**Question 1: In Order Traversal**

Given a binary tree, return the inorder traversal of its nodes' values.

**NOTE:** Using recursion is not allowed.

**Problem Constraints**

1 <= number of nodes <= 105

Logic –

* We will do the recursive calls as shown below.
* inorderprint(root->left);
* print(root->val);
* inorderprint(root->right);
* Instead of printing the value we can pass a Arraylist as a parameter to another function (helper) and append the A.val in it.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public void helper(TreeNode A, ArrayList ans) {

        if (A==null) return;

        helper(A.left, ans);

        ans.add(A.val);

        helper(A.right, ans);

    }

    public int[] inorderTraversal(TreeNode A) {

        ArrayList<Integer> temp = new ArrayList<Integer>();

        helper(A, temp);

        int[] ans = new int[temp.size()];

        for (int i=0;i<temp.size();i++) {

            ans[i] = temp.get(i);

        }

        return ans;

    }

}

Note – Pre order and Post order traversal are similar to the above code, the only difference is the order of recursive calls.

**Order for Preorder Order for Post order**

print(root->val); postorderprint(root->left);

preorderprint(root->left); postorderprint(root->left);

preorderprint(root->right); print(root->val);

**Question 2: Level Order Traversal**

Given a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

**Problem Constraints**

1 <= number of nodes <= 105

Logic – Start by adding the root node and a null value ( The null value will represent the end of a level. ) to a queue and perform the below steps:

* Get the front value of the queue in a variable (x).
* Check if x is null or not. If it is null then it means one level is over, so we will add a null value to the last of the queue and continue to step 1.
* If x is not null then we will add the left and the right nodes of x to the end of the queue.
* Finally remove the front value of the queue.
* Repeat this process till the length of the queue becomes 1 (At last only a null value will be left in the queue).

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public ArrayList<ArrayList<Integer>> levelOrder(TreeNode A) {

        Queue<TreeNode> q = new LinkedList<TreeNode>();

        q.add(A);

        q.add(null);

        ArrayList<ArrayList<Integer>> ans = new ArrayList();

        ArrayList<Integer> temp = new ArrayList();

        int maxSize = 0;

        while(q.size() > 1) {

            TreeNode x = q.peek();

            // One level is completed

            if (x==null) {

                q.add(null);

                q.remove();

                ans.add(new ArrayList<>(temp));

                temp.clear();

                continue;

            }

            temp.add(x.val);

            maxSize = Math.max(maxSize, temp.size());

            if(x.left != null) q.add(x.left);

            if(x.right != null) q.add(x.right);

            q.remove();

        }

        return ans;

    }

}

**Question 3: Serialize Binary Tree**

Given the **root node**of a Binary Tree denoted by **A**. You have to Serialize the given Binary Tree in the described format.

Serialize means encode it into a **integer array** denoting the **Level Order Traversal** of the given Binary Tree.

**NOTE:**

 In the array, the NULL/None child is denoted by -1.

 For more clarification check the Example Input.  
  
**Problem Constraints**

1 <= number of nodes <= 105

Logic – Similar to Level order traversal, only difference is to add -1 if left node or the right node of the front value of the queue is null. Another difference is to add -1 to the answer and remove it from the queue whenever a -1 is found in the queue.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public ArrayList<Integer> solve(TreeNode A) {

        Queue<TreeNode> q = new LinkedList<TreeNode>();

        ArrayList<Integer> ans = new ArrayList();

        q.add(A);

        q.add(null);

        while(q.size() > 1) {

            TreeNode x = q.peek();

            if (x == null) {

                q.add(null);

                q.remove();

                continue;

            }

            if (x.val == -1) {

                ans.add(x.val);

                q.remove();

                continue;

            }

            ans.add(x.val);

            TreeNode y = new TreeNode(-1);

            if(x.left != null) q.add(x.left);

            else q.add(y);

            if(x.right != null) q.add(x.right);

            else q.add(y);

            q.remove();

        }

        return ans;

    }

}

**Question 4: Deserialize Binary Tree**

You are given an integer array **A** denoting the **Level Order Traversal** of the Binary Tree.

You have to Deserialize the given Traversal in the Binary Tree and return the **root** of the Binary Tree.

**NOTE:**

* In the array, the NULL/None child is denoted by -1.
* For more clarification check the Example Input.

**Problem Constraints**

1 <= number of nodes <= 105

Logic – We can do this simply by using a queue data structure.

