Homework # 9
due Monday, April 8, 10:00 PM

In the previous assignment, we implemented the HexBoard ADT using a binary search tree (BST), but left removal undone, and used a sub-optimal iterator representation. In this assignment, we will update the iterator data structure to use a stack of "pending" nodes, and also add removal capability to the ADT.

1 Concerning the Iterator Data Structure

We will be using the "Stack" technique for iterating over the binary search tree as explained in the Navigating Trees handout.

The iterator data structure will be changed to use a stack of nodes. It will also keep track of the current value, in case the client wishes to remove it. The data structure has a lot of redundancy which will be checked with a `wellFormed` method in the iterator class.

The iterator's `wellFormed` will call the outer classes `wellFormed` method. Doing so requires Java syntax that may be unfamiliar: *Class*`.this.`*method*`(...)`. After checking the outer class' invariant, the iterator will see if it (the iterator) is stale. if the iterator is stale, then any problems with its representation may be due to inconsistent changes in the main class which are *not* the responsibility of the iterator class. Recall that we check the data structure to protect against errors of *implementation*. A stale iterator shows errors of *use*, not implementation. Make sure that the developer is not blamed for errors by the client! Our solution clones the pending stack so that the copy can be popped while checking things.

The nodes in the stack should represent a path down the tree in which only the 'GT' ancestors are given: only the *greater* ancestors are given (the ones whose keys come after those of later nodes). The first (deepest) node must be a node without any GT ancestors. The current value if not null should be the value in the immediate predecessor of the top node of the stack (if the stack is not empty) or the last value in the tree (if the stack is empty).

2 Concerning Removal

If the node being removed has a left child, then the immediate predecessor can be used to replace this node's data. Otherwise (if there is no left child), the right child (if any) can replace this node entirely. Draw pictures to help you handle the replacement correctly.

Collections have two ways to remove elements: one through a method of the collection class and one through the iterator. Removal code is complex and thus it would be poor software construction to implement this capability twice. You should arrange one of the ways to delegate to the other, while ensuring that the iterator's version stays up-to-date. One of the directions of delegation will work better (more efficiently) than the other; we want you to find the better one.

If you use a loop to implement removal, you will need to have three cases for node removal (at the root, a left chid or a right child). For that reason, we recommend you use a recursive

helper method that is only called when you know that the hextile to remove is actually present.

3 The Test Suite

We provide a full complement of tests:

TestHexBoard Our ADT test suite.

TestInvariantChecker Updated data structure checker tester.

TestEfficiency Updated efficiency tests.

RandomTest The random test now includes removal. As before, it is available in the JAR.

4 What you need to do

You need to

- Implement removal.
- Implement the invariant data structure checker.
- Update the iterator methods.