Homework # 11
due Monday, April 22, 10:00 PM

In this assignment, you will re-implement HexBoard one last time using a simple hash table with chained (bucket) hashing. In this assignment, we come back full circle in a way, because our original implementation used a hash-table from the library. This time we are doing our own implementation. We follow the previous assignment in giving a Map view of the hex board, but will *not* have the "row" view. Unlike a binary search tree, the hash table data structure does not provide an efficient way to implement a row subset.

1 Concerning the Hash Table Data Structure

For this assignment, you will implement a hash table implementation of a set of hex tiles so that we can check if a tile has already been placed on a hex coordinate by using the hex coordinate's hash code.

Please read the textbook, especially sections 11.2–11.4, pages 581–602 (3rd ed., pp. 569–590). We will be using the technique of "Chained Hashing" on page 600 (3rd ed., p. 588), with a linked list of map entries. We will hash the hex coordinate of the hex tile, not the hex tiles themselves, because of the need to implement `terrainAt`.

In order to remain consistent with the test suites' expectations, you should use a "mod" approach to handle negative hash code values (-2 mod 7 = 5) rather than taking the absolute value of the remainder. Furthermore, you should always add new elements at the *head* of the linked list. (It's easier to add at the head anyway!)

Unlike the textbook, our hash table will increase the array size when the table has many elements in it. It will start as size seven, and then whenever the table would be three-quarters full, we will increase the array size to the next prime that is larger than twice the current capacity. On the other hand, if the table ever gets empty, the array should be reverted to the initial size. We don't shrink the table otherwise, because it's hard to handle iteration if the elements move around.

Once a new array is allocated, all entries need to be placed in their correct places, which are possibly different than before because the array has changed size. Go through the array in order, and each chain in order, replacing each entry. As mentioned before, always add entries at the *front* of the chain.

The invariant of the data structure is that

1. The array is not null;
2. The number of entries is correctly summed in the "size" field;
3. The length of the array is a prime number;
4. The array is neither too full nor too empty;
5. Every entry is in the correct chain;
6. The chains contain no duplicates (which also prevents loops).

You will need to be careful to check these properties in a particular order to make sure that an undetected problem doesn't cause a check for a different problem to get stuck in an infinite loop.

2 Concerning the Views of the HexBoard

As previously seen, a `HexBoard` can be viewed as a Set of hex tiles, as a Map from hex coordinates to terrains and finally as a Set of entries for the map. We will keep these views for this assignment. As before, we use `AbstractSet` and `AbstractMap` to provide default implementations for most of the methods. We will need to override several of the methods for functionality or efficiency. You are encouraged to reuse any of the code from the solution to Homework #10 that is useful, especially the code that doesn't reference the data structure directly.

We need an iterator over the hex board directly (returning tiles) and over the entry set (returning entries). But it is poor software practice to implement the iteration algorithm more than once. So, then similarly as was done in the previous assignment, you should implement the entry set iterator so that it iterates over the elements in order in the hash table, and then implement the hex board iterator to use the entry set iterator for the "real" work. Since each entry will actually be a linked list node, it can be cast to this internal node and the hex tile from the node returned.

The iterator should return the elements in order over the array starting at zero, and within each chain, from the head to the tail. You are left free to implement the iterator as you feel makes the most sense to you, but it must be reasonably efficient, and obey the usually restrictions. We do not require an invariant for the iterator, but you are welcome to create one.

3 What You Need To Do

You need to complete HexBoard.java (including its invariant), and its nested classes.

4 Files

In your repository, you will find the following:

src/edu/uwm/cs351/HexBoard.java Skeleton of `HexBoard`.

src/TestHexBoard.java Test cases for the ADT.

src/TestInternals.java For testing the data structure invariant.

src/TestEfficiency.java Testing efficiency of the ADT.

src/edu/uwm/cs/util/Primes.java Utility class for generating and checking primes.

Random testing is available for this homework, but it doesn't test the Map and Set views.