Prioritizing Streaming Data

Interactively storing and serving data based on priority.

Overview

For this project, you will implement a futuristic automated emergency room (ER) triage system. Your system will receive data in the form of patient and injury pairs over time and automatically assign patients to doctors as the resources become available depending on the severity of the patient's injury. To accomplish this, you will implement a data structure called a *priority* queue – more specifically, a *heap-based priority queue*. A priority queue is similar to a stack or a queue in that it stores and serves data in a specific order. It differs from these two data structures in that the order of data insertion does not affect the order of data serving (though the data clearly must be added to the structure before it can be served). Instead, the order of data serving is based on a *priority* of each datum – in this scenario, the priority of each patient is commensurate with the severity of their injury or age of the patient (depending on the comparator being used). Importantly, a priority queue accomplishes this task in an efficient way. Despite this specific setting, your implementation will be generic enough to work with all types of data and could be applied to other real world problems.

Learning Goals:

- Implement an efficient abstract priority queue using an array-based heap.
- Write code adhering to the provided style guide.
- Use abstract comparators in a meaningful context.
- Demonstrate your ability to read and understand a specification by implementing it.
- Continue developing your ability to write JUnit test cases from a specification.

Project Specification:

Background

In this project, you will be implementing the core of an automatic ER triage system. Essentially, a medical triage is a system in which patients come and go and its behavior is similar to a processor executing tasks following a priority schedule. Patients will arrive one at a time with varying severity of conditions. The system will also provide the possibility of ordering by patient age instead of their condition. With a finite number of doctors and ER beds, we must decide which patients get treated in which order automatically. Naturally, we want the system to have the patients with the more severe conditions be treated first. To accomplish this, you will implement a priority queue using an array-based heap.

The main interface and data structure in this program is a priority queue. Patients are enqueued when they arrive and dequeued as they are treated and released. The priority is on the patient's condition, which has these levels in increasing order of priority: LIGHT, MILD, SEVERE, CRITICAL. Thus, all CRITICAL patients would be treated first, SEVERE patients second, etc. To offer more flexibility, this application can also prioritize patients by age.

The priority queue is implemented with a heap in this application. A heap is a binary tree which is organized based on a *heap-invariant* (the heap ordering property) defined on the priority of each element in the tree. In our case, each element in the tree will represent a specific patient with priority commensurate with the severity of the injury (or their age if that is the chosen priority). We want the patients with worse conditions to be treated first, so our heap invariant is: every child node will have less than or equal severity (priority) than its parent. Therefore, it can be inferred that the root node will always have the highest priority of any element in the heap. This describes a max heap. We must maintain the heap-invariant as patients arrive, enqueued onto the priority queue (added to the heap) and are treated, dequeued from the priority queue (removed from the heap). Note that in this program, the heap can also be configured to function as a min heap. In that case, the root has the lowest priority, and child nodes have priority less than or equal to their parent.

While there are many ways to store a binary tree in memory, we opt for the space efficient array-based allocation. The array implementation also provides a simple way to navigate the heap by calculating parent and child indexes.

Maintaining the Heap Property

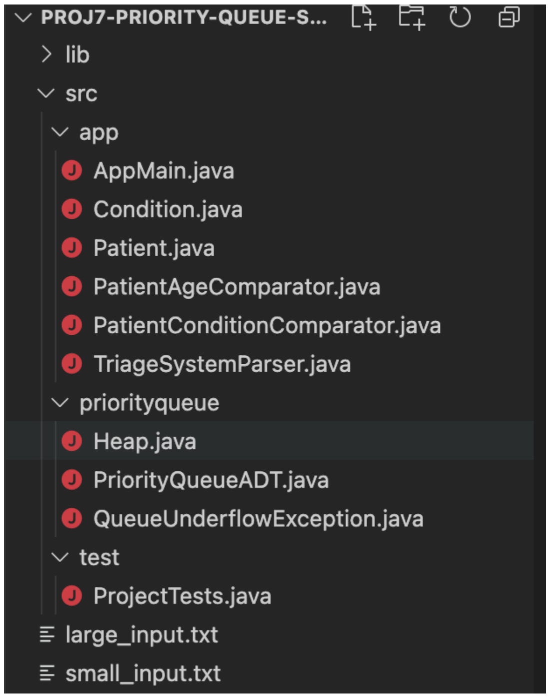
As stated above, there are two cases when the internal structure of the heap might need to change to maintain the heap invariant: adding elements onto the heap and removing elements off of the heap. Since the `Heap` class is implementing a priority queue, it will implement the `PriorityQueue` interface. Therefore, the method for adding a new data element to the Heap will be the `enqueue` method, and the method for removing a data element will be `dequeue`.

Adding an element to the heap. When an element is added to the heap, it is initially stored at the next available index in the array. Then, in order to maintain the heap property, it is swapped with it's parent if and only if it has higher priority than its current parent. Once it's parent has higher priority or it is the new root of the heap, we're done. This procedure is implemented in the `bubbleUp` method. We suggest a recursive approach.

