7 Searching and Hashing

Searching is a process of checking and finding an element from a list of elements. Let A be a collection of data elements, *i.e.*, A is a linear array of say *n* elements. If we want to find the presence of an element "*data*" in A, then we have to search for it. The search is successful if *data* does appear in A and unsuccessful if otherwise. 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. Fibanocci Search

7.1. LINEAR OR SEQUENTIAL SEARCHING

In linear search, each element of an array is read one by one sequentially and it is compared with the desired element. A search will be unsuccessful if all the elements are read and the desired element is not found.

7.1.1. ALGORITHM FOR LINEAR SEARCH

Let A be an array of *n* elements, A[1],A[2],A[3], ... A[*n*]. "*data*" is the element to be searched. Then this algorithm will find the location "*loc*" of data in A. Set *loc* = – 1,if the search is unsuccessful.

1. Input an array A of n elements and "data" to be searched and initialise *loc* = – 1.
2. Initialise $i = 0$; and repeat through step 3 if ($i < n$) by incrementing *i* by one .
3. If (data = A[*i*])
   (*a*) loc = *i*
   (*b*) GOTO step 4
4. If (loc > 0)
   (*a*) Display "data is found and searching is successful"
5. Else
   (*a*) Display "data is not found and searching is unsuccessful"
6. Exit

PROGRAM 7.1

```
//PROGRAM TO IMPLEMENT SEQUENTIAL SEARCHING
//CODED AND COMPILED USING TURBO C

#include<conio.h>
#include<stdio.h>

void main()
{
	char opt;
	int arr[20],n,i,item;
	clrscr();
	printf ("\nHow many elements you want to enter in the array : ");
	scanf ("%d",&n);

	for(i=0; i < n;i++)
	{
		printf ("\nEnter element %d : ",i+1);
		scanf ("%d", &arr[i]);
	}
	printf ("\n\nPress any key to continue....");
	getch();
	do
	{
		clrscr();
		printf ("\nEnter the element to be searched : ");
		scanf ("%d",&item); //Input the item to be searched
		for(i=0;i < n;i++)
		{
			if item == arr[i])
			{
				printf ("\n%d found at position %d\n",item,i+1);
				break;
			}
		}/*End of for*/
		if (i == n)
			printf ("\nItem %d not found in array\n",item);
		printf ("\n\nPress (Y/y) to continue : ");
		fflush(stdin);
```

```
            scanf ("%c",&opt);
        }while(opt == 'Y' || opt == 'y');
    }
```

7.1.2. TIME COMPLEXITY

Time Complexity of the linear search is found by number of comparisons made in searching a record.

In the best case, the desired element is present in the first position of the array, *i.e.*, only one comparison is made. So $f(n) = O(1)$.

In the Average case, the desired element is found in the half position of the array, then $f(n) = O[(n + 1)/2]$.

But in the worst case the desired element is present in the nth (or last) position of the array, so n comparisons are made. So $f(n) = O(n + 1)$.

7.2. BINARY SEARCH

Binary search is an extremely efficient algorithm when it is compared to linear search. Binary search technique searches "data" in minimum possible comparisons. Suppose the given array is a sorted one, otherwise first we have to sort the array elements. Then apply the following conditions to search a "data".

1. Find the middle element of the array (*i.e.*, $n/2$ is the middle element if the array or the sub-array contains n elements).
2. Compare the middle element with the data to be searched, then there are following three cases.
   (*a*) If it is a desired element, then search is successful.
   (*b*) If it is less than desired data, then search only the first half of the array, *i.e.*, the elements which come to the left side of the middle element.
   (*c*) If it is greater than the desired data, then search only the second half of the array, *i.e.*, the elements which come to the right side of the middle element.

Repeat the same steps until an element is found or exhaust the search area.

7.2.1. ALGORITHM FOR BINARY SEARCH

Let A be an array of n elements A[1],A[2],A[3],..... A[n]. "Data" is an element to be searched. "mid" denotes the middle location of a segment (or array or sub-array) of the element of A. LB and UB is the lower and upper bound of the array which is under consideration.

