

## 8 Searching

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful. The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.

There are several types of searching techniques; one has some advantage(s) over other. Following are the three important searching techniques:

1. Linear or Sequential Searching
2. Binary Searching
3. Hashing

The records that are stored in a list being searched must conform to the following minimal standards:

- Every record is associated to a key.
- Keys can be compared for equality or relative ordering.
- Records can be compared to each other or to keys by first converting records to their associated keys

### 8.1 Linear or Sequential Search

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

For **example**, if an array A[] is declared and initialized as, `int A[] = { 10, 8, 2, 7, 3, 4, 9, 1, 6, 5 }`; and the value to be searched is `VAL = 7`, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, `POS = 3` (index starting from 0).

#### Algorithm for Linear Search

```

LINEAR_SEARCH(A, n, val)
    Step 1: Initialize SET pos = -1 and I = 0
    Step 2: Repeat Step 3 while I < n
    Step 3: If A[I] = val
                SET pos = I
                Print pos
                Go to Step 5
        [End of if]
        I = I + 1
    [End of Loop]
    Step 4: if pos = -1
        Print "Value is not present in the array"
    Step 5: Exit
  
```

#### Source Code

```

#include <stdio.h>
#include <conio.h>
#define size 20
void Linear_search(int [],int, int);
int main()
{
    int A[size],num,i,n;
    printf("\n Enter the number of elements in the array : ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i=0;i<n;i++)
        scanf("%d", &A[i]);
  
```



```

printf("\n Enter the number that has to be searched : ");
scanf("%d", &num);
Linear_search(A,n,num);
getch();
return 0;
}
void Linear_search(int A[],int n, int val)
{
    int i,pos=-1;
    for(i=0;i<n;i++)
    {
        if(A[i] == val)
        {
            pos=i;
            printf("\n %d is found in the array at position= %d", val,i+1);
        }
    }
    if (pos == -1)
        printf("\n %d does not exist in the array", val);
}

```

### ✚ Complexity of Linear Search Algorithm

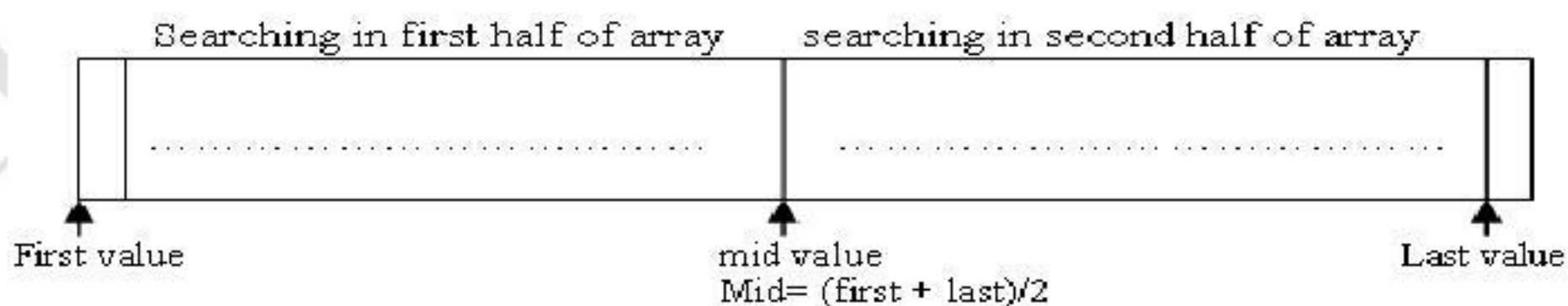
Linear search executes in  $O(n)$  time where  $n$  is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases,  $n$  comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array.

## 8.2 Binary Search

Binary search is an extremely efficient algorithm. This search technique searches the given item in minimum possible comparisons. To do this binary search, first we need to sort the array elements. The logic behind this technique is given below:

- ✓ First find the middle element of the array
- ✓ Compare the middle element with an item.
- ✓ There are three cases:
  - If it is a desired element then search is successful
  - If it is less than desired item then search only the first half of the array.
  - If it is greater than the desired element, search in the second half of the array.

Repeat the same process until element is found or exhausts in the search area. In this algorithm every time we are reducing the search area.



### Running example:

Take input array  $a[] = \{2, 5, 7, 9, 18, 45, 53, 59, 67, 72, 88, 95, 101, 104\}$



For key = 2

low	high	mid	
0	13	6	key < A[6]
0	5	2	key < A[2]
0	1	0	

Terminating condition, since  $A[mid] == 2$ , return 1(successful).

