1666. Change the Root of a Binary Tree

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

Problem Description

In this problem, we are given the root of a binary tree and a leaf node. Our task is to reroot the binary tree so that this leaf node becomes the new root of the tree. The transformation needs to follow certain steps, and they must be applied starting from the leaf node in question towards the original root (but not including it). The steps are:

1. If the current node (`cur`) has a left child, that child should now become the right child of `cur`.
2. The previous parent of `cur` now becomes the left child of `cur`. It's important to ensure here that the old connection between `cur` and its parent is severed, resulting in the parent having at most one child.

The final requirement is to return the new root of the rerooted tree, which is the leaf node we started with. Throughout this process, we must make sure that the `parent` pointers in the tree's nodes are updated correctly, or the solution will be incorrect.

Intuition

The solution follows a simple yet effective approach. Beginning with the leaf node that's supposed to become the new root, we move upwards towards the original root, performing the required transformation at each step:

- We flip the current node's left and right children (if the left child exists).
- We then make the current node's parent the new left child of the current node, ensuring that we cut off the parent's link to the current node.
- We proceed to update the current node to be the former parent, and repeat the process until we reach the original root.

With each step, the `parent` pointer of the nodes is updated to reflect the rerooting process. This ensures that once we reach the original root, our leaf node has successfully become the new root with the entire binary tree rerooted accordingly. Notably, the final step is to sever the `parent` connection of the leaf node, as it is now the root of the tree and should not have a parent.

Solution Approach

To implement the solution, we follow a specific set of steps that involve traversing from the leaf node up to the original root and rearranging the pointers in such a way that the leaf node becomes the new root.

Here's a walkthrough of the implementation using the solution code given:

- We start by initializing `cur` as the leaf node, which is our target for the new root. We will move `cur` up the tree, so we also initialize `p` to be the parent of `cur`.
- We then enter a loop that will continue until `cur` becomes equal to the original root. Inside this loop, we perform the steps needed to reroot the tree:
  - We store the parent of the parent (`gp`) for a later step, as it will become the parent of `p` in the next iteration.
  - If `cur.left` exists, we make it the right child of `cur` by doing `cur.right = cur.left`.
  - We then set `cur.left` to `p`, making the original parent (`p`) the new left child of `cur`.
  - After changing the child pointers, we update the parent pointer by setting `p.parent` to `cur`.
  - We need to ensure that `p` will no longer have `cur` as a child, hence we check whether `cur` was a left or right child and then set the respective child pointer in `p` to `None`.
  - We move up the tree by setting `cur` to `p` and `p` to `gp`.
- After the loop, the original leaf node is now at the position of the new root, but the `parent` pointer for this new root is still set. We must set `leaf.parent` to `None` to finalize the rerooting process.
- The last step is to return `leaf`, which is now the new root following the rerooting algorithm described above.

The main algorithmic concept here is the traversal and pointer manipulation in a binary tree. The traversal is not recursive but iterative, using a `while` loop to go node by node upwards towards the original root. The pointers manipulations involve changes to both child and parent pointers, as per the rerooting steps.

Remember that while the mechanism may appear simple, it is crucial to handle the pointers correctly to avoid any cycles or incorrect hierarchy in the transformed tree structure.

Example Walkthrough

Let's consider a small binary tree and the steps required to reroot it using the solution described.

Original Binary Tree:

```
      a
     / \
    b   c
   / \
  d   e
 /
f
```

Goal: Reroot the tree so the leaf node `f` becomes the new root.

Starting Point: Node `f`.

Using our approach:

1. Initialize `cur` as node `f`. Since `f` has no left child, we do not modify its right child.
2. Set `p` as the parent of `f`, which is node `d`.
3. In the loop:
   - Set `gp` (grandparent) as the parent of `d`, which is node `b`.
   - Since `cur.left` does not exist, we move to the next step.
   - Make `d` the left child of `f` by setting `cur.left` to `p`, so now `f` points to `d` on the left.
   - Set the parent pointer of `d` (`p.parent`) to `f`.
   - Determine if `f` was a left or right child of `d`. In this case, `f` is the left child, so set `d.left` to `None`.
   - Move up the tree: `cur` now becomes `d`, and `p` becomes `b`.
4. Next loop iteration:
   - Set `gp` as the parent of `b`, which is node `a`.
   - Now `cur.left` exists (`e`), so set `cur.right` to `cur.left` (make `d.right` point to `e`).
   - Make `b` the left child of `d` by setting `cur.left` to `p`.
   - The parent pointer of `b` (`p.parent`) is set to `d`.
   - Determine if `d` was a left or right child of `b`. `d` is the left child, so set `b.left` to `None`.
   - Move up: `cur` now is `b`, and `p` is `a`.
5. Final loop iteration:
   - `gp` would be the parent of `a`, which does not exist since `a` is the original root.
   - `cur.left` does not exist (tree doesn't have `b.left` anymore), so no changes to `cur.right`.
   - Make `a` the left child of `b` by setting `cur.left` to `p`.
   - The parent pointer of `a` (`p.parent`) is now set to `b`.
   - Determine if `b` was a left or right child of `a`. `b` is the left child, so set `a.left` to `None`.
   - Since `a` has no parent (`gp`), exit the loop.

Final Step: Set the `parent` of the new root (`f`) to `None` as it should not have a parent.

Resulting Rerooted Binary Tree:

```
      f
     /
    d
   / \
  e   b
       \
        a
         \
          c
```

Now, `f` is the new root, and the tree is correctly rerooted, preserving the left and right subtrees' hierarchy where applicable.

