Assignment 7

Priority Queues

Introduction

In Part 1 you will be implementing the `PriorityQueue` interface using an array-based `Heap` data structure that will store `Comparable` objects (objects that implement the `Comparable interface`). A reference-based list implementation of a `PriorityQueue` is provided for you (`LinkedPriorityQueue.java` and `ComparableNode.java`), so you can run the tester and compare the run times of the two implementations.

In Part 2, you will implement a small application modeling the line at a Music Concert (called `ConcertLine.java`). The ConcertLine class uses the `HeapPriorityQueue` you implemented from Part 1.

Although there is a tester provided for this assignment, it does not include a comprehensive set of sets for each method. You should add your own tests for any test cases not considered.

NOTE: The automated grading of your assignment will include some different and additional tests to those found in the `A7Tester.java` file. For all assignments, you are expected to write additional tests until you are convinced each method has full test coverage.

Submission and Grading

Attach `HeapPriorityQueue.java`, `Attendee.java`, and `ConcertLine.java` to the BrightSpace assignment page. Remember to click **submit** afterward. You should receive a notification that your assignment was successfully submitted.

If you chose not to complete some of the methods required, you **must** provide a stub for the incomplete method(s) in order for our tester to compile. If you submit files that do not compile with our tester, you will receive a zero grade for the assignment. It is your responsibility to ensure you follow the specification and submit the correct files. Additionally, your code must not be written to specifically pass the test cases in the tester, instead, it must work on all valid inputs. We may change the input values during grading and we will inspect your code for hard-coded solutions.

Be sure you submit your assignment, not just save a draft. All late and incorrect submissions will be given a zero grade. A reminder that it is OK to talk about your assignment with your classmates, but not to share code electronically or visually (on a display screen or paper). Plagiarism detection software will be run on all submissions.

Objectives

Upon finishing this assignment, you should be able to:

- Write an implementation of an array-based Heap
- Write code that uses an implementation of the Priority Queue ADT
- Write code that works with objects that implement the `Comparable<E>` interface

Instructions

The `LinkedPriorityQueue` implementation provided does not make use of the $O(logn)$ heap insertion and removal algorithms we discussed in lecture. Instead, when an item is inserted, the queue is searched from beginning to end until the correct position to insert the item is found.

The `HeapPriorityQueue` insertion implementation you write should improve on this implementation so that the insert runs in $O(logn)$ using the `bubbleUp` algorithm covered in lecture. Similarly, your `removeMin` should use the `bubbleDown` algorithm.

Part 2 asks you to complete the implementation of the `Attendee` and `ConcertLine` classes. Using these classes with a priority queue will give you experience building a small application that models an line-up at a concert, where attendees are given priority depending on the type of ticket they purchased. Attendees waiting to enter the concert are stored in the priority queue such that high priority attendees are let in (ie. removed from the priority queue) first.

Note that the priority queue in this assignment works with Comparable items. All classes that implement Java's Comparable interface must provide an implementation of the `compareTo` method, which is a method that allows us to compare two objects and return a number that represents which of the two should come first when sorted. The compareTo method is helpful for a priority queue as it can be used to compare priority values to determine how items should be ordered within the priority queue.

When implemented correctly, an object's compareTo method has the following behaviour:

- returns 0 if the two objects being compared are equal
- returns a negative value if this object should be ordered before the other object
- returns a positive value if this object should be ordered after the other object

For example:

```
Integer a = 7;
Integer b = 9;
a.compareTo(b); // returns a negative value.
a.compareTo(a); // returns 0

String x = "computer";
String y = "science";
x.compareTo(y); // returns a negative value.
y.compareTo(x); // returns a positive value
```

Part 1 Instructions

1. Download and unzip all of the files found in `a7-files.zip`.

2. Read the comments in `HeapPriorityQueue.java` carefully Add the constructors and required method stubs according to the interface in order to allow the tester to compile.

3. Compile and run the test program A7Tester.java with LinkedPriorityQueue.java to understand the behaviour of the tester:
```
javac A7Tester.java
java A7Tester linked
```

   Note 1: ignore any warnings about unchecked or unsafe operations. For example:
```
LinkedPriorityQueue.java uses unchecked or unsafe operations.
Recompile with -Xlint:unchecked for details.
```

   Note 2: the A7Tester is executed with the word "linked" as an argument. This argument allows us to run all of the tests against the LinkedPriorityQueue implementation provided for you. You will notice it is extremely slow with larger file input sizes (for the largest input size we will only test it with the heap implementation).

4. Compile and run `A7Tester.java` with `HeapPriorityQueue.java` and repeat the steps below:
```
javac A7Tester.java
java A7Tester
```
   (a) Uncomment one of the tests in the tester
   (b) Implement one of the methods in HeapPriorityQueue.java
   (c) Once all of the tests have passed move on to Part 2.

   Note 3: You can run the tester on the LinkedPriorityQueue by running as follows:
```
java A7Tester linked
```
   When the word `"linked"` is added after `A7Tester` it will run the `LinkedPriorityQueue`, otherwise it will run the `HeapPriorityQueue`. Notice how much slower the `LinkedPriorityQueue` is! It even skips the test with the largest number of elements, and is still noticeably slower!

HINT: I made a number of helper methods in my implementation. The names of my helper methods are `isLeaf`, `minChild`, and `swap`. It might be a good idea to implement your own helper methods (possibly even more than I implemented for my solution).

Part 2 Instructions

For this part of the assignment, you will be creating an application to support the modelling of a line to get into a concert. You are asked to write the software that will manage handling attendees waiting to be enter the concert based on the type of ticket purchased and their arrival time. There are three types of tickets: (1) VIP tickets, (2) Premier tickets, and (3) regular tickets.

Imagine you are checking people in to a concert. Each time someone arrives they enter a different lineup based on the type of ticket they purchased, and are asked to wait until the next booth is open to process their ticket and let them in to the concert.

You must make sure all high priority attendees who purchased VIP tickets are allowed to enter before lower priority attendees. For example:

- An attendee (A) enters with a regular (level 3) ticket. You add them to the queue.
- An attendee (B) enters with a VIP (level 1) ticket. You add them to the queue ahead of the previous person (A), as attendees with VIP tickets must be handled first.
- Another attendee (C) enters, also with a VIP (level 1) ticket. You add them to the queue ahead of person A (level 3) but behind the person B (also priority level 1) because both B and C have the same priority level, but C arrived at a later time.
- A ticket is ready to be processed, so person B gets helped and leaves the line (priority queue) and enters the concert.
- An attendee (D) enters with a premier (level 2) ticket. You add them to the queue ahead of attendee A (priority level 3) but behind attendee C (priority level 1).

The given tester (`A7Tester.java`) will test the functionality of your ConcertLine implementation and mimics a scenario similar to the one above. You are given a working implementation (LinkedPriorityQueue) that models the scenario above, but it is inefficient. Your task is to implement a more efficient version in HeapPriorityQueue. Read the tester and documentation carefully to help you understand how the classes you will be writing will be used.