Ashish Kumar
22BET10194

Problem 1

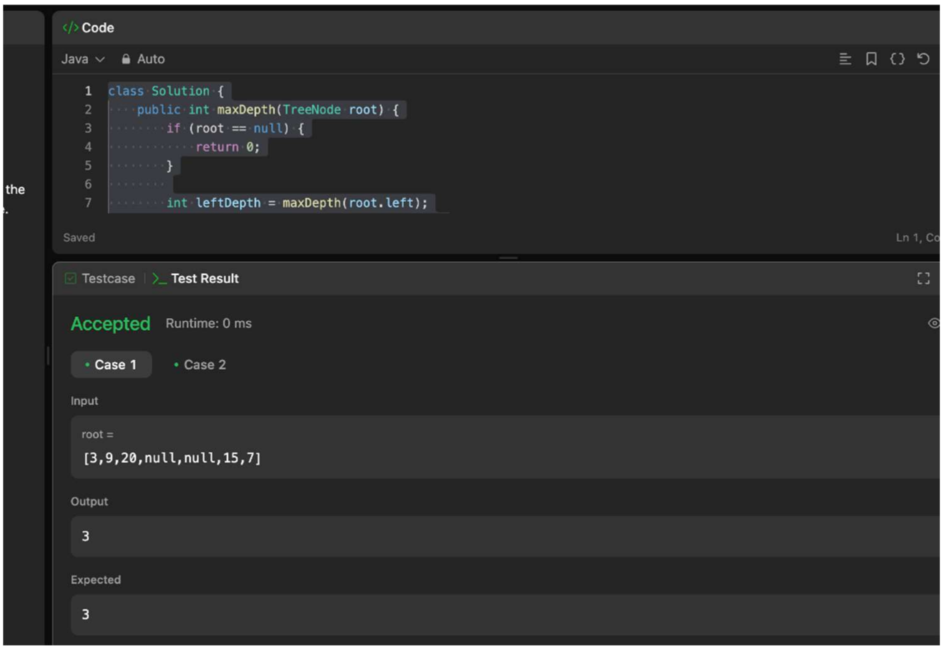
Aim:

Maximum Depth of Binary Tree

Code:

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

        int leftDepth = maxDepth(root.left);
        int rightDepth = maxDepth(root.right);

        return Math.max(leftDepth, rightDepth) + 1;
    }
}
```

Output:



Problem 2

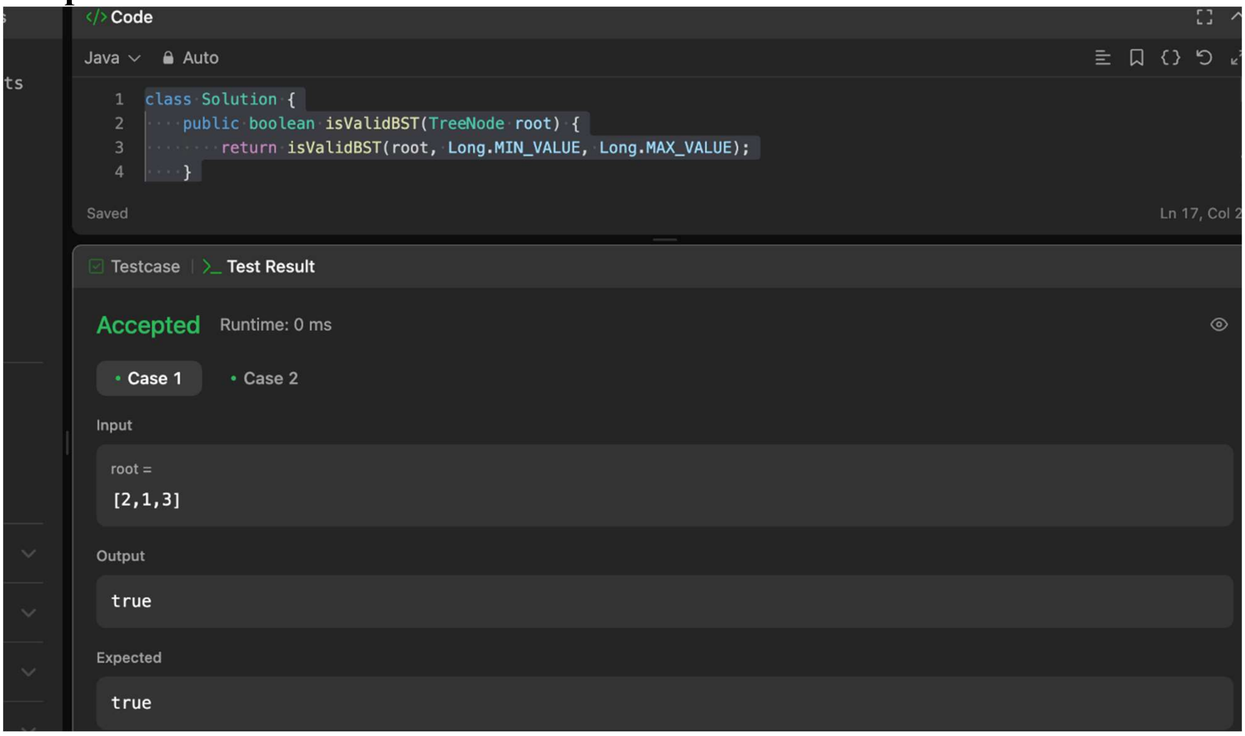
Aim:

Validate Binary Search Tree

Code:

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    private boolean isValidBST(TreeNode node, long min, long max) {
        if (node == null) {
            return true;
        }

        if (node.val <= min || node.val >= max) {
            return false;
        }

        return isValidBST(node.left, min, node.val) && isValidBST(node.right, node.val, max);
    }
}
```

