Data Structure

GATEWAY CLASSES

Unit-IV

Trees

ONE SHOT

KAPIL SIR

BCS301 Data Structure

AKTU: Syllabus

Unit-IV

(2) Pre | In | Post → Create Tree
(3) Expression → Create Tree

IV	Trees: Basic terminology used with Tree, Binary Trees, Binary Tree Representation: Array Representation and Pointer(Linked List) Representation, Binary Search Tree, Strictly Binary Tree ,Complete Binary Tree . A Extended Binary Trees, Tree Traversal algorithms: Inorder, Preorder and Postorder, Constructing Binary Tree from given Tree Traversal, Operation of Insertation , Deletion, Searching & Modification of data in Binary Search . Threaded Binary trees, Traversing Threaded Binary trees. Huffman coding using Binary Tree. Concept & Basic Operations for AVL Tree , B Tree & Binary Heaps

Q.23. Define complete binary tree, strictly binary tree and extended binary tree. 2020-21, 2 marks

Q.24. Define Binary tree. In order and post order traversal of a tree are given below:

Preorder: H D I B J E K A F C G

Postorder: A B D H I E J K C F G

Construct the binary tree and determine the postorder traversal of the tree. 2020-21, 7 marks

Q.25. In a complete binary tree if the number of nodes is 1000000. What will be the height of complete binary tree. 2022-23, 2 marks

Q.26. The order of nodes of a binary tree in inorder and postorder traversals are as follows:

Inorder : B I D A C G E H F

Post order: I D B G C H F E A

(i) Draw the corresponding binary tree.

(ii) Write the pre order traversal of the same tree. 2022-23, 10 marks

Tree Data Structure

- So far, we have been studying mainly linear types of data structures : Strings, Arrays, Linked lists, Stacks and Queues.
- Now we will discuss about a nonlinear data structure called a *tree*.
- This data structure is mainly used to represent data containing a **hierarchical relationship** between elements, e.g., records, family trees and tables of contents.
- Tree is a data structure which allows us to associate a **parent – child relationship** between various pieces of data and allows us to arrange our records, data and files in a hierarchical fashion.
- Operating systems uses tree data structure to manage our files using **hierarchical file system**.
- First we investigate a special kind of tree, called a *binary tree*, which can be easily maintained in the computer.

