HASHING TECHNIQUES

★ Introduction :- ① Why Hashing.
② Ideal Hashing
③ Modulus Hash function
④ Drawbacks
⑤ Solutions

⇒ Why We need Hashing ?

Actually Hashing is needed for Searching, so for Searching we already has (i) Linear Searching — O(n)
(ii) Binary Searching — O(log n)

Now, Hashing used O(1) time (i.e. less than log n time)

Now, If key are not in any order we will perform Linear Search.

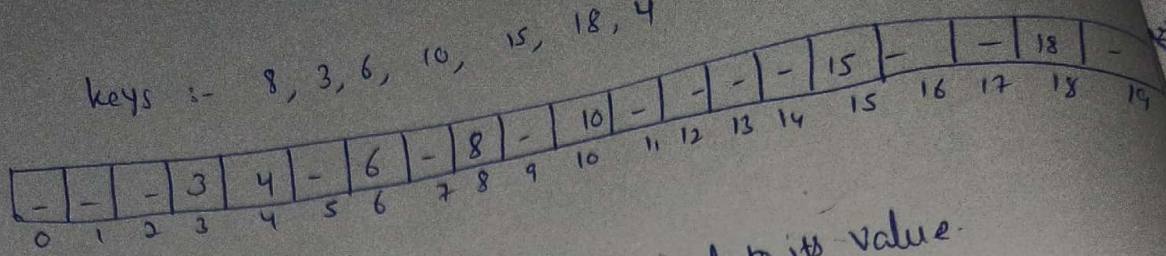
8	3	6	2	9	11	12

Now, If keys are in sorted order, then we will perform Binary Search

2	3	9	10	15	16	19

Now, for Hashing, we have to arrange these keys in a order such that they will find in O(1) time.

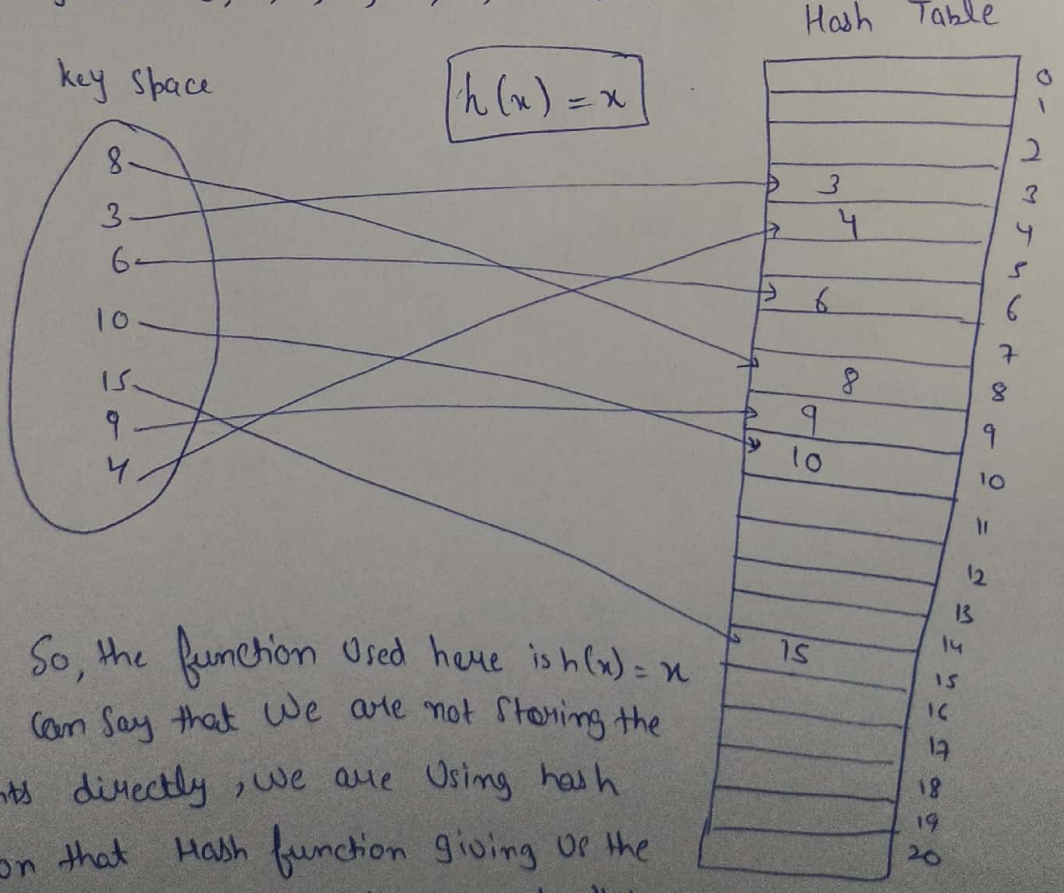
keys :- 8, 3, 6, 10, 15, 18, 4



- keys will be stored at index, equal to its value.
- Rest of the indices will be empty
- So, just go to index and find that element.
- Now, Drawback is that it require so much space as the size of array depends upon largest element in the list.

So, we have to reduce this more memory consumption, so as to learn Hashing.

keys :- 8, 3, 6, 10, 15, 9, 4 (for one to one)



So, the function used here is $h(x) = x$

So, we can say that we are not storing the elements directly, we are using hash function that Hash function giving us the index and we are storing the key at that index given by Hash function

Now, we have to map the keys on hash table, so there are basically 4 types of mapping :-

- functions
  - One - one
  - One - many
  - many - one
  - many - many

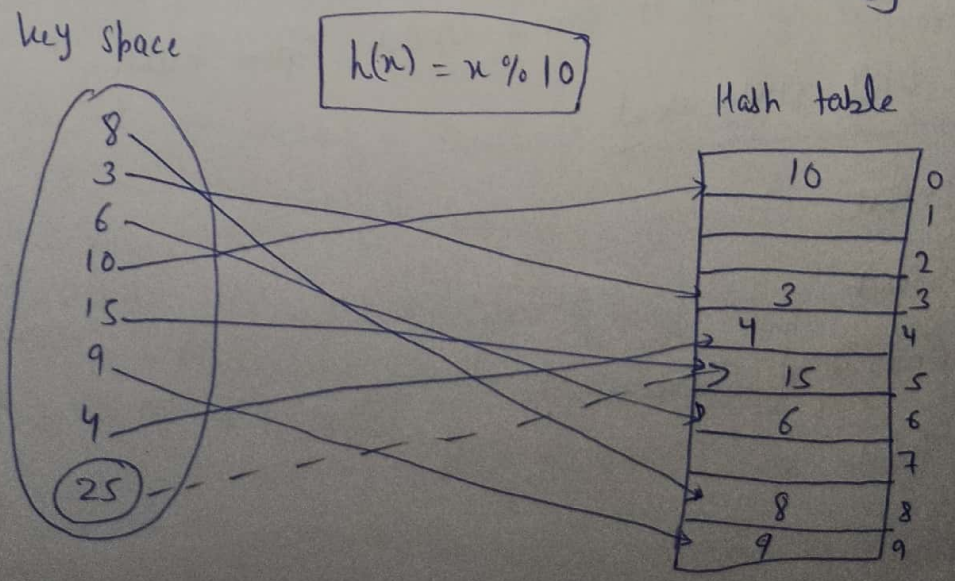
So, Anytime, if we want to search any element, we used hash function. So, when we say function, it supports two mapping :-

- one - one
- many - one

And we called one-one is called Ideal hashing, bcz for inserting Searching, time Complexity is $O(1)$, it means constant time.

- Now its Drawback is, the space required is very large, Suppose if we have a key = 100, then we have to create An array of 100 size.
- But who is responsible for this Drawback? Hash function, bcz Hash function is giving the index
- And if we want to reduce the size, then modify Hash function.

Now, Modify :- $h(x) = x \% 10$ [x % size we want]



Drawback with this Hash function is, Suppose we have keys ... 0, it will mapped to $25\%10=5$, (15) is already present there, two keys will mapped at same place, bcz we modify Hash functi... and when two keys are mapped at same place, we called it Calli... and this type of hash function is many-one (ie $h(x) = x \% 10$)

Solution :- To Resolving this Collision

* Open Hashing → we can use Extra space
  - chaining
* Closed Hashing → we can't use Extra space
  - Open Addressing → if a place given by hash function is already occupied, than we can store it at any other free space, so we will not strictly placed at that space given by hash function, we may store it at any other Address. So that's how it is Open Addressing.
    1. Linear Probing
    2. Quadratic Probing
    3. Double Hashing

* **Chaining** :- To resolve the Collisions, chaining is used and it comes under open hashing.

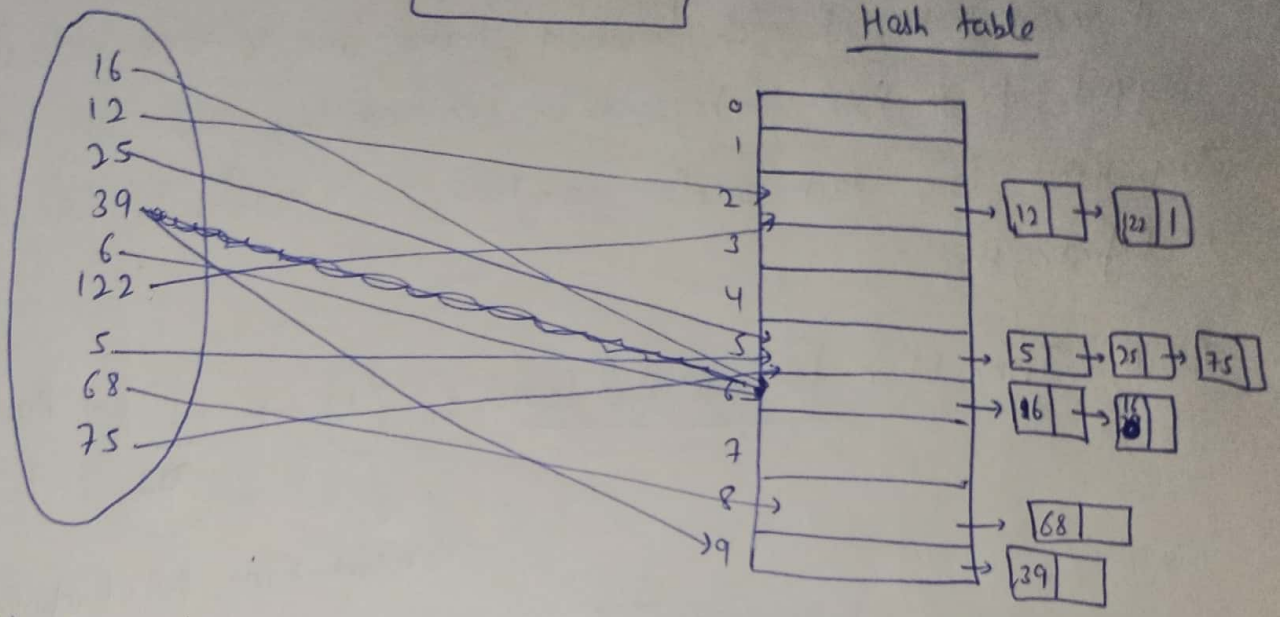
We will study about :-

1. Insert
2. Search
3. Analysis of Search
4. Delete

keys :- 16, 12, 25, 39, 6, 122, 5, 68, 75

key space

$h(x) = x \% 10$



As this is a chaining, so the key will not directly inserted, but a node is made (ie. Linked List) Containing that key. So, it is like a pointer to linked list, so, it is just like Array of pointer to linked list.

Here, we can see that, we have size = 10, but we can store as many elements as we want in Linked list, thats why its called open Hashing.

⇒ Now, for Searching, using Hash function go to that index and Search in Linked List (or in chain),

key = 12, 75 → Succesful Search

key = 65 → unsuccesful Search

⇒ Analysis :- Let, n = 100, no. of keys and size of hash table = 10

Now, we say we can add as many keys as we want, so here, we see a Loading factor, $\lambda = n/size$

Note :- As we do Data structures Earlier, we do analysis on the basis of no. of elements, but here in Hashing Technique, Analysis will be

Here, $\lambda = \frac{100}{10}$

$\boxed{\lambda = 10}$

it means that, at each location, there are 10 keys, So we can Say that, if we have 100 keys, then we can Say that are uniformly distributed, (i.e. each location can have 10-10 keys), we are assuming this as Expecting this.

Now, Time taken for Succesful Search :- (1) + ($\frac{\lambda}{2}$) → Average Comparison

(1) → Constant time for Computing Hash function

So, $\boxed{t_s = 1 + \frac{\lambda}{2}}$

Now, Time taken for unsuccesful Search :- (1) + (λ) → we may be checking Entire Linked List.

(1) → Computing time for Hash function

So, $\boxed{t_{us} = 1 + \lambda}$

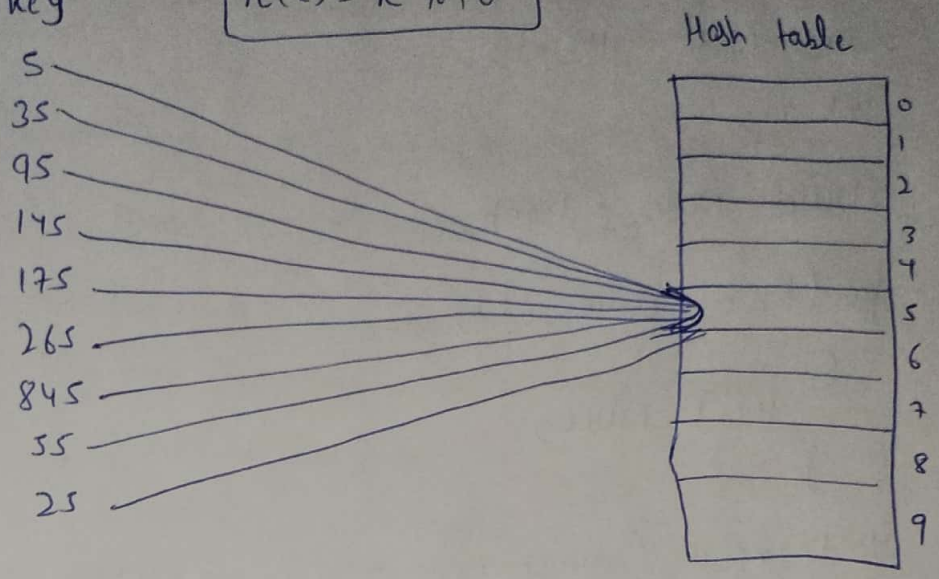
⇒ For Deletion, use Hash function, go to that index, and search for the key and then delete it, if it is found, it is as same as deletion in Linked List. So operations are same as in Linked List.

Note :- Hash function is not particular, it depends upon Programmer and we have to selected a proper hash function to distribute the keys uniformly, and if u dont know, how to Seled Hash function, then you don't know hashing.

"We will learn further about ideal of Hash function".

ppose, key

$h(x) = x \% 10$



So, this shows that $h(x) = x \% 10$ is not a suitable hash function for those set of keys as keys are not distributed uniformly, so it depends upon the programmer, which hash function he/she will use to maitain the concept of Loading factor and while creating Hash function just look at the keys, you want to insert.

★ Code for chaining :-

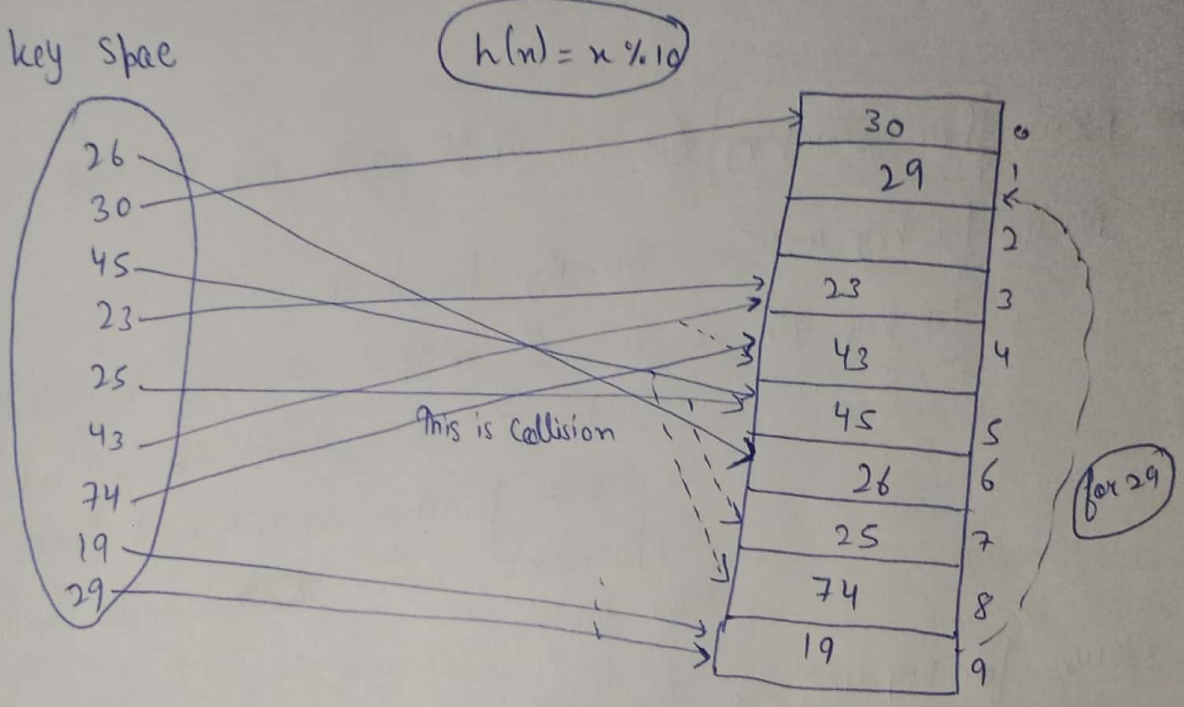
```
Struct Node
{
   int data;
   Struct Node * next;
} * first = NULL;

int hash (int key)
{
   return key % 10;
}
```

```
int main()
{
    struct Node * H[10];
    int i;
    struct Node * temp;
    for (i = 0; i < 10; i++)
    {
        H[i] = NULL;
    }
    Void insert (struct Node * H[], int key);
    insert (h, 12);
    insert (h, 22);
    insert (h, 42);
    Struct Node * Search (struct Node *p, int key);
    int hash (int key);
    temp = Search (h[hash (22)], 22);
    printf ("%d ", temp -> data);
}

Void insert (struct Node * H[], int key)
{
    int hash (int key);
    int index = hash (key);
    Void Sorted insert (struct Node * *p, int x);
    Sorted insert (&h[index], key);
}
```

```
Void Sortedinsert (struct Node **P, int x)
{
    struct Node *t, *q = NULL, *h = *P;
    t = (struct Node *) malloc (sizeof (struct Node));
    t -> data = x;
    t -> next = NULL;
    if (*P == NULL)
        *P = t;
    else
    {
        while (h && h -> data < x)
        {
            q = h;
            h = h -> next;
        }
        if (h == *P)
        {
            t -> next = *P;
            *P = t;
        }
        else
        {
            t -> next = q -> next;
            q -> next = t;
        }
    }
}
```

```
Struct Node * Search (struct Node *P, int key)
{
    Struct Node *q = NULL;
    while (P != NULL)
    {
        if (key == P→ data)
        {
            q→ next = P→ next;
            P→ next = first;
            first = P;
            Return first;
        }
        q = P;
        P = P→ next;
    }
}
```