Output:



Problem 3

Aim: Symmetric Tree

Code :
```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null) {
            return true;
        }
        return isMirror(root.left, root.right);
    }

    private boolean isMirror(TreeNode t1, TreeNode t2) {
        if (t1 == null && t2 == null) {
            return true;
        }
        if (t1 == null || t2 == null) {
            return false;
        }

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

Output:

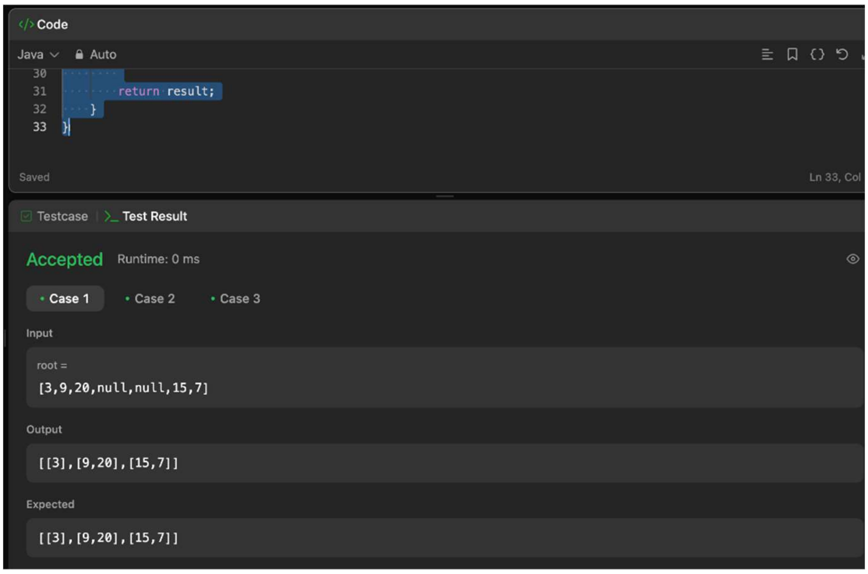
Problem 4

Aim: Binary Tree Level Order Traversal

Code:

```
import java.util.*;

