Week 13	Introduction to Hashing. Hashing - Perfect hashing functions. Hash table Hash Functions, Operations, Hash collision, Application.

Introduction to Hashing

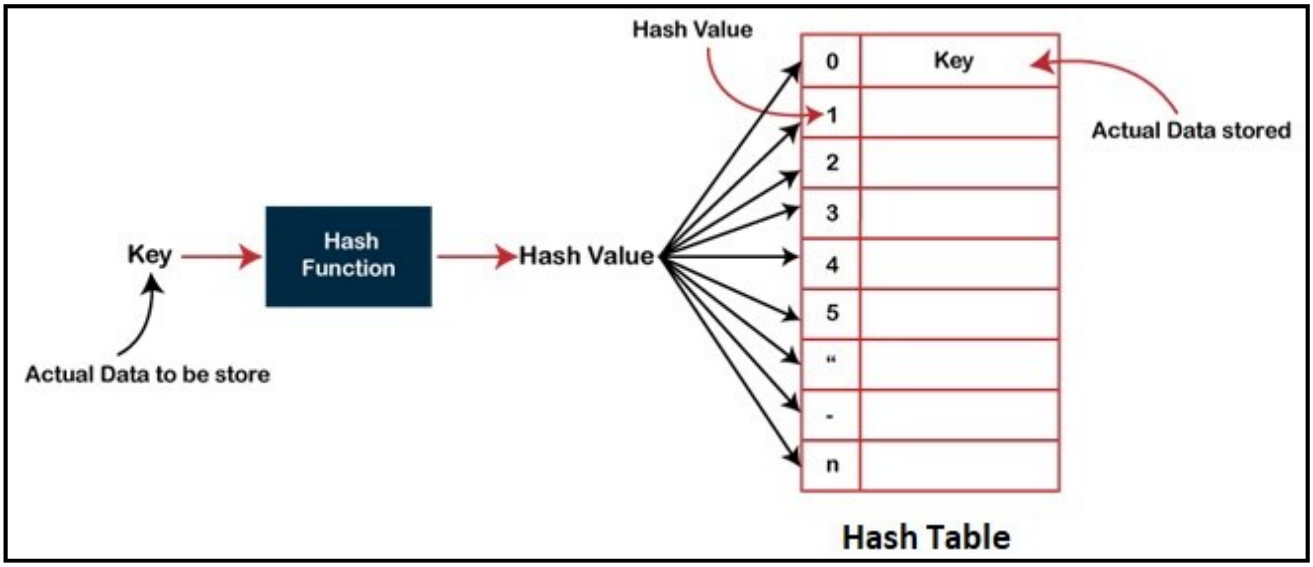
- Searching is the most common operation applied to collections of data.
- Searching problem attempts to find a key item in a collection of data items.
- It's not only used to check if a key item is in the collection, but can also be used in adding new items to the collection and removing existing items.
- So, Searching operation must perform fast and efficiently.
- The linear search technique can be used to search an item. The linear search is simple to implement but not very efficient.
- The binary search technique is efficient, but it must be applied on sorted collection of data.
- To overcome these problems, Hashing technique can be used.
- Hashing technique provides direct access to the search keys.

Hashing

- Hashing is a technique of mapping keys, and values into the hash table by using a hash function.
- It is done for faster/direct access to elements.
- In Hashing technique, the hash table and hash function are used.
- It uses hash tables to store the data items. Each value in the hash table has assigned a unique key.
- Using the hash function, we can calculate the slot or key location at which the value can be stored.
- Hashing technique uniquely identifies a key item from a collection of similar items.
- Operations on Hash Table:
  - ✓ Insertion
  - ✓ Searching
  - ✓ Deletion
- Examples of Hashing:
  - ✓ In schools, the teacher assigns a unique roll number to each student. Later, the teacher uses that roll number to retrieve information about that student.
  - ✓ A library has an infinite number of books. The librarian assigns a unique number to each book. This unique number helps in identifying the position of the books on the bookshelf.

Hash Function

- Hash function converts or maps the keys to specific entries in the table.
- Hash function generates hash value. Hash value specifies the slot or location of key in hash table.



