Searching and Hashing

9

9.1 Sequential Search (Linear search)

Sequential search is performed in a linear way i.e. it starts from the beginning of the list and continues till we find the item or reach the end of list. The item to be searched is compared with each element of the list one by one, starting from the first element. For example, consider the array in figure 9.1, suppose we want to search for value 12 in this array. This value is compared with `arr[0]`, `arr[1]`, `arr[2]`,.....`arr[10]`, `arr[11]`. Since the value is found at index 11, the search is successful.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
96	19	85	9	16	29	2	36	41	67	53	12	66	6	75	82	89	23	93	45

Figure 9.1

Now suppose we have to find value 56 in the above array. This value is compared with `arr[0]`, `arr[1]`, `arr[2]`,.....`arr[19]`. The value was not found even after examining all the elements of the array so the search is unsuccessful. The program for sequential search is-

```
/*P9.1 Sequential search in an array*/
#include <stdio.h>
#define MAX 50

main()
{
        int i,n,item,arr[MAX],index;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        printf("Enter the elements : \n");
        for(i=0; i<n; i++)
                scanf("%d", &arr[i]);
        printf("Enter the item to be searched : ");
        scanf("%d", &item);
        index = LinearSearch(arr,n,item);
        if(index == -1)
                printf("%d not found in array\n",item);
        else
                printf("%d found at position %d\n",item,index);
}

int LinearSearch(int arr[],int n,int item)
{
        int i=0;
        while(i<n && item!=arr[i])
                i++;
        if(i<n)
                return i;
        else
                return -1;
}
```

The function `LinearSearch()` returns location of the item if found, or -1 otherwise.

The number of comparisons required to search for an item depends on the position of element inside the array. The best case is when the item is present at the first position and in this case only one comparison is done. The worst case occurs when the item is not present in the array and in this case n comparisons are required where n is the total number of elements. Searching for an item that is present at the i^{th} position requires i comparisons. Now let us find out the average number of comparisons required in a successful search assuming that the probability of searching of all elements is same.

(1 + 2 + 3 + 4 + ...+ n)/n = (n+1)/2

So in both average and worst case the run time complexity of linear search is O(n).

This search is good for data structures like linked lists, because no random access to elements is required. Sequential search in linked lists is given in chapter 3.

In the while loop of the `LinearSearch()` we are doing two comparisons in each iteration, the comparison `i<n` can be avoided by using a sentinel value. In the last location of the array i.e. `arr[n]`, we can temporarily place the value of item that is being searched. This value terminates the search when the search item is not present in the array.

Search 12

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
96	19	85	9	16	29	2	36	41	67	53	12	66	6	75	82	89	23	93	45	12

Search 56

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
96	19	85	9	16	29	2	36	41	67	53	12	66	6	75	82	89	23	93	45	56

The function for sequential search using sentinel is-

```
int LinearSearch(int arr[],int n,int item)
{
        int i=0;
        while(item!=arr[i])
                i++;
        if(i<n)
                return i;
        else
                return -1;
}
```

The number of comparisons in an unsuccessful search can be reduced if the array is sorted. In this case we need not search for the item till the end of the list, we can terminate our search as soon as we find an element that is greater or equal to the search item(in ascending array). If the element is equal to the search item the search is successful otherwise it is unsuccessful. The function for sequential search in an ascending order array is-

```
int LinearSearch(int arr[],int n,int item)
{
        int i=0;
        while(i<n && arr[i]<item)
                i++;
        if(arr[i]==item)
                return i;
        else
                return -1;
}
```

9.2 Binary Search

The prerequisite for binary search is that the array should be sorted. Firstly we compare the item to be searched with the middle element of the array. If the item is found there, our search finishes successfully otherwise the array is divided into two halves, first half contains all elements to the left of the middle element and the other one consists of all the elements to the right side of the middle element. Since the array is sorted, all the elements in the left half will be smaller than the middle element and the elements in the right half will be greater than the

middle element. If the item to be searched is less than the middle element, it is searched in left half otherwise it is searched in the right half.

Now search proceeds in the smaller portion of the array(subarray) and the item is compared with its middle element. If the item is same as the middle element, search finishes otherwise again the subarray is divided into two halves and the search is performed in one of these halves. This process of comparing the item with the middle element and dividing the array continues till we find the required item or get a portion which does not have any element.

To implement this procedure we will take 3 variables viz. `low`, `up` and `mid` that will keep track of the status of lower limit, upper limit and middle value of that portion of the array, in which we will search the element. If the array contains even number of elements, there will be two middle elements, we'll take first one as the middle element. The value of the `mid` can be calculated as-

```
mid = (low+up)/2 ;
```

The middle element of the array would be `arr[mid]`, the left half of the array would be `arr[low]`..... `arr[mid-1]` and the right half of the array would be `arr[mid+1]`.... `arr[up]`.

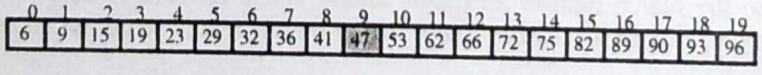
The item is compared with the mid element, if it is not found then the value of `low` or `up` is updated for selecting the left or right half. When `low` becomes greater than `up`, the search is unsuccessful as there is no portion left in which to search.

If item > arr[mid]
    Search will resume in right half which is arr[mid+1] .....arr[up]
    So low = mid+1, up remains same
If item < arr[mid]
    Search will resume in left half which is arr[low] .....arr[mid-1]
    up = mid-1, low remains same
If item == arr[mid], search is successful
    Item found at mid position
If low > up , search is unsuccessful
    Item not present in array

Let us take a sorted array of 20 elements and search for elements 41, 62, and 63 in this array one by one. The portion of array in which the element is searched is shown with a bold boundary in the figure.