Python Solution

```python
class Solution:
    def flipBinaryTree(self, root: 'Node', leaf: 'Node') -> 'Node':
        # Initialize current node as the leaf node.
        current_node = leaf
        parent_node = current_node.parent

        # Traverse up the tree until we reach the root.
        while current_node != root:
            # Keep reference to the grandparent node.
            grandparent_node = parent_node.parent

            # If current node has a left child, move it to the right side.
            if current_node.left:
                current_node.right = current_node.left

            # Make the parent node a left child of the current node.
            current_node.left = parent_node
            parent_node.parent = current_node

            # Disconnect the current node from the old place in the tree.
            if parent_node.left == current_node:
                parent_node.left = None
            elif parent_node.right == current_node:
                parent_node.right = None

            # Move one level up the tree
            current_node = parent_node
            parent_node = grandparent_node

        # Once the root is reached, we disconnect the leaf from its parent.
        leaf.parent = None

        # The original leaf node is now the new root of the flipped tree.
        return leaf

```

Java Solution

```java
class Solution {
    public Node flipBinaryTree(Node root, Node leaf) {
        // Initialize current node as the leaf node to be flipped up to the new root position.
        Node currentNode = leaf;
        // Initialize parent node as the parent of the current node.
        Node parentNode = currentNode.parent;

        // Iterate until the current node is the original root.
        while (currentNode != root) {
            // Store the grandparent node (parent of the parent node) for later use.
            Node grandParentNode = parentNode.parent;

            // If the current node has a left child, move it to the right child position.
            if (currentNode.left != null) {
                currentNode.right = currentNode.left;
            }

            // Flip the link direction between current node and parent node.
            currentNode.left = parentNode;

            // Update the parent's parent to point back to the current node.
            parentNode.parent = currentNode;

            // Disconnect the current node from its original position in its parent node.
            if (parentNode.left == currentNode) {
                parentNode.left = null;
            } else if (parentNode.right == currentNode) {
                // It must be from the right child position.
                parentNode.right = null;
            }

            // Move up the tree: Set current node to parent, and parent node to grandparent.
            currentNode = parentNode;
            parentNode = grandParentNode;
        }

        // After flipping the tree, the original leaf node has no parent.
        leaf.parent = null;

        // Return the new root of the flipped binary tree (which was the original leaf node).
        return leaf;
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node with an additional parent pointer
class Node {
public:
    int val;        // The value of the node
    Node* left;     // Pointer to the left child
    Node* right;    // Pointer to the right child
    Node* parent;   // Pointer to the parent node
};

class Solution {
public:
    Node* flipBinaryTree(Node* root, Node* leaf) {
        Node* current = leaf;            // Initialize current node to leaf
        Node* parent = current->parent; // Parent node of the current node

        // Continue flipping the tree until we reach the root
        while (current != root) {
            Node* grandParent = parent->parent; // Grandparent node of the current node

            // If there is a left child, make it the right child for flip operation
            if (current->left) {
                current->right = current->left;
            }

            // Connect the parent to the current node's left for flip operation
            current->left = parent;

            // Update the parent's new parent to be the current node
            parent->parent = current;

            // Disconnect the old links from the parent to the current node
            if (parent->left == current) {
                parent->left = nullptr; // Previous left child
            } else if (parent->right == current) {
                parent->right = nullptr; // Previous right child
            }

            // Move up the tree
            current = parent;
            parent = grandParent;
        }

        // Finally, ensure the new root (originally the leaf) has no parent
        leaf->parent = nullptr;

        // Return new root of the flipped tree
        return leaf;
    }
};

```

Typescript Solution

```typescript
// Node class definition for a binary tree with a parent reference.
interface Node {
    val: number;
    left: Node | null;
    right: Node | null;
    parent: Node | null;
}

/**
 * Flips the binary tree so that the path from the specified leaf node to the root becomes the rightmost path in the resulting tree,
 *
 * @param {Node} root - The root node of the binary tree.
 * @param {Node} leaf - The leaf node that will become the new root after flipping.
 * @return {Node} - The new root of the flipped binary tree (which is the leaf node).
 */
var flipBinaryTree = function(root: Node, leaf: Node): Node {
    let currentNode: Node | null = leaf; // Start with the leaf node.
    let parent: Node | null = currentNode.parent; // Get the parent of the current node.

    // Iterate until we reach the root of the initial tree.
    while (currentNode !== root) {
        const grandParent: Node | null = parent ? parent.parent : null; // Save the grandparent node.

        if (currentNode.left !== null) {
            currentNode.right = currentNode.left; // Move the left subtree to the right.
        }
        currentNode.left = parent; // The parent becomes the left child of the current node.

        if (parent) {
            parent.parent = currentNode; // Update the parent's parent to the current node.

            // Disconnect the current node from its former parent node.
            if (parent.left === currentNode) {
                parent.left = null;
            } else if (parent.right === currentNode) {
                parent.right = null;
            }
        }

        // Move up the tree.
        currentNode = parent;
        parent = grandParent;
    }

    if (leaf) {
        leaf.parent = null; // Detach the new root from any parents.
    }

    // The initial leaf node is now the root of the flipped tree.
    return leaf;
};

```

Time and Space Complexity

Time Complexity

The given code processes each node starting from the `leaf` towards the `root`, reversing the connections by making each node's parent its child until it reaches the `root`. Each node is visited exactly once during this process. Therefore, the time complexity is `O(N)`, where `N` is the number of nodes from `leaf` to `root` inclusive, which in the worst case can be the height of the tree.

Space Complexity

The space complexity is `O(1)` since the function only uses a fixed amount of additional space for pointers `cur`, `p`, and `gp` regardless of the input size. The modifications are done in place, so no additional space that depends on the size of the tree is required.