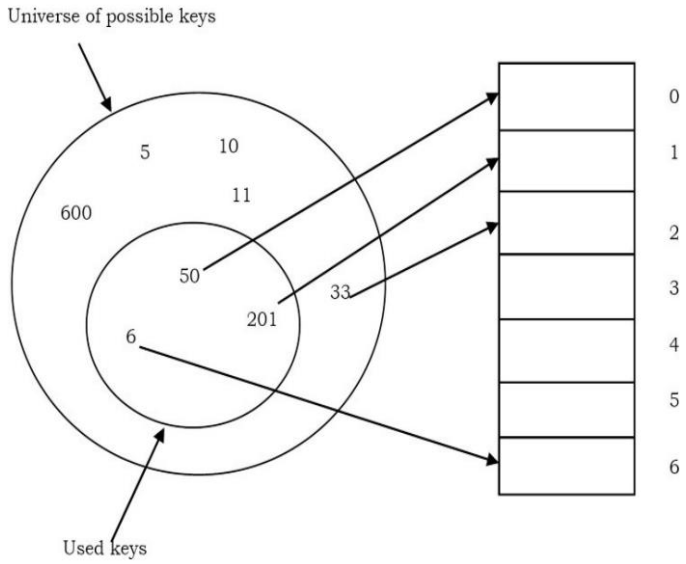
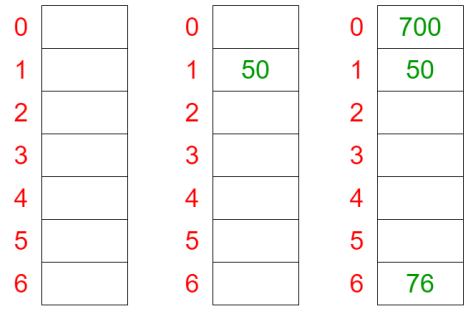
Name	**Hatim Yusuf Sawai**
UID no.	**2021300108**
Experiment No.	**10**

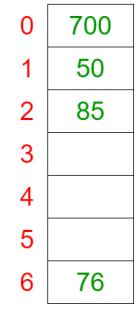
AIM:	**Program on Open Addressing Hashing**
Program	
PROBLEM STATEMENT:	Implement Quadratic Probing technique for collision resolution in an open addressing hash table
THEORY:	**Hashing:** Hashing is a technique used for storing and retrieving information as quickly as possible. It is used to perform optimal searches and is useful in implementing symbol tables.  **Components of Hashing:** 1. Hash Table 2. Hash Functions 3. Collisions 4. Collision Resolution Techniques

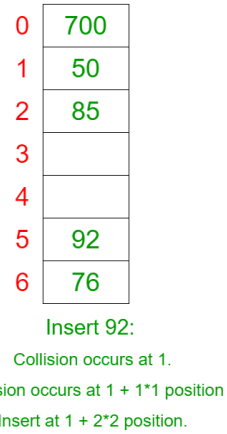
	Hash Table: Hash table is a generalization of array. With an array, we store the element whose key is k at a position k of the array. That means, given a key k, we find the element whose key is k by just looking in the k th position of the array. This is called direct addressing. Hash table or hash map is a data structure that stores the keys and their associated values, and hash table uses a hash function to map keys to their associated values. The general convention is that we use a hash table when the number of keys actually stored is small relative to the number of possible keys. **Hash Function:** Hash table or hash map is a data structure that stores the keys and their associated values, and hash table uses a hash function to map keys to their associated values. The general convention is that we use a hash table when the number of keys actually stored is small relative to the number of possible keys. **Characteristics of Good Hash Functions:** 1. Minimize **collision** 2. Be **easy** and quick to **compute** 3. **Distribute key** values **evenly** in the hash table 4. Use **all** the **information** provided in the key 5. Have a **high load factor** for a given set of keys **Collisions:** Hash functions are used to map each key to a different address space, but practically it is not possible to create such a hash function and the problem is called collision. Collision is the condition where two records are stored in the same location. **Collision Resolution Techniques:** The process of finding an alternate location is called collision resolution. There are a number of collision resolution techniques, and the most popular are seperate chaining and open addressing.

