PEPCODING

PURSUIT OF EXCELLENCE AND PEACE

TREES MODULE

+91 11 4019 4461

PepCoding, 3rd Floor, 15 Vaishali,
Pitampura Opposite Metro Pillar 347,
Above Karur Vysya Bank, New Delhi,
Delhi 110034

www.pepcoding.com

/pepcoding

LEAF AT SAME LEVEL

```
public static boolean check(TreeNode root) {
        level = -1;
        return helper(root, 0);
    }

    static int level = -1;
    public static boolean helper(TreeNode node, int l) {
        if (node == null) {
            return true;
        }
        if (node.left == null &&node.right == null) {
            if (level == -1) {
                level = l;
                return true;
            } else {
                if (l != level) {
                    return false;
                }
                return true;
            }
        }

        booleanlres = helper(node.left, l + 1);
        if (lres == false) {
            return false;
        }
        booleanrres = helper(node.right, l + 1);
        return rres;
    }
```

What?

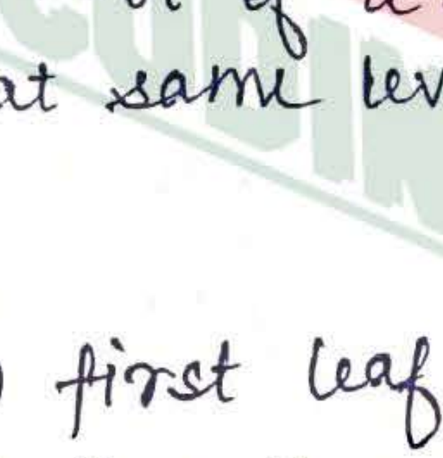
Given root of a binary tree check whether all leaves are at same level or not.

How?

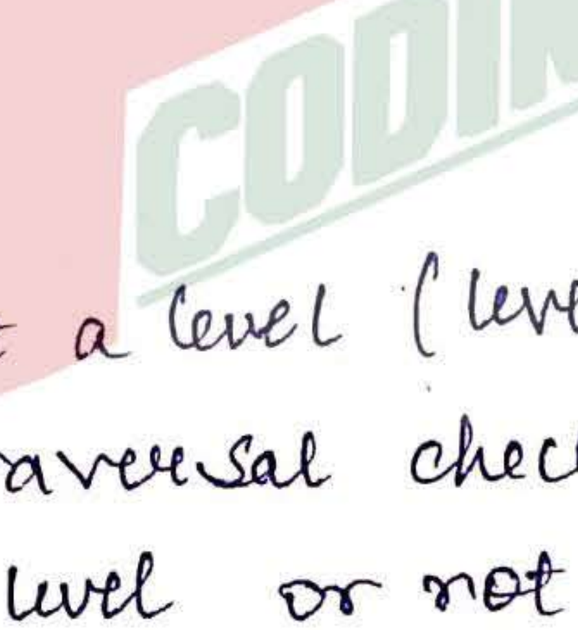
* When first leaf arrives set a level (level for all) and then throughout the traversal check if current level matches to the .all level or not.

* If level matches traversal is continued else result is false.

ex:



All leaves at same level.



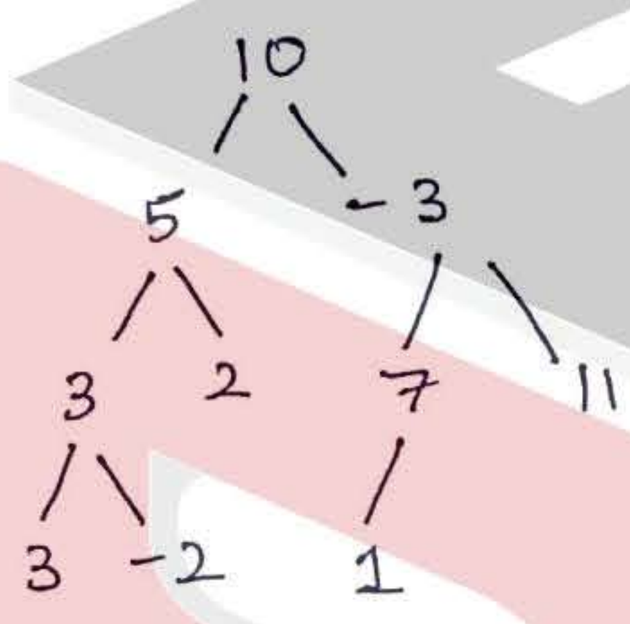
Leaves at different level.

COUNT NON LEAF NODES

```
public static int countNonLeafNodes(TreeNode root) {
        if (root == null) {
                return 0;
        }
        if (root.left == null &&root.right == null) {
                return 0;
        }
        int lc = countNonLeafNodes(root.left);
        int rc = countNonLeafNodes(root.right);
        if (root.left != null || root.right != null) {
                return 1 + lc + rc;
        }
        return lc + rc;
}
```

What? Count the number of non-leaf nodes in a tree.

ex:



Here, non-leaf nodes are
[10, 5, -3, 3, 7] = 5.

How? * If you are at a null node or a leaf node, return 0.

* Any other node checks that if any of its children is present, it returns left count + right count +1 (1 means it is also a non-leaf node), else it returns the sum of counts from left and right only.

BINARY TREE PATHS

```
public static ArrayList<String>binaryTreePaths(TreeNode root) {
        ArrayList<String> al = new ArrayList<>();
        al = helper(root, "", al);
        return al;
    }
    private static ArrayList<String> helper(TreeNode node, String psf, ArrayList<String> ls) {
        if (node == null) {
            return ls;
        }
        if (node.right == null &&node.left == null) {
            psf += node.val;
            ls.add(psf);
            return ls;
        }
        psf += node.val + "->";
        ls = helper(node.left, psf, ls);
        ls = helper(node.right, psf, ls);
        return ls;
    }
```

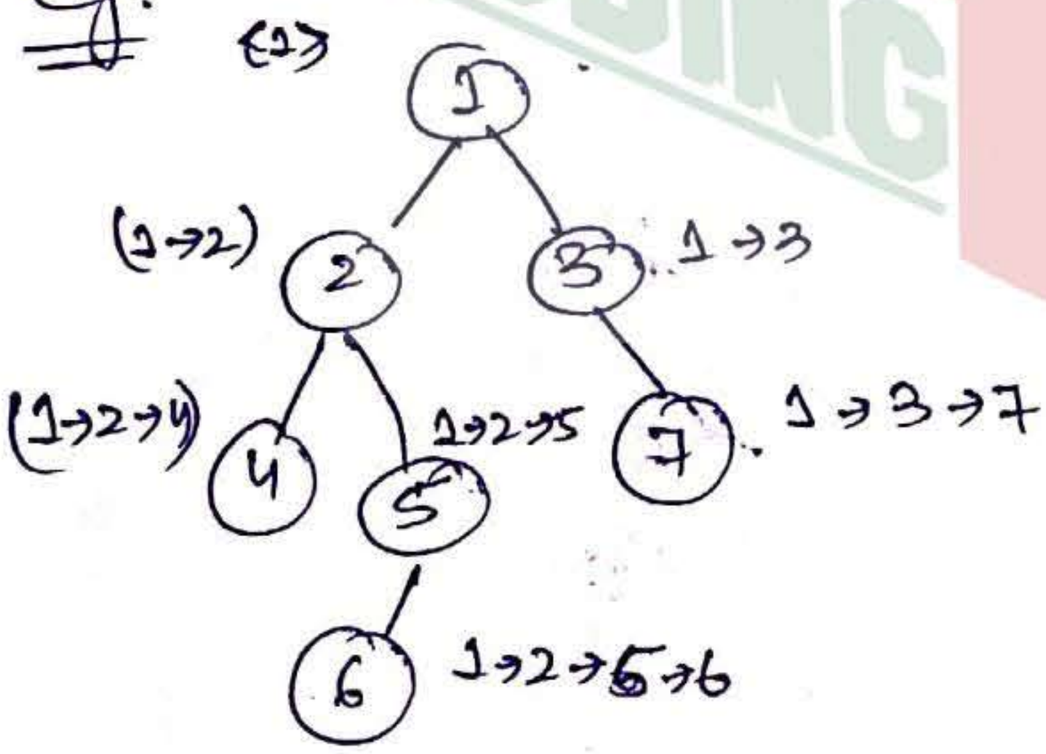
What?

Given root of a Binary Tree, Return an arraylist containing all root to leaf paths.

How?

We will maintain a psf that will contain path from root to current node. At leaf add psf to list.

Eg:



(Preorder Traversal)

[1→2→4 , 1→2→5→6 , 1→3→7]

Ans

PRINT BST ELEMENTS IN GIVEN RANGE

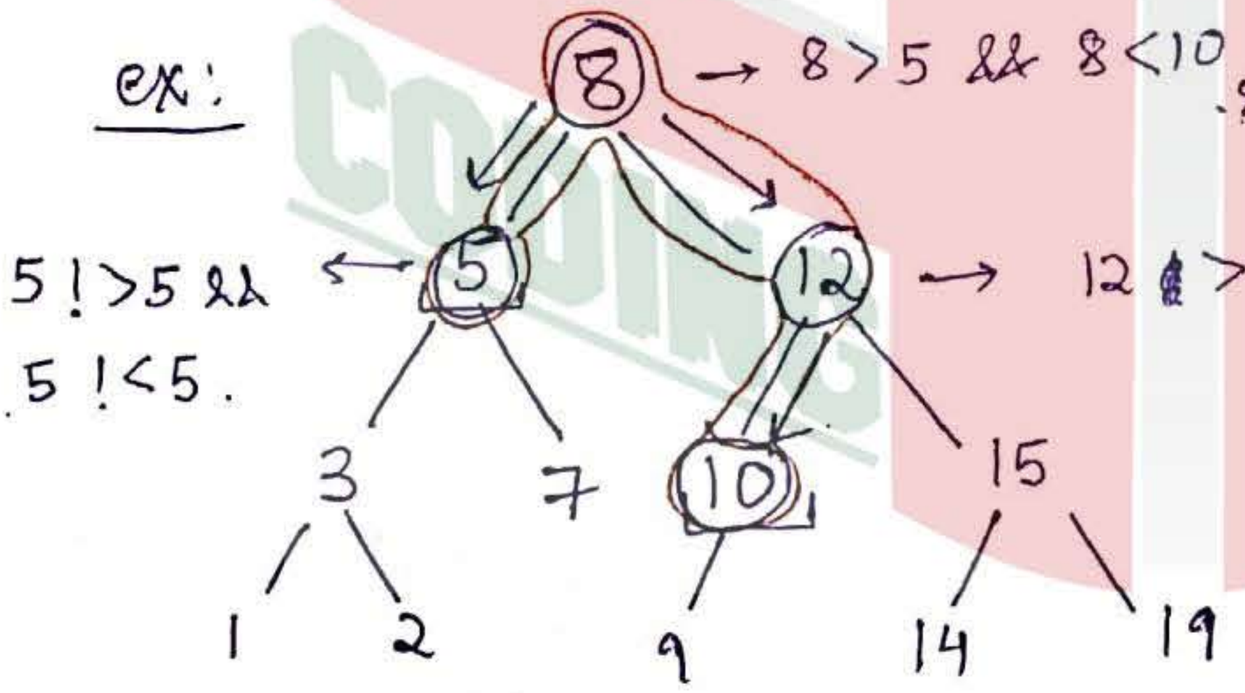
```
public static void printNearNodes(TreeNode node, int k1, int k2) {
        if (node == null) {
                return;
        }
        if (k1 <node.val)
                printNearNodes(node.left, k1, k2);
        if (node != null &&node.val>= k1 &&node.val<= k2) {
                System.out.print(node.val + " ");
        }
        if (k2 >node.val)
                printNearNodes(node.right, k1, k2);
}
```

What? Given a BST, and start and end of a range of numbers, print all nodes of BST within the range, both start and end inclusive.

How? * In a BST, if node.data > start, then answers might be present in left, also ;
and if node.data < end, then answers might be present in right.

* Check nodes in inorder, if in range, print.

ex:



Nodes are printed in inorder: 5, 8, 10

K FAR AND K DOWN

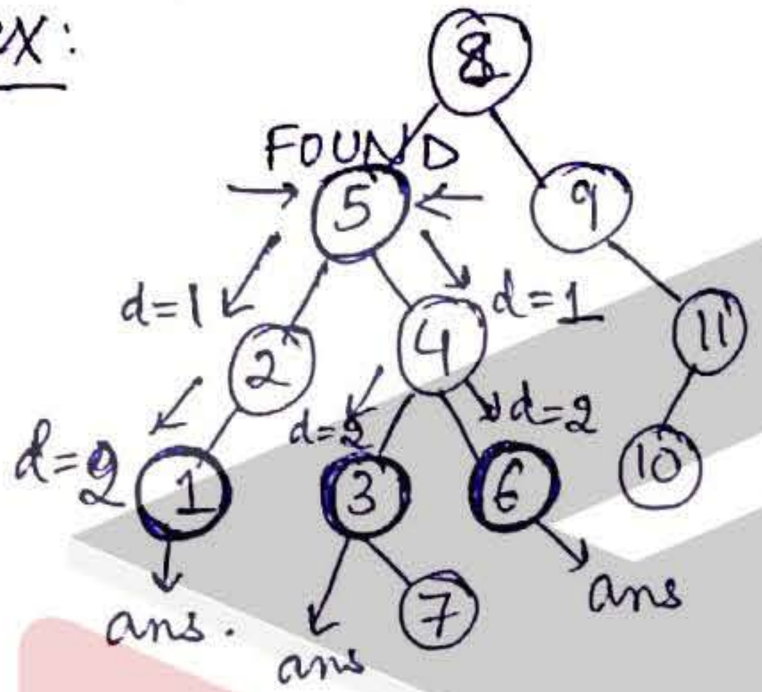
```
static List<Integer> l;
        public static List<Integer>distanceK(TreeNode root, TreeNode target, int K) {
                l = new ArrayList<>();
                List<TreeNode> list = new ArrayList<>();
                list = find(root, target);
                for (int i = 0; i<list.size(); i++) {
                        TreeNode node = list.get(i);
                        if (i != 0) {
                                if (i == K) {
                                        l.add(list.get(K).val);
                                }
                                if (list.get(i - 1) == node.right) {
                                        Kdown(node.left, 0, K - i - 1);
                                } else {
                                        Kdown(node.right, 0, K - i - 1);
                                }
                        } else {
                                Kdown(node, 0, K - i);
                        }
                }
                return l;
        }
        public static void Kdown(TreeNode node, int cd, int k) {
                if (node == null) {
                        return;
                }
                if (cd == k) {
                        l.add(node.val);
                }
                Kdown(node.left, cd + 1, k);
                Kdown(node.right, cd + 1, k);
        }
        public static List<TreeNode> find(TreeNode root, TreeNode target) {
                if (root == null) {
                        return new ArrayList<>();
                }
                if (root.val == target.val) {
                        List<TreeNode> list = new ArrayList<TreeNode>();
                        list.add(root);
                        return list;
                }
                List<TreeNode> left = find(root.left, target);
                if (left.size() > 0) {
                        left.add(root);
                        return left;
                }
                List<TreeNode> right = find(root.right, target);
                if (right.size() > 0) {
                        right.add(root);
                        return right;
                }
                return new ArrayList<>();
        }
```

What? a) Find k nodes down to a given node.
b) Find nodes k distance away to a given node.

How? For K down:

* Find the target node.
* Keep count of depth after the find has been done.
* If a node is at depth k from found node, print it.

Ex:



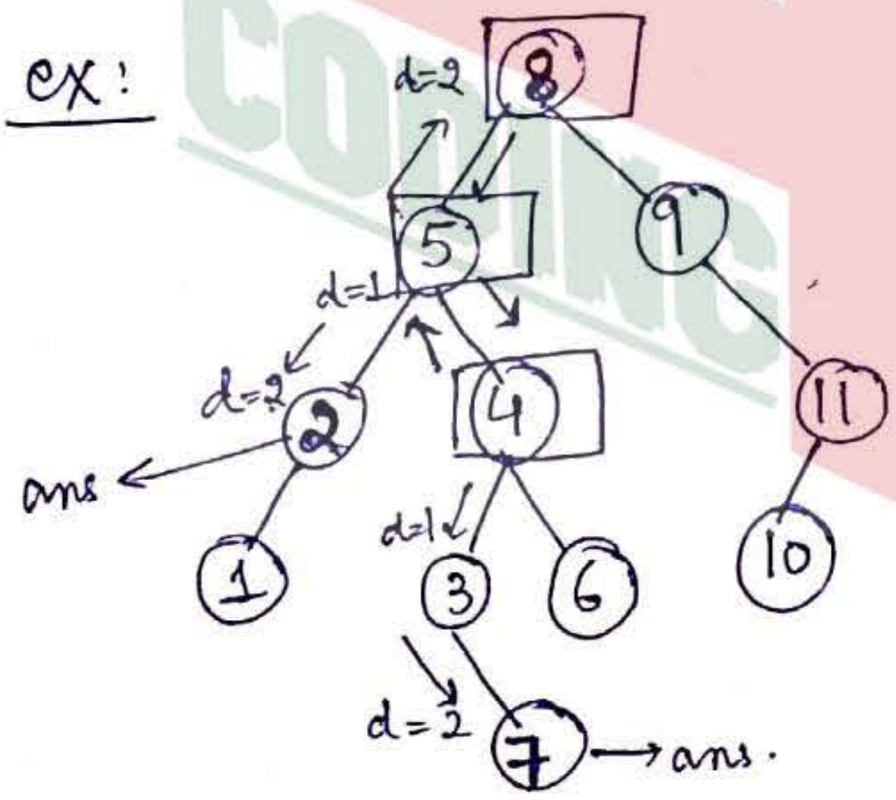
target = 5, k = 2..

⇒ ans = [1, 3, 6].

For k far:

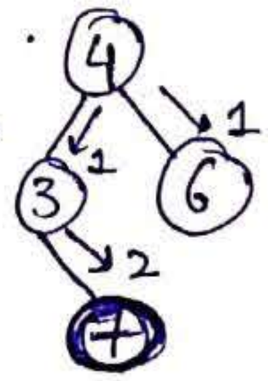
* Use find to get the target node to root path.
* Now for each node in the path, if it is ith node, then call K- down for (k-i) value as depth [(k-i-1) for left/right]

Ex:



target = 4, k = 2 ⇒ ans = [7, 2, 8]

Target to root path: [4, 5, 8].



1) Call 2 down for 4. →
2) Call (2-1) down for 5.
3) Call (2-0) down for 8.

Answer:
1) 7 (2 down for 4).
2) 1 down for 5 is 0 down for 5. left because (i-1) was 5. right(4)
⇒ 2 (0 down for 2).
3) 8 (0 down for 8).

* At each i != 0, we make a call (k-i-1) down for get(i).left if get(i-1) was get(i).right, else for get(i).right.

LINEARIZE

```
public void LinearTree() {
        Node n = LinearTree(root);
    }
    private Node LinearTree(Node node) {
        if(node.children.size() == 0) {
            return node;
        }
        Node otail = LinearTree(node.children.get(node.children.size()-1));
        for(int i=node.children.size()-2; i>=0; i--) {
            Node child = node.children.get(i);
            Node ctail = LinearTree(child);
            ctail.children.add(node.children.remove(i+1));
        }
        return otail;
    }
```

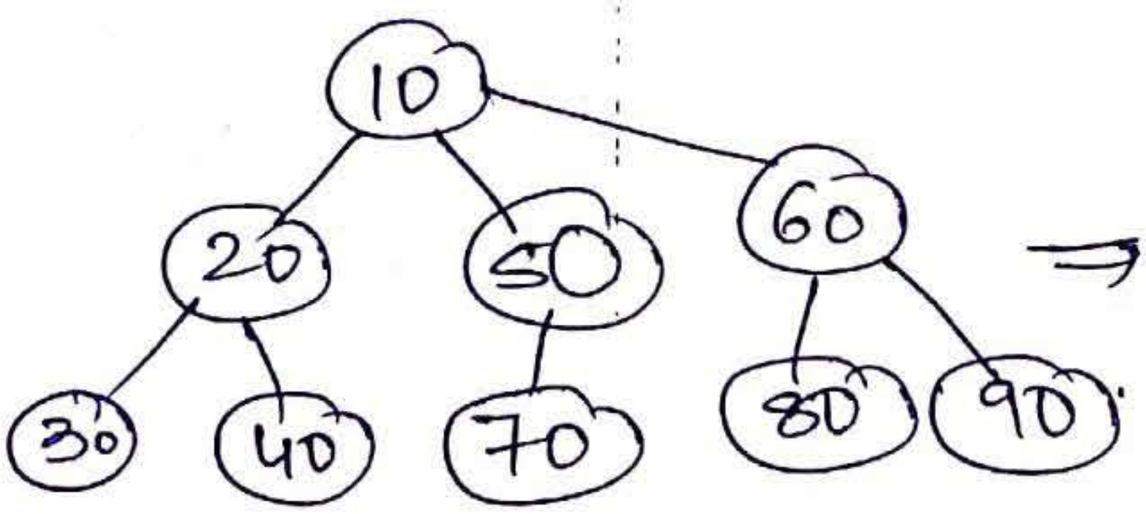
What?

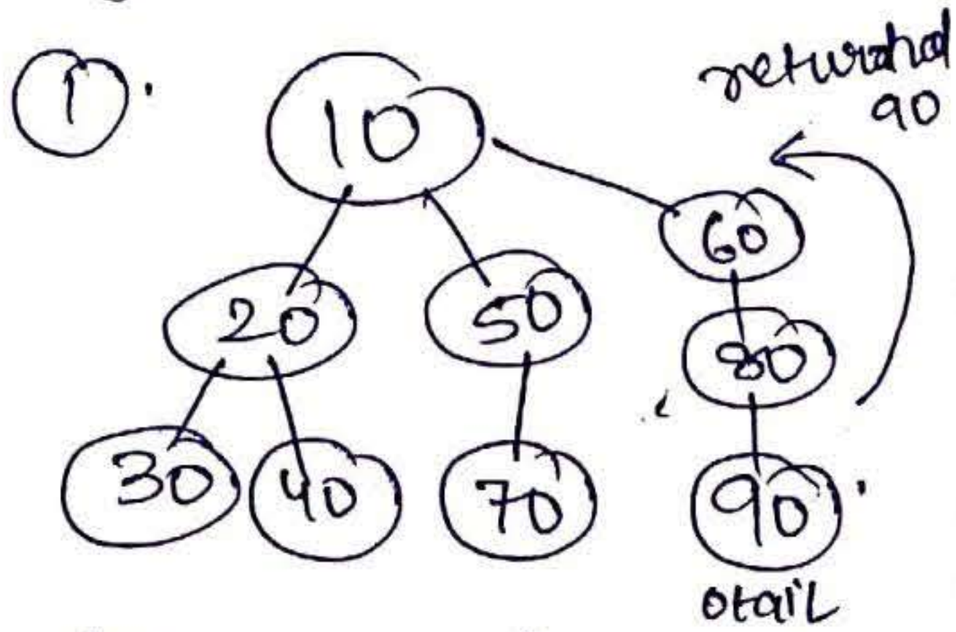
Given root of a generic Tree. Linearize a Tree ie each node has only one child and only 1 node is a leaf node.

How?

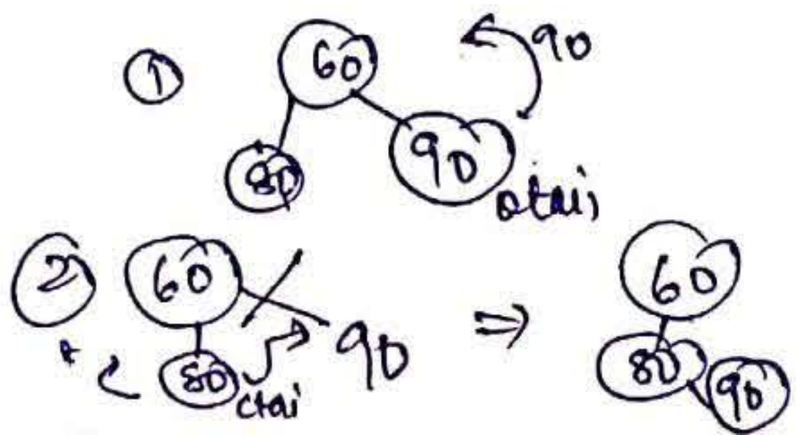
* By keeping faith that children will linearize themselves.
* An variable old tail is maintained which consist of node that will be the last node of the linearize subtree.
* An variable ctail is maintained which is the tail of the current child. returned after ~~tanla~~ linearizing that child.
* Now list will be all the nodes next child will now be appended to current child using ctail.children.add.

eg:

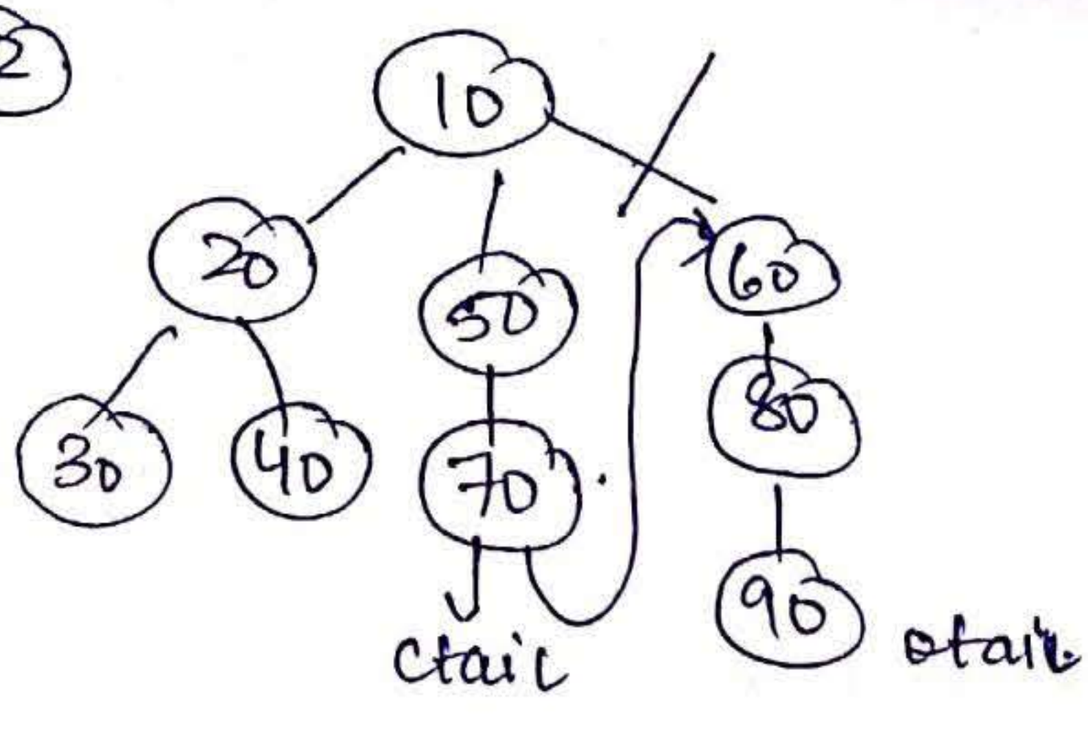




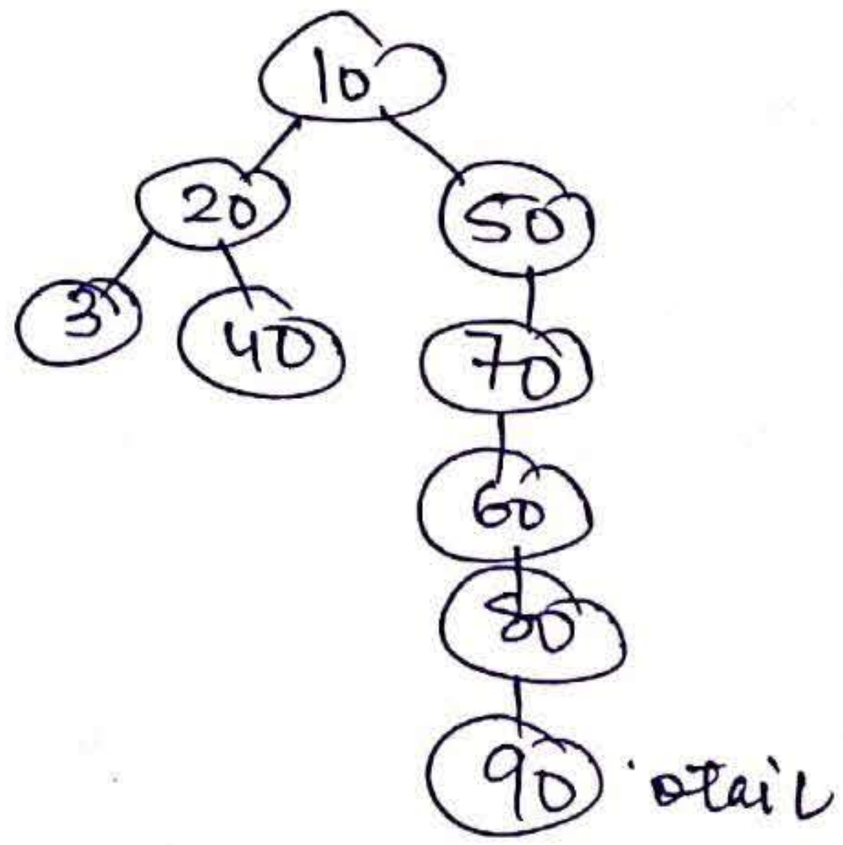
(Last child linearized first)



2

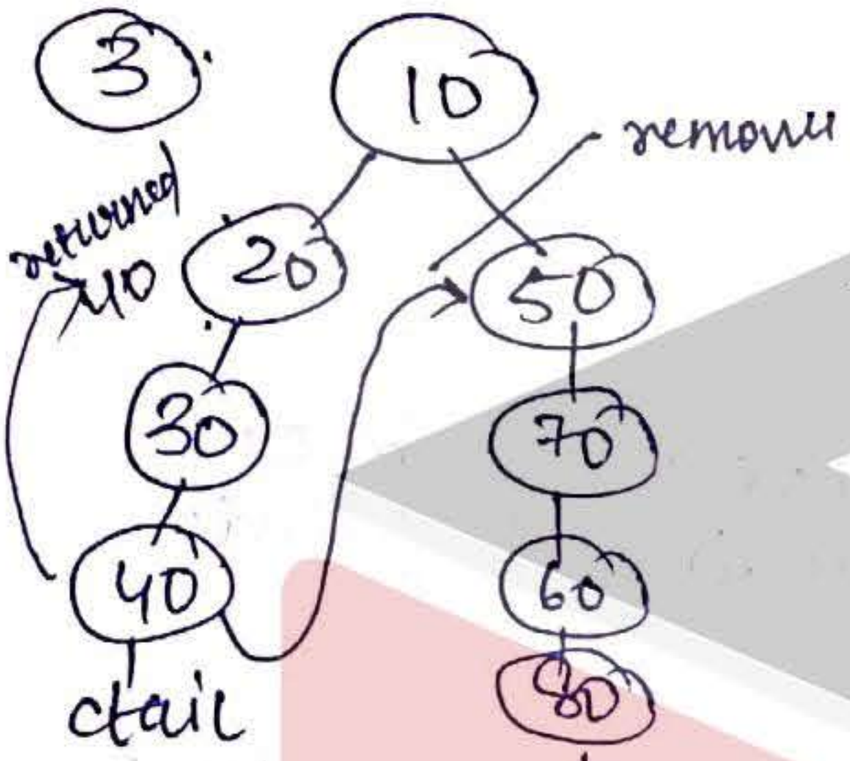


⇒



3

returned



removed

⇒

Ans

otail is returned.

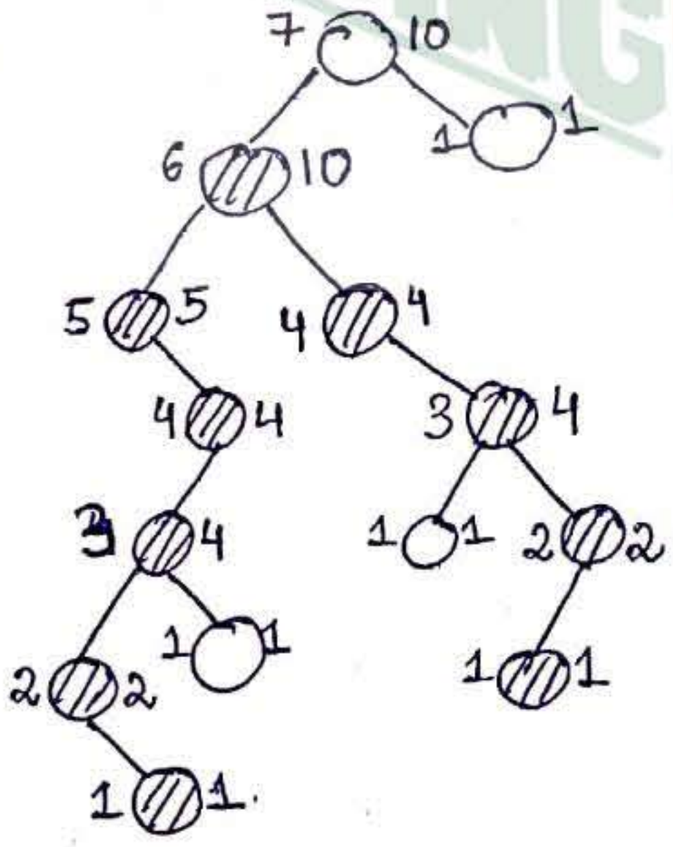
DIAMETER

```
private class DiaPair {
        int ht;
        int dia;
    }
    public int diameter2() {
        DiaPairdp = diameter2(root);
        return dp.dia;
    }
    private DiaPair diameter2(Node node) {
        if (node == null) {
            DiaPair bp = new DiaPair();
            bp.ht = 0;
            bp.dia = 0;
            return bp;
        }
        DiaPairlp = diameter2(node.left);
        DiaPairrp = diameter2(node.right);
        DiaPairmp = new DiaPair();
        mp.dia = Math.max(lp.ht + rp.ht + 1, Math.max(lp.dia, rp.dia));
        mp.ht = Math.max(lp.ht, rp.ht) + 1;
        return mp;
    }
```

What? Find the diameter of binary tree. The diameter is the number of nodes on longest path between two end nodes.

