CS 2420 Section 2 and Section 3
Spring 2016
Assignment 5: Hash Tables
Due: 11:59 p.m. March 21 (Monday), 2016
Total Points: 100 points

Part I [15 points]: Submit your solution via Canvas.

Using a hash table T of size m = 11 (i.e., $T[0\ldots10]$) with hash function $hash(x) = x \% m$, show the hash table results after performing the following operations:

- Insert the following keys in the given order: 26 42 5 44 92 59 40 36 12.
- Delete the following keys in the given order: 38 12
- Search for the following keys: 44 50
- Insert key 29. (If the number of probes reaches m and it still cannot find an available slot to put the new item, you can stop and indicate that the hash table is too full to search for an available slot).

For each of the following approaches, show its resultant hash table.

(a) Linear probing, i.e., $h_i(x) = (hash(x) + i) \% m$, for $i = 0, 1, 2, \ldots, m\text{-}1$.
(b) Quadratic probing, i.e., $h_i(x) = (hash(x) + i^2) \% m$, for $i = 0, 1, 2, \ldots, m\text{-}1$.
(c) Double hashing using the secondary hash function $hash_2(x)=x\%9+1$, i.e., $h_i(x) = (hash(x)+i\times hash_2(x)) \% m$, for $i = 0, 1, 2, \ldots, m\text{-}1$. Note that this secondary hash function does not follow our discussion in class, but theoretically we can pick any function as the secondary hash function.

Please refer to **Ch6.HashTableIllustation.pdf** posted under "ClassNotes" folder for each expected step. In other words, you should clearly indicate **the probing locations for each inserted, deleted, or searched item together with the status information** (i.e., 0 represents "EMPTY", 1 represents "ACTIVE", and 2 represents "DELETED".)

Part II [85 points]:

You are required to develop a simple system to help the Postal Service to quickly calculate the total distance between two zip codes (e.g., the zip code of the sender and the zip code of the recipient).

The dataset you will use for this task is a CSV file (zipcodes.csv) with three columns, namely, zip code, latitude, and longitude. In total, there are 42522 lines. The first eight lines are shown as below:

```
00501,40.81,-73.04
00544,40.81,-73.04
00601,18.16,-66.72
00602,18.38,-67.18
00603,18.43,-67.15
00604,18.43,-67.15
00605,18.43,-67.15
00606,18.18,-66.98
```

You must create two hash table classes. One hash table class uses the separate chaining approach to resolve collisions, as discussed in class. The other hash table class uses the quadratic probing approach to resolve collisions, as discussed in class.

The separate chaining-based hash table contains the following public methods:

- **[5 points]** Insert: insert a new entry with ZIP code, latitude, and longitude into the hash table (assuming the new entry is not in the hash table). To achieve the constant time performance, the new entry should be put at the head of the linked list located by the hash function, as we discussed in class.
- **[5 points]** Remove: remove the entry with a given ZIP code from the hash table.
- **[5 points]** Search: determine whether there is an entry in the hash table whose ZIP code matches a given ZIP code. If yes, return "true"; otherwise return "false".
- **[5 points]** ComputeDistance: return the distance between two valid zip codes using the haversine formula defined in this assignment.
- **[5 points]** CountTableEntry: return the number of elements in a given index of the hash table
- **[5 points]** PrintTableEntry: print the elements in the hash table at each slot (index)

[5 points] Since each key (e.g., ZIP code) is a string, it must be prehashed so it can be represented as a nonnegative integer. To this end, you may go through the string a character at a time, convert each character to an integer (a.k.a., ASCII value), and add up the integer values to convert the key to a nonnegative integer. You can then use the division method (e.g., the converted integer modulo the size of the hash table) to get the index of the hash table. All this has to be implemented in a private function called Hash.

Below is the HashTableS.h file, which contains C++ HashTableS class interface for the separate chaining-based hash table, for your reference.

```
class HashTableS // separate chaining
{
   private:
            int m; // the size of the table
            Element ** T; // the hash table
            int Hash(string x);  // the hash function

  public:
            HashTableS(int size);
            void Insert(string x, double y, double z);  // insert a new element with key x and other info
            void Remove(string x);  // remove an element whose key is x
            bool Search(string x);   // search an element whose key is x and return true if found
            double ComputeDistance(string x1, string x2);   // return the distance between two zip codes.
                                                            // assume both zip codes are valid ones.
            int CountTableEntry(int i); // return the number of elements in the ith slot of the hash table
           void PrintTableEntry(); // print the elements in the hash table

};
```

[25 points] The quadratic probing-based hash table supports the same five operations, namely Insert, Remove, Search, ComputeDistance, and PrintTableEntry as in the separate chaining-based hash table. The difference is that when a collision happens, "probe" the next cell of the hash table based on the quadratic probing, which has to be implemented in a private function called QuadraticProbe. Here, as discussed in class, each element of the hash table is associated with a "flag" with 0, 1, and 2 representing "EMPTY", "ACTIVE", and "DELETED", respectively.

