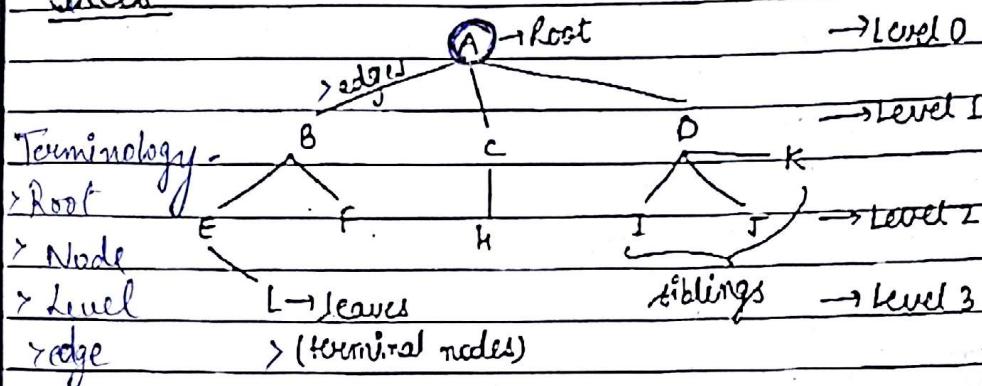


Trees

> Degree of node - no. of successors of the node

> degree of tree - degree of the node having max. degree (Here = 3)

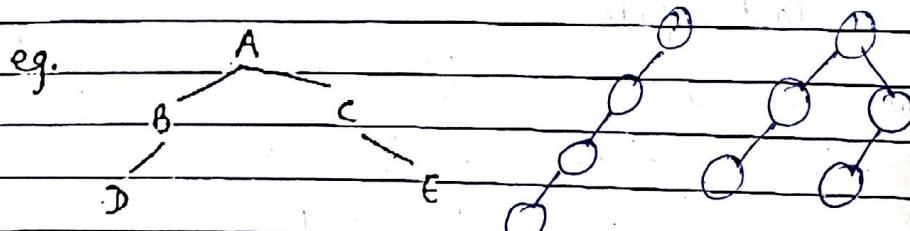
> Path - Connecting edges b/w source and destination

# No backtracking

e.g. path (A → L): A → B → E → I

> Depth of tree - (Max. level of tree + 1) Here, d = 4

→ Binary Trees - Each parent node can have at most two child nodes



① 2 binary trees are said to be similar when they have the identical structures only

→ Complete Binary Tree - Every node should have two children except the terminal / last nodes

Depth of a CBT =

$$D_n = \lceil (\log_2 n) + 1 \rceil$$

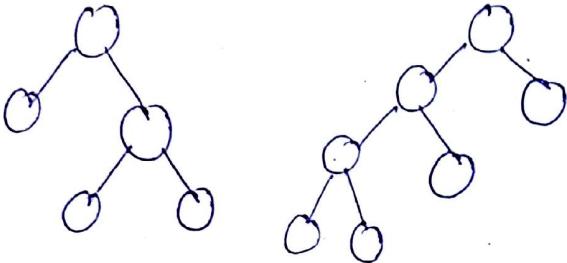
↑ greatest integer fn

① copy = same structure + same nodes

## strictly binary tree

- BT with every node  $N$  having either 0 or 2 children.
- also called as extended 2-Tree or 2-Tree.
- nodes with 2 children = internal node  
" " 0 " = external "

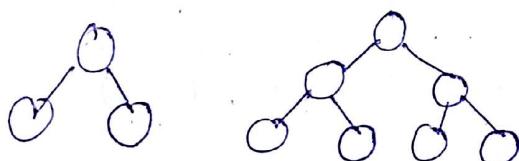
e.g.



## Complete Binary tree

- spcl type of strictly binary tree where all leaves are at the same level.

e.g.



- each node should have 2 children except the last node.

$$\text{Total nodes} = 2^0 + 2^1 + 2^2 + \dots + 2^d$$

$$= 1 + 2 \frac{(2^d - 1)}{2 - 1}$$

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

$$T = 2^{d+1} - 1$$

$$\text{or } 2^{d+1} = T + 1$$

Taking log on both sides,

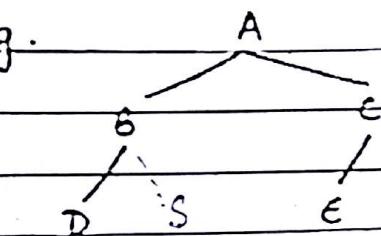
$$d+1 = \log_2(T+1)$$

$$\text{or } d = \log_2(T+1) - 1$$

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

## → Representation of Tree -

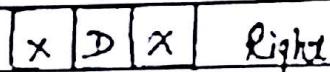
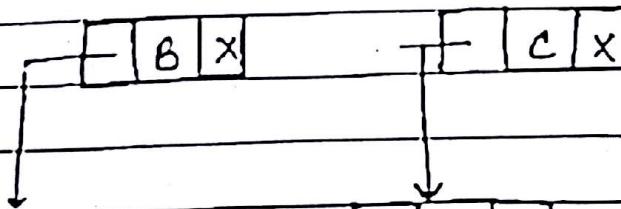
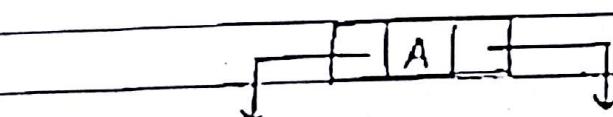
e.g.



$$\begin{aligned} \text{Left child} &= 2I \\ \text{Right } II &= 2I + 1 \end{aligned}$$

Array:	A	1	Root = 1 (A)
	B	2	
	C	3	
	D	4	(2x2)
	S	5	(2x2) + 1
	E	6	(2x3)

Linked list :-



Stack -

	INFO	LEFT	RIGHT
1)	D	0	0
2)	A	4	6
3)			
4)	B	1	0
5)			
6)	C	7	0
7)	E	0	0

Struct node

```

S struct node * left;
char data;
struct node * right;
}
  
```

→ Traversals -

True Traversals

Levelorder



starts from Preorder

→ ROOT

Root

Left

Right

Inorder → same as Post

Left

Root

Right

Postorder → starts from extreme left

left

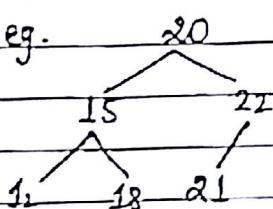
Right

Root

leaf

→ Pre-order Traversal using stack -

eg.



1) Stack  $\emptyset$

2) Process 20

Stack:  $\emptyset, 20$  P: 20

3) Process 15

Stack:  $\emptyset, 20, 15$  P: 20, 15

4) Process 12

P: 20, 15, 12

5) Pop top of stack of 18.

Stack:  $\emptyset, 22$  P: 20, 15, 12, 18

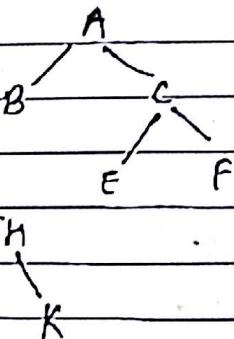
6) Pop and process 22

Stack:  $\emptyset$  P: 20, 15, 12, 18, 22

7) Process 21

P: 20, 15, 12, 18, 22, 21

Ans)



1) Stack  $\emptyset$

2) Process A

P: A

S:  $\emptyset, C$

3) Process B

P: A, B

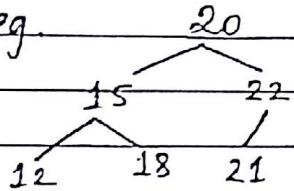
S:  $\emptyset, C$

4) Process D

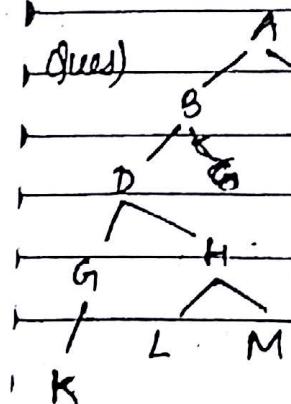
P: A, B, D

S:  $\emptyset, C, G$

→ In-order Traversal using Stack:



- 1) Push on stack:  $\emptyset, 20, 15, 12$
  - 2) Process 12
  - 3) Process 15
  - 4) Pop & process 18  
S:  $\emptyset, 20$       P:  $12, 15$
  - 5) Pop & process 20  
S:  $\emptyset$       P:  $12, 15, 18$
  - 6) Push 22 & 21  
S:  $\emptyset, 22, 21$
  - 7) Process 21
  - 8) Process 22  
P:  $12, 15, 18, 20, 21, 22$



- 1) Push onto stack:  $\emptyset, A, B, D, G, K$
  - 2) Process K
  - 3) Process G
  - 4) Process D  $P: K, G, D$
  - 5) Push on stack  $\emptyset, H, L$

$S: \emptyset, A, B, H, L$

6) Process L

7) Process H

P: K, G, D, L, H

8) Push on stack M

S:  $\emptyset, A, B, M$

9) Pop and process M

P: K, G, D, L, H, M

S:  $\emptyset, A, B,$

10) Process B

11) Process A

P: K, G, D, L, H, M, B, A

12) Push C & E

S:  $\emptyset, C, E$

13) Pop & process E

14) Process C

P: K, G, D, L, H, M, B, A

→ Preorder (INFO, LEFT, RIGHT, ROOT)

1) Set TOP = 1, STACK[1] = NULL and PTR := ROOT

2) Repeat steps 3 to 5 while PTR ≠ NULL

3) Apply Process to INFO[PTR]

4) If RIGHT[PTR] ≠ NULL

    Set TOP := TOP + 1 and STACK[TOP] := RIGHT[PTR]

5) If LEFT[PTR] ≠ NULL

    Set PTR := LEFT[PTR]

else

    Set PTR := STACK[TOP] and TOP := TOP - 1

3) Inorder (INFO, LEFT, RIGHT, ROOT) -

Set TOP := 1, STACK[1] = NULL and PTR := ROOT

Repeat steps 1 to 3 while PTR ≠ NULL

- (a) Set  $\text{TOP} := \text{TOP} + 1$  and  $\text{STACK}[\text{TOP}] = \text{PTR}$
- (b) Set  $\text{PTR} := \text{LEFT}[\text{PTR}]$
- 3) Set  $\text{PTR} := \text{STACK}[\text{TOP}]$  and  $\text{TOP} := \text{TOP} + 1$
- 4) Repeat steps 5 to 7 while  $\text{PTR} \neq \text{NULL}$  (Backtracking)
- 5) Apply process to  $\text{INFO}[\text{PTR}]$
- 6) If  $\text{RIGHT}[\text{PTR}] \neq \text{NULL}$ 
  - (a) Set  $\text{PTR} := \text{RIGHT}[\text{PTR}]$
  - (b) Go to step 3
- 7) Set  $\text{PTR} := \text{STACK}[\text{TOP}]$  and  $\text{TOP} := \text{TOP} + 1$
- 8) Exit

→ Postorder (INFO, LEFT, RIGHT, ROOT) -

- 1) Set  $\text{TOP} := 1$ ,  $\text{STACK}[1] = \text{NULL}$  and  $\text{PTR} := \text{ROOT}$
- 2) Repeat steps 3 to 5 while  $\text{PTR} \neq \text{NULL}$
- 3) Set  $\text{TOP} := \text{TOP} + 1$  and  $\text{STACK}[\text{TOP}] := \text{PTR}$
- 4) If  $\text{RIGHT}[\text{PTR}] \neq \text{NULL}$ 
  - Set  $\text{TOP} := \text{TOP} + 1$  and  $\text{STACK}[\text{TOP}] := -\text{RIGHT}[\text{PTR}]$
- 5) Set  $\text{PTR} := \text{LEFT}[\text{PTR}]$
- 6) Set  $\text{PTR} := \text{STACK}[\text{TOP}]$  and  $\text{TOP} := \text{TOP} - 1$
- 7) Repeat while  $\text{PTR} > 0$ 
  - (a) Apply process to  $\text{INFO}[\text{PTR}]$
  - (b) Set  $\text{PTR} := \text{STACK}[\text{TOP}]$  and  $\text{TOP} := \text{TOP} - 1$
- 8) If  $\text{PTR} < 0$ 
  - (a) Set  $\text{PTR} := -\text{PTR}$
  - (b) Go to step 2
- 9) Exit

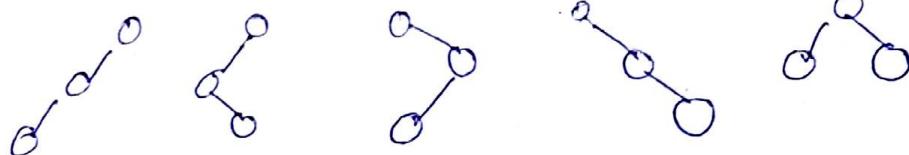
## Counting binary trees

if  $n = 1$ , one binary tree  
 $n = 2$



2 trees

for  $n = 3$ ,



In general, No. of BT =  $(2^n - n)$

→ Construction of Binary Tree -  
Any two order combinations / traversals are given using which we have to form / construct our binary tree

e.g. Pre - A B D E C F

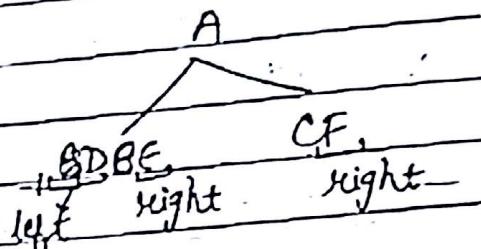
N L R

In - D B E A C F

L N R

First check in pre-order and then inorder for each element  
→ root node

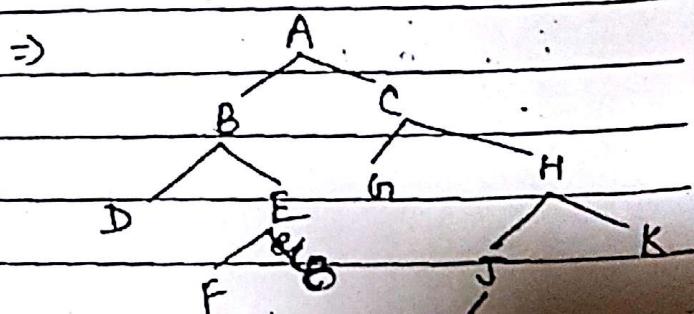
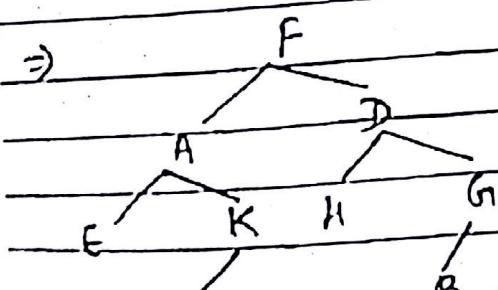
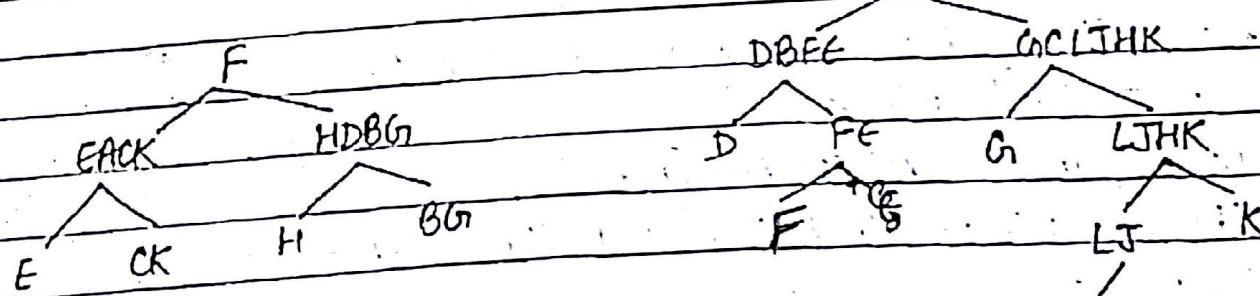
D B E, A C F  
left sub-tree      form right subtree



