Unit 7 (Searching)

1. Search technique
2. Sequential, Binary and Tree search
3. General search tree
4. Hashing
    - Hash function and hash tables
    - Collision resolution technique

Search technique

- Searching is the process of finding an element in a given list. In this process, we check items that are available in the given list or not.
- Internal Search: In this search, searching is performed on main or primary memory.
- External Search: In this search, searching is performed in secondary memory
- Three types of searching techniques,
    - Linear or sequential search
    - Binary search
    - Hashing

Sequential Search:

Linear Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful. Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

Algorithm:

```
LinearSearch(A, n,key)
{
        for(i=0;i<n;i++)
        {
                if(A[i] == key)       return i;
        }
```

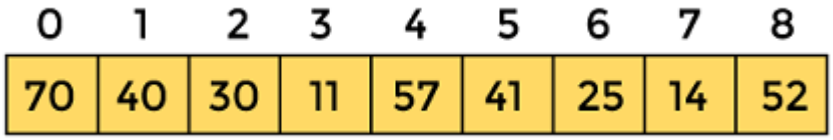
```
        return -1;//-1 indicates unsuccessful search
    }
```

Working of Linear search

Now, let's see the working of the linear search Algorithm.

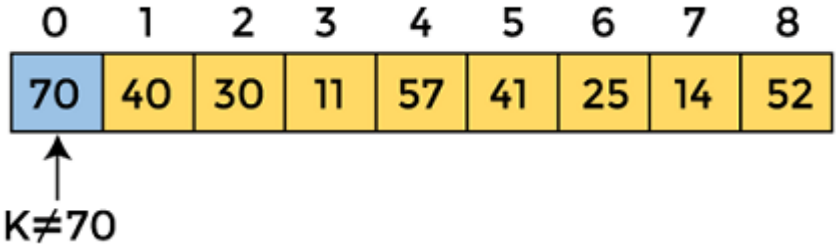
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

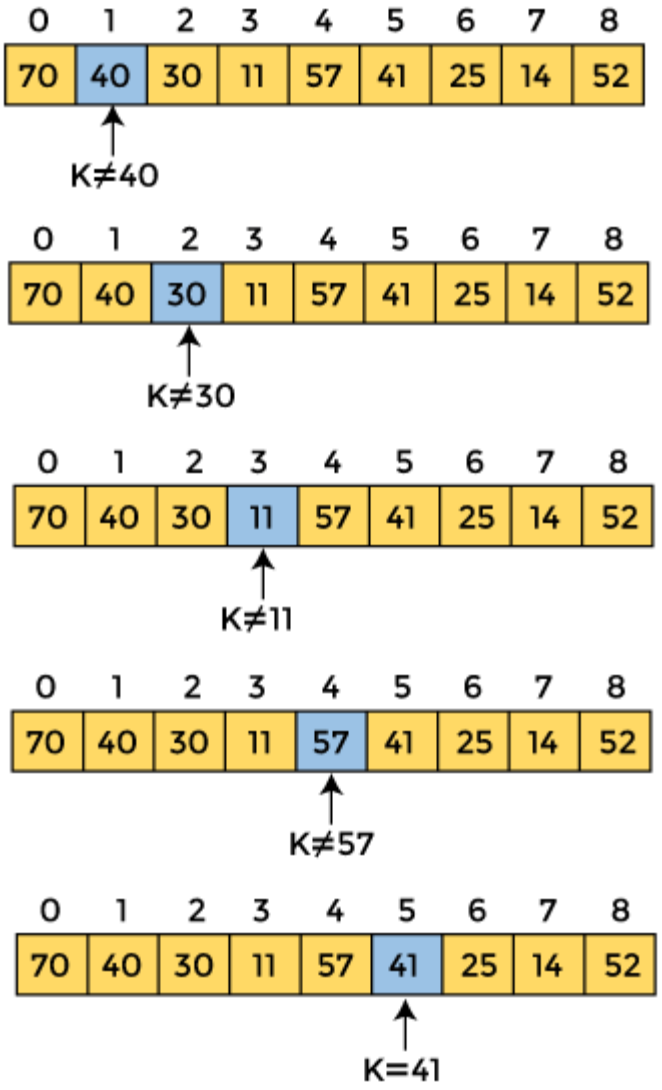


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.

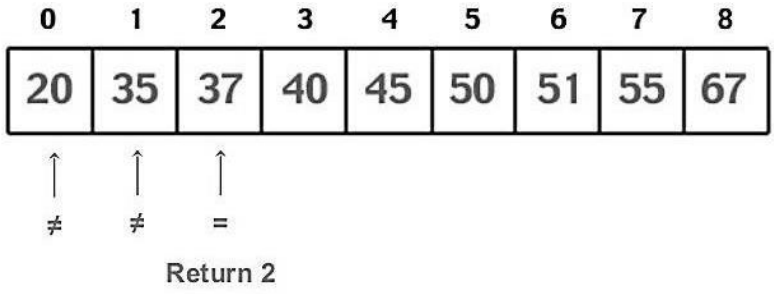


The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So algorithm will return the index of the element matched.

Example 2: find key=37



Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

- **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
- **Average Case Complexity** - The average case time complexity of linear search is **O(n).**
- **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**

The time complexity of linear search is **O(n)** because every element in the array is compared only once.

2. Space Complexity

Space Complexity	**O(1)**

- The space complexity of linear search is O(1).

Implementation of Linear Search

Now, let's see the programs of linear search in different programming languages.

Program: Write a program to implement linear search in C language.

```
#include <stdio.h>

int linearSearch(int a[], int n, int val) {

  // Going through array sequencially
```

```
  for (int i = 0; i < n; i++)
    {
        if (a[i] == val)
        return i+1;
    }
  return -1;
}
int main() {
  int a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array
  int val = 41; // value to be searched
  int n = sizeof(a) / sizeof(a[0]); // size of array
  int res = linearSearch(a, n, val); // Store result
  printf("The elements of the array are - ");
  for (int i = 0; i < n; i++)
  printf("%d ", a[i]);
  printf("\nElement to be searched is - %d", val);
  if (res == -1)
  printf("\nElement is not present in the array");
  else
  printf("\nElement is present at %d position of array", res);
```

```
    return 0;

}
```

Binary Search:

In this article, we will discuss the Binary Search Algorithm. Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element. Otherwise, the search is called unsuccessful. Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

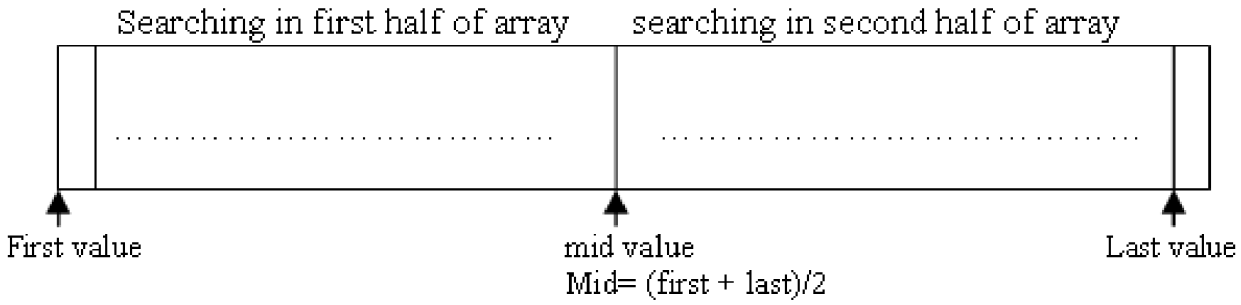
Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

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

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

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

NOTE: Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.