Below is the HashTableQ.h file, which contains C++ HashTableQ class interface for the quadratic probing-based hash table, for your reference.

```
class HashTableQ // quadratic probing
{
  private:
          int m; // the size of the table
          Element * T; // the hash table
          int Hash(string x);  // the hash function
          int QuadraticProbe(string x, int i);  // the quadratic probing function

  public:
          HashTableQ(int size);  // set the size of the table
          void Insert(string x, double y, double z);  // insert a new element with key x
          void Remove(string x);  // remove an element whose key is x
          bool Search(string x);   // search an element whose key is x and return true if found
          double ComputeDistance(string x1, string x2); // return the distance between two zip codes.
                                                        // assume both zip codes are valid ones.
         void PrintTableEntry(); // print the elements in the hash table
};
```

Write a driver program to do the following tasks (Make sure to use modularization so you do not have too many duplicated code in your main program):

1. **[10 points]** Demonstrate the correctness of all the public member functions in both classes using a reasonable number of data. You may refer to Ch6.HashTableIllustration.pdf posted under the "ClassNotes" folder to come up with your own testing data and operations. **The grader will use his own driver code to test the correctness of all the public member functions.**
2. Read zipcodes.csv file to know the total number of entries.
3. **[5 points]** Choose four possible sizes for the separate chaining-based hash table so the load factor, which is defined as the ratio of the number of elements to the size of the hash table, is close to 0.3, 0.4, 0.5, and 2, respectively. For example, if the number of elements is 20 and the load factor is 0.3, the size of the hash table is 66. *For simplicity, you do not need to choose a prime number as the size of the hash table*. For each hash table size, do the following task:
   - Print out the load factor and its corresponding hash table size
   - Create a hash table using the computed hash table size.
   - Read zipcodes.csv file and insert all the entries into the hash table using the ZIP codes as keys.

- Print out the average length of the linked lists (i.e., the average number of elements of the linked lists) using

$$len = \frac{\sum_{i=0}^{m-1} length(t_i)}{m}$$

where length(t_i) is the number of elements in the ith slot of the hash table and m is the size of the hash table.

- Read the zip code from "searchzip.txt" file line by line and perform the search operation. If the searched zip code is a valid one, save it to a new file called "validzip.txt". Measure the search time to find all valid zip codes stored in "searchzip.txt" file. (Note: You may need to name validzip.txt differently for different choices of the hash table. However, they should contain the same results if your program is correct.)
- Read the valid zip code from "validzip.txt" file line by line, compute the distance between all adjacent valid zip code pairs. Measure the time to compute the distance between all adjacent valid zip code pairs stored in "validzip.txt" file. It should be noted that the time to get the latitude and longitude information is included in this computation as well. The distance d must be calculated using the haversine formula.

$$d = R \times 2 \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Where R is the radius of the Earth (3961 miles or 6373 km), Φ1 and Φ2 are the latitudes of the points 1 and 2, respectively, and λ1 and λ2 are the longitudes of points 1 and 2, respectively.

4. **[5 points]** Choose three possible sizes for the quadratic probing-based hash table so the load factor is close to 0.3, 0.4, and 0.5, respectively. For each hash table size, do the following task:
   - Create a hash table using the computed hash table size.
   - Read zipcodes.csv file and insert all the entries into the hash table using the ZIP codes as keys.
   - Read the zip code from "searchzip.txt" file line by line and perform the search operation. If the searched zip code is a valid one, save it to a new file called "validzip.txt". Measure the search time to find all valid zip codes stored in "searchzip.txt" file.
   - Read the valid zip code from "validzip.txt" file line by line, compute the distance between all adjacent valid zip code pairs. Measure the time to compute the distance between all adjacent valid zip code pairs stored in "validzip.txt" file using the haversine formula.
5. **[5 points]** On the console/screen, write your summary on the following:
   - Comparison among four separate chaining-based hash tables in terms of the average length of the linked lists, the search time to find all valid zip codes stored in "searchzip.txt", and the time to compute the distance between all adjacent valid zip codes stored in "validzip.txt".
   - Comparison among three quadratic probing-based hash tables in terms of the search time to find all valid zip codes stored in "searchzip.txt" and the time to compute the distance between all adjacent valid zip codes stored in "validzip.txt".
   - Your final suggestions on the best hash table to be used for the problem.