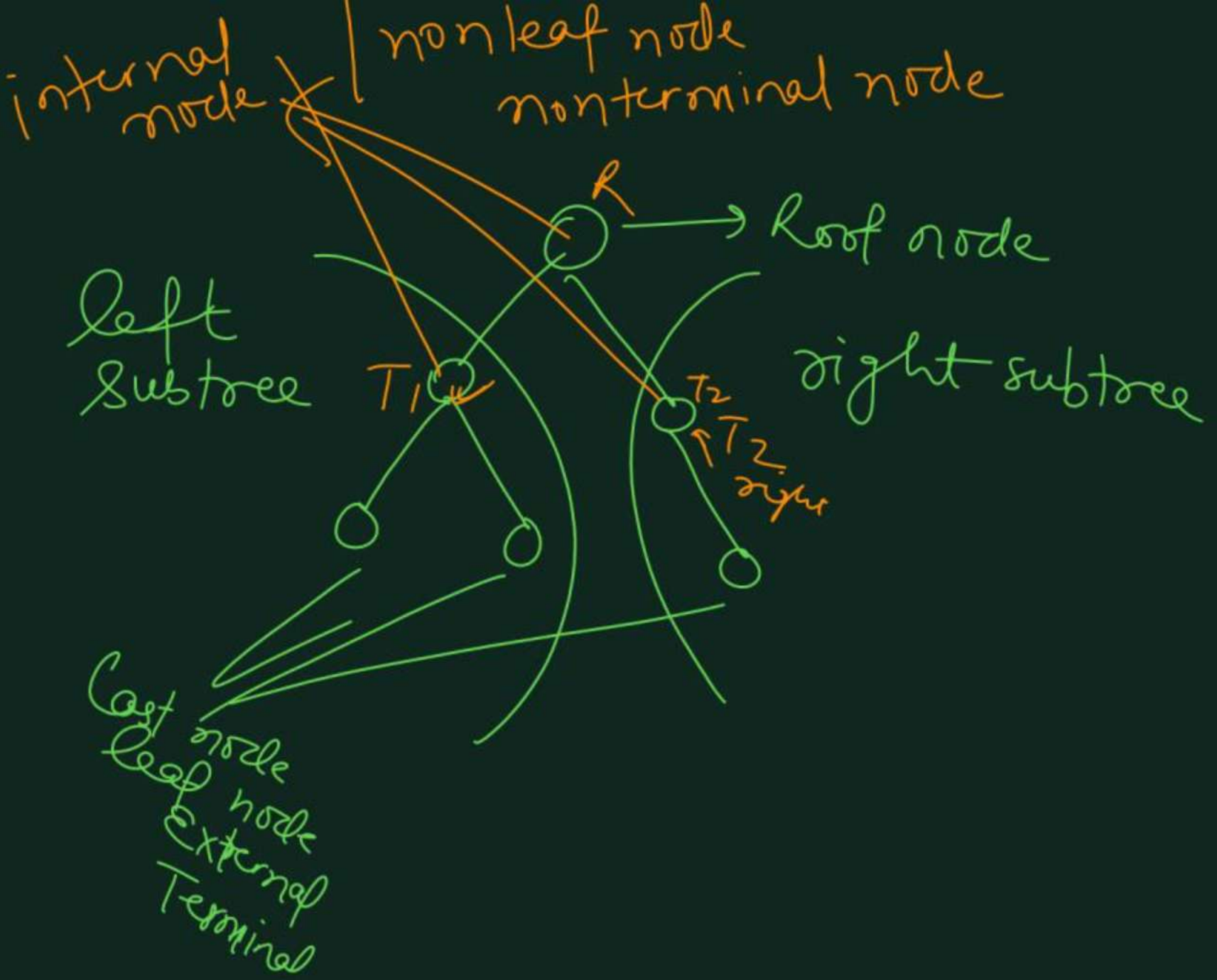
BINARY TREES

- It is a finite set of elements and each node of binary tree can have at most 2 children (branches).
- In other words, A *binary* tree T is defined as a finite set of elements, called *nodes*, such that:

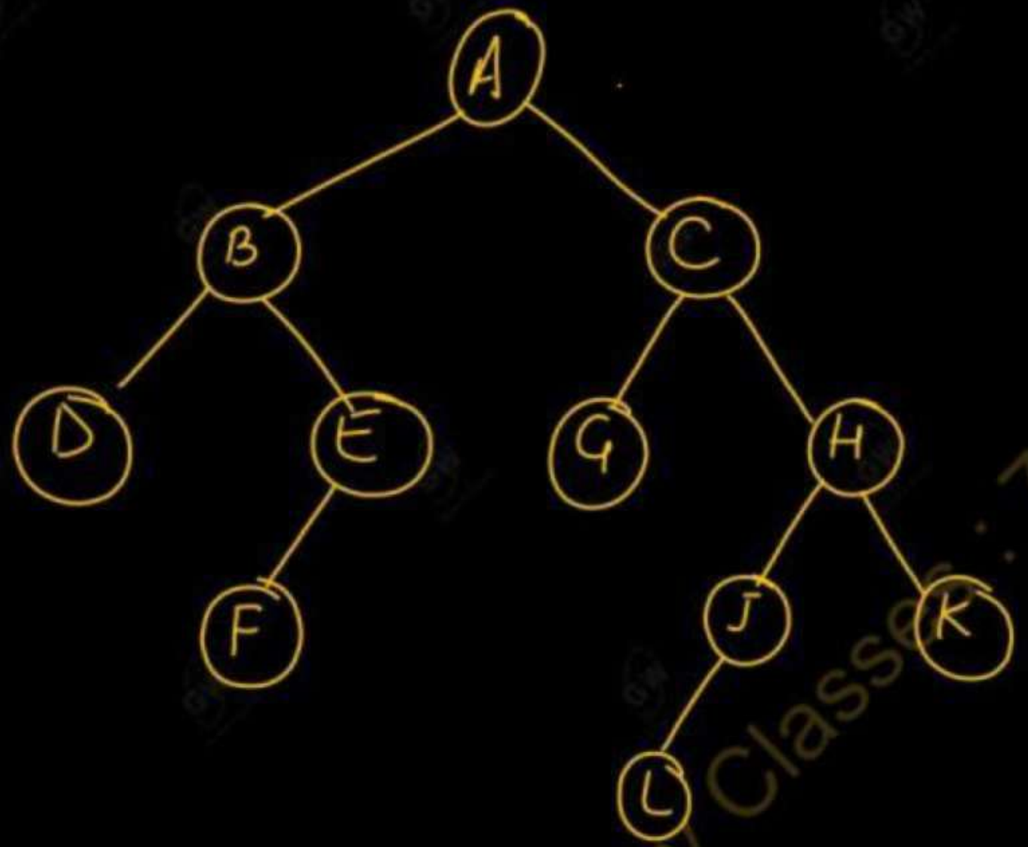
(a) T is empty (called the null tree or *empty tree*), or

(b) T contains a distinguished node *R*, called the *root* of *T*, and the remaining nodes of *T* form an ordered pair of disjoint binary trees T_1 and T_2.

- If *T* does contain a root *R*, then the two trees T_1 and T_2 are called, respectively, the *left* and *right* subtrees of R.
- If T_1 is nonempty, then its root is called the *left successor* of *R*; similarly, if T_2 is nonempty, then its root is called the *right successor* of *R*.
- A binary tree T is frequently presented by means of a diagram.


internal node
nonleaf node
nonterminal node
R
Root node
left subtree
T_1
T_2
T_2 right
right subtree
leaf node
External Terminal

- Consider the following Binary Tree –



A = Root node
B onwards – left subtree (A)
C onwards – right subtree (A)
B = left successor of A
C = right successor of A
D F G L K = leaf nodes
A B E C H J = internal nodes
or
non leaf nodes

- T consists of 11 nodes, represented by the letters *A* through *L*, excluding *I*.
- The root of *T* is the node *A* at the top of the diagram.
- A left-downward slanted line from a node *N* indicates a left successor of *N*, and a right-downward slanted line from *N* indicates a right successor of N.

- Observe that:

(a) *B* is a left successor and *C* is a right successor of the node *A*.

