CSC230 Lab 9

Due: Apr. 10th, 11:59pm

Goal: This lab will teach you about hash function.

Part I

Write a **lab9.cpp** file. Inside this file, implement a hash function called **hashcode()**. The head of this function should be:

long hashcode(char* s)

This function takes a c-type string as parameter, and calculates the hash code of this string. The mathematic definition of this hash function follows the hashcode() definition in JAVA. That is:

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

where s[i] denotes the ith character of the string, and n is the length of s.

Compression functions

A simple compression function can look like this

h(i) = |i| mod N.

In fact, it just shuffles the buckets to different indices. A better compression function is

h(i) = ((ai + b) mod p) mod N,

where p is a large prime that's substantially bigger than N. (You can replace the parentheses with absolute values if you like; it doesn't matter much.)

For this lab, we use **h(s) mod 10007**. The bottom line is whether you have too many collisions or not in Part II. If so, you'll need to improve your hash code or compression function or both.

Part II

Download the provided files on Canvas to your working directory. When you run the code, provide one file name as the parameter at the command line. The program should read the contents of the file, calculate the h(s) mod 10007 value, checks whether some other string generates the same value. If there, increase the collision counter by one. After processing all the inputs, the program prints out the total number of strings and the number of collisions. For example, we list the following command line and the results of this command.

```
jli$ ./a.out f1
Total Input is 10000
Collision # is 4744
```

A tutorial on collision probability

People are always surprised when they find out how many collisions occur in a working hash table. You might have the misimpression that there won't be many collisions at all until the table is nearly full. Let's analyze how many collisions you should expect to see if your hash code and compression function are good. Here, we define a "collision" to be the event where a newly inserted key has to share its bucket with one or more previously inserted keys. (We count that as only one collision, regardless of how many keys are already in the bucket.)

If you have N buckets and a good (pseudorandom) hash function, the probability of any two keys colliding is 1/N. So when you have i keys in the table and insert key i + 1, the probability that the new key does **NOT** collide with any old key is $(1 - 1/N)^i$. If you insert n distinct items, the expected number that **WON'T** collide with any previous item is

$$\sum_{i=0}^{n-1}(1-\frac{1}{N})^i = N - N(1-\frac{1}{N})^n$$

so the expected number of collisions is

$$n - N + N(1-\frac{1}{N})^n$$

Now, for any n and N you test, you can just plug them into this formula and see if the number of collisions you're getting is in the ballpark of what you should expect to get. For example, if you have N = 100 buckets and n = 100 keys, expect about 36.6 collisions.

Wrap up

You should write the code inside lab9.cpp file. Jar you C++ files and the downloaded data files into lab9.jar. Submit the completed file to Canvas.