PG4200 Exam November 2021

The exam should **NOT** be done in group: each student has to do the exercises on his/her own. During the exam period, you are not allowed to discuss any part of this exam with any other student or person, not even the lecturer (i.e., do not ask questions or clarification on the exam). In case of ambiguities in these instructions, do your best effort to address them (and possibly explain your decisions in some code comments). Failure to comply to these rules will result in a failed exam and possible further disciplinary actions.

This exam for **PG4200** is composed of 5 exercises. You have **24** hours to complete these tasks.
This exam will have a grade in the *A-F* range. Each exercise will be scored between 0 and 10 points. In order to pass, you must have at least 25 points, and you must provide an answer to all the exercises (i.e. do not leave any exercise empty).

Each exercise must be in its own file (e.g., *.txt* and *.java*), with name specified in the exercise text. All these files need to be in a single folder, which then needs to be zipped in a *zip* file with name *pg4200_<id>.zip*, where you should replace <id> with the unique id you received with these instructions. If for any reason you did not get such id, use your own student id, e.g. *pg4200_701234.zip*. No "*rar*", no "*tar.gz*", etc. If you submit a *rar* or *a tar.gz* format (or any other format different from *zip*), then an examiner will mark your exam as failed without even opening such file.

You need to submit all source code (eg., *.java*), and no compiled code (*.class*) or libraries (*.jar*, *.war*).

Note: there is **NO** requirement about writing test cases. However, keep in mind that, if your implementations are wrong, you will fail the exam. As such, it is recommended to write test cases for your implementations, to make sure your code works correctly. However, if you do so, do **NOT** submit them as part of your delivery (as the writing of the test cases will not be evaluated).

Exercise 1:

File: Ex01.txt

a. Write a regular expression that matches archives names correctly for this exam. The name of the file must begin with "pg4200" and must end with "zip".

   For example, your regular expression

   should match: "pg4200_00000.zip", "pg4200_12345.zip"

   should **NOT** match: "pg4201_00000.zip", "pg4200_ab000.zip", "pg4200_00000.rar", "pg4200_00000zip"

b. Write a regular expression that matches sentences that are about the course (for example, a filter for mattermost that only matches sentences when they are about the course, and flags them for your attention).

   In this case, a sentence is a collection of characters that ends with a punctuation mark (full stop – '.' , exclamation mark – '!', or question mark – '?'). To be relevant, the sentence must contain one or more of the words: algorithms, algdat, pg4200.

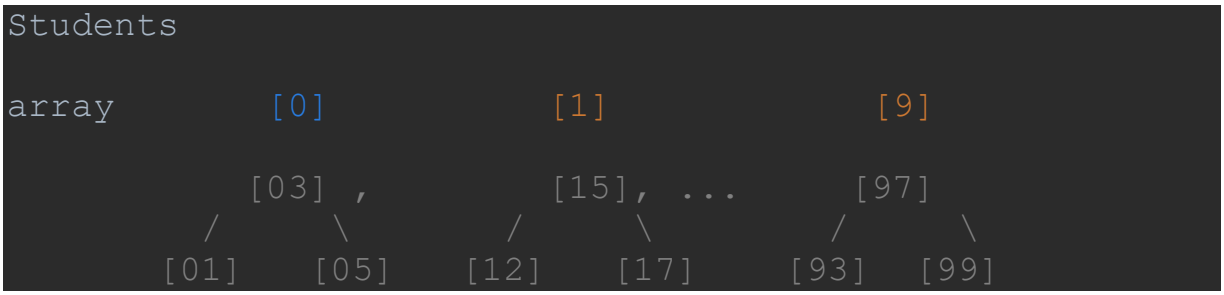
Exercise 2:

File: Ex02.java

Write a class called “StudentMap” that implements the interface *MyMap*, from *org.pg4200.les05.MyMap*. Using only basic data structures (arrays, trees), implement the functionality as follows:

- For each of the 10 digits, you will have a binary tree storing all the students whose student id starts with that digit. The ordering within the tree will be based on the order of the student ids.
- The binary trees will be accessed from an ArrayList of length 10 (where the index of each element of the array is the first digit of the student ids of the students in that tree).
- The operations defined in the *MyMap* interface (put, get, delete) should maintain this structure. This means that, when adding a new student to the map, the student should be added in the appropriate position in the correct binary tree.
- In addition, implement a method “Student getByName(String search_name)” that returns the first student found whose first or last name matches the search_name string.

See example below for a more visual understanding of the structure:



Exercise 3:

File: Ex03.txt

Compare your implementation of the *StudentMap* in Ex02 to an *ArrayList*, a *BinaryTree*, and a *HashMap*. Does your implementation always perform better than the others? If yes, explain why. If not, explain in which cases your implementation could perform worse. Consider for this comparison only the operations *put*, *get*, and *delete*, defined in the interface *MyMap*.

What is the performance of the method “getByName” compared to the operations defined in the *MyMap* interface?

Exercise 4:

File: Ex04.java

Tasks:

a. Create a pipeline that considers all the students that have taken a particular course (e.g. Algorithms and Data Structures), looks at the number of points they got in the exam, and computes the average grade for the entire course. Use the Student class below for the purpose of this exercise.

```
public class Student {
   public int student_id;
   public String name;
   // a map that takes a course id as a key and stores the grade
as a value, for each course this student has taken.
   Map<String, Double> examPoints;
}
```

b. Given a class representing a course at the university:

```
public class Course {
   String course_code;
   String topics;
   String evaluation;
   Map<Student, int> points;
}
```

For all the courses that are evaluated by means of a project or an exam, and whose topics include “programming”, arrange them in decreasing order of points they obtained in the “pg4200” exam, and return their names. Each student should only appear once, even if they passed several such courses.

Exercise 5:

File: Ex05.java

Write a compression algorithm for a set of simple chess moves.
Each chess move consists of
<number><piece><origin><destination><check> where:
<**number**> - the number of the move (i.e. 1 for the 1st, 2 for the 2nd, 3 for the 3rd, etc.)
<**piece**> - which of the pieces is being moved (pawn, rook, knight, bishop, queen, King)
<**origin**>, <**destination**> - designation of the respective square, in the form E2, H1, A8, etc. Squares are designated with a letter A-H and a number 1-8.
<**check**> - a move that results in a check will be marked with !

e.g.
1pe2e4 - 1. Pawn from e2 to e4.
2pe7e5 - 2. Pawn from e7 to e5.
1pc2c32pd7d63qd1a4!4pb7b5

1pf2f32pe7e53pg2g44qd8h4!

You **MUST** come up with a compression algorithm that is **customized** for this problem (i.e. do not use Huffman or LZW, but rather use *DnaCompressor* and the exercises in class as inspiration).

In your implementation, you can use the classes BitWriter and BitReader in package *org.pg4200.les11*;