Assignment objectives

In this lab we will investigate the effects of different hash table sizes and hash functions on the number of collisions that occur while inserting data into a hash table using simple hash techniques. For more details about hashing, see Lecture 6, and also Chapter 7.3 in the textbook.

You will be counting collisions, not probes.

Introduction

Your program will:

- Read an input file containing a list of names
- Store each name in 12 different hash tables
- Count the total number of *collisions* that occur with each hash table (***do not count probes***)

Why are there 12 hash tables? For the same input list (containing N names):

- You will use hash tables of 4 different sizes (N, 2N, 5N, 10N)
- You will use 3 different hash functions on each size of hash table (H1, H2, H3)
- Insert all names into all 12 combinations of these

Finally, you will run your program on 3 different data files, and record all of the results (36 numbers) on a separate worksheet. (Collisions, not probes.)

More details

Class information

Class name: Lab5

This class requires only one public method:

public void DoTheStuff(String filename): This method takes a single filename as an argument, and displays *all the required output for one data file* – 12 different result numbers.

In other words, it should be possible to complete the entire Results Table by running calling this method three times (one for each of the three data files).

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

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

Use arrays

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

Resolving collisions

For collision resolution, your program should use *closed hashing with linear probing*. This technique is described in the lecture notes, and page 272 of the textbook.

Hash table size

For each data file, you will eventually produce output for hash tables with four different sizes related to the size of the input file (N == number of names). The hash table sizes are: N, 2N, 5N, 10N. You'll need to read the entire data file *before you start hashing*, so you can create arrays of the appropriate sizes. You cannot do hashing in a dynamic array.

Hash functions

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

`H1(string, N)` – 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 N. For example, if the string is BENNY, the sum of the letters is 2+5+14+14+25 = 60, and then for the hash value would be 60 mod N.

`H2(string, N)` – 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 N. For BENNY the intermediate result would be 2*1 + 5*26 + 14*676 + 14*17576 + 25*456976 = 11680060. For the final answer you will take this partial result mod N. **WARNING: Watch out for integer overflow with long names in the data file!**

`H3(string, N)` – *Invent your own hash function!* Pull one right out of your imagination, or Google around. Write good and clear comments in your Java code describing how your hash function works. **If you found it online, give the source.**

Input files

There are three data files for you to use.

- 37_names.txt
- 333_names.txt
- 5163_names.txt

Tip: There are no newlines in these data files; the names are separated by commas. Read the contents as a single string and then use str.split() to divide it into an array of strings. Then loop over this array to process the strings as if they are "input".

Sample output

The following table shows the correct numbers for hash functions H1 and H2 for the given data files. All of your numbers for H1 and H2 should be exactly the same as these! (Unless I made a mistkae, which wouldn't be for the first time.)

Are you getting gigantic numbers, like in the millions? You are probably counting probes. See details above, in the section "**What are collisions? What are probes?**".

THIS IS NOT THE HAND-IN WORKSHEET. THIS IS JUST SHOWING YOU THE CORRECT ANSWERS SO YOU CAN SEE IF YOUR CODE IS WORKING.

Input file	**Declared size of hash table array**	**Hash function H1 #collisions (NOT probes)**	**Hash function H2 #collisions (NOT probes)**	**Hash function H3 #collisions (NOT probes)**
37_names.txt	37	19	20	Ymmv
37_names.txt	74	10	6	Ymmv
37_names.txt	185	7	5	Ymmv
37_names.txt	370	7	1	Ymmv
333_names.txt	333	289	177	Ymmv
333_names.txt	666	289	77	Ymmv
333_names.txt	1665	289	27	Ymmv
333_names.txt	3330	289	15	Ymmv
5163_names.txt	5163	5107	2615	Ymmv
5163_names.txt	10326	5107	1305	Ymmv
5163_names.txt	25815	5107	517	Ymmv
5163_names.txt	51630	5107	258	Ymmv

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 some other 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 have to 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.

You still have to DO the probing after each collision, because you have to put your item in the hash table somewhere. But you will NOT count the probes. You will only count the collisions.

Does the collision itself also count as a probe? Maybe, maybe not, it depends on your definition. IMHO this distinction is superficial/meaningless. (Why?) In any case, for us *it does not matter, because we are not counting probes.*

Each item can only have zero or one collision. Either its primary intended bucket is available right now, or not. The total number of collisions while hashing a total of N strings can never be bigger than N. ***If your numbers are bigger than N, you are probably counting probes. In case I have forgotten to mention this: we are NOT COUNTING PROBES.***

Submission information

Due date: As shown on Learning Hub.

Submit the following to the drop box on Learning Hub:

- Java source code (*.java file).
- The completed Results Table (Word, PDF, Excel, plain text—as always, I just want to see the numbers. Present them in whatever medium is easiest for you, but do present them in the same order/arrangement in a tabular format.)
- File names are not important.
- Please *do not zip* or otherwise archive your code. Plain Java files only.
- Please *do not zip* or include your entire project directory.

Marking information

This lab is worth 20 points.

Remember that a portion (probably 4/20 points) will be allocated to the COMP 3760 coding guidelines.

Did you write your Name/ID/Set# inside *every file* that you submitted?

Virtual donut

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

Can you find a hash function (for H3) that gives a lower score than H2 for *all* data files and *all* hash table sizes? (Many have tried, few have succeeded.)

P.S. Remember: collisions, not probes.