DSA.

- Data Structure is a method of organising large amount of data more efficiently. So, that any operation on that data become easy.

- Needs :-

→ Provides fast searching & sorting of data.
→ It tells how data can be stored & accessed in its elementary level.

- 2 types :-

Primitive	Non - Primitive
• They are fundamental data type which are supported by a Programming languages	• They are those data Structure which are created using primitive data struct.
• int, real, integer, char	• Stack, tree, linked list.
• Operated directly by machine level instructions	• Operated directly by not by machine level.

Linear

→ A data structure that maintain a linear relationship among its element.

→ Array, linked list, Stack, queues.

Non-linear

→ A data structure in which data is not arranged in sequential order.

→ Trees & Graph.

Static DS.

→ In static data struc. the size of structure is ~~not~~ fixed.

→ Content of ds can be modified without changing memory space allocated to it.

→ Array.

Dynamic DS.

→ The size of structure is not fixed.

→ Modified & can be changed during run-time.

→ Linked list.

Operations on Data Structure :

→ Searching :- Used to find the location of one or more data items.

→ Inserting :- Used to add new items in given list of data item.

→ Deleting :- Used to delete data item from the given list.

→ Sorting :- Used in arranging items in asc. or des. acc to need.

→ Merging :- List of 2 sorted items can be combined to form a single list.

Data Abstraction :-

→ Data Abstraction is reduction of particular body of data to a simplified representation of whole.

Abstraction is the process of removing characteristics from something to reduce it to set of essential elements.

Data abstraction creates a simplified representation of data, while hiding its complexity & associated operations.

Used in mainly OOPS while working with DBMS.

⊙ Abstract Data type :-

When data structure do operations like adding or deleting an item from a list, accessing a item in list, or searching & sorting an item in a list. Then it is called abstract data type.

⊙ Algorithm :-

→ It is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get desired output.

Adv.	Disadv :-
• It is easy to understand	• Time consuming
• Step-wise soln to a given problem.	• Branching & looping are difficult to show.

⊙ Algo to add 2 no's :-

→ Step 1 :- START ADD.

Step 2 :- Get values of a & b

Step 3 :- c ← a+b

Step 4 :- display c

Step 5 :- STOP

⊙ Time Complexity	Space Complexity.
• Represent amount of time req. by the algo to run the competition.	• Represents amount of memory space required by algo in its life cycle.
• T(n) = c * n (n = steps).	• Sum of fixed part & variable part is the total space required by algo.
	• S(P) = C + SP(I) (fixed, variable)

⊙ Types of Analysis :-

→ Best Case :-

Least amount of time to execute a specific set of input.

→ Average Case :-

Average time taken for input execution.

→ Worst Case :-

Max amount of time to execute a specific set of input.

⊙ Asymptotic Notations :-

→ They are languages that allows us to analyze an algorithms running time by identifying its behaviour as the input size of algorithm increases.

3 Types.

- Big-O :- Specially describes worst case scenario.
- Omega :- Specifically describes best case scenario
- Thetha :- Represents average complexity of algo.

→ Big-O (O) :-

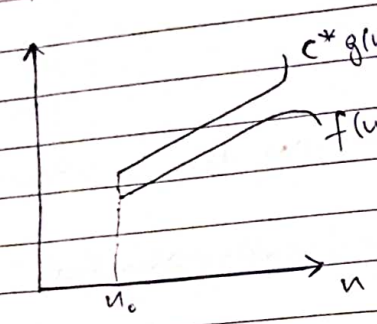
It is the method used to express the upper bound of running time of an algorithm.

→ Omega (Ω) :-

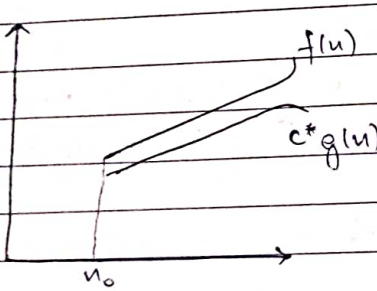
Omega gives the "lower bound" of algo run time.

