Chapter 8 Searching

It is a process of finding an element within the list of elements stored in any order.

It is not necessary that the data item we are searching for must be present in the list.

If the searched item is present in the list then the searching algorithm (or program) can find that data item, in which case we say that the search is successful, but if the searched item is not present in the list, then it cannot be found and we say that the search is unsuccessful.

Types of Searching:

1. Linear /Sequential Searching:

- It is the simplest technique to find out an element in an unordered list.
- We search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.
- Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons. **The time complexity of linear search is O (n).**

Steps:

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function

Step 4 - If both are not matched, then compare search element with the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.

Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Chapter 8 Searching

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, and 20.

Index	0	1	2	3	4	5	6	7	8	9	10	11
list	45	39	8	54	77	38	24	16	4	7	9	20

- 24 is compared with first element (45). If not matched, move to next element.
- 24 is compared with second element (39). If not matched, move to next element.
- 24 is compared with third element (8). If not matched, move to next element.
- 24 is compared with fourth element (54). If not matched, move to next element.
- 24 is compared with fifth element (77). If not matched, move to next element.
- 24 is compared with Sixth element (38). If not matched, move to next element.
- 24 is compared with seventh element (24).Matched.

Sequential search efficiency:

- The number of comparisons of keys done in sequential search of a list of length n is
  i. Unsuccessful search : n comparisons
  ii. Successful search, best case : 1 comparison
  iii. Successful search, worst case : n comparisons
  iv. Successful search, average case : (n + 1)/2 comparisons
  v. In any case, the number of comparison is O(n)

2. Binary Search:

- It is an extremely efficient algorithm.
- This search technique searches the given item in minimum possible comparisons.
- To do the binary search, first we have to sort the array elements.
- The logic behind this technique is given below.
  i. First find the middle element of the array.
  ii. Compare the middle element with an item.
  iii. There are three cases:
    a. If it is a desired element then search is successful,
    b. If it is less than the desired item then search only in the first half of the array.
    c. If it is greater than the desired item, search in the second half of the array.
  iv. Repeat the same steps until an element is found or search area is exhausted.

Requirements:

i. The list must be ordered.
ii. Rapid random access is required, so we cannot use binary search for a linked list.

Chapter 8 Searching

Binary search efficiency:

i. In all cases, the no. of comparisons in proportional to n. Hence, no. of comparisons in binary search is O (log n), where n is the no. of items in the list.
ii. Obviously binary search is faster than sequential search, but there is an extra overhead in maintaining the list ordered.
iii. Binary search is best suited for lists that are constructed and sorted once, and then repeatedly searched.

Algorithm:

Given a table k of n elements in searching order searching for value x.

1. $Initialize: low \leftarrow 0, high \leftarrow n-1$

2. $Perform\ Search: Repeat\ through\ step\ 4\ while\ low <= high.$

3. $Obtain\ index\ of\ midpoint\ of\ interval: mid \leftarrow (low + high)/2$

4. $Compare:$

   $If\ X < k[mid] then\ high \leftarrow mid - 1$

   $Else\ if\ X > k[mid] then\ low \leftarrow mid + 1$

   $Else\ Write\ (\text{“}Search\ is\ successful\text{”})$

   $Return\ (mid)$

5. $(\text{“}Search\ is\ unsuccessful\text{”})$

   $Return$

6. $Finished$

Example 1:

0	1	2	3	4	5	6	7	8	9
22	43	68	100	120	330	420	555	560	570

Search 43

Insertion	Low	High	Mid	Remarks
1.	0	9	4	X<k[4]
2.	0	3	1	X=k[1]

Data found in Location 1.

Search 333

Insertion	Low	High	Mid	Remarks
1.	0	9	4	X>k[4]
2.	5	9	7	X<k[7]
3.	5	6	5	X>k[5]
4.	6	6	6	X<k[6]
5.	6	5	5	Low>High

Search Value 333 is not found.

