Experiment 6

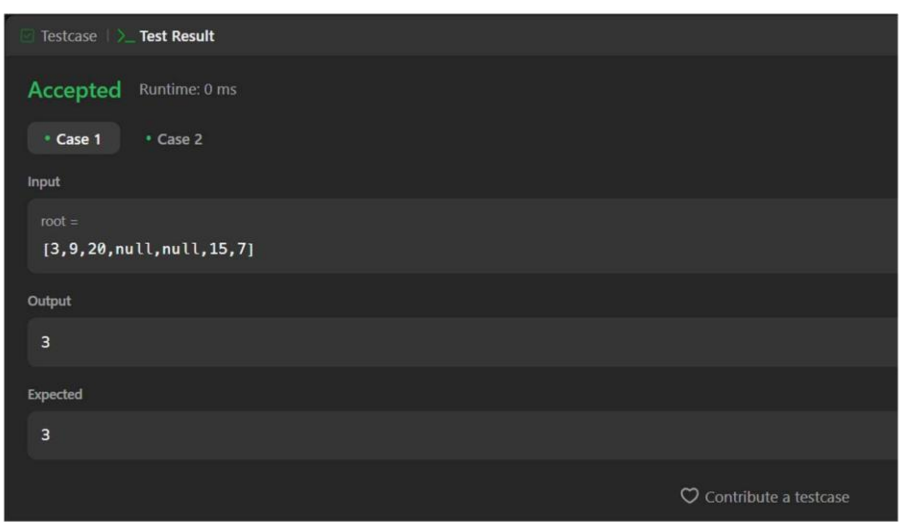
Student Name: KHUSHBOO **UID: 22BET10197**
Branch: BE-IT **Section/Group: 22BET_IOT_701/A**
Semester: 6th **Date of Performance: 28-2-25**
Subject Name: AP-II **Subject Code: 22ITP-351**

Problem 1. Maximum Depth of Binary Tree

- **Algorithm:**
    1. Base case: If the root is nullptr, return 0.
    2. Recursively find the maximum depth of the left subtree.
    3. Recursively find the maximum depth of the right subtree.
    4. Return the maximum of the left and right depths, plus 1 (for the current node).
- **Code:**

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        return max(leftDepth, rightDepth) + 1;
    }
};
```

- **Output:**



Problem 2. Validate Binary Search Tree

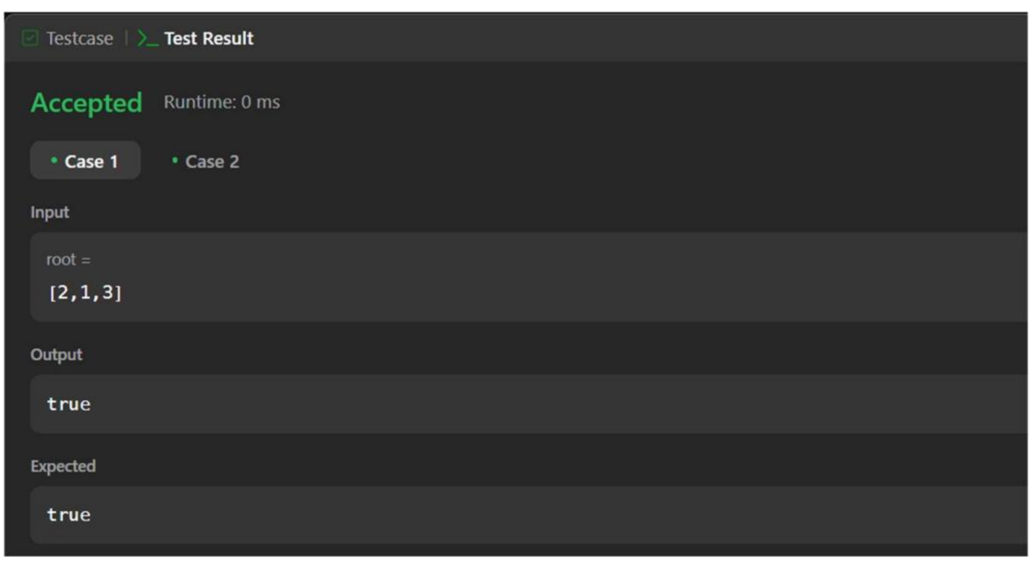
- **Algorithm:**
  1. Use a helper function with a range (min, max).
  2. Base case: If the root is nullptr, it's a valid BST.
  3. Check if the current node's value is within the range (min, max).
  4. Recursively validate the left subtree with the range (min, root->val).
  5. Recursively validate the right subtree with the range (root->val, max).

- **Code:**

