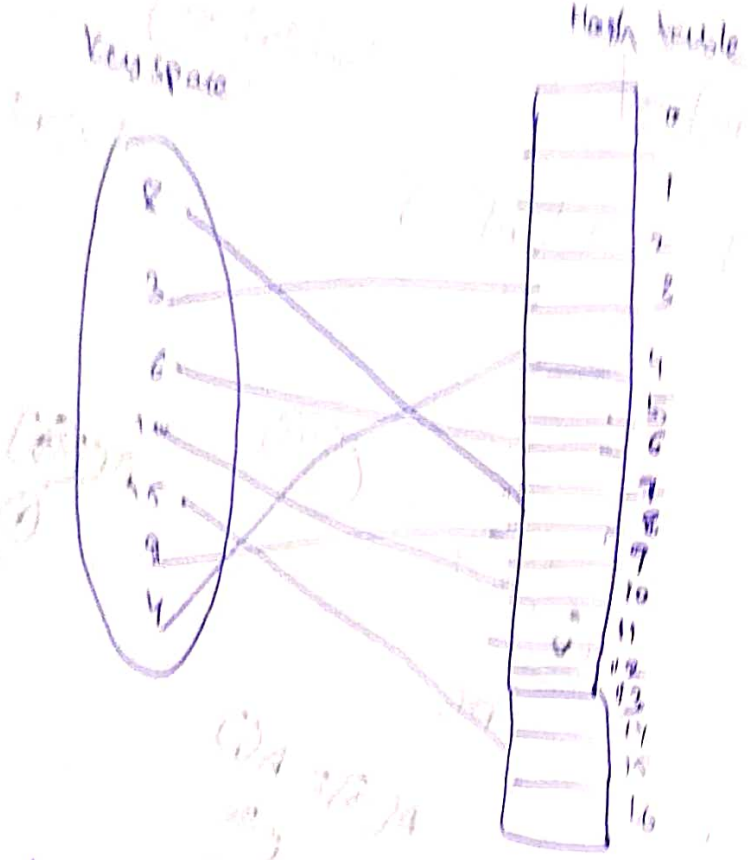
Hashing Technique:-

* Hashing is fastest searching method.

Keys : 8, 3, 6, 10, 15, 9, 4



$h(x) = x$, (Ideal Hash)

Open hashing:-

⇒ Chaining

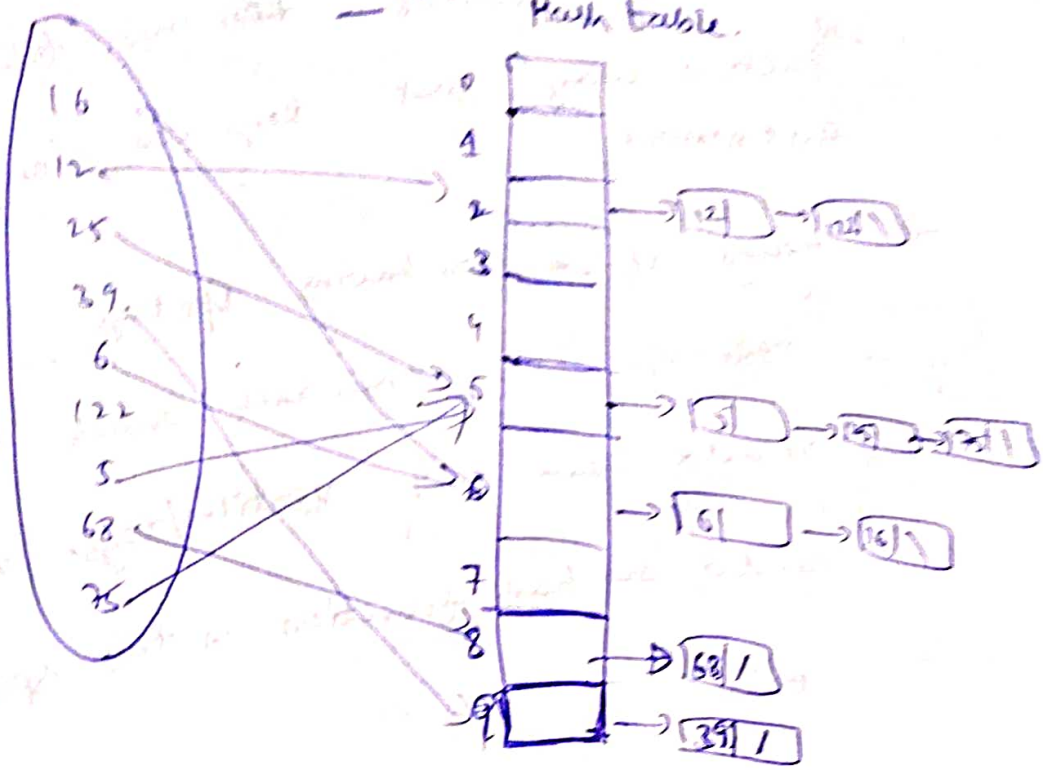
Closed hashing:-

⇒
* Linear Probing
* Quadratic Probing
* Double Hashing

Chaining:-

Keys: 16, 12, 25, 39, 6, 122, 5, 62, 35.

$h(x)_{x\%10}$



→ here our function $h(x) = x \% 10$;

since for 5, 35 we need to store them in index 5 only, so store them in index as linked list.

→ this array is an array of addresses

→

n = 100 (keys); Size = 10;
↓
(Hash table)

$\therefore \alpha = \frac{100}{10} = 10$

↓
Loading factor.

→ time taken for successful search = $1+\frac{\lambda}{2}$ - average

→ here, we need to select hash function in such a way that keys are uniformly distributed.

→ means if we are having 90 keys, size of hash table = 10; then each index must/nearly should have 9 vertices/keys. we need to build our hash function in this specification only

code chaining:-

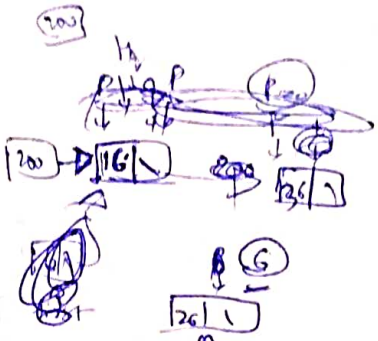
→ struct Node
{
int data;
struct Node * next;
}

p = &H[0]
220

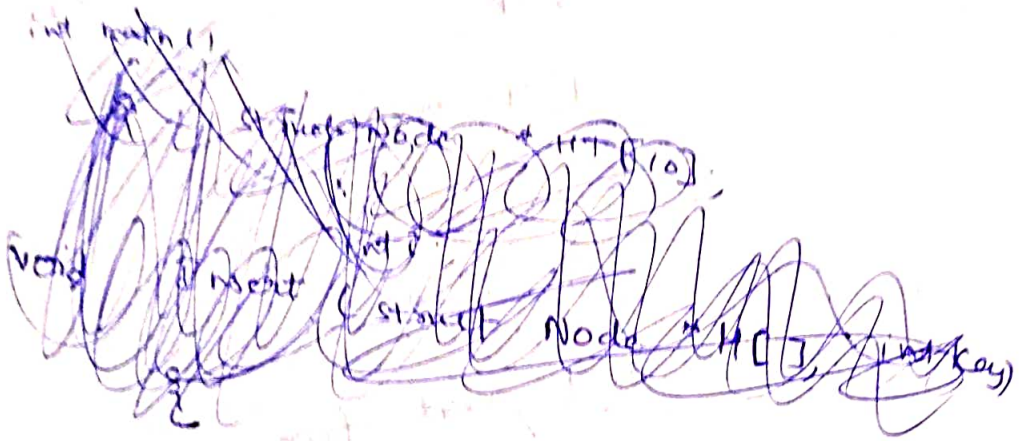
*H = &H[index]

void sorted Insert (struct node **H, int x)
{
struct Node *t, *q = NULL, *p = *H;
*first = *H;

```
t = (struct node *)malloc (sizeof( struct node));
t->data = x;
t->next = NULL;
if ( *H == NULL)
  {
    *H = t;
  }
else
  {
    while ( p != NULL && p->data < x)
      { q = p;
        p = p->next;
      }
    if (p == first)
      { t->next = first;
        first = t;
      }
    else
      { t->next = q->next;
        q->next = t;
      }
  }
}
```



```
struct node * search ( struct node *p, int key)
{
    while ( p != NULL)
    {
        if (key == p->data)
        {
            return p;
        }
        p = p->next;
    }
    return NULL;
}
```

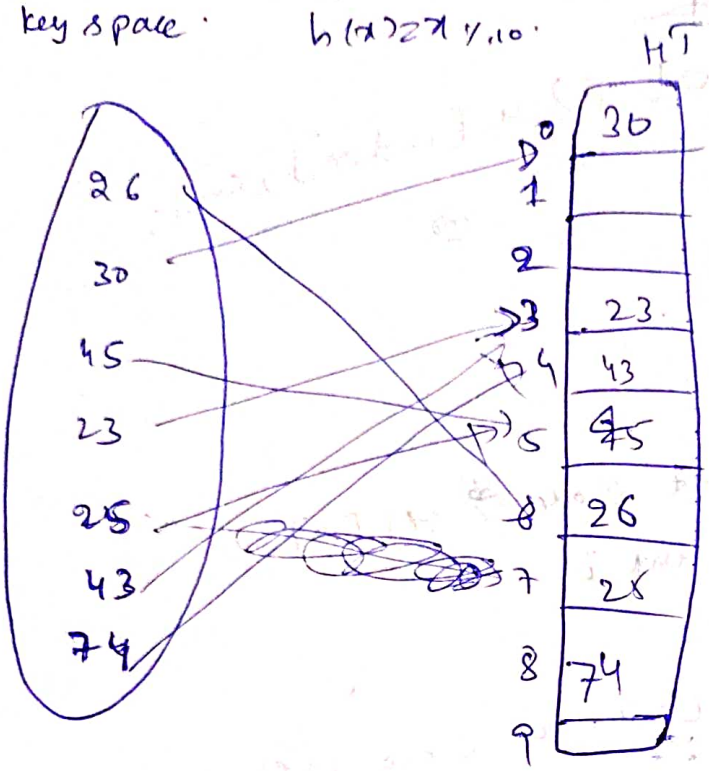


```
int hash( int key)
{
    return key % 10;
}
Void Insert ( struct Node *H[] , int key)
{
```

```
    int index = hash(key);
    H[index]
    SortedInsert(&H[index], key);
}

int main()
{
    struct Node *HT[10];
    int i;

    for(i=0; i<10; i++)
    {
        HT[i] = NULL;
    }

    Insert(HT, 12);
    Insert(HT, 22);
    Insert(HT, 42);
    temp = search(HT[hash(key)], key);
}
```

Linear probing:-

key space: $h(x) = x \% 10$



$h'(x) = (h(x) + f(i)) \% 10$ where $f(i) = i$

→

here 45 is stored at Index '5'.

But our '25' should also be stored at Index '5'. But HT[5] is occupied. so, try to store '25' at the next free index, when there is collision. means we need to store '25' in Index '7'. This is called probing

$h'(a) = (h(a) + f(i)) \% 10$ $\quad f(i) = i$

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

$h'(26) = (26 + f(0)) \% 10 = 6$

so for 26 go to index 6 & store 26.

$\Rightarrow h'(30) = (30 + 0) \% 10 = 0$

$\therefore HT[0] = 30$

$\Rightarrow h'(45) = (45 + f(i)) \% 10 = (45 + f(0)) \% 10$

$= 5$

$HT[5] = 45;$

$\Rightarrow h'(23) = (23 + f(i)) \% 10 = (23 + f(0)) \% 10$

$= 3$

$HT[3] = 23;$

$\Rightarrow h'(25) = (25 + f(i)) \% 10 = 5.$

but in HT[5] we already have 45

so, take $i = 1$

$h'(25) = (25 + f(1)) \% 10 = (25 + 1) \% 10 = 6$

but HT[6] = 26; so take $i = 2$

$h'(25) = (25 + f(2)) \% 10 = 7$

$\therefore HT[7] = 25.$

→ searching :-

while searching go to, an index = key % 10; if we find our key it's successful. otherwise, go on searching from next index until end of hash table.

for eg: let us search 74. so go to 74 % 10 = 4 index 4. HT[4] > 43; so search from index 5, to index 3 until we find it. while going in b/w index 5 to index 3, if we find a gap, then search is unsuccessful.

loading factor $\lambda = \frac{n}{size}$

average successful search = $t = \frac{1}{\lambda} \ln\left(\frac{1}{1-\lambda}\right)$

average unsuccessful search = $t = \frac{1}{1-\lambda}$.

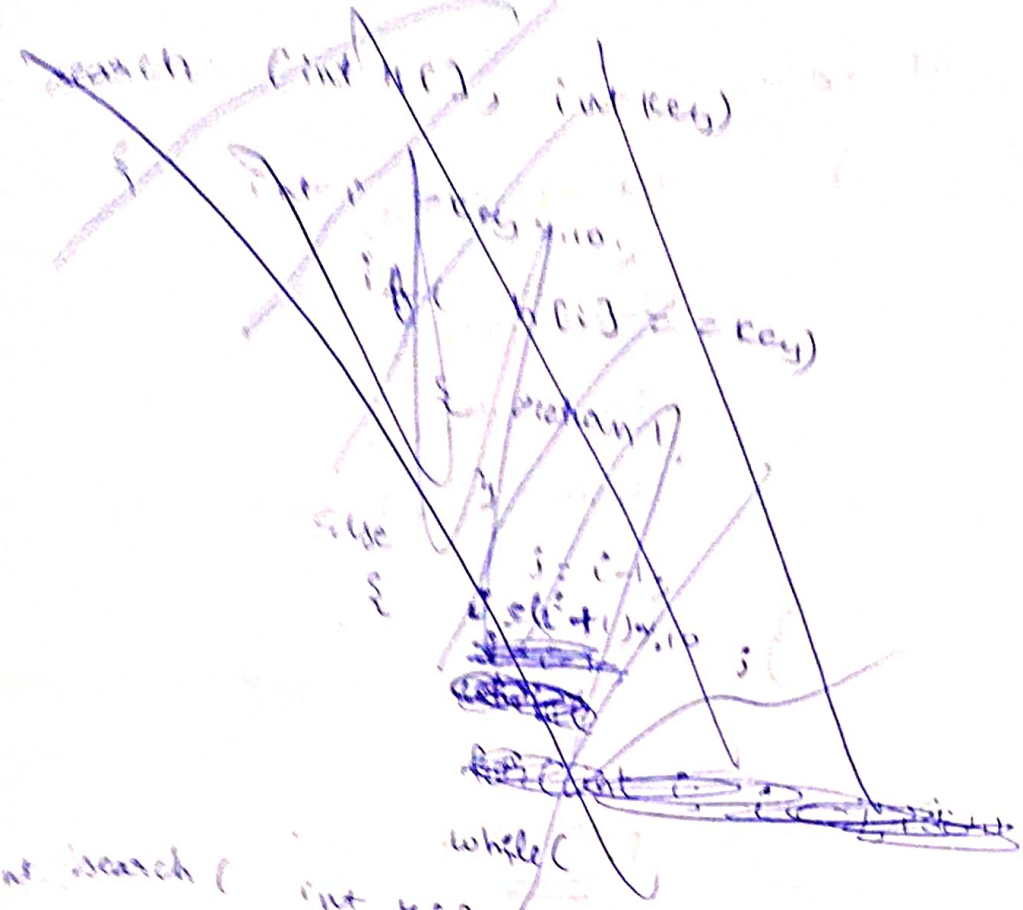
$\lambda \leq 0.5$ (must & should).

deletion is not easy in Linear Probing

code:

```
#include<stdio.h>

void hash(int key)
{
    return key%size;
}

void Insert(int H[], int key)
{
    int index = hash(key)
    if(H[index]!=0)
    {
        index = Probe(H, key);
    }
    H[index]=key;
}

int Probe(int H[], int key)
{
    int x = key%size;
```

```
while ( H[x] != -1)
{
    x = (x+1) % Size;
}
H[x] = key;
return x;
}

int search( int H[], int key)
{
    int i = key % 10;
    if( h[i] == key)
    { return 1;
    }
    else
    { i = (i+1) % 10;
      while( h[i] != -1 && h[i] != key)
      {
          i = (i+1) % 10;
      }
      if (h[i] == -1)
      { return 0;
      }
      else
      { return 1;
      }
    }
}
```



```
int search ( int h[], int key)
{ int i = key % 10;
  if ( h[i] = = key)
    { return i;
    }
  else
    {   i = (i+1) % 10;
        while ( h[i] != -20 && h[i] != key)
          { i = (i+1) % 10;
          }
        if (h[i] = = -20)
          { return 0;
          }
        else
          { return i;
          }
    }
}
```

