EXPERIMENT – 5

Student Name: DEBAJYOTI PAUL **UID: 22BCS11219**

Branch: BE-CSE **Section/Group: IOT-637-B**

Semester: 6 **Date of Performance: 20-02-2025**

Subject Name: Advanced Programming 2 **Subject Code: 22CSP-351**

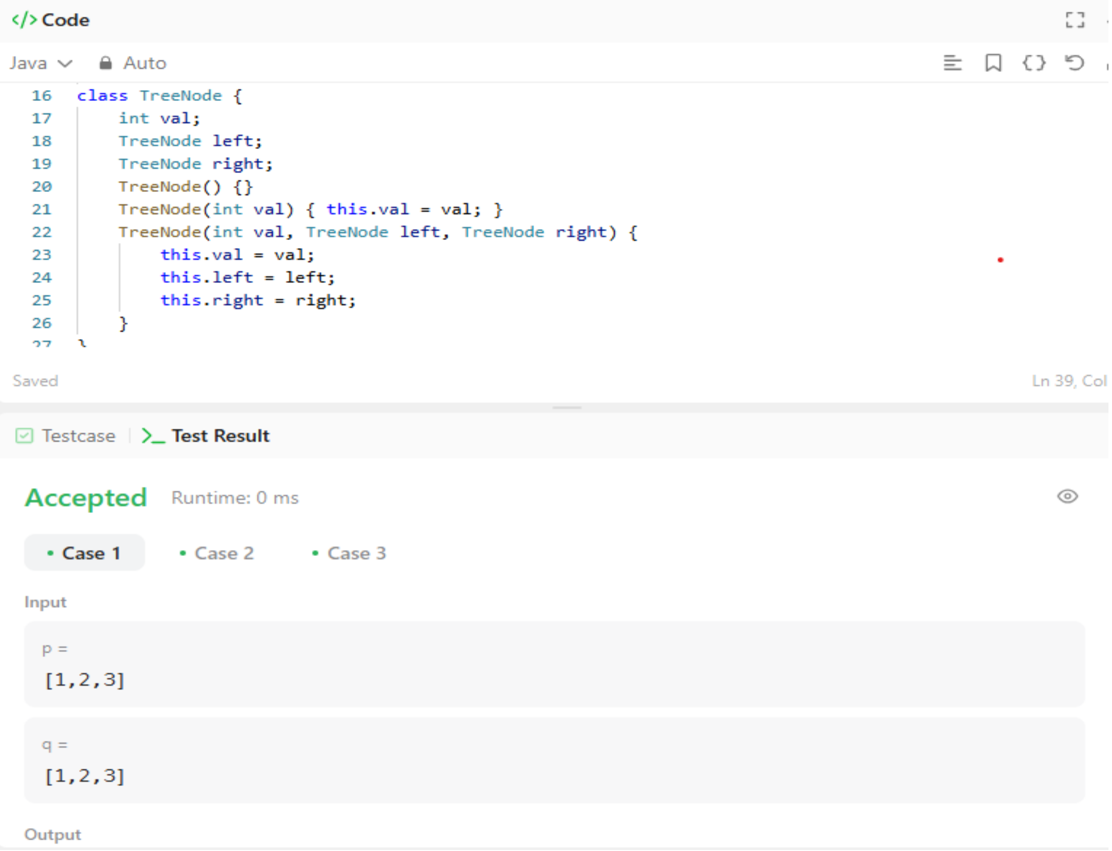
1. **Aim:**Given the roots of two binary trees p and q, write a function to check if they are the same or not.Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

2. **CODE:**

```
 class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

    this.val = val;

    this.left = left;

    this.right = right;

}

}
```

```
class Solution {

    public boolean isSameTree(TreeNode p, TreeNode q) {

        if (p == null && q == null) return true; // Both trees are empty

        if (p == null || q == null) return false; // One tree is empty

        if (p.val != q.val) return false; // Node values do not match



        // Recursively check left and right subtrees

        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);

    }

}
```