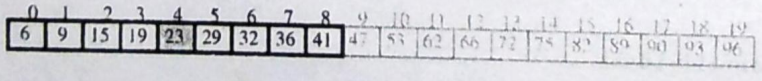
Search 41

low = 0, up = 19,
mid = (0+19)/2 = 9



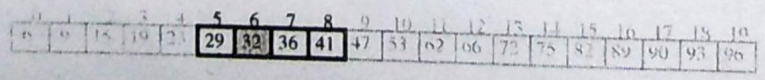
41 < 47
Search in left half
up = mid-1 = 8

low = 0, up = 8,
mid = (0+8)/2 = 4



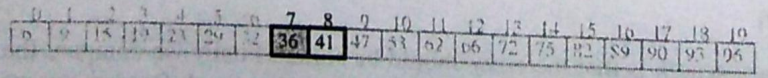
41 > 23
Search in right half
low = mid+1 = 5

low =5, up = 8,
mid = (5+8)/2 = 6



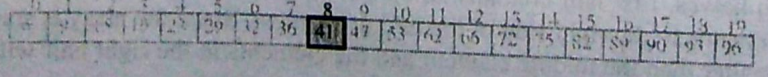
41 > 32
Search in right half
low = mid+1 = 7

low = 7, up = 8,
mid = (7+8)/2 = 7



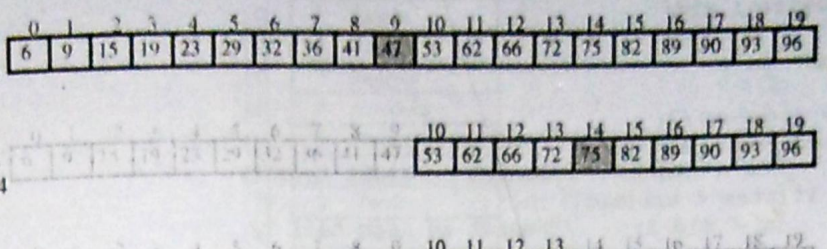
41 > 36
Search in right half
low = mid+1 = 8

low = 8, up = 8,
mid = (8+8)/2 = 8

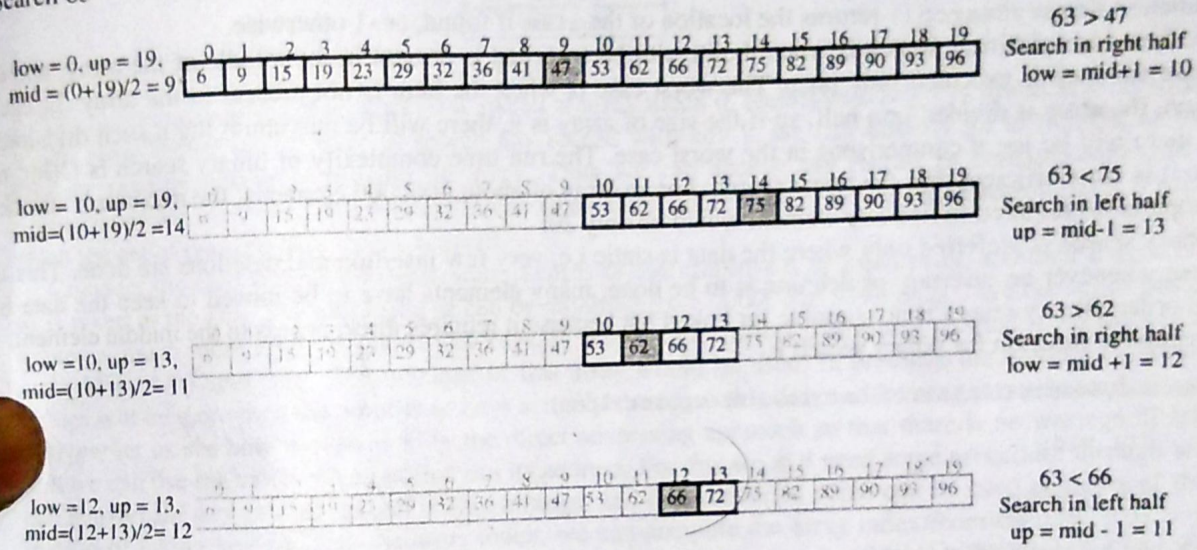


41 found

Search 62



Search 63



Now low=12 and up=11, the value of low has exceeded the value of up so the search is unsuccessful. The program for binary search is-

```
/*P9.4 Binary search in an array*/
#include <stdio.h>
#define MAX 50

main()
{
        int i,size,item,arr[MAX],index;
        printf("Enter the number of elements : ");
        scanf("%d",&size);
        printf("Enter the elements (in sorted order) : \n");

        for(i=0; i<size; i++)
                scanf("%d",&arr[i]);
        printf("Enter the item to be searched : ");
        scanf("%d",&item);
        index = BinarySearch(arr,size,item);
        if(index == -1)
                printf("%d not found in array\n",item);
        else
                printf("%d found at position %d\n",item,index);
}
```

```
int BinarySearch(int arr[],int size,int item)
{
        int low=0,up=size-1,mid;

        while(low<=up)
        {
                mid = (low+up)/2;
                if(item > arr[mid])
                        low = mid+1;    /*Search in right half */
                else if(item < arr[mid])
                        up = mid-1;     /*Search in left half */
                else
                        return mid;
        }
        return -1;
}
```

The function `BinarySearch()` returns the location of the `item` if found, or -1 otherwise.

The best case of binary search is when the item to be searched is present in the middle of the array, and in this case the loop is executed only once. The worst case is when the item is not present in the array. In each iteration, the array is divided into half, so if the size of array is n, there will be maximum log n such divisions. Thus there will be log n comparisons in the worst case. The run time complexity of binary search is O(log n) and so it is more efficient than the linear search. For an array of about 1000000 elements, the maximum number of comparisons required to find any element would be only 20.

Binary search is preferred only where the data is static i.e. very few insertion and deletions are done. This is because whenever an insertion or deletion is to be done, many elements have to be moved to keep the data in sorted order. Binary search is not suitable for linked list because it requires direct access to the middle element. The recursive function for binary search is-

```
int RBinarySearch(int arr[],int low,int up,int item)
{
        int mid;

        if(low>up)
                return -1;
        mid = (low+up)/2;
        if(item > arr[mid]) /*Search in right half */
                RBinarySearch(arr,mid+1,up,item);
        else if(item < arr[mid])/*Search in left half */
                RBinarySearch(arr,low,mid-1,item);
        else
                return mid;
}
```

9.3 Hashing

We have seen different searching techniques where search time is dependent on the number of elements. Sequential search, binary search and all the search trees are totally dependent on number of elements and many key comparisons are involved. Now we'll see another approach where less key comparisons are involved and searching can be performed in constant time i.e. search time is independent of the number of elements.