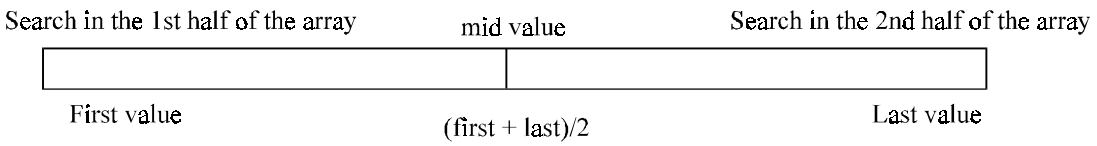
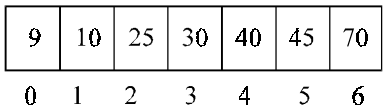


Fig. 7.1

1. Input an array A of n elements and "data" to be sorted

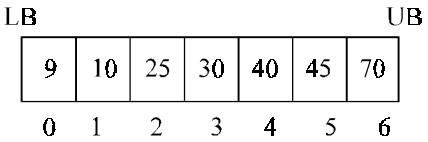
2. LB = 0, UB = *n*; mid = int ((LB+UB)/2)
3. Repeat step 4 and 5 while (LB <= UB) and (A[mid] ! = data)
4. If (data < A[mid])
   (*a*) UB = mid–1
5. Else
   (*a*) LB = mid + 1
6. Mid = int ((LB + UB)/2)
7. If (A[mid]== data)
   (*a*) Display “the data found”
8. Else
   (*a*) Display “the data is not found”
9. Exit

Suppose we have an array of 7 elements



Following steps are generated if we binary search a data = 45 from the above array.

Step 1:



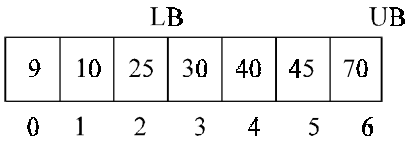
LB = 0; UB = 6

mid = (0 + 6)/2 = 3

A[mid] = A[3] = 30

Step 2:

Since (A[3] < data) - *i.e.,* 30 < 45 - reinitialise the variable LB, UB and mid



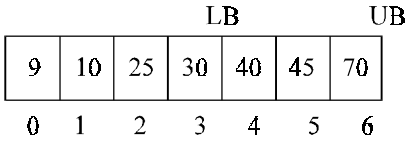
LB = 3 UB = 6

mid = (3 + 6)/2 = 4

A[mid] = A[4] = 40

Step 3:

Since (A[4] < data) - *i.e.,* 40 < 45 - reinitialise the variable LB, UB and mid



LB = 4 UB = 6

mid = (4 + 6)/2 = 5

A[mid] = A[5] = 45

Step 4:

Since (A[5] == data) - *i.e.*, 45 == 45 - searching is successful.

PROGRAM 7.2

```
//PROGRAM TO IMPLEMENT THE BINARY SEARCH
//CODED AND COMPILED USING TURBO C

#include<conio.h>
#include<stdio.h>

void main()
{
	char opt;
	int arr[20],start,end,middle,n,i,item;
	clrscr();
	printf ("\nHow many elements you want to enter in the array : ");
	scanf ("%d",&n);
	for(i=0; i < n; i++)
	{
		printf ("\nEnter element %d : ",i+1);
		scanf ("%d",&arr[i]);
	}
	printf ("\n\nPress any key to continue...");
	getch();
	do
	{
		clrscr();
		printf ("\nEnter the element to be searched : ");
		scanf ("%d",&item);
		start=0;
		end=n – 1;
		middle=(start + end)/2;
```

```
        while(item != arr[middle] && start <= end)
        {
                if (item > arr[middle])
                        start=middle+1;
                else
                        end=middle-1;
                middle=(start+end)/2;
        }
        if (item==arr[middle])
                printf("\n%d found at position %d\n",item,middle + 1);
        if (start>end)
                printf ("\n%d not found in array\n",item);
        printf ("\n\nPree (Y/y) to continue : ");
        fflush(stdin);
        scanf ("%c",&opt);
    }while(opt == 'Y' || opt == 'y');
}/*End of main()*/
```

7.2.2. TIME COMPLEXITY

Time Complexity is measured by the number $f(n)$ of comparisons to locate "data" in A, which contain n elements. Observe that in each comparison the size of the search area is reduced by half. Hence in the worst case, at most $\log_2 n$ comparisons required. So $f(n) = O([\log_2 n]+1)$.

