Write a function to find the 2nd largest element in a binary search tree.

Here's a sample binary tree node class:

```
class BinaryTreeNode(object):



    def __init__(self, value):

        self.value = value

        self.left  = None

        self.right = None



    def insert_left(self, value):

        self.left = BinaryTreeNode(value)

        return self.left



    def insert_right(self, value):

        self.right = BinaryTreeNode(value)

        return self.right
```

Python 3.6

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.

```
def inorder_print(node):

    if node:

        inorder_print(node.left)

        print(node.value)

        inorder_print(node.right)
```

Python 3.6

This takes O(n)$O(n)$ time (we're traversing the whole tree, so we're looking at all nn items) and O(h)$O(h)$ space, where hh 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 lg(n)$lg(n)$, so we have O(lg(n))$O(lg(n))$ 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 nn, and a space complexity of O(n)$O(n)$ 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)$O(n)$ time and O(h)$O(h)$ space, where hh is the max *height* of the tree (which is lg(n)$lg(n)$ 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 nn if not).
We can do better than O(n)$O(n)$ time and O(h)$O(h)$ space.

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

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

Overview

The **call stack** is what a program uses to keep track of function calls. The call stack is made up of **stack frames**—one for each function call.

For instance, say we called a function that rolled two dice and printed the sum.

```
def roll_die():

    return random.randint(1, 6)



def roll_two_and_sum():

    total = 0

    total += roll_die()

    total += roll_die()

    print(total)



roll_two_and_sum()
```

Python 3.6

First, our program calls `roll_two_and_sum()`. It goes on the call stack:

```
roll_two_and_sum()
```

That function calls `roll_die()`, which gets pushed on to the top of the call stack:

```
roll_die()
roll_two_and_sum()
```

Inside of `roll_die()`, we call `random.randint()`. Here's what our call stack looks like then:

```
random.randint()
roll_die()
```

```
roll_two_and_sum()
```

When `random.randint()` finishes, we return back to `roll_die()` by removing ("popping") `random.randint()`'s stack frame.

```
roll_die()
roll_two_and_sum()
```

Same thing when `roll_die()` returns:

```
roll_two_and_sum()
```

We're not done yet! `roll_two_and_sum()` calls `roll_die()` *again*:

```
roll_die()
roll_two_and_sum()
```

Which calls `random.randint()` again:

```
random.randint()
roll_die()
roll_two_and_sum()
```

`random.randint()` returns, then `roll_die()` returns, putting us back in `roll_two_and_sum()`:

```
roll_two_and_sum()
```

Which calls `print()()`:

```
print()()
roll_two_and_sum()
```

What's stored in a stack frame?

What *actually* goes in a function's stack frame?

A stack frame usually stores:

- Local variables

- Arguments passed into the function
- Information about the caller's stack frame
- The *return address*—what the program should do after the function returns (i.e.: where it should "return to"). This is usually somewhere in the middle of the caller's code.

> Some of the specifics vary between processor architectures. For instance, AMD64 (64-bit x86) processors pass some arguments in registers and some on the call stack. And, ARM processors (common in phones) store the return address in a special register instead of putting it on the call stack.

The Space Cost of Stack Frames

Each function call creates its own stack frame, taking up space on the call stack. That's important because it can impact the *space complexity* of an algorithm. *Especially* when we use **recursion**.

For example, if we wanted to multiply all the numbers between 1 and n, we could use this recursive approach:

```
def product_1_to_n(n):

    return 1 if n <= 1 else n * product_1_to_n(n - 1)
```

Python 3.6

What would the call stack look like when `n = 10`?

First, `product_1_to_n()` gets called with `n = 10`:

```
    product_1_to_n()    n = 10
```

This calls `product_1_to_n()` with `n = 9`.

```
    product_1_to_n()    n = 9
    product_1_to_n()    n = 10
```

Which calls `product_1_to_n()` with `n = 8`.

```
    product_1_to_n()    n = 8
    product_1_to_n()    n = 9
    product_1_to_n()    n = 10
```

And so on until we get to `n = 1`.

```
    product_1_to_n()    n = 1
    product_1_to_n()    n = 2
    product_1_to_n()    n = 3
    product_1_to_n()    n = 4
    product_1_to_n()    n = 5
    product_1_to_n()    n = 6
    product_1_to_n()    n = 7
    product_1_to_n()    n = 8
    product_1_to_n()    n = 9
    product_1_to_n()    n = 10
```

Look at the size of all those stack frames! The entire call stack takes up $O(n)$ space. That's right—we have an $O(n)$ space cost even though our function itself doesn't create any data structures!

