Unit 3

Sorting and Searching

3.1 Concept of sorting:

Definition: Sorting is systematic arrangement of data

Sorting are of two types.

1) Internal sorting
2) External sorting

3.1.1 Internal Sorting

Internal sorting is a sorting in which the data resides in the main memory of the computer.

* Various methods of internal sorting are

1) Bubble sort
2) Insertion sort
3) Selection sort
4) Quick sort
5) Radix sort. and so on.

External sorting:

For many applications its not possible to store the entire data on main memory for two

reasons i) amount of main memory available is smaller than amount of data.

ii) The main memory is a volatile device & thus will lost the data when power is shut down

* To overcome these problems the data is stored on the secondary storage devices.

* The technique which is used to sort the data which resides on the secondary storage device are called external sorting.

3.1.2 Sort order:

The sorting is a technique by which we expect the list of elements to be arranged as we expect.

* Sorting order is nothing but the arrangement of the elements in some specific manner.

* Usually its of two types.

i) Ascending order

ii) Decending order.

Ascending order : The elements are arranged from low value to high value.

* In other words elements are in increasing order.

eg:- 10, 50, 40, 20, 30

can be arranged in ascending order after applying some Sorting technique as

10, 20, 30, 40, 50.

Descending order: The elements are arranged from high value to low value.

* In other words elements are in decreasing order.
* Its a reverse of ascending order.

eg:- 10, 50, 40, 20, 30

can be arranged in descending order after applying some sorting technique as

50, 40, 30, 20, 10.

3.1.3. Sort stability

The sorting stability means comparing the records of same value and expecting them in the same order even after sorting them.

eg:- (Pune, Balgandharva)
(Pune, Shaniwarwada)
(Nasik, Panchavati)
(mumbai, Grateway of India)

* The ascending order for the alphabets will be M, N, P.

(mumbai, Grateway of India)
(Nasik, Panchavati)
(Pune, Balgandharva)
(Pune, Shaniwarwada)

3.14. Efficiency and Passes:

One of the major issue in sorting algorithm is its efficiency.

* we denote the efficiency of sorting algorithm in terms of time complexity.
* Time complexity is given in terms of big-O notation
* There are $O(n^2)$ & $O(n \log n)$ time complexities.
* Bubble sort, insertion sort, selection, shell sort has time complexity $O(n^2)$.
* Merge sort, quick sort has time complexity $O(n \log$
* Quick sort is fastest algorithm
* Bubble sort is slowest algorithm

Passes: The phases in which the elements are move to acquire their proper position is called passes.

eg:- 10, 30, 20, 50, 40

Pass 1 : 10, 20, 30, 50, 40

Pass 2 : 10, 20, 30, 40, 50.

Sorting techniques :

Sorting is an important activity and every time we insert or delete the data we need to sort the remaining * various alg are developed for ~~sorting~~ sorting such as.

1. Bubble Sort
2. Selection Sort
3. Insertion sort
4. Radix sort
5. Shell Sort
6. Merge Sort
7. Quick sort.

3.2 Bubble sort

This is the simplest kind of sorting method * we do this bubble sort procedure in several iteration which is called Passes.

Algorithm :

1. Read the total number of elements say n
2. store the elements in the array.
3. set the i=0.
4. compare the adjacent elements.

5. Repeat step 4 for all n elements.
6. Increment the value of i by 1 and repeat step 4,5 for $i < n$
7. Print the sorted list of elements.
8. Stop.

eg:- consider 5 unsorted elements are.

45, -40, 190, 99, 11.

* first store those elements in an array a.

a

0	45
1	-40
2	190
3	99
4	11

Pass 1

* compare each element with neighbouring element.
* Compare 45 and -40
* since 45 > -40 interchange.

0	-40
1	45
2	190
3	99
4	11

* Compare 45 and 190 that is a[1] and a[2]
* since 45 is smaller no interchange.

1

0	-40
1	45
2	190
3	90
4	11

* compare 190 & 90 that is a[2] & a[3].
* Since 190 is greater, interchange 190 & 90.

0	-40
1	45
2	90
3	190
4	11

* Compare 190 & 11.
  Since 190 > 11, interchange.

0	-40
1	45
2	90
3	11
4	190

After first pass the numbers are not sorted.
So we go for pass 2.

Pass 2

* compare -40 & 45, no interchange.
* compare 45 & 90, no interchange.
* compare 90 & 11,
  Since 90 > 11, interchange.

0	-40
1	45
2	11
3	90
4	190

* Compare 90 & 190, no interchange.

* After 2nd Pass the numbers are not sorted so go for Pass 3.

Pass 3

* Compare a[0] & a[1], no interchange.
  -40 & 45

* Compare a[1] & a[2], interchange.
  45 > 11

-40
11
45
99
190

* Compare 45 & 99, no interchange.
* Compare 99 & 190, no interchange.

* Finally the last pass the array will hold all the sorted elements -40, 11, 45, 99, 190.

Python Program:

```
def Bubble (arr, n):
    i = 0
    for i in range (n-1):
        for j in range (0, n-i-1):
```

```
if (arr[j] > arr[j+1]):
    temp = arr[j]
    arr[j] = arr[j+1]
    arr[j+1] = temp
    Print("\n Pass#", (i+1))
    Print (arr)
print ("\nProgram for bubble sort")
print ("\n How many elements are there in array?")
n = int(input())
array = []
i = 0
for i in range (n):
print("\n Enter element in array")
item = int(input())
array.append(item)
print ("original array is \n")
print (array)
print("\n Sorted array is")
Bubble (array, n)
```

Output:

Program for bubble sort.
How many elements are there in array?
5
Enter elements in array.
30

```
Enter elements in array
10
Enter elements in array
20
Enter elements in array
50
Enter elements in array
40
Original array is
[30,10,20,50,40]
Sorted array is
Pass # 1
[10,20,30,40,50]
Pass # 2
[10,20,30,40,50]
Pass # 3
[10,20,30,40,50]
Pass # 4
[10,20,30,40,50]
```

3.3. Selection Sort:

* Scan the array to find its smallest element & swap it with the first element.
* Then starting with the second element scan the entire list to find the smallest element and swap it with the second element.
* Then starting from the third element the entire list is scanned in order to find the next smallest element

* continuing this fashion we can sort the entire list
* Generally, on pass i ($0 \le i \le n-2$) the smallest element is searched among last $(n-1)$ element & is swapped with A[i]

$A[0] \le A[1] \le \cdots \le A[i-1] \mid A[i], \ldots A[k] \ldots A[n-1]$

A[k] is smallest so swap a[i] & a[k]

The list get sorted after n-1 Pass.

eg:-
consider the element 70, 30, 20, 50, 60, 10, 40.
we can store these elements in array A as:

70	30	20	50	60	10	40
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

1st pass:

70	30	20	50	60	10	40

i

scan for finding smallest element.

