**LEETCODE ARCHIVE TREES**

**Q94. Binary Tree Inorder Traversal**

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

**Example 1:**



**Input:** root = [1,null,2,3]

**Output:** [1,3,2]

**Example 2:**

**Input:** root = []

**Output:** []

**Example 3:**

**Input:** root = [1]

**Output:** [1]

**Constraints:**

* The number of nodes in the tree is in the range [0, 100].
* -100 <= Node.val <= 100

**Follow up:** Recursive solution is trivial, could you do it iteratively?

Solution:

public class TreeNode {  
 public int val;  
 public TreeNode left;  
 public TreeNode right;  
  
 public TreeNode() {  
 }  
  
 public TreeNode(int val) {  
 this.val = val;  
 }  
  
 public TreeNode(int val, TreeNode left, TreeNode right) {  
 this.val = val;  
 this.left = left;  
 this.right = right;  
 }  
}

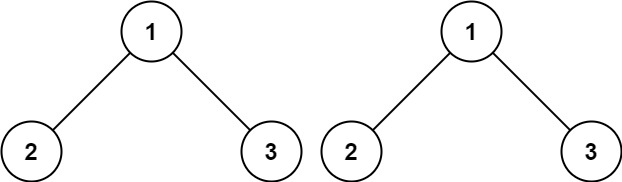
class Solution {  
 public List<Integer> inorderTraversal(TreeNode root) {  
  
 List<Integer> list = new ArrayList<>();  
  
 inorder(root, list);  
  
 return list;  
  
 }  
  
 private void inorder(TreeNode root, List<Integer> list) {  
  
 if (root != null) {  
 if (root.left != null) {  
 inorder(root.left, list);  
 }  
  
 list.add(root.val);  
  
 if (root.right != null) {  
 inorder(root.right, list);  
 }  
 }  
 }  
}

**Q100. Same Tree**

Given the roots of two binary trees p and q, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

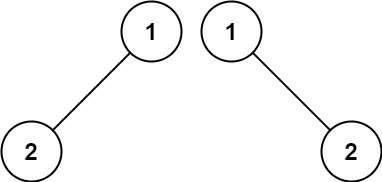
**Example 1:**



**Input:** p = [1,2,3], q = [1,2,3]

**Output:** true

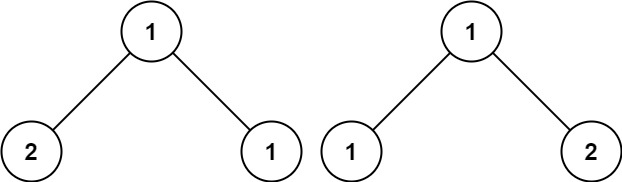
**Example 2:**



**Input:** p = [1,2], q = [1,null,2]

**Output:** false

**Example 3:**



**Input:** p = [1,2,1], q = [1,1,2]

**Output:** false

**Constraints:**

* The number of nodes in both trees is in the range [0, 100].
* -104 <= Node.val <= 104

Solution:

class Solution {  
 public boolean isSameTree(TreeNode p, TreeNode q) {  
 boolean flag = false;  
  
 if(p == null && q == null) return true;  
 if(p == null && q != null) return false;  
 if(p != null && q == null) return false;  
  
 if(p.val != q.val) return false;  
  
 return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);  
 }  
}

**Q101. Symmetric Tree**

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

**Example 1:**



**Input:** root = [1,2,2,3,4,4,3]

**Output:** true

**Example 2:**



**Input:** root = [1,2,2,null,3,null,3]

**Output:** false

**Constraints:**

* The number of nodes in the tree is in the range [1, 1000].
* -100 <= Node.val <= 100

**Follow up:** Could you solve it both recursively and iteratively?

Solution:

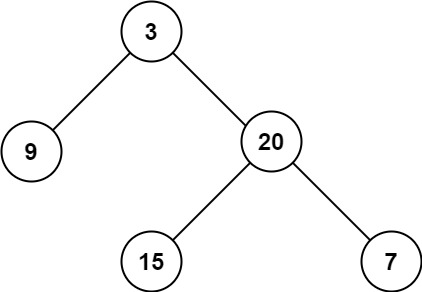
class Solution {  
 public boolean isSymmetric(TreeNode root) {  
 return isMirrorTree(root.left, root.right);  
 }  
  
 public boolean isMirrorTree(TreeNode p, TreeNode q) {  
 boolean flag = false;  
  
 if (p == null && q == null) return true;  
 if (p == null && q != null) return false;  
 if (p != null && q == null) return false;  
  
 return p.val == q.val && isMirrorTree(p.left, q.right) && isMirrorTree(p.right, q.left);  
 }  
}

**Q104. Maximum Depth of Binary Tree**

Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** 3

**Example 2:**

**Input:** root = [1,null,2]

**Output:** 2

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -100 <= Node.val <= 100

Solution:

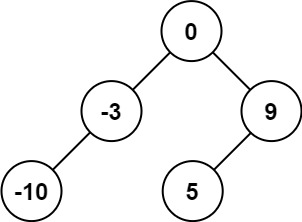
class Solution {  
 public int maxDepth(TreeNode root) {  
  
 if (root == null) return 0;  
 int leftDepth = 0;  
 int rightDepth = 0;  
 if (root.left != null) {  
 leftDepth = maxDepth(root.left);  
 }  
  
 if (root.right != null) {  
 rightDepth = maxDepth(root.right);  
 }  
  
 return Math.*max*(leftDepth, rightDepth) + 1;  
 }  
}

**Q108. Convert Sorted Array to Binary Search Tree**

Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

A height-balanced binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

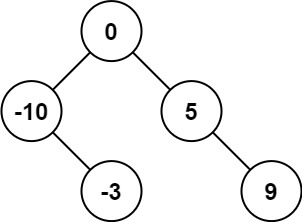
**Example 1:**



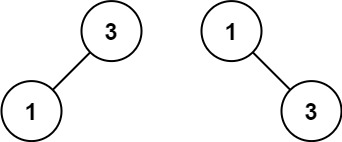
**Input:** nums = [-10,-3,0,5,9]

**Output:** [0,-3,9,-10,null,5]

**Explanation:** [0,-10,5,null,-3,null,9] is also accepted:



**Example 2:**



**Input:** nums = [1,3]

**Output:** [3,1]

**Explanation:** [1,null,3] and [3,1] are both height-balanced BSTs.

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums is sorted in a **strictly increasing** order.

Solution:

class Solution {  
 public TreeNode sortedArrayToBST(int[] nums) {  
 if (nums.length == 1) return new TreeNode(nums[0]);  
 return TreeBuilder(nums, 0, nums.length - 1);  
 }  
  
 private TreeNode TreeBuilder(int[] nums, int start, int end) {  
  
 if (start <= end) {  
 int mid = (start + end) / 2;  
 TreeNode root = new TreeNode(nums[mid]);  
 root.left = TreeBuilder(nums, start, mid - 1);  
 root.right = TreeBuilder(nums, mid + 1, end);  
 return root;  
 } else return null;  
 }  
  
}

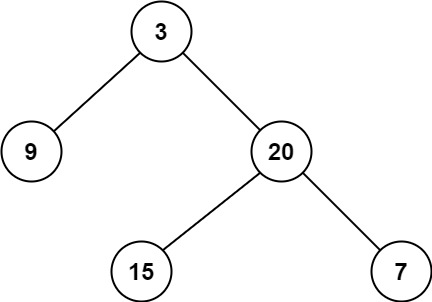
**Q111. Minimum Depth of Binary Tree**

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Note: A leaf is a node with no children.

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** 2

**Example 2:**

**Input:** root = [2,null,3,null,4,null,5,null,6]

**Output:** 5

**Constraints:**

* The number of nodes in the tree is in the range [0, 105].
* -1000 <= Node.val <= 1000

Solution:

class Solution {  
 public int minDepth(TreeNode root) {  
  
 if (root == null) return 0;  
  
 int minLeft = minDepth(root.left);  
 int minRight = minDepth(root.right);  
  
 if (minLeft == 0 || minRight == 0) return 1 + Math.*max*(minLeft, minRight);  
  
 return 1 + Math.*min*(minLeft, minRight);  
  
 }  
}

**Q112. Path Sum**

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

**Example 1:**



**Input:** root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

**Output:** true

**Explanation:** The root-to-leaf path with the target sum is shown.

**Example 2:**



**Input:** root = [1,2,3], targetSum = 5

**Output:** false

**Explanation:** There two root-to-leaf paths in the tree:

(1 --> 2): The sum is 3.

(1 --> 3): The sum is 4.

There is no root-to-leaf path with sum = 5.

