# DIGITAL NOTES

# ON

# DATA STRUCTURES USING C++

**B.TECH II YEAR - I SEM**

**(2018-19)**

**Department of Computer Science and Engineering**

**SRI INDU INSTITUTE OF ENGINEERING & TECHNOLOGY**
**Sheriguda(V),Ibrahimpatnam(M),Hyderabad**
**R.R Dist., Telangana State-501510**

**JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD**

**II Year B.Tech. CSE – I Sem**

| L | T/P/D | C |
|---|---|---|
| 4 | -/-/- | 4 |

**(CS302ES)DATA STRUCTURES THROUGH C++**

**Course Objectives:**

- To understand the basic concepts such as Abstract Data Types, Linear and Non Linear Data structures.
- To understand the notations used to analyze the Performance of algorithms.
- To understand the behavior of data structures such as stacks, queues, trees, hash tables, search trees, Graphs and their representations.
- To choose an appropriate data structure for a specified application.
- To understand and analyze various searching and sorting algorithms.
- To learn to implement ADTs such as lists, stacks, queues, trees, graphs, search trees in C++ to solve problems.

**Course Outcomes:**

- Ability to choose appropriate data structures to represent data items in real world problems.
- Ability to analyze the time and space complexities of algorithms.
- Ability to design programs using a variety of data structures such as stacks, queues, hash tables, binary trees, search trees, heaps, graphs, and B-trees.
- Able to analyze and implement various kinds of searching and sorting techniques.

**UNIT-I**

C++ Programming Concepts: Review of C, input and output in C++, functions in C++- value parameters, reference parameters, Parameter passing, function overloading, function templates, Exceptions-throwing an exception and handling an exception, arrays, pointers, new and delete operators, class and object, access specifiers , friend functions, constructors and destructor, Operator overloading, class templates, Inheritance and Polymorphism.
Basic Concepts - Data objects and Structures, Algorithm Specification-Introduction, Recursive algorithms, Data Abstraction, Performance analysis- time complexity and space complexity, Asymptotic Notation-Big O, Omega and Theta notations, Complexity Analysis Examples, Introduction to Linear and Non Linear data structures.

**UNIT-II**

Representation of single, two dimensional arrays, sparse matrices-array and linked representations.
Linear list ADT-array representation and linked representation, Singly Linked Lists- Operations-Insertion, Deletion, Circularly linked lists-Operations for Circularly linked lists, Doubly Linked Lists-Operations- Insertion, Deletion.
Stack ADT, definition, array and linked implementations, applications-infix to postfix conversion, Postfix expression evaluation, recursion implementation, Queue ADT, definition, array and linked Implementations, Circular queues-Insertion and deletion operations.

**UNIT-III**

Trees – definition, terminology, Binary trees-definition, Properties of Binary Trees, Binary Tree ADT, representation of Binary Trees-array and linked representations, Binary Tree traversals, Threaded binary trees, Priority Queues –Definition and applications, Max Priority Queue ADT-implementation-Max Heap-Definition, Insertion into a Max Heap, Deletion from a Max Heap.

**UNIT-IV**

Searching - Linear Search, Binary Search, Hashing-Introduction, hash tables, hash functions, Overflow Handling, Comparison of Searching methods.

Sorting- Insertion Sort, Selection Sort, Radix Sort, Quick sort, Heap Sort, Merge sort, Comparison of Sorting methods.

**UNIT-V**

Graphs –Definitions, Terminology, Applications and more definitions, Properties, Graph ADT, Graph Representations- Adjacency matrix, Adjacency lists, Graph Search methods - DFS and BFS, Complexity analysis,

Search Trees -Binary Search Tree ADT, Definition, Operations- Searching, Insertion and Deletion, Balanced search trees-AVL Trees-Definition and Examples only, B-Trees- Definition and Examples only, Red-Black Trees-Definitions and Examples only, Comparison of Search Trees.

**TEXT BOOKS:**

1. Data structures, Algorithms and Applications in C++, 2nd Edition, Sartaj Sahni, Universities Press.
2. Data structures and Algorithms in C++, Adam Drozdek, 4th edition, Cengage learning.

**REFERENCE BOOKS:**

1. Data structures with C++, J. Hubbard, Schaum's outlines, TMH.
2. Data structures and Algorithms in C++, M.T. Goodrich, R. Tamassia and D. Mount, Wiley India.
3. Data structures and Algorithm Analysis in C++, 3rd edition, M. A. Weiss, Pearson.
4. Classic Data Structures, D. Samanta, 2nd edition, PHI.

# UNIT-I

**C++ programming language** was developed in 1980 by Bjarne Stroustrup at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.
**Bjarne Stroustrup** is known as the **founder of C++ language**
It was develop for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C com C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program ponent

## Input and output in C++

The standard C++ library is iostream and standard input / output functions in C++ are:
1.cin
2.cout

**cin**

It is the method to take input any variable / character / string.
Syntax:
cin>>variable / character / String / ;

**Cout**

This method is used to print variable / string / character.
Syntax:
cout<< variable / charcter / string;

**program to add two numbers**

```
#include <iostream.h>
#include<conio.h>
Void main()
{
int a,b,c;
cout<<"enter values for a and b";
cin>>a>>b;
c=a+b;
cout<<"the total is:<<c;
}
```

## Functions in C++

A function is a block of code that performs a specific task. It has a name and it is reusable i.e. it can be executed from as many different parts as required. It also optionally returns a value to the calling function.
A complex problem may be decomposed into a small or easily manageable parts or modules called functions. Functions are very useful to read, write, debug and modify complex programs They can also be incorporated in the main program. The main() itself is a function is invoking the other functions to perfom various tasks.

```
return_type function_name(argument_list)
{ statement 1;
   .
   .
statements n;
[return value];
}
```

**CALL BY VALUE**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main(

```
#include <iostream>
using namespace std;
void change(int data);
int main()
{
int data = 3;
change(data);
cout << "Value of the data is: " << data<< endl;
return 0;
}
void change(int data)
{
data = 5;
}
```

**Call BY REFERENCE**

In call by reference, original value is modified because we pass reference (address).
Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
 *x=*y;
 *y=swap;
}
int main()
{
 int x=500, y=100;
 swap(&x, &y);  // passing value to function
 cout<<"Value of x is: "<<x<<endl;
 cout<<"Value of y is: "<<y<<endl;
 return 0;
}
```

**Function Overloading**

In C++, it is possible to make more than one function with same name, this concept is known as function overloading. We can use same function name for different purpose with different number and types of arguments.
In function overloading function names will be same but Types of arguments, Order of arguments, Number of arguments must be different.

The C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

Following advantages to use function overloading in your program:

- Eliminating the use of different function names for the same operations.
- Helps to understand, debug and group easily.
- Easy maintainability of the code.
- Better understandability of the relationship b/w the program and the outside world.

```
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
       return a + b;
   }
static int add(int a, int b, int c)
   {
       return a + b + c;
   }
};
int main(void) {
   Cal C;                                  //    class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

**Function Templates**

In C++, **function templates** are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called **template type parameters**. Once we have created a function using these placeholder types, we have effectively created a “function stencil”.

```
 include <iostream>
 template <typename T>
const T& max(const T& x, const T& y)
{
   return (x > y) ? x : y;
}

int main()
{
   int i = max(3, 7); // returns 7
   std::cout << i << '\n';

   double d = max(6.34, 18.523); // returns 18.523
```

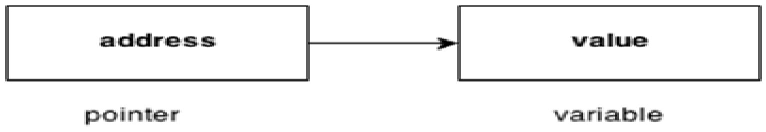
```
        std::cout << d << '\n';

        char ch = max('a', '6'); // returns 'a'
        std::cout << ch << '\n';

        return 0;
    }
```

**C++ Arrays**
Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.
**C++ Array Types**
There are 2 types of arrays in C++ programming:

- Single Dimensional Array
- Multidimensional Array

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array
        //traversing array
        for (int i = 0; i < 5; i++)
        {
            cout<<arr[i]<<"\n";
        }
}
```

**C++ Pointers**
The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



**Advantage of pointer**
1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.
2) We can return multiple values from function using pointer.
3) It makes you able to access any memory location in the computer's memory.
**Usage of pointer**
There are many usage of pointers in C++ language.
1) Dynamic memory allocation
In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.
2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

```
#include <iostream>
using namespace std;
int main()
{
int number=30;
int ∗   p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
  return 0;
}
```

**new and delete Operators**

new is used to allocate memory for a variable, object, array, array of objects.. etc at run time. *"To* declaring memory at run time is known as dynamic memory allocation*."* we use memory by this method (dynamic allocation) when it is not known in advance how much memory space is needed**.**

since "these operators manipulate memory on the free store, so they are also known as free store operators. the new operator can be used to create objects of any type, it takes the following general form*:*

pointer_variable = new data_type;

**delete Operator :**

delete operator is used to Deallocates the dynamically allocated memory.

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete.
delete takes the following general form:
1. delete pointer_variable;
2. delete []pointer_variable;

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

    void main()
    {

              int size,i;
              int *ptr;

              cout<<"\n\tEnter size of Array : ";
              cin>>size;

              ptr = new int[size];


     for(i=0;i<5;i++)        //Input arrray from user.
     {
            cout<<"\nEnter any number : ";
            cin>>ptr[i];
    }

    for(i=0;i<5;i++)        //Output arrray to console.
    cout<<ptr[i]<<", ";

    delete[] ptr;
    //deallocating all the memory created by new operator

   }
```

**C++ Exception Handling**
Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

**C++ Exception Handling Keywords**
In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

The C++ try block is used to place the code that may occur exception. The catch block is used to handle the exception.

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  if( y == 0 ) {
    throw "Attempted to divide by zero!";
  }
  return (x/y);
}
int main () {
  int i = 25;
  int j = 0;
  float k = 0;
  try {
    k = division(i, j);
    cout << k << endl;
  }catch (const char* e) {
    cerr << e << endl;
  }
  return 0;
}
```

**C++ Object and Class**
Since C++ is an object-oriented language, program is designed using objects and classes in C++.
**C++ Object**
In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.
In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.
Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

```
Student s1;  //creating an object of Student
```

**C++ Class**
In C++, object is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

```
#include <iostream>
using namespace std;
class Student {
   public:
       int id;//data member (also instance variable)
       string name;//data member(also instance variable)
       void insert(int i, string n)
        {
            id = i;
            name = n;
        }
       void display()
        {
            cout<<id<<"  "<<name<<endl;
        }
};
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```

**Access specifiers (public, protected, private) in C++**
C++ provides three access specifiers: public, protected and private
**public**
Data members or Member functions which are declared as public can be accessed anywhere in the program (within the same class, or outside of the class).
**protected**
Data members or Member functions which are declared as protected can be accessed in the derived class or within the same class.

**private**
Data members of Member functions which are declared as private can be accessed within the same class only i.e. the private Data members or Member functions can be accessed within the public member functions of the same class.

**C++ Friend function**
If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
By using the keyword friend compiler knows the given function is a friend function.
For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

**Declaration of friend function in C++**

```
class class_name
{
   friend data_type function_name(argument/s);           // syntax of friend function.
};

#include <iostream>
using namespace std;
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

**C++ Constructor**

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor
- Copy Constructor

**C++ Default Constructor**

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

**C++ Parameterized Constructor**

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

**C++ Copy Constructor**

A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.

```
#include <iostream>
using namespace std;
class A
{
   public:
    int x;
    A(int a)              // parameterized constructor.
    {
      x=a;
    }
    A(A &i)               // copy constructor
    {
        x = i.x;
    }
};
int main()
{
  A a1(20);               // Calling the parameterized constructor.
 A a2(a1);                //  Calling the copy constructor.
 cout<<a2.x;
  return 0;
}
```

**C++ Destructor**

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.
A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
           cout<<"Constructor Invoked"<<endl;
        }
        ~Employee()
        {
            cout<<"Destructor Invoked"<<endl;
        }
    };
    int main(void)
    {
        Employee e1; //creating an object of Employee
        Employee e2; //creating an object of Employee
        return 0;
    }
```

**C++ Operators Overloading**

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.
The advantage of Operators overloading is to perform different operations on the same operand.

**Operator that cannot be overloaded are as follows:**

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

**Syntax of Operator Overloading**

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
 }
```

```
#include <iostream>
using namespace std;
class Test
{
   private:
      int num;
   public:
       Test(): num(8){}
       void operator ++()         {
          num = num+2;
       }
       void Print() {
           cout<<"The Count is: "<<num;
       }
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

**class template**
class template provides a specification for generating classes based on parameters. *Class templates* are generally used to implement containers. A class template is instantiated by passing a given set of types to it as template arguments

```
   // class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

**C++ Inheritance**
In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

**Types Of Inheritance**
C++ supports five types of inheritance:
- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

**Derived Classes**
A Derived class is defined as the class derived from the base clas

**The Syntax of Derived class**:

```
class derived_class_name :: visibility-mode base_class_name
```

```
{
   // body of the derived class.
}
```

**Visibility of Inherited Members**

| Base Class Visibility | Derived Class Visibility | | |
|---|---|---|---|
| | PUBLIC | PRIVATE | PROTECTED |
| Private | Not Inherited | Not Inherited | Not Inherited |
| Protected | Protected | Private | Protected |
| Public | Public | Private | Protected |

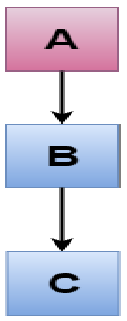
**Single Inheritance**

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class

```
 #include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
     Programmer p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

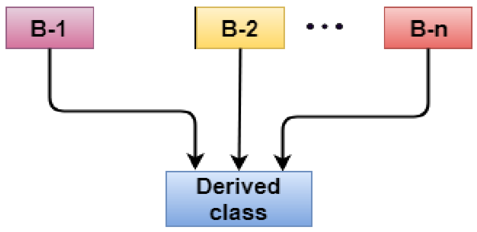
**Multilevel Inheritance**

Multilevel inheritance is a process of deriving a class from another derived class.



```
#include <iostream>
using namespace std;
 class Animal {
    public:
  void eat() {
    cout<<"Eating..."<<endl;
  }
    };
    class Dog: public Animal
    {
        public:
      void bark(){
     cout<<"Barking..."<<endl;
      }
    };
    class BabyDog: public Dog
    {
        public:
      void weep() {
     cout<<"Weeping...";
      }
    };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

**Multiple Inheritance**
Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



**Syntax of the Derived class:**

```
class D : visibility B-1, visibility B-2, ?
{
    // Body of the class;
}
```

Let's see a simple example of multiple inheritance.

```
#include <iostream>
using namespace std;
class A
{
    protected:
     int a;
    public:
    void get_a(int n)
    {
       a = n;
    }
};

class B
{
    protected:
    int b;
    public:
    void get_b(int n)
    {
       b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
       std::cout << "The value of a is : " <<a<< std::endl;
       std::cout << "The value of b is : " <<b<< std::endl;
       cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

**Polymorphism**
Polymorphism is an important and basic concept of OOPS. Polymorphism specifies the ability to assume several forms. It allows routines to use variables of different types at different times.
In C++, An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.
Types of polymorphism
There are two types of polymorphism in C++

1. Static or Compile time polymorphism
2. Dynamic or Run time polymorphism

**1) Static or compile time polymorphism**
In this type of polymorphism behavior of functions and operators decide at compile time. Thus, it is known as static or compile time polymorphism.

There are two types of static polymorphism:

- Function overloading
- Operator overloading

**2) Dynamic or run time polymorphism**
Dynamic polymorphism is basically used for runtime time member function binding. Thus, it is known as dynamic polymorphism.
There are following types of dynamic polymorphism:

- Virtual functions.
- Dynamic binding

**C++ virtual function**

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

**Algorithm:**
An algorithm is a finite set of step by step instructions to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used toperform a task.
In computer science, an algorithm can be defined as follows:

An algorithm is a sequence of unambiguous instructions used for solving aproblem, which can be implemented (as a program) on a computer.

**Properties**

Every algorithm must satisfy the following properties:

1. Definiteness - Every step in an algorithm must be clear and unambiguous
2. Finiteness – Every algorithm must produce result within a finite number of steps.
3. Effectiveness - Every instruction must be executed in a finite amount of time.
4. Input & Output - Every algorithm must take zero or more number of inputs and must produce at least one output as result.

**Performance Analysis**

In computer science there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one.

Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.When there are multiple alternative algorithms to solve a problem, we analyses them and pick the one which is best suitable for our requirements. Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all remaining elements.Performance analysis of an algorithm is performed by using the following measures:

1. Space Complexity
2. Time Complexity

**Space complexity**

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.

**3. Data Space:** It is the amount of memory used to store all the variables and constants.

**NOTE:** When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack. That means we calculate only the memory required to store Variables, Constants, Structures, etc

Consider the following piece of code...

```
    int square(int a)
    {
    return a*a;
}
```

In above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value. That means, totally it requires 4 bytes of memory to complete its execution.

**Time complexity**

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution. Generally, running time of an algorithm depends upon the following:

- Whether it is running on Single processor machine or Multi processor machine.
- Whether it is a 32 bit machine or 64 bit machine
- Read and Write speed of the machine.
- The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,

**NOTE:** When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.

Consider the following piece of code...

```
Algorithm Search (A, n, x)
{                // where A is an array, n is the size of an array and x is the item to be searched.
for i := 1 to n do
{
if(x=A[i]) then
{
write (item found at location i) return;
}
}
write (item not found)
}
```

For the above code, time complexity can be calculated as follows:
Cost is the amount of computer time required for a single operation in each line. Repetition is the amount of computer time required by each operation for all its repetitions, so above code requires '*n*' units of computer time to complete the task.

