Chapter-4
Searching

Introduction

The process of finding the location of a specific data item or record with a given key value or finding the locations of all records which satisfy one or more conditions in a list is called Searching.

Types:

1. External Searching

   External searching means searching the records using the keys where there are many records, which resides in files stored on disk.

2. Internal Sorting

   Internal searching means searching the records using the keys where there are less number of records residing entirely within the computers main memory.

Need of Searching

Computer systems are often used to store large amount of data from which individual records must be retrieved according to some search condition or key. So one of the most important applications of computer system is information retrieval. To retrieve any kind of information, we need to search for it. While searching, if the item exist in the given list or file then the search is said to be successful otherwise unsuccessful. There are different types of search algorithms that can be used to search data. Some of them are

1. Linear Search
2. Binary Search
3. Hashing

Complexity of Searching

The complexity of searching algorithms is measured in terms of number of comparisons required to find an item from the list. The time required for search operation depends on the complexity of searching algorithms. While searching a particular item in the list, we have to consider the following three cases

Best Case: The best case is that in which the element is found during the first comparisons.

Worst Case: The worst case is that in which the element is found at the end of the list or item is not in the list.

Average Case: The average case is that in which the element is found in the comparisons more than best case and less than worst case.

Searching Algorithms

1. Linear Searching / Sequential Searching

Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

A search is said to be unsuccessful if all the elements are read and the desired element is not found. A linear search algorithm can be applied to both sorted and unsorted list. In case of sorted list, search starts from 0^{th} element and continues until the element is found or an element whose value is greater that the value being searched. For unsorted list, search starts from 0^{th} element and continues until the element is found or up to the last element in the list.

Advantage

1. It is simple way for finding an element in a list.
2. It can be applied to both sorted and unsorted list.

Disadvantage

1. It is very slow process so can be applied to small size list only

Algorithm

Let A[N] be an array of size N.

1. Read the elements of array A
2. Read the Data to be searched.
3. Set loc=0
4. Repeat step 5 until loc<N or A[loc] = Data
5. Increment loc by 1.
6. If(loc<N)
   Display “Search is successful”
   else
   Display “Search is unsuccessful”
7. Exit

C Implementation

```
#include<stdio.h>
int linerSearch(int arr[], int key);
int main()
{
	int arr1[]= {23,45,21,55,234,1,34,90};
	int searchKey = 34;
	printf("\nKey %d found at index %d",searchKey,linerSearch(arr1,searchKey));
	int arr2[]= {123,445,421,595,2134,41,304,190};
	searchKey = 421;
	printf("\nKey %d found at index %d",searchKey,linerSearch(arr2,searchKey));
}

int linerSearch(int arr[], int key)
{
	for(int i=0;i<8;i++)
	{
		if(arr[i] == key)
		{
			return i; // if search successful it will return its index
		}
	}
	return -1; // if search unsuccessful it will return -1.
}
```

Output:

```
Key 34 found at index 6
Key 421 found at index 2
```

Analysis of Linear Search

1. Best Case: If the desired item is found at the first position in the list then we will have to make only one comparison. Therefore the best case efficiency of linear search is O(1).
2. Worst Case: If the desired item is stored at the last position in the list or if it doesn’t exist in the list, we will have to make N comparisons, where N is the number of items in the list. Therefore the worst case efficiency of linear search is O(N).

3. Average Case: The average case efficiency of linear search can be determined by finding the average number of comparisons in best case and worst case i.e. O((1+N)/2) which is similar to O(N).

Binary searching

Linear searching is inefficient searching solution for large list. An alternate solution that offers better efficiency for large list is binary search algorithm. This algorithm helps us to search data in few comparisons. To apply binary search algorithm, the list to be searched must be sorted.

In binary search, we first compare the key with the item in the middle position of the array. If there is a match, the search is successful. Otherwise, if the key is lesser than the middle key, the item to search should lie in the lower half of array but if the key is greater than the middle key, the item to be searched should lie in the upper half of array. So we repeat this procedure on the lower and upper half of the array until the element is found or the search area is exhausted.

