Assignment 1
COMP 250, Winter 2022

Prepared by Prof. Michael Langer and T.A.s
Posted: Fri. Jan. 28, 2022
Due: Fri. Feb. 11 at 23:59 (midnight)

[document last modified: February 5, 2022]

General instructions

- 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 a grade of about 50/100. (We sometimes modify the exposed tests and grader after the assignment has been released, so the exact total number of points might change slightly over time.) These tests correspond exactly to the exposed tests on Ed. We will use private tests for the remaining points out of 100.

  We strongly encourage you to come up with more creative test cases, and to share these test cases on the discussion board.

  Ed should be used for code submission only. You should write, compile and test your code in your IDE (Eclipse or IntelliJ).
- **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

In this assignment we will model a real life situation that you are familiar with: registering for courses at university! We will model a set of courses and a set of students who register for the courses. We will also model details such as a course waitlist. (Note we will use the term "course' rather than "class" so that the latter term can be reserved for talking about the Java classes.)

There are two main learning goals in this assignment. The first is that you get some experience with arrays and linked lists. Your task will be to implement a Java class that uses an array of singly linked lists. (A similar class will come up again later the course when we cover hashmaps.) The second learning goal is that you get some experience working with multiple classes at a time, and in general that you get comfortable with object oriented programming in Java.

Here are the classes you will use:

SLinkedList

A full implemented linked list class is given to you. You must use this class instead of the Java **LinkedList** class.

Student

A fully implemented **Student** class is given to you. You must not modify it. (If you submit a **Student.java** file, then that file will be ignored and the original will be used instead.)

This class has three fields:

- **id** – a positive int
- **name** – a String
- **courseCodes** – a linked list of course codes (Strings) of the courses that the student is registered in or on the waitlist for; The maximum number of courses in this list is three, and note that that a student might be on three waitlists but registered for no courses, and the student has to wait.

It also contains the following methods:

- a constructor **Student(int id, String name)**.
- **isRegisteredOrWaitlisted(String course)** – returns true if the student is either registered for the course or on the waitlist, and false otherwise.
- **addCourse(String course)** – adds the name of the course to **courseCodes**. The method doesn't return anything (void).
- **dropCourse(String course)** – removes the course from the student's **courseCodes**. list. The method doesn't return anything (void).

Note: The official course registration information will be stored in the **Course** class (coming up next). It will be your task there to ensure that the **courseCodes** attributes of the different **Student** objects are correct. With an incorrect implementation, it could happen that a **Student** has, say, "COMP250" in its **courses**, but the **Course** object corresponding to "COMP250" has no record of this student (which would of course be an error).

Course

You will write a class **Course** that keeps track of the students taking a course. The Course class has the following fields:

- **code** - a string that uniquely encodes the course; examples will be given in the exposed tests.
- **capacity** – a positive integer; this is the current maximum number of students that can register for a course; in particular, for a Student to register for a course, it is necessary that the number of students currently registered for that Course is strictly less than the capacity. (There are additional necessary conditions which will be described later.)
- **studentTable** – an array of **SLinkedList<Student>**, which we refer to informally as the 'table'. Each array slot in the table will store a linked list (possibly null or possibly empty) of Student objects.[1] The order of the students within each linked list is not important.

  The length of the array is defined to be equal to the capacity of the course. The length of the array (and hence course capacity) is initialized by the Course constructor – see below.

  Note that in a real situation, the registration capacity ("the cap") would be the number of chairs in the classroom. Do not confuse the chairs in a real classroom with the slots in the array of our model. In particular, we are *not* requiring that there is one student per array slot. Indeed it can easily happen that there are multiple students in some slots (list size greater than 1), and zero students in other slots. This is why we have a linked list of students at each array slot, rather than just a single student.
- **size** – the number of students currently registered for a course; this does not include students on the waitlist (below)
- **waitlist**: a **SLinkedList<Student>** object.

  The waitlist has a queue behavior, namely a student enters at the end of the queue (tail of the linked list) and is removed at the front (head of the linked list); so position matters. We will cover queues in detail in an upcoming lecture, but the basic idea just described is simple enough for now.

