# Unit-4 Hashing and File Organisation

**Searching**

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

Based on the type of search operation, these algorithms are generally classified into two categories:

1) Sequential Search
2) Interval Search

# Hashing and File Organisation

## Searching

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

Based on the type of search operation, these algorithms are generally classified into two categories:

1) Sequential Search
2) Interval Search

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

search element **12**

**Step 1:**

search element (12) is compared with first element (65)

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |
|  | 12 |  |  |  |  |  |  |  |

Both are not matching. So move to next element

**Step 2:**

search element (12) is compared with next element (20)

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |
|  |  | 12 |  |  |  |  |  |  |

Both are not matching. So move to next element

**Step 3:**

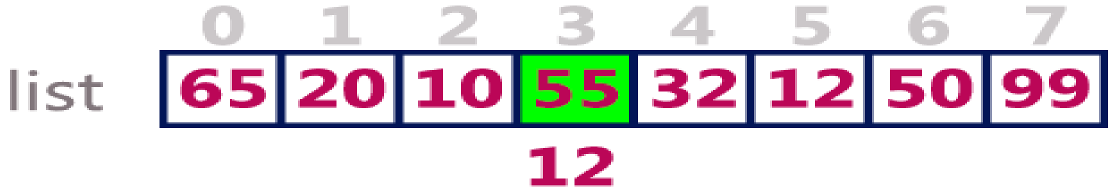
search element (12) is compared with next element (10)

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |
|  |  |  | 12 |  |  |  |  |  |

Both are not matching. So move to next element

## Step 4:

search element (12) is compared with next element (55)

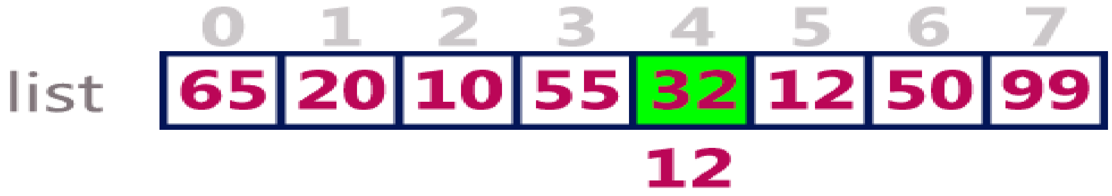


|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |
|  |  |  |  | 12 |  |  |  |  |

Both are not matching. So move to next element

## Step 5:

search element (12) is compared with next element (32)

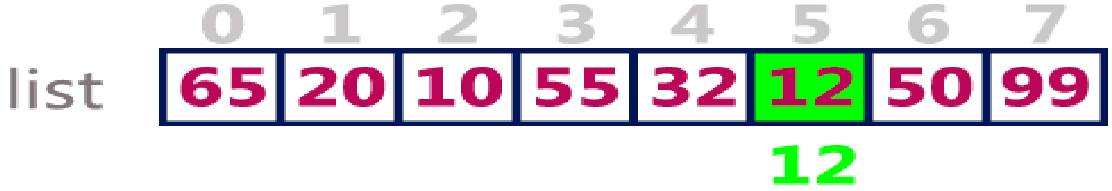


|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |
|  |  |  |  |  | 12 |  |  |  |

Both are not matching. So move to next element

## Step 6:

search element (12) is compared with next element (12)

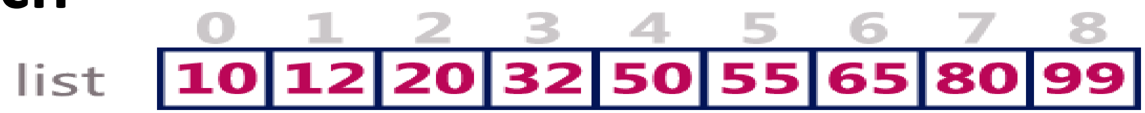


|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| list | 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |
|  |  |  |  |  |  | 12 |  |  |