Advantage

1. It provides better efficiency for searching in large list.

Disadvantage

1. It requires items in the array to be sorted.
2. It is not suitable where there are many insertion and deletions.

Algorithm

Let A[N] be an array of sorted elements and key is the item to be searched.

1. Input the sorted array of N elements A[N] and key to be searched.
2. Set lowerbound = 0
3. Set upperbound = N-1
4. Set Mid = (lowerbound + upperbound) / 2
5. If A[Mid] == key
    a. Display “The item is found”
    b. Exit
6. If key < A[Mid]
    a. Set upperbound =Mid -1
7. If key > A[Mid]
    a. Set lowerbound =Mid +1
8. If lowerbound<=upperbound
    a. Goto step 4
9. Display “The item is not found”
10. Exit

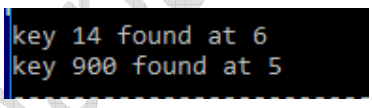
C-Implementation

```
#include<stdio.h>
int binarySearch(int [],int,int);
int main()
{
        int arr[] = {2, 4, 6, 8, 10, 12, 14, 16};
        int searchkey=14;
        int length=8;
        printf("\nkey %d found at %d",searchkey,binarySearch(arr,searchkey,length));
        int arr1[] = {6,34,78,123,432,900};
        searchkey=900;
        length=6;
        printf("\nkey %d found at %d",searchkey,binarySearch(arr1,searchkey,length));
}
```

```
int binarySearch(int inputArr[], int key,int length)
{
        int lowerbound = 0;
        int upperbound= length-1;
        while (lowerbound<= upperbound)
        {
                int mid = (lowerbound + upperbound) / 2;
                if (key == inputArr[mid])
                {
                        return mid; //if search successful return its index
                }

                if (key <inputArr[mid])
                {
                        upperbound = mid - 1;
                }
                else
                {
                        lowerbound = mid + 1;
                }
        }
        return -1; // if search unsuccessful return -1
}
```

Output:



Example:
Suppose an array A has the following elements 5, 8, 12, 25, 30, 40, 55. Trace the binary search if key =30

Solution:
Let Lb represent the lower bound and Ub represent the upper bound
Initially,

5	8	12	25	30	40	55
0	1	2	3	4	5	6

Lb Ub

i.e. Lb=0 and Ub=6

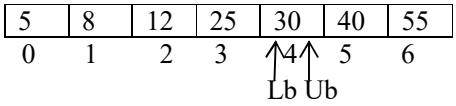
Step1:
mid =(Lb + Ub)/2 = (0 + 6)/2 = 3
A[mid] = A[3] = 25
Since key > A[mid] reinitializing Lb as Lb = mid +1 = 3+1 =4

5	8	12	25	30	40	55
0	1	2	3	4	5	6

Lb Ub

i.e. Lb= 4 and Ub =6

Step2:
mid =(Lb + Ub) /2 = (4 + 6) / 2 = 5
A[mid] = A[5] = 40
Since key < A[mid], reinitializing Ub as Ub = mid -1 = 5 – 1 =4



i.e. Lb= 4 and Ub =4

Step3:

mid = (Lb + Ub) / 2 = (4 + 4) /2 = 4
A[mid] = a[4] =30
Since key =A[mid], thus search is successful.

Analysis of Binary search

- The best case for this algorithm would be if the item to be searched is present at the middlemost position in the array. In this case, the desired item is found in just one comparison and hence the best case efficiency of binary search is O(1).
- The worst case efficiency would be if the desired record is not in the array or found in the last bisection. In this case, bisection continues until there is only one element left for comparisons.

