CHAPTER 6:
SEARCHING TECHNIQUES

Content

- Linear Search
- Binary Search
- Hashing Concept
- Hash Functions
- Collision Resolution Techniques

Somaiya TRUST

Searching Techniques

Searching means to find whether a particular value is present in an array or not.

If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.

There are two popular methods for searching the array elements:

- **linear search**
- **binary search.**

Linear Search

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value.

It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

Somaiya TRUST

Linear Search

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:      Repeat Step 4 while I<=N
Step 4:              IF A[I] = VAL
                          SET POS = I
                          PRINT POS
                          Go to Step 6
                     [END OF IF]
                      SET I = I + 1
               [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Algorithm for linear search

Somaiya TRUST

Binary Search

Binary search is a searching algorithm that works efficiently with a sorted list.

The mechanism of binary search can be better understood by an analogy of a telephone directory

Binary Search Algorithm

- Consider an array A[] that is declared and initialized as int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
- and the value to be searched is VAL = 9.

- The algorithm will proceed in the following manner.
- BEG = 0, END = 10, MID = (0 + 10)/2 = 5 Now, VAL = 9 and A[MID] = A[5] = 5
- A[5] is less than VAL, therefore, we now search for the value in the second half of the array.
- So, we change the values of BEG and MID.
- Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8 VAL = 9 and A[MID] = A[8] = 8
- A[8] is less than VAL, therefore, we now search for the value in the second half of the segment.
- So, again we change the values of BEG and MID.
- Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9 Now, VAL = 9 and
- A[MID] = 9

Somaiya TRUST

Binary Search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:             SET MID = (BEG + END)/2
Step 4:             IF A[MID] = VAL
                        SET POS = MID
                        PRINT POS
                        Go to Step 6
                    ELSE IF A[MID] > VAL
                        SET END = MID - 1
                    ELSE
                        SET BEG = MID + 1
                    [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Algorithm for binary search

Somaiya TRUST

Time Complexity of Linear and Binary Search

Key	Array of Employees' Records
Key 0 → [0]	Employee record with Emp_ID 0
Key 1 → [1]	Employee record with Emp_ID 1
Key 2 → [2]	Employee record with Emp_ID 2
.....	.....
.....	.....
Key 98 → [98]	Employee record with Emp_ID 98
Key 99 → [99]	Employee record with Emp_ID 99

Figure 15.1 Records of employees

Somaiya TRUST

Time Complexity of Linear and Binary Search

Key	Array of Employees' Records
Key 00000 ⟶ [0]	Employee record with Emp_ID 00000
.....	.....
Key n ⟶ [n]	Employee record with Emp_ID n
.....	.....
Key 99998 ⟶ [99998]	Employee record with Emp_ID 99998
Key 99999 ⟶ [99999]	Employee record with Emp_ID 99999

Figure 15.2 Records of employees with a five-digit Emp_ID

Somaiya TRUST

How to reduce time complexity of searching?

- In order to keep the array size down to the size that we will actually be using (100 elements), use just the last two digits of the key to identify each employee.
- For example, the employee with Emp_ID 79439 will be stored in the element of the array with index 39.
- Similarly, the employee with Emp_ID 12345 will have his record stored in the array at the 45th location

Somaiya TRUST

Hashing

- **Convert a five-digit key number to a two-digit array index.**
- **We need a function which will do the transformation.**
- **In this case, we will use the term hash table for an array and the function that will carry out the transformation will be called a hash function.**

Hash Table

Hash table is a data structure in which keys are mapped to array positions by a hash function.

Somaiya TRUST

Hash Function

- A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table.
- The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed.
- It produces a unique set of integers within some suitable range in order to reduce the number of collisions.

Somaiya TRUST

Hash Function

Properties of a Good Hash Function

- **Low cost:** The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches
- **Determinism:** A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value.
- **Uniformity:** A good hash function must map the keys as evenly as possible over its output range.

Somaiya TRUST

Hash Functions

Division Method

- It is the most simple method of hashing an integer x.
- This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as h(x) = x mod M

Example 15.1 Calculate the hash values of keys 1234 and 5462.

Solution Setting M = 97, hash values can be calculated as:

```
h(1234) = 1234 % 97 = 70
h(5642) = 5642 % 97 = 16
```

Somaiya TRUST

Hash Functions

Multiplication Method

- Step 1: Choose a constant A such that $0 < A < 1$
- Step 2: Multiply the key k by A.
- Step 3: Extract the fractional part of kA.
- Step 4: Multiply the result of Step 3 by the size of hash table (m). Hence, the hash function can be given as:
- . Knuth has suggested that the best choice of A is " (sqrt5 – 1) /2 = 0.6180339887

$$h(k) = \lfloor m\ (kA \bmod 1) \rfloor$$

Example 15.2 Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.

Solution We will use A = 0.618033, m = 1000, and k = 12345

$$\begin{aligned} h(12345) &= \lfloor 1000\ (12345 \times 0.618033 \bmod 1) \rfloor \\ &= \lfloor 1000\ (7629.617385 \bmod 1) \rfloor \\ &= \lfloor 1000\ (0.617385) \rfloor \\ &= \lfloor 617.385 \rfloor \\ &= 617 \end{aligned}$$

Somaiya TRUST

Hash Function

Mid-Square Method

Step 1: Square the value of the key. That is, find k2

Step 2: Extract the middle r digits of the result obtained in Step 1.

