Experiment-6

Student Name: Anshuman Raj **UID:** 22BET10081

Branch: BE-IT **Section:** 22BET_IOT-702 'A'

Semester: 6th **Date of Performance:**07/03/25

Sub Name: Advanced Programming Lab-2 **Subject Code:** 22ITP-351

Problem 1

1. Aim:

Given the root of a binary tree, return its maximum depth.

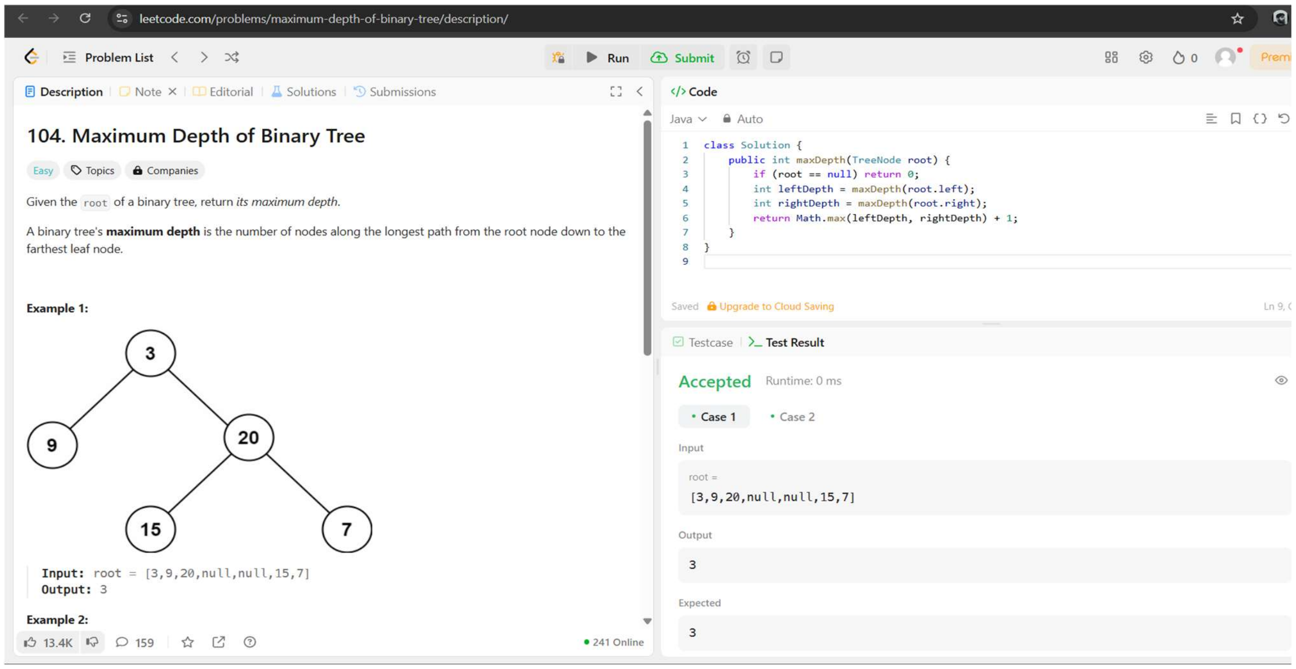
2. Objective:

1. Implement a function to compute the maximum depth of a binary tree.
2. Use recursion (DFS) and iteration (BFS) to solve the problem.
3. Ensure correct handling of edge cases, including an empty tree.

3. Code:

```
import java.util.LinkedList;
import java.util.Queue;
class Solution {
public int maxDepth(TreeNode root) {
if (root == null) return 0;
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);
int depth = 0;
while (!queue.isEmpty()) {
int size = queue.size();
depth++;
for (int i = 0; i < size; i++) {
TreeNode node = queue.poll();
if (node.left != null) queue.offer(node.left);
if (node.right != null) queue.offer(node.right);
}
}
return depth;
}
}
```

4. Output:


leetcode.com/problems/maximum-depth-of-binary-tree/description/

104. Maximum Depth of Binary Tree

Given the `root` of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:

```
Input: root = [3,9,20,null,null,15,7]
Output: 3
```

