**Experiment 6**

**Name: Akansha Sonker Branch: BE in IT**

**Uid: 22BET10023 Semester: 6TH**

**Section/Group: 22BET\_IO1\_702/B Date Of Performance: 28/2/2025**

**Subject Name: Advanced Programming Lab-2 Subject Code: 22ITT-367**

* **Aim:**

1. Determine the maximum depth (or height) of a binary tree by counting the number of nodes along the longest path from the root to a leaf node.
2. To verify whether a given binary tree satisfies the properties of a Binary Search Tree (BST), ensuring all left descendants are less and right descendants are greater than the node.
3. To check whether a binary tree is symmetric around its center, meaning the left and right subtrees are mirror images of each other.
4. To traverse a binary tree level by level, returning the nodes' values at each depth from top to bottom.
5. To construct a height-balanced Binary Search Tree (BST) from a sorted array, ensuring minimal height for optimal search time.
6. To implement an inorder traversal of a binary tree, visiting nodes in the order: left subtree, root, right subtree.
7. To build a binary tree using given inorder and postorder traversal sequences, reconstructing the tree structure accurately.
8. To find the kth smallest element in a BST by performing an inorder traversal, leveraging the tree's ordered properties.
9. To link each node in a binary tree to its next right node at the same level, forming a connected structure across each level.

* **Code:**

1. **Maximum depth of Binary Tree:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

// Recursive DFS approach

public int maxDepth(TreeNode root) {

if (root == null) return 0;

int leftDepth = maxDepth(root.left);

int rightDepth = maxDepth(root.right);

return Math.max(leftDepth, rightDepth) + 1;

}

// Iterative BFS approach (Level Order Traversal)

public int maxDepthBFS(TreeNode root) {

if (root == null) return 0;

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

queue.add(root);

int depth = 0;

while (!queue.isEmpty()) {

int levelSize = queue.size();

for (int i = 0; i < levelSize; i++) {

TreeNode node = queue.poll();

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

depth++;

}

return depth;

}

// Iterative DFS approach (using stack)

public int maxDepthDFS(TreeNode root) {

if (root == null) return 0;

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

Stack<Integer> depths = new Stack<>();

stack.push(root);

depths.push(1);

int maxDepth = 0;

while (!stack.isEmpty()) {

TreeNode node = stack.pop();

int currentDepth = depths.pop();

maxDepth = Math.max(maxDepth, currentDepth);

if (node.left != null) {

stack.push(node.left);

depths.push(currentDepth + 1);

}

if (node.right != null) {

stack.push(node.right);

depths.push(currentDepth + 1);

}

}

return maxDepth;

}

}

// Main method for testing

public class MaxDepthBinaryTree {

public static void main(String[] args) {

Solution solution = new Solution();

TreeNode root = new TreeNode(3);

root.left = new TreeNode(9);

root.right = new TreeNode(20);

root.right.left = new TreeNode(15);

root.right.right = new TreeNode(7);

System.out.println("Max Depth (Recursive DFS): " + solution.maxDepth(root));

System.out.println("Max Depth (Iterative BFS): " + solution.maxDepthBFS(root));

System.out.println("Max Depth (Iterative DFS): " + solution.maxDepthDFS(root));

}

}

1. **Validate Binary Search Tree:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

// Validate Binary Search Tree using Recursion

public boolean isValidBST(TreeNode root) {

return validate(root, null, null);

}

private boolean validate(TreeNode node, Integer low, Integer high) {

if (node == null) return true;

if ((low != null && node.val <= low) || (high != null && node.val >= high)) return false;

return validate(node.left, low, node.val) && validate(node.right, node.val, high);

}

// Inorder traversal approach

public boolean isValidBSTInorder(TreeNode root) {

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

TreeNode prev = null;

while (!stack.isEmpty() || root != null) {

while (root != null) {

stack.push(root);

root = root.left;

}

root = stack.pop();

if (prev != null && root.val <= prev.val) return false;

prev = root;

root = root.right;

}

return true;

}

}

// Main method for testing

public class ValidateBinarySearchTree {

public static void main(String[] args) {

Solution solution = new Solution();

TreeNode root = new TreeNode(2);

root.left = new TreeNode(1);

root.right = new TreeNode(3);

System.out.println("Is Valid BST (Recursive): " + solution.isValidBST(root));

System.out.println("Is Valid BST (Inorder Traversal): " + solution.isValidBSTInorder(root));