* We know that the root node will always be the first element of level order traversal.
* Create a root node and push the root node into the queue.Now, run a loop until the queue is empty and keep a variable, let’s say idx, for denoting the current index in the Level Order Traversal.
* Pop the Node: If the node is not NULL, then the element at index idx will be the left child, and the element at i+1 will be the right child. Create those children and push the left child and right child of the node, respectively, in the queue.
* After building the tree, return the root node.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public TreeNode solve(ArrayList<Integer> A) {

        TreeNode ans = new TreeNode(A.get(0));

        Queue<TreeNode> q = new LinkedList<TreeNode>();

        q.add(ans);

        int idx = 1;

        while(q.size() > 0) {

            TreeNode x = q.peek();

            q.remove();

            if (x == null) continue;

            int left = A.get(idx);

            int right = A.get(idx+1);

            if (left == -1) x.left = null;

            else x.left = new TreeNode(left);

            if (right == -1) x.right = null;

            else x.right = new TreeNode(right);

            q.add(x.left);

            q.add(x.right);

            idx+=2;

        }

        return ans;

    }

}

**Question 5: Balanced Binary Tree**

Given a root of binary tree **A**, determine if it is height-balanced.

A height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.  
  
**Problem Constraints**

1 <= size of tree <= 100000

Logic – Similar to calculation of height of the tree

* If the difference of height of lst and rst is greater than 1, we will return -1, else we will return the height for the further calculations.
* In the first step we calculate the height of the left and right subtree, if any of the subtree returns -1, it means a subtree in the bottom is not balanced so we return -1.
* If the difference of height of lst and rst is greater than 1 then we will return -1.
* Else we return maximum of height of lst and rst + 1.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    // Post order traversal

    public int helper(TreeNode A) {

        if(A == null) return 0;

        // Calculate the height of the left and right subtree

        int lst = helper(A.left);

        int rst = helper(A.right);

        // If any one of the height return -1 it means the subtree somewhere in the bottom is not balanced

        if (lst == -1 || rst == -1) return -1;

        // If absolute difference is greater than 1 it means the subtree is not balanced

        if (Math.abs(lst-rst) > 1) return -1;

        // If absolute difference is less than or equal to 1 return height

        return Math.max(lst, rst) + 1;

    }

    public int isBalanced(TreeNode A) {

        int temp = helper(A);

        if (temp > 0) return 1;

        return 0;

    }

}

**Question 6: Invert the Binary Tree**

Given a binary tree **A**, invert the binary tree and return it.

Inverting refers to making the left child the right child and vice versa.  
  
**Problem Constraints**

1 <= size of tree <= 100000

* Logic – We do simple pre order traversal and swap the left and right node of the tree while accessing the root.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public void helper(TreeNode A) {

        if (A==null) return;

        TreeNode temp = A.left;

        A.left = A.right;

        A.right = temp;

        helper(A.left);

        helper(A.right);

    }

    public TreeNode invertTree(TreeNode A) {

        helper(A);

        return A;

    }

}

**Question 7: Right View of Binary tree**

Given a binary tree of integers denoted by root **A**. Return an array of integers representing the right view of the Binary tree.

Right view of a Binary Tree is a set of nodes visible when the tree is visited from Right side.  
  
**Problem Constraints**

1 <= Number of nodes in binary tree <= 100000

0 <= node values <= 10^9

Logic - We can start by adding the root node and a null value ( The null value will represent the end of a level. ) in a queue and perform the below steps:

* Get the front value of the queue in a variable (x).
* Check if x is null or not. If it is null then it means one level is over, so we will add a null value to the last of the queue and add the value of the node present after the null node. To access it, first, we remove the null node, and later we peek() the queue. After peeking we continue to step 1.
* If x is not null then we will add the right node first and then the left node of x to the end of the queue. We add the right node first because we want the right view.
* Finally remove the front value of the queue.
* Repeat this process till the length of the queue becomes 1 (At last only a null value will be left in the queue).

The same can be done for a **left view of the binary tree**, the only difference will be in step 3, instead of adding the right node we will add the left node.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public ArrayList<Integer> solve(TreeNode A) {

        ArrayList<Integer> ans = new ArrayList();

        Queue<TreeNode> q = new LinkedList<TreeNode>();

        ans.add(A.val);

        q.add(A);

        q.add(null);

        while(q.size() > 1) {

            TreeNode x = q.peek();

            if (x==null) {

                q.remove();

                TreeNode y = q.peek();

                ans.add(y.val);

                q.add(null);

                continue;

            }

            if (x.right != null) q.add(x.right);

            if (x.left != null) q.add(x.left);

            q.remove();

        }

        return ans;

    }

}

**Question 8: Invert the Binary Tree**

Given a binary tree, return a 2-D array with vertical order traversal of it. Go through the example and image for more details.



**NOTE:** If 2 Tree Nodes shares the same vertical level then the one with lesser depth will come first.  
  
**Problem Constraints**

0 <= number of nodes <= 105

Logic –

We can do level order traversal of the given Binary Tree.  
While traversing the tree, we can maintain Horizontal Distances.  
We initially pass the horizontal distance as 0 for root.  
For the left subtree, we pass the Horizontal Distance as the Horizontal distance of root minus 1.  
For the right subtree, we pass the Horizontal Distance as the Horizontal Distance of root plus 1.

For every horizontal distance value, we maintain a list of nodes in a hash map. Whenever we see a node in traversal, we go to the hash map entry and add the node to the hash map using horizontal distance as a key in a map.

