Experiment 6

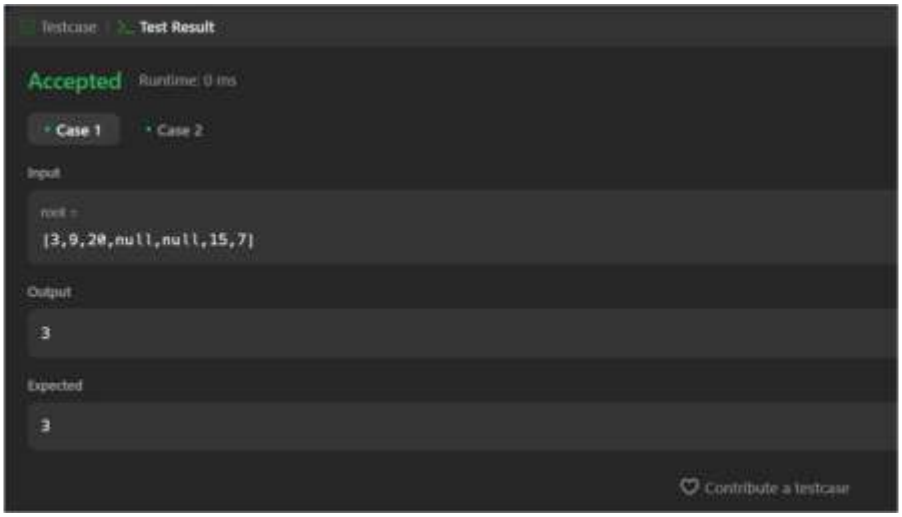
Student Name: SHREYA **UID: 22BET10172**
Branch: BE-IT **Section/Group: 22BET_IOT_702/A**
Semester: 6th **Date of Performance: 07-2-25**
Subject Name: Advanced Programming Lab-2 **Subject Code: 22ITP-351**

Problem 1. Maximum Depth of Binary Tree

- **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:**



Accepted Runtime: 0 ms

Input

```
root =
[3,9,20,null,null,15,7]
```

Output

```
3
```

Expected

```
3
```

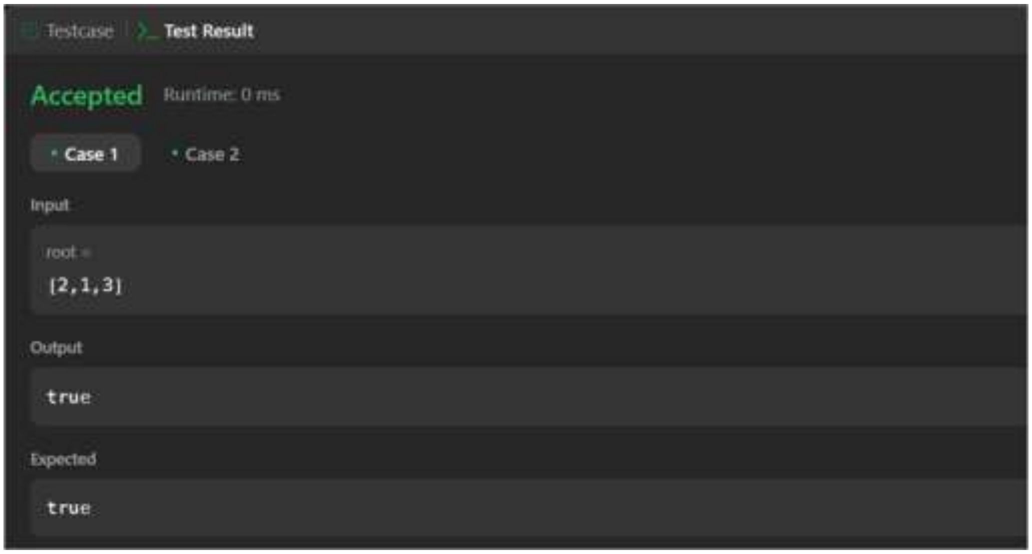
Problem 2. Validate Binary Search Tree

- **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:**



```
Accepted   Runtime: 0 ms

Input
root =
[2,1,3]

Output
true

Expected
true
```

