Jntuk396.blogspot.com

JNTUK396.BLOGSPOT.COM

CHAPTER 2

Hashing

LEARNING OBJECTIVE

In this chapter, we will discuss another data structure known as hash table and hashing techniques. 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.

2.1 INTRODUCTION TO STATIC HASHING

In large databases, it is almost impossible to search all the index values through all its level and then access the desired data. In such cases, hashing is a better alternative. Hashing calculates the direct location of data on the disk without using an index structure.

We know that 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 id `Emp_ID` acts as an index into the array where the employee's record will be stored as shown in Fig. 2.1.

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 2.1 Records of employees

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

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. 2.2.

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 2.2 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*.

Correspondingly, hashing is the process of mapping large amount of data to a smaller table by using a hash function. Hashing facilitates next level searching techniques which is efficient enough to handle large amount of data in a given collection.

Using hashing process, the hash data structure can be implemented that can store or retrieve data items in an average time disregard to the collection size.

Types of Hashing

There are two types hashing techniques—static hashing and dynamic hashing. In static hashing, when a key value is provided, the hash function always computes the same address. Also, the number of available buckets remains unchanged at all times. In contrast, dynamic hashing is a hashing technique that adds or removes data buckets dynamically and on-demand.

2.2 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).

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

Figure 2.3 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 2.4 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.

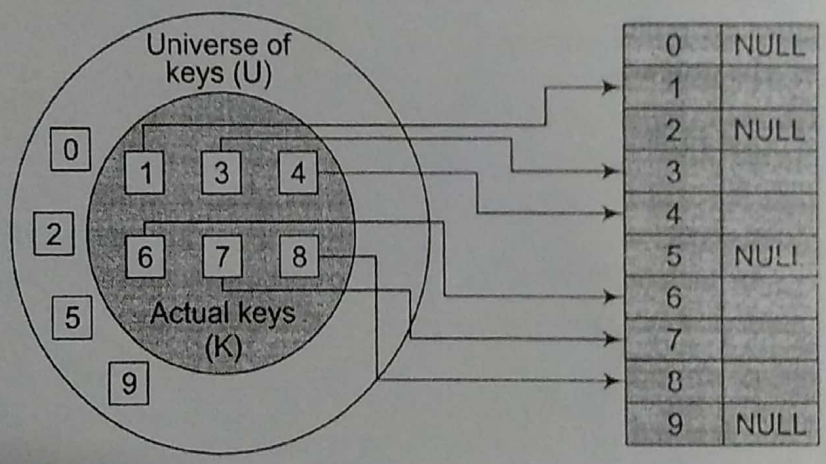


Figure 2.3 Direct relationship between key and index in the array

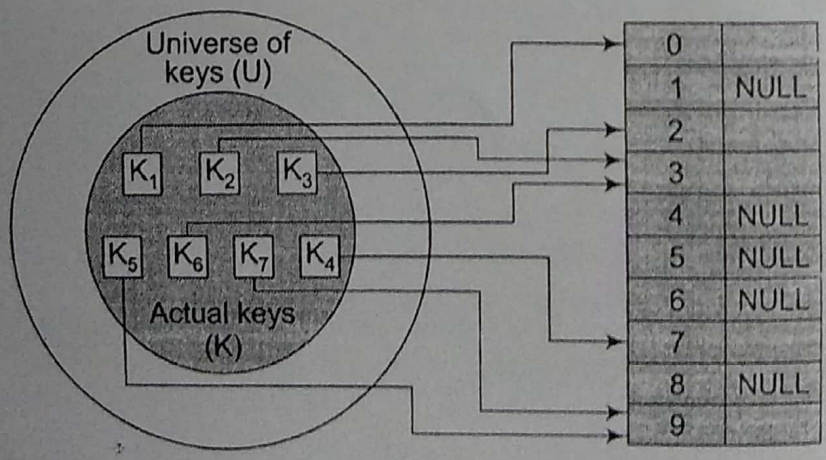


Figure 2.4 Relationship between keys and hash table index

2.3 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 se

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

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.