  So in 1^{st} bisection, search space is reduced to N/2 elements.
  In 2^{nd} bisection, search space is reduced to N/4 elements.
  Similarly, in i^{th} bisection, search space is reduced to $N/2^i$ elements.
  Suppose after i^{th} bisection, the search space is reduced to 1 element
  i.e. $N/2^i = 1$
  $N = 2^i$
  $i = \log_2 N$ (Note: if $a = b^c$ then $c = \log_b a$)
  so the list can be bisected maximum $\log_2 N$ times and each bisection requires only one comparisons. Hence the worst case efficiency of binary search is O(logN)

Hashing

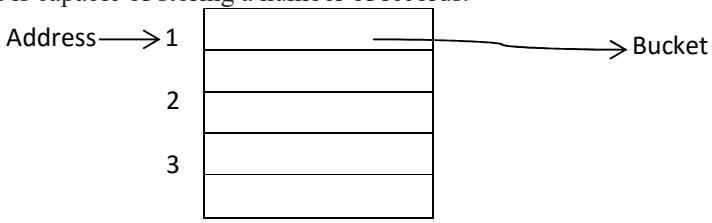
The main operation used by the sequential search and binary search is comparison of keys. A different approach to searching calculates the position of the key in the tablebased on the value of the key.When the key is known, the position in the table can be accessed directlyThis means that the search time is reduced from O(n), as in a sequentialsearch, or from O(log n), as in a binary search, to 1 or at least O(1) therefore regardless of thenumber of elements being searched, the run time is always the same.

"Hashing is one of the best methods of finding the information in which the offset address of desired record is calculated using hash function and read the record at the resultant offset address using hash table".

Terminologies

1. Hash Table

A hash table is simply an array that is addressed via a hash function. The basic idea behind a Hash table is to establish a mapping between the set of all possible keys and positions in the array using hash function. The hash table is divided into a number of buckets and each bucket is capable of storing a number of records.



So simply a hash table is a table where data to be searched is stored.

2. Hash Function

A hash function can be defined as a function that is used to compute the address of a record in hash table. A hash function takes search key as input and translate it into hash table index i.e. hash value. So a hashing function operates on a key to give

its hash value, which a number that represents the position (index) at which the record can be found in hash table.If a hash functiontransforms different keys into different hash value, it is called a perfect hash function.

3. Hash of Key (Hash Value)

Let H be a hash function and K is a key then the valueH(K) is called hash of key which is an index at which the record with a key value K must be kept or can be found in hash table.

Example:

Let us considerthat the hashing function is H(K) = K % 4, so by using this function on key 3, 5, 8 and 10, we get a hash table in which keys are mapped to their corresponding address.

H(3) = 3 % 4 =3
H(5) = 5 % 4 =1
H(8) = 8 % 4 =0
H(10) = 10 % 4 =2

Location	Keys
0	8
1	5
2	10
3	3

i.e.

8	5	10	3
0	1	2	3

Now to search any key, we follow the following steps

1. Given a key, the hash function convert it into a hash value i.e. array index/location
2. The record is then retrieved at the location generated.

Example: Let key is10; first we generate the array index by applying hash function to the search key as H(10) = 10 % 4 = 2, and directly access the item through array index 2.

Advantage of Hashing

It provides better efficiency than linear and binary search because it requires only onecomparison to access the desired record in ideal case.

Disadvantages of Hashing

1. Hash Collision/ Clash: It is a situation that occurs when two non-identical keys are hashed into the same location. But two different can't occupy the same location.

Example:

Let us consider that the hash function is H(K) = K % 4, where K is key. Consider the following set of keys

3, 4, 8 and 10

we get a hash table in which keys are mapped to their corresponding address as below

Location	Keys
0	4, 8
1	
2	10
3	3

So the keys 4 and 8 are hashed to same location 0, therefore results **collision.**

2. Hash table can't be accessed sequentially, so any post operation that requires sequential access to data items may not be very efficient.

Types of Hash Function

Before implementing hashing, we have to select a hash function which provides uniform distribution of keys in the address space. Some of the popular hash functions are

a. End Folding Method

- In this method, the key is divided into several parts. These parts are combined or folded together and are often transformed in a certain way**such as addition** to create the target address.
  Example: let us take some 7 digit keys :**4766934, 5656975, 4685637, 3547807, 7569664**
  Now, we divide the key into parts and add the parts together. Dividing the keys into parts of 2, 3 and 1 digit and adding them to combine as shown in the below table.

