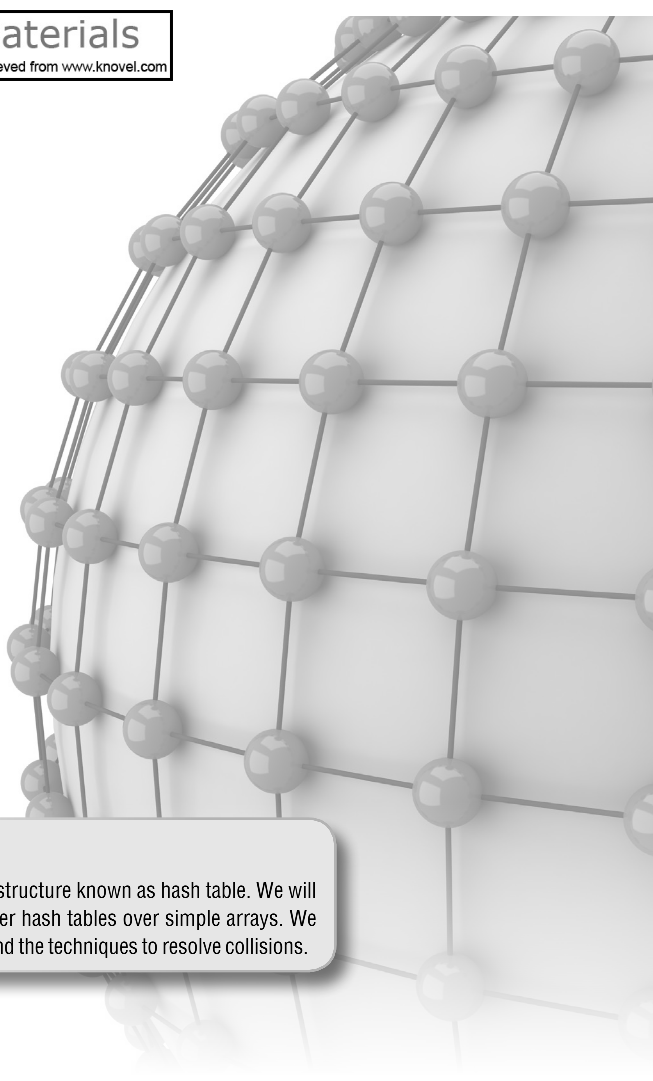

CHAPTER 15

Hashing and Collision

LEARNING OBJECTIVE

In this chapter, we will discuss another data structure known as hash table. We will see what a hash table is and why do we prefer hash tables over simple arrays. We will also discuss hash functions, collisions, and the techniques to resolve collisions.

15.1 INTRODUCTION

In Chapter 14, we discussed two search algorithms: *linear search* and *binary search*. Linear search has a running time proportional to `O(n)`, while binary search takes time proportional to `O(log n)`, where `n` is the number of elements in the array. Binary search and binary search trees are efficient algorithms to search for an element. But what if we want to perform the search operation in time proportional to `O(1)`? In other words, is there a way to search an array in constant time, irrespective of its size?

There are two solutions to this problem. Let us take an example to explain the first solution. In a small company of 100 employees, each employee is assigned an `Emp_ID` in the range 0–99. To store the records in an array, each employee's `Emp_ID` acts as an index into the array where the employee's record will be stored as shown in Fig. 15.1.

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

In this case, we can directly access the record of any employee, once we know his `Emp_ID`, because the array index is the same as the `Emp_ID` number. But practically, this implementation is hardly feasible.

Let us assume that the same company uses a five-digit `Emp_ID` as the primary key. In this case, key values will range from 00000 to 99999. If we want to use the same technique as above, we need an array of size 100,000, of which only 100 elements will be used. This is illustrated in Fig. 15.2.

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

It is impractical to waste so much storage space just to ensure that each employee's record is in a unique and predictable location.

Whether we use a two-digit primary key (`Emp_ID`) or a five-digit key, there are just 100 employees in the company. Thus, we will be using only 100 locations in the array. Therefore, in order to keep the array size down to the size that we will actually be using (100 elements), another good option is to 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.

In the second solution, the elements are not stored according to the *value* of the key. So in this case, we need a way to 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.*

15.2 HASH TABLES

Hash table is a data structure in which keys are mapped to array positions by a hash function. In the example discussed here we will use a hash function that extracts the last two digits of the key. Therefore, we map the keys to array locations or array indices. A value stored in a hash table can be searched in `O(1)` time by using a hash function which generates an address from the key (by producing the index of the array where the value is stored).

Figure 15.3 shows a direct correspondence between the keys and the indices of the array. This concept is useful when the total universe of keys is small and when most of the keys are actually used from the whole set of keys. This is equivalent to our first example, where there are 100 keys for 100 employees.

