freelance_Project available to buy contact on 8007592194

SR.NO	Project NAME	Technology
1	Online E-Learning Platform Hub	React+Springboot+MySql
2	PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3	Tour and Travel management System	React+Springboot+MySql
4	Election commition of India (online Voting System)	React+Springboot+MySql
5	HomeRental Booking System	React+Springboot+MySql
6	Event Management System	React+Springboot+MySql
7	Hotel Management System	React+Springboot+MySql
8	Agriculture web Project	React+Springboot+MySql
9	AirLine Reservation System / Flight booking System	React+Springboot+MySql
10	E-commerce web Project	React+Springboot+MySql
11	Hospital Management System	React+Springboot+MySql
12	E-RTO Driving licence portal	React+Springboot+MySql
13	Transpotation Services portal	React+Springboot+MySql
14	Courier Services Portal / Courier Management System	React+Springboot+MySql
15	Online Food Delivery Portal	React+Springboot+MySql
16	Muncipal Corporation Management	React+Springboot+MySql
17	Gym Management System	React+Springboot+MySql
18	Bike/Car ental System Portal	React+Springboot+MySql
19	CharityDonation web project	React+Springboot+MySql
20	Movie Booking System	React+Springboot+MySql

freelance_Project available to buy contact on 8007592194

21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql
41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48		React+Springboot+MySql
49		React+Springboot+MySql
50		React+Springboot+MySql

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/cceesept2023

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project

DataStructure By Ashok Pate

Q. Why there is a need of data structure?

- There is a need of data structure to achieve 3 things in programming:
    1. efficiency
    2. abstraction
    3. reusability

Q. What is a Data Structure?

Data Structure is a way to store data elements into the memory (i.e. into the main memory) in an organized manner so that operations like addition, deletion, traversal, se arching, sorting etc... can be performed on it efficiently

Two types of Data Structures are there:

1. Linear / Basic data structures : data elements gets stored / arranged into the memory in a linear manner (e.g. sequentially) and hence can be accessed linearly / sequentially.

- Array - Structure & Union
- Linked List - Stack
- Queue

2. Non-Linear / Advanced data structures : data elements gets stored / arranged into the memory in a non-linear manner (e.g. hierarchical manner) and hence can be accessed non-linearly.

- Tree (Hierarchical manner)
- Binary Heap
- Graph
- Hash Table(Associative manner)

+ Array: It is a basic / linear data structure which is a collection / list of logically related similar type of data elements gets stored/arranged into the memory at contiguos locations

+ Structure: It is a basic / linear data structure which is a collection / list of logically related similar and disimmilar type of data elements gets stored/arranged into the memory collectively i.e. as a single entity/record. sizeof of the structure = sum of size of all its members.

+ Union: Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).

Q. What is a Program?

- A Program is a finite set of instructions written in any programming language (either in a high level programming language like C, C++, Java, Python or in a low level programming language like assembly, machine etc...) given to the machine to do specific task.

Q. What is an Algorithm?

- An algorithm is a finite set of instructions written in any human understandable language (like english), if followed, acomplishesh a given task. - Pseudocode : It is a special form of an algorithm, which is a finite set of instructions written in any human understandable language (like english) with some programming constraints, if followed, acomplishesh a given task. - An algorithm is a template whereas a program is an implementation of an algorithm.

 # Algorithm : to do sum of all array elements

Step-1: initially take value of sum is 0.
Step-2: scan an array seqentially from first element max till last element, and add each array element into the sum.
Step-3: return final sum.

 # Pseudocode : to do sum of all array elements

```
Algorithm ArraySum(A, n){//whereas A is an array of size n
   sum=0;//initially sum is 0
   for( index = 1 ; index <= size ; index++ ) {
   sum += A[ index ];//add each array element into the sum
   }
   return sum;
}
```

- An Algorithm is a solution of a given problem.
- Algorithm = Solution
- One problem may has many solutions. For example

Sorting : to arrange data elements in a collection/list of elements either in an ascending order or in descending order.

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

- When one problem has many solutions/algorithms, in that case we need to select an efficient solution/algorithm, and to decide efficiency of an algorithm we need to do their analysis.
- Analysis of an algorithm is a work of determining / calculating how much time i.e. computer time and space i.e. computer memory it needs to run to completion.

- There are two measures of an analysis of an algorithms:
  1. Time Complexity of an algorithm is the amount of time i.e. computer time it needs to run to completion. 2. Space Complexity of an algorithm is the amount of space i.e. computer memory it needs to run to Completion.

- **# Space Complexity** of an algorithm is the amount of space i.e. computer memory it needs to run to completion.
- Space Complexity = Code Space + Data Space + Stack Space (applicable only for recursive algo)
- Code Space = space required for an instructions
- Data Space = space required for simple variables, constants & instance variables.
- Stack Space = space required for function activation records (local vars, formal parameters, return address, old frame pointer etc...).
- Space Complexity has two components:
  1. Fixed component : code space and data space (space required for simple vars & constants).
  2. Variable component : data space for instance characteristics (i.e. space required for instance vars) and stack space (which is applicable only in recursive algorithms).

 # Asymptotic Analysis : It is a mathematical way to calculate time complexity and space complexity of an algorithm without implementing it in any programming language.

- In this type of analysis, analysis can be done on the basis of basic operation in that algorithm. e.g. in searching & sorting algorithms comparison is the basic operation and hence analysis can be done on the basis of no. of comparisons, in addition of matrices algorithm addition is the basic operation and hence on the basis of addition operation analysis can be done.

"**Best case time complexity" :** if an algo takes minimum amount of time to run to completion then it is referred as best case time complexity.

"Worst case time complexity" : if an algo takes maximum amount of time to run to completion then it is referred as worst case time complexity.

"Average case time complexity" : if an algo takes neither minimum nor maximum amount of time to run to completion then it is referred as an average case time complexity

 # Asympotic Notations:

1. Big Omega (Ω) : this notation is used to denote best case time complexity – also called as asymptotic lower bound, running time of an algorithm cannot be less than its asymptotic lower bound.

2. Big Oh (O) : this notation is used to denote worst case time complexity - also called as asymptotic upper bound, running time of an algorithm cannot be more than its asymptotic upper bound.

3. Big Theta (θ) : this notation is used to denote an average case time complexity - also called as asymptotic tight bound, running time of an algorithm cannot be less than its asymptotic lower bound and cannot be more than its asymptotic upper bound i.e. it is tightly bounded.

---

1. Linear Search / Sequential Search:

Algorithm :

Step-1 : Scan / Accept value of key element from the user which is to be search.

Step-2 : Start traversal of an array and compare value of the key element with each array element sequentially from first element either till match is found or max till last element**, if key is matches with any of array element then return true otherwise return false if key do not matches with any of array element.**

```
# Pseudocode:
Algorithm LinearSearch(A, size, key){
   for( int index = 1 ; index <= size ; index++ ){
   if( arr[ index ] ==  key )
   return true;
   }
   return false;
}
```


Linear Search Actual implementation Program:-

```
import java.util.Scanner;
import java.security.Key;
import java.util.Arrays;

class LinearSearch {

//Accept Arrays method
    public  int[] acceptArrays() {
        Scanner sc=new Scanner(System.in);
        int n;
```

```
        System.out.println("Enter the size of Arrays: ");
        n=sc.nextInt();
        int arr[]=new int[n];
        System.out.println("Enter Element of Arrays");
        int i;
        for (i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        return arr;
    }

//Print arrays method
    public static void printArrays(int arr[]) {
        System.out.println("Arrays Element: "+Arrays.toString(arr));
    }

//Linearsearch recursion method
    public static int LinearSearchRecursive(int[] arr,int left,int
right,int k) {
        if (right<left) {
            return -1;
        }
        else if (arr[left]==k) {
            return left;
        }
        else if (arr[right]==k) {
            return right;
        }
        else
        return LinearSearchRecursive(arr,left+1,right-1,k);
    }

//linearsearch without recursion method
    public static int LinearSearchWithout(int arr[],int key){
        int n;
        for(int i=0;i<arr.length;i++){
            if (arr[i]==key) {
                return i;
            }
        }
        return -1;
    }

//printrecord method
    public static void printRecord(int result,int key) {
```

```
        if(result==-1){
            System.out.println("Element is not Found");
        }
        else{
            System.out.println("Element is found: "+key+ " At index
postion: "+result);
        }
    }


    public static void main1(String[] args) {
        LinearSearch Li=new LinearSearch();
        int arr[]=new int[]{10,25,55,45,76,80,125,35,88,90};
        System.out.println("Arrays Element: "+Arrays.toString(arr));
        int key=125;
        int result=Li.LinearSearchWithout(arr, key);

        if(result==-1){
            System.out.println("Element is not Found");
        }
        else{
            System.out.println("Element is found: "+key+ " At index
postion: "+result);
        }
    }


    public static void main2(String[] args) {
        LinearSearch Li=new LinearSearch();
        Scanner sc=new Scanner(System.in);
        int arr[]=new int[10];
        System.out.println("Enter Element of Arrays");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        System.out.println("Arrays Element: "+Arrays.toString(arr));
        System.out.println("Enter the key you want to search");
        int key=sc.nextInt();
        int result=Li.LinearSearchWithout(arr, key);

        if(result==-1){
            System.out.println("Element is not Found");
        }
        else{
            System.out.println("Element is found: "+key+ " At index
postion: "+result);
```

```
        }
    }


    public static void main3(String[] args) {
        LinearSearch Li=new LinearSearch();
        Scanner sc=new Scanner(System.in);
        int n;
        System.out.println("Enter the size of Arrays: ");
        n=sc.nextInt();
        int arr[]=new int[n];
        System.out.println("Enter Element of Arrays");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        System.out.println("Arrays Element: "+Arrays.toString(arr));
        System.out.println("Enter the key you want to search");
        int key=sc.nextInt();
        int result=Li.LinearSearchWithout(arr, key);

        if(result==-1){
            System.out.println("Element is not Found");
        }
        else{
            System.out.println("Element is found: "+key+ " At index
postion: "+result);
        }
    }


    public static void main4(String[] args) {
        LinearSearch Li=new LinearSearch();
        Scanner sc=new Scanner(System.in);
        int n;
        System.out.println("Enter the size of Arrays: ");
        n=sc.nextInt();
        int arr[]=new int[n];
        System.out.println("Enter Element of Arrays");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        System.out.println("Arrays Element: "+Arrays.toString(arr));
        System.out.println("Enter the key you want to search");
        int key=sc.nextInt();
        System.out.println("Without Recursion");
        int result=Li.LinearSearchWithout(arr, key);
```

```
        Li.printRecord(result, key);
        System.out.println("With Recursion");
        result=Li.LinearSearchRecursive(arr, 0, arr.length-1, key);
        Li.printRecord(result, key);
    }

    public static void main5(String[] args) {
        LinearSearch Li=new LinearSearch();
        Scanner sc=new Scanner(System.in);
        int []arr=Li.acceptArrays();
        Li.printArrays(arr);

        System.out.println("Enter the key you want to search");
        int key=sc.nextInt();
        int choice=0;
        System.out.println("Press 0 : Exit");
        System.out.println("Press 1 : Binary Search Without
Recusion");
        System.out.println("Press 2 : Binary Search With Recursion");
        System.out.println("Enter Your Choice: ");
        do {
            switch (choice) {
                case 1:
                System.out.println("Without Recursion");
                int result=Li.LinearSearchWithout(arr, key);
                Li.printRecord(result, key);
                    break;

                case 2:
                System.out.println("With Recursion");
                result=Li.LinearSearchRecursive(arr, 0, arr.length-1,
key);
                Li.printRecord(result, key);
                break;
                default:
                System.out.println("Invalid Choice");
                    break;
            }
        } while (choice!=0);
    }
}
```

Best Case: If key element is found at very first position in only 1 comparison then it is considered as a best case and running time of an algorithm in this case is O(1) => hence time complexity of linear search algorithm in base case = Ω(1).

Worst Case: If either key element is found at last position or key element does not exists, in this case maximum n no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is O(n) => hence time complexity of linear search algorithm in worst case = O(n).
Average Case: If key element is found at any in between position it is considered as an average case and running time of an algorithm in this case is O(n/2) => O(n) => hence time complexity = θ(n)

---

2. Binary Search / Logarithmic Search / Half Interval Search : -

This algorithm follows divide-and-conquer approach.

To apply binary search on an array prerequisite is that array elements must be in a sorted manner.

Step-1: Accept value of key element from the user which is to be search.

Step-2: In first iteration, find/calculate mid position by the formula mid=(left+right)/2, (by means of finding mid position big size array gets divided logically into 2 subarrays, left subarray and right subarray, left subarray => [left to mid-1] & right subarray => [mid+1 to right].

Step-3 : Compare value of key element with an element which is at mid position, if key matches in very first iteration in only one comparison then it is considered as a best case, if key matches with mid pos element then return true otherwise if key do not matches then we have to go to next iteration, and in next iteration we go to search key either into the left subarray or into the right subarray.

Step-4 : Repeat Step-2 & Step-3 till either key is found or max till subarray is valid, if subarray is not valid then key is not found in this case return false.

Actual implementation of Binary Search :-

```
import java.util.Scanner;
import java.util.Arrays;

 class BinarySearch {
//Accept Arrays Method
     public int[] acceptArrays(){
        Scanner sc=new Scanner(System.in);
        int n;
        System.out.println("Enter the size of Arryas");
        n=sc.nextInt();
        int arr[]=new int[n];
        System.out.println("Enter Element of Arrays ");

        for(int i=0; i<arr.length;i++){
```

```
            arr[i]=sc.nextInt();
        }
        return arr;
    }
//Print Arrays Method
     public void printArrays(int arr[]){
        System.out.println("You Entered Element Before sort:
"+Arrays.toString(arr));
        Arrays.sort(arr);
        System.out.println("You Entered Element After sort:
"+Arrays.toString(arr));
     }
//Method for Binary search
    public int BinarySeaarchImplementation(int arr[],int key){
        int left=0;
        int right= arr.length-1;

        while (left<=right) {
            int mid=(left + right)/2;

            if(arr[mid]==key)
            return mid;

            if(arr[mid] < key){

                left=mid+1;
            }
            else{
                right=mid-1;
            }

        }
        return -1;
    }
//Method with recursion Binary search
    public int BinarySeaarchRecursion(int arr[],int key,int left,int
right){
            // int left;
            // int right=arr.length-1;
            while(right>=left && left<=arr.length-1){
                int mid=(left+right)/2;
                if(arr[mid]==key)
                return mid;
                if(arr[mid]<key){
                    return BinarySeaarchRecursion(arr,
key,mid+1,right);
```

```
                }
                else{
                    return BinarySeaarchRecursion(arr, key,left,mid-
1);
                }
            }
            return -1;
    }
//Method for print result
    public void printRecord(int result,int key){
        if(result==-1){
            System.out.println("Key is not found");
        }
        else{
            System.out.println("Key is found : "+key);
        }
    }


    public static void main1(String[] args) {
        Scanner sc=new Scanner(System.in);

        int []arr=new int[]{10,20,30,40,50,60,70,80,90,100};
        int key=75;
        // key=50;
        BinarySearch bi=new BinarySearch();
        int result=bi.BinarySeaarchImplementation(arr,key);
        if(result==-1){
            System.out.println("Key is not found");
        }
        else{
            System.out.println("Key is found : "+key);
        }
    }

    public static void main2(String[] args) {
        BinarySearch bi=new BinarySearch();
        Scanner sc=new Scanner(System.in);
        int []arr=new int[10];
        System.out.println("Enter Element of arrays in sorted
mannar");
        int i;
        for(i=0; i<=arr.length-1;i++){
            arr[i]=sc.nextInt();
        }
        /*For loops for print the arrays.
```

```
            for(i=0;i<=arr.length;i++){
             System.out.println(arr[i]);
         }
         */
        //Insted of this directly print arrays by toString.
        System.out.println("You Entered Element is:
"+Arrays.toString(arr));
        System.out.println("Enter the key you want to find in
arrays:");
        int key=sc.nextInt();
        int result=bi.BinarySeaarchImplementation(arr,key);
        if(result==-1){
            System.out.println("Key is not found");
        }
        else{
            System.out.println("Key is found : "+key);
        }
    }

    public static void main3(String[] args) {
        BinarySearch bi=new BinarySearch();
        Scanner sc=new Scanner(System.in);
        int n;
        n=sc.nextInt();
        int []arr=new int[n];
        System.out.println("Enter Element of arrays in sorted
mannar");
        int i;
        for(i=0; i<=arr.length-1;i++){
            arr[i]=sc.nextInt();
        }
        System.out.println("You Entered Element is:
"+Arrays.toString(arr));

        System.out.println("Enter the key you want to find in
arrays:");
        int key=sc.nextInt();
        int result=bi.BinarySeaarchImplementation(arr,key);
        if(result==-1){
            System.out.println("Key is not found");
        }
        else{
            System.out.println("Key is found : "+key);
        }
    }
```

```
    public static void main4(String[] args) {
        BinarySearch bi=new BinarySearch();
        Scanner sc=new Scanner(System.in);
        int n;
        System.out.println("Enter the size of Arryas");
        n=sc.nextInt();
        int[]arr=new int[n];
        System.out.println("Enter Element of Arrays ");
        int i;
        for(i=0; i<=arr.length-1;i++){
            arr[i]=sc.nextInt();
        }
        System.out.println("You Entered Element Before sort:
"+Arrays.toString(arr));

        Arrays.sort(arr);
        System.out.println("You Entered Element After sort:
"+Arrays.toString(arr));

        System.out.println("Enter the key you want to find in
arrays:");
        int key=sc.nextInt();
        int result=bi.BinarySeaarchImplementation(arr,key);
        System.out.println("With out recursion");
        if(result==-1){
            System.out.println("Key is not found");
        }
        else{
            System.out.println("Key is found : "+key);
        }
    }

    public static void main5(String[] args) {
        BinarySearch bi=new BinarySearch();
        Scanner sc=new Scanner(System.in);
        int n;
        n=sc.nextInt();
        int[]arr=new int[n];
        System.out.println("Enter Element of Arrays ");
        int i;
        for(i=0; i<=arr.length-1;i++){
            arr[i]=sc.nextInt();
        }
        System.out.println("You Entered Element Before sort:
"+Arrays.toString(arr));
```

```
        Arrays.sort(arr);
        System.out.println("You Entered Element After sort:
"+Arrays.toString(arr));

        System.out.println("Enter the key you want to find in
arrays:");
        int key=sc.nextInt();
        System.out.println("With out recursion");
        int result=bi.BinarySeaarchImplementation(arr,key);
        bi.printRecord(result, key);

        System.out.println("With recursion");
        result=bi.BinarySeaarchRecursion(arr, key, 0, arr.length-1);
        bi.printRecord(result, key);
    }

    public static void main(String[] args) {
        Test bi=new Test();
        Scanner sc=new Scanner(System.in);
        int[] arr = bi.acceptArrays();
        bi.printArrays(arr);
        int option=0;
        do {
            System.out.println("Enter the key you want to find in
arrays:");
            int key=sc.nextInt();
            System.out.println("Press 0 : Exit");
            System.out.println("Press 1 : Binary Search Without
Recusion");
            System.out.println("Press 2 : Binary Search With
Recursion");
            System.out.println("Enter Your Choice: ");
            option=sc.nextInt();
            switch (option) {
                case 1:
                System.out.println("With out recursion");
                int result=bi.BinarySeaarchImplementation(arr,key);
                bi.printRecord(result, key);
                    break;
                case 2:
                System.out.println("With recursion");
                result=bi.BinarySeaarchRecursion(arr, key, 0,
arr.length-1);
                bi.printRecord(result, key);
```

```
                    break;

                default:
                    break;
            }
        } while (option!=0);
    }
}
```

