Circular Data; State Change

Due: 2/22/2011 10:00pm

Warning: Start this problem set *early*, and read it *carefully*! (you've been warned!)

Portfolio Problems

Designing Mutable Lists

Start a new project for this portfolio and import the files from `Lists.zip` available from the assignment page. You should have three files: `Node.java`, `LoS.java`, and `LinkedListExamples.java`.

In the last lab we used a wrapper class (`Bank`) to implement a mutable list of accounts. This is because we cannot simply change an empty list into a nonempty list (it's just not possible). The example files show a similar technique to maintaining a mutable list, but instead of replacing the original list with a new list in our wrapper, we directly change the structure of the list (inserting or removing elements) by modifying/mutating the `Node`s involved.

The `LoS` and `Node` classes represent a *collection* of `String`s organized as a list. Instead of having two different classes for the empty list and for nonempty list, we have a class that represents a `Node` in a list (similar to our *Cons* class that contained the data and a link to the next item), and a subclass that represents the end of the list (a *sentinel*). An `LoS` list always contains one `Node`. If the list is empty, the `Node` is our special *sentinel* node.

The following pictures illustrate the structure of an empty *linked list* and a *linked list* after we added three `String`s, "abc", "def", and "ghi":

```
+------+
| LoS  |
+------+  +----------+
| node |->| Sentinel |
+------+  +----------+
          | ""       |
          | null     |
          +----------+

+------+
| LoS  |
+------+  +-------+    +-------+    +-------+    +----------+
| node |->|  Node | +->|  Node | +->|  Node | +->| Sentinel |
+------+  +-------+ |  +-------+ |  +-------+ |  +----------+
          | "abc" | |  | "def" | |  | "ghi" | |  | ""       |
          | next  |-+  | next  |-+  | next  |-+  | null     |
          +-------+    +-------+    +-------+    +----------+
```

1. Study the code and classes. Make sure you understand what is going on. Add an example to each test method defined in the `LinkedListExamples` class.
2. Add tests for the methods that are not tested.
3. Design the method `removeNode` for the `LoS` class that removes the `Node` that contains given `String`.
4. Design the method `size` that counts the number of nodes in a list (`LoS`), not including the *sentinel* node.

Pair Programming Assignment

Submission and Project Management

From now on, make a separate project for every problem assigned. Save the first problem (project) in a directory named `Assignment-06-1` (with a `src` directory), the second problem in `Assignment-06-2`, etc.

6.1 Problem

In this problem we will model a university registrar system.

1. Start a Java project (i.e., `Assignment-06-1` and define the classes and interfaces that implement the class diagram shown. Notice, that we will need to break the circularity of this class diagram.

```
+-------------------------------------------------+
|              +-------------------+              |
|              |                   |              |
|              v                   |              v
|          +------+                |          +------+
|          | ALoS |<-------------+ |          | ALoC |<------------+
|          +------+              | |          +------+             |
|          +------+              | |          +------+             |
|             / \                | |             / \               |
|             ---                | |             ---               |
|              |                 | |              |                |
|     ----------------           | |     ----------------          |
|     |              |           | |     |              |          |
| +-------+    +---------------+ | | +-------+    +--------------+ |
| | MTLoS |    | ConsLoS       | | | | MTLoC |    | ConsLoC      | |
| +-------+    +---------------+ | | +-------+    +--------------+ |
| +-------+  +-| Student first | | | +-------+  +-| Course first | |
|            | | ALoS    rest  |-+ |            | | ALoC rest    |-+
|            | +---------------+   |            | +--------------+
|            |                     |            |
|            v                     |            v
|    +--------------+              |     +-------------+
|    | Student      |              |     | Course      |
|    +--------------+              |     +-------------+
|    | String name  |              |     | String name |
|    | int id       |              |     | int credits |
+----| ALoC courses |              +-----| ALoS roster |
     +--------------+                    +-------------+
```

2. Define examples of at least three `Student`s and eight `Course`s, with every student enrolled in at least two courses.

3. Design the method `addCourse` in the class `Student` that enrolls this student in the given course. Throw an exception if the student is already enrolled in the course, or if the student's credits would be over 20 after the enrollment is completed.

4. Design the method `dropCourse` that drops this student from the given course. Throw an exception if the student is not enrolled in the given course, or if it would result in this student being registered for fewer than two courses after the drop is processed.

5. A student meets an interesting fellow student and wonders whether or not they might meet during one of the classes our student is enrolled in. Design the method `canMeet` that determines this student is taking a course that the other student is also enrolled in.

6. The registrar keeps a list of all students and a list of all courses. These are set up at the beginning of each quarter. Design the class

`Registrar` to manage these lists and include your original examples in the registrar's database.

7. Design the method `register` that consumes the name of a student, the student's id, and a course name, and enrolls the given student in the given course. The same restrictions on registration apply as before. Additionally, the method should throw an exception if any of the information given does not exist (no student with the given name and id, or no course with the given name).

