951. Flip Equivalent Binary Trees

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

Problem Description

In this problem, we are given two binary trees represented by their root nodes `root1` and `root2`. We need to determine if one tree can be transformed into the other tree through a series of flip operations. A flip operation consists of swapping the left and right children of a given node. The trees are considered flip equivalent if one can be transformed into the other by doing any number of flips, possibly zero. We should return `true` if the trees are flip equivalent and `false` otherwise.

Intuition

To solve this problem, we use a recursive approach that is a form of Depth-First Search (DFS). The main idea is that if two trees are flip equivalent, then their roots must have the same value, and either both pairs of the left and right subtrees are flip equivalent, or the left subtree of one tree is flip equivalent to the right subtree of the other tree and vice versa.

To implement this idea, we compare the current nodes of both trees:

1. If both nodes are `None`, then the trees are trivially flip equivalent.
2. If only one node is `None` or the values of the nodes differ, the trees are not flip equivalent.
3. If the nodes have the same value, we then recursively check their subtrees.

We have two cases for recursion to check for flip equivalence:

- Case 1: Left subtree of `root1` is flip equivalent to left subtree of `root2` **AND** right subtree of `root1` is flip equivalent to right subtree of `root2`.
- Case 2: Left subtree of `root1` is flip equivalent to right subtree of `root2` **AND** right subtree of `root1` is flip equivalent to left subtree of `root2`.

If either of these cases is true, we conclude that the trees rooted at `root1` and `root2` are flip equivalent.

This approach works well because it exploits the property that a tree is defined not just by its nodes and their values, but by the specific arrangement of these nodes. By checking all possible flip combinations of subtrees, we can effectively determine flip equivalence between the two given trees.

Solution Approach

The solution code implements a recursive function called `dfs` which stands for depth-first search. This function continuously dives deeper into the subtrees of both trees, simultaneously, for as long as it finds matching node values and structure that adhere to flip equivalence. Below are the aspects of the implementation explained:

- **Base Cases:** The first checks in the `dfs` function handle the base cases. If both nodes are equal, this means that we have reached equivalent leaves, or both nodes are `None`, which means the subtrees are empty and trivially flip equivalent. We thus return `true`. If one of the nodes is `None` while the other isn't, or if the nodes' values are not equal, the function immediately returns `false` indicating the trees are not flip equivalent at this level.

- **Recursive Checks:** The `dfs` function then makes two critical recursive calls representing the two possible scenarios to check for equivalence:

  1. The first scenario checks if the left subtree of `root1` is equivalent to the left subtree of `root2` **and** if the right subtree of `root1` is equivalent to the right subtree of `root2`. This corresponds to the situation where no flips are necessary at the current level of the trees.

  2. The second scenario checks if the left subtree of `root1` is equivalent to the right subtree of `root2` **and** if the right subtree of `root1` is equivalent to the left subtree of `root2`. Here, it matches the case where a flip at the current node would render the subtrees equivalent.

- **Logical OR Operation:** The `or` operation between the two checks is used to state that if either of these scenarios holds true, then the trees rooted at `root1` and `root2` are flip equivalent at the current level.

- **Efficient Short-Circuiting:** The use of the `and` within each scenario creates short-circuiting behavior. This means if the first check within a scenario returns `false`, the second check is not performed, saving unnecessary computation.

The `dfs` function is invoked with the root nodes of the two trees. The return value of this initial call will be the result for flip equivalence of the entire trees. This method effectively uses the call stack as a data structure to hold the state of each recursive call, allowing the algorithm to backtrace through the nodes of the trees and unwind the recursion with the correct answer.

No other explicit data structures are used, which implies that the space complexity of the solution is primarily dependent on the height of the recursion tree, while the time complexity depends on the size of the binary trees being compared.

Example Walkthrough

Let's take a small example of two binary trees to illustrate the solution approach:

Let tree A have the following structure:

```
      1
    /   \
   2     3
  / \   / \
 4   5 6   7
```

And let tree B have a structure that is a flipped version of tree A:

```
      1
    /   \
   3     2
  / \   / \
 7   6 5   4
```

We want to determine if tree B can be transformed into tree A via flip operations.

Application of the Solution Approach:

Step 1: We begin with the `dfs` function call on the root of tree A (which has the value 1), and the root of tree B (also value 1).

Step 2: Since both root nodes' values are identical and not `None`, we check their subtrees, implementing the two scenarios described in the solution approach.

Step 3: We check the left subtree of A (2) with the left subtree of B (3) and the right subtree of A (3) with the right subtree of B (2). Since the values don't match, we proceed to the next step without making any further recursive calls in this scenario.

Step 4: We then check the left subtree of A (2) with the right subtree of B (2) and the right subtree of A (3) with the left subtree of B (3). Here, as we first match the values, we need to delve deeper recursively.

Recursive Step: For each matched pair (2 with 2, 3 with 3), we proceed recursively:

- For the matched pair of nodes with value 2:
  - Compare left child of A's 2 (4) with right child of B's 2 (4).
  - Compare right child of A's 2 (5) with left child of B's 2 (5).
  - Both child pairs match in value; hence, recursion into them will return `true`.
- For the matched pair of nodes with value 3:
  - Compare left child of A's 3 (6) with the right child of B's 3 (6).
  - Compare right child of A's 3 (7) with the left child of B's 3 (7).
  - Again, both child pairs match in value and further recursive checks would yield `true`.

As all recursive checks return `true`, we can say that tree A is flip equivalent to tree B; hence, the initial call to `dfs` from the roots will return `true`, validating the flip equivalence of the two binary trees.

