Homework #4
due Monday, February 19, 10:00 PM

In this assignment we continue our investigation of implementing "sequence" ADTs. This week, it will be re-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. Not all the fields will be explicit. Instead some will be "model" fields (as explained below).

Use this link to accept the assignment:

`https://classroom.github.com/a/4i4l3NG0`

As with all homeworks, the metacognitive reflection questions are mostly due on Friday at 10pm.

1 The (New!) Implementation of PartSeq

The `LinkedPartSeq` class will have the same public declarations as the DynamicArrayPartSeq that you implemented before (except no way to specify an initial capacity), but the data structure (private fields) will be completely different. We will declare a node class as a "`private static class`" inside the class. Such a class (one declared inside another class) is called a "nested" class. The node class should have three fields: the data (of type `Part`), its `function` (a String) and the next node. It should have a constructor taking values for the data fields, and using null as the initial value of the "next" pointer.. The class should have no other methods. Despite what it says in the textbook, do not write methods in the "Node" class.

The design in the textbook starting on page 232 (225, 3rd ed.) has some redundancy that isn't needed for efficiency. This week's online activity will help you decide which of the fields to make *model* fields: fields that are not stored but are conceptually part of the ADT. For the model fields, you will declare private methods to compute their values on demand, if these fields are needed. Then in the (public) methods, you should *use* the private method(s) to avoid the need to re-compute the model field(s).

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 or points to a node in the list. It cannot point to a node that is 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.

4. No part or function is null.

5. The tail pointer is correctly specified.

We have implemented the first check 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 includes the following files:

src/TestInvariant.java Tests of `wellFormed`.

src/TestLinkedPartSeq.java Same tests as in Homework #2, omitting locked tests and some constructor tests.

src/TestEfficiency.java Efficiency tests.

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

lib/homework4.jar JAR file containing the other ADTs, and random testing.

There are no locked tests for this homework assignment.

3 What you need to do

We recommend you follow these steps:

1. First, make sure all compiler errors are fixed. This will require that you declare the nested class and most (and maybe all?) of the data structure fields.

2. Implement the "model" fields according to what the activity this week recommends. This step along with the previous will establish the data structure.

3. Then implement `wellFormed` checking and use the invariant tests to help ensure that the invariant is correctly checked.

4. Implement `size`, `start`, `isCurrent` and `getCurrent`. You will also need to put in some basic error checks into `addBefore` and `addAfter`. Now check that your code passes the very first sequence tests (`testS0`n).

5. Then implement `addBefore` completely. This will require pointer manipulation. Make sure you can pass the next group of sequence tests (`testS1`n).

6. After this, you should implement `advance`. As with Homework #2, you may (eventually) find it useful to use a private helper method to share code with other methods, but no need to worry about that now. Just pass the next set of tests (`testS2`n).

7. Now you are ready to implement `addAfter` and check your implementation with the next set of tests (`testS3`n).

8. The next task is to implement `removeCurrent` using the next test of tests (`testS4`n).

9. The `clone` method requires rather more work. Pleas review the textbook starting at page 220 (p213 in the 3rd ed.), with some extra information on page 235 (p228 in the 3rd ed.). Then implement and test with the next set of tests (`testS5`n).

10. Then make sure the regular robot methods work (`getPart`, `addPart` and `removePart`) checking with the abstract robot test methods (`test`mn).

11. Check that interactions between the robot methods and the sequence methods work correctly (`testX`mn).

12. Once all the regular tests are working, use random testing to find any remaining functional bugs.

13. Then use `TestEfficiency` to fix any efficiency problems. Recall that if a test takes more than one second, see if perhaps a method which should be constant time actually has a loop in it.

14. Finally, if you changed anything to account for efficiency, re-check with the earlier tests.

There are a lot of steps, but just work through them. If you are stuck for more than ten minutes, make sure to post a question to Piazza with all the clues and then do something else for a while.

4 Objectives

In doing this homework, it is intended that the student will learn

- to implement a sorted container ADT with an (exogenous) linked list,
- to add and remove nodes from a linked list,
- to check a linked list data structure,
- to use "model" fields to save space and organize code,
- to use two pointers to mark progress through a linked list, enabling removal,
- to draw linked structure precisely showing where pointers are and to what they point, and
- to analyze code to find unsafe deferencings.