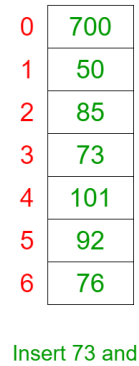
	Open Addressing: It is an Array-based implementation of hash table which uses 3 types of probing techniques to minimize collisions: **Quadratic Probing:** The interval between probes increases proportionally to the hash value (the interval thus increasing linearly, and the indices are described by a quadratic function). The problem of **Primary Clustering** can be eliminated if we use the quadratic probing method. Hashing Function is given by: **h(k,i) = (h'(k)+c1*i+c2*i*i)%m** where h'(k) is an auxiliary hash function, C1 and C2 are positive auxiliary constants, and i = 0,1,2…,m-1. The initial position probed is T[h'(k)]; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This method works much better than linear probing, but to make full use of the has table, the values of c1,c2 & m are constrained.

Quadratic Probing Example







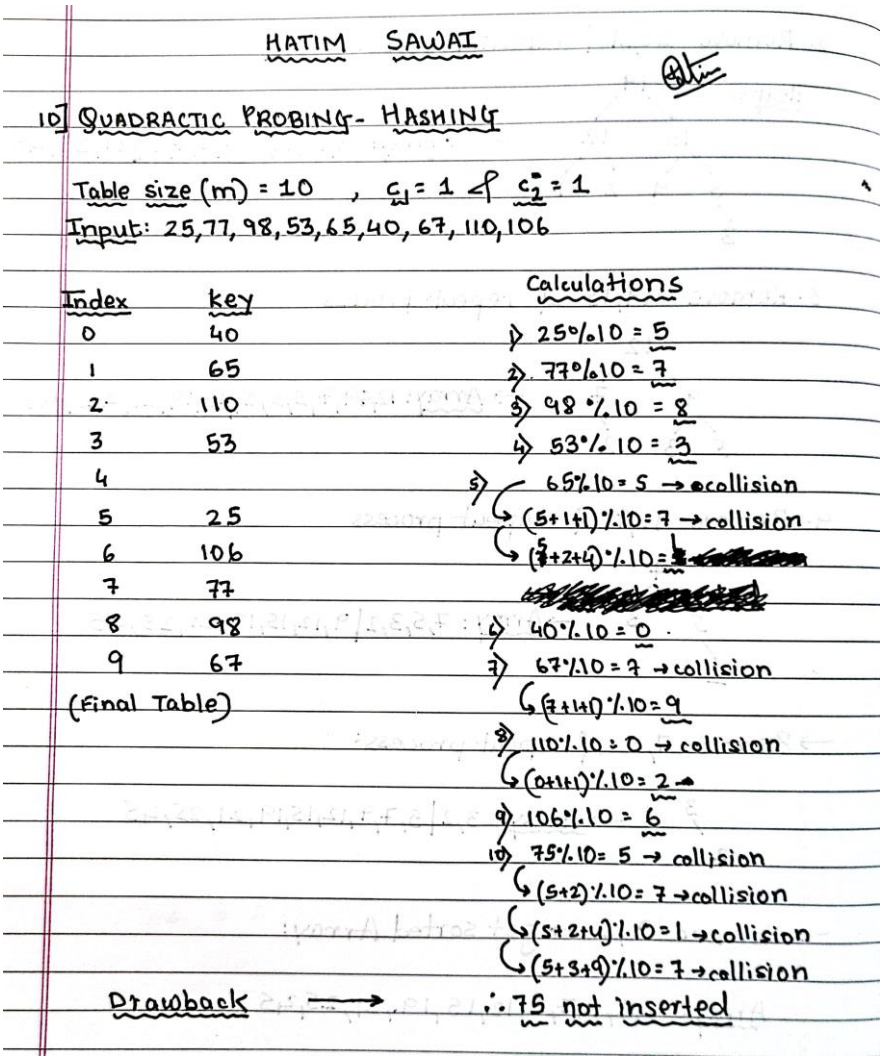


Disadvantages:

Even though clustering is avoided by quadratic probing, still there are chances of clustering. Clustering is caused by multiple search keys mapped to the same hash key. Thus, the probing sequence for such search keys is prolonged by repeated conflicts along the probing sequence. Both linear and quadratic probing use a probing sequence that is independent of the search key.