3. LEETCODE SS:



```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

Accepted Runtime: 0 ms

Input

p =
[1,2,3]

q =
[1,2,3]

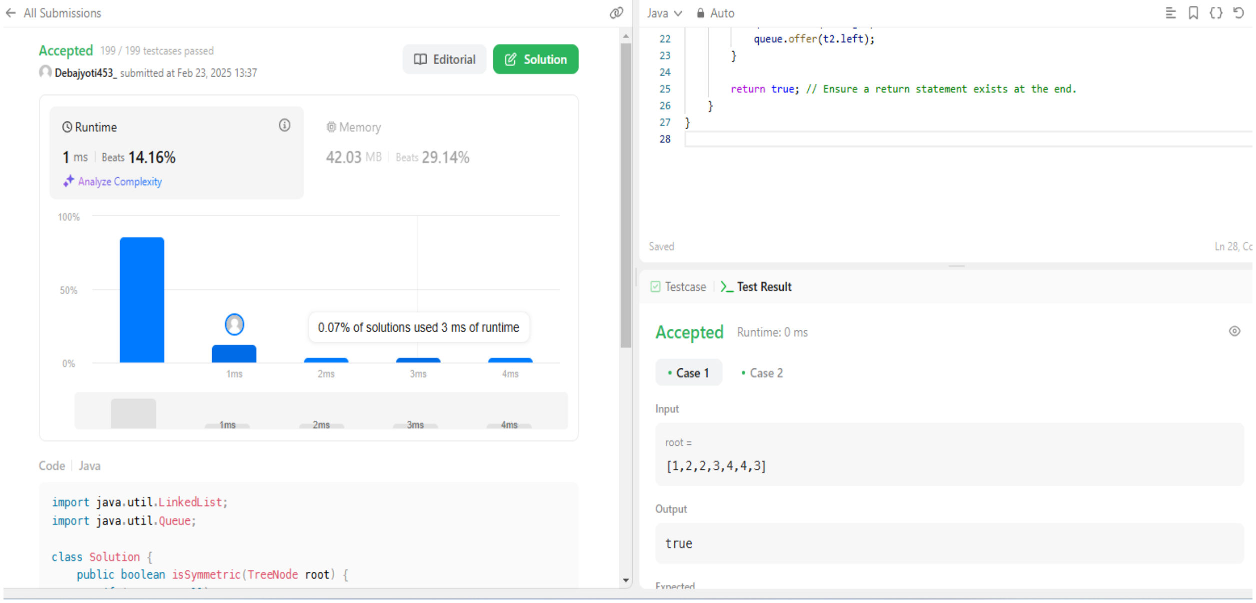
Output

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

5. **CODE:**

```
   import java.util.LinkedList;
import java.util.Queue;
class Solution {
  public boolean isSymmetric(TreeNode root) {
    if (root == null) return true;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root.left);
    queue.offer(root.right);
    while (!queue.isEmpty()) {
      TreeNode t1 = queue.poll();
      TreeNode t2 = queue.poll();
            if (t1 == null && t2 == null) continue;
      if (t1 == null || t2 == null || t1.val != t2.val) return false;
            queue.offer(t1.left);
      queue.offer(t2.right);
      queue.offer(t1.right);
      queue.offer(t2.left);
    }
        return true; // Ensure a return statement exists at the end.
    }

}
```

6. LEETCODE SS:



Accepted 199 / 199 testcases passed

Debajyoti453_ submitted at Feb 23, 2025 13:37

Runtime

1 ms | Beats 14.16%

Memory

42.03 MB | Beats 29.14%

0.07% of solutions used 3 ms of runtime

Code | Java

```
import java.util.LinkedList;
import java.util.Queue;