- For example, suppose we have the following set of keys: 765, 431, 96, 142, 579 and a hash table, T, containing M = 13 elements. We can define a simple hash function as follows:

  h(key) = key % M

- To add keys to the hash table, we apply the hash function to determine the entry in which the given key should be stored.
- Applying the hash function to key 765 yields a result of 11, which indicates 765 should be stored in location 11 of the hash table.
- Likewise, if we apply the hash function to the next four keys in the list, we find:

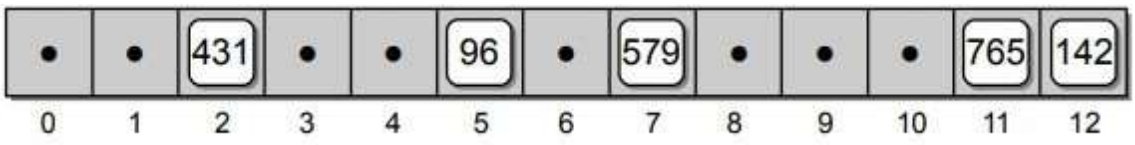
  h(431) => 2

  h(96) => 5

  h(142) => 12

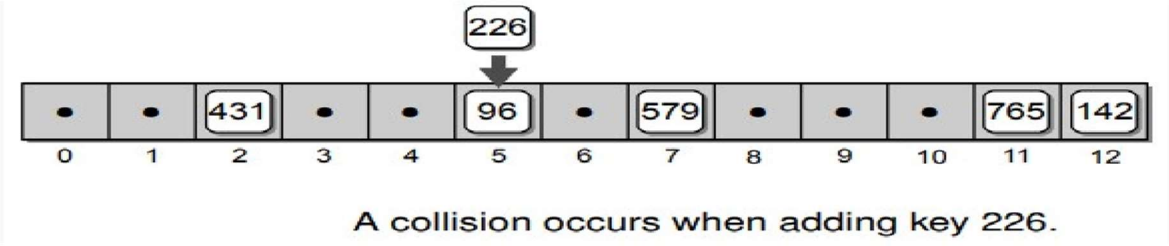
  h(579) => 7

- Figure below illustrates the insertion of the first five keys into the hash table:



Hash Collision

- Consider what happens when we attempt to add key 226 to the hash table. The hash function maps this key to location 5, but that location already contains key 96. The result is a collision, which occurs when two or more keys map to the same hash location.



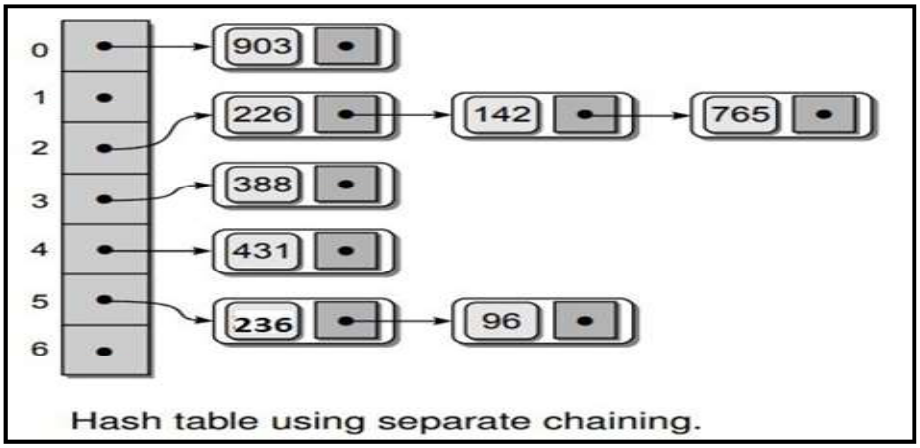
A collision occurs when adding key 226.

- When the hash function generates the same slot/location for multiple keys, there will be a conflict (what value to be stored in that index). This is called a **hash collision.**
- **Hashing Function is said to be Perfect Hashing Function if it results no collisions.**
- To resolve these collisions, we can use following collision resolution techniques:
  - ✓ Open Hashing: It is also known as closed addressing.
  - ✓ Closed Hashing: It is also known as open addressing.

Open Hashing

- ✓ In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.
- ✓ In chaining method, each slot/location is capable of holding multiple keys.
- ✓ To store multiple keys, linked lists can be used.
- ✓ For example, suppose we have the following set of keys: 903, 226, 236, 96, 142, 388, 431, 765 and a hash table, T, containing M = 7 elements and a simple hash function as follows:

h(key) = key % M



Hash table using separate chaining.