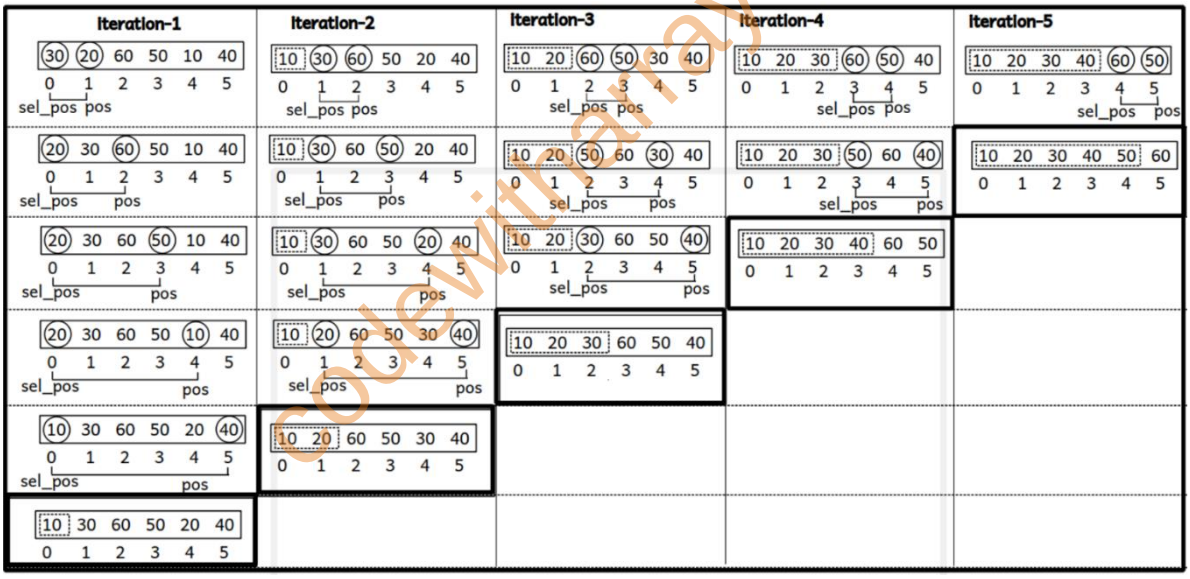
Best Case: if the key is found in very first iteration at mid position in only 1 comparison OR if key is found at root position it is considered as a best case and running time of an algorithm in this case is O(1) = Ω(1).

Worst Case: if either key is not found or key is found at leaf position it is considered as a worst case and running time of an algorithm in this case is O(log n) = O(log n).

Average Case: if key is found at non-leaf position it is considered as an average case and running time of an algorithm in this case is O(log n) = θ(log n).

---

1. **Selection Sort: -**



- In this algorithm, in first iteration, first position gets selected and element which is at selected position gets compared with all its next position elements sequentially, if an element at selected position found greater than any other position element then swapping takes place and in first iteration smallest element gets setteled at first position.

- In the second iteration, second position gets selected and element which is at selected position gets compared with all its next position elements, if an element selected position found greater than any other position element then swapping takes place and in second iteration second smallest element gets setteled at second position, and so on in maximum (n-1) no. of iterations all array elements gets arranged in a sorted manner.

Actual Implementation Of Selection Sorting :-

```
import java.util.Arrays;
import java.util.Scanner;
class SelectionSortMain {
    public static void AcendingSelectionSort(int arr[]) {
        for (int i = 0 ,k=1; i < arr.length-1; i++) {
            for (int j = i+1; j < arr.length; j++) {
                if (arr[i]>arr[j]) {
                    int temp=arr[i];
                    arr[i]=arr[j];
                    arr[j]=temp;
                }
            }
            System.out.println("Pass"+ k++ +":"+Arrays.toString(arr));
        }
    }
    public static void DecendingSelectionSort(int arr[]) {
        for (int i = 0, k=1; i < arr.length-1; i++) {
            for (int j = i+1; j < arr.length; j++) {
                if (arr[i]<arr[j]) {
                    int temp=arr[i];
                    arr[i]=arr[j];
                    arr[j]=temp;
                }
            }
            System.out.println("Pass"+ k++ +":"+Arrays.toString(arr));
        }
    }
    public static void main1(String[] args) {

        int []arr={6,4,2,8,3,1};
        System.out.println("Acending Selection Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        AcendingSelectionSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));
```

```
    }
    public static void main2(String[] args) {
       Scanner sc=new Scanner(System.in);
        System.out.println("Enter array Size: ");
        int size=sc.nextInt();
        int arr[]=new int[size];
        System.out.println("Enter array Elements: ");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        System.out.println("Acending Selection Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        AcendingSelectionSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));
        System.out.println("Decending Selection Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        DecendingSelectionSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));
    }
    public static void main(String[] args) {
       Scanner sc=new Scanner(System.in);
        System.out.println("Enter array Size: ");
        int size=sc.nextInt();
        int arr[]=new int[size];
        System.out.println("Enter array Elements: ");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        int choice=0;
        do {
            System.out.println("Press 0: Exit");
            System.out.println("Press 1: Acending Selection Sort");
            System.out.println("Press 2: Decending Selection Sort");
            System.out.println("Enter Your Choice: ");
            choice=sc.nextInt();
            switch (choice) {
                case 1:
                System.out.println("Acending Selection Sort:");
                System.out.println("Before sorting:
"+Arrays.toString(arr));
                AcendingSelectionSort(arr);
                System.out.println("After
Sorting:"+Arrays.toString(arr));
                    break;
                case 2:
                System.out.println("Decending Selection Sort:");
```

```
                System.out.println("Before sorting:
"+Arrays.toString(arr));
                DecendingSelectionSort(arr);
                System.out.println("After
Sorting:"+Arrays.toString(arr));

                default:
                System.out.println("Invalid Choice");
                break;
            }
        } while (choice!=0);
    }
}
/* Output:
Acending Selection Sort:
Before sorting: [6, 4, 2, 8, 3, 1]
After Sorting:[1, 2, 3, 4, 6, 8]
*/
/*
Acending Selection Sort:
Before sorting: [6, 4, 2, 8, 3, 1]
Pass1:[1, 6, 4, 8, 3, 2]
Pass2:[1, 2, 6, 8, 4, 3]
Pass3:[1, 2, 3, 8, 6, 4]
Pass4:[1, 2, 3, 4, 8, 6]
Pass6:[1, 2, 3, 4, 6, 8]
After Sorting:[1, 2, 3, 4, 6, 8]
Decending Selection Sort:
Before sorting: [8, 6, 4, 3, 2, 1]
Pass8:[8, 6, 4, 3, 2, 1]
Pass6:[8, 6, 4, 3, 2, 1]
Pass4:[8, 6, 4, 3, 2, 1]
Pass3:[8, 6, 4, 3, 2, 1]
Pass2:[8, 6, 4, 3, 2, 1]
After Sorting:[8, 6, 4, 3, 2, 1]
*/
```


- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **O(n^2)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n^2)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order,

but its elements are in descending order. The worst-case time complexity of selection sort is **O(n^2)**.

- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

---

2. Bubble Sort : -

- In this algorithm, in every iteration elements which are at two consecutive positions gets compared, if they are already in order then no need of swapping between them, but if they are not in order i.e. if prev position element is greater than its next position element then swapping takes place, and by this logic in first iteration largest element gets setteled at last position, in second iteration second largest element gets setteled at second last position and so on, in max (n-1) no. of iterations all elements gets arranged in a sorted manner.

Actual Implementation Of Bubble Sorting:-

```
import java.util.Arrays;
import java.util.Scanner;
class BubbleSortMain {
public static void BubbleSort(int arr[]) {
    for (int i = 1; i < arr.length; i++) {
        for (int j = 0; j < arr.length-1; j++) {
        //compare consective elements
        //if left elements is greater than right element ,than swap
them
            if (arr[j]>arr[j+1]) {
            int temp=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
            }
        }
        System.out.println("Pass"+i+":"+Arrays.toString(arr));
    }
}
public static void EfficientBubbleSort(int arr[]) {
    for (int i = 1; i < arr.length; i++) {
        for (int j = 0; j < arr.length-1-i; j++) {
        //compare consective elements
        //if left elements is greater than right element ,than swap
them
            if (arr[j]>arr[j+1]) {
```

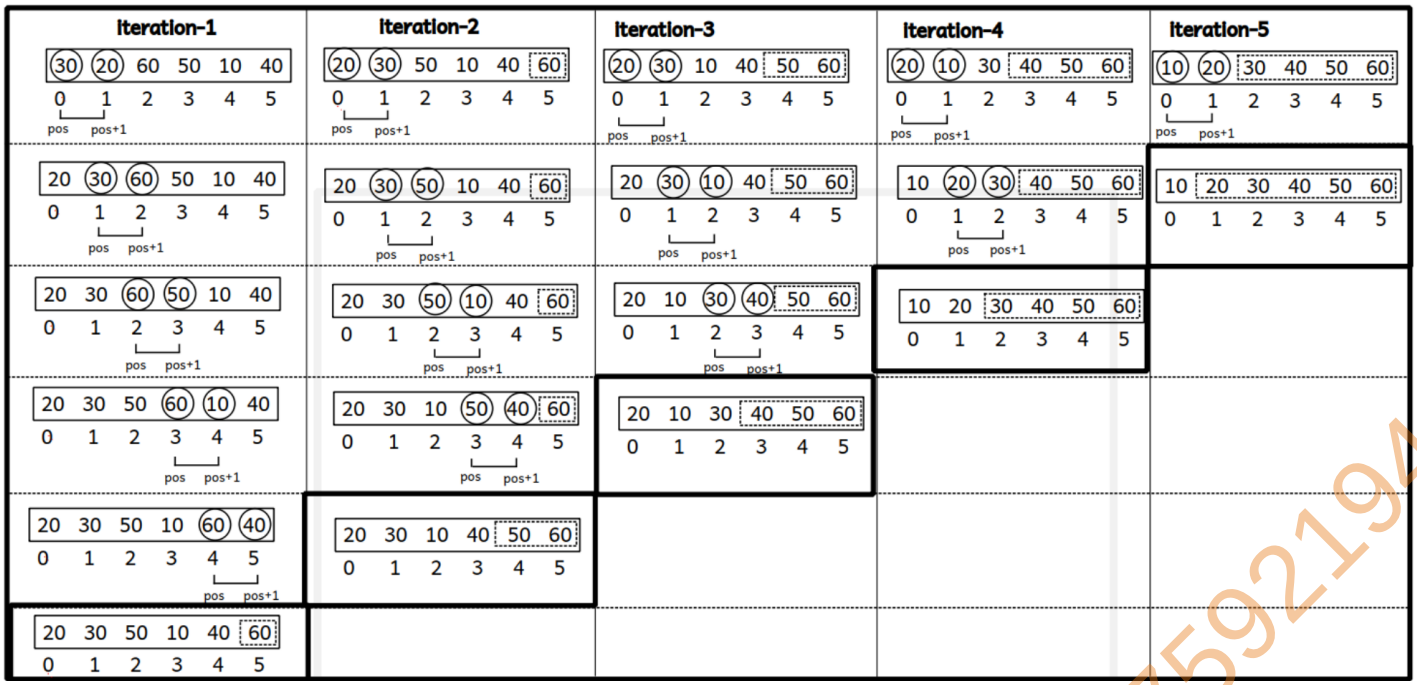
```
            int temp=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
            }
        }
        System.out.println("Pass"+i+":"+Arrays.toString(arr));
    }
}
public static void MoreEfficientBubbleSort(int arr[]) {
    for (int i = 1; i < arr.length; i++) {
        boolean swapFalg=false;
        for (int j = 0; j < arr.length-1-i; j++) {
        //compare consective elements
        //if left elements is greater than right element ,than swap
them
            if (arr[j]>arr[j+1]) {
            int temp=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
            swapFalg=true;
            }
        }
        if (swapFalg==false) {
            System.out.println("Pass"+i+":"+Arrays.toString(arr));
            break;
        }
    }
}

public static void main1(String[] args) {
    int []arr={6,4,2,8,3,1};
    System.out.println("Bubble Sort:");
    System.out.println("Before sorting: "+Arrays.toString(arr));
    BubbleSort(arr);
    System.out.println("After Sorting:"+Arrays.toString(arr));

    }
    public static void main2(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter array Size: ");
        int size=sc.nextInt();
        int arr[]=new int[size];
        System.out.println("Enter array Elements: ");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
```

```
        System.out.println("Bubble Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        BubbleSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));
    }
    public static void main3(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter array Size: ");
        int size=sc.nextInt();
        int arr[]=new int[size];
        System.out.println("Enter array Elements: ");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        System.out.println("Normal Bubble Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        BubbleSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));
        System.out.println("\n");
        System.out.println("Efficent Bubble Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        EfficientBubbleSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));
    }
    public static void main4(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter array Size: ");
        int size=sc.nextInt();
        int arr[]=new int[size];
        System.out.println("Enter array Elements: ");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        // int[]arr={1,2,3,4,5,6};
        System.out.println("Normal Bubble Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        BubbleSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));

        System.out.println("\n");

        System.out.println("Efficent Bubble Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        EfficientBubbleSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));
        System.out.println("\n");
```

```
        System.out.println("More Efficent Bubble Sort:");
        System.out.println("Before sorting: "+Arrays.toString(arr));
        MoreEfficientBubbleSort(arr);
        System.out.println("After Sorting:"+Arrays.toString(arr));
    }
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter array Size: ");
        int size=sc.nextInt();
        int arr[]=new int[size];
        System.out.println("Enter array Elements: ");
        for (int i = 0; i < arr.length; i++) {
            arr[i]=sc.nextInt();
        }
        int choice=0;
       do {
        System.out.println("Press 0: Exit");
        System.out.println("Press 1: Normal Bubble Sort");
        System.out.println("Press 2: Efficent Bubble Sort");
        System.out.println("Press 3: More Efficent Bubble Sort");
        System.out.println("Enter Your Choice: ");
        choice=sc.nextInt();
        switch (choice) {
            case 1:
            System.out.println("Normal Bubble Sort:");
            System.out.println("Before sorting:
"+Arrays.toString(arr));
            BubbleSort(arr);
            System.out.println("After Sorting:"+Arrays.toString(arr));
                break;
            case 2:
            System.out.println("Efficent Bubble Sort:");
            System.out.println("Before sorting:
"+Arrays.toString(arr));
            EfficientBubbleSort(arr);
            System.out.println("After
Sorting:"+Arrays.toString(arr));
                break;
            case 3:
            System.out.println("More Efficent Bubble Sort:");
            System.out.println("Before sorting:
"+Arrays.toString(arr));
            MoreEfficientBubbleSort(arr);
            System.out.println("After
Sorting:"+Arrays.toString(arr));
```

```
                break;

            default:
            System.out.println("Invalid Choice");
                break;
            }
        } while (choice!=0);
    }
}
/*
Normal Bubble Sort:
Before sorting: [1, 2, 3, 4, 5, 6]
Pass1:[1, 2, 3, 4, 5, 6]
Pass2:[1, 2, 3, 4, 5, 6]
Pass3:[1, 2, 3, 4, 5, 6]
Pass4:[1, 2, 3, 4, 5, 6]
Pass5:[1, 2, 3, 4, 5, 6]
After Sorting:[1, 2, 3, 4, 5, 6]

Efficent Bubble Sort:
Before sorting: [1, 2, 3, 4, 5, 6]
Pass1:[1, 2, 3, 4, 5, 6]
Pass2:[1, 2, 3, 4, 5, 6]
Pass3:[1, 2, 3, 4, 5, 6]
Pass4:[1, 2, 3, 4, 5, 6]
Pass5:[1, 2, 3, 4, 5, 6]
After Sorting:[1, 2, 3, 4, 5, 6]

More Efficent Bubble Sort:
Before sorting: [1, 2, 3, 4, 5, 6]
Pass1:[1, 2, 3, 4, 5, 6]
After Sorting:[1, 2, 3, 4, 5, 6]
*/
```


- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**



- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.
- The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.

