
- **Insertion** : Adding new elements to the data structure
- **Deletion**: Removing an element from the data structure.
- **Sorting** : Rearranging the elements in some type of order(e.g Increasing or Decreasing)
- **Merging** : Combining two similar data structures into one

## Abstract Data Types and the C++ Class:-

### Introduction to C++ Class:

- Class is a blueprint of data members and member functions. Class is a user defined data type like structures and unions in C.
- By default class variables are private.

Syntax

```
class class_name
{
private:
         //data members and member functions declarations
public:
        //data members and member functions declarations
protected:
       //data members and member functions declarations
};
```

Example

```
class student
{
   public:
        char name[30];
        int roll_no, age;
        void getData() ;
        void display();
};
```

1) a class name: student

2) Data members: the data that makes up the class (e.g. name. roll_no and age).

3) Member functions: the set of operations that may be applied to the objects of a class (e.g. getData(), display() ).

4) Levels of program access: these control the level of access to data members and member functions from program code that is outside the class. There are three levels of access to class members: public(Access outside of the class), protected(Immediate derived class can access) and private(can't access outside of the class).

## <u>Declaring Class Objects and Invoking Member Functions:-</u>

- The class objects are declared and created in the same way that variables are declared and created.
- The data members and member functions of class can be accessed using the dot ( . ) or arrow (→) operator with the object.
- The public data members are accessed, the private data members are not allowed to be accessed directly by the object, it can access by public member functions.
- While implementing the member functions outside of the class we use special reference called **scope resolution operator (::)**.

**<u>Example:</u> Implement a program on basic concepts of C++ and Data Abstraction and Encapsulation.**

```
#include <iostream.h>
class student
{
   private:
        char name[30];
        int roll_no, age;
   public:
        void getData() ;
        void display();
};
void student::getData( )
{
cout<<"Enter Roll number, Age, and Name of
student: ";
cin>>roll_no>>age>>name;
}
void student::display( )
{
cout<<"RollNumber: "<<roll_no<<endl;
cout<<"Age: "<<age<<endl;
cout<<"Name of student: "<<name;
}
```

```
int main()
{
        //declaring object to the class number
        student S;

        S.getData();
        S.display();

        //declaring pointer to the object
        student *ptrS;
        ptrS = new student;
        //creating & assigning memory

        //calling    member    function    with
pointer
        ptrS->getData();
        ptrS->display();

        return 0;
}
```

## Data abstraction and Encapsulation:-

- The binding the data and functions into a single unit called class is known as **encapsulation**.
- The data is not accessible to the outside world, and only those functions which are enclosed in the class can access it.
- Abstraction is seperating the logical properties from their implementation. It shows only the essential features of the application and hiding the details.
- In C++, classes can provide methods to the outside world to access & use the data variables, keeping the variables hidden from direct access. This can be done using access specifiers(**private**).

**Constructor, Destructor:-** These are special member functions of the class.

| | |
|---|---|
| • Constuctor is a member function, which **initialize the data members** of an object. | • Destuctor is a member function, which **freeing the memory associated data members** of the class |
| • It call automatically when object of the class is created. | • It call automatically when the lifetime of an object ends |
| • It will write in public access specifier, name is same as class name and not specify any retun type or values. | • It will write in public access specifier, name is same as class name, not specify any retun type or values, not pasing any parameters and preceeded by **~(tield).** |
| • If we not define, memory is allocated to the data members but not initialized so it may contain **undefined** values. | • If we not define, freeing the memory but if data members is pointing to some other objects, that pointing is not deleted. |

**Example: Implement a program on Constructors and destructors using C++**

```
#include<iostream.h>
class StudentCon
{
   private:
            int rollno,age;
   public:
         /* Default constructor */
         StudentCon()
         {
         cout<<"Default Constructor"<<endl;
         rollno=0;age=0;
         }
```

```
/* Parameterized constructor */
StudentCon(int r, int a)
{
 rollno = r;
 age = a;
}
/* Copy constructor */
StudentCon (const StudentCon &s2)
{
rollno = s2.rollno;
age    = s2.age;
}
```

```
        /* Destructor */
        ~StudentCon()
        {
        cout<<"Destructor  has  called"<<endl;

        }
        void display()
        {
        cout<<rollno<<" "<<age<<endl;
        }
};
```

```
int main()
{
   StudentCon s1;        // Default constructor
   StudentCon s2(545, 28);    // Parameterized
   StudentCon s3 = s2;      // Copy constructor
   cout<<"Parameterized constructor : ";
   s2.display();
   cout<<"Copy constructor : ";
   s3.display();

   return 0;
}
```

## Operator Overloading (Polymorphism):

- **Operator overloading** is a compile-time polymorphism in which the operator is overloaded to provide the **special meaning to the user-defined data type**.
- To overload an operator must write a function in **public** access specifies.

**Syntax:** returnType operator symbol (arguments)

**{ ... .. ... }**

**This pointer:**

- 'this' pointer is a constant pointer that holds the memory address of the current object. If ‘this’ pointer is written in member function of a class, it can access data members of that class for given object.

**Friend Function:**

- It is non-member function of a class but can access all members of a class (Both public and private data)
- While defining friend function out of the class, no need to use scope resolution (: :) operator and class name.

**Syntax:**

| | |
|---|---|
| Declaration with in class: | **friend returnType function_name(parameters);** |
| Calling through main: | **function_name(parameters);** |
| Definition outside of the class: | **returntype function_name(parameters) { - - - }** |

**Example:** Implement a program on operator overloading, this pointer and friend function using C++

```
#include <iostream.h>
class Test
{
  private:
        int count;
        int a,b;
  public:
        void get() {count=5;}
        void put()
        {  cout<<"Count: "<<count<<endl;  }
        void operator ++()
        {  count = count+1;  }

        Test input(int a,int b)
        {
        this->a=a+b;
        this->b=a-b;
        return *this;
        }
         void output()
        { cout<<"a,b values are:";
        cout<<a<<" "<<b<<endl;}

        friend void dosth(Test);
};
```

```
void dosth(Test t)
{
t.count=t.count+100;
cout<<"total count: "<<t.count;
}

int main()
{
   Test t;
   t.get();
   t.put();
   ++t;
   t.put();


   Test t1,t2;
   t2=t1.input(10,5);
   t1.output();
   t2.output();

   dosth(t);
   return 0;

}
```

## MISCELLANEOUS TOPICS:

**Structure:**

- Both the structure and classes are almost equivalent.
- Structure is collection of different data items. The default access specifier is **public**.
- Class is blue print of data members and member functions. The default access specifier is **private.**

**Union:**

- Union reserves the storage for one of its largest data member at run time.
- Unions are used in Linked List and Inheritance.

**Static:**

- Static data member is similar to global variable throughout the class, because it is not similar data member in the class.
- Each object doesn't have its own copy; we have **only one copy**, it shares by all objects of the class.

**Examples:**

```
struct student
{
int rollno, age, marks;
char name[20];
}S1; // it allocates 26 bytes of memory
```

```
union student
{
int rollno, age, marks;
char name[20];
}S2; // it allocates only 20 bytes of memory
```

```
class func
{  public:
     func() {
          int a=10;
          static int b=5;
          a++; b++;
          cout<<a<<b; }   };
```

```
void main()
{
func f1; //11,6
func f2; //11,7
func f3; //11,8
func f4; // 11,9
}
```

**ADT (Abstract Data Type):-**

- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- The process of providing only the essentials and hiding the details is known as abstraction

The model is only focusing on the following.

- The Data, which are affected by the program
- The Operations, which are identified by problem

**ADT Notation**:

```
ADT type_data_structure
{
Instances/Objective/Data: Describes the structure of the data used in ADT
Operations: Describe the valid operation for this ADT
}
```

**Example:** List ADT

ADT List

{

**Instances:** A list contains elements of same type arranged in sequential order

**Operations:**

| | |
|---|---|
| isEmpty() | – Return true if the list is empty, otherwise return false. |
| isFull() | – Return true if the list is full, otherwise return false. |
| size() | – Return the number of elements in the list. |
| get() | – Return an element from the list at any given position. |
| insert() | – Insert an element at any position of the list. |
| remove() | – Remove the first occurrence of any element from a non-empty list. |
| replace() | – Replace an element at any position by another element. |

}

## ARRAY:-

- The **array** is a basic abstract data type that holds an ordered collection of items accessible by an integer index. These items can be anything from primitive types such as integers to more complex types.
- Since it's an ADT, it doesn't specify an implementation

**ADT ARRAY**

```
{       Instances:
                Set of ordered pair (index, value), no two pairs having same index
                Index ←one dimension / two dimensional / multi dimensional array
                Value ← primitive data types like int, float..
        Operations:
                Create();               // create an empty array
                Store(index,value);     //add or replace the pair into an array
                Retrieve(index);        //return the pair with index value
}
```

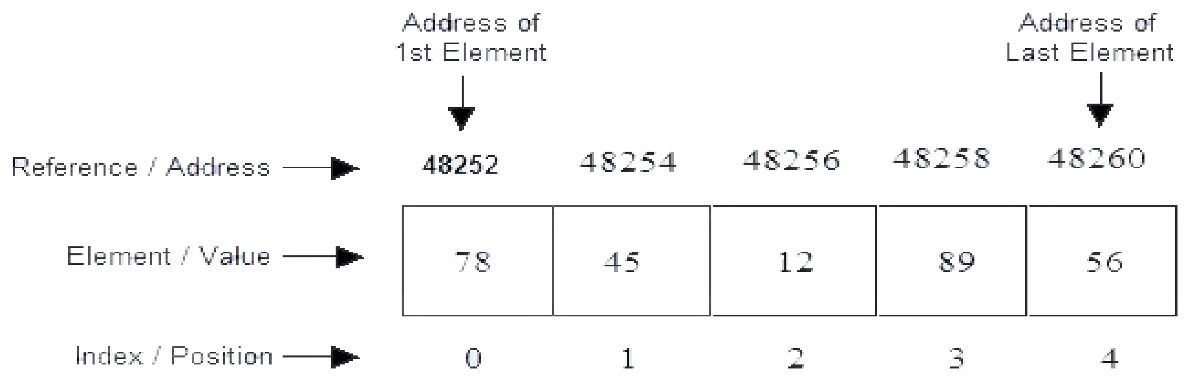
**Example: Implementation of an Array ADT using C++**

```
#include <iostream.h>
class Array
{
	int a[100];
	int size;
public:
	void create(int size);
	void store(int index, int value);
	int retrive(int index);
	int search(int v);
	void display( );
};
void Array::create(int s)
{
a= new int[s];
size= s;
}
void Array::store(int i, int v)
{
if(i< size)
	a[i]= v;
}
int Array::retrive(int i)
{
return(a[i]);
}
int Array::search(int v)
{
	for( int i=0; i < size; i++)
	  if(a[i] == v)
		return i;
	return (-1);
}
void Array::display( )
{
	cout<<"Array Elements are: ";
	for( int i=0; i< size; i++ )
		cout<< a[i]<<" ";
}
int main()
{
	Array a1;
	a1.create(10);
	a1.store(0,5);
	a1.store(1,15);
	a1.store(2,25);
	a1.store(3,35);
	a1.store(4,45);
	cout<<"Retrived Element:";
	a1.retrive(2);
	cout<<"Searched Element:";
	a1.search(75);
	a1.display();
	return 0;
}
```

## Storage Representation of Array:

### One Dimensional Array:

- An array is a collection of items stored at contiguous memory locations.
- This makes it easier to calculate the position of each element i.e., the memory location of the first element of the array (Base Address).



Address of an element we can find easily

A[index] = Base Address + (Size of data type * index)

### Multi Dimensional Arrays:

- A multi-dimensional **array** is **an array of arrays**. 2-dimensional arrays are the most commonly used. They are used to store data in a tabular manner (mXn).
- It can be represented in memory using any of the following two ways
  1. Column-Major Order
  2. Row-Major Order

**Column-Major Order**:

- In this method the elements are stored column wise, i.e. m elements of first column are stored in first m locations, m elements of second column are stored in next m locations and so on

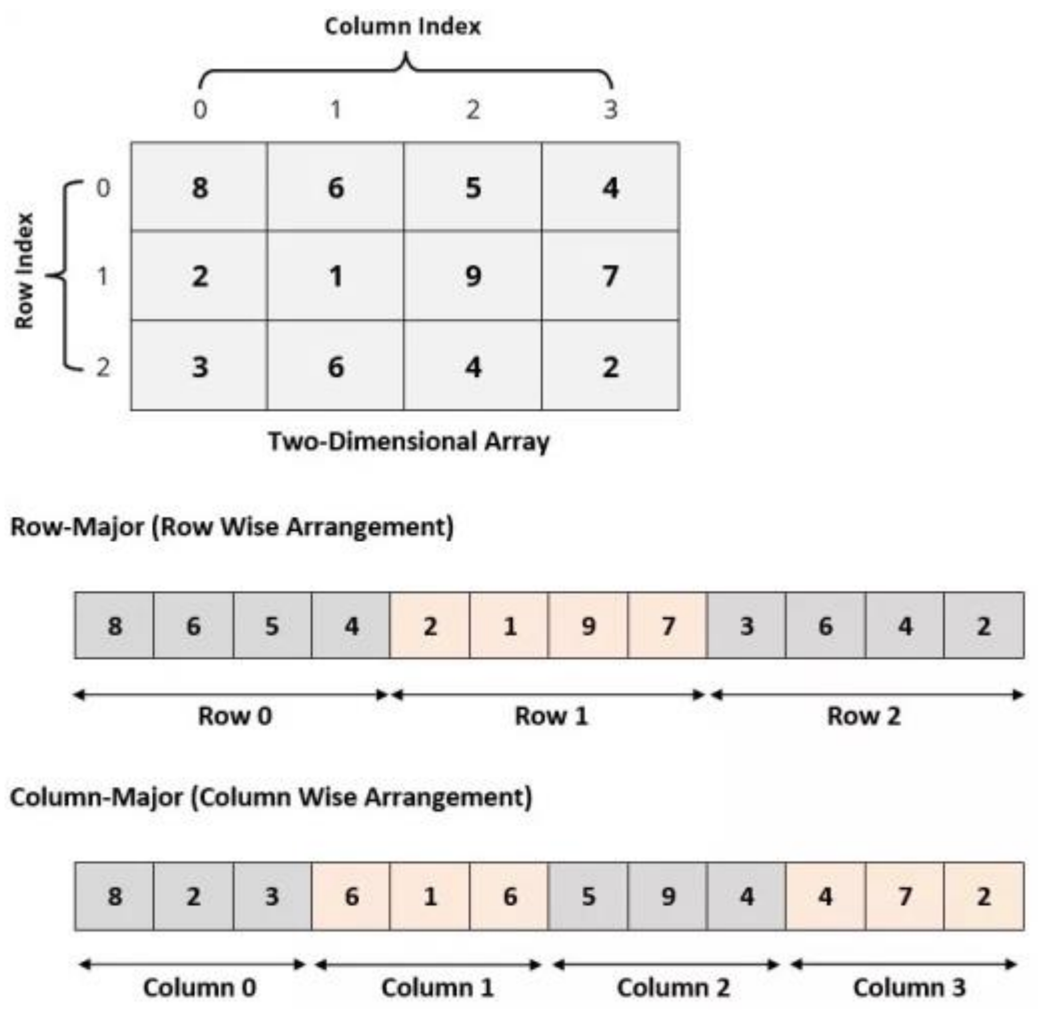
  **Address of A [ i ][ j ] = BaseAddress + Size of datatype * [ i  + (m *j)]**

**Row-Major Order:**

- In this method the elements are stored row wise, i.e. n elements of first row are stored in first n locations, n elements of second row are stored in next n locations and so on

  **Address of A [ i ][ j ] = BaseAddress + Size of datatype * [ ( i * n)  + j ]**

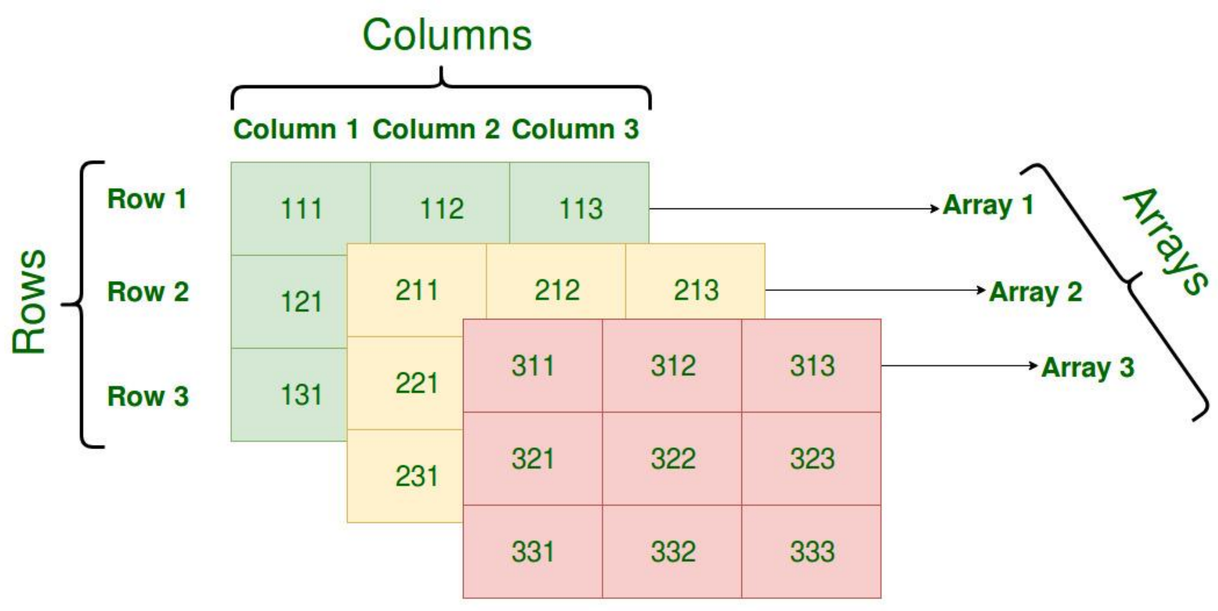
Example:



Row major Order:Address of A[2][3] = 1000+2 *[(2*4)+3]= 1000+22=1022

Column major Order:Address of A[1][2] = 1000+2 *[1+(3*2)]= 1000+14=1014

**Three Dimensional Arrays: array** is an **array** of **arrays** of **arrays (rows, columns, depth)**



## Polynomial Abstract Data Type:-

Polynomial is a result of terms, where each has a form $aX^e$, where “X” is a variable, “a” is the coefficient, “e” is the exponent.

ADT Polynomial

{

Instances:

Set of ordered pair of coefficient and exponent

//$P(X) = a_0X^{e_0} + a_1X^{e_1} + \cdots + a_nX^{e_n}$

//Where $a_i$ is nonzero float coefficient and exponent $e_i$ is non negative integer

Operations:

*Polynomial();* //construct the polynomial p(x)=0

*Polynomial Addpoly(Polynomial A, Polynomial B);*
//returns result of two polynomials

*Polynomial Subpoly(Polynomial A, Polynomial B);*
//returns subtraction of two polynomials

*Polynomial Multpoly(Polynomial A, Polynomial B);*
//returns multiplication of two polynomials

*flaot Evaluate(flaot f);*
//Evaluate the polynomial using f and return result

*}*

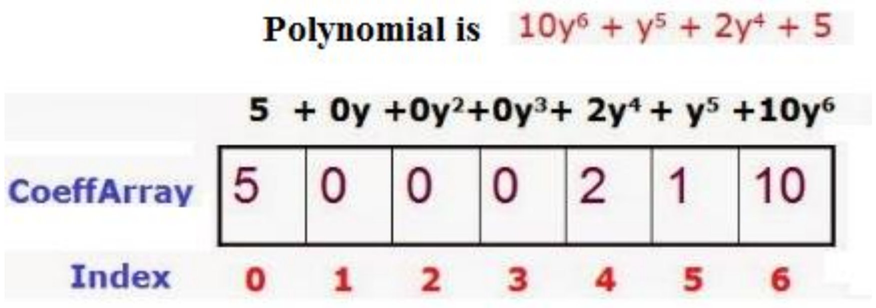
**Polynomial Representation:**

Polynomial may be represented using array (or) linked lists.

**Polynomial as Array Representation**

It is asresulted that exponent of a expression are arranged from 0 to highest value(degree) which is represented by subscripts(index) of respective exponents are placed at appropriate index in the array.

**Representation 1:**


Polynomial is $10y^6 + y^5 + 2y^4 + 5$
$5 + 0y + 0y^2 + 0y^3 + 2y^4 + y^5 + 10y^6$
CoeffArray
Index

| CoeffArray | 5 | 0 | 0 | 0 | 2 | 1 | 10 |
|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

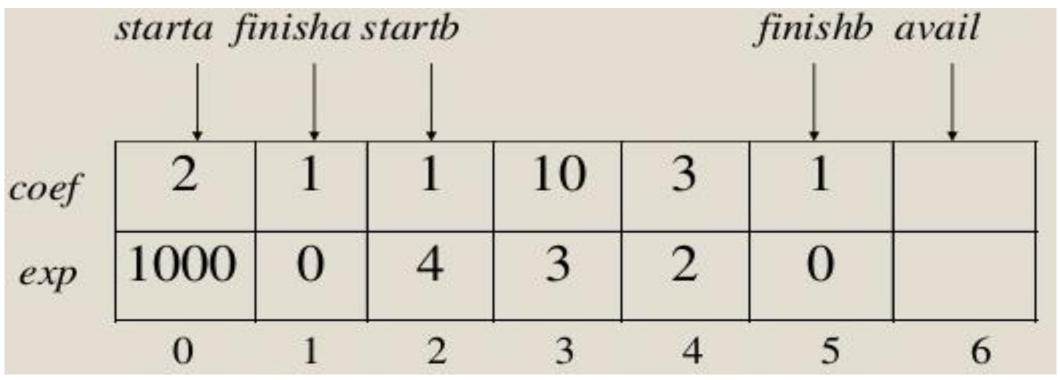
**Advantage:** Easy to Store and Represent

**Disadvantage:** Huge array size is required, waste of space

**Representation 2:**

$A(X)=2X^{1000}+1$

$B(X)=X^4+10X^3+3X^2+1$



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| coef | 2 | 1 | 1 | 10 | 3 | 1 | |
| exp | 1000 | 0 | 4 | 3 | 2 | 0 | |

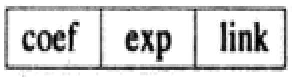
Code for representations:

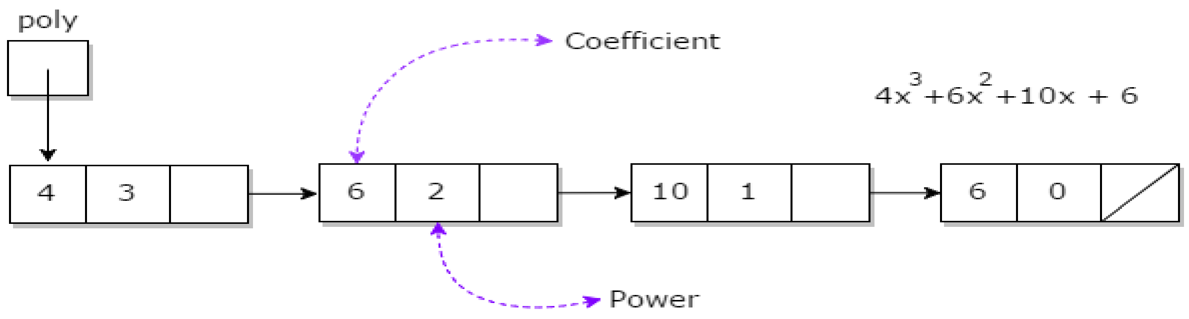
| | |
|---|---|
| class Polynomial<br>{<br>private:<br>    int degree;<br>    float coef[degree+1];<br>………<br>}; | class Term<br>{     **friend Polynomial;**<br>private:<br>    int exp;<br>    float coef;<br>};<br>class Polynomial<br>{ private:<br>    Term *termArray;<br>………<br>}; |

**Polynomial as Linked Representation:**

A Polynomial node has mainly two fields. Exponent and coefficient.

**Node of a Polynomial:**





In each node the exponent field will store the corresponding exponent and the coefficient field will store the corresponding coefficient. Link field points to the next item in the polynomial.

**Addition of two Polynomials:**

Let A and B be the two polynomials represented by the array/linked list.

1. while Both A and B are having terms (not null), repeat step 2.

2. If powers of the two terms are equal

        then insert the result of the terms into the result Polynomial

                Advance(move to next term) A

                Advance(move to next term) B

    Else if the power of the first polynomial A> power of second polynomial B

        Then insert the term from first polynomial A into result polynomial

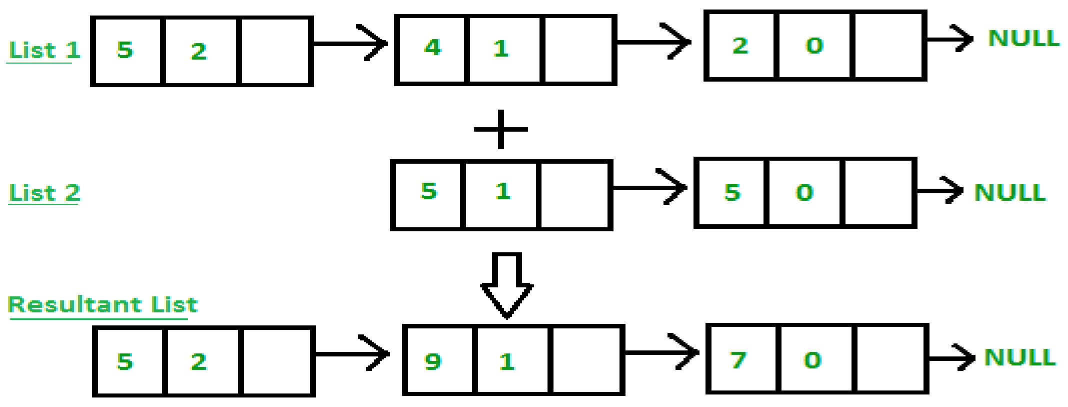
                Advance(move to next term) A

        Else insert the term from second polynomial B into result polynomial

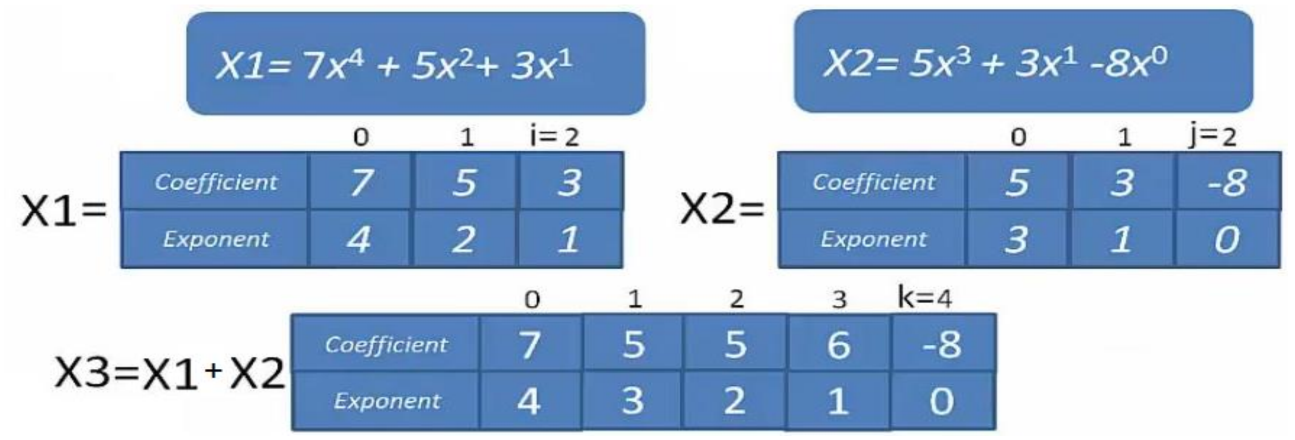
                Advance(move to next term) B

3. copy the remaining terms from the non empty polynomial into the result polynomial.

Ex: Addition of two polynomials using Linked List



Ex: Addition of two polynomials using arrays



$X1= 7x^4 + 5x^2 + 3x^1$

X1=

| | 0 | 1 | i=2 |
|---|---|---|---|
| Coefficient | 7 | 5 | 3 |
| Exponent | 4 | 2 | 1 |

$X2= 5x^3 + 3x^1 - 8x^0$

X2=

| | 0 | 1 | j=2 |
|---|---|---|---|
| Coefficient | 5 | 3 | -8 |
| Exponent | 3 | 1 | 0 |

X3=X1+X2

| | 0 | 1 | 2 | 3 | k=4 |
|---|---|---|---|---|---|
| Coefficient | 7 | 5 | 5 | 6 | -8 |
| Exponent | 4 | 3 | 2 | 1 | 0 |

## Sparse Matrix:-

- A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If majority of the elements of the matrix have **0 value**, then it is called a **sparse matrix**.

**Why to use Sparse Matrix?**

- **Storage:** There are lesser non-zero elements than zeros, lesser memory can be used to store only those non zero elements.
- **Computing time:** Computing time can be saved for traversing only non-zero elements..

➢ Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use. So, instead of storing zeroes, we store only non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

**Representation:** Sparse Matrix Representations can be done in following ways

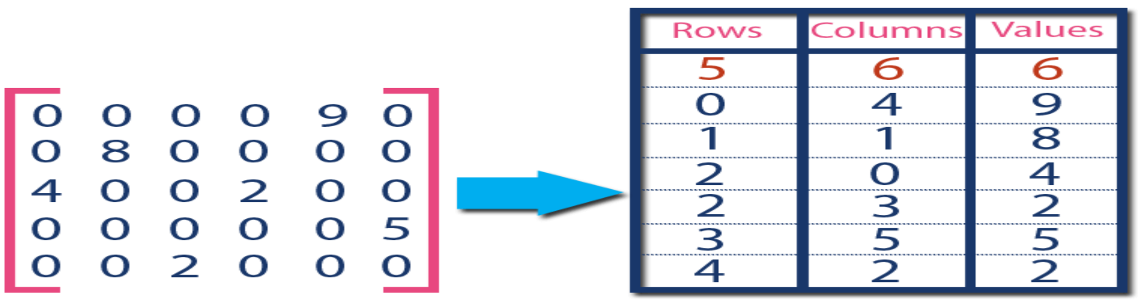
1. Array representation(Triplet Representation)
2. Linked list representation

**Method 1: Using Arrays**

2D array is used to represent a sparse matrix in which there are three columns named as

- **Row**: Index of row, where non-zero element is located
- **Column**: Index of column, where non-zero element is located
- **Value**: Value of the non zero element located at index – (row,column)

In this representation, the 0throw stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix} \rightarrow$$

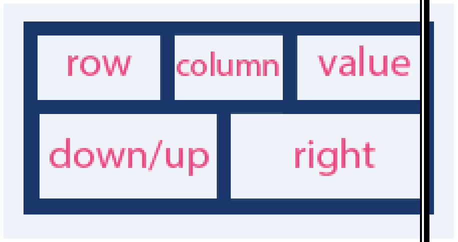
| Rows | Columns | Values |
|---|---|---|
| 5 | 6 | 6 |
| 0 | 4 | 9 |
| 1 | 1 | 8 |
| 2 | 0 | 4 |
| 2 | 3 | 2 |
| 3 | 5 | 5 |
| 4 | 2 | 2 |

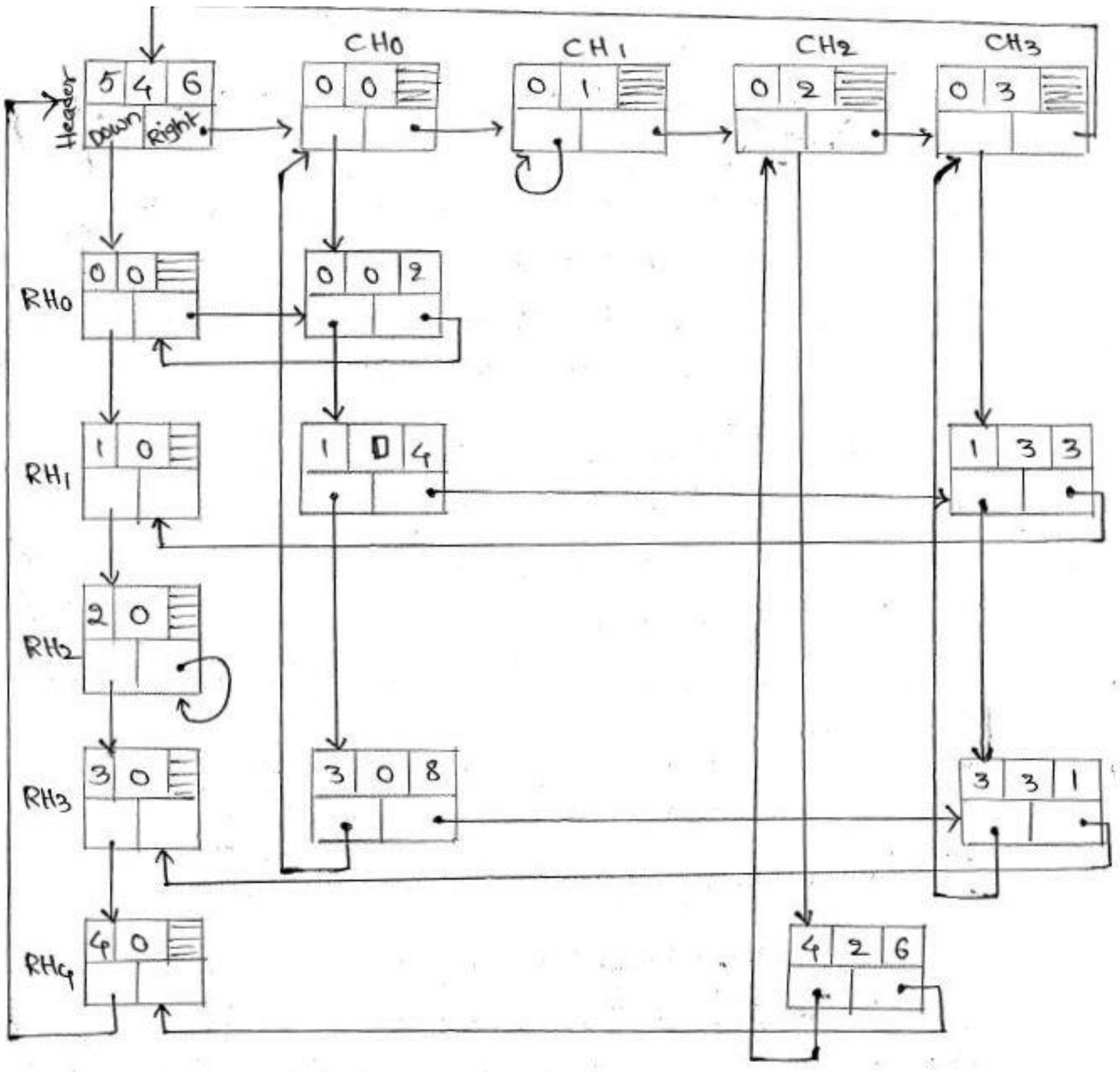
| | |
|---|---|
| class SparseTerm<br>{ **friend SparseMatrix;**<br>private:<br>int row,col,value;<br>}; | class SparseMatrix<br>{<br>private:<br>SparseTerm *termArray;<br>………<br>}; |

**Method 2: Using Linked Lists**

In linked list, each node has five fields. These fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index
- **Down:** Address of next non-zero in the same Column
- **Right:** Address of next non-zero in the same row





**ADT SparseMatrix**

{

**Instances:**

A set of triples, <row, col, value>

//where row and col are integers and form a unique combination.

Operations:

**void create(n);**//creates a SparseMatrix that can hold **n** non-zero elements information.

**SparseMatrix transpose(SparseMatrix A);** //It return the matrix produced by interchanging the row and column value of every triple.

**SparseMatrix add(SparseMatrix A, SparseMatrix B);** //if dimensions of A and B are the same return the matrix else return error.

**SparseMatrix multiply(SparseMatrix A, SparseMatrix B); //** if number of columns in A equals number of rows in B return the matrix C produced by multiplying A by B according to the formula: C[i][j]= Σ(A[i][k]*B[k][j]) where C[i][j] is the (i,j) th element else return error.

}

**Sparse Matrix Transpose:**

To transpose a matrix we must interchange the rows and columns.

**Example**: Sparse matrix A, Transpose Sparse Matrix B

| | row | col | value |
|---|---|---|---|
| A.t[0] | 6 | 6 | 8 |
| [1] | 0 | 0 | 15 |
| [2] | 0 | 3 | 22 |
| [3] | 0 | 5 | -15 |
| [4] | 1 | 1 | 11 |
| [5] | 1 | 2 | 3 |
| [6] | 2 | 3 | -6 |
| [7] | 4 | 0 | 91 |
| [8] | 5 | 2 | 28 |

| | row | col | value |
|---|---|---|---|
| B.t[0] | 6 | 6 | 8 |
| [1] | 0 | 0 | 15 |
| [2] | 0 | 4 | 91 |
| [3] | 1 | 1 | 11 |
| [4] | 2 | 1 | 3 |
| [5] | 2 | 5 | 28 |
| [6] | 3 | 0 | 22 |
| [7] | 3 | 2 | -6 |
| [8] | 5 | 0 | -15 |

We move these triples consecutively in to the transpose matrix, as we insert new triples, we must move elements to maintain the correct order. We should “find all the elements in column 0 and store them in row 0 of the transpose matrix, find all the elements in column 1 and store them in row 1 etc.

