```
/**
* Author: İlknur Baş
* ID: 21601847
* Section: 2
* Assignment: 4
*/
```

Part 1

There are 4 txt files in my homework folder. The load factor of a hash table is different in each txt file. The outputs for each txt file can be seen by uncommenting the specified line in the main.cpp. Main.cpp is displaying the results for each collision that are given in the assignment. I have preferred to use automatic array in my hash table implementation. When HashTable is called, array is created in order to do the operations that are asked. Also, in my implementation, 0's indicate empty index, -1's indicate deleted items.

Insert function algorithms
Inside the function, I have been checking for each collision strategy whether the item is already located.
In linear collision strategy, when we are looking for an empty spot to insert, it will stop either when we find an empty spot or the hash table is full. I have added the bold code that can be seen in the following lines which indicates that we come back to the first index that we have looked hence the hash is full.

```
int index=hash(item);
int index2=hash(item);
while( arr_1[index]!=0 && arr_1[index]!=-1 ) { //keep looking to find empty spot
        index = (index2 + i) %tablesize;
        i++;
        if (index ==index2){
                a=false;
                break;
        }
}
```

In quadratic collision strategy, I have noticed that in some cases even if the hash table is not full, infinite loops may occur. Assume, the hash table is like the following: 0 1 2 3 0.(0's indicates that indexes are empty.) And we want to insert 7. 7 cannot be inserted even if there is an empty spot in hash table when we have used quadratic collision strategy. The bold code in the following lines avoids infinite loops. Also, we can avoid infinite loops with comparing table size and i value. It is the same logic with coming back to the first index that we have looked.

```
while( arr_1[index]!=0 && arr_1[index]!=-1 ) { //keep looking to find empty spot
        index = (index2 + (i*i)) %tablesize;
        i++;
        if (index ==index2){ // i == tablesize
                a=false;
                break;
        }
}
```

Double collision strategy is same with linear collision strategy when we are dealing how to avoid the infinite loops. Assume all indexes are occupied, it will check for empty spot until we reach the first index that we have looked. Also while writing the algorithm I have noticed that, we can avoid infinite loops with comparing table size and i value. Also, the hash2(item) having the value of 0 must be eliminated otherwise, when we are finding unoccupied place to insert, the index will always the same and will reduce the efficiency.

```
int index=hash(item);
int index2=hash(item);
int i=1;
while( arr_1[index]!=0 && arr_1[index]!=-1 && hash2(item)!=0) {
        index = (index2 + (i*hash2(item)) ) %tablesize;
        i++;
        if (index ==index2){ //or table size == i
                a=false;
                break;
        }
}
```

Remove function algorithms

In my hash table, for all collision strategies, 0 represents empty spots, -1 represents deleted spots which are empty. When the *bool* ***remove****(const int item)* function is called, I have implemented a new function in order to find the index of the deleted item which is *int* ***searchForFindingIndex (const int item).*** This will return-2 when the deleted item is not found otherwise, we will change the value of the deleted item to -1. (in order to prevent the confusion when using search operation.) The following code represents how the remove function works.

```
int index = searchForFindingIndex( item );
if (index == -2) {
        cout <<item<<" not removed. There is no such item to be removed"<<endl;
        a=false;
}else{
        arr_1[index] = -1;
        a=true;
        cout <<item <<" removed. after remove" <<endl;
        display();
}
```

Search function algorithms

When ***search****(const int item, int& numProbes)* function is called, the unsuccessful search will occur by either encountering with an empty spot (which in my case empty spots' values are 0) or coming back to the beginning index that we have searched. The following code piece represents how the search function works. For each collision strategy the assignment of variable *index*(can be seen below code piece) is different. Also, in my implementation, first checking whether the index is empty or not and then, move on the next index. In double probing, hash2(item) having the value of 0 must be eliminated and the added code can be seen inside the cpp, I have not added to the report.

```
int index=hash(item);
int index2=hash(item);
int i=1;
numProbes=1;

if( arr_1[index]!=0) {
        while ( arr_1[index]!=item ) {
                index = (index2 + i) %tablesize;
                i++;
                numProbes++;
                if (arr_1[index]==0 || index ==index2 ){
                        a=false;
                        break;
                }
        }
}else{
        a=false;
}
```

For the simplicity, ***void analyzeTheoretical()*** function analyzes the theoretical values of average number of probes for successful and unsuccessful searches for the specific collision strategy.

