PA 4
Due Date: Friday 12 April 2024 by 11:59 PM

General Guidelines.

The APIs given below describe the required methods in each class. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment.

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, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! It is also considered academic dishonesty to share your code with another student. Anyone who is found to have violated this policy will receive an automatic 0 on the assignment and may be subject to further consequences according to the university's policies.

Note on grading and provided tests: The provided tests are to help you as you implement the project and (in the case of auto-graded sections) to give you an idea of the number of points you are likely to receive. Please note that the points indicated when you run these tests locally are not your final grade. Your solution will be graded by the TAs after you submit. Please also note that these test cases are not likely to be exhaustive and find every possible error. Part of programming is learning to test and debug your own code, so if something goes wrong, we can help guide you in the debugging process but we will not debug your code for you.

Project Overview.

In this project, you will build a system for a Clinic simulation that handles patients according to urgency levels.

Part 1. Patient.java

In a file called `Patient.java`, write a class representing a patient. The following API gives you the required methods (which of course should be `public`), but you can always include other methods as needed, but make good design choices about what needs to be public vs. what can be private. Each patient has a *name*, an *urgency level* (which is an integer where a higher value means greater urgency), and a *time in* (which is represented as a long so that a lower value means the patient came in earlier).

Note: The Patient class should implement the Comparable interface.

Patient.java API

Method Signature	**Description**
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
int 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, time in, and position in the queue. These values should be separated by commas. For example: "Alice Jones, 10, 1000, 3"

*You can assume that Patient names are unique.

Part 2. PHashtable.java

In a class called PHashtable.java, implement a hashtable for storing patient information. The score you get on this section will vary depending on your collision management method. If you use separate chaining or linear probing correctly you will get full credit. But if you use quadratic probing, you can get a few extra points.

The following API gives you the required methods (which of course should be `public`), but you can always include other methods as needed, but make good design choices about what needs to be public vs. what can be private.

PHashtable.java API

Method Signature	**Description**
`PHashtable()`	`Constructor–creates an empty hashtable with a starting capacity of 11`
`PHashtable(int cap)`	`Constructor–creates an empty hashtable with a starting capacity of` *`cap`*
`Patient get(String name)`	`Return the Patient with the given` *`name`*`.`
`void put(Patient p)`	`Put Patient` *`p`* `into the table.`
`Patient remove(String name)`	`Remove and return the Patient with the given` *`name`*`.`
`int size()`	`Return the number of patients in the table.`

Specific Guidelines:

- Assume that keys (names) are unique, so if *put* is called on a name that already exists in the table, you should update the value in the table.
- Manage your load factor the way we discussed in class (depending on your collision management system).
- When you resize, make sure you resize so that the amortized expected cost of the operations is O(1).
- Make your table size a prime number, even when resizing.
- Don't resize the table to be less than the starting capacity.

Part 3. PatientQueue.java

In a class called `PatientQueue.java`, implement a data structure for managing the patients according to their priority level. You can compare their priorities using the *`compareTo`* method you wrote for the `Patient` class. For this part, the data structures you are allowed to use are: (1) one array and (2) one PHashtable.

In a file called `Patient.java`, write a class representing a patient. The following API gives you the required methods (which of course should be `public`), but you can always include other methods as needed, but make good design choices about what

needs to be public vs. what can be private. Make sure you pay attention to the extra requirements for these methods.

PatientQueue.java API

Method Signature	**Description**	**Additional Requirements**
PatientQueue()	Constructor: creates a new PatientQueue with a starting capacity of 11.	
PatientQueue(int cap)	Constructor: creates a new PatientQueue with a starting capacity of *cap*.	
void insert(Patient p)	Insert patient *p* into the queue.	The expected runtime should be no worse than O(logN) where N is the number of patients currently in the queue.
Patient removeNext() throws EmptyQueueException*	Remove and return the next patient in the queue, which should be the person with the highest priority according to the *compareTo* method of Patient. Throw the exception if the queue is empty.	The expected runtime should be no worse than O(logN) where N is the number of patients currently in the queue.
Patient getNext() throws EmptyQueueException*	Return but do not remove the next patient in the queue, which should be the person with the highest priority according to the *compareTo* method of Patient. Throw the exception if the queue	The worst-case runtime should be no worse than O(1).

	is empty.	
int size()	Return the number of patients currently in the queue.	The worst-case runtime should be no worse than O(1).
boolean isEmpty()	Return true if the queue is empty and false if it isn't.	The worst-case runtime should be no worse than O(1).
Patient remove(String name)	Remove and return the patient with the given *name*. Return null if the patient is not in the queue.	The expected runtime should be no worse than O(logN) where N is the number of patients currently in the queue.
Patient update(String name, int urgency)	Update the urgency level of the patient with the given *name*. The new urgency value is passed as parameter *urgency*. Return the updated patient or *null* if the patient does not exist in the table.	The expected runtime should be no worse than O(logN) where N is the number of patients currently in the queue.

