

Experiment 7

Advanced Programming Lab – II

Submitted By – Ratnesh Tiwari

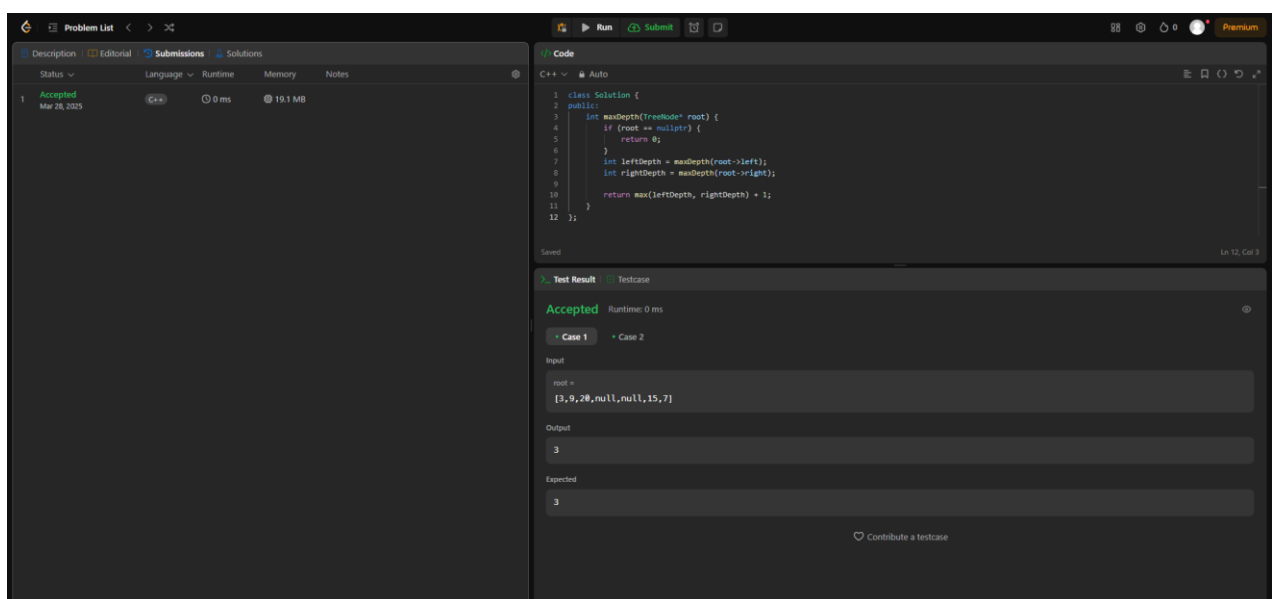
UID – 22BCS17201

Section – 22BCS_IOT – 609 (B)

1. Maximum Depth of Binary Tree: <https://leetcode.com/problems/maximum-depth-of-binary-tree/>

```
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;
    }
};
```