ALGORITHM:	1. Create OQHash class with int[] hashtable & int size members 2. Initialize constructor and set size,c1 & c2 3. Create methods for hashing **CreateHash Method:** 1. for i=0 to i<size: 2. set hashtable[i] to -1 3. end for loop **quadraticProbe Method:** 1. initialize I to 0 & count=0 2. initialize index = key%size 3. while hashtable[index] is not equal to -1: 4. index = (index+(c1*i)+(c2*i*i))%size 5. if hashtable[index] is not equal to -1: 6. increment I & count 7. end if 8. if count is more than size-1: 9. return -1 10. end if 11. end while loop 12. return index **HashInsert Method:** 1. index = **quadraticProbe(key)** 2. if index is not equal -1: 3. set hashtable[index] = key 4. else print element not inserted 5. end if **HashDisplay Method:** 1. for i=0 to i<size: 2. print I & hashtable[i] → prints index and key 3. end for loop

PROBLEM SOLVING:



HATIM SAWAI

10] QUADRACTIC PROBING - HASHING

Table size (m) = 10 , $c_1 = 1$ & $c_2 = 1$

Input: 25, 77, 98, 53, 65, 40, 67, 110, 106

Index	key
0	40
1	65
2	110
3	53
4	
5	25
6	106
7	77
8	98
9	67

(Final Table)

Calculations

1) 25%10 = 5
2) 77%10 = 7
3) 98 %10 = 8
4) 53% 10 = 3
5) 65%10 = 5 → collision
   (5+1+1)%10 = 7 → collision
   (5+2+4)%10 = 1
6) 40%10 = 0
7) 67%10 = 7 → collision
   (7+1+1)%10 = 9
8) 110%10 = 0 → collision
   (0+1+1)%10 = 2
9) 106%10 = 6
10) 75%10 = 5 → collision
    (5+2)%10 = 7 → collision
    (5+2+4)%10 = 1 → collision
    (5+3+9)%10 = 7 → collision

Drawback → ∴ 75 not inserted

PROGRAM:

HashCheck.java:

```
import java.util.Scanner;
import hashds.OQHash;
public class HashCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter size of hash table: ");
        int size = sc.nextInt();
        System.out.print("Enter c1 and c2: ");
        int c1 = sc.nextInt();
        int c2 = sc.nextInt();
        OQHash hash = new OQHash(size,c1,c2);
```

```
        hash.createHashTable();
        int choice,flag,key;
        while(true) {
            System.out.println("Select an Option:\n1. Insert\t2.
Delete\t3. Search\t4. Display");
            choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter no. of keys to insert: ");
                    int n = sc.nextInt();
                    for (int i=0;i<n;i++) {
                        System.out.print("Enter key "+(i+1)+": ");
                        key = sc.nextInt();
                        hash.HashInsert(key);
                    }
                    System.out.println("\nFinal Hash Table: ");
                    hash.HashDisplay();
                    break;
                case 2:
                    System.out.print("Enter key to delete: ");
                    key = sc.nextInt();
                    hash.HashDelete(key);
                    System.out.println("Hash table after deletion:");
                    hash.HashDisplay();
                    break;
                case 3:
                    System.out.print("Enter key to search: ");
                    key = sc.nextInt();
                    hash.HashSearch(key);
                    break;
                case 4:
                    System.out.println("Hash table:");
                    hash.HashDisplay();
                    break;
                default:
                    System.out.println("Invalid choice");
```

```
            }
            System.out.print("Do you want to continue?(1/0): ");
            flag = sc.nextInt();
            if (flag == 0) {
                break;
            }
        }
        sc.close();
    }
}
```

