SORTING

OVERALL : 13.5.2020

→ Sorting means arranging the data according to their values in some specified order, where order can be either ascending or descending.

→ Ex: {6, 3, 8, 1, 4} → {1, 3, 4, 6, 8} → ascending order

{6, 3, 8, 1, 4} → {8, 6, 4, 3, 1} → Descending

→ The data may be anything like strings or records.

→ The key on which sorting is performed is also known as the sort key

→ In our algorithms, we will perform sorting on a list of integer values only.

→ Also we are using the ascending order in this concept (sort order).

→ Sorting helps to bring out an item easily and faster way from the given data.

→ Basically there are 2 types of sorting.

(i) Internal sorting: is the process of sorting the data small enough and placed in main memory at a time.

(ii) External sorting: is the process of sorting the large amount of data, which can't be stored in main memory at a time.

→ secondary memory can be used here.

→ In this case, we are going to discuss external sorting only.

Selection Sort:

→ We could arrange the number in ascending order.

→ The most efficient technique to find the smallest number and put in the first place, then find the second smallest number and put it in the second place and so on.

→ Process:

→ Let 'n' of elements present in an array.

→ First we could search the smallest element from array (a[0] ... a[n-1]). Then exchange it with a[0].

→ Then we could search for 2nd smallest element from the array [a[1] ... a[n-1]] and exchange it with a[1] position.

→ Similarly we could continue the process, till the whole array is sorted.

The whole process is:-

Pass 1:

1. Search the smallest element from a[0] ... a[n-1].
2. Exchange the element with a[0].

Result: a[0] is sorted.

Pass 2:

1. Search the smallest element from a[1] ... a[n-1]
2. Exchange the element with a[1]

Result: a[0], a[1] are sorted.

- - - - - - -

Pass n-1

1. Search the smallest element from a[n-2] ... a[n-1]
2. Exchange the element with a[n-2].

Result: a[0], a[1] ... a[n-2] sorted.

→ Now all elements except the last one have been put on their proper place.

→ The remaining last element a[n-1] will definitely be the largest of all and it is automatically at its proper place.

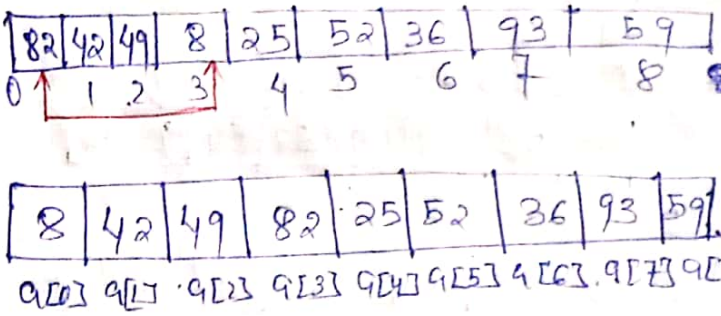
→ So, we need only n-1 passes to sort the array.

Example:- Sort the element by using selection sort.

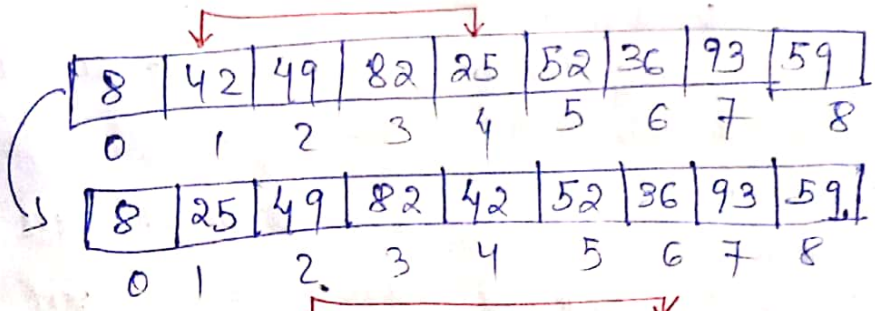
82, 42, 49, 8, 25, 52, 36, 93, 59

a[0] to a[8]

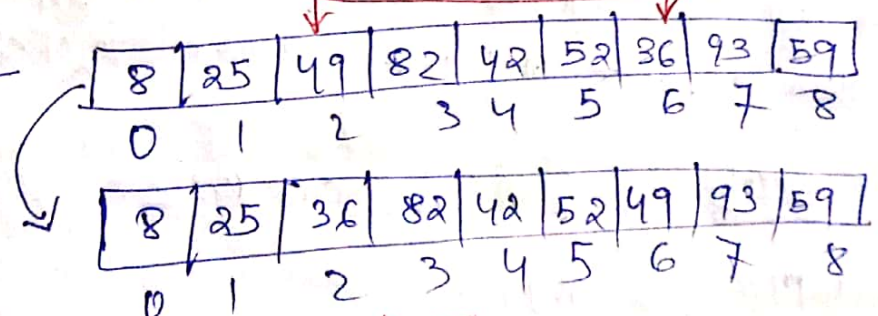
Pass-1:-



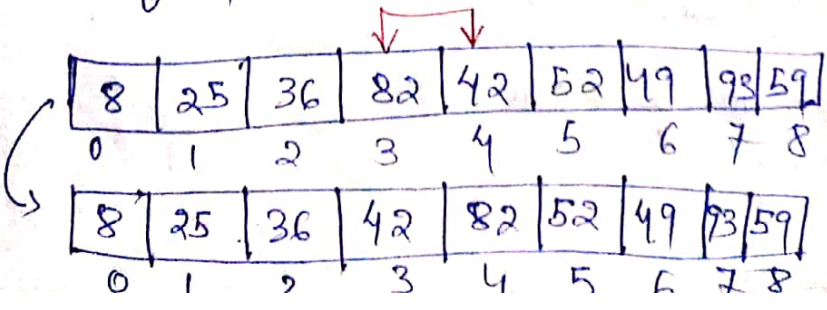
Pass-2



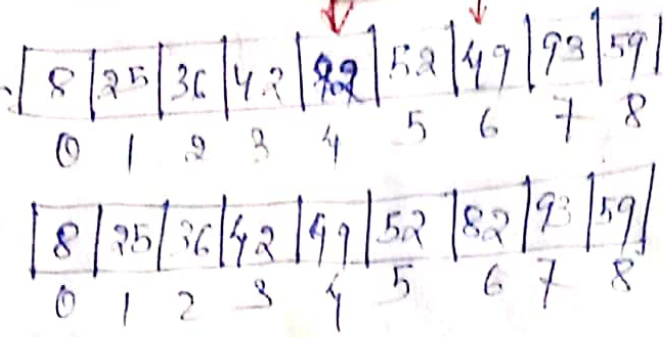
Pass-3



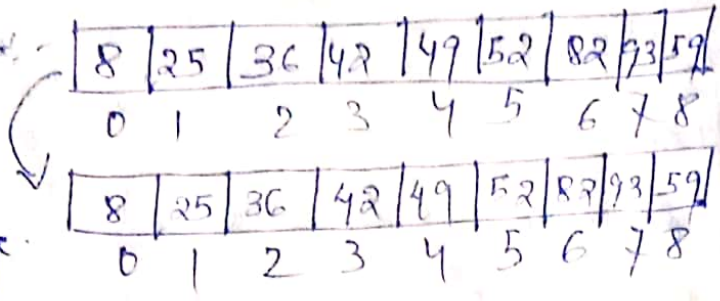
Pass-4



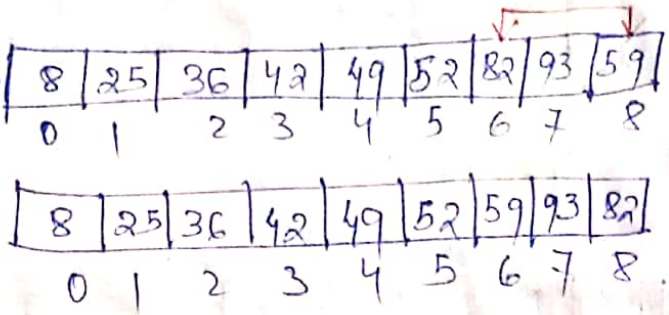
Pass-5:



