1123. Lowest Common Ancestor of Deepest Leaves

Medium · Tree · Depth-First Search · Breadth-First Search · Hash Table · Binary Tree · Leetcode Link

Problem Description

In this problem, we're given the root of a binary tree. Our goal is to find the lowest common ancestor (LCA) of its deepest leaves. We are considering the following:

- A leaf node is defined as a node with no children.
- The root node has a depth of 0. If a node is at depth `d`, then its children's depth is `d + 1`.
- The lowest common ancestor for a set of nodes`S` is the deepest node `A` such that every node from `S` is in the subtree rooted with `A`.

We need to determine this common ancestor and return it.

Intuition

The solution approach is a depth-first search (DFS) algorithm that proceeds as follows:

1. Starting from the root, we perform DFS to explore the tree.
2. Each step of the DFS returns two values: the possible LCA node at this point and the depth of the deepest leaf node in the current node's subtree.
3. We compare the depths of left and right subtrees.
4. The current node could be:
   - The LCA if both left and right subtree depths are equal, meaning they both contain leaves of the deepest level.
   - Not the LCA if one subtree is deeper than the other. In this case, the potential LCA is in the deeper subtree.
5. The depth is incremented as we return back up the tree.
6. Once the DFS is complete, the first element of the return value from the DFS initiated at the root will be the LCA of the deepest leaves.

By following this approach, the solution effectively finds the deepest level of the tree and then tracks back up the tree to find the lowest common ancestor of the nodes at this level.

Solution Approach

The provided solution uses a recursive depth-first search (DFS) strategy for traversing the binary tree, which we implement in the `dfs` helper function. Here's a step-by-step approach to how the algorithm works:

1. The `dfs` function is called recursively for each node starting with the `root`. This function returns two things:
   - The potential LCA node at the current subtree.
   - The depth of the deepest leaf in the current subtree.
2. On each call of the `dfs` function:
   - We check if the current node is `None` (base case):
     - If it is, we return `None` and a depth of `0`.
   - Otherwise, we recursively call `dfs` on the left child and right child.
3. Each recursive call will provide us with:
   - The potential LCA nodes `l` and `r` from the left and right subtrees, respectively.
   - The depths `d1` and `d2` representing the maximum depths in those subtrees.
4. We then compare the depths of the deepest leaves in the left and right subtrees:
   - If `d1 > d2`, the left subtree is deeper, so we return `l` (the left child's LCA) and `d1 + 1` (the new depth).
   - If `d1 < d2`, the right subtree is deeper, so we return `r` (the right child's LCA) and `d2 + 1`.
   - If `d1 == d2`, both left and right subtrees have leaves at the same depth, hence, the current `root` node is their LCA, and we return `root` and either `d1 + 1` or `d2 + 1` (as they are equal).
5. At the top level of the recursion, we call `dfs(root)` and are interested only in the first item of the tuple, which represents the lowest common ancestor of the deepest leaves.

This approach leverages the nature of DFS to explore and evaluate potential LCA nodes at varying depths of the tree, effectively utilizing recursion and tuple-unpacking to concisely express the critical decision logic within a binary tree traversal algorithm.

Example Walkthrough

Let's consider a simple binary tree to illustrate the solution approach:

```
        A
       / \
      B   C
     / \
    D   E
   /     \
  F       G
```

In this tree, the deepest leaves are F and G, both at depth 3 from the root A. We wish to find their lowest common ancestor.

1. Begin the DFS with the root node A.
2. Call `dfs(A)`, which proceeds to its children:
   - Call `dfs(B)`.
     - Call `dfs(D)`:
       - Call `dfs(F)`:
         - Reached a leaf node, return `(F, 1)` (node, depth).
     - `dfs(D)` returns `(F, 1)` and now we check `D`'s right branch.
       - Call `dfs(E)`.
         - Call `dfs(None)` on left and returns `(None, 0)`.
         - Call `dfs(G)` on right:
           - Reached a leaf node, return `(G, 1)`.
         - `dfs(E)` compares depths `0` and `1`, and returns `(G, 2)`.
     - Now `dfs(B)` compares the info from `D` and `E`. `1` and `2` are not the same, so it takes the larger (from `E`), and returns `(G, 3)`.
   - `dfs(A)` now needs to check the right subtree with `dfs(C)`:
     - Call `dfs(C)` on left and right and returns `(None, 1)` for both as `C` is a leaf node.
3. Back to `dfs(A)`, we now compare the results from `B` and `C`, which are `(G, 3)` and `(C, 1)`.
   - We see the left subtree has a greater depth, so we take its LCA (node G) and increment the depth for return, which becomes `(G, 4)`.
4. Since `A` is the top-level call, we return the first element, G, the LCA of the deepest leaves (F and G).

So the lowest common ancestor of the deepest leaves F and G is node E. However, notice that E is not the root; hence, the algorithm will correctly identify A as the actual LCA. The reason is that A is the lowest common ancestor that also contains E and hence F and G, which are the deepest leaves.

