Experiment 6

Student Name: Shivansh Singh **UID: 22BET10105**
Branch: IT **Section/Group: 22BET_IOT-701/A**
Semester: 6th **Date of Performance:05.03.25**
Subject Name: AP Lab - 2 **Subject Code: 22ITP-351**

1. **Aim:** To develop a strong understanding of binary trees and BSTs, focusing on traversal, construction, validation, and optimization techniques for efficient problem-solving .
   - i.) Maximum Depth of Binary Tree
   - ii.) Validate Binary Search Tree
   - iii.) Symmetric Tree
   - iv.) Binary Tree Level Order Traversal
   - v.) Convert Sorted Array to Binary Search Tree
   - vi.) Binary Tree Inorder Traversal
   - vii.) Construct Binary Tree from Inorder and Postorder Traversal
   - viii.) Kth Smallest element in a BST
   - ix.) Populating Next Right Pointers in Each Node

2. **Objective:**

- Understand and implement tree traversal techniques, including inorder, preorder, postorder, and level-order traversal.
- Improve proficiency in recursive and iterative approaches for binary tree problems.
- Solve problems related to tree depth calculation, symmetry checking, and BST validation.
- Enhance knowledge of constructing binary trees from inorder and postorder traversals.
- Optimize algorithms for searching, inserting, and retrieving elements in a BST efficiently.
- Strengthen logical reasoning in handling tree-based operations like balancing and connectivity.
- Implement solutions with optimal time and space complexity using recursion and BFS/DFS techniques.
- Gain hands-on experience with problem-solving on LeetCode and other coding platforms.

3. **Code:**

<u>Problem 1: Maximum Depth of Binary Tree</u>

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (!root) return 0;
```

```
        return 1 + max(maxDepth(root->left), maxDepth(root->right));
    }
};
```

Problem 2: Validate Binary Search Tree

```
class Solution {
public:
    bool isValid(TreeNode* node, long minVal, long maxVal) {
        if (!node) return true;
        if (node->val <= minVal || node->val >= maxVal) return false;
        return isValid(node->left, minVal, node->val) &&
               isValid(node->right, node->val, maxVal);
    }
    bool isValidBST(TreeNode* root) {
        return isValid(root, LONG_MIN, LONG_MAX);
    }
};
```

Problem 3: Symmetric Tree

```
class Solution {
public:
    bool isMirror(TreeNode* t1, TreeNode* t2) {
        if (!t1 && !t2) return true;  // Both are NULL
        if (!t1 || !t2) return false; // Only one is NULL
        return (t1->val == t2->val) &&
               isMirror(t1->left, t2->right) &&
               isMirror(t1->right, t2->left);
    }

    bool isSymmetric(TreeNode* root) {
        if (!root) return true;
        return isMirror(root->left, root->right);
    }
};
```

Problem 4: Binary Tree Level Order Traversal

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
     vector<vector<int>> result;
        if (!root) return result;
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int levelSize = q.size();
            vector<int> level;
            for (int i = 0; i < levelSize; i++) {
                TreeNode* node = q.front();
                q.pop();
                level.push_back(node->val);
```

```
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
            result.push_back(level);
        }
        return result;
    }
};
```

Problem 5: Convert Sorted Array to Binary Search Tree

```
class Solution {
public:
    TreeNode* helper(vector<int>& nums, int left, int right) {
        if (left > right) return nullptr;  // Base case
        int mid = left + (right - left) / 2;  // Middle element
        TreeNode* root = new TreeNode(nums[mid]);

        // Recursively construct left and right subtrees
        root->left = helper(nums, left, mid - 1);
        root->right = helper(nums, mid + 1, right);
        return root;
    }
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return helper(nums, 0, nums.size() - 1);
    }
};
```

Problem 6: Binary Tree Inorder Traversal

```
class Solution {
public:
    void inorderHelper(TreeNode* root, vector<int>& result) {
        if (!root) return;
        inorderHelper(root->left, result);  // Left subtree
        result.push_back(root->val);        // Root
        inorderHelper(root->right, result); // Right subtree
    }
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        inorderHelper(root, result);
        return result;
    }
};
```