→ Theta (Θ) :-

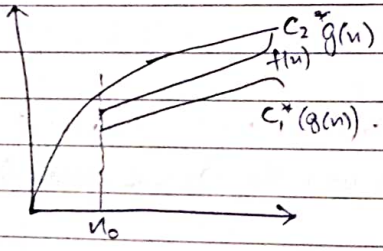
It is used when upper bound & lower bound of an algo are in same magnitude



Big-O $f(n) = O(g(n))$.



Omega $f(n) = \Omega(g(n))$



Theta $f(n) = \Theta(g(n))$.

⊙ Time – Space Trade off :-

→ It is a way of solving problem in :-

- less time by using more memory

or.

- By solving a problem in very little space by spending a long time.

→ As it is not possible to achieve both at a same time.

⊙ Recursion :-

It is a process of repeating items in a self-similar way.

If a program allows to call a func inside the same func.

Syntax :-

```
Void m1()
{
    m1()   //recursive func.
}
```

⊙ factorial using recursion.

```
#include <stdio.h>
#include<conio.h>

int fact(int);
void main()

{
    int n, f;
    printf("enter a value");
    scanf("%d", &n);
    f = fact(n);
    printf("the factorial is %d", f);
    getch();
}
int fact(int n)
{
    if(n==0)
    return(n);
    else
    return n* fact(n-1);
}.
```

Unit → 5

⊙ File :-

It is a collection of records. The file size is limited by size of memory & storage medium.

⊙ File Organisation :-

It is used to describe the way in which the records are stored in terms of blocks, & the blocks are placed on storage medium.

→ 3 types of organizing file :-

- Sequential access file organisation
- Direct access " " "
- Indexed sequential access" ".

⊙ Sequential :-

- In this, all records are stored in a sequential order.
- Storing & sorting in contigous block within files on tape or disk.
- In this search start from beg. & add" of records at end of file.

Adv.	Disadv.
• It is simple to program & easy to design.	• Sequetting file is time consuming process.
• Sequential file is best use of storage.	• Random searching is not possible.

⊙ Direct access f.o :-

- In this, all records are stored in direct access storage device such as hard disk.
- Randomly placed throughout the file.
- This is useful for immediate access to large amount of information.
- It is also called as hashing.
- Used in accessing large database.

Adv.	Disadv.
• Sorting of records are not req.	• It is expensive
• It updates several files quickly.	• Does not provide back-up facility
• Access the records (desire) immediately.	• Less storage than sequential.

⊙ Indexed Sequential :-

- This type of f.o combines both seq. files & direct access f.o . data can be accessed using index both (seq. or random).
- This file have multiple keys. Records are stored randomly on direct access (magnetic disk) by primary key.

Adv.	Disadv.
• Records can be inserted at middle of file.	• Req. unique keys.
• Access records very fast.	• Req. more storage.
	• Expensive.

- A tree is a collection of nodes.
- Height of tree is total no. of nodes from root to a leaf is longest path.

AVL Tree.

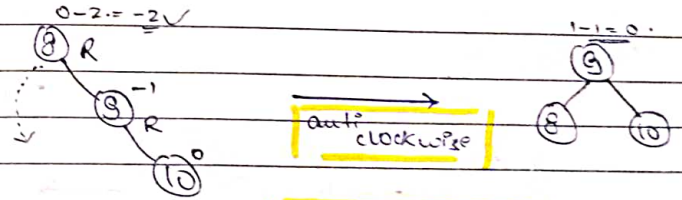
- **AVL Tree -:**

AVL Tree is a self balancing BST where the difference b/w heights of left & right subtree cannot be more than one for all nodes.

Q. 55, 25, 65, 9, 8, 15 ?

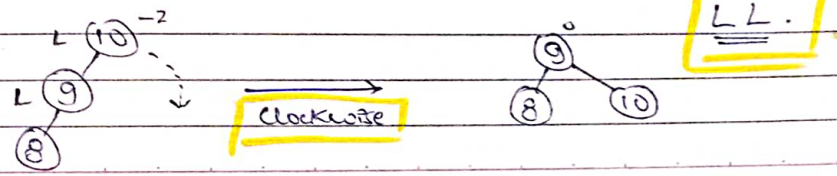
→ Balanced factor should be -1, 0, 1 taken as balanced.