(*b*) The left sub tree of the root *A* consists of the nodes *B, D, E* and *F*, and the right subtree of *A* consists of the nodes C, G, *H, J, K* and *L*.

- Any node *N* in a binary tree *T* has either 0, 1 or 2 successors.
- The nodes *A, B*, C and *H* have two successors, the nodes *E* and *J* have only one successor, and the nodes *D, F, G, L* and *K* have no successors.
- The nodes with no successors are called ***terminal nodes.*** / leaf / external / last

- The above definition of the binary tree *T* is recursive since *T* is defined in terms of the binary sub trees T_1 and T_2.
- This means, in particular, that every node *N* of T contains a left and a right subtree.
- Moreover, if *N* is a terminal node, then both its left and right sub trees are empty.
- Binary trees *T* and *T'* are said to be ***similar*** if they have the same structure or, in other words, if they have the same shape.
- The trees are said to be ***copies*** if they are similar and if they have the same contents at corresponding nodes.

Converting Algebraic Expressions into binary Tree

- Consider any algebraic expression *E* involving only binary operations, such as

$$E = (a - b)/((c * d) + e)$$

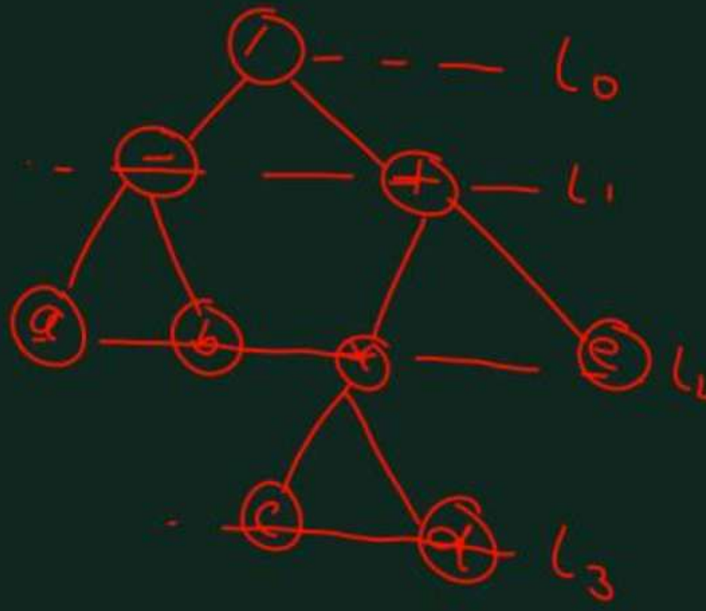
- *E* can be represented by means of the binary tree *T as* pictured in next figure.
- That is, each variable or constant in *E* appears as an "internal" node in *T* whose left and right subtrees correspond to the operands of the operation. For example:

(a) In the expression E, the operands of $+$ are $c * d$ *and* e.

(b) In the tree *T*, the sub trees of the node + correspond to the sub expressions $c * d$ *and* e.

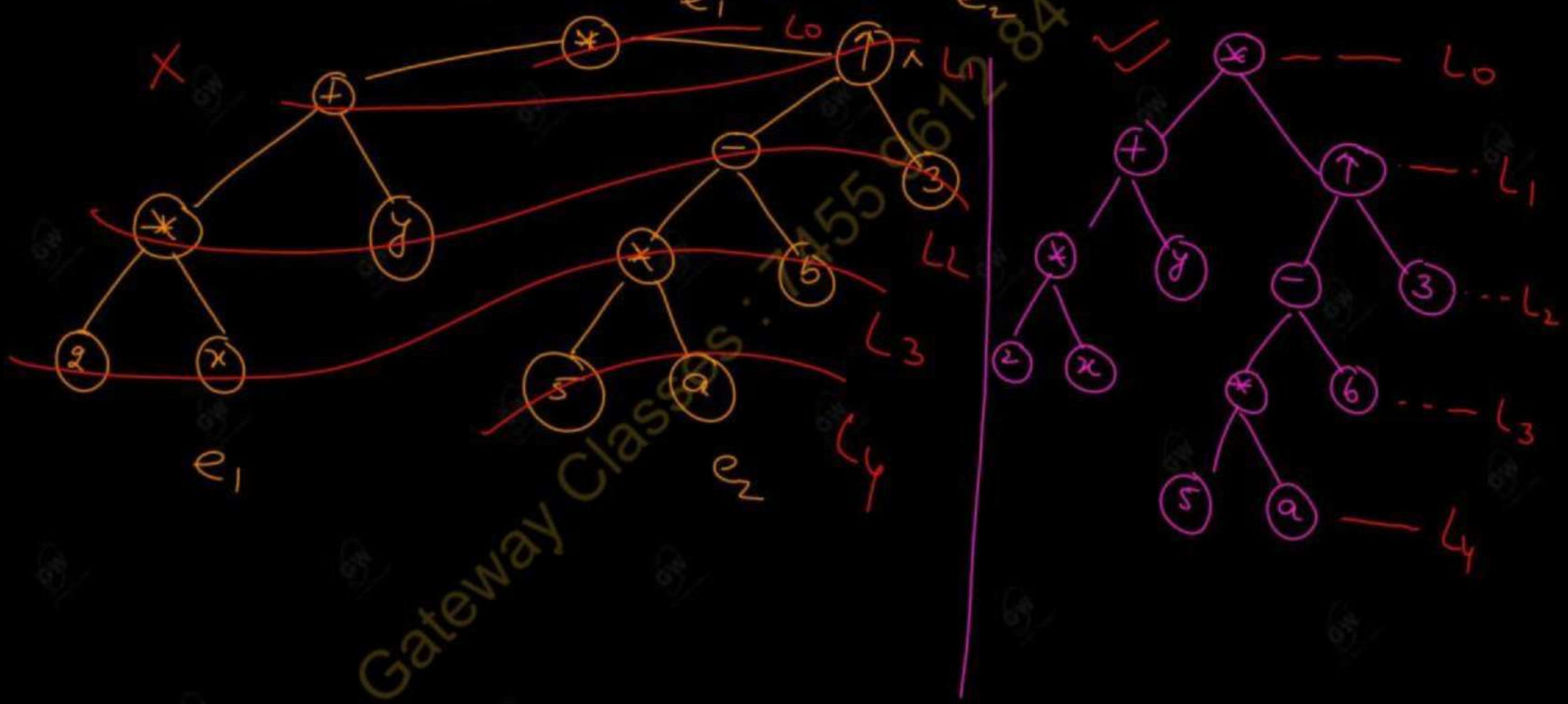
- Clearly every algebraic expression will correspond to a unique tree, and vice versa.