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

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 2.1 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
```

2.4.2 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 2.2 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`

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

2.4.3 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.

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

Example 2.3 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.

2.4.4 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 2.4 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

2.5 SECURE HASH FUNCTIONS

Secure hash functions or *secure hashing algorithms* (SHA) are a family of cryptographic functions designed to secure data. For this, it transforms the data using a hash function and an algorithm that consists of bitwise operations, modular additions, and compression functions. The hash function produces a fixed length string that is entirely different from the original. The SHA algorithms are designed to be one-way functions, meaning that once they are transformed into their respective hash values, it is virtually impossible to transform them back into the original data. The SHA-1, SHA-2, and SHA-5 form the family of SHA functions which are successively designed with increasingly stronger encryption to protect the data from attacks by the hacker.

Note SHA-0 algorithm is now obsolete due to the widely exposed vulnerabilities.

A common application of SHA is to encrypt passwords. Whenever user enters a password, a hash function generates a hashed value. This hashed value is stored in the server. The server only needs to keep a track of hash value to authenticate the user rather than the actual password.

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

The other advantage of using SHA algorithms on server side is that even if an attacker hacks the database, they will only find the hashed values and not the actual passwords. With those hashed value they will never be able to access user's accounts because a hashed value will again

Scanned by CamScanner

JNTUK396.BLOGSPOT.COM

1. Open addressing (or closed hashing)
2. Chaining (or open hashing or closed addressing)

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

2.6.1 Open Addressing Techniques

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.

2.6.1.1 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, $h'(k) = (k \bmod m)$, and `i` is the probe number that varies from `0` to `m-1`.

Therefore, for a given key `k`, first the location generated by $[h'(k) \bmod m]$ 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 $[h'(k) + 1] \bmod m$. Similarly, if the location is occupied, then subsequent probes generate the address as $[h'(k) + 2] \bmod m$, $[h'(k) + 3] \bmod m$, $[h'(k) + 4] \bmod m$, $[h'(k) + 5] \bmod m$, 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: $[h'(k)] \bmod m$, $[h'(k) + 1] \bmod m$, $[h'(k) + 2] \bmod m$, $[h'(k) + 3] \bmod m$, $[h'(k) + 4] \bmod m$, $[h'(k) + 5] \bmod m$, and so no, until a vacant location is found.

Example 2.5 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.

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

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

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

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

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.

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

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 tabl

While searching for a value in a hash table, the array index is re-computed and the key of the eleme stored at that location is compared with the value that has to be searched. If a match is found, th the search operation is successful. The search time in this case is given as O(1). If the key do 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 n present, or
- the search function terminates because it reaches the end of the table and the value is n present.

In the worst case, the search operation may have to make n-1 comparisons, and the running time the search algorithm may take O(n) time. The worst case will be encountered when after scannin 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 arra index computed by the hash function and the actual location of the element increases, thereb increasing the search time.

Pros and Cons

Linear probing finds an empty location by doing a linear search in the array beginning fro position h(k). Although the algorithm provides good memory caching through good locality reference, the drawback of this algorithm is that it results in clustering, and thus there is a high risk of more collisions where one collision has already taken place. The performance of line 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 searc increases with the size of the cluster. In addition to this, when a new value has to be inserted into th table at a position which is already occupied, that value is inserted at the end of the cluster, whic again increases the length of the cluster. Generally, an insertion is made between two cluste that are separated by one vacant location. But with linear probing, there are more chances th subsequent insertions will also end up in one of the clusters, thereby potentially increasing th cluster length by an amount much greater than one. More the number of collisions, higher th probes that are required to find a free location and lesser is the performance. This phenomenon called *primary clustering*. To avoid primary clustering, other techniques such as quadratic probin and double hashing are used.

PROGRAMMING EXAMPLE

1. Write a program to implement static hashing (using linear probing method).

```
#include<iostream>
#include<stdlib.h>
using namespace std;
#define MAX 10
class Hash
{
        private:
                int arr[MAX];
        public:
```

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

```
                Hash();
                int create(int);
                void linear_prob(int,int);
                void display();
};
Hash::Hash()
{
        int i;
        for(i=0;i<MAX;i++)
                arr[i]=-1;
}
int Hash::create(int num)
{
        int key;
        key = num%10;
        return key;
}
void Hash::linear_prob(int key,int num)
{
        int flag,i,count = 0;
        flag = 0;
        if(arr[key]==-1)//if the location indicated by hash key is empty
                arr[key]=num;//place the number in the hash table
        else
        {
                for(i=key+1;i<MAX;i++)//moving linearly down
                if(arr[i]==-1) // searching for empty location
                {
                        arr[i]=num; //placing the number at empty location
                        flag=1;
                        break;
                }
//From key position to the end of array we have searched empty location
//and now we want to check empty location in the upper part of the array
                for(i=0;i<key&&flag==0;i++)//array from 0th to keyth loaction will
be scanned
                        if(arr[i]==-1)
                        {
                                arr[i]=num;
                                flag=1;
                        break;
                        }
        } //outer else
}
void Hash::display()
{
        int i;
        cout<<"\n The Hash Table is..."<<endl;
        cout<<"\n INDEX \t VALUE";
        for(i=0;i<MAX;i++)
```

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

```
                cout<<"\n "<<i<<" "<<arr[i];
}
main()
{
        int num,key;
        Hash ht;
        do
        {
                cout<<"\n Enter The Number : ";
                cin>>num;
                key = ht.create(num); //returns hash key
                ht.linear_prob(key,num);
        }while(num != -1);
        ht.display();
}
```

Output

```
Enter The Number : 56
Enter The Number : 78
Enter The Number : 98
Enter The Number : -1
The Hash Table is...
INDEX   VALUE
0        -1
1        1
2        11
3        21
4        31
5        45
6        56
7        -1
8        78
9        98
```

2.6.1.2 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.

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

Example 2.6 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

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

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

Step 5: Key = 63

h(63, 0) = [63 mod 10 + 1 × 0 + 3 × 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 × 0 + 3 × 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 × 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 next location. Thus probe, i = 1, this time.

Key = 101

h(101,1) = [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

Searching a Value using Quadratic Probing

While searching a value using the quadratic probing technique, the array index is re-comp and the key of the element stored at that location is compared with the value that has to be searc If the desired key value matches with the key value at that location, then the element is pre in the hash table and the search is said to be successful. In this case, the search time is give O(1). However, if the value does not match, then the search function begins a sequential se 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 i not present, or

JNTUK396.BLOGSPOT.COM

Scanned by CamScanner

- 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.

2.6.1.3 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 2.7 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 $h_2 = (k \bmod 8)$.

JNTUK396.BLOGSPOT.COM Scanned by CamScanner

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,

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

Scanned by CamScanner

$= [3 + (0 \times 7)] \bmod 10$

$= 3 \bmod 10$

$= 3$

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

Step 7: Key = 92

$h(92, 0) = [92 \bmod 10 + (0 \times 92 \bmod 8)] \bmod 10$

$= [2 + (0 \times 4)] \bmod 10$

$= 2 \bmod 10$

$= 2$

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

$h(92, 1) = [92 \bmod 10 + (1 \times 92 \bmod 8)] \bmod 10$

$= [2 + (1 \times 4)] \bmod 10$

$= (2 + 4) \bmod 10$

$= 6 \bmod 10$

$= 6$

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

$h(92, 2) = [92 \bmod 10 + (2 \times 92 \bmod 8)] \bmod 10$

$= [2 + (2 \times 4)] \bmod 10$

$= [2 + 8] \bmod 10$

$= 10 \bmod 10$

$= 0$

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

$h(101, 0) = [101 \bmod 10 + (0 \times 101 \bmod 8)] \bmod 10$

$= [1 + (0 \times 5)] \bmod 10$

$= 1 \bmod 10$

$= 1$

Scanned by CamScanner

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.

2.6.1.4 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		

Difference between Double Hashing and Rehashing

While double hashing means applying two different functions on the same key at a time, rehashing on the other hand, means applying the same hashing function again and again on a key to generate a unique mapping value.

PROGRAMMING EXAMPLE

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