Problem 7: Construct Binary Tree from Inorder and Postorder Traversal

```
class Solution {
public:
unordered_map<int, int> inorderIndexMap; // Stores index of each value in inorder
    TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder,
                          int inStart, int inEnd, int& postIndex) {
```

```
        if (inStart > inEnd) return nullptr;
        // Last element in postorder is the root of the current subtree
        int rootVal = postorder[postIndex--];
        TreeNode* root = new TreeNode(rootVal);
        // Find root's index in inorder array
        int inIndex = inorderIndexMap[rootVal];
        // Recursively build right and left subtrees
        root->right = buildTreeHelper(inorder, postorder, inIndex + 1, inEnd, postIndex);
        root->left = buildTreeHelper(inorder, postorder, inStart, inIndex - 1, postIndex);
        return root;
    }
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        int n = inorder.size();
        int postIndex = n - 1; // Start from last element of postorder
        // Store inorder indices for quick lookup
        for (int i = 0; i < n; i++)
            inorderIndexMap[inorder[i]] = i;
        return buildTreeHelper(inorder, postorder, 0, n - 1, postIndex);
    }
};
```

Problem 8: Kth Smallest element in a BST

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        int count = 0, result = -1;
        inorder(root, k, count, result);
        return result;
    }
private:
    void inorder(TreeNode* node, int k, int& count, int& result) {
        if (!node) return;
        inorder(node->left, k, count, result);
        count++;
        if (count == k) {
            result = node->val;
            return;
        }
        inorder(node->right, k, count, result);
    }
};
```

Problem 9: Populating Next Right Pointers

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return nullptr;  // Edge case
        Node* start = root;  // Start from the root
        while (start->left) {  // Traverse each level
            Node* curr = start; // Traverse current level
```

```
            while (curr) {
                // Connect left child to right child
                curr->left->next = curr->right;
                // Connect right child to next node's left child (if exists)
                if (curr->next)
                    curr->right->next = curr->next->left;
                curr = curr->next; // Move to the next node in the level
            }
            start = start->left; // Move to next level
        }
        return root;
    }
};
```