How? * Each node calculates its diameter as max of height of left subtree + height of right subtree + 1 for its current node, and left subtree diameter and right subtree diameter.

ex:



height ○ diameter

CEIL FLOOR

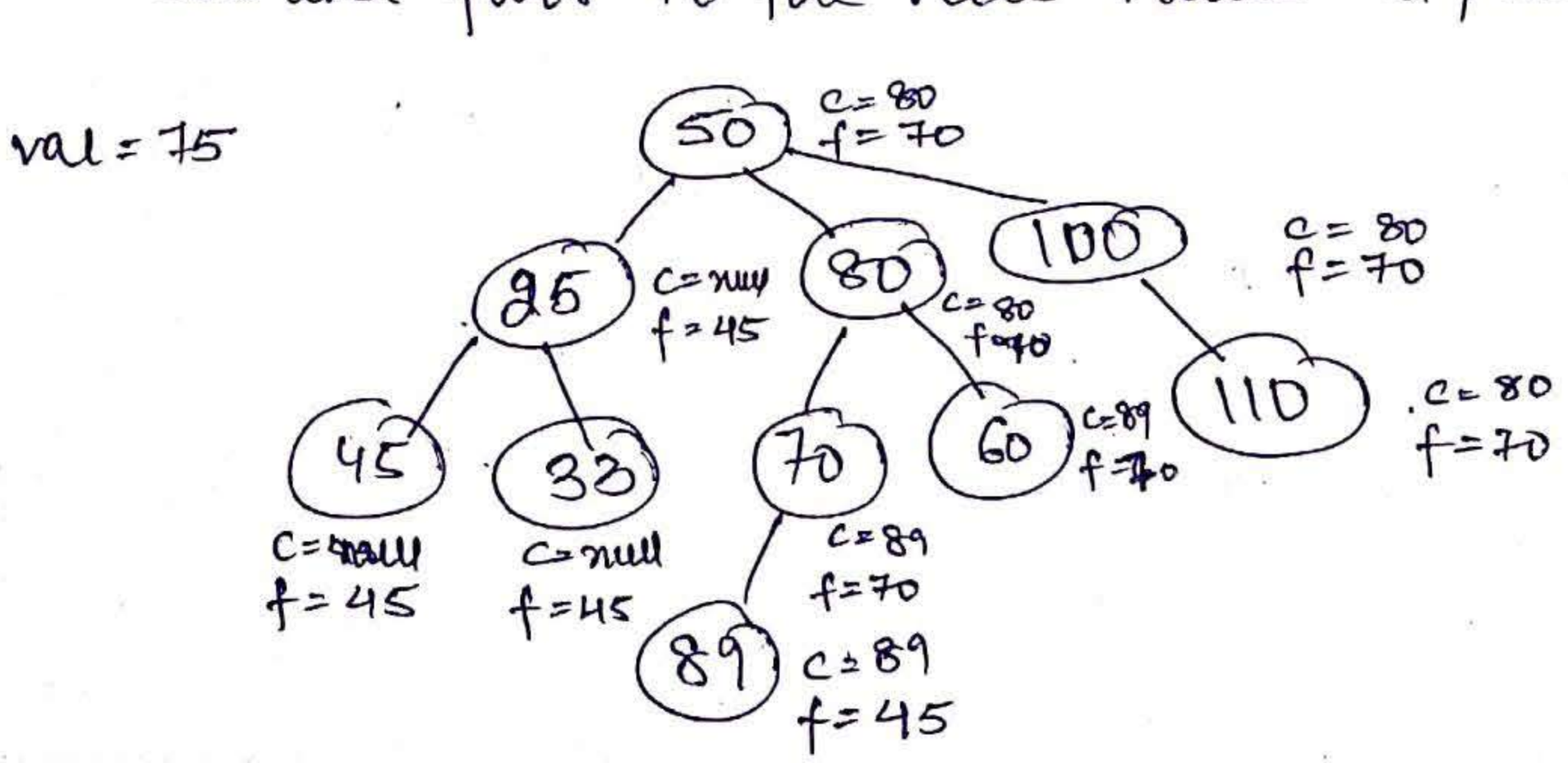
```
private Integer ceil;
        private Integer floor;
        public void ceilFloor(int data){
                ceil = null;
                floor = null;
                ceilFloor(data, root);
                System.out.println("Ceil = " + ceil);
                System.out.println("Floor = " + floor);
        }
        private void ceilFloor(int data, Node node){
                for(Node child: node.children){
                        ceilFloor(data, child);
                }
                if(node.data> data){
                        ceil = ceil == null? node.data: Math.min(ceil, node.data);
                }
                if(node.data< data){
                        floor = floor == null? node.data: Math.max(floor, node.data);
                }
}
```

<u>What?</u>

Given root of a generic tree and a value. Find the just smaller (floor) and ceil (just larger) of the given value.

<u>How?</u>

* We know that floor is the maximum of all the numbers smaller then current value.
* Ciel is the minimum of all the number greater than current value.
* Based on this property we will compare to current ciel and floor to the node value in postorder.



<u>Ans</u> ciel=80
floor=70

IS BALANCED

```
private class BalPair {
        int ht;
        booleanbal;
    }
    public booleanIsBalanced() {
        BalPairrp = IsBalanced(root);
        return rp.bal;
    }
    private BalPairIsBalanced(Node node) {
        if (node == null) {
            BalPair bp = new BalPair();
            bp.ht = 0;
            bp.bal = true;
            return bp;
        }
        BalPairlp = IsBalanced(node.left);
        BalPairrp = IsBalanced(node.right);
        BalPairmp = new BalPair();
        mp.ht = Math.max(lp.ht, rp.ht) + 1;
        mp.bal = Math.abs(lp.ht - rp.ht) <= 1 &&lp.bal&&rp.bal;
        return mp;
    }
```

<u>What?</u>

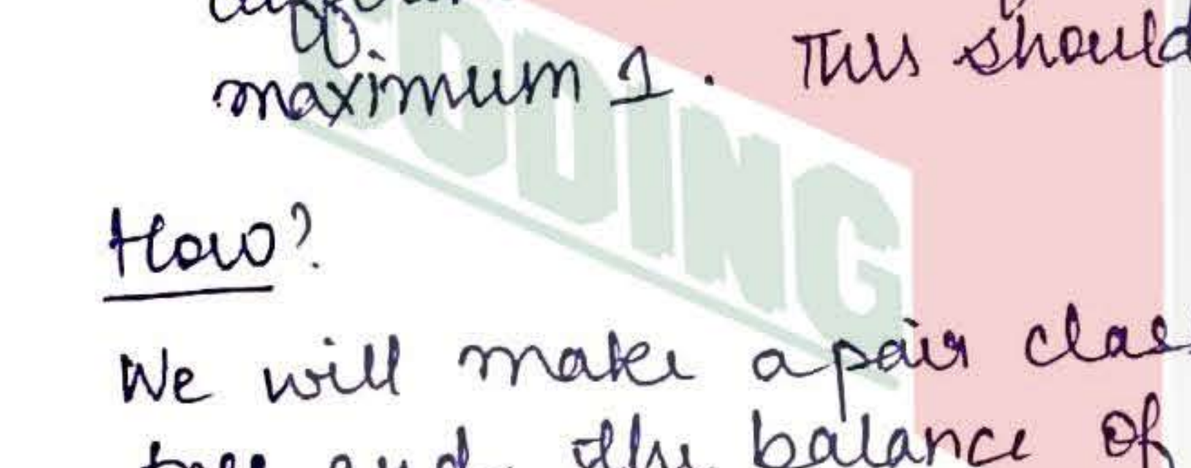
Given root of a Binary Tree. Check if tree is balanced or not. A Tree is set to be balanced if its Balancing factor bal = |(left tree height − right tree height)| <= 1 that is difference betwee left and right subtree height is maximum 1. This should be True for each node.

<u>How?</u>

We will make a pair class that will store ht of a tree and the balance of that tree.

For a tree to be balanced its left and right children need to be balanced and height diffence between both should be less than equal to 1.

bal = left ht − right ht



(Balanced)

(ImBalanced)

At (2) |balance| is greater than 1. So not balanced

NODE AT A DISTANCE

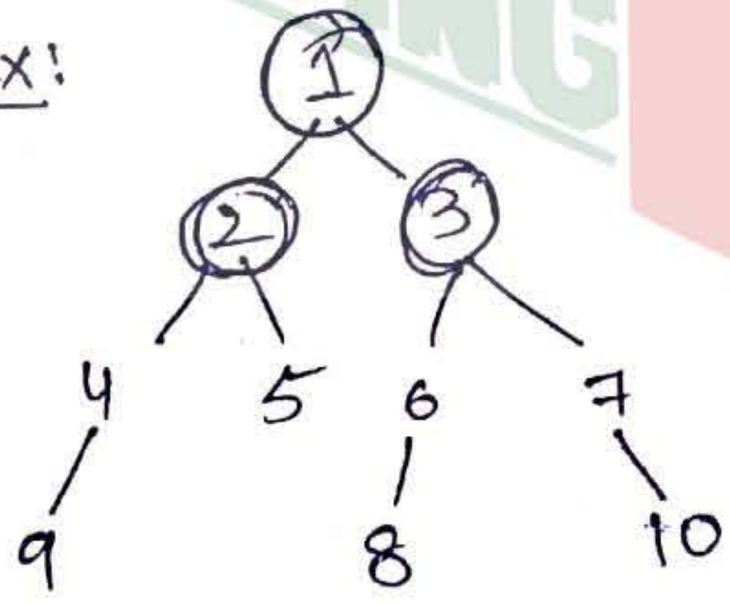
```
public static int KDistantfromLeaf(TreeNode node, int k) {
		count = 0;
		helper(node, k, 0, new ArrayList<TreeNode>(), new HashSet<TreeNode>());
		return count;
	}

	static int count = 0;
	public static void helper(TreeNode node, int k, int l, ArrayList<TreeNode>psf, HashSet<TreeNode> visited) {
		if (node == null) {
			return;
		}
		if (node.left == null &&node.right == null) {
			psf.add(node);
			if (l - k >= 0 &&psf.size() > 0 &&visited.contains(psf.get(l - k)) == false) {
				count++;
				visited.add(psf.get(l - k));
			}
			psf.remove(psf.size() - 1);
			return;
		}
		psf.add(node);
		helper(node.left, k, l + 1, psf, visited);
		helper(node.right, k, l + 1, psf, visited);
		psf.remove(psf.size() - 1);
	}
```


what?

Given root of a Binary Tree and a number K. find the count of nodes that are at a distance K from a leaf nodes.

how?

ex!



k=2

For 9 → psf [1, 2, 4, 9] L-K = 3-2 = 1

So set = <2> count = 1

5 → psf [1, 2, 5] L-K = 2-2 = 0

set <2,1> count = 2.

8 → PSF [1, 3, 6, 8] L-K = 3-2 = 1

Set = <2,1,3> count = 3

We will maintain a path so far (psf) that contains all nodes till leaf, from that list (L-K) (L is current level) is the node at distance k from leaf. To avoid duplication Hashset is maintained.

Ans = 3

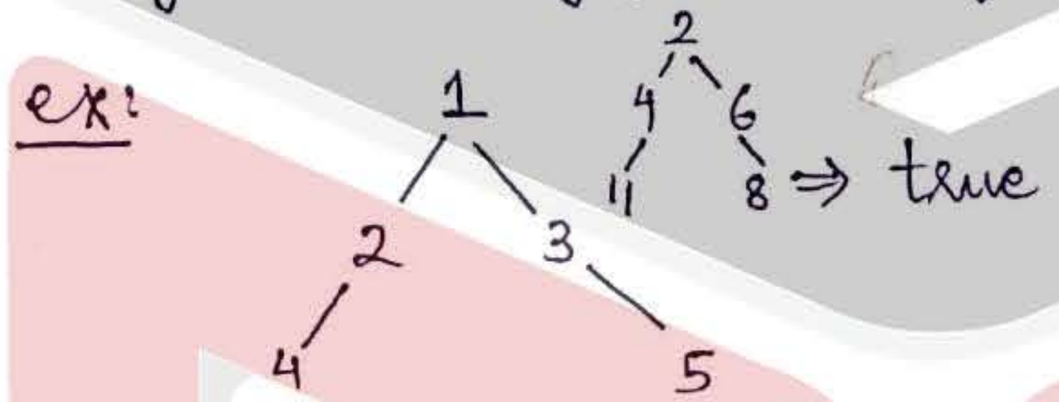
TWO MIRROR TREES

```
public static booleanareMirror(TreeNode a, TreeNode b) {
        if (a == null && b == null) {
                return true;
        }
        if (a == null || b == null) {
                return false;
        }
        booleanlr = areMirror(a.left, b.right);
        booleanrr = areMirror(a.right, b.left);
        return a.val == b.val&&lr&&rr;
    }
```

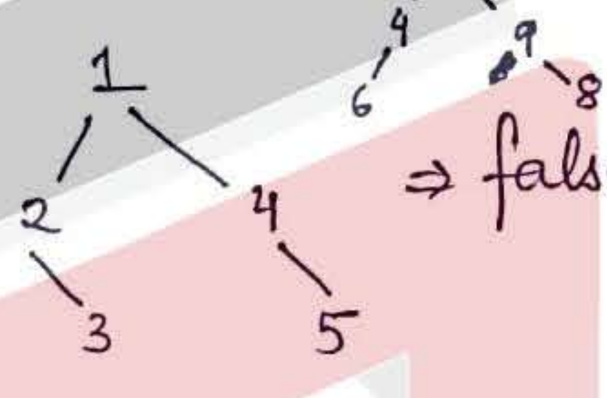
What? Given two trees, find if they are mirror of each other.

* A tree is a mirror tree, if left structure is like right & right is as left.

ex:



⇒ true

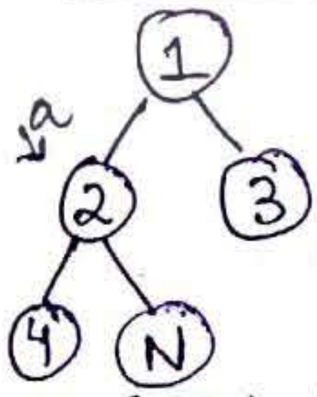


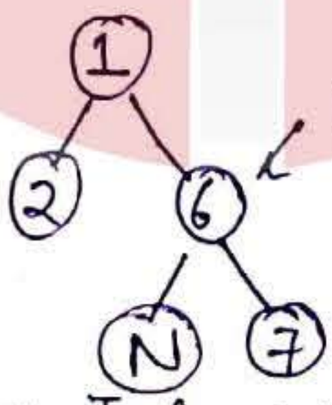
⇒ false.

How? * A leaf node is a mirror subtree always (just check data)

* For any node in a, if left is null and for corresponding node b if right is null, return true, but if only one is null, return false.

ex: Case 1:

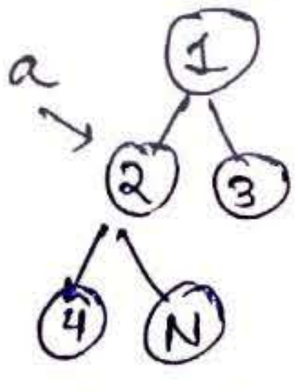


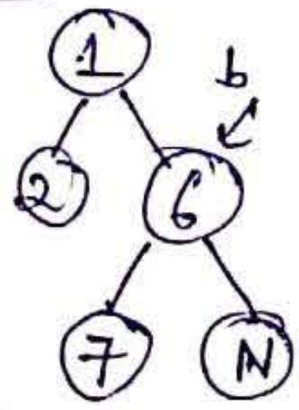


2. left is not null $\xleftrightarrow{T}$ 6. right is not null

2. right is null $\xleftrightarrow[T]{}$ 6. left is null.

Case 2:





2. right is null, 6. left is not null

⇒ return false.

* If left subtree returns true and right subtree returns true, then check if a.data == b.data, return true.

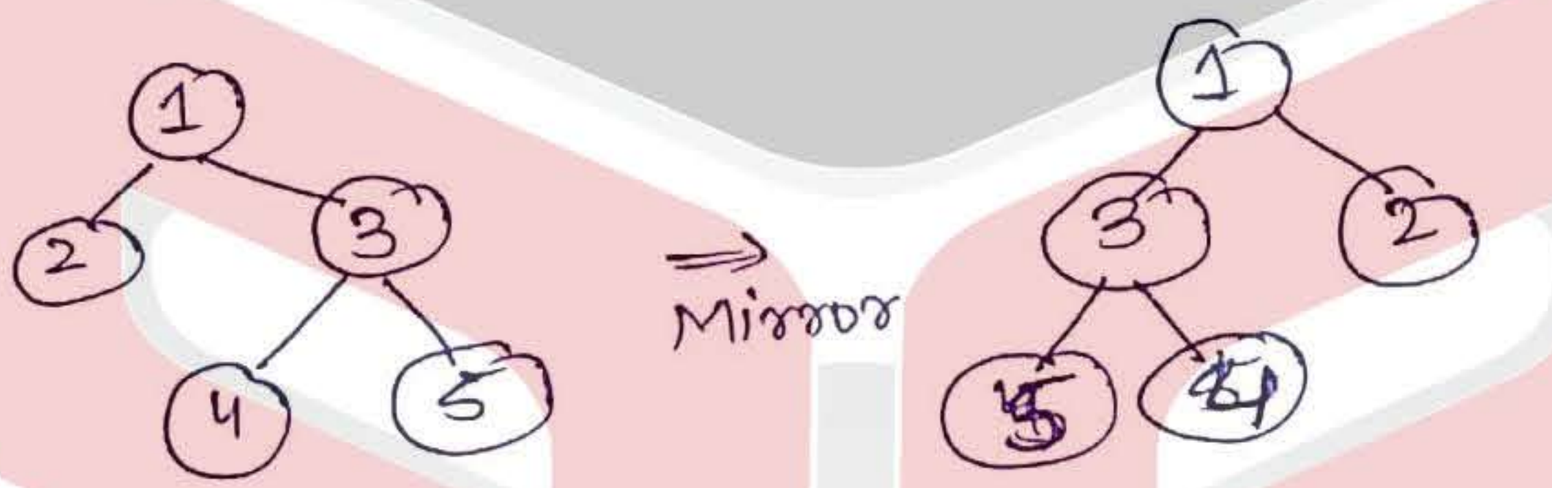
MIRROR TREE

```
public static void mirror(TreeNode node) {
        if (node == null) {
            return;
        }
        mirror(node.left);
        mirror(node.right);

        TreeNode temp = node.left;
        node.left = node.right;
        node.right = temp;
    }
```

<u>what?</u>

Given root of binary Tree. Convert the given Tree to mirror Tree. Mirror Tree is a tree which has exact opposite structure to current tree ie left subtree becomes right and vice versa.

ex!



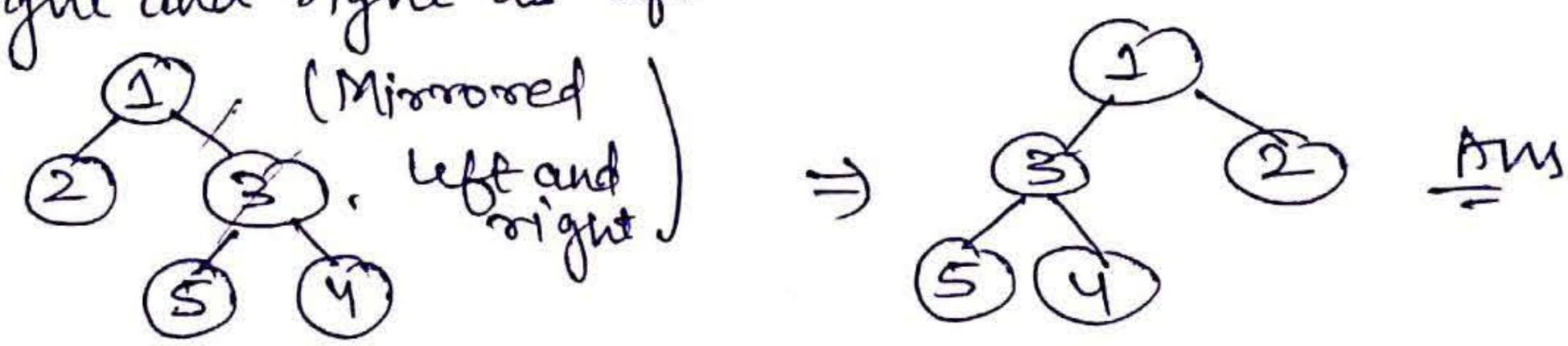
<u>How?</u>

On keeping faith that left and right subtree will mirror themselves. We switch the subtrees as per mirror tree condition.

Firstly make left call that will reccursively mirror left subtree.

Second make right call that will reccursively mirror right subtree

Now swap left and right by making left node as right and right as left.



Ans

FOLDABLE BINARY TREE

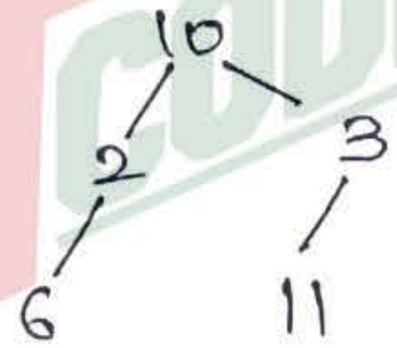
```
public booleanisFoldable(TreeNode node) {
                        mirror(node.left);
                        return checkStructureSimilar(node.left, node.right);
                }

                public booleancheckStructureSimilar(TreeNode n1, TreeNode n2) {
                        if (n1 == null && n2 == null) {
                                return true;
                        }
                        if (n1 != null && n2 != null) {
                                return checkStructureSimilar(n1.left, n2.left)
                                &&checkStructureSimilar(n1.right, n2.right);
                        } else {
                                return false;
                        }
}
public void mirror(TreeNode node) {
        if (node == null) {
                return;
        }
        mirror(node.left);
        mirror(node.right);

        TreeNode temp = node.left;
        node.left = node.right;
        node.right = temp;
}
```

<u>What?</u>
Given a binary tree check if it can be folded or not
A tree can be folded if its left and right subtree has same structure.

<u>ex:</u>

This tree is foldable

ex:

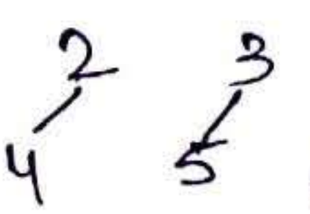


This tree is not foldable.

<u>How?</u>
To check if tree is foldable or not we will take mirror of left subtree and now compare structure of left and right subtree.
We are taking mirror because left and right structure can be similar when after mirror they are same.

<u>eg:</u>

Now comparing



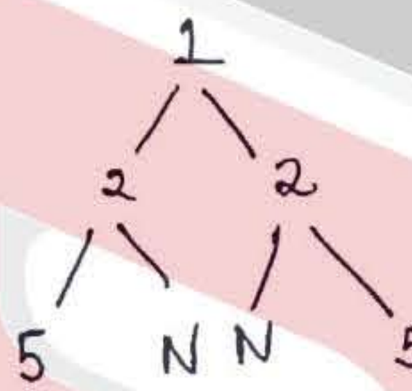
Same Structure

SYMMETRIC TREE

```
public static booleanisSymmetric(TreeNode root) {
        return ishalfmirror(root, root);
    }

    public static booleanishalfmirror(TreeNoderootl, TreeNoderootr) {
        if (rootl == null &&rootr == null) {
            return true;
        }
        if (rootl == null || rootr == null) {
            return false;
        }
        booleanlr = ishalfmirror(rootl.left, rootr.right);
        booleanrr = ishalfmirror(rootl.right, rootr.left);
        return lr&&rr&&rootl.val == rootr.val;
    }
```

What? Tell if a tree is symmetric or not, a tree is symmetric if it is a mirror image of itself or not.

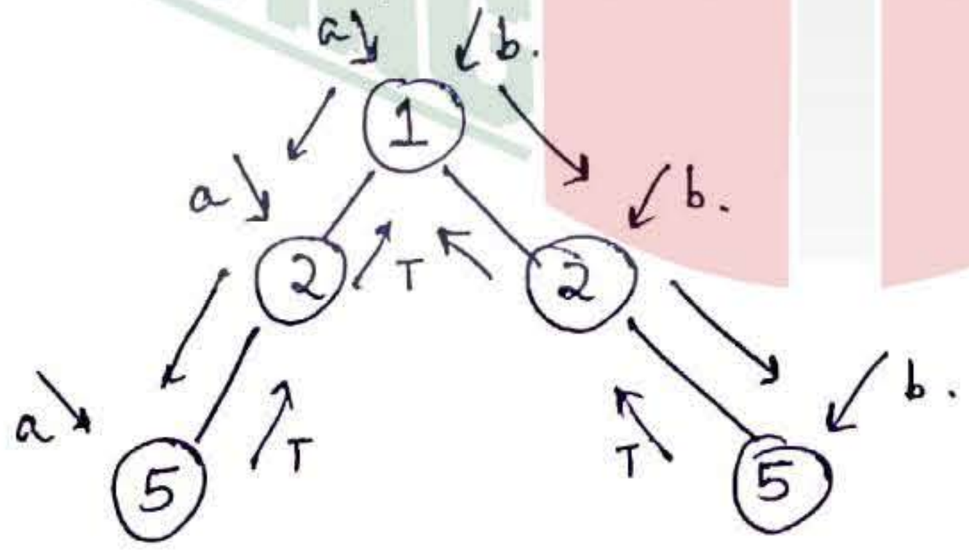
ex:



⇒ True

How? * Break root.left and root.right as 2 subtrees and check if left subtree is mirror of right subtree, then the tree is symmetric.

ex:



⇒ True

(Virtually we consider left and right subtree as two trees and then check if both are mirror of each other.)

ANCESTOR IN BINARY TREE

```
public static booleanprintAncestors(TreeNode node, int x) {
        if (node == null) {
                return false;
        }
        if (node.val == x) {
                return true;
        }
        if (printAncestors(node.left, x) == true) {
                System.out.print(node.val + " ");
                return true;
        }
        if (printAncestors(node.right, x) == true) {
                System.out.print(node.val + " ");
                return true;
        }
        return false;
}
```

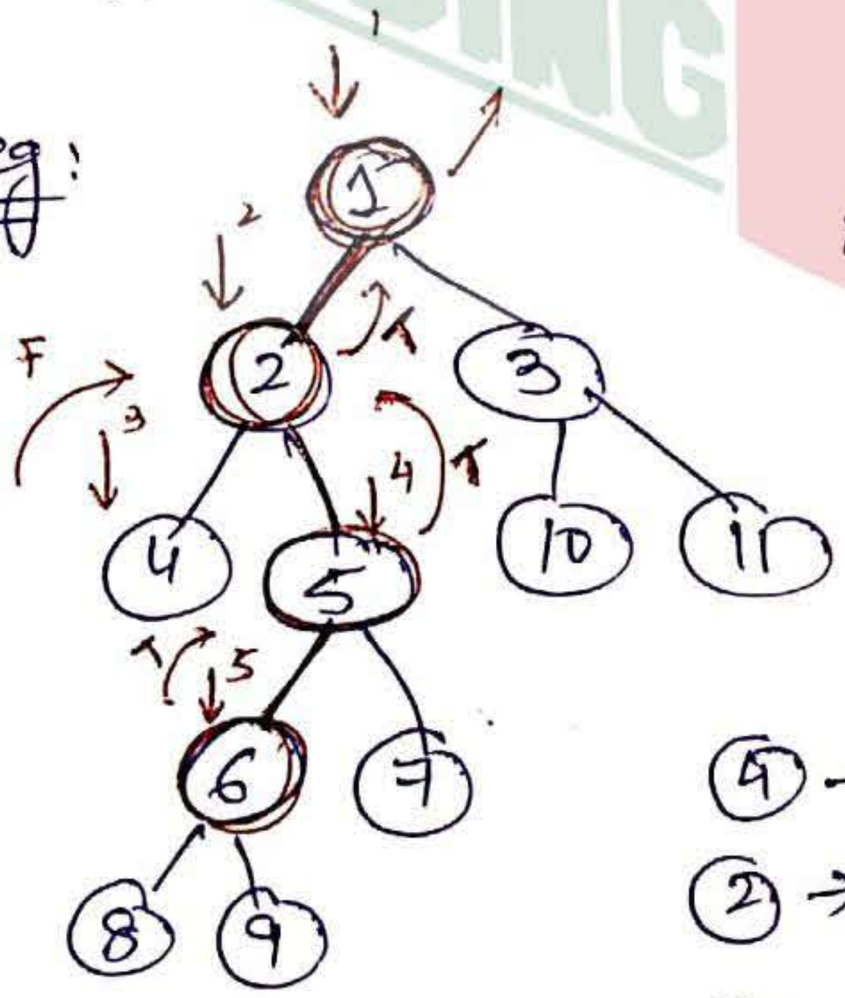
What?

Given root of a Binary Tree and a target key. Print all the ancestors of the key in the given Binary Tree.

How?

* We will use find code as soon as the node is found we will return true.
* Since we are returning True as soonas ~~first~~ value is found. All the ancestors will be visited while returning.

eg:



Key = 6.

Ans = 5 2 1

(4) → Print 5

(2) → Print 2.

(1) → Print (1).

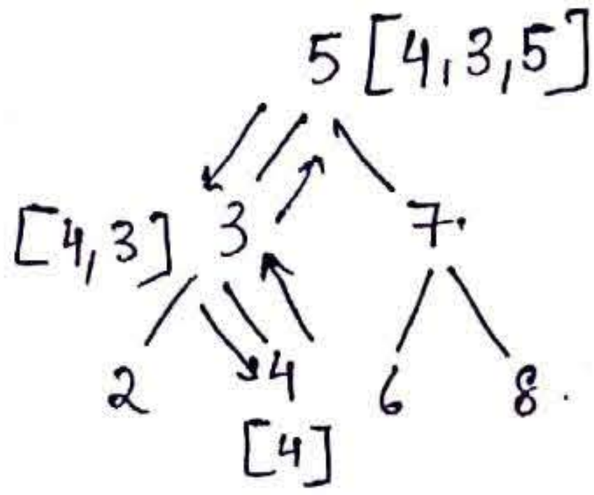
PTH COMMON ANCESTOR IN BST

```
public int PthAncestor(TreeNode root, int x, int y, int p, ArrayList<Integer> list) {
                // Write your code here
                ArrayList<Integer>findx = find(root, x);
                ArrayList<Integer>findy = find(root, y);
                int i = findx.size() - 1;
                int j = findy.size() - 1;
                for (; i>= 0 && j >= 0; i--, j--) {
                        int v1 = findx.get(i);
                        int v2 = findy.get(j);
                        if (v1 != v2) {
                                break;
                        }
                }
                if (i + p <findx.size()) {
                        return findx.get(i + p);
                }
                return -1;
        }

        public static ArrayList<Integer> find(TreeNode root, int val) {
                if (root == null) {
                        return new ArrayList<>();
                }
                if (root.left == null &&root.right == null) {
                        ArrayList<Integer> list = new ArrayList<>();
                        if (root.data == val) {
                                list.add(root.data);
                        }
                        return list;
                }
                ArrayList<Integer>lres = find(root.left, val);
                if (lres.size() > 0) {
                        lres.add(root.data);
                        return lres;
                }
                ArrayList<Integer>rres = find(root.right, val);
                if (rres.size() > 0) {
                        rres.add(root.data);
                        return rres;
                }
                ArrayList<Integer> list = new ArrayList<>();
                if (root.data == val) {
                        list.add(root.data);
                }
                return list;
        }
```

What? Given a BST, and two node values x and y, find pth ($p >= 1$) common ancestor of two nodes x and y in BST. 1'st common ancestor is the lowest common ancestor.

How? * Keep two arraylists that store leaf to root paths in BST.
* Use find function to store path in BST.

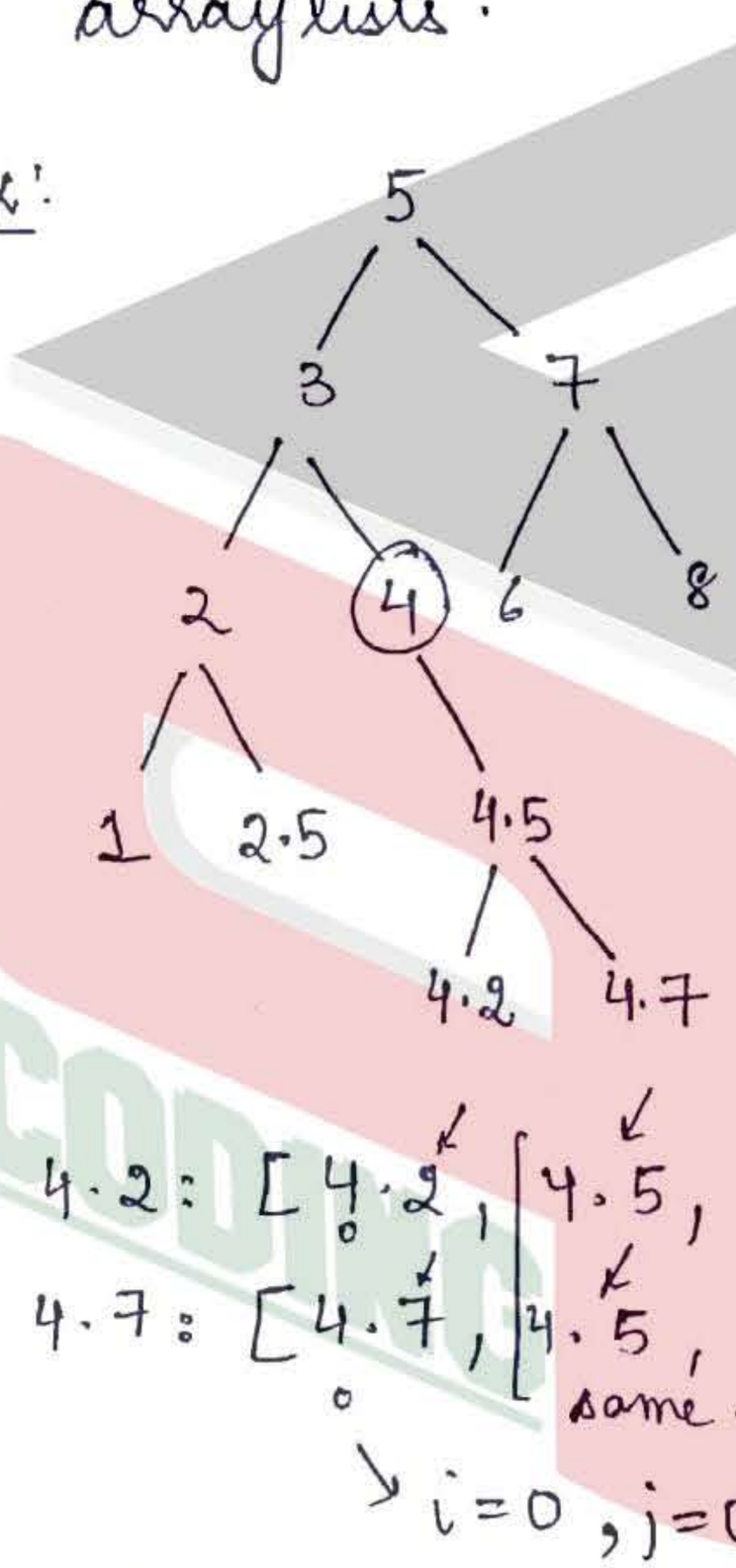
ex:



Find 4: [4,3,5]

* Now check the pth common ancestor using two arraylists.

ex:



$x = 4.2$, $y = 4.7$, $p = 2$

4.2: [4.2, 4.5, 4, 3, 5]

4.7: [4.7, 4.5, 4, 3, 5]

* If lists are found, now keep i,j at index where we get first unequal value.

4.2: [4.2, 4.5, 4, 3, 5]
4.7: [4.7, 4.5, 4, 3, 5]
same →
↳ i=0, j=0

Return i+p ⇒ (0+2) = 2.

