**Part 1:** Describe briefly your design of the HashTable class and how you implement the collision resolution strategies (e.g., how to decide when to stop probing).

Firstly, I declared 5 helper methods in the HashTable.h file. These helper methods are:

**1-** int hash2(int itemToBeAdded): This method is our second hash function and it reverses to the taken parameter. (For example, hash(123456) = 654321)

**2-** int probe(int& addr, const int item, const int search, bool& isFound): This method calculates the probe count and returns it. While the function is doing that, it updates the item’s address if the item is found. If the item is not found, it updates isFound parameter as false and breaks while loop.

**3-** int getNumber(const string line): This method reads the number in the given .txt file. To Illustrate, “S 24” instruction in the given .txt file, this function returns 24.

**4-** int getNoOfProbes(): This function calculates the total number of probes in the given HashTable.

**5-** int isFull(): This function checks whether the HashTable is full or not.

**6-** void readFile( const string fileName ): This function reads the given .txt file and converts the lines to instructions.

Secondly, used data members of HashTable implementation:

**1-** int size : Holds the size of the HashTable.

**2-** int currentSize : Holds the current size of the HashTable.

**3-** int\* items : 1-Dimensional integer array that keeps the items.

**4-** CollisionStrategy strategy : Holds the type of collision strategy.

Thirdly, the required functions in HashTable:

**1-** HashTable( const int tableSize, const CollisionStrategy option ): This constructor takes the size of the table and the collision strategy as inputs. In my implementation, **-1** means **empty**, **-2** means **removed** otherwise it is **occupied.** It creates a dynamically allocated array for items and assigns every index’s item [0 to n-1] to -1 which means every index is empty.

**2-** ~HashTable(): This destructor deletes the dynamically allocated array to avoid memory leaks in the program.

**3-** bool insert( const int item ): This function inserts the given item to the HashTable if the HashTable is not full. This function uses the search( const int item, int& numProbes ) function to check the given item whether is in the HashTable or not. If the search function returns false or the HashTable is full, the function returns false. Otherwise, we insert the given item to the address, that was calculated by probe function, and the function increments the current size by 1 and returns true.

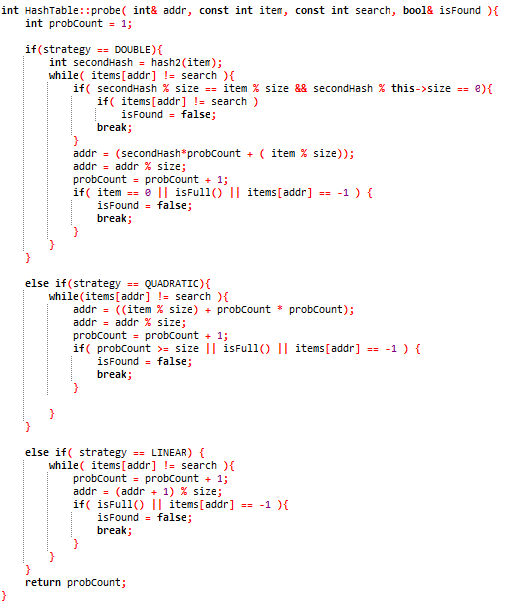
**4-** bool remove( const int item ): This function uses the search function to determine whether the item exists in the HashTable or not. If it exists then it uses the probe function to find its address. After finding the address of the given item, it removes it and assigns **-2** (removed) to items[address].

**5-** bool search( const int item, int& numProbes ): This function checks the given item exists in the HashTable or not. While doing that, it calculates the required number of probes to reach that item.

**6-** void display(): Displays the HashTable in the required format.

**7-** void analyze( int& numSuccProbes,int& numUnsuccProbes ): This function updates its parameters as the total successful and unsuccessful probe count. Also, this function **prints** the average successful and unsuccessful probe count in **double** form by dividing the total (un)successful probe count by the size of the table.