For key = 103

low	high	mid	
0	13	6	key > A[6]
7	13	10	key > A[10]
11	13	12	key > A[12]
13	13	-	

Terminating condition  $high == low$ , since  $A[0] != 103$ , return 0(unsuccesful).

For key = 67

low	high	mid	
0	13	6	key > A[6]
7	13	10	key < A[10]
7	9	8	

Terminating condition, since  $A[mid] = 67$ , return 9(successful).

### Algorithm for Binary Search

```

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

### Efficiency:

From the above algorithm we can say that the running time of the algorithm is

$$T(n) = T(n/2) + O(1)$$

$$= O(\log n) \text{ (verify).}$$

In the best case output is obtained at one run i.e.  $O(1)$  time if the key is at middle. In the worst case the output is at the end of the array so running time is  $O(\log n)$  time. In the average case also running time is  $O(\log n)$ . For unsuccessful search best, worst and average time complexity is  $O(\log n)$ .

### **Program to search an element in an array using binary search.**

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10
void selection_sort(int [], int);           // Added to sort array
void Binary_search(int [],int,int,int);
int main()
{
    int arr[size],num,i,n,lower_bound,upper_bound;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    selection_sort(arr, n);           // Added to sort the array
    printf("\n The sorted array is: \n");
    for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
    printf("\n\n Enter the number that has to be searched: ");
    scanf("%d", &num);
    lower_bound = 0;
    upper_bound = n-1;
    Binary_search(arr,lower_bound,upper_bound,num);
    getch();
    return 0;
}
void Binary_search(int arr[],int lower_bound,int upper_bound,int val)
{
    int beg,end,mid,pos;
    beg = lower_bound;
    end = upper_bound;
    pos=-1;
    while(beg<=end)
    {
        mid = (beg + end)/2;
        if (arr[mid] == val)
        {
            pos = mid;
            printf("\n %d is present in the array at index position %d", val,pos);
            break;
        }
        else if (arr[mid]>val)
            end = mid-1;
        else
            beg = mid+1;
    }
    if (pos==-1)
        printf("\n %d does not exist in the array", val);
}
```



```

void selection_sort(int arr[],int n)
{
    int i,j,pos,temp;
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(arr[pos]>arr[j])
                pos=j;
        }
        temp = arr[i];
        arr[i] = arr[pos];
        arr[pos] = temp;
    }
}

```

```

Enter the number of elements in the array: 5
Enter the elements:
Enter 1 element:10
Enter 2 element:20
Enter 3 element:5
Enter 4 element:30
Enter 5 element:15

The sorted array is:
5      10     15     20     30

Enter the number that has to be searched: 15
15 is present in the array at index position 2

```

### 8.3 Hashing

- Hashing is a technique where we can compute the location of the desired record in order to retrieve it in a single access (or comparison). Let there is a table of  $n$  employee records and each employee record is defined by a unique employee code, which is a key to the record and employee name. If the key (or employee code) is used as the array index, then the record can be accessed by the key directly. If  $L$  is the memory location where each record is related with the key. If we can locate the memory address of a record from the key then the desired record can be retrieved in a single access. A function that transforms an element into an array index is called hash function. If  $H$  is the hash function and key is the elements, then  $H(\text{key})$  is called as hash of key and is the index at which the element should be placed.
- It is an efficient searching technique in which key is placed in direct accessible address for rapid search.
- Hashing provides the direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. This technique uses a hashing function say  $H$  which maps the key with the corresponding key address or location.
- **Collision:** suppose that two elements  $k_1$  and  $k_2$  are such that  $H(k_1) = H(k_2)$ . Then when element  $k_1$  is entered into the table, it is inserted at position  $H(k_1)$ . But when  $k_2$  is hashed, an attempt may be made to insert the element into the same position where the element  $k_1$  is stored. Clearly two elements cannot occupy the same position. Such a situation is known as a hash collision or simply collision.
- **Hash Function:** The basic idea of hash function is the transformation of the key into the corresponding location in the hash table. A Hash function  $H$  can be defined as a function that takes key as input and transforms it into a hash table index.

Some of the hash functions are as follows

Division method, mid square method and folding method.