Both are matching. So we stop comparing and display element found at index 5.

```
// Linear Search Logic
 for(i = 0; i < size; i++)
 {
   if(sElement == list[i])
   {
     printf("Element is found at %d index", i);
     break;
   }
 }
 if(i == size)
   printf("Given element is not found in the list!!!");
```

# Binary Search

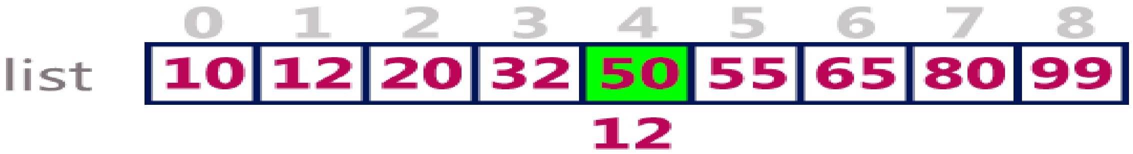


| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

search element 12

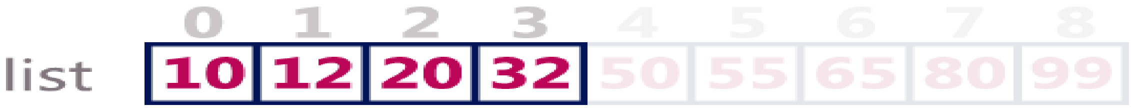
**Step 1:**

search element (12) is compared with middle element (50)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

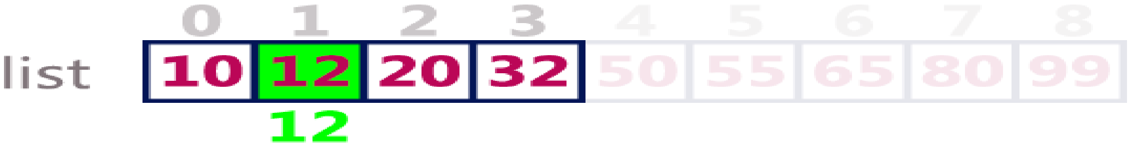
Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

**Step 2:**

search element (12) is compared with middle element (12)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| list | 10 | 12 | 20 | 32 | 50 | 55 | 65 | 80 | 99 |

**Both are matching. So the result is "Element found at index 1"**

search element 80

```
first = 0;
  last = size - 1;
  middle = (first+last)/2;
   while (first <= last) {
    if (list[middle] < sElement)
      first = middle + 1;
    else if (list[middle] == sElement) {
      printf("Element found at index %d.\n",middle);
      break;
    }
    else
      last = middle - 1;

    middle = (first + last)/2;
  }
  if (first > last)
    printf("Element Not found in the list.");
```

**Hashing** is another approach in which time required to search an element doesn't depend on the total number of elements.

Using hashing data structure, a given element is searched with **constant time complexity**.

Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

**Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key.**

Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.

In this data structure, we use a concept called **Hash table** to store data.

All the data values are inserted into the hash table based on the hash key value.

