© Yifeng Zeng

Description

156. Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

```
For example:
Given a binary tree {1,2,3,4,5},

    1
   / \
  2   3
 / \
4   5


return the root of the binary tree [4,5,2,#,#,3,1].

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

Idea

The problem description says the original right nodes turned into left leaf nodes. And from the example, we can see 3 (original right node) turned into left leaf node of its sibling (node 2), and their (2 and 3) original parent (node 1) becomes the sibling's (node 2) left node.

```
     1 (parent)
    / \
   2   3 (original right node)
(sibling)
```

```
becomes

     2 (sibling -> new root)
    / \
   3   1  (parent -> new right child)
(original right -> new left child)
```

Similarly, the original right node 5 becomes new left child, sibling node 4 becomes new root, and parent 2 becomes new right child.

```
     2 (parent)
    / \
   4   5 (original right node)
(sibling)

    4 (sibling -> new root)
   / \
  5   2 (parent -> new right child)
(original right -> new left child)
```

So we can have a piece of sudo code:

```
sibling.left = parent;
sibling.right = parent.right;
```

Because we changed the sibling.left and sibling.right, so we need to record the original children before making any changes, then we can have a piece of sudo code before making changes:

```
left = sibling.left;
right = sibling.right;
sibling.left = parent;
sibling.right = parent.right;
```

We are actually making changes of the sibling node, so we can treat sibling as a current node, so for its parent, we need to store it from the last iteration, as well as the originalRight node and also update the sibling.

I believe the core algorithm is the above 4 lines of sudo code, the rest is how we handle the iteration, how we start and end the iteration. It's a very straightforward implementation problem. We only use a few pointers so space complexity is O(1) and the time complexity is O(H) where H is the height of the tree.

Java

```java
class Solution {
    public TreeNode upsideDownBinaryTree(TreeNode root) {
        TreeNode sibling = root;
        TreeNode left = null;
        TreeNode right = null;
        TreeNode parent = null;
        TreeNode originalRight = null;

        while (sibling != null) {
            left = sibling.left;
            right = sibling.right;
            sibling.left = originalRight;
            sibling.right = parent;
            parent = sibling;
            sibling = left;
            originalRight = right;
        }

        return parent;
    }
}
```

C++ code after renaming.

```cpp
class Solution {
public:
    TreeNode* upsideDownBinaryTree(TreeNode* root) {
        TreeNode* left = NULL;
        TreeNode* right = NULL;
        TreeNode* prev = NULL;
        TreeNode* prevRight = NULL;

        while (root != NULL) {
            left = root->left;
            right = root->right;
            root->left = prevRight;
            root->right = prev;
            prev = root;
```

```
            root = left;
            prevRight = right;
        }

        return prev;
    }
};
```

Summary

- Pure implementation, straightforward