70	30	20	50	60	10	40

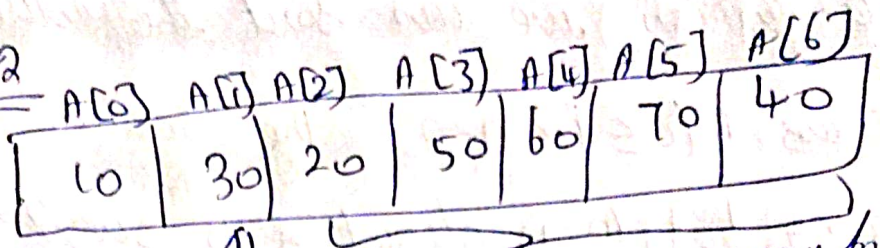
i

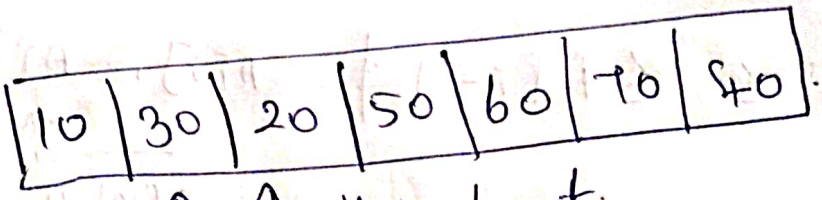
Smallest element found.

Now swap A[i] with smallest element. we get.

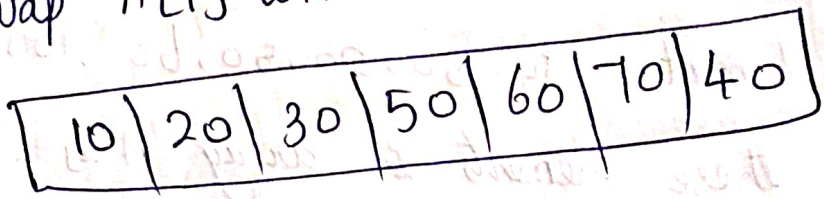
10	30	20	50	60	70	40

Pass 2

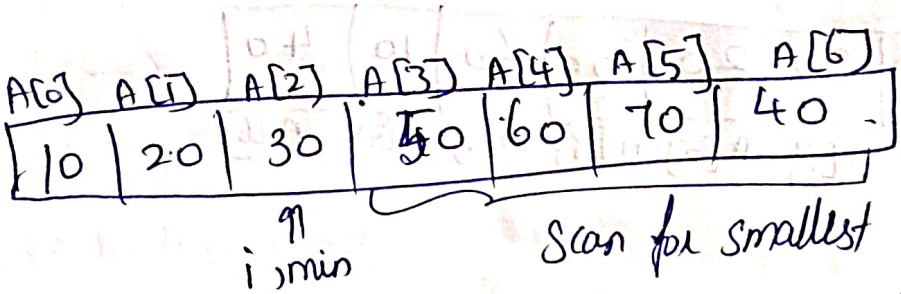




Swap A[i] with smallest element, the array becom

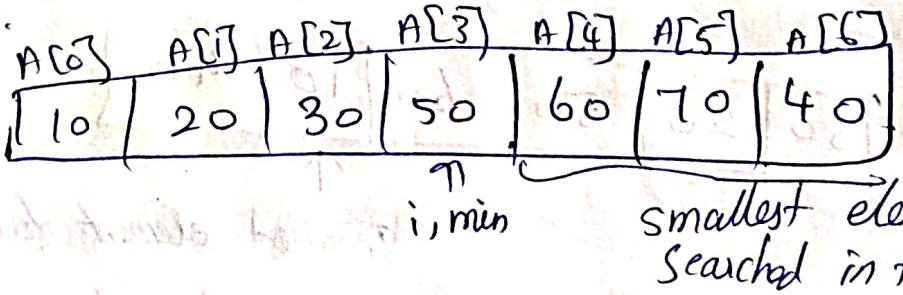


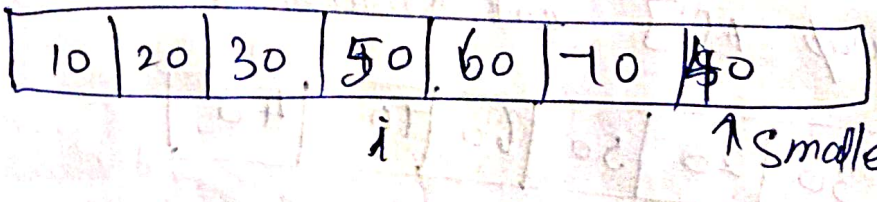
Pass 3



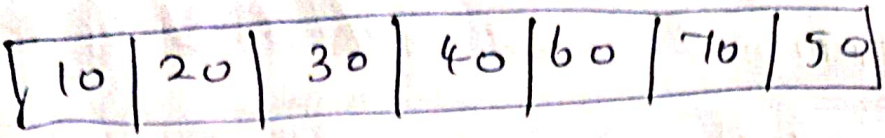
As there is no smallest element than 30 we will increment i pointer.

Pass 4

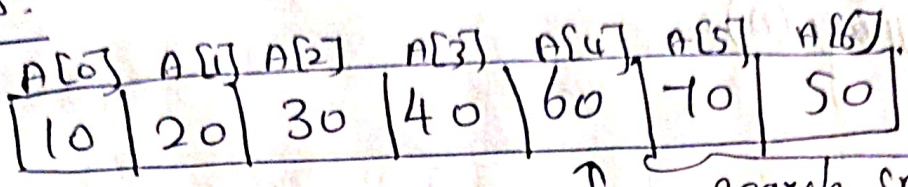


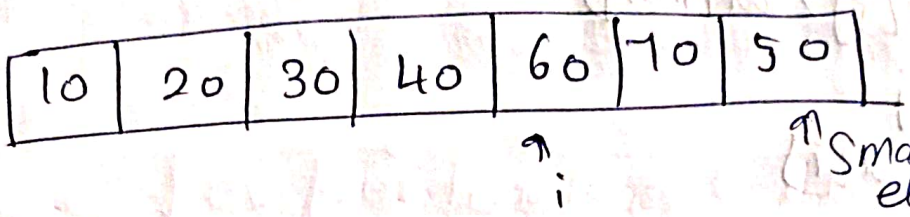


Swap A[i] with Smallest element, the array becomes

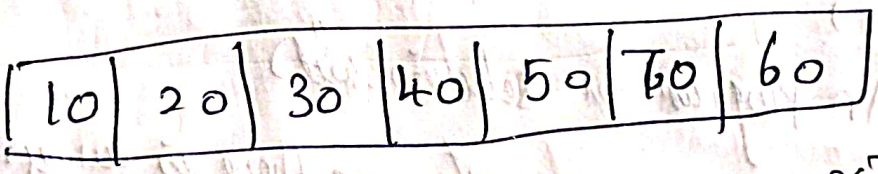


5th Pass:

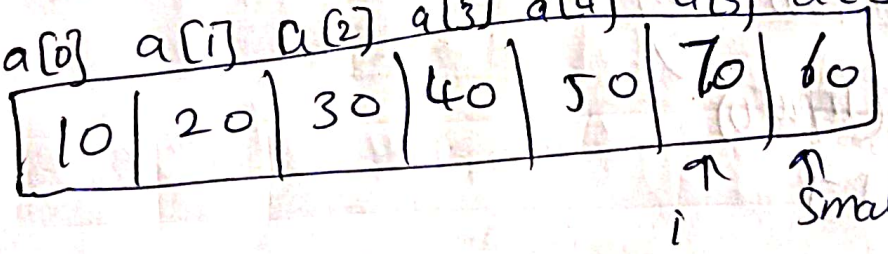




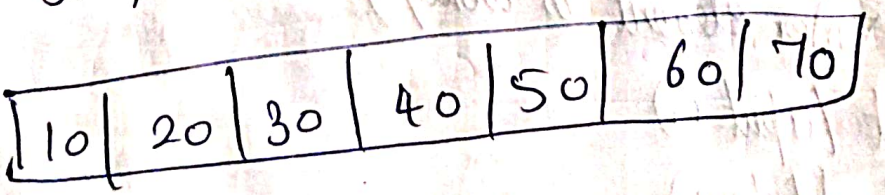
Swap A[i] with smallest element. The array becomes.