The ***void analyze(double& numSuccProbes, double& numUnsuccProbes)*** function, search for the existing values in the hash table and calculates the successful probes according to that. When calculating the average number of probes for unsuccessful search, I have counted the number of probes starting from an index to an empty spot, and I do this for each index. In some cases when calculating the average number of probes for unsuccessful search, infinity loops can be occur. I have also added a constraint in order to prevent it, the calculation will stop when we can back to the first index that we had looked.

Part 2

The data tables are given as follows. Note that, in the data table where load factor is 0.96, the theoretical result of quadratic and double hashing must be same. However, in my case, in quadratic probing the occupied size is 29 not 30. Insertion of an item cannot be done, because of the reason that I have explained under the insert function algorithms/ quadratic probing title.(In data table, it is indicated with red color.)

Linear when table size is 31

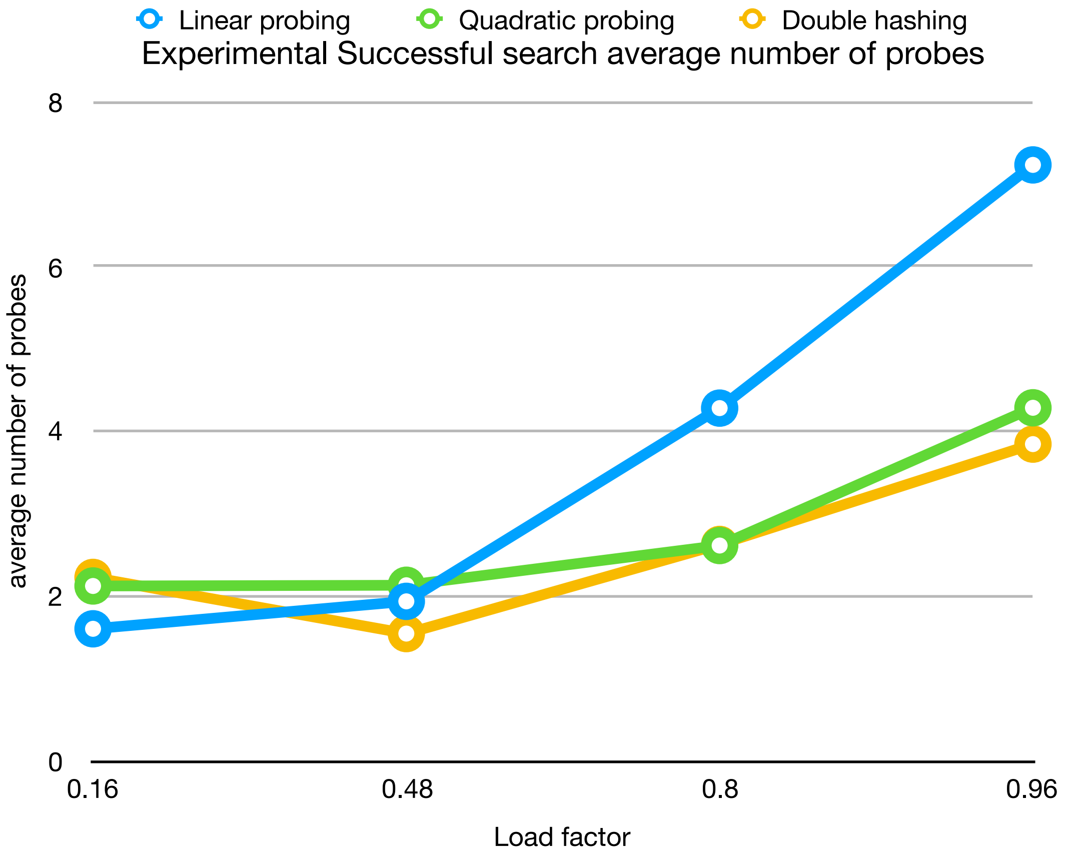
Table Size = 31	Number of index that are occupied : 5	Number of index that are occupied : 15	Number of index that are occupied : 25	Number of index that are occupied : 30
	a=0.16129	a=0.483871	a= 0.806452	a=0.967742
Experimental Successful search average number of probes	1.6	1.93333	4.28	7.23333
Experimental Unsuccessful search average number of probes	1.90323	2.93548	15.0323	16
Theoretical Successful search average number of probes	1.09615	1.46875	3.08333	16
Theoretical Unsuccessful search average number of probes	1.2108	2.37695	13.8472	481

