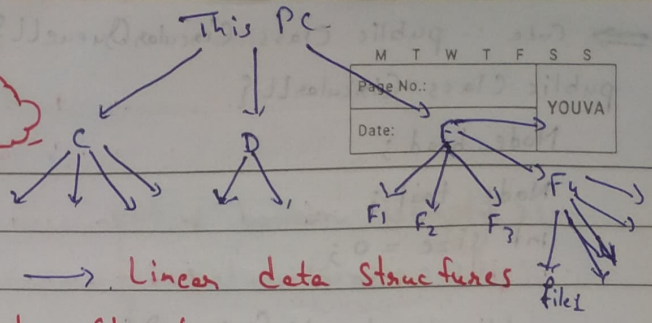
Binary Trees.

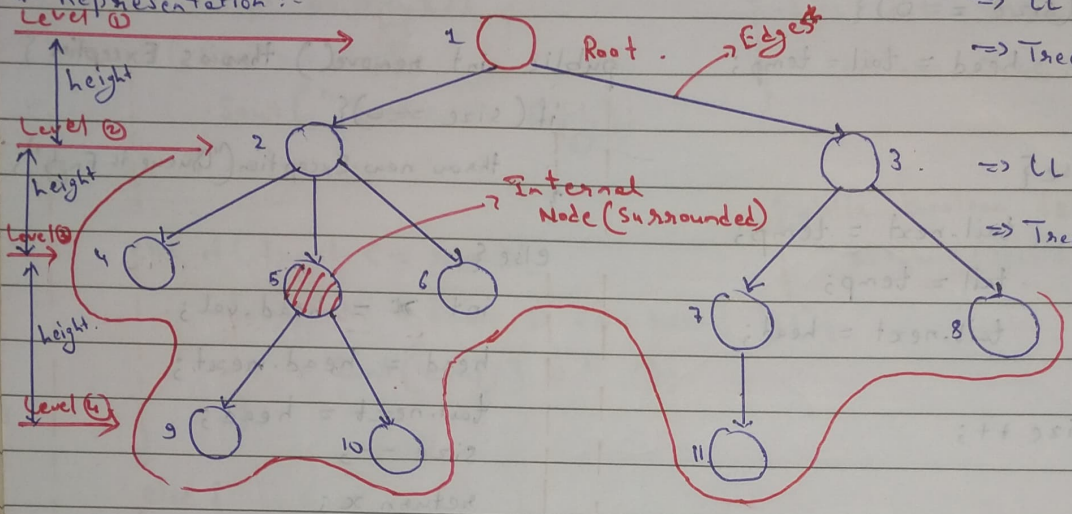


→ What is a Tree DS?

Array, LL, Stack, Queue → Linear data structures

Tree → Heirarchical data Structure.

* Representation :-



⇒ LL : head
⇒ Tree : Root

⇒ LL : tail
⇒ Tree : Leaf Nodes

Terminology :-

→ Root → Jiska koi Parent Nehi hota or Starting Point. case

→ Child Node → Ex①:- (2) & (3) are Child of (1) → For this (1) is Parent Node

→ Parent Node → Ex②:- (2) is parent Node then (4) (5) (6) are Child Nodes of (2)

→ Sibling Nodes → Means Nodes with a common Parents.

→ Ex*:- (4), (5), (6) Nodes have Common Parent (2).

→ Leaf Node → with 0 childrens Ex:- (4), (9), (10), (6), (11), (8)

→ Internal Node → Surrounded Node Ex:- (5) & (7).

→ Ancestor Node :- → Ex:- 9 ke Ancestor → (1), (2), (5)

→ Descendent Node :- → Ex :- 2 ke Descendants → (4), (5), (6), (9), (10).

→ Level :- Generations.

→ Number of edges :- → size - 1

→ Height :- level - 1

→ Size :- No. of Nodes.

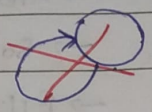
Ex:-
Levels = 4
Size = 11
Edge = 10
Height = 3.

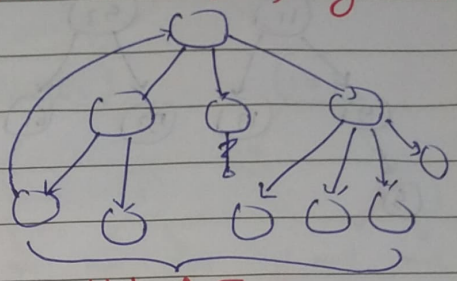
Important Properties of trees:-

→ Traversing in a tree is done by Depth First Search (DFS) & Breadth First Search (BFS) Algorithm

→ It has No Loop & No Circuit.

→ It has No self-Loop

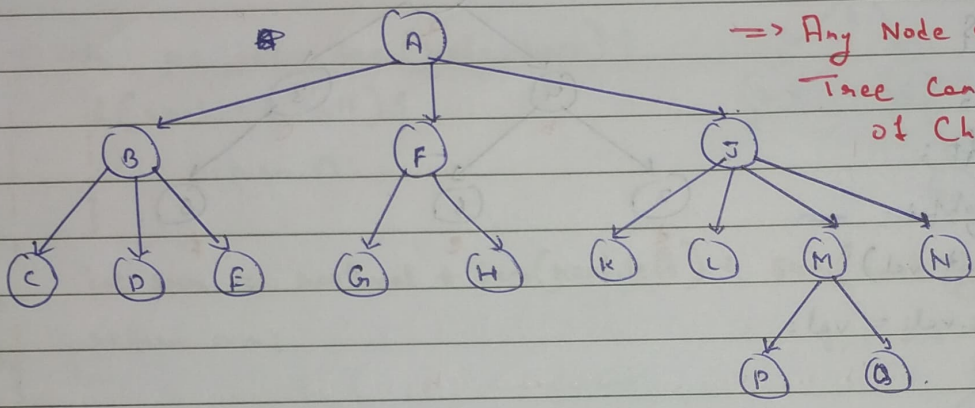




Not A Tree.