---

3. Insertion Sort:

- In this algorithm, in every iteration one element gets selected as a key element and key element gets inserted into an array at its appropriate position towards its left hand side elements in a such a way that elements which are at left side are arranged in a sorted manner, and so on, in max (n-1) no. of iterations all array elements gets arranged in a sorted manner.
- This algorithm works efficiently for already sorted input sequence by design and hence running time of an algorithm is O(n) and it is considered as a best case.

Actual implementation Of Insertion Sorting :-

```
import java.util.Arrays;
import java.util.Scanner;
class InsertionSortMain {
    public static void insertion(int arr[]) {
       //This is only insertion technique not sorting.
       //Only if last element is unsorted this code use not for other
       //thats why in next code imporve when whole array is unsorted.
```

```
        int temp=arr[arr.length-1];
        int j;
        for ( j =arr.length-2; j >=0 && arr[j]>temp; j--) {
            arr[j+1]=arr[j];
        }
        arr[j+1]=temp;
    }
    public static void insertionSort(int arr[]) {
        //n-1 passes: in each pass consider ith element as last element
of arrays
         for (int i = 1; i <arr.length; i++) {
         //copy last element into temp variaable
         int temp=arr[i];
         int j;
         // comapre temp with all element before that,find approriate
postion for the element
         for ( j =i-1; j >=0 && arr[j]>temp; j--) {
             arr[j+1]=arr[j];
         }
         arr[j+1]=temp;
         System.out.println("Pass"+i+":"+Arrays.toString(arr));
    }
}
    public static void main1(String[] args) {
         int arr[]={1,3,4,6,8,2};
         System.out.println("Insertion Technique:");
         System.out.println("Before insertion:"+Arrays.toString(arr));
         insertion(arr);
         System.out.println("After insertion:"+Arrays.toString(arr));
    }
    public static void main2(String[] args) {
         int arr[]={10,23,65,34,56,75,25};
         System.out.println("Insertion Sort:");
         System.out.println("Before insertion:"+Arrays.toString(arr));
         insertionSort(arr);
         System.out.println("After insertion:"+Arrays.toString(arr));
    }
    public static void main(String[] args) {
         Scanner sc=new Scanner(System.in);
         System.out.println("Enter array Size: ");
         int size=sc.nextInt();
         int arr[]=new int[size];
         System.out.println("Enter array Elements: ");
         for (int i = 0; i < arr.length; i++) {
             arr[i]=sc.nextInt();
         }
```

```
        System.out.println("Insertion Sort:");
        System.out.println("Before insertion:"+Arrays.toString(arr));
        insertionSort(arr);
        System.out.println("After insertion:"+Arrays.toString(arr));
    }
}
/*
Insertion Technique:
Before insertion:[1, 3, 4, 6, 8, 2]
After insertion:[1, 2, 3, 4, 6, 8]
Insertion Sort:
Before insertion:[10, 23, 25, 65, 34, 56, 75]
Pass2:[10, 23, 25, 65, 34, 56, 75]
Pass3:[10, 23, 25, 65, 34, 56, 75]
Pass4:[10, 23, 25, 34, 65, 56, 75]
Pass5:[10, 23, 25, 34, 56, 65, 75]
Pass6:[10, 23, 25, 34, 56, 65, 75]
After insertion:[10, 23, 25, 34, 56, 65, 75]
*/
```

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.
- **The space complexity** of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

Limitations of an array data structure:

1. Array is static, i.e. size of an array is fixed, its size cannot be either grow or shrink during runtime.

2. Addition and deletion operations on an array are not efficient as it takes O(n) time, and hence to overcome these two limitations of an Array data structure Linked List data structure has been designed.

Linked List: It is a basic/linear data structure, which is a collection/list of logically related similar type of elements in which, an address of first element in a collection/list is stored into a pointer variable referred as a head pointer and each element contains actual data and link to its next element i.e. an address of its next element (as well as an addr of its previous element).

An element in a Linked List is also called as a Node.

Four types of linked lists are there:

1.Singly Linear Linked List 2.Singly Circular Linked List

3.Doubly Linear Linked List 4.Doubly Circular Linked List

- Basically we can perform addition, deletion, traversal etc... operations onto the linked list data structure.
- We can add and delete node into and from linked list by three ways: add node into the linked list at last position, at first position and at any specific position, simillarly we can delete node from linked list which is at first position, at last position and at any specific position.

1. Singly Linear Linked List: It is a type of linked list in which

- head always contains an address of first element, if list is not empty.
- each node has two parts:

i. data part :

- it contains actual data of any primitive/non-primitive type.

ii. pointer part (next) :

- it contains an address of its next element/node.
- last node points to NULL, i.e. next part of last node contains NULL.
  - Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
  - A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
  - The last node of the list contains pointer to the null

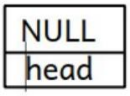
Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.

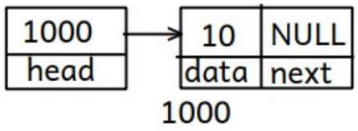
- We can store values of primitive types or objects in the singly linked list.

SINGLY LINEAR LINKED LIST

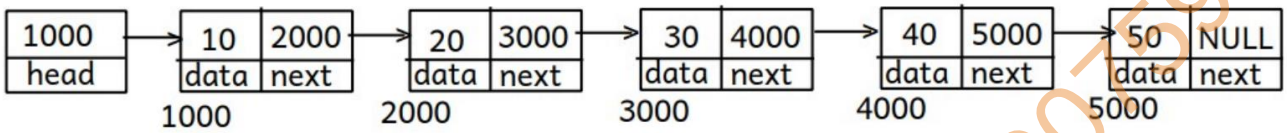
1) singly linear linked list --> list is empty



2) singly linear linked list --> list contains only one node



3) singly linear linked list --> list contains more than one nodes



Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Insertion

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .


```
/*Singly Linked List complete code*/

import java.util.Iterator;
import java.util.Scanner;
//Create node and traversal of that node.
class Node {
    int data;
```

```
    Node next;
    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
class LinkedList2 implements Iterable<Integer>,
Iterator<Integer> {
    Node head;
    public boolean empty() {
        return this.head == null;
    }
    // Add node at first position logic 1
    /*
     * public void addFirst(int data) {
     * Node newNode=new Node(data);
     * if(this.empty()){
     * this.head=newNode;
     * }
     * else{
     * newNode.next=this.head;
     * this.head=newNode;
     * }
     * }
     */
    // Add node at first position logic 2
    public void addFirst(int data) {
        Node newNode = new Node(data);
        if (!this.empty()) {
            newNode.next = this.head;
        }
        this.head = newNode;
    }
    // Add node at last Position
    public void addLast(int data) {
        Node newNode = new Node(data);
        if (this.empty())
            this.head = newNode;
        else {
            Node trav = this.head;
            while (trav.next != null) {
```

```
                trav = trav.next;
            }
            trav.next = newNode;
        }
    }
    // find out node
    public Node find(int data) {
        Node trav = this.head;
        while (trav != null) {
            if (trav.data == data)
                return trav;
            trav = trav.next;
        }
        return null;
    }
    // getcount of LinkedList
    public int getcount() {
        int count = 0;
        Node trav = this.head;
        while (trav != null) {
            ++count;
            trav = trav.next;
        }
        return count;
    }
    // add node at perticular position
    public void addAtPosition(int data, int position) {
        if (position <= 0 || position > (this.getcount() + 1))
{
            System.out.println("Invalid Position");
        }
        if (position == 1) {
            this.addFirst(data);
        } else if (position == this.getcount() + 1) {
            this.addLast(data);
        } else {
            Node trav = this.head;
            for (int count = 1; count < position - 1; count++)
{
                trav = trav.next;
            }
```

```
            Node newNode = new Node(data);
            newNode.next = trav.next;
            trav.next = newNode;
        }
    }
    // remove at first position
    public void removeFirst() {
        if (this.empty()) {
            System.out.println("LinkedList is Empty");
        }
        this.head = this.head.next;
    }
    // remove at last Postion
    public void removeLast() {
        if (this.empty()) {
            System.out.println("Linked list empty");
        }
        if (this.head.next == null) {
            this.head = null;
        } else {
            Node trav = this.head;
            while (trav.next.next != null) {
                trav = trav.next;
            }
            trav.next = null;
        }
    }
    // remove at perticular Position
    public void removeAtPosition(int position) {
        if (position <= 0 || position > this.getcount()) {
            System.out.println("Invalid Position");
        }
        if (position == 1) {
            this.removeFirst();
        } else if (position == this.getcount()) {
            this.removeLast();
        } else {
            Node trav = this.head;
            for (int count = 1; count < position - 1; count++)
{
                trav = trav.next;
```

```
            }
            Node temp = trav.next;
            trav.next = temp.next;
            temp = null;
        }
    }
    // delete whole linked list
    public void clear() {
        while (!this.empty()) {
            this.removeFirst();
        }
    }
    // Display list
    public void display() {
        Node temp = this.head;
        if (this.empty()) {
            System.out.println("Linked list does not exist");
        } else {
            System.out.print("Head--> ");
            while (temp != null) {
                System.out.print(temp.data + "--> ");
                temp = temp.next;
            }
        }
        System.out.println("Null");
    }
    // reverse printing list
    public void reverse(Node head) {
        if (head == null)
            return;
        reverse(head.next);
        System.out.print(head.data + "<--");
    }
    Node trav;
    @Override
    public Iterator<Integer> iterator() {
        this.trav = this.head;
        return this;
    }
    @Override
    public boolean hasNext() {
```

```
            return this.trav != null;
        }
        @Override
        public Integer next() {
            int data = this.trav.data;
            this.trav = this.trav.next;
            return null;
        }
}
public class LinkedList {
        private static Scanner sc = new Scanner(System.in);
        public static void acceptRecord(int[] data) {
            System.out.print("Enter data    :   ");
            data[0] = sc.nextInt();
        }
        public static void acceptPosition(int[] position) {
            System.out.print("Enter position    :   ");
            position[0] = sc.nextInt();
        }
        public static int menuList() {
            System.out.println("0.Exit");
            System.out.println("1.Add first.");
            System.out.println("2.Add last.");
            System.out.println("3.Add at position.");
            System.out.println("4.Remove first.");
            System.out.println("5.Remove last.");
            System.out.println("6.Remove from position.");
            System.out.println("7.Print List In Same Order");
            System.out.println("8.Print List In Reverse Order ");
            System.out.print("Enter choice  :   ");
            return sc.nextInt();
        }
        // public static void iterateLinkedList( LinkedList2 list )
{
        // for( int element : list )
        // System.out.print(element+" ");
        // System.out.println();
        // }
        public static void main(String[] args) {
            int choice;
            int[] data = new int[1];
```

```
        int[] position = new int[1];
        LinkedList2 list = new LinkedList2();
        while ((choice = LinkedList.menuList()) != 0) {
            switch (choice) {
                case 1:
                    LinkedList.acceptRecord(data);
                    list.addFirst(data[0]);
                    break;
                case 2:
                    LinkedList.acceptRecord(data);
                    list.addLast(data[0]);
                    break;
                case 3:
                    LinkedList.acceptRecord(data);
                    LinkedList.acceptPosition(position);
                    list.addAtPosition(data[0], position[0]);
                    break;
                case 4:
                    list.removeFirst();
                    break;
                case 5:
                    list.removeLast();
                    break;
                case 6:
                    LinkedList.acceptPosition(position);
                    list.removeAtPosition(position[0]);
                    break;
                case 7:
                    // Program.iterateLinkedList(list);
                    System.out.println("LL in same order: ");
                    list.display();
                    break;
                case 8:
                    System.out.println("Linked list in reverse
order print: ");
                    list.reverse(list.head);
                    break;
                default:
                    System.out.println("Invalid choice");
                    break;
            }
```

```
        }
        list.clear();
    }
}
```

Singly Linked List Time Complexity

Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Limitations of Singly Linear Linked List:

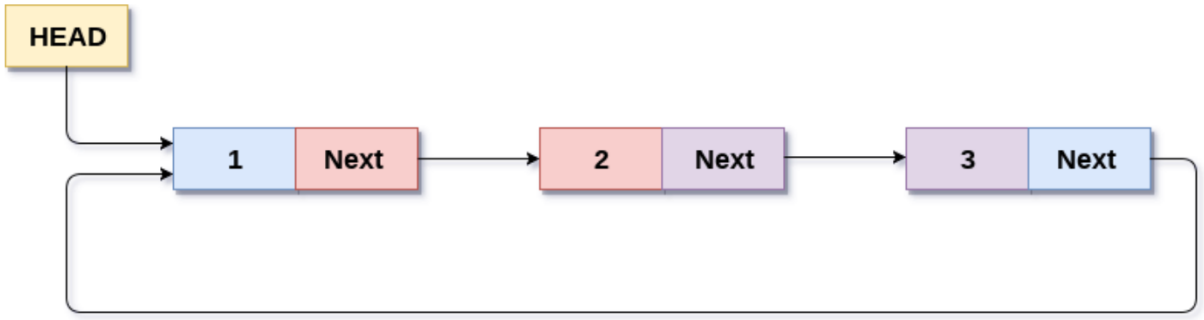
- Add node at last position & delete node at last position operations are not efficient as it takes O(n) time.
- We can start traversal only from first node and can traverse the list only in a forward direction.
- Previous node of any node cannot be accesed from it.
- Any node cannot be revisited
- to overcome this limitation Singly Circular Linked List has been designed.

