Interview Cake

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

Here's a sample binary tree node class:

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

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

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

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 `null`). 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.

JavaScript

```
function findLargest(rootNode) {
  if (!rootNode) {
    throw new Error('Tree must have at least 1 node');
  }
  if (rootNode.right) {
    return findLargest(rootNode.right);
  }
  return rootNode.value;
}
```

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?

JavaScript

```
function findLargest(rootNode) {
  if (!rootNode) {
    throw new Error('Tree must have at least 1 node');
  }
  if (rootNode.right) {
    return findLargest(rootNode.right);
  }
  return rootNode.value;
}

function findSecondLargest(rootNode) {
  if (!rootNode || (!rootNode.left && !rootNode.right)) {
    throw new Error('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 (rootNode.left && !rootNode.right) {
    return findLargest(rootNode.left);
  }

  // Case: we're at parent of largest, and largest has no left subtree,
  // so 2nd largest must be current node
  if (
    rootNode.right
    && !rootNode.right.left
    && !rootNode.right.right
  ) {
    return rootNode.value;
  }

  // Otherwise: step right
  return findSecondLargest(rootNode.right);
}
```

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?**