Suppose we have keys which are in the range 0 to n-1, and all of them are unique. We can take an array of size n, and store the records in that array based on the condition that key and array index are same. For example, suppose we have to store the records of 15 students and each record consists of roll number and name of the student. The roll numbers are in the range 0 to 14 and they are taken as keys. We can take an array of size 15 and store these records in it as-

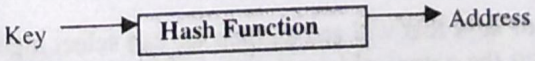
[0]	0	Devanshi
[1]	1	Saachi
[2]	2	Raghav
[3]	3	Supreet
[4]	4	Anushka
[5]	5	Prajwal
[6]	6	Shivani
[7]	7	Parul
[8]	8	Shriya
[9]	9	Samarth
[10]	10	Arnav
[11]	11	Dyuthi
[12]	12	Anjali
[13]	13	Sanjana
[14]	14	Niharika

Figure 9.2

The record with key(roll number) 0 is stored at array index 0, record with key 1 is stored at array index 1 and so on. Now whenever we have to search any record with key k, we can directly go to index k of the array, because random access is possible in array. Thus we can access any record in constant time and no key comparisons are involved. This method is known as direct addressing or key-indexed search but it is useful only when the set of possible key values is small.

Consider the case when we have to store records of 500 employees of a company, and their 6 digit employee id is taken as the key. The employee id can be anything from 000000 to 999999, so here the set of possible key values is much more than the number of keys. If we adopt the direct addressing method, then we will need an array of size 10^6 and only 500 locations of this array would be used. In practice, the number of possible key values will be more than the number of keys actually stored, so this direct addressing is rarely used.

Now let us see how we can modify the direct addressing approach so that there is no wastage of space and still we can use the value of key to find out its address. For this we will need some procedure through which we can convert the key into an integer within a range, and this converted value can be used as index of the array. Instead of taking key equal to the array index, we can compute the array index from the key. This process of converting a key to an address(index of array) is called hashing or key to address transformation and it is done through hash function. A hash function is used to generate an address from a key or we can say that each key is mapped on a particular array index through hash function. The hash function takes a key as input and returns the hash value of that key which is used as the address for storing the key in the array. Keys may be of any type like integers, strings etc but hash value will always be an integer.



We can write this as-

`h(k) = a`

Here `h` is the hash function, `k` is the key and `a` is the hash value of the key `k`. Now the key `k` can be stored at array index `a`, which is also known as its home address. This process of generating addresses from keys is called hashing the key and the array in which insertion and searching is done through hashing is called hash table. Each entry of hash table consists of a key and the associated record.

For inserting a record, we generate an address(index) by applying hash function to the key, and insert the record at that address. For accessing any record, we apply the same hash function to the key and then access the record at the address given by hash function.

Let us take the example of storing records of books where ISBN numbers of the books are keys. Suppose we take a hash function that maps a key to address by adding the digits of the key. For example if ISBN is 8183330487, then the record of this book will be stored at index 45. The figure 9.3 shows the records of 4 books stored in a hash table.

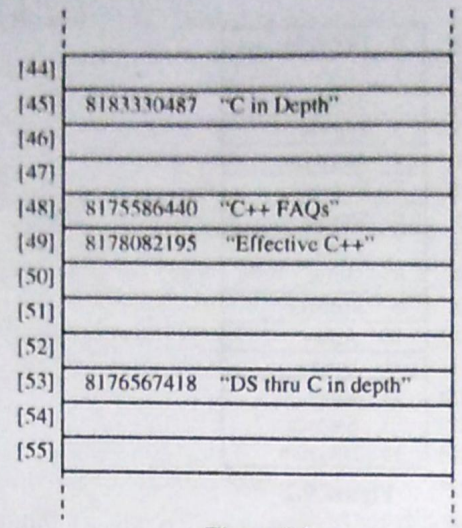


Figure 9.3

Now suppose we have to store a book with ISBN 8173380843. The sum of the digits is 45, i.e. the address given by our hash function is 45, but this address is already occupied. This situation is called **collision**. Collision occurs when the hash function generates the same address for different keys. The keys which are mapped to same address are called **synonyms**. For resolving collision we have different collision resolution techniques about which we will study in detail later in this chapter. Ideally a hash function should give unique addresses for all keys but this is practically not possible. So we should try to select a hash function that minimizes collision. While making algorithms that use hashing we have to mainly focus on these two things-

1. Selecting a hash function that converts keys to addresses.
2. Resolving the collision.

9.3.1 Hash functions

Now we know that hash function generates a table address from a given key. It works like mapping interface between key and hash table. If the size of hash table is m, then we need a hash function that can generate addresses in the range 0 to m-1. Basically we have two criteria for choosing a good hash function-

1. It should be easy to compute.
2. It should generate addresses with minimum collision i.e. it should distribute the keys as uniformly as possible in the hash table.

If we know beforehand about the types of keys that will arrive, then we can select a good hash function that gives minimum collision for these keys. But the nature of keys is generally not known in advance; hence the general approach is to take a function that can work on the key such that addresses generated are distributed randomly.

Now we will see some techniques for making hash functions. In all hashing functions that we will discuss, we will assume key to be an integer. If the key is not an integer then it can easily be converted into one. For example an alphanumeric key can be converted to an integer by using the ASCII values of the characters.

9.3.1.1 Truncation (or Extraction)

This is the easiest method for computing address from a key. Here we take only a part of the key as address, for example we may take some rightmost digits or leftmost digits. Let us take some 8 digit keys and find addresses for them.

82394561, 87139465, 83567271, 85943228

Suppose table size is 100 and we decide to take 2 rightmost digits for getting the hash table address, so the addresses of above keys will be 61, 65, 71 and 28 respectively.

This method is easy to compute but chances of collision are more because last two digits can be same in many keys.

9.3.1.2 Midsquare Method

In mid square method, the key is squared and some digits or bits from the middle of this square are taken as address. Generally the selection of digits depends on the size of the table. It is important that the same digits should be selected from the squares of all the keys. Suppose our keys are 4 digit integers and the size of table is 1000; we will need 3 digit addresses. We can square the keys and take the 3rd, 4th and 5th digits from each squared number as the hash address for the corresponding key.

Key:	1337	1273	1391	1026
Squares of key:	1787569	1620529	1934881	1052676
Address:	875	205	348	526

If the key is too large to square then we can take a part of the key and perform midsquare method on that part rather than on the whole key. For example if we have 7 digit keys, we can take a part of the key which contains first 4 digits of the key and square that part.