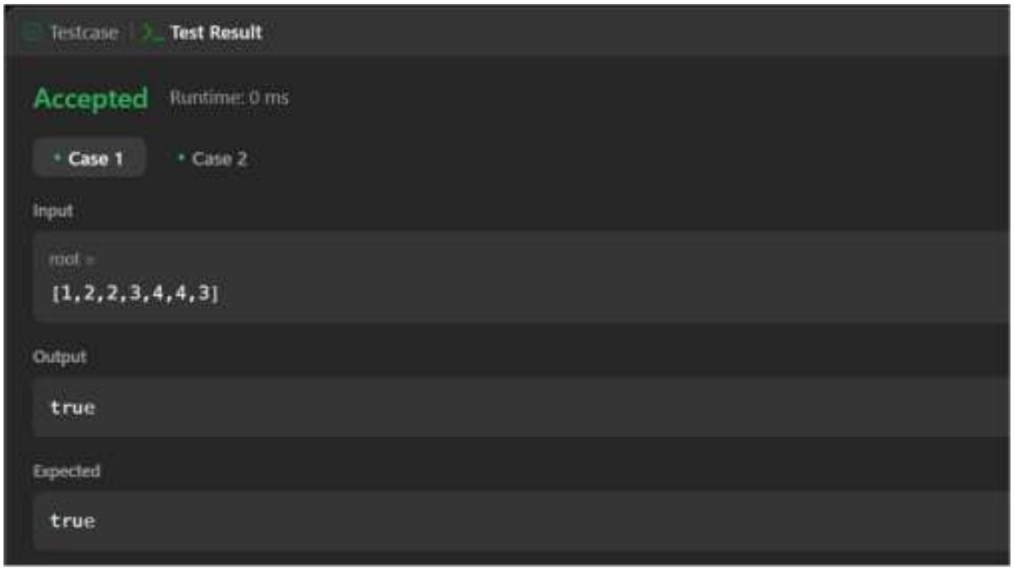
Problem 3. Symmetric Tree

- **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:**



Accepted Runtime: 0 ms

Input

```
root =
[1,2,2,3,4,4,3]
```

Output

```
true
```

Expected

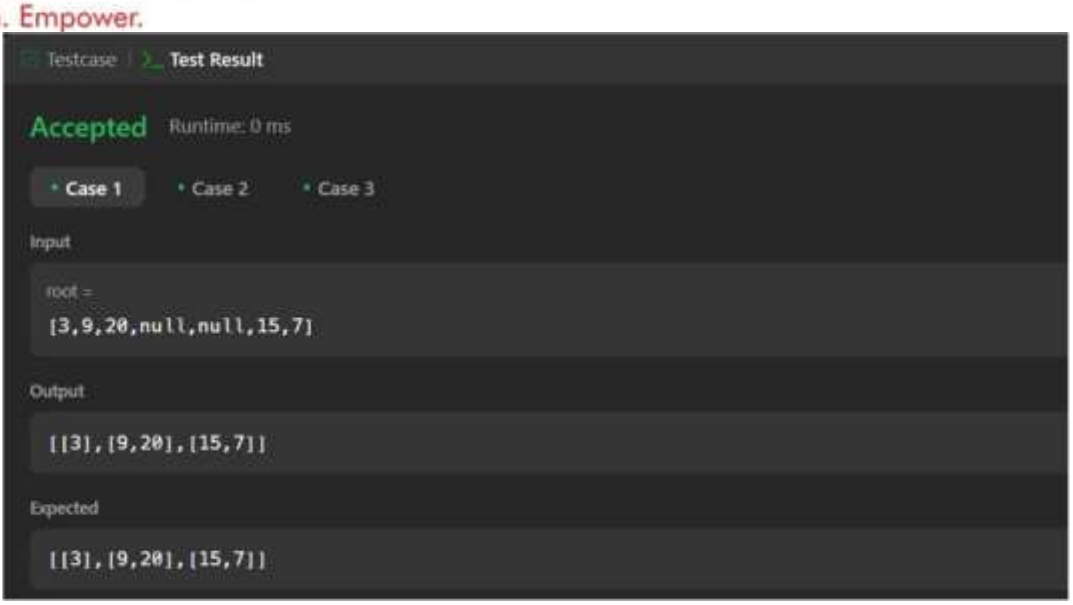
```
true
```

Problem 4. Binary Tree Level Order Traversal

- **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:**



Accepted Runtime: 0 ms

Input

```
root =
[3,9,20,null,null,15,7]
```

