High-scores

Introduction

Your sister is trying to build an application that keeps track of high-scores for games. And of course, you help her once in a while. This is one of those moments. Your sister is currently working on a high-score table. The frontend is working fine but she is struggling with the backend. One of the problems she is facing it that the current implementation works for up to roughly 100 to 1000 players but when real data is fed into the application the high-score tables grinds to a hold or is very slow at best. After looking at her code you soon discover what the problem is. The algorithm she uses to get the high-scores is very inefficient. The algorithm she come up with looks as follows.

1. Make a copy of the list of players
2. Find the player with the highest high-score
3. Add the high-score to a temporary high-score list
4. Remove the player from the copied list
5. If there are high-scores left in the copied list then continue at step 2.
6. If the list contains more high-scores then asked for, remove the unwanted high-scores from the end
7. Return the list containing the wanted high-scores.

Can you help your sister?

It is your task to come up with a better, more efficient, implementation of the high-score list. After going over the material from Datastructures you decide to investigate three different approaches.

Approach 1: Sorting after adding to list

Your first approach is to sort the list with players after a new player has been added to the high-score list. There are two different variants you want to investigate.

Variant 1: Using insertion-sort or selection-sort

This variant looks a lot like the implementation your sister used, but you suspect it to be more efficient because you just sort the list and don't have to duplicate it and remove high-scores once they have been added to the list that is returned.

Variant 2: Using buckets-sort

This variant looks promising since from theory it should be faster then either insertion- or selection-sort. There might be room for some extra optimizations when using bucket-sort. One of the possible tricks could be to keep the buckets 'alive' once the list is sorted. When a new high-score is added to the list you only need to sort the bucket in which the new high-score is added. It is up to you if you want to implement this kind of optimizations or not.

Approach 2: Use a priority queue

According to your Datastructres teacher this is also very promising if there are a lot of high-scores to process. After I quick look at the code you are not sure that it will work in this case. Nonetheless you are giving it a try.

Start-project

Your sister provided you with a class and interface that you should use in order to make the integration of your code into her codebase flawlessly. The code can be found in the file names `Documents/Projects/high-score-student.zip` on the VLO. The project also contains some JUnit tests to get you started.

You have to implement the sorting and priority queue code by yourself. Supporting datastructures like java.util.ArrayList can be used if you find it necessary.

Because running all the required experiments takes quite some time you are encouraged to work in groups of two.

Assignment

- Implement three different versions, in three different classes, of the high-score list. One version uses either insertion- or selection-sort, another version uses bucket-sort and the last version uses a priority-queue.
- When you choose to add some sort of optimization to the second variant of sorting after adding, please explain the optimization as non-technical as possible. (e.g. don't translate the code into normal language but describe it using functional terms.)
- Expand the provided unit test with more unit tests and use these tests on all three version of the high-score list. You can use the setup method of the JUnit test class to switch between different implementations. This can be a manual process.
- Implement the finding of high-scores based on firstname, lastname, and the combination of firstname and lastname.
- Investigate how the efficient the different implementations are. Start with adding 100 (different!) high-scores to the high-score list and measure the time it took. Do this at least 10 times and use the mean of those 10 measurements. After that you double the number of high-scores and you take another 10 measurements. And so forth and so forth until running it takes more the 15 seconds to add the players to the list or you reach 10.000.000 players. (**Make sure you use the -Xint JVM option when measuring the time!**)
- Put the measurements in a table and determine what the efficiency is for each implementation, is this what you expected knowing the theory?
- Plot in a single graph the measurements of the three different versions.
- Which algorithm you will advise your sister to use and describe why?

Grading

The table below describes how many points you can get or lose if you have not or not properly implemented certain parts. Your initial score is a 7. The final score is determined by increasing or decreasing the starting score by the number of points in the table. If your final score is 6 or higher, you have passed the practical.
Remember that when evaluating your code we use a more extensive test set than that you have received in the project. It therefore is worthwhile that you add unique unit tests to the project yourself. If the tests you added overlap with this more extensive test set, then that is not a problem.
The points mentioned in this table are subject to change!

Code has compilation errors	-4,0
The provided unit tests fail	-4,0
Unit tests that ensure correct implementation of requirement are missing	-3,0
Approach 1, variant 1 is not implemented	-2,0
Approach 1, variant 2 is not implemented	-2,0
Approach 2 is not implemented	-2,0
Report is missing	-3,0
Report does not conform to requirements as specified in the study guide	-2,0
Sorting algorithms or priority queue not implemented by using own code	-1,0
For bucket-sort a optimization is implemented correctly and explained in the report	+1,0
For every approach and variant, the raw data is present as a appendix to the report	+0,5
For every approach and variant, the determination of efficiency (Big-O) is correct	+0,5
A single plot/graph contains for every approach and variant a curve	+0,5
The report is understandable for non-technical people	+1,0
The report is not in PDF format	-1,0

NOTE: Please note that while the assignment is in English you are only required to use English if your teacher asks you to or is English speaking by birth.