865. Smallest Subtree with all the Deepest Nodes

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

Problem Description

The problem requires us to find the smallest subtree in a given binary tree which includes all the deepest nodes. The definition of the depth of a node is the distance from the node to the root of the tree. The deepest nodes are the ones that are the farthest from the root, i.e., they have the highest depth. The subtree must include the node and all of its descendants.

Intuition

To solve this problem, we can utilize a depth-first search (DFS) to navigate through the tree. The key is to identify the depth of the deepest nodes and to find the lowest common ancestor (LCA) of all the deepest nodes. Here's the intuition behind the solution:

1. Start at the root and traverse the tree using DFS.
2. For each node, calculate the depth of its left and right child recursively.
3. Compare the depth of the left and right children:
   - If one child is deeper, return that child's node and its depth.
   - If both children are at the same depth, it means this node is the LCA of the deepest nodes (as it's common to all of them), so return the current node and its depth.
4. The base case for the recursion occurs when we reach a leaf node (no children), where we return the node itself and depth as 0.
5. Continue this process until all nodes have been visited.
6. The result of the entire function will be the LCA node where the depth of the left and right subtrees are equal and highest indicating it's the smallest subtree which contains all the deepest nodes.

Using this approach, the `dfs` function traverses the tree and effectively finds the depths of all nodes, bubbling up the necessary information to identify both the depth of the deepest nodes and the LCA, which is the root of the smallest subtree containing all the deepest nodes.

Solution Approach

The implementation of the solution uses a recursive depth-first search (DFS) algorithm to navigate through the binary tree and determine the depths of all nodes, as well as to find the subtree that contains all the deepest nodes.

Here is the step-by-step approach:

1. **DFS Function `dfs(root)`:** The helper function `dfs` is defined within the `subtreeWithAllDeepest` method. It takes a node (initially the root) as an argument and returns two values: the node itself and the depth of that node. It operates recursively until it hits the base case, which is when `root is None`. At this point, it returns `None` for the node and `0` for the depth.

2. **Left and Right Recursion Calls:** Within the `dfs` function, it first calls itself on `root.left` to explore the left subtree and `root.right` to explore the right subtree. Each call returns a tuple containing a node and its respective depth.

3. **Comparison of Depths:**

   - If the left subtree's depth is greater than the right subtree's depth, the left child node is part of the subtree containing all deepest nodes, and therefore returns the left child node and its depth incremented by 1 (to account for the current node's depth).
   - If the right subtree's depth is greater than the left subtree's depth, the right child node is part of the subtree containing all deepest nodes, and it returns the right child node and its depth incremented by 1.
   - If both depths are equal, it means that the current node is the lowest common ancestor (LCA) of the deepest nodes, as it is at the same distance from the deepest nodes of both subtrees. In this case, it returns the current node and one of the depths incremented by 1.
4. **Returning Result:** After traversing the entire tree and applying the above logic at each node, the wrapper function `subtreeWithAllDeepest` returns the first element of the tuple returned by `dfs(root)`, which is the LCA node of all deepest nodes, hence the root of the smallest subtree containing all the deepest nodes.

By following these steps, the solution provided efficiently uses DFS and concepts of tree depth and Lowest Common Ancestor to arrive at the smallest subtree encompassing all deepest nodes of the binary tree.

Example Walkthrough

Let's consider a small binary tree for this walkthrough to illustrate the solution approach:

```
    3
   / \
  9  20
    /  \
   15   7
         \
          4
```

In this tree, the deepest nodes are 9, 15, and 4, all with a depth of 2. We need to find the smallest subtree that includes all these nodes. Following the solution approach:

1. **Start DFS with Root (Node 3):**

   - `dfs(3)` is called.
   - Inside `dfs`, recursive calls will be made for `dfs(9)` and `dfs(20)`.
2. **DFS on Left Subtree (Node 9):**

   - `dfs(9)` is a leaf node; thus, it returns `(9, 1)`.
3. **DFS on Right Subtree (Node 20):**

   - Calls are made to `dfs(15)` and `dfs(7)`.
4. **DFS on Node 15:**

   - It is a leaf node; thus, it returns `(15, 1)`.
5. **DFS on Node 7:**

   - Calls are made to its left child (which is `None`) and `dfs(4)`.
   - `dfs(4)` is a leaf node; thus, it returns `(4, 1)`.
6. **Comparison of Depths of Children of Node 7:**