**Example:Implement a program for finding Sparse matrix and transpose of sparse matrix**

```
#include<iostream>
class matrix
{
int a[20][20],b[20][20],c[20][20],i,j,x,y,m,n;
  public:
        void input(int,int);
        void sparse();
        void transpose();
};
void matrix::input(int x,int y)
{
        m=x; n=y;
        cout<<"enter the matrix: \n";
        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        cin>>a[i][j];
        cout<<"the matrix is: \n";
        for(i=0;i<m;i++)
        {
        cout<<"\n";
        for(j=0;j<n;j++)
                cout<<a[i][j];
        }
}
void matrix::sparse()
{
        int k=1;
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        if(a[i][j]!=0)
                        {
                        b[k][0]=i;
                        b[k][1]=j;
                        b[k][2]=a[i][j];
                        k++;
                        }
                }
        }
        b[0][0]=m;
        b[0][1]=n;
        b[0][2]=k-1;
        cout<<"the sparse matrix ";
        for(i=0;i<k;i++)
        {
                cout<<"\n";
                for(j=0;j<3;j++)
                        cout<<b[i][j];
        }
}
void matrix::transpose()
        {
        int i,j,k,n;
        c[0][0]=b[0][1];
        c[0][1]=b[0][0];
        c[0][2]=b[0][2];
        k=1;
        for(i=0;i<b[0][1];i++)
                for(j=1;j<=b[0][2];j++)
                        if(i==b[j][1])
                        {
                                c[k][0]=b[j][1];
                                c[k][1]=b[j][0];
                                c[k][2]=b[j][2];
                                k++;
                        }
        cout<<" Transpose of sparse matrix:”;
        for(i=0;i<k;i++)
        {
                cout<<"\n";
                for(j=0;j<3;j++)
                        cout<<c[i][j];
        }
        }
int main()
{
int m,n;
matrix ob;
cout<<"enter order of matrix: \n";
cin>>m>>n;
ob.input(m,n);
ob.sparse();
ob.transpose();
return 0;
}
```

**Sparse Matrix Multiplication**:

Given *A* and *B* where *A* is *m*□*n* and *B* is *n*□*p*, the product matrix *D* has dimension *m*□*p*. Its <*i*, *j*> element is

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

for $0 \le i < m$ and $0 \le j < p$.

In general, you multiply a values at row 'i' in matrix A with a column in the matrix B and store the sum of the row operation as a result in the resultant matrix.

However, since this problem involves sparse matrices, we can ignore the multiplication with the column in matrix B if the value in matrix A is 0(zero). This small optimization helps us in avoiding K operations where K is the number of rows in the matrix B or the number of columns in the matrix A.

```
i←A.rows; j←B.cols; k←A.cols/B.rows
        sum += A[i][k]*B[k][j];
    C[i][j] = sum;
```

$$\begin{bmatrix}1 & 0 & 0\\1 & 0 & 0\\1 & 0 & 0\end{bmatrix}\begin{bmatrix}1 & 1 & 1\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix} = \begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$$

**Figure 2.5:** Multiplication of two sparse matrices

Matrix A

| Row | Column | Value |
|---|---|---|
| 1 | 2 | 10 |
| 1 | 4 | 12 |
| 3 | 3 | 5 |
| 4 | 1 | 15 |
| 4 | 2 | 12 |

Matrix B

| Row | Column | Value |
|---|---|---|
| 1 | 3 | 8 |
| 2 | 4 | 23 |
| 3 | 3 | 9 |
| 4 | 1 | 20 |
| 4 | 2 | 25 |

Matrix C=A*B

| Row | Column | Value |
|---|---|---|
| 1 | 1 | 240 |
| 1 | 2 | 300 |
| 1 | 4 | 230 |
| 3 | 3 | 45 |
| 4 | 3 | 120 |
| 4 | 4 | 276 |

```
C[1][4] = A[1][2]*B[2][4] =   10*23 =   230
C[1][1]= A[1][4]*B[4][1] =    12*20 =   240
C[1][2]= A[1][4]*B[4][2] =    12*25 =   300
C[3][3]= A[3][3]*B[3][3] =    5*9   =    45
C[4][3]= A[4][1]*B[1][3] =    15*8  =   120
C[4][4]= A[4][2]*B[2][4] =    12*23 =   276
```

**SYLLABUS:**

# UNIT-II

**Templates in C++:** Template Functions, Template Class, Using Templates to Represent Container Classes, **Sub typing and Inheritance in C++**, **The Stack Abstract Data Type,** Evaluation of Expressions (Postfix Notation), convert Infix to Postfix, **The Queue Abstract Data Type,** Circular Queues

## C++ Templates:

- Templates are powerful features of C++ which allows you to write generic programs.
- Using templates you can create a single function or a class to work with different data types
- Templates are often used for the purpose of code reusability and readability of the programs.

The concept of templates can be used in two different ways:

1. Function Templates
2. Class Templates

### 1. Function Templates:

- A function template works in a similar to a normal function, with one key difference.
- A single template function can work with different data types at once but, a single normal function can only work with one set of data types.
- Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

**Declare a function template:**

- A function template starts with the keyword **template** followed by template parameter/s inside < >(angular brackets)

**Syntax:**

```
template <class T>
T function_name(T arg)
{
   ... .. ...
}
```

- In the above code, T is a template argument that accepts different data types (int, float), and class is a keyword.

- When, an argument of a data type is passed to function_name( ), compiler generates a new version of function_name( ) for the given data type.

## 2. Class Templates:

- Like function templates, you can also create class templates for generic class operations.
- Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.
- Class templates make it easy to reuse the same code for all data types.

**Declare a class template:**

**Syntax:**

```
template <class T>
class className
{
   ... .. ...
public:
   T data;
   T function_name(T arg);
   ... .. ...
};
```

**Create a class template object:**

- You need to define the data type inside a < > when creation of class template object.

**Syntax:** className<dataType> classObject;

For example:

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

**Example** : Simple calculator using Class template, Program to add, subtract, multiply and divide two numbers using class template

```
#include <iostream.h>
template <class T>
class Calculator
{
private:
    T num1, num2;
public:
Calculator(T n1, T n2)
{
num1 = n1;
num2 = n2;
 }
void display ()
{
cout << "Numbers are: " << num1 << "
and " << num2 << "." << endl;
cout << "Addition is: " << add();
cout << "Subtraction is:" << subtract();
cout << "Product is: " << multiply();
cout << "Division is: " << divide();
}
T add() { return num1 + num2; }
T subtract() { return num1 - num2; }
T multiply() { return num1 * num2; }
T divide() { return num1 / num2; }
};
```

```
void main()
{
Calculator<int> intCalc(2, 1);
Calculator<float> floatCalc(2.4, 1.2);

cout << "Int results:" << endl;
intCalc.displayResult();

cout << endl << "Float results:" <<
endl;
floatCalc.displayResult();
}
```

**Output**

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

## Container Classes:-

- A **container class** is a data type that is capable of holding a collection of items.
- In C++, container classes can be implemented as a class, along with member functions to add, remove, and examine items
- If it is a container class no need to write implementation for operations, directly perform the operations using predefined functions by include library file **(.h file)**
- There are several ways to allocate the memory for container classes like Arrays, Linked List, Vector ...etc

Ex: String is a container class, so by including string.h file directly perform the operations like strlen( ),strcmp( ),strrev( ), .. etc

### Templates to Represent Container Classes:

```
template <class T>
class Bag
{
        T *a;
        int count;
        static int Capacity;
public:
        Bag(int Capacity=20);
        void insert(T item);
        T remove();
        int bagsize();
};
```

**Container Class Program:** Bag operations (**Bag.h** )

```
class Bag
{
        int *a;
        int count;
        static int Capacity;
public:
        Bag(int Capacity=20);
        void insert(int item);
        int remove();
        int bagsize();
};

Bag::Bag(int Capacity=20)
{
        a= new int[Capacity];
        count=0;
}
int Bag::bagsize()
{
        return (count);
}
```

```
void Bag::insert(int item)
{
        if(bagsize()<Capacity)
        {
                data[count]=item;
                count++;
        }
}
int Bag::remove()
{
        if(bagsize()>0)
        {
                int item=a[count];
                count--;
        }
        return item;
}
void display()
{
        for(int i=0;i<bagsize();i++)
        {
                cout<<a[i];
        }
}
```

## Sub typing and Inheritance in C++:-

- Inheritance allows the child class to acquire the properties (the data members) and functionality (the member functions) of parent class. OR IS-A relationship is also called Inheritance.
- Child class: A class that inherits another class is known as child class, it is also known as derived class or subclass.
- Parent class: The class that is being inherited by other class is known as parent class, super class or base class.
- Implementation of public inheritance is called **Sub Typing**

**Advantages:** The main advantages of inheritance are code reusability and readability.

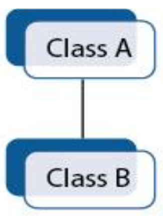
| Access | Public | Protected | private |
|---|---|---|---|
| Same class | Yes | Yes | Yes |
| Derived classes | Yes | Yes | No |
| Outside classes | Yes | No | No |

**Syntax of Inheritance:**

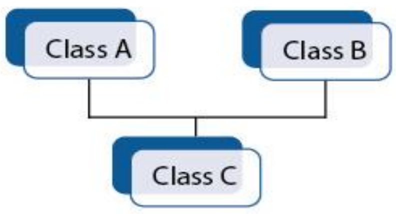
```
class parent_class
{
   //Body of parent class
};
class child_class : access_modifier parent_class
{
  //Body of child class
};
```

**Types of Inheritance in C++:**

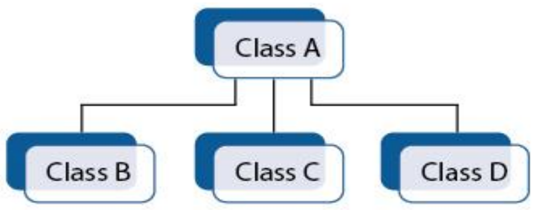
1) Single inheritance
2) Multilevel inheritance
3) Multiple inheritances
4) Hierarchical inheritance
5) Hybrid inheritance



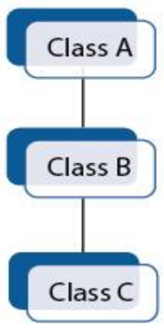
Single Inheritance



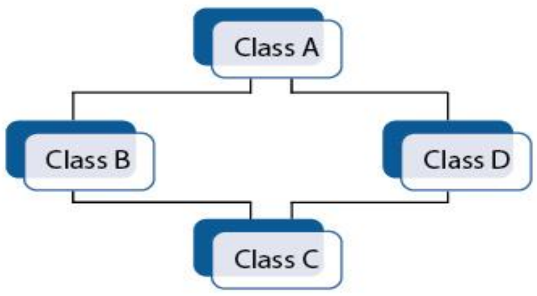
Multiple Inheritance



Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance

## 1. Single inheritance

- In Single inheritance one class inherits one class exactly.
- For example: Lets say we have class A and B

  **→B inherits A**

## 2. Multiple Inheritance

- In multiple inheritances, a class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.

For example: Lets say we have class A, class B and class C

**→C inherits A and B both**

3. **Multilevel Inheritance**

- In this type of inheritance one class inherits another child class.
- For example: Lets say we have class A, class B and class C

  **→C inherits B and B inherits A**

4. **Hierarchical Inheritance**

- In this type of inheritance, one parent class has more than one child class.
- For example: Lets say we have class A, class B, class C and class D

  **→Class B, C and D inherits class A**

5. **Hybrid Inheritance**

- Hybrid inheritance is a combination of more than one type of inheritance, that follows multiple and hierarchical inheritance both can be called hybrid inheritance.
- For example, Lets say we have class A, class B, class C and class D

  **→B and C are Childs of parent class A, and class D is child of both B and C**

**Example of Single inheritance:**

```
#include <iostream>
class A
{
public:
  A( ) {    cout<<"A class; }
};
class B: public A
{
public:
  B( ) {    cout<<" B class"; }
};
```

```
int main()
{
  //Creating object of class B
  B obj;
  return 0;
}
Output:

A class
B class
```

**Example of Multiple Inheritances:**

```
#include <iostream>
class A
{
public:
  A( ) {    cout<<"A class; }
};
class B
{
public:
  B( ) {    cout<<" B class"; }
};
class C: public A, public B
{
public:
  C( ) {    cout<<" C class"; }
};

int main()
{
  //Creating object of class C
  C obj;
  return 0;
}
Output:
A class
B class
C class
```

**Example of Multilevel inheritance:**

```
class A
{
public:
  A( ) {    cout<<"A class; }
};
class B: public A
{
public:
  B( ) {    cout<<" B class"; }
};
class C: public B
{
public:
  C( ) {    cout<<" C class"; }
};
```

```
int main( )
{
  //Creating object of class C
  C obj;
  return 0;
}
```

Output:

```
A class
B class
C class
```

**Example of Hierarchical inheritance:**

```
class A
{
public:
  A( ) {    cout<<"A class; }
};
class B: public A
{
public:
  B( ) {    cout<<" B class"; }
};
class C: public A
{
public:
  C( ) {    cout<<" C class"; }
};
class D: public A
{
public:
  D( ) {    cout<<" D class"; }
};
```

```
int main() {
   //Creating object of class B,C,D
   B obj1;
   C obj2;
   D obj3;
   return 0;
}
```

Output:

```
A class
B class
A class
C class
A class
D class
```

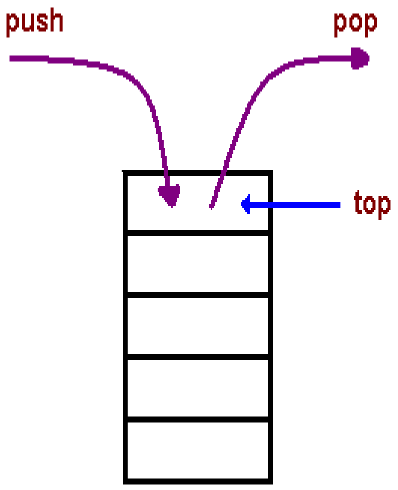
**Example of Hybrid inheritance:**

```
class A
{
public:
  A( ) {    cout<<"A class; }
};
class B: public A
{
public:
  B( ) {    cout<<" B class"; }
};
class C: public A
{
public:
  C( ) {    cout<<" C class"; }
};
class D: public B, public C
{
public:
  D( ) {    cout<<" D class"; }
};
int main()
{
   //Creating object of class D
   D obj;
   return 0;
}
```

Output:

```
A class
B class
A class
C class
D class
```

## Stack Abstract Data Type:-



- Stack is a linear data structure in which insertion and deletion can perform at the same end called **top** of stack.
- When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop.
- Stack is also called as Last-In-First-Out (LIFO) list which means that the last element that is inserted will be the first element to be removed from the stack.
- When a stack is completely full, it is said to be Stack is **Overflow** and if stack is completely empty, it is said to be Stack is **Underflow**.

**The basic operations performed in a Stack:**

1. Push( )

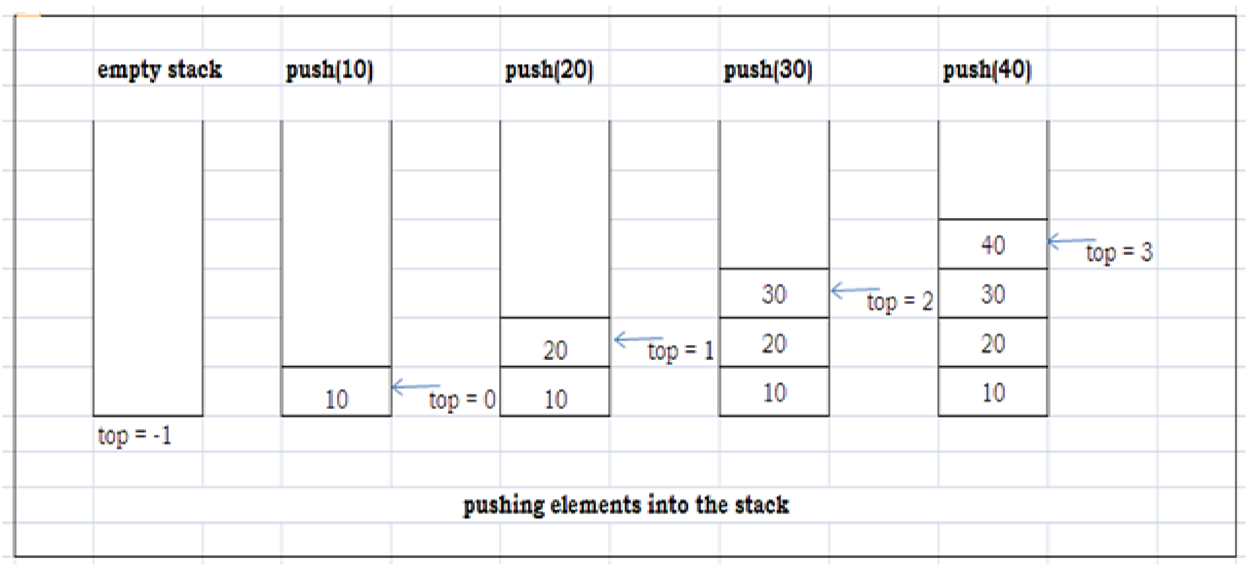
2. Pop( )

3.peek()

```
ADT Stack
{
        instances:
                Elements are stored in array called STACK, insertion and deletions can
                perform at TOP end.
        operations:
                Stack( int size)  -   initialize STACK with size and TOP= -1
                IsEmpty()         –   returns true if stack is empty otherwise false
                IsFull()          –   returns true if stack is full otherwise false
                push(x)           –   add element x at the top of the stack
                pop()             –   remove top element from the stack
                peek()            –   get top element of the stack without removing it
}
```

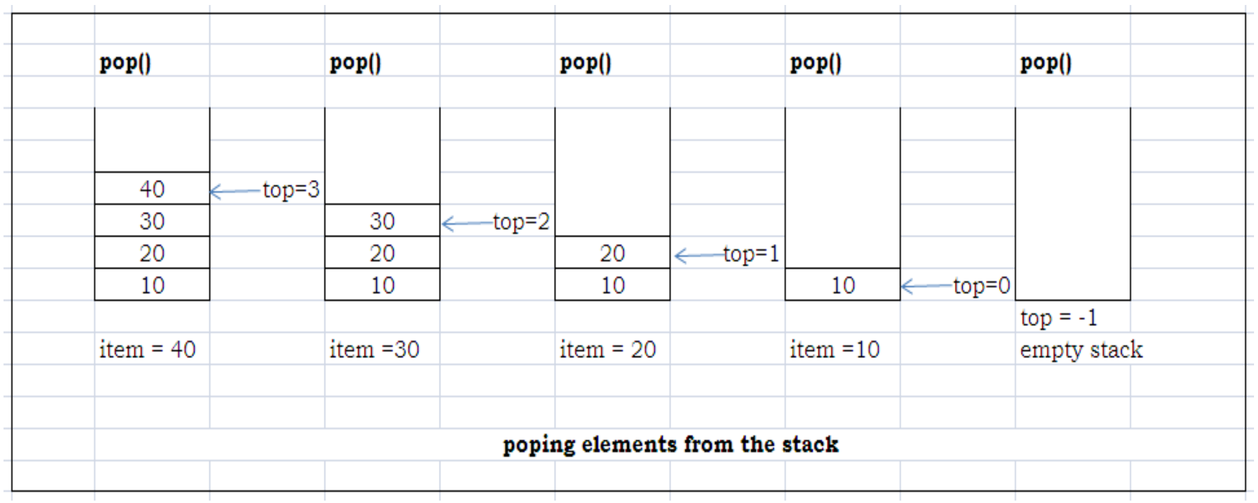
**Algorithm for PUSH operation**

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element at top location.



**Algorithm for POP operation**

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.



**Example: Implementation of Stack ADT using Array's**

```
#include<iostream.h>

class stack
{
        int *s;
        int top;
        int size;
public:
        stack(int n)
        {
                s=new int[n];
                top=-1;
                size=n;
        }
bool IsEmpty() {return (top==-1); }
bool IsFull()  {return (top >= (size-1)); }
        void push(int);
        void pop();
        void display();
};

void stack::push(int item)
{
if(IsFull())
        cout<<"Stack is full(Overflow)";
else
        {
                top=top+1;
                s[top]=item;
        }
}
```

```
void stack::pop()
{
        int item=0;
if(IsEmpty())
        cout<<"stack  is
empty(Underflow)";
else
        {
                item = s[top];
                top=top-1;
                cout<<item;
        }
}

void stack::display( )
{
        if(top== -1)
                cout<<"stack empty";
        else
        {
                for(int i=top; i>=0; i--)
                        cout<<stk[i]<<" ";
        }
}

void main()
{
        stack sa(5);

        sa.push(10);
        sa.push(20);
        sa.push(30);
        sa.pop();
        sa.pop();
        sa.push(40);
        sa.display();
}
```

**Applications of Stacks**

- ✓ Stack is used to reversing the given string.
- ✓ Stack is used to evaluate a postfix expression.
- ✓ Stack is used to convert an infix expression into postfix/prefix form.
- ✓ Stack is used to matching the parentheses in an expression.
- ✓ In recursion, all intermediate arguments and return values are stored on the processor's stack.

## Evaluation of Expressions:-

- An expression is defined as the combination of operands (variables, constants) and operators arranged as per the syntax of the language.
- An expression can be represented using three different notations. They are infix, postfix and prefix notations:

→**Prefix:** An arithmetic expression in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as **polish notation**.

Example: + A B

→**Infix:** An arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: A + B

→**Postfix:** An arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as suffix notation OR **reverse polish notation**.

Example: A B +

**Operator Precedence**: When an expression consist different level of operators we follow it.

We consider five binary operations: +, -, *, / and ^ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest): **^ , * , /, +, -**

**Operator Associativity**: When an expression consist more than same level precedence operators we follow it.

Basically we have Left to Right associativity and Right to Left Associativity. Most of the operators are follows Left to Right but some of the operators are follow Right to left Associativity like Unary(+/-), ++/-- ,Logical negation (!), Pointer and address (*,&), Conditional Operators and Assignment operators(=,+=,-=,*=,/=,%=).

**x = a / b – c + d * e – a * c**

For example, a = 4, b = c = 2, d = e = 3 then the value of x is found as

((4 / 2) – 2) + (3 * 3) – (4 * 2)

=0 + 9 – 8

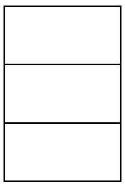
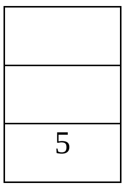
=1

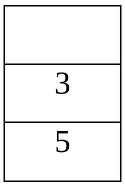
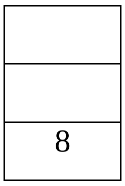
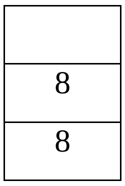
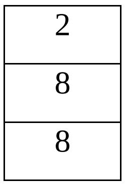
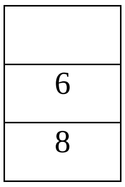
## EVALUATION OF POSTFIX EXPRESSION:

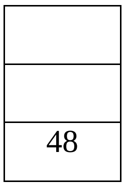
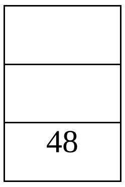
The standard representation for writing expressions is infix notation. But the compiler uses the postfix notation for evaluating the expression rather than the infix notation. It is an easy task for evaluating the postfix expression than infix expression because there are no parentheses. To evaluate an expression we scan it from left to right. The postfix expression is evaluated easily by the use of a stack.

To evaluate a postfix expression use the following steps...

1. Read the **poststring** from left to right
2. Initialize an **empty Stack**
3. Repeat until end of the **poststring**
   - **i.** If the scanned character is operand, then push it on to the Stack.
   - **ii.** If the scanned character is operator (+ , - , * , / etc.,), then pop top two elements from the stack, perform the operation with the operator then push result back on to the Stack.
4. Finally! We have one element in the stack, perform a pop operation and display the popped value as **final result**.

| **Postfix Expression is 5 3 + 8 2 - *** | | |
|---|---|---|
| **Symbol** | **Stack** | **Evaluation** |
| Initially |  Stack is empty | |
| 5 |  Push(5) | |

| | | |
|---|---|---|
| 3 |  <br>Push(3) | |
| + |  <br>Value1=pop()<br>Value2=pop()<br>Result=Value2+Value1<br>Push(Result) | Value1=3<br>Value2=5<br>Result=5+3=8<br>Push(8) |
| 8 |  <br>Push(8) | |
| 2 |  <br>Push(2) | |
| - |  <br>Value1=pop()<br>Value2=Pop()<br>Result=Value2-Value1<br>Push(Result) | Value1=2<br>Value2=8<br>Result=8-2=6<br>Push(6) |

| | | |
|---|---|---|
| * |  Value1=pop()<br>Value2=Pop()<br>Result=Value2*Value1<br>Push(Result) | Value1=6<br>Value2=8<br>Result=8*6=48<br>Push(48) |
| End of Expression |  Result=pop() | Final Result is 48 |

## Conversion of INFIX to POSTFIX:

Procedure to convert from infix expression to postfix expression is as follows.

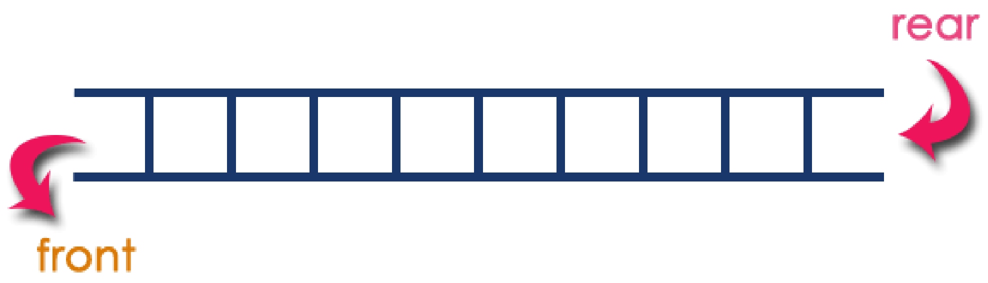
1. Initialize an empty stack
2. Push “(“onto Stack, and add “)” to the end of Infix string.
3. Scan the Infix string from left to right until end of the infix
    - **i.** If the scanned character is “(“, pushed into the stack.
    - **ii.** If the scanned character is “)”, pop the elements from the stack up to encountering the “(“, and add the popped elements to postfix string except parentheses.
    - **iii.** If the scanned character is an operand, add it to the Postfix string.
    - **iv.** If the scanned character is an Operator, compare the precedence of the character with the element on top of the stack. If top of Stack has lower precedence over the scanned character then push the operator into the stack else pop the element from the stack and add it to postfix string and push the scanned character to stack.

Example: a * (b + c) *d)

| Token | Stack | | | | Postfix String |
|---|---|---|---|---|---|
| | ( | | | | |
| a | ( | | | | a |
| * | ( | * | | | a |
| ( | ( | * | ( | | a |
| b | ( | * | ( | | ab |
| + | ( | * | ( | + | ab |
| c | ( | * | ( | + | abc |
| ) | ( | * | | | abc+ |
| * | ( | * | | | abc+* |
| d | ( | * | | | abc+*d |
| ) | | | | | abc+*d* |

## Queue Abstract Data Type:-

- Queue is a linear data structure in which elements can be inserted from one end called **rear** and deleted from other end called **front**
- The deletion or insertion of elements can take place only at the front or rear end called dequeue and enqueue. The first element that gets added into the queue is the first one to get removed from the queue. Hence the queue is referred to as First-In-First-Out list (FIFO).



```
ADT Queue
{
        Instances:
                Elements are stored in array called QUEUE, insertion can perform at REAR
                end and deletions can perform at FRONT end.
        Operations:
                Queue( int size)     -initialize QUEUE with size and FRONT=REAR= -1
                IsEmpty()    – returns true if Queue is empty, otherwise false
                IsFull()     – returns true if Queue is full, otherwise false
                enqueue(x)     –     add element x at rear end of the Queue
                dequeue()      –     remove element at front end of the Queue
}
```

**Operations performed on Queue:**

There are two possible operations performed on a queue. They are enqueue and dequeue.

- ✓ enqueue: Allows inserting an element at the rear of the queue.
- ✓ dequeue: Allows removing an element from the front of the queue.

Algorithm for ENQUEUE operation

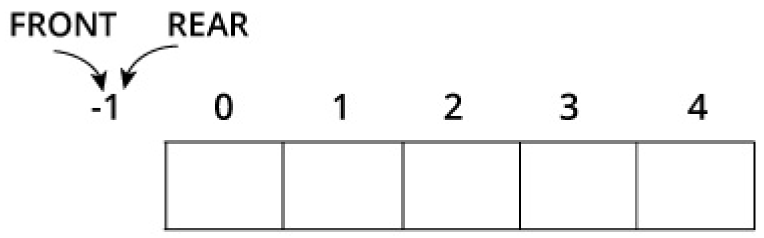
1. Check whether queue is FULL. **(rear >= SIZE-1)**
2. If it is **FULL**, then display an error message "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
3. If it is **NOT FULL**, then increment **rear** value by one (rear++) and set **queue[rear] = value.**

Algorithm for DEQUEUE operation

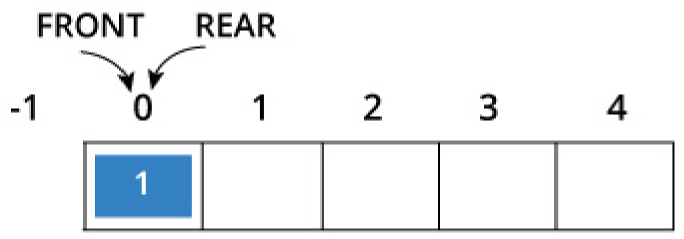
1. Check whether queue is EMPTY. **(front == -1)**
2. If it is **EMPTY,** then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
3. If it is **NOT EMPTY**, then display **queue[front]** as deleted element, increment the front value by one **(front ++).** If we are deleting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

Let us consider a queue, which can hold maximum of five elements.

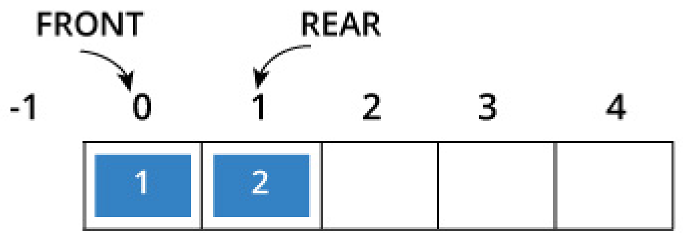
→Initially the queue is empty.



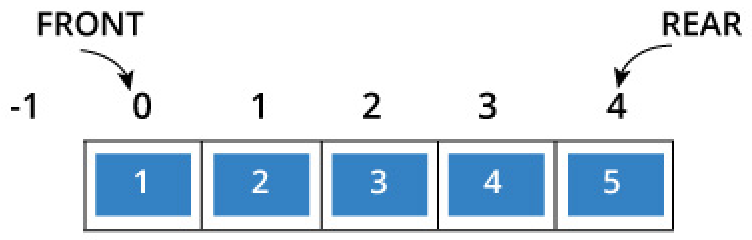
Empty Queue



EnQueue first element



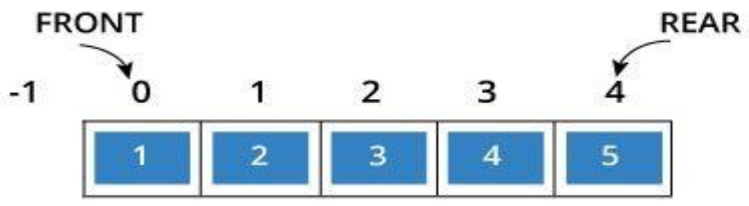
EnQueue

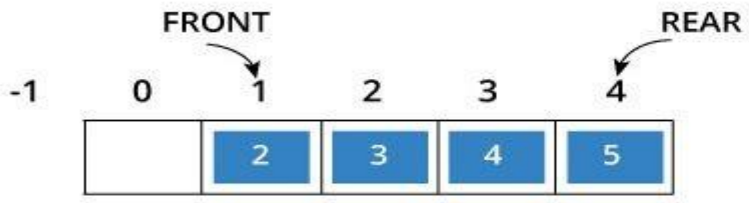


EnQueue

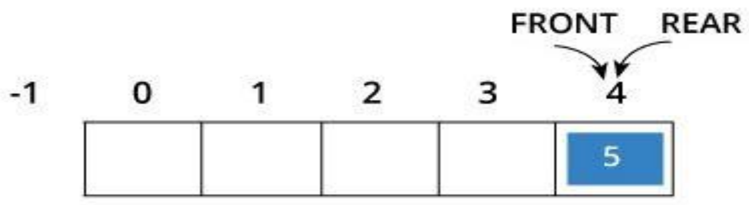
An element can be added to the queue only at the rear end of the queue. Before adding an element in the queue, it is checked whether queue is full. If the queue is full, then addition cannot take place. Otherwise, the element is added to the end of the list at the rear end. If we are inserting first element into the queue then change front to 0 (Zero)

Now, delete an element 1. The element deleted is the element at the front of the queue. So the status of the queue is:

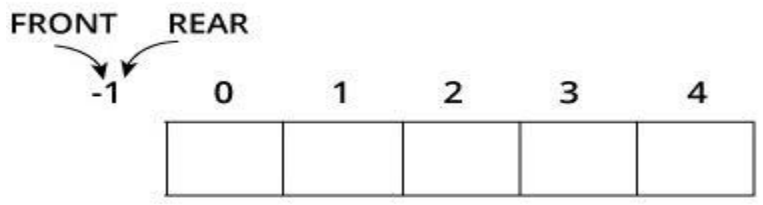




DeQueue



DeQueue last element



Empty Queue

When the last element delete 5. The element deleted at the front of the queue. So the status of the queue is empty. So change the values of front and rear to -1 **(front=rear= -1)**

The dequeue operation deletes the element from the front of the queue. Before deleting and element, it is checked if the queue is empty. If not the element pointed by front is deleted from the queue and front is now made to point to the next element in the queue.

**Applications of Queue:**

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

**Example: Implementation of Queue ADT using Array's**

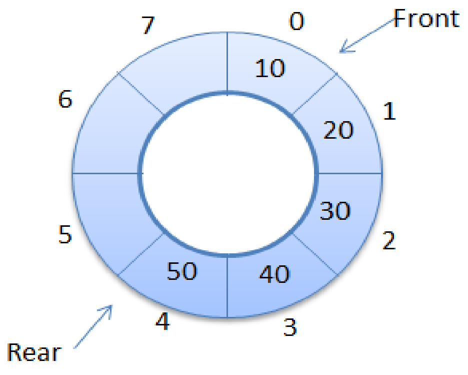
```
#include <iostream.h>
#include<stdlib.h>

class Queue
{
private:
   int *Q, front, rear, size;

public:
         Queue(int size);
         void enQueue(int item);
         void deQueue();
         void display();
};

Queue::Queue(int n)
         {
      Q = new int[n];
      front = -1;
      rear = -1;
      size = n;
   }

 void Queue :: enQueue(int item)
{
      if( rear >= size - 1)
         cout << "Queue is full";
      else
      {
         if(front == -1) front = 0;

         rear++;
         Q[rear] = item;
         cout << "Inserted " << Q[rear];
      }
   }
void Queue :: deQueue()
{
      if(front == -1)
         cout << "Queue is empty" << endl;
      else
      {
         cout << "Deleted: " <<Q[front];
         if(front >= rear)
         {
            front = -1;    rear = -1;
         }
         else
            front++;
      }
   }


   void Queue :: display()
   {
      if(front == -1)
         cout << "Empty Queue" << endl;
      else
      {
         cout << "\n Front -> " << front;
         cout << "Queue Elements are:  ";
         for(int i=front; i<=rear; i++)
            cout << Q[i] << " ";
         cout << "\n Rear -> " << rear ;
      }
   }

void main()
{
         Queue qu(5);

         qu.enQueue(10);
         qu.enQueue(20);
         qu.enQueue(30);
         qu.enQueue(40);

         qu.display();

         qu.deQueue();
         qu.deQueue();
         qu.deQueue();

         qu.display();

}
```

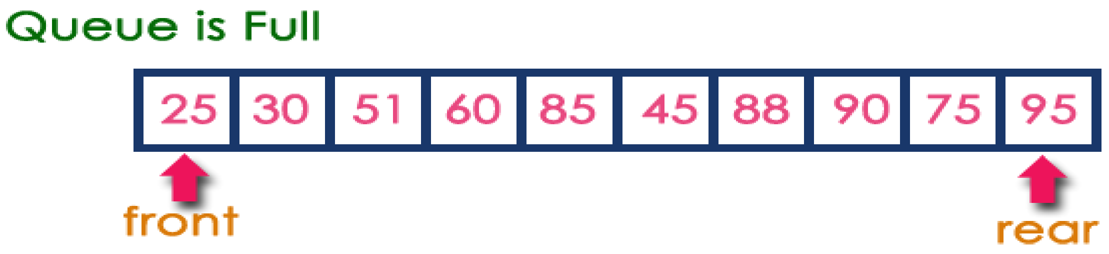
## Circular Queue:

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

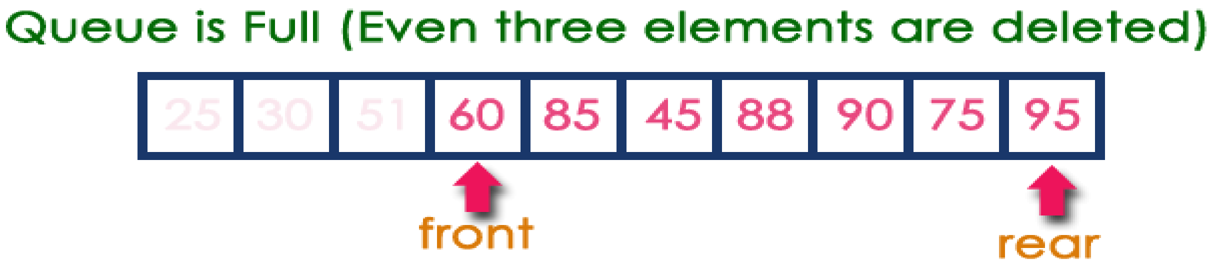
**Need of Circular Queue**

- In a normal Queue, we can insert elements until queue becomes full.
- If queue becomes full, we cannot insert the next element until all the elements are deleted.
- For example consider the queue below... After inserting all the elements into the queue.





