Write a function to find the 2nd largest element in a 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

Our first thought might be to do an in-order traversal of the BST

Sometimes we have a BST and we want to **go through the items in order from smallest to largest**. This is called an "in-order traversal."

We can write a recursive algorithm for this:

- Everything in the left subtree is smaller, so print that first, then,
- print the current node, then
- print the right subtree.

Ruby

```ruby
def inorder_print(node)
    if node
        inorder_print(node.left)
        print node.value
        inorder_print(node.right)
    end
end
```

This takes $O(n)$ time (we're traversing the whole tree, so we're looking at all n items) and $O(h)$ space, where h is the *height* of the tree (this is the max depth of the call stack during our recursion).

If the tree is **balanced** its height is lgn, so we have $O(\lg n)$ space. If we can't make any assumptions about the "shape" of the tree, in the worst case it's just one continuous line, giving us a height of n, and a space complexity of $O(n)$ for our recursive in-order traversal.

and return the second-to-last item. This means looking at *every node in the* BST. That would take $O(n)$ time and $O(h)$ space, where h is the max *height* of the tree (which is lgn if the tree is balanced,

Formally, a tree is said to be **balanced** if the *difference between the depths* of *any node's* left tree and right tree is no greater than 1.

Thus a 'balanced' tree 'looks full', without any apparent chunks missing or any branches that end much earlier than other branches.

but could be as much as n if not).

We can do better than $O(n)$ time and $O(h)$ space.

We can do this in *one* walk from top to bottom of our BST. This means $O(h)$ time (again, that's $O(\lg n)$ if the tree is balanced, $O(n)$ otherwise).

A clean recursive implementation will take $O(h)$ space in the call stack,

The **call stack** is what a program uses to keep track of what function it's currently running and what to do with that function's return value.

Whenever you call a function, a new **frame** gets pushed onto the call stack, which is popped off when the function returns. As functions call other functions, the stack gets taller. In recursive functions, the stack can get as tall as the number of times the function calls itself. This can cause a problem: the stack has a limited amount of space, and if it gets too big you can get a **stack overflow** error.

but we can bring our algorithm down to $O(1)$ space overall.

Breakdown

Let's start by solving a simplified version of the problem and see if we can adapt our approach from there. **How would we find *the largest* element in a BST?**

A reasonable guess is to say **the largest element is simply the "rightmost" element**.

So maybe we can start from the root and just step down right child pointers until we can't anymore (until the right child is `nil`). At that point the current node is the largest in the whole tree.

Is this sufficient? We can prove it is by contradiction:

If the largest element *were not* the "rightmost," then the largest element would either:

1. be in some ancestor node's left subtree, or
2. have a right child.

But each of these leads to a contradiction:

1. If the node is in some ancestor node's left subtree it's *smaller* than that ancestor node, so it's not the largest.
2. If the node has a right child that child is larger than it, so it's not the largest.

So the "rightmost" element *must be* the largest.

How would we formalize getting the "rightmost" element in code?

We can use a simple recursive approach. At each step:

1. If there is a right child, that node and the subtree below it are all greater than the current node. So step down to this child and recurse.
2. Else there is no right child and we're already at the "rightmost" element, so we return its value.

Ruby

```ruby
def find_largest(root_node)
    if !root_node
        raise Exception, 'Tree must have at least 1 node'
    end
    if root_node.right
        return find_largest(root_node.right)
    end
    return root_node.value
end
```

Okay, so we can find the largest element. **How can we adapt this approach to find the *second* largest element?**

Our first thought might be, "it's simply the parent of the largest element!" That seems obviously true when we imagine a nicely balanced tree like this one:

```
.       ( 5 )
       /     \
     (3)     (8)
    /  \     /  \
  (1)  (4) (7)  (9)
```

But what if the largest element itself has a left subtree?

```
.       ( 5 )
       /     \
     (3)     (8)
    /  \     /  \
  (1)  (4) (7)  (12)
                /
              (10)
              /  \
            (9)  (11)
```

Here the parent of our largest is 8, but the second largest is 11.

Drat, okay so the second largest isn't necessarily the parent of the largest...back to the drawing board...