```
class Solution {
  public:
    bool isValidBST(TreeNode* root) {
      return isValidBSTHelper(root, LONG_MIN, LONG_MAX);
    }

    bool isValidBSTHelper(TreeNode* root, long minVal, long maxVal) {
      if (root == nullptr) {
        return true;
      }
      if (root->val <= minVal || root->val >= maxVal) {
        return false;
      }
      return isValidBSTHelper(root->left, minVal, root->val) &&
          isValidBSTHelper(root->right, root->val, maxVal);
    }
  };
```

- **Output:**



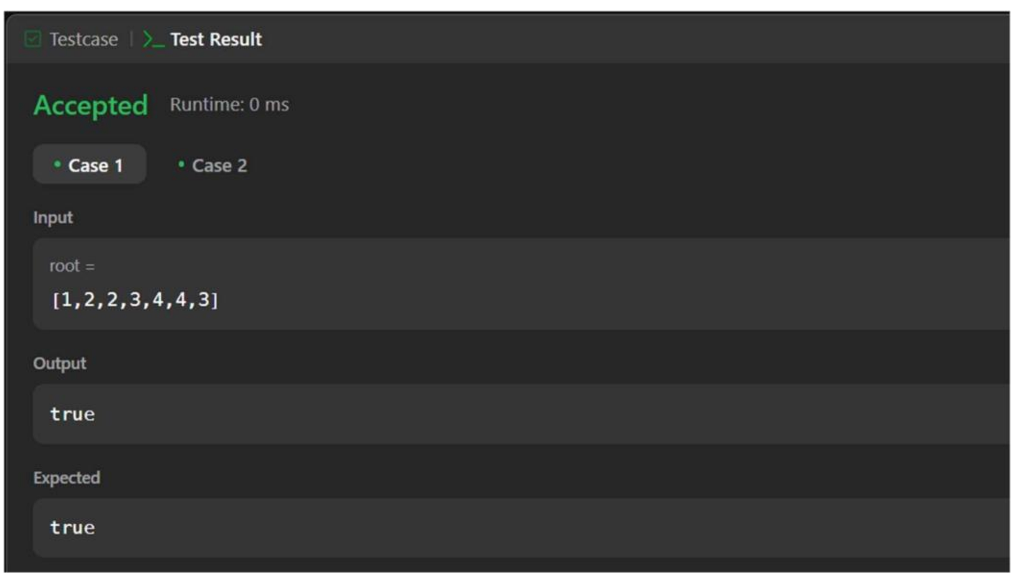
Problem 3. Symmetric Tree

- **Algorithm:**
  1. Create a helper function to check if two trees are mirrors of each other.
  2. Base case: If both trees are nullptr, they are symmetric.
  3. Base case: If one is nullptr and the other is not, they are not symmetric.
  4. Check if the values are equal.
  5. Recursively check if the left of the first tree is the mirror of the right of the second tree, and vice versa.
- **Code:**

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (root == nullptr) {
            return true;
        }
        return isMirror(root->left, root->right);
    }

    bool isMirror(TreeNode* left, TreeNode* right) {
        if (left == nullptr && right == nullptr) {
            return true;
        }
        if (left == nullptr || right == nullptr) {
            return false;
        }
        return (left->val == right->val) && isMirror(left->left, right->right) &&
               isMirror(left->right, right->left);
    }
};
```

