Assignment 3
Hashing

Release Date	Due Date
February 13, 2020	February 27, 2020

Objectives

- Experience working with Hash Tables and Hash Maps
- Understand hashing techniques
- Experience various techniques for hashing
- Practice developing high-performance solutions
- Compare theoretical vs empirical complexities
- Understand, at times, somewhat vague, requirements to develop a computing application
- Design and develop different solutions to a computing problem

Problem Specification

We will continue to work on searching in various different data structures and related problems. In this assignment, you will get experience in implementing these search algorithms and comparing their performances when hashing techniques are used to store data.

The dataset used will be again from Guild Wars 2 which is an MMORPG that allows you to equip a number of bags that give you inventory space. Each bag now stores 125 items. You will write a program to simulate this inventory system and search through it for various items using hashing techniques.

In this assignment, items will consist of only weapons. The text file included has data for ~750 different weapons in the game. There are 4 columns which list a weapon's “Item name”, “Min Strength”, “Max Strength”, and “Rarity”.

Write an application to solve the following problem:

1) Read in the “items” text file and store them in an array, called *itemsArray*.

   - Create a data structure to represent an item. This data structure should store the item's name, minimum strength, maximum strength, rarity, and current strength (current strength is not in the data set file. It will be assigned later).

2) Create *n* number of bags and populate them completely with items *randomly chosen* from the previous item list (duplicate items are allowed). Compute a pseudo-random number r in the range [0,itemsArray.length) and use itemsArray[r] to populate. Remember, a bag stores 125 items. When you place an item in a bag, randomly assign its "current strength" to a number between its minimum strength and maximum strength (inclusive).

   - For example, the Hatchet has a minimum strength of 134, and a maximum strength of 163. A random value between those two values (inclusive) should be assigned to the item, such as 145, 157, or 163.

- Fill your *n* bags sequentially. For example, the 1st item randomly chosen will go in the 1st bag, and the 2nd will also go into the 1st bag... up to the 125th item which will yet again go in the 1st bag. The 126th item will now go in the 2nd bag, and so on. The bags are meant to represent one contiguous inventory in sections of 125 items.

3) The 125 items in a bag will be stored using different hashing techniques, the size of the hash table for each bag will be 199. If the hash function is “ideal”, it will map each item in the bag to a different location, but we can’t guarantee that. So, we need to include a collision handling technique. You will practice implementing two types of such techniques, namely *open hashing* and *closed hashing*. Open hashing will use separate chaining, i.e., each slot in the hash table will be a linked list (why not take this opportunity to use your implementation of the linked list from HW2). Closed hashing will store all records directly in the hash table and use a probing technique to find an empty slot. Probing techniques you will implement are linear, pseudo-random, and double-hashing.

4) You get to decide three different hashing functions – take it as a competition among yourselves to design and implement good hashing functions for the application at hand.

5) Randomly choose an item from the original items array and then search for **all** occurrences of it in your simulated bag system. Both the name of the item and the rarity of the item must match. Report the item you are searching for. If found, report the bag numbers in which it was found, the hash table slots within the bag, and the item's current strengths. Record the time spent on searching.

- For example, if you were searching for a Rare Hatchet...
    - You might find it in the 12th slot of the 5th bag.
    - You might find a Fine Hatchet, but this would not count as a match, because the rarity does not match.

6) Test the speed of your search algorithm for each of the different hashing technique used. Repeat the process in step 5) multiple times, and report the average time searching. For this step, you do not have to print anything except the final average time that you calculated.

7) Your program should work with an arbitrary value for *n*. Generate sample output files for cases where *n* = 1, 10, 100, 1000, and 10000.

   Pause: If your code runs very fast (especially for small input sizes) then the elapsed time (using system supplied time functions) might be too small to be recorded. What can you do to remedy the situation?

