=> Unit-3 :- Decrease and Conquer :-

* The decrease and conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and solution to its smaller instance.

* It can be done in both Top-Down (or) Bottom-Up approach

* Incremental approach :-

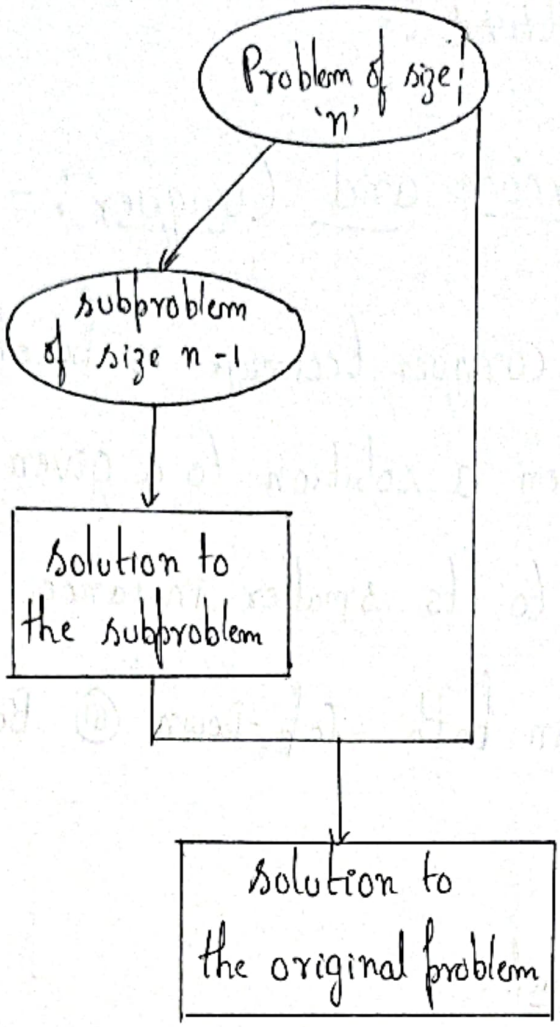
Starting with a solution to smallest instance of problem, and then moving iteratively upwards.

=> Three major variations of Decrease and Conquer :-

(i) Decrease by a constant :-

Size of an instance is reduced by the same constant on each iteration of algorithm.

Typically, this constant is equal to 1.



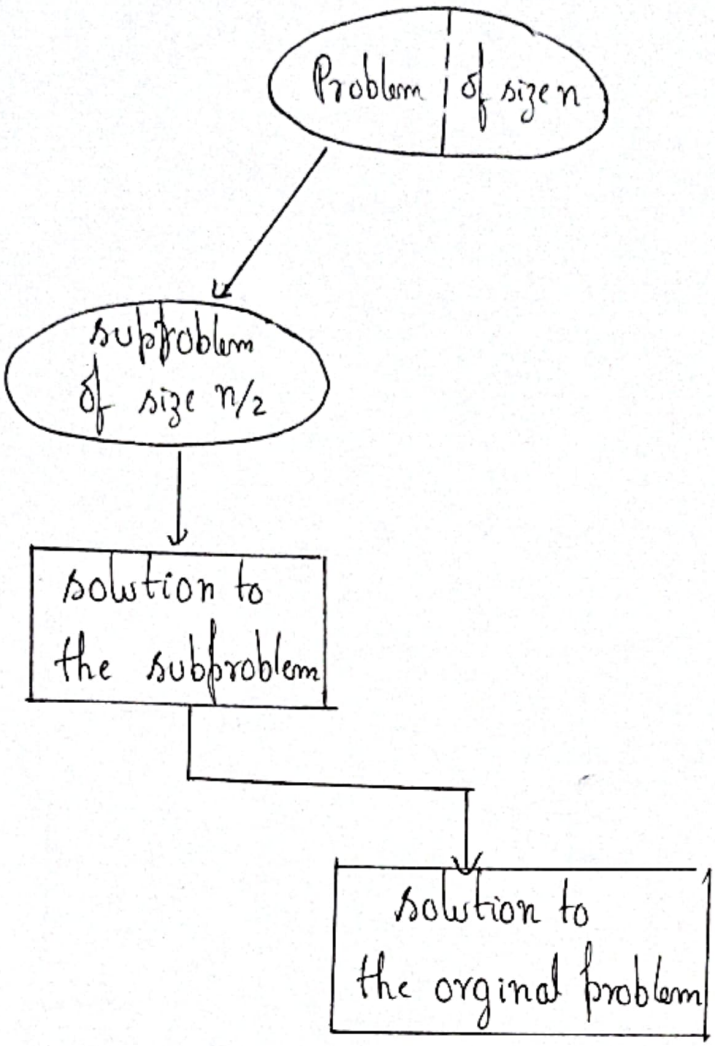
Computing a^n :-

$$f(n) = \begin{cases} f(n-1) \cdot a, & \text{if } n > 0 \\ 1, & \text{if } n = 0 \end{cases}$$

(ii) Decrease-by-constant factor :-

* Reducing a problem instance by the same constant factor on each iteration

* Typically, this factor is equal to two. (Decrease-by half)



$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{, if } n = 0 \end{cases}$$

(iii) Decrease by variable size factor :-

The size-reduction pattern changes from one iteration of algorithm to another.

Ex :- Euclid's GCD Algorithm.

Decrease by variable size

Ex:- GCD of 2 numbers

gcd(60, 42)

↓

gcd(42, 18)

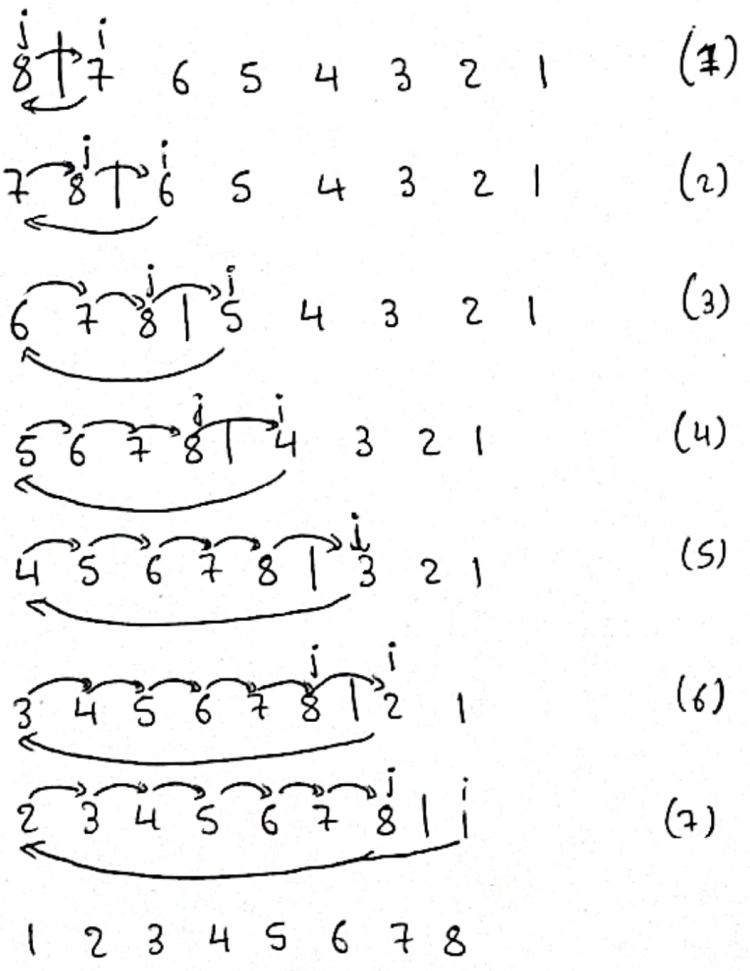
↓

gcd(18, 6)

↓

gcd(6, 0)

=> Insertion Sort :- Inplace & Stable Algorithm



Algorithm :-

```
InsertionSort(A[0....n-1])
//I/p :- An array A[0...n-1] of n elements
//o/p :- The sorted array in ascending order

for i ← 1 to n-1
    t ← A[i]
    j ← i-1
    While (j ≥ 0 And A[j] > t
        A[j+1] ← A[j] ; j ← j-1
    end while
    A[j+1] ← t
end for
```

Analysis :-

Input Size ⇒ n

Basic ~~Comparison~~ operation ⇒ Comparison

⇒ Best Case Scenario :-

When the array is already sorted in ascending order, then only 1 comparison occurs in each iteration

$$= \sum_{i=1}^{n-1} 1$$

$$\therefore C_{Best}(n) = n-1 \in \theta(n)$$

⇒ Worst Case Scenario :-

When the array is sorted in descending order

Maximum number of comparison made in each iteration

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} i-1+1$$

$$= \sum_{i=1}^{n-1} i$$

$$= \frac{n(n-1)}{2}$$

$$\therefore C_{worst}(n) \in \theta(n^2)$$

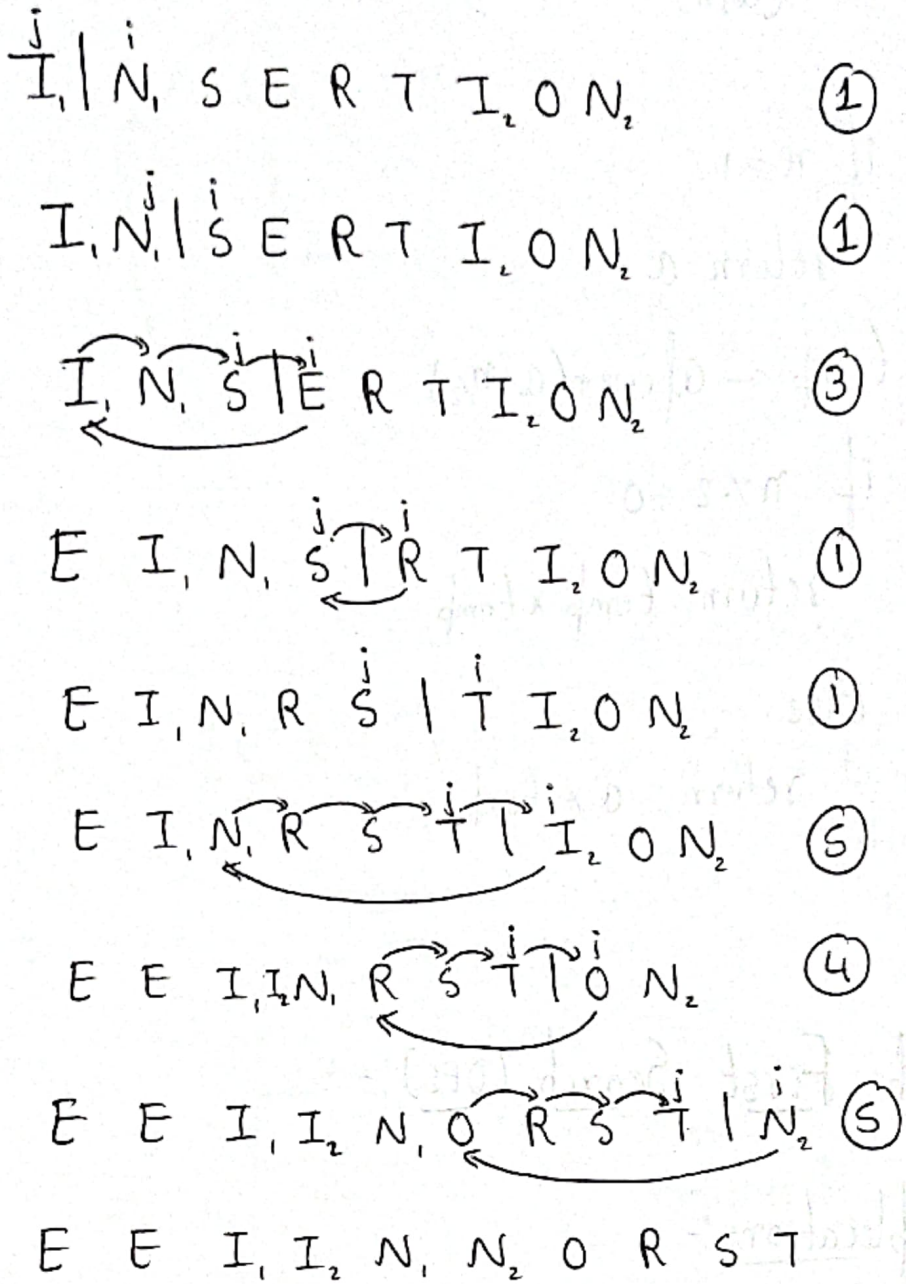
⇒ Average Case :-

The input array is some random input.

$$C_{avg}(n) = O(n^2)$$

Ex:-

INSERTION



⇒ Computing a^n using Decrease & Conquer :-

```
apowern(a,n)
{
    if n = 1
      return a
    temp ← apowern(a, n/2)
    if n%.2 = 0
      return temp x temp
    Else
      return a x temp
}
```

⇒ Depth First Search (DFS):-

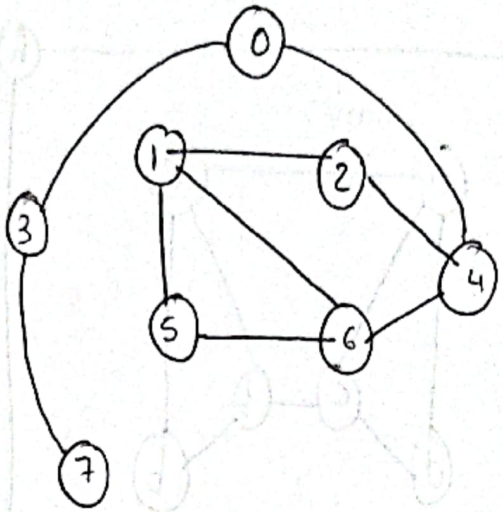
Applications :-

(i) To check wether graph is connected (or) not

(ii) To check cycle in a graph

(iii) To check the attriculation point of a graph

(The vertex on removal of which, graph becomes disconnected.



DFS :- 0 3 7 4 2 1 5 6

Visited array :

0	1	2	3	4	5	6	7
1	6	5	2	4	7	8	3

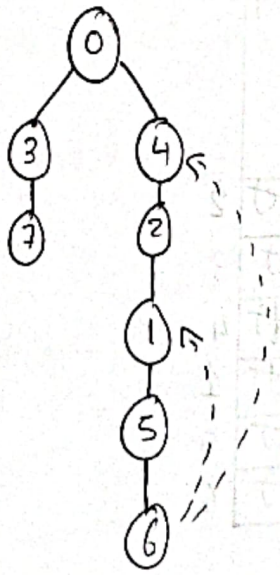
→ Order in which vertices are visited.