```
#include<iostream>
using namespace std;
int ht[10], i, found = 0, key;
```

Scanned by CamScanner

```
void insert_val();
void search_val();
void delete_val();
void display();
int main()
{
        int option;
        for (i=0;i<10;i++ ) //to initialize every element as -1
        ht[i] = -1;
        do
        {
                cout<<"\n MENU \n 1.Insert \n 2.Search \n 3.Delete \n 4.Display \n
5.Exit";
                cout<<"\n Enter your option.";
                cin>>option;
                switch (option)
                {
                        case 1:
                                insert_val();
                                break;
                        case 2:
                                search_val();
                                break;
                        case 3:
                                delete_val();
                                break;
                        case 4:
                                display();
                                break;
                        default:
                                cout<<"\n Invalid choice entry!!!\n" ;
                                break;
                }
        }while (option!=5);
        return 0;
}
void insert_val()
{
        int val, f = 0;
        cout<<"\n Enter the element to be inserted : ";
        cin>>val;
        key = val%10;
        if(ht[key] == -1)
        {
                ht[key] = val;
        }
        else
        {
                for ( i = key + 1;i %10 != key ;i++ )
                {
                        if(ht[i] == -1)
                        {
                                ht[i] = val;
                                break;
                        }
                }
                if (i==key)
                        cout<<"\n List is full";
```

Scanned by CamScanner

```
        }
}
void display()
{
        for (i = 0;i < 10;i++)
                cout<<"\t"<<ht[ i ];
}
void search_val()
{
        int val, flag = 0;
        cout<<"\n Enter the element to be searched :: ";
        cin>>val;
        key = val%10;
        if ( ht[ key ] == val )
                flag = 1;
        else
        {
                for (i = key + 1;i < 10;i++)
                {
                        if(ht[i] == val)
                        {
                                flag =1;
                                key = i;
                                break;
                        }
                }
        }
        if (flag == 0)
        {
                for (i = 0;i < key;i++)
                {
                        if (ht[ i ] == val)
                        {
                                flag = 1;
                                key = i;
                                break;
                        }
                }
        }
        if (flag == 1)
        {
                found=1;
                cout<<"\n The item searched was found at position "<<key + 1 ;
        }
        else
        {
                key = -1;
                cout<<"\n The item searched was not found in the hash table";
        }
}
void delete_val()
{
        search_val();
        if (found==1)
        {
                if(key!=-1)
                {
```

Scanned by CamScanner

```
                    cout<<"\n The element deleted is "<< ht[ key ];
                    ht[key] = -1;
                }
        }
}
```

Output

```
MENU
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your option.1

Enter the element to be inserted : 1

MENU
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your option.4
-1      1       -1      -1      -1      -1      -1      -1      -1      -1
MENU
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your option.1

Enter the element to be inserted : 21

MENU
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your option.4
-1      1       21      -1      -1      -1      -1      -1      -1      -1
MENU
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your option.1