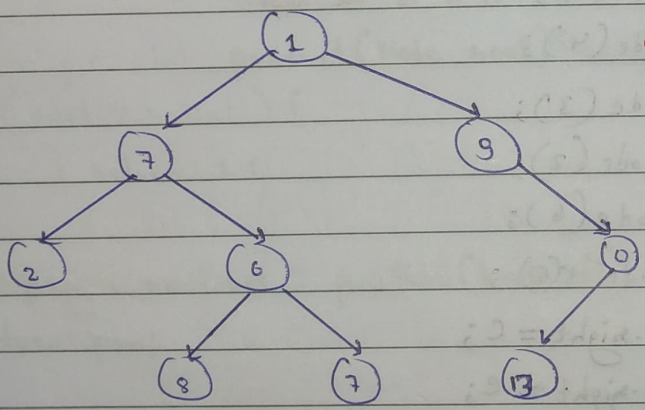
Types of Trees:-

(1) Generic Trees

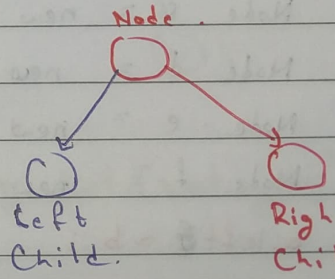


⇒ Any Node in a Generic Tree can have Any No. of Child Nodes.

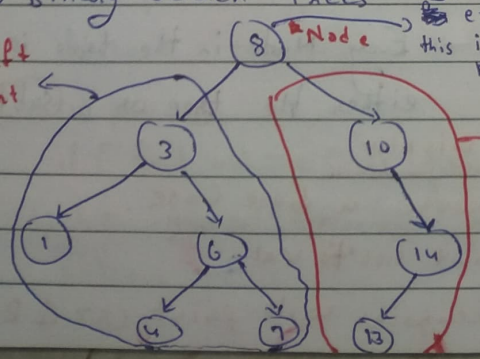
(2) Binary Trees



⇒ Any Node can have 0, 1 or 2 Child Nodes.



(3) Binary Search Trees

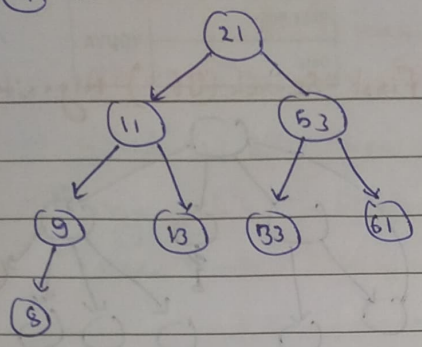


⇒ Iske baare me baad me acche se Padhenge.

⇒ Values in Left Part Nodes will be always Less then Parent Node

⇒ Values in Right Part Nodes will be Always Greater then Parent Node

(4) AVL Trees :-



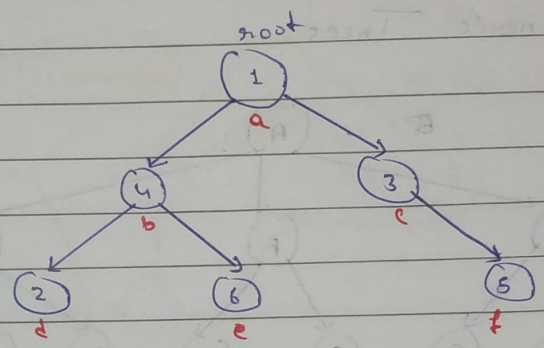
⇒ BST's which are Self Balancing.

Applications of Tree DS :-

- → Hierarchical ds
- → Searching efficient
- → Sorting
- → Dynamic Data.
- → Efficient Insertion & Deletion
- → Easy to Implement.

Implementation :-

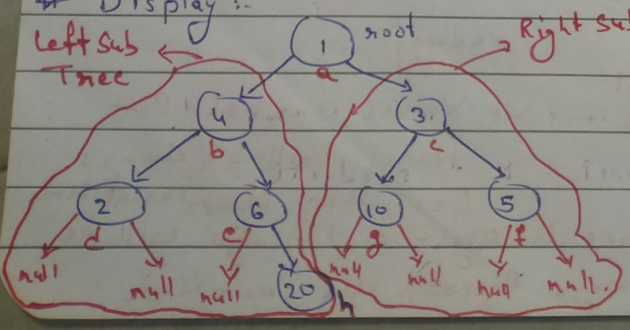
→ Creating A Node Class :-




```
Class Node {
    int val;
    Node left;
    Node right;
    Node(int val){
        this.val = val;
    }
}
```

```
psvm() {
    Node a = new Node(1); // a is the Root.
    Node b = new Node(4);
    Node c = new Node(3);
    Node d = new Node(2);
    Node e = new Node(6);
    Node f = new Node(5);
    a.left = b;    a.right = c;
    b.left = d;    b.right = e;
    c.right = f;
}
```