Example 2:

```
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;
    }
}
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

root =
[3,9,20,null,null,15,7]

Output

3

Expected

3

Maximum Depth

5. Learning Outcomes:

1. Recursive DFS Approach: Use recursion to find the max depth by exploring left and right subtrees.
2. Iterative BFS Approach: Use a queue for level-order traversal to compute depth iteratively.
3. Time Complexity: O(N)O(N) for both DFS and BFS, as all nodes are visited once.

Problem 2

1. Aim:

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

2. Objective:

1. Verify if a given binary tree is a valid Binary Search Tree (BST).
2. Implement recursive (DFS) and iterative (inorder traversal with stack) approaches.
3. Ensure BST properties are maintained using inorder traversal or min-max range validation.

3.Code:

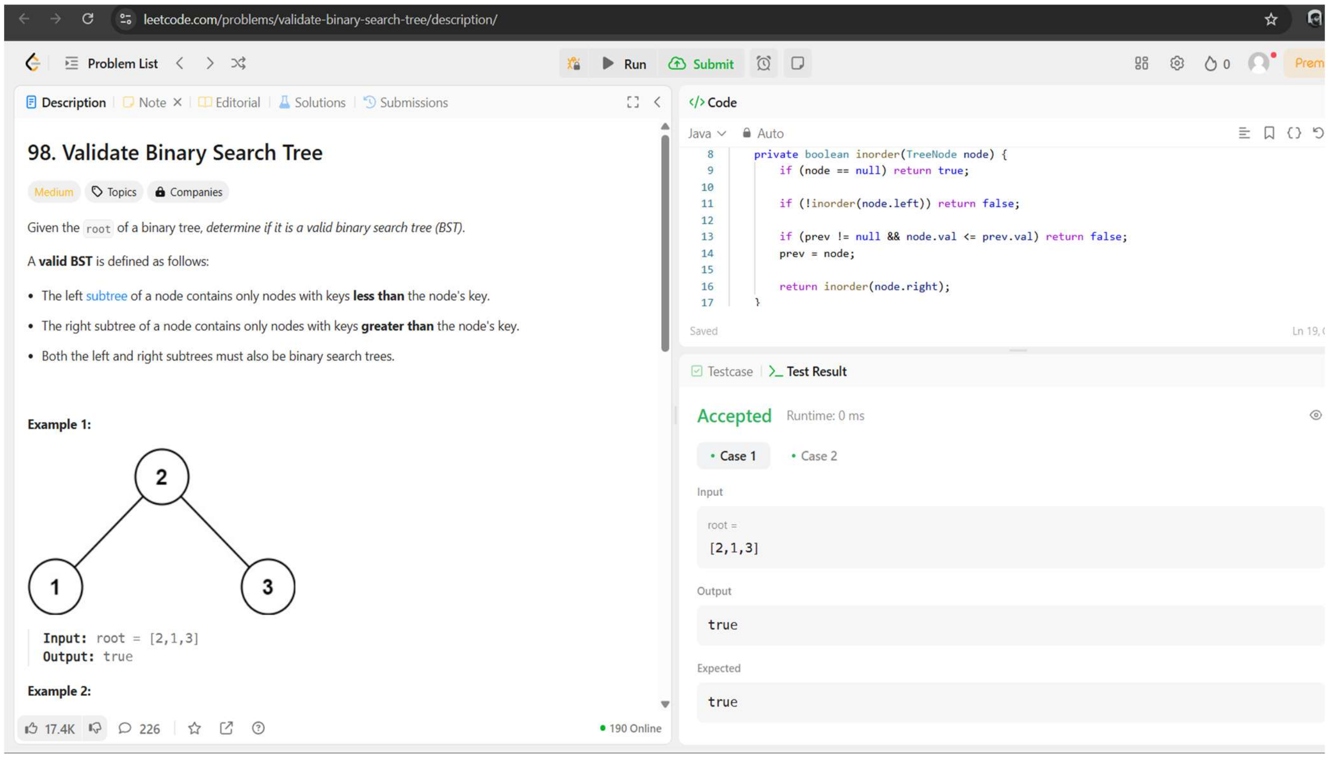
```
import java.util.Stack;

class Solution {

public boolean isValidBST(TreeNode root) {

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

TreeNode prev = null;

while (!stack.isEmpty() || root != null) {

while (root != null) {

stack.push(root);

root = root.left;

}

root = stack.pop();

if (prev != null && root.val <= prev.val) return false;

prev = root;

root = root.right;

}
```

return true;

}

}

4.Output:



leetcode.com/problems/validate-binary-search-tree/description/

98. Validate Binary Search Tree

Given the `root` of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:

```
Input: root = [2,1,3]
Output: true
```

Example 2:

```java
    private boolean inorder(TreeNode node) {
        if (node == null) return true;

        if (!inorder(node.left)) return false;

        if (prev != null && node.val <= prev.val) return false;
        prev = node;

        return inorder(node.right);
    }
```

Saved

Accepted Runtime: 0 ms

Input

```
root =
[2,1,3]
```

Output

```
true
```

Expected

```
true
```

Binary Search Tree

5.Learning Outcomes:

1. BST Property: Left subtree values < root < Right subtree values.
2. Inorder Traversal: A valid BST should produce a strictly increasing sequence.
3. Recursive Approach: Use min-max constraints to validate BST structure.
4. Iterative Approach: Use a stack for inorder traversal to check order without recursion.

Problem 3

1.Aim:

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

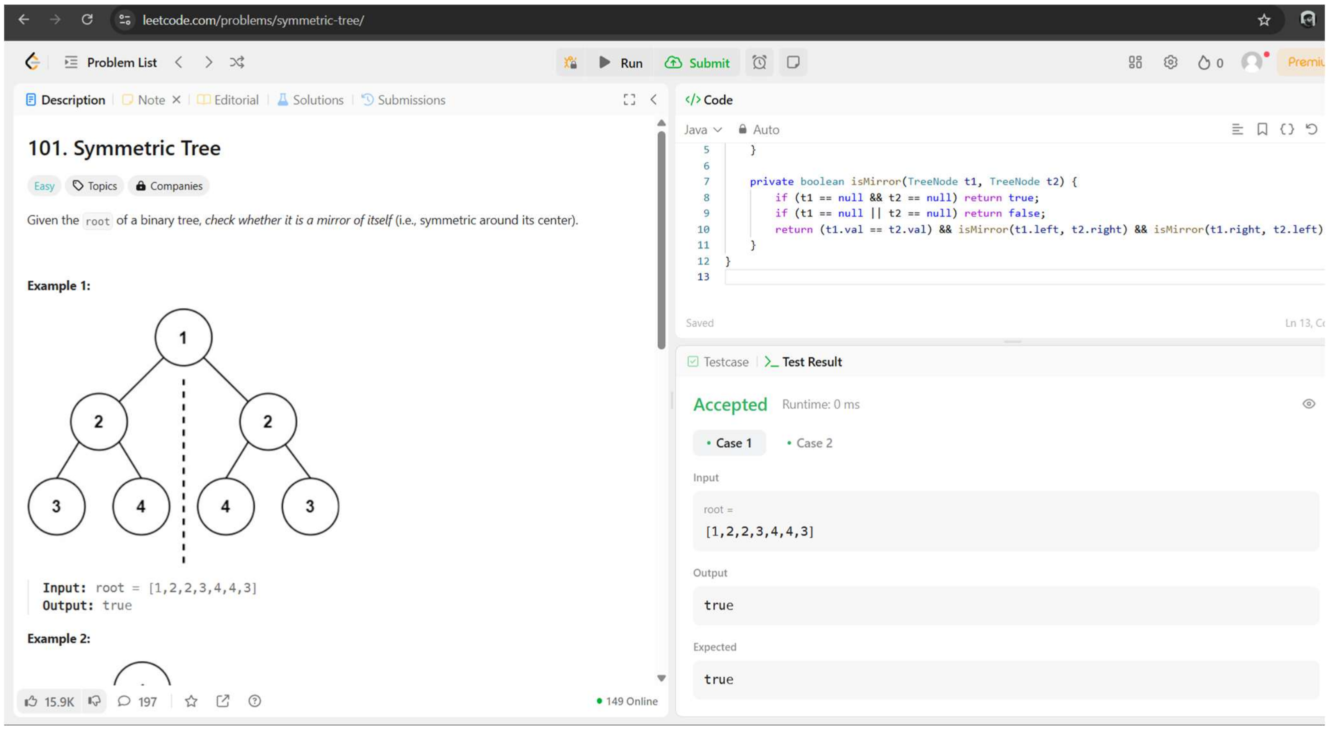
2.Objective:

1. Determine if a given binary tree is symmetric (mirror image of itself).
2. Implement recursive (DFS) and iterative (BFS with queue) approaches.
3. Ensure symmetry by comparing left and right subtrees pairwise.
4. Optimize for time complexity O(N)O(N) and space complexity O(H)O(H) or O(N)O(N).

3.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);

}

}
```