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13

marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Actual Implementation Of Circular Singly Linked List :-

```
class LinkedList{

    static class Node{
        private int data;
        private Node next;

        public Node(int data){
            this.data = data;
            this.next = null;
        }
    }//end of Node class

    private Node head;//head is a reference of type Node class
    private int nodesCount;

    public LinkedList(){
        this.head = null;
        this.nodesCount=0;
    }

    boolean isListEmpty( ){
        return ( head == null );
    }

    void addNodeAtLastPosition(int data){
        //step-1: create a newnode
        Node newNode = new Node(data);
```

```
        //step-2: //if list is empty then attach newly created node to
the head
        if( isListEmpty() ){
            head = newNode;
            //attach first node into the next part of newly created
node added at last position
            newNode.next = head;
            nodesCount++;
        }else{//step-3: if list is not empty
            //- start traversal of the list from first node
            Node trav = head;
            //step-4: traverse the list till last node
            while( trav.next != head ){
                trav = trav.next;
            }
            //step-5: attach newly created node to the head
            trav.next = newNode;
            //step-6: attach first node into the next part of newly
created node added at last position
            newNode.next = head;
            nodesCount++;
        }
    }

    int getNodesCount(){
        return ( this.nodesCount );
    }

    void displayList( ){
        if(isListEmpty() )
            throw new RuntimeException("list is empty");
        else{//if list is not empty
            System.out.print("list is : ");
            //start tarversal of the list first node
            Node trav = head;

            //traverse the list till last node including it
            do{
                System.out.printf("%4d", trav.data);
                trav = trav.next;
```

```
            }while( trav != head );

            System.out.println();
            System.out.println("no. of nodes in a list are :
"+getNodesCount());
        }
    }
}//end of LinkedList class

public class SingCirLinkedListMain {
    public static void main(String[] args) {
        //create an empty linked list
        LinkedList l1 = new LinkedList();

        l1.addNodeAtLastPosition(11);
        l1.addNodeAtLastPosition(22);
        l1.addNodeAtLastPosition(33);
        l1.addNodeAtLastPosition(44);
        l1.addNodeAtLastPosition(55);
        l1.addNodeAtLastPosition(66);

        try{
            l1.displayList();
        }catch(RuntimeException e){
            System.out.println( e.getMessage() );
        }
    }
}
```

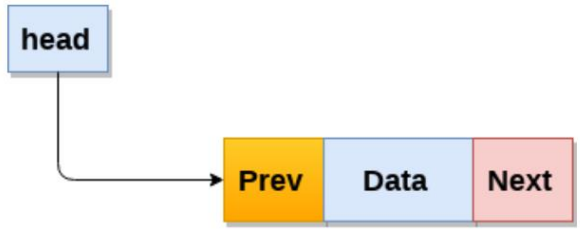
limitations of scll:

- in scll, addLast(), addFirst(), deleteLast() & deleteFirst() operations are not efficient as it takes O(n) time.
- we can traverse scll only in a forward direction
- prev node of any node cannot be accessed from it
- to overcome limitations of singly linked list (slll & scll) doubly linked list has been designed.

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three

parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Actual Implementation Of Doubly Linked List:-

```
import java.util.Scanner;
class LinkedList{
    static class Node{
        private int data;
        private Node prev;//prev is a reference of type Node class
        private Node next;//next is a reference of type Node class

        Node(int data){
            this.data = data;
            this.prev = null;
            this.next = null;
        }
    }//end of Node class

    private Node head;
    private int nodesCount;
    LinkedList(){
        head = null;
        nodesCount = 0;
```

```
    }

    boolean isListEmpty( ){
        return ( head == null );
    }

    int getNodesCount( ){
        return ( this.nodesCount );
    }

    void addNodeAtLastPosition(int data){
        //step-1: create a newnode
        Node newNode = new Node(data);
  //step-2: if list is empty -> attach newly created node to the head
        if( isListEmpty( ) ){
            head = newNode;
            nodesCount++;
        }else{//step-3: if list is not empty
            //start traversal of the list from first node
            Node trav = head;
            //traverse the list till last node
            while( trav.next != null )
                trav = trav.next;//move trav towards its next node
          //attach newly created node to the last node
            trav.next = newNode;
      //attach cur last node into the prev part of newly created node
            newNode.prev = trav;
            nodesCount++;
        }
    }

    void addNodeAtFirstPosition(int data){
        //step-1: create a newnode
        Node newNode = new Node(data);
   //step-2: if list is empty -> attach newly created node to the head
        if( isListEmpty( ) ){
            head = newNode;
            nodesCount++;
        }else{//step-3: if list is not empty
```

```
      //attach cur first node into the next part of newly created node
            newNode.next = head;
   //attach newly created node into the prev part of cur first node
            head.prev = newNode;
            //attach newly created node to the head
            head = newNode;
            nodesCount++;
        }
    }

    void displayList( ){
        if( isListEmpty() )
            throw new RuntimeException("list is empty");
        else{//if list is not empty
            //start traversal of the list from first node
            Node trav = head;
            Node temp = null;

            System.out.print("list in a forward dir is  : ");
            //traverse the list till last node
            while( trav != null ){
                temp = trav;
                System.out.printf("%4d", trav.data);
                trav = trav.next;
            }
            System.out.println();

            //start traversal from last node
            trav = temp;
            System.out.print("list in a backward dir is : ");
            //traverse the list till first node
            while( trav != null ){
                temp = trav;
                System.out.printf("%4d", trav.data);
                trav = trav.prev;//move trav towards its prev node
            }
            System.out.println();

  System.out.println("no. of nodes in a list are : "+getNodesCount());
        }
```

```
    }

    void addNodeAtSpecificPosition(int pos, int data){
        if( pos == 1 )//if pos is the first position
            addNodeAtFirstPosition(data);
        else if( pos == getNodesCount() + 1 )//if pos is the last
position
            addNodeAtLastPosition(data);
        else{//if pos is in between position

            //create a newnode
            Node newNode = new Node(data);

            int i=1;
            //start traversal from first node
            Node trav = head;
            //traverse the list till (pos-1)th node
            while( i < pos-1 ){
                i++;
                trav = trav.next;
            }
   //attach cur (pos)th node into next part of newly created node
            newNode.next = trav.next;
  //attach (pos-1)th node into prev part of newly created node
            newNode.prev = trav;
 //attach newly created node into the prev part of cur (pos)th node
            trav.next.prev = newNode;
 //attach newly created node into the next part of cur (pos-1)th node
            trav.next = newNode;
            nodesCount++;
        }
    }

}//end of LinkedList class

public class DoublyLinLinkedListMain {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int pos;
```

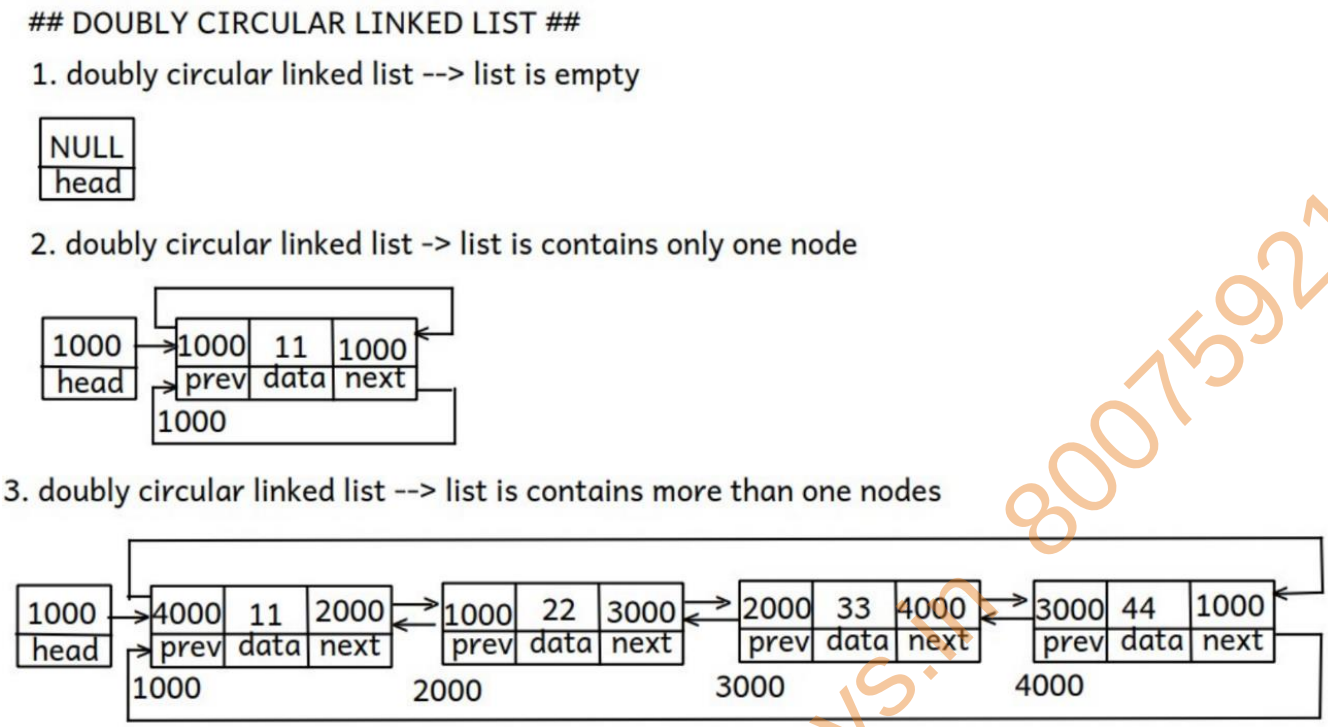
```
        //create an empty linked list
        LinkedList l1 = new LinkedList();

        l1.addNodeAtLastPosition(11);
        l1.addNodeAtLastPosition(22);
        l1.addNodeAtLastPosition(33);
        l1.addNodeAtLastPosition(44);
        l1.addNodeAtLastPosition(55);
        l1.addNodeAtLastPosition(66);
        try{
            l1.displayList();
        }catch(RuntimeException e){
            System.out.println(e.getMessage());
        }
        //l1.addNodeAtFirstPosition(5);
        while( true ){
            //step-1: accept position from user
            System.out.print("enter the pos : ");
            pos = sc.nextInt();

            //step-2: validate position
            if( pos >= 1 && pos <= l1.getNodesCount() + 1 )
                break;

            System.out.println("invalid position");
        }
        l1.addNodeAtSpecificPosition(pos, 99);
        try{
            l1.displayList();
        }catch(RuntimeException e){
            System.out.println(e.getMessage());
        }
    }
}
```

limitations of dlll:

- in dlll, addLast() and deleteLast() operations are not efficient as it takes O(n) time.
- we can start traversal only from first node in O(1) time.
- to overcome limitations of dlll, dcll linked list has been designed.

Circular Doubly Linked List

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



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

Memory Management of Circular Doubly linked list

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1. Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4. The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1. In circular doubly linked list, the last

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.

Advantages of Doubly Circular Linked List:

- DCLL can be traverse in forward as well as in a backward direction.
- Add last, add first, delete last & delete first operations are efficient as it takes O(1) time and are convenient as well.
- Traversal can be start either from first node (i.e. from head) or from last node (from head.prev) in O(1) mtime.
- Any node can be revisited.
- Previous node of any node can be accessed from it
- # Array v/s Linked List => Data Structure:
- Array is static data structure whereas linked list is dynamic data structure.
- Array elements can be accessed by using random access method which is efficient than sequential access method used to access linked list elements.
- Addition & Deletion operations are efficient on linked list than on an array.
- Array elements gets stored into the stack section, whereas linked list elements gets stored into heap section.
- In a linked list extra space is required to maintain link between elements, whereas in an array to maintained link between elements is the job of the compiler.
- searching operation is faster on an array than on linked list as on linked list we cannot apply binary search.

 # Applications of linked list in computer science

- To implementation of stacks and queues
- To implement advanced data structures like tree, hash table, graph
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list
- representing sparse matrices

 # Applications of linked list in real world :

- Image viewer : Previous and next images are linked, hence can be accessed by next and previous button.
- Previous and next page in web browser
- We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list. - Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Stack:

It is a collection/list of logically related similar type elements into which data elements can be added as well as deleted from only one end referred top end.
In this collection/list, element which was inserted last only can be deleted first, so this list works in last in first out/first in last out manner, and hence it is also called as LIFO list/FILO list.
We can perform basic three operations on stack in O(1) time: Push, Pop & Peek.

1. Push : to insert/add an element onto the stack at top position
    step1: check stack is not full
    step2: increment the value of top by 1
    step3: insert an element onto the stack at top position.

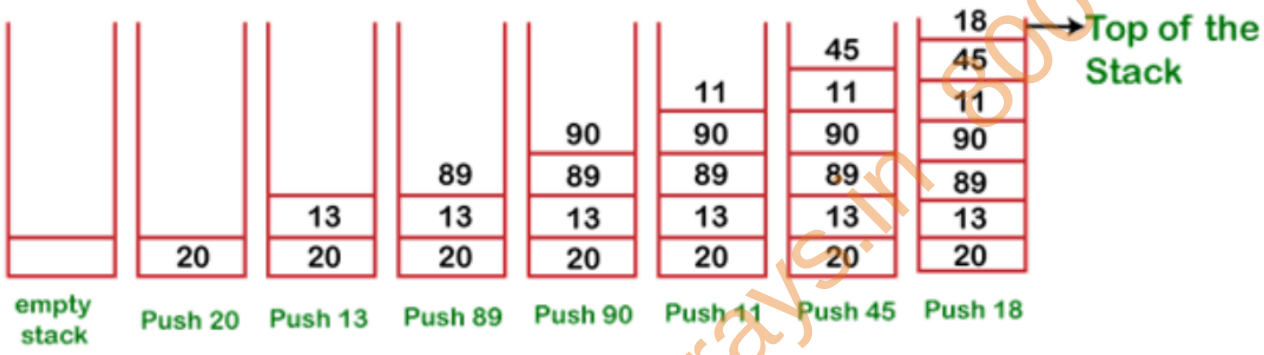
2. Pop : to delete/remove an element from the stack which is at top position
    step1: check stack is not empty
    step2: decrement the value of top by 1

3. Peek : to get the value of an element which is at top position without push & pop.
    step1: check stack is not empty
    step2: return the value of an element which is at top position

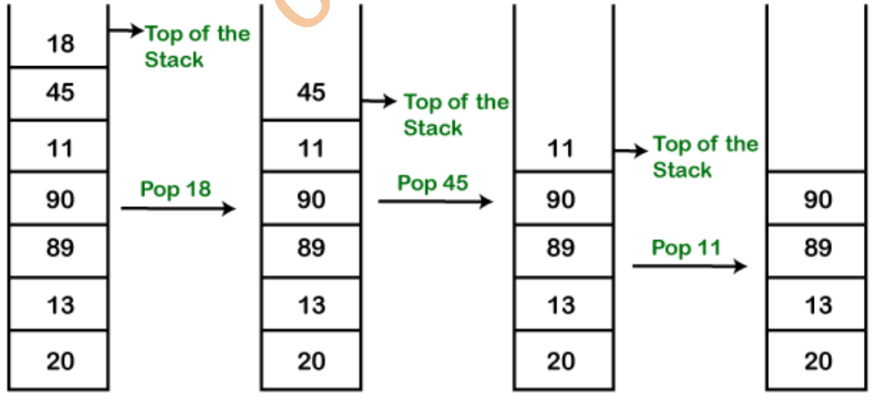
Stack Empty : top == -1
Stack Full : top == SIZE-1
 # Applications of Stack:

- Stack is used by an OS to control of flow of an execution of program.
- In recursion internally an OS uses a stack.
- undo & redo functionalities of an OS are implemented by using stack.
- Stack is used to implement advanced data structure algorithms like DFS: Depth First Search traversal in tree & graph.
- Stack is used in an algorithms to covert given infix expression into its equivalent postfix and prefix, and for postfix expression evaluation.

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.