⇒ ans = 4

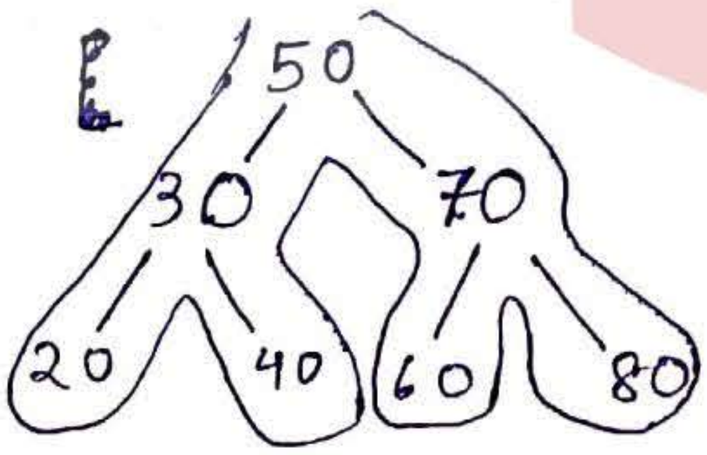
4 is the 2nd common ancestor.

PREDECESSOR AND SUCCESSOR

```
static class Result {
        TreeNode pre = null;
        TreeNodesucc = null;
    }
    public static void findPreSuc(TreeNode root, Result r, int key) {
        if (root == null) {
            return;
        }
        findPreSuc(root.left, r, key);
        if (root.val< key) {
            r.pre = root;
        }
        if (root.val> key) {
            if (r.succ == null) {
                r.succ = root;
            }
        }
        findPreSuc(root.right, r, key);

}
```

What? Find inorder predecessor and successor of a node in BST.

How? * If values are $< k$, update predecessor everytime.
* Last predecessor update will be the value just smaller than k.
* Update successor only one time when value of node $> k$.
* Since BST is sorted in inorder, comparisons are done in inorder.

ex: 

$k = 65$.

Predecessor = ~~null~~ ~~20~~ ~~30~~ ~~40~~ ~~50~~ 60

Successor = null 70

ans = [60, 70]

KTH SMALLEST ELEMENT IN BST

```
public static int kthSmallest(TreeNode root, int k) {
        l = 1;
        small = 0;
        helper(root, k);
        int res = small;
        return res;
    }

    static int l = 1;
    static int small = 0;
    public static void helper(TreeNode node, int k) {
        if (node == null) {
            return;
        }
        helper(node.left, k);
        if (l == k) {
            small = node.val;
            l = k + 1;
            return;
        } else {
            l++;
        }
        helper(node.right, k);
}
```

What?

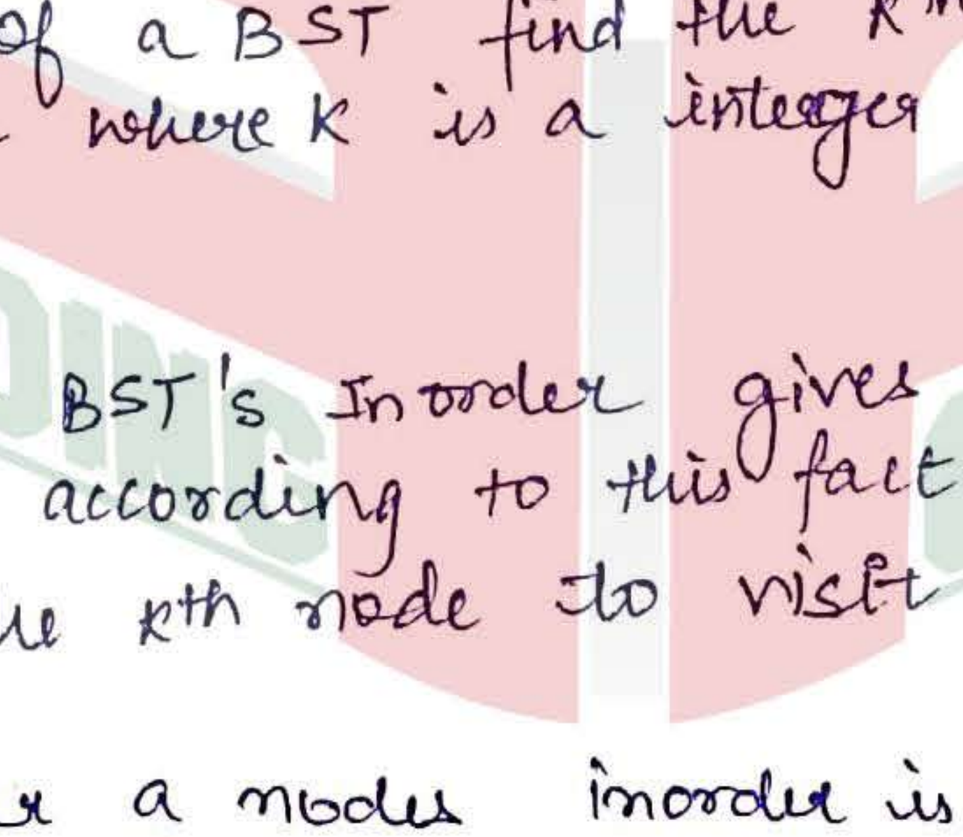
Given root of a BST find the k^{th} smallest number in the tree where K is a integer.

How?

*We know that BST's Inorder gives elements in sorted order. So according to this fact k^{th} smallest element will be the k^{th} node to visit inorder..

* So whenever a nodes inorder is executed variable L is incremented. when L==K then that is k^{th} smallest number.

eg:



K = 3.

Ans = 40

KTH LARGEST ELEMENT IN BST

```
public static int kthLargest(TreeNode root, int k) {
        // write your code here.
        helper(root, k);
        l = 1;
        int res = lar;
        lar = 0;
        return res;
    }

    static int l = 1;
    static int lar = 0;
    public static void helper(TreeNode node, int k) {
        if (node == null) {
            return;
        }
        helper(node.right, k);
        if (l == k) {
            lar = node.val;
            l = k + 1;
            return;
        } else {
            l++;
        }
        helper(node.left, k);
    }
```

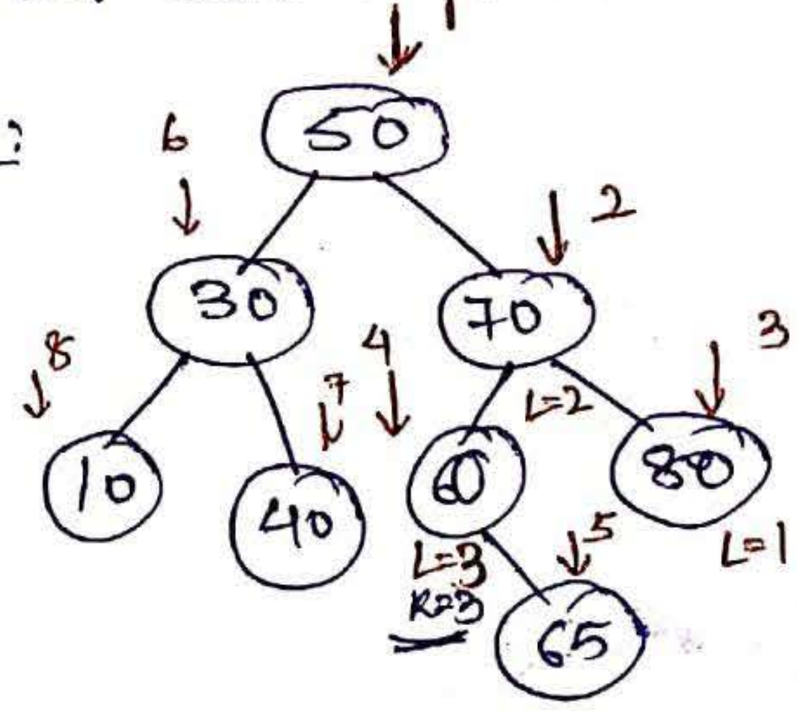
what?

Given root of a BST. Find the kth largest number in the tree.

How?

* We know the BST's Inorder gives elements in sorted and Reverse Inorder gives decreasing sorted order. so by following Reverse sorted order kth element visited in Inorder is kth largest.

eg:



k=3

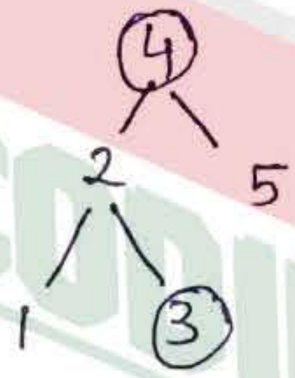
Ans = 60

CLOSEST BST VALUE TREE

```
public List<Integer>closestKValues(TreeNode root, double target, int k) {
        // Write your code here.
        List<Integer>ans = new ArrayList<>();
        LinkedList<Integer> list = new LinkedList<>();
        closestK(root, target, k, list);
        while (list.size() != 0) {
                ans.add(list.removeLast());
        }
        return ans;

    }

public void closestK(TreeNode root, double target, int k, LinkedList<Integer> list) {
        if (root == null) {
                return;
        }
        closestK(root.left, target, k, list);
        if (list.size() < k) {
                list.addLast(root.data);
        } else if (Math.abs(list.getFirst() - target) >Math.abs(root.data - target)) {
                list.removeFirst();
                list.addLast(root.data);
        }
        closestK(root.right, target, k, list);
}
```


what? Given a BST root and a target value, find k values in BST closest to target. k will be given.

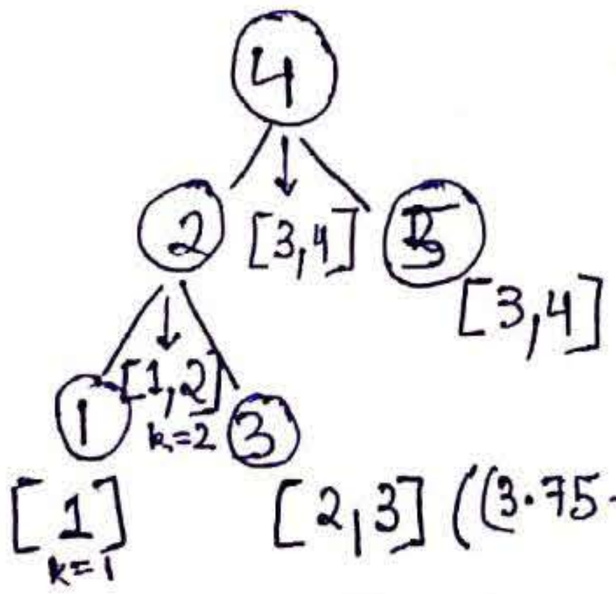
ex:



target = 3.75
k = 2
ans = [3, 4]

How?
* Keep a queue / linked list.
* While (list.size() < k), add every node in queue (b'coz we need k near nodes).
* When k nodes are present in queue, if current node has less absolute difference from target, remove First and add current in Last.
* This works because BST is sorted in inorder.

ex:



target = 3.75
k = 2.
ans = [3,4].

((3.75-2) < (3.75-1))

MINIMUM DEPTH OF BINARY TREE

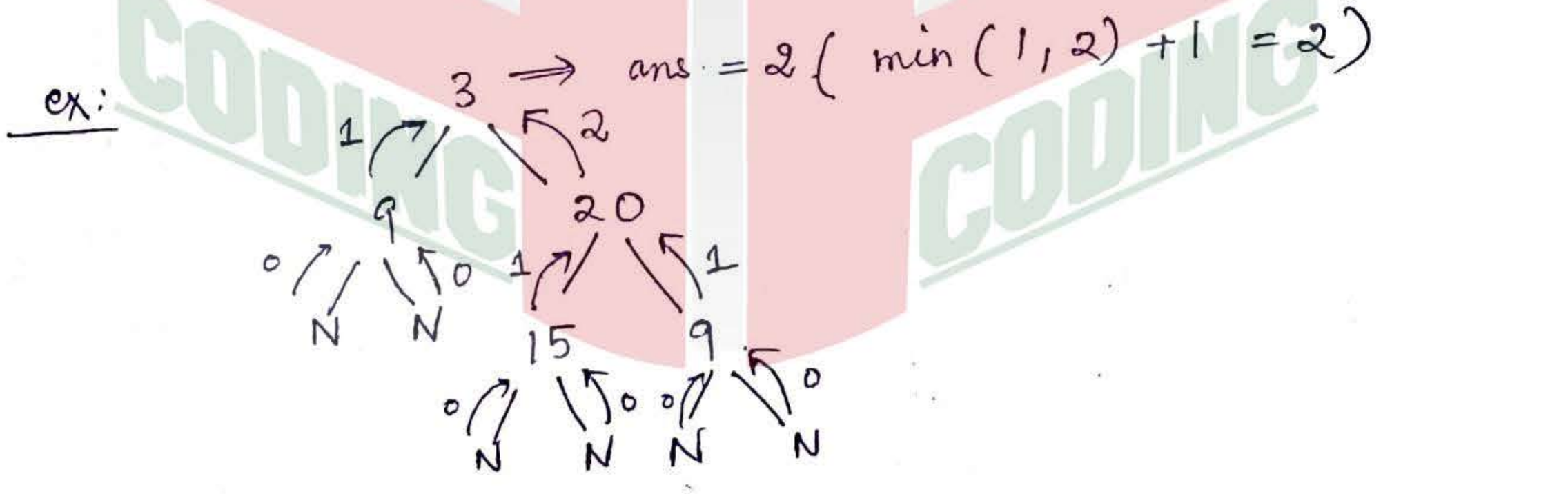
```
public static int minDepth(TreeNode root) {
        if (root == null)
            return 0;
        int lht = minDepth(root.left);
        int rht = minDepth(root.right);
        if (root.left != null &&root.right == null)
            return lht + 1;
        else if (root.right != null &&root.left == null)
            return rht + 1;
        else
            return Math.min(lht, rht) + 1;
    }
```

What? Find minimum depth of tree, where minimum depth is number of nodes along the shortest path from the root node down to nearest leaf node.

How? * Leaf returns minimum depth = 0.

* Each node gets two depth values, from left and right, if left is null, return rightdepth+1, if right is null, return left depth + 1, else return min (left depth, right depth) + 1.

ex:



AVERAGE OF LEVELS IN BINARY TREE

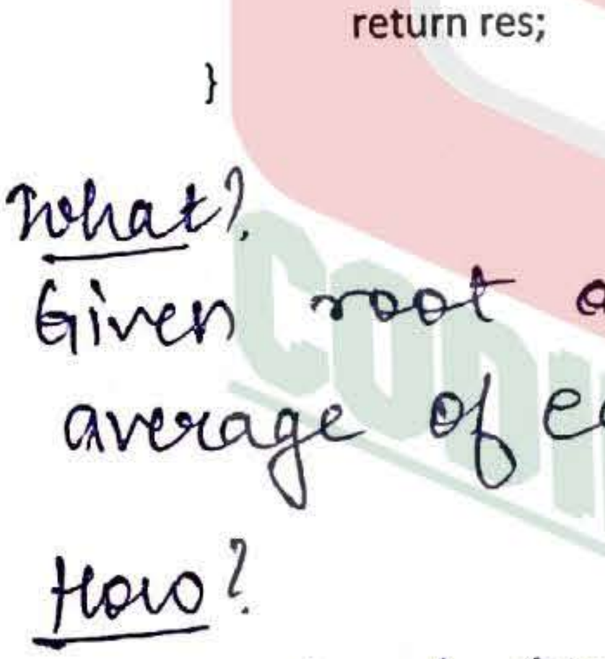
```
public static List<Double>averageOfLevels(TreeNode root) {
        List<Double> res = new ArrayList<Double>();
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        queue.add(null);
        int count = 0;
        long sum = 0;
        while (queue.size() > 0) {
                TreeNode rm = queue.removeFirst();
                if (rm == null) {
        res.add(sum * 1.0 / count);
        sum = 0;
        count = 0;
        if (queue.size() != 0) {
                queue.addLast(rm);
        }
} else {
        sum += rm.val;
        count++;
        if (rm.left != null) {
                queue.add(rm.left);
        }
        if (rm.right != null) {
                queue.add(rm.right);
        }
}
        return res;
}
```


What?

Given root of a Binary Tree, return a list containg average of each level of the Tree.

How?

Apply level order and maintain count and sum of all nodes at each level.



sum=10 count=1 Avg=10/1=10

sum=50 count=2 Avg=50/2=25

sum=150 count=3 Avg=150/3=50

sum=150 count=2 Avg=75

Ans [10, 25, 50, 75]

BINARY TREE K LEVEL SUM

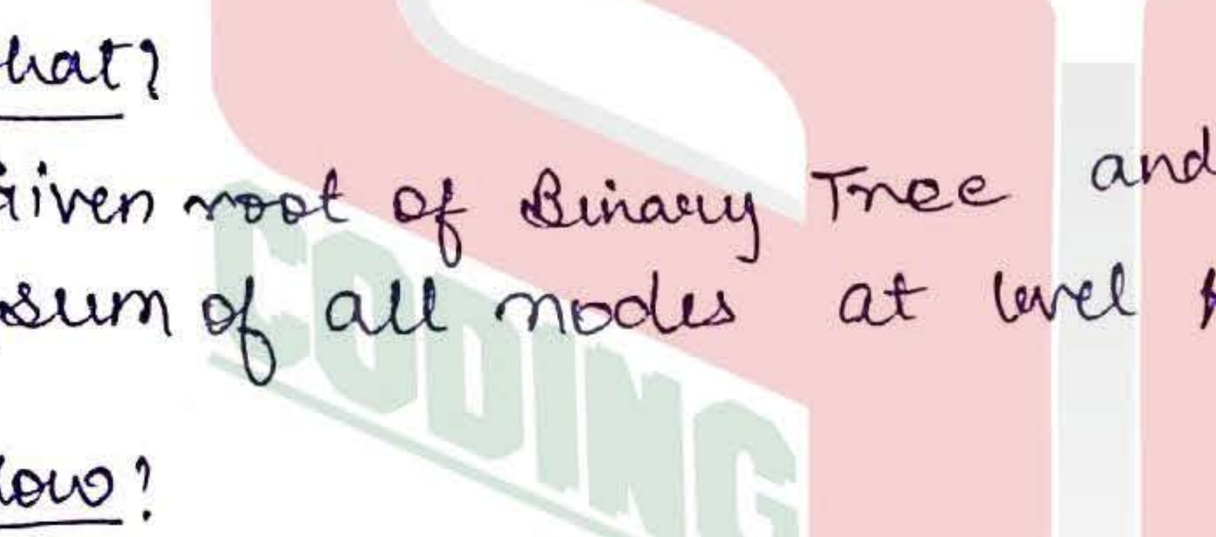
```
public static void printKlevelOrder(TreeNode root, int k) {
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        queue.add(null);
        int level = 0;
        long sum = 0;
        while (queue.size() > 0) {
            TreeNode rm = queue.removeFirst();
            if (rm == null) {
                if (level == k) {
                    System.out.println(sum);
                    break;
                }
                sum = 0;
                level++;
                if (queue.size() != 0) {
                    queue.addLast(rm);
                }
            } else {
                sum += rm.val;
                if (rm.left != null) {
                    queue.add(rm.left);
                }
                if (rm.right != null) {
                    queue.add(rm.right);
                }
            }
        }
```

What?

Given root of Binary Tree and an Integer K. Give the sum of all nodes at level K.

How?

We will Apply level order when level (current = K) will break the loop. and print Sum.



Ans = 22

MAXIMUM NODE LEVEL

```
public int maxNodeLevel(Node node) {
        // Write your code here.
        LinkedList<Node> queue = new LinkedList<>();
        queue.add(node);
        queue.add(null);
        int level = 0;
        int count = 0;
        int maxlevel = 0;
        int maxCount = 0;
        while (queue.size() > 0) {
            Node rm = queue.removeFirst();
            if (rm == null) {
                maxlevel = maxCount< count ? level : maxlevel;
                maxCount = Math.max(count, maxCount);
                level++;
                count = 0;
                if (queue.size() != 0) {
                    queue.addLast(rm);
                }
            } else {
                count++;
                if (rm.left != null) {
                    queue.add(rm.left);
                }
                if (rm.right != null) {
                    queue.add(rm.right);
                }
            }
        }
        return maxlevel;
    }
```

width of binary Tree is also no. of max nodes at a level.

Ans = 4 (same method)

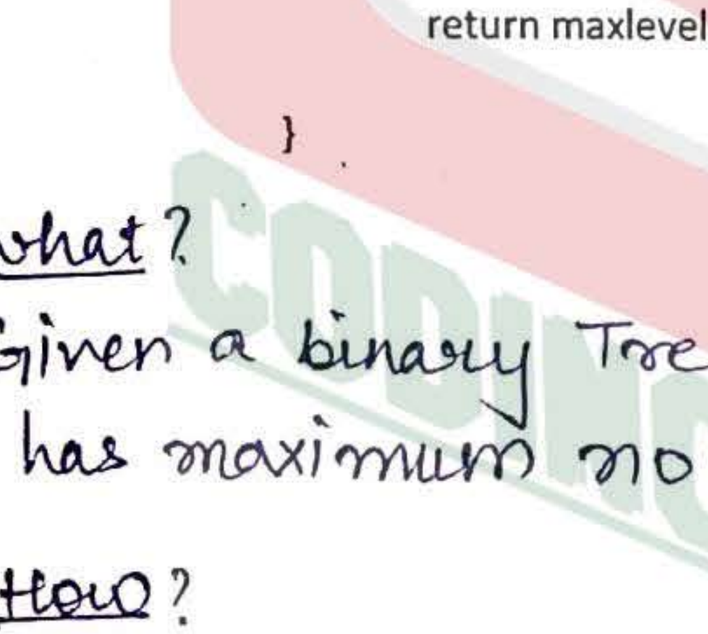
what?

Given a binary Tree. Find the level of Tree which has maximum no. of nodes.

How?

We will use level order to calculate this level with maximum nodes as max level.

eg:



Ans = 1 level 1

0 → 1 | null → count=1, level=0, maxlevel=0

1 → 2 | 3 | null → count=2, level=1, maxlevel=1

2 → 4 | 5 | 6 | 9 | null → count=4, level=2, maxlevel=2

3 → 7 | 8 | 10 | null → count=3, level=3

maxlevel = 2

LEAF UNDER BUDGET

```
public static int getCount(TreeNode node, int bud) {
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(node);
        queue.add(null);
        int level = 1;
        int count = 0;
        while (queue.size() > 0) {
            TreeNode rm = queue.removeFirst();
            if (rm == null) {
                level++;
                if (queue.size() != 0) {
                    queue.addLast(rm);
                }
            } else {
                if (rm.left == null &&rm.right == null && bud - level >= 0) {
                    bud -= level;
                    count++;
                    continue;
                }
                if (rm.left != null) {
                    queue.add(rm.left);
                }
                if (rm.right != null) {
                    queue.add(rm.right);
                }
            }
        }
        return count;
    }
```

what?

Given root of a binary Tree and a number representing budget. The cost of visiting a leaf nodes is equal to the level of that leaf node. Find the maximum number of leaf nodes that can be visited in the given budget.

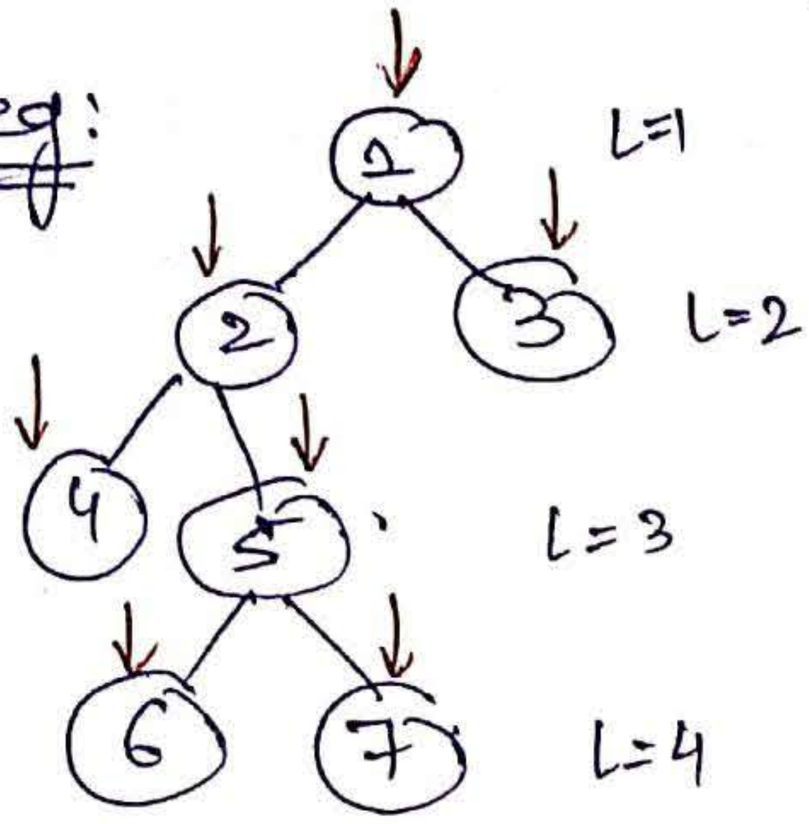
How?

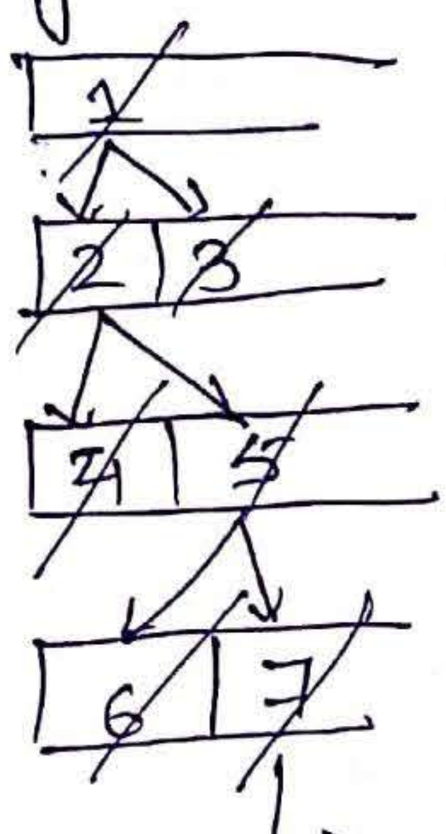
Since we need to maximise the no. of nodes we will take all nodes of less cost that is why we choose leaves level by level.

By choosing leaves level by level we are giving opportunity to least cost leafs first.

Budget = 12

eg:





3 is a leaf bud -= 2
count=1 bud = 10

count = 2 bud = 10-3
= 7

count = 3 bud = 7-4
= 3

leaf but bud is exhausted.

Ans = 3

LONGEST CONSECUTIVE SEQUENCE IN BINARY TREE

```
public int longestConsecutive(TreeNode root) {
                    // add code here.
                    maxlen = 0;
                    helper(root, null, 0);
                    if (maxlen == 0) {
                            return -1;
                    } else {
                            return maxlen + 1;
                    }
            }

public static int maxlen = 0;
public void helper(TreeNode node, TreeNode par, int cl) {
        if (node == null) {
                return;
        }
        cl = par != null &&node.key - 1 == par.key ? cl + 1 : 0;
        maxlen = Math.max(cl, maxlen);
        helper(node.left, node, cl);
        helper(node.right, node, cl);

}
```

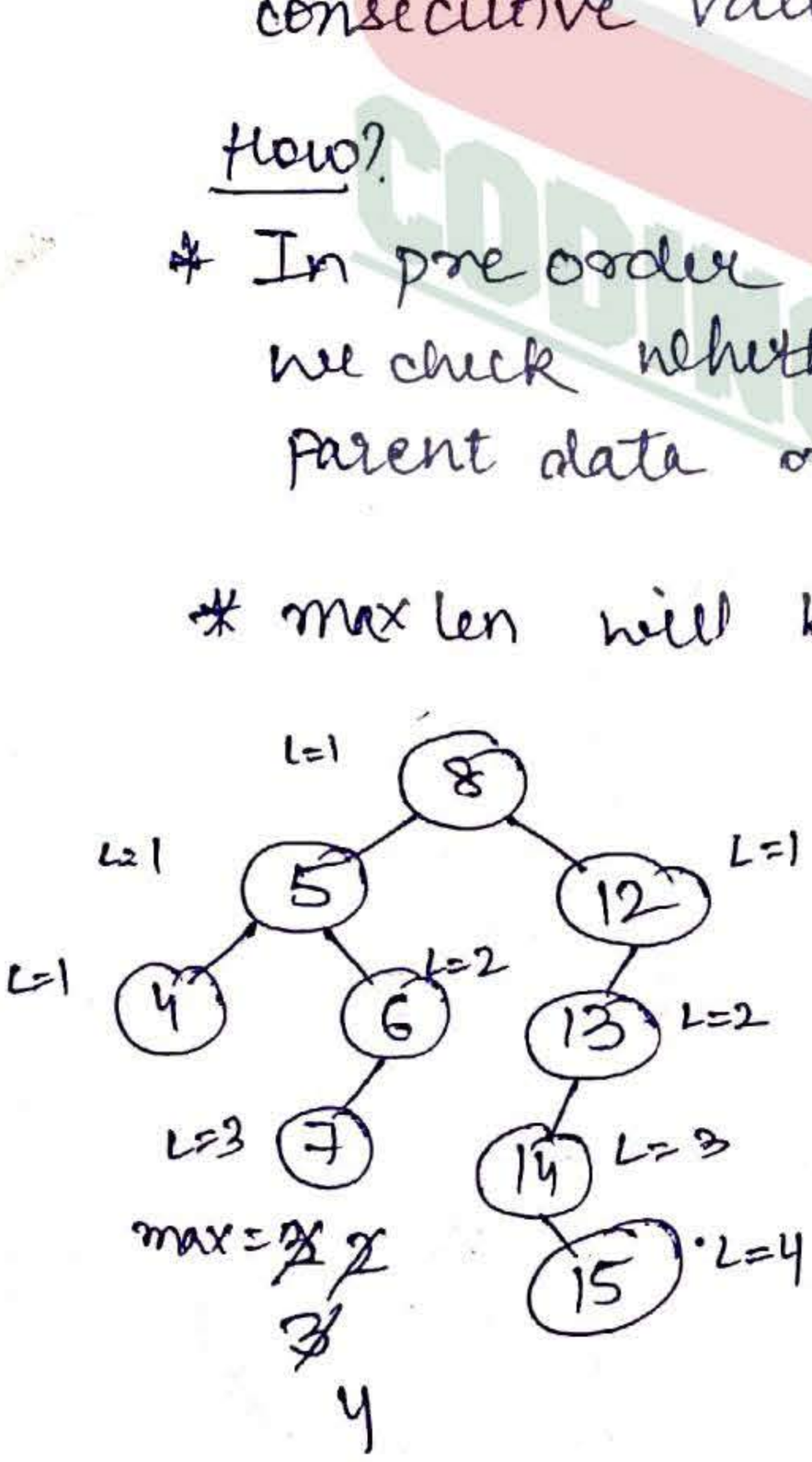
<u>What?</u>

Given root of a Binary Tree find the length of the longest path which comprises of nodes with consecutive values in increasing order.

<u>How?</u>

* In preorder of the traversal. (Before the calls) we check whether node.data is consecutive to parent data or not if yes then cn is incremented

* max len will be max of all such possible sequence.



Ans = 4

12 → 13 → 14 → 15

PRINT LEAF NODES FROM PRE OF BST

```
public static void print(int[] arr) {
        Stack<Integer>st = new Stack<>();
        for (int i = 1; i<arr.length; i++) {
            if (arr[i - 1] >arr[i]) {
                st.push(arr[i - 1]);
            } else {
                int count = 0;
                while (st.size() > 0 &&st.peek() <arr[i]) {
                    st.pop();
                    count++;
                }
                if (count != 0)
                    System.out.print(arr[i - 1] + " ");
            }
        }
        System.out.print(arr[arr.length - 1] + " ");
    }
```

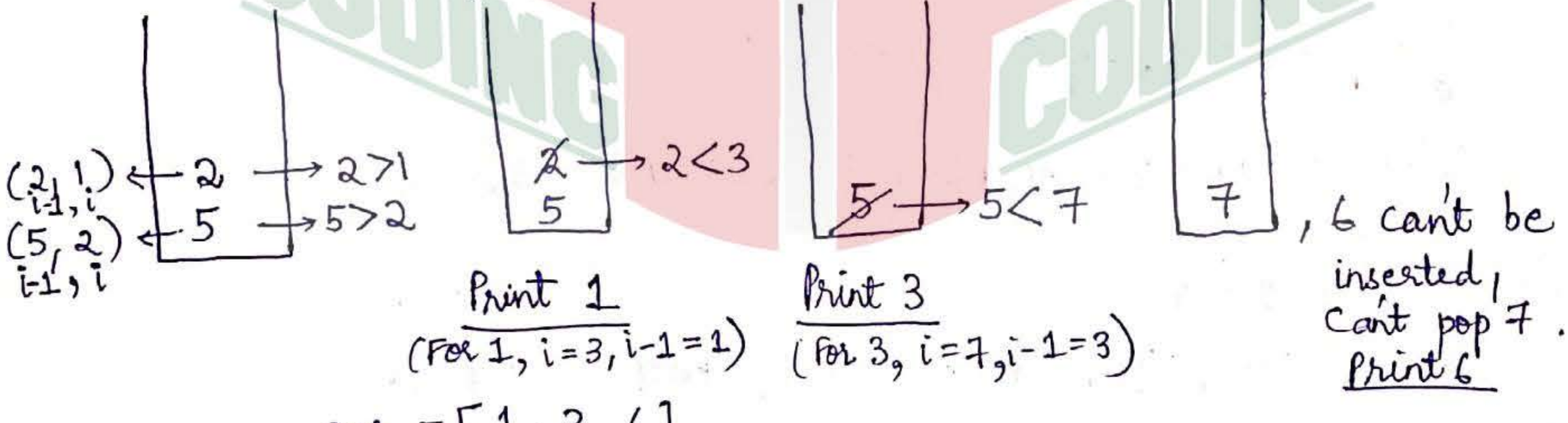
What? Given preorder traversal of BST, print leaf nodes without building the tree.

ex: [890, 325, 290, 530, 965]

ans = [290, 530, 965].

How? * We use a stack to do this.

ex: [5, 2, 1, 3, 7, 6]



Print 1
(For 1, i=3, i-1=1)

Print 3
(For 3, i=7, i-1=3)

Print 6

ans = [1, 3, 6].

```
i=1 to end.
  if (a[i-1] ≥ a[i]
      push(i-1) (sorted order is coming, search leaf)
  else.
      //pop (in sorted order)
      // till you can't pop more.
      if count of pop > 0, print a[i-1], it was leaf.
```

Print last of array as last node is always a leaf in preorder.

array: $\overset{0}{[5}, \overset{1}{2}, \overset{2}{1}, \overset{3}{3}, \overset{4}{7}, \overset{5}{6}]$

a) $i=1, i-1=0$

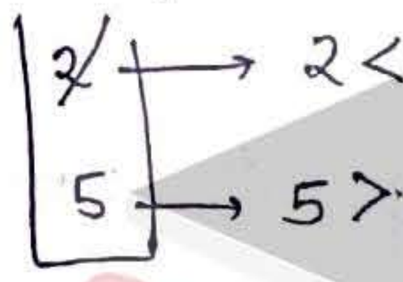
$a[i-1] > a[i]$, i.e; $5>2$, push 5 $\boxed{5}$

b) $i=2, i-1=1$

$a[i-1] > a[i]$, i.e; $2>1$, push 2 $\boxed{\begin{matrix}2\\5\end{matrix}}$

c) $i=3, i-1=2$

$a[i-1] < a[i]$, i.e; $2<3$, pop elements from stack till peek $< a[i]$.