}

}

1. **Symmetric Tree:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

// Check if a tree is symmetric (mirror of itself)

public boolean isSymmetric(TreeNode root) {

if (root == null) return true;

return isMirror(root.left, root.right);

}

private boolean isMirror(TreeNode t1, TreeNode t2) {

if (t1 == null && t2 == null) return true;

if (t1 == null || t2 == null) return false;

return (t1.val == t2.val) && isMirror(t1.right, t2.left) && isMirror(t1.left, t2.right);

}

// Iterative approach using queue

public boolean isSymmetricIterative(TreeNode root) {

if (root == null) return true;

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

queue.add(root.left);

queue.add(root.right);

while (!queue.isEmpty()) {

TreeNode t1 = queue.poll();

TreeNode t2 = queue.poll();

if (t1 == null && t2 == null) continue;

if (t1 == null || t2 == null || t1.val != t2.val) return false;

queue.add(t1.left);

queue.add(t2.right);

queue.add(t1.right);

queue.add(t2.left);

}

return true;

}

}

// Main method for testing

public class SymmetricTree {

public static void main(String[] args) {

Solution solution = new Solution();

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(2);

root.left.left = new TreeNode(3);

root.left.right = new TreeNode(4);

root.right.left = new TreeNode(4);

root.right.right = new TreeNode(3);

System.out.println("Is Symmetric (Recursive): " + solution.isSymmetric(root));

System.out.println("Is Symmetric (Iterative): " + solution.isSymmetricIterative(root));

}

}

1. **Binary Tree Level Order Traversal:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

// Binary Tree Level Order Traversal

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null) return result;

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

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

List<Integer> currentLevel = new ArrayList<>();

for (int i = 0; i < levelSize; i++) {

TreeNode node = queue.poll();

currentLevel.add(node.val);

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

result.add(currentLevel);

}

return result;

}

}

// Main method for testing

public class BinaryTreeLevelOrderTraversal {

public static void main(String[] args) {

Solution solution = new Solution();

TreeNode root = new TreeNode(3);

root.left = new TreeNode(9);

root.right = new TreeNode(20);

root.right.left = new TreeNode(15);

root.right.right = new TreeNode(7);

System.out.println("Level Order Traversal: " + solution.levelOrder(root));

}

}

1. **Convert Sorted Array to Binary Search Tree:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

// Convert Sorted Array to Binary Search Tree

public TreeNode sortedArrayToBST(int[] nums) {

if (nums == null || nums.length == 0) return null;

return buildBST(nums, 0, nums.length - 1);

}

private TreeNode buildBST(int[] nums, int left, int right) {

if (left > right) return null;

int mid = left + (right - left) / 2;

TreeNode node = new TreeNode(nums[mid]);

node.left = buildBST(nums, left, mid - 1);

node.right = buildBST(nums, mid + 1, right);

return node;

}

// Binary Tree Level Order Traversal

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null) return result;

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

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

List<Integer> currentLevel = new ArrayList<>();

for (int i = 0; i < levelSize; i++) {

TreeNode node = queue.poll();

currentLevel.add(node.val);

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

result.add(currentLevel);

}

return result;

}

}

1. **Binary Tree Inorder Traversal:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

// Convert Sorted Array to Binary Search Tree

public TreeNode sortedArrayToBST(int[] nums) {

if (nums == null || nums.length == 0) return null;

return buildBST(nums, 0, nums.length - 1);

}

private TreeNode buildBST(int[] nums, int left, int right) {

if (left > right) return null;

int mid = left + (right - left) / 2;

TreeNode node = new TreeNode(nums[mid]);

node.left = buildBST(nums, left, mid - 1);

node.right = buildBST(nums, mid + 1, right);

return node;

}

// Binary Tree Inorder Traversal

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

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

TreeNode current = root;

while (current != null || !stack.isEmpty()) {

while (current != null) {

stack.push(current);

current = current.left;

}

current = stack.pop();

result.add(current.val);

current = current.right;

}

return result;

}

// Binary Tree Level Order Traversal

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null) return result;

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

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

List<Integer> currentLevel = new ArrayList<>();

for (int i = 0; i < levelSize; i++) {

TreeNode node = queue.poll();

currentLevel.add(node.val);

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

result.add(currentLevel);

}

return result;

