Data Structures and Algorithms Lab

Lab 13 **Marks 05**

Instructions

Work on this lab individually. You can use your books, notes, handouts etc. but you are not allowed to borrow anything from your peer student.

Marking Criteria

Show your work to the instructor before leaving the lab to get some or full credit.

What you must do

Implement a class **HashTable** for storing names.

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

public:
	HashTable (int size);		/**Constructor, store "EMPTY" to indicate free
					   location in the HashTable*/
	~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 (curSize/size)*/
};
```

Note that the constructor of the **HashTable** class will allocate the array (table) such that it can contain up to **size** names.

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 **size** (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 value(string name)
{
	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')).

You are required to implement the following 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). 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. If the name is found, then this function should return **true**. Otherwise, it should return **false**.

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.

`void display ()`

This function will **display** the contents of the hash table on screen, **index by index**. 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, ask the user to enter the **size** of the table. After that it should display the menu to the user.

```
Enter the size of the hash table: 11

1. Insert a name
2. Search for a name
3. Remove a name
4. Display the table
5. Display the load factor of the table
6. Exit

Enter your choice:
```

☺ ☺ ☺ **BEST OF LUCK** ☺ ☺ ☺