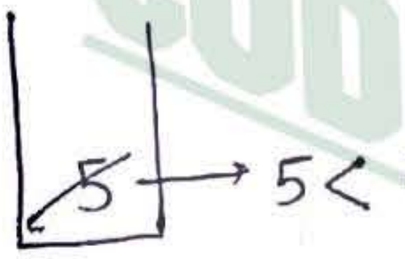


Since count = 1, we popped 1 element, print $a[i-1]$

i.e., print $\boxed{1}$

d) $i=4, i-1=3$

$a[i-1] < a[i]$, i.e; $3<7$, pop elements from stack till peek $< a[i]$.



Since count = 1, print $a[i-1]$, print $\boxed{3}$

e) $i=5, i-1=4$.

$a[i-1] > a[i]$, i.e; $7>6$, push 7 $\boxed{7}$

Print last node b'coz last of preorder is always leaf,

$\therefore$ print $\boxed{6}$.

$\Rightarrow$ answer = $[1, 3, 6]$

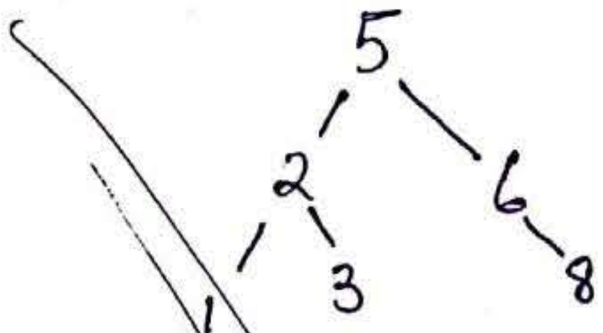
PRINT COMMON NODES IN BST

```
public static void printCommon(TreeNode root1, TreeNode root2) {
        // write your code here.
        Stack<TreeNode> st1 = new Stack<>();
        Stack<TreeNode> st2 = new Stack<>();
        addLeftNodes(st1, root1);
        addLeftNodes(st2, root2);

        while (st1.size() != 0 && st2.size() != 0) {
            if (st1.peek().val == st2.peek().val) {
                System.out.print(st1.peek().val + " ");
                addLeftNodes(st1, st1.pop().right);
                addLeftNodes(st2, st2.pop().right);
            } else if (st1.peek().val< st2.peek().val) {
                addLeftNodes(st1, st1.pop().right);
            } else if (st1.peek().val> st2.peek().val) {
                addLeftNodes(st2, st2.pop().right);
            }
        }
}

public static void addLeftNodes(Stack<TreeNode>st, TreeNode root) {
        while (root != null) {
            st.push(root);
            root = root.left;
        }
}
```

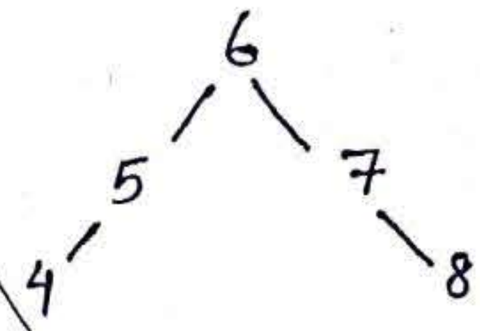
What? Given two BST, print all nodes that have same data in both BST.

How?
* Create two stacks, for both trees.
* Insert root of both trees, and all left nodes of roots.
* If value is same in both (at top), then tops are popped and left nodes of the right of popped nodes are added in respective stacks.
* If top in stack1 is less than top in stack2, then left nodes of right of top of stack1 may have equal nodes to nodes in stack2.
* If top in stack1 is greater than top in stack2, then left nodes of right of top of stack2 may have equal nodes to nodes in stack1.

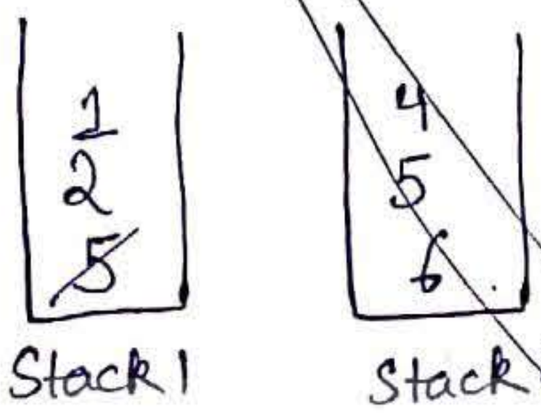
ex: Tree 1 [5, 2, 1, 3, 6, 8]



Tree 2 [6, 5, 4, 7, 8]

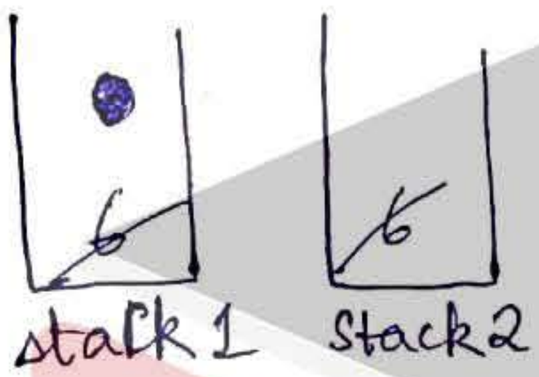


a)



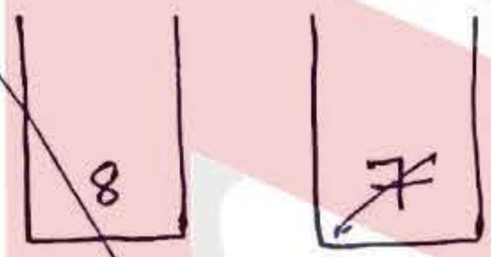
5 < 6, add left nodes of 6 in stack 1.

b)



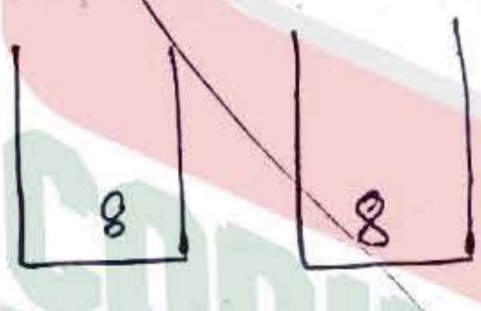
6 = 6, print [6] and add left nodes of 8 in stack 1 and 7 in stack 2.

c)



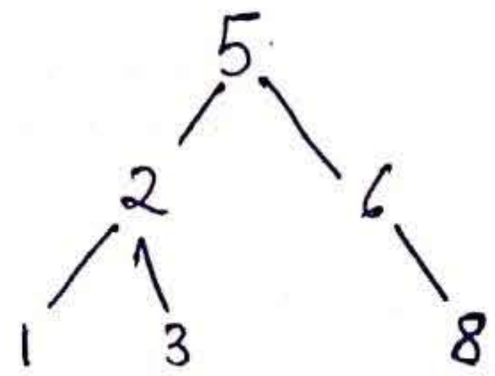
8 > 7, add left nodes of 8 in stack 2.

d)

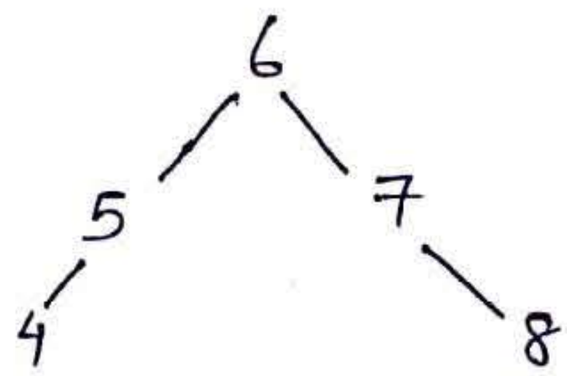


8 = 8, print [8]

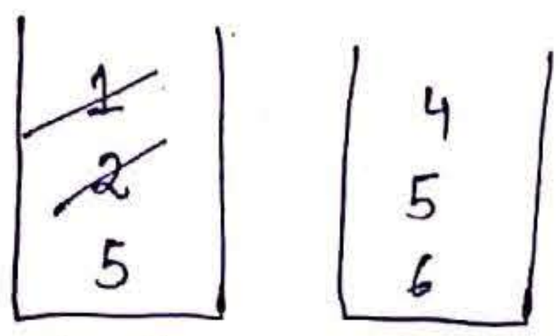
ex: Tree 1 [5, 2, 1, 3, 6, 8]



Tree 2 [6, 5, 4, 7, 8]



a)

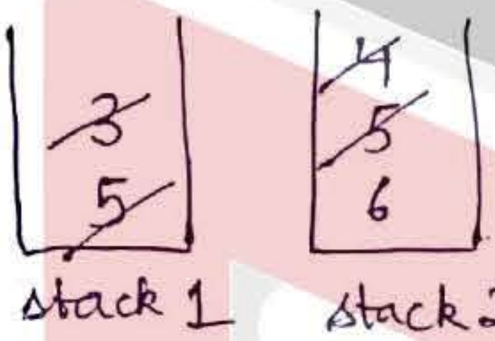


stack 1 stack 2.

→ 1 < 4, add left nodes of 1. right no left nodes of 1 present.

→ 2 < 4, add left nodes of 2. right, i.e. 3.

b)

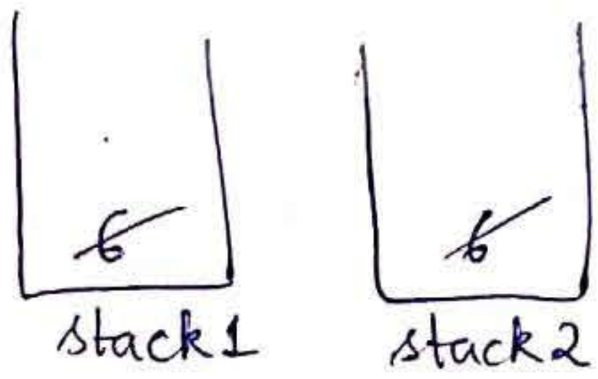


→ 3 < 4, add left nodes of 3. right, 3. right is null.

→ 5 > 4, add left nodes of 4. right, 4. right is null.

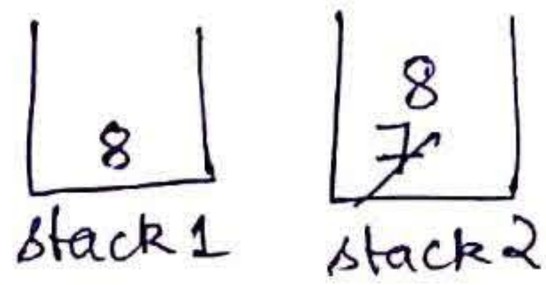
→ 5 = 5, print and add left nodes of 5. right = 6 in stack 1 and 5. right = null in stack 2.

c)



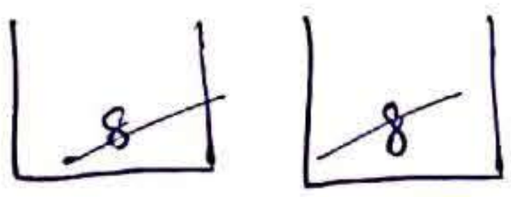
→ 6 = 6, add left nodes of 8 in stack 1 and 7 in stack 2.

d)



→ 8 > 7, add left nodes of 8 in stack 2.

e)



→ 8 = 8, no rights present.

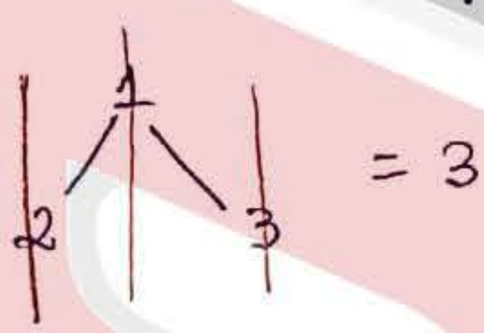
ans = [5, 6, 8]

VERTICAL WIDTH OF BTREE

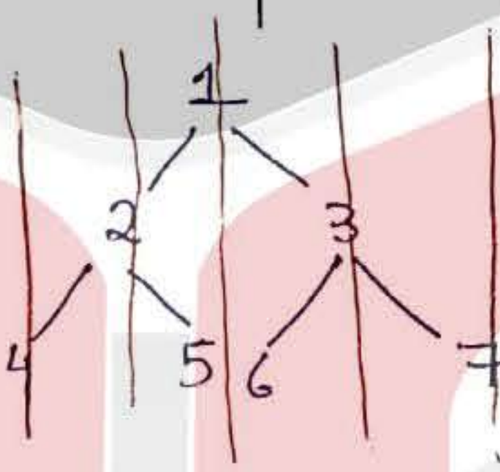
```
HashSet<Integer> set;
public int verticalWidth(TreeNode root) {
        // Write your code here

        set = new HashSet<>();
        vWidth(root, 0);
        return set.size();
    }

    public void vWidth(TreeNode root, int l) {
        if (root == null) {
            return;
        }
        if (!set.contains(l)) {
            set.add(l);
        }
        vWidth(root.left, l - 1);
        vWidth(root.right, l + 1);
    }
```

What? Find vertical width of tree. Width of binary tree is number of vertical paths in that tree.

ex:



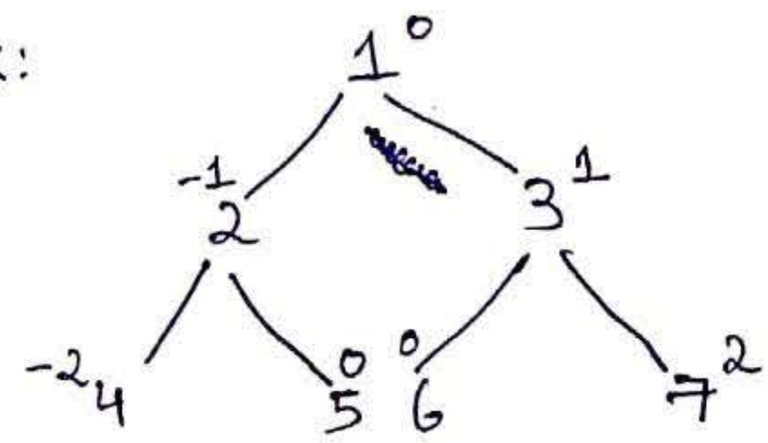
= 3



= 5

How? * Make a HashSet storing widths as integers.
* If you go left, width = width - 1.
* If you go right, width = width + 1.
* If set does not contain current width, add it in set.
* Return size of set.

ex:



set ⇒ | 0 | -1 | -2 | 1 | 2 |

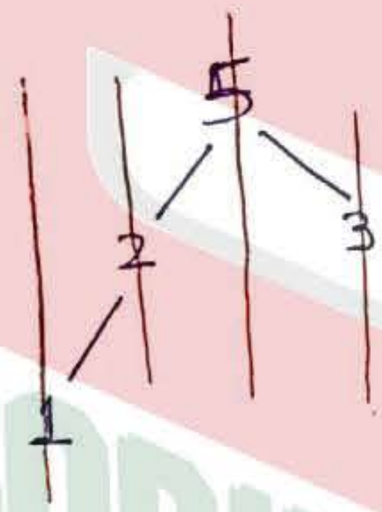
set size = 5.

VERTICAL SUM

```
TreeMap<Integer, Integer> map;

        public void printVertical(TreeNode root) {
                map = new TreeMap<>();
                vSum(root, 0);
                for (int i : map.keySet()) {
                        System.out.print(map.get(i) + " ");
                }
        }

        public void vSum(TreeNode root, int l) {
                if (root == null) {
                        return;
                }
                if (!map.containsKey(l)) {
                        map.put(l, root.data);
                } else {
                        map.put(l, map.get(l) + root.data);
                }
                vSum(root.left, l - 1);
                vSum(root.right, l + 1);
        }
```

What? Find vertical sum of nodes that are in same vertical line.

ex:



$\Rightarrow$ ans = [1, 2, 5, 3]

Use TreeMap for sorted order.

How? * Move on width strategy, make a HashMap < Int vs Int > storing sum for each width.

* If width already exists, add current node's data in it, else create its entry as width vs data.

ex:

$0 \rightarrow 5 \not{8}\ 12$
$-1 \rightarrow 2$
$-2 \rightarrow 1$
$1 \rightarrow 6$
$2 \rightarrow 7$

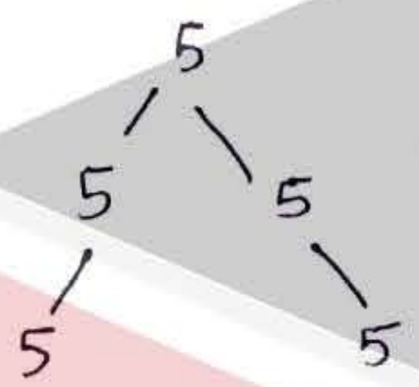
ans $\Rightarrow$ [1, 2, 12, 6, 7]

UNIVALUED BINARY TREE

```
public booleanisUnivalTree(TreeNode root) {
                if(root == null){
                   return true;
                }
          booleanlr = isUnivalTree(root.left);
          booleanrr = isUnivalTree(root.right);
                if(lr&&rr&& (root.left == null || (root.left.data == root.data)) && (root.right == null || (root.right.data
== root.data))){
                   return true;
                }
                return false;
             }
```

What? Return if the tree is a univalued tree or not?
* univalued tree is a tree with all values same.

ex:



⇒ true (all nodes are 5).

How? * If you are a leaf / null, return true.

* Case 1:

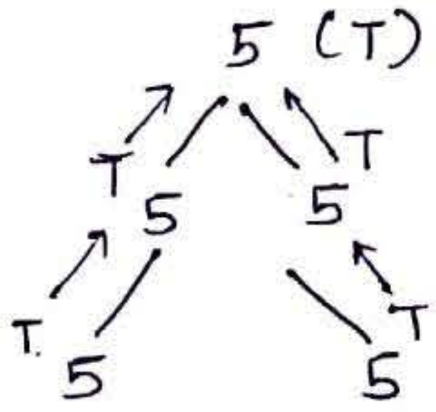
Case 2:

Case 3:

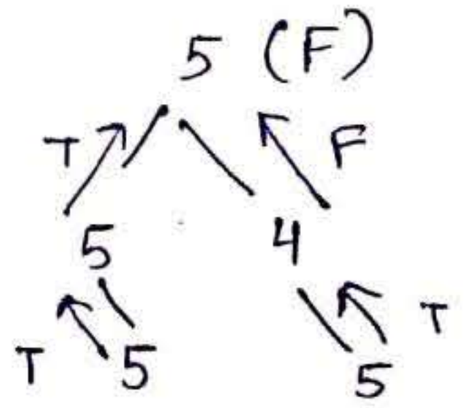
If left and right subtrees return true, check if your both children, if any present, have same value as current node.

* Return true if all same.

ex:



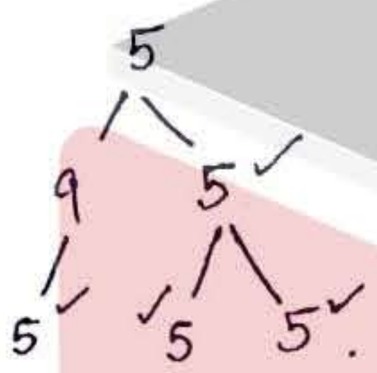
ex:



UNIVALUE SUBTREES Return count of all univalue subtrees. In a univalue subtree, all nodes of subtree have same value.

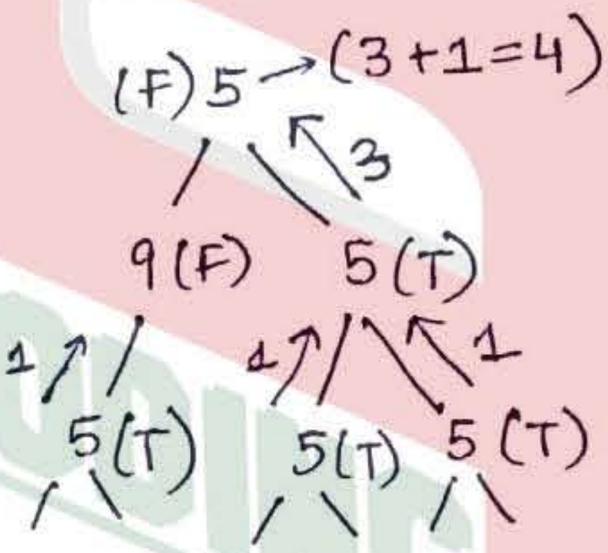
```
public static int countUnivalSubtrees(TreeNode root) {
        // write your code here.
        if (root == null) {
                return 0;
        }
        if (root.left == null &&root.right == null) {
                root.isUni = true;
                return 1;
        }
        int lc = countUnivalSubtrees(root.left);
        int rc = countUnivalSubtrees(root.right);
        root.isUni = (root.left == null || (root.val == root.left.val&&root.left.isUni))
                && (root.right == null || (root.val == root.right.val&&root.right.isUni));
        if (root.isUni) {
                return lc + rc + 1;
        }
        return lc + rc;
}
```

ex:



Nodes with ✓ represent univalued subtrees.
* Each leaf node is a univalue subtree.
⇒ Total 4 univalue subtrees.

How?



* Null nodes return 0.
* Leaf node returns 1 (adding 1 to count of univaluetrees). and marks its isUni property true.

* Each node checks its left and right count. If both have isUni true, it returns left count + right count + 1, 1 for its own subtree, else returns left count + right count.

VALIDATE BST

```
public booleanisValidBST(TreeNode root) {
                        if (root == null) {
                                return true;
                        }
                        if (root.left == null &&root.right == null) {
                                return true;
                        }
                        booleanleftres = isValidBST(root.left);
                        booleanrightres = isValidBST(root.right);
                        booleanmyres = true;
                        if (root.left != null) {
                                TreeNodermax = rmax(root.left);
                                myres = myres&&root.left.val<root.val&&rmax.val<root.val;
                        }
                        if (root.right != null) {
                                TreeNodelmax = lmax(root.right);
                                myres = myres&&root.right.val>root.val&&lmax.val>root.val;
                        }
                        myres = myres&&leftres&&rightres;
                        return myres;
}

public TreeNodelmax(TreeNode root) {
        while (root.left != null) {
                root = root.left;
        }
        return root;
}

public TreeNodermax(TreeNode root) {
        while (root.right != null) {
                root = root.right;
        }
        return root;
}
```

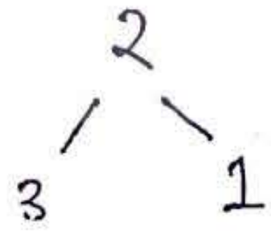
→ finds min value in node.right subtree.

→ finds max value in node.left subtree.

what? Determine if the given binary tree is a valid BST.

ex:

⇒ ans = true.



⇒ ans = false.

How? * A leaf or a null node is always a valid BST.
* In postorder, check BST properties for current node.
* If left is not null, then current data > left max and if right is not null, then current data < right min.
* If both above are true and both left and right results are true, return true for current node forms BST, else false.

LARGEST BST

```
public static int largestBst(TreeNode node) {
        // write your code here.
        pair rp = Bst(node);
        return rp.size;
    }

    public static pair Bst(TreeNode node) {
        if (node == null) {
            return new pair(Integer.MIN_VALUE, Integer.MAX_VALUE, 0);
        }
        pair lp = Bst(node.left);
        pair rp = Bst(node.right);
        pair np = new pair(Integer.MIN_VALUE, Integer.MAX_VALUE, 0);

        np.max = Math.max(rp.max, Math.max(node.val, lp.max));
        np.min = Math.min(node.val, Math.min(lp.min, rp.min));
        np.flag = false;

        if (node.val>lp.max&&node.val<rp.min&&lp.flag == true &&rp.flag == true) {
            np.size = lp.size + rp.size + 1;
            np.flag = true;
        } else {
            np.size = Math.max(lp.size, rp.size);
            np.flag = false;
        }
        return np;
    }

    public static class pair {
        int max;
        int min;
        int size;
        boolean flag;

        public pair(int max, int min, int size) {
            this.size = size;
            this.max = max;
            this.min = min;
            flag = true;
        }
    }
```

What ?

Given root of a Binary Tree. Find no. of nodes in the largest subtree which is a Binary Search Tree (BST)

How ?

* For a BST major condition is that left subtree max should be less than current value and right subtree min should be greater than current value.

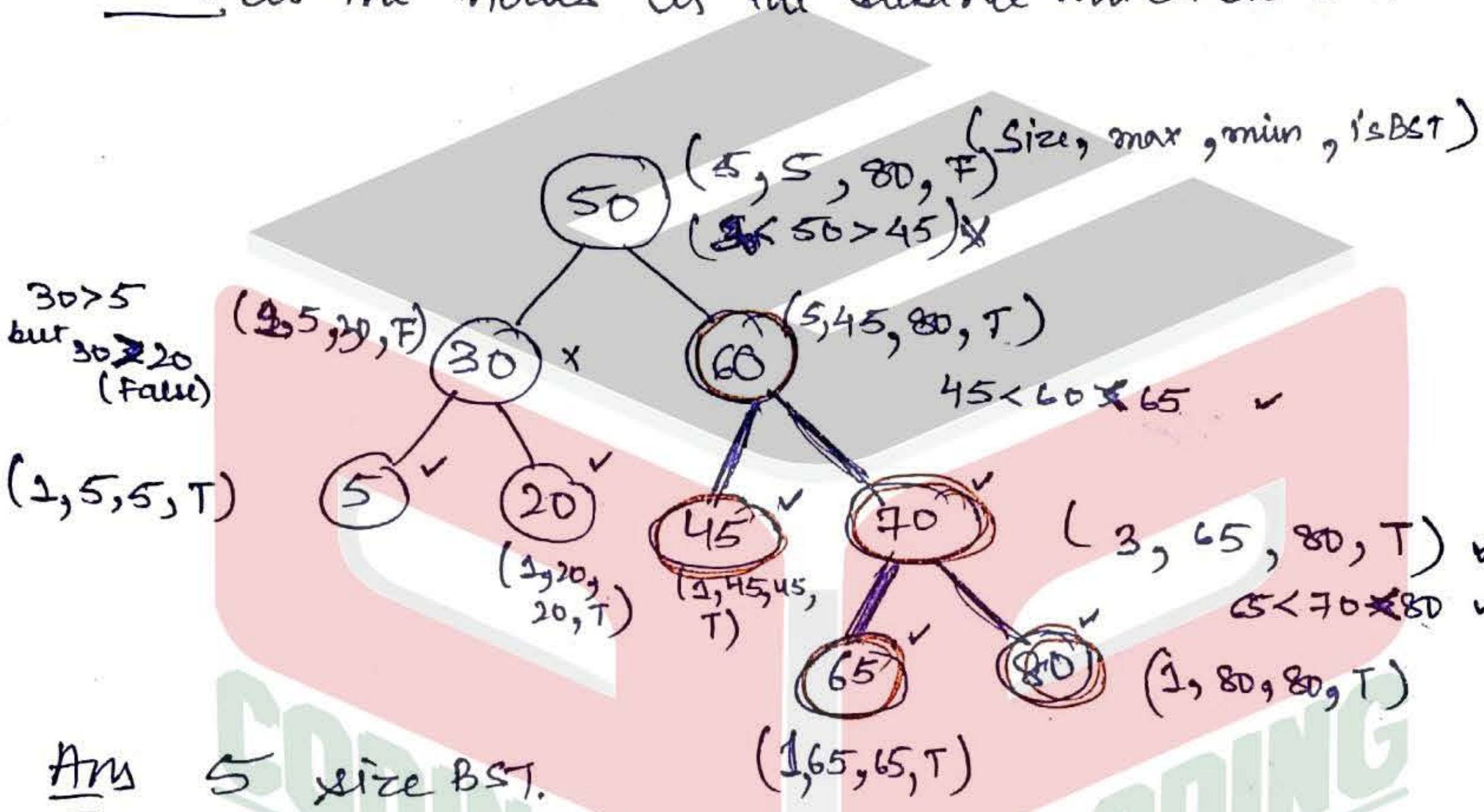
i.e All values in left subtree should be smaller and all values in right subtree should be greater.

* flag denotes whether the current subtree rooted at current node is a bst or not

  flag = true → when left is bst, right is BST and condition for cur. is true.

  flag = false → otherwise.

* size is the nodes in the subtree which is BST



Ans 5 size BST.

MINIMUM DISTANCE BETWEEN BST NODES

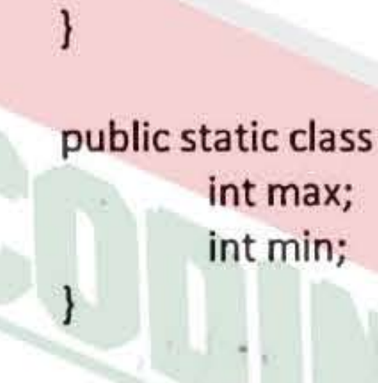
```
public int minDiffInBST(TreeNode root) {
                diff = Integer.MAX_VALUE;
                helper(root);
                return diff;
        }

        static int diff = Integer.MAX_VALUE;
        public static pair helper(TreeNode node) {
                if (node == null) {
                        pair l = new pair();
                        l.max = Integer.MIN_VALUE;
                        l.min = Integer.MAX_VALUE;
                        return l;
                }
                if (node.left == null &&node.right == null) {
                        pair l = new pair();
                        l.max = node.data;
                        l.min = node.data;
                        return l;
                }
                pair lp = helper(node.left);
                pair rp = helper(node.right);
                pair np = new pair();

                np.max = Math.max(rp.max, Math.max(node.data, lp.max));
                np.min = Math.min(node.data, Math.min(lp.min, rp.min));
                int d1 = node.left != null ? Math.abs(node.data - lp.max) : Integer.MAX_VALUE;
                int d2 = node.right != null ? Math.abs(node.data - rp.min) : Integer.MAX_VALUE;
                diff = Math.min(diff, Math.min(d1, d2));
                return np;
        }

        public static class pair {
                int max;
                int min;
        }
```


<u>What</u>?

Given a BST root. Find the minimum difference between the values of any 2 different nodes in tree.

<u>How</u>?

<u>eg</u>:



Ans = 1

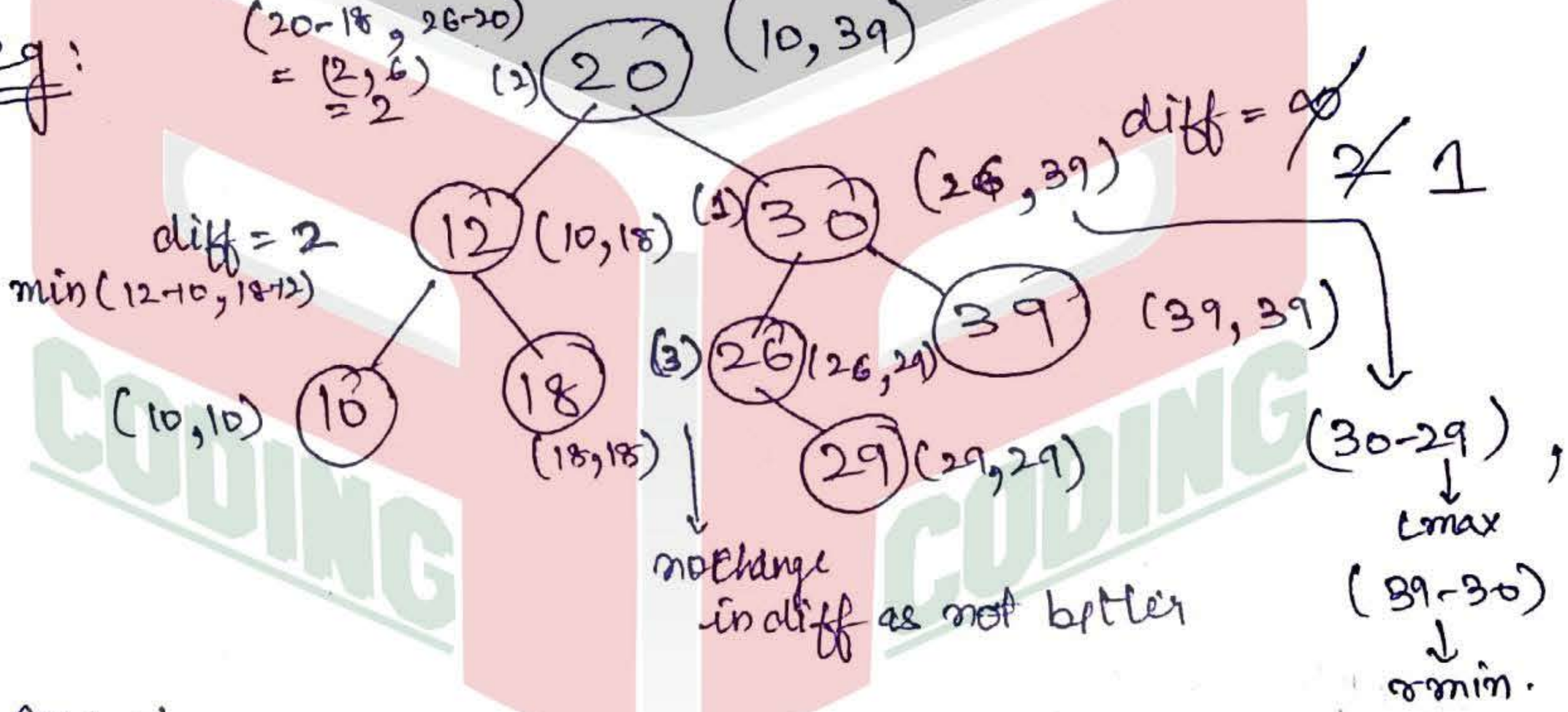
How?

Make a pair class that stores maximum and minimum value of a given subtree which is returned from the function.

For a node minimum difference will be between the just smaller and just greater node. To find just larger value & just smaller value we will use left pair and right pair.

Minimum difference will be compared to the absolute difference of <u>left max & node val</u> & <u>right min & node val</u>.