However, when the set `K` of keys that are actually used is smaller than the universe of keys `(U)`, a hash table consumes less storage space. The storage requirement for a hash table is `O(k)`, where `k` is the number of keys actually used.

In a hash table, an element with key `k` is stored at index `h(k)` and not `k`. It means a hash function `h` is used to calculate the index at which the element with key `k` will be stored. This process of mapping the keys to appropriate locations (or indices) in a hash table is called *hashing*.

Figure 15.4 shows a hash table in which each key from the set `K` is mapped to locations generated by using a hash function. Note that keys k_2 and k_6 point to the same memory location. This is known as *collision*. That is, when two or more keys map to the same memory location, a collision is said to occur. Similarly, keys k_5 and k_7 also collide. The main goal of using a hash function is to reduce the range of array indices that have to be handled. Thus, instead of having U values, we just need K values, thereby reducing the amount of storage space required.

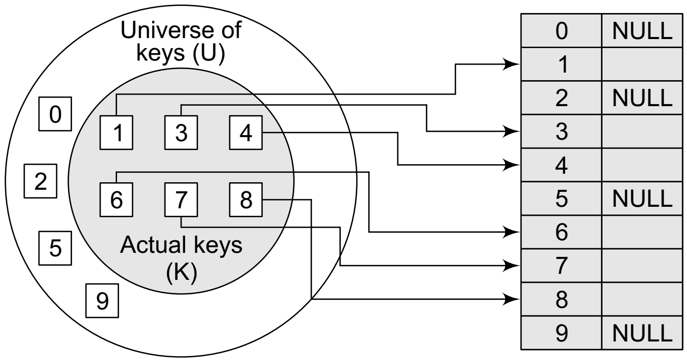

Figure 15.3 Direct relationship between key and index in the array

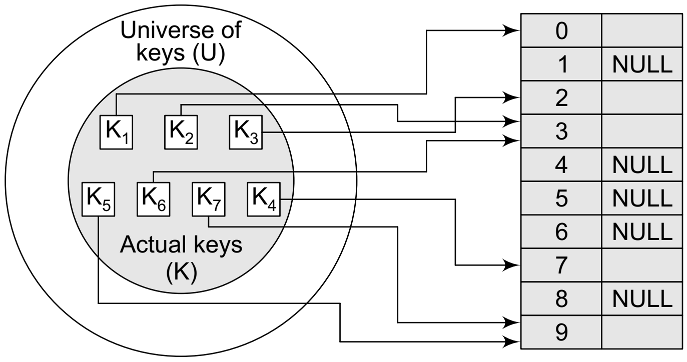

Figure 15.4 Relationship between keys and hash table index

15.3 HASH FUNCTIONS

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. In practice, there is no hash function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

In this section, we will discuss the popular hash functions which help to minimize collisions. But before that, let us first look at the properties of a good 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. For example, if binary search algorithm can search an element from a sorted table of n items with $\log_2 n$ key comparisons, then the hash function must cost less than performing $\log_2 n$ key comparisons.

Determinism A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value. However, this criteria excludes hash functions that depend on external variable parameters (such as the time of day) and on the memory address of the object being hashed (because address of the object may change during processing).

Uniformity A good hash function must map the keys as evenly as possible over its output range. This means that the probability of generating every hash value in the output range should roughly be the same. The property of uniformity also minimizes the number of collisions.

15.4 DIFFERENT HASH FUNCTIONS

In this section, we will discuss the hash functions which use numeric keys. However, there can be cases in real-world applications where we can have alphanumeric keys rather than simple numeric keys. In such cases, the ASCII value of the character can be used to transform it into its equivalent numeric key. Once this transformation is done, any of the hash functions given below can be applied to generate the hash value.

15.4.1 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
```

The division method is quite good for just about any value of `M` and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for `M`.

For example, suppose `M` is an even number then `h(x)` is even if `x` is even and `h(x)` is odd if `x` is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread the hashed values uniformly.

Generally, it is best to choose `M` to be a prime number because making `M` a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values. `M` should also be not too close to the exact powers of 2. If we have

$h(x) = x \bmod 2^k$

then the function will simply extract the lowest `k` bits of the binary representation of `x`.

The division method is extremely simple to implement. The following code segment illustrates how to do this:

```
int const M = 97; // a prime number
int h (int x)
{ return (x % M); }
```

A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

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
```

15.4.2 Multiplication Method

The steps involved in the multiplication method are as follows:

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:

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

where `(kA mod 1)` gives the fractional part of `kA` and `m` is the total number of indices in the hash table.

