Homework #10
due Monday, April 15, 10:00 PM

In this assignment, you will extend the HexBoard ADT to be viewable as a map and as an entry set, and also to permit a row of hex tiles to be seen as a collection in its own right.

1 Multiple Views

The HexBoard ADT is an extension of the standard Collection ADT (or more precisely, the Set ADT, which is made explicit in this homework assignment).

A hex board has additional functionality beyond that of Set: we have the `terrainAt` method. Indeed, a hex board could be seen as a map from hex coordinates to terrains. Sometimes, such a conceptual idea needs to be formalized to allow the ADT to be used in new contexts. The contexts may expect that the ADT obeys the standard Map interface (see below). So in this homework, we will add the ability to view the contents of a hex board through the lens of a Map.

The same data structure will be used for both ADT views: Set and Map. Changes through either view should affect the (shared) data structure. We say that the map is *backed* by the (data structure of) the hex board. In particular, the map will not have *any* of its own fields, and certainly not any redundant information that could get inconsistent. Unlike an iterator, the map should *never* go stale.

In our case, the data structure is a binary search tree. The Map interface has methods that will need to operate on that data structure. You should make sure that you don't implement the same algorithm in multiple places: this is likely to multiply bugs. Rather you should make sure one implementation can be used by the other(s).

In this homework assignment, you are going to add further view, one which has limited access to the data structure. The `row` method will return a set that is backed by the main data structure. Again, any changes to the row set will be reflected in the main ADT and vice versa. The row has no redundant information, and will never be stale. Again, this sort of view allows a client to connect the ADT with different tools that expect standard interfaces. And again, you should not implement minor variations on BST algorithms over; make sure to reuse (not copy!) code.

2 The Standard Map interface

The Java collection framework defines a generic Map interface that gives table-like functionality. The interface Map has a large number of methods:

size() Returns the number of entries in the map.

isEmpty() Returns whether the map is empty (has no entries).

containsKey(k) Returns true if there is an entry for the given key.

containsValue(v) Returns true if there is an entry whose value matches the given value.

get(k) Returns the value associated with the given key, or null if there is no entry.

put(k,v) Add an entry (if no entry for key) or modify the existing entry. Throws an exception if key is null. Return the *previous* value associated with this key, or null.

remove(k) Remove the entry for this key (if any). Return the *previous* value associated with this key, or null.

putAll(m) Add all the entries from the parameter map to this map.

clear() Remove all the entries.

keySet() Return the Set holding the keys of this map.

values() Return the Collection holding the values of all the entries in this map.

entrySet() Return this map viewed as a Set of entries.

As usual, there is a class `AbstractMap` that implements many of these operations in terms of the entry set, but we override some like `get` for efficiency and others like `put` because they throw an "unsupported operation" exception.

Several of the methods which might normally be expected to take a parameter of key type (`get`, `containsKey` and `remove`) instead take a parameter of type `Object`, which might be of any conceivable type or even null. In no case should these methods crash, even if the parameter is null or of a non-key type.

The `put` and `remove` methods of the Map ADT return the *previous* value, or `null` if there was no previous entry for this key. Note that the Map's semantics imply that keys will be unique (as in previous Homework). We will not be able to associate two different values with a single key.

2.1 Concerning the Entry Set of Map

The *entry set* of a map is a set of associations (instances of `Map.Entry<K,V>`) that is backed by the map. Again that means that the entry set will not have its own fields, and so will never go stale. Instead it will use the shared data structure.

Maps also have "key sets" and "value collections." The implementations for these latter sets in the abstract class `AbstractMap` uses the entry set to do the real work. So you can ignore them.

In the implementation of an entry set, one usually uses the abstract class `AbstractSet` to do most of the work. As with `AbstractCollection`, the `add` method of this method just throws an "unsupported operation" exception. Previously, one needed to override this behavior, but entry sets are not required to implement "add" behavior because a map cannot have two entries with the same keys and different values.

It also has `contains` and `remove` methods that take a parameter of type `Object`. If you need to override the default behavior (using the iterator), you will need to first check whether the parameter is an entry object at all: use type `Map.Entry<?,?>`, and check the types of the key and value objects. Don't try to check whether it is an entry of a particular type because erasure prevents that from being checkable. You just get warnings and then your code will crash later on.

Of course the entry set has an iterator, which returns entry objects.

3 Implementation Considerations

Every new public method you add should check the invariant. The invariant checking is done for the developer's (your!) benefit.

We encourage you to change the definition of the nodes used in the BST so that each node implements the Entry interface. The reason is that the client may get an entry and change the value to a new value. If the entry is actually the node, this behavior is easy to implement. We

provide a class `AbstractEntry` which makes it easy to bolt on entry behavior; you will still need to implement the getters and setters. There is no setter for the key because Entry recognizes that changing the key may invalidate an entry.

We don't want to define two or three classes that each need to navigate around a binary search tree. Since the entry set iterator is the most general (it will actually be returning nodes), it is easier to use an entry set iterator to construct a hex board iterator (return hex tiles) than the other way around. Thus we recommend that you repurpose the code for the hex board iterator for the entry set iterator. Then the hex board iterator can just use the entry set iterator to do the heavy lifting, and just adjust the result returned by `next()`.

