Write a function to check that a binary tree is a valid binary search tree.

Here's a sample binary tree node class:

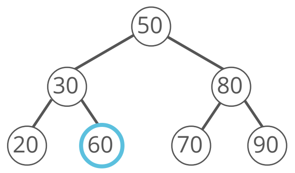
```javascript
function BinaryTreeNode(value) {
    this.value = value;
    this.left  = null;
    this.right = null;
}

BinaryTreeNode.prototype.insertLeft = function(value) {
    this.left = new BinaryTreeNode(value);
    return this.left;
};

BinaryTreeNode.prototype.insertRight = function(value) {
    this.right = new BinaryTreeNode(value);
    return this.right;
};
```

Gotchas

Consider this example:



Notice that when you check the blue node against its parent, it seems correct. However, it's greater than the root, so it should be in the root's right subtree. So we see that **checking a node against its parent isn't sufficient to prove that it's in the correct spot**.

We can do this in $O(n)$ time and $O(n)$ additional space, where n is the number of nodes in our tree. Our additional space is $O(\lg n)$ if our tree is balanced.

Breakdown

One way to break the problem down is to come up with a way to confirm that a single node is in a valid place relative to its ancestors. Then if every node passes this test, our whole tree is a valid BST.

What makes a given node "correct" relative to its ancestors in a BST? Well, it must be greater than any node it is in the right subtree of, and less than any node it is in the left subtree of.

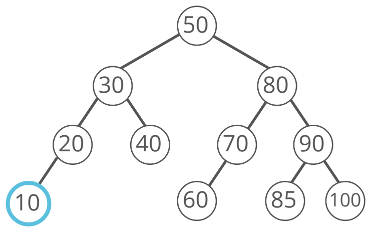
So we could do a walk through our binary tree, **keeping track of the ancestors for each node and whether the node should be greater than or less than each of them**. If each of these greater-than or less-than relationships holds true for each node, our BST is valid.

The simplest ways to traverse the tree are depth-first and breadth-first. They have the same time cost (they each visit each node once). Depth-first traversal of a tree uses memory proportional to the depth of the tree, while breadth-first traversal uses memory proportional to the breadth of the tree (how many nodes there are on the "level" that has the most nodes).

Because the tree's breadth can as much as double each time it gets one level deeper, **depth-first traversal is likely to be more space-efficient than breadth-first traversal**, though they are strictly both $O(n)$ additional space in the worst case. The space savings are obvious if we know our binary tree is balanced—its depth will be $O(\lg n)$ and its breadth will be $O(n)$.

But we're not just storing the nodes themselves in memory, we're also storing the value from each ancestor and whether it should be less than or greater than the given node. Each node has $O(n)$ ancestors ($O(\lg n)$ for a balanced binary tree), so that gives us $O(n^2)$ additional memory cost ($O(n \lg n)$ for a balanced binary tree). We can do better.

Let's look at the inequalities we'd need to store for a given node:



Notice that we would end up testing that the blue node is <20, <30,and <50. Of course, <30 and <50 are implied by <20. So instead of storing each ancestor, we can just keep track of a `lowerBound` and `upperBound` that our node's value must fit inside.

Solution

We do a depth-first walk through the tree, testing each node for validity as we go. A given node is valid if it's greater than all the ancestral nodes it's in the right sub-tree of and less than all the ancestral nodes it's in the left-subtree of. Instead of keeping track of each ancestor to check these inequalities, we just check the largest number it must be greater than (its `lowerBound`) and the smallest number it must be less than (its `upperBound`).

JavaScript

```
function bstChecker(treeRoot) {

    // start at the root, with an arbitrarily low lower bound
    // and an arbitrarily high upper bound
    var nodeAndBoundsStack = [];
    nodeAndBoundsStack.push({node: treeRoot, lowerBound: -Infinity, upperBound: Infinity});

    // depth-first traversal
    while (nodeAndBoundsStack.length) {
        var nodeAndBounds = nodeAndBoundsStack.pop();
        var node = nodeAndBounds.node,
            lowerBound = nodeAndBounds.lowerBound,
            upperBound = nodeAndBounds.upperBound;

        // if this node is invalid, we return false right away
        if (node.value < lowerBound || node.value > upperBound) {
            return false;
        }

        if (node.left) {
            // this node must be less than the current node
            nodeAndBoundsStack.push({node: node.left, lowerBound: lowerBound, upperBound: node.value

        }
        if (node.right) {
            // this node must be greater than the current node
            nodeAndBoundsStack.push({node: node.right, lowerBound: node.value, upperBound: upperBour
        }
    }

    // if none of the nodes were invalid, return true
    // (at this point we have checked all nodes)
    return true;
}
```

Type code!

Instead of allocating a stack ourselves, we could write a **recursive function** that uses the **call stack**. This would work, but it would be **vulnerable to stack overflow**. However, the code does end up quite a bit cleaner:

JavaScript

```
function bstCheckerRecursive(treeRoot, lowerBound, upperBound) {
    lowerBound = lowerBound || -Infinity;
    upperBound = upperBound ||  Infinity;

    if (!treeRoot) return true;

    if (treeRoot.value > upperBound || treeRoot.value < lowerBound) {
        return false;
    }

    return bstCheckerRecursive(treeRoot.left, lowerBound, treeRoot.value) &&
           bstCheckerRecursive(treeRoot.right, treeRoot.value, upperBound);

}
```

Complexity

$O(n)$ time and $O(n)$ additional space, where n is the number of nodes in our tree. Our additional space is $O(\lg n)$ if our tree is balanced.

Did you get it right?

✔ Yes, I'm expert on this ⟳ Not quite, review later

Like this problem? Pass it on!

Share (https://www.facebook.com/sharer/sharer.php?u=https://www.interviewcake.com/question/javascript/bst-checker)

Tweet (https://twitter.com/intent/tweet?text=Solved%20this%20coding%20interview%20question%21&via=interviewcake&relate checker)

Yo, follow along!

(https://www.facebook.com/InterviewCake) (https://twitter.com/interviewcake)

Subscribe to our weekly question email list » (/free-weekly-coding-interview-problem-newsletter)

Programming interview questions by company:

- Google interview questions (/google-interview-questions)
- Facebook interview questions (/facebook-interview-questions)
- Amazon interview questions (/amazon-interview-questions)

Programming interview questions by language:

- Java interview questions (/java-interview-questions)
- Python interview questions (/python-interview-questions)
- Ruby interview questions (/ruby-interview-questions)
- JavaScript interview questions (/javascript-interview-questions)
- **NEW:** Testing and QA interview questions (/testing-and-qa-interview-questions)
- **NEW:** SQL interview questions (/sql-interview-questions)