Homework # 5
due Monday, October 7, 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 `SortedCollections`: ordered lists of paintings. You will also implement several Comparators to determine how paintings should be ordered in a sorted collection. We provide a driver to demonstrate one possible application of the `SortedCollection` ADT.

1 Concerning the Painting ADT

Each Painting has three properties:

File file A file containing a picture of the painting.

String name The name of the painting.

String artist The name of the artist.

int year The year the painting was completed.

int value The value of the painting in USD.

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

1.1 Painting Order

What order should paintings be given in a collection? Well, that depends on what one would like to do with the collection! One may want to order them by year to see historical developments, or they may want to order by value to see what expensive paintings look like. 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 `Painting` objects. In Java, the `Comparator<T>` interface is defined as follows:

```
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 `Painting Comparator`, and `p1/p2` are `Paintings`...

`c1.compare(p1,p2) < 0` : p1 is less than p2

`c1.compare(p1,p2) = 0` : p1 is equal to p2, or more precisely p1 is equivalent to p2

`c1.compare(p1,p2) > 0` : p1 is greater than p2

When we impose a comparator's ordering onto a sorted collection, we put the least painting (according to the comparator) first (at the front of the collection), and then follow with paintings of increasing "value" (also according to the comparator). In the case of paintings that are equivalent, they could be in any order relative to one another and the collection would be sorted. For this assignment, the newly added painting should be placed immediately after all paintings the comparator says it is equivalent to. In this homework, we will implement four different ways to order paintings:

Alphabetical_artist Order paintings in alphabetical order by artist.

Alphabetical_name Order paintings in alphabetical order by name.

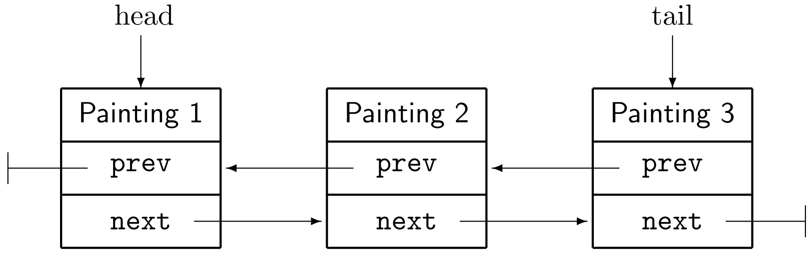
ValueDescending Order paintings with highest value first, followed by less valuable paintings in descending order. It may be helpful to note that according to this comparator higher values belong before lower values, and it should judge a painting with a higher value as 'lesser than' a painting with a lower value to impose this ordering. If we wanted the values in *ascending* order, it would be the reverse.

DateDescending Order more recent paintings first, followed by older paintings 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 **SortedCollection** ADT

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



As in Homework 3, the `SortedCollection` ADT will extend the `AbstractCollection<E>` class. This means that again much 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 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 collection.

Because the collection is sorted, we have an extra requirement for inserting new paintings. When adding multiple paintings consecutively, they might be added in any order, but one might consider what orders might be more likely. In particular we might sometimes add items in an already-sorted order. One such case might be cloning the contents of a sorted collection into a new sorted collection. If this case is more likely, and might occur sometimes by design, we should consider efficiency for this special case. Specifically, if we are adding something that should be placed after everything already in the collection, we should notice this immediately, and we can add it in constant time. To this end, we require that you search starting at the tail when adding. This is also why you must insert new paintings *after* any equivalent paintings.

2.1 Static Nested Classes

As mentioned in Section 1, the `SortedCollection` class *must* be nested to access the private members (fields and methods) of the `Painting` class. In Java, unless we declare a nested class as `static`, it may only be instantiated through an existing instance of its outer class. For example, if `SortedCollection` wasn't declared `static`, we would be forced to instantiate it through a Painting:

```
Painting p = new Painting(...);
SortedCollection c = p.new SortedCollection(...);
```

With this structure, we would be *unable* to instantiate a collection unless we already had a painting to create it with. This would be a poor fit for our purposes, as we may want to instantiate a collection before any paintings exist. When we declare the nested `SortedCollection` class as `static`, we are enabling it to be instantiated on its own, whether or not any paintings exist. It will still, however, have access to the private members of any `Painting` objects it has a reference to.

3 Demo

The supplied driver will use your code to sort a number of paintings by some criteria and then display them to you. Try changing the chosen comparator in the driver to sort the paintings differently!

4 What You Need To Do

Your responsibility, as mentioned above, is to implement the necessary methods in `SortedCollection` 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 `SortedCollection` and its iterator.

src/TestSortedCollection.java Test suite for the `SortedCollection` class.

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

src/TestEfficiency.java Testing to make sure SortedCollection operations are efficient.

5.2 ADT

src/edu/uwm/cs351/Painting.java Class containing skeletonized `SortedCollection` class.

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

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

src/edu/uwm/cs351/util/DateDescending.java Skeleton file for Comparator #3.

src/edu/uwm/cs351/util/ValueDescending.java Skeleton file for Comparator #4.

5.3 Driver

src/edu/uwm/cs351/Demo.java Application to display collections of paintings.

5.4 Paintings

Paintings/*.jpg Various files containing pictures of paintings.

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).