1650. Lowest Common Ancestor of a Binary Tree III

Medium Tree Hash Table Binary Tree

Problem Description

In this problem, we are given a binary tree where each node not only has references to its left and right children but also a reference to its parent node. We need to find the lowest common ancestor (LCA) of two given nodes in this binary tree. By definition, the LCA is the deepest node that is an ancestor to both nodes. To clarify, a node can be an ancestor to itself according to this problem's definition.

Intuition

The intuition behind the solution to finding the lowest common ancestor (LCA) relies on a simple observation: If we traverse the tree starting from nodes `p` and `q` upwards towards the root by following parent references, the paths will eventually intersect at some node - this node will be the LCA. By traveling up through the parent nodes, both paths must either converge at the LCA or at the root of the tree.

However, since `p` and `q` might not be at the same depth (distance from the root), simply moving upwards will not guarantee that they meet at the LCA. To address this, the solution uses two pointers `a` and `b`, initially pointing to `p` and `q`. They move upwards synchronously by following the parent references. However, if any pointer reaches the root (where the parent reference is `None`) without finding the LCA, it switches to point at the other node (`a` to `q` and `b` to `p`). This pointer switch effectively aligns the traversal paths of `p` and `q` such that after a complete cycle (root to the other node and up again), the paths have the same length, ensuring that `a` and `b` will meet at the LCA.

Solution Approach

To implement the solution for finding the lowest common ancestor, the approach uses two pointers and takes advantage of the parent reference in each node. The algorithm does not require additional data structures or complex patterns - it's a straightforward loop with a condition that exploits the structure of the binary tree.

Here is a step-by-step walkthrough of the implementation:

1. Initialize two pointers, `a` and `b`, both pointing to the given nodes `p` and `q`, respectively.
2. Enter a loop that continues until the pointers meet (`a == b`), which would indicate that the LCA has been found.
3. In each iteration of the loop, move each pointer to its parent. However, when a pointer would move to a `None` reference (indicating it has reached the root), it switches to the other node:
   - Pointer `a` moves to `a.parent` if `a` has a parent, otherwise it switches to point at `q`.
   - Pointer `b` moves to `b.parent` if `b` has a parent, otherwise it switches to point at `p`.
4. The loop exits when `a` and `b` meet, meaning the LCA has been found. Since we switch pointers when we reach the root, both pointers traverse a path that is equal to the sum of the lengths of the paths from `p` to the root and from `q` to the root. Therefore, they are guaranteed to meet at the LCA.

This implementation ensures that both pointers traverse the same total path length, which means they will intersect at the LCA, not before, no matter their initial positions in the tree.

Here is how the code from the Reference Solution Approach embodies the described algorithm:

```
class Solution:
    def lowestCommonAncestor(self, p: 'Node', q: 'Node') -> 'Node':
        a, b = p, q
        while a != b:
            a = a.parent if a.parent else q
            b = b.parent if b.parent else p
        return a
```

The `while` loop represents the traversal and meeting point discovery, and the condition within the loop handles the pointer switching logic. The resultant `a` (or `b`, since they're equal when the loop exits) is returned as the LCA of nodes `p` and `q`.

Example Walkthrough

Consider this simple binary tree as our example:

```
    1
   / \
  2   3
 / \
4   5
```

Node 1 is the root of the tree. Each node except the root has a parent reference. We want to find the lowest common ancestor (LCA) of nodes 4 and 5. According to the problem statement, node 2 is an ancestor of both 4 and 5 and is the lowest one, so it is our expected answer.

Let's walk through the solution approach step-by-step:

1. We initialize two pointers: `a` points to node 4 and `b` points to node 5.
2. We enter the loop:
   1. Both pointers `a` and `b` start moving upwards. For the first iteration:
      - Pointer `a` moves to its parent, which is node 2.
      - Pointer `b` moves to its parent, which is also node 2.
   2. Now, `a` and `b` both point to the same node (node 2), the loop condition `a != b` is not satisfied, and thus we exit the loop.
3. Pointer `a` (or `b`, since they point to the same node) now points to node 2, which is indeed the LCA of nodes 4 and 5.

The pointers met at node 2 without needing to switch to point at the other node, as nodes 4 and 5 are at the same depth. However, if we were to find the LCA of nodes 4 and 3, the switch would occur because the initial paths from nodes 4 and 3 to the root are of different lengths.

As a result, the algorithm works as expected, and we return node 2 as the LCA for nodes 4 and 5.

Solution Implementation

Python

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.parent = None


class Solution:
    def lowestCommonAncestor(self, node_p: 'Node', node_q: 'Node') -> 'Node':
        """Find the lowest common ancestor of two nodes in a binary tree with parent pointers.

        Args:
            node_p (Node): The first node.
            node_q (Node): The second node.