- 8, 9, 10



RR Rotation

- 10, 9, 8

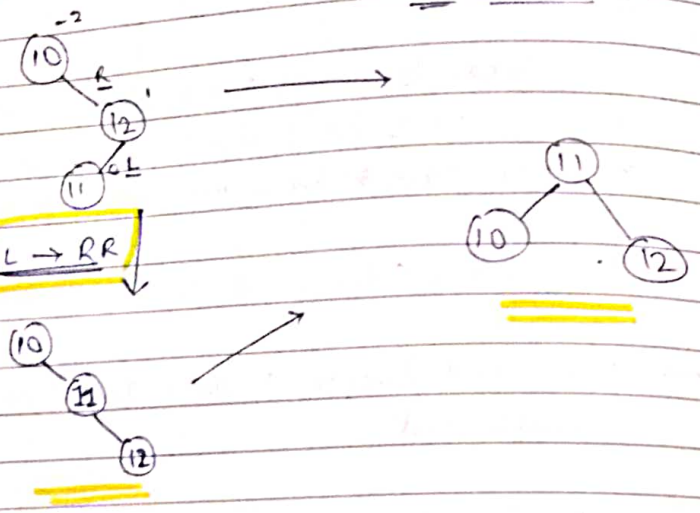


LL.

RL Rotation

2 Rotations

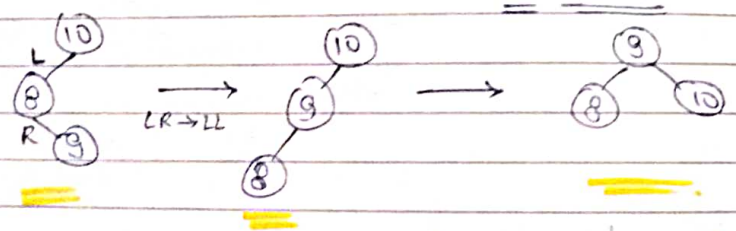
① 10, 12, 11

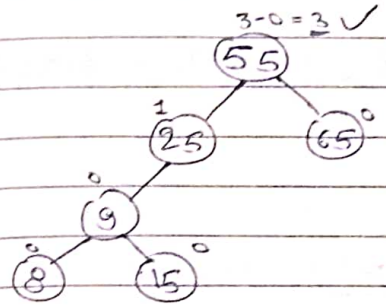


LR Rotation

2 Rotations

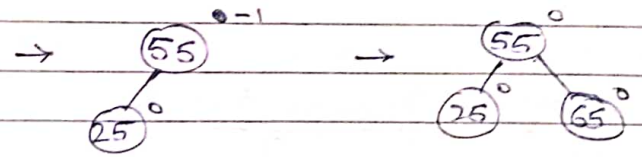
② 10, 8, 9

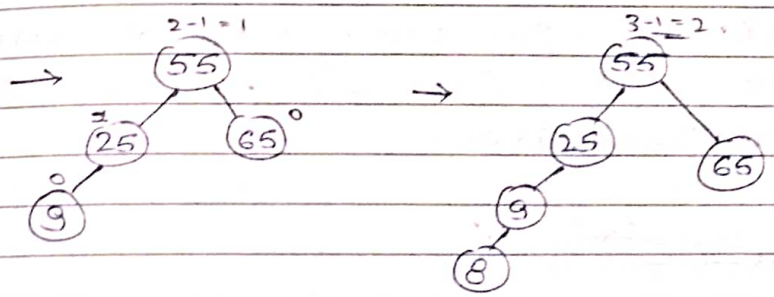


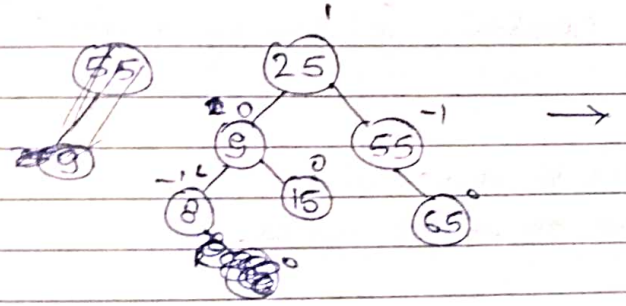


✗ wrong way.

