Homework # 14
due Monday, December 18, 10:00 PM

In this assignment, you will implement a priority queue using a dynamic circular array. Use this link to accept the assignment:

`https://classroom.github.com/a/Y3AWJp_2`

This homework is due during finals week. If you do it, your lowest assignment score will be dropped. Otherwise the zero you get on it will be dropped.

1 Concerning the PriorityQueue ADT

As described in Section 10.1, a `PriorityQueue` is a container that enables elements to be removed by priority. It's neither LIFO nor FIFO; rather the least element (according to a comparator) is the first one out. The public operations are the same as for the `Queue` interface:

size() Return the number of elements in the priority queue.

add(elem) Add an element (which may be null) to the priority queue and return true.

offer(elem) Same as "`add`." If the priority queue is space limited (which will not be the case for us), it can return false.

element() Return the first element, or throw a `NoSuchElementException` if the priority queue is empty.

peek() Return the first element, or null if the priority queue is empty.

remove() Remove and return the first element, or throw a `NoSuchElementException` if the priority queue is empty.

poll() Remove and return the first element, or null if the priority queue is empty.

iterator() Return an iterator over the elements. The order is undefined, but for us it will be in sorted order.

The textbook describes the `heap` data structure which works well for this ADT. In this homework, we experiment with a data structure that is conceptually simpler, but not as efficient in general.

Many implementations (including the one you are doing for this homework assignment) of `PriorityQueue` are generic and take a comparator that defines the order. If no comparator was given. then the element type is assumed to be `Comparable` ("natural ordering").

2 The Circular Array Data Structure

In this assignment, we will use a circular array, described in the first few pages of Section 7.3 of the textbook. Please read this part.

In the textbook, the data structure is used for a Queue ADT where elements are only accessed on the *ends*: adding to one end and removing from the other. But for PriorityQueue, elements need to be added in order, which could be at any point within the queue, not just the rear. Furthermore, the iterator has a "remove" method which permits any element to be removed from the priority queue as well. As a result, we must be able to add and remove elements from anywhere within the priority queue.

The big cost with a circular-array-based queue is shifting elements over. This should be minimized by shifting toward/from the end that is closest to the point where the insertion/deletion is happening.

For example, suppose we have five elements in our priority queue:

13	25			3	6	10

`head`=4, `rear`=2

Suppose we add 7 to the priority queue. We find where to add it using binary-search, and find out we want to add it after 6 and before 10. This is closer to the front, and so to add it, we shift the head back:

13	25		3	6	7	10

`head`=3, `rear`=2

Suppose that we then wish to remove "10." This is closer to the end now, so we shift elements back from the rear:

25			3	6	7	13

`head`=3, `rear`=1

Notice that the shifting may need to wrap around the end of the array.

The algorithms for working with a circular array have several cases. Make sure to work with small examples to check your ideas. Make sure you understand what your code does. Don't "guess" code; it will be disaster when debugging. We provide very few "normal" tests. Most of the testing will be done with random testing. If you don't recall how random testing works, see Lab exercise # 2. In particular, random testing must be repeated over and over until you pass. It is not sufficient to just fix your code to pass one of the tests output for you.

We have provided you with the data structure invariants. Make sure you assert the invariant in all the required places.

3 Concerning the `CircularArrayPriorityQueue`

The task for this assignment is implementing a priority queue using the circular array data structure, in which the elements are kept in sorted order. In particular the `poll` and `peek` operations should have constant-time implementations.

3.1 Concerning `AbstractQueue`

Our implementation will be built on top of `AbstractQueue`; the class extends it. The `AbstractQueue` class implements most things. In particular, it will implement `add`, `element` and `remove` using `offer`, `peek` and `poll`. Don't override the former implementations. You also need to implement `size` and `iterator`.

3.2 Equivalent elements

Elements that are equivalent should be treated in strict FIFO order. For example, if the queue holds colored numbers and is ordered only by the numeric value, then if someone adds a red 5 and a green 10 and then adds a blue 5, then the red 5 should come out first followed by the blue 5 and finally by the green 10. The binary search algorithm can be adapted so that when we add an element, it always goes after any existing equivalent elements. We don't need to worry about binary search for `remove` (why not?) which is good since it is harder to adapt a binary search lookup operation to handle removals in this way.

4 What you need to do

Here is our recommended order of work:

1. Stub all the required methods so that there are no compiler errors. Make sure that all `@Override` annotations are commented with the reason, as always. Any public declarations that are not so marked need documentation comments.

2. Draw pictures for different cases of the "offer" algorithm.

3. Implement all the methods except the iterator, and test using the normal ADT tests. You should be able to pass all of them except `test8`.

4. Implement the iterator except for removal, and make sure you can pass `test8` also.

5. Figure out how to implement the iterator's `remove` method (by drawing examples on paper) and then code your idea.

6. Use random testing repeatedly to find all your ADT bugs.

7. Check your algorithms with efficiency testing, going back to do random testing if you had to make any changes. Every efficiency test should complete in a fraction of a second—a second or more for a single test is too slow.

Implementation note When this homework was tested out, one of us used helper methods to handle the frequent wrapping of indices; the other did it manually with "if" statements. The first person was able to solve the problem in only a couple of rounds of random testing after passing the normal tests. The second person needed a half-dozen or more rounds of random testing.