Now consider the following situation after deleting three elements from the queue...



- This situation also says that Queue is full.
- We cannot insert the new element because; '**rear**' is still at last position, Even though we have empty positions in the queue we cannot make use.
- This is the major problem in normal queue.

**To overcome this problem we use circular queue.**

**Operations on Circular Queue:**

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at REAR position.

  **Steps:**

  1. Check whether queue is Full or not
     Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  2. If it is full then display Queue is full.
  3. If queue is not full then, check
     if (rear == SIZE – 1 && front != 0) if it is true then set **rear=0** and insert element.

- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from FRONT position.

  **Steps:**

  1. Check whether queue is Empty or not means check (front==-1).
  2. If it is empty then display Queue is empty.
  3. If queue is not empty, Check
     if (front==rear) if it is true then set front=rear= -1
     else check if (front==size-1), if it is true then set front=0 and return the element.

**Example: Implementation of Circular Queue ADT using Array's**

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

class Circular_Queue
{
   private:
         int *cq;
         int front, rear, size;
   public:
         Circular_Queue(int n)
         {
            cq = new int[n];
            rear = front = -1;
            size = n;
         }
```

```
void enQueue(int item)
    {
 if ((front == 0 && rear >= size-1) ||
 (front == rear+1))
       {
            cout<<"Queue Overflow \n";
            return;
       }
       else
      {
   if(front == -1)      front = 0;
   rear = (rear + 1) % size;
   cq[rear] = item;
   cout << "Inserted: " << cq[rear];
        }
   }
 void deQueue()
 {
    if (front == -1)
    {
 cout<<"Queue Underflow\n";
          return ;
    }
 cout<<"Deleted : "<< cq[front];
if(front == rear)
 {
    front = -1;
    rear = -1;
       }
else
  front = (front+1) % size;

 }

 void display()
 {
    int i;
    if (front == -1)
    {
          cout<<"Queue is empty\n";
          return;
    }
    else
    {
            cout << "\n Front -> " << front;
         cout << "\n Elements -> ";
 for(i = front; i != rear; i= (i+1) % size)
                    cout << cq[i] << " ";
            cout << cq[i] << " ";
            cout << "\n Rear -> " << rear;
               }
            }
 };

 void main()
 {
 Circular_Queue qu(4);

 qu.enQueue(10);
 qu.enQueue(20);
 qu.enQueue(30);
 qu.enQueue(40);
 qu.display();

 qu.deQueue();
 qu.deQueue();
 qu.deQueue();
 qu.display();

 qu.enQueue(50);
 qu.enQueue(60);
 qu.display();
 }
```

## Implementation of Queue using Stacks:

- For Queue we are having two ends, one is called FRONT end and other called REAR end
- For Stack we have only one end called TOP.
- So, for implementing the queue using stack we need TWO STACKS, one for insertion other for deletion.

**enQueue(int item):**

1. Push item to InputStack S1

**deQueue():**

1. IF (both InputStack S1 and OutputStack S2 are empty) THEN error.
2. IF (OutputStack is empty)

   Pop all elements from InputStack and push them into OutputStack (one element at a time).
3. Pop element from OutputStack and return.

- Initially we have two empty stacks S1, S2 and corresponding pointer variables TOP1, TOP2.
- First, we are reading the elements and pushed into the stack S1, for every push operation we need to increment the TOP1 value.

```
Algorithm enqueue(item)
{
        if(TOP1>=size-1)
                cout<<"Queue is Full(Overflow)";
        else
        {
                TOP1=TOP1+1;
                S1[TOP1]=item;
        }
}
```

- Now we want to delete an element according to queue operation, the first inserted element processed first.
- Here it is not possible if we use only one stack, so transfer all the elements from stacks S1 to S2.
- Now we can delete the top element in stack S2 using TOP2.
- After performing delete operation transfer all the elements from stack S2 to S1 stack.
- Now, if we want to insert an element, directly pushed into stack S1
- So, for very deletion operation we need to transfer the elements from one stack to another stack.

For deletion the following operation is needed.

```
Algorithm deQueue()
{
        while(TOP1 !=-1)
        {
                TOP2 = TOP2 +1;
                S2[TOP2] = S1[TOP1];
                TOP1 = TOP1 -1;
        }

        cout<<"Deleted element is: "<<S2[TOP2];
        TOP2 = TOP2 -1;


        while(TOP2 !=-1)
        {
                TOP1 = TOP1 +1;
                S1[TOP1] = S2[TOP2];
                TOP2 = TOP2 -1;
        }
}
```

## Priority Queue:

- Priority Queue is a data structure having the collection of elements, which is associated with priorities.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

  ADT Priorty_Queue

  {

  **Instances:**

  Finite collection of elements associated with some priority

  Both front and rear with in max size

  **Operations:**

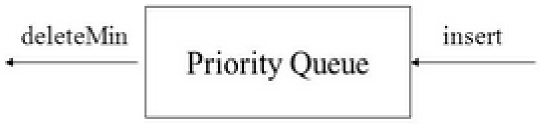
  Create(); IsEmpty(); IsFull(); enQueue(item);

  deQueue(); display();

  }

**Basic Operations in Priority Queue:**

It allows two operations



1. Insert (enQueue)
2. DeleteMin (deQueue)

- The smaller element in priority Queue having high priority, so find the smaller element and delete it first.

**Implement priority queue:**

**Array:** A simple implementation is to use array of following structure.

- insert() operation can be implemented by adding an item at end of array in O(1) time.
- DeleteMin () operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

**Linked List**, time complexity of all operations with linked list remains same as array. The advantage with linked list is DeleteMin() can be more efficient as we don't have to move items.

| **Implementation** | **Insertion** | **Deletion** |
|---|---|---|
| Unordered array | O(1) | O(n) |
| Ordered array | O(n) | O(1) |
| Linked list | O(n) | O(1) |
| Binary Heap | O(logn) | O(logn) |

# UNIT - III

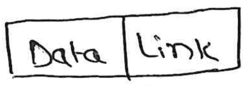
Syllabus :-

Introduction, Single Linked List, Circular linked list, Double linked list, stack using Linked list, Queue using Linked list, Polynomial using Linked list, Sparse matrix using linked list, Available space list, Equivalence classes, Generalized lists, Template class chain.

Introduction :-

- Arrays having some of the drawbacks like
  - (i) Before the compilation process need to give the maximum size of the array (int)
  - (ii) Arrays are stored in fixed distance a part (Ex:- 1000, 1002, ---)
  - (iii) Insertion and Deletion of elements in between require the lot of elements need to change the positions.
- Linked list will overcome all these problems. It uses Dynamic memory allocation (new) and Deallocation (delete).
- For insertion & Deletion of elements in between no need to change the positions, just changing the addresses.

* Linked List is a linear collection of data elements. these elements are called nodes. For each node we are having two fields

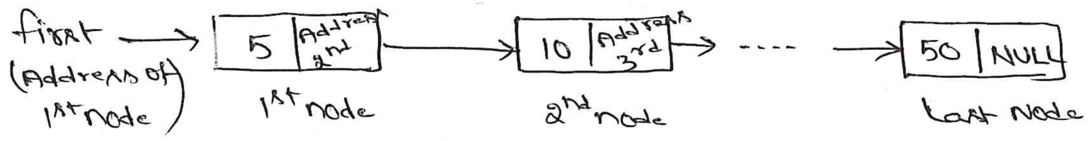


Node

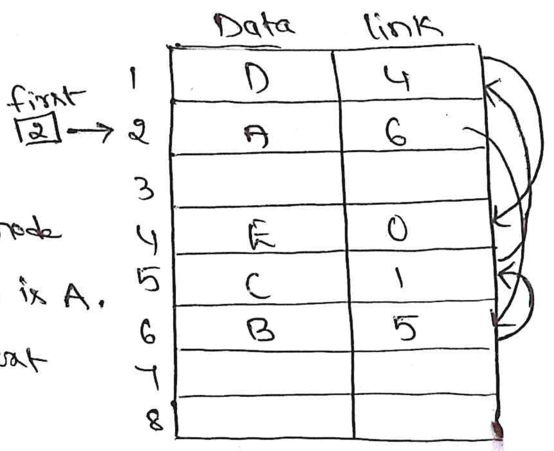
→ Data field used to store the element (information)

→ link field used to store the address of next node of same data type. so it is a pointer.

- The last node in the list is not having any next nodes. so, the link field will changes to NULL(0).
- first pointer, it stores the address of the starting node in the list.
- If first == NULL, then the list is empty list.



## Memory Representation :-



|  |  | Data | link |
|---|---|---|---|
| 1 |  | D | 4 |
| 2 | first 2 → | A | 6 |
| 3 |  |  |  |
| 4 |  | E | 0 |
| 5 |  | C | 1 |
| 6 |  | B | 5 |
| 7 |  |  |  |
| 8 |  |  |  |

- First is used to store address of Starting node.
- Here first = 2, So the starting node Stores at address of 2, element is A.
- link stores the address of next node, which is 6.
- we repeat this procedure until we reach a position, where the link field is zero (0) or NULL, then we denote last element in the list.

* Remember that the nodes in the linked list need not be consecutive memory locations.

## Memory Allocation and Deallocation :-

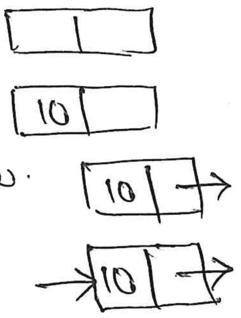
- If we want to add new node into already existing list in the memory, we first find unused memory (free space) then stores the information.
- Computer will maintain a list of all free memory cells (unused cells). That list is called freepool. The list is pointed by the variable called AVAIL. It stores address of first free space of memory.
- After taking memory cell from freepool for inserting element the AVAIL is pointing to next free memory cell.
- Collecting all remaining space into freepools, this process is called Garbage Collection.

* new operator :- It creates new dynamic object of specified type and returns a pointer that points to new object

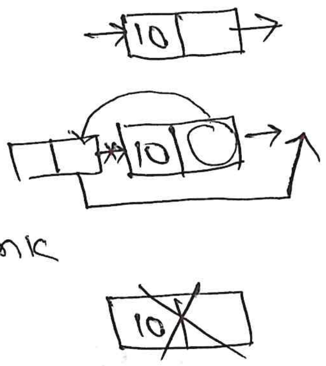
Ex:- node *ptr; ptr = new node;

delete operator:- To destroy the dynamically allocated variable / object and free the space occupied by object.
- It is explicitly deleting the memory locations
EX:- delete ptr;

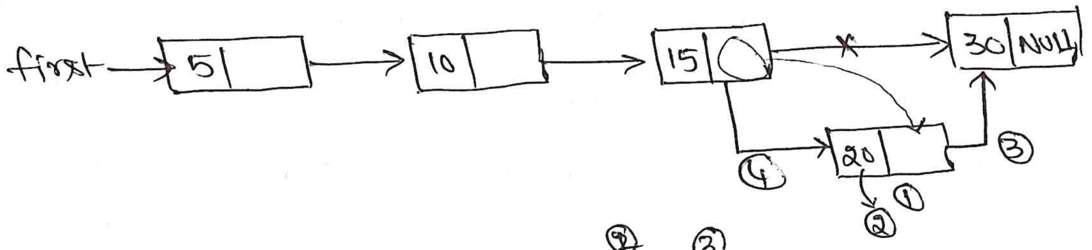
* Inserting the new node into the list is very easier
1. Get the new node
2. Set the data field
3. Set the link field to the next node.
4. Set the previous node link to the current node

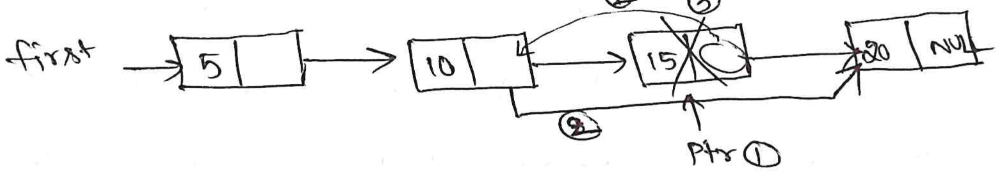


* Deleting the node from the list is very easier.
1. Identify, which node to be delete
2. Copy the current node link field to its previous node link field then automatically creating new link
3. Delete the current node.



EX:-



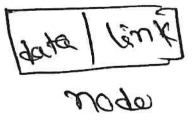


## Single linked List & Chains:-

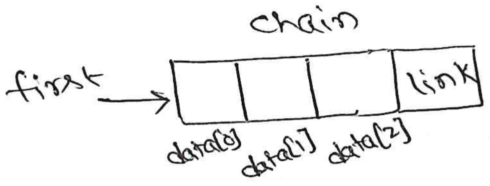
- Single linked list is a simple type of linked list, in which each node contains one data part and one link part.

Define Node:-

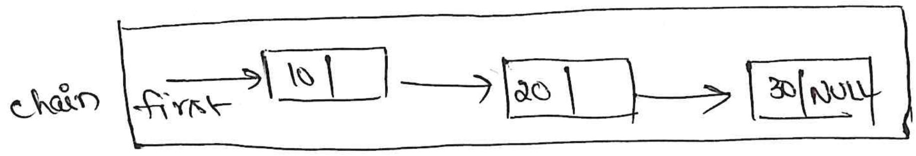
```
class node
{
    int data;
    node *link;
};
```



* Chain consists zero or more no. of nodes.



first → data[0]
first → data[1]
first → data[2]
first → link



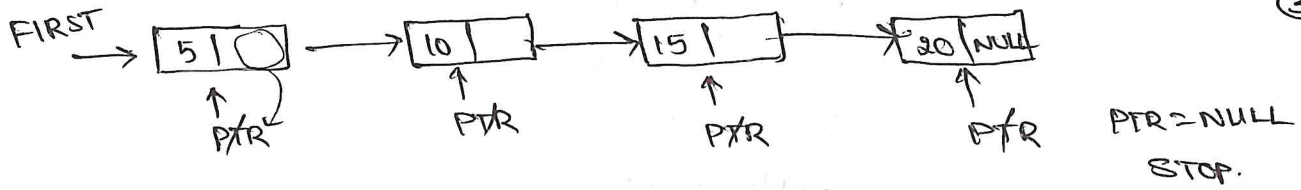
- Chains can be implemented in two ways
  (i) Friend class
  (ii) Nested classes

```
class chain;
class node
 {
     friend class chain;
     private:
         int data;
         node *link;
 };
 class chain
  {
     ---
  };
```

```
class Chain
{
   private:
     class node
       { int data;
         node *link;
       };
   public:
     ---
};
```

* Traversing a linked list :-
  - Traversing a linked list means accessing the nodes of the list inorder to perform some processing on them.
  - linked list contains the pointer FIRST, which stores the address of the starting node in the list
  - For the last node the LINK field address is NULL.
  - We are taking one pointer PTR for accessing the nodes.

```
Algorithm Traversal()
{
    node *PTR;
    Set PTR = FIRST;
    Repeat while PTR != NULL
        Apply process PTR → data;
        Set PTR = PTR → Link;
    End loop
}
```



- Counting number of nodes in a list

```
Algorithm Countnodes( )
{
    Set Count = 0;
    node * PTR = FIRST;
    Repeat while PTR != NULL
        Set count = Count + 1;
        Set PTR = PTR -> link;
    End loop;
    Print Count;
}
```

* Search an element in a list :-

- There are two outcomes for search, one is node address when search is successful otherwise NULL.

```
Algorithm Search (item)
{
    node *POS = NULL, *PTR = FIRST;
    Repeat while PTR != NULL
        if item == PTR->data then
            Set POS := PTR;
        else
            Set PTR = PTR -> link;
    end loop
    return (POS);
}
```

** AbstractDataType SLL

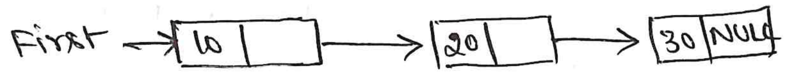
```
{ instances :
        Finite collection of data elements
  Operations :
        Create ( ); // create an empty list *FIRST = NULL
        insert_beg (item)
        insert_end (item)
        insert_pos ( pos, item);
        delete_beg();
        delete_end();
        delete_pos(pos);
        display();
    };
```

* <u>Insertion</u> at <u>beginning</u> of <u>the list</u> :-

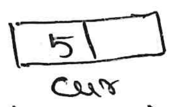
- For inserting new node into the list
  1. Create a node [cur]
  2. insert the value of a node into data field [cur→data] ← item
  3. the new node link field will store address of the starting node [cur → link] ← first
  4. Now the new node is the starting node of the list so change the FIRST to current node.

```
    Algorithm insert_beg(item)
    {
                                      // creating new node called
                                         current node with memory
           node *cur = new node;
insert field  Set  cur → data = item;
into cur      Set  cur → link = first;

           Set  first = cur;
    }
```

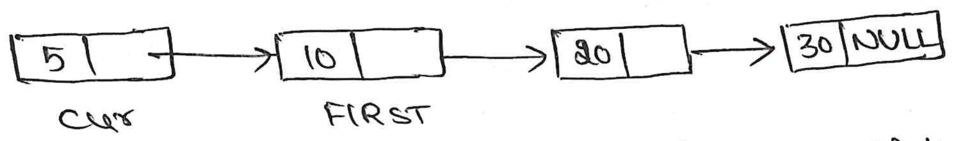
<u>Example</u>:- Add new node containg 5 into the list

List :


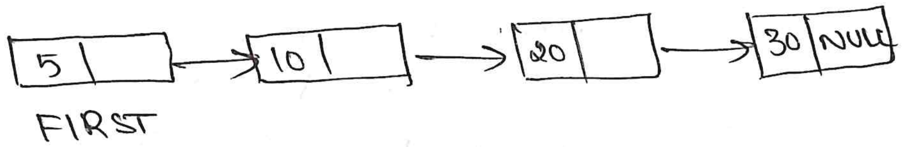
- Allocate the memory for current node and insert data of 5



- Add the newnode at beginning of the list by taking the <u>link</u> part of <u>cur</u> contains address of <u>first</u>.



- Now FIRST will changes to the starting node of the list

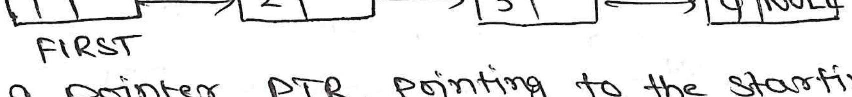


## * insertion at end of list :-

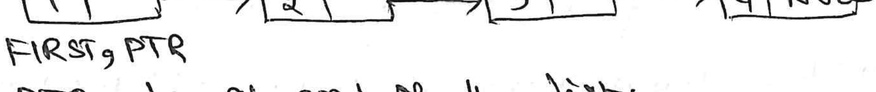
- Initially we are taking one pointer PTR for accessing the nodes in a list and it is initialized with FIRST.
- Now, create a new node of cur and insert the fields of data part with item and link part with NULL, because new node is inserting at end of the list.
- Now, moving the pointer PTR to the last node of the list.
- Add, newnode (cur) address to the link field of lastnode, then we are creating link between cur and previous last node.

```
Algorithm insert_end (item)
{
    node *PTR = FIRST;
    node * cur = new node; // create an empty node.
    Set cur → data = item; }  insert fields
    set cur → link = NULL; }
    Repeat while PTR → link != NULL
            Set PTR = PTR → Link;
    end loop
    Set PTR → Link = cur;
}
```

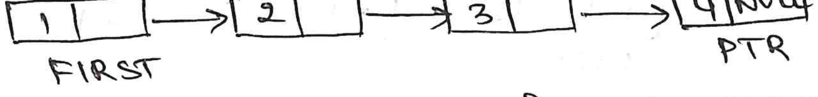
Example :- insert newnode of 5 at end of list.



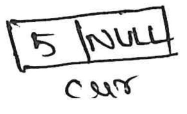
- Take a pointer PTR, pointing to the starting node of list



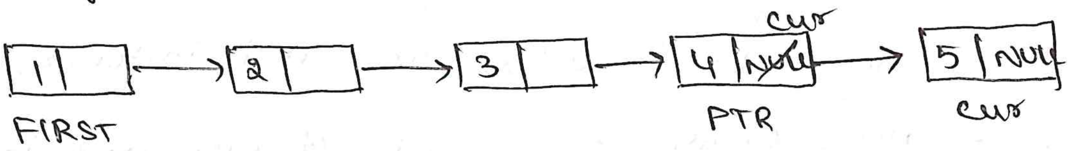
- move PTR to at end of the list.



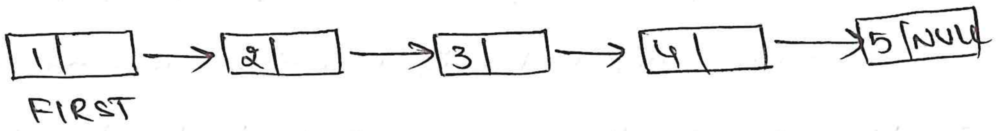
- Create new node and add the field to cur node



- Now add current node at end of the list by changing the link field of PTR by cur



- Now the list after adding the newnode is

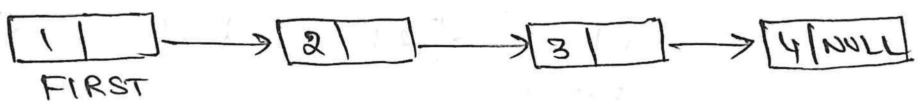


* Insertion of a node at particular Position:-

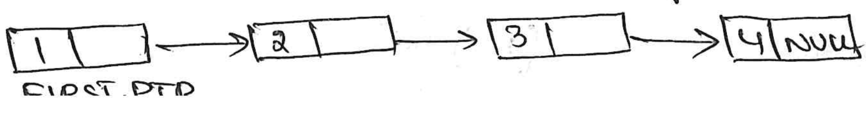
- Create a newnode of cur and insert item at data part.
- Create one pointer PTR and it is pointing to starting node.
- Now, moving pointer PTR from one node to another node up to the pos (Position).
- Now insert the link of cur with PTR→link and change the PTR of link field to address of cur.

```
Algorithm insert_pos (pos, item)
{
    node *PTR = first;
    node * cur = new node;
    i=1;
    Repeat while i< pos-1
            PTR = PTR → link;
            i++;
    end loop
fields { Set cur → data = item;
       { set cur → link = PTR → link;

}   PTR → link = cur;
```

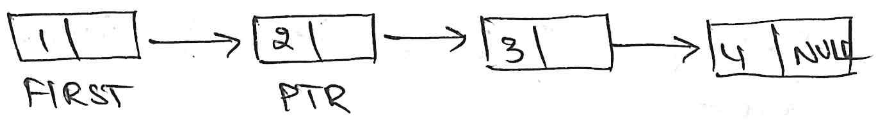
Example:- insert a node of 5 at 3rd position



- Take a pointer PTR, pointing to starting node of a list.

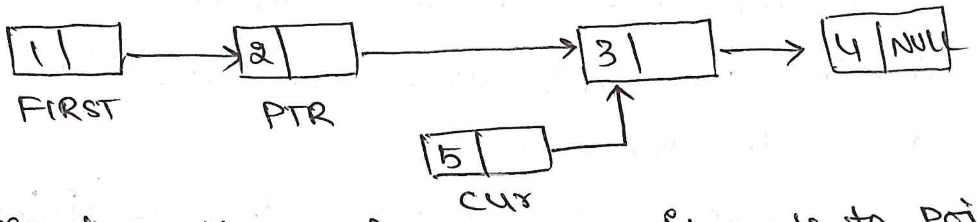


- move the Pointer PTR to POS=3 ⇒ POS-1=2

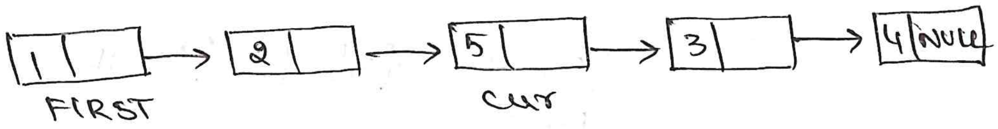


- Allocate memory for new node and insert the fields of data with item and link field with PTR→link

- Add the new node into the list After PTR node



- Change the address field of PTR, it needs to Point cur node after inserting into the list, so PTR→link = cur

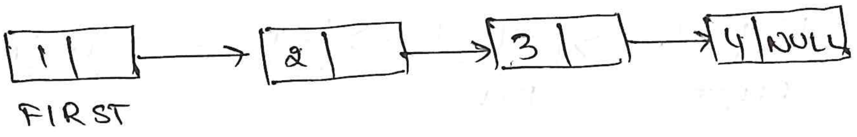


* Deletion at beginning of the list :-

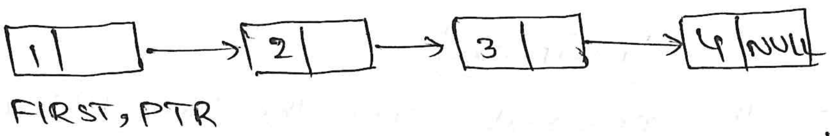
- First, we need to check wether the List is containing the nodes (or) not. IF nodes are presented then only we can delete the nodes from list Otherwise not possible.
- we are taking a Pointer PTR, pointing to the starting node of the list.
- Now, we are changing the FIRST position to the next node, because after deleting first node from the list second node is the starting node in the List.
- NOW, we can delete the PTR node from the list.

```
Algorithm delete-beg( )
{
        if FIRST == NULL
              print "LIST is empty";
              Exit;
        PTR = FIRST;
        FIRST = FIRST→link;
}       delete PTR;
```

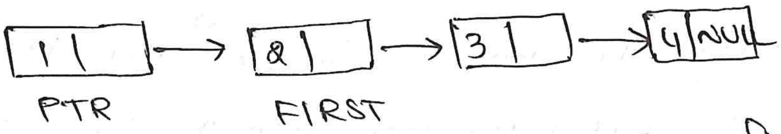
Example :- Deleting the starting node from the list



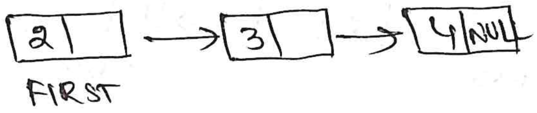
- Take one point PTR, which is pointing to the starting node of the list



- Change the FIRST position to the next node because after deleting first node from the list, 2nd node is starting node in the list.



- Now, delete the PTR from the list. Then final list is

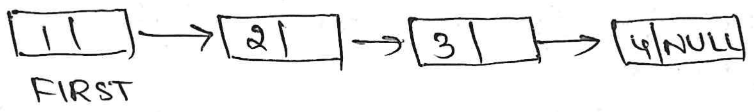


* Deletion of a node at End of the list :-

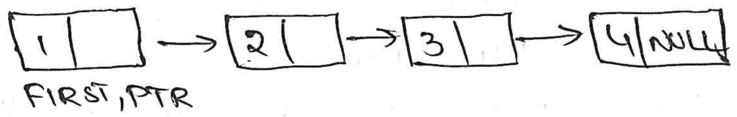
- First, we need to check wether the list containing the nodes or not, if list containing the element then we can delete the nodes otherwise not possible
- We are taking pointer PTR, pointing to starting node in list
- Now move PTR to the last in the list meanwhile PREPTR is pointing to the previous node of the last node (PTR).
- Change the link field of PREPTR node to NULL, because after deleting last from list PREPTR node is the last node.
- Delete the PTR node from the list.

```
Algorithm delete_end ()
{     if FIRST == NULL
            print "List is empty";
            Exit;
      PTR = FIRST;
      Repeat while PTR→LINK != NULL
            PREPTR = PTR;
             PTR = PTR → LINK;
      PREPTR → Link = NULL;
}  [illegible]
```

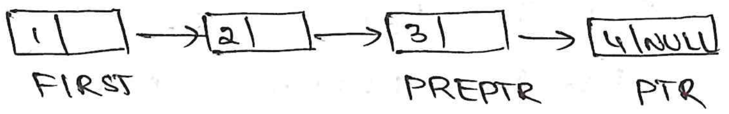
Example :- Delete the last node from the given list



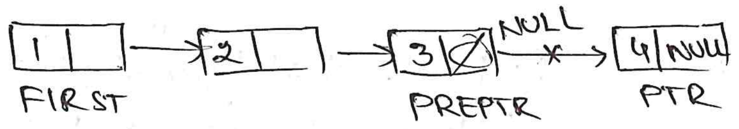
- Take a pointer PTR, which is pointing to the starting node in the list

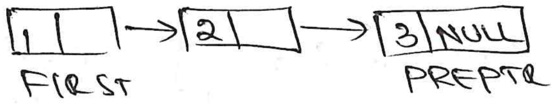


- Move the pointer PTR, to end of the list meanwhile take PREPTR, which is pointing the previous of PTR node



- Now change the link field of PREPTR to NULL, now breaking the link between PREPTR node and PTR node, then delete the last node (PTR), from the list



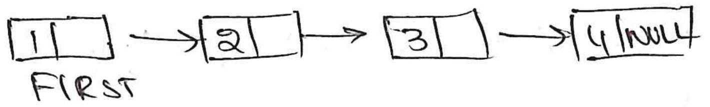


* <u>Deleting particular node from the list</u> :-

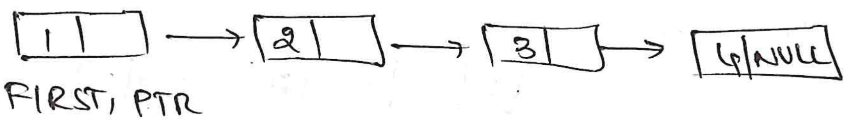
- First, check wether the list containing the nodes or not, if it consists then we can delete the nodes otherwise not possible
- Initially take a pointer PTR, pointing to the starting node of list.
- Now move PTR upto the POS value, meanwhile take PREPTR, which is pointing previous of PTR.
- Now copy the link field of PTR into link field of PREPTR then creating the link between PREPTR node and PTR next node.
- Now, we can delete the PTR node from the list

```
Algorithm delete_pos (POS)
{   if FIRST == NULL
            print "empty list"; Exit;
    PTR = FIRST;
    i = 1;
    Repeat while i < POS
            PREPTR = PTR;
            PTR = PTR -> link;
    PREPTR -> link = PTR -> link;
}   delete PTR;
```

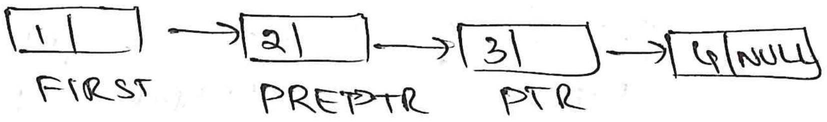
Example:- Delete a node at position of 3 from the list



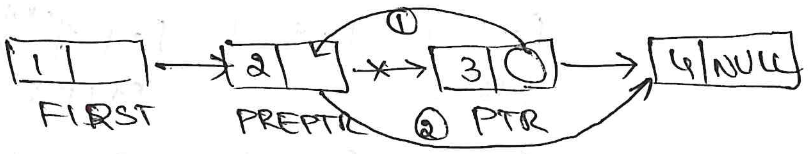
- Take a pointer PTR, pointing to starting node

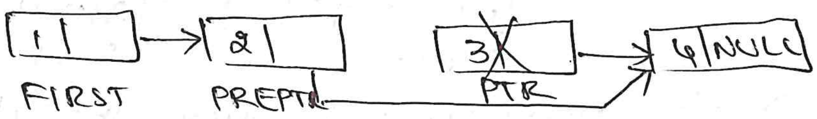


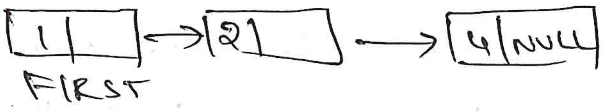
- move PTR to the 3rd position meanwhile PREPTR is pointing to previous of PTR Node.



- Now Copy the link field of PTR into link field of PREPTR, then PREPTR is pointing the next node of PTR, then delete the PTR node from the list







Circular Linked List :-

- A linked list where the last node is pointing to starting node of the list is called circular linked list.
- There is no beginning and ending of list.
- The operations of circular linked list is
  1. insertion of a node
  2. Deletion of a node

1. Insertion of a node:-

- We can insert new node into circular linked list in 2 ways
  1. Insertion of a node at beginning
  2. Insertion of a node at ending

i) At beginning:-

- Create a newnode of Cur and insert the fields of data part and link part

    Cur → data = item;
    Cur → link = first

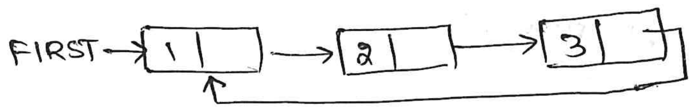
- Take a pointer PTR, it pointing to the starting node of list.
- Move pointer to the last node and change the link field to Cur node

    Ptr → link = Cur.

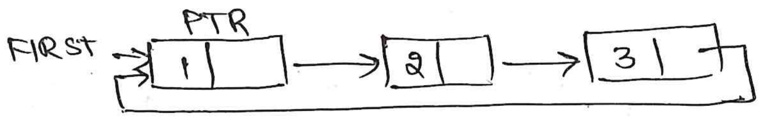
- Move the FIRST to the Cur nodes because Cur is first node in the list.

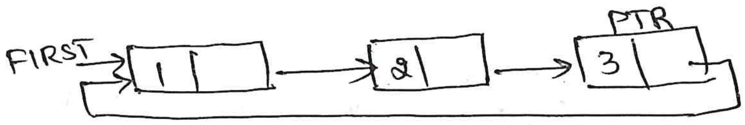
```
Algorithm insert_beg (item)
{
    node *PTR = FIRST;
    node * Cur = new node;
    cur → data = item;
    cur → link = FIRST;
    Repeat while PTR → ~~link~~ != FIRST
        PTR = PTR → Link
    PTR → link = Cur;
    FIRST = Cur;
}
```

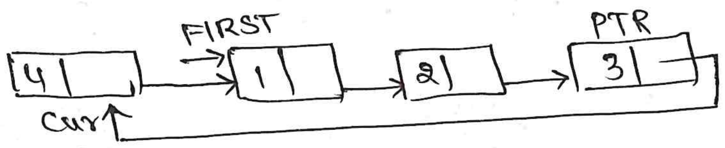
Example :-

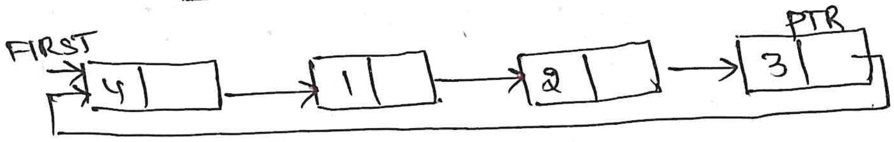
1) 

2) newnode of Cur

3) 

4) 

5) 

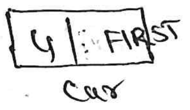
6) 

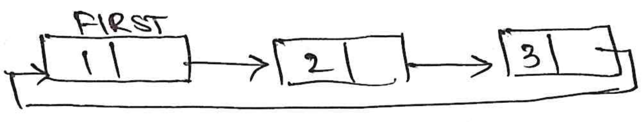
(ii) At Ending :-

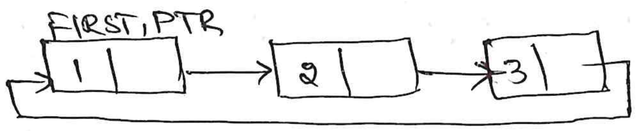
- Create a new node and insert the fields
  Cur → data = item
  Cur → link = FIRST
- Take pointer PTR, Pointing to the Starting node in a list
- Move the pointer PTR to the last node in the list
- Change the address of last node to the cur node, then the newnode is inserted into the list

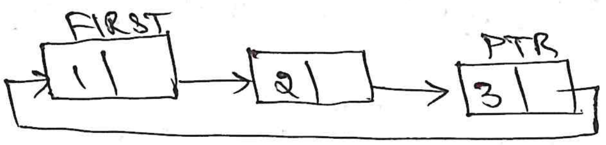
```
Algorithm insert_end (item)
{
        node * cur = new node ;
        Cur → data = item;
        Cur → link = FIRST ;