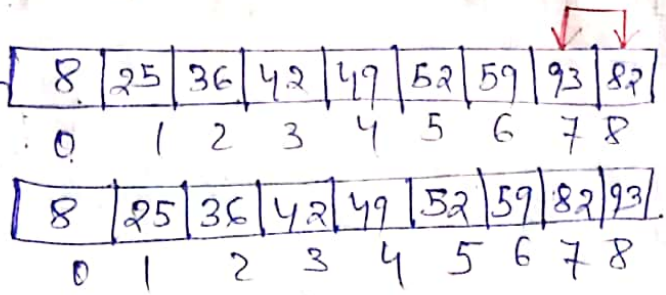
Pass-6:



Pass-7:



Pass-8:



→ All the elements on the array are sorted.

Analysis:

→ In selection sort, the no. of comparisons does not depend on the order of data.

→ So, there could be $n-1$ passes required to sort n elements.

Time complexity:- $O(n^2)$ for all cases.

Space Complexity:- $O(1)$.

→ It is not a stable sort.

Q-1 15, 20, 10, 30, 50, 18, 5, 45 → Sort the no. by using Selection Sort.

Q-2 40, 30, 9, 20, 10, 50 → " "

Bubble Sort:

→ It proceeds by scanning the list and exchanging the adjacent elements, if they are out of order each respect to each other.

→ It compared each element to the adjacent element and exchange them, if they are not in order. a[i] can be compared with a[i+1] and

→ if a[i] > a[i+1], then they will be exchanged.

→ In bubble sort there may be n-1 passes in a single pass.

→ It is a simple and easy sorting technique. But it is not an efficient technique for practical implementation.

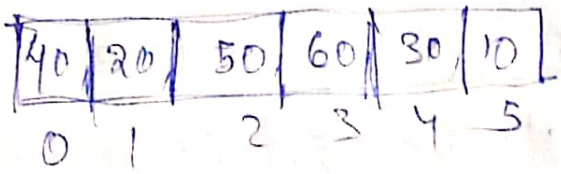
Procedure:

Pass-1: Compare a[0] and a[1], if a[0] > a[1] then exchange them
Compare a[1] and a[2] if a[1] > a[2], then exchange them.
Compare a[n-2] & a[n-1], if a[n-2] > a[n-1], then exchange them
Result: a[n-1] is sorted

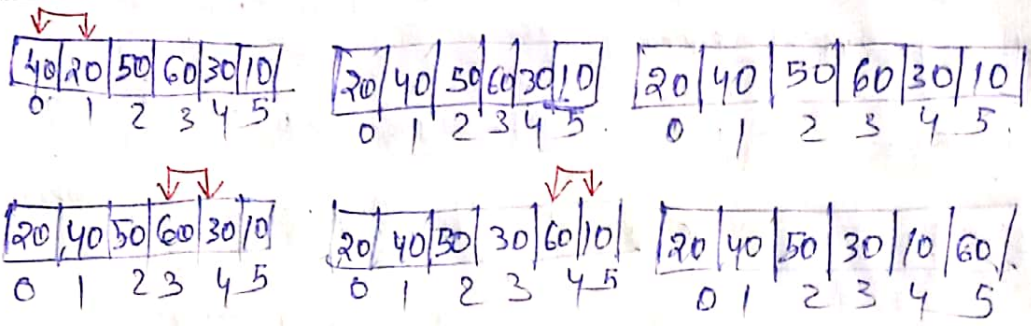
Pass-2: Compare a[0] & a[1], if a[0] > a[1], then exchange them.
Compare a[n-3] and a[n-2], if a[n-3] > a[n-2], then exchange them.
Result: a[n-2], a[n-1] are sorted.

Pass n-1: Compare a[0] > a[1], then exchange them.
Result: a[1], a[2] .... a[n-2], a[n-1] are sorted

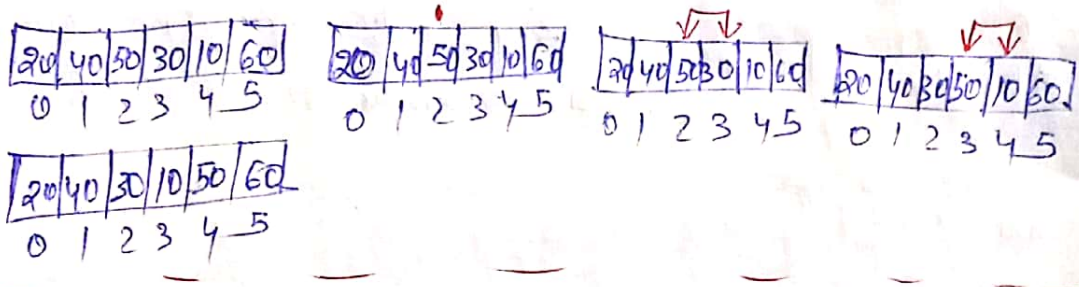
Example: Sort the following numbers using Bubble sort.



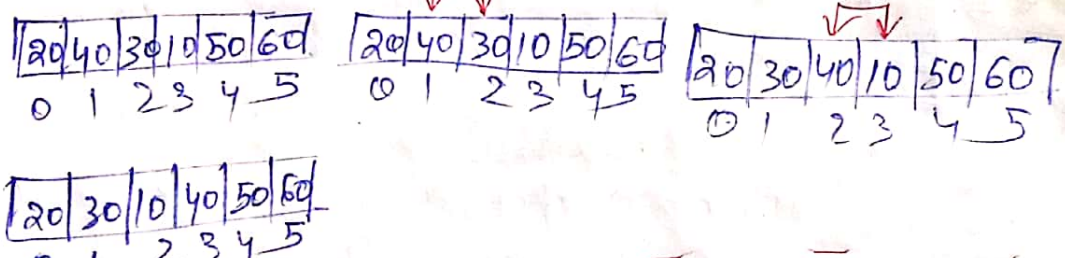
Pass-1:



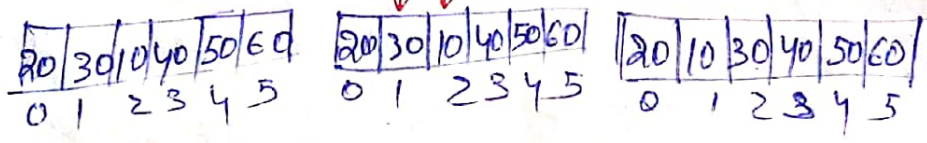
Pass-2



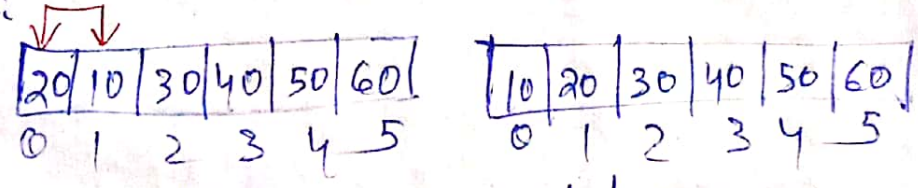
Pass-3



Pass-4



Pass-5:



All numbers are sorted in array.

