Lab 10
Data Structures
BS DS Fall 2024 Morning/Afternoon

Task 01: Hash Table Implementation

In this task, you are going to implement a class **HashTable** for storing names. The class declaration should be as shown below:

```
class HashTable {
private:
   string* table;
   int S; // Table size
   int n; // Current number of elements

public:
   HashTable(int size);
   ~HashTable();
   bool isEmpty(); // Checks whether the hash table is empty or not
   bool isFull();   // Checks whether the hash table is full or not
   double loadFactor(); // Calculates & returns the load factor of the hash table
(n/S)
   int getHashValue(string name);
   bool insert(string name);
   bool search(string name);
   bool remove(string name);
   void display();
};
```

Implement Member Functions

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). Following is the implementation of a helper function (**getHashValue**) which takes a string as argument and returns the sum of the ASCII values of all the characters in that string:

```
int getHashValue(string name);
```

If we call the above function on the string **"ali"** it will return 310 (i.e. 97(**'a'**) + 108 (**'l'**) + 105 (**'i'**)). Now, implement the following 4-member functions of the **HashTable** class:

```
bool insert(string name);
bool search(string name);
bool remove(string name);
void display();
```

i. insert (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 at the slot. If no empty slot is found, then this function should return **false**.

ii. search (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**.

iii. 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**”.

iv. 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

Also, write a menu-based driver function to illustrate the working of various 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
4. Display the Hash Table
5. Display Load Factor of the table
6. Exit

Enter your choice:
```

Task 02

For given list of fruits (file attached), show at least two different workings for finding unique Hash Values. For 86 fruits, I have found a way that requires array size 973, therefore if you find some way that requires array size less than 973, you will get bonus marks, however, for array size close to 1000 will be appreciated for full marks, otherwise we will appreciate your unique effort and give you reasonable marks but if we find similar (not same, for same it will be considered as copy case) way, we will reduce marks of both (or multiple) students.

Task 03

Suppose that integers in the range 1 through 100 are to be stored in a hash table using the hash function $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{x} \% \boldsymbol{S}$, where $\boldsymbol{S}$ 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 $\boldsymbol{S}$ (10, 20, 30, …100), and tabulate the results in MS Excel.

You are required to write a function **experiment** that creates a HashTable of the given size as parameter, inserts random integers into it until a collision occurs, and then returns the total number of integers successfully inserted before the collision.

```
int experiment(int tableSize) {
//implementation of the following function.
}
```

The following function is used to generate random numbers within a specified range.

Note: The function call **getRandomNumber (s,e)**can be used to get a random number in the range from **s** to **e**(both inclusive).

```
int getRandomNumber(int start, int end) {
   return rand() % (end - start + 1) + start;
}
```

Use following drive program to test your experiment function.

```
Int main() {
   srand(static_cast<unsigned>(time(0)));  // seed random generator

   const int numExperiments = 50;
   cout << "Table Size\tAverage Inserted\n";
   cout << "-----------\t----------------\n";

   // Run experiments for table sizes 10, 20, 30, ..., 100
   for (int S = 10; S <= 100; S += 10) {
      double totalInserted = 0.0;

      for (int i = 0; i < numExperiments; i++)
         totalInserted += experiment(S);

      double averageInserted = totalInserted / numExperiments;

      cout << setw(5) << S << "\t\t" << fixed << setprecision(2)
          << averageInserted << endl;
   }
```