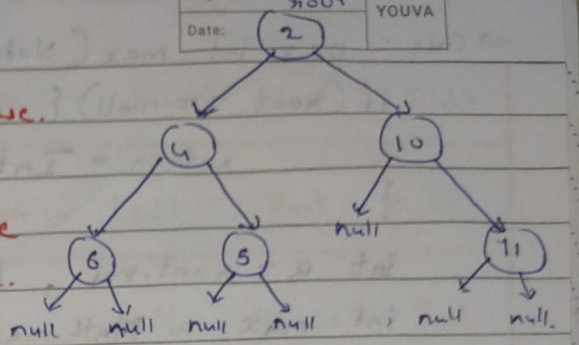
Display :-



→ Every Node in the tree is either the tree or a Subtree

To Display :-

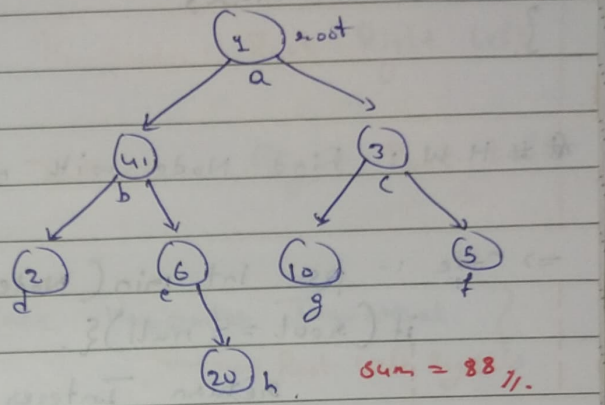
1. Ensure a base Case.
2. Print root's val.
3. Recursion will print LST & RST.



⇒ Code :- psv display (Node root){
if(root == null) return ; // Base Case.
Sout (root.val + " ");
display(root.left); // Left Sub Tree
display(root.right); // Right Sub Tree.
}

⇒ Output :- 2 4 6 5 10 11.

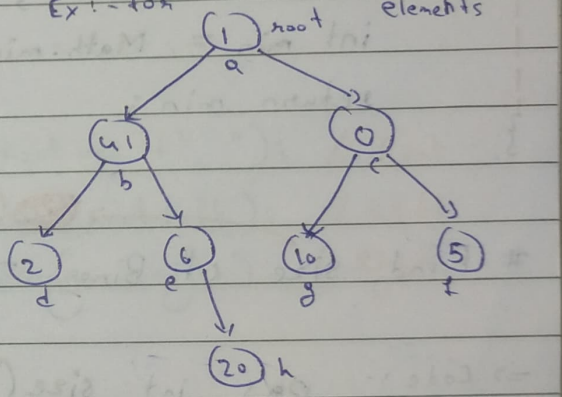
~~Sum~~ Find Sum of tree Nodes. → 88 //



⇒ Code :- ps int Sum(Node root){
if(root == null){
return 0;
}
int sum = root.val + Sum(root.left) + Sum(root.right);
return sum;
}

H.W :- ① Find product of tree Nodes. → 1,476,000 //

Ex :- for ~~Non~~ Product of Non-Zero elements



⇒ Code :- ps int product(Node root){
if(root == null){
return 1;
}
int pro = root.val * product(root.left) * product(root.right);
return pro;
}

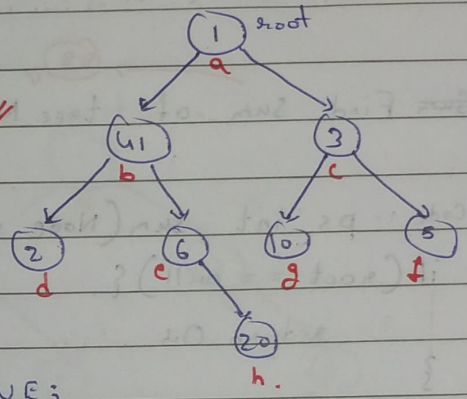
② Find product of Non-zero elements of nodes. → 492000 //

⇒ Code :- ps int product(Node root){
if(root == null) return 1;
int pro;
if(root.val == 0) pro = 1 * product(root.left) * product(root.right);
else pro = root.val * product(root.left) * product(root.right);
return pro;
}

Find Node with max Value :- → (41)

=> Code :-

```
ps : int max(Node root){
    if(root == null){
        return = Integer.MIN_VALUE;
    }
    int a = root.val , b = max(root.left) , c = max(root.right);
    int max = Math.max(a, Math.max(b,c));
    return max;
}
```



→ (1)

H.W :- Find Node with min Value :-

=> Code :-

```
ps int min(Node root){
    if(root == null){
        return Integer.MAX_VALUE;
    }
    int a = root.val , b = min(root.left) , c = min(root.right);
    int min = Math.min(a, Math.min(b,c));
    return min;
}
```

→ (8)

Find size of Binary Tree :-

=> Code :-

```
ps int size(Node root){
    if(root == null) return 0;
    int size = 1 + size(root.left) + size(root.right);
    return size;
}
```

4 - 1 => 3.

→ (4) → Level - 1.