OQHash.java:

```
package hashds;
public class OQHash {
    private int[] hashtable;
    private int size;
    int c1,c2;
    public OQHash(int size,int c1,int c2) {
        this.size = size;
        hashtable = new int[size];
        this.c1 = c1;
        this.c2 = c2;
    }
    public void createHashTable() {
        for (int i = 0; i < size; i++) {
            hashtable[i] = -1;
        }
    }
    public int quadraticProbe(int key) {
        int i = 0;
        int count=0;
        int index = key%size;
        if(hashtable[index]!=-1) {
            System.out.println("Key inserted at: "+index);
        }
        while (hashtable[index]!=-1) {
```

```
        index = (index+(c1*i)+(c2*i*i))%size;
        if(hashtable[index]!=-1) {
            System.out.println("Collision occured at: "+index);
            i++;
            count++;
        }
        if(count>size-1) {
            return -1;
        }
    }
    return index;
}
public void HashInsert(int key) {
    int index = quadraticProbe(key);
    if(index!=-1) {
        System.out.println("Key Inserted at: " + index);
        hashtable[index] = key;
    }
    else {
        System.out.println("Element not inserted!");
    }
}
public boolean HashSearch(int key) {
    int index=key%size;
    int i=0;
    while(hashtable[index]!=-1) {
        if(hashtable[index]==key) {
            System.out.println("Key found at: "+index);
            return true;
        }
        index = (index+(c1*i)+(c2*i*i))%size;
        i++;
    }
    System.out.println("Key not found!");
    return false;
}
```

```
    public void HashDelete(int key) {
        int index = key % size;
        int i = 0;
        while (hashtable[(index + i) % size] != key) {
            i++;
        }
        hashtable[(index + i) % size] = -1;
    }
    public void HashDisplay() {
        System.out.println("Index\tkey");
        for (int i = 0; i < size; i++) {
            if (hashtable[i]!=-1) {
                System.out.println("["+i+"]\t"+hashtable[i]);
            } else {
                System.out.println("["+i+"]");
            }
        }
        System.out.println();
    }
}
```

OUTPUT:

```
PS D:\Data Structures\Exp10> cd "d:\Data Structures\Exp10\"
Enter size of hash table: 10
Enter c1 and c2: 1 1
Select an Option:
1. Insert       2. Delete       3. Search       4. Display
1
Enter no. of keys to insert: 9
Enter key 1: 25
Key Inserted at: 5
Enter key 2: 77
Key Inserted at: 7
Enter key 3: 98
Key Inserted at: 8
Enter key 4: 53
Key Inserted at: 3
```

```
Enter key 5: 65
Collision occured at: 5
Collision occured at: 7
Key Inserted at: 1
Enter key 6: 40
Key Inserted at: 0
Enter key 7: 67
Collision occured at: 7
Key Inserted at: 9
Enter key 8: 110
Collision occured at: 0
Key Inserted at: 2
Enter key 9: 106
Key Inserted at: 6
Enter key 10: 75
Collision occured at: 5
Collision occured at: 7
Collision occured at: 1
Collision occured at: 7
Collision occured at: 5
Collision occured at: 5
Collision occured at: 7
Collision occured at: 1
Collision occured at: 7
Collision occured at: 5
Element not inserted!

 Final Hash Table:
 Index   key
 [0]     40
 [1]     65
 [2]     110
 [3]     53
 [4]
 [5]     25
 [6]     106
 [7]     77
 [8]     98
 [9]     67

 Do you want to continue?(1/0): 0
PS D:\Data Structures\Exp10>
```

CONCLUSION:	In this experiment, we learned how to implement a hash table using arrays and how to implement quadratic probing using constants to minimize collisions occurring while mapping keys.