Compiled by: Er. Sandesh S Poudel

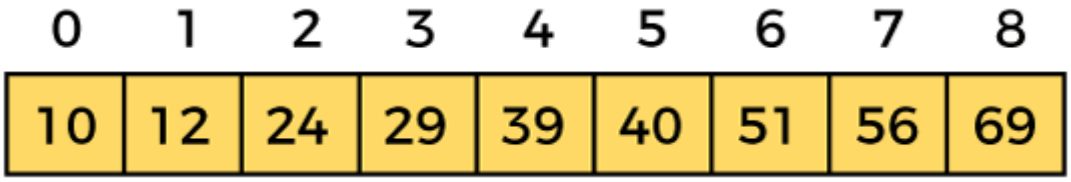
Algorithm

Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search

```
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
            set pos = mid
            print pos
            go to step 6
        else if a[mid] > val
            set end = mid - 1
        else
            set beg = mid + 1
        [end of if]
        [end of loop]
Step 5: if pos = -1
        print "value is not present in the array"
        [end of if]
Step 6: exit
```

Working of Binary search

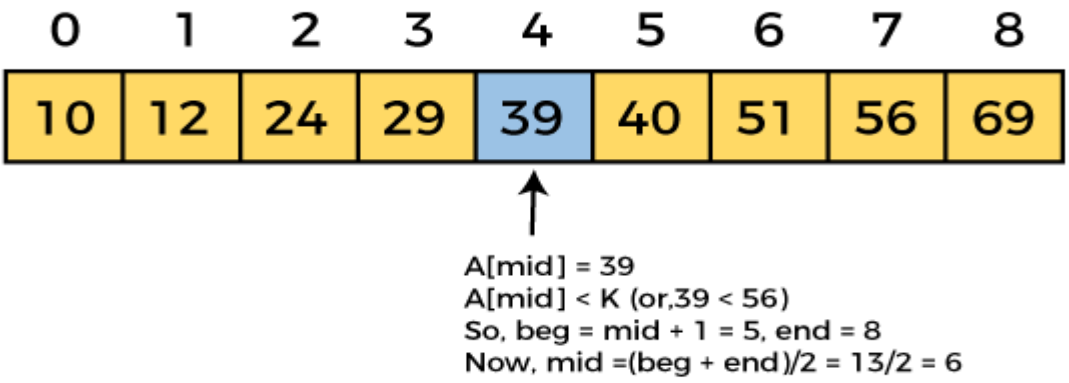
Let the elements of array are -

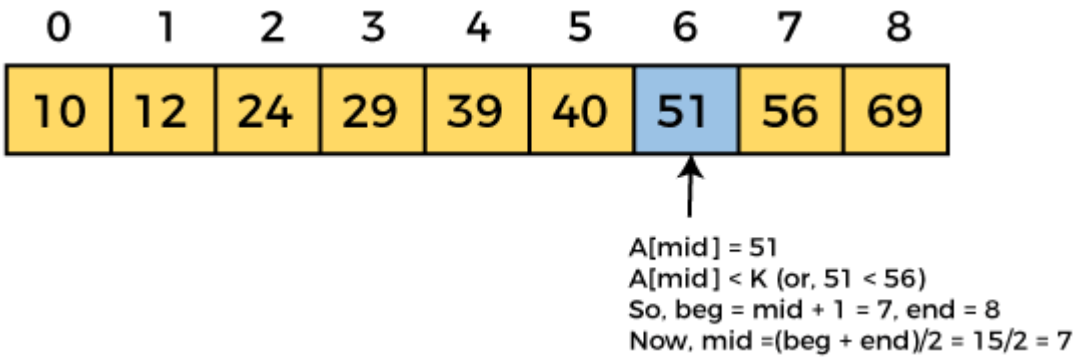


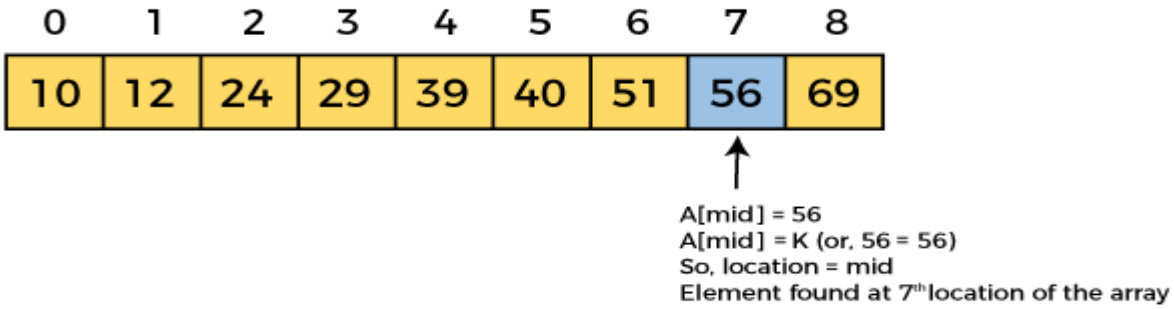
Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array - **mid = (beg + end)/2**

So, in the given array - **beg** = 0, **end** = 8, **mid** = (0 + 8)/2 = 4. So, 4 is the mid of the array.







Now, the element to search is found. So algorithm will return the index of the element matched.

Example 2:

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

For key = 2

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

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

For key = 103

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

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

For key = 67

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

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

Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

- **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**
- **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
- **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

2. Space Complexity

Space Complexity	**O(1)**

- The space complexity of binary search is O(1).

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

T(n) = T(n/2) + O(1) = O(logn) (verify).

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

Implementation of Binary Search

Now, let's see the programs of Binary search in different programming languages.

Program: Write a program to implement Binary search in C language.

```
#include <stdio.h>
int binarySearch(int a[], int beg, int end, int val)
{
   int mid;
   if(end >= beg)
   {       mid = (beg + end)/2;
/* if the item to be searched is present at middle */
      if(a[mid] == val)
      {
         return mid+1;
      }
         /* if the item to be searched is smaller than middle, then it can only be in left subarray */
      else if(a[mid] < val)
      {
         return binarySearch(a, mid+1, end, val);
      }
         /* if the item to be searched is greater than middle, then it can only be in right subarray */
      else
```

```
        {
            return binarySearch(a, beg, mid-1, val);
        }
    }
    return -1;
}
int main() {
  int a[] = {11, 14, 25, 30, 40, 41, 52, 57, 70}; // given array
  int val = 40; // value to be searched
  int n = sizeof(a) / sizeof(a[0]); // size of array
  int res = binarySearch(a, 0, n-1, val); // Store result
  printf("The elements of the array are - ");
  for (int i = 0; i < n; i++)
  printf("%d ", a[i]);
  printf("\nElement to be searched is - %d", val);
  if (res == -1)
  printf("\nElement is not present in the array");
  else
  printf("\nElement is present at %d position of array", res);
  return 0;
}
```