Later on we traverse from min horizontal distance to max horizontal distance and append all the lists in hash map using horizontal distance as a key in a map.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public class myNode {

        int depth;

        TreeNode val;

        myNode(TreeNode x, int y) {

            depth = y;

            val = x;

        }

    }

    public ArrayList<ArrayList<Integer>> verticalOrderTraversal(TreeNode A) {

        int minDepth = Integer.MAX\_VALUE;

        int maxDepth = Integer.MIN\_VALUE;

        Queue<myNode> q = new LinkedList<myNode>();

        HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer, ArrayList<Integer>>();

        q.add(new myNode(A, 0));

        q.add(new myNode(null, -1));

        while(q.size() > 1) {

            myNode x = q.peek();

            if (x.val == null) {

                q.remove();

                q.add(new myNode(null, -1));

                continue;

            }

            if (x.val.left != null) q.add(new myNode(x.val.left, x.depth-1));

            if (x.val.right != null) q.add(new myNode(x.val.right, x.depth+1));

            minDepth = Math.min(minDepth, x.depth);

            maxDepth = Math.max(maxDepth, x.depth);

            if (!map.containsKey(x.depth)) {

                map.put(x.depth, new ArrayList<Integer>());

            }

            ArrayList<Integer> temp = map.get(x.depth);

            temp.add(x.val.val);

            map.put(x.depth, temp);

            q.remove();

        }

        ArrayList<ArrayList<Integer>> ans = new ArrayList<ArrayList<Integer>>();

        for (int i=minDepth;i<=maxDepth;i++) {

            ans.add(map.get(i));

        }

        return ans;

    }

}

For **Top View of Binary tree** we can simply return the first elements of all the lists in the vertical order traversal.

**Question 9: Binary Tree From Inorder And Preorder**

Given preorder and inorder traversal of a tree, construct the binary tree.

**NOTE:** You may assume that duplicates do not exist in the tree.  
  
**Problem Constraints**

1 <= number of nodes <= 105

Logic –  
  
The first element of the preorder array is the root and some of the next elements will be left subtree and some of them will be right subtree.  
  
In the in-order array, if we know the root then the left part of the array will be the left subtree and the right part will be the right subtree.  
  
So, we find the first element of the preorder in the in-order array. Then, create the tree node (let this node be x) with the value first element and find the left subtree and right subtree recursively.  
  
After finding them recursively, we simply connect it to x and return it.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public TreeNode helper(ArrayList<Integer> A, HashMap<Integer, Integer> B, int As, int Ae, int Bs, int Be) {

        if(As > Ae || Bs > Be) return null;

        TreeNode x = new TreeNode(A.get(As));

        int idx = B.get(A.get(As));

        TreeNode lst = helper(A, B, As+1, As+idx-Bs, Bs, idx-1);

        TreeNode rst = helper(A, B, As+idx-Bs+1, Ae, idx+1, Be);

        x.left = lst;

        x.right = rst;

        return x;

    }

    public TreeNode buildTree(ArrayList<Integer> A, ArrayList<Integer> B) {

        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

        for (int i=0; i<B.size();i++) {

            map.put(B.get(i), i);

        }

        return helper(A, map, 0, A.size()-1, 0, B.size()-1);

    }

}

**Question 10: Binary Tree From Inorder And Post order**

Given the inorder and postorder traversal of a tree, construct the binary tree.

**NOTE:** You may assume that duplicates do not exist in the tree.  
  
**Problem Constraints**

1 <= number of nodes <= 105

Logic –

The last element of the post order array is the root and some of the previous elements will be right subtree and some of them will be left subtree.  
  
In the in-order array, if we know the root then the left part of the array will be the left subtree and the right part will be the right subtree.  
  
So, we find the last element of the post order in the in-order array. Then, create the tree node (let this node be x) with the value last element and find the left subtree and right subtree recursively.  
  
After finding them recursively, we simply connect it to x and return it.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public TreeNode helper(ArrayList<Integer> A, HashMap<Integer, Integer> B, int As, int Ae, int Bs, int Be) {

        if (As > Ae || Bs > Be) return null;

        TreeNode x = new TreeNode(A.get(Ae));

        int idx = B.get(A.get(Ae));

        x.left = helper(A, B, As, As+idx-Bs-1, Bs, idx-1);

        x.right = helper(A, B, As+idx-Bs, Ae-1, idx+1, Be);

        return x;

    }

    public TreeNode buildTree(ArrayList<Integer> B, ArrayList<Integer> A) {

        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

        for (int i=0;i<B.size();i++) {

            map.put(B.get(i), i);

        }

        return helper(A, map, 0, A.size()-1, 0, B.size()-1);

    }

}

**Question 11: ZigZag Level Order Traversal BT**

Given a binary tree, return the zigzag level order traversal of its nodes values. (ie, from left to right, then right to left for the next level and alternate between).  
  