What if we'd used an iterative approach instead of a recursive one?

```python
def product_1_to_n(n):

    # We assume n >= 1

    result = 1

    for num in range(1, n + 1):

        result *= num



    return result
```

Python 3.6

This version takes a constant amount of space. At the beginning of the loop, the call stack looks like this:

```
product_1_to_n()    n = 10, result = 1, num = 1
```

As we iterate through the loop, the local variables change, but we stay in the same stack frame because we don't call any other functions.

```
product_1_to_n()    n = 10, result = 2, num = 2
product_1_to_n()    n = 10, result = 6, num = 3
product_1_to_n()    n = 10, result = 24, num = 4
```

In general, even though the compiler or interpreter will take care of managing the call stack for you, it's important to consider the depth of the call stack when analyzing the space complexity of an algorithm.

Be especially careful with recursive functions! They can end up building huge call stacks.

What happens if we run out of space? It's a **stack overflow**! In Python 3.6, you'll get a `RecursionError`.

If the *very last* thing a function does is call another function, then its stack frame might not be needed any more. The function *could* free up its stack frame before doing its final call, saving space.

This is called **tail call optimization** (TCO). If a recursive function is optimized with TCO, then it may not end up with a big call stack.

In general, most languages *don't* provide TCO. Scheme is one of the few languages that guarantee tail call optimization. Some Ruby, C, and Javascript implementations *may* do it. Python and Java decidedly don't.

but we can bring our algorithm down to O(1)*0*(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 `None`). 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.

```
def find_largest(root_node):



    if root_node.right:

        return find_largest(root_node.right)

    return root_node.value
```

Python 3.6

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?

```
def find_largest(root_node):
    if root_node is None:
        raise ValueError('Tree must have at least 1 node')
    if root_node.right is not None:
        return find_largest(root_node.right)
    return root_node.value



def find_second_largest(root_node):
    if (root_node is None or
            (root_node.left is None and root_node.right is None)):
        raise ValueError('Tree must have at least 2 nodes')

    # Case: we're currently at largest, and largest has a left subtree,
    # so 2nd largest is largest in said subtree
    if root_node.left and not root_node.right:
        return find_largest(root_node.left)
```

```
    # Case: we're at parent of largest, and largest has no left subtree,
    # so 2nd largest must be current node
    if (root_node.right and
            not root_node.right.left and
            not root_node.right.right):
        return root_node.value

    # Otherwise: step right
    return find_second_largest(root_node.right)
```

Python 3.6

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

But that hh space in the call stack ↴

Overview

The **call stack** is what a program uses to keep track of function calls. The call stack is made up of **stack frames**—one for each function call.

For instance, say we called a function that rolled two dice and printed the sum.

```
def roll_die():
    return random.randint(1, 6)


def roll_two_and_sum():
    total = 0
    total += roll_die()
```

```
    total += roll_die()

    print(total)


roll_two_and_sum()
```

Python 3.6

First, our program calls `roll_two_and_sum()`. It goes on the call stack:

```
roll_two_and_sum()
```

That function calls `roll_die()`, which gets pushed on to the top of the call stack:

```
roll_die()
roll_two_and_sum()
```

Inside of `roll_die()`, we call `random.randint()`. Here's what our call stack looks like then:

```
random.randint()
roll_die()
roll_two_and_sum()
```

When `random.randint()` finishes, we return back to `roll_die()` by removing ("popping") `random.randint()`'s stack frame.

```
roll_die()
roll_two_and_sum()
```

Same thing when `roll_die()` returns:

```
roll_two_and_sum()
```

We're not done yet! `roll_two_and_sum()` calls `roll_die()` *again*:

```
roll_die()
roll_two_and_sum()
```

Which calls `random.randint()` again:

```
random.randint()
roll_die()
roll_two_and_sum()
```

`random.randint()` returns, then `roll_die()` returns, putting us back in `roll_two_and_sum()`:

```
roll_two_and_sum()
```

Which calls `print()()`:

```
print()()
roll_two_and_sum()
```

What's stored in a stack frame?