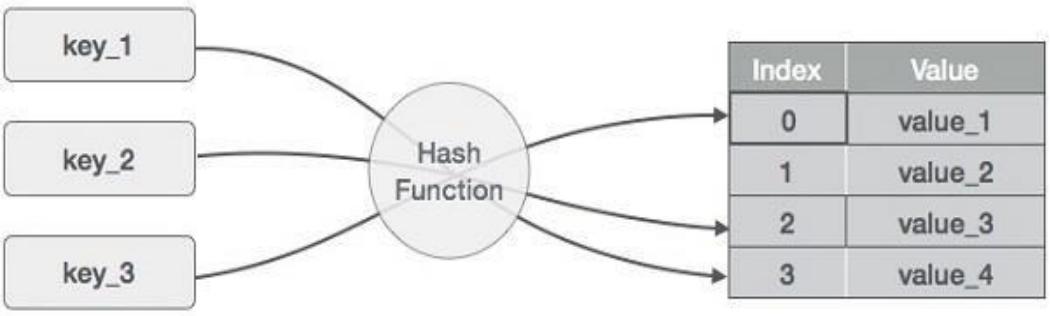
Tree Search: Same as Binary Search Tree.

Hashing:

Hashing is the transformation of a string of character into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shortest hashed key than to find it using the original value. It is also used in many encryption algorithms. A hash code is generated by using a key, which is a unique value. Hashing is a technique in which given key field value is converted into the address of storage location of the record by applying the same operation on it.

A function that transforms a key into a table index is called a hash function. A common hash function is **h(x)=x mod SIZE** if key=27 and SIZE=10 then hash address=27%10=7

The advantage of hashing is that allows the execution time of basic operation to remain constant even for the larger side.



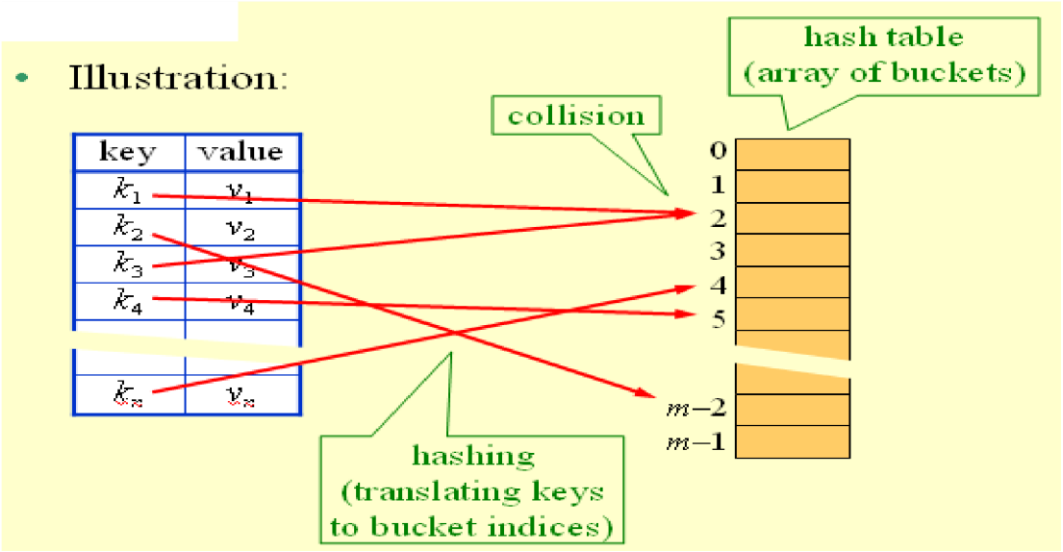
Why we need Hashing?

Suppose we have 50 employees, and we have to give 4 digit key to each employee (as for security), and we want after entering a key, direct user map to a particular position where data is stored.

If we give the location number according to 4 digits, we will have to reserve 0000 to 9999 addresses because anybody can use anyone as a key. There is a lot of wastage.

In order to solve this problem, we use hashing which will produce a smaller value of the index of the hash table corresponding to the key of the user.

Hash-table principles:



Hash function and hash tables

- Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
- Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.
- Let us take a hash function h(x)=x % 10

  If we want to store keys 8,3,6,4,10,81,72,25

10	0
81	1
72	
3	2
4	3
25	4
6	5
	6
8	7

8

9

Assume we have a set of integers 54, 26, 93, 17, 77, 31. Our first hash function required to be as "remainder method" simply takes the item and divide it by table size, returning remainder as its hash value i.e.

h item = item % (size of table)

Let us say the size of table = 11, then

54 % 11 = 10 26 % 11 = 4 93 % 11 = 5

17 % 11 = 6 77 % 11 = 0 31 % 11 = 9

ITEM	HASH VALUE
54	10

26	4
93	5
17	6
77	0
31	9

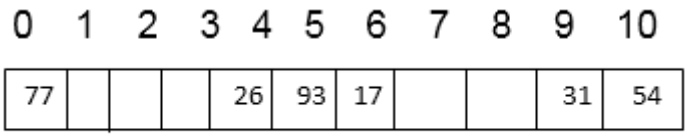


Fig: Hash Table

Now when we need to search any element, we just need to divide it by the table size, and we get the hash value. So we get the O (1) search time.

Now taking one more element 44 when we apply the hash function on 44, we get (44 % 11 = 0), But 0 hash value already has an element 77. This Problem is called as Collision.

Application of Hash Tables:

Some application of Hash Tables are:

1. **Database System:** Specifically, those that are required efficient random access. Usually, database systems try to develop between two types of access methods: sequential and random. Hash Table is an integral part of efficient random access because they provide a way to locate data in a constant amount of time.
2. **Symbol Tables:** The tables utilized by compilers to maintain data about symbols from a program. Compilers access information about symbols frequently. Therefore, it is essential that symbol tables be implemented very efficiently.
3. **Data Dictionaries:** Data Structure that supports adding, deleting, and searching for data. Although the operation of hash tables and a data dictionary are similar, other Data Structures may be used to implement data dictionaries.
4. **Associative Arrays:** Associative Arrays consist of data arranged so that n^{th} elements of one array correspond to the n^{th} element of another. Associative Arrays are helpful for indexing a logical grouping of data by several key fields.

Hash Function

Hash Function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. There is no need to "reverse engineer" the hash function by analyzing the hashed values.

Characteristics of Good Hash Function:

1. The hash value is fully determined by the data being hashed.
2. The hash Function uses all the input data.
3. The hash function "uniformly" distributes the data across the entire set of possible hash values.
4. The hash function generates complicated hash values for similar strings.

Some Popular Hash Function is:

1. Division Method:

Choose a number m smaller than the number of n of keys in k (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently a minimum number of collisions). The hash function is:

$$h(k) = k \bmod m$$

$$h(k) = k \bmod m + 1$$

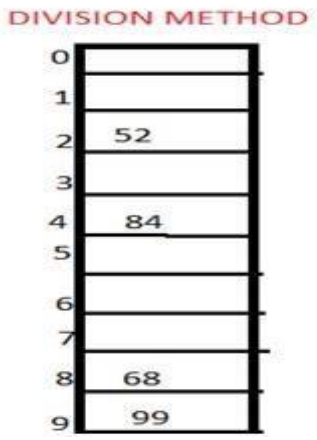
- For example:-if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.
- Then: h(key)= record% table size.

