Experiment 5

Student Name: Amrita
UID: 22BCS16592
Branch: BE-CSE
Section/Group: 22BCS_IOT-638/B
Semester: 6th
Date of Performance: 21/02/2025
Subject Name: AP Lab-II
Subject Code: 22CSP-351

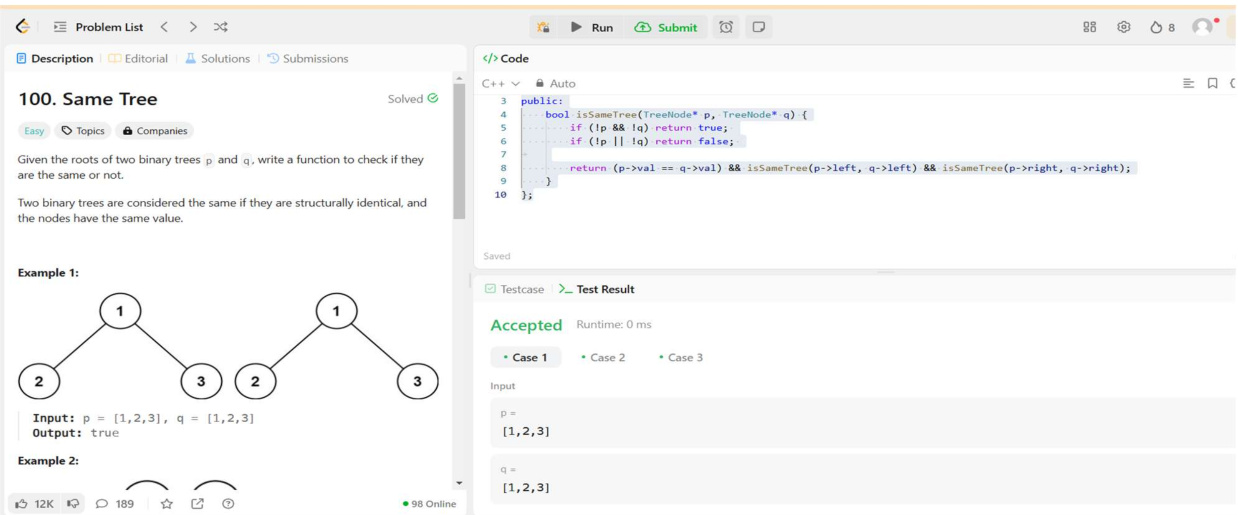
Same Tree

1. **Aim:** Given the roots of two binary trees p and q, write a function to check if they are the same or not.Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

2. **Code:**

```
class Solution {
public:
bool isSameTree(TreeNode* p, TreeNode* q) {
if (!p && !q) return true;
if (!p || !q) return false; return (p->val == q->val) && isSameTree(p->left, q->left) &&
isSameTree(p->right,>right);
        }
    };
```