→ So after we insert new element we will right it by rotations.







Binary Tree.

⊙ If all the nodes in a tree has atmost 2 children.

→ Strictly Binarytree :-

A BT in which every node has either 0 or 2 no of children.

→ Complete Binary tree :-

A BT in which every internal node has exactly 2 children & all leaf nodes are at same level.

⊙ Representation :-

A tree contain following parts :-

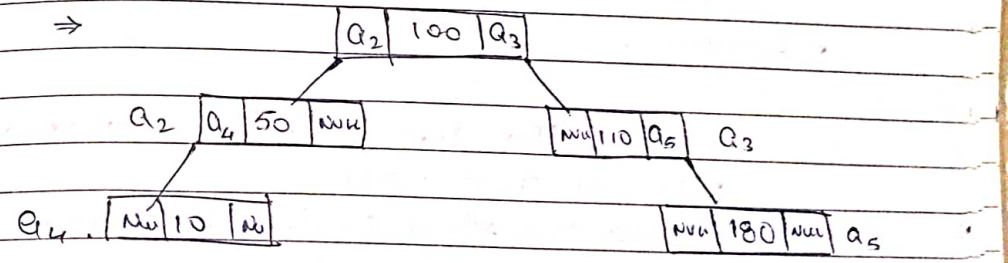
- Data
- Pointer to left-child
- Pointer to right child.

```
Struct node
{
    int data;
    Struct node * left;
    Struct node * right;
};
```

⊙ Linked representation :-

→ 100, 50, 10, 110, 180.

⇒



⊙ Tree Traversal :-

Process of visiting each node exactly once.

- Breadth tree traversal :- level by level counted
- Depth tree traversal :-
  - Pre
  - Post
  - In

```
Inorder (root)
{
    if (root ≠ NULL)
    Inorder (root → left)
    print (root → data)
    Inorder (root → right).
}.
```

Traversing Algo -:

- **Inorder -:**

Traverse the left sub-tree Inorder.
Visit the root node.
Traverse the right subtree Inorder.

Binary Search Tree

- It is a binary tree in which value of left child to the parent node is less & the value of right child to its parent node is greater.

Insertion -:

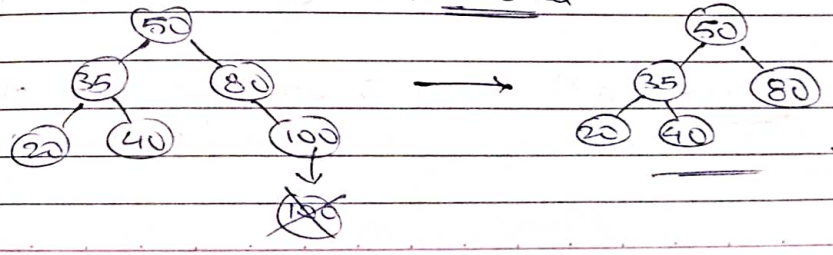
→ whenever an item to be inserted, we first find its proper location.

→ We start searching from root node, then if item is less than root's data, we search for empty location in root's left subtree & insert the item.

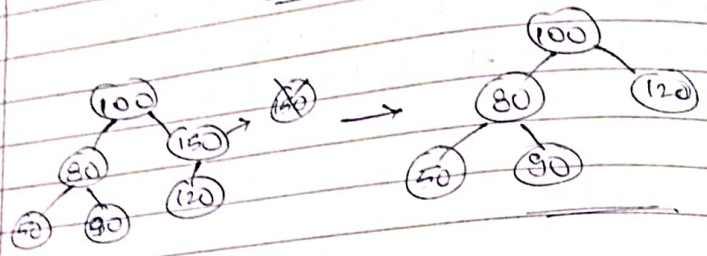
→ Otherwise we search for empty location in root right subtree & insert item.