        node * PTR = FIRST ;
        Repeat while PTR → link != NULL
                    PTR = PTR → link ;
        PTR → link = Cur ;
}
```

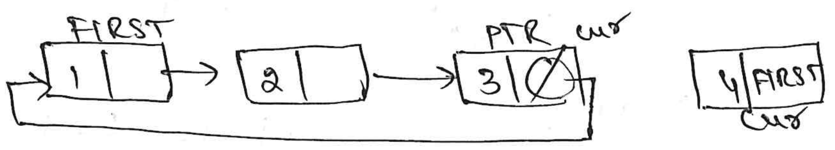
Example :-

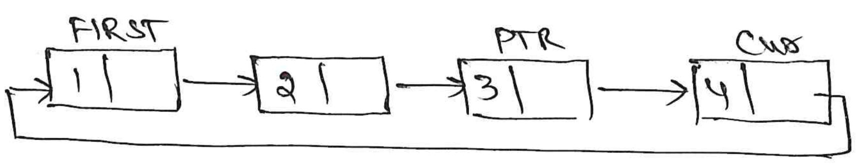
1)


2)


3)


4)


5)


6)


## 2) Deletion of a node:-

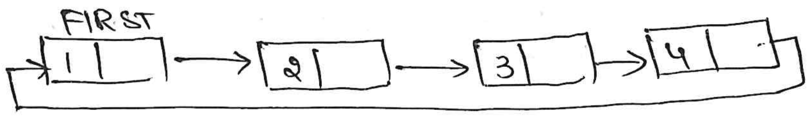
- We can delete a node from circular linked list in 2 ways
  - (i) Deletion of a node at beginning
  - (ii) Deletion of a node at ending

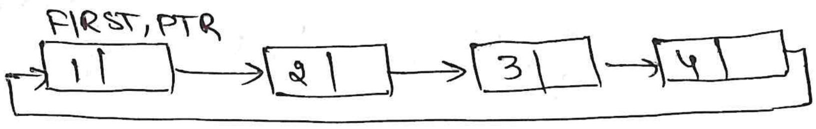
(i) At beginning:-

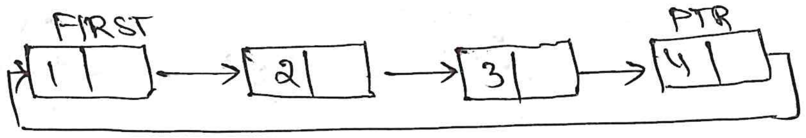
- Initially check wether the list containing the elements or not, if we have, we can delete otherwise not possible.
- Take a pointer PTR, pointing to the starting node in the list.
- Move the pointer PTR to the last node in the list
- Change the link of last node to 2nd node of list (first → link
- Delete the starting node and change FIRST to 2nd node in list.

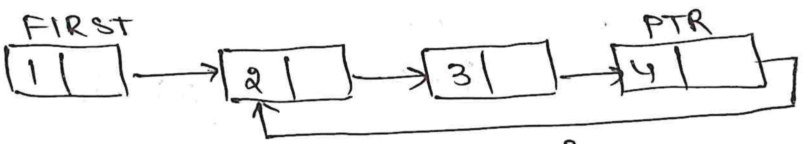
```
Algorithm delete_beg()
{
        if FIRST == NULL
            print " List is empty ";
            Exit;
        node *PTR = FIRST;
        Repeat while PTR → link != FIRST
                PTR = PTR → link;
        PTR → link = FIRST → link;
            delete FIRST;
        FIRST = PTR → link;
}
```

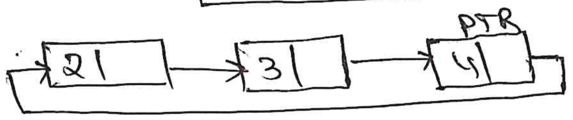
Example :-

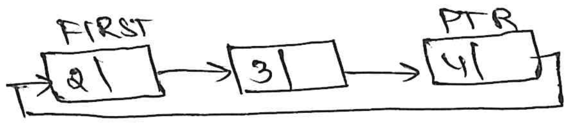
1)


2)


3)


4)


5)


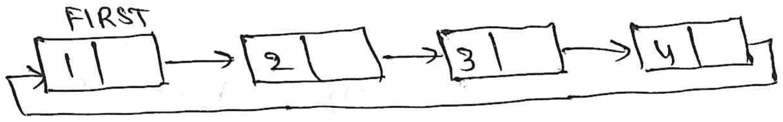
6)


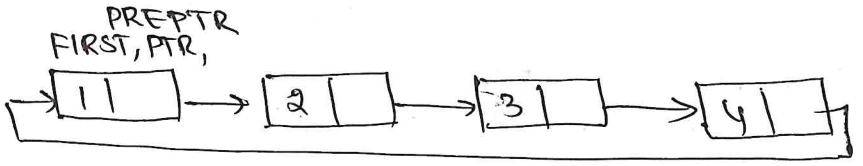
ii) At Ending:-

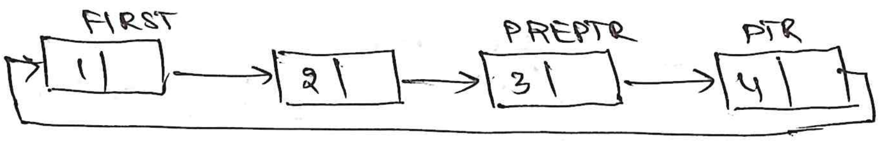
- Take pointer PTR, initially pointing to the starting node.
- Move the pointer PTR to the last node, meanwhile move PREPTR, it points to previous of PTR node
- Change the address of PREPTR link to starting node
- Now, delete the PTR node from the list

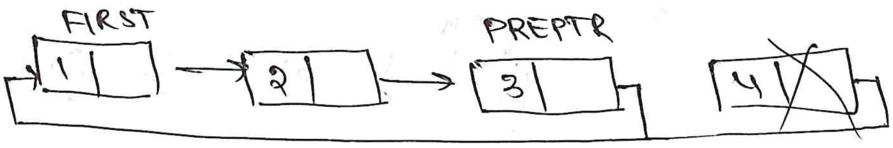
```
Algorithm delete_end ( )
{
    if   first == NULL
         print "List is empty";
         Exit;
    node *PTR = FIRST, *PREPTR = FIRST;
    Repeat while PTR->link != first
         PREPTR = PTR;
         PTR = PTR -> link;
    PREPTR -> link = FIRST
    delete PTR;
}
```

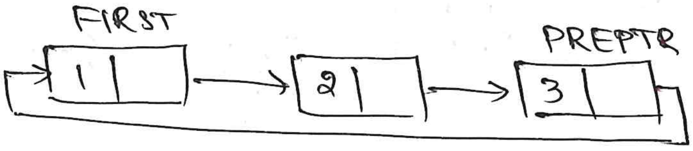
Example:-

1)


2)


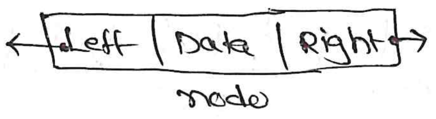
3)


4)


5)


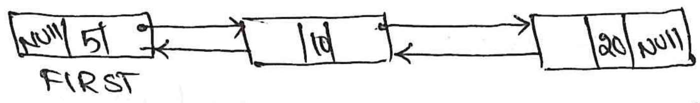
## Double linked list :-

- A double linked list is complex type of linked list, in which Pointer right and pointer left. Right pointing to the successor (next) node and Left pointer pointing to predecessor (previous) node in the list.



- The Structure of the ~~data~~ node in double linked list is

```
class node
{
    int data;
    node *left, *right;
}
```



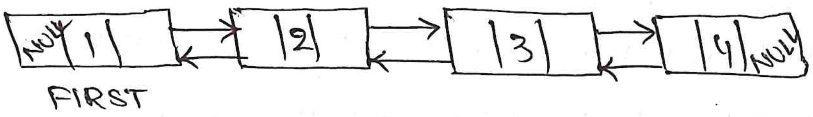
- If we have single node in the list then both left & right are NULL
- For the first node always left is NULL and for last node always right is NULL
- There are Two operations on double linked list, those are
  1) Insertion   2) Deletion

1) Insertion :- We can insert newnode in the list in 3 ways
(i) At beginning (ii) At ending (iii) At given position

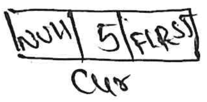
(i) At beginning :-

```
Algorithm insert_beg(item)
{
    node *cur = new node;
    cur → left = NULL;
    cur → data = item;
    cur → right = FIRST;
    FIRST → left = cur;
    FIRST = cur;
}
```

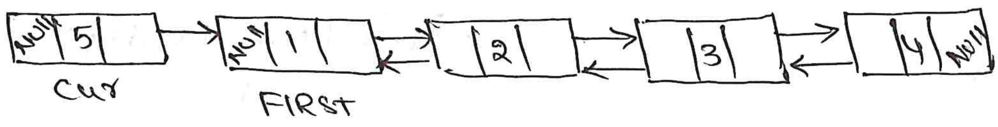
Example:- insert newnode (cur) at beginning of the list



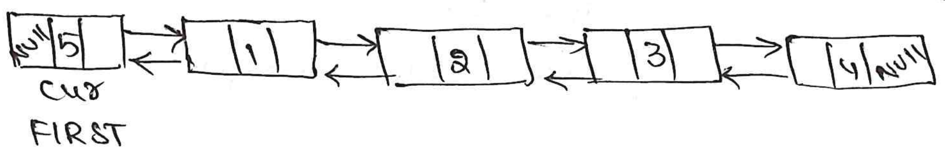
- create the new node and insert the field values



- Add the newnode before the starting node



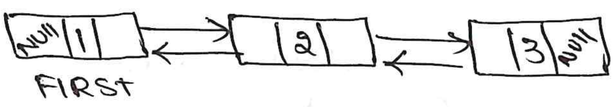
- change the left field of FIRST and change the Cur node as starting node of list by pointing FIRST.



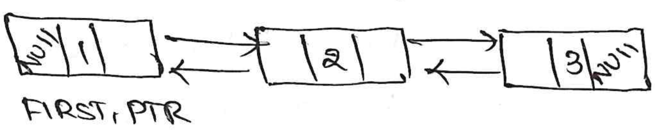
(ii) At end of the list:-

```
Algorithm insert_end (item)
{
    node *cur = new node;
      node *PTR = FIRST;
    Repeat while PTR→right != NUll
            PTR = PTR → link;

    cur → left = PTR
    cur → data = item;
    cur → right = NUll;

}   PTR → right = cur;
```

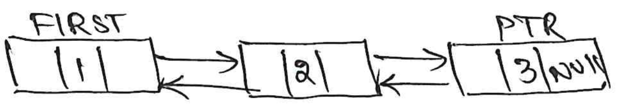
Example:- inserting a newnode at end of the list



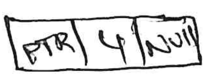
- Take a pointer PTR, pointing to starting node in the list

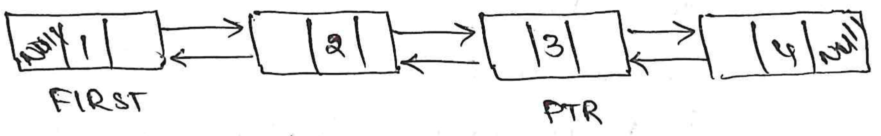


- move the pointer PTR to the last node in the list



- create new node and insert the fields

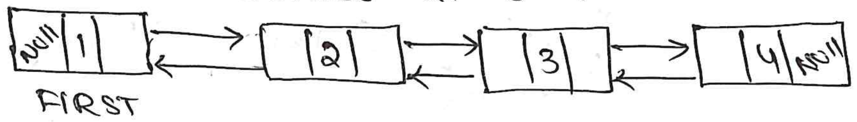




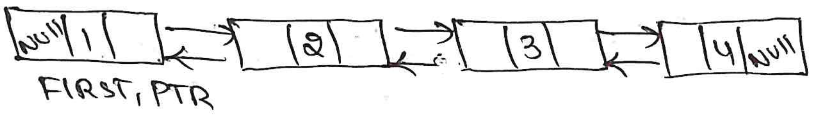
(ii) At given position:-

```
Algorithm  insert_POS(pos, item)
{
    node *cur = new node;
    node * PTR = FIRST;
    i=1;
    Repeat while i < POS-1
                PTR = PTR -> right;
                i++;
    cur -> left = PTR;
    cur -> data = item;
    cur -> right = PTR -> right;

    PTR -> right -> left = cur;
    PTR -> right = cur;
}
```

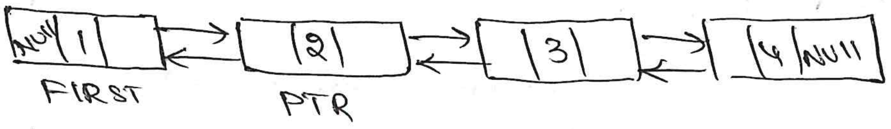
Example :- insert a newnode at 3rd position



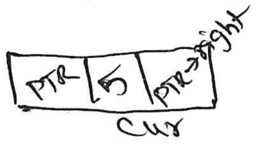
- Take a pointer PTR, pointing to the starting node in the list



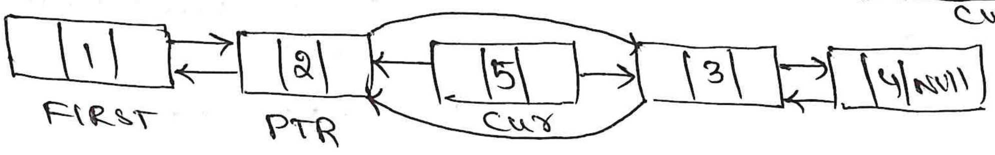
- Move the PTR to the 3rd location means [POS-1](2)



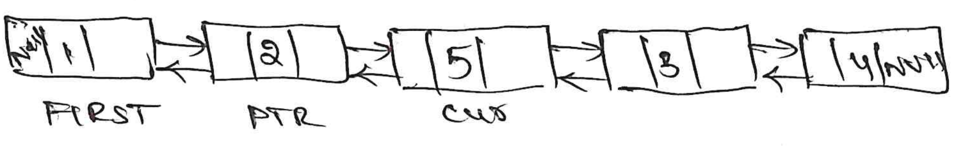
- Create the new node and insert the fields



-



- Change the address of PTR next node left field and PTR node right field to point current node



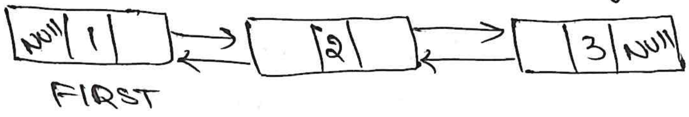
2) Deletion of a Node:-

- we can delete a node from double linked list in 3 ways
  (i) At beginning (ii) At ending (iii) At position.

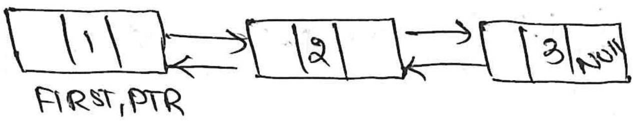
(i) At beginning:-

```
Algorithm delete_beg()
{   if FIRST == NULL
        print "List is empty";
        Exit;
    node *PTR = FIRST;
    FIRST = FIRST -> right;
    FIRST -> left = NULL;
}   delete PTR;
```

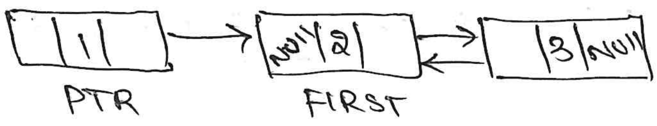
Example:- Delete a node at beginning of the list.



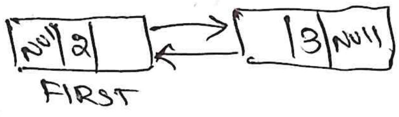
- Take a pointer PTR, pointing to the starting node of a list.



- Move FIRST pointer to the next node and change left field NU



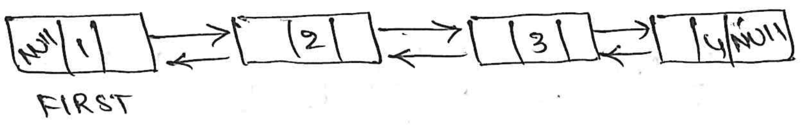
- Now delete the pointer PTR



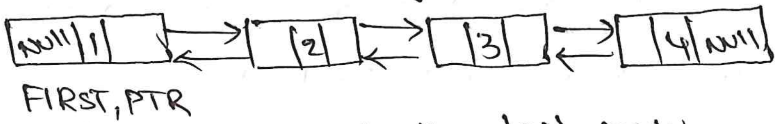
(ii) At Ending :-

```
Algorithm delete_end()
{   if FIRST == NULL
        print "List is empty";
        Exit;
    node *PTR = FIRST;
    Repeat while PTR -> right != NULL
        PTR = PTR -> right;
    PTR -> left -> right = NULL;
}   delete PTR;
```

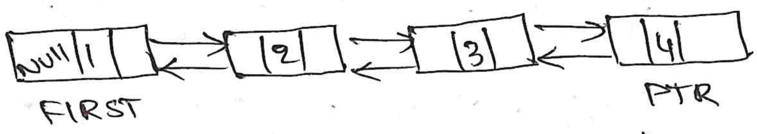
Example:- Deletion of a node at last position in the list



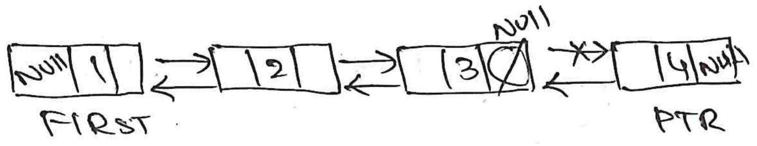
- Take a pointer PTR, initially pointing to the starting node

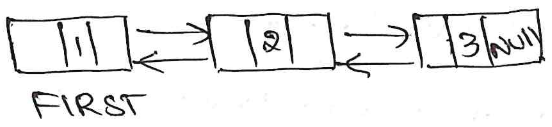


- Move the pointer PTR to the last node



- Change the ~~right~~ link of PTR previous node then delete the PTR node.

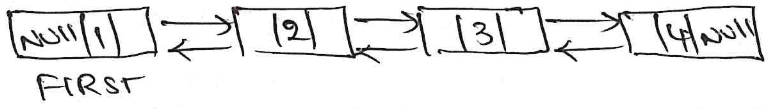




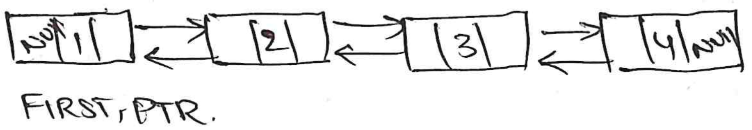
(iii) At given position:-

```
Algorithm delete_pos(pos)
{   if FIRST == NULL
        print "List is empty";
        Exit;
    node *PTR = FIRST;
    i=1;
    Repeat while i < pos
            PTR = PTR -> right;
            i++;
    PTR -> left -> right = PTR -> right;
    PTR -> right -> left = PTR -> left;
    delete PTR;
}
```

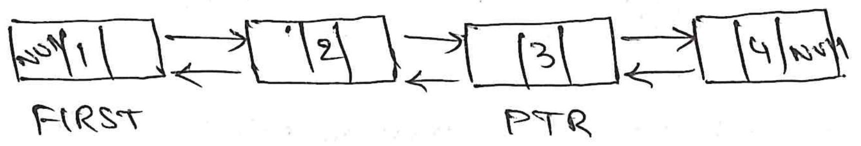
Example:- Deletion of 3rd node from the double linked list



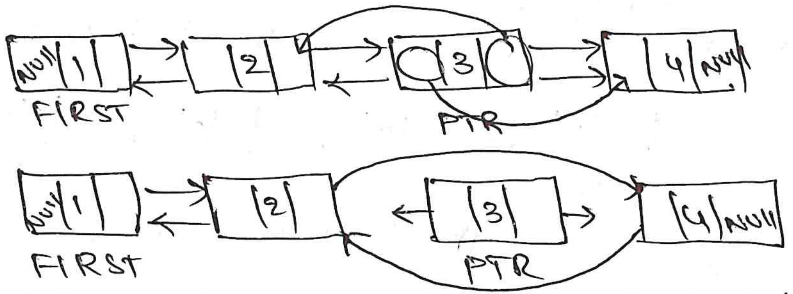
- Take a pointer PTR, initially pointing to starting node



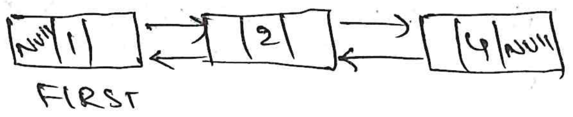
- Move the pointer PTR to the 3rd node in the list



- Now copy the PTR node link fields to the previous node and next ~~fir~~ node

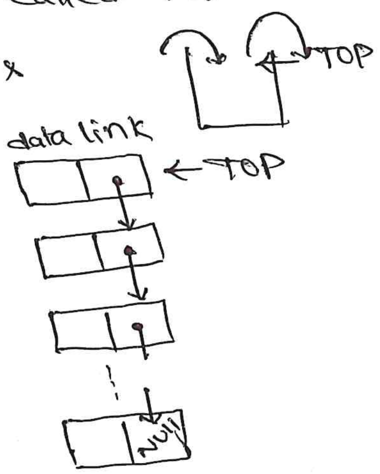


- Now delete the PTR node from the list



## Stack using Linked List :-

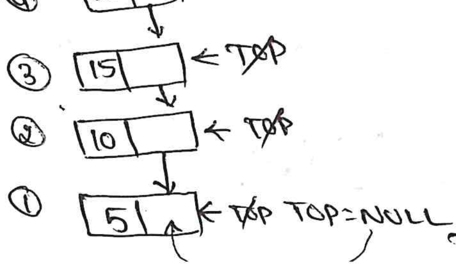
- Stack is a Linear datastructure, in which elements can be inserted & deleted from same end called TOP.
- Initially TOP = NULL, then the stack is not containing elements.



- Stack ADT is

```
class stack
{
    private :
            stack *TOP;
    public :
            bool isempty();
            void push(item);
            void pop();
};
```

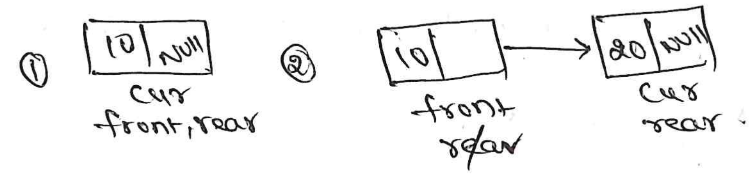
PUSH :- Inserting an element into the stack.

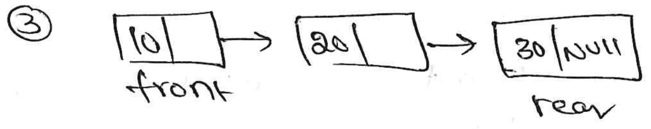


```
Algorithm PUSH (item)
{
    stack *cur = new stack; // create an empty node
    if TOP == NULL
        cur -> data = item;
        cur -> link = NULL;
        TOP = cur;
    else
        cur -> data = item;
        cur -> link = TOP;
```

Cur -> data = item;
cur -> link = TOP;
~~TOP~~ = cur;

Ex:-

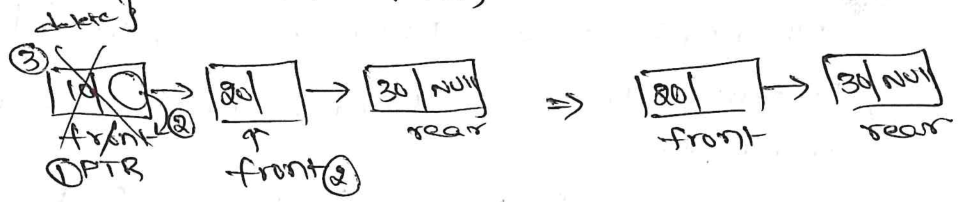




Dequeue :- Deletion of nodes from the Queue.

```
Algorithm Dequeue ( )
{       if front == NULL
            print " Queue is empty";
            Exit
        Queue *PTR = front;
        front = front → link;
        delete PTR;
}
```

Ex:-

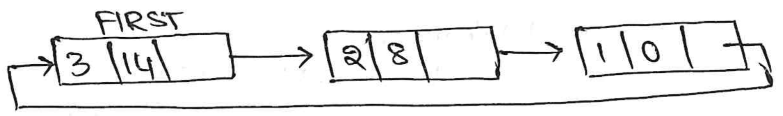


Polynomials :-

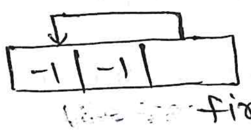
Refer UNIT-I

Circular list representation of Polynomials:-

- circular list means last node link field is pointing to the first node. $P(x) = 3x^{14} + 2x^{8} + 1$



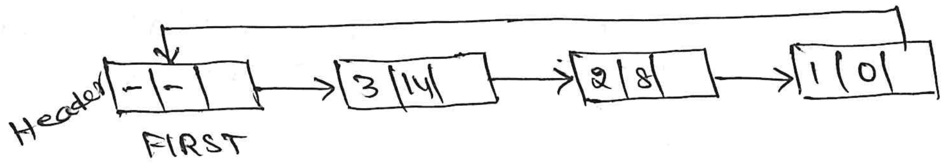
- It causes some problems during addition (or) some other operations of polynomial. So we are introducing zero polynomial
- zero polynomial is represented as header node, either polynomial equation is zero or Nonzero adding header node.



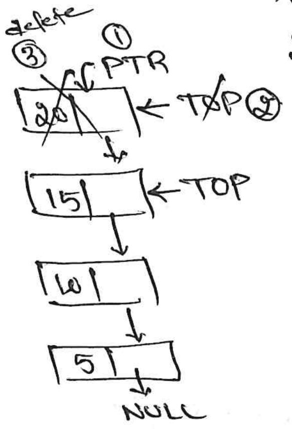
Zero polynomial

- The coef and exp values in zero polynomial (Header) is -1, So other polynomial exp are always greater, copy all terms of second polynomial to resultent polynomial.

$P(x) = 3x^{14} + 2x^{8} + 1$



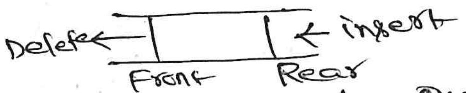
POP :- Deleting a node from the Stack



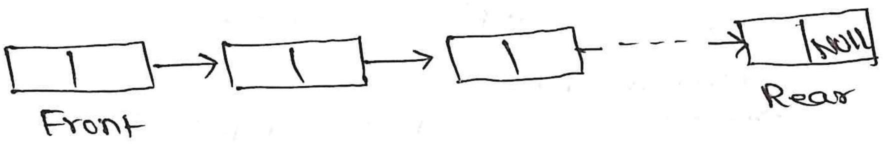
```
Algorithm pop()
{
    if TOP == NULL
        print "stack is underflow";
        Exit;
    Stack *PTR = TOP;
    TOP = TOP -> link;
    delete PTR;
}
```

Queue using Linked List :-

- Queue is a linear datastructure in which elements can be inserted from one end called REAR and deleted from other end called FRONT.



- Initial values of FRONT = REAR = NULL, when the Queue is empty.



- Queue ADT is

```
class Queue
{ private:
        Queue *front, *rear;
  Public:
        bool isempty();
        void enqueue (item);
        void dequeue ();
};
```

Enqueue :- Inserting an element into the Queue

```
Algorithm Enqueue (item)
{
    Queue *cur = new Queue;    // create empty node
    cur -> data = item;
    cur -> link = NULL;
    if front == NULL
            front = rear = cur;
    else
            rear -> link = cur;
            rear = cur;
}
```

## Sparse Matrix:- Refer UNIT-I

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

### * Sparse Matrix Input:

- For representing the sparse matrix in linked representation, first read the header node.
- In header node, contains no. of rows, no. of columns and no. of non-zero terms in the matrix.
- The subsequent input consists triplet form of $(i, j, a_{ij})$

  $i \leftarrow$ row number $j \leftarrow$ column number

  $a_{ij} \leftarrow$ non-zero element.
- Now the order of input will be taken on row major order.

  header node → 1st non-zero → 2nd non zero → --- → last non-zero

$$\underbrace{(5,4,6)}_{\text{no. of rows, cols, non-zero}} \rightarrow (1,1,2) \rightarrow (2,1,4) \rightarrow (2,4,3) \rightarrow (4,1,8) \rightarrow (4,4,1) \rightarrow (5,3,6).$$

### * Delete Sparse Matrix:-

- All the nodes in the sparse matrix are deleted at a time. It is done by deleting the header node.

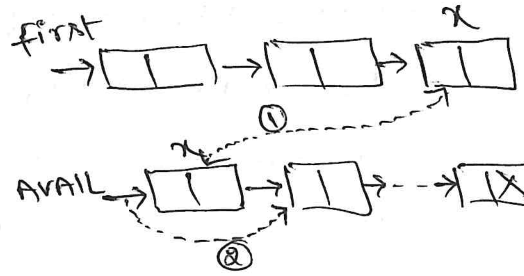
  delete [ ] header ;
- Initially the header node is pointing to AVAIL, later for each iteration we are adding one node at a time to the freepool. Finaly header node contain zero elements.

### Available space list:-

- Destructor will used to delete all the nodes and add those memory cells to freepool (AVAIL).
- We can create a node by using new, delete the nodes by delete.
- All the deleted nodes are pointing to AVAIL (Available space)
- There are three functions getnode(), retnode(), Nlist().
- Three steps for adding node into AVAIL
  - (i) break the link b/w current node & next node
  - (ii) add the current node to AVAIL
  - (iii) replace current node by zero.

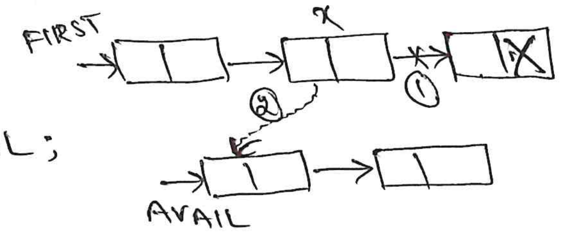
(i) getnode( ) - Take the node from available space list and added to new list

```
getnode ( )
{
   node *x;
   if (AVAIL)
   {
       x = AVAIL;
       AVAIL = AVAIL→link;
   }
   return x;
}
```



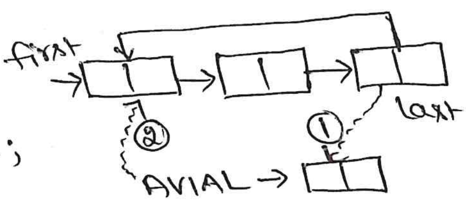
(ii) retnode( ) - It remove the nodes from the list and added to available space

```
retnode (node *x)
{
    x→link = AVAIL;
    AVAIL = x;
    x = 0;
}
```



(iii) Nlist( ) - It remove all the nodes at a time

```
Nlist( )
{   if (last)
    { first = last→link;
      last→link = AVIAL;
      AVAIL = first;
      last = 0;
    }
}
```



Equivalence classes :-

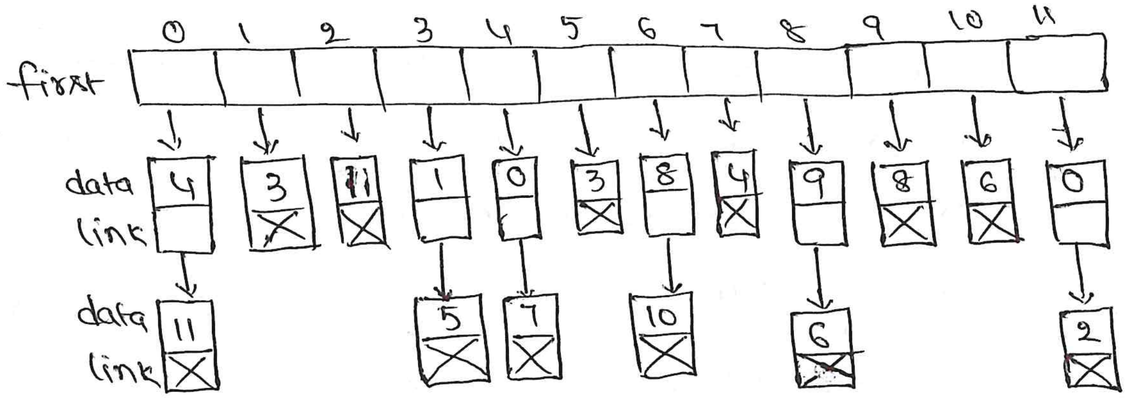
- A relation '≡' over a set $\underline{S}$, is said to be equivalence relation iff it is reflexive, symmetric & transitive over $\underline{S}$.
- Let any polygons $x, y, z$
  - (i) $x \equiv x$ (≡ is reflexive)
  - (ii) $x \equiv y$, $y \equiv x$ (≡ is symmetric)
  - (iii) $x \equiv y$, $y \equiv z$, $z \equiv x$ (≡ is transitive).

Ex! - $0 \equiv 4,\ 3 \equiv 1,\ 6 \equiv 10,\ 8 \equiv 9,\ 7 \equiv 4,\ 6 \equiv 8,\ 3 \equiv 5,\ 2 \equiv 11,\ 11 \equiv 0$

- We are getting three equivalence classes

$\{0, 2, 4, 7, 11\}$ $\{1, 3, 5\}$ $\{6, 8, 9, 10\}$

- Each Equivalence class defines as signal net, it is used to correctness of the marks.



- There are two phases in the algorithm
  (i) The equivalence pairs (i, j) are read in and stored.
  (ii) Begin at <u>0</u> find all pairs of form (0, j), if 0 & j are same class, find transitivity (j, k), k is same class
- Let the no. of input pairs are <u>m</u> and objects are <u>n</u>, take pair[n][n], if element contain pair[i][j] = True. Both are same class.
- In node we are containing data field & link field, need to maintain all pairs. So, we are taking one dimensional array of size <u>n</u>, pointed by <u>first</u>.