Time Complexity in the average case is almost approximately equal to the running time of the worst case.

7.3. INTERPOLATION SEARCH

Another technique for searching an ordered array is called interpolation search. This method is even more efficient than binary search, if the elements are uniformly distributed (or sorted) in an array A.

Consider an array A of n elements and the elements are uniformly distributed (or the elements are arranged in a sorted array). Initially, as in binary search, low is set to 0 and high is set to n – 1.

Now we are searching an element key in an array between A[low] and A[high]. The key would be expected to be at mid, which is an approximately position.

$$mid = low + (high - low) \times ((key - A[low])/(A[high] - A[low]))$$

If key is lower than A[mid], reset high to mid–1; else reset low to mid+1. Repeat the process until the key has found or low > high.

Interpolation search can be explained with an example below. Consider 7 numbers :

2, 25, 35, 39, 40, 47, 50

CASE 1: Say we are searching 50 from the array

Here $n = 7$ Key = 50

low = 0

high = n – 1 = 6
mid = 0+(6–0) × ((50–2)/(50–2))
= 6 × (48/48)
= 6
if (key == A[mid])
⇒ key == A[6]
⇒ 50 == 50
⇒ key is found.
CASE 2: Say we are searching 25 from the array
Here n = 7 Key = 25
low = 0
high = n – 1 = 6
mid = 0 + (6–0) × ((25–2)/(50–2))
= 6 × (23/48)
= 2.875 Here we consider only the integer part of the mid
i.e., mid = 2
if (key == A[mid])
⇒ key == A[2]
⇒ 25 == 25
⇒ key is found.
CASE 3: Say we are searching 34 from the array
Here n = 7 Key = 34
low = 0
high = n – 1 = 6
mid = 0 + (6 – 0) × ((34 – 2)/(34 – 2))
= 6 × (32/48)
= 4
if(key < A[mid])
⇒ key < A[4]
⇒ 34 < 40
so reset high = mid–1
⇒ 3
low = 0
high = 3
Since(low < high)
mid = 0+(3–0) × ((34–2)/(39–2))
= 3 × (32/37)
= 2.59 Here we consider only the integer part of the mid
i.e., mid = 2

if (key < A[mid])
⇒ key < A[2]
⇒ 34 < 35
so reset high = mid–1
⇒ 1
low = 0
high = 1
Since (low < high)
mid = 0+(1–0) × ((34–2)/(25–2))
= 3 × (32/23)
= 1
here (key > A[mid])
⇒ key > A[1]
⇒ 34 > 25
so reset low = mid+1
⇒ 2
low = 2
high = 1
Since (low > high)
DISPLAY " The key is not in the array"
STOP

ALGORITHM

Suppose A be array of sorted elements and key is the elements to be searched and low represents the lower bound of the array and high represents higher bound of the array.

1. Input a sorted array of *n* elements and the key to be searched
2. Initialise low = 0 and high = n – 1
3. Repeat the steps 4 through 7 until if(low < high)
4. Mid = low + (high – low) × ((key – A[low]) / (A[high] – A[low]))
5. If(key < A[mid])
   (*a*) high = mid–1
6. Elseif (key > A[mid])
   (*a*) low = mid + 1
7. Else
   (*a*) DISPLAY " The key is not in the array"
   (*b*) STOP
8. STOP