Example 2

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, ***found***

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

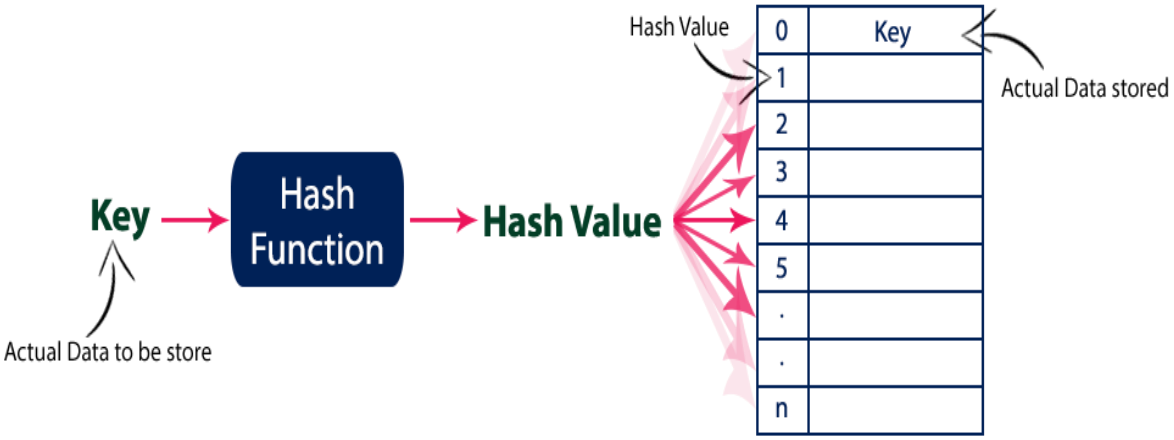
If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, ***found***

Chapter 8 Searching

Hashing:

- Hashing is the process of indexing and retrieving element in a data structure to provide faster way of finding the element using hash key. Hash key is a value which provides the index value where the actual data is to likely to be stored in data structure.
- It is the technique of representing longer records by shorter values called keys. Which are generated from a string of text using a mathematical function.
- The keys are placed in a table called hash table where the keys are compared for finding the roots.



Why hashing?

If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O (log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that function that would tell us the index for a given value. With this function our search is reduced to just one probe, giving us a constant runtime O (1). Though, it cannot guarantee a constant runtime for every case. Such a function is called a hash function. A hash function is a function which when given a key, generates an address in the table.

Hash Tables:

- A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named1, a slot named 2, and so on.

Chapter 8 Searching

- Hash table is just an array which maps a key (data) into data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity.
- In hash table, data is stored in an array format, where each data value has its own unique index value. Access to data becomes fast if we know the index of desired data.

Hash Function:

- Hash function is a mathematical formula which takes a piece of data as input and outputs an integer (i.e. hash value) which maps the data to a particular index in hash table.
- The main aim of a hash function is that elements should be uniformly distributed. It produces a unique set of integers within some suitable range in order to reduce the number of collisions.
- In practice, there is no hash function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

Characteristics of Good Hash Function

A good hash function should have the following characteristics:

- Minimize collision
- Be easy and quick to compute
- Distribute key values evenly in the hash table
- Use all the information provided in the key
- Have a high load factor for a given set of keys

How to Choose Hash Function?

The basic problems associated with the creation of hash tables are:

- An efficient hash function should be designed so that it distributes the index values of inserted objects uniformly across the table.
- An efficient collision resolution algorithm should be designed so that it computes an alternative index for a key whose hash index corresponds to a location previously inserted in the hash table.
- We must choose a hash function which can be calculated quickly, returns values within the range of locations in our table, and minimizes collisions.

Chapter 8 Searching

Different Hash Functions:

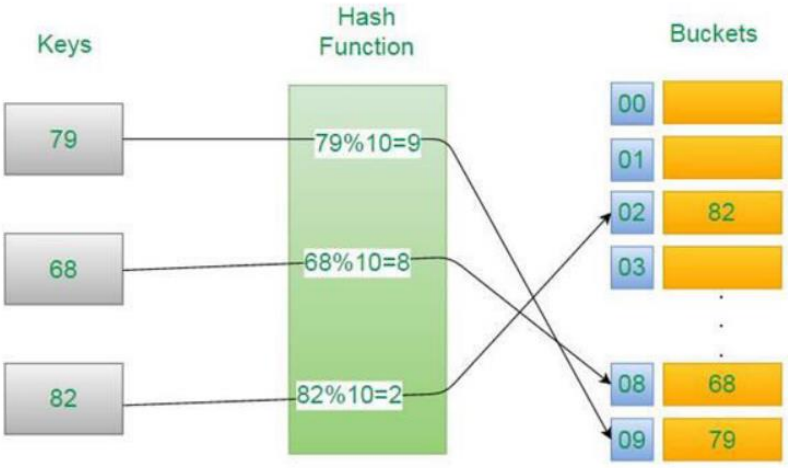
1. Folding Method:

- In this method, the given key is partitioned into subparts k1, K2, k3, k4 ..... kn each of which has the same length as the required address. Now add all these parts together and ignore the carry.
- For example:- if number of buckets be 100 and last address/index be 99, then the given key for which hash code is calculated is divided into parts of two digits from beginning as shown below:
- h(95073) = h(95 + 07 + 3)
    - = h(105) //ignoring the carry = 5
- Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.
- Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

Key	**5678**	**321**	**34567**
Parts	56 and 78	32 and 1	34,56 and 7
Sum	134	33	97
Hash Value	34(ignore the last carry)	33	97

2. Division Method:

- It is the most simple method of hashing an integer x. This method divides x by M (slots available) and then uses the remainder obtained.
- In this case, the hash function can be given as **z**
- For example: - Let us say apply division approach to find hash value for some values considering number of buckets be 10 as shown below.



3. Mid-Square Method:

- It is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

- The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.
- In it, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as: **h (k) = s** where s is obtained by selecting r digits from k^2.
- Example: Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.
- Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2.
- When k = 1234, k^2 = 1522756, h (1234) = 27
- When k = 5642, k^2 = 31832164, h (5642) = 21
- Observe that the 3rd and 4th digits starting from the right are chosen.

Chapter 8 Searching

Collisions:

- There is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.
- Two records cannot be stored in the same location of a hash table normally.

Collision Resolution and Clustering:

- Except direct hashing, none of the hashing methods are one-to- one mapping. Collisions are likely even if we have big table to store keys and hence require collision resolution techniques. Each collision resolution method can be used independently with each hash function. As data are added and collision are resolved, hashing tend to cause data to group within the list.

Collision Resolution Techniques:

1. **Separate Chaining (Open Hashing)**
2. **Open Addressing (Closed Hashing)**
    a. **Linear Probing**
    b. **Quadratic Hashing**
    c. **Double Hashing**

Separate Chaining (Open Hashing):

- In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. As new collisions occur, the linked list grows to accommodate those collisions forming a chain.
- Chained hash tables with linked lists are widely used due to the simplicity of the algorithms to insert, delete, and search a key. The code for these algorithms is exactly the same as that for inserting, deleting, and searching a value in a single linked list.
- Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use h (k) = k mod m. In this case, m=9. Initially, the hash table can be given as:

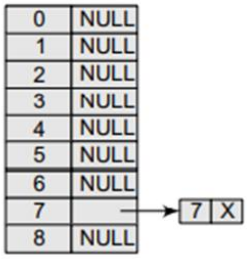
Chapter 8 Searching

Step 1 Key = 7

h(k) = 7 mod 9

= 7

Create a linked list for location 7 and store the key value 7 in it as its only node.

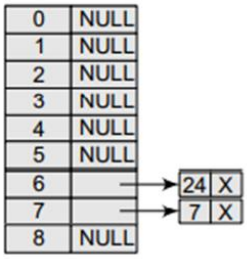


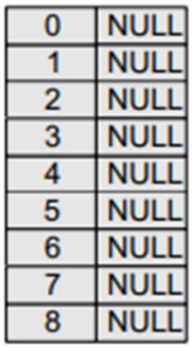
Step 2 Key = 24

h(k) = 24 mod 9

= 6

Create a linked list for location 6 and store the key value 24 in it as its only node.

