DATA STRUCTURE & ALGORITHMS (DSA)

DS - 63000 Saurabh

DATE 27 Ap

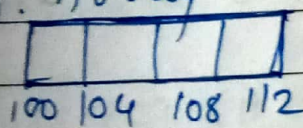
~~class Employee~~

⊙ Linear Data structure → (sequential)
1) Array (contigous)
2) structure & union
3) Linked list
4) stack
5) queue
6) class (oops)

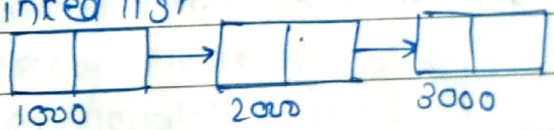
⊙ Advance DS → Data elements stored arranged into Non-linear / Hierarchical manner
∴ can be accessed Non-linearly
1) Binary Heap
2) Tree
3) Graph
4) HASH TABLE (Associative Manner → (key-value) pair)

⊙ Contiguous vs Non-Contiguous

eg. Array



eg. Linked list



⊙ Search → to search / to find an element from collection / list
↳ 1) linear search
2) Binary search

⊙ Algorithm → 1) Iterative (Non-Recursive) Approach
2) Non-Iterative

① Asympotic Notations → Asympotic lower Bound

1) Big Omega (Ω): Best case Time complexity
↳ If Algo takes min. Amt. of time to Run to completion

2) Big Oh (O): Worst case → Asymptotic Upper Bound (Run time of Algo can not be more than its Asymptotic upper Bound)
↳ Max. Amt. of time to Run to completion

3) Big Theta (Θ): Avarage Case: Asymp. Tight Bound

① Linear Search: Best case (Ω) → Ω(1)
Worst case (O) → O(n)
Avarage Case (Θ) → Θ(n)

(*) Iterative Approach for Linear Search (Non-Recursive)

```
① public class searchingmain {
    pub stat. void display (int [] arr) {
        println ("Array elements:");
    for (int i=0; i< Arr.length ; i++) {
        printf ("%4d", arr[i]);
    }
    println();
}
——— main (———) {
int Arr[] = {40, 80, 90, 50, 10, 20, 30, 60, 70, 100};

display (Arr);

println ("Enter key");
int key = sc.nextInt();
```

```
if ( linearSearch (Arr, key) )
        println ( key + " Found " + comparisons);
else
        println ( key + " NOT Found" );
} }
public static int comparisons = 0;

public static boolean linearSearch (int [] Arr, int key)
        for (int i = 0; i < Arr.length ; i++)
        {   comparisons++;
            if (key == Arr[i]) // if key matches
                return true;
        }
        return false ; // if key not found
}
```

to check

⊙ RECURSIVE LINEAR SEARCH

```
public static boolean reclinearSearch
                    (int [] Arr , int key,
                                    int index)
{
Base cond^n: // if ( index == Arr.length)
                    // if key do not matches
                Return false ;
            if ( key == Arr[i])
                return true;
            comparisons++;
        return (reclinearchSearch (Arr, key, index+1)
                // Recursive Func^n Call
}
```

```
main ( ——— )
{
    int [] Arr = {40, 80, 90, 50, 10, 20, - - - };
    int Key = sc.nextInt();

    if (recLinearSearch (Arr, Key, 0) // Initialization
        Println (Key+ " Found " + comparisons) ;
    else
        Println (Key+ " Not Found " + comparisons);
}
```

Lev 1 → Lev 2 → Lev 3 → Lev 4

① Recursive Algorithm Takes Extra space.

② IF NO. is Repeated → only First ~~Occ~~ occurance considered as Match.

③ **BINARY SEARCH ALGORITHM**

↳ Also called Logarithmic search
↳ Half Interval search
④ ↳ Divide & Conquer Approach
⑤ ↳ Array Must be Sorted (Default → Ascending)

arr →	10	20	30	40	50	60	70	80	90	100
index →	0	1	2	3	4	5	6	7	8	9
	Left				(mid)					Right

↳ Calculate Mid Position = $\left(\frac{\text{left} + \text{Right}}{2}\right)$

mid = (0+9)/2 = 4

↳ Implicit Division into logical two subArray → 10 20 30 40
→ 60 70 80 90 100

For Left Sub Array → 10 (0 Left) 20 (1) 30 (2) 40 (3 Right (mid-1))

Left remains same
Right will be (mid-1)

For Right SubArray → 60 70 80 90 100
5 6 7 8 9
(mid+1) Right

if key = 50 → key found at 1st iteration
(mid)

Best case → if key found in very 1st iteration
in only 1 comparisons.
↳ for any input size, no. of comparisons = 1

10 20 30 40 50 60 70 80 90 100
left mid Right

left SubArray : 10 20 30 40
left (mid-1)

Right SubArray : 60 70 80 90 100
(mid+1) Right

① if key = 50 → 1st iteration key found
Bcaz (mid = 50 = key)
∴ Best case : $\Omega(1)$

② if key = 100
50
mid
10 20 30 40 60 70 80 90 100
(mid-1) (mid+1)

if key == mid → NO
key < mid → left sub Array
key > mid → Right sub Array
100 > 50 → ∴ search key in Right sub Array

left mid Right

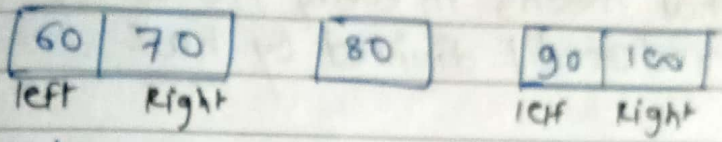
Right Sub Array : 60 70 80 90 100

(5) (6) (7) (8) (9)

↳ Find new mid = $\frac{(5+9)}{2} = \frac{14}{2} = 7$

new left sub Array : 60 70

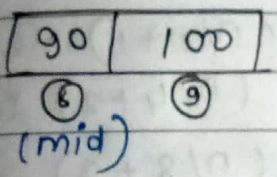
new Right Sub Array : 90 100



if key == mid (is 100 == 80) → NO

then search key in left sub Array / Right sub Array

100 > 80 → search in Right SubArray

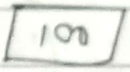


mid = $\frac{(8+9)}{2} = \frac{17}{2} = 8$

No Left Sub Array

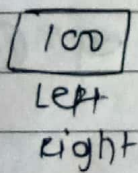
Right sub Array = 100

Righ Sub Array



is 100 == 90 → NO

key mid



→ mid = $\frac{(9+9)}{2} = 9$

if mid == key (100 == 100) → True.

① Worst case : if key NOT found or key found at Leaf position e.g. 100, 125, 95

② Avg. case : if key found at Non-leaf condition.
e.g. 80, 70, 90

③ Best case : if key found at Root position
e.g. 50

⊙ In this Algo, in every iteration 1 comparison happens & Array gets divided into SubArray's & next iteration will search key either into left SubArray or Right SubArray

⊙ Array divided logically into 2 sub Arrays. that means in every iteration search space is reduced / divided by half.

