Assignment 6

Objectives

- More practice implementing an interface in Java
- Exposure to the Priority Queue ADT
- Practice implementing a Heap
- Exposure to the Comparable<E> interface

Introduction

This assignment has two parts:

Part 1 asks you to implement 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 `PriorityQueue` is provided for you (`LinkedPriorityQueue.java` and `ComparableNode.java`) so you can run the `Part1Tester` and compare running times of the two implementations.

Part 2 asks you to implement a small application modeling an emergency room in a hospital, in which you will use your `HeapPriorityQueue`.

The `LinkedPriorityQueue` implementation provided does not make use of the $O(\log n)$ **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 $O(\log n)$ using the bubbleUp algorithm covered in lecture. Similarly, your removeMin should use the bubbleDown algorithm.

Part 2 asks you to implement the `Patient` and `EmergencyRoom` classes. Using these classes with a priority queue will give you experience building a small application that models an emergency room in a hospital. Patients waiting to be helped are stored in the priority queue such that high priority patients are helped (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;
```
`a.compareTo(b)` returns a *negative* value.

```
Integer b = 9;
```
`a.compareTo(a)` returns 0

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

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

Part I

1. Download the files: `A6Tester.java`, `PriorityQueue.java`, `HeapPriorityQueue.java`, `LinkedPriorityQueue.java`, `ComparableNode.java`, `HeapEmptyException.java` and `HeapFullException.java`.

2. Read the comments in `HeapPriorityQueue.java` carefully
   a. **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 `A6Tester.java` with `LinkedPriorityQueue.java` to understand the behavior of the tester:

```
javac A6Tester.java
java A6Tester linked
```

   Note #1: ignore any warnings about unchecked or unsafe operations. For example:

```
Note: .\LinkedPriorityQueue.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for details.
```

   Note #2: the A6Tester is executed with the word "linked" as an argument, similar to how we added the file name of mazes as an argument for the previous assignment.

   This argument allows us to run all of the tests against the `LinkedPriorityQueue` file 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 the test program `A6Tester.java` with `HeapPriorityQueue.java` and repeat step a and b below

```
javac A6Tester.java
java A6Tester
```

   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 II.

5. You can run the tester on the `LinkedPriorityQueue` by running as follows:

```
java A6Tester linked
```

   Notice how much slower it is! It even skips a test and it still SLOW!

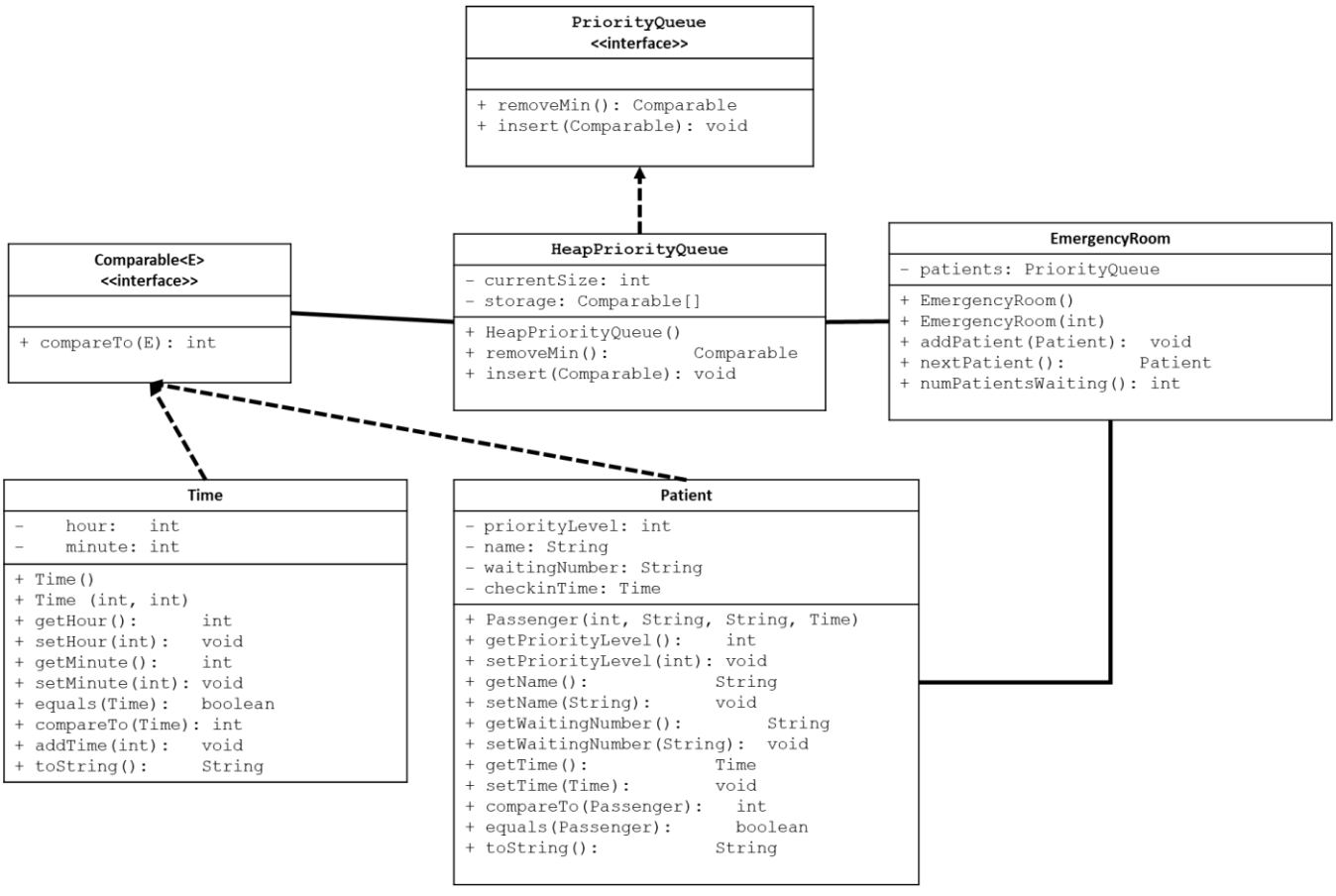
Part II

For this part of the assignment, you will be creating an application to support the modelling of an emergency room in a hospital. You are asked to write the software that will manage handling patients waiting to be helped based on the priorityLevel of their ailment and their arrivalTime.

Imagine… you are checking people in to an emergency room at a hospital. Each time someone arrives they get a waiting number ticket (which is a letter followed by numbers), and are asked to wait until their ticket number is called. You must make sure all high priority patients are helped before lower priority patients. For example:

- A patient (A) enters with a priority level 3 ailment. You add them to the queue.
- A patient (B) enters with a priority level 1 ailment. You add them to the queue ahead of the previous person (A), as priority level 1 ailments **must be handled first**.
- Another patient (C) enters, also with a priority level 1 ailment. You add them to the queue ahead of person A (priority 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 doctor is now ready to help a patient, so person B gets helped and leaves the waiting room (priority queue).
- A patient (D) enters with priority level 2. You add them to the queue ahead of passenger A (priority level 3) but behind passenger C (priority level 1).

The given tester `A6Tester.java` will test the functionality of your `EmergencyRoom` 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`. The classes involved are represented in the UML diagram below. Read the tester and documentation carefully to help you understand how the classes you will be writing will be used.



Submission

Submit only your `HeapPriorityQueue.java, Patient.java` and `EmergencyRoom.java`

Please be sure you submit your assignment, not just save a draft. **ALL late** and **incorrect** submissions will be given a **ZERO** grade.

Even if you choose not to complete all parts of the assignment, you must **submit ALL files** and they **MUST compile with the full, uncommented tester** or you will receive a **ZERO** grade.

If you submit files that do not compile, or that do not use the correct method names you will receive a **zero grade** for the assignment. It is your responsibility to ensure you follow the specification and submit the correct files.

Your code must **not** be written to specifically pass the test cases in the testers, instead, it must work on all valid inputs. We will change the input values, add extra tests and we will inspect your code for hard-coded solutions.

A reminder that it is OK to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution. We will use plagiarism detection software on your assignment submissions.