**Example 3:**

**Input:** root = [], targetSum = 0

**Output:** false

**Explanation:** Since the tree is empty, there are no root-to-leaf paths.

**Constraints:**

* The number of nodes in the tree is in the range [0, 5000].
* -1000 <= Node.val <= 1000
* -1000 <= targetSum <= 1000

Solution:

class Solution {  
  
 public boolean hasPathSum(TreeNode root, int targetSum) {  
 if (root == null) return false;  
  
 targetSum = targetSum - root.val;  
  
 if (root.left == null && root.right == null) {  
 return targetSum == 0;  
 }  
  
 return hasPathSum(root.left, targetSum) || hasPathSum(root.right, targetSum);  
 }  
}

**Q144. Binary Tree Preorder Traversal**

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

**Example 1:**



**Input:** root = [1,null,2,3]

**Output:** [1,2,3]

**Example 2:**

**Input:** root = []

**Output:** []

**Example 3:**

**Input:** root = [1]

**Output:** [1]

**Constraints:**

* The number of nodes in the tree is in the range [0, 100].
* -100 <= Node.val <= 100

**Follow up:** Recursive solution is trivial, could you do it iteratively?

class Solution {  
 public List<Integer> preorderTraversal(TreeNode root) {  
 List<Integer> list = new ArrayList<>();  
 preOrder(root, list);  
 return list;  
 }  
  
 private void preOrder(TreeNode node, List<Integer> list) {  
 if (node == null) return;  
  
 list.add(node.val);  
 if (node.left != null) {  
 preOrder(node.left, list);  
 }  
  
 if (node.right != null) {  
 preOrder(node.right, list);  
 }  
  
 }  
}

**Q145. Binary Tree Postorder Traversal**

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

**Example 1:**



**Input:** root = [1,null,2,3]

**Output:** [3,2,1]

**Example 2:**

**Input:** root = []

**Output:** []

**Example 3:**

**Input:** root = [1]

**Output:** [1]

**Constraints:**

* The number of the nodes in the tree is in the range [0, 100].
* -100 <= Node.val <= 100

**Follow up:** Recursive solution is trivial, could you do it iteratively?

Solution:

class Solution {  
 public List<Integer> postorderTraversal(TreeNode root) {  
 List<Integer> list = new ArrayList<>();  
 postOrder(root, list);  
 return list;  
 }  
  
 private void postOrder(TreeNode node, List<Integer> list) {  
 if (node == null) return;  
  
 if (node.left != null) {  
 postOrder(node.left, list);  
 }  
 if (node.right != null) {  
 postOrder(node.right, list);  
 }  
  
 list.add(node.val);  
 }  
}

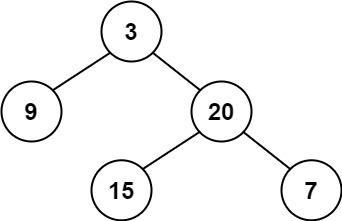
**Q110. Balanced Binary Tree**

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as:

a binary tree in which the left and right subtrees of every node differ in height by no more than 1.

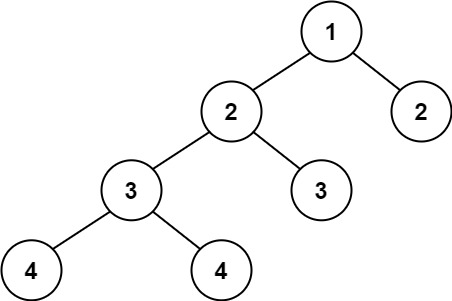
**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** true

**Example 2:**



**Input:** root = [1,2,2,3,3,null,null,4,4]

**Output:** false

**Example 3:**

**Input:** root = []

**Output:** true

**Constraints:**

* The number of nodes in the tree is in the range [0, 5000].
* -104 <= Node.val <= 104

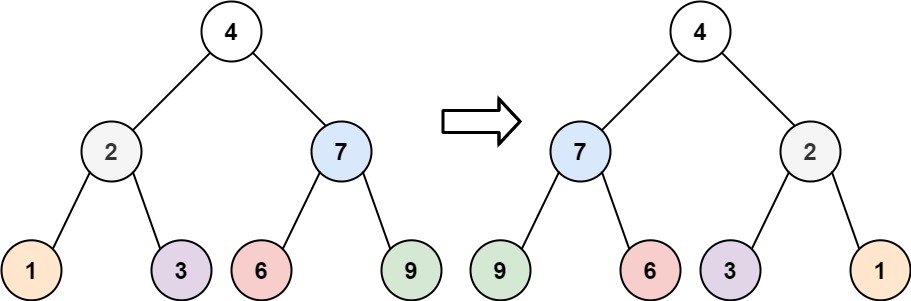
Solution:

class Solution {  
  
 boolean isBal = true;  
  
 public boolean isBalanced(TreeNode root) {  
 height(root);  
 return isBal;  
 }  
  
 private int height(TreeNode node) {  
  
 if (node == null) return 0;  
  
 int leftHeight = height(node.left);  
 int rightHeight = height(node.right);  
  
 if (Math.*abs*(leftHeight - rightHeight) > 1) {  
 isBal = false;  
 }  
 return 1 + Math.*max*(leftHeight, rightHeight);  
 }  
}

**Q226. Invert Binary Tree(\*\*)**

Given the root of a binary tree, invert the tree, and return its root.

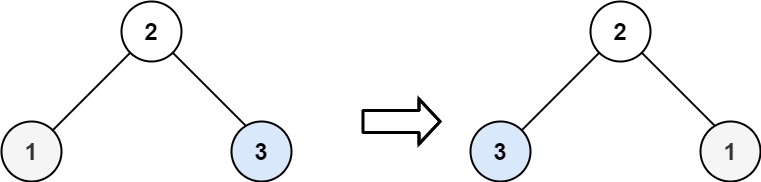
**Example 1:**



**Input:** root = [4,2,7,1,3,6,9]

**Output:** [4,7,2,9,6,3,1]

**Example 2:**



**Input:** root = [2,1,3]

**Output:** [2,3,1]

**Example 3:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 100].
* -100 <= Node.val <= 100
* Solution:

class Solution {  
 public TreeNode invertTree(TreeNode root) {  
 if (root == null) return null;  
  
 TreeNode left = invertTree(root.left);  
 TreeNode right = invertTree(root.right);  
  
 root.right = left;  
 root.left = right;  
 return root;  
 }  
}

**Q235. Lowest Common Ancestor of a Binary Search Tree**

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

**Example 1:**



**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

**Output:** 6

**Explanation:** The LCA of nodes 2 and 8 is 6.

**Example 2:**



**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

**Output:** 2

**Explanation:** The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

**Example 3:**

**Input:** root = [2,1], p = 2, q = 1

**Output:** 2

**Constraints:**

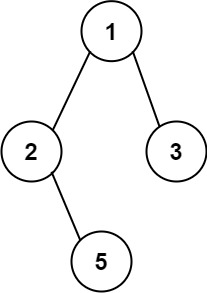
* The number of nodes in the tree is in the range [2, 105].
* -109 <= Node.val <= 109
* All Node.val are **unique**.
* p != q
* p and q will exist in the BST.

Solution:

public class Solution {  
 public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
  
 if (root == null) return null;  
  
 if (root.val == p.val || root.val == q.val) {  
 return root;  
 }  
  
 TreeNode left = lowestCommonAncestor(root.left, p, q);  
 TreeNode right = lowestCommonAncestor(root.right, p, q);  
 if (left == null) return right;  
 if (right == null) return left;  
  
 return root;  
  
 }  
}

**Q257. Binary Tree Paths**

Given the root of a binary tree, return *all root-to-leaf paths in****any order***. A **leaf** is a node with no children. **Example 1:**



**Input:** root = [1,2,3,null,5]

**Output:** ["1->2->5","1->3"]

**Example 2:**

**Input:** root = [1]

**Output:** ["1"]

**Constraints:**

* The number of nodes in the tree is in the range [1, 100].
* -100 <= Node.val <= 100

Solution:

class Solution {  
 public List<String> binaryTreePaths(TreeNode root) {  
 List<String> list = new ArrayList<>();  
 dfs(root, "", list);  
 return list;  
 }  
  
 private void dfs(TreeNode node, String path, List<String> list) {  
  
 if (node == null) return;  
  
 path += String.*valueOf*(node.val);  
  
 if (node.left == null && node.right == null) {  
 list.add(path);  
 return;  
 }  
  
 if (node.left != null) {  
 dfs(node.left, path + "->", list);  
 }  
  
 if (node.right != null) {  
 dfs(node.right, path + "->", list);  
 }  
 }  
}

**404. Sum of Left Leaves**

Given the root of a binary tree, return *the sum of all left leaves.* A **leaf** is a node with no children. A **left leaf** is a leaf that is the left child of another node. **Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** 24

**Explanation:** There are two left leaves in the binary tree, with values 9 and 15 respectively.

**Example 2:**

**Input:** root = [1]

**Output:** 0

**Constraints:**

* The number of nodes in the tree is in the range [1, 1000].
* -1000 <= Node.val <= 1000 **Solution**:

class Solution {  
 public int sumOfLeftLeaves(TreeNode root) {  
 int sum = 0;  
 if (root == null) return 0;  
 if (isLeaf(root.left)) {  
 sum += root.left.val;  
 } else {  
 sum += sumOfLeftLeaves(root.left);  
 }  
  
 sum += sumOfLeftLeaves(root.right);  
 return sum;  
 }  
  
 private boolean isLeaf(TreeNode node) {  
  
 if (node == null) return false;  
 if (node.right == null && node.left == null) return true;  
 return false;  
 }  
}

**Q501. Find Mode in Binary Search Tree**

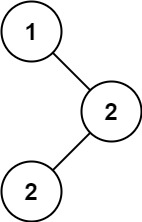
Given the root of a binary search tree (BST) with duplicates, return *all the*[*mode(s)*](https://en.wikipedia.org/wiki/Mode_(statistics))*(i.e., the most frequently occurred element) in it*.

If the tree has more than one mode, return them in **any order**.

Assume a BST is defined as follows:

* The left subtree of a node contains only nodes with keys **less than or equal to** the node's key.
* The right subtree of a node contains only nodes with keys **greater than or equal to** the node's key.
* Both the left and right subtrees must also be binary search trees.

**Example 1:**



**Input:** root = [1,null,2,2]

**Output:** [2]

**Example 2:**

**Input:** root = [0]

**Output:** [0]

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -105 <= Node.val <= 105

**Solution:**

class Solution {

public int[] findMode(TreeNode root) {  
 List<Integer> list = new ArrayList<>();  
 list = dfs(root, list);  
 System.*out*.println(list);  
 return getMode(list);  
}  
  
private List<Integer> dfs(TreeNode node, List<Integer> list) {  
 if (node == null) return List.*of*();  
 list.add(node.val);  
 if (node.left != null) {  
 list = dfs(node.left, list);  
 }  
 if (node.right != null) {  
 list = dfs(node.right, list);  
 }  
  
 return list;  
}  
  
private int[] getMode(List<Integer> list) {  
 Map<Integer, Integer> map = new HashMap<>();  
 for (Integer n : list) {  
 if (map.containsKey(n)) {  
 map.put(n, map.get(n) + 1);  
 } else {  
 map.put(n, 1);  
 }  
 }  
  
 Integer max = Collections.*max*(map.values());  
  
 int frequency = Collections.*frequency*(map.values(), max);  
  
 System.*out*.println("Max is : " + max + "Frequency : " + frequency);  
 List<Integer> res = new ArrayList<>();  
 for (Map.Entry<Integer, Integer> entry : map.entrySet()) {  
 if (entry.getValue().equals(max)) {  
 res.add(entry.getKey());  
 }  
 }  
  
 System.*out*.println(res);  
 int i = 0;  
 int[] arr = new int[res.size()];  
 for (Integer num : res) {  
 arr[i++] = num;  
 }  
  
 System.*out*.println(Arrays.*toString*(arr));  
 return arr;  
}

}

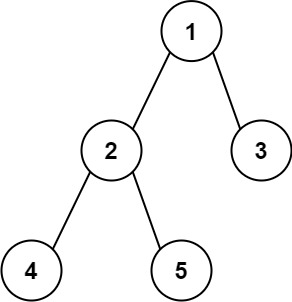
**Q543. Diameter of Binary Tree**

Given the root of a binary tree, return *the length of the****diameter****of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.