3. **Output:**



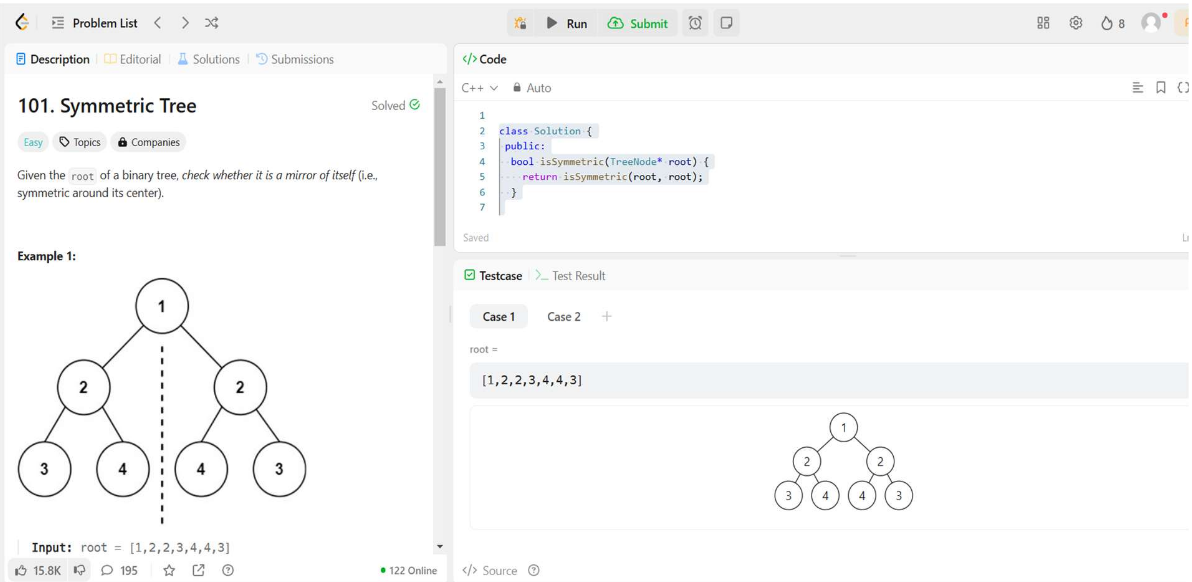
Symmetric Tree

1. **Aim:** Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center)

2. Code:

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

 private:
  bool isSymmetric(TreeNode* p, TreeNode* q) {
    if (!p || !q)
      return p == q;
    return p->val == q->val && isSymmetric(p->left, q->right) && isSymmetric(p->right,
q->left);
  }
};
```

3. Output:



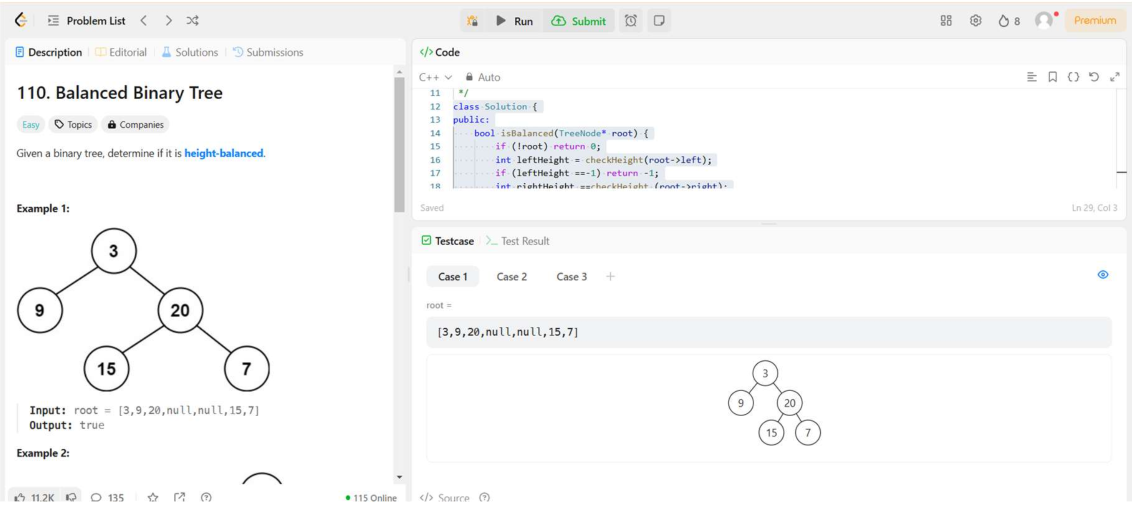
Balanced Binary Tree

1. **Aim:** Given a binary tree, determine if it is height-balanced.
2. **Code:**

```
class Solution {
public:
    bool isBalanced(TreeNode* root) {
        if (!root) return 0;
        int leftHeight = checkHeight(root->left);
        if (leftHeight ==-1) return -1;
        int rightHeight ==checkHeight (root->right);
```

```
        if (rightHeight==-1) return -1;
        if (abs(leftHeight -rightHeight)>1) return -1;
        return max (leftHeight,rightHeight) +1;