  The waitlist has a maximum size, which is 0.5 times the capacity of the course (and rounded down, since size must be an integer).[2] The mechanism for enforcing this maximum is described in the put() method later.

[1]As we will see in later lectures, a similar 'array of linked lists' data structure is part of a more elaborate data structure called a hash map, more specifically, a hash set. Getting experience with this simplified version now will make hash maps/sets easier to understand later.

[2]Casting to an int will round it down.

[updated Jan. 30: See also update in put().] The course capacity and waitlist behave in a way that is similar to how it works in CS at McGill: when a course is full and then some registered student drops the course, the student at the front of the waitlist can register; also, when demand for a course is larger than expected, the administration finds a new lecture room with larger capacity, and this allows all students on the waitlist to register. Specifically, when a student is in a course waitlist, there are two ways the student can register for the course: (1) when another student who is registered for the course (not on the waitlist) withdraws from the course; in that case, the first student in the waitlist is automatically removed from the waitlist and is registered (added to the table).[3] (2) when the waitlist is full and another student wishes to register, a larger table is created and all students currently in the waitlist are moved to this larger table. (But note that this new arriving student sadly gets put on the now-empty weight list.) See the changeArrayLength() and put() methods below.

A few methods in the **Course** class are given to you:

- constructor **Course**(String code) – the default list length of the array is 10; each array slot is initialized to **null**.

  Please see the comments in the starter code. Java can be a bit finicky about the array of linked lists of student type, namely SLinkedLinked<Student> []. So when you create linked lists elsewhere in your code, you may need to borrow the syntax from there; your IDE will help you.

- constructor **Course**(String code, int capacity) constructor – the capacity parameter defines the length of the array; each array slot is initialized to **null**.

- **toString()** – this allows you to print a Course object. For example, try the following code in a test file and see how the printout changes once you have added a few students:

```
Course comp250 = new Course("COMP250", 3);
System.out.println(comp250);
```

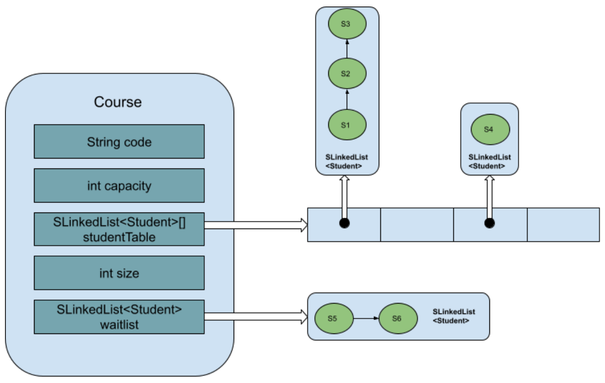

Figure 1: An example of the Course object.

[3]This is different from how it works at McGill, where a student on a waitlist is notified that there is an open slot and given a deadline to register, if they still want to.

Your Task

Implement the following methods of the **Course** class:

- **changeArrayLength**(int m) – void method;

  Note: This is a helper method for put(). We mention it first because you will need to be familiar with it in order to fully understand the more fundamental put() method, which will be explained next.

  This method creates a new student table for this course; this new table has the given length m; the method moves all registered students to this table, as described in more detail below. Note that this method itself does not move the waitlisted students. The waitlisted students are moved in a separate step in the put() method.

  This method creates a new array of SLinkedList<Student>, moves all the registered students into these linked lists, and changes this course's studentTable field so it references this new table instead. Each student in the current table will be moved to the new table. This will require in most cases that the student is moved to a different array slot index. The slot index for the SLinkedList<Student> that they are moved to is $id \ \% \ m$, where m is the new length of the array (the parameter of the method).

  The motivation for increasing the capacity of the array is that it keeps the average lengths of the linkedlists below 1, which tends to mean a fast traversal of each linked list. (This is a very important concept in hash tables and hash sets as we'll see in a few weeks.)

- **put**(Student s) – returns true if the method successfully registers the student or adds the student onto the waitlist; returns false otherwise, namely in one of the three following cases: (1) if the student was already registered in the course, or (2) the student was already on the waitlist, or (3) the student had already reached their personal maximum of three courses.

  Assuming none of these three conditions was met, the student can then be either registered for the course or put on the waitlist. If the course is not yet at capacity, then the student can be registered. In that case, the student's id is first used to determine where the student goes in the student table, namely in which linked list it goes: then, the student is put in the linked list in array slot $id \ \% \ m$, where m is the length of the array (the course capacity).

  [Updated: Feb. 5] Note that you should go directly to slot $id \ \% \ m$ rather than iterating through all the slots. Indeed this is the entire point of taking the mod!

  [updated Jan. 30:] If the course had already reached its registration capacity, then the student is added to the waitlist. If the waitlist was already at its full capacity, then the put() method increases the registration capacity of the course by 50%, as mentioned earlier – see waitlist field. (Increasing the capacity of the course by expanding the table is analogous to finding a bigger room). Use the changeArrayLength() method as a helper here. The put() method then registers all students currently on the waitlist, by moving them from the waitlist to the studentTable. (Note that the waitlist capacity is 50% of the course capacity, which was conveniently chosen such that all students in the waitlist can be registered.) Finally, the new student is added to the empty waitlist.

  For example, if the studentTable array length is 10 and the course is full (10 students registered) and the waitlist was already full (5 students on waitlist), the method would increase the table size

– that is, capacity – to 15 and make a new empty waitlist. The new student would then be added to this empty waitlist. Note that the maximum size of the new waitlist would be 7, since the course capacity has increased.

- **get**(int id) – returns a Student, assuming that the student is either registered for the course or on the waitlist. If there is no student in the table or waitlist that has this id, get() should return null.

  [Updated: Feb. 5] As with the put() method, you should go directly to slot $id \ \% \ m$ rather than iterating through all the slots. The same goes for remove() below.

- **remove**(int id) – removes the Student associated with this id; if the id is not found in the table or on the waitlist, then it should return null; otherwise, it should return the Student associated with the id.

  If the student that is removed was registered, then this student should be replaced by the student who is first in the waitlist queue. If the student who is removed was on the waitlist, then they should just be removed from the waitlist.

- **getCourseSize**() – returns the number of students registered in the course (not in the waitlist). Your code should maintain the public size variable that keeps track of the number of students registered. Note that a successful put() and remove() will only change the size field if the number of students that are registered changes.

- **getRegisteredIDs**() – returns an array of int[], namely registered student id's. The length of the array is the size (number of students) in the course.

- **getRegisteredStudents**() – returns an array of type Student[], namely the registered Students. The length of the array is the current size (number of students) of the course.

- **getWaitlistedIDs()** – returns an array of type int[], namely the ids of students in the waitlist.

- **getWaitlistedStudents**() – returns an array of Students in the waitlist.

  For each of these four 'get' methods, the order of the students/id's in the arrays does not matter. All that matters is that the set of students/ id's returned should be correct.

How to begin?

We suggest that you begin by implementing a basic version of the put(), get(), and remove methods so that you can register students for a course, and have students drop a course. Once you have a basic version of these method working, you can start pushing the limits: fill up a course, and start using the waitlist; register a student for multiple courses so that the student reaches their capacity; etc. Implement the five get methods and ensure that they are working. Finally, address the case where the waitlist is full and another student wishes to join, by implementing the changeArrayLength() method using it to generalize your put() method.

Submission

Please follow the instructions on Ed Lessons Assignment 1.

- Ed does not want you to have a package name. Therefore you should remove the package name before uploading to Ed.
- You should only submit the **Course.java** class. No imports are allowed.
- You may submit multiple times. Your assignment will graded using your most recent submission.
- You should not be editing/debugging your code in Ed, unless you are making only very small changes. Rather you should write your code using your IDE.
- If you submit code that does not compile, you will automatically receive a grade of 0 for that submission. Since we grade only your latest submission, you must ensure that your latest submission compiles!
- The deadline is midnight Fri. Feb. 11. On Ed, this deadline is coded as 12:00 AM on Saturday Feb. 12. Similarly, on Ed, the two day window for late submission ends at 12:00 AM on Monday Feb. 14.

  Note that Fri. Feb. 11 is also the date of Quiz 2. Therefore you would be better off getting started early on the assignment and finishing it in advance of the Quiz.

Good luck and happy coding!