Closed Hashing

In Closed hashing, three techniques are used to resolve the collision:

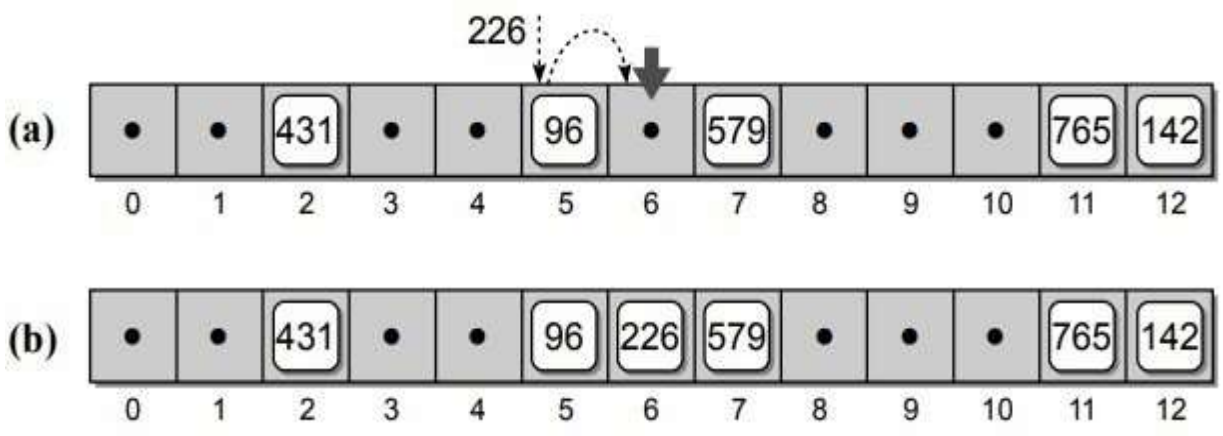
1. Linear probing
2. Quadratic probing
3. Double Hashing technique
4. Rehashing technique

Linear Probing

- ✓ To resolve the collision, the simplest approach is to use a linear probe, which checks the table slots in sequential order starting with the first location immediately following the original hash location.

Insert Operation

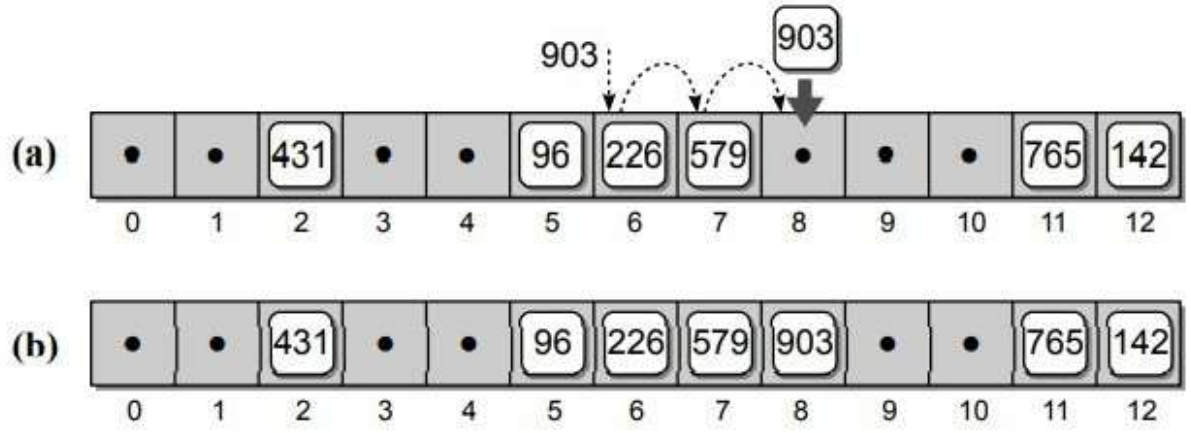
- ✓ For key value 226, the linear probe finds slot 6 available, so the key can be stored at that position.



Resolving a collision for key 226 requires adding the key to the next slot.

- ✓ When key 903 is added, the hash function maps the key to index 6, but we just added key 226 to this entry.
- ✓ Instead of deleting 226 and adding 903 to slot 6, probe to find another slot.