What *actually* goes in a function's stack frame?

A stack frame usually stores:

- Local variables
- Arguments passed into the function
- Information about the caller's stack frame
- The *return address*—what the program should do after the function returns (i.e.: where it should "return to"). This is usually somewhere in the middle of the caller's code.

> Some of the specifics vary between processor architectures. For instance, AMD64 (64-bit x86) processors pass some arguments in registers and some on the call stack. And, ARM processors (common in phones) store the return address in a special register instead of putting it on the call stack.

The Space Cost of Stack Frames

Each function call creates its own stack frame, taking up space on the call stack. That's important because it can impact the *space complexity* of an algorithm. *Especially* when we use **recursion**.

For example, if we wanted to multiply all the numbers between 1 and n, we could use this recursive approach:

```
def product_1_to_n(n):

    return 1 if n <= 1 else n * product_1_to_n(n - 1)
```

Python 3.6

What would the call stack look like when `n = 10`?

First, `product_1_to_n()` gets called with `n = 10`:

```
    product_1_to_n()    n = 10
```

This calls `product_1_to_n()` with `n = 9`.

```
    product_1_to_n()    n = 9
    product_1_to_n()    n = 10
```

Which calls `product_1_to_n()` with `n = 8`.

```
    product_1_to_n()    n = 8
    product_1_to_n()    n = 9
    product_1_to_n()    n = 10
```

And so on until we get to `n = 1`.

```
    product_1_to_n()    n = 1
    product_1_to_n()    n = 2
    product_1_to_n()    n = 3
    product_1_to_n()    n = 4
    product_1_to_n()    n = 5
    product_1_to_n()    n = 6
    product_1_to_n()    n = 7
```

```
product_1_to_n()    n = 8
product_1_to_n()    n = 9
product_1_to_n()    n = 10
```

Look at the size of all those stack frames! The entire call stack takes up $O(n)$ space. That's right—we have an $O(n)$ space cost even though our function itself doesn't create any data structures!

What if we'd used an iterative approach instead of a recursive one?

```
def product_1_to_n(n):

    # We assume n >= 1

    result = 1

    for num in range(1, n + 1):

        result *= num



    return result
```

Python 3.6

This version takes a constant amount of space. At the beginning of the loop, the call stack looks like this:

```
product_1_to_n()    n = 10, result = 1, num = 1
```

As we iterate through the loop, the local variables change, but we stay in the same stack frame because we don't call any other functions.

```
product_1_to_n()    n = 10, result = 2, num = 2
product_1_to_n()    n = 10, result = 6, num = 3
product_1_to_n()    n = 10, result = 24, num = 4
```

In general, even though the compiler or interpreter will take care of managing the call stack for you, it's important to consider the depth of the call stack when analyzing the space complexity of an algorithm.

Be especially careful with recursive functions! They can end up building huge call stacks.

What happens if we run out of space? It's a **stack overflow**! In Python 3.6, you'll get a `RecursionError`.

If the *very last* thing a function does is call another function, then its stack frame might not be needed any more. The function *could* free up its stack frame before doing its final call, saving space.

This is called **tail call optimization** (TCO). If a recursive function is optimized with TCO, then it may not end up with a big call stack.

In general, most languages *don't* provide TCO. Scheme is one of the few languages that guarantee tail call optimization. Some Ruby, C, and Javascript implementations *may* do it. Python and Java decidedly don't.

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:

```
def find_largest(root_node):



    current = root_node

    while current:

        if not current.right:

            return current.value

        current = current.right
```

Python 3.6

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.

```
def find_largest(root_node):

    current = root_node

    while current:

        if not current.right:

            return current.value

        current = current.right




def find_second_largest(root_node):

    if (root_node is None or

            (root_node.left is None and root_node.right is None)):

        raise ValueError('Tree must have at least 2 nodes')



    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 and not current.right:
            return find_largest(current.left)


        # Case: current is parent of largest, and largest has no children,
        # so current is 2nd largest
        if (current.right and
                not current.right.left and
                not current.right.right):
            return current.value


        current = current.right
```

Python 3.6

Complexity

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