9.3.1.3 Folding Method

In this method, the key is broken into different parts where the length of each part is same as that of the required address, except possibly the last part. After this these parts are shifted such that the least significant digits of all parts are in same line and then these parts are added. The address of the key can then be obtained by ignoring the final carry in the sum. Suppose we have a table of size 1000, and we have to find address for a 12 digit key. Since the address should be of 3 digits, we will break the key in parts containing 3 digits.

738239456527 => 738 239 456 527

After this, these parts are shifted and added.

```
  738
  239
  456
+ 527
 1960
```

Now ignoring the final carry 1, the hash address for the key is 960. This technique is called shift folding; it can be modified to get another folding technique called boundary folding. In this method, the key is assumed to be written on a paper which is folded at the boundaries of the parts of the key, so all even parts are reversed before addition.

```
  738
  932  (Reversed)
  456
+ 725  (Reversed)
 2851
```

Now ignoring the final carry 2, the hash address for the key is 851.

9.3.1.4 Division Method(Modulo-Divison)

In this method, the key is divided by the table size and the remainder is taken as the address for hash table. In C language, this operation is provided by % operator. This method ensures that we will get the address in the limited range of the table. If the size of table is m, then we will get the addresses in the range {0, 1,....., m-1}.

$H(k) = k \bmod m$

Collisions can be minimized if the table size is taken to be a prime number. Let us take table of size 97 and see how the following keys will be inserted in it.

82394561, 87139465, 83567271

82394561 % 97 = 45
87139465 % 97 = 0
83567271 % 97 = 25

So here the hash address of above keys will be 45, 0 and 25.

We can combine other hashing methods with division method and it will ensure that addresses are in the range of hash table.

9.3.2 Collision Resolution

An ideal hash function should perform one to one mapping between set of all possible keys and all hash table addresses but this is almost impossible, and no hash function can totally prevent collisions. A collision occurs whenever a key is mapped to an address that is already occupied, and the different collision resolution techniques suggest for an alternate place where this key can be placed. The two collision resolution techniques that we will study are-

1. Open Addressing (Closed Hashing)
2. Separate Chaining (Open Hashing)

Any of these collision resolution techniques can be used with any hash function.

In the process of searching, the given key is compared with many keys and each key comparison is known as a probe. The efficiency of a collision resolution technique is defined in terms of the number of probes required to find a record with a given key.

9.3.2.1 Open Addressing (Closed Hashing)

In open addressing, the key which caused the collision is placed inside the hash table itself but at a location other than its hash address. Initially a key value is mapped to a particular address in the hash table. If that address is already occupied then we will try to insert the key at some other empty location inside the table. The array is assumed to be closed and hence this method is named as closed hashing. We will study three methods to search for an empty location inside the table.

(i) Linear Probing
(ii) Quadratic Probing
(iii) Double Hashing

9.3.2.1.1 Linear Probing

If address given by hash function is already occupied, then the key will be inserted in the next empty position in the hash table. If the address given by hash function is a and it is not empty, then we will try to insert the key at next location i.e. at address `a+1`. If address `a+1` is also occupied then we will try to insert at next address i.e. `a+2` and we will keep on trying successive locations till we find an empty location where the key can be inserted. While probing the array for empty positions, we assume that the array is closed or circular i.e. if array size is N then after
$(N-1)^{th}$ position, search will resume from 0^{th} position of the array.

If a function h returns address 5 for a key and table size is 11, then we will search for empty locations in this sequence - 5, 6, 7, 8, 9, 10, 0, 1, 2, 3, 4. If location 5 is empty, there will be no collision and we can insert the key there, otherwise we will linearly search these locations and insert the key when we find an empty one. Note that we have taken our array to be closed so after the last location(10^{th}) we probe the first location of array(0^{th}).

Let us take a table of size 11, and insert some keys in it taking a hash function-

h(key) = key % 11

h(29) = 29%11 = 7
h(46) = 46%11 = 2
h(18) = 18%11 = 7
h(36) = 36%11 = 3
h(43) = 43%11 = 10
h(21) = 21%11 = 10
h(24) = 24%11 = 2
h(54) = 54%11 = 10

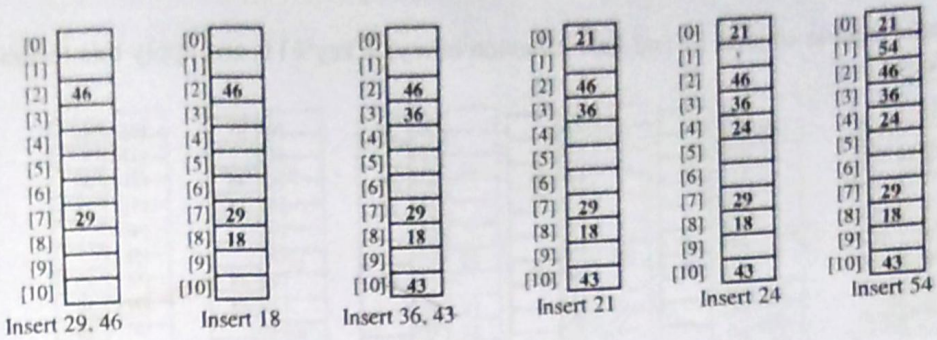


Figure 9.4 Linear Probing

First 29 and 46 are inserted at 7th and 2nd positions of the array respectively. Next 18 is to be inserted but its hash address 7 is already occupied so it will be inserted in the next empty location which is 8th position. After this 36 and 43 are inserted without any collision. Now 21 is to be inserted but its hash address 10 is not empty, so it is inserted in next empty location which is 0th position. Now 24 is to be inserted whose hash address is 2 which is not empty, so next location 3 is probed which is also not empty, and finally it is inserted in the next empty position which is 4th position. In a similar manner 54 is inserted in 1st position.

The formula for linear probing can be written as-

H(k, i) = (h(k) + i) mod Tsize

Here the value of `i` varies from 0 to `Tsize-1`. We have done mod `Tsize` so that the resulting address does not cross the valid range of array indices. So we will search for empty locations in the sequence-

h(key), h(key)+1, h(key)+2, h(key)+3 ..... all modulo Tsize

Let us take the example of inserting 54 in the above table-

H(54, 0) = (10 + 0) % 11 = 10 (not empty)
H(54, 1) = (10 + 1)% 11 = 0 (not empty)
H(54, 2) = (10 + 2)% 11 = 1 (Empty, Insert the key)

This is the way insertion is done in case of linear probing. For searching a key, first we check the hash address position in the table, if key is not available at that position, then we sequentially search the keys after that hash address position. The search terminates if we get the key or reach an empty location or reach the position where we had started. In the last two cases the search is unsuccessful.