Output

```
[[3],[9,20],[15,7]]
```

Expected

```
[[3],[9,20],[15,7]]
```

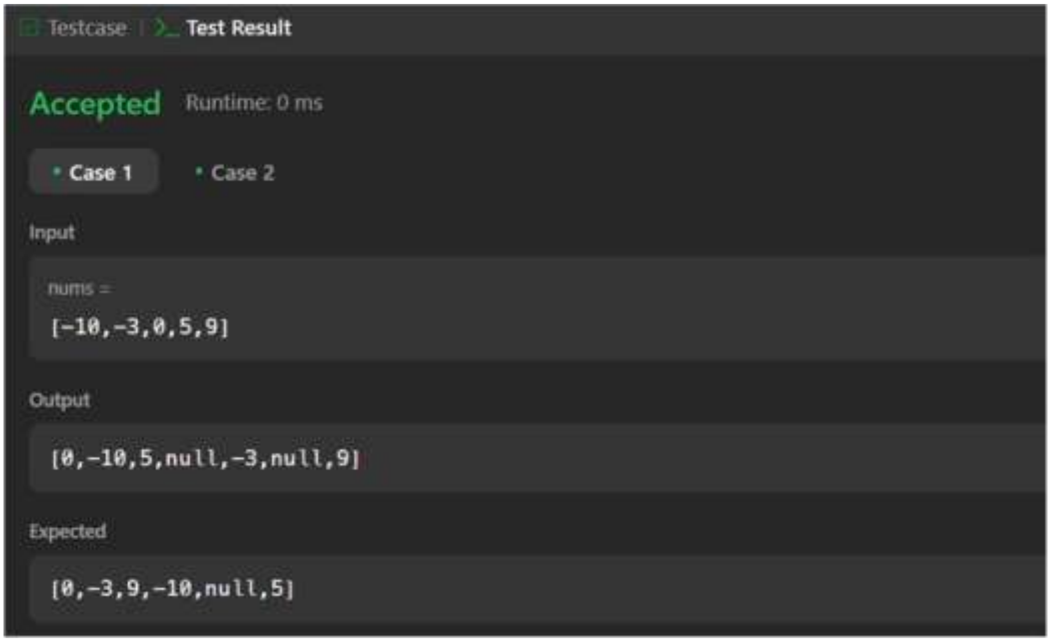
Problem 5. Convert Sorted Array to Binary Search Tree

- **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:**



Accepted Runtime: 0 ms

Input

nums =

```
[-10,-3,0,5,9]
```

Output

```
[0,-10,5,null,-3,null,9]
```

Expected

```
[0,-3,9,-10,null,5]
```

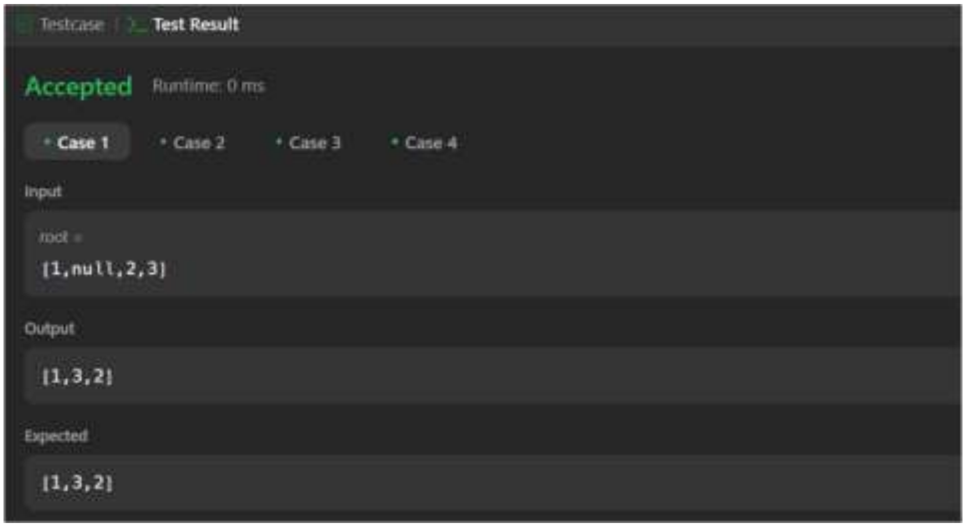
Problem 6. Binary Tree Inorder Traversal