**Example 1:**



**Input:** root = [1,2,3,4,5]

**Output:** 3

**Explanation:** 3 is the length of the path [4,2,1,3] or [5,2,1,3].

**Example 2:**

**Input:** root = [1,2]

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -100 <= Node.val <= 100

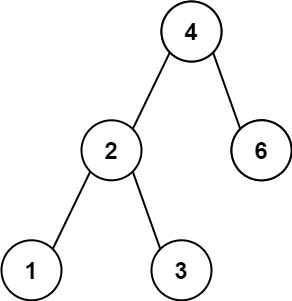
**Solution:**

class Solution {  
 public int diameterOfBinaryTree(TreeNode root) {  
 int[] res = new int[1];  
 if (root == null) return 0;  
 findDiameter(root, res);  
 return res[0];  
 }  
  
 private int findDiameter(TreeNode node, int[] res) {  
  
 if (node == null) return 0;  
  
 int left = findDiameter(node.left, res);  
 int right = findDiameter(node.right, res);  
  
 res[0] = Math.*max*(res[0], left + right);  
  
 return 1 + Math.*max*(left, right);  
 }  
}

**Q 530. Minimum Absolute Difference in BST**

Given the root of a Binary Search Tree (BST), return *the minimum absolute difference between the values of any two different nodes in the tree*.

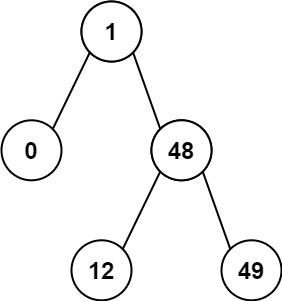
**Example 1:**



**Input:** root = [4,2,6,1,3]

**Output:** 1

**Example 2:**



**Input:** root = [1,0,48,null,null,12,49]

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [2, 104].
* 0 <= Node.val <= 105

**Solution:**

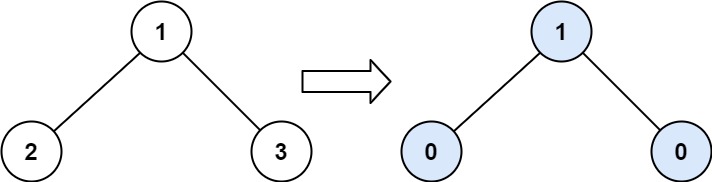
class Solution {  
 public int getMinimumDifference(TreeNode root) {  
 List<Integer> list = new ArrayList<>();  
 inorder(root, list);  
 System.*out*.println(list);  
 int minDiff = findMinDiff(list);  
 return minDiff;  
 }  
  
 private void inorder(TreeNode node, List<Integer> list) {  
 if (node == null) {  
 return;  
 }  
  
 if (node.left != null) {  
 inorder(node.left, list);  
 }  
 list.add(node.val);  
  
 if (node.right != null) {  
 inorder(node.right, list);  
 }  
  
 }  
  
 private int findMinDiff(List<Integer> list) {  
 int min = Integer.*MAX\_VALUE*;  
 for (int i = 0; i < list.size(); i++) {  
 for (int j = i + 1; j < list.size(); j++) {  
 if (Math.*abs*(list.get(i) - list.get(j)) < min) {  
 min = Math.*abs*(list.get(i) - list.get(j));  
 }  
 }  
 }  
  
 return min;  
 }  
}

**Q 563. Binary Tree Tilt**

Given the root of a binary tree, return *the sum of every tree node's****tilt****.*

The **tilt** of a tree node is the **absolute difference** between the sum of all left subtree node **values** and all right subtree node **values**. If a node does not have a left child, then the sum of the left subtree node **values** is treated as 0. The rule is similar if the node does not have a right child.

**Example 1:**



**Input:** root = [1,2,3]

**Output:** 1

**Explanation:**

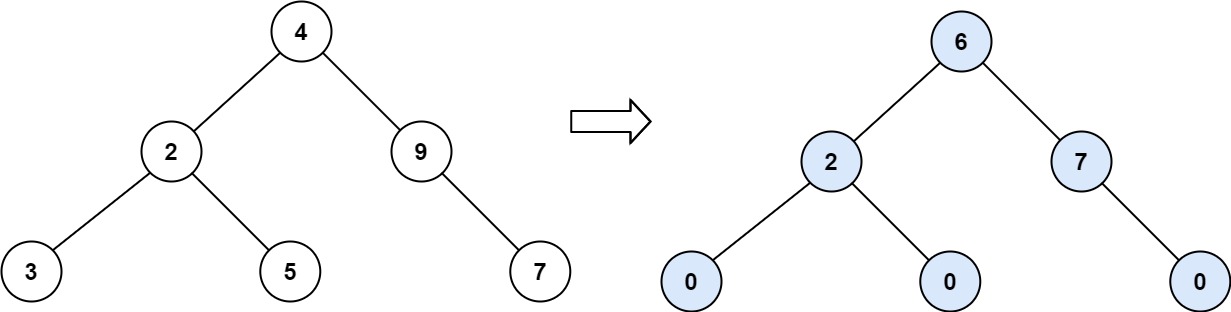
Tilt of node 2 : |0-0| = 0 (no children)

Tilt of node 3 : |0-0| = 0 (no children)

Tilt of node 1 : |2-3| = 1 (left subtree is just left child, so sum is 2; right subtree is just right child, so sum is 3)

Sum of every tilt : 0 + 0 + 1 = 1

**Example 2:**



**Input:** root = [4,2,9,3,5,null,7]

**Output:** 15

**Explanation:**

Tilt of node 3 : |0-0| = 0 (no children)

Tilt of node 5 : |0-0| = 0 (no children)

Tilt of node 7 : |0-0| = 0 (no children)

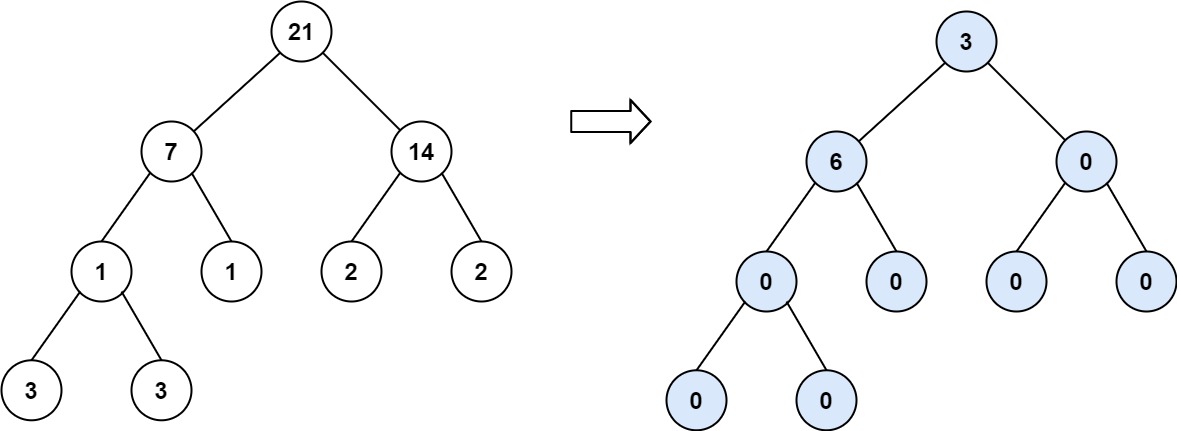
Tilt of node 2 : |3-5| = 2 (left subtree is just left child, so sum is 3; right subtree is just right child, so sum is 5)

Tilt of node 9 : |0-7| = 7 (no left child, so sum is 0; right subtree is just right child, so sum is 7)

Tilt of node 4 : |(3+5+2)-(9+7)| = |10-16| = 6 (left subtree values are 3, 5, and 2, which sums to 10; right subtree values are 9 and 7, which sums to 16)

Sum of every tilt : 0 + 0 + 0 + 2 + 7 + 6 = 15

**Example 3:**



**Input:** root = [21,7,14,1,1,2,2,3,3]

**Output:** 9

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -1000 <= Node.val <= 1000

**Solution:**

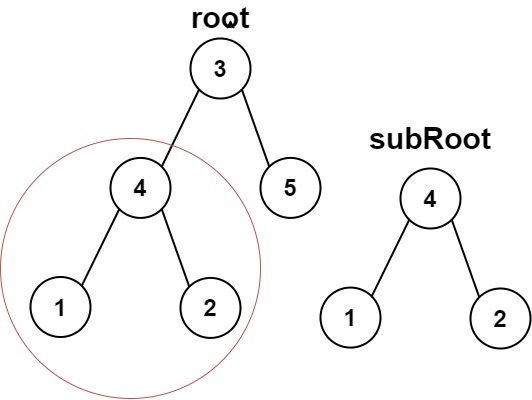
class Solution {  
 int sum = 0;  
  
 public int findTilt(TreeNode root) {  
 findTiltSum(root);  
 return sum;  
 }  
  
 private int findTiltSum(TreeNode node) {  
 if (node == null) return 0;  
  
 int leftTilt = findTiltSum(node.left);  
 int rightTilt = findTiltSum(node.right);  
  
 sum += Math.*abs*(leftTilt - rightTilt);  
  
 return leftTilt + rightTilt + node.val;  
 }  
}

**Q 572. Subtree of Another Tree**

Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

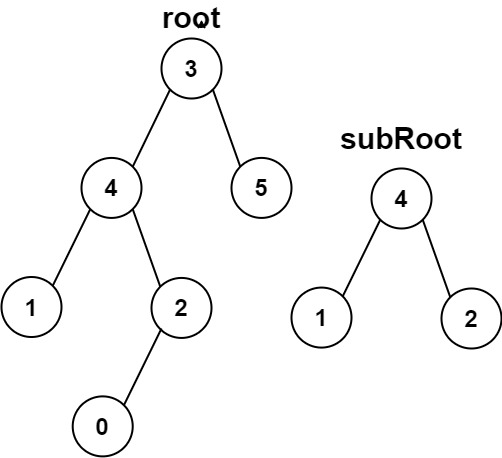
**Example 1:**



**Input:** root = [3,4,5,1,2], subRoot = [4,1,2]

**Output:** true

**Example 2:**



**Input:** root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]

**Output:** false  **Constraints:**

* The number of nodes in the root tree is in the range [1, 2000].
* The number of nodes in the subRoot tree is in the range [1, 1000]. Solution:

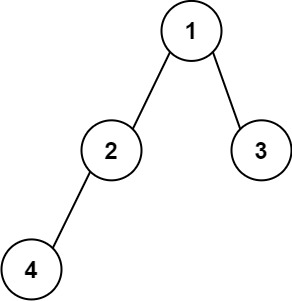
class Solution {  
 public boolean isSubtree(TreeNode root, TreeNode subRoot) {  
  
 if (root == null && subRoot == null) return true;  
 if (root == null || subRoot == null) return false;  
 if (isSame(root, subRoot)) return true;  
 return isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);  
 }  
  
 private boolean isSame(TreeNode node, TreeNode subRoot) {  
  
 if (node == null && subRoot == null) return true;  
 if (node == null || subRoot == null) return false;  
  
 if (node.val != subRoot.val) return false;  
  
 return node.val == subRoot.val && isSame(node.left, subRoot.left) && isSame(node.right, subRoot.right);  
 }  
}

**Q606. Construct String from Binary Tree**

Given the root of a binary tree, construct a string consisting of parenthesis and integers from a binary tree with the preorder traversal way, and return it.

Omit all the empty parenthesis pairs that do not affect the one-to-one mapping relationship between the string and the original binary tree.

**Example 1:**

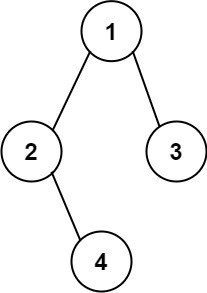


**Input:** root = [1,2,3,4]

**Output:** "1(2(4))(3)"

**Explanation:** Originally, it needs to be "1(2(4)())(3()())", but you need to omit all the unnecessary empty parenthesis pairs. And it will be "1(2(4))(3)"

**Example 2:**



**Input:** root = [1,2,3,null,4]

**Output:** "1(2()(4))(3)"

**Explanation:** Almost the same as the first example, except we cannot omit the first parenthesis pair to break the one-to-one mapping relationship between the input and the output.

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -1000 <= Node.val <= 1000

**Solution:**

class Solution {  
 public String tree2str(TreeNode root) {  
 if (root == null) return "";  
 if (root.left == null && root.right == null) return String.*valueOf*(root.val);  
  
 String leftString = tree2str(root.left);  
 String rightString = tree2str(root.right);  
  
 if (rightString == "") {  
 return String.*valueOf*(root.val + "(" + leftString + ")");  
 } else {  
 return String.*valueOf*(root.val + "(" + leftString + ")" + "(" + rightString + ")");  
 }  
 }  
}

Q**617. Merge Two Binary Trees**

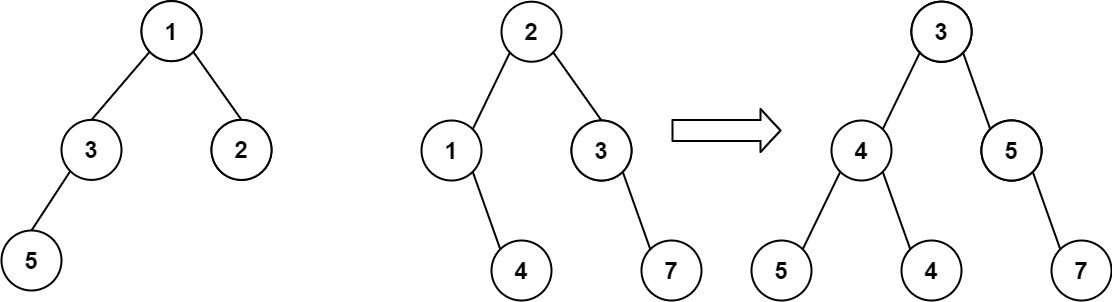
You are given two binary trees root1 and root2.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return *the merged tree*.

**Note:** The merging process must start from the root nodes of both trees.

**Example 1:**



**Input:** root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]

**Output:** [3,4,5,5,4,null,7]

**Example 2:**

**Input:** root1 = [1], root2 = [1,2]

**Output:** [2,2]

**Constraints:**

* The number of nodes in both trees is in the range [0, 2000].
* -104 <= Node.val <= 104

**Solution**:

class Solution {  
 public TreeNode mergeTrees(TreeNode root1, TreeNode root2) {  
  
 if (root1 == null && root2 == null) return null;  
  
 if (root1 == null && root2 != null) return root2;  
  
 if (root1 != null && root2 == null) return root1;  
  
 TreeNode left = new TreeNode();  
 TreeNode right = new TreeNode();  
 int rootValue = root1.val + root2.val;  
 TreeNode node = new TreeNode(rootValue);  
 node.left = mergeTrees(root1.left, root2.left);  
 node.right = mergeTrees(root1.right, root2.right);  
  
 return node;  
  
 }  
}

**Q637. Average of Levels in Binary Tree**

Given the root of a binary tree, return *the average value of the nodes on each level in the form of an array*. Answers within 10-5 of the actual answer will be accepted.

**Example 1:**

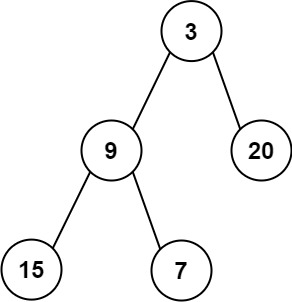


**Input:** root = [3,9,20,null,null,15,7]

**Output:** [3.00000,14.50000,11.00000]

Explanation: The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11.Hence return [3, 14.5, 11].

**Example 2:**



**Input:** root = [3,9,20,15,7]

**Output:** [3.00000,14.50000,11.00000] **Constraints:**

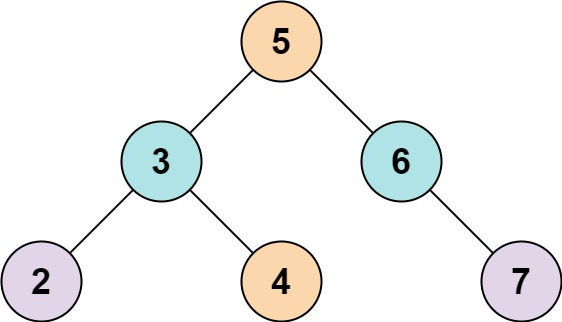
* The number of nodes in the tree is in the range [1, 104].
* -231 <= Node.val <= 231 - 1

class Solution {  
 public List<Double> averageOfLevels(TreeNode root) {  
 List<Double> list = new ArrayList<>();  
 findAvg(root, list);  
 return list;  
 }  
  
 private void findAvg(TreeNode node, List<Double> list) {  
 if (node == null) return;  
 Queue<TreeNode> queue = new LinkedList<>();  
 queue.add(node);  
 while (!queue.isEmpty()) {  
 int size = queue.size();  
 Double sum = 0.0;  
 Double avg = 0.0;  
 for (int i = 0; i < size; i++) {  
 TreeNode temp = queue.poll();  
 // System.out.print(temp.val+" ");  
 sum += temp.val;  
 if (temp.left != null) {  
 queue.add(temp.left);  
 }  
 if (temp.right != null) {  
 queue.add(temp.right);  
 }  
  
 }  
 avg = sum / size;  
 // System.out.println();  
 list.add(avg);  
 }  
 }  
}

**Q653. Two Sum IV - Input is a BST**

Given the root of a Binary Search Tree and a target number k, return *true if there exist two elements in the BST such that their sum is equal to the given target*.

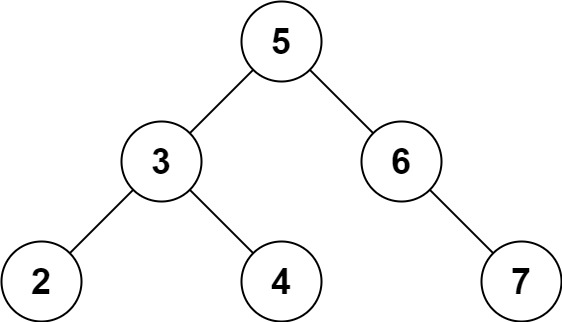
**Example 1:**



**Input:** root = [5,3,6,2,4,null,7], k = 9

**Output:** true

**Example 2:**



**Input:** root = [5,3,6,2,4,null,7], k = 28

**Output:** false

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -104 <= Node.val <= 104
* root is guaranteed to be a **valid** binary search tree.
* -105 <= k <= 105

**Solution:**

class Solution {  
 public boolean findTarget(TreeNode root, int k) {  
 if (root == null) return false;  
 if (root.left == null && root.right == null) return false;  
 List<Integer> list = new ArrayList<>();  
 dfs(root, list);  
 return ifTargetExist(list, k);  
 }  
  
 private void dfs(TreeNode node, List<Integer> list) {  
 if (node == null) return;  
 if (node.left != null) {  
 dfs(node.left, list);  
 }  
 list.add(node.val);  
 if (node.right != null) {  
 dfs(node.right, list);  
 }  
 }  
  
 private boolean ifTargetExist(List<Integer> list, int k) {  
 boolean flag = false;  
 for (int i = 0; i < list.size(); i++) {  
 for (int j = i + 1; j < list.size(); j++) {  
 if (list.get(i) + list.get(j) == k) {  
 flag = true;  
 break;  
 }  
 }  
 }  
  
 return flag;  
 }  
}

**Q559. Maximum Depth of N-ary Tree**

Given a n-ary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

*Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).*

**Example 1:**



**Input:** root = [1,null,3,2,4,null,5,6]

**Output:** 3

**Example 2:**



**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

**Output:** 5

**Constraints:**

* The total number of nodes is in the range [0, 104].
* The depth of the n-ary tree is less than or equal to 1000.