**Asymptotic Notation**

Asymptotic notation of an algorithm is a mathematical representation of its complexity. Majorly, we use THREE types of Asymptotic Notations and those are:
1. Big - Oh (O)
2. Omega (Ω)
3.Theta (Θ)

**Big - Oh Notation (O)**

- Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.
- Big - Oh notation always indicates the maximum time required by an algorithm for all input values.
- Big - Oh notation describes the worst case of an algorithm time complexity.
- It is represented as O(T)

**Omega Notation (Ω)**

- Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.
- Omega notation always indicates the minimum time required by an algorithm for all input values.
- Omega notation describes the best case of an algorithm time complexity.
- It is represented as **Ω** (T)

**Theta Notation (Θ)**

- Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.
- Theta notation always indicates the average time required by an algorithm for all input

values.

- Theta notation describes the average case of an algorithm time complexity.
- It is represented as **Θ** (T)
-

**Example**

Consider the following piece of code

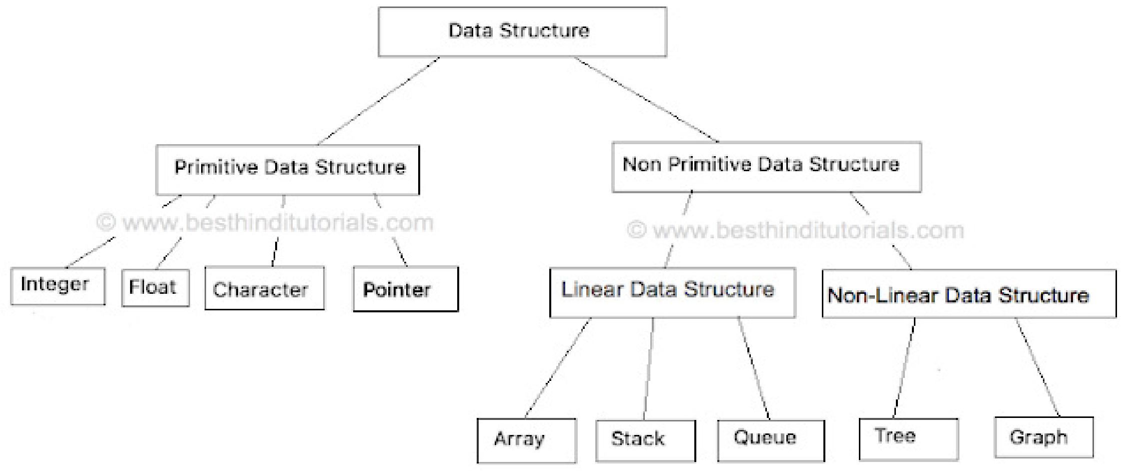
```
Algorithm Search (A, n,x)
{                // where A is an array, n is the size of an array and x is the item to be searched.
for i := 1 to n do
{
if(x=A[i]) then
{
write (item found at location i) return;
}
}
write (item not found)
}
```

The time complexity for the above algorithm

- Best case is **Ω** (1)
- Average case is **Θ** (n/2)
- Worst case is **O** (n)

**What is Data Structure?**

Data may be organized in many different ways: The logical or mathematical model of a particular organization of data is called *data structure*. Data structures are generally classified into primitive and non- primitive data structures



Based on the organizing method of a data structure, data structures are divided into two types.

- Linear Data Structures
- Non - Linear Data Structures

**Linear Data Structures**

If a data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure. For example:

- Arrays
- Lists (Linked List)

- Stacks
- Queues

**Non - Linear Data Structures**

If a data structure is organizing the data in random order, then that data structure is called as Non-Linear Data Structure. For example:

- Trees
- Graphs
- Dictionaries
- Heaps , etc

# Unit-II

## Array as an ADT

The **array** is a basic abstract data type that holds an ordered collection of items accessible by an integer index. Since it's an ADT, it doesn't specify an implementation, but is almost always implemented by an array data structure or dynamic array.

## Single dimensional Array

An array is collection of homogeneous elements that are represented under a single variable name

A linear array is a list of a finite number of *n* homogeneous data elements (that is data elements of the same type) such that

- The elements are referenced respectively by an index set consisting of *n*
- consecutive numbers
- The elements are stored respectively in successive memory locations
- The number *n* of elements is called the *length* or *size* of the array.
- The index set consists of the integer 0,1, 2, … n-1.
- Length or the number of data elements of the array can be obtained from the index set by

  **Length = UB – LB + 1**

  where UB is the largest index called the upper bound and LB is the smallest index called the lower bound of the arrays

- Element of an array A may be denoted by

- Subscript notation A1, A2, , …. , An
- Parenthesis notation A(1), A(2), …. ,A(n)
- Bracket notation A[1], A[2], ….. , A[n]
- The number K in A[K] is called subscript or an index and A[K] is called a subscripted variable

## 1.5.2 Representation of single dimensional array in memory

✓ Let LA be a linear array in the memory of the computer.
✓ LOC(LA[K]) = address of the element LA[K] of the array LA
✓ The elements of LA are stored in the successive memory locations.

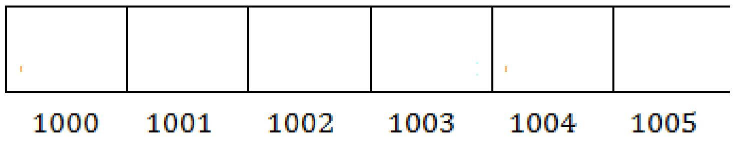
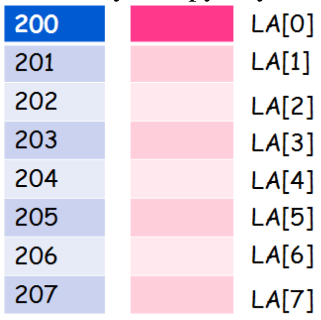


Fig. memory representation of an array of elements

Computer does not need to keep track of the address of every element of LA, but need to track only the address of the first element of the array denoted by
Base(LA)
and called the *base address* of LA. Using this address, the computer calculates the address of any element of LA by the following formula:
**LOC(LA[K]) = Base(LA) + w(K – LB)**
where **w** is the number of words per memory cell of the array LA [**w** is the size of the **data type**] .

**Example:** Find the address for LA [6]. Each element of the array occupy 1 byte LOC(LA[K]) = Base(LA) + w(K – lower

bound)
LOC(LA[6]) = 200 + 1(6 – 0) = 206



| | | |
|---|---|---|
| 200 | | LA[0] |
| 201 | | LA[1] |
| 202 | | LA[2] |
| 203 | | LA[3] |
| 204 | | LA[4] |
| 205 | | LA[5] |
| 206 | | LA[6] |
| 207 | | LA[7] |

**Representation of two dimensional array**

For example, consider the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

In the row-major order layout (adopted by C for statically declared arrays), the elements of each row are stored in consecutive positions:

To find the position of the  element in an array of size mxn

Loc(A[i][j])=base(A) +W{m(j-lowerbound of column index)+ (i-lowerbound of row index)}

Where  W indicates the size of an element

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

In Column-major order (traditionally used by Fortran), the elements of each column are consecutive in memory:

To find the position of the  element in an array of size mxn

Loc(A[i][j])=base(A) +W{n(i-lowerbound of row index)+ (j-lowerbound of column index)}

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 7 | 2 | 5 | 8 | 3 | 6 | 9 |

**Sparse Matrix**
Matrices with a relatively high proportion of zero entries are called sparse matrices. Two general types of n-square sparse matrices occur in various applications. They are:

- Triangular matrix
- Tridiagonal matrix

In **triangular matrix**, all entries above the main diagonal are zero or equivalently, where nonzero entries can only occur on or below the main diagonal.

In **tridiagonal matrix**, where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal.

$$\begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & 2 & 1 & & \\ 1 & 3 & 3 & 1 & \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix}$$

(a) Triangular matrix

$$\begin{pmatrix} 1 & 4 & & \\ 3 & 4 & 1 & \\ & 2 & 3 & 4 \\ & & 1 & 3 \end{pmatrix}.$$

(b) Tridiagonal matrix

**Sparse Matrix Representations**

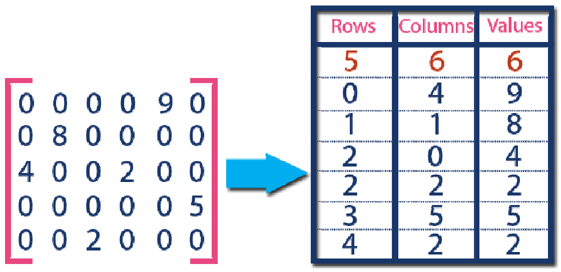
A sparse matrix can be represented by using two representations, those are:

- Triplet Representation
- Linked Representation

**Triplet Representation**

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the $0^{th}$ row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the table:



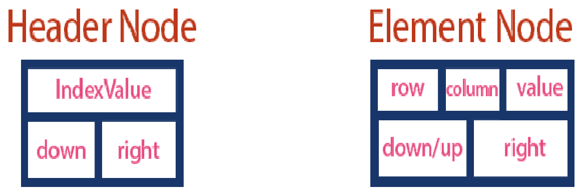
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

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

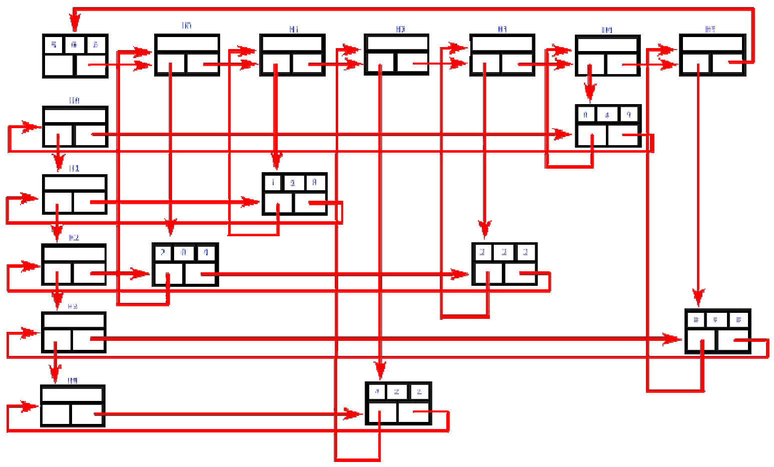
In above example matrix, there are only 6 non-zero elements ( those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above table. Here the first row in the right side table is filled with values 5, 6 & 6 which indicate that it is a sparse matrix with 5 rows, 6 columns and 6 non-zero values. Second row is filled with 0, 4, and 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

**Linked Representation**

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the fig.



Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below figure.



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.

**SINGLY LINKED LIST**

A singly linked list, or simply a linked list, is a linear collection of data items. The linear order is given by means of POINTERS. These types of lists are often referred to as linear linked list.

*Each item in the list is called a node.

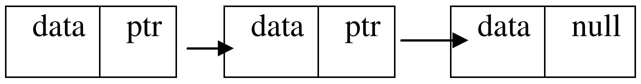
*Each node of the list has two fields:

**1.Information**- contains the item being stored in the list.

**2.Next address**- contains the address of the next item in the list.

*The last node in the list contains NULL pointer to indicate that it is the end of the list.

Conceptual view of Singly Linked List

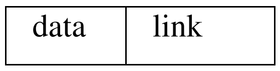


Operations on Singly linked list:

- creation
- Insertion of a node
- Deletions of a node
- Traversing the list

Structure of a node: **Method -1:**

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



**Method -2:**

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

Insertions: To place an elements in the list there are 3 cases :

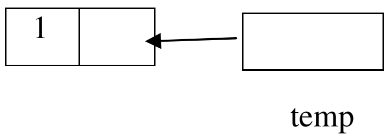
1.At the beginning

2.End of the list

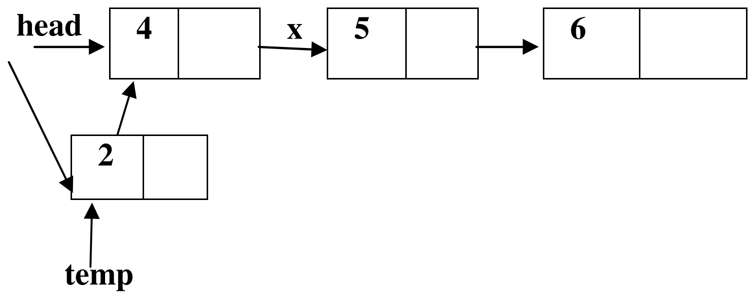
3.At a given position

**Creating a node:**

```
Void create(){
node *temp;
temp=new node;
temp->data=1;
temp->link=null;
if(head==NULL)
head=temp;
}
```

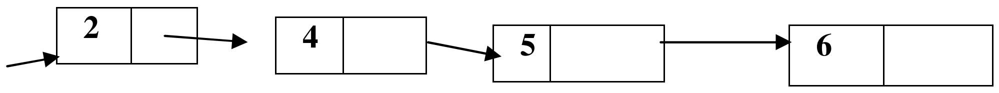


**case 1:Insert at the beginning**



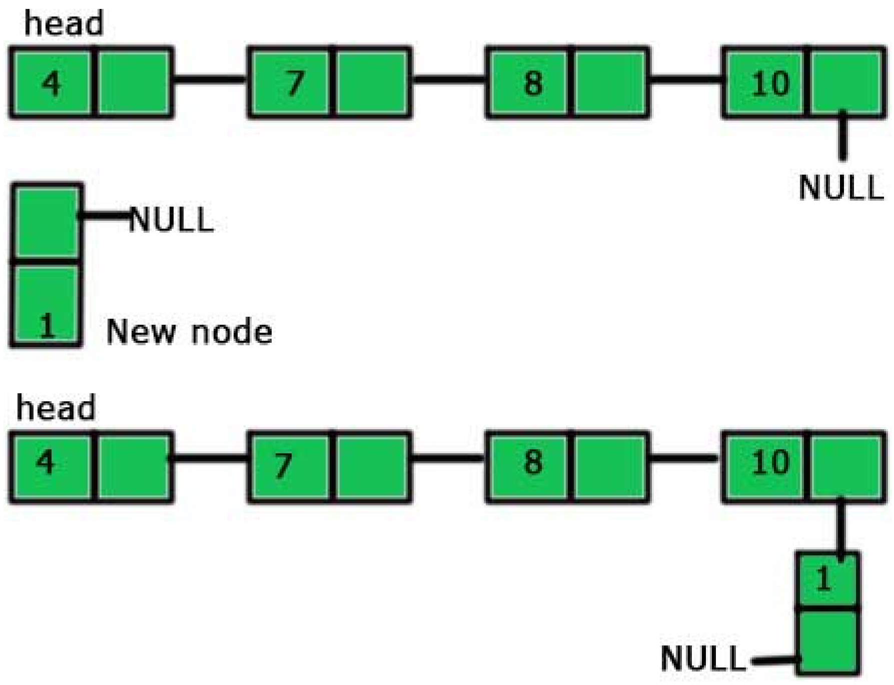
**head** is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

**After insertion**



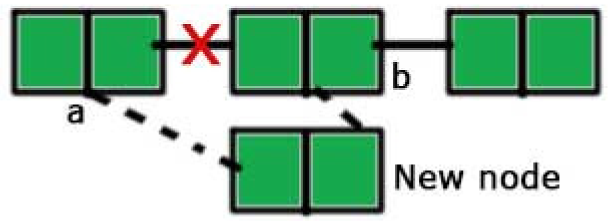
```
                template <class T>
void list<T>::insert_front()
{
struct node <T>*t,*temp;
cout<<"Enter data into node:";
cin>>item;
temp=create_node(item);
if(head==NULL)
head=temp;
else
{temp->link=head;
head=temp;
}}
```

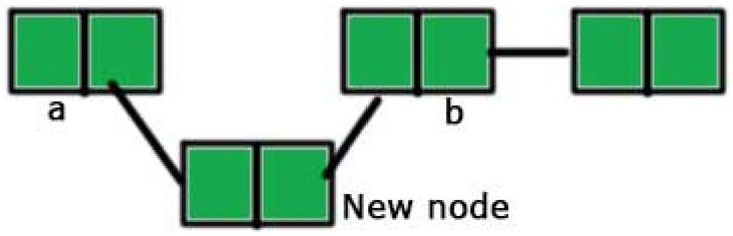
**case 2:**Inserting end of the list



```
template <class T>
void list<T>::insert_end()
{
struct node<T> *t,*temp;
int n;
cout<<"Enter data into node:";
cin>>n;
temp=create_node(n);
if(head==NULL)
head=temp;
else
{t=head; while(t-
>link!=NULL)
t=t->link;
t->link=temp;
}
}
```

