Assignment 4
COMP 250, Fall 2021

Prepared by Prof. Michael Langer and T.A.s

Posted: Fri. Nov. 19, 2021
Due: Fri. Dec. 3 at 23:59 (midnight)

[document last modified: November 24, 2021]

General instructions (similar to Assignments 1-3)

- Search for the keyword *updated* to find any places where the PDF has been updated.
- T.A. office hours will be posted on mycourses under the “Office Hours” tab. For zoom OH, if there is a problem with the zoom link or it is missing, please email the T.A. and the instructor.
- If you would like a TA to look at your solution code outside of office hours, you may post a *private* question on the discussion board.
- We will provide you with some examples to test your code. See file ExposedTests.java which is part of the starter code. If you pass all these tests, you will get at least 60/100. These tests correspond exactly to the exposed tests on Ed. We will also use private tests for the remaining points out of 100, as we did on Assignment 3.

  We strongly encourage you to come up with more creative test cases, and to share these test cases on the discussion board. There is a crucial distinction between sharing test cases and sharing solution code. We encourage the former, whereas the latter is a serious academic offense.

  Ed should be used for code submission only. You should write, compile and test your code locally (e.g. in Eclipse or IntelliJ) for development.
- **Policy on Academic Integrity:** See the Course Outline Sec. 7 for the general policies. You are also expected to read the posted checklist PDF that specifies what is allowed and what is not allowed on the assignment(s).
- **Late policy:** Late assignments will be accepted up to two days late and will be penalized by 10 points per day. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

Please see the submission instructions at the end of this document for more details.

Introduction

The main purpose of this assignment is to give you some "hands on" experience with a heap implementation of a priority queue. For the heap that was presented in the lectures, users can add and remove priorities and can access the min priority. However, that heap only stores priorities; it does not store the objects that have these priorities.[1] Such a heap on its own is not so useful since removing the minimum priority doesn't help us if we don't know which object is associated with that priority. Instead, we write a priority queue class that uses a heap but represents a set of (object, priority) pairs rather than just priorities. For our example, the objects will be strings. The class also allows one to perform more general operations such as changing the priority associated with the string object.

In addition to working with a heap, this assignment will also give you some experience working with a hash map. The reason you need a hash map is given below.

ERPriorityQueue

Imagine the emergency room (ER) of a hospital. Patients who come to the ER are assessed (triage). We associate the urgency of the patient's problem with a single value: the priority.[2] As in the heap lectures, we use a min heap to cater to patients with more urgent situations first. So three patients with priority values say 3, 4, 8 would be seen in that order. We use double valued priorities since we want to have flexibility, for example, to add a patient with priority 3.2 between patients with priorities 3 and 4. You should assume the priorities are always be strictly greater than 0.

The ERPriorityQueue class allows us to organize (name, priority) pairs using nodes which are stored in a heap. Each node in the heap is an instance of a **Patient** class which has two fields (name, priority). The heap implementation will use an ArrayList of these Patients. The Patient class is given to you in the starter code. For grading purposes, we have included a toString() and equals() method in the Patient class. For technical reasons having to do with the grader, we have also made Patient a static inner class.

The ERPriorityQueue class has two fields: an ArrayList of Patient objects called **patients** and a HashMap **nameToIndex**. Both of these fields are public for grading purposes. The HashMap nameToIndex is defined as follows: the key is a string and the value is the index of that string in the ArrayList. This allows us to find a patient (given their name) in the priority queue.

The ERPriorityQueue class has many methods. Some of the methods are already implemented for you, namely a constructor, parent(), leftChild(), rightChild(). The rest are described below, and your task will be to implement them.

Heads up: if you read about heaps, you will find that one often refers to the priorities of a heap as "keys". However, in our example, we also work with a hashmap which has a set of (key,value) pairs, where the key is the patient name. Using the term "key" for both the priority and the name would be confusing. So in this assignment, we use the term "key" only when referring to the names in the namesToIndex map.

[1]This is very different from the Java PriorityQueue class, which stores objects that can be compared to each other. However, for this assignment, you are not allowed to use the Java PriorityQueue class.

[2]We ignore the differences between patients, such as whether they need an xray or blood test or covid test, etc.

Your Task

Implement the following methods in the ERPriorityQueue class. The starter code contains the method signatures for all of these methods. Note that some of these methods would naturally be private helper methods; we make them all public to simplify the grader implementation. We also suggest that you add your own helper methods, in particular, swap(), isEmpty(), isLeaf().

You may assume that your methods will be tested on valid inputs only e.g. indices that correspond valid slot locations in the ArrayList for the first two methods below. However, for testing purposes, you may find it helpful to add your own checks for validity.

