



## Experiment- 5

**Student Name:** Ankit Vashisth

**UID:** 22BCS13378

**Branch:** BE-CSE

**Section/Group:** KPIT-901/B

**Semester:** 6<sup>th</sup>

**Date of Performance:** 20/02/25

**Subject Name:** AP Lab - 2

**Subject Code:** 22CSP-351

**1. Aim:** Binary Tree Zigzag level order traversal

### **2. Objective:**

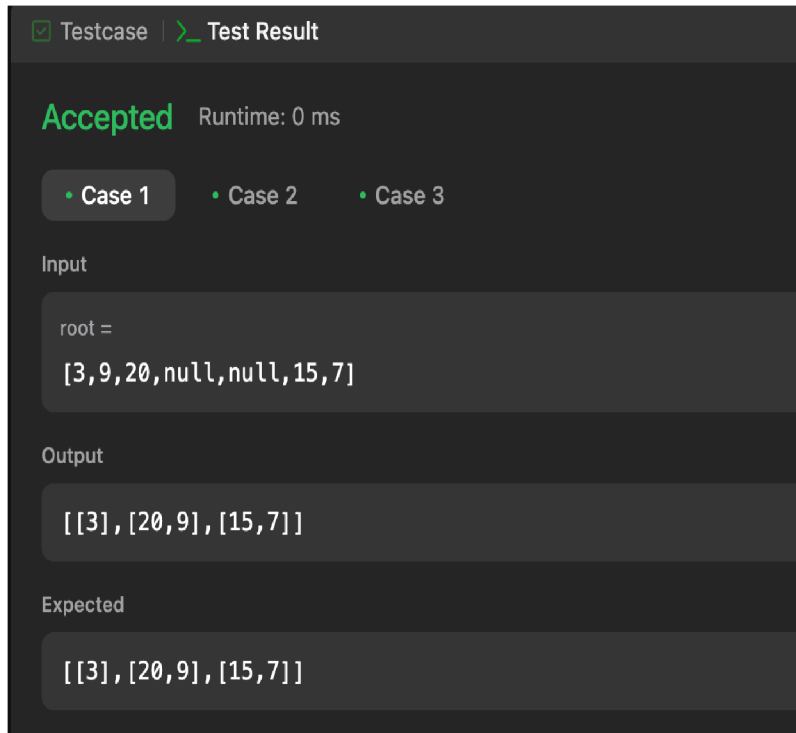
Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and alternate between).

### **3. Implementation/Code:**

```
class Solution { public:
    void solve(vector<vector<int>>& ans, TreeNode* temp, int level) {
        if (temp == NULL) return;        if (ans.size() <= level)
        ans.push_back({});        if (level % 2 == 0)
        ans[level].push_back(temp->val);        else
        ans[level].insert(ans[level].begin(), temp->val);        solve(ans, temp-
        >left, level + 1);        solve(ans, temp->right, level + 1);
    }

    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> ans;        solve(ans, root, 0);
        return ans;
    }
};
```

#### 4. Output :



The screenshot shows a test result interface with a dark background. At the top, there are two tabs: 'Testcase' (selected) and 'Test Result'. Below the tabs, the status 'Accepted' is displayed in green, followed by 'Runtime: 0 ms'. There are three case selection buttons: 'Case 1' (selected), 'Case 2', and 'Case 3'. Under the 'Input' section, the text 'root =' is followed by the array '[3,9,20,null,null,15,7]'. Under the 'Output' section, the array '[ [3], [20,9], [15,7] ]' is shown. Under the 'Expected' section, the same array '[ [3], [20,9], [15,7] ]' is shown, indicating a successful test.

```
Testcase | Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

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

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

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

#### 5. Learning Outcome:

1. Understand the traversal process of a binary tree in a zigzag manner.
2. Learn how to efficiently manipulate data structures (vectors) while performing level-order traversal.
3. Implement an optimal traversal approach using recursion and alternating insertions.
4. Analyze the time complexity of the traversal process, which runs in  $O(N)$  time.

### QUESTION 2

1. **Aim:** Construct Binary Tree from Inorder and Postorder Traversal

2. **Objective:**

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.

3. **Implementation/Code:**

```
class Solution { public:
    TreeNode* solve(vector<int>&inorder, int InStart, int InEnd, vector<int>& postorder,
    int PostStart, int PostEnd, map<int, int>&mpp){    if(InStart>InEnd || PostStart>
    PostEnd) return NULL;

        TreeNode* node = new TreeNode(postorder[PostEnd]);

        int node_position = mpp[node->val];

        int leftLen = node_position - InStart;

        node->left = solve(inorder,InStart, node_position-1, postorder, PostStart, PostStart +
        leftLen-1, mpp);
        node->right = solve(inorder,node_position+1, InEnd, postorder, PostStart+leftLen,
        PostEnd-1, mpp);
        return node; }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        map<int, int>mpp;
        for(int i=0;i<inorder.size();++i){
            mpp[inorder[i]] = i;
        }

        TreeNode* root = solve(inorder, 0, inorder.size()-1, postorder, 0, postorder.size()-1,
        mpp);
        return root;
    }
};
```

#### 4. Output:

```
Testcase | Test Result
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]
```

#### 5. Learning Outcome:

1. Understand the concept of constructing a binary tree using inorder and postorder traversals.
2. Learn how to efficiently find root positions using a hash map for quick lookups.
3. Implement a recursive approach to build the tree while maintaining correct left and right subtree boundaries.
4. Analyze the time complexity of  $O(N)$  due to optimized hash map lookups for element positioning.