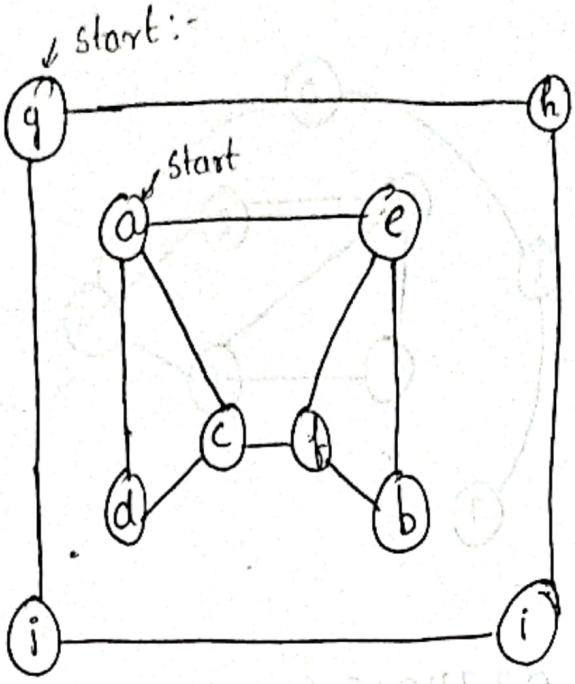
Stack :

~~dfs(6)~~	3
~~dfs(5)~~	4
~~dfs(1)~~	5
~~dfs(2)~~	6
~~dfs(4)~~	7
~~dfs(7)~~	1
~~dfs(3)~~	2
~~dfs(0)~~	8

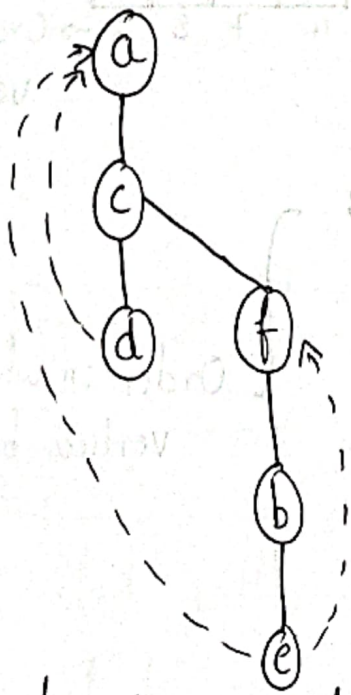
Order in which vertices became dead end

Spanning Tree :-

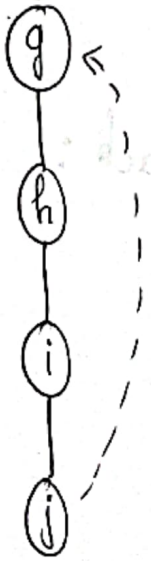




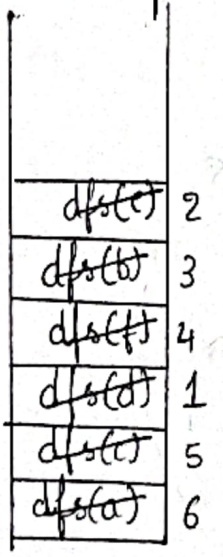
Spanning Trees :-

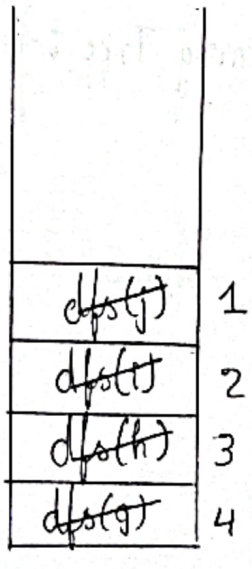


Order of Visit :- a c d f b e



Order of Visit :- g h i j





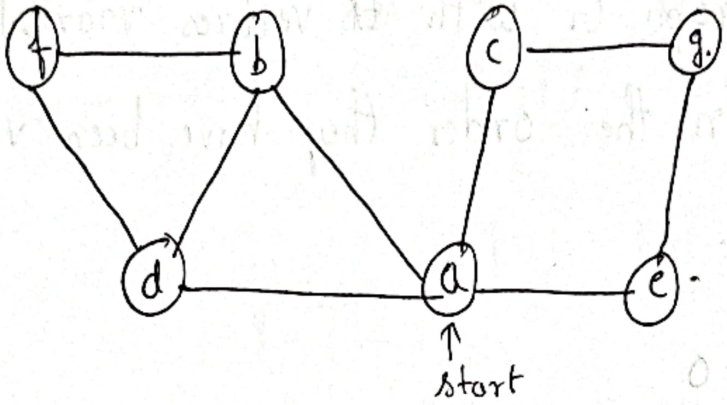
Algorithm :-

```
DFS (G)
// Implements a depth first search traversal of a given graph.
// I/P :- Graph G (V, E)
// O/P :- Graph G with its vertices marked with consecutive
integers in the order they have been visited by DFS
traversal.

count ← 0
for each vertex V in V do
    if V is marked with 0
        dfs(V)

dfs(V)
// Visit recursively all the unvisited vertices connecte to
Vertex v by a path and number them in the order they
are encountered via global variable count
count ← count + 1 ; mark v with count
for each vertex w in v adjacent to V do
    if w is marked with 0
        dfs(w)
```

Ex:- For the given graph,

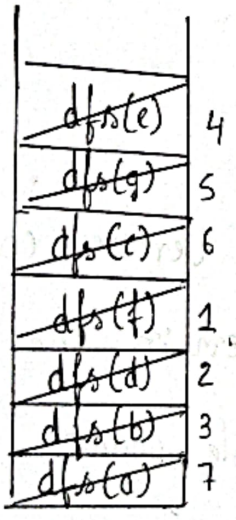
(i) Write adjacency matrix & list

(ii) Starting from vertex 'a', construct dfs tree and give the order in which vertices were reached first time and order in which vertices become dead end



Soln :-

Order of visit = a b d f c g e



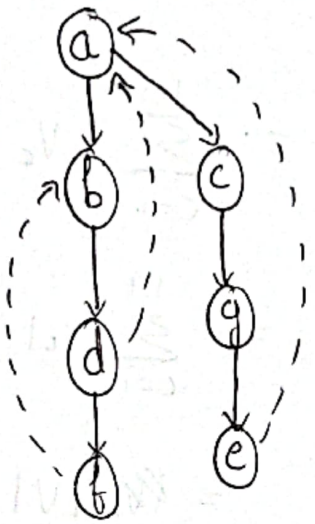
Order of Dead-End :-

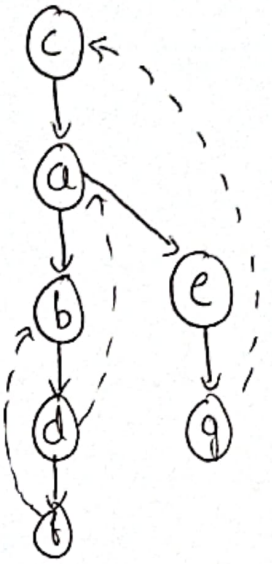
f d b e g c a

Adjacency matrix :-

	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	1	0	0	1	0	1	0
c	1	0	0	0	1	0	1
d	1	1	0	0	0	1	0
e	1	0	0	0	0	0	1
f	0	1	0	1	0	0	0
g	0	0	1	0	1	0	0

Spanning Tree:-





(2) Let 'G' be a graph with 'n' vertices and 'm' edges (i) all its DFS forest (for traversal from different vertices), all have same number of trees. => True

(ii) All its dfs forest will have same number of tree edges & back edges. => True

Soln:- For any connected undirected graph with 'v' vertices then the number of Tree edges in the graph.

$$|E^{(tree)}| = |V| - 1$$

$$|E^{(back)}| = |E| - |E^{(tree)}|$$

$$= |E| - (|V| - 1)$$

$$= |E| - |V| + 1$$

Disconnected Graph :-

$$|E^{(tree)}| = \sum_{c=1}^{|c|} |E_c^{(tree)}|$$

$$= \sum_{c=1}^{|c|} |V_c| - 1$$

$$= \sum_{c=1}^{|c|} |V_c| - \sum_{c=1}^{|c|} 1$$

$$= |V| - |c|$$

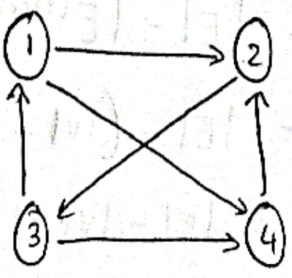
$$|E^{(back)}| = |E| - (|V| - |c|)$$

$$= |E| - |V| + |c|$$

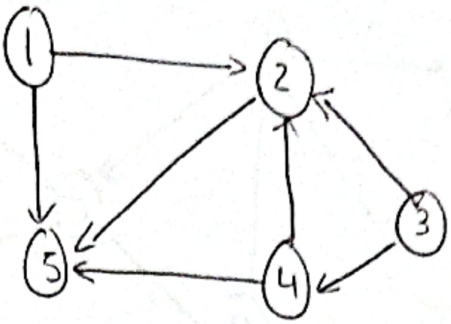
=> Strongly Connected Graphs / Components :-

A graph is called Strongly connected if there is a path from vertex 'u' to vertex 'v'. for each vertices pair $u, v \in V$

Ex:-



=> Strongly Connected



DFS :- 1 2 5
4 2
3 1 5

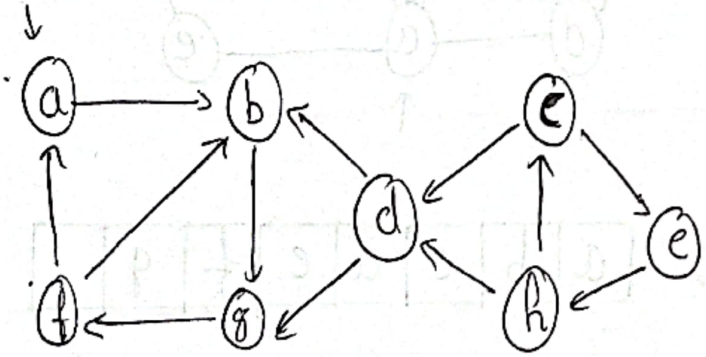
} 3 - components

⇒ Kosaraju's Algorithm :-

⇒ Steps to find strongly connected components :-

(i) Conduct the DFS and note order of dead end

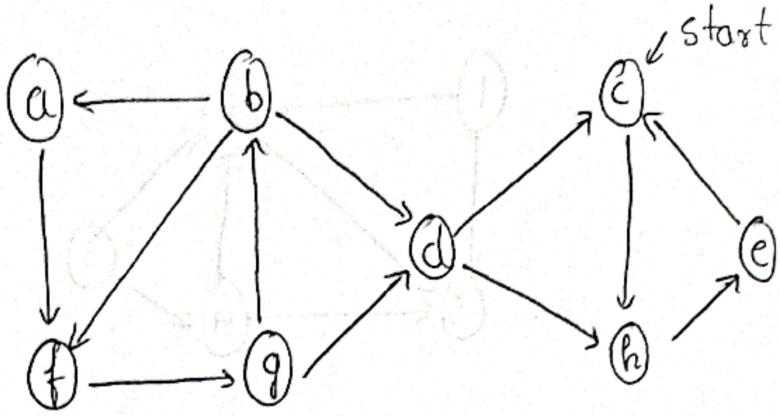
(ii) Change direction of all edges of digraph and ~~the~~ do dfs on the new digraph starting from the vertex that became dead end in last.



DFS :- a b g f (f g b a) → order of dead end
c e h d (d h e c) → order of dead end

Descending order of dead end :- c e h d a b g f

Step 2:-

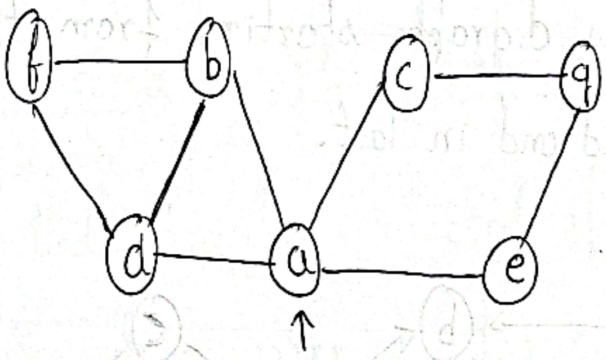


DFS:- c h e (ehc)
d
a f g b (bgfa)

} 3 - Strongly connected components.

⇒ Breadth First Search:-

Ex ①:-

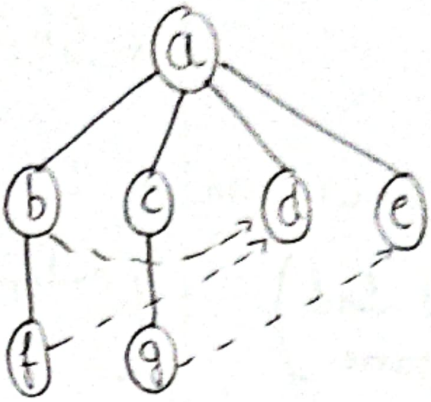


Queue :

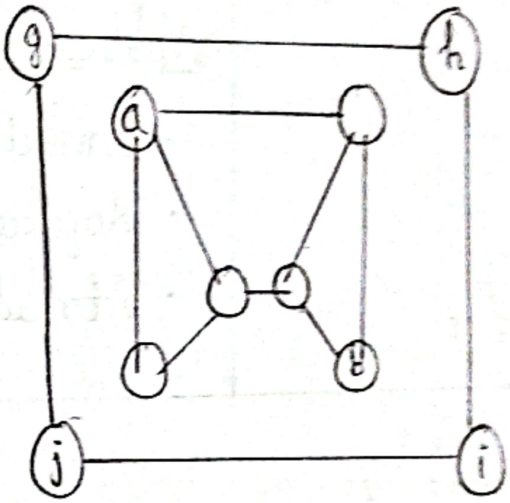
a	b	c	d	e	f	g	

Order of visit :- a b c d e f g

BFS Spanning Tree :-

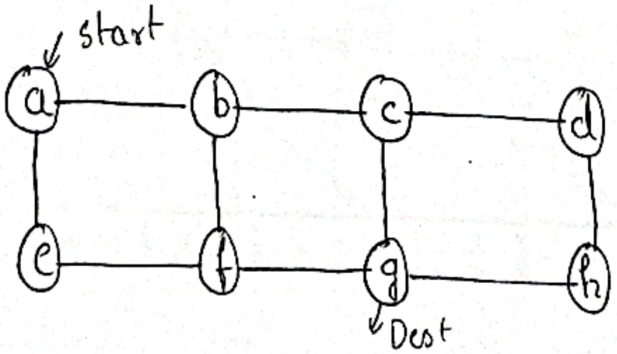


Ex ② :-



BFS	DFS
* Queue is used	* Stack is used
* It is called one-ordering (Order of visit and dead end in same)	* It is Two-ordering (Order of visit and dead end is different)
* It has Tree edge, cross edge & back edge	* It has Tree edge & ~~Cross~~ back edge
→ Application :- • Connectivity • Acyclicity • Minimum path edge	→ Application :- • Connectivity • Acyclicity • Articulation Point

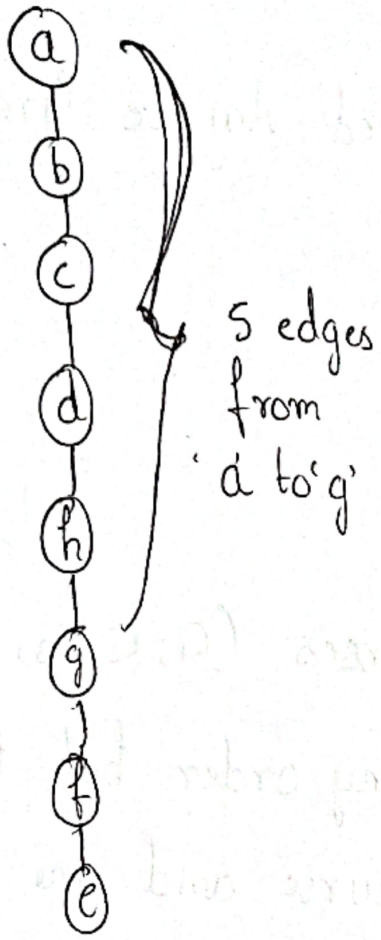
=> Minimum Path Edge :-



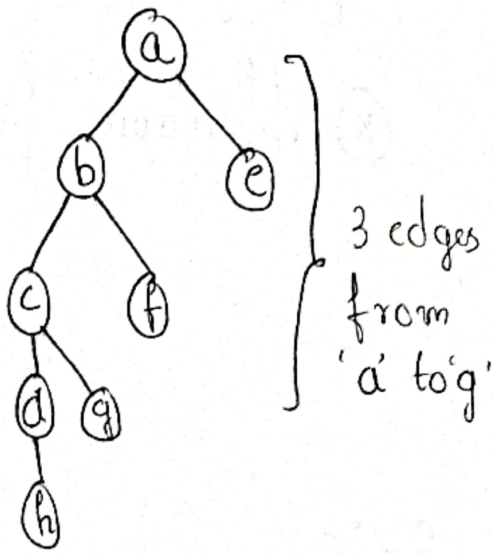
DFS :- a b c d h g f e

BFS :- a b e c f g d h

DFS



BFS



(*) The efficiency of BFS & DFS is dependent on the data structure used to implement the graph.

⇒ Topological Sorting :- (Directed graph)

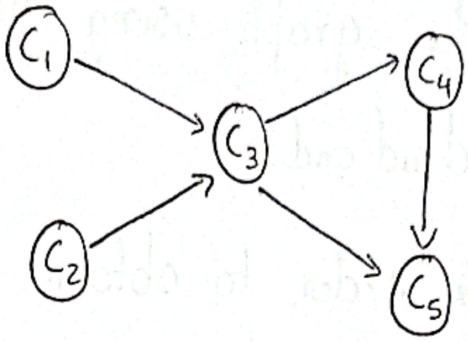
⊛ Applicable for Directed Acyclic Graph (DAG)

Ex :-

Consider a set of 5 courses $(C_1, C_2, C_3, C_4, C_5)$, the courses can be taken in any order but there are prerequisites for ~~each~~ some course and one course per term ~~in each~~ in max.

