CS5010, Fall 2019

Lab 4: ADTs

Summary

Lab goals:

- To understand the behavior of stacks and queues
- To practice writing comprehensive unit tests.
- To implement priority queue operations.
- To define a data structure using Java generics.

Complete this assignment with your homework teammates, using your group GitHub repo. Use this as an opportunity to practice using GitHub to manage group work. Some suggestions:

- Make sure each person has the latest code from master: run `git pull` while on the master branch.
- Each person should then create their own branch.
- Divide up the work so that each group member is working on a different piece of the problem.
- Don't touch code that other people are editing. This is key to avoiding merge conflicts.
- When you're done, create pull requests to merge everyone's changes to master.

Problem 1: Testing implementations of ADTs

For this problem you will write a collection of JUnit tests for implementations of two ADTs: a generic immutable Stack and a generic immutable Queue. Your tests will 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 to 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:

- `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.
- You can assume all implementations of the ADTs have a constructor.
- Do not worry about tests on undefined cases that would yield exceptions e.g. like trying to remove an element from an empty data structure.
- You are not being asked to implement these ADTs, only to define test cases. Sample interfaces are available for your reference in Code_From_Lectures > Lab5.

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:

```
Stack push(E element); // Adds an element to the Stack

Stack pop(); // Removes the most recently added element

E top(); // Returns but does not remove the most recent element.
```

Example:

- Assume that elements 7, 4, and 5 are pushed to a new `Stack<Integer>` in that order.
- Calling `pop` would produce a stack containing 4 and 7.
- Calling `top` on that stack would return 4.
- Calling `pop` on that stack would produce a stack containing 7.

You can see an interface for the Stack ADT in the Code_From_Lectures repo > Lab5.

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:

```
Queue enqueue(E element); // Adds an element to the Queue

Queue dequeue(); // Removes the least recently added element

E front(); // Returns but does not remove the least recent element.
```

Example:

- Assume that elements 7, 4, and 5 are enqueued to a new `Queue<Integer>` in that order.
- Calling `dequeue` would produce a queue containing 4 and 5.
- Calling `front` on that queue would return 4.
- Calling `dequeue` on that queue would produce a queue containing 5.

You can see an interface for the Queue ADT in the Code_From_Lectures repo > Lab5.

Problem 2: Priority Queue ADT

A priority queue is a data structure that 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.

Implement Priority Queue ADT using the generic data structure of your choice. Try to come up with the most efficient implementation for all the methods discussed below. Note that the main characteristic of a priority queue is that you can get the highest priority element in O(1). The type of the stored data should be generic type that is comparable. Hint: your class declaration should involve something like: `public class MyPriorityQueue<E extends Comparable<E>>`. Another hint: you may find it easiest to implement Priority Queue with Integer elements first, test it, and only afterwards make it generic.

The Priority Queue ADT (mutable):

- Constructor – create an empty Priority Queue.
- `void insert(E e)` – insert the object into the queue. Use the Comparable method compareTo() to implement the ordering.
- `E remove()` – removes and returns the object at 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 and appropriate exception if the Priority Queue is empty.
- `boolean isEmpty()` – returns true if the queue is empty

Some suggestions for how to divide the work up:

- Start by working together to create an interface for the ADT and a dummy implementation i.e. a concrete class with the interface methods generated by IntelliJ. Merge all changes into the master branch.
- Next, each person should pull the latest code from master and create their own branch. One person (or two working together if you're in a group of three) can tackle the implementation while the other can tackle writing unit tests.

Problem 3: Emergency room triage system

For this problem, you will use the Priority Queue you implemented for problem 2 to create a system that could be used to organize treatment of patients coming in to a hospital emergency room.

When patients arrive at the emergency room, they need to be queued for treatment based on the urgency of their condition, their arrival time, and the amount of time their treatment is anticipated to take. You will need to create a `Patient` class that implements interface `Comparable` in order to decide how to order patients in the treatment queue.

Patients will be removed from the treatment queue when their treatment begins. A patient's treatment will start as soon as a room becomes available for that patient. You will need to design and implement a room manager to keep track of the treatment rooms in the hospital and their status (occupied or available). Once a patient has taken a room, it should remain occupied until their treatment is complete (based on the anticipated treatment time for that patient).