236. Lowest Common Ancestor of a Binary Tree

Medium Tree Depth-First Search Binary Tree

Problem Description

In this problem, we are given the roots of a binary tree and two nodes from this tree. Our task is to find the lowest common ancestor (LCA) of these two nodes. The LCA of two nodes `p` and `q` is defined as the deepest node in the tree that has both `p` and `q` as descendants (with a node being allowed to be a descendant of itself).

Intuition

The solution to this problem is based on a recursive traversal of the binary tree. When we traverse the tree, we are looking for three possible conditions at each node:

1. One condition is that we have found either node `p` or `q`. In this case, the current node could potentially be the LCA.
2. The second condition is that one of the nodes `p` or `q` is found in the left subtree and the other is found in the right subtree of the current node. If this is the case, then the current node is the LCA for `p` and `q`, as it sits above both nodes in the tree.
3. The third condition is that both nodes are located in either the left subtree or the right subtree of the current node. In this instance, the lowest common ancestor will be deeper in the tree, so we continue searching in the subtree that contains both nodes.

The base case for our recursion occurs when we reach a `null` node (indicating that we've reached the end of a path and haven't found either `p` or `q`), or when we find one of the nodes `p` or `q`.

To implement this intuition, we use a recursive algorithm that will search for `p` and `q` starting from `root`. At each step, we make a recursive call to search the left and right subtrees. If both recursive calls return non-`null` nodes, it means we've found `p` and `q` in different subtrees of the current node, and thus the current node is the LCA. If only one of the subtrees contains one of the nodes (or both), we return that subtree's node, propagating it up the call stack. If neither subtree contains either of the nodes, we return `null`.

This recursive approach continues until the LCA is found, or we've confirmed that `p` and `q` are not present in the binary tree.

Solution Approach

To implement the solution for finding the lowest common ancestor, the algorithm employs a depth-first search (DFS) pattern, which is a type of traversal that goes as deep as possible down one path before backing up and trying another. This pattern is particularly useful for working with tree data structures. Here is an overview of how the DFS is applied in this solution:

1. **Recursive Function**: The solution defines a recursive function `lowestCommonAncestor` which takes the current node (`root`), and the two nodes we are finding the LCA for (`p` and `q`). Recursion leverages the call stack to perform a backtracking search, allowing us to explore all paths down the tree and retrace our steps.
2. **Base Case**: The base case of the recursion occurs when either the current node is `null`, or the current node matches one of the nodes we're looking for (`p` or `q`). In this case, the function returns the current node, which may be `null` or the matching node.
3. **Search Left and Right Subtrees**: If the base case is not met, the algorithm recursively calls `lowestCommonAncestor` for the left and right children of the current node. These calls effectively search for the nodes `p` and `q` in both subtrees.
4. **Postorder Traversal**: Since the recursion explores both left and right subtrees before dealing with the current node, this represents a postorder traversal pattern. After both subtrees have been searched, the algorithm processes the current node.
5. **LCA Detection**:
   - If both the left and right subtree recursive calls return non-`null` nodes, it means both `p` and `q` have been found in different subtrees, and therefore the current node is their lowest common ancestor.
   - If only one of the calls returns a non-`null` node, that indicates the current subtree contains at least one of the two nodes, and potentially both. The non-`null` node is returned up the call stack.
   - If both calls return `null`, it means neither `p` nor `q` was found in the current subtree, and `null` is returned.
6. **Propagation of the LCA**: The LCA, once found, is propagated up the call stack to the initial call, and ultimately returned as the result of the function.

Here is the implementation of the algorithm in the reference solution provided:

```python
class Solution:
    def lowestCommonAncestor(
        self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode'
    ) -> 'TreeNode':
        if root is None or root == p or root == q:
            return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        return root if left and right else (left or right)
```

In the provided code snippet, the `if` condition handles the base case, while the calls to `lowestCommonAncestor(root.left, p, q)` and `lowestCommonAncestor(root.right, p, q)` implement the recursive search. The final return statement decides which node to return based on whether the left and/or right calls found the nodes `p` and `q`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the following binary tree:

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

In this binary tree, let's say we want to find the lowest common ancestor (LCA) of the nodes D and E.

1. We start our recursive function `lowestCommonAncestor` at the root node A.
2. Since the root is neither null, nor is it D or E, we proceed to make recursive calls to both the left and right subtrees of A.
3. The left recursive call goes to node B. Again, B isn't null, D, or E, so we continue.
4. Recursively, for node B, we go left to D and right to E. Node D matches one of our target nodes, so the left call returns D to the call on B. Similarly, on the right side, the node E matches the other target node, so the right call returns E to the call on B.
5. With both sides of B returning a non-null node (D on the left and E on the right), we determine that B must be the LCA because it's the node where both target nodes D and E subtrees split. The function returns node B up to the call on A.
6. Since the recursive call on the left of A returned B (non-null) and the call on the right of A (to C) will return null (since neither D nor E is in that subtree), the final decision at A is based on whether one or both sides are non-null.
7. Since the right call returns null, and the left call returns B (non-null), the function finally returns B, the LCA of D and E.

Following the recursive definition used in our solution, we can confirm that node B is the lowest common ancestor of nodes D and E in this binary tree.