#### Division Method:

Choose a number  $m$ , which is larger than the number of keys  $k$ . i.e.,  $m$  is greater than the total number of records in the TABLE. The hash function  $H$  is defined by  $H(k) = k \pmod{m}$  Where  $H(k)$  is the hash address (or index of the array) and here  $k \pmod{m}$  means the remainder when  $k$  is divided by  $m$ .

Let a company has 90 employees and 00, 01, 02, ..... 99 be the two digit 100 memory address (or index or hash address) to store the records. We have employee code as the key. Choose  $m$  in such a way that it is greater than 90. Suppose  $m = 93$ . Then for the following employee code (or key  $k$ ):

$$H(k) = H(2103) = 2103 \pmod{93} = 57$$

$$H(k) = H(6147) = 6147 \pmod{93} = 9$$

$$H(k) = H(3750) = 3750 \pmod{93} = 30$$

Then a typical employee hash table will look like as below table.



Hash Address	Employee Code (keys)	Employee Name and other Details
0		
1		
..		
..		
..		
9	6147	Anish
..		
..		
30	3750	Saju
..		
..		
57	2103	Rarish
..		
..		
99		

Hash Table

So if you enter the employee code to the hash function, we can directly retrieve  $TABLE[H(k)]$  details directly. Note that if the memory address begins with 01-m instead of 00-m, then we have to choose the hash function  $H(k) = k \pmod{m} + 1$ .

#### ✚ 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.

K	4147	3750	2103
$K^2$	17197609	14062500	4422609
$H(k)$	97	62	22

~~17197609~~  
~~14062500~~  
~~4422609~~

Hash Address	Employee Code (keys)	Employee Name and other Details
0		
1		
..		
..		
..		
22	2103	Girl
..		
..		
62	3750	Sunil
..		
..		
..		
97	4147	Renjith
..		
99		

Hash Table with Mid Square Division

### **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  $k_1, k_2, \dots, k_n$ , 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  $k_1 + k_2 + \dots + k_n$ . The hash value is produced by ignoring the last carry, if any.

Here we are dealing with a hash table with index form 00 to 99, i.e., two-digit hash table. So we divide the  $K$  numbers of two digits.

K	2103	7148	12345
$k_1 k_2 k_3$	21, 03	71, 46	12, 34, 5
$H(k)$ $= k_1 + k_2 + k_3$	$H(2103)$ $= 21+03 = 24$	$H(7148)$ $= 71+46 = 19$	$H(12345)$ $= 12+34+5 = 51$

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

#### 8.3.1 Hash Collision and Handling of Hash Collision

It is possible that two non-identical keys  $K_1, K_2$  are hashed into the same hash address. This situation is called Hash Collision.

Let us consider a hash table having 10 locations as in table.

Location	Keys	Records
0	210	
1	111	
2		
3	883	
4	344	
5		
6		
7		
8	488	
9		

**Division Method is used to hash the key.**

$H(K) = K \text{ (mod) } m$ ; Here  $m$  is chosen as 10. The Hash function produces any integer between 0 and 9 inclusions, depending on the value of the key. If we want to insert a new record with key 500 then

$$H(500) = 500 \text{ (mod) } 10 = 0.$$

The location 0 in the table is already filled (i.e., not empty). Thus collision occurred.

Collisions are almost impossible to avoid but it can be minimized considerably by introducing any one of the following techniques:

- Open addressing
- Chaining
- Bucket addressing



### ✚ Open Addressing (Linear Probing)

In open addressing method, when a key is colliding with another key, the collision is resolved by finding a nearest empty space by probing the cells. Suppose a record R with key K has a hash address  $H(k) = h$ . then we will linearly search  $h + i$  (where  $i = 0, 1, 2, \dots, m$ ) locations for free space (i.e.,  $h, h + 1, h + 2, h + 3, \dots$  hash address).

To understand the concept, let us consider a hash collision which is in the hash table Table 1. If we try to insert a new record with a key 500 then  $H(500) = 500 \pmod{10} = 0$ .

The array index 0 is already occupied by H(210). With open addressing we resolve the hash collision by inserting the record in the next available free or empty location in the table. Here the key 111 also occupies next location, i.e., array hash index 1. Next available free location in the table is array index 2 and we place the record in this free location.

The position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found. This type of probing is called Linear Probing.

The main disadvantage of Linear Probing is that substantial amount of time will take to find the free cell by sequential or linear searching the table.

### Quadratic Probing

Suppose a record with R with key k has the hash address  $H(K) = h$ . Then instead of searching the location with address  $h, h + 1, h + 2, \dots, h + i, \dots$ , we search for free hash address  $h, h + 1, h + 4, h + 9, h + 16, \dots, h + i^2, \dots$ .

