Homework # 6
due Monday, October 15, 10:00 PM

This assignment will focus on the concepts of generics, linked lists with dummy nodes, and implementing iterators.

1 The Data Structure

For this Homework, you will implement a generic implementation of the standard collection interface using a cyclic singly-linked list with a dummy node. The list contains a dummy node even when it represents the empty sequence. This dummy node has an uninteresting data value (null). At the start, when the collection is empty, the next field of the dummy node will point back to the dummy node. After elements are added, the dummy node’s next pointer refers to the first node.

Using a dummy node means that our data structure will not need to use null pointers as links ever, which makes the implementation simpler. In particular, the precursor (see below) will never be null.

2 New Concepts

In this week, we use some new concepts: nested (non-static) classes, abstract collection classes and stale iterators. It is also the first assignment where you implement a generic class.

2.1 Nested classes

So far, we have mainly only used “static nested” classes. A (normal) nested class is interesting: it is considered to be “inside” the object in which it was created and thus has access to all the (private) fields and (private) methods as if they were its own. This is not inheritance: it doesn’t actually get any fields or methods from the surrounding class, but it can access them. It is even more confusing if the nested class extends some other class. But we won’t do that for this assignment.

Nested classes are often used to implement iterators since an iterator needs access to private internals of the collection class.

2.2 Collection ADT and AbstractCollection class

For this assignment, you will write a class that implements the standard Java Collection interface which has the following methods (in addition to ones every object has):

size() Return number of elements in the collection.

isEmpty() Return whether the collection is empty.
contains(Object) Return whether the collection contains the parameter.

iterator() Return an iterator to the elements.

toArray() Return an array of all elements.

add(T) Add an element to the collection and return true.

remove(Object) Remove an element from the collection and return true, or return false if it was not found.

containsAll(Collection) Return true if every element in the parameter is also present in this collection.

addAll(Collection) Add all the elements from the parameter to this collection, returning true if anything was added.

removeAll(Collection) Remove all the elements from the parameter from this collection, returning true if anything was removed.

retainAll(Collection) Remove all the elements in this collection that do not also occur in the parameter; returning true if anything was removed.

clear() Remove everything from the collection.

Now, as it happens, some of these operations are more fundamental than others. For example, if iterators are working, it’s easy to write toArray:

```java
class Collection{
    public Object[] toArray() {
        Object[] result = new Object[size()];
        Iterator<E> e = iterator();
        for (int i=0; e.hasNext(); i++)
            result[i] = e.next();
        return result;
    }
}
```

(This code is copyright ©2004 by Sun Microsystems.)

Notice how the code calls iterator() which is the (unimplemented) iterator method in the same class. As it happens, if iterators are working, then everything else can be implemented in terms of iterators.

For this reason, the Java collections framework includes AbstractCollection an abstract class that does precisely this: implement everything using iterators, with that crucial part omitted: the iterator() method is defined as “abstract,” that is unimplemented.

Now size() is also not implemented, even though it would be perfectly possible (albeit inefficient) to implement that method with iterators. It is left abstract because presumably each collection has a more efficient way to keep track of the size than iterating through the whole collection. The clear method is similar: it is easy enough to implement using iterators:
public void clear() {
    Iterator<E> e = iterator();
    while (e.hasNext()) {
        e.next();
        e.remove();
    }
}

(This code is copyright ©2004 by Sun Microsystems.)

Indeed the abstract class includes this implementation, but notes that a more efficient implementation should be possible.

The add() method is implemented in the AbstractCollection class, but the implementation is not useful: it simply throws an exception that the operation (add) is “unsupported.” Indeed, there’s no way one can add an element using an iterator. So, again, extenders are encouraged to override this method with a proper implementation.

2.3 Concurrent modification exceptions

In Homework #3, you had to prevent one iterator from going stale when another iterator was also traversing the list. In Java, if you use a stale iterator (by calling any method on it), it throws a “ConcurrentModificationException” exception. Ensuring that iterators are not used when they are stale is called “fail fast” semantics.