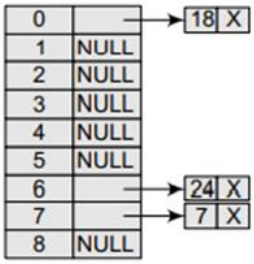




Step 3 Key = 18

h(k)= 18 mod 9 = 0

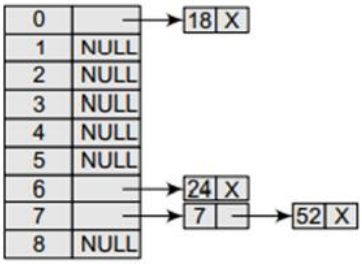
Create a linked list for location 0 and store the key value 18 in it as its only node.



Step 4 Key = 52

h(k) = 52 mod 9 = 7

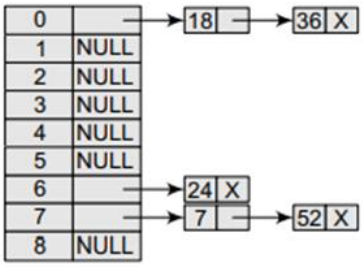
Insert 52 at the end of the linked list of location 7.



Step 5: Key = 36

h(k)= 36 mod 9 = 0

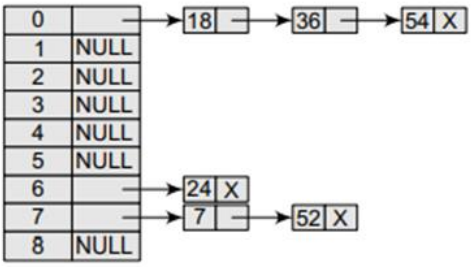
Insert 36 at the end of the linked list of location 0.



Step 6: Key = 54

h(k) = 54 mod 9 = 0

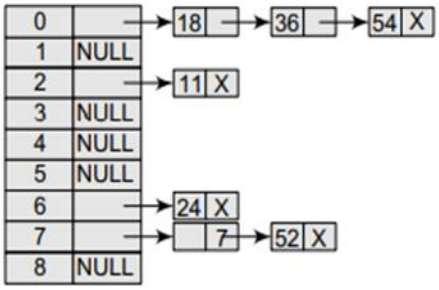
Insert 54 at the end of the linked list of location 0.



Step 7: Key = 11

h(k)= 11 mod 9 = 2

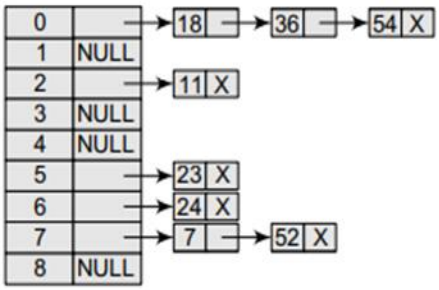
Create a linked list for location 2 and store the key value 11 in it as its only node.



Step 8: Key = 23

h(k) = 23 mod 9 = 5

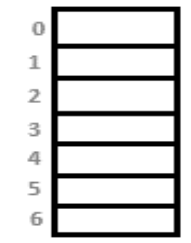
Create a linked list for location 5 and store the key value 23 in it as its only node.



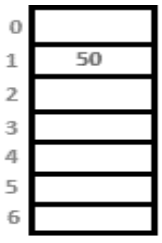
Example1 :

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

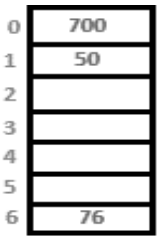
Then, h(50) = 50 mod 7 = 1
h(700) = 700 mod 7 = 0
h(76) = 76 mod 7 = 6
h(85) = 85 mod 7 = 1
h(92) = 92 mod 7 = 1
h(73) = 73 mod 7 = 3
h(101) = 101 mod 7 = 3