$$E = (a-b)/((c*d)+e)$$

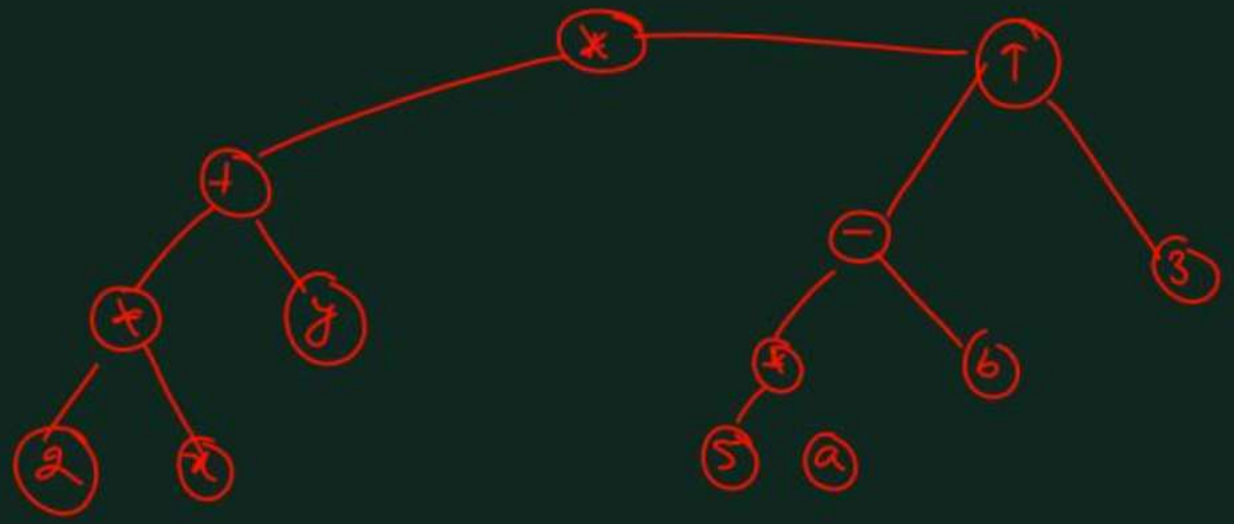


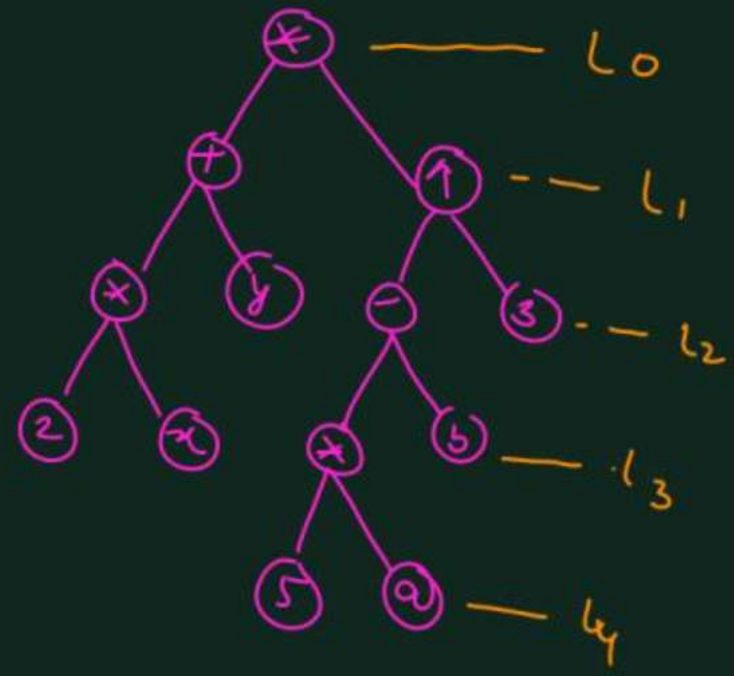
EXAMPLE : Consider the following algebraic expression *E and draw binary tree:-*

