235. Lowest Common Ancestor of a Binary Search Tree

Medium Tree Depth-First Search Binary Search Tree Binary Tree Leetcode Link

Problem Description

In this problem, we are given a binary search tree (BST). Our task is to find the lowest common ancestor (LCA) of two given nodes, `p` and `q`, within this BST. The lowest common ancestor is the furthest node from the root that is an ancestor of both nodes. This means that the LCA is the node from which both `p` and `q` are descended, and it could be either of the nodes themselves if one is an ancestor of the other.

Intuition

The intuition behind the solution stems from the properties of a BST. In a BST, for any node `n`, all nodes in the left subtree of `n` have values less than `n`, and all nodes in the right subtree have values greater than `n`. When searching for the LCA of `p` and `q`, there are three possibilities:

1. The values of both `p` and `q` are less than the value of the current node. This means the LCA must be in the left subtree.
2. The values of both `p` and `q` are greater than the value of the current node. This means the LCA must be in the right subtree.
3. One value is less than the current node's value and the other is greater (or one of them is equal to the current node's value). This means that the current node is the LCA because `p` and `q` are in different subtrees.

The given solution iterates over the tree without using recursion. It repeatedly compares the current node's value with `p` and `q` values to determine the direction of the search or to conclude that the current node is the LCA.

Solution Approach

In the provided solution, the algorithm uses iteration to navigate through the tree and find the LCA. It does so without the need for additional data structures, which keeps the space complexity at a constant O(1). There is also no recursion involved, which means the solution does not rely on the call stack and thus does not incur extra space cost due to recursion stack frames.

The algorithm can be broken down into the following steps:

1. Start with the root of the BST as the current node.
2. Compare the values of the current node (`root.val`) with the values of `p` and `q`.
3. If both `p` and `q` have values less than `root.val`, the LCA must be in the left subtree. Therefore, update the current node to be `root.left` and continue the search.
4. If both `p` and `q` have values greater than `root.val`, the LCA must be in the right subtree. Therefore, update the current node to be `root.right` and continue the search.
5. If the value of `p` is less than `root.val` and the value of `q` is greater than `root.val`, or vice versa, this implies that `p` and `q` lie in different subtrees of the current node. Thus, the current node must be their LCA.
6. In a case where either `p` or `q` is equivalent to `root.val`, it means that either `p` or `q` is the LCA itself, as it is an ancestor of the other node (`a node to be a descendant of itself`).

The algorithm continues the iteration until the condition in step 5 or 6 is met. Once either condition is met, it returns the current node, which is the LCA.

Here's a breakdown of the code:

```python
while 1:  # Start an indefinite loop that will break once LCA is found
    if root.val < min(p.val, q.val):  # Check if both nodes are in the right subtree
        root = root.right
    elif root.val > max(p.val, q.val):  # Check if both nodes are in the left subtree
        root = root.left
    else:
        return root  # Found the LCA or one of the nodes is the LCA itself
```

The loop will always terminate because of the BST property which ensures that, with each step, the current node is moving closer to the LCA until it is found.

Example Walkthrough

Let's consider a binary search tree (BST) for our example:

```
        6
       / \
      2   8
     / \ / \
    0  4 7  9
      / \
     3   5
```

Assume we need to find the lowest common ancestor (LCA) of nodes `p` with value 2 and `q` with value 8.

Following the solution approach:

1. We start with the root of the BST, which has a value of 6.
2. We compare the values of `p` and `q` to the current node (root), which is 6.
    - `p.val` is 2 and `q.val` is 8.
3. Since 6 is greater than 2 and less than 8, it is neither solely in the left subtree nor in the right subtree relative to both `p` and `q`. Therefore, we have found the condition from step 5: `p` and `q` lie in different subtrees of the current node.
4. According to the algorithm, we have found the LCA because one node is in the left subtree and the other node is in the right subtree of node 6.

So, the lowest common ancestor of nodes 2 and 8 in this BST is the node with the value 6. The search stops here and we return the current node (root) as the LCA.