PROGRAM 7.3

```
//PROGRAM TO IMPLEMENT INTERPOLATION SEARCH
//CODED AND COMPILED USING TURBO C++

#include<conio.h>
#include<iostream.h>

class interpolation
{
      int Key;
      int Low,High,Mid;
      public:
            void InterSearch(int*,int);
};

//This function will search the element using interpolation search
void interpolation::InterSearch(int *Arr,int No)
{
      int Key;
      //Assigning the pointer low and high
      Low=0;High=No–1;
      //Inputting the element to be searched
      cout<<"\n\nEnter the Number to be searched = ";
      cin>>Key;

      while(Low < High)
      {
            //Finding the Mid position of the array to be searched
            Mid=Low+(High–Low)*((Key–Arr[Low])/(Arr[High]–Arr[Low]));
            if (Key < Arr[Mid])
                  //Re-initializing the high pointer if the
                  //key is greater than the mid value
                  High=Mid–1;
            else if (Key > Arr[Mid])
                  //Re initializing the low pointer if the
                  //key is less than the mid value
                  Low=Mid+1;
            else
            {
                  //if the key value is equal to the mid value
```

```
                    //of the array, the key is found
                    cout<<"\nThe key "<<Key<<" is found at the location "<<Mid;
                    return;
                }
        };
        cout<<"\n\nThe Key "<<Key<<" is NOT found";
}

void main()
{
        int *a,n,*b;
        interpolation Ob;
        clrscr();
        cout<<"\n\nEnter the number of elements : ";
        cin>>n;

        a=new int[n];
        b=a;
        //Input the elements in the array
        for (int i=0;i<n;i++)
        {
                cout<<"\nEnter the "<<i<<" element : ";
                cin>>*a;
                a++;
        }
        //calling the InterSearch function using objects
        Ob.InterSearch(b,n);
        cout<<"\n\nPress any key to continue...";
        getch();
}
```

7.4. FIBONACCI SEARCH

A possible improvement in binary search is not to use the middle element at each step, but to guess more precisely where the key being sought falls within the current interval of interest. This improved version is called fibonacci search. Instead of splitting the array in the middle, this implementation splits the array corresponding to the fibonacci numbers, which are defined in the following manner:

$F_0 = 0, F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ for $n >= 2$.

Let's assume that our array has the F_{n-1} ($n = F_{n-1}$) elements. Now we divide the array into two subsets according to the both preceding fibonacci numbers and compare 'item' to the element at the position F_{n-2}. If 'item' is greater than the element, we continue

in the right subset, otherwise in the left subset. In this algorithm we don't have to make a division to calculate the middle element, but we get along only with additions and subtractions. In our implementation we assumed that the fibonacci numbers are given explicitly (*e.g.*, as constants in the frame program).

PROGRAM 7.4

```
//PROGRAM TO IMPLEMENT THE FIBONACCI SEARCH
//CODED AND COMPILED IN TURBO C

#include<iostream.h>
#include<conio.h>

//This function will find the fibonacci number
int fib(int n)
{
        int f1,f2,temp;
        f1=0;f2=1;
        for (int i=0;i<n;i++)
        {
                temp=f2;
                f2=f1+f2;
                f1=temp;
        }
        return(f2);
}

//Function to search an item using fibonacci numbers
int fibonacci_search(int list[],int n,int item)
{
        int f1,f2,t,mid;

        for (int j=1;fib(j)<n;j++);
        f1=fib(j–2);            //find lower fibonacci numbers
        f2=fib(j–3);            //f1=fib(j–2), f2=fib(j–3)
        mid=n–f1+1;
        while (item != list[mid])    //if not found
                if (mid<0 || item > list[mid])
                {               //look in lower half
                        if (f1==1)
                                return(–1);
```

```
                mid=mid+f2;//decrease fibonacci numbers
                f1 = f1–f2;
                f2 = f2–f1;
        }
        else
        {       //look in upper half
                if (f2==0)      //if not found return –1
                        return(–1);
                mid=mid–f2;//decrease fibonacci numbers
                t=f1–f2;                //this time, decrease more
                f1=f2;          //for smaller list
                f2=t;
        }
        return(mid);
}


void main()
{
        int loc,n,item,list[50];
        cout<<"\n\nEnter the total number of list : ";
        cin>>n;

        cout<<"\nEnter the elements in the list:";
        for (int i=0;i<n;i++)
        {
                cout<<"\nInput "<<i<<" th number : ";
                cin>>list[i];
        }

        cout<<"\nEnter the number to be searched :";
        cin>>item;
        loc=fibonacci_search(list,n,item);
        if (loc != –1)
                cout<<"\nThe number is in the list";
        else
                cout<<"\nThe number is not in the list";
        getch();
}
```

