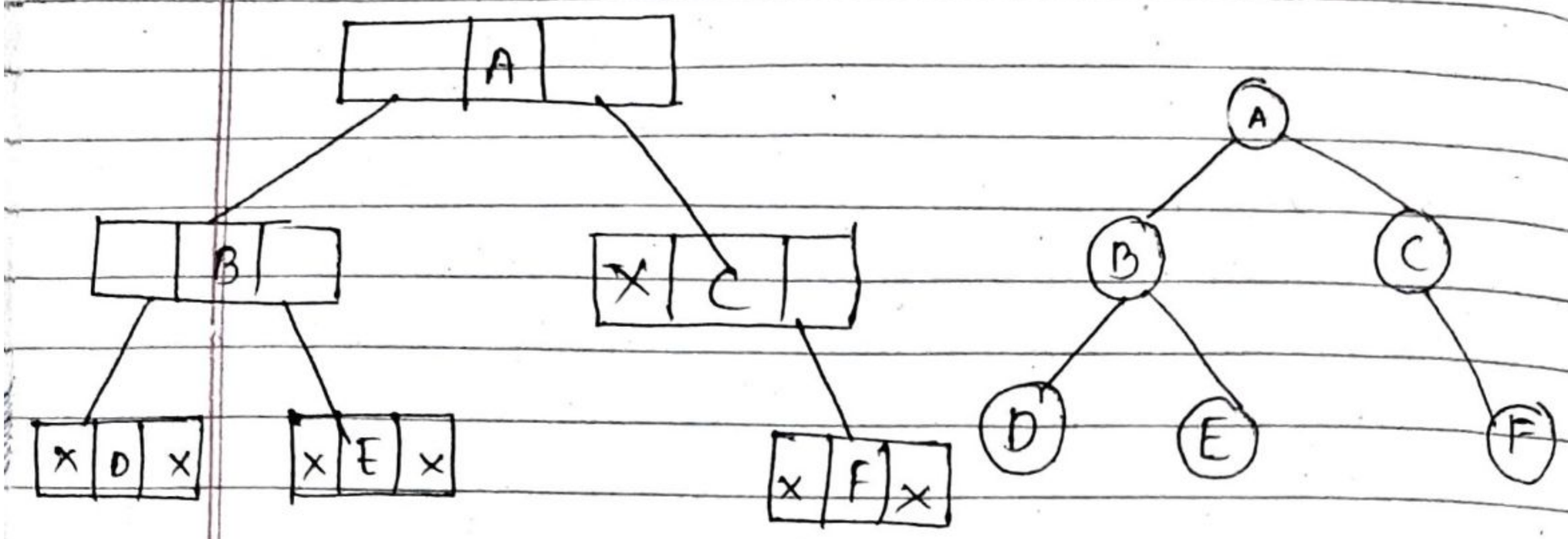
TREES

(use to file system, routing etc)

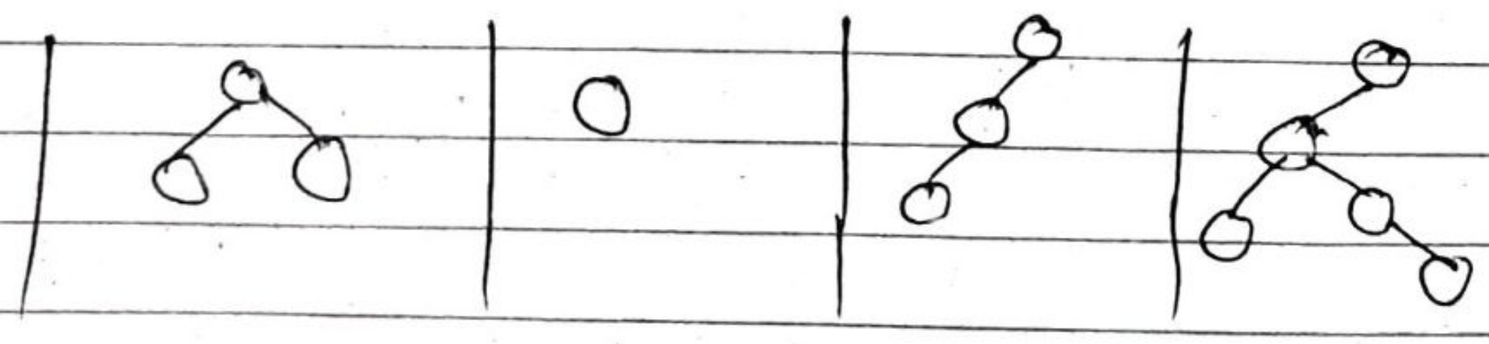
Tree can be defined as a collection of nodes linked together to simulate a hierarchy.




```
Struct   node
{
  char/int data
       struct  node *left;
       Struct  node * right;
};
```

BINARY TREE

(each node can have atmost a 2 children)



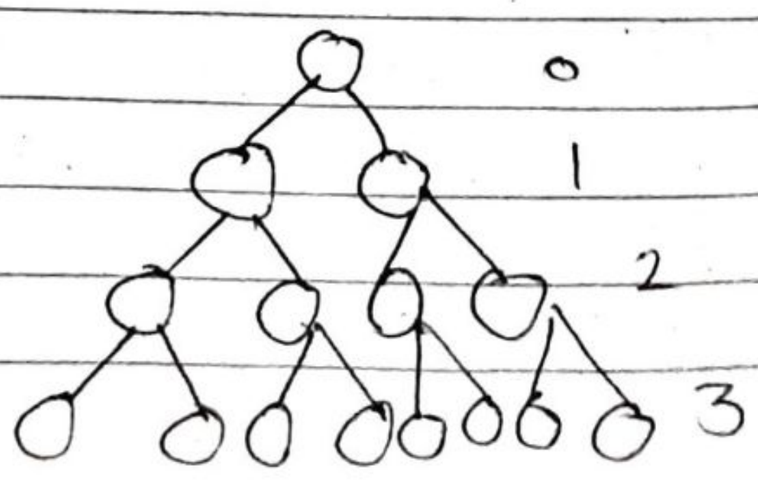
no. of node $= 2^{h+1} - 1$

$= 2^{3+1} - 1$

$= 2^4 - 1$

$= 16 - 1$

$= 15$



DATE / /

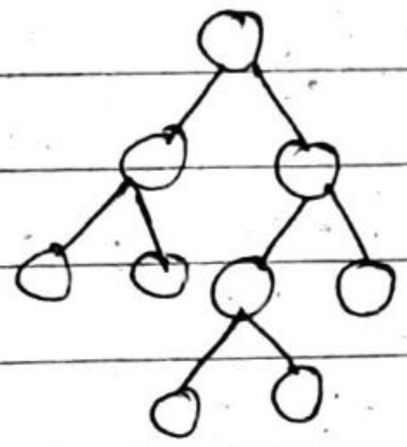
min node = h+1
= 3+1
= 4

max node = $2^{h+1} - 1$

Types of trees

1) Full proper BT

each node containing either 0 or 2 child.

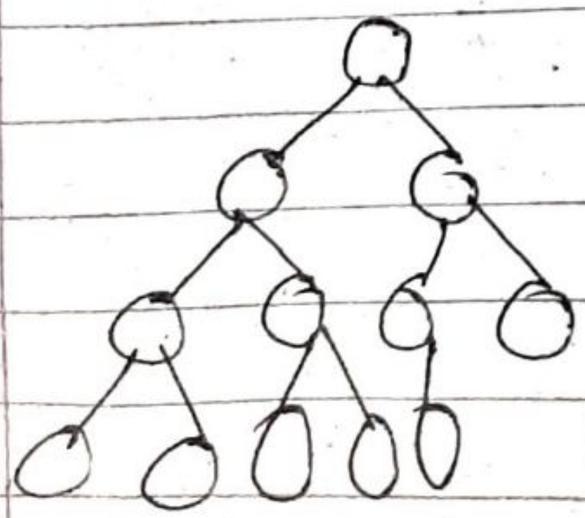


Min node = 2h+1

Max node = $2^{h+1} - 1$

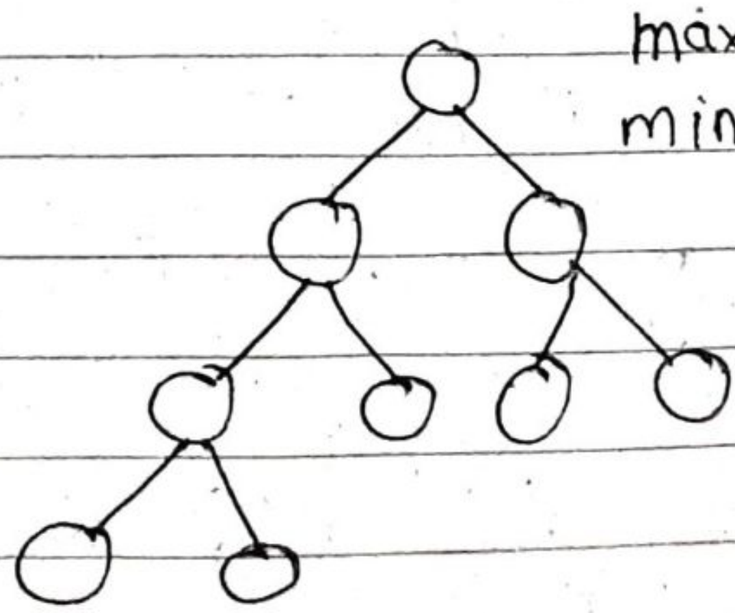
2) Complete binary tree

All its level, except the last level, have the maxmimum numbers of the possible nodes and all of the nodes of the last level appears as far left as possible.



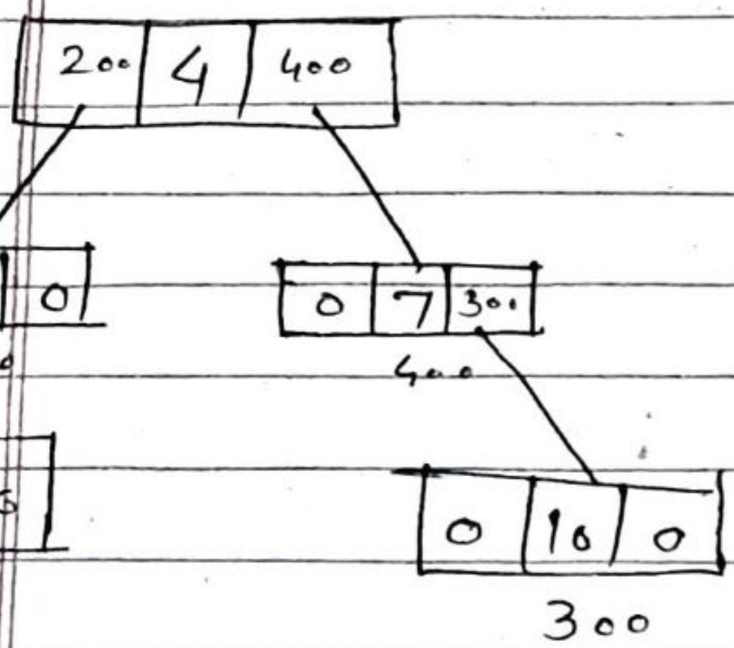
max node = $2^{h+1} - 1$

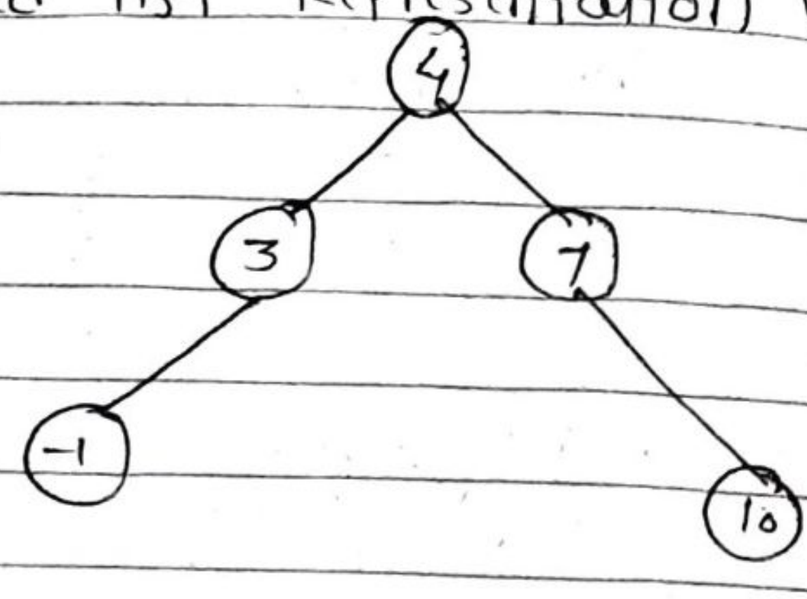
min node = 2^h



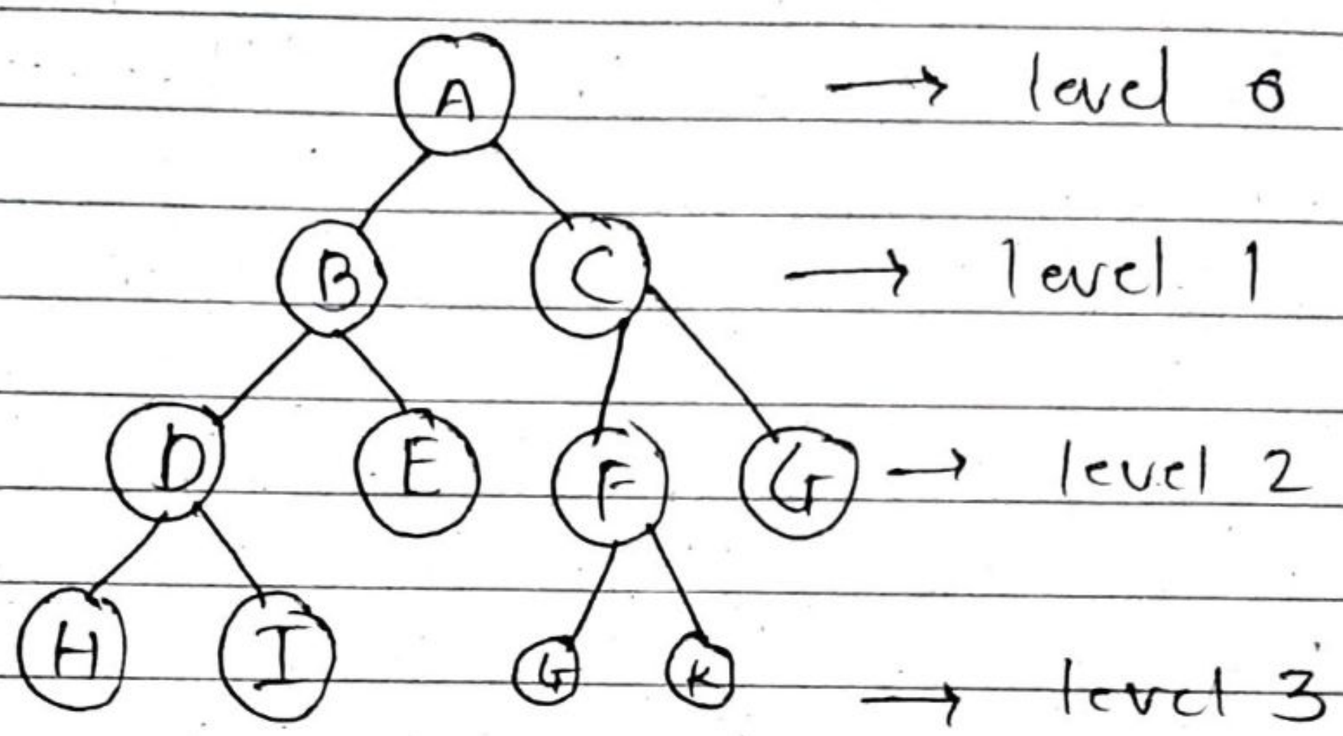
Binary tree implementation

Linked list Representation ( )





Array Representation ( )



Case I :

A	B	C	D	E	F	G	H	I	-	-	G	K	-	-	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- If a node is at i^{th} index
  - left child would be at $[(2*i)+1]$
  - Right $[(2*i)+2]$
  - parent $\left[\frac{(i-1)}{2}\right]$ floor value

Case II.

A	B	C						2
1	2	3	—	—	—	—	—	9

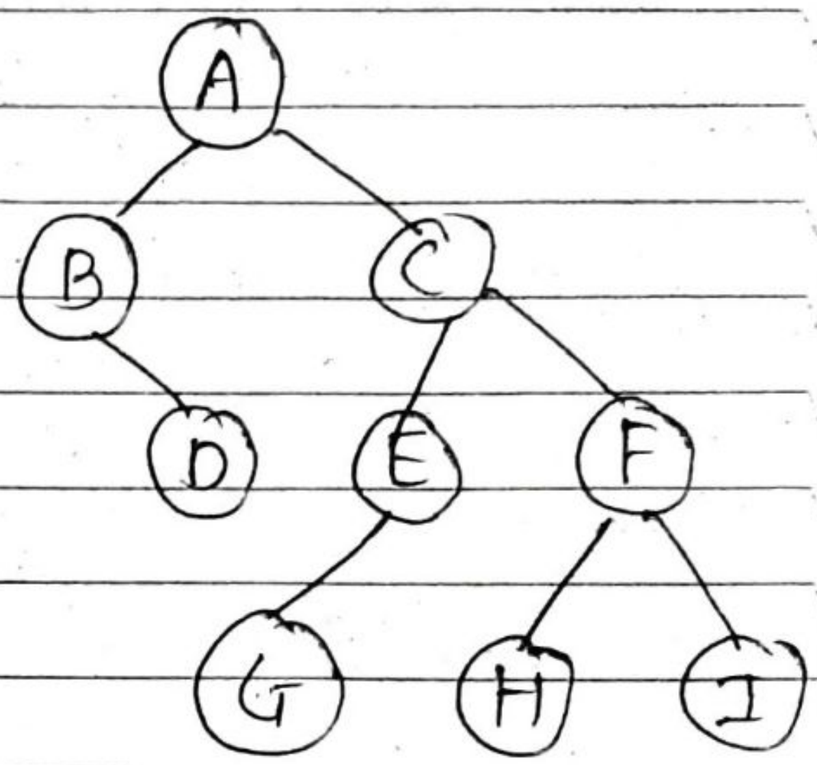
- left child would be at $[(2*i)]$
- Right $[(2*i)+1]$
- Parent $\left[\frac{i}{2}\right]$

PAGE No.
DATE / /

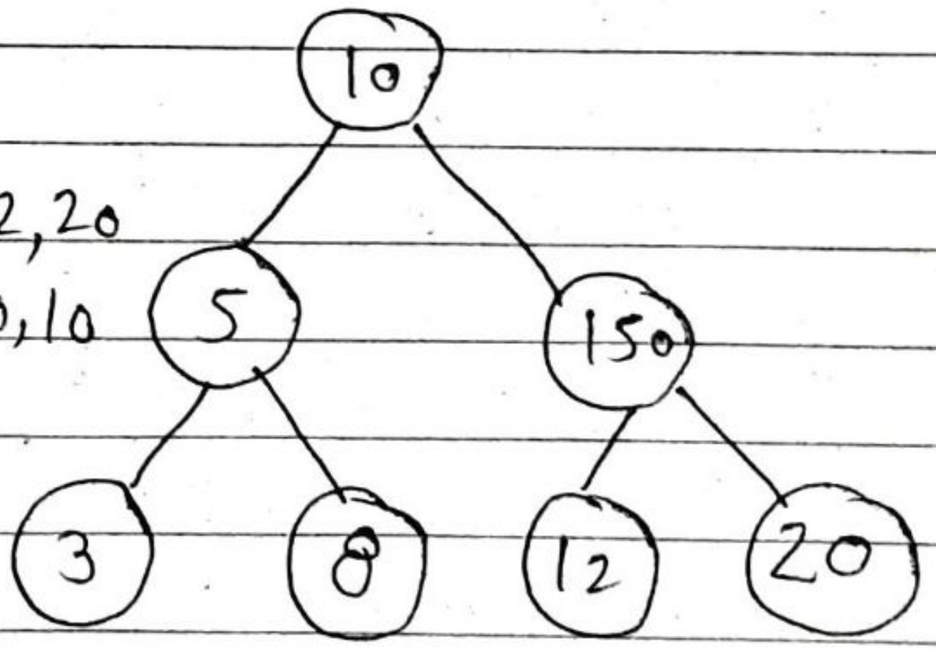
Binary tree traversal

Inorder – Left Data Right
preorder – Data Left Right
postorder – Left Right Data

Inorder : BDAGEGHFI
preorder : ABDCEGFHI
postorder : BBGEHIFCA



Inorder : 3,5,8,10,12,150,20
preorder : 10,5,3,8,150,12,20
postorder : 3,8,5,12,20,150,10



Inorder Algo :

DATE / /

1) Inorder Algo :

```
Void inorder (node *root)
{
    if (root != NULL)
    {
        inorder ( root→left );
        cout << root→data;
        inorder (root→ right);
    }
}
```

2) preorder Algo

```
void preorder (node *node)
{
    if (root ! NULL)
    { cout<<root→ data;
        preorder (root→left);
        preorder (root→ right);
    }
}
```

3) Post-order Algo

```
Void postorder (node* temp)
{
    if (temp == NULL)
        return ;
    else {
        postorder (root→left );
        postorder (root→ right);
        cout <<root→ data;
    }
}
```

DATE / /

* Construct a BT from preorder and Inorder.

→ preorder : 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7
Inorder : 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7

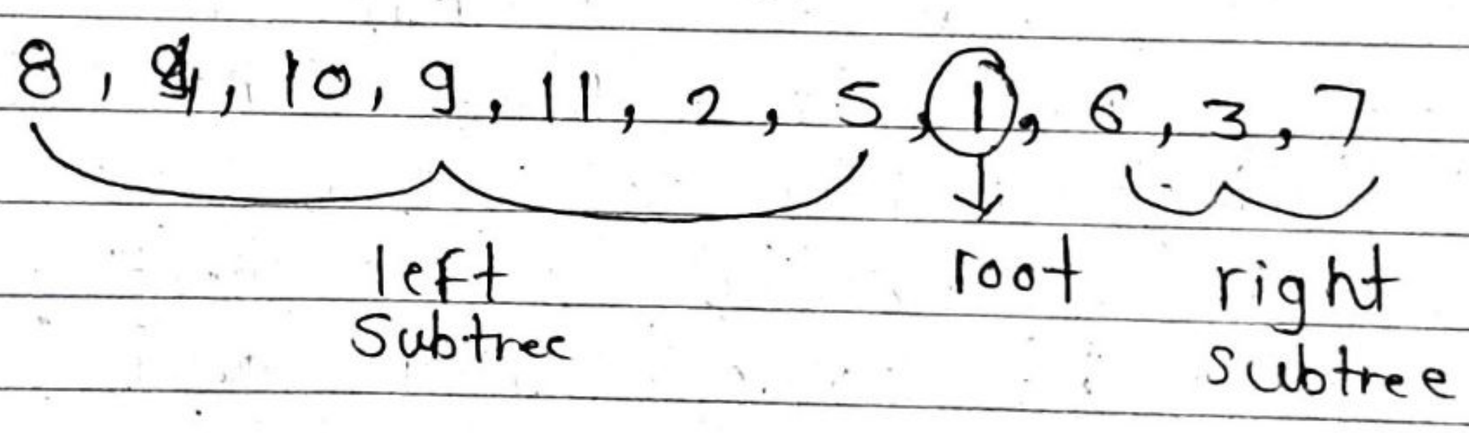
Firstly we've to find root of BT.
by using preorder traversal we can find root.

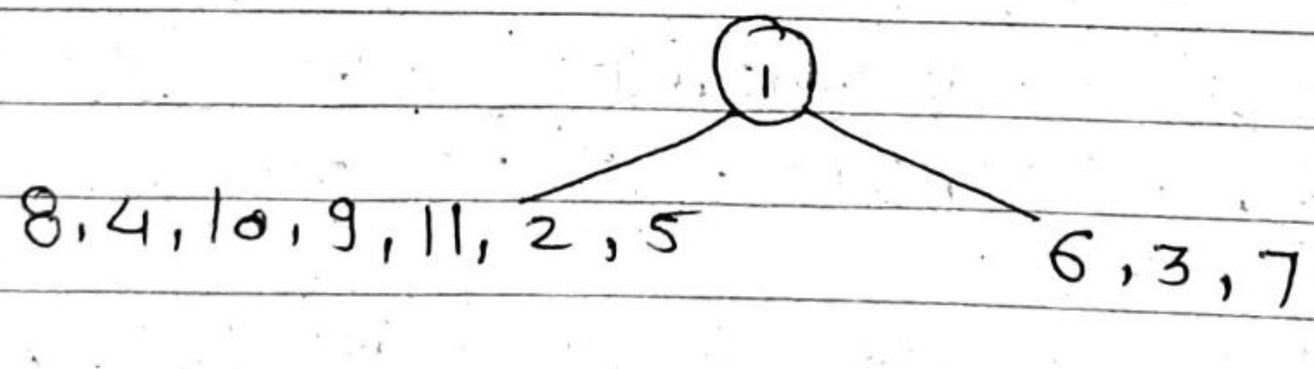
as preporder is Root → L → R

Here 1 is the root. of BT

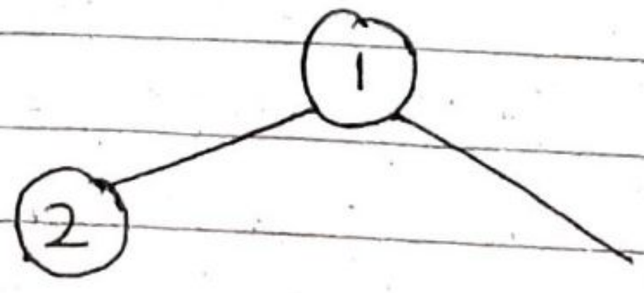
(1)

Now will find left and right child of the root from given Inorder.