Find Levels / Height of Binary Tree :-

=> Code :-

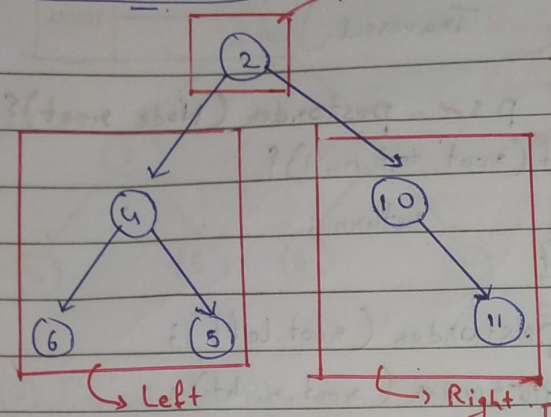
```
ps int level(Node root){
    if(root == null) return 0;
    int Level = 1 + Math.max(level(root.left), level(root.right));
    return Level;
}
```

(BFS)

Traversals :-

T.C => O(n)

S.C => O(n) → Recursive Stack Space



PreOrder :- Root Left Right

InOrder :- Left Root Right

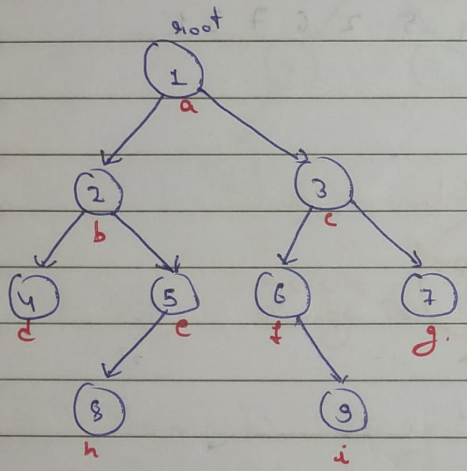
PostOrder :- Left Right Root

Reverse PreOrder :- Root Right Left

Reverse PostOrder :-
Right Left Root

Reverse InOrder :- Right Root Left

Leet Code Q. No. 144 { Binary Tree Preorder Traversal. → Root Left Right }




```
=> Code :- psv preorder(Node root){
    if(root == null){
        return;
    }
    Sout(root.val + " "); // Root
    preorder(root.left); // Left
    preorder(root.right); // Right.
}
```

=> 1 (4 8 2 5) (3 6 7 9)

=> 1 2 4 (8 5) 3 (6 9) 7

=> 1 2 4 5 8 3 6 9 7 //

LeetCode Q. No. 94 { Binary Tree Inorder Traversal → Left Root Right. }

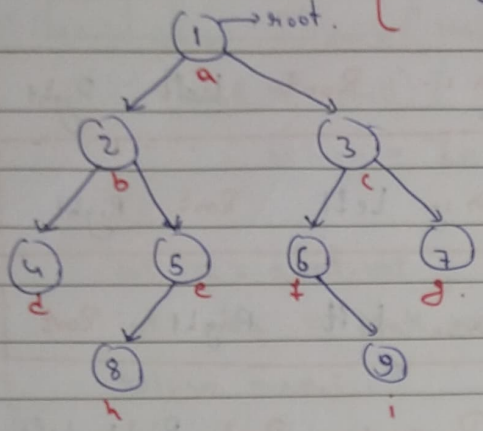
```
=> Code :- psv inorder(Node root){
    if(root == null){
        return;
    }
    inorder(root.left); // Left
    Sout(root.val + " "); // Root
    inorder(root.right); // Right.
}
```

=> (2 4 5 8) 1 (3 6 7 9)

=> 4 2 (5 8) 1 (6 9) 3 7

=> 4 2 8 5 1 6 9 3 7 //

LeetCode Q No. (145) { Binary Tree Postorder Traversal. } → Left Right Root.

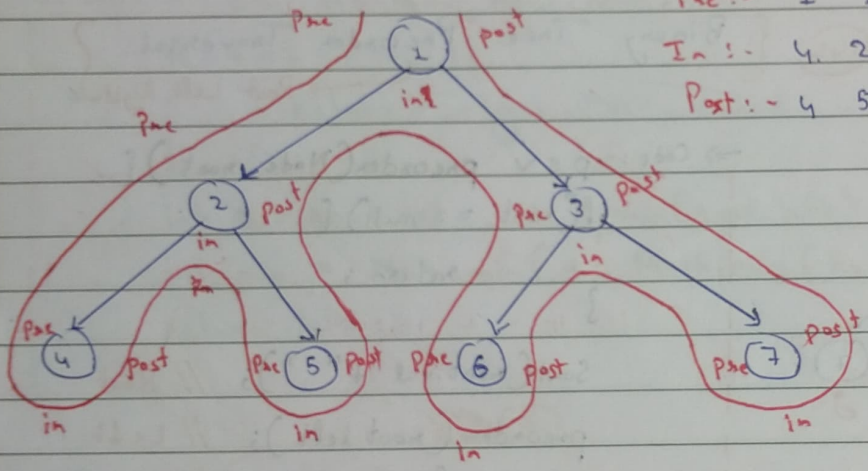


→ 4 8 5 2 9 6 7 3 1 //

⇒ Code :-

```
psv postorder (Node root){
    if (root == null){
        return;
    }
    postorder (root.left);
    postorder (root.right);
    Sout (root.val + " ");
}
```

⇒ 2nd Method To Print Traversals :-



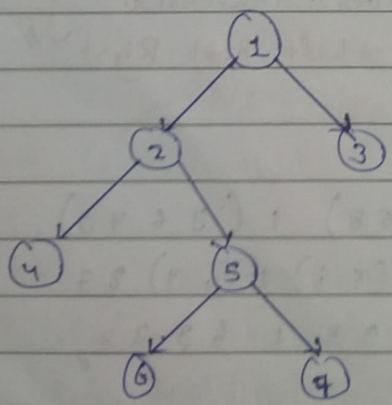
Pre :- 1 2 4 5 3 6 7 //

In :- 4 2 5 1 6 3 7

Post :- 4 5 2 6 7 3 1

Types of Binary Trees.