Enter the element to be inserted : 6

MENU
1.Insert
2.Search
```

Scanned by CamScanner

and deleting takes more time because these operations scan the entries of the selected location for the desired key.

In the worst case, searching a value may take a running time of `O(n)`, where n is the number of key values stored in the chained hash table. This case arises when all the key values are inserted into the linked list of the same location (of the hash table). In this case, the hash table is ineffective.

Scanned by CamScanner

Example 2.8 Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use h(k) = k mod m.

Solution In this case, m=9. Initially, the hash table can be given as:

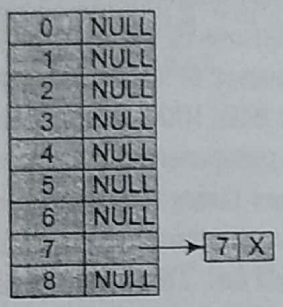
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

Step 1: Key = 7

h(k) = 7 mod 9

= 7

Create a linked list for location 7 and store the key value 7 in it as its only node.

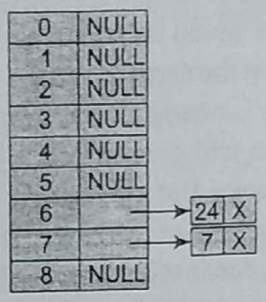


Step 2: Key = 24

h(k) = 24 mod 9

= 6

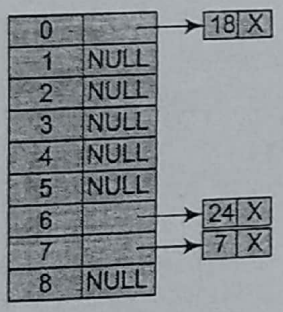
Create a linked list for location 6 and store the key value 24 in it as its only node.



Step 3: Key = 18

h(k)= 18 mod 9 = 0

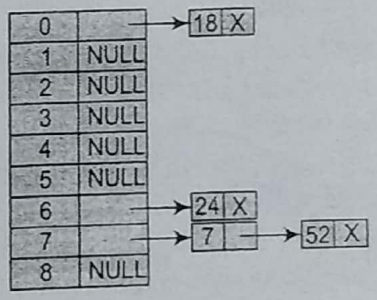
Create a linked list for location 0 and store the key value 18 in it as its only node.



Step 4: Key = 52

h(k) = 52 mod 9 = 7

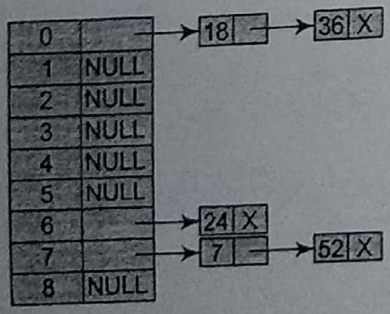
Insert 52 at the end of the linked list of location 7.



Step 5: Key = 36

h(k)= 36 mod 9 = 0

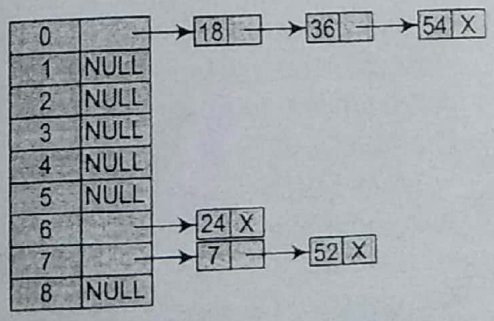
Insert 36 at the end of the linked list of location 0.



Step 6: Key = 54

h(k) = 54 mod 9 = 0

Insert 54 at the end of the linked list of location 0.



Step 7: Key = 11

h(k)= 11 mod 9 = 2

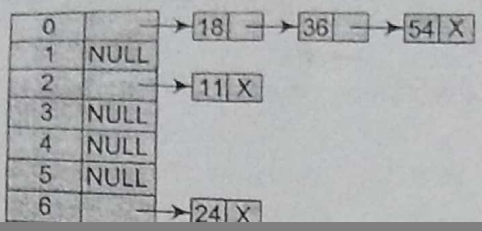
Create a linked list for location 2 and store the key value 11 in it as its only node.

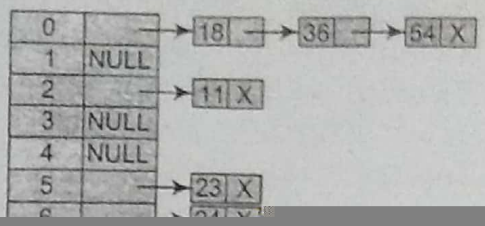
Step 8: Key = 23

h(k) = 23 mod 9 = 5

Create a linked list for location 5 and store the key value 23 in it as its only node.

Scanned by CamScanner





Scanned by CamScanner

2.6.3 Comparison between Open Hashing and Closed Hashing Techniques used for Resolving Collisions

We have studied open addressing (or closed hashing) and chaining (or open hashing) methods for resolving collisions (or overflow conditions) in the previous sections. Both the techniques are useful and widely applied in real-world applications. While open hashing is simpler to implement, closed hashing is more efficient for storing a small number of records. Table 2.2 highlights the differences between the two techniques.

Table 2.2 Differences between open hashing and closed hashing techniques

Open Hashing	Closed Hashing
1. It is also known as closed addressing.	1. It is also known as open addressing.
2. Keys are stored in linked lists attached to cells of a hash table.	2. Keys are stored directly at an index in the hash table. It does not uses any linked list.
3. Each index maintains a list of keys that are to be stored at the same index and the keys are linked to each other.	3. The index at which the key is actually stored may vary depending on the keys already stored in the hash table.
4. For example, if two keys hash to the same index, then they both are inserted in the list maintained at the same index entry.	4. For example, if element 12 is hashed to index 5 and if the index already has another value stored in it, then 12 will be inserted at any other free location.
5. Efficient for large number of records.	5. Efficient for small number of records.

2.7 DYNAMIC HASHING

2.7.1 Motivation for Dynamic Hashing

Till now, we have been using static hashing techniques only. In static hashing, following operations can be performed.

- *Insertion:* When a new data item has to be inserted, the static hash function h uses the key to compute the bucket address where the record will be stored. Therefore, bucket address h(K).