Removing an element from the heap. Only the root is removed from a heap. When removing the root from the heap, first get the data from the root to return, then take the current last element in the array and move it to replace the root element. Then, in order to maintain the heap invariant, we check to see which of the current root's children has higher priority. If the current root has less priority than the greater priority child, we swap the root with this node. Otherwise, the heap invariant is satisfied. We repeat this process of swapping this element down the tree until the heap invariant is obtained. This procedure is implemented in the `bubbleDown` method. We suggest a recursive approach.

Code Structure

In this context, let's briefly analyze the code structure (see Figure 1).



```
PROJ7-PRIORITY-QUEUE-S...
  > lib
  v src
    v app
        AppMain.java
        Condition.java
        Patient.java
        PatientAgeComparator.java
        PatientConditionComparator.java
        TriageSystemParser.java
    v priorityqueue
        Heap.java
        PriorityQueueADT.java
        QueueUnderflowException.java
    v test
        ProjectTests.java
  large_input.txt
  small_input.txt
```

Figure 1: Code structure.

The **`Heap.java`** file contains the definition of a Heap class, which is to be implemented by yourself. The class already has standard methods to help you during your fun journey to implement a top notch priority queue for the triage system. Before diving into the assignment, it is worth mentioning some cool stuff about your Heap implementation. First, it uses Java generics to make the data structure code be valid regardless of which type of elements are stored in it, i.e a `Heap<int>` stores integer values, a `Heap<Patient>` stores patient objects. Second, the Heap accepts a comparator object to define the order (priority) of its elements. Finally, the Heap accepts the boolean flag `isMaxHeap` to define if the instance will be a max heap, i.e highest priority first, or a min heap, i.e lowest priority first.

The **`Patient.java`** file contains all the relevant patient information for your triage. You do not need to implement anything on this file, although you'll probably need to check it multiple times during the assignment. Patients' health conditions are defined in the **`Condition.java`** file with an `Enum`. This is an elegant way to define the constant values of conditions that the triage system uses and, once more, you don't need to write anything in it.

The **`TriageSystemParser`** class implements all the necessary operations to parse a triage input data file (for e.g. **`small_input.txt`** or **`large_input.txt`**) and it is also implemented for you. It reads the entire file to memory and stores its content in the **`patients`** `ArrayList` attribute. Another important thing about this file is that it serves as a practical example of the ***code style guide*** you must follow - removing unnecessary lines of code; proper indenting; optimal use of blank lines and spaces for operators; use of camelCase for variable/parameter and method names; descriptive variable and method names.

The **`AppMain.java`** class contains the main method of the program to test and demonstrate your heap implementation. You can change the input file, type of prioritization (age or condition), and whether you use a max or a min heap. See the **Testing** section below for more details.

`Comparator Interface`

The program makes use of the Java `Comparator` interface. Like the `Comparable` interface, it allows data to be compared and thus ordered. Comparator is more flexible as the way data can be compared can be implemented by different Comparator classes. These are the two Comparators used in this program:

`PatientAgeComparator.java`

The `compare` method performs the comparison between the age attribute of patients pt1 and pt2. The age of a patient is an integer.

`PatientConditionComparator.java`

The `compare` method for this Comparator performs the comparison between the condition attribute of patients pt1 and pt2. The Condition levels in increasing severity are: LIGHT, MILD, SEVERE, CRITICAL. These are modeled by an Enum Object, which encapsulates data types that model a set of discrete values. (take a look at the **`ordinal`** method for **`Enum`**). A LIGHT condition is considered less than a CRITICAL condition, for example.

All classes that implement Comparator must implement the **`compare`** method:

`public int compare(T element1, T element2)`

The `compare` method returns strictly positive numbers when **`element1 > element2`**, strictly negative numbers when **`element1 < element2`**, and zero when **`element1 = element2.`**

You can find more about the `Comparator` interface here Comparator (Java Platform SE 8).

Tasks

To complete the assignment, you must implement all of the methods, marked with a TODO tag in the source code. ***Do not modify any existing instance variables, method signatures, or any of the starter code provided. Do not add any new classes to the project. You may add any additional instance variables or helper methods you wish. Do not import any additional and/or external code libraries.***

`Heap.java`

The TODO list below is sorted in a possible implementation order, although feel free to approach the tasks in any order that you want.

`public Heap(Comparator<T> comparator, boolean isMaxHeap)`

The constructor initializes the type of Comparator to be used as well as a boolean parameter which designates whether the Heap class will act as a min or max heap. Any other class variables can also be initialized in the constructor, such as the array that will implement the heap. The array must be generically typed so it can store any type of Object, and it must be initialized to `INIT_SIZE`.

`public void bubbleUp(int index)`

The `bubbleUp` method is responsible for identifying if the priority of node `i` is smaller than, or equal to, its parent's priority. If this is not the case, `bubbleUp` must be applied together with `swap` operations to fix the heap-invariant. We suggest a recursive approach.

`public void bubbleDown(int index)`
The `bubbleDown` method is responsible for propagating the correct `swap` operations in order to maintain the heap invariant when elements are removed from the heap. We suggest a recursive approach.

