Homework # 13
due Monday, May 10, midnight

In this assignment you will implement a priority queue using a linked list and mergesort pages 630–640 (3rd ed: pp. 614–624). You will use "comparator" objects.

1 Concerning Merge Sort

The mergesort algorithm has the following structure:

mergesort(s) where "s" is a list of elements

1. If "s" is empty or has only one element, return it unchanged.
2. Otherwise:
   (a) split "s" into two parts, "s_1" and "s_2".
   (b) mergesort(s_1) recursively
   (c) mergesort(s_2) recursively
   (d) merge the resulting sorted elements into one
   (e) return the result.

As you may have noticed, sorting lists in place is tricky because of the pointers. For this assignment, you must sort using lists again (not by copying the values into an array and sorting there). We recommend that your mergesort algorithm take a node to use as head, and then a length, so that it is not necessary to "snip" apart the list (by inserting null pointers) before recursive calls. The solution also passes around the dummy node to be used for the convenience of `merge` which must construct a new linked list from two lists. Ultimately, it is up to you to design the merge sort algorithm, but it must use merge sort!

2 Concerning the Queue ADT

As you noticed in the previous homework, the Queue ADT does not use `enqueue`, `dequeue` and `front` operations. Instead it adds onto the Collection ADT the following six operations, which implement the three operations with different results if there is an error:

Operation	Exception error	Funny value error
enqueue	`add(x)`	`offer(x)`
dequeue	`remove()`	`poll()`
front	`element()`	`peek()`

The `add` method throws an `IllegalStateException` if the queue is full, whereas `offer` returns false in such situations. The `remove` and `element` methods throw a `NoSuchElementException` if the queue is empty whereas `poll` and `peek` simply return `null` when it is empty.

Additionally, queues often cannot handle null values. This will be the case for this homework assignment. Thus `add` and `offer` should throw `NullPointerException` if the client attempts to add null pointers to the queue.

3 Concerning Priority Queues

A priority queue is a queue in which we use a priority to determine the order elements come out. The way we determine which element is highest priority (first in ascending order, usually) is with a comparator. As you saw in the previous assignment, the priority queue takes the comparator in the constructor. Unlike the previous assignment (and unlike the standard library implementation of priority queues), we also permit the comparator to be *changed* at any point.

Java's standard library class uses a "natural comparator" as the default comparator. The natural comparator calls the `compareTo` method. Our class, on the other hand, uses an "indifferent comparator" as the default comparator. The *indifferent* comparator doesn't care about any ordering. In other words, everything is the same to it. An indifferent comparator always returns zero when passed two objects to compare.

When implementing the priority queue class, there is an abstract class, `AbstractQueue`, that handles most of the work for an implementor of a queue that does not handle null elements. It implements the three exception throwing methods (on the left in the diagram) using the other ones (on the right). Thus the class will need to implement:

- `offer`, `poll` and `peek`
- `size` and `iterator` (for collections)
- `getComparator` and `setComparator` (new methods)
  The second method does a mergesort operation to get the elements into order.
- `reverse` (new method).
  This method logically reverses the priority of all elements from this time forth. This operation should be constant-time. You don't need to change the linked list itself.

4 The Data Structure

You will implement the priority queue using a generic cyclic doubly-linked list with a dummy node. As usual, there is a "size" field for efficiency. There will be a comparator and a field indicating whether the list is in "reverse" mode. There is also a "version" field for implement fail-fast semantics for iterator staleness.

Here are the (hopefully obvious by this time) data structure invariants:

- The dummy node cannot be null;
- The dummy's data must be null;
- The list must be properly cyclic with prev/next fields consistent;
- The "size" field must have the proper value.
- The comparator cannot be null;
- There can be no null elements in the list;
- The elements must be in order according to the comparator.

The "reverse" field can be either true or false without affecting the rest of the data structure (that is, there is nothing to check). Rather it affects the way the operations are implemented. If "reverse" is false, we add to the tail of the list and remove from the head, but if it is true, we add to the head and remove from the tail. In either case, a newly added element may need to be migrate further because of its relative priority.

Even after `reverse()` is called, the elements in the list will *still* be in the same order as before, and new elements are still inserted using the same comparator, only that we start at a different end. Only if some elements are treated the same by a comparator will they end up in a different place after being added.

Almost all the methods need to behave differently if "reversed" is true. For example, if the queue is reversed, the item to return is on the opposite end of the linked list. If it is reversed, you should search in the opposite direction when inserting.

5 What you need to do

You need to write the (simple) methods: `offer`, `poll`, `peek`, `size` and `reverse`. The `iterator` method simply returns an instance of a private nested class, which is given. Then you need to complete the implementation of the (most challenging) method: `setComparator`. This method should either do nothing if the same comparator is already in use, do nothing after checking the whole list for sortedness if it is already sorted according to the given comparator, or use merge sort if sorting is necessary. Make sure your sort is stable!

6 Files

The repository for this homework includes the following files:

src/TestPriorityQueue{1,2}.java JUnit test cases, including some locked tests.

src/TestEfficiency.java JUnit test cases for efficiency.

src/TestExhaustive.java JUnit test cases for sorting all permutations of lists up to 6 elements.

src/TestInvariant.java Invariant tests.

src/edu/uwm/cs351/util/PriorityQueue.java skeleton file.

This homework also includes random tests, run `RandomTest` in the JAR file.