* The EmptyQueueException class is provided.

Specific Guidelines:

- Assume that keys (names) are unique.
- Your structure should be dynamic, so you will need to resize when appropriate. When you resize, make sure you resize so that the amortized expected cost of the operations is O(logN).
- Don't resize the queue to be less than the starting capacity.

Part 4. Clinic.java

In a file called `Clinic.java`, implement a class to represent a clinic according to the API given below. Make sure you pay attention to the additional requirements for each method. Your Clinic will use the PatientQueue you implemented earlier.

Clinic.java API

Clinic(int er_threshold)	Constructor: Create a new Clinic instance. The er_threshold indicates the highest level of urgency the clinic can handle. If a patient's urgency level exceeds that, they are sent to the ER instead.	
int er_threshold()	Return the ER threshold.	
String process(String name, int urgency)	Process the new patient with the given *name* and *urgency*. If their urgency level exceeds *er_threshold*, send them to the ER. If not, add them to the patient queue. Return the name of the patient.	The expected runtime should be no worse than O(logN) where N is the number of patients currently in the queue.
String seeNext()	Send the next patient in the queue to be seen by a doctor.	The expected runtime should be no worse than O(logN) where N is the number of patients currently in the queue.
boolean handle_emergency(String name, int urgency)	A patient with the given *name* experiences an emergency while waiting. Update their urgency level. If it exceeds the er_threshold, send them directly to the ER (and remove them from the queue). Otherwise, update their urgency level. Return true if the Patient was sent to the ER and false otherwise.	The expected runtime should be no worse than O(logN) where N is the number of patients currently in the queue.
void walk_out(String name)	A patient with the given *name* walks out before being seen by a doctor.	The expected runtime should be no worse than O(logN) where N

	`Remove them from the queue.`	`is the number of patients currently in the queue.`
`int processed()`	`Return the number of patients that have been processed.`	
`int sentToER()`	`Return the number of patients that have been sent to the ER.`	
`int seenByDoctor()`	`Return the number of patients that have been seen by a doctor at the clinic.`	
`int walkedOut()`	`Return the number of patients that walked out before being seen by a doctor.`	

In addition to the logic described above, your Clinic should also keep track of

- the number of patients that are processed
- the number of patients seen by a doctor
- the number of patients sent to the ER
- the number of patients who walked out before being seen by a doctor

Part 5. Simulation.java

For this part, you don't actually have to implement anything as the Simulation is already implemented. But you do need to run your code with the simulation to make sure it all works properly.

Testing & Grading

You are provided with the following test files.

- Text files:
  - *patient_file_1.txt*
- *PHashtableTest.java*: This tests *PHashtable.java*.
- *PatientQueueTest.java:* This tests *PatientQueue.java*.
- *Simulation.java*: Run simulation to see the results of using your new classes to simulate a Clinic. It expects the following command-line arguments:
  `<er_threshold> <pWalkin> <pDocAvailable> <pEmergency> <pWalkOut>`

- `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 in order to make sure your results look correct.

The grading is based on the following, but as usual, there are possible deductions that can be made based on what we see in your code.

Item	**Total Points**
`PHashtable`	`Autograder = 14` `Additional points for using quadratic probing correctly: 5`
`PatientQueue`	`Autograder = 22`
`Clinic`	`Simulation = 14`

A few more notes about grading:

- We use the autograder to give us an idea of the correctness of your implementation, but we can also adjust grading based on what we see when we look at your code.
- Your implementation needs to work well with the simulation but also produce reasonable results.
- Possible deductions include (but are not necessarily limited to):
    - Code does not compile or does not run on lectura (up to 100% of the points).
    - Bad Coding Style (up to 5 points) – this is generally only deducted if your code is hard to read, which could be due to bad documentation, lack of helpful comments, bad indentation, repeating code instead of abstracting out methods, writing long methods instead of smaller, more reusable ones, etc.

- Not following directions or meeting runtime requirements (up to 100% of the points depending on the situation)
- Late Submission (5 points per day)--see the policy in the syllabus for specifics
- Inefficient code (up to 50% of the points)

Submission Procedure.

To submit, please upload the following files to **lectura** by using **turnin**. Once you log in to lectura, you can submit using the following command:

```
turnin csc345pa4 Patient.java PatientQueue.java PHashtable.java Clinic.java
```

Upon successful submission, you will see this message:

```
Turning in:
     Patient.java -- ok
     PatientQueue.java -- ok
     PHashtable.java -- ok
     Clinic.java -- ok
All done.
```

Note: Your submission must be able to run on **lectura**, so make sure you give yourself enough time before the deadline to check that it runs and do any necessary debugging. I recommend finishing the project locally 24 hours before the deadline to make sure you have enough time to deal with any submission issues.