Python Solution

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


class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        # Loop indefinitely until the lowest common ancestor is found
        while True:
            # Find the smaller and larger values of p and q
            smaller_value = min(p.val, q.val)
            larger_value = max(p.val, q.val)

            # If both p and q are on the right of the current node, move right
            if root.val < smaller_value:
                root = root.right
            # If both p and q are on the left of the current node, move left
            elif root.val > larger_value:
                root = root.left
            # If we are in a position where p and q are on different sides
            # of root, or one of them is equal to root, then root is the LCA
            else:
                return root

```

Java Solution

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

    // Constructor to create a new node with a given value
    TreeNode(int value) {
        val = value;
    }
}

class Solution {

    // Function to find the lowest common ancestor of two nodes in a binary search tree
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode firstNode, TreeNode secondNode) {
        // Traverse the tree starting with the root
        while (root != null) {
            // If both nodes are greater than current node, search in right subtree
            if (root.val < Math.min(firstNode.val, secondNode.val)) {
                root = root.right; // Move to the right child
            }
            // If both nodes are less than current node, search in left subtree
            else if (root.val > Math.max(firstNode.val, secondNode.val)) {
                root = root.left; // Move to the left child
            }
            // We've found the lowest common ancestor node
            else {
                return root;
            }
        }
        // In case the while loop exits without returning (it shouldn't in proper usage), return null
        return null;
    }
}

```

C++ Solution

```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 in a binary search tree.
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // Loop until the lowest common ancestor is found.
        while (true) {
            // If both p and q are greater than root, then LCA must be in the right subtree.
            if (root->value < min(p->value, q->value)) {
                root = root->right;
            }
            // If both p and q are less than root, then LCA must be on the left subtree.
            else if (root->value > max(p->value, q->value)) {
                root = root->left;
            }
            // If we are in a state where p and q are on different sides of the root,
            // or one of them is the root itself, we've found the LCA.
            else {
                return root;
            }
        }
    }
};

```

Typescript Solution

```typescript
// Definition of the binary tree node class interface to illustrate the structure of nodes in the binary tree.
interface ITreeNode {
    val: number;
    left: ITreeNode | null;
    right: ITreeNode | null;
}

/**
 * Finds the lowest common ancestor (LCA) of two nodes in a binary search tree.
 *
 * @param root The root node of the binary search tree.
 * @param nodeP The first node to find the LCA for.
 * @param nodeQ The second node to find the LCA for.
 * @returns The lowest common ancestor of nodeP and nodeQ.
 */
function lowestCommonAncestor(root: ITreeNode | null, nodeP: ITreeNode | null, nodeQ: ITreeNode | null): ITreeNode | null {
    // Continue searching for the LCA as long as the current root is not null.
    while (root) {
        // If both nodes are smaller than the current root, LCA must be in the left subtree.
        if (root.val > nodeP.val && root.val > nodeQ.val) {
            root = root.left;
        // If both nodes are larger than the current root, LCA must be in the right subtree.
        } else if (root.val < nodeP.val && root.val < nodeQ.val) {
            root = root.right;
        // If we are in a situation where one node is on the left and the other is on the right,
        // or one of the nodes is equal to the root, the current root is the LCA.
        } else {
            return root;
        }
    }

    // If the root is null, it means we haven't found the LCA (which should not happen in a valid BST with both nodes present).
    return null;
}

```

Time and Space Complexity

The time complexity of the given code is `O(H)`, where `H` is the height of the binary tree. This is because the code traverses the tree from the root to the lowest common ancestor (LCA) without visiting any nodes more than once. In the worst case scenario, when the binary tree is unbalanced, the height `H` could be linear in respect to the number of nodes, `N`, resulting in a time complexity of `O(N)`. However, in a balanced tree, time complexity is `O(log N)` since the height of the tree is logarithmic in respect to the number of nodes.

The space complexity of the code is `O(1)` regardless of the tree's structure, because it uses a fixed amount of space for pointers and without any recursive calls or additional data structures that depend on the size of the tree.