<u>eg:</u>



<u>Ans =></u>

node max = max(node.val, left max, right max)

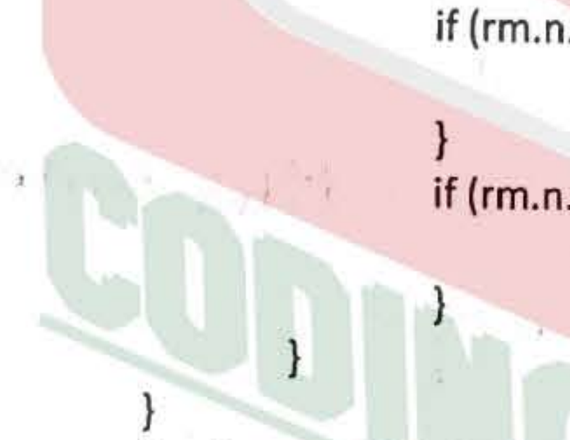
node min = min(node.val, left.min, right min)

TOP VIEW OF BINARY TREE

```
public static class Qnode {
        TreeNode n;
        int cl;

        Qnode(TreeNode n, int l) {
            this.n = n;
            this.cl = l;
        }
    }

    static TreeMap<Integer, Integer> map;

static void printTopView(TreeNode root) {
    // add your code
    Qnodexnode = new Qnode(new TreeNode(Integer.MIN_VALUE), 0);
    map = new TreeMap<>();
    LinkedList<Qnode> queue = new LinkedList<>();
    queue.addLast(new Qnode(root, 0));
    queue.addLast(xnode);
    while (queue.size() > 0) {
        Qnode rm = queue.removeFirst();
        if (rm.n == null) {
            if (queue.size() > 0) {
                queue.addLast(xnode);
            }
        } else {
            if (!map.containsKey(rm.cl)) {
                map.put(rm.cl, rm.n.val);
            }
            if (rm.n.left != null) {
                queue.addLast(new Qnode(rm.n.left, rm.cl - 1));
            }
            if (rm.n.right != null) {
                queue.addLast(new Qnode(rm.n.right, rm.cl + 1));
            }
        }
    }
    for (int v : map.keySet()) {
        System.out.print(map.get(v) + " ");
    }
}
```


What? Given root of binary tree, print top view of binary tree.

ex:

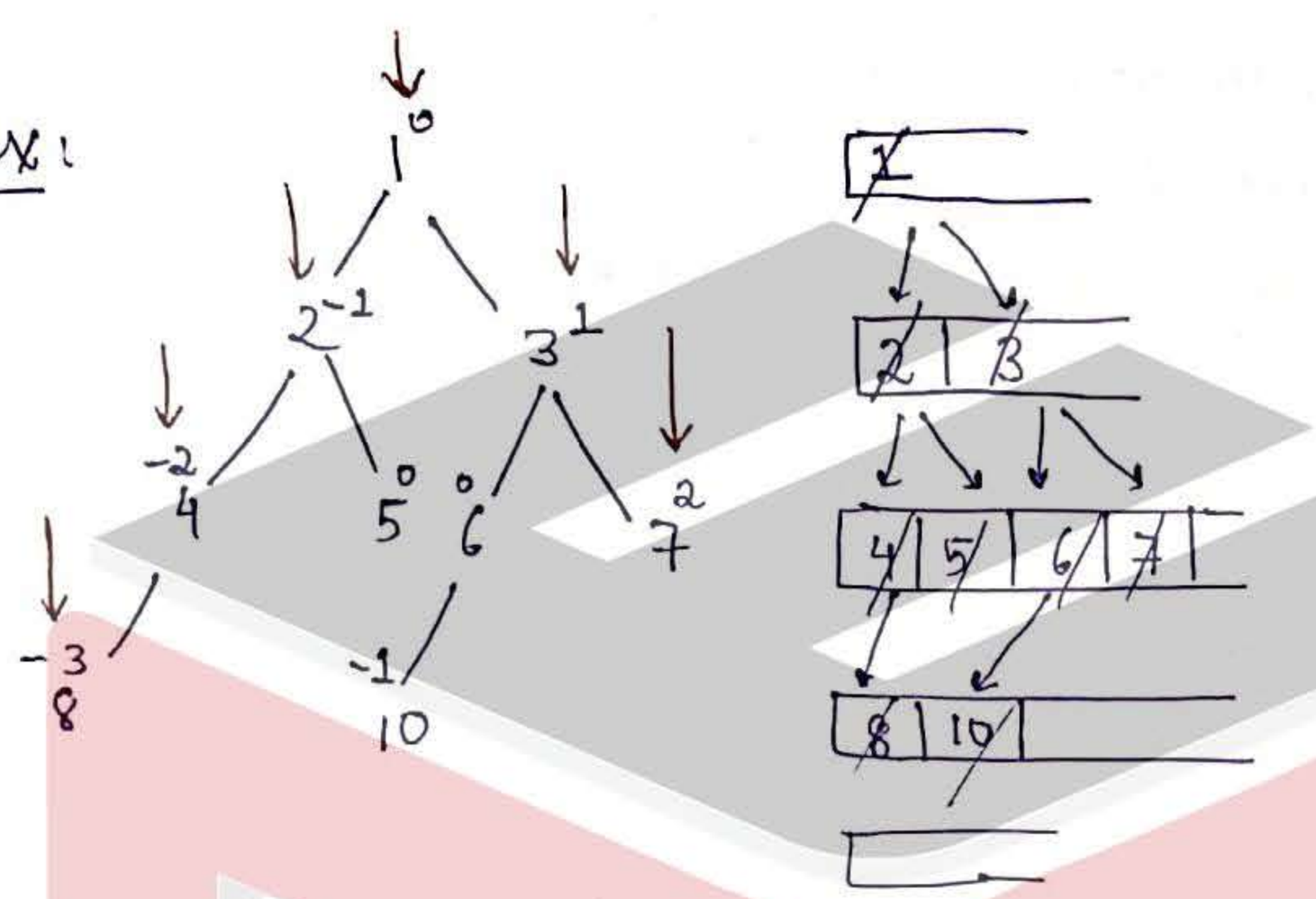


$\Rightarrow [4, 2, 1, 3, 6]$

How? * Use vertical width method while traversing level order wise. Maintain a hashmap / treemap to store val in front of width.

* If width has not occured yet, add width with current node.data.

ex:



$$\begin{bmatrix} 0 \to 1 & -3 \to 8 \\ -1 \to 2 & \\ 1 \to 3 & \\ -2 \to 4 & \\ 2 \to 7 & \end{bmatrix}$$

∴ ans is : [8, 4, 2, 1, 3, 7]

[Result is shown in sorted widths manner, hence we use treemap.]

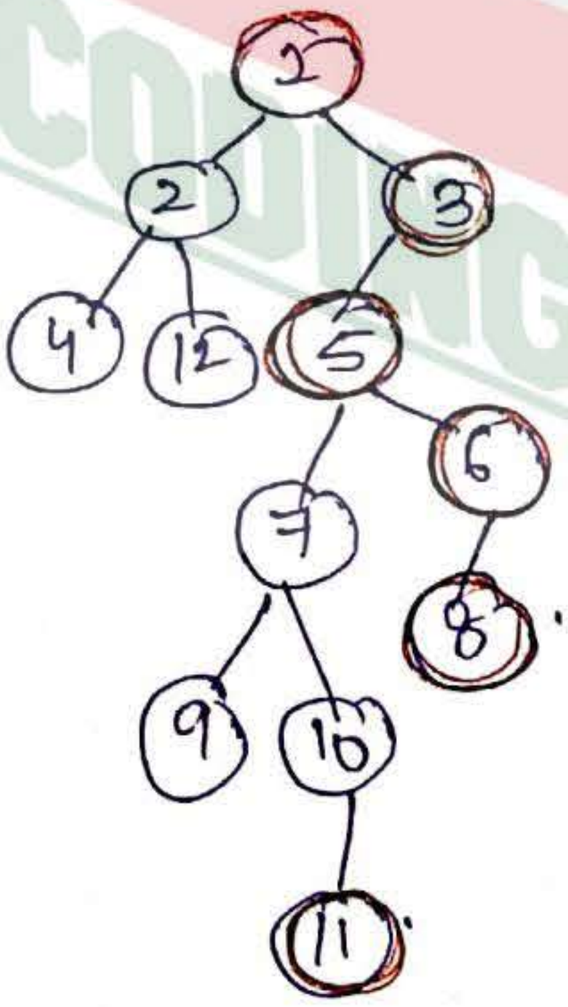
BINARY TREE RIGHT SIDE VIEW

```
void rightView(TreeNode root) {
                        LinkedList<TreeNode> queue = new LinkedList<>();
                        queue.add(root);
                        TreeNodennode = new TreeNode(Integer.MIN_VALUE);   //delimeter node
                        queue.add(nnode);
                        TreeNodeprev = root;
                        while (queue.size() > 0) {
                                TreeNodern = queue.removeFirst();
                                if (rn.data == nnode.data) {
                                        System.out.print(prev.data + " ");
                                        if (queue.size() != 0) {
                                                queue.addLast(nnode);
                                        }
                                } else {
                                        if (rn.left != null) {
                                                queue.add(rn.left);
                                        }
                                        if (rn.right != null) {
                                                queue.add(rn.right);
                                        }
                                }
                                prev = rn;
                        }
                }
```

What?

Given a root of the Binary Tree. Print the Right view of the tree.

eg:

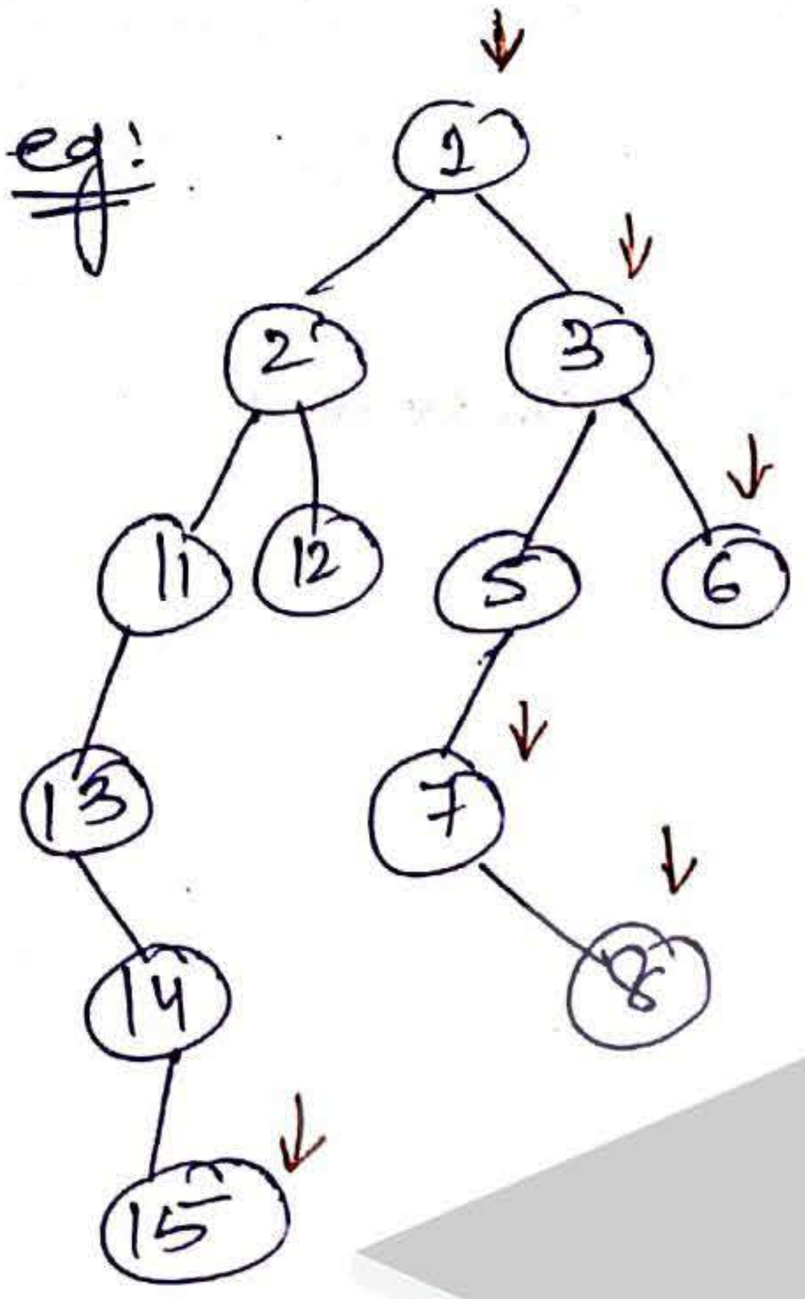


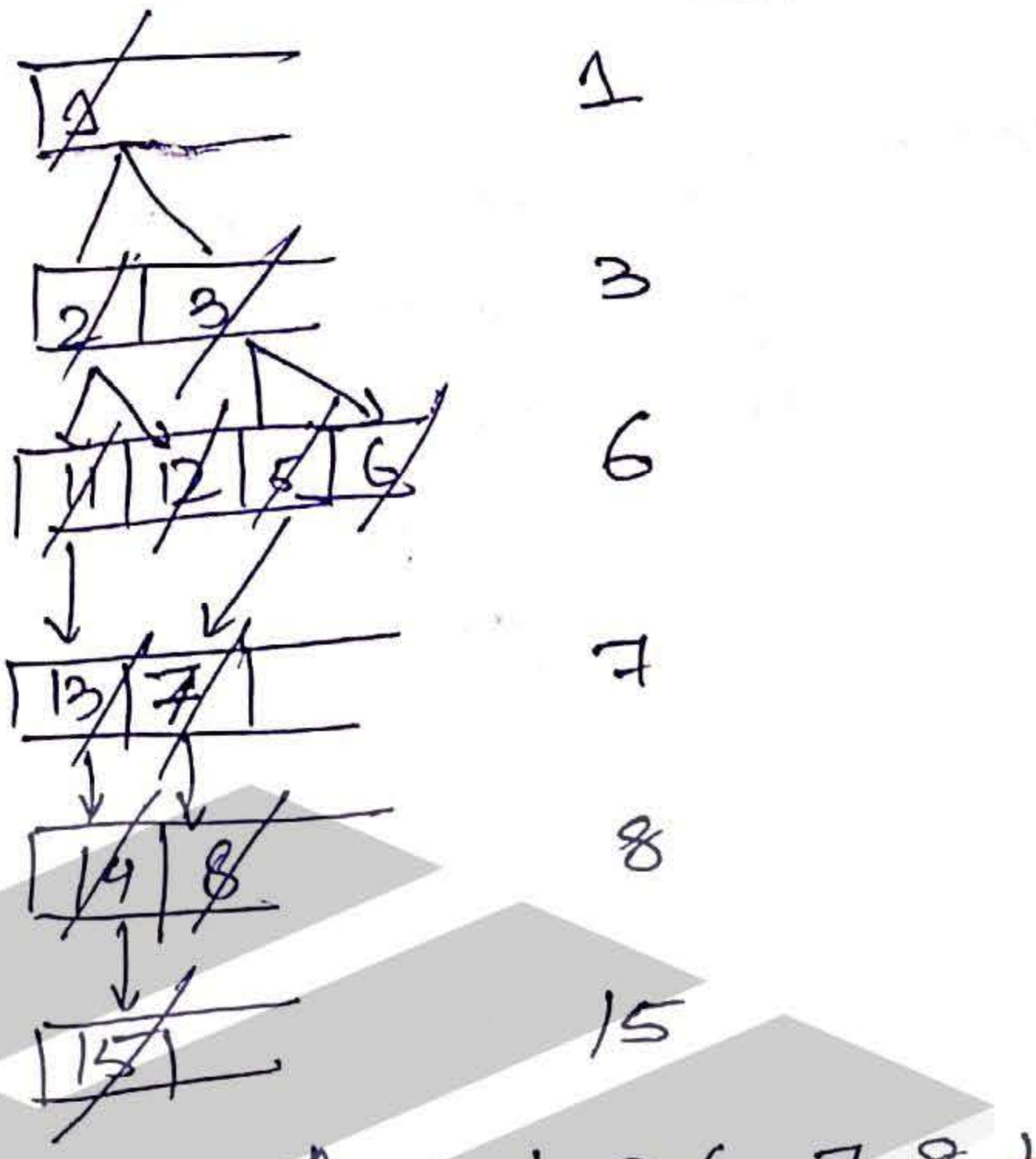
Right view = 1 3 5 6 8 11

How?

We will use level order to solve this problem last node to be visited at each level will be part of the right side view of tree.

eg:





1

3

6

7

8

15

Ans= 1 3 6 7 8 15

BINARY TREE CAMERAS

```
public static int count = 0;
        public static int minCameraCover(TreeNode root) {
                HashSet<TreeNode> set = new HashSet<>();
                set.add(null);
                helper(root, null, set);
                int res = count;
                count = 0;
                return res;
        }
        public static void helper(TreeNode node, TreeNode parent, HashSet<TreeNode> set) {
        if (node == null) {
                return;
        }
        helper(node.left, node, set);
        helper(node.right, node, set);
        if((parent == null && !set.contains(node))|| !set.contains(node.left) || !set.contains(node.right)) {
                count++;
                set.add(node);
                set.add(parent);
                set.add(node.left);
                set.add(node.right);
        }
}
```

What?

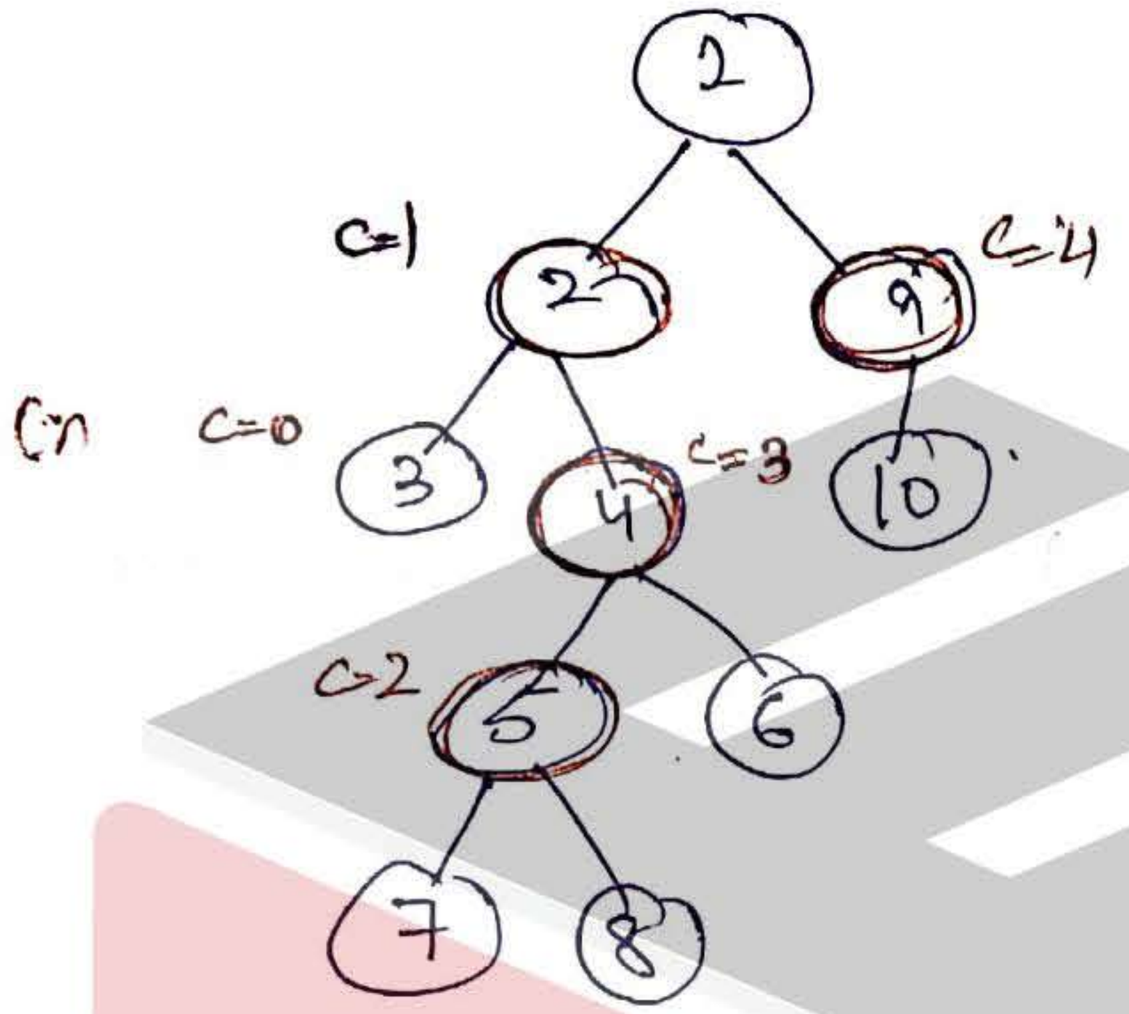
Given root of Binary Tree, and cameras are installed on the nodes of the tree. Each camera at node can monitor its parent, itself and its immediate children. You have to calculate the minimum number of cameras needed to monitor all nodes of the tree.

How?

We will use the greedy approach to solve this problem At each node post order we will check if that nodes any of the children is not in visited set i'e they are not in Camera Cover. we will increment Camera Count.

Or incrementing Camera Count will all the nodes that are covered by that camera in set.

In case of root since parent is null we will check if root is present or not if not then visit it and increment count.



At 2 since neither 3 or 4 is set so count incremente.

⇒ count=1 set <2,3,4,1>

At 5, count = 2.

set <1,2,3,4,5,7,8,9>

At 4 count=3 6 not in set

set <1,2,3,4,5,6,7,8>

At 9 count=4 10 is not in set

set <1,2,3,4,5,6,7,8,9,10>

Ans = 4

UNIQUE BST

```
public static int numTrees(int n) {
        if (n == 0 || n == 1) {
                return 1;
        }
        if (n == 2) {
                return 2;
        }
        int[] CatallanArray = new int[n + 1];
        CatallanArray[0] = 1;
        CatallanArray[1] = 1;
        CatallanArray[2] = 2;
        for (int i = 3; i<= n; i++) {
                int val = 0;
                int left = 0, right = i - 1;
                while (left != i) {
                        CatallanArray[i] += CatallanArray[left] * CatallanArray[right];
                        left++;
                        right--;
                }
        }
        return CatallanArray[n];
}
```

What? Given a number N, find how many unique BST are possible from N nodes.

How?

Catalan is used here.

For N = 0, Null tree = 1.

N = 1,  = 1.

N = 2,  = 2.

N = 3, we can use catalan array to calculate value, $C_0C_2 + C_1C_1 + C_2C_0$.

$= 1(2) + 1(1) + 2(1) = 5.$

CONSTRUCT STRING FROM BTREE

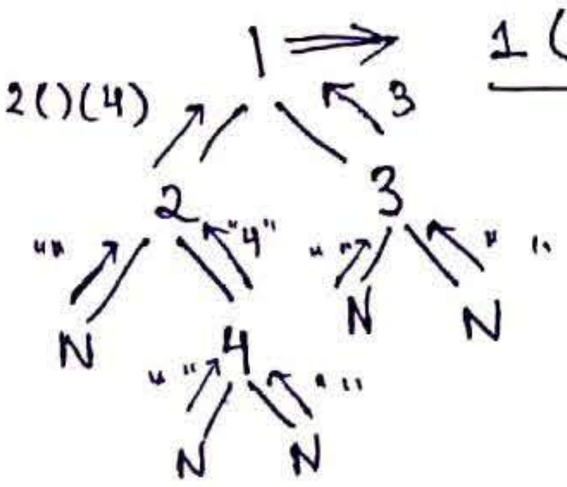
```
public static String tree2str(TreeNode root) {
        if (root == null) {
                return "";
        }
        if (root.left == null &&root.right == null) {
                return root.val + "";
        }
        if (root.right == null) {
                return root.val + "(" + tree2str(root.left) + ")";
        }
        return root.val + "(" + tree2str(root.left) + ")" + "(" + tree2str(root.right) + ")";
}
```

what? You are given the root of binary tree, construct a string of parenthesis and integers from binary tree in preorder traversing way. Represent null node by "()" and omit all unnecessary empty parenthesis pairs that do not affect one-to-one mapping of tree with string.

ex: [1,2,3,4] ⇒ "1(2(4))(3)"

How? * Return empty string for null node.
* Leaf node returns its value as string.
* For each node, if its right is null, string returned is "(" + left tree string + ")".
* If both nodes are present or right is present, return string as "(" + left + ")" + "(" + right + ")". Even if left is null but right is not null, for maintaining one-to-one mapping, left "()" is added.

ex: 1 ⇒ 1(2()(4))(3)



POST FROM IN AND PRE

```
public static void printPostOrder(int in[], int pre[], int n) {
        // Your code here
        printPost(0, n - 1, 0, n - 1, in, pre);
    }

    public static void printPost(int ins, int ine, int prs, int pre, int[] in, int[] prea) {
        if (ins >ine || prs> pre) {
            return;
        }
        int i = 0;
        for (i = ins; i<= ine; i++) {
            if (in[i] == prea[prs]) {
                break;
            }
        }
        int lhs = i - ins;
        printPost(ins, i - 1, prs + 1, prs + lhs, in, prea);
        printPost(i + 1, ine, prs + lhs + 1, pre, in, prea);
        System.out.print(prea[prs] + " ");
    }
```

What? Given inorder and preorder traversals of tree, print its postorder.

How?

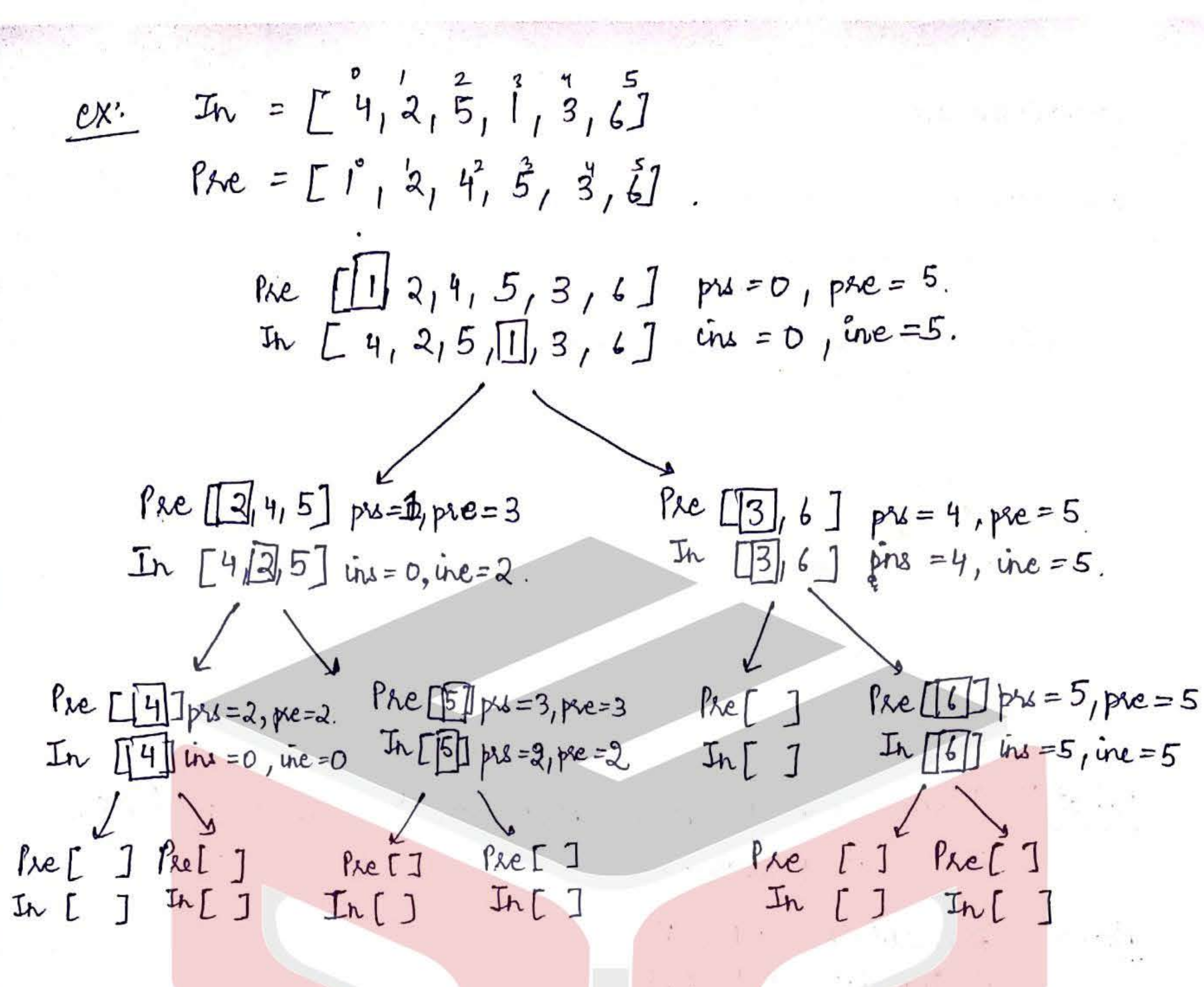
* Preorder : N L R
* Inorder : L N R.
* Take 4 indices; ins, ine as start and end of inorder, and prs, pre as start and end of preorder arrays.
* In preorder, find prea[prs] in inorder array.
* lhs ⇒ Index found where prea[prs] is and lhs = i − ins, where i is index.
* Two calls break as left and right where.

for left call : Pre : prs+1 to prs + lhs
In : ins to i−1.

for right call : Pre : prs + lhs +1 to pre
In : i+1 to ine.

ex: In = [4, 2, 5, 1, 3, 6]
(0) (1) (2) (3) (4) (5)

Pre = [1, 2, 4, 5, 3, 6]
(0) (1) (2) (3) (4) (5)



Print in postorder as: [4, 5, 2, 6, 3, 1]

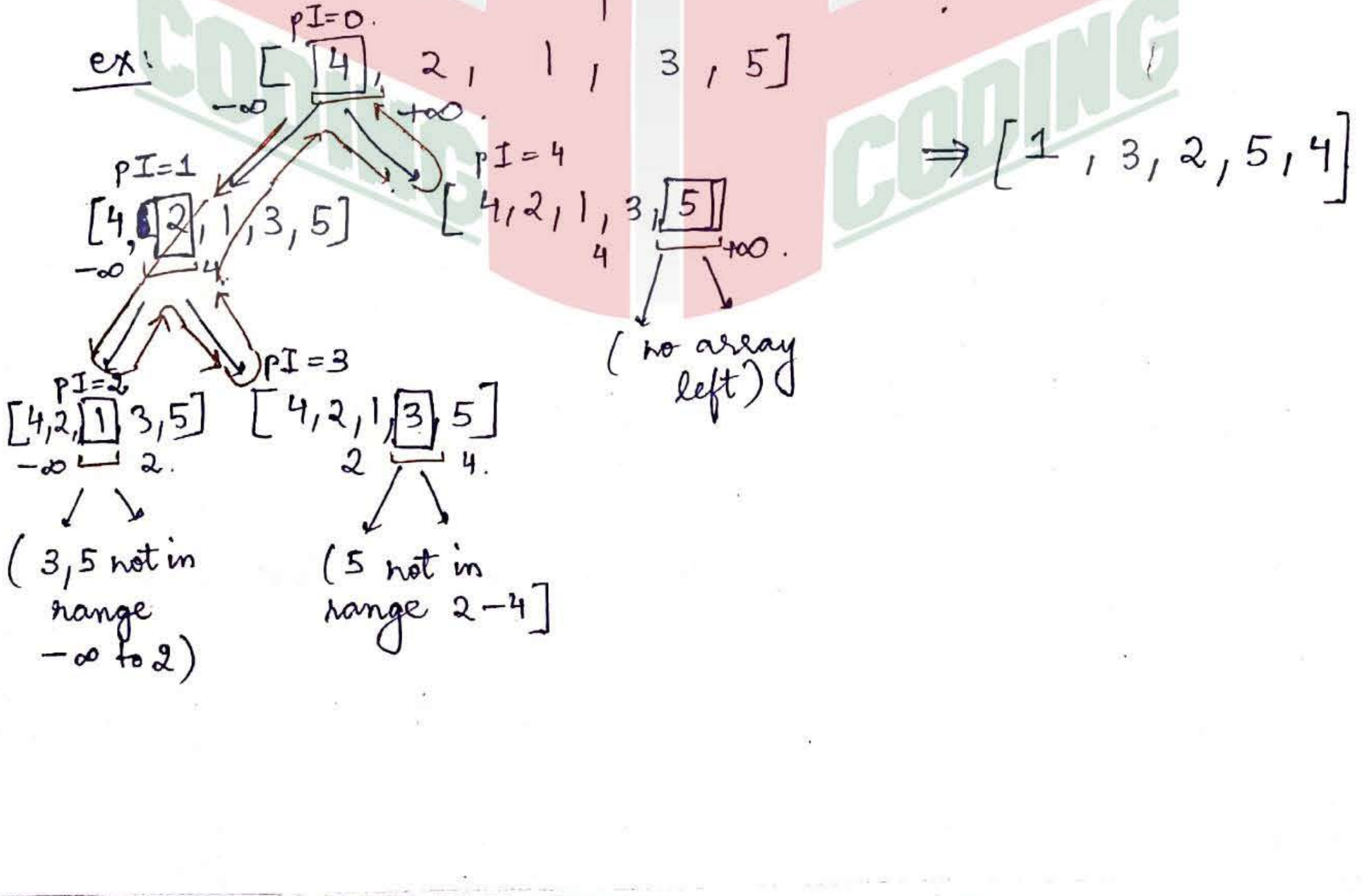
PRE TO POST

```
static int preIndex = 0;
        public static void postfrompre(int[] pre) {
                findpost(pre, pre.length, Integer.MIN_VALUE, Integer.MAX_VALUE);
        }

        private static void findpost(int pre[], int n, int minval, int maxval) {
                if (preIndex>= pre.length || pre[preIndex] >maxval || pre[preIndex] <minval)
                        return;

                int val = pre[preIndex];
                preIndex++;
                findpost(pre, n, minval, val);
                findpost(pre, n, val, maxval);
                System.out.print(val + " ");
        }
```

What? Given an array representing preorder, print its postorder.

How? * Take $-\infty$ and $+\infty$ as two limits of minvalue and max value in BST.
* Keep static preindex = 0 to access indexes in array.
* In preorder, keep value = pre[preindex] and make calls for $-\infty$ to value, and value to $+\infty$.
* Print value in postorder.

ex:



CONVERT SORTED ARRAY INTO BST

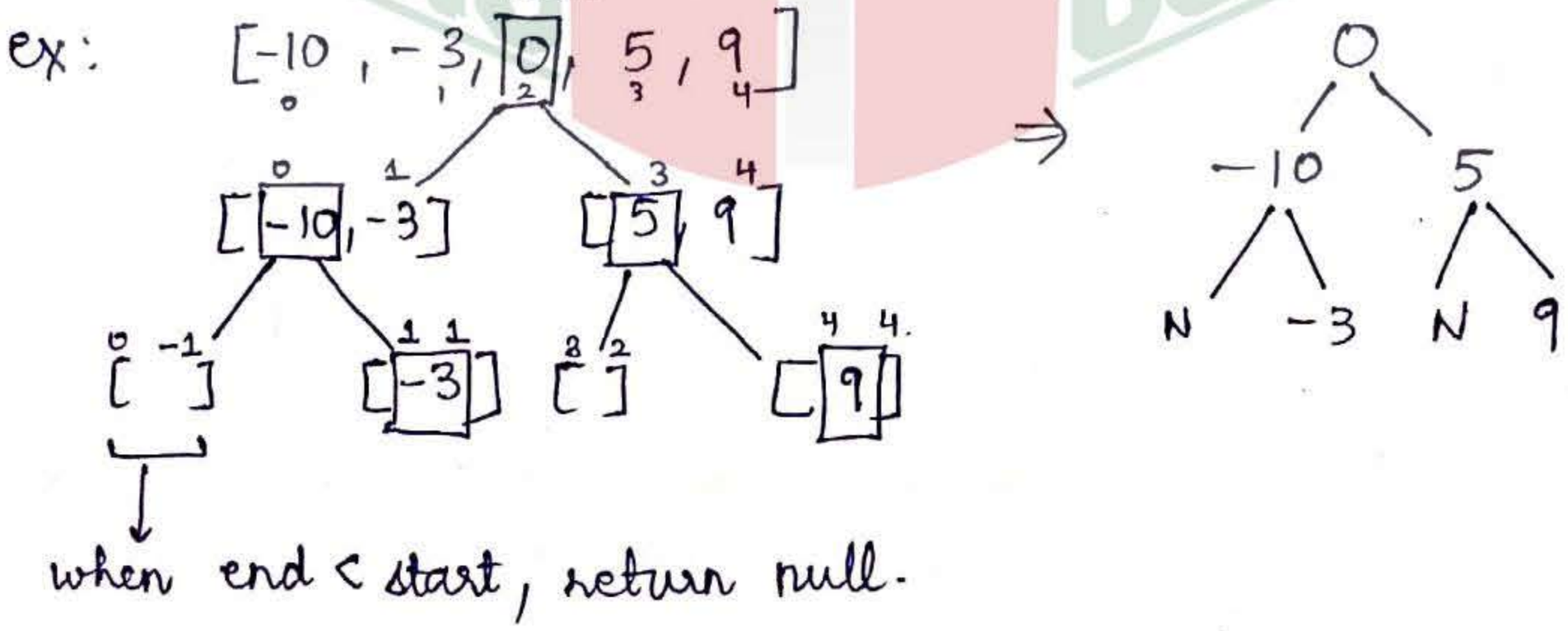
```
public static TreeNodesortedArrayToBST(int[] nums) {
        return ArrToBst(0, nums.length - 1, nums);
    }

    public static TreeNodeArrToBst(int start, int end, int[] nums) {
        if (end < start) {
            return null;
        }
        if (start == end) {
            TreeNode leaf = new TreeNode(nums[start]);
            return leaf;
        }
        int mid = (start + end) / 2;
        TreeNode node = new TreeNode(nums[mid]);
        node.left = ArrToBst(start, mid - 1, nums);
        node.right = ArrToBst(mid + 1, end, nums);
        return node;
    }
```

What? Given a sorted array, convert it into height balanced binary search tree. (A height balanced tree is tree where depth of two subtrees of node differ by atmost 1).