⊙ eg. Size of Array = 1000
iteration 1 : input size of Array = 1000 = n
After 1st iteration : (n/2 + 1) Comparison

1000 = n
500 = n/2 → After 1st iteration : (n/2 + 1)
250 = n/4 → After 2nd iteration : (n/4 + 2) comparison
125 = n/8 (n/8 + 3)
62 = n/16 (n/16 + 4)

$(n/2^m) + m$

$$T(n) = (n/2^k) + k \longrightarrow \text{eqn } (1)$$

if $n = 2^k$

$\log n = \log 2^k$

$\log n = k \log 2$

$\log n = k$ ∴ $k = \log n$

$T(n) = n/2^k + k$

$T(n) = n/2^k \cdot 2^k/2^k + \log n$

$$T(n) = 1 + \log n$$ → By Assumption 1 Additive constant can be neglected

worst case → $T(n) = \log n$

	Best case	Worst case	Avg. case
Linear Search	$\Omega(1)$	$O(n)$	$\Theta(n)$
Binary search	$\Omega(1)$	$O(\log n)$	$\Theta(\log n)$

Binary Search Program

```
main( )
{  int [] arr =

    if ( BinarySearch (arr, key) )
        println (key + " Found") ;

}

public static Boolean BinarySearch (int [] arr, int key)
  {
      int left = 0 ;
      int right = arr.length - 1 ;
step → (2) // calculate mid position  while (left <= right) {
      int mid = (left + right) / 2 ;

step → (3) // compare value of key with ele which at mid.
          comparisons ++ ;
      if (key == arr[mid])
            return true .
  // if key do not match with mid pos. element
      we will search key either left/ right subArray

      if (key < arr[mid])
            right = mid - 1 ; // goto seach key
                                  into left subArray
      else
          left = mid + 1 ; // go into right subArray.
  }
```

step (4) → // Repeat step 2 & step 3 till either key is found or more till subarray is

return false;

}

⊙ To decide efficiency of Algorithm
↳ Compare their Avg. case
↳ $\theta(n) > \theta(\log n)$

∴ Binary search → Efficient than → Linear search
Avg. case Time complexity $O(\log n)$ $O(n)$

⊙ Recursive Binary Search Function

main

```
{
    if (recBinarySearch(arr, key, 0, arr.length-1))
        println("found");

}
public static boolean recBinarySearch(int[] arr,
                                      int key, int left, int right
{
    if (left > right)
        return false;
    int mid = (left + right)/2;    comparison++;
    if (key == arr[mid])
        return true;
```

```
if (key < arr[mid])
return (recBinarySearch (arr, key, left, mid-1));
else
return (recBinarysearch (arr, key, mid+1, right));
}
```

SELECTION SORT

↳ element at selected position will be compared to next all elements, and ~~to~~ swapped position.

n = 6

Iteration 1 : ① 30 20 60 50 10 40
0 1 2 3 4 5
sel pos pos++ 30>20

↓ Smallest element will be settled to its fix position i.e, 1st position

② 20 30 60 50 10 40
sel pos pos++ 20<60

③ 20 30 60 50 10 40
sel pos pos++ 20<50

④ 20 30 60 50 10 40
sel pos pos++ 20>10 (swap)

pos++ till ~~less~~ till array length

⑤ 10 30 60 50 20 40
sel pos pos++ 10<40

(n-1) → comparisons

Iteration 2 : ① 10 30 60 50 20 40 30<60
sel pos pos

② 10 30 60 50 20 40 30<50
sel pos pos

③ 10 30 60 50 20 40 30>20 (swap)
sel pos pos

④ : 10 20 60 50 30 40 20<40
sel pos pos

(n-2) → comparisons

selpos pos

Iteration 3: ① 10 20 60 50 30 40 — 60 > 50 (swap)

3rd smallest element got fix to 3rd position:

② 10 20 50 60 30 40 — 50 > 30 (swap) (selpos, pos)

③ 10 20 30 60 50 40 — 30 < 40 (selpo, pos)

Iteration 4: ① 10 20 30 60 50 40 — 60 > 50 (swap) (selpos, pos)

4th smallest element got fix:

∴ ② 10 20 30 50 60 40 — 50 > 40 (swap) (selpos, pos)

∴ 10 20 30 40 60 50

Iterations 5: ① 10 20 30 40 60 50 — 60 > 50 (swap) (selpos, pos)

∴ 10 20 30 40 50 60

(Array Got Sorted) *

∴ No. of Iterations = (Array length - 1) [~~of~~ (n-1) Iterations]

∴ Total No. of Comparisons : $(n-1)+(n-2)+(n-3)+\cdots+1$

$= \frac{n*(n-1)}{2}$

$T(n) = O(n(n-1)/2)$

$= O((n^2-n)/2)$ div. const. can be neglected

$= O(n^2-n)$

Assumption 2 : If running time of an Algo is having Polynomial then in its time complexity only leading term will be considered.

e.g. $(n^3+n^2+5) \rightarrow O(n^3)$

$\therefore T(n) = O(n^2-n) = O(n^2)$

Selection sort : Best case $\Omega(n^2)$ Worst case $O(n^2)$ Avg. case $\Theta(n^2)$

Even if Array sorted / unsorted } same no. of Iterations & comparisons
$\therefore$ Best = worst = Avg.

Features of Sorting Algorithm :

1) Inplace ⇒ if sorting algo do not take extra space (space other than actual data element & const. space) to sort data elements in a collection / list of elements

2) Not Adaptive ⇒ if a sorting algorithm works efficiently for already sorted input sequence then it is referred as an Adaptive

3) Not Stable ⇒ if a sorting Algorithm relative order of two elements having same key value remains same even after sorting then such sorting algorithm is referred as Stable

main (—)

```
{ int[]arr = {30, 20, 60, 50, 10, 40};
  print(" Before sorting :");
   displayArrayElements(arr);
}
Public static void displayArrayElements (int[]arr)
 {
    for (int i=0; i<arr.length; i++)
     {
       printf("%d ", arr[i]);
     }
    println();
 }
  Public static void selectionSort (int[] arr)
 { int iterations, comparisons;
   for (int sel_pos = 0 ; sel_pos < Arr.length-1 ; sel_pos
    { iterations ++;
    for (int POS = sel_pos +1 ; POS < Arr.length;
                                          POS++)
    comparisons ++
     {
       if (arr[selpos] > arr[POS])
          swapArrayElements (arr, sel_pos, POS);
     }
  }
}
```

```
public static void swapArrayElements (int arr[], int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

29/4/2022

Bubble Sort

Best case : $\Omega(n^2)$	Inplace → yes
Worst case : $O(n^2)$	Adaptive → No
Avg. case : $\Theta(n^2)$	Stable → yes

Total no. comp. : $(n-1) + (n-2) + (n-3) + \cdots + 1$

$= \frac{n(n-1)}{2}$

$T(n) = O(n(n-1)/2)$

$T(n) = O(n^2 - n/2)$

$T(n) = O(n^2 - n)$

$T(n) = O(n^2)$