4. Output:

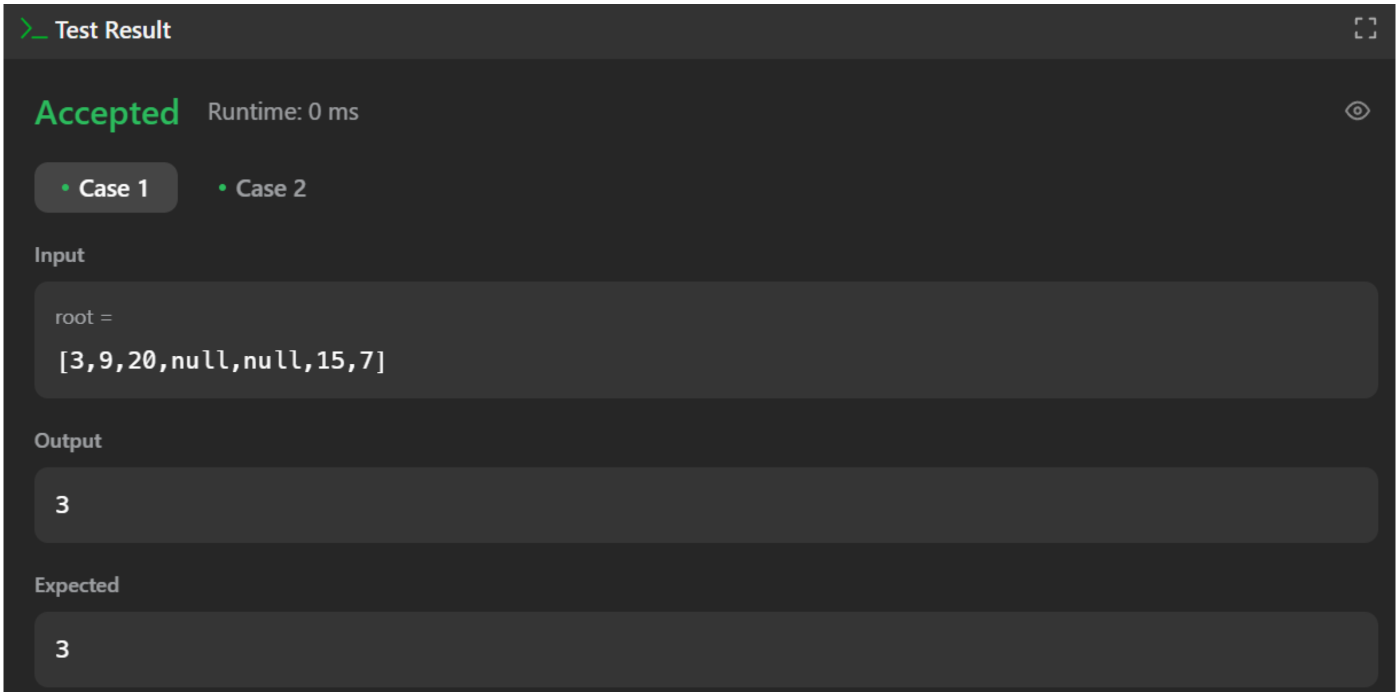

Fig 1. Maximum Depth of Binary Tree

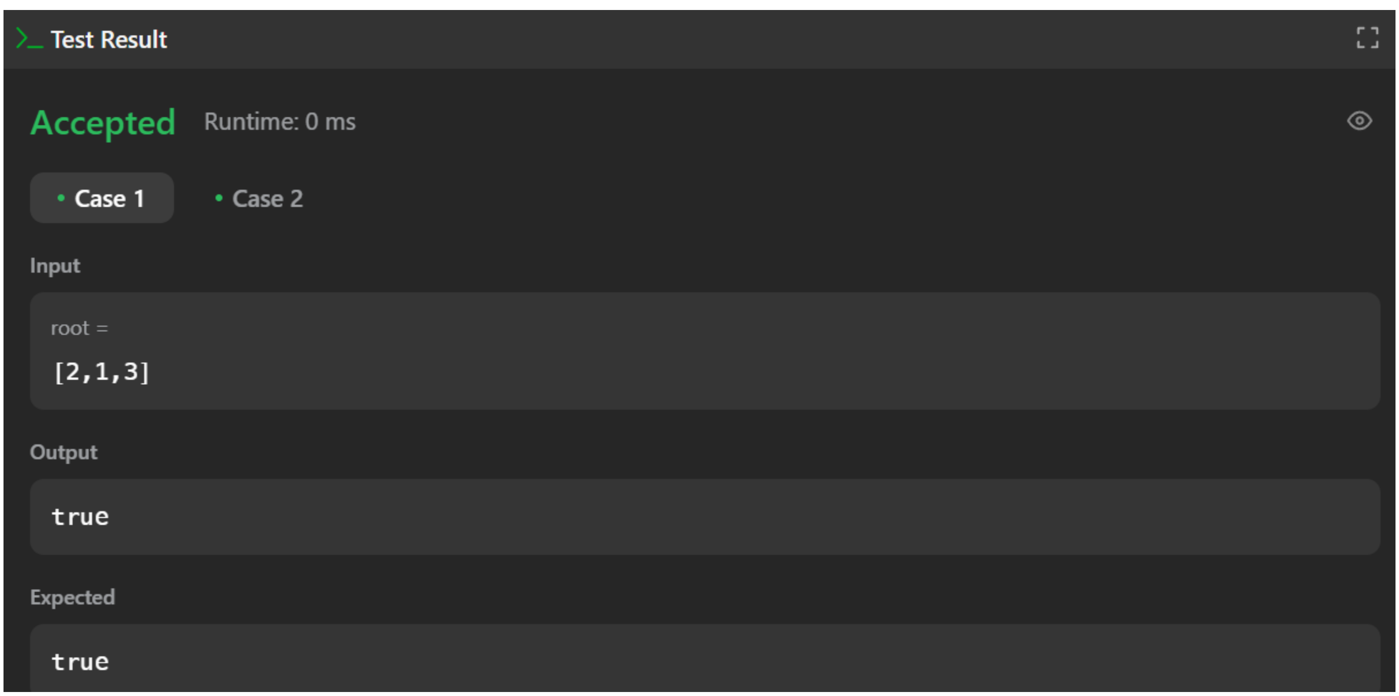

Fig 2. Validate Binary Search Tree

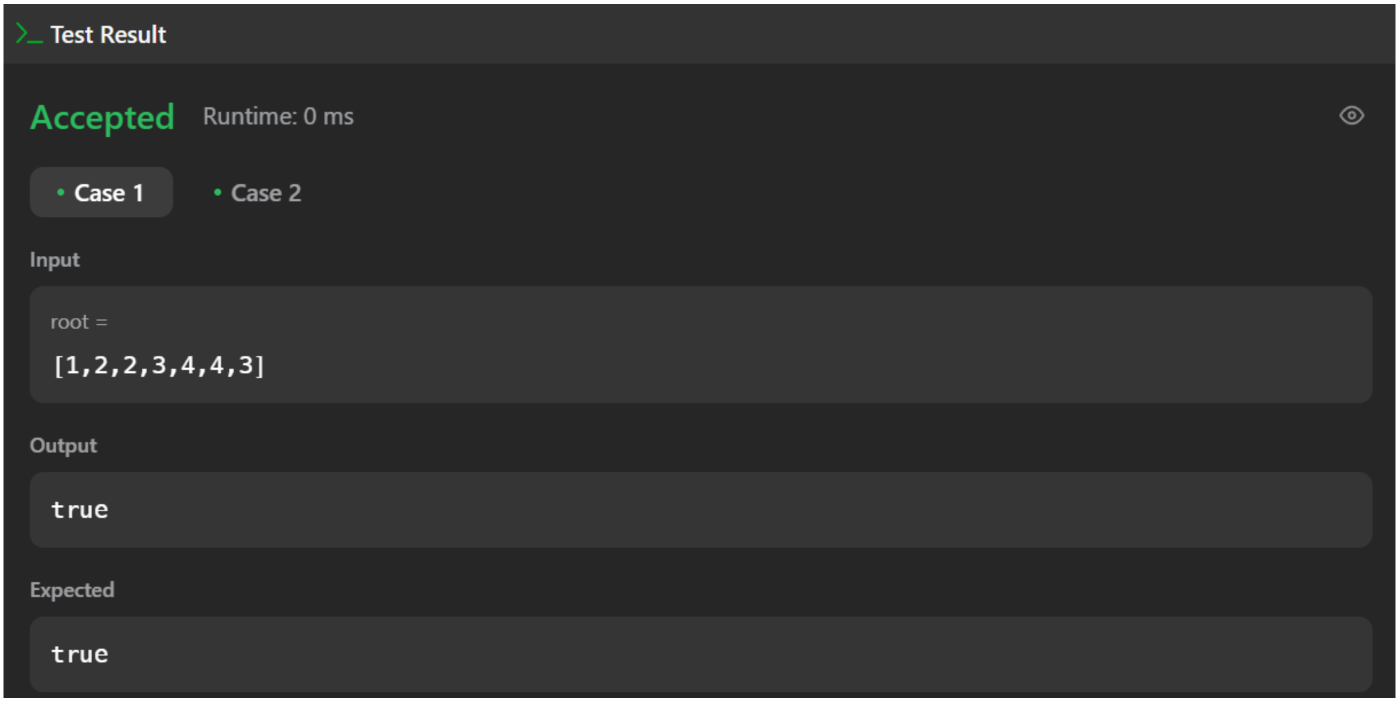

Fig 3. Symmetric Tree

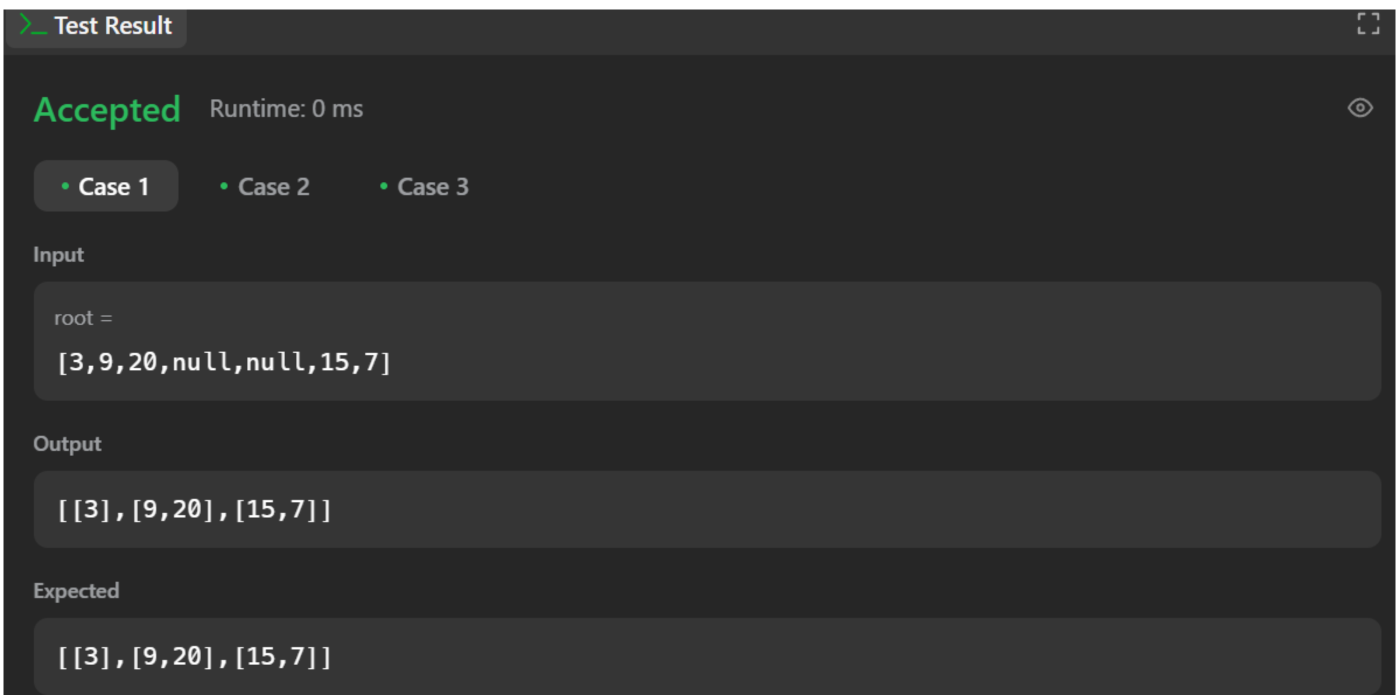

Fig 4. Binary Tree Level Order Traversal

