Project 5 : Perfect Hashing

Due

Please check due date on BlackBoard.

Objectives

The objective of this programming assignment is to increase your understanding of hashing by implementing a perfect hashing algorithm and by collecting statistics on its performance. For extra credit you will also research and learn how to serialize data in C++.

Introduction

In this project, you will implement the perfect hashing scheme using universal hash functions. We summarize the main ideas in the internal links in the previous sentence. You may need to do additional research.

First, perfect hashing is hashing without collisions. Perfect hashing shows that if you hash n items into a table of size n^2 with a randomly chosen hash function, then the probability of not having any collisions is greater than 1/2. What happens if you are unlucky and you get a collision? Then, pick another hash function and try again. With independent trials, the expected number of attempts you have to make in order to achieve zero collisions is less than 2.

A table of size n^2 is really big and a huge waste of memory. So, in our perfect hashing scheme, we don't directly hash to a table of size n^2. First, we hash into a *primary hash table* of size n. There will be some collisions since this . To resolve the collisions at a slot of the primary hash table, we create a *secondary hash table*. If t items collide at a certain slot of the primary hash table, then we create a secondary hash table of size t^2 and use perfect hashing to store the t items. The expected number of slots used by all of the secondary hash tables is less than $2n$. If you are thinking that this hashing scheme is just separate chaining with the linked lists replaced by hash tables, that is pretty close — just remember that the linked lists are replaced by *collision-free* hash tables.

How do you search for an item in this hashtable? You have to hash twice. First, you hash the item to find its slot in the primary hash table. If that slot is not empty, then you find

its slot in the secondary hash table. If the slot in the secondary hash table is also non-empty, then you compare the item against the item stored in the secondary hash table. If there's a match, you found the item. Otherwise, the item is not in the hash table.

Note that each secondary hash table has its own hash function, since it might have been necessary to try a few hash functions before you found one that did not result in any collisions. So, the hash function would have to be stored in the secondary hash table.

The perfect hashing scheme described above requires the ability to "randomly pick a hash function." In particular, we have to be able to randomly pick a *different* hash function if the one we just tried doesn't work because it resulted in a collision. How do we do that? This accomplished by "universal hashing". (Never mind the word "universal". It is a bit of a misnomer. It should really be called "randomized hashing", but most people think hashing is already random, so "randomized random" doesn't make much sense either.)

The universal hash functions presented in the links above provide a method for generating random hash functions. First, we need a prime number p that is larger than any key that will be hashed. Then, we select two random integers a and b, such that $1 \le a \le p - 1$ and $0 \le b \le p - 1$. Then, we can define a hash function $h_{a,b}( )$ using these two random integers:

$$h_{a,b}(x) = ((a x + b) \bmod p) \bmod m$$

where m is the table size (which does not need to be prime in this scheme). Thus, for every pair of a and b, we get a hash function.

To hash strings we first convert the string into a number, then we use the hash function above to guarantee "universality". In one scheme, we pick a random constant c such that $1 \le c \le p - 1$. Then, we interpret each character of the string as a number (think ASCII), so a string str becomes a sequence of numbers d[0] d[1] d[2] d[3] ... d[t]. Now we can convert the string into a number:

$$g_c(str) = \left(\left(\sum_{i=0}^{len} x_i \cdot c^i \right) \mod p \right)$$

g(str) =

We would have to make sure that the arithmetic does not result in any overflows. (This can be accomplished by "modding out" by p at every step.)

The last remaining thing to point out is that this perfect hashing scheme only works if we know all the keys in advance. (Otherwise, we cannot tell how many items hash into the same slot of the primary hash table.) There are several applications where we would know the keys in advance. One example is when we burn files onto a CD or DVD. Once the disc is finalized, no additional files can be added to the disc. We can construct a perfect hash table of the these filenames (perhaps to tell us the location of the file on the disc) and burn the hash table along with the files onto the disc. Another example is place names for a GPS device. Names of cities and towns will not change very often. We can build a perfect hash table for place names. When the GPS device is updated, a new hash table will have to be constructed, but updates are not frequent events. This last example is the basis of your programming project.