```
void Equivalence()
 {
    initialize first;
    while more pairs
     {
        process the pair (i, j);
        take input of next pair;
     }
    initialize for output;
    for (each object)
        output the equivalence class
        that contains in the object.
 }
```

## Generalized Lists :-

### (i) Representation :-

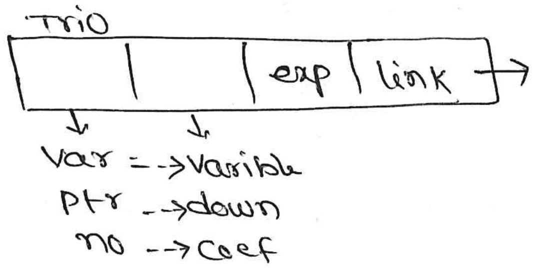
Definition: A generalized list, a finet sequence of elements, ($n \geq 0$), $a_0, a_1, \cdots a_{n-1}$ where $a_i$ is either atom/list.

The elements $a_i$, $0 \leq i \leq n-1$ that are not atoms then sublists.

- Let $A = (a_0, a_1, a_2, \cdots a_{n-1})$  $n \leftarrow$ length.

  $\uparrow$ name of list (capitals)  $\uparrow$ atoms (Lowercase)

- If $a_0$ is head then $(a_1, a_2, \cdots a_{n-1})$ are tail.
- Let polynomial $P(x,y,z)$, it can represent by using four variables like coef, exp x, exp y, exp z
- Node representation contain four fields like type of node, name of node or coef, exponential and link field



- Node is defined as

```
enum Triple { var, ptr, no }
class polynode
{
    polynodes *link;
    int exp;
    Union { char variable;
            polynodes *down;
          } int coef;
}; Triple Trio;
```

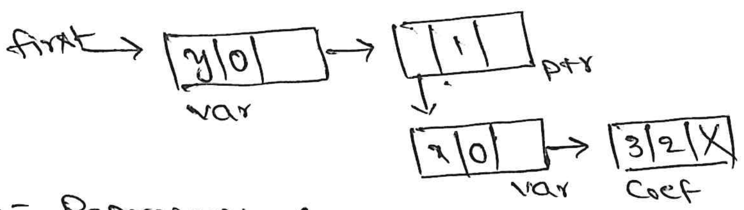
- Three Types of nodes

(i) Trio ≡ var → header node points to the list
variable → name of variable
exp → zero

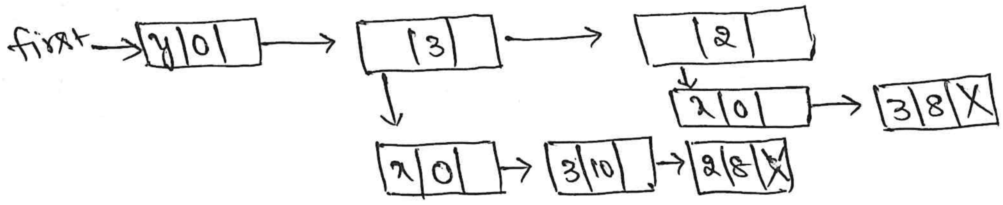
(ii) Trio ≡ ptr → down points to one or more nodes
exp → corresponding variable

(iii) Trio == no → coef
exp

Ex:- $3x^2y$



Ex:- Represent $((3x^{10}+2x^8)y^3+3x^8y^2)$



Generalized lists:-

A = ( ) – empty list, length = 0

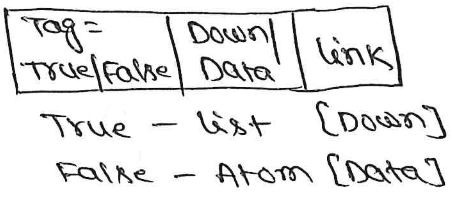
B = (a, (b, c)) – length = 2, a is atom, second is list (b, c)

C = (B, B, ( )) – length = 3, Two lists & one empty list

D = (a, D) – length = 2, one is atom, recursive list.

- list may be shared to another list as a sublist or recursive list

- each node contain 3 fields

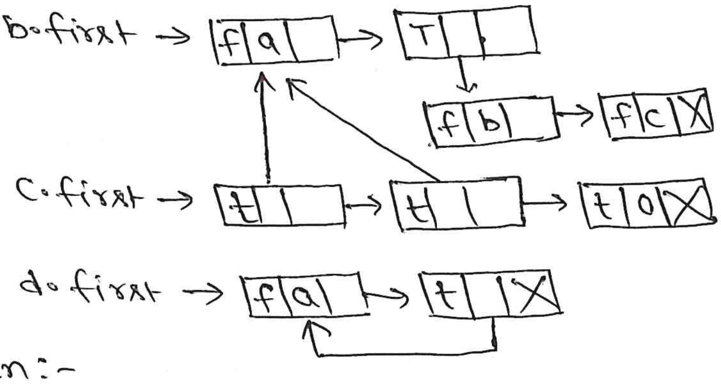


A = ( )     a.first = NULL     – empty list

B = (a, (b, c))     b.first →

C = (B, B, ( ))     c.first →

D = (a, D)     d.first →



Template class Chain:-

(i) Implement Chains with Templates

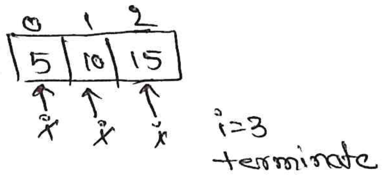
- Generally templates are used for code reusability.
- chainlist <T> class is friend of node<T> class that means, node <int> can be accessed by chainlist <int> but not chainlist <float>
- An empty chain of integer list can be created as

chainlist <int> intlist;

(ii) Chain Iterators:-

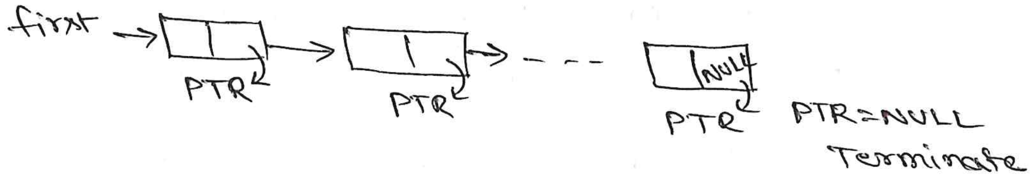
- Iterator is one of the object to access elements in the list one by one.

Ex:-
```
void main( )
{   int a[3] = {0,1,2}
    for (int i=0; i!=3; i++)
            cout << a[i];
}
```



- In chains

```
for (node *PTR; PTR != NULL; PTR = PTR→link)
        cout << PTR→data;
```



- ChainIterator class is nested class of chain with public access specifier.

```
ChainIterator begin()
    { return chainIterator (first); }
ChainIterator end( )
    { return chainIterator (NULL); }
```

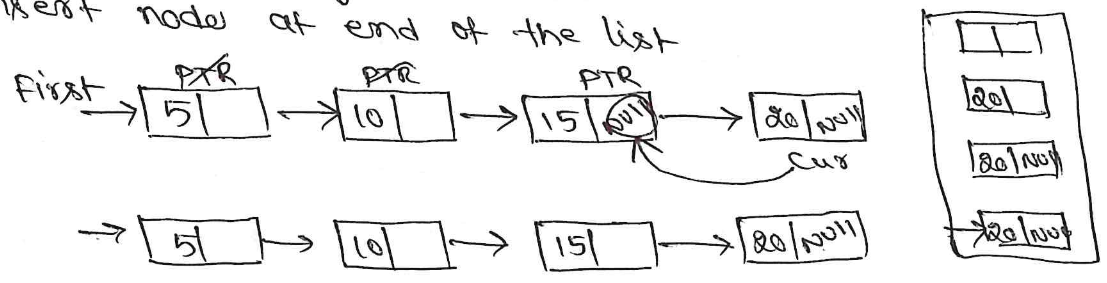
- Sum of elements in chain

```
sum = accumulate (begin(), end());
```

(iii) Chain Operators:-

- Chains can perform there are three operations
  - (i) insert node at end of the list
  - (ii) Concatenate Two lists
  - (iii) Reversing a list

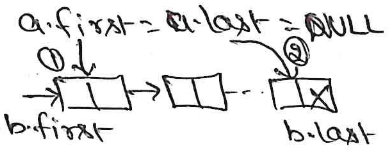
(i) insert node at end of the list

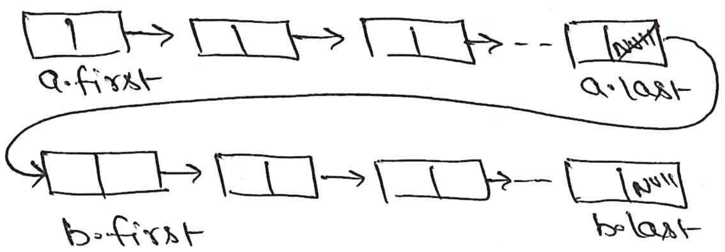


```
Algorithm insert_end (item)
{
    node *PTR = FIRST;
    node *cur = new node;      }  create node &
    cur -> data = item;        }  insert fields
    cur -> link = NULL;        }
    Repeat while( PTR -> Link != NULL)
          PTR = PTR -> link
    PTR -> link = cur;
}
```

(ii) Concatenate Two lists :-

```
Algorithm concatenate (a, b)
{
    if (a.first != NULL)
    {
        a.last -> link = b.first;
        a.last = b.last;
    }
    else
    {
        a.first = b.first;
        a.last = b.last;
    }
    b.first = b.last = NULL;
}
```





(iii) Reversing a list :-

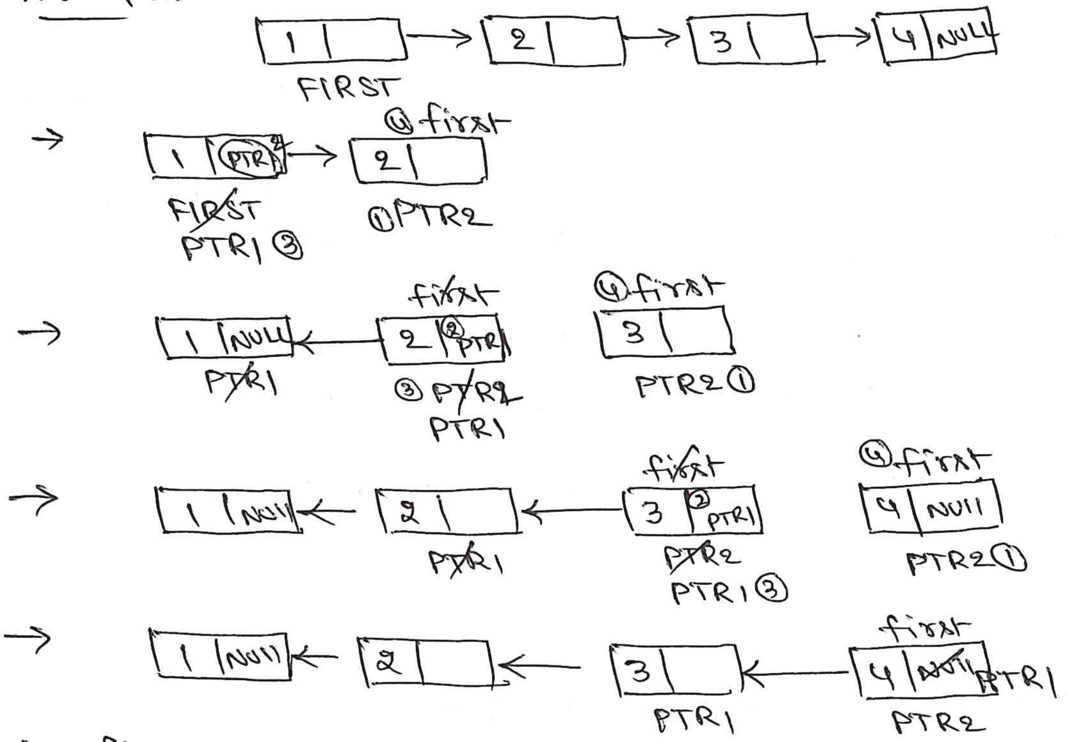
- First create a single linked list (chain) having nodes.
- Initialy Take Two pointers PTR1 = NULL & PTR2
- Repeat the following process upto the last node
   i. Take first Two nodes from the list and PTR2 pointing to the second node.
   (ii) change the first nodes link field with PTR1 value
   (iii) Now pointing the first node as PTR1

(iv) change the FIRST, Pointing to PTR2 because PTR2 is the starting node after reversing.

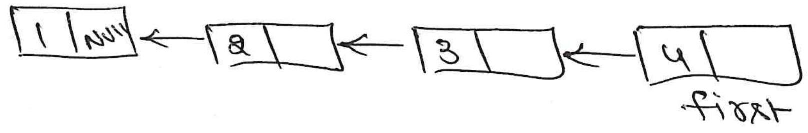
- After moving to the last node change the link field of first node after reversing by PTR1.
- Finally we are reversing the given chain (single linked list).

```
Algorithm reverse ( )
{
        PTR1 = NULL, PTR2;
        Repeat while first→link != NULL
                PTR2 = first → link;
                first → link = PTR1;
                PTR1 = first;
                first = PTR2;

        first → link = PTR1;
}
```

Example:-



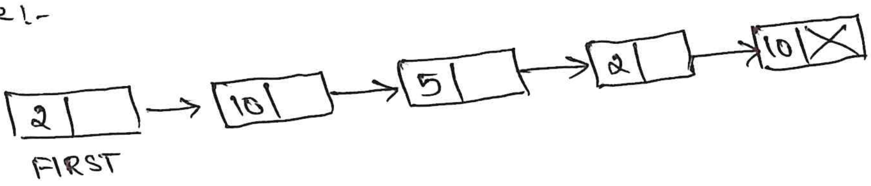
→ final reverse list is



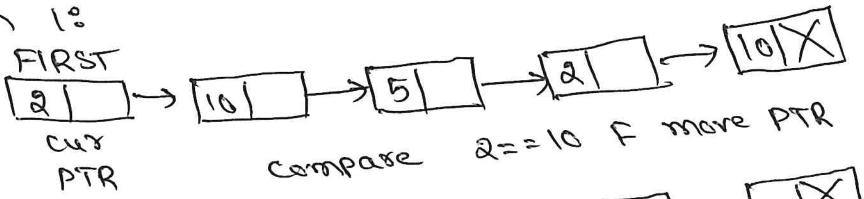
## Removal of Duplicates from the list:-

```
Algorithm    removedup ( )
{
    node * cur = first;
    Repeat while  cur != NULL
        node * PTR = cur;
        Repeat while PTR→link != NULL
            if (PTR→link→data == cur→data)
                PTR→link = PTR→link→link;
            else
                PTR = PTR→link;
        end loop
        cur = cur → link;
    end loop
}
```

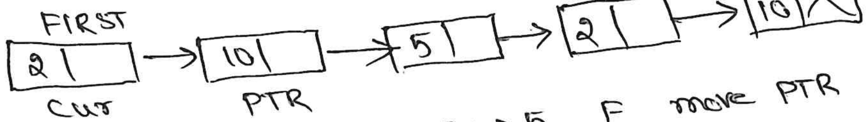
Example:-



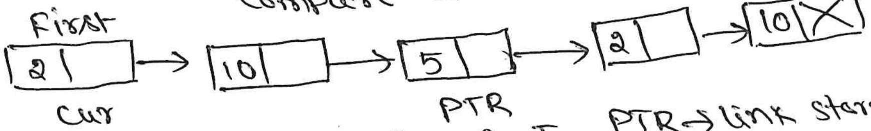
Iteration 1:



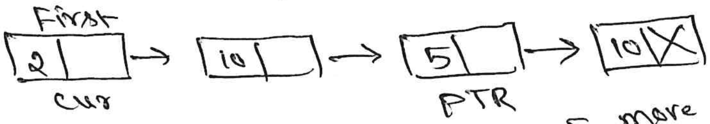
compare 2==10 F move PTR



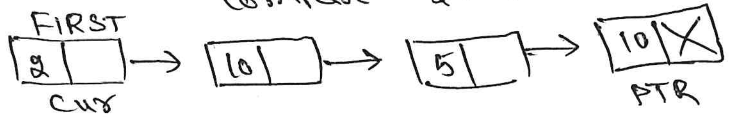
compare 2==5 F move PTR



compare 2==2 T, PTR→link store address of 10 by removing 2

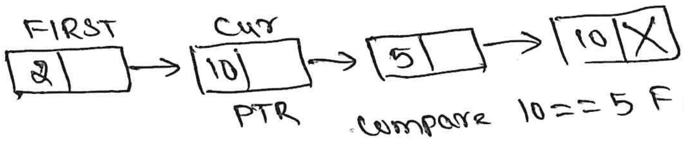


compare 2==10 F, move PTR to next node

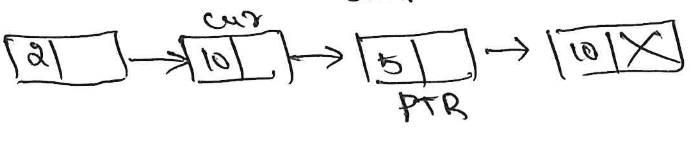


Stop iteration.

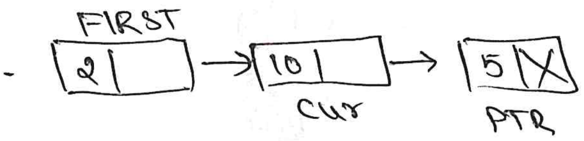
Iteration : 2



compare 10==5 F, move PTR

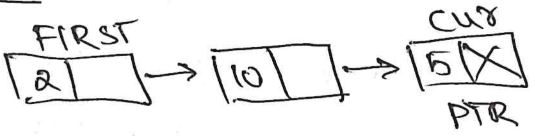


Compare 10==10 T, PTR→link stores the address of next node of 10 which is NULL



Stop Iteration

Iteration 3 :

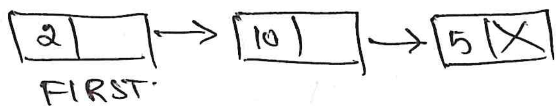


PTR→link is NULL stop

cur = cur→link

cur is NULL so stop all iterations

→ Finally our list without duplicates is



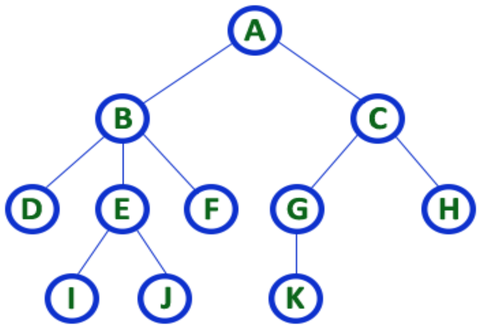
**SYLLABUS:** **UNIT-IV**

**TREES:** Terminology, **Binary Trees:** The Abstract Data Type, Properties of Binary Tress, Binary Tree Representations, Binary Tree Traversal (Inorder, Preorder, and Postorder Traversals), **Thread Binary Trees:** Threads, Inorder Traversal of a Threaded Binary Tree, Inserting a Node into a Threaded Binary Tree, **Priority Queues**: Heaps, Definition of a Max Heap, Insertion into a Max Heap, Deletion from a Max Heap, **Binary Search Trees:** Definition, Searching a Binary Search Tree, Insertion into a Binary Search Tree, Deletion from a Binary Search Tree, Height of Binary Search Tree.

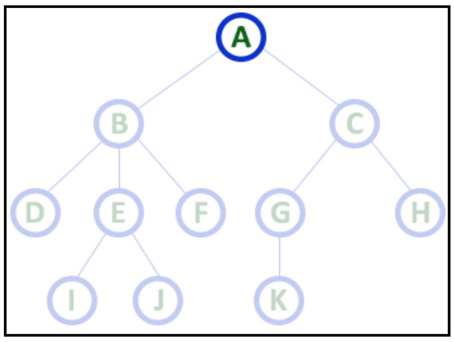
## TREEs:

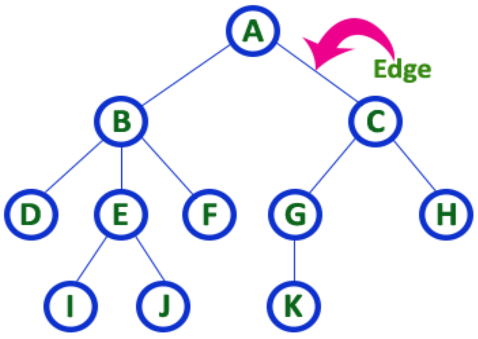
**Definition:** Tree is a non-linear data structures in which collection of elements are arranged in hierarchical structure. In Tree every individual element is called as Node

Example:



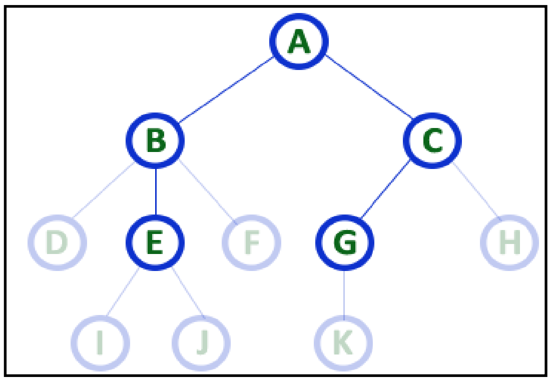
**1. Root:** A tree contains unique first node which is shown at top of the tree is called as Root Node **(or)** The node which has no parent node called Root Node. Every tree must have only one root node. Root node is the origin of tree. Here A is root node.

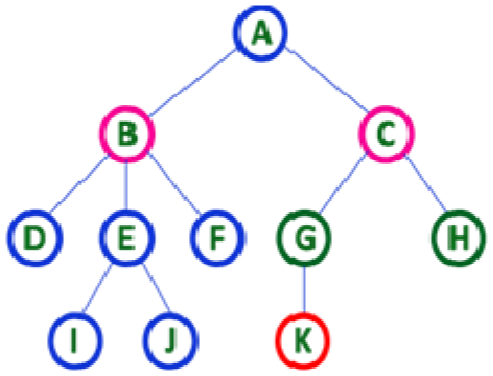




**2. Edge:** In a tree, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

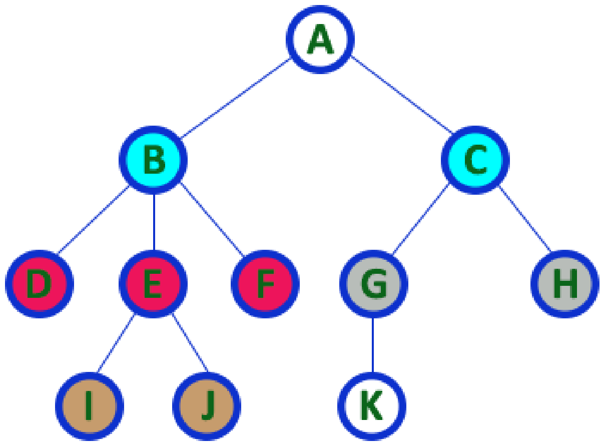
**3. Parent** In a tree, an immediate predecessor of a node called as parent node. Parent node can also be defined as "The node which has child / children". In following diagram A,B,C,E & G are parent nodes.





**4. Child:** In a tree, an immediate successor of a node 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.

**5. Siblings:** In a tree, nodes which belong to same Parent are called as SIBLINGS. The nodes with same parent are called as Sibling nodes.

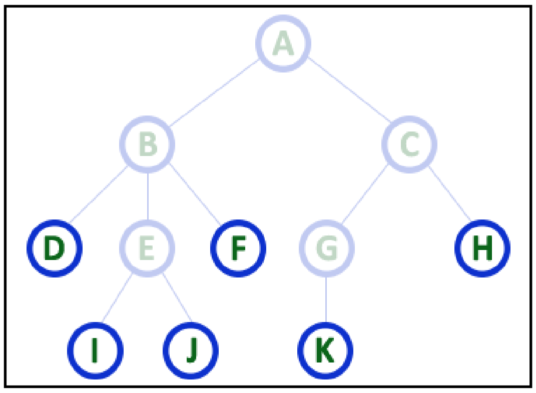


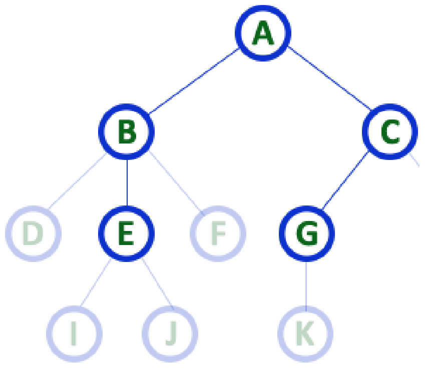
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

**6. Leaf/ External Nodes/ Terminal Nodes**

In a tree, the node which does not have child is called as LEAF/External/Terminal Node. Leaf is a node with no child. Here D,I,J,F,K & H are leaf nodes.



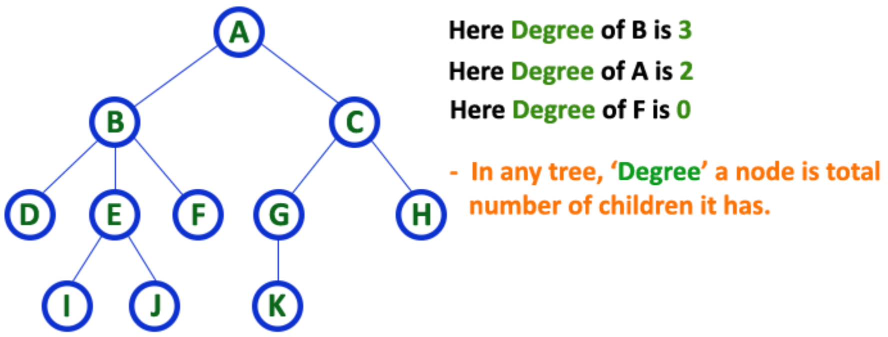


**7. Internal Node/Non-Terminal Node:**

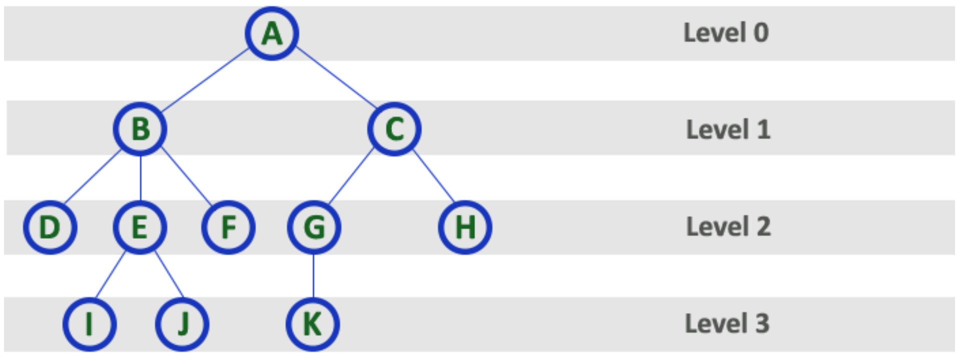
In a tree, the node which has at-least one child is called as INTERNAL Node/Non-Terminal Node. In a tree, nodes other than leaf nodes are called as Internal Nodes. Here A,B,C,E & G are internal nodes.

**8. Degree**

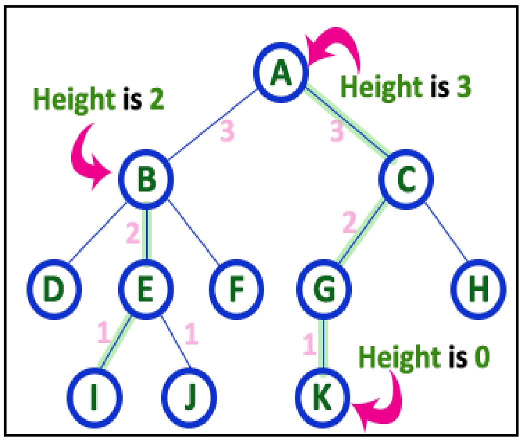
In a tree, the total number of children of a node is DEGREE of that Node. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'

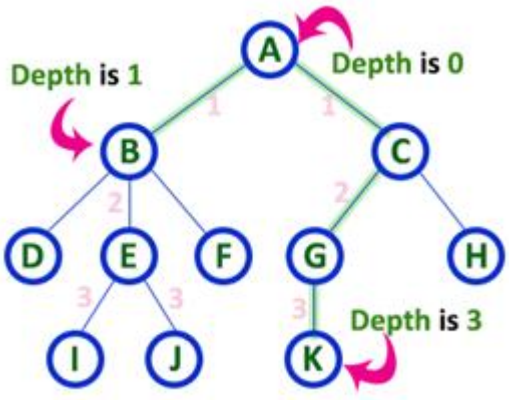


**9. Level:** In a tree, Level is rank of hierarchy, the whole tree is leveled. The level of root node is Level 0 and the immediate children of root node are at Level 1 and their immediate children's are at Level 2 and so on.



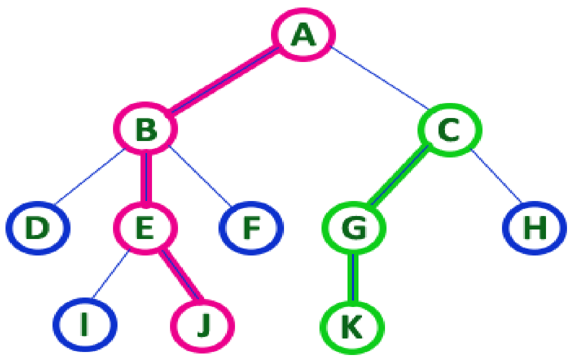
**10. Height:** In a tree, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. Height of the root node is said to be height of the tree. Height of all leaf nodes is '0'.





**11. Depth:** In a tree, the total number of edges 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 is said to be Depth of the tree. Depth of the root node is '0'.

**12. Path:** In a tree, the sequence of Nodes and Edges from source node to destination node is called as PATH between those 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.

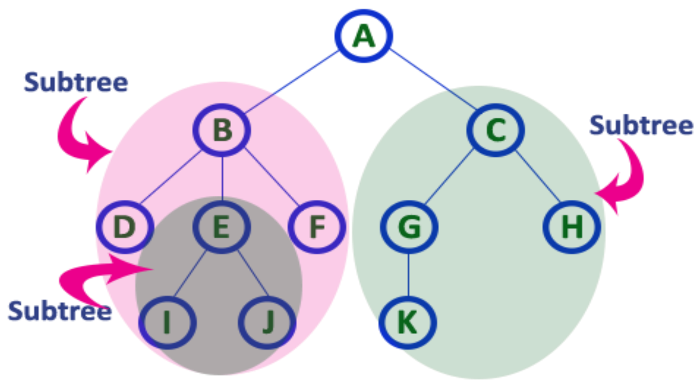


- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

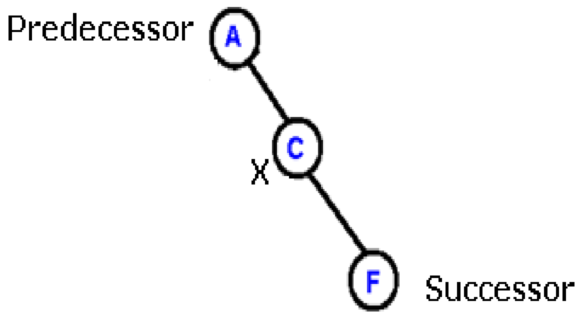
**13. Sub Tree:** In a tree data structure, Every child node will form a sub-tree on its parent node.



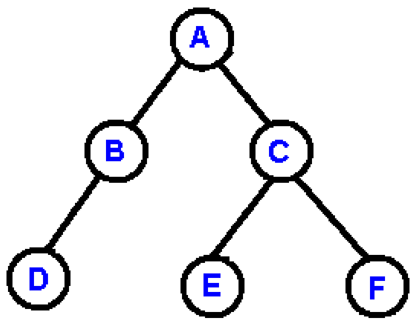
**14. Predecessor:** Consider the node X, then the node previous to node X is called predecessor node.

**15. Successor:** Consider the node X, then the node that comes next to node X is called successor node.

**16. Ancestors:** In a tree, parent of a node or parent of parent node, or itself is the Ancestors.
**17. Descendant:** In a tree, child of a node or child of child node or itself is the Descendant.



## Binary Tree:-



**Definition:** A Binary tree is a finite set of nodes which may be either empty or consists of single node called root node, and two disjoint nodes called left child and right child.

- In Binary tree the maximum degree of any node is almost 2.
- A Binary tree may consist Zero degree nodes or One degree nodes or Two degree nodes.

**Left child:** The node present to the left of the parent node is called left child.

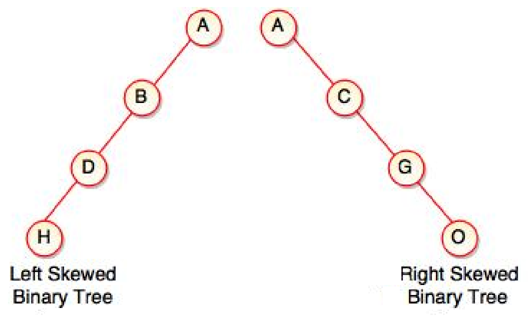
**Right child:** The node present to the right of the parent node is called right child.

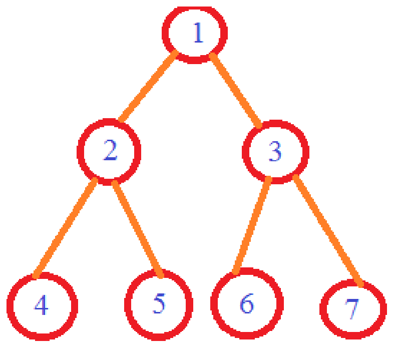
## TYPES OF BINARY TREES:

**Skewed Binary Tree:** It is a binary tree in which new nodes can be added only to one side of the binary tree then it is a skewed binary tree.

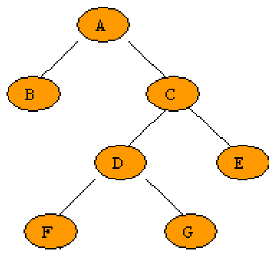
**Strictly binary Tree**: Every non-terminal node in a binary tree consists of non-empty set of left sub-tree and right sub-tree then it is called strictly binary Tree.

**Complete binary tree:** It is a binary tree in which every level consists of maximum number of possible nodes except last level and all the nodes are inserted from left to right.





Complete binary tree



Strictly binary tree

## ADT of Binary Tree:

```
Abstract datatype Binary_tree
{
        instances:
                a finite set of nodes either empty or consisting of a root node, left Binary_tree and
                right Binary_tree
        operations:
                Binary_tree();          // create an empty binary tree
                bool Isempty(bt);       //return true iff the binary tree is empty
                Binary_tree maketree((Binary_tree bt1,item,Binary_tree bt2) - return binary tree
                                whose left subtree is bt1 and whose right subtree is bt2 and whose
                                root node contains data item.
                Binary_tree LeftSubtree(bt) -  return the left subtree of bt.
                Binary_tree RightSubtree(bt) - return the right subtree of bt.
                Binary_tree RootData(bt) - return the data in the root node of bt.
}
```

## PROPERTIES OF BINARY TREES:

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

1. For a given binary tree contains **N** nodes then number of Edges in binary tree is **N-1**
2. If *h* = height of a binary tree, then Minimum number of nodes = **h+1** and Maximum number of nodes=$\mathbf{2^{h+1}-1}$
3. If n = number of nodes of a binary tree, then minimum height of the binary tree is $\mathbf{\log(n+1)-1}$ and maximum height is **n-1**
4. In complete binary tree at depth D, the maximum number of node are $2^D$ can contain at most one node at level 0 (the root), it can contain at most $2^L$ node at level L.

5. For given i value, where i is position of anode, then
    a. The position of the parent node is **i/2**
    b. If left child exists, then position is **2i**
    c. If left child exists, then position is **2i+1**

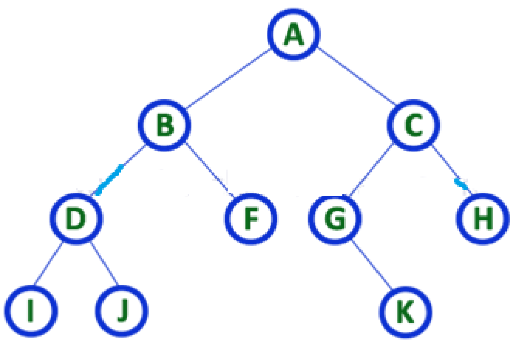
## BINARY TREE REPRESENTATION:

We can represent binary trees in **two** ways

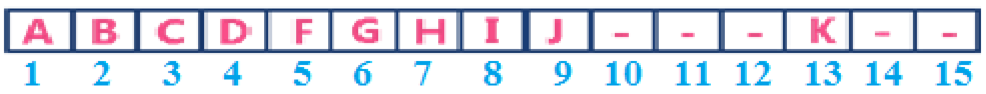
1. Array representation of binary trees.
2. Linked representation of binary trees.

### 1. Array representation of binary trees:

- An array can be used to store the nodes of a binary tree in sequential order.
- An array of size **$2^{k+1}$** is declared where; **k** is the depth of the tree
- For example if the depth of the binary tree is 3, then maximum $2^{(3+1)}$-1 = 15 elements will be present in the node and hence the array size will be 16. This is because the elements are stored from position one leaving the position 0 vacant.



- The root element is always stored in position **1**, the successive memory locations are taken by its left and right child's.
- An array of bigger size is declared so that later new nodes can be added to the existing tree.
- The following binary tree can be represented using arrays as shown.



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | F | G | H | I | J | - | - | - | K | - | - |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Advantages:

– Direct access to any node can be possible
– Finding of parent or left or right childs of a node is faster.

Disadvantages:

– Memory wastage if we use skewed binary trees
– The array size is fixed so the maximum depth of a tree is fixed
– Insertion and deletions of a node is costlier as other nodes has to adjust to proper positions.

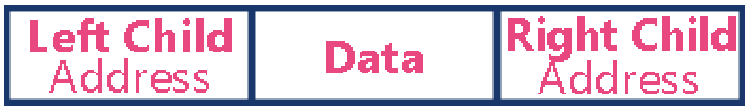
2. **Linked representation of binary trees:**

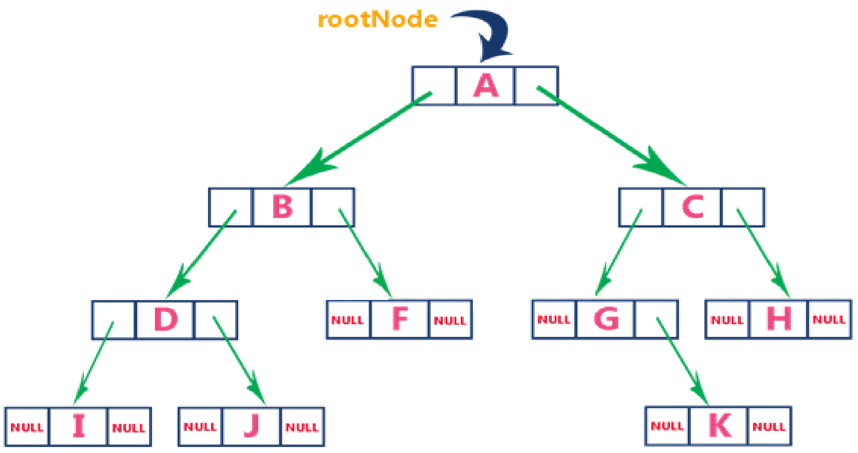
In linked representation every node consists of three fields.

- Data: It will store the information of given node.
- Left Child: It will store the left child address if it is existed, otherwise it is NULL.
- Righ Child: It will store the right child address if it is existed, otherwise it is NULL.

The structure of the node in C++