Push operation



Pop operation

Actual Implementation Of Stack:-

```
import java.util.Scanner;
class Stack {
    private int arr[];
    private int top;
    public Stack(int size) {
        arr = new int[size];
        top = -1;
    }
    public void push(int value) {
        if (isFull()) {
            throw new RuntimeException("Stack is full");
        }
        top++;
        arr[top] = value;
    }
    public void pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is Empty");
        }
        top--;
    }
    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is Empty");
        }
        return arr[top];
    }
    public boolean isEmpty() {
        return top == -1;
    }
    public boolean isFull() {
        return top == arr.length - 1;
    }
}
class StackMain {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Stack s = new Stack(10);
        int choice = 0;
        int value;
        do {
```

```
            System.out.println("\n 0.Exist");
            System.out.println("\n 1.Push");
            System.out.println("\n 2.Pop");
            System.out.println("\n 3.Peek");
            System.out.println("\n Enter choice: ");
            choice = sc.nextInt();
            switch (choice) {
                case 1:
                    try {
                        System.out.println("Enter value to push");
                        value = sc.nextInt();
                        s.push(value);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                    break;
                case 2:
                    try {
                        value = s.peek();
                        s.pop();
                        System.out.println("Popped : " + value);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                    break;
                case 3:
                    try {
                        value = s.peek();
                        System.out.println("Peek: " + value);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                    break;
                default:
                    System.out.println("Invalid choice");
                    break;
            }
        } while (choice != 0);
        sc.close();
    }
}
```

- Infix, prefix, postfix are expression notations.
- Infix means operator in-between operant => A + B
- Prefix means operator is before the operant => + A B
- Postfix means operator is after the operant => A B +
- Infix -- > result
- Infix -- > postfix -- > result
- Infix -- > prefix -- > result

```
Precedence :1)   ()
            2)    $          $ means x^y
            3)  * / %
            4)  + -
```

- **Infix to Postfix**

```
                6   7   5    2   1     8    4     3
Question    5 + 9 - 4 * ( 8 - 6 / 2 ) + 1 $ ( 7 - 3 )

Step 1:     5 + 9 - 4 * ( 8 - 6 2 / )  + 1 $ ( 7 - 3 )

Step 2:     5 + 9 - 4 *  8  6 2 / -  + 1 $ ( 7 - 3 )

Step 3:     5 + 9 - 4 *  8  6 2 / -  + 1 $  7  3 -

Step 4:     5 + 9 - 4 *  8  6 2 / -  + 1  7  3 - $

Step 5:     5 + 9 - 4  8  6 2 / - *  + 1  7  3 - $

Step 6:     5  9 + - 4  8  6 2 / - *  + 1  7  3 - $

Step 7:     5  9 +  4  8  6 2 / - * -  + 1  7  3 - $

Step 8:     5  9 +  4  8  6 2 / - * - 1  7  3 - $  +
```

- **Infix to Prefix**

```
                6   7   5     2   1     8   4     3
Question    5 + 9 - 4 * ( 8 - 6 / 2 ) + 1 $ ( 7 - 3 )

Step 1:     5 + 9 - 4 * ( 8 - /6 2 ) + 1 $ ( 7 - 3 )

Step 2:     5 + 9 - 4 * - 8  /6 2  + 1 $ ( 7 - 3 )
Step 3:     5 + 9 - 4 * - 8  /6 2  + 1 $ - 7 3

Step 4:     5 + 9 - 4 * - 8  /6 2  + $ 1  - 7 3
```

```
Step 5:    5 + 9 – * 4  - 8  /6 2  + $ 1  - 7 3

Step 6:    + 5 9 – * 4  - 8  /6 2  + $ 1  - 7 3

Step 7:    - + 5 9  * 4  - 8  /6 2  + $ 1  - 7 3

Step 8:    + - + 5 9  * 4  - 8  /6 2  $ 1  - 7 3
```

Postfix Evaluation implementation:

Question 5 9 + 4 8 6 2 / - * - 1 7 3 - $ +

1)traverse postfix from left to right
2) if symbol is operand ,push on stack
3) if symbol is operator ,pop two arguments
 From stack ,calculate result & push on stack, first poped 2nd argument
 Second poped 1st argument
4) repeat 2 & 3 step until all values from expression.
5) pop the final result from stack.

```
import java.util.Stack;
class PostfixEvaluation {
    public static int calc(int a,int b,char operator) {
        switch (operator) {
            case '$': return (int) Math.pow(a, b);
            case '*': return a * b;
            case '/': return a / b;
            case '%': return a % b;
            case '+': return a + b;
            case '-': return a - b;
        }
        return 0;
    }
    public static int solvePostfix(String post) {
        //Stack of operand
        Stack<Integer> s=new Stack<Integer>();
        //traverse postfic from left to right
        for (int i = 0; i < post.length(); i++) {
            //get eachsym from expression
            char sym=post.charAt(i);
```

```
            //if sym is operand
            if (Character.isDigit(sym)) {
                // s.push(sym);
                //convert it to int push on stack
                //e.g '5' -->tostring--> "5"-->parseInt()-->5
                String operand=Character.toString(sym);
                s.push(Integer.parseInt(operand));
            }
            else{
                //pop two operands from stack
                int b=s.pop();
                int a=s.pop();
                // int c= a sym b;
                //calculate the result
                int c=calc(a, b, sym);
                //push the result on stack
                s.push(c);
            }
        }//repeat for all syms in expression
        //pop final result from stack and return
        return s.pop();
    }
public static void main(String[] args) {
    PostfixEvaluation b=new PostfixEvaluation();
    String postfix="59+4862/-*-173-$+";
    int result=solvePostfix(postfix);
    System.out.println("Result : "+result);
    }
}
```

Prefix Evaluation implementation:

Question: + - + 5 9 * 4 – 8 / 6 2 $ 1 – 7 3

1)traverse prefix from right to left
2) if symbol is operand ,push on stack
3) if symbol is operator ,pop two arguments

From stack ,calculate result & push on stack, first poped 1st argument Second poped 2nd argument
4) repeat 2 & 3 step until all values from expression.
5) pop the final result from stack.

```
import java.util.Stack;
class PrefixEvaluation {
    public static int calc(int a,int b,char operator) {
        switch (operator) {
            case '$': return (int) Math.pow(a, b);
            case '*': return a * b;
            case '/': return a / b;
            case '%': return a % b;
            case '+': return a + b;
            case '-': return a - b;
        }
        return 0;
    }
    public static int solvePrefix(String pre) {
        //Stack of operand
        Stack<Integer> s=new Stack<Integer>();
        //traverse prefix from right to left
        for (int i = pre.length()-1; i >=0; i--) {
            //get eachsym from expression
            char sym=pre.charAt(i);
            //if sym is operand
            if (Character.isDigit(sym)) {
                // s.push(sym);
                //convert it to int push on stack
                //e.g '5' -->tostring--> "5"-->parseInt()-->5
                String operand=Character.toString(sym);
                s.push(Integer.parseInt(operand));
            }
            else{
                //pop two operands from stack
                int a=s.pop();
```

```
                int b=s.pop();
                // int c= a sym b;
                //calculate the result
                int c=calc(a, b, sym);
                //push the result on stack
                s.push(c);
            }
        }//repeat for all syms in expression
        //pop final result from stack and return
        return s.pop();
    }

public static void main(String[] args) {
    String prefix="+-+59*4-8/62$1-73";
    int result=solvePrefix(prefix);
    System.out.println("Result : "+result);
    }
}
```

Infix to Postfix implementation:

1)traverse infix from left to right.
2)if operand found, append to postfix.
3)if operator found, push to stack
- If priority of topmost operator on stack greater than equal to Priority of current operator, pop it from stack & append to postfix.

4)if opening (parenthesis found, push it on stack.
5)if closing) found, pop operator from stack & append to postfix until opening is found also pop & discard opening (from stack.
6)if all symbol from infix is completed, pop one by one & append to postfix.

```
import java.util.Stack;
class infixToPostfix {
    public static int pri(char operator) {
        switch (operator) {
            case '$': return 10;
```

```
            case '*': return 7;
            case '/': return 7;
            case '%': return 7;
            case '+': return 3;
            case '-': return 3;
        }
        return 0;
    }
    public static String solInfixToPostfix(String infix) {
            Stack<Character> s=new Stack<Character>();
            StringBuilder post=new StringBuilder();
            for (int i = 0; i <infix.length(); i++) {
                char sym=infix.charAt(i);
                if (Character.isDigit(sym)) {
                   post.append(sym);
                }
                else if (sym=='(') {
                        s.push(sym);
                    }
                else if (sym==')') {
                    while (s.peek()!='(') {
                    post.append(s.pop());
                  }
                    s.pop();
                        }
                else{
         while (!s.empty() && pri(s.peek()) >= pri(sym)) {
                        post.append(s.pop());
                    }
                    s.push(sym);
                }
            }
            while (!s.isEmpty()) {
                post.append(s.pop());
            }
            return post.toString();
```

```
            }
public static void main(String[] args) {
    String infix="5+9-4(8-6/2)+1$(7-3)";
    System.out.println("The given infix expression: 5+9-4(8-
6/2)+1$(7-3) ");
    String postfix=solInfixToPostfix(infix);
    System.out.println("infix to postfix result: "+postfix);
    }
}
```

Infix To Prefix implementation:

```
import java.util.Stack;
class infixToPrefix {

        public static int pri(char operator) {
            switch (operator) {
                case '$': return 10;
                case '*': return 7;
                case '/': return 7;
                case '%': return 7;
                case '+': return 3;
                case '-': return 3;
            }
            return 0;
        }
        public static String solInfixToPrefix(String infix) {
  /*  1.traverse infix expression from right to left.
      2.if operand is found append to prefix.
      3.if operator is found push it on the stack.
        If priority of topmost operator on stack greater than
Priority of current operator, pop it from stack & append to
prefix.
      4.if closing ) is found then push it on stack.
      5.if opening ( is found pop all operators from stack one
bye one and append to prefix also pop and discard closing.
```

```
      6.when all sym from infix are done pop all operator from
stack one by one and append to prefix.
      7.Reverse the prefix and return.
 */
                Stack<Character> s=new Stack<Character>();
                StringBuilder pre=new StringBuilder();
                for (int i = infix.length()-1; i>=0; i--) {
                    char sym=infix.charAt(i);
                    if (Character.isDigit(sym)) {
                       pre.append(sym);
                    }
                    else if (sym==')') {
                            s.push(sym);
                        }
                    else if (sym=='(') {
                        while (s.peek()!=')') {
                        pre.append(s.pop());
                      }
                        s.pop();
                                }
                    else{
       while (!s.empty() && pri(s.peek()) > pri(sym)) {
                            pre.append(s.pop());
                        }
                        s.push(sym);
                    }
                }
                while (!s.isEmpty()) {
                    pre.append(s.pop());
                }
                return pre.reverse().toString();
                }
    public static void main(String[] args) {
        String infix="5+9-4(8-6/2)+1$(7-3)";
        System.out.println("The given infix expression: 5+9-
4(8-6/2)+1$(7-3) ");
```

```
        String prefix=solInfixToPrefix(infix);
        System.out.println("infix to prefix result: "+prefix);
        }
    }
```

Parenthesis Balanced implementation:

```
import java.util.Stack;
class ParanthisisBalanced {
    public static boolean isBalanced(String expr) {
        Stack<Character> s=new Stack<Character>();
        String open="([{",close=")]}";
        //1.traverse expression from left to right
        for (int i = 0; i < expr.length(); i++) {
        char sym=expr.charAt(i);
        //2.if opening ( is found , push on stack
        if (open.indexOf(sym)!=-1) {
            s.push(sym);
        }//3.if closing ) is found,
        else if (close.indexOf(sym)!=-1) {
            //if stack is empty (some opening is less)
            if (s.empty())
                return false;
            // pop one from stack
            char pop=s.pop();
            //compare if they are matching if yes continue
            if (close.indexOf(sym)!=open.indexOf(pop)) {
               //if not matching return false.
                return false;
            }
        }//4.when whole expression is done if stack is
empty,return true else return false
    }
    return s.isEmpty();
```

```
}
    public static void main(String[] args) {
        String infix="5+9-4(8-6/2)+1$(7-3)";
        boolean balanced=isBalanced(infix);
        System.out.println("Balanced: "+balanced);
    }
}
```

Queue:

- It is a collection/list of logically related similar type of elements into which elements can be added from one end referred as rear end, whereas elements can be deleted from another end referred as a front end.
- In this list, element which was inserted first can be deleted first, so this list works in first in first out manner, hence this list is also called as FIFO list/LILO list.
- Two basic operations can be performed on queue in O(1) time.
    1. Enqueue: to insert/push/add an element into the queue from rear end.
    2. Dequeue: to delete/remove/pop an element from the queue which is at front end.

There are different types of queue:

1. Linear Queue (works in a fifo manner)
2. Circular Queue (works in a fifo manner)

3. Priority Queue:

- it is a type of queue in which elements can be inserted from rear end randomly (i.e. without checking priority), whereas an element which is having highest priority can only be deleted first.
- Priority queue can be implemented by using linked list, whereas it can be implemented efficiently by using binary heap.

3. **Double Ended Queue (deque)** : it is a type of queue in which elements can added as well as deleted from both the ends

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.

Applications of Queue

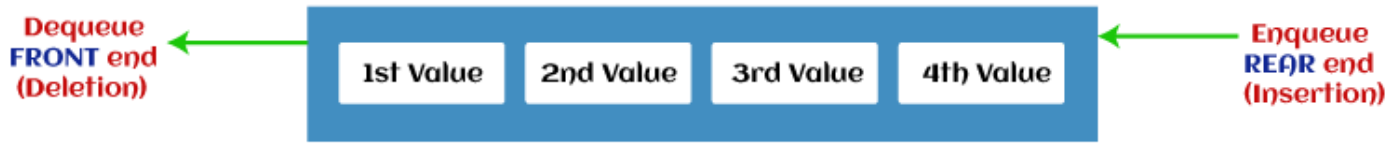
Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

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

Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Simple Queue or Linear Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.

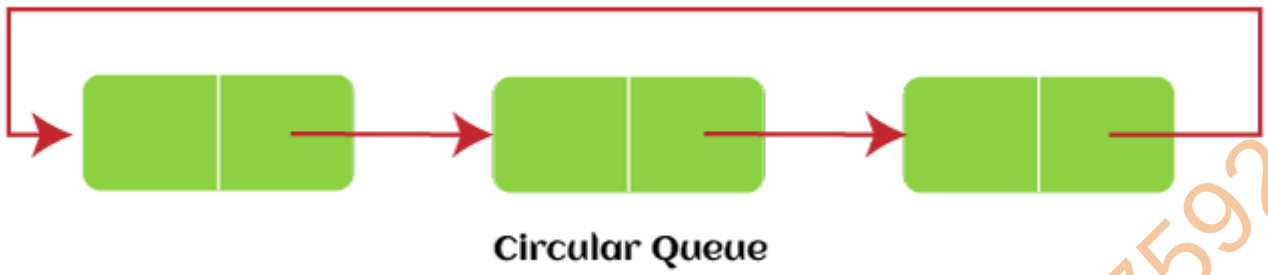


The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the

space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -



The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -

Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

- **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.

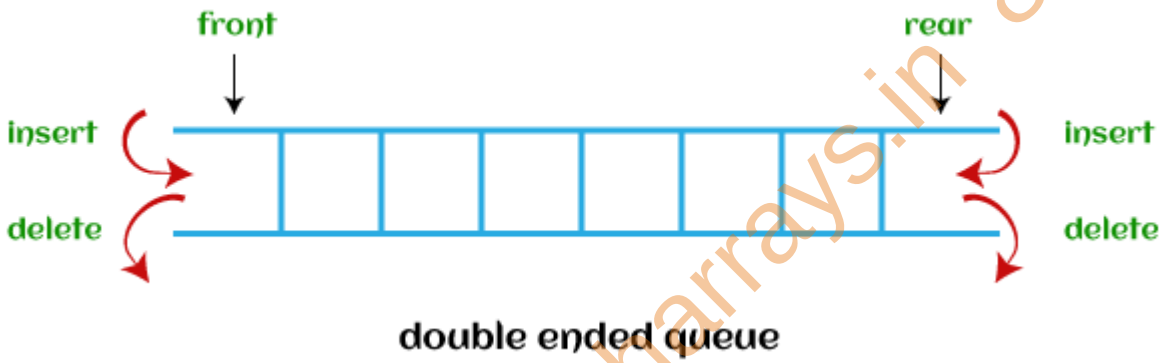
- **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

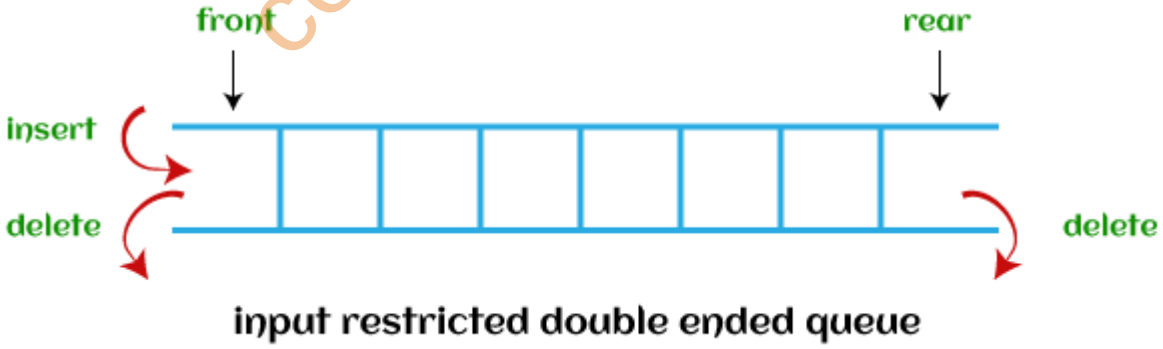
Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

The representation of the deque is shown in the below image -

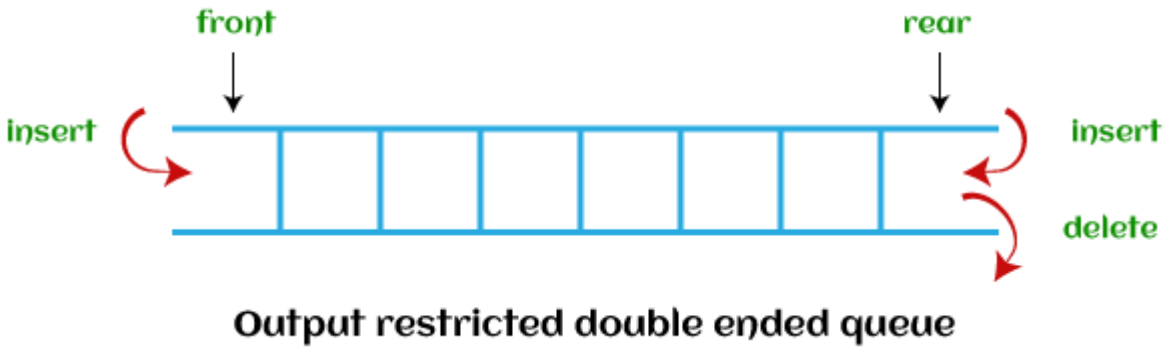


There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Now, let's see the operations performed on the queue

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

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

Now, let's see the ways to implement the queue.