(1) Full Binary Tree.



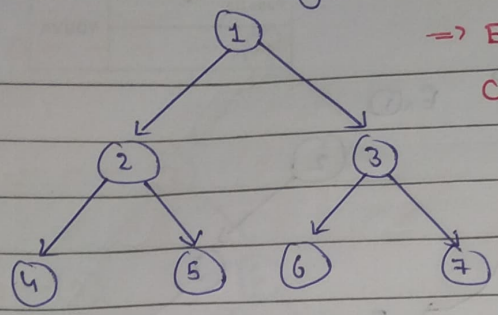
⇒ Every Node should have 0 or 2 Child Nodes.

M T W T F S S

Page No.: YOUVA

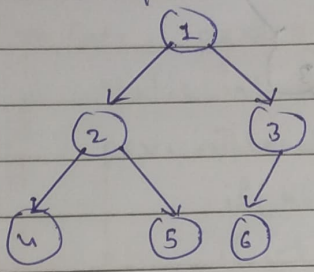
Date:

(2) Perfect Binary Tree.



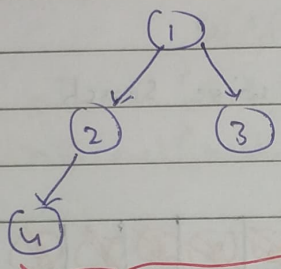
⇒ Every Node has 2 Child Nodes Except the Nodes of the Last Level which have 0 Child Nodes.

(3) Complete Binary Tree. → heaps.

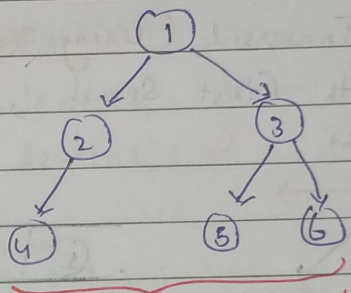


⇒ Is a perfect binary tree but its Last Level can be ≤ inComplete

⇒ Last Level need to be in continous manner from Left to Right.



It is a Complete Binary Tree.

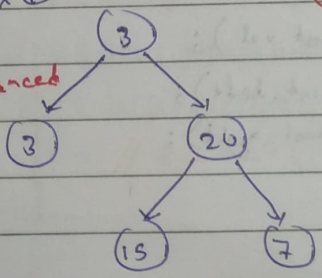


Last Level is Not in Continous from Left to Right.

It is Not Complete Binary Tree

(4) Balanced Binary Tree → V. imp.

Ex (1)

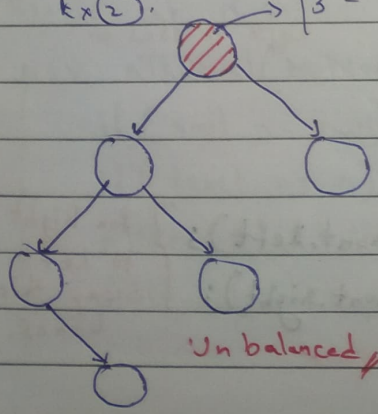


Balanced

→ where diff. b/w Levels(LST) & Levels(RST) <= 1. } For every Node

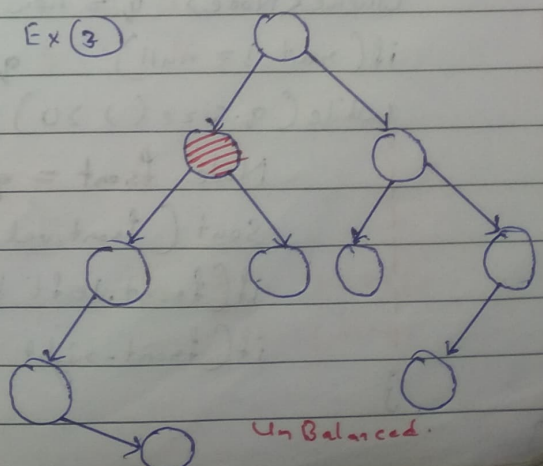
$|\text{Levels(LST)} - \text{levels(RST)}| <= 1$ ✓

Ex (2). → |3 - 1| is Not <= 1.