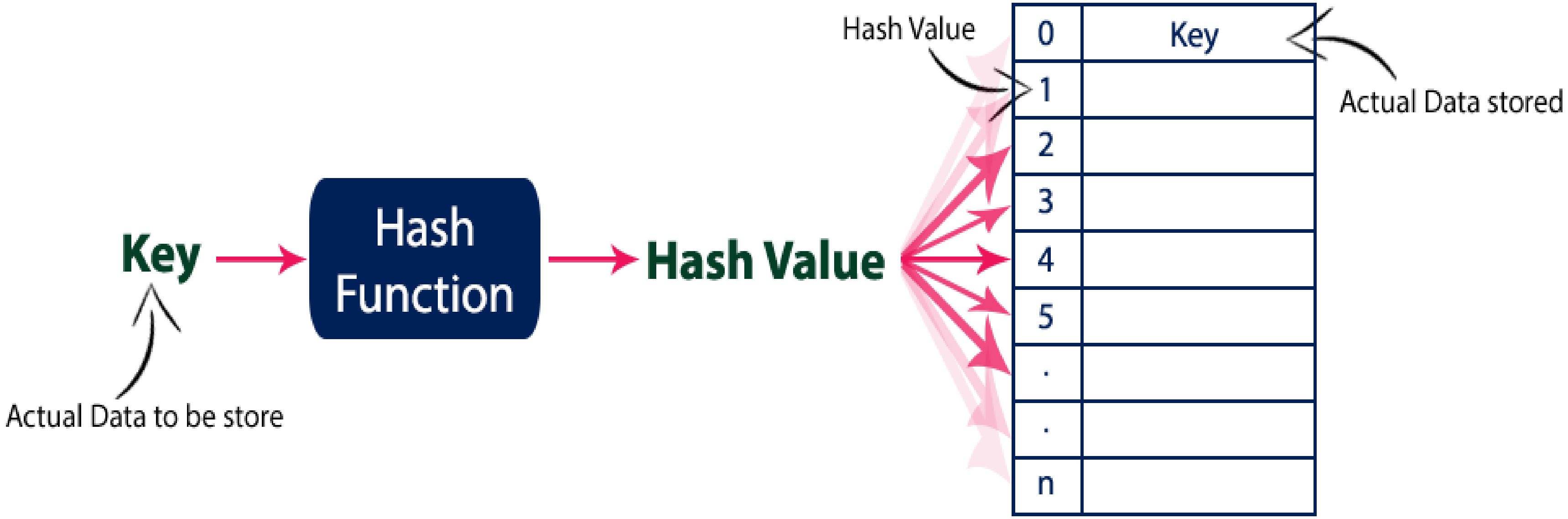
The hash key value is used to map the data with an index in the hash table.

And the hash key is generated for every data using a **hash function**.

A hash function is defined as follows...

Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.

Basic concept of hashing and hash table is shown in the following figure...



| Sr.No. | Key | Hash | Array Index |
|---|---|---|---|
| 1 | 1 | 1 % 20 = 1 | 1 |
| 2 | 2 | 2 % 20 = 2 | 2 |
| 3 | 42 | 42 % 20 = 2 | 2 |
| 4 | 4 | 4 % 20 = 4 | 4 |
| 5 | 12 | 12 % 20 = 12 | 12 |
| 6 | 14 | 14 % 20 = 14 | 14 |
| 7 | 17 | 17 % 20 = 17 | 17 |
| 8 | 13 | 13 % 20 = 13 | 13 |
| 9 | 37 | 37 % 20 = 17 | 17 |

***Linear Probing:*** In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let **hash(x)** be the slot index computed using hash function and **S** be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
..................................................
..................................................

| Sr.No. | Key | Hash | Array Index | After Linear Probing, Array Index |
|---|---|---|---|---|
| 1 | 1 | 1 % 20 = 1 | 1 | 1 |
| 2 | 2 | 2 % 20 = 2 | 2 | 2 |
| 3 | 42 | 42 % 20 = 2 | 2 | 3 |
| 4 | 4 | 4 % 20 = 4 | 4 | 4 |
| 5 | 12 | 12 % 20 = 12 | 12 | 12 |
| 6 | 14 | 14 % 20 = 14 | 14 | 14 |
| 7 | 17 | 17 % 20 = 17 | 17 | 17 |
| 8 | 13 | 13 % 20 = 13 | 13 | 13 |
| 9 | 37 | 37 % 20 = 17 | 17 | 18 |

# Types of Hash Functions

1. Division method

In this the hash function is dependent upon the remainder of a division. For example:-if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.