Analysis :-

→ Bubble sort is not an efficient sorting technique, because, it may perform multiple swapping operations in a single pass.

→ The time complexity is O(n).

→ It should not be used for large list.

→ Applicable for smaller lists only.

→ It is a stable sort.

Q-1 2, 4, 1, 6, 8, 5, 3, 7 → Sort using ~~Merge~~ Bubble sort.

Q-2 10, 5, 2, 1, 8, 7, 3, 15 → ?.

Insertion Sort

→ The insertion sort proceeds by inserting each element at the proper place in a sorted list.

→ This is the same technique used by card players for arranging cards.

→ Let the list is divided into two parts - sorted and unsorted.

→ Initially the sorted part contains only the first element and unsorted part contains the rest of the elements.

→ In each pass, the first element of the unsorted part is taken and inserted into the sorted part at appropriate place.

→ If n no. of elements present in the list. then n-1 passes required to sort the whole list.

Procedure:

Pass-1: Sorted part : a[0]

Unsorted part: a[1], a[2] . . . a[n-1]

a[1] is inserted before or after a[0].

Result: a[0] and a[1] are sorted.

Pass-2: Sorted part: a[0], a[1]

Unsorted part: a[2] a[3] . . . a[n-1]

a[2] is inserted before a[0] or between a[0] and a[1] or after a[1].

Result: a[0], a[1] and a[2] are sorted.

Pass-n-1: Sorted part: a[0], a[1] . . . a[n-2]

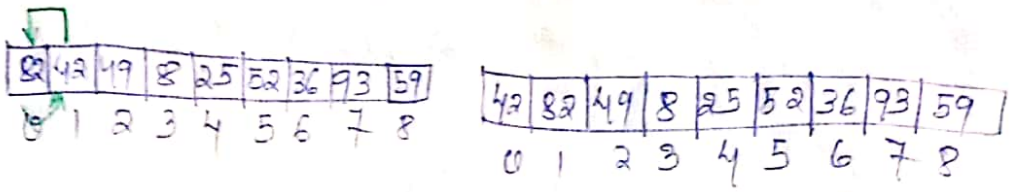
Unsorted part: a[n-1].

a[n-1] is inserted at its proper place among a[0], a[1] . . . . a[n-2].

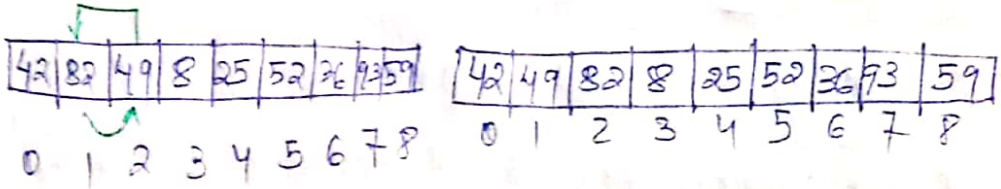
Result: List is sorted. a[0], a[1] . . . a[n-1]

Example : 82, 42, 49, 8, 25, 52, 36, 93, 59

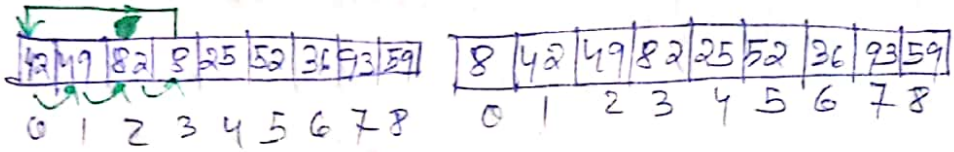
pass-1



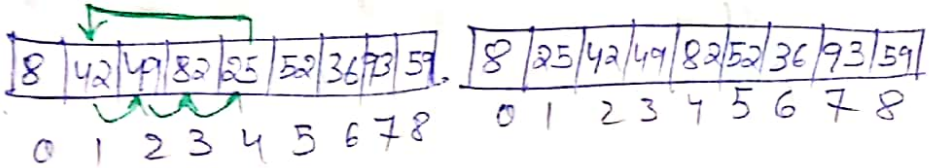
pass-2



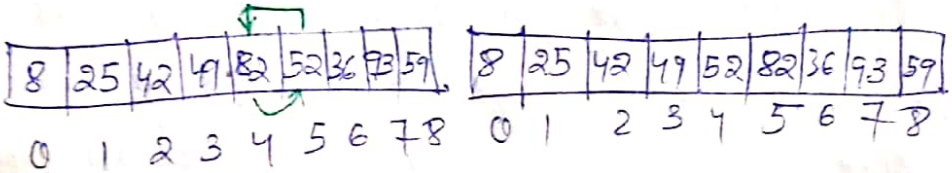
pass-3



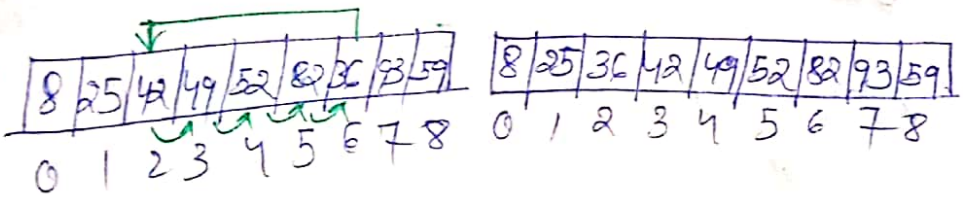
pass-4



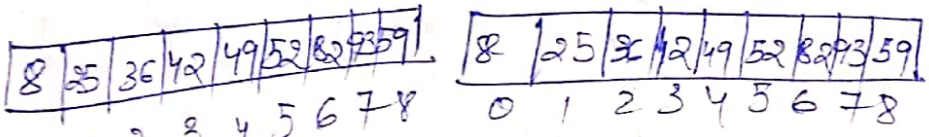
pass-5



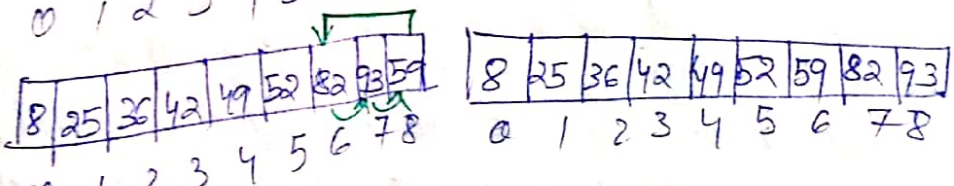
pass-6



pass-7



pass-8



Analysis :-

→ The advantage of insertion sort is easy and very efficient, when the numbers are sorted.

→ The time complexity is O(n).

→ The disadvantages of this sorting is the no. of movements.

→ Insertion sort is a stable sort.

Q:1 23, 78, 45, 8, 35, 55 → Using Insertion sort.

Q:2 5, 1, 3, 9, 1, 11, 2, 4 → //

MERGE SORT:

→ Merge sort was developed by Jon von Neumann in 1945.