The greatest advantage of this method is that it works practically with any value of `A`. Although the algorithm works better with some values, the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of `A` is

```
" (sqrt5 - 1) /2 = 0.6180339887
```

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`

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

15.4.3 Mid-Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

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

The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

In the mid-square method, the same `r` digits must be chosen from all the keys. Therefore, the hash function can be given as:

$$h(k) = s$$

where `s` is obtained by selecting `r` digits from k^2.

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.

15.4.4 Folding Method

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide `k` into parts $k_1, k_2, \ldots, k_n$, 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 $k_1 + k_2 + \ldots + k_n$. The hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000, then there are 1000 locations in the hash table. To address these 1000 locations, we need at least three digits; therefore, each part of the key must have three digits except the last part which may have lesser digits.

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

15.5 COLLISIONS

As discussed earlier in this chapter, collisions occur when the hash function maps two different keys to the same location. Obviously, two records cannot be stored in the same location. Therefore, a method used to solve the problem of collision, also called *collision resolution technique*, is applied. The two most popular methods of resolving collisions are:

1. Open addressing
2. Chaining

In this section, we will discuss both these techniques in detail.

15.5.1 Collision Resolution by Open Addressing

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position. In this technique, all the values are stored in the hash table. The hash table contains two types of values: *sentinel values* (e.g., –1) and *data values*. The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, then the location is free and the data value can be stored in it. However, if the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. If even a single free location is not found, then we have an `OVERFLOW` condition.

The process of examining memory locations in the hash table is called *probing*. Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing.

Linear Probing

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:

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

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.

Note Linear probing is known for its simplicity. When we have to store a value, we try the slots: `[h′(k)] mod m`, `[h′(k) + 1]mod m`, `[h′(k) + 2]mod m`, `[h′(k) + 3]mod m`, `[h′(k) + 4]mod m`, `[h′(k) + 5]mod m`, and so no, until a vacant location is found.

Example 15.5 Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

Let `h′(k) = k mod 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.

```
      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.

Searching a Value using Linear Probing

The procedure for searching a value in a hash table is same as for storing a value in a hash table.

While searching for a value in a hash table, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. The search time in this case is given as `O(1)`. If the key does not match, then the search function begins a sequential search of the array that continues until:

- the value is found, or
- the search function encounters a vacant location in the array, indicating that the value is not present, or
- the search function terminates because it reaches the end of the table and the value is not present.

In the worst case, the search operation may have to make `n-1` comparisons, and the running time of the search algorithm may take `O(n)` time. The worst case will be encountered when after scanning all the `n-1` elements, the value is either present at the last location or not present in the table.

Thus, we see that with the increase in the number of collisions, the distance between the array index computed by the hash function and the actual location of the element increases, thereby increasing the search time.

Pros and Cons

Linear probing finds an empty location by doing a linear search in the array beginning from position `h(k)`. Although the algorithm provides good memory caching through good locality of reference, the drawback of this algorithm is that it results in clustering, and thus there is a higher risk of more collisions where one collision has already taken place. The performance of linear probing is sensitive to the distribution of input values.

As the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. In addition to this, when a new value has to be inserted into the table at a position which is already occupied, that value is inserted at the end of the cluster, which again increases the length of the cluster. Generally, an insertion is made between two clusters that are separated by one vacant location. But with linear probing, there are more chances that subsequent insertions will also end up in one of the clusters, thereby potentially increasing the cluster length by an amount much greater than one. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called *primary clustering*. To avoid primary clustering, other techniques such as quadratic probing and double hashing are used.

Quadratic 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:

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

where `m` is the size of the hash table, $h'(k) = (k \bmod m)$, `i` is the probe number that varies from `0` to `m-1`, and c_1 and c_2 are constants such that c_1 and $c_2 \neq 0$.

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 $h'(k) \bmod m$ 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 c_1, c_2, and `m` need to be constrained.

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

$$\begin{aligned} h(72, 0) &= [72 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10 \\ &= [72 \bmod 10] \bmod 10 \\ &= 2 \bmod 10 \\ &= 2 \end{aligned}$$

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

$$\begin{aligned} h(27, 0) &= [27 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10 \\ &= [27 \bmod 10] \bmod 10 \\ &= 7 \bmod 10 \\ &= 7 \end{aligned}$$

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 mod 10 + 1 X 0 + 3 X 0] mod 10
         = [36 mod 10] mod 10
         = 6 mod 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 mod 10 + 1 X 0 + 3 X 0] mod 10
         = [24 mod 10] mod 10
         = 4 mod 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 mod 10 + 1 X 0 + 3 X 0] mod 10
         = [63 mod 10] mod 10
         = 3 mod 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 mod 10 + 1 X 0 + 3 X 0] mod 10
        = [81 mod 10] mod 10
        = 81 mod 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

Step 7
```
      Key = 101
h(101,0) = [101 mod 10 + 1 X 0 + 3 X 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 X 1 + 3 X 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

Searching a Value using Quadratic Probing

While searching a value using the quadratic probing technique, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If the desired key value matches with the key value at that location, then the element is present in the hash table and the search is said to be successful. In this case, the search time is given as `O(1)`. However, if the value does not match, then the search function begins a sequential search of the array that continues until:

- the value is found, or
- the search function encounters a vacant location in the array, indicating that the value is not present, or
- the search function terminates because it reaches the end of the table and the value is not present.

