CS 251 Summer 2020
Project 2: Priority Queues and Hashtables
Due date:

General Guidelines

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. Keep in mind that anything that does not *need* to be public should generally be kept private (instance variables, helper methods, etc.).

Unless otherwise stated in this handout, you are welcome (and encouraged) to add to/alter the provided java files as well as create new java files as needed.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor!

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work!

Note on using starter code: You are not required to use the starter code. However, keep in mind that (1) some of the API methods are already implemented for you and (2) if you do not use the starter code, you still need all of the defined methods below without changing any return types or names. *Your code must work with the tests that are provided.*

Note on grading and provided tests: The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

Project Overview

In this project, you will implement a processing queue for processing patients at a clinic. There are 4 parts to this project, each with its own set of tests. You are encouraged to run the tests after each part is completed before moving on.

In the medical profession, it does not usually make sense to process patients in a strict FIFO way. Instead, the urgency level of each patient must be taken into account so that those in need of immediate attention can be seen before those with milder maladies. You will be implementing a processing system for patients at a clinic. This clinic has a specific *capacity* and is set up to

handle patients with urgency levels up to and including a specific emergency threshold (*er_threshold)*, after which patients must be sent to the ER. If the clinic is full, the patient with the highest urgency level is also sent to the ER in order to make room.

The *Patient.java* class is already implemented and provided for you. You may add to the class, but do not change any of the variables and methods already implemented. It is highly recommended that you take some time to look through this file to see what operations are already available to you. They may be useful later on. The following API summarizes the methods that are available in *Patient.java*.

Patient.java API

Patient(String name, int urgency, long time_in)	constructor: creates a Patient with the given name*, urgency level, and time_in
String name()	returns Patient's name
int urgency()	returns Patient's urgency level
void setUrgency(int urgency)	sets a Patient's urgency level
long time_in()	returns Patient's time in
long compareTo(Patient other)	compares two Patients according to prioritization (positive result means higher priority relative to *other*); priority is based on (1) urgency level and (2) time in, meaning that a higher urgency level automatically gets priority; if the urgency levels are equal, an earlier time in gets priority
String toString()	returns a String containing Patient's name, urgency level, and time in
String toStringWithPos()	returns a String containing Patient's name, urgency level, and position in queue
String toStringForTesting()	returns a String containing Patient's name and urgency level

int posInQueue()	returns the index at which the Patient is stored in PatientQueue (-1 if the Patient is not in the PQ)
void setPosInQueue(int pos)	sets Patient's *posInQueue* variable to the passed in *posy*

*You can assume that Patient names are unique.

Part 1. PatientQueue.java (Basic Operations)

You must implement an **array-based max-heap priority queue** to process the patients. Your implementation must include the following methods.

Warning: If you do not implement the PQ as an array-based max-heap PQ, you will get a 0 for Part 1.

PatientQueue.java

PatientQueue(int capacity)	Create a new PQ with given capacity
int insert(Patient p)	insert the patient into the PQ and adjust the order as necessary; return the index of the Patient's final position in the queue; if the PQ is full, return -1
Patient delMax()	remove and return the patient with the highest urgency level; adjust the heap as necessary after removal; return null if the queue is empty
int size()	return the number of patients currently in the PQ
Patient getMax()	return but do not remove the Patient with the highest urgency level; return null if the PQ is empty
boolean isEmpty()	return true if the PQ is empty; false else

Patient[] getArray()	returns the underlying structure for testing

Part 2. PHashtable.java

A simple binary heap PQ has good runtime for both *insert* and *delMax*, but this alone is not ideal for the clinic's queue because sometimes patients who start out with a low emergency level might experience an emergency while waiting. Likewise, some patients may get tired of waiting and walk out. This requires us to be able to *update* patients in the queue and *remove* specific patients from the queue. This means, though, that we need to first find the specific patient in the queue (based on the name), and PQs are not built for fast searching. The good news is that with the help of another data structure, we can do this in reasonably fast time.

In this section, you will implement a basic hashtable that will store the patients for quick searching. The key value is the patient's name, and you should use the built-in Java hashcode function for Strings, as well as modular hashing.

You will implement the hashtable using separate chaining to handle collisions. However, since we know the capacity of the clinic, we can set the array size to be that capacity, which will make our *expected* runtime for *put* and *get* and *remove* all $O(1)$.

You may use ArrayList objects to implement the chains. You should only initialize a new list when necessary. Otherwise, leave the array element null.

The following methods must be implemented in PHashtable. There are also two helper methods provided for getting an appropriate prime number.

PHashtable.java API

PHashtable(int capacity)	Constructor: create a new PHashtable with the underlying array having size *p* where *p* is the first prime number that is greater than or equal to *capacity*
Patient get(String name)	return the Patient with the given name from the table; if the Patient is not in the table, return *null*
void put(Patient p)	put the Patient *p* into the table;

	if the Patient already exists in the table, do nothing
Patient remove(String name)	remove and return the Patient with the given name; if the Patient does not exist in the table, return *null*
int size()	return the number of patients in the table
ArrayList<Patient>[] getArray()	returns the underlying structure for testing

Note: For the hash function, you should use Java's hashcode function for Strings and use modular hashing on the result to make it fit into the table. However, sometimes you may get a negative number. You should deal with that by adding the size of the modulus to the negative value. For example, if *s* is the String key, and *n* is the size of the hashtable, and $h = s.hashCode()\%n < 0$, you should reset *h* to $h + n$.