Then:

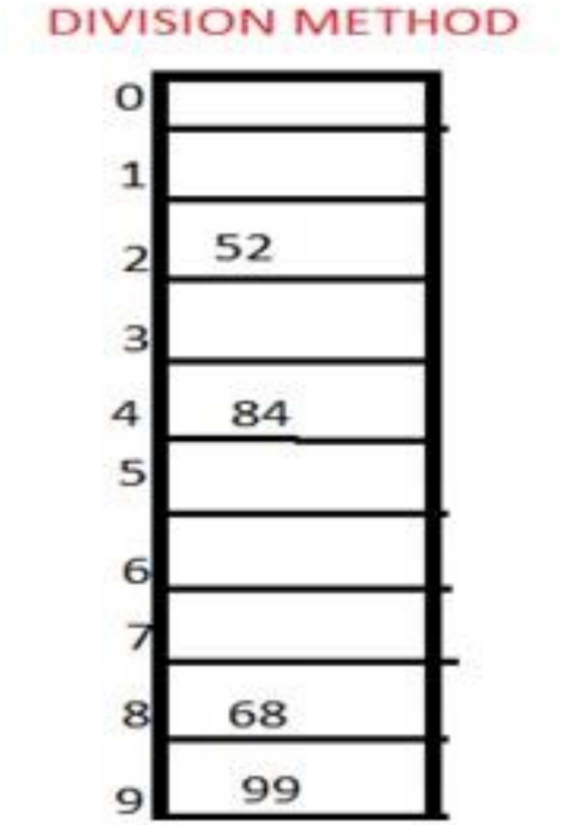
h(key)=record% table size.

2=52%10

8=68%10

9=99%10

4=84%10



DIVISION METHOD

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 52 |
| 3 | |
| 4 | 84 |
| 5 | |
| 6 | |
| 7 | |
| 8 | 68 |
| 9 | 99 |

## 1. Mid square method

In this method firstly key is squared and then mid part of the result is taken as the index.

For example: consider that if we want to place a record of 3101 and the size of table is 1000. So 3101*3101=9616201 i.e. h (3101) = 162 (middle 3 digit)

## 1. Digit folding method

In this method the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key. For example: consider a record of 12465512 then it will be divided into parts i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

H(key)=124+655+12
=791

Ex. (SSN) 123-45-6789

1. Divide into 3 parts: **123, 456 and 789** .

2. Add them.

   **$123+456+789=1368$**

3. $h(k)=k \bmod M$

   where $M = 1000$

   $h(1368) = 1368 \bmod 1000$

   $= \mathbf{368}$

1. Divide into five parts: **12, 34, 56, 78 and 9** .

2. Add them.

   **$12 + 34 + 56 + 78 + 9 = 189$**

3. $h(k)=k \bmod M$

   where $M = 1000$

   $h(189) = 189 \bmod 1000$

   $= \mathbf{189}$

# Characteristics of good hashing function

- ✓ The hash function should generate different hash values for the similar string.
- ✓ The hash function is easy to understand and simple to compute.
- ✓ The hash function should produce the keys which will get distributed, uniformly over an array.
- ✓ A number of collisions should be less while placing the data in the hash table.
- ✓ The hash function is a perfect hash function when it uses all the input data.

# Collision

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

## Collision resolution technique

If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below.