**Problem Constraints**

1 <= number of nodes <= 105

Logic – We can use 2 stacks to solve this problem.

* Initially stack s1 will contain only root. We will pop all values from the stack s1 and add the left and right nodes in the stack s2 till the stack s1 becomes empty.
* Here, one level is completed so we will add all the nodes in the ans for this level.
* Now will start popping the stack s2 and insert the right and left nodes in the stack s1 till the stack s2 becomes empty.
* Here again one level is completed so we will add all the nodes in the ans for this level.
* We will repeat this process till both the stacks become empty.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNode A) {

        Stack<TreeNode> s1 = new Stack<TreeNode>();

        Stack<TreeNode> s2 = new Stack<TreeNode>();

        ArrayList<ArrayList<Integer>> ans = new ArrayList<ArrayList<Integer>>();

        ArrayList<Integer> temp = new ArrayList<Integer>();

        s1.push(A);

        while(!(s1.empty() && s2.empty())) {

            while(!s1.empty()) {

                TreeNode x = s1.pop();

                temp.add(x.val);

                if (x.left != null) s2.push(x.left);

                if (x.right != null) s2.push(x.right);

            }

            if(!temp.isEmpty()) {

                ans.add(temp);

                temp = new ArrayList<Integer>();

            }

            while(!s2.empty()) {

                TreeNode x = s2.pop();

                temp.add(x.val);

                if (x.right != null) s1.push(x.right);

                if (x.left != null) s1.push(x.left);

            }

            if(!temp.isEmpty()) {

                ans.add(temp);

                temp = new ArrayList<Integer>();

            }

        }

        return ans;

    }

}

**Question 12: Odd and Even Levels**

Given a binary tree of integers. Find the difference between the sum of nodes at **odd** level and sum of nodes at **even** level.

**NOTE:** Consider the level of root node as 1.  
  
**Problem Constraints**

1 <= Number of nodes in binary tree <= 100000

0 <= node values <= 109

Logic –

Use level Order traversal to calculate the sum of nodes at odd level and even level.

If the level is odd add the node value in the odd variable,  
Else, add it in variable storing the sum of even levels.

After completing the traversal, return odd - even.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int helper(TreeNode A) {

        // flag = 0 means even

        // flag = 1 means odd

        Queue<TreeNode> q = new LinkedList<TreeNode>();

        q.add(A);

        q.add(null);

        int level = 1;

        int ans1 = 0;

        int ans2 = 0;

        int temp = 0;

        while(q.size() > 1) {

            TreeNode x = q.peek();

            if (x==null) {

                q.add(null);

                q.remove();

                if(level % 2 == 1) ans1 += temp;

                else if (level % 2 == 0) ans2 += temp;

                temp = 0;

                level += 1;

                continue;

            }

            temp += x.val;

            if(x.left != null) q.add(x.left);

            if(x.right != null) q.add(x.right);

            q.remove();

        }

        if(level % 2 == 1) ans1 += temp;

        else if (level % 2 == 0) ans2 += temp;

        return ans1-ans2;

    }

    public int solve(TreeNode A) {

        return helper(A); // (helper(A, 1) - helper(A, 0));

    }

}

**Question 13: Valid Binary Search Tree**

You are given a binary tree represented by root **A**.

Assume a BST is defined as follows:

1) The left subtree of a node contains only nodes with keys less than the node's key.

2) The right subtree of a node contains only nodes with keys greater than the node's key.

3) Both the left and right subtrees must also be binary search trees.  
  
**Problem Constraints**

1 <= Number of nodes in binary tree <= 105

0 <= node values <= 232-1

Logic – There are 3 methods to solve this problem.

* Inorder traversal – The inorder traversal has to be a sorted array.
* Preorder traversal – We will check if the node value is between the given range and update the ranges for the left and right nodes.
* Postorder traversal – Get the max and min nodes recursively from lst and rst and compare current node value with max of lst and min value of rst.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public class myNode {

        long min;

        long max;

        boolean isBst;

        public myNode(long x, long y, boolean z) {

            min = x;

            max = y;

            isBst = z;

        }

    }

    public myNode helper(TreeNode A) {

        // Return true and a range in which everything will fit

        if (A==null) return new myNode((long)Integer.MAX\_VALUE+1, (long)Integer.MIN\_VALUE-1, true);

        // Get min and max nodes in left and right range

        myNode lst = helper(A.left);

        myNode rst = helper(A.right);

        // Check if Max from lst < A.val < Min from Rst

        if (lst.isBst && rst.isBst && A.val > lst.max && A.val < rst.min) {

            // If yes return new minimum and maximum and true

            return new myNode(Math.min(A.val, lst.min), Math.max(A.val, rst.max), true);

        }

        // Return a range in which nothing will fit

        return new myNode((long)Integer.MIN\_VALUE-1, (long)Integer.MAX\_VALUE+1, false);

    }

    public int isValidBST(TreeNode A) {

        myNode x = helper(A);

        if(x.isBst) return 1;

        return 0;

    }

}

**Question 14: Sorted Array To Balanced BST**

Given an array where elements are sorted in ascending order, convert it to a height Balanced Binary Search Tree (BBST).