Pass 6. a[0] a[1] a[2] a[3] a[4] a[5] a[6].



Swap A[i] with smallest element, the array becomes



This is a sorted array.

Python program

```
def SelectionSort (arr, n):
 for i in range (n):
  Min = i
  for j in range (i+1, n):
  if (arr [j] < arr [min]):
   min = j
   temp = arr [i]
   arr [i] = arr [min]
   arr [min] = temp
   print (arr)
   print ("\n Program for selection sort")
   print ("\n How many elements are there in array?")
   n = int (input ())
   array = []
   i = 0
   for i in range (n):
   print ("\n Enter element in array")
   item = int (input ())
   array . append (item)
  print ("original array is \n")
  print (array)
  print ("\n sorted array is")
  Selectionsort (array, n)
```

o/p

Program for selection sort

How many elements are there in array ?

5

Enter element in array

30

Enter element in array

10

Enter element in array

20

Enter element in array

40

original array is

[30, 50, 10, 20, 40]

sorted array is

[10, 20, 30, 40, 50]

Insertion sort:

In this method the elements are inserted at their appropriate place.

* Hence is the name insertion sort.

eg: Consider the element as

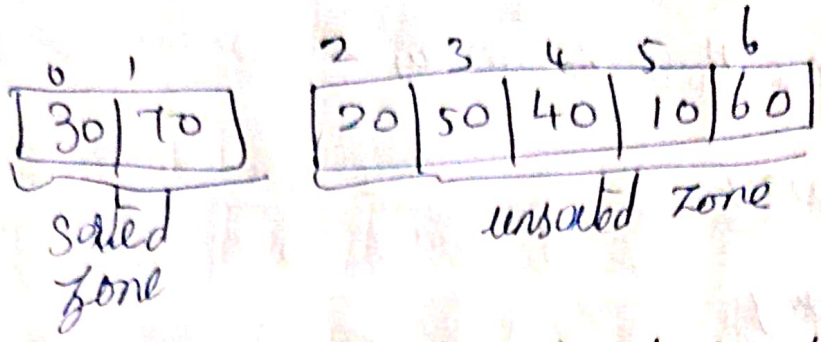
0	1	2	3	4	5	6
30	70	20	50	40	10	60

The process starts with first element

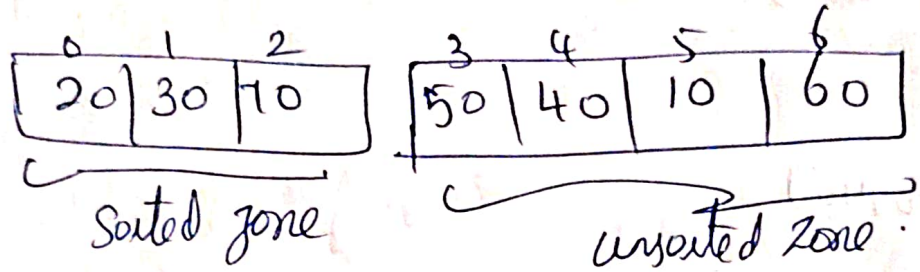
0						
30	70	20	50	40	10	60

sorted zone — unsorted zone

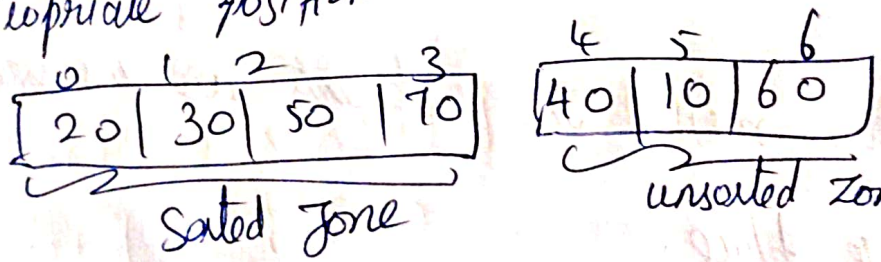
Compare 70 with 30 and insert it at its position.

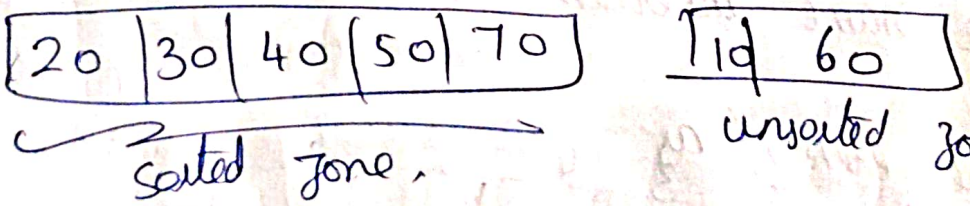


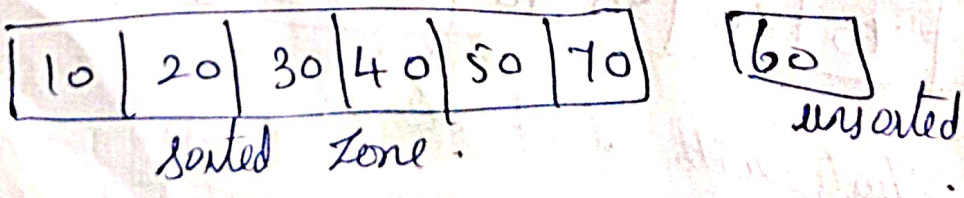
Compare 20 with the elements in sorted zone and insert it in that zone appropriate position.

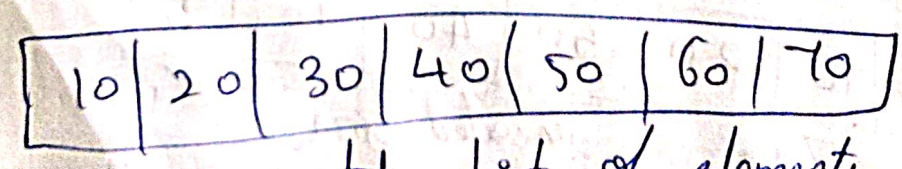


Compare 50 with the elements in sorted zone & insert in appropriate position.









Algorithm

Although it is very natural to impl insertion using recursive alg but it is very efficient to impl it using bottom up approach.

Algorithm insert_sort (A[0...n-1])

```
for i ← 1 to n-1 do.
{   temp ← A[i].
    j ← i - 1
    while (j >=0) AND (A[j] > temp) do
    {
      A [j+1] ← A [j]
      j ← j - 1
    }
    A [j+1] ← temp
}
```

Analysis:

when an array of element is almost sorted then it is best case complexity.

* The best case time complexity of insertion sort is O(n)

Advantage of insertion sort:

1) Simple to implement.
2) Efficient when we want to sort small number of element.

3) more efficient than most other simple $O(n^2)$ alg such as selection sort or bubble sort

4) This is stable

5) It is called in-place sorting alg. In-place sorting alg is an alg in which the i/p is overwritten by o/p and to execute the sorting method it does not require any more additional space.

Program:

```
def Insertsort (arr, n):
i = 1
for i in range (n):
temp = arr [i]
j = i-1
while ((j >= 0) & (arr [j] > temp)):
arr [j+1] = arr [j]
j = j-1
arr [j+1] = temp
print (arr)
print ("\n program for Insertion Sort")
print ("\n Program How many elements are there in array?")
n = int (input ())
array = []
i = 0
for i in range (n):
print ("\n Enter element in array")
```

```
item = int(input())
array.append(item)
print("original array is \n")
print(array)
print("\n Sorted array is")
InsertSort(array,n)
```

o/p

```
Program for Insertion sort
How many elements are there in array?
5
Enter element in array
30
Enter element in array
10
Enter element in array
50
Enter element in array
40
Enter element in array
20
original array is
[30, 10, 50, 40, 20]
Sorted array is
[10, 20, 30, 40, 50]
```

3-5 Merge sort:-

Merge sort is a sorting alg in which array is divived repeatedly. The sub arrays are sorted independently and then these subarrays are combined together to form a final sorted list.

Merge sort on an input array with n elements consists of three steps.

Divide: Partition array into two sub lists S1 & S2 with $\frac{n}{2}$ elements each.

Conquer: Then sort sub list S1 & sublist S2.

Combine: merge S1 & S2 into a unique sorted group.

Program:

Divide the lists into two sublists.

```
def merge_sort (A, l, r):
  if l >= r:
    return
  mid = (l+r)//2
  merge_Sort (A, l, mid)
  merge_Sort (A, mid +1, r)
  merge (A, l, r, mid)
# merge the list
def merge (A, l, r, mid):
```

```
# creating left & right sublists
Left = A[l: mid+1]
Right = A[mid+1: r+1]
i = 0
j = 0
k = l
while i < len(left) and j < len(Right):
if Left[i] <= Right[j]:
A[k] = Left[i]
i = i+1
else:
A[k] = Right[j]
j = j+1
k = k+1
while i < len(left):
A[k] = Left[i]
i = i+1
k = k+1
while j < len(Right):
A[k] = Right[j]
j = j+1
k = k+1
# Driver code
print("\n It Program for merge sort")
print("How many elements you want to sort?")
n = int(input())
A = []
i = 0
for i in range(n):
```

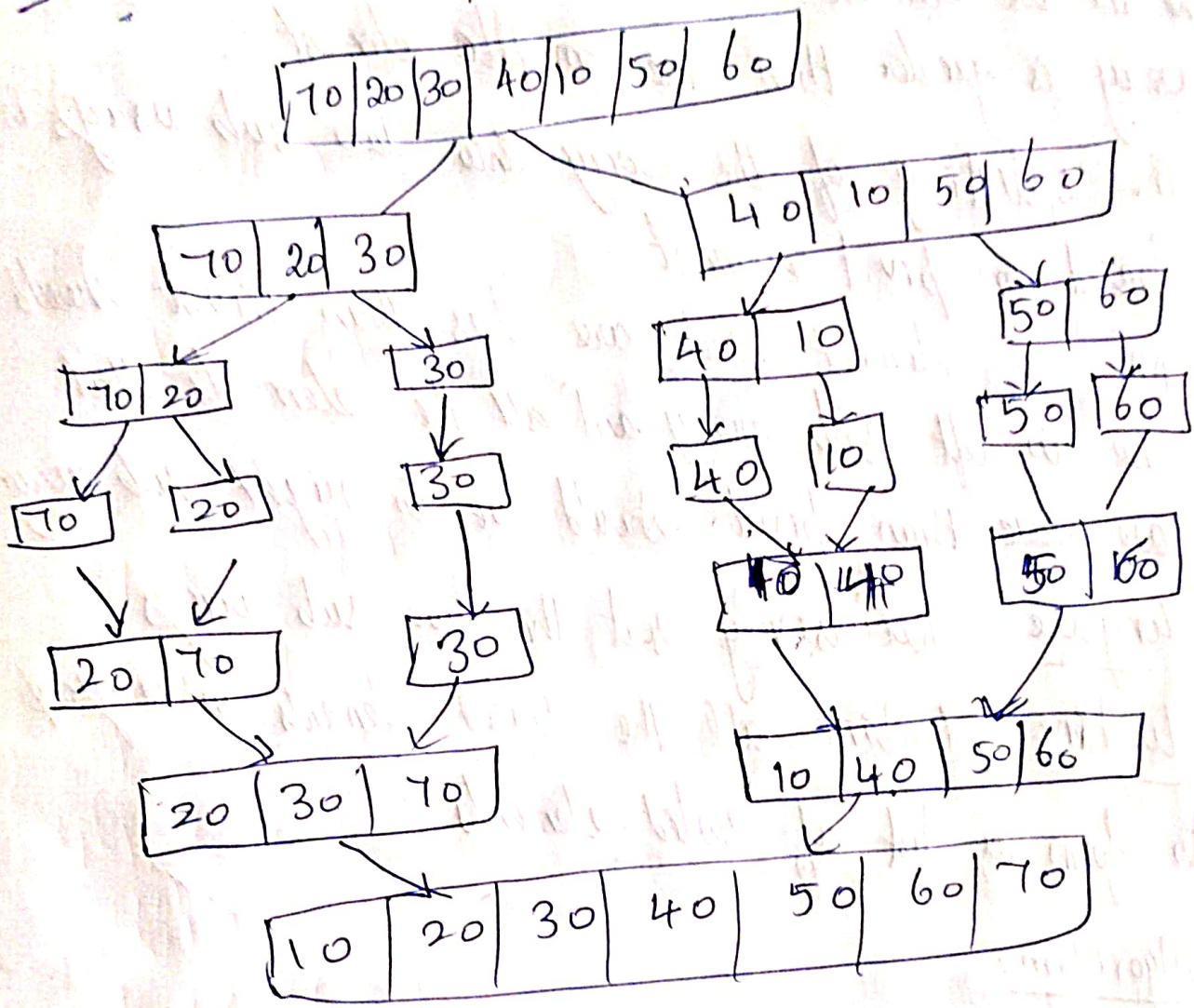
```
print (" Enter the elements :")
num = int(input())
A.append (num)
print (" original list is . . . )
print (A)
merge_sort (A, 0, n-1)
print (" sorted list is . . ")
print (A)
```

o/p How many elements you want to sort?

```
5
Enter the elements :
30
Enter the element
10
Enter the element
40
Enter the element
50
Enter the element
20
Original list is . . . .
[30, 10, 40, 50, 20]
Sorted list is . . .
[10, 20, 30, 40, 50]
```

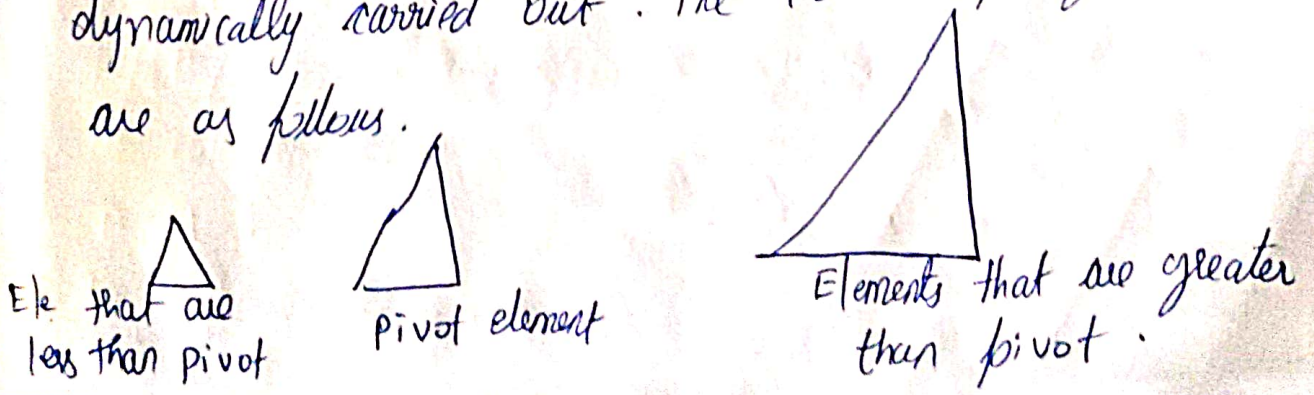
eg:- Consider the following elements for sorting using merge sort 70, 20, 30, 40, 10, 50, 60.

Soln Now we will split this list into two sublists.



3.6. Quick sort

Quick sort is a sorting alg that uses the divide & conquer strategy. In this method division is dynamically carried out. The three steps of quick sort are as follows.



Divide:

* Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element.
* The splitting of the array into two sub arrays is based on pivot element.
* All the element that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.

Conquer: Recursively sort the two sub arrays.

Combine: Combine all the sorted elements in a group to form a list of sorted elements.

Searching Techniques

If we want to find particular record efficiently from the given list of elements then there are various methods of searching that element.

* This is called searching method.
* Various alg based on these search method are known as searching alg.

Basic characteristics:

1. Should be efficient.
2. Less number of computation.
3. Space occupied by searching alg must be less.

most commonly used alg:

1. Sequential or linear search
2. Indexed sequential search.
3. Binary search.

Linear search

Sequential search is technique in which the given list of element is scanned from beginning.

* The key element is compared with every element of

the list. If found the searching is stopped otherwise it will be continued to the end of the list.

* Although this is a simple method. There are some unnecessary comparison involved in this method
* Time complexity of this algorithm is $O(n)$
* The time complexity will increase linearly with the value of n

eg :-

Roll no	Name	marks.
15	Ajith	96
2	Anand	40
13	Lalita	81
1	madhav	50
12	Anu	78
3	Jaya	94.

* search Student Roll no 12 then key - roll no will see the every record whether it is of roll no = 12.
* we obtain such a record at array [4] location.

Python Pgm

```
def search (arr, x):
for i in range (len (arr)):
  if arr [i] == x:
  return i
return -1
print ("In How many elements are there in array?")
```

```
n = int(input())
array.append(item)
print("Resultant array is \n")
print(array)
print("\n Enter key element to be searched:")
key = int(input())
location = Search(array, key)
print("The element is present at index":location)
```

o/p How many elements are there in array?

5
Enter element in array
30
Enter element in array
10
Enter element in array
40
Enter element in array
50
Enter element in array
20

Resultant array is
[30, 10, 40, 50, 20]
Enter key element to be searched
40
The element is present in index 2

3.10 Binary search:

Binary search is a searching alg in which the list of elements is divided into two sublists and key element is compared with the middle element
* If match found the location of middle element is

* returned, otherwise search into either of the halves depending upon the result produced through the match.

algorith for Binary search:

1. If (low > high)
2. return
3. mid = (low + high) / 2
4. If (x == a[mid])
5. return (mid)
6. if (x < a[mid])
7. Search for x in a[low] to [mid-1]:
8. else
9. Search for x in a[mid+1] to a[high]

eg:- -40, 11, 33, 37, 42, 45, 99, 100.

Search 99 using binary search.

0	1	2	3	4	5	6	7
-40	11	33	37	42	45	99	100

Step1 Now the key element to be searched is 99

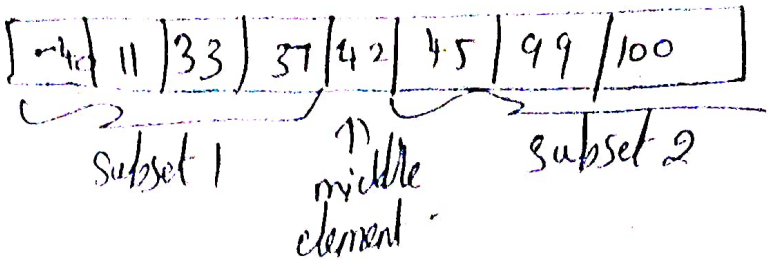
∴ Key = 99.

Step2 Find the middle element of the array.

compare with key.

42 < 99

Now handle only subset 2.



again divide subset 2 into 2 & find mid of subset 2.

mid is 99 & match is found at the 7th posit.

ie array[6].

Program

```
def BinRsearch (arr, key, low, high):
if (high >= low):
m = (low + high) // 2
if (arr[m] == KEY):
return m
elif (arr[m] > key):
return BinRsearch (arr, KEY, low, m-1)
else
return BinRsearch (arr, KEY, m+1, high)
else
return -1.
```

Driver code:

```
print ("In How many elements are there in Array ?")
n = int (input ())
array = []
i = 0
for i in range (n):
```

```
print (" Enter element in array")
item = int (input())
array.append (item)
print (" Resultant array is \n")
print (array)
print ("\n Enter the key element to be searched :")
key = int (input())
location = BinRsearch (array, key, 0, len (array)-1)
if location != -1)
print ("The element is present at index :", location)
else
print ("\n the element is not present in the list")
```