WORST CASE PERFORMANCE

Beginning with an array containing F_{j-1} elements, the length of the subset is bounded to F_{j-1}-1 elements. To search through a range with a length of F_{n-1} at the beginning we have to make *n* comparisons in the worst case. F_n = (1/sqrt(5))*((1+sqrt(5))/2)*n*, that's approximately c*1,618*n* (with a constant c). For N+1 = c*1,618*n* elements we need *n* comparisons, *i.e.*, the maximum number of comparisons is O (ld N).

7.5. HASHING

The searching time of the each searching technique, that were discussed in the previous section, depends on the comparison. *i.e.*, *n* comparisons required for an array A with *n* elements. To increase the efficiency, *i.e.*, to reduce the searching time, we need to avoid unnecessary comparisons.

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. For notational and coding convenience, we assume that the keys in *k* and the address in L are (decimal) integers. So the location is selected by applying a function which is called *hash function* or *hashing function* from the key *k*. Unfortunately such a function H may not yield different values (or index or many address); it is possible that two different keys k_1 and k_2 will yield the same hash address. This situation is called *Hash Collision*, which is discussed in the next topic.

7.5.1. 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. Hash functions are of two types:

1. Distribution- Independent function
2. Distribution- Dependent function

We are dealing with Distribution - Independent function. Following are the most popular Distribution - Independent hash functions :

1. Division method
2. Mid Square method
3. Folding method.

7.5.1.1. Division Method

TABLE is an array of database file where the employee details are stored. 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 number *m* is usually chosen to be prime number to minimize the collision. 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 (mod m) means the remainder when k is divided by m.

For example:

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 grater than 90. Suppose m = 93. Then for the following employee code (or key k) :

H(k) = H(2103) = 2103 (mod93) = 57

H(k) = H(6147) = 6147 (mod 93) = 9

H(k) = H(3750) = 3750 (mod93) = 30

Then a typical employee hash table will look like as in Fig. 7.2.

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

Fig. 7.2. 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 (mod m)+1.

7.5.1.2. Mid Square Method

The key k is squared. Then the hash function H is defined by

$$H(k) = k^2 = l$$

Where l is obtained by digits from both the end of k^2 starting from left. Same number of digits must be used for all of the keys. For example consider following keys in the table and its hash index :

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

~~17~~1976~~09~~

~~140~~62~~500~~

~~44~~226~~09~~

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

Fig. 7.3. Hash table with mid square division

7.5.1.3. Folding Method

The key K, k_1, k_2,..... k_r is partitioned into number of parts. The parts have same number of digits as the required hash address, except possibly for the last part. Then the parts are added together, ignoring the last carry. That is

$$H(k) = k_1 + k_2 + ... + k_r$$

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

Fig. 7.4

Extra milling can also be applied to even numbered parts, k_2, k_4, ... are each reversed before the addition.

K	2103	7148	12345
k_1, k_2, k_3	21, 03	71, 46	12, 34, 5
Reversing k_2, k_4	21, 30	71, 64	12, 43, 5
$H(k) = k_1 + k_2 + k_3$	H(2103) = 21+30 = 51	H(7148) = 71+64 = 55	H(12345) = 12+43+5 = 60

Fig. 7.5

H(7148) = 71 + 64 = 155, here we will eliminate the leading carry (*i.e.*, 1). So H(7148) = 71 + 64 = 55.

7.5.2. 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.

Location	*(Keys)*	*Records*
0	210	
1	111	
2		
3	883	
4	344	
5		
6		

7		
8	488	
9		

Fig. 7.6

Let us consider a hash table having 10 locations as shown in Fig. 7.6. 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 three techniques:

1. Open addressing
2. Chaining
3. Bucket addressing

7.5.2.1. Open Addressing

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 $i = 0, 1, 2, \ldots\ldots m$) locations for free space (*i.e.*, h, $h + 1$, $h + 2$, $h + 3$ ..... hash address).