```
public static void ~~main~~ BubbleSort (int[] arr) {
    int iterations = 0; boolean flag = true;
    for (int it = 0; it < arr.length-1; it++)
    {   iterations++; flag = false;
        for (int pos = 0; pos < arr.length-1-it; pos++) {
comparisons++;
            if (arr[pos] > arr[pos+1]) {
                swap (arr, pos, pos+1); flg = true;
            }
        }
}
```

Best case : If Array is already sorted (10 20 30 40 50) then No need of swapping

⊙ By design Bubble Sort is Not Adaptive, but at implementation level we can make it an adaptive by adding logic of flag.

Efficient Bubble Sort: Best case: $\Omega(n)$
worst case: $O(n^2)$
Avg case: $\Theta(n^2)$
Inplace: yes
Adaptive: yes
stable: yes

⊙ Avg. case of Both Bubble sort & selection sort is same
∴ Efficiency is same for Both

Insertion Sort

```
for (i=1 ; i< size ; i++) { // For loop for Iterations
    int key = arr[i];
    int j = i-1;
    // if index is valid && compare value of key with
        an element at that index
    while ( j>=0  && key < arr[j]) {
        // Shift element towards its right hand side by
            1 pos
        arr[j+1] = arr[j];
        j--;
    }
    // insert key at its appropriate pos
        arr[j+1] = key;
```

Best case : $\Omega(n)$ Adaptive → yes
Worst case : $O(n^2)$ Stable → yes
Avg. case : $\Theta(n^2)$ Inplace → yes

- Insertion Sort is By design efficient

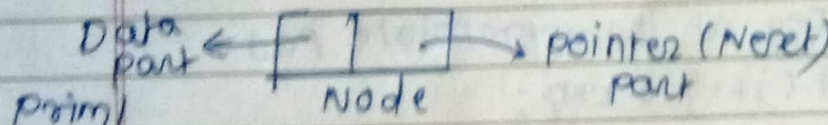
⊙ LINKED LIST

1) Singly LL : each node contains link to its next node only (No. of link = 1)
   - Linear
   - Circular

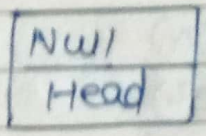
2) Doubly LL : Each node contains link to its next node & prev. node (No. of links = 2)
   - Linear
   - Circular

① Singly Linear Linked List

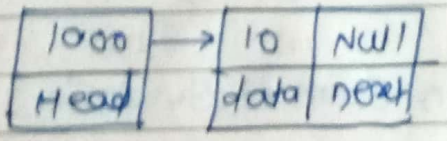
↳ Head Always Contains address of 1st element if list is Not empty



Primitive / Non-primitive Data

1) 

list is Empty

2) 

only 1 node

~~class List~~ List HAS A Node → HAS-A Relationship (Composition)

Next page

```
Public class SingleLinkedListMain {
    ——, ——— void main (———~)
    { // Create empty linked list
      LinkedList ll = new LinkedList();

      ll.addNodeAtLastPosition(10);
      ll.addNodeAtLastPosition(20);
      ———, ——— (30);
```

(Composition)

```
Class LinkedList {

    Static Class Node {
        private int data; // data of int type
        private Node Nexet; // Reference to nexet node
    (8 Bytes)
    }

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

private Node head; // Reference to first node in list
private int nodesCount;
public LinkedList()
{   this.head = null;   this.nodesCount = 0;
}
Void addNodeAtLastPosition (int data) {
    // Step1: Create New Node Runtime
Node newNode = new Node (data);

// Step2: check if List is Empty then attach
    newly created node to Head
if (isListEmpty())
{
        head = newNode;  nodesCount++;
} else      // Step3: if list is Not Empty start traversal
{                                  From First Node
                                   till last node
Node trav = head;
While (trav.next != Null) {
        trav = trav.next; // Move trav to nexet
}                                          Node
trav.next = newNode; // Attach newNode to last
nodesCount++;
}
```

```
void displayList() {  if (isListEmpty())
                      {throw new RuntimeException("List is Empty");
else {                }
Node trav = head;     while(trav != null) {⊙
}
```

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

```
printf(".1.d->", trav.data);
trav = trav.next;
⊙
println("null");
}
```

⊙ Algorithm to Add Node At last position

1) Create a New Node At ~~Runty~~ RunTime (new ())
2) check if List is Empty or NOT
   - ↳ if Empty → yes → Head will be NewNode
   - ↳ if List is Not Empty
     - ↳ Then traverse from First Node to last Node

*) We can Add As many as we want no. of nodes into SLLL at last position in O(n) Time

Best case : Ω(1) → if List is Empty
Worst case : O(n)
Avg. Case : Θ(n)

```
Public class SinglelinkedListMain {
    —"— void main (——)
{   // Create an empty LL
    LinkedList l1 = new LinkedList();
l1.addNodeAtLastPosition(data:10); // (20)(30)(40)(
```

// if LL is Empty it ~~sho~~ will not be displayed handle exception.

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

[Make Before Break] → Always create New Link First & then only break old Links.

Add Node At First Position

Step 1: Create New Node

Step 2: Check if list is Empty
if yes: Head = newNode

Step 3: If List is NOT Empty; Attach newly created node to Head
Store address of current first node into the next part of newly created node
newNode.next = head;

```
Void addNodeAtFirstPosition (int data)
{   // ① → create New Node
    Node newNode = new Node (data);
// ② → check if list Empty
    if (isListEmpty())
    {       head = newNode;  // if Empty then Attach newly
            nodescount ++;      created node to head
    }
    else
    {  // ③ → if list NOT Empty
       newNode.next = head;  // Store on address of
                                current first node in
                                Next part of newly
            head = newNode;     created node
    nodescount++;} // Attach newely created
    }       Node to head
```

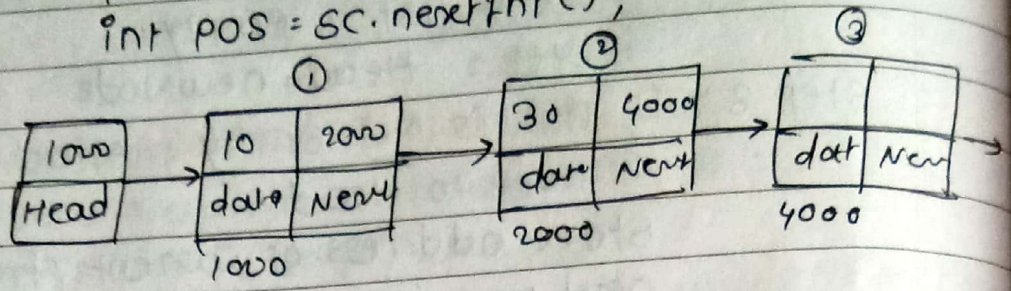
Best case : Ω(1)
Worst case : O(1)
Avg. Case : Θ(1)

⊙ Add Node At particular position In Between List

Step1: Accept the position At which node to be added

```
int pos = sc.nextInt();
```

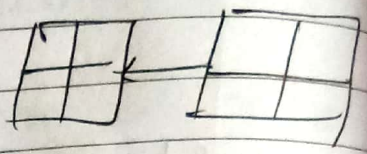
Step 2:



validate position