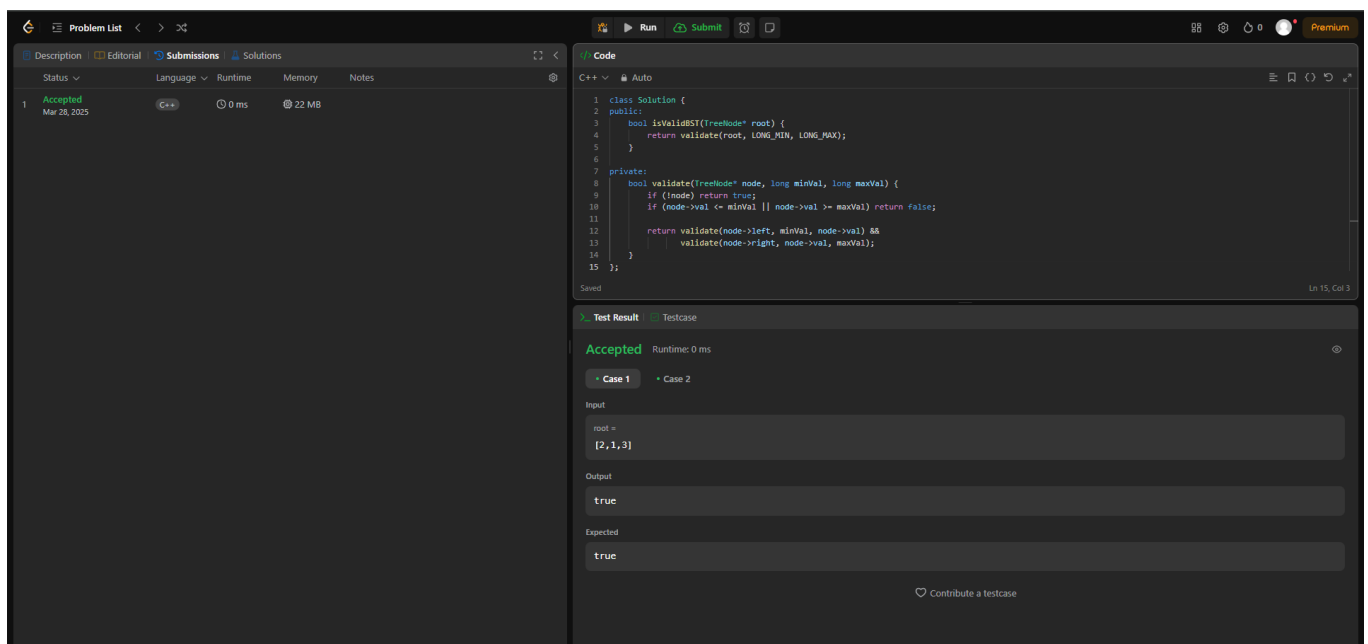


2. Validate Binary Search Tree: <https://leetcode.com/problems/validate-binary-search-tree/>

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

private:
    bool validate(TreeNode* node, long minVal, long maxVal) {
        if (!node) return true;
        if (node->val <= minVal || node->val >= maxVal) return false;

        return validate(node->left, minVal, node->val) &&
            validate(node->right, node->val, maxVal);
    }
};
```



3. Symmetric Tree: <https://leetcode.com/problems/symmetric-tree/>

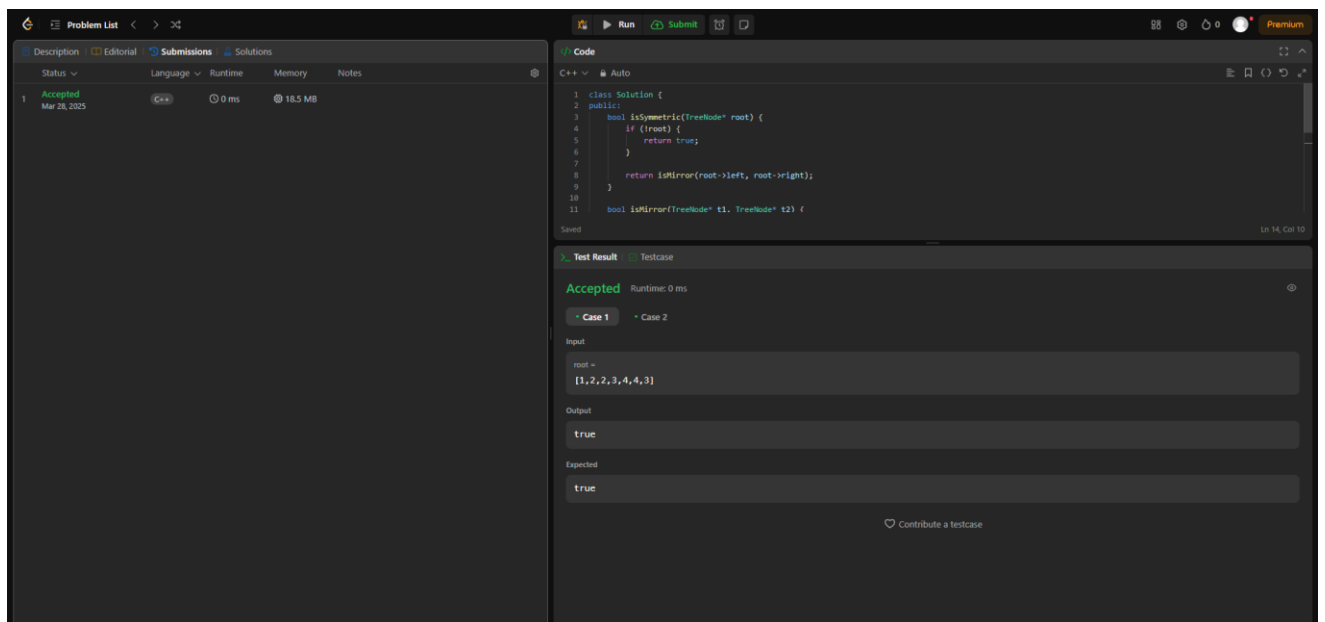
```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (!root) {
            return true;
        }