$$E = (2x + y)(5a - b)^3$$



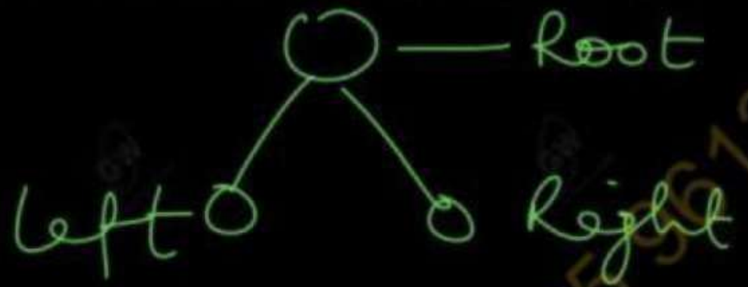
$$E = \overbrace{(2x+y)} \cdot \overbrace{(5a-b)}^{3}$$





Tree Terminology

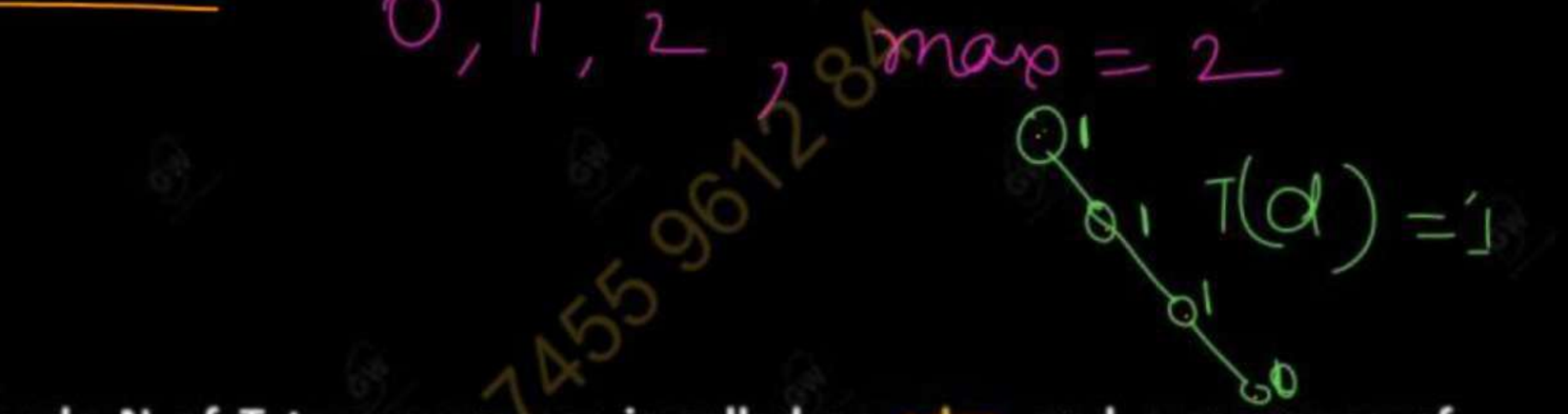
- **Root : -** Tree contain a special node called the root node of the tree.



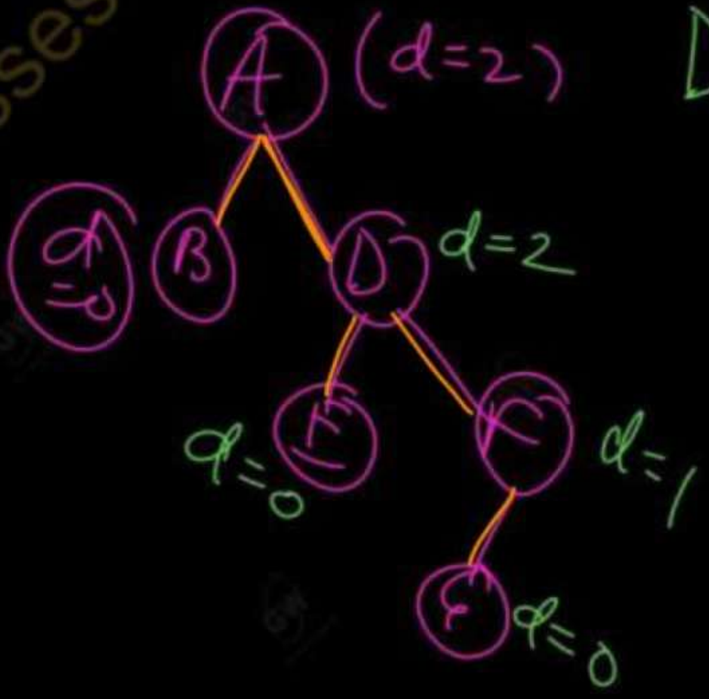
- **Node:-** The each element of tree is called node.
- **Left and Right successor : -** Suppose *N* is a node in *T* with left successor S_1 and right successor S_2. Then *N* is called the *parent* (or *father)* of S_1 and S_2.

- **Left and right child : -** Analogously, is called the *left child* (or *son)* of *N*, and S_2 is called the right child (or son) of N. Furthermore, S_1 and S_2 are said to be siblings (or *brothers*).

- **Degree of a node:-** In binary tree each node can have at most two children, it means the degree of a node is 2 in binary tree.