e.g. In: E A C K F H D B G  
Pre: F A E K C D H G B

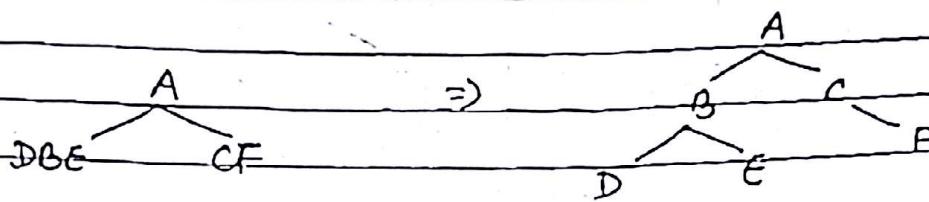
Post: D F E B G L J K H C A

In: D B F E A G C L J H K



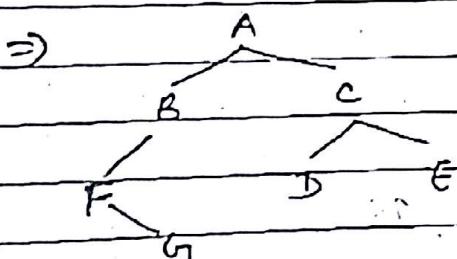
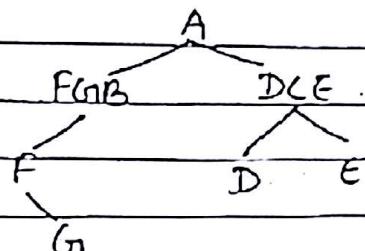
eg. Post: DEBFCA

In: DBEACF



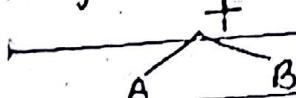
Post: GFBDCEA

In: FGIBADCE

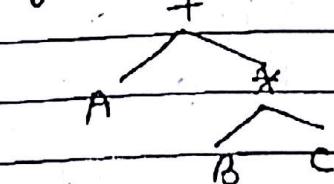


### → Expression Trees -

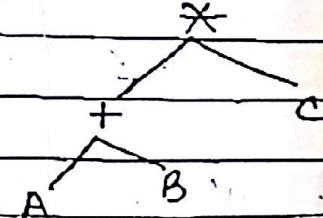
eg.  $A + B$



eg.  $-A + B * c$



eg.  $(A + B) * c$



Leaves : tree operands

other nodes : operators

eg.  $ab + cde + **$   
 construct expression tree from postfix.

Symbol

a

b

+

c

d

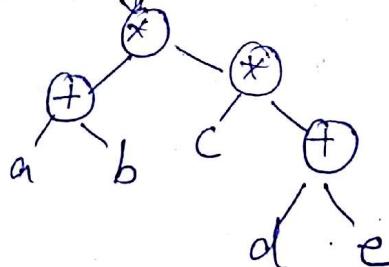
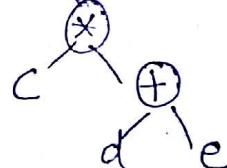
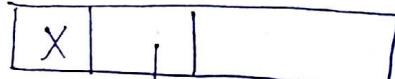
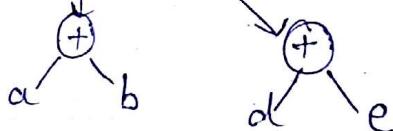
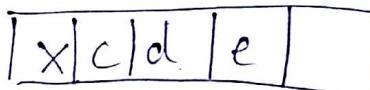
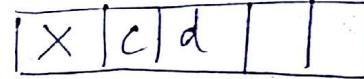
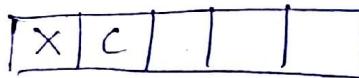
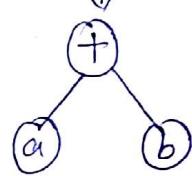
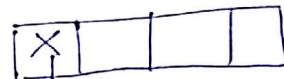
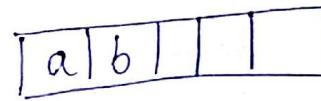
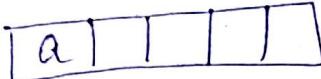
e

+

\*

\*

Stack



## Binary search trees

operations:

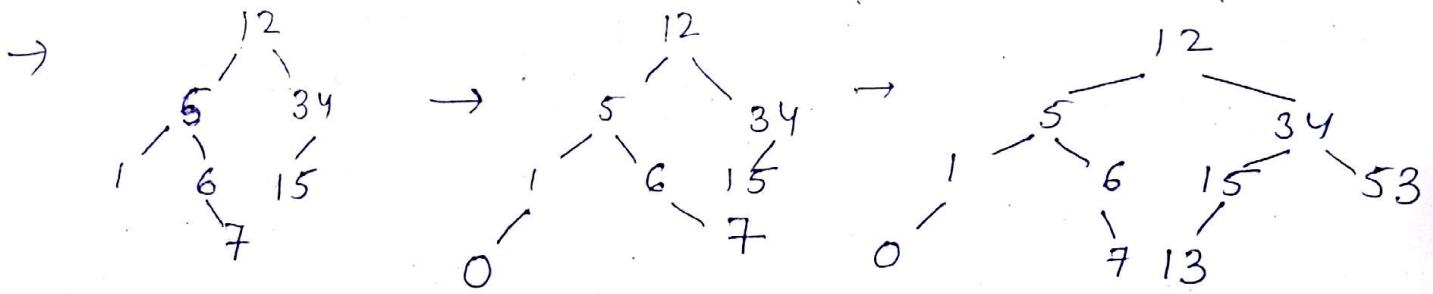
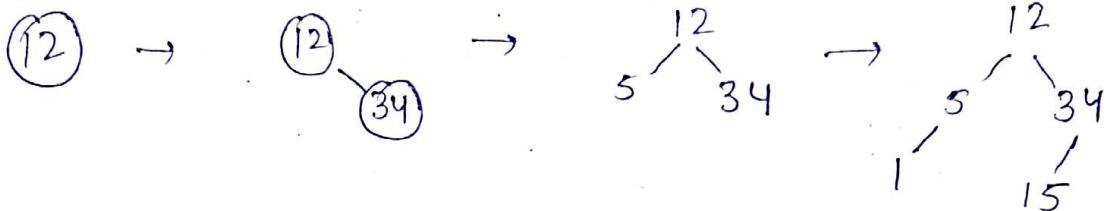
Inserion  
deletion  
searching

getting data  
in sorted order

(inorder traversal of BST)

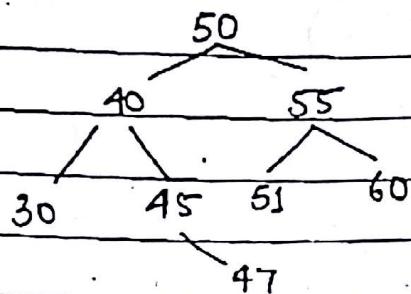
### ① Insertion

e.g. 12, 34, 5, 1, 15, 6, 7, 0, 13, 4, 53



ans.

→ Binary Search Tree (BST) -

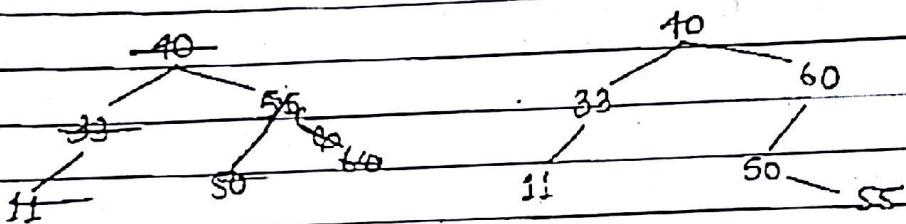


Left child always has smaller value than root value while right child always has higher value than root.

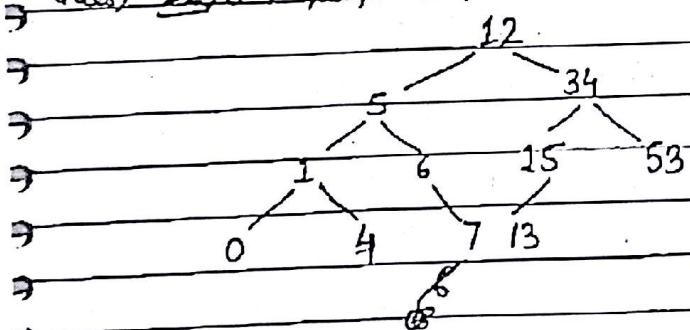
→ root (always the first element)

→ Ques) 40, 60, 50, 33, 55, 11 - form a binary search tree

→ 11, 33, 40, 50, 55, 60

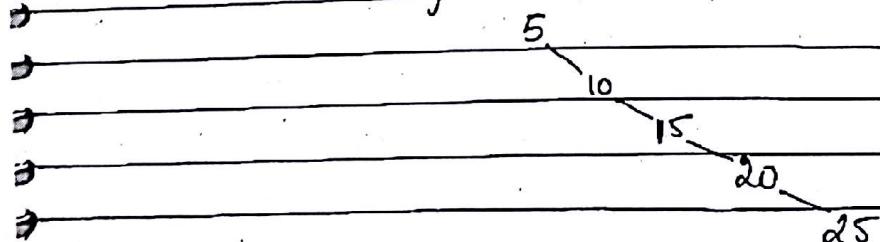


→ Ques) root 12, 34, 5, 1, 15, 6, 7, 0; 13, 4, 53



→ Right-skewed tree - All the elements of BST are on the right side.

eg. 5, 10, 15, 20, 25 → ASCENDING ORDER



→ Left-skewed tree - All the elements of BST are on the left side (Descending Order)

## BST insertion

```
1. if (tree = NULL)
    tree → left = tree → right = NULL
    tree → num = digit; // item
else
    if (digit < tree → num)
        tree → left = insert (tree → left, digit);
    else if (digit > tree → num)
        tree → right = insert (tree → right, digit);
    else
        if (digit == tree → num)
            printf (" Duplicate node ");
            exit (0).
```

2. Return

## BST searching

```
1. void search ( struct rec *tree, digit )
2. if (tree = NULL)
    printf (" No. does not exist ").
else if (digit == tree → num)
    printf ("%d \n", digit );
else if (digit < tree → num)
    search ( tree → left, digit );
else
    search ( tree → right, digit );
```

3. Return

Date \_\_\_\_\_

Disadvantages - i) BSTs are not balanced. This leads to wastage of memory space while storing the data in any way

### Deletion:-

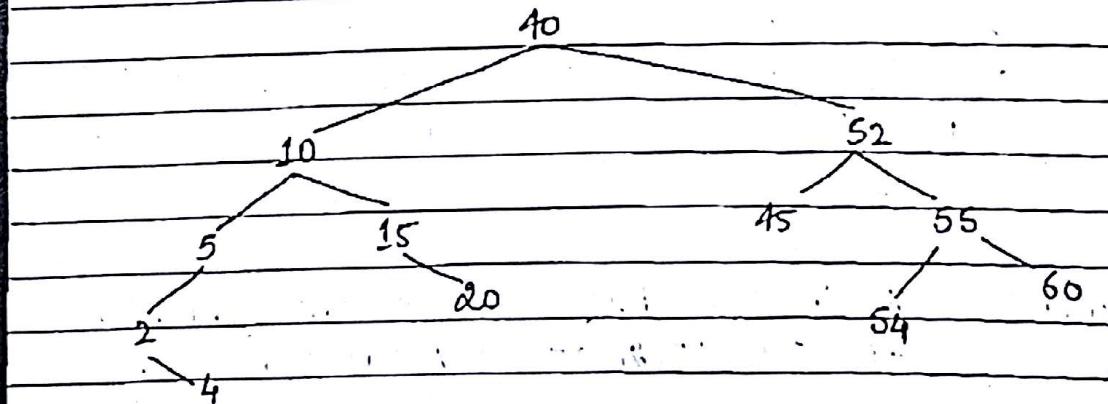
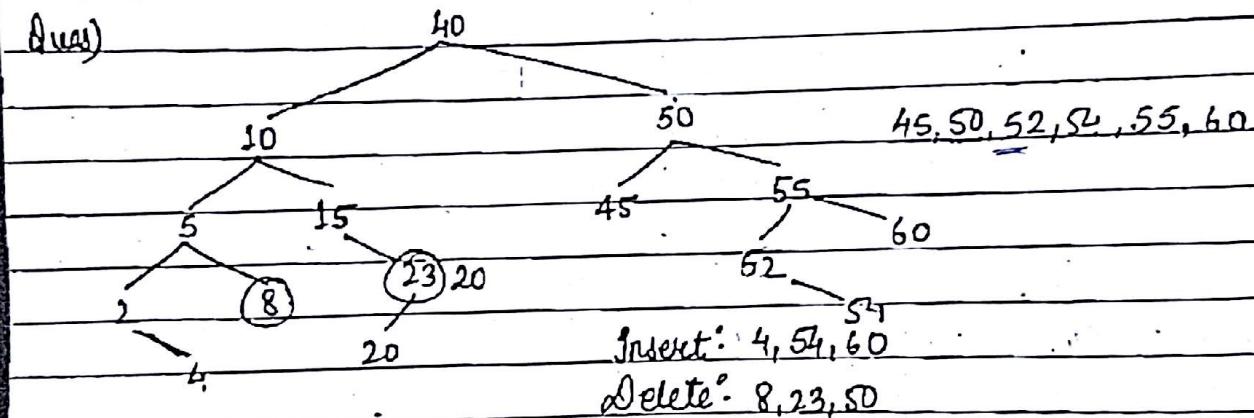
These can be three types of nodes in BSTs -

i) No child - node is simply deleted

ii) Single child - child takes the place of its parent after parent node is deleted.

iii) Two children - Inorder successor takes the place of the parent node

Ques)

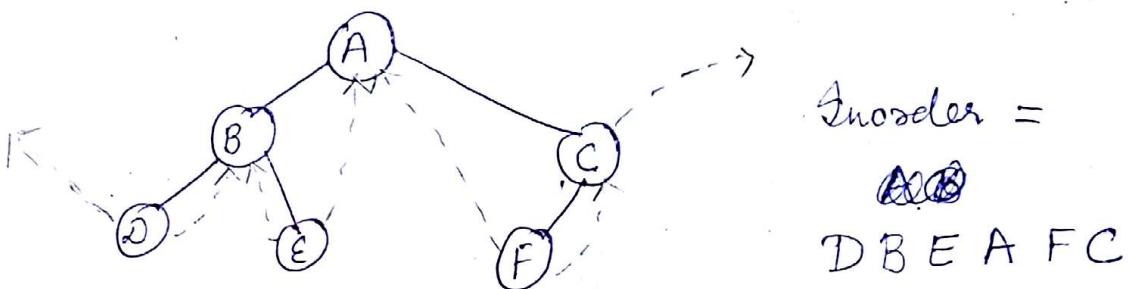


## Threaded binary tree

In a linked representation of a binary tree, the no. of null links are actually more than non-null pointers.

- ④ Idea is to replace all the null pointers by the appropriate pointer values called 'threads'.

$\Rightarrow L_{child} =$  points to previous node in inorder  
 $R_{child} =$       "      "      next      "      "



- These trees make in-order tree traversal a little faster.

→ Heap (tree) :-

Root is either smaller or greater than both its children.

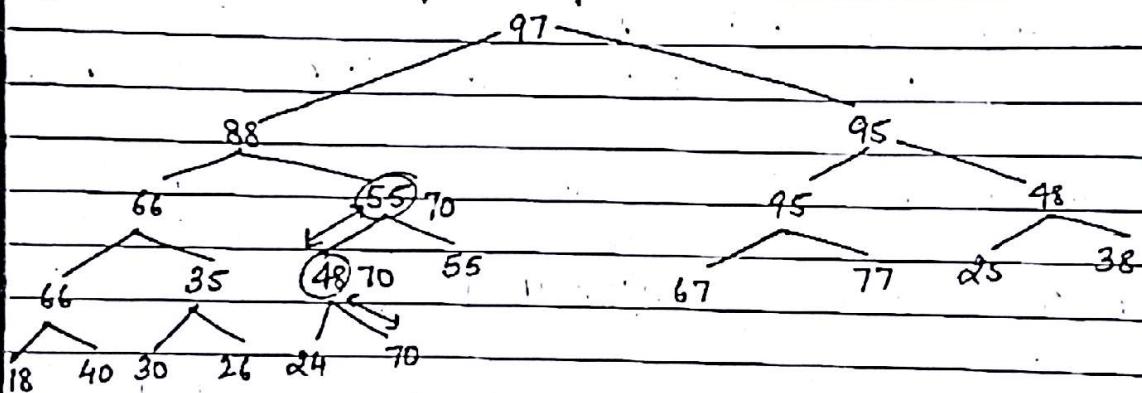
Maxheap - Root  $\geq$  both children

Minheap - Root  $<$  both children

Insertion -

Always try to make a CBT while inserting a new node.

e.g. Insert 70 to the given heap.

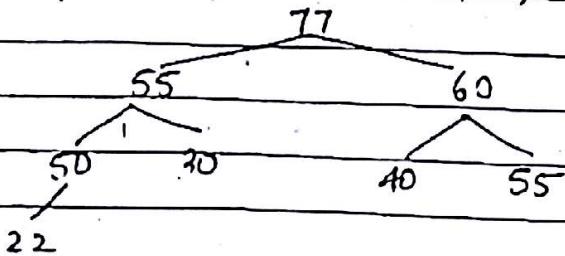


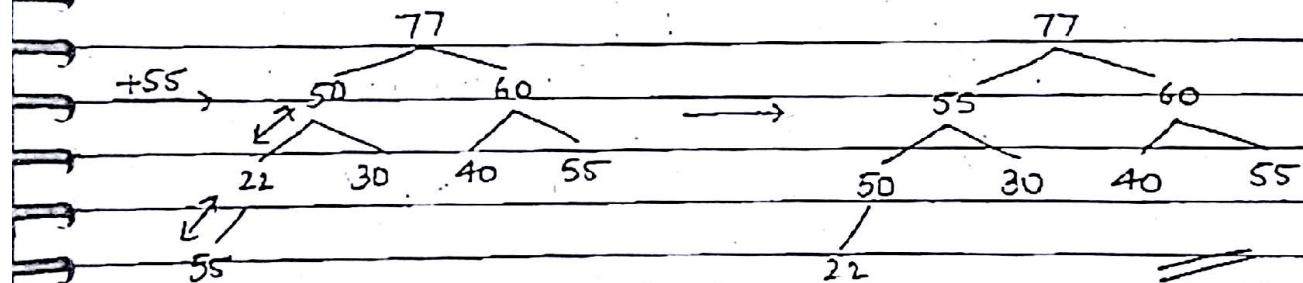
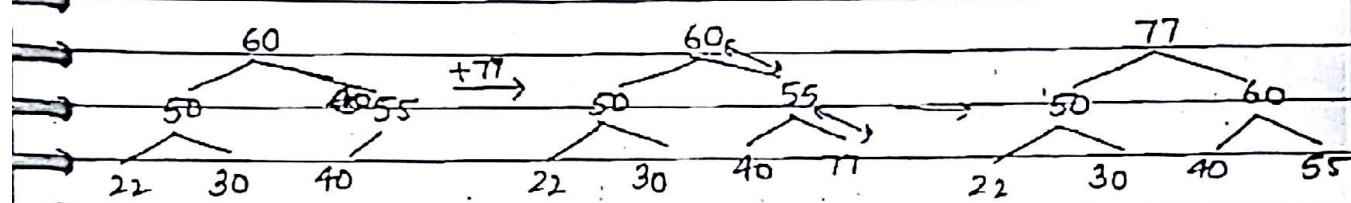
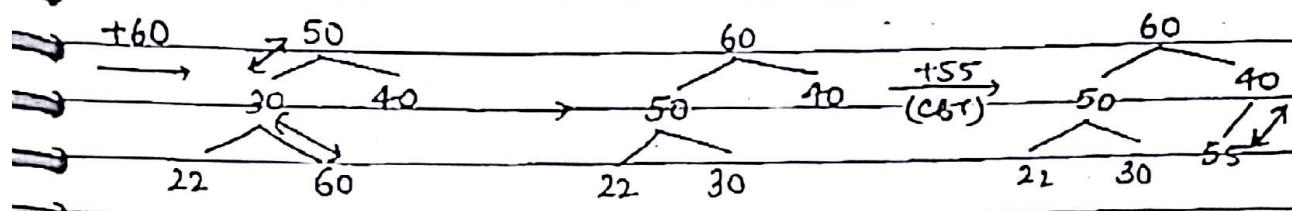
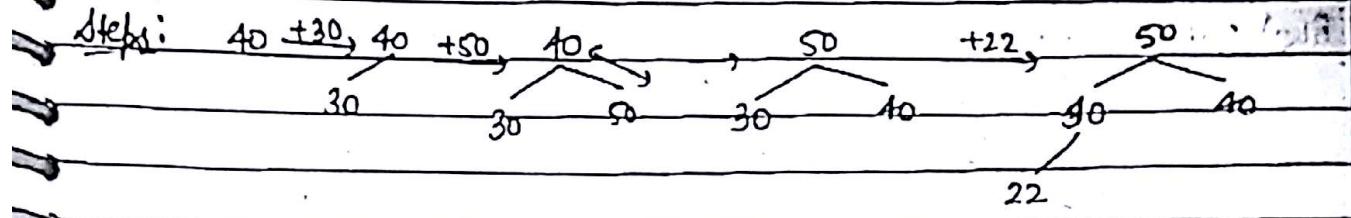
i) In order to complete the binary tree [CBT], we initially add 70 onto the right side of 48.

ii) Then we compare the value of child 70 & parent 48 and thus interchange the nodes.

iii) Second comparison with node 55 result in another interchange.

e.g. Form a heap: 40, 30, 50, 22, 60, 55, 77, 55

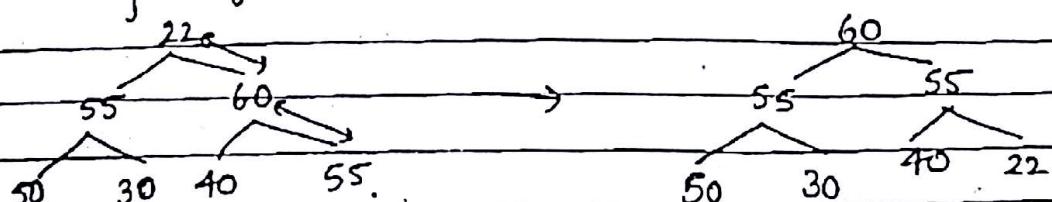




→ Deletion (from the top root)-

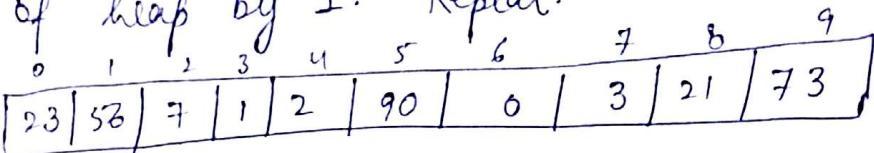
Replace the root with terminal node and then compare and interchange values.

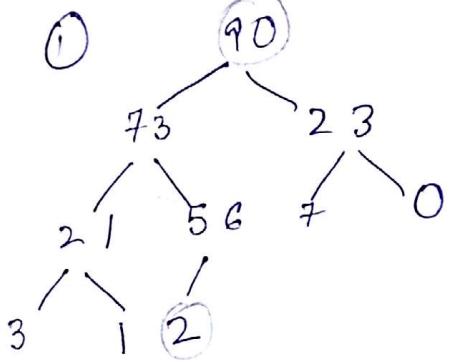
e.g. deleting 77 from above heap



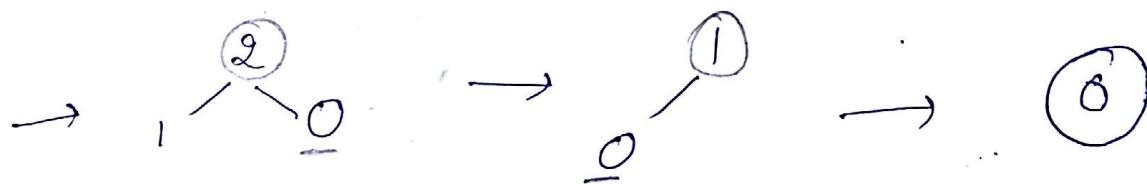
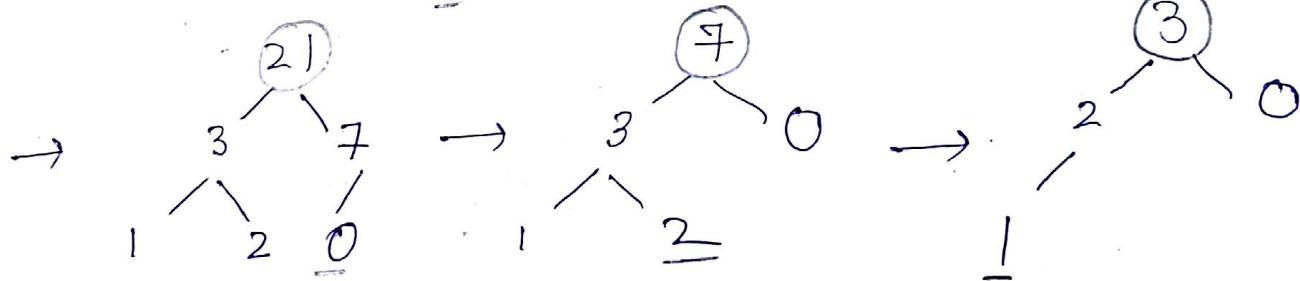
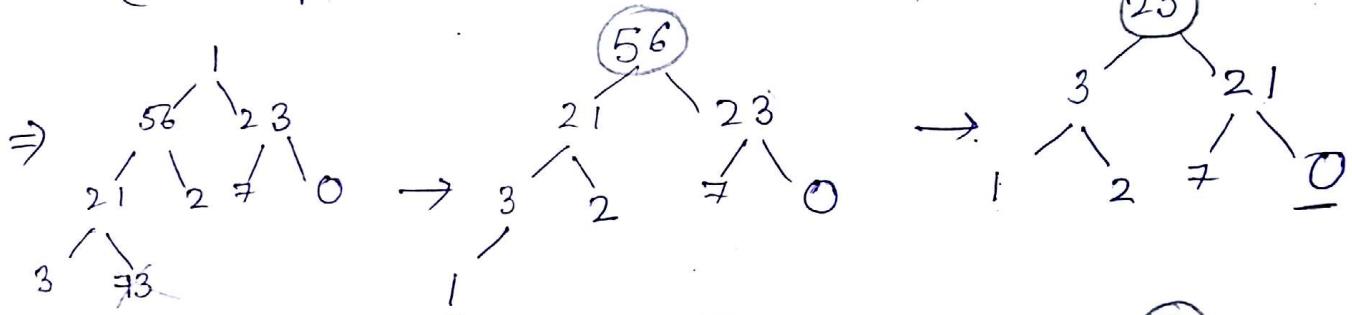
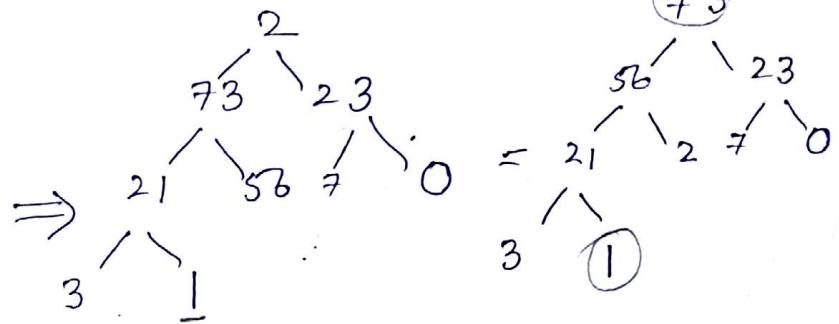
## Heap Sort

- ① create a heap from the given array  
 ② swap root node & last node, decrease the size of heap by 1. Repeat.

e.g. 



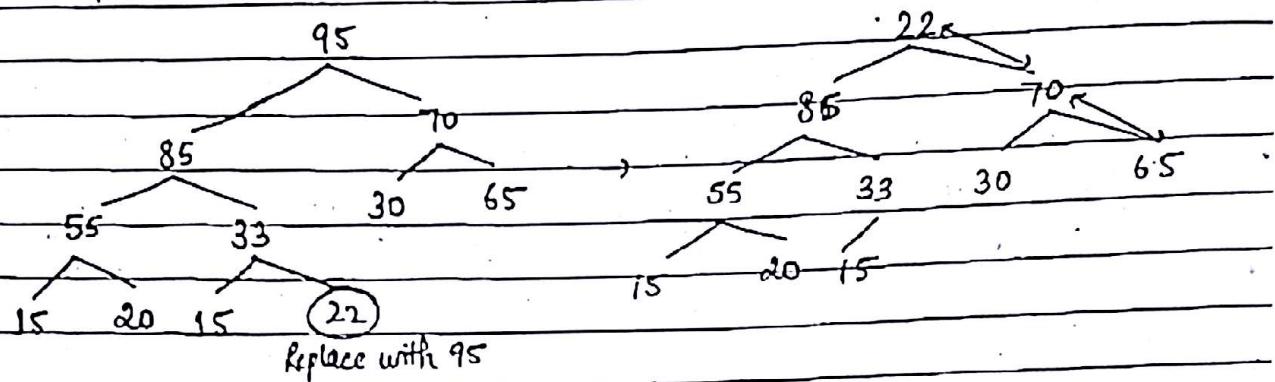
(Max-heap).



O/P = 90, 73, 56, 23, 21, 7, 3, 21, 1, 0.