Part 3. NewPatientQueue (Advanced Operations)

NOTE: You should start this section by copying your existing *PatientQueue.java* into a new file called *NewPatientQueue.java.* This will save you unnecessary work and ensure that your alterations don't break the Part1 tests. This shouldn't happen if you do it correctly, but just in case…

Now that you have a Hashtable for the patients, you can store pointers to each patient both in the PQ and in the HT. This allows fast searching (by patient name), and once you retrieve the Patient object, you can access the patient's position in the PQ and thereby access it quickly, allowing for more advanced operations in the PQ in reasonably fast time.

NOTE: If you implement this part correctly, it should still pass the earlier test cases, but it is recommended that you back up your original PatientQueue file so that you don't lose your original work. You should start by adding a PHashtable instance variable.

The following additional methods must be implemented in *NewPatientQueue.java* in addition to those that already exist from Part 1. Also, keep in mind that *some of the existing methods might need to be updated as well.*

NOTE: Assuming that *get* and *put* and *remove* are all expected to be $O(1)$ operations in your hashtable (a reasonable assumption based on the specifications), each of the following functions should be no worse than $O(logN)$ where N is the number of patients in the PatientQueue. Also, remember, that the binary heap must always be in the correct order after every operation!

NewPatientQueue.java API (advanced)

Patient remove(String s)	remove and return the Patient with name *s* from the PatientQueue
void update(String s, int urgency)	update the emergency level of the patient with name *s* to *urgency* in the PatientQueue

Part 4. Clinic.java

Now you will put everything together. The Clinic class models how patients are processed as they come in to the clinic. The clinic is created with a given *capacity* and an *er_threshold*, indicating when the clinic becomes too crowded or when a patient needs to be upgraded to the ER.

The following methods must be implemented in *Clinic.java.* The program must also keep track of the number of patients who are processed, the number of patients who are sent to the ER, the number of patients who are seen by a doctor, and the number of patients who walk out. There are already variables available for this, and for the most part, they are incremented in the necessary methods.

Clinic.java API

Clinic(int cap, int er_threshold)	Constructor: create a new Clinic with capacity *cap* and the given *er_threshold*
int er_threshold()	return the *er_threshold*
int capacity()	return the *capacity*
String process(String name, int urgency)	Process a new patient with the given name and urgency level; if the patient's urgency level exceeds *er_threshold* send them directly to the ER and return null; otherwise, try adding them to the Queue; if the Queue is not

	full, return their name; otherwise, do the following: ● if the new patient's urgency level is higher than the current max, send the new patient to the ER and return null ● if the new patient's urgency level is less than or equal to the current max, delete the max and send the max patient to the ER, then insert the new patient into the queue; return the name of the max patient
void seeNext()	The patient with highest precedence that is currently in the queue will get seen by a doctor; return the name of the patient who is seen (or null if the queue is empty)
boolean handle_emergency(String name, int urgency)	Patient with name *name* experiences an emergency, causing their urgency level to increase to *urgency;* if their urgency level exceeds *er_threshold*, send them to the ER and return true; else, update their *urgency* level in the PatientQueue and return false (i.e. return true if they are removed from the queue)
void walk_out(String name)	Patient with name *name* walks out; remove them from the PatientQueue

The private methods *sendToER(Patient p)* and *seeDoctor(Patient p)* are provided for you and should not be changed! There are also getter methods for the four counter variables that should not be changed.

Submission. You must include the following files in your submission, as well as any additional *java* files that are needed for your submission.

- Patient.java

- PatientQueue.java
- PHashtable.java
- NewPatientQueue.java
- Clinic.java

Testing

You are provided with the following test files.

- Text files:
  - *patient_file_1.txt*
  - *p1_test3_output_25_19.txt*
  - *p2_test3_output_25_12.txt*
- *Part1Test.java:* This tests *PatientQueue.java* and takes the name of a patient input file (i.e. *patient_file_1.txt*) as an argument. It also uses *p1_test3_output_25_19.txt* to test the underlying data structure. Note that if you fail Test 3 (testing the data structure), you receive a 0 for Part 1, so don't try to get around using the array!
- *Part2Test.java*: This tests *PHashtable.java* and takes the name of a patient input file (i.e. *patient_file_1.txt*) as an argument. It also uses *p2_test3_output_25_12.txt* to test the underlying data structure. Note that if you fail Test 3 (testing the data structure), you receive a 0 for Part 2, so don't try to get around using the array of lists!
- *Part3Test.java:* This tests *NewPatientQueue.java* and takes the name of a patient input file (i.e. *patient_file_1.txt*) as an argument.
- *Part4SimTest.java*: This tests *Clinic.java* (and everything else along with it) by running a simulation. Don't be surprised if running this helps you discover new errors in other parts of your code! The arguments for running the simulation are:
  `<patient file name> <clinic capacity> <er_threshold> <pWalkin> <pDocAvailable> <pEmergency> <pWalkOut>`
    - `patient file name: the name of the patient input file`
    - `clinic capacity: the capacity for your clinic (an integer)`
    - `er_threshold: the threshold emergency level for sending Patients to the ER (an integer)`
    - `pWalkin: the probability (in percentage form) that a patient will walk in (i.e. 45 for 45%)`
    - `pDocAvailable: the probability (in percentage form) that a doctor is available (i.e. 56 for 56%)`
    - `pEmergency: the probability (in percentage form) that a patient experiences an emergency (i.e. 17 for 17%)`
    - `pWalkOut: the probability (in percentage form) that a patient walks out (i.e. 34 for 34%)`

It is recommended that you run the simulation several times with several different sets of arguments before you submit.

Grading Rubric

A. **Part 1 (25 points)**
B. **Part 2 (25 points)**
C. **Part 3 (25 points)**
D. **Part 4 (25 points)**