- The line drawn from a node N of T to a successor is called an edge, and a sequence of consecutive edges is called a path.



Degree(Tree) = max degree = 2

- **Terminal nodes or external nodes or leaf nodes or last nodes:** - A terminal node is called a leaf, and a path ending in a leaf is called a branch. In other words, node with degree 0 is called the terminal node.

Nodes with degree 0 = Leaf node

- **Non Terminal nodes or internal nodes:** - A binary tree can have 0,1,2 successor. A node of T which has any number of successor is called non terminal nodes.

Nodes with One degree = Internal

- **Level of node:-** Each node in a binary tree T is assigned a *level* number, as follows.

- The root R of the tree *T* is assigned the level number 0, and every other node is assigned a level number which is 1 more than the level number of its parent.

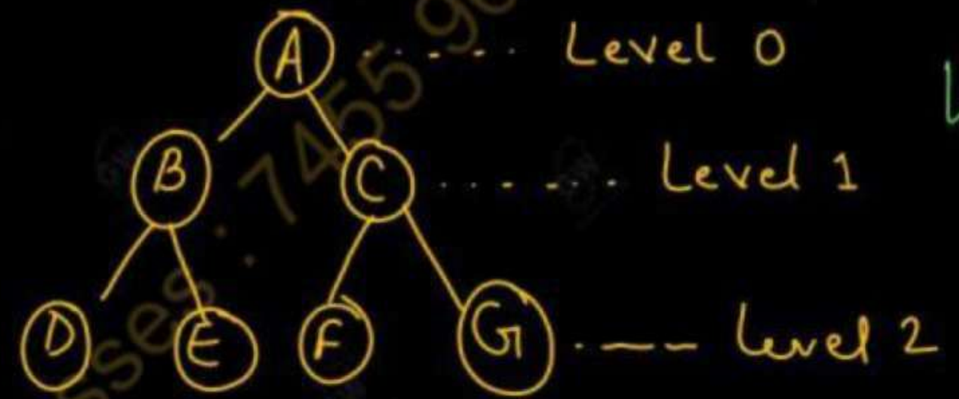
- **Generation : -** Those nodes with the same level number are said to belong to the same generation.
- **Siblings : -** The node with same parents.

B & C = Siblings
E & F = Cousins

- The depth (or height) of a tree T is the maximum number of nodes in a branch of T. This turns out to be 1 more than the largest level number of T. The height of empty tree is 0. The height of a tree containing a single node is 1. The formula used to find the height of a tree is

$$h = l_{max} + 1$$

- The following tree T has depth 3.