**Solution:**

class Solution {  
 public int maxDepth(Node root) {  
  
 if(root == null) return 0;  
  
 int max = 0;  
  
 for(int i = 0; i < root.children.size() ; i++) {  
 max = Math.*max*(max, maxDepth(root.children.get(i)));  
 }  
 return 1 + max;  
  
 }  
}

**589. N-ary Tree Preorder Traversal**

Given the root of an n-ary tree, return *the preorder traversal of its nodes' values*.

Nary-Tree input serialization is represented in their level order traversal. Each group of children is separated by the null value (See examples)

**Example 1:**



**Input:** root = [1,null,3,2,4,null,5,6]

**Output:** [1,3,5,6,2,4]

**Example 2:**



**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

**Output:** [1,2,3,6,7,11,14,4,8,12,5,9,13,10]

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* 0 <= Node.val <= 104
* The height of the n-ary tree is less than or equal to 1000.

**Solution:**

public class Node {  
 public int val;  
 public List<Node> children;  
  
 public Node() {  
 }  
  
 public Node(int \_val) {  
 val = \_val;  
 }  
  
 public Node(int \_val, List<Node> \_children) {  
 val = \_val;  
 children = \_children;  
 }  
}

class Solution {  
 public List<Integer> preorder(Node root) {  
  
 List<Integer> list = new ArrayList<>();  
 dfs(root, list);  
 return list;  
 }  
  
 private void dfs(Node node, List<Integer> list) {  
 if (node == null) return;  
 list.add(node.val);  
 for (int i = 0; i < node.children.size(); i++) {  
 dfs(node.children.get(i), list);  
 }  
 }  
}

Q**590. N-ary Tree Postorder Traversal**

Given the root of an n-ary tree, return *the postorder traversal of its nodes' values*.

Nary-Tree input serialization is represented in their level order traversal. Each group of children is separated by the null value (See examples)

**Example 1:**



**Input:** root = [1,null,3,2,4,null,5,6]

**Output:** [5,6,3,2,4,1]

**Example 2:**



**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

**Output:** [2,6,14,11,7,3,12,8,4,13,9,10,5,1]

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* 0 <= Node.val <= 104
* The height of the n-ary tree is less than or equal to 1000.

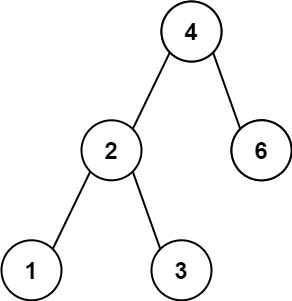
**Solution:**

class Solution {  
 public List<Integer> postorder(Node root) {  
 List<Integer> list = new ArrayList<>();  
 dfs(root, list);  
 return list;  
 }  
  
 private void dfs(Node node, List<Integer> list) {  
 if (node == null) return;  
  
 for (int i = 0; i < node.children.size(); i++) {  
 dfs(node.children.get(i), list);  
 }  
 list.add(node.val);  
 }  
}

Q**783. Minimum Distance Between BST Nodes**

Given the root of a Binary Search Tree (BST), return *the minimum difference between the values of any two different nodes in the tree*.

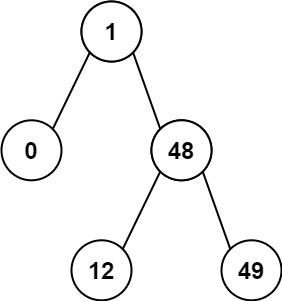
**Example 1:**



**Input:** root = [4,2,6,1,3]

**Output:** 1

**Example 2:**



**Input:** root = [1,0,48,null,null,12,49]

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [2, 100].
* 0 <= Node.val <= 105

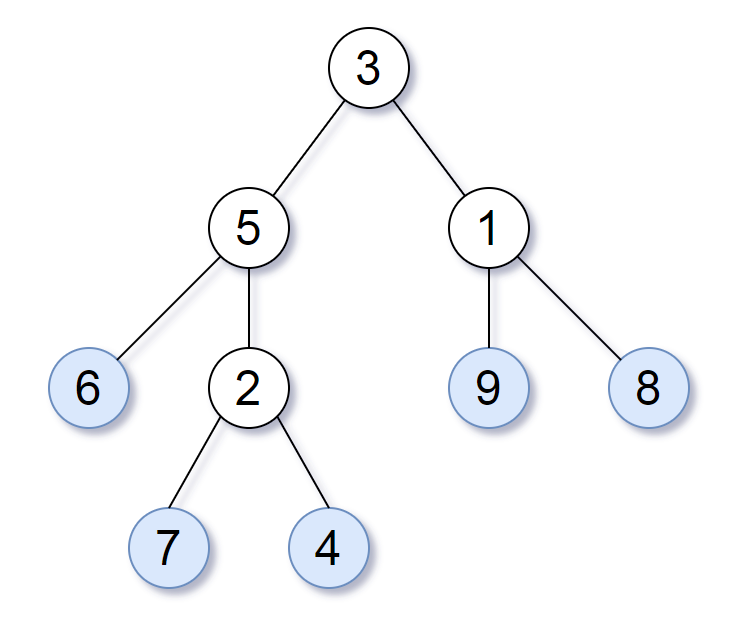
**Note:** This question is the same as 530: <https://leetcode.com/problems/minimum-absolute-difference-in-bst/>

**Solution:**

class Solution {  
 public int minDiffInBST(TreeNode root) {  
 List<Integer> list = new ArrayList<>();  
 dfs(root, list);  
 return minDiff(list);  
 }  
  
 private void dfs(TreeNode node, List<Integer> list) {  
 if (node == null) return;  
 if (node.left != null) {  
 dfs(node.left, list);  
 }  
 list.add(node.val);  
  
 if (node.right != null) {  
 dfs(node.right, list);  
 }  
 }  
  
 private int minDiff(List<Integer> list) {  
  
 int min = Integer.*MAX\_VALUE*;  
 for (int i = 0; i < list.size(); i++) {  
 for (int j = i + 1; j < list.size(); j++) {  
 if (Math.*abs*(list.get(i) - list.get(j)) < min) {  
 min = Math.*abs*(list.get(i) - list.get(j));  
 }  
 }  
 }  
  
 return min;  
 }  
}

**Q872. Leaf-Similar Trees**

Consider all the leaves of a binary tree, from left to right order, the values of those leaves form a **leaf value sequence***.*

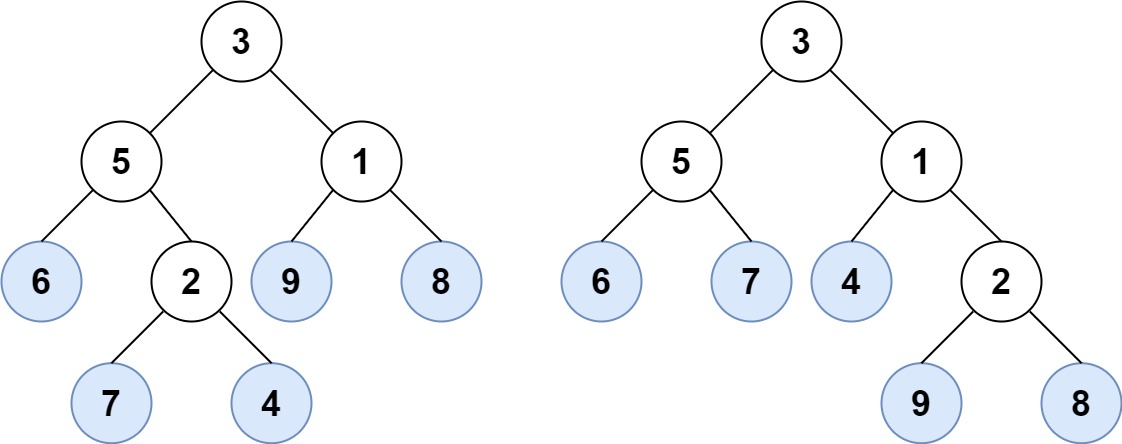


For example, in the given tree above, the leaf value sequence is (6, 7, 4, 9, 8).

Two binary trees are considered *leaf-similar* if their leaf value sequence is the same.

Return true if and only if the two given trees with head nodes root1 and root2 are leaf-similar.

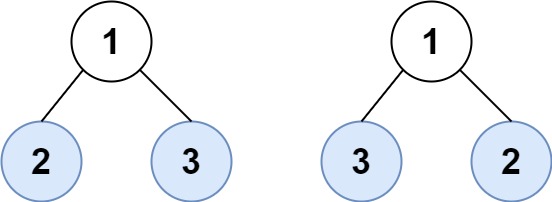
**Example 1:**



**Input:** root1 = [3,5,1,6,2,9,8,null,null,7,4], root2 = [3,5,1,6,7,4,2,null,null,null,null,null,null,9,8]

**Output:** true

**Example 2:**



**Input:** root1 = [1,2,3], root2 = [1,3,2]

**Output:** false

**Constraints:**

* The number of nodes in each tree will be in the range [1, 200].
* Both of the given trees will have values in the range [0, 200].