Course	Prerequisites
C_1 & C_2	None
C_3	C_1 & C_2
C_4	C_3
C_5	C_4, C_3

The graph is written as,



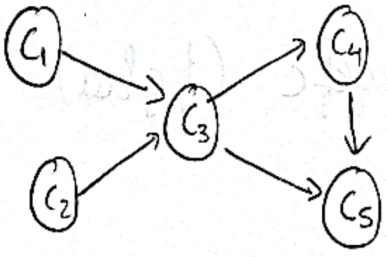
Topological ordering :-

$C_1\ C_2\ C_3\ C_4\ C_5$

(or)

$C_2\ C_1\ C_3\ C_4\ C_5$

⇒ Topological Sorting using DFS :-



DFS :- $C_1\ C_3\ C_4\ C_5$ $(C_5\ C_4\ C_3\ C_1)$

C_2

Order of dead end :- $C_5\ C_4\ C_3\ C_1\ C_2$

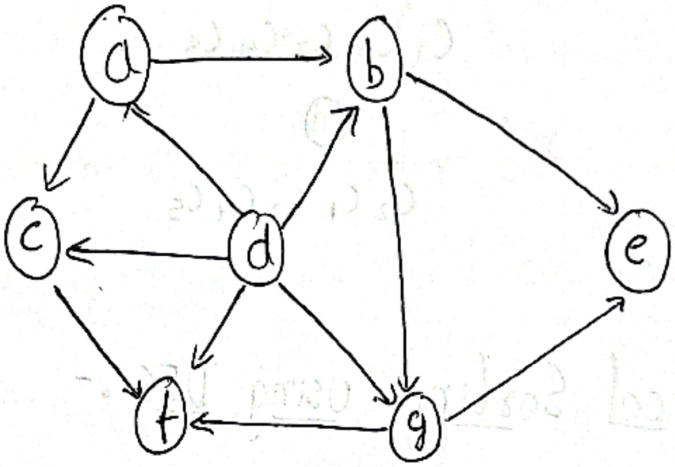
∴ Topological Sorting :- $C_2\ C_1\ C_3\ C_4\ C_5$

=> Topological Sorting using DFS :-

(i) Traverse the graph using DFS & note down order of dead end

(ii) Reverse the order to obtain Topological sorting

Ex(2) :-

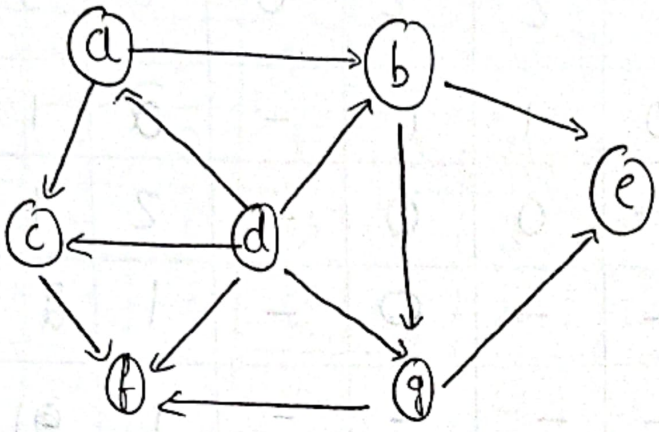


DFS :- abgefc (fgbca)
d
e

Order of dead end :- efgbcade

∴ Topological Sorting :- dacbgfe

⇒ Source removal method for Topological Sorting:-



Adjacency Matrix:-

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	0	0	0	0	1	0	1
c	0	0	0	0	0	1	0
d	1	1	1	0	0	1	1
e	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0
g	0	0	0	0	1	1	0
Indegrees	1	2	2	0	2	3	2

(*) There should be atleast one vertex with zero $^{in}_{r}$degree

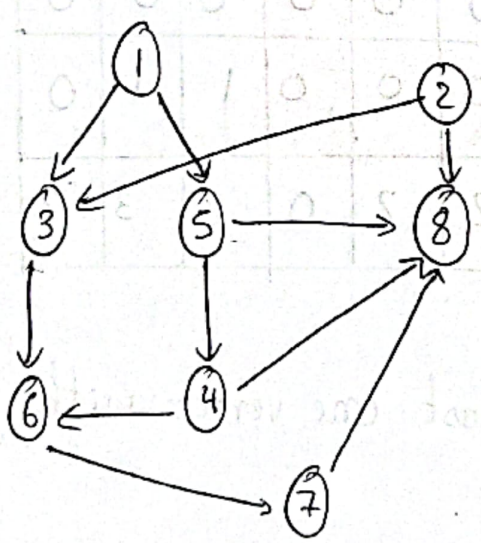
Indegrees :-

a	b	c	d	e	f	g
1	2	2	0	2	3	2
0	1	1	-	d	1	1
-	0	0	-	2	1	1
-	-	0	-	1	d	0
-	-	-	-	1	0	0
-	-	-	-	0	0	-

Queue :- | d | a | b | c | g | e | f |

∴ Topological order :- d a b c g e f

Ex :- ② :-



1	2	3	4	5	6	7	8
0	0	2	1	1	2	1	4
-	0	1	1	0	2	1	3
-	-	0	1	0	2	1	3
-	-	0	0	-	2	1	2
-	-	-	0	-	1	1	2
-	-	-	-	-	0	0	1
-	-	-	-	-	-	0	1
-	-	-	-	-	-	-	0

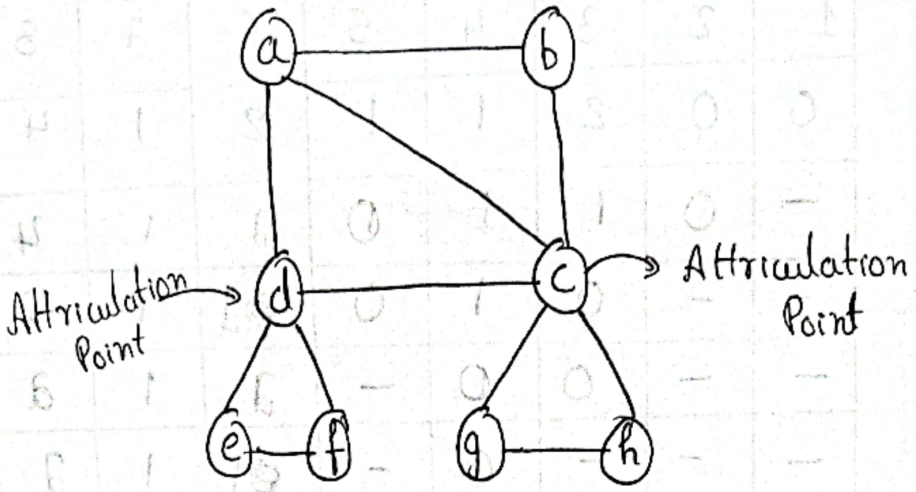
Queue :- | ~~1~~ | ~~2~~ | ~~5~~ | ~~3~~ | ~~4~~ | ~~6~~ | ~~7~~ | ~~8~~ |

Topological order :- 12534678

=) Attriculation Point & Biconnected Components :-

* It is the node in a graph which on removal makes the graph disconnected.

* It is not desirable in graph.



Steps :-

① Traverse the graph using DFS

② While traversing, number each vertex of tree with two numbers

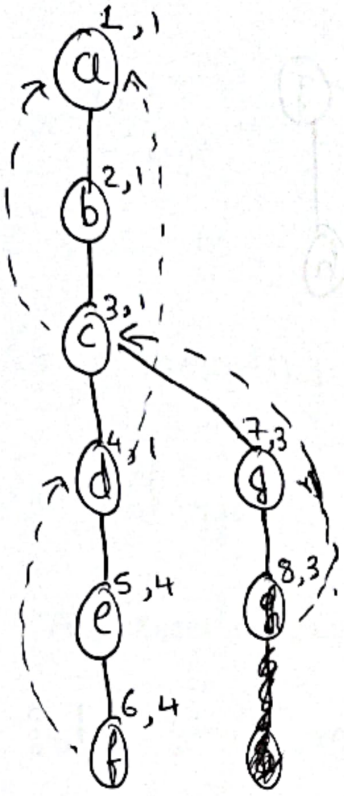
(i) First number (L-Value) is the order in which vertex was visited.

(ii) Second number (R-Value) is the lowest numbered vertex

③ Root is an attriculation point, if it has 2 children

④ Any other vertex other than root is attriculation point if RValue[child] ≥ LValue[Parent]

then Parent is an Attriculation Point



	a	b	c	d	e	f	g	h
L-V	1	2	3	4	5	6	7	8
R-V	1	1	1	1	4	4	3	3

$R[c] < L[b]$ => B is not attriculation point

$R[d] \not< L(c)$

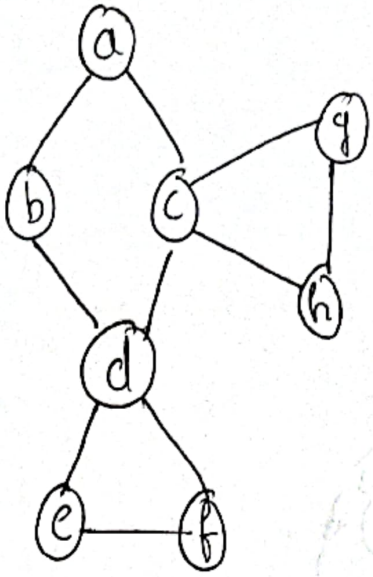
$R[g] \geq L(c)$ => C is attriculation point

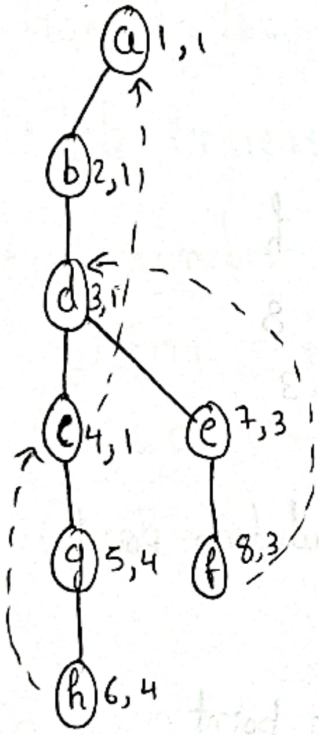
$R[e] \not\geq L(d)$ => d is attriculation point

$R[f] < L(e)$ => e is not attriculation point

$R[h] < L(g)$ => g is not attriculation point

Ex-②:-





	a	b	c	d	e	f	g	h
LV	1	2	4	3	7	8	5	6
RV	1	1	1	1	3	3	4	4

=> Transform and Conquer :-

① Instance Simplification :-

Given instance is simplified (or) transformed into a simple (or) convenient form. Ex :- Presorting

② Representation Change :-

The representation of the instance is changed, i.e Changing Data Structure

③ Problem Reduction :-

Transformation to a form to which the algorithm is already available.

=> Instance Simplification :-

-> Finding Uniqueness of array by presorting

```
Unique(A[0...n-1])
//I/P :- An ordered array A[0...n-1]
//O/P :- True if A is unique false otherwise
    for i ← 0 to n-2 do
        if A[i] = A[i+1]
            return false
    end for
    return True
```

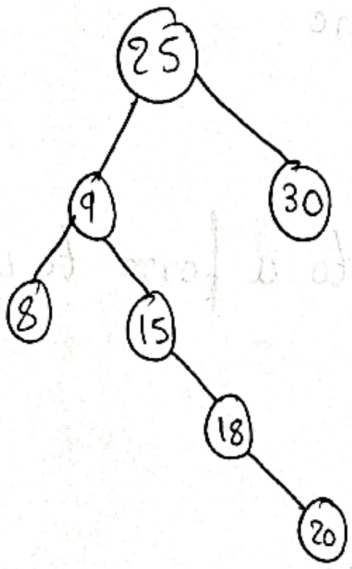
→ Examples for Presorting:-

- Uniqueness array
- Binary Search
- Computing mode in an array

⇒ Representation Change :-

Array :- 25, 30, 9, 15, 18, 20, 8 (No of comparison = 7)

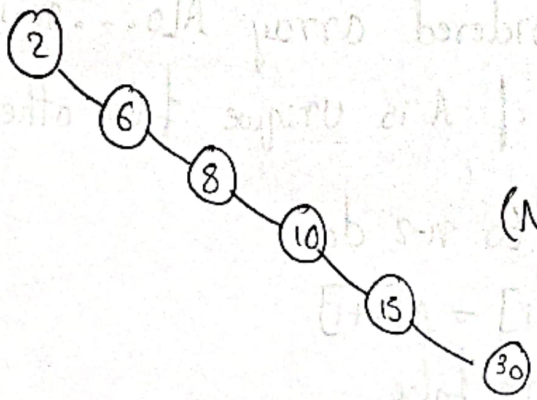
Binary Search :- Tree



(No of comparison = 3)

Issue :-

Array :- 2 6 8 10 15 30 (No of comparison = 6)

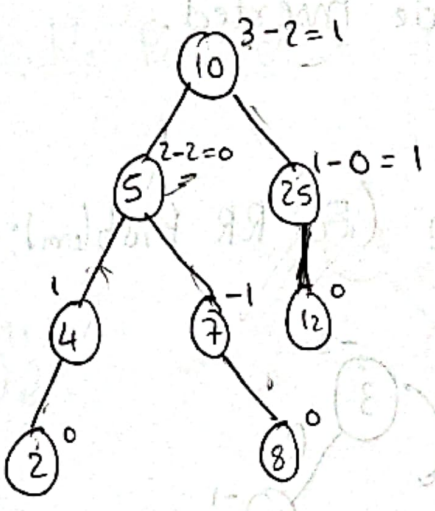


(No of comparison = 6)

→ Converting unbalanced Tree to a balanced Tree :-

→ AVL Tree :-

It is A Binary Search Tree in which balance factor of every node defined as difference b/w height of left and right subtree is either 0 or -1 or +1



⇒ Perfectly balanced AVL Tree

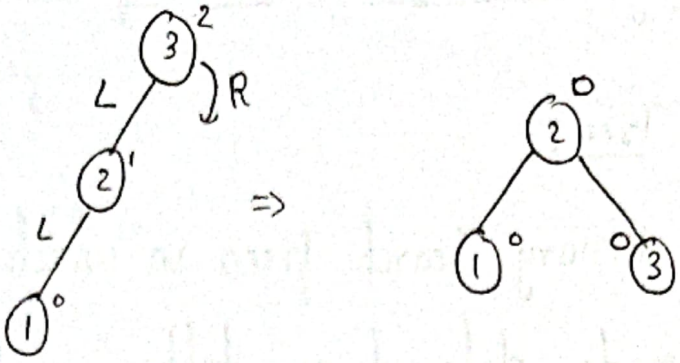
✗ All AVL Trees are Binary Search Tree

→ AVL Rotations / Local Transformations :-

(i) Single Rotation → Left
→ Right

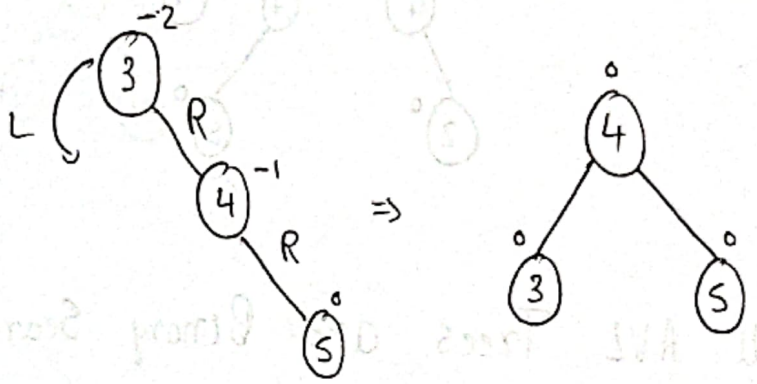
(ii) Double Rotation → Left-Right
→ Right-Left

→ Single Right Rotation (For LL Problem):-

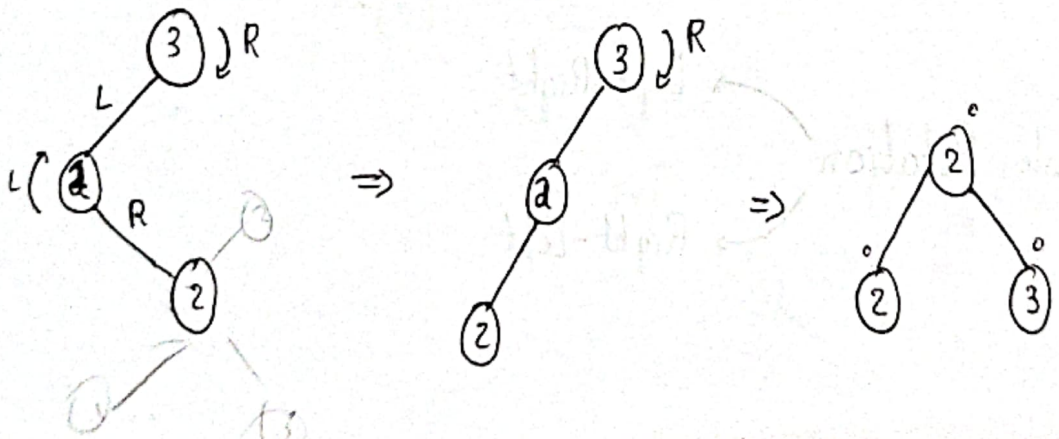


From the node that is imbalanced move 2 steps towards the new node inserted.