**To avoid infinite loops,** I used the probe function. Implementation of the probe function is given in below:



**Figure 1:** Implementation of the probe function

If the collision strategy is Double, the first if statement becomes true, and the program goes in it. Firstly, it checks the index [item % size], if it is not equal to the input item, I used my second hash function and it updates the address as desired. There are **4** **stopping conditions** of this if statement:

**1-** (Second Hash % size == item % size and Second hash function % size == 0). This means Double Hashing will not be functional therefore it breaks the while-loop.

**2-** If the item is equal to 0, we can not use the second function and there will be an infinite loop. Therefore, we should break the while-loop to avoid that.

**3-** If the HashTable is full, it means that there is no empty location, therefore we do not need to make more probes.

**4-** If index[address] is equal to -1 (empty), this means this location is empty, therefore there is no need to make more probes to find an empty location.

If the collision strategy is Quadratic, the second if statement becomes true, and the program goes in it. Firstly, it checks the index [item % size], if it is not equal to the input item, it starts to apply Quadratic Hashing. There are **3 stopping conditions** of this if statement:

**1-** If the probe count is greater or equal to the size of the HashTable, it means we checked every possible index for quadratic hashing. Therefore, we do not need to make more probes.

**2-**If the HashTable is full, it means that there is no empty place, therefore there is no need to make more probes.

**3-** If index[address] is equal to -1 (empty), this means this location is empty, therefore there is no need to make more probes to find an empty location.

If the collision strategy is Linear, the third if statement becomes true, and the program goes in it. Firstly, it checks the index [item % size], if it is not equal to the input item, it starts to apply Linear Hashing. There are **2 stopping conditions** of this if statements:

**1-**If the HashTable is full, it means that there is no empty place, therefore there is no need to make more probes.

**2-** If index[address] is equal to -1 (empty), this means this location is empty, therefore there is no need to make more probes to find an empty location.