52%10 =2

68%10 =8

99%10 =9

84%10 =4



2. Multiplication Method:

The multiplication method for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we increase this value by m and take the floor of the result. The hash function is:

$$h(k) = \lfloor m(k A \bmod 1) \rfloor$$

Where "k A mod 1" means the fractional part of k A, that is, k A -⌊k A⌋.

3. Mid Square Method:

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

H (k) = L

Where L is obtained by deleting digits from both ends of k^2. We emphasize that the same position of k^2 must be used for all of the keys.

4. Folding Method:

The key k is partitioned into a number of parts k_1, k_2.... k_n where each part except possibly the last, has the same number of digits as the required address. Then the parts are added together, ignoring the last carry.

$$H(k) = k^1 + k^2 + \ldots\ldots + k^n$$

Example: Company has 68 employees, and each is assigned a unique four- digit employee number. Suppose L consist of 2- digit addresses: 00, 01, and 02....99. We apply the above hash functions to each of the following employee numbers:

3205, 7148, 2345

(a) Division Method: Choose a Prime number m close to 99, such as m =97, Then

H (3205) = 4, H (7148) = 67, H (2345) = 17.

That is dividing 3205 by 17 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

(b) Mid-Square Method:

k = 3205 7148 2345

k^2= 10272025 51093904 5499025

h (k) = 72 93 99

Observe that fourth & fifth digits, counting from right are chosen for hash address.

(c) Folding Method: Divide the key k into 2 parts and adding yields the following hash address:

H (3250) = 32 + 50 = 82 H (1748) = 17 + 48 = 55

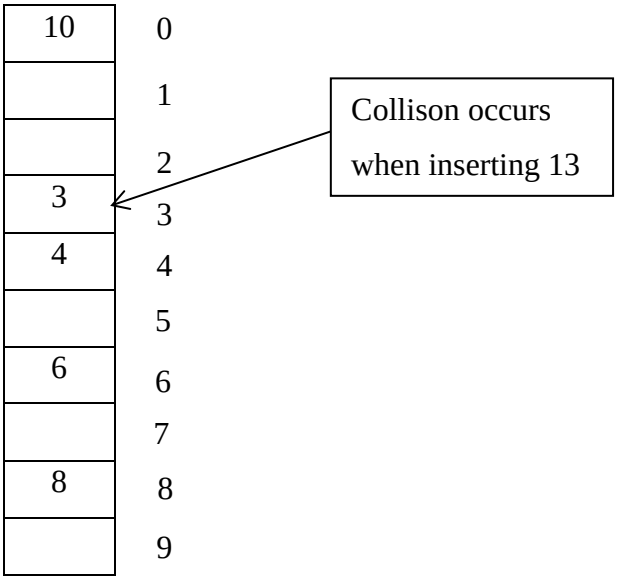
H (2345) = 23 + 45 = 68

Collision Resolving Techniques

- Two keys mapping to the same location in the hash table is called "Collision"
- Collisions can be reduced with a selection of a good hash function.

Let us consider the following key 8,3,13,6,4,10 and hash size of 10.

h(x)=x % 10



If two or more than two records trying to insert in a single index of a hash table then such a situation is called hash collision.

Collision: According to the Hash Function, two or more item would need in the same slot. This is said to be called as Collision.

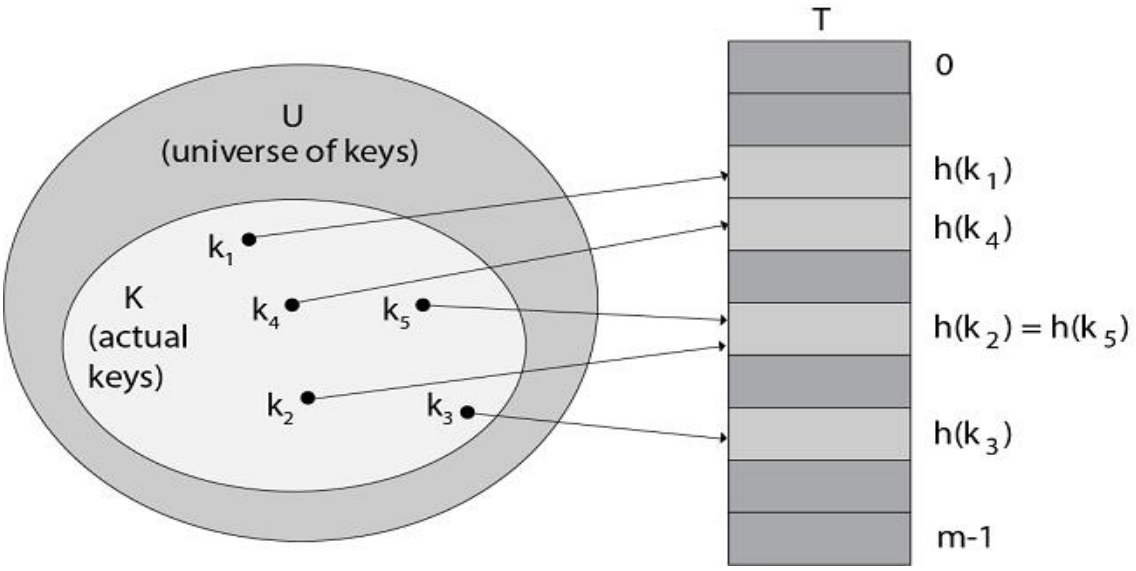


Figure: using a hash function h to map keys to hash-table slots. Because keys K2 and k5 map to the same slot, they collide.

Collision Resolution Technique

There are two main methods used to implement hashing:

1. Hashing with open addressing
2. Hashing with Chaining

Hashing with Open Addressing

In Open Addressing, all elements are stored in hash table itself. That is, each table entry consists of a component of the dynamic set or NIL. When searching for an item, we consistently examine table slots until either we find the desired object or we have determined that the element is not in the table. Thus, in open addressing, the load factor α can never exceed 1.

The advantage of open addressing is that it avoids Pointer. In this, we compute the sequence of slots to be examined. The extra memory freed by not sharing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collision and faster retrieval. The process of examining the location in the hash table is called Probing.

Open Addressing Techniques

Three techniques are commonly used to compute the probe sequence required for open addressing:

1. Linear Probing.
2. Quadratic Probing.
3. Double Hashing.

Linear probing:

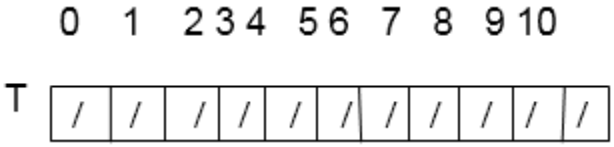
A hash-table in which a collision is resolved by putting the item in the next empty place within the occupied array space. It starts with a location where the collision occurred and does a sequential search through a hash table for the desired empty location. Hence this method searches in straight line, and it is therefore called linear probing. The hash function of linear probing is:

h (k, i) = (h' (k) + i) mod m

Where 'm' is the size of hash table and h' (k) = k mod m. for i=0, 1....m-1.

Example: Consider inserting the keys 24, 36, 58,65,62,86 into a hash table of size m=11 using linear probing, consider the primary hash function is h' (k) = k mod m.