**Balanced tree :** a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.  
  
**Problem Constraints**

1 <= length of array <= 100000

Logic –

* For a BST, all values lower than the root go in the left part of the root, and all values higher go in the right part of the root.
* To balance the tree, we will need to make sure we distribute the elements almost equally in the left and right parts.
* So we choose the mid part of the array as the root and divide the elements around it.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    // DO NOT MODIFY THE ARGUMENTS WITH "final" PREFIX. IT IS READ ONLY

    public TreeNode helper(final int[] A, int l, int r){

        if (l>r) return null;

        int mid = (l+r)/2;

        TreeNode root = new TreeNode(A[mid]);

        root.left = helper(A, l, mid-1);

        root.right = helper(A, mid+1, r);

        return root;

    }

    public TreeNode sortedArrayToBST(final int[] A) {

        return helper(A, 0, A.length-1);

    }

}

**Question 15: BST nodes in a range**

Given a binary search tree of integers. You are given a range **B and C**.

Return the count of the number of nodes that lie in the given range.  
  
**Problem Constraints**

1 <= Number of nodes in binary tree <= 100000

0 <= B < = C <= 109

Logic –

We do the post order in this question. We get the number of nodes from lst (x) and number of nodes of rst(y) between the given range and if the current node is between the range then we return x+y+1 else return x+y.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int helper(TreeNode A, int B, int C) {

        if (A==null) return 0;

        int lst = helper(A.left, B, C);

        int rst = helper(A.right, B, C);

        int count = 0;

        if (A.val >= B && A.val <= C) count = 1;

        return lst+rst+count;

    }

    public int solve(TreeNode A, int B, int C) {

        return helper(A, B, C);

    }

}

**Question 16: Check for BST with One Child**

Given **preorder** traversal of a binary tree, check if it is possible that it is also a preorder traversal of a Binary Search Tree (BST), where each internal node (non-leaf nodes) have **exactly one** child.

**Problem Constraints**

1 <= number of nodes <= 100000

Logic –

In Preorder traversal, descendants (or Preorder successors) of every node appear after the node.

We can say, if all internal nodes have only one child in a BST, then all the descendants of every node are either smaller or larger than the node.

To check the above condition:

* Scan the last two nodes of preorder & mark them as min & max.
* Scan every node down the preorder array. Each node must be either smaller than the min node or larger than the max node. Update min & max accordingly.
* If the current value is greater than previous value then it means we move right, so we update min, else we update the max.

Code –

public class Solution {

    public String solve(ArrayList<Integer> A) {

        int mn = Integer.MIN\_VALUE;

        int mx = Integer.MAX\_VALUE;

        for(int i=1;i<A.size();i++) {

            // Check if the current value lies between min and max

            if (A.get(i) >= mn && mx >= A.get(i)) {

                // If the current value is greater than previous value it means we move to right

                // So we change the min

                if(A.get(i) > A.get(i-1)) {

                    mn = A.get(i-1) + 1;

                // Else we move to left

                // So we change the max

                } else {

                    mx = A.get(i-1) -  1;

                }

                continue;

            }

            return "NO";

        }

        return "YES";

    }

}

**Question 17: Two Sum BST**

Given a binary search tree **A**, where each node contains a positive integer, and an integer **B**, you have to find whether or not there exist two different nodes X and Y such that X.value + Y.value = B.

Return 1 to denote that two such nodes exist. Return 0, otherwise.  
  
**Problem Constraints**

1 <= size of tree <= 100000

1 <= B <= 109

Logic-

Take the inorder traversal and apply 2 pointer method to find the 2 nodes with the given sum.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public void inorder(TreeNode A, int B, ArrayList<Integer> ans) {

        if (A==null) return;

        inorder(A.left, B, ans);

        ans.add(A.val);

        inorder(A.right, B, ans);

    }

    public int t2Sum(TreeNode A, int B) {

        ArrayList<Integer> ans = new ArrayList();

        inorder(A, B, ans);

        int i = 0;

        int j = ans.size()-1;

        while(j>i) {

            if(ans.get(i) + ans.get(j) == B) return 1;

            else if (ans.get(i) + ans.get(j) < B) i+=1;

            else j-=1;

        }

        return 0;

    }

}

**Question 14: Largest BST Subtree**

You are given a Binary Tree **A** with **N** nodes.

Write a function that returns the size of the largest subtree, which is also a Binary Search Tree (BST).

If the complete Binary Tree is BST, then return the size of the whole tree.

**NOTE:**

* The largest subtree is the subtree with the most number of nodes.