```
if (pos >= 1 && pos <= count + 1)
```

count_nodes → By traversing



```
void displayList () {
    if (IsListEmpty()) {
        ~~ret~~ throw new RunTimeException
                    (" List is Empty! ");
    }
    else {
        // traverse LL from First Node
        Node trav = head; // trav is ref. of Node class
                          // traverse till last node
                             including it
        while (trav != Null)
        {
            printf(" %d →", trav.data); // display data part
            trav=trav.next; // move trav to next
        }
        println (" Null "); // this will printed At last of LL
        println (" no. of nodes in LL " + Count_Nodes() );
                                          ↳ define this Method in class LinkedList
        println ("no. of nodes" + this.getNodes(count));
    }
```

class LinkedList() {

Static class Node & its Logic

int count_Nodes() { Time : O(n)
    int cnt = 0;
    if (!isListEmpty()) { // if list is Not Empty
        Node trav = head;
        while (trav != NULL) {
            cnt++;
            trav = trav.next; // move to next
        }
    }
    return cnt;
}

// count_node() method take time O(n) which is not efficient
∴ other way : List HAS-A count of nodes
∴ create one more mem. of LinkedList class
    private int nodesCount;
initialize it in constructor LinkedList()
    this.nodesCount = 0;
And maintain the count everywhere we are adding newNode.

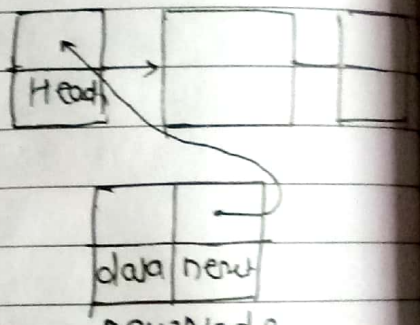
int getNodesCount() { Time : O(1)
    return (this.nodesCount);
}

Step 1 : validate position.
    If (pos >= 1 && pos <= ll.getNodesCount() + 1)
    ↳ true then position is valid

```
main (---)
{
    while (true) {
        // Step 1: Accept position from user
        int pos = sc.nextInt();

        // step 2: validate position
        if (pos >= 1 && pos <= ll.getNodesCount() + 1)
            Break; // while will break

        system.out.println ("invalid position");
    }
}

class LinkedList {
    void addNodeAtSpecificPosition (int pos, int data)
    {
        if (pos == 1)
            addNodeAtFirstposition (data);
        elseif (pos == getNodesCount() + 1)
            addNodeAtLastposition (data);
        else {
            // create New node First
            Node newNode = new Node(data);
            i = 1;
            // start traversal from 1st Node
            trav = head;
            // traverse till (pos-1)th Node
            while (i < pos-1) {
                i++;
            } trav = trav.next;
            // Attach cur (pos)th node to next part of newly
                created node.
            newNode.next = trav.next;
```

```
// attach newly created node into next part of
   (pos-1)th node
      trav.next = newNode;
      nodescount++;
   }
}
```

```
class LL {                    Node head;
{
    class Node {
        String data;
        Node next;
    // constructor:
        Node (String data) {      // while creating new node
            this.data = data;     new part will
            this.next = null;     By default null
        }
    }
public void addFirst (String data) {
        if (head == null)
① create New Node
        Node newNode = new Node (data);
② check if LL is empty
        if (head == null) {
            head = newNode;
            return;
        }
③ If Head is not null
        newNode.next = head;
        head = newNode;
}
```




```
public void addlast (String data)
{       Node newNode = new Node (data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node currNode = head;
        while (currNode.next != null) {
            currNode = currNode.next;
        }
        currNode.next = newNode;
}
```

TREE

NULL → C++
null → Java

DATE

```
public static void main(String[] args)
{
    LL list = new LL(); // object of class LL
    list.addFirst("a");
    list.addFirst("is");
    list.printList();
    list.addLast("list");
    list.printList();
}
```

```
public void printList() {
    if (head==null) { System.out.println("Empty"); return; }
    Node currNode = head;
    while (currNode.next != null)
    {
        System.out.println(currNode.data + "→");
        currNode = currNode.next;
    }
    System.out.println("null");
}
```

Search And Delete Node

```
~~boolean SearchAndDelete (int Data)~~
    boolean SearchNode (int data, Node [] temp)
    {
      for (Node trav = head; trav != null;
                                trav = trav.next)
      {
        temp[0] = trav;
        if (data == trav.data) { temp[1] = trav;
            return true;
        }
      }
      temp[0] = null; temp[1] = null;
      return false;
    }
}
```

/* SearchNode() Function:
On success this function returns true and it also returns an addr of node which is to be deleted in temp[1] as well as an address of its previous node in temp[0] and on failure this function returns false and in temp ref it will return null
*/

```
boolean SearchAndDelete (int data)
    {
      Node [] temp = {null, null};

    // search a node whose data matches with "data"

    if (!searchNode (data, temp))
        return false;
    }
```

// if Node is found
if (temp[0] == null) // if Node found At 1st position
print("Node having data part : "+data+
" is found : temp(1).data :
" + temp(1).data);
else
print (" temp(0).data :" + temp[0].data
+ " temp(1).data :" + temp(1).data);
~~return false~~ ;
// if node which is to be delete is At 1st position.
if (temp(0) == null)
head = head.next ; // Attach our second
node to head
else
temp(0).next = temp(1).next ;
// Attach next node of
a node which to be
deleted to its prev node.
nodescount--;
return true;
}

priority queue can be implemented using SearchAndDelete() Function.

Q. Display linked list in a Reverse order without modifying original linked list & using extra space.

Soln 1: Allocate memory dynamically for an Array of integers of size count
Traverse linked list & while traversing keep on adding data part of each node into an array
Display Array in Reverse from Right to Left

Soln 2: maintain Stack of integers
Traverse linked list & while traversing it keep on pushing data part of each node onto stack
Pop element from stack one by one & Display them

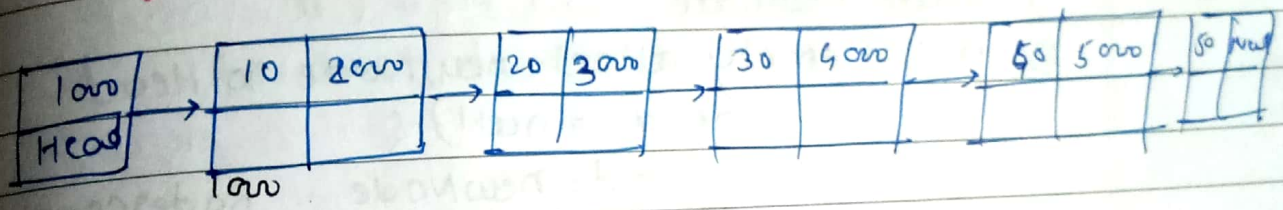
Sol 3: In Recursion internally an OS Maintain stack

// wrapper Func^n From inside actual rec Func^n get called.

```
void displayListInReverseOrder() {
    if(IsListIsEmpty()) {
        throw new RunTimeExcepHon("List is Empty");
    }
    else {
        print("List in a Reverse order");
    displayListInReverseOrder(head); // InitializaHon
    }
}
```

```
void displayListInReverseOrder ( Node trav)
{
    if (trav == null)
        return ;
    displayListInReverseOrder (trav.next);
    System.out.printf ("%d", trav.data);
}
```

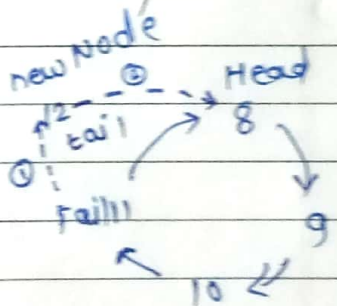
2) Reverse Linked list By traversing only once without using extra space.