Keys	Dividing the Key and adding	Result	Address
4766934	47 + 6693 +4	6744	744
5656975	56 +5697 + 5	5758	758
4685637	46 + 8563 +6	8616	16
3547807	35 +4780+7	4822	822
7569664	75+6966+4	7045	45

The resulting numbers can be divided modulo Table Sizeor, if the size of the table is 1,000, the hash address can be from 0 to 999 so the first three digits can be used for the address by truncation the higher digit of result. We get the following hash values for the keys.

H(4766934) = 744
H(5656975) = 758
H(4685637) = 616
H(3547807) = 822
H(7569664) = 45

b. Mid SquareMethod

In this method, we first square the given key and take some digits from the middle of the number as an address for the corresponding key.

Example: Consider the following keys. 2636, 5657, 5869, 4756, 6079. Also let the table size is 100 so the hash address can be from 0 to 99. Now taking square of these keys and take 4^{th} and 5^{th} digits from the square of each key as their corresponding address.

Key	$(Key)^2$	Address
2636	6948496	84
5657	32001649	01
5869	34445161	45
4756	22619536	19
6079	36954241	54

We get the following hash values for the keys.

H(2636) = 84
H(5657) = 01
H(5869) = 45
H(4756) = 19
H(6079) = 54

c. Division or Modular Method

In this method, we apply modulus operation on each key. This involves dividing each key by the number equal to the size of the table. The remainder is considered as the address of the record corresponding to the key value.

i.e. H(K) = K Mod Tsize

The above hash function is best if Tsize is a prime number less than or equal to table size otherwise we can use the following hash function

H(K) = (K Mod P) Mod Tsize for some prime numbers greater than Tsize

Example: Consider the following keys. 36475611, 47566933, 75669353.Now assuming that the table size is 44. The nearest smallest prime number lesser or equal to table size is 43. So we using the hash function as H(K) = K % 43, we get the following has values for the keys.

H(36475611) =1
H(47566933) = 32

H(75669353) = 17

d. Extraction or Truncation Method

In this method, part of the numeric key is considered as an address for that corresponding key.

Example: Consider the following key. 123, 376, 478, 324. Now let us consider the table size is 100, i.e. the hash address can range from 0 to 99. Taking two numbers of the rightmost digit as the address for each corresponding key we get

Key	Address
123	23
376	76
478	78
324	24

We get the following hash values for the keys.

H(123) = 23
H(376) = 76
H(478) =78
H(324) =24

e. Radix Transformation Method

Using the radix transformation, the key *K* is expressed in a numerical system using a different radix. If *K* is the decimal number345, then its value in base 9 (nonal) is 423. This value is then divided modulo*TSize,* and the resulting number is used as the address of the location to which *K*should be hashed.

Example: Consider the following key. 345, 124, 276,500. Now let us consider the table size is 100, i.e. hash address can range frrm 0 to 99.

Key(decimal)	Radix 9	Address
345	423	23
124	147	47
276	336	36
500	615	15

We get the following hash values for the keys.

H(345) = 23
H(124) = 47
H(276) =36
H(500) =15

Collision Resolution Technique

A collision is said to be occurred when more than one key maps to same hash value in the hash table. The following ways resolves the collision.

1. Open Addressing / Probing
2. Chaining

1. Open Addressing / Probing

In this method, records that produce collision are stored in alternate position in the hash table. An alternate location is obtained by searching the hash table until an unused position is found. This process is called probing. The different probing sequences are explained below.

a. Linear Probing method

The simplest method to resolve collision is to start with the address where collision occurred and do a sequential search for the empty location. Hence this method searches in a straight line and hence called as linear probing. The array should be considered circular so that when the last location is reached, the search proceeds to the first location of the array.