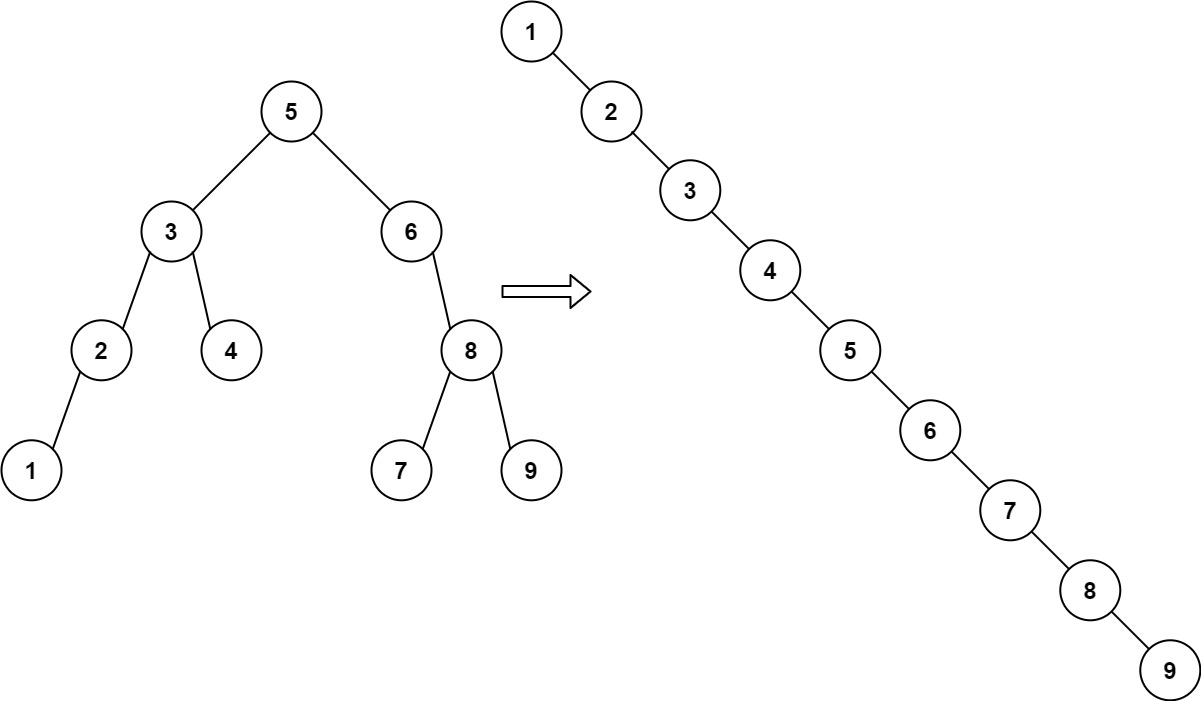
**Solution:**

class Solution {  
 public boolean leafSimilar(TreeNode root1, TreeNode root2) {  
 List<Integer> leaves1 = new ArrayList<>();  
 List<Integer> leaves2 = new ArrayList<>();  
 dfs(root1, leaves1);  
 dfs(root2, leaves2);  
 return haveEqualLeaves(leaves1, leaves2);  
 }  
  
 private void dfs(TreeNode node, List<Integer> list) {  
 if (node == null) return;  
 if (node.left != null) {  
 dfs(node.left, list);  
 }  
 if (isLeaf(node)) {  
 list.add(node.val);  
 }  
 if (node.right != null) {  
 dfs(node.right, list);  
 }  
 }  
  
 private boolean isLeaf(TreeNode node) {  
 if (node == null) return false;  
 if (node.left == null && node.right == null) return true;  
 return false;  
 }  
  
 private boolean haveEqualLeaves(List<Integer> leaves1, List<Integer> leaves2) {  
 boolean flag = false;  
 if (leaves1.size() != leaves2.size()) return false;  
  
 for (int i = 0, j = 0; i < leaves1.size() && j < leaves2.size(); i++, j++) {  
 if (leaves1.get(i) == leaves2.get(j)) {  
 flag = true;  
 } else {  
 flag = false;  
 break;  
 }  
 }  
 return flag;  
 }  
}

**Q897. Increasing Order Search Tree**

Given the root of a binary search tree, rearrange the tree in **in-order** so that the leftmost node in the tree is now the root of the tree, and every node has no left child and only one right child.

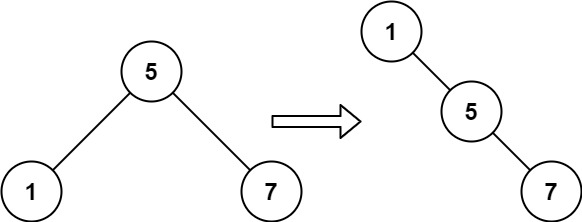
**Example 1:**



**Input:** root = [5,3,6,2,4,null,8,1,null,null,null,7,9]

**Output:** [1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]

**Example 2:**



**Input:** root = [5,1,7]

**Output:** [1,null,5,null,7]

**Constraints:**

* The number of nodes in the given tree will be in the range [1, 100].
* 0 <= Node.val <= 1000

**Solution:**

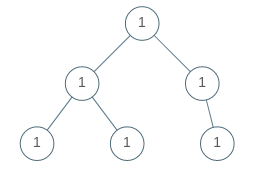
class Solution {  
 public TreeNode increasingBST(TreeNode root) {  
 List<Integer> list = new ArrayList<>();  
 inorder(root, list);  
 System.*out*.println(list);  
 TreeNode res = createIncOrderSearchTree(list);  
 return res;  
 }  
  
 private void inorder(TreeNode node, List<Integer> list) {  
 if (node == null) return;  
 if (node.left != null) {  
 inorder(node.left, list);  
 }  
 list.add(node.val);  
 if (node.right != null) {  
 inorder(node.right, list);  
 }  
 }  
  
 private TreeNode createIncOrderSearchTree(List<Integer> list) {  
 TreeNode root = new TreeNode(list.get(0));  
 TreeNode node = root;  
 for (int i = 1; i < list.size(); i++) {  
 node.right = new TreeNode(list.get(i));  
 node.left = null;  
 node = node.right;  
 }  
  
 return root;  
 }  
}

**Q965. Univalued Binary Tree**

A binary tree is **uni-valued** if every node in the tree has the same value.

Given the root of a binary tree, return true*if the given tree is****uni-valued****, or*false*otherwise.*

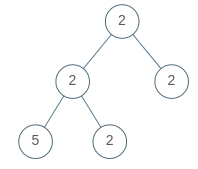
**Example 1:**



**Input:** root = [1,1,1,1,1,null,1]

**Output:** true

**Example 2:**



**Input:** root = [2,2,2,5,2]

**Output:** false

**Constraints:**

* The number of nodes in the tree is in the range [1, 100].
* 0 <= Node.val < 100

**Solution:**

class Solution {  
 public boolean isUnivalTree(TreeNode root) {  
 if (root == null) return false;  
 Set<Integer> set = new HashSet<>();  
 dfs(root, set);  
  
 return set.size() == 1;  
 }  
  
 private void dfs(TreeNode node, Set<Integer> set) {  
 if (node == null) return;  
 if (node.left != null) {  
 dfs(node.left, set);  
 }  
  
 set.add(node.val);  
  
 if (node.right != null) {  
 dfs(node.right, set);  
 }  
 }  
}

**Q1022. Sum of Root To Leaf Binary Numbers**

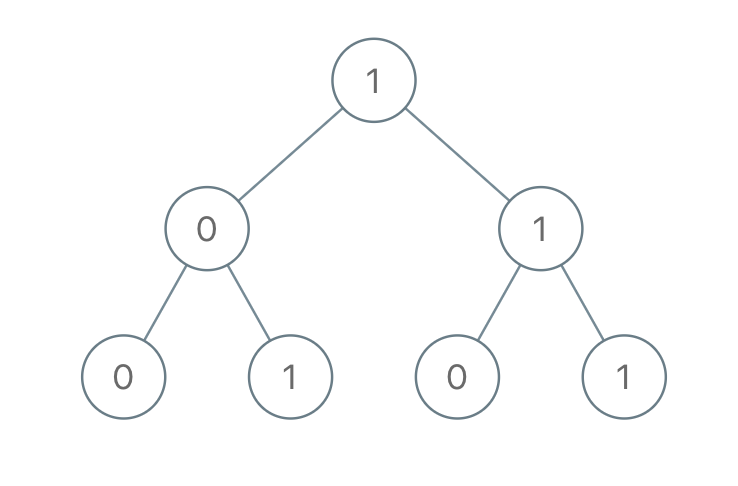
You are given the root of a binary tree where each node has a value 0 or 1. Each root-to-leaf path represents a binary number starting with the most significant bit.

* For example, if the path is 0 -> 1 -> 1 -> 0 -> 1, then this could represent 01101 in binary, which is 13.

For all leaves in the tree, consider the numbers represented by the path from the root to that leaf. Return *the sum of these numbers*.

The test cases are generated so that the answer fits in a **32-bits** integer.

**Example 1:**



**Input:** root = [1,0,1,0,1,0,1]

**Output:** 22

**Explanation:** (100) + (101) + (110) + (111) = 4 + 5 + 6 + 7 = 22

**Example 2:**

**Input:** root = [0]

**Output:** 0

**Constraints:**

* The number of nodes in the tree is in the range [1, 1000].
* Node.val is 0 or 1.

**Solution:**

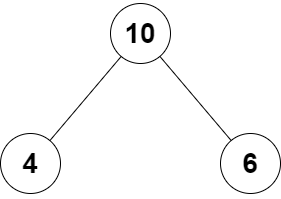
class Solution {  
 public int sumRootToLeaf(TreeNode root) {  
 List<String> list = new ArrayList<>();  
 dfs(root, "", list);  
 int n = convertToDecimalSum(list);  
 return n;  
 }  
  
 private void dfs(TreeNode node, String path, List<String> list) {  
 if (node == null) {  
 return;  
 }  
  
 path += String.*valueOf*(node.val);  
  
 if (node.left == null && node.right == null) {  
 list.add(path);  
 return;  
 }  
  
 if (node.left != null) {  
 dfs(node.left, path, list);  
 }  
  
 if (node.right != null) {  
 dfs(node.right, path, list);  
 }  
 }  
  
 private int convertToDecimalSum(List<String> list) {  
 int sum = 0;  
 for (String s : list) {  
 sum += Integer.*parseInt*(s, 2);  
 }  
 return sum;  
 }  
}

**Q2236. Root Equals Sum of Children**

You are given the root of a **binary tree** that consists of exactly 3 nodes: the root, its left child, and its right child.

Return true *if the value of the root is equal to the****sum****of the values of its two children, or*false*otherwise*.

**Example 1:**



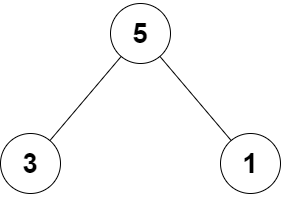
**Input:** root = [10,4,6]

**Output:** true

**Explanation:** The values of the root, its left child, and its right child are 10, 4, and 6, respectively.

10 is equal to 4 + 6, so we return true.

**Example 2:**



**Input:** root = [5,3,1]

**Output:** false

**Explanation:** The values of the root, its left child, and its right child are 5, 3, and 1, respectively.

5 is not equal to 3 + 1, so we return false.

**Constraints:**

* The tree consists only of the root, its left child, and its right child.
* -100 <= Node.val <= 100

**Solution:**

class Solution {  
 public boolean checkTree(TreeNode root) {  
 return root.val == (root.left.val + root.right.val);  
 }  
}

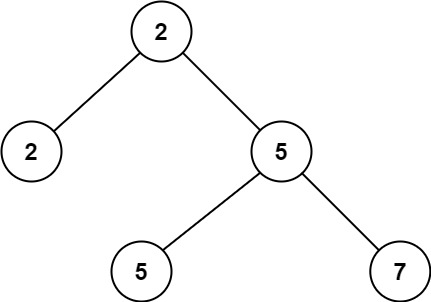
**Q671. Second Minimum Node In a Binary Tree**

Given a non-empty special binary tree consisting of nodes with the non-negative value, where each node in this tree has exactly two or zero sub-node. If the node has two sub-nodes, then this node's value is the smaller value among its two sub-nodes. More formally, the property root.val = min(root.left.val, root.right.val) always holds.

Given such a binary tree, you need to output the **second minimum** value in the set made of all the nodes' value in the whole tree.

If no such second minimum value exists, output -1 instead.

**Example 1:**

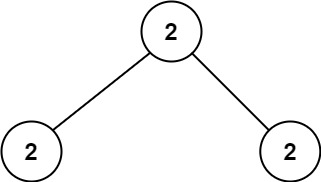


**Input:** root = [2,2,5,null,null,5,7]

**Output:** 5

**Explanation:** The smallest value is 2, the second smallest value is 5.

**Example 2:**



**Input:** root = [2,2,2]

**Output:** -1

**Explanation:** The smallest value is 2, but there isn't any second smallest value.

**Constraints:**

* The number of nodes in the tree is in the range [1, 25].
* 1 <= Node.val <= 231 - 1
* root.val == min(root.left.val, root.right.val) for each internal node of the tree.**Solution:**

class Solution {  
 public int findSecondMinimumValue(TreeNode root) {  
  
 Set<Integer> set = new HashSet<>();  
 dfs(root, set);  
 return secondMin(set);  
 }  
  
 private void dfs(TreeNode node, Set<Integer> set) {  
 if (node == null) {  
 return;  
 }  
  
 if (node.left != null) {  
 dfs(node.left, set);  
 }  
  
 set.add(node.val);  
  
 if (node.right != null) {  
 dfs(node.right, set);  
 }  
 }  
  
 private int secondMin(Set<Integer> set) {  
 if (set.size() == 1) return -1;  
 List<Integer> list = new ArrayList<>(set);  
 Collections.*sort*(list);  
 return list.get(1);  
 }  
}

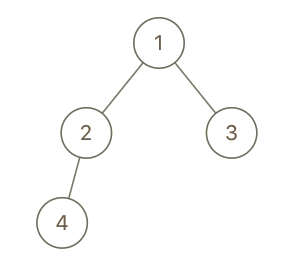
Q**993. Cousins in Binary Tree**

Given the root of a binary tree with unique values and the values of two different nodes of the tree x and y, return true *if the nodes corresponding to the values*x*and*y*in the tree are****cousins****, or*false*otherwise.*

Two nodes of a binary tree are **cousins** if they have the same depth with different parents.

Note that in a binary tree, the root node is at the depth 0, and children of each depth k node are at the depth k + 1.

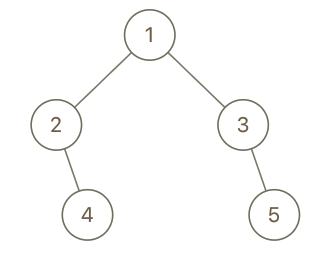
**Example 1:**



**Input:** root = [1,2,3,4], x = 4, y = 3

**Output:** false

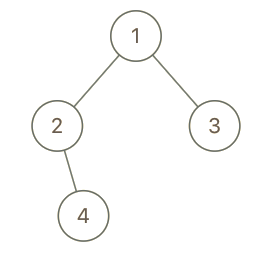
**Example 2:**



**Input:** root = [1,2,3,null,4,null,5], x = 5, y = 4

**Output:** true

**Example 3:**



**Input:** root = [1,2,3,null,4], x = 2, y = 3

**Output:** false

**Constraints:**

* The number of nodes in the tree is in the range [2, 100].
* 1 <= Node.val <= 100
* Each node has a **unique** value.
* x != y
* x and y are exist in the tree.

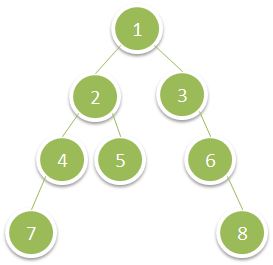
**Solution:**

class Solution {  
 public boolean isCousins(TreeNode root, int x, int y) {  
 if (root == null) return false;  
  
 TreeNode xParent = null;  
 TreeNode yParent = null;  
  
 Queue<TreeNode> queue = new LinkedList<>();  
 queue.offer(root);  
 while (!queue.isEmpty()) {  
 int size = queue.size();  
 for (int i = 0; i < size; i++) {  
 TreeNode node = queue.poll();  
 if (node.left != null) {  
 queue.offer(node.left);  
  
 if (node.left.val == x) {  
 xParent = node;  
 }  
  
 if (node.left.val == y) {  
 yParent = node;  
 }  
 }  
  
 if (node.right != null) {  
 queue.offer(node.right);  
  
 if (node.right.val == x) {  
 xParent = node;  
 }  
  
 if (node.right.val == y) {  
 yParent = node;  
 }  
 }  
 if (xParent != null && yParent != null) {  
 return xParent != yParent;  
 }  
 }  
  
 if ((xParent == null && yParent != null) || (xParent != null && yParent == null)) {  
 return false;  
 }  
 }  
  
 return false;  
 }  
}

**Q1302. Deepest Leaves Sum**

Given the root of a binary tree, return *the sum of values of its deepest leaves*.

**Example 1:**



**Input:** root = [1,2,3,4,5,null,6,7,null,null,null,null,8]

**Output:** 15

**Example 2:**

**Input:** root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]

**Output:** 19

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* 1 <= Node.val <= 100

**Solution:**

class Solution {  
 public int deepestLeavesSum(TreeNode root) {  
 if (root == null) {  
 return 0;  
 }  
 List<List<Integer>> res = new ArrayList<>();  
 Queue<TreeNode> queue = new LinkedList<>();  
 queue.offer(root);  
 int sum = 0;  
 while (!queue.isEmpty()) {  
 int size = queue.size();  
 List<Integer> rowList = new ArrayList<>();  
 for (int i = 0; i < size; i++) {  
 TreeNode node = queue.poll();  
 rowList.add(node.val);  
 if (node.left != null) {  
 queue.offer(node.left);  
 }  
  
 if (node.right != null) {  
 queue.offer(node.right);  
 }  
 }  
  
 res.add(rowList);  
 System.*out*.println(rowList);  
 }  
  
 List<Integer> leaf = res.get(res.size() - 1);  
  
 return leaf.stream().mapToInt(Integer::intValue).sum();  
 }  
  
}

**Q1379. Find a Corresponding Node of a Binary Tree in a Clone of That Tree**

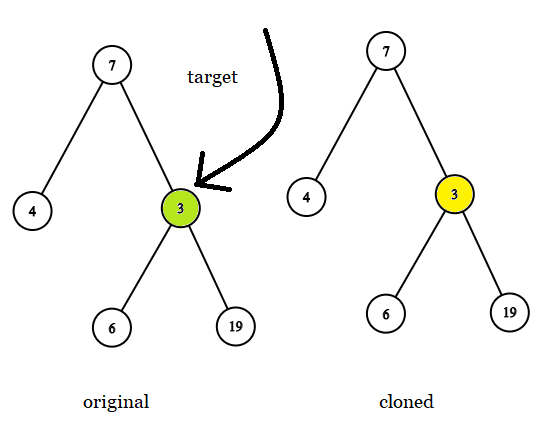
Given two binary trees original and cloned and given a reference to a node target in the original tree.

The cloned tree is a **copy of** the original tree.

Return *a reference to the same node* in the cloned tree.

**Note** that you are **not allowed** to change any of the two trees or the target node and the answer **must be** a reference to a node in the cloned tree.

**Example 1:**

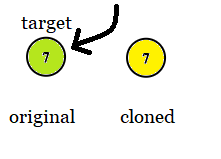


**Input:** tree = [7,4,3,null,null,6,19], target = 3

**Output:** 3

**Explanation:** In all examples the original and cloned trees are shown. The target node is a green node from the original tree. The answer is the yellow node from the cloned tree.