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> currentLevel = new ArrayList<>();

            for (int i = 0; i < levelSize; i++) {
                TreeNode currentNode = queue.poll();
                currentLevel.add(currentNode.val);

                if (currentNode.left != null) {
                    queue.offer(currentNode.left);
                }
                if (currentNode.right != null) {
                    queue.offer(currentNode.right);
                }
            }
            result.add(currentLevel);
        }

        return result;
    }
}
```

Output:



Problem 5

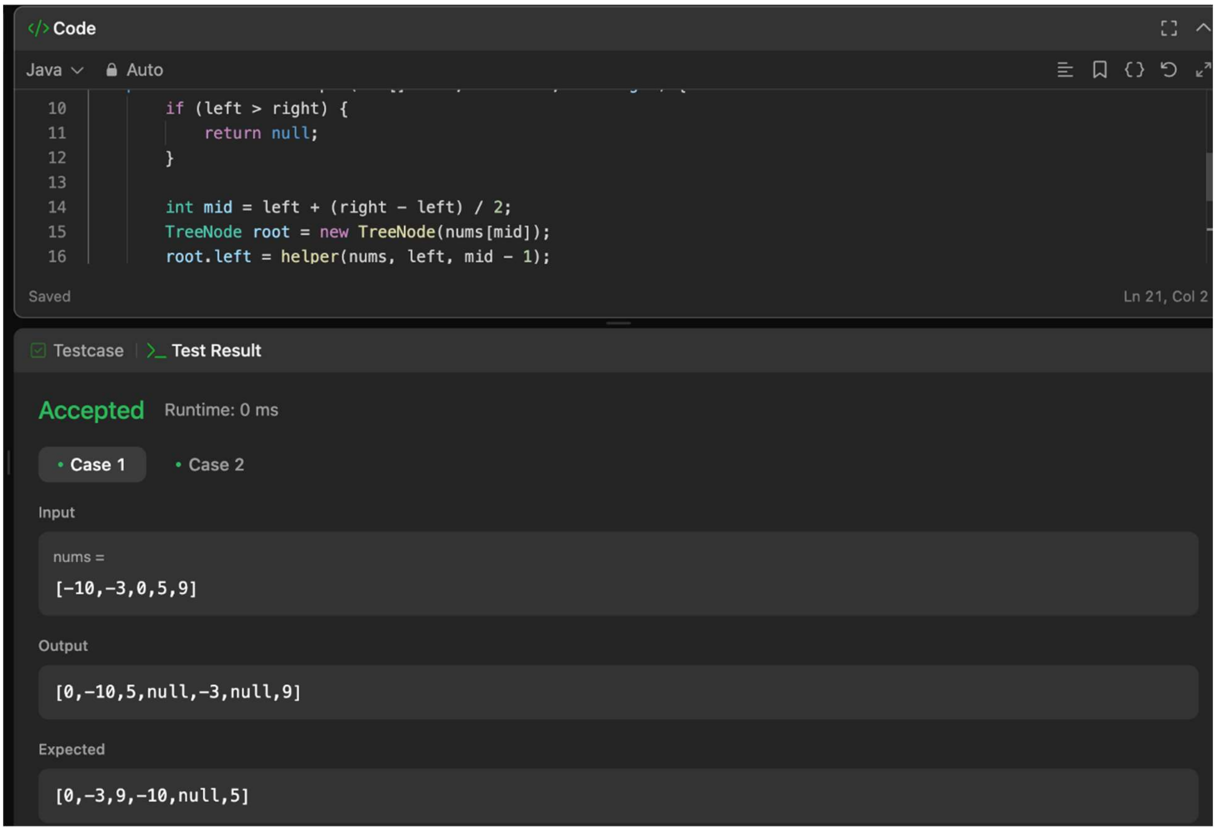
Aim:

Convert Sorted Array to Binary Search Tree

Code:

```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        if (nums == null || nums.length == 0) {
            return null;
        }
        return helper(nums, 0, nums.length - 1);
    }

    private TreeNode helper(int[] nums, int left, int right) {
        if (left > right) {
            return null;
        }