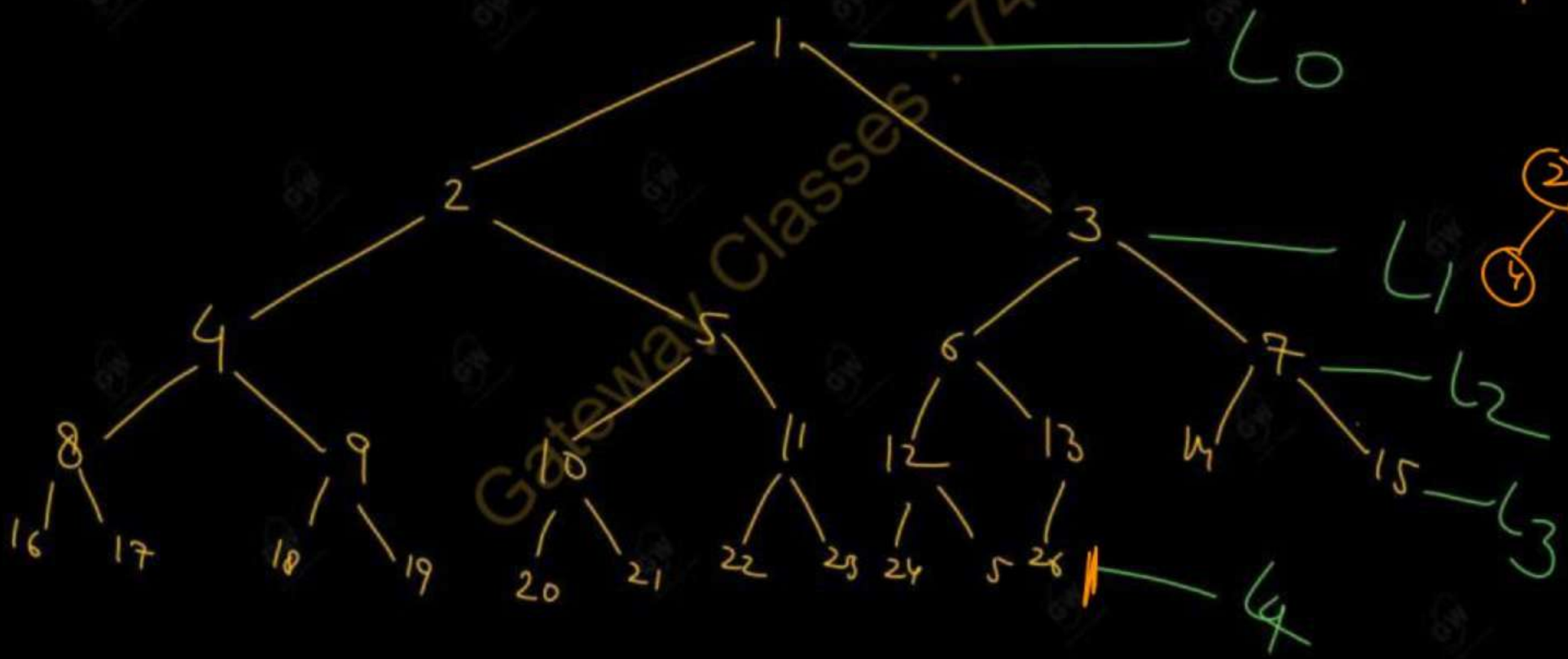
height = 2+1 = 3

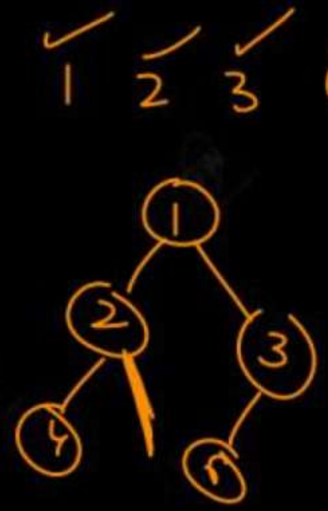
□ See one more example in next slide.

- Binary trees T and *T* are said to be *similar* if they have the same structure or, in other words, if they have the same shape.
- The trees are said to be copies if they are similar and if they have the same contents at corresponding nodes.

Complete Binary Tree -

- Consider any binary tree T. Each node of T can have at most two children.
- Accordingly, one can show that level of T can have at most 2^r nodes.
- The tree T is said to be complete if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.
- Consider the following complete binary tree -





- One can easily determine the children and parent of any node K in any complete tree T Specifically, the left and right children of the node K are, respectively, 2 * K and 2 * K + 1, and the parent of K is the node [K / 2].
- For example, the children of node 9 are the nodes 18 and 19, and its parent is the node [9 / 2] = 4. The depth d, of the complete tree *T*, with n nodes is given by -

$$D = [log_2 n + 1]$$

- This is a relatively small number.
- For example, if the complete tree *T*, has n = 1000, 000 nodes, then its depth D = 21.

Imp

$D = \lfloor \log_2 n + 1 \rfloor$

$= \lfloor \log_2 1000000 + 1 \rfloor$

$= \lfloor \log_2 2^{20} + 1 \rfloor$

$= 20 + 1$

$= 21$

REPRESENTATION OF BINARY TREES IN MEMORY

- Let T be a binary tree.
- Three are two ways to represent T in memory.

1. Linked List representation of Binary Tree in memory - The first and usual way is called the link representation of T and is analogous to the way linked lists are represented in memory.

2. Sequential representation of binary tree in Memory - The second way, which uses a single array, called the sequential representation of T.

1. Linked Representation of Binary Trees -

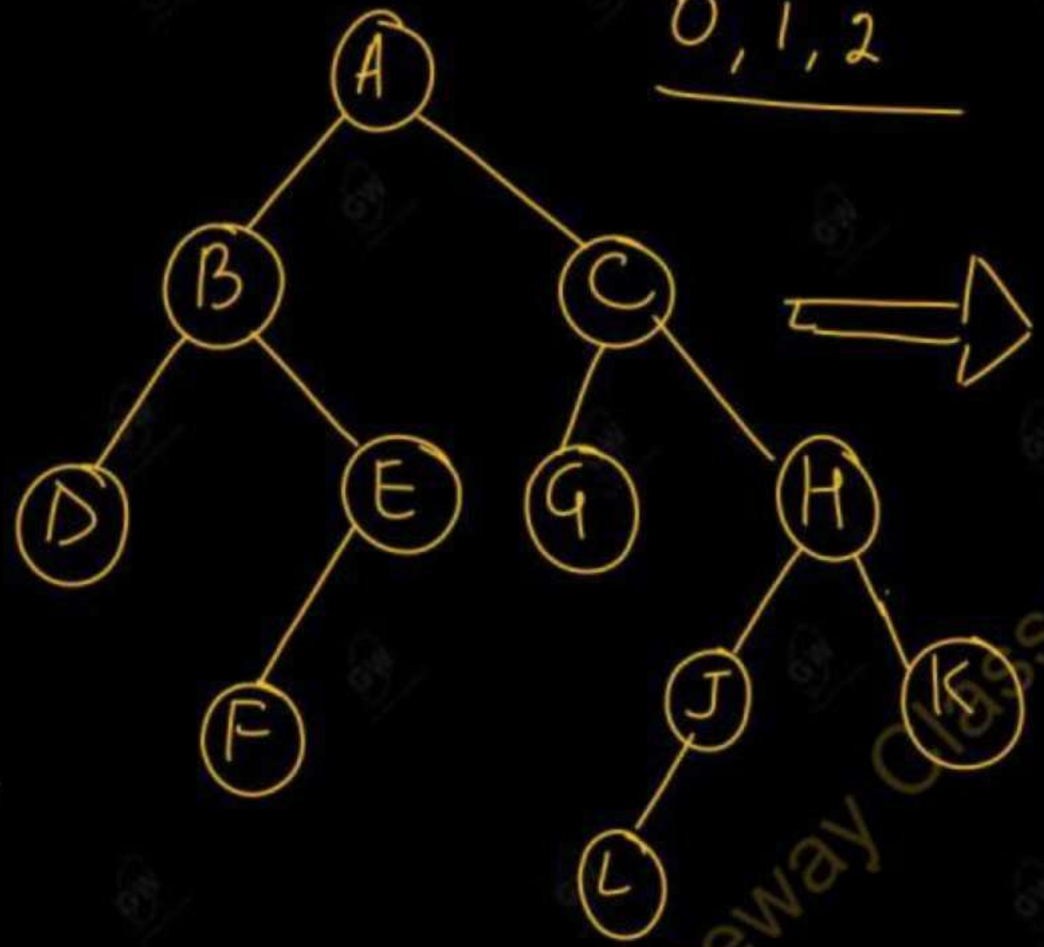
- Consider a binary tree T.
- In this method, T will be maintained in memory by means of a linked representation which uses three parallel arrays, INFO, LEFT and RIGHT, and a pointer variable ROOT as follows.
- First of all, each node N of T will correspond to a location K such that:-