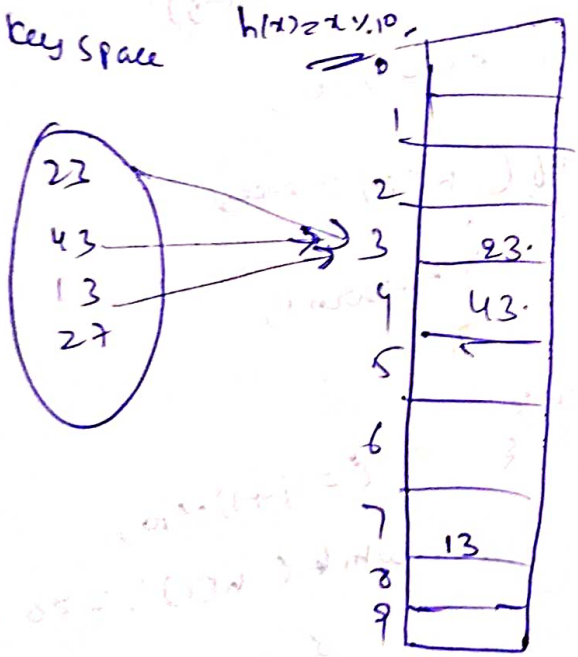
```
int main()
{
    int HT[10]={0};
    Insert(HT,12);
    Insert(HT,25);
    Insert(HT,35);
}
```

Quadratic Probing:-

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

where $f(i) = i^2$

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



$h'(23) = 23 \% 10 = 3$

$h'(43) = (h(43) + f(i)) \% 10$ (i=1)

$= 3$

LH $h'(43) = 4$

$h'(13) = (h(13) + i \cdot f(1)) \% 10$ for $i = 0, 1, 2, 3$.

$= 3$. ✓

$h'(13) = 4$

$h'(13) = 7$

* Double Hashing:-

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

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

$h'(x) = (h_1(x) + i * h_2(x)) \% 10$

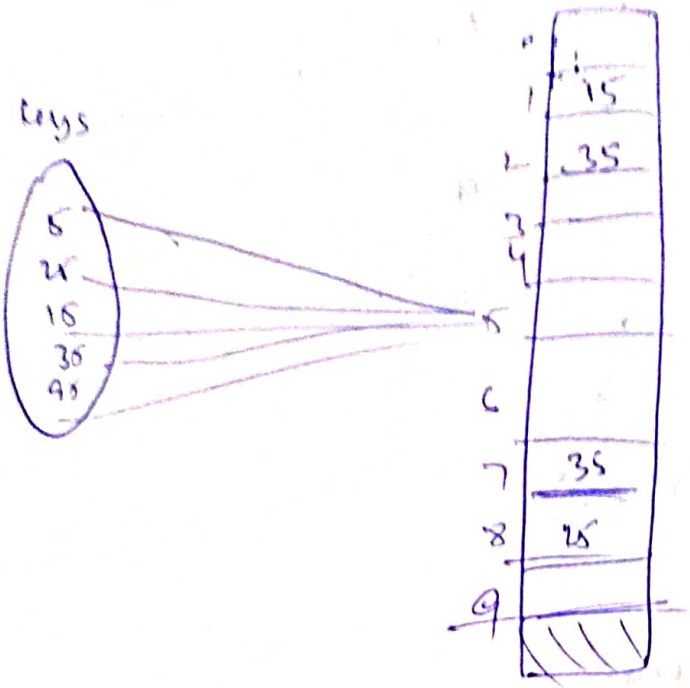
where $i = 0, 1, 2, \ldots$

→ here one function $h_1(x)$ is used to store our Numbers in an array.

→ other function is used when there is a collision.

→ here 'R' is some value less than 'size-1' of array and nearest to 'size-1'.

R = 7



$h'(5) = (h_1(5) + 0 * h_2(5)) \% 10 = 5$

$h'(25) = (h_1(25) + i * h_2(25)) \% 10$ (i=1)

$= (5 + 1 * (7 + (25 \% 7))) \% 10$

$= (5 + 3) \% 10 = 8$

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

$= (5 + 1 (7 - (15 \% 7))) \% 10$

$= 1$

- Math functions:-

h(x) = (x % 2 == 0) ? 1

it is better if we take
large as prime numbers.