        return isMirror(root->left, root->right);
    }

    bool isMirror(TreeNode* t1, TreeNode* t2) {
        if (!t1 && !t2) {
            return true;
        }

        if (!t1 || !t2) {
            return false;
        }

        return (t1->val == t2->val) &&
            isMirror(t1->left, t2->right) &&
            isMirror(t1->right, t2->left);
    }
};
```



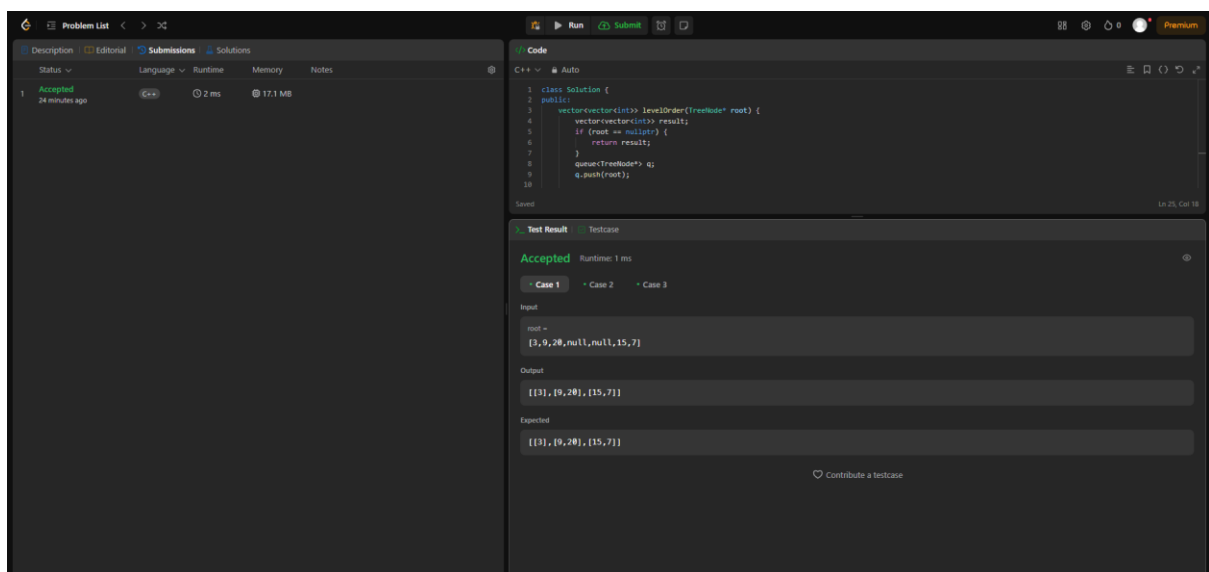
4. Binary Tree Level Order Traversal: <https://leetcode.com/problems/binary-tree-level-order-traversal/>

```
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;
    }
};
```



5. Convert Sorted Array to Binary Search Tree: <https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/>

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

    TreeNode* buildTree(vector<int>& nums, int left, int right) {
        if (left > right) {
            return nullptr;
        }

        int mid = left + (right - left) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = buildTree(nums, left, mid - 1);
        root->right = buildTree(nums, mid + 1, right);

        return root;
    }
};
```

The screenshot shows the LeetCode interface for the problem "Convert Sorted Array to Binary Search Tree". The left sidebar contains the problem description, a submission status of "Accepted", and a performance graph. The main area displays the C++ code for the solution. The right sidebar shows the test results for two cases.