To understand the concept, let us consider a hash collision which is in the hash table shown in Fig. 7.6.

If we try to insert a new record with a key 500 then

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

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 next location, *i.e.*, array hash index 1, is also occupied by the key 111. Next available free location in the table is array index 2 and we place the record in this free location.

Location	*(Keys)*	*Records*
0	210	
1	111	
2	500	
3	883	
4	344	
5		
6		
7		
8	488	
9		

Fig. 7.7

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. Other two techniques, which are discussed in the following sections, will minimize this searching time considerably.

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, $h + 1$, $h + 2$,..... $h + i$ ....., we search for free hash address h, $h + 1$, $h + 4$, $h + 9$, $h + 16$, ... $h + i^2$,..... .

DOUBLE HASHING

Second hash function H_1 is used to resolve the collision. Suppose a record R with key k has the hash address $H(k) = h$ and $H1(k) = h^1$, which is not equal to m. Then we linearly search for the location with addresses

h, $h + h^1$, $h + 2h^1$, $h + 3h^1$, ... $h + i\,(h^1)^2$ (where $i = 0, 1, 2,$ .....).

Note : The main drawback of implementing any open addressing procedure is the implementation of deletion.

7.5.2.2. 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 in Fig. 7.8 can represented using linked list as in Fig. 7.9.

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		

Fig. 7.8

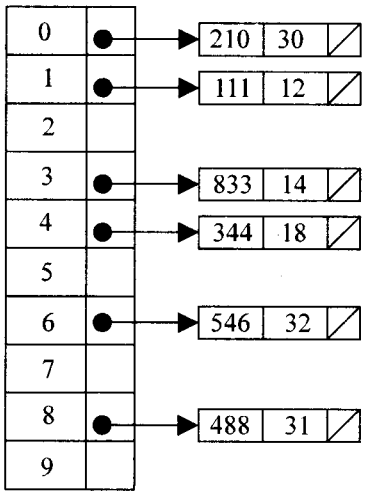


Fig. 7.9

If we try to insert a new record with a key 500 then H(500) = 500(mod 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 as shown in Fig. 7.10.

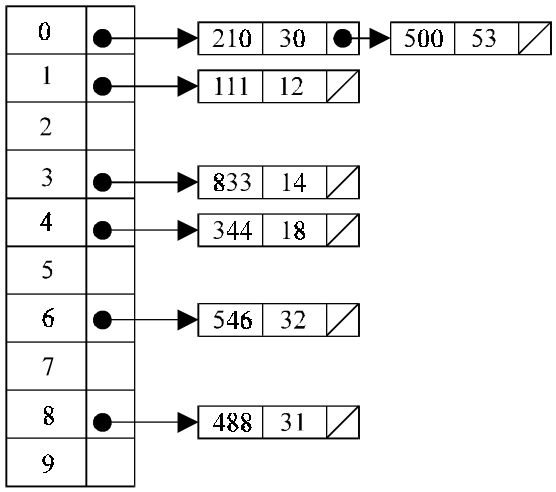


Fig. 7.10

7.5.2.3.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.

