1676. Lowest Common Ancestor of a Binary Tree IV

Problem Description

The problem is based on a fundamental concept from graph theory applied to binary trees, known as the "lowest common ancestor" (LCA). Specifically, we are given the root node of a binary tree where all node values are unique, and an array of `TreeNode` objects that are present in the tree. Our task is to determine the LCA of all of these nodes. The LCA is defined as the lowest (deepest) node in the tree that has every given node in the array as a descendant (including the possibility of a node being a descendant of itself). The concept of "lowest" here refers to the deepest node in the hierarchy of the tree. In other words, we are looking for a shared ancestor of these nodes that is as far down the tree as possible.

Intuition

To solve this problem, we can use a depth-first search (DFS) algorithm that explores each subtree rooted at the provided `root` node. Since all node values are unique, we can use a set to store the values of all the nodes we're trying to find the LCA for. This allows us to efficiently check if a node is part of our target group.

The recursive DFS function will traverse down the tree in a post-order fashion (left, right, node). Here's what happens at each step:

1. If we reach a `None` node or a node whose value is in our set of target nodes, we return that node up the call stack. Returning a target node signals that we've found one of the nodes we're interested in.
2. We then call the DFS function on the left and the right children of the current node.
3. Upon receiving the results from both subtrees:
   - If both calls returned a non-`None` node, it means that both subtrees contain at least one target node each. In this case, the current node is the LCA, as it is the lowest node that has descendants from both subtrees.
   - If only one of the calls returned a non-`None` node, it implies that all the target nodes are located in one subtree. Hence, the result from that subtree is the current LCA for the nodes encountered so far.
   - If both calls returned `None`, neither subtree contains any of the target nodes, and thus we also return `None`.
4. Initially, the DFS is called on the `root` node and the set of values that correspond to our target nodes.
5. Ultimately, the last non-`None` node that is returned by our DFS function is the LCA of all the target nodes.

This recursive algorithm effectively prunes branches of the tree that do not contain any of the target nodes to find the LCA efficiently.

Solution Approach

The implementation of the solution consists of a depth-first search (DFS) function called `dfs`, which is responsible for traversing the binary tree to find the lowest common ancestor (LCA) of the given set of nodes. Here's how the implementation unfolds in detail:

1. A set `s` created to store the values of the nodes provided in the list `nodes`. This is done to allow for quick membership checks, as we want to know whether we have encountered one of the target nodes during traversal.

```
s = {node.val for node in nodes}
```

2. The `dfs` function is then defined to perform a recursive post-order traversal (visit left subtree, then right subtree, then the node itself). Like so:

```
def dfs(root):
    if root is None or root.val in s:
        return root
    left = dfs(root.left), dfs(root.right)
    if left and right:
        return root
    return left or right
```

- The base case of the recursion checks if the current node is `None` or if its value is in the set `s` (meaning it's one of the target nodes). In either scenario, it returns the node itself – `None` indicates we're at a leaf's child, while a node with value in `s` is one of the desired LCA nodes.
- If the current node is not `None` and not in the set `s`, the function recursively calls itself for the left and right children of the current node (`dfs(root.left)` and `dfs(root.right)` respectively).
- After the recursive calls, if both `left` and `right` are not `None`, it means that we've found target nodes in both the left and right subtrees. Thus, the current node must be their LCA, and it is returned as the result.
- If only one of `left` or `right` contains a target node (i.e., one of them is not `None`), that node is returned. This is because the tree branch that returned `None` doesn't contain any of the target nodes, so the LCA must be on the branch that returned a node.

3. Finally, the `dfs` function is called with the root of the binary tree. The tree is traversed, and the lowest common ancestor of all the nodes in `nodes` is returned.

```
return dfs(root)
```

The `dfs` function effectively navigates the tree structure, identifying the points where paths to each of the target nodes diverge, pinpointing the node that serves as the common ancestor with the least depth.

By using a recursive approach, the algorithm avoids checking non-relevant parts of the tree and leverages the call stack to backtrack until it finds the common ancestor. The set data structure enables efficient lookups to determine if a node is part of the target group. Together, these approaches allow the provided solution to efficiently find the LCA with a time complexity that is proportional to the number of nodes in the tree, O(N), where N is the number of nodes in the tree.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Imagine a binary tree that looks like this:

```
    3
   / \
  5   1
 / \ / \
6  2 0  8
  / \
 7   4
```

Let's say we want to find the lowest common ancestor (LCA) of the nodes with values `[5, 1, 6, 2, 8]`.