Example 15.3 Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

Solution Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so $r = 2$.

When $k = 1234$, $k^2 = 1522756$, $h(1234) = 27$

When $k = 5642$, $k^2 = 31832164$, $h(5642) = 21$

Observe that the 3rd and 4th digits starting from the right are chosen.

Somaiya TRUST

Hash Function

Folding Method

- Step 1: Divide the key value into a number of parts. That is, divide k into parts k1 , k2 , ..., kn , where each part has the same number of digits except the last part which may have lesser digits than the other parts.
- Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn . The hash value is produced by ignoring the last carry, if any

Example 15.4 Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.

Solution

Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

COLLISION

✅ What is a Collision?

In **hashing**, a **collision** occurs when **two different keys** generate the **same hash value** (i.e., same index in the hash table).

📌 Example:

Assume a simple hash function:

```
c

index = key % 5;
```

Now for keys:

- Key = 10 → `10 % 5 = 0`
- Key = 15 → `15 % 5 = 0`

👉 **Both keys map to index 0 → This is a collision**

🧠 Why Do Collisions Happen?

- Because the **hash table size is limited**
- And the number of **possible keys is usually infinite**

No matter how good the hash function is, **collisions are inevitable** when multiple keys are mapped into a fixed-size table.

Somaiya TRUST

COLLISION RESOLUTION

1. Open addressing
2. Chaining

🔷 1. Open Addressing

- **If a slot is full**, probe for another empty slot in the table

Method	Description
Linear Probing	Check the next index sequentially (`index+1`)
Quadratic Probing	Use quadratic formula to find next index
Double Hashing	Use a second hash function for resolving

🔷 2. Chaining (Separate Chaining)

- Each index holds a **linked list** (or other structure)
- Colliding keys are stored in the **same slot** as a list

Somaiya TRUST

OPEN ADDRESSING

- Open Addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself.
- So at any point, the size of the table must be greater than or equal to the total number of keys.
- This approach is also known as closed hashing.
- This entire procedure is based upon probing. We will understand the types of probing

- ***Insert(k):*** *Keep probing until an empty slot is found. Once an empty slot is found, insert k.*
- ***Search(k):*** *Keep probing until the 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 the search may fail. So slots of deleted keys are marked specially as "deleted".*
  The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Somaiya TRUST

OPEN ADDRESSING

- Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing

Somaiya TRUST

LINEAR PROBING

What is Linear Probing?

Linear Probing is a **collision resolution technique** used in **open addressing hash tables.**

When a collision occurs (i.e., a hash index is already occupied), it searches **linearly** for the **next available slot.**

- **The simplest approach to resolve a collision is linear probing.**
- **In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision:**
- h(k, i) = [h¢(k) + i] mod m
- Where m is the size of the hash table,
- h¢(k) = (k mod m),
- and i is the probe number that varies from 0 to m–1.
- Therefore, for a given key k, first the location generated by [h¢(k) mod m] is probed because for the first time i=0. If the location is free, the value is stored in it, else the second probe generates the address of the location given by [h¢(k) + 1]mod m.
- Similarly, if the location is occupied, then subsequent probes generate the address as [h¢(k) + 2]mod m, [h¢(k) + 3]mod m, [h¢(k) + 4]mod m, [h¢(k) + 5]mod m, and so on, until a free location is found.

Somaiya TRUST

LINEAR PROBING

Example 15.5 Consider a hash table of size 10. Using linear probing, insert the keys 72, 36, 24, 63, 81, 92, and 101 into the table.

Let $h'(k) = k \bmod m$, $m = 10$

Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
−1	−1	−1	−1	−1	−1	−1	−1	−1	−1

Step 1

Key = 72

h(72, 0) = (72 mod 10 + 0) mod 10

= (2) mod 10

= 2

Since T[2] is vacant, insert key 72 at this location.

0	1	2	3	4	5	6	7	8	9
−1	−1	72	−1	−1	−1	−1	−1	−1	−1

Step 2

Key = 27

h(27, 0) = (27 mod 10 + 0) mod 10

= (7) mod 10

= 7

Since T[7] is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
−1	−1	72	−1	−1	−1	−1	27	−1	−1

Step 3

Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10

= (6) mod 10

= 6

Since T[6] is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
−1	−1	72	−1	−1	−1	36	27	−1	−1

Step 4

Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10

= (4) mod 10

= 4

Since T[4] is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
−1	−1	72	−1	24	−1	36	27	−1	−1

Step 5

Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10

= (3) mod 10

= 3

Since T[3] is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
−1	−1	72	63	24	−1	36	27	−1	−1

Step 6

Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10

= (1) mod 10

= 1

Since T[1] is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	−1	36	27	−1	−1

Step 7

Key = 92

h(92, 0) = (92 mod 10 + 0) mod 10

= (2) mod 10

= 2

Now T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for the next location. Thus probe, i = 1, this time.

Key = 92

h(92, 1) = (92 mod 10 + 1) mod 10

= (2 + 1) mod 10

= 3

Now T[3] is occupied, so we cannot store the key 92 in T[3]. Therefore, try again for the next location. Thus probe, i = 2, this time.