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

        return root;
    }
}
```

Output:



Problem 6

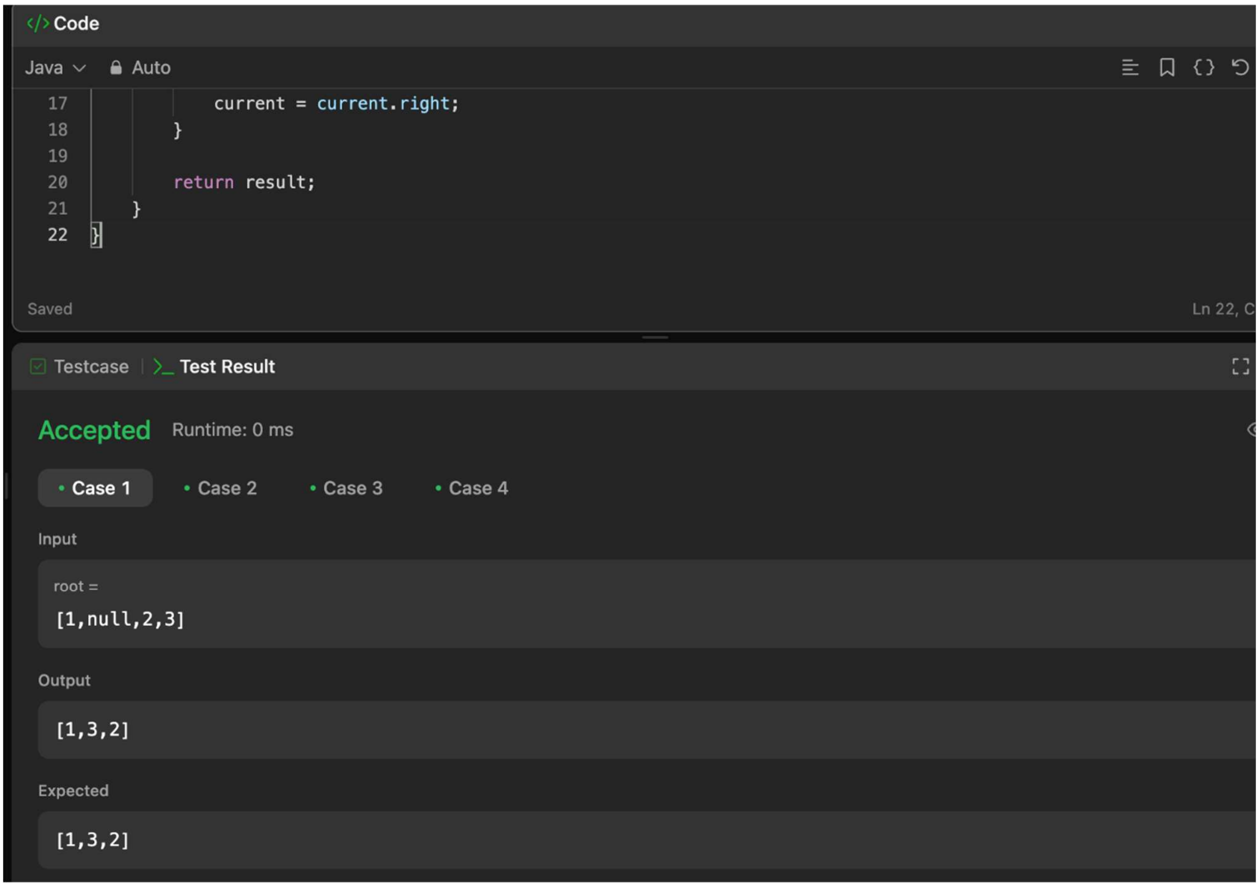
Aim:

Binary Tree Inorder Traversal

Code:

```
import java.util.*;

class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode current = root;

        while (current != null || !stack.isEmpty()) {
            while (current != null) {
                stack.push(current);
                current = current.left;
            }

            current = stack.pop();
            result.add(current.val);
            current = current.right;
        }