    }
    bool isBalanced(TreeNode* root) {

        return checkHeight(root) !=-1;
    }

};
```

3. **Output:**



<u>Path Sum</u>

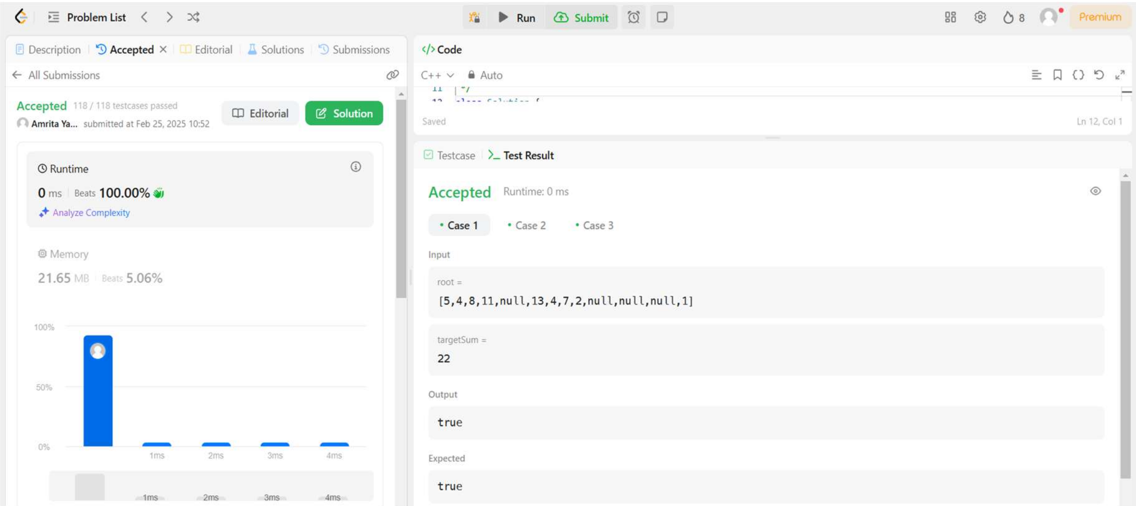
1. **Aim:** Given the root of a binary tree and an integer target Sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals target Sum. A leaf is a node with no children.

2. **Code:**

```
class Solution {
public:
        bool hasPathSum(TreeNode* root, int targetSum) {
            if (!root) {
                return false;
            }

            if (!root->left && !root->right) {
```

```
            return targetSum == root->val;
        }

        bool left_sum = hasPathSum(root->left, targetSum - root->val);
        bool right_sum = hasPathSum(root->right, targetSum - root->val);

        return left_sum || right_sum;
    }
};
```

3. **Output:**



Count Complete Tree

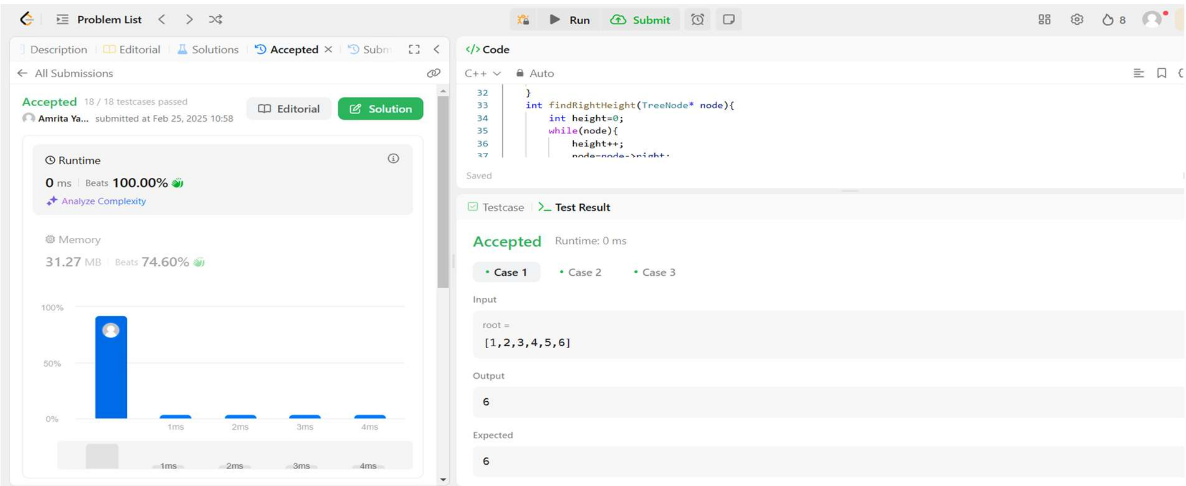
1. **Aim:** Given the root of a complete binary tree, return the number of the nodes in the tree. According to Wikipedia, every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes CO3inclusive at the last level h. Design an algorithm that runs in less than O(n) time complexity.

2. **Code:**