class Solution {
    public boolean isSymmetric(TreeNode root) {
```

```
                queue.offer(t2.left);
            }

            return true; // Ensure a return statement exists at the end.
        }
}
```

Accepted Runtime: 0 ms

Input

root =

[1,2,2,3,4,4,3]

Output

true

Expected

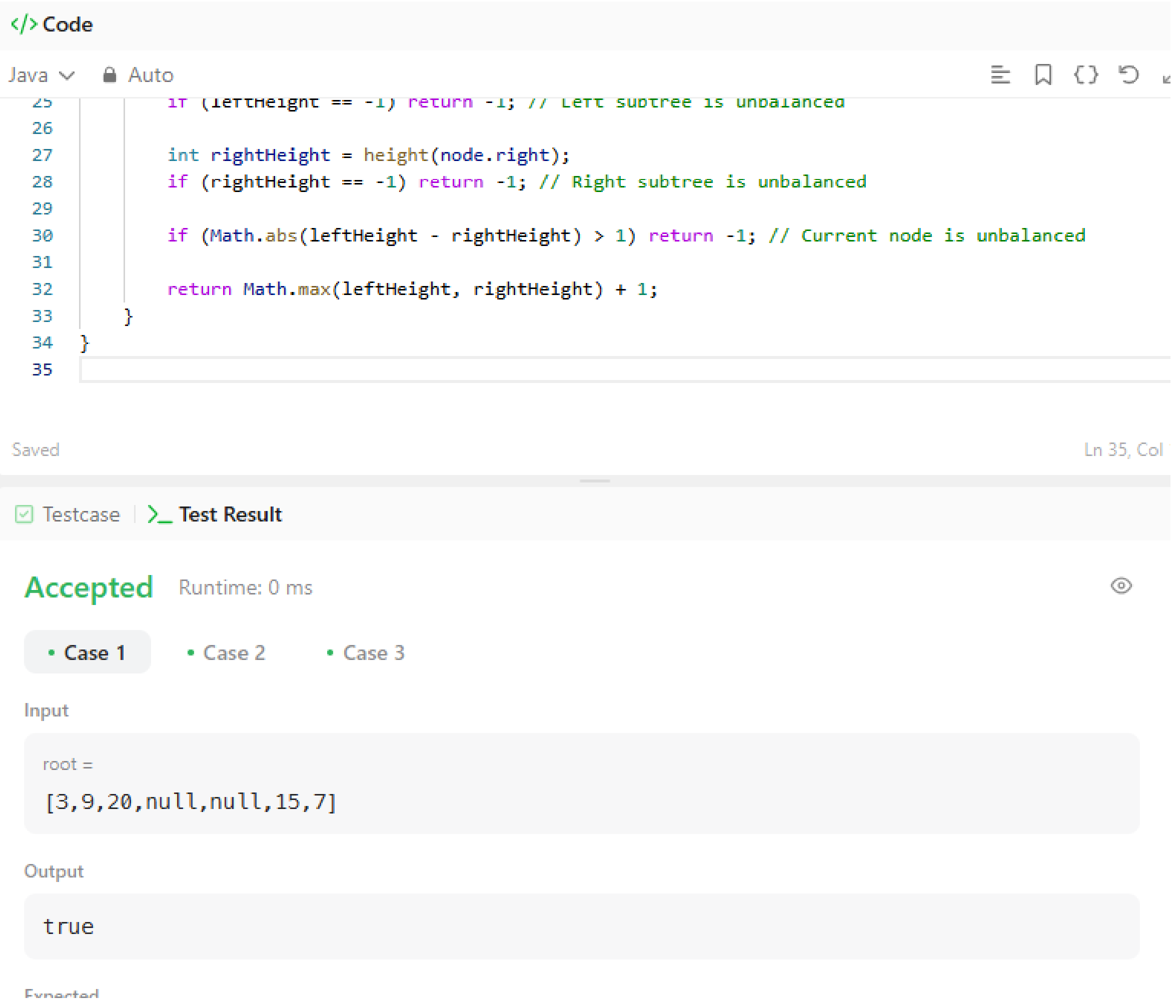
7. **Aim:** - Given a binary tree, determine if it is height-balanced.

8. CODE:

```
 class Solution {

public boolean isBalanced(TreeNode root) {

    return height(root) != -1;

}

private int height(TreeNode node) {

    if (node == null) return 0;

            int leftHeight = height(node.left);

    if (leftHeight == -1) return -1;
```

```
        int rightHeight = height(node.right);

        if (rightHeight == -1) return -1;

        if (Math.abs(leftHeight - rightHeight) > 1) return -1;

        return Math.max(leftHeight, rightHeight) + 1;

    }

}
```

9. LEETCODE SS:



```
        if (leftHeight == -1) return -1; // Left subtree is unbalanced

        int rightHeight = height(node.right);
        if (rightHeight == -1) return -1; // Right subtree is unbalanced

        if (Math.abs(leftHeight - rightHeight) > 1) return -1; // Current node is unbalanced

        return Math.max(leftHeight, rightHeight) + 1;
    }
}

```

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

root =

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

Output