Key = 92

h(92, 2) = (92 mod 10 + 2) mod 10

= (2 + 2) mod 10

= 4

Now T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore, try again for the next location. Thus probe, i = 3, this time.

LINEAR PROBING

```
Key = 92
h(92, 3) = (92 mod 10 + 3) mod 10
         = (2 + 3) mod 10
         = 5
```

Since `T[5]` is vacant, insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

Step 8

```
Key = 101
h(101, 0) = (101 mod 10 + 0) mod 10
          = (1) mod 10
          = 1
```

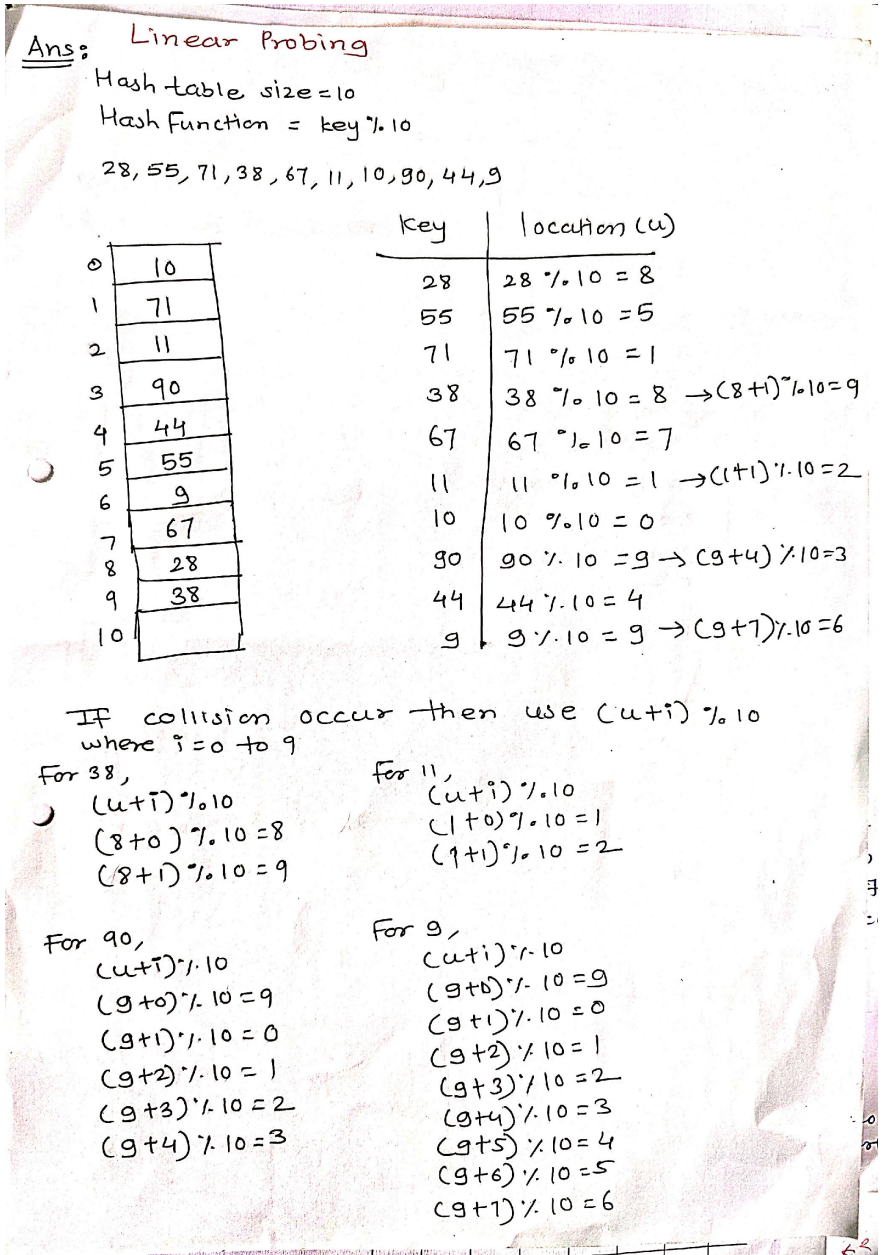
Now `T[1]` is occupied, so we cannot store the key 101 in `T[1]`. Therefore, try again for the next location. Thus probe, `i = 1`, this time.

```
Key = 101
h(101, 1) = (101 mod 10 + 1) mod 10
          = (1 + 1) mod 10
          = 2
```

`T[2]` is also occupied, so we cannot store the key in this location. The procedure will be repeated until the hash function generates the address of location 8 which is vacant and can be used to store the value in it.