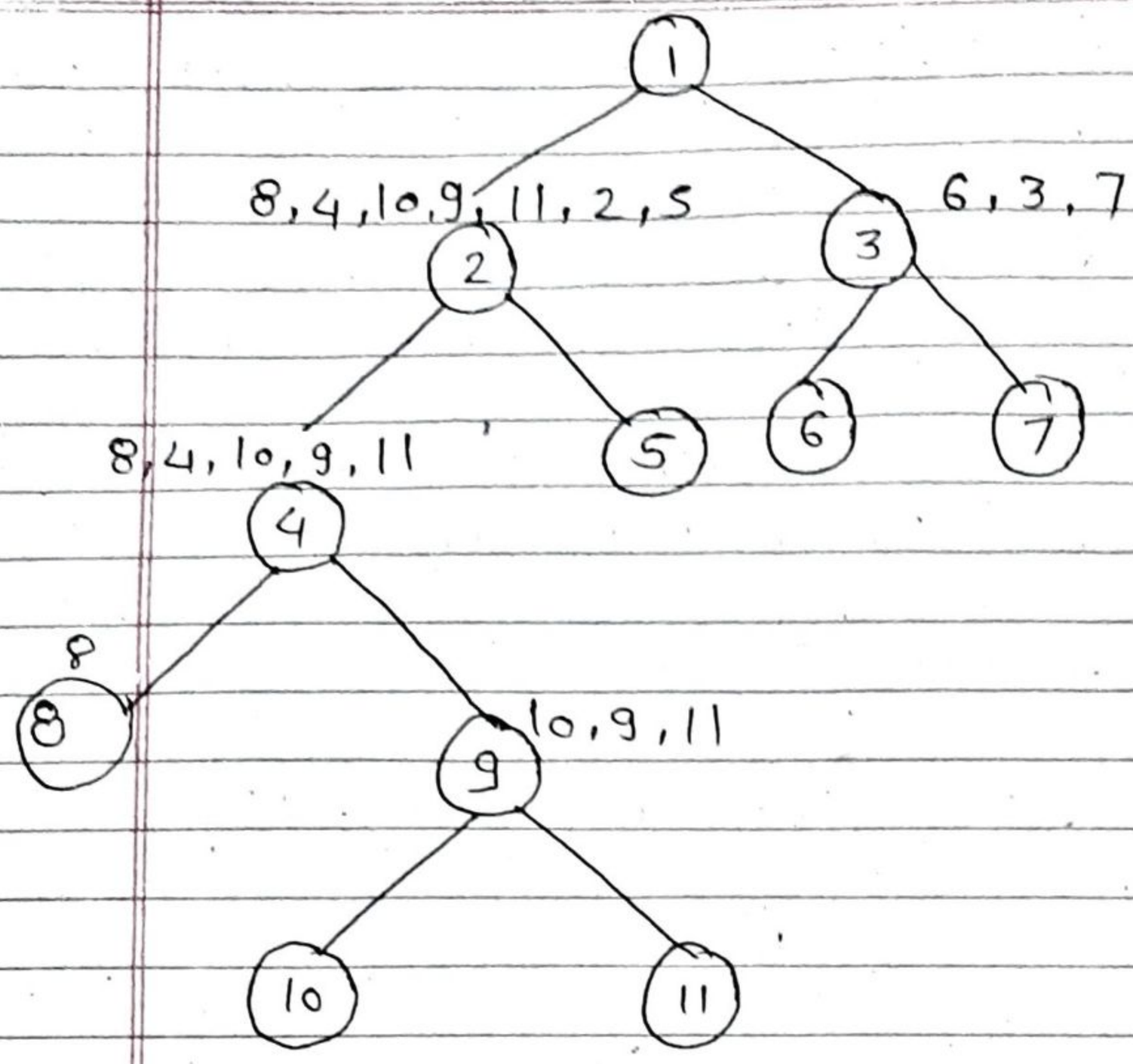


for finding root of Lsubtree will use preorder and scan from left to right.



to find Lchild & RLidld scan Inorder..

search predessor & successor



- To find root search preorder &
- To find its left & right child scan the Inorder & search its predcessor and succesor and check in preorder which comes first and form a Tree.

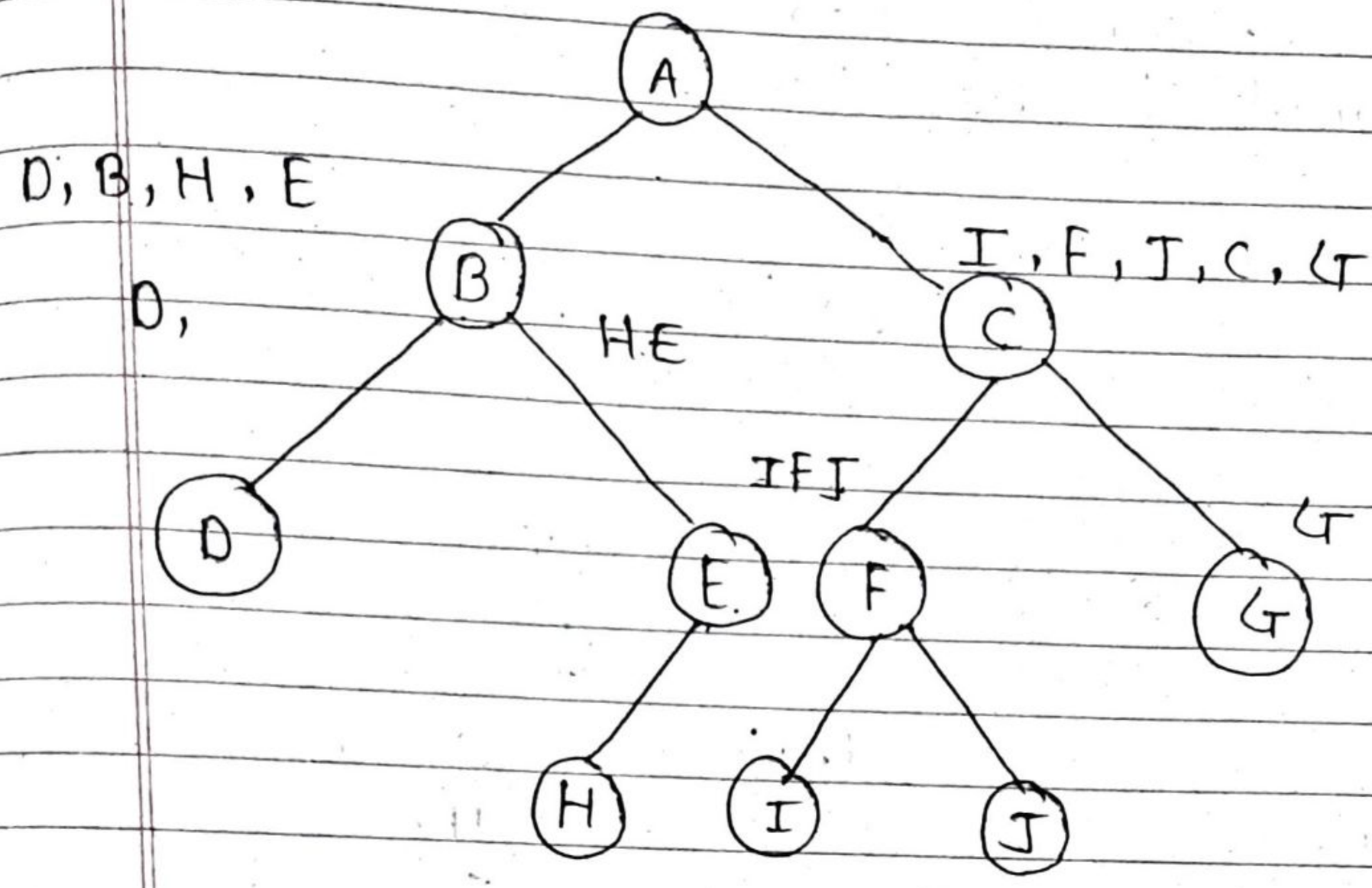
2) Inorder → DBHE(A)IFJCG
preorder → ABDEHCFIJG

root from preorder will be A.

(A)

To find its left & right child Scan Inorder & find its predcessor and succesor once you find its predcessor & succesor check in preorder

which comes first. repeat -- until the whole tree is formed.



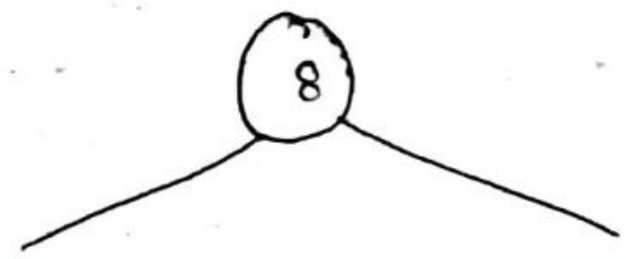
* Construct a BT from Postorder and Inorder.

Postorder : 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
Inorder : 9, 5, 1, 7, 2, 12, (8), 4, 3, 11

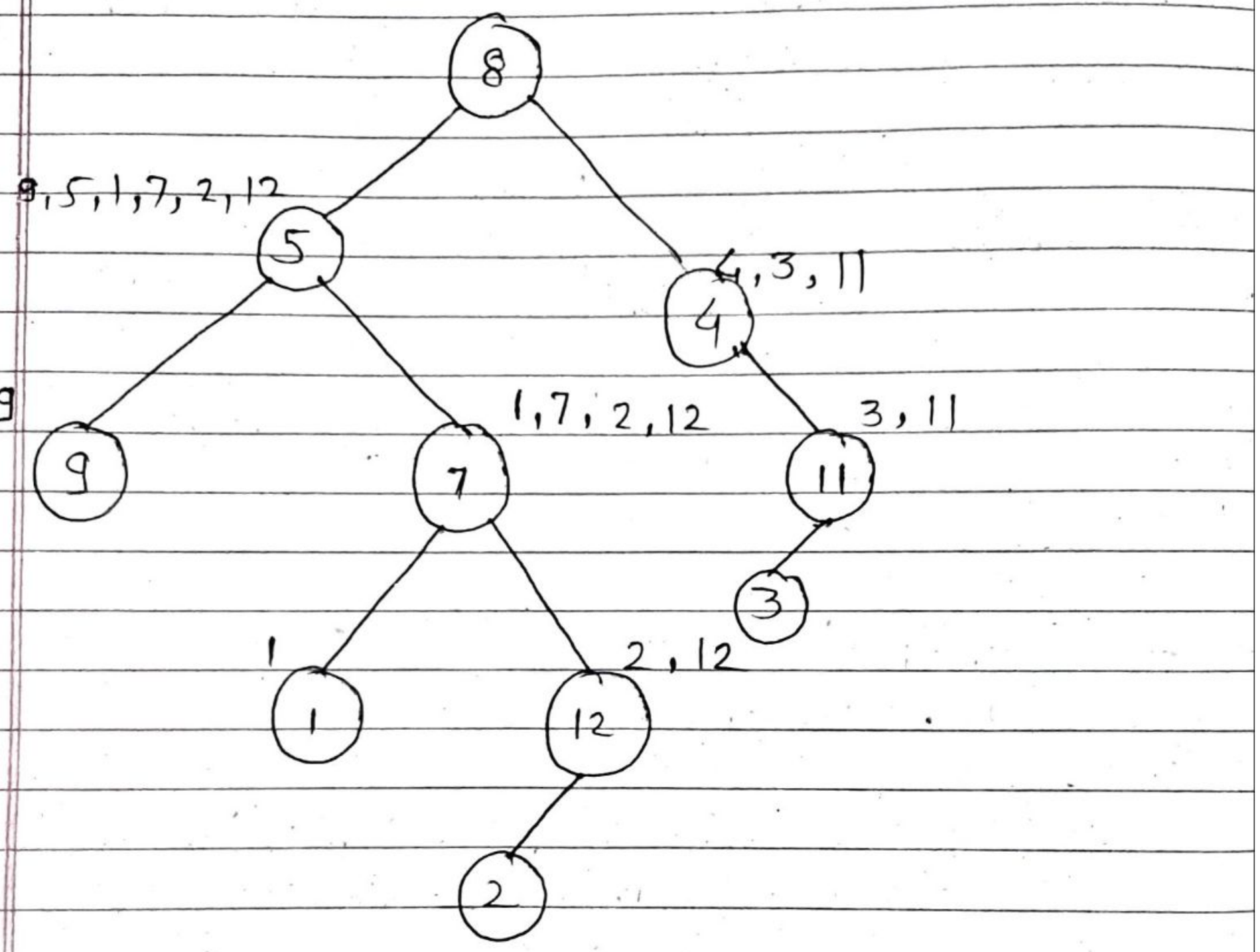
From postorder will get root 8.
As postorder is root → left → right.

(8)

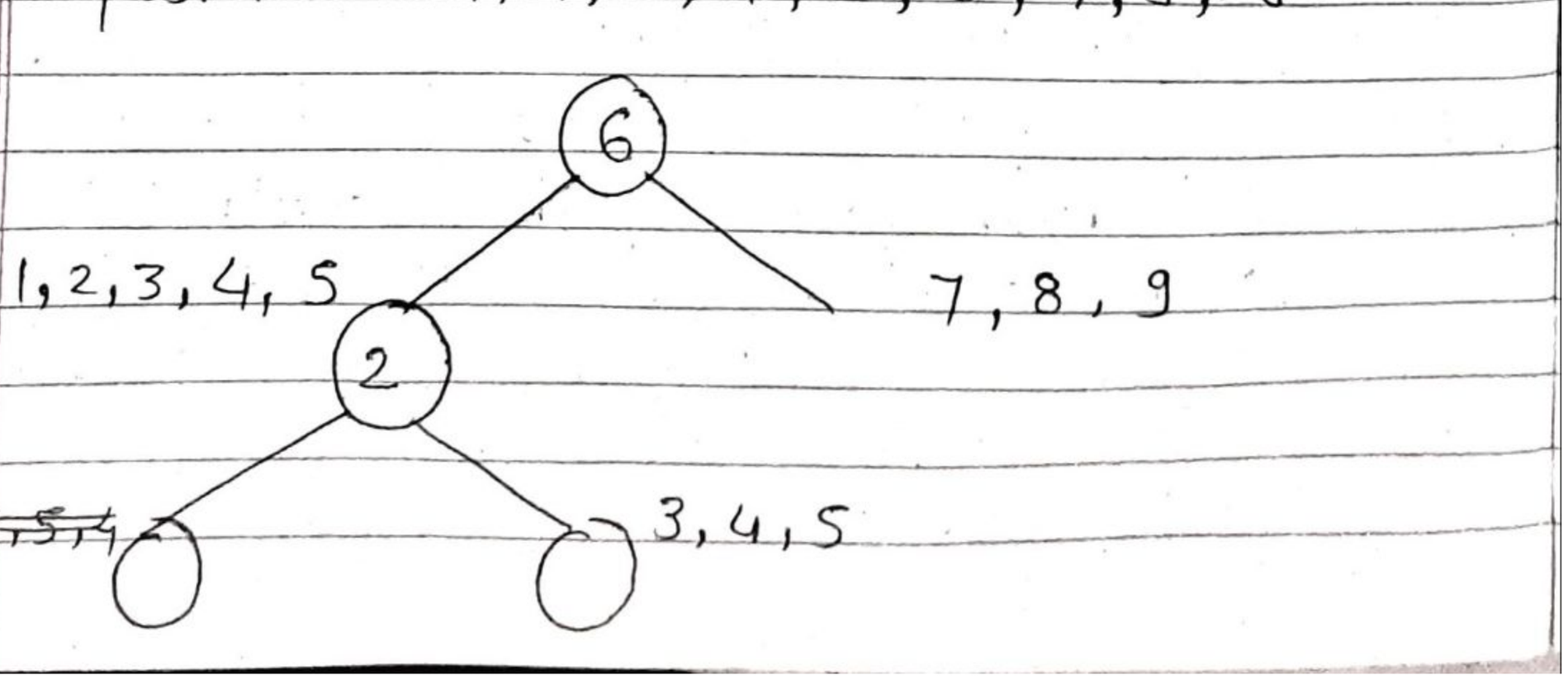
To find left & right subtree will scan Inorder and will locate 8. on the left of 8 all are predcessor & on left of 8 are successor.

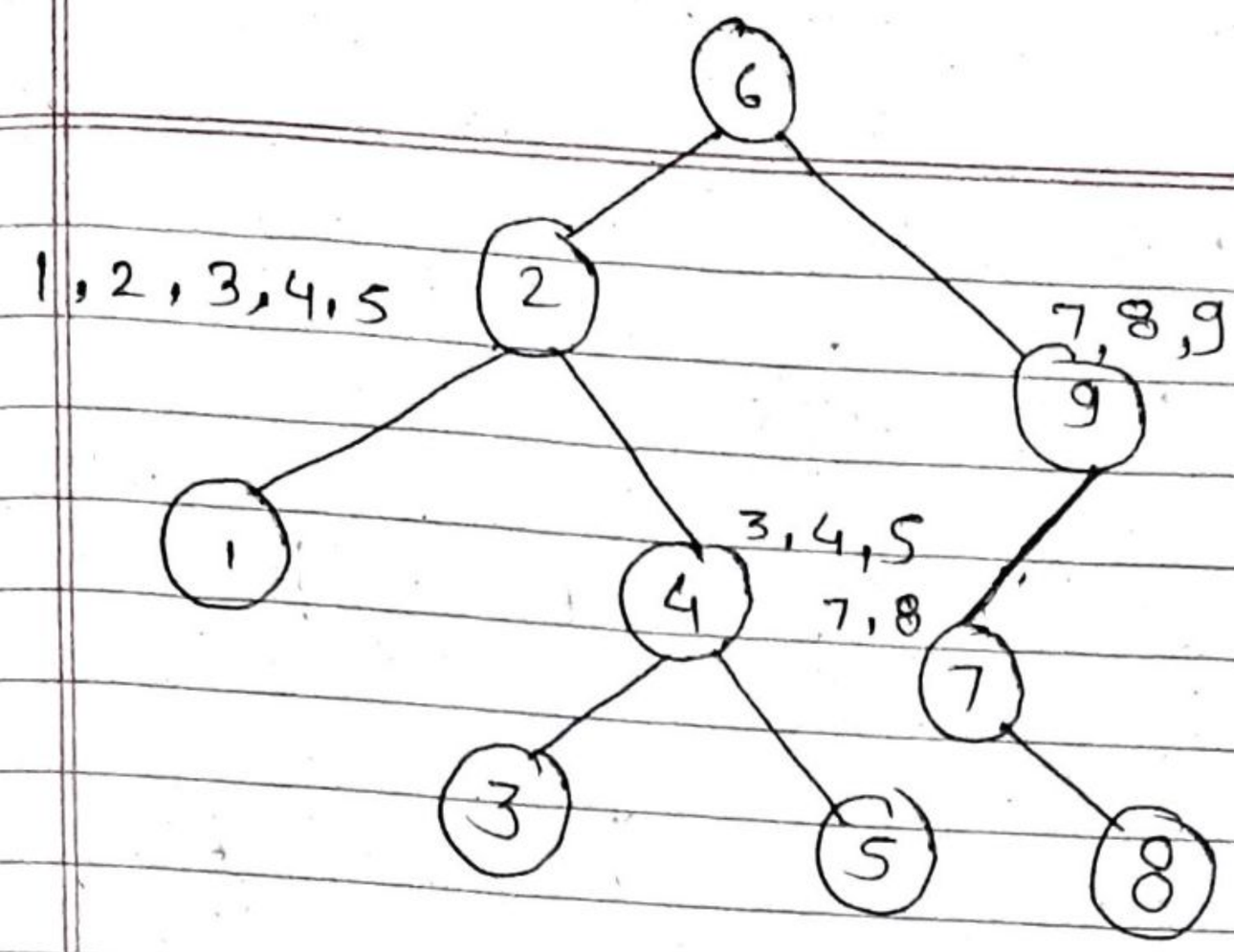


So, once will locate now scan postorder from right to left & see which elements comes first & mark it respectively.



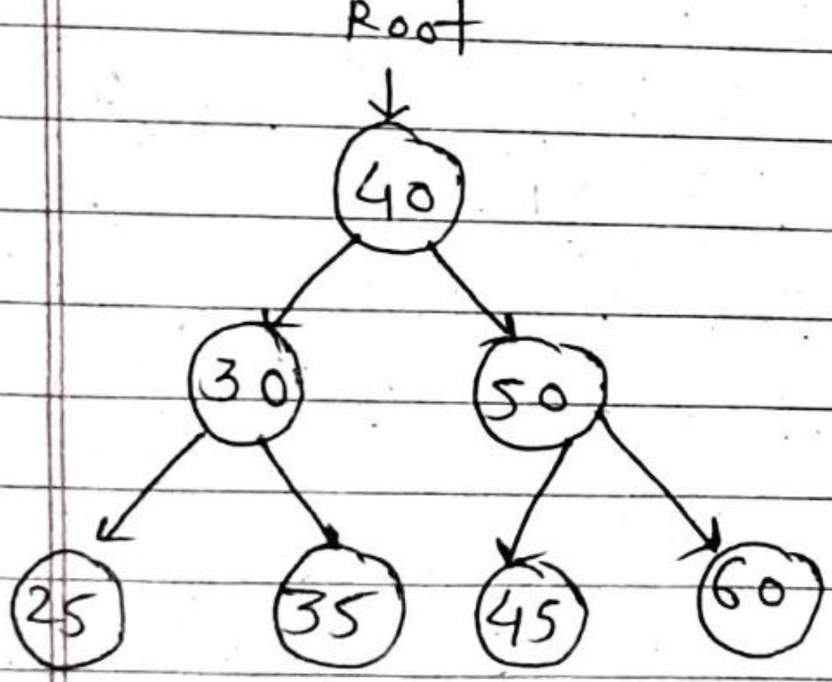
(2) Inorder : 1, 2, 3, 4, 5, 6, 7, 8, 9
postorder : 1, 3, 5, 4, 2, 8, 7, 9, 6



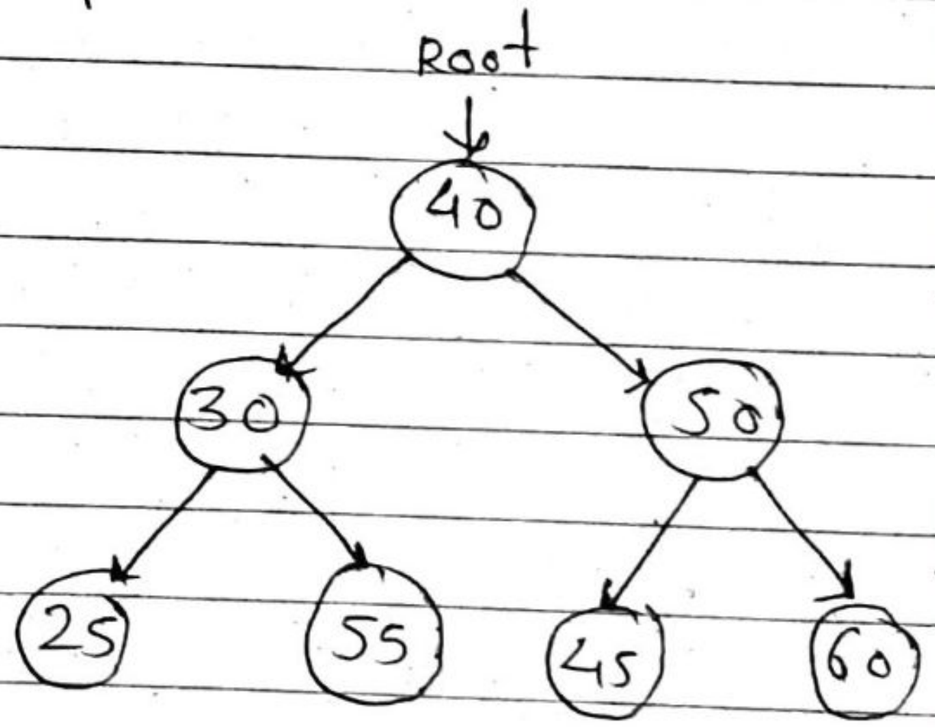


* Binary Search Tree

BST follows same order to arrange the elements. In BST the value of left node must be smaller than the parent node, & the value of right node must be greater than the parent node.



This is BST



Not a BST

- Searching is easy as we have hint that which subtree has the desired element.
- As compared to array & linked lists, insertion and deletion operations are faster.

PAGE No
DATE / /

* Creating BST. from given keys

4, 2, 3, 6, 5, 7, 1

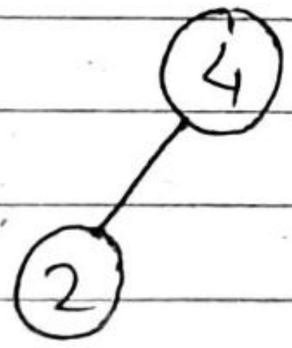
Step 1 : we will inser 4 as the root of the tree.

Step 2 : Then, read the next element, if it is smaller that the root node, insert it as the root of the left subtree, & move to the next element.