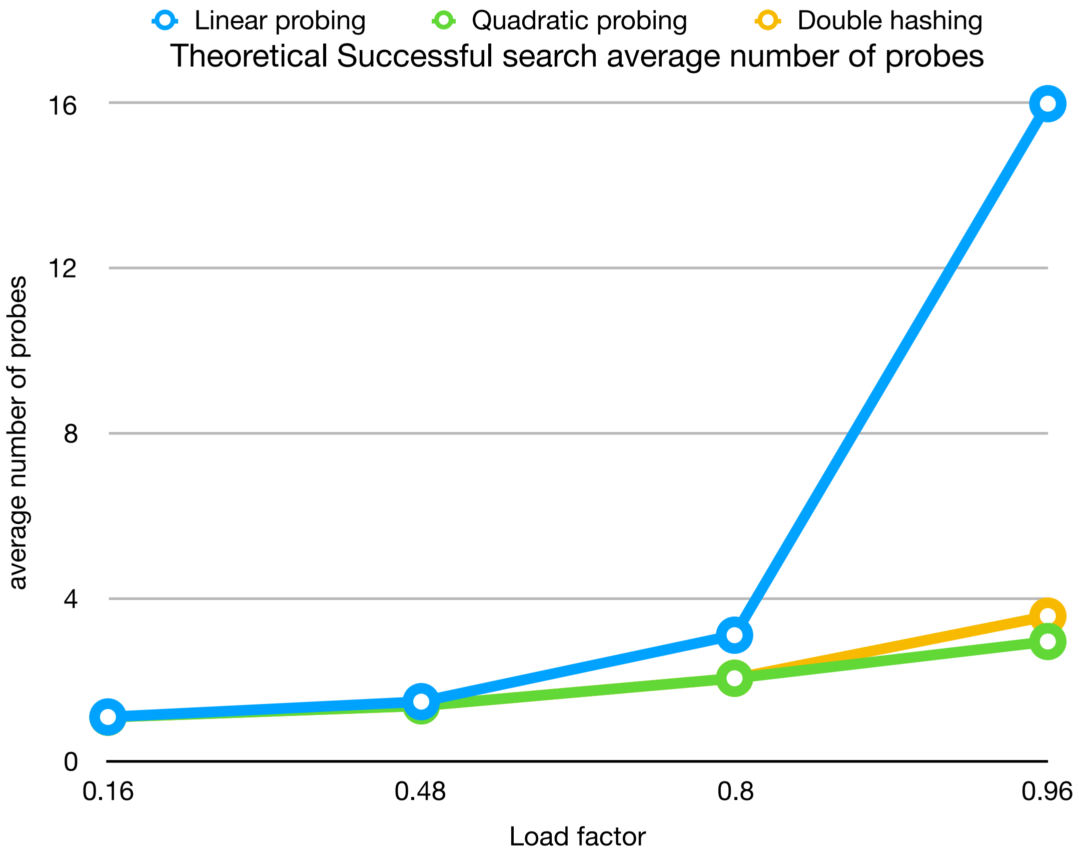
Quadratic when table size is 31

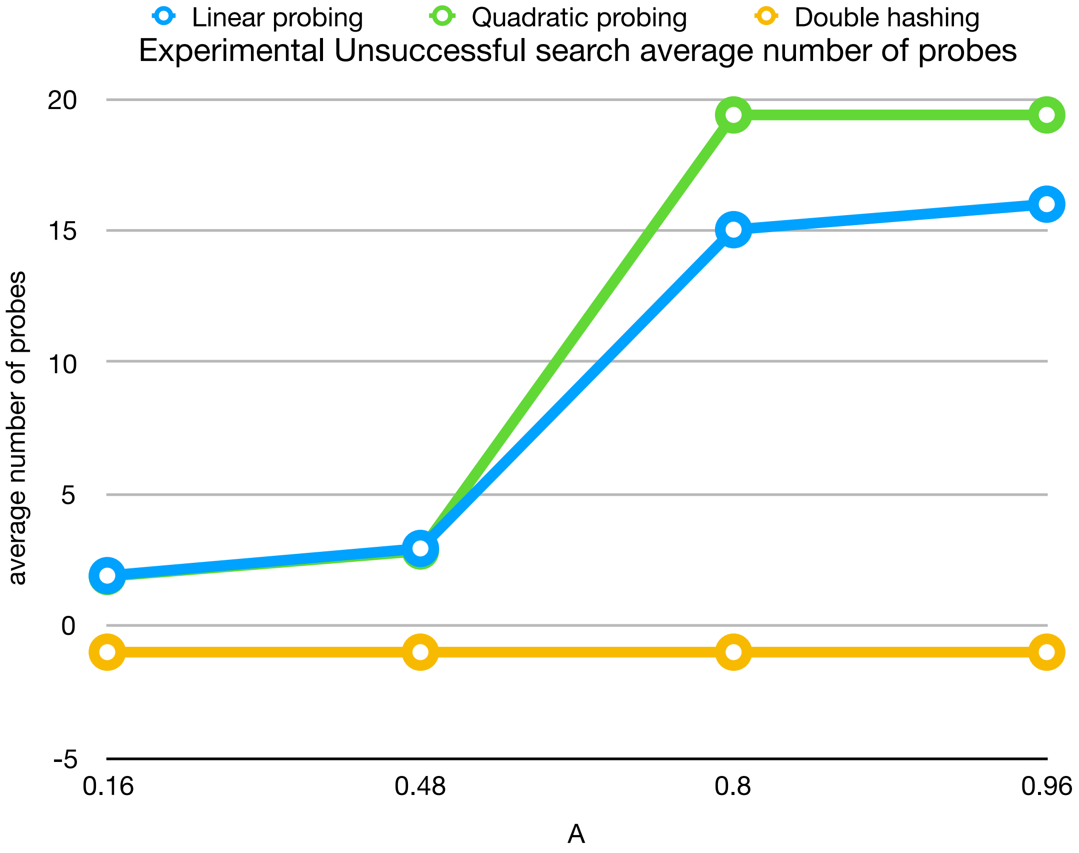
Table Size = 31	Number of index that are occupied : 5	Number of index that are occupied : 15	Number of index that are occupied : 25	Number of index that are occupied : 30
	a=0.16129	a=0.483871	a= 0.806452	a=0.967742
Experimental Successful search average number of probes	2.12	2,12889	2.6112	4.28391
Experimental Unsuccessful search average number of probes	1.87097	2.83871	19.3871	19.3871
Theoretical Successful search average number of probes	1.09052	1.36689	2.03636	2.92986
Theoretical Unsuccessful search average number of probes	1.19231	1.9375	5.16667	15.5

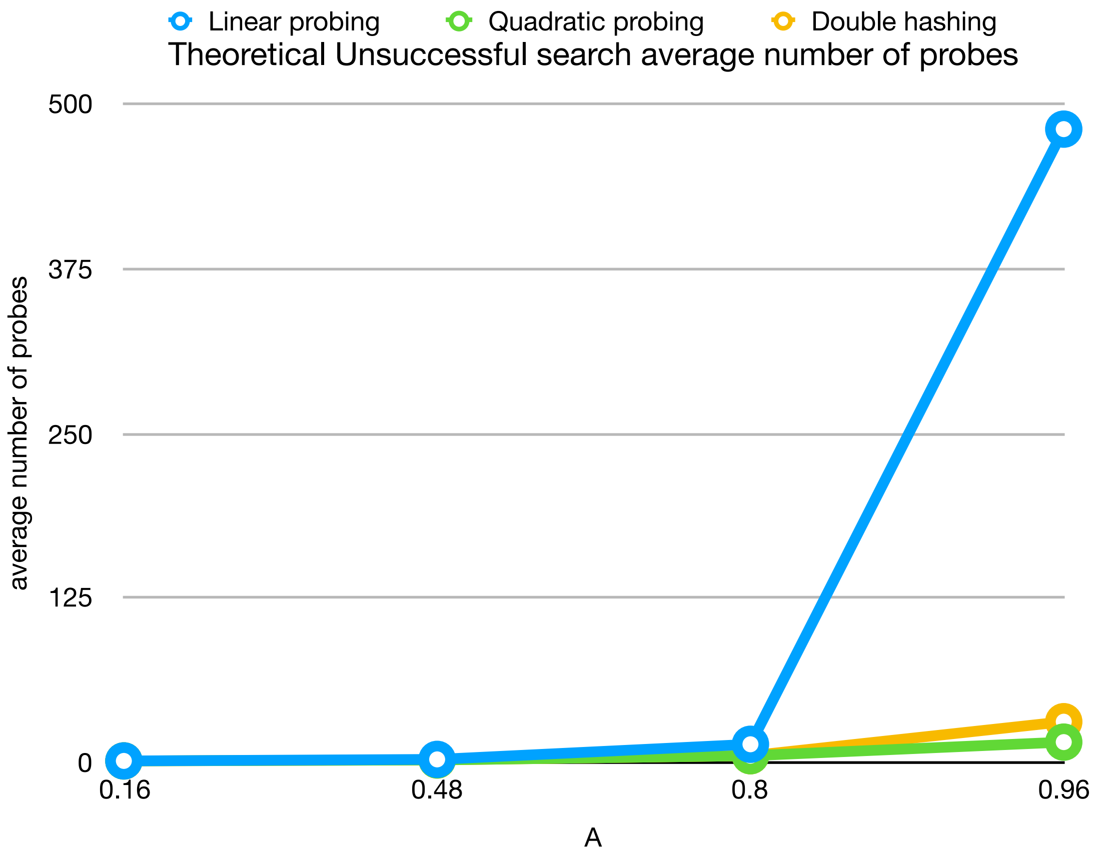
Double when table size is 31

Table Size = 31	Number of index that are occupied : 5	Number of index that are occupied : 15	Number of index that are occupied : 25	Number of index that are occupied : 30
	a=0.16129	a=0.483871	a= 0.806452	a=0.967742
Experimental Successful search average number of probes	2.224	1.54193	2.62445	3.8428
Experimental Unsuccessful search average number of probes	-1	-1	-1	-1
Theoretical Successful search average number of probes	1.09052	1.36689	2.03636	3.54845
Theoretical Unsuccessful search average number of probes	1.19231	1.9375	5.16667	31




Linear probing
Quadratic probing
Double hashing
Theoretical Successful search average number of probes
average number of probes
16
12
8
4
0
0.16
0.48
0.8
0.96
Load factor


Linear probing
Quadratic probing
Double hashing
Experimental Unsuccessful search average number of probes
average number of probes
20
15
10
5
0
-5
0.16
0.48
0.8
0.96
A



Part 3

First of all, in my experiments I have used table size 31. Since it is a prime number, it would be beneficial for the distribution of items to the hash table. The thing that I have realized related to theoretical formulas is that their results varies according to the value of load factor. Hence, I try to observe the changes when different load factors used with same size.

Successful search

As it can be seen from the graphs, theoretical and experimental values are similar. **For successful search**, I have used the **theoretica**l formula which is given by the course slides. And, noticed that this value is changing according to the *a* value, it is the only factor. The formula for quadratic and double hashing is the same. However, linear probing formula was different. What I have noticed from the graph, when the *a* is smaller than 0.5, the average number of probes for successful search is very similar. On the other hand, when *a* is larger than 0.5, there occurs a drastic change in the linear probing compared to other collision strategies. The reason is in linear probing, it is easy for clusters can merge together and create a new cluster that contains more items when the hash table is close to full. When we are searching for an item, there is a high chance that long probe searches can be occur, we might not find the item at first time search. In the quadratic collision and double hashing (last one depends on the hash2 function) strategy, since we are searching indexes like i+1,i+4,i+9…, the clustering of more number of elements can be reduced. For that reason, when searching happens for an item, there is a high chance that the counter will be less compared to linear probing. Also, in the graph after the load factor is larger than 0.5, there is a change but not drastically in the quadratic probing line. However, still we can say that the close to the full, the worse for the searching process. When it is close to the full, the cluster number might become lesser, that means clusters can merge and that results more step when searching for an item. But again compared to linear probing, not many step.

Experimental results are similar to the theoretical ones. When the load factor is larger than 0.5, the change in the linear probing line grows faster compared to others. Its reason again is explained by the clustering problem which I mentioned about in the previous lines. Also, in the theoretical graph when load factor is 0.96, the difference between linear probing and quadratic, double probing is approximately 12, in the experimental graph is 3. There is not much drastic change in my experimental graph compared to theoretical one. I think its reason can be explained as follows. In experimental graph, the result varies according to number of probes when searching for an item. Hence, the items locations are important. However, still, the experimental graph is highly similar to the theoretical one.

For the efficiency of the operation “search” for successful search, the load factor should be less than 0.5. That means the hash table should not be more than half full of the size. In that case, time complexity of the average case finding an item can be O(1). In cases, when the load factor is larger than 0.5, it is not beneficial to use hash table, however, when it is used, quadratic or double hashing collision strategy should be preferred over linear collision strategy.

Unsuccessful search

For unsuccessful search, I have used the **theoretical** formula which is given by the course slides. The graph is similar to graph called Theoretical Successful search average number of probes. First of all, I have noticed that the values, average number of probe, are highly similar when the load factor is less than 0.5 for each collision strategy. As the load factor increases, the graph lines for each collision strategy becomes more distinct. To consider, linear probing after the value of load factor becomes 0.8, there is a huge change in the line. The reason is that since the table become closer to being full and the clusters are merging, unsuccessful searches will become more costly. Finding an empty index will become harder, that is the reason why the searching process takes longer for an item. Hence, the average number of probes will be larger. And also compared to successful search, it makes sense that the unsuccessful average number of probes are higher. Again, for quadratic and double hashing collision strategies’ average number of probes for unsuccessful search is same since the formulas are the same. In addition to that, there is no such drastic change in the graph line compared to linear probing. In my opinion, its reason is that quadratic collision strategy (in some cases double hashing, depends on the function) reduces the clustering problem that linear probing occurs.

My experimental result actually is not very similar to the theoretical ones. I think its reason is about secondary clustering. It makes sense that when linear collision strategy is used, the average number of probes will increase as it the hash table become closer to being full. The reason is that clusters are merging and forming a bigger cluster. In my experiment, the same thing happened for the quadratic collision strategy. It is called secondary clustering.[1] Since the table is close to full, unsuccessful searches took longer.

In conclusion, hash tables allows O(1) time complexity for average case in search operation. It happens when the load factor is less than 0.5. However, when the hash is close to the full, searching operation's time complexity become O(n). It depends on the implementation also the table size must be prime number in order to make the search function's time complexity O(1).

References

[1] "Data Structures and Algorithms in Java." *Google Kitaplar*, Google, books.google.com.tr/books?id=iFc0DwAAQBAJ&pg=PT959&lpg=PT959&dq=Secondary+clustering+hashing&source=bl&ots=8oxxWjn3Tp&sig=ACfU3U3RpmgbuMhEnh1yVm03iPDp04dEow&hl=tr&sa=X&ved=2ahUKEwiRwKPX67npAhVl2aYKHcfDBv4Q6AEwFHoECBQQAQ#v=onepage&q=Secondary clustering hashing&f=false.