- **Output:**

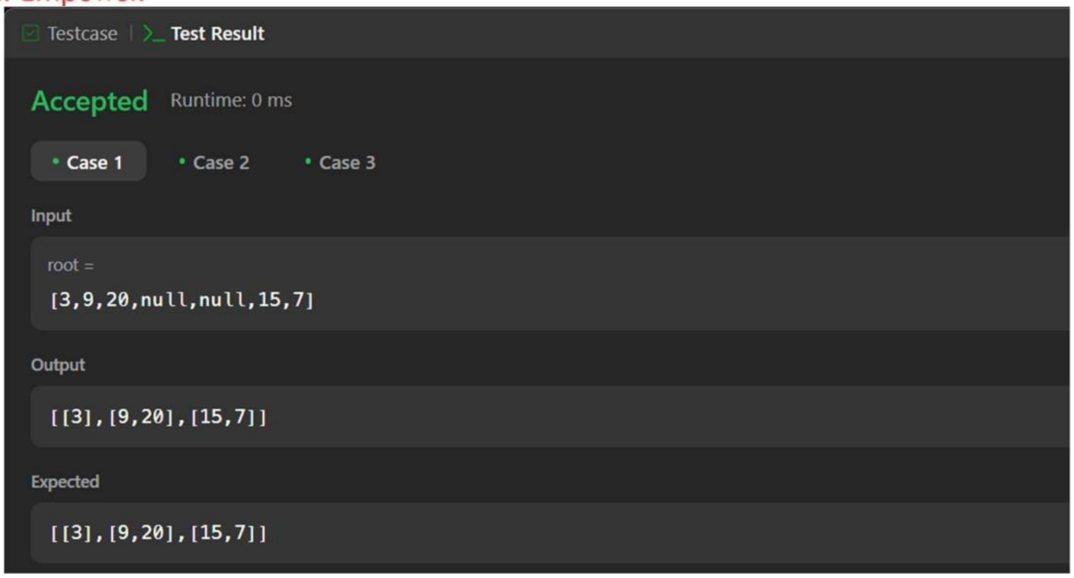


Problem 4. Binary Tree Level Order Traversal

- **Algorithm:**
    1. Use a queue for level-order traversal.
    2. Enqueue the root node.
    3. While the queue is not empty:
        - Get the number of nodes at the current level.
        - For each node at the current level:
            - Dequeue the node.
            - Add its value to the current level's list.
            - Enqueue its left and right children (if they exist).
        - Add the current level's list to the result.
- **Code:**

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

- **Output:**



Problem 5. Convert Sorted Array to Binary Search Tree

- **Algorithm:**
  1. Create a helper function that takes the array and start/end indices.
  2. Base case: If start > end, return nullptr.
  3. Find the middle index.
  4. Create a new node with the middle element.
  5. Recursively create the left subtree with the left half of the array.
  6. Recursively create the right subtree with the right half of the array.
  7. Return the new node.
- **Code:**

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return sortedArrayToBSTHelper(nums, 0, nums.size() - 1);
    }