        Returns:
            Node: The lowest common ancestor of node_p and node_q.
        """
        # Start both pointers at the given nodes.
        pointer_a = node_p
        pointer_b = node_q

        # Continue traversing up the tree until both pointers meet at the common ancestor.
        while pointer_a != pointer_b:
            # If pointer_a has a parent, move to the parent; otherwise, go to the other node's initial position.
            pointer_a = pointer_a.parent if pointer_a.parent else node_q

            # Do the same for pointer_b, going to the initial position of node_p if there's no parent.
            pointer_b = pointer_b.parent if pointer_b.parent else node_p

        # Once they meet, that's the lowest common ancestor.
        return pointer_a
```

Java

```
class Solution {
    // This method finds the lowest common ancestor (LCA) of two nodes in a binary tree where nodes have parent pointers.
    public Node lowestCommonAncestor(Node firstNode, Node secondNode) {
        // Initialize two pointers for traversing the ancestors of the given nodes.
        Node pointerA = firstNode;
        Node pointerB = secondNode;

        // Traverse the ancestor chain of both nodes until they meet.
        while (pointerA != pointerB) {
            // If pointerA has reached the root (parent is null), start it at secondNode,
            // otherwise, move it to its parent.
            pointerA = pointerA.parent == null ? secondNode : pointerA.parent;

            // If pointerB has reached the root (parent is null), start it at firstNode,
            // otherwise, move it to its parent.
            pointerB = pointerB.parent == null ? firstNode : pointerB.parent;
        }

        // When pointerA and pointerB meet, we have found the LCA.
        return pointerA;
    }
}
```

C++

```
// Definition for a Node provided by the problem statement.
class Node {
public:
    int value;      // The integer value held by the node.
    Node* left;     // Pointer to the left child of the node.
    Node* right;    // Pointer to the right child of the node.
    Node* parent;   // Pointer to the parent of the node.
};

class Solution {
public:
    // Function to find the lowest common ancestor of two nodes in a binary tree with parent pointers.
    Node* lowestCommonAncestor(Node* firstNode, Node* secondNode) {
        // Create pointers to traverse the ancestor hierarchy of each node.
        Node* currentFirst = firstNode;
        Node* currentSecond = secondNode;

        // Continue looping until the two pointers meet at the lowest common ancestor.
        while (currentFirst != currentSecond) {
            // If currentFirst has a parent, move up the tree, otherwise, jump to secondNode.
            currentFirst = currentFirst->parent ? currentFirst->parent : secondNode;
            // If currentSecond has a parent, move up the tree, otherwise, jump to firstNode.
            currentSecond = currentSecond->parent ? currentSecond->parent : firstNode;
        }
        // When currentFirst and currentSecond meet, we've found the lowest common ancestor.
        return currentFirst;
    }
};
```

TypeScript

```
// Type definition for a Node within a binary tree.
type Node = {
  value: number;   // The integer value held by the node.
  left: Node | null;   // Reference to the left child of the node, if any.
  right: Node | null;  // Reference to the right child of the node, if any.
  parent: Node | null; // Reference to the parent of the node, if any.
};

// Function to find the lowest common ancestor (LCA) of two nodes in a binary tree.
// The tree structure includes parent pointers.
function lowestCommonAncestor(firstNode: Node | null, secondNode: Node | null): Node | null {
  // Initialize pointers to traverse the ancestor hierarchy starting from each node.
  let currentFirst: Node | null = firstNode;
  let currentSecond: Node | null = secondNode;

  // Iterate until the two pointers meet at the LCA.
  while (currentFirst !== currentSecond) {
    // If currentFirst has a parent, move up the hierarchy, otherwise go to secondNode's position.
    currentFirst = currentFirst && currentFirst.parent ? currentFirst.parent : secondNode;
    // If currentSecond has a parent, move up the hierarchy, otherwise go to firstNode's position.
    currentSecond = currentSecond && currentSecond.parent ? currentSecond.parent : firstNode;
  }
  // When currentFirst === currentSecond, we've found the LCA.
  return currentFirst;
}

// Example usage:
// Assuming there exists an established binary tree with parent pointers and two nodes, nodeA and nodeB.
// let lca = lowestCommonAncestor(nodeA, nodeB);
```

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.parent = None


class Solution:
    def lowestCommonAncestor(self, node_p: 'Node', node_q: 'Node') -> 'Node':
        """Find the lowest common ancestor of two nodes in a binary tree with parent pointers.

        Args:
            node_p (Node): The first node.
            node_q (Node): The second node.

        Returns:
            Node: The lowest common ancestor of node_p and node_q.
        """
        # Start both pointers at the given nodes.
        pointer_a = node_p
        pointer_b = node_q

        # Continue traversing up the tree until both pointers meet at the common ancestor.
        while pointer_a != pointer_b:
            # If pointer_a has a parent, move to the parent; otherwise, go to the other node's initial position.
            pointer_a = pointer_a.parent if pointer_a.parent else node_q

            # Do the same for pointer_b, going to the initial position of node_p if there's no parent.
            pointer_b = pointer_b.parent if pointer_b.parent else node_p

        # Once they meet, that's the lowest common ancestor.
        return pointer_a
```

Time and Space Complexity

The given Python function finds the lowest common ancestor (LCA) of two nodes in a binary tree where nodes have a pointer to their parent.

Time Complexity:

The time complexity of the code is `O(h)` where `h` is the height of the tree. This is because, in the worst case, both nodes `p` and `q` could be at the bottom of the tree, and we would traverse from each node up to the root before finding the LCA. Since we are moving at most up to the height of the tree for both `p` and `q`, the time complexity remains `O(h)`.

Space Complexity:

The space complexity of the code is `O(1)`. This is due to the fact that we are only using a fixed number of pointers (`a` and `b`) regardless of the size of the input tree, and no additional data structures or recursive stack space are used.