Solution Implementation

Python

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


class Solution:
    def lowestCommonAncestor(self, root: TreeNode, node1: TreeNode, node2: TreeNode) -> TreeNode:
        # If the root is None, or the root is one of the nodes we're looking for,
        # we return the root as the LCA (Lowest Common Ancestor)
        if root is None or root == node1 or root == node2:
            return root

        # Look for the LCA in the left subtree
        left_lca = self.lowestCommonAncestor(root.left, node1, node2)

        # Look for the LCA in the right subtree
        right_lca = self.lowestCommonAncestor(root.right, node1, node2)

        # If both left and right LCA are non-null, it means one node is in the left
        # subtree and the other is in the right, so root is the LCA
        if left_lca and right_lca:
            return root

        # Otherwise, if one of the LCAs is non-null, return that one
        return left_lca if left_lca else right_lca
```

Java

```java
// Definition for a binary tree node.
class TreeNode {
    int val;           // The value of the node
    TreeNode left;     // Reference to the left child
    TreeNode right;    // Reference to the right child

    // Constructor to initialize the node with a value
    TreeNode(int x) {
        val = x;
    }
}

class Solution {
    /**
     * Finds the lowest common ancestor of two nodes in a binary tree.
     * @param root The root node of the binary tree.
     * @param p The first node to find the ancestor for.
     * @param q The second node to find the ancestor for.
     * @return The lowest common ancestor node or null if not found.
     */
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // If the root is null or root is either p or q, then root is the LCA
        if (root == null || root == p || root == q) return root;

        // Recurse on the left subtree to find the LCA of p and q
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        // Recurse on the right subtree to find the LCA of p and q
        TreeNode right = lowestCommonAncestor(root.right, p, q);

        // If finding LCA in the left subtree returns null, the LCA is in the right subtree
        if (left == null) return right;
        // If finding LCA in the right subtree returns null, the LCA is in the left subtree
        if (right == null) return left;

        // If both left and right are non-null, we've found the LCA at the root
        return root;
    }
}
```

C++

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

class Solution {
public:
    /* Function to find the lowest common ancestor (LCA) of two given nodes in a binary tree.
     *
     * @param root The root node of the binary tree.
     * @param firstNode The first node for which the LCA is to be found.
     * @param secondNode The second node for which the LCA is to be found.
     * @return The LCA of the two nodes.
     */
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* firstNode, TreeNode* secondNode) {
        // If root is nullptr or root is one of the nodes, then root itself is the LCA.
        if (!root || root == firstNode || root == secondNode) return root;

        // Recursively find the LCA in the left subtree.
        TreeNode* leftLCA = lowestCommonAncestor(root->left, firstNode, secondNode);
        // Recursively find the LCA in the right subtree.
        TreeNode* rightLCA = lowestCommonAncestor(root->right, firstNode, secondNode);

        // If both the left and right subtrees contain one of the nodes each, then the current root is the LCA.
        if (leftLCA && rightLCA) return root;

        // If only one subtree contains both the nodes, return that subtree's LCA.
        return leftLCA ? leftLCA : rightLCA;
    }
};
```

TypeScript

```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 (LCA) of two nodes in a binary tree.
 * The LCA is defined as the lowest node in T that has both p and q as descendants
 * (where we allow a node to be a descendant of itself).
 *
 * @param root - The root node of the binary tree.
 * @param p - The first node to find the LCA for.
 * @param q - The second node to find the LCA for.
 * @return The LCA node or null if either p or q is not present in the tree.
 */
function lowestCommonAncestor(
    root: TreeNode | null,
    p: TreeNode | null,
    q: TreeNode | null,
): TreeNode | null {
    /**
     * Recursively searches the binary tree to find the LCA of nodes p and q.
     *
     * @param currentNode - The current node being inspected in the binary tree.
     * @return The LCA TreeNode, if it exists or null if not.
     */
    function findLCA(currentNode: TreeNode | null): TreeNode | null {
        // If the current node is null, or we have found either p or q, return the current node
        if (currentNode === null || currentNode === p || currentNode === q) {
            return currentNode;
        }

        // Recursively search in the left subtree for one of the nodes
        const leftSubtree = findLCA(currentNode.left);
        // Recursively search in the right subtree for the other node
        const rightSubtree = findLCA(currentNode.right);

        // If both the left and right subtree calls return non-null, it means we have found the nodes p and q in both subtrees,
        // so the current node is the LCA.
        if (leftSubtree !== null && rightSubtree !== null) {
            return currentNode;
        }

        // If only one of the subtrees contains one of the nodes, return that subtree; if neither contains any, return null.
        return leftSubtree !== null ? leftSubtree : rightSubtree;
    }

    // Start the LCA search from the root node
    return findLCA(root);
}
```

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


class Solution:
    def lowestCommonAncestor(self, root: TreeNode, node1: TreeNode, node2: TreeNode) -> TreeNode:
        # If the root is None, or the root is one of the nodes we're looking for,
        # we return the root as the LCA (Lowest Common Ancestor)
        if root is None or root == node1 or root == node2:
            return root

        # Look for the LCA in the left subtree
        left_lca = self.lowestCommonAncestor(root.left, node1, node2)

        # Look for the LCA in the right subtree
        right_lca = self.lowestCommonAncestor(root.right, node1, node2)

        # If both left and right LCA are non-null, it means one node is in the left
        # subtree and the other is in the right, so root is the LCA
        if left_lca and right_lca:
            return root

        # Otherwise, if one of the LCAs is non-null, return that one
        return left_lca if left_lca else right_lca
```

Time and Space Complexity

The time complexity of the given code for finding the lowest common ancestor (LCA) in a binary tree is `O(N)`, where `N` is the number of nodes in the tree. This complexity arises because, in the worst-case scenario, the code will have to visit each node once to determine the LCA.

The space complexity of the code is `O(H)`, where `H` is the height of the binary tree. This is due to the recursive calls on the stack while the algorithm traverses down to find `p` and `q`. In the worst case of a skewed tree, `H` can be `N`, thereby having a space complexity of `O(N)`.