}

}

// Main method for testing

public class BinaryTreeTraversal {

public static void main(String[] args) {

Solution solution = new Solution();

int[] nums = {-10, -3, 0, 5, 9};

TreeNode root = solution.sortedArrayToBST(nums);

System.out.println("Level Order Traversal: " + solution.levelOrder(root));

System.out.println("Inorder Traversal: " + solution.inorderTraversal(root));

}

}

1. **Construct Binary Tree From Inorder and Postorder Traversal:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

// Convert Sorted Array to Binary Search Tree

public TreeNode sortedArrayToBST(int[] nums) {

if (nums == null || nums.length == 0) return null;

return buildBST(nums, 0, nums.length - 1);

}

private TreeNode buildBST(int[] nums, int left, int right) {

if (left > right) return null;

int mid = left + (right - left) / 2;

TreeNode node = new TreeNode(nums[mid]);

node.left = buildBST(nums, left, mid - 1);

node.right = buildBST(nums, mid + 1, right);

return node;

}

// Construct Binary Tree from Inorder and Postorder Traversal

public TreeNode buildTree(int[] inorder, int[] postorder) {

if (inorder == null || postorder == null || inorder.length != postorder.length) return null;

Map<Integer, Integer> inorderMap = new HashMap<>();

for (int i = 0; i < inorder.length; i++) {

inorderMap.put(inorder[i], i);

}

return buildTreeHelper(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1, inorderMap);

}

private TreeNode buildTreeHelper(int[] inorder, int inStart, int inEnd, int[] postorder, int postStart, int postEnd, Map<Integer, Integer> inorderMap) {

if (inStart > inEnd || postStart > postEnd) return null;

TreeNode root = new TreeNode(postorder[postEnd]);

int inRoot = inorderMap.get(root.val);

int leftTreeSize = inRoot - inStart;

root.left = buildTreeHelper(inorder, inStart, inRoot - 1, postorder, postStart, postStart + leftTreeSize - 1, inorderMap);

root.right = buildTreeHelper(inorder, inRoot + 1, inEnd, postorder, postStart + leftTreeSize, postEnd - 1, inorderMap);

return root;

}

// Binary Tree Inorder Traversal

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

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

TreeNode current = root;

while (current != null || !stack.isEmpty()) {

while (current != null) {

stack.push(current);

current = current.left;

}

current = stack.pop();

result.add(current.val);

current = current.right;

}

return result;

}

// Binary Tree Level Order Traversal

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null) return result;

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

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

List<Integer> currentLevel = new ArrayList<>();

for (int i = 0; i < levelSize; i++) {

TreeNode node = queue.poll();

currentLevel.add(node.val);

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

result.add(currentLevel);

}

return result;

}

}

// Main method for testing

public class BinaryTreeTraversal {

public static void main(String[] args) {

Solution solution = new Solution();

int[] nums = {-10, -3, 0, 5, 9};

TreeNode root = solution.sortedArrayToBST(nums);

System.out.println("Level Order Traversal: " + solution.levelOrder(root));

System.out.println("Inorder Traversal: " + solution.inorderTraversal(root));

int[] inorder = {9, 3, 15, 20, 7};

int[] postorder = {9, 15, 7, 20, 3};

TreeNode constructedTree = solution.buildTree(inorder, postorder);

System.out.println("Constructed Tree Level Order: " + solution.levelOrder(constructedTree));

}

}

1. **Kth Smallest Element in a BST:**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

// Convert Sorted Array to Binary Search Tree

public TreeNode sortedArrayToBST(int[] nums) {

if (nums == null || nums.length == 0) return null;

return buildBST(nums, 0, nums.length - 1);

}

private TreeNode buildBST(int[] nums, int left, int right) {

if (left > right) return null;

int mid = left + (right - left) / 2;

TreeNode node = new TreeNode(nums[mid]);

node.left = buildBST(nums, left, mid - 1);

node.right = buildBST(nums, mid + 1, right);

return node;

}

// Construct Binary Tree from Inorder and Postorder Traversal

public TreeNode buildTree(int[] inorder, int[] postorder) {

if (inorder == null || postorder == null || inorder.length != postorder.length) return null;

Map<Integer, Integer> inorderMap = new HashMap<>();

for (int i = 0; i < inorder.length; i++) {

inorderMap.put(inorder[i], i);

}

return buildTreeHelper(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1, inorderMap);

