1644. Lowest Common Ancestor of a Binary Tree II

Medium Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

The task is to find the lowest common ancestor (LCA) of two given nodes, `p` and `q`, in a binary tree. The LCA is defined as the lowest node (furthest from the root) that has both nodes `p` and `q` as descendants. It's important to note that a node can be a descendant of itself according to the problem statement. If either `p` or `q` doesn't exist in the tree, the result should be `null`. Every node in the tree has a unique value.

Intuition

To solve this problem, we can use a depth-first search (DFS) traversal. The main idea is to recursively explore the tree, starting from the root and moving towards the leaves, to find the nodes `p` and `q`. We should check three conditions during traversal:

1. Whether the current node is `p` or `q`.
2. Whether one of the node's left descendants is `p` or `q`.
3. Whether one of the node's right descendants is `p` or `q`.

The lowest common ancestor will be the node where both the left and right subtree searches report finding either `p` or `q`. In other words, it's the point in the tree where paths from the root to `p` and `q` diverge.

If `p` and `q` are on the same branch, the LCA will be the one higher up on the path. If `p` is an ancestor of `q`, then `p` is the LCA and vice versa. To implement this, we can do a post-order traversal and return three possible values:

- `True` if `p` or `q` is found,
- `False` if neither is found,
- A node if it is the ancestor of both `p` and `q`.

A global variable or a nonlocal in Python, which Python allows to be modified inside a nested function, is used to keep track of the answer. Once we find `p` and `q`, we assign the LCA to this variable.

Solution Approach

The solution uses a Depth-First Search (DFS) traversal to go through each node in the binary tree and check for the presence of nodes `p` and `q`. To do this, we define a helper function `dfs` within the `lowestCommonAncestor` method.

Here is a breakdown of how the `dfs` function works:

- The function takes three arguments: `root`, `p`, and `q`, where `root` is the current node of the tree that we are exploring.
- It begins with a base case that checks if the `root` is `None`, meaning we have reached the end of a path without finding a node. If this is the case, it returns `False`.
- **Left Recursion**: Recursively call `dfs` for the left child of the current node (`root.left`).
- **Right Recursion**: Recursively call `dfs` for the right child of the current node (`root.right`).

These recursive calls will do a post-order traversal of the tree. After these calls, we have three pieces of information:

- `l`: Whether node `p` or `q` has been found in the left subtree.
- `r`: Whether node `p` or `q` has been found in the right subtree.
- The value of the current node.

Using this information, we can detect the LCA:

- If `l` and `r` are both `True`, it implies that `p` is found in one subtree and `q` in the other, making the current node their LCA, so we set `ans` to `root`.
- If one of `l` or `r` is `True` and the current node's value matches `p` or `q`, the current node is the LCA - this happens when one is a descendant of the other.

After the visit to each node, we return a boolean to indicate if either `p` or `q` has been found in the current subtree or if the current node is `p` or `q`. This boolean is the OR of:

- `l` (result from the left subtree),
- `r` (result from the right subtree), and
- a check whether the current `root` matches either `p` or `q`.

Finally, `lowestCommonAncestor` initializes a variable `ans` to `None`, which is used to store the LCA. We declare `ans` as `nonlocal` inside `dfs` so that it can be modified within the nested function. The `dfs` function is then called with the original `root`, `p`, and `q`. The `ans` is returned as the final result of the `lowestCommonAncestor` method. If `p` and `q` are both present in the tree, `ans` will be their lowest common ancestor; if either is not present, `ans` will remain `None`.

This approach efficiently utilizes the single pass post-order DFS traversal to not just search for `p` and `q` but also to identify the LCA without any additional storage or multiple passes through the tree.

Example Walkthrough

Let's consider a binary tree and walk through the solution to find the lowest common ancestor (LCA) for two given nodes, `p` and `q`.

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

For our example, let's assume we want to find the LCA of nodes 5 and 4.