Typically, this ability is handled by adding an integer version stamp to every container and iterator. The version is incremented every time the container is changed. If an iterator notices that the version doesn’t match, it throws the required exception before performing any method. This is not part of the invariant: if the invariant fails, it is because the ADT is incorrect; if the version stamp doesn’t match it is because the ADT was misused.

The iterator version only changes if the container was changed under its control (i.e., using remove).

2.4 Generics

In Homework #3, you used generic classes. This week you will be creating your own generic class, LinkedCollection. The purpose of this is to familiarize you with making classes that are not pinned down to a specific element type and to get experience with implementing iterator objects. Inside the generic class, you can use type parameter E everywhere where an explicit type was used (e.g., Appointment in Homework #4). Furthermore, since the Node class is also generic, you must create instances of Node using this parameter as in new Node<E>(...).

\footnote{In rare cases, it need not.}
3 What You Need To Do

You need to implement `LinkedCollection`, a generic cyclic singly-linked list with dummy node implementing `Collection` by extending `AbstractCollection` and overriding the following methods:

- `clear()` Remove all elements in the collection;
- `size()` Return the number of elements in the collection;
- `add(E x)` Add an element to the end of the collection (where `E` is the generic type parameter for the elements);
- `iterator()` Return an iterator over the elements in the container.

Make sure that you mark each method as `@Override` to make sure you got the method signature correct.

The iterator should provide the standard iterator methods described in an earlier Homework:

- `hasNext()` Return true if there still remain elements to be returned.
- `next()` Return the next element in the container. If there is no such element (`hasNext()` should have returned false), then throw an instance of `NoSuchElementException`.
- `remove()` Remove the last element returned by `next()` from the collection. Throws an instance of class `IllegalStateException` if `next()` has not yet been called or if the element has already been removed. (You will need to keep track of this condition somehow in the iterator.)

This is accomplished by returning an instance of a nested class `MyIterator` which will need to have implementations of these methods plus `wellFormed`. The public methods of the iterator should implement “fail fast semantics” by throwing an instance of `ConcurrentModificationException` if the version of the iterator doesn’t match the version of the container.

Your class should use the same style as in the previous homework assignments: it asserts the correctness of the invariant before and after the two methods (`clear` and `add`) that change the data structure, and at the beginning of the two methods (`size` and `iterator`) that merely observe the state. The invariant should also be checked at the end of the constructor and when the iterator invariant is checked.

Except for some locked tests that make sure you understand some implications of the data structure, the test suite doesn’t use locked tests. We assume you know what the standard collection methods do. But if you fail a test, you should first consider whether you understand what the test is accomplishing at the abstract level. Only then try to work out the details of the implementation.
### 4 Design

The main list class is required to have a tail pointer, as well as a field tracking the number of items in the list, and a version stamp.

There are many possible designs for the external iterator. Here is the one we want you to use: the iterator keeps one pointer `precursor` and a field `hasCurrent` that is true only if an element has been returned by `next()` and not yet removed. Additionally, the iterator keeps track of the version of the collection that it tracks so it can determine if it has gone stale.

The "cursor" of the iterator will be a *ghost* field: one that isn’t there, but which can be computed from the other fields. In particular, as with Homework #4, the cursor would point to the node after the precursor. If the current element is removed, the Java iterator (unlike the “cursor” concept from the **Sequence** ADT) does not go forward and so it moves back to the same node as the precursor. Thus the ghost field can be implemented as a method

```java
private Node<E> getCursor() {
    if (hasCurrent) return precursor.next;
    else return precursor;
}
```

You may add this method to your iterator class; it helps make it clear what the “cursor” would be if it were a field. The advantage of keeping the cursor a ghost field is that we save space, and also don’t need to check for consistency.
5 Files

The directory homework6.git repository contains the following files:

- src/TestLinkedCollection.java This driver extends an abstract collection test suite (TestCollection) specifically for the linked collection class. Do not try to run the abstract class by itself.
- src/TestInternals.java Test driver for invariants of list and iterator.
- src/TestEfficiency.java Efficiency tests for your implementation.
- src/UnlockTests.java Unlock the three data structure questions.
- src/edu/uwm/cs351/LinkedCollection.java This is the skeleton file you should work with.

This homework also has random tests, as with some previous assignments.