**case 3: Insert at a position**





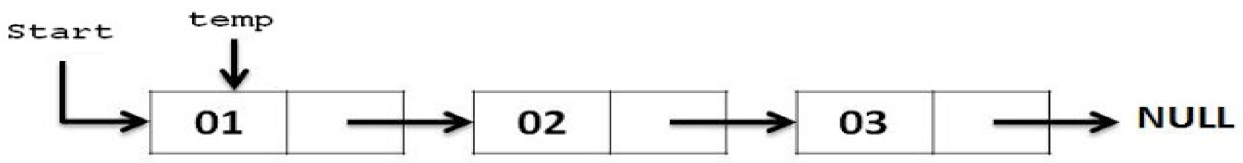
```
template <class T>
void list<T>::Insert_at_pos(int pos)
{struct node<T>*cur,*prev,*temp;
int c=1;
cout<<"Enter data into node:";
cin>>item
temp=create_node(item);
if(head==NULL)
head=temp;
else{
prev=cur=head;
if(pos==1)
{
temp->link=head;
head=temp;
}
else
{
while(c<pos)
{c++;
prev=cur;
cur=cur->link;
}
prev->link=temp;
temp->link=cur;
} } }
```

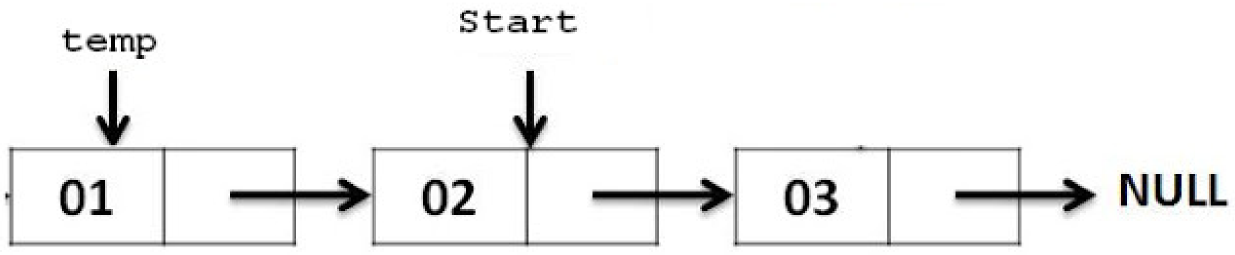
**Deletions:** Removing an element from the list, without destroying the integrity of the list itself

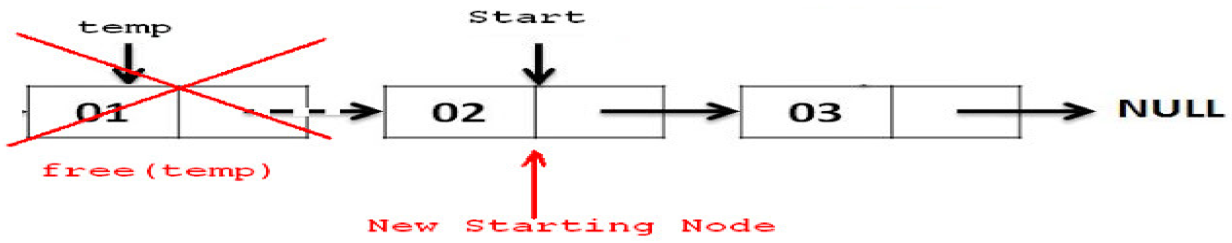
To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

**Delete a node at beginning of the list**







```
del_at beg(){
node *list;
list =head;
if(list->data==n)
{
head=list->next;
list->next=NULL;
}
delete list;
```

**Delete a node at end of the list**

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

**Deleting a Specific Node from the list**

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

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

A singly linked list has the disadvantage that we can only traverse it in one direction. Many applications require searching backwards and forwards through sections of a list. A useful refinement that can be made to the singly linked list is to create a doubly linked list.The distinction made between the two list types is that while singly linked list have pointers going in one direction, doubly linked list have pointer both to the next and to the previous element in the list. The main advantage of a doubly linked list is that, they permit traversing or searching of the list in both directions.

In this linked list each node contains three fields.

- One to store data
- Remaining are self referential pointers which points to previous and next nodes in the list

| | | |
|---|---|---|
| prev | data | next |

**Implementation of node using structure**

**Method -1:**

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

**Implementation of node using class**

**Method -2:**

```
           class node
{
public:
int data;
node *prev;
node * next;
            };
```

**Operations on Double Linked List**

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

- Insertion
- Deletion
- Display

**Insertion**

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

- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

**Inserting At Beginning of the list**

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

- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head == NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

**Inserting At End of the list**

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

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head == NULL**)
- **Step3-** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

**Inserting At Specific location in the list (After a Node)**

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

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head == NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous**& **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

**Deletion**

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

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

**Deleting from Beginning of the list**

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

- **Step 1 -** Check whether list is **Empty** (**head == NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.

- **Step 4 -** Check whether list is having only one node (**temp → previous**is equal to **temp → next**)
- **Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

**Deleting from End of the list**

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

- **Step 1 -** Check whether list is **Empty** (**head == NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- **Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- **Step 7 -** Assign **NULL** to **temp → previous → next** and delete **temp**.

**Deleting a Specific Node from the list**

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

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

### Displaying a Double Linked List

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

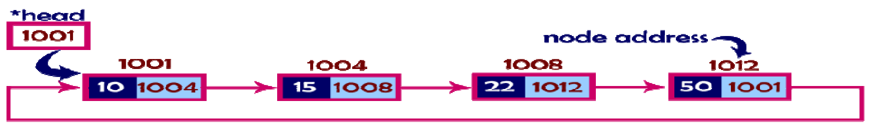
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Display **'NULL <--- '**.
- **Step 5 -** Keep displaying **temp** → **data** with an arrow (**<===>**) until **temp** reaches to the last node
- **Step 6 -** Finally, display **temp** → **data** with arrow pointing to **NULL**(**temp** → **data** **--->** **NULL**).

### Circular Linked List

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

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

### Example



### Operations

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

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined** functions.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 5 -** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

### Insertion

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

- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

**Inserting At Beginning of the list**

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

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

**Inserting At End of the list**

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

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

**Inserting At Specific location in the list (After a Node)**

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

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head == NULL**)
- **Step 3 -** If it is **Empty** then, set **head = newNode** and **newNode → next = head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data**is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next = newNode** and **newNode → next = head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next = temp → next** and **temp → next = newNode**.

**Deletion**

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

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

**Deleting from Beginning of the list**

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

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.

**Step 4 -** Check whether list is having only one node (**temp1 → next == head**)

**Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)

**Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next == head** )

**Step 7 -** Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

**Deleting from End of the list**

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

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.

**Step 4 -** Check whether list has only one Node (**temp1 → next == head**)

**Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

**Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)

**Step 7 -** Set **temp2 → next** = **head** and delete **temp1**.

**Deleting a Specific Node from the list**

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

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

**Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

**Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

**Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

**Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)

**Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

**Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).

**Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.

**Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).

**Step 1 1-** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).

**Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

**Displaying a circular Linked List**

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

**Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

**Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.

**Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

**Step 4 -** Keep displaying **temp** → **data** with an arrow (**--->**) until **temp**reaches to the last node

**Step 5 -** Finally display **temp** → **data** with arrow pointing to **head** → **data**.

## STACK

Stack is a linear data structure and very much useful in various applications of computer science.

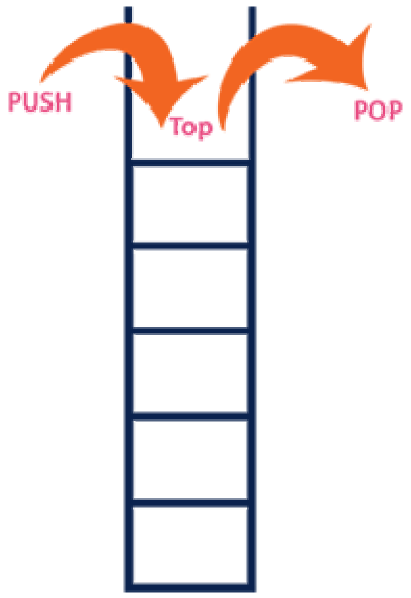
**Definition**

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

Like array and linked list, stack is also a linear data structure, but the only difference is that in case of former two, insertion and deletion operations can take place at any position.

In a stack the last inserted element is always processed first. Accordingly, stacks are called as Last-in-First-out (LIFO) lists. Other names used for stacks are "piles" and "push-down lists".

Stack is most commonly used to store local variables, parameters and return addresses when a function is called.



Stack Model of Stack

The variable top always keeps track of the top most element or position.

**STACK OPERATIONS**

- PUSH: “Push” is the term used to insert an element into a stack.
- POP: “Pop” is the term used to delete an element from a stack.

Two additional terms used with stacks are “overflow” & “underflow”. Overflow occurs when we
try to push more information on a stack that it can hold. Underflow occurs when we try to pop an item from a stack which is empty.

**REPRESENTATION OF STACKS**

A stack may be represented in the memory in various ways. Mainly there are two ways. They are:
1. Using one dimensional arrays(Static Implementation)
2. Using linked lists(Dynamic Implementation)

**Representation of Stack using Array**

First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, items of the stack can be stored in sequential manner.
Top is a pointer to point the
position of array upto which it is filled with the items of stack.
With this representation following two statements can be stated:
EMPTY: Top<l
FULL: Top>=u+l-1

**Algorithms for Stack Operations**

**Algorithm PUSH** (STACK, TOP, MAXSTK, ITEM)
This algorithm pushes an ITEM onto a stack.

```
Step 1: If TOP = MAXST K, then: \\ Check Overflow?
Print: OVERFLOW, and Exit.
Step 2: Set TOP := TOP + 1. \\ increases TOPby 1
Step 3: Set STACK[TOP] := ITEM. \\ Inserts ITEM in new TOP position.
Step 4: Exit.
```

Here we have assumed that array index varies from 1 to SIZE and Top points the location of the current top most item in the stack.
Algorithm POP (STACK, TOP, ITEM)
This algorithm deletes the top element of STACK and assigns it to the variable ITEM.

```
Step 1: If TOP = NULL, then: \\ Check Underflow.
Print: UNDERFLOW, and Exit.
Step 2: Set ITEM := STACK[TOP]. \\ Assigns TOP element to ITEM.
Step 3: Set TOP := TOP – 1. \\ Decreases TOP by 1.
Step 4: Exit.
```

Top is a pointer to point the position of array upto which it is filled with the items of stack.
With this representation following two statuses can be stated:
EMPTY: Top<l
FULL: Top>=u+l-1

**Algorithm: POP_**LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM).
This algorithm deletes the top element of a linked stack and assigns it to the variable ITEM.

```
Step 1: If TOP = NULL, then: \\ Check Underflow?
Write: UNDERFLOW and Exit.
Step 2: Set ITEM := STACK[TOP]. \\ Copies the TOP element of STACK into ITEM.
Step 3: Set TEMP := TOP, and
TOP := LINK[TOP]. \\ Reset the position of TOP.
Step 4: [Add node to the AVAIL list.]
Set LINK[TEMP] := AVAIL, AVAIL := TEMP.
Step 5: Exit.
```

**APPLICATIONS OF STACKS**
1. Reversing elements in a list or string.
2. Recursion Implementation.
3. Memory management in operating systems.
4. Evaluation of postfix or prefix expressions.
5. Infix to postfix expression conversion.
6. Tree and Graph traversals.
7. Checking for parenthesis balancing in arithmetic expressions.
8. Used in parsing.

**Arithmetic Expressions**
An expression is a collection of operators and operands that represents a specific value. In above definition,
 Operator is a symbol (+,-,>,<,..) which performs a particular task like arithmetic or logical or relational operation etc .,
 Operands are the values on which the operators can perform the task.
Expression Types
Based on the operator position, expressions are divided into THREE types. They are as follows :
1. Infix Expression
2. Postfix Expression
3. Prefix Expression

**1.Infix Expression**

In infix expression, operator is used in between operands. This notation is also called as polish notation. The general structure of an Infix expression is:

**operand <operator> operand**

**eg: A+B**

**2.Postfix Expression**

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands". This notation is also called as reverse - polish notation. The general structure of Postfix expression is:

**Operand operand <operator>**

**Eg:AB+**

**3.Prefix Expression**

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator". The general structure of Prefix expression is:

**<Operator > operand operand**

**Ex:+AB**

In 2nd and 3rdnotations, parentheses are not needed to determine the order evaluation of the operations in any arithmetic expression.

| Eg.: Infix | Prefix | Postfix | |
|---|---|---|---|
| 1. A+B*C | A+{*BC} | A+{BC*} | |
| | +A*BC | ABC*+ | { } indicate partial translation |
| 2. (A+B)*C | {+AB}*C | {AB+}*C | |
| | *+ABC | AB+C* | |

The computer usually evaluates an arithmetic expression written in infix notation in two steps.

1. It converts the expression to postfix notation and
2. It evaluates the postfix expression.

In each step, the stack is the main tool that is used to accomplish the given task.

2.5.1 Transform ing Infix Expression into Postfix Expression

In the infix expression we assume only exponentiation (^ or **), multiplication(*), division(/), addition (+), subtraction(-) operations. In addition to the above operators and operands they may contain left or right parenthesis. The operators three levels of priorities are:

High priority : ^ or **

Next high priority:*,/

Lowest +,-

The following algorithm transforms the infix expression Q into the equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parenthesis.

**ALGORITHM INF IX-TO-POSTF IX (Q)**

```
Step1: Push „(„ onto stack and add „)" to the end of Q.
Step 2: Scan Q from left to right and repeat steps 3 to 6 for each element of Q until STACK is empty.
Step 3: If an operand is encountered, add it to P.
Step 4: If a left parentheses is encountered, push it onto STACK.
Step 5: If an operator Θ is encountered, then:
a) Repeatedly pop from STACK and add to P each operator (on the top of
STACK) which has the same precedence as or higher precedence than Θ.
b) Add Θ to the STACK
[End of If Structure]
Step 6: If a right parentheses is encountered, then:
a) Repeatedly pop from STACK and add to P each operator (on the top of
STACK) until a left parentheses is encountered.
b) Remove the left parenthesis. [Do not add this to P]
Step 7: Exit.
```

Consider the following infix expression Q: A + (B * C - (D / E ^ F) * G) * H

| Symbol Scanned | Stack Contents | Expression P |
|---|---|---|
| Initial | ( | |
| (1) A | ( | A |
| (2) + | ( + | A |
| (3) ( | ( + ( | A |
| (4) B | ( + ( | AB |
| (5) * | ( + ( * | AB |
| (6) C | ( + ( * | ABC |
| (7) - | ( + ( - | ABC * |
| (8) ( | ( + ( - ( | ABC* |
| (9) D | ( + ( - ( | ABC*D |
| (10) / | ( + ( - ( / | ABC*D |
| (11) E | ( + ( - ( / | ABC*DE |
| (12) ^ | ( + ( - ( / ^ | ABC*DE |
| (13) F | ( + ( - ( / ^ | ABC*DEF |
| (14) ) | ( + ( - | ABC*DEF^/ |
| (15) * | ( + ( - * | ABC*DEF^/ |
| (16) G | ( + ( - * | ABC*DEF^/G |
| (17) ) | ( + | ABC*DEF^/G*- |

|  |  |  |
|---|---|---|
| (18) * | ( + * | ABC*DEF^/G*- |
| (19) H | ( + * | ABC*DEF^/G*-H |
| (20) ) |  | ABC*DEF^/G*-H * + |

OUTPUT P: ABC*DE^/G*-H * + (POSTFIX FORM)

## EVALUATION OF A POSTF IX EXPRESSION

Suppose P is an arithmetic expression written in postfix . The following algorithm uses a STACK to hold operands during the evaluation of P.

```
Algorithm EVALPOSTF IX(p)
Step 1: Add a right parentheses „)‟ at the end of P. //this acts as a sentinel
Step 2: Scan P from left to right and repeat steps 3 and 4 for each element of P until the
sentinel „)‟ is encountered.
Step 3: If an operand is encountered, push on STACK.
Step 4: If an operator Ө is encountered, then:
a) Remove the two top elements of STACK, where A is the top element and B isthe next- to-top
element.
b) Evaluate B Ө A.
c) Push the result of (b) back on STACK
[End of If structure]
[End of step2 loop]
Step 5: Set VALUE equal to top element on STACK.
Step 6: Return(VALUE)
Step 7: Exit.
```

**EXAMPLE**: Consider the following postfix expression:
P: 5, 6, 2, +, *, 12, 4, /, -

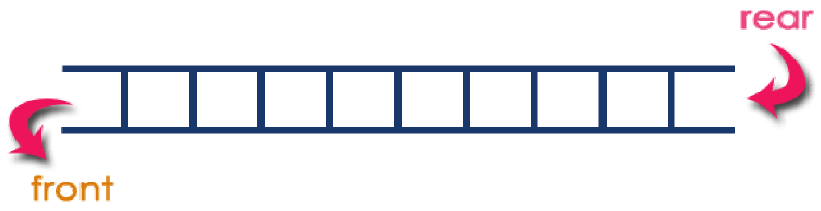
| Symbol Scanned | STACK |
|---|---|
| (1) 5 | 5 |
| (2) 6 | 5, 6 |
| (3) 2 | 5, 6, 2 |
| (4) + | 5, 8 |
| (5) * | 40 |
| (6) 12 | 40, 12 |
| (7) 4 | 40, 12, 4 |
| (8) / | 40, 3 |
| (9) - | 37 |
| (10) ) |  |

The result of this postfix expression is 37.

**QUEUE ADT**

Queue is a linear data structure used in various applications of computer science. Like people stand in a queue to get a particular service, various processes will wait in a queue for their turn to avail a service.

Queue is a linear list in which insertions takes place at one end called the **rear or tail** of the queue and deletions at the other end called as **front or head** of the queue.