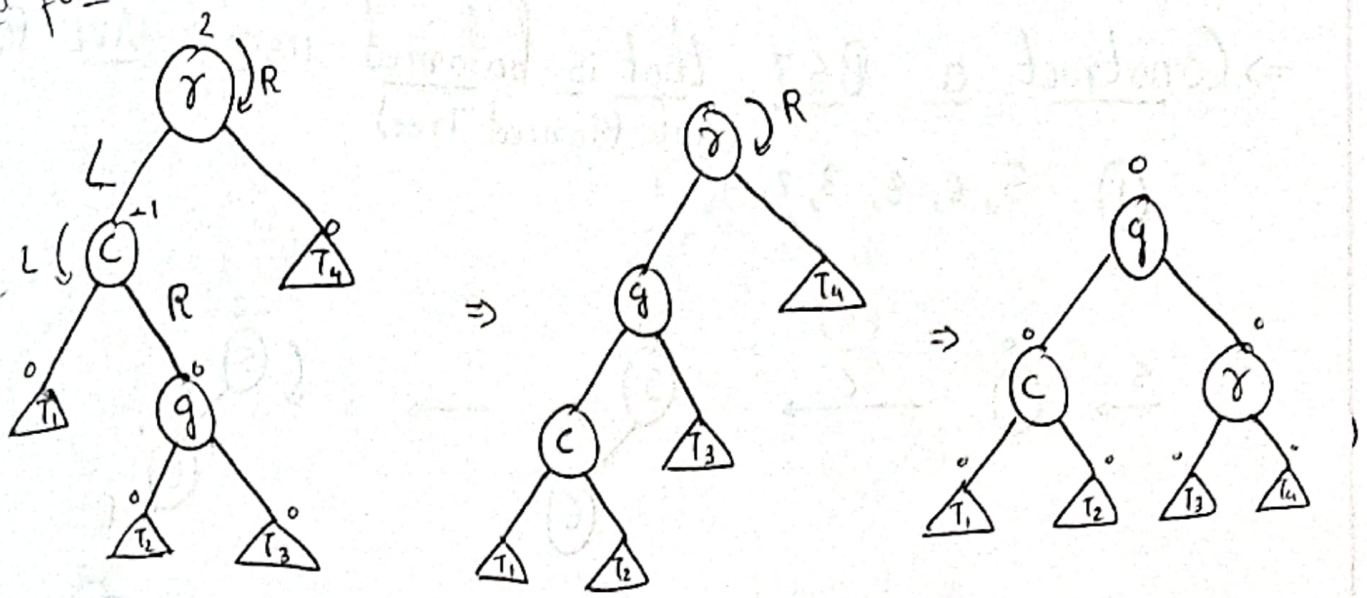
→ Single Left Rotation (For RR Problem):-



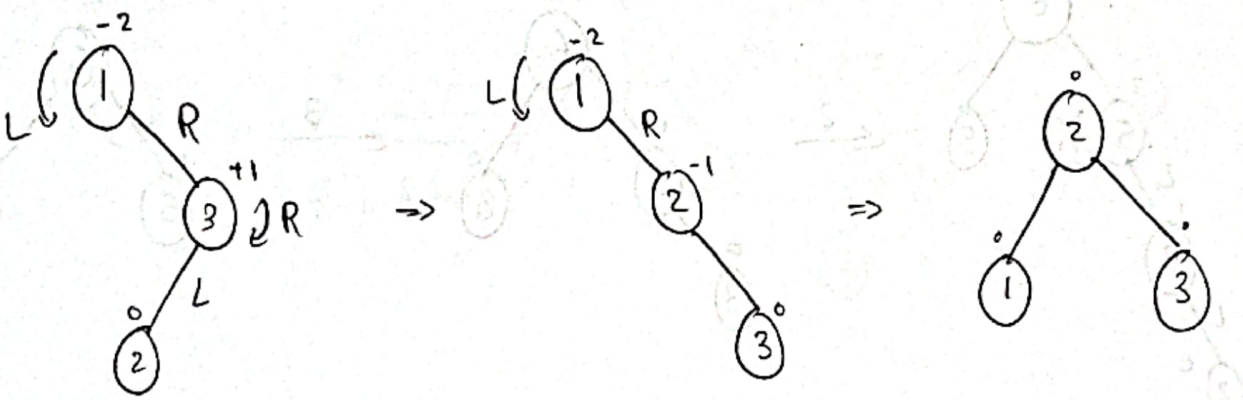
→ Left-Right Double Rotation (For LR Problem):-



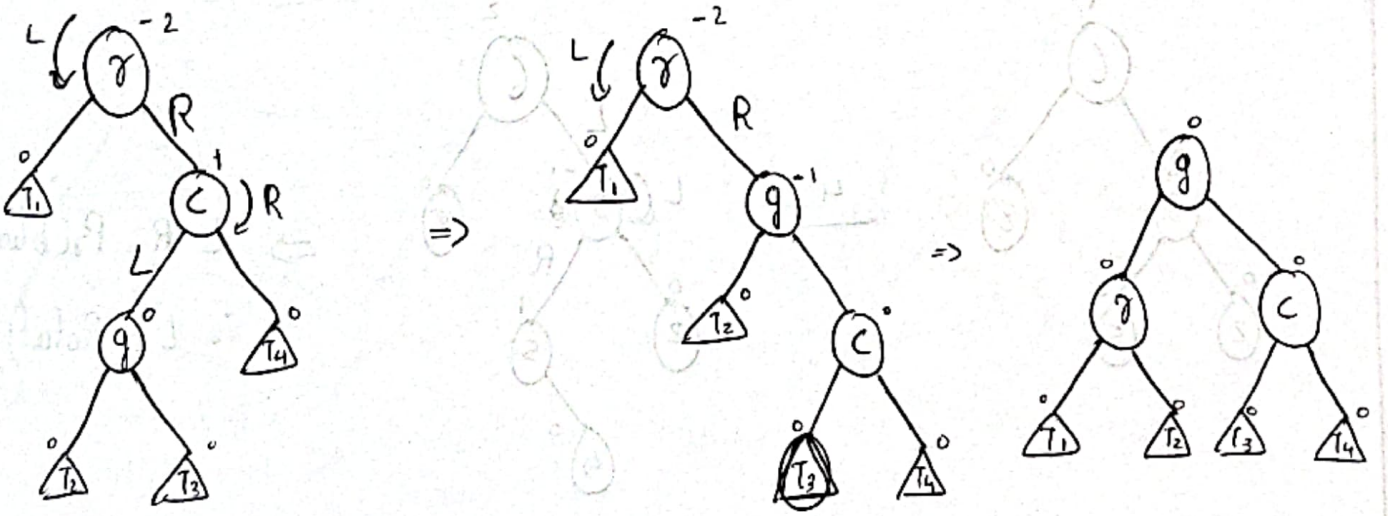
General form :-



→ Right - Left Double Rotation :- (For RL Problem)

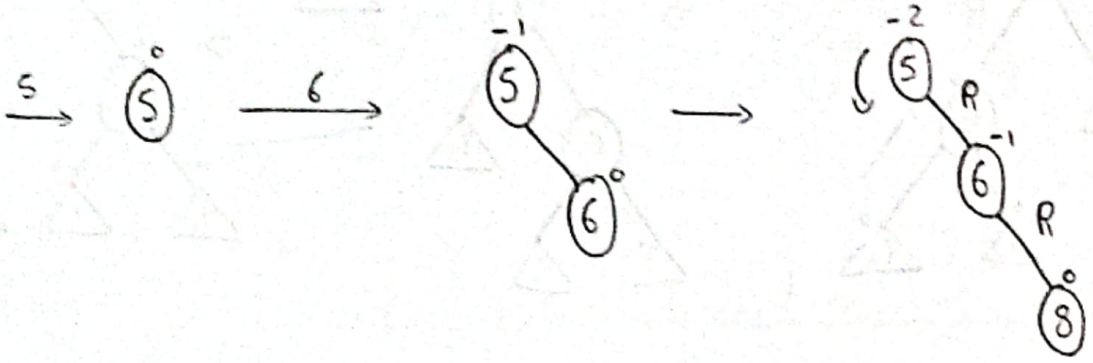


→ General form :-

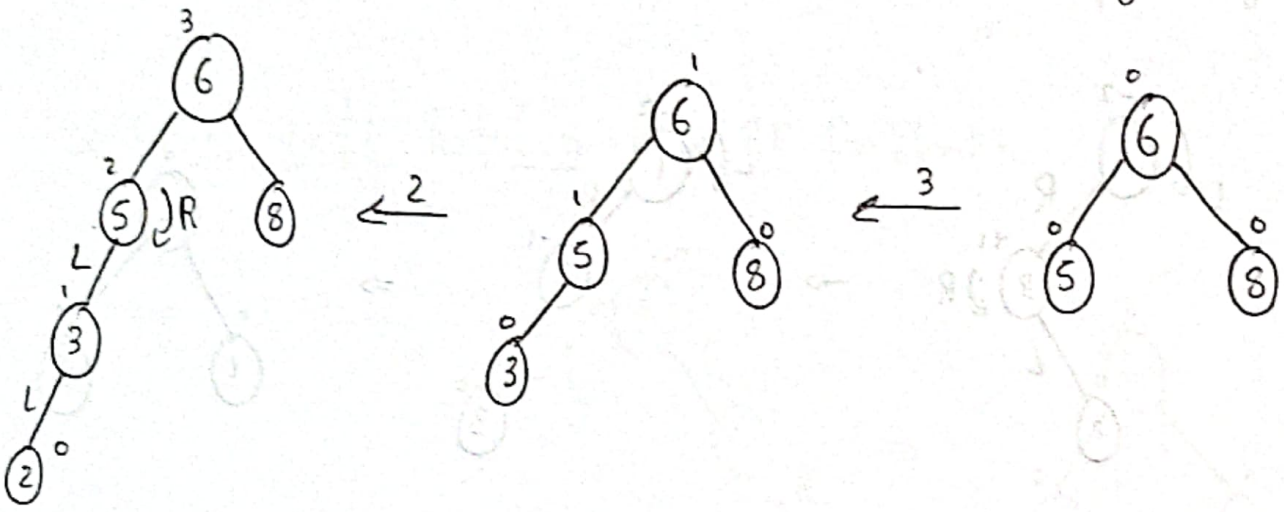


⇒ Construct a BST that is balanced using AVL Rotation
(AVL Balanced Tree)

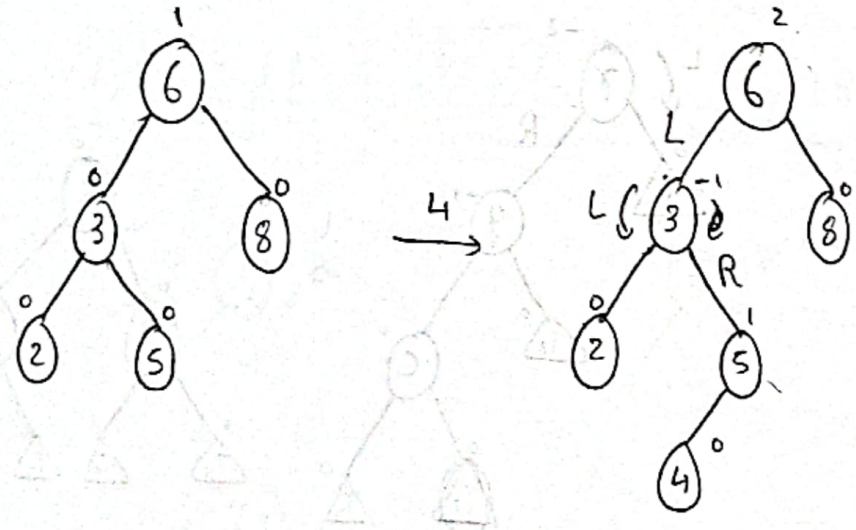
① 5, 6, 8, 3, 2, 4, 7



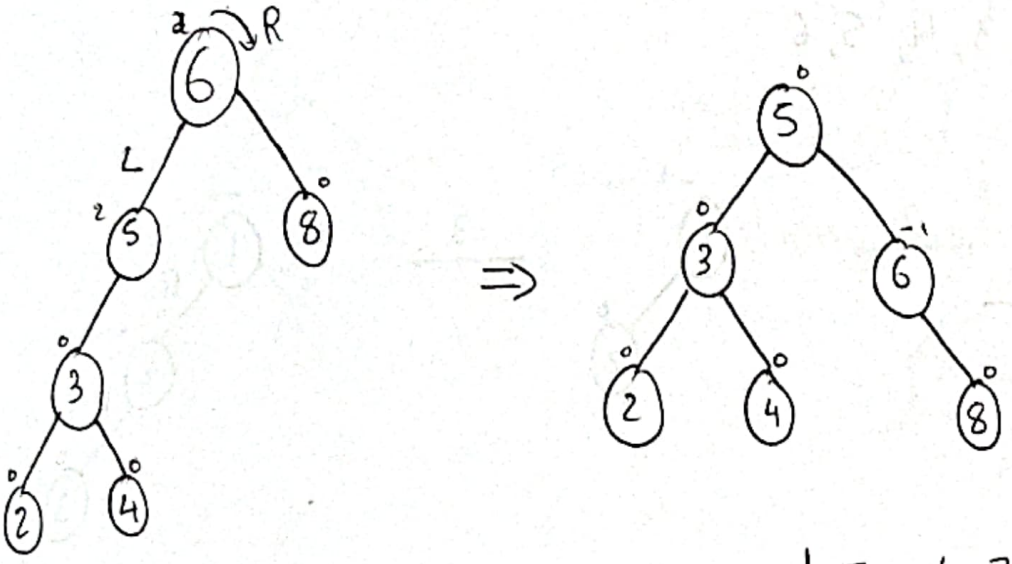
↓↓ LR Rotation of 5



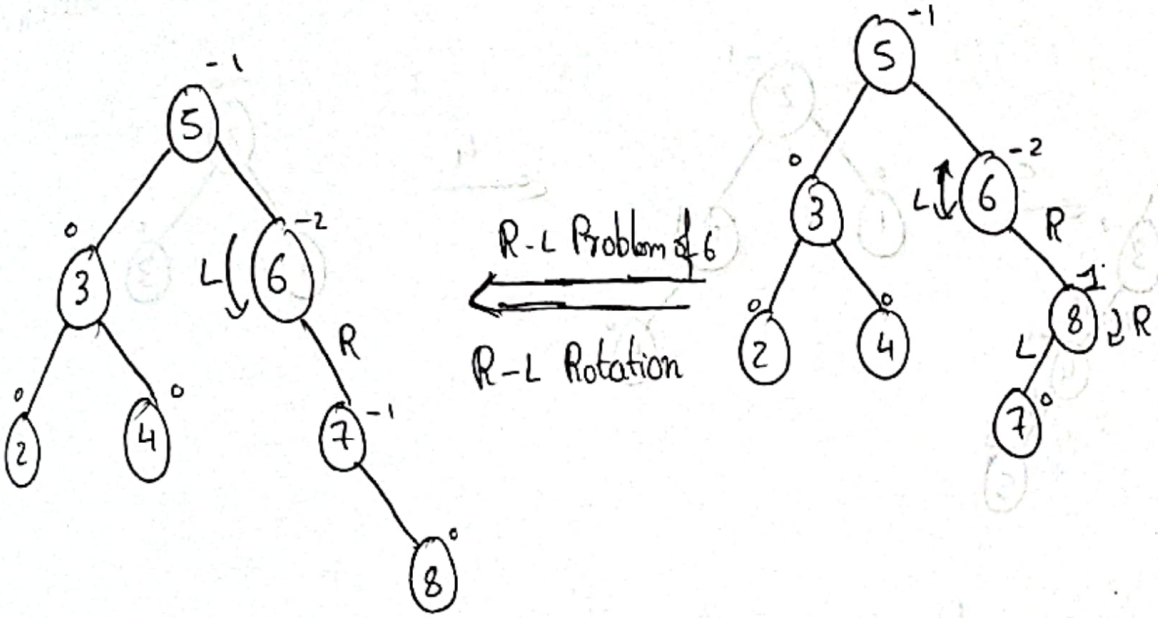
↓↓ LL Problem of 5 (Right Rotation)