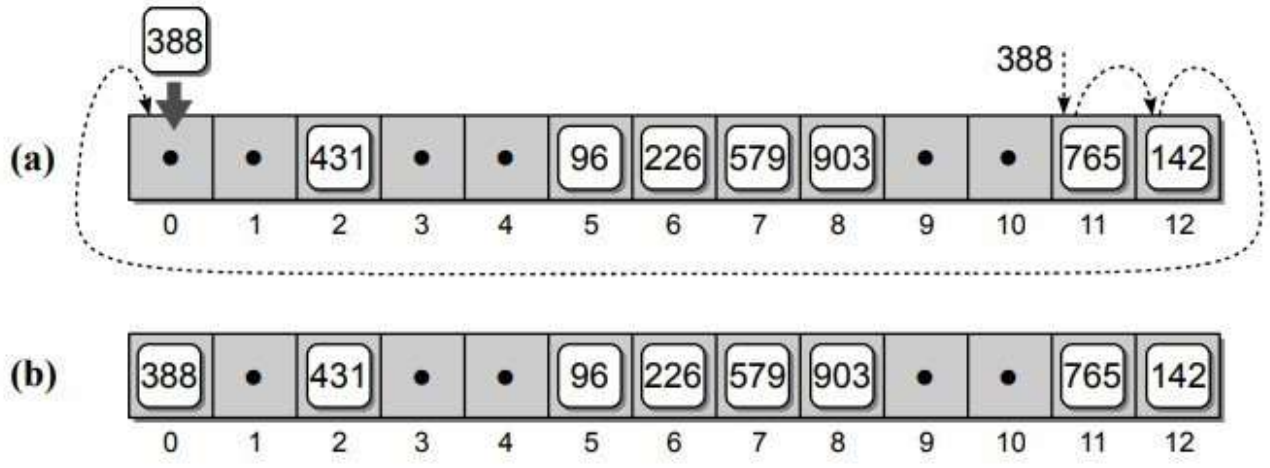
✓ In the case of key 903, the linear probe leads us to slot 8, as illustrated in Figure.



Adding key 903 to the hash table: (a) performing a linear probe; and (b) the result after adding the key.

✓ If the end of the hash table is reached during the probe, we have to go back to the first slot and continue until either an available slot is found or all slots have been examined.

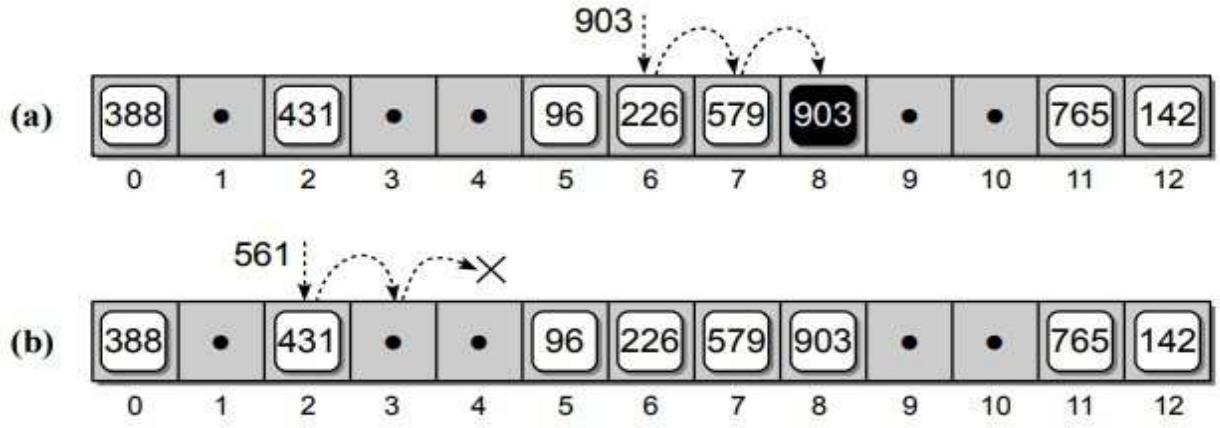
✓ For example, if we add key 388 to the hash table, the hash function maps the key to slot 11, which contains key 765. The linear probe, as illustrated in Figure, requires go back to the beginning of the hash table.



Adding key 388 to the hash table: (a) performing a linear probe; and (b) result after adding the key.

Searching Operation

- ✓ Searching a hash table for a specific key is very similar to the insert operation.
- ✓ The target key is mapped to an initial slot in the table and then it is determined if that entry contains the key.
- ✓ If the key is not at that location, the probe must be used to find the target key.
- ✓ In this case, the probe continues until the target key is located, a null reference is encountered, or all slots have been examined.
- ✓ When either of the last two situations (a null reference is encountered, or all slots have been examined) occurs, this indicates the target key is not in the table.
- ✓ Figure illustrates the searches for key 903, which is in the table, and key 561, which is not in the table.