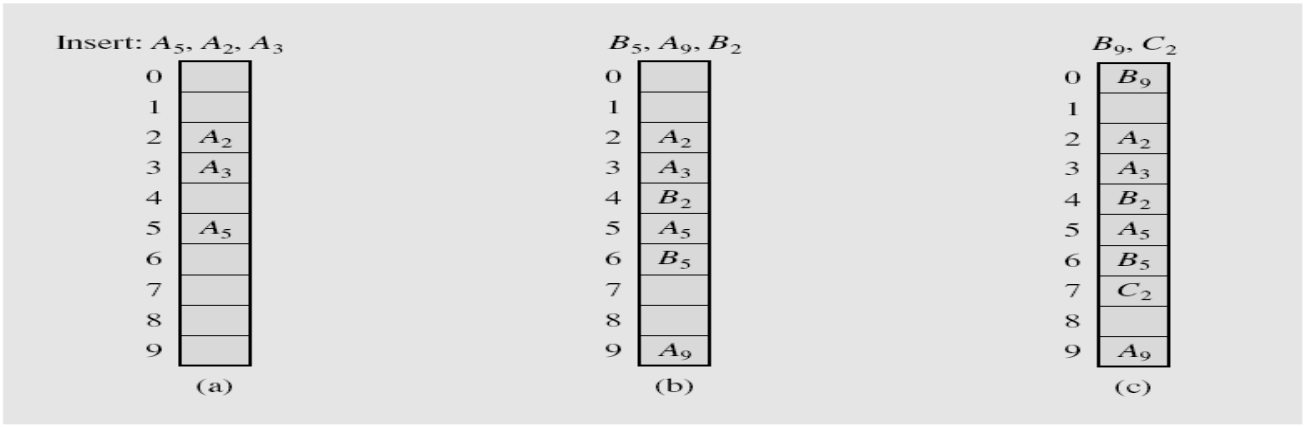
Suppose a record R with key K has address H(K) = h results collision then of searching is done in the location with address h, h+1, h+2……..h+i modulo division Tsize

Example 1: Suppose the size of table is TSize= 19, and assuming that h(K) = 9 for some K, the resulting sequence of probes is

9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 0, 1, 2, 3, 4, 5, 6, 7, 8

Example 2:

Resolving collisions with the linear probing method. Subscripts indicate the home positions of the keys being hashed.



b. Quadratic Probing method

Suppose a record R with key K has address H(K) = h results collision then instead of searching the location with address h, h+1, h+2……..h+i we linearly search the location with address like h, h+1, h-1, h+4, h-4, h+9, h-9……h+i^2, h-i^2 modulo division Tsize **or simpley** h, h+1, h+4, h+9…..h+i^2modulo division Tsize.

Note: if the hash value is negative then compute hash value as

Hash value = Tsize - address

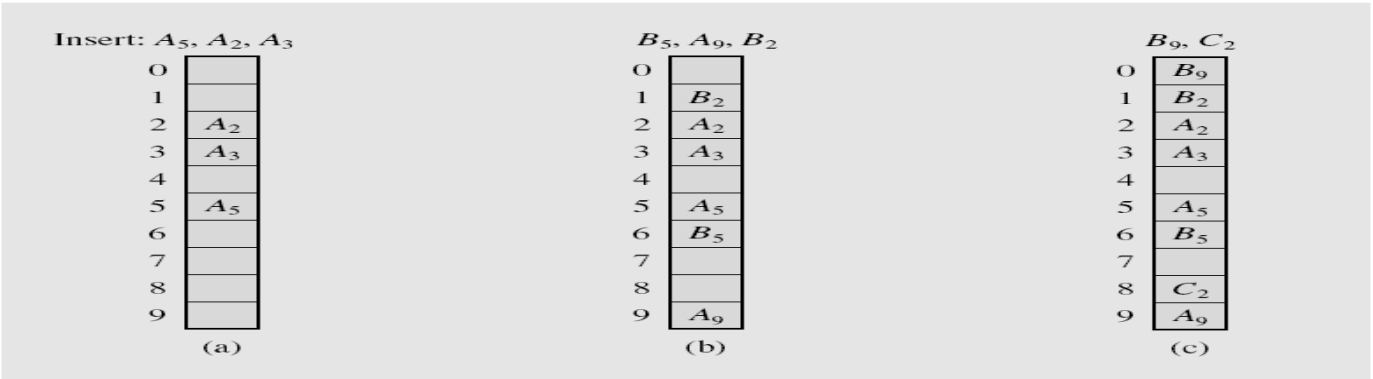
Example 1:

Suppose the size of table is TSize= 19, and assuming that h(K) = 9 for some K, the resulting sequence of probes is

9, 10, 8, 13, 5, 18, 0, 6, 12, 15, 3, 7, 11, 1, 17, 16, 2,14, 4

Example 2:

Using quadratic probing for collision resolution.



c. Double Hashing method

In this method, whenever there is a collision, a second hash function hp is applied to obtain alternate position. So this method utilizes two hash functions, one for accessing the primary position of a key, h, and a second function, hp, for resolving conflicts. The second hash function can be defined as

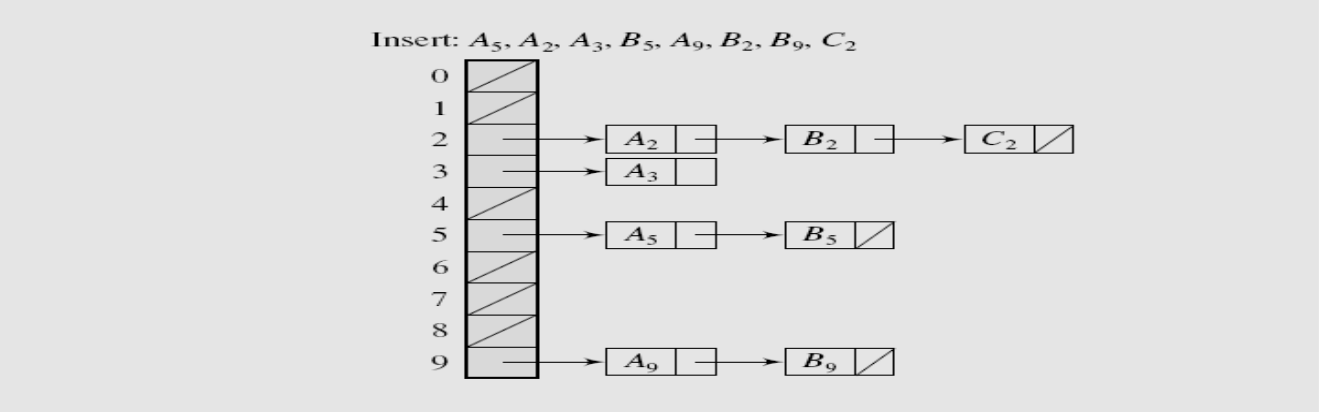
hp(K) = R = (K Mod R) where R is nearest prime number smaller than or equal to table size. Again if collision occurs then we probe linearly. However using two hash functions can be time-consuming, especially for sophisticated functions.

2. Chaining:

In this method, we use pointers (references) to resolve hash collision. Here keys do not have to be stored in the table itself. Here hash table is an array of pointers. Each position of table is associated with the first node of a distinct linked list or chain. All keys that hash to a particular position in the hash table are stored in the linked list corresponding to that particular position. Hence each linked list is a distinct list of keys whose keys have same hash value. One of the main advantages of this technique is that the table can never overflow, because the linked lists are extended only upon the arrival of new keys. However, if the linked list is very long then these list degrades the retrieval performance. Also another problem associated with this technique is that additional space is necessary for maintaining linked list.

Example:

In chaining, colliding keys are put on the same linked list.