First, we create a set of these values, which is `{5, 1, 6, 2, 8}` for quick look-up during the traversal.

We then start with the `dfs` function on the root of the tree, which is the node with value 3.

1. The current node 3 is not `None`, and its value is not in the set `{5, 1, 6, 2, 8}`, so we proceed with the recursive calls for its left and right children.
2. For the left child (value 5):
   - The value 5 is in the set, so we return this node immediately without further recursion.
3. For the right child (value 1):
   - The value 1 is in the set, so we return this node immediately without further recursion.

Since the value 5 node was found in the left subtree and the value 1 node was found in the right subtree, and the current node (value 3) is the parent to both subtrees, by the rules of the algorithm, node 3 is the LCA for nodes 5 and 1.

However, since we need to find the LCA of the nodes `[5, 1, 6, 2, 8]`, we need to see if node 3 is also the LCA for the other values.

4. For the left child (value 5):
   - Again, we return the node with value 5 as we did previously. This time, let's explore below it to account for 2 and 6.
   - Node 6 is in the set and is a left child, so it's returned.
   - Node 2 is in the set and is a right child, so it's returned.

Since both 6 and 2 are on the left subtree (under node 5), and node 5 is returned for both, node 5 is the LCA for values `[5, 6, 2]`.

5. For the right child (value 1):
   - Node 0 is in the set and is the left child, so it's returned.

Now, we have the LCAs for the left and right subtrees as node 5 and node 1, respectively.

6. Finally, since node 3 is the parent of both 5 and 1, and no further common ancestors exist that are lower than node 3 for the entire set `{5, 1, 6, 2, 8}`, node 3 remains the LCA for all specified nodes in the set.

Therefore, the function would return the node with value 3 as the LCA.

This example demonstrates how the `dfs` function operates to efficiently find the lowest common ancestor by first populating a set with the values of all the target nodes and then traversing the binary tree recursively to determine the node that serves as the common ancestor to all the target nodes.

Python Solution

```python
class TreeNode:
    def __init__(self, value):
        self.val = value
        self.left = None
        self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', nodes: List['TreeNode']) -> 'TreeNode':
        # Perform depth-first search to find the lowest common ancestor.
        def find_lca(current_node):
            # Base case: If current node is None or is target nodes set, return it.
            if current_node is None or current_node in target_nodes_set:
                return current_node

            # Recursively search the left and right subtrees.
            left_ancestor = dfs(current_node.left)
            right_ancestor = dfs(current_node.right)

            # If both left and right are not None, current node is the lowest common ancestor.
            if left_ancestor and right_ancestor:
                return current_node

            # Otherwise, return the non-None value or None.
            return left_ancestor or right_ancestor

        # Convert the list of nodes to a set for faster lookup.
        # Note that the original code used node.val, assuming unique values for simplicity.
        # Here we use the nodes themselves for the matching, which is more general.
        target_nodes_set = set(nodes)

        # Start the depth-first search from the root.
        return find_lca(root)
```

Note: In the comment, it's mentioned that the original code assumed unique values for node `val`s. However, the algorithm works directly on the node objects in the revised version. This provides a more direct approach in case the tree has non-unique values but distinct nodes.

To run this code, you would need to import the `List` type from the `typing` module in Python 3 by adding the following line before the class definition:

```
from typing import List
```

Java Solution

```java
class Solution {
    // Use a set to keep track of the values of the nodes we're looking for.
    private Set<Integer> targetNodeValues = new HashSet<>();

    /**
     * Finds the lowest common ancestor of all nodes in the array within the binary tree.
     *
     * @param root  The root of the binary tree.
     * @param nodes An array of nodes for which the lowest common ancestor is to be found.
     * @return The lowest common ancestor node.
     */
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode[] nodes) {
        // Populate the set with the values of all target nodes.
        for (TreeNode node : nodes) {
            targetNodeValues.add(node.val);
        }
        // Start DFS to find the lowest common ancestor.
        return findCommonAncestorDFS(root);
    }

    /**
     * Recursive function to find the lowest common ancestor.
     *
     * @param currentNode The node currently being visited in the DFS.
     * @return The lowest common ancestor if it exists, or one of the target nodes, or null.
     */
    private TreeNode findCommonAncestorDFS(TreeNode currentNode) {
        // If reached the end of a path or found one of the target nodes, return current node.
        if (currentNode == null || targetNodeValues.contains(currentNode.val)) {
            return currentNode;
        }
        // Recurse on left subtree.
        TreeNode left = findCommonAncestorDFS(currentNode.left);
        // Recurse on right subtree.
        TreeNode right = findCommonAncestorDFS(currentNode.right);

        // If only one side returned a node, return that node.
        if (left == null) {
            return right;
        }
        if (right == null) {
            return left;
        }
        // If nodes are found on both sides, current node is the lowest common ancestor.
        return currentNode;
    }
}

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 * }
 */
```