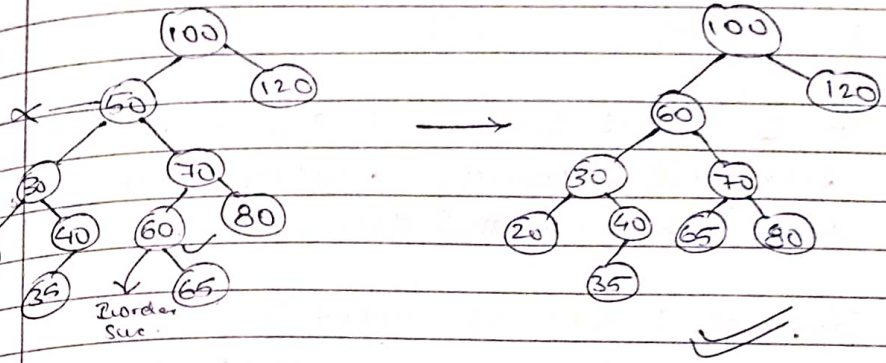
Deletion -:

No child

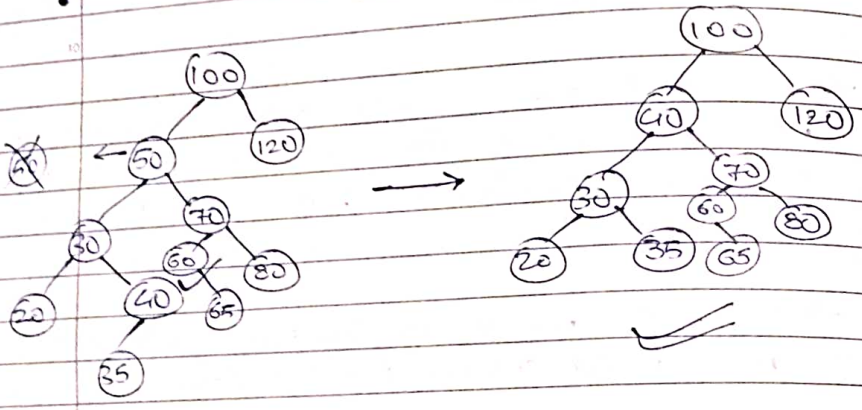


1 child.





2 children.



- 1st method :- Left subtree ची sabse bada member. (Inorder predecessor)
- 2nd method :- Right subtree ची sabse chota member (Inorder successor)

⊙ B-Tree :-

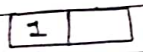
It is a self balanced search tree in which every node contain multiple keys & has more than 2 children.

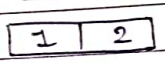
⊙ Construct B-tree of order 3 ?

$m = 3 \rightarrow$ max child $\rightarrow 3$

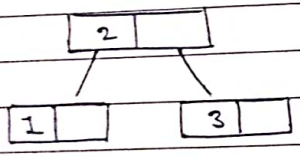
max key $\rightarrow m-1 \Rightarrow 2$

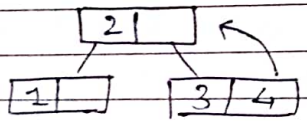
- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

1st  → 2 Keys.

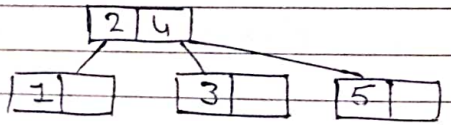
2nd 

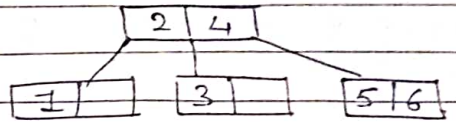
split from 2.

3rd 

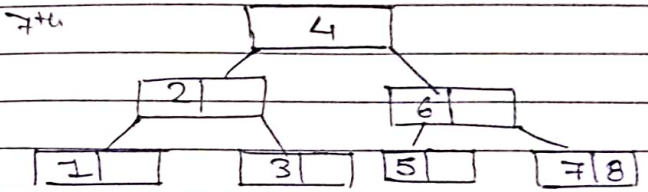
4th 

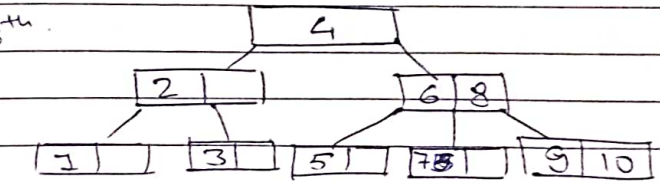
again split.