Somaiya TRUST

LINEAR PROBING Example:



Ans: Linear Probing

Hash table size = 10

Hash Function = key % 10

28, 55, 71, 38, 67, 11, 10, 90, 44, 9

0	10
1	71
2	11
3	90
4	44
5	55
6	9
7	67
8	28
9	38
10	

Key	location (u)
28	28 % 10 = 8
55	55 % 10 = 5
71	71 % 10 = 1
38	38 % 10 = 8 → (8+1) % 10 = 9
67	67 % 10 = 7
11	11 % 10 = 1 → (1+1) % 10 = 2
10	10 % 10 = 0
90	90 % 10 = 9 → (9+4) % 10 = 3
44	44 % 10 = 4
9	9 % 10 = 9 → (9+7) % 10 = 6

If collision occur then use (u+i) % 10
where i = 0 to 9

For 38,
(u+i) % 10
(8+0) % 10 = 8
(8+1) % 10 = 9

For 11,
(u+i) % 10
(1+0) % 10 = 1
(1+1) % 10 = 2

For 90,
(u+i) % 10
(9+0) % 10 = 9
(9+1) % 10 = 0
(9+2) % 10 = 1
(9+3) % 10 = 2
(9+4) % 10 = 3

For 9,
(u+i) % 10
(9+0) % 10 = 9
(9+1) % 10 = 0
(9+2) % 10 = 1
(9+3) % 10 = 2
(9+4) % 10 = 3
(9+5) % 10 = 4
(9+6) % 10 = 5
(9+7) % 10 = 6

Somaiya TRUST

LINEAR PROBING

✅ Pros of Linear Probing:

Advantage	Description
✅ Simple to implement	Easy logic with just an increment to next index
✅ Cache-friendly	Accesses are in consecutive memory → better CPU cache performance
✅ No pointers required	Entire hash table is in one array, no linked lists needed
✅ Less memory overhead	No extra space required per entry (unlike chaining which uses linked lists)
✅ Good for low load factor	Works efficiently when table isn't too full (less collisions)

❌ Cons of Linear Probing:

Disadvantage	Description
❌ Clustering	**Primary clustering** – contiguous filled slots form, increasing collisions
❌ Slower at high load factor	Performance drops as table fills up (>70%)
❌ Deletion is tricky	Deleting without breaking search chain needs special handling (e.g., dummy markers)
❌ Fixed probe direction	Always moves forward – can miss available spots far behind

QUADRATIC PROBING

✅ What is Quadratic Probing?

Quadratic Probing is an **open addressing** collision resolution technique in hash tables.

When a collision occurs at index `i`, instead of checking the next slot (as in linear probing), it probes using a **quadratic formula**:

- **In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision:**
- **h(k, i) = [h¢(k) + c1 i + c2 i2] mod m**
- **where m is the size of the hash table,**
- **h¢(k) = (k mod m),**
- **i is the probe number that varies from 0 to m–1,**
- **and c1 and c2 are constants such that c1 and c2 π 0.**

- Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search.
- For a given key k, first the location generated by h¢(k) mod m is probed.
- If the location is free, the value is stored in it, else subsequent locations probed are offset by factors that depend in a quadratic manner on the probe number i.
- Although quadratic probing performs better than linear probing, in order to maximize the utilization of the hash table, the values of c1 , c2 , and m need to be constrained

Somaiya TRUST

QUADRATIC PROBING

Example 15.6 Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table. Take $c_1 = 1$ and $c_2 = 3$.

Solution

Let $h'(k) = k \bmod m$, $m = 10$

Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

We have,

$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$

Step 1 Key = 72

$h(72, 0) = [72 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [72 \bmod 10] \bmod 10$

$= 2 \bmod 10$

$= 2$

Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step 2 Key = 27

$h(27, 0) = [27 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [27 \bmod 10] \bmod 10$

$= 7 \bmod 10$

$= 7$

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step 3 Key = 36

$h(36, 0) = [36 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [36 \bmod 10] \bmod 10$

$= 6 \bmod 10$

$= 6$

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

$h(24, 0) = [24 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [24 \bmod 10] \bmod 10$

$= 4 \bmod 10$

$= 4$

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step 5 Key = 63

$h(63, 0) = [63 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [63 \bmod 10] \bmod 10$

$= 3 \bmod 10$

$= 3$

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

$h(81,0) = [81 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [81 \bmod 10] \bmod 10$

$= 81 \bmod 10$

$= 1$

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Somaiya TRUST

QUADRATIC PROBING

Step 7 Key = 101

h(101,0) = [101 mod 10 + 1 × 0 + 3 × 0] mod 10
= [101 mod 10 + 0] mod 10
= 1 mod 10
= 1

Since T[1] is already occupied, the key 101 cannot be stored in T[1]. Therefore, try again for next location. Thus probe, i = 1, this time.