→ The main task of merge sort is merging two sorted lists into a single sorted list.

→ This sort may perform on different cases.

Case 1: Two sorted arrays is given, and combine these two arrays and store in another array.

Case 2: Top down merge sort, which perform recursively. This case is mostly required in real world implementation.

Case 3: Bottom up merge sort, which perform iteratively.

→ The process of combining sorted arrays is called merging.

→ In this concept, Case-2 is discussed here.

Process: Case-2:

→ This algorithm is based on the divide and conquer technique. The list is recursively divided, till we get single element lists which are obviously sorted.

→ Then, the lists are merged repeatedly to get a single sorted list.

→ The procedure for top-down merge sort:-

1. Divide the list into two sublists of almost equal size.
2. Sort the left sublist recursively using merge sort.

3. Sort the left sublist recursively using merge sort.

4. Merge the two sorted sublists.

→ The condition for recursion is terminated, when the sublist contains only one element.

To divide the list, find mid = $\frac{low + high}{2}$.

Example

8, 5, 89, 30, 42, 92, 64, 4, 21, 56, 3

array a[] | 8 | 5 | 89 | 30 | 42 | 92 | 64 | 4 | 21 | 56 | 3 |

low 0 1 2 3 4 5 6 7 8 9 10 high.

mid

So, to divide the list we have to find out the mid position, then divide it left sublist and right sublist.

mid = $\frac{low + high}{2} = \frac{0+10}{2} = 5$ low = 0, high = 10.

mid = a[5] = 92.

→ So, divide the list into two sublist:-

Left sub-list

| 8 | 5 | 89 | 30 | 42 | 92 |

0 1 2 3 4 5

Right sublist.

| 64 | 4 | 21 | 56 | 3 |

0 1 2 3 4

→ Now, we first focus on left sublist and follow the above procedure to get single element on the list.

| 8 | 5 | 89 | 30 | 42 | 92 |

low 0 1 2 3 4 5 high

mid

mid = $\frac{low + high}{2}$ = $\frac{0+5}{2}$ = 2.

So, mid = a[2] = 89.

→ Now, the list is divided into two sub-sub list.

| 8 | 5 | 89 |

0 1 2

| 30 | 42 | 92 |

3 4 5

and combine

→ So, we can repeat the procedure, and finally get left sublist as: | 5 | 8 | 30 | 42 | 89 | 92 |.

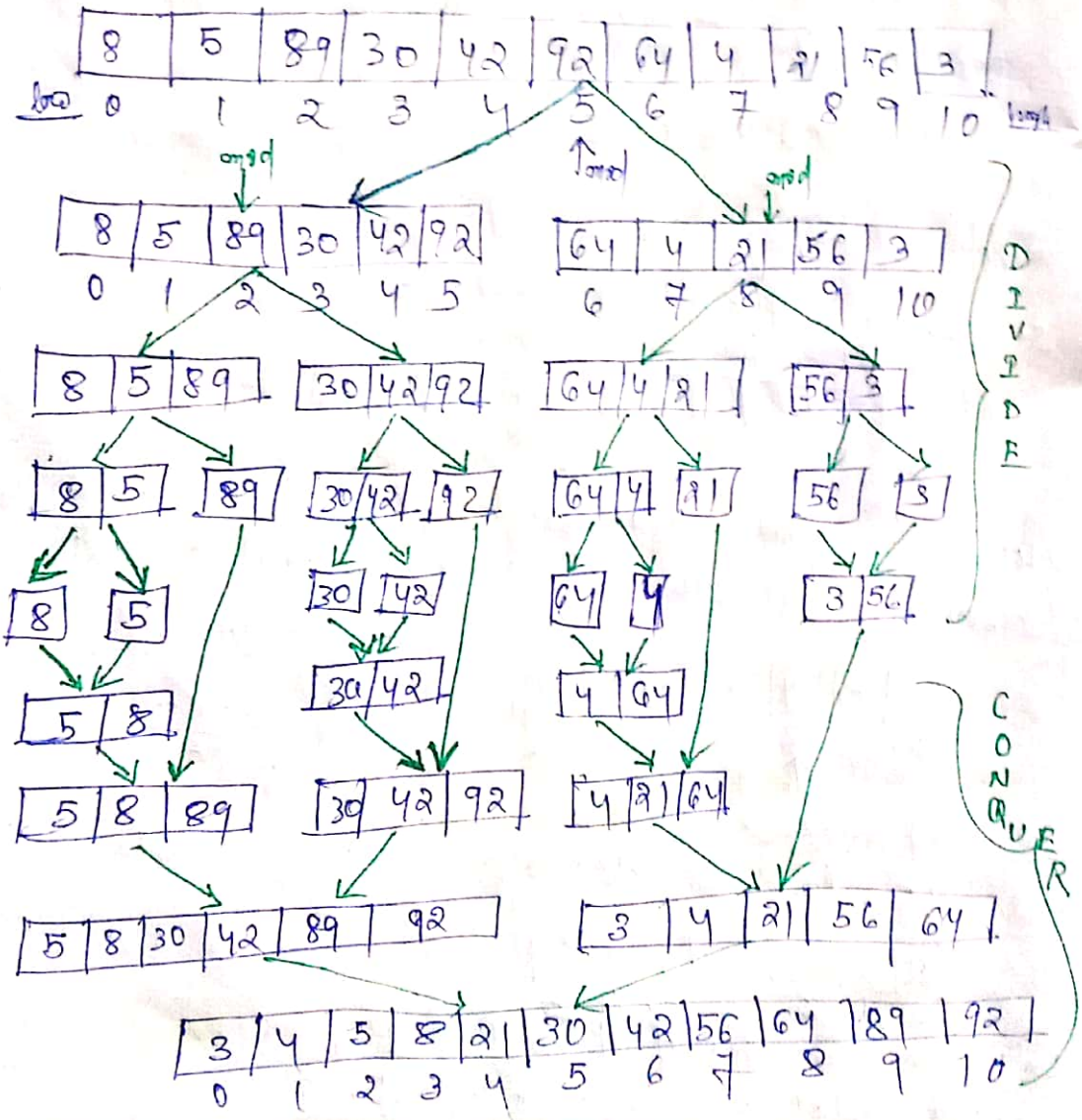
→ Similarly we can follow the procedure to sort the right sub list and finally get right sub list as

3	4	21	56	64
0	1	2	3	4

→ Now, we perform the final merging operation to left sublist and right sublist and get the sorted list as given.

3	4	5	8	21	30	42	56	64	87	92
0	1	2	3	4	5	6	7	8	9	10

→ The whole process can represented through hierarchical structure.



Analysis :-

→ The time complexity of merge sort is $O(n \log n)$ in both worst and average case.

→ It is a stable sort.

→ It requires an extra space, i.e. $O(n)$.

→ It is more effective for larger list. So we avoid to implement on smaller list.

→ Requires more memory space to store the sub elements of splitted list.

Q-1 10, 5, 8, 2, 7, 1, 3, 15 → using Merge Sort.

Q-2 13, 19, 25, 29, 41, 59, 95 → " "

Quick Sort:

→ Quick sort was given by C.A.R. Hoare on 1962.

→ It is based on divide and conquer technique. In this technique, a problem is divided into small problems, which are again divided into smaller problems and so on.

→ Quick sort is a very fast sorting technique and its average case performance is O(nlogn).

→ No additional space is required for sorting.

→ It is also known as partition exchange sort. It is very fast because, less exchanges are needed for sorting position.

→ We will choose any element, called as pivot, which is basically 1st in position 1st in the list.

→ The following process is to be followed:-

(i) All the elements left of the pivot are less than or equal to the pivot (<=)

(ii) Right of pivot greater (>=) or equal to the pivot.

(iii) Any element of the list can be taken as pivot. But here we have taken 1st element as pivot.

Procedure Let we have an array a[], with low and up as the lower and upper bounds.

(i) Take the first element of list as pivot.

(ii) The list is partitioned in such a way, that pivot comes at its proper place.

(iii) After this partition, all elements to the left of pivot are < than or = to pivot.

(iv) All elements to the right of pivot are > or = to pivot.

(v) Create two sublists, left and right of pivot.
Left sublist is a[low] ... a[pivotloc-1]
and right sublist is a[pivotloc+1] ... a[up].

(vi) The left sublist is sorted using quicksort recursively.

(vii) The right " " " " recursively

(viii) When the sublist contains only one element or no element, the recursion process is terminated.

(ix) No need of combining the sorted sublists at the end.

Procedure to Partition the list (pivot at its proper place)

(i) Compare the pivot with a[i], and increment i, if a[i] < pivot element. So, the index value of i moves from ~~right to left~~ left to right and stops

(ii) when we get an element >= to the pivot.

↳ also compare the pivot with a[j], and decrement j, if a[j] > pivot. So, the index variable j moves from right to left, and stops when we get an element <= to pivot.

(iii) If $i < j$, Exchange the value of a[i] and a[j] (increment i and decrement j).

else

Do not exchange, increment i.

(iv) Repeat steps (i), (ii), (iii) till the value of $i <= j$. (the moment i exceeds j).

(v) When value of i more than j, we have found proper place for pivot element as given by j.

Hence, value of a[low] and a[j]

Hence, now pivot is to be placed at position j.

→ pivot was at position a[low], so, we can place it at a[j] by exchanging their value.

→ Now, pivot is at position j, which is its final position.

Example:

pivot element

→ low = 0, up = 11

i = 1, j = 11, pivot = a[low] = 48

48	44	19	59	72	80	42	65	82	8	95	68
0	1	2	3	4	5	6	7	8	9	10	11

i=1, j=11

→ 44 < 48, Increment i

48	44	19	59	72	80	42	65	82	8	95	68

i=1 → j=11

19 < 48, Increment i,

48	44	19	59	72	80	42	65	82	8	95	68

i=2 → j=11

59 > 48, stop i at 3.

48	44	19	59	72	80	42	65	82	8	95	68

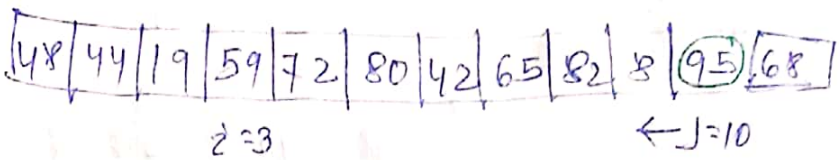
i=3 j=11

68 > 48, Decrement j,

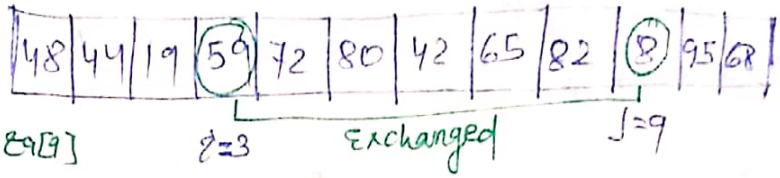
48	44	19	59	72	80	42	65	82	8	95	68

i=3 ← j=11

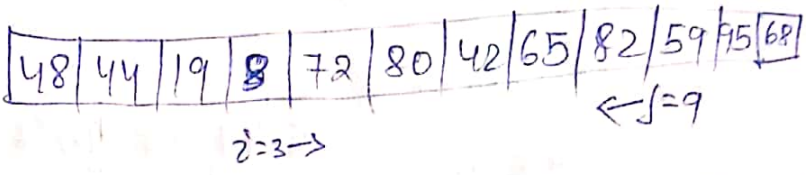
→ 95 > 48, Decrement j,



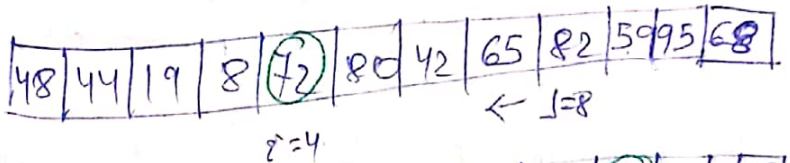
→ 8 < 48, stop j at 9.
Now both i & j stopped
Since i < j, Exchange a[3] & a[9]
3 9



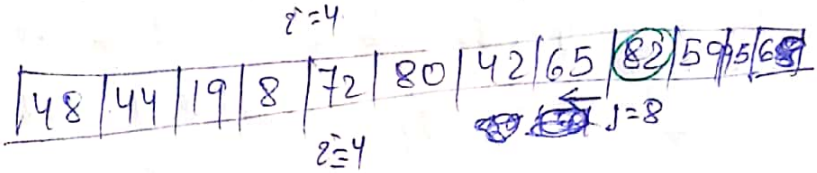
→ Increment i & Decrement j.



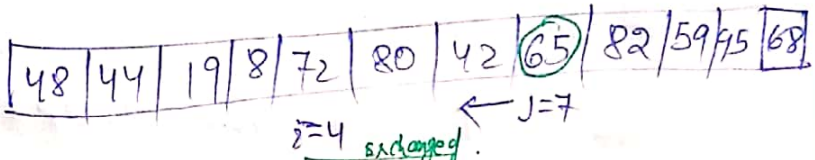
→ 72 > 48, stop i at 4.



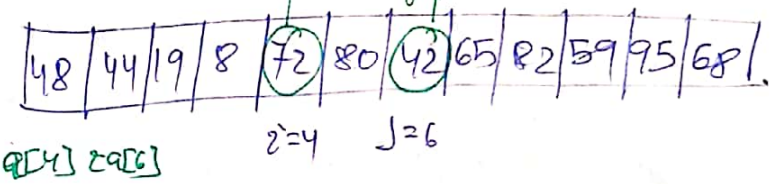
→ 82 > 48, decrement j



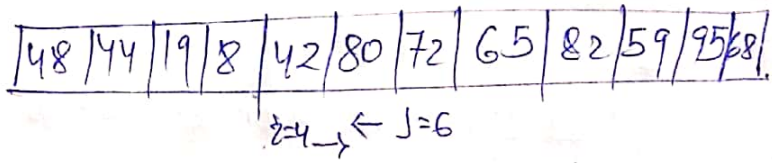
→ 65 > 48, decrements



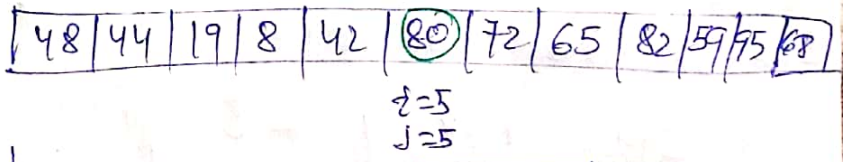
→ 42 < 48, stop j at 6.
→ Both i & j stopped.
Since i < j, Exchange a[4] & a[6]
4 6



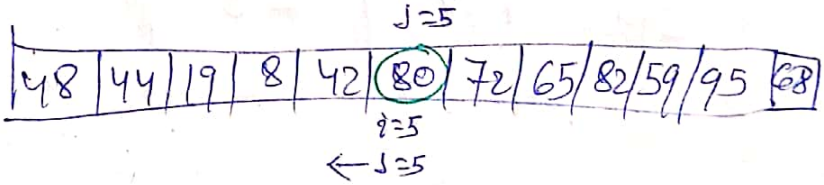
→ Increment i & Decrement j.



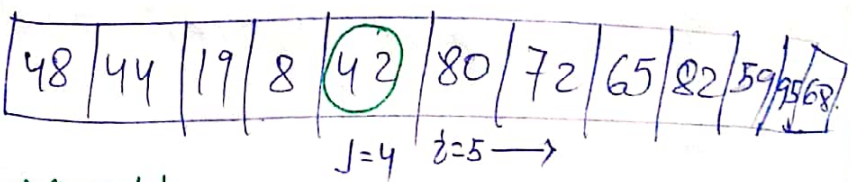
→ 80 > 48, stop i at 5



→ 80 > 48, decrement j.

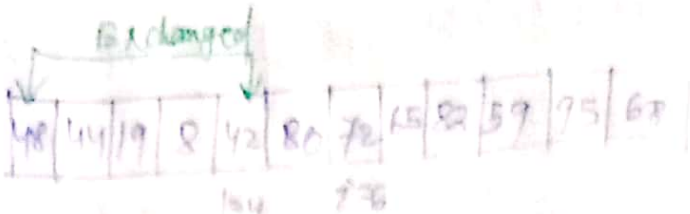


→ 42 < 48, stop j at 4
Both i & j stopped.
i & j not less than j
5 4
So, no exchange, i & j incremented.



→ Now, $i = 6$, $j = 4$, so $i > j$, so

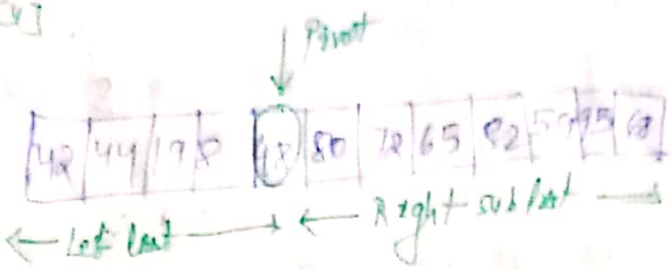
Exchanged



stop movement of i & j & the location for pivot

Exchange a[0] and a[4].

→ pivot placed at proper place.



→ The location of the pivot is the value of j, so, the function partition will return the value of j.

→ Now left sublist, low = 0, up = 3

Right " , low = 5, up = 11

Analysis of Quick sort

→ The time complexity of Quick sort in best case and average case is $O(n \log n)$.

→ The worst case performance is $O(n^2)$.

→ space complexity is $O(\log n)$

→ It is not a stable sort.

Q-1 52, 37, 63, 14, 17, 8, 6, 25 → using Quick sort.

Q-2 10, 60, 30, 50, 20, 40, 5, 70 → " "

HASHING :-

→ To reduce the many key comparisons in searching technique and other efficient operations and operations a constant time to access any record from the array, hashing concept is introduced.

→ To reduce the wastage of space, we can modify the direct addressing approach and use the key value to find out its address.

→ The process of converting a key to an address (index of array) is called hashing or key to address transformation.

→ It is possible through hash function.

→ The hash function takes a key as an input and returns the hash value of that key, which is used as the address for storing the key in the array.

→ The keys may be any type, integers, string etc. But hash value will always be an integer.

Key → Hash function → Address

So, we can define this as :-

$h(K) = a$, where h → hash function
K → key
a → hash value

→ Now, K can be stored at array index a. (Hashed address)

→ The process of generating addresses from keys is called hashing, the key and the array in which insertion and searching is done through hashing is called hash table.

→ Each entry of hash table consists of a key and associated record.

→ For searching a record we generate an address (index) by applying hash function to the key and access the record at that address.

→ For accessing any record we apply the same hash function to the key and then access the record at that address given by hash function.

Example

ISBN no of book is 8183330487. Then the record of that book will be stored at index 45.

→ When add the ISBN book no → 8183330487 → 45. So, store the record at location 45 index.

→ But another book no having same 45, then it is not stored at the same index [45].

This situation is called collision.

→ So, collision occurs, when the hash function generates the same address for different keys.

→ To reduce the collision, we should try to select a hash function, that minimizes collision.

→ Two factors are important to choose a good hash function:-

(1) Selecting a hash function that converts keys addresses.

(2) Resolving the collision.

Hash Functions

→ The hash function generates a table address from a given key. It becomes some mapping between key and hash table

→ Let 'n' be the size of hash table. So, we need a hash function that can generate addresses in the range 0 to n-1.

→ To choose a good hash function, two criteria are required:-

(1) it should be easy to compute

(2) it should generate addresses with minimum collision.

→ The general approach is to have a function that can work on the keys such that addresses generated are distributed.

→ There are some of the techniques to perform.

Truncation (or Extraction):

→ This is the easiest method for computing address from a key. Here we take only a part of the keys as address.

We may take rightmost or leftmost digits.

→ Let us take 8 digit keys:

Keys: 82894561, 87139465, 83567211, 85943228

Let table size = 100.

We take 2 rightmost digit by getting the hash table address.

→ So, the addresses of above keys are:- 61, 65, 75 and 28 respectively.

→ More collision are created on this method.

Mid square Method :

→ In this method the key is squared and some digits are taken from the middle of the square. These are taken as address.

→ The selection of digits depends on the size of the table.

→ It is important that same digits should be selected from the squares of all keys.

→ Let we have 4 digit integers as keys and size of the table is 1000.

→ We will need 3 digit addresses.

key →	1337	1273	1391
Square of key →	1785769	1620529	1934881
Address →	576	205	348

→ If the key is too large to square, then we can take a part of the key and perform mid square method on that part rather than the whole key.

I.H.P Division Method (Modulo-Division)

→ In this method the key is divided by the table size and remainder is taken as the address for each table.

→ It is provided by % operator (modulo operator).

→ Let the sized table is m; so, we will get the addresses in the range {0, 1, ... m-1}.

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

H → hash function
k → key
m → size of hash table

→ If the table size is a prime number, then collisions can be minimized.

→ Let the table size be 97 (prime no.)

k = 92394861 % 97 = 45 → hash address
k = 87139465 % 97 = 0 → "
k = 93567241 % 97 = 23 → "

} Inserted into the table.

→ If one combines other hashing methods with division method, then it could be more useful technique.

Collision Resolution

→ A hash function should perform one to one mapping between set of all possible keys and the hash table addresses.

→ But this is a difficult solution and no hash function can totally prevent collisions.

→ If collision occurs, whenever a key is mapped to an address that is already occupied, and the collision resolution techniques suggest for an alternate place, where this key can be accommodated.

→ The two collision resolution techniques are:-

1. Open Addressing (Closed Hashing)
2. Separate chaining (Open Hashing)

Any of these collision resolution techniques can be used with any hash function.

→ In the process of searching, the given key is compared with many keys and each key comparison is known as probe.