    TreeNode* sortedArrayToBSTHelper(vector<int>& nums, int start, int end) {
        if (start > end) {
            return nullptr;
        }
        int mid = start + (end - start) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = sortedArrayToBSTHelper(nums, start, mid - 1);
        root->right = sortedArrayToBSTHelper(nums, mid + 1, end);
        return root;
    }
};
```

- **Output:**

Problem 6. Binary Tree Inorder Traversal

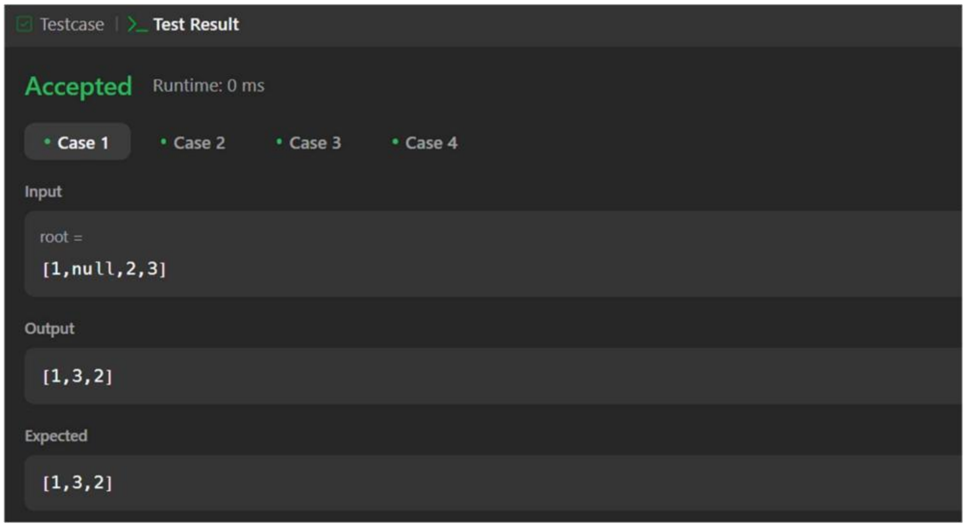
- **Algorithm:**
  1. Use a recursive helper function
  2. If root is null return.
  3. Call the function on the left child
  4. add the current root value to the answer vector.
  5. Call the function on the right child.

- **Code:**

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        inorderTraversalHelper(root, result);
        return result;
    }

    void inorderTraversalHelper(TreeNode* root, vector<int>& result) {
        if (root == nullptr) {
            return;
        }
        inorderTraversalHelper(root->left, result);
        result.push_back(root->val);
        inorderTraversalHelper(root->right, result);
    }
};
```

- **Output:**



Problem 7. Construct Binary Tree from Inorder and Postorder Traversal

- **Algorithm:**
  1. The last element in the postorder array is the root of the tree.
  2. Find the root's index in the inorder array.
  3. Recursively construct the left subtree using the left part of the inorder array and the corresponding left part of the postorder array.
  4. Recursively construct the right subtree using the right part of the inorder array and the corresponding right part of the postorder array.
  5. Return the root.