Assignment

Your assignment is to apply the perfect hashing scheme described above to a file containing approximately 16,000 city names in the United States. The data comes from geonames.org. In addition to the names of all the cities with population above 1000, the file also has the latitude and longitude of each of these cities. Here are a few lines from the file:

```
Abington, MA
42.10482 -70.94532
Abita Springs, LA
30.47853 -90.03758
Abram, TX
26.1998 -98.41113
Absarokee, MT
45.5205 -109.44294
```

The data for each city is stored in two lines of text. The first line is the name of the city followed by the state. The second line of text has the latitude and the longitude of the city (in that order). You should treat the city name and state as a single entity to be hashed — i.e., hash the string "Abington, MA" (comma included) rather than "Abington". You may assume that the city names with the state designation included is a unique string, even though this is not entirely true in real life. (For example, there are 3 separate cities in Indiana named "Georgetown". Two of these have been removed from our file.)

A sample input file with sample output is available.

You are required to write a program which takes a file provided as a command line argument and creates a hash table using perfect hashing. You will use the city names to create the hashes. While creating the hash table, the program must also print out some statistics about it.

- Values for Prime 1 ($g_c()$) , Prime2 ($h_{a,b}()$), and hash values (a,b,c)
- Number of cities read from the file.
- Maximum number of collisions in a slot of the primary hash table.
- For 0 <= *i* < 10 the the number of primary hash table slots that have *i* collisions.
- List of cities (and the latitude/longitude) in the primary hash table slot that has the largest number of collisions. If there are more than one pick the slot that has the lowest key.
- For 0 < *j* <= 10 the number of secondary hash tables that tried *j* hash functions to create a collision free hash table. Do not include the primary hash table slots that did not have any collisions from the statistics.
- The number of secondary hash tables with more than 1 item.
- Average number of hash functions tried per slot of the primary hash table that had at least two items. Do not include the primary hash table slots that did not have any collisions from the statistics.

You will also be required to prove your hashing by retrieving the latitude/longitude for a place from the hash table(s).

Constraints and Requirements

There will not be many constraints placed on this project. You are required decide your own way forward with this project. For instance, hash tables are typically stored as arrays. You may, however, use any data structure you prefer for this as long as it meets the constraints below:

- Since your project has to run on GL you may not use C++11, or any C++11 structure.
- You must have different classes for your main hash table (and methods) and your secondary hash table (and methods).
- Your main driver must be Project5.cpp and the executable must be Project5.out.
- Your code must be logically organized and commented properly.
- Run time must be linear.
- The initial seed for the random number generator should be 0 and when it is reseeded should be a linear progression (i.e. += 1)
- You must use Prime1 = 16890581 and Prime2 = 17027399 for the two primes.

- Your project must be fully functioning and stand alone. This means you must write the main driver as well as all supporting classes.
- You must supply a const SEARCH_FOR_CITY at the top of the driver for the name of the place being looked up. We do reserve the right to change this place name. In case this city is not found, return "n/a" (without quotes).

Code Structure

This project gives you a lot of freedom in implementation. Keep in mind your project should still follow all coding standards, be commented well and be organized well. Proper OOP principles should be adhered to, this means a solid class design with good interfaces and flow.

What to Submit

Read the course project submission procedures. *Submission closes by script immediately after midnight*. Submit well before the 11:59pm deadline, because 12:00:01 might already be late (the script takes only a few seconds to run).

You should copy over all of your C++ source code under the src directory. You must also supply a Makefile. Do not submit the text file, or your binary file if you are doing extra credit.

Make sure that your code is in the ~/cs341proj/proj5/src directory and not in a subdirectory of ~/cs341proj/proj5/.

You should have a run: target in your Makefile such that:

make run FILE=inputfile.txt

Extra Credit

For extra credit you will create a utility program (Project5ToBin.cpp) that will read the text file in and rewrite it to disk as a binary file, both files names should be specified on the command line. This process is called serialization and you will need to research how to do this on your own.

You will then copy your Project5.cpp to Project5EC.cpp. In this driver you will read US_Cities_LL.bin and create your hash tables from it. You should also add this to your Makefile with different target (runbin) to run Project5EC.out

make rubin FILE=inputfile.bin

NOTE: To get credit for the extra credit you will be turning in 2 driver programs with the correct names and functions.

Addendum:

Nothing yet.