C++ Solution

```cpp
#include <unordered_set>
#include <vector>
#include <functional>

// 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:
    /**
     * This function finds the lowest common ancestor (LCA) of a list of nodes in a binary tree.
     */
    TreeNode* lowestCommonAncestor(TreeNode* root, const std::vector<TreeNode*>& nodes) {
        // Create an unordered set to store the values of all target nodes for constant time checks.
        std::unordered_set<int> targetNodeValues;
        for (auto& node : nodes) {
            targetNodeValues.insert(node->val);
        }

        // Define a depth-first search lambda function to find the LCA of the nodes.
        std::function<TreeNode*(TreeNode*)> findCommonAncestorDFS = [&](TreeNode* currentNode) {
            // If we've reached a nullptr or one of the target nodes, return the current node.
            if (currentNode || targetNodeValues.count(currentNode->val)) {
                return currentNode;
            }
            // Recur on the left and right subtrees.
            TreeNode* leftLCA = dfs(currentNode->left);
            TreeNode* rightLCA = dfs(currentNode->right);

            // If both sides return a non-nullptr, we've found the LCA.
            if (leftLCA && rightLCA) {
                return currentNode;
            }
            // Otherwise, if one side is nullptr, return the non-nullptr result.
            return leftLCA ? leftLCA : rightLCA;
        };

        // Begin the search from the root of the tree.
        return dfs(root);
    }
};
```

Typescript Solution

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

    constructor(val: number) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}

/**
 * Finds the lowest common ancestor (LCA) of a given set of nodes in a binary tree.
 * @param {TreeNode | null} root - The root of the binary tree.
 * @param {TreeNode[]} nodes - The list of nodes for which to find the LCA.
 * @return {TreeNode | null} - The LCA node or null if not found.
 */
function lowestCommonAncestor(root: TreeNode | null, nodes: TreeNode[]): TreeNode | null {
    // Create a set to store the values of the nodes for efficient lookup
    const targetNodeValuesSet = new Set();
    for (const node of nodes) {
        nodeValuesSet.add(node.val);
    }

    // Helper function to recursively find the LCA
    function dfs(currentNode: TreeNode | null): TreeNode | null {
        // If the current node is null or the value is in the set, return the current node
        if (currentNode === null || nodeValuesSet.has(currentNode.val)) {
            return currentNode;
        }

        // Recursively search in the left and right subtrees
        const leftLCA: TreeNode | null = dfs(currentNode.left);
        const rightLCA: TreeNode | null = dfs(currentNode.right);

        // If both left and right LCA are found, the current node is the LCA
        if (leftLCA && rightLCA) {
            return currentNode;
        }

        // Otherwise, return whichever side LCA is found, or null if neither is found
        return leftLCA || rightLCA;
    }

    // Start the LCA search from the root of the tree
    return dfs(root);
}

// Example usage:
const root = new TreeNode(3);
// root.left = new TreeNode(5);
// root.right = new TreeNode(1);
// const nodes = [root.left, root.right];
// console.log(lowestCommonAncestor(root, nodes)); // Output should be TreeNode with val = 3
```

Time and Space Complexity

Time 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 Depth First Search (DFS) function `dfs` is called once for each node in the tree in the worst case. The search involves checking if a node's value is in the set `s` of the nodes' values we are looking for and then recursively searching left and right children. Since the set `s` uses a hash set for lookup, the check for `if root.val in s` is $O(1)$, so it does not add more than a constant factor to the complexity.

Space Complexity

The space complexity of the code also is $O(N)$ in the worst-case scenario. This consists of two parts:

1. The space used by the set `s`, which in the worst case could contain all nodes and thus take up $O(N)$ space.
2. The call stack due to recursion, which in the worst case (a degenerate tree, for example) could go up to $O(N)$.

The recursive calls could return early if the nodes are found quickly, which may lead to lower space usage in practice, but the worst-case space complexity remains $O(N)$.