Interview Cake

Write a function to see if a binary tree is *"superbalanced"* (a new tree property we just made up).

A tree is "superbalanced" if the difference between the depths of any two leaf nodes is no greater than one.

Here's a sample binary tree node class:

C++

```
class BinaryTreeNode
{
public:
    int value_;
    BinaryTreeNode* left_;
    BinaryTreeNode* right_;

    BinaryTreeNode(int value) :
        value_(value),
        left_(nullptr),
        right_(nullptr)
    {
    }

    ~BinaryTreeNode()
    {
        delete left_;
        delete right_;
    }

    BinaryTreeNode * insertLeft(int value)
    {
        this->left_ = new BinaryTreeNode(value);
        return this->left_;
    }

    BinaryTreeNode * insertRight(int value)
    {
        this->right_ = new BinaryTreeNode(value);
        return this->right_;
    }
};
```

Gotchas

Your first thought might be to write a recursive function, thinking, "the tree is balanced if the left subtree is balanced and the right subtree is balanced." This kind of approach works well for some other tree problems.

But this isn't quite true. Counterexample: suppose that from the root of our tree:

- The left subtree only has leaves at depths 10 and 11.
- The right subtree only has leaves at depths 11 and 12.

Both subtrees are balanced, but from the root we will have leaves at 3 different depths.

We could instead have our recursive function get the vector of distinct leaf depths for each subtree. That could work fine. But let's come up with an iterative solution instead. It's usually better to use an iterative solution instead of a recursive one because it avoids stack overflow.

We can do this in $O(n)$ time and $O(n)$ space.

What about a tree with only one leaf node? Does your function handle that case properly?

Breakdown

Sometimes it's good to start by rephrasing or "simplifying" the problem.

The requirement of "the difference between the depths of any two leaf nodes is no greater than 1" implies that we'll have to compare the depths of *all possible pairs* of leaves. That'd be expensive—if there are n leaves, there are $O(n^2)$ possible pairs of leaves.

But we can simplify this requirement to require less work. For example, we could equivalently say:

- "The difference between the min leaf depth and the max leaf depth is 1 or less"
- "There are at most two distinct leaf depths, and they are at most 1 apart"

If you're having trouble with a recursive approach, try using an iterative one.

To get to our leaves and measure their depths, we'll have to traverse the tree somehow. **What methods do we know for traversing a tree?**

Depth-first and breadth-first are common ways to traverse a tree. Which one should we use here?

The worst-case time and space costs of both are the same—you could make a case for either.

But one characteristic of our algorithm is that it could **short-circuit** and return `false` as soon as it finds two leaves with depths more than 1 apart. So maybe we should **use a traversal that will hit leaves as quickly as possible...**

Depth-first traversal will generally hit leaves before breadth-first, so let's go with that. How could we write a depth-first walk that also keeps track of our depth?

Solution

We do a depth-first walk through our tree, keeping track of the depth as we go. When we find a leaf, we add its depth to an array of `depths` *if* we haven't seen that depth already.

Each time we hit a leaf with a new depth, there are two ways that our tree might now be unbalanced:

1. There are more than 2 different leaf depths
2. There are exactly 2 leaf depths and they are more than 1 apart.

Why are we doing a depth-first walk and not a breadth-first one? You could make a case for either. We chose depth-first because it reaches leaves faster, which allows us to short-circuit earlier in some cases.

C++

```
bool isBalanced(const BinaryTreeNode* treeRoot)
{

    // a tree with no nodes is superbalanced, since there are no leaves!
    if (treeRoot == nullptr) {
        return true;
    }

    // will have up to 3 elements
    size_t depths[3];
    size_t depthCount = 0;

    // nodes will store pairs of a node and the node's depth
    stack<pair<const BinaryTreeNode*, size_t>> nodes;
    nodes.push(make_pair(treeRoot, 0));

    while (!nodes.empty()) {

        // get a node and its depth from the top of stack and pop it
        const BinaryTreeNode* node = nodes.top().first;
        size_t depth = nodes.top().second;
        nodes.pop();

        // case: we found a leaf
        if (!node->left_ && !node->right_) {

            // we only care if it's a new depth
            if (depthCount == 0 ||
                find(depths, depths + depthCount, depth) == depths + depthCount) {
                depths[depthCount] = depth;
                ++depthCount;

                // two ways we might now have an unbalanced tree:
                //   1) more than 2 different leaf depths
                //   2) 2 leaf depths that are more than 1 apart
                if (depthCount > 2 ||
                        (depthCount == 2 &&
                        max(depths[0], depths[1]) - min(depths[0], depths[1]) > 1)) {
                    return false;
                }
            }
```

```
            }
        }

        // case: this isn't a leaf - keep stepping down
        else {
            if (node->left_) {
                nodes.push(make_pair(node->left_, depth + 1));
            }
            if (node->right_) {
                nodes.push(make_pair(node->right_, depth + 1));
            }
        }
    }

    return true;
}
```

Complexity

$O(n)$ time and $O(n)$ space.

For time, the worst case is the tree *is* balanced and we have to iterate over *all* n *nodes* to make sure.

For the space cost, we have two data structures to watch: `depths` and `nodes`.

`depths` will never hold more than three elements, so we can write that off as $O(1)$.

Because we're doing a depth first search, `nodes` will hold at most d nodes where d is the depth of the tree (the number of levels in the tree from the root node down to the lowest node). So we *could* say our space cost is $O(d)$.

But we can also relate d to n. In a balanced tree, d is $O(\log_2(n))$ (/concept/binary-tree#property2). And the *more unbalanced* the tree gets, the closer d gets to n.

In the worst case, the tree is a straight line of right children from the root where every node in that line also has a left child. The traversal will walk down the line of right children, adding a new left child to `nodes` at each step. When the traversal hits the rightmost node, `nodes` will hold *half* of the n total nodes in the tree. Half n is $O(n)$, so our worst case space cost is $O(n)$.

What We Learned

This is an intro to some tree basics. If this is new to you, don't worry—it can take a few questions for this stuff to come together. We have a few more coming up.

Particular things to note:

Focus on **depth-first vs breadth-first traversal**. You should be very comfortable with the differences between the two and the strengths and weaknesses of each.

You should also be very comfortable coding each of them up.

One tip: **Remember that breadth-first uses a queue and depth-first uses a stack** (could be the call stack or an actual stack object). That's not just a clue about implementation, it also helps with figuring out the differences in behavior. Those differences come from whether we visit nodes in the order we see them (first in, first out) or we visit the last-seen node first (last in, first out).