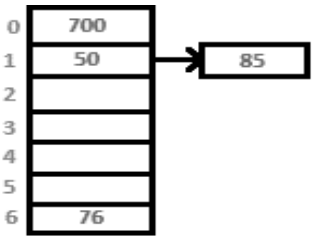
Initial Empty Table



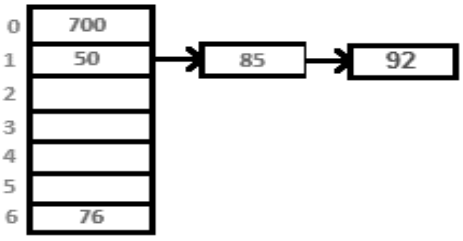
Insert 50



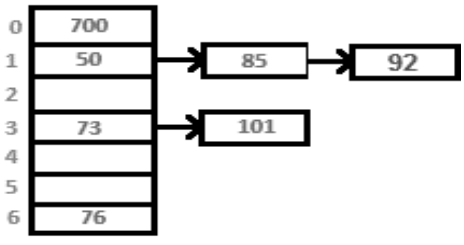
Insert 700 and 76



Insert 85: Collision Occurs, add to chain



Inser 92 Collision Occurs, add to chain



Insert 73 and 101

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- Wastage of Space (Some Parts of hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case.
- Uses extra space for links.

Open Addressing:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys

Insert (k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search (k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete (k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.

Open Addressing is done following ways:

a. **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 for x using hash function and m be the table size.

$$\boldsymbol{hash}(\boldsymbol{x}) = \boldsymbol{k \bmod m}$$

$$\boldsymbol{hash}(\boldsymbol{x}, \boldsymbol{i}) = (\boldsymbol{hash}(\boldsymbol{x}) + \boldsymbol{i}) \,\%\, \boldsymbol{m \; where \; i \; is \; prob \; or \; collision \; number}$$

- Calculate the hash key. **h´(k) = k** $mod\ m$
- If hashTable [key] is empty, store the value directly, hashTable [key] = data.
- If the hash index already has some value, check for next index.
- h**(k,** i**) = (**h**´(k) +** i**)**$mod\ m$**;**
- If the next index is available hashTable [key], store the value. Otherwise try for next index.
- Do the above process till we find the space.

Example 1:

Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92 and 101 into the table.

- Let h'(k) = k mod m, m = 10

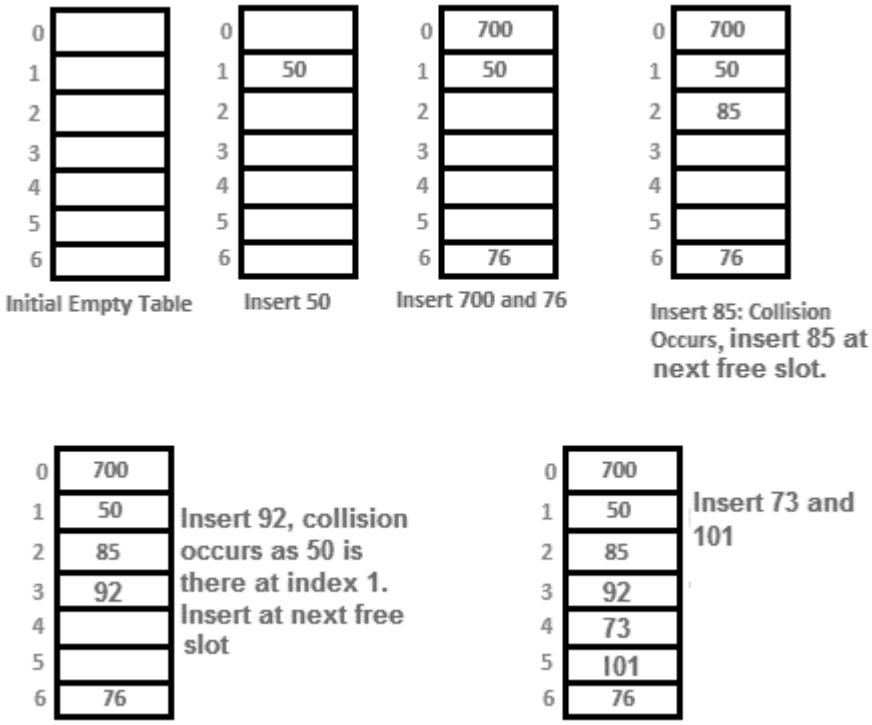
k	**h(k,i)= (h'(k) + i) mod 10**
72	h(72,0)=(72 mod 10 + 0) mod 10=2 mod 10 =2
27	h(27,0)=(27 mod 10 + 0) mod 10=7 mod 10 =7
36	h(36,0)=(36 mod 10 + 0) mod 10=6 mod 10 =6
24	h(24,0)=(24 mod 10 + 0) mod 10=4 mod 10 =4
63	h(63,0)=(63 mod 10 + 0) mod 10=3 mod 10 =3
81	h(81,0)=(81 mod 10 + 0) mod 10=1 mod 10 =1
92	h(92,0)=(92 mod 10 + 0) mod 10=2 mod 10 =2(A[2] is occupied) Then i=1, h(92,1)=(92 mod 10 + 1) mod 10=3 mod 10 =3(A[3] is occupied) Then i=2, h(92,2)=(92 mod 10 + 2) mod 10=4 mod 10 =4(A[4] is occupied) Then i=3, h(92,3)=(92 mod 10 + 3) mod 10=5 mod 10 =5
101	h(101,0)=(101 mod 10 + 0) mod 10=1 mod 10 =1 (A[1] is occupied) Then i=1, h(101,1)=(101 mod 10 + 1) mod 10=2 mod 10 =2(A[2] is occupied) Repeat process until i=7.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	81	**72**	**63**	**24**	**92**	**36**	**27**	**101**	

