Assignment objectives

In this lab you will investigate the number of collisions and probes that occur while inserting items into a hash table using simple hash techniques. You will examine the effects of:

- Using different hash functions
- Using different sizes of hash table

Overview

You'll be writing a "hash simulator" that is given inputs (1) a list of keys (Strings, aka items to be hashed), and (2) an integer size of hash table to use for the simulation.

The Hash Simulator

Your simulator will fully (re)process the same input list three times – one for each of three built-in hash functions, and store some **results** in an array. For collision resolution your simulator should use *closed hashing with linear probing.* This technique is described in the lecture notes and on page 272 of the textbook.

Here's what your hash simulator will do when it is given a list of keys and a hash table size:

```
With H1 as the hash function:
    Create a new, empty hash table of the given size
        (hash table is also an array of Strings)
    Set collision count to 0
    Set probe count to 0
    For each item S in the input data
        Use H1 to find the hash value of S for the current table size
        Store the key in the hash table
            Count a collision if it occurs
            Count probes as needed
    end-for
    store the collision count and probe count in the results array
Repeat the above with H2 as the hash function
Repeat the above with H3 as the hash function
Return the results array
```

Coding requirements

Please be careful to spell the names of these items EXACTLY as shown here. If you do not, we will not be able to run your code.

Required class name: HashSimulator

Required public methods (4 in total below):

`public int[] runHashSimulation (String[], int)`

This method simulates inserting items into a hash table given a list of N input strings and a size to be used for the hash table.

- Param1 is an array of strings, the key values to be hashed
- Param2 is an int, the size of the hash table to be used

- Returns an array of 6 ints, the following results of the hash simulations:
  - results[0] is the # of *collisions* that occur when hashing with H1()
  - results[1] is the # of *probes* that occur when hashing with H1()
  - results[2] is the # of *collisions* that occur when hashing with H2()
  - results[3] is the # of *probes* that occur when hashing with H2()
  - results[4] is the # of *collisions* that occur when hashing with H3()
  - results[5] is the # of *probes* that occur when hashing with H3()
- No console output!

All three of the hash functions MUST use the names as shown here. The signatures for all three hash functions are the same.

```
public int H1(String, int)
public int H2(String, int)
public int H3(String, int)
```

- Param1 is a string, the key to be hashed
- Param2 is the table size currently being used
- Returns an int, the hash value of the key
- No console output!
- See below for definitions of the hash functions

Hash functions

You'll have three hash functions defined in your class. These should all be declared `public` so that the testing program can call them separately from the hash simulator.

A hash function (for our purposes in this lab) takes two arguments: 1) a string; 2) HTsize, the size of the hash table; and returns an integer in the range of 0 to HTsize-1 inclusive.

`int H1(name, HTsize)` – Let A=1, B=2, C=3, etc. Then the hash function H1 is the sum of these values for the letters in the string, mod HTsize. For example, if the string is BENNY, the sum of the letters is 2+5+14+14+25 = 60, and the returned hash value would be 60 mod HTsize.

`int H2(name, HTsize)` – For the i^{th} letter in the string (counting from 0), multiply the character value (A=1, B=2, C=3) times 26^i. Add up these values, and take the result mod HTsize. For BENNY the intermediate result would be 2*1 + 5*26 + 14*676 + 14*17576 + 25*456976 = 11680060, and the return value will be 11680060 mod HTsize.

WARNING 1: Probably some names in the data file will cause integer overflow if you use the int data type for these calculations!

WARNING 2: And don't use float, either.

`int H3(name, HTsize)` – *Invent your own hash function!* Pull one right out of your imagination, or Google around. NOTE: You MUST write comments in your Java code describing how your hash function works, and where you got it from.

Input files and sample output

There are three data files for you to use while testing:

- 37names.txt

- 798names.txt
- 5746names.txt

You should run hash simulations for a variety of hash table sizes. Pay attention to the effects of:

- Using the three different hash functions
- Using different sizes of hash table

Here are my results for running simulations with hash table sizes of N, 2N, 5N, 10N, and 100N:

File	HT size	H1 collisions	H1 probes	H2 collisions	H2 probes
37names.txt	37	25	189	13	103
	74	13	31	7	16
	185	12	29	2	3
	370	12	29	1	1
	3700	12	29	0	0
798names.txt	798	749	278282	395	11326
	1596	749	278271	196	454
	3990	749	278271	72	110
	7980	749	278271	41	49
	79800	749	278271	3	3
5746names.txt	5746	5689	16174085	4621	536301
	11492	5689	16174081	4222	95470
	28730	5689	16174081	2084	8303
	57460	5689	16174081	1876	6399
	574600	5689	16174081	267	304

NOTE: Your results for H3 will depend on your own hash function!

Other requirements and notes

COMP 3760 Coding Requirements

Did you write your name/ID in the code comments?
Did you follow all the other requirements?

main()

You will need to write a main program to test your HashSimulator class. Note: main() is solely for your own testing purposes. We will not be using or marking your main program. We recommend that you put your main() method in a separate driver class, so that you will be calling/running your HashSimulator class the same way that we do in testing. You're welcome to submit your main() program, but we probably won't use it.

Reading the data files

Your main will need to read a data file and create a plain Array of Strings (not an ArrayList!). We don't care how you do this.

Sourcing info for H3()

Your H3() function *must* include comments explaining your hash function and where you got it from.

Helpers

You are free to write and use any private members and helper functions that you wish to use.

Use arrays

Implement your hash table as a plain array of Strings in Java. *Do not use* any of the hash-related data types that are available in Java (HashMap, HashSet, etc.) for this part. We are implementing our own hash table here!

What are collisions? What are probes?

For more information, see the lecture notes about collision handling. The strategy you must use here is **closed hashing with linear probing**.

The hash function for an item tells you which bucket the item "wants to be in". A *collision* occurs if this bucket already has another item in it *right now*. At this moment, number of collisions++. It is important to note that this does *not* (necessarily) mean that there was another item with the same hash value. The item in this bucket could have been put there because of other collisions and probing.

So this one item you are processing right now will either go straight in, or it will have a collision. If there is a collision, then there will also be *probes*. Probes occur *only* after a collision. If your current item has a collision, you must find another place to put it. Count one probe for every *subsequent* bucket that you examine until you find an empty one; this count includes the empty one, but it does NOT count the bucket that had the initial collision.

The collision-count for each item you process can only be zero or one—either its designated bucket is available, or it is not. Therefore, the total number of *collisions* while hashing a total of

N keys can never be more than N-1. However, the number of probes can be MUCH larger than that.

NOTE: If you reach the end of the array without finding an empty bucket, “wrap around” and continue from the beginning (element 0). You can assume that your array will always be big enough for the number of data items you will need to store.

Submission information

Due date: As shown on Learning Hub.

Submit the following to the drop box on Learning Hub:

- Java source code (*.java file).
- Please *do not zip* or otherwise archive your code. Plain Java files only.
- Please *do not include* your entire project directory.

Marking information

This lab is worth 20 points.

Remember that a portion (likely 4-5 points) will be allocated to the COMP 3760 Coding Requirements.

Did you write your Name/ID/Set# in your code?

Virtual donut

🍩 *This part is not required; it is just for fun.* 🍩

Can you find a hash function (for H3) that gives a lower *collision count* than H2 for *all* the sample data files and for *all* the hash table sizes shown in the sample output in this handout? Many have tried to do this; few have succeeded!