        return result;
    }
}
```

Output:



Problem 7

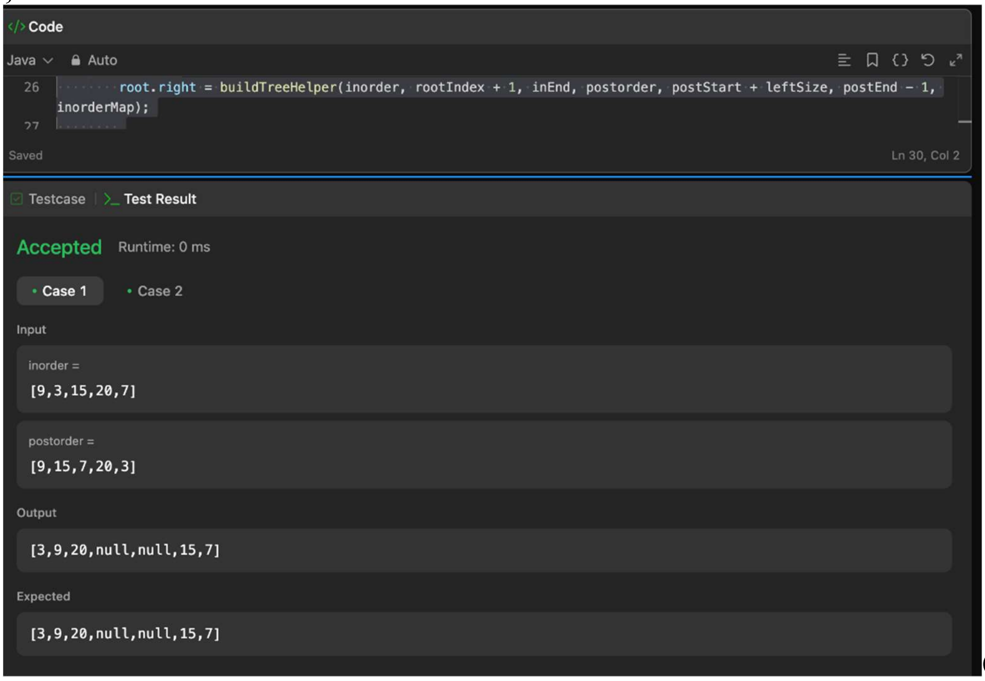
Aim:

Construct Binary Tree from Inorder and Postorder Traversal

Code:

```
import java.util.*;
class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        if (inorder == null || postorder == null || inorder.length != postorder.length) {
            return null;
        }
        Map<Integer, Integer> inorderMap = new HashMap<>();
        for (int i = 0; i < inorder.length; i++) {
            inorderMap.put(inorder[i], i);
        }
        return buildTreeHelper(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1,
inorderMap);
    }
    private TreeNode buildTreeHelper(int[] inorder, int inStart, int inEnd, int[] postorder, int postStart, int
postEnd, Map<Integer, Integer> inorderMap) {
        if (inStart > inEnd || postStart > postEnd) {
            return null;
        }
         int rootVal = postorder[postEnd];
        TreeNode root = new TreeNode(rootVal);
        int rootIndex = inorderMap.get(rootVal);
        int leftSize = rootIndex - inStart;
```