- *Search:* To access a particular record, the same hash function is used to compute the address of the bucket where the data is stored.
- *Delete:* The delete operation is simply a search operation followed by a deletion operation.

In static hashing, we have seen that in order to handle overflow situations (or collision), overflow chaining or linear probing is used. The condition of bucket-overflow is known as *collision*. This is a fatal state for any static hash function. In this case, open hashing or overflow chaining can be used.

In open hashing, when buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. (Refer to Fig. 2.7)

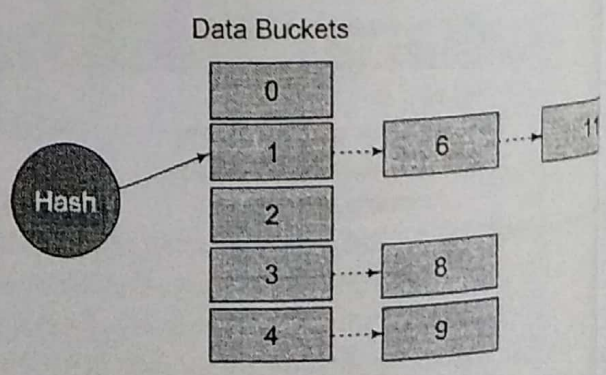


Figure 2.7 Open hashing

Scanned by CamScanner

In closed hashing, when a hash function generates an address at which data is already stored, the next free bucket is allocated to it. (Refer to Fig. 2.8)

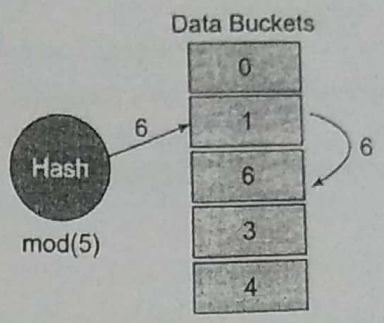


Figure 2.8 Closed hashing

The main issue with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Hence, dynamic hashing is used. *Dynamic hashing* is flexible enough to allow data buckets to be added and removed dynamically as per the demand. There are two types of dynamic hashing—dynamic hashing with directories and dynamic hashing without directories, which are discussed as follows.

2.7.2 Dynamic Hashing (or Extended Hashing) Using Directories

To understand, the concept of dynamic hashing, look at the Fig. 2.9. In dynamic hashing, only a portion of a bit map (known as the hash index) is used to build a directory. This hash index is then used to calculate the bucket addresses. If there are n bits in the hash index, then it can address 2^n different buckets. When all the buckets are full, an overflow occurs and the number of bits (therefore, the number of buckets) are increased. You can perform the following operations on a dynamic hash table.

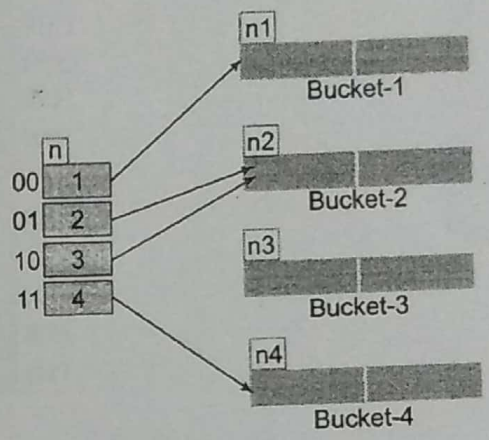


Figure 2.9 Dynamic hashing

- *Querying:* Look at the hash index and use it to calculate the bucket address.
- *Update:* To update data stored, first query the hash table and then update the data value.
- *Deletion:* To delete data, query to locate the data and then delete it.
- *Insertion:* Compute the address of the bucket where data should be inserted. If the bucket has space, add data to it. However, if the bucket is already full, then split the bucket into two and follow the steps given below. (Refer to Fig. 2.10).
  - Add bit(s) to the hash value.
  - Depending on the number of bits added in the hash value, the directory size grows accordingly. That is, the size of the directory grows from 2^k to 2^{k+1} if only one bit is increased.
  - Recompute the hash function.

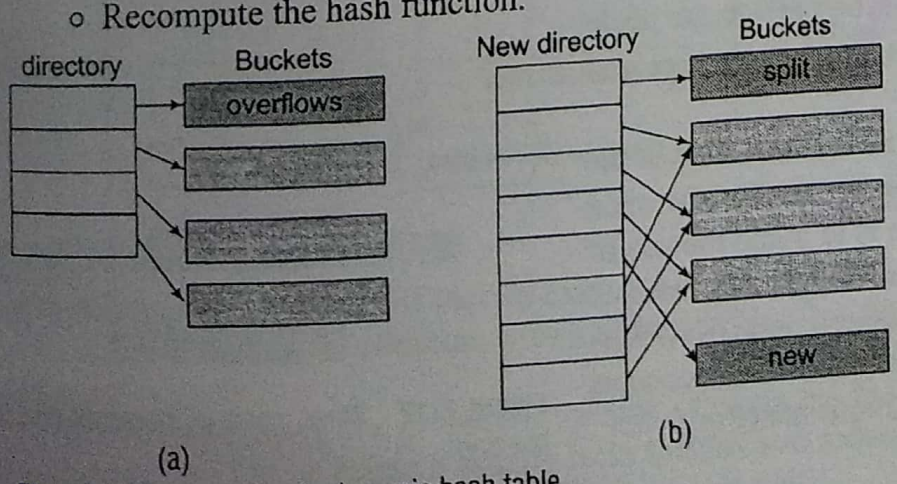


Figure 2.10 Insertion in dynamic hash table

Scanned by CamScanner

Note Hashing is usually performed when data is discrete and random. If data is organized in some order and the queries require a certain range of data, indexing should be preferred.

Note that each bucket can hold only a fixed number of items. Dynamic hashing accesses data stored in buckets indirectly through an index. Though this hashing technique imposes some performance overhead, but space efficiency is maintained and the overhead is also acceptably low.

Note Hashing algorithms have high complexity than indexing. All hash operations are done in constant.

To better understand the concept of dynamic hashing using directories, consider the example given below.

Example 2.9 Refer to the keys and hash values given in the table (Fig. 2.11(a)) and insert new key-value pairs.

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C2	110 010
C3	110 011

Figure 2.11(a)

Solution

We know that size of the directory is 2^r, where r is the number of bits used from the hash value to index the directory.

Initially, let r = 2. Therefore, size of the directory = r^2 = 4.

Let h(k,p) be defined as p least significant bits in h(k). For example,