}

private TreeNode buildTreeHelper(int[] inorder, int inStart, int inEnd, int[] postorder, int postStart, int postEnd, Map<Integer, Integer> inorderMap) {

if (inStart > inEnd || postStart > postEnd) return null;

TreeNode root = new TreeNode(postorder[postEnd]);

int inRoot = inorderMap.get(root.val);

int leftTreeSize = inRoot - inStart;

root.left = buildTreeHelper(inorder, inStart, inRoot - 1, postorder, postStart, postStart + leftTreeSize - 1, inorderMap);

root.right = buildTreeHelper(inorder, inRoot + 1, inEnd, postorder, postStart + leftTreeSize, postEnd - 1, inorderMap);

return root;

}

// Kth Smallest Element in a BST

public int kthSmallest(TreeNode root, int k) {

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

TreeNode current = root;

int count = 0;

while (current != null || !stack.isEmpty()) {

while (current != null) {

stack.push(current);

current = current.left;

}

current = stack.pop();

count++;

if (count == k) return current.val;

current = current.right;

}

return -1; // This should never be reached if k is valid

}

// Binary Tree Inorder Traversal

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

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

TreeNode current = root;

while (current != null || !stack.isEmpty()) {

while (current != null) {

stack.push(current);

current = current.left;

}

current = stack.pop();

result.add(current.val);

current = current.right;

}

return result;

}

// Binary Tree Level Order Traversal

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null) return result;

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

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

List<Integer> currentLevel = new ArrayList<>();

for (int i = 0; i < levelSize; i++) {

TreeNode node = queue.poll();

currentLevel.add(node.val);

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

result.add(currentLevel);

}

return result;

}

}

// Main method for testing

public class BinaryTreeTraversal {

public static void main(String[] args) {

Solution solution = new Solution();

int[] nums = {-10, -3, 0, 5, 9};

TreeNode root = solution.sortedArrayToBST(nums);

System.out.println("Level Order Traversal: " + solution.levelOrder(root));

System.out.println("Inorder Traversal: " + solution.inorderTraversal(root));

int[] inorder = {9, 3, 15, 20, 7};

int[] postorder = {9, 15, 7, 20, 3};

TreeNode constructedTree = solution.buildTree(inorder, postorder);

System.out.println("Constructed Tree Level Order: " + solution.levelOrder(constructedTree));

int k = 3;

System.out.println("Kth Smallest Element in BST: " + solution.kthSmallest(root, k));

}

}

1. **Populating Next Right Pointers in Each Node:**

public class Solution {

public Node connect(Node root) {

if (root == null) return null;

// Start with the root node

Node leftmost = root;

// Traverse level by level

while (leftmost.left != null) {

Node current = leftmost;

while (current != null) {

// Connect left child to right child

current.left.next = current.right;

// Connect right child to the next node's left child

if (current.next != null) {

current.right.next = current.next.left;

}

// Move to next node in the current level

current = current.next;

}

// Move to the next level

leftmost = leftmost.left;

