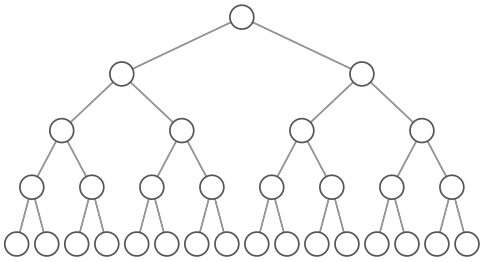
Write a function to check that a binary tree

A **binary tree** is a **tree** where every node has two or fewer children. The children are usually called `@left` and `@right`.

```ruby
class BinaryTreeNode

    attr_accessor :value, :left, :right

    def initialize(value)
        @value = value
        @left  = nil
        @right = nil
    end
end
```

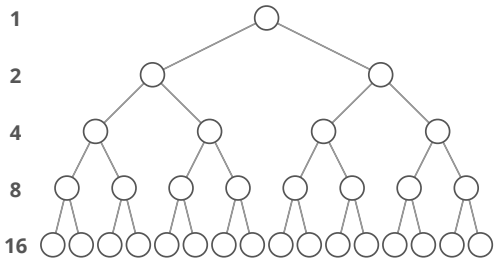
This lets us build a structure like this:



That particular example is special because every level of the tree is completely full. There are no "gaps." We call this kind of tree "**perfect**."

Binary trees have a few interesting properties when they're perfect:

Property 1: the number of total nodes on each "level" doubles as we move down the tree.



Property 2: the number of nodes on the last level is equal to the sum of the number of nodes on all other levels (plus 1). In other words, about *half* of our nodes are on the last level.

Let's call the number of nodes n, and the height of the tree h. h can also be thought of as the "number of levels."

If we had h, how could we calculate n?

Let's just add up the number of nodes on each level! How many nodes are on each level?

If we zero-index the levels, the number of nodes on the xth level is exactly 2^x.

1. Level 0: 2^0 nodes,
2. Level 1: 2^1 nodes,
3. Level 2: 2^2 nodes,
4. Level 3: 2^3 nodes,
5. *etc*

So our total number of nodes is:

$$n = 2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{h-1}$$

> Why only up to 2^{h-1}? Notice that we started counting our levels at 0. So if we have h levels in total, the last level is actually the "$h-1$"-th level. That means the number of nodes on the last level is 2^{h-1}.

But we can simplify. Property 2 tells us that the number of nodes on the last level is (1 more than) half of the total number of nodes, so we can just take the number of nodes on the last level, multiply it by 2, and subtract 1 to get the number of nodes overall. We know the number of nodes on the last level is 2^{h-1}, So:

$$n = 2^{h-1} * 2 - 1$$

$$n = 2^{h-1} * 2 - 1$$

$$n = 2^{h-1} * 2^1 - 1$$

$$n = 2^{h-1+1} - 1$$

$$n = 2^h - 1$$

So that's how we can go from h to n. What about the other direction?

We need to bring the h down from the exponent. That's what logs are for!

First, some quick review. $\log_{10}(100)$ simply means, **"What power must you raise 10 to in order to get 100?"**. Which is 2, because $10^2 = 100$.

We can use logs in algebra to bring variables down from exponents by exploiting the fact that we can simplify $\log_{10}(10^2)$. What power must we raise 10 to in order to get 10^2? That's easy—it's 2.

So in this case we can take the $\log_2$ of both sides:

$$n = 2^h - 1$$

$$n + 1 = 2^h$$

$$\log_2((n+1)) = \log_2(2^h)$$

$$\log_2(n+1) = h$$

So that's the relationship between height and total nodes in a perfect binary tree.

is a valid binary search tree

A **binary search tree** is a **binary tree** in which, for each node:

1. The node's value is greater than all values in the left subtree.
2. The node's value is less than all values in the right subtree.

BSTs are useful for quick lookups. If the tree is **balanced**, we can search for a given value in the tree in $O(\lg n)$ time.