The main disadvantage of the linear probing technique is primary clustering. When about half of the table is full, there is tendency of cluster formation i.e. groups of records stored next to each other are created. In the above example a cluster of indices 10, 0, 1, 2, 3, 4 is formed. If a key is mapped to any of these indices then it will be stored at index 5 and the cluster will become bigger. Suppose a key is mapped to index 10, then it will be stored at 5, far away from its home address. The number of probes for inserting or searching this key will be 7. Clustering increases the number of probes to search or insert a key, and hence the search and insertion times of the records also increase.

9.3.2.1.2 Quadratic Probing

In linear probing, the colliding keys are stored near the initial collision point, resulting in formation of clusters. In quadratic probing this problem is solved by storing the colliding keys away from the initial collision point. The formula for quadratic probing can be written as-

H(k, i) = (h(k) + i^2) mod Tsize

The value of `i` varies from 0 to `Tsize-1` and `h` is the hash function. Here also the array is assumed to be closed. The search for empty locations will be in the sequence-

h(k), h(k)+1, h(k)+4, h(k)+9, h(k)+16, ... all mod Tsize

Let us take a table of size 11 and hash function h(key) = key%11, and apply this technique for inserting the following keys.

h(46) = 46%11 = 2
h(28) = 28%11 = 6
h(21) = 21%11 = 10
h(35) = 35%11 = 2
h(57) = 57%11 = 2
h(39) = 39%11 = 6
h(19) = 19%11 = 8
h(50) = 50%11 = 6

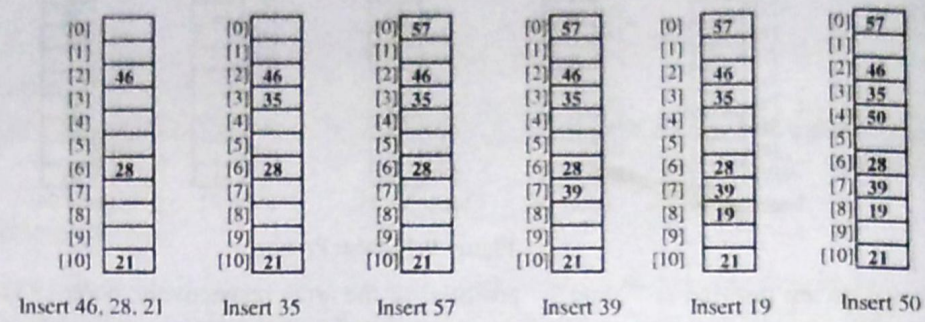


Figure 9.5 Quadratic Probing

Keys 46, 28, 21 are inserted without any collision. Now 35 is to be inserted whose hash address is 2, which is not empty so next location (2+1)%11 = 3 is tried and it is empty so 35 is inserted in location 3. To insert 57, first location 2 is tried it is not empty so next location (2+1)%11 = 3 is tried, it is also not empty so next location (2+4)%11 = 6 is tried, it is also not empty so next location (2+9)%11 = 0 is tried and it is empty so 57 is inserted there. Similarly key 39 is inserted at position 7. The key 19 is inserted without any collision. The key 50 is inserted at location 4.

H(50, 0) = (6 + 0) % 11 = 6 (not empty)
H(50, 1) = (6 + 1^2) % 11 = 7 (not empty)
H(50, 2) = (6 + 2^2) % 11 = 10 (not empty)
H(50, 3) = (6 + 3^2) % 11 = 4 (Empty, Insert the key)

Quadratic probing does not have the problem of primary clustering as in linear probing, but it gives another type of clustering problem known as secondary clustering. Keys that have same hash address will probe the same sequence of locations leading to secondary clustering. For example in the above table, keys 46, 35 and 57 are all mapped to location 2 by the hash function h, and so the locations that will be probed in each case are same.

46 - 2, 3, 6, 0, 7, 5,.....
35 - 2, 3, 6, 0, 7, 5,.....
57 - 2, 3, 6, 0, 7, 5,.....

In secondary clustering, clusters are not formed by records which are next to each other but they are formed by records which follow the same collision path.

Another limitation of quadratic probing is that it can't access all the positions of the table. An insert operation may fail in spite of empty locations inside the hash table. This problem can be alleviated by taking size of hash table to be a prime number, and in that case at least half of the locations of the hash table will be accessed. In linear probing, an insert operation will fail only when the table is fully occupied.

9.3.2.1.3 Double Hashing

In double hashing, the increment factor is not constant as in linear or quadratic probing, but it depends on the key. The increment factor is another hash function and hence the name double hashing. The formula for double hashing can be written as-

H(k, i) = (h(k) + i h'(k)) mod Tsize

The value of `i` varies from 0 to `Tsize-1` and `h` is the hash function, `h'` is the secondary hash function. The search for empty locations will be in the sequence-

h(k), h(k)+ h'(k), h(k)+ 2 h'(k), h(k)+ 3 h'(k), ... all mod Tsize

Let us see how some keys can be inserted in the table taking these two functions-

h(k) = key%11

h'(k) = 7 - (key % 7)

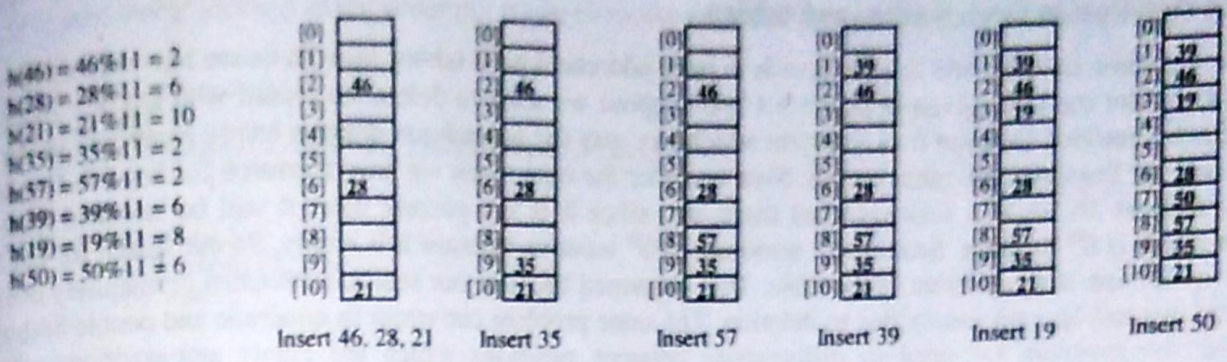


Figure 9.6 Double Hashing

(i) Insertion of 46, 28, 21 - No collision, all are inserted at their home addresses

