156. Binary Tree Upside Down

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

Problem Description

The problem deals with a unique operation on a binary tree named 'upside down transformation.' To understand this transformation, imagine the binary tree made of a set of top-down connected left nodes. The following steps will guide you through this transformation:

1. The leftmost child becomes the new root of the tree.
2. The parent of this new root becomes its right child.
3. The sibling of the new root, which is the original right node, becomes its left child.

This process is then applied recursively to the subtree rooted at the new root (which was the left child of the original tree). What's important to note here is that every right node in the binary tree is always a leaf and has a sibling, which ensures the consistency of the transformation.

Here is what happens during the transformation:

- We start with the leftmost node since it will become the new root.
- We recursively transform the subtree rooted with this node as described.
- We change the pointers of the parent and the sibling to fulfill the requirements of the upside down transformation.

The goal of the exercise is to return the new root after this transformation.

Intuition

The transformation process naturally lends itself to a recursive approach. By following the recursive tree traversal, we can handle the reconfiguration from the bottom up, which avoids dealing with incomplete or inconsistent states.

Here's the intuition behind the solution:

- If the current node is null or it doesn't have a left child, we have reached a base case where nothing needs to change; simply return the current node.
- We apply the transformation to the left subtree and keep the new root returned by the recursive function call.
- Knowing that the transformation rules should apply to one level at a time, we now handle the parent (current root) and its children (leaves, in this context). We attach the parent to the right of the leftmost child which is now the new root and attach the sibling (right child of the root) to its left.
- Finally, the original root's left and right children should point to null to complete the transformation for the current level.

By continuously applying this logic, we process all levels of the tree until we reach the new root, which is then returned.

Solution Approach

The solution to the problem uses a recursive algorithm that follows the depth-first search pattern:

1. The algorithm starts at the root of the binary tree and checks if it's needed to proceed with the transformation. If `root` is `None` or `root.left` is `None`, it indicates that we've either reached the end of a branch or a node that doesn't have a left child (and hence, cannot be flipped). In this case, the `root` itself is returned.

2. Once we confirm that there is a left child to process, we call the same function recursively with the left child of the current root. The expectation is that this call will return the new root for the entire subtree that's rooted at `root.left`.

```
new_root = self.upsideDownBinaryTree(root.left)
```

3. After the recursive call returns the `new_root`, which is the leftmost child now acting as the root of the upside down subtree, we must assign the original root as the right child and the original right child as the left child of the `new_root`.

```
root.left.right = root
root.left.left = root.right
```

4. Once the reassignment of children is done, it's important to detach the original root from its children to prevent cycles and to reflect that it is now a child node without further children.

```
root.left = None
root.right = None
```

5. Finally, we know that the `new_root` has gone through the necessary transformations and has the rest of the tree (that we're not currently looking at) correctly configured. Hence, `new_root` is returned, which, on the final return, will be the new root of the completely transformed tree.

Here's how to visualize the algorithm at every recursive step:

- Identify the leftmost node that will become the new root.
- Make the parent a child of this new root by updating the relevant pointers.
- Remove the parent's original connections.

This recursive approach is efficient because it takes `O(n)` time due to visiting each node once. It's important to note that a recursive approach is managed via the call stack, which could potentially lead to stack overflow on extremely deep trees, although this issue is not common.

This pattern is a particular instance of "tree traversal", where you usually visit each node and perform an operation. In this case, the operation involves changing the pointers within the tree.

Example Walkthrough

Let's consider a binary tree to understand the solution step by step. Imagine we have a binary tree as follows, represented in its original state:

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

Here, '1' is the root, '2' is its left child, '3' is its right child, and so on. According to our 'upside down' transformation rules, '4' will become the new root. The steps of the transformation would be:

1. Using recursion, we go all the way to the leftmost child, which in our example is node '4'. As there are no more left children, this node will be returned as the `new_root`.

2. On the way back up the recursion, we now consider node '2' as the original root with node '4' being the `new_root` returned from the previous recursive call.

   - We transform node '2' so that it becomes the right child of node '4' (since node '4' is the leftmost child and is therefore the new root of the subtree).
   - We assign the original right child of node '2', which is node '5', to be the left child of node '4'.

   After this step, the tree under '4' looks like this:

```
4
 \
  2
   \
    5
```

3. Node '2' is now a child, so its original connections to '4' and '5' should be broken.

4. The recursion then considers node '1' with the `new_root` being node '4'.

   - Node '1' now becomes the right child of node '2'.
   - The original right child of node '1', which is node '3', becomes the left child of node '2'.

5. The connections from node '1' to its children are severed as well. The tree now looks like this:

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

6. The recursion ends as we have traversed all nodes. The `new_root` (node '4') is returned.

The tree has been completely transformed, according to our rules, and the new structure has been achieved via several recursive calls, each modifying the structure according to the 'upside down' rules defined.