Example 2:

Let us consider a simple hash function as “key mod 7” and sequence of keys

as 50, 700, 76, 85, 92, 73, 101. Use linear probing and insert the keys into table.



Advantages:

- No extra space

Disadvantages:

- Searching Difficult
- Primary Clustering
- Secondary Clustering

b. **Quadratic Hashing:**

- In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision**:**

$$\boldsymbol{h(k,i) = [h'(k) + i^2]\, mod\ m}\ where\ m\ is\ the\ size\ of\ the\ hash\ table,$$

$$\boldsymbol{h'(k) = (k\ mod\ m)}, i\ is\ the\ probe\ number\ that\ varies\ from\ 0\ to\ m-1.$$

- It eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search.

$$let\ hash(x) be\ the\ slot\ index\ computed\ using\ hash\ function.$$

$$If\ slot\ hash(x)\ \%\ S\ is\ full, then\ we\ try\ (hash(x) + 1*1)\ \%\ S$$

$$If\ (hash(x) + 1*1)\%\ S\ is\ also\ full, then\ we\ try\ (hash(x) + 2*2)\%\ S$$

$$If\ (hash(x) + 2*2)\ \%\ S\ is\ also\ full, then\ we\ try\ (hash(x) + 3*3)\ \%\ S$$

……

……..

Example 1:

Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table.

Let h'(k) = k mod m,

m = 10

k	**h(k,i)=(h'(k)+i^2) mod 10**
72	h(72,0)=(72 mod 10 + 0^2) mod 10=2 mod 10 =2
27	h(27,0)=(27 mod 10 + 0^2) mod 10=7 mod 10 =7
36	h(36,0)=(36 mod 10 + 0^2) mod 10=6 mod 10 =6
24	h(24,0)=(24 mod 10 + 0^2) mod 10=4 mod 10 =4
63	h(63,0)=(63 mod 10 + 0^2) mod 10=3 mod 10 =3
81	h(81,0)=(81 mod 10 + 0^2) mod 10=1 mod 10 =1
101	h(101,0)=(101 mod 10 + 0^2) mod 10=1 mod 10 =1 [is occupied] h(101,1)=(101 mod 10 + 1^2) mod 10=2 mod 10 =2 [is occupied] h(101,2)=(101 mod 10 + 2^2) mod 10=5 mod 10 =5

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	101	36	27		

Example 2:

Let us consider a simple hash function as "key mod 10" and sequence of keys as 42, 16, 91, 33, 18, 27, 36.

k	$h(k,i)=(h'(k)+i^2) \bmod 10$
42	$h(42,0)=(42 \bmod 10 + 0^2) \bmod 10=2 \bmod 10 =2$
16	$h(16,0)=(16 \bmod 10 + 0^2) \bmod 10=6 \bmod 10 =6$
91	$h(91,0)=(91 \bmod 10 + 0^2) \bmod 10=1 \bmod 10 =1$
33	$h(33,0)=(33 \bmod 10 + 0^2) \bmod 10=3 \bmod 10 =3$
18	$h(18,0)=(18 \bmod 10 + 0^2) \bmod 10=8 \bmod 10 =8$
27	$h(27,0)=(27 \bmod 10 + 0^2) \bmod 10=7 \bmod 10 =7$
36	$h(36,0)=(36 \bmod 10 + 0^2) \bmod 10=6 \bmod 10 =6$ [is occupied] $h(36,1)=(36 \bmod 10 +1^2) \bmod 10=7 \bmod 10 =7$ [is occupied] $h(36,2)=(36 \bmod 10 + 2^2) \bmod 10=10 \bmod 10 =0$

0	1	2	3	4	5	6	7	8	9
36	91	42	33			16	27	18	

Advantages:

- No extra space
- Primary Clustering Resolved

Disadvantages:

- Secondary Clustering
- No guarantee of finding slot

c. **Double Hashing:**

- It uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name double hashing.
- In double hashing, we use another hash function hash2(x) and look for $i * hash2(x)$ slot in i'th rotation.

$$\boldsymbol{hash1(x) = k \bmod m}$$

$$\boldsymbol{hash(x,i) = \big(hash1(x) + i * hash2(x)\big) \bmod m}$$

$$\boldsymbol{where\ i\ is\ the\ probe\ number\ that\ varies\ from\ 0\ to\ m-1, and}$$

$$\boldsymbol{hash2(x) = k \bmod R, R\ is\ a\ (m-1)\ or\ (m-2)\ just\ less\ than\ the\ table\ size\ m}$$

$Let\ hash(x)\ be\ the\ slot\ index\ computed\ using\ hash\ function.$

$If\ slot\ hash(x)\ \%\ m\ is\ full, then\ we\ try\ (hash(x) + 1 * hash2(x))\ \%\ m$

$If\ \big(hash(x) + 1 * hash2(x)\big)\%\ m\ is\ also\ full, then\ we\ try\ \big(hash(x) + 2 * hash2(x)\big)\%\ m$

$If\ \big(hash(x) + 2 * hash2(x)\big)\%\ m\ is\ also\ full, then\ we\ try\ \big(hash(x) + 3 * hash2(x)\big)\ \%\ m$

……

……

Example 1:

Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table. Take h1 = (k mod 10) and h2 = (k mod 8). [Let m = 10]

k	$h(k,i)=[h_1(k)+ih_2(k)] \bmod m$

72	h(72,0)=[72 mod 10 + 0(72 mod 8)]mod 10=2
27	h(27,0)=[27 mod 10 + 0(27 mod 8)]mod 10=7
36	h(36,0)=[36 mod 10 + 0(36 mod 8)]mod 10=6
24	h(24,0)=[24 mod 10 + 0(24 mod 8)]mod 10=4
63	h(63,0)=[63 mod 10 + 0(63 mod 8)]mod 10=3
81	h(81,0)=[81 mod 10 + 0(81 mod 8)]mod 10=1
92	h(92,0)=[92 mod 10 + 0(92 mod 8)]mod 10=2 [Collision since A[2] is occupied.] h(92,1)=[92 mod 10 + 1(92 mod 8)]mod 10=(2+4) mod 10 = 6 [Collision since A[6] is occupied.] h(92,2)=[92 mod 10 + 2(92 mod 8)]mod 10=(2+2*4) mod 10= 0
101	h(101,0)=[101 mod 10 + 0(101 mod 8)]mod 10=1[Collision since A[1] is occupied.] h(101,1)=[101 mod 10 + 1(101 mod 8)]mod 10=6[Collision since A[6] is occupied.] h(101,2)=[101 mod 10 + 2(101 mod 8)]mod 10=1[Collision since A[1] is occupied.] Repeat the entire process until a vacant location is found. We will see that we have to probe many times to insert the key 101 in the hash table.

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24		36	27		

Advantages:

- No extra space
- No primary Clustering
- No Secondary Clustering

Disadvantages:

- Requires more computation time as two hash functions need to be computed.