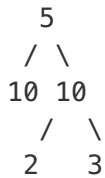


## Equal Tree Partition

Given a binary tree with  $n$  nodes, your task is to check if it's possible to partition the tree to two trees which have the equal sum of values after removing **exactly** one edge on the original tree.

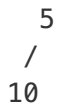
### Example 1:

**Input:**

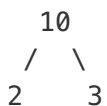


**Output:** True

**Explanation:**



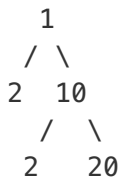
Sum: 15



Sum: 15

### Example 2:

**Input:**



**Output:** False

**Explanation:** You can't split the tree into two trees with equal sum after removing exactly one edge on the tree.

### Note:

1. The range of tree node value is in the range of  $[-100000, 100000]$ .
2. 1

## Solution 1

The idea is to use a hash table to record all the different sums of each subtree in the tree. If the total sum of the tree is `sum`, we just need to check if the hash table contains `sum/2`.

The following code has the correct result at a special case when the tree is `[0, -1, 1]`, which many solutions dismiss. I think this test case should be added.

Java version:

```
public boolean checkEqualTree(TreeNode root) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>();
    int sum = getsum(root, map);
    if(sum == 0) return map.containsKey(sum, 0) > 1;
    return sum%2 == 0 && map.containsKey(sum/2);
}

public int getsum(TreeNode root, Map<Integer, Integer> map ){
    if(root == null) return 0;
    int cur = root.val + getsum(root.left, map) + getsum(root.right, map);
    map.put(cur, map.containsKey(cur, 0) + 1);
    return cur;
}
```

C++ version:

```
bool checkEqualTree(TreeNode* root) {
    unordered_map<int, int> map;
    int sum = getsum(root, map);
    if(sum == 0) return map[sum] > 1;
    return sum%2 == 0 && map.count(sum/2);
}

int getsum(TreeNode* root, unordered_map<int, int>& map){
    if(root == NULL) return 0;
    int cur = root->val + getsum(root->left, map) + getsum(root->right, map);
    map[cur]++;
    return cur;
}
```

written by [Hao Cai](#) original link [here](#)

## Solution 2

Cutting an edge means cutting off a proper subtree (i.e., a subtree but not the whole tree). I collect the sums of these proper subtrees in a set and check whether half the total tree sum is a possible cut.

```
def checkEqualTree(self, root):
    def sum(node):
        if not node:
            return 0
        s = node.val + sum(node.left) + sum(node.right)
        if node is not root:
            cuts.add(s)
        return s
    cuts = set()
    return sum(root) / 2 in cuts
```

Alternatively, I collect all subtree sums in a *list* so that the whole tree's sum is at the end. No need for a hash set's searching speed, as I'm searching the collection only once.

```
def checkEqualTree(self, root):
    def sum(root):
        if not root:
            return 0
        sums.append(root.val + sum(root.left) + sum(root.right))
        return sums[-1]
    sums = []
    sum(root)
    return sums.pop() / 2 in sums
```

Oh, an alternative ending (not sure what I like better):

```
sums = []
return sum(root) / 2 in sums[:-1]
```

Note: I used Python 3. In Python 2, change the `/ 2` to `/ 2.0`.

written by [StefanPochmann](#) original link [here](#)

## Solution 3

```
class Solution {
    boolean equal = false;
    long total = 0;

    public boolean checkEqualTree(TreeNode root) {
        if (root.left == null && root.right == null) return false;
        total = getTotal(root);
        checkEqual(root);
        return equal;
    }

    private long getTotal(TreeNode root) {
        if (root == null) return 0;
        return getTotal(root.left) + getTotal(root.right) + root.val;
    }

    private long checkEqual(TreeNode root) {
        if (root == null || equal) return 0;

        long curSum = checkEqual(root.left) + checkEqual(root.right) + root.val;
        if (total - curSum == curSum) {
            equal = true;
            return 0;
        }
        return curSum;
    }
}
```

written by [shawngao](#) original link [here](#)

From [LeetCoder](#).