```
class Node
{  public:
        Node *left;
        int data;
        Node  *right;
};
```





Advantages:

– No memory Wastage

– Size of the depth is not fixed

– Insertion and deletions can perform at any node with out moving other nodes.

Disadvantages:

– No direct access for a node.

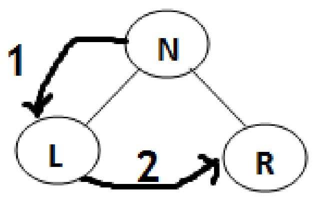
– Additional space for storing left and right child address and at leaf level we are storing NULL.

## <u>BINARY TREE TRAVERSALS:-</u>

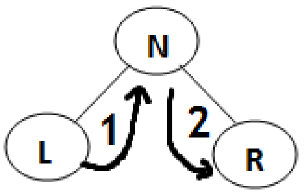
- Traversing a binary tree is the process of visiting every nodes in the tree exactly once in systematic way.
- In linear data structures we can traverse the elements in sequential order, but in non-linear data structures we can traverse the elements in different ways.
- There are different algorithms for traversal

(i) Pre Order Traversal
(ii) In Order Traversal
(iii) Post Order Traversal
} **Depth First Traversal**

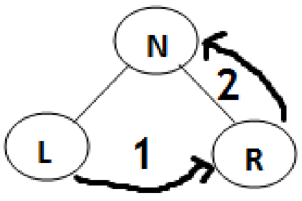
(iv) Level Order Traversal – **Breadth First Traversal**



**Pre Order (N L R)**



**In Order (L N R)**

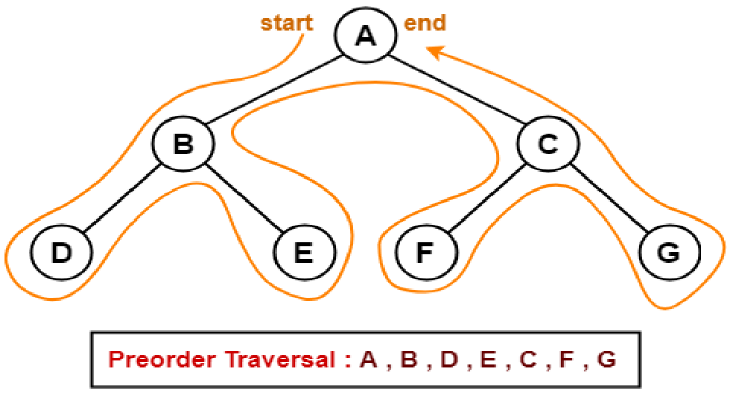


**Post Order (L R N)**

(i) **Preoder Traversal:-**

- To traverse a non-empty binary tree in preorder the following operations are performed recursively at each node.
  - Visit the root node
  - Traverse the left sub-tree
  - Traverse the right sub-tree
- In this root node is visited before traversing its left and right sub-trees.

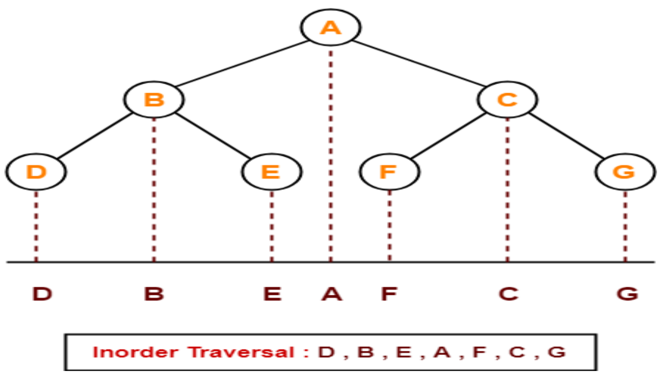
```
void Preorder (node *root)
{
    if (root == NULL)
        return;
        cout<< root->data;
     Preorder(root->left);
     Preorder(root->right);
}
```



(ii) **Inoder Traversal:-**

- To traverse a non-empty binary tree in Inorder the following operations are performed recursively at each node.
  - Traverse the left sub-tree
  - Visit the root node
  - Traverse the right sub-tree
- In this left sub-tree is traversed recursively before visiting root node.
- After visiting the root node the right sub-tree is traversed recursively.

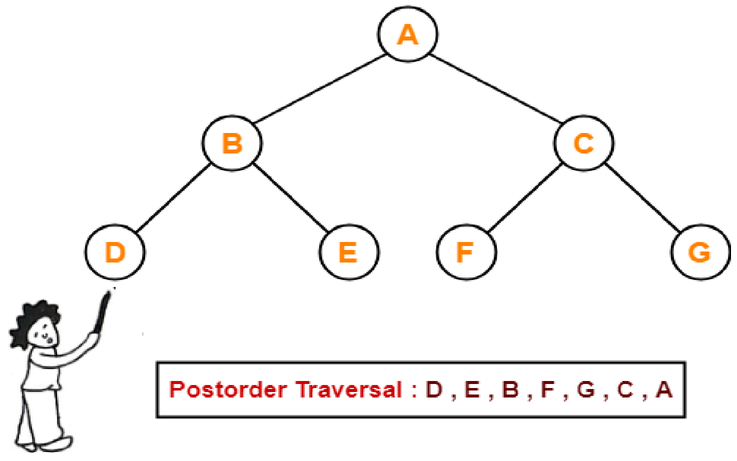
```
void Inorder (node *root)
{
          if (root == NULL)
                    return;
          Inorder(root->left);
                 cout<< root->data;
          Inorder(root->right);
}
```



(iii) **Postoder Traversal:-**

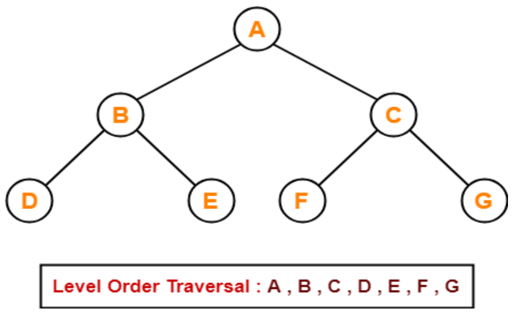
- To traverse a non-empty binary tree in Postorder the following operations are performed recursively at each node.
  - Traverse the left sub-tree
  - Traverse the right sub-tree
  - Visit the root node
- In this left sub-tree is traversed recursively before traversing right sub-tree, After traversing right sub-tree visit root node.

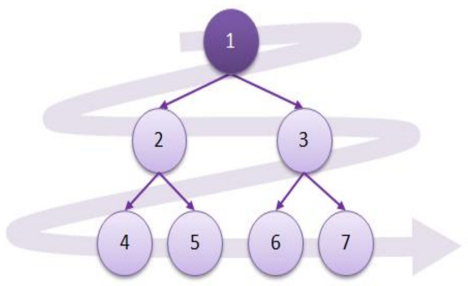
```
void PostOrder(node *root)
{
     if (root == NULL)
          return;
      PostOrder(root->left);
      PostOrder(root->right);
           cout<< root->data;
}
```



(iv) **Level Order Traversal:-**

- Level Order Traversal is also called as Breadth First Traversal
- In Level Order Traversal all the nodes are visited at a level are accessed before going to the next level from **left to right.**

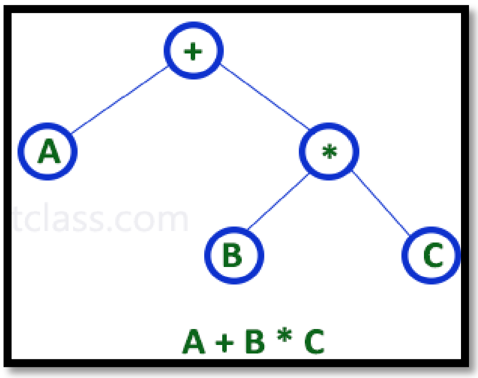


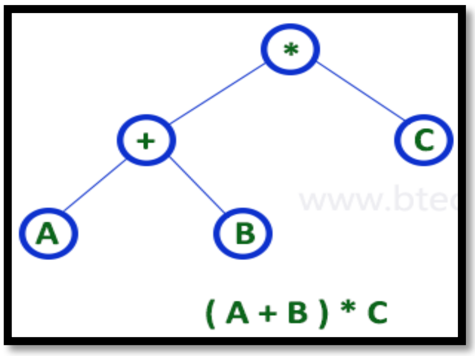


**Level Order Traversal: 1, 2, 3, 4, 5, 6, 7**

**Expression Tree:**

- The trees are used to represent an expression and those are called expression trees.
- The following expression is represented using the binary tree, where the leaves represent the operands and the internal nodes represent the operators.





**PreOrder : +A*BC**
**InOrder: A+B*C**
**PostOrder : ABC*+**

**PreOrder : *+ABC**
**InOrder: A+B*C**
**PostOrder : AB+C***

## <u>**Threaded Binary Tree:-**</u>

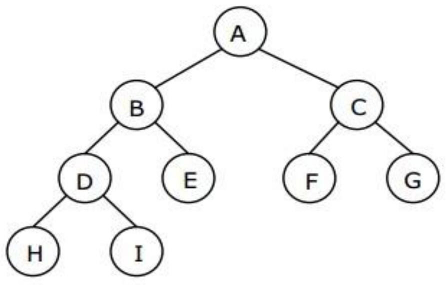
- In binary tree, the leaf nodes have no children. Therefore the left and right fields of the leaf nodes are made NULL. But NULL waste memory space so to avoid NULL in the node we will set threads.
- The number of nodes containing in the tree is less than the number of NULL pointers.

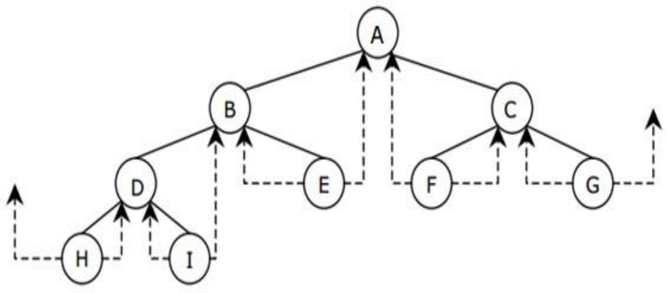
<u>**THREADS:**</u>

- The NULL pointers are replaced by a pointer to the inorder predecessor or inorder successor of a node. These special pointers are called Threads. The binary tree congaing the threads are called Threaded binary tree.
- We are representing the threads by using <u>**arrows(→).**</u>

To construct threads we use the following rules.

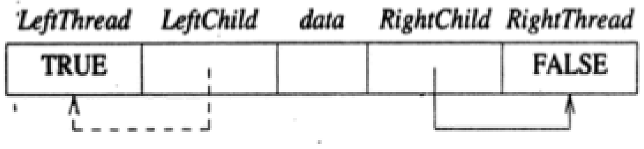
- If ptr→left is NULL, replace ptr→left with a pointer of its inorder predecessor.
- If ptr→right is NULL, replace ptr→right with a pointer of its inorder successor.





The structure of a threaded binary tree node is as follows

```
class Node
{
        bool leftThread, rightThread;
        char data;
        Node *leftChild, *rightChild;
};
```



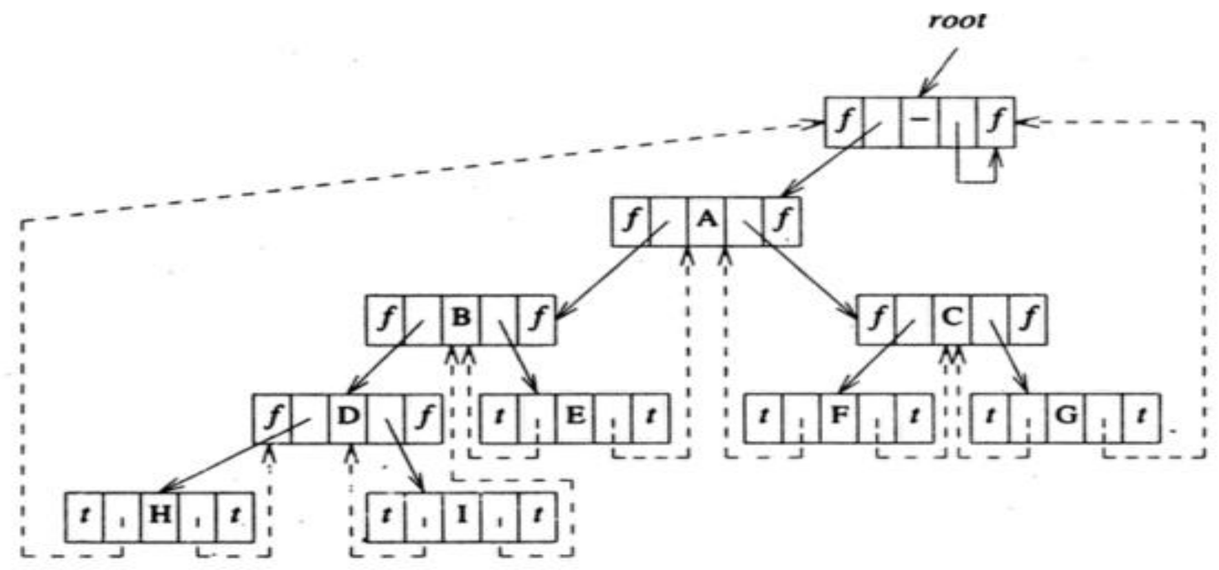
- The leftThread and rightThread values are either TRUE or FALSE
  - If leftThread is pointing to thread (inorder predecessor) then it is TRUE otherwise FALSE
  - Similarly rightThread is pointing to thread (inorder successor) then it is TRUE otherwise FALSE
- Here the nodes H and F having NULL pointer because those nodes are not having inorder predecessor and inorder successor respectively.
- In order to avoid those situations a Head node can be used. In head node left child pointing to the root node and right child is pointing to itself.

**Advantages:**

- Wastage of memory by NULL pointers is utilized by using threads.
- In traversal, the predecessor node and successor node of any node can be accessed effectively.

**Disadvantages:**

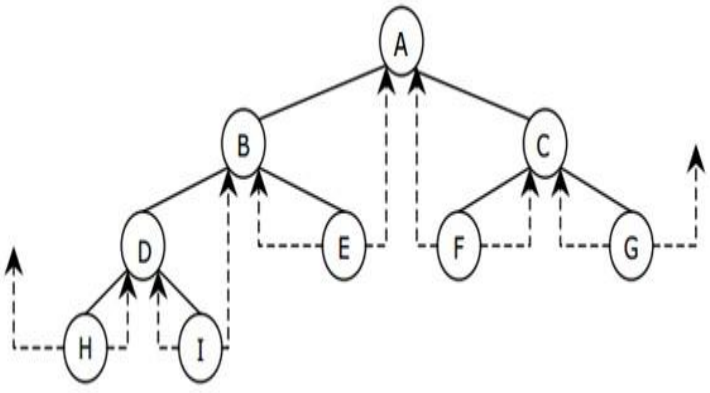
- Insertion and deletion of a node is much complex because thread link manipulations required for every insertion and deletion.



f= FALSE; t = TRUE

**INORDER TRAVERSAL OF A THREADED BINARY TREE:**

- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right
- If we follow a link to the right, we go to the leftmost node, print it, and continue



**InOrder Traversal is: H-D-I-B-E-A-F-C-G**

**INSERTION OF NODE INTO THREADED BINARY TREE:**

Inserting a new node **X as the left child** of a node:

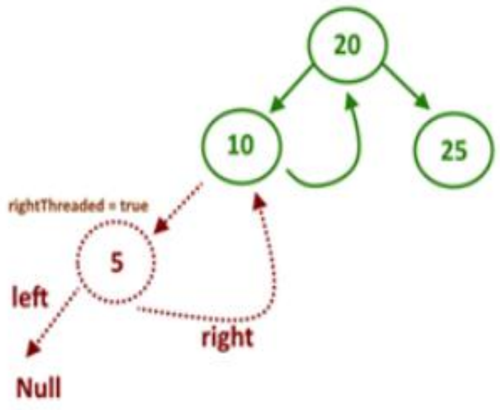
CASE 1: If a node has an empty left subtree, then the insertion is simple.

CASE 2: If the a node having non-empty left subtree, then this left child is made the left of new node after insertion.

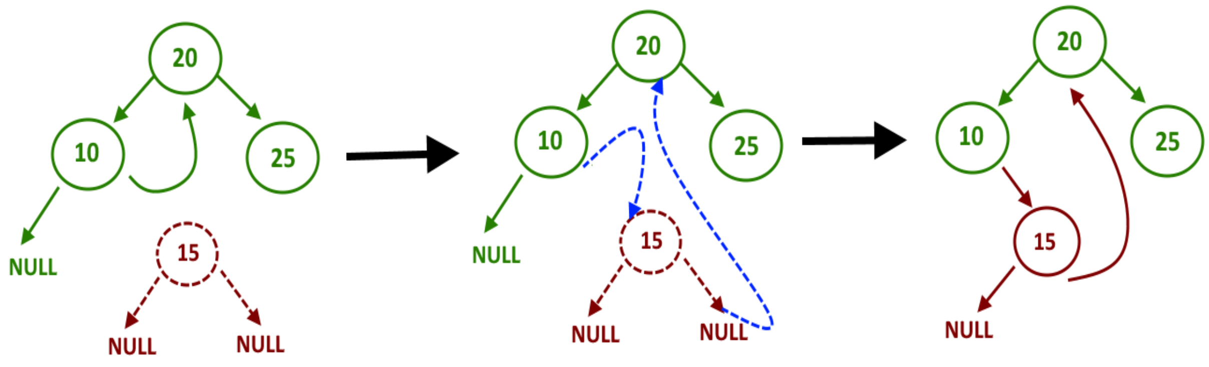
Inserting a node **X as the right child** of a node:

CASE 1: If a node has an empty right subtree, then the insertion is simple.

CASE 2: If the a node having non-empty right subtree, then this right child is made the right of new node after insertion.



Inserting node 5



Inserting Node 15 into threaded binary tree

## Priority Queue:-

- Priority Queue is a data structure having the collection of elements, which is associated with priorities.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

```
ADT Priorty_Queue
{
Instances:
        Finite collection of elements associated with some priority
        Both front and rear with in max size
Operations:
        Create();      IsEmpty();     IsFull();       enQueue(item);
        deQueue();     display();
}
```

**Basic Operations in Priority Queue:**

It allows two operations

1. Insert (enQueue)
2. DeleteMin (deQueue)

deleteMin
Priority Queue
insert

- The smaller element in priority Queue having high priority, so find the smaller element and delete it first.

**Implement priority queue:**

**Array:** A simple implementation is to use array of following structure.

- insert() operation can be implemented by adding an item at end of array in O(1) time.
- DeleteMin () operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

**Linked List**, time complexity of all operations with linked list remains same as array. The advantage with linked list is DeleteMin() can be more efficient as we don't have to move items.

## HEAP:-

A Heap tree is complete binary tree with the property of the value at each node is as larger as (as smaller as) the value of its child nodes. Heap tree also called as Binary Heap.

There are two types of heap and they are as follows...

1. Max Heap (Root node must be greater than its child nodes)
2. Min Heap (Root node must be lesser than its child nodes)

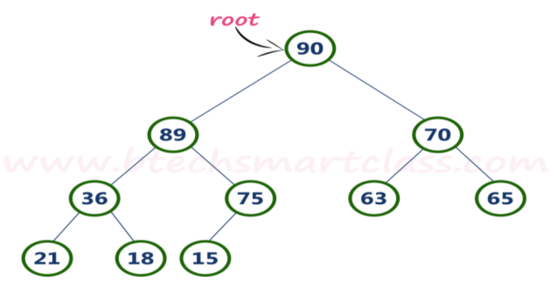
Every heap data structure has the following properties...

**Property #1 (Ordering):** Nodes must be arranged in an order according to values based on Max heap or Min heap.

**Property #2 (Structural):** All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

## Max Heap/ Max Tree:

- Max heap is a specialized complete binary tree. It is also called as max tree. A max tree is a tree in which the value of parent node is larger than the values of its children.
- The following tree is satisfying both Ordering property and Structural property according to the Max Heap data structure



**Operations on Max Heap:**

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

**Finding Maximum Value Operation in Max Heap**

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

**Insertion Operation in Max Heap:**

**Algorithm**

Step 1: Insert the newNode as last leaf from left to right.

Step 2: Compare newNode value with its Parent node.

Step 3: If newNode value is greater than its parent, then swap both of them.

Step 4: Repeat step 2 and step 3 until newNode value is less than its parent nede (or) newNode reached to root.

**Example:**

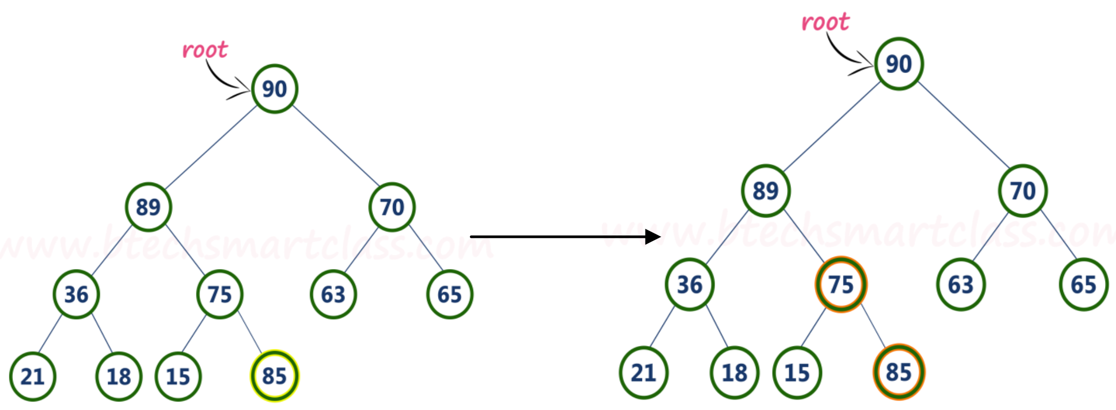
Consider the above max heap. **Insert a new node with value 85.**

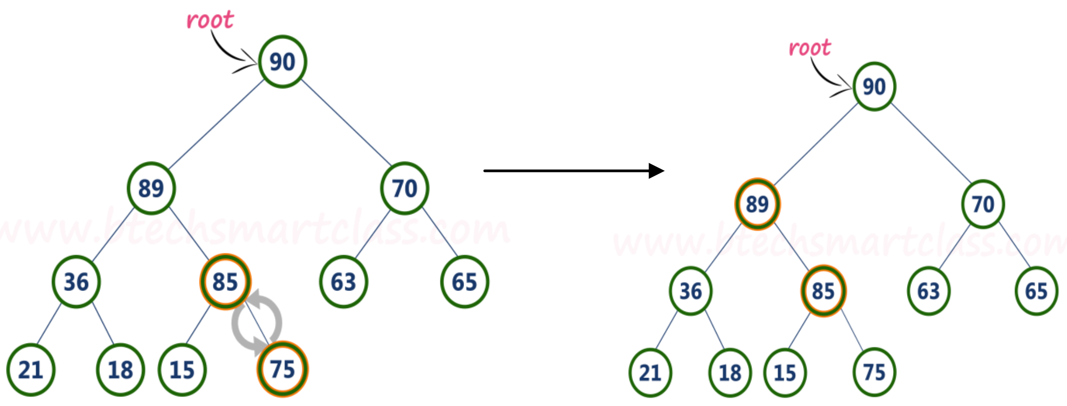
**Step 1 -** Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75.

**Step 2 -** Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**

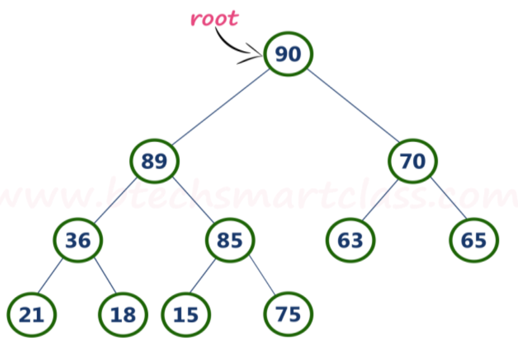
**Step 3 -** Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After swapping, max heap is...

**Step 4 -** Now, again compare newNode value (85) with its parent node value (89).





Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows...



## Deletion Operation in Max Heap

Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

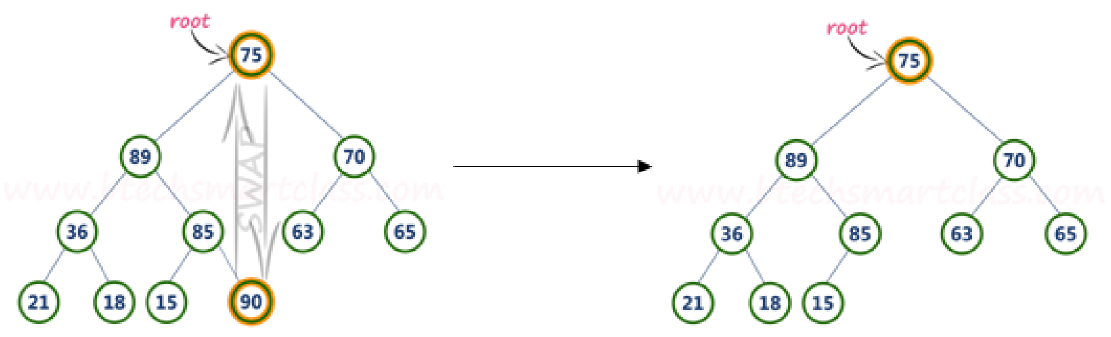
### Algorithm:

```
Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step3 − Heapify (Fix the heap):
            if the heap property holds true
                then you are done.
            else if the node value >= its parent nodes value
                then swap them, and repeat step 3.
            else
            swap the node with the largest child node, and  repeat step 3.
```

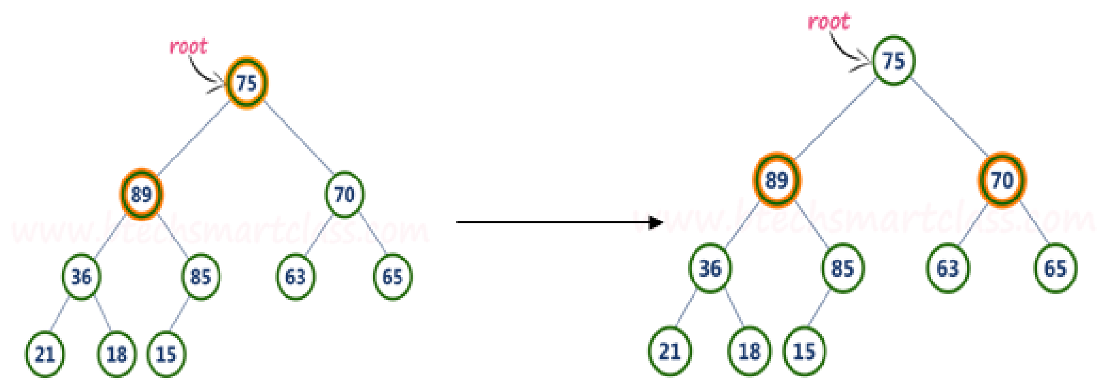
**Example**

Consider the above max heap. **Delete root node (90) from the max heap.**

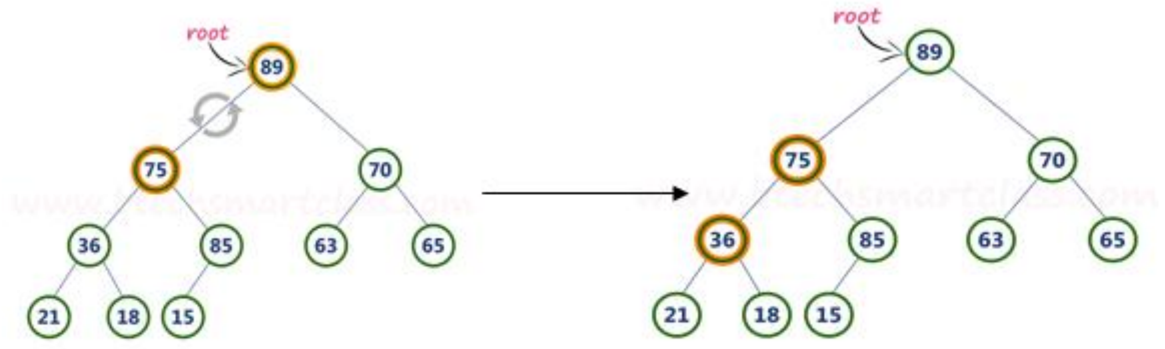
**Step 1 - Swap** the **root node (90)** with **last node 75** in max heap. After swapping max heap is



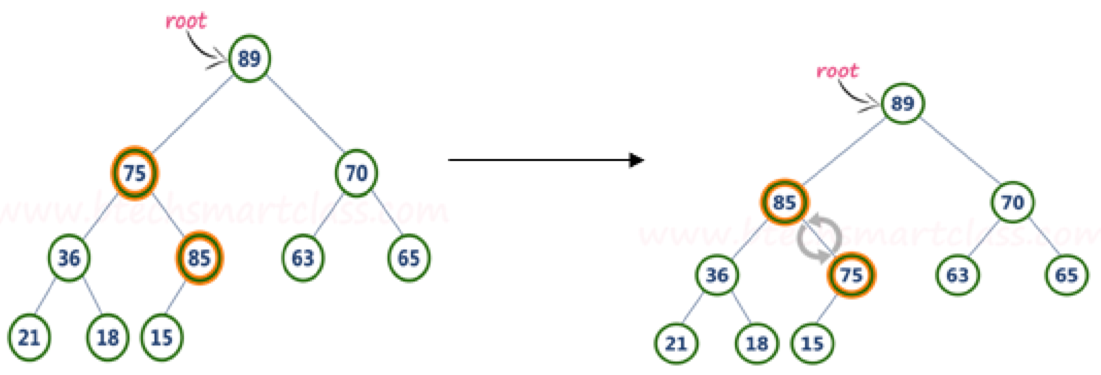
- **Step 2 - Delete** last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...
- **Step 3 -** Compare **root node (75)** with its **left child (89)**. Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



- **Step 4 -** Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.

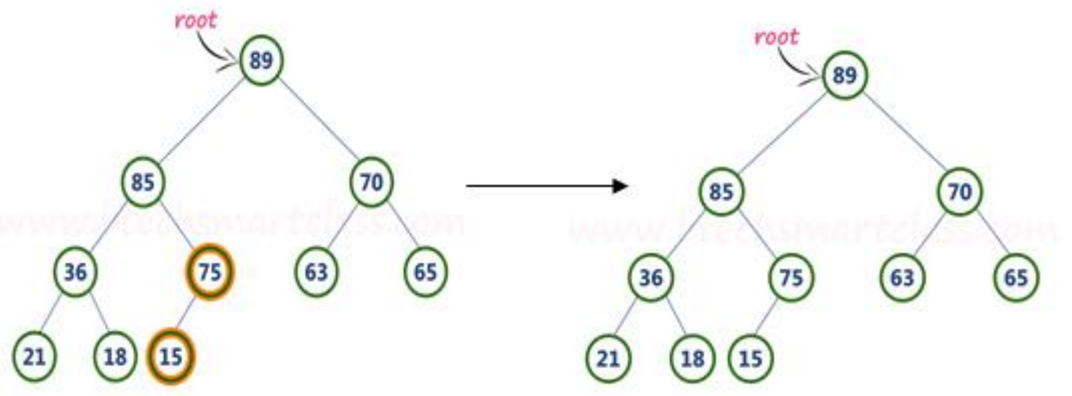


- **Step 5 -** Now, again compare **75** with its **left child (36)**. Here, node with value **75** is larger than its left child. So, we compare node **75** with its right child **85**.
- **Step 6 -** Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...



- **Step 7 -** Now, compare node with value **75** with its left child (**15**). Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.

  Finally, max heap after deleting root node (**90**) is as follows...

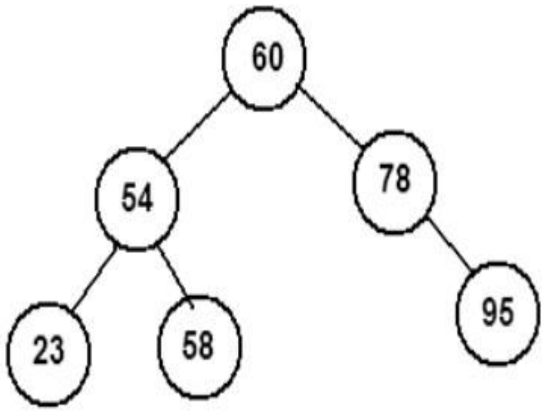


## Binary Search Tree:-

A Binary tree is said to be Binary Search Tree it should satisfies the following

1. Left Sub-tree value lesser than the root node
2. Right Sub-tree value greater than or equal to the root node
3. Both left sub-tree and right sub-tree are also recursively satisfies these properties and itself a binary search tree.

- Binary search tree is also called Ordered Binary Tree.
- The reason why we go for a Binary Search tree is to improve the searching efficiency. The average case time complexity of the search operation in a binary search tree is O( log n ).

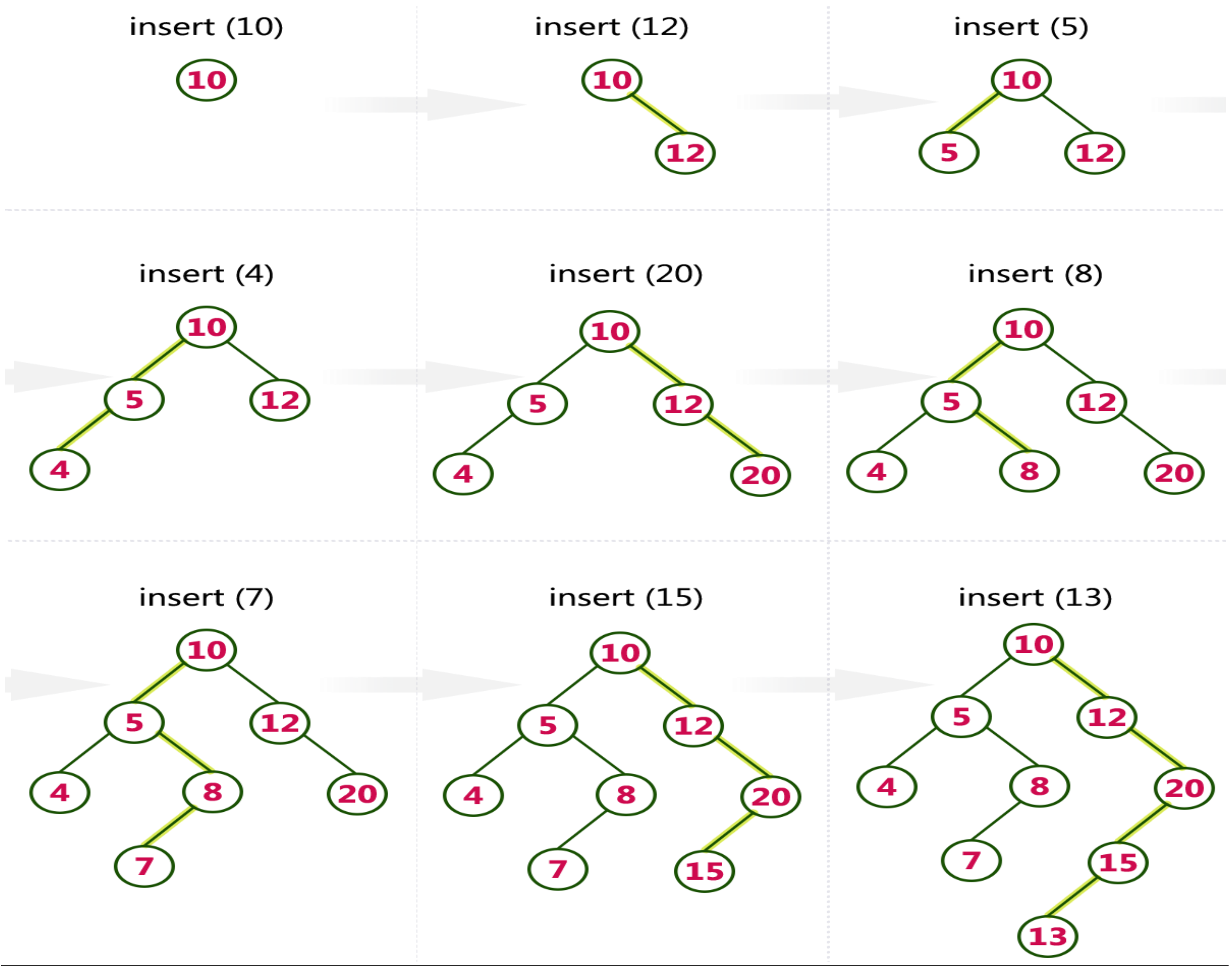


## Operations on Binary Search Tree

The following operations can be performed on BST.

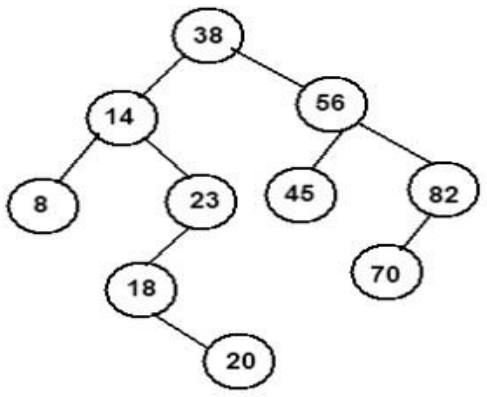
1. Create
2. Search
3. Insertion
4. Deletion

### Creation of Binary Search Tree:



Consider the following list of numbers. A binary search tree can be constructed using this list of numbers, as shown.

38, 14, 8, 23, 18, 20, 56, 45, 82, 70



- Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38.
- Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less than 38 move towards left of 38. Now 8 is compared with 14, and as it is less than 14 and also 14 does not have any child, 8 is attached as the left child of 14.

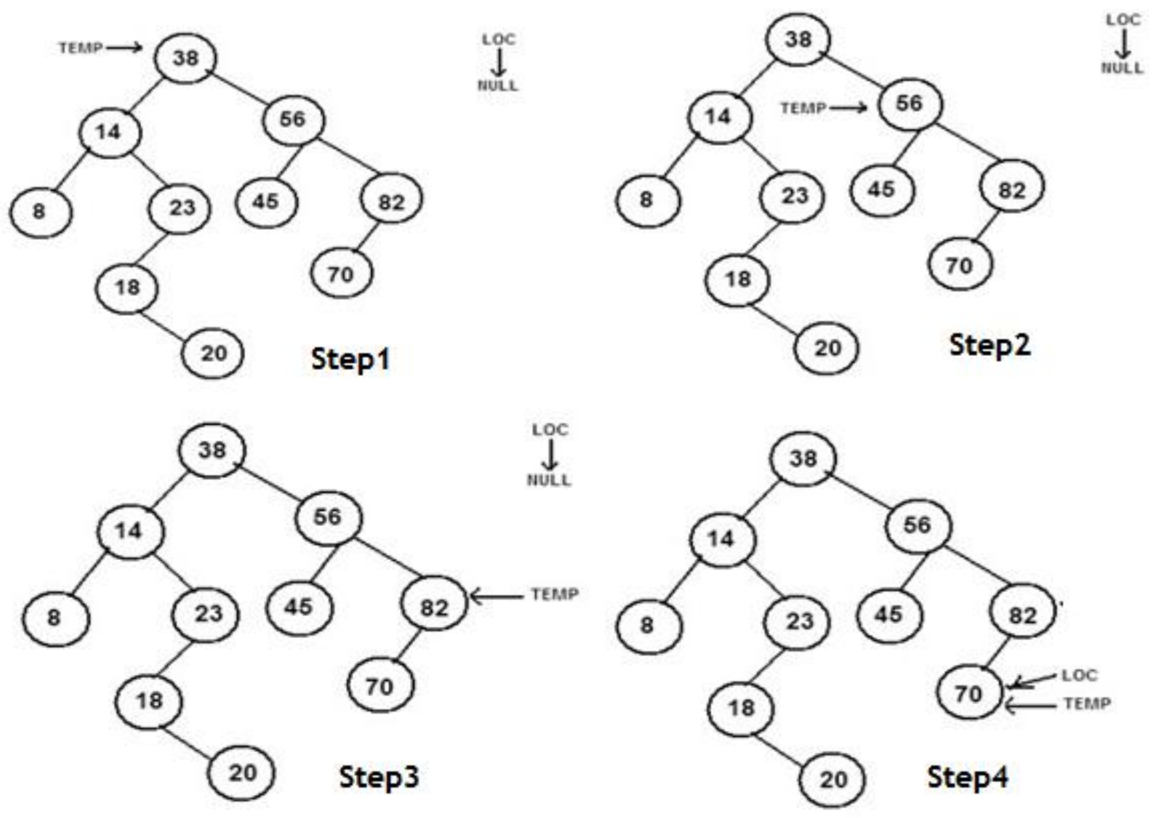
- This process is repeated until all the numbers are inserted into the tree. Remember that if a number to be inserted is greater than a particular node element, then we move towards the right of the node and start comparing again.

## Search Operation In Binary Search Tree:

- The search operation on a BST returns the address of the node where the element is found. The pointer LOC is used to store the address of the node where the element is found.
- Initially the pointer TEMP is made to point to the root node.

**Process:**

- Let us search for a value 70 in the following BST. Let KEY = 70.
- The KEY value is compared with 38. As KEY is greater that 38, move TEMP to the right child of 38, i.e., 56.
- KEY is greater than 56 and hence we move TEMP to the right child of 56, which is 82.
- Now since KEY is lesser than 82, TEMP is moved to the left child of 82.
- The KEY value matches here and hence the address of this node is stored in the pointer LOC.



**Algorithm SEARCH( ROOT, KEY )**