1. We initiate the `lowestCommonAncestor` method with the root node (3) and nodes `p` (5) and `q` (4).
2. We enter the `dfs` function with the current `root` (node 3) and check for `p` and `q`.
3. Since `root` is not `None`, it's not the base case, so we proceed.
4. **Left Recursion**: We call `dfs(root.left, p, q)` which is `dfs(5, 5, 4)`.
   - Entering the `dfs` function again for node 5, we check the left and right children.
   - **Left Recursion**: Calling `dfs(5.left, p, q)` which is `dfs(6, 5, 4)` returns `False`.
   - **Right Recursion**: Calling `dfs(5.right, p, q)` which is `dfs(2, 5, 4)`.
     - For node 2, we again explore its children.
     - **Left Recursion**: Calling `dfs(2.left, p, q)` which is `dfs(7, 5, 4)` returns `False`.
     - **Right Recursion**: Calling `dfs(2.right, p, q)` which is `dfs(4, 5, 4)` returns `True` because `root` matches `q`.
   - Since `dfs(2, 5, 4)` found `q`, it returns `True` to `dfs(5, 5, 4)` call.
5. Returning to `dfs(5, 5, 4)`, the left call returned `False`, but the right returned `True`, and since the current `root` is `p` (5), we find that 5 is indeed the LCA because it is an ancestor to `q` (4).
6. The result of `True` is then carried up and we exit the left side of the root.
7. **Right Recursion**: We now call `dfs(root.right, p, q)` which is `dfs(1, 5, 4)`.
   - This path will not find either `p` or `q`, and will therefore return `False`.
8. With `dfs(3, 5, 4)`, since the `l` (Left Recursion) returned `True` and `r` (Right Recursion) returned `False`, and no match for `p` or `q` is found at the current node (3), we simply propagate the `True` back up.
9. Since the LCA has been set to node 5 within `dfs(5, 5, 4)`, the global `ans` variable will now hold the reference to the LCA.
10. Finally, `lowestCommonAncestor` returns `ans` which is node 5, and this node is indeed the LCA of nodes 5 and 4 as per our tree structure.

In conclusion, the example walks through the DFS approach to efficiently find the LCA by using a helper `dfs` function that handles recursive traversal and updates an ancestor variable when both nodes are found within the subtrees.

Python Solution

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


class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        # This variable will hold the lowest common ancestor once it is found.
        self.ancestor = None

        def dfs(current_node):
            """
            Perform a depth-first search to find the lowest common ancestor.

            Args:
            current_node (TreeNode): The current node being visited.

            Returns:
            bool: True if the current node is ancestor or is a subtree containing p or q.
            """
            if current_node is None:
                return False

            # Search left subtree for p or q
            left = dfs(current_node.left)

            # Search right subtree for p or q
            right = dfs(current_node.right)

            # Check if current node is either p or q
            mid = current_node == p or current_node == q

            # If any two of the three flags left, right, mid are True, current_node is an ancestor
            if mid + left + right >= 2:
                self.ancestor = current_node

            # Return True if the current node is p, q, or if p or q is in the subtree rooted at current_node
            return mid or left or right

        # Call dfs to initiate the depth-first search
        dfs(root)

        return self.ancestor

# The provided solution can be used to find the lowest common ancestor (LCA) of two nodes in a binary tree.

```

Java Solution

```java
class Solution {

    // The variable 'answer' will hold the solution.
    private TreeNode answer;

    // This method returns the lowest common ancestor of two nodes in the binary tree.
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // Initiate the depth-first search.
        findLowestCommonAncestor(root, p, q);
        return answer;
    }

    // Helper method for a DFS to identify if the current node is part of the path to p or q.
    private boolean findLowestCommonAncestor(TreeNode currentNode, TreeNode p, TreeNode q) {
        // Base case: if the current node is null, return false.
        if (currentNode == null) {
            return false;
        }

        // Traverse the left side of the tree and store if p or q was found in the left subtree.
        boolean foundInLeftSubtree = findLowestCommonAncestor(currentNode.left, p, q);
        // Traverse the right side of the tree and store if p or q was found in the right subtree.
        boolean foundInRightSubtree = findLowestCommonAncestor(currentNode.right, p, q);

        // If both left and right subtrees contain p or q, then the current node is the LCA.
        if (foundInLeftSubtree && foundInRightSubtree) {
            answer = currentNode;
        }

        // If either left or right subtree contains p or q, and the current node is either p or q,
        // then the current node is the LCA.
        if ((foundInLeftSubtree || foundInRightSubtree) && (currentNode.val == p.val || currentNode.val == q.val)) {
            answer = currentNode;
        }