```
h(C1) = 110001
h(C1, 2) = 01
h(C1, 4) = 0001
```

Now, look at the Fig. 2.11(a) which shows the directory and buckets corresponding to the keys and hash values given in the table.

1. *Let us insert a new key value pair in the table – C5(110101)*

Since r = 2 and h(C5,2) = 01, we need to follow the 01th entry in the directory. But this entry already has A1 and B1 stored in the bucket. So, overflow of bucket takes place. (Fig. 2.11(b))

To resolve this overflow situation, let r = 3. Correspondingly, make the size of the directory 2^3 (i.e., with 8 entries). Also, duplicate the pointers to the new half. (Fig. 2.11(c))

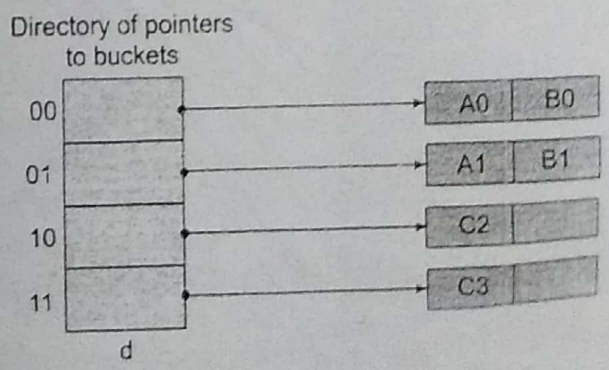


Figure 2.11(b)

Scanned by CamScanner

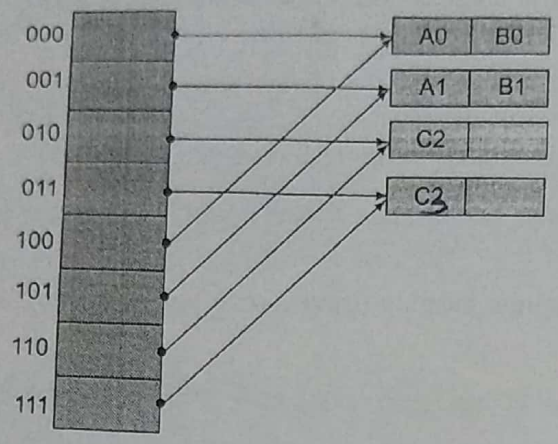


Figure 2.11(c)

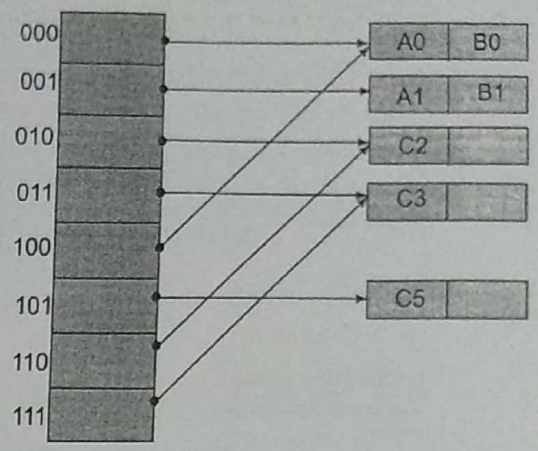


Figure 2.11(d)

Now, rehash `A1`, `B1` and inset `C5` using the new hash function `h(C5, 3)`.

2. *Let us insert a new key value pair in the table – C1(110001)*

Now, `r = 3` and `h(C1, 3) = 001`. We need to follow the 001th entry in the directory. But this entry already has `A1` and `B1` stored in the bucket. So, the bucket again overflows.

To resolve this overflow situation, let `r = 4`. Correspondingly, make the size of the directory 2^4 (i.e., with 16 entries). Also duplicate the pointers to the new half.

While inserting the new record, just take care of one thing. If the bucket is full, then before inserting we need to split the bucket. For example, consider Fig. 2.11(e) given below. Even after increasing the size of the directory, the buckets in the 0001th entry are full. So, we have to split the bucket to accommodate the new record.

Now, see that after splitting the buckets at 0001th, we have `A1` and `C1` in the initial buckets and `B1` moved to the bucket corresponding to the next available 0001th entry in the directory, as shown in Fig. 2.11(f). Why did we move `B1`? The reason to this is that, if there are more than one pointer to a bucket, then allocate a new bucket. Move the second half of the bucket entries to the new bucket.

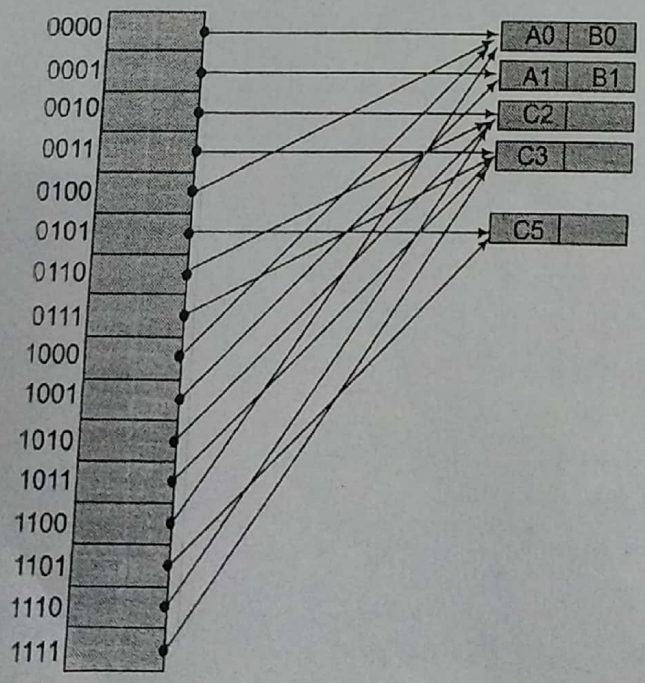


Figure 2.11(e)

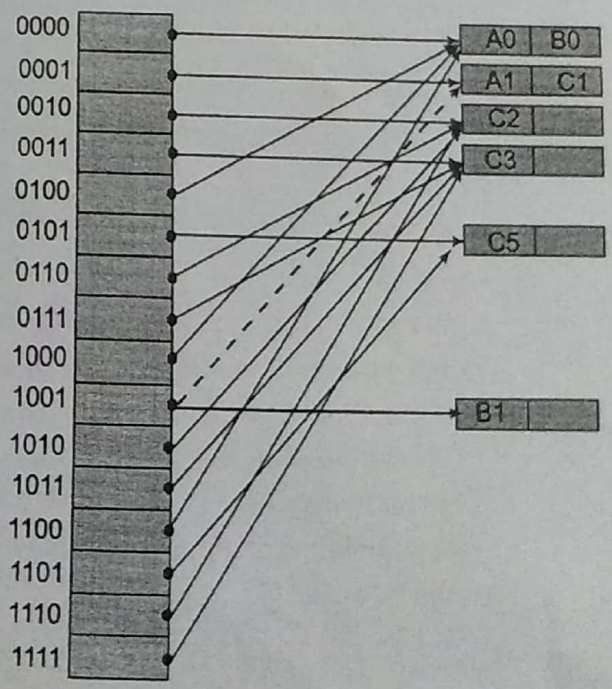


Figure 2.11(f)

Scanned by CamScanner

3. *Let us insert a new key value pair in the table – C4(110100)*

Now, r = 4 and h(C4, 4) = 0100. We need to follow the 0100th entry in the directory. A new bucket can be easily assigned to this entry without expanding the size of the directory (Refer to Fig. 2.11(g)).

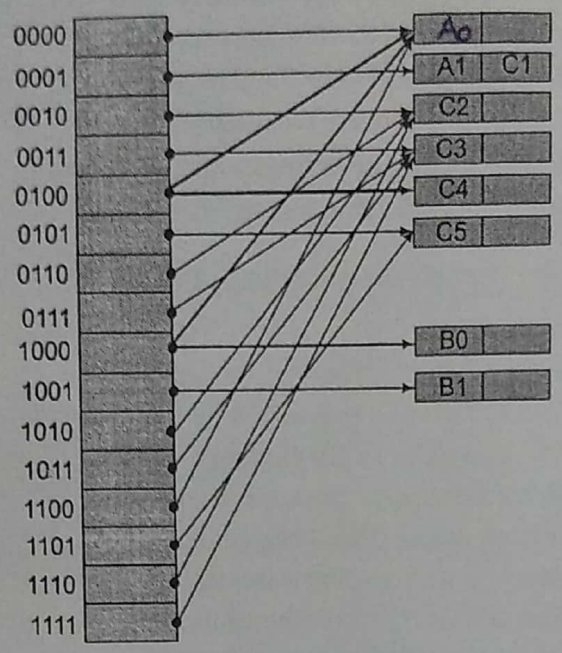


Figure 2.11(g)

Advantages

- Only doubling directory rather than the whole hash table used in static hashing.
- Only rehash the entries in the buckets that overflows.

2.7.3 Directory-less Dynamic Hashing

As the name suggests, directory-less dynamic hashing performs hashing without using directories. In this technique, we assume that there is a large memory that is big enough to store all records.

Whenever an overflow occurs, the memory space is increased resulting in wasting too much memory. Therefore, best efforts are made to avoid overflow. We only add one page and rehash the table.

The directory-less technique uses one of the traditional rehash functions, just like the open addressing. Its main aim is to avoid the use of directories as was done in the previous method. For this, the directory-less technique deals with overflow pages temporarily. The main concepts or steps used in this technique are (Fig. 2.12):