5th 

6th 

again split.

7th 

8th 

⊙ B + tree :-

They are in extension to B- trees :-

It consist of 2 types of nodes.

- Leaf node
- Internal node.

⊙ Threaded Binary tree :-

Searching.

⊙ Sequential Search :-

→ This search is applicable to a table organised either as a array or a linked list.

Complexity = O(n).

→ When using sequential search we check each & every element of list, comparing it with given key, one element at a time.

Prog :-

```
int RecLinearSearch(int arr[], int size,
                    int Key)
{
    int loc;
    if (arr[size] == Key)
        return size;

    else if (size == -1)
        return -1;

    else return (loc = RecLinearSearch(arr[],
                 size-1,
                 Key );
}
```

Binary Search :-

- For binary search, list of elements should be sorted.
- Binary search looks for searched element by comparing the middle most element of list.
- If match occurs, then index of item is returned
- If middle element is greater than item, then item is searched in sub array to left of midde element.
- Otherwise the item is searched in the sub array to right of middle element.
- Process continues until size of subarray reduces to zero.

Complexity = O(log n)

```
int RecBinarySearch (int a[], int lb, int ub, int item)
{
    if (lb <= ub)
    { int mid = (lb + ub) / 2 ;
      if (item == a[mid])
          return mid;
      else if (item < a[mid])
          return RecBinarySearch(a, lb, mid-1, item);
      else return RecBinarySearch (a, mid+1, ub, item);
    }
    return -1;
}
```

mid = (first + last) / 2 = (0 + 4) / 2 = 2 ✓

1	2	3	5	6
0	1	2	3	4

✓5✓

Comparing to mid if yes then searching is completed but if greater then.

⊙ Algo for linear Search :-

(i) $i = 0$
(ii) if $i > n$; go to step 7 (n → max size)
(iii) if $A[i] = x$; go to step 6
(iv) $i = i + 1$
(v) Go to step 2
(vi) Print element x found at i
(vii) Print element not found.
(viii) exit.

Eg.

2	4	6	3	1	5
0	1	2	3	4	5

$i = 0$

$x = 3$ → to search
$n = 6$.

→ $i = 0$
$0 > 6$ false
$A[i] = 3$
$A[0] = 3$
$2 = 3$ false.

$i = i + 1 = 0 + 1$.

$i = i + 1 = 1 + 1$

→ $A[1] = 3$
$4 = 3$ false

→ $A[2] = 3$
$6 = 3$ false

$i = i + 1 = 2 + 1$
$A[3] = 3$
$3 = 3$ true. exit.

⊙ Algo for binary Search :-

(i) Set beg = lb & end = ub & mid = $\left(\frac{lb + ub}{2}\right)$ int.

(ii) Repeat step 3 & 4 while beg <= end & a[mid] != item.

(iii) if item < a[mid] then
set end = mid - 1
else
set beg = mid + 1

(iv) Set mid = int $\left(\frac{beg + end}{2}\right)$ go to step 2.

(v) if a[mid] = item then
set loc = mid
else
set loc = NULL

(vi) exit.

Eg.

2	3	5	8	9
0	1	2	3	4

mid = 2.

2nd (i) $0 <= 4$ true $a[2] != 8$ true.
(ii) $8 < 5$ false.
else
beg = 2+1 end = 4.

(iii) mid = int $\left(\frac{7}{2}\right)$ = 3 ✓ $a[3] != 8$ false
to step 5.

a[3] = item.

8 = 8 true.

DFS.

- BFS

Sorting.

- Sorting is the process of arranging a list of elements in a particular order.

- Bubble Sorting -:

This sorting algorithm is comparision-based algorithm in which each pair of adjacent element is compared & elements are swapped if they are not in order.