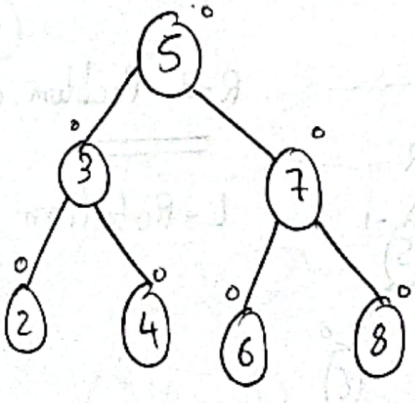


⇒ L-R Problem of 6
∴ LR Rotation



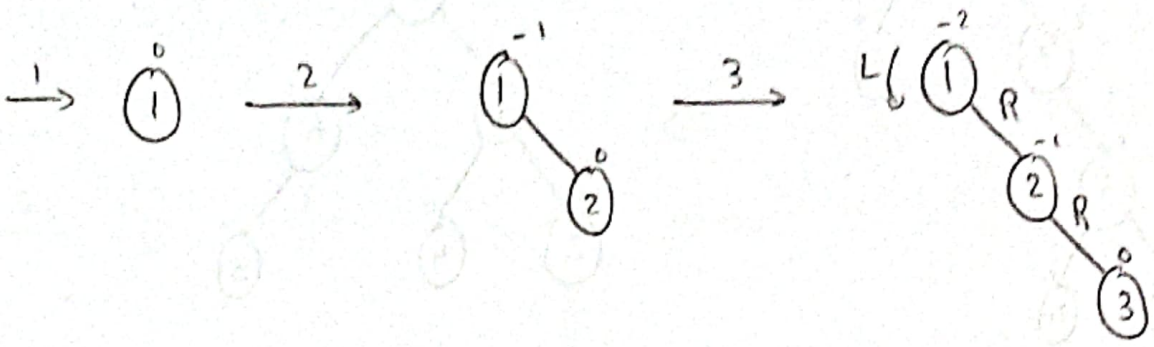
↓ Insert 7



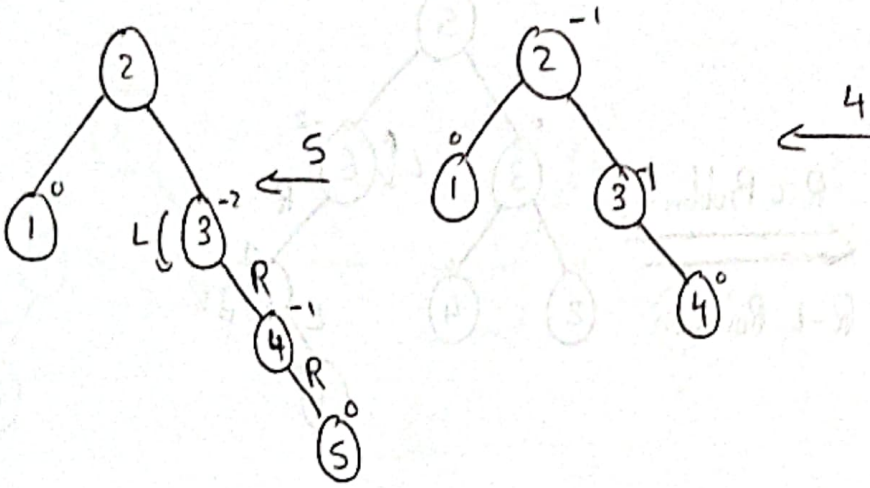


⇒ Fully balanced Binary Search Tree

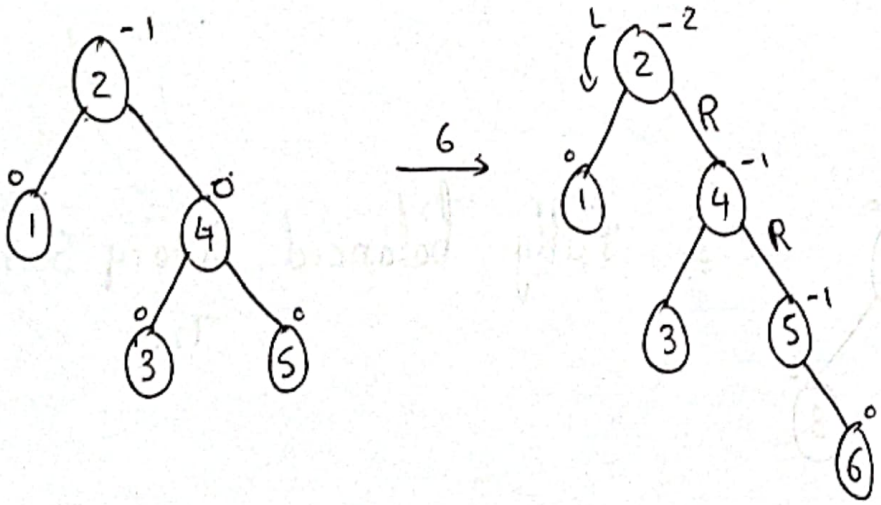
(ii) 1, 2, 3, 4, 5, 6



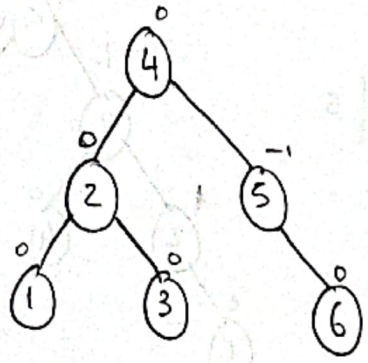
RR Problem of '1'
L - Rotation



RR Problem of '3'
L - Rotation

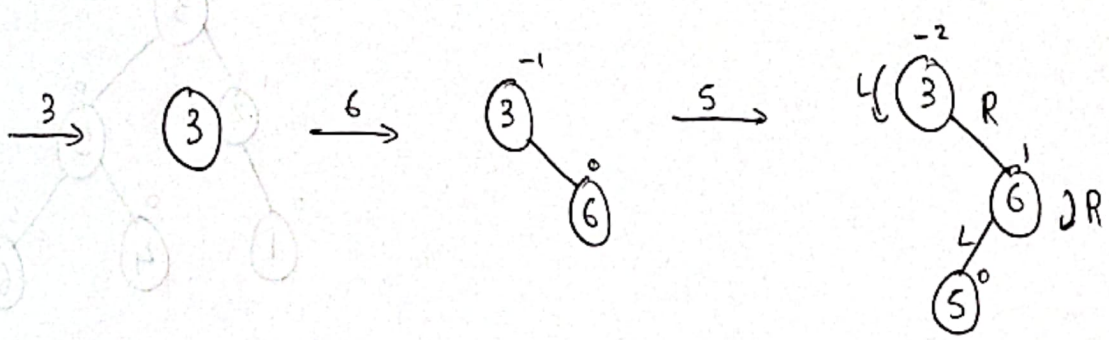


R-R Problem of '2'
L-Rotation

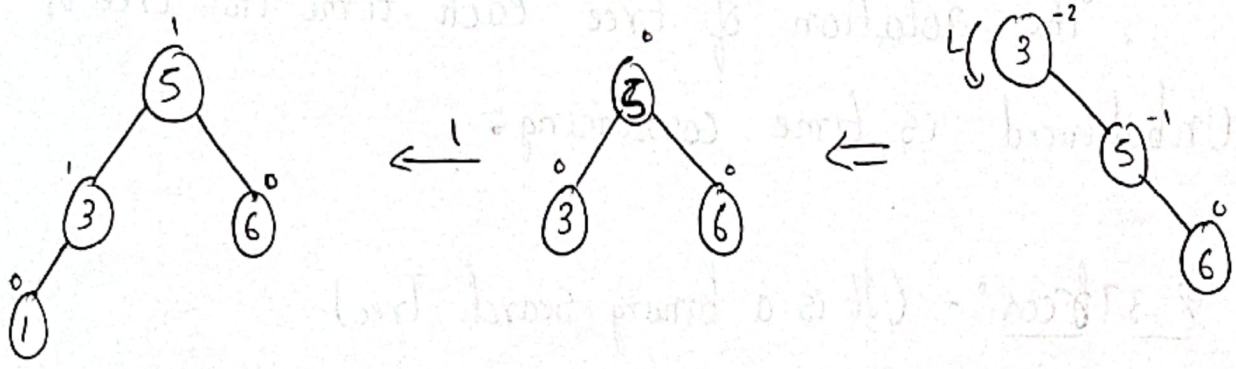


⇒ Fully balanced
Binary Search Tree

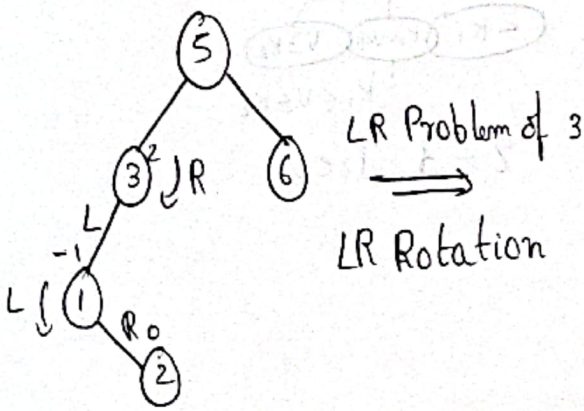
(iii) 3, 6, 5, 1, 2, 4



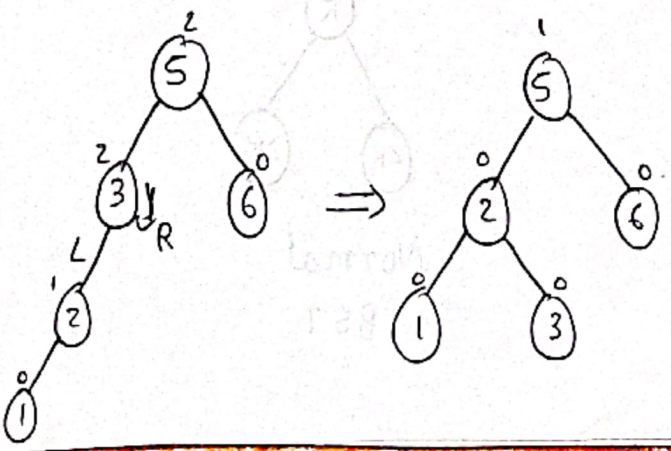
RL Problem of 3
RL Rotation

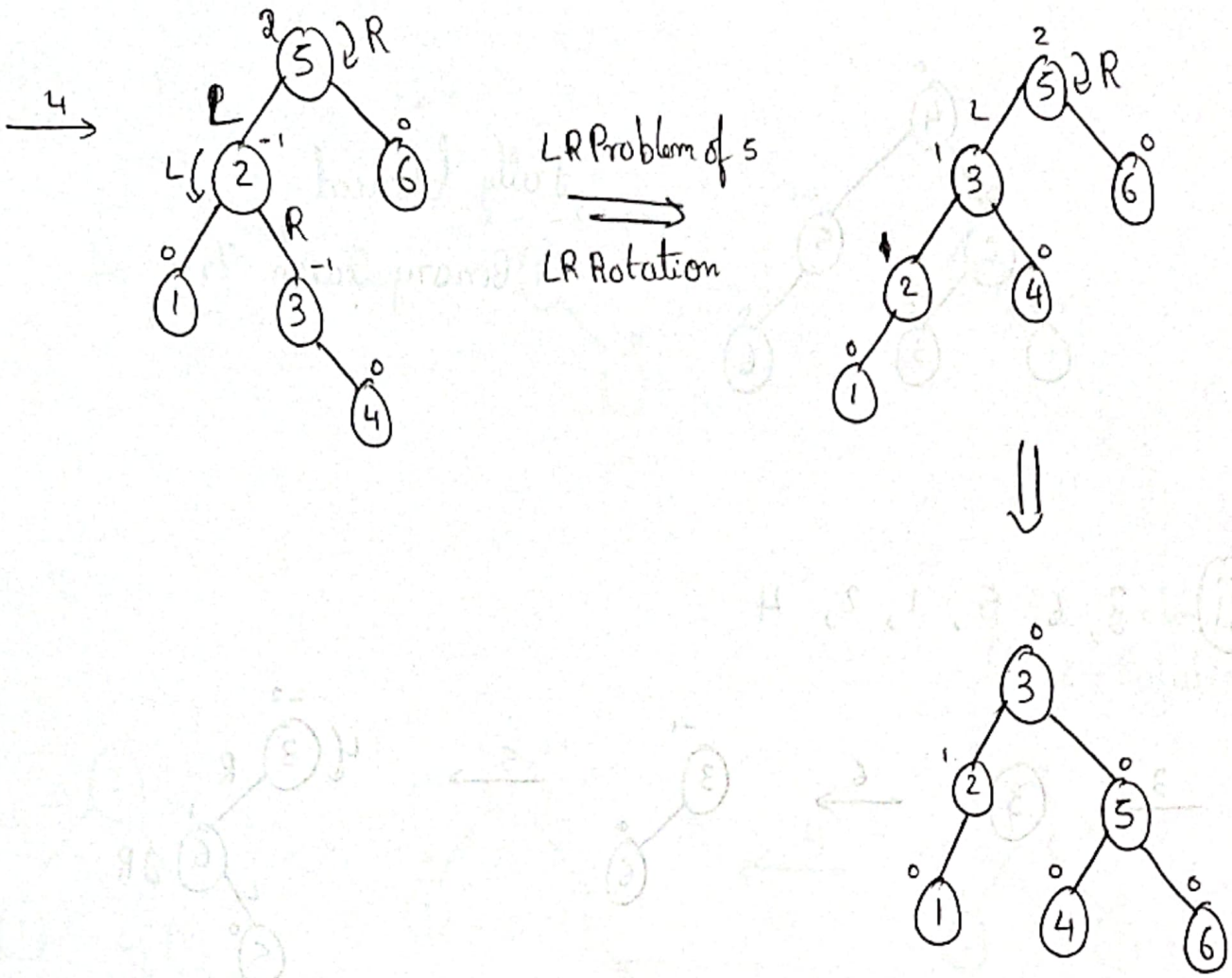


2



LR Problem of 3
LR Rotation

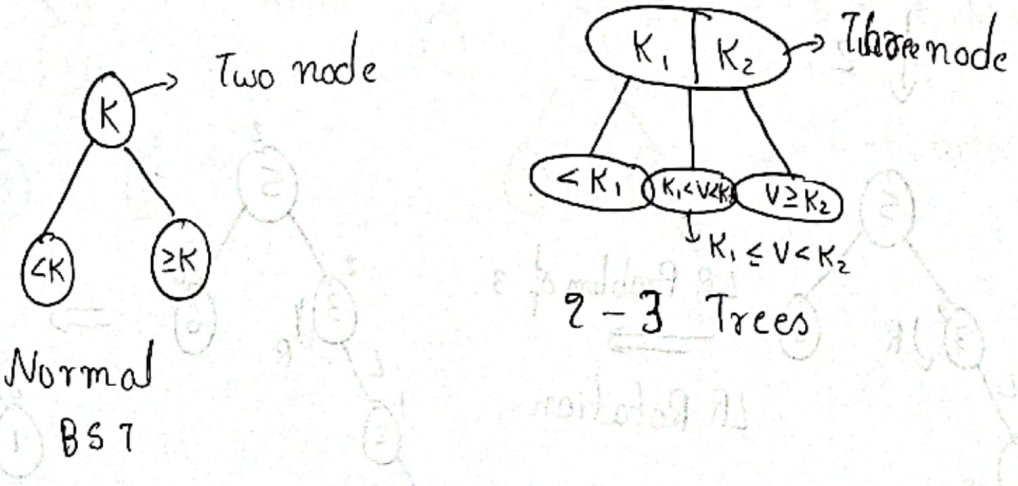




→ Major disadvantage of AVL Trees :-

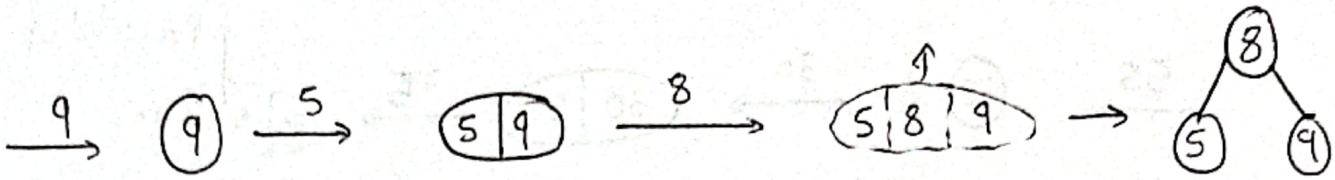
The rotation of tree each time the tree is Unbalanced is time consuming :-