```
Node t1 = head;
Node t2 = t1.next;
     t1.next = null;
Node t3 = t2.next;
t2.next = t1
t1 = t2 ; // move t1 towards next node
t2 = t3; // move t2 towards next node.
while (t2 != null)

head = t1 ; // Attach current last node to head
```

~~Display List~~

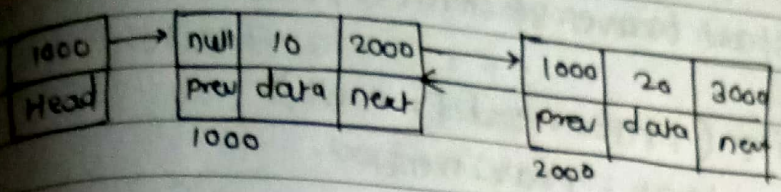
```
public class CircularLinkedList {
    private Node head;
    private Node tail;
    public CircularLinkedList (~~Node head, Node tail~~)
    { this.head = ~~head~~; null;
      this.tail = ~~tail~~; null;
    }
    public class Node {
            int val;
            Node next;
        public Node (int val)
        { this.val = val;
        }

public void insert (int val) {
   // create New Node first
  Node ~~new~~ Node = new Node (val);
     if (head == null)   // if List Empty
     { head = newNode;
       tail = newNode;
       return;
     }
  ① tail.next = ~~Node~~ newNode;
  ② newNode.next = head;
     ③  tail = ~~Node~~ newNode;
}

Public void Display ( ) {
     Node ~~newNode~~ = head;
       if (head != null) {
         do {
        print (node.val + "→");
          node = node.next;
        } while (node != head);
```



```
public void delete (int val)
{   Node trav/node = head;
    if (trav == null){
        return;
    }
    if (~~A~~ trav.val == val) {    // if val at hea
        head = head.next;
        tail.next = head;
    ~~}~~ return;
    }
    do {
        Node n = ~~nod~~ trav.next;
        if (n.val == val) {
            trav.next = n.next;
            break;
        }
        trav = trav.next;
    } while (trav != head);
}
```

DB-63000-Sawant
Date

Doubly linear linked list




```
class DLLL {
    static class Node {
        Private int data ;
        -"- Node prev ;
        -"- Node next ;

        Node (int data) {
            this.data = data.
            this . prev = null;
            this.next = null ;
            }
    }
    Private Node head ;
    Private int nodescount ;

    DLLL () {
        head = null ;
        nodescount = 0 ;
    }
void addNode At Lastposition (int data)
{       // create New Node
    Node newNode = new Node ( data) ;
// If list empty → attach newNode to head

    if ( isListEmpty ()) {
            head = newNode ;  nodescount ++ ;
    }
```

null		null
prev	data	next

newNode

```
else {  // IF List Not empty
        // Start traverse from first node to last

        while ( trav. next != null )
              trav = trav. next

        trav. next = newNode ;
         newNode. prev = trav ;
          nodescount ++ ;
    }
}


void displayList () {
      if ( isListEmpty () )
           throw ———— (" list Empty " );
  else {
      Node trav = head ;

       while ( trav != null )
          {
            printf (" %4 d ", trav. data) ;
              trav = trav. next ;
          }
        println () ;
    }
}
    // Reversal of List
  trav = temp ;
       while ( trav != null ) {
              temp = trav ;
            printf (" %4 d ", trav. data) ;
             trav = trav. prev ;
          }
          println () ;
}  }
```

```
Void addNodeAtFirstPosition (int data) {
        Node newNode = new Node (data);
if (isListEmpty ()) {
        head = newNode;
          nodesCount ++;
    }
else {
        newNode.next = head;
          head.prev = newNode;
            head = newNode;
              nodesCount ++;
        }
}

Void addNodeAtSpecificPosition (int pos, int data) {
        Node newNode = new Node (data);
        if (pos == 1)
            addFirstPosition (data);
        else if (pos == getNodesCount () + 1)
              addNodeAtLastPosition (data);
        else {
          int i = 1;
        Node trav = head;
          while (i < pos - 1) {
                  i++;
                trav = trav.next;
          }
          newNode.next = trav.next;
          newNode.prev = trav;
          trav.next.prev = newNode;
            trav.next = newNode;
```

4 May

⊙ Dynamic Stack (By using linked list) Doubly circular

stack → LIFO

[Push ⇒ addlast() O(1)
  Pop ⇒ deletelast() O(1)

OR

if (head == null) ⇒ list is empty ⇒ stack is empty

Peek ⇒ get data part of last node

addFirst()
deleteFirst()

Labwork : Implement Dynamic stack

Applications of stack:

1) Stack is used by an OS to control flow of execution of prog.
2) In Recursion internally an OS uses a stack
3) Undo & Redo functionalities of an OS implemented by using stack
4) Stack is used to implement advance data structure algo like DFS: DepthFirst Search
5) Stack used in Algo to convert infix expression into its postfix & prefix & for postfix expression evaluation

⊙ Stack For Expression Conversions & Evaluation

① Infix: a * b / c * d + e - f * g + h

↓

Postfix: ① ab* / c * d + e - f * g + h

② ab*c/ * d + e - f * g + h

③ ab*c/d * + e - f * g + h

④ ab*c/d*e + - f * g + h

⑤ ab*c/d*e+f - * g + h

⑥ ab*c/d*e+f-g * + h

⑦ ab*c/d*e+f-g*h + +

ab*c/d* +e -fg* +h

db*c/d*e + - fg* +h

ab*c/d*e+fg* - +h

ab*c/d*e+fg*-h +

(*) Algorithm to convert infix expression into equi. Postfix

1) Start scanning infix expression from left to Right

2)

```
if (cur element is an operand) {
        append into postfix expression
} else
  { if (cur element is an operator)
      if (stack is not empty && &
      priority (topmost element)
        >= priority (cur element )
while ( ! is stackempty () && &
        priority (topmost element)
        >= priority (cur element)
    {
      pop element from stack &
      append it to postfix expression
    }
  // push current element onto the stack
}
// step 3: repeat step 2 till end of infix expression
// step 4: pop All element's from stack one by
        one & append them to postfix
```

```
import java.util.Stack;

String infixToPostfix ( String infix)
// initially we need empty stack of an operators
    Stack<Character> S = new Stack<Character>();

    StringBuilder postfix = new StringBuilder();

// step1 : start scanning infix expression from left to Right
    For (int index=0; index < infix.length(); index++)
    {    // get element at cur_ele
        char cur_ele = infix.charAt(index);
    // if cur_ele is operand, Append it into postfix
        ~~postfix.append(cur_ele);~~
        if (isOperand(cur_ele))
            postfix.append(cur_ele);
        else { // if cur.ele is an operator
    // if stack is not empty & priority (topmost) >= priority
                                                    (cur.ele)
            While (!S.Empty() && priority(S.peek())
                                >= priority(cur_ele))
            {
                postfix.append(S.peek());
                S.pop();
            }
                S.push(cur_ele);
        }
    } // step4 : repeat-2 & -3 step till end
    // step5 : pop all remaining element from stack
        & append them to postfix expression.
    while (!S.empty()) {
            postfix.append(S.peek());
            S.pop();
    }
// convert StringBuilder class object to String objed.
return (postfix.toString());
```