Problem Description: Given an array `nums` representing a sorted (ascending) integer array from a search tree, convert `nums` to a binary search tree. Return `root`, the root of the binary search tree.

Submission Status: Accepted. Runtime: 0 ms, Memory: 22.91 MB. 100.00% passed.

Code:

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

    TreeNode* buildTree(vector<int>& nums, int left, int right) {
        if (left > right) {
            return nullptr;
        }

        int mid = left + (right - left) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = buildTree(nums, left, mid - 1);
        root->right = buildTree(nums, mid + 1, right);

        return root;
    }
};
```

Test Results:

Case 1: Input: `nums = [-10,-3,0,5,9]`. Output: `[0,-10,5,null,-3,null,9]`. Expected: `[0,-3,0,-10,null,5]`. Status: Accepted.

Case 2: Input: `nums = [-10,-3,0,5,9]`. Output: `[0,-10,5,null,-3,null,9]`. Expected: `[0,-3,0,-10,null,5]`. Status: Accepted.

6. Binary Tree Inorder Traversal: <https://leetcode.com/problems/binary-tree-inorder-traversal/>

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

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

The screenshot shows a LeetCode problem page for "Binary Tree Inorder Traversal". The left sidebar displays the problem description, a submission status of "Accepted" with 71/71 test cases passed, and performance metrics: 0 ms runtime, 10.82 MB memory, and 66.54% beats. The main area shows the C++ code for the solution, which is a recursive function. The right sidebar shows the "Test Result" section, indicating the solution is "Accepted" with a runtime of 0 ms. It lists four test cases, with Case 1 selected, showing an input of [1,null,2,3] and an output of [1,2,2].

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

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

7. Binary Tree Zigzag Level Order Traversal: <https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/>

```

class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (root == nullptr) {
            return result;
        }
        deque<TreeNode*> q;
        q.push_back(root);
        bool leftToRight = true;

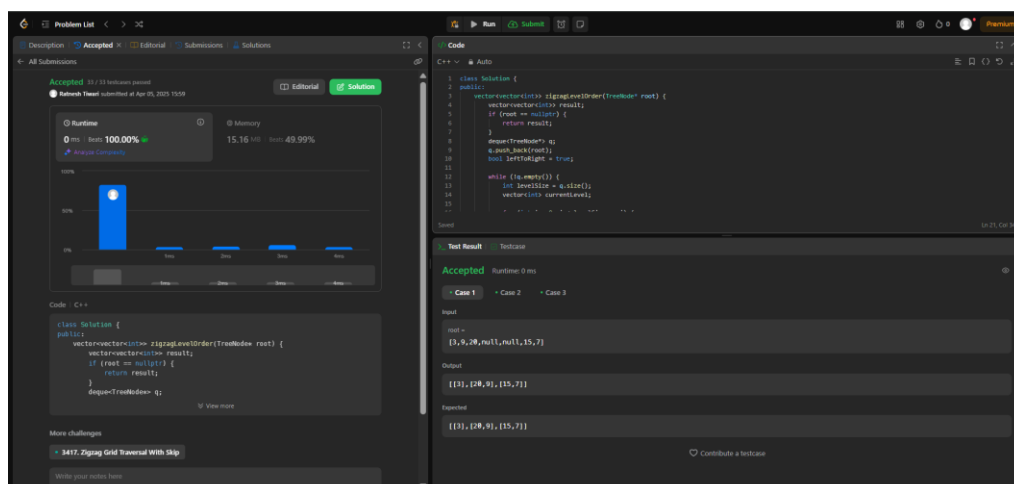
        while (!q.empty()) {
            int levelSize = q.size();
            vector<int> currentLevel;

            for (int i = 0; i < levelSize; ++i) {
                TreeNode* node = q.front();
                q.pop_front();
                currentLevel.push_back(node->val);

                if (node->left) {
                    q.push_back(node->left);
                }
                if (node->right) {
                    q.push_back(node->right);
                }
            }

