Finding players

Introduction

After you helped your sister with the high-score table, she once again needs your help. As you might have noticed the `HighScoreList` interface contained a method that can be used to search for high-scores based on the players first- and/or lastname. Finding high-scores based on first- and/or lastnames is a perfect opportunity to use a symbol table. There is however a little catch with symbol tables. The standard implementation for a symbol table only allows a single value to be stored under a certain key. In this case it is however necessary to store multiple values under the same key. After all there are more people called Nick or Sarah. Now it is your task to implement this special brand of symbol tables. And of course there are multiple solution that needs to be investigated…
You will be investigating the influence of three different collision resolution strategies.

Provided code

You don't have to build everything from scratch. Besides the `Player` and `HighScoreList` classes, there are 5 more classes and 3 test-classes. Out of these 5 classes there is one utility class `NameReader` and its test-class `NameReaderTest` that you don't have to worry about. They are there for your convenience. One other class named `HighScorePlayerFinder` should also not of your concern. This class acts as a single point of contact for the test-classes and the 3 classes that you have to implement.

Implementations

You have to implement three different collision resolution strategies. Namely linear probing, quadratic probing and double hashing. There are already three classes present for these strategies. The methods within those classes are empty, it is up to you to write code.
The names of the classes should speak for themselves. Make sure your somehow are able to detect collisions when you add players to a symbol table.
Be sure you understand what is meant by a collision. If you look at slide 23 of this week presentation, when adding license plate 05-M there are 3 collisions before it is added to the array!

Adding tests

If you want to add tests to the tests that already have been defined by `HighScorePlayerFinderTest`, there is a dedicated test-class available. This class `ExtendedPlayerFinderTest` does not have any methods or what so ever, feel free to add your own tests.

Report

Your will be conduction four experiments per collision resolution strategy. It is you task to investigate the influence of the size of the internal array on the amount of collisions that occur when adding 10.000 players to a symbol table. You should run the experiment with the following sizes 10.501, 11.701, 13.309 and 15.401.
You don't have to write any code to add the players to the symbol tables, there is already a method `collisionsShouldHappen` within the test-class

`HighScorePlayerFinderTest` that takes care of it. You do have to set the correct size before each experiment however! The provided code uses two external files containing first- and lastnames to generate random combinations.

Collision detection

Please state how you detect collisions in each of the resolution strategies, please include some code snippets.
TIP: You can use the old-school Java approach by using System.out.println() for printing the number of collisions after each experiment.
Pro-TIP: investigate how you can easily copy-paste comma or semi-comma separated values into your favorite spreadsheet application. This can be quite a time saver…..

Tables

For all experiments together create a single table that has the size of the internal array in rows and the number of collisions per strategy in columns.

Graphs

Your report should also include a single graph showing the relation between the size of the internal array and the number of collisions.
TIP: You can use Excel or any other spreadsheet application to create graphs based on your data.

Fair comparison

What do you think of the setup of this experiment? Are the different strategies tested in the same way or not? Was it a fair comparison? If so, explain why you think so. If not, explain what you think can be improved.
Which strategy would you use?

Deliverables

Please team up with a classmate and work on the assignment in groups of two.
You have to upload a single ZIP-file to Moodle that contains the following.

- Your report in PDF-format.
- The source files of your version of the project, preferably in the default Maven structure.
- The Maven pom.xml, especially if you made changes to it, also preferably in the default location for a Maven project.

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 **4,5**. 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 5,5 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 might be subject to change!

Report is not in PDF-format	-1,0
Provided tests fail	-2,0
Linear probing not implemented	-0,5
Quadratic probing not implemented	-0,5
Double Hashing not implemented	-0,5
Linear probing correctly implemented	0,5
Quadratic probing correctly implemented	0,5
Double Hashing correctly implemented	0,5
Extra test have been added and are meaningful	0,5
Collision counts presented in separate tables or table is missing	-0,5
Per strategy a separate graph or graph is missing	-0,5
Arguments and reasoning test setup make sense	0,5
Arguments or reasoning test setup missing	-0,5
Report complies with requirements as described in study manual	0,5