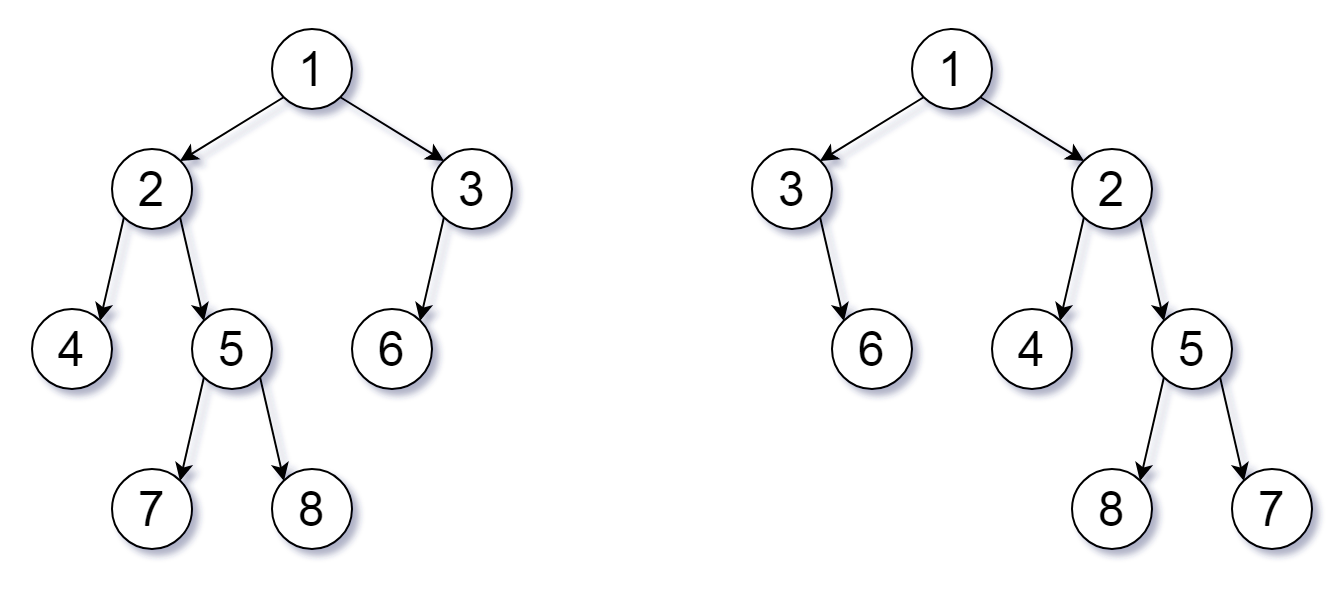
**Flip Equivalent Binary Trees**

For a binary tree **T**, we can define a **flip operation** as follows: choose any node, and swap the left and right child subtrees.

A binary tree **X** is *flip equivalent* to a binary tree **Y** if and only if we can make **X** equal to **Y** after some number of flip operations.

Given the roots of two binary trees root1 and root2, return true if the two trees are flip equivalent or false otherwise.

**Example 1:**



**Input:** root1 = [1,2,3,4,5,6,null,null,null,7,8], root2 = [1,3,2,null,6,4,5,null,null,null,null,8,7]

**Output:** true

**Explanation:** We flipped at nodes with values 1, 3, and 5.

**Example 2:**

**Input:** root1 = [], root2 = []

**Output:** true

**Example 3:**

**Input:** root1 = [], root2 = [1]

**Output:** false

**Constraints:**

* The number of nodes in each tree is in the range [0, 100].
* Each tree will have **unique node values** in the range [0, 99].

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val, left, right) {

\* this.val = (val===undefined ? 0 : val)

\* this.left = (left===undefined ? null : left)

\* this.right = (right===undefined ? null : right)

\* }

\*/

/\*\*

\* @param {TreeNode} root1

\* @param {TreeNode} root2

\* @return {boolean}

\*/

var flipEquiv = function(root1, root2) {

};

Solution

Approach 1: Recursion

**Intuition**

If root1 and root2 have the same root value, then we only need to check if their children are equal (up to ordering.)

**Algorithm**

There are 3 cases:

* If root1 or root2 is null, then they are equivalent if and only if they are both null.
* Else, if root1 and root2 have different values, they aren't equivalent.
* Else, let's check whether the children of root1 are equivalent to the children of root2. There are two different ways to pair these children.

class Solution {

public boolean flipEquiv(TreeNode root1, TreeNode root2) {

if (root1 == root2)

return true;

if (root1 == null || root2 == null || root1.val != root2.val)

return false;

return (flipEquiv(root1.left, root2.left) && flipEquiv(root1.right, root2.right) ||

flipEquiv(root1.left, root2.right) && flipEquiv(root1.right, root2.left));

}

}

**Complexity Analysis**

* Time Complexity: O(min(N\_1, N\_2))*O*(*min*(*N*1​,*N*2​)), where N\_1, N\_2*N*1​,*N*2​ are the lengths of root1 and root2.
* Space Complexity: O(min(H\_1, H\_2))*O*(*min*(*H*1​,*H*2​)), where H\_1, H\_2*H*1​,*H*2​ are the heights of the trees of root1 and root2.

Approach 2: Canonical Traversal

**Intuition**

Flip each node so that the left child is smaller than the right, and call this the *canonical representation*. All equivalent trees have exactly one canonical representation.

**Algorithm**

We can use a depth-first search to compare the canonical representation of each tree. If the traversals are the same, the representations are equal.

When traversing, we should be careful to encode both when we enter or leave a node.

class Solution {

public boolean flipEquiv(TreeNode root1, TreeNode root2) {

List<Integer> vals1 = new ArrayList();

List<Integer> vals2 = new ArrayList();

dfs(root1, vals1);

dfs(root2, vals2);

return vals1.equals(vals2);

}

public void dfs(TreeNode node, List<Integer> vals) {

if (node != null) {

vals.add(node.val);

int L = node.left != null ? node.left.val : -1;

int R = node.right != null ? node.right.val : -1;

if (L < R) {

dfs(node.left, vals);

dfs(node.right, vals);

} else {

dfs(node.right, vals);

dfs(node.left, vals);

}

vals.add(null);

}

}

}

**Complexity Analysis**

* Time Complexity: O(N\_1 + N\_2)*O*(*N*1​+*N*2​), where N\_1, N\_2*N*1​,*N*2​ are the lengths of root1 and root2. (In Python, this is \min(N\_1, N\_2)min(*N*1​,*N*2​).)
* Space Complexity: O(N\_1 + N\_2)*O*(*N*1​+*N*2​). (In Python, this is \min(H\_1, H\_2)min(*H*1​,*H*2​), where H\_1, H\_2*H*1​,*H*2​ are the heights of the trees of root1 and root2.)