In the worst case, the search operation may take `n-1` comparisons, and the running time of the search algorithm may be `O(n)`. The worst case will be encountered when after scanning all the `n-1` elements, the value is either present at the last location or not present in the table.

Thus, we see that with the increase in the number of collisions, the distance between the array index computed by the hash function and the actual location of the element increases, thereby increasing the search time.

Pros and Cons

Quadratic probing resolves the primary clustering problem that exists in the linear probing technique. Quadratic probing provides good memory caching because it preserves some locality of reference. But linear probing does this task better and gives a better cache performance.

One of the major drawbacks of quadratic probing is that a sequence of successive probes may only explore a fraction of the table, and this fraction may be quite small. If this happens, then we will not be able to find an empty location in the table despite the fact that the table is by no means full. In Example 15.6 try to insert the key 92 and you will encounter this problem.

Although quadratic probing is free from primary clustering, it is still liable to what is known as *secondary clustering*. It means that if there is a collision between two keys, then the same probe sequence will be followed for both. With quadratic probing, the probability for multiple collisions increases as the table becomes full. This situation is usually encountered when the hash table is more than full.

Quadratic probing is widely applied in the Berkeley Fast File System to allocate free blocks.

Double Hashing

To start with, double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name *double hashing*. 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:

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

where m is the size of the hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as $h_1(k) = k \bmod m$, $h_2(k) = k \bmod m'$, 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 $[h_1(k) \bmod m]$ 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 $[h_2(k) \bmod m]$ 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.

Pros and Cons

Double hashing minimizes repeated collisions and the effects of clustering. That is, double hashing is free from problems associated with primary clustering as well as secondary clustering.

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

$$\begin{aligned} h(72, 0) &= [72 \bmod 10 + (0 \times 72 \bmod 8)] \bmod 10 \\ &= [2 + (0 \times 0)] \bmod 10 \\ &= 2 \bmod 10 \\ &= 2 \end{aligned}$$

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

$$\begin{aligned} h(27, 0) &= [27 \bmod 10 + (0 \times 27 \bmod 8)] \bmod 10 \\ &= [7 + (0 \times 3)] \bmod 10 \\ &= 7 \bmod 10 \\ &= 7 \end{aligned}$$

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 mod 10 + (0 × 36 mod 8)] mod 10
         = [6 + (0 × 4)] mod 10
         = 6 mod 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 mod 10 + (0 × 24 mod 8)] mod 10
         = [4 + (0 × 0)] mod 10
         = 4 mod 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 mod 10 + (0 × 63 mod 8)] mod 10
         = [3 + (0 × 7)] mod 10
         = 3 mod 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 mod 10 + (0 × 81 mod 8)] mod 10
         = [1 + (0 × 1)] mod 10
         = 1 mod 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

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`.

Rehashing

When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently. Consider the hash table of size 5 given below. The hash function used is `h(x) = x % 5`. Rehash the entries into to a new hash table.

0	1	2	3	4
	26	31	43	17

Note that the new hash table is of 10 locations, double the size of the original table.

0	1	2	3	4	5	6	7	8	9

Now, rehash the key values from the old hash table into the new one using hash function—`h(x) = x % 10`.

0	1	2	3	4	5	6	7	8	9
	31		43			26	17		

Programming Example

1. Write a program to show searching using closed hashing.

```
#include <stdio.h>
#include <conio.h>
int ht[10], i, found = 0, key;
void insert_val();
void search_val();
void delete_val();
void display();
int main()
{
        int option;
        clrscr();
        for ( i = 0;i < 10;i++ ) //to initialize every element as ‘-1’
               ht[i] = -1;
        do
        {
               printf( "\n MENU \n1.Insert \n2.Search \n3.Delete \n4.Display \n5.Exit");
                         printf( "\n Enter your option.");
                         scanf( "%d", &option);
                         switch (option)
                         {
                             case 1:
                                  insert_val();
                                  break;
                             case 2:
                                  search_val();
                                  break;
                             case 3:
                                  delete_val();
                                  break;
                             case 4:
                                  display();
                                  break;
                             default:
                                  printf( "\nInvalid choice entry!!!\n" );
                                  break;
                         }
                }while (option!=5);
```