Unbalanced

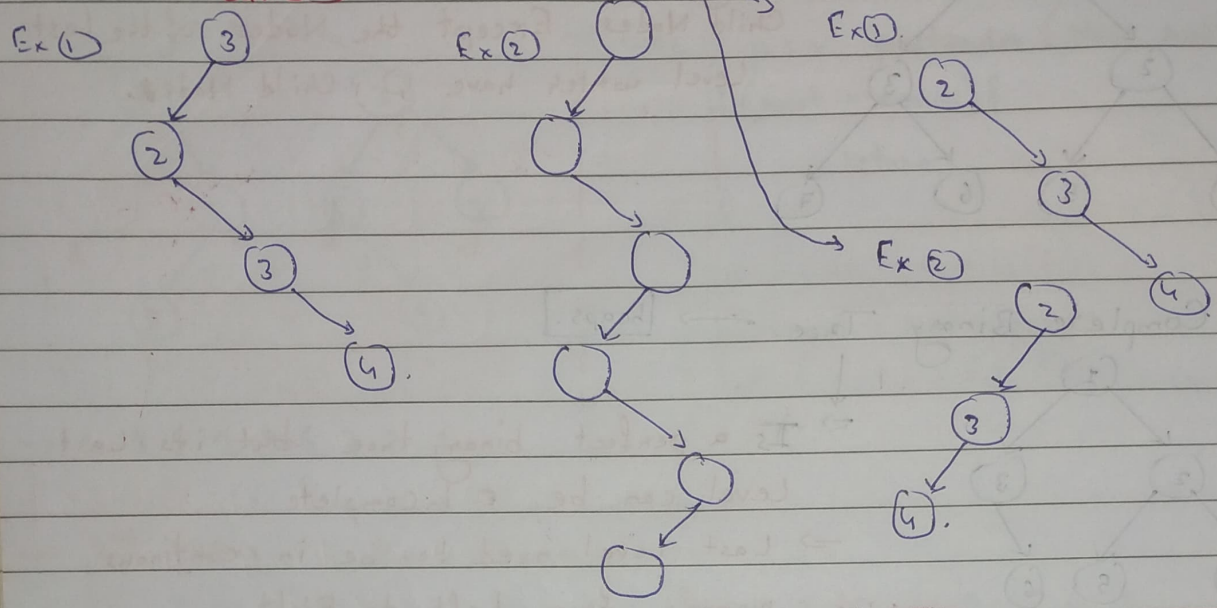
Ex (3)



Unbalanced.

(5) Degenerate & Skewed Binary Trees

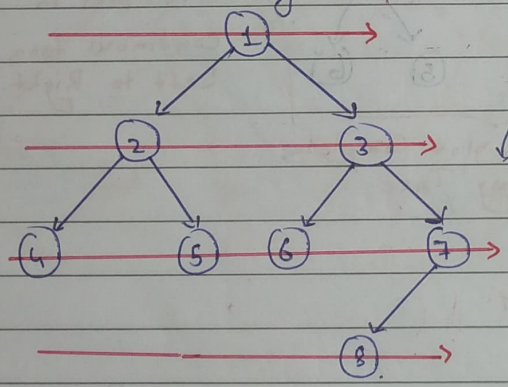
→ [0 & or 1 child]

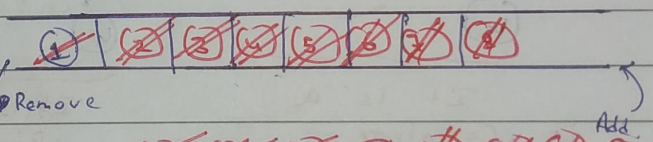


Level Order Traversal (Using Queue).

BFS → Breadth First Search, Level wise search

→ Left to Right.






```
q.add(root)
while( ){
    front = q.remove( );
    Sout (front.val );
    q.add(front.left);
    q.add(front.right);
}
```

Output :-

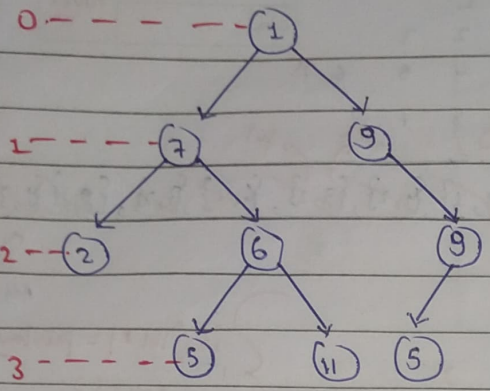
1 2 3 4 5 6 7 8

→ Code :-

```
psv levelOrder(Node root){
    Queue<Node> q = new LinkedList<>();
    if(root != null)    q.add(root);
    while(q.size() >0){
        Node front = q.remove();
        Sout (front.val + " ");
        if(front.left != null) q.add (front.left);
        if(front.right != null) q.add (front.right);
    }
}
```

for Right to left Swap this lines.

Print Elements of nth level (DFS)



Print 2nd Level
- L to R → 2 6 9.
- R to L → 9 6 2.

Hint :-

preorder (root, level) }
Inorder
Post Order
} Out put will be same for all.

⇒ Code :- // Left to Right.

Change is Print when level is equal to n.

```
Static int n;
psv nThLevel(Node root, int level){
    if(root == null) return;
    if (level == n)
        Sout(root.val + " ");
    nThLevel(root.left, level+1);
    nThLevel(root.right, level+1);
}
}
```

→ For Right to Left Swap these Lines.

Level Order Traversal (without using Queue) → [Using nth Level].