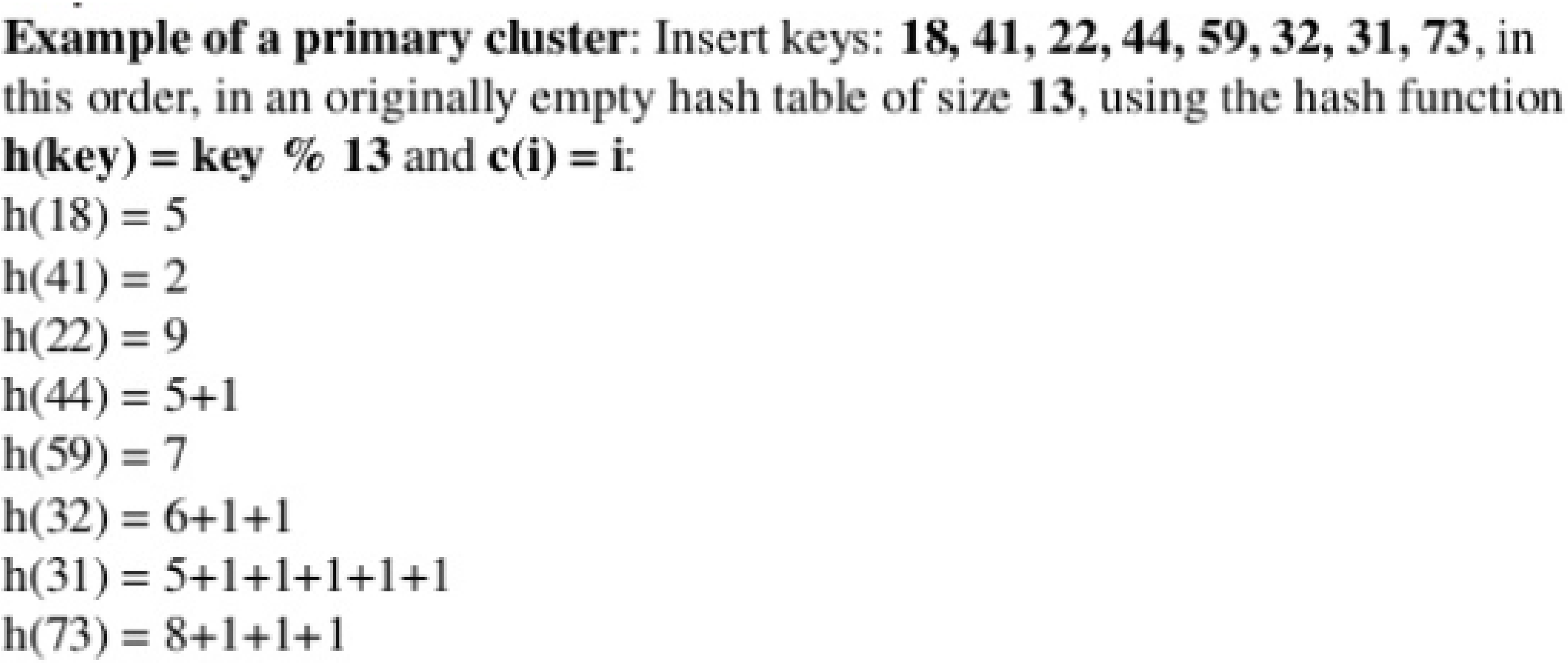
# Linear probing

It is very easy and simple method to resolve or to handle the collision. In this collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table size. Consider that following keys are to be inserted that are 56,64,36,71.

$$H(x,i) = (H(x) + i)(\mathrm{mod}\ M)$$

**Clustering:** The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.



**Example of a primary cluster**: Insert keys: **18, 41, 22, 44, 59, 32, 31, 73**, in this order, in an originally empty hash table of size **13**, using the hash function **h(key) = key % 13** and **c(i) = i**:

h(18) = 5

h(41) = 2

h(22) = 9

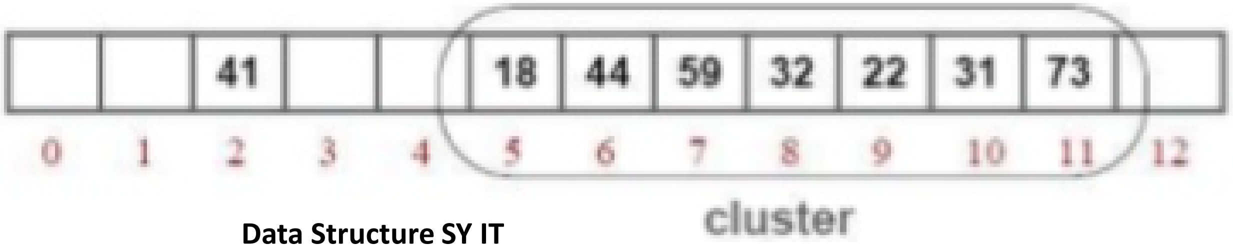
h(44) = 5+1

h(59) = 7

h(32) = 6+1+1

h(31) = 5+1+1+1+1+1

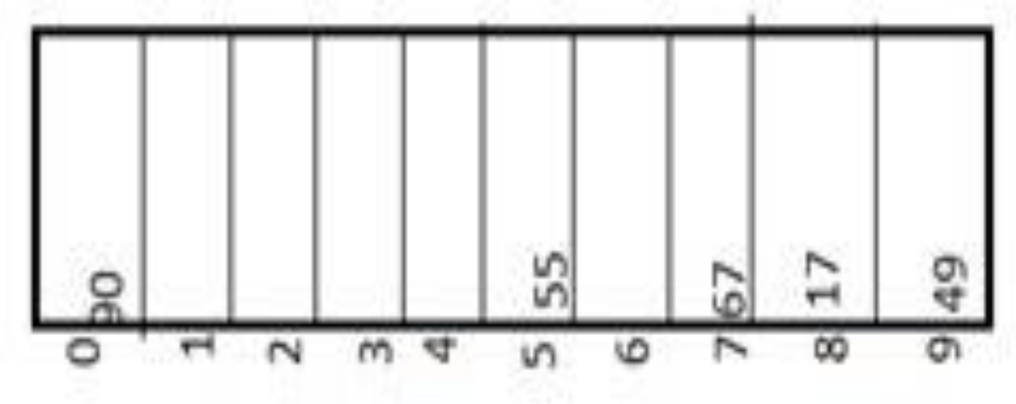
h(73) = 8+1+1+1



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  | 41 |  |  | 18 | 44 | 59 | 32 | 22 | 31 | 73 |  |

# Quadratic probing

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the H(key)=(H(key)+x*x)%table size. Let us consider we have to insert following elements that are:-67, 90,55,17,49.



- Example:
  - Table Size is 11 (0..10)
  - Hash Function: $h(x) = x \bmod 11$
  - Insert keys:
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 → $2+1^2=3$
    - 25 mod 11 = 3 → $3+1^2=4$
    - 24 mod 11 = 2 → $2+1^2$, $2+2^2=6$
    - 10 mod 11 = 10
    - 9 mod 11 = 9 → $9+1^2$, $9+2^2 \bmod 11$,

      $9+3^2 \bmod 11 = 7$

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 2 |
| 3 | 13 |
| 4 | 25 |
| 5 | |
| 6 | **24** |
| 7 | **9** |
| 8 | 30 |
| 9 | 20 |
| 10 | 10 |

# Double hashing

It is a technique in which two hash function are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

It must never evaluate to zero.
Must sure about the buckets, that they are probed.
The hash functions for this technique are:

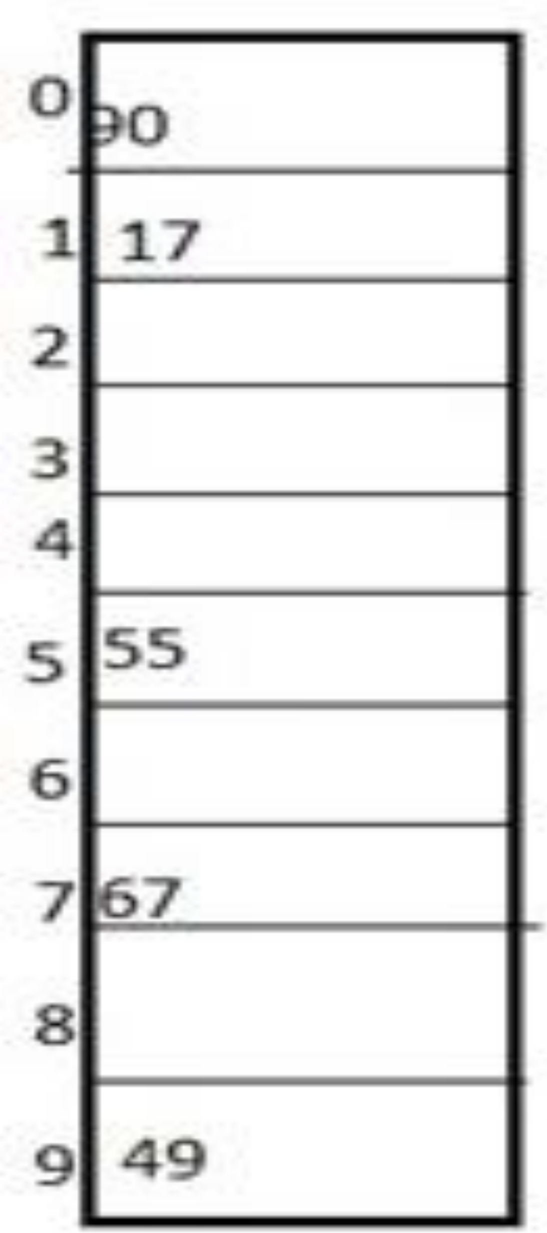
H1(key)=key % table size
H2(key)=P-(key mod P)