Ways to implement the queue

There are two ways of implementing the Queue:

- **Implementation using array:** The sequential allocation in a Queue can be implemented using an
- **Implementation using Linked list:** The linked list allocation in a Queue can be implemented using a linked

---

Linear queue Implementation:

```
import java.util.Queue;
import java.util.Scanner;
class LinearQueue{
```

```
    private int[]arr;
    private int rear, front;
    public  LinearQueue(int size) {
        arr=new int[size];
        rear=-1;
        front=-1;
    }
    public boolean isFull() {
        return rear==arr.length-1;
    }
    public boolean isEmpty(){
        return rear==front;
    }
    public void push(int value) {
        if (isFull()) {
            throw new RuntimeException("Queue Is Full");
        }
        rear++;
        arr[rear]=value;
    }

    public void pop() {
        if (isEmpty()) {
            throw new RuntimeException("Queue Is Empty");
        }
        front++;
    }
    public int peek(){
        if (isEmpty()) {
            throw new RuntimeException("Queue Is Empty");
        }
        return arr[front+1];
    }
}

class LinearQueueMain {
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        LinearQueue Lq = new LinearQueue(10);
        int choice ,val;
        do {
            System.out.println("\n 0.Exist");
            System.out.println("\n 1.Push");
            System.out.println("\n 2.Pop");
            System.out.println("\n 3.Peek");
            System.out.println("\n Enter choice: ");
            choice = sc.nextInt();
            switch (choice) {
                case 1:
                    try {
                        System.out.println("Enter value to push");
                        val = sc.nextInt();
                        Lq.push(val);
                        System.out.println("Push :"+val);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                    break;
                case 2:
                    try {
                        val = Lq.peek();
                        Lq.pop();
                        System.out.println("Popped : " + val);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                    break;
                case 3:
                    try {
                        val = Lq.peek();
                        System.out.println("Peek: " + val);
                    } catch (Exception e) {
```

```
                        System.out.println(e.getMessage());
                    }
                    break;
                default:
                    System.out.println("Invalid choice");
                    break;
            }
        } while (choice != 0);
    }
}
```

Circular Queue:

```
1) If(rear==max-1)
        Rear=0;
   Else
        Rear=rear+1;

2) Rear=(rear+1) % max;
   Dry Run:
              Rear
               -1  --> 0
                  0   --> 1
                  1   --> 2
                  2   --> 3
                  3   --> 4
                  4   --> 5
                  5   --> 0
```

- In linear queue (using array) when rear reaches last index, further elements cannot be added, even if space is available due to deletion of elements from front. Thus space utilization is poor.
- Circular queue allows adding elements at the start of array if rear reaches last index and space is free at the start of the array.
- Thus rear and front can be incremented in circular fashion i.e 0 1,2,3… n-1. So they are said to be circular queue.
- However queue full and empty conditions become tricky.

Circular Queue Implementation:

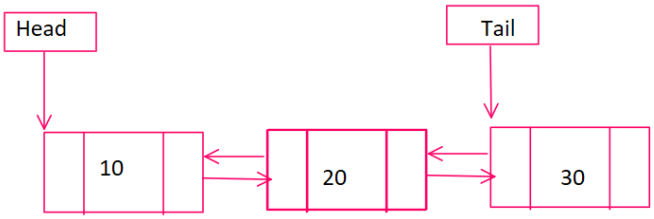
```
import java.util.Scanner;
class CircularQueue{
    private int[]arr;
    private int rear, front;
```

```
    public  CircularQueue(int size) {
        arr=new int[size];
        rear=-1;
        front=-1;
    }
    public boolean isFull() {
        return (front==-1 && rear==arr.length-1) || (front==rear && front!=-1);
    }
    public boolean isEmpty(){
        return (rear==front && front==-1);
    }
    public void push(int value) {
        if (isFull()) {
            throw new RuntimeException("Queue Is Full");
        }
        rear=(rear+1)%arr.length;
        arr[rear]=value;
    }

    public void pop() {
        if (isEmpty()) {
            throw new RuntimeException("Queue Is Empty");
        }
        front=(front+1)%arr.length;
        if(front==rear){
            front=-1;
            rear=-1;
        }
    }
    public int peek(){
        if (isEmpty()) {
            throw new RuntimeException("Queue Is Empty");
        }
        int index=(front+1)%arr.length;
        return arr[index];
    }
```

```
}

class CircularQueueMain {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        CircularQueue CQ = new CircularQueue(10);
        int choice ,val;
        do {
            System.out.println("\n 0.Exist");
            System.out.println("\n 1.Push");
            System.out.println("\n 2.Pop");
            System.out.println("\n 3.Peek");
            System.out.println("\n Enter choice: ");
            choice = sc.nextInt();
            switch (choice) {
                case 1:
                    try {
                        System.out.println("Enter value to push");
                        val = sc.nextInt();
                        CQ.push(val);
                        System.out.println("Push :"+val);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                    break;
                case 2:
                    try {
                        val = CQ.peek();
                        CQ.pop();
                        System.out.println("Popped : " + val);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                    break;
                case 3:
                    try {
```

```
                        val = CQ.peek();
                        System.out.println("Peek: " + val);
                    } catch (Exception e) {
                        System.out.println(e.getMessage());
                    }
                    break;

                default:
                    System.out.println("Invalid choice");
                    break;
            }
        } while (choice != 0);
    }
}
```

Dequeue: In double ended queue, values can be added or deleted from front end or rear end.



O(1)
Push-Front
Pop-Front
O(1)

O(1)
Push-back
Pop-back
O(1)

Priority Queue: Not a First-In-First-Out.

- In priority queue, element with highest priority is removed first.
- Internally elements are sorted in sorted manner.
- Can be implemented using array and that time use insertion sort logic.
- Using linked list also and that time use insertion sort.
  Push= O(n);
  Pop= O(1);
- Using heap (array representation of binary tree).
  Push=O(log n)
  Pop=O(log n)

```
import java.util.PriorityQueue;
```

```
class PriorityQueueMain {
    public static void main(String[] args) {
        //offer --> add
        //poll ---> remove

        PriorityQueue<Integer> q=new PriorityQueue<Integer>();
        q.offer(5);
        q.offer(9);
        q.offer(6);
        q.offer(2);
        while (!q.isEmpty()) {
            System.out.println(q.poll());
        }
    }
}//output: 2 5 6 9
```

Applications of Queue:

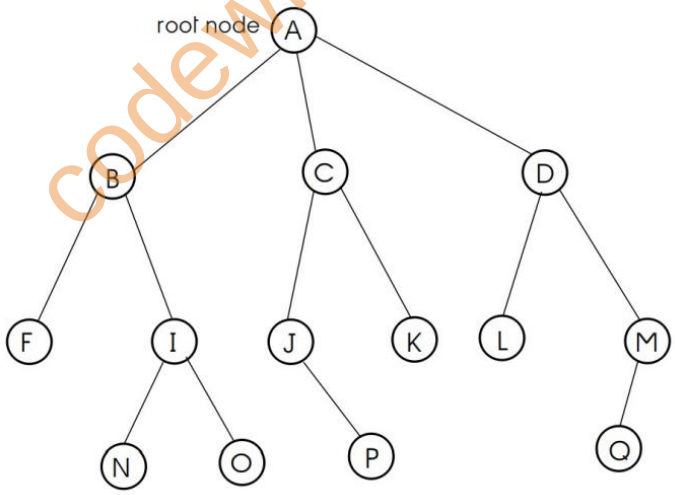
- Queue is used to implement OS data structures like job queue, ready queue, message queue, waiting queue etc...
- Queue is used to implement OS algorithms like FCFS CPU Scheduling, Priority CPU Scheduling, FIFO Page Replacement etc...
- Queue is used to implenent an advanced data structure algorithms like BFS: Breadth First Search Traversal in tree and graph.
- Queue is used in any application/program in which list/collection of elements should works in a first in first out manner or whereaver it should works according to priority.

Tree:

- It is a non-linear / advanced data structure which is a collection of finite no. of logically related similar type of data elements in which, there is a first specially designated element referred as a root element, and remaining all elements are connected to it in a hierarchical manner, follows parent-child relationship.

- **siblings/brothers:** child nodes of same parent are called as siblings.
- **ancestors:** all the nodes which are in the path from root node to that node.
- **descedents:** all the nodes which can be accessible from that node.
- **degree of a node** = no. of child nodes having that node
- **degree of a tree** = max degree of any node in a given tree
- **leaf node/external node/terminal node:** node which is not having any child node OR node having degree 0.
- **non-leaf node/internal node/non-terminal node:** node which is having any no. of child node/s OR node having non-zero degree.
- **level of a node** = level of its parent node + 1
- **level of a tree** = max level of any node in a given tree (by assuming level of root node is at level 0).
- **depth of a tree** = max level of any node in a given tree.

- as tree data structure can grow upto any level and any node can have any number of child nodes, operations on it becomes unefficient, so restrictions can be applied on it to achieve efficiency and hence there are diefferent types of tree.



Tree: Data Structure

Binary tree: it is a tree in which each node can have max 2 number of child nodes, i.e. each node can have either 0 OR 1 OR 2 number of child nodes.

OR

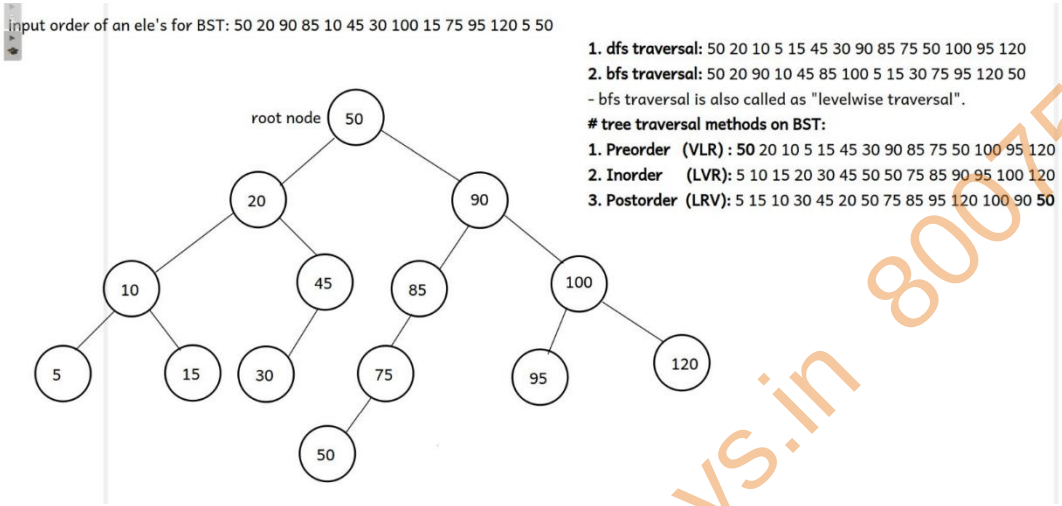
Binary tree: it is a set of finite number of elements having three subsets:

1. root element

2. left subtree (may be empty)

3. right subtree (may be empty)

- Binary Search Tree(BST): it is a **binary tree** in which left child is always smaller than its parent and right child is always greater than or equal to its parent.



Tree Data Structure

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

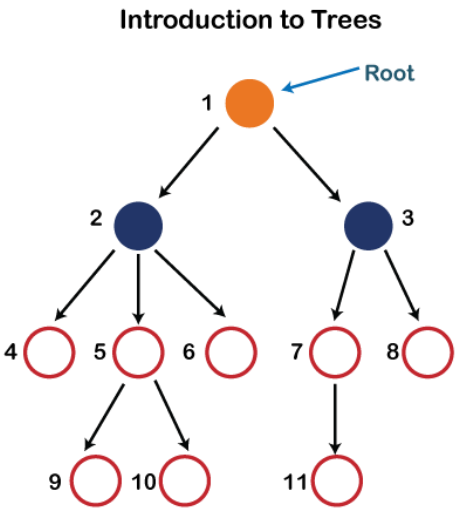
- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure.

Let's consider the tree structure, which is shown below:



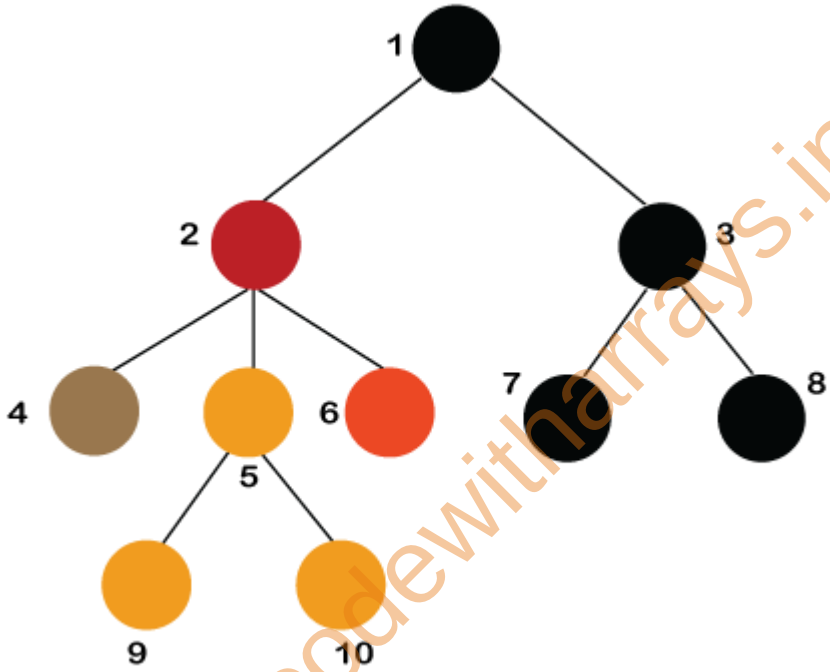
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.

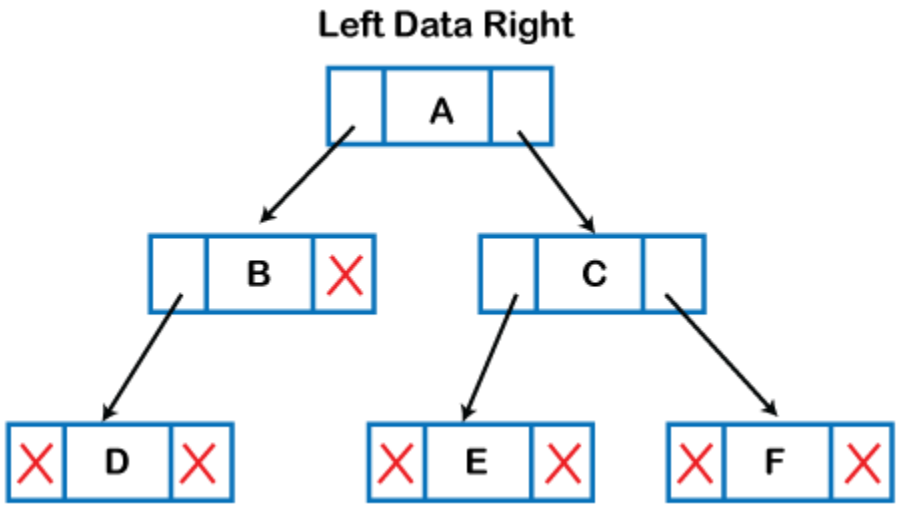


- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees

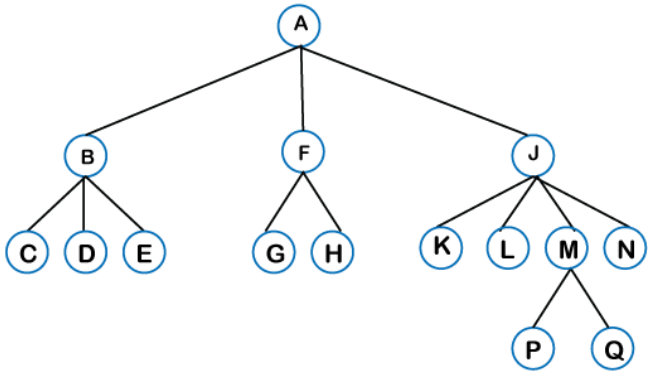
The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

The following are the types of a tree data structure:

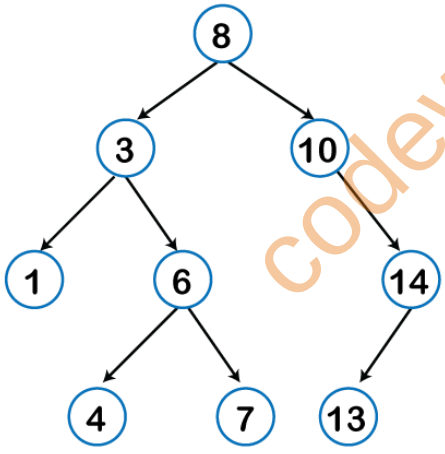
- **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.
Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

- **Binary tree**

: Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



To know more about the binary tree, click on the link given below:

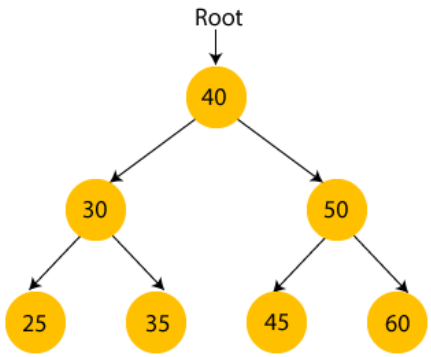
- **Binary Search tree**

  : Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer). Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

What is a Binary Search tree?

