Homework # 6  
due Monday, March 6, 10:00 PM

This assignment will focus on the concepts of generics, dummy nodes, and linked-list 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 (a reference to itself). At the start, when the collection is empty, the link field of the dummy node will point back to the dummy node. After elements are added, the dummy node’s link pointer refers to the first node.

The main list class is required to have a “tail” pointer, as well as a field tracking the number of items in the list. We provide an invariant checker for you. The dummy node will always be the next node after the tail node. Do not keep an explicit field to remember the dummy or the head.

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

2 Generics

In some earlier programs, you may have 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., Ball 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>(...).

3 What You Need To Do

You need to implement LinkedCollection, a generic cyclic singly-linked list with dummy node implementation of 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) and return true;
- **iterator()** Return an iterator over the elements in the container.

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 should assert the correctness of the invariant before and after the two methods (`clear` and `add`) that change the invariant, 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.

### 4 Design of the Iterator

There are many possible designs for the iterator. Here is the one to be used in this homework assignment: the iterator keeps two pointers `precursor` and `cursor`, similar to the cursor in the textbook implementation of a Sequence with a linked list. You will also need a copy of the version. Do not add any more fields, especially not a "called next" field! The two pointers coincide (point to the same node) when the iterator is first created and after the current value has been removed. A newly created iterator will point both to the dummy node:

![Diagram of iterator design]

After the first time `next()` is called, we have the following situation:

![Diagram after first `next()` call]

At this point, if we call `remove()`, we end up with:

![Diagram after `remove()`]

If we then call `next()` two more times, we have the following picture:
If now we call \texttt{remove()}, we end up with the following picture (notice that \texttt{tail} changed).

5 \hspace{1cm} Files

The directory \texttt{homework6.git} repository contains the following files:

\texttt{src/TestLinkedCollection.java} This driver extends an abstract collection test suite (\texttt{TestCollection}) specifically for the linked collection class.

\texttt{src/TestInternals.java} Test driver for invariants of list and iterator.

\texttt{src/edu/uwm/cs351/LinkedCollection.java} This is the skeleton file you should work with.