**Problem Constraints**

1 <= N <= 105

Logic –

* Get the height of largest left BST and largest BST right BST and max and min nodes recursively from lst and rst.
* Compare current node value with max of lst and min value of rst. If it is in the range return the new max and min nodes and the new height will be height of largest BST on left + height of largest BST on right +1.
* Else return max(height of largest BST on left, height of largest BST) as height and pass max and min in such a way that nothing will satisfy the above condition

(max of lst < A.val < min of rst)

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public class myNode{

        int min;

        int max;

        int size;

        public myNode(int x, int y, int z) {

            min = x;

            max = y;

            size = z;

        }

    }

    public myNode helper(TreeNode A) {

        // Return the height of the tree = 0 and a range in which all numbers from -infinity to infinity will fit

        if(A==null) return new myNode(Integer.MAX\_VALUE, Integer.MIN\_VALUE, 0);

        // Get height of the largest subtree from left and right

        myNode lst = helper(A.left);

        myNode rst = helper(A.right);

        // Check if the current value is between max of lst and min of rst

        if(A.val > lst.max && A.val < rst.min) {

            // If yes add 1 to the height of lst and rst

            return new myNode(Math.min(A.val, lst.min), Math.max(A.val, rst.max), lst.size + rst.size + 1);

        }

        // Else return the max of lst and rst

        return new myNode(Integer.MIN\_VALUE, Integer.MAX\_VALUE, Math.max(lst.size, rst.size));

    }

    public int solve(TreeNode A) {

        return helper(A).size;

    }

}

**Question 15: Recover Binary Search Tree**

Two elements of a binary search tree (BST), represented by root **A** are swapped by mistake. Tell us the 2 values swapping which the tree will be restored.

A solution using O(n) space is pretty straightforward. Could you devise a constant space solution?  
  
**Problem Constraints**

1 <= size of tree <= 100000

**Logic –** There are 3 ways to solve this problem.

* Perform the inorder traversal and find the elements that are not sorted.
* Maintain 2 variables first and second and update them while performing the inorder traversal. This will save some space.
* Use Morris Traversal.

**Code –**

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int first = -1;

    public int second = -1;

    public TreeNode prev = null;

    public void helper(TreeNode A) {

        if (A==null) return;

        helper(A.left);

        if(prev != null && prev.val > A.val) {

            if (first == -1) first = prev.val;

            second = A.val;

        }

        prev = A;

        helper(A.right);

    }

    public ArrayList<Integer> recoverTree(TreeNode A) {

        helper(A);

        ArrayList<Integer> temp = new ArrayList<Integer>();

        temp.add(second);

        temp.add(first);

        return temp;

    }

}

**Question 16: Least Common Ancestor**

Find the lowest common ancestor in an unordered binary tree **A,** given two values, **B** and **C,** in the tree.

Lowest common ancestor: the lowest common ancestor (LCA) of two nodes and w in a tree or directed acyclic graph (DAG) is the lowest (i.e., deepest) node that has both v and w as descendants.

**Problem Constraints**

1 <= size of tree <= 100000

1 <= B, C <= 109

Logic –

We can find the LCA in 2 ways:  
  
Linear solution using path calculation :  
1) Find a path from the root to n1 and store it in a vector or array.  
2) Find a path from the root to n2 and store it in another vector or array.  
3) Traverse both paths till the values in arrays are the same. Return the common element just before the mismatch  
  
Linear solution using recursion :  
We traverse from the bottom (post-order traversal), and once we reach a node that matches one of the two nodes, we return that node.  
  
( if (A.val == B || A.val == C) return A; )  
  
If the node does not match the given nodes, we recursively call the function for the left subtree and right subtree.  
  
TreeNode lst = findLCA(A.left, B, C);  
TreeNode rst = findLCA(A.right, B, C);  
  
If we get a value not equal to null from left and right, then the current node is the LCA, so we return it.  
  
if(lst != null && rst != null) return A;  
  
If one of the values is null then we simply return the value which is not null.  
  
if(lst == null && rst != null) return rst;  
if(lst != null && rst == null) return lst;  
  
Otherwise, if both are null we return null.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public TreeNode helper(TreeNode A, int B, int C) {

        if(A==null) return null;

        TreeNode lst = helper(A.left, B, C);

        TreeNode rst = helper(A.right, B, C);

        if(A.val == B || A.val == C) return A;

        if(lst != null && rst != null) return A;

        if(lst == null && rst != null) return rst;

        if(lst != null && rst == null) return lst;

        return null;

    }

    public boolean isPresent(TreeNode A, int B) {

        if (A==null) return false;

        if(A.val == B) return true;

        return isPresent(A.left, B) || isPresent(A.right, B);

    }

    public int lca(TreeNode A, int B, int C) {

        if(!isPresent(A, B) || !isPresent(A, C)) return -1;

        return helper(A, B, C).val;

    }

}

**Question 17: Diameter of binary tree**

Given a Binary Tree **A** consisting of **N** integer nodes, you need to find the **diameter** of the tree.

