Homework #10

due Monday, April 16, 10:00 PM

In this assignment, you will re-implement Dictionary as “Map” container class using the same data structure. A “Map” has an associated “entry set” and that set will have an (in-order) iterator.

1 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(Object)** Returns true if there is an entry for the given key.
- **containsValue(Object)** Returns true if there is an entry whose value matches the given value.
- **get(Object)** Returns the value associated with the given key, or null if there is no such entry.
- **put(K,V)** Add an entry (if no entry for key) or update the existing entry with the new value. Return the previous value associated with this key, or null.
- **remove(Object)** Remove the entry for this key (if any). Return the previous value associated with this key, or null.
- **putAll(Map)** 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.

The put and remove methods of the Map ADT return the previous value, or null if there was no previous entry for this key.
1.1 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. Maps also have “key sets” and “value collections.” The implementations for these latter collections in the abstract class `AbstractMap` uses the entry set to do the real work.

In saying the entry set is *backed* by the map, we mean that the set has none of its own fields; it simply uses the data (nodes etc) of the map. In other words, if the map changes, so does the set and if something is removed from the set, it is actually removed from the map.

The entry set has the `contains` and `remove` methods of any set, but it’s important to keep in mind that these methods are checking/removing if the argument is an entry object in the set. They do not deal with keys! And checking the entry should use the `equals` method.

The entry set associated with a map need not support the “add” operation because in general the operation cannot be implemented. For example, a map cannot have entries mapping a key to two different values; how could the entry set permit adding both?

2 Retrofitting Dictionary as a Map

For this assignment, we recommend that you start with the solution to Homework #8 and retrofit it to work as a `Map`. This section described the tasks that you will need to do. All the public methods that you add should override an existing method and so should be declared `@Override` and thus do not need Javadoc. We strongly recommend you use Eclipse’s “Override/Implement Methods” feature to generate stubs, which ensures you get the declaration right.

2.1 Nodes and Entries

Every `Map` can be viewed as a set of `Entry` objects, so you will need to make entry wrappers for each node, or more easily, have each node be an `Entry` by extending the `AbstractEntry` abstract class. The abstract class takes care of equality checks, but leaves the implementation of `getKey`, `getValue` and `setValue` methods to you. The implementation should be very easy.

Equality is important because an object doesn’t need to be a tree node to be in the entry set, it only needs to be equal to a node in the tree. You will also find equality useful so that you don’t have to handle the trickiness of comparing values with `equals` while not crashing on null.

We can leave the invariant checking unchanged.

2.2 Existing Methods

The `Dictionary` class has a `get` method that has a somewhat different semantics (throwing an exception if the key isn’t in the dictionary). Fortunately it takes a `Name` object as a parameter, whereas the `get` method required by `Map` takes an arbitrary object. We are preserving the semantics of a dictionary, so you will need to leave the existing `get` method in place while adding a new one which performs almost the same task. (Our solution was done by copying, pasting and modifying code.) Since we are not implementing a generic `Map`, but rather a dictionary which has `Name` objects as keys, we can use `instanceof` to check if the argument to `get` is the correct type.

The `put` method has the correct parameter types, but the existing method is `void`. Fortunately, changing this method to return a value will not disturb existing clients, which will now simply ignore the return value. To get the return value, your code can simply remember the result of calling the (new) `get` method.
The `values()` method should be deleted: the implementation from `AbstractMap` will work (better).

The `size` method will not need to be changed, and similarly with `copy` and `toString`.

2.3 New Methods

The abstract class requires only that we implement `entrySet`. You should use “Override/Implement Methods” from the “Source” menu to create the stub correctly. Then have it return a new (or saved) instance of a new nested class `EntrySet` (see more below).

For efficiency, you will need to implement `containsKey` (using known), `remove` (using a new helper method) and `clear` (very easy). Unlike the previous assignment, we have no parent pointers and thus cannot just find a node and remove it. Instead, you’ll need a new helper method to remove in the style of `doPut` but removing instead of adding.

2.4 The Entry Set

The entry set should be a nested (non-static) class with none of its own fields. This is very important: the entry set should only use the fields of the outer class. Do not declare ANY fields in the “entry set” class!

Since it is a set of entries, we can use `AbstractSet` to do most of the work for us, as in Homework #9. The abstract class leaves us to implement `size` (easy) and `iterator`. The latter method (again, we recommend that you use Eclipse to create the stub) should simply return an instance of the iterator (see below).

Unlike in previous assignments, we will not override the non-implementation of `add`, as explained in the section about `Map` above.