Python Solution

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

class Solution:
    def flipEquiv(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> bool:
        # Helper function to perform depth-first search.
        def is_flip_equivalent(node1, node2):
            # If both nodes are the same, or both are None, trees are flip equivalent.
            if not node1 and not node2:
                return True
            # If one of the nodes is None or values are not equal, trees not flip equivalent.
            if not node1 or not node2 or node1.value != node2.value:
                return False
            # Recursively check if subtrees are flip equivalent:
            # 1. Without flipping children.
            # 2. With flipping children.
            return (is_flip_equivalent(node1.left, node2.left) and \
                    is_flip_equivalent(node1.right, node2.right)) or \
                   (is_flip_equivalent(node1.left, node2.right) and \
                    is_flip_equivalent(node1.right, node2.left))

        # Initiate the depth-first search from the root nodes of both trees.
        return is_flip_equivalent(root1, root2)

```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val;            // Value of the node
    TreeNode left;      // Reference to the left child
    TreeNode right;     // Reference to the right child

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

public class Solution {
    /**
     * Determines if two binary trees are flip equivalent.
     * Flip equivalent binary trees are trees that are
     * the same when either flipped or not flipped at any level of their descendants.
     *
     * @param root1 the root of the first binary tree
     * @param root2 the root of the second binary tree
     * @return true if the binary trees are flip equivalent, false otherwise
     */
    public boolean flipEquiv(TreeNode root1, TreeNode root2) {
        return isFlipEquiv(root1, root2);
    }

    /**
     * Helper method to perform depth-first search to determine flip equivalence.
     *
     * @param node1 the current node being compared in the first tree
     * @param node2 the current node being compared in the second tree
     * @return true if the subtrees rooted at the given nodes are flip equivalent, false otherwise
     */
    private boolean isFlipEquiv(TreeNode node1, TreeNode node2) {
        // Both nodes are null, they are flip equivalent (base case)
        if (node1 == null && node2 == null) {
            return true;
        }

        // If one of the nodes is null, or the values are not equal, they are not flip equivalent
        if (node1 == null || node2 == null || node1.val != node2.val) {
            return false;
        }

        // Recursively check for flip equivalence for both children without flipping and with flipping
        // Check both possibilities: not flipped and flipped
        boolean checkWithoutFlip = isFlipEquiv(node1.left, node2.left) && isFlipEquiv(node1.right, node2.right);
        boolean checkWithFlip = isFlipEquiv(node1.left, node2.right) && isFlipEquiv(node1.right, node2.left);

        // Return true if either possibility resulted in flip equivalence
        return checkWithoutFlip || checkWithFlip;
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;            // The value of the node.
    TreeNode *left;     // Pointer to the left child.
    TreeNode *right;    // Pointer to the right child.
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Check whether two binary trees are flip equivalent.
    bool flipEquiv(TreeNode* root1, TreeNode* root2) {
        return isFlipEquivalent(root1, root2);
    }

    // Helper function to check recursively if two trees are flip equivalent.
    bool isFlipEquivalent(TreeNode* root1, TreeNode* root2) {
        // If both nodes are null, they are flip equivalent.
        if (root1 == root2) return true;

        // If one of the nodes is null or the values don't match, they aren't flip equivalent.
        if (!root1 || !root2 || root1->val != root2->val) return false;

        // Check if children are flip equivalent in two ways:
        // 1. Without flipping (left with left and right with right)
        // 2. With flipping (left with right and right with left)
        return (isFlipEquivalent(root1->left, root2->left) && isFlipEquivalent(root1->right, root2->right)) ||
               (isFlipEquivalent(root1->left, root2->right) && isFlipEquivalent(root1->right, root2->left));
    }
};

```

Typescript Solution

```typescript
interface TreeNode {
  val: number;            // The value of the node.
  left: TreeNode | null;  // Pointer to the left child.
  right: TreeNode | null; // Pointer to the right child.
}

// Check whether two binary trees are flip equivalent.
function flipEquiv(root1: TreeNode | null, root2: TreeNode | null): boolean {
  return isFlipEquivalent(root1, root2);
}

// Helper function to check recursively if two trees are flip equivalent.
function isFlipEquivalent(root1: TreeNode | null, root2: TreeNode | null): boolean {
  // If both nodes are null, they are flip equivalent.
  if (root1 === root2) return true;

  // If one of the nodes is null or the values don't match, they aren't flip equivalent.
  if (!root1 || !root2 || root1.val !== root2.val) return false;

  // Check if children are flip equivalent in two ways:
  // 1. Without flipping (left with left and right with right)
  // 2. With flipping (left with right and right with left)
  return (isFlipEquivalent(root1.left, root2.left) && isFlipEquivalent(root1.right, root2.right)) ||
         (isFlipEquivalent(root1.left, root2.right) && isFlipEquivalent(root1.right, root2.left));
}

```

Time and Space Complexity

The code defines a recursive function `dfs` which checks if two binary trees are flip equivalent. The `dfs` function is called for each corresponding pair of nodes in the two trees. At each step, the code performs constant time operations before potentially making up to two more recursive calls.

The time complexity is `O(N)` where `N` is the smaller number of nodes in either `root1` or `root2`. This is because the recursion stops at the leaf nodes or when a mismatch is found. In the worst case, every node in the smaller tree will be visited.

The space complexity is also `O(N)` due to the recursion stack. In the worst case, the recursion goes as deep as the height of the tree, which can be `O(N)` in the case of a skewed tree (a tree in which every internal node has only one child). For a balanced tree, the space complexity will be `O(log N)` because the height of the tree would be logarithmic relative to the number of nodes.