Key = 101

h(101,0) = [101 mod 10 + 1 × 1 + 3 × 1] mod 10

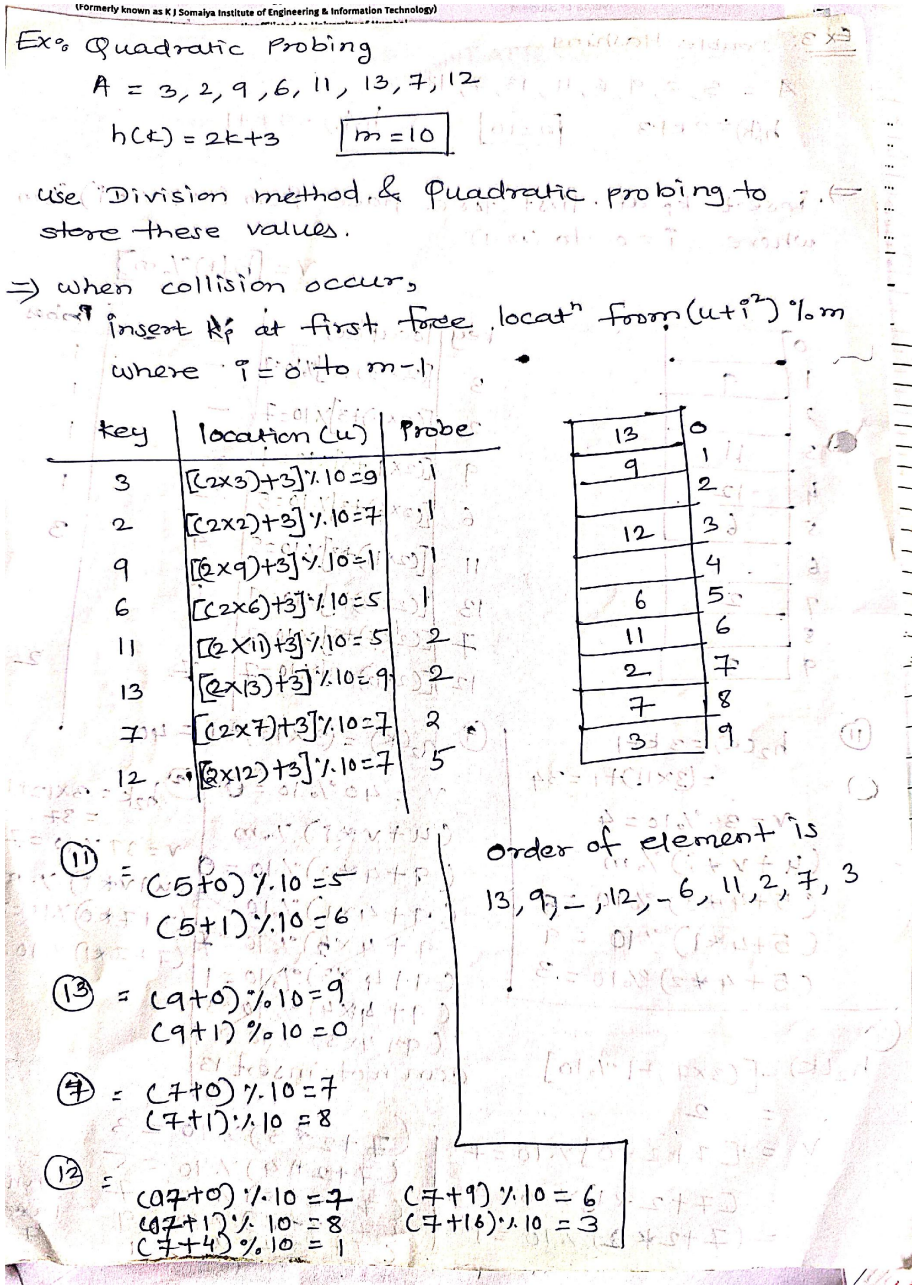
= [101 mod 10 + 1 + 3] mod 10
= [101 mod 10 + 4] mod 10
= [1 + 4] mod 10
= 5 mod 10
= 5

Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	101	36	27	–1	–1

Somaiya TRUST

QUADRATIC PROBING Example



(Formerly known as K J Somaiya Institute of Engineering & Information Technology)

Ex: Quadratic Probing

A = 3, 2, 9, 6, 11, 13, 7, 12

h(k) = 2k+3 $\boxed{m = 10}$

use Division method & Quadratic probing to store these values.

⇒ when collision occur,
insert key at first free locatn from $(u+i^2)\%m$
where i = 0 to m-1

key	location (u)	Probe
3	[(2x3)+3]%10=9	1
2	[(2x2)+3]%10=7	1
9	[(2x9)+3]%10=1	1
6	[(2x6)+3]%10=5	1
11	[(2x11)+3]%10=5	2
13	[(2x13)+3]%10=9	2
7	[(2x7)+3]%10=7	2
12	[(2x12)+3]%10=7	5

13	0
9	1
	2
12	3
	4
6	5
11	6
2	7
7	8
3	9

(11) = (5+0)%10=5
(5+1)%10=6

(13) = (9+0)%10=9
(9+1)%10=0

(7) = (7+0)%10=7
(7+1)%10=8

(12) = (07+0)%10=7 (7+9)%10=6
(07+1)%10=8 (7+16)%10=3
(7+4)%10=1

order of element is
13, 9, -, 12, -, 6, 11, 2, 7, 3

Somaiya TRUST

QUADRATIC PROBING

✅ Advantages of Quadratic Probing:

Advantage	Description
✅ Reduces Clustering	Avoids **primary clustering** seen in linear probing
✅ Better Spread of Keys	Keys spread more uniformly across the table
✅ No pointers needed	Like other open addressing methods, it uses a simple array
✅ Works well under 50% load	Good performance when the table is not too full
✅ Deterministic	Always gives a predictable and repeatable probe sequence

❌ Disadvantages of Quadratic Probing:

Disadvantage	Description
❌ Secondary Clustering	Keys with same initial hash follow same probe sequence
❌ Difficult to Guarantee Finding Empty Slot	Without a proper hash function or table size (preferably **prime**)
❌ Complicated Deletion	Like other probing methods, deletion must use special markers
❌ Harder to Implement	Requires squaring and modulus, which is more complex than linear probing
❌ Table May Not Be Fully Probed	May not reach every slot unless table size is prime

Somaiya TRUST

DOUBLE HASHING

✅ What is Double Hashing?