When an element is to join the queue, it is inserted at the rear end of the queue and when an element is to be deleted, the one at the front end of the queue is deleted automatically.

In queues always the first inserted element will be the first to be deleted. That"s why it is also called as FIFO – First-in First-Out data structure (or FCFS – First Come First Serve data structure)

**APPLICATIONS of QUEUE**

- CPU Scheduling (Round Robin Algorithm)
- Printer Spooling
- Tree & Graph Traversals
- Palindrome Checker
- Undo & Redo Operations in some Software"s

**OPERATIONS ON QUEUE**

The queue data structure supports two operations:

Enqueue: Inserting an element into the queue is called enqueuing the queue. After the data have been inserted into the queue, the new element becomes the rear.

Dequeue: Deleting an element from the queue is called dequeuing the queue. The data at the front of the queue are returned to the user and removed from the queue.

**IMPLEMENTATION OF QUEUES**

Queues can be implemented or represented in memory in two ways:

1. Using Arrays (Static Implementation).
2. Using Linked Lists (Dynamic Implementation).

2.6.1.1 Implementation of Queue Us ing Arrays

A common method of implementing a queue data structure is to use another sequential data structure, viz, arrays. With this representation, two pointers namely, Front and Rear are used to indicate two ends of the queue. For insertion of next element, pointer Rear will be adjusted and for deletion pointer Front will be adjusted.

However, the array implementation puts a limitation on the capacity of the queue. The number of elements in the queue cannot exceed the maximum dimension of the one dimensional array. Thus a queue that is accommodated in an array Q[1:n], cannot hold more than n elements. Hence every insertion of an element into the queue has to necessarily test for a QUEUE FULL condition before executing the insertion operation. Again, each deletion has to ensure that it is not attempted on a queue which is already empty calling for the need to test for a QUEUE EMPTY condition before executing the deletion operation.

**Algorithm of insert operation on a queue**

```
Procedure INSERTQ (Q, n, ITEM, REAR)
// this procedure inserts an element ITEM into Queue with capacity n
Step 1: if(REAR=n ) then
Write:"QUEUE_FULL" and Exit
Step 2: REAR=REAR + 1 //Increment REAR
Step 3: Q[REAR]= ITEM //Insert ITEM as the rear element
Step 4: Exit
```

It can be observed that addition of every new element into the queue increments the REAR variable. However, before insertion, the condition whether the queue is full is checked

**Algorithm of delete operation on a queue**

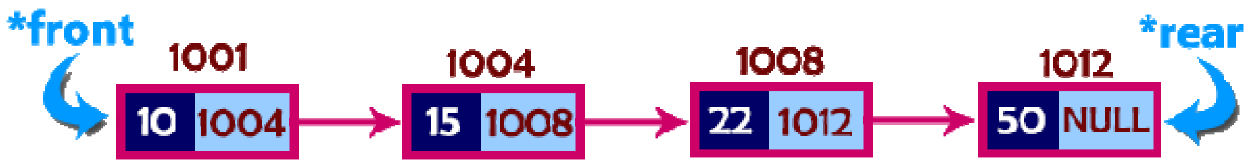
```
Procedure DELETEQ (Q, FRONT, REAR, ITEM)
Step 1: If (FRONT >REAR) then:

Write: "QUEUE EMPTY" and Exit.

Step 2: ITEM = Q[FRONT]
Step 3: FRONT =FRONT + 1
Step 4: Exit.
```

To perform delete operation, the participation of both the variables FRONT and REAR is essential. Before deletion, the condition FRONT =REAR checks for the emptiness of the queue. If the queue is not empty, the element is removed through ITEM and subsequently FRONT is incremented by 1.

**Linked Representation**

The major problem with the queue implemented using array is, It will work for only fixed number of data values. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of

the implementation). The Queue implemented using linked list can organize as many data values as we want.
In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



**Algorithm for Insertion:**

```
LINKQ_INSERT(INFO, LINK, FRONT, REAR, ITEM, AVAIL).
This algorithm inserts an element ITEM into a linked queue.
Step 1: [OVERF LOW?]
If AVAIL := NULL, then:
Write: OVERFLOW and Exit.
Step 2: [Remove first node from AVAIL list.]
Set NEW := AVAIL, and
AVAIL := LINK[AVAIL].
Step 3: Set INFO[NEW] := ITEM, and
LINK[NEW] := NULL. [Copies ITEM into new node.]
Step 4: If FRONT = NULL, then:
Set FRONT := NEW and REAR := NEW.
[If Q is empty then ITEM is the first element in the queue Q.]
Else:
Set LINK[REAR] := NEW, and
REAR := NEW.
[REAR points to the new node appended to the end of the list.]
Step 5: Exit.
```

**Algorithm for deletion**:

```
LINKQ_DELETE(INFO, LINK, FRONT, REAR, ITEM, AVAIL).
This algorithm deletes the front element of the linked queue and stores it in ITEM.
Step 1: [UNDERF LOW?]
If FRONT = NULL, then:
Write: UNDERFLOW and Exit.

Step 2: Set ITEM := INFO[FRONT]. [Save the data value of FRONT.]
Step 3: Set NEW := FRONT, and [Reset FRONT to the next position.]
Set FRONT := LINK[FRONT].
Step 4: Set LINK[NEW] := AVAIL, and [Add node to the AVAIL list.]
```

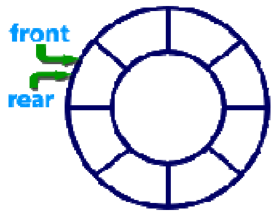
AVAIL := NEW
Step 5: EXIT

**C IRCULAR QUEUE**

One of the ma jor problems with the linear queue is the lack of proper utilization of space.Suppose that the queue can store 10 elements and the entire queue is full. So, it means that the queue is holding 10 elements. In case, some of the elements at the front are deleted, the element at the last position in the queue continues to be at the same position and there is no efficient way to find out that the queue is not full.

In this way, space utilization in the case of linear queues is not efficient. This problem is arising due to the representation of the queue.

The alternative representation is to depict the queue as circular. In case, we are representing the queue using arrays, then, a queue with n elements starts from index 0 and ends at n-1. So, clearly, the first element in the queue will be at index 0 and the last element will be at n-1 when all the positions between index 0 and n-1 (both inclusive) are filled. Under such circumstances, front will point to 0 and rear will point to n-1. However, when a new element is to be added and if the rear is pointing to n-1, then, it needs to be checked if the position at index 0 is f ree. If yes, then the element can be added to that position and rear can be adjusted accordingly. In this way, the utilization of space is increased in the case of a circular queue.
In a circular queue, front will point to one position less to the first element. So, if the first element is at position 4 in the array, then the front will point to position 3. When the circular queue is created, then both front and rear point to index 1.Also, we can conclude that the circular queue is empty in case both front and rear point to the same index. Figure depicts a circular queue



**Enqueue(value) - Inserting va lue into the Circular Queue**

Step 1: Check whether queue is FULL.
If ((REAR == SIZE-1 && FRONT == 0) || (FRONT == REAR+1))
Write “Queue is full ” and Exit
Step 2: If ( rear == SIZE - 1 && front != 0 ) then:
SET REAR := -1.
Step 4: SET REAR = REAR +1
Step 5: SET QUEUE[REAR] = VALUE
Step 6: Exit and check 'front == -1' if it is TRUE, then set front=0

**DeQueue() - De leting a value from the Circular Queue**

```
Step 1: Check whether queue is EMPTY?
If (FRONT == -1 && REAR == -1)
Write “Queue is empty” and Exit.

Step 2: DISPLAY QUEUE[FRONT]
Step 3: SET FRONT := FRONT +1 .
Step 4: If( FRONT = SIZE, then:
SET FRONT := 0.
Step 5: Exit
```

# UNIT-III

## Tree - Terminology

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. Tree is a very popular non-linear data structure used in wide range of applications. A tree data structure can be defined as follows...

DEF:Tree is a non linear data structure which organizes the data in a hierarchical manner.

**Key properties of Tree**

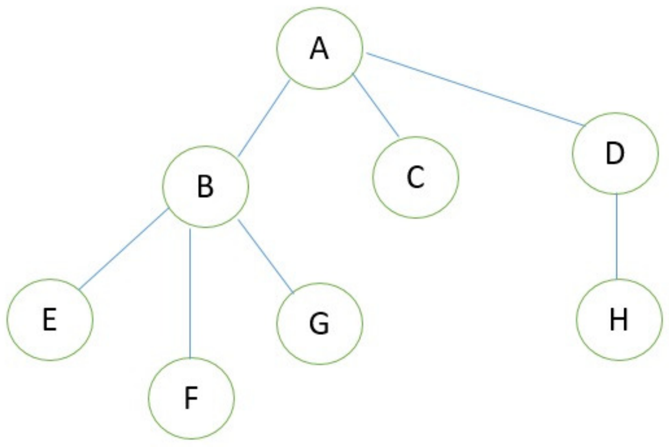
A tree T is represented by nodes and edges, which includes:

1. **T** is empty (called null or empty tree).
2. **T** has a left subtree and right subtree

**Terminology**

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

In the figure **A** is the root Node



**Edge** :- 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.

**Parent** :- Node which is predecessor of any node is called as PARENT NODE i.e. node which has branch from it to any other node is called as parent node. In the above figure node **B** is the parent node of **E**, **F**, and **G** node and **E**, **F**, and **G** are called children of **B**.

**Child** :- Node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node.

**Siblings** :- Nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes. Children of same parent are called Sibling.

**Leaf** :- Node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. In the figure **E,F,C,G,C,H** are the leaf nodes

**Level** :- Root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on… In the above figure, the **node A** is at level 0, the **node B**, **C**, **D**are at level 1, the **nodes E**, **F**, **G**, **H** are at level 2.

**Tree Height** :- Number of edges in the longest path from the root to the leaf node.
**the height of the tree is 3**

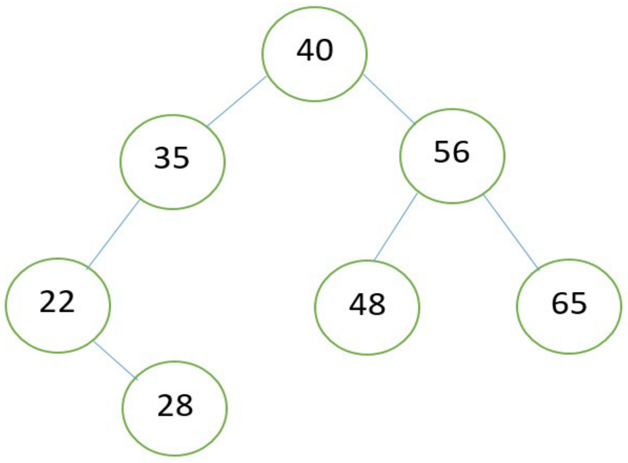
**Predecessor**:-While displaying the tree, if some particular nodes previous to some other nodes than that node is called the predecessor of the other node. In our example, **the node E is predecessor of node B**.

**Successor**:-The node which occurs next to some other node is called successor of that node. In our example, **B is successor of F and G**.

**Internal Nodes**:-The nodes in the tree which are other than leaf nodes and the root node are called as internal nodes. These nodes are present in between root node and leaf nodes in the tree that's why these nodes are called as internal node. Here, **B and D are internal nodes**.

**Degree of tree** :-The maximum degree of the node in the tree is called the degree of the tree. Here, the **degree of tree is 3**.

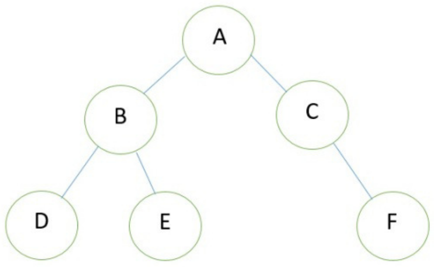
**binary tree**
A binary tree is a finite set of nodes that is either empty or consist a root node and two disjoint binary trees called the left subtree and the right subtree.
In other words, a binary tree is a non-linear data structure in which each node has maximum of two child nodes. The tree connections can be called as branches
A binary tree is a special case of an ordered tree, where ***k*** is 2.



**1) Complete Binary Tree**

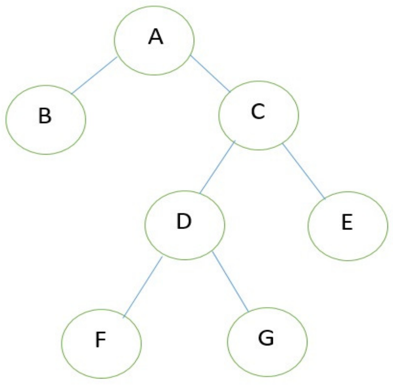
A binary tree **T** is said to be complete binary tree if -

1. All its levels, except possibly except possibly the last, have the maximum number of nodes and
2. All the nodes at the last level appears as far left as possible.



**2) Strictly Binary Tree**

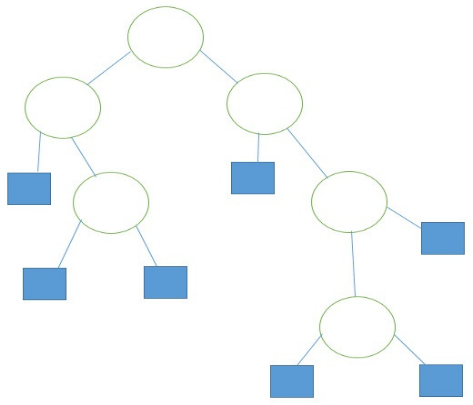
When every non leaf node in a binary tree is filled with left and right subtrees, the tree is called a strictly binary tree.



**3) Extended Binary Tree**

The binary tree that is extended with zero (no nodes) or left or right node or both the nodes is called an extended binary tree or a 2- tree.

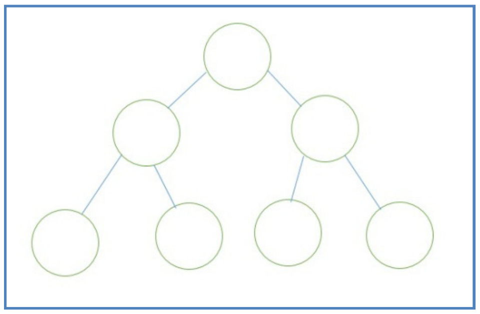
The extended binary tree is shown in figure above. The extended nodes are indicated by square box. Maximum nodes in the binary tree have one child (nodes) shown at either left or right by adding one or more children, the tree can be extended. The extended binary tree is very useful for representation of algebraic expressions. The left and right child nodes denote operands and parent node indicates operator.



**4) Full Binary Tree**

A Binary Tree is full binary tree if and only if -

1. Each non- leaf node has exactly two child nodes.
2. All leaf nodes are at the same level.



**Properties of Binary trees**

1) The maximum number of nodes at level 'l' of a binary tree is $2^{l-1}$.
2) Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$.
3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is $\lceil \log_2(N+1) \rceil$
4) A Binary Tree with L leaves has at least $\lceil \log_2 L \rceil + 1$ levels
5) In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

**Representation of Binary Trees**

Let **T** be a Binary Tree. There are two ways of representing **T** in the memory as follow

1. **Sequential Representation of Binary Tree.**

2. **Link Representation of Binary Tree.**

**1) Linked Representation of Binary Tree**

Consider a Binary Tree T. T will be maintained in memory by means of a linked list representation which uses three parallel arrays; INFO, LEFT, and RIGHT pointer variable ROOT as follows. In Binary Tree each node N of Twill correspond to a location k such that

- LEFT [k] **contains the location of the left child of node** N**.**
- INFO [k] **contains the data at the node** N**.**
- RIGHT [k] **contains the location of right child of node** N**.**

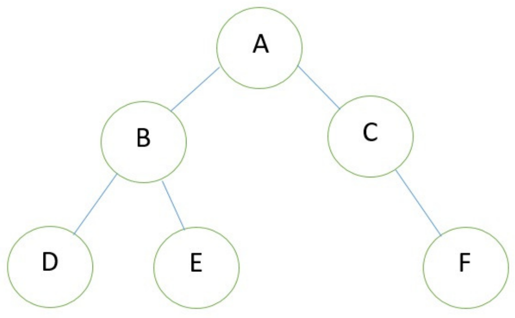
**Representation of a node:**

| | | |
|---|---|---|
| LEFT [k] | INFO [k] | RIGHT [k] |

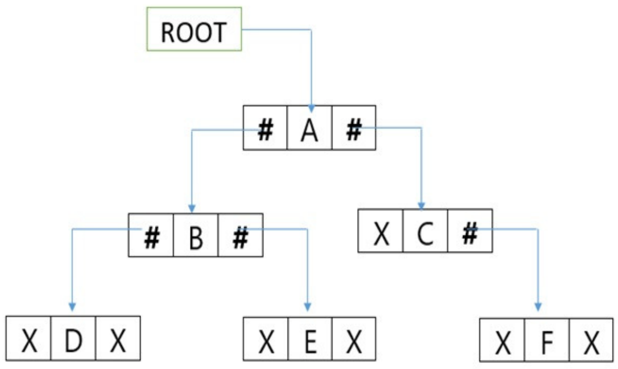
In this representation of binary tree root will contain the location of the root **R** of **T**. If any one of the subtree is empty, then the corresponding pointer will contain the null value if the tree **T** itself is empty, the **ROOT** will contain the null value.

**Example**
Consider the binary tree **T** in the figure. A schematic diagram of the linked list representation of **T** appears in the following figure. Observe that each node is pictured with its three fields, and that the empty subtree is pictured by using x for null entries.