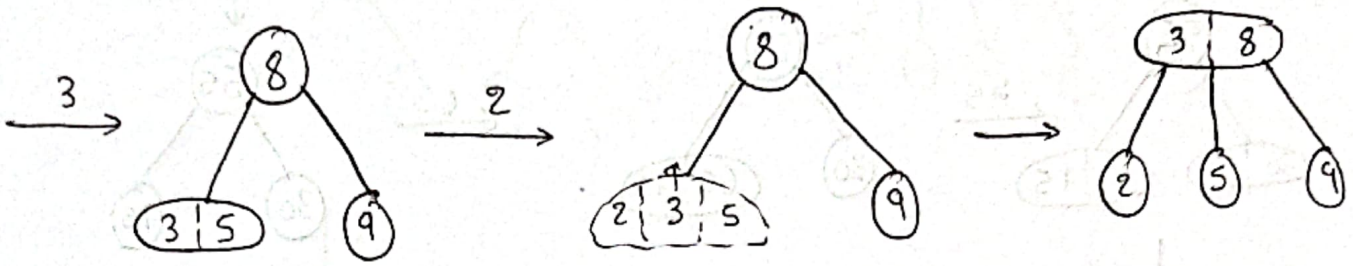
→ 2-3 Trees :- (It is a binary search Tree)

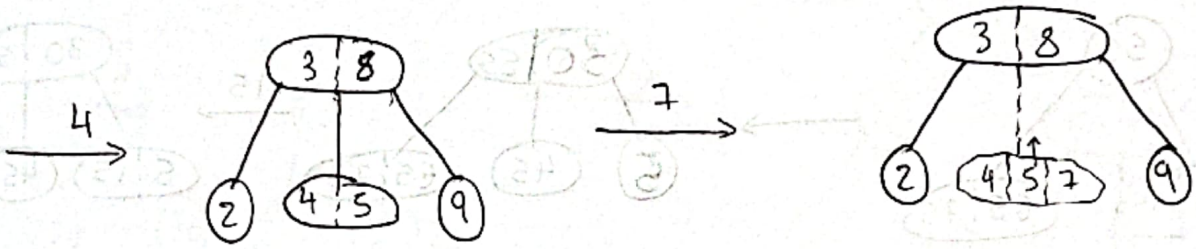


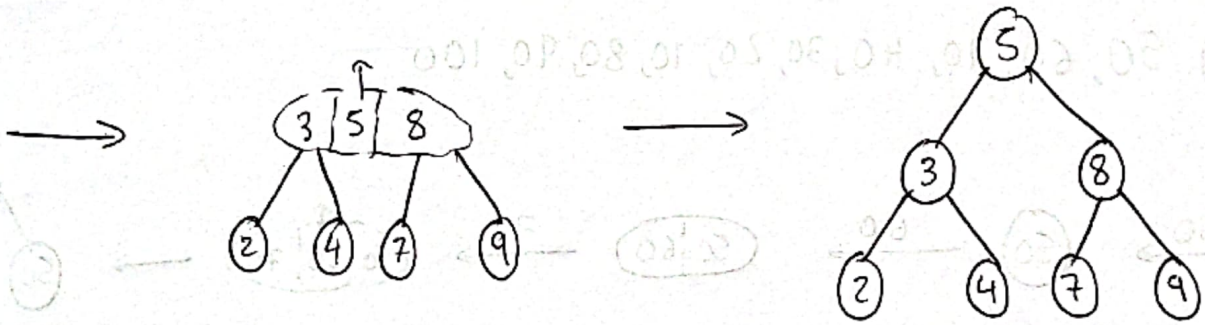
→ Construct a 2-3 Tree :-

(i) 9, 5, 8, 3, 2, 4, 7



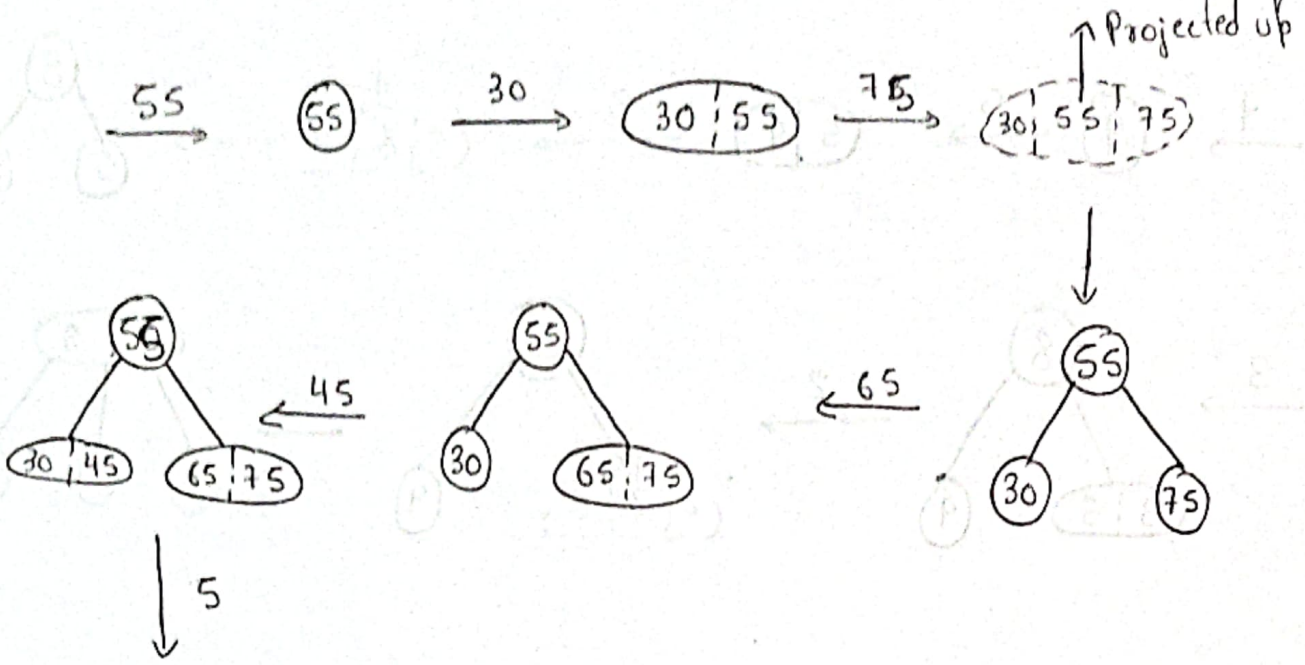


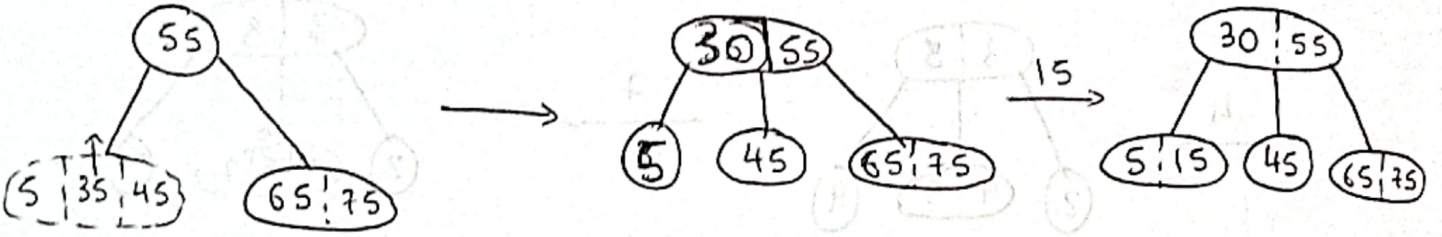




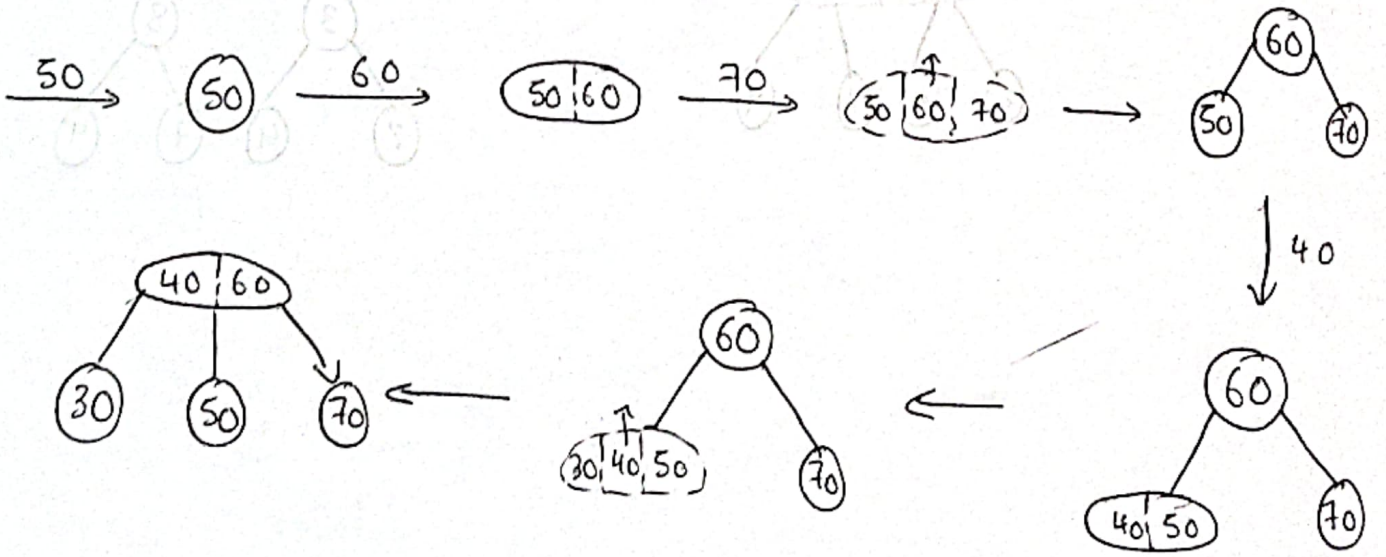
→ Construct a 2-3 Tree:-

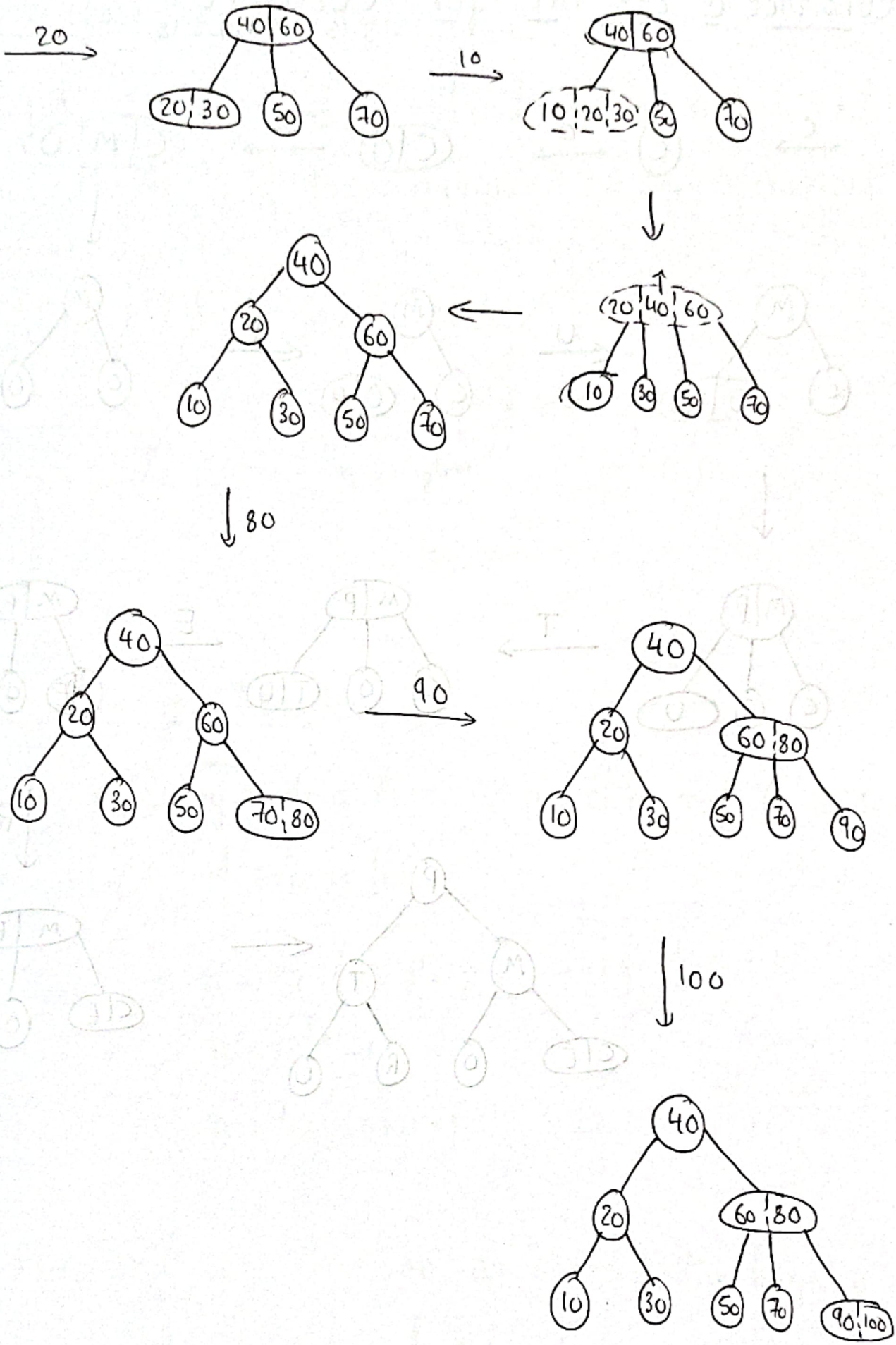
(ii) 55, 30, 75, 65, 45, 5, 15





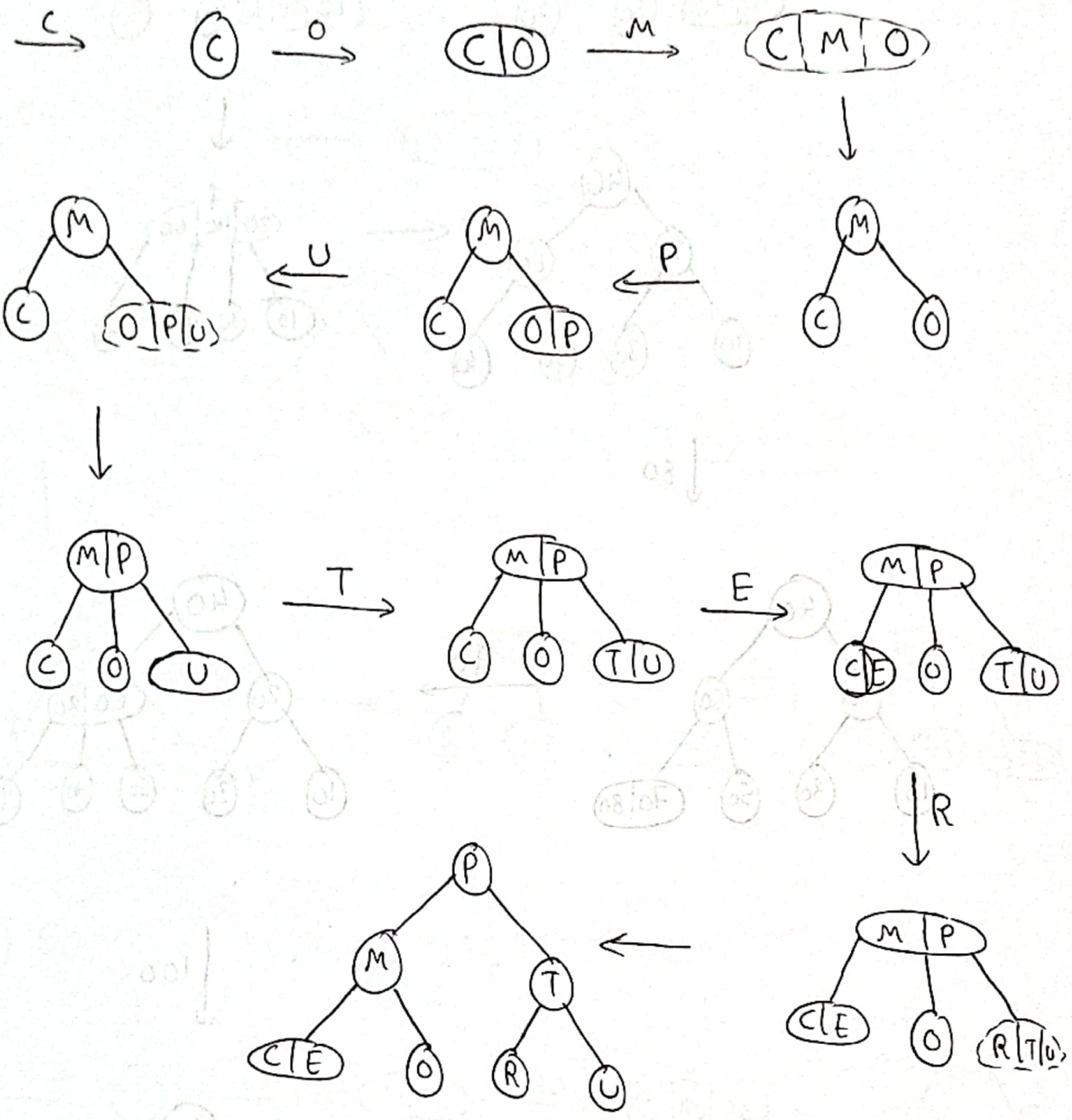
(ii) 50, 60, 70, 40, 30, 20, 10, 80, 90, 100