Solution: Initial state of hash table



Insert 24. We know h (k, i) = [h' (k) + i] mod m

Now h (24, 0) = [24 mod 11 + 0] mod 11 = (2+0) mod 11 = 2 mod 11 = 2

Since T [2] is free, insert key 24 at this place.

Insert 36. Now h (36, 0) = [36 mod 11 + 0] mod 11 = [3+0] mod 11 = 3

Since T [3] is free, insert key 36 at this place.

Insert 58. Now h (58, 0) = [58 mod 11 +0] mod 11 = [3+0] mod 11 =3

Since T [3] is not free, so the next sequence is

h (58, 1) = [58 mod 11 +1] mod 11 = [3+1] mod 11= 4 mod 11=4

T [4] is free; Insert key 58 at this place.

Insert 65. Now h (65, 0) = [65 mod 11 +0] mod 11 = (10 +0) mod 11= 10

T [10] is free. Insert key 65 at this place.

Insert 62. Now h (62, 0) = [62 mod 11 +0] mod 11 = [7 + 0] mod 11 = 7

T [7] is free. Insert key 62 at this place.

Insert 86. Now h (86, 0) = [86 mod 11 + 0] mod 11 = [9 + 0] mod 11 = 9

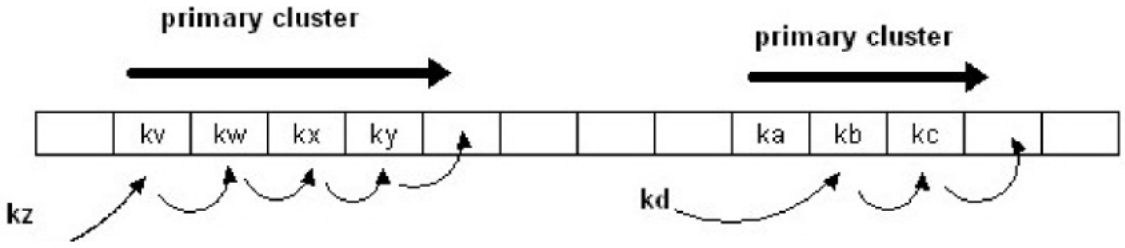
T [9] is free. Insert key 86 at this place.

Thus,

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	24	36	58	/	/	62	/	86	65

Disadvantage:

Clustering problem:



Example:
Insert keys {89, 18, 49, 58, 69} with the hash function h(x)=x mod 10 using linear probing.

solution

when **x=89:**

h(89)=89%10=9

insert key 89 in hash-table in location 9

when **x=18:**

h(18)=18%10=8

insert key 18 in hash-table in location 8

when **x=49:**

h(49)=49%10=9 (Collision occur) so insert key 49 in hash-table in next possible vacant location of 9 is 0

when **x=58:**

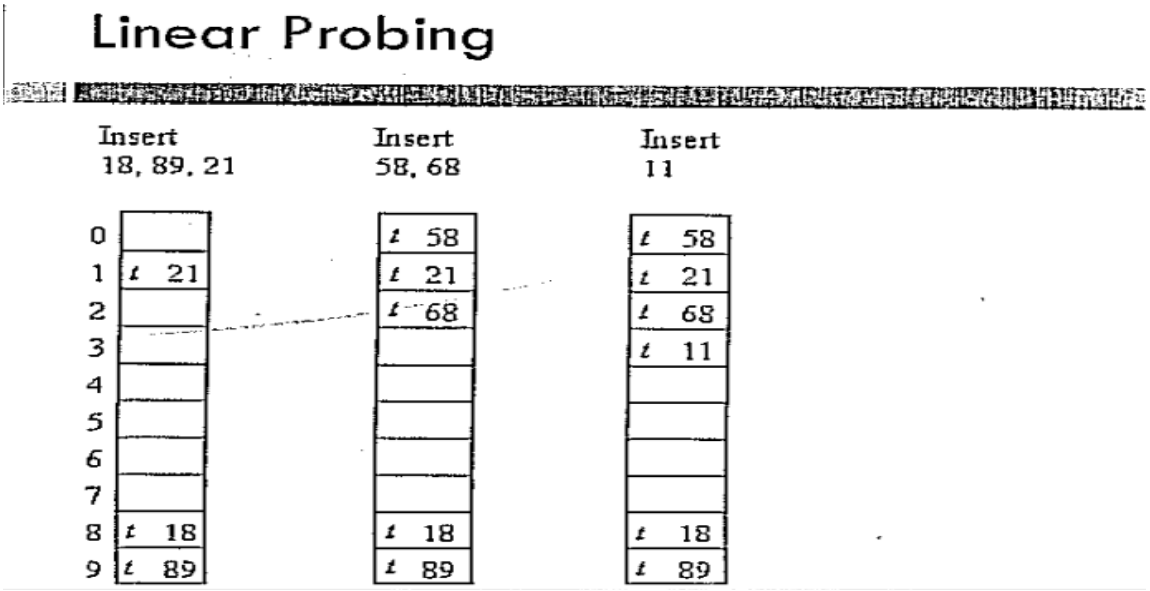
h(58)=58%10=8 (Collision occur) insert key 58 in hash-table in next possible vacant location of 8 is 1 (since 9, 0 already contains values).

when **x=69:**

h(89)=69%10=9 (Collision occur) insert key 69 in hash-table in next possible vacant location of 9 is 2 (since 0, 1 already contains values).

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Fig Hash-table for above keys using linear probing



Quadratic Probing:

Quadratic probing is a collision resolution method that eliminates the primary clustering problem take place in a linear probing. The hash function of quadratic probing is:

h (k, i) = (h' (k) + i^2) mod m

Where 'm' is the size of hash table and h' (k) = k mod m. for i=0, 1....m-1.

When collision occur then the quadratic probing works as follows:

(Hash value + 1^2)% table size

if there is again collision occur then there exist rehashing.

(hash value + 2^2)%table size

if there is again collision occur then there exist rehashing.

(hash value = 3^2)% table size

in general in ith collision

hi(x)=(hash value +i^2)%size

Example:

Insert keys {89, 18, 49, 58, 69} with the hash-table size 10 using quadratic probing.

solution:

when x=89:

h(89)=89%10=9

insert key 89 in hash-table in location 9

when x=18:

h(18)=18%10=8

insert key 18 in hash-table in location 8

when x=49:

h(49)=49%10=9 (Collision occur) so use following hash function,

h1(49)=(49 + 1)%10=0 hence insert key 49 in hash-table in location 0

when x=58:

h(58)=58%10=8 (Collision occur) so use following hash function,

h1(58)=(58 + 1^2)%10=9

again collision occur use again the following hash function ,

h2(58)=(58+ 2^2)%10=2

insert key 58 in hash-table in location 2

when x=69:

