Binary Tree Paths - Revision Notes

Problem Statement

Given the root of a binary tree, return all root-to-leaf paths in any order.

A **leaf** is a node with no children.

Example

Example 1

Input:

```
  1
 / \
2   3
 \
  5
```

root = [1,2,3,null,5]
Output: ["1->2->5", "1->3"]

Example 2

Input: root = [1]
Output: ["1"]

Logic and Approach

Understanding the Problem

- We need to find all root-to-leaf paths in the given binary tree.
- A **root-to-leaf path** means starting from the root and going to a leaf node (a node with no children).
- The path should be stored as a **string**, with values separated by "->".

Approach (Depth-First Search - DFS)

We use **DFS (Depth-First Search)** to traverse the tree:

1. Maintain a string s that stores the current path.
2. If we reach a **leaf node**, store the complete path in the answer.
3. Recursively explore left and right children, appending "->" between nodes.

Code Implementation

class Solution {

```
public:
  void helper(TreeNode* root, string s, vector<string>& ans) {
    if (root == NULL) return; // Base case: If node is NULL, return

    string a = to_string(root->val); // Convert node value to string

    // If it's a leaf node, store the complete path
    if (root->left == NULL && root->right == NULL) {
      s += a;
      ans.push_back(s);
      return;
    }

    // Recursive calls for left and right children
    helper(root->left, s + a + "->", ans);
    helper(root->right, s + a + "->", ans);
  }

  vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> ans;
    helper(root, "", ans);
    return ans;
  }
};
```

Code Breakdown

Helper Function (helper)

- **Base Case:** If root == NULL, return.
- Convert root->val to string.
- **If leaf node is reached:**
  - Append the node value to s.

    - Store the final path in ans.
- **Recursive Calls:**
    - Call helper for the left child with s + a + "->".
    - Call helper for the right child with s + a + "->".

BinaryTreePaths Function

- Calls the helper function with an empty string and stores all paths in ans.
- Returns ans as the final result.

Time and Space Complexity Analysis

- **Time Complexity:** O(N) (each node is visited once)
- **Space Complexity:** O(H) (recursion stack height, worst case O(N) for skewed trees)

Conclusion

- We used **DFS recursion** to generate all root-to-leaf paths.
- The approach efficiently constructs paths and stores them in a vector.
- Works within given constraints (1 \leq N \leq 100).
- **Key learning:** DFS traversal and handling strings dynamically.

🔥 **Great for practicing recursion and tree traversal!** 🚀