20
40 60
20 30
50
70
10
40 60
10 20 30
50
70
20 40 60
10
30
50
70
40
20
60
10
30
50
70
80
40
20
60
10
30
50
70 80
90
40
20
60 80
10
30
50
70
90
100
40
20
60 80
10
30
50
70
90 100

→ Construct a 2-3 Tree for COMPUTER

3 15 13 16 21 20 5 18



Worst Case in 2-3 Trees:-

When every node in the tree is a two node then the height of tree increases, so the operations require more time

Total number of nodes,

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^h$$

$$n = 2^{h+1} - 1$$

$$h = \log_2(n+1) - 1$$

Best Case in 2-3 Trees:-

When every node in tree is a three node (2-key & 3 child) then height of tree is less.

$$n = 2\,(3^0 + 3^1 + 3^2 + \dots 3^h)$$

$$n = 3^{h+1} - 1$$

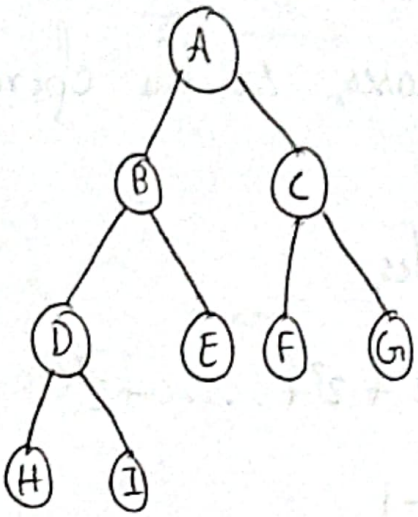
$$n = \log_3(n+1) - 1$$

∴ For any 2-3 Tree on an average, the height is

$$\log_2(n+1) - 1 \leq h \leq \log_3(n+1) - 1$$

∴ Efficiency class $\in \Theta(\log n)$

→ Array Representation of Trees :-



1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I

→ Some properties :-

⊛ First $n/2$ location → Parent nodes

⊛ For any parent, i ⇒ Left Child = $2i$

⊛ For any parent, i ⇒ Right Child = $2i+1$

⊛ For any child i ⇒ Parent $i/2$

⇒ Heap :-

It is a Data Structure i.e a Binary Tree

→ Two properties of Heap :-

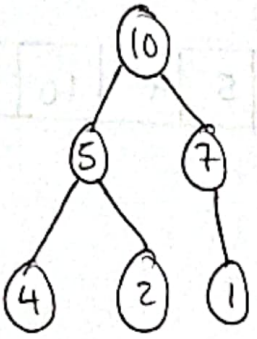
① Trees shape requirement :- It is binary tree i.e complete i.e all level are full except possibly the last level, the rightmost leafs may be missing

(ii) It should follow Parental Dominance :-

The key at each node is greater than (or) equal to the keys at its children.

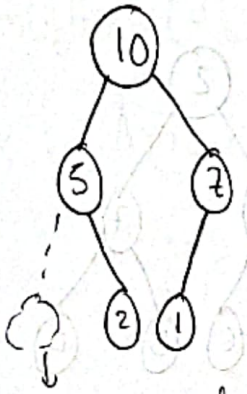
→ Example :-

(i)



The path from root to any leaf node gives elements in sorted order

(ii)



⇒ Not a heap

Missing leaf node

Note :- In a heap,

For any 'i' such that $1 \le i \le \lfloor n/2 \rfloor$,

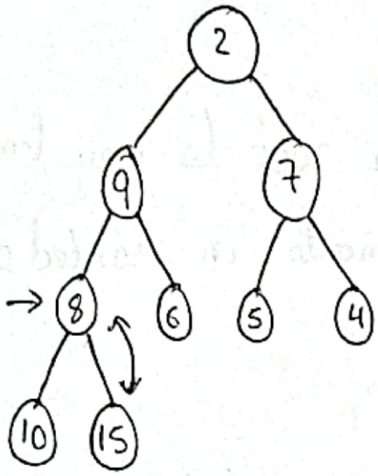
$$H[i] \ge \text{Max}\{H[2i], H[2i+1]\}$$

→ Construct a heap for the values :-

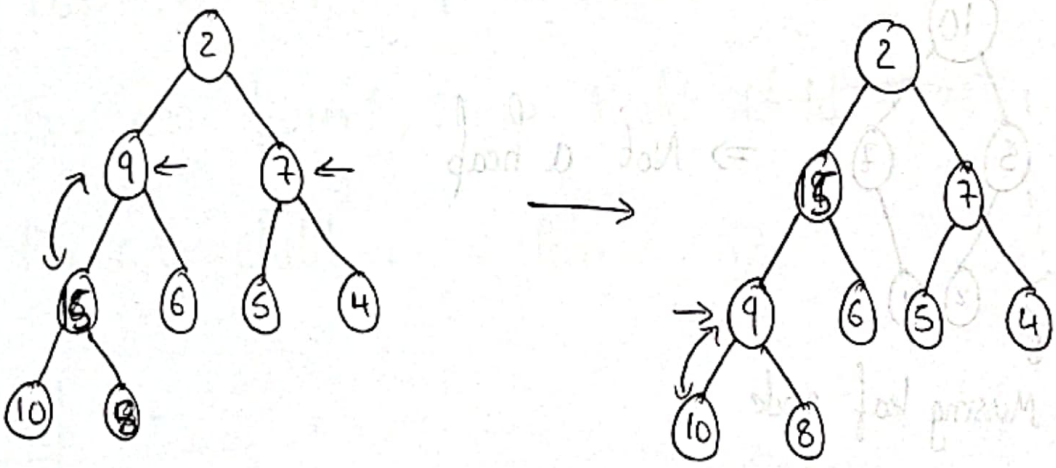
2, 9, 7, 8, 6, 5, 4, 10, 15

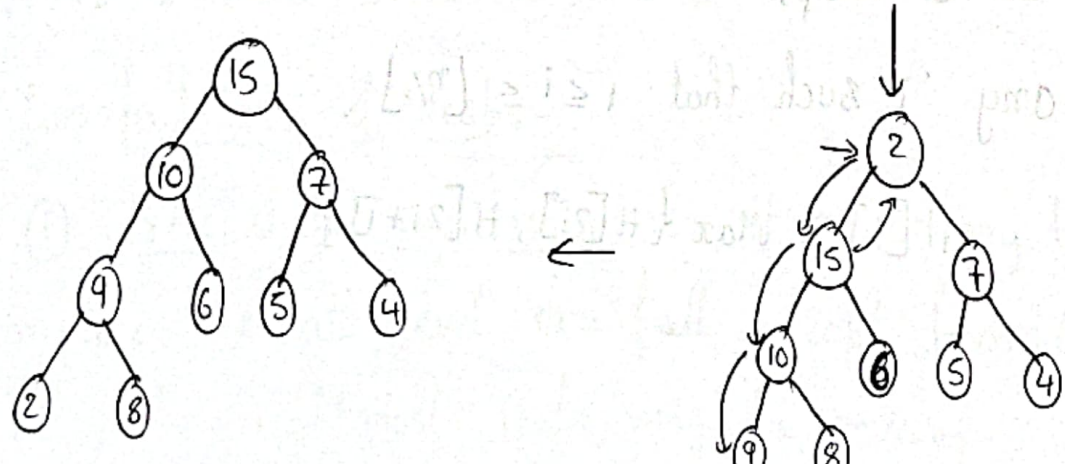
→ Bottom Up Approach :- (Phase - I)

Step 1 :- Construct a Complete BT



2	9	7	8	6	5	4	10	15



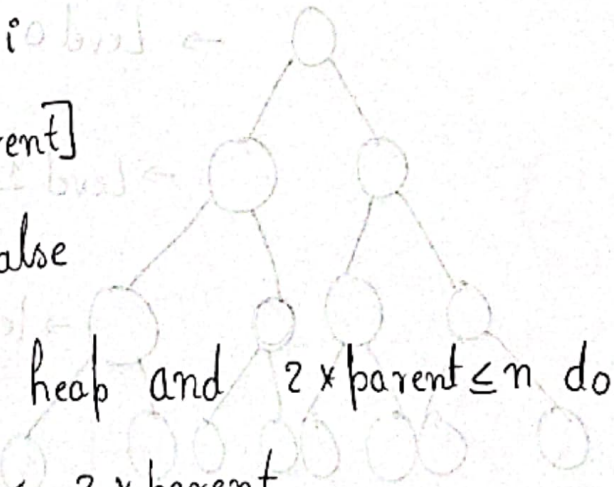


Array :-

2	9	7	8 ↙	6	5	4	10	15
2	9	7 ↙	15	6	5	4	10	8
2	9 ↙	7	15	6	5	4	10	8
2	15	7	9 ↙	6	5	4	10	8
2 ↙	15	7	10	6	5	4	9	8
15	2 ↙	7	10	6	5	4	9	8
15	10	7	2 ↙	6	5	4	9	8
15	10	7	9	6	5	4	9	8

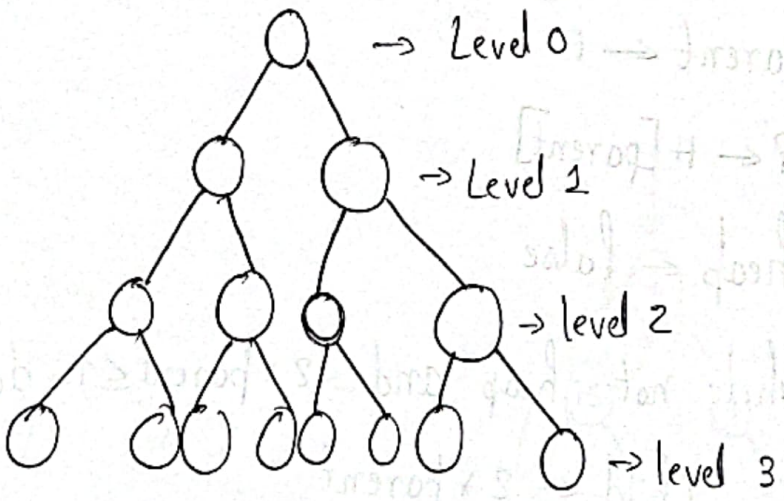
Algorithm :-

```
BottomUpHeap (H[1.... n])
// I/P : Array  H[1... n]
// O/P :  Heap  H [1.... n]

for i ← n/2 down to 1 do
    parent ← i
    P ← H[parent]
    heap ← false
    while not heap and 2*parent ≤ n do
        child ← 2 * parent
```



```
        if child < n
            if H[child+1] > H[child]
                child ← child+1
            end if
        end if

        if P > H[child]
            heap ← true
        Else
            H[parent] ← H[child]
            parent ← child
    end while
    H[parent] ← P
end for
```

⇒ Efficiency Analysis :-



Eg. Full heap Tree.

For a full Tree of level 'K', the number of ~~trees~~ nodes in the tree is

$$n = 2^{K+1} - 1$$

The height of tree is,

$$h = \lfloor \log_2 n \rfloor$$

For exc:-

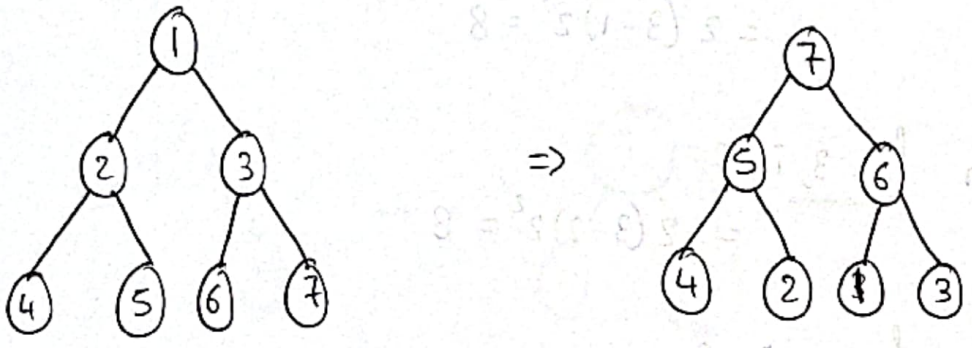
$$n = 15$$

$$h = \lfloor \log_2 15 \rfloor = \lfloor 3.9 \rfloor = 3$$

(ii) else

$$h = \lceil \log_2 (n+1) \rceil - 1$$

$$h = \lceil \log_2 (16) - 1 \rceil = 3$$



Min Heap

Max heap.

* Worst Case in when we have to convert a min heap into max heap which takes maximum comparisions each time.