```
static boolean isOperand (char ch)
{
    if ( ch >= 65 && ch <= 90) ||  (ch >= 97 && ch <= 122)
                                   || (ch >= 48 && ch <= 57))
    return true ;
}

public static int priority (char opr)
{
    switch (opr) { case '(' : return 0 ;
        case '+' :
        case '-' : return 1 ;

        case '*' :
        case '/' : return 2 ;

        default : return -1 ;
    }
}
```

capital

small

digits

```
public static void String InfixToPrefix ( String Infix)

    Stack <character> S = new ——— ();
    StringBuilder prefix = new ——— ();

for (int index = infix.length ()-1 ; index >=0 ; index --)
{
    char cur_ele = infix.charAt (index);

    if ( isOperand (cur_ele))
            prefix.append (cur_ele);
    else {
    while ( ! S.empty () &&
            priority (S.peek ()) > priority (cur_ele))
        {
            prefix.append (S.peek ());
            S.pop ();
        }
    S.push (cur_ele);
    }
}
    while ( ! S.empty ()) {
            prefix.append (S.peek ());
            S.pop ();
        }

return ( prefix.reverse ().toString ());
```

Evaluation for postfix

infix: 4 + 5 * 9 / 7 + 3 - 6

postfix: 4 + 59 * /7 + 3 - 6

4 + 59*7 / + 3 - 6

459*7 / + + 3 - 6

459*7/+3 + - 6

o/p: [459*7/+3+6-]

step 1: Start scanning postfix expression from left to right

step 2:

```
If (cur.ele is operand)
    Push it onto stack
else If ( Cur_ele is an operator)
  {  → pop two elements from stack in such a
       way that
     operand 2 = First popped element
     operand 1 = Second popped element
  → Perform Cur_ele (operator) operation
       on op1 & op2
     result = op1 (opr) op2
  → push back result onto stack
}
```

step 3: pop final Result from stack

4 5 9 * 7 / + 3 + 6 -

stack (operands)

9			
5	45	45	6
4	4	4	

op1 = 5
op2 = 9
result = 5 * 9
push = 45

op1 = 45
op2 = 7
result = 45 / 7
= 6 (push into stack)

op1 = 4
op2 = 6
result = 6+4 = 10 (push)

result = 10+3 13	6	13

result = 13 - 6 (push)

stack

7 → Expected o/p)

Queue

It is linear / basic data structure which is collection / list of logically related similar type of data elements in which elements can be added into it from one end (Rear end) and can be deleted from it at (Front end)

Element was inserted first can be deleted first ow manner or last in last ow manner.

∴ LILO / FIFO

4 Types of Queue :

1) Linear queue (FIFO)
2) circular queue (FIFO)
3) Priority queue (NO FIFO)
   ↳ Type of queue in which elements can be added into from Rear end randomly (i.e. without checking priority) where as element having highest priority deleted first
4) Double ended queue (deque)
   ↳ Elements can be added / deleted from both the sides

Operations performed in O(1) For Queue

1) Enqueue → insert / add from Rear end
2) Dequeue → delete / remove from Front end.

For Dequeue : 4 operations in O(1)

i) Push-back() ⇒ addLast()
ii) Push-Front() ⇒ addFirst()
iii) Pop-back() ⇒ deleteLast()
iv) Pop-Front() → deleteFirst()

} can be implemented using Doubly circular linked list.

⊙ 2 Types of Deque

↳ 1) Input restricted dequeue

↳ elements can be added → only from one end

elements can be deleted → from both ends

2) Output restricted dequeue

↳ elements can be added → from both ends

elements can be deleted ⇒ from only one end

5 May 2022

queue → is static as well as Dynamic

1. Stack implementation (using Array) of queue
2. Dynamic implementation of queue (using linked list)

Static Implementation of Queue (Using Array)

int arr[5], int Front, int rear;

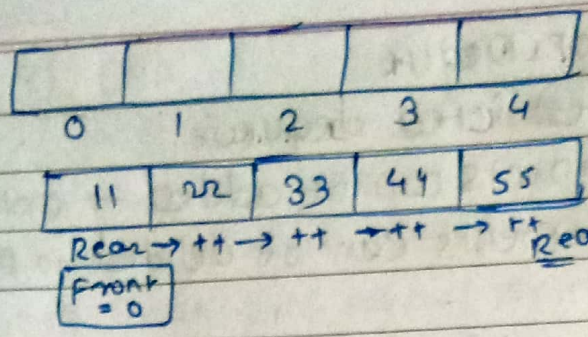
physically
↳ Array

logically
↳ queue

```
class LinearQueue {
    int[] arr;
    int Front;
    int rear;
    LinearQueue (int size) {
        arr = new int[size];
        Front = -1;
        rear = -1;
    }
```

// Enqueue → Add From Rear End

1) → If Queue is Not Full then insert / add element
2) → Increment val. of Rear (If Not Full)
3) → Insert From Rear
4) → If val. of Front = -1 make it as 0



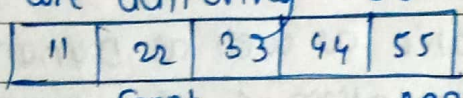
Queue Full : rear = size - 1

Queue Empty : rear = = -1 OR Front > rear

// Dequeue → Delete / Remove from front end

1) check if queue is empty
2) IF NOT empty : increment val. of Front by 1
   [by means of increanting front by 1 we are achieving deletion logically]

e.g.



Queue → from front to end

When Front = = Size OR (Front > Rear) → Empty Queue

```
(6) void enqueue (int ele)
{   // Step 1 : Check if queue is Not Full
        if ( isQueueFull() )
    {
        Throw new RunTimeException ("Queue is Full");

    }
    else { // if queue is Not Full then insert
           // Step 2 : Increment Rear value
        rear ++;
        // Step 3 : Insert an element From Rear
        arr [rear] = ele;

        if (Front = = -1)
                Front = 0 ;

    }
}
```

```
boolean isQueueFull() {
    return (rear == arr.length - 1);
}
boolean isQueueEmpty() {
    return (rear == -1 || front > rear);
}

int getFront() { // to get element at arr[front]
  if (isQueueEmpty())
  {   throw new RuntimeException("Queue is empty");
  }
  else {
      return (arr[front]);
  }

void dequeue() { // to delete from end

  if (isQueueEmpty())
  { throw new RuntimeException("Queue is empty");
  }
  else {
        // if queue is not empty, inc. front by 1
  front++;
  }
```

Limitation of linear queue

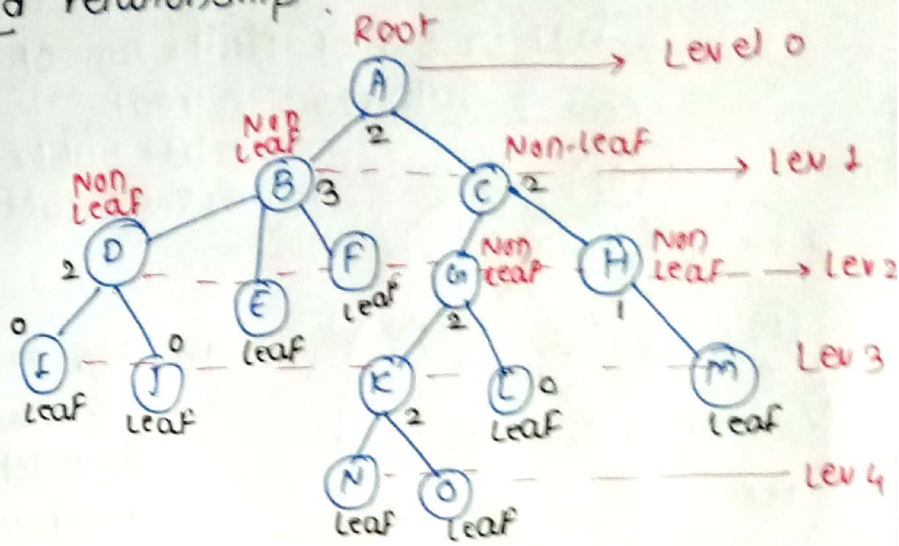
↳ vacant places can not be reutilized

∴ to overcome limitation circular queue is designed.