```
true
```

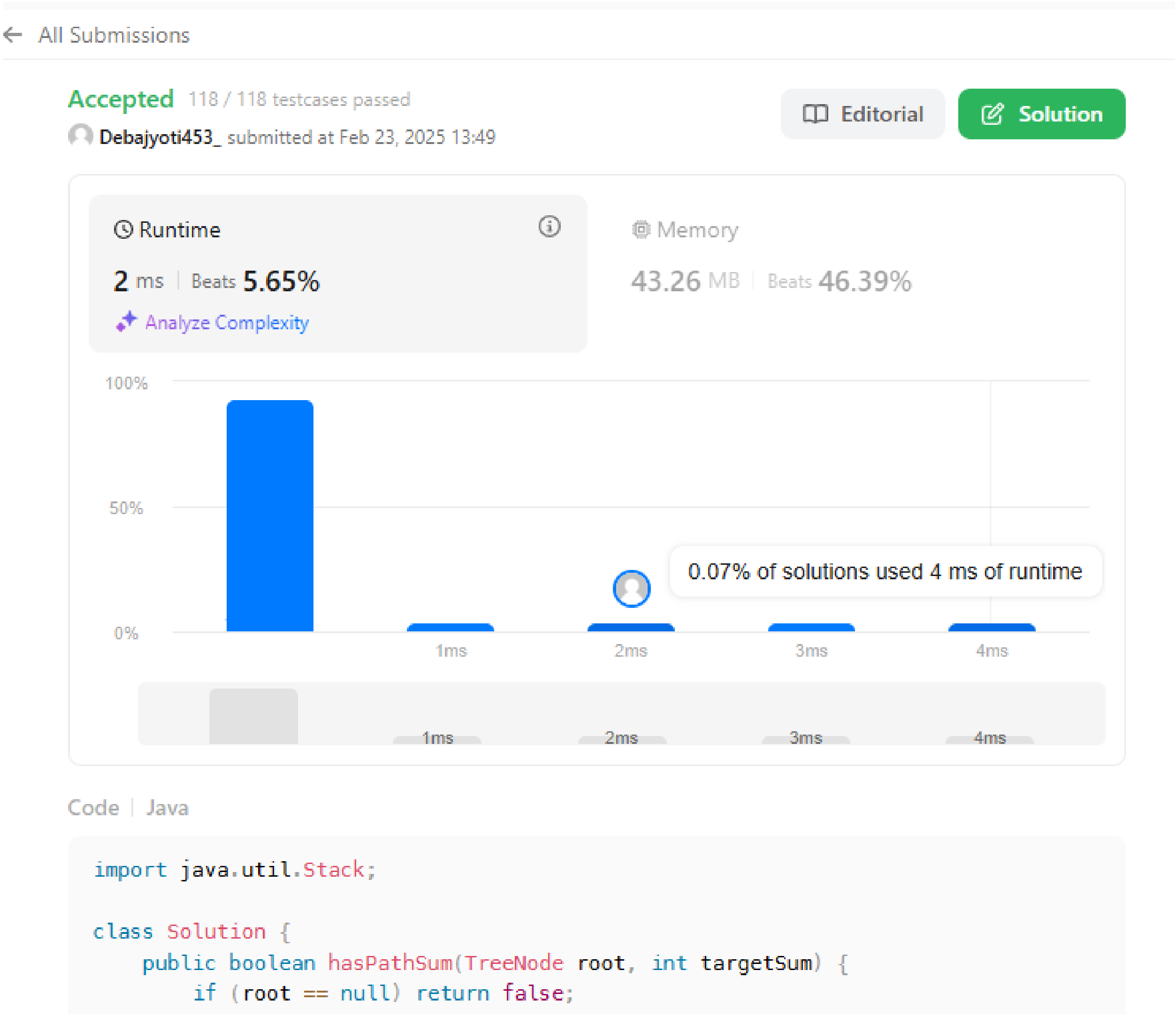
Expected

10. **Aim:** - Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

11. **CODE:**

```
  import java.util.Stack;

class Solution {

  public boolean hasPathSum(TreeNode root, int targetSum) {

    if (root == null) return false;