**Binary Tree**



**Linked Representation of the Binary Tree**



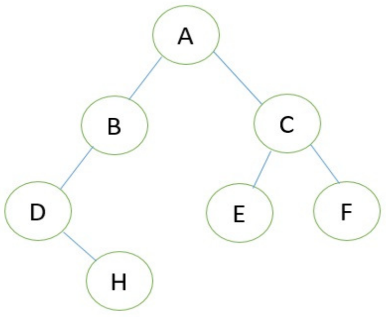
**2) Sequential representation of Binary Tree**

Let us consider that we have a tree **T**. let our tree **T** is a binary tree that us complete binary tree. Then there is an efficient way of representing **T** in the memory called the sequential representation or array representation of **T**. This representation uses only a linear array TREE as follows:

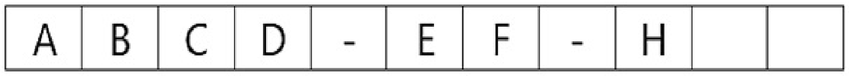
- The root **N** of **T** is stored in **TREE [1]**.
- If a node occupies **TREE [k]** then its left child is stored in **TREE [2 * k]** and its right child is stored into **TREE [2 * k + 1]**.

**For Example:**

Consider the following Tree:



Its sequential representation is as follows



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

**Binary Tree traversals**

**Tree traversal** is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traversal. There are basically three traversal techniques for a binary tree that are,

- Preorder traversal
- Inorder traversal
- Postorder traversal

**1) Preorder traversal**

To **traverse a binary tree in preorder**, following operations are carried out:

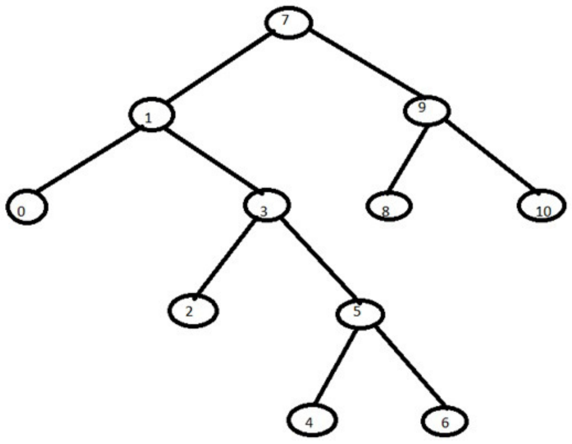
1. Visit the root.
2. Traverse the left sub tree of root.
3. Traverse the right sub tree of root.

**Note:** Preorder traversal is also known as NLR traversal.

**Algorithm:**

```
Algorithm preorder(t)
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Visit(t);
                Preorder(t->lchild);
                Preorder(t->rchild);
        }
}
```

**Example:** Let us consider the given binary tree,



Therefore, the preorder traversal of the above tree will be: **7,1,0,3,2,5,4,6,9,8,10**

**2) Inorder traversal**

To traverse a binary tree in inorder traversal, following operations are carried out:

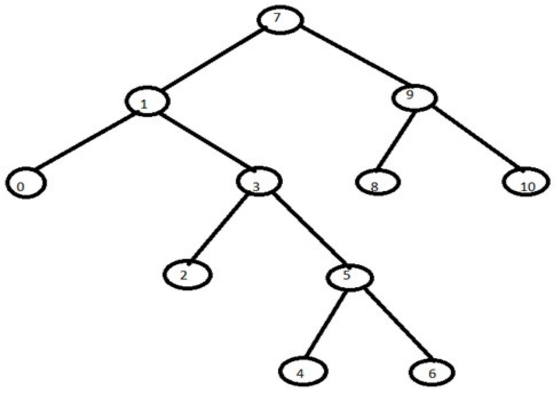
1. Traverse the left most sub tree.
2. Visit the root.
3. Traverse the right most sub tree.

**Note:** Inorder traversal is also known as LNR traversal.

**Algorithm:**

```
Algorithm inorder(t)

/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Inorder(t->lchild);
                Visit(t);
                Inorder(t->rchild);
        }
}
```

**Example:** Let us consider a given binary tree.



Therefore the inorder traversal of above tree will be: **0,1,2,3,4,5,6,7,8,9,10**

**3) Postorder traversal**

To traverse a binary tree in postorder traversal, following operations are carried out:

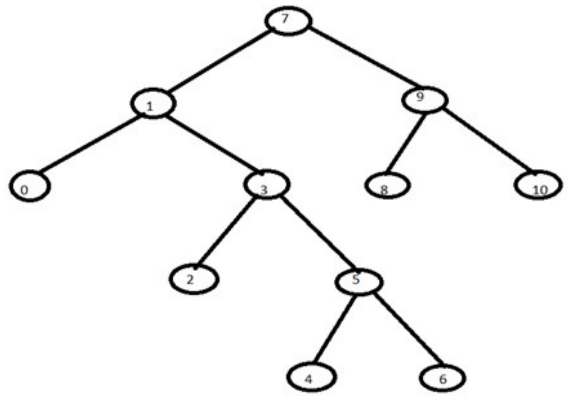
1. Traverse the left sub tree of root.
2. Traverse the right sub tree of root.
3. Visit the root.

**Note:** Postorder traversal is also known as LRN traversal.

**Algorithm:**

```
Algorithm postorder(t)

/*t is a binary tree .Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Postorder(t->lchild);
                Postorder(t->rchild);
                Visit(t);
        }
}
```

**Example:** Let us consider a given binary tree.



Therefore the postorder traversal of the above tree will be: **0,2,4,6,5,3,1,8,10,9,7**

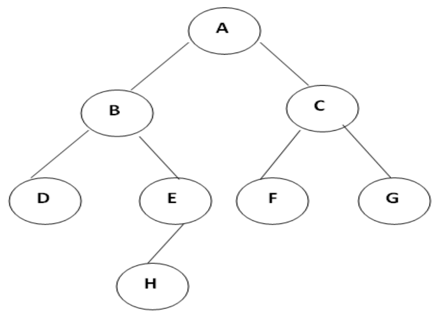
**Threaded Binary Tree**

A binary tree can be represented by using array representation or linked list representation. When a binary tree is represented using linked list representation. If any node is not having a child we use a NULL pointer. These special pointers are threaded and the binary tree having such pointers is called a threaded binary tree. Thread in a binary tree is represented by a dotted line. In linked list representation of a binary tree, they are a number of a NULL pointer than actual pointers. This NULL pointer does not play any role except indicating there is no child. The **threaded binary tree** is proposed by A.J Perlis and C Thornton. There are three ways to thread a binary tree.

1. In the in order traversal When The right NULL pointer of each leaf node can be replaced by a thread to the successor of that node then it is called a right thread, and the resultant tree called a right threaded tree or right threaded binary tree.
2. When The left NULL pointer of each node can be replaced by a thread to the predecessor of that node under in order traversal it is called left thread and the resultant tree will call a left threaded tree.
3. In the in order traversal, the left and the right NULL pointer can be used to point to predecessor and successor of that node respectively. then the resultant tree is called a fully threaded tree.

In the threaded binary tree when there is only one thread is used then it is called as one way threaded tree and when both the threads are used then it is called the two way threaded tree. The pointer point to the root node when If there is no in-order predecessor or in-order successor.

**Consider the following binary tree:**



The in-order traversal of a given tree is **D B H E A F C G**. Right threaded binary tree for a given tree is shown below:

Advantages of Thread Binary Tree

Non-recursive pre-order, in-order and post-order traversal can be implemented without a stack**.**

Disadvantages of Thread Binary Tree

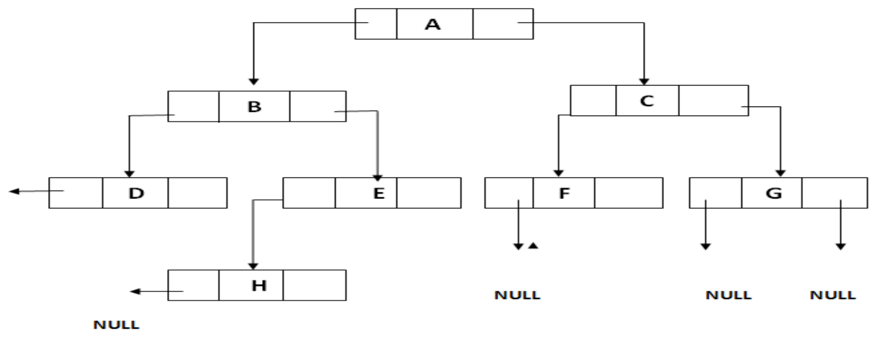
1. Insertion and deletion operation becomes more difficult.
2. Tree traversal algorithm becomes difficult.
3. Memory required to store a node increases. Each node has to store the information whether the links is normal links or threaded links.

**Types of Threaded of Binary Tree**

There are three types of Threaded Binary Tree,

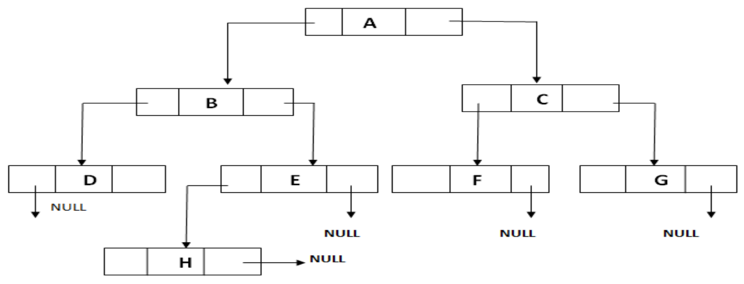
**1) Right Threaded Binary Tree**

Right threaded binary tree for a given tree is shown:



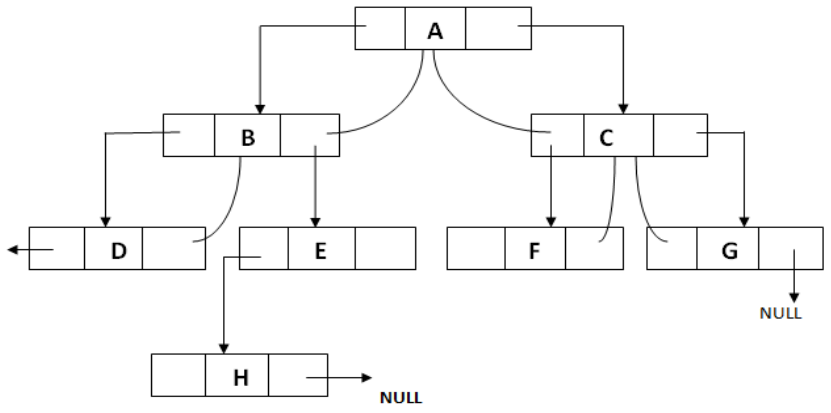
## 2) Left Threaded Binary Tree

Left threaded binary tree for a given tree is shown:



## 3) Fully Threaded Binary Tree

Fully threaded binary tree for a given tree is shown:



## PRIORITY QUEUE

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

In priority queue every element is associated with some priority. Normally the priorities are specified using numerical values. In some cases lower values indicate high priority and in some cases higher values indicate high priority.In priority queues elements are processed according to their priority but not according to the order they are entered into the queue.

**Ex:** let P be a priority queue with three elements a, b, c whose priority factors are 2,1,1 respectively. Here, larger the number, higher is the priority accorded to that element.When a new element d with higher priority 4 is inserted, d joins at the head of the queue superseding the remaining elements. When elements in the queue have the same priority, then the priority queue behaves as an ordinary queue following the principle of FIFO amongst such elements.

The working of a priority queue may be likened to a situation when a file of patients waits for their turn in a queue to have an appointment with a doctor. All patients are accorded equal priority and follow an FCFS scheme by appointments. However, when a patient with bleeding injuries is brought in, he/she is accorded higher priority and is immediately moved to the head of the queue for immediate attention by the doctor. This is priority queue at work. There are two types of priority queues they are as follows...

1. Max Priority Queue
2. Min Priority Queue

**Max Heap Data structure**

Heap data structure is a specialized binary tree based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on thier values. A heap data structure some times also called as Binary Heap.

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

1. **Max Heap**
2. **Min Heap**

Every heap data structure has the following properties...

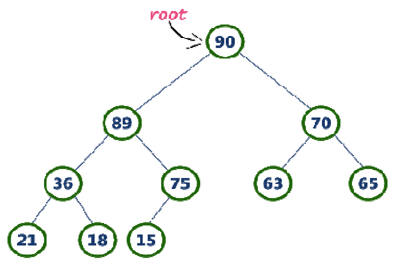
**Property #1 (Ordering):** Nodes must be arranged in an order according to their values based on Max heap or Min heap.

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

**Max Heap**

Max heap data structure is a specialized full binary tree data structure. In a max heap nodes are arranged based on node value**.**

Example



Above 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**

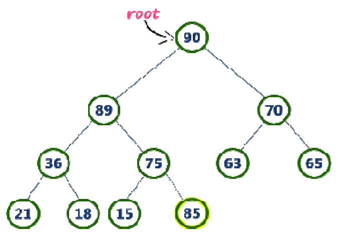
**Finding Maximum Value Operation in Max Heap**

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

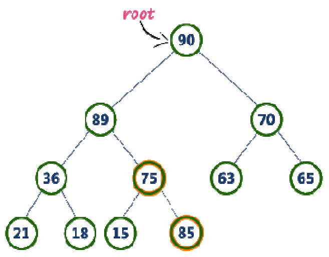
- **Step 1 -** Insert the **newNode** as **last leaf** from left to right.
- **Step 2 -** Compare **newNode value** with its **Parent node**.
- **Step 3 -** If **newNode value is greater** than its parent, then **swap** both of them.
- **Step 4 -** Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

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

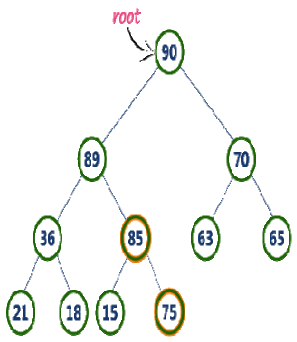
- **Step 1 -** Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



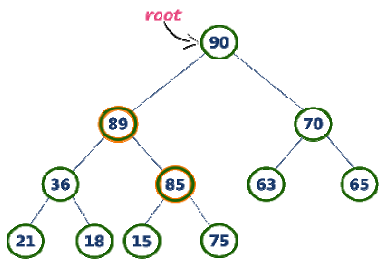
- **Step 2 -** Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



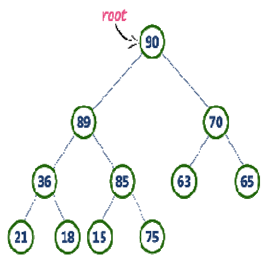
- **Step 3 -** Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After swapping, max heap is as follows...



- **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**

In a max heap, deleting last node is very simple as it does not disturb max heap properties.

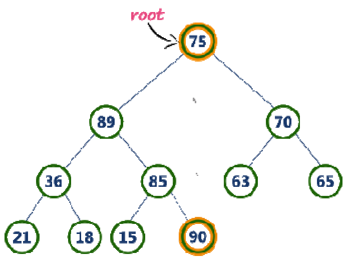
Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete root node from a max heap...

- **Step 1 - Swap** the **root** node with **last** node in max heap
- **Step 2 - Delete** last node.
- **Step 3 -** Now, compare **root value** with its **left child value**.
- **Step 4 -** If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto **Step 6**
- **Step 5 -** If **left child value is larger** than its **right sibling**, then **swap root** with **left child** otherwise **swap root** with its **right child**.
- **Step 6 -** If **root value is larger** than its left child, then compare **root value** with its **right child** value.
- **Step 7 -** If **root value is smaller** than its **right child**, then **swap rootwith right child** otherwise **stop the process**.
- **Step 8 -** Repeat the same until root node fixes at its exact position.

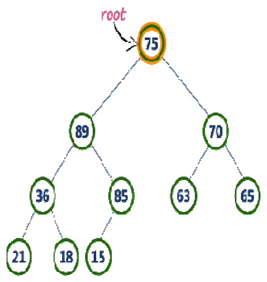
Example

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

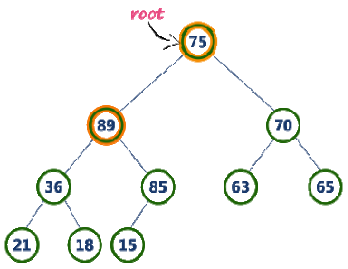
- **Step 1 - Swap** the **root node (90)** with **last node 75** in max heap. After swapping max heap is as follows...



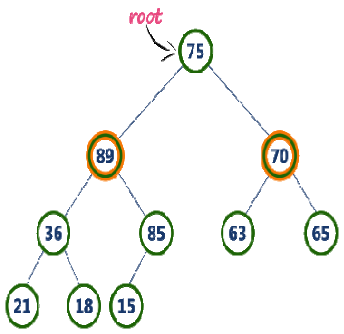
- **Step 2 - Delete** last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



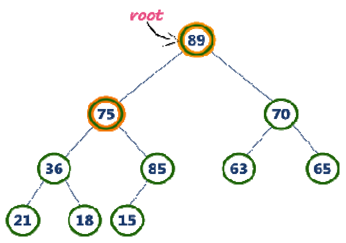
- **Step 3 -** Compare **root node (75)** with its **left child (89)**.