        // Return true if the current node is either p or q or if p or q is found in either subtree.
        return foundInLeftSubtree || foundInRightSubtree || currentNode.val == p.val || currentNode.val == q.val;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    /**
     * Finds the lowest common ancestor (LCA) of two given nodes in a binary tree.
     * @param root The root node of the binary tree.
     * @param p The first node for which LCA is to be found.
     * @param q The second node for which LCA is to be found.
     * @return The LCA of nodes p and q.
     */
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        findLowestCommonAncestor(root, p, q);
        return ancestor;
    }

private:
    TreeNode* ancestor = nullptr; // Holds the lowest common ancestor once found

    /**
     * Helper method to perform DFS to find LCA.
     * @param current The current node being looked at.
     * @param nodeP The first node for which LCA is to be found.
     * @param nodeQ The second node for which LCA is to be found.
     * @return True if the current subtree contains either nodeP or nodeQ.
     */
    bool findLowestCommonAncestor(TreeNode* current, TreeNode* nodeP, TreeNode* nodeQ) {
        if (!current) {
            return false; // Base case: reached the end of a branch
        }

        bool left = findLowestCommonAncestor(current->left, nodeP, nodeQ); // Search LCA in the left subtree
        bool right = findLowestCommonAncestor(current->right, nodeP, nodeQ); // Search LCA in the right subtree

        // If both left and right are true, current is the LCA
        if (left && right) {
            ancestor = current;
        }

        // If either left or right is true and current is either nodeP or nodeQ, current is the LCA
        if ((left || right) && (current->val == nodeP->val || current->val == nodeQ->val)) {
            ancestor = current;
        }

        // Return true if current is nodeP or nodeQ or if left or right are true
        return left || right || current->val == nodeP->val || current->val == nodeQ->val;
    }
};

```

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 = this.right = null;
    }
}

/**
 * Finds the lowest common ancestor (LCA) of two nodes in a binary tree.
 * @param {TreeNode} root - The root of the binary tree.
 * @param {TreeNode} node1 - The first node to find the LCA for.
 * @param {TreeNode} node2 - The second node to find the LCA for.
 * @return {TreeNode | null} - The LCA of node1 and node2, or null if none found.
 */
const lowestCommonAncestor = (
    root: TreeNode,
    node1: TreeNode,
    node2: TreeNode
): TreeNode | null => {
    // Declaration of the LCA variable.
    let answer: TreeNode | null = null;

    // Depth-first search (DFS) function to traverse the tree and find the LCA.
    const depthFirstSearch = (currentNode: TreeNode | null): boolean => {
        if (!currentNode) {
            // If the current node is null, return false indicating the node is not part of the ancestry of node1 or node2.
            return false;
        }

        // Recursively search in the left subtree.
        const leftSearch = depthFirstSearch(currentNode.left);
        // Recursively search in the right subtree.
        const rightSearch = depthFirstSearch(currentNode.right);

        // If both left and right subtrees return true, the current node is the LCA.
        if (leftSearch && rightSearch) {
            answer = currentNode;
        }

        // If either the left or right subtree returns true and the current node equals one of the nodes we're looking for,
        // the current node is the LCA.
        if (
            (leftSearch || rightSearch) &&
            (currentNode.val === node1.val || currentNode.val === node2.val)
        ) {
            answer = currentNode;
        }

        // Return true if the current node is equal to node1 or node2 or if either the left or right subtree returns true.
        return (
            leftSearch || rightSearch || currentNode.val === node1.val || currentNode.val === node2.val
        );
    };

    // Initialize the search.
    depthFirstSearch(root);

    // Return the identified LCA or null if none found.
    return answer;
};

```

Time and Space Complexity

The provided code implements a Depth First Search (DFS) algorithm to find the lowest common ancestor (LCA) of two nodes in a binary tree.

Time Complexity:

To determine the time complexity, we consider that the function dfs is called recursively for every node in the tree until it finds the nodes `p` and `q`.

- It visits each node exactly once.
- The work done at each node is constant (comparisons and boolean checks).

Therefore, the time complexity is `O(N)`, where `N` is the number of nodes in the binary tree.

Space Complexity:

The space complexity is determined by the depth of the recursion stack which is the height of the binary tree.

- In the worst case for a skewed tree (like a linked list), the recursion stack could be as deep as the number of nodes, which would be `O(N)`.
- In the best case for a balanced tree, the height of the tree would be `log(N)`, and thus the space complexity would be `O(log(N))`.

The space complexity of the algorithm is `O(H)`, where `H` is the height of the tree, which is `O(N)` in the worst case and `O(log(N))` in the best case.