Fig 5. Convert Sorted Array to Binary Search Tree

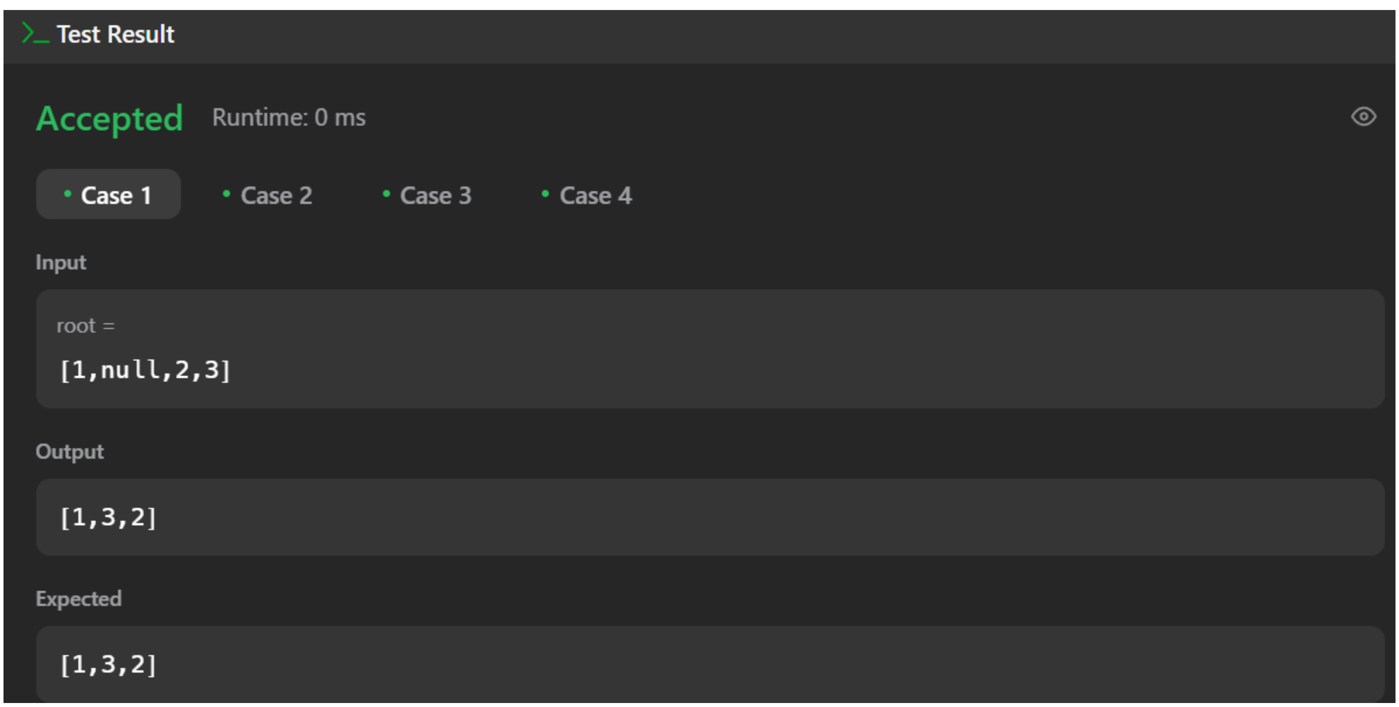

Fig 6. Binary Tree Inorder Traversal

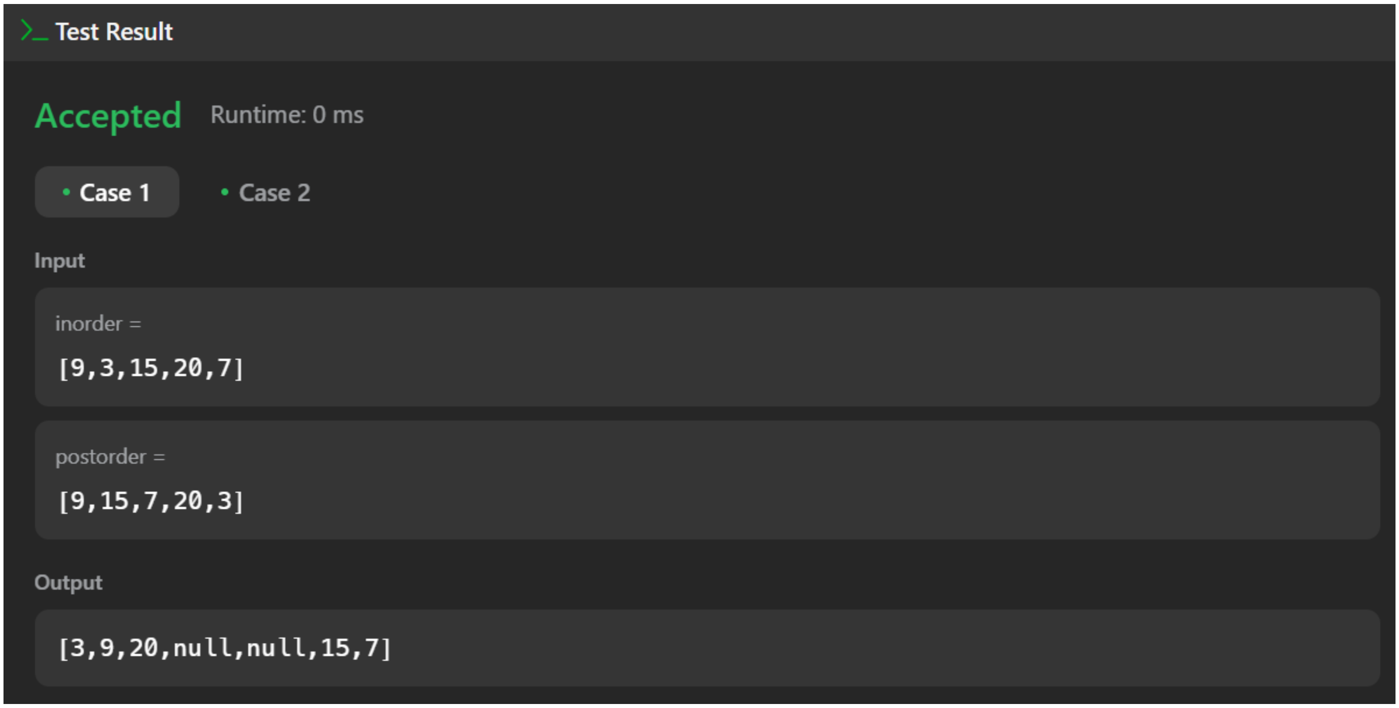

Fig 7. Construct Binary Tree from Inorder and Postorder Traversal

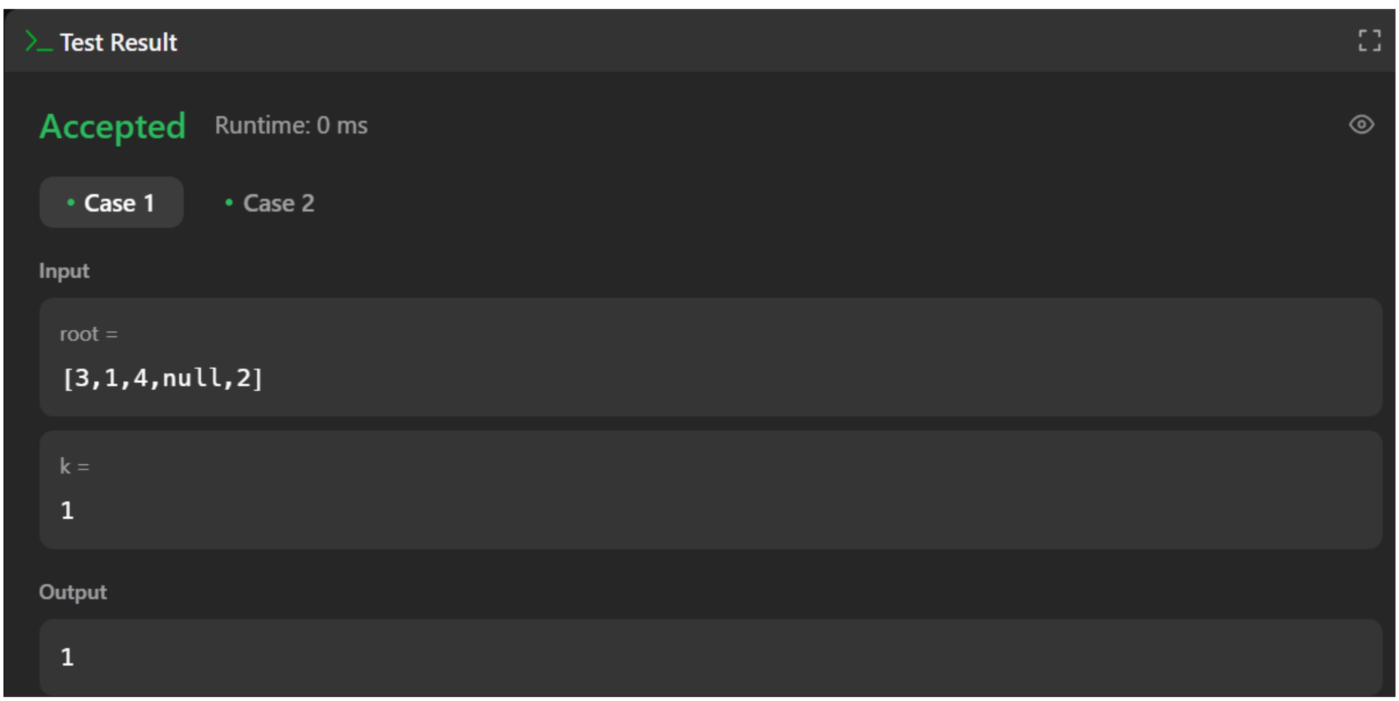

Fig 8. Kth Smallest element in a BST

Fig 9. Populating Next Right Pointers

5. Learning Outcomes:

- Develop a strong understanding of tree traversal techniques, including DFS (inorder, preorder, postorder) and BFS (level-order traversal).
- Gain proficiency in solving binary tree and BST-related problems using recursive and iterative approaches.
- Learn to construct binary trees from given traversal sequences (inorder & postorder) and convert sorted arrays to height-balanced BSTs.
- Master techniques for validating BST properties, finding the kth smallest element, and checking tree symmetry.
- Improve problem-solving skills in binary tree depth calculation, level order traversal, and next pointer population.
- Optimize code for time and space efficiency using divide-and-conquer, recursion, and iterative methods.
- Strengthen debugging and logical reasoning skills by solving complex tree-based problems.
- Gain hands-on experience with LeetCode-style problems to prepare for technical interviews and coding challenges.