Project DS7

Due Date: May 2

Purpose

This is it! Finally!

The purpose of this last project is to implement a hash table using a hash function that is a variation on one developed by Carter and Wegman and that incorporates closed hashing, implemented with a custom probing scheme, similar to linear probing.

Problem

Miquel Sprout, a professor at a prestigious New England college, analyzes old English manuscripts to determine authorship and other details. You, the owner of a software company called FunProgCo, are called in to write some software to help him find patterns in text. In order to do this, the number of distinct words in a manuscript is required. You decide that a hash table is the perfect way to store such words and frequencies.

Input

The program must first prompt the user for various information:

- the desired hash table size (integer)
- the value for A, the hash function constant (float)
- the name of the input file (string)

The input file will consist of plain, ASCII text containing a passage from a published book. This file is also "cleaned" of extraneous punctuation and capitalization; it contains **only** words, all in lower case; see the course web page for samples.

You must do some **error checking**. In particular, it could turn out that the table size is too small. This is logically possible because the user may experiment with different table sizes. You should also check that the input data file is valid. In either case, if there is an error, the program should display an error message and quit. Finally, there is the possibility of an infinite loop; see below for more information.

Output

The program should display the contents of the hash table, at most 20 **non-empty** rows at a time. The next 20 rows should be displayed when the user presses the `<enter>` key, as you've done in a previous project. If there are fewer than 20 rows left in the table, then all of the remaining rows should be displayed.
For each **non-empty** location in the table, the following should be displayed:

- the index in the table
- the word found at that index
- the number of times that word was found in the input text

- the number of collisions at that index. Each time a word w is hashed to a location i, a collision occurs if there exists a word x at that location such that $w \neq x$. If this happens, the collision count at location i should be incremented. Note that because closed hashing is used, one word may produce multiple collisions.

After the complete table is displayed, the total number of collisions should be shown as well as α - the load factor.

Assume a text file that contains the following 12 words:

```
eat i a
tea ate balm eat a
oh a blam to
```

A sample run might using the file above would look like the following:

```
Enter table size: 12
Enter value for A: 0.63
Enter file name: ds7input1.txt

Index  Word     Frequency  Collisions
-------------------------------------
    1  i                1           3
    2  blam             1           1
    3  a                3           0
    4  tea              1           1
    6  eat              2           1
    7  balm             1           0
    8  to               1           0
    9  ate              1           0
   11  oh               1           0

Total collisions: 6
Alpha: 0.75
```

The numeric values in the table should be right justified, while the strings should be left justified. The alpha value should be shown with two decimal places.

Specifics

You must implement a hash table that uses custom probing (below) with an accompanying hash function, following these guidelines:

- Use OOP principles in your program. The hash table should be defined in a class, along with its associated methods. *Only* items that are associated with the hash table should be included in the class. The program that *uses* the hash table should **not** be defined within the class. In other words, you should make a hash table object that contains data members for storing a string, the frequency count, and the number of collisions. Doing the stuff that is unique for Professor Sprout, such as reading a text input file, is not part of hash table; it just so happens that a hash table is used to help find the required solution. So reading the text file should not be part of the class.

The class should contain the following public methods **only**:

1. a constructor called `hashTable()`
2. an insert method: `void insert (string)`
3. a display method: `void display ()`

and at least one private method:

4. the hash function: `int hash (string)`

A sample `main()` would look like this:

```
int main () {
   hashTable myTable;

   readData (myTable);

   myTable.display();
   return 0;
}
```

- The hash function is defined as follows:

  Let K = key which is a word in the file.

  Let x_i denote a letter in K, from 1..n.

  Let T = the table size.

  Let A = floating point constant input by the user s.t. $0 < A < 1$.

$$h(K) = (h_1(x_1) + h_2(x_2) + h_1(x_3) + h_2(x_4)...h_j(x_n)) \ \% \ T$$

where $j = 1$ when n is odd, and $j = 2$ when n is even, and

$$h_i = \lfloor T \times (key \times A \ \% \ 1) \rfloor$$

when i is odd, and

$$h_i = \lfloor T \times (key \times \tfrac{1}{A} \ \% \ 1) \rfloor$$

when i is even.

- The hash table must store strings and associated frequencies. Initialize the table so that all of the frequency counts are 0. This should be done in the constructor.
- If there is a collision on the hash index i, then the item should be inserted into the next location $i + 2$. If that also results in a collision, then the next index should be $i + 4$, then $i+6$, etc. For example, if the initial index is 4 and there is a collision, the next index to check would be 6 $(i + 2)$. The next one after that would be 8 $(i + 4)$, then 10 $(i + 6)$, etc.
- Because the user will input the size of the hash table, you must assume that some collisions will result in wrap-around.
- Note that, depending on the table size, the hashing scheme could result in an infinite loop, even if the hash table is not full. You must check for this and stop the program with an error message that the item can not be stored.

Notes

- As always, comment your program.
- Be sure that your program reads the words from a text file properly before you move on to storing the words in the hash table.
- Implementing this program should not be too onerous, especially since you've had some practice in Lab. However, take time to create good test data and to run with several different input values. Make sure your program really runs correctly!
- Submit the code as usual through Canvas, as in `gousieDS7.zip` or `gousieDS7.cpp`, depending on if you use multiple files as in the former or just one as in the latter (this is a pretty short program). There is no hard copy to turn in.

I got distracted by learning.
– Amy Hopkinson '09, in a Theory of Computation class.