        Stack<TreeNode> nodeStack = new Stack<>();

    Stack<Integer> sumStack = new Stack<>();

        nodeStack.push(root);

    sumStack.push(targetSum - root.val);

        while (!nodeStack.isEmpty()) {

      TreeNode current = nodeStack.pop();

      int currentSum = sumStack.pop();

      if (current.left == null && current.right == null && currentSum == 0) {

        return true;

      }

            if (current.right != null) {

        nodeStack.push(current.right);

        sumStack.push(currentSum - current.right.val);

      }

            if (current.left != null) {
                nodeStack.push(current.left);

                sumStack.push(currentSum - current.left.val);

            }

        }

                return false;

    }

}
```

12. LEETCODE SS:



Accepted 118 / 118 testcases passed

Debajyoti453_ submitted at Feb 23, 2025 13:49

2 ms | Beats 5.65%

43.26 MB | Beats 46.39%

0.07% of solutions used 4 ms of runtime

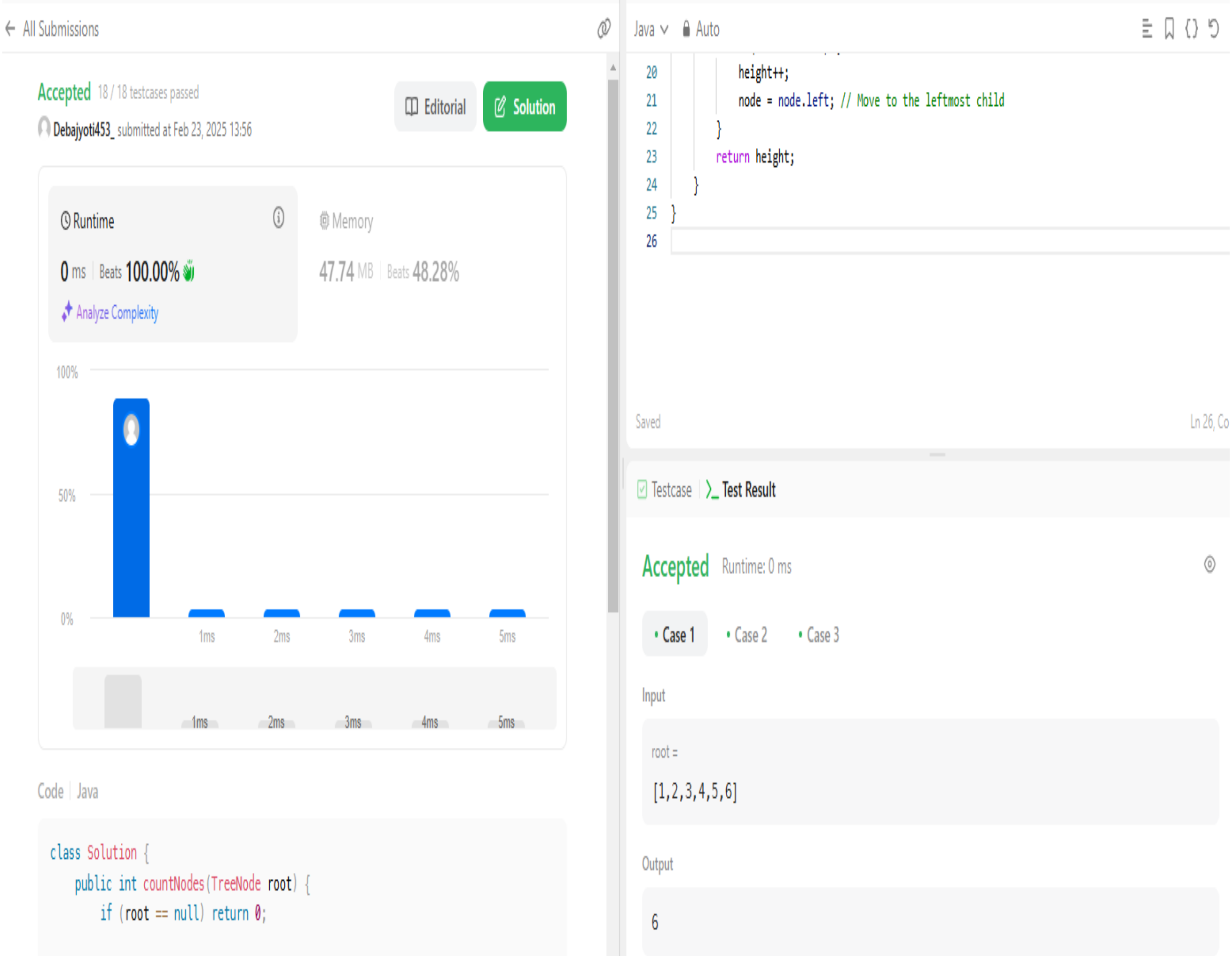
```java
import java.util.Stack;