The **diameter** of a tree is the number of edges on the longest path between two nodes in the tree.  
  
**Problem Constraints**

0 <= N <= 105

Logic – We use the post order traversal here.

* We calculate the values for left subtree and right subtree and maintain a global max value of lst + rst.
* Finally, return maximum value of lst and rst + 1.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int ans = 0;

    public int helper(TreeNode A) {

        if (A==null) return 0;

        int lst = helper(A.left);

        int rst = helper(A.right);

        ans = Math.max(lst+rst, ans);

        return Math.max(lst, rst) + 1;

    }

    public int solve(TreeNode A) {

        helper(A);

        return ans;

    }

}

**Question 18: Common Nodes in Two BST**

Given two BST's **A** and **B**, return the **(sum of all common nodes in both A and B) % (109 +7)**.

In case there is no common node, return 0.

**NOTE:**

Try to do it one pass through the trees.  
  
**Problem Constraints**

1 <= Number of nodes in the tree A and B <= 105

1 <= Node values <= 106

Logic – There are 3 ways to do this.

* Traverse Tree A and store each value in a Hash map, and later on traverse Tree B and check if the element is present in the Hash map. If present in the hash map, add to the answer variable.
* Create in order arrays and find the common elements in the 2 array.
* We traverse both the trees at a time by using stacks which will save some space.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public long helper(TreeNode A, HashMap<Integer, Integer> map) {

        if(A==null) return 0;

        long ans = 0;

        if(map.containsKey(A.val)) {

            int x = map.get(A.val);

            ans += A.val;

            // Remove the element from map

            if (x==1) map.remove(A.val);

            else map.put(A.val, x-1);

        }

        ans += helper(A.left, map);

        ans += helper(A.right, map);

        return ans;

    }

    public void preorder(TreeNode A, HashMap<Integer, Integer> map) {

        if(A==null) return;

        if (!map.containsKey(A.val)) map.put(A.val, 0);

        map.put(A.val, map.get(A.val)+1);

        preorder(A.left, map);

        preorder(A.right, map);

    }

    public int solve(TreeNode A, TreeNode B) {

        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

        preorder(A, map);

        return (int)(helper(B, map) % (1000\*1000\*1000+7));

    }

}

**Question 19: Distance between Nodes of BST**

Given a binary search tree.  
Return the distance between two nodes with given two keys **B** and **C**. It may be assumed that both keys exist in BST.

**NOTE**: Distance between two nodes is number of edges between them.  
  
**Problem Constraints**

1 <= Number of nodes in binary tree <= 1000000

0 <= node values <= 109

Logic – We start from the root and for every node, we do following.

* If both keys are greater than the current node, we move to the right child of the current node.
* If both keys are smaller than current node, we move to left child of current node.
* If one keys is smaller and other key is greater, current node is Lowest Common Ancestor (LCA) of two nodes.
* We find distances of current node from two keys and return sum of the distances

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int findDis(TreeNode A, int B) {

        if(A==null) return 0;

        if(A.val == B) return 1;

        int lst = findDis(A.left, B);

        int rst = findDis(A.right, B);

        if(lst==0 && rst==0) return 0;

        return Math.max(lst, rst) + 1;

    }

    public int solve(TreeNode A, int B, int C) {

        TreeNode temp = A;

        while(temp != null) {

            if(temp.val < B && temp.val < C) {

                temp = temp.right;

            } else if(temp.val > B && temp.val > C) {

                temp = temp.left;

            } else {

                return findDis(temp, B) + findDis(temp, C) - 2;

            }

        }

        return -1;

    }

}

**Question 20: Identical Binary Trees**

Given two binary trees, check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.  
  
**Problem Constraints**

1 <= number of nodes <= 105

Logic –

* Check if the number of branches of the trees are same or not. We can do that recursively and if we find one of the 2 tree nodes to be null then the trees are not identical.
* If the branches are identical we check for the values if the values are same then we traverse left and right subtrees recursively.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int isSameTree(TreeNode A, TreeNode B) {

        // Check if branches of tree are not identical

        if ((A==null && B!=null) || (A!=null && B==null)) return 0;

        // If branches of the trees are identical

        if (A==null && B==null) return 1;

        // If branches are identical check for their values

        if(A.val == B.val) {

            return isSameTree(A.left, B.left) & isSameTree(A.right, B.right);

        }

        return 0;

    }

}

**Question 21: Equal Tree Partition**

Given a binary tree **A**. Check whether it is possible to partition the tree to two trees which have equal sum of values after removing exactly one edge on the original tree.

**Problem Constraints**

1 <= size of tree <= 100000

0 <= value of node <= 109

Logic –

After removing some edge from parent to child,  
(where the child cannot be the original root)  
the subtree rooted at child must be half the sum of the entire tree.

Let’s record the sum of every subtree. We can do this recursively using depth-first search.  
After, we should check that half the sum of the entire tree occurs somewhere in our recording  
(and not from the total of the entire tree.)

