226. Invert Binary Tree

Easy | Tree | Depth-First Search | Breadth-First Search | Binary Tree

Problem Description

The given problem is a classic tree manipulation problem which involves inverting a binary tree. In other words, for every node in the tree, you need to swap its left and right children. This operation should be applied recursively to all nodes of the tree, thus flipping the structure of the entire tree. As a result, the leftmost child becomes the rightmost child and vice versa, effectively creating a mirror image of the original tree. The challenge lies not just in performing the swap, but also in traversing the tree correctly to ensure all nodes are covered. Your task is to implement a function that takes the root of the binary tree as input and returns the new root of the inverted tree.

Intuition

To achieve the inversion of the tree, we have to traverse it and for each node visited, its left and right children are swapped. This is a typical use case for a Depth-First Search (DFS) traversal. The DFS algorithm starts at the root node and explores as far as possible along each branch before backtracking. This perfectly suits our need to reach every node in order to invert the entire tree.

The solution approach is recursive in nature:

1. Start with the root node.
2. Swap the left and right child nodes of the current node.
3. Recursively apply the same procedure to the left child node (which after swapping becomes the right child node).
4. Recursively apply the same procedure to the right child node (which after swapping becomes the left child node).
5. Return back to the previous stack call and continue this process until all nodes are visited.

At the end of the recursion, all nodes have their children swapped, and hence the tree is fully inverted, respecting the mirror image condition. Since the inversion needs to happen at each node, the time complexity is O(n), where n is the number of nodes in the tree, because each node is visited once.

Solution Approach

The solution leverages the Depth-First Search (DFS) algorithm to traverse the tree and invert it at each node. To explain this step-by-step:

1. A helper function `dfs()` is defined which will carry out the depth-first traversal and inversion. This function takes one argument: the current `root` node being visited.
2. Inside `dfs()`, a base case is present where if the `root` is `None` (indicating either an empty tree or the end of a branch), the function simply returns as there's nothing to invert.
3. If the node is not `None`, the function proceeds to swap the left and right child nodes.
   - This swapping is done with the Python tuple unpacking syntax: `root.left, root.right = root.right, root.left`.
4. After the swap, `dfs()` is recursively called first with `root.left` and then with `root.right`. Note that after the swap, the original right child is now passed as `root.left` and vice-versa, hence following the inverted structure.
   - These two recursive calls ensure that every child node of the current `root` will also get inverted.
5. The recursion will reach the leaf nodes and backtrack to the root, effectively inverting the subtrees as it goes up the call stack.
6. Finally, once the root node's children are swapped and the recursive calls for its children are done, the whole tree is inverted. `dfs(root)` completes its execution and the modified root node is returned by the `invertTree()` function.

Data structure used:

- A binary tree data structure is utilized with nodes following the definition of `TreeNode` which includes the `val`, `left`, and `right` attributes representing the node's value and its pointers to its left and right children respectively.

Pattern used:

- The pattern is recursion facilitated by DFS which is appropriate for tree-based problems where operations need to be performed on all nodes.

By applying this approach, each and every node in the tree is visited exactly once, and it is guaranteed that the tree will be inverted correctly. The time complexity of this approach is `O(n)` where `n` is the total number of nodes in the tree, as each node is visited once during the traversal.

Example Walkthrough

Let's assume we have a simple binary tree:

```
      1
    /   \
   2     3
  / \   /
 4   5 6
```

We want to invert this tree using the described solution approach. Here's how it happens step by step:

1. We call `invertTree` on the root of the tree (node with value 1). This node is not `None`, so we proceed.
2. Inside the `invertTree` function, we swap the children of node 1. Now the tree looks like this:

```
      1
    /   \
   3     2
  /     / \
 6     5   4
```

3. We call `invertTree` recursively on the left child (node with value 3 which was originally the right child of 1). Node 3 also isn't `None`, so its children (node with value 6) are swapped, but since it's a leaf node with no children, the tree structure remains the same at this point.
4. Next, we proceed to the right child of node 1 which is now the node with value 2. We swap the children of node 2. Now, the binary tree is:

```
      1
    /   \
   3     2
  /     / \
 6     4   5
```

5. The node with value 2's left and right children (4 and 5) are leaf nodes and don't have children to swap. So they're left as is once reached by the recursive calls.
6. After all recursive calls have completed, we have successfully inverted every node in the tree. The final structure of the binary tree is now:

```
      1
    /   \
   3     2
    \   / \
     6 4   5
```

The tree is now a mirror image of its original structure, and each step of our recursive DFS approach allowed us to visit and invert every node in the tree to achieve this. The `invertTree` function would then return the new root of this inverted tree, which completes the process.