### 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:

$$h(k, i) = [h_1(k) + i h_2(k)] \pmod{m}$$

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

### ✚ 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.

### ✚ Chaining

In chaining technique the entries in the hash table are dynamically allocated and entered into a linked list associated with each hash key. The hash table can be represented using linked list as in following figure.

Location	Keys	Records
0	210	30
1	111	12
2		
3	883	14
4	344	18
5		
6	546	32
7		
8	488	31
9		



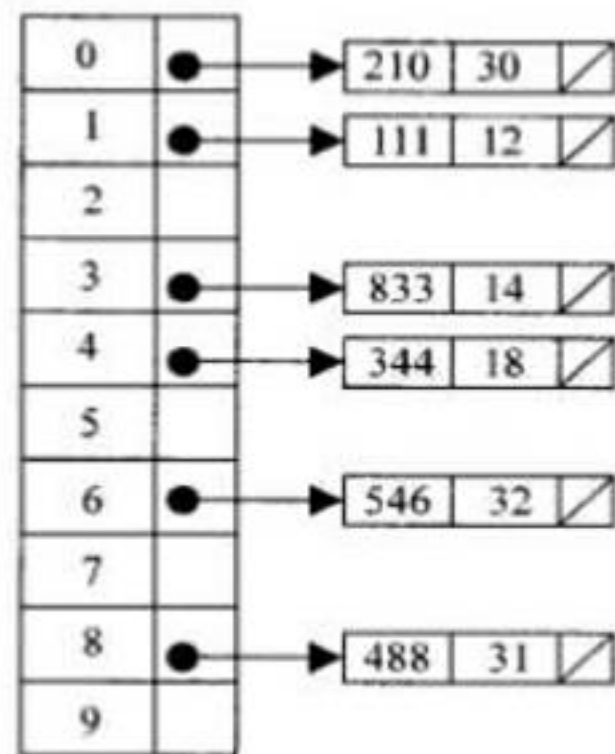


Figure: Chaining

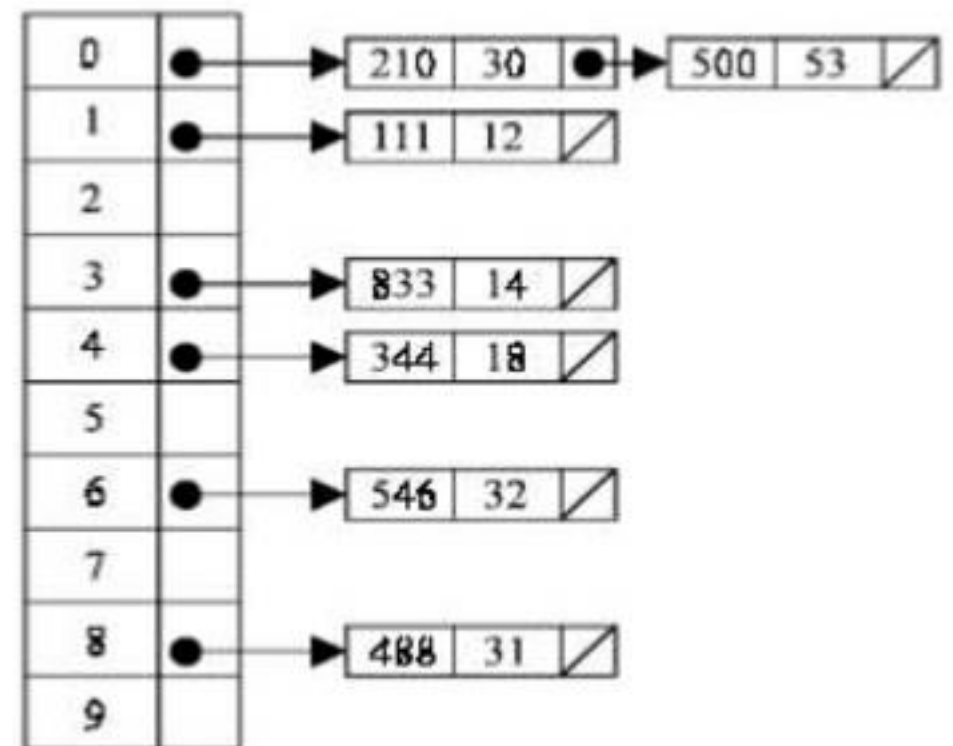


Figure: Linked List at 0 index

If we try to insert a new record with a key 500 then  $H(500) = 500 \pmod{10} = 0$ . Then the collision occurs in normal way because there exists a record in the 0th position. But in chaining corresponding linked list can be extended to accommodate the new record with the key.

