HASHING TECHNIQUE

- **Why hashing ?**

There are three methods of searching

1. Linear Search – O(n)
2. Binary Search – O(logn) // But in binary search, elements must be arranged in sorted order, so we have to do some extra work
3. Hashing – O(1)

↳ It is the fastest method, but requires very large space as it depends on maximum element

DRAWBACK

A

0	1	2	3	4	5	6	7	8	9
3	6	8	8	10	12	15	15	15	20

C

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	3	0	0	6	0	8	0	10	0	12	0	0	15	0	0	0	0	20

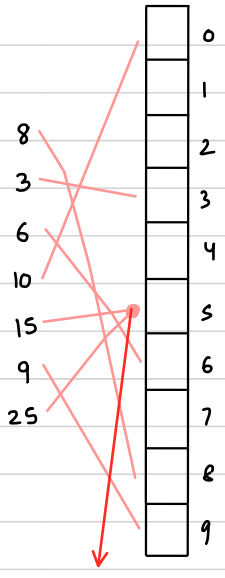
- **Ideal Hashing**

$h(x) = x$ hash function

Mapping

- One to one ✓ func
- One to many
- many to one ✓ func
- many to many

$h(x) = x \% 10$ gives remainder many-one



COLLISION

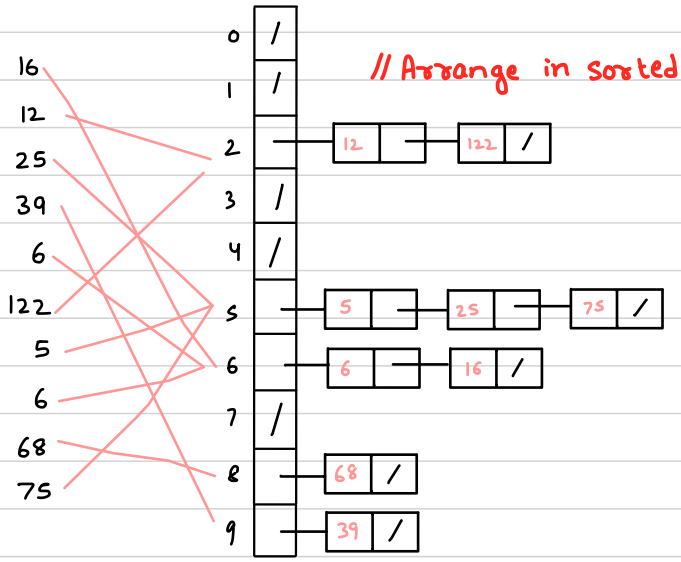
- How to avoid COLLISION ?

Open Hashing // No space restriction
- Chaining

Closed Hashing // Limited Space
- Open Addressing // If collision, store element elsewhere in free space
  1. Linear Probing
  2. Quadratic Probing
  3. Double Hashing

  ↖ Methods for storing in free space

- CHAINING



// Arrange in sorted order

Arrange on basis of first digit

$h(x) = x \% 10$ ← Hash Function

$n = 100$ No of keys

$size = 10$ No of index

$\lambda = \frac{n}{size}$ ← LOADING FACTOR

Average Successful Search

$t = 1 + \frac{\lambda}{2}$

Average Unsuccessful Search

$t = 1 + \lambda$

If keys are : 5, 35, 95, 145, 175, 265, 845

Then modify your hash function

PROGRAM

```
Void Insert ( struct Node *H[], int key)
{
        int index = hash(key);
        SortedInsert ( &H[index], key);
}

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

void SortedInsert( struct Node **H, int x)
{
      Struct Node *t, *q = NULL, *p = *H;
      t = (Struct Node *) malloc ( sizeof (Struct Node));
      t -> data = x;
      t -> next = NULL;

      if ( *H = NULL)
            *H = t;
      else
      {
            while ( p && p -> data < x)
            {
                  q = p;
                  p = p -> next;
            }

            if ( p == *H)
            {
                  t -> next = *H;
                  *H = t;
            }
            else
            {
                  t -> next = q -> next;
                  q -> next = t;
            }
      }
}
```

```
Struct Node
{
      int data;
      Struct Node *next;
};
```

```
void main()
{
        struct Node * HashTable[10], *temp;
        int i;

        for (i=0; i<10; i++)
                HashTable[i]= NULL;

        Insert (HashTable, 12);
        Insert (HashTable, 22);
        Insert (HashTable, 42);

        temp = Search (HashTable[hash(21)], 21);
}


Struct Node * Search (struct Node *p, int key)
{
      while (p != NULL)
      {
            if (key == p→data)
                  return p;
            else
                  p = p→next;
      }
      return NULL;
}
```

LINEAR PROBING

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

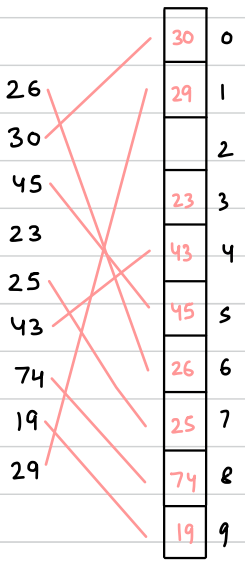
$h'(x) = (h(x) + f(i)) \,\%\, 10$ // For probing

$f(i) = i$

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

↓

In case of collision, insert at next free space



$h'(25) = (h(25) + f(0)) \,\%\, 10$
$= (5 + 0) \,\%\, 10 = 5$

$h'(25) = (h(25) + f(1)) \,\%\, 10$
$= (5 + 1) \,\%\, 10 = 6$

$h'(25) = (h(25) + f(2)) \,\%\, 10$
$= (5 + 2) \,\%\, 10 = 7$

$h'(29) = (h(29) + f(0)) \,\%\, 10$
$= (9 + 0) \,\%\, 10 = 9$
$= (9 + 1) \,\%\, 10 = 0$
$= (9 + 2) \,\%\, 10 = 1$

Cyclic behaviour

For searching:

Key : 45 ✓ — $45 \,\%\, 10 = 5$ — found at index 5

Key : 74 ✓ — $74 \,\%\, 10 = 4$
- Not found at index 4
- Search Next index until 74 is found or blank space is found
- Found at index 8

// Search takes more than constant time

Key : 40 ✗ — $40 \,\%\, 10 = 0$
- Not found at index 0
- At index 2, blank space found
- Element does not exist

ANALYSIS

$\lambda = \frac{n}{size} = \frac{9}{10} = 0.9$

DRAWBACK

** $\lambda \leq 0.5$ // Table should be atmost half filled so, there are blank spaces and searching can be made faster

Average Successful Search

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

Average Unsuccessful Search

$t = \frac{1}{1-\lambda}$

DELETION IS NOT EASY IN LINEAR PROBING

PROGRAM

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

int probe(int H[], int key)
{
    int index = hash(key);
    int i = 0;

    while (H[(index + i) % 10] != 0)
        i++;
    return (index + i) % 10;
}

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) % 10] != key)
        i++;
    return (index + i) % 10;
}
```

```
void main()
{
    struct Node * HashTable[10], *temp;
    int i;

    for (i = 0; i < 10; i++)
        HashTable[i] = NULL;

    Insert(HashTable, 12);
    Insert(HashTable, 22);
    Insert(HashTable, 42);

    temp = Search(HashTable, 35);
}
```

QUADRATIC PROBING

The drawback of linear probing is that elements cluster together and form a group. To resolve this issue, quadratic probing is introduced.

$h'(x) = (h(x) + f(i)) \% 10$ where $f(i) = i^2$

$i = 0, 1, 2 \ldots$

$h'(23) = (h(13) + f(0)) \% 10$

$= (3 + 0) \% 10 = 3$

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

$= (3 + 0) \% 10 = 3$

$= (3 + 1) \% 10 = 4$

$h'(13) = (h(13) + f(2)) \% 10$

$= (3 + 4) \% 10 = 7$

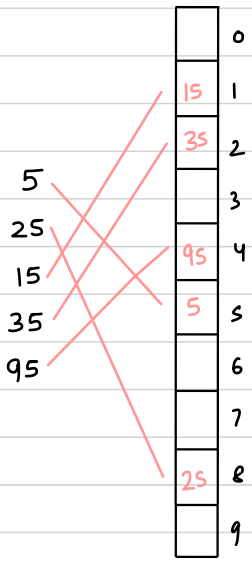
Average Successful Search

$$\frac{-\log_e(1-\lambda)}{\lambda}$$

Average Unsuccessful Search

$$\frac{1}{1-\lambda}$$

DOUBLE HASHING



$h1(x) = x \% 10$

$h2(x) = R - (x \% R)$

$h'(x) = (h1(x) + i * h2(x)) \% 10$

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

where R is prime num smaller than size of Hash Table

$h'(25) = (5 + 1 * 3) \% 10 = 8$

$7 - (25 \% 7)$

$7 - 4 = 3$

$h'(15) = (5 + 1 * 6) \% 10 = 1$

$7 - (15 \% 7)$

$7 - 1 = 6$

$h'(35) = (5 + 1 * 7) \% 10 = 2$

$7 - (35 \% 7)$

$7 - 0 = 7$

$h'(95) = (5 + 3 * 3) \% 10 = 4$

$7 - (95 \% 7)$

$7 - 4 = 3$