Python Solution

```python
class TreeNode:
    """Definition for a binary tree node."""
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        """Invert a binary tree.

        Args:
        root (Optional[TreeNode]): The root node of the binary tree.

        Returns:
        Optional[TreeNode]: The root node of the inverted binary tree.
        """
        def invert(node):
            """Helper function to perform depth-first search and invert the tree.

            Args:
            node (TreeNode): The current node to swap its children.
            """
            if node is None:
                return
            # Swap the left child with the right child
            node.left, node.right = node.right, node.left
            # Recursively invert the left subtree
            invert(node.left)
            # Recursively invert the right subtree
            invert(node.right)

        # Start inverting the tree from the root
        invert(root)
        # Return the root of the inverted binary tree
        return root

```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val; // The value contained in the node
    TreeNode left; // Reference to the left child
    TreeNode right; // Reference to the right child

    // Constructor for creating a leaf node
    TreeNode() {}

    // Constructor for creating a node with a specific value
    TreeNode(int val) { this.val = val; }

    // Constructor for creating a node with a specific value and left/right children
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// A solution class containing the method to invert a binary tree.
class Solution {

    // Inverts a binary tree and returns the root of the inverted tree.
    public TreeNode invertTree(TreeNode root) {
        // Start the depth-first search inversion from the root node
        depthFirstSearchInvert(root);
        // Return the new root after inversion
        return root;
    }

    // A helper method that uses Depth-First Search to invert the given binary tree recursively.
    private void depthFirstSearchInvert(TreeNode node) {
        // Base case: If the current node is null, there's nothing to invert; return immediately
        if (node == null) {
            return;
        }

        // Swap the left and right children of the current node
        TreeNode tempNode = node.left;
        node.left = node.right;
        node.right = tempNode;

        // Recursively invert the left subtree
        depthFirstSearchInvert(node.left);
        // Recursively invert the right subtree
        depthFirstSearchInvert(node.right);
    }
}

```

C++ Solution

```cpp
#include <functional> // Include the functional header for std::function

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructor to initialize the node values
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor to initialize the node values with given left and right children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Method to invert a binary tree
    TreeNode* invertTree(TreeNode* root) {
        // Lambda function to recursively traverse the tree in a depth-first manner and invert it
        std::function<void(TreeNode*)> depthFirstSearch = [&](TreeNode* node) {
            // If the node is null, return immediately as there is nothing to invert
            if (!node) {
                return;
            }

            // Swap the left and right children of the current node
            std::swap(node->left, node->right);

            // Invert the left subtree
            depthFirstSearch(node->left);
            // Invert the right subtree
            depthFirstSearch(node->right);
        };

        // Start depth-first search from the root to invert the entire tree
        depthFirstSearch(root);

        // Return the root of the inverted tree
        return root;
    }
};

```

Typescript Solution

```typescript
// TreeNode class definition
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = (val === undefined ? 0 : val); // Assign the node's value or default it to 0
        this.left = (left === undefined ? null : left); // Assign the left child or default it to null
        this.right = (right === undefined ? null : right); // Assign the right child or default it to null
    }
}

/**
 * Inverts a binary tree by swapping all left and right children.
 *
 * @param {TreeNode | null} treeRoot - The root of the binary tree to invert.
 * @return {TreeNode | null} - The new root of the inverted binary tree.
 */
function invertTree(treeRoot: TreeNode | null): TreeNode | null {
    // Recursive function to traverse the tree and swap children
    function invertNode(node: TreeNode | null): void {
        if (node === null) {
            return; // If the node is null, do nothing
        }
        [node.left, node.right] = [node.right, node.left]; // Swap the left and right children
        invertNode(node.left); // Recursively invert the left subtree
        invertNode(node.right); // Recursively invert the right subtree
    }

    invertNode(treeRoot); // Start inverting from the root node
    return treeRoot; // Return the new root after inversion
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the number of nodes in the binary tree. This is because the function `dfs` is called exactly once for each node in the tree.

The space complexity of the code is also `O(n)` in the worst case, corresponding to the height of the tree. This happens when the tree is skewed (i.e., each node has only one child). In this case, the height of the stack due to recursive calls is equal to the number of nodes. However, in the average case (a balanced tree), the space complexity would be `O(log n)`, as the height of the tree would be logarithmic with respect to the number of nodes.