(ii) Insertion of 35 – Collision at 2.
Next probe is done at - (2 + 1(7-35%7)) %11 = 9. Location 9 is empty, so 35 is inserted there

(iii) Insertion of 57 – Collision at 2
Next probe is done at - (2 + 1(7 - 57%7)) % 11 = 8, Location 8 is empty, so 57 is inserted there

(iv) Insertion of 39 – Collision at 6
Next probe is done at - (6 + 1(7-39%7))%11 = 9, Location 9 is not empty
Next probe is done at - (6 + 2(7-39%7))%11 = 1, Location 1 is empty, so 39 is inserted there

(v) Insertion of 19 – collision at 8
Next probe is done at - (8 + 1(7-19%7))%11 = 10, Location 10 is not empty
Next probe is done at - (8 + 2(7-19%7))%11 = 1 Location 1 is not empty
Next probe is done at - (8 + 3(7-19%7))%11 = 3, Location 3 is empty, so 19 is inserted there

(vi) Insertion of 50 – Collision at 6
Next probe is done at - (6 + 1(7- 50%7))%11 = 1, Location 1 is not empty
Next probe is done at - (6 + 2(7- 50%7))%11 = 7, Location 7 is empty, so 50 is inserted there

The problem of secondary clustering is removed in double hashing because keys that have same hash address probe different sequence of locations. For example 46, 35, 57 all hash to same address 2, but their probe sequences are different.

46 – 2, 5, 8, 0, 3, 6,.....
35 – 2, 9, 5, 1, 8, 4.....
57 – 2, 8, 3, 9, 4, 10.....

While selecting the secondary hash function we should consider these two points – first that it should never give a value of 0, and second that the value given by secondary hash function should be relatively prime to the table size.

Double hashing is more complex and slower than linear and quadratic probing because it requires two time calculation of hash function.

Load factor of a hash table is generally denoted by λ and is calculated as-

$\lambda = n/m$

Here n is the number of records, and m is the number of positions in the hash table.

In open addressing the load factor is always less than 1, as the number of records cannot exceed the size of the table. To improve efficiency, it is desirable that there should always be some empty locations inside the table i.e. the size of table should be more than the number of actual records to be stored. If a table becomes dense i.e. load factor is close to 1, then there will be more chances of collision hence deteriorating the search efficiency. Leaving some locations empty is wastage of space but improves search time so it is a space vs. time tradeoff.

A disadvantage of open addressing is that key values are far from their home addresses, which increases the number of probes. Another problem is that hash table overflow may occur.

9.3.2.1.4 Deletion in open addressed tables

We have seen how to insert and search records in open addressed hash tables, now let us see how records can be deleted. Consider the table given in figure 9.4 and suppose we want to delete the record with key 21. Suppose we do this by marking location 0 as empty(in whichever way the application denotes empty locations) so that it can be used for inserting any other record. Now consider the case when we have to search for key 54. This key maps to address 10, so it is first searched there, and since it is not present there it will be searched in next location which is 0^{th} location. Search will terminate at 0^{th} location because it is empty. So our search for key 54 failed even though it was present in the table. This happened because our search terminated prematurely due to a location that had become empty due to deletion. The same problem can occur in quadratic and double hashing. To avoid this problem we need to differentiate between positions which are empty and positions which previously contained a record but now are empty(vacated) due to deletion. The searching should not terminate when we reach a location that is empty because a record was deleted from there. In the implementation we have done this by a taking a variable `status` in each record.

9.3.2.1.5 Implementation of open Addressed tables

In our program, we will store information of employees in a hash table, and the employee ID of an employee is taken as the key. We will declare a structure `employee` as -

```
struct employee
{
        int empid;
        char name[20];
        int age;
};
```

We will declare another structure named `Record`.

```
struct Record
{
        struct employee info;
        enum type_of_record status;
};
```

Here variable `status` is of type `enum type_of_record` and it can have one of these values - `EMPTY`, `DELETED`, `OCCUPIED`.

After this we will declare an array of type `Record` and this is our hash table.

```
struct Record table[MAX];
```

First of all we need to initialize the locations of the array to indicate that they are empty. This is done by setting the status field of all the array records to EMPTY.

If `table[i]` contains a record then value of `table[i].status` will be `OCCUPIED`, if `table[i]` is empty because of deletion of a record, i.e. it is empty now but was previously occupied then value of `table[i].status` will be `DELETED`, otherwise if `table[i]` is empty(was never occupied) then value of `table[i].status` will be `EMPTY`.

The function `search()` returns -1 if key is not found otherwise it returns the index of the array where key was found. Whenever we search for a key, our search will terminate only when we reach a location `table[i]` whose `status` is `EMPTY`. While inserting, we can stop and insert a new record whenever we reach a location `table[i]` whose `status` is `EMPTY` or `DELETED`. Deletion of a record stored at location `table[i]` is done by setting status to `DELETED`.

The search will terminate when an `EMPTY` location is encountered or the key is found. Thus it is important that at least one location in the table is left `EMPTY` so that the search can terminate if key is not present.