```
temp = ROOT, loc = NULL
while temp ≠ NULL
        If KEY = = temp → data ;
                loc = temp ;
                break;
        If KEY <  temp → data
                temp = temp → left
        else
                temp=temp → right
```

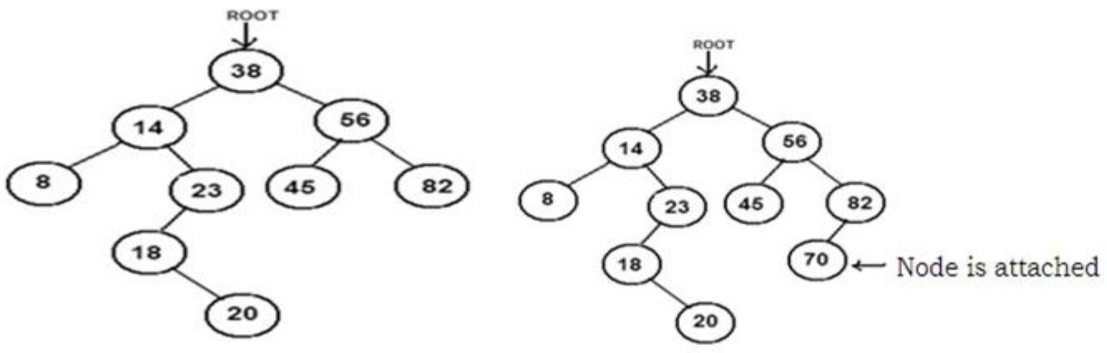
## Insert Into In A Binary Search Tree:

The BST itself is constructed using the insert operation described below. Consider the following list of numbers. A binary tree can be constructed using this list of numbers.

38, 14, 8, 23, 18, 20, 56, 45, 82.

For example we want to insert the element is 70. While inserting a node into the binary search tree first we have find the appropriate position in the binary search tree. We start comparing the node value 70 with the root if it is greater than the root then it is inserted on the right branch of the root else on the left branch of the root.

Now compare the node 70 with root node 38. As node 70 is greater than the root 38 we will move to the right subtree. Now compare node 70 with the node 56 as it greater then move to right and compare node 70 with node 82 as it less than the node 82 we attach 70 as left child of node 82. The diagram is shown below.



**Algorithm INSERT( ROOT, item )**

1. Read the value for the node which is to be created and store it into a node called Cur.
2. Initially if(root!=NULL) then root = Cur
3. Again read the next value of node created in Cur
4. If (Cur → data <  root → data) then attach the Cur node as a left child of root otherwise attach the Cur node as a right child of  root node.
5. Repeat step3 and step4 for constructing required binary search tree completely.

## Deletion From A Binary Search Tree:

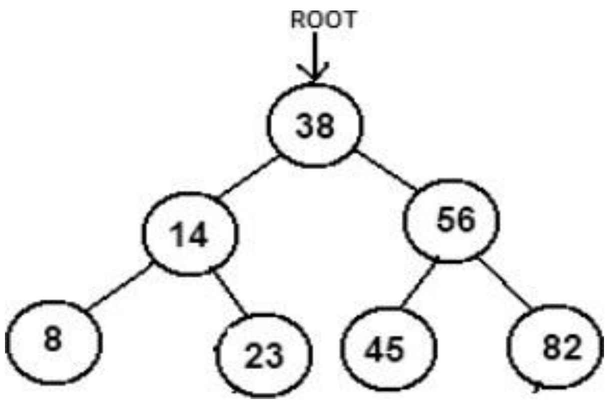
The deletion of a node from a binary search tree occurs with three possibilities

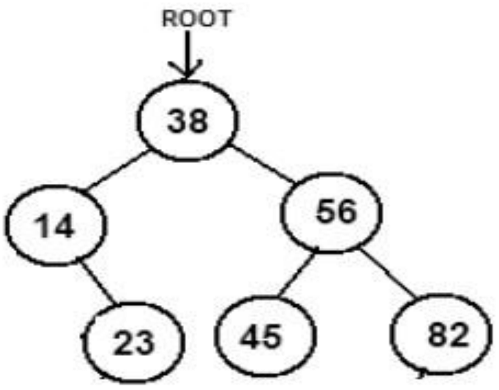
1. Deletion of a leaf node.
2. Deletion of a node having one child.
3. Deletion of a node having two children.

**1. Deletion of a leaf node**

This is the simplest deletion in which we can simple remove it from the tree. For example consider the binary search tree.

From the above tree diagram the node we want to delete is the node 8(temp), then we will set the left pointer of its parent (node 14) to NULL. Then after deletion the binary search tree is as follows.





**Algorithm**

```
if(temp→left = = NULL && temp→right = = NULL)
    if(parent → left = = temp)
            parent → left = NULL
      else
             parent → right = NULL
```

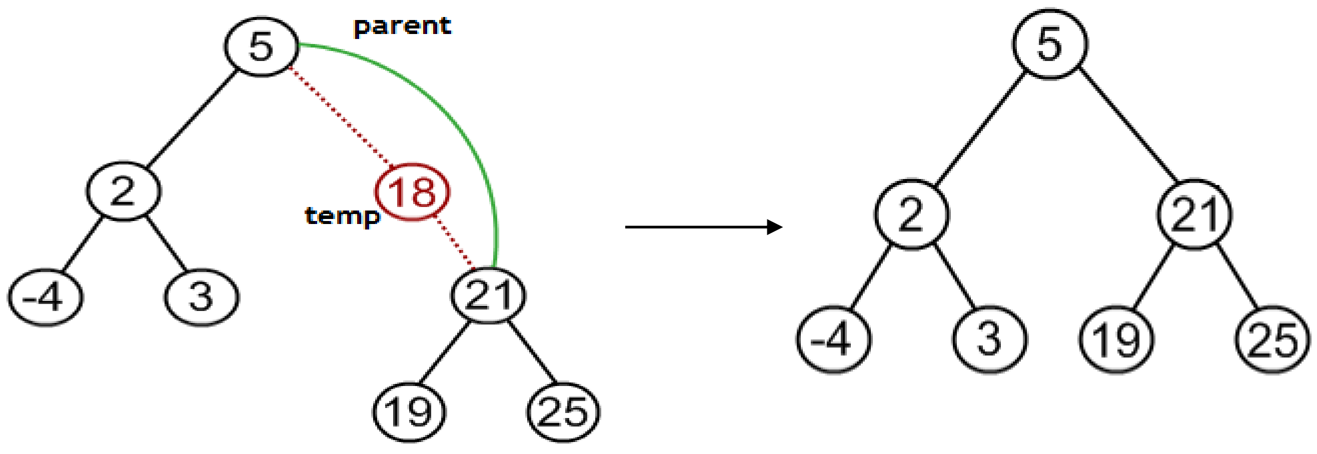
**2. Deletion of a node having one child**

The node if we want to delete is having only one child ( i.e. either left or right child), delete it and replace it with its child. From the diagram the node we want to delete is having the value 18 then we simple copy node 21 at the place of 18 and set the node free.

**Algorithm**

```
if(temp → left !=NULL && temp → right == NULL)
     if(parent → left ==temp)
          parent → left = temp → left
     else
           parent → right = temp → left
     delete temp
```

```
if(temp → left = =NULL && temp → right != NULL)
    if(parent → left = =temp)
        parent →left = temp → right
    else
        parent → right = temp → right
    delete temp
```



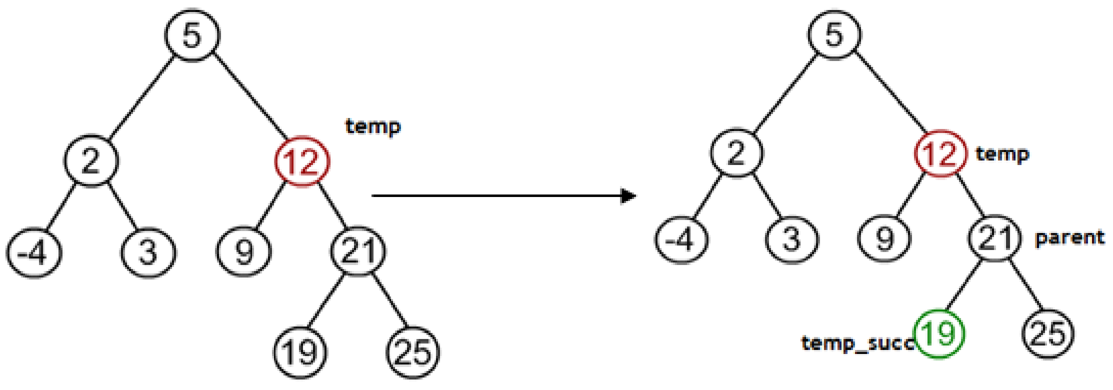
**3. Deletion of a node having two children**

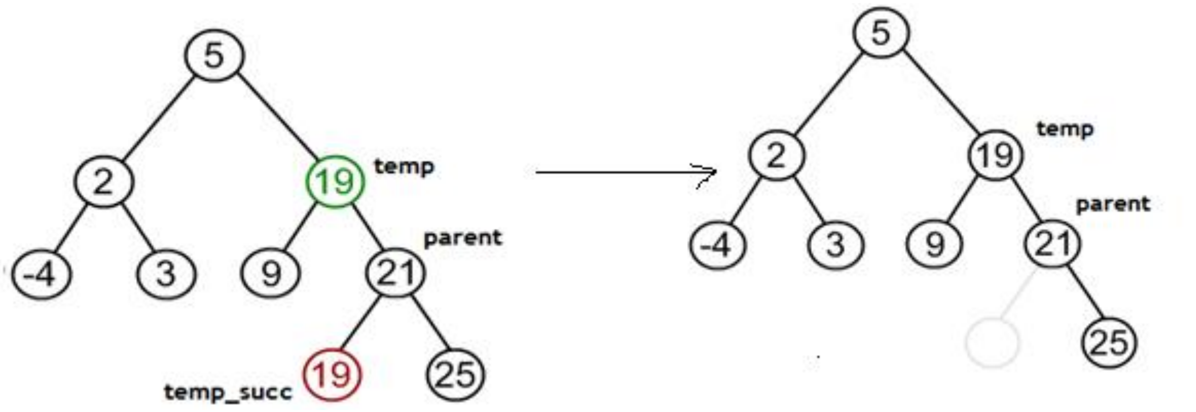
Suppose the node to be deleted is called N, We replace the value of N with either its inorder successor (the left-most child of the right subtree) or the inorder precedessor(the right-most child of the the left subtree).

The node if we want to delete is having two children. From the diagram the node we want to delete is having the value 12 then we find the inorder successor of the node 12 is 19 and it is copied at the place of 12 and set the node 21 left pointer to right of 19.

**Algorithm**

```
if(temp - > left != NULL && temp - > right != NULL)
    parent = temp
    temp_succ = temp - > right
    while(temp_succ - > left != NULL)
        parent =  temp_succ
        temp_succ = temp_succ - > left
    temp - > data = temp_succ - > data
    parent - > left = temp_succ->right
    delete temp_succ
```





**HEIGHT OF A BINARY SEARCH TREE**

- The height of a binary search tree with “n” elements can become as large as “n”.
- For instance, when the values like 1, 2, . . n are inserted into the empty binary search tree.
- If insertions and deletions are made at random then the height of the binary search tree is O(log n) on average.
- Search trees with worst case height of O(log n) are called balanced search trees. These trees permit insertions, deletions and searches to be performed at time O(h). for example, AVL trees, Red / Black Trees, B-Trees, 2 – 3 Trees etc.

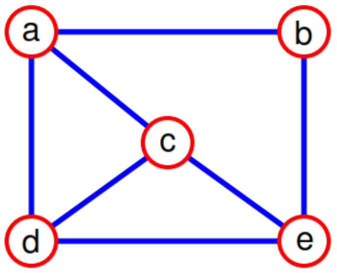
**Balanced Tree** : A balanced tree is a tree where both left child and right child having same number of nodes.

*********

**SYLLABUS:** **UNIT-V**

**Graph:** Terminology, Graph Representation, Abstract Data Type, **Elementary Graph Operation**: Depth First Search, Breadth First Search, Connected Components, Bi-connected Components, Spanning Trees, **Minimum Cost Spanning Trees**: Kruskal S Algorithm, Prim s Algorithm, Sollin' s Algorithm, **Shortest Paths and Transitive Closure**: Single Source/All Destination Nonnegative Edge Cost, Single Source/All Destination: General Weights, All-Pairs Shortest Path, Transitive Closure.

## Graph:



A graph is defined as **G = (V,E):**

V: set of vertices

E: set of edges connecting the vertices in V,

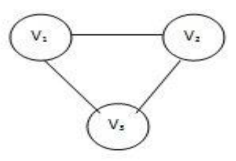
An edge E = (u,v) is a pair of vertices.

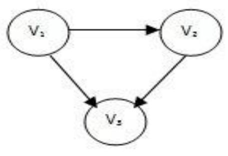
V= {a,b,c,d,e}

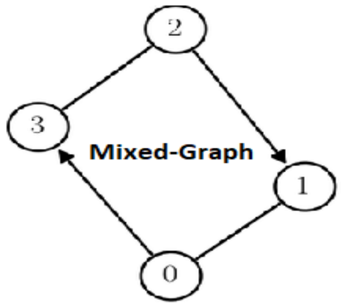
E={(a,b),(a,c),(a,d),(b,e),(c,d),(c,e),(d,e)}

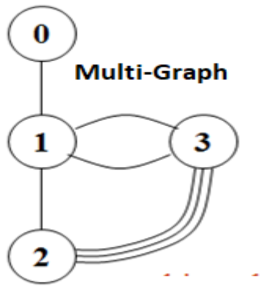
## Graph Terminology:-

- **Undirected Graph**: An Undirected graph is a graph, in which all the edges have no direction. The edge (u,v) is identical to (v,u).
- **Directed Graph:** A directed graph (di-graph) is a graph, in which all the edges have direction.
- **Mixed Graph:** A mixed graph is a graph, in which some of the edges having the direction and some of the edge are not having the direction.
- **Multi-Graph:** In a multi-graph, there can be more than one edge from vertex P to vertex Q. In a simple graph there is at most one.

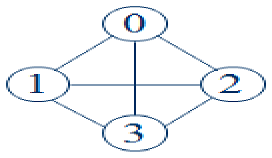

Undirected Graph

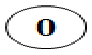

Directed Graph

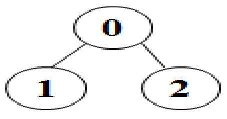


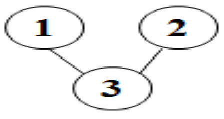


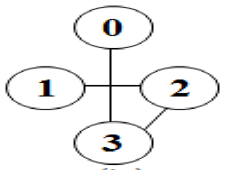
- **Sub-Graph:** A sub-graph of G is G', it consists V(G') is a subset of V(G) and E(G') is a subset of E(G)


$G_1$


(i)

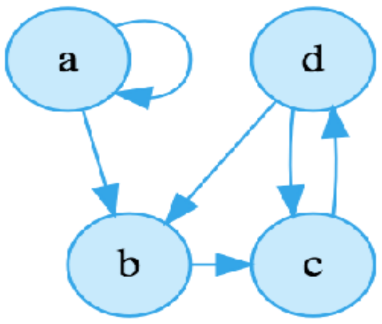

(ii)


(iii)

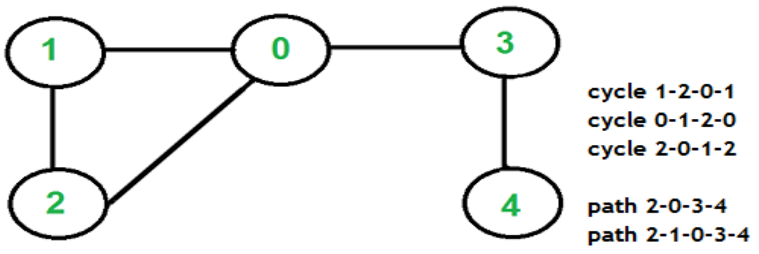

(iv)

Some of the subgraph of $G_1$

- **Adjacency**: Let an edge E is having in between pair of vertices (u,v) then u and v are adjacent to each other.
    - (v0, v1) is an edge in an undirected graph, v0 and v1 are adjacent
    - (v0, v1) is an edge in an directed graph, v0 is adjacent to v1, and v1 is adjacent from v0
- **Path:** It is sequence of vertices; every vertex is adjacent to the next vertex.
- **Cycle:** It is path containing minimum of three vertices, the stating vertex and last vertex must be same.
- **Loop (Self-Edge):** A self loop is an edge that connects a vertex to itself.

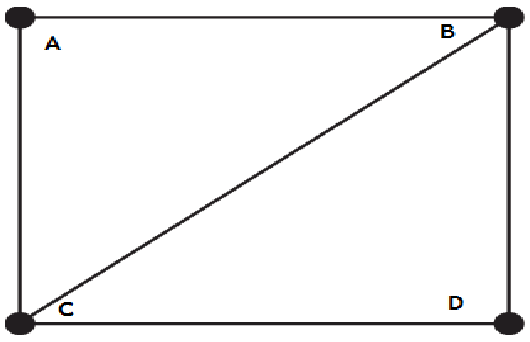


Self Loop at Vertex a

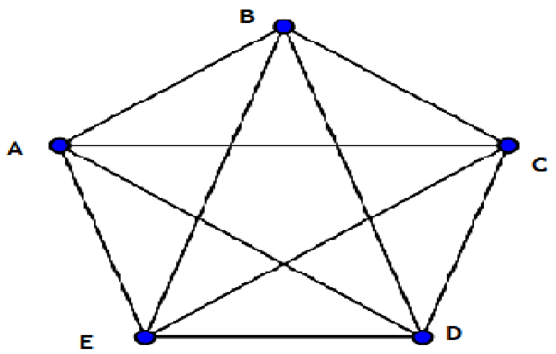


- **Connected Graph**: A graph is said to be connected, if there exist a path between any two vertices u and v otherwise the graph is disjoint or disconnected graph.

  - The total number of edges in connected graph is **n-1**
- **Complete Graph**: A graph is said to complete graph if there exist an edge from every vertex to every other vertex.
  - The total number of edges in undirected complete graph is **n(n-1)/2**
  - The total number of edges in directed complete graph is **n(n-1)**



Connected Graph



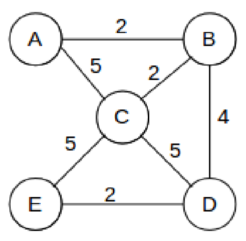
Complete Graph

- **Weighted Graph**: A graph is said to be weighted graph, if every edge in the graph is assigned by some weights or values or lengths.
- **Degree:** The degree of a vertex is the number of edges incident to that vertex

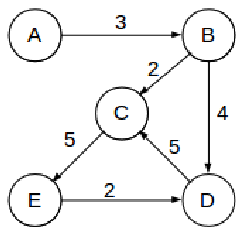
  For directed graph,

  – The in-degree of a vertex v is the numbers of edges are terminated (head).

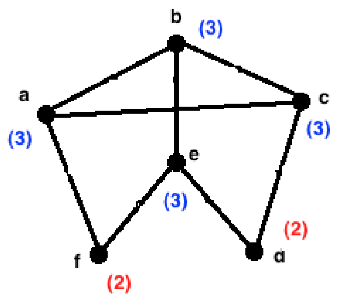
  – The out-degree of a vertex v is the numbers of edges are started (tail)



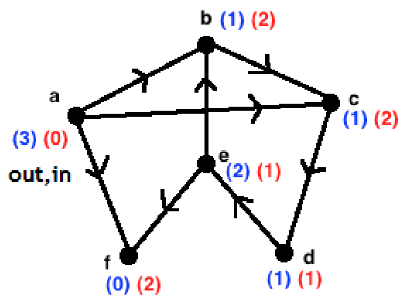
Undirected



Directed



Un-Directed Graph



Directed Graph

## Graph ADT:-

ADT Graph

{

**Instances:** a nonempty set of vertices and a set of edges, where each edge is in between a pair of vertices.

**Operations:**