How? given arr = [-10, -3, 0, 5, 9].

1) Take out mid ((start+end)/2).
2) node.left = array from 0 to mid-1 .
3) node.right = array from mid+1 to length] recursive calls.

ex:



when end < start, return null.

PREORDER AND BST

```
public static booleanpreOrder(int[] arr) {
        // write your code here.
        int root = Integer.MIN_VALUE;
        Stack<Integer>st = new Stack<>();
        for (int i = 0; i<arr.length; i++) {
            if (arr[i] < root) {
                return false;
            } else {
                if (st.size() == 0) {
                    st.push(arr[i]);
                } else {
                    if (arr[i] <st.peek()) {
                        st.push(arr[i]);
                    } else {
                        while (st.size() > 0 &&st.peek() <arr[i]) {
                            root = st.pop();
                        }
                        st.push(arr[i]);
                    }
                }
            }
        }
        return true;
    }
```

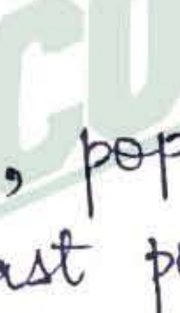
What? Find whether given array can represent preorder traversal of BST or not.

How? * Initialize an integer root = $-\infty$.
* For each value in array, if value < root, return false, else push in stack if either stack is empty or value < top of stack.
* If value > top of stack, pop till it is greater and change root to last popped value. Now push value.

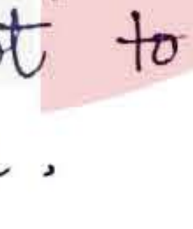
Ex: [4, 2, 1, 5, 8]

c) 1 → 1 < 2 and 1 > $-\infty$, push 1 in stack.
b) 2 → 2 < 4 and 2 > $-\infty$, push 2 in stack.
a) 4 → 4 > $-\infty$, push 4 as stack is empty.

root = $-\infty$.
d) 5 > 1, pop 1;
5 > 2, pop 2;
5 > 4, pop 4.



e) root = 4



f) 8 < 5, push 8.

⇒ Return true.

MAKE BINARY TREE FROM LINKED LIST

```
public static class LinkedListNode {
        int data;
        LinkedListNode next;
        LinkedListNode(int d) {
            data = d;
            next = null;
        }
    }

    public static class BinaryTree {
    public TreeNode convert(LinkedListNode head, TreeNode node) {
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        if (head == null) {
            node = null;
            return null;
        }
        node = new TreeNode(head.data);
        q.add(node);
        head = head.next;
        while (head != null) {
            TreeNode parent = q.poll();
            TreeNodeleftChild = null, rightChild = null;
            leftChild = new TreeNode(head.data);
            q.add(leftChild);
            head = head.next;
            if (head != null) {
                rightChild = new TreeNode(head.data);
                q.add(rightChild);
                head = head.next;
            }
            parent.left = leftChild;
            parent.right = rightChild;
        }
        return node;
    }
}
```


what? Given a linked list representation of complete binary tree, construct a binary tree from it s.t for every i, 2i+1 and 2i+2 are its left and right child respectively.

How? * Use a queue to create tree.
* Insert head of linked list initially in queue, if it is null return null.
* while queue has elements, for each pop from queue, assign left as head if head exists, and right if right exists. (head.next at each insertion in queue).

ex: head.
1 → 2 → 3 → 4 → 5.

a) 9 [1.]

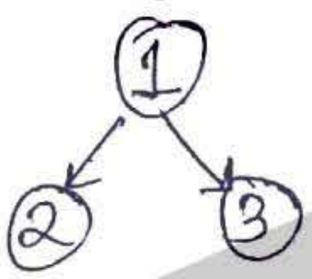
head
1 → 2 → 3 → 4 → 5

while:

b) Pop 1.
Make 1 parent (1)
Make left as (2)
Make right as (3).

[1]
[1|2]
[1|2|3]

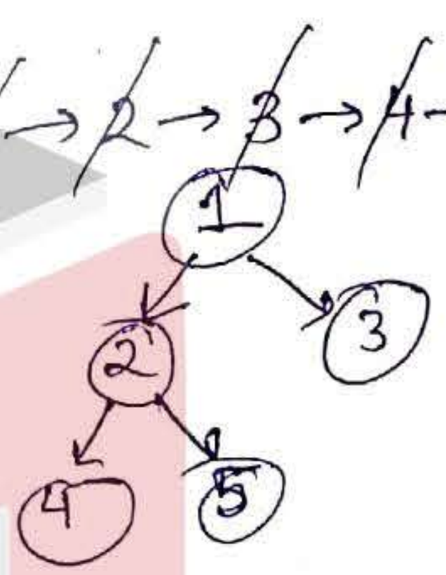
head
1 → 2 → 3 → 4 → 5



c) Pop 2.
Make 2 parent (2)
Make left as (4)
Make right as (5)

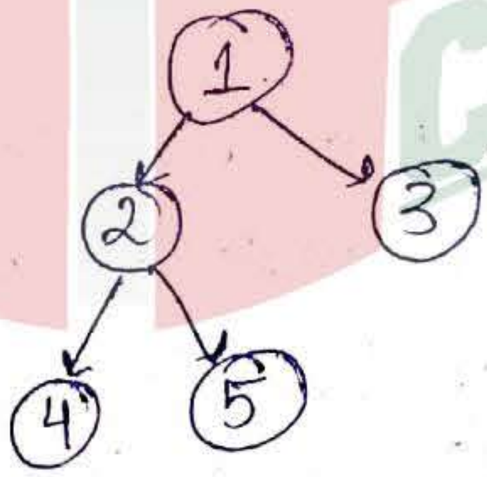
[1|2|3]
[1|2|3]
[1|2|3|4]
[1|2|3|4|5]

1 → 2 → 3 → 4 → 5



d) head is at null,
now each pop makes popped as parent, and their lefts and rights are null.

⇒ Tree formed:



CONVERT BST TO SORTED CIRCULAR DOUBLY LINKED LIST

```
public static class Pair {
        TreeNode head;
        TreeNode tail;
        Pair(TreeNode h, TreeNode t) {
            this.head = h;
            this.tail = t;
        }
    }
    public static TreeNodetreeToDoublyList(TreeNode root) {
        // write your code here.
        Pair p = TtoDLL(root);
        if (p.head != null &&p.tail != null) {
            p.head.left = p.tail;
            p.tail.right = p.head;
        }
        return p.head;
    }

    public static Pair TtoDLL(TreeNode root) {
        if (root == null) {
            return new Pair(null, null);
        }
        if (root.left == null &&root.right == null) {
            return new Pair(root, root);
        }
        Pair lp = TtoDLL(root.left);
        Pair rp = TtoDLL(root.right);
        Pair cp = new Pair(null, null);
        if (lp.tail == null) {
            cp.head = root;
        } else if (lp.tail != null) {
            lp.tail.right = root;
            root.left = lp.tail;
            cp.head = lp.head;
        }
        if (rp.head == null) {
            cp.tail = root;
        } else if (rp.head != null) {
            root.right = rp.head;
            rp.head.left = root;
            cp.tail = rp.tail;
        }
        return cp;
    }
```


what? Convert BST to sorted circular doubly linked list. Left and right pointers should work as prev and next pointers. Return pointer to first element in BST.

How? * Keep a Pair with head and tail of list and return this pair.

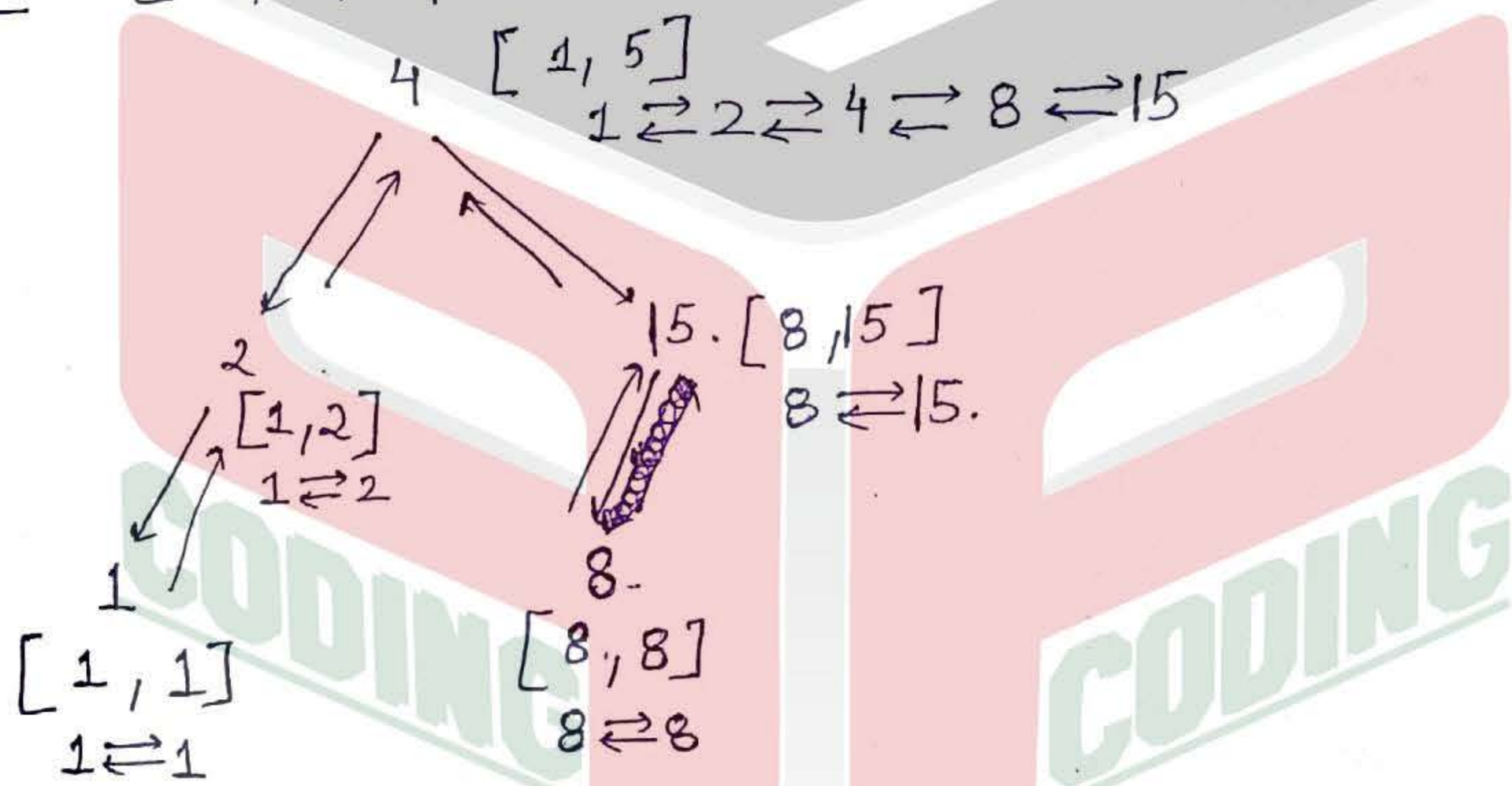
* Null returns Pair (head, tail) as (null, null).
* Leaf returns Pair as (leaf, leaf).
* Every other node checks for its head and tail and joins the left pair to right pair.

Case 1: LP (left pair) RP (right pair) MP (my pair)
(null, null) (x, y) (node, y) → node.next = x.
x.prev = node

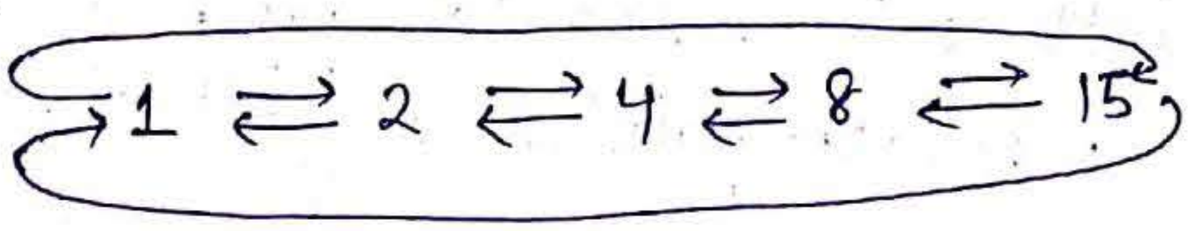
Case 2: LP (x, y) RP (null, null) MP (x, node) → y.next = node
node.prev = y.

Case 3: LP (x, y) RP (z, w) MP (x, w) → y.next = node
z.prev = node
node.prev = y
node.next = z.

ex: [4, 2, 1, 15, 8]



* In the end, make sure to make list cyclic.
head.left = tail.
tail.right = head.



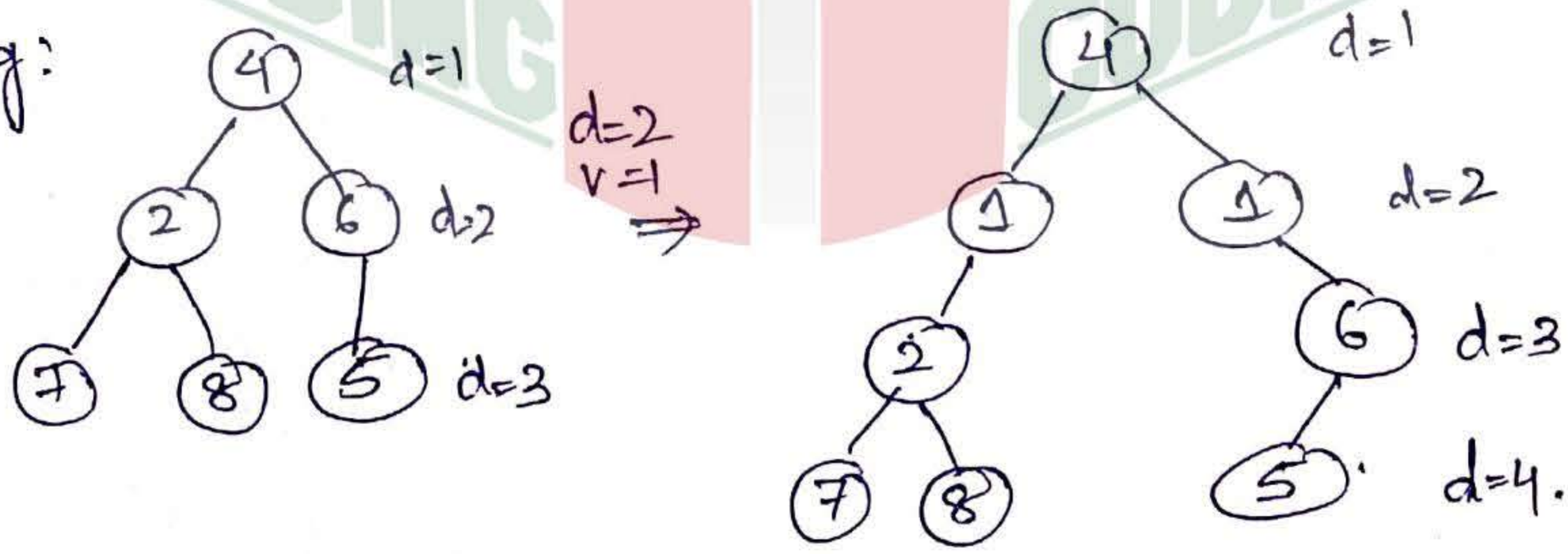
ADD ONE ROW TO TREE

```
public static TreeNodeaddOneRow(TreeNode root, int v, int d) {
        helper(root, v, 1, d);
        if (d == 1) {
            TreeNode n = new TreeNode(v);
            n.left = root;
            root = n;
        }
        return root;
    }
public static void helper(TreeNode root, int v, int l, int d) {
    if (root == null) {
        return;
    }
    helper(root.left, v, l + 1, d);
    helper(root.right, v, l + 1, d);
    if (l == d - 1) {
        TreeNode n = new TreeNode(v);
        n.left = root.left;
        root.left = n;
        TreeNode n2 = new TreeNode(v);
        n2.right = root.right;
        root.right = n2;
    }
}
```

What?

Given root of Binary Tree, value v and depth d. add a row of nodes with value v at the given depth (d). The root node is at depth 1.

eg:

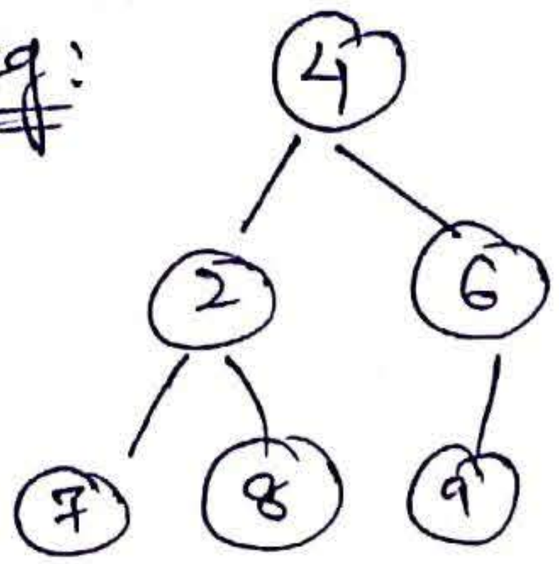


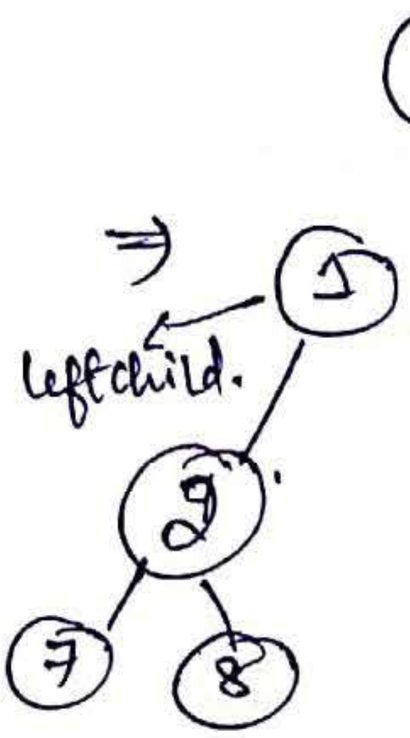
How?

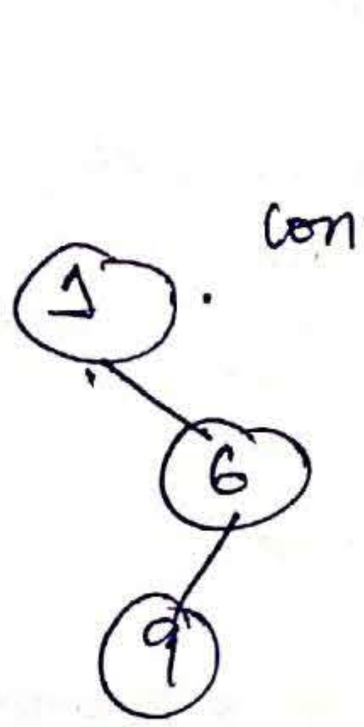
When a new node is inserted Both current children and parent connection is to be made.

So since node is inserted at level d we will attach new node to parent and its children to new node.

eg:



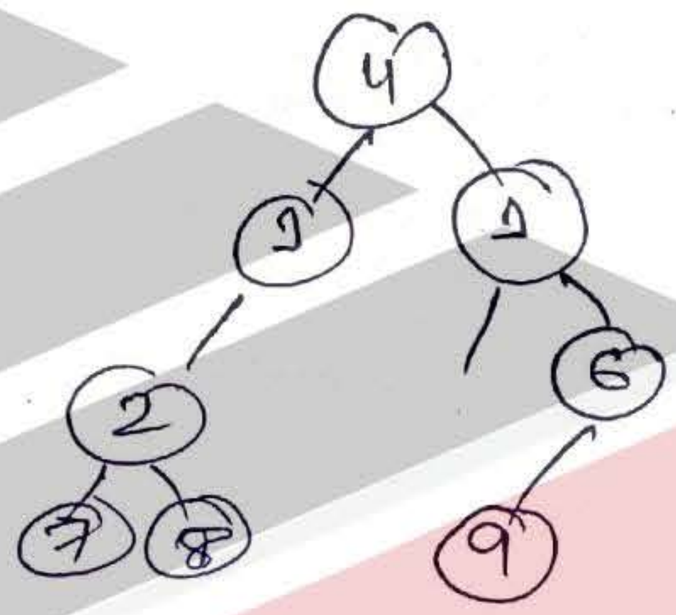




connecting child.

right child to new right child.

parent left = node left.
parent right = node right.



Ans

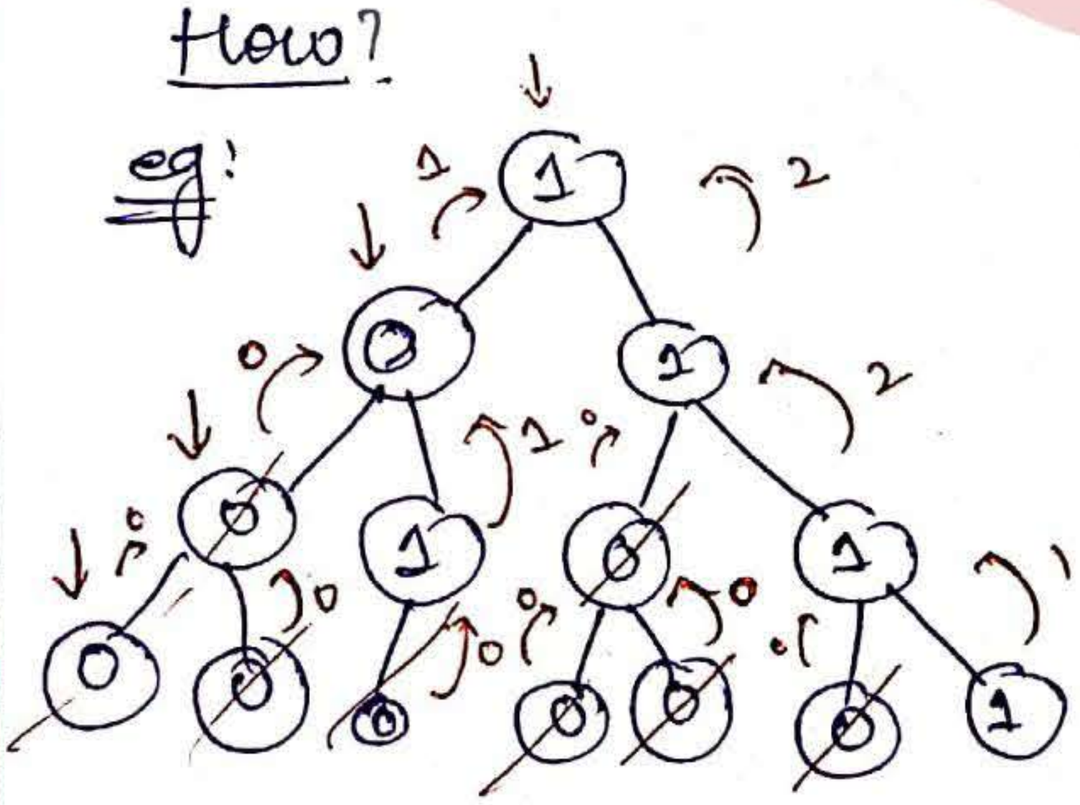
BINARY TREE PRUNING

```
public static TreeNodepruneTree(TreeNode root) {
        pair rp = helper(root);
        return rp.node;
    }

    public static class pair {
        TreeNode node;
        int count;
        public pair(TreeNode n, int c) {
            this.node = n;
            this.count = c;
        }
}

public static pair helper(TreeNode root) {
    if (root == null) {
        return new pair(null, 0);
    }
    pair lp = helper(root.left);
    root.left = lp.node;
    pair rp = helper(root.right);
    root.right = rp.node;
    if (lp.count == 0 &&rp.count == 0 &&root.val == 0) {
        return new pair(null, 0);
    }
    return new pair(root, lp.count + rp.count + root.val);
}
```

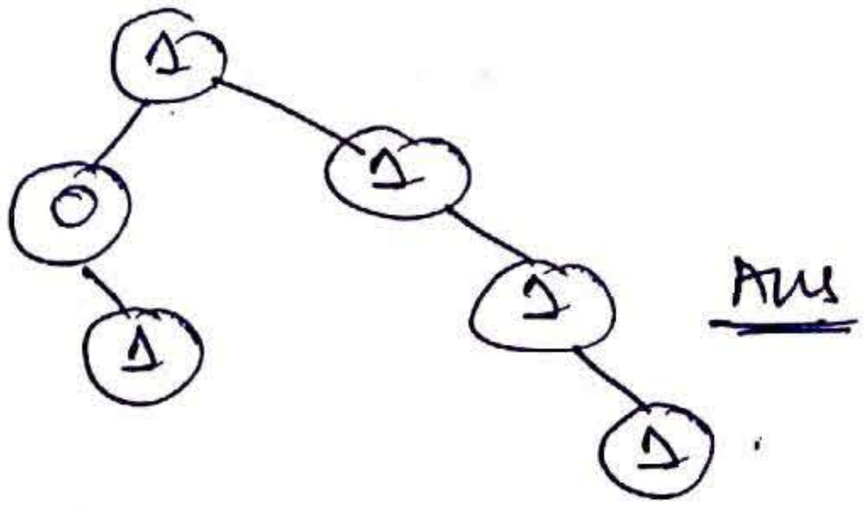

What?

Given node (root) of Binary Tree. All the nodes of Binary Tree has either value 0 or 1. Remove all such subtree whose count of nodes having 1 as value is zero.

How?



⇒



* At each level count of 1's
  = left count + right count + 1
* If count = 0 then return null.

MAXIMUM PATH SUM FROM LEAF TO LEAF

```
public static int maxPathSum(Node root) {
        // your code here
        max = Integer.MIN_VALUE;
        helper(root);
        return max;
    }

    static int max = Integer.MIN_VALUE;
    private static int helper(Node node) {
        if (node == null) {
            return 0;
        }
        int lsum = helper(node.left);
        int rsum = helper(node.right);

        int rv = Math.max(lsum, rsum) + node.data;
        max = Math.max(lsum + rsum + node.data, max);
        return rv;
    }
```

what?

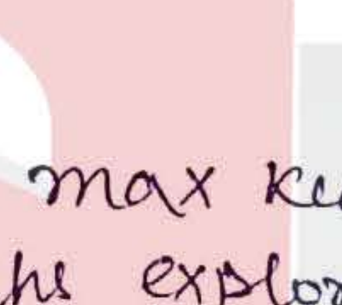
Given the root of a non empty Binary Tree. Find the maximum Path sum such that path starts from one leaf and end at other leaf.

How?

Static variable max keeps the max of all the leaf to leaf paths explored so far.

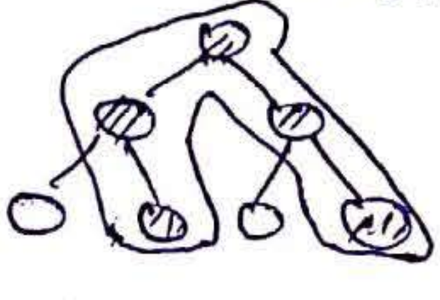
For any node return value is the maximum node to leaf path from current node to all leafes in its subtree

rv = max (lsum, rsum) + node.data.



or

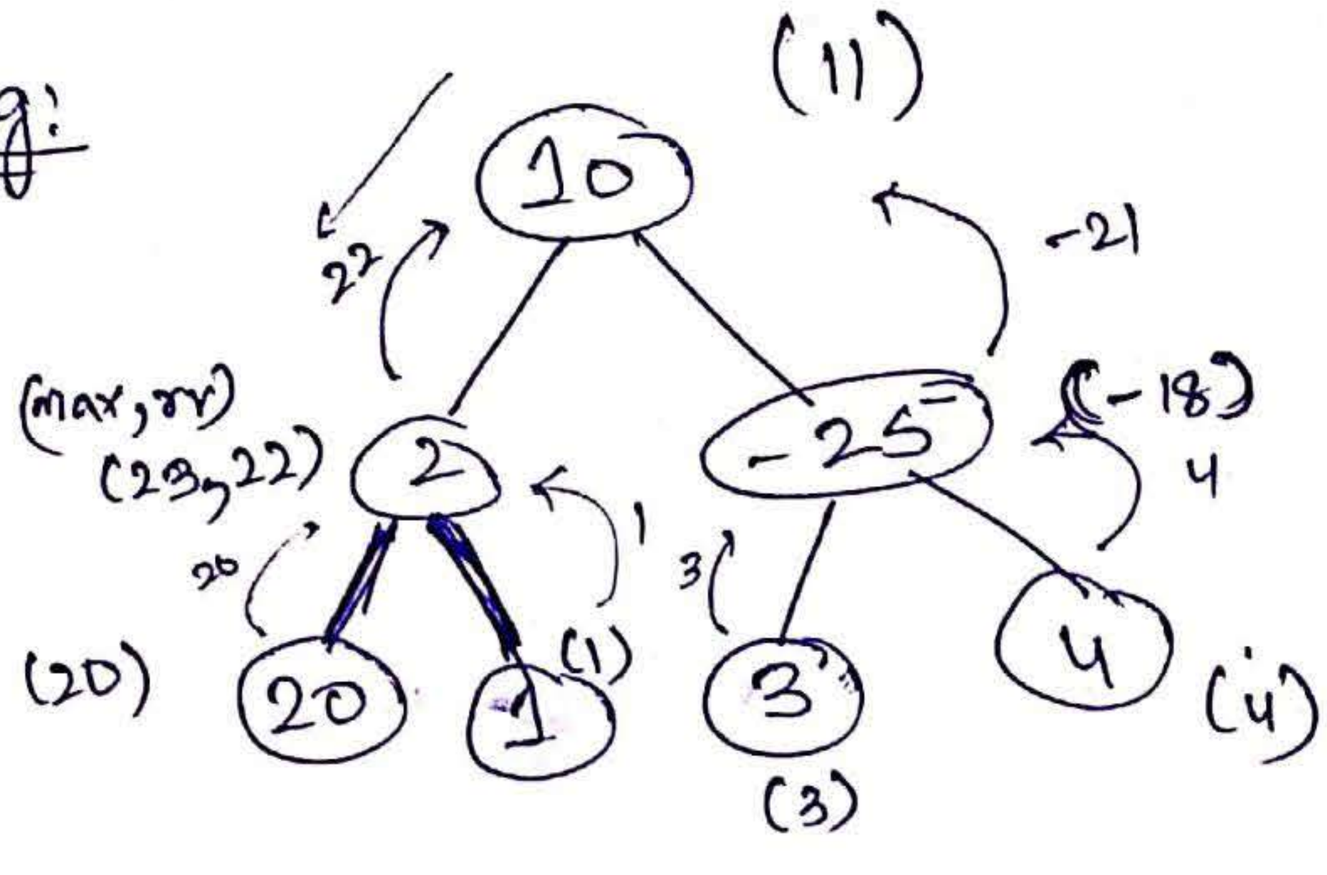
Max path will be compared to max leaf to leaf path from current node



max = max [lsum + rsum + node.data, max]

( ) → max calculated at that node

eg:



Ans = 23

max = ~~26~~ 23

MAXIMUM PATH SUM FROM ANY NODE

```
public int max = Integer.MIN_VALUE;
        public int findMaxSum(Node root) {
            maxPath(root);
            int res = max;
            max = Integer.MIN_VALUE;
            return res;

        }

        public int maxPath(Node root) {
            if (root == null) {
                return 0;
            }
            int ls = maxPath(root.left);
            int rs = maxPath(root.right);

            max = Math.max(max, root.data);
            max = Math.max(max, Math.max(ls + root.data, rs + root.data));
            max = Math.max(max, root.data + ls + rs);

            int rv = Math.max(root.data, Math.max(ls + root.data, rs + root.data));
            return rv;
        }
```

what ?

Given the root of a non empty binary Tree. Find the maximum path sum such that path may start and end at any node in the tree.

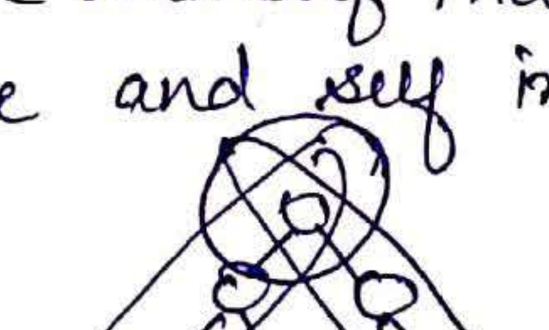
How ?

* We will keep a static variable max that will keep the maximum of all paths calculated so far. And at last result will be in max..

*For any node return value will be the maximum of all the paths that includes current node as the root.

