# Experiment-3

**Student Name: Anupreet Kaur** **UID: 22BCS50071**
**Branch: BE-CSE** **Section/Group: 22BCS_NTPP_602-A**
**Semester: 6th** **Date of Performance: 10/02/2025**
**Subject Name: AP Lab** **Subject Code: 22CSP-351**

## 1. Aim: Divide and Conquor.

- Problem 1.2.1: Binary Tree Level Order Traversal
- Problem 1.2.2: Longest Nice Substring
- Problem 1.2.2: Binary Tree Inorder Traversal

## 2. Objective:

To understand and apply the Divide and Conquer approach to solve problems efficiently by breaking them into smaller subproblems and combining solutions.

## 3. Theory:

Divide and Conquer is a problem-solving paradigm that recursively divides a problem into smaller subproblems, solves each independently, and then merges the results. It is widely used in tree traversals, sorting algorithms, and string manipulations.

- **Binary Tree Level Order Traversal:** Uses a queue to process nodes level by level.
- **Longest Nice Substring:** Divides the string based on character conditions and recursively finds the longest valid substring.
- **Binary Tree Inorder Traversal:** Recursively visits left subtree, root, and right subtree.

## 4. Code:

### Binary Tree Level Order Traversal

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

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

```
        // Queue for BFS
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        // Perform BFS
        while (!queue.isEmpty()) {
            int levelSize = queue.size(); // Get the number of nodes at this level
            List<Integer> currentLevel = new LinkedList<>();

            for (int i = 0; i < levelSize; i++) {
                TreeNode currentNode = queue.poll(); // Get the current node
                currentLevel.add(currentNode.val); // Add the node's value to the current level

                if (currentNode.left != null) queue.offer(currentNode.left); // Add left child to queue
                if (currentNode.right != null) queue.offer(currentNode.right); // Add right child to queue
            }

            // Add the current level's result to the final list
            result.add(currentLevel);
        }

        return result;
    }
}
```

## Implement Queue using Stacks

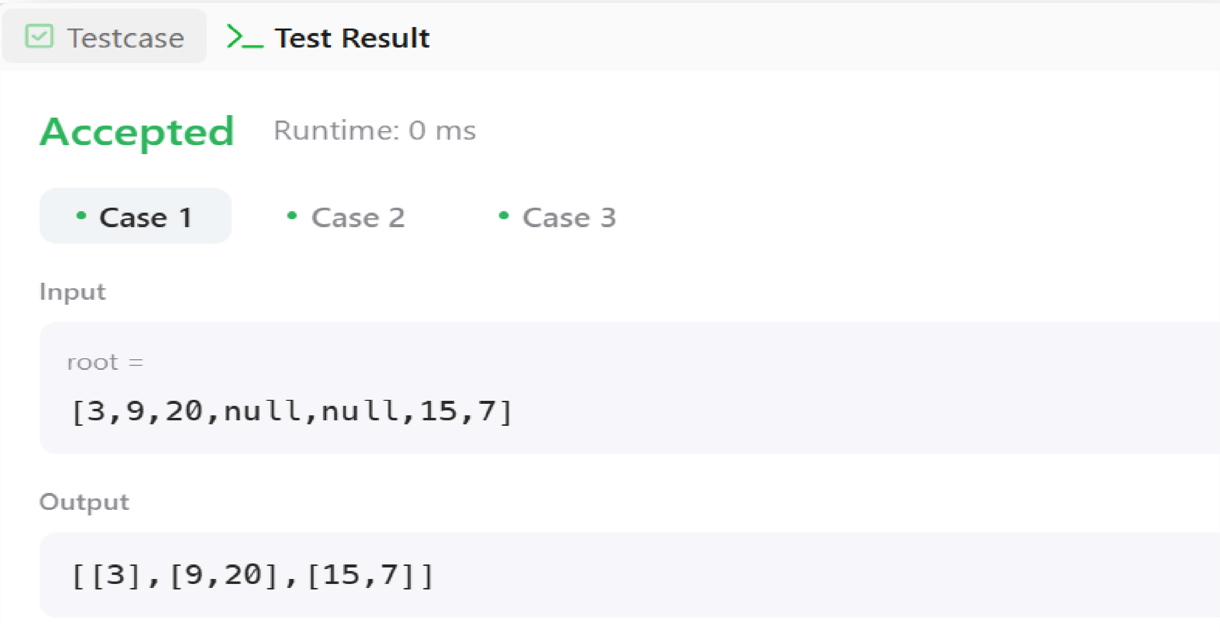
```
class Solution {
    public String longestNiceSubstring(String s) {
        if (s.length() < 2) return "";

        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (s.contains(Character.toString(Character.toUpperCase(ch))) &&
                s.contains(Character.toString(Character.toLowerCase(ch)))) {
                continue;
            }

            String left = longestNiceSubstring(s.substring(0, i));
            String right = longestNiceSubstring(s.substring(i + 1));

            return left.length() >= right.length() ? left : right;
        }
        return s;
    }
}
```