For moving any key in level 'i' requires

$$2(h-i) \text{ comparisons}$$

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ Keys}} 2(h-i)$$

$$= \sum_{i=0}^{h-1} 2(h-i)2^i$$

$$\vdots$$

$$= 2(n - \log_2(n+1))$$

For $h = 3$, $i = 0$

$$= 2(3-0)2^0 = 6$$

For $h = 3$, $i = 1$

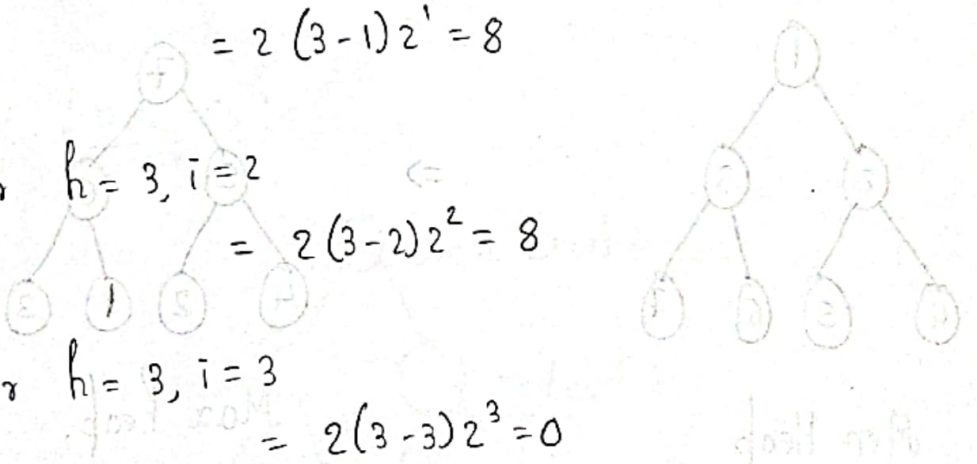
$$= 2(3-1)2^1 = 8$$

For $h = 3$, $i = 2$

$$= 2(3-2)2^2 = 8$$

For $h = 3$, $i = 3$

$$= 2(3-3)2^3 = 0$$

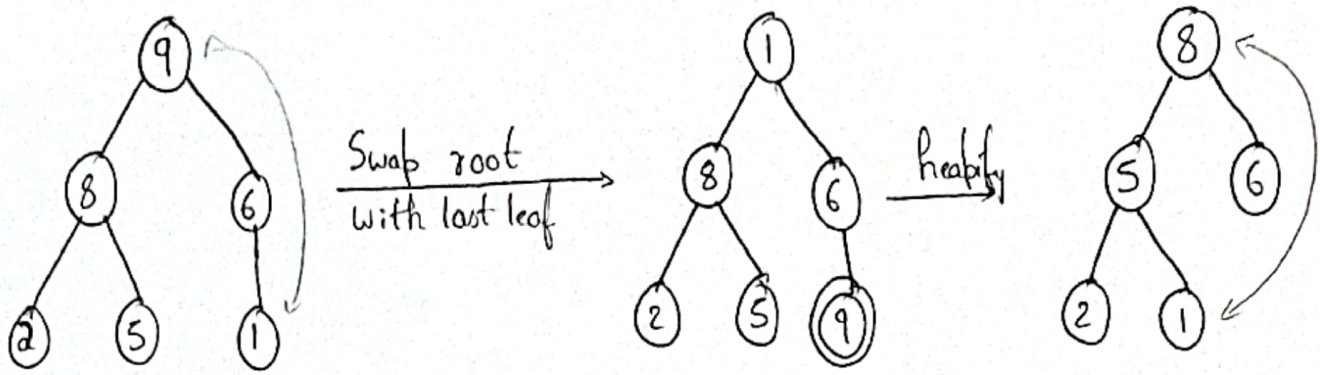


$\therefore$ Total number of comparision $= 6+8+8+0 = 22$

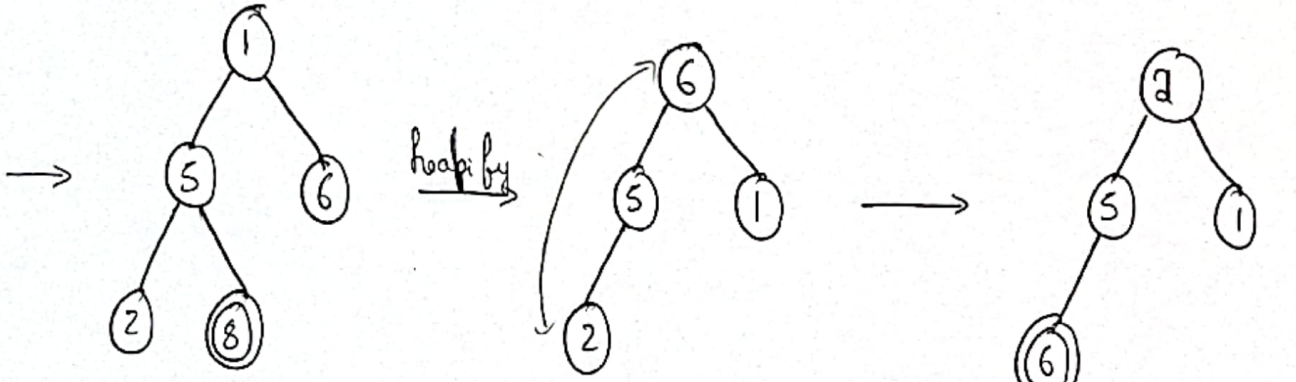
From Formula,

Total no of comparision $= 2(15 - \log_2(16)) = 2(11) = 22$

⇒ Heap Deletion :- (Phase - II)

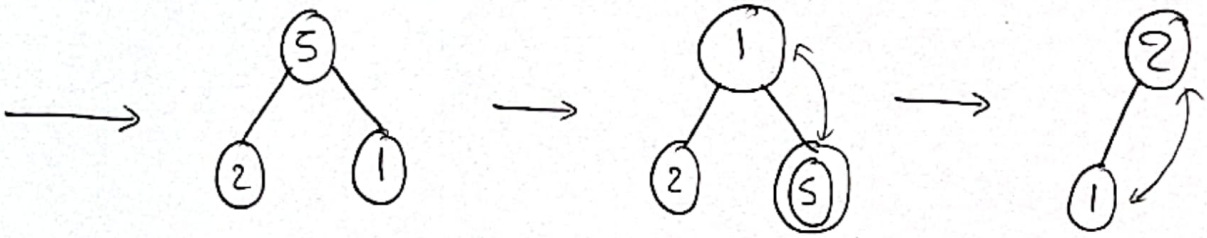


Size of heap
reduced by '1'
(Deleted '9')

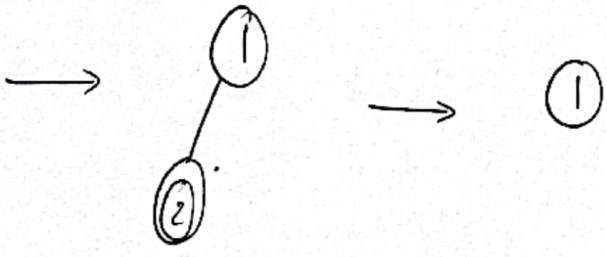


(Deleted '8')

(Deleted '6')



(Deleted '5')

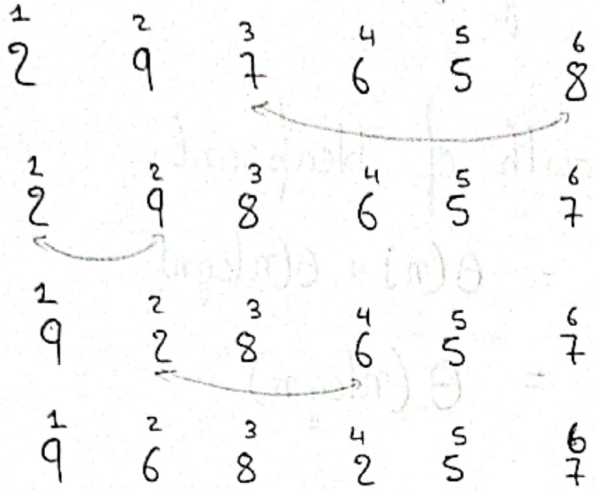


(Deleted '2')

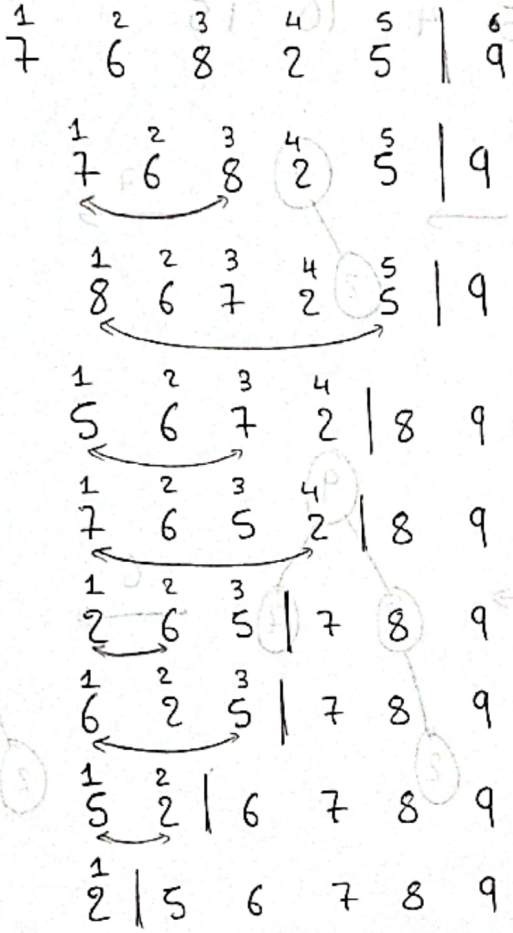
⇒ Sort the given numbers using Heap Sort:-

① 2 9 7 6 5 8

→ Construction:-



→ Deletion



Sorted Array :- 2 5 6 7 8 9

Analysis:-

Order of growth of Construction :- $\Theta(n)$

Order of growth of Deletion :- $\Theta(n \log n)$

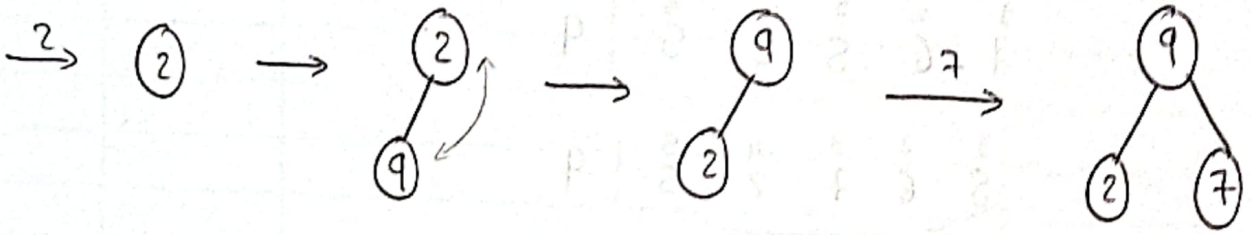
∴ Order of growth of Heapsort

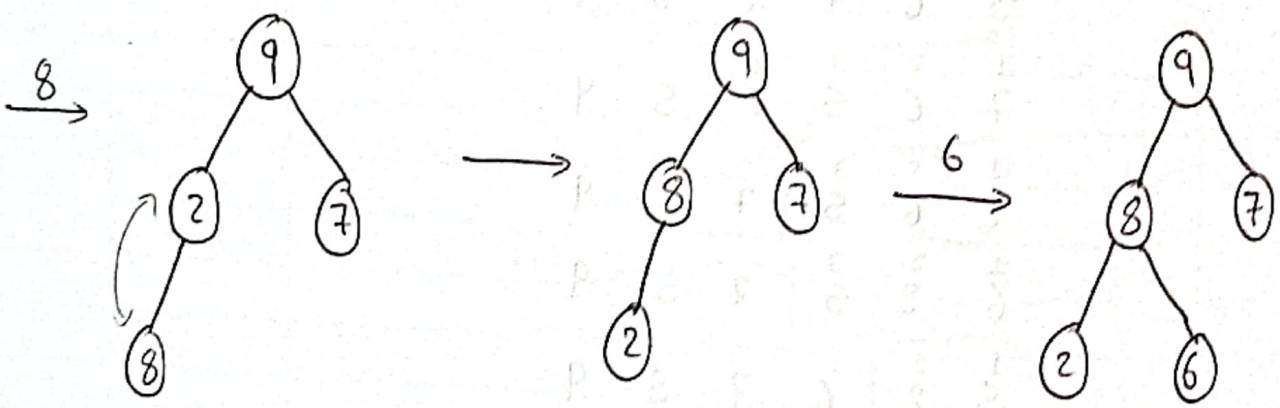
$= \Theta(n) + \Theta(n \log n)$

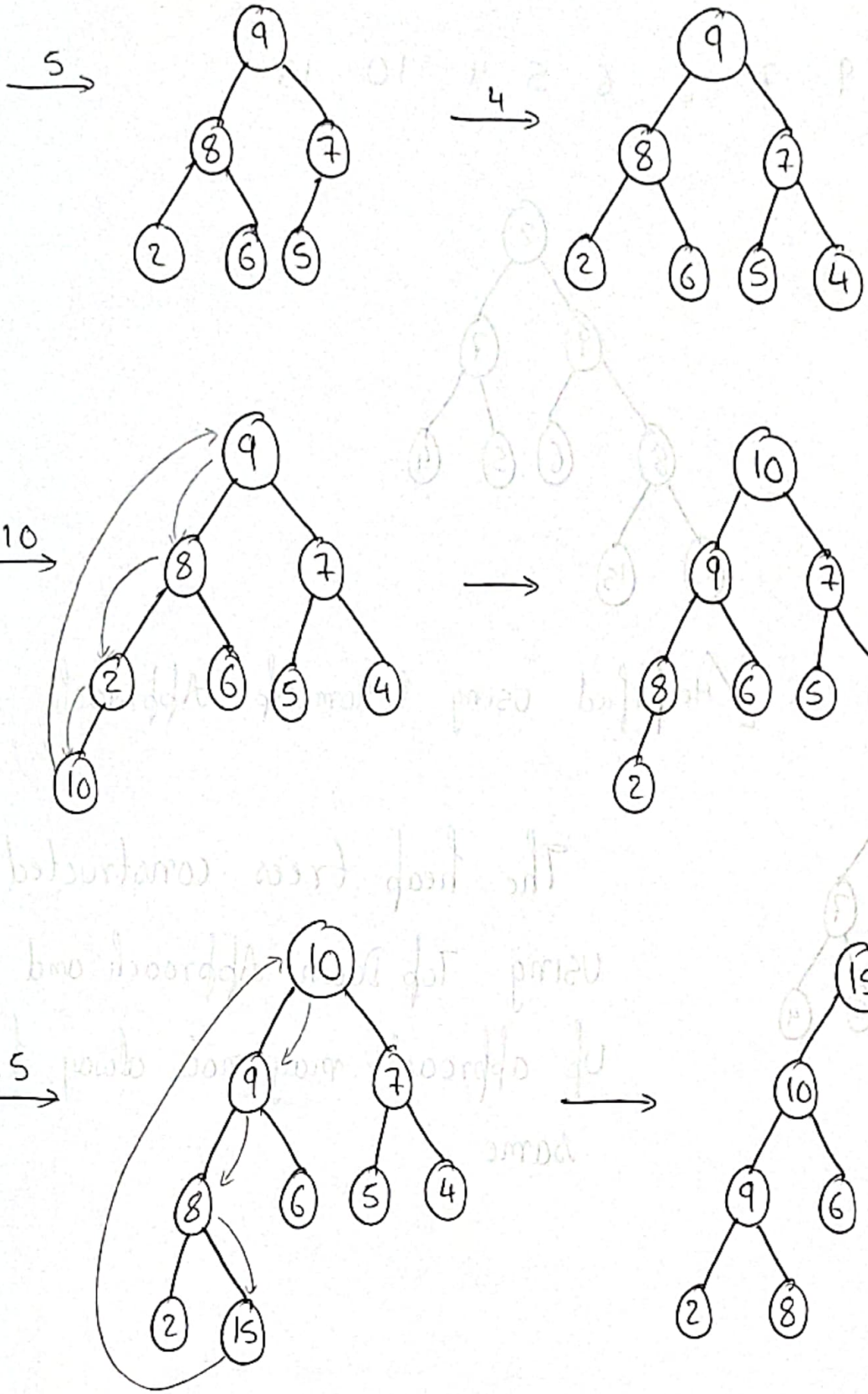
$= \Theta(n \log n)$

⇒ Top - Down Approach of Heap Construction :-

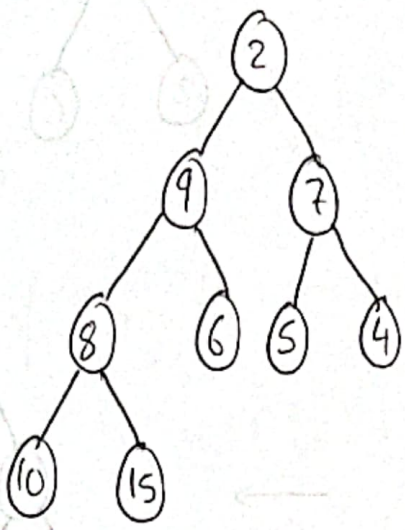
2 9 7 8 6 5 4 10 15



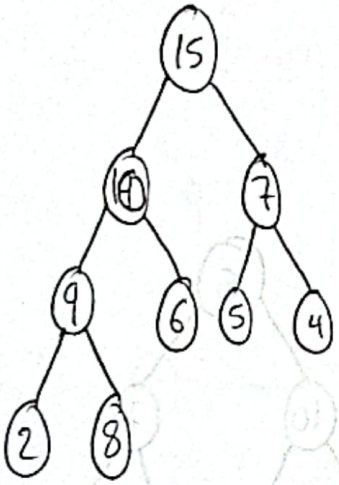



5
9
8
7
2
6
5
4
9
8
7
2
6
5
4
10
9
8
7
2
6
5
4
10
10
9
7
8
6
5
4
2
15
10
9
7
8
6
5
4
2
15
15
10
7
9
6
5
4
2
8

→ 2 9 7 8 6 5 4 10 15



Heapified using Bottom Up Approach



The heap trees constructed using Top Down Approach and Bottom Up approach may not alway be the same