Wait. No. The second largest *is* the parent of the largest *if the largest does not have a left subtree*. If we can handle the case where the largest *does* have a left subtree, we can handle all cases, and we have a solution.

So let's try sticking with this. **How do we find the second largest when the largest has a left subtree?**

It's the *largest* item in that left subtree! Whoa, we freaking *just wrote* a function for finding the largest element in a tree. We could use that here!

How would we code this up?

Ruby

```ruby
def find_largest(root_node)
    if !root_node
        raise Exception, 'Tree must have at least 1 node'
    end
    if root_node.right
        return find_largest(root_node.right)
    end
    return root_node.value
end

def find_second_largest(root_node)
    if !root_node || (!root_node.left && !root_node.right)
        raise Exception, 'Tree must have at least 2 nodes'
    end

    # case: we're currently at largest, and largest has a left subtree,
    # so 2nd largest is largest in said subtree
    if root_node.left && !root_node.right
        return find_largest(root_node.left)
    end

    # case: we're at parent of largest, and largest has no left subtree,
    # so 2nd largest must be current node
    if root_node.right && \
            !root_node.right.left && \
            !root_node.right.right
        return root_node.value
    end

    # otherwise: step right
    return find_second_largest(root_node.right)
end
```

Okay awesome. This'll work. It'll take $O(h)$ time (where h is the height of the tree) and $O(h)$ space.

But that h space in the call stack is avoidable. **How can we get this down to constant space?**

Solution

We start with a function for getting **the largest** value. The largest value is simply the "rightmost" one, so we can get it in one walk down the tree by traversing rightward until we don't have a right child anymore:

```ruby
def find_largest(root_node)
    current = root_node
    while current
        return current.value if !current.right
        current = current.right
    end
end
```

With this in mind, we can also find the *second largest* in one walk down the tree. At each step, we have a few cases:

1. **If we have a left subtree but not a right subtree**, then the current node is the largest overall (the "rightmost") node. The second largest element must be the largest element in the left subtree. We use our `find_largest()` function above to find the largest in that left subtree!
2. **If we have a right child, but that right child node doesn't have any children**, then the right child must be *the largest element* and our current node must be *the second largest element*!
3. **Else, we have a right subtree with more than one element**, so the largest and second largest are somewhere in that subtree. So we step right.

Ruby

```ruby
def find_largest(root_node)
    current = root_node
    while current
        return current.value if !current.right
        current = current.right
    end
end

def find_second_largest(root_node)
    if !root_node || (!root_node.left && !root_node.right)
        raise Exception, 'Tree must have at least 2 nodes'
    end

    current = root_node

    while current
        # case: current is largest and has a left subtree
        # 2nd largest is the largest in that subtree
        if current.left && !current.right
            return find_largest(current.left)
        end

        # case: current is parent of largest, and largest has no children,
        # so current is 2nd largest
        if current.right && \
                !current.right.left && \
                !current.right.right
            return current.value
        end

        current = current.right
    end
end
```

Complexity

We're doing *one* walk down our BST, which means $O(h)$ time, where h is the height of the tree (again, that's $O(\lg n)$ if the tree is balanced, $O(n)$ otherwise). $O(1)$ space.

What We Learned

Here we used a **"simplify, solve, and adapt" strategy**.

The question asks for a function to find the *second* largest element in a BST, so we started off by *simplifying* the problem: we thought about how to find the *first* largest element.

Once we had a strategy for that, we *adapted* that strategy to work for finding the *second* largest element.

It may seem counter-intuitive to start off by solving the *wrong* question. But starting off with a simpler version of the problem is often *much* faster, because it's easier to wrap our heads around right away.

One more note about this one:

Breaking things down into *cases* is another strategy that really helped us here.

Notice how simple finding the second largest node got when we divided it into two cases:

1. The largest node has a left subtree.
2. The largest node *does not* have a left subtree.

Whenever a problem is starting to feel complicated, try breaking it down into cases.

It can be really helpful to actually draw out sample inputs for those cases. This'll keep your thinking organized and also help get your interviewer on the same page.
