Homework 2: Building a student search system

The scenario

You are a new, but prospective, young developer working for the IT department of Global University™, a large educational institution, with representative universities all over the world.

Due to the excellent study programs and the quality of teaching offered by Global University™ in its institutions, they have quickly amassed an enrolment of around a *million students* worldwide. Unfortunately, whereas the Universities excel in teaching, they lack proper information systems… For a long time, different branches and departments across universities used various Excel sheets and CSV documents to keep track of the students in their institutions.

In order to have the most relevant information about their students quickly and easily accessible, the management of Global University™ (finally) decided to modernize their systems and make an **easily searchable database** of all students across every institution. Each University exported their own students into a *CSV-formatted file* and sent them to the central server to be merged into a single file.

However, due to certain technical issues, the resulting CSV dataset mixed up all the student rows, creating a randomized file of *1,000,000 (1 million) students*. The dataset is a semicolon-separated (;) file with the following properties recorded for each student (in this order):

- Global University™ student ID (global ID for all students)
- student's name and surname
- date of birth
- university name
- department code
- full department name
- year of enrolment

The CSV file (available here) contains records for 1,000,000 (one million) enrolled students, with some example data looking like this (notice the ; separators between values):

global_student_list_sf.csv

```
2063092;Callum Campbell;20-05-1993;Global University of Great Britain;IR;International Relations;2018
1064989;Woodrow Schneider;17-09-1997;Global University of United States;ELL;English Language and Literature;2016
3059703;Justus Wuckert;12-11-1995;Global University of Australia;GLL;German Language and Literature;2018
2082031;Eden Walker;24-11-1995;Global University of Great Britain;ELL;English Language and Literature;2016
9065345;Viktória Ringlóciová;30-07-1993;Global University of Slovakia;CE;Civil Engineering;2020
7090334;Nikola Adamić;22-02-1992;Global University of Croatia;AM;Applied Mathematics;2016
1053337;Hershel Breitenberg;26-05-1997;Global University of United States;BUS;Business;2020
4048962;Jörn Wieland;16-10-1992;Global University of Germany;GBE;Genetics and Bioengineering;2016
11055905;Фаина Воронцов;01-06-1997;Global University of Russia;GLL;German Language and Literature;2017
7060127;Josipa Abramović;24-12-1994;Global University of Croatia;IT;Information Technologies;2017
```

1053337;Hershel Breitenberg;26-05-1997;Global University of United States;BUS;Business;2020
7090334;Nikola Adamić;22-02-1992;Global University of Croatia;AM;Applied Mathematics;2016

10049475;Aron Ek;12-01-1995;Global University of Sweden;ARC;Architecture;2020
303376;Emie Dare;01-05-1997;Global University of Australia;MAN;Management;2017
6032620;Sergio Cortés;04-07-1992;Global University of Spain;CS;Computer Science;2017
…

This is where you step in.

Task 1: Create a search system using a sorted dataset + binary search

It is now up to you to take this large CSV file and make it into a *searchable system*. Brushing up on your Data Structures and Algorithms knowledge, the first idea that comes to mind is **binary search**. But, to apply binary search, you need to first have a *sorted collection* of data.

So, you get to work.

Part 1: Implement the utils & sort logic

After examining the sorting algorithm performance, you decide on **merge sort**. To be able to sort the file, you need to do a few prerequisite steps.

For one, you need to create a **Student.java** class that will hold the relevant information for each student record. The Student class should contain the student ID, full student name, date of birth, university name, department code, department name and the student's year of enrolment. Moreover, the Student class should be **Comparable**, so you can use it in a sorting algorithm later.

Next up, you need a way to work with the input file, as well as a way to save the sorted result. In the **FileUtils.java** file you will need to implement:

- **public static Student[] readFile(String filePath)** → given a file path, *read all lines* from the file and return an array containing the **Student objects**.
- **public static void writeToFile(Student[] students, String filePath)** → given an array of *Student objects* and a *file path*, create a **new CSV file** containing the given data.

Afterwards, you need the *sorting algorithm* itself. The students need to be sorted according to their **student ID**. In the **MergeSort.java** class, implement:

- **public static void sort(Student[] students)** → this method is supposed to receive an array of Student objects, and sort it using the merge sort algorithm
    - The "student ID" property should be used as the basis of sorting.
- **any additional methods** which are required for merge sort to work.

Part 2: Implement the search logic

With the sort logic (and helper methods) in place, you can now move onto the **searching**. The search needs to be done using the **student ID** property.

In the **BinarySearch.java** file, implement:

- **public static int search(Student[] students, int key)** → given an array of Students, this method should return the **index (position)** of the student in the array *if they exist*, or -**1** if *they do not* exist.
    - The “student ID” property should be used as the basis of search.
- A public static property **numSteps** that will keep track of *how many iterations* it takes to find a given student ID.

Part 3: Putting it all together

With all the required classes in place, you can get to work. Implement the *search system* inside the **main()** method of **StudentSearchV1.java**.