For efficiency, you should override `clear`. There’s no need to copy the code from the outer class—instead you should call it. You will need to use the `Dictionary.this` syntax to reference methods in the outer class.

For efficiency, you also need to override and implement `contains` and `remove`, but this task is complicated in that these methods are passed potential entries (disguised as `Object`). You cannot safely check the type of the argument as an `Entry<Name, Object>` because of erasure, and we do not permit you to use the “raw type” `Entry`. Instead you will need to use the type `Entry<?, ?>` (yes, those question marks are required and legal!) in `instanceof` checks and in casts. Your code should not have any warnings, nor suppress any warnings.

Then, you need to be careful that your methods only return true only if the entire entry (including both the key and the value) was present. We found that using the `equals` methods on entries made this requirement easier to satisfy. For `remove`, once you have determined that an entry is there, you should delegate to the main class `remove` (or its helper method) to remove the entry.

2.5 The Iterator of the Entry Set

Unlike the previous assignment, the iterator this time will be a typical in-order iterator, of entries. You may choose any (correct and efficient) implementation technique for the iterator, but we strongly recommend that you use a stack as explained in the Navigating Trees handout that can be found on the class website.

You are not expected to define the invariant for the iterator, but should check the outer invariant in method calls of the iterator. You are also required to implement “fail fast” semantics, using versioning. So you will need to add a version field to the main class and a local copy in the iterator.
2.6 Three Ways to Remove

If you have been keeping track, you can see that we have three different ways to remove something from the map:

1. Call the remove method directly (with a key parameter). The return value is the value from the former association.

2. Call the remove method on the entry set with an entry (not a key). The return value is a boolean indicating whether the entry existed and was removed.

3. Get an iterator for the entry set (or the related key set or value collection) and call remove after next returns the entry one wishes to remove.

For the recommended implementation, if a node to be removed has two children, it must be replaced with its greatest predecessor, that is, the rightmost child in its left sub-tree. That way the pending stack of the iterator will not be disturbed. We recommend that you implement a single helper method in the TreeMap class that removes its (key) argument and use it from all remove methods.

3 What You Need To Do

You need to retrofit the file Dictionary.java as a Map while keeping the old functionality. For this assignment, we give a suggested order so that you can avoid getting lost in the minor details before implementing the important functionality.

1. Unlock tests to test your knowledge of AbstractMap and iteration using a stack.

2. Change the class to extend AbstractMap<Name, Object>. Then let Eclipse write the stub of entrySet for you. Delete the old implementation of values().

3. Write the second get method and update the existing implementation of put to use it to return the old value.

4. Change the Node class to extend AbstractEntry<Name, Object> and then implement the three required methods (one of which is already implemented to just throw an exception).

5. Define the nested classes EntrySet and MyIterator. The first should extend AbstractSet (with an appropriate type parameter) with the second should implement Iterator (again with appropriate type parameters). Accept Eclipse’s stubs for the methods. Change the entrySet method to return an instance of the EntrySet class.

6. Implement the size() and iterator() methods in the EntrySet class.

7. At this point, the code should compile without error. Make sure it passes the TestDictionary tests. Also, make sure it passes the first group of tests (test0XX) (without iteration or removal) in TestDictionaryAsMap.

8. Implement the iterator (without removal for now). As explained above, we recommend you use the stack technique.

9. You should be able to pass the iterator tests up to where fail-fast semantics are tested (that is, up to but not including test180).
10. Add the version field, and update it as needed (in `doPut`, for example), when the tree structure changes. Implement fail-safe semantics in the iterator.

11. At this point, you should be able to pass all the tests for iterators (without removal), up to but not including `test200`.

12. Implement removal in the main class. (Use Eclipse to generate the override stub, and then work from there.)

13. Implement the `clear` methods in the main class and the entry set class.


15. Now, you should be able to pass all the “as map” tests. Make sure these are all working.

16. Once everything is working, implement `containsKey(Object)`, `EntrySet.contains(Object)`, `EntrySet.remove(Object)` for efficient tree maps. Use Eclipse’s generated override stubs, or use the `@Override` annotation by hand. Make sure that the regular tests still pass. Then you should be able to pass the efficiency tests.

17. Once you pass all JUnit tests, try the random tests.

4 Files

In your repository, you will find the following:

- `src/edu/uwm/cs351/util/AbstractEntry.java` A default implementation of `Map.Entry`. Please don’t modify this.
- `src/UnlockTest.java` Unlock all the tests without running them.
- `src/TestDictionary.java` Homework #8 tests.
- `src/TestDictionaryAsMap.java` Specifically for maps.
- `src/TestEfficiency.java` Efficiency tests.

The JAR file includes the ability for random testing.