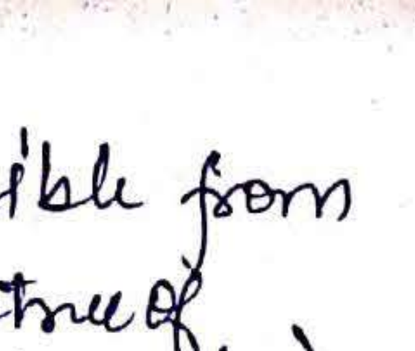
* Those possible paths are single node path (node.data), leftsubtree and self included path (ls + node.data) rightsubtree and self included path (rs + node.data)

eg:

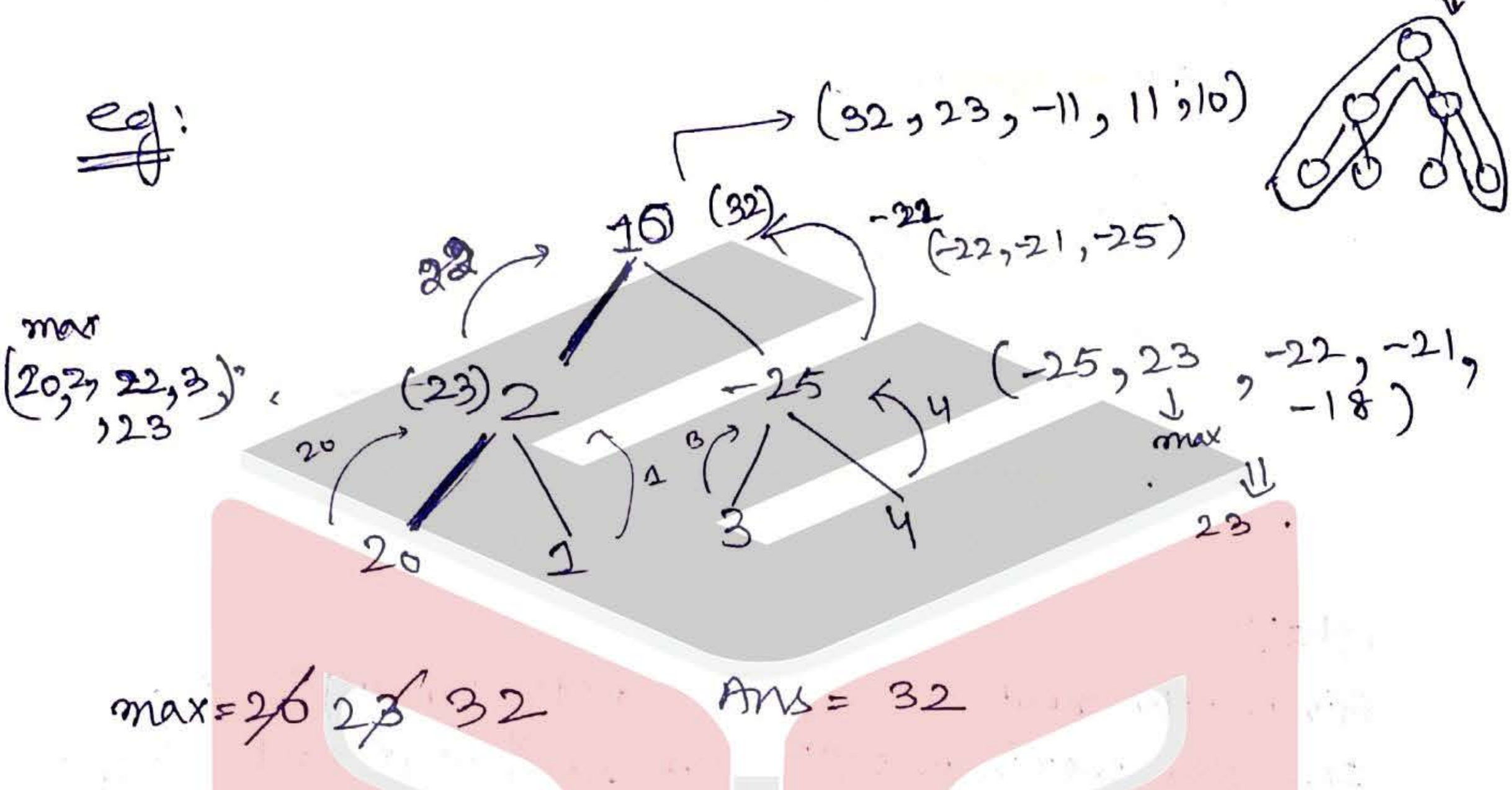


Maximum path will be all max paths possible from current node inclusive or exclusive (in subtree of current node)

max = maximum(max , node.val , node.val + ls + rs,
node.val + ls , node.val + rs)



eg:



max = ~~20~~ ~~23~~ 32

Ans = 32

BST TO GREATER SUM TREE

```
public static int sum = 0;
        public static TreeNodebstToGst(TreeNode root) {
                helper(root);
                sum = 0;
                return root;
        }
        public static void helper(TreeNode node) {
                if (node == null) {
                        return;
                }
                helper(node.right);
                node.val += sum;
                sum = node.val;
                helper(node.left);
        }
```

What?

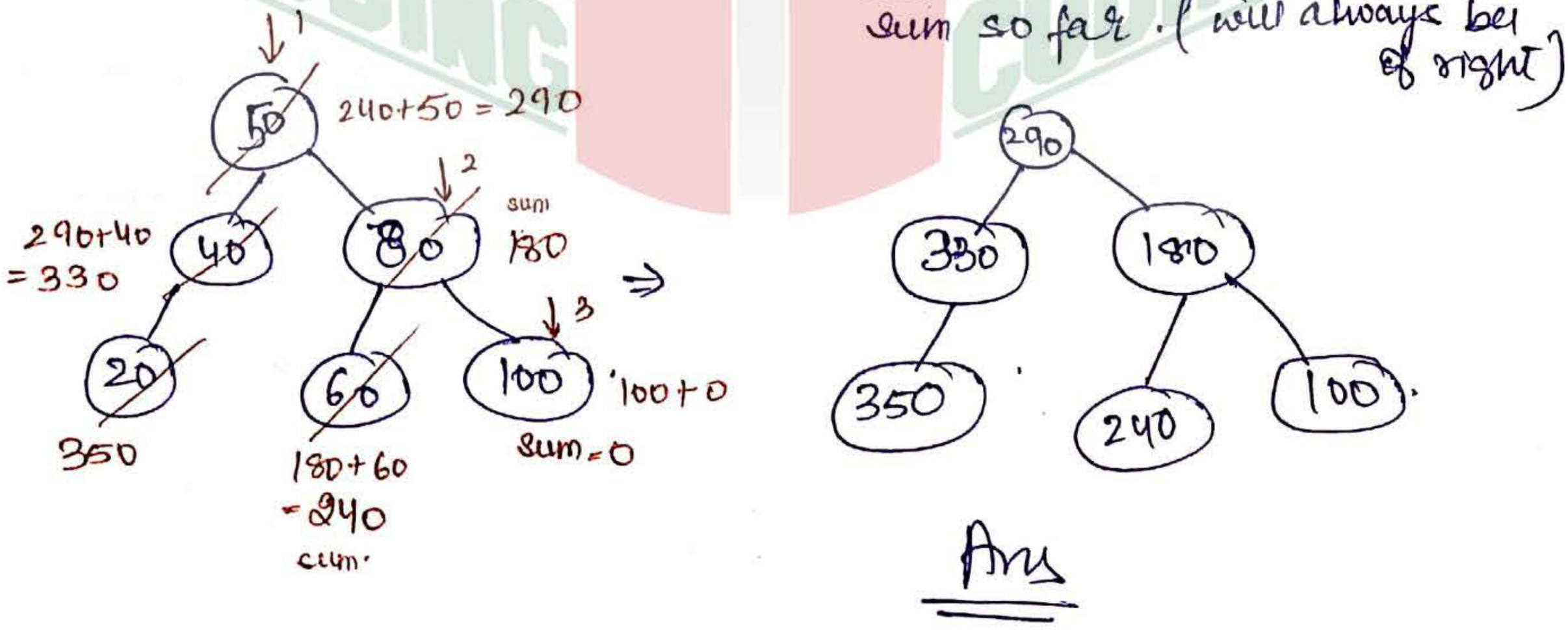
Given root of BST replace each node with sum of all the greater nodes from current node.

How?

We will follow reverse ~~post~~ in order to solve this problem as it is BST all the greater nodes from current nodes is in right.

curSum = node.val + sum

sum so far. (will always be of right)



Ans

PATH SUM

```
public static booleanhasPathSum(TreeNode root, int sum) {
        if (root == null)
            return false;
        return hasPathSumHelper(root, sum, root.val);
    }

    public static booleanhasPathSumHelper(TreeNode root, int sum, int mysum) {
        if (sum == mysum) {
            if (root.left == null &&root.right == null)
                return true;
        }
        boolean res1 = false, res2 = false;
        if (root.left != null)
            res1 = hasPathSumHelper(root.left, sum, mysum + root.left.val);
        if (root.right != null)
            res2 = hasPathSumHelper(root.right, sum, mysum + root.right.val);
        return res1 || res2;
    }
```

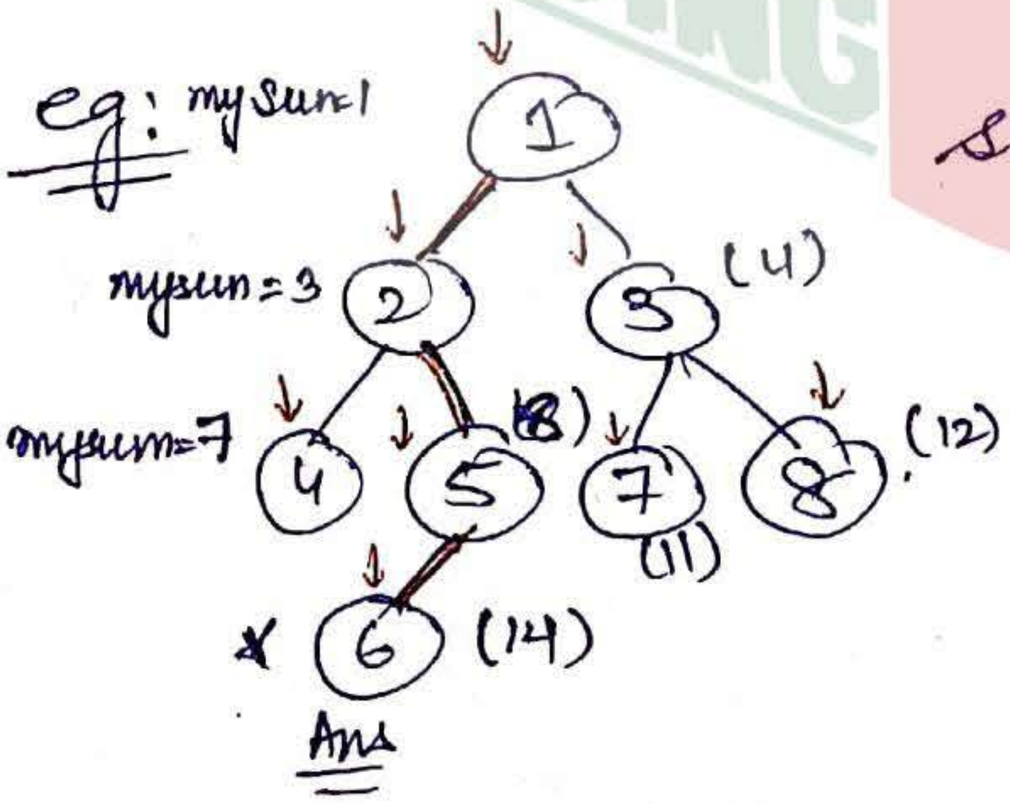
What?

Given root of a Binary Tree and a sum. Find whether a root to leaf path of given sum exist in the Tree or not.

How?

At each node add current node value to mysum (sum so far) If current mysum is equal to sum and node is leaf then ans is true.

eg: mySum=1



Sum = 14.

Ans = True

Ans

SUM TREE

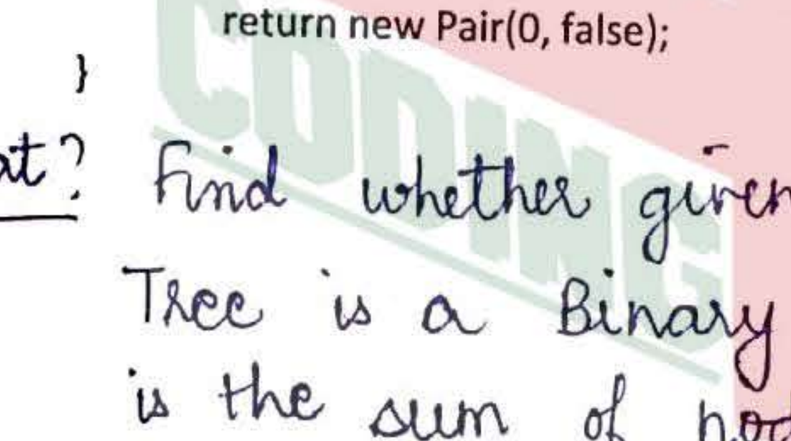
```
public static class Pair {
        int sum;
        booleanist;

        Pair(int s, booleanI) {
            sum = s;
            ist = I;
        }
    }

    public static booleanisSumTree(TreeNode node) {
        return isSumTreeh(node).ist;
}

public static Pair isSumTreeh(TreeNode root) {
    if (root == null) {
        return new Pair(0, true);
    }
    if (root.left == null &&root.right == null) {
        return new Pair(root.val, true);
    }
    Pair lr = isSumTreeh(root.left);
    if (lr.ist == false) {
        return new Pair(0, false);
    }
    Pair rr = isSumTreeh(root.right);
    if (rr.ist == false) {
        return new Pair(0, false);
    }
    if (lr.ist&&rr.ist&&root.val == lr.sum + rr.sum) {
        return new Pair(lr.sum + rr.sum + root.val, true);
    }
    return new Pair(0, false);
}
```


What? Find whether given tree is sum tree or not? A Sum Tree is a Binary Tree in which value of every node is the sum of nodes present in its left and right subtrees.

How? * A leaf node is always a sum tree.

* Return a pair having boolean issumtree (ist) and integer sum where sum is the sum returned by a node of its subtree.

ex:



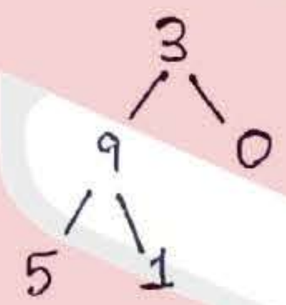
⇒ Ans = true with all nodes as sums of left and right subtrees.

SUM ROOT TO LEAF NUMBERS

```
public static int sumNumbers(TreeNode root) {
        return sum(root, 0, 0);
    }

    public static int sum(TreeNode root, int total, int mysum) {
        if (root == null)
            return total;
        if (root.left == null &&root.right == null) {
            mysum += root.val;
            total += mysum;
            return total;
        }
        mysum += root.val;
        total = sum(root.left, total, mysum * 10);
        total = sum(root.right, total, mysum * 10);
        return total;
    }
```

What? Given binary tree with each node having value b/w 0-9, each root to leaf path represents a number. find sum of all root-to-leaf numbers.

ex:



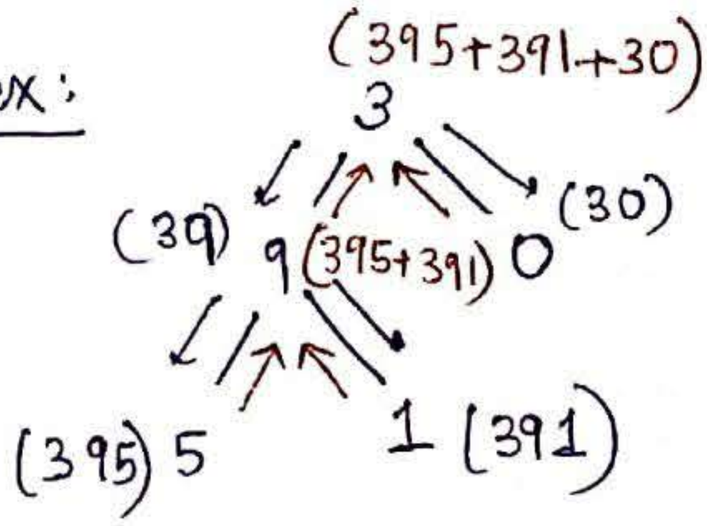
$= 395 + 391 + 30$

$= 816$

How?

* Keep two variables, total and mysum.
* Total adds the call values in it and mysum is used to create the number.
* From leaf, add leaf.data in mysum and mysum in total.
* Each call goes with mysum * 10.

ex:



node	mysum	total.
5	+5	5
1	+1	+1
0	+0	+0

leaf: +5 in mysum, +5 in total. if 5 is leaf.

BINARY TREE TILT

```
public int findTilt(TreeNode root) {
                helper(root);
                int res = tsum;
                tsum = 0;
                return res;
        }
        static int tsum = 0;
        public int helper(TreeNode root) {
                if (root == null) {
                        return 0;
                }
                int ls = helper(root.left);
                int rs = helper(root.right);
                tsum += Math.abs(ls - rs);
                return ls + rs + root.val;
        }
```

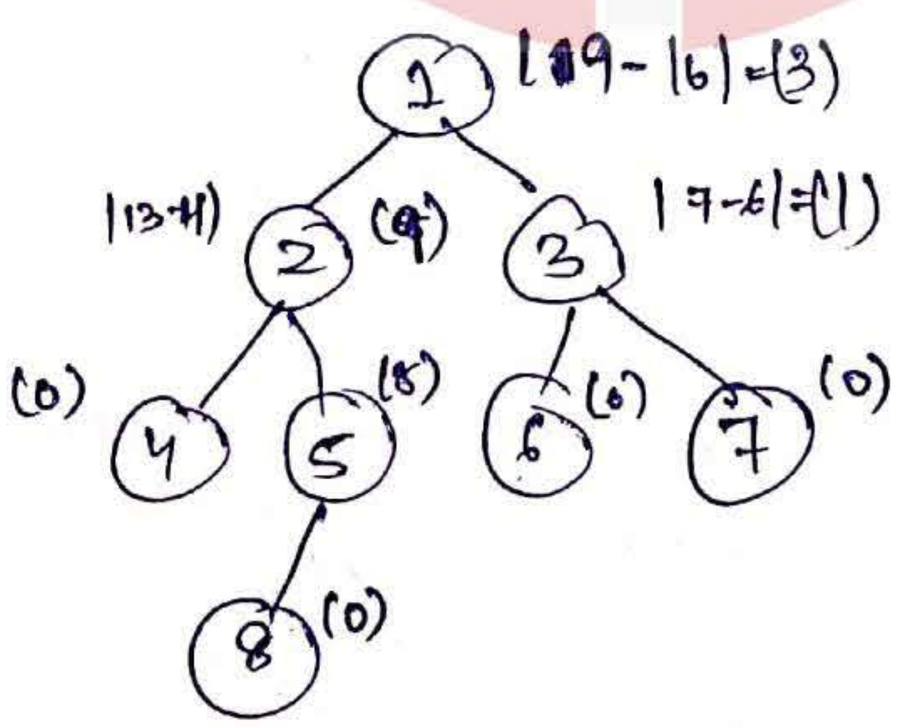
what?

Given root of a binary Tree . Find tilt of the tree where tilt is sum of absolute difference of sum of left subtree and right subtree. for each node.

How?

* Get the sum of left subtree (lsum) and right subtree (rsum) and calculate the difference and add to the tilt (tsum)

* Every nodes return value is the sum of whole tree rooted at current node.



tilt = 0+0+8+0+1+3
= 21

Ans = 21

PERFECT BINARY TREE

```
public static int findDepth(TreeNode root) {
        int d = 0;
        while (root != null) {
            d++;
            root = root.left;
        }
        return d;
    }

    public static booleanisPerfect(TreeNode root) {
        int d = findDepth(root);
        return isPerfectRec(root, d, 0);
}

public static booleanisPerfectRec(TreeNode root, int d, int level) {
        if (root.left == null &&root.right == null) {
            if (level != d - 1) {
                return false;
            }
            return true;
        }
        if (root.left == null || root.right == null) {
            return false;
        }
        return isPerfectRec(root.left, d, level + 1) &&isPerfectRec(root.right, d, level + 1);
}
```

<u>What?</u> Check whether the given tree is perfect binary tree or not? A perfect binary tree has all leaves at same depth (level) and each node, if an internal node, has both children present.

<u>How?</u> * Find depth of the tree.
* In Perfect tree function, check if each leaf is at same level as the depth of tree, return true, else return false.
* If any internal node has either child null, return false.

<u>ex:</u>

⇒ True (All leaves at level = 3).
⇒ All internal nodes have 2 children.

⇒ False.
⇒ 3 at level 2, 4,5 at level 3.

LOWEST COMMON ANCESTOR IN BINARY TREE

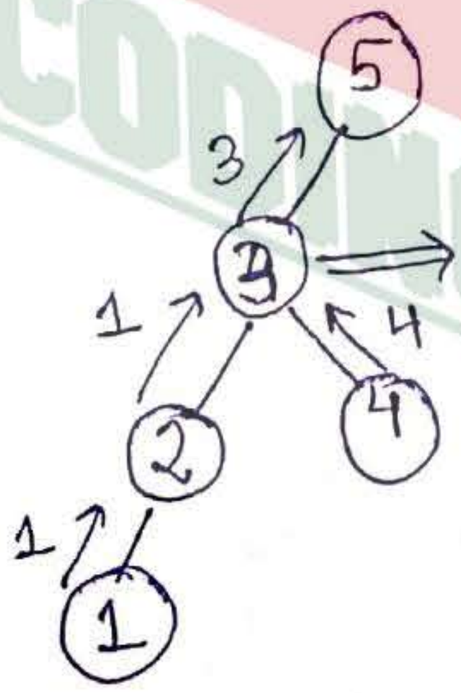
```
public TreeNodelca(TreeNode root, int n1, int n2) {
                    if (root == null)
                            return null;
                    if (root.data == n1 || root.data == n2)
                            return root;
                    TreeNodeleftlca = lca(root.left, n1, n2);
                    TreeNoderightlca = lca(root.right, n1, n2);
                    if (leftlca != null &&rightlca != null)
                            return root;
                    return (leftlca != null) ? leftlca : rightlca;
            }
```

What? Find the lowest common ancestor in a binary tree of given two nodes.

How? * Make left and right calls from each node, if the node is either of the two given nodes, return the node itself.

* Check both leftlca and rightlca, if both are not null, that means you found both, so return current node.
* If leftlca is not null, return leftlca, else return rightlca.

ex:



n1 = 1, n2 = 4

At 3, both leftlca (1) and rightlca (4) are not null, hence 3 is the lca of (1,4).

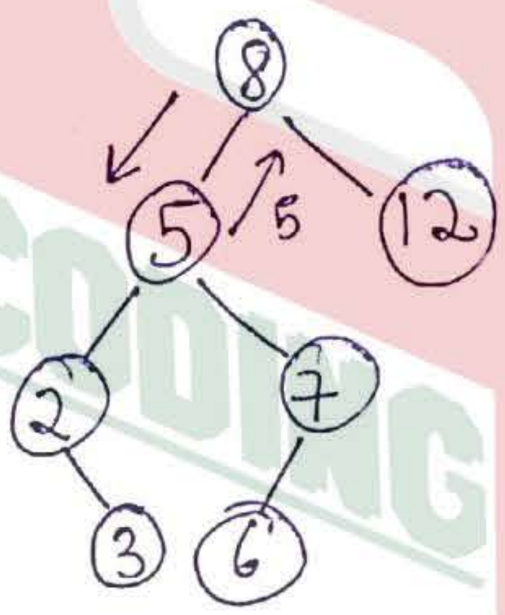
LOWEST COMMON ANCESTOR IN BST

```
TreeNode LCA(TreeNode node, int n1, int n2) {
        if ((node.data>= n1 &&node.data<= n2) || (node.data>= n2 &&node.data<= n1)) {
                return node;
        }
        if (node.data>= n1 &&node.data>= n2) {
                return LCA(node.left, n1, n2);
        }
        if (node.data<= n1 &&node.data<= n2) {
                return LCA(node.right, n1, n2);
        }
        return null;
}
```

What? Find the lowest common ancestor in a BST of two given nodes.

How? * In a BST, check if a node data is greater than both n1 and n2, lca lies in left, if it is less than both n1 and n2, then it lies in right.
* For all other possibility, return the node.

ex:



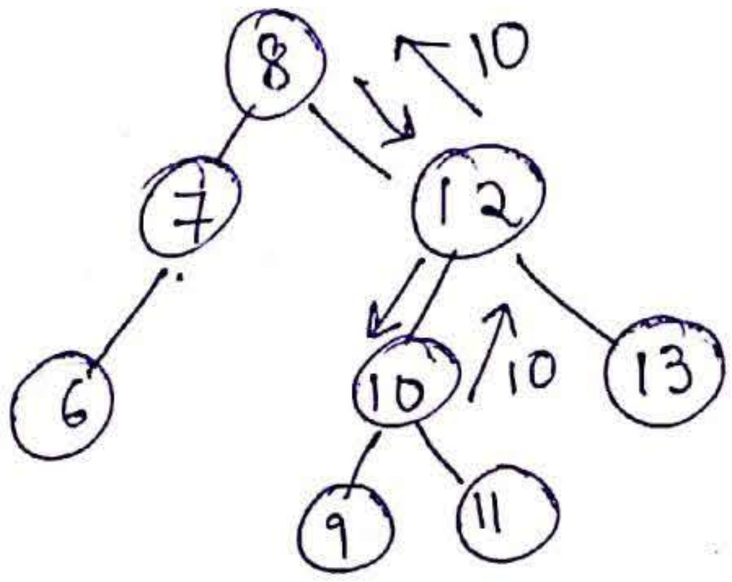
n1 = 3, n2 = 6.

a) 8 > 3 && 8 > 6, move left.

b) 5 > 3 && 5 < 6, in range, 5 is lca, return 5.

ans = 5.

ex:



n1 = 9, n2 = 11.

a) 8 < 9 && 8 < 11, move right.

b) 12 > 9 && 12 > 11, move left.

c) 10 > 9 && 10 < 11, in range, return 10 as 10 is lca.

ans = 10

BST ITERATOR

```
static class BST_Iterator {
        LinkedList<TreeNode>st = new LinkedList<>();
        public BST_Iterator(TreeNode root) {
                pushAll(root);
        }
        private void pushAll(TreeNode n) {
                while (n != null) {
                        st.addFirst(n);
                        n = n.left;
                }
        }
        /**
         * @return the next smallest number
         */
        public int next() {
                TreeNode n = st.removeFirst();
                pushAll(n.right);
                return n.val;
        }
        /**
         * @return whether we have a next smallest number
         */
        public booleanhasNext() {
                return st.size() != 0;
        }
}
```

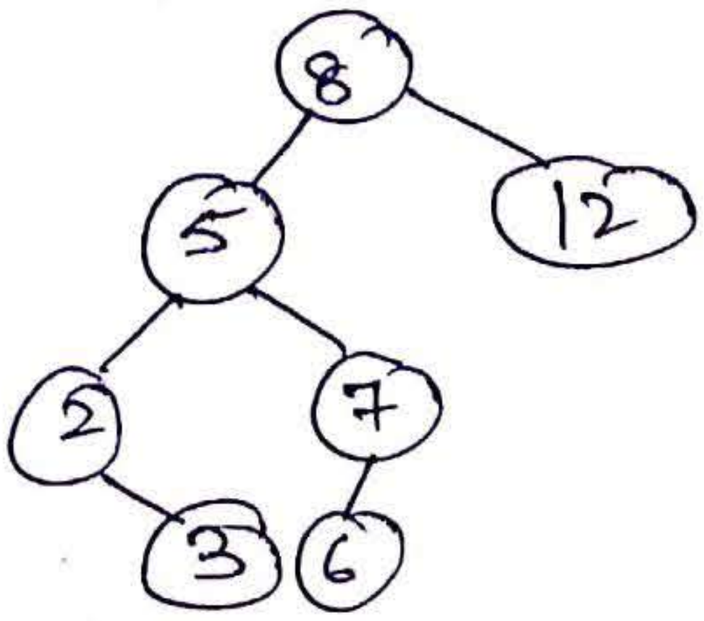
What?

Implement an Iterator over a BST. In this Iterator calling next() will give the next smallest number. and function hasNext() returns whether we have next smaller number or not.

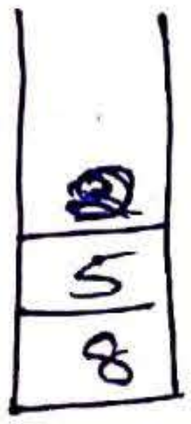
How?

* In this we will keep a stack to track the next smallest.
* since it is a BST nodes at left will be smaller than nodes at right.
* So initially in stack All the left nodes are added.

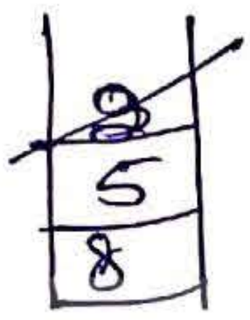
eg:



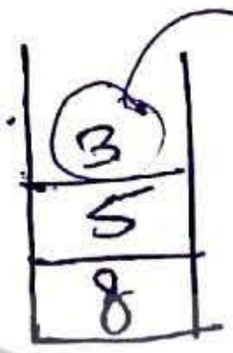
Initially



1) next() →



Print 8 .



⇒ 3 is added

At the right children are less than root but greater than printed node so they are added.

2) next() →

Print 3 . (no right nodes)

(Push all will add all left nodes of right)

3) next() →

Print 5

⇒

(All nodes to the left of the current right are added)

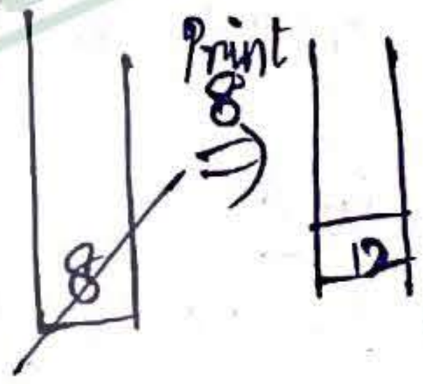
4) next()

Print 6

5) next()

Print 7

(6) next()



7) next() ⇒

Print 12 .

Ans 2 3 5 6 7 12 .

BOUNDARY OF BINARY TREE

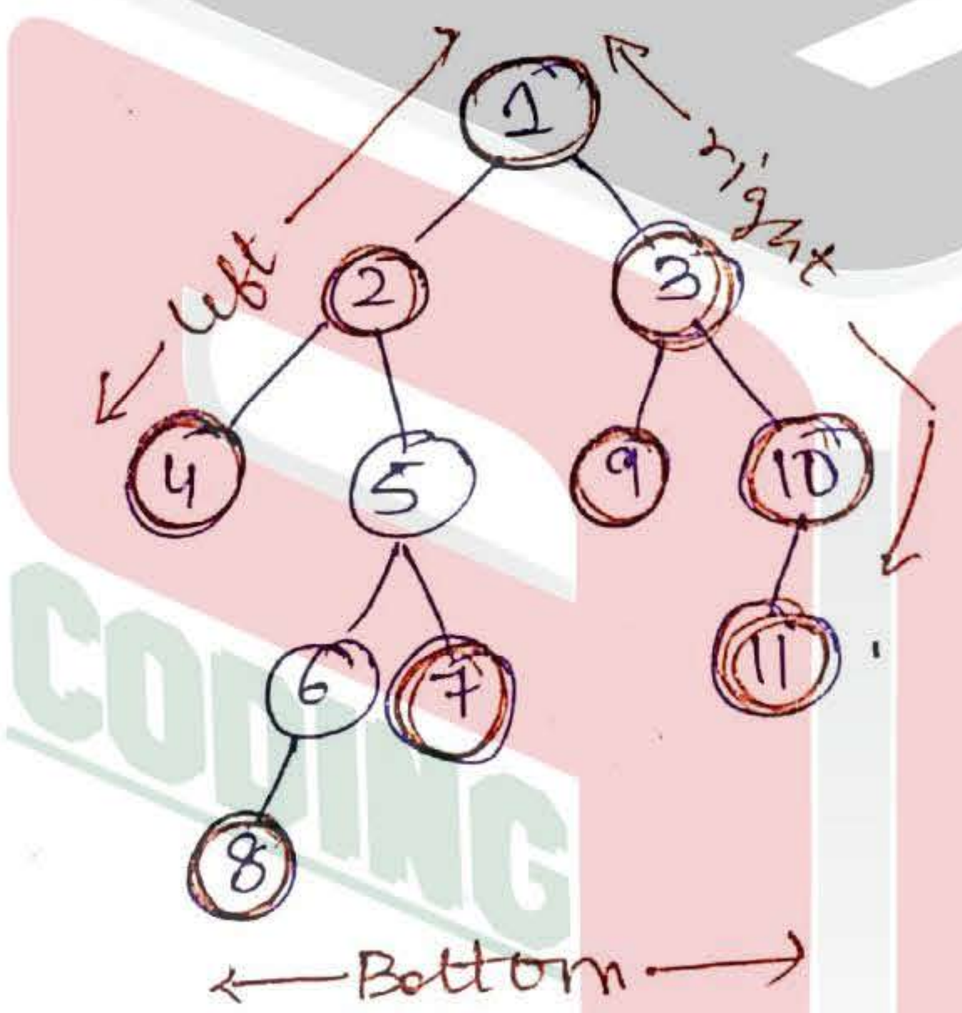
```
public List<Integer>boundaryOfBinaryTree(TreeNode root) {
                List<Integer>ans = new LinkedList<Integer>();
                if (root == null)
                        return ans;
                ans.add(root.data);
                leftBoundry(ans, root.left);
                rightBoundry(ans, root.right);
                return ans;
        }

        public void leftBoundry(List<Integer>ans, TreeNode root) {
                if (root == null)
                        return;
                ans.add(root.data);
                if (root.left != null) {
                        leftBoundry(ans, root.left);
                        bottomBoundry(ans, root.right);
                } else {
                        leftBoundry(ans, root.right);
                }
}

public void bottomBoundry(List<Integer>ans, TreeNode root) {
        if (root == null)
                return;
        // if its a leaf add it to ans
        if (root.left == null &&root.right == null) {
                ans.add(root.data);
        } else {
                bottomBoundry(ans, root.left);
                bottomBoundry(ans, root.right);
        }
}

public void rightBoundry(List<Integer>ans, TreeNode root) {
        if (root == null)
                return;
        if (root.right != null) {
                bottomBoundry(ans, root.left);
                rightBoundry(ans, root.right);
        } else {
                rightBoundry(ans, root.left);
        }
        ans.add(root.data);
}
```

What?

Given root of a Binary Tree Return the values of its boundary Anticlockwise direction starting from root. Boundary includes left boundary, leaves and right boundary in order without duplicate nodes.

How?

* Left Boundary is the nodes in the path of left most leaf of the Binary Tree.
* Right Boundary is the nodes in the path of right most leaf of the Binary Tree.
* Bottom Boundary is all the leafs of the tree
* Since we want anticlockwise order we will add left boundary nodes in preorder. and right boundary is post order.