Linear Probing :- It is a Collision resolution technique and it comes under closed Hashing, so we will use the space of hash table only.



Now, we will use Linear Probing Here, by using modified hash function

$$h'(x) = (h(x) + f(i)) \% 10 \quad \text{where } f(i) = i$$

$i = 0, 1, 2 \ldots$

Now, there is a Collision at index 5, so move to next index, there is also not any free space, then again goto next index and placed (25) there, So we get free space after 2 Probes (ie. after 2 Collisions), since (25) is mapped at index (5) but placed at index, this is known as Open Addressing. Now we will see how hash function used.

for $i=0$ (1) $h'(x) = (h(x) + f(i)) \% 10$

$h'(25) = (h(25) + f(0)) \% 10$

$h'(25) = (5+0) \% 10 = \boxed{5}$

for $i=1$ (2) $h'(25) = (5+1) \% 10 = \boxed{6}$

for $i=2$ (3) $h'(25) = (5+f(2)) \% 10 \Rightarrow (5+2) \% 10 = \boxed{7}$

Now for (i=0) $h'(29) = (h(29) + f(0)) \% 10 = (9+0) \% 10 = 9$

for (i=1) $h'(29) = (h(29) + f(1)) \% 10 = (9+1) \% 10 = 0$

for (i=2) $h'(29) = (h(29) + f(2)) \% 10 = (9+2) \% 10 = 1$

This sh... circular manner ... of %10 at last

Avg. Succes... Avg...

⇒ Now, For Searching also, we will again use this hash function

Now, for key 45 → Directly found at index 5

for key 74 → it will mapped at index 4, but found at index 8, so it is not on 4th index, than we Search further and it will found after 5 Comparisons, hence Linear Probing is little time taking.

Now, for key 40 → This will be unsuccessful Search (if an element is not found, it means that index will be empty, than we can say element is not found.

⇒ Analysis :- Loading factor ⇒ $\lambda = \frac{n}{Size}$

$\lambda = \frac{9}{10} \Rightarrow \lambda = 0.9$

"Loading factor will always less than (1) in Probing bcz, we Can't exceed the size as we done in chaining".

V.V.V. Imp :- $\lambda \leq 0.5$ will always, in Example we have Size = 10 but we inserted 9 elements, but this is wrong, its just for explanation Purpose, Actually no. of elements will always be half of the size so, as we know for unsuccessful Search, time taken can be large to find free space, so to reduce this time taken, $\lambda \leq 0.5$ if there are $\lambda \geq 0.5$

Avg. Successful Search time taken :- $t_s = \frac{1}{\lambda} \ln\left(\frac{1}{1-\lambda}\right)$

Avg. UnSuccessful Search time taken :- $t_{us} = \frac{1}{1-\lambda}$

These are known formulas, we don't have to Analyse them.

★ Drawbacks :- • The first is $\lambda \leq 0.5$, So lots of space will be free.

• The Second is, clusters can be formed, as suppose if we want to search an element and we go to its index but we don't found then we will Compare with next indices which are not even mapped to index (i.e. index resembles searching element) and this is known as clustering. So Linear Probing is Primary clustering.

Imp ⇒ Now, Delete, Go to that index and then Search for that key, if it is found then we can delete it, but we can't afford to have a free space there, bcz if we wanto search another key which is below that index, it will never be found, bcz loop will break after seeing a free space, So, we have to shift, but we actually can't shift also bcz suppose there is a key at index 6, and key at index 7 and we deleted a key at index, but in this case we can't shift, bcz the key at index 7 is mapped there only, So the only thing we can do is to take all the elements and then again insert requried elements but this is so time Consuming and known as Re-Hashing. So this is the reason "We Avoid Deletion in Linear Probing".

★ Code for Linear Probing :-

```
# include <stdio.h>
# define SIZE 10 || SIZE will be a Symbolic Constant that
                    anywhere SIZE appears in this Source file
                    it will be replaced with 10

int main()
{
    int h[10]= {0};
    insert (h, 10);
    insert (h, 25);
    insert (h, 35); || In debug area we can see that 35 will be inserted
    insert (h, 26);     at index(6) bcz no free space availaible at index 5
                    || In debug area we can see that 26 will inserted
                       at index (7) bcz no free space availaible at index 6
printf (" key found at %d", Search (h, 26)); || This will show that 26 is
                                                inserted at index 7
}
int hash (int key)
{
   return key % SIZE;
}
int Probe (int h[], int key)
{
    int index = hash (key);
    int i=0;
    while (h [(index + i) % SIZE] != 0)
            i++;
    return (index + i) % SIZE;
}
```

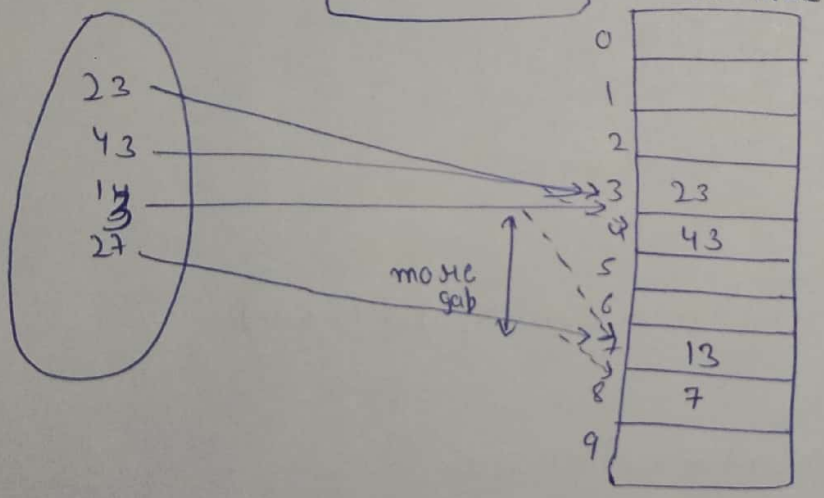
```
void insert (int h[], int key)
{
    int index = hash (key);
    if (h[index] != 0)
    {
        index = Probe (h, key);
    }
    h[index] = key;
}

int Search (int h[], int key)
{
    int index = hash (key);
    int i = 0;
    while ( h[ (index + i) % SIZE] != key)
        i++;
    return (index + i) % SIZE;
}
```

★ **Quadratic Probing** :- It is also a Collision resolution technique and it comes under Open Addressing and Closed Hashing.

We know in Linear Probing there was a problem of clustering (i.e. group of elements made block) So to avoid this we introduce Quadratic Probing.

Key Space

$h(x) = x \% 10$



$h'(43) = (h(43) + f(0)) \% 10$
$= 3$
$h'(43) = (h(43) + f(1)) \% 10$
$= 4$

$h'(13) = (h(13) + f(0)) \% 10$
$= 3$
$h'(13) = (h(13) + f(1)) \% 10$
$= 4$
$h'(13) = (h(13) + f(2)) \%$
$= 3 + 4$
$h'(13) = 7$

Modified hash function :- $h'(x) = (h(x) + f(i)) \% 10$ where $f(i) = i^2$, $i = 0, 1, 2 \ldots$

In Linear Probing, $f(i) = i$
In Quadratic Probing, $f(i) = i^2$ → This is difference.

In Quadratic Probing, we give some more gap for next free space.

⇒ **Analysis** :- Formulas are known, we don't have to Analyse.

Average Successful Search :- $\frac{-\log_e(1-\lambda)}{\lambda}$

Average Unsuccessful Search :- $\frac{1}{1-\lambda}$ e

Code for Quadratic Probing :-

```
# include <stdio.h>
# define SIZE 10

int main()
{
    int h[10]= {0};
    insert (h,23);
    insert (h,43);
    insert(h,13); // We can see in debug area that 13 will be
                  inserted at index 7 bcz no free space is available
                  at index 3 & 4.
    insert(h,27); // We can see in debug area that 27 will be inserted
                  at index 8 bcz no free space is available at
                  index 7

  printf (" key found at "%d", Search (h,13));
}
int hash (int key)
{
   return key % SIZE;
}
int Probe (int h[], int key)
{
    int index = hash(key);
    int i=0;
    while (h[(index + (i*i)) % SIZE] != 0)
        i++;
    return (index + (i*i)) % SIZE;
}
```

```
Void insert (int h[], int key)
{
    int index = hash(key);
    if (h[index] != 0)
    {
        index = Probe(h, key);
    }
    h[index] = key;
}

int Search (int h[], int key)
{
    int index = hash(key);
    int i = 0;
    while(h[(index + (i * i)) % SIZE] != key)
        i++;
    return (index + (i * i)) % SIZE;
}
```

Double Hashing :- It is also a method for resolving Collision and the idea here, we have two Hash functions one is basic hash function and the other is for resolving Collision.

$$h_1(x) = x \% 10$$

$$h_2(x) = R - (x \% R)$$ R → It is a primary no. Smaller than the size of hash table, and in our example

Size of Hash table = 10, So the nearest prime no. is 7

So, R = 7

$$h_2(x) = 7 - (x \% 7)$$

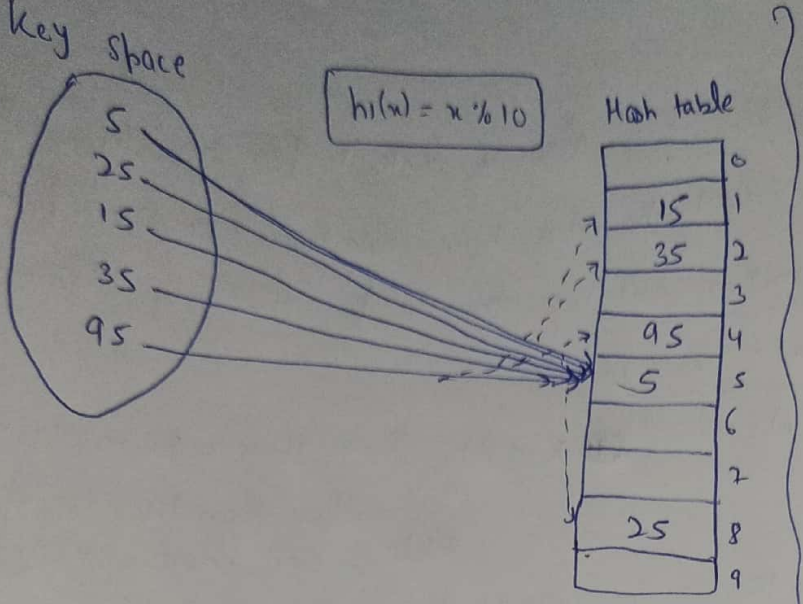
⇒ Actually $h_2(x)$ has some desired properties :-

- it should not give index 0.
- It should try to probe all the locations, it means when there is a Collision, it should not give indices in same pattern as we see in linear & Quadratic probing (i.e in Linear → 1+1+.., in Quad. → 1+4+9--) It give different indices such that all the locations are utilized. Or else we can design any other hash function, but it mandatory having these desired properties.

So, we will use basic hash function $h(x) = x \% 10$

But if there is a Collision, then we will use $h'(x) = (h_1(x) + i * h_2(x)) \% 10$

where, $i = 0, 1, 2, 3 \ldots$



$h'(25) = (h_1(25) + 1 * h_2(25))$

$= (5 + 1 * 3) \% 10$

$7 - (25 \% 7)$

$7 - 4 = 3$

$h'(25) = 8$

$h'(15) = (h_1(15) + 1 * h_2(15)) \% 10$

$= (5 + 1 * 6) \% 10$

$7 - (15 \% 7)$

$7 - 1 = 6$

$h'(15) = 1$

Now, for 35, $h'(35) = (h_1(35) + 1 * h_2(35)) \% 10$

$= (5 + 1 * 7) \% 10$

$7 - (35 \% 7)$

$7 - 0 = 7$

$h'(35) = 2$

Now, for 95, there are more than one Collisions

$i = 1$

$h'(95) = (h_1(95) + 1 * h_2(95)) \% 10$

$= (5 + 1 * 3) \% 10$

$7 - (95 \% 7)$

$7 - 4 = 3$

$h'(95) = 8$

↓

Collision

$i = 2$

$h'(95) = (h_1(95) + 2 * h_2(95)) \% 10$

$= (5 + 2 * 3) \% 10$

$h'(95) = 1$ → Collision

$i = 3$

$h'(95) = (h_1(95) + 3 * h_2(95)) \% 10$

$= (5 + 3 * 3) \% 10$

$h'(95) = 4$

Code for Double Hashing :-

```
# include <stdio.h>
# define SIZE 10
# define SIZE1 7

int main()
{
    int h[10] = {0};
    insert (h, 5);
    insert (h, 25);
    insert (h, 15);
    insert (h, 35);
    insert (h, 95);
    printf (" key found at %d", Search (h, 95)); // this will show that (95) is inserted at index 4 bcz there will be a collisions at indices 5, 8 & 1.
}

int hash
{
    return key % SIZE;
}

int hash1
{
    return SIZE1 - (key % SIZE1);
}

int double hash (int h[], int key)
{
    int index = hash (key);
    int index1 = hash1 (key);
    int i = 0;
```

```
    while (h[(index + (i * index1)) % SIZE] != 0)
        i++;
    return (index + (i * index1)) % SIZE;
}

void insert (int h[], int key)
{
    int index = hash(key);
    if (h[index] != 0)
    {
        index = double hash (h, key);
    }
    h[index] = key;
}

int Search (int h[], int key)
{
    int index = hash(key);
    int index1 = hash1(key);
    int i=0;
    while (h[(index + (i * index)) % SIZE] != key)
        i++;
    return (index + (i * index)) % SIZE;
}
```

Hash - Ideas Functions :- For selecting a hash-function, a programmer or designer must take care that the values in a hash table will be uniformly distributed ;

The thing we have to take care in chaining, hash table size can be anything but in Linear Probing Hash table size must be double than values inserted.

Now, There are three types of Function Ideas :-

① Modulus :- $h(x) = x \% \text{Size}$

↓

it is suggested that size must be as a prime no. to reduce no. of collisions

② Mid Square :- This method suggests that whatever the key, you have to square that key and take out the middle digit

key = 11 → $(11)^2$ → (1 2 1) so 11 will stored at index 1

key = 13 → $(13)^2$ → (1 6 9) so 16 will stored at index 6

if key = 56 → $(56)^2$ → (3 1 3 6) take 1 or 3 any and if suppose index will be 9 but it is not in table, then perform modulus on it

③ Folding :- We are taking digits and make a single no. and this no. will be the index, if it is large then, we can again use modulus

key ⇒ 12 33 47 ⇒ 12
33
47
92 → index

And if key is a string than we can convert it using ASCII Code

key ⇒ "ABC" → 65 66 67 → Add → 198 → if large perform mode.

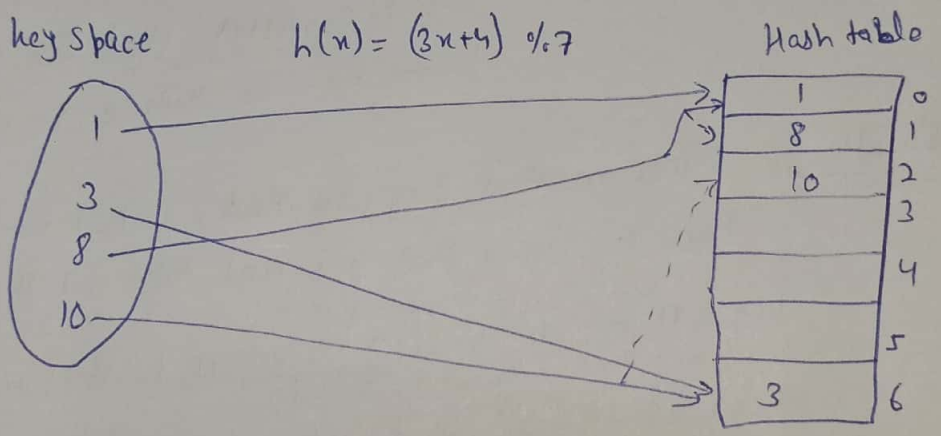
65
66
67
198 → index

Practise

Ques-1 :- Consider the hash table of size seven, with starting index zero, and a hash function $(3x+4)\%7$. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that - denotes empty location in table.

Answer-1

keys ⇒ 1, 3, 8, 10



for k=1,

$h(1) = (3+4)\%7 = 0$

$h(3) = (9+4)\%7 = 13\%7 = 6$

$h(8) = (3(8)+4)\%7 = 0 \rightarrow$ So (8) will go to index 1

$h(10) = (3(10)+4)\%7 = 6 \rightarrow$ So (10) will go to index 2

(a) 8, -, -, -, 10, 1, 3

(b) 1, 8, 10, -, -, -, 3

(c) 1, -, -, -, 3, 8, 10

(d) 1, 10, 8, -, -, 3, -