Binary Tree Preorder Traversal - Explanation & Dry Run

Problem Statement

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

Preorder Traversal Rule:

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

Code Explanation

C++ Implementation

```
class Solution {
public:
   void helper(TreeNode* root, vector<int> &ans) {
      if (root == NULL) return; // Base case
      ans.push_back(root->val); // Visit the root
      helper(root->left, ans);  // Traverse left subtree
      helper(root->right, ans); // Traverse right subtree
   }

   vector<int> preorderTraversal(TreeNode* root) {
      vector<int> ans;
      helper(root, ans);
      return ans;
   }
};
```

Breakdown of Code:

1. **helper Function:**
   - Recursively visits each node in **preorder sequence**.
   - Base case: If root == NULL, return immediately.
   - Adds root->val to the result list (ans).
   - Calls itself recursively for **left** and then **right** subtree.

2. **preorderTraversal 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) → ans = [1]
2. helper(2) → ans = [1, 2]
3. helper(3) → ans = [1, 2, 3]
4. Base case reached (NULL nodes return).

Output: [1, 2, 3]

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

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

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> preorderTraversal(TreeNode* root) {
   vector<int> ans;
   if (root == NULL) return ans;
   stack<TreeNode*> st;
   st.push(root);
```

```
    while (!st.empty()) {
        TreeNode* node = st.top();
        st.pop();
        ans.push_back(node->val);

        if (node->right) st.push(node->right); // Push right first
        if (node->left) st.push(node->left);   // Push left second
    }
    return ans;
}
```

Advantages of Iterative Approach:

- **Avoids recursion depth issues** (useful for deep trees).
- **Faster execution in some cases due to reduced function call overhead.**

Conclusion

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

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