La Salle College
420PA3AS – Advanced Object Oriented Programming

Michelle M. Khalife Fall

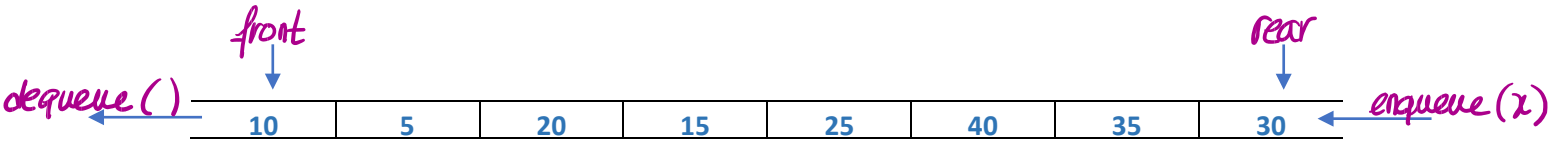
FINAL PROJECT

Part 1 - Implementing a Generic Circular Priority Queue using Arrays

A queue is an abstract data type in which data is inserted at the *rear* and removed from the *front.* The operations are respectively called *enqueue()* and *dequeue().*

The queue is characterized by a *front* variable that points to its first element and a *rear* variable that points to its last element.

The queue is initially *empty*: *front* and *rear* are set to -1. On an *enqueue*(), *rear* moves down. On a *dequeue()*, *front* moves down. When the following numbers are *enqueued* (10, 5, 20, 15, 25, 40, 35, and 30), the queue becomes *full*.



A. Use an array to implement a queue data structure with the following main methods. T stands for the type of data you have in your queue.

Methods	**Description**
`Queue()`	Constructor, creates an array of length 20
`Queue(List<E>)`	Constructor, creates a queue from a list of objects, such that the size of the queue is 2x the number of elements received
`bool isEmpty()`	Returns true if the queue is empty, false otherwise
`bool isFull()`	Returns true if the queue is full, false otherwise
`void enqueue(T)`	Inserts an element of type T at the rear of the queue
`T dequeue()`	Deletes the element at the front of the queue
`int getSize()`	Returns the number of elements in the queue
`void resize()`	Resizes the queue when it's full
`void displayAllElements()`	Prints out the elements in the queue starting with *front*

Instead of shifting the elements back to index 0 as front moves beyond the midpoint and rear converges to length-1, consider turning your queue into a circular queue.

B. Turn your queue into a generic class so it can accept a professor. A professor is characterized by the following attributes:

- a 4-digit id,
- a name (last name, first name),
- a seniority level (0.00 – 60.00), and
- a hiring date

C. To turn your queue into a priority queue, inherit from your main queue and override the enqueue() method.

A professor with higher seniority than another professor gets inserted ahead in the queue. Let your professor class implement the Comparable interface and override the compareTo() method. It two seniorities are equal, default to the date of hire; the more senior professor is the one who was hired first.

Then, add the following methods:

Methods	**Description**
`void enqueue(T)`	Enqueue an object at the right index based on its priority. *The priority is determined by the seniority. Move all objects accordingly.*
`void displayElement(int)`	Given an id, search and print the element
`void displayHigherPriorityElements(T)`	Display all elements higher than given prof's seniority
`void displayLowerPriorityElements (T)`	Display all elements lower than given prof's seniority

D. Test your classes in a main application

E. Generate JavaDoc for your queue classes.

F. Create a jar file from your class.

Part 2 - Implementing a Shallow Match Algorithm using Multiple Data Structures

La Salle College has many departments. A department may have multiple programs. For example, the Computer Science Department lists Programming, Networking, Gaming, Mobile, Business Intelligence, and Artificial Intelligence. The courses that belong to these programs fall under different disciplines.

BI1: CB1, CB2, CB3, CB4, CB5, CB6, CB7, CB8, CB9
I-1: 113, CV4, JV1, JV2,
I-2: AM1, AT5, CV5, J05, JP5, JV3, JV4, L04, …

A professor may teach any course in the department provided they have the matching discipline. Disciplines are added to the professor's profile, based on their academic background and professional experience.

Prior to the beginning of the semester, the department releases a list of available courses. There may be more than one section available per course.

Professors, on the other hand, need to submit their selections. First, they must specify how many total hours they would like to teach, per week. This is a positive number that cannot exceed 30hrs. Second, they need to list, in order of preference, the course(s) they wish to teach, and for each course, how many groups they would like to teach.

The algorithm needs to enqueue the professors in the seniority-based priority queue upon reading the *profs.txt* file. Then, with every dequeue(), the algorithm can retrieve the professor's requested hours and course selections from their *profId_select.txt* file. The algorithm checks the first selection, and cross references with the list of available courses. Available courses are read from an *courses_f22.txt* file (line format: <courseId, title, discipline, hours, numOfGroups>) into a hashmap(string, course) where the courseId is the key, and the course ref is the value. If a course is found and the number of groups is positive, then it can be assigned to the professor, provided the professor has the matching discipline. Decrement the number of groups in the map entry. Repeat until you reach the hours requested by the professor.

A 45hrs course is offered 3hrs a week, a 60hrs course 4hrs a week, a 75hrs course 5hrs, and a 90hrs course 6hrs.

If there is a match and the course is attributed to the teacher, add the courseId to the professor's listOfAffectedCourses.

Below is a simplistic representation of the environment that only considers currently offered courses.

Department: courseMap<courseId, Course>, listOfProfs<Professor>;
Course : id, title, discipline, numberOfHours, numOfGroups=0;
Professor : id, name, seniority, hiringDate, setOfDisciplines<String>, listOfAffectedCourses<Course>;

A. Class diagram – complete and/or edit the representation as you see fit and draw the class diagram.

B. Class implementations – implement the classes above (and any other that you need) paying particular attention to:

a. Department: create a partially parametrized constructor that accepts an ArrayList<Professor> listOfProfs and creates a new HashMap<String, Course> courseMap.
b. Course: make sure there's a copy constructor for your course.
c. Professor: when the professor object is constructed, the ArrayList<Course> listOfAffectedCourses is null.

C. Main application:

a. Open the *profs.txt* file, while !EOF, read each line and construct a professor object from the data available. Use a Set<String> myProfDisciplines = new HashSet<String>(); to which you can add the disciplines being read. Add the object to an array list *listOfProfs* as well as a priority queue *profProcessingQueue. Use a try catch block for opening the file. If the file is not available, throw an exception.*

b. Use a partially parameterized constructor to create a computer science department object. Pass *listOfProfs.*

c. Open the *courses_f22.txt* file, while !EOF, read each line and construct a course object from the data available. Then, add it to the department's *courseMap*.

You can use a simple Scanner object to read the file; instead of System.in, pass the file path.
Scanner sc = new Scanner(new File("C:\\Users\...\profs.txt"));

You can also use a try catch block for opening the file. If the file is not available, throw an exception.

D. Matching algorithm:

a. Start by examining the *profProcessingQueue*. When you receive the first element, use their id to look up their *profId_select.txt* file. Build the file path by using string concatenation. Open the file and read the first line: max requested hours. Subsequent line(s) contain(s) the selection.

b. Read the selection's *courseId*. Lookup *courseId* in *courseMap*. If there is no entry, the entry maps to null, or the *numOfGroups* = 0, then the course is no longer available. Otherwise, find the course's discipline.

   If the discipline is also contained in the professor's *setOfDisciplines*, then proceed to check the *numOfGroups* requested by the professor. The algorithm can assign as many groups to the profs as they requested, as long as they remain under their *maxWeeklyHours and maxWeeklyHours <= 30*.

   Assigning the course to the professor consists of using the copy constructor for the course in *courseMap*, changing the value in *numOfGroups* to reflect how many groups were assigned to the professor, and adding the newly created course to the *listOfAffectedCourses* in the professor object. For example, if the available course is PA3 with *numOfGroups* = 5, and the professor requested 3 groups, then *currentProfessor.listOfAffectedCourses.add(new Course (c));*

   Don't forget to decrement the *numOfCourses* from the *courseMap* entry.

c. Move to the next course selection. You are in 3 levels: the priority queue, the course selection, and finally the *courseMap*. When you've reached the limit, move to the next person in the queue, until the queue is empty.

You will need some helper functions/methods. For instance, you could implement a getWeeklyHours() for a course, such that given a total number of hours, the method returns the weekly hours.

When you're done, printout the affectations.