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int ans = 0;

    public long findSum(TreeNode A) {

        if(A==null) return 0;

        return findSum(A.left) + findSum(A.right) + A.val;

    }

    public long helper(TreeNode A, long totSum) {

        if(A==null) return 0;

        long temp = helper(A.left, totSum) + helper(A.right, totSum) + A.val;

        if(temp == totSum/2) {

            ans = 1;

        }

        return temp;

    }

    public int solve(TreeNode A) {

        helper(A, findSum(A));

        return ans;

    }

}

**Question 22: Next Pointer Binary Tree**

Given a binary tree,

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Assume perfect binary tree and try to solve this in constant extra space.  
  
**Problem Constraints**

1 <= Number of nodes in binary tree <= 100000

0 <= node values <= 10^9

Logic – There are 2 methods to solve this

* We can solve this by using the level order traversal and while traversing each level we can connect the next of the current node to the node present at the right of the current node in the queue.
* Iterative way (SC: O(1))
* Maintain a variable equal curr equal to root.
* Traverse the curr = curr.left.
* While traversing make the connections for the lower levels, using current level.
* Repeat this till curr.left becomes null.

Code –

/\*\*

 \* Definition for binary tree with next pointer.

 \* public class TreeLinkNode {

 \*     int val;

 \*     TreeLinkNode left, right, next;

 \*     TreeLinkNode(int x) { val = x; }

 \* }

 \*/

public class Solution {

    public void connect(TreeLinkNode root) {

        TreeLinkNode curr = root;

        while(curr != null && curr.left != null) {

            TreeLinkNode temp = curr;

            while(temp != null) {

                temp.left.next = temp.right;

                if(temp.next != null) temp.right.next = temp.next.left;

                temp = temp.next;

            }

            curr = curr.left;

        }

    }

}

**Question 23: Kth Smallest Element In BST**

Given a binary search tree represented by root **A**, write a function to find the **Bth** smallest element in the tree.

**Problem Constraints**

1 <= Number of nodes in binary tree <= 100000

0 <= node values <= 10^9

Logic –

We do the inorder traversal as it is gives the sorted array for the BST.

While doing the traversal we maintain some global variables k = 1and ans = -1, and check if the value of k is equal to B or not, if yes then we update the value of ans to value of current node.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int k = 1;

    public int ans = -1;

    public void inorder(TreeNode A, int B) {

        if(A==null) return;

        inorder(A.left, B);

        if(k == B) {

            ans = A.val;

        }

        k+=1;

        inorder(A.right,B);

    }

    public int kthsmallest(TreeNode A, int B) {

        inorder(A, B);

        return ans;

    }

}

**Question 24: Symmetric Binary Tree**

Given a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

**Problem Constraints**

1 <= number of nodes <= 105

Logic –

We use a helper function for this problem, which takes left node (A) and right node (B) of root as parameters.

We compare the left value of the node A with right value of node B and the right value of node A with left value of node B recursively.

While recursion if one of the parameters becomes null we return false, as it will not form a symmetric binary tree.

If both become null then we return true.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int helper(TreeNode A, TreeNode B) {

        if (A==null && B==null) return 1;

        if (A==null || B==null) return 0;

        return helper(A.left, B.right) & helper(B.left, A.right);

    }

    public int isSymmetric(TreeNode A) {

        return helper(A.left, A.right);

    }

}

**Question 25: Sum binary tree or not**

Given a binary tree. Check whether the given tree is a **Sum-binary Tree** or not.

**Sum-binary Tree** is a Binary Tree where the value of a every node is equal to sum of the nodes present in its left subtree and right subtree.

An empty tree is Sum-binary Tree and sum of an empty tree can be considered as 0. A leaf node is also considered as SumTree.

Return 1 if it sum-binary tree else return 0.

**Problem Constraints**

1 <= length of the array <= 100000

0 <= node values <= 50

Logic –

* Return 0 if node is null.
* Return node value if node is root.
* Recursively call the function for left and right sub trees.
* If Sum of lst + Sum of rst != A.val then make the global variable ans = 0 (false).
* Finally, return the sum of lst + sum of rst + A.val.

Code –

/\*\*

 \* Definition for binary tree

 \* class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) {

 \*      val = x;

 \*      left=null;

 \*      right=null;

 \*     }

 \* }

 \*/

public class Solution {

    public int ans = 1;

    public int helper(TreeNode A) {

        if(A==null) return 0;

        if(A.left == null && A.right == null) return A.val;

        int lst = helper(A.left);

        int rst = helper(A.right);

        if(lst + rst != A.val) {

            ans = 0;

        }

        return lst + rst + A.val;

    }

    public int solve(TreeNode A) {

        helper(A);

        return ans;

    }

}

**Question 26: Path Sum**

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.  
  
**Problem Constraints**

1 <= number of nodes <= 105

-100000 <= B, value of nodes <= 100000

Logic –

Code –

Day 68 completed