- upHeap(int index) **(10 points)** – returns nothing (void); performs the upHeap operation starting at the given index. Your implementation should have worst case time complexity $O(\log_2 n)$. [updated: Nov. 23] Unlike in the heap lectures, in this assignment we allow equal priority values. To handle this possibility for upHeap, swap until the priority of the element's parent is less or equal to the priority of the element being upheaped.

- downHeap(int index) **(10 points)** – returns nothing (void) ; performs the downHeap operation starting at the given index. Note that the method signatures of upHeap and downHeap are slightly different from the one given in the pseudocode in the lectures. Your implementation should have worst case time complexity $O(\log_2 n)$. [Updated: Nov 23] As mentioned in upHeap above, we allow equal priority values. Therefore, swap until the priority of the element is less than or equal to the minimum priority of the element's two children. If there are two children that have an equal priority that is less than the priority of the element being downheaped, then swap the element with the left child. (A strict policy is needed here for grading purposes.)

- contains(String name) **(5 points)** – returns a boolean, indicating whether the given string (patient name) is in the heap. Your implementation should have time complexity $O(1)$.

- getPriority(String name) **(5 points)** – returns a double which is the priority associated with the given patient name; all patient priorities will be greater than 0; return -1 if the given name does not correspond to a patient. Your implementation should have time complexity $O(1)$.

- peekMin() **(3 points)** – returns the name of the patient with minimum priority; returns **null** if the queue is empty. [Updated: Nov 20.] Your implementation should have time complexity $O(1)$.

- getMinPriority() **(2 points)** – returns the minimum priority of all patients in the queue; returns -1 if the queue is empty. Your implementation should have time complexity $O(1)$.

- removeMin() **(10 points)** – returns a string – removes the patient that has the minimum priority and returns their name; returns null if queue is empty. Your implementation should have worst case time complexity $O(\log_2 n)$.

- add(String name, double priority) **(10 points)** – adds a new patient to the queue; returns a boolean, indicating whether or not the name was added; return false only if the patient is already in the queue; Your implementation should have worst case time complexity $O(\log_2 n)$.

- add(String name) **(5 points)** – adds a new patient to the queue but now the priority is a default, namely the double value `POSITIVE_INFINITY` [Updated: Nov. 22]. Likewise, it returns a boolean, indicating whether or not the name was added; return false only if the patient is already in the queue. Your implementation should have worst case time complexity $O(1)$. [updated: Nov. 23]

- remove(String name) **(10 points)** – removes the patient from the priority queue; it returns a boolean: true if the patient with that name was indeed in the queue and false otherwise;

  Your implementation should have worst case time complexity $O(\log_2 n)$. [Updated: Nov. 22] Similar to the removeMin() method from the lecture, it should replace the node to be removed with the last element of the arraylist. Then, it must only modify the ancestors and/or descendants of the removed node.

- changePriority(String name, double newPriority) **(10 points)** – modifies the priority of the patient and adjusts the priority queue, if necessary; returns a boolean – returns true if the patient is indeed in the heap and false otherwise

  The idea of this method is that it can happen that the priority of patients in the waiting room can change over time. The patient may have a test done at the hospital (blood test, x-ray, covid, etc) and may get a positive or negative result that changes their priority. Your implementation should have worst case time complexity $O(\log_2 n)$. [Updated: Nov. 22] In particular, it must only modify the ancestors and/or descendants of the node for which we changed the priority and must have a best case behavior of $O(1)$, e.g. if no modification is required.

- removeUrgentPatients(double threshold) **(10 points)** – removes the patients whose priority value is less than or equal to the given threshold; you can assume that the threshold is positive. Recall that having a low priority value means having a "high priority" i.e. urgent. [Updated: Nov 23] The method returns an ArrayList of the urgent patients that have been removed. Your implementation should have worst case time complexity $O(n \log_2 n)$.

- removeNonUrgentPatients(double threshold) **(10 points)** – removes the patients whose priority value is greater than or equal to the given threshold; you can assume the threshold is positive. [Updated: Nov 23] The method returns an ArrayList of the non-urgent patients that have been removed. Your implementation should have worst case time complexity $O(n \log_2 n)$.

Submission

Please follow the instructions on Ed Lessons Assignment 4.

- Ed does not want you to have a package name. Therefore you should remove the package name before uploading to Ed.

- We will check that your file imports only what you need, namely ArrayList, HashMap. If you use other imports, your code will not pass this check and you will not be able to submit.

- You may submit multiple times. Your assignment will graded using your most recent submission.

- If you submit code that does not compile, you will automatically receive a grade of 0. Since we grade only your latest submission, you must ensure that your latest submission compiles!

- The deadline is midnight Fri. Dec. 3. On Ed, this deadline is coded as 12:00 AM on Saturday Dec. 4. Similarly, on Ed, the two day window for late submission ends at 12:00 AM on Monday Dec. 6.

Good luck and happy coding!