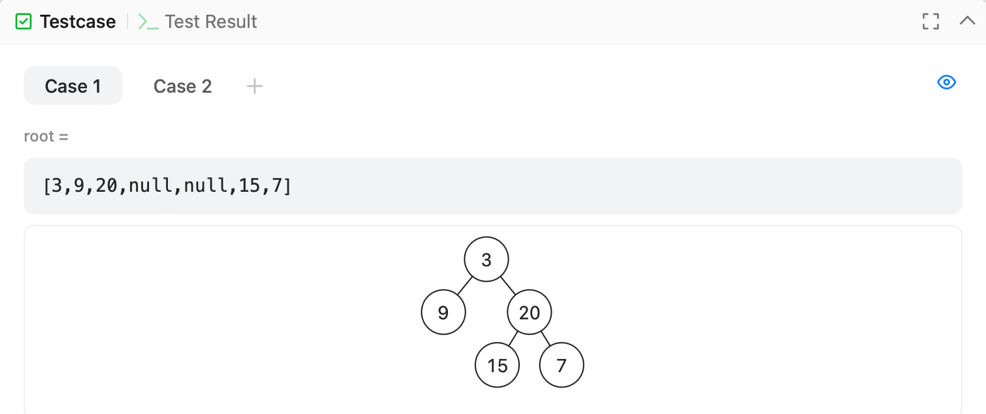
}

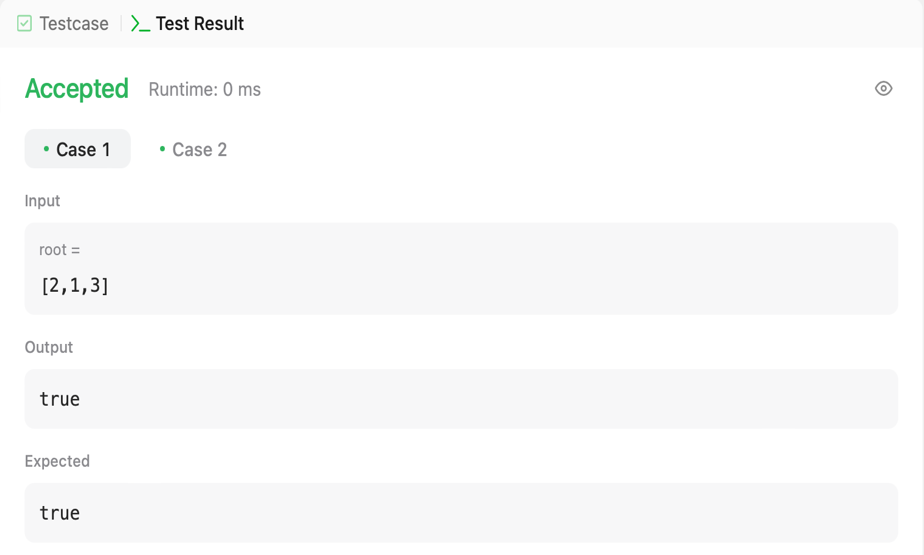
return root;