8. Design the method `withdraw` that consumes the name of a student, the student's id, and a course name, and drops the given student from the given course. The same restrictions apply as above.

Note 1: Most of these methods require several helper methods. At least half of the grade for this homework will be assessing the design of these helpers – whether you truly follow the rule *one task = one method*.

Note 2: All of these methods (except `canMeet`), are defined only to *affect* a change in the registrar system. Design your tests carefully. At least one half of the grade for this homework will be assessing the design and implementation your tests for these methods.

Note 3: Did you notice that the above two criteria cover two halves of the grade for the homework? Well, there is some *wiggle room*, but we really want you to understand how important these two issues are (i.e., helpers and testing).

6.2 Problem: Mutating Object State

Start a new Java project (i.e., `Assignment-06-2` for this problem. Here we extend what we learned in the portfolio problem. We would like to build a list in such a way that we can start either at the front, or at the back, and move through the list in either direction. In order to do so, we have decided on the structures to represent the following scenarios:

```
        +-------+
        | Deque |
        +-------+
     +--| head  |
     |  | tail  |--+
     |  +-------+  |
     |             |
     v             v
+----------+   +----------+
|   Head   |   |   Tail   |
+----------+   +----------+
| ""       |   | ""       |
| next     |-->| null     |
| null     |<--| prev     |
+----------+   +----------+
```

```
        +-------+
        | Deque |
        +-------+
     +--| head  |
     |  | tail  |---------------------------------+
     |  +-------+                                 |
     |                                            |
     v                                            v
+----------+   +----------+   +----------+   +----------+
| Head     |   | Node     |   | Node     |   | Tail     |
+----------+   +----------+   +----------+   +----------+
| ""       |   | "abc"    |   | "def"    |   | ""       |
| abcnode  |-->| pqrnode  |-->| tailnode |-->| null     |
| null     |<--| headnode |<--| abcnode  |<--| pqrnode  |
+----------+   +----------+   +----------+   +----------+
```

Instead of one *sentinel* node, we have two of them: one marking the head of the queue and the other marking the tail of the queue. Additionally, we have a new field in each node, a reference to the previous item in the list.

The class diagram for modeling this data would then be:

```
         +------------+
         |   Deque    |
         +------------+
      +--| Node head  |
      |  | Node tail  |--+
      |  +------------+  |
      |                  |
      +-------+ +--------+
              | |
+----------+  | |  +-----------+
|     +--+ |  | |  | +--+      |
|     |  | |  | |  | |  |      |
|     |  v v  v v  v v  |      |
|     | +-------------+ |      |
|     | |    Node     | |      |
|     | +-------------+ |      |
|     | | String data | |      |
|     +-| Node   next | |      |
|       | Node   prev |-+      |
|       +-+---------+-+        |
|        / \       / \         |
|       +---+     +---+        |
|         |         |          |
| +-----------+  +-----------+ |
| |   Head    |  |   Tail    | |
| | ""        |  | ""        | |
+-| Node next |  | null      | |
  | null      |  | Node prev |-+
  +-----------+  +-----------+
```

1. Define the classes `Node`, `Head`, `Tail`, and `Deque` that implement *doubly-linked* lists of `String`s. Use the code from the portfolio problems as your model.

2. Make examples of three lists: the empty list, a list with two values (`"abc"` and `"def"`) shown in the drawing at the beginning of this problem, and a list with three values, ordered lexicographically from the `head` to the `tail`.

3. Design the method `size` that counts the number of nodes in a list (`Deque`), not including the two sentinel nodes (the `head` and the `tail`).

4. Design the method `addAtHead` for the class `Deque` that consumes a `String` and inserts it at the `head` of this list. Be sure to fix up all the links correctly!

5. Design the method `addAtTail` for the class `Deque` that consumes a `String` and inserts it at the `tail` of this list. Again, be sure to fix up all the links correctly!

6. Design the method `removeFromHead` for the class `Deque` that removes the first node from this `Deque`. Beside the obvious *effect*, it should produce the `String` that was removed.

   Throw an exception, if an attempt is made to remove from an empty list.

7. Design the method `removeFromTail` for the class `Deque` that removes the last node from this `Deque`. Beside the obvious *effect*, it should produce the `String` that was removed.

   Throw an exception, if an attempt is made to remove from an empty list.

8. Design the method `insertSorted` for the class `Deque` that consumes a `String` and inserts it into this sorted list in the correct order.

9. Design the method `removeSorted` for the class `Deque` that removes the node that contains the given `String` from this `Deque`.

   Throw an exception, if the there is no such `String`.

10. Design the method `toLowerCase` for the class `Deque` that changes all the `String`s to lower case.

    The class `String` defines the method `toLowerCase` that produces a new `String` with all letters changed to lower case.

    Note: Do not use this method with special characters without looking up the formal documentation for the method.