Double Hashing is an open addressing collision resolution technique used in hash tables.

When a collision occurs, it uses a **second hash function** to calculate the step size for probing the next slot.

- In double hashing, we use two hash functions rather than a single function.
- The hash function in the case of double hashing can be given as:
- **h(k, i) = [h1 (k) + ih2 (k)] mod m**

- where m is the size of the hash table,
- h1 (k) and h2 (k) are two hash functions given as h1 (k) = k mod m, h2 (k) = k mod m',
- i is the probe number that varies from 0 to m–1,
- and m' is chosen to be less than m. We can choose m' = m–1 or m–2.

- When we have to insert a key k in the hash table, we first probe the location given by applying [h1 (k) mod m] because during the first probe, i = 0.
- If the location is vacant, the key is inserted into it, else subsequent probes generate locations that are at an offset of [h2 (k) mod m] from the previous location.
- Since the offset may vary with every probe depending on the value generated by the second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing

Somaiya TRUST

DOUBLE HASHING

Example 15.7 Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table. Take $h_1 = (k \bmod 10)$ and $h_2 = (k \bmod 8)$.

Solution

Let $m = 10$

Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

We have,

$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$

Step 1 Key = 72

$h(72, 0) = [72 \bmod 10 + (0 \times 72 \bmod 8)] \bmod 10$
$= [2 + (0 \times 0)] \bmod 10$
$= 2 \bmod 10$
$= 2$

Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step 2 Key = 27

$h(27, 0) = [27 \bmod 10 + (0 \times 27 \bmod 8)] \bmod 10$
$= [7 + (0 \times 3)] \bmod 10$
$= 7 \bmod 10$
$= 7$

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Somaiya TRUST

Step 3 Key = 36

$h(36, 0) = [36 \bmod 10 + (0 \times 36 \bmod 8)] \bmod 10$
$= [6 + (0 \times 4)] \bmod 10$
$= 6 \bmod 10$
$= 6$

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

$h(24, 0) = [24 \bmod 10 + (0 \times 24 \bmod 8)] \bmod 10$
$= [4 + (0 \times 0)] \bmod 10$
$= 4 \bmod 10$
$= 4$

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step 5 Key = 63

$h(63, 0) = [63 \bmod 10 + (0 \times 63 \bmod 8)] \bmod 10$
$= [3 + (0 \times 7)] \bmod 10$
$= 3 \bmod 10$
$= 3$

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

$h(81, 0) = [81 \bmod 10 + (0 \times 81 \bmod 8)] \bmod 10$
$= [1 + (0 \times 1)] \bmod 10$
$= 1 \bmod 10$
$= 1$

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

DOUBLE HASHING

Step 7

Key = 92

h(92, 0) = [92 mod 10 + (0 × 92 mod 8)] mod 10
= [2 + (0 × 4)] mod 10
= 2 mod 10
= 2

Now T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for the next location. Thus probe, i = 1, this time.

Key = 92

h(92, 1) = [92 mod 10 + (1 × 92 mod 8)] mod 10
= [2 + (1 × 4)] mod 10
= (2 + 4) mod 10
= 6 mod 10
= 6

Now T[6] is occupied, so we cannot store the key 92 in T[6]. Therefore, try again for the next location. Thus probe, i = 2, this time.

Key = 92

h(92, 2) = [92 mod 10 + (2 × 92 mod 8)] mod 10
= [2 + (2 × 4)] mod 10
= [2 + 8] mod 10
= 10 mod 10
= 0

Since T[0] is vacant, insert the key 92 in T[0]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	–1	36	27	–1	–1

Step 8

Key = 101

h(101, 0) = [101 mod 10 + (0 × 101 mod 8)] mod 10
= [1 + (0 × 5)] mod 10
= 1 mod 10
= 1

Now T[1] is occupied, so we cannot store the key 101 in T[1]. Therefore, try again for the next location. Thus probe, i = 1, this time.

Key = 101

h(101, 1) = [101 mod 10 + (1 × 101 mod 8)] mod 10
= [1 + (1 × 5)] mod 10
= [1 + 5] mod 10
= 6

Now T[6] is occupied, so we cannot store the key 101 in T[6]. Therefore, try again for the next location with probe i = 2. Repeat the entire process until a vacant location is found. You will see that we have to probe many times to insert the key 101 in the hash table. Although double hashing is a very efficient algorithm, it always requires m to be a prime number. In our case m=10, which is not a prime number, hence, the degradation in performance. Had m been equal to 11, the algorithm would have worked very efficiently. Thus, we can say that the performance of the technique is sensitive to the value of m.

Somaiya TRUST

DOUBLE HASHING Example

Double Hashing..

By default = 10.