4.Output:



101. Symmetric Tree

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

Example 1:

```
Input: root = [1,2,2,3,4,4,3]
Output: true
```

Example 2:

```
    }

    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)
    }
}
```

Accepted Runtime: 0 ms

```
Input
root =
[1,2,2,3,4,4,3]
Output
true
Expected
true
```

Mirror Binary Tree

5.Learning Outcomes:

1. Symmetric Tree Property: A tree is symmetric if the left subtree is a mirror of the right subtree.
2. Recursive Approach: Use DFS to compare corresponding nodes in mirrored positions.
3. Iterative Approach: Use BFS with a queue to check symmetry level by level.

Problem 4

1.Aim:

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

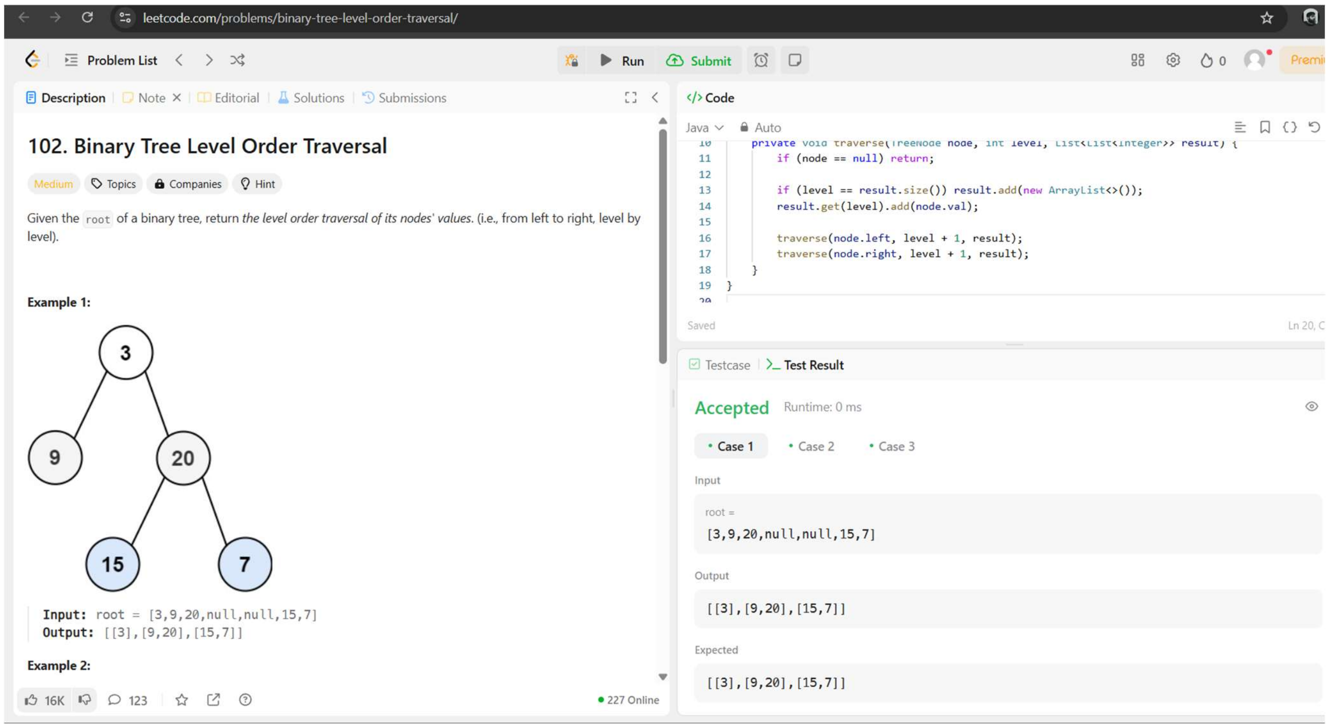
2.Objective:

1. Determine if a given binary tree is symmetric (mirror image of itself).
2. Implement recursive (DFS) and iterative (BFS with queue) approaches.
3. Ensure symmetry by comparing left and right subtrees pairwise.
4. Optimize for time complexity O(N)O(N) and space complexity O(H)O(H) or O(N)O(N).

3.Code:

```
import java.util.*;
class Solution {
public List<List<Integer>> levelOrder(TreeNode root) {
List<List<Integer>> result = new ArrayList<>();
traverse(root, 0, result);
return result;
}
private void traverse(TreeNode node, int level, List<List<Integer>> result) {
if (node == null) return;
if (level == result.size()) result.add(new ArrayList<>());
result.get(level).add(node.val);
traverse(node.left, level + 1, result);
traverse(node.right, level + 1, result);
}
}
```

4.Output:



102. Binary Tree Level Order Traversal

Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Example 1:

```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

Example 2:

```java
    private void traverse(TreeNode node, int level, List<List<Integer>> result) {
        if (node == null) return;

        if (level == result.size()) result.add(new ArrayList<>());
        result.get(level).add(node.val);

        traverse(node.left, level + 1, result);
        traverse(node.right, level + 1, result);
    }
}
```

Saved

Accepted Runtime: 0 ms

Input

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

Output

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

Expected

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

Traversal of nodes

5.Learning Outcomes:

- BFS Approach: Use a queue to process nodes level by level.
- DFS Approach: Use recursion to store values at corresponding depth levels.
- Time Complexity: O(N)O(N)O(N) for both approaches; Space Complexity: O(N)O(N)O(N) for BFS, O(H)O(H)O(H) for DFS.

Problem 5

1.Aim:

Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

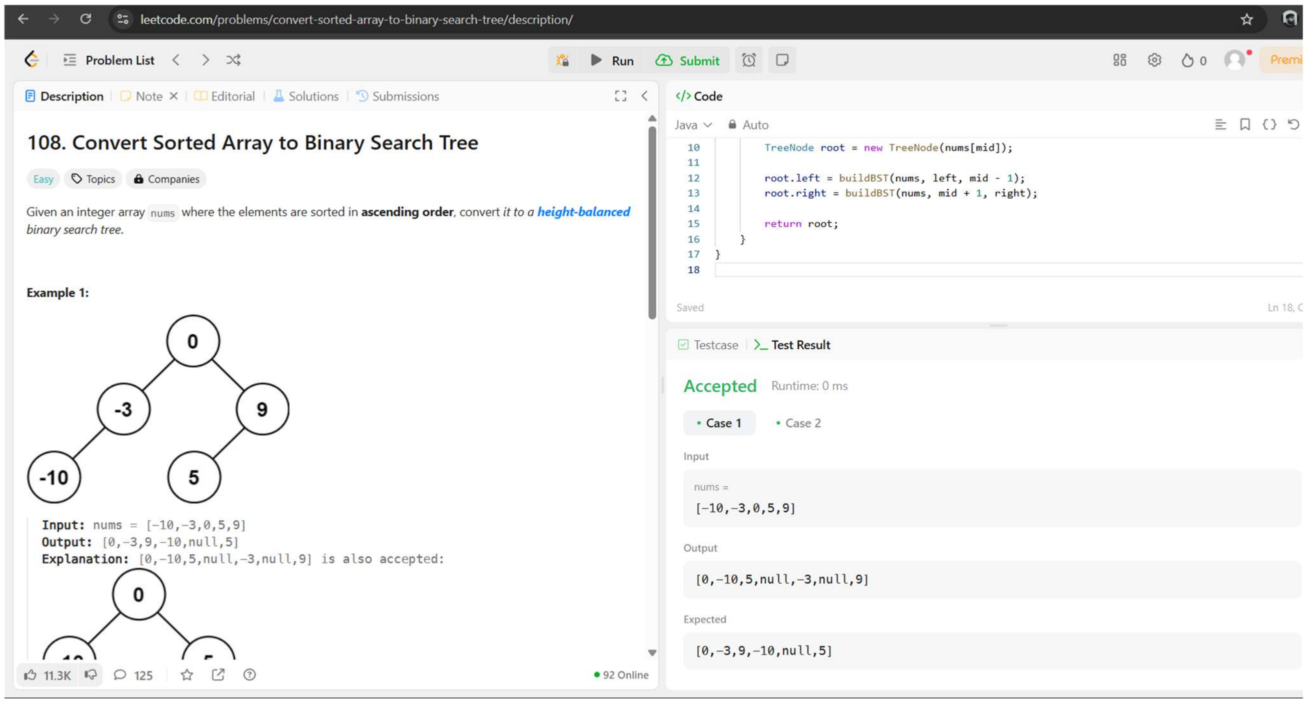
2.Objective:

1. Convert a sorted array into a height-balanced Binary Search Tree (BST).
2. Use a divide & conquer approach to ensure minimal tree height.
3. Implement a recursive solution that selects the middle element as the root.

3.Code:

```
class Solution {
public TreeNode sortedArrayToBST(int[] nums) {
return buildBST(nums, 0, nums.length - 1);
}
private TreeNode buildBST(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 = buildBST(nums, left, mid - 1);
root.right = buildBST(nums, mid + 1, right);

return root;
}
}
```

4.Output:



108. Convert Sorted Array to Binary Search Tree

Easy · Topics · Companies

Given an integer array `nums` where the elements are sorted in **ascending order**, convert *it to a* ***height-balanced*** *binary search tree.*

Example 1:

```
Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: [0,-10,5,null,-3,null,9] is also accepted:
```

```java
        TreeNode root = new TreeNode(nums[mid]);

        root.left = buildBST(nums, left, mid - 1);
        root.right = buildBST(nums, mid + 1, right);