Python Solution

```python
# Class definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val        # Node's value
        self.left = left      # Left child
        self.right = right    # Right child

class Solution:
    def upsideDownBinaryTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        # Base case: if the root is None or the root doesn't have a left child,
        # the tree cannot be flipped, so return the root as is.
        if root is None or root.left is None:
            return root

        # Recursive case: dive into the left subtree to find the new root after flipping.
        new_root = self.upsideDownBinaryTree(root.left)

        # Once the recursion unwinds, the original root's left child's right child
        # becomes the original root (making the left child the new parent).
        root.left.right = root

        # The left child's left child becomes the original root's right child.
        root.left.left = root.right

        # Erase the original root's left and right children, since they've been reassigned.
        root.left = None
        root.right = None

        # Return the new root of the flipped tree.
        return new_root

```

Java Solution

```java
class Solution {

    /**
     * Transforms a given binary tree to an upside-down binary tree.
     * In the upside-down version, the original left child becomes the new root,
     * the original root becomes the new right child, and the original right child
     * becomes the new left child.
     *
     * @param root The root of the binary tree to be transformed.
     * @return The new root of the upside-down binary tree.
     */
    public TreeNode upsideDownBinaryTree(TreeNode root) {
        // Base case: if the tree is empty or the root has no left child,
        // the tree does not change.
        if (root == null || root.left == null) {
            return root;
        }

        // recursively call the function on the left subtree.
        // This recursive call goes down to the leftmost child and this node
        // will become the new root of the upside down tree.
        TreeNode newRoot = upsideDownBinaryTree(root.left);

        // The current root's left child's right subtree becomes
        // the current root (One level up).
        root.left.right = root;

        // The current root's right child becomes the new left child
        // of current root's left child.
        root.left.left = root.right;

        // The current root's left and right children are set to null
        // because they have been moved up the tree.
        root.left = null;
        root.right = null;

        // Return the new root, which is the leftmost child of the original tree.
        return newRoot;
    }

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

        TreeNode() {}

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    // Constructor to initialize the node with a value and optional children
    TreeNode(int x = 0, TreeNode *left = nullptr, TreeNode *right = nullptr)
        : val(x), left(left), right(right) {}
};

class Solution {
public:
    // This function converts a given binary tree to an "upside down" binary tree
    // where the original left child becomes the new root.
    TreeNode* upsideDownBinaryTree(TreeNode* root) {
        // If the root is null or the left child is null, return the root itself because
        // no further processing is required (end of the branch or tree is empty).
        if (!root || !root->left) return root;

        // Recursively call function on the left child to handle subtrees and
        // to find the new root of the entire tree.
        TreeNode* newRoot = upsideDownBinaryTree(root->left);

        // The original root becomes the right child of its left child,
        // as per the "upside down" transformation.
        root->left->right = root;

        // The original right child becomes the left child of the new parent,
        // which is the original left child of the root.
        root->left->left = root->right;

        // The original root should not have any left or right children now,
        // so both are set to nullptr.
        root->left = nullptr;
        root->right = nullptr;

        // Return the new root, which is at the bottom-left of the original tree.
        return newRoot;
    }
};

```

Typescript Solution

```typescript
// Type definition for a binary tree node.
type TreeNode = {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
};

// Function that creates a new TreeNode with default values.
const createTreeNode = (val: number, left: TreeNode | null = null, right: TreeNode | null = null): TreeNode => {
    return { val, left, right };
};

// Function to convert a given binary tree to an "upside down" binary tree
// where the original left child becomes the new root.
const upsideDownBinaryTree = (root: TreeNode | null): TreeNode | null => {
    // If the root is null or the left child is null, return the root itself because
    // no further processing is required (it's the end of the branch or the tree is empty).
    if (!root || !root.left) return root;

    // Recursively call the function on the left child to handle subtrees and
    // to determine the new root of the entire tree.
    const newRoot: TreeNode | null = upsideDownBinaryTree(root.left);

    // The current root's left child's right child is set to the current root,
    // following the rules of "upside down" transformation.
    if (root.left) root.left.right = root;

    // The current root's left child's left child is set to the current root's right child.
    if (root.left) root.left.left = root.right;

    // The current root should not have any left or right children now,
    // so both are set to null.
    root.left = null;
    root.right = null;

    // The new root, which is the leftmost leaf node of the original tree, is returned.
    return newRoot;
};

```

Time and Space Complexity

The given Python function `upsideDownBinaryTree` recursively transforms a binary tree into an upside-down binary tree as defined in a specific problem description.

Time Complexity:

The recursion visits each node exactly once. Therefore, the time complexity of the algorithm is based on the number of nodes `n` in the binary tree. Since each call processes constant time work excluding the recursive calls, the overall time complexity can be described as `O(n)`.

Space Complexity:

The space complexity of the algorithm is `O(h)`, where `h` is the height of the binary tree. This is due to the recursive call stack. In the worst case, if the tree is completely unbalanced (e.g., a linked list form), the height of the tree would be `n`, making the space complexity degenerate to `O(n)`. In a balanced tree, the height `h` would be `log(n)`, leading to a space complexity of `O(log(n))`.