If frequent deletions are performed, then there will be many locations which will be marked as DELETED and this will increase the search time. In this case we'll have to reorganize the table by reinserting all the records in an empty hash table. So if many deletions are to be performed then open addressing is not a good choice. We have used linear probing in the program; it can be easily modified for quadratic or double hashing.

```
/*p9.6 Linear probing*/
#include <stdio.h>
#define MAX 50
enum type_of_record {EMPTY, DELETED,OCCUPIED};
struct employee
{
        int empid;
        char name[20];
        int age;
};
struct Record
{
        struct employee info;
        enum type_of_record status;
};
void insert(struct employee emprec,struct Record table[]);
int search(int key,struct Record table[]);
void del(int key,struct Record table[]);
void display(struct Record table[]);
int hash(int key);
main()
{
        int i,key,choice;
        struct Record table[MAX];
        struct employee emprec;
        for(i=0; i<=MAX-1; i++)
                table[i].status = EMPTY;

        while(1)
        {
                printf("1.Insert a record\n");
                printf("2.Search a record\n");
                printf("3.Delete a record\n");
                printf("4.Display table\n");
                printf("5.Exit\n");
                printf("Enter your choice\n");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1 :
                           printf("Enter empid,name,age : ");
                           scanf("%d%s%d",&emprec.empid,emprec.name,&emprec.age);
                           insert(emprec,table);
                           break;
                        case 2 :
                           printf("Enter a key to be searched : ");
                           scanf("%d",&key);
                           i = search(key,table);
                           if(i==-1)
                                printf("Key not found\n");
                           else
                                printf("Key found at index %d\n",i);
                           break;
                        case 3:
                           printf("Enter a key to be deleted\n");
                           scanf("%d",&key);
                           del(key,table);
```

```
                        break;
                    case 4:
                        display(table);
                        break;
                    case 5:
                        exit(1);
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
int search(int key,struct Record table[])
{
        int i,h,location;
        h = hash(key);
        location = h;

        for(i=1; i!=MAX-1; i++)
        {
                if(table[location].status == EMPTY)
                        return -1;
                if(table[location].info.empid == key)
                        return location;
                location = (h+i)%MAX;
        }
        return -1;
}/*End of search()*/
void insert(struct employee emprec, struct Record table[])
{
        int i,location,h;
        int key = emprec.empid;        /*Extract key from the record*/
        h = hash(key);
        location = h;

        for(i=1; i!=MAX-1; i++)
        {
         if(table[location].status==EMPTY || table[location].status==DELETED)
         {
                table[location].info = emprec;
                table[location].status = OCCUPIED;
                printf("Record inserted\n\n");
                return;
         }
         if(table[location].info.empid == key)
         {
                printf("Duplicate key\n\n");
                return;
         }
         location = (h+i)%MAX;
        }
        printf("Record can't be inserted : Table overFlow\n\n");
}/*End of insert()*/
void display(struct Record table[])
{
        int i;
        for(i=0; i<MAX; i++)
        {
                printf("[%d]   : ",i);
                if(table[i].status==OCCUPIED)
                {
                 printf("Occupied:%d  %s",table[i].info.empid,table[i].info.name);
                 printf("    %d\n", table[i].info.age);
                }
                else if(table[i].status==DELETED)
                        printf("Deleted\n");
```

```
            else
                    printf("Empty\n");
        }
}/*End of display()*/
void del(int key,struct Record table[])
{
        int location = search(key, table);
        if(location == -1)
                printf("Key not found");
        else
                table[location].status = DELETED;
}/*End of del()*/
int hash(int key)
{
        return(key%MAX);
}/*End of hash()*/
```

9.3.2.2 Separate chaining

In this method, linked lists are maintained for elements that have same hash address. Here the hash table does not contain actual keys and records but it is just an array of pointers, where each pointer points to a linked list. All elements having the same hash address i will be stored in a separate linked list, and the starting address of that linked list will be stored in the index i of the hash table. So array index i of the hash table contains a pointer to the list of all elements that share the hash address i. Each element of linked list will contain the whole record with key. These linked lists are referred to as chains and hence the method is named as separate chaining. Let us take an example and see how collisions can be resolved using separate chaining. Suppose we take a hash table of size 7 and hash function H(key) = key%7.

H(4895) = 4895 % 7 = 2
H(6559) = 6559 % 7 = 0
H(5912) = 5912 % 7 = 4
H(4047) = 4047 % 7 = 1
H(6766) = 6766 % 7 = 4
H(4390) = 4390 % 7 = 1
H(4640) = 4640 % 7 = 6
H(4900) = 4900 % 7 = 0
H(4411) = 4411 % 7 = 1

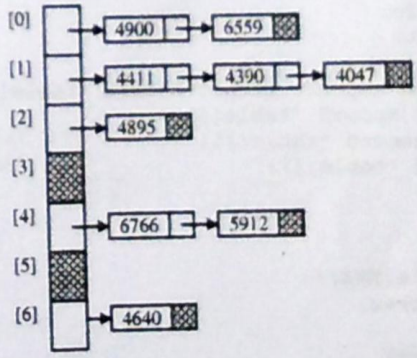


Figure 9.7 Separate Chaining

Here keys 4411, 4390, 4047 all hash to the same address i.e. array index 1, so they are all stored in a separate linked list whose starting address is stored in location 1 of the array. Similarly other keys are also stored in respective lists depending on their hash addresses.

If linked lists are short, performance is good but if lists become long then it takes time to search a given key in any list. To improve the retrieval performance we can maintain the lists in sorted order.

For inserting a key, first we will get the hash value through hash function, then that key will be inserted in the beginning of the corresponding linked list. Searching a key is also same, first we will get the hash value through hash function, and then we will search the key in corresponding linked list. For deleting a key, first that key will be searched and then the node holding that key will be deleted from its linked list.

In open addressing, accessing any record involved comparisons with keys which had different hash values which increased the number of probes. In chaining, comparisons are done only with keys that have same hash values.

In open addressing, all records are stored inside the hash table itself so there can be problem of hash table overflow and to avoid this, enough space has to be allocated at the compilation time. In separate chaining, there will be no problem of hash table overflow because linked lists are dynamically allocated so there is no limitation on the number of records that can be inserted. It is not necessary that the size of table be more than the number of records. Separate chaining is best suited for applications where the number of records is not known in advance.

In open addressing, it is best if some locations are always empty. If records are large then this results in wastage of space. In chaining there is no wastage of space because the space for records is allocated when they arrive.

Implementation of insertion and deletion is simple in separate chaining. The main disadvantage of separate chaining is that it needs extra space for pointers. If there are n records and the table size is m, then we need extra space for n+m pointers. If the records are very small then this extra space can prove to be expensive.