Date الموعد

Ques) Perform deletion -



$\rightarrow$  AVL Search Tree Adelson, Velskii & Landis  
(Balanced Binary Search Tree)

$$|R(T_b) - R(T_R)| \leq 1 \rightarrow \text{Necessary condition.}$$

where  $h(T_L)$  = height of left subtree       $h(T_R)$  = height of right subtree  
Balancing factor: -1, 0, +1  $\Rightarrow$  balanced

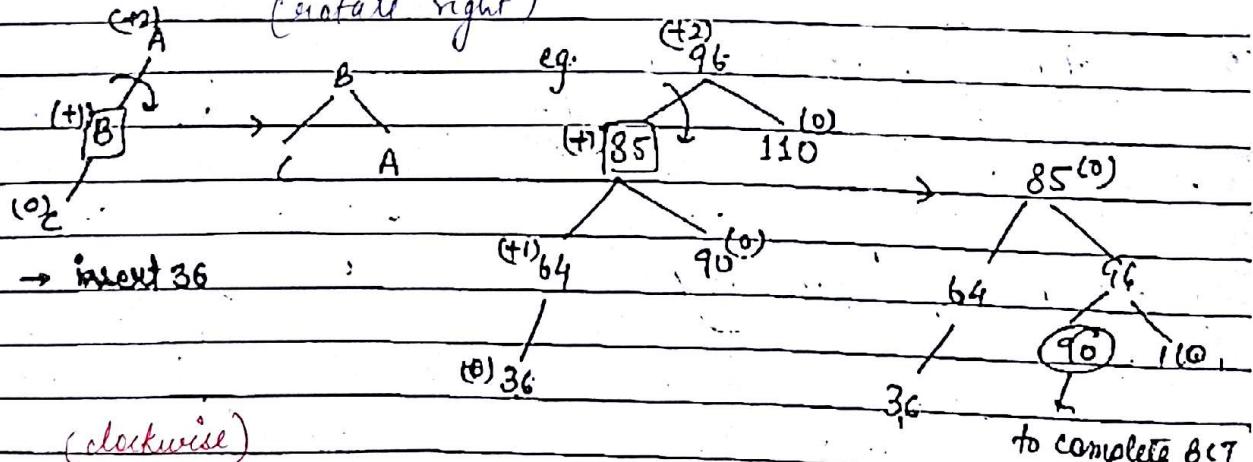
Any other value of Balancing factor means tree is not balanced.

At such points, rotation needs to be formed.

$$BF = R(T_L) - R(T_R)$$

1) ~~RR~~ Rotation - element inserted at left side of left child

~~cosy~~ (erectate right)

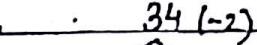


# cw rotation about the node immediately next to the node having the unbalancing factor (+2)

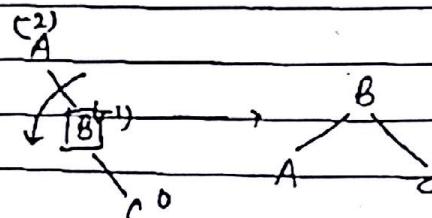
AVL tree = Balanced binary search tree

2) RR Rotation - element inserted on right side of right child

e.g. 34 (+2)

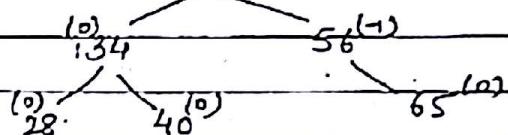


(Insert 65)



# acw rotation

(anti-clockwise)

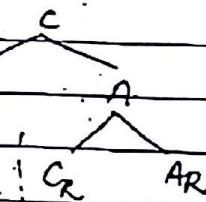
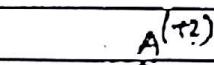


3) LR Rotation - Insertion at right of left

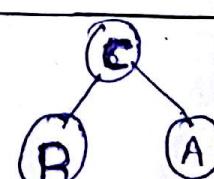
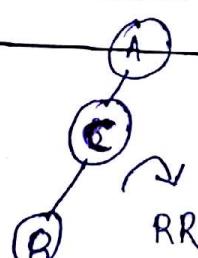
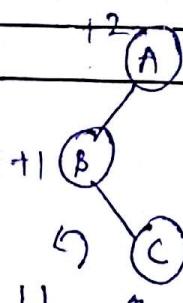
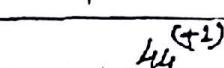
$LR = RR + LL$

# RR followed by LL

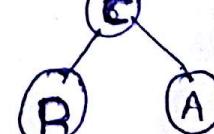
$LL + RR$



$C_L$  goes to left subtree  
 $C_R$  goes to right subtree

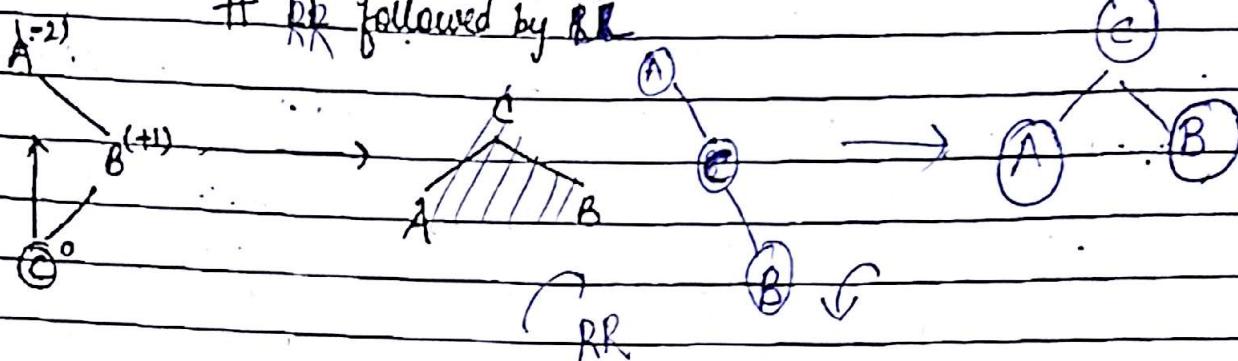


RR

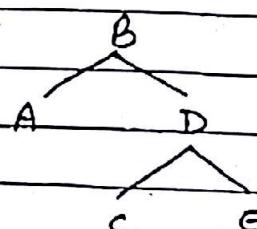
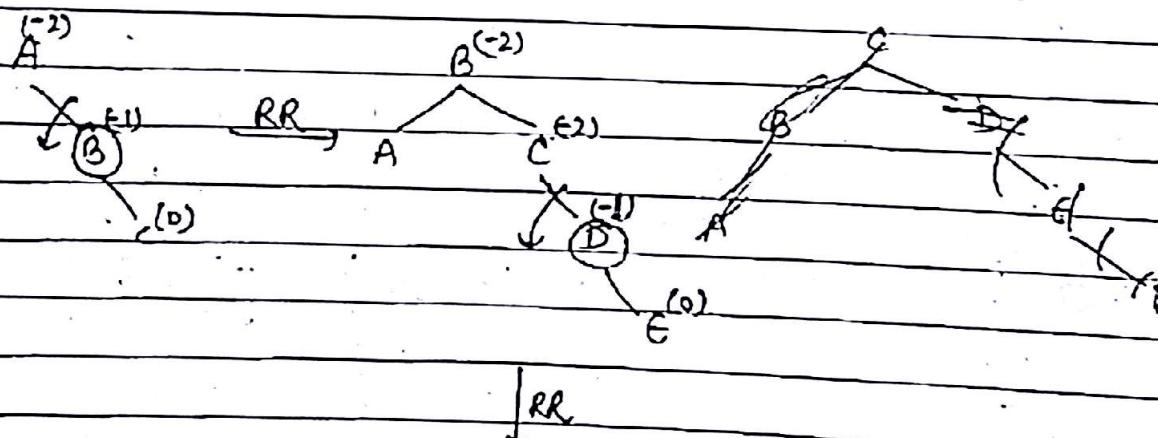
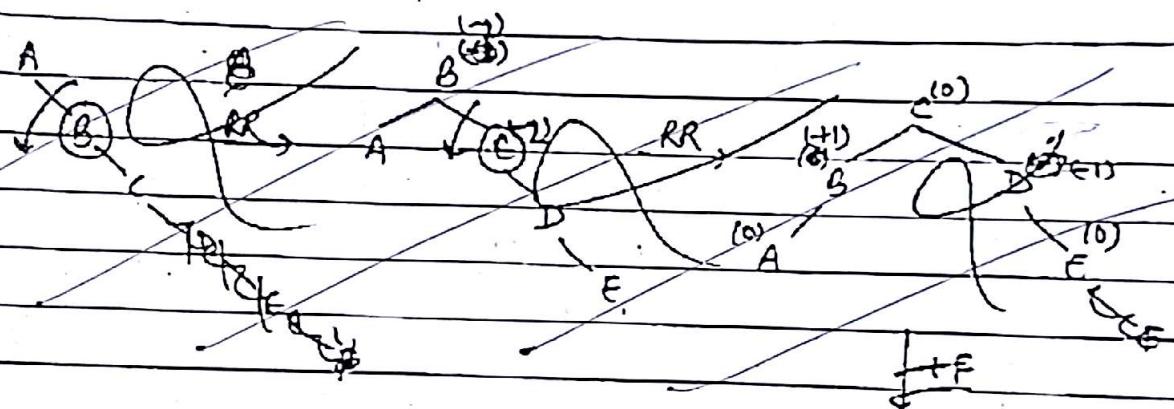


4) RL Rotation - insertion at left of right

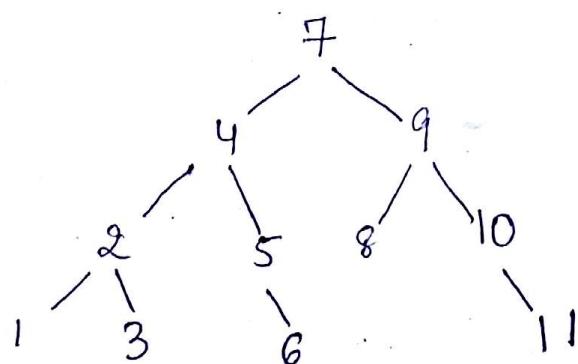
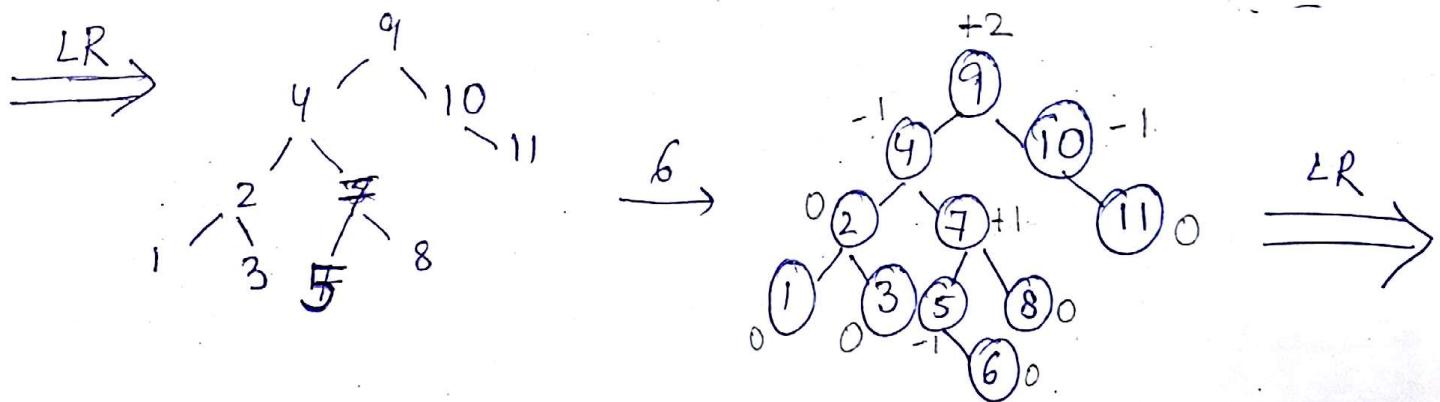
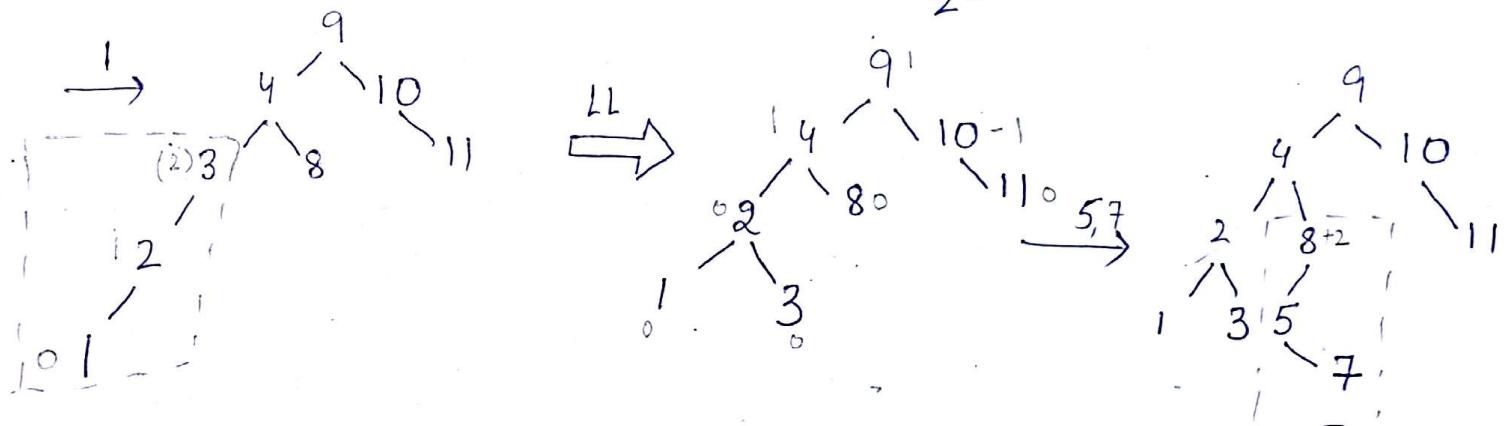
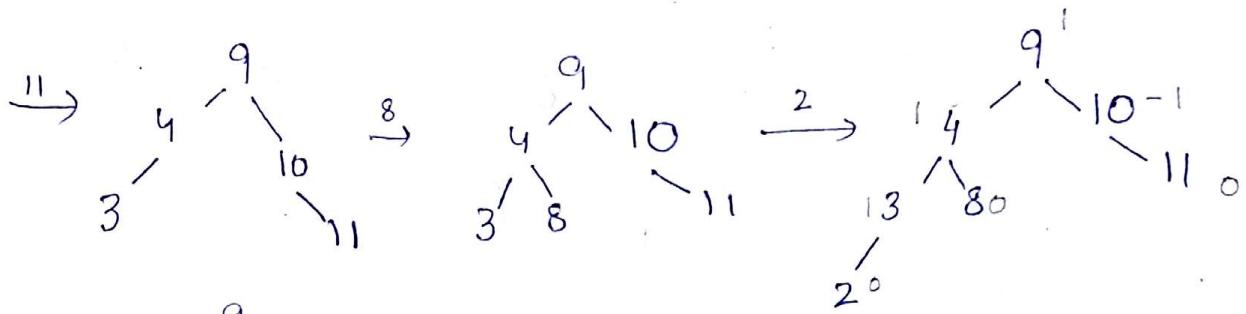
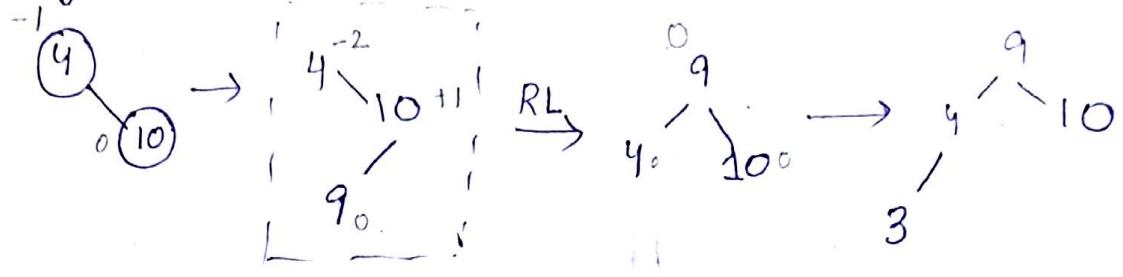
# RR followed by RL



Ques) A, B, C, D, E, F → form an AVL Search Tree



eg. 4, 10, 9, 3, 11, 8, 2, 1, 5, 7, 6



Q. e.g. 55, 66, 77, 15, 11, 33, 22, 35, 25