Methods defined in the `PriorityQueueADT` interface that `Heap` must implement: See the comments in the starter code.

```
public void enqueue(T item);
public T dequeue() throws QueueUnderflowException;
public T peek() throws QueueUnderflowException;
public boolean isEmpty();
public int size();
```

We suggest you implement the following as helper methods. These can be called by other methods to carry out the tasks of `enqueue, dequeue bubbleUp, and bubbleDown`. Note: We do not test the private methods.

```
private int getLeftChildOf(int parentIndex)
private int getRightChildOf(int parentIndex)
private int getParentOf(int childIndex)
private void swap(int index1, int index2)
private void expandCapacity()// since the array may need to be lengthened.
```

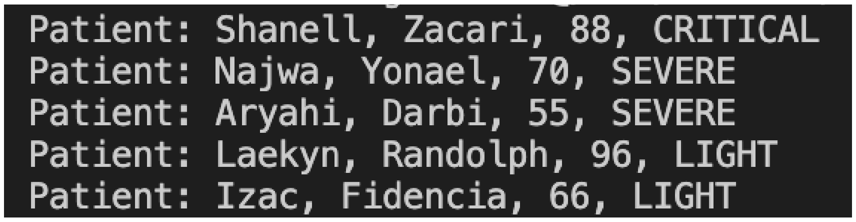
Testing

We suggest that you develop your code by debugging from the main method instead of the unit tests. You can alter the main method code to fully test your `Heap` class methods. Once you have a working program, run the unit tests, and if they pass, submit the project to Gradescope.

Ways you can test your code from the main method in `AppMain.java`:

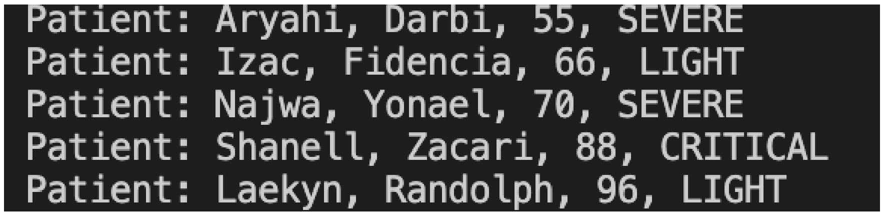
1. Initially, the code reads in data from the `small_input.txt` file. You should open that file just to see how the data for patients is stored. You can also test your code on the `large_input.txt` file as well.
2. The `Comparator` classes can be set to prioritize age or condition. Try both to test that your code is correctly ordering on that priority.
3. Change the priority of the queue from max to min by setting `isMaxHeap` to `false`. Check that your code works correctly as a `minHeap`.
4. Use the debugger to visualize the way data is stored on the array and how it is adjusted upon `enqueue` and `dequeue` operations. Note that the array length will be >= the amount of data in the heap. The array capacity will have to be expanded when full.

Here is the output for the data in **`small_input.txt,`** based on severity of patient condition. When the conditions of two patients are the same, then the oldest one comes first.



```
Patient: Shanell, Zacari, 88, CRITICAL
Patient: Najwa, Yonael, 70, SEVERE
Patient: Aryahi, Darbi, 55, SEVERE
Patient: Laekyn, Randolph, 96, LIGHT
Patient: Izac, Fidencia, 66, LIGHT
```

Here is the output for the data in **`small_input.txt`** ordered on patient age from youngest to oldest.



```
Patient: Aryahi, Darbi, 55, SEVERE
Patient: Izac, Fidencia, 66, LIGHT
Patient: Najwa, Yonael, 70, SEVERE
Patient: Shanell, Zacari, 88, CRITICAL
Patient: Laekyn, Randolph, 96, LIGHT
```

Export and Submit

Step 1: Export your project
Use the Archive extension or other means to create the correct zip file. If your zip file is not in the correct form, the autograder will not process your code and you will not receive any credit.

Step 2: Submit the zip file to Gradescope
Log into Gradescope, select the assignment, and submit the zip file for grading.

There are usually more tests in the autograder than provided with the starter code. If your code passes all provided tests, it is a good indication that your code will pass all of the autograder tests. If that is not the case, you are likely not considering some of the “edge” cases in your algorithm. Re-examine your code, use the debugger to troubleshoot. Pose specific questions on Piazza for some outside help. Remember that these projects take longer than you estimate. Starting early means you have more time to get help if you are stuck.

The autograder will not run successfully if you do submit a **correctly formatted zip file**- it has to have the same **names and directory structure as described above**. The autograder will also not run if your code **does not compile**, or if you i**mport libraries** that were not specifically allowed in the instructions.

Remember, you can re-submit the assignment to gradescope as many times as you want, until the deadline. If it turns out you missed something and your code doesn't pass 100% of the tests, you can keep working until it does. Attend office hours for help or post your questions in Piazza.

Academic Honesty Policy Reminder

Copying partial or whole solutions, obtained from other students or elsewhere, is **academic dishonesty**. Do not share your code with your classmates, and do not use your classmates' code. Posting solutions to the project on public sites **is not allowed**. If you are confused about what constitutes academic dishonesty, you should re-read the course policies.