Homework #10
due Monday, November 18, 10:00 PM

In this assignment, you will implement a StudentReportMap ADT to be viewable as a map and as an entry set, and also to permit a set of at-risk students to be seen as a collection in its own right. You will augment your BST with additional references that maintain the at risk set in a linked-list style.

1 Multiple Views

The StudentReportMap ADT is an extension of the standard AbstractMap ADT. It implements a map, which associate `Student` objects to `Grade` objects. We do so using Java's `Map` interface, which allows us to interact with the data in other ways: as a set of entries (each entry containing a key and a value), as a set of keys, and as a collection of values. To support these different modes of interaction, we will define classes that implement different ADTs on the same set of data.

The same data structure will be used for all ADT views: whether collection or map. Changes through any view should affect the (shared) data structure. In particular, each view other than the map will not have *any* of its own fields, and certainly not any redundant information that could get inconsistent. Unlike an iterator, these views 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 a further view, one which has limited access to the data structure. The `atRiskSet` method will return a set that is backed by the main data structure. This view 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 (or value) 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.

clear() Remove all the entries.

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

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. We provide an implementation for `KeySet` and we use the inherited values collection. You don't have to do anything for this part.

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 class 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 Concerning AtRiskSet

The at risk set is a view that contains a subset of the students in the key set. It is meant to represent the subset of students in the class that the instructor considers to be in danger of failing. The instructor can add and remove students from this subset, and students are added by default when newly added to the map if their grade is low. Every at-risk student must be in the map. Not every student in the map is in the at risk set. A student not in the map cannot be added to the at risk set. A student removed from the at risk set is still in map. When a new student is added to the map, that student is automatically added to the at risk set if their grade is low enough. Changing a student's grade does not change whether they are in the at risk set. Rather, once a student is in the map, they must be added to or removed form the at risk set directly. Removing a student from the map also removes that student from the at risk set.

To achieve this, we make our data structure a little more complicated. We add two additional fields to each node, `nextAtRisk` and `prevAtRisk`. These will be null for a student that is not in the at risk set. If a student is in that set, the corresponding references will point to the next and previous student in that set, if any. We also add `firstAtRisk` to point to the first at risk student, and `manyAtRisk` to remember how many students are in this set. These fields are added to the `StudentReportMap` class and are parrt of the shared data structure. You can think of the nodes as being part of a BST and a doubly linked list at the same time.

`AtRiskSet` extends `AbstractSet`, but its behavior differs slightly. A student cannot be added to the at risk set if they are not already in the map. In this case, the `add` method should return false. When a student is removed from the at risk set, they are not also removed from the map. That students node should still be in the BST, but should no longer be linked to the other at-risk students via `nextAtRisk` and `prevAtRisk`. Lastly, when a student is added to the map, if their grade value is below a certain constant value, we will automatically add them to the at risk set.

Despite the fact that this set is given, by what is essentially a linked list, you will notice that it is faster to search the tree than to search the at risk links. For example, if one searches the tree for a student, and finds that this student's node has at-risk connections, the student is in the at risk set. Make use of this wherever possible.

4 Many Ways to Remove

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

- The view as a Map from Student to Grade.
- The entry set of the Map view, a Set of Entry objects from Student to Grade.
- The iterator of the entry set over Entry objects from Student to Grade.
- The at risk set, a Set of Student objects.
- The iterator over the at risk set, an Iterator over Student 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 student, and one only if passed an object satisfying the Entry ADT. One of them doesn't remove from the map, but only its own view. 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 shared remove method.

5 What You Need To Do

We have provided code that implements all basic BST functions. We have also provided the `KeySet` implementation and a significant amount of skeleton code.

1. Add the new fields to the BST and the node class. Update existing code to use these new fields appropriately.

2. Implement all methods of the `StudentRecordMap` class. You may use the provided helper methods.

3. Implement the `EntrySet` class. The iterator is given.

4. Implement the `AtRiskSet` class. We have provided an iterator once again.

6 Files

In your repository, you will find the following:

src/edu/uwm/cs351/StudentReportMap.java Skeleton ADT to work on. It includes implementations for some private methods and iterators that we have already seen, as well as the code for `KeySet`.

src/edu/uwm/cs351/Student.java The data type used as a key in the map.

src/edu/uwm/cs351/Grade.java The data type used as a value in the map.

src/TestInternals.java Run this to test your invariant checkers.

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

src/TestEfficiency.java Efficiency tests.

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

src/snapshot/Snapshot.java This code takes snapshots of your source code whenever you run unit tests. Make sure to push your cs351Log. You may have to update your java to at least version 7 to get this working.