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

**Example:** Let us consider we have to insert 67, 90,55,17,49.

In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 again the bucket is full and in this case we have to use the second hash function which is H2(key)=P-(key mode P) here p is a prime number which should be taken smaller than the hash table so value of p will be the 7.

i.e. H2(17)=7-(17%7)=7-3=4 that means we have to take 4 jumps for placing the 17. Therefore 17 will be placed at index 1.



| | |
|---|---|
| 0 | 90 |
| 1 | 17 |
| 2 | |
| 3 | |
| 4 | |
| 5 | 55 |
| 6 | |
| 7 | 67 |
| 8 | |
| 9 | 49 |

```
void insert()
{
 int key,index,i,hkey;
 printf("\nEnter a value to insert into hash
table\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 for(i=0;i<TABLE_SIZE;i++)
   {
    index=(hkey+i)%TABLE_SIZE;
    if(h[index] == NULL)
    {
       h[index]=key;
        break;
    }    }
   if(i == TABLE_SIZE)
    printf("\nElement cannot be inserted\n");
}
```

67,
90,55,17,49

67%5 =2

90%5 =0

55%5=0

56%5=1

17%5 =3

49%5 =4

| Index | hashtable |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |

```
void search()
{ int key,index,i,hkey;
 printf("\nenter search element\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 for(i=0;i<TABLE_SIZE; i++) {
   index=(hkey+i)%TABLE_SIZE;
   if(h[index]==key)    {
     printf("Searching value is found at index %d",index);
     break;    }  }
    if(i == TABLE_SIZE)
   printf("\nSearching value is not found\n"); }
```

```
void display()
{
  int i;
  printf("\nElements in the hash table are \n");
  for(i=0;i< TABLE_SIZE; i++)
  printf("\n(%d) %d", i, h[i]);
}
```

# Linear probing with Chaining and without replacement

In collision handling method **chaining is a concept which introduces an additional field with data i.e. chain.** A separate chain table is maintained for colliding data. When collision occurs, we store the second colliding data by linear probing method. The address of this colliding data can be stored with the first colliding element in the chain table, without replacement.

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 33

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 1 | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | 4 | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 33

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 1 | 2 |
| 2 | 71 | -1 |
| 3 | | -1 |
| 4 | 4 | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 33

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 1 | 2 |
| 2 | 71 | 3 |
| 3 | 11 | -1 |
| 4 | 4 | 5 |
| 5 | 64 | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | 89 | -1 |
| 9 | | -1 |

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 33

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 1 | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | 4 | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 33

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 1 | 2 |
| 2 | 71 | -1 |
| 3 | | -1 |
| 4 | 4 | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 22

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 1 | 2 |
| 2 | 71 | 3 |
| 3 | 11 | -1 |
| 4 | 4 | 5 |
| 5 | 64 | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | 89 | -1 |
| 9 | | -1 |

Thus any element which gives hash key as 1 will be stored by linear probing at empty location but a chain is maintained so that **traversing the hash table will be efficient.**

# Linear probing with Chaining and with replacement

Excessive Collision in linear probing could become a major problem . One way of dealing with collision is by means of chaining. All records mapped to same location are stored in a chain.

