ASSIGNMENT 7

1. **Aim** – Insert the keys into a hash table of length m using open addressing using double hashing with h(k)=1+(kmod(m-1))

2. **Objective** – To implement hash table for inserting and searching keys that are mapped with the values using double hashing.

3. **Theory –**

In computing, a hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Open Addressing
Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).
Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot’s key doesn’t become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as “deleted”.
Insert can insert an item in a deleted slot, but the search doesn’t stop at a deleted slot.

In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.[18] The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. (This method is also called **closed**

hashing; it should not be confused with "open hashing" or "closed addressing" that usually mean separate chaining.)

Double hashing is a collision resolving technique in **Open Addressed** Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Double hashing is a computer programming technique used in hash tables to resolve hash collisions, in cases when two different values to be searched for produce the same hash key. It is a popular collision-resolution technique in open-addressed hash tables. Double hashing is implemented in many popular libraries.

Double hashing can be done using :
(hash1(key) + i * hash2(key)) % TABLE_SIZE
Here hash1() and hash2() are hash functions and TABLE_SIZE
is size of hash table.
(We repeat by increasing i when collision occurs)

First hash function is typically hash1(key) = key % TABLE_SIZE

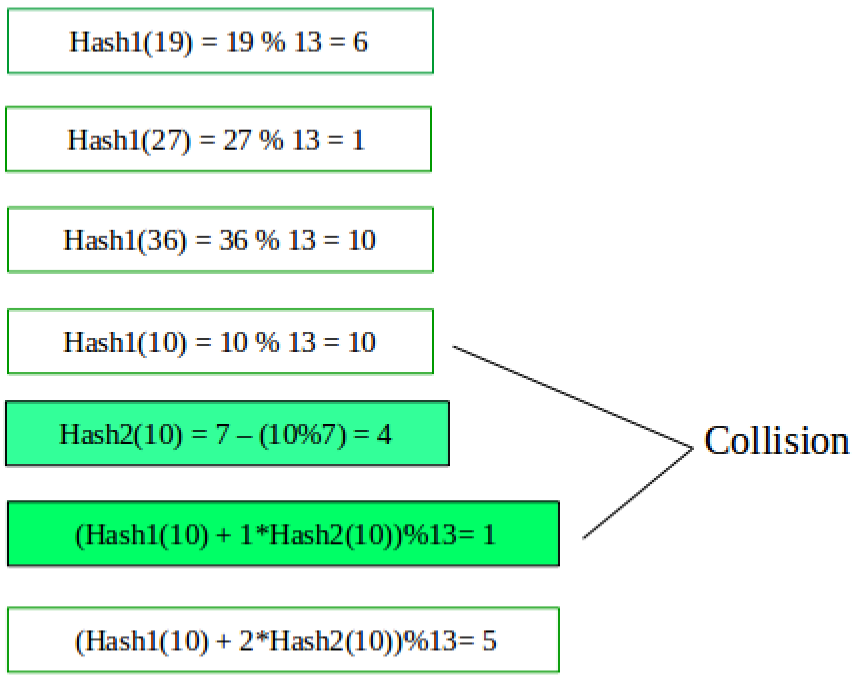
A popular second hash function is : **hash2(key) = PRIME – (key % PRIME)** where PRIME is a prime smaller than the TABLE_SIZE.

A good second Hash function is:

- It must never evaluate to zero
- Must make sure that all cells can be probed

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)



Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

4. **Algorithm –**

Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;

and so on…

Assumption

- There are no more than 20 elements in the data set.
- Hash functions will return an integer from 0 to 19.
- Data set must have unique elements.

```
    string hashTable[21];
    int hashTableSize = 21;
```

Insert

```
  void insert(string s)
    {
        //Compute the index using the hash function1
        int index = hashFunc1(s);
        int indexH = hashFunc2(s);
        //Search for an unused slot and if the index exceeds the
hashTableSize roll back
        while(hashTable[index] != "")
            index = (index + indexH) % hashTableSize;
        hashTable[index] = s;
    }
```

Search

```
  void search(string s)
    {
        //Compute the index using the hash function
        int index = hashFunc1(s);
        int indexH = hashFunc2(s);
         //Search for an unused slot and if the index exceeds the
hashTableSize roll back
        while(hashTable[index] != s and hashTable[index] != "")
            index = (index + indexH) % hashTableSize;
        //Is the element present in the hash table
        if(hashTable[index] == s)
            cout << s << " is found!" << endl;
        else
            cout << s << " is not found!" << endl;
    }
```

5. **Program code –**

```
#include<iostream>

using namespace std;



class hashTable

{
```

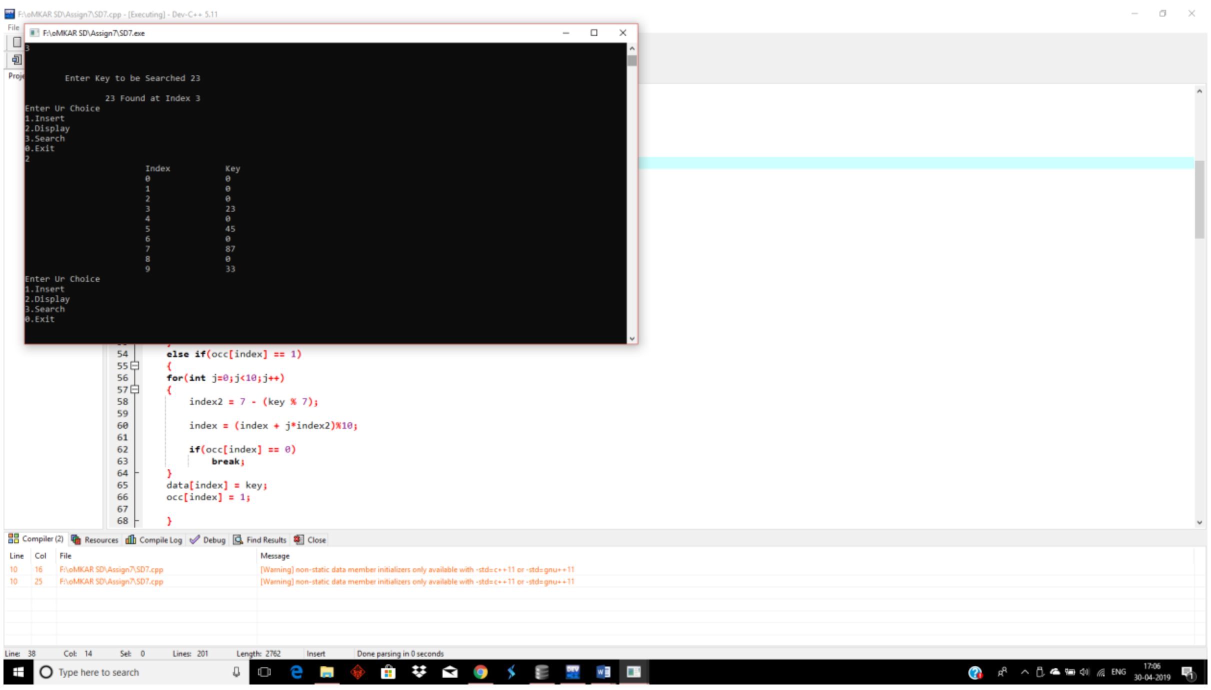
```
public:
        int data[10],occ[10];
        int key,index=0,index2=0,n;

hashTable()
{

        for(int i=0;i<10;i++)
        {
                occ[i]=0;
                data[i]=0;
        }

}
void insert();
void calIndex();
void display();
void search();
void delet();
};

void hashTable::insert()
{

        cout<<"\n\n\tHow many Keys u Want To Enter?? ";
        cin>>n;
```

```
for(int i=0;i<n;i++)
{
        cout<<"\n\n\tEnter Key Value";
        cin>>key;

        index = (key % 10);
        callndex();
}

}

void hashTable::callndex()
{

        if(occ[index]==0)
        {
        data[index] = key;
        occ[index] = 1;
        }
        else if(occ[index] == 1)
        {
        for(int j=0;j<10;j++)
        {
                index2 = 7 - (key % 7);
                index = (index + j*index2)%10;


                if(occ[index] == 0)
                        break;
        }
        data[index] = key;
        occ[index] = 1;


        }
}



void hashTable::display()
{
        cout<<"\t\t\tIndex "<<"\t\tKey\n";
        for(int i=0;i<10;i++)
        cout<<"\t\t\t"<<i<<"\t\t"<<data[i]<<"\n";


}


/*
void hashTable::delet()
{
        int del;
        cout<<"\n\n\tEnter Key to be Deleted ";
        cin>>del;
        for(int i=0;i<10;i++)
        {
        if(data[i]==del)
        {
                cout<<"\n\t\t"<<del<<" Deleted from Index "<<i<<"\n";
                data[i]=0;
                occ[i]=0;
        }
        }



}
*/

void hashTable::search()
{
        int search;
        cout<<"\n\n\tEnter Key to be Searched ";
        cin>>search;

        for(int i=0;i<10;i++)
        {
        if(data[i]==search)
        {
                cout<<"\n\t\t"<<search<<" Found at Index "<<i<<"\n";
```

```
		}
		}
	}



	int main()
	{
		int ch;
		hashTable h1;

		do{

		cout<<"Enter Ur Choice\n1.Insert\n2.Display\n3.Search\n0.Exit\n";
		cin>>ch;

		switch(ch)
		{
			case 1: h1.insert();
					break;
			case 2: h1.display();
					break;
			case 3: h1.search();
					break;

		}
```

```
}while(ch!=0);


}
```

6. **Output screen shots/Copy Past From Terminal –**



F:\oMKAR SD\Assign7\SD7.cpp - [Executing] - Dev-C++ 5.11

F:\oMKAR SD\Assign7\SD7.exe

```
3


        Enter Key to be Searched 23

                23 Found at Index 3
Enter Ur Choice
1.Insert
2.Display
3.Search
0.Exit
2
                        Index           Key
                        0               0
                        1               0
                        2               0
                        3               23
                        4               0
                        5               45
                        6               0
                        7               87
                        8               0
                        9               33
Enter Ur Choice
1.Insert
2.Display
3.Search
0.Exit
```

```
        else if(occ[index] == 1)
        {
        for(int j=0;j<10;j++)
        {
            index2 = 7 - (key % 7);

            index = (index + j*index2)%10;

            if(occ[index] == 0)
                break;
        }
        data[index] = key;
        occ[index] = 1;

        }
```

```
Line  Col  File                          Message
10    16   F:\oMKAR SD\Assign7\SD7.cpp   [Warning] non-static data member initializers only available with -std=c++11 or -std=gnu++11
10    25   F:\oMKAR SD\Assign7\SD7.cpp   [Warning] non-static data member initializers only available with -std=c++11 or -std=gnu++11
```

```
Line: 38   Col: 14   Sel: 0   Lines: 201   Length: 2762   Insert   Done parsing in 0 seconds
```

7. **Conclusion –**

Associative arrays: Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).

Database indexing: Hash tables may also be used as disk-based data structures and database indices (such as in dbm).

Caches: Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.

Object representation: Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.

Hash Functions are used in various algorithms to make their computing faster