Largest Smaller BST Key

Given a root of a binary search tree and a key **x**, find the largest key in the tree that is smaller than **x**.

Example: if an in-order list of all keys in the tree is {2, 3, 4, 7, 17, 19, 21, 35, 89} and **x** is 19, the biggest key that is smaller than **x** is 17.

Hints & Tips

- Some programming languages don't have an implementation of a tree data structure. If this is the case with your language of choice, simply use an object / associative array for each node, with a key, left child and right child.
- Some tend to first look for **x** in the tree and then look for its predecessor. However, **x** in not necessarily a key in the given tree, just some key with a given value. Moreover, even if **x** is the tree, finding it first doesn't help.
- To get a 5 stars feedback for problem solving, your peer must be able to explain why it's possible to always store the last key smaller than **x** without comparing it to the previously stored key.
- If your peer is stuck, offer them to think about what they know of binary search trees. If it doesn't help, ask how can this be applied for the solution.

Solution

While the code to solve this question is pretty simple, it takes some understanding of binary trees.

child as the next step, based on a comparison between that node's key and x. If the current node holds a key smaller than x, we proceed to its right sub-tree looking for larger keys. Otherwise, we proceed to its left sub-tree looking for smaller keys.

Finding the key

During this iteration, when the current key is smaller than x we store it as our result and keep looking for a larger key that is still smaller than x.

It's important to understand why we always store the last key without comparing it to the value stored beforehand: if we have stored a key before, we then chose to continue its right sub-tree. Therefore, all following keys will always be larger than and previously stored keys.

```
function findLargestSmallerKey(root, x):
   result = null
   while (root != null):
      if (root.key < x):
         result = root.key
         root = root.right
      else
         root = root.left
   return result
```

Runtime complexity: we scan the tree once from the root to the the leaves and do constant number of actions for each node. if the tree is balanced the complexity is **O(log n)**. Otherwise, it could be up to **O(n)**.