Homework # 6 (v 1.1)
due Monday, March 4, 10:00 PM

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

1 The Data Structure

For this Homework, you will implement a *generic* implementation of the book's Sequence ADT using a (non-cyclic) singly-linked list with a "dummy" node. The list contains this dummy node even when it represents the empty sequence. This dummy node has an uninteresting data value (null). The list will not be cyclic; the list will terminate in a null pointer.

Since we have a dummy node at the head of the list, the precursor should never be null. As a result, the cursor will *always* be `precursor.next`, and there is no need to have a regular field for it. Instead "cursor" will be converted to be a *model field*, a conceptual field without a real presence. Instead of a field, the code will define a *private* method that computes the cursor. It's private because the cursor pointer is still a data structure internal value even though it is no longer implemented with a field.

Similarly, we will convert "tail" to also be a model field, not a "real" field. Again, the class will defined a private method that computes the tail.

These changes to the data structure has a number of positive implication:

1. The existence of a dummy node means the precursor is never null, and we can avoid a special case when doing something to the front of the linked list.
2. Since the cursor is computed, then all the code we had before to assign the cursor can be dropped without a trace. Similarly with the tail; we no longer need to update it *ever.*
3. Furthermore, now that we aren't storing the cursor or tail, they can never be inconsistent and the invariant becomes simpler.

It's true that we must now write `getCursor()` where we once could simply read the value of the cursor field, but this minor change is completely overshadowed by the simplifications just described. If you do it right, the code you write for this homework can be cleaner and shorter than the solution to Homework #4 despite adding genericity and using a similar data structure.

It is also the case that computing the tail takes a non-negligible amount of time, but as it happens the "tail" field was hardly ever useful in the earlier implementation. Only one method, `addAll`, used it, and this method is not expected to be constant-time.

We also rename `manyNodes` as `manyItems` to make it clear that it counts the elements in the sequence, not the nodes (since we now have a dummy node as well).

Thus in summary, the class will have three fields: `dummy`, `precursor` and `manyItems`.

1.1 Concerning the Sequence ADT

In previous assignments, we implemented an ADT HexTileSeq. In this assignment, we generalize the ADT into Sequence to work for any element type, and also an operation to sort the elements of the sequence. The class is declared with a generic type parameter `E`. (It is a common mistake to think that the parameter is called `<E>`, but the angle brackets are just there for grouping and are not part of the name.)

For this assignment, we provide the solution to Homework #4 in your repo. We encourage you to copy the greater part of the body of the class into the new generic class and make changes in the copy. If Eclipse adds any bogus "import" statements, make sure to remove them.

Everywhere that the class refers to hex tiles (as opposed to "elements" of the sequence) must be adjusted. If it refers to the type `HexTile`, the code should be changed to use the generic type parameter "`E`". Any time that the code refers to the whole ADT `HexTileSeq`, you should change it to refer to `Sequence<E>`. Everywhere that the code uses the non-generic class `Node`, you should change it to use `Node<E>`. Don't change the Node class itself to say `Node<E>`; for clarity, we use `T` as the generic type parameter of the `Node` class.

2 Using comparators

A comparator is a special object that has an opinion about what order elements should be placed. It takes two objects of the same type (the generic parameter) and returns:

- A negative number if the first should come before the second.
- Zero if it doesn't matter what order (they are considered equal).
- A positive number if the first should come after the second.

Assuming we consider sorting in ascending order as canonical, it is easy to remember the meaning: `comp.compare(o1,o2)` is greater than zero if `o1` is greater than `o2`, is equal to zero if they are equal and is less than zero if `o1` is less than `o2`.

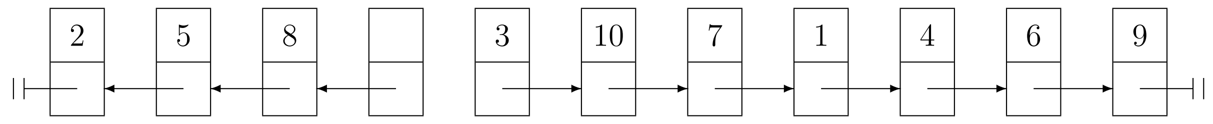
A comparator doesn't have any elements of its own; it simply judges the two elements presented to it.

3 Insertion Sort on a Linked List

The textbook describes the insertionsort algorithm in section 12.1 (after the inferior selectionsort algorithm–do not confuse them!). It shows how the algorithm works on an array. With linked lists it has the same basic structure, but with a singly-linked list, we cannot move backwards. As a result, web examples of insertion sort using linked lists lack the "adaptive" property—that sorting a mostly sorted list can be done in linear time.

Our ADT will implement a `sort` method that takes a comparator (`java.util.Comparator`) for the parameter type "E." It should use insertionsort to sort the elements in ascending order according to the comparator. It should not create new nodes; it should not use other methods (e.g. `advance()`). Rather it should do everything with its own pointers.

The trick we will use to keep the sort efficient is to represent the sorted list in *reverse order*. So for example, the fourth line in the picture on page 623 (p. 607 in the third edition) should be represented as follows:



The textbook picture would lead one to believe that the sorted list (list on the left) starts empty, but of course, we can skip the first line and start immediately with a single-element list on the second line.

Keeping the dummy node at the head of the reversed sorted list makes it much easier to insert each element.

Once the sorting is done, we have the list sorted in reverse order. So, we take another traversal of the list reversing all the pointers. The process is a simplified variant of the sorting process just described: it simply removes each element from the remainder of the (reverse order) list and places it at the front of the (correct order) list, that is, just after the dummy.

Our solution uses three while loops and a single "if" statement in 26 lines of code. If your code starts showing special cases, you are probably not using the dummy node to good effect.

4 What You Need To Do

You need to implement the `Sequence` ADT using a generic acyclic singly-linked list with dummy and with a `sort` method using `insertionsort` so that sorting an already sorted list takes only linear time.

5 Files

The directory `homework6.git` repository contains the following files:

src/TestSequence.java Unit tests for the `Sequence` ADT, most copied from previous assignments.

src/TestInvariant.java Test driver for invariants of the sequence.

src/TestEfficiency.java Efficiency tests for your implementation.

src/TestExhaustive.java Testing sorting for all permutations of lists with fewer than seven elements.

src/edu/uwm/cs351/Sequence.java This is the skeleton file you should work with.

src/edu/uwm/cs351/HexTileSeq.java Solution to Homework #4.

This homework also has random tests, but they do not test sorting.