### Bucket Addressing

Another solution to the hash collision problem is to store colliding elements in the same position in table by introducing a bucket with each hash address. A bucket is a block of memory space, which is large enough to store multiple items.

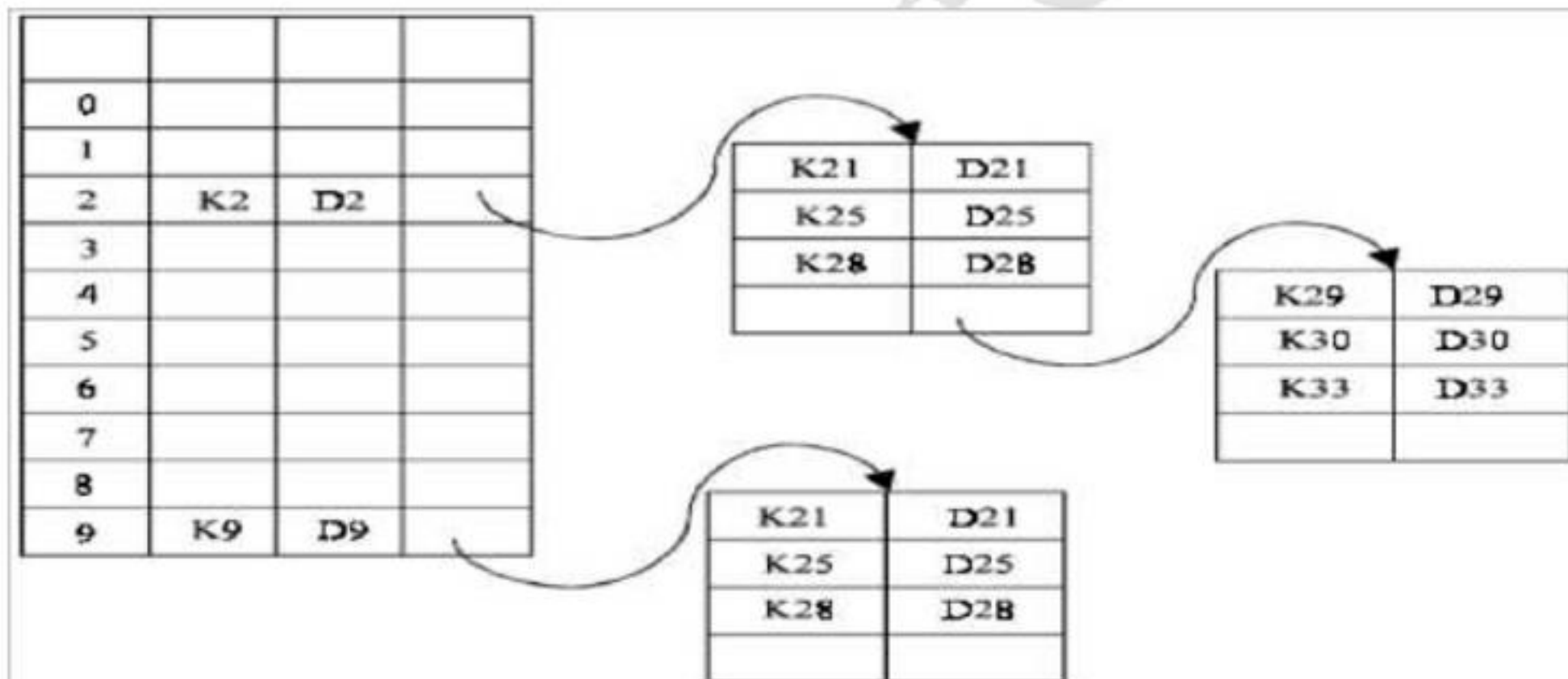


Figure: Avoid Collision Using bucket

Figure above shows how hash collision can be avoided using buckets. If a bucket is full, then the colliding item can be stored in the new bucket by incorporating its link to previous bucket.

### Hash Deletion

A data can be deleted from a hash table. In chaining method, deleting an element leads to the deletion of a node from a linked list. But in linear probing, when a data is deleted with its key the position of the array index is made free. The situation is same for other open addressing methods.