```
class circularQueue {

⊙ boolean is_QueueFull () {
        return ( Front == (rear + 1) %. arr.length);
  }

⊙ boolean isQueueEmpty () {
     return ( rear == -1 && Front == rear);
  }

⊙ Void enqueue (int ele)
  {
      if (isQueueFull ()) {
          throw RunTimeException ("Queue is Full");
      }
      else {
          rear = (rear+1) %. arr.length ;

          arr [rear] = ele ;
            if (Front == -1)
                Front = 0;
      }
  }

⊙ Void dequeue () {
        if ( isQueueEmpty ())
    {  throw new RunTimeException ("Queue empty");
    }
    else {
          if ( Front == rear) {
              Front = rear - 1 ;
          }
          else {  Front = (Front+1) %. arr.length;
          }
  }     }
```

Advance Data Structure → TREE

⊙ Tree is Non Linear / Advance data structure which is collection of finite no. of logically related similar type of data elements in which there is. First element → Root element and remaining all elements are connected to root in hierarchical manner. Follows parent-child relationship.



⊙ Ancestors:
↳ All nodes which are in path from root node to that node.
e.g. For I → D, B, A are Ancestors

⊙ Descendents:
All nodes Ascending From it
e.g For K → N, O descendents

⊙ Sibilings: child nodes of same parents

⊙ Degree of Node: No. of child nodes having that node.
eg. B → D, E, F ∴ B (3)
parent child

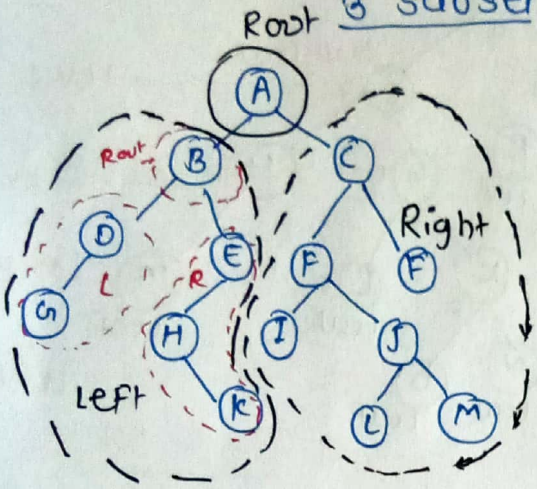
⊙ Degree of Tree: max. degree of any node in a given tree
e.g. B has degree 3 ∴ Deg. of Tree = 3

⊙ Level of Node: (level of Parent + 1)

⊙ Level of Tree / Depth of Tree = max. level of Any node in a tree.
∴ Depth of Tree = 4

① Data Structure is Dynamic → can grow upto any level & any Node can have any no. of child nodes operations on it becomes inefficient, so restrictions can be applied on it to achieve efficiency and hence there are different types of Tree

① Binary Tree → each node can have Max. 2 child nodes
or each node can have degree = 0 / 1 / 2
↳ It is set of finite no. of elements having 3 subsets : 1) root
2) Left Subtree (may be empty)
3) Right Subtree (May be empty)



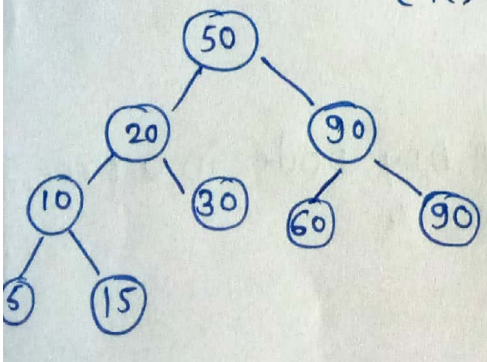
Further Restrictions can be applied on Binary tree to achieve Add / Delete / search operations efficiency expected in O(log n) time
↳ Binary Search Tree can be Formed

② Binary Search Tree

	Addition	Deletion	searching
Array →	O(n)	O(n)	O(log n)
LinkedList →	O(1)	O(1)	O(n)
Binary Search Tree →	O(log n)	O(log n)	O(log n)

disadvg. disadv.

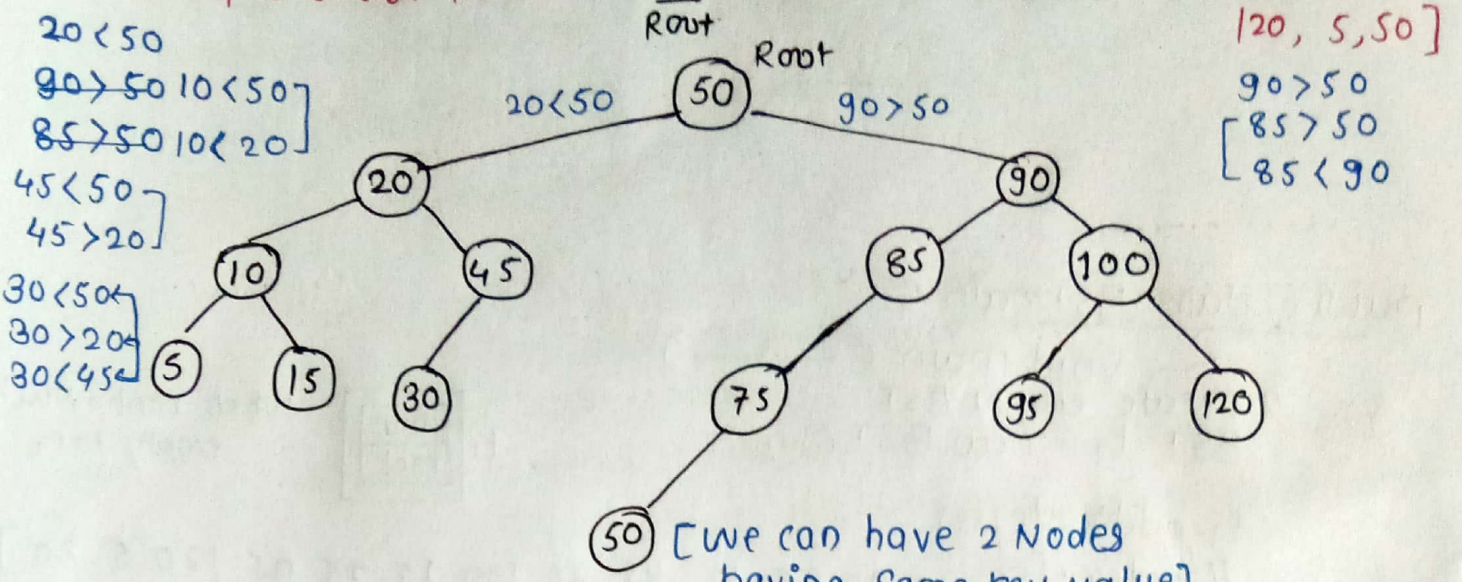
① Binary Search Tree → It is Binary Tree in which
(L<P) Left child is Alway smaller than parent
(R>=P) Right child is Always greater or equal to its parent.



While Adding node into BST
First Find its appropriate position (comp. values)
then Add node to resultant position

Start traversal From → Root Node

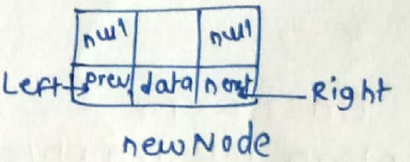
① Input order for BST : [50, 20, 90, 85, 10, 45, 30, 100, 15, 75, 95, 120, 5, 50]



50 [we can have 2 Nodes having same key value]

② Each Node in BST : 3 parts

1. data part : contains actual data of any primitive/ Non-primitive
2. Left Link
3. Right Link