   - Since node 7's left child is `None` and the right child has a depth of 1, it returns `(4, 2)` (depth incremented by 1).
7. **Comparison of Depths of Children of Node 20:**

   - `dfs(15)` returned `(15, 1)` and `dfs(7)` returned `(4, 2)`.
   - Since node 7's (right child) subtree is deeper, `dfs(20)` returns `(4, 3)` (depth incremented by 1).
8. **Comparison of Depths of Children of Node 3:**

   - The left child returned `(9, 1)` and the right child returned `(4, 3)`.
   - Since the right child is deeper, `dfs(3)` would return `(4, 4)`.

At this point, it may seem like node 4 is the answer, but we have not yet considered the fact that the answer should include all deepest nodes, not just the deepest one.

9. **However, there's an oversight:**

   - When comparing depths of node 3's children, we find the left and right subtrees have unequal depths (1 and 3).
   - According to our solution approach, if both children have the same depth, then the current node is the LCA.
   - But since the depths are different, we mistakenly consider the right subtree to hold all the deepest nodes, which is not the case.

To correct this, we should identify that the node 7 being part of that subtree results in a deeper subtree but doesn't include all the deepest nodes. Hence, while node 4 is the deepest, the subtree rooted at node 3 is the smallest subtree that contains all deepest nodes (9, 15, and 4).

10. **Correcting the Depth Equality Condition:**

- To ensure we are not merely finding the deepest node, we would need to refactor the logic such that, when the depth of the deepest nodes are equal (like for 9, 15, and 4 at depth 2), we return the current node as the LCA.
- For node 20, as its children nodes 15 and 4 are deepest nodes, it should return `(20, 3)`.
- For node 3, since its children nodes 9 and 20 include all deepest nodes, it should correctly return `(3, 4)`.

After implementing this correction, the function identifies node 3 as the root of the smallest subtree containing all the deepest nodes (9, 15, and 4), which is what the problem asks for.

In the actual DFS implementation, the comparison step would need to account for this logic to ensure that when there are multiple deepest nodes with equal maximum depth, the node that connects them (their LCA) is returned.

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 subtreeWithAllDeepest(self, root: TreeNode) -> TreeNode:
        # Helper function to perform depth-first search and find deepest nodes
        def depth_first_search(node):
            # Base case: if the node is None, return None and depth 0
            if not node:
                return None, 0

            # Recursively find the deepest nodes and their depth in the left subtree
            left_subtree, left_depth = depth_first_search(node.left)
            # Recursively find the deepest nodes and their depth in the right subtree
            right_subtree, right_depth = depth_first_search(node.right)

            # If the left subtree is deeper, return the left subtree and its depth
            if left_depth > right_depth:
                return left_subtree, left_depth + 1
            # If the right subtree is deeper, return the right subtree and its depth
            elif left_depth < right_depth:
                return right_subtree, right_depth + 1
            # If both subtrees have the same depth, return the current node as the common ancestor
            else:
                return node, left_depth + 1

        # Call the helper function and return the first element of the tuple,
        # which is the subtree that contains all the deepest nodes
        return depth_first_search(root)[0]

```

Java Solution

```java
import javafx.util.Pair;

// Definition for a binary tree node.
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 {
    // Entry method to find the smallest subtree that contains all the deepest nodes.
    public TreeNode subtreeWithAllDeepest(TreeNode root) {
        return deepSubtreeDfs(root).getKey();
    }

    // Helper method that returns the smallest subtree containing all the deepest nodes
    // along with the depth of that subtree.
    private Pair<TreeNode, Integer> deepSubtreeDfs(TreeNode node) {
        // Base case: if the current node is null, return a pair of null and 0 depth.
        if (node == null) {
            return new Pair<>(null, 0);
        }

        // Recursively find the pair of subtree and depth for the left child.
        Pair<TreeNode, Integer> leftPair = deepSubtreeDfs(node.left);
        // Recursively find the pair of subtree and depth for the right child.
        Pair<TreeNode, Integer> rightPair = deepSubtreeDfs(node.right);

        // Depth of left subtree.
        int leftDepth = leftPair.getValue();
        // Depth of right subtree.
        int rightDepth = rightPair.getValue();