}

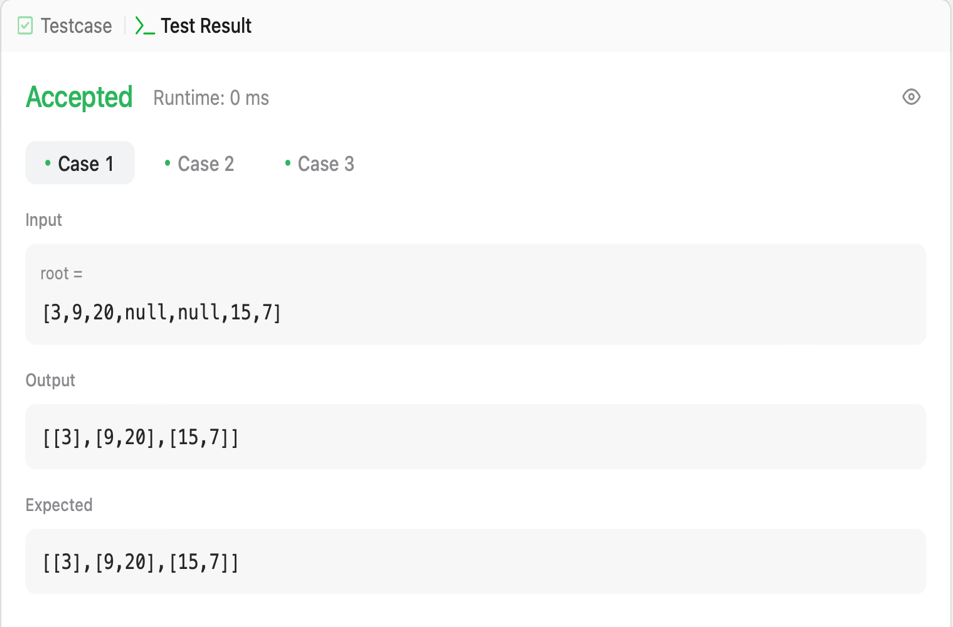
}

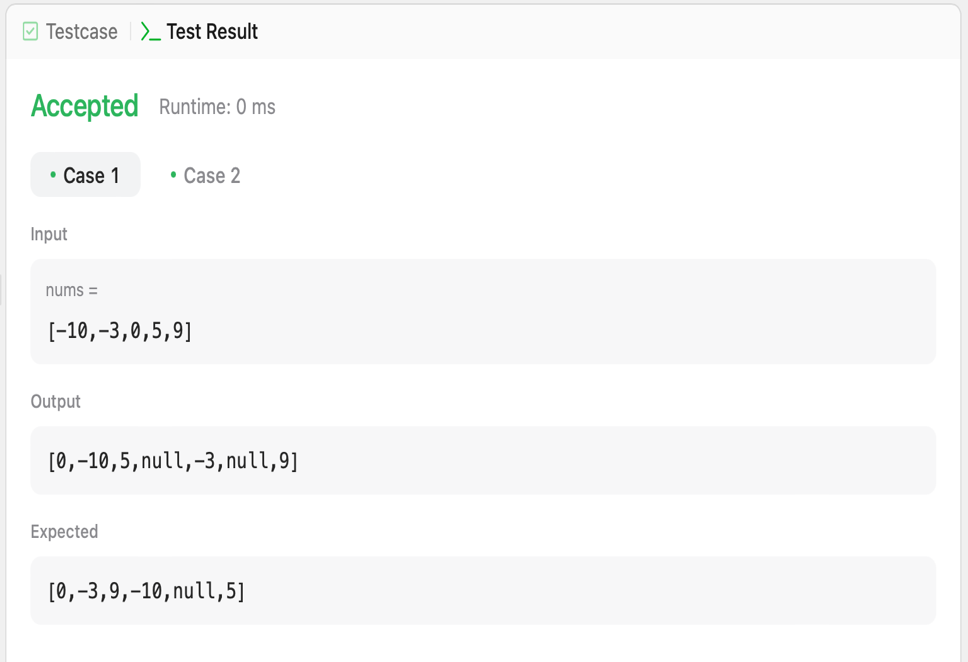
* **Output:**

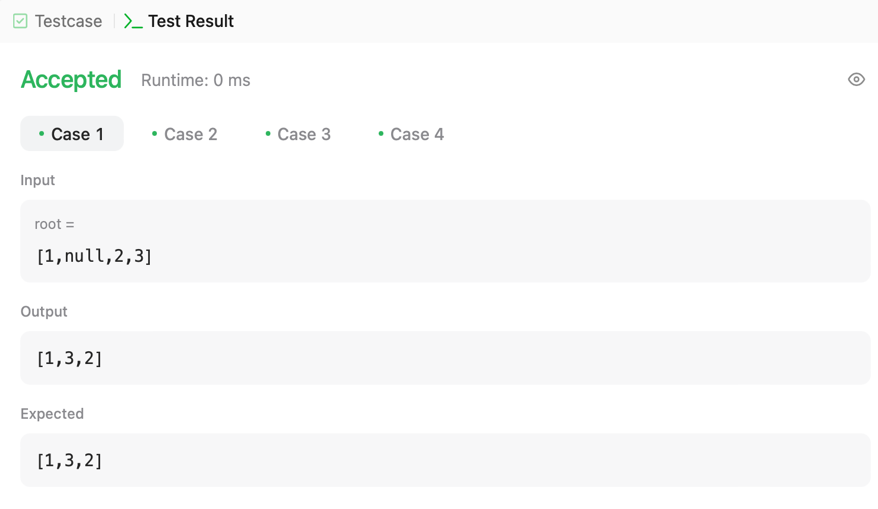