It is left up to you how to design the “UI / UX” of the system. You are also free to add any additional helper methods if you need them. The main logic of the system should be as follows:

- When the application is started:
    - The *unsorted* file is loaded into an array
    - It is then *sorted* using merge sort.
    - The *sorted array* is saved into a *new CSV file* of the *same format* as the original..
- The user is then asked to *type in a student ID* they want to find, *or* -1 if they want to close the program.
- If the user types in a student ID, the application will *run binary search* on the sorted student array.
    - If the student is **found**, the application needs to *print out* all available details about the student in a *nicely formatted way*, along with the *number of steps* it took to locate the student.
    - If the student is **not found**, the application should print out an error message, and the *number of steps* it took to determine that the student does not exist in the array.
- When a search is done, the user is prompted to enter a student ID again. If they enter it, the search process should repeat. If they enter -1, the application should terminate.

Here is an example of how the interaction could look like:

```
/home/aldin-sxr/.jdks/corretto-18.0.2/bin/java -javaagent:/home/aldin-sxr/Documents/idea
Loading the students...
Sorting the student array...
Saving the sorted file...
============================
System is ready.

Enter the ID of the student you want to retrieve, or -1 to exit: 500
Student ID: 500
Name and surname: Geneviève Fabre
Date of birth: 18-02-1997
University: Global University of France
Department code: CS
Department: Computer Science
Year of enrolment: 2018

The student was retrieved in 18 steps.
```

```
Enter the ID of the student you want to retrieve, or -1 to exit: 9
The student with the requested ID does not exist.
The search was completed in 19 steps.
```

```
Enter the ID of the student you want to retrieve, or -1 to exit: -1
Thank you for using the student search system.
```

Some **expected values** you can use for testing:

- ID = 5076116 → Suzanne Gillet; 1 step
- ID = 101 → Hillard Herzog; 20 steps
- ID = 4010301 → Rosmarie Konrad; 16 steps
- ID = 2069217 → Dean Anderson; 2 steps
- ID = 9044867 → Lenka Mokroš; 12 steps

(Optional) Part 4: Enabling additional sorting criteria

While it is not required at this stage, you realize that the University higher-ups might, at a certain point in future, start requesting the student data sorted by names, departments, birth dates, etc. To prepare for that you decide to implement **Comparator** classes for the remaining properties.

To make this work, you should:

- Implement **Comparator** classes for all remaining Student properties.
- **Modify** the **sort()** method to *accept a Comparator* as the second argument.
    - **public static void sort(Student[] students, Comparator<Student> comparator)**

- If a Comparator is passed into the sort() method, sorting should be done using that Comparator's property, **not** the student ID.

Task 2: Create a search system using red-black trees

While the solution with binary search works and the management is happy, you start thinking - this student database often gets updated; is it not a bit "wasteful" to have to re-sort the file every time that happens, make a new file, and load it into the system? It would be a lot better if you could use a *data structure* that is *already designed* for *fast search and retrieval operations*.

After some thinking, it hits you: a *binary search tree*, or better yet, a **red-black tree**. You get to work once again.

Part 1: Implement a red-black tree

Since you already have **Student.java** from the previous student search implementation, you can *re-use it*. However, you do need to implement a **Node.java** and **RedBlackTree.java** class.

The **Node.java** class should have references to the *key* - student ID and the *value* - the Student object. Moreover, it should contain references to the *left* and *right* children, as well as the *color* of the node.

In the **RedBlackTree.java** class, you should implement:

- **public Value get(Key key)** → given a key (student ID), return the entire Student object
  - If the student with a given key **does not exist**, return *null*.
- **public void put(Key key, Value value)** → given a key (student ID) and a value (Student object), *insert* the student into the proper position in the tree
- **any additional methods and properties** which are required for a red-black tree to work
- a property **numSteps** that will keep track of *how many steps* it takes to find a given student ID
- a method **countRedLinks()** that will count *how many red links* there are in the tree.

Part 2: Implement a file reader

With the red-black tree completed, you now need a way to *read* the students *into* a red-black tree. To do that, inside the **FileUtils.java** class you implement the following:

- **public static RedBlackTree<Integer, Student> readFile(String filePath)** → given a file path, *read all lines* from the file and return a **red-black tree** containing all **Student objects**.

Part 3: Putting it all together (again)

With all the required classes in place, you can get to work. Implement the *new search system* inside the **main()** method of **StudentSearchV2.java**.

It is left up to you how to design the “UI / UX” of the system. You are also free to add any additional helper methods if you need them. The main logic of the system should be as follows:

- When the application is started, the *unsorted* file is loaded into a *red-black tree*.
  - The application should print out *how many red links* exist in the tree.
- The user is then asked to *type in a student ID* they want to find, *or* -1 if they want to close the program.
  - If the student is **found** in the tree, the application needs to *print out* all available details about the student in a *nicely formatted way*, along with the *number of steps* it took to locate the student.
  - If the student is **not found** in the tree, the application should print out an error message, and the *number of steps* it took to determine that the student does not exist in the array.
- When a search is done, the user is prompted to enter a student ID again. If they enter it, the search process should repeat. If they enter -1, the application should terminate.