```
root.left = buildTreeHelper(inorder, inStart, rootIndex - 1, postorder, postStart, postStart + leftSize - 1,
inorderMap);
        root.right = buildTreeHelper(inorder, rootIndex + 1, inEnd, postorder, postStart + leftSize, postEnd
- 1, inorderMap);
        return root;
    }
}
```



Output:

Problem 8

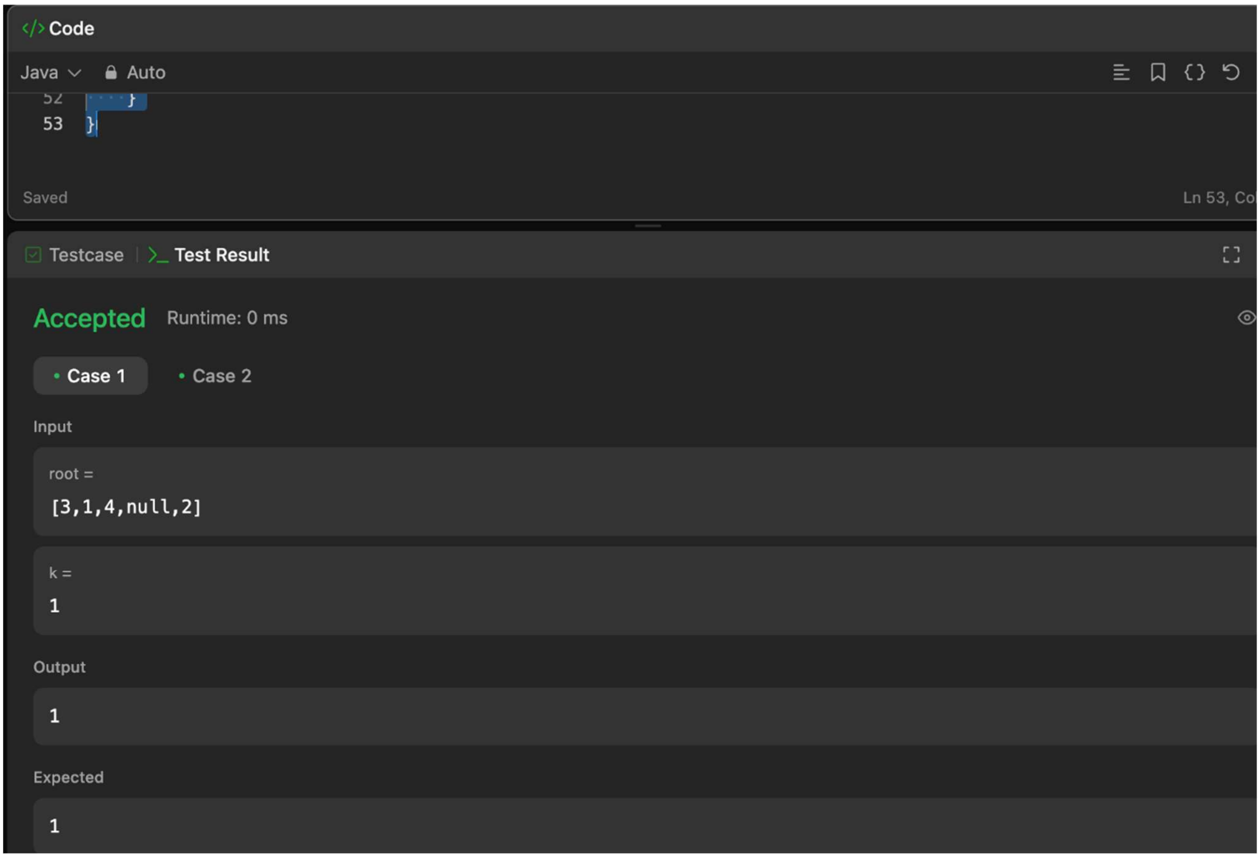
Aim:

Kth Smallest element in a BST

Code:

```
import java.util.*;

class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        if (inorder == null || postorder == null || inorder.length != postorder.length) {
            return null;
        }
        Map<Integer, Integer> inorderMap = new HashMap<>();
        for (int i = 0; i < inorder.length; i++) {
            inorderMap.put(inorder[i], i);
        }
        return buildTreeHelper(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1,
inorderMap);
    }

    private TreeNode buildTreeHelper(int[] inorder, int inStart, int inEnd, int[] postorder, int
postStart, int postEnd, Map<Integer, Integer> inorderMap) {
        if (inStart > inEnd || postStart > postEnd) {
            return null;
        }

        int rootVal = postorder[postEnd];
```

```
        TreeNode root = new TreeNode(rootVal);
        int rootIndex = inorderMap.get(rootVal);
        int leftSize = rootIndex - inStart;

        root.left = buildTreeHelper(inorder, inStart, rootIndex - 1, postorder, postStart, postStart +
leftSize - 1, inorderMap);
        root.right = buildTreeHelper(inorder, rootIndex + 1, inEnd, postorder, postStart + leftSize,
postEnd - 1, inorderMap);

        return root;
    }

    public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode current = root;
        int count = 0;

        while (current != null || !stack.isEmpty()) {
            while (current != null) {
                stack.push(current);
                current = current.left;
            }

            current = stack.pop();
            count++;
            if (count == k) {
                return current.val;
            }

            current = current.right;
        }

        return -1; // Should not reach here if k is valid
    }
}
```
Output:



<u>Problem 9</u>

Aim:

Populating Next Right Pointers in Each Node

Code:

```
class Solution {
    public Node connect(Node root) {
        if (root == null) {
            return null;
        }
```

```
        Node leftmost = root;

        while (leftmost.left != null) {
            Node current = leftmost;

            while (current != null) {
                // Connect left child to right child
                current.left.next = current.right;

                // Connect right child to the left child of next node
                if (current.next != null) {
                    current.right.next = current.next.left;
                }

                // Move to the next node in the current level
                current = current.next;
            }

            // Move to the next level
            leftmost = leftmost.left;
        }

        return root;
    }
}
```

Output: