```
                loc=mid;
                flag=i;
                break;
            }
        else if (item < a[mid])
        {
            end=mid - 1;
        }
            else
            beg=mid + 1,
        }
    }
            if(flag==1)
        {
        printf("\n Search in successful");
        printf(" Position of the item %d\n", loc + 1);
            }
        else
            printf("\n search is not successful");
        getch();
    }
```

☞ **Note :** The limitation of using the Binary search is that, the number must be in sorted order. If the number are not in sorted order then firstly you have to sort the number in Ascending order.

8.2.4 Complexity of Binary Search Algorithm

The complexity is measured by the No. $f(n)$ of comparisons to locate item in Data where Data Contains n elements.

- Observe that each comparison reduces the sample size in half. Hence we require at must $f(n)$ comparison to locate item where

$$(2)^{f(n)} > n. \qquad f(n) = \log_2(n) + 1$$

That is the running time for the worst case is approximately equal to $\log_2 n$.

8.3 COMPARISON OF SEQUENTIAL AND BINARY SEARCH

Consider the above set of elements

11	2	8	14	56	24	16	1	89	3

Now we sort the above array

1	2	3	8	11	14	16	24	56	89

Suppose we want to search 24 in the above sorted array of numbers. By linear search method we require 8 comparisons to search element 24 while by binary search method it takes only 3 comparison. The advantage of the binary search method is that, in each iteration it reduces the number of elements to be searched from n to $\frac{n}{2}$. On the other hand, linear

search method checks sequentially for every element, which makes it inefficient. There disadvantage of binary search is that it works only on sorted lists. So when searching is performed on unsorted list then linear search is the only option.

8.4 ANALYSIS OF SEQUENTIAL AND BINARY SEARCH

- The performance of the sequential search is measured by counting the numb comparisons used to find the element. There are two cases for the sequential se one is search on sorted array and second search on unsorted array.

  In the case of unsorted array

  Best Case

  In best case, if the element to be searched is at first position then linear se requires only one comparison.

  In average case, if the element is not present in either first position or at position then the searching an element requires average number of comparison

  In worst case, if the element is present in the last location then the numb comparison required is n.

  In the case of sorted array

  In best case, if the element to be searched is at first position then linear se requires only one comparison.

  In average case, if the element is not present in either first position or at position then the searching an element requires average number of comparisor

  In worst case, if the element is present in the last location then the numb comparison required is n.

- Binary search can be performed only on sorted array. For the performance ana of the binary search method again we take a sorted array.

  100, 200, 300, 400, 500, 600, 700

  In best case, if the element to be searched is there in the middle of the array function requires only one comparison to yield best case.

  In average case, the total numbers of comparison needed to find all the 7 elen of the above array and dividing it by 7 yields, 2.428 comparisons per succe search. This is almost equal to $\log_2 n$.

  In worst case, it is the maximum number of comparison the binary search algo has taken to search the element. For the array shown in above the maxi number of comparison required always are 3. Hence, 3 is almost equal to $\log_2 n$

8.5 HASH TABLE AND HASHING

Introduction : In all the search algorithm considered so for, the location of the it determined by a sequence of comparisons. In each case, a data item saught is repea compared with items in certain locations of the Data structure. However the numb Actual Comparison depends on the Data structure and the search algo used.

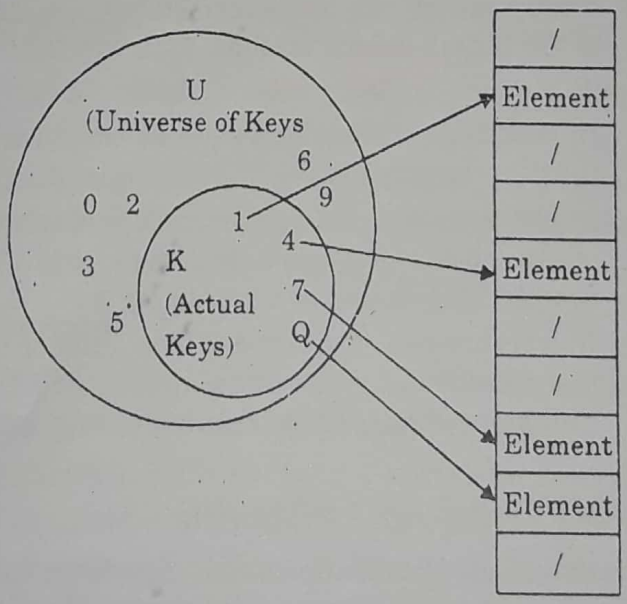
For example :

- In an Array and linked list, the Linear search requires $O(n)$ comparisons.
- In a Sorted Array, the Binary search requires $O(\log_2 n)$ comparisons.
- In a Binary Search Tree, search requires $O(\log_2 n)$ comparisons.

- However there are some applications that require search to be performed in constant time *i.e.*, O(*i*). Ideally it may not be possible, but still we can achieve a performance very close to it.
- This is possible using a Data structure known as Hash Table.
- A Hash Table in basic sense, is a generalization of the simpler notion of an ordinary array. Directly addressing into an array makes it possible to access any data item (Element) of the Array in O(1) Time. *For example*, if a[1...100] is an ordinary Array, then the nth Data element 1 < n < 100, can be directly accessed as a[n]. However Direct addressing is applicable only when we can allocate an array that has one position for every possible key.
- In addition Direct Addressing suffers from following problem :
  - (*i*) If the total number of possible keys is very large, it may not be possible to allocate an array of that size because of the memory available in the system or the applications software does not permit it.
  - (*ii*) If the actually number of keys is very small as compared to total number of possible keys, lot of space in the array will be waste. Now we have to Remove these disadvantages of direct access table with the help of special Data structure *i.e.*, known as *Hash Table*.

8.5.1 Direct Address Tables

Direct Addressing is a simple technique that works quite well when the universe 'U' of keys is reasonably small.



[Implementing a dynamic set by a Direct Address Table T, where the elements are stored in the table itself].

8.5.2 Hash Table

A Hash Table is a Data Structure in which the location of a Data item is determined directly as a function of the data item rather than by a sequence of comparisons. Under ideal conditions, the time required to locate a data item in a Hash Table is O(1) *i.e.*, it is constant and does not depend on the number of data items stored.

search method checks sequentially for every element, which makes it inefficient. There is one disadvantage of binary search is that it works only on sorted lists. So when searching is to be performed on unsorted list then linear search is the only option.

8.4 ANALYSIS OF SEQUENTIAL AND BINARY SEARCH

- The performance of the sequential search is measured by counting the number of comparisons used to find the element. There are two cases for the sequential search one is search on sorted array and second search on unsorted array.

  In the case of unsorted array

  Best Case

  In best case, if the element to be searched is at first position then linear search requires only one comparison.

  In average case, if the element is not present in either first position or at last position then the searching an element requires average number of comparisons.

  In worst case, if the element is present in the last location then the number of comparison required is n.

  In the case of sorted array

  In best case, if the element to be searched is at first position then linear search requires only one comparison.

  In average case, if the element is not present in either first position or at last position then the searching an element requires average number of comparisons.

  In worst case, if the element is present in the last location then the number of comparison required is n.

- Binary search can be performed only on sorted array. For the performance analysis of the binary search method again we take a sorted array.

  100, 200, 300, 400, 500, 600, 700

  In best case, if the element to be searched is there in the middle of the array then function requires only one comparison to yield best case.

  In average case, the total numbers of comparison needed to find all the 7 elements of the above array and dividing it by 7 yields, 2.428 comparisons per successful search. This is almost equal to $\log_2 n$.

  In worst case, it is the maximum number of comparison the binary search algorithm has taken to search the element. For the array shown in above the maximum number of comparison required always are 3. Hence, 3 is almost equal to $\log_2 n$.

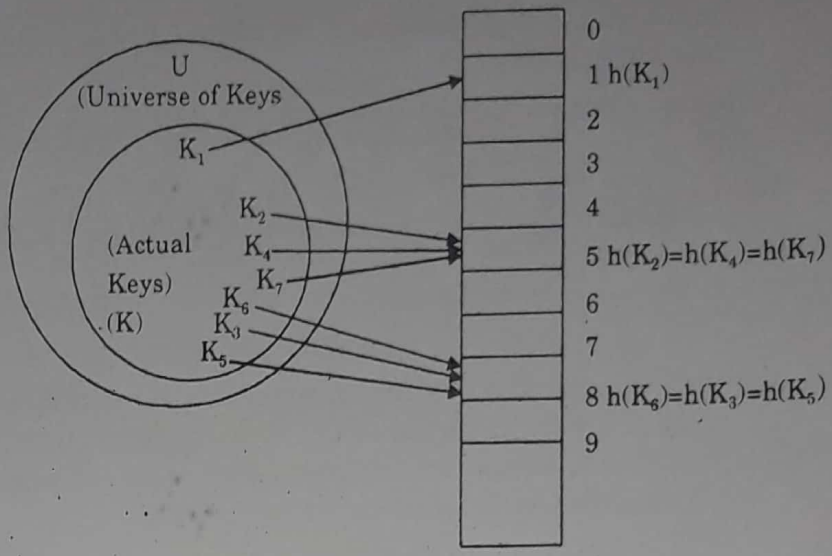
8.5 HASH TABLE AND HASHING

Introduction : In all the search algorithm considered so for, the location of the item is determined by a sequence of comparisons. In each case, a data item saught is repeatedly compared with items in certain locations of the Data structure. However the number of Actual Comparison depends on the Data structure and the search algo used.

For example :

- In an Array and linked list, the Linear search requires $O(n)$ comparisons.
- In a Sorted Array, the Binary search requires $O(\log_2 n)$ comparisons.
- In a Binary Search Tree, search requires $O(\log_2 n)$ comparisons.

Implementing a dynamic set by a hash table T[0.... m–1], where the elements are stored in the table itself.



- The mapping of more than one key maps to the same slot is known as *collision*.

8.5.3 How to Resolve the Collisions

We can resolve the collisions with the help of following strategies :

- Collision resolution by separate chaining.
- Collision resolution by open addressing.

8.6 HASH FUNCTIONS

A hash function *h* is simply a mathematical formula that manipulates the key in some form to compute the index for this keys in the hash table.

In general, we say that a hash function h maps the universe U of keys into the slot of a hash table T [0.... m–1]. This process of mapping keys to appropriate slots in a hash table is known as hashing.

Different Hash Functions

There is variety of hash functions. But before we discuss them, lets what are main considerations while choosing a particular hash function.

The main consideration while choosing a particular hash function h are :

- It should be possible to compute it efficiently.
- It should distribute the keys uniformly across the hash table *i.e.*, it should keep number of collisions as minimum as possible.

8.6.1 Division Method

In division method, key K to be mapped into one of the *m* slots in the hash table is divided by m and the remainder of this division is taken as index into the hash table *i.e.*,

The has function is $h(k) = k \bmod m$

8.6.4 Folding Method

The folding method also operates in two steps :

- In the first step, the key value k is divided into number of parts *i.e.*, $K_1, K_2, \ldots K_r$, where each part has the same number of digits except the last part, which can have lesser digits.
- In the second step, these parts are added together and the hash value is obtained by last carry it away.

Problem : *Consider a hash table with 100 slots i.e., m = 100 and key values K = 9235, 714, 71458.*

Solution :

K	9235	714	71458
Parts	92, 35	71, 4	71, 45, 8
Sum of pairs	127	75	114
h(k)	27	75	14

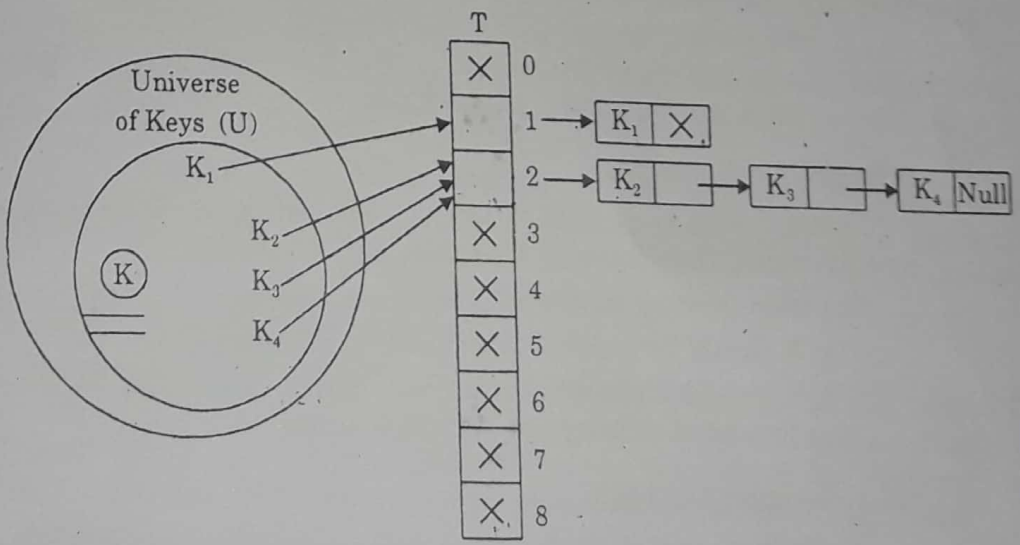
In above example, we have assumed that keys are numeric. However if the keys are alphanumeric, then the ASCII codes of fixed number of character can be added together to transfer the character keys to its equivalent numeric key and then any of the above hash functions can be applied.

8.7 RESOLVING COLLISIONS

A collision is a process that occurs when more than one key maps to same slot in the hash table. Though we can keep collisions to a certain minimum level, but we cannot eliminate them altogether. Therefore we need some mechanism to handle them.

8.7.1 Collision Resolution by Separate Chaining

In this scheme, all the elements whose keys has to the same hash table slot are put in a one linked list. Thus, the slot i in the hash table contain a Pointer to the head of the linked list of all the elements that hashes to value i. If there is no such element that hash to value i, the slot i contains Null pointer.



where mod is a modulus operator and is supported by Pascal Language in C|C++|Java we use % sign in place of mod.

- The above hash function will map the keys in the range 0 to m–1 and is acceptable is C|C++ and Java. But if some language supports the index from 1 onwards, then the Hash function becomes

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

For example : Consider a hash table with 9 slots now the key 132 will map which slot ?

$$h(132) = K \bmod m \Rightarrow 132 \bmod 9 \Rightarrow 7$$

Since it requires only a single division operation *i.e.*, hashing is quite fast.

8.6.2 Multiplication Method

The multiplication method operates in two steps :

- In the first step, the key value K is multiplied by a constant A in the range $0 < A < 1$ and extract the fractional part of value KA. *i.e.*, Hash Function is

$$h(k) = \text{floor}\,(m(kA \bmod 1))$$

A is constant $\Rightarrow \left(\frac{\sqrt{(5-1)}}{2}\right) = 0.6180339887$

Example : Consider a hash table with 10000 slots *i.e.*, $m = 10{,}000$ then the hash function

$$h(k) = \text{floor}\,(m(KA \bmod 1))$$

will map the key 1 2 3 4 5 6 to slot 41 *i.e.*,

$$h\,(123456) = \text{floor}\,(10000 \times (123456 \times 0.61803 \bmod 1))$$
$$h\,(123456) = \text{floor}\,(10000 \times (76300.0041151 \bmod 1))$$
$$h\,(123456) = \text{floor}\,(10000 \times 0.0041151)$$
$$123456 = \text{floor}\,(41.151)$$
$$h\,(123456) = 41$$

8.6.3 Mid Square Method

The mid square method also operates in two steps :

- In the first step, the square of the key value is taken, In the second step, the hash value is obtained by deleting digits from end of the squared value *i.e.*, K^2. It is important to note that same position of K^2 must be used for all keys.

i.e., The hash function is

$$h(k) = s$$

where s is obtained by deleting digits from both sides of K^2.

Problem : *Consider a hash table with 100 slots i.e., m = 100 and key values k=3205, 7148, 2345.*

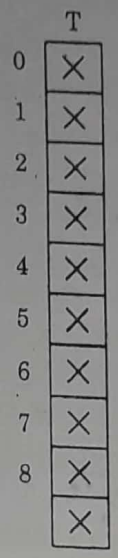
Solution :

K	3205	7148	2345
K^2	10272025	51093904	05499025
h(K)	(72)	(93)	(99)

The hash values are obtained by taking fourth and fifth digits counting from right.

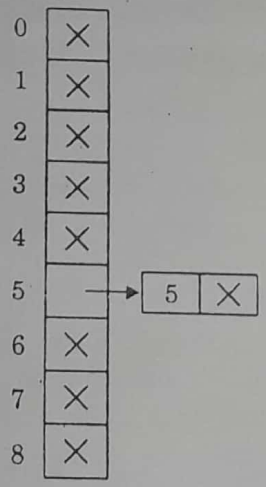
Problem : *Let us consider the insertion of elements 5, 28, 19, 15, 20, 33, 12, 17, 10 into a chained hash table. Ley us suppose that hash table has 9 slots and the hash function be h(k) = k mode 9.*

Solution : To begin with the chained hash table in initialized with Null Pointers as shown below :



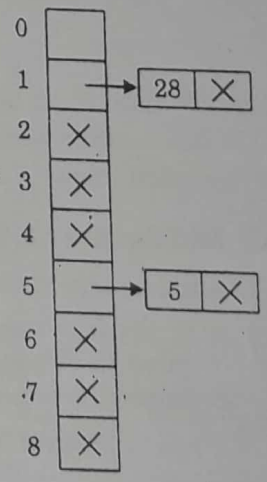
Initial state of
Chained Hash Table

h (5) = k mod 9 = 5 mod 9 = 5



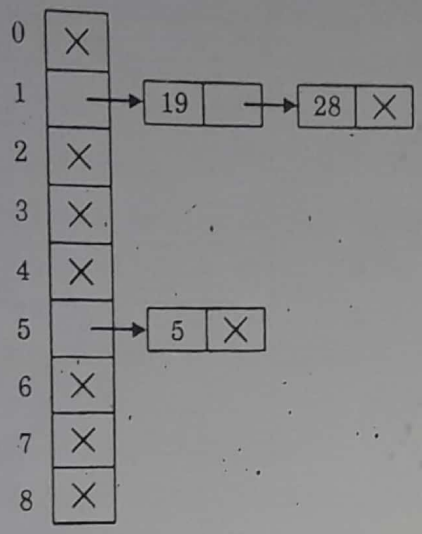
Chained hash table after Inserting 5

h (28) = 28 mod 9 = 1



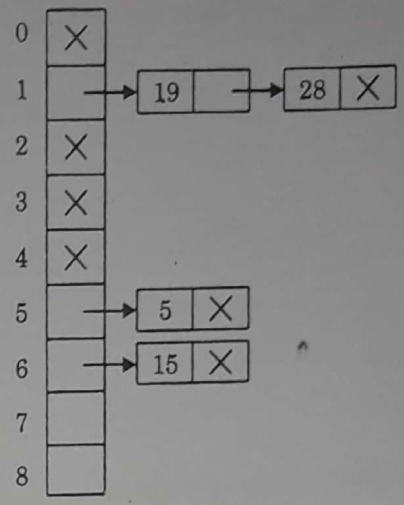
Chained hash table after Inserting 5, 28

h (19) = 19 mod 9 = 1



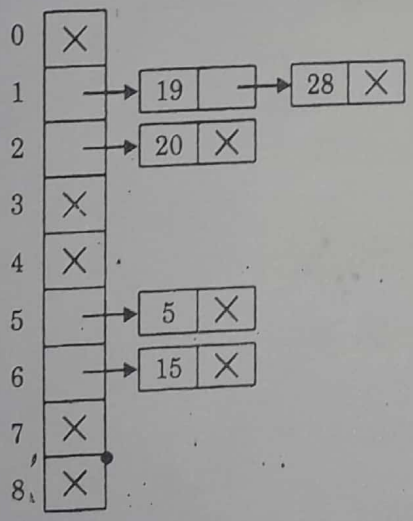
Chained hash table after inserting 5, 28, 19

h (15) = 15 mod 9 = 6



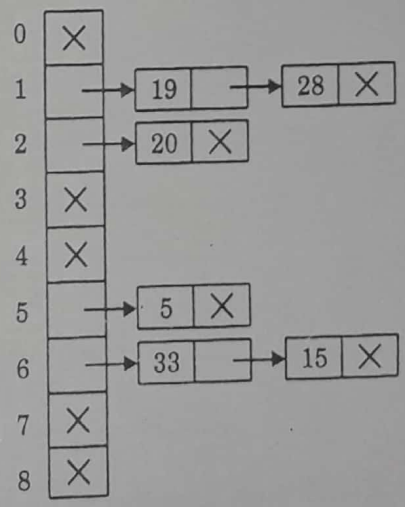
Chained hash table after inserting 5, 28, 19, 15

h (20) = 20 mod 9 = 2

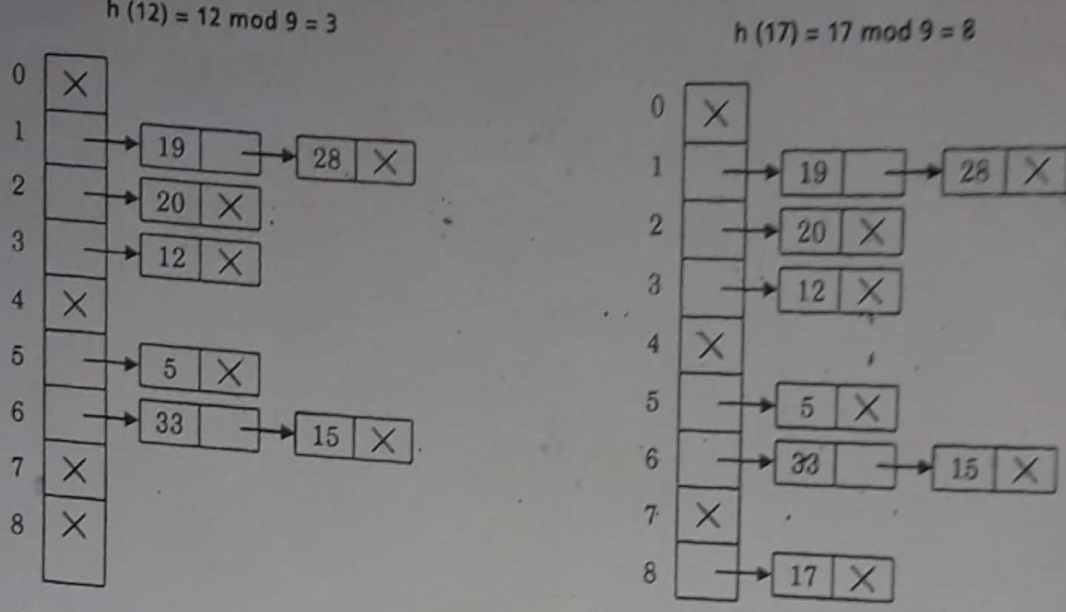


Chained hash table after inserting 5, 28, 19, 15, 20

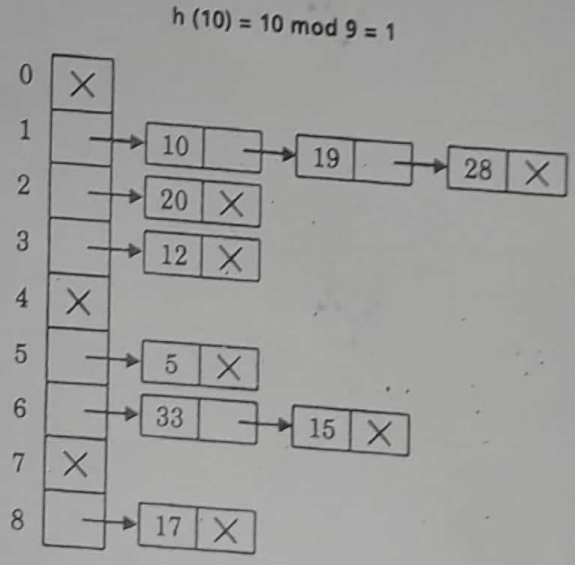
h (33) = 33 mod 9 = 6



Chained hash table after inserting 33



Chained hash table after inserting 12



This is Final Answer

8.7.2 Collision Resolution by Open Addressing

In the open addressing scheme all the elements of the dynamic set are stored in the hash table itself. *i.e.*, each entry of the hash table either contains the element of the dynamic set or some sentinel value to indicate that slot is free.

- In order to insert a key in a hash table under this scheme, if the slot to which key is hashed is free then the element is stored at that slot. In case the slot is filled, then other slots are examined systematically in the forward direction. If no such slot is found then overflow conditions occurs.

- In each case the process of examining the slots in the hash table is called probing. Probing is actually divided into three categories :
  (*a*) Linear Probing (*b*) Quadratic Probing (*c*) Double Hashing

(a) Linear Probing

The Linear probing uses the following hash function.

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

for $i = 0, 1, 2, \ldots\ldots m - 1$

where m is the size of has table and

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

and i is the prob number.

Problem : *Consider inserting the keys 76, 26, 37, 59, 21, 65, 88 into a hash table of size m = 11 using Linear Probing. Further consider that the primary hash function*

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

Solution :

×	×	×	×	×	×	×	×	×	×	×
0	1	2	3	4	5	6	7	8	9	10

Initial state of Hash Table

Step 1 :

$$h\,(76, 0) = (k \bmod m + i) \bmod m$$
$$h\,(76, 0) = (76 \bmod 11 + 0) \bmod 11$$
$$h\,(76, 0) = (10) \bmod 11 = 10$$

–	–	–	–	–	–	–	–	–	–	76
0	1	2	3	4	5	6	7	8	9	10

Step 2 :

$$h\,(26, 0) = (26 \bmod 11 + 0) \bmod 11$$
$$= (4) \bmod 11 = 4$$

				26						76
0	1	2	3	4	5	6	7	8	9	10

Step 3 :

$$h\,(37, 0) = (37 \bmod 11 + 0) \bmod 11$$
$$= (4) \bmod 11 = 4$$

Since slot T(4) is occupied the next prob sequence is computed as

$$h\,(37, 1) = (37 \bmod 11 + 1) \bmod 11$$
$$= (4 + 1) \bmod 11 = 5 \bmod 11 = 5$$

×	×	×	×	26	37	×	×	×	×	76
0	1	2	3	4	5	6	7	8	9	10

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

Since T[4] is already occupied thus

$$T(59, 1) = (59 \bmod 11 + 1) \bmod 11$$
$$T(59, 1) = (4 + 1) \bmod 11 = (5) \bmod 11 = 5$$

T[5] is already occupied thus

$$T(59, 2) = (59 \bmod 11 + 2) \bmod 11$$
$$= (4 + 2) \bmod 11 = (6 \bmod 11) = 6$$

×	×	×	×	26	37	59	×	×	×	76
0	1	2	3	4	5	6	7	8	9	10

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

Since T[10] is already occupied so we have

$$h(21, 1) = (21 \bmod 11 + 1) \bmod 11$$
$$= (10 + 1) \bmod 11 = (11 \bmod 11) = 0$$

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

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

Since T[10] is already occupied so we have

$$T(65, 1) = (65 \bmod 11 + 1) \bmod 11$$
$$= (10 + 1) \bmod 11$$
$$= (11 \bmod 11) = 0$$

Since T[0] is already occupied so we have

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

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

$$T(88, 0) = (88 \bmod 11 + 0) \bmod 11$$
$$= (0 + 0) \bmod 11 = 0$$

T[0] is a already occupied so we have

$$T(88, 1) = (88 \bmod 11 + 1) \bmod 11$$
$$= (0 + 1) \bmod 11 = 1 \bmod 11 = 1$$

T[1] is already occupied so we have

$$T(88, 2) = (88 \bmod 11 + 2) \bmod 11$$
$$= (0 + 2) \bmod 11 = (2 \bmod 11) = 2$$

So we have the Final Hash Table :

21	65	88	X	26	37	59	X	X	X	76
0	1	2	3	4	5	6	7	8	9	10

This is the Final Hash Table Allocation with the help of Linear Probing.

Linear Probing is very easy to implement, but it suffers from a problem known as Primary Clustering. Here by a cluster we mean a block of occupied slots and primary clustering refers to many such blocks separated by free slots. Therefore once clusters are formed there are more chances that subsequent inserting will also end up in one of the cluster and thereby increasing the size of cluster. Therefore increasing the number of probes required to find a free slot and hence worsening the performance further.

To avoid the problems of Primary clustering we have Quadratic Probing and Double Hashing.

(b) Quadratic Probing

The quadratic probing uses the following hash function :

$$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$$

For $i = 0, 1, 2, \ldots m - 1$

where m is the size of the Hash Table and

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

is the basic has function and c_1 and $c_2 \neq 0$ are given constants and i is the probe number.

Problem : *Consider inserting the keys 76, 26, 37, 59, 21, 65, 88 into a hash Table of size $m = 11$ using quadratic probing with $c_1 = 1$ and $c_2 = 3$. Further consider that the primary hash function is*

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

Solution :

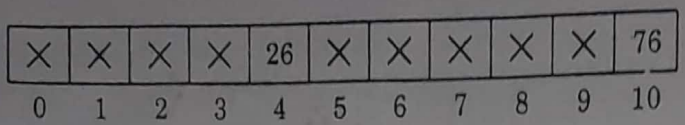
T	–	–	–	–	–	–	–	–	–	–	–
	0	1	2	3	4	5	6	7	8	9	10

Initially, empty hash table

$$T(76, 0) = (k \bmod m + i + 3i^2) \bmod m$$
$$T(76, 0) = (76 \bmod 11 + 0 + 0) \bmod 11$$
$$= (10) \bmod 11 = 10$$

T	X	X	X	X	X	X	X	X	X	X	76
	0	1	2	3	4	5	6	7	8	9	10

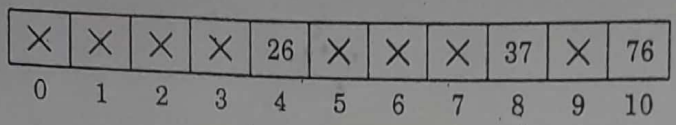
$$T(26, 0) = (22 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$T(26, 0) = (4 + 0 + 0) \bmod 11 = 4 \bmod 11 = 4$$



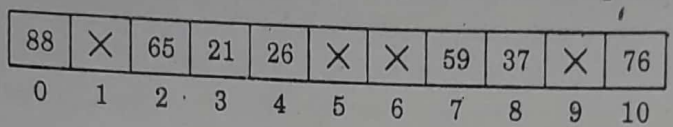
$$T(37, 0) = (37 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$= (4 + 0 + 0) \bmod 11 = 4$$

since T[4] is already occupied so we have

$$T(37, 1) = (37 \bmod 11 + 1 + 3 \times 1^2) \bmod 11$$
$$T(37, 1) = (4 + 1 + 3) \bmod 11 = 8 \bmod 11 = 8$$



This process is going for similarly and finally we get the following hash table.



This is Final Answer with Quadratic Probing.

(c) Double Hashing

Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. Double Hashing uses the hash function of the form

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

for $i = 0, 1, 2, \ldots\ldots m - 1$

where m is the size of the hash table.

$$h_1(k) = k \bmod m$$

and

$$h_2(k) = [k \bmod m']$$

are two Auxiliary hash function, where m' is chosen slightly less than m *i.e.*, m – 1 or m – 2.

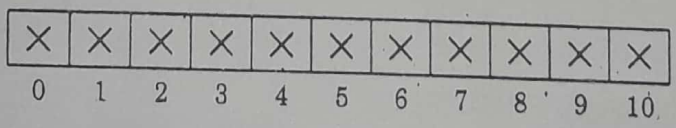
Problem : *Consider inserting the keys 76, 26, 37, 59, 21, 6, 88 into a hash table of size m = 11 using double hashing. Further consider that the Auxiliary hash functions are*

$$h_1(k) = k \bmod 11$$

and

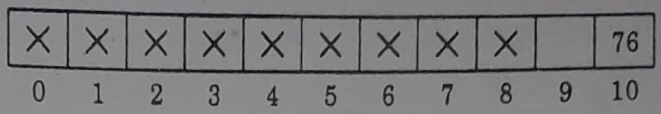
$$h_2(k) = k \bmod 9$$

Solution :



Initially empty hash table

$$h_1(76) = 76 \bmod 11 = 10$$
$$h_2(76) = 76 \bmod 9 = 4$$
$$h(76, 0) = [h_1(k) + ih_2(k)] \bmod m$$
$$h(76, 0) = (10 + 0 \times 4) \bmod 11 = 10 \bmod 11 = 10$$

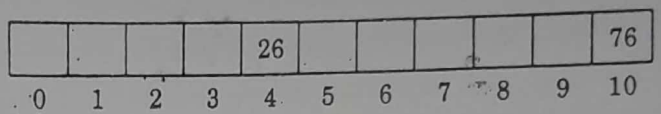


$h_1(26) = 26 \bmod 11 = 4$

$h_1(26) = 26 \bmod 9 = 1$

$h(26, 0) = [h_1(k) + ih_2(k)] \bmod m$

$h(26, 0) = [4 + 0 \times 1] \bmod 11 = 4 \bmod 11 = 4$



$h_1(37) = 37 \bmod 11 = 4$

$h_2(37) = 37 \bmod 9 = 1$

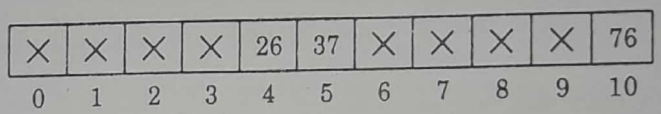
$h(37, 0) = [h_1(k) + ih_2(k)] \bmod m$

$h(37, 0) = [4 + 0 \times 1] \bmod 11 = (4 \bmod 11) = 4$

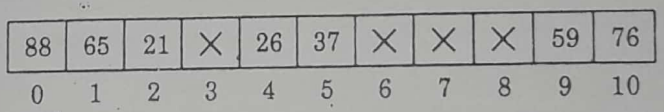
Since T[4] is already occupied so we have

$T(37, 1) = (4 + 1 \times 1) \bmod 11 = 5 \bmod 11 = 5$

So we have



This process is going on and finally we get the Final Hash Table



This is the Final Hash Table Allocates the keys with the help of Double Hashing.

8.8 REHASHING

If at any stage the hash table becomes nearly full, the Running Time for the operations will start taking too much time and even the inset operation may fail for open addressing with quadratic probing. This can happen if there are too many deletions intermixed with too many insertions.

In such a situation, the best possible solution is as stated below :

- Create a new hash table of size double than the original hash Table.
- Scan the original hash table and for each key, compute the new hash value and insert into the new hash table.
- Free the memory occupied by the original hash table.

SUMMARY

- Searching is a method by which, we can search any data item according to our Requirement.
- Binary Search is more efficient the Linear Search, but the Limitation of Binary Search is that, the data items must be sorted in Ascending Order.
- A Hash Table is a Data Structure in which, the location of a Data item is determined by a mediator function, that is known as Hash Function.
- We can resolve the collisions with the Help of two techniques :
  - **A.** Collision Resolution by seperate chaining.
  - **B.** Collision Resolution by open Addressing.
- Hash Function Plays a very important role in Hashing, becuase, without hash function we can't directly Access the Hash Table.

PROBLEMS

1. Differentiate between Linear Search and Binary search according to their advantages and disadvantages.
2. Write a program to input 10 numbers and search any particular number according to Linear search.
3. Write a program to input 10 numbers and search any particular number according to Binary search.
4. What do you mean by hashing ?
5. What are the different method of hashing ?
6. What is linear probing, how it differ from quadratic probing ?
7. What are the limitations of Direct Address Tables ?
8. What is a hash table ? What are the advantages of hash tables over direct address tables ?
9. What is a hash function ? What should be the characteristics of a good hash function ?
10. Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 59, 88 into a hash table with $m = 11$ slots using open addressing with primary hash function h_1 (k) = k mod m. Illustrate the result of inserting of these keys using linear probing using quadratic probing with $c_1 = 1$ and $c_2 = 2$ and using double hashing with h_2 (k) = 1 + (k mod $m-$ 1).