            if (!leftToRight) {
                reverse(currentLevel.begin(), currentLevel.end());
            }
            result.push_back(currentLevel);
            leftToRight = !leftToRight;
        }
        return result;
    }
};

```



8. Construct Binary Tree from Inorder and Postorder Traversal:

<https://leetcode.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/>

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        int postIndex = postorder.size() - 1;
        return build(inorder, postorder, 0, inorder.size() - 1, postIndex);
    }

    TreeNode* build(vector<int>& inorder, vector<int>& postorder, int inStart, int inEnd, int& postIndex) {
        if (inStart > inEnd) {
            return nullptr;
        }

        int rootVal = postorder[postIndex--];
        TreeNode* root = new TreeNode(rootVal);

        int rootIndex = find(inorder.begin(), inorder.end(), rootVal) - inorder.begin();

        root->right = build(inorder, postorder, rootIndex + 1, inEnd, postIndex);
        root->left = build(inorder, postorder, inStart, rootIndex - 1, postIndex);

        return root;
    }
};
```

The screenshot displays the LeetCode submission interface for the problem "Construct Binary Tree from Inorder and Postorder Traversal". The left panel shows the problem details, including the runtime graph and the C++ code. The right panel shows the test results, indicating that the solution is accepted.

Runtime Graph: The graph shows the runtime of the solution across multiple test cases. The x-axis represents the number of test cases (0 to 30), and the y-axis represents the runtime in milliseconds (0 to 30 ms). The solution's runtime is consistently low, around 3 ms.

C++ Code: The code is a C++ implementation of the solution. It defines a class `Solution` with a public method `buildTree` that takes two vectors, `inorder` and `postorder`, and returns a `TreeNode*`. The method uses a recursive helper function `build` to construct the binary tree.

Test Result: The test result shows that the solution is accepted. The runtime is 3 ms, and the memory usage is 27.10 MB. The test cases are as follows:

| Case | Input | Output | Expected |
|--------|--|-------------------------|-------------------------|
| Case 1 | inorder = [9,3,15,20,7] postorder = [9,15,7,20,3] | [3,9,20,null,null,15,7] | [3,9,20,null,null,15,7] |
| Case 2 | | | |

9. Kth Smallest element in a BST: <https://leetcode.com/problems/kth-smallest-element-in-a-bst/>

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        stack<TreeNode*> s;
        while (true) {
            while (root) {
                s.push(root);
                root = root->left;
            }
            root = s.top();
            s.pop();
            if (--k == 0) {
                return root->val;
            }
            root = root->right;
        }
    }
};
```

The screenshot displays the LeetCode interface for the problem 'Kth Smallest Element in a BST'. On the left, the 'Runtime' tab shows a performance graph with a single blue bar at 100.00% efficiency, 0 ms runtime, and 24.49 MB memory usage. Below the graph, the C++ code is shown. The main editor in the center contains the same C++ code as provided in the text block. On the right, the 'Test Result' tab is active, showing 'Accepted' status with a runtime of 0 ms. It lists two test cases: 'Case 1' with input root = [3,1,4,null,2] and k = 1, and 'Case 2' with the same input and k = 2. The output for both cases is 1.

10. Populating Next Right Pointers in Each Node:

<https://leetcode.com/problems/populating-next-right-pointers-in-each-node/>

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) {
            return nullptr;
        }
        Node* levelStart = root;

        while (levelStart->left) {
            Node* current = levelStart;
            while (current) {
                current->left->next = current->right;
                if (current->next) {
                    current->right->next = current->next->left;
                }
                current = current->next;
            }
            levelStart = levelStart->left;
        }
        return root;
    }
};
```

The screenshot shows a LeetCode interface with the solution for the problem 'Populating Next Right Pointers in Each Node'. The left sidebar displays the problem description, a runtime graph, and a list of related challenges. The main area shows the C++ code for the solution, and the bottom right shows the test results.

Runtime Graph: The graph shows the runtime of the solution across different test cases. The x-axis represents the number of nodes (from 1 to 1000), and the y-axis represents the runtime in milliseconds (from 0 to 100). The solution shows a consistent runtime of approximately 11ms across all test cases.

Test Results: The solution is marked as 'Accepted' with a runtime of 3ms. The input for Case 1 is [1,2,3,4,5,6,7], and the output is [1,2,3,4,5,6,7].