Assignment objectives

In this lab you will investigate the effects of different hash table sizes and hash functions on the number of collisions and probes that occur while inserting items into a hash table using simple hash techniques.

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 process the input list fully three times – one for each of three different hash functions. 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 called with an array of Strings 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

Required class name: HashSimulator

Required public methods:

- runHashSimulation()—a method that simulates inserting N items a hash table given a list of N input strings and a size to use for the hash table
- H1(), H2(), and H3()—three different hash functions as described below

Method signatures:

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

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

```
int H1(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

```
int H2(String, int)
```

- (Same description as H1)

```
int H3(String key, int)
```

- (Same description as H1)

Hash functions

You'll have three hash functions defined in your class. These should all be declared `public` so that my 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 the values of 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: Some names in the data file will cause integer overflow if you use the int data type for intermediate calculations!**

`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.

Other requirements and notes

main()

You will need to write a main program with which you can test your HashSimulator class. Note: main() is solely for your own testing purposes. I will not be marking your main program. I

recommend that you put your main() method in a separate driver class, so that you will be creating/calling your HashSimulator class the same way that I am.

Reading the data files

Your main will need to read a data file and create an Array of Strings. I don't care how you do this. I'll gladly share my own, probably simple-minded, code for it if you want it.

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.

5. All other COMP 3760 Coding Requirements—of course!

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!

Tip: Always code and test incrementally. Start out by displaying results for ***just one hash function, just one hash table***. Make sure this works correctly before adding more complexity.

Input files and sample output

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

- 37names.txt
- 641names.txt
- 5575names.txt

Later I will provide sample results for the given input files, giving correct counts for the collisions and probes that occur with hash functions H1 and H2.

Your results for H3 will vary depending on your own hash function!

What are collisions? What are probes?

For more information, see the lecture notes about collision handling. The strategy we are using 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++. Note that this does *not* necessarily mean that there was a previous item with the same hash value, because 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 have a collision, or not. 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. There is one probe for every *extra* bucket

that you look at until you find an empty one to put the item. This includes the bucket in which you finally put the item.

The collision itself is NOT a probe.

Each item you process can only have zero or one collision—either its designated bucket is available *right now*, 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.

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?