When we get to the row iterator, we will need to start (and end) an iterator in the middle of the tree. The easiest way to handle this is to give the entry set iterator the ability to start in the middle, by giving a starting row. This can be done as an alternate constructor. Ending early is most easily accomplished if the entry set iterator also has a way to "peek" the next value (or check if the next entry has a coordinate on a particular row).

When creating each new view, it is recommended to start with `AbstractSet` or `AbstractMap` (as appropriate) and only override implementations when necessary: first those needed to avoid compiler errors, then those needed for functionality and only later those needed for efficiency. Use Eclipse's "Source > Override/Implement Methods" to make sure that you override correctly.

The row set in particular doesn't have an efficient way to count up the elements. To get good efficiency, we'd need to update the data structure, and for this homework assignment we are not changing the data structure. So, find an easy (albeit inefficient) way to implement `size()`. Then you will need to come up with a better way to *easily* determine if the set is empty, since the implementation in `Abstract` set uses `size()`.

4 Many Ways to Remove

In this assignment, we will have many different ADTs all working with the same shared data structure:

- The HexBoard, a Set of HexTile objects.
- The HexBoard's Iterator over HexTile objects.
- The view as a Map from HexCoordinate to Terrain.
- The entry set of the Map view, a Set of Entry objects from HexCoordinate to Terrain.
- The iterator of the entry set over Entry objects from HexCoordinate to Terrain.
- The row set, a Set of HexTile objects.
- The iterator over the row set, an Iterator over HexTile objects.

Each of these views has a `remove` method, some of which take an argument of type `Object`. The iterator `remove` methods don't take an argument and just remove the most recently returned value (from `next`). Of those taking an argument, two work only if passed a hex tile, one only if passed a hex coordinate and the last only if passed an object satisfying the Entry ADT. But your code should only have one method that removes something from the binary search tree. So each of the methods taking an argument should first check the argument's type and then figure out how to call the hex board's remove method.

5 What You Need To Do

Here is a recommend order of updating the code from the previous assignment.

1. Make sure that the code passes all tests except the new ones: `test` (locked), `testMnn` (map) and `testRnn` (row) tests. As you make further changes, make sure that the original tests still pass (we want to avoid going backwards, project *regression*, so these tests are called "regression tests").

2. Change the `Node` class to be an entry by using `AbstractEntry`. Run regression tests.

3. Repurpose the existing `MyIterator` class into an `EntrySetIterator` class which implements an iterator over `Entry<HexCoordinate,Terrain>`. Most of the class will be unchanged, but you will need to change `next()` to return a node (now that noes are entries!) instead of hex tiles. Then write `MyIterator` anew to simply create an instance of an entry set iterator and delegate all the real work to that iterator (`hasNext` and `remove` can delegate completely to the base iterator). Only `next()` will need to do a little work to construct a hex tile on the fly. Once this done, make sure you pass all the regression tests.

4. Create an `EntrySet` nested class for the entry set (use `AbstractSet` with the correct type paramater) and implement `size()` (easy) and `iterator()` (just use the new class you created!). Then create a `MyMap` nested class that extends `AbstractMap` with the correct type parameters, and implements the `entrySet()` method to create an instance of the entry set class. Make sure you have no "raw" type warnings or any type errors. Once this is working, change the stub in `asMap` to create an instance of your map class. **Please** comment on all overrides why the override is happening! Make sure you still pass the regression tests, and now you can see how many of the `testM` tests you pass!

5. You will find that you need to implement one or two methods for functionality, with code that delegates to an already implemented method in the main class. After this, your code should now pass all the `testM` tests. If you find failures, look at what operation is happening, and trace to see where perhaps you neglected to do something.

6. Once the map stuff is all working, we turn to the "row" part of the assignment: Add a second constructor to the entry set iterator to start at a particular row. It should do something similar to what we did when finding the "first in row" but this time push nodes onto the pending stack whenever we go left. You should also add a version of `hasNext` that allows a client to determine if there is a next *on a particular row*. Then add a second constructor in `MyIterator` that takes a row and initializes it using the new constructor of the entry set iterator iterator. The solution also remembers this row so that it can be used in `hasNext` when calling the new method just added to the entry set. The regular constructor should make sure it sets things up so that they work as before. Once this is all done, make sure you pass all regression tests, including all map tests.

7. Now add a `Row` nested class that extends `AbstractSet` with the correct type parameter (no raw types or type errors!) and overrides the `iterator` method to use the new constructor of `MyIterator` and the `size()` method to do something simple. After this is done, your code should pass most of the row tests (`testRnn`), except a few. The failures will show which overrides need to be done for functionality. Perform the needed overrides, again delegating BST messiness to the main class. Your should now pass all the `TestHexBoard` tests.

8. It remains still to provide efficient implementations for `contains`/`Key` and `remove` in the nested set/map classes. You will also need to override `get` in the map class, and `isEmpty` in the row class. As mentioned previously make sure to commnt on all overrides the reason why.

6 Files

In your repository, you will find the following:

src/edu/uwm/cs351/HexBoard.java Solution to last homework assignment

src/TestHexBoard.java Test cases for the updated ADT.

src/TestEfficiency.java Efficiency tests.

src/UnlockTest.java Unlock all the tests without running them.

src/edu/uwm/cs351/util/AbstractEntry.java A default implementation of `Map.Entry`; do not modify it.

The JAR file includes random testing for the main class and the "row" functionality, but does not test map or entry set functionality.