CSCI 262 Data Structures

Spring 2019

Lab 8: Hashtable

(Back to Labs)

Goals

- Experiment with hashtables

Overview

This lab will walk you through some experiments concerning hashtables. As usual, we ask that you work with a partner or a team.

Instructions

Step 0: download lab8.zip, which contains main.cpp (the code you will experiment with), and dictionary.txt (a list of English words). Create a project just containing main.cpp, and put dictionary.txt into the cmake-build-debug directory.

Start by browsing through main.cpp to see what it is doing. The comments should guide you pretty well, but here is a brief overview :

First, take a look at the hash functions (hash1 - hash4), to see roughly what they are doing or where they come from. These functions all generate a hash code for strings. [Note that "hash5" is not actually a hash function in that it is not deterministic - instead, it simply generates random numbers. Recall that the O(1) average complexity of hashtable operations relies on the assumption of uniform distribution of hashcodes. We will use the hash5 function to get a sense of what would happen if we truly had a uniform distribution of hashcodes.]

Next, look at main(). There are a couple of lines of code (lines 92 and 93, if you have line numbering on) which you will modify to experiment with various properties of the hashtable. (The basic mechanism here is that you will edit these lines, recompile, and run. Nothing fancy like an actual user interface!) After those lines, the code creates a sort-of hashtable (with chaining) using an array of vectors of string. This is a perfectly workable hashtable, but would not be optimally efficient in practice due to the use of vectors instead of linked lists for chaining. However, for our purposes, vectors are a better fit (see comment in the code).

The next bit of code reads in every word from dictionary.txt, calculates its hash code using whichever hash function is setup in line 93 of the code (as provided, this is hash1), and then adds it to the vector at the index calculated by taking the hash code mod the table size - just as we described the process in lecture.

Finally, main() creates a map of integers to integers, which will be used to store a *histogram* of the counts of hashtable indices according to the number of entries at the index. That is, we will count up how many hashtable indices contain 0 entries, how many contain 1 entry, how many contain 2 entries, etc., and then we print this information out ordered by the sizes.

This will give us some intuition about the quality of each hash function: if we have indices with many entries, it means looking up entries in the hashtable will be expensive for certain words. If we have many indices containing zero entries, it means we aren't distributing our hashcodes well across the available indices. Recall that the ideal hashtable situation is that every key stored in the table is in an index by itself.

Here's an example of what the histogram looks like (fake data, though):

```
size |  count
-----+-------
   0 |  10000
   1 |   4000
   2 |   1500
   3 |    200
   4 |      2
```

The above histogram would tell us that the largest "bucket" in our hash table only holds 4 entries - and there are only 2 such buckets. That's pretty good - it means we have look at 4 values at most to determine whether something is in our hashtable or not. On the other hand, we have 10,000 indices which contain zero entries - that may indicate that our hashtable is larger than it needs to be.

Step 1: hash function quality

For this step, you will be answering **questions 1 - 3** in the lab 11 quiz in Canvas. To get your experimental results, modify line 93 to substitute in each hash function (hash1 - hash4) in turn, then record your answers to each question. Think about what each question is asking, and what it says about the quality of each hash function. Are there clear qualitative differences between the different functions? Which one(s) perform best?

Step 2: the birthday paradox

In class I made the claim that collisions were inevitable. We are going to test that assertion (up to a point - we don't have infinite memory on our computers) by increasing the size of the hashtable to see if we can get every string to occupy a unique index in the hashtable. The initial size of the hashtable (as provided) is 1.333333 times the number of words we will store in the hashtable, which gives us a ratio of dictionary size to hashtable size of 0.75. Use the best hashing function you found in step 1. Try increasing the multiplier to 2, then larger numbers. See how large you can get before every word occupies its own entry (your histogram output should indicate this by having no sizes greater than 1). Record your conclusion as the answer to **question 4**.

Step 3: random

Recall that the O(1) average complexity of hashtable operations relies on the assumption of uniform distribution of hashcodes. The function `hash5` provides a way to test what would happen if our hashcodes truly were uniformly distributed, because hash5 samples numbers from a uniform distribution (as best as can be done using a pseudo-random number generator, or PRNG). Try running the code with hash5 and a multiplier of 1.333333, and compare the results to your answers to questions 1 - 3. What can you conclude about the quality of the different hash functions? Answer **question 5**.

Additional reading:

- this site lists a few different hash functions, including one equivalent to our hash1, and one very similar to our hash3, with some background information about each.
- You can read more about the Birthday paradox on Wikipedia.