CMPSC 130A: Programming Project #1B

Fall 2021

Due: Monday October 25, 2021 at 11:59PM

Late Policy for Programming Projects

- **For programming projects, late submissions will be accepted (see below).**
- **10% of the grade will be deducted for every 0+-24 hours of delayed submission.**
- **No fractional credit, i.e., if you miss the deadline by an hour, there will be a 10% deduction.**

Objective

The objective of this programming assignment is to reinforce your advanced and deep understanding of hashing techniques by implementing a perfect hashing algorithm and collecting some statistics on its performance.

Introduction

In this project, you will implement the perfect hashing scheme using universal hash functions. We summarize the main ideas here. For details and proofs, please refer to the resources shared with this assignment.

First, perfect hashing is hashing without collisions. 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 (see Theorem 5.2 in PerfectHashing.pdf). 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.

Now, a table of size n^2 is really big and a huge waste of memory. So, in our perfect hashing scheme, we do not 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. 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 $2 \cdot n$ (see Theorem 5.3 in PerfectHashing.pdf). 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. You must notice that you can easily use the code base of your project #1A here with some minor modifications.

How do you search for an item in this hash table? 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 is 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 does not work because it resulted in a collision. How do we do that? This is accomplished by "universal hashing" (see UniversalHashing.pdf). (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 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 $1 \leq a \leq p - 1$ and $0 \leq b \leq p - 1$. Then, we can define a hash function $h_{a,b}(X)$ using these two random integers:

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

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. The random hash functions chosen this way satisfies the definition of “universal” and has provably good performance (see Theorem 5.4 in UniversalHashing.pdf).

To hash strings we first convert the string into a number, then we use the hash function above to guarantee “universality”. In Project #1A, you designed your own scheme for hashing arbitrary strings into numbers using Homer’s rule. In order for you to become familiar with C++ library on hashing, we would like you to instead consult C++ STD library and use the appropriate API for doing the conversion (you may wonder why are we doing this when you have already developed this functionality in Project #1A? Well, hashing is a favorite topic for Tech Interviews and having a comprehensive understanding both at the low level as well as familiarity with the C++ STD library on hashing is extremely important. The examiner may ask you a question on hashing and may ask you to develop your code using C++ library interface): https://en.cppreference.com/w/cpp/utility/hash

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 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. There are many such applications where the dataset is known in advance.

Project Description

In this project also you have to develop a variant of a separate chaining based hashing scheme to a file containing approximately 350,000 dictionary words with one word per line. Here are a few lines from the file:

```
preservations
```

```
preservative
preservatives
preservatize
preservatory
preserve
```

Since implementing universal hash functions can be a little bit tricky, a C++ class Hash24.h that implements universal hash functions is provided. The 24 in Hash24 indicates that the methods in the class work with values as large as 2^{24} which is approximately 16 million. This is more than large enough for the purposes of this project. To use the Hash24 class, simply create a Hash24 object and then use it to invoke the universal hash function:

```
Hash24 h1;
...
index = h1.hash("test");
```

The Hash24 object h1 “remembers” the randomly chosen a, b and c used in the universal hash function. So, you can store h1 and retrieve it later and it can be used to compute the same hash function.

You will implement your programs just like in Project #1A. First you will read the dictionary words from a text file, create a hash table using perfect hashing. While creating the hash table, the first program must also print out some statistics about the hash table (see implementation details). Next you will read the list of query words from the second file and execute the queries on the hash table

Implementation Details

For this project, you should use PA1B_dataset.txt available on gauchospace to build the hash table. For development purposes, we have also provided a smaller dataset PA1B_dataset_10000.txt with only 10k words.

Dictionary

Your should create a hash table class called Dictionary which implements the following methods:

- A constructor that takes the name of a filename (dataset) and filename (parameters):
  `Dictionary(String fname, String paramFileName)`
  This constructor should use the information in the file to construct the hash table using the perfect hashing scheme described above. The size of the primary hash table should be tsize. While constructing the hash table, this constructor should print out the following statistics (see Input and Output section for the formats):
  – a dump of the hash function used (use dump() from Hash24).
  – number of words read in.
  – primary hash table size.
  – maximum number of collisions in a slot of the primary hash table.
  – for each i between 0 and 20 (inclusive), the number of primary hash table slots that have i words.
  – all the words in the primary hash table slot that has the largest number of collisions. If there is more than one such slot, pick one arbitrarily.
  – for each j between 1 and 20 (inclusive), the number of secondary hash tables that tried j hash functions in order to find a hash function that did not result in any collisions for the secondary hash table. Include only the cases where at least 2 words hashed to the same primary hash table slot. (i.e., we exclude the primary hash table slots that did not have any collisions from the calculations.)
  – The average number of hash functions tried per slot of the primary hash table t that had at least two items. (As before, we exclude the primary hash table slots that did not have any collisions from the calculations.)