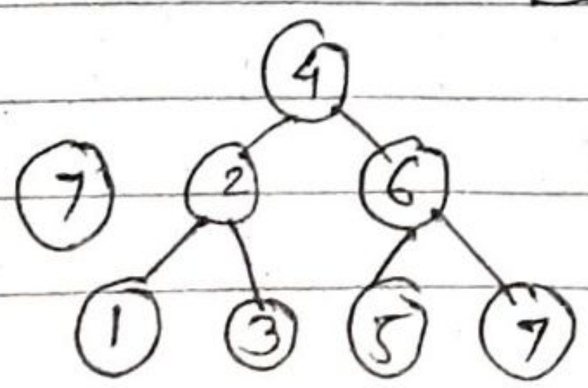
step3 : otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

1)

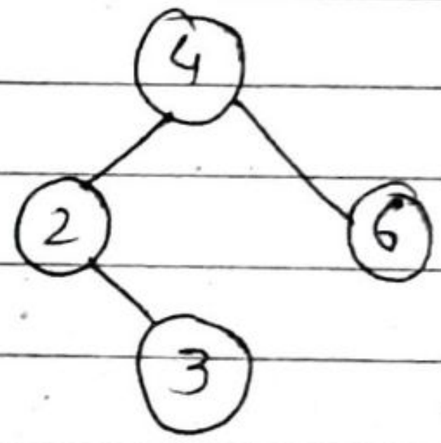
2)



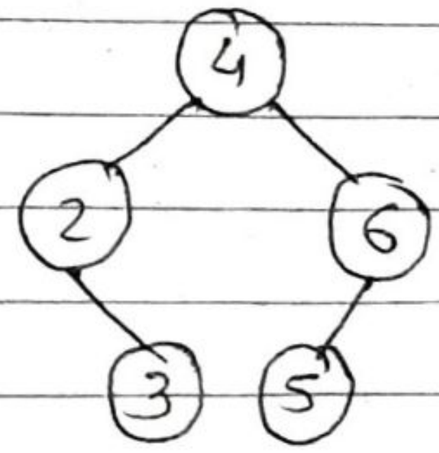
3)



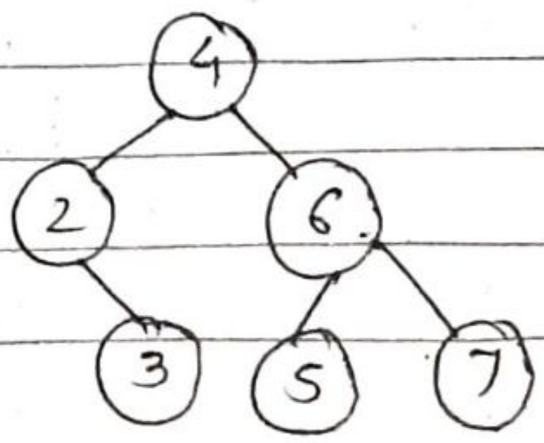
4)



5)



6)

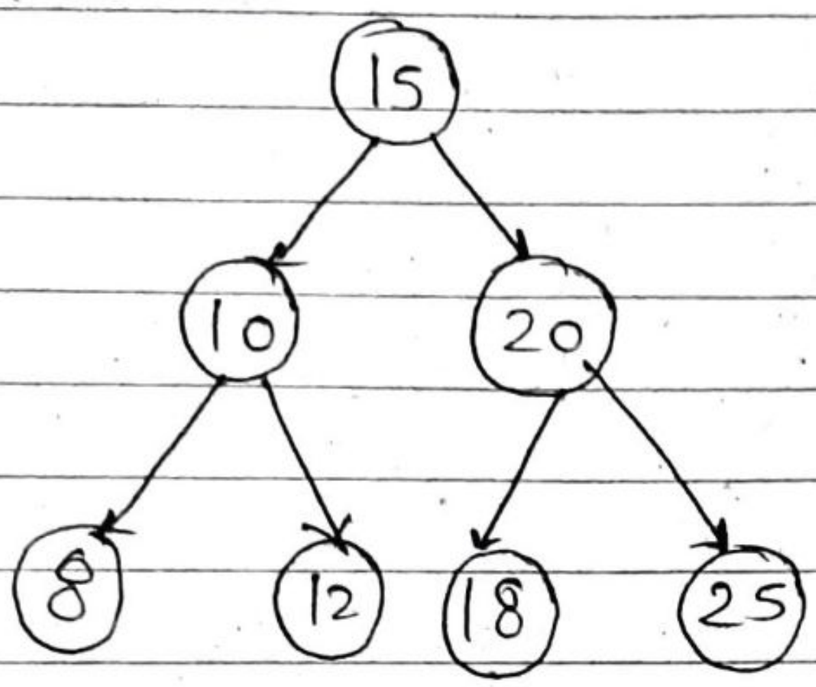


PAGE No.
DATE / /

* Delete the BST

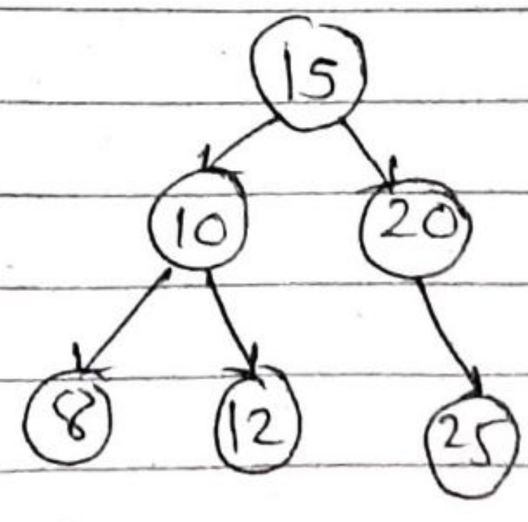
Case 1 : Deleting a node without children.

- Check whether node is left or right child of its parent.
- Set the left or right pointer to null for the parent accordingly.
- Free the memory of node.



Delete t = 18

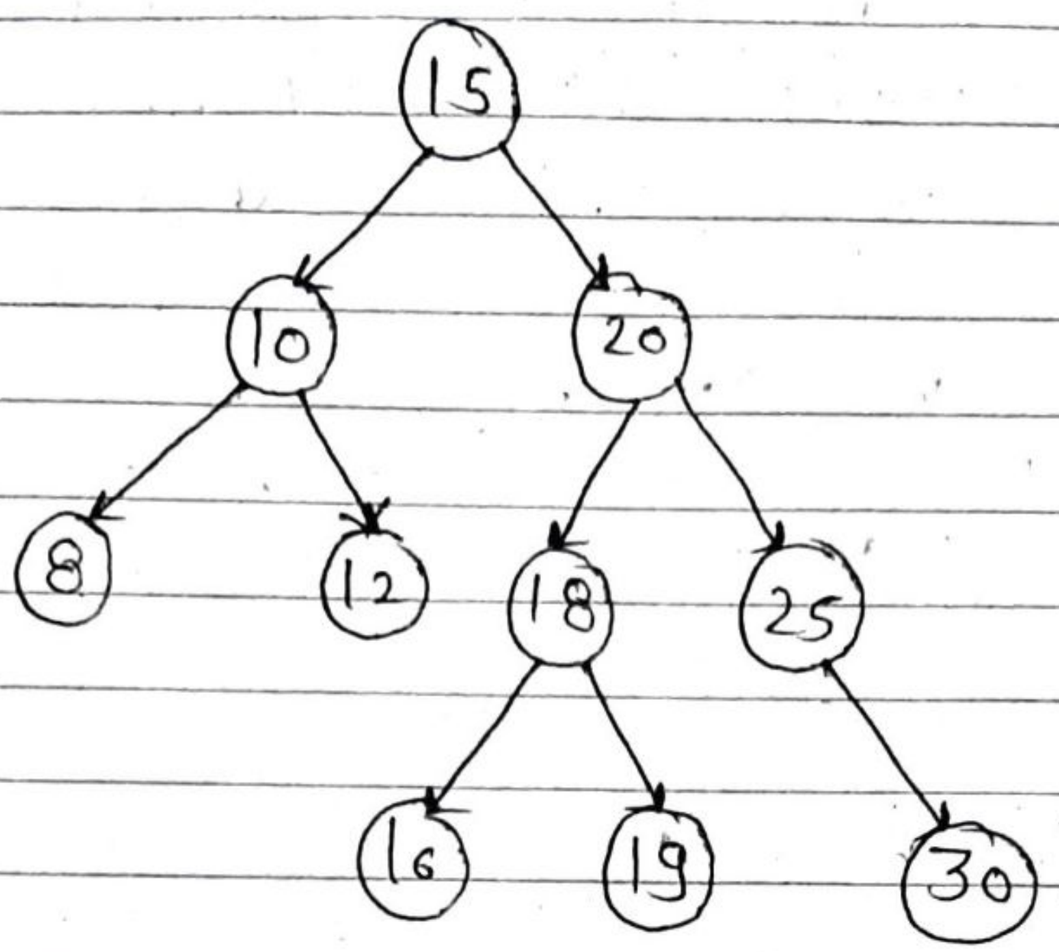
```
if (t→left == NULL && t→right == NULL)
{
    if(!P! = NULL)
    {
        if( P→left == t)
        {
            P→left = NULL;
        }
    else if ( P→right == t)
    {
        P→ right = NULL;
    }
    else
    {
        root = NULL;
    }
} Free (t);
```



PAGE No
DATE / /

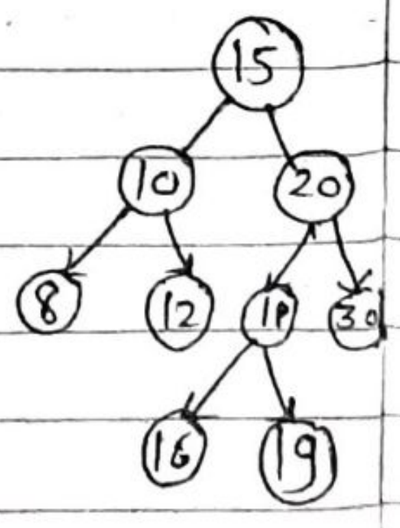
Case 2: The node has only right child

- Search the node 't' and parent 'p'
- Replace the node 't' with its right child.
- Free the memory of t.



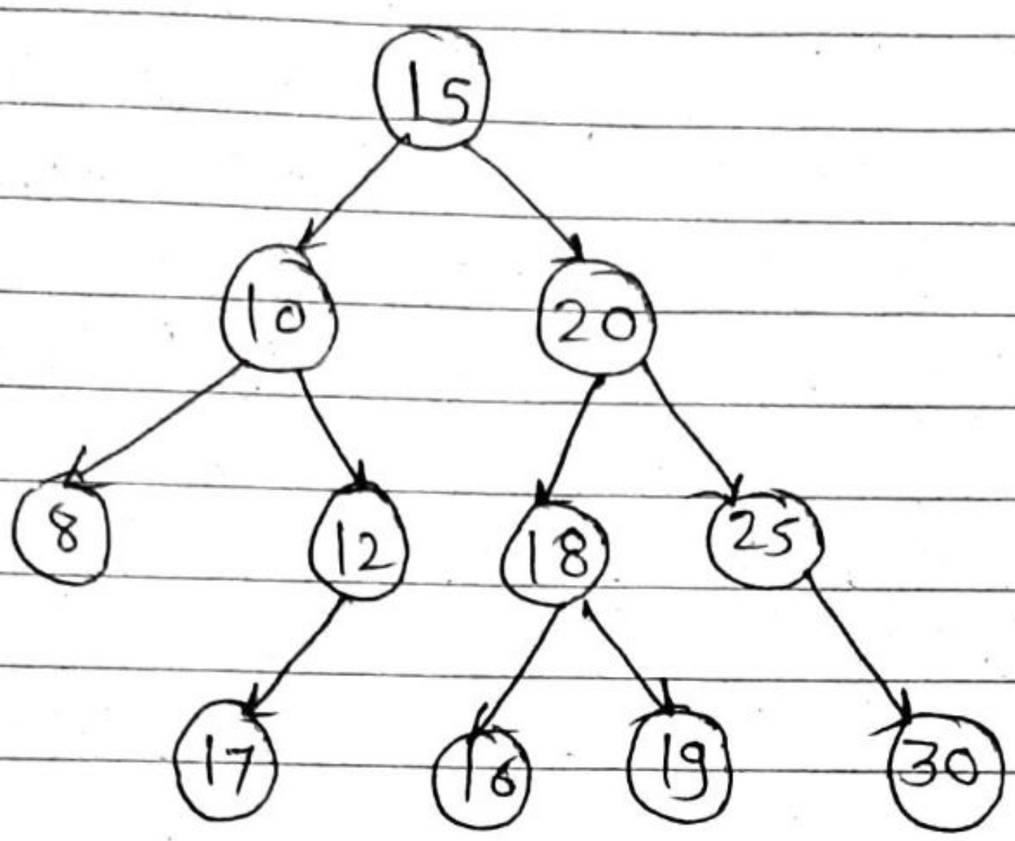
Delete t=25

```
if (t→left == NULL && t→right != NULL)
{
    if (p != NULL)
    {
        if (p→left == t)
        {
            p→left == t→right
        }
    else if (p→right == t)
    {
        p→right == t→right
    }
    else
    {
        root = t→right
    }
} }   Free (t);
```



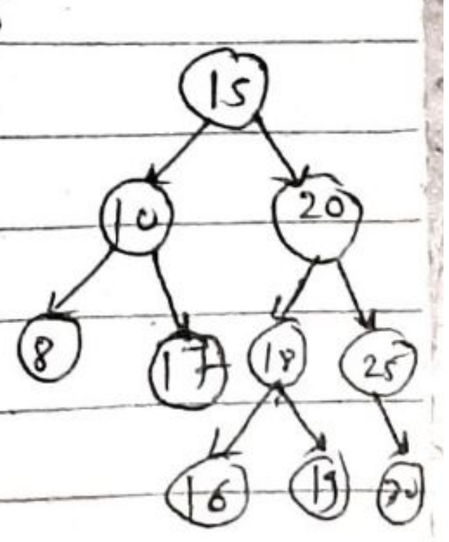
Case 3 : The node has only left child

- Search the node & parent.
- Replace the node with its lelf child.
- Free the memory of t.



Delete t = 12

```
if (t→left != NULL && t→right = NULL)
{
    if (P != NULL)
    {
        if (P→left == t)
        {
            P→left == t→left;
        }
    else if (P→right == t)
        {
            P→right == t→left;
        }
    else
        {
            root = t→left;
        }
    free (t);
}
```

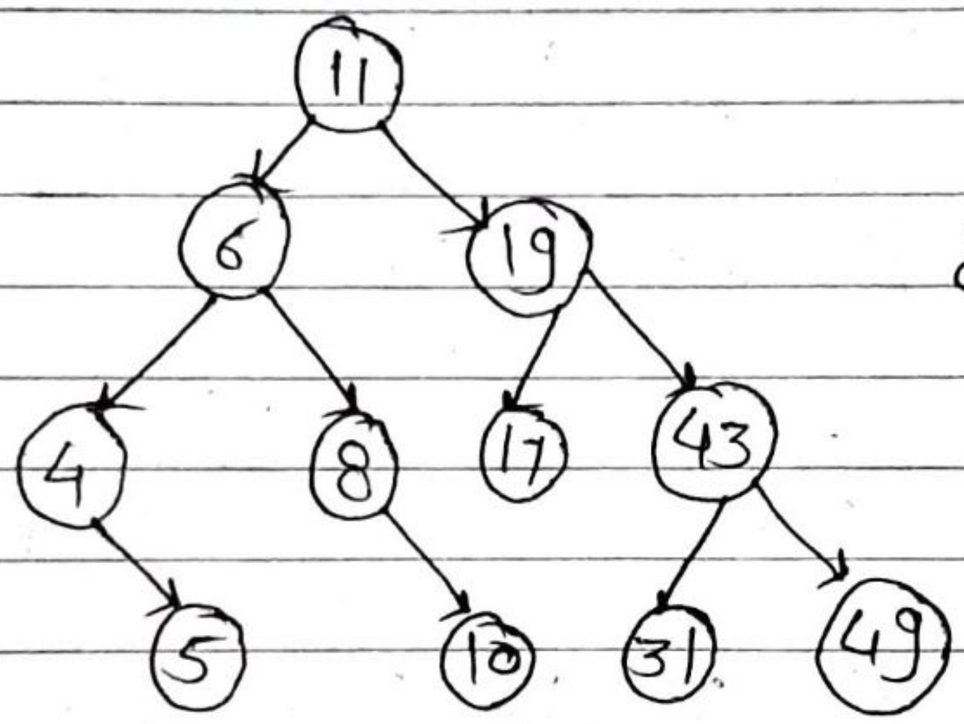


Case 4: The node has two children.

- Search the node.
- Replace the node with the inorder successor / inorder predcessor.

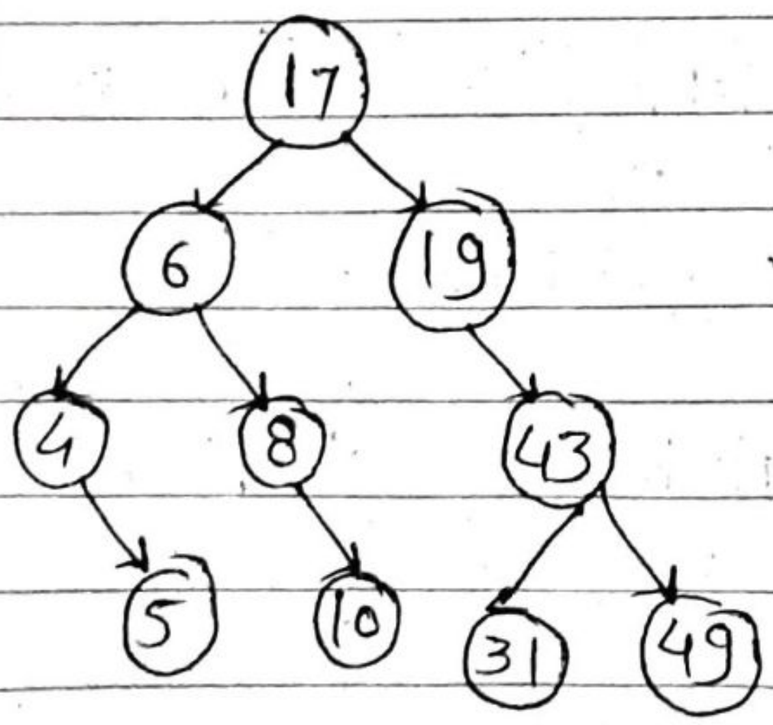
Inorder successor : minimum value of right subtree.

Inorder predcessor : maximum value of left subtree.

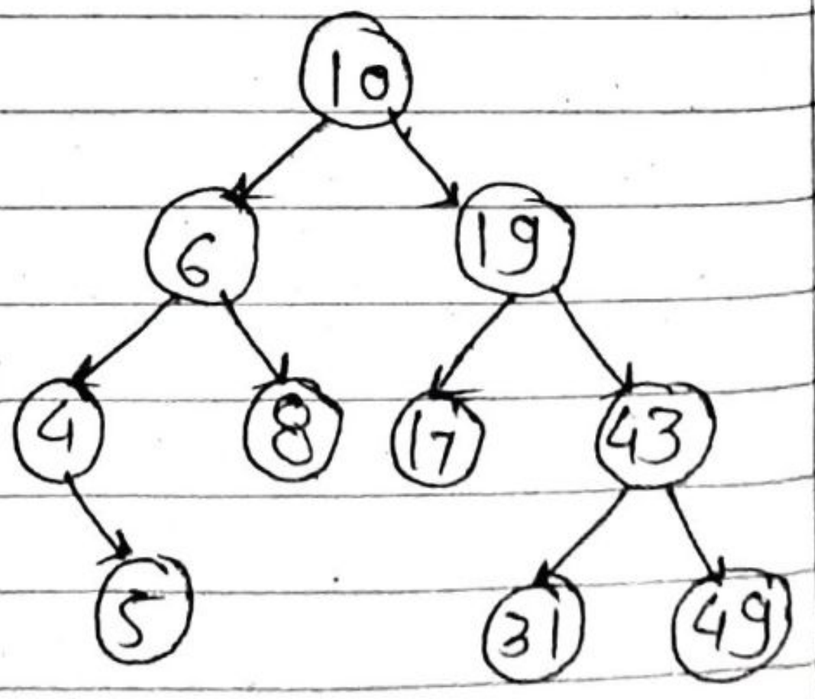


delete t = 11

Inorder successor



Inorder predeccessor



inorder of tree: 4,5,6,8,10,11,17,19,31,43,49

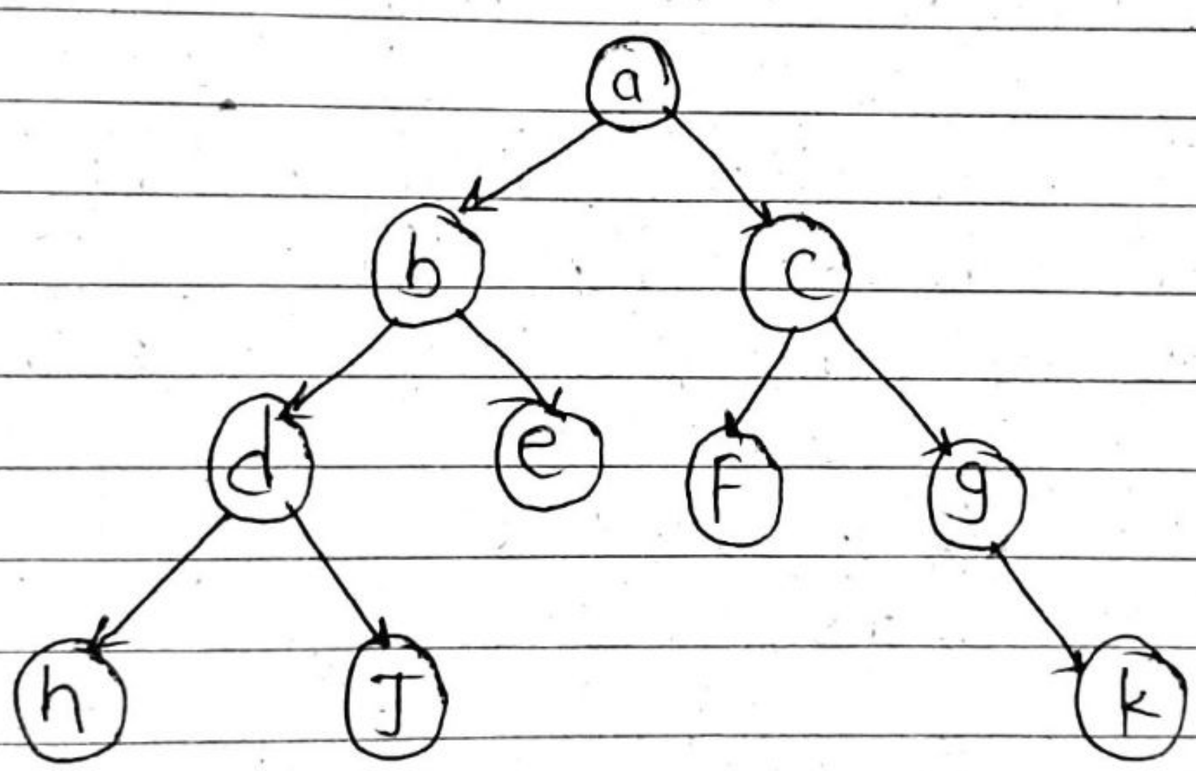
inorder Predm.

inorder successor

depth = level

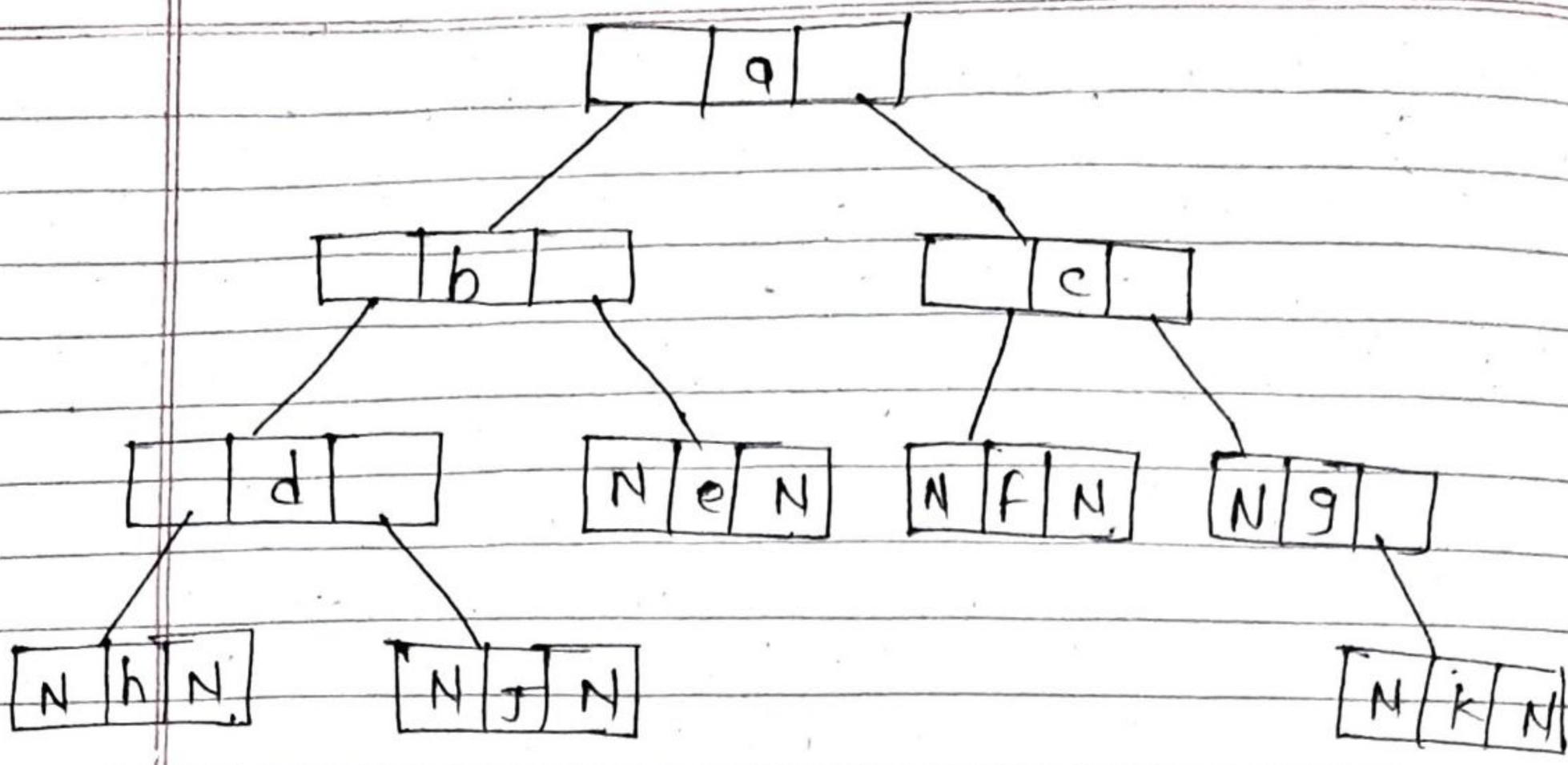
```
if (t->left != NULL && t->right != NULL)
{
        Node * inSuc = t->right;
        while (inSuc->left != NULL)
        {
                inSuc = inSuc->left;
        int val = inSuc->data;
        fnDelete (inSuc->data);

        t->data = val;
}       }
```

* Threaded Binary Tree.