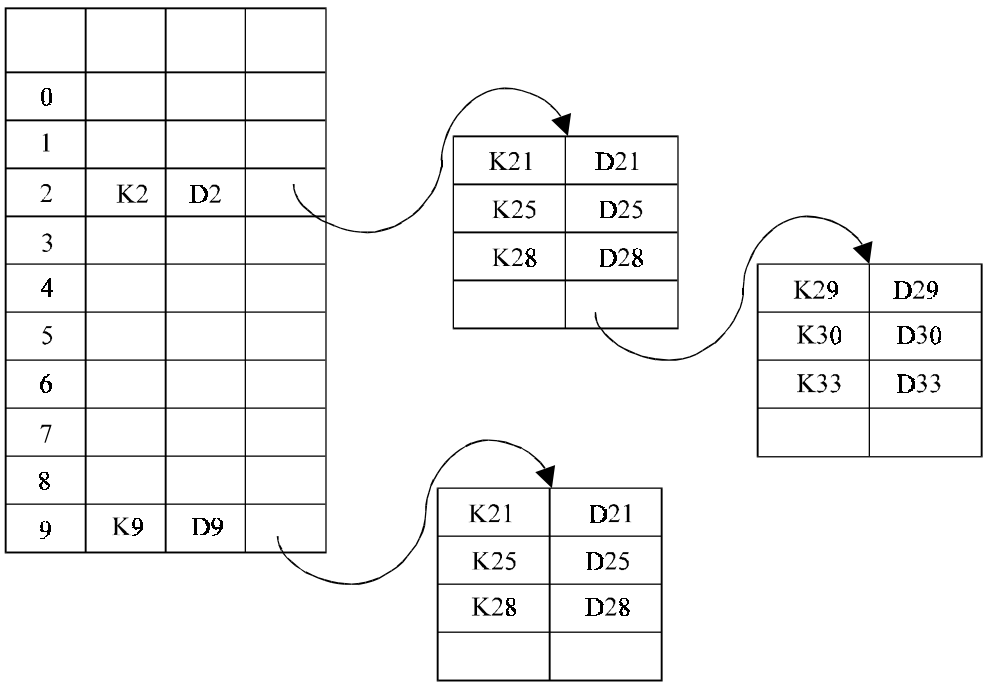


Fig. 7.11. Avoiding collision using buckets

Fig. 7.11 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.

7.5.3. 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.

SELF REVIEW QUESTIONS

1. Write an algorithm for binary search and discuss its speed compared with linear search. [*MG - MAY 2003* (*BTech*)]
2. Write a general algorithm for inserting a name into the structure using hashing techniques. [*MG - MAY 2003* (*BTech*)]
3. Compare and contrast between sequential search and binary search.
   [*ANNA - DEC 2003* (*BE*), *MG - MAY 2002* (*BTech*)
   KERALA - DEC 2002 (*BTech*), *ANNA - DEC 2004* (*BE*)]
4. Write about the complexities of linear search, binary search and fibonaci search. [*MG - MAY 2000* (*BTech*)]
5. What is hashing ? Explain with illustrative examples. Discuss any two hashing techniques you are familiar with. [*ANNA - DEC 2003* (*BE*), *Calicut - APR 1995* (*BTech*)]
6. Write a note on Hashing functions.
   [*CUSAT - MAY 2000* (*BTech*), *Calicut - APR 1997* (*BTech*)
   ANNA - MAY 2003 (*BE*), *ANNA - MAY 2004* (*MCA*)
   KERALA - MAY 2002 (*BTech*), *KERALA - DEC 2002* (*BTech*)]
7. Explain the organization of hash table. How are collisions handled? [*CUSAT - NOV 2002* (*BTech*)]
8. What is clustering in a Hash table? Describe two methods for collision resolution. [*ANNA - DEC 2003* (*BE*)]
9. Write down the binary search algorithm and obtain the complexities of both worst and average cases. [*ANNA - DEC 2004* (*BE*)]
10. Describe various hashing techniques.
    [*KERALA - DEC 2003* (*BTech*), *ANNA - MAY 2003* (*BE*)]
11. Explain (i) hash collision (ii) Linear Searching. [*KERALA - JUN 2004* (*BTech*)]
12. Write a recursive procedure for binary search method. [*KERALA - JUN 2004* (*BTech*)]
13. Write an algorithm for binary search method by iteration method. [*KERALA - MAY 2003* (*BTech*)]
14. What are the hash collision resolution techniques? [*KERALA - NOV 2001* (*BTech*), *KERALA - DEC 2002* (*BTech*)]

15. What is searching ? Compare various search methods. [*KERALA - MAY 2001* (*BTech*)]
16. Write an algorithm for a hashing method. [*KERALA - MAY 2001* (*BTech*)]
17. Describe in detail one hash table method with a suitable method. Explain different probing technique. [*KERALA - MAY 2002* (*BTech*)]
18. What is meant by collision processing ? [*KERALA - NOV 2001* (*BTech*)]
19. Explain open addressing technique. [*KERALA - NOV 2001* (*BTech*)]