Homework #4
due Monday, September 30, 10:00 PM

In this assignment we continue our investigation of implementing "sequence" ADTs. This week, a sequence variant will be implemented using a linked-list. Read Chapter 4 carefully and especially make sure that you read Section 4.5, pages 232–238 (225–231, 3rd ed.) which specifically say how to implement Sequence with linked lists. We will instead be implementing SequentialGallery which has a different interface (`insert` instead of `addBefore` and `addAFter`) and with a different implementation (three fields instead of four, implementing a linked-list data structure).

1 Concerning Implementation of SequentialGallery

The SequentialGallery class will be somewhat similar to the RasterSeq you implemented before, except with a number of differences. There is no way to specify an initial capacity, and some of the ADT operations are quite different. In a SequentialGallery, we cannot have a current location between elements, `remove` also moves the current location forward, and we replace `addBefore` and `addAfter` with `insert`, which adds the element before the current element, and makes the new element the current.

Furthermore, the data structure will be completely different. Declare a node class as a "`private static class`" inside the SequentialGallery class. Such a class (one declared inside another class) is called a "nested" class. The node class should have two fields: the data (of type `Painting`) and the next node. It should have a constructor but no other methods. Despite what it says in the textbook, do not write methods in the "Node" class.

You should follow the recommended design starting on page 232 (225, 3rd ed.) for the fields of the SequentialGallery class, respecting the fact that the ADT operations are different, and with the following design change: the fields `tail` and `cursor` are omitted. This makes the invariant simpler, and removes special cases from the code. There is no need to keep track of the `tail` because when there is no current element, the `precursor` will point to the last node (if any). We can easily detect when a node is the last node. (How?) The current node (what the cursor would point to) will always be either `precursor.next` if `precursor` is not null, or else it will be `head`.

Also, unlike in the textbook, the "cursor" is a *ghost* field, one that does not actually exist in the implementation. It can always be determined from the value of the (real) `precursor` field:

- If the `precursor` field is null, the cursor should be the same as the head.
- If the `precursor` points to a node in the list, the cursor should be the next node, if any, in the list.

Our implementation uses a private helper method `getCursor()` to perform this computation. This simplifies our code.

Unlike in the past assignment, `clone` requires some work for you to do: the linked list must be copied, node by node, and the `precursor` and `head` pointers of the clone made to point to the appropriate copied nodes.

1.1 The Invariant

The invariant is more complex than in the previous implementation. It has the following parts:

1. The list may not include a cycle, where the `next` link of some node points back to some earlier node.

2. The `precursor` field is either null (in which case, the cursor should be the same as the head) or points to a node in the list (in which case, the cursor should be the next node, if any, in the list). It cannot point to a node no longer in the list—the node must be reachable from the head of the list.

3. The field `manyNodes` should accurately represent the number of nodes in the list.

We have implemented the first part for you; you should implement the other parts yourself. You should do this early on in developing the implementation—it will help you catch bugs as quickly as possible. We provide code to test the invariant checker.

Be very careful that you *never* change any fields in the invariant checker, `wellFormed`. It is *ONLY* supposed to check the invariant and return true or false (with a report). It should never change things.

2 Files

The repository `homework4.git` includes the following files:

src/TestSequentialGallery.java Updated test driver.

src/TestEfficiency.java Efficiency unit tests.

src/TestInternals.java Call out to the invariant checker tests.

src/edu/uwm/cs351/SequentialGallery.java Skeleton file.

lib/homework4.jar JAR file containing the other ADTs.