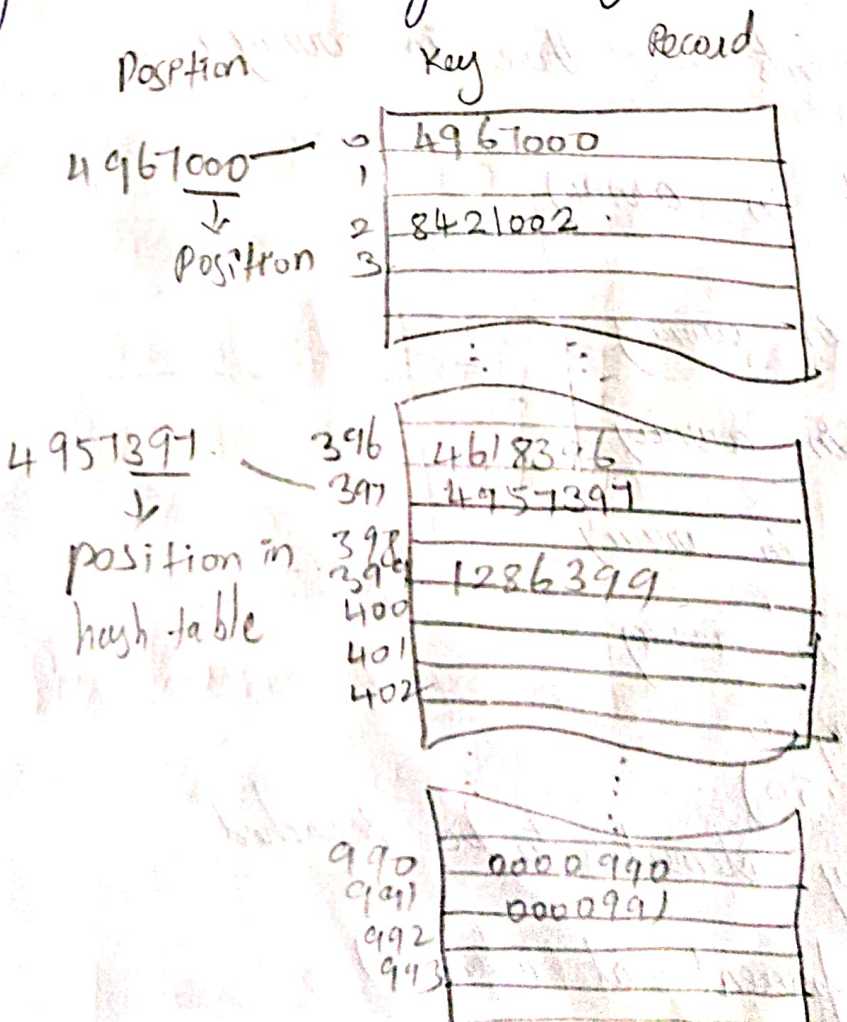
Ex

How many elements are there in array?
5
Enter element in array
10
Enter element in array
20
Enter element in array
30
Enter element in array
40
Enter element is array
Resultant array is
[10, 20, 30, 40, 50]
Enter the key element to be searched :
20
The element present at index : 1

3.11. Hashing:

Hashing is an effective way to reduce the number of comparisons.

* eg:- manufacturing company has inventory file that consis of less than 1000 parts. Each part is having 7 digit unique * the no: is called key. and the particular key record consists of the part name.
* If 1000 parts then 1000 array element can be stored
* array will be indexed from 0 to 999.
* Since key is 7 digit convert to 3 digit by taking only last three digit of key.



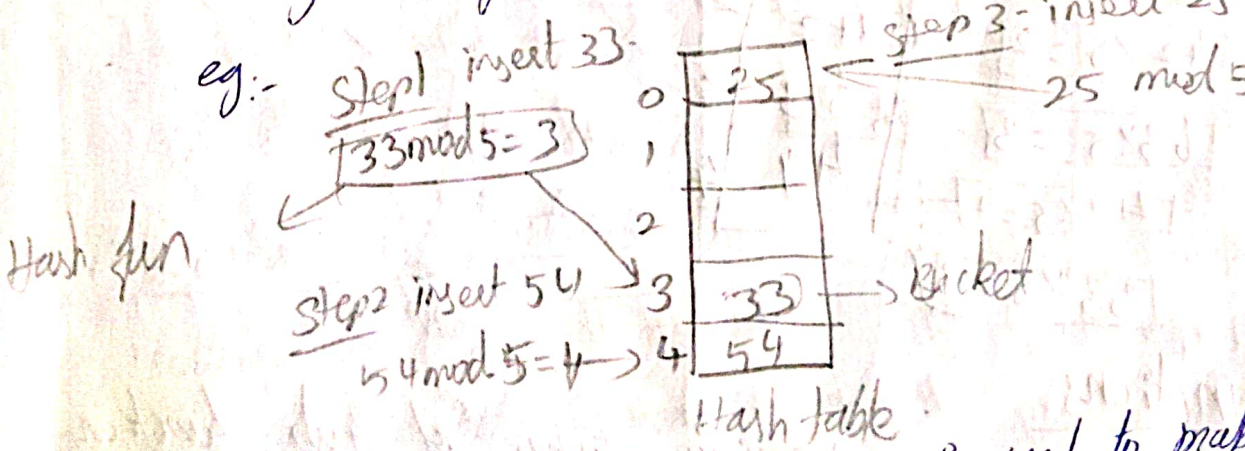
* 1st key 196700 is stored at 0th position
* 2nd key 8421002 is stored in the 2nd position by taking last 3 no's
* This method of searching is called hashing.
* The function that convert the key (4 digit) into array position is called hash function.

Here has fun is $h(key) = key \% 100$.

Basic concept of Hashing

1. Hash table: Hash table is data structure used for storing and retriving data quickly.
* Every entry in the hash table is made using hash function.

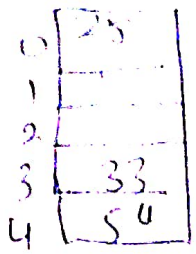
2. Hash function:
* Hash function is used to place data in hash table.
* Similarly has fun is used to retrive data from hash table.



Bucket: The hash fun H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.

4) Collision: collision is situation in which hash fun returns the same address for more than one record.

eg:-



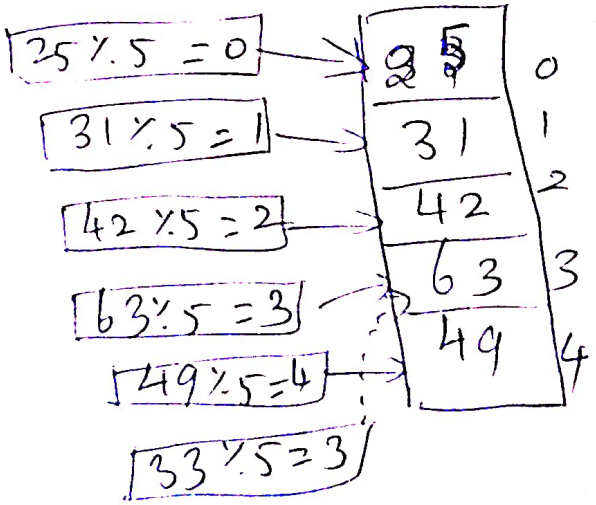
if we want to insert 55
then 55 mod 5 = 0.
But 0th position is already filled with 25, now 55 is demanding same location. this is Hence collision occur.

5) Probe: Each calculation of an address & test for sucess is known as a probe.

6) Synonym: The set of keys that has to same location are called synonyms.