$(u + v * i) \% u$

↓

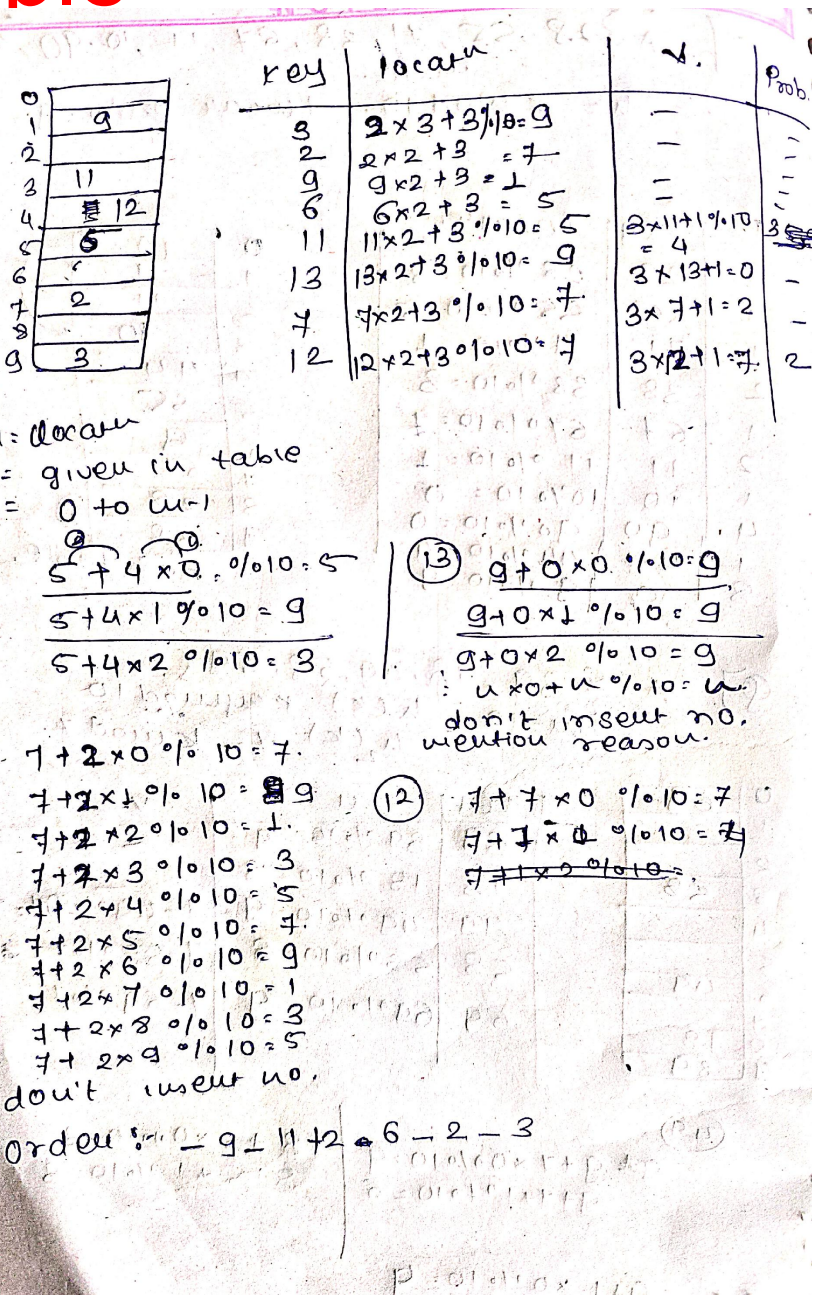
location.

$A = \{3, 12, 9, 6, 11, 13, 7, 12\}$.

Given: $u = 10$;

$h(k) = 2k + 3$

$h_2(k) = 3k + 1$.



DOUBLE HASHING

✅ Pros of Double Hashing:

Advantage	Description
✅ Minimizes Clustering	Avoids both **primary** and **secondary** clustering
✅ More Uniform Probing	Second hash provides better distribution of keys
✅ Efficient with Large Tables	Especially good when the table size is large and prime
✅ Higher Load Factor Tolerance	Performs well even at 70–80% table usage
✅ Better Performance	Fewer collisions compared to linear and quadratic probing

❌ Cons of Double Hashing:

Disadvantage	Description
❌ More Complex	Requires two good hash functions and careful design
❌ Slower Hashing	Extra computation due to second hash function
❌ Deletion Handling is Tricky	Must use dummy/deleted markers or lazy deletion
❌ h2(key) must never return 0	Otherwise, same slot is probed repeatedly (infinite loop risk)
❌ Difficult to Implement Correctly	Must choose table size and hash functions carefully

REHASHING

✅ What is Rehashing?

Rehashing is the process of **resizing a hash table** and **recomputing the positions of existing elements** using a **new hash function or new table size**.

> 🔄 It is done when the **load factor** becomes too high, causing **too many collisions** and degrading performance.

📌 Why Rehashing is Needed:

- To **reduce collisions**
- To **improve search and insert efficiency**
- To accommodate **more elements**

🔧 How Rehashing Works:

1. **Create a new hash table** with a **larger size** (usually double the current size or next prime number)
2. **Reinsert all existing elements** into the new table using the **new hash function or updated table size**
3. Discard the old table

REHASHING

🧠 Example:

Old table size = 5

Hash function: `h(key) = key % 5`

Insert keys: 10, 15, 20 → all collide at index 0

👉 Rehash:

- New table size = 10
- Recalculate hash: `h(key) = key % 10`
  - 10 → 0
  - 15 → 5
  - 20 → 0 → collision resolved by probing

Now, elements are **better distributed**.