h(89)=69%10=9 (Collision occur) so use following hash function,

h1(69)=(69 + 1^2)%10=0

again collision occur use again the following hash function ,

h2(69)=(69+ 2^2)%10=3

insert key 69 in hash-table in location 3

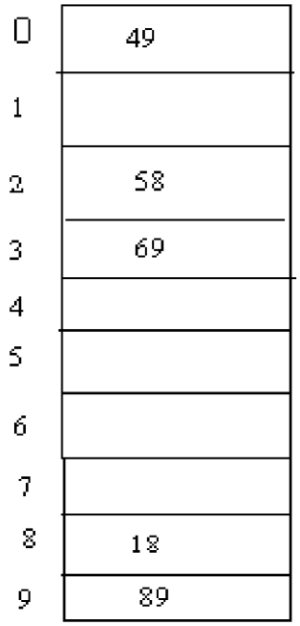
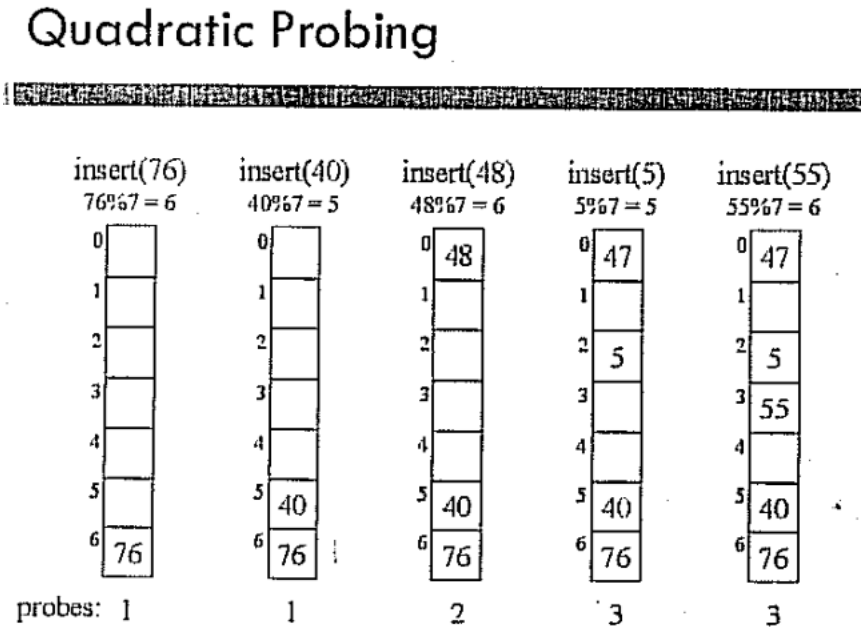


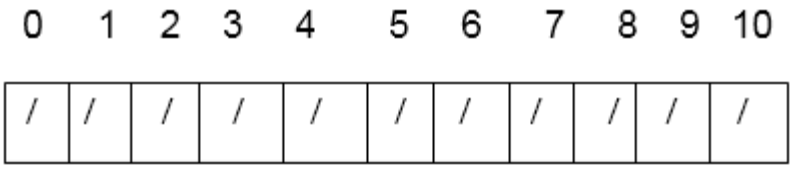
fig: Hash table for above keys using quadratic probing



Example: Consider inserting the keys 74, 28, 36,58,21,64 into a hash table of size m =11 using quadratic probing. Further consider that the primary hash function is h' (k) = k mod m.

Solution: For Quadratic Probing, we have

h (k, i) = [h' (k) $+i^2$) mod m



This is the initial state of hash table

$h(k, i) = [k \bmod m + i^2] \bmod m$

Insert 74.

h (74,0)= (74 mod 11+0^2) mod 11 = (8) mod 11 = 8

T [8] is free; insert the key 74 at this place.

Insert 28.

h (28, 0) = (28 mod 11 + 0^2) mod 11 = (6) mod 11 = 6.

T [6] is free; insert key 28 at this place.

Insert 36.

h (36, 0) = (36 mod 11 + 0^2) mod 11 = (3) mod 11=3

T [3] is free; insert key 36 at this place.

Insert 58.

h (58, 0) = (58 mod 11 + 0^2) mod 11 = (3) mod 11 = 3

T [3] is not free, so next probe sequence is computed as

h (59, 1) = (58 mod 11 + 1^2) mod 11 = (3 + 1) mod 11 =7 mod 11= 7

T [7] is free; insert key 58 at this place.

Insert 21.

h (21, 0) = (21 mod 11 + 0) = (10 + 0) mod 11 = 10

T [10] is free; insert key 21 at this place.

Insert 64.

h (64, 0) = (64 mod 11 + 0) = (9 + 0) mod 11 = 9.

T [9] is free; insert key 64 at this place.

Thus, after inserting all keys, the hash table is

0	1	2	3	4	5	6	7	8	9	10
/	/	/	36	/	/	28	58	74	64	21

Double Hashing:

Double Hashing is one of the best techniques available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. Double hashing uses a hash function of the form **h (k, i) = (h_1(k) + i*h_2 (k)) mod m** Where h_1 and h_2 are auxiliary hash functions and m is the size of the hash table. h_1 (k) = k mod m or h_2 (k) = k mod m'. Here m' is slightly less than m (say m-1 or m-2).

Example: Consider inserting the keys 76, 26, 37,59,21,65 into a hash table of size m = 11 using double hashing. Consider that the auxiliary hash functions are h_1 (k)=k mod 11 and h_2(k) = k mod 9.

Solution: Initial state of Hash table is

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 76.

h_1(76) = 76 mod 11 = 10, h_2(76) = 76 mod 9 = 4

h (76, 0) = (10 + 0 x 4) mod 11 = 10 mod 11 = 10

T [10] is free, so insert key 76 at this place.

2. Insert 26.

h_1(26) = 26 mod 11 = 4, h_2(26) = 26 mod 9 = 8

h (26, 0) = (4 + 0 x 8) mod 11 = 4 mod 11 = 4

T [4] is free, so insert key 26 at this place.

3. Insert 37.

h_1(37) = 37 mod 11 = 4, h_2(37) = 37 mod 9 = 1

h (37, 0) = (4 + 0 x 1) mod 11 = 4 mod 11 = 4

T [4] is not free, the next probe sequence is

h (37, 1) = (4 + 1 x 1) mod 11 = 5 mod 11 = 5

T [5] is free, so insert key 37 at this place.

4. Insert 59.

$h_1(59) = 59 \bmod 11 = 4$, $h_2(59) = 59 \bmod 9 = 5$

$h(59, 0) = (4 + 0 \times 5) \bmod 11 = 4 \bmod 11 = 4$

Since, T [4] is not free, the next probe sequence is

$h(59, 1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$

T [9] is free, so insert key 59 at this place.

5. Insert 21.

$h_1(21) = 21 \bmod 11 = 10$, $h_2(21) = 21 \bmod 9 = 3$

$h(21, 0) = (10 + 0 \times 3) \bmod 11 = 10 \bmod 11 = 10$

T [10] is not free, the next probe sequence is

$h(21, 1) = (10 + 1 \times 3) \bmod 11 = 13 \bmod 11 = 2$

T [2] is free, so insert key 21 at this place.

