PROGRAMMING ASSIGNMENT #5
CS 2223 B-TERM 2019
CLOSED HASHING
&
DIJKSTRA'S ALGORITHM

SIXTY POINTS
DUE: FRIDAY, DECEMBER 6, 2019 6 PM

1. (5 Points) Read a file and hash the words:
   Read a text file (Raven.txt from Canvas) and hash the words using the suggested hash function from *Levitin*, page 269:

$$h \leftarrow 0; \textbf{ for } i \leftarrow 0 \textbf{ to } s-1 \textbf{ do } h \leftarrow (h * C + ord(c_i)) \mod m,$$

   where:

   h is the computed hash value,
   s is the length of the word being hashed,
   c_i is the i^{th} character of the word,
   $ord(c)$ is the numerical value of character c in the alphabet in use,
   C is a constant larger than every $ord(c_i)$, and
   m is the modulus defining the size of our hash table.

   We will take $ord(c)$ to be the ASCII value of the characters in each word. Thus, $ord(c) \in \{39, 65, 66, \ldots, 90, 97, 98, \ldots 122\}$ for $c \in \{', A, B, \ldots, Z, a, b, \ldots, z\}$, respectively[1]. All other characters are discarded; we define words as unbroken sequences (strings!) of consecutive alphabetic characters, both upper and lower case, plus apostrophes, in the lines read for the file. (In C++, the function **int**(letter) returns the ASCII value of character 'letter'.)

   For our "Raven" text file, we will take $C = 123$ and $m = 1000$. These represent parameters that can be changed to alter the shape and composition of our resulting hash table. For now, we will use these values for the sake of consistency. You are free to experiment with changing these parameters on your own, of course!

[1]That apostrophe will introduce a couple of artifacts given our text file. Working with 'real' data gets messy. See if you can find the artifacts, but consider them unique words.

2. (10 Points) Create a Hash Table using Closed Hashing (Open Addressing):

   a. Build a hash table of size 1000, i.e. entries from 000 to 999, from the hash values computed in Part 1 above. Load them into the table in the order they occur in the file, *discarding duplicates*. Our table will be far from full–you needn't worry about re-sizing it–we will examine the effects of its load factor in Part 3 below.

   b. Display the table, starting with table entry 0, in lines of the form:

      Hash Address, Hashed Word, Hash Value of Word

      (Remember, the Hash Address will not match the Hash Value when the resolution of a collision forces a word to be cascaded down the table. Also, remember that the table should be thought of as a circular list so that a word which cascades past the bottom end of the table gets wrapped to the top in looking for an open place in the table.)

      Note: Your program may perform Parts 1 & 2 simultaneously, if you wish.

3. (20 Points) Explore the Hash Table:

   Add to your code routines that answer these questions. Some answers may not be unique. You may resolve these issues by presenting either *any* correct answer or *all* correct answers.

   a. How many non-empty addresses are there in the table? What does that make the load factor, α, for our table?

   b. What is the longest empty area in the table, and where is it?

   c. What is the longest (largest) cluster in the table, and where is it?

   d. What hash address results from the greatest number of distinct words, and how many words have that address?

   e. What word is placed in the table farthest from its actual hash address, and how far away is it from its actual hash address?

4. (25 Points) Implement Dijkstra's Algorithm with weighted graphs like this:

Your code should read a text file and then ask for input from the console for start and destination nodes and use Dijkstra's algorithm to find the "length" of the shortest path between them. It should also display the sequence of nodes that constitute this shortest path.

You can assume the associated graph, read from a text file, is connected and that the input file is consistent. (The main diagonal will consist exclusively of zeroes, there will be no negative edges, etc.)

```
10
0   50  7   10  0   0   0   0   0   0
50  0   30  0   3   0   99  0   0   0
7   30  0   6   27  15  0   0   0   0
10  0   6   0   0   11  0   0   4   0
0   3   27  0   0   12  120 105 0   0
0   0   15  11  12  0   0   119 5   0
0   99  0   0   120 0   0   2   0   67
0   0   0   0   105 119 2   0   122 66
0   0   0   4   0   5   0   122 0   190
0   0   0   0   0   0   67  66  190 0
```