        // If left subtree is deeper, return the left child and its depth increased by one.
        if (leftDepth > rightDepth) {
            return new Pair<>(leftPair.getKey(), leftDepth + 1);
        }

        // If right subtree is deeper, return the right child and its depth increased by one.
        if (leftDepth < rightDepth) {
            return new Pair<>(rightPair.getKey(), rightDepth + 1);
        }

        // If both subtrees have the same depth, the current node is the LCA of the deepest nodes,
        // so return the current node and the depth increased by one.
        return new Pair<>(node, leftDepth + 1);
    }
}

```

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) {}
};

// Utility type to hold a pair of TreeNode* and int (depth)
using TreeNodeDepthPair = pair<TreeNode*, int>;

class Solution {
public:
    TreeNode* subtreeWithAllDeepest(TreeNode* root) {
        return depthFirstSearch(root).first;
    }

    TreeNodeDepthPair depthFirstSearch(TreeNode* node) {
        // Base case: if the node is null, return pair of nullptr and depth 0
        if (!node) return {nullptr, 0};

        // Recursively find the deepest subtree for the left and right children
        TreeNodeDepthPair leftSubtree = depthFirstSearch(node->left);
        TreeNodeDepthPair rightSubtree = depthFirstSearch(node->right);

        // Compare the depths
        int leftDepth = leftSubtree.second, rightDepth = rightSubtree.second;

        // If the left subtree is deeper, return the left subtree with depth increased by 1
        if (leftDepth > rightDepth) return {leftSubtree.first, leftDepth + 1};

        // If the right subtree is deeper, return the right subtree with depth increased by 1
        if (leftDepth < rightDepth) return {rightSubtree.first, rightDepth + 1};

        // If both subtrees have the same depth, the current node is the common ancestor
        // Return the current node with the increased depth
        return {node, leftDepth + 1};
    }
};

```

Typescript Solution

```typescript
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

// Type to represent a pair of TreeNode and depth
type TreeNodeDepthPair = [TreeNode | null, number];

/**
 * Finds the subtree that contains all the deepest nodes.
 * @param root - The root of the binary tree.
 * @returns The subtree with all deepest nodes.
 */
function subtreeWithAllDeepest(root: TreeNode | null): TreeNode | null {
  return depthFirstSearch(root)[0];
}

/**
 * Performs a depth-first search to find the deepest subtree.
 * @param node - The current node in the binary tree.
 * @returns A tuple containing the deepest subtree and its depth.
 */
function depthFirstSearch(node: TreeNode | null): TreeNodeDepthPair {
  // Base case: if the node is null, return tuple of null and depth 0
  if (!node) return [null, 0];

  // Recursively find the deepest subtree for the left and right children
  const leftSubtree: TreeNodeDepthPair = depthFirstSearch(node.left);
  const rightSubtree: TreeNodeDepthPair = depthFirstSearch(node.right);

  // Extract the depths from the left and right subtree
  const leftDepth: number = leftSubtree[1], rightDepth: number = rightSubtree[1];

  // If the left subtree is deeper, return the left subtree with depth increased by 1
  if (leftDepth > rightDepth) return [leftSubtree[0], leftDepth + 1];

  // If the right subtree is deeper, return the right subtree with depth increased by 1
  if (leftDepth < rightDepth) return [rightSubtree[0], rightDepth + 1];

  // If both subtrees have the same depth, the current node is the common ancestor
  // Return the current node with the increased depth
  return [node, leftDepth + 1];
}
```

Additionally, a `TreeNode` constructor function may be used to create TreeNode instances in TypeScript:

```typescript
function createTreeNode(
  val: number,
  left: TreeNode | null = null,
  right: TreeNode | null = null
): TreeNode {
  return {
    val,
    left,
    right
  };
}

```

Time and Space Complexity

The time complexity of the code is `O(N)`, where `N` is the number of nodes in the binary tree. This is because the recursive `dfs` function traverses every node exactly once to compute the depth of the deepest leaves and to find the common ancestor.

The space complexity of the code is also `O(N)` in the worst case, which occurs in the case of a skewed tree (where each node has only one child except for the leaf node). This is because the recursion stack would then contain `N` function calls. In the average case of a balanced tree, the space complexity would be `O(log N)`, which corresponds to the height of the tree.