6. Insert 65.

$h_1(65) = 65 \bmod 11 = 10$, $h_2(65) = 65 \bmod 9 = 2$

$h(65, 0) = (10 + 0 \times 2) \bmod 11 = 10 \bmod 11 = 10$

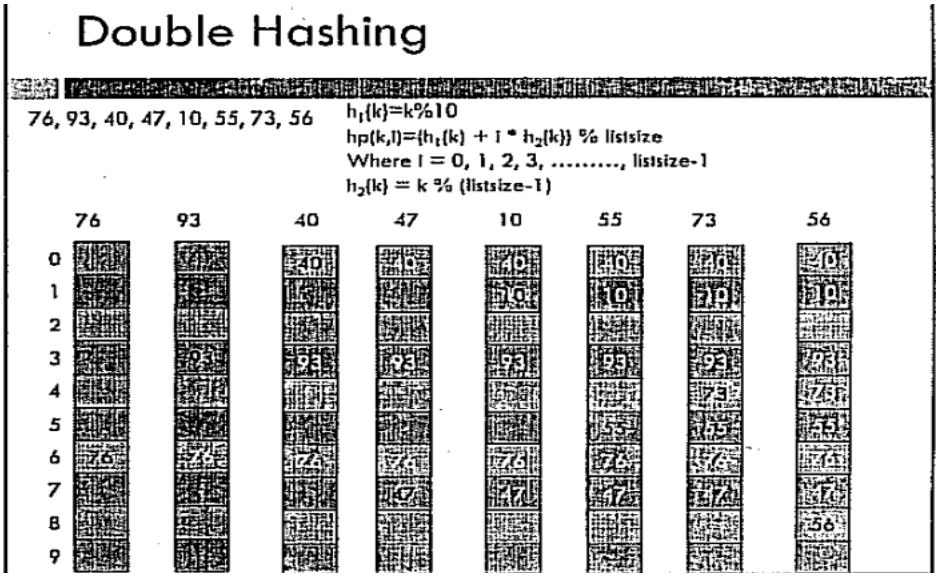
T [10] is not free, the next probe sequence is

$h(65, 1) = (10 + 1 \times 2) \bmod 11 = 12 \bmod 11 = 1$

T [1] is free, so insert key 65 at this place.

Thus, after insertion of all keys the final hash table is

0	1	2	3	4	5	6	7	8	9	10
/	65	21	/	26	37	/	/	/	59	76



Example: Consider inserting the keys 10, 22, 31,4,15,28,17,88 and 59 into a hash table of length m = 11 using open addressing with the primary hash function h' (k) = k mod m .Illustrate the result of inserting these keys using linear probing h (k, i) = (h' (k) + i) mod m, using quadratic probing h (k, i) = (h' (k) + i^2) mod m, and using double hashing with h (k, i) = (h_1(k) + i*h_2 (k)) mod m where h_1(k) = (k mod m) and h_2(k) = (k mod (m-1)).

Solution: Using Linear Probing the final state of hash table would be:

0	22
1	88
2	/
3	/
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Using Quadratic Probing, the final state of hash table would be h (k, i) = (h' (k) + i^2) mod m where m=11 and h' (k) = k mod m.

0	22
1	88
2	/
3	17
4	4
5	/
6	28
7	59
8	15
9	31
10	10

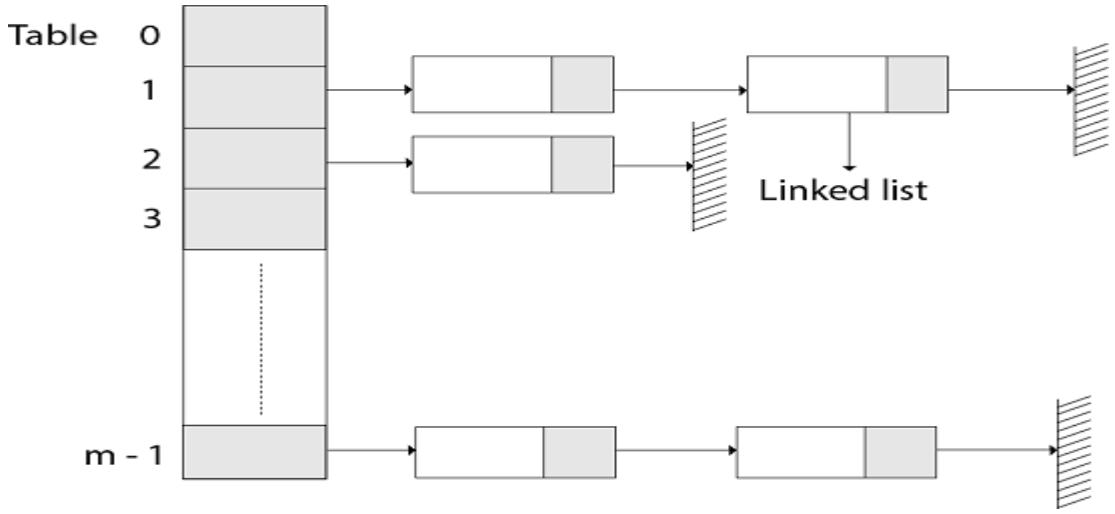
Using Double Hashing, the final state of the hash table would be:

0	22
1	/
2	59
3	17
4	4
5	15
6	28
7	88
8	/
9	31
10	10

Hashing with Chaining

In Hashing with Chaining, the element in S is stored in Hash table T [0...m-1] of size m, where m is somewhat larger than n, the size of S. The hash table is said to have m slots. Associated with the hashing scheme is a hash function h which is mapping from U to {0...m-1}.Each key k ∈S is stored in location T [h (k)], and we say that k is hashed into slot h (k). If more than one key in S hashed into the same slot then we have a **collision**.

In such case, all keys that hash into the same slot are placed in a linked list associated with that slot, this linked list is called the chain at slot. The load factor of a hash table is defined to be ∝=n/m it represents the average number of keys per slot. We typically operate in the range m=θ(n), so ∝ is usually a constant generally ∝<1.



Collision Resolution by Chaining:

In chaining, we place all the elements that hash to the same slot into the same linked list, As fig shows that Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.

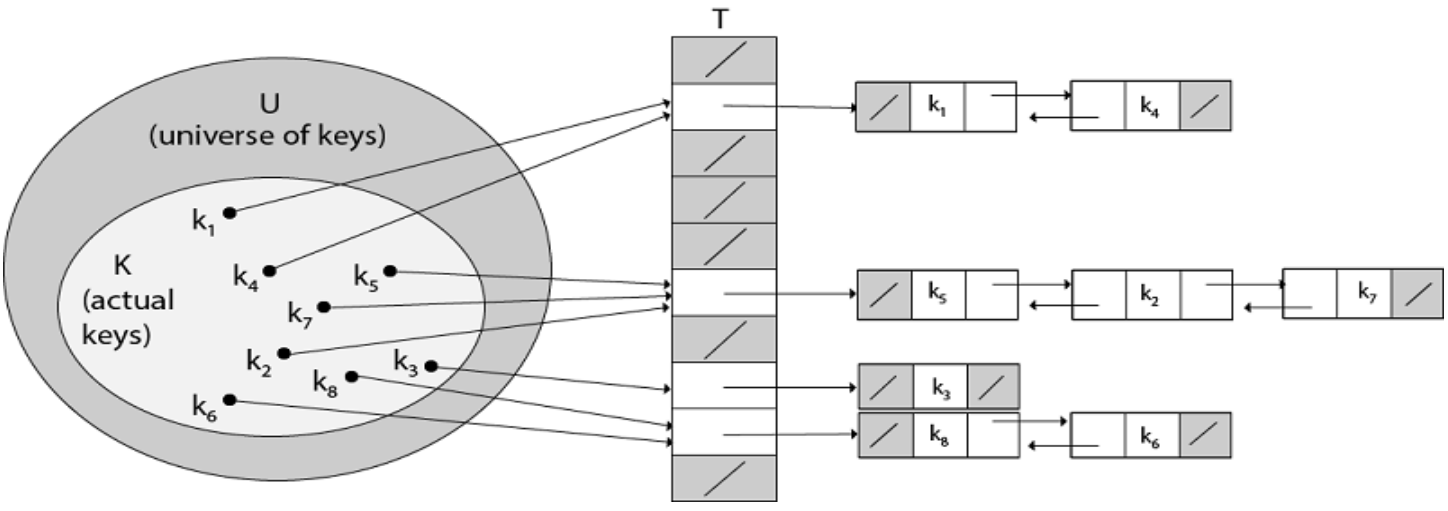


Fig: Collision resolution by chaining.