Python Solution

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def lcaDeepestLeaves(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        # Helper function to perform a depth-first search on the tree.
        def dfs(node):
            # Base case: if the current node is None, it corresponds to a depth of 0.
            if node is None:
                return None, 0

            # Recursively find the lowest common ancestor and depth of the left subtree.
            left_lca, left_depth = dfs(node.left)
            # Recursively find the lowest common ancestor and depth of the right subtree.
            right_lca, right_depth = dfs(node.right)

            # If the left subtree is deeper, return the left LCA and its depth increased by one.
            if left_depth > right_depth:
                return left_lca, left_depth + 1
            # If the right subtree is deeper, return the right LCA and its depth increased by one.
            if left_depth < right_depth:
                return right_lca, right_depth + 1

            # If both subtrees have the same depth, then this node is the lowest common ancestor.
            # Return the current node and the depth of the subtree.
            return node, left_depth + 1

        # Call the DFS helper function and return the lowest common ancestor. The second value of the tuple is ignored.
        return dfs(root)[0]

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
public class TreeNode {
     int val;
     TreeNode left;
     TreeNode right;
     TreeNode() {}
     TreeNode(int val) { this.val = val; }
     TreeNode(int val, TreeNode left, TreeNode right) {
         this.val = val;
         this.left = left;
         this.right = right;
     }
}

class Solution {
    /**
     * Finds the lowest common ancestor (LCA) of the deepest leaves in a binary tree.
     *
     * @param root the root of the binary tree.
     * @return the TreeNode representing the LCA of the deepest leaves.
     */
    public TreeNode lcaDeepestLeaves(TreeNode root) {
        return depthFirstSearch(root).getKey();
    }

    /**
     * Helper method to perform depth-first search to find the LCA of the deepest leaves.
     *
     * @param node the current node under consideration.
     * @return a Pair containing the current LCA node and the depth of the subtree rooted at the node.
     */
    private Pair<TreeNode, Integer> depthFirstSearch(TreeNode node) {
        if (node == null) {
            // Base case: if the current node is null, return a pair of (null, 0)
            return new Pair<>(null, 0);
        }
        // Recursively find the depth and LCA in the left subtree
        Pair<TreeNode, Integer> leftPair = depthFirstSearch(node.left);
        // Recursively find the depth and LCA in the right subtree
        Pair<TreeNode, Integer> rightPair = depthFirstSearch(node.right);

        int leftDepth = leftPair.getValue(), rightDepth = rightPair.getValue();

        if (leftDepth > rightDepth) {
            // If the left subtree is deeper, return the LCA and depth of the left subtree
            return new Pair<>(leftPair.getKey(), leftDepth + 1);
        }
        if (leftDepth < rightDepth) {
            // If the right subtree is deeper, return the LCA and depth of the right subtree
            return new Pair<>(rightPair.getKey(), rightDepth + 1);
        }
        // If both subtrees have the same depth, the current node is the LCA
        return new Pair<>(node, leftDepth + 1);
    }
}

/**
 * A helper class to store a pair of objects.
 *
 * @param <K> the type of the first element.
 * @param <V> the type of the second element.
 */
class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}

```

C++ Solution

```cpp
// 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:
    // Function to find the lowest common ancestor of the deepest leaves.
    TreeNode* lcaDeepestLeaves(TreeNode* root) {
        return depthFirstSearch(root).first;
    }

    // Helper function to perform depth-first search.
    // It will return a pair consisting of the lowest common ancestor at the current subtree and the depth of the deepest leaves.
    pair<TreeNode*, int> depthFirstSearch(TreeNode* node) {
        if (!node) {
            // If the node is null, return a pair of nullptr and depth 0.
            return {nullptr, 0};
        }

        // Recursively look for deepest leaves in the left and right subtrees.
        auto [leftSubtreeLCA, leftDepth] = depthFirstSearch(node->left);
        auto [rightSubtreeLCA, rightDepth] = depthFirstSearch(node->right);

        if (leftDepth > rightDepth) {
            // If the left subtree is deeper, return the left subtree's LCA and depth.
            return {leftSubtreeLCA, leftDepth + 1};
        } else if (leftDepth < rightDepth) {
            // If the right subtree is deeper, return the right subtree's LCA and depth.
            return {rightSubtreeLCA, rightDepth + 1};
        } else {
            // If both subtrees have the same depth, return the current node as the LCA, as both its left and right subtree have th
            return {node, leftDepth + 1};
        }
    }
};

```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = val === undefined ? 0 : val;
        this.left = left === undefined ? null : left;
        this.right = right === undefined ? null : right;
    }
}

// Finds the lowest common ancestor of the deepest leaves in a binary tree.
function lcaDeepestLeaves(root: TreeNode | null): TreeNode | null {
    // A depth-first search function that returns both the potential lowest common ancestor and the depth.
    const depthFirstSearch = (node: TreeNode | null): [TreeNode | null, number] => {
        // If the current node is null, return null and depth 0.
        if (node === null) {
            return [null, 0];
        }
        // Recursively find the left and right children's deepest nodes and depths.
        const [leftAncestor, leftDepth] = depthFirstSearch(node.left);
        const [rightAncestor, rightDepth] = depthFirstSearch(node.right);

        // If the left subtree is deeper, return the left child's ancestor and increase the depth by 1.
        if (leftDepth > rightDepth) {
            return [leftAncestor, leftDepth + 1];
        }
        // If the right subtree is deeper, return the right child's ancestor and increase the depth by 1.
        if (leftDepth < rightDepth) {
            return [rightAncestor, rightDepth + 1];
        }
        // If both subtrees have the same depth, the current node is their lowest common ancestor, depth is increased by 1.
        return [node, leftDepth + 1];
    };

    // Start the depth-first search from the root and return the lowest common ancestor.
    return depthFirstSearch(root)[0];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(N)` where `N` is the number of nodes in the tree. This is because the depth-first search (`dfs`) function visits every node exactly once to calculate the deepest leaf nodes.

Space Complexity

The space complexity of the code is `O(H)` where `H` is the height of the tree. This space is used by the recursion stack of the depth-first search. In the worst case (a skewed tree), the space complexity can be `O(N)`.