Searching the hash table: (a) a successful search for key 903 and (b) an unsuccessful search for key 561

Delete Operation

- ✓ Deleting from a hash table is a bit more complicated than an insertion.
- ✓ A search can be performed to find the key in hash table.
- ✓ After finding the key, simply remove it by setting the corresponding table location to None.

Modified Linear Probing

- ✓ In Simple Linear probing, probing takes place to next slot/location (i.e c=1)

  slot = (home + i) % M

  where i = 1, 2, . . . M − 1. home is the home position, which is the index to which the key was originally mapped by the hash function

- ✓ Suppose we use a simple linear probe (i.e c = 1) to build the hash table using the keys: 765, 431, 96, 142, 579, 226, 903, 388.

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 6
h(96)  => 5           h(903) => 6   => 7  => 8 => 9
h(142) => 12          h(388) => 11  => 12 => 0
```

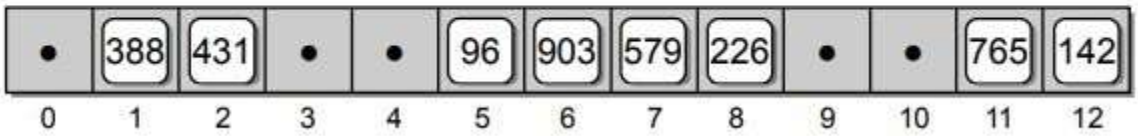
- ✓ We can improve the linear probe by skipping multiple slots instead of probing the immediate successor of each slot.

  slot = (home + i) % M

  where i = 1, 2, . . . M − 1. home is the home position, which is the index to which the key was originally mapped by the hash function.

- Suppose we use a linear probe with c = 3 to build the hash table using the keys: 765, 431, 96, 142, 579, 226, 903, 388.

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 8
h(96)  => 5           h(903) => 6
h(142) => 12          h(388) => 11  => 1
```



The hash table using a linear probe with $c = 3$.

Quadratic Probing

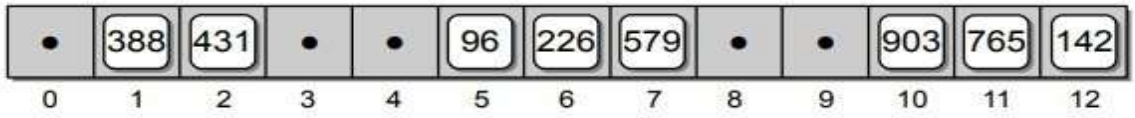
- Quadratic Probing increases the distance between each probe.

  slot = (home + i^2) % M

  where i = 1, 2, . . . M – 1

- Quadratic Probing typically reduces the number of collisions.

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 6
h(96)  => 5           h(903) => 6   => 7  => 10
h(142) => 12          h(388) => 11  => 12 => 2  => 7 => 1
```



The hash table using a quadratic probe.

Double Hashing

- In double hashing, when a collision occurs, the key is hashed by a second function and the result is used as the constant factor in the linear probe:

  slot = (home + i * hp(key)) % M

- A simple choice for the second hash function takes the form:

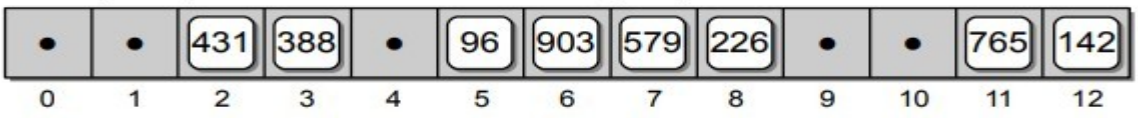
  hp(key) = 1 + key % P

  where P is some constant less than M.

- For example, Suppose we build the hash table using the keys: 765, 431, 96, 142, 579, 226, 903, 388 and a hash table, T, containing M = 13 elements and we define a second hash function:

  hp(key) = 1 + key % 8

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 8
h(96)  => 5           h(903) => 6
h(142) => 12          h(388) => 11  => 3
```