→ The efficiency of a collision resolution technique is defined in terms of the no. of probes required to find a record with given key.

Open Addressing (Closed Hashing):

→ In open addressing the key which caused the collision is placed inside the hash table itself, but a location other than the hash address.

→ If that address is already occupied, then we need to try to insert the key at some other empty location inside the table.

→ The array is assumed to be closed and hence this method is termed as closed hashing.

→ There are 3 methods to search for an empty location inside the table:-

(1) Linear probing
(2) Quadratic
(3) Double hashing.

Linear Probing:

→ If address given by hash function is already occupied then the key will be inserted in the next empty position in the hash table.

→ If the address given by hash function is a and if it is not empty, then we will try to insert the key at next location i.e address a+1

→ If the address a+1 is also occupied, then we will try to insert at next location and so on.

→ While probing the array for empty positions, we assume that the array is circular.

→ Let the array size is N, then after (N-1)th position, search will resume from 0th position of the array.

$$h(key) = key \% tablesize$$

Example: Let us take a hash table of size 11.

So, array is 0 to 10.

Insert:- 29, 46, 18, 36, 43, 21, 24, 54

$h(29) = 29\%11 = 7$

$h(46) = 46\%11 = 2$

$h(18) = 18\%11 = 7$

$h(36) = 36\%11 = 3$

$h(43) = 43\%11 = 10$

$h(21) = 21\%11 = 10$

$h(24) = 24\%11 = 2$

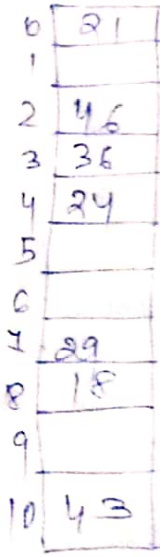
$h(54) = 54\%11 = 10$

Insert 29, 46

Insert 18

Insert 36, 43

Insert 21



Insert 24

Insert 54

The formula for linear probing can be written,

$$H(K, i) = (h(K) + i) \bmod \text{Table size}$$

i varies from 0 to Table size − 1

→ The resulting address does not cross the valid range of array address.

→ So, we will search for empty locations in the sequence :-

h(key), h(key)+1, h(key)+2 .... all modulo Table size

Example Insert 54 in table ; K = 54, i = 0, 1, ...

$h(54) = 54 \% 11 = 10$

$H(54, 0) = (10 + 0) \% 11 = 10$ (Not empty)

$H(54, 1) = (10 + 1) \% 11 = 0$ (Not empty)

$H(54, 2) = (10 + 2) \% 11 = 1$ (Empty, Insert the key)

→ This is the way inserting is done in case of linear probing.

→ The main disadvantage of the linear probing technique is primary clustering (Position 10, 0, 1, 2, 3, 4 is formed).

Quadratic Probing :

→ In this technique, colliding keys away from the original position (collision point).

The formula for quadratic probing is :-

$$H(K, i) = (h(K) + i^2) \bmod \text{Table size}$$

i varies from 0 to Table size − 1

h → hash function

→ Array is assumed to be closed.

The search for empty location sequence is:

$h(k)+0^2, h(k)+1^2, h(k)+2^2, h(k)+3^2$ ... all mod Tsize.

$\Rightarrow h(k), h(k)+1, h(k)+4, h(k)+9$ ... all mod Tsize.

Example: Tablesize = 11.

$h(key) = key \% 11$

46, 28, 21, 35, 57, 39, 19, 50

So, $h(46) = 46 \% 11 = 2$ → location in Array

$h(28) = 28 \% 11 = 6$ → " "

$h(21) = 21 \% 11 = 10$ → " "

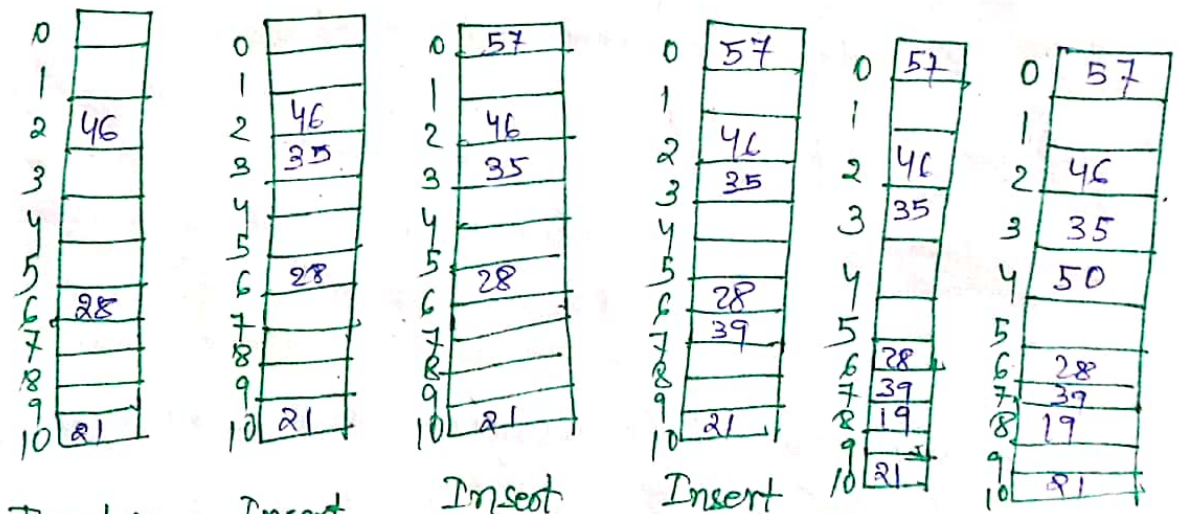
$h(35) = 35 \% 11 = 2$ → occupied, so, next location.

$h(57) = 57 \% 11 = 2$ → occupied, next, next location.

$h(39) = 39 \% 11 = 6$ → " next location.

$h(19) = 19 \% 11 = 8$ → location in array.

$h(50) = 50 \% 11 = 6$ → occupied, next next location.



The key 50 is inserted at location 4.

$H(50,0) = (6+0) \% 11 = 6$ (not empty)

$H(50,1) = (6+1^2) \% 11 = 7$ (")

$H(50,2) = (6+2^2) \% 11 = 10$ (")

$H(50,3) = (6+3^2) \% 11 = 4$ (Empty, so Inserted).

→ This technique removes the secondary clustering problem

→ The keys 44, 33, and 37 are now occupied to locations by the hash function h.

→ So, the locations that could be probed are each, each are same.

→ Another disadvantage of Q probing is that, it can't access all the positions of the table.

Double Hashing

In double hashing, the increment factor is not constant as in linear or quadratic probing, but it depends on the key.

The increment factor is another hash function and hence the name is double hashing.

The formula for double hashing is :-

$$H(k,i) = (h_1(k) + i\,h'(k)) \bmod TSize$$

i → 0 to TSize − 1

h → hash function

h' → secondary hash function

The search for empty locations will be in sequence:

$$h(k),\ h(k)+h'(k),\ h(k)+2h'(k),\ h(k)+3h'(k),\ \ldots \text{ all mod } TSize$$

$$h(k) = key \,\%\, 11$$
$$h'(k) = 7 - (key \,\%\, 7)$$

8 elements are there so, 7 is the key.