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

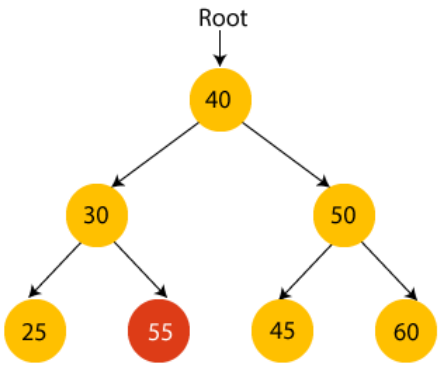
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree

Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

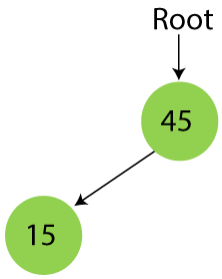
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.

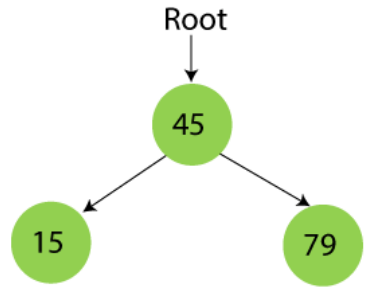
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



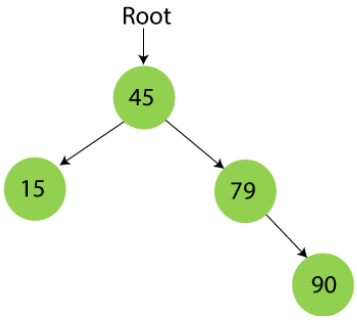
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



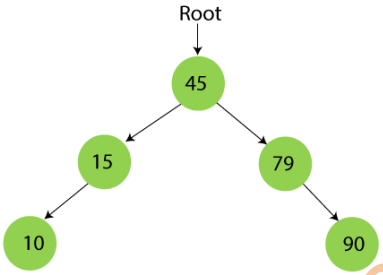
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



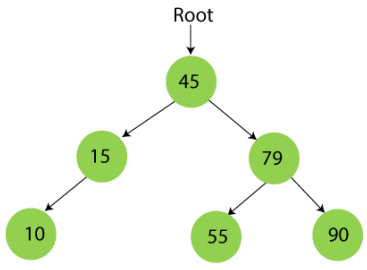
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



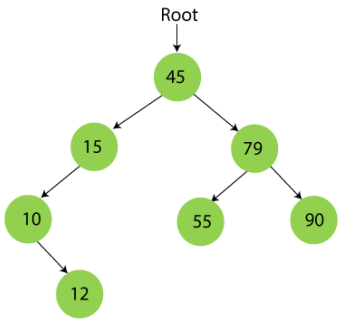
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



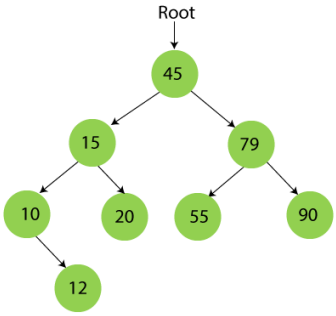
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



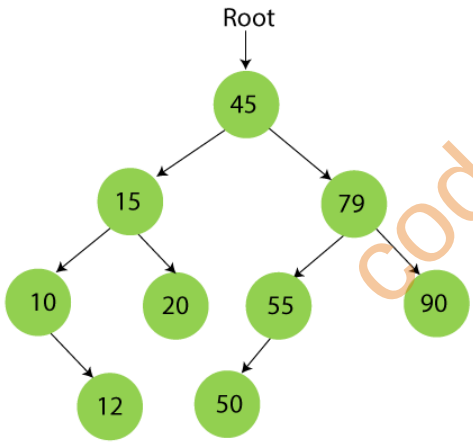
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

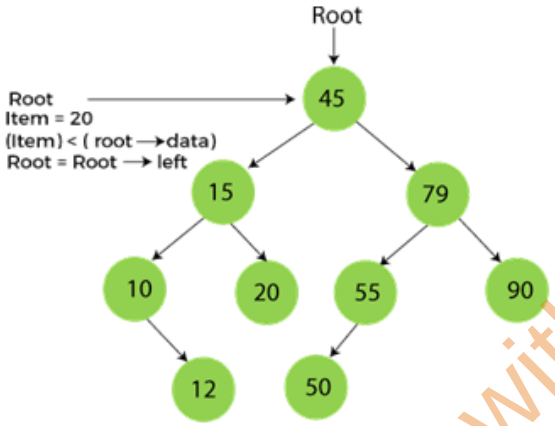
Searching in Binary search tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

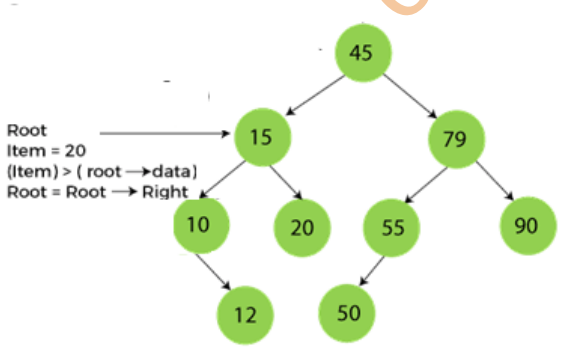
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

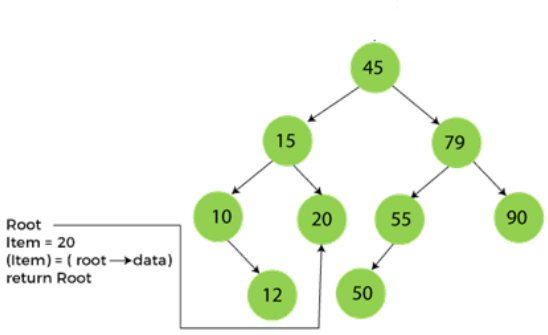
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Algorithm to search an element in Binary search tree

1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item **< root** → data)
5. return Search(root → left, item)
6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

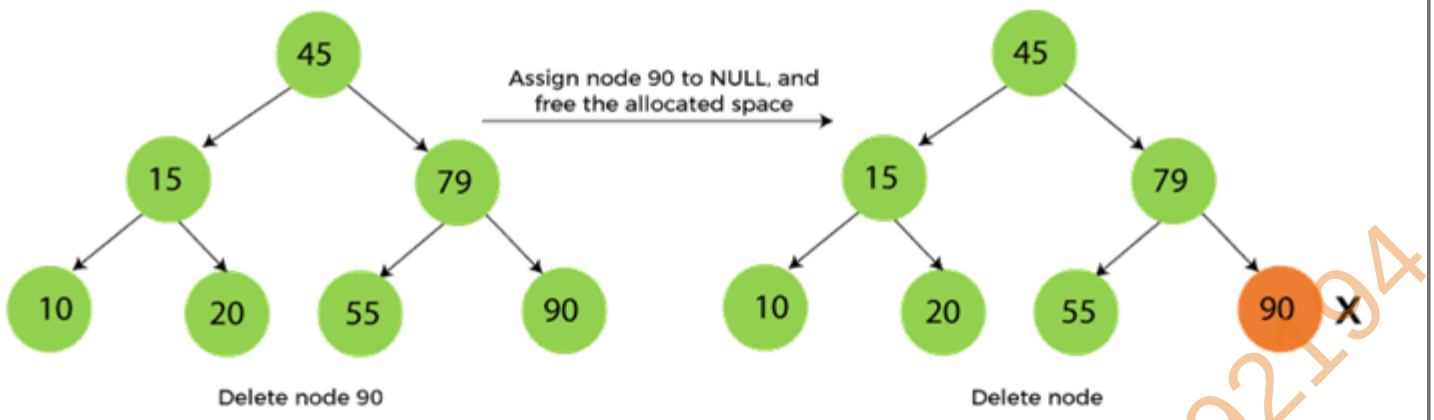
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

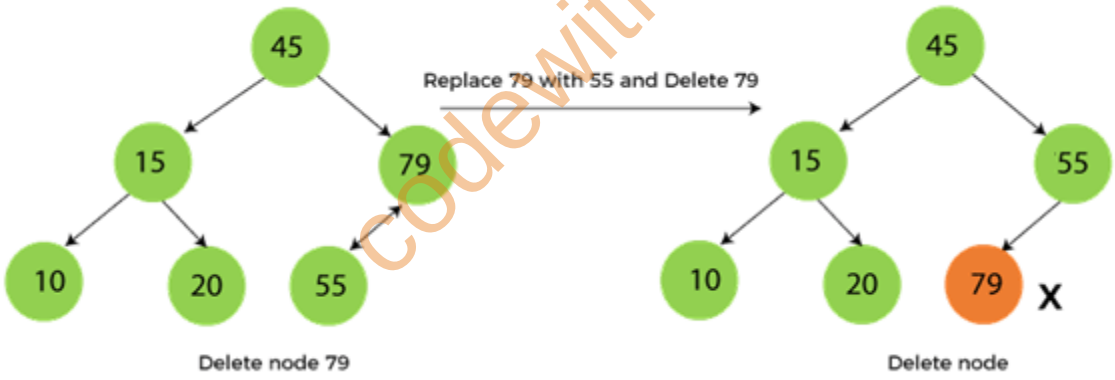


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



When the node to be deleted has two children

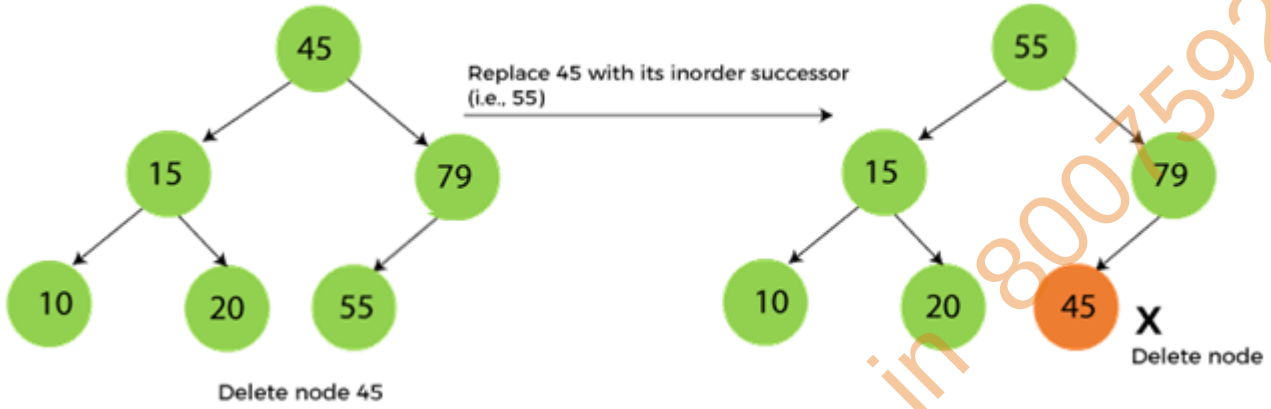
This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.

- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

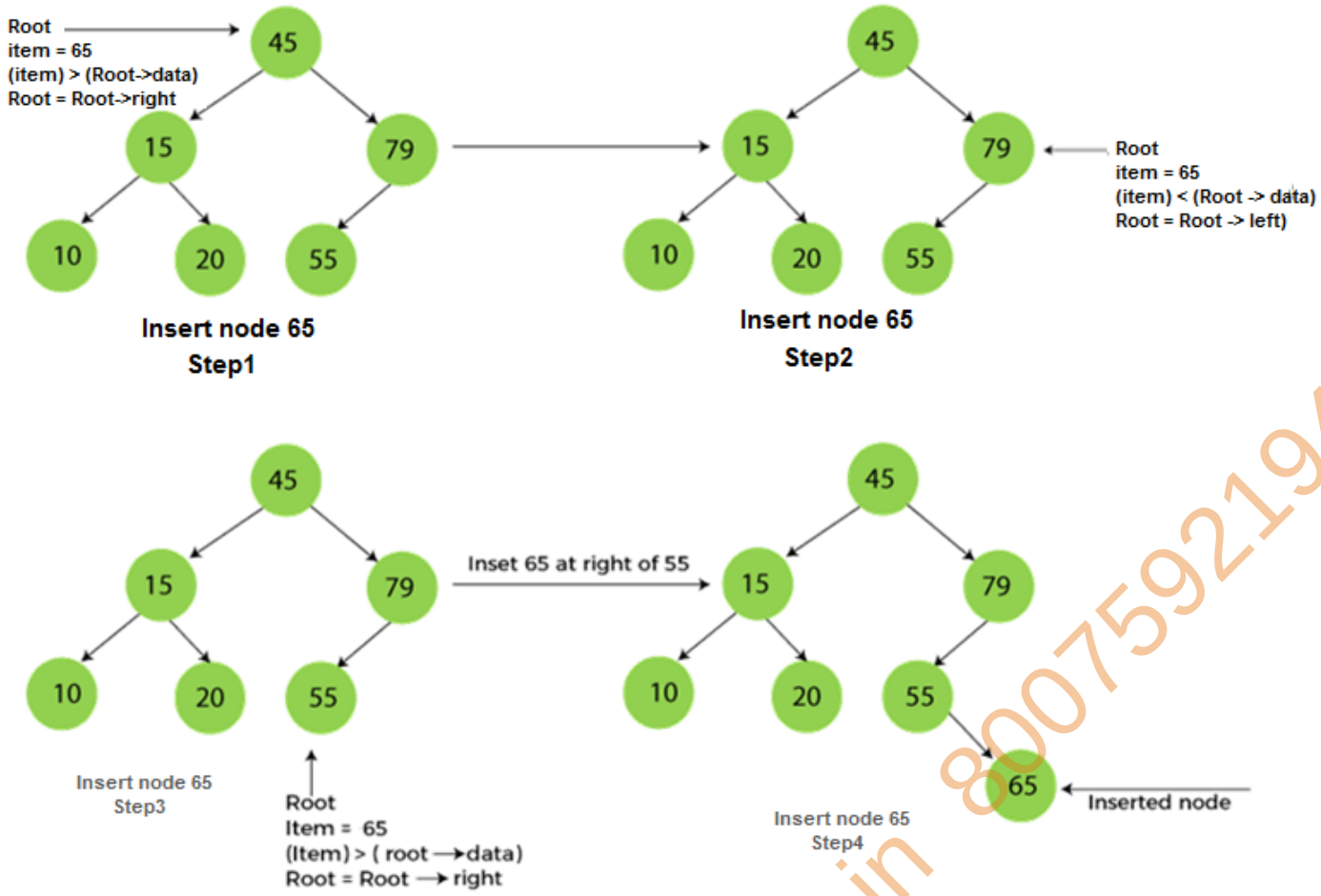


Now let's understand how insertion is performed on a binary search tree.

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.