```
Create();                 //return an empty graph
InsertVertex(V);          // insert V in to the graph
InsertEdge(u,v);          // insert new edge between u and v
DeleteVertex(V);          // delete V from the graph
DeleteEdge(u,v);          // delete an edge from graph between u and v
Bool IsEmpty(graph); //return TRUE else return FALSE
Adjacent(V);              // return a list of all vertices that are adjacent to V
int degree(V);            // returns degree of a vertex V
```

}

## Graph Representations:

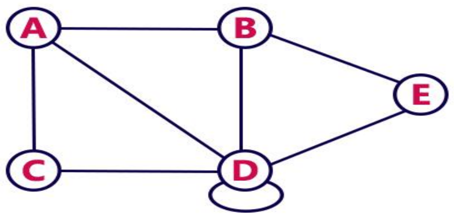
Graph data structure is represented using following representations...

1. Adjacency Matrix (sequential or Array Representation)
2. Adjacency List (Linked List Representation)

### 1. Adjacency Matrix:

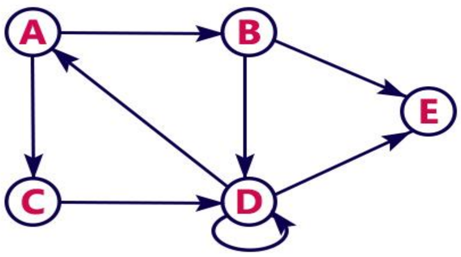
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. The elements are filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

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



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

- The sum of all values in a row is equivalent to the out degree of a vertex.
- The sum of all values in a column is equivalent to the in degree of a vertex.

**Advantages:**

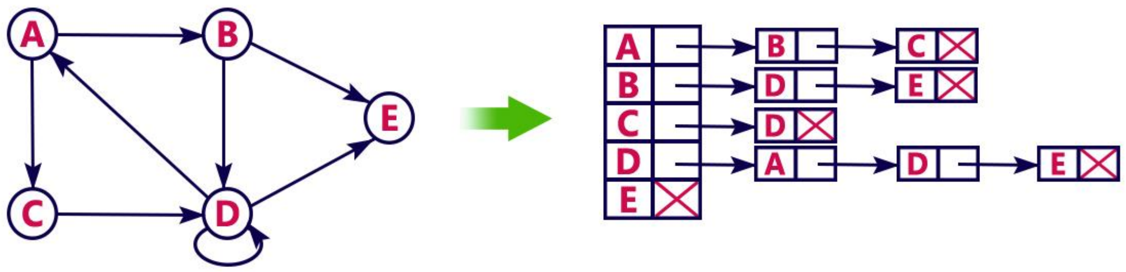
- Easy to store and manipulate matrices.
- Finding the cycles and path in a graph is easy.

**Disadvantage:** The space requires for a graph having N vertices is NXN, but the matrix contain only few elements.

## 2. Adjacency List:

- In this representation, every vertex of a graph contains list of its adjacent vertices.
- The structure contain list of all the vertices, every vertex having its own link to its own list, the list contain all adjacent vertices.

For example, consider the following graph representation implemented using linked list...



**Advantage:** It is clearly showing what the adjacent vertices for a vertex are.

## Elementary Operations:

The following are some graph operations:

a) Traversals

– Depth First Search (DFS) preorder tree traversal

– Breadth First Search (BFS) level order tree traversal

b) Spanning Trees

c) Connected Components, BiConnected Components

## Graph Traversals:

Traversals mean visiting all the vertices in a graph at once in systematic way.

There are two types of graph traversals, they are

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

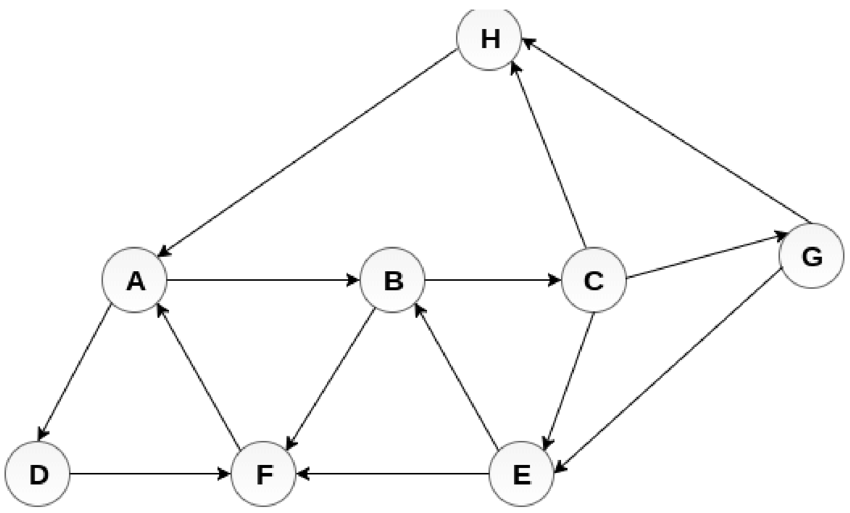
## Depth First Search (DFS):-

– In DFS we are using the application of STACK.
– In graphs there is no root node or start vertex, so we can take any arbitrary vertex as starting vertex.

The process/Algorithm of DFS is

i. Push the start vertex in to STACK
ii. Repeat process until STACK is empty
   a. Pop the vertex at top of the STACK
   b. Process the vertex
   c. Push the adjacent vertices in to the STACK for popped vertex.

**Example**: Consider the graph G along with its adjacency list, given in the figure below.



**Adjacency Lists**

A : B, D

B : C, F

C : E, G, H

G : E, H

E : B, F

F : A

D : F

H : A

## Solution:

**Let the starting vertex of the graph is H**.

1. Push H onto the stack

   STACK : H

2. POP the top element of the stack i.e. H, print it and push all the adjacent vertices of H onto the stack that are is ready state.

   Print H
   STACK : A

3. Pop the top element of the stack i.e. A, print it and push all the adjacent vertices of A onto the stack that are in ready state.

   Print A
   Stack : B, D

4. Pop the top element of the stack i.e. D, print it and push all the adjacent vertices of D onto the stack that are in ready state.

   Print D
   Stack : B, F

5. Pop the top element of the stack i.e. F, print it and push all the adjacent vertices of F onto the stack that are in ready state.

   Print F
   Stack : B

6. Pop the top of the stack i.e. B and push all the adjacent vertices

   Print B
   Stack : C

7. Pop the top of the stack i.e. C and push all the adjacent vertices.

   Print C
   Stack : E, G

8. Pop the top of the stack i.e. G and push all its adjacent vertices.

   Print G
   Stack : E

9. Pop the top of the stack i.e. E and push all its adjacent vertices.

Print E
Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph using DFS is: H → A → D → F → B → C → G → E

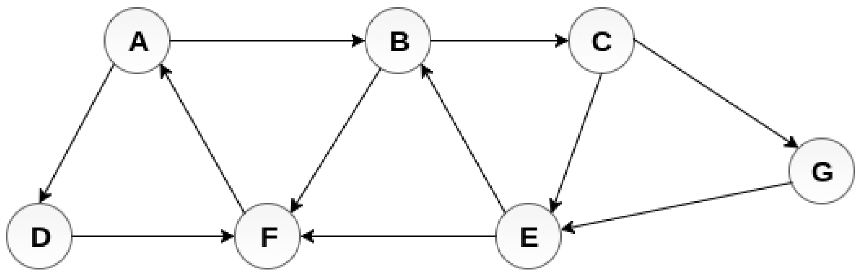
## Breadth First Search (BFS):-

- In BFS we are using the application of QUEUE.
- In graphs there is no root node or start vertex, so we can take any arbitrary vertex as starting vertex.

The process/Algorithm of BFS is

i. Enqueue the start vertex in to QUEUE
ii. Repeat process until QUEUE is empty
   a. Dequeue the vertex at front of the QUEUE
   b. Process the vertex
   c. Enqueue the adjacent vertices in to the QUEUE at rear end, for dequeue vertex.

**Example**: Consider the graph G along with its adjacency list, given in the figure below.



**Adjacency Lists**

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

**Solution:** Lets start vertex of the graph is Node A.

1. Insert node A into QUEUE

QUEUE = {A}

2. Delete the Node A from QUEUE and insert all its adjacent vertices.

QUEUE = {B, D}
Print  A

3. Delete the node B from QUEUE and insert all its adjacent vertices.

QUEUE = {D, C, F}
Print  B

4. Delete the node D from QUEUE and insert all its adjacent vertices. Since F is the only neighbour of it which has been inserted, we will not insert it again.

QUEUE = {C, F}
Print D

5. Delete the node C from QUEUE and insert all its adjacent vertices.

QUEUE = {F, E, G}
Print C

6. Remove F from QUEUE and add all its adjacent vertices. Since all of its adjacent vertices has already been added, we will not add them again.

QUEUE = {E, G}
Print F

7. Remove E from QUEUE, all of E's adjacent vertices has already been added to QUEUE therefore we will not add them again.

QUEUE = {G}
Print E

8. Remove G from QUEUE, the adjacent vertices has already been added to QUEUE therefore we will not add them again.

QUEUE = { }
Print G

Hence, the QUEUE is now becomes empty and all the nodes of the graph have been traversed.

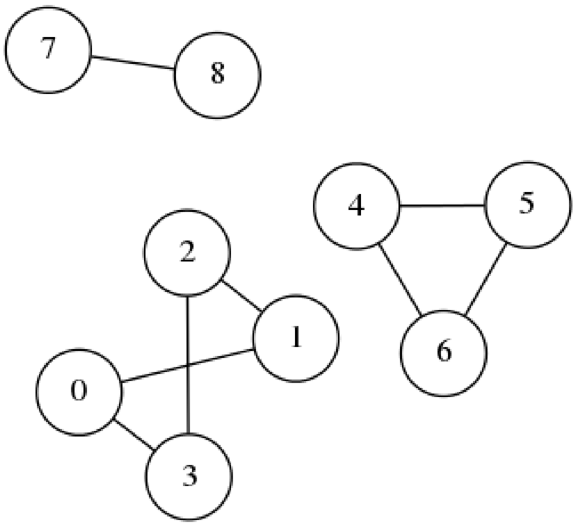
The printing sequence of the graph using BFS is: A → B → D → C → F → E → G

## Applications of DFS:

- Undirected Graph
  - Connected Components
  - Articulation Points
  - Bi-Connected Components
- Directed Graph
  - Cyclic or Acyclic Graph

**Connected Component**:

Connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.



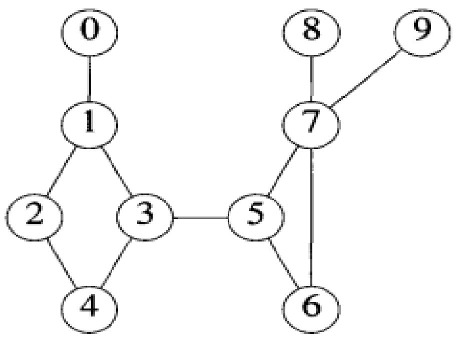
3 Connected Components

**Articulation point**: An Articulation point in a connected graph is a vertex that, if delete, would break the graph into two or more pieces (disjoint graphs).

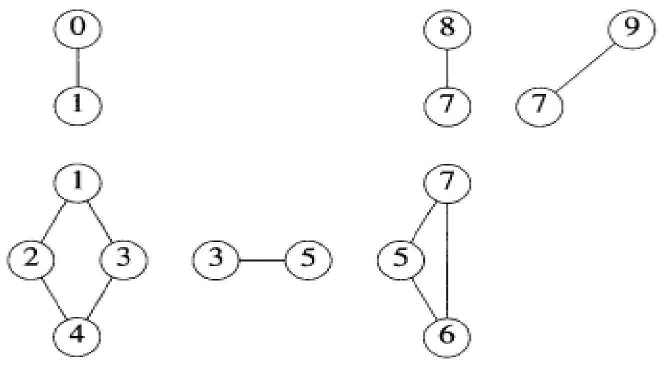
**Biconnected graph**: A graph with no articulation point is called biconnected graph. In other words, a graph is biconnected if and only if any vertex is deleted, the graph remains connected.

**Biconnected component:**

- A biconnected component of a graph is a maximal biconnected subgraph, a biconnected subgraph that is not contained in a larger biconnected subgraph.
- A graph that is not biconnected can divide into biconnected components, sets of nodes mutually accessible via two distinct paths.



(a) Connected graph



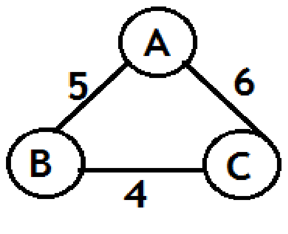
(b) Biconnected components

**Spanning Tree:-**

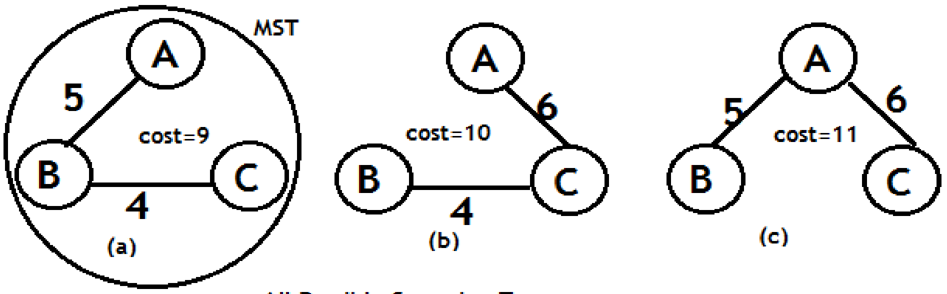
- A spanning tree of a graph is a subgraph in includes all of the vertices of a graph, without any cycles.
- A graph may have one or more number of possible spanning trees.
- If a graph containing N vertices then the total numbers of possible spanning trees are **$N^{N-2}$** and every spanning tree is having maximum number of edges will be **N-1.**

## Minimum Cost Spanning Tree:

- In weighted connected graph, **the minimum cost spanning tree** is a spanning tree that has minimum weights than all other spanning trees.
- Weight or cost of spanning tree is sum of weights of all edges in the spanning tree.



Weighted Connected Graph



All Possible Spanning Trees

There are three popular techniques for finding the minimum cost spanning tree for any graph.

1. Kruskal's Algorithm
2. Prim's Algorithm
3. Sollin's Algorithm

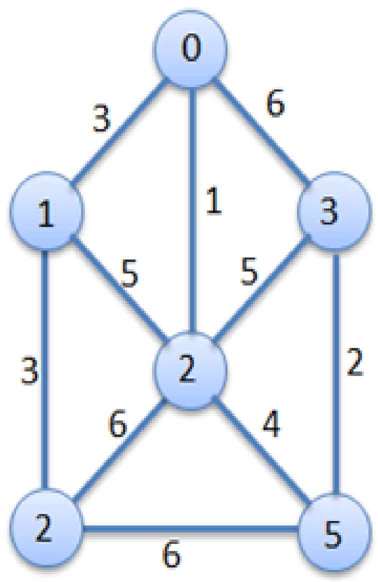
## 1. Kruskal's Algorithm:

- Kruskal's Algorithm follows greedy method.
- In Greedy algorithm, **first off all we check all possibilities of a given problem, then at the first stage we select that which can give optimal solution.**
- Every time picking the minimum weighted edge from the graph and draw the edge between vertices in minimum cost spanning tree and it shouldn't form any cycles.
- The same process is repeated until all the vertices are connected then we will find the minimum cost spanning tree for the given graph.

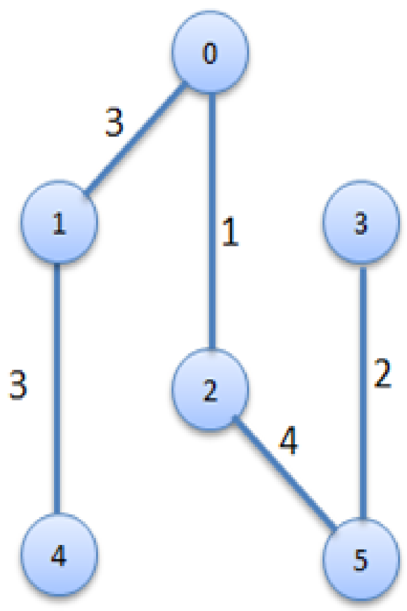
Algorithm;

1. Sort all Edges of weights in ascending order
2. Initially the MST is empty
3. Repeat until the graph containing N-1 number of edges
   a. Pick an edge and insert into MST, if it is not forming the cycle then consider the edge.
   b. Otherwise remove the edge from the MST.
4. Return MST.

Example:



A Simple Weighted Graph



Minimum-Cost Spanning Tree

**Example:**

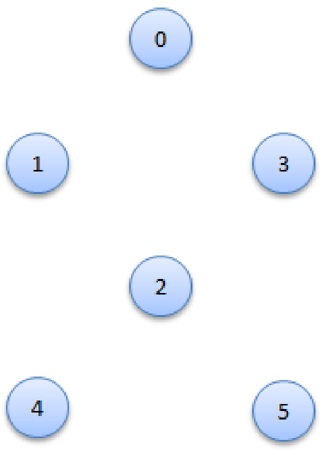
Procedure for finding Minimum Spanning Tree

**Step1.** Edges are sorted in ascending order by weight.

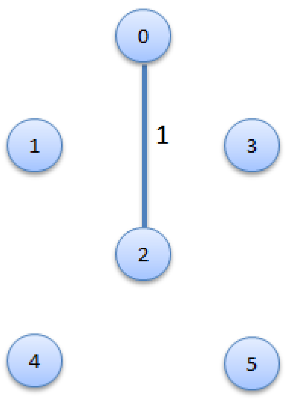
| **Edge No.** | **Vertex Pair** | **Edge Weight** |
|---|---|---|
| E1 | (0,2) | 1 |
| E2 | (3,5) | 2 |
| E3 | (0,1) | 3 |
| E4 | (1,4) | 3 |
| E5 | (2,5) | 4 |
| E6 | (1,2) | 5 |
| E7 | (2,3) | 5 |
| E8 | (0,3) | 6 |
| E9 | (2,4) | 6 |
| E10 | (4,5) | 6 |

**Step2.** Edges are added in sequence.

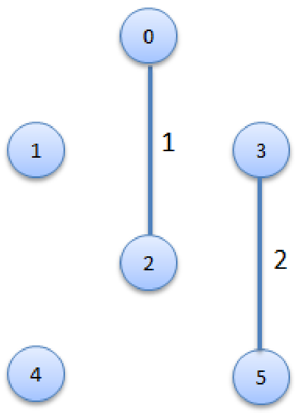
Graph



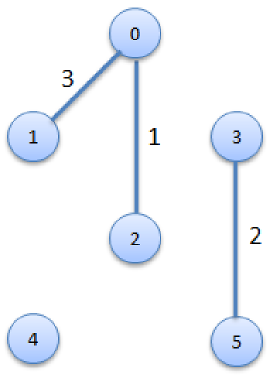
Add Edge E1



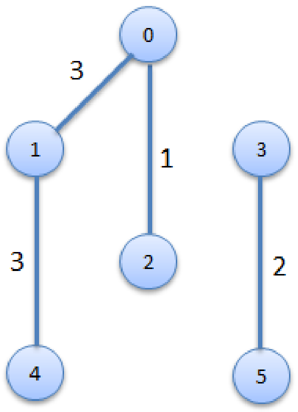
Add Edge E2



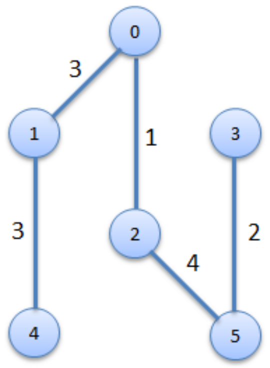
Add Edge E3



Add Edge E4



Add Edge E5



**Minimum Cost** = 1+2+3+3+4 = 13

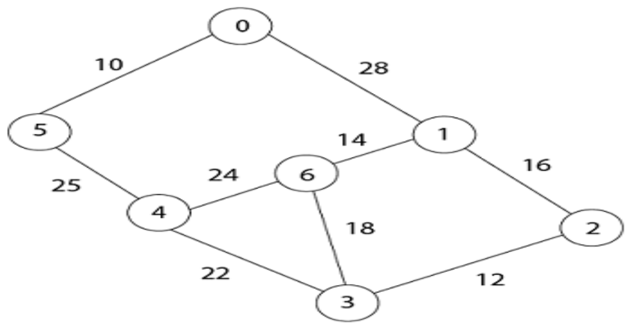
## 2. Prim’s Algorithm:

– Prim’s Algorithm is used to find the minimum cost spanning tree.
– It is a greedy algorithm. It starts with an empty spanning tree MST. At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

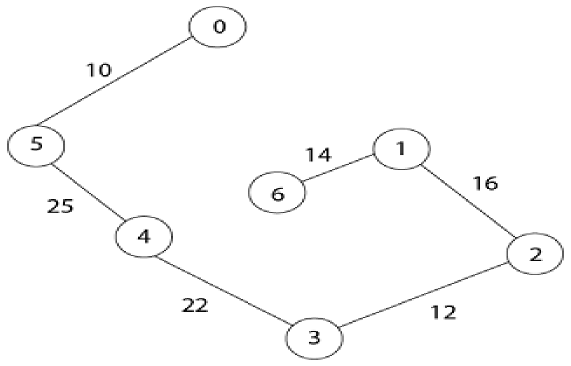
Algorithm:

1. Create MST set that keeps track of vertices and included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the start vertex so that it is picked first.
3. While MST set doesn't include all vertices.
    a. Pick vertex **u** which is not is MST set and has minimum key value. Include 'u' to MST set.
    b. Update the key value of all adjacent vertices of **u**. To update, iterate through all adjacent vertices. For every adjacent vertex **v**, if the weight of edge (u,v) less than the previous key value then update key value as a weight of new edge.

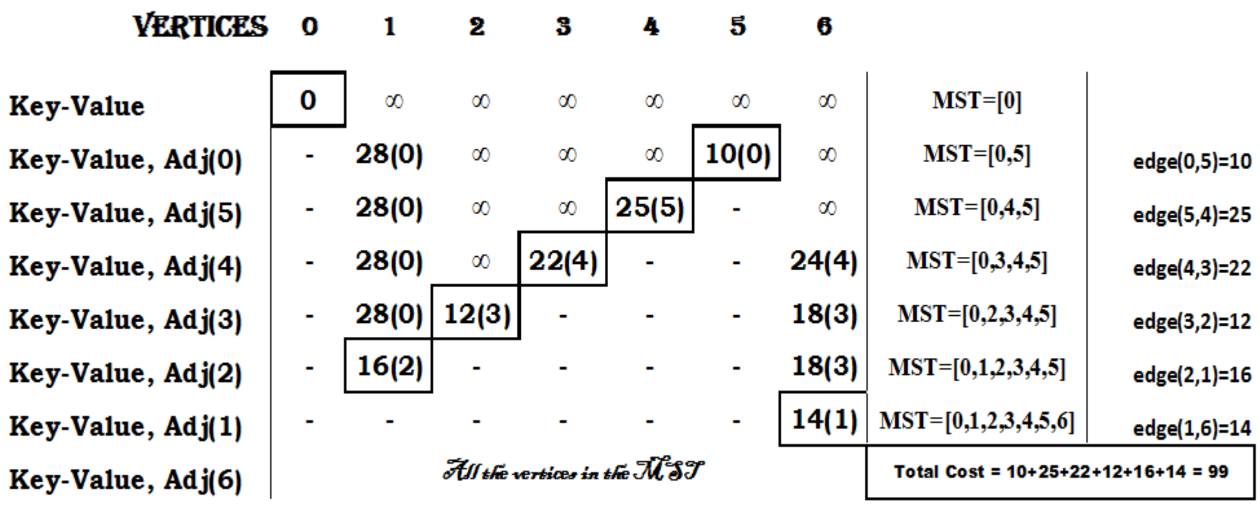
**Example:**



Weighted Connected Graph

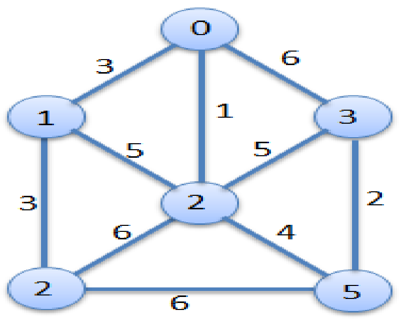


Minimum Cost Spanning Tree

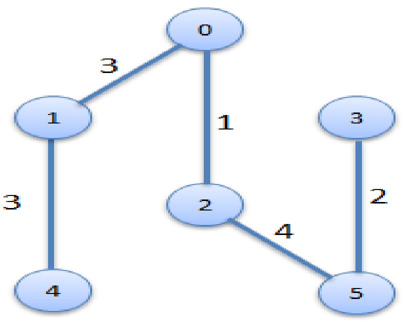


| VERTICES | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | |
|---|---|---|---|---|---|---|---|---|---|
| Key-Value | $\boxed{0}$ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | MST=[0] | |
| Key-Value, Adj(0) | - | 28(0) | ∞ | ∞ | ∞ | $\boxed{10(0)}$ | ∞ | MST=[0,5] | edge(0,5)=10 |
| Key-Value, Adj(5) | - | 28(0) | ∞ | ∞ | $\boxed{25(5)}$ | - | ∞ | MST=[0,4,5] | edge(5,4)=25 |
| Key-Value, Adj(4) | - | 28(0) | ∞ | $\boxed{22(4)}$ | - | - | 24(4) | MST=[0,3,4,5] | edge(4,3)=22 |
| Key-Value, Adj(3) | - | 28(0) | $\boxed{12(3)}$ | - | - | - | 18(3) | MST=[0,2,3,4,5] | edge(3,2)=12 |
| Key-Value, Adj(2) | - | $\boxed{16(2)}$ | - | - | - | - | 18(3) | MST=[0,1,2,3,4,5] | edge(2,1)=16 |
| Key-Value, Adj(1) | - | - | - | - | - | - | $\boxed{14(1)}$ | MST=[0,1,2,3,4,5,6] | edge(1,6)=14 |
| Key-Value, Adj(6) | All the vertices in the MST | | | | | | | Total Cost = 10+25+22+12+16+14 = 99 | |

**Example2:**



A Simple Weighted Graph

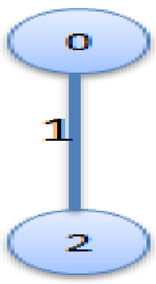


Minimum-Cost Spanning Tree

Procedure for finding Minimum Spanning Tree

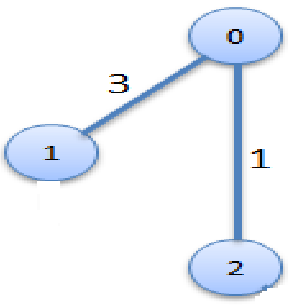
**Step1**

| No. of Nodes | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Distance | - | 3 | **1** | 6 | ∞ | ∞ |
| Distance From | | 0 | 0 | 0 | | |



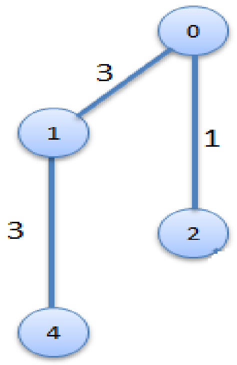
**Step2**

| No. of Nodes | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Distance | - | **3** | - | 5 | 6 | 4 |
| Distance From | | 0 | | 2 | 2 | 2 |



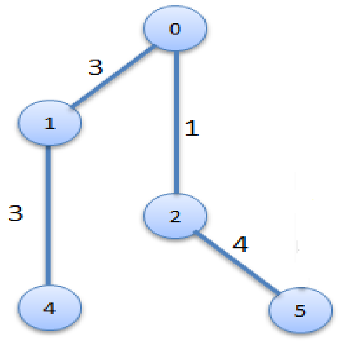
**Step3**

| No. of Nodes | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Distance | - | - | - | 5 | **<u>3</u>** | 4 |
| Distance From | | | | 2 | 1 | 2 |



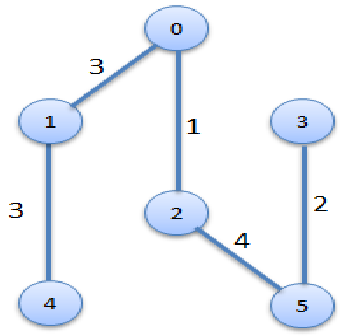
**Step4**

| No. of Nodes | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Distance | - | - | - | 5 | - | **<u>4</u>** |
| Distance From | | | | 2 | | 2 |



**Step5**

| No. of Nodes | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Distance | - | - | - | **<u>3</u>** | - | - |
| Distance From | | | | 2 | | |



**Minimum Cost** = 1+2+3+3+4 = 13

## <u>3. Sollin's Algorithm:</u>

- Sollin's Algorithm is used to find the minimum cost spanning tree (MST).
- It was the first algorithm to find the MST.
- It follows Greedy Algorithm similar to Kruskal's and Prim's Algorithm.

Process (Algorithm):

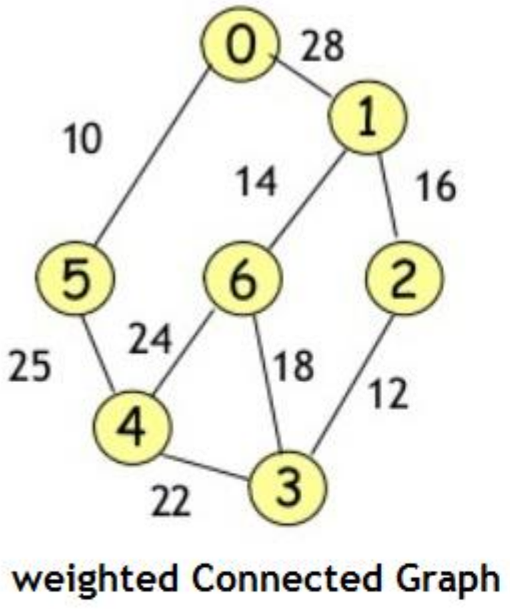
Step1: Each vertex to be a component, denote the graph T

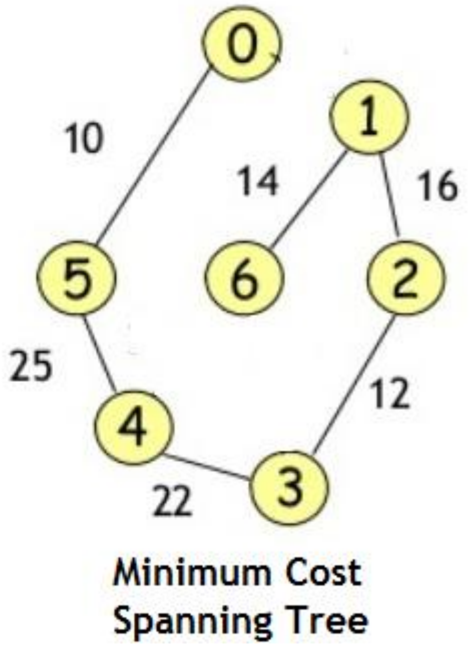
Step2: For each component C in T, find the cheapest edge connecting to another component

Step3: If this edge is not in T, add it to T, Repeat **step2 until all the components are connected.**

.

**Example:**



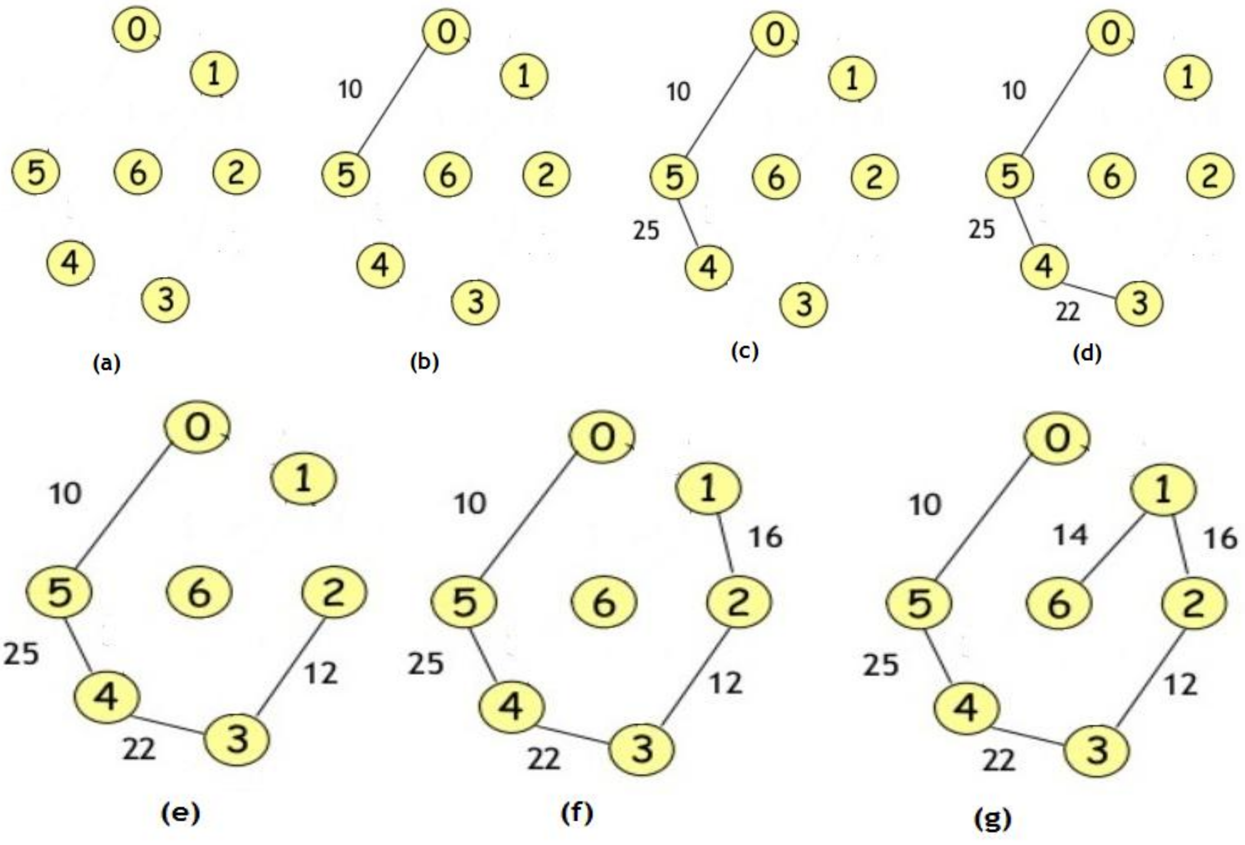


**Step1:** Write all the components in the given graph.

**Step2:** Choose the starting vertex as 0, choose cheapest edge as 10 and it not forming any cycles. Select vertex 5.

**Step3:** At vertex 5, choose cheapest edge as 25 and it not forming any cycles. Select vertex 4.

**Step4:** At vertex 4, choose cheapest edge as 22 and it not forming any cycles. Select vertex 3.



**Minimum Cost** = 10+25+22+12+16+14 = 99

**Step5:** At vertex 3, choose cheapest edge as 12 and it not forming any cycles. Select vertex 2.

**Step6:** At vertex 2, choose cheapest edge as 16 and it not forming any cycles. Select vertex 1.

**Step7:** At vertex 1, choose cheapest edge as 14 and it not forming any cycles. Select vertex 6.

**Step8**: All the vertices are in MST. So stop the procedure.

## Single Source Shortest Path:

### Dijkstra's Algorithm:

– Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph $G = (V, E)$, where all the edges are non-negative (i.e., $w(u, v) \geq 0$ for each edge ).

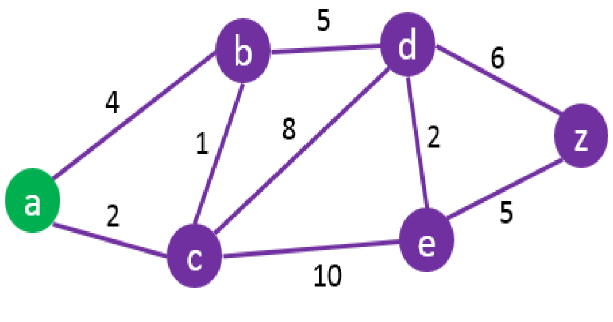
– Dijkstra's algorithm is follow greedy algorithm that solves the shortest path problem for a directed graph G. Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights.

Algorithm:

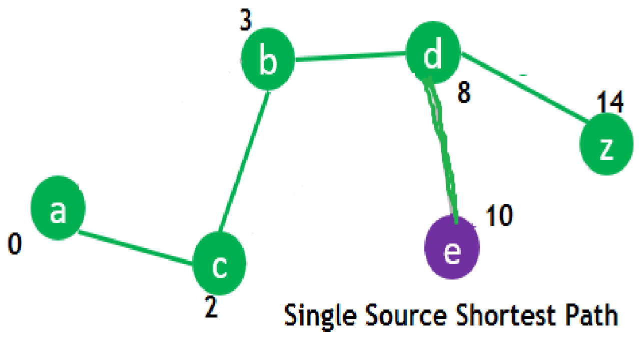
1. Start with the empty Shortest Path Tree (SPT), keep track to vertices and included into SPT.
2. Assign a distance value to all the vertices, initialize all the distances with $+\infty$ (Infinity) except the source vertex. Distance of source vertex to source vertex will be 0.
3. Repeat the following steps until all vertices are added to SPT.
   a) Pick the vertex ***u*** which is not in ***SPT[ ]*** and has minimum distance.
   b) Add vertex ***u*** to ***SPT[].***
   c) Update the key value of all adjacent vertices of **u**. To update, iterate through all adjacent vertices. For every adjacent vertex **v**, if the distance from source vertex to adjacent vertex (u,v) less than the previous key value then update key value of new distance.

```
For adjacent vertex v, if v is not in SPT[] and
distance[v] > distance[u] + edge u-v weight
        update distance[v] = distance[u] + edge u-v weight
```

**Example:**



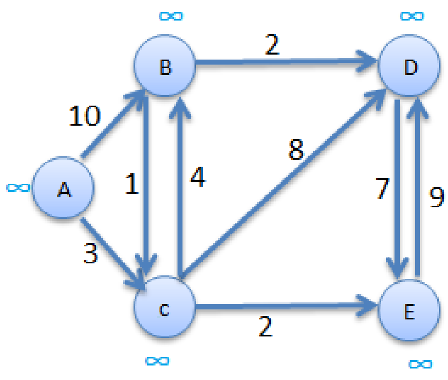
Weighted Connected Graph



Single Source Shortest Path

|  | VERTICES | a | b | c | d | e | z |  |
|---|---|---|---|---|---|---|---|---|
|  | Key-Value | $\boxed{0}$ | ∞ | ∞ | ∞ | ∞ | ∞ | SPT=[a] |
| d[a]=0 | Key-Value, Adj(a) | - | 4(a) | $\boxed{2(a)}$ | ∞ | ∞ | ∞ | SPT=[a,c] |
| d[c]=2 | Key-Value, Adj(c) | - | $\boxed{2+1(c)}$ | - | 2+8(c) | 2+10(c) | ∞ | SPT=[a,b,c] |
| d[b]=3 | Key-Value, Adj(b) | - | - | - | $\boxed{3+5(b)}$ | 2+10(c) | ∞ | SPT=[a,b,c,d] |
| d[d]=8 | Key-Value, Adj(d) | - | - | - | - | $\boxed{8+2(d)}$ | 8+6(d) | SPT=[a,b,c,d,e] |
| d[e]=10 | Key-Value, Adj(e) | - | - | - | - | - | $\boxed{8+6(d)}$ | SPT=[a,b,c,d,e,z] |
| d[z]=14 | Key-Value, Adj(z) |  |  |  | All the vertices in the MST |  |  | SPT=[a,b,c,d,e,z] |

Example

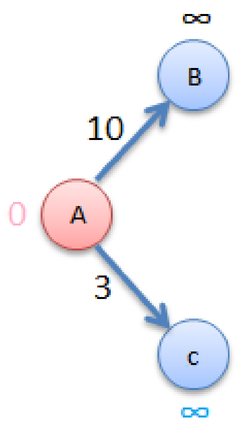


Procedure for Dijkstra's Algorithm

**Step1**

Consider A as source vertex

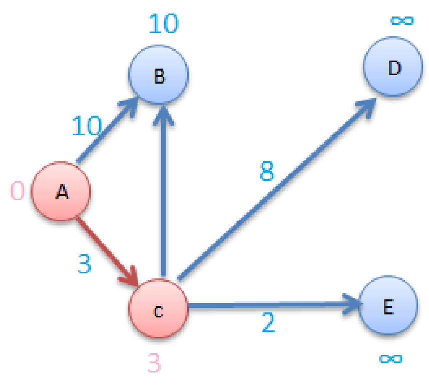
| No. of Nodes | A | B | C | D | E |
|---|---|---|---|---|---|
| Distance | **<u>0</u>** | ∞ | ∞ | ∞ | ∞ |
| Distance From |  |  |  |  |  |



**Step2**

Consider A as source vertex

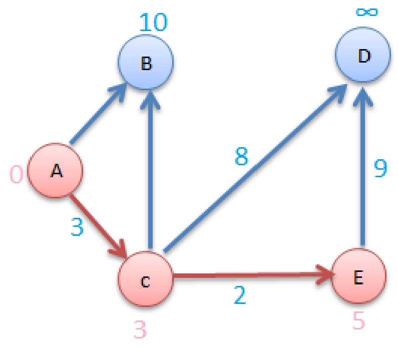
| No. of Nodes | A | B | C | D | E |
|---|---|---|---|---|---|
| Distance | - | 10 | **<u>3</u>** | ∞ | ∞ |
| Distance From |  | A | A |  |  |



**Step3**

Now consider vertex C

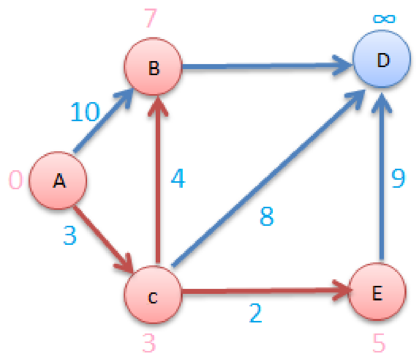
| No. of Nodes | A | B | C | D | E |
|---|---|---|---|---|---|
| Distance | - | 7 | - | 11 | <u>**5**</u> |
| Distance From | | C | | C | C |



**Step4**

Now consider vertex E

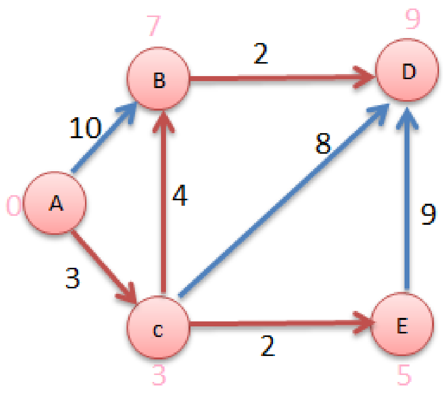
| No. of Nodes | A | B | C | D | E |
|---|---|---|---|---|---|
| Distance | - | <u>**7**</u> | - | 11 | - |
| Distance From | | C | | C | |



**Step5**

Now consider vertex B

| No. of Nodes | A | B | C | D | E |
|---|---|---|---|---|---|
| Distance | - | - | - | <u>**9**</u> | - |
| Distance From | | | | B | |



**Step5**

Now consider vertex D, All the vertices are in SPT

| No. of Nodes | A | B | C | D | E |
|---|---|---|---|---|---|
| Distance | - | - | - | - | - |
| Distance From | | | | | |

Thus we get all shortest path vertex as

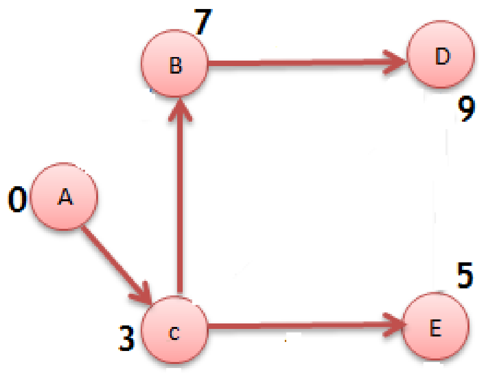
Weight from A to C is 3

Weight from A to E is 5 (A-C-E)

Weight from A to B is 7 (A-C-B)

Weight from A to D is 9 (A-C-B-D)

These are the shortest distance from the source's' in the given graph.



## All-Pair Shortest Path:

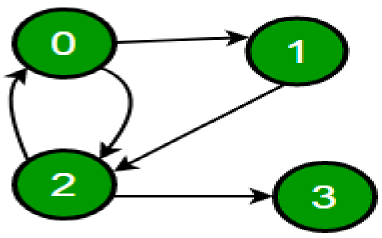
### Transitive Closure:

- The Transitive closure of directed or undirected graph G is, directed graph that has all the vertices and an edge (u,v) iff there is a path from u to v in the graph.
- Transitive closure of a matrix $A^+$ is

  $A^+$ [i][j] = 1, if there is a path of length>0 from I to j

  $A^+$ [i][j] = 0, otherwise

Example:



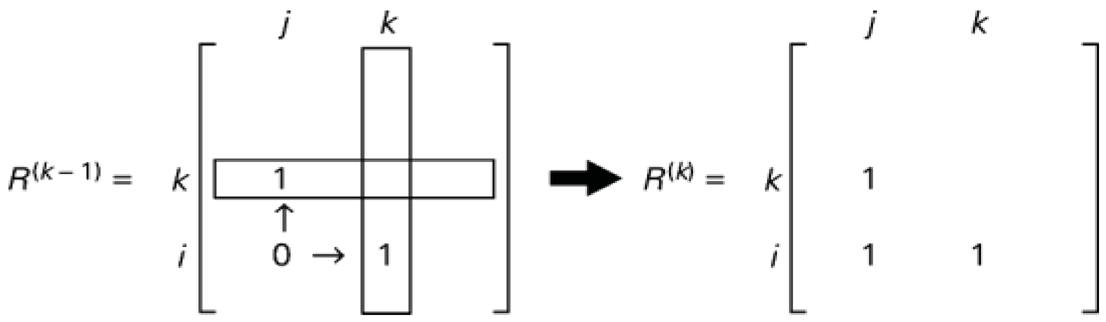
$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

### All-Pairs Shortest Paths:

- It aims to figure out the shortest path from each vertex U to every other V.
- All pairs-shortest path problem is aim to find just the distance from each vertex to each another vertex in the graph.

### Warshall Algorithm:

- This algorithm construct the transitive closure of given digraph through series of matrices $R^0$, $R^1$, $R^2$, $R^3$,….., $R^n$ .
- The order of all series of matrices is nXn, where is the number of vertices in the given digraph.
- $R^0$ is equivalent to Adjacency matrix and the final matrix $R^n$ is the Transitive closure matrix.
- In matrix $R^k$ the element at ith row and jth column will be '1', if there exist a path of positive length from vertex i to vertex j with intermediate nodes of k, otherwise '0'.



- Initially we are finding $R^0$ matrix, in that number of intermediate nodes are equal to zero. So, it is called adjacency matrix.

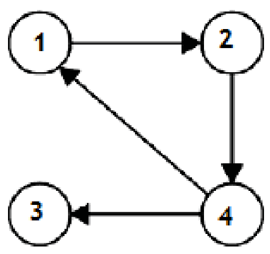
$$R^0 \rightarrow R^1 \rightarrow R^2 \rightarrow R^3 \rightarrow \ldots \rightarrow R^n$$

- In each matrix Rk, the elements can be found by using the formula

$$R^k[i,j] = \{ R^{k-1}[i,j] \text{ OR } (R^{k-1}[i,k] \text{ AND } R^{k-1}[k,j]) \}$$

```
ALGORITHM  Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

Example:



$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 1 & 0 \end{array} \qquad T = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 \end{array}$$

$$R^{(0)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{0} \\ 2 & \boxed{0} & 0 & 0 & 1 \\ 3 & \boxed{0} & 0 & 0 & 0 \\ 4 & \boxed{1} & 0 & 1 & 0 \end{array}$$

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

Let the elements in $R^1$ matrix, means number of internal nodes will be **1**

$R^1[1,1] = \{ R^0[1,1]$ **OR** $(R^0[1,1]$ **AND** $R^0[1,1]) \} = \{0 \text{ OR } (0 \text{ AND } 0)\} = 0$

$R^1[1,2] = \{ R^0[1,2]$ **OR** $(R^0[1,1]$ **AND** $R^0[1,2]) \} = \{1 \text{ OR } (0 \text{ AND } 1)\} = 1$

$R^1[1,3] = \{ R^0[1,3]$ **OR** $(R^0[1,1]$ **AND** $R^0[1,3]) \} = \{0 \text{ OR } (0 \text{ AND } 0)\} = 0$

$R^1[1,4] = \{ R^0[1,4]$ **OR** $(R^0[1,1]$ **AND** $R^0[1,4]) \} = \{0 \text{ OR } (0 \text{ AND } 0)\} = 0$

$R^1[2,1] = \{ R^0[2,1]$ **OR** $(R^0[2,1]$ **AND** $R^0[1,1]) \} = \{0 \text{ OR } (0 \text{ AND } 0)\} = 0$

$R^1[2,2] = \{ R^0[2,2]$ **OR** $(R^0[2,1]$ **AND** $R^0[1,2]) \} = \{0 \text{ OR } (0 \text{ AND } 1)\} = 0$

Similarly we can find the remaining elements in $R^1$ matrix

$$R^{(1)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex 1 (note a new path from 4 to 2); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., 1 and 2 (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., 1,2 and 3 (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., 1,2,3 and 4 (note five new paths).

The strategy adopted by the Floyd-Warshall algorithm is **Dynamic Programming**. The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines to the elements in each matrix. Each execution of line of element takes O (1) time. The algorithm thus runs in time $\theta(n^3)$.

## Time Complexity:

| | |
|---|---|
| Traversals(BFS, DFS) | O(V+E) |
| Kruskal's Algorithm(MST) | O(E logV) |
| Prim's Algorithm(MST) | O((V+E) log V) |
| Sollin's Algorithm(MST) | O(V+E) |
| Dijkstra's Algorithm(Single Source) | O(E logV) |
| Floyd-Warshall Algorithm (All-Pair) | $O(V^3)$ |