Example 46, 28, 21, 35, 57, 39, 19, 50

$h(46) = 46 \% 11 = 2$

$h(28) = 28 \% 11 = 6$

$h(21) = 21 \% 11 = 10$

$h(35) = 35 \% 11 = 2$

$h(57) = 57 \% 11 = 2$

$h(39) = 39 \% 11 = 6$

$h(19) = 19 \% 11 = 8$

$h(50) = 50 \% 11 = 6$

$H(35, 1)$

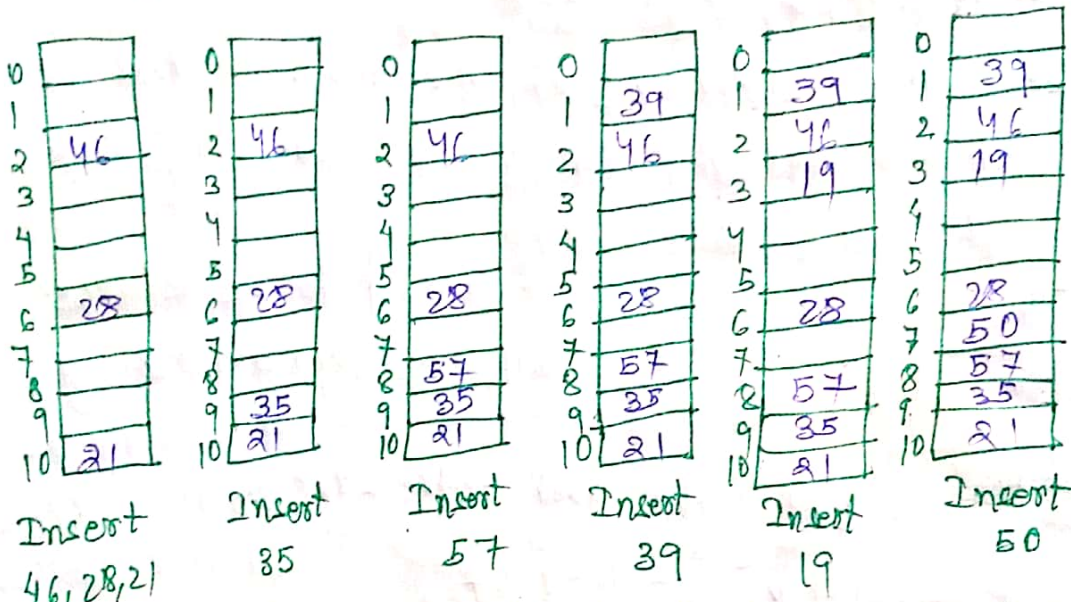
$= (h(35) + 1 \cdot h'(35)) \bmod 11$

$= (2 + 1 \cdot (7 - (35 \% 7))) \% 11$

$= 2 + 1 \cdot (7 - 0) \% 11$

$= 9 \% 11 = 9$ → empty.

So, 35 is inserted at location 9.



→ While selecting the second hash function, one must consider these two points!

(i) It should never give a value of 0.

(ii) The value given by second hash function should be relatively prime to the table size.

→ Double hashing is more complex and slower than linear and quadratic probing, because it requires two times calculation of hash function.

→ Hash table overflow may occur.

Separate Chaining (open hashing)

→ In this method linked list concept is used to maintain the elements having same hash address.

→ In this case the hash table does not contain actual keys and records, but it just an array of pointers.

→ All elements having the same address & will be stored in a separate linked list and the starting address of that linked list will be stored in the index of hash table.

→ So, array index [i] of hash table contains a pointer to all elements that share the hash address i

→ The linked lists are referred to as chains.

So, the method is called as separate chaining.

Example Let the hash table size is 7.

So, $H(key) = key \% 7$

Let the keys are:- 4895, 6559, 5912, 4047, 6766, 4390, 4640, 4900, 4411.

$H(4895) = 4895 \% 7 = 2$

$H(6559) = 6559 \% 7 = 0$

$H(5912) = 5912 \% 7 = 4$

$H(4047) = 4047 \% 7 = 1$

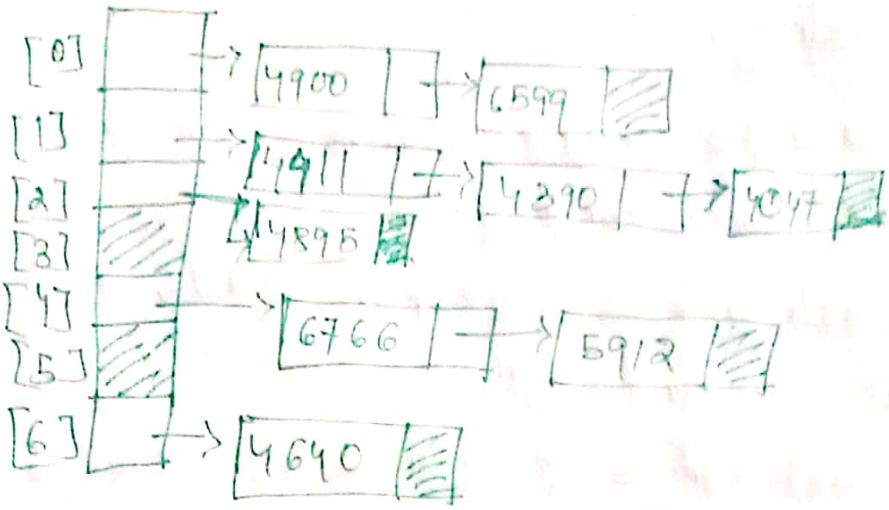
$H(6766) = 6766 \% 7 = 4$

$H(4390) = 4390 \% 7 = 1$

$H(4640) = 4640 \% 7 = 6$

$H(4900) = 4900 \% 7 =$

$H(4411) = 4411 \% 7 = 1$



→ In this case the keys 4411, 4390, 4047 all hash to the same address, index[i] = 1.

→ So, they all stored in separate linked list whose starting address is stored in location 1 of the same array.

→ Similarly other keys are also stored in respective lists depending on their hash address.

→ In this technique, comparison are done only with keys that have same hash values, which differs from open addressing.

→ In open addressing all records are stored inside the hash table itself, so, there can be problem of hash table overflow.

→ To avoid this, enough space is required at compilation time.

→ But in separate chaining, overflow could not occur because linked lists are dynamically allocated.

→ It is not necessary that the size of the table is more than the no. of records.

→ The disadvantage of separate chaining is, it needs extra space for pointers.

Bucket Hashing :

→ This technique postpones collisions but does not resolve them completely.

→ Here hash table is made up of buckets, where each bucket can hold multiple records.

→ Each hash address carry a bucket, we can store multiple records at a given hash address.

→ Collisions occur only after a bucket is full.

→ Let us this example, each bucket can hold 3 records. So, we can store 3 records without collision.

→ But, a collision occurs, when a 4th record goes to the same hash address above.

Example :

H(4895) = 4895 % 7 = 2
H(6559) = 6559 % 7 = 0
H(5912) = 5912 % 7 = 4
H(4047) = 4047 % 7 = 1
H(6766) = 6766 % 7 = 4

H(4390) = 4390 % 7 = 1
H(4640) = 4640 % 7 = 6
H(4900) = 4900 % 7 = 0
H(4411) = 4411 % 7 = 1.

B → Bucket.

Block Bucket.

→ The disadvantage of this technique is wastage of space, many buckets could not be occupied / partially occupied.

→ Not prevent collisions but only defers them. (Collisions solve using open addressing).