class Solution {
    public boolean hasPathSum(TreeNode root, int targetSum) {
        if (root == null) return false;
```

13. **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 CO3 inclusive at the last level h. Design an algorithm that runs in less than O(n) time complexity.

14. **CODE:**

```
class Solution {
    public int countNodes(TreeNode root) {
        if (root == null) return 0;


        int leftHeight = getHeight(root.left);
        int rightHeight = getHeight(root.right);


        if (leftHeight == rightHeight) {
            // Left subtree is a full tree
            return (1 << leftHeight) + countNodes(root.right);
        } else {
            // Right subtree is a full tree
            return (1 << rightHeight) + countNodes(root.left);
        }
    }


    private int getHeight(TreeNode node) {
        int height = 0;

        while (node != null) {

            height++;

            node = node.left; // Move to the leftmost child

        }

        return height;

    }

}
```

15. LEETCODE SS:



Accepted 18 / 18 testcases passed

Debajyoti453_ submitted at Feb 23, 2025 13:56

Runtime

0 ms | Beats 100.00%

Analyze Complexity

Memory

47.74 MB | Beats 48.28%

Code | Java

```
class Solution {
    public int countNodes(TreeNode root) {
        if (root == null) return 0;
```

```
            height++;
            node = node.left; // Move to the leftmost child
        }
        return height;
    }
}
```

Saved

Ln 26, Co

Accepted Runtime: 0 ms

Input

root =

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

Output

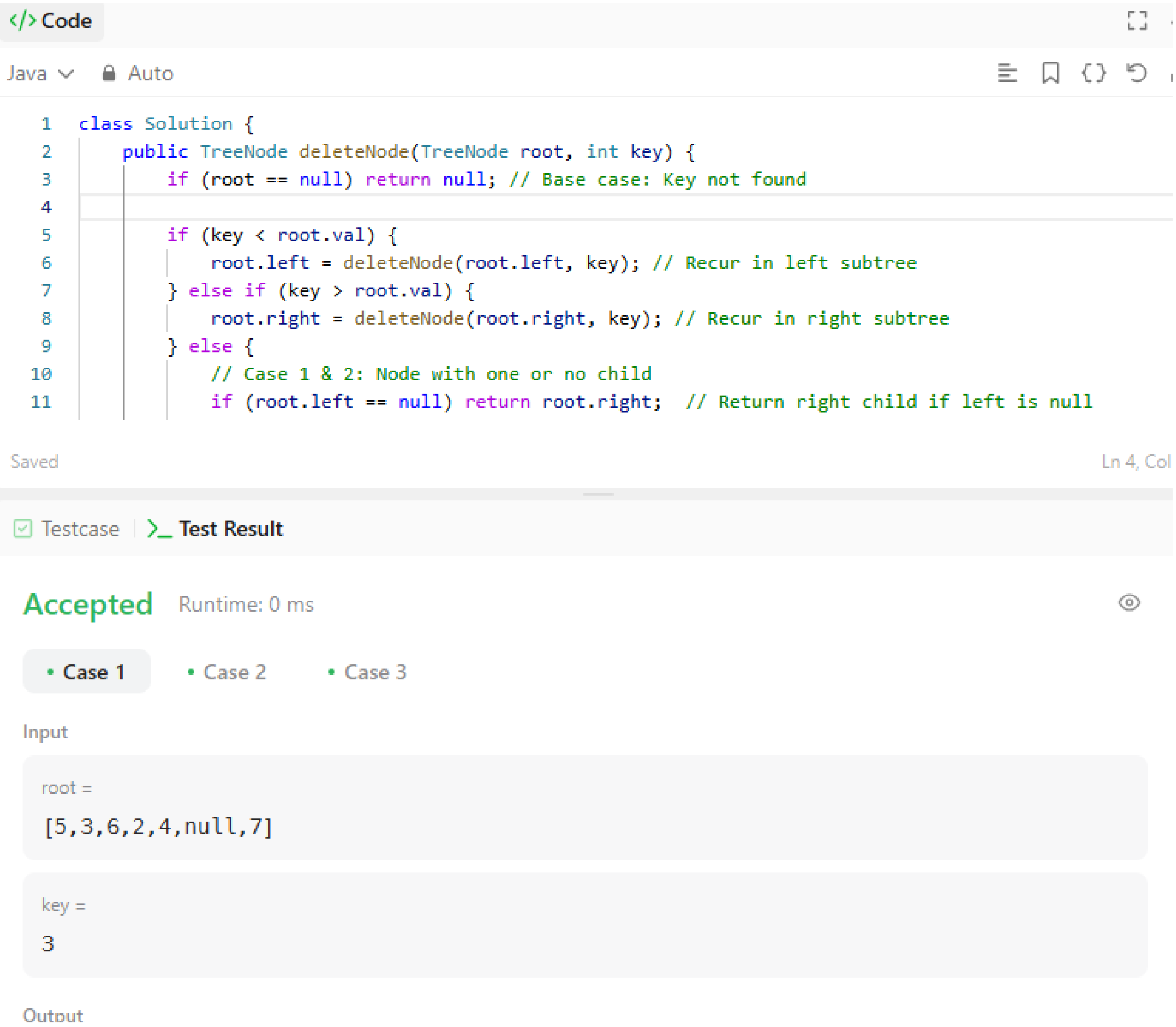
6

16. **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

17. **CODE:**

