Hash Tables

In this lab, you will write a program to hash strings. You will implement the three different hash functions described below and one hash function of your choosing. Use separate chaining for collision resolution. You can make use of the STL in your implementation, e.g. list, vector. Place your HashTable class in a file named HashTable.cc.

Hash Functions

You need to implement these three hash functions that take string keys. For each function, calculate the value for the key and bring the result into the range of your hash table. You also need to provide your own hash function that takes a string as the key.

1. This hash function adds up the ASCII values of the characters in the key.
2. This hash function uses the first three characters of the key, therefore all keys are assumed to be at least three characters long. The value for a key *k* is calculated as: k[0] + 27 * k[1] + 729 * k[2] where 27 represents the number of letters in the English alphabet, plus the blank, and 729 is 27^2.
3. This hash function uses all characters in the key and calculates
   $\sum_{i=0,...,keysize-1} k[keysize - i - 1] * 37^{i}$. You can compute this polynomial function (of 37) by using Horner's rule, for example:
   $h = k0 + (37 * k1) + (37^2 * k2)$ can be computed recursively by $h = ((k2) * 37 + k1) * 37 + k0$.

Methods you need include in your HashTable class

- **void print( )** - Print the hash table in the following format. Each array location will be printed on a separate line. The line will start with the number of the array location followed by a colon (:) and followed by one tab. You will then print each word stored at that hash array location separated by a comma (,) and a single space. After all words at that hash array location are printed, you will output a return. Here is a sample of what your output should look like
- **void processFile(String filename)** - Add all of the words in the file specified as "filename" to the hash table. Here is a sample input file. Use push back to insert each of the words into the list at the hash location that the words hashes to.
- **void printStats( )** - Print the hash statistics: total number of collisions (keep track of this during insertions), the length of the longest list, and the average length of all lists.

Your main function should take the input file as a command line parameter, insert the values in the file into the hash table, print out the hash table to the screen, and finally print the stats.

For further testing, you can use the following dictionary input files: dict5.txt, dict7.txt.

Open Addressing Hashing

Write a program in a file named OpenAddressCollisionTests.java (file should have a main method), that randomly generates an input sequence of integer keys and performs collision resolution in an open addressing hashing scheme using the following probing sequences: linear probing, quadratic probing, and cubic probing. Cubic probing is simply using an offset of i^3 away from the original collision location, versus i for linear and i^2 for quadratic. Then run some tests varying table sizes and maximum number of elements inserted into the hash

table to see when the number of collisions dominates the cost of inserts. Use table sizes that are prime numbers and maximum number of elements inserted at 25%, 50%, 75%, and 90% of the size of the hash table.

Rubric

- 20 points - Attendance
- 60 points - 4 Hash functions and main test method
- 20 points - Open Hashing Collision Testing