Sample output

```
Here are my three hashing functions:
…

n=2
Bag 1 using Open Hashing with my Hashing Function1:
       (Fine Fireman's Axe, 180);  (Fine Skale Zapper, 188);
       Rare Lionguard Axe, 377
       Rare Flame Cleaver, 624
       Basic Pistol, 111
       ...
Bag 1 using Open Hashing with my Hashing Function2:
       Fine Fireman's Axe, 180;
       Fine Skale Zapper, 188
       Rare Lionguard Axe, 377
       Rare Flame Cleaver, 624
       Basic Pistol, 111
       ...
Bag 1 using Open Hashing with my Hashing Function3:
       (Fine Fireman's Axe, 180);  (Basic Pistol, 111);
       Rare Lionguard Axe, 377
       (Rare Flame Cleaver, 624); (Fine Skale Zapper, 188);
       ...
Bag 1 using Linear-Probing Closed Hashing with my Hashing Function1:
       Masterwork Bandit Revolver, 735
       Fine Champion's Mace, 163
       Rare Pirate Bloodletter, 397
       Fine Carving Knife, 191
       Fine Jeev's Dusty Focus, 239
       ...
Bag 1 using Pseudo-Random-Probing Closed Hashing with my Hashing Function1:
       Masterwork Bandit Revolver, 735
       Fine Champion's Mace, 163
       Rare Pirate Bloodletter, 397
       Fine Carving Knife, 191
       Fine Jeev's Dusty Focus, 239
       ...

Bag 1 using Double-Hashing-Probing Closed Hashing with my Hashing Function1:
       Masterwork Bandit Revolver, 735
       Fine Champion's Mace, 163
       Rare Pirate Bloodletter, 397
       Fine Carving Knife, 191
       Fine Jeev's Dusty Focus, 239
       ...
. . .
Bag 2: using Open Hashing with my Hashing Function1:
       Masterwork Bandit Revolver, 735
       (Fine Champion's Mace, 163);   (Fine Carving Knife, 191);
       (Rare Pirate Bloodletter, 397); (Fine Jeev's Dusty Focus, 239);
       ...

Searching for Rare Ebon Vanguard Hammer...
Found in (bag 1, slots 12, 24, 35. Strengths: 901, 899, 953), (bag 2, slots 12. Strengths: 901)
Average search time: yyyyy nanoseconds.
```

Output Requirements

- Report the value of *n*.
- If n=1, print the contents of bags as stored for each hashing technique and the search results.
- If n <= 10, Print out the first 5 slots of each bag's hash table. Otherwise, don't print out the contents of the bags.
- Step 6's output requirements (i.e., average time spent searching).

Design Requirements

Code Documentation

For this assignment, you must include documentation for your code. This includes how to compile and run your program. Also, you **must** give outputs of your program with *n* = 1, 10, 100, 1000, and 10000. If you don't provide outputs for these 5 inputs, it means your program does not execute properly.

Coding Conventions and Programming Standards

You must adhere to all conventions in the CS 3310 Java coding standard. This includes the use of white spaces for readability and the use of comments to explain the meaning of various methods and attributes. Be sure to follow the conventions for naming files, classes, variables, method parameters and methods. Read the material linked from our class web-pages (in case you can't recall programming styles and conventions from your CS1 and CS2 courses).

Testing

Make sure you test your application with several different values capturing different cases, to make sure it works.

Assignment Submission

- Generate a .zip file that contains all your files, including:
  - Plagiarism Declaration
  - Source code files
  - Including any input or output files
  - Documentation of your code – e.g. using Javadoc if using Java
  - A brief report (in a pdf file) on your observations of comparing theoretical vs empirically observed time complexities. Note this report will include (a) a brief description of problem statement(s), (b) algorithms descriptions (if these are standard, commonly known algorithms, then just mention their names along with customization to your specific solution(s), otherwise give the pseudo-code of your algorithms, (c) theoretically derived complexities of the algorithms used in your code, (d) table(s) of the observed time complexities, and (e) plots comparing theoretical vs. empirical along with your observations (e.g. do theoretical agree with your implementation, why? Why not?).

- Don't forget to follow the naming convention specified for submitting assignments