(1) INFO [K] contains the data at the node N.

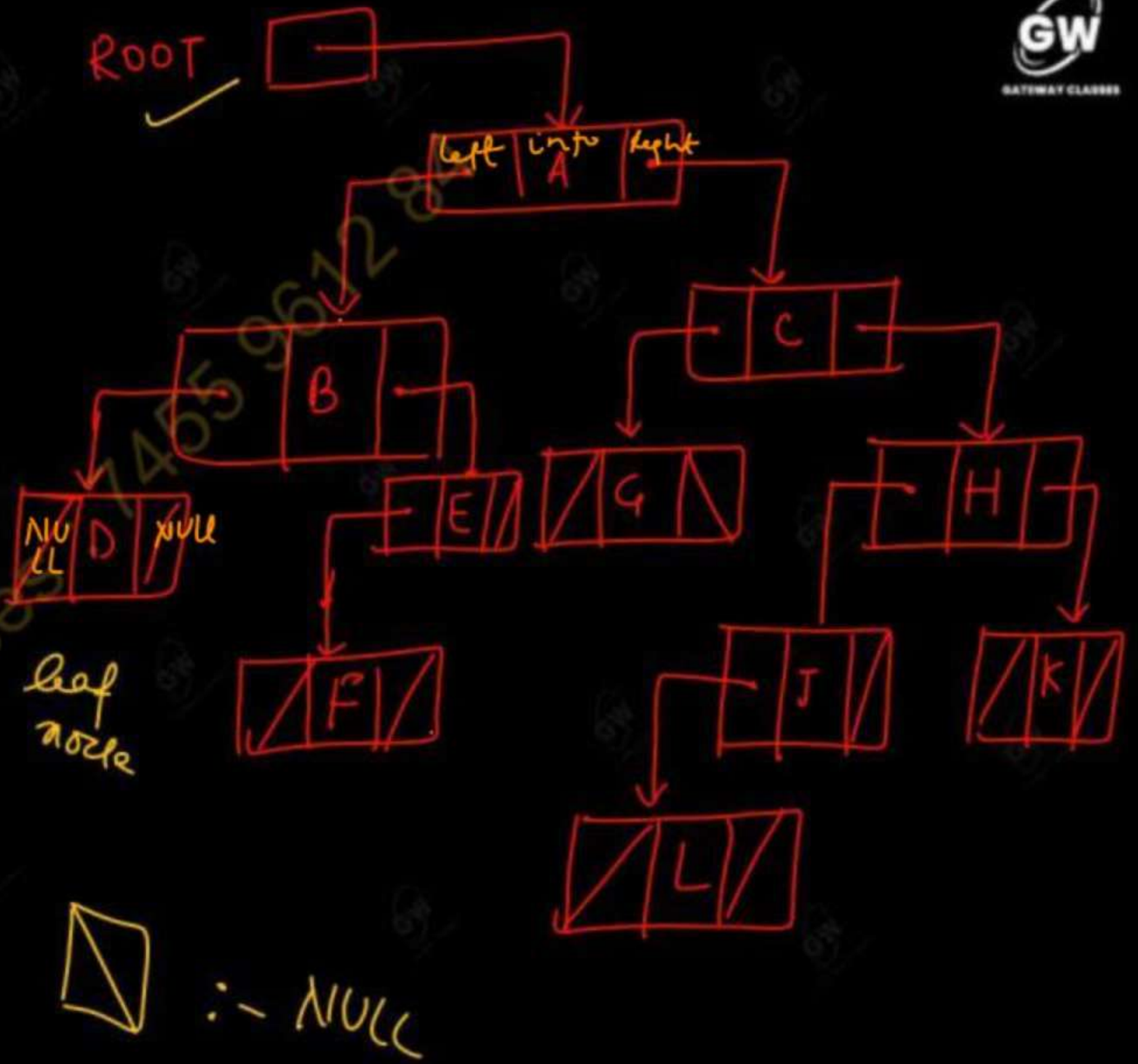
(2) LEFT [K] contains the location of the left child of node N.

(3) RIGHT [K] contains the location of the right child of node N.

- Furthermore, ROOT will contain the location of the root R of T.
- If any subtree is empty, then the corresponding pointer will contain the null value; if the tree T itself is empty, then ROOT will contain the null value.