Here is an example of how the interaction could look like (very similar to the first implementation):

```
/home/aldin-sxr/.jdks/corretto-18.0.2/bin/java -javaagent:/home/aldin-sxr/Documents/idea
Loading the students into the tree...
The tree was built with 253952 red links.
========================================
System is ready.

Enter the ID of the student you want to retrieve, or -1 to exit: 500
Student ID: 500
Name and surname: Geneviève Fabre
Date of birth: 18-02-1997
University: Global University of France
Department code: CS
Department: Computer Science
Year of enrolment: 2018

The student was retrieved in 19 steps.
```

```
Enter the ID of the student you want to retrieve, or -1 to exit: 9
The student with the requested ID does not exist.
The search was completed in 22 steps.
```

```
Enter the ID of the student you want to retrieve, or -1 to exit: -1
Thank you for using the student search system.
```

Some **expected values** you can use for testing:

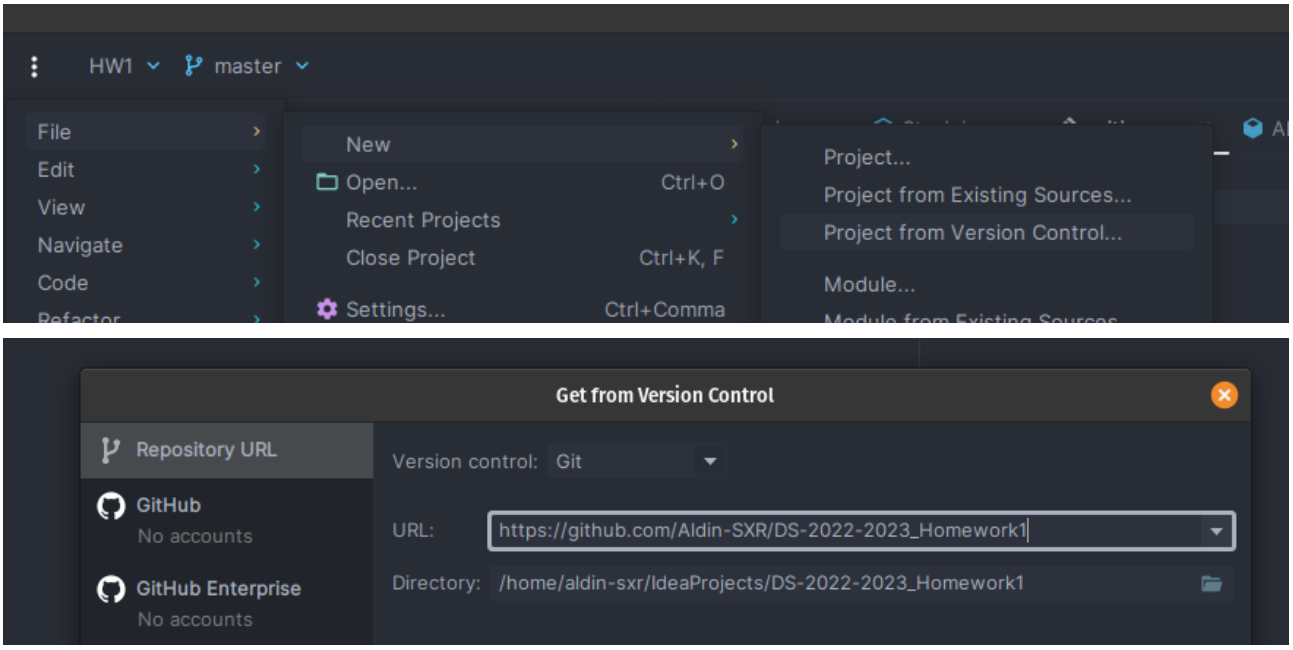
- ID = 5076116 → Suzanne Gillet; 15 steps

- ID = 101 → Hillard Herzog; 21 steps
- ID = 4010301 → Rosmarie Konrad; 24 steps
- ID = 2069217 → Dean Anderson; 22 steps
- ID = 9044867 → Lenka Mokroš; 21 steps

Skeleton code

To help you get started, with the homework, you have the **homework skeleton** code: https://github.com/Aldin-SXR/DS-2022-2023_Homework2

To import this existing code, you can use Intellij IDEA's **File > New > Project From Version Control…**, and provide the URL.



Alternatively, you can download the code, and use File > New > Project From Existing Sources…, or just copy and paste all code (although not recommended 🙂).

You must not:

- remove any of the methods in the existing files, rename them or change their *signatures* (except the sort() method in Task 1, Part 4).

You should:

- implement the missing method bodies for the required functionalities, and make sure they return proper output (if any)
- implement any additional helper methods / variables / classes, if you need them for the solution.

NOTE: You do not need to upload the original and sorted files with your homework; just upload the homework code.

Deadline

The assignment will be open on LMS until **June 14th, 23:55**.

Please send the professor an email if you need any further explanations, help or hints.
Good luck.