⇒ Code :-

```
psvm(TreeNode root){
    n = sc.nextInt();
    int x = n;
    for(int j=0; j<x; j++){
        n=j;
        nThLevel(a, 0);
        // Sout();
    }
}
```

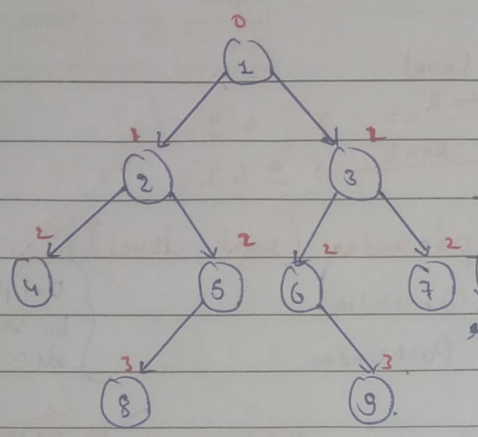
→ This Line is to Print like this :-

1
7 9
2 6 9
5 11 5.

```
Static int n;
// Left to Right
psv nThLevel(Node root, int level){
    if (root == null) return;
    if (level == n) Sout(root.val + " ");
    nThLevel(root.left, level+1);
    nThLevel(root.right, level+1);
}
```

→ For Right to Left Swap these Lines.

Level Order Traversal (using Queue) → Level wise Printing. BFS.



1
2 3
4 5 6 7
8 7.

prevlevel => 0; node lvl.

{1,0}, {2,1}, {3,1}, {4,2}, {5,2}, {6,2}, {7,2}, {8,3}, {9,3}

remove Add

1
2 3
4 5 6 7
8 9.

```
if(lvl != prevlevel)
    sout();
    prevlevel++;
}
```

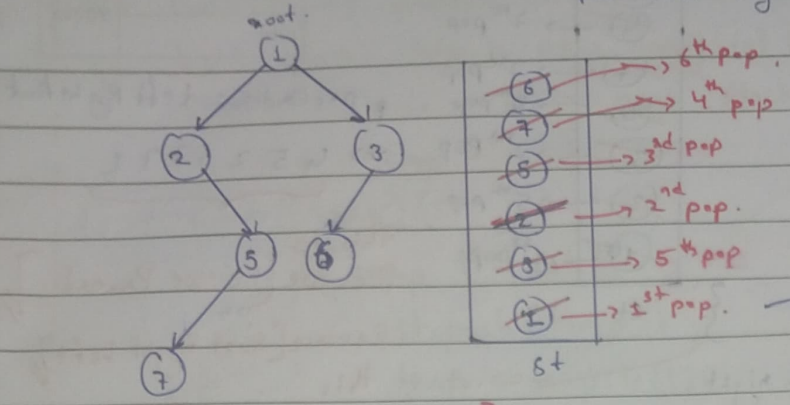
⇒ Code :-

```
public void levelOrder(Node root){
    int prevlevel = 0;
    Queue<Pair> q = new LinkedList<>();
    if(root!=null) q.add(new Pair(root, 0));
    while(q.size()>0){
        Pair front = q.remove();
        Node temp = front.node;
        int lvl = front.level;
        if(lvl != prelevel){
            Sout();
            prevlevel++;
        }
        Sout(temp.val + " ");
        if(temp.left != null) q.add(new Pair(temp.left, lvl+1));
        if(temp.right != null) q.add(new Pair(temp.right, lvl+1));
    }
}

public Class Pair{
    Node node;
    int level;
    Pair(Node node, int level){
        this.node = node;
        this.level = level;
    }
}
```

Preorder Traversal (Iterative)

Root Left Right.



⇒ [1 2 5 7 3 6]

⇒ print push right push left.

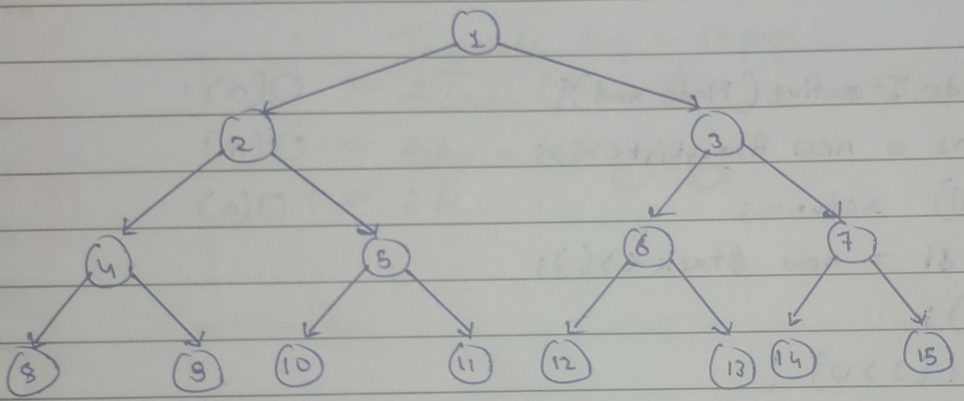
Recursive
↓
Stack
↓
Auxiallary Space.

Time Complexity :- O(n)

Space Complexity :- O(n)

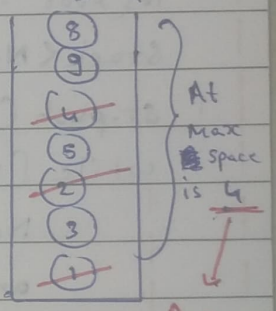
Auxillary Space :- $O(\log_2 n)$
↳ where h is level of tree

Best Case ⇒ O(1)



n = 15

$\log_2 n \cong$ level = 4



For a balanced B.T. pre :- 1 2 4

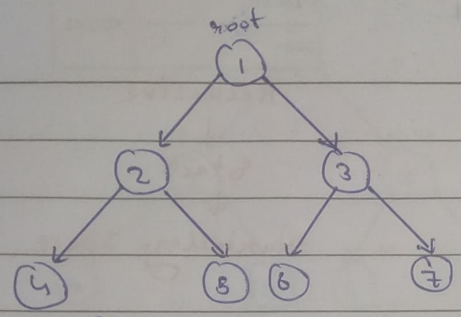
if n are nodes then height/levels $\cong \log_2 n$.

