Homework # 8
due Monday, April 1, 10:00 PM

In this homework assignment, you will re-implement the HexBoard ADT (with some changes) using a binary search tree (BST) data structure. A BST permits an efficient lookup mechanism (as opposed to linear search) so there is an efficiency test to ensure your code can handle half a million entries.

1 The Binary Search Tree Data Structure

Please read section 9.5 in the textbook for a description of the binary search tree data structure. Alternatively, there are many web-pages/lecture notes on BST, the wiki page may be a good starting point (https://en.wikipedia.org/wiki/Binary_search_tree). In the textbook (as well as some online sources), a separate `BTNode` class is used; we will *not* do that. Use a nested node class as before.

Linked lists and arrays support insertion, removal or finding an entry in time proportional to the number of entries in the container. Trees, on the other hand, offer a significant efficiency advantage in that they generally support these operations in time proportional to the *log* of the number of entries (assuming the tree is kept balanced).

In order to achieve the potential time efficiency of binary search trees, one needs to keep the tree roughly balanced. In CompSci 535, you will learn some of the techniques used to do this (as it happens, the tree-based containers in Java's libraries use "red-black" trees). But in this course, we will ignore the problems of unbalanced trees. Do not make any attempt to re-balance your tree. The efficiency tests we run will make sure to construct a balanced tree.

2 Concerning the Modified HexBoard ADT

As with previous assignments, the hex board ADT keeps track of a terrain for each hex coordinate as well as also being a collection of hex tiles (created on demand). We are making two changes in the ADT (as well as changing the implementation!):

- Elements are kept in sorted order.
- Removal is not supported.

Previously, we used a built-in hash map to store the information. Hash maps store information in a somewhat unpredictable order, as we shall see in a later homework assignment, but in this assignment, we define an order on hex coordinates and the iteration order of the hex tiles will reflect this order.

For this homework assignment, we do not support removal. That is in order to reduce the complexity of the implementation. Removal will be added in the following homework assignment.

The efficiency tests check to see that you built the tree correctly. If you wrote the code efficiently, this test shouldn't take more than 15 seconds or so. If you wrote the inefficient (but easy) technique, the test will take much longer and you will lose points.

3 Private Helper Methods

The implementation will make use of several helper methods, some of which must be recursive:

compare(h1,h2) Compare two hex coordinates so that it does one full row before the next one in order. That means that the `b()` coordinate (the row) must be checked first, and then `a()` if the rows are the same. This method can be implemented with a single "if" statement.

getFirst/LastRow() Get the number ("b") of the first (least) / last (greatest) row of any hex tile on the board. Return zero if the board is empty.

getFirst/LastInRow(int) Return the hex tile furthest to the left/right in the given row on the board, or null if there are no hex tiles in that row.

isInProperOrder(r,lo,hi) Check that all nodes in the subtree rooted at "r" have valid (non-null) coordinates that are in the exclusive range given. They should also have non-null terrains. The bounds ("lo" or "hi") can be null which means no bound. So if "lo" is null, it means that there is no lower bound for the data in the subtree. The null subtree satisfies any bounds since it has no nodes in it.

countNodes(r) Return the number of nodes in the given (acyclic) subtree. A null subtree has zero nodes. Make sure you don't call it on a tree that might have a cycle.

The first one is used by all the methods that need to figure out what order to use. The next few are used by the iterator, and the last two are important for the invariant checker.

4 Concerning the iterator

In order to make this homework easier, the iterator for this assignment will not work with nodes. Rather it will just use `terrainAt` on a series of coordinates to find hex tiles.

The iterator will iterate over one row at a time starting from the first row (`getFirstRow`) up to the last row. For each row, it will start with the first hex tile in the row (using `getFirstInRow`) and then go up to the last one, checking each coordinate. It should keep track of the current row ("b") it's working on, and also the next column ("a") to check (as long as it's not past the last hex tile in the row). It will need to pass a lot of empty spots. Make sure you don't return null for the empty spots.

5 What you need to do

You need to implement the helper methods, and define the fields needed to implement the data structure, plus a "version" field for fail-fast iteration. You need to implement the `wellFormed` method to check the invariant, and implement the constructor and the mandatory methods for a collection or a hex board.

Then you need to implement the other methods needed in the class You do not need to implement any "remove" methods.

We provide a modified test case for the ADT, a new invariant test and a modified efficiency test. We also provide random testing.