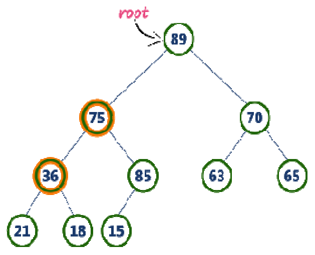
Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



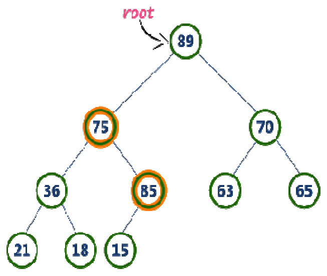
- **Step 4 -** Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



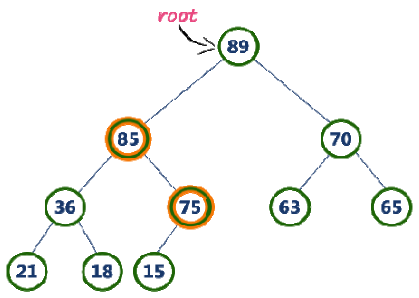
- **Step 5 -** Now, again compare **75** with its **left child (36)**.



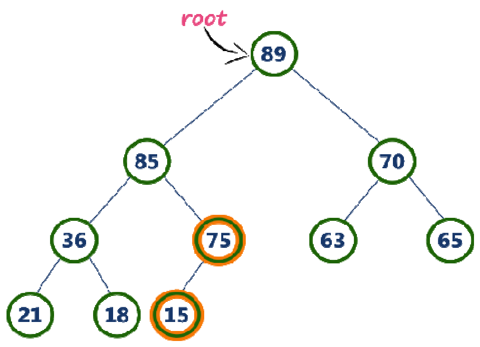
Here, node with value **75** is larger than its left child. So, we compare node **75** with its right child **85**.



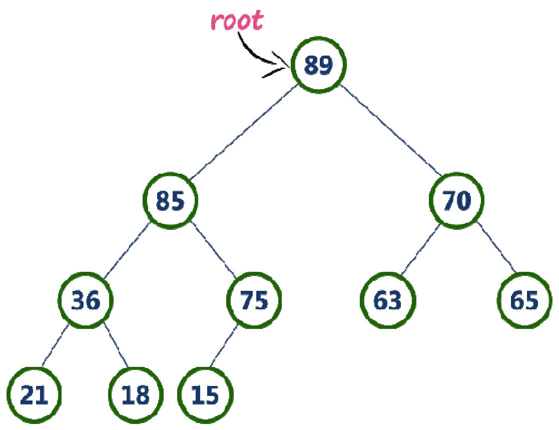
- **Step 6 -** Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...



- **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...



# UNIT-IV

Searching is the process of finding element in a given list. In this process we check item is available in given list or not.

**Type of searching**

1. Internal search
2. External search

**Internal Search:** In this search, Searching is performed on **main or primary memory**.
**External Search:** In this search, Searching is performed in **secondary memory**.

**Complexity Analysis**

The Complexity analysis is used to determine, the algorithm will take amount of resources (such as time and space) necessary to execute it.
There are two types of complexities: 1) **Time Complexity** and 2) **Space Complexity**.

**Searching Techniques**

There are basically three types of searching techniques:

- Linear or sequential search
- Binary search
- Interpolation search

**Linear/ Sequential Search**

Linear/Sequential searching is a searching technique to find an item from a list until the particular item not found or list not reached at end. We start the searching from 0th index to N-1 index in a sequential manner, if particular item found, returns the position of that item otherwise return failure status or -1.

**Example**

Consider the following list of elements and the element to be searched...

**LINEAR_SEARCH (LIST, N, ITEM, LOC, POS)**

```
In this algorithm, LIST is a linear array with N elements,
and ITEM is a given item to be searched. This algorithm finds
the location LOC of ITEM into POS in LIST, or sets POS := 0,
if search is unsuccessfull.
1.      Set POS  :=  0;
2.      Set LOC  :=  1;
3.      Repeat STEPS a and b while LOC <= N
   a.   If ( LIST[LOC] = ITEM) then
      i.          SET POS := LOC;  [get position of item]
      ii.         return;
   b.   Otherwise
      i.          SET LOC := LOC + 1 ; [increment counter]
4.      [END OF LOOP]
```

5. return

**Complexity of Linear search**

The complexity of search algorithm is based on number of comparisons C, between ITEM and LIST [LOC]. We seek C (n) for the worst and average case, where n is the size of the list.

**Worst Case:** The worst case occurs when ITEM is present at the last location of the list, or it is not there at al. In either situation, we have

**C (n) = n**

Now, C (n) = n is the worst case complexity of linear or sequential search algorithm.

**Average case:** In this case, we assume that ITEM is there in the list, and it can be present at any position in the list. Accordingly, the number of comparisons can be any of the numbers 1, 2, 3, 4 …n. and each number occurs with probability p = 1/n. Then,

**C (n) = n/2**

**Binary Search**

It is special type of search work on sorted list only. During each stage of our procedure, our search for **ITEM** is reduced to a restricted segment of elements in **LIST** array. The segment starts from index LOW and spans to **HIGH**.

**LIST [LOW], LIST [LOW+1], LIST [LOW+2], LIST [LOW+ 3]….. LIST[HIGH]**

The **ITEM** to be searched is compared with the middle element **LIST[MID]** of segment, where **MID** obtained as:

**MID = ( LOW + HIGH )/2**

We assume that the **LIST** is sorted in ascending order. There may be three results of this comparison:

1. If **ITEM = LIST[MID]**, the search is successful. The location of the ITEM is **LOC := MID**.
2. If **ITEM < LIST[MID]**, the ITEM can appear only in the first half of the list. So, the segment is restricted by modifying **HIGH = MID – 1**.
3. If **ITEM > LIST[MID]**, the ITEM can appear only in the second half of the list. So, the segment is restricted by modifying **LOW = MID + 1**.

Initially, **LOW** is the start index of array, and HIGH is the last index of array. The comparison goes on. With each comparison, low and high approaches near to each other. The loop will continue till low<high.

Consider the following list of elements and the element to be searched...

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 15 | 18 | 28 | 35 | 58 | 63 | 87 | 98 | 99 |

Let the item to be searched is: 15

Step1: Initially

1. List[low] = list[0] = 12
2. List[high] = list[9] = 99
3. Mid= (low + high)/2 = (0+9)/2 = 4
4. List[mid]= list[4] = 35

Step2: Since item (15) < list [mid] (35); high = mid -1 = 3

1. List[low] = list[0] = 12
2. List[high] = list[3] = 28
3. Mid =(low + high)/2 = (0+3)/2 = 1
4. List [mid] = list [1] = 15.

Step3: Since item (15) = list [mid] (15);
It is success, the location 1 is returned.

**BINARY SEARCH (LIST, N, ITEM, LOC, LOW, MID, HIGH)**
Here LIST is a sorted array with size N, ITEM is the given information.
The variable LOW, HIGH and MID denote, respectively,
the beginning, end and the middle of segment of LIST.
The algorithm is used to find the location LOC of ITEM in LIST array.
If ITEM not there LOC is NULL.

```
1.   Set LOW := 1; LOW := N;
2.   Repeat step a  to d while LOW <= HIGH and ITEM != LIST(MID)
     a. Set MID := (LOW + HIGH)/2
     b. If ITEM = LIST(MID), then
        i.    Set LOC := MID
        ii.   Return
     c. If ITEM < LIST(MID) then
        i.    Set HIGH := MID - 1;
     d. Otherwise
        i.    Set LOW := MID + 1;
3.   SET  LOC := NULL
4.   return
```

**Complexity of Binary search**
The complexity measured by the number f(n) of comparisons to locate ITEM in LIST where LIST contains n elements. Each comparison reduces the segment size in half. Hence, we require at most f(n) comparisons to locate ITEM, where:
**$2^c >= n$**
Approximately, the time complexity is equal to $\log_2 n$. It is much less than the time complexity of linear search.

**Hashing (Hash table, Hash functions and its characteristics)**
**Hashing**
There are many possibilities for representing the dictionary and one of the best methods for representing is **hashing**. Hashing is a type of a solution which can be used in almost all situations. Hashing is a technique which uses less key comparisons and searches the element in **O(n)** time in the worst case and in an average case it will be done in **O(1)** time. This method generally used the hash functions to map the keys into a table, which is called a **hash table**.

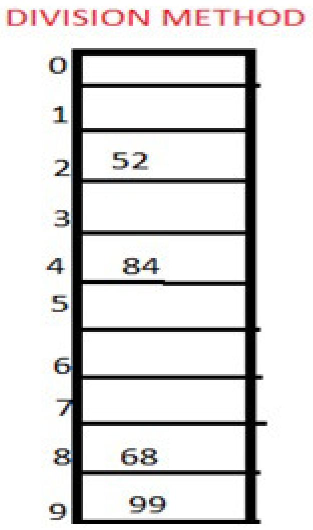
**1) Hash table**
Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisons and then searching time is dependent upon the size of the hash table.

**2) Hash function**
Hash function is a function which is applied on a key by which it produces an integer, which can be used as an address of hash table. Hence one can use the same hash function for accessing the data from the hash table. In this the integer returned by the hash function is called hash key.

**Types of hash function**
There are various types of hash function which are used to place the data in a hash table,

**1. Division method**
In this the hash function is dependent upon the remainder of a division. For example:-if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.
**Then:**
h(key)=record% table size.

    2=52%10
    8=68%10
    9=99%10
    4=84%10



**2. Mid square method**
In this method firstly key is squared and then mid part of the result is taken as the index. For example: consider that if we want to place a record of 3101 and the size of table is 1000. So 3101*3101=9616201 i.e. **h (3101) = 162 (middle 3 digit)**

**3. Digit folding method**

In this method the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key. For example: consider a record of 12465512 then it will be divided into parts i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

H(key)=124+655+12
=791

**Characteristics of good hashing function**

1. The hash function should generate different hash values for the similar string.
2. The hash function is easy to understand and simple to compute.
3. The hash function should produce the keys which will get distributed, uniformly over an array.
4. A number of collisions should be less while placing the data in the hash table.
5. The hash function is a perfect hash function when it uses all the input data.

**Collision**

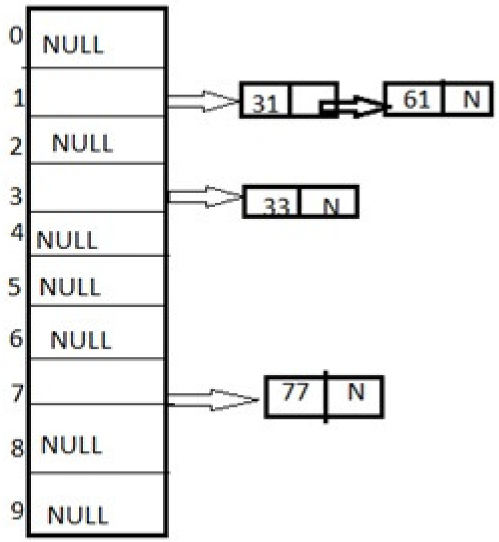
It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

**Collision resolution technique**

If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below.

**1) Chaining**

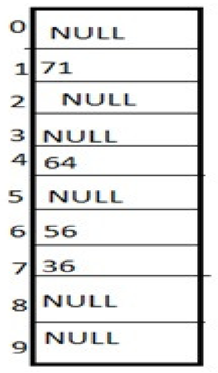
It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.



**Example:** Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are 31,33,77,61. In the above diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

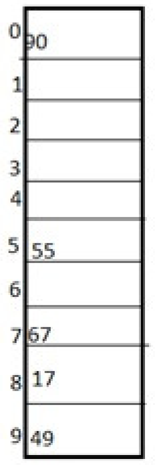
**2) Linear probing**
It is very easy and simple method to resolve or to handle the collision. In this collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

**Example:** Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table size. Consider that following keys are to be inserted that are 56,64,36,71.



In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

**3) Quadratic probing**
This is a method in which solving of clustering problem is done. In this method the hash function is defined by the H(key)=(H(key)+x*x)%table size. Let us consider we have to insert following elements that are:-67, 90,55,17,49.



In this we can see if we insert 67, 90, and 55 it can be inserted easily but at case of 17 hash function is used in such a manner that :-(17+0*0)%10=17 (when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so (17+1*1)%10=8.in this case bucket 8 is empty hence we will place 17 at index 8.

**4) Double hashing**

It is a technique in which two hash function are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

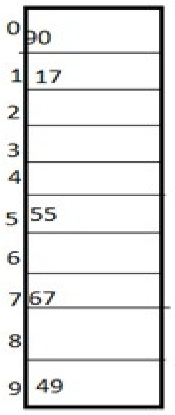
1. It must never evaluate to zero.
2. Must sure about the buckets, that they are probed.

The hash functions for this technique are:

H1(key)=key % table size
H2(key)=P-(key mod P)

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

**Example:** Let us consider we have to insert 67, 90,55,17,49.



In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 again the bucket is full and in this case we have to use the second hash function which is H2(key)=P-(key mode P) here p is a prime number which should be taken smaller than the hash table so value of p will be the 7.

i.e. H2(17)=7-(17%7)=7-3=4 that means we have to take 4 jumps for placing the 17. Therefore 17 will be placed at index 1.

**Sorting:**

Sorting is nothing but arranging the data in ascending or descending order.
The term **sorting** came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

**Sorting** arranges data in a sequence which makes searching easier.

The techniques of sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

**Internal Sorting**: If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

**External Sorting:** When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

**Complexity of sorting algorithm**

The complexity of sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted. The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems. The most noteworthy of these considerations are:

- The length of time spent by the programmer in programming a specific sorting program
- Amount of machine time necessary for running the program
- The amount of memory necessary for running the program

**The Efficiency of Sorting Techniques**

To get the amount of time required to sort an array of 'n' elements by a particular method, the normal approach is to analyze the method to find the number of comparisons (or exchanges) required by it. Most of the sorting techniques are data sensitive, and so the metrics for them depends on the order in which they appear in an input array.

Various sorting techniques are analyzed in various cases and named these cases as follows:

- Best case
- Worst case
- Average case

Hence, the result of these cases is often a formula giving the average time required for a particular sort of size 'n.' Most of the sort methods have time requirements that range from O(nlog n) to O($n^2$).

Here are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

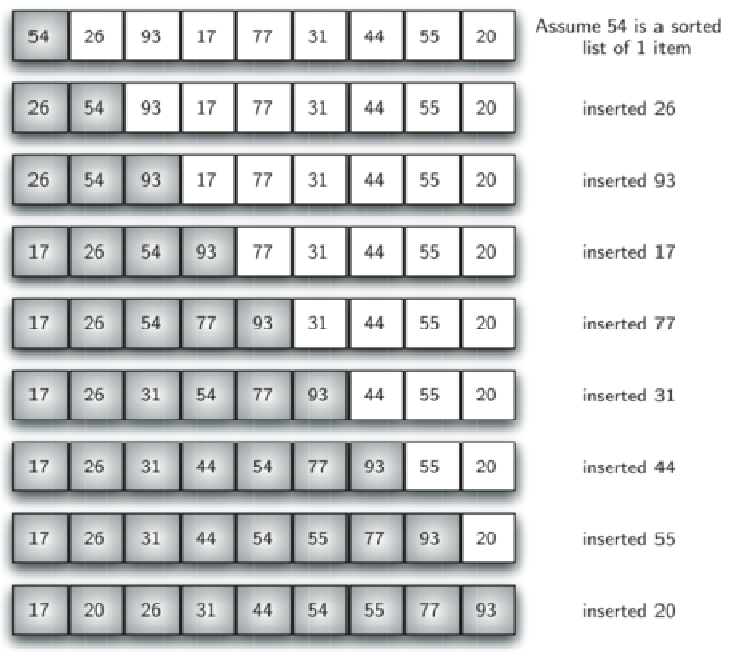
- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Heap Sort

**Insertion Sort**

**i**nsertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Start with a sorted list of 1 element on the left, and N-1 unsorted items on the right. Take the first unsorted item (element #2) and insert it into the sorted list, moving elements as necessary. We now have a sorted list of size 2, and N -2 unsorted elements. Repeat for all elements.



**Time Complexity:**

In worst case,each element is compared with all the other elements in the sorted array.
For N elements, there will be N2 comparisons. Therefore, the time complexity is O(N2).

**Selection Sort Algorithm**

Selection sort algorithm starts by compairing first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, leave the elements as it is. Then, again first element and third element are compared and swapped if necessary. This process goes on until first and last element of an array is compared. This completes the first step of selection sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, comparison starts from second element because after first step, the required number is automatically placed at the first (i.e, In case of sorting in ascending order, smallest element will be at first and in case of sorting in descending order, largest element will be at first.). Similarly, in third step, comparison starts from third element and so on..

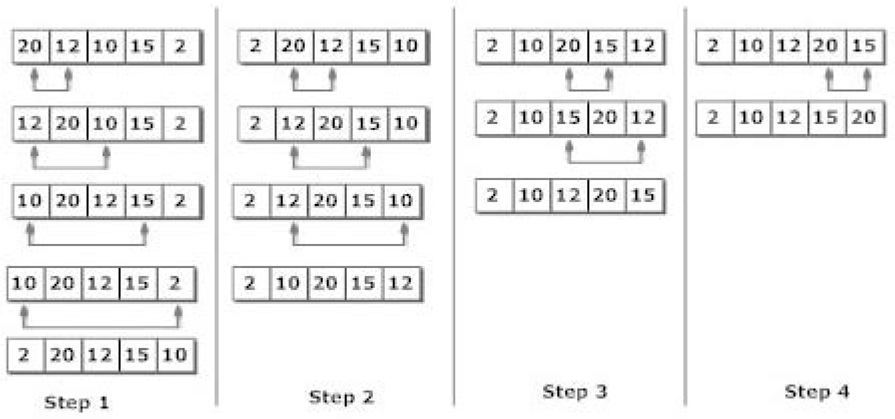


Figure: Selection Sort

**Radix sort**
Radix sort processes the elements the same way in which the names of the students are sorted according to their alphabetical order. There are 26 radix in that case due to the fact that, there are 26 alphabets in English. In the first pass, the names are grouped according to the ascending order of the first letter of names.