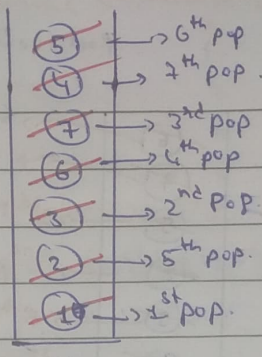
⇒ Code :-

```
public static void preorderIterative(Node root){
    Stack<Node> st = new stack<>();
    st.push(root);
    while(st.size() > 0){
        Node temp = st.pop();
        Sout(temp.val + " ");
        if(temp.right != null) st.push(temp.right);
        if(temp.left != null) st.push(temp.left);
    }
}
```

(2) → Left Right Root.

PreOrder Traversal (Iterative);





M	T	W	T	F	S	S
Page No.:						YOUVA
Date:						

preOrder = Left Right Root

=> 4 5 2 6 7 3 1

This is Reverse [Reverse Preorder] // .
or
Reverse [Root Right Left] //

ans => { 1, 3, 7, 6, 2, 5, 4 }

=> ~~pop~~ add ans, push left push right.

Reverse → ans => Ans. { 4, 5, 2, 6, 7, 3, 1 } //.

PreOrder //.

What is this

→ preOrder X

→ inOrder ~~=>~~ X. ?

→ It is [Reverse preorder] //

=> Code :-

```
public void postorderIterative (Node root){
    List<Integer> ans = new ArrayList<>();
    if(root == null) return;
    Stack<Node> st = new Stack<>();
    st.push (root);
    while (st.size() > 0) {
        Node temp = st.pop();
        ans.add (temp.val);
        if(temp.left != null) st.push(temp.left);
        if(temp.right != null) st.push(temp.right);
    }
    Collections.reverse (ans);
    System.out(ans);
}
```

T.C => O(n)

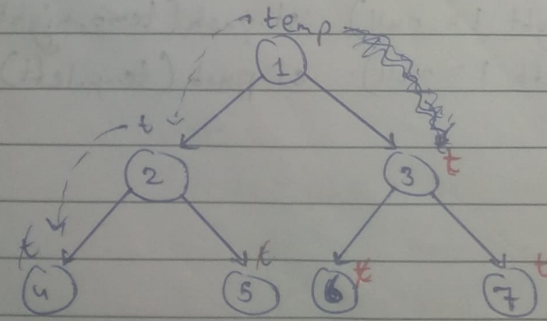
S.C => O(n)

A.S => O(n).

(3)

In Order Traversal (Iterative)

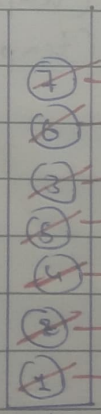
Left root right.



Stack & a variable (Node)

temp =>

ans => { 4 2 5 1 6 3 7 }




```
while (true) {
    if (temp != null) {
        st.push(temp);
        temp = temp.left;
    }
    else{
        Node top = st.pop();
        print;
        temp = top.right.
    }
}
```

⇒ Code :-

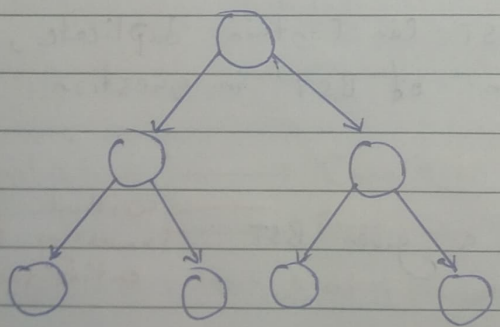
```
public List<Integer> inorderTraversal(TreeNode root){
    List<Integer> ans = new ArrayList<>();
    Stack<TreeNode> st = new Stack<>();
    TreeNode temp = root;
    while(true){
        if(temp != null){
            st.push(temp);
            temp = temp.left;
        }
        else{
            if(st.size()==0) break;
            TreeNode top = st.pop();
            ans.add(top.val);
            temp = top.right;
        }
    }
    return ans;
}
```

Time Complexity :- O(n).

Space Complexity :- O(n).

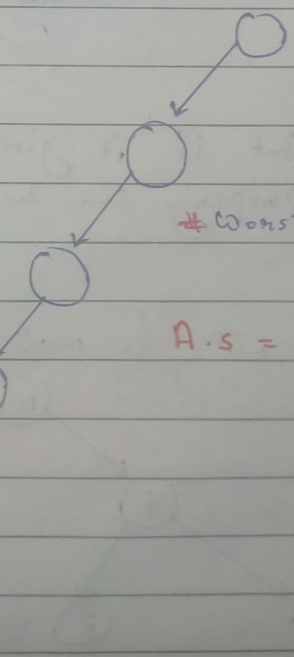
Auxillary space :- max space of stack O(h), → height.

* Best Case :- Balanced B.T.



A.S :- $O(\log_2 n)$,

* Worst Case :-



A.S = O(n)/O(h).