-

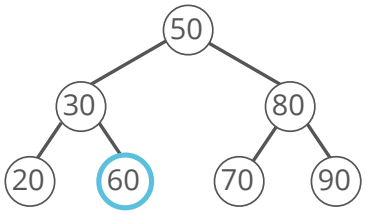
Here's a sample binary tree node class:

Ruby

```
class BinaryTreeNode

    attr_accessor :value
    attr_reader :left, :right

    def initialize(value)
        @value = value
        @left  = nil
        @right = nil
    end

    def insert_left(value)
        @left = BinaryTreeNode.new(value)
        return @left
    end

    def insert_right(value)
        @right = BinaryTreeNode.new(value)
        return @right
    end
end
```

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

> **Depth-first traversal** is a method for walking through a tree or graph where you go as deep as possible down a path before "fanning out." Your set of visited nodes will shoot out from the starting point along one path, with more single paths progressively shooting off of that one as each path hits a dead end.
>
> **Depth-first search** or **DFS** uses depth-first traversal to search for something in a tree or graph.
>
> Depth-first traversal is often compared with **breadth-first traversal**.
>
> Advantages:
>
> - Depth-first traversal on a binary tree *generally* requires less memory than breadth-first.
> - Depth-first traversal can be easily implemented with recursion.

Disadvantages

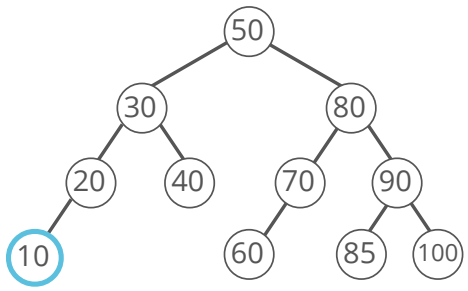
- A DFS doesn't necessarily find the shortest path to a node, while breadth-first search does.

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 `lower_bound` and `upper_bound` 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 `lower_bound`) and the smallest number it must be less than (its `upper_bound`).

```ruby
def is_binary_search_tree(root)

    # start at the root, with an arbitrarily low lower bound
    # and an arbitrarily high upper bound
    node_and_bounds_stack = []
    node_and_bounds_stack.push([root, -Float::INFINITY, Float::INFINITY])

    # depth-first traversal
    while !node_and_bounds_stack.empty?
        node, lower_bound, upper_bound = node_and_bounds_stack.pop()

        # if this node is invalid, we return false right away
        if (node.value <= lower_bound) || (node.value >= upper_bound)
            return false
        end

        if node.left
            # this node must be less than the current node
            node_and_bounds_stack.push([node.left, lower_bound, node.value])
        end
        if node.right
            # this node must be greater than the current node
            node_and_bounds_stack.push([node.right, node.value, upper_bound])
        end
    end

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

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:

```ruby
def is_binary_search_tree(root, lower_bound=-Float::INFINITY, upper_bound=Float::INFINITY)
    if not root
        return true
    end

    if (root.value >= upper_bound || root.value <= lower_bound)
        return false
    end

    return is_binary_search_tree(root.left, lower_bound, root.value) \
        && is_binary_search_tree(root.right, root.value, upper_bound)
end
```

Checking if an in-order traversal of the tree is sorted is a great answer too, especially if you're able to implement it without storing a full list of nodes.

Complexity

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

The time cost is easy: for valid binary search trees, we'll have to check *all* n *nodes*.

Space is a little more complicated. Because we're doing a depth first search, `node_and_bounds_stack` 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 $\log_2 n$ (/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 the stack at each step. When the traversal hits the rightmost node, the stack will hold *half* of the n total nodes in the tree. Half n is $O(n)$, so our worst case space cost is $O(n)$.

Bonus

What if the input tree has duplicate values?

What We Learned

We could think of this as a **greedy**

A **greedy** algorithm iterates through the problem space taking the optimal solution "so far," until it reaches the end.

The greedy approach is only optimal if the problem has "optimal substructure," which means stitching together optimal solutions to subproblems yields an optimal solution.

approach. We start off by trying to solve the problem in just one walk through the tree. So we ask ourselves what values we need to track in order to do that. Which leads us to our stack that tracks upper and lower bounds.

We could also think of this as a sort of "**divide and conquer**" approach. The idea in general behind divide and conquer is to break the problem down into two or more subproblems, solve them, and then use that solution to solve the original problem.

In *this* case, we're dividing the problem into subproblems by saying, "This tree is a valid binary search tree if the left subtree is valid and the right subtree is valid." This is more apparent in the recursive formulation of the answer above.

Of course, it's just fine that our approach *could be* thought of as greedy or *could be* thought of as divide and conquer. It can be both. The point here isn't to create strict categorizations so we can debate whether or not something "counts" as divide and conquer.

Instead, the point is to recognize the underlying *patterns* behind algorithms, so we can get better at thinking through problems.

Sometimes we'll have to kinda smoosh together two or more different patterns to get our answer.