Homework # 5

due Monday, October 9, 10:00 PM

In this assignment we implement “endogenous” “doubly-linked” lists. An endogenous linked-list is one where the objects themselves are the nodes in the list. Doubly-linked lists are linked-lists that can be traversed forward and backward. The abstract data types you will be implementing are Groups: ordered lists of students. You will also implement several Comparators to determine how students should be ordered in a group. We provide a driver to demonstrate one possible application of the Group ADT.

1 Concerning the Student ADT

Each Student has three properties:

- **String _name** Name of the student.
- **double _grade** Grade percentage of the student, in range [0, 100].
- **double _attendance** Attendance percentage of the student, also in range [0, 100].

Additionally, each student has a previous and a next pointer that point to students before and after it in its ordered Group (see below). These pointers, however, are not given public accessors because any operations related to them are only meaningful in the context of a group. As such, it is necessary that we nest the Group class inside the Student class to give it (and no one else) the necessary access to these pointers.

1.1 Student Order

What order should students be given in a group? Well, that depends on what the teacher would like to do with the group! They may want to alphabetize them for grade entry, or they may want to order by attendance to see who hasn’t been coming to class. Is it possible to formalize these various orders into real code?

Yes! In Java, the Comparator<T> interface allows us to define our own way of comparing objects of type T, and subsequently enforce ordering based on those comparisons. T is a generic type that can be seen as a placeholder for some type to be determined later. For this assignment, we will use comparators over Student objects. In Java, the Comparator<T> interface is defined as follows:

```java
public interface Comparator<T> {
    /**
     * Compares its two arguments for priority. Returns a negative integer,
     * zero, or a positive integer if the comparator considers its first argument
     * as less than, equal to, or greater than the second, respectively.
     */
    int compare(T o1, T o2);
}
```

Each instance of a class implementing this interface defines its own way to compare objects of generic type T. We will use compare to determine the following relationships: Assume c1 is a Student Comparator, and s1/s2 are Students...

\[
c1\text{.compare}(s1,s2) < 0 : \text{s1 is less than s2}
\]
\[
c1\text{.compare}(s1,s2) = 0 : \text{s1 is equal to s2, or rather s1 is indistinguishable from s2}
\]
\[
c1\text{.compare}(s1,s2) > 0 : \text{s1 is greater than s2}
\]

When we impose a comparator’s ordering onto a group, we put the least student (according to the comparator) first (at the head of the group), and then follow with students of increasing “value” (also according to the comparator). In the case of students being indistinguishable, the newest student added to the group should be placed immediately after all students the comparator says it is equal to. In this homework, we will implement three different ways to order students:

**Alphabetical** Order students in alphabetical order by name.

**GradeDescending** Order students with highest grade first, followed by lower grades in descending order. It may be helpful to note that according to this comparator higher grades belong before lower grades, and it should judge a student with higher grade as 'lesser than' a student with lower grade to impose this ordering. If we wanted the grades in ascending order, it would be the reverse.

**AttendanceDescending** Order students with highest attendance percentage first, followed by lower attendance percentages in descending order.

Each of these techniques is represented by a Java class that implements the Comparator interface, overrides the toString method and defines a static getInstance method to return a previously created instance.

### 2 Concerning the Group ADT

The Group ADT is given a comparator at construction, and always maintains an ordering on its students according to that comparator. Internally, it maintains a pointer to its first student (head) and its last student (tail). If there is only one student in the Group, that student will be both the head and tail.

As in Homework 3, the Group ADT will extend the AbstractCollection<E> class. This means that again most of the work is already done for you! It is recommended you re-read the section on the Collection ADT in the handout for Homework 3 to recall which methods must be completed (and why) when extending AbstractCollection. You are also responsible for implementing and enforcing the invariant of Group and its iterator (details inside class). As with previous homeworks, the iterator should be fail-fast and its invariant should not be checked if its version doesn’t match the version of its group.
3 Grouper Driver

Our driver is designed to be an aid for teachers to generate groups based on certain specifications they desire. For example, if two students distract each other you may want to place them in separate groups. On the other hand, if two students are friends and deserve a reward you may want to place them in the same group. In other instances it may be useful to you to place students performing poorly with students performing well in hopes that some wisdom may be imparted, or place students with the best attendance alongside students with the worst attendance to ensure that a group won’t all be absent.

Our driver will allow these specifications to be set and groups to be generated that, to the extent that is possible, satisfy them. Some terminology used in the driver:

Do Group A Group of students that the teacher would like to be placed together.

Don’t Group A Group of students that the teacher would like to be placed separate.

Default Group Any students who weren’t placed into a Do or Don’t group. If this default group is ordered by grade or attendance, students will be placed into groups by highest/lowest pair. If it’s ordered alphabetically, students will be placed randomly into groups.

4 What You Need To Do

Your responsibility, as mentioned above, is to implement the necessary methods in Group and its nested MyIterator class to finish it off (with the help of AbstractCollection) as a modifiable collection with all methods required by that contract. You must also implement the comparators described above. Note that no work is required of you for the driver, and until you’ve finished all your time should be spent on the preceding items.

5 Files

The directory homework5.git repository includes the following files:

5.1 Tests

src/UnlockTests.java Run this class to unlock all the tests without running them.

We recommend you run this before writing any code to ensure you understand how the ADTs should behave before you go about implementing them.

src/TestInternals.java Test suite for the invariants of Group and its iterator.

src/TestGroup.java Test suite for the Group class.

src/TestComparators.java Test suite for the comparator classes.

5.2 ADT

src/edu/uwm/cs351/Student.java Class containing skeletonized Group class.

src/edu/uwm/cs351/util/Alphabetical.java Skeleton file for Comparator #1.

src/edu/uwm/cs351/util/GradeDescending.java Skeleton file for Comparator #2.

src/edu/uwm/cs351/util/AttendanceDescending.java Skeleton file for Comparator #3.
5.3 Driver

src/edu/uwm/cs351/Grouper.java Application to make groups from students.

src/edu/uwm/cs351/Generation.java Helper application to perform and display group assignments.

classes//*.xml XML files containing fictional data on students in various classes. These are complete fiction generated randomly, and any resemblance to actual persons, living or dead, is purely coincidental.

The skeletonized files must be completed and exist in your homework5.git repository on andrew.cs.uwm.edu before the deadline (10:00 PM, Monday).