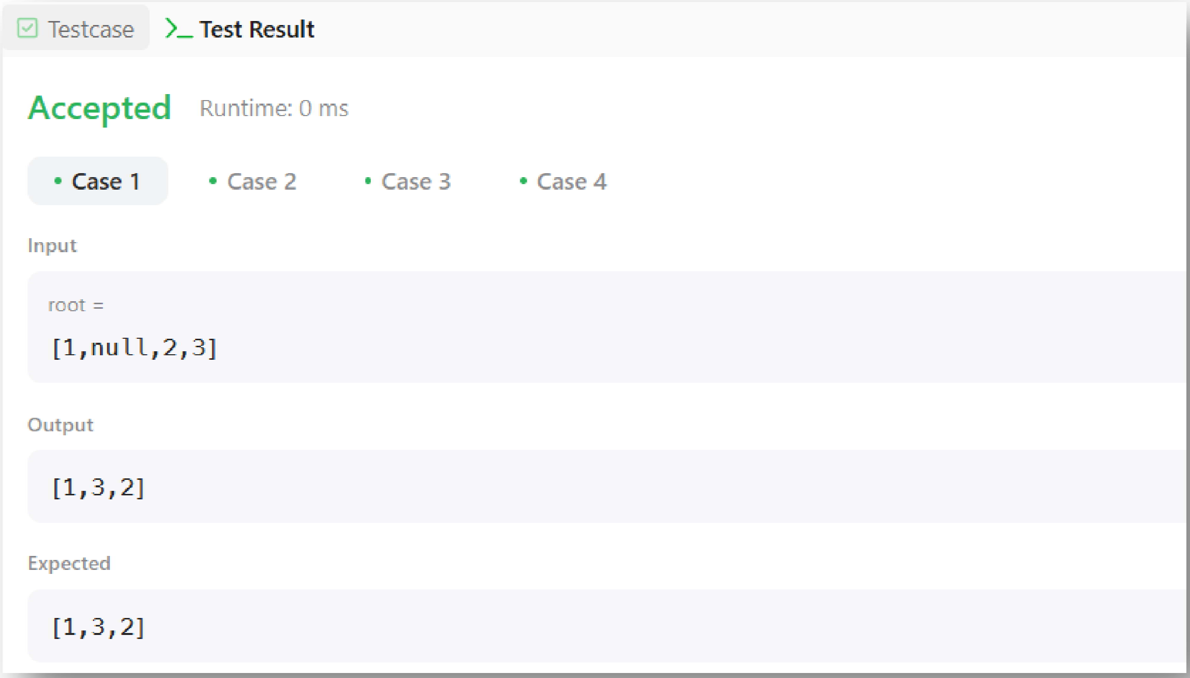
## Binary Tree Inorder Traversal

```
import java.util.*;

class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result; // Base case
```

```
        // Divide step: Recursively get left and right subtree inorder traversal
        List<Integer> leftPart = inorderTraversal(root.left);
        List<Integer> rightPart = inorderTraversal(root.right);

        // Conquer step: Combine left, root, and right results
        result.addAll(leftPart);
        result.add(root.val);
        result.addAll(rightPart);

        return result;
    }
}
```

## 6. Output:




## 7. Learning Outcomes:

- Understand the recursive and iterative approaches to solving tree-based problems.
- Learn how Divide and Conquer simplifies complex problems by breaking them into smaller subproblems.
- Implement efficient solutions for tree traversal and string processing problems.
- Analyze time complexity and optimize recursive solutions.