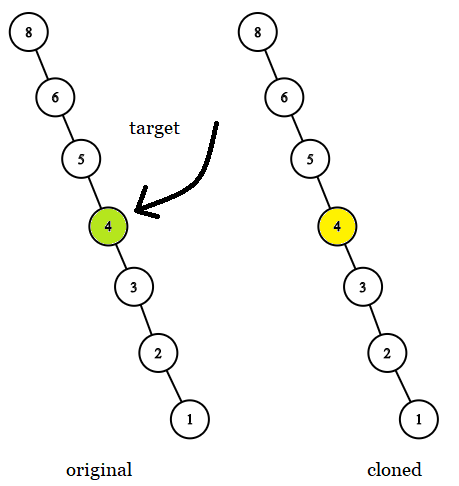
**Example 2:**



**Input:** tree = [7], target = 7

**Output:** 7

**Example 3:**



**Input:** tree = [8,null,6,null,5,null,4,null,3,null,2,null,1], target = 4

**Output:** 4 **Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* The values of the nodes of the tree are unique.
* target node is a node from the original tree and is not null.

**Solution:**

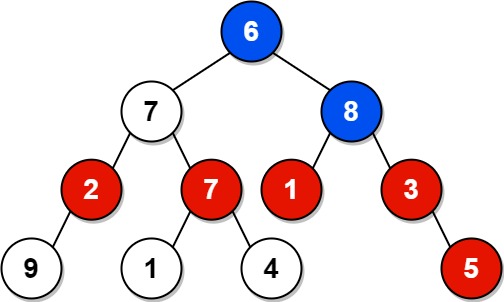
class Solution {  
 public final TreeNode getTargetCopy(final TreeNode original, final TreeNode cloned, final TreeNode target) {  
  
 if (original == null || cloned == null) return null;  
  
 if (original.val == cloned.val && cloned.val == target.val) {  
 return cloned;  
 }  
 TreeNode left = getTargetCopy(original.left, cloned.left, target);  
 TreeNode right = getTargetCopy(original.right, cloned.right, target);  
 if (left == null) return right;  
 if (right == null) return left;  
 return null;  
 }  
}

**Q1315. Sum of Nodes with Even-Valued Grandparent**

Given the root of a binary tree, return *the sum of values of nodes with an****even-valued grandparent***. If there are no nodes with an **even-valued grandparent**, return 0.

A **grandparent** of a node is the parent of its parent if it exists.

**Example 1:**



**Input:** root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]

**Output:** 18

**Explanation:** The red nodes are the nodes with even-value grandparent while the blue nodes are the even-value grandparents.

**Example 2:**



**Input:** root = [1]

**Output:** 0

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* 1 <= Node.val <= 100

**Solution:**

class Solution {  
 public int sumEvenGrandparent(TreeNode root) {  
 int[] sum = new int[1];  
 dfs(root, sum);  
 return sum[0];  
 }  
  
 private void dfs(TreeNode node, int[] sum) {  
 if (node == null) return;  
  
 if (node.left != null) {  
 dfs(node.left, sum);  
 }  
 if (node.left != null && node.val % 2 == 0) {  
 if (node.left.left != null) {  
 sum[0] += node.left.left.val;  
 }  
 if (node.left.right != null) {  
 sum[0] += node.left.right.val;  
 }  
 }  
 if (node.right != null && node.val % 2 == 0) {  
 if (node.right.left != null) {  
 sum[0] += node.right.left.val;  
 }  
 if (node.right.right != null) {  
 sum[0] += node.right.right.val;  
 }  
 }  
  
 if (node.right != null) {  
 dfs(node.right, sum);  
 }  
 }  
}

**Q98. Validate Binary Search Tree**

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.

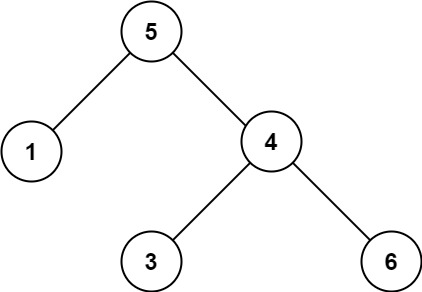
**Example 1:**



**Input:** root = [2,1,3]

**Output:** true

**Example 2:**



**Input:** root = [5,1,4,null,null,3,6]

**Output:** false

**Explanation:** The root node's value is 5 but its right child's value is 4.

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -231 <= Node.val <= 231 - 1

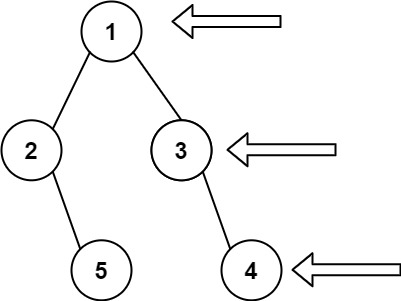
**Solution:**

class Solution {  
 public boolean isValidBST(TreeNode root) {  
 List<Integer> list = new ArrayList<>();  
 dfsIn(root, list);  
 return isValid(list);  
 }  
  
 private void dfsIn(TreeNode node, List<Integer> list) {  
 if (node == null) return;  
 if (node.left != null) {  
 dfsIn(node.left, list);  
 }  
 list.add(node.val);  
 if (node.right != null) {  
 dfsIn(node.right, list);  
 }  
 }  
  
 private boolean isValid(List<Integer> list) {  
 System.*out*.println(list);  
 for (int i = 0; i < list.size() - 1; i++) {  
 if (list.get(i) >= list.get(i + 1)) {  
 return false;  
 }  
 }  
 return true;  
 }  
}

**Q199. Binary Tree Right Side View**

Given the root of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.

**Example 1:**



**Input:** root = [1,2,3,null,5,null,4]

**Output:** [1,3,4]

**Example 2:**

**Input:** root = [1,null,3]

**Output:** [1,3]

**Example 3:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 100].
* -100 <= Node.val <= 100

**Solution:**

class Solution {  
 public List<Integer> rightSideView(TreeNode root) {  
  
 if (root == null) return List.*of*();  
 List<Integer> list = new LinkedList<>();  
 findRightView(root, list, 0);  
 return list;  
 }  
  
 private void findRightView(TreeNode node, List<Integer> list, int level) {  
 if (node == null) return;  
 if (list.size() <= level) {  
 list.add(level, node.val);  
 } else {  
 list.set(level, node.val);  
 }  
  
 findRightView(node.left, list, level + 1);  
 findRightView(node.right, list, level + 1);  
 }  
}

**Approach 2:**

class Solution {  
 public List<Integer> rightSideView(TreeNode root) {  
  
 if (root == null) return List.*of*();  
 List<Integer> list = new LinkedList<>();  
 findRightView(root, list, 0);  
 return list;  
 }  
  
 private void findRightView(TreeNode node, List<Integer> list, int level) {  
 if (node == null) return;  
  
 Queue<TreeNode> queue = new LinkedList<>();  
  
 List<List<Integer>> levels = new ArrayList<>();  
 queue.add(node);  
 while (!queue.isEmpty()) {  
 int size = queue.size();  
 List<Integer> eachLevel = new ArrayList<>();  
 for (int i = 0; i < size; i++) {  
 TreeNode temp = queue.poll();  
 eachLevel.add(temp.val);  
 if (temp.left != null) {  
 queue.add(temp.left);  
 }  
 if (temp.right != null) {  
 queue.add(temp.right);  
 }  
 }  
 levels.add(eachLevel);  
 }  
 for (List<Integer> list1 : levels) {  
 list.add(list1.get(list1.size() - 1));  
 }  
 }  
}

**Q. Find left view of a Binary Tree**

class Solution {  
 public List<Integer> leftSideView(TreeNode root) {  
  
 if (root == null) return List.*of*();  
 List<Integer> list = new LinkedList<>();  
 findLeftView(root, list, 0);  
 return list;  
 }  
  
 private void findLeftView(TreeNode node, List<Integer> list, int level) {  
 if (node == null) return;  
 if (list.size() == 0) {  
 list.add(level, node.val);  
 }   
  
 findLeftView(node.left, list, level + 1);  
 findLeftView(node.right, list, level + 1);  
 }  
}

**Approach 2:**

class Solution {  
 public List<Integer> leftSideView(TreeNode root) {  
  
 if (root == null) return List.*of*();  
 List<Integer> list = new LinkedList<>();  
 findLeftView(root, list, 0);  
 return list;  
 }  
  
 private void findLeftView(TreeNode node, List<Integer> list, int level) {  
 if (node == null) return;  
  
 Queue<TreeNode> queue = new LinkedList<>();  
  
 List<List<Integer>> levels = new ArrayList<>();  
 queue.add(node);  
 while (!queue.isEmpty()) {  
 int size = queue.size();  
 List<Integer> eachLevel = new ArrayList<>();  
 for (int i = 0; i < size; i++) {  
 TreeNode temp = queue.poll();  
 eachLevel.add(temp.val);  
 if (temp.left != null) {  
 queue.add(temp.left);  
 }  
 if (temp.right != null) {  
 queue.add(temp.right);  
 }  
 }  
 levels.add(eachLevel);  
 }  
 for (List<Integer> list1 : levels) {  
 list.add(list1.get(0));  
 }  
 }  
}

**107. Binary Tree Level Order Traversal II (Bottom to top)**

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

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[15,7],[9,20],[3]]

**Example 2:**

**Input:** root = [1]

**Output:** [[1]]

**Example 3:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 2000].
* -1000 <= Node.val <= 1000

**Solution:**

class Solution {  
 public List<List<Integer>> levelOrderBottom(TreeNode root) {  
 if (root == null) return List.*of*();  
 List<List<Integer>> list = new ArrayList<>();  
 bfs(root, list);  
 return list;  
 }  
  
 private void bfs(TreeNode node, List<List<Integer>> list) {  
  
 Queue<TreeNode> queue = new LinkedList<>();  
 Stack<List<Integer>> stack = new Stack<>();  
 queue.add(node);  
 while (!queue.isEmpty()) {  
 int size = queue.size();  
 List<Integer> eachLevel = new ArrayList<>();  
  
 for (int i = 0; i < size; i++) {  
 TreeNode temp = queue.poll();  
 eachLevel.add(temp.val);  
  
 if (temp.left != null) {  
 queue.add(temp.left);  
 }  
  
 if (temp.right != null) {  
 queue.add(temp.right);  
 }  
 }  
 stack.push(eachLevel);  
 }  
 while (!stack.isEmpty()) {  
 list.add(stack.pop());  
 }  
 }  
}