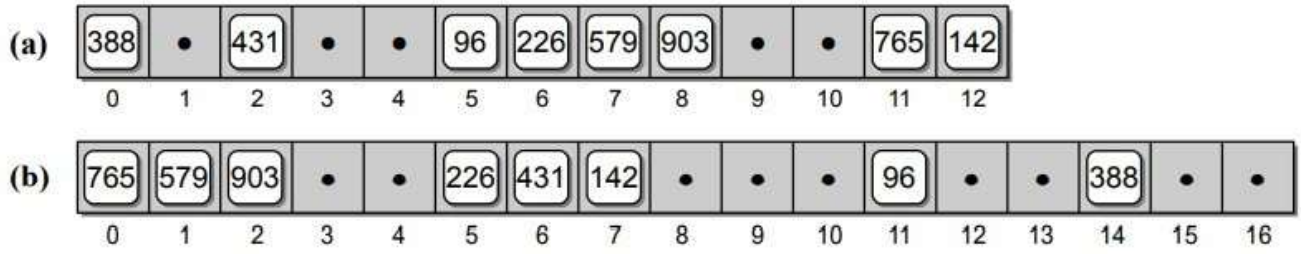


The hash table using double hashing.

- Rehashing means hashing again.
- It is a technique in which new hash table is created using existing hash table.
- During Rehashing, we cannot simply copy the contents from the old hash table to the new hash table.
- Instead, we have to rebuild or rehash the entire table by adding each key to the new hash table using hash function.
- For example, suppose we create a hash table of size M = 17:

```
h(765) => 0                    h(579) => 1
h(431) => 6                    h(226) => 5
h(96)  => 11                   h(903) => 2
h(142) => 6  => 7              h(388) => 14
```

- The original hash table using a linear probe is shown in Figure (a) and the new larger hash table is shown in Figure (b).



- The result of enlarging the hash table from 13 elements to 17.

Applications of Hash Table

Hash tables are implemented where

- constant time lookup and insertion is required
- cryptographic applications
- indexing data is required

Program #1: Python Program to implement Hash Functions.

```
class Hash:
    def __init__(self):
        self.buckets=[[],[],[],[],[]]

    def insert(self,key):
        buc_index = key % 5
        self.buckets[buc_index].append(key)
        print(key,"inserted in Bucket No.",buc_index+1)

    def search(self,key):
        buc_index = key % 5
        if key in self.buckets[buc_index]:
            print(key,"present in bucket No.",buc_index + 1)
        else:
            print(key,"is not present in any of the buckets")

    def display(self):
        for i in range(0,5):
            print("\nBucket No.",i+1,end=":")
            for j in self.buckets[i]:
                print(j,end="-->")


hsh = Hash()

print("Hash operations\n\t1.Insert\n\t2.Search\n\t3.Display\n\t4.Quit")
ch=int(input("Enter your choice\n"))

while ch in [1,2,3]:
    if ch == 1:
        key=int(input("\nEnter key to be inserted\n"))
        hsh.insert(key)
        print("-----------------------------------------------------")
    elif ch == 2:
        key=int(input("\nEnter key to be searched\n"))
        hsh.search(key)
        print("-----------------------------------------------------")
    elif ch == 3:
        hsh.display()
        print("\n-----------------------------------------------------")
    print("\nHash operations\n\t1.Insert\n\t2.Search\n\t3.Display\n\t4.Quit")
    ch=int(input("Enter your choice\n"))
```

Output #1:

```
Hash operations
        1.Insert
        2.Search
        3.Display
        4.Quit
Enter your choice
1
Enter key to be inserted
45
45 inserted in Bucket No. 1
----------------------------------------
Hash operations
        1. Insert
        2.Search
        3.Display
        4.Quit
Enter your choice
1
Enter key to be inserted
49
49 inserted in Bucket No. 5
----------------------------------------
Hash operations
        1. Insert
        2.Search
        3.Display
        4.Quit
Enter your choice
1
```

```
Enter key to be inserted
83
83 inserted in Bucket No. 4
----------------------------------------
Hash operations
        1. Insert
        2.Search
        3.Display
        4.Quit
Enter your choice
1
Enter key to be inserted
55
55 inserted in Bucket No. 1
----------------------------------------
Hash operations
        1. Insert
        2.Search
        3.Display
        4.Quit
Enter your choice
3
Bucket No. 1:45-->55-->
Bucket No. 2:
Bucket No. 3:
Bucket No. 4:83-->
Bucket No. 5:49-->
----------------------------------------
Hash operations
        1. Insert
```

2. Search

3.Display

4.Quit

Enter your choice

2

Enter key to be searched

55

55 present in bucket No. 1

Hash operations

1. Insert

2.Search

3.Display

4.Quit

Enter your choice

2

Enter key to be searched

65

65 is not present in any of the buckets

Hash operations

1. Insert

2.Search

3.Display

4.Quit

Enter your choice

4