Ans = [1 2 4 8 7 9 11 10 3]

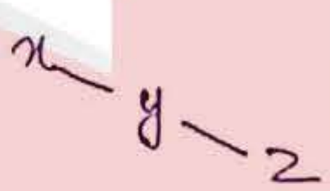
AVL TREE

```
public class Node {
		int data;
		Node left;
		Node right;
		int ht;
		int bal;
	}

	private Node root;

	private int getHeight(Node node) {
		if (node == null) {
			return 0;
		}
		int lht = node.left != null ? node.left.ht : 0;
		int rht = node.right != null ? node.right.ht : 0;
		return Math.max(lht, rht) + 1;
}

private int getBalance(Node node) {
		if (node == null) {
			return 0;
		}
		int lht = node.left != null ? node.left.ht : 0;
		int rht = node.right != null ? node.right.ht : 0;
		return lht - rht;
}
public void addNode(int val) {
	root = addNode(root, val);
	}

	private Node addNode(Node node, int val) {
		if (node == null) {
			Node bn = new Node();
			bn.data = val;
			bn.ht = 1;
			bn.bal = 0;
			return bn;
		}
		if (val>node.data) {
			node.right = addNode(node.right, val);
		} else if (val<node.data) {
			node.left = addNode(node.left, val);
		}
node.ht = getHeight(node);
node.bal = getBalance(node);

if (node.bal> 1) {
	// left area
	if (getBalance(node.left) > 0) {
		// L-L
		node = rightRotate(node);
	} else {
		// L-R
```

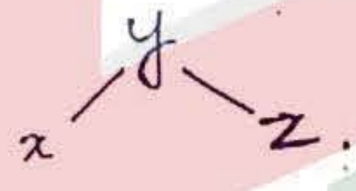
```
                node.left = leftRotate(node.left);
                node = rightRotate(node);
            }
        } else if (node.bal< -1) {
            // right area
            if (getBalance(node.right) < 0) {
                // R-R
                node = leftRotate(node);
            } else {
                // R-L
                node.right = rightRotate(node.right);
                node = leftRotate(node);
            }
        }
        return node;
    }

    private Node leftRotate(Node x) {
        Node y = x.right;
        x.right = y.left;
        y.left = x;

        x.ht = getHeight(x);
        x.bal = getBalance(x);
        y.ht = getHeight(y);
        y.bal = getBalance(y);

        return y;
    }

    private Node rightRotate(Node x) {
        Node y = x.left;
        x.left = y.right;
        y.right = x;
        x.ht = getHeight(x);
        x.bal = getBalance(x);
        y.ht = getHeight(y);
        y.bal = getBalance(y);
        return y;
    }

    public void remove(int val) {
        root = remove(root, val);
    }
    private Node remove(Node node, int val) {
        if (node == null) {
            return null;
        }
        if (val>node.data) {
            node.right = remove(node.right, val);
        } else if (val<node.data) {
            node.left = remove(node.left, val);
        } else {
            if (node.left == null &&node.right == null) {
                return null;
            } else if (node.left == null) {
                node = node.right;
```

```
            } else if (node.right == null) {
                  node = node.left;
            } else {
                  int lmax = max(node.left);
                  node.data = lmax;
                  node.left = remove(node.left, lmax);
            }
      }
      node.ht = getHeight(node);
      node.bal = getBalance(node);
      if (node.bal> 1) {
            // left area
            if (val<node.left.data) { // getBalance(node.left) > 0
                  // L-L
                  node = rightRotate(node);
            } else {
                  // L-R
                  node.left = leftRotate(node.left);
                  node = rightRotate(node);
            }
      } else if (node.bal< -1) {
            // right area
            if (val>node.right.data) { // getBalance(node.right) < 0
                  // R-R
                  node = leftRotate(node);
            } else {
                  // R-L
                  node.right = rightRotate(node.right);
                  node = leftRotate(node);
            }
      }
      return node;
}
```

Left Rotation :



⇒

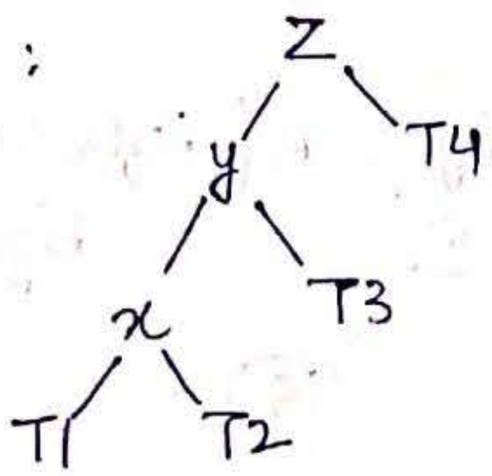


Right Rotation :

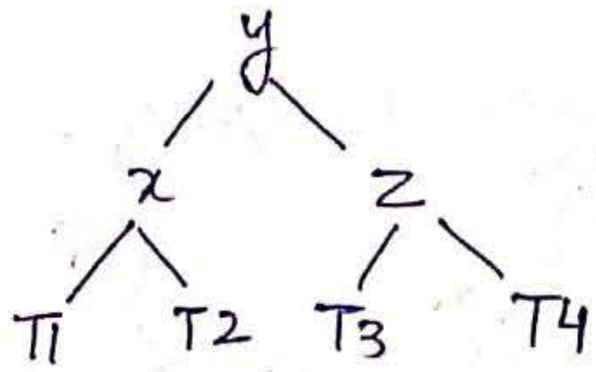
⇒

4 Cases :

1) L-L :

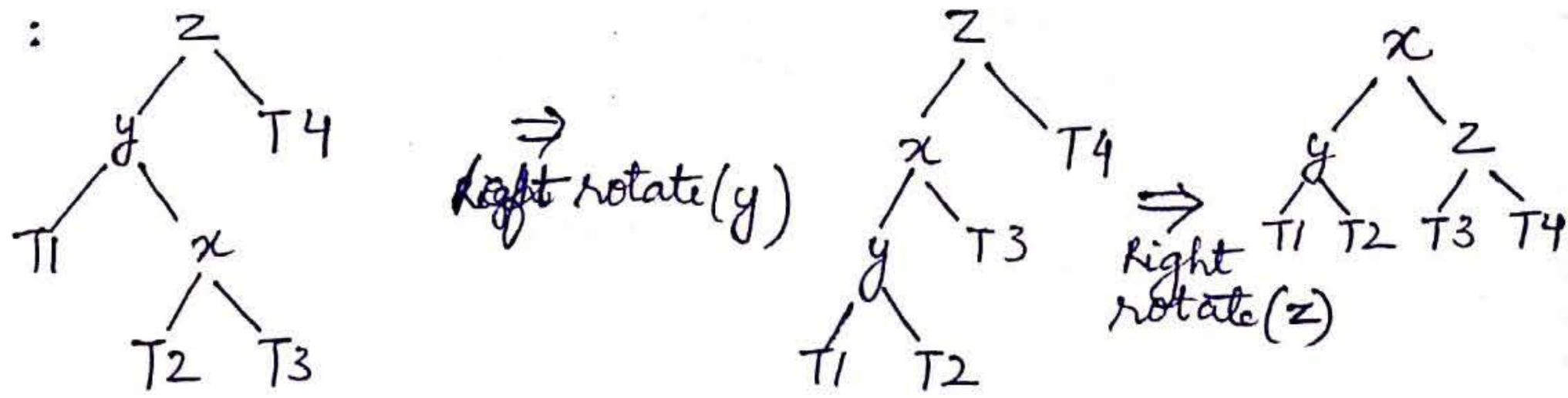


right Rotate (z) ⟶

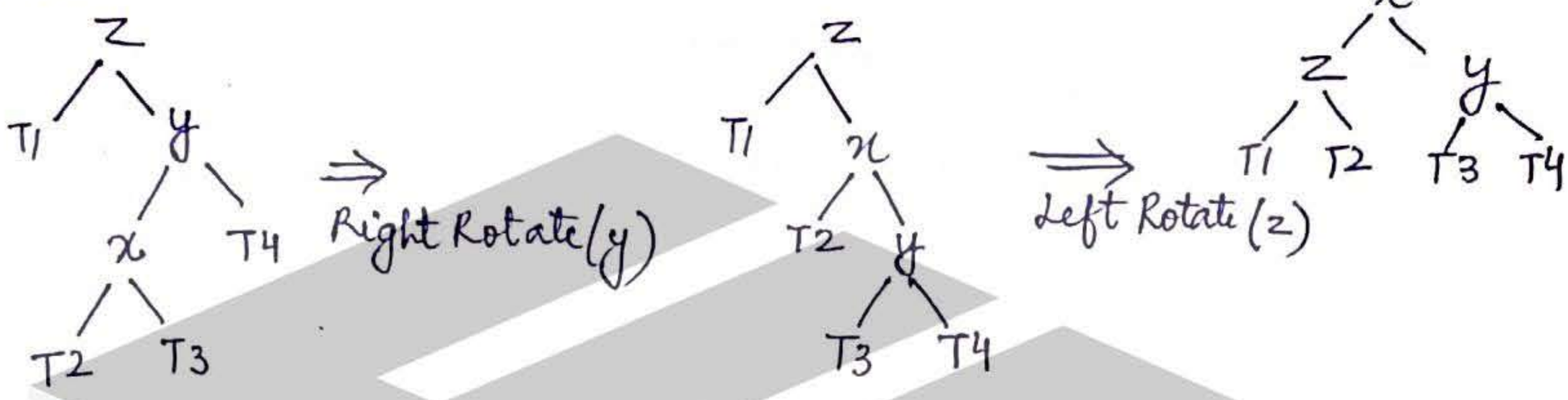


* z goes on right of y, y.right becomes z.left.

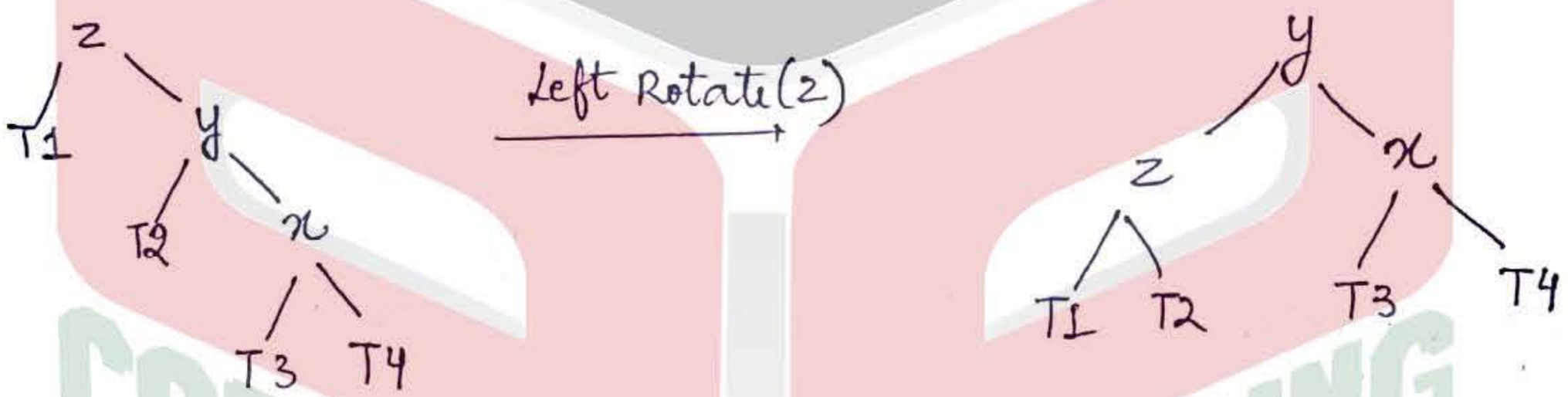
2) L-R :



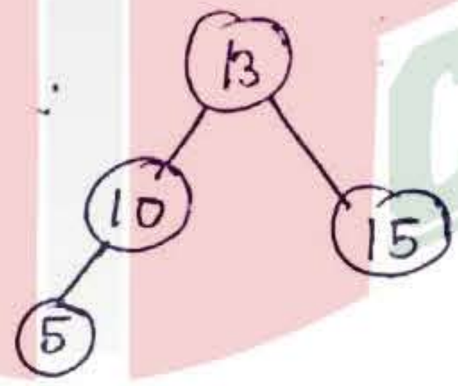
3) R-L:



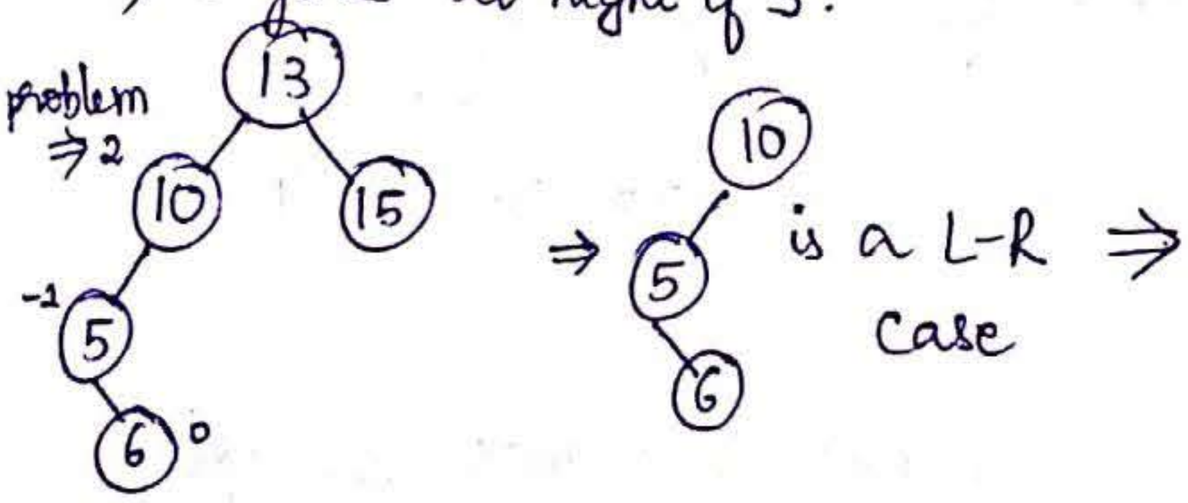
4) R-R :



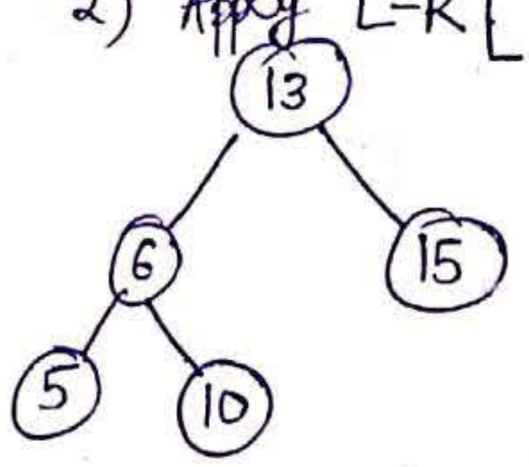
ex: Insertion: Insert (6) in :



1) 6 goes at right of 5.



2) Apply L-R [leftrotate(5), then RightRotate(10)]



ITERATOR

```
private class Pair {
        Node node;
        int counter = 0;
    }
    @Override
    public Iterator<Integer> iterator() {
        return new BTPOIterator(root);
    }
    private class BTPOIterator implements Iterator<Integer> {
        Node next;
Stack<Pair>st;
public BTPOIterator(Node root) {
        st = new Stack<>();
        Pair rootp = new Pair();
        rootp.node = root;
        st.push(rootp);
        moveNext();
}
@Override
public booleanhasNext() {
        return next != null;
}
@Override
public Integer next() {
        if (hasNext() == false) {
                throw new NoSuchElementException();
        }
        Node ret = next;
        next = null;
        moveNext();
        return ret.data;
}
private void moveNext() {
        while (st.size() > 0) {
                Pair top = st.peek();
                if (top.counter == 0) {
                        top.counter++;
                        if (top.node.left != null) {
                                Pair leftp = new Pair();
                                leftp.node = top.node.left;
                                st.push(leftp);
                        }
                } else if (top.counter == 1) {
                        top.counter++;
                        if (top.node.right != null) {
                                Pair rightp = new Pair();
                                rightp.node = top.node.right;
                                st.push(rightp);
                        }
} else if (top.counter == 2) {
        top.counter++;
        next = top.node;
        break;
} else {
```

```
                    st.pop();
                }
            }
        }
    }
```

What?

Implement Post order Iterator for Binary tree. In this hasNext() funtion returns whether a next element exist or not. Next() function returns the next post oorder element.

How?

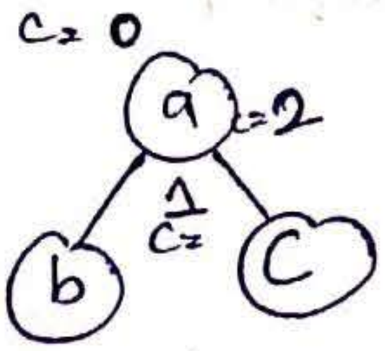
* We will implement this using stack and keeping track of state of each node.
* Pair object stores current node and the state of current node(counter). state denote following things

counter = 0 → element is just pushed in stack and is in preorder.
↳ At this state increment the counter and add left child.

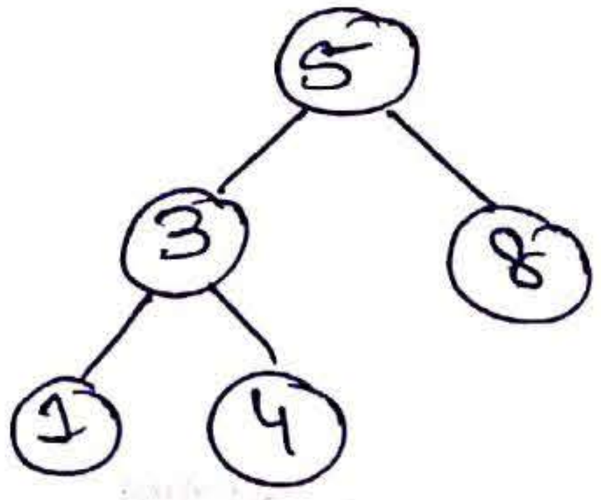
Counter = 1 → element(Node) has pushed its left child and is in Inorder.
↳ At this state increment the counter and add the right child.

counter = 2 → Node has pushed its right child and is in postorder. Set next and return
↳ Increment counter.

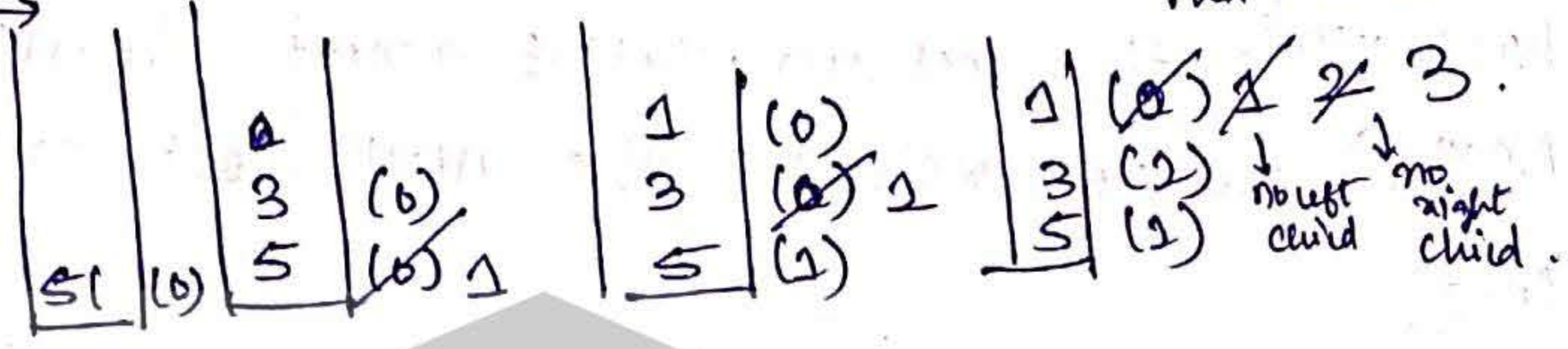
counter > 2 → pop from stack all the statements are executed



eg:

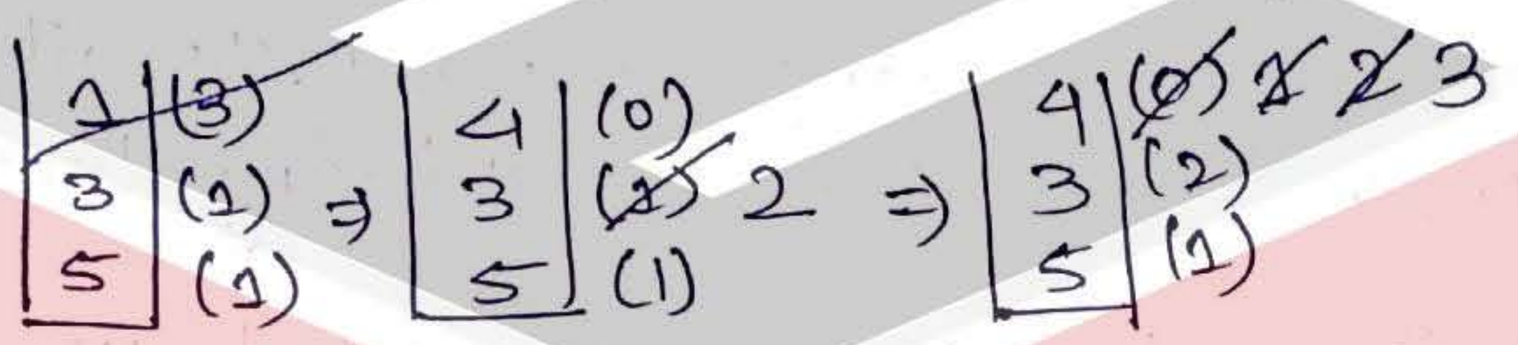


① next() → next = 1



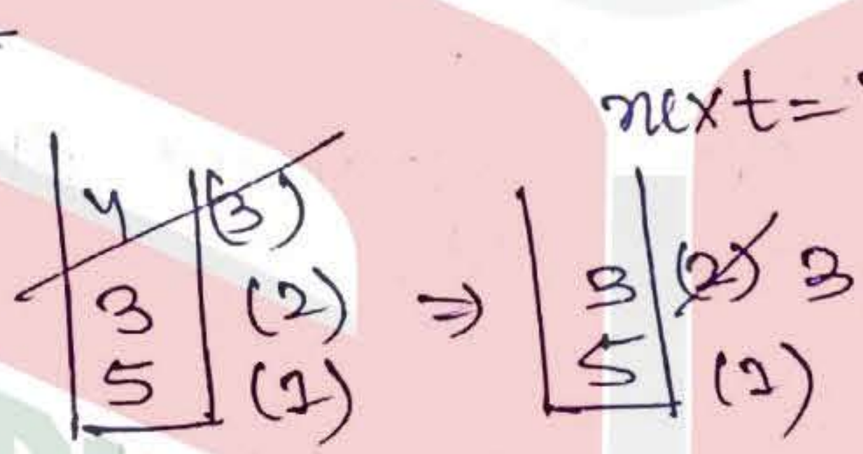
print 1

② next() next = 4



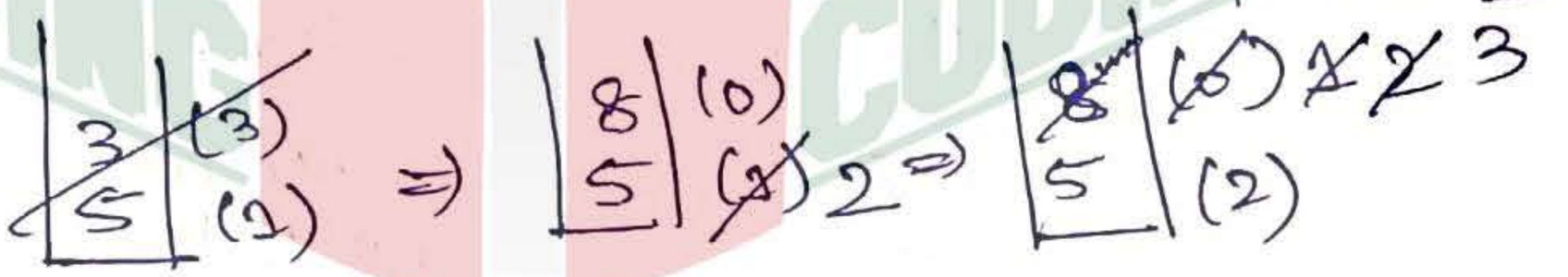
print 4

③ next() next = 3.



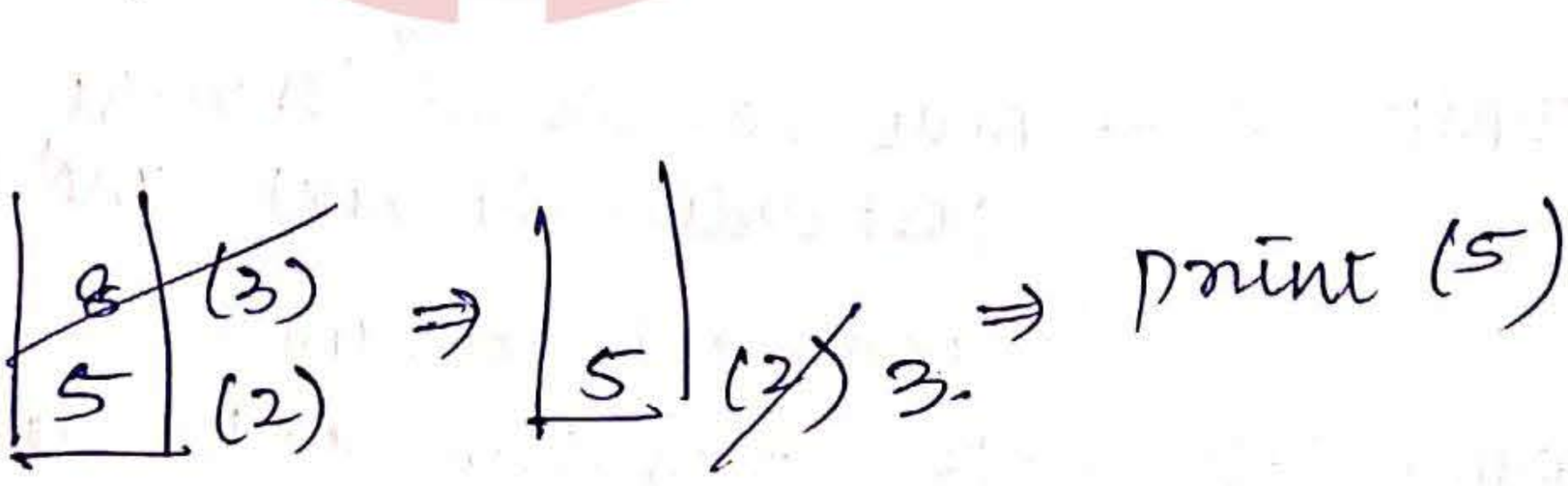
print 3

④ next() next = 8



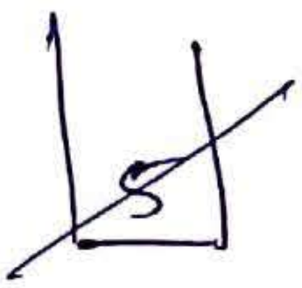
print 8

⑤ next()



⇒ print (5)

⑥ next() → no next available



TARGET SUM

```
private class Pair {
		Node node;
		int wc;
		Pair(Node node) {
			this.node = node;
			this.wc = 0;
		}
	}
	public void printTSPOHSpace(int tar) {
		Stack<Pair> ls = new Stack<>();
		Stack<Pair>rs = new Stack<>();
ls.push(new Pair(root));
rs.push(new Pair(root));
while (ls.size() > 0 &&rs.size() > 0) {
		Pair ltop = traversalFromLeft(ls);
		Pair rtop = traversalFromRight(rs);
		if (ltop != null &&rtop != null) {
			System.out.print(ltop.node.data + " " + rtop.node.data);
			if (ltop.node.data + rtop.node.data> tar) {
				rtop.wc++;
			} else if (ltop.node.data + rtop.node.data< tar) {
				ltop.wc++;
			} else {
				if(ltop.node.data<rtop.node.data){
					System.out.print("*");
				}
				ltop.wc++;
				rtop.wc++;
			}
			System.out.println();
		}
	}
}

private Pair traversalFromLeft(Stack<Pair> ls){
	while (ls.size() > 0) {
		Pair ltop = ls.peek();
		if (ltop.wc == 0) {
			if (ltop.node.left != null) {
				ls.push(new Pair(ltop.node.left));
			}
		} else if (ltop.wc == 1) {
			return ltop;
		} else if (ltop.wc == 2) {
			if (ltop.node.right != null) {
				ls.push(new Pair(ltop.node.right));
			}
		} else {
			ls.pop();
		}
		ltop.wc++;
	}
	return null;
}
```

```
private Pair traversalFromRight(Stack<Pair>rs){
        while (rs.size() > 0) {
                Pair rtop = rs.peek();
                if (rtop.wc == 0) {
                        if (rtop.node.right != null) {
                                rs.push(new Pair(rtop.node.right));
                        }
                } else if (rtop.wc == 1) {
                        return rtop;
                } else if (rtop.wc == 2) {
                        if (rtop.node.left != null) {
                                rs.push(new Pair(rtop.node.left));
                        }
                } else {
                        rs.pop();
                }
                rtop.wc++;
        }
        return null;
}
```

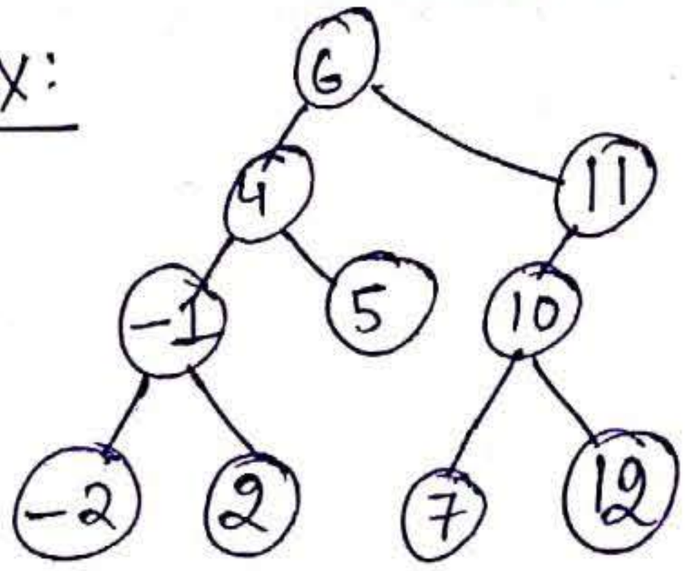
What? Print all pairs in a binary tree where sum of the pair is equal to given target.

How? We use 2 functions :

1) Left traversal function which returns the first inorder element initially, and then subsequent inorder elements.
2) Right traversal function which returns the first inorder element from right (in reverse order) and then its previous inorder elements reversely.

* For a target sum pair, we check that if the sum of the pair returned by 2 functions is greater than target, then right traversal returns other inorder element, as for ex: if $2+8=10 >$ target, then reduce 10 to get answer. Similarly, if $2+8=10 <$ target, then left traversal returns next inorder element to get a better answer.

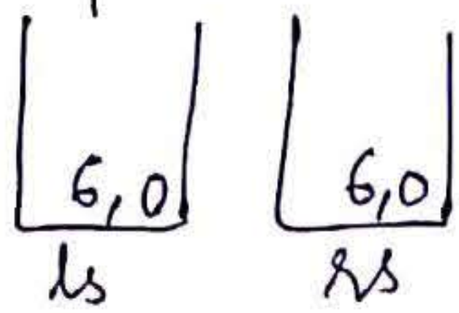
ex:

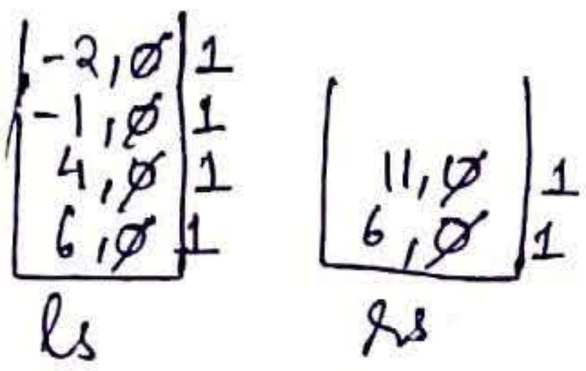


target = 9.

1) ltop = leftTraversal, rstop = rightTraversal.

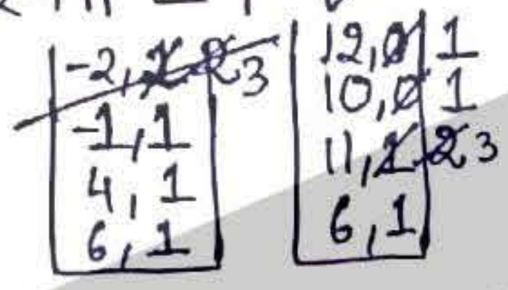
* Keep two stacks, with root initially.



→ Ist Pair: 

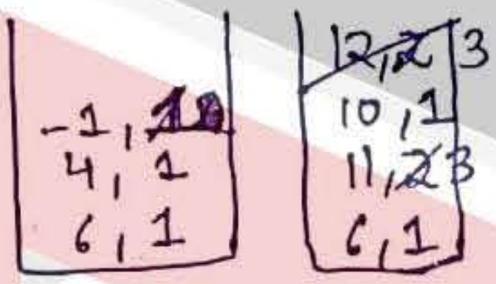
⇒ ls has top = -2, rs has top = 11.

$-2+11=9$ ✓ Print [-2, 11] ⇒ rs.top.wc++, ls.top.wc++.

→ IInd Pair 

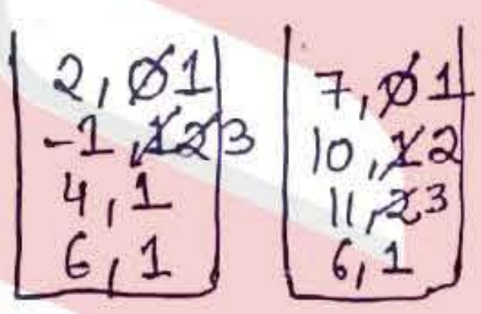
⇒ ls has top = -1, rs has top = 12.

$-1+12>9$ ⇒ rs.top.wc++.

→ IIIrd Pair 

⇒ ls has top = -1, rs has top = 10.

$-1+10=9$ ✓ Print [-1, 10] ⇒ rs.top.wc++ ls.top.wc++.

→ IVth Pair 

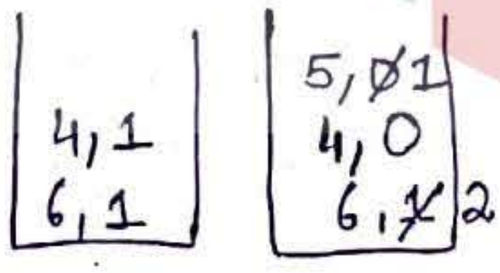
⇒ ls has top = 2, rs has top = 7.

$2+7=9$ ✓ Print [2, 7] ⇒ rs.top.wc++ ls.top.wc++.

→ Vth Pair 

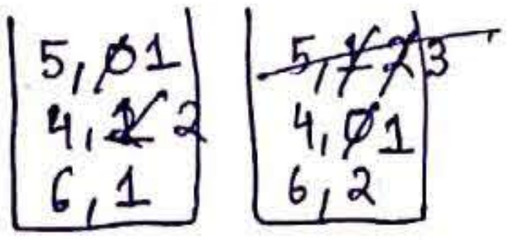
⇒ ls has top = 4, rs has top = 6.

$4+6=10>9$, rs.top.wc++.

→ VIth Pair 

⇒ ls has top = 4, rs has top = 5.

$4+5=9$ ✓ Print [4, 5] ⇒ rs.top.wc++ ls.top.wc++.

→ VIIth Pair 

⇒ To avoid duplicacy, we stop here, else we get same pairs reversely repeated.

Ans: [(-2, 11), (-1, 10), (2, 7), (4, 5)].