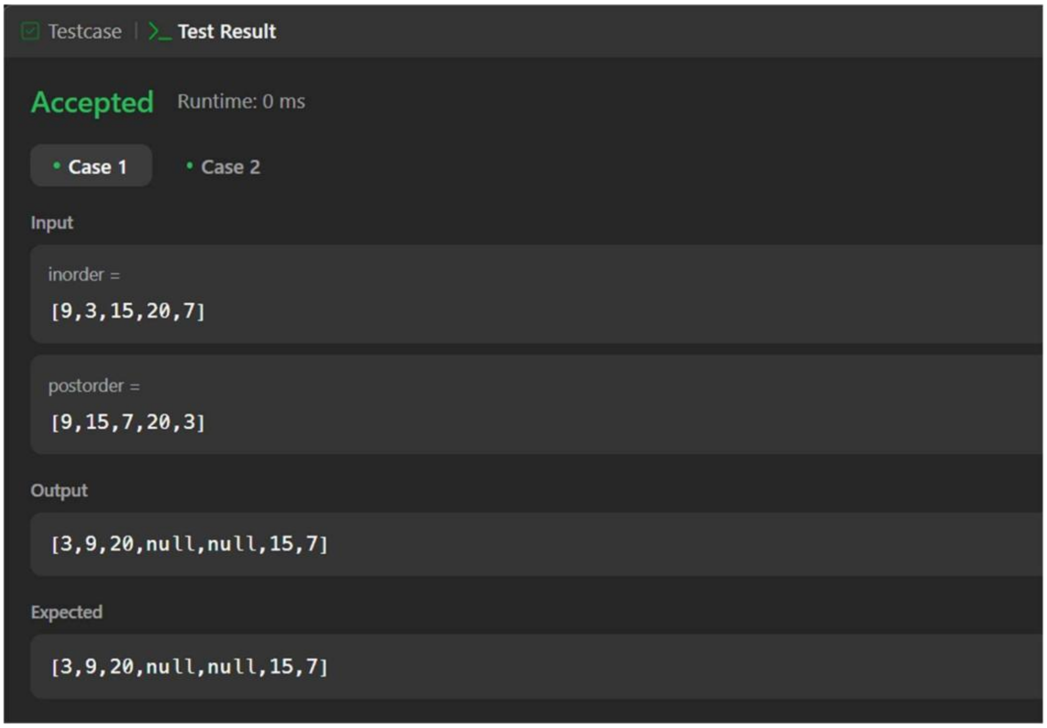
- **Code:**

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        int postIndex = postorder.size() - 1;
        unordered_map<int, int> inMap;
        for (int i = 0; i < inorder.size(); ++i) {
            inMap[inorder[i]] = i;
        }
        return buildTreeHelper(inorder, postorder, 0, inorder.size() - 1, postIndex, inMap);
    }

    TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int inStart, int inEnd,
int& postIndex, unordered_map<int, int>& inMap) {
        if (inStart > inEnd) {
            return nullptr;
        }
        TreeNode* root = new TreeNode(postorder[postIndex]);
        postIndex--;
        int inRoot = inMap[root->val];
        root->right = buildTreeHelper(inorder, postorder, inRoot + 1, inEnd, postIndex, inMap);
```

```
        root->left = buildTreeHelper(inorder, postorder, inStart, inRoot - 1, postIndex, inMap);
        return root;
    }
};
```

- **Output:**



Problem 8. Kth Smallest Element in a BST

- **Algorithm:**
    1. Perform an inorder traversal of the BST.
    2. Keep a counter to track the number of nodes visited.
    3. When the counter reaches k, return the current node's value.

- **Code:**

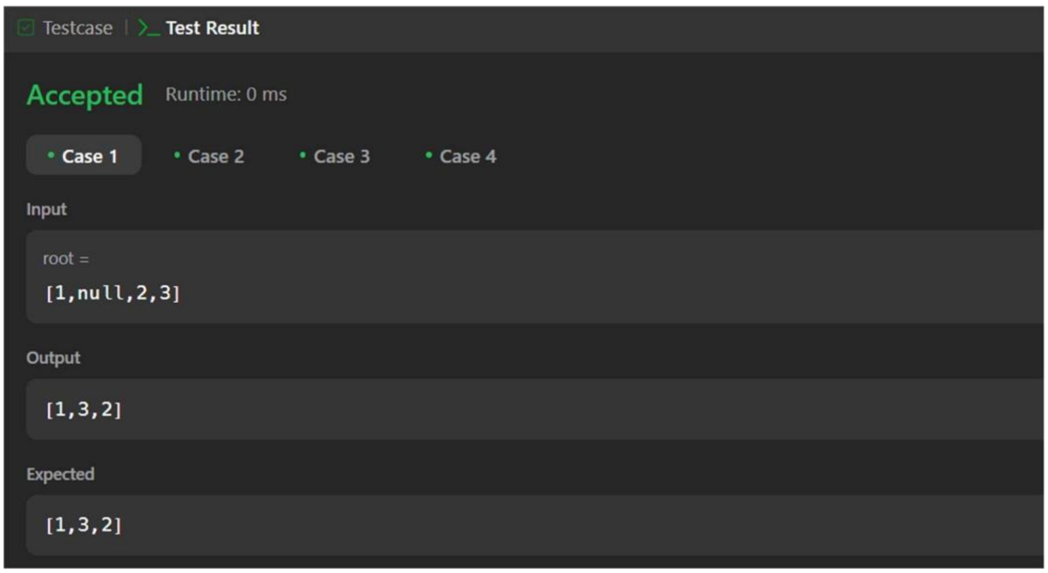
```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        int count = 0;
        int result = 0;
        kthSmallestHelper(root, k, count, result);
        return result;
    }

    void kthSmallestHelper(TreeNode* root, int k, int& count, int& result) {
        if (root == nullptr) {
            return;
        }
```

