Computational Problem Solving CSCI-603
Hash Function Evaluation Lab 7

1 Introduction

First we will study some properties of hash functions.

In both open addressing and chaining, collisions degrade performance. Many keys mapping to the same location in a hash table result in a linear search. Clearly, good hash functions should minimize the number of collisions. How might the "goodness" or "badness" of a given hash function be measured by looking at the hash table after it has loaded its entries?

This may be best explained by an example. Look at Figure 1. A program entered 5 keys – A, B, C, D, and E – into a chaining hash table of length 10.)

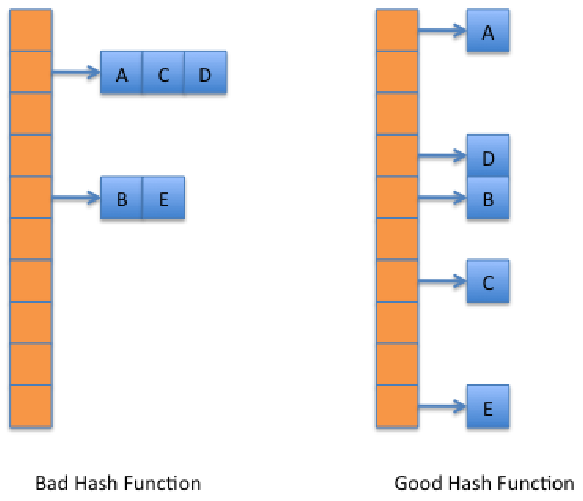

Figure 1: Two hypothetical hash functions applied to the same hash table array and the same stream of keys being put in the table

The figure shows the application of two different hash functions to this scenario. Clearly the second function is better, but how can you quantify that based on what you can measure in the tables?

2 Problem-Solving

1. Consider an integer-to-string conversion function that simply adds their ordinal letter values together. (a=1, b=2, etc.) As an example of the encoding, here is how the first key given in part (a) converts to a number.

   $$\text{'l','a','d'} \rightarrow 12 + 1 + 4 = 17$$

   (a) What is the hash value for each of the following strings?

   ```
   "lad"   "but"   "is"   "chin"

   "be"    "fun"   "blab" "too"
   ```

   (b) Consider an open-addressing hash table of size 12. Draw the contents of the table if these 8 strings are added in the order given (top row left-to-right, then second row). Do not worry about any associated value for each key.

   (c) We will define a *collision* as any time two keys have the same hash value (after the modulus operation). How many collisions occur during the insertion of these 8 keys?

   (d) We can further define a *probe* as each time the contents of a cell in the table are tested. How many probes occur during the insertion of these 8 keys?

2. Write code that implements a hash function that takes a string `s` and sums up the ordinal values of the characters in `s` scaled by 31 to the power of the index at which that character occurs in the string, e.g.:

   'l','a','d' → `ord('l') + ord('a')*31 + ord('d')*31*31`.

3. Consider a hash map used to keep track of the number of times each different word appears in a document. That is, the hash map will keep the mapping between words and the number of times the word appears. As the document is read, for each word, you will need to create or update the given entry.
   Write code that performs this operation for a given word `w`. You can assume that the hash map has operations `put(key,value)`, `get(key)` and `contains(key)`, and that `get` will throw a `KeyError` exception if passed a key not in the map. Consider how you can minimize the amount of operations performed in the map.

3 Implementation

The goal of this assignment is to examine how effective different hash functions are for storing English words. The program you will create and submit should be called `hashtable.py`.

3.1 Instrumenting the Hashmap

For the first part of the implementation, you will add some code to a hashmap implementation to keep track of both the number of collisions and the number of probes required to add and search for keys in a hash map. In particular, you should start from the `hashmap.py` provided in the course website. The `put` function should count both collisions and probes (as defined in the problem solving portion of this assignment) and the `get` and `contains` functions should add to the count of probes.

The next step is to modify the hash map implementation to use different hash functions. That is, the constructor should accept an argument of a hash function in addition to any other argument(s) it might need.

Then, you should write at least two significantly different hash functions for strings. By significantly different, we intend that they compute the hash in qualitatively different ways (not just changing some constants). They should attempt to be good hash functions, not just trivial ones!

3.2 Main method implementation

Now, it will be time to perform some testing. You will write a main method that the words in a text file into a hash map to count how many times each word appears. After it does so, it should then iterate through the hash map to find one word that has appeared the maximum number of times (if multiple words are tied, printing any such word is fine). It should also then print out how many probes and how many collisions occurred during this entire process. Write the main method so that it will test the same words on hash maps with *different hash functions and different maximum load factors* and print out the statistics for each separate case.

To remove punctuation from a word (and between words), you can use the `re` [regular expression] module. First `import re` and then the function call `re.split('\W+',t)` will take a line of text in the variable `t` and return a list of the words in `t` (skipping spaces and punctuation). Note that this will also split the word `don't` into `don` and `t` but for this exercise we will not worry about such details. You should also convert all words to lower case.

3.3 Evaluation

To get some good data upon which to test your hash functions, you should get two full novels from Project Gutenberg (`http://www.gutenberg.org`). These should each contain at least 50,000 words. You should also use a dictionary such as the one provided in Ubuntu or OS X at `/usr/share/dict/words` (this file actually contains different dictionaries in the two operating systems!). Send each of these through your main method, testing your two hash functions as well as Python's builtin `hash` function over three different maximum load factors of your choice.

Then, you should collect and report on the data that you have obtained. Compare the results from the different input files, hash functions and load factors in tabular and/or graphical form, **putting this in a PDF document**.

4 Grading

- (20%) Problem Solving
- (60%) Implementation
  - 15% Hash map instrumentation
  - 15% Hash functions
  - 25% Main testing method
  - 5% Style
- (20%) Evaluation, including report

5 Submission

One team member should upload all the source files to his/her CS account and run the following try command:

```
try grd-603 lab7-1 hashtable.py report.pdf
```

6 Problem Solving Solutions

1. (a)
   - lad: 17 (12+1+4)
   - but: 43 (2+21+20)
   - is: 28 (9+19)
   - chin: 34 (3+8+9+14)
   - be: 7 (2+5)
   - fun: 41 (6+21+14)
   - blab: 17 (2+12+1+2)
   - too: 50 (20+15+15)

   (b) For this one, ignore the brackets (which are the original modded hash values)

0	
1	
2	too
3	
4	is
5	lad
6	fun[5]
7	but
8	be[7]
9	blab[5]
10	chin
11	

   (c) 3 collisions

   (d) for each added key, probes are 1, 1, 1, 1, 2, 2, 5, 1. Total 14.

2.

```
h = 0
mult = 1
for ch in s:
    h += ord(ch)*mult
    mult *= 31
```

3. Several things are possible. This is best for efficiency (I think, given that we are external to the table):

```
try:
    count = map.get(w)
    map.put(w,count+1)
except KeyError:
    map.put(w,1)
```