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 33

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 1 | 6 |
| 2 | 22 | -1 |
| 3 | 33 | -1 |
| 4 | 4 | 5 |
| 5 | 64 | -1 |
| 6 | 11 | 7 |
| 7 | 71 | -1 |
| 8 | | -1 |
| 9 | 89 | -1 |

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 33

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 1 | 2 |
| 2 | 71 | -1 |
| 3 | | -1 |
| 4 | 4 | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

**For example**, consider elements: 0, 1, 4, 71, 64, 89, 11, 22

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

| Index | Key | Chain |
|---|---|---|
| 0 | | -1 |
| 1 | | -1 |
| 2 | | -1 |
| 3 | | -1 |
| 4 | | -1 |
| 5 | | -1 |
| 6 | | -1 |
| 7 | | -1 |
| 8 | | -1 |
| 9 | | -1 |

Consider elements: 10, 51,63, 61, 73, 53, 58, 13, 55, 6
And store these element using linear probing with chaining without replacement and with replacement

Consider elements: 10, 51,63, 61, 73, 53, 58, 13, 55, 6

And store these element using lin

without replacement and with rep

10%10=0

51%10 =1

63%10 =3

61%10 =1

73%10=3

53%10=3

58%10=8

13%10=3

55%10=5

6%10=6

| Index | Key | Chain |
|---|---|---|
| 0 | 10 | -1 |
| 1 | 51 | 2 |
| 2 | 61 | -1 |
| 3 | 63 | 4 |
| 4 | 73 | 7 |
| 5 | 55 | -1 |
| 6 | 6 | -1 |
| 7 | 53 | 9 |
| 8 | 58 | -1 |
| 9 | 13 | -1 |

# File organisation

- A file is a collection of records where each record consists of one or more fields.
- A method of storing records in a file.
- Primary objective of file organisation is to provide a means for record retrieval and update.

# Commonly used file organisations

- Sequential file
- Relative file
- Direct file
- Indexed sequential file
- Index file

# Sequential File

- Most common type of file
- A fixed format is used for record
- All records are of same length
- Position of each filed in record and length of filed is fixed
- Records are physically ordered on the value of one of the filed called ordering filed

# Example

| Name | Roll NO | Year | Marks |
|---|---|---|---|
| aaa | 1 | 1 | 60.48 |
| bbb | 2 | 2 | 78.4 |
| ccc | 4 | 1 | 54.6 |

# Operations

- Creation
- Reading
- Insertion
- Deletion
- Updation
- Searching

- “r” – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.

- “w” – Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

- “a” – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

- “r+” – Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.

- “w+” – Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist a new file is created. Returns NULL, if unable to open file.

- “a+”

# Creation

```
void create()
{
FILE *fp;
If(!(fp=fopen(“abc.txt”, “r”);
fp=foepn(“abc.txt”, “w”);
close(fp);
}
```

```
int main()
{
  int num;
  FILE *fptr;

  fptr = fopen("C:\\program.txt","w");

  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }
  printf("Enter num: ");
  scanf("%d",&num);

  fprintf(fptr,"%d",num);
  fclose(fptr);
  return 0;
}
```

```c
#include<stdio.h>
void main( )
{
FILE *fp ;
char ch ;
fp = fopen("file_handle.c","r") ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf("%c",ch) ;
}
fclose (fp ) ;
}
```

```
#include <stdio.h>
main(){
   FILE *fp;
   fp = fopen("file.txt", "w");//opening file
   fprintf(fp, "Hello file by fprintf...\n");//writing data into file

   fclose(fp);//closing file
}
```

```
#include <stdio.h>
main(){
   FILE *fp;
   char s[255]
   fp = fopen("file.txt", "r");
   while(fscanf(fp, "%s", buff)!=EOF){
   printf("%s ", s );
   }
   fclose(fp);
}
```

```
int main()
{
   int num;
   FILE *fptr;

   if ((fptr = fopen("C:\\program.txt","r")) == NULL){
       printf("Error! opening file");

       exit(1);
   }

   fscanf(fptr,"%d", &num);

   printf("Value of n=%d", num);
   fclose(fptr);

   return 0;
}
```