Data Structures & Algorithms LAB – Fall 2015

(BS-SE-F14 Morning & Afternoon)

Lab # 12

Instructions:

- **Attempt the following tasks exactly in the given order.**
- Make sure that there are no ***dangling pointers*** or ***memory leaks*** in your programs.
- Indent your code properly.
- Use meaningful variable and function names. Follow the naming conventions.
- Use meaningful prompt lines/labels for all input/output.

Task # 1 *(Max Time: 20 Minutes)*

In this lab you are going to implement a class **`HashTable`** for storing names. The definition of your class should be as shown below:

```
class HashTable {
  private:
      string* table;          // Dynamic array of strings to hold names
      int S;                  // Total number of slots in the table
      int n;                  // Current number of elements present in the table

  public:
      HashTable (int size);
                  // Constructor to allocate and initialize an empty hash table of the specified size
      ~HashTable ();          // Destructor
      bool isEmpty ();        // Checks whether hash table is empty or not
      bool isFull ();         // Checks whether hash table is full or not
      double loadFactor ();   // Calculates & returns the load factor of the hash table (n/S)
};
```

Implement the five public member functions of the **`HashTable`** class shown and described above.

Task # 2 *(Max Time: 40 Minutes)*

In order to insert or search names in the hash table, you should use a hash function which adds up the ASCII values of all the characters in the given name and then takes the MOD of the resulting sum by **`S`** (which is the table size). Here is a function which takes a string as argument and returns the sum of the ASCII values of all the characters in that string:

```
int HashTable::getHashValue (string name)    // Private member function of HashTable class
{
      int temp = 0;
      for (int i=0; i < name.length(); i++)
      {
            temp = temp + name[i];
      }
      return temp;
}
```

If we call the above function on the word "asad" it will return 409 (i.e. 97('a') + 115 ('s') + 97 ('a') + 100('d')).

Now, you have to implement the following three member functions of the **`HashTable`** class:

`bool insert (string name)`

This function will use the above-mentioned hash function to determine the location at which "name" can be inserted in the hash table. If that location is already occupied (a collision) then this function should use **linear probing (with increment of 1)** to resolve that collision (i.e. it should look at the indices after that location, one by one, to search for an empty slot). During its working, this function should **display the sequence of indices that are traversed when inserting an element**. This function should return **true**, if eventually an empty slot is found and "name" is stored there. If no empty slot is found, then this function should return **false**.

`bool search (string name)`

This function will search for the given "name" in the hash table. It will accomplish this by using the above-mentioned hash function and linear probing. This function should also **display the sequence of indices that are traversed at the time of searching for an element**. If the name is found then this function should return **true**. Otherwise, it should return **false**.

`void display ()`

This function will display the contents of the hash table on screen, along with their indices. For indices which are empty, this function should display the word "**EMPTY**".

Also, write a menu-based driver function to illustrate the working of different functions of the **`HashTable`** class. The driver program should, first of all, ask the user to enter the size of the table. After that it should display the following menu to the user.

```
Enter the size of Hash Table: 11

1. Insert a name
2. Search for a name
3. Remove a name    (See Task # 3 below)
4. Display the Hash Table
5. Display Load Factor of the table
6. Exit

Enter your choice:
```

Task # 3 *(Max Time: 20 Minutes)*

Add the following public member function to the **`HashTable`** class:

`bool remove (string name)`

This function will try to remove the given "name" from the hash table. This function should return **true**, if the name is found and removed. And it should return **false** if the given name is not found in the table. As discussed in class, you should make sure that the search function still works properly after the remove function has executed.

Task # 4 *(Max Time: 30 Minutes)*

Suppose that integers in the range 1 through 100 are to be stored in a hash table using the hash function $h(x) = x\%S$, where S is the table's size (capacity). Write a program that generates random integers in this range (1 to 100) and inserts them into the hash table until a collision occurs. The program should carry out this experiment 50 times and calculate the **average number of integers that can be inserted into the hash table before a collision occurs**. Run the program with various values of S (10, 20, 30, …100), and tabulate the results in MS Excel.

Note: You can use the following two functions to generate random numbers in a given range. Use the function **`initialize()`** to initialize the random number generator (you will need to call this function only once at the start of your program). The function **`getRandomNumber (int start, int end)`** can be used to get a random number in the range from **`start`** to **`end`** (both inclusive).

```
#include <ctime>
void initialize ()
{
      srand ( time(NULL) );
}
int getRandomNumber (int start, int end)
{
      return ( rand() % (end-start+1) ) + start;
}
```