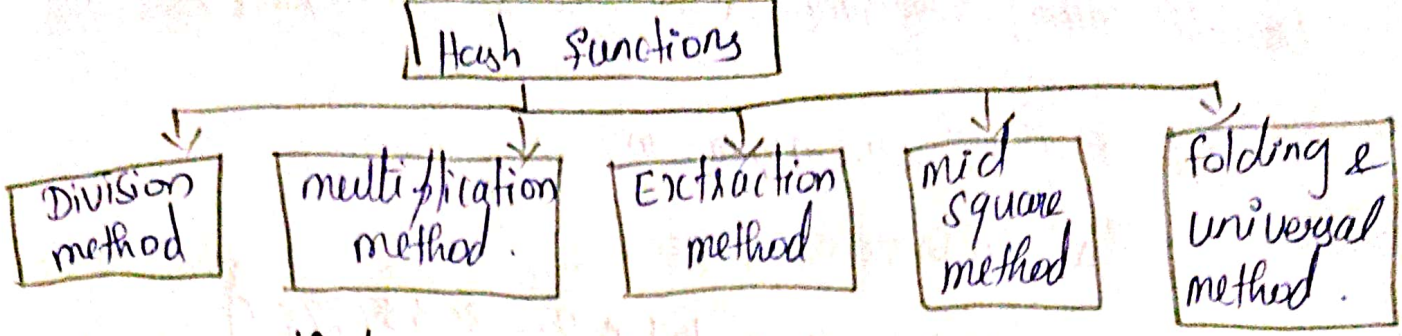
* in above given table computation 25 & 55 are synonym.

7) overflow: when hash table becomes full & new records need to be inserted then it is called overflow.



Hash functions

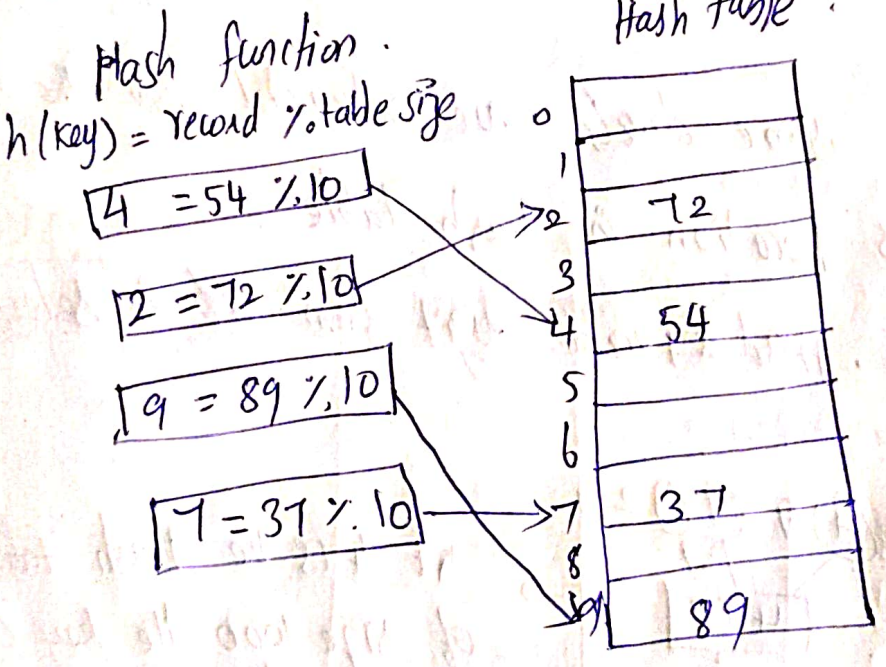
There are various types of hash function or hash methods which is used to place the elements in hash table.



Division method

The hash function depends upon the remainder of division.

* The divisor is table length.
* If the record 54, 72, 89, 37 is to be placed in hash table and if the size of table is 10 then.



Multiplicative hash function:

1. Multiply the key 'k' by a constant A wher A is in range $0 < A < 1$. Then extract fractional part of A.
2. multiply this fractional part by m & take the floor.

$$h(k) = m \{kA\}$$

fractional part.

Donald knuth suggested to use $A = 0.6180339887$.

eg:-

Let key $k = 107$, assume $m = 50$.

$A = 0.6180339887$

$h(k) = m * (107 * 0.6180339887)$

$= 50 \times 66.12$

$= 50 \times 0.12$ → fractional part.

$= 6$

$h(k) = 6$

That mean 107 can be stored at index 6 in hash table.

Extraction

In this method some digits are extracted from the key to form the address location in hash table.

eg:- Suppose first, third & fourth digit from left is selected for hash key.

4 9 7 8 2 4

478

→ At 478 the hash table of size 1000 the key can be stored.

Midsquare

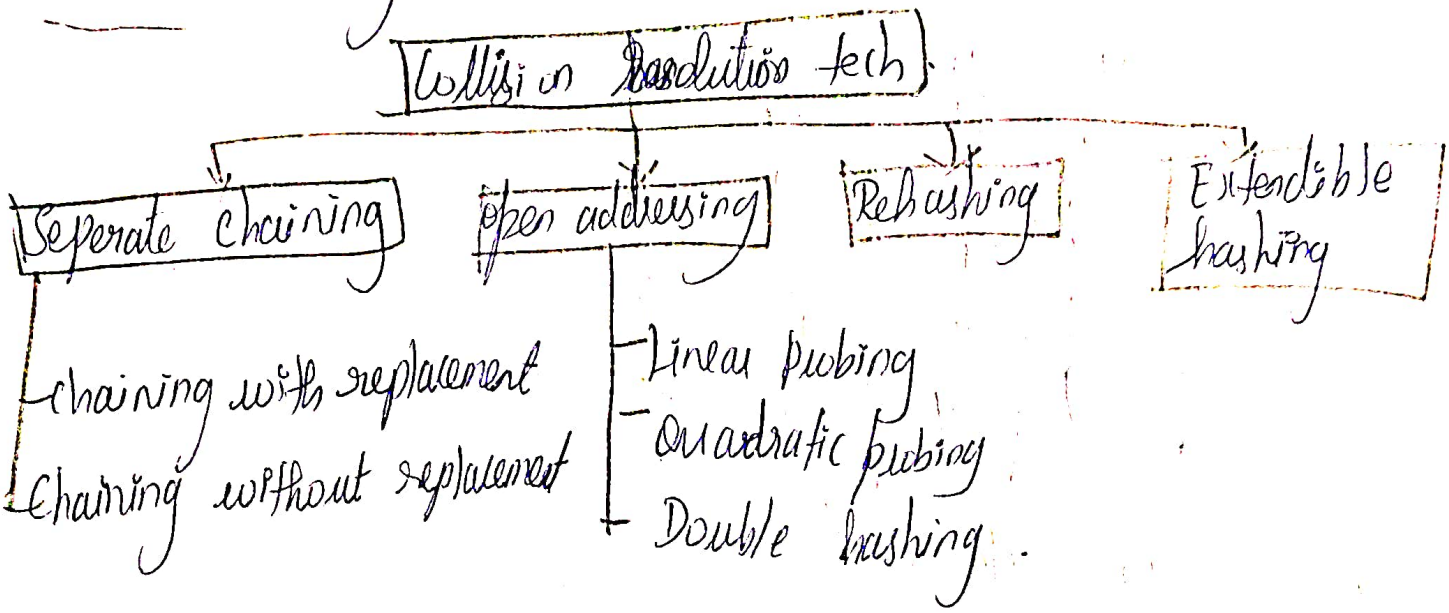
1. Square the key
2. Extract the middle part of the result. This will indicate the location of key element in hash table.

Key = 3111

$(3111)^2 = 9676321$

Collision Handling

If collision occur then it should be handled by applying some techniques. such techniques are called collision handling techniques.



Chaining

1. Chaining without replacement

A separate chain table is maintained for colliding data. When collision occurs we store the second colliding data by linear probing method. The address of this colliding data can be stored with first colliding element in the chain table, without replacement.

* The draw back of this method is finding next empty location.

eg :- 131, 3, 4, 21, 61, 6, 71, 8, 9.

Index	Data	chain
0	-1	-1
1	131	2
2	21	5
3	3	-1
4	4	-1
5	61	-1
6	6	-1
7	71	-1
8	8	-1
9	9	-1

Chaining with replacement :

eg. 131, 21, 31, 4, 5, 2

Index	Data	chain
0	-1	-1
1	131	2
2	21	3
3	31	-1
4	4	-1
5	5	-1
6		
7		
8		
9		

now next element to insert is 2, but the index 2 is filled with 21 so replace the 21 to next empty position & 2 will be placed in place of 21

index	Data	chain
0	-1	-1
1	131	-1
2	2	-1
3	31	-1
4	4	-1
5	5	-1
6	21	-1
7	-1	-1
8	-1	-1
9	-1	-1

Open addressing

Open addressing is a collision handling tech in which the entire hash table is searched in systematic way for empty cell to insert new item if collision occurs. Various tech used are - 1. Linear probing, 2. Quadratic Probing 3) Double hashing

i) Linear probing

When collision occur is when 2 records demand for the same location in the hash table, then the collision can be solved by placing second record linearly down wherever the empty location is found.

eg :- 131, 21, 31, 4, 5, 61, 7, 8.

index	data
0	
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	

2) Quadratic probing

Quadratic probing operates by taking the original hash value & adding successive values of an arbitary quadratic polynomial to the starting value. This method uses formula -

$$H_i(key) = (Hash(key) + i^2) \% m$$

m is table size or any prime no :-

eg :- Insert elements in hash table with table size 10.

37, 90, 55, 22, 11, 17, 49, 87.

37 % 10 = 7
90 % 10 = 0
55 % 10 = 5
22 % 10 = 2
11 % 10 = 1

now 17 % 10 = 7 & collision occur.

& bucket 7 has already 37. Now apply quadratic Probing

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

$(17 + 0^2) \% 10 = 7$

$(17 + 1^2) \% 10 = 8$ when $i = 1$.

the bucket 8 is empty hence place element at index 8.

Then comes 49 will be placed at index 9.

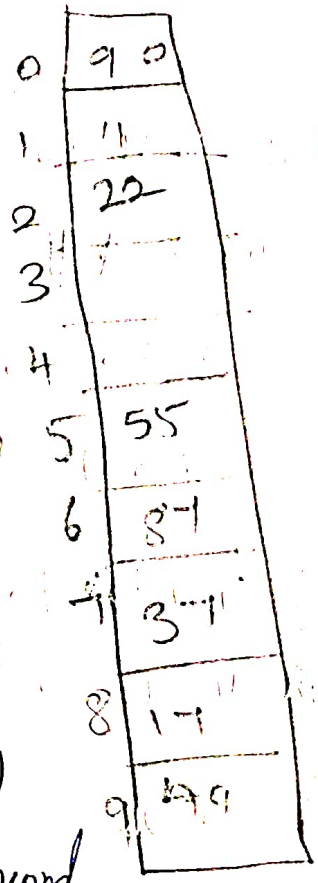
Now 87. $(87+0) \% 10 = 7$

$(87+1) \% 10 = 8$. already occupied

$(87+2) \% 10 = 1$ already "

$(87+3) \% 10 = 6$. " "

So 87 at 6th inder.



3) Double hashing

Double hashing is tech in which a second hash fun is applied to the key when collision occurs. By appling the second hash fun we will get the number of position from the 1st of collision to insert.

2 important rules i) It must never elaluate to zero.

2) make sure that all cells can be probed.

$$H_1(key) = key \bmod tablesize$$
$$H_2(key) = M - (key \bmod m)$$

1) Smaller no: small prime no of the table.

Consider no: 37, 90, 45, 22, 17, 49, 55.

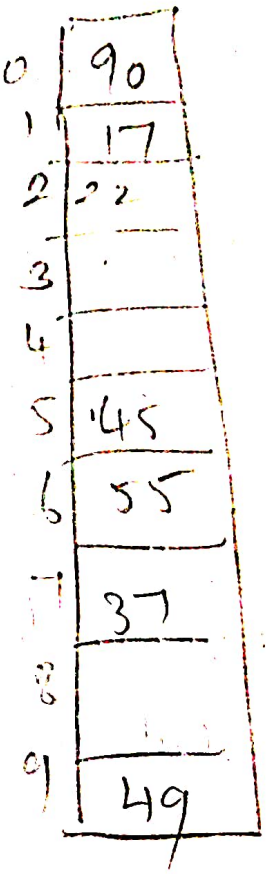
insert 37, 90, 45, 22.

Now insert 17 it collide with 37

$H_1(17) = 17 \% 10 = 7$

$H_2(key) = M - (key \% m)$.

$= 7 - (17 \% 7) = 4$.



so take 4 jumps & place at index 1

Now insert 49 & 55.

$H_1(55) = 55 \% 10 = 5$ · collision.

$H_2(55) = 7 - (55 \% 7) = 7 - 6 = 1$

jump from 55 to one index & it takes 6 th index to place 55.

Load factor

Load factor is a measure that helps in deciding when to increase the capacity of hash table. formula to calculate load factor.

Load factor = initial capacity of hash table * Load factor of hash table

eg:- initial capacity of hash table is 12 & load factor of hash table is 0.75 ∴ Then load factor is $12 \times 0.75 = 9$.

* So at most 9 elements can be stored in hash table.

* If we want to insert 10th entities add double the size of hash table.

$9 * 2 = 18$.

* load factor need to be kept low, so time complexity is O(1).

Rehashing

Rehashing is a technique in which the table ~~size~~ is resized. ie the size of table is doubled by creating new table. It is preferable the total size of table is prime number. Situation for rehashing.

* when table is completely full.
* with quadratic probing when the table is filled half.
* when insertion fail due to overflow.

In this situation we have to transfer entries from old table to new table.

Consider. 37, 90, 55, 22, 17, 49 & 87.

37 % 10 = 7
90 % 10 = 0
55 % 10 = 5
22 % 10 = 2
17 % 10 = 7
49 % 10 = 9

Now the table is almost full we try to insert more element collision will occur & further insertion fail. Hence rehash by doubling the size. The old size is 10 & double

the size becomes 20. But 20 is not prime no.
we will prefer to make the table size as 23.
And new hash function will be.

H(key) = key mod 23

37 % 23 = 14
90 % 23 = 21
55 % 23 = 9
22 % 23 = 22
17 % 23 = 17
49 % 23 = 3
87 % 23 = 18.