  Note that this constructor cannot begin constructing secondary hash tables until all of the data has been read in. So, construction of the hash table takes two passes. The first pass reads in each word from the file and figures out where it belongs in the primary hash table. The second pass looks at each slot in the primary hash table and creates a secondary hash table for each slot where this is needed.

- **`bool find(String word) ;`**
  The find() method should query the hash table for the string word and return true if **`word`** is present in the hash table else return false. This method should have O(1) time complexity.

Input and Output

Your program should take three filenames as parameters. Gradescope will pass the parameters filename via argv[1] dataset filename via argv[2] and the query words filename via argv[3].

Example: ./project1b parameters.const dataset.data queries.query

Sample Output format:

```
*** Hash24 dump ***
prime1 = 16890581
prime2 = 17027399
random a = 5065039
random b = 9597616
random c = 16236226
Number of words = 350000
Table size = 16000
Max collisions = 6
# of primary slots with 0 words = 5958
# of primary slots with 1 words = 5801
# of primary slots with 2 words = 2965
# of primary slots with 3 words = 963
# of primary slots with 4 words = 260
# of primary slots with 5 words = 41
# of primary slots with 6 words = 12
# of primary slots with 7 words = 0
# of primary slots with 8 words = 0
# of primary slots with 9 words = 0
# of primary slots with 10 words = 0
# of primary slots with 11 words = 0
# of primary slots with 12 words = 0
# of primary slots with 13 words = 0
# of primary slots with 14 words = 0
# of primary slots with 15 words = 0
# of primary slots with 16 words = 0
# of primary slots with 17 words = 0
# of primary slots with 18 words = 0
# of primary slots with 19 words = 0
# of primary slots with 20 words = 0
```

```
** Words in the slot with most collisions ***
hello
welcome
to
data
structures
Class
Number of hash functions tried:
# of secondary hash tables trying 1 hash functions = 3141
# of secondary hash tables trying 2 hash functions = 808
# of secondary hash tables trying 3 hash functions = 220
# of secondary hash tables trying 4 hash functions = 57
# of secondary hash tables trying 5 hash functions = 7
# of secondary hash tables trying 6 hash functions = 7
# of secondary hash tables trying 7 hash functions = 1
# of secondary hash tables trying 8 hash functions = 0
# of secondary hash tables trying 9 hash functions = 0
# of secondary hash tables trying 10 hash functions = 0
# of secondary hash tables trying 11 hash functions = 0
# of secondary hash tables trying 12 hash functions = 0
# of secondary hash tables trying 13 hash functions = 0
# of secondary hash tables trying 14 hash functions = 0
# of secondary hash tables trying 15 hash functions = 0
# of secondary hash tables trying 16 hash functions = 0
# of secondary hash tables trying 17 hash functions = 0
# of secondary hash tables trying 18 hash functions = 0
# of secondary hash tables trying 19 hash functions = 0
# of secondary hash tables trying 20 hash functions = 0
Average # of hash functions tried = 1.3508606
Queries:
hello found at 4752
welcome found at 6541
to not found
cs130a not found
```

The project needs to be implemented in C++ and uploaded to Gradescope's "Project 1B" assignment. It will be graded using autograder, so be very precise about the input and output formats discussed above. Your submission should contain source files along with a makefile. Please DO NOT upload PA1B_dataset.txt file to gradescope. The name of the executable generated by makefile must be project1b. Please note that it is possible that your program could have some unexpected behavior on Gradescope's

autograder compared to whatever machine you wrote the code on, so submit your program early and make sure it is bug free on Gradescope. You are to implement your own Hash Table, so do not use any library that automatically does this for you (i.e. it is NOT OK to use STL containers other than vector, string and fstream). Gradescope score will be manually overridden if any such library is used.

Note: In order to check your program thoroughly, we will manually look at the Dictionary class implementation and the output statistics from running project1b on PA1B_dataset.txt file. Please make sure you name the executables as mentioned above (this can be done using makefile). Not adhering to the instruction might result in significant delays in grading or no score for the assignment.

Autograder Specification:

Please find attached hash24.cpp, hash24.h, sample_output.txt, sample makefile and test input files.

Make use of modified hash24.cpp and hash24.h provided to you.
Gradescope will pass the parameters filename via argv[1], dataset filename via argv[2] and the query words filename via argv[3].
Your executable has to be named **project1b** (ref sample makefile)

Example: ./project1b parameters.const dataset.data queries.query

Each parameter file will have four entries in it

1. Primary hash table size
2. Value for random_a
3. Value for random_b
4. Value for random_c

Primary hash function has to be initialized using ***Hash24(unsigned long rand_a, unsigned long rand_b, unsigned long rand_c);*** constructor by passing in values from the parameter file random_a, random_b and random_c.

Use **Hash24();** constructor for all secondary hash functions.

Please refer to the sample_output.txt for output formatting.