Will first represent this tree in link list format.

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



Now, we know the TBT which are have many nodes that have an empty left child or empty right child or both.

So, we can utilize these fields, in such a way so that the
- empty left child of a node points to its predecessor.
- empty right child of a node points to its succesor.

we will use Inorder here.

h, d, j, b, e, a, f, c, g, k

Now, to convert to BT to TBT

Step I : keep the leftmost and rightmost NULL pointers as NULL.
because they are null

NULL ← h, d, j, b, e, a, f, c, g, k → NULL

step 2: change all other NULL pointers as
Left pointer → Inorder predcessor
Right pointer → Inorder Successor.

Inorder successor of right pointers.
h → d , I → b , e → a , f → c

Inorder predecessor of left pointers.
I → d , e → b , f → a , g → c , k → g

Now, this pointers are pointing to the address of thier ancestors. And others are pointing to thier child.
To make the difference, we will make use of flag 0 & 1.

- If pointers i.e. Left or right pointers are pointing to ancestors then it will be 0.
- If pointers i.e. Left or right pointers are pointing to child then flag will be 1.

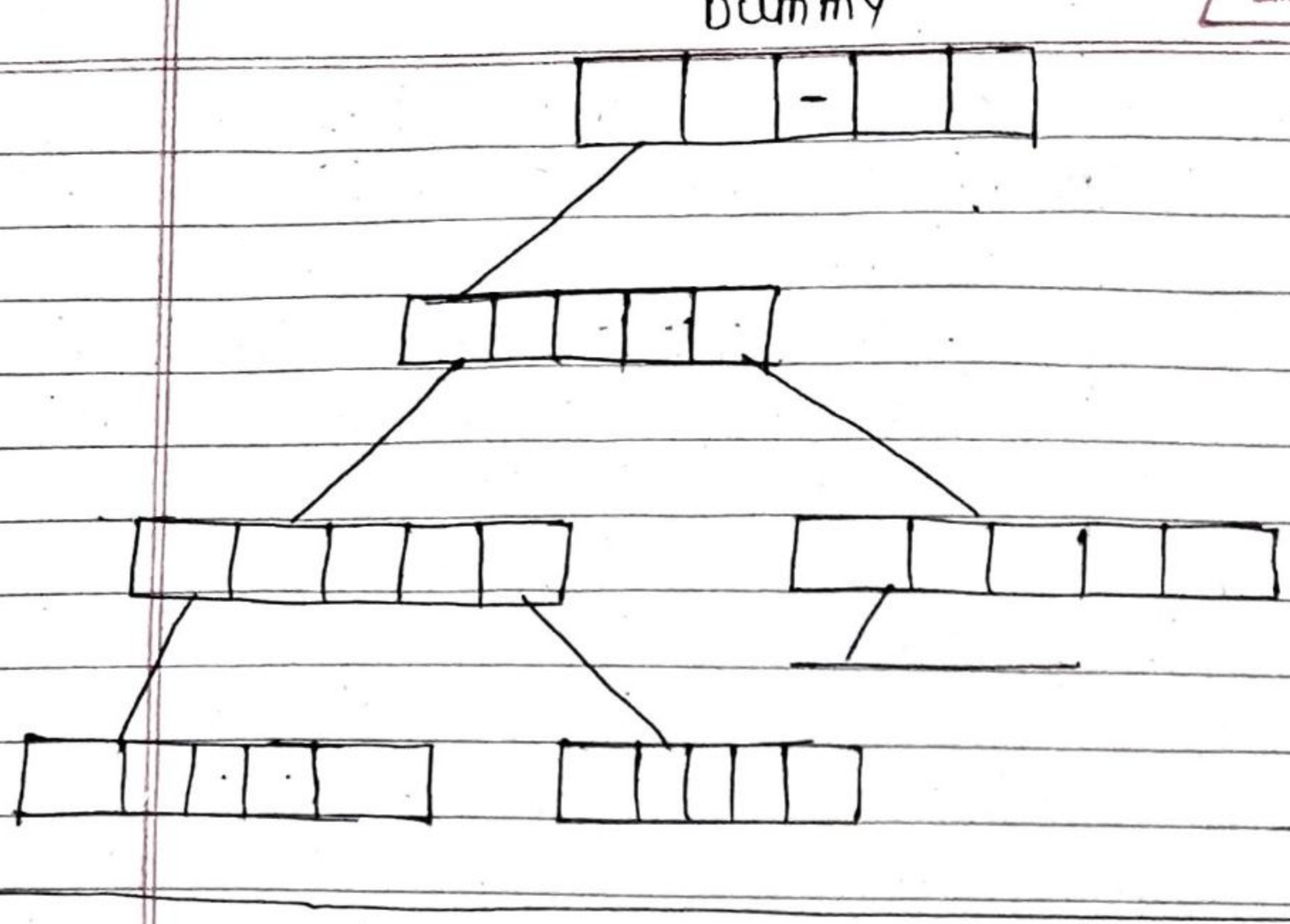
- Now, the leftmost & rightmost pointers are not pointing to anyone. For that we will create one dumy node where they will point.

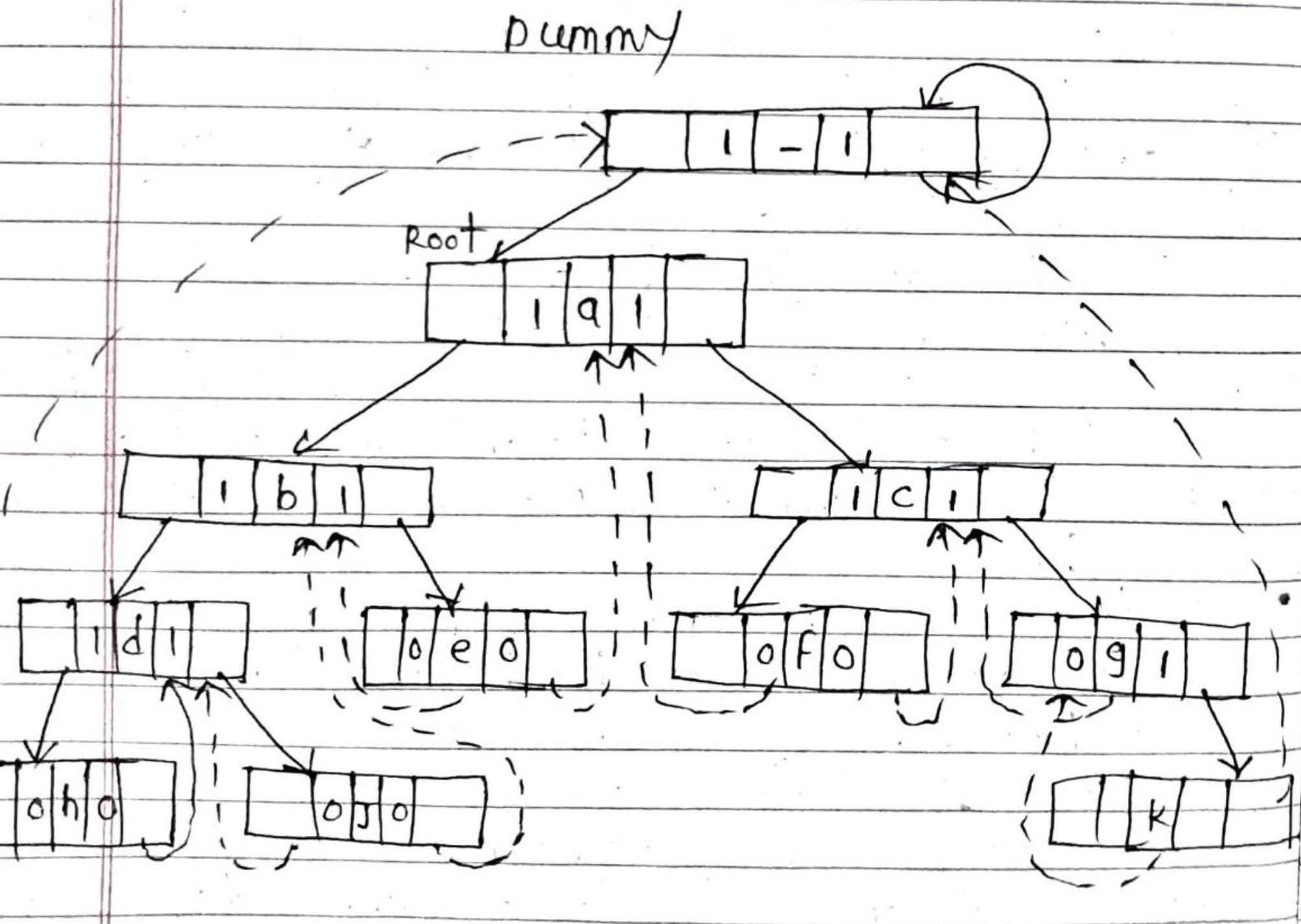
- Structure of a node now

left pointer	left Flag	data	right Flag	right pointer

- There will not data in dummy.
- The right pointer of dummy will point to itself.

PAGE No.
DATE / /



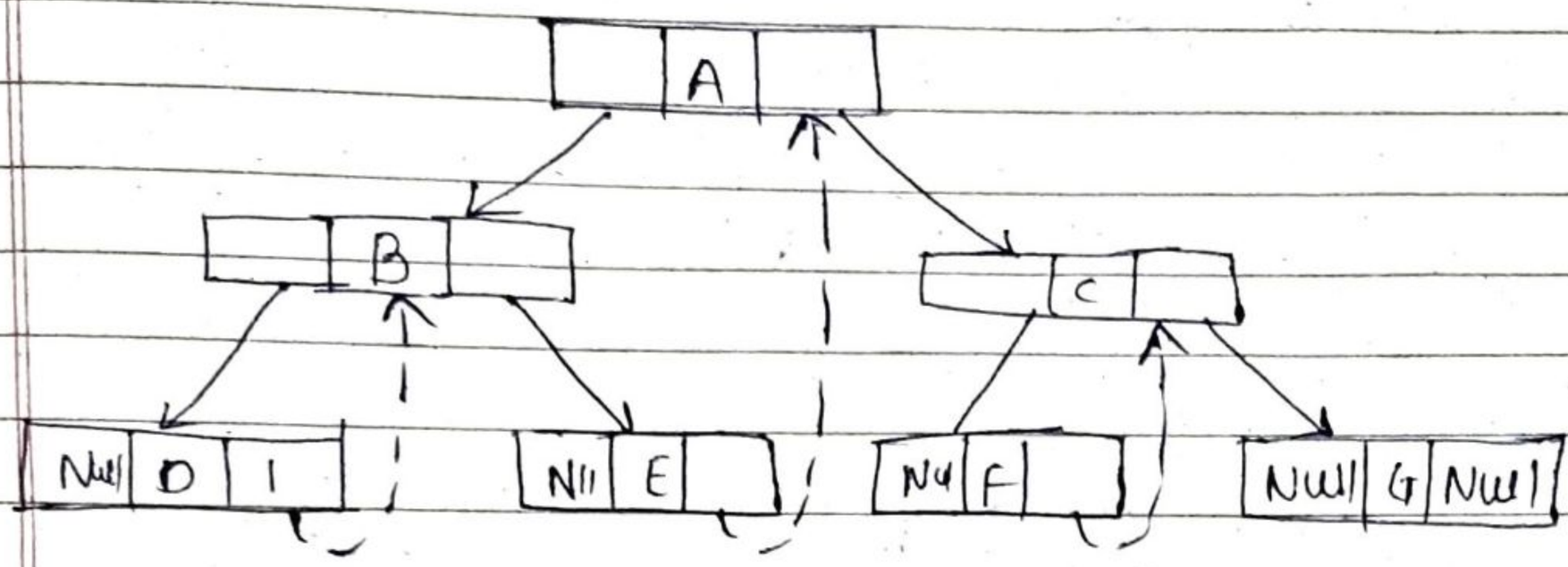



```
struct node
{
    char data;
    int L-Flag;
    int R-flag;
    struct node * left;
}; struct node * right;
```

PAGE No.
DATE / /

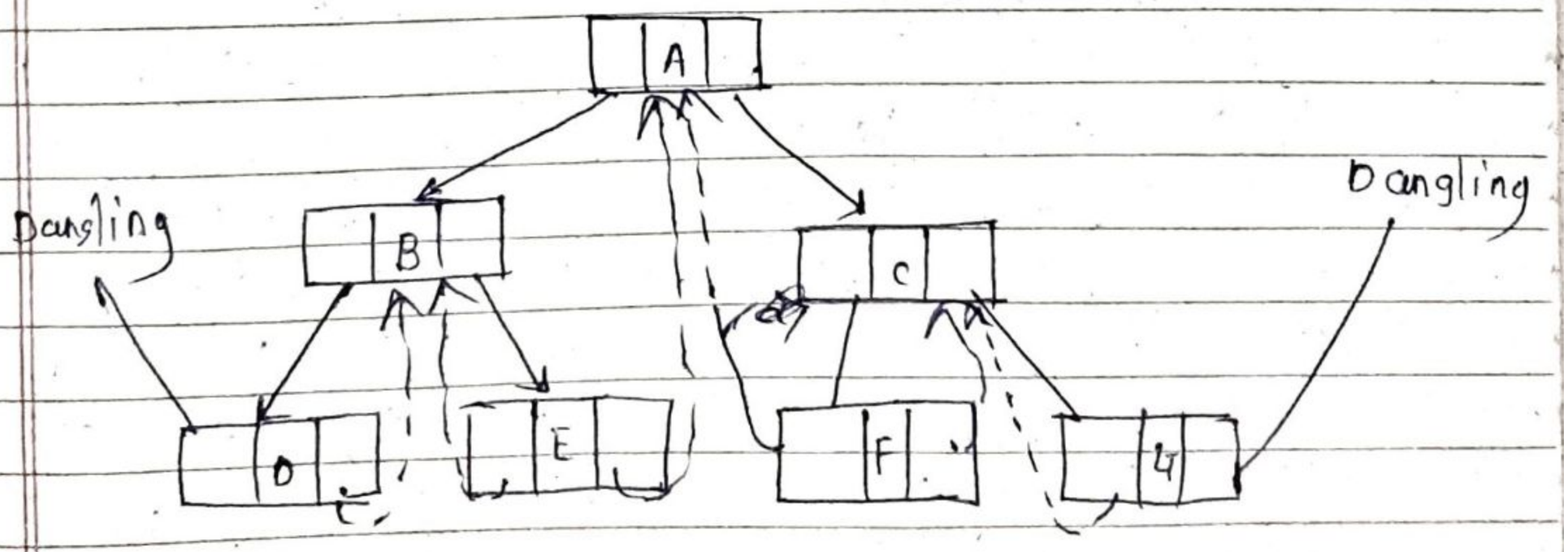
1) Single Threaded / one way Threading

- use only right threaded in this case.
- for implementation of thread use in-order successor of tree.



2) Double threaded / two way threading

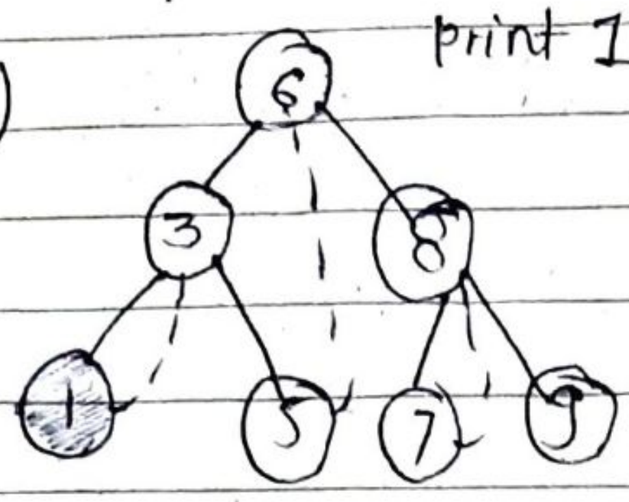
- Left pointer of first node & right pointer of last node will contain null value.
- for implementation of thread use in-order successor & in-order predcessor of tree.



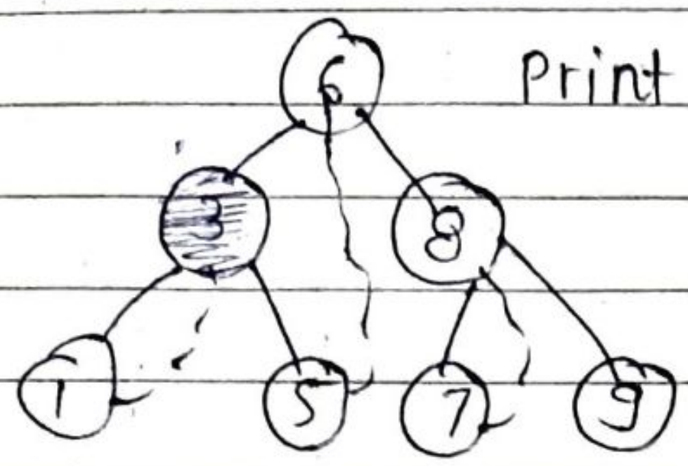
PAGE No
DATE / /

Traversal of Threaded Binary Tree (Inorder)

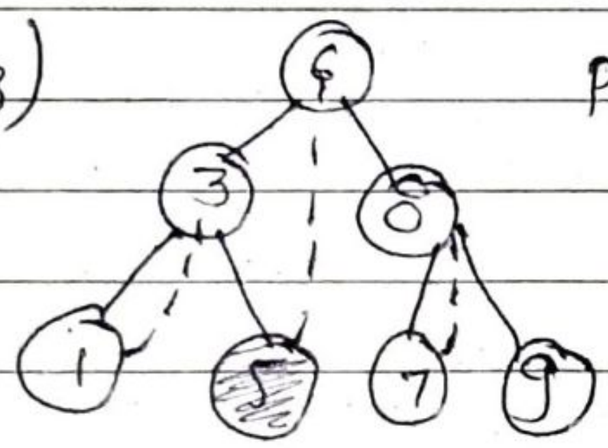
1) print 1



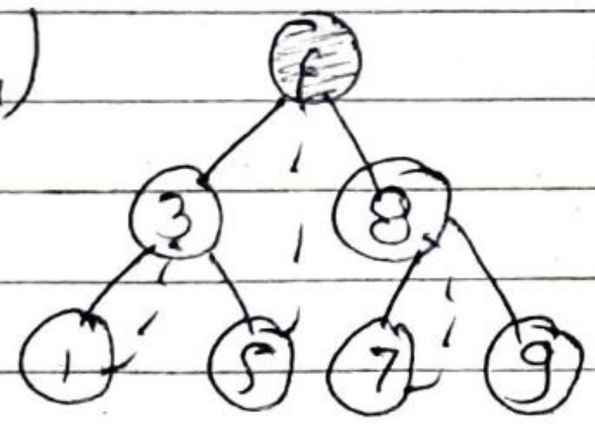
2) Print 3



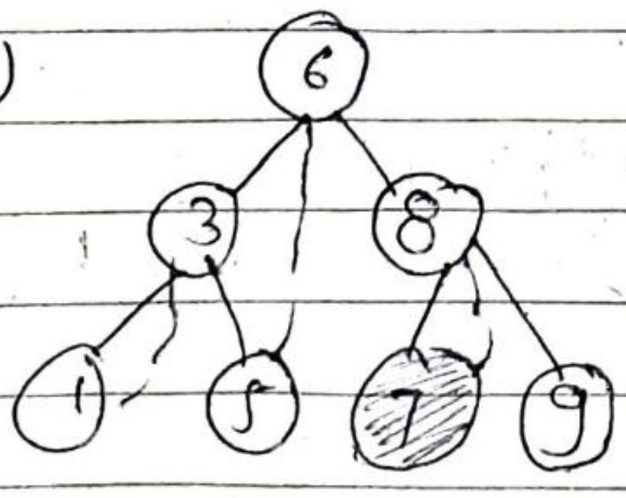
3) print 5



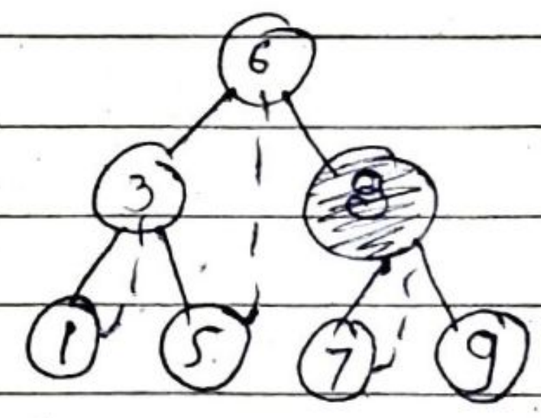
4) print 6



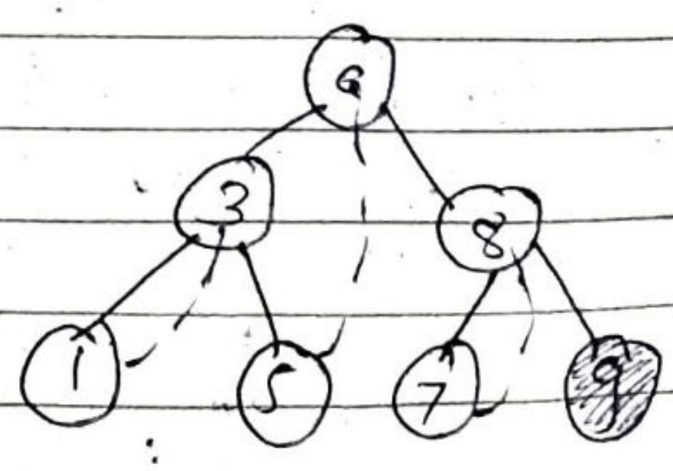
5)



6) print 8



7) print 9



O/P :

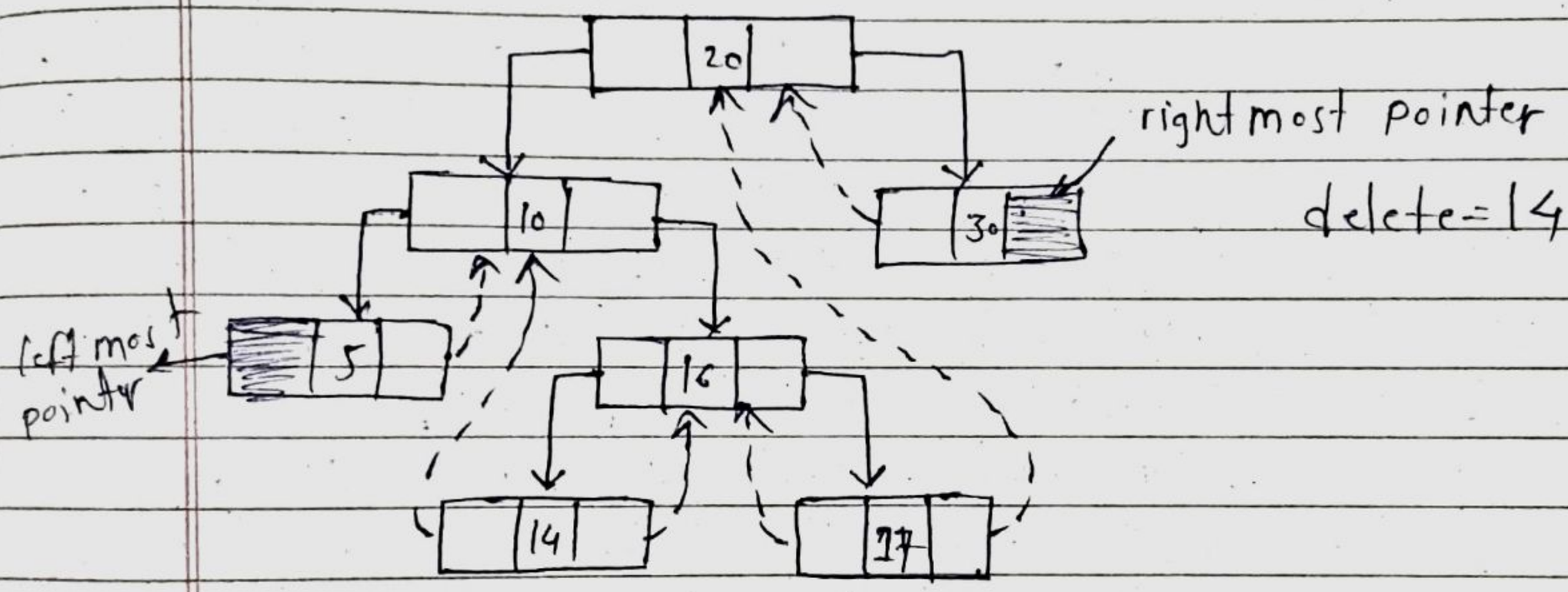
1
3
5
6
7
8
9

PAGE NO.
DATE / /

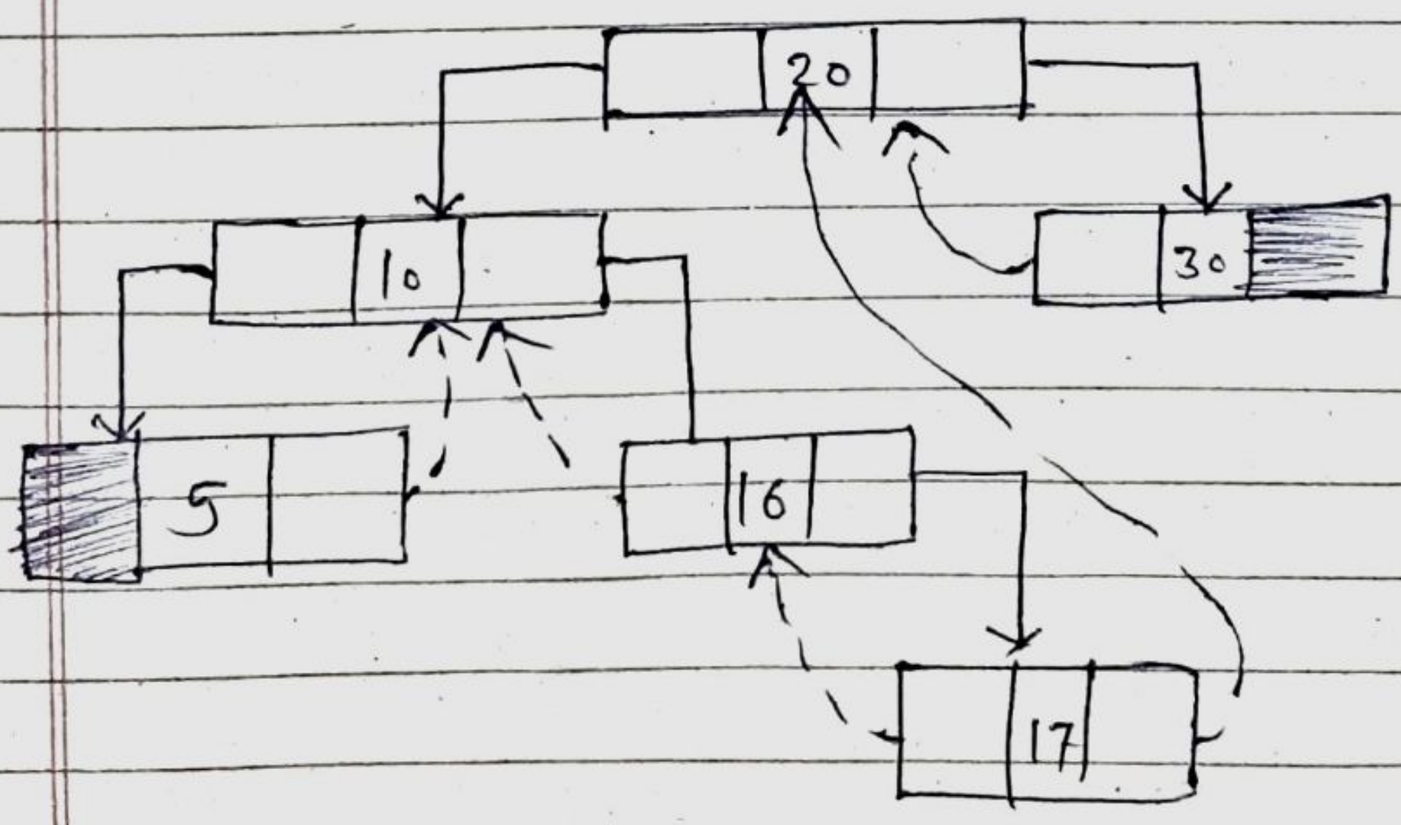
* Deletion in TBT

Case 1 : A leaf node

- Delete a node directly
- Thread parent of deleted node to left/right pointer of its inorder predecessor or inorder succesor.



Inorder : 5 10 14 16 17 20 30



Inorder : 5 10 16 17 20 30

PAGE No.
DATE / /

Case 2 : Node is to be deleted which has left child.

- Delet the node
- After deleting it, point ~~its~~ left child to ~~its~~ the ~~inorder predecessor or the~~ parent of its deleted parent, and there left pointer to respective inorder predecessor.

p→right = s

case 3 : Node is to be deleted which has right child.

- Delet the node
- After deleting it, point its right child to the parent of its deleted parent. And there right pointer to respective inorder Successor.

s→left = p

PAGE No.
DATE / /

* AVL Tree [Balanced Binary ST ree]

- Only BST can converted in AVL.

- Balanced Factor = (height of LST - height of RST)

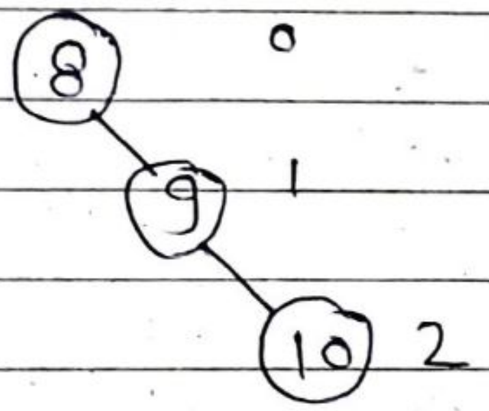
+1, -1, 0 → then only it is BT.

↓

if other than this value comes then will convert it into AVL.

Make BST

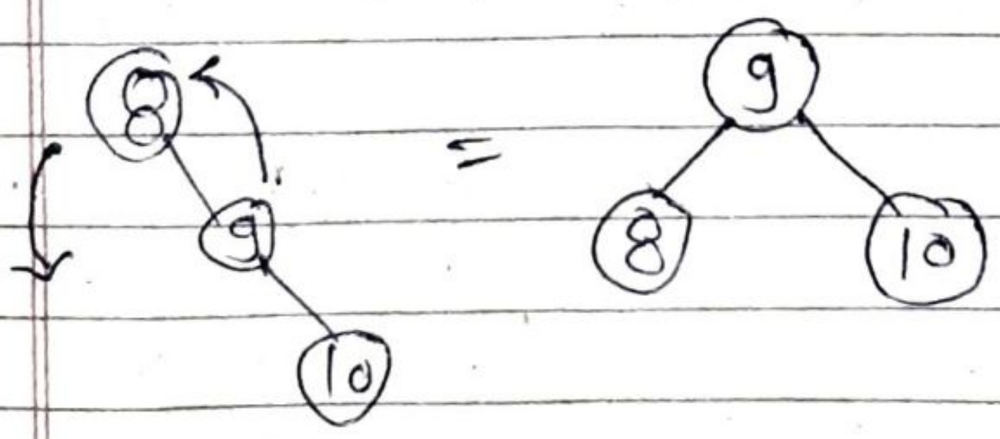
1) Ex: 8, 9, 10 RR



8 = 0 - 2 = -2
9 = 0 - 1 = -1
10 = 0 - 0 = 0

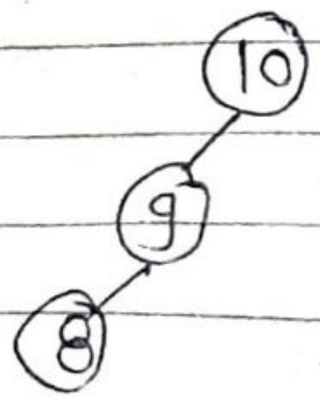
This is RR unbalanced
To make it balance Rotate it anticlockwise from 8-9

3 - 2 =
3 - 1 = 2



1 - 1 = 0
0 - 0 = 0

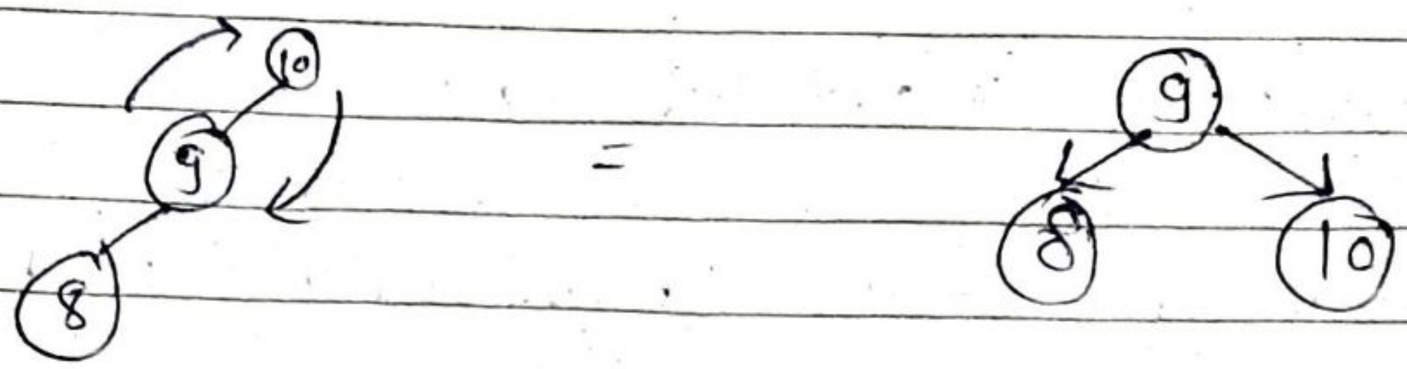
2) Ex: 10, 9, 8 LL



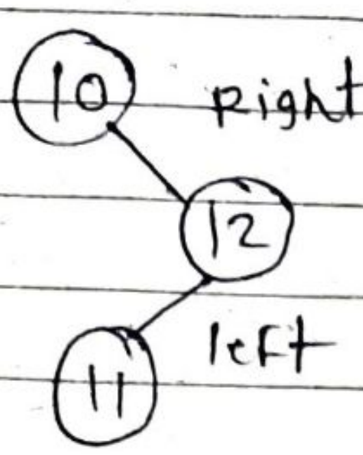
10 = 2 - 0 = 2
9 = 1 - 0 = 1
8 = 0 - 0 = 0

PAGE No.
DATE / /

Rotate it clockwise

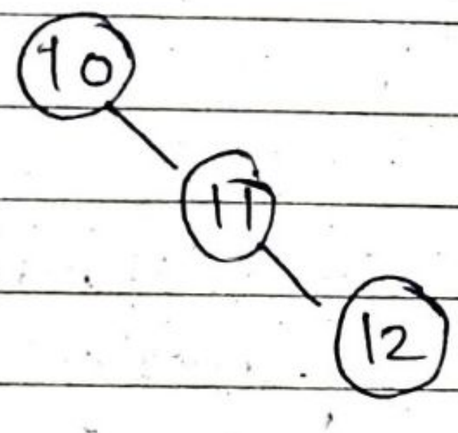


3) Ex: 10, 12, 11 RL (two Rotations)

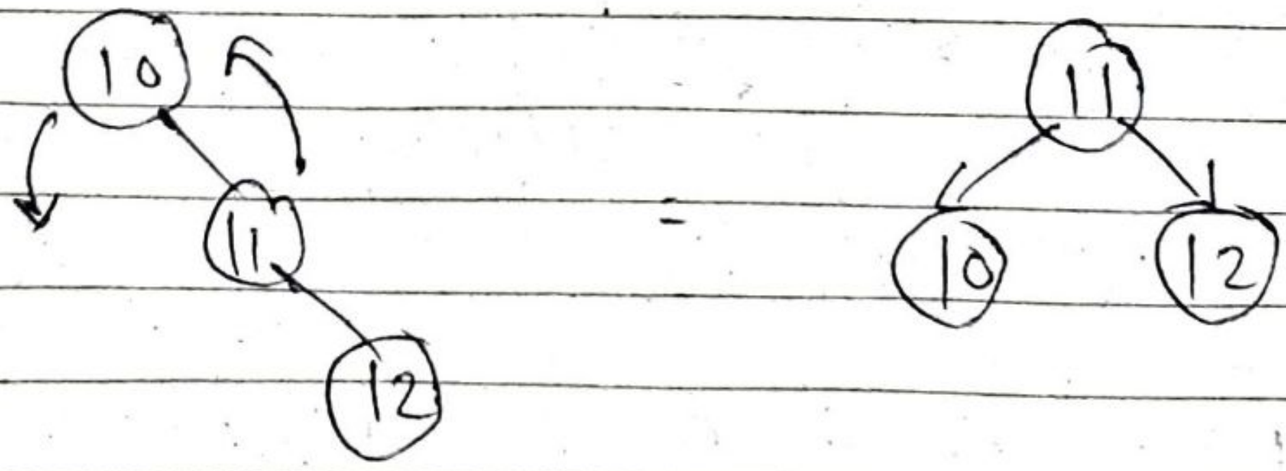


10 = 0 - 2 = -2
12 = 0 - 1 = -1
11 = 0 - 0 = 0

- 1st rotation → RL → RR

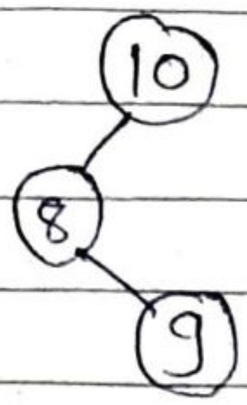


- 2nd rotation → RR anticlowise

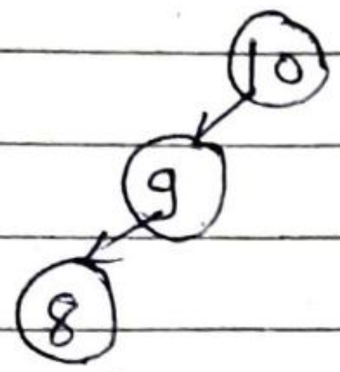


PAGE No
DATE / /

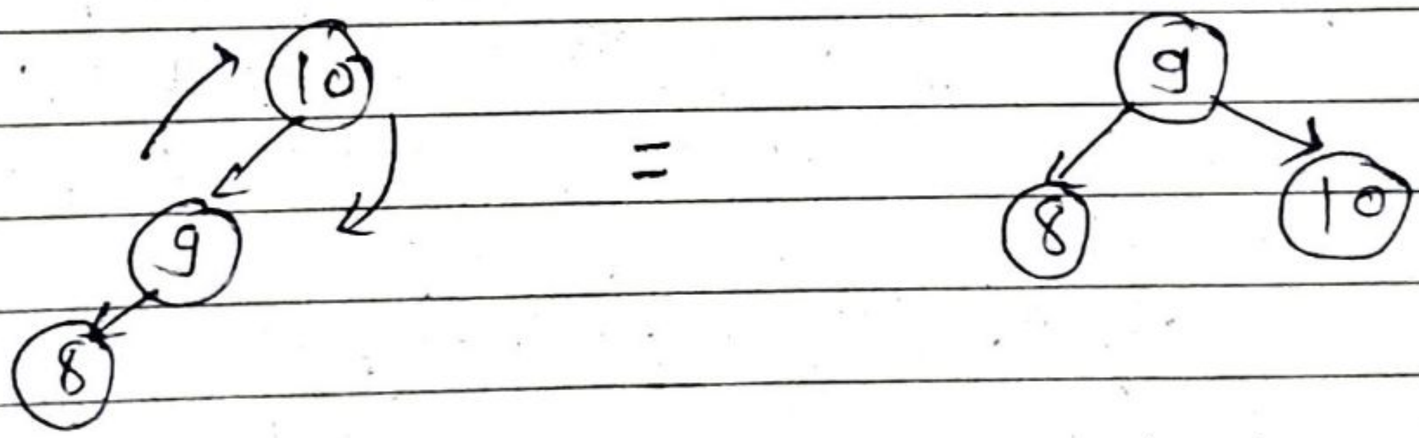
4) EX : 10, 8, 9 LR



- 1st rotation LL



- 2nd rotation LL clockwise.



PAGE No.
DATE / /

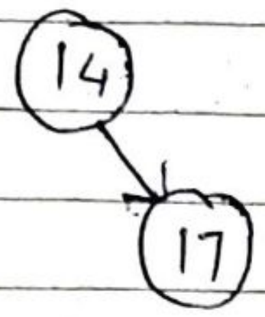
a) Construct AVL tree by inserting the following data.

14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

After inserting each node check the balanced factor.

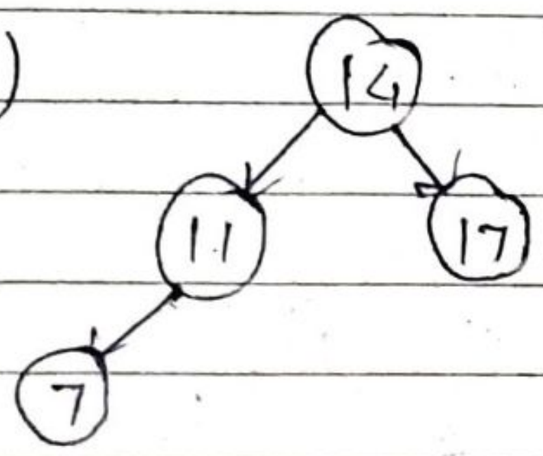
1) 

BF = 0

2) 

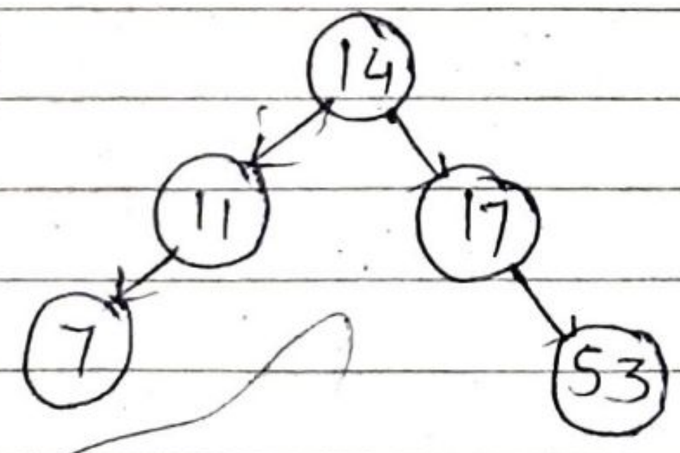
~~BF = 0 - 0 = 0~~

14 = 1
17 = 1

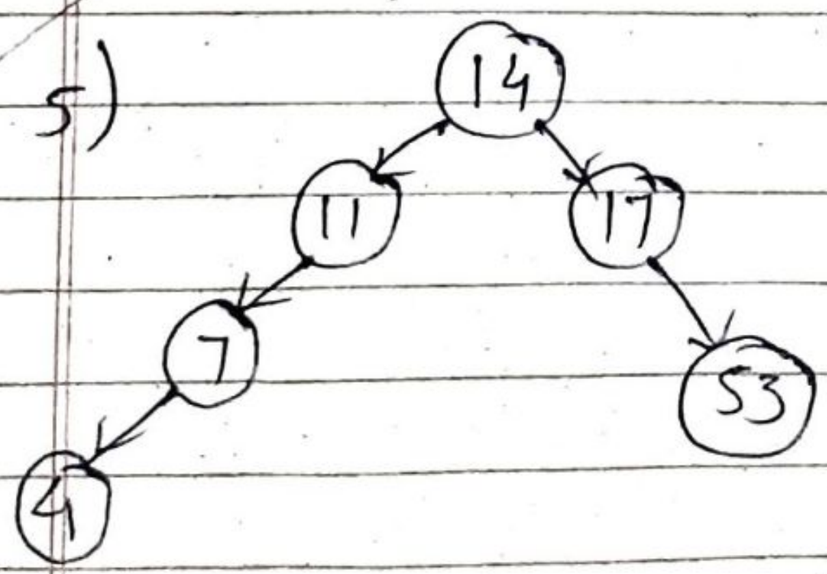
3) 

~~BF 7 = 0 - 0 = 0 1~~
~~11 = 1 - 1 = 0~~
~~14 = 2 - 1 = 1~~

14 = 1 11 = 1
17 = 0 7 = 0

4) 

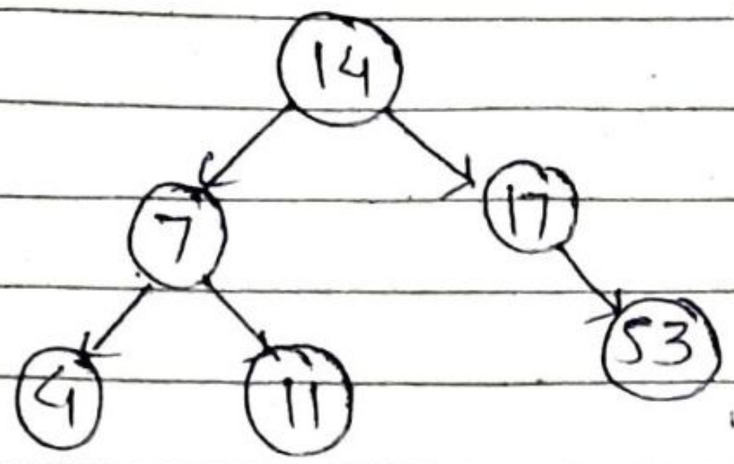
7 = 0 17 = 1
11 = 1 53 = 0
14 = 0

5) 

14 = 1
17 = 2
53 = 0
11 = 2
7 = 1
4 = 0

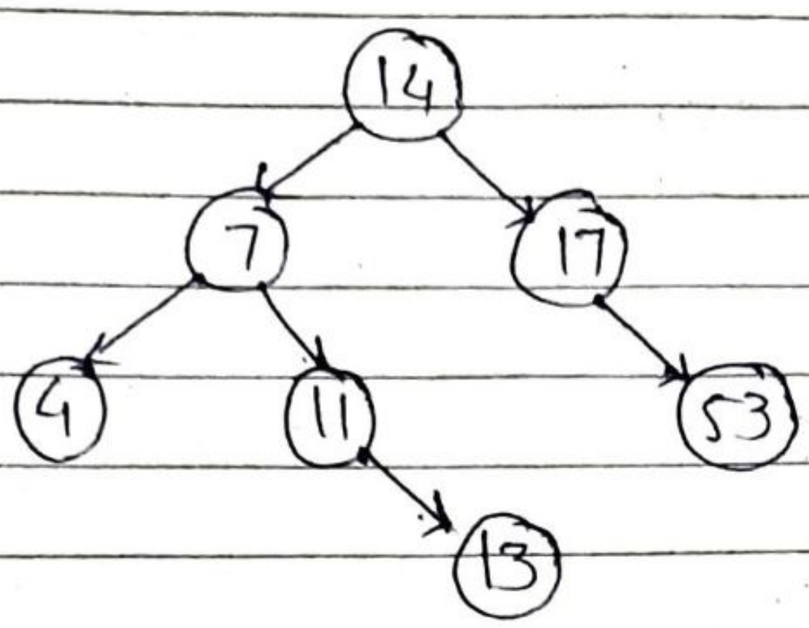
PAGE No.
DATE / /

Here because of LL we will rotate this



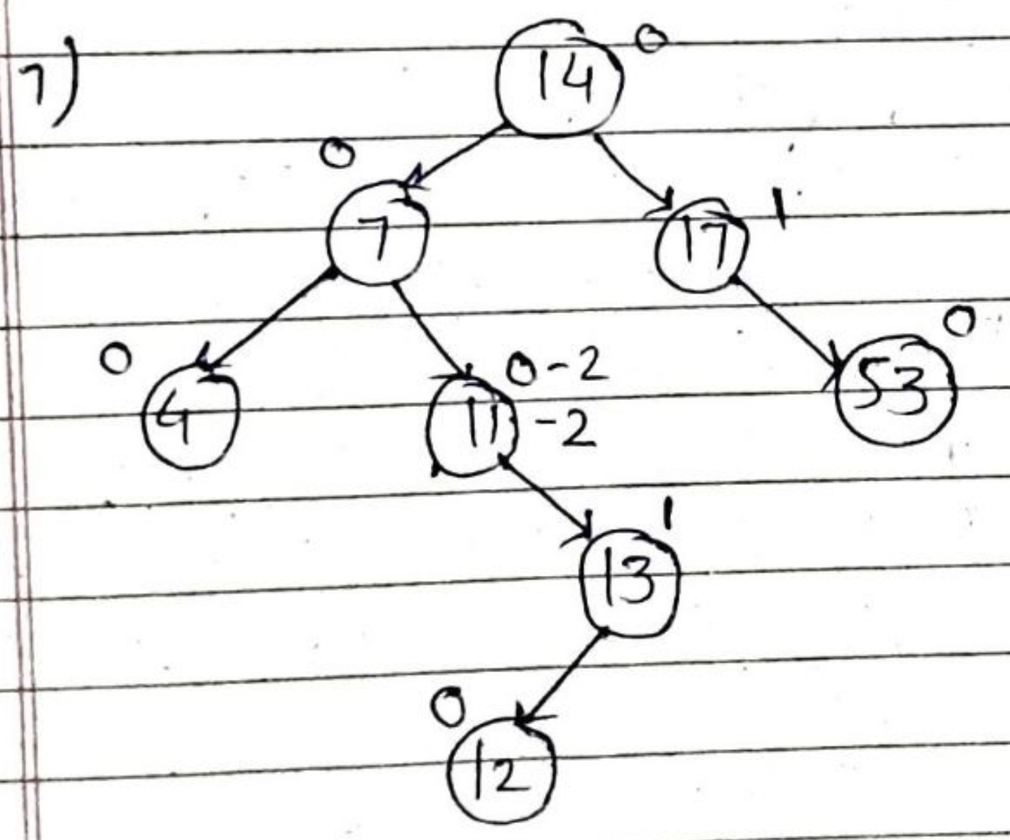
14 = 2-2 = 0
7 = 1-1 = 0
4 = 0 , 11 = 0
17 = 2-1 = -1
53 = 0

6)



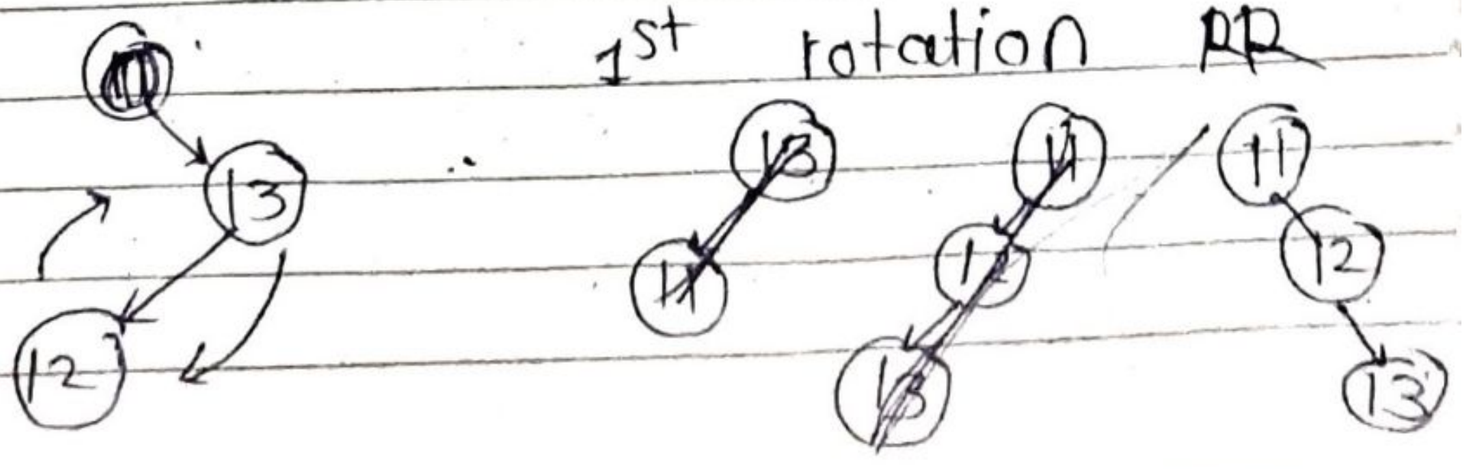
14 = 2-2 = 0
7 = 1-1 = 0
4 = 0
11 = 0-1 = -1
17 = 2-1 = -1
53 = 0

7)

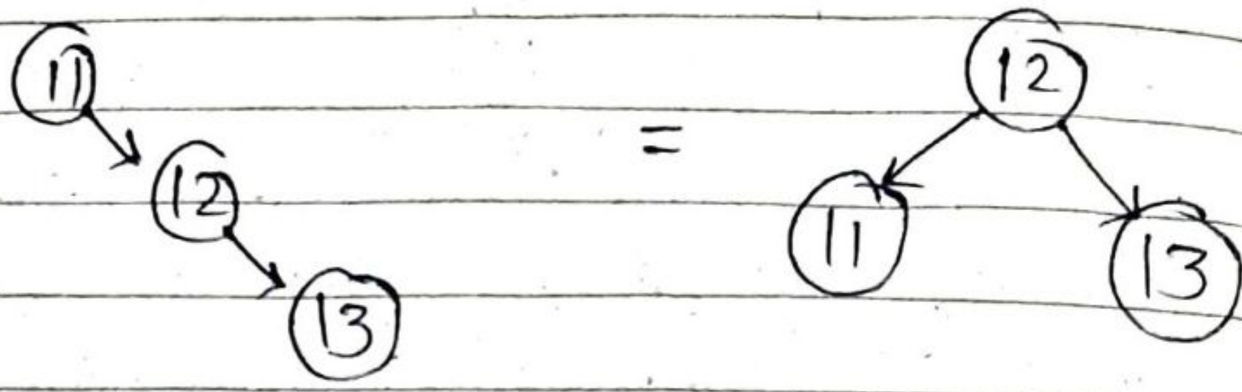


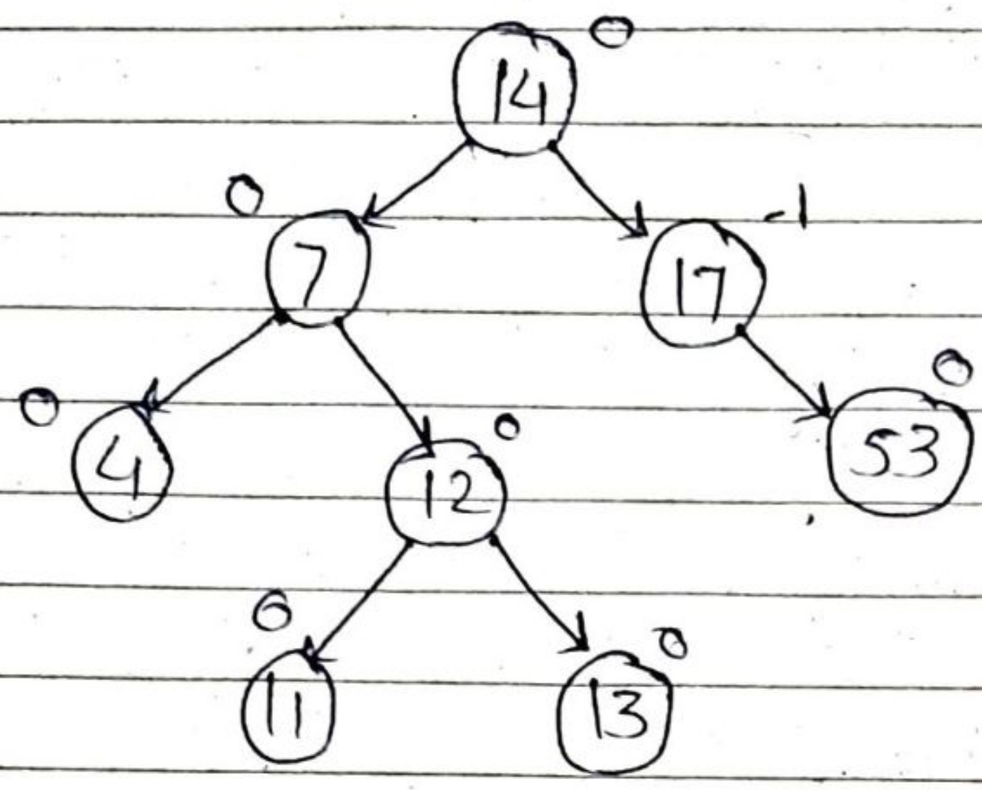
Here the unbalance is ~~LR~~ RL

1st rotation ~~RR~~

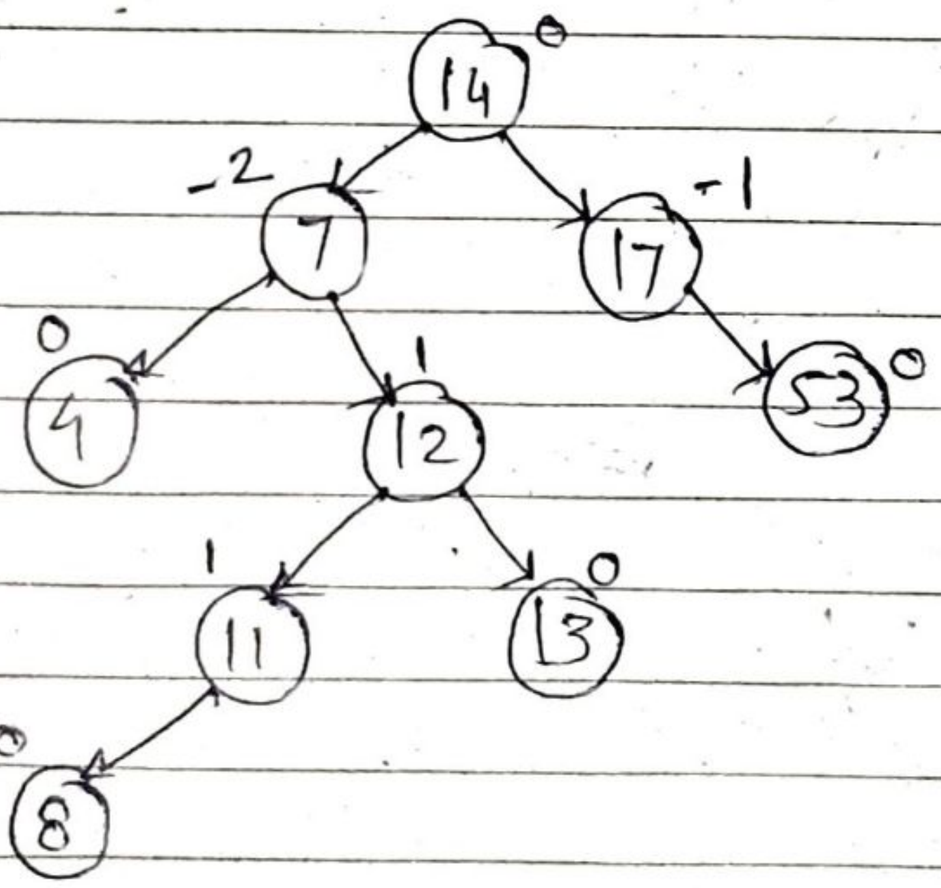


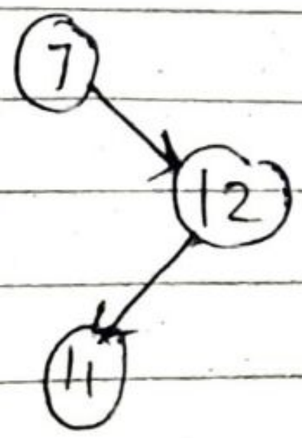
2nd rotatation





8)

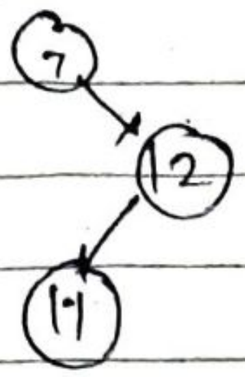




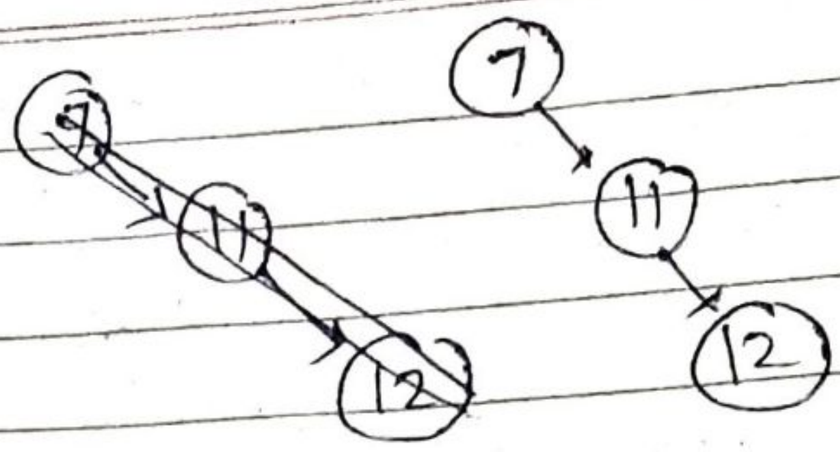
This is RRL

PAGE No.
DATE / /

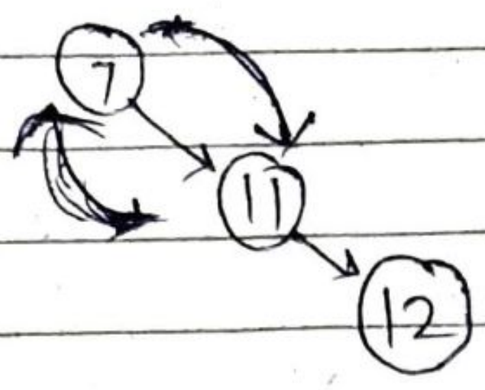
RR

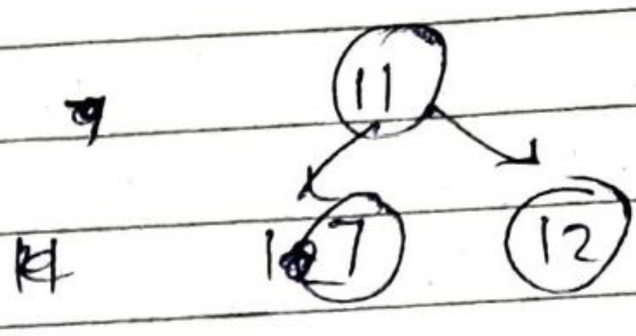


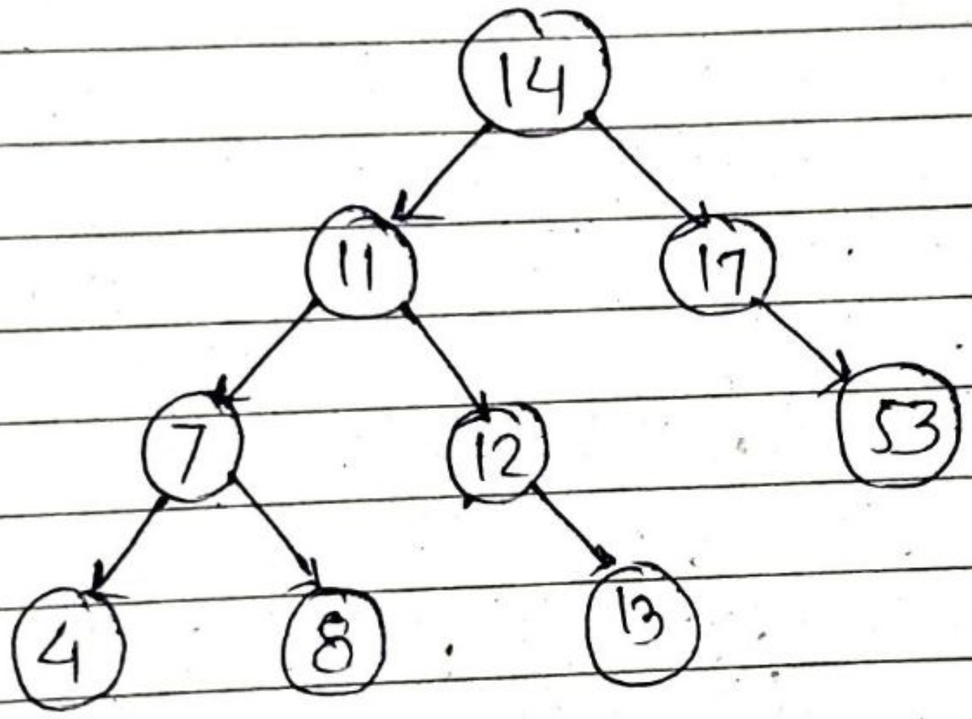
RR



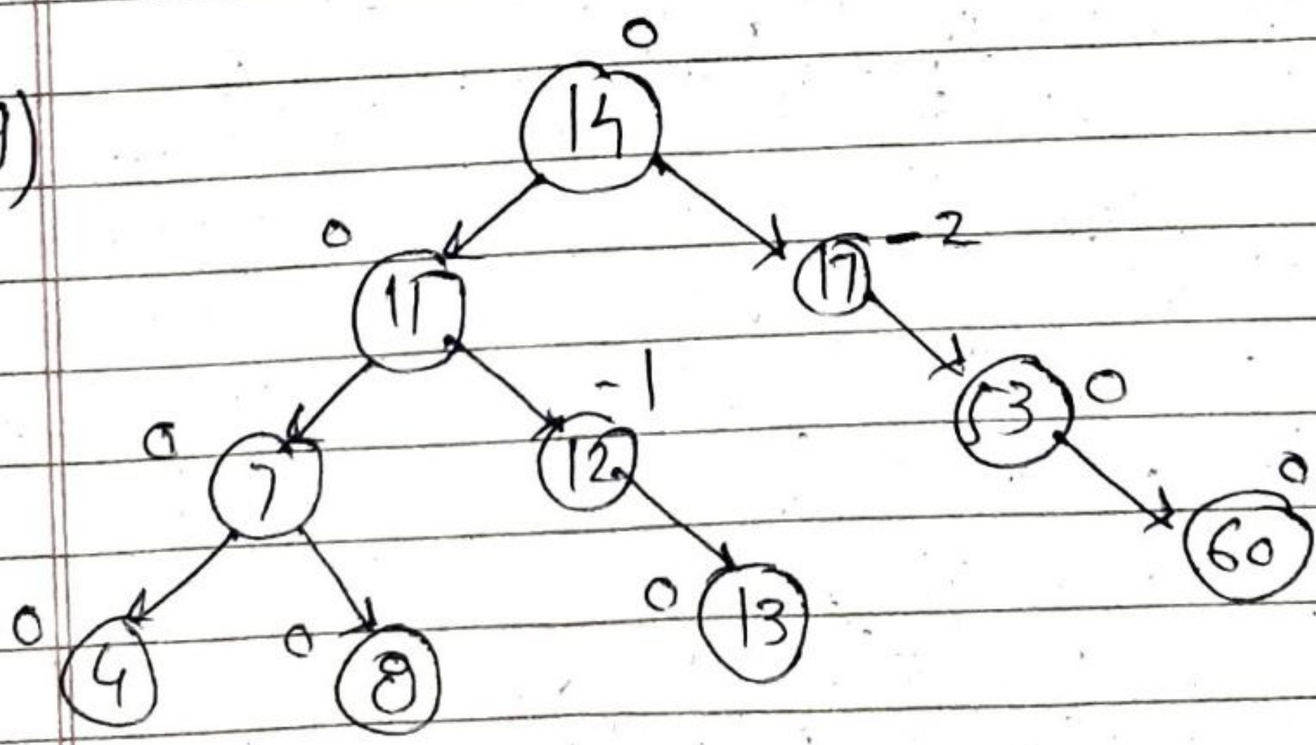
RR



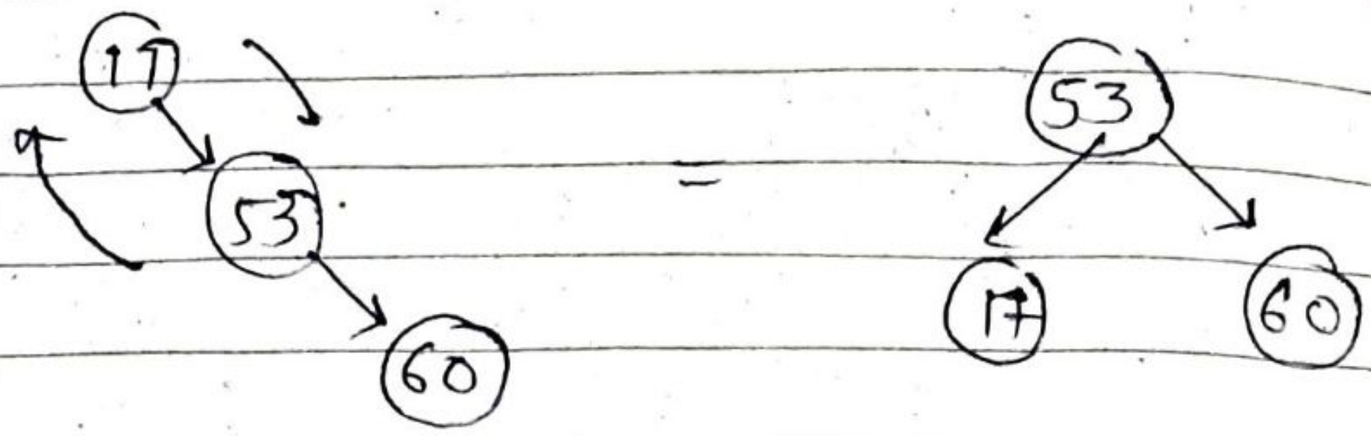


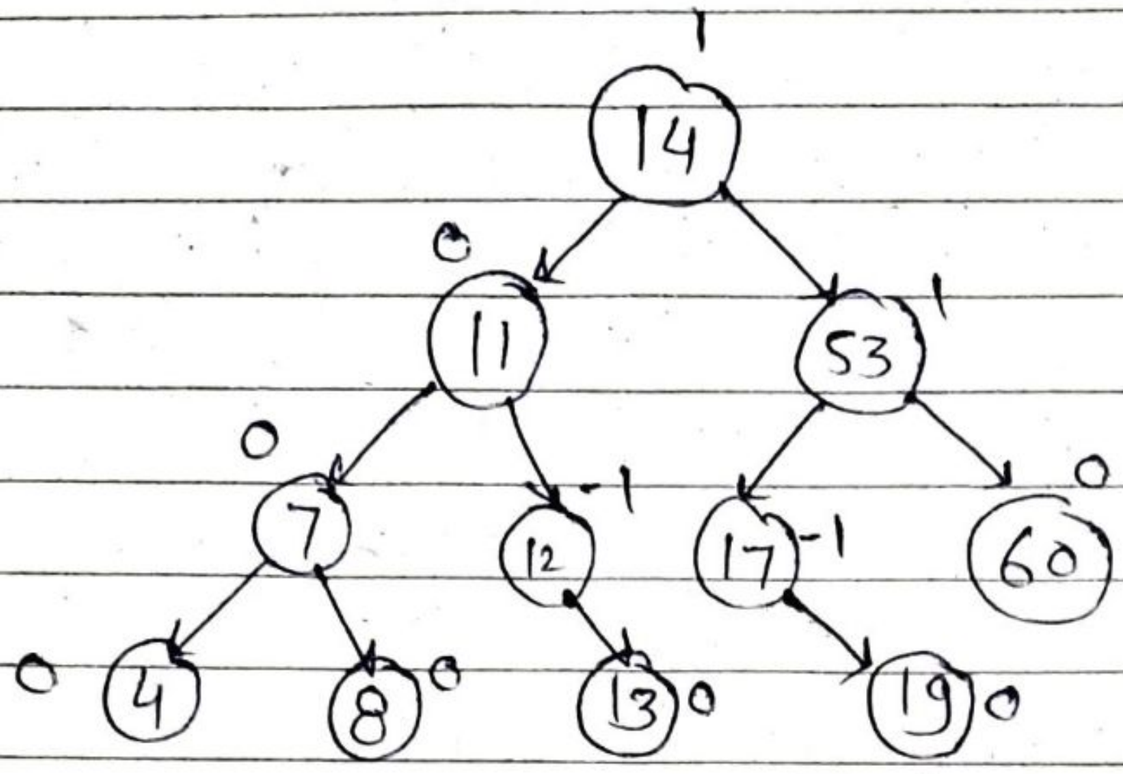


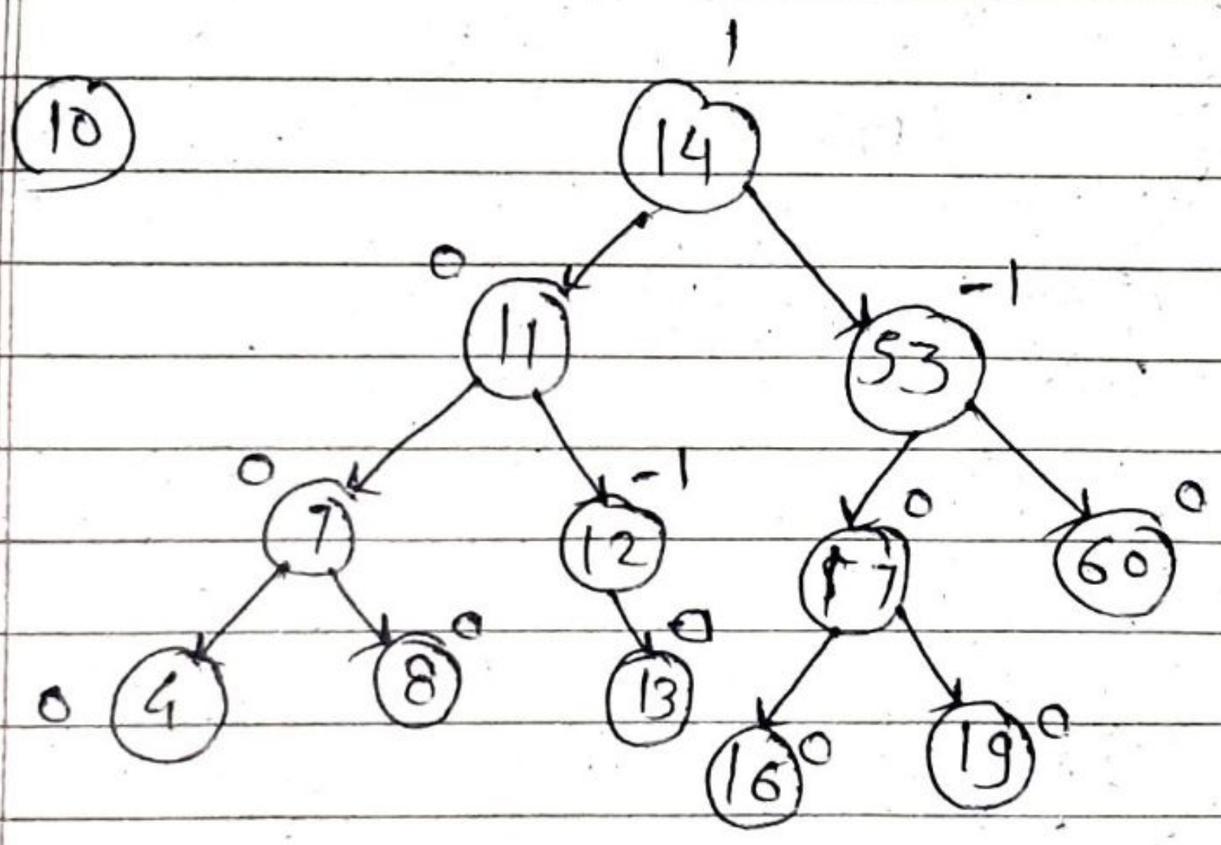
9)

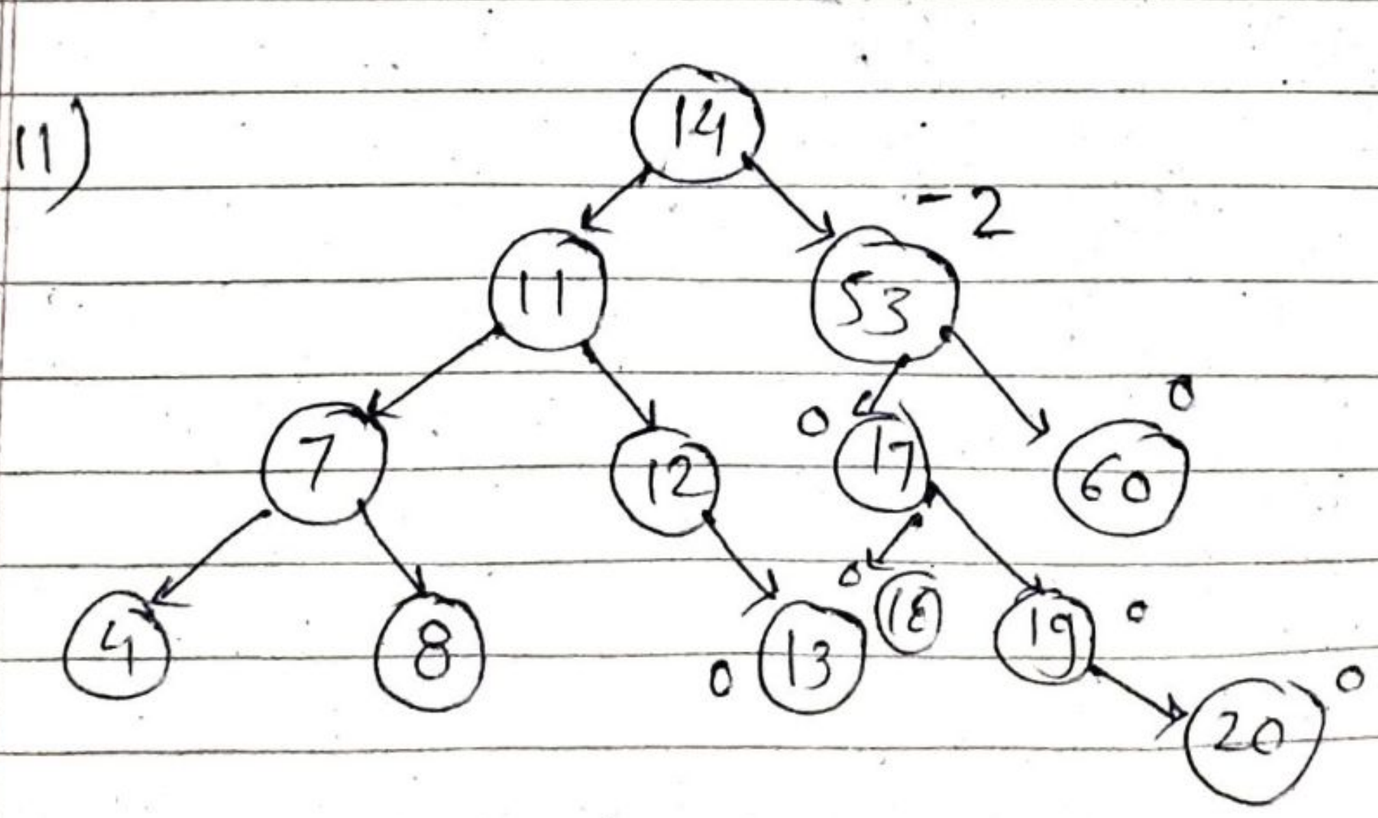


RR

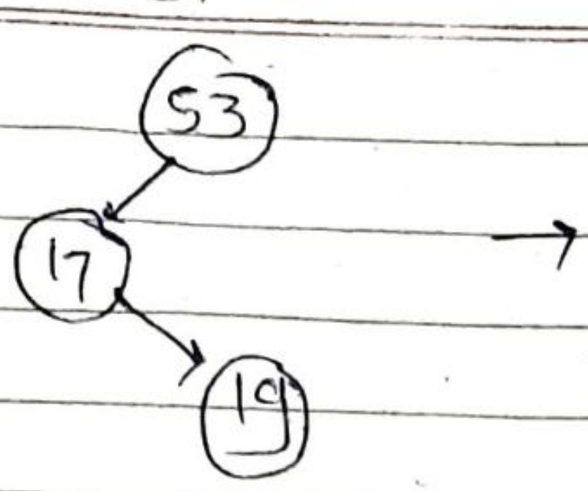




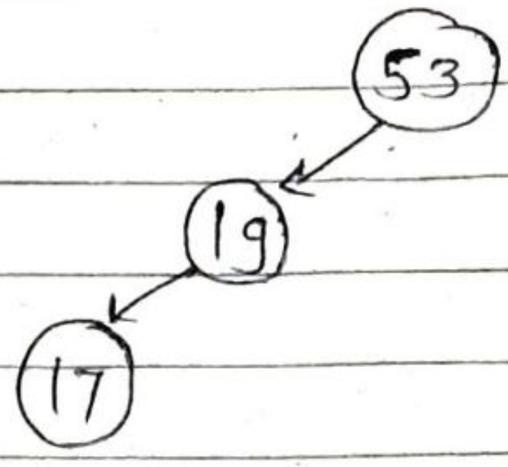




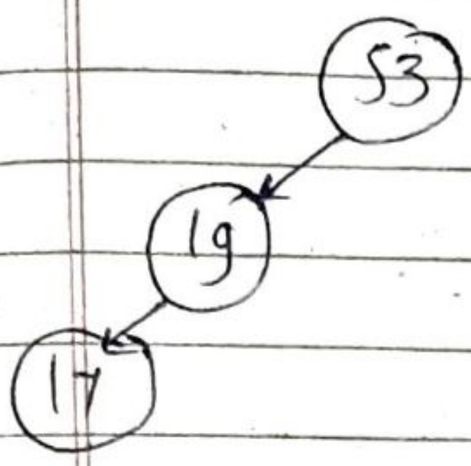
LR



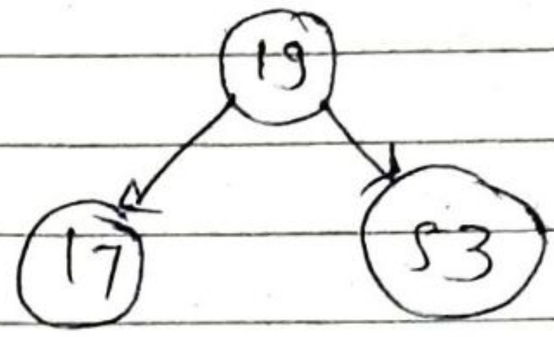
→

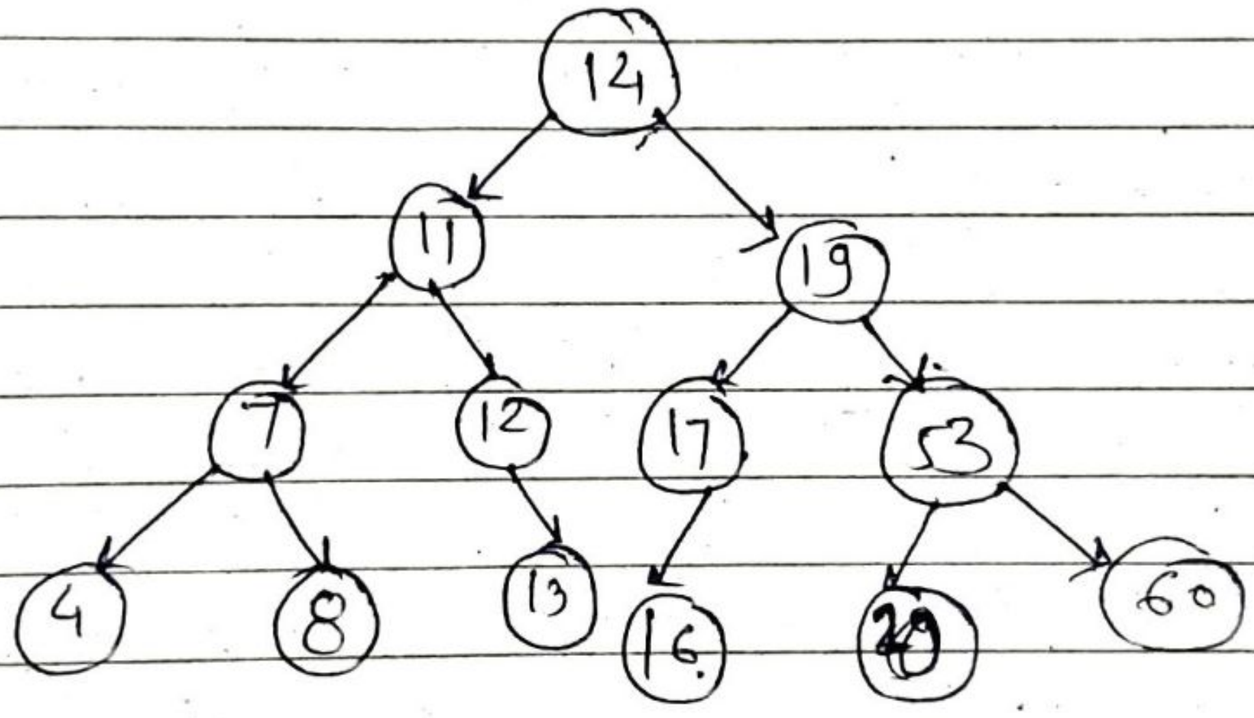


LL



→





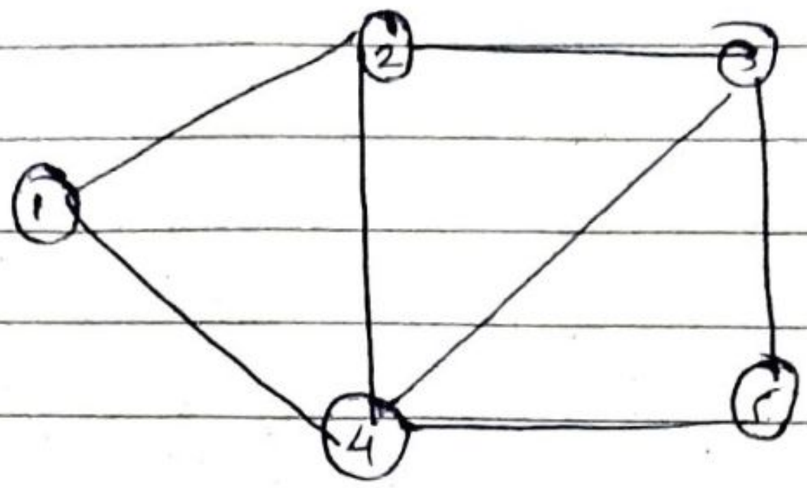
Final AVL.

GRAPH

PAGE No.
DATE / /

* Adjacency Matrix

n x n where n is no. of vertex

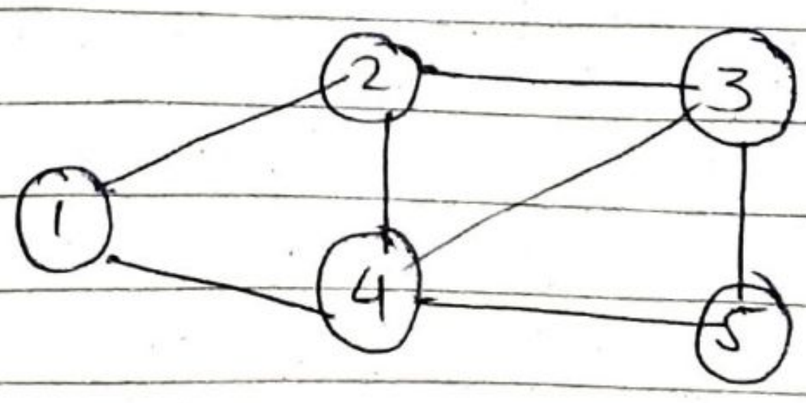


	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

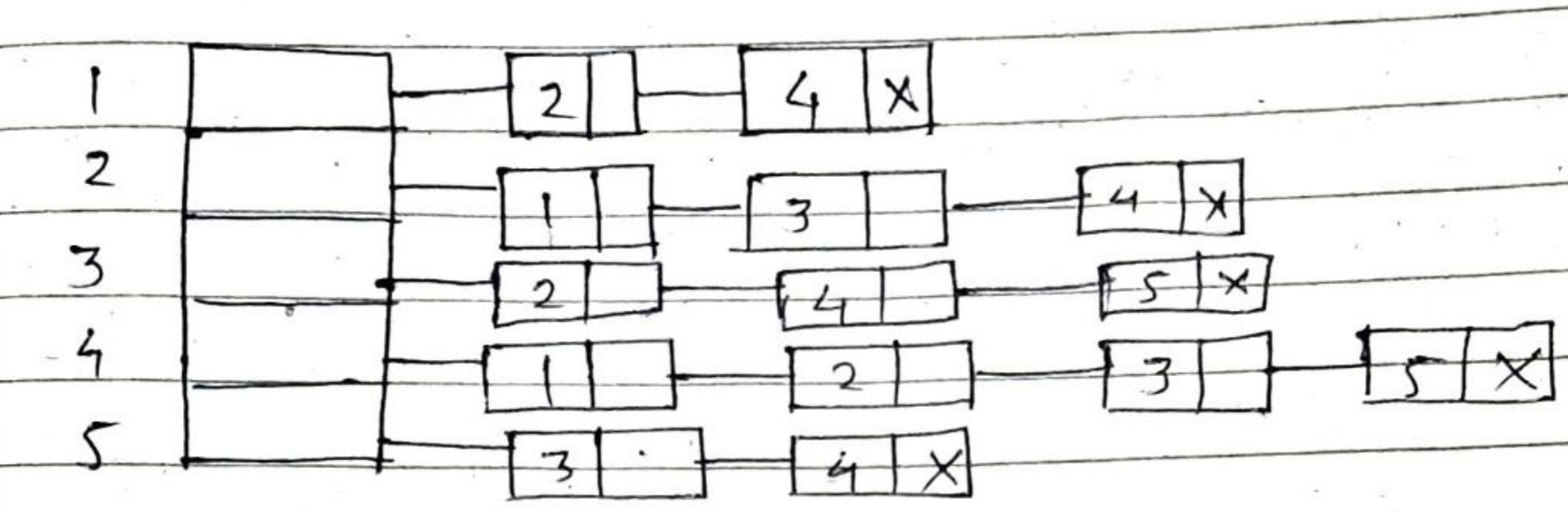
Space Com
$O(n^2)$

It is a matrix A[n][n] where n is no. of vertices, a[i][j] = 1 if i & j are adjancent otherwise a[i][j] = 0.

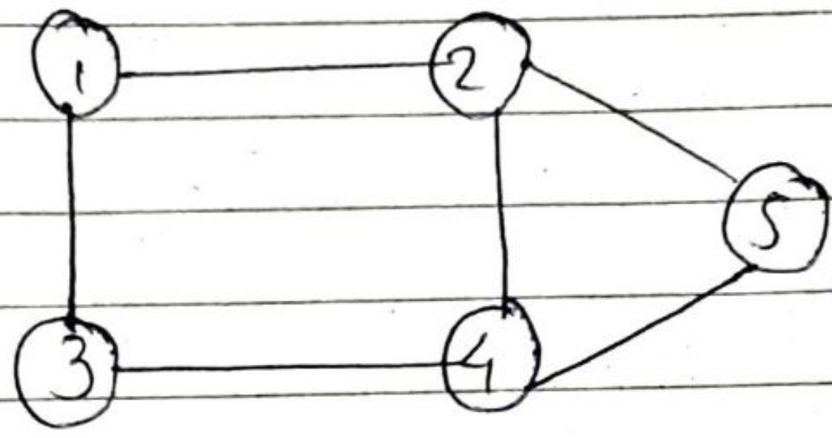
* Adjacency List



Space Com ☺ (n+2e)



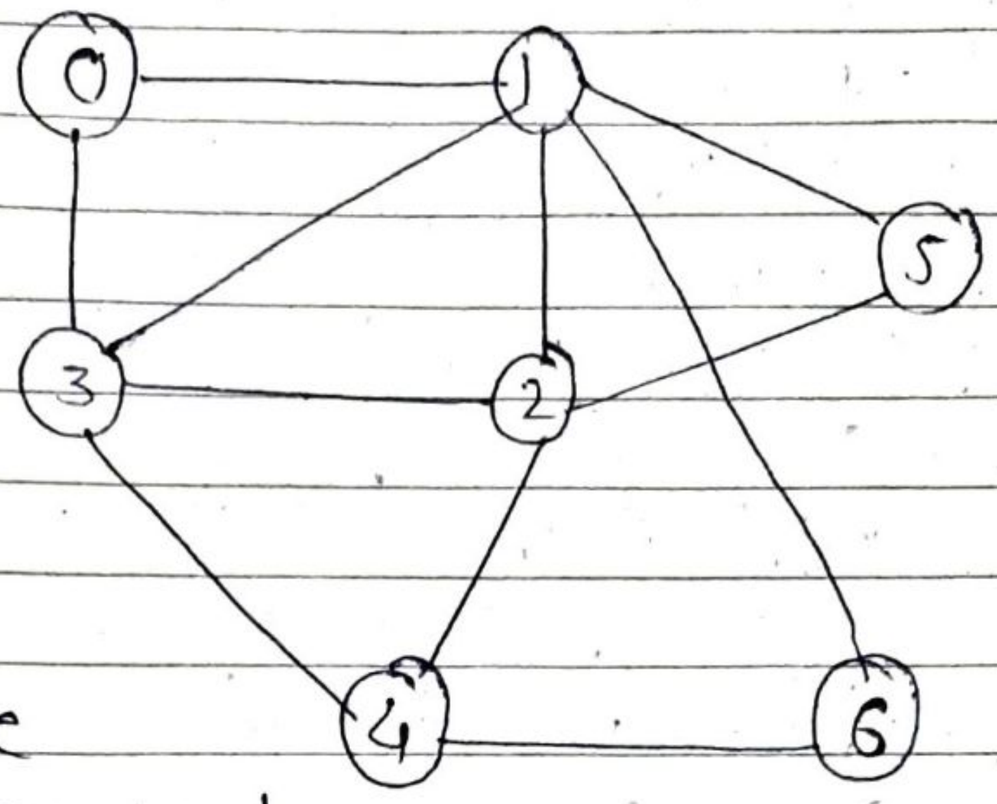
* Adjancy multi-List (Edge based)



PAGE No.
DATE / /

Graph traversal

1) DFS (Depth first search)
2) BFS (Breadth first search)



[BFS] queue

taking 0 as root.

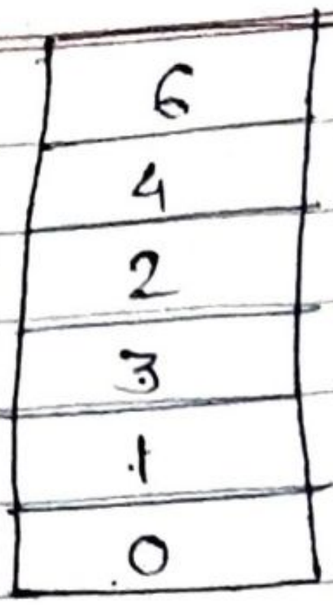
queue -	0	1	3	2	5	6	4

Result - 0 1 3 2 5 6 4

:-

1) Take the element insert into queue.
2) print this element.
3) insert its adjacent vertex of that element.
4) Again do same for next element, but only insert the unvisited adjacent Vertices.

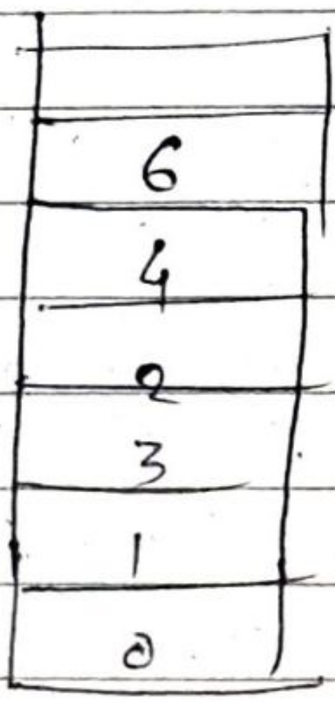
[DFS] ~~qu~~ stack



Result : 0, 1, 3, 2, 4, 6

stack

1) Take root, insert into the stak.
2) print it.
3) Take adjance vertex (any one only) of that vextex and insert into staek and print it.
(can't insert visited / repeated vertex).

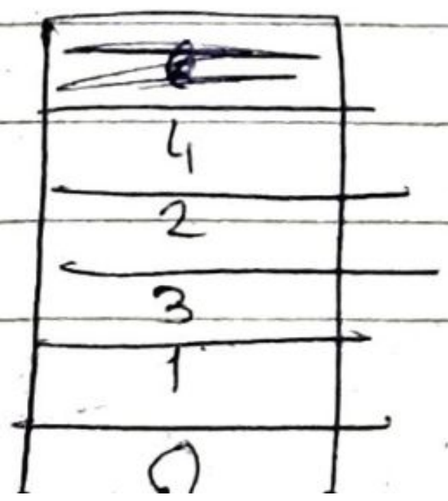


Now, 5 is remaing or the unvisited adjacent vertex of 6 is remaining.

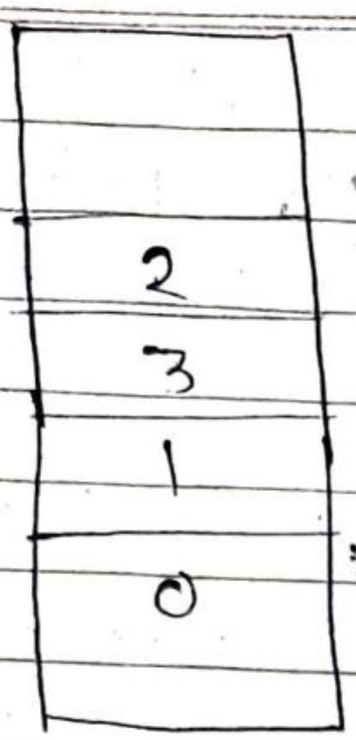
Hence, backtracking.

- So pop 6, and search for the unvisited A.vertices of the element that is on top of the stack.

- Do repeate until you find the unvisited adjacent vertex.

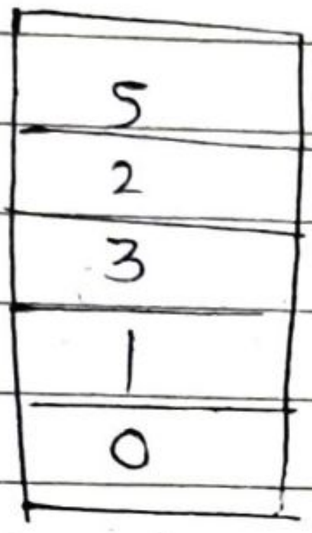


top element 4 has no unvisited av.

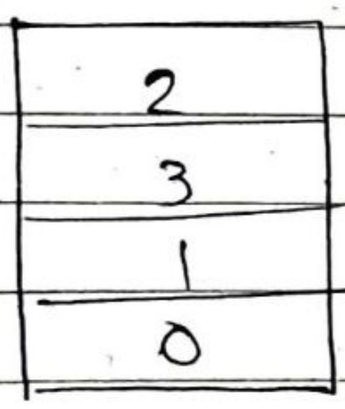


Hence, 2 has unvisited adjacent vertex thatis 5.

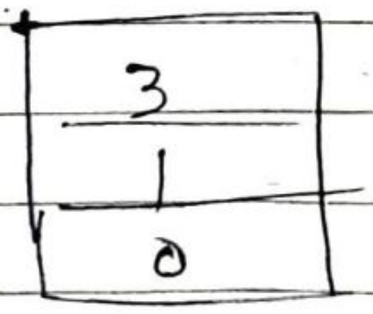
push 5 into stack.



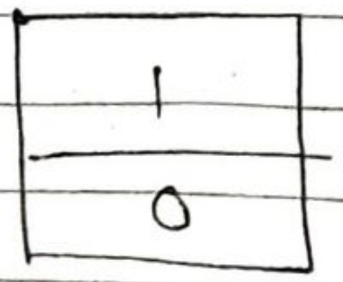
Now, check if 5 has any unvisited adj. vertex. if not then again backtrack.