- **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:**



Accepted Runtime: 0 ms

Input

root =

```
[1,null,2,3]
```

Output

```
[1,3,2]
```

Expected

```
[1,3,2]
```

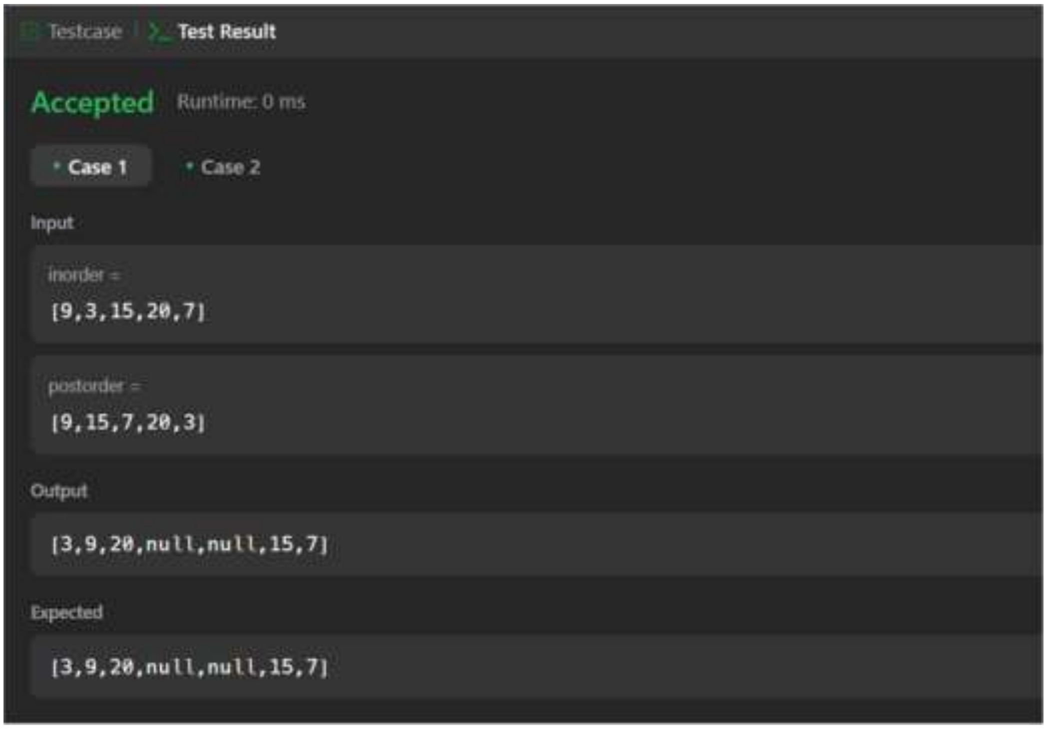
Problem 7. Construct Binary Tree from Inorder and Postorder Traversal

- **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:**



Accepted Runtime: 0 ms

Input

inorder =

```
[9,3,15,20,7]
```

postorder =

```
[9,15,7,20,3]
```

Output

```
[3,9,20,null,null,15,7]
```

Expected

```
[3,9,20,null,null,15,7]
```

Problem 8. Kth Smallest Element in a BST

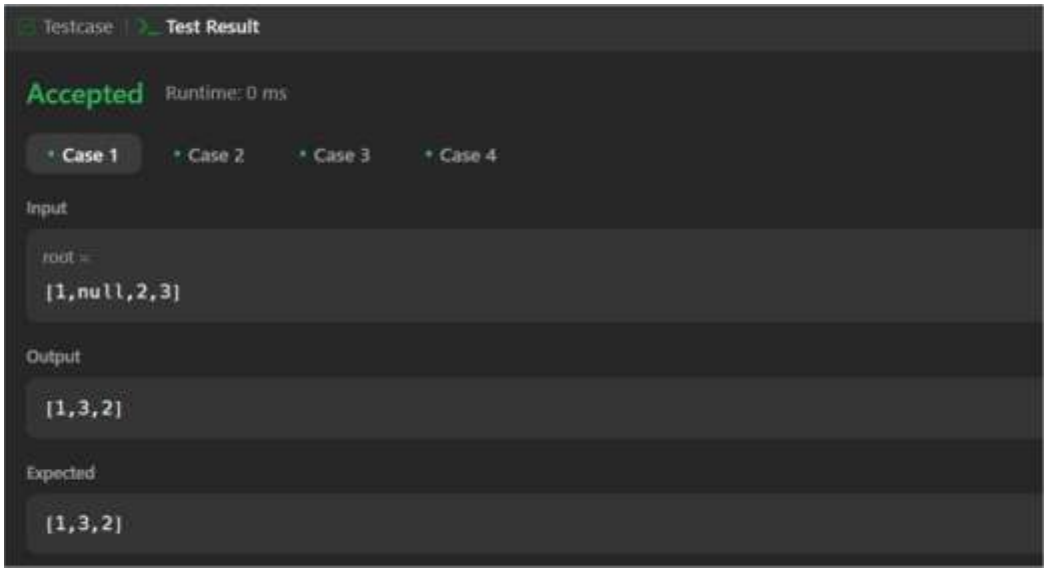
- **Code:**

```
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:**



Accepted Runtime: 0 ms

Input

root =