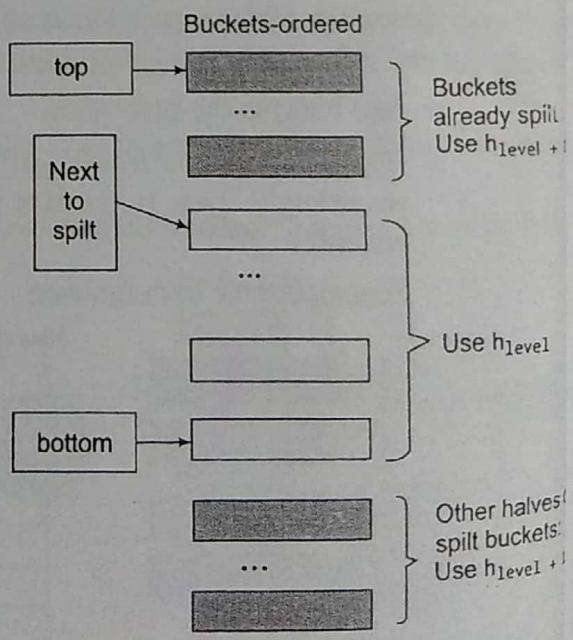


Figure 2.12 Steps used in directory-less dynamic hashing

Scanned by CamScanner

- Use a hash function to calculate the address.
- Split the bucket whenever it overflows.
- The buckets are always split in linear order. Eventually bucket with overflow pages will also be split.
- Buckets are split in rounds from the top level to the bottom level in the following manner.
  - A new round begins as soon as the bottom bucket has been split.
  - Next bucket splits when some condition is met.
  - Any bucket can have overflow pages.

For example, if we have records a0, b0 stored in page 00, c2 stored in page 01, a1 and b1 stored in page 10 and finally c3 stored in page 11 refer to (Fig. 2.13(a)). The moment we try to enter record c5, an overflow occurs in page 01. So, we split the first page that is 00. Just observe that it has now one additional bit for indexing (refer to Fig. 2.13(b)).

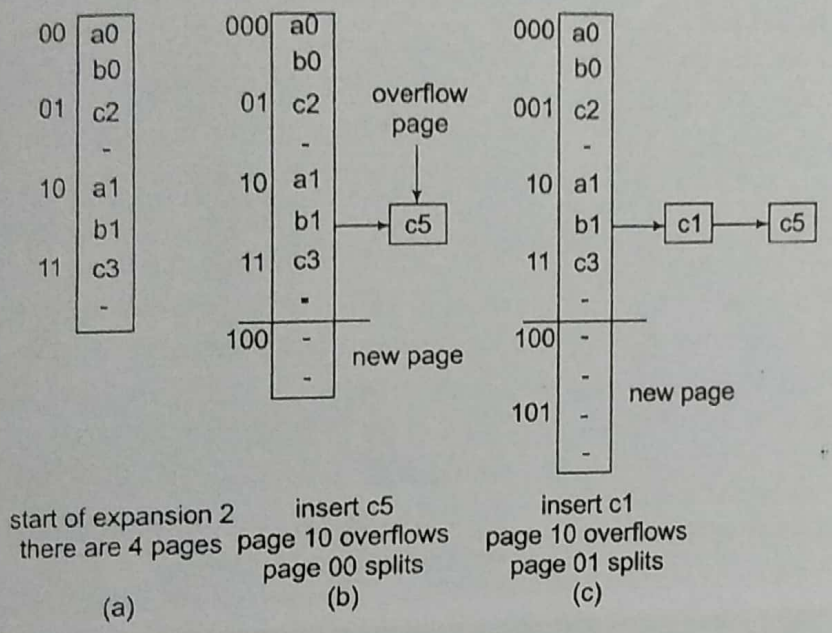
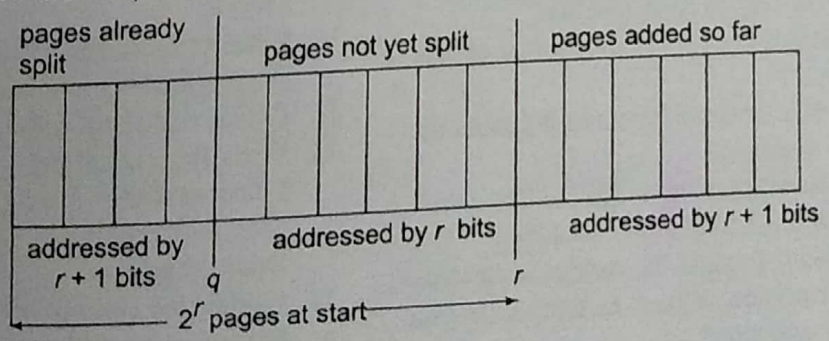


Figure 2.13 Exmple for directory-less dynamic hashing

Again, if we insert another record c1 in page 10, then overflow occurs and now it is the time page 01 to be split (refer to Fig. 2.13(c)).

Suppose we are at r, there are 2^r pages indexed by r bits as shown below.



2.8 PROS AND CONS OF HASHING

One advantage of hashing is that no extra space is required to store the index as in the case of other data structures. In addition, a hash table provides fast data access and an added advantage of rapid updates.

On the other hand, the primary drawback of using the hashing technique for inserting and retrieving data values is that it usually lacks locality and sequential retrieval by key. This makes insertion and retrieval of data values even more random.

Scanned by CamScanner

All the more, choosing an effective hash function is more of an art than a science. It is not uncommon (in open-addressed hash tables) to create a poor hash function.

2.9 APPLICATIONS OF HASHING

Hash tables are widely used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information. A few typical examples where hashing is used are given here.

Hashing is used for database indexing. Some database management systems store a separate file known as the index file. When data has to be retrieved from a file, the key information is first searched in the appropriate index file which references the exact record location of the data in the database file. This key information in the index file is often stored as a hashed value.

In many database systems, file and directory hashing is used in high-performance file systems. Such systems use two complementary techniques to improve the performance of file access. While one of these techniques is caching which saves information in the memory, the other is hashing which makes looking up the file location in the memory much quicker than most other methods.

Hashing technique is used to implement compiler symbol tables in C++. The compiler uses a symbol table to keep a record of all the user-defined symbols in a C++ program. Hashing facilitates the compiler to quickly look up variable names and other attributes associated with symbols. Hashing is also widely used for Internet search engines.

Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects. Hash Functions are also used in various algorithms in order to make their computing faster and safer.

Real World Applications of Hashing

CD Databases For CDs, it is desirable to have a world-wide CD database so that when users put their disk in the CD player, they get a full table of contents on their own computer's screen. These tables are not stored on the disks themselves, i.e., the CD does not store any information about the songs, rather this information is downloaded from the database. The critical issue to solve here is that CDs have no ID numbers stored on them, so how will the computer know which CD has been put in the player? The only information that can be used is the track length, and the fact that every CD is different.

Basically, a big number is created from the track lengths, also known as a 'signature'. This signature is used to identify a particular CD. The signature is a value obtained by hashing. For example, a number of length of 8 or 10 hexadecimal. digits is made up; the number is then sent to the database, and that database looks for the closest match. The reason being that track length may not be measured exactly.

Drivers Licenses/Insurance Cards Like our CD example, even the driver's license numbers or insurance card numbers are created using hashing from data items that never change: date of birth, name, etc.

Sparse Matrix A sparse matrix is a two-dimensional array in which most of the entries contain a 0. That is, in a sparse array there are very few non-zero entries. Of course, we can store 2D array as it is, but this would lead to sheer wastage of valuable memory. So another possibility is to store the non-zero elements of the spare matrix as elements in a 1D array. That is by using hashing, we can store a two-dimensional array in a one-dimensional array. There is a one-to-one correspondence between the elements in the sparse matrix and the elements in the array. This concept is clearly visible in Fig. 2.14.

If the size of the sparse matrix is `n` × `n`, and there are `N` non-zero entries in it, then from the coordinates `(i,j)` of a matrix, we determine an index `k` in an array by a simple calculation. Thus, we have `k=h(i,j)` for some function `h`, called a hash function.

Scanned by CamScanner

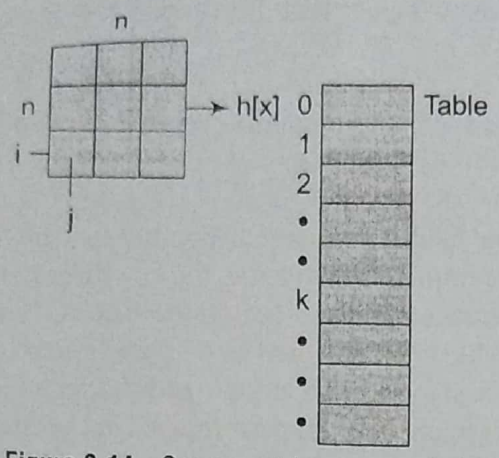


Figure 2.14 Sparse matrix

The size of the 1D array is proportional to `N`. This is far better from the size of the sparse matrix that required storage proportional to `n × n`. For example, if we have a triangular sparse matrix `A`, then an entry `A[i, j]` can be mapped to an entry in the 1D array by calculating the index using the hash function `h(i, j) = i(i-1)/2 + j`.

File Signatures File signatures provide a compact means of identifying files. We use a function, `h[x]`, the file signature, which is a property of the file. Although we can store files by name, signatures provide a compact identity to files.

Since a signature depends on the contents of a file, if any change is made to the file, then the signature will change. In this way, the signature of a file can be used as a quick verification to see if anyone has altered the file, or if it has lost a bit during transmission. Signatures are widely used for files that store marks of students.

Game Boards In the game board for tic-tac-toe or chess, a position in a game may be stored

Scanned by CamScanner