```
class Solution {
public:
  int findLeftHeight(TreeNode* node){
    int height=0;
    while(node){
```

```
            height++;
            node=node->left;
        }
        return height;
    }
    int findRightHeight(TreeNode* node){
        int height=0;
        while(node){
            height++;
            node=node->right;
        }
        return height;
    }
    int countNodes(TreeNode* root) {
        if(root==NULL){
            return 0;
        }
        int lh=findLeftHeight(root);
        int rh=findRightHeight(root);
        if(lh==rh){
            return (1<<lh)-1;
        }
        return 1+countNodes(root->left)+countNodes(root->right);
    }
};
```

3. Output:



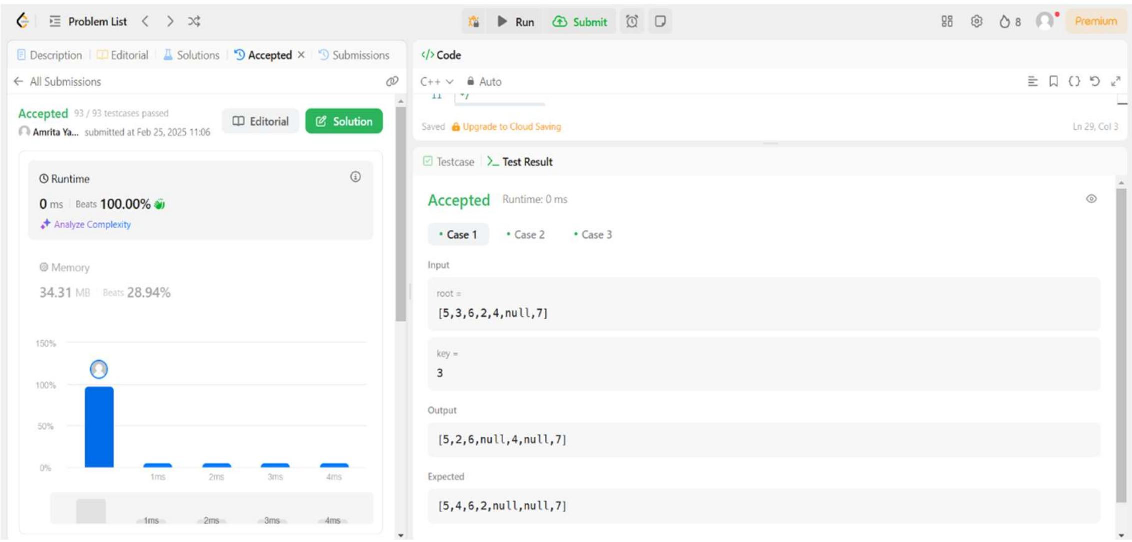
Delete Node in a BST

1. **Aim:** Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST. Basically, the deletion can be divided into two stages: Search for a node to remove. If the node is found, delete the node.

2. Code:

```
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if(root)
            if(key < root->val) root->left = deleteNode(root->left, key);
            else if(key > root->val) root->right = deleteNode(root->right, key);
            else{
                if(!root->left && !root->right) return NULL;
                if (!root->left || !root->right)
                    return root->left ? root->left :
root>right;
                TreeNode* temp = root->left;
                while(temp->right != NULL) temp = temp->right;
                root->val = temp->val;
                root->left = deleteNode(root->left, temp->val);
            }
        return root;
    }
};
```

3. Output:



Diameter of Binary Tree

1. **Aim:** Given the root of a binary tree, return the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root. The length of a path between two nodes is represented by the number of edges between them.

2. Code:

```
class Solution {
public:
    pair<int, int> diameterOfBinaryTreeFast(TreeNode* root){
        if(!root){
            pair<int, int> p = make_pair(0, 0);
            return p;
        }
        pair<int, int> ans;
        pair<int, int> left =  diameterOfBinaryTreeFast(root->left);
        pair<int, int> right =  diameterOfBinaryTreeFast(root->right);
        ans.first = max(left.first, max(right.first, left.second+right.second+1));
        ans.second = max(left.second, right.second) + 1;
        return ans;
    }
    int diameterOfBinaryTree(TreeNode* root) {
        return diameterOfBinaryTreeFast(root).first - 1;
    }
};
```

3. Output: