Implementing Container with Linked List and Generics (seqLL)
CS 162 - Spring 2018

Due: 11pm, Monday, April 9
Worth: 100 points

Goals
- to implement the sequence class using a different data representation
- to practice maintaining the state of a more complex object
- to further explore using linked lists
- to use generics to make the container independent of the element type

Overview
In this assignment, you will build the sequence class using a generic linked list as the representation instead of an array. A sequence of type Double is stored like:

![Linked List Diagram]

The class should also be generic which allows users to have elements of different types such as String:

![Generic Linked List Diagram]

We're giving you the fully commented class outline. This project is discussed in the text on pages 232 through 238. Your notes concerning generics, Chapter 5 (especially sections 5.3 and 5.4), and appendices E and F may be helpful.

Understand the data representation and keep the class invariant in mind.

The pictures above show the five instance variables. As in the first sequence assignment, it is important to understand the data representation and the method specifications. Know the class invariant and maintain it in each method. There are a multitude of JUnit tests to help you find bugs. You won't have as many bugs if you firmly understand what you are doing.

Provided Files
Copy the project directory at ~katz162/labs/seqLL. It contains three files LinkedSeq.java, Node.java, and TestSeqLL.java

These files compile cleanly together. After you copy the directory and create a new project, build the project and run the JUnit tests. Then start making changes. If it doesn’t build and run, clean and build again.

The Node.java file contains a generic node class implementation. This class has four methods (getData, setData, setLink, and getLink) for maintaining nodes in a singly linked list. The node elements are generic and the type must be specified when a node is created. However, you will be using it within a generic class. Therefore, you can refer to the nodes generically as in this example:

```java
head = new Node<E>(element, head);
```

The TestSeqLL.java file contains JUnit tests for the LinkedSeq class. Your class should pass all the tests with no failures.

Finally, the LinkedSeq.java file contains the start of the implementation for the generic sequence class. The clone and toString methods are already implemented. We have also included a verifyInvariant method. Leave them as is.

The verifyInvariant method is called by the JUnit tests. DO NOT call it from within your implementation. It is not a constant-time test and should not be called from the implementation. It is provided in hopes that it will trap mistakes quickly and thus aid in your debugging.

Please carefully read the pre- and post-conditions for the sequence methods. In particular, if there is no current item, addBefore places the item at the front, and addAfter places the item at the end. There are five instance variables as discussed in the text. Your implementation must match the specification described in the comments.

Your code should throw IllegalStateException as noted in the comments. The other exceptions will happen or not, and you do not need to add code to detect, throw, or handle them.
About the Class Representation

To implement the class, you must understand how the instance variables represent the object’s state. You need to know the class invariant. On page 234, the author suggests that you “write the invariant in large letters on a sheet of paper and pin it up in front of you as you work.” We include it, in somewhat large letters, here. It is a long comment in LinkedSeq.java.

precursor will be very useful for add and remove operations. Remember to update both precursor and cursor in the advance operation and to set precursor to null and cursor to head in the start operation.

```text
- the number of items in the sequence is maintained in manyNodes
- head points to the first node, if any, or it is null
- tail points to the last node, if any, or it is null
- if there is a current item, cursor points to it
  and precursor points to the node before it, if any
- if there is no current item, cursor and precursor are both null
```

How to Get This to Work

Start early and work on it when you can think. You cannot do this assignment in a couple of hours when you are tired. Getting it to compile is not enough. Think through what each method needs to do. Do this thinking away from the computer.

Don't try to do it all at once. Get one of the item adds (addBefore or addAfter) and the basics working. Then do the other item add. Then do the remove or the container adds.

Draw pictures. Have plenty of scratch paper and draw pictures of what you are trying to do in each method. Draw an empty sequence. Show all the instance variables (p, c, t, h and arrows). Draw a one-node sequence created by an addBefore. Where does cursor point? How about precursor? If you apply an advance operation, where do they point? Now if you addBefore again, what does it look like? You did put the new node at the front, right? Draw pictures studying what happens when.

If you have the sequence of doubles on the front of this handout, where would an addBefore put the node? What instance variables would be updated? What would the values of cursor and precursor be when you were done? What would happen if you did an addAfter instead?

We can't say enough about drawing pictures. Have paper and pencil beside you at all times.

Understand the specification. Know what each method should do. Check the pre-conditions and throw an IllegalStateException if they are not met. Set all the instance variables in the constructor.

Check the class invariant. Each time you complete a method, make sure the class invariant is true. Check every instance variable. This note refers to mental checks (thinking, not code). DO NOT call the verifyInvariant method.

Consider all possibilities. Does your code work on an empty sequence, a one item sequence with no current item, a one item sequence with a current item, a multi-item sequence with the current item at various places and no current item, and so forth? There is a checklist table on page 234 that might help. Often, doing something to the front item or the last item will be a different case. Consider the possibilities and handle them. Draw pictures.

Review your code. After you have it working, set your code aside for a while. Come back to it and read each method. Does it make sense? Does it work? Is there a cleaner way to write it?

Test the program and submit. You should be able to get through all the JUnit tests in TestSeqLL with no errors. Run extra test cases. Just getting your program to compile is not enough. It has to work. When you are satisfied that your program works as expected, submit your directory as the seqLL assignment.