In the second pass, the names are grouped according to the ascending order of the second letter. The same process continues until we find the sorted list of names. The bucket are used to store the names produced in each pass. The number of passes depends upon the length of the name with the maximum letter.

In the case of integers, radix sort sorts the numbers according to their digits. The comparisons are made among the digits of the number from LSB to MSB. The number of passes depend upon the length of the number with the most number of digits.

**Consider the array of length 6 given below. Sort the array by using Radix sort.**
A = {10, 2, 901, 803, 1024}
**Pass 1: (Sort the list according to the digits at 0's place)**
10, 901, 2, 803, 1024.
**Pass 2: (Sort the list according to the digits at 10's place)**
02, 10, 901, 803, 1024
**Pass 3: (Sort the list according to the digits at 100's place)**
02, 10, 1024, 803, 901.
**Pass 4: (Sort the list according to the digits at 1000's place)**
02, 10, 803, 901, 1024

**Therefore, the list generated in the step 4 is the sorted list, arranged from radix sort.**

**Algorithm**

- **Step 1**:Find the largest number in ARR as LARGE

- **Step 2**: [INITIALIZE] SET NOP = Number of digits
  in LARGE
- **Step 3**: SET PASS =0
- **Step 4**: Repeat Step 5 while PASS <= NOP-1
- **Step 5**: SET I = 0 and INITIALIZE buckets
- **Step 6**:Repeat Steps 7 to 9 while I<n-1< li=""></n-1<>
- **Step 7**: SET DIGIT = digit at PASSth place in A[I]
- **Step 8**: Add A[I] to the bucket numbered DIGIT
- **Step 9**: INCREMENT bucket count for bucket
  numbered DIGIT
  [END OF LOOP]
- **Step 10**: Collect the numbers in the bucket
  [END OF LOOP]
- **Step 11**: END

**Quick Sort**
Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

**Algorithm**

**PARTITION (ARR, BEG, END, LOC)**

- **Step 1**: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
- **Step 2**: Repeat Steps 3 to 6 while FLAG =
- **Step 3**: Repeat while ARR[LOC] <=ARR[RIGHT]
  AND LOC != RIGHT
  SET RIGHT = RIGHT - 1
  [END OF LOOP]
- **Step 4**: IF LOC = RIGHT
  SET FLAG = 1
  ELSE IF ARR[LOC] > ARR[RIGHT]
  SWAP ARR[LOC] with ARR[RIGHT]
  SET LOC = RIGHT
  [END OF IF]
- **Step 5**: IF FLAG = 0
  Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
  SET LEFT = LEFT + 1
  [END OF LOOP]
- **Step 6**:IF LOC = LEFT
  SET FLAG = 1
  ELSE IF ARR[LOC] < ARR[LEFT]
  SWAP ARR[LOC] with ARR[LEFT]

```
    SET LOC = LEFT
    [END OF IF]
    [END OF IF]
```

- **Step 7**: [END OF LOOP]
- **Step 8**: END

**QUICK_SORT (ARR, BEG, END)**

- **Step 1**: IF (BEG < END)
```
  CALL PARTITION (ARR, BEG, END, LOC)
  CALL QUICKSORT(ARR, BEG, LOC - 1)
  CALL QUICKSORT(ARR, LOC + 1, END)
  [END OF IF]
```
- **Step 2**: END

**Merge sort**

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

**Algorithm**

**Step 1**: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0

```
Step 2: Repeat while (I <= MID) AND (J<=END)
IF ARR[I] < ARR[J]
SET TEMP[INDEX] = ARR[I]
SET I = I + 1
ELSE
SET TEMP[INDEX] = ARR[J]
SET J = J + 1
[END OF IF]
SET INDEX = INDEX + 1
[END OF LOOP]
Step 3: [Copy the remaining
elements of right sub-array, if
any]
IF I > MID
Repeat while J <= END
SET TEMP[INDEX] = ARR[J]
SET INDEX = INDEX + 1, SET J = J + 1
[END OF LOOP]
[Copy the remaining elements of
left sub-array, if any]
ELSE
Repeat while I <= MID
SET TEMP[INDEX] = ARR[I]
SET INDEX = INDEX + 1, SET I = I + 1
[END OF LOOP]
[END OF IF]
```

**Step 4**: [Copy the contents of TEMP back to ARR] SET K = 0

```
Step 5: Repeat while K < INDEX
SET ARR[K] = TEMP[K]
SET K = K + 1
[END OF LOOP]
```

**Step 6**: Exit

**MERGE_SORT(ARR, BEG, END)**

```
Step 1: IF BEG < END
SET MID = (BEG + END)/2
CALL MERGE_SORT (ARR, BEG, MID)
CALL MERGE_SORT (ARR, MID + 1, END)
MERGE (ARR, BEG, MID, END)
[END OF IF]
```

**Step 2**: END

**Heap sort**

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.

**Algorithm**

```
HEAP_SORT(ARR, N)
Step 1: [Build Heap H]
Repeat for i=0 to N-1
CALL INSERT_HEAP(ARR, N, ARR[i])
[END OF LOOP]

Step 2: Repeatedly Delete the root element
Repeat while N > 0
CALL Delete_Heap(ARR,N,VAL)
SET N = N+1
[END OF LOOP]

Step 3: END
```

**Comparison of Sorting methods.**

The comparison of sorting methods is performed based on the **Time complexity** and **Space complexity** of sorting methods. The following table provides the time and space complexities of sorting methods. These Time and Space complexities are defined for **'n'** number of elements.

| Sorting technique | Time complexity | | |
|---|---|---|---|
| | Best case | Average case | Worst case |
| Insertion sort | $O(n^2)$ | $O(n^2)$ | O(n) |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Quick sort | $O(n^2)$ | O(n log n) | O(n log n) |
| Merge sort | O(n log n) | O(n log n) | O(n log n) |
| Heap sort | O(n log n) | O(n log n) | O(n log n) |

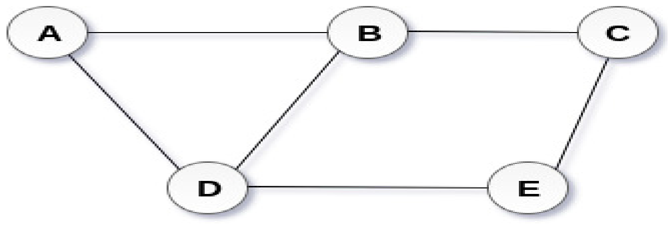
# UNIT-V

## Graph

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

**Definition**

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.
A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



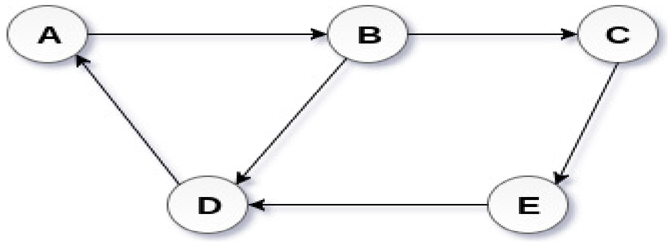
Undirected Graph

## Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Directed Graph

## Graph Terminology

### Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

### Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

### Simple Path

If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

### Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

### Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

### Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

### Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

### Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

### Loop

An edge that is associated with the similar end points can be called as Loop.

### Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

### Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

## Graph Representation

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

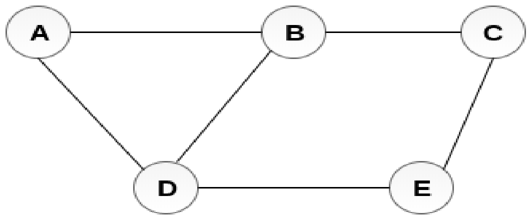
There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.

**1. Sequential Representation**

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension **n x n**.

An entry $M_{ij}$ in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between $V_i$ and $V_j$.

An undirected graph and its adjacency matrix representation is shown in the following figure.



Undirected Graph

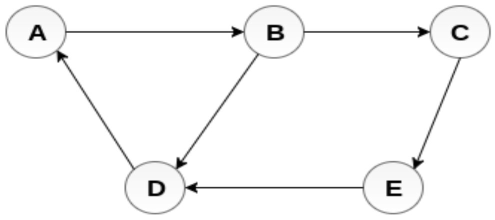
$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix

in the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry $A_{ij}$ will be 1 only when there is an edge directed from $V_i$ to $V_j$.

A directed graph and its adjacency matrix representation is shown in the following figure.



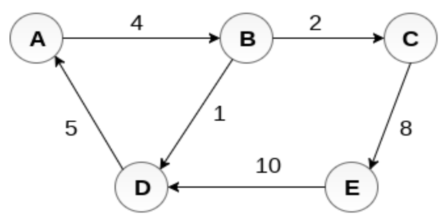
Directed Graph

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

Adjacency Matrix

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non-zero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



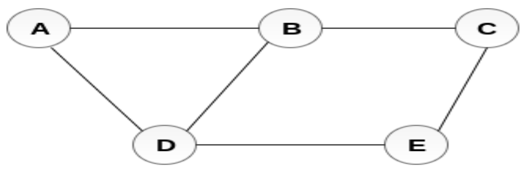
**Weighted Directed Graph**

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

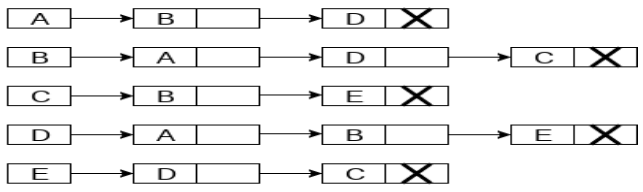
**Adjacency Matrix**

**Linked Representation**
In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.



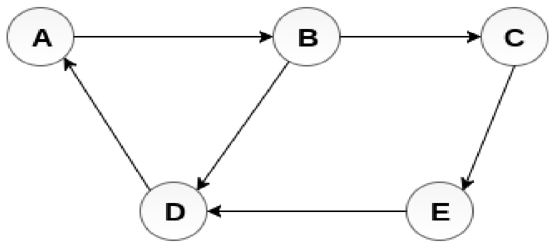
**Undirected Graph**

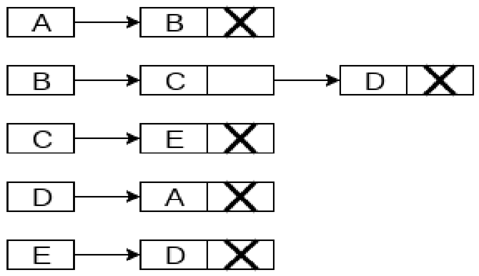


**Adjacency List**

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.

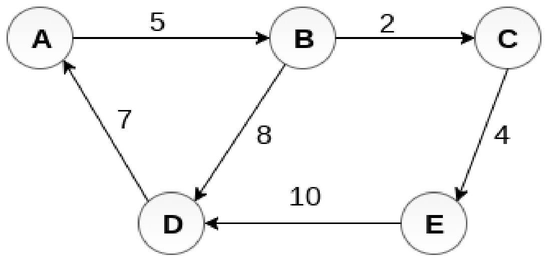


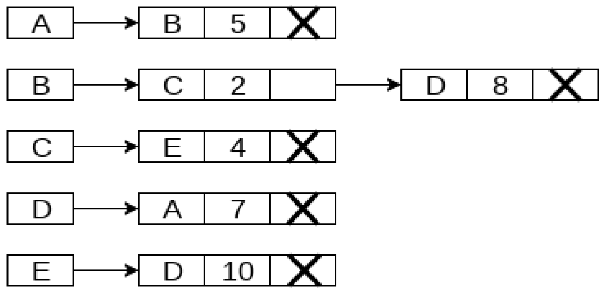


**Directed Graph** **Adjacency List**

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.





**Weighted Directed Graph** **Adjacency List**

Graph Traversal Algorithm

In this part of the tutorial we will discuss the techniques by using which, we can traverse all the vertices of the graph.

Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs. Lets discuss each one of them in detail.

- Breadth First Search
- Depth First Search

**Breadth First Search (BFS) Algorithm**
Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

**Algorithm**

```
Step 1: SET STATUS = 1 (ready state)
for each node in G
Step 2: Enqueue the starting node A
and set its STATUS = 2
(waiting state)
Step 3: Repeat Steps 4 and 5 until
QUEUE is empty
Step 4: Dequeue a node N. Process it
and set its STATUS = 3
(processed state).
Step 5: Enqueue all the neighbours of
N that are in the ready state
(whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]
Step 6: EXIT
```