```
[1,null,2,3]
```

Output

```
[1,3,2]
```

Expected

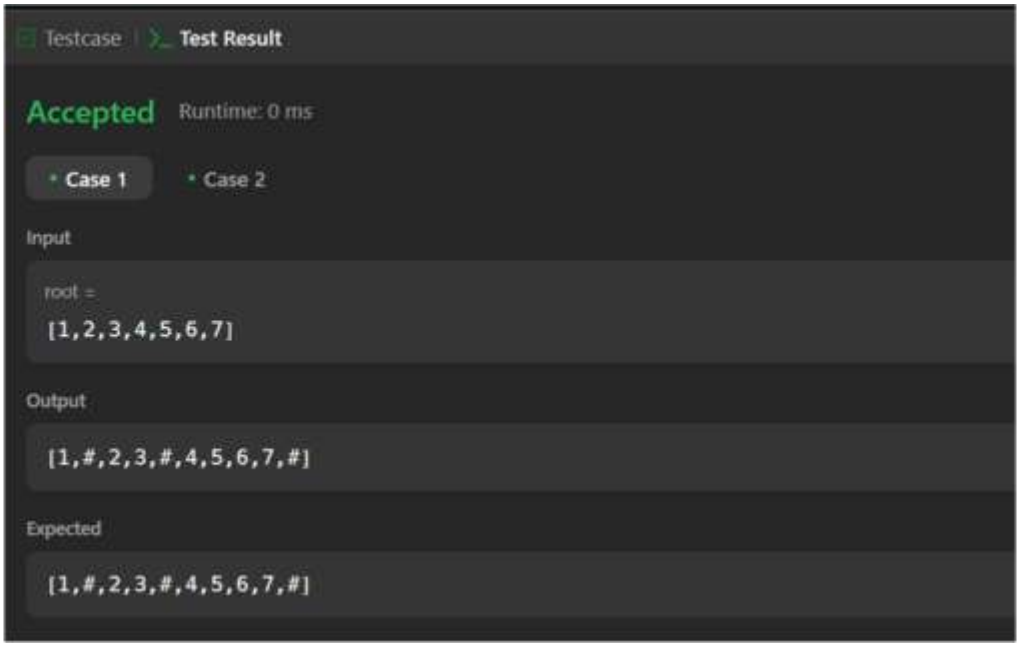
```
[1,3,2]
```

Problem 9. Populating Next Right Pointers in Each Node

- **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:**



Accepted Runtime: 0 ms

Input

root =

```
[1,2,3,4,5,6,7]
```

Output

```
[1,#,2,3,#,4,5,6,7,#]
```

Expected

```
[1,#,2,3,#,4,5,6,7,#]
```

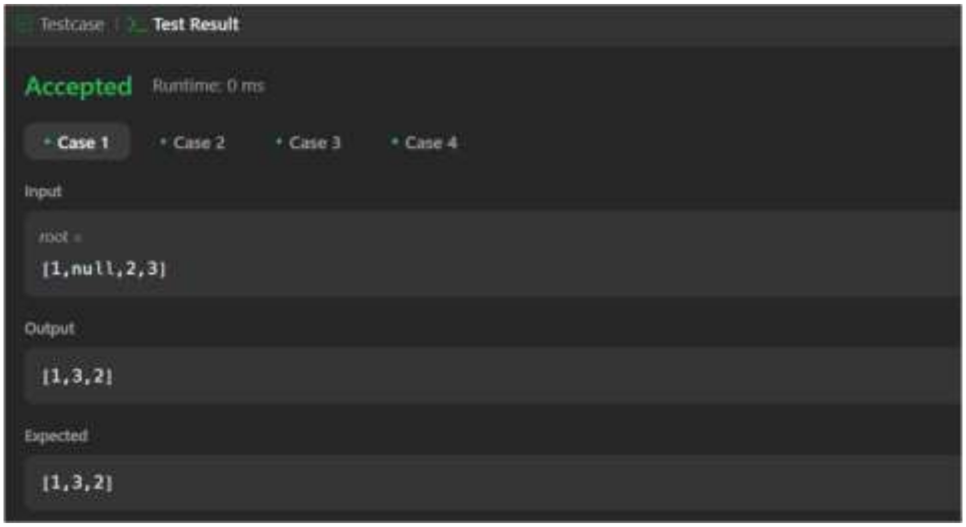
Problem 10. Binary Tree Inorder Traversal

- **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:**



Accepted Runtime: 0 ms

Input

root =

```
[1,null,2,3]
```

Output

```
[1,3,2]
```

Expected

```
[1,3,2]
```