Somaiya TRUST

COLLISION RESOLUTION BY CHAINING

✅ What is Chaining?

Chaining is a **collision resolution technique** used in **hash tables.**

When **two or more keys hash to the same index**, instead of searching for a new location, we store them **in a list (chain)** at that index.

> 🧠 **Each slot** in the hash table holds a **linked list (or chain)** of all elements that map to the same index.

📘 Example:

Suppose:

```
c

h(key) = key % 5
```

Insert keys: `10, 15, 20`

```
matlab

10 % 5 = 0
15 % 5 = 0
20 % 5 = 0
```

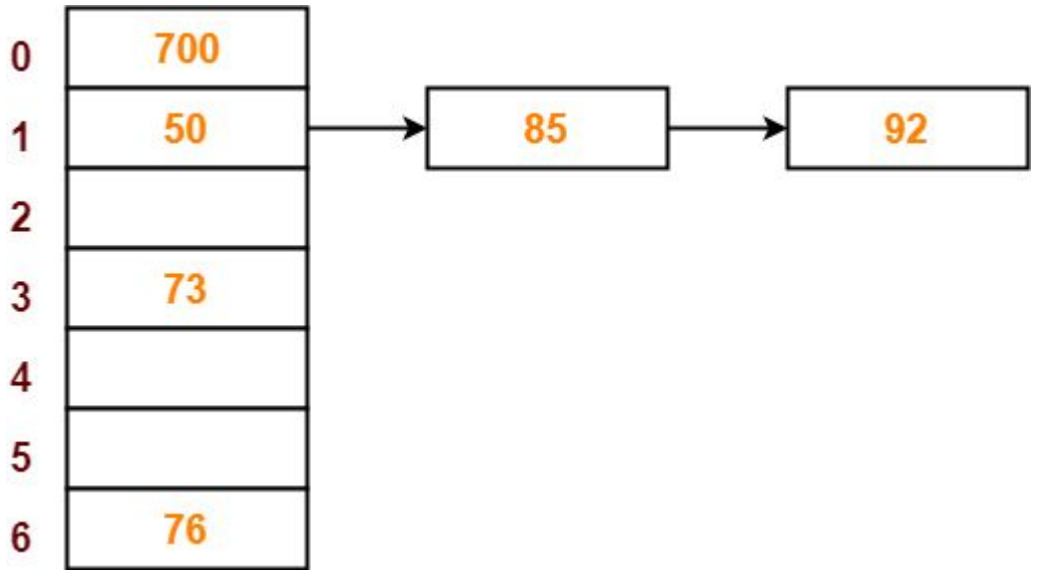
All collide at index `0`. Using chaining:

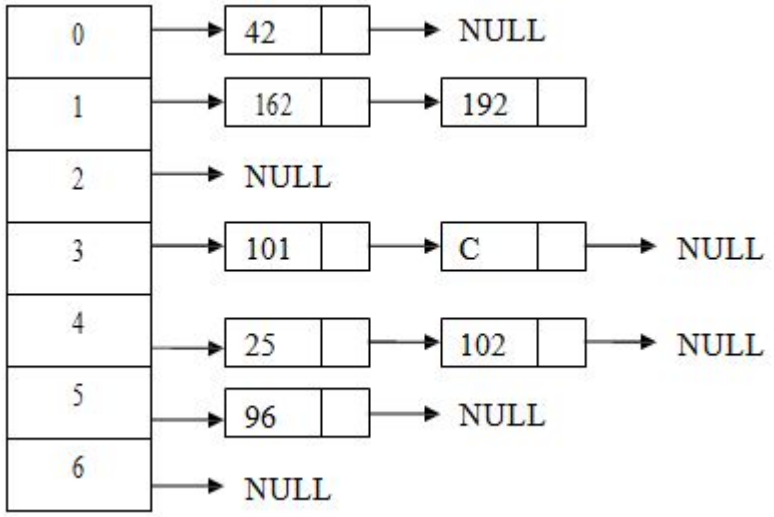
```
cpp

Index 0 → 10 → 15 → 20
Index 1 → NULL
Index 2 → NULL
...
```

Somaiya TRUST

COLLISION RESOLUTION BY CHAINING





Somaiya TRUST

SAMPLE QUESTIONS

QUESTION NO.	SAMPLE QUESTIONS MODULE 6	
1	Apply linear probing hash functions to insert values in the Hash table of size 11. Show the number of collisions that occurs in each technique. 27, 72, 63, 42, 36, 18, 29,101	
2	Consider a hash table with size = 10. Using quadratic probing, insert the keys 12, 34, 45, 36, 34, 18, 29, and 101 into the table. Take c1=1, c2= 3.	
3	Compare and Contrast Linear search and Binary search algorithm	
4	What is collision? Explain the various techniques to resolve a collision. Which technique do you think is better and why?	
5	Using the modulo division method, hash the following elements in a table of size 10. Use Linear probing to resolve the collisions. 28, 55, 71, 67, 11, 10, 90, 44	
6	Explain Linear search alogrithm with example	
7	Explain Binary search with algorithm and Example	
8	Explain the Various Hash functions with example	

Somaiya TRUST