**Depth First Search (DFS) Algorithm**

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

**Algorithm**

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS
= 1) and set their
STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT
```

**Binary Search Trees:**

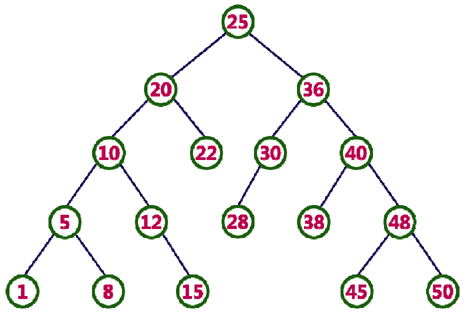
In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In binary tree, the elements are arranged in the order they arrive to the tree from top to bottom and left to right.

A binary tree has the following time complexities...

- **Search Operation - O(n)**
- **Insertion Operation - O(1)**
- **Deletion Operation - O(n)**

To enhance the performance of binary tree, we use special type of binary tree known as **Binary Search Tree**

Suppose T is a binary tree. Then T is called a binary search tree if each Node N of tree T has the following property: The value at N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.



In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.

**Search Operation in BST**

In a binary search tree, the search operation is performed with **O(log n)**time complexity. The search operation is performed as follows...

**Step 1 -** Read the search element from the user.
**Step 2 -** Compare the search element with the value of root node in the tree.
**Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
**Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.
**Step 5 -** If search element is smaller, then continue the search process in left subtree.
**Step 6-** If search element is larger, then continue the search process in right subtree.
**Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node
**Step 8 -** If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

**Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

**Algorithm for Insertion**

a)Compare ITEM with the root node N of the tree.

i)If  ITEM< N , Proceed to the left Child of N.

ii)If ITEM > N, proceed to the right child of N.

b)Repeat Step (a) until one of the following occurs:

i)We meet a node N such that ITEM=N. In this case the search is successful

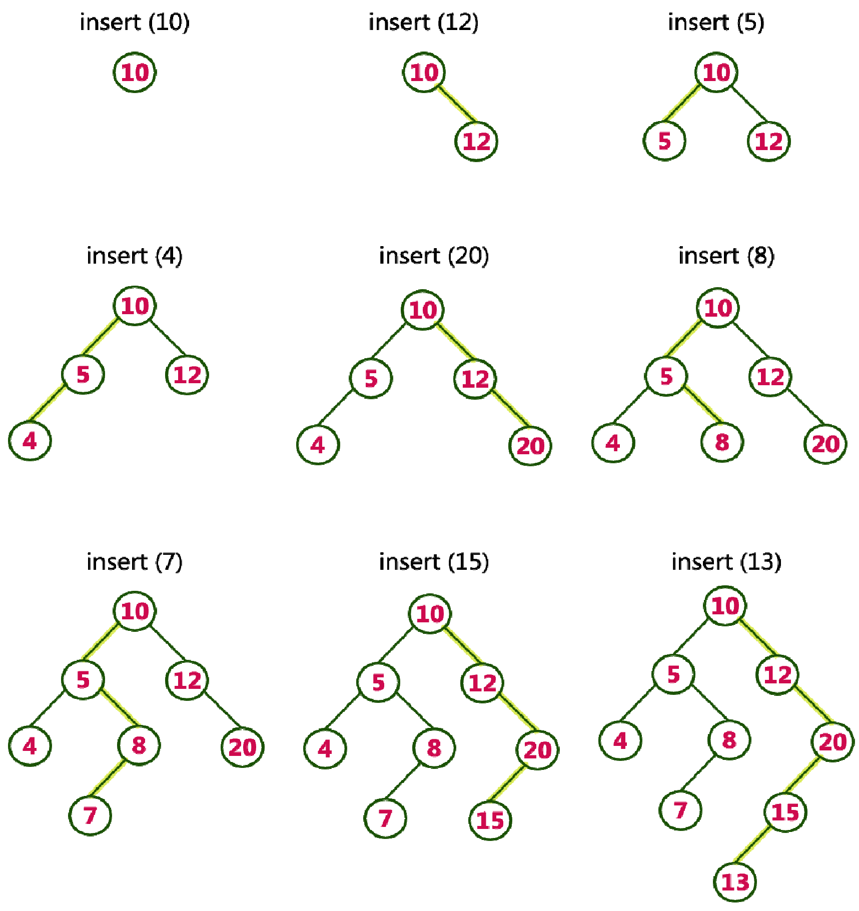
ii)We meet an empty subtree, which indicates that the search is unsuccessful,   and we insert ITEM in place of the empty subtree

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

**10,12,5,4,20,8,7,15 and 13**

Above elements are inserted into a Binary Search Tree as follows...

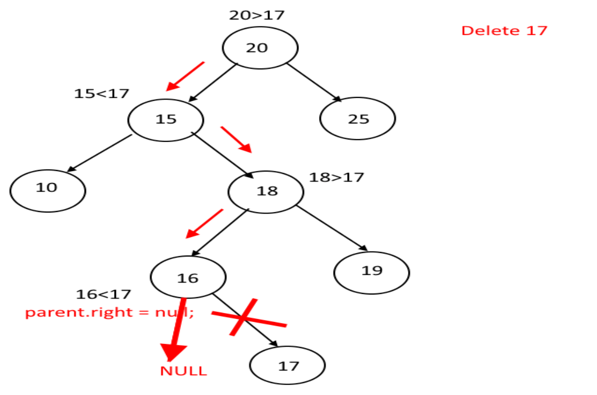


## Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)**time complexity. Deleting a node from Binary search tree includes following three cases...
Case 1: Deleting a Leaf node (A node with no children)
Case 2: Deleting a node with one child
Case 3: Deleting a node with two children

### Case 1: Deleting a leaf node

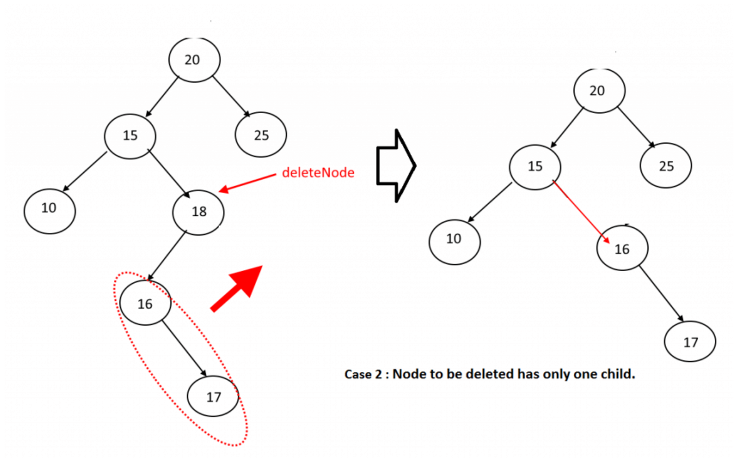
We use the following steps to delete a leaf node from BST...
**Step 1 - Find** the node to be deleted using **search operation**
**Step 2 -** Delete the node using **free** function (If it is a leaf) and terminate the function.



Case 1 : Node to be deleted is a leaf node ( No Children).

### Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...
**Step 1 - Find** the node to be deleted using **search operation**
**Step 2 -** If it has only one child then create a link between its parent node and child node.
**Step 3 -** Delete the node using **free** function and terminate the function.



Case 3: **Deleting a node with two children**

We use the following steps to delete a node with two children from BST...

**Step 1 - Find** the node to be deleted using **search operation**

**Step 2 -** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.

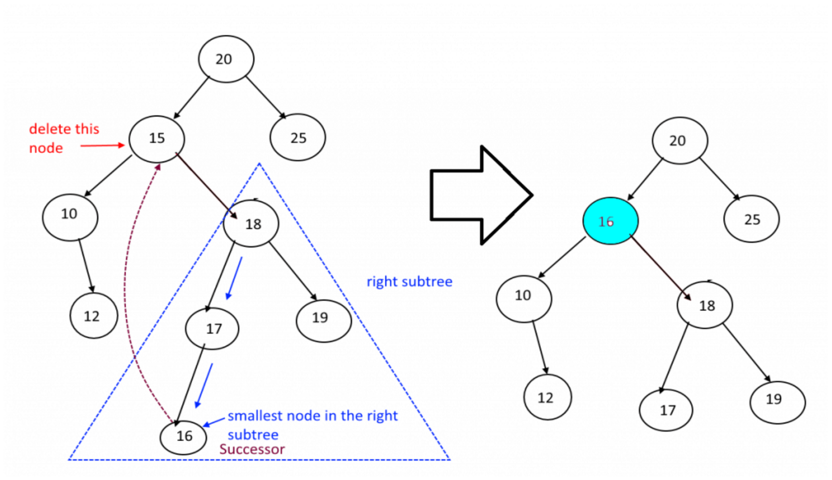
**Step 3 - Swap** both **deleting node** and node which is found in the above step.

**Step 4 -** Then check whether deleting node came to **case 1** or **case 2**or else goto step 2

**Step 5 -** If it comes to **case 1**, then delete using case 1 logic.

**Step 6-** If it comes to **case 2**, then delete using case 2 logic.

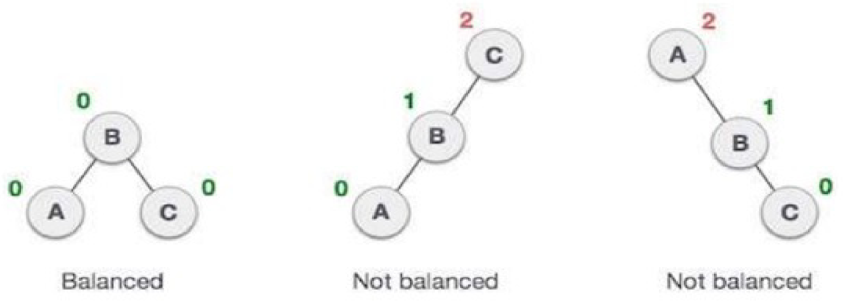
**Step 7 -** Repeat the same process until the node is deleted from the tree.



## AVL tree

In early 60's of 19th century E.M. Landis and G.M. Adelson- Velsky formed a self - balancing BST (binary search tree) data structure. This data structure is known by AVL tree.

An AVL tree is a binary search tree with self – balancing condition. The condition assures that the difference between the height of left and right sub tree cannot be greater than one. This difference between left sub tree and right sub tree is known as **Balance Factor**



Balance Factor= Height of Left Subtree - Height of Right Subtree

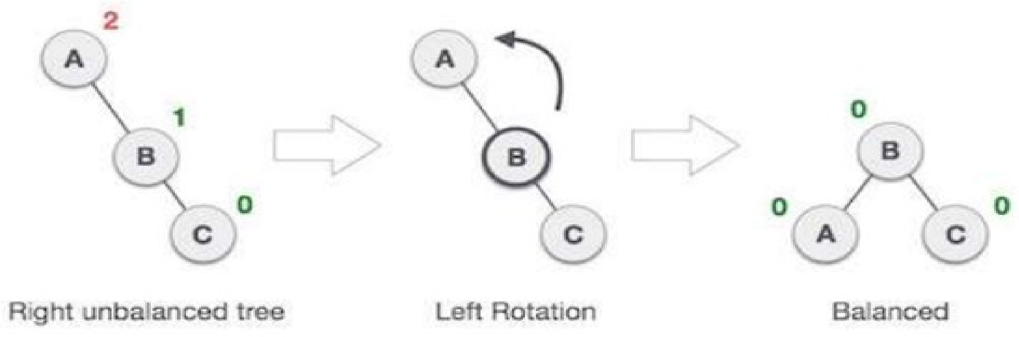
If the value of balance factor is greater than one then the tree is balanced using some rotational techniques and these rotational techniques are known as AVL rotation.

AVL tree have following rotation techniques to balance itself.

- Left rotation
- Right rotation
- Left - Right rotation
- Right - Left rotation

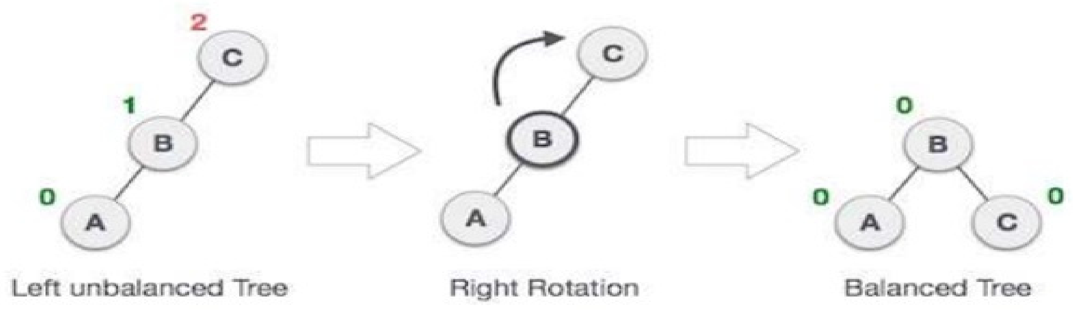
In above techniques the first two are single rotations and next two are double rotations.

**Left Rotation**
If the node is inserted into the right subtree of the right subtree then we can perform the left rotation to balance the unbalanced tree.



In above figure we can see the left rotation of the tree. Using this technique we balance an unbalance tree of height 2.

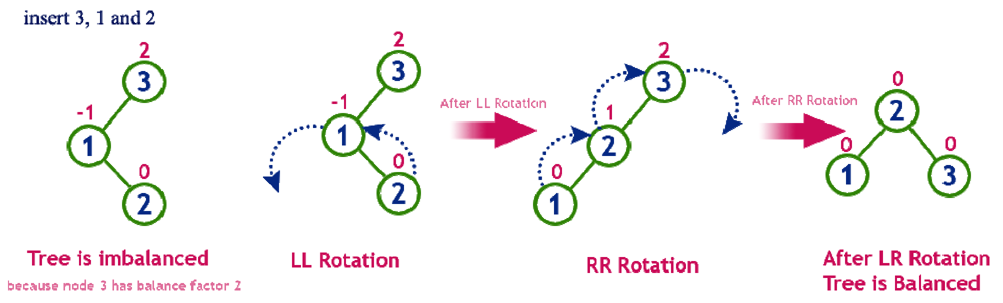
**Right Rotation**
If the node is inserted into the left subtree of the leftsubtree then we can perform the right rotation to balance the unbalanced tree.



In above figure we can see the right rotation of the tree. Using this technique we balance an unbalance tree of height 2.

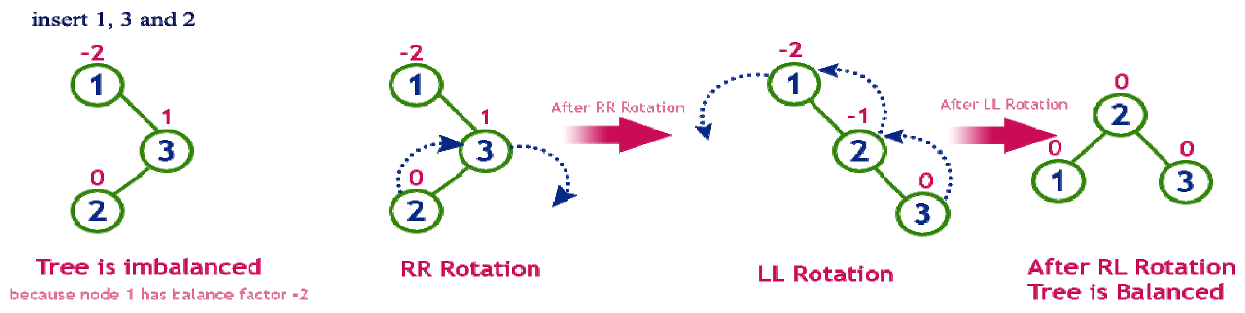
## Left Right Rotation (LR Rotation)

The LR Rotation is sequence of single left rotation followed by single right rotation. In LR Rotation, at first every node moves one position to left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



## Right Left Rotation (RL Rotation)

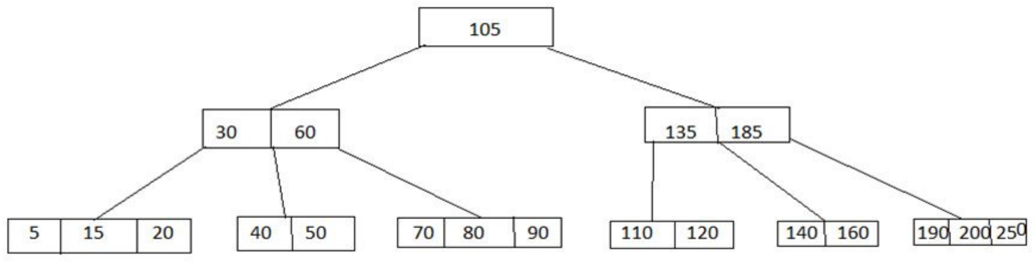
The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



## B TREE

A **B tree** is designed to store sorted data and allows search, insertion and deletion operation to be performed in logarithmic time. As In multiway search tree, there are so many nodes which have left subtree but no right subtree. Similarly, they have right subtree but no left subtree. As is known, access time in the tree is totally dependent on the level of the tree. So our aim is to minimize the access time which can be through balance tree only.
For balancing the tree each node should contain n/2 keys. So the B tree of order n can be defined as:

- All leaf nodes should be at same level.
- All leaf nodes can contain maximum n-1 keys.
- The root has at least two children.
- The maximum number of children should be n and each node can contain k keys. Where, k<=n-1.
- Each node has at least n/2 and maximum n nonempty children.
- Keys in the non-leaf node will divide the left and right sub-tree where the value of left subtree keys will be less and value of right subtree keys will be more than that particular key.



## Red Black Tree

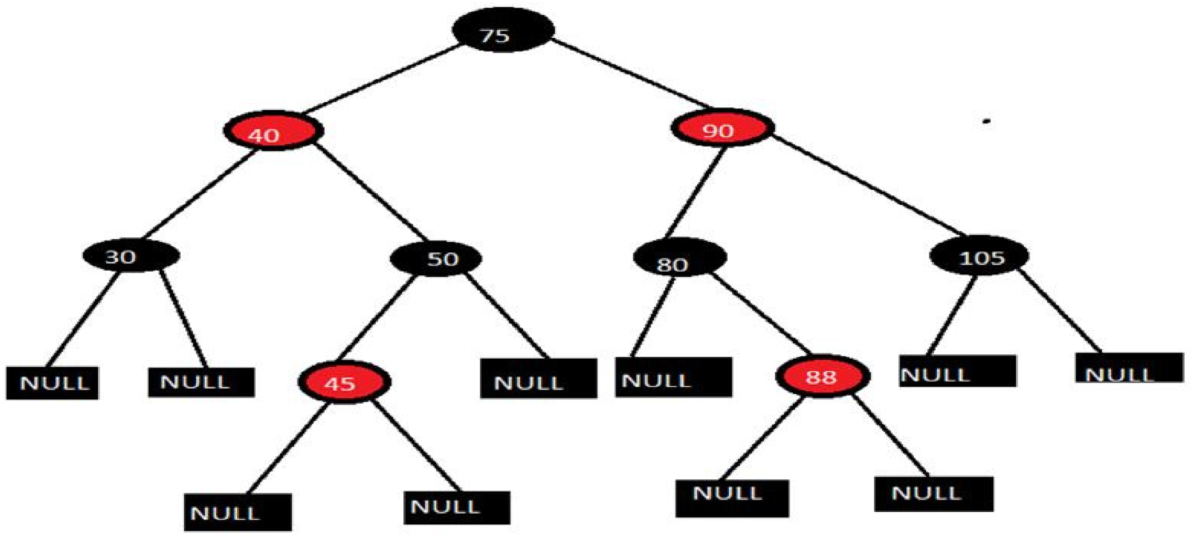
A **Red Black Tree** is a type of self-balancing binary search tree, in which every node is colored with a red or black. The red black tree satisfies all the properties of the binary search tree but there are some additional properties which were added in a Red Black Tree. The height of a Red-Black tree is O(Logn) where (n is the number of nodes in the tree).

**Properties of Red Black Tree**

- The root node should always be black in color.
- Every null child of a node is black in red black tree.

- The children of a red node are black. It can be possible that parent of red node is black node.
- All the leaves have the same black depth.
- Every simple path from the root node to the (downward) leaf node contains the same number of black nodes.

**Representation of Red Black Tree**



While representing the red black tree color of each node should be shown. In this tree leaf nodes are simply termed as null nodes which means they are not physical nodes. It can be checked easily in the above-given tree there are two types of node in which one of them is red and another one is black in color. The above-given tree follows all the properties of a red black tree that are

- It is a binary search tree.
- The root node is black.
- The children's of red node are black.
- All the root to external node paths contain same number of black nodes.

**Example:** Consider path 75-90-80-88-null and 75-40-30-null in both these paths 3 black nodes are there.

**Advantages of Red Black Tree**

- Red black tree are useful when we need insertion and deletion relatively frequent.
- Red-black trees are self-balancing so these operations are guaranteed to be O(logn).
- They have relatively low constants in a wide variety of scenarios.

**Comparison of Search Trees**

The comparison of search trees is performed based on the Time complexity of search, insertion and deletion operations in search trees. The following table provides the Time complexities of search trees. These Time complexities are defined for 'n' number of elements.

| **Search tree** | **Average case** | | | **Worst case** | | |
|---|---|---|---|---|---|---|
| | Insert | Delete | Search | Insert | Delete | Search |
| Binary Search Tree | O(log n) | O(log n) | O(log n) | O(n) | O(n) | O(n) |
| AVL Tree | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ |
| B tree | O(log n) | O(log n) | O(log n) | O(n) | O(n) | O(n) |
| Red Black Tree | O(log n) | O(log n) | O(log n) | O(n) | O(n) | O(n) |