Question:

1. For the given data 1, 16, 49, 36, 25, 64, 0, 81, 4 and 9. Show the contents of the hash table having the size 10 using

i. Linear probing for collision resolution
ii. Quadratic probing for collision resolution
iii. Double Hashing for collision resolution

Given primary hash function H(K) = K MOD 10 for all cases.

Solution:

Using linear probing, the final state of hash table would be

Location	Key
0	0
1	1
2	81
3	9
4	64
5	25
6	16
7	36
8	4
9	49

Collision occurs for the keys 36, 81, 4 and 9, so probing in the sequence h, h+1, h+2, h+3, h+i sequence for each keyuntil empty location is found.

Using quadratic probing, the final hash table would be

Location	Key
0	0
1	1
2	81
3	4
4	64
5	25
6	16
7	36
8	9
9	49

Collision occurs for the key 36, 81, 4 and 9 so probing in the sequence h, h+1, h -1, h+4, h-4, h+9, h-9…..h+i^2, h-$i^2$for each key until empty location is found.

OR

Location	Key
0	0
1	1
2	81
3	9
4	64
5	25
6	16
7	36
8	4
9	49

Collision occurs for the key 36, 81, 4 and 9 so probing in the sequence h, h+1, h+4, h+9…..h+i^2 for each key until empty location is found.

Using double hashing, the final state of hash table would be

Location	Key
0	0
1	1
2	9
3	81
4	64
5	25
6	16
7	36
8	4
9	49

Collision occurs for the key 36,81,4 and 9 so next probe sequence is computed by applying second hash function Hp(K) = R – (K MOD R), where R is nearest smallest prime number lesser or equal to table size.

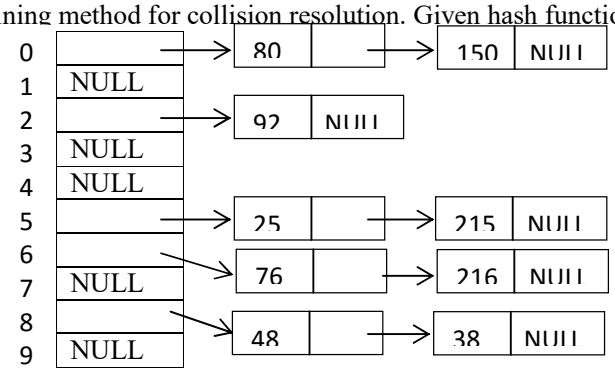
Hp(36) = 7 – (36 MOD 7) = 6, Since 6th location is not empty, so prove linearly in the sequence h,h+1,h+2…h+I until empty location is found

Hp(81) = 7 – (81 MOD 7) = 7 – 4 =3, Since 3rd location is empty, so insert key 36 at this place.

Hp(4) = 7 – (4 MOD 7) = 7 -4 = 3, Since 3rd location is not empty so probe linearly in sequence h, h+1, h+2…….. h+i until empty location is found.

Hp(9) = 7 – (9 MOD 7) = 7 – 2 = 5, Since 5th location is not empty so probe linearly in sequence h, h+1, h+2, h+i until empty location is found.

2. For the following keys 80, 92, 25, 76, 48, 150, 215, 216 and 38. Show the content of the hash table having table size 10 using chaining method for collision resolution. Given hash function as H(K) = K MOD 10.



Load Factor

Load factor is the ratio of numbers of elements in the table to the table size.

$$\text{Load Factor} \quad LF = \frac{\text{number of elements in the table}}{\text{table size}}$$

The performance of collision resolution technique depends upon load factor. The efficiency of hash function with a collision resolution procedure is measured by the average number of probes(key comparisons) needed to find the location. The efficiency mainly depends upon the load factor (lf).

Efficiency of Hashing

In hashing, we can access desired record in just one comparison in ideal case. Therefore the efficiency of hashing is ideally O(1).

The efficiency of hashing get reduced because of collision. So efficiency of hashing depends on how good or poor the hash function is. A hash function is said to be **good or perfect** if it results uniform distribution of records in hash table. A hash function is said to be poor, if it results a lot of collision.