        return root;
    }
}
```

Saved

Accepted Runtime: 0 ms

Input

```
nums =
[-10,-3,0,5,9]
```

Output

```
[0,-10,5,null,-3,null,9]
```

Expected

```
[0,-3,9,-10,null,5]
```

Height Balanced Binary Tree

5.Learning Outcomes:

1. Balanced BST Property: Choosing the middle element ensures a height-balanced structure.
2. Divide & Conquer: Recursively split the array into left and right subtrees.
3. Recursive Approach: Construct the tree top-down, ensuring efficient height management.
4. Time Complexity: O(N)O(N), as each element is processed once.
5. Space Complexity: O(log N)O(\log N), due to recursive function calls (depth of recursion).

Problem 6

1.Aim:

Given the root of a binary tree, return the inorder traversal of its nodes' values.

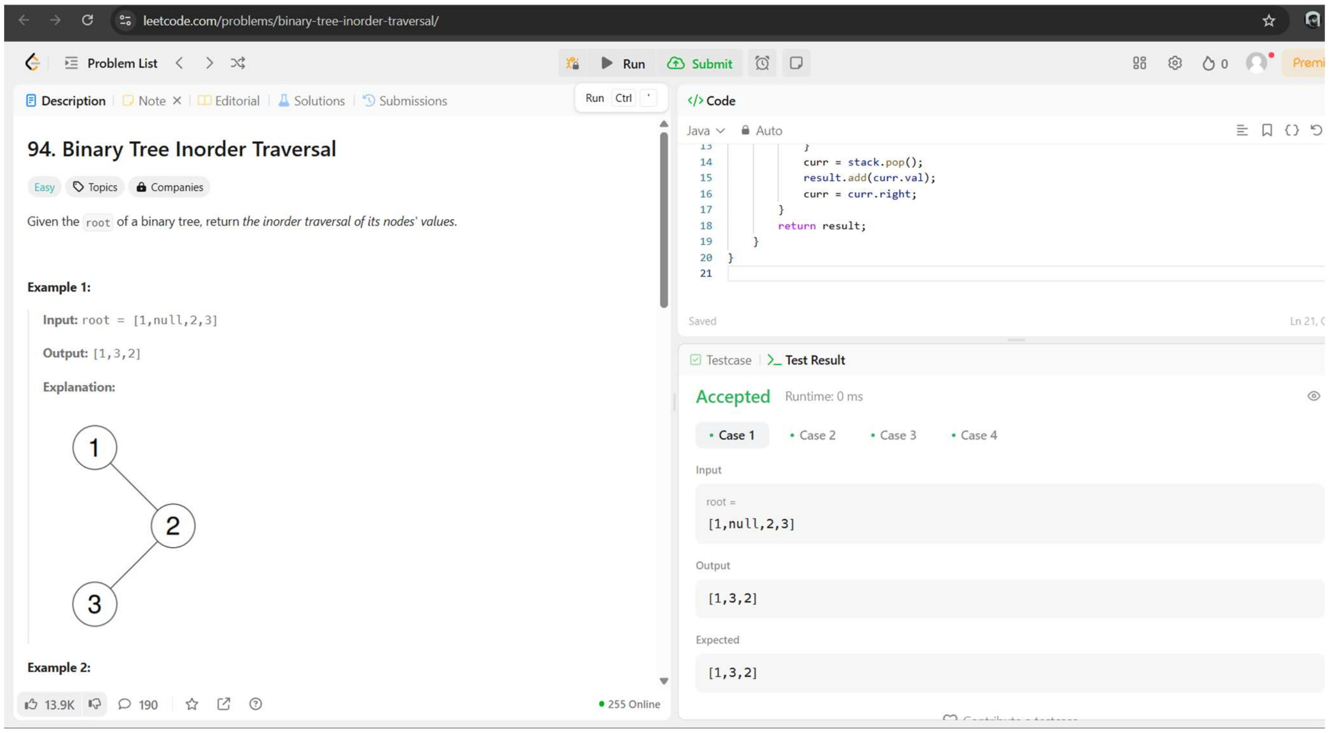
2.Objective:

1. Find a peak element in an array where nums[i] > nums[i-1] and nums[i] > nums[i+1].
2. Implement an efficient algorithm to locate a peak in O(log n) time using Binary Search.
3. Ensure the solution works for various edge cases, including single-element arrays and strictly increasing/decreasing sequences.
4. Optimize space usage by solving the problem in-place with O(1) extra space.

3.Code:

```
import java.util.*;
class Solution {
public List<Integer> inorderTraversal(TreeNode root) {
List<Integer> result = new ArrayList<>();
Stack<TreeNode> stack = new Stack<>();
TreeNode curr = root;

while (curr != null || !stack.isEmpty()) {
while (curr != null) {
stack.push(curr);
curr = curr.left;
}
curr = stack.pop();
result.add(curr.val);
curr = curr.right;
}
return result;
}
}
```

4.Output:



94. Binary Tree Inorder Traversal

Given the `root` of a binary tree, return *the inorder traversal of its nodes' values*.

Example 1:

Input: root = [1,null,2,3]

Output: [1,3,2]

Explanation:

Example 2:

```
            }
            curr = stack.pop();
            result.add(curr.val);
            curr = curr.right;
        }
        return result;
    }
}
```

Saved

Accepted Runtime: 0 ms

Input

```
root =
[1,null,2,3]
```

Output

```
[1,3,2]
```

Expected

```
[1,3,2]
```

Binary Tree Inorder Traversal

5.Learning Outcomes:

1. Inorder Traversal: Follows the order Left → Root → Right.
2. Recursive Approach: Uses DFS with an implicit call stack.
3. Iterative Approach: Uses a stack to manually simulate recursion.

Problem 7

1.Aim:

Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return the binary tree.

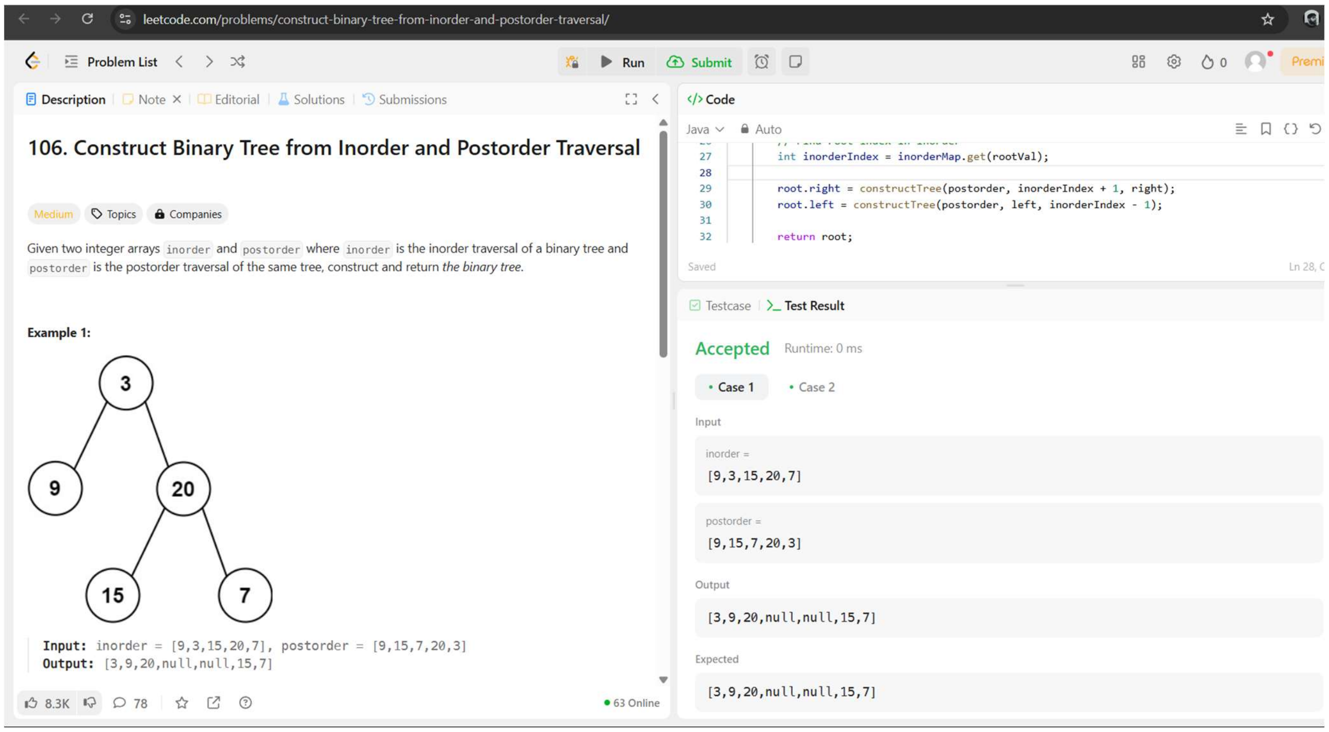
2.Objective:

1. Construct a binary tree from given inorder and postorder traversal arrays.
2. Use postorder traversal to determine the root node.
3. Use inorder traversal to identify left and right subtrees.
4. Implement an efficient recursive approach with HashMap optimization for quick lookups.

3.Code:

