Homework #4
due Monday, October 2, 10:00 PM

In this assignment we continue our investigation of implementing “sequence” ADTs. This week, it will be reimplemented 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 Song which has a slightly different interface (insert instead of addBefore and addAfter) and with a different implementation (three instead of five fields, but then with two additional song-related fields).

1 Concerning Implementation of Song

The Song class will have the same public declarations as the Song you implemented before (except no way to specify an initial capacity), but the data structure (private fields) will be completely different. Declare a node class as a “private static class” inside the Song 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 Note) 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 Song class, modulo the fact that the ADT methods are different, and the following design change. We are changing the design: 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). The current node (what the cursor would point to) will always be either precursor.next if precursor is not null. Otherwise it will be head.

Unlike in the past assignment, clone requires some (difficult!) work for you to do: the list must be copied, cell by cell, and 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.

Unlike the textbook, for us, 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.

2 Files

The repository homework4.git includes the following files:

src/TestSong.java Updated test driver.

src/TestInvariantChecker Call out to the invariant checker tests.

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

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