**Binary Tree Postorder Traversal - Explanation & Dry Run**

**Problem Statement**

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

**Postorder Traversal Rule:**

- Traverse the **left** subtree.
- Traverse the **right** subtree.
- Visit the **root** node.

---

**Code Explanation**

**C++ Implementation**

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    void helper(TreeNode* root, vector<int> &ans) {
        if (root == NULL) return; // Base case
        helper(root->left, ans);  // Traverse left subtree
        helper(root->right, ans); // Traverse right subtree
        ans.push_back(root->val); // Visit the root
    }

    vector<int> postorderTraversal(TreeNode* root) {
```

```
        vector<int> ans;
        helper(root, ans);
        return ans;
    }
};
```

**Breakdown of Code:**

1. **helper Function:**
   - Recursively visits each node in **postorder sequence**.
   - Base case: If root == NULL, return immediately.
   - First calls itself recursively for **left** subtree.
   - Then calls itself recursively for **right** subtree.
   - Finally, adds root->val to the result list (ans).
2. **postorderTraversal Function:**
   - Creates an empty vector ans to store the traversal.
   - Calls the helper function with the root node.
   - Returns the final result.

---

**Dry Run with Examples**

**Example 1:**

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

**Tree Structure:**

```
  1
   \
    2
   /
  3
```

**Recursive Calls:**

1. helper(1) → helper(NULL) → ans = []
2. helper(2) → helper(3) → ans = []
3. ans.push_back(3) → ans = [3]
4. ans.push_back(2) → ans = [3, 2]

5. ans.push_back(1) → ans = [3, 2, 1]

**Output: [3, 2, 1]**

---

**Example 2:**

**Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]**

**Tree Structure:**

```
    1
   / \
  2   3
 / \   \
4   5   8
   / \   \
  6   7   9
```

**Recursive Calls:**

1. helper(4) → ans = [4]
2. helper(6) → ans = [4, 6]
3. helper(7) → ans = [4, 6, 7]
4. helper(5) → ans = [4, 6, 7, 5]
5. helper(2) → ans = [4, 6, 7, 5, 2]
6. helper(9) → ans = [4, 6, 7, 5, 2, 9]
7. helper(8) → ans = [4, 6, 7, 5, 2, 9, 8]
8. helper(3) → ans = [4, 6, 7, 5, 2, 9, 8, 3]
9. helper(1) → ans = [4, 6, 7, 5, 2, 9, 8, 3, 1]

**Output: [4, 6, 7, 5, 2, 9, 8, 3, 1]**

---

**Edge Cases**

1. **Empty Tree:**
    - **Input:** root = []
    - **Output:** []
2. **Single Node:**
    - **Input:** root = [1]

- **Output:** [1]

---

**Time & Space Complexity**

- **Time Complexity:** O(N), as every node is visited once.
- **Space Complexity:** O(N) (worst case for recursion stack if the tree is skewed).

---

**Follow-up: Iterative Solution (Using Stack)**

To solve the problem iteratively, we can use a stack:

```
vector<int> postorderTraversal(TreeNode* root) {
   vector<int> ans;
   if (!root) return ans;
   stack<TreeNode*> st;
   TreeNode* last = NULL;
   TreeNode* curr = root;

   while (!st.empty() || curr) {
      if (curr) {
         st.push(curr);
         curr = curr->left;
      } else {
         TreeNode* node = st.top();
         if (node->right && node->right != last) {
            curr = node->right;
         } else {
            ans.push_back(node->val);
            last = node;
            st.pop();
         }
      }
   }
   return ans;
```

}

**Advantages of Iterative Approach:**

- **Avoids recursion depth issues** (useful for deep trees).
- **More memory-efficient in some cases**.

---

**Conclusion**

- We explored a recursive solution for **Postorder Traversal** of a binary tree.
- Dry-ran multiple test cases to understand recursive calls.
- Discussed an **iterative stack-based** approach as an alternative.
- **Postorder traversal follows the sequence: Left → Right → Root.**

This concludes our explanation of **Binary Tree Postorder Traversal**!