```
import java.util.*;
class Solution {
private int postIndex;
private Map<Integer, Integer> inorderMap;
public TreeNode buildTree(int[] inorder, int[] postorder) {
postIndex = postorder.length - 1;
inorderMap = new HashMap<>();
for (int i = 0; i < inorder.length; i++) {
inorderMap.put(inorder[i], i);
}
return constructTree(postorder, 0, inorder.length - 1);
}
private TreeNode constructTree(int[] postorder, int left, int right) {
if (left > right) return null;
int rootVal = postorder[postIndex--];
TreeNode root = new TreeNode(rootVal);
int inorderIndex = inorderMap.get(rootVal);
root.right = constructTree(postorder, inorderIndex + 1, right);
root.left = constructTree(postorder, left, inorderIndex - 1);
return root;
}
}
```

4.Output:



leetcode.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/

106. Construct Binary Tree from Inorder and Postorder Traversal

Given two integer arrays `inorder` and `postorder` where `inorder` is the inorder traversal of a binary tree and `postorder` is the postorder traversal of the same tree, construct and return *the binary tree*.

Example 1:

```
Input: inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]
Output: [3,9,20,null,null,15,7]
```

```java
        int inorderIndex = inorderMap.get(rootVal);

        root.right = constructTree(postorder, inorderIndex + 1, right);
        root.left = constructTree(postorder, left, inorderIndex - 1);

        return root;
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
inorder =
[9,3,15,20,7]
postorder =
[9,15,7,20,3]
```

Output

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

Expected

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

Binary Tree from Inorder & Postorder Traversal

5.Learning Outcomes:

1. Understanding Tree Construction: Learn how to build a binary tree using inorder and postorder traversals.
2. Recursive Problem-Solving: Develop recursive logic to construct subtrees efficiently.
3. Optimizing with HashMap: Use a HashMap for quick lookup of node positions in inorder traversal.
4. Traversal Order Insights: Understand how postorder (left → right → root) helps in identifying roots and inorder (left → root → right) helps in splitting subtrees.

Problem 8

1.Aim:

Given the root of a binary search tree, and an integer k, return the kth smallest value (1-indexed) of all the values of the nodes in the tree.

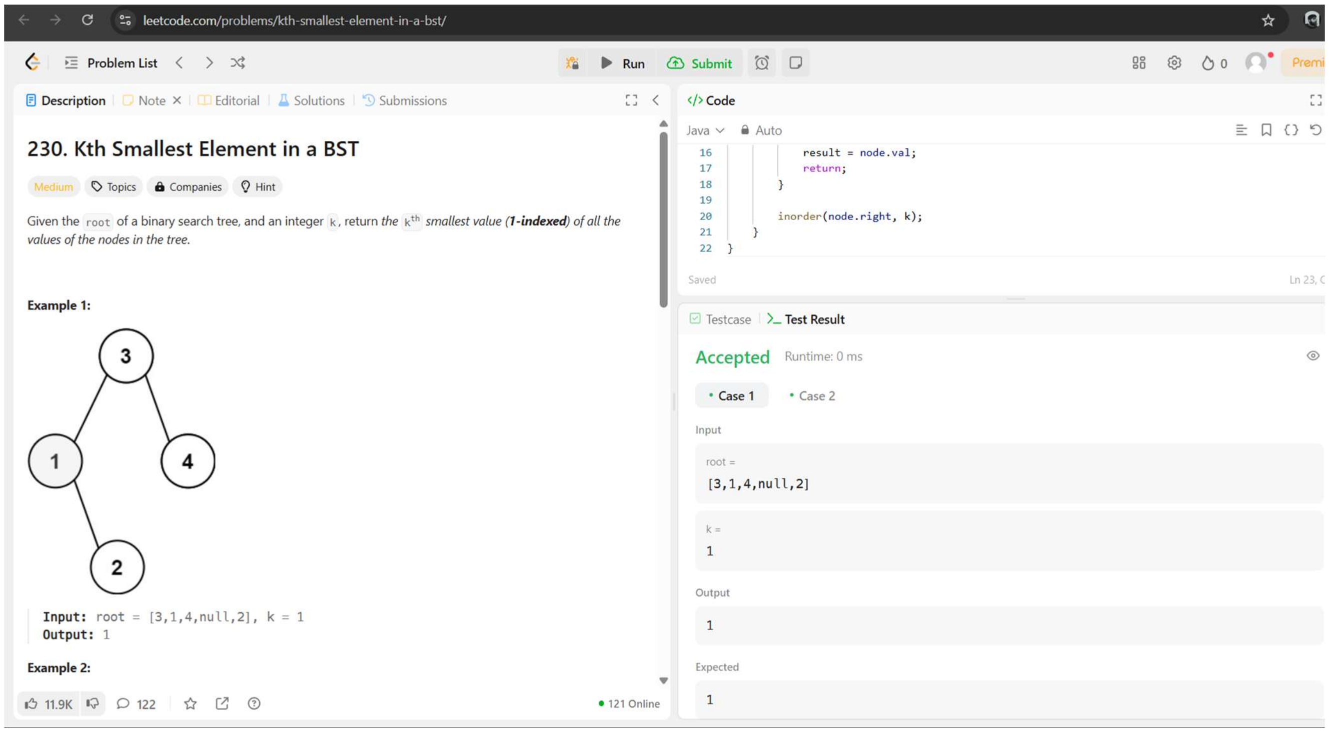
2.Objective:

1. Find the k-th smallest element in a Binary Search Tree (BST).
2. Use inorder traversal (Left → Root → Right) to get elements in sorted order.
3. Implement both recursive (DFS) and iterative (stack-based) approaches.

3.Code:

```
class Solution {
private int count = 0, result = 0
public int kthSmallest(TreeNode root, int k) {
inorder(root, k);
return result;
}
private void inorder(TreeNode node, int k) {
if (node == null) return;

inorder(node.left, k);

count++;
if (count == k) {
result = node.val;
return;
}

inorder(node.right, k);
}
}
```

4.Output:


leetcode.com/problems/kth-smallest-element-in-a-bst/

230. Kth Smallest Element in a BST

Given the `root` of a binary search tree, and an integer `k`, return the k^{th} *smallest value (**1-indexed**) of all the values of the nodes in the tree.*

Example 1:

```
Input: root = [3,1,4,null,2], k = 1
Output: 1
```

Example 2:

```
            result = node.val;
            return;
        }

        inorder(node.right, k);
    }
}
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
root =
[3,1,4,null,2]
```

```
k =
1
```

Output

```
1
```

Expected

```
1
```

Kth Smallest Element in a BST

5.Learning Outcomes:

1. Find the k-th smallest element in a Binary Search Tree (BST).
2. Use inorder traversal (Left → Root → Right) to get elements in sorted order.
3. Implement both recursive (DFS) and iterative (stack-based) approaches.

Problem 9

1.Aim:

You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

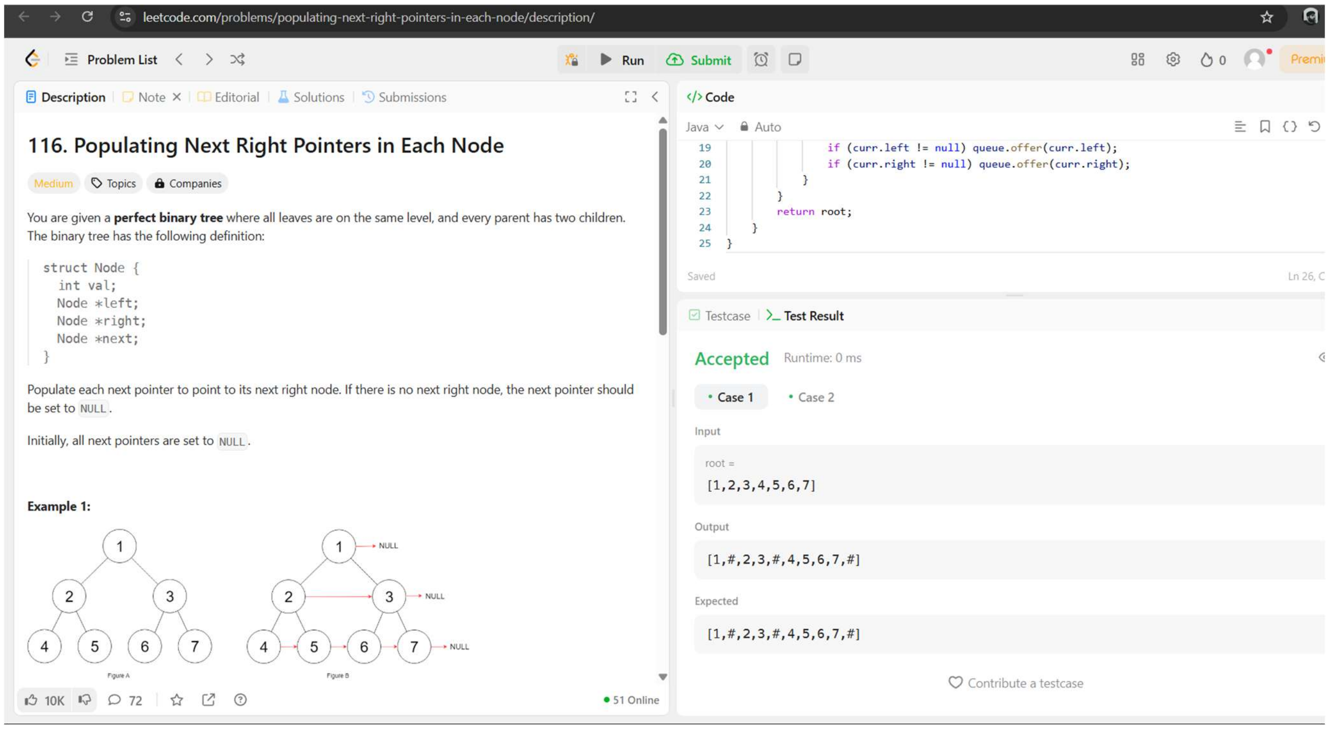
2.Objective:

1. Populate the next right pointers for each node in a perfect binary tree.
2. Use BFS (Queue-based) and O(1) space pointer traversal approaches.
3. Ensure that each node's next pointer connects to its right neighbor at the same level.

3.Code:

```
import java.util.*;
class Solution {
public Node connect(Node root) {
if (root == null) return null;

