Introduction

We have learned about binary search trees in CIS 120, plus last week in lecture. As a reminder, a binary search tree has the **BST property**, which is that a node’s key is greater than every key in its left subtree, or less than every key in its right subtree.

Definitions

A **BST** is the entire data structure. A **node** is one entry in the BST, along with its associated variables, pointers, and key. A **key** is the data associated with the node. Note that a binary search tree can be used in several contexts, such as a set, a map, or a multi-set. In this light, a BST being used as a map would have both keys and values.

Parent Pointers

In Java, it is convenient to have a BST node point back to its parent instead of just its left child and right child. This permits iterative traversals not just down the tree, but also back up the tree. It requires just a bit of extra bookkeeping, which we will briefly explore in the next exercise.

Java representation

How might we represent a BST in Java?

Insertion with Parent Pointers

How would we do BST insertion with parent pointer bookkeeping? **Hint**: do BST insertion as normal, then worry about the parent pointer addition at the end.

Successor

The successor of a key $x$ in a BST is the next-largest key in the BST (or null if $x$ is already the maximum key). Write a method to find the successor.

You should be able to work out, via symmetry, how to find the predecessor of a node in a BST.

Spiral-Order Traversal

In CIS 120, we learned about four types of tree traversals: in-order, pre-order, post-order, and level-order. We can introduce a new type of traversal, **spiral-order**, in which the nodes at level 0 (i.e. the root) are traversed left-to-right, the nodes at level 1 are traversed right-to-left, the nodes at level 2 are traversed left-to-right, etc. Implement the spiral-order traversal in Java.

In-Order Traversal

The in-order traversal of a BST is such that you print all of the left children of a node, then the node itself, then all of the right children of a node.
**Recursive Solution**
Implement the in-order traversal recursively.

**Iterative Solution**
Recursion is not always an appropriate solution in Java, namely if we have a very deep tree. Implement the in-order traversal iteratively. (The pseudocode for this is going to get messier, unfortunately.) Since we have parent pointers, do not use a stack.

**Bonus: Deletion with Parent Pointers**
How would we do BST deletion with parent pointer bookkeeping? (Save this exercise for the end of lab, if time permits.)