In separate chaining the load factor denotes the average number of elements in each list and it can be greater than 1.

```
/*P9.9 Separate chaining*/
#include <stdio.h>
#include<stdlib.h>
#define MAX 11

struct employee
{
        int empid;
        char name[20];
        int age;
};
struct Record
{
        struct employee info;
        struct Record *link;
};
void insert(struct employee emprec, struct Record *table[]);
int search(int key, struct Record *table[]);
void del(int key, struct Record *table[]);
void display(struct Record *table[]);
int hash(int key);

main()
{
        struct Record *table[MAX];
        struct employee emprec;
        int i,key,choice;
        for(i=0;i<=MAX-1;i++)
                table[i] = NULL;

        while(1)
        {
                printf("1.Insert a record\n");
                printf("2.Search a record\n");
                printf("3.Delete a record\n");
                printf("4.Display table\n");
                printf("5.Exit\n");
                printf("Enter your choice\n");
                scanf("%d",&choice);

                switch(choice)
                {
                        case 1 :
                                printf("Enter the record\n");
                                printf("Enter empid, name, age : ");
                                scanf("%d%s%d",&emprec.empid,emprec.name,&emprec.age);
                                insert(emprec,table);
```

```
                    break;
                case 2 :
                    printf("Enter a key to be searched : ");
                    scanf("%d",&key);
                    i = search(key,table);
                    if(i==-1)
                      printf("Key not found\n");
                    else
                      printf("Key found in chain %d\n",i);
                    break;
                case 3:
                    printf("Enter a key to be deleted\n");
                    scanf("%d",&key);
                    del(key,table);
                    break;
                case 4:
                    display(table);
                    break;
                case 5:
                    exit(1);
            }
        }
}/*End of main()*/
void insert(struct employee emprec,struct Record *table[])
{
        int h,key;
        struct Record *tmp;

        key = emprec.empid;  /*Extract the key from the record*/
        if(search(key, table)!=-1)
        {
              printf("Duplicate key\n");
              return;
        }
        h = hash(key);

        /*Insert in the beginning of list h*/
        tmp=malloc(sizeof(struct Record));
        tmp->info = emprec;
        tmp->link = table[h];
        table[h] = tmp;
}/*End of insert()*/
void display(struct Record *table[])
{
        int i;
        struct Record *ptr;

        for(i=0; i<MAX; i++)
        {
           printf("\n[%d]  ", i);
           if(table[i]!=NULL)
           {
                ptr = table[i];
                while(ptr!=NULL)
                {
                      printf("%d %s %d\t",ptr->info.empid,ptr->info.name,ptr->info.age);
                      ptr = ptr->link;
                }
           }
        }
        printf("\n");
}/*End of display()*/
int search(int key,struct Record *table[])
```

```
{
        int h;
        struct Record *ptr;
        h = hash(key);
        ptr = table[h];
        while(ptr!=NULL)
        {
                if(ptr->info.empid == key)
                        return h;
                ptr = ptr->link;
        }
        return -1;
}/*End of search()*/
void del(int key,struct Record *table[])
{
        int h;
        struct Record *tmp, *ptr;

        h = hash(key);
        if(table[h]==NULL)
        {
                printf("Key %d not found\n",key);
                return;
        }
        if(table[h]->info.empid == key)
        {
                tmp=table[h];
                table[h]=table[h]->link;
                free(tmp);
                return;
        }
        ptr = table[h];

        while(ptr->link!=NULL)
        {
                if(ptr->link->info.empid == key)
                {
                        tmp=ptr->link;
                        ptr->link=tmp->link;
                        free(tmp);
                        return;
                }
                ptr=ptr->link;
        }
        printf("Key %d not found\n",key);
}/*End of del()*/
int hash(int key)
{
        return(key%MAX);
}/*End of hash()*/
```

9.3.3 Bucket Hashing

This technique postpones collisions but does not resolve them completely. Here hash table is made up of buckets, where each bucket can hold multiple records. There is one bucket at each hash address and this means that we can store multiple records at a given hash address. Collisions will occur only after a bucket is full. In the example given below we have taken a hash table in which each bucket can hold three records. So we can store three records at the same hash address without collision. A collision occurs only when a fourth record with the same hash address arrives.

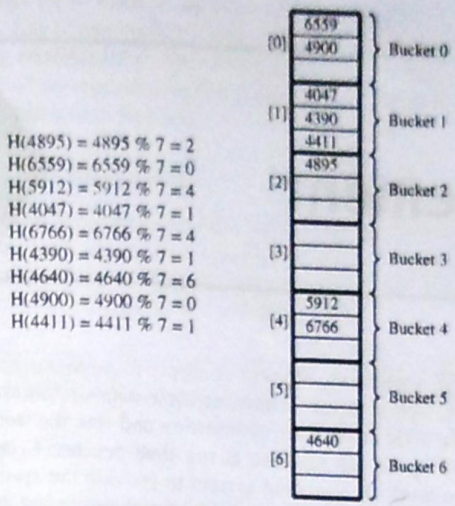


Figure 9.8 Bucket Hashing

A disadvantage of this technique is wastage of space since many buckets will not be occupied or will be partially occupied. Another problem is that this method does not prevent collisions but only defers them, and when collisions occur they have to be resolved using a collision resolution technique like open addressing.

Exercise

1. An array contains the following 18 elements.
   12 19 23 27 30 34 45 56 59 61 76 79 83 85 88 90 94 97
   Perform binary search to find the elements 27, 32, 61, 97 in the array. Show the values of up, mid and low in each step.
2. Find the hash addresses of the following keys using the midsquare method, if the size of table is 1000.
   342, 213, 432, 542, 132, 763, 298
3. Find the hash addresses of the following keys using the folding method, if the size of table is 1000.
   321982432, 213432183, 343541652, 542313753
4. Find the hash addresses of the keys in exercise 3 using the boundary folding method, if the size of table is 1000.
5. How many collisions occur if the hash addresses are generated using the modulo division method, where the table size is 64.
   9893, 2341, 4312, 7893, 4531, 8731, 3184, 5421, 4955,1496
   How many collisions occur if the table size is changed to 67.
6. Insert the following keys in an array of size 17 using the modulo division method. Use linear probing to resolve collisions. 94, 37, 29, 40, 84, 88, 102, 63, 67, 120, 122
7. Insert the following keys in an array of size 17 using the modulo division method. Use quadratic probing to resolve collisions.
   94, 37, 29, 40, 84, 88, 102, 63, 67, 120, 122
8. Insert the following keys in an array of size 17 using the modulo division method. Use double hashing to resolve collisions. Take h'(k) = (key%7) + 1 as the second hash function.
   94, 37, 29, 40, 84, 88, 102, 63, 67, 120, 122
9. What is the length of the longest chain if the following keys are inserted in a table of size 11 using modulo division and separate chaining.
   1457 2134 8255 4720 6779 2709 1061 3213