```
        kthSmallestHelper(root->left, k, count, result);
        count++;
        if (count == k) {
            result = root->val;
            return;
        }

        kthSmallestHelper(root->right, k, count, result);
    }
};
```

- **Output:**



Problem 9. Populating Next Right Pointers in Each Node

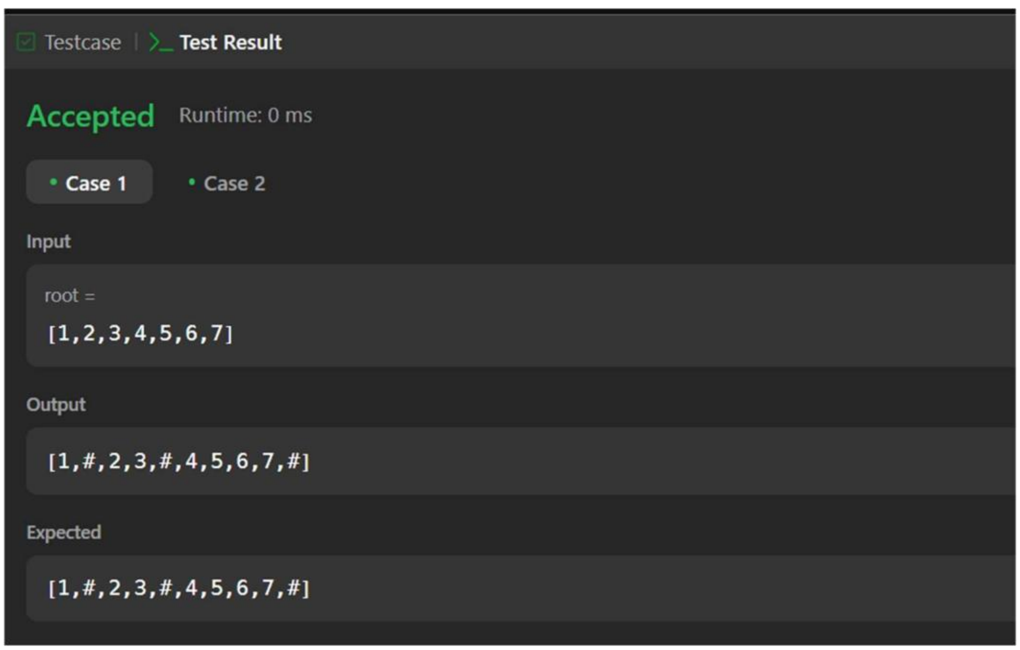
- **Algorithm:**
  1. Use a level-order traversal with a queue.
  2. For each level, connect the nodes' next pointers to the next node in the queue.
  3. If it's the last node in the level, set its next pointer to nullptr.

- **Code:**

```
class Solution {
public:
    Node* connect(Node* root) {
        if (root == nullptr) {
            return nullptr;
        }
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            int levelSize = q.size();
            for (int i = 0; i < levelSize; ++i) {
                Node* node = q.front();
                q.pop();
```

```
                if (i < levelSize - 1) {
                    node->next = q.front();
                } else {
                    node->next = nullptr;
                }
                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }
        }
        return root;
    }
};
```

- **Output:**



Problem 10. Binary Tree Inorder Traversal

- **Algorithm:**
    1. Use a recursive helper function
    2. If root is null return.
    3. Call the function on the left child
    4. add the current root value to the answer vector.
    5. Call the function on the right child.

- **Code:**

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        inorderTraversalHelper(root, result);
        return result;
    }

    void inorderTraversalHelper(TreeNode* root, vector<int>& result) {
        if (root == nullptr) {
            return;
        }
        inorderTraversalHelper(root->left, result);
        result.push_back(root->val);
        inorderTraversalHelper(root->right, result);
    }
};
```

- **Output:**