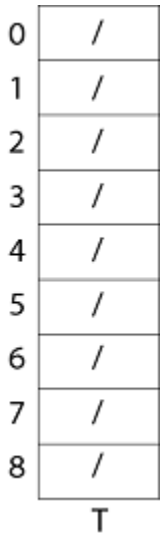
Each hash-table slot T [j] contains a linked list of all the keys whose hash value is j.

For example, h (k1) = h (k4) and h (k5) = h (k7) =h (K2). The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

Analysis of Hashing with Chaining:

Example: let us consider the insertion of elements 5, 28, 19,15,20,33,12,17,10 into a chained hash table. Let us suppose the hash table has 9 slots and the hash function be h (k) =k mod 9.

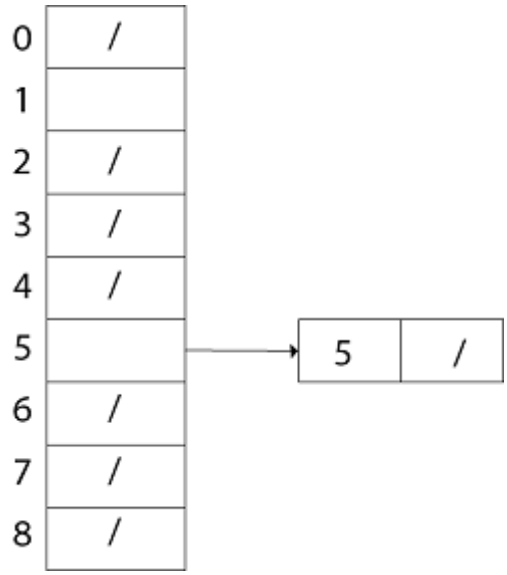
Solution: The initial state of chained-hash table



Insert 5:

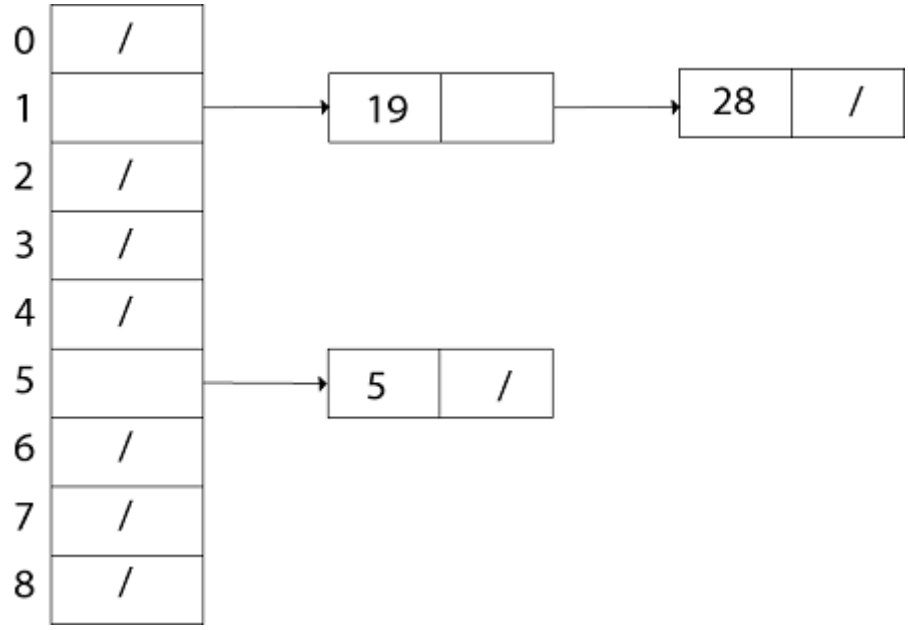
h (5) = 5 mod 9 =5

Create a linked list for T [5] and store value 5 in it.



Similarly, **insert 28,** h (28) = 28 mod 9 =1. Create a Linked List for T [1] and store value 28 in it.

Now **insert 19,** h (19) = 19 mod 9 = 1. Insert value 19 in the slot T [1] at the beginning of the linked-list.



Now **insert15**, h (15) = 15 mod 9 = 6. Create a link list for T [6] and store value 15 in it.

Similarly, **insert 20**, h (20) = 20 mod 9 = 2 in T [2].

Insert 33, h (33) = 33 mod 9 = 6

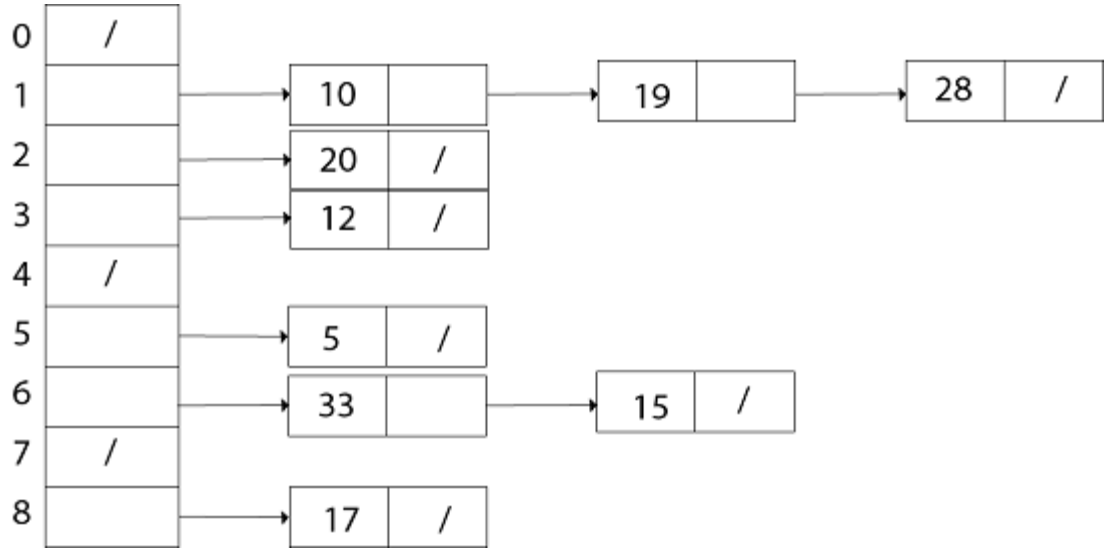
In the beginning of the linked list T [6]. Then,

Insert 12, h (12) = 12 mod 9 = 3 in T [3].

Insert 17, h (17) = 17 mod 9 = 8 in T [8].

Insert 10, h (10) = 10 mod 9 = 1 in T [1].

Thus the chained- hash- table after inserting key 10 is



Example: