PART I:

In this homework, we use **open addressing** strategy which is one of the collision resolution strategy for hash tables. In our implementation we use **linear probing, quadratic probing, and double hashing** to resolve the collusion in the hash table. In my HashTable class I have **HashTable constructor** that requires size and collision strategy as a parameter, **HashTable destructor** that destruct the array and the items in the array if there are some existence, **insert function** that insert the item in the corresponding index if the operation is successful returns true otherwise false, **remove function** that search for that given item and if the method finds the item then remove it if the operation is successful returns true otherwise false, **search function** that searches the array for the given item and returns number of probes while searching the HashTable if the operation is successful returns true otherwise false, **display function** that displays the elements of the HashTable, **analyze function** that analyses the HashTable by using successful and unsuccessful searches, also, it returns the average number of probes for each cases. These functions are defined as public in HashTable class.

```
public:
    HashTable(const int tableSize, const CollisionStrategy option);
    ~HashTable();

    bool insert(const int item);
    bool remove(const int item);
    bool search(const int item, int& numProbes);
    void display();
    void analyze(double& numSuccProbes, double& numUnsuccProbes);
```

Also, in my HashTable class I have other functions that are defined as private. **hash function** takes the item and calculate the first index according to $hash(i) = i \ mod \ tableSize$, **'function' function** calculate the next index if it cannot find the correct index previous step, **secondHash function** reverse the given number and use it for double hashing, **loopChecker function** checks whether there is a loop or not (for linear probing and double hashing it checks (numProbes == tableSize), for quadratic probing it checks (numProbes == (tableSize/2 + 1))). Finally, I have private variables; these are **tableSize** that holds the whole table size, **currentSize** that holds the number of items in the HashTable currently, **hashTable** which is an array of TableItem elements' pointers, and **option** that holds the collision strategy of the table.

```
private:
    int hash(const int item);
    int function(const int item, const int prob);
    int secondHash(const int item);
    bool loopChecker(const int numProbes);

    int tableSize;
    int currentSize;
    TableItem** hashTable;
    CollisionStrategy option;
```

```
class TableItem{
public:
    TableItem(int _item = 0){
        item = _item;
        tombStone = false;
    }
private:
    int item;
    bool tombStone;
    friend class HashTable;
};
```

In order to resolve the problem that occurs in the search, insert and remove functions because when we remove an element from the HashTable we erase the item in the corresponding slot of the array however removing can cause premature and incorrect stopping cases for other functions. Therefore, I create an object called TableItem that holds the integer variable **item** and the Boolean variable named **tombStone** that holds 1 if the item deleted and 0 otherwise. So that, I resolve the problem that emerges because of the remove function. For the loop problem that can be occur in the hash table I write a function called loopChecker and it checks the loops for each collision strategies (I mentioned above in this document before).

In my implementation I just consider different operations in my 'function' and loopChecker function because other operations are the same for each of the collision strategies.

Pseudocodes:

Insertion (item):

If the item is not already on the table and table is not full

index ← hash(item)

numProbes ← 0

While finding the location which is NULL or deleted until there is no loop

Increase numProbes

Index ← (hash(item) + function(item, numProbes)) mod tableSize

If there is no loop

Insert the item to the calculated index

Increase the current size

If the insertion completed return true, otherwise false

Remove (item):

If table size is not zero

index ← hash(item)

numProbes ← 0

While finding the item until confronting with NULL or loop

Increase numProbes

Index ← (hash(item) + function(item, numProbes)) mod tableSize

If found

Remove the item to the calculated index (Set object as deleted)

Decrease the current size

If the remove operation completed return true, otherwise false

Search (item, numProbes):

If the item is not already on the table and table is not full

index ← hash(item)

numProbes ← 0

While finding the item until confronting with NULL or loop

Increase numProbes

Index ← (hash(item) + function(item, numProbes)) mod tableSize

If there is no loop

Insert the item to the calculated index

Increase the current size

If the search operation completed return true, otherwise false

Analyze (avNumSuccProbes, avNumUnsuccProbes):

//SUCCESSFUL SEARCH

Collect the items which are in the table already

For each of the items call search function

avNumSuccProbes ← avNumSuccProbes + number of probes

avNumSuccProbes ← avNumSuccProbes / current size

//UNSUCCESSFUL SEARCH

If the collision strategy is not double hashing

Call search method for every calculated index value

//except for items which are already in the table

avNumUnsuccProbes ← avNumUnsuccProbes + number of probes

avNumUnsuccProbes ← avNumUnsuccProbes / table size

Else

avNumUnsuccProbes ← -1

PART II:

Linear:

```
I 0
I 1
I 2
I 6
I 7
I 8
I 12
I 0
I 1
I 2
I 6
I 7
I 8
I 12
I 13
I 14
I 15
R 0
R 1
R 2
I 26
I 27
I 28
S 0
S 1
S 2
R 9
S 9
```

```
0 inserted
1 inserted
2 inserted
6 inserted
7 inserted
8 inserted
12 inserted
0 not inserted
1 not inserted
2 not inserted
6 not inserted
7 not inserted
8 not inserted
12 not inserted
13 inserted
14 inserted
15 inserted
0 removed
1 removed
2 removed
26 inserted
27 inserted
28 inserted
0 not found after 10 probes
1 not found after 9 probes
2 not found after 8 probes
9 not removed
9 not found after 1 probes
========================================
0: 26
1: 27
2: 28
3: 13
4: 14
5: 15
6: 6
7: 7
8: 8
9:
10:
11:
12: 12
========================================
Average Successful Search: 1.9
Average Unsuccessful Search: 5.23077
```

Quadratic:

```
I 0
I 1
I 2
I 13
I 14
I 15
I 0
I 1
I 2
R 0
R 1
R 2
S 0
S 1
S 2
I 26
I 27
I 28
I 12
I 24
S 12
S 24
```

```
0 inserted
1 inserted
2 inserted
13 inserted
14 inserted
15 inserted
0 not inserted
1 not inserted
2 not inserted
0 removed
1 removed
2 removed
0 not found after 4 probes
1 not found after 4 probes
2 not found after 3 probes
26 inserted
27 inserted
28 inserted
12 inserted
24 inserted
12 found after 1 probes
24 found after 1 probes
========================================
0: 26
1: 27
2: 28
3: 15
4: 13
5: 14
6:
7:
8:
9:
10:
11: 24
12: 12
========================================
Average Successful Search: 1.625
Average Unsuccessful Search: 2.46154
```

Double:

```
I 0
I 1
I 2
I 130
I 131
I 132
I 0
I 1
I 2
I 12
I 142
S 142
R 0
R 1
R 2
I 260
I 390
S 390
S 260
```

```
0 inserted
1 inserted
2 inserted
130 inserted
131 inserted
132 inserted
0 not inserted
1 not inserted
2 not inserted
12 inserted
142 inserted
142 found after 2 probes
0 removed
1 removed
2 removed
260 inserted
390 inserted
390 found after 2 probes
260 found after 1 probes
========================================
0: 260
1:
2: 390
3: 131
4:
5: 130
6: 142
7: 12
8:
9:
10:
11:
12: 132
========================================
Average Successful Search: 2
Average Unsuccessful Search: -1
```

For all those test inputs and outputs, I use hash tables with 13 size. In my test files I try to test the cases that are; inserting same item, removing item that is not defined, inserting an item to the location that holds deleted property, searching items that are whether defined or not defined and so on. Also, I test the number of probes according to these tests.

PART III:

For linear probing in the given example table, I found average successful search as 1.9 and average unsuccessful search as 5.23 approximately. Also, for this hash table the load factor α is 10/13 ≅ 0.769. Therefore, theoretical average number of successful probes is $\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right] \cong 2.665$; and theoretical average number of successful probes is $\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right] \cong 9.87$. Here the load factor value measures how full a hash table is and a hash table should not be too loaded and uniformly distributed if we want to get better performance. However, for given example we have a bit loaded table but it is also nearly uniformly distributed. Therefore, although, we know that to get average performance table should not be too loaded, because table is generally uniformly distributed and does not contain loop there is a difference between theoretical and empirical results.

For quadratic probing in the given example table, I found average successful search as 1.625 and average unsuccessful search as 2.46 approximately. Also, for this hash table the load factor α is 8/13 ≅ 0.615. Therefore, theoretical average number of successful probes is $\frac{-\ln(1-\alpha)}{\alpha} \cong 1.55$; and theoretical average number of successful probes is $\frac{1}{1-\alpha} \cong 2.6$. Again, we know that to get close to the better performance we need to less loaded and uniformly distributed table. In the given table for quadratic probing we have close to average load factor, and our items is nearly uniformly distributed so that we have similar analyze outputs for both theoretical and empirical rates.

For double hashing in the given example table, I found average successful search as 2 approximately. Also, for this hash table the load factor α is 7/13 ≅ 0.538. Therefore, theoretical average number of successful probes is $\frac{-\ln(1-\alpha)}{\alpha} \cong 1.435$. Again, we know that to get close to the better performance we need to less loaded and uniformly distributed table. In the given table for double hashing we have too close load factor value to the 50% however in our table because we have a bit more same items that corresponds to occupied spaces rather than other examples, we have greater empirical value than theoretical one.

As a result, firstly, we could see that in linear probing although the table is close to the average one, we have a big difference for this table rather than other tables. Secondly, we could see that when we close to the average load rate of the table and average distribution, we get closer and closer to the theoretical value.