The complexity of the Binary Search tree

Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

Implementation of Binary search tree

```
import java.util.Queue;
import java.util.Scanner;
import java.util.Stack;
import java.util.LinkedList;
class BinarySearchTree{
     class Node{
        private int data;
        private Node leftchild;
        private Node rightchild;
        public Node(){
            data=0;
            leftchild=null;
            rightchild=null;
        }
        public Node(int data){
            this.data=data;
            this.leftchild =null;
            this.rightchild=null;
        }
        public int getData(){
            return data;
        }
    }
        private Node root;
        public BinarySearchTree(){
            root=null;
        }
       public void add(int value){
            Node newNode=new Node(value);
            if (root==null) {
                root=newNode;
                return;
            }
            else{
                Node
trav=root;
                while (true) {
                    if (value< trav.data) {
                        if (trav.leftchild !=null) {
                            trav=trav.leftchild;
                        }
                        else{//no child in left
                            trav.leftchild=newNode;
                            break;
```

```
                    }
                }
                else{// if(value>=trav.data)
                    if (trav.rightchild!=null) {
                        trav=trav.rightchild;
                    }
                    else{//no child in right
                        trav.rightchild=newNode;
                        break;
                    }
                }
            }
        }
    }
    public void preorder(Node trav) {
        if (trav==null) {
            return;
        }
        System.out.print(trav.data+ ",");
        preorder(trav.leftchild);
        preorder(trav.rightchild);
    }
    //wrraper function
    public void preorderMethod() {
        System.out.println("Preorder output: ");
        if (root==null) {
            System.out.println("Tree is empty");
        }
        preorder(root);
        System.out.println();
    }
    /* Preorder output:
       50,30,10,20,40,90,70,60,80,100
    */
    public void inorder(Node trav) {
        if (trav==null) {
            return;
        }
        inorder(trav.leftchild);
        System.out.print(trav.data+ ",");
        inorder(trav.rightchild);
    }
    //wrraper function
    public void inorderMethod() {
        System.out.println("Inorder output: ");
        if (root==null) {
```

```
            System.out.println("Tree is empty");
        }
        inorder(root);
        System.out.println();
    }
    /*
    Inorder output:
    10,20,30,40,50,60,70,80,90,100,
    */
    public void Postorder(Node trav) {
        if (trav==null) {
            return;
        }
        Postorder(trav.leftchild);
        Postorder(trav.rightchild);
        System.out.print(trav.data+ ",");
    }
    //wrraper function
    public void PostorderMethod() {
        System.out.println("Postorder output: ");
        if (root==null) {
            System.out.println("Tree is empty");
        }
        Postorder(root);
        System.out.println();
    }
    /*
    Postorder output:
    20,10,40,30,60,80,70,100,90,50,
    */
    //For delete we need to follow postorder traversal.
    public void deleteAll(Node trav){
       if (trav==null) {
            return;
       }
        deleteAll(trav.leftchild);
        deleteAll(trav.rightchild);
        trav.leftchild=null;
        trav.rightchild=null;
        trav=null;
    }
    //wrapper function
    public void deleteAll() {
        deleteAll(root);
        root=null;
    }
```

```
    public int height(Node trav) {
        if (trav==null) {
            return -1;
        }
        int hl=height(trav.leftchild);
        int hr=height(trav.rightchild);
        int max=hl > hr ? hl : hr;
        return max + 1;
    }
    //wrapper function
    public int height() {
        return height(root);
    }

//preorder without recursion method
    public void PreorderNoRec(Node node) {
        Stack<Node> s=new Stack<>();
        if (node==null) {
            return;
        }
        Node trav= root;
        while (trav!=null ||!s.isEmpty()) {
            while (trav!=null) {
                System.out.print(trav.data+" ");
                if (trav.rightchild!=null) {
                    s.push(trav.rightchild);
                }
                trav=trav.leftchild;
            }
            if (!s.isEmpty()) {
                trav=s.pop();
            }
        }
    }
    public void preOrderNORecMethod(){
        System.out.println("preOrderNORec");
        PreorderNoRec(root);
        System.out.println();
    }
    //inorder without recursion method
    public void inOrderNoRecMethod(){
        System.out.println("inOrderNoRec");
        inOrderNoRec(root);
        System.out.println();
    }
```

```
    private void inOrderNoRec(Node node){
        Stack<Node> s = new Stack<>();
        Node trav = root;
        while (trav != null || !s.isEmpty()) {
            while (trav != null) {
                s.push(trav);
                trav = trav.leftchild;
            }
            if (!s.isEmpty()) {
                trav = s.pop();
                System.out.print(trav.data + " ");
                trav = trav.rightchild;
            }
        }
    }
    //Postorder without recursion method.
    public void postOrderNoRecMethod(){
        System.out.println("PostOrderNoRec");
        PostorderNoRec(root);
        System.out.println();
    }
    private void PostorderNoRec(Node node){
        Stack<Node> s = new Stack<>();
        Node trav=root;
        while (trav!=null ||!s.isEmpty()) {
            while (trav!=null) {
                s.push(trav);
                trav=trav.leftchild;
            }
            if (!s.isEmpty()) {
                trav=s.pop();
            }
            if (trav.rightchild!=null ){//&& error
                s.push(trav);
                trav=trav.rightchild;
            }
            else{
                System.out.println(trav.data);
                // error here
                trav=null;
            }
        }
    }
    //BFS-Breadth First Search-Level wise search.Time
complexity is O(n)
    public Node BreadthFirstSearch(int key) {
```

```
            Queue<Node> q= new LinkedList<>();
                //offer -- >push  poll -- > pop
            //step 1: push the root element on queue
            q.offer(root);
            while (!q.isEmpty()) {
               //step 2: then pop the element from queue
            Node trav=q.poll();
            //step 3: match with key element if match return
            if (key==trav.data) {
                return trav;
            }
            //setp 4: else push both left and right on queue
            if (trav.leftchild!=null) {
                q.offer(trav.leftchild);
            }
            if (trav.rightchild!=null) {
                q.offer(trav.rightchild);
            }
            System.out.print(trav.data+" ");
        }
            return null;
    }
            //DFS-Depth First Search-Time complexity is O(n)
            public Node DepthFirstSearch(int key) {
            Stack<Node> q= new Stack<>();
            //step 1: push the root element on stack
            q.push(root);
            while (!q.isEmpty()) {
            //step 2: then pop the element from stack
            Node trav=q.pop();
            //step 3: match with key element if match return
            if (key==trav.data) {
                return trav;
            }
            //setp 4: else push both left and right on stack
            //first psuh right then left or viseversa no problem.
            if (trav.rightchild!=null) {
                q.push(trav.rightchild);
            }
            if (trav.leftchild!=null) {
                q.push(trav.leftchild);
            }
            System.out.print(trav.data+ " ");
        }
            return null;
    }
```

```
    //Binary-search method.Time complexity is O(h)
    public Node BinarySearch(int key){
        Node trav=root;
        while (trav!=null) {
            if (key==trav.data) {
                return trav;
            }
            if (key<trav.data) {
                trav=trav.leftchild;
            }
            else{ //key>trav.data
                trav=trav.rightchild;
            }
            System.out.print(trav.data+" ");
        }
        return null;
    }
    //Binary-search with recursion homework.
    public Node[] BinarySearchWithParent(int key) {
        Node trav=root;
        Node parent=null;
        while (trav!=null) {
            if (key==trav.data) {
                return new Node[]{trav, parent};
            }
            parent=trav;
            if (key<trav.data) {
                trav=trav.leftchild;
            }
            else{ //key>trav.data
                trav=trav.rightchild;
            }
            System.out.print(trav.data+" ");
        }
        return new Node[]{null, null};
    }

    public void delete(int value){
        Node current, parent;
        // find the node to be deleted along with its parent
        Node[] arr = BinarySearchWithParent(value);
        current = arr[0];
        parent = arr[1];
        // if node is not found, throw the exception
        if (current == null){
            throw new RuntimeException("Node not found.");
```

```
            }
            // if node has left as well right child
            if (current.leftChild != null && current.rightChild != null){
                // find its successor with its parent
                parent = current;
                Node successor = current.rightChild;
                while (successor.leftChild != null){
                    parent = successor;
                    successor = successor.leftChild;
                }
                // overwrite data of node with successor data
                current.value = successor.value;
                // mark successor node to be deleted
                current = successor;
            }
            // if node has right child (or doesn't have left child)
            if (current.leftChild == null){
                if (current == root){
                    root = current.rightChild;
                } else if (current == parent.leftChild){
                    parent.leftChild = current.rightChild;
                } else {
                    parent.rightChild = current.rightChild;
                }
            }
            // if node has left child (or doesn't have right child)
            else {
                if (current == root){
                    root = current.leftChild;
                } else if (current == parent.leftChild){
                    parent.leftChild = current.leftChild;
                } else {
                    parent.rightChild = current.leftChild;
                }
            }
        }
}
class BinarySearchTreeMain {
public static void main(String[] args) {
        BinarySearchTree t = new BinarySearchTree();
        Scanner sc=new Scanner(System.in);
        t.add(50);
        t.add(30);
        t.add(10);
        t.add(90);
        t.add(100);
```

```
        t.add(40);
        t.add(70);
        t.add(80);
        t.add(60);
        t.add(20);
        t.preorderMethod();
        t.inorderMethod();
        t.PostorderMethod();


        t.preOrderNORecMethod();
        t.inOrderNoRecMethod();


    System.out.println("Enter Element to find: ");
    int val=sc.nextInt();
    BinarySearchTree.Node temp = t.BreadthFirstSearch(val);
        if (temp==null) {
            System.out.println("Element is not found");
        }
        else{
            System.out.println("Breadth-First-Search Found:
"+temp.getData());
        }
        temp=t.DepthFirstSearch(val);
        if (temp==null) {
            System.out.println("Element is not found");
        }
        else{
            System.out.println("Depth-First-Search Found:
"+temp.getData());
        }
        temp=t.BinarySearch(val);
        if (temp==null) {
            System.out.println("Element is not found");
        }
        else{
            System.out.println("Binary-Search Found:
"+temp.getData());
        }
    BinarySearchTree.Node [] arr=t.BinarySearchWithParent(val);
        if (arr[0]==null) {
            System.out.println("Element is not found");
        }
        else if (arr[1]==null) { //root node found
```

```
            System.out.println("Binary-Search Found:
"+arr[0].getData()+" With parent :"+arr[1]);
        }
        else{ //node found with parent
            System.out.println("Binary-Search Found:
"+arr[0].getData()+" With Parent :"+arr[1].getData());
        }
    System.out.println("Height of a tree: "+t.height());
    t.deleteAll();
    t.inorderMethod();
    System.out.println("Height of a tree: "+t.height());
   }
}
```

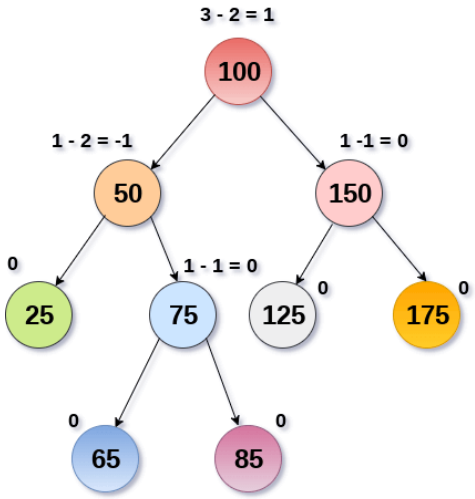
AVL tree

- It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.
- We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the ***difference between the height of the left subtree and the height of the right subtree***. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.
- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Why AVL Tree?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if

the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Rotations

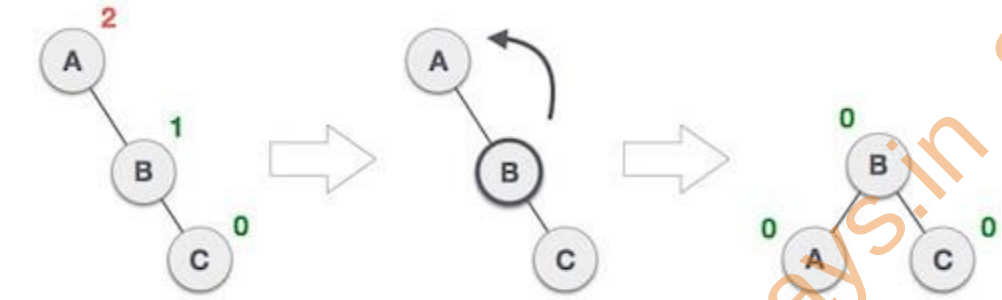
We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.
The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation
is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2

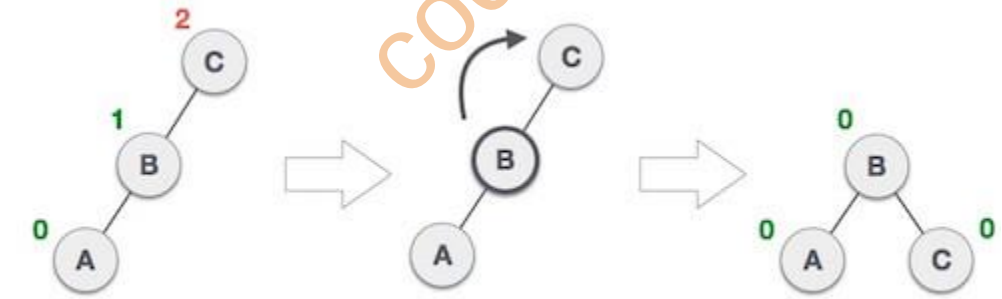


Right unbalanced tree Left Rotation Balanced

In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation
is clockwise rotation, which is applied on the edge below a node having balance factor 2.



Left unbalanced Tree Right Rotation Balanced Tree

In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation

= LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

Red-Black Tree

- **The red-Black tree** is the binary search tree. The prerequisite of the Red-Black tree is that we should know about the binary search tree. In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node. As we know that the time complexity of binary search in the average case is $\log_2 n$, the best case is O(1), and the worst case is O(n).
- When any operation is performed on the tree, we want our tree to be balanced so that all the operations like searching, insertion, deletion, etc., take less time, and all these operations will have the time complexity of $\boldsymbol{\log_2 n}$.
- ***The red-black tree*** is a self-balancing binary search tree. AVL tree is also a height balancing binary search tree then **why do we require a Red-Black tree**. In the AVL tree, we do not know how many rotations would be required to balance the tree, but in the Red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents either the red or black color of a node to ensure the balancing of the tree.

Splay tree

- The splay tree data structure is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations. Here, ***splaying*** means the recently accessed node. It is a ***self-balancing*** binary search tree having no explicit balance condition like **AVL** tree.
- It might be a possibility that height of the splay tree is not balanced, i.e., height of both left and right subtrees may differ, but the operations in splay tree takes order of **logN** time where **n** is the number of nodes.
- Splay tree is a balanced tree but it cannot be considered as a height balanced tree because after each operation, rotation is performed which leads to a balanced tree.

freelance_Project available to buy contact on 8007592194

SR.NO	Project NAME	Technology
1	E-Learning HUB	React+Springboot+MySql
2	PG MATES	React+Springboot+MySql
3	Tour and Travel	React+Springboot+MySql
4	Marriage Hall booking	React+Springboot+MySql
5	Bus ticket booking Mini Project	React+Springboot+MySql
6	Quizz App /Exam Portal Mini Project	Springboot,MySql,JSP,Html
7	Event Management System	React+Springboot+MySql
8	Hotel Mangement System	React+Springboot+MySql
9	Agriculture Web Project	React+Springboot+MySql
10	AirLine Reservation System	React+Springboot+MySql
11	E-Commerce Web Project	React+Springboot+MySql
12	Sport Ground Booking	React+Springboot+MySql
13	CharityDonation web project	React+Springboot+MySql
14	Hospital Management Project	React+Springboot+MySql
15	Online voting System Mini project	Springboot,MySql,JSP,Html
16	E-Commerce shop mini project	Springboot,MySql,JSP,Html
17	Job Portal web project	React+Springboot+MySql
18	Insurance policy Portal	React+Springboot+MySql
19	Transpotation Services portal	React+Springboot+MySql
20	E-RTO Driving licence portal	React+Springboot+MySql
21	doctor Appointment Portal	React+Springboot+MySql
22	Online food delivery Project	React+Springboot+MySql
23	Muncipal Corporation Management	React+Springboot+MySql
24	E-College Portal Project	React+Springboot+MySql
25	Gym Management	React+Springboot+MySql
X 26	Bike Booking System Portal	React+Springboot+MySql
27	Food Waste Management Portal	React+Springboot+MySql
28	Online Pizza delivery Portal	React+Springboot+MySql
29	Fruite Delivery portal	React+Springboot+MySql
30	HomeRental Booking Project	React+Springboot+MySql
31	FarmerMarketplace	React+Springboot+MySql

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/cceesept2023

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project