Ex -: 10 14 2 11 6

→ 10 14 2 11 6
comp.

→ 10 14 2 11 6 swap.
comp.

→ 10 2 14 11 6 again swap
comp.

→ 10 2 11 14 6
swap

→ 10 2 11 6 14.

1st pass

10 2 11 6 14 — Swap

Pass 1:
- 10 2 11 6 14 — comp
- 2 10 11 6 14
- " "
- " "
- 2 10 6 11 14 .

Pass 2:
- 2 10 6 11 14 — no change — comp
- 2 10 6 11 14 — Swap — comp

Pass 3:
- 2 6 10 11 14 — Sorted order

Complexity = $O(n^2)$

⊙ Insertion Sort :-

→ In insertion sort, every pass moves a element from unsorted portion to sorted portion until all the element are sorted.

(adge vale ko se compare)

⊙ Ex :-

→ 16 15 4 13 2 1
A[0] A[1] A[2] A[5].

Pass 1 :- 16 15 4 13 2 1
comp. (15 < 16) (shift will occur)
{ if ahead elem is bigger than next element then shift occurs }

Pass 2 :- 15 16 4 13 2 1
comp. 4 < 15

Pass 3 :- 4 15 16 13 2 1

Pass 4 :- 4 15 16 13 2 1
shift.
no shift

Pass 5 :- 4 13 15 16 2 1

Pass 6 : 2 4 13 15 16 1

Pass 7 : 1 2 4 13 15 16 → Sorted.

Complexity = $O(n^2)$.

⊙ Selection Sort :-

→ In this sorting list is divided into 2 parts :-

Sorted & unsorted part.

The smallest element is selected from unsorted array & swapped with leftmost element

→ Ex :-

15 , 14 , 2 , 13 , 4

swap

Pass 1 :- 2 , 14 , 15 , 13 , 4.

swap.

Pass 2 :- 2 , 4 , 15 , 13 , 14.

swap

Pass 3 :- 2 , 4 , 13 , 15 , 14

swap.

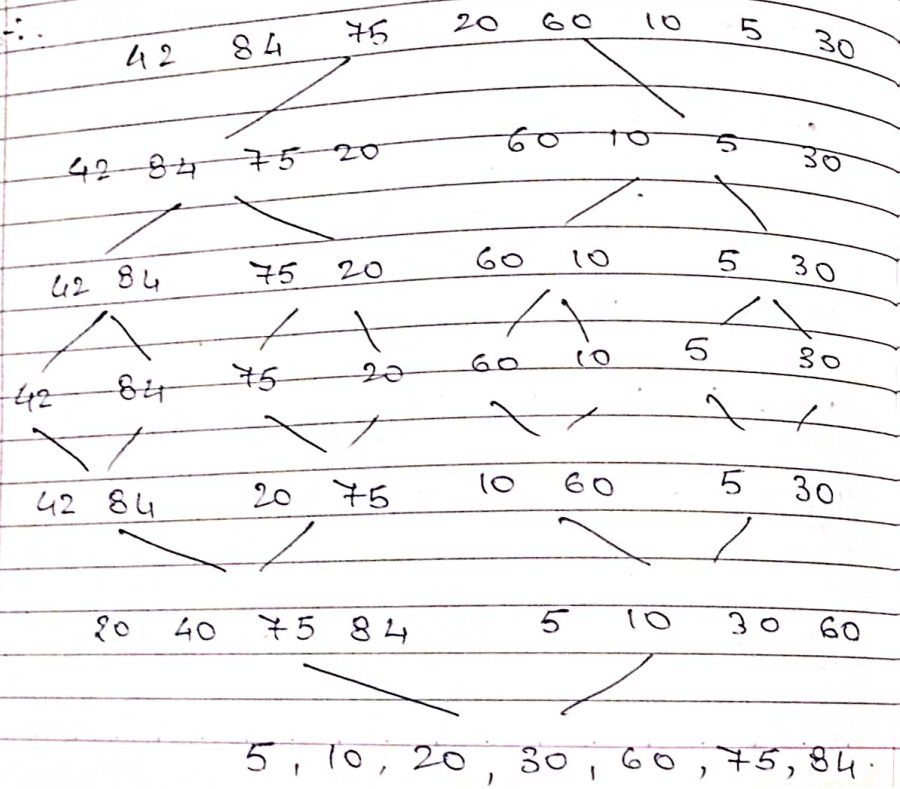
Pass 4 :- 2 , 4 , 13 , 14 , 15 → sorting.