```
class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) return null; // Base case: Key not found
                if (key < root.val) {
            root.left = deleteNode(root.left, key); // Recur in left subtree
        } else if (key > root.val) {
            root.right = deleteNode(root.right, key); // Recur in right subtree
        } else {
            // Case 1 & 2: Node with one or no child
            if (root.left == null) return root.right;  // Return right child if left is null
            if (root.right == null) return root.left;  // Return left child if right is null
            TreeNode successor = findMin(root.right);  // Find inorder successor
            root.val = successor.val;                  // Copy successor value to root
            root.right = deleteNode(root.right, successor.val); // Delete successor
        }
        return root;
    }
    private TreeNode findMin(TreeNode node) {
        while (node.left != null) {
            node = node.left; // Find the leftmost node
        }
        return node;
    }
}
```

18. LEETCODE SS:



```java
class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) return null; // Base case: Key not found

        if (key < root.val) {
            root.left = deleteNode(root.left, key); // Recur in left subtree
        } else if (key > root.val) {
            root.right = deleteNode(root.right, key); // Recur in right subtree
        } else {
            // Case 1 & 2: Node with one or no child
            if (root.left == null) return root.right;  // Return right child if left is null
```

Accepted Runtime: 0 ms

Input

root =

[5,3,6,2,4,null,7]

key =

3

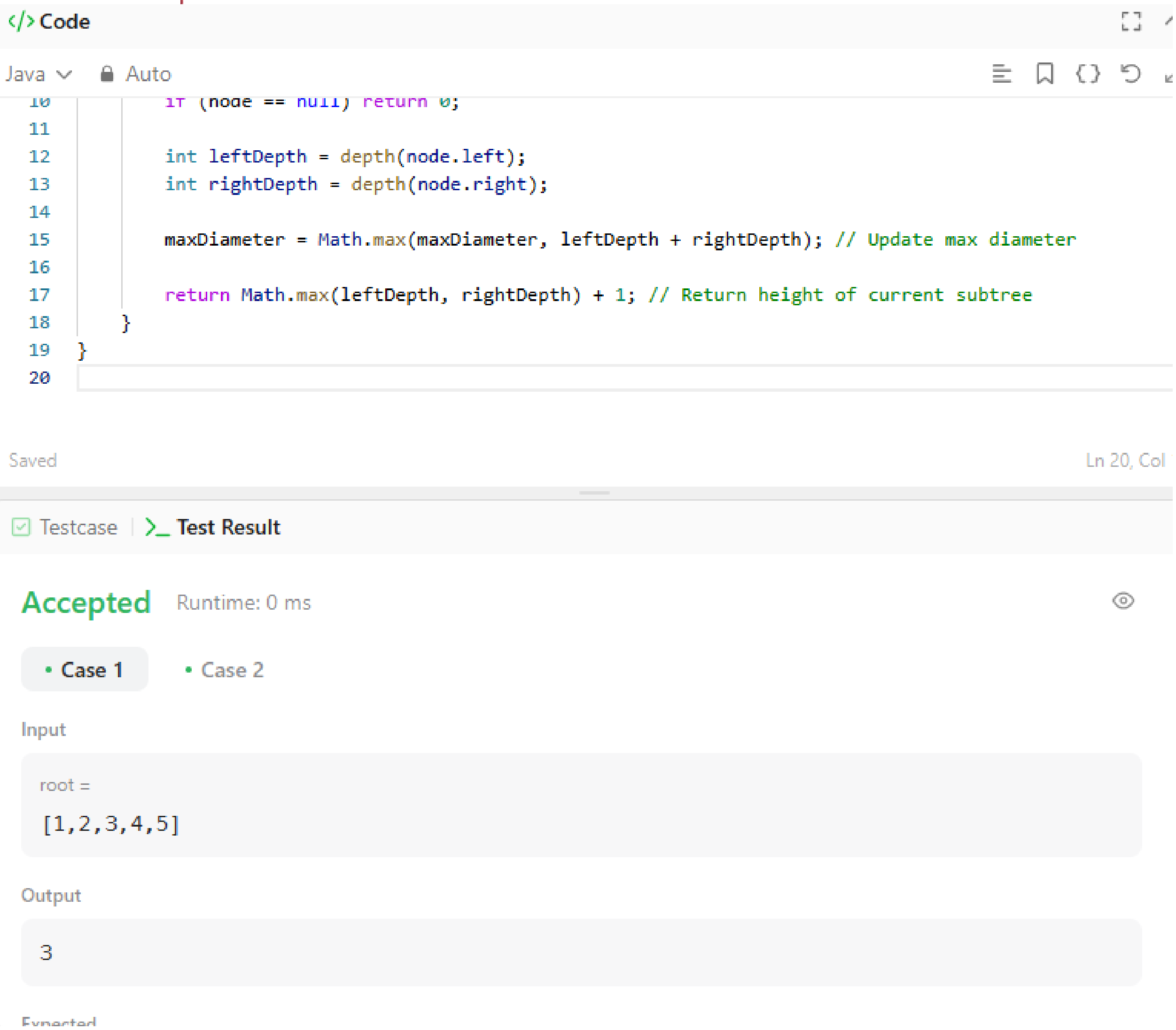
Output

19. **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.

20. **CODE:**

```
class Solution {

    private int maxDiameter = 0; // Stores the maximum diameter found

    public int diameterOfBinaryTree(TreeNode root) {

        depth(root);

        return maxDiameter;

    }

    private int depth(TreeNode node) {

        if (node == null) return 0;

        int leftDepth = depth(node.left);

        int rightDepth = depth(node.right);

        maxDiameter = Math.max(maxDiameter, leftDepth + rightDepth); // Update
     max diameter

        return Math.max(leftDepth, rightDepth) + 1; // Return height of current
     subtree

    }

}
```

21. **LEETCODE SS:**



```
        if (node == null) return 0;

        int leftDepth = depth(node.left);
        int rightDepth = depth(node.right);

        maxDiameter = Math.max(maxDiameter, leftDepth + rightDepth); // Update max diameter

        return Math.max(leftDepth, rightDepth) + 1; // Return height of current subtree
    }
}
```

Accepted Runtime: 0 ms

Input

root =

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

Output

```
3
```

Expected

22. Learning Outcomes:

- Understand tree traversal (recursive comparison of nodes).
- Compare node values and recursively verify left and right subtrees.
- Understand tree height calculation using recursion.
- Learn the definition of a height-balanced tree .
- Apply recursive depth-first traversal to check balance.