****

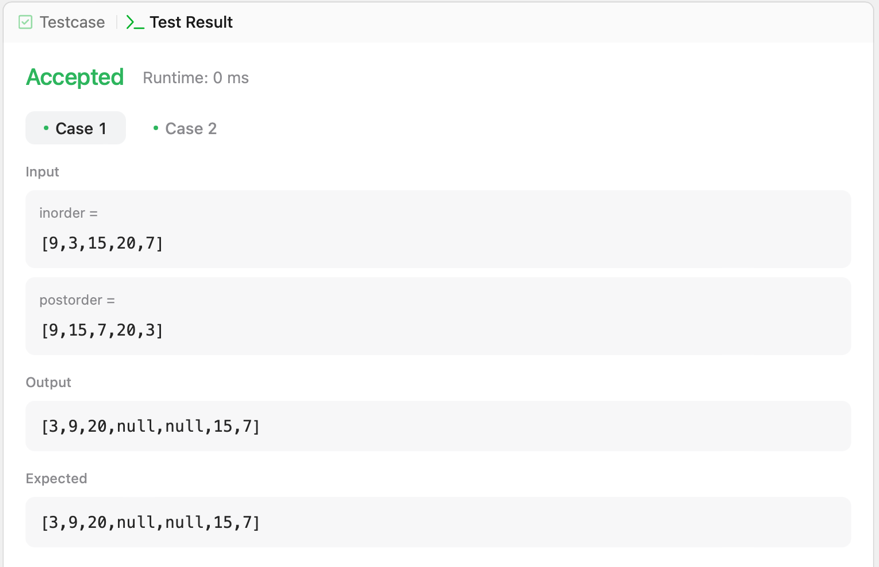
****

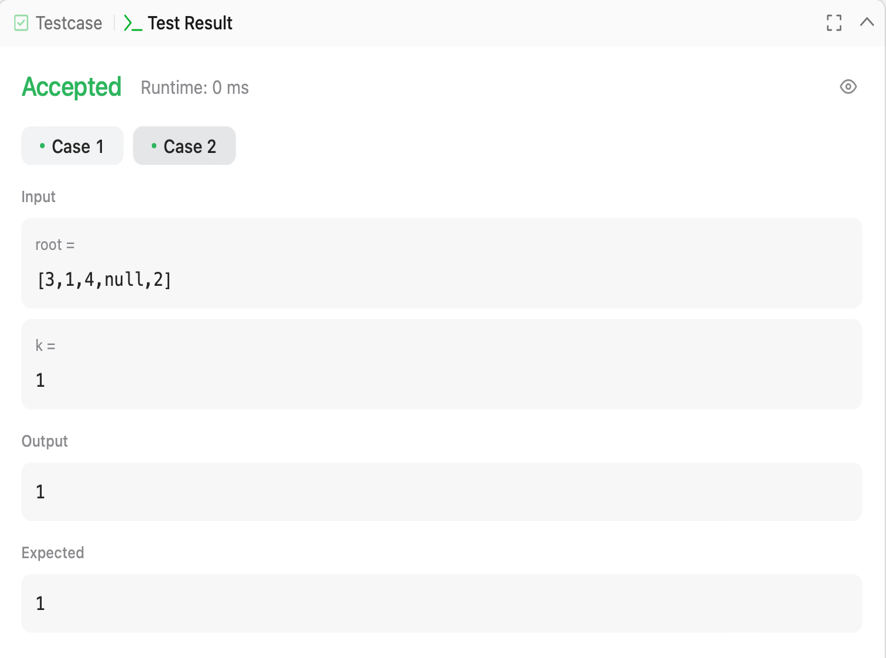
****

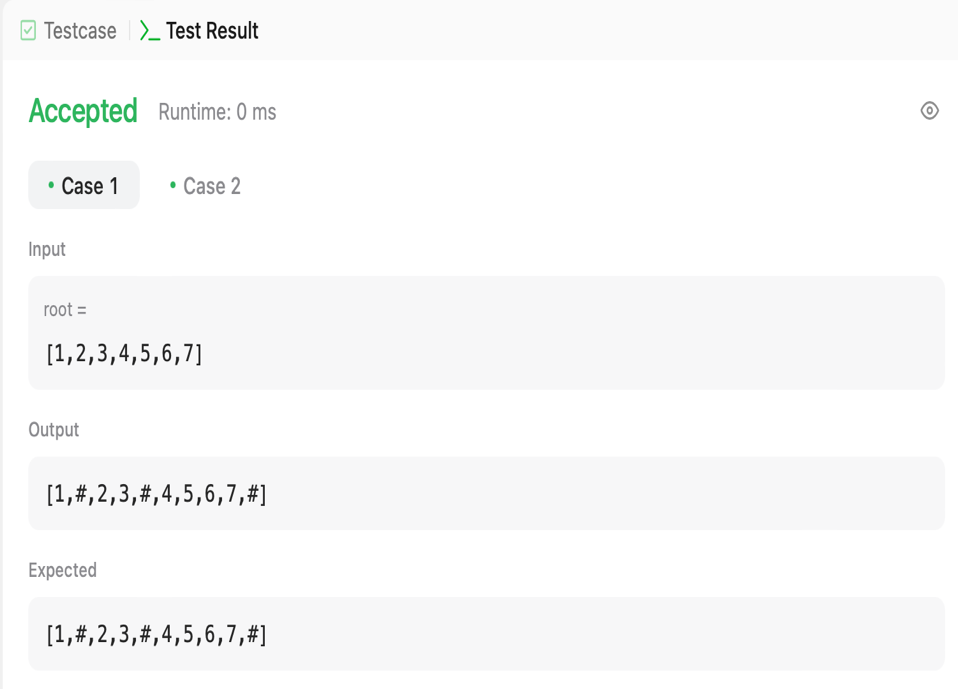
****

****

****

****

****

****