⊙ Merge Sort :-

Merge sort is a sorting technique based on divide & conquer rule.

We divide into equal parts, sort elements & then merge them.

Ex :-



⊙ Quick Sort :-

In this a large array is partitioned into 2 arrays one of which holds the value smaller than the specified value (pivot) based on which partition is done & another part hold value greater than pivot element.

Then quick sort call itself recursively twice to sort the sub arrays.

Partition 1st	Key value	Partition 2nd
Key > values.	pivot.	Key < values.

Ex :- 42, 84, 75, 20, 60, 10, 90, 50, 5, 30?

i j

max size = 10.

Let's take Key value 1st elem → 42.

→ if a[j] < Key if not the j++.
if true i++.

42 < 84 false.

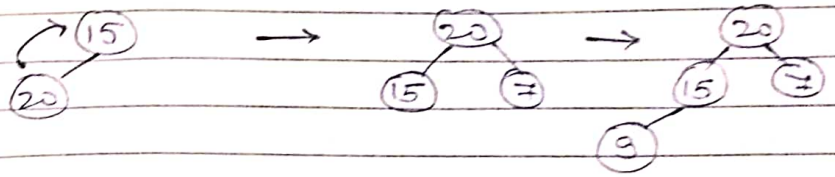
42 84 75 20

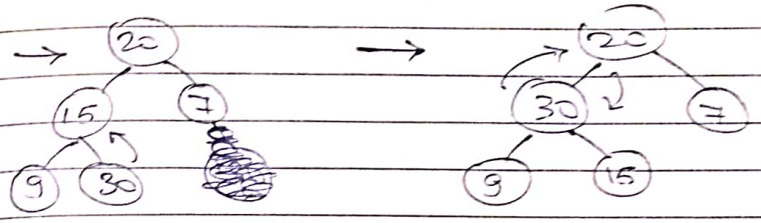
i j → [j]

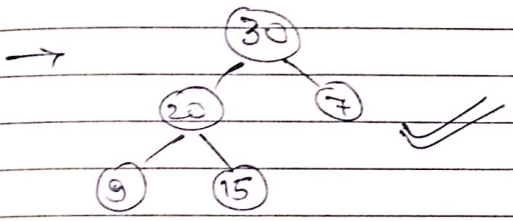
Heap Sort

Q. | 15 | 20 | 7 | 9 | 30 | max heap.

(i)







| 30 | 20 | 7 | 9 | 15 |

deletion.

last element is ~~sorted~~ swapped.

| 15 | 20 | 7 | 9 | 30 |

we have to do rotation.

| 20 | 15 | 7 | 9 | 30 |

| 9 | 15 | 7 | 20 | 30 |

| 15 | 9 | 7 | 20 | 30 |

| 7 | 9 | 15 | 20 | 30 |

sorted ✓

Shell Sort

Q. 23 29 15 19 31 7 9 5 2 ?

Hashing

- Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.

- Search Key ⇒ 24, 52, 91, 67 table size = 10.

By K mod Tsize or Division form.

24 mod 10 = 4 → hash value.

24 / 10 = 2 rem ⇒ 4

0	
1	91
2	52
3	
4	24
5	
6	
7	67
8	
9	

52 → 2
91 → 1
67 → 7

- **Hash function :-**
- **Division :-** K mod n.
- **Mid Square →** 1 2 3
  ↓
  2^2 → 4th position.
- **Folding :-** 123 456 → 123
  456
  579 → pos^n.

(*) **Collision :-**

→ Agr 62 & 52 to dono same pos^n m aaenge to isko bolte h collision.

→ Cluster is the collection of those collided values.