CS 5010: Programming Design Paradigm, Fall 2017

HW2 – ADTs and Object-Oriented Design

Submission is due October 2nd, 6 pm

Assignment Goals

- To understand the behavior of stacks and queues
- To be able to implement priority queue operations
- To define a data structure using Java generics
- To propose design of Emergency Room simulator and implement it using Priority Queue

---

There are two sections of problems in this assignment:

Section 1: Testing Implementations of ADTs

This section describes two ADTs that access elements of a collection in different orders – Stack and Queue. The ADT's operations, description of the ADT's properties, and an example sequence of operations are given. Here we assume each ADT operates on collections of integers.

The goal:

Your job is to **write a collection of Junit tests** that would determine whether **any** provided implementation of each ADT satisfies the respective ADT's properties:

- If your tests run against a **correct implementation** of the ADT→ **all your tests must pass.**
- If your tests run against an **incorrect implementation** of an ADT → **at least one of your tests must fail.**

Thus, your challenge is smartly design test cases that cover the expected behavior of the ADT; it is NOT about the sheer number of tests that you propose.

Your submission should include:

- Provide a Junit file `QueueTest.java` containing a single `QueueTest` class with all your tests for Queue ADT and `StackTest.java` containing a single `StackTest` class for Stack ADT.
- Do NOT worry about tests on undefined cases that would yield exceptions/errors (like trying to remove an element from an empty data structure).
- Compile your `QueueTest.java` and `StackTest.java` file against the provided interfaces and dummy classes to make sure you match the names we will assume when grading your work.
    - Note that the provided dummy classes are NOT the correct implementations of the ADTs. Your test cases will fail against the dummy file. What matters is that your code ***compiles*** when using the dummy classes.
- **You are NOT being asked to implement these ADTs. Your goal is to define test cases.**

Here are the ADTs:

1. **Stack - LIFO-order ("last in, first out")**

A Stack is an ADT for accessing elements of a set in the order of most recently added to least-recently added. The operations on stacks are:

```
push: Stack, int → Stack // adds an integer element to the Stack

pop: Stack → Stack       // removes the most recently-added element

top: Stack → int // returns, but does NOT remove, the most-recently
                                                        added element
```

Example:

- Assume that elements 7, 4, and 5 are pushed to a new stack (in that order).
- Calling `pop` would produce a stack containing 4 and 7.
- Calling `top` on that stack would produce 4.
- Calling `pop` on that stack again would produce a stack containing 7.

2. **Queue - FIFO-order ("first in, first out")**

A Queue is an ADT for accessing elements of a set in the order of least-recently added to most-recently added. The operations on queues are:

```
enqueue: Queue, int → Queue // adds an element to the Queue

dequeue: Queue → Queue   // removes the least recently-added element

front: Queue →int    // returns, but does NOT remove, the least-recently
                                                        added element
```

Of course, you can assume a constructor for both Stack and Queue implementations. The constructors will return Stack/Queue with NO elements.

Example:

- Assume that elements 7, 4, and 5 are enqueued to a new queue (in that order).
- Calling `dequeue` would produce a queue containing 4 and 5.
- Calling `front` on that queue would produce 4.
- Calling `dequeue` on that queue again would produce a queue containing 5.

NOTES:

- "→" refers to the output TYPE of the method
- Please see supplementary files for the EXACT interface and DUMMY concrete implementation

Section 2: Emergency room simulator

Problem formulation:

Seattle General Hospital has just been bought out by an HMO, and it is budget-crunch time. You have been hired to model the flow of patients in their emergency room and answer some key questions:

1. "How long do patients wait to be treated?"
2. "What is the average treatment time?"
3. "What is an optimal number of treatment rooms that balances cost against patient service?"

Specification:

You are going to design an event-driven simulation that models the treatment of patients in a hospital emergency room. The number of examination rooms in use will be entered by the user. When a patient arrives, the nurse checks its temperature and blood pressure and records these along with the name, age, i.d. and the time of arrival of the patient. Based on the examination she assigns the patient emergency rating. If there is an open treatment room, the patient is treated immediately. If there is no treatment room available, patients wait for an open room. Patients are treated according to their urgency rating; patients with the same urgency rating are treated in order of arrival. Once treatment begins, the examination room is occupied until treatment is finished, even if a more urgent patient arrives in the meanwhile

All the relevant data about a patient:

1. Time of arrival
2. The urgency of patient's condition
3. How long the treatment will take

should be simulated by your program (that means that **you need to design Patient Generator** that will produce patients and simulate the above parameters for them)

Events:

This program will be an event-driven simulation and includes two different kinds of events:

- An arrival event occurs when a patient arrives. When one arrival occurs, another arrival event needs to be scheduled. An arrival event is an external event.
- A departure event occurs when an examination room becomes available because a patient's treatment is finished. If there are patients waiting, the next scheduled patient is moved into the examination room.

The simulation is to run for enough time to come up with reliable conclusion. We suggest starting with 1-minute simulation and once your program run smoothly you can try to extend it

(up to 8 real hours). Please document the actual time you run your simulation. Let's call this time MAX_TIME, after MAX_TIME no new arrival events should be generated. The patients that are still being treated and any that are waiting after the MAX_TIME deadline finish normally.