Queue<Node> queue = new LinkedList<>();
queue.offer(root);
while (!queue.isEmpty()) {
int size = queue.size();
Node prev = null;
for (int i = 0; i < size; i++) {
Node curr = queue.poll();
if (prev != null) prev.next = curr;
prev = curr;

if (curr.left != null) queue.offer(curr.left);
if (curr.right != null) queue.offer(curr.right);
}
}
return root;
}
}
```

4.Output:



leetcode.com/problems/populating-next-right-pointers-in-each-node/description/

Problem List | Run | Submit | Premi

Description | Note | Editorial | Solutions | Submissions

116. Populating Next Right Pointers in Each Node

Medium | Topics | Companies

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Example 1:

Figure A | Figure B

10K | 72 | 51 Online

Code

Java | Auto

```
                if (curr.left != null) queue.offer(curr.left);
                if (curr.right != null) queue.offer(curr.right);
            }
        }
        return root;
    }
}
```

Saved | Ln 26, C

Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 | Case 2

Input

root =

```
[1,2,3,4,5,6,7]
```

Output

```
[1,#,2,3,#,4,5,6,7,#]
```

Expected

```
[1,#,2,3,#,4,5,6,7,#]
```

Contribute a testcase

Populating Next Right Pointers in Each Node

5.Learning Outcomes:

1. Next Pointers in a Perfect Binary Tree: Every node should point to its next right sibling.
2. BFS Approach (Queue-Based): Uses a queue for level-order traversal.
3. O(1) Space Approach: Uses already established next pointers to traverse without extra memory.

Problem 10

1.Aim:

Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and alternate between).

2.Objective:

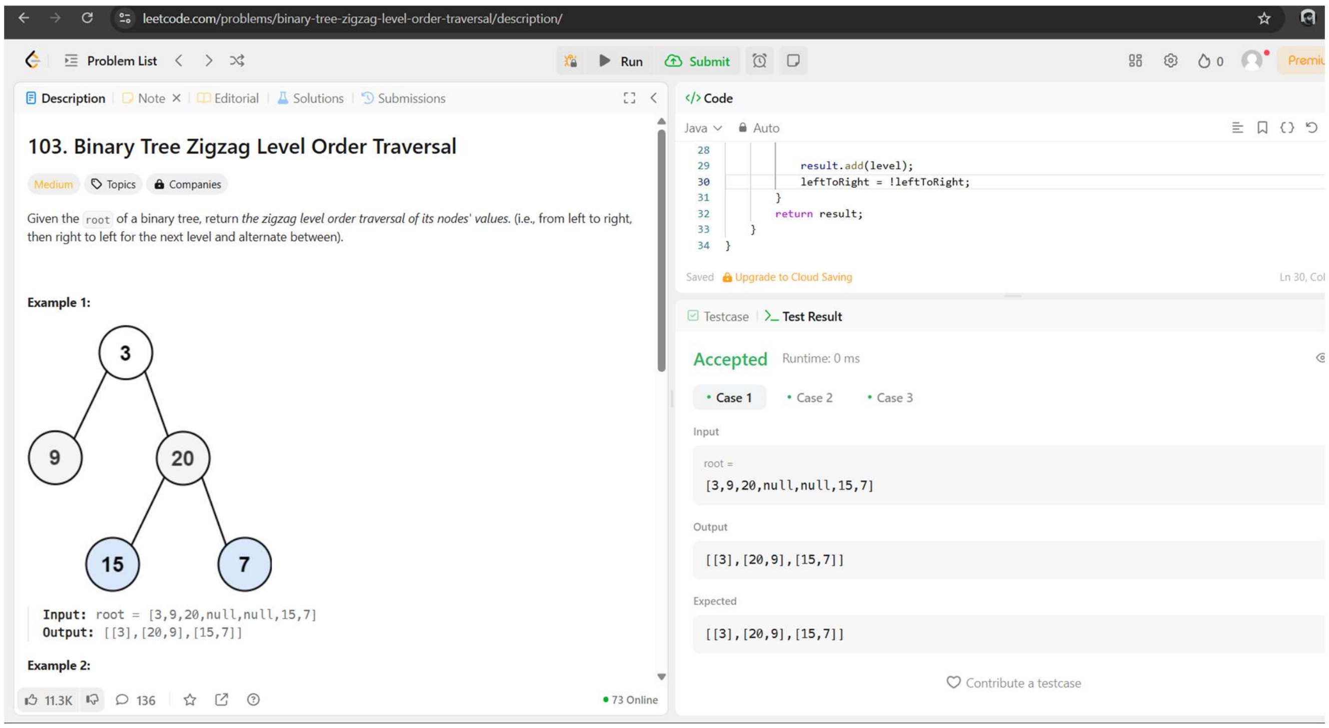
1. Perform a zigzag level order traversal of a binary tree (alternate left-to-right and right-to-left).
2. Use BFS (queue-based level order traversal) to process nodes level by level.
3. Efficiently reverse order using a deque (LinkedList) for O(1) insertions at both ends.

3.Code:

```
import java.util.*;

class Solution {
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
List<List<Integer>> result = new ArrayList<>();
if (root == null) return result;
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);
boolean leftToRight = true;

while (!queue.isEmpty()) {
int size = queue.size();
LinkedList<Integer> level = new LinkedList<>();
for (int i = 0; i < size; i++) {
TreeNode node = queue.poll();
if (leftToRight) {
level.addLast(node.val);
} else {
```

```
level.addFirst(node.val);
}

if (node.left != null) queue.offer(node.left);
if (node.right != null) queue.offer(node.right);
}
result.add(level);
leftToRight = !leftToRight; // Toggle direction for next level
}
return result;
}
}
```

4.Output:



103. Binary Tree Zigzag Level Order Traversal

Given the `root` of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

Example 1:

```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]
```

Example 2:

```
            result.add(level);
            leftToRight = !leftToRight;
        }
        return result;
    }
}
```

Saved Upgrade to Cloud Saving Ln 30, Col

Accepted Runtime: 0 ms

Input

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

Output

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

Expected

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

Zigzag Binary Tree

5.Learning Outcomes:

1. Understanding Zigzag Traversal: Learn how to traverse a binary tree in a left-to-right, then right-to-left order.
2. Using BFS for Level Order Traversal: Utilize a queue to process nodes level by level efficiently.
3. Deque for Order Reversal: Implement O(1) insertions at both ends using LinkedList (Deque).
4. Direction Toggle Technique: Use a boolean flag to switch traversal direction at each level.