Exercise 10 EE312 Fall 2014 TWO POINTS

Hopefully, the exercises here will be good practice for Exam 2.

Good luck, you should have no problem completing Part 1 in recitation. Part 2 is very (very) good practice for the final!

Part 1: Evaluation of Hash Functions for Strings. As I'm sure you're well aware, the performance of a hash table relies on having a "good" hash function. Well, in this exercise you'll evaluate a couple of hash functions and see what sort of different that might make. We'll keep it fairly simple. I've written some code for you that loads the dictionary (from Project 1) into a vector (each element in the vector is one word from the dictionary). There are some 40,000+ words in that dictionary, so this gives us a reasonable sample of strings to work with. Your job is to simply run a hash function on each word in the dictionary and see what table index that word would hash to. OK, it's not that simple, but that's the basic idea. Here's what I want you to do.

- Run a hash function for each word in the vector to get the "hashcode" for that word.
- Take each hashcode you get, and convert that hashcode into a table index
- Construct a histogram of all the table indices you create. In other words, count how many times you have table index 0, how many times you have table index 1, how many times you have table index 2, and so on.
- Print the maximum value, mean value, and standard deviation of that histogram. Since the histogram is a direct calculation of the chain lengths that you would have in a hash table, the mean value is the load factor, and the max and standard deviation are "quality measures". The smaller these two numbers are, the better your hash function has performed.

Now, a few more details. I want you to compare two different hash functions and compare three different methods for mapping hashcodes to table indices. That gives you a grand total of six sets of statistics. The two hash functions are:

1. *SimpleHash* – return an unsigned integer that is the sum of all the ASCII values in the characters that make up a string.
2. *SmartHash* – return an unsigned integer that is the sum of all ASCII values in the characters that make up a string, except multiply the running sum by 33 before adding each subsequent character.

If these descriptions don't make perfect sense to you (they should), don't worry. I've provided implementations of both functions in the starter file. Keep in mind that hashcodes can be arbitrarily large (subject to the size of integer we're using – 32-bit in our case). The hash tables themselves are much smaller, so we need to convert the hash code into an array index. For this exercise, I want you to just worry about the simplest possible method for converting hashcodes to array index, take the hashcode modulo the array size. However, there are schools of thought that say we should consider other methods, so if you have time, it takes only a couple of minutes to implement all three of the conversion methods below. See if the extra complexity of these other methods is worth anything... The three different methods for mapping hashcodes to table indices are as follows

1. Use a simple % operation that takes the hashcode % table_size. Use a table_size that is 65,536 (the first power of 2 that is larger than our dictionary)

2. Use a simple % operation that takes the hascode % table_size, but this time use 65,519 for the table size (65,519 is prime).
3. Use the Knuth multiplication method with a table_size of 65,536. I've provided an implementation of the Knuth multiplication method.

Again, I only need you to try table indexing method #1 for this exercise, but if you have time, you can try all three table index mapping methods with both of the hash codes and print out six sets of results. To get you started, I've coded up one of those six sets already (Knuth multiplication with Simple Hashing). Do the same thing (and print the same statistics) for the five other combinations.

Part 2 (left as a study problem – I don't expect you'll have time during recitation). When I did the HashTable example in class, I kinda skipped the part where the array of chains needs to be resized once the load factor gets to be above a threshold. Recall that as more and more elements are added to the table, the average chain will (naturally) get longer. The length of the average chain is called the "load factor", and it can easily be calculated as the ratio of the number of elements in the table divided by the size of the array. We typically want to keep the load factor below some constant (e.g., less than 0.8). When we talked about the HashTable example, I pointed out that we'd (roughly) double the size of that array every time the load factor exceeded 0.8. Of course, I totally neglected to write the code to do this. That's where you come in.

I've copied/pasted the HashTable example into the exercise directory. I've also updated HashTable::insert so that it check to see if the load factor is too large. If the load factor is too large, then insert will invoke a function called "resize". You must write resize. To receive full credit, your code should not only work correctly, but it must also not allocated any new Cell objects. Simply allocate a new array of Cell pointers (i.e., use "new Chain[cap * 2]" or something like that), and then re-link all the Cells from the old table into their new positions in the new table. Keep in mind that the cells may not hash into the same table index when you resize the array! So, if Cell x and Cell y were in the same Chain before you resize, they may be in totally different Chains after you resize. Oh, and please do not create any memory leaks when you write resize.

All of the code you write for this exercise should fit nicely inside Exercise9.cpp. Hopefully you will not need to modify main.cpp or any of the .h files