```
Class BST
{
    static class Node {
        int data;
        Node left; // refer to left child of node
        Node right; // refer to right child of node

        Node(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }
private Node root; // type of Node refers to root node in BST

    BST() {
        this.root = null;
    }

void addNode (int data) { // create new Node first
{       Node newNode = new Node (data);
    if (isBSTEmpty())
        {   root = newNode

        }
```

```
else { // if BST Not empty start trav. to find appropriate position
    Node trav = root;
```

```
public class BSTMain {
    void main ( )
{   // create empty BST
    BST t1 = new BST ();

    t1.addNode(50);
    // order: [50 20 90 85 10 45 30 100 15 75 95 120 5 20]
              root  level 1

    try {
        t1.preorder ();
        } catch (RunTimeException ~~("BST is empty~~ e)
            { print (e.messege ());
            }
}
}
```

When root = Null empty tree

```
class BST {
boolean isBSTEmpty () {
        return (this.root == null);
}
```

⊙ There Are 2 Tree Traversal

① BFS / levelwise traversal

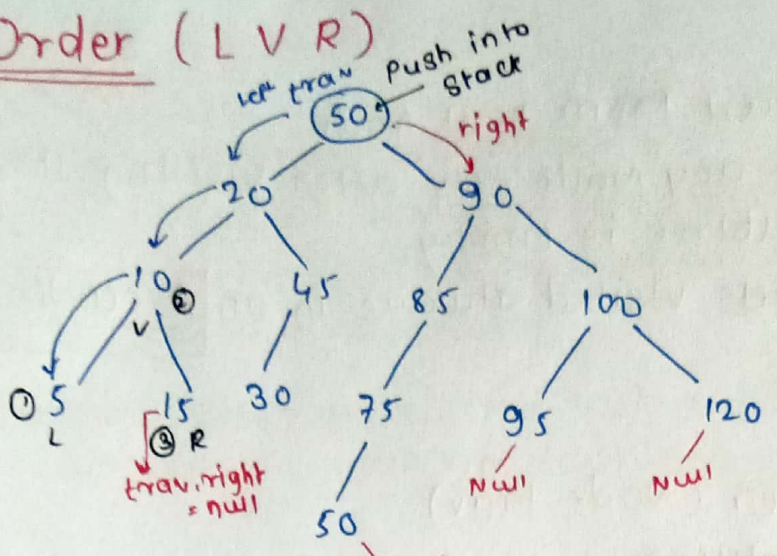
② DFS traversal
- 2.1 preorder → (VLR)
- 2.2 Inorder → (LVR)
- 2.3 postorder → (LRV)

① BFS → traverse Always From Root
- → Nodes in BST gets visited levelwise From left to Right

```
void addNode ( int data)
{
  Node newNode = new Node (data);
      if (isBSTEmpty())  // if BST empty
        {
          root = newNode;
        }
       else { // if BST is Not empty
          // start trav From root node
          Node trav = root;
          while (true) { // traverse to find appropriate position.
              if (data < trav.data) // Node will part of left subtree
                { if (trav.left == null)
                    {
                      trav.left = newNode;  // Attach newly created node as left child of current node.
                      break;
                    } else { trav = trav.left; }   // if current Node has left subtree
                } else {                  // node will be part of Right subtree of current
                    if (trav.right == null)
                      {
                        trav.right = newNode;
                        break;
                      }
                    else { // if curNode has Right subtree
                        trav = trav.right;
                        }
                }
          } // end of while
        }
}

void preorder () // Wrapper Function
{       if (isBSTEmpty()) {
            throw new RunTimeException ("BST is empty");
        }
        else {
            print ("preorder traversal is: ");
            preorder(root); // initialization
            println();
}       }       }
```

```
void preorder(Node trav) {
        // Base condition
          if (trav == null)
              return;
        // VLR
        printf("%d", trav.data); // visit current Node
          preorder(left trav.left); // visit left subtree
           preorder(trav.right); // visit right subtree
}
// Recursive function for AddNode
void recAddNode(int data)   // Wrapper function
{   if (isBSTempty())   // if BST empty Node will be root
      { root = newNode new Node(data);
      }
      else { // if BST is Not empty
          recAddNode(root, data); // For traversal we are calling recursive func starting from root
          }
}

Void recAddNode(Node trav, int data)
{     // Base condition
      if (trav == null)
      {     return;
          if (data < trav.data) // Node will part of left sub tree of current Node
            {  if (trav.left == null) // if left subtree of cur Node is null
                {  trav.left = new Node(data); // create New Node attach it as left child of cur Node
                   return;
                }
                else { // if cur Node is having left sub tree
                    recAddNode(trav.left, data);
                    }                    // go to left subtree
            }
            if (trav.right == null) // if right sub tree of cur Node is null
              { trav.right = new Node(data);
                return;
              }
```

⊙ InOrder (L V R)



Stack <Node>

~~8~~	~~15~~	~~30~~
~~10~~	~~20~~	~~45~~
20	~~20~~	~~50~~
50	50	
~~50~~		
~~75~~		
~~85~~	~~95~~	120
90	~~100~~	

inorder(LVR) : 5 10 15 20 30 45 50 50 75 85 90 95 100 120

```
void nonRecInOrder()
{   while (trav != null || !s.empty())
    {
        while (trav != null)
        {   // push cur Node onto stack
            s.push(trav);  ---- 45|30|null|90
            // go to its left subtree
            trav = trav.left;
        }
        // if stack is not empty
        if (!s.empty())
        {   // pop node from stack & catch it into trav
            trav = s.peek(); 5|10|15|20|30|45|50|50|75|85|90|95|100|120
            s.pop();
        // if we are popping node from stack
        either its left subtree is empty or already visited
        hence visit current Node
        printf("%d", trav.data);
        // goto its Right subtree
        trav = trav.right;
        }
    }
}
```