Implementation:

- You will need an event list for this assignment, which will be modeled by priority queue. **For this assignment, you are asked to write your own implementation for** Priority Queue ADT**.** A priority queue behaves like a queue, except that objects are not always added at the rear of the queue. Instead, objects are added according to their priority. If two objects are equal, they are handled first-in, first-out. Removal is always from the front. **Note that the main characteristic of Priority Queue is that you can get the highest priority element in O(1). This is also a requirement for your implementation.**
- Implement Priority Queue ADT using generic data structure of your choice. Try to come up with the most efficient implementation for all the methods discussed below. **Be prepared to discuss the complexity of implementation of all the methods.**
- **The type of the stored data should be generic type that is comparable**. (Hint: your class declaration should involve: `public class MyPriorityQueue<E extends Comparable<E>>`). Note: you may find it beneficial to implement Priority Queue with Integer elements, test it and only afterwards extend it to be generic.
- **The concrete class that implements the ADT below must be called `MyPriorityQueue`**

Priority Queue ADT is as follows:

  - constructor - create an empty Priority Queue.
  - `void insert (E e)` - insert the object in the queue. Use the Comparable method `compareTo( )` to implement the ordering.
  - `E remove()` - removes and returns the object from the front. Throw an appropriate exception if the Priority Queue is empty.
  - `E front()` - returns the object at the front without changing the Priority Queue. Throw an appropriate exception if the Priority Queue is empty.
  - `boolean isEmpty()`
  - `List testForwardTraversal()` and `List testReverseTraversal()` - test methods to be used by the unit tests; used to make sure all of the links are correct (going forward and backward) by traversing the queue and constructing a list containing its contents.

In your simulation, you will use one queue (= arrival queue) for the patients arriving in the ER. The patients are queued by order of urgency and by arrival time for two patients with the same urgency. If an examination room is available, a patient is dequeued from the arrival queue and transferred to the examination room. Simulate the patients being currently examined with another queue (= examination queue). Patients in the examination queue are ordered by their departure time (= time of start of examination + duration of examination). As patients are "routed" to an examination room, your code should decide which non-busy room gets the patient so that all rooms are equally busy. For example, if there were 2 rooms, you would not want one

to be 75% busy and the other only 15% busy (each room has its own medical personal). Have your code spread the visits between the 2 so both work out to be equally busy (not the same, but close). You can do so with another priority queue (= room queue). Order the available rooms in the room queue by their "busy" time. The least busy room should be at the front of the queue and is the next one that will receive a patient. Create classes as needed to model the different elements going the queues. Remember that any data put in a priority queue must be Comparable.

The rest of the design up to you (user provided input is described below). Come up with an appropriate object-oriented design based on examples/ideas from the class. Test any implementation you provide (using both white and black box tests).

Simulation:

Write a class `ERSimulator` that will run your simulation. The user input will be:

1. Number of rooms
2. Type of simulation (random and preset for testing).

Output the results (see below) in a nice readable format. Run the simulation by filling the arrival queue with new patients as given by your event generator class. Dequeue patients from the arrival queue and route them to the examination queue. The simulation is over when the duration of the simulation (= 8 hours) has passed and the examination queue is empty. Tabulate your results as the simulation runs and report your results at the end of the simulation.

Results:

As the simulation runs, collect data to answer these questions. Display the answers to the user in an easily readable form:

1. How many examination rooms are in the system?
2. For how long did the simulation run (in hours)? Note: this will be longer than 8 hours since some patients might still be being treated.
3. How many patients were treated in all?
4. What was the average wait for treatment (in minutes). Note that sometimes the wait time is zero (if the patient is moved into an examination room without being placed in the patient queue). In addition to the average overall wait, provide two other values:
    - the average wait for patients with urgency levels from 1 to 4 (highest priority).
    - the average wait for patients with urgency levels 9 or 10 (lowest priority).
5. What was the average duration of treatment (in minutes)?
6. How many patients were treated in each examination room?
7. What percentage of time was each examination room busy?

Suggestions:

As always, you will have success if you code and test in pieces. Get your individual classes working first. Then put them together in the simulation.

When testing, you may want to start with simple numbers instead of the random numbers. Remember that the objective of testing is to prove that the program is working correctly. This is easier to do with simple data.

You could run the simulation for a shorter period during testing so that you do not have to examine a huge amount of data. Though not a requirement, you might think about adding a second input field to enter the number of hours to run the simulation. This might aid in testing. **Anyway, NEVER hardcode any magic number in your code.**

Documentation, Style & Testing:

1. Make sure you have documented your public interfaces well. Remember, you are building these classes from scratch. No one has any idea of what they do except you. You need to communicate these ideas to others.
2. Use appropriate style that was discussed in class. This includes (but not limited to) named constants, private methods, and throwing exceptions when appropriate.
3. Develop Junit tests for your implementation of Priority Queue ADT. The test code coverage should comply to our course standards.
4. Follow the EXACT naming of Priority Queue methods that is provided above – this is important for tests that we/TAs might run on your implementation.
5. Carefully test your code and include information about the testing you performed in your report.
6. Design flexible and maintainable code. Use Keep It Simple Principle

Written Report (typed and turned in as a pdf file):

You must turn in a short report that discusses your program, describes the class design, and discusses issues you encountered while working on it. This report will help you during your codewalk. Your report should cover the following:

1. Planning: How did you plan and organize your program? What does it do? Include user instructions if appropriate.
2. Implementation: How is your program organized? What are the major classes? How do the objects interact? Remember to clarify your design, since we know nothing about your code. Explain this well.
3. Testing: How did you test your code? What sort of bugs did you encounter? Are there any unresolved problems in the code? Be sure to list any ways in which the program falls short of the specification.
4. Evaluate this project. What did you learn from it? Was it worth the effort? This could include things you learned about specifications and interfaces, design issues, Java language issues, debugging, etc.
5. Finally, based on the testing you have performed, what do you think is the optimal number of examination rooms? Make sure to back up your recommendation by referring to your results, including average waiting times.

GOOD LUCK