Homework #10 (v 1.1)
due Monday, April 17, 10:00 PM

In this assignment, you will make a generic “Map” container class using a binary search tree, without parent pointers. A “Map” has an associated “entry set” and that set will have an iterator. Unlike the Homework #9, the iterator will use a stack, as explained in the “Navigating Trees” handout.

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(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 such entry.
- **put(k,v)** Add an entry (if no entry for key) or update the existing entry with the new value. 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 if the parameter is null or of a non-key type. It’s easy to compare the key to null and thus handle a null parameter as a special case, but it is not easy to check the key’s type. Because of “type erasure” it is not legal (or even useful) to check x instanceof K where K is the key type parameter to the generic class. Instead what one must do is to cast to the key type parameter and then proceed. If deep in the algorithm, the comparator is called with something illegal (not really of type K), a ClassCastException will
be called. In order to assist you, we have written a private helper method \texttt{asKey} which takes an arbitrary object and returns \texttt{null} (which you can test against) or something that apparently is indeed of the key type.

The \texttt{put} and \texttt{remove} methods of the \texttt{Map} ADT return the \texttt{previous} value, or \texttt{null} if there was no previous entry for this key. Note that the \texttt{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. Furthermore, we will not allow \texttt{null} key in the map.

There are two constructors in the class you will write: one takes a comparator for the key type, which must not be \texttt{null}. The other doesn’t, but if this constructor is used, your code should assume that the key type is comparable, that is, implements \texttt{Comparable<K>}. Then this assumption is used to create a comparator that uses the \texttt{compareTo} method to do its work.

1.1 Concerning the Entry Set of Map

The \texttt{entry set} of a map is a set of associations (instances of \texttt{Map.Entry<K,V>}) that is “backed” by the map. Since each node implements that interface, the entry set is basically a set of nodes. Maps also have “key sets” and “value collections.” The implementations for these latter collections in the abstract class \texttt{AbstractMap} uses the entry set to do the real work. So you can ignore them.

In saying the entry set is \texttt{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 \texttt{entrySet} method needs to return a set object. The skeleton file does that already; it returns the same object every time: an instance of a nested class \texttt{EntrySet} that you will need to write. As usual, we will extend the abstract class \texttt{AbstractSet} to do most of the work. As with \texttt{AbstractCollection}, the \texttt{add} method just throws an “unsupported operation” exception. Previously, we needed to override this behavior, but this time, we don’t. 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?

You will need to implement/override \texttt{size}, \texttt{contains}, \texttt{remove} and \texttt{clear}. (Your code will need to use the \texttt{TreeMap.this.method()} syntax to call a method on the outer object from within a nested class.) The set will also have an iterator (see below).

The \texttt{contains} and \texttt{remove} methods are confusing to implement since they are passed potential entries (disguised as \texttt{Objects}). You will need to perform a cast (to the type \texttt{Entry<?,?>}) if the argument is indeed an entry (using \texttt{instanceof}). Only if both the key and the value of the entry are identical to the association in the map, should \texttt{contains} return true or \texttt{remove} remove the entry.

For \texttt{remove}, once you have found the node and checked that its value matches the argument’s you will then need to call \texttt{remove} in the main \texttt{TreeMap} class.

1.2 Concerning the Iterator of the Entry Set

As in previous assignments, we will create an instance of a nested class to implement the iterator. The iterator will use a stack of pointers to traverse the tree. You are required to implement the invariant which is more complex than when we had parent pointers.

The iterator will have one node reference for the “current” node (that includes the last entry returned by \texttt{next()}). This allows us to remember what needs to be removed if \texttt{remove()} is called, and also obviates the need for a \texttt{calledNext} field.
More importantly, the iterator will have a stack of node pointers for the next node and all its “right” (greater) ancestors; these represent pending nodes that will be traversed later in the iteration, and permit us to get back up the tree despite not having any parent pointers.

Your invariant checker will need to check that the pending stack makes sense and that it is consistent with the “current” node (unless null). Perhaps the easiest way to check the stack is to create the expected value of the stack in a temporary stack, and then compare it to what the stack is. To do this, traverse from the root down to the top node on the stack, pushing any node onto the temporary stack whose key is greater than or equal to that of the top node.

Then if the current is not null, check that it is the immediate predecessor of the top node of the stack (or if the stack is empty, then the current node must be the “last” node in the tree).

1.3 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.

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.

2 What You Need To Do

You need to complete the file TreeMap.java by implementing the following methods. For this assignment, we give a suggested order so that you can avoid getting lost in the minor details before implementing the important functionality.

- The constructors for TreeMap. For one of them you need to create a comparator similar to the default comparator in the TestUtil class.
- wellFormed, checkInRange, count
  These methods are used for invariant checking and are very similar to what you did in Homework #8. Don’t start the iterator’s wellFormed() yet.
- size () (and EntrySet.size()), put(K, V)
  These methods are essential to building the BST.
- The MyIterator fields and constructor.
- MyIterator.hasNext() and MyIterator.next()
  These methods are central to getting iterators to work.
  Now you have enough to test building maps except for removing (or clearing) them.
bullet clear() and EntrySet.clear()
This will enable you to pass “clear” tests.

bullet Now implement the iterator wellFormed() method, and test using the TestInternals test.

bullet TreeMap.remove(Object) and MyIterator.remove()
(Make sure to use @Override so that Eclipse will tell you if the argument type is wrong.)
You should be able to pass all tests in TestTreeMap now. Don’t continue until you can.

bullet containsKey(Object), get(Object), EntrySet.contains(Object), EntrySet.remove(Object)
The basis of efficient tree maps. Use the @Override annotation.
You should pass the efficiency tests now.

3 Files
In your repository, you will find the following:

src/edu/uwm/cs351/util/TreeMap.java Skeleton file of TreeMap.

src/TestTreeMap.java Test cases similar to those from Homework #9.

src/TestInternals.java Test cases for the invariant checker and for the helper methods.

src/TestEfficiency.java Efficiency tests.

src/TestUtil.java Utility methods for the test suites.

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

src/edu/uwm/cs351/util/AbstractEntry.java A default implementation of Map.Entry,
you do not need to modify it.