**Part 2:**

It is better to make HashTable size a prime number. Therefore,

Size of the HashTable: 11

**Content of the input.txt file**:

S 7

S 12

D 2

I 12

S 12

D 12

I 12

I 24

I 1

I 23

I 39

I 46

S 46

I 6

D 24

S 24

I 24

S 12

S 24

S 1

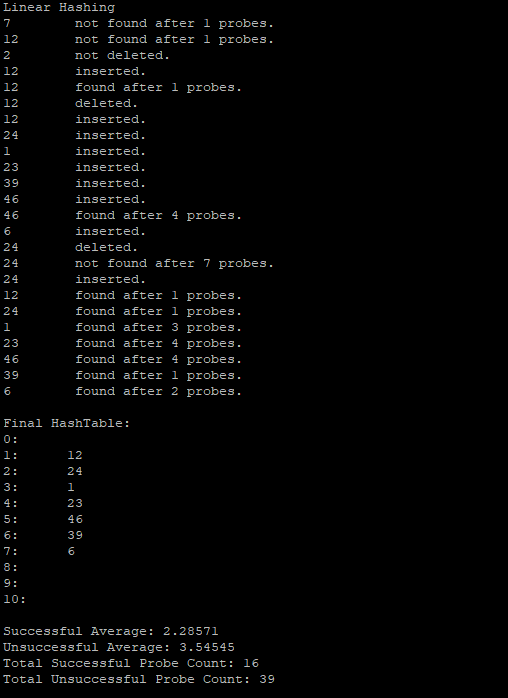
S 23

S 46

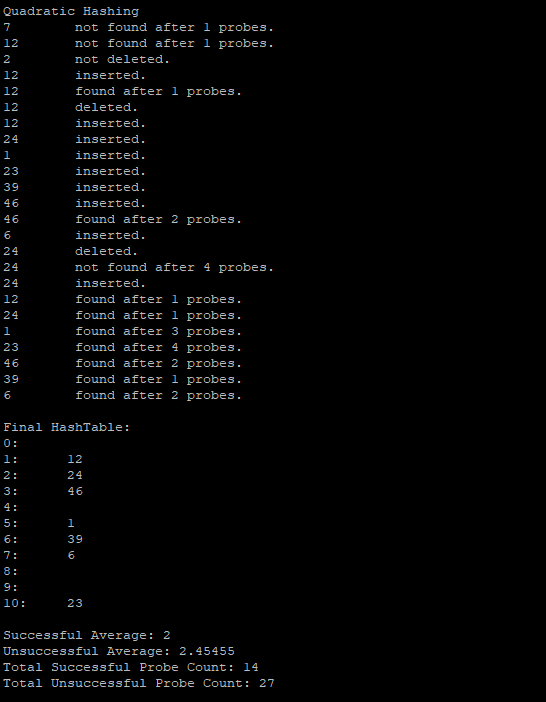
S 39

S 6

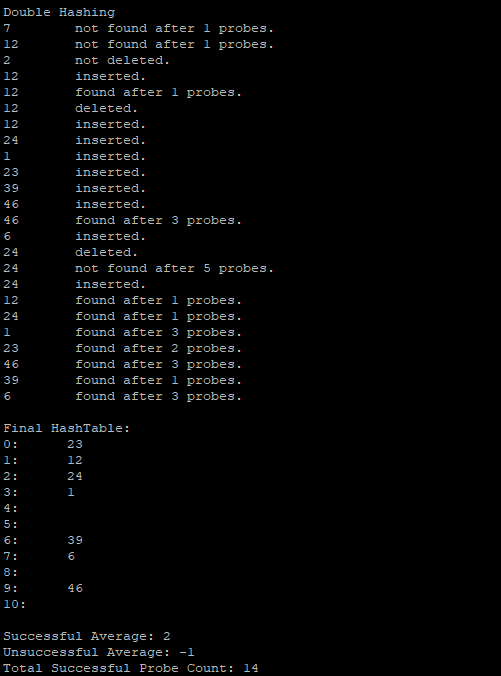
**Output of the program:**

****

**Figure 2:** Output of the Linear Hashing

****

**Figure 3:** Output of the Quadratic Hashing



**Figure 4:** Output of the Double Hashing

**Part 3:**

**Linear Hashing:**

Load factor α = =

For linear hashing, theoretical average probes are:

In my case, these values are:

Experimental results in my program:



**Figure 5:** Experimental Average of Linear Hashing

**Quadratic Hashing:**

For Quadratic Hashing, theoretical average probes are:

In my case, these values are:

Experimental results in my program:



**Figure 6:** Experimental Average of Quadratic Hashing

**Double Hashing:**

For Double Hashing, theoretical average probe is:

In my case, this value is:

Experimental results in my program:



**Figure 7:** Experimental Average of Double Hashing

**Discussion:**

For Linear Hashing, the experimental results are close enough to the theoretical results. For a successful search, the experimental result is greater than the theoretical average successful search. On the other hand, for an unsuccessful search, the experimental result is less than the theoretical average unsuccessful search.

For Quadratic Hashing, the experimental results are close to the theoretical results. It has some similarities with Linear Hashing, but it also has some dissimilarities with Linear Hashing. Their similarity is that in both hashing average successful probe count is greater than the theoretical average, both hashing average unsuccessful probe count is less than the theoretical unsuccessful average probe count. Their dissimilarity is that Quadratic Hashing’s total average (Successful Average + Unsuccessful Average) is considerably less than Linear Hashing’s total average. In Linear Hashing, my example input.txt file caused clustering, however, in Quadratic Hashing, the input.txt file caused less clustering. This could be the reason why Quadratic Hashing’s total average is less than Linear Hashing’s total average.

For Double Hashing, we can say that the experimental results are close to the theoretical results. For Quadratic Hashing and Double Hashing, their theoretical successful average probe counts are the same. Their successful average probe counts are also the same in experimental results.