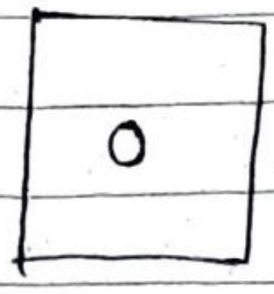
popd 5.



No unvisited AV.
poped 2



No unvisited AV.
poped 3

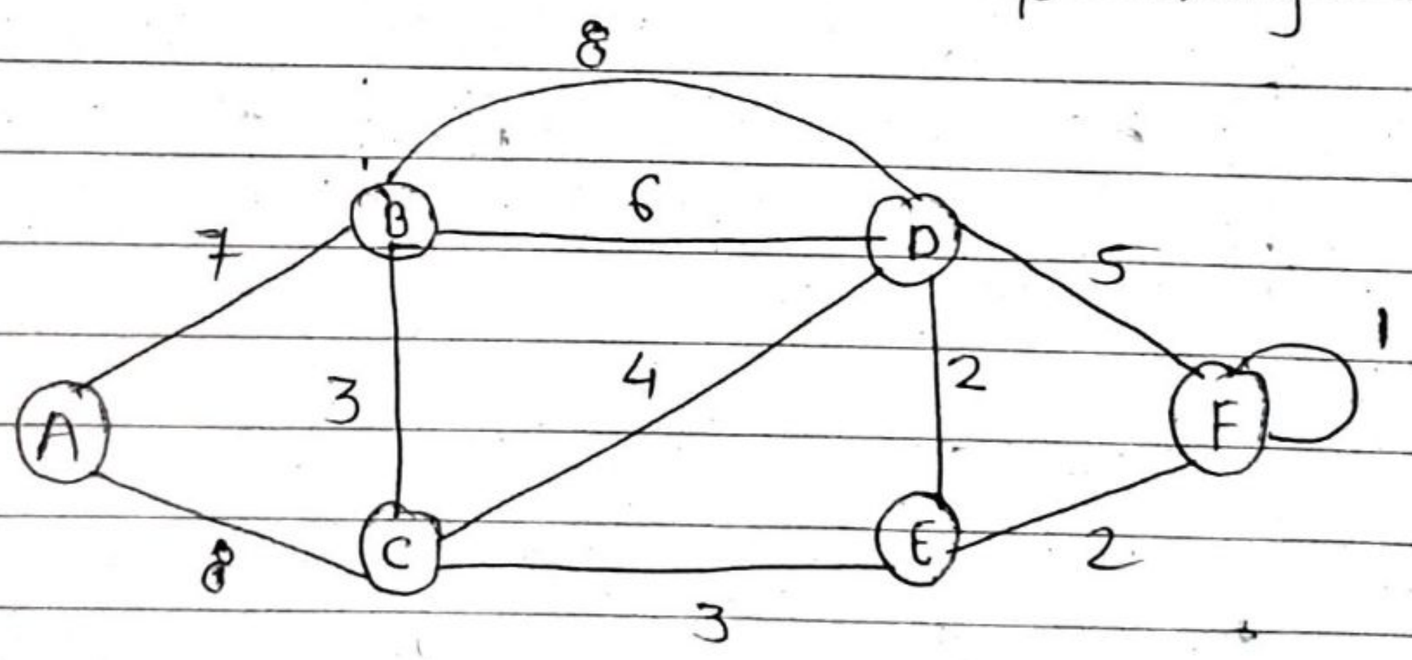


No unvisited AV.
poped 1

No unvisited AV
poped 0

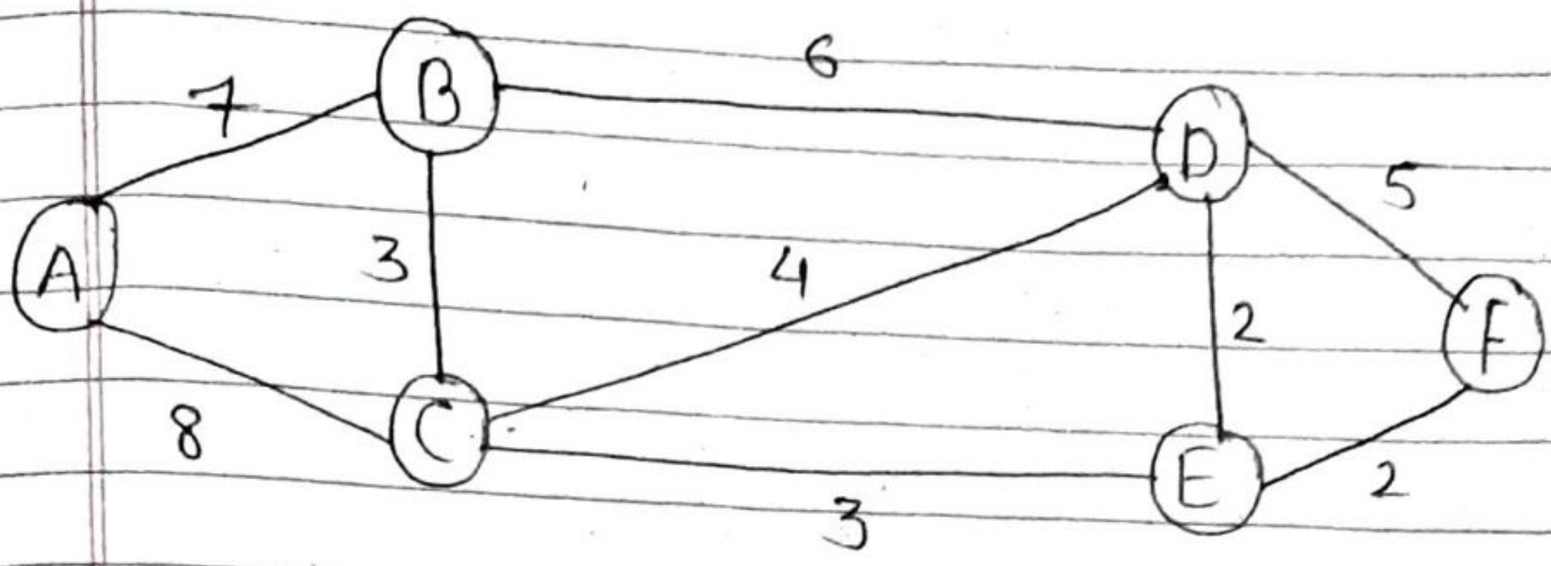
Now, stack is empty & we can conclude the end of DFS.

* prim's Algorithm. (for finding minimum spanning tree).



Step1: Now, Here loop eadge is there i.e F remove the edge.

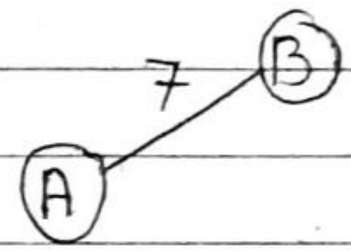
Step2: Remove the parallel edge from B-D which is haing maximum weight.



Step 3 : Choose any one node as root.

Step 4 : check outgoing edges from A, choose one edge that is having minimum weight.

Step NO	Edge	vertex Set
initial	—	{A}
1	(A, B)	{A, B}

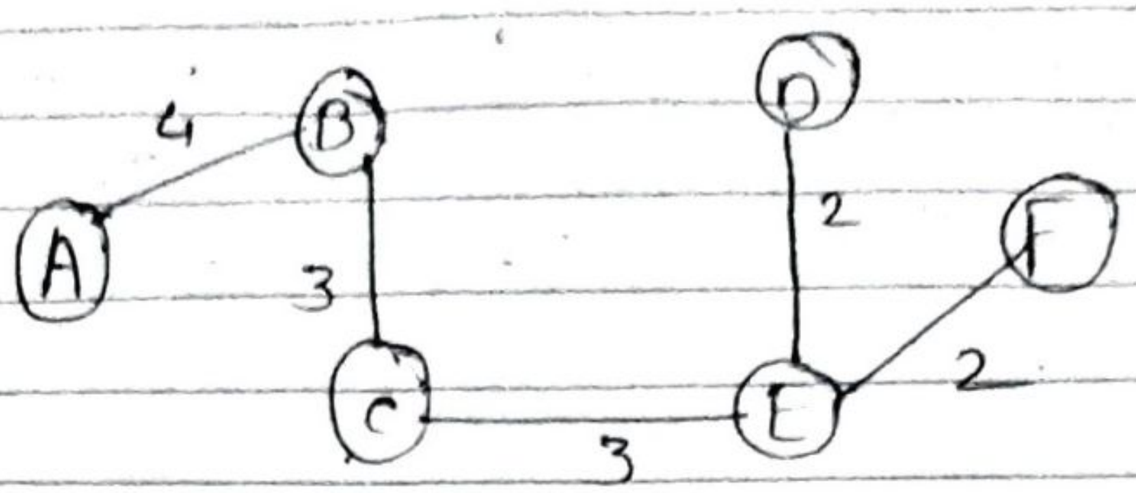


Step 5 : Now, check the outgoing edges from both A & B & choose the ededge with minimum weight.

initial	—	{A}
1	(A, B)	{A, B}
2	(B, C)	{A, B, C}
3	(C, E)	{A, B, C, E}
4	(E, F)	{A, B, C, E, F}
5	(F, D)	{A, B, C, E, F, D}

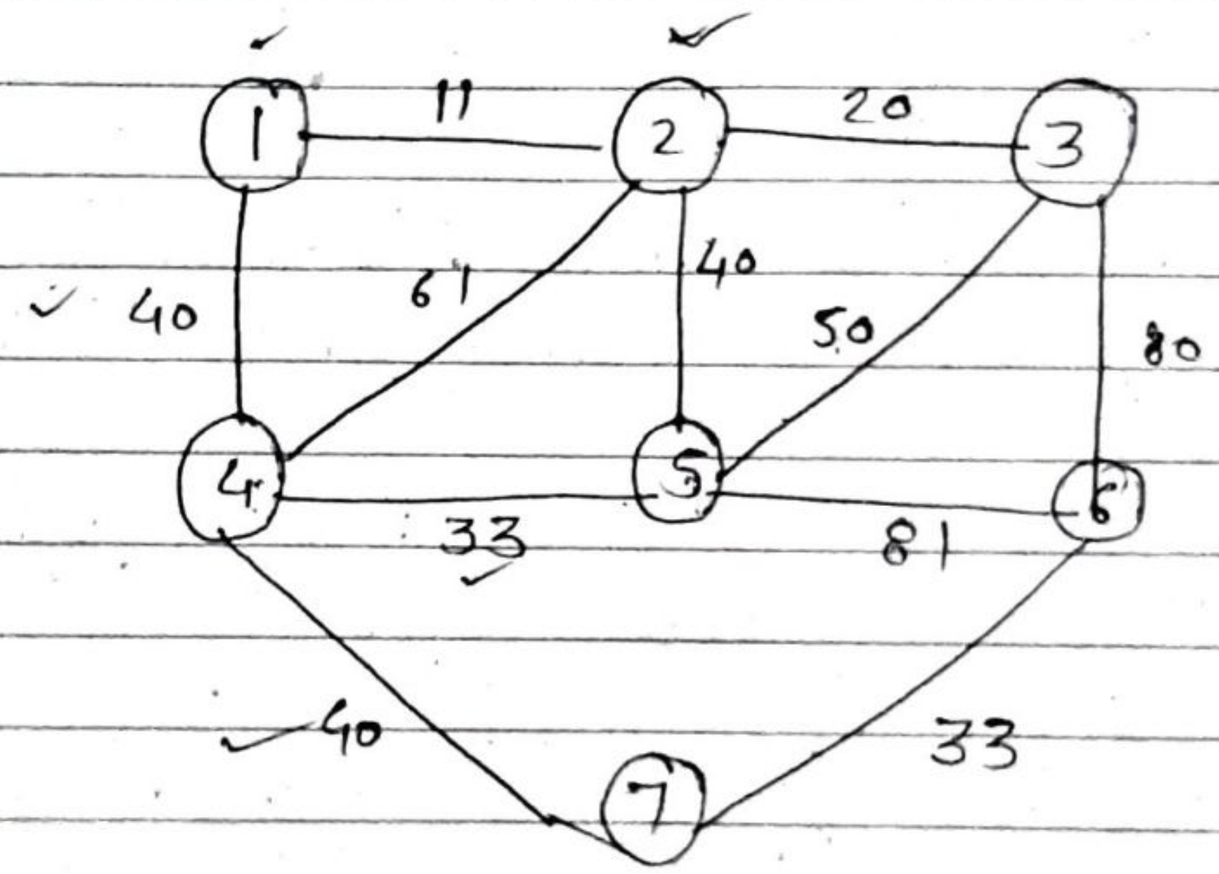
PAGE No.
DATE / /

Now Draw minimum spanning tree from edges



Σ edges = 14

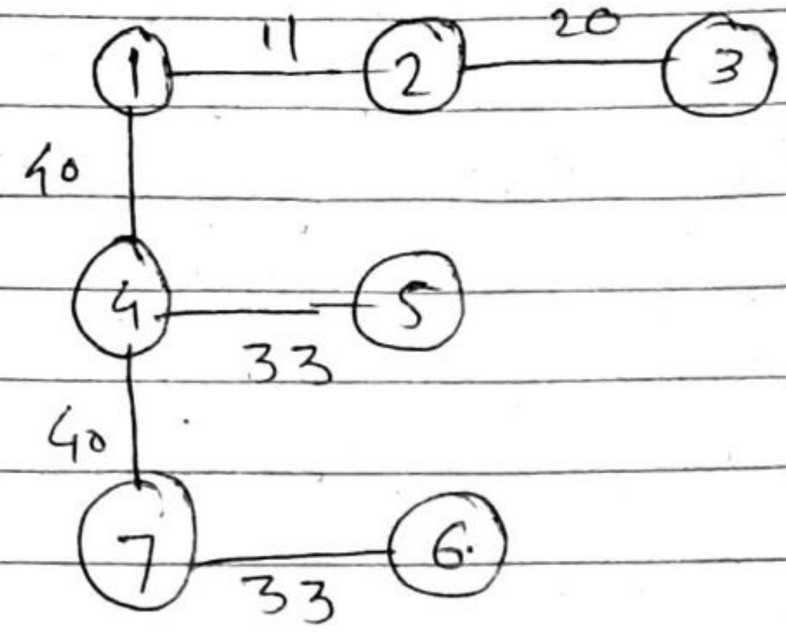
2]



Step. No.	Edge	Vertex set
Initial	—	{1}
1)	(1,2)	{1,2}
2)	(2,3)	{1,2,3}
3)	(1,4)	{1,2,3,4}
5)	(4,5)	{1,2,3,4,5}
6)	(4,7)	{1,2,3,4,5,7}
7)	(7,6)	{1,2,3,4,5,7,6}

PAGE No.
DATE / /

spanning tree



$\sum$ edges = 177

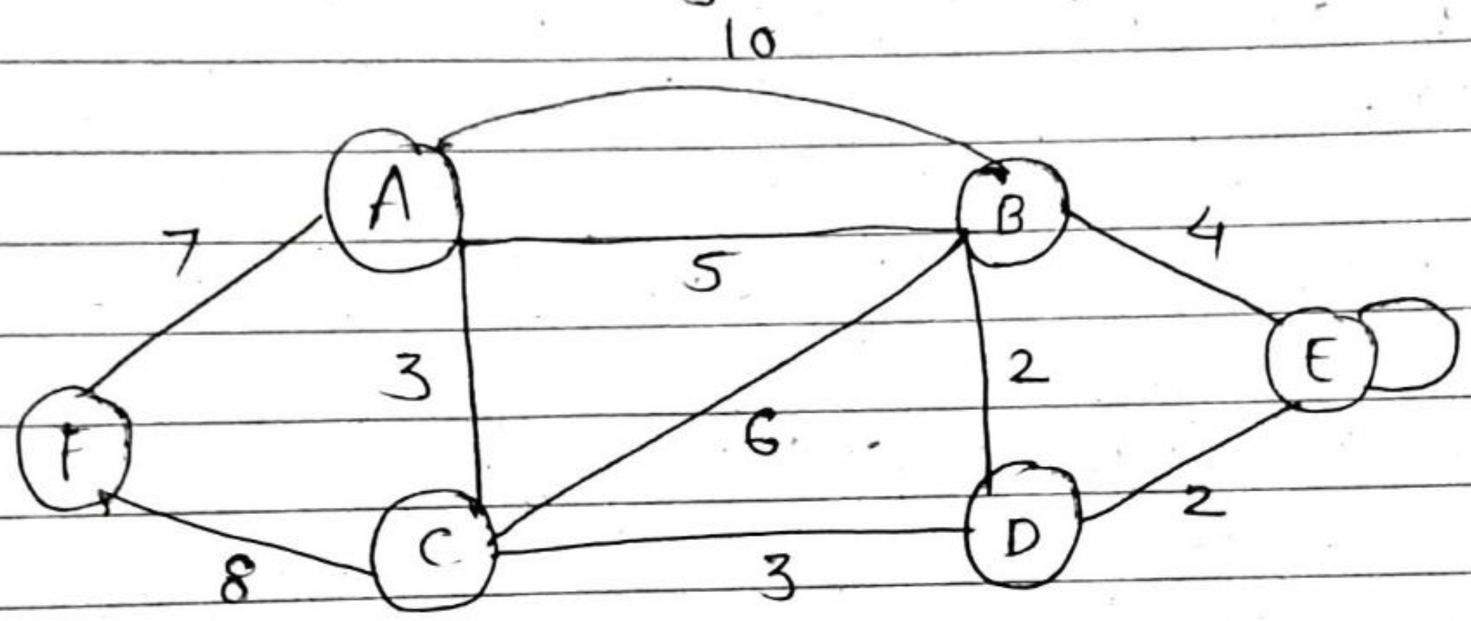
Vertex = 7

edges = 7-1 = 6

↓

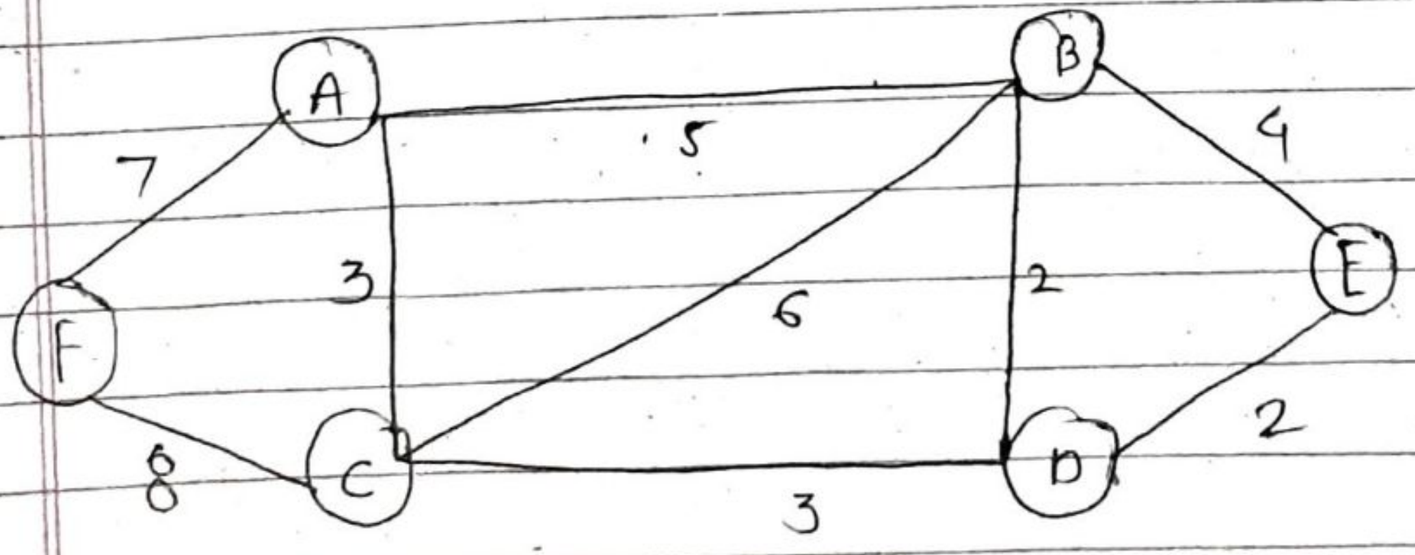
MST

* Krushkal's Algorithm.



Step 1: Remove all the loops & parallel edge (with maximum weight).

And while forming MST there shouldn't be the cycle.



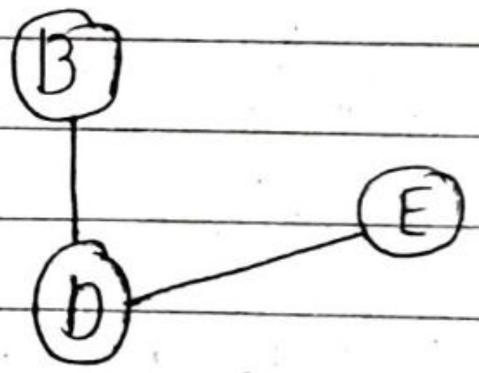
PAGE No.
DATE / /

Step2: Now, pair the edges in sorted order (on the weight basis).

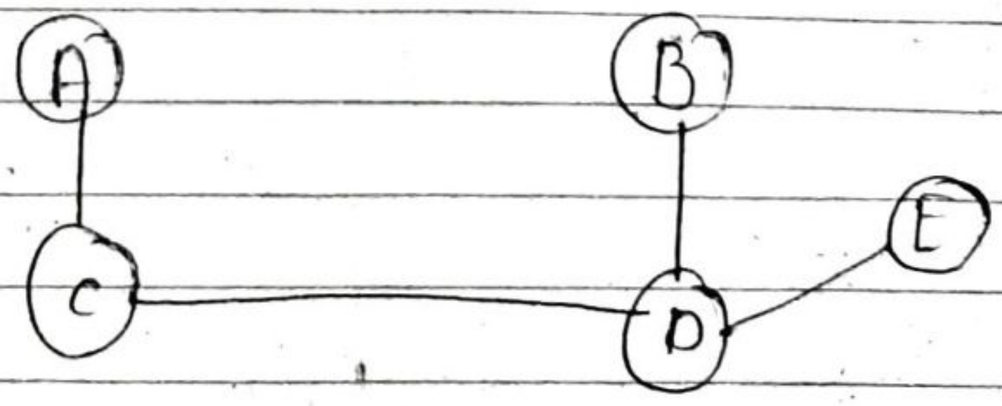
1) → ←
2) → BD, DE
3) → AC, CD
4) → BE.
5) → AB
6) → BC
7) → AF
8) → FC

Now form MST

1) BD, DE



2) AC, CD



PAGE No
DATE / /

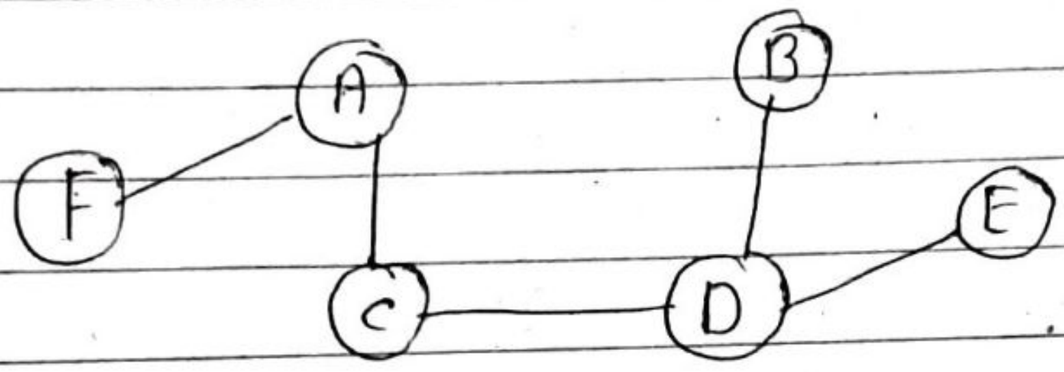
3) BE

Now, If we join this edge it will creted a cycle. Hence we'll eliminate this pair of edge.

4) AB
eliminated

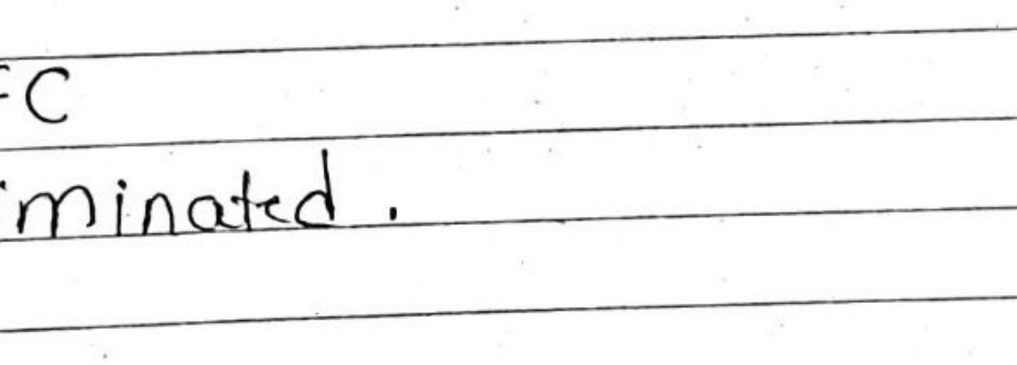
5) BC
eliminated.

6) AF



7) FC
eliminated.

Final MST



Σedges = 16
Vertex = 6
edges = 6-1 = 5