CS 337: Algorithms: Design & Practice

Lab#6: Hash Tables & Hash Functions – A Refresher and Collision Analysis

In this lab we will learn about hash functions that are used for indexing into hash tables. We will explore basic properties of hash functions.

Hash Tables: First, if you are not familiar with hash tables please review what they are and how they work. At the very least, you should be familiar with the terms: *collision*, *open addressing*, *chaining*, and *load factor*.

In this lab you will explore the core of hash tables, the hashing function. You will start by using the hashing function provided in the standard library of programming languages Python, Java, or Go. (Why not C?) Then you will implement and test your own hashing function.

Prior to hashing, we need some data to hash. For this lab, we will be using the same files as lab 4. So you can reuse much of the file reading apparatus your wrote for that lab.

Rather than building an inverted index (as in Lab 4), for this lab we will use a simple concordance in which the annotations of the words are the number of times the word appears. (A concordance contains all the words in a text, along with some sort of annotation. The inverted index for lab 4 is a concordance; just not the one to be used in this lab.) For instance, here is the concordance of the requested form on just the file GibonOne_95.txx

```
   chapter   1
     death   2
   passage   1
   against   1
      part   1
     roman   1
expedition   1
    jovian   1
       his   1
     saves   1
      xxiv   1
       and   2
        of   5
        by   1
 residence   1
successful   1
    tigris   1
         a   1
  election   1
disgraceful  1
         i   1
       the   5
   retreat   2
    julian   3
      army   1
        at   1
  persians   1
   antioch   1
    treaty   1
        he   1
```

Task#1: Write a program in Java or Python to read the data file from GibonOne_95.txx. You should create a dictionary (in Python) or a hash table (in Java) such that the keys are words and the values are the frequency of the words. Put the words into all lower case. Once the dictionary/table is populated a user should be able to put in an unlimited number of queries to determine if a word is in the concordance and if so how many times it appears. Your program should also print out the number of unique words in the concordance. For example, here is a trace of my program with only the the file GibonOne_95.txx:

```
The concordance has 30 unique words
Query the Concordance!
Word to ask about (hit return to quit): the
           the appears 5 times
Word to ask about (hit return to quit): julius
        julius is not in the concordance
Word to ask about (hit return to quit): tigris
        tigris appears 1 times
Word to ask about (hit return to quit): saves
         saves appears 1 times
Word to ask about (hit return to quit): hello
         hello is not in the concordance
Word to ask about (hit return to quit):
```

Once you have this working, expand your concordance to cover the entire set of documents in the data set.

Show the output of your program for five sample searches, three successful, two failed. Your output should be for the entire data set; again, the sample above is for only GibonOne_95.txx.

Hash Functions: Next, do a performance analysis of the hash function used in Python or Java or Go and of a hash function you will write based on the algorithm given below.

Python has a library called **hashlib** that provides a **hash()** function that is used to for hash code computations in dictionaries. Here is how it is defined:

`hash`(*object*)

> Python hash function. Defined in the library **hashlib**. Returns the hash value of the object (if it has one). Hash values are integers.

In Java, there is a method **hashCode()** available for objects. We will use the one for the **String** type. Here is its definition.

`int hashCode()`

> Java method. Returns a hash code for this object.

Example use (hash values shown may not match your results):

Python

```
>>> import hashlib
>>> hash("Bryn Mawr")
1536120907
```

Java

```
String s = "Bryn Mawr";
System.out.println(s.hashCode());
PRINTS-> 978918218
```

Inevitably, Go is a little different.   Here is the core:

```
    import "hash/fnv"

    input := "Bryn Mawr";
    h := fnv.New64a()
    h.Write([]byte(input))
    fmt.Printf("%v\n", h.Sum64())
PRINTS->7901606081717252057
```

Note that in both Python and Java, the value returned is an integer (could be positive or negative). In Go, the number must be positive.

Task#2: Write Python, Java or Go programs to compute the hash codes as shown above. Your program should show the hash values for the strings: Bryn Mawr, k-Cass, Haverford and Swathmore. This program will be really short (and largely a copy of the code above).

Task#3: Collision performance of built-in hash functions. In order to examine the collision performance of library hash functions, you can create an array of counts (initialized to zero). The size of the array should be the number of words in the concordance. Each index in the array represents a value where a hash code could be mapped. Then, for each word, map the hash code generated to an entry in the array. For example, if the array has *N* entries, the hash code mapping a zip code to an index in the array will be[1]:

$$\text{index} = \text{h(word) mod N}$$

Where *h()* is the hash function. (You might need to include an absolute value function in also.) The value at any index in the array will be the number of words that indexed (collided) into that location. Thus if the entry at 6532 = 3, it implies three words had the same hash code (6532). Depending on the value of *N* you will experience 0 or more collisions. How many collisions occur?

There are ~69000 unique words in the entire data set. Starting with *N*=70,000 run your program and record the total number of collisions. Repeat, increasing *N* by 70,000 each time and recording the number of collisions. (Keep in mind that a value of 2 indicates 1 collision.) Do this until the number of collisions reaches 0. Realistically, close to zero is all that is achievable; with a table size of 800 million I still had more than 20 collisions using the Java library hash. So define "close" fairly liberally, maybe less that 100? For each value of *N*, also compute the *load factor* (ie the percentage of spaces in the array that are unused).

Plot the two data sets (*N* vs. # collisions and *N* vs. load factor) in separate plots.

[1] Remember Lab#1?!!!

Task#4: Your own hashing function implementation

Horner's method is a common hashing algorithm. It is given by the algorithm below.

Horner's method:

```
Given S : a string
Result: an integer in the full range of integers
Let:    sm=0
        mul= a 2 or 3 digit prime number > 50
With each ch in the characters of the string
        Let vch = integer value of the character
        sm = sm*mul + vch
return sm
```

Implement Horner's method and repeat task #2.

Task#5: repeat task 3

Repeat task 3, using your implementation of Horner's method. Is Horner's method better, worse or about the same as the library hasher? (Be sure to clearly define "better".) Does the choice of the value of "mul" have a significant effect? Why should "mul" be prime?

What to submit:

1. A report containing a discussion on this lab: implementation and use of hash tables, the performance of hash functions, based on the data you collected. Include and use the plots in your narrative. You should only have two plots. Each should include lines for both the library hasher and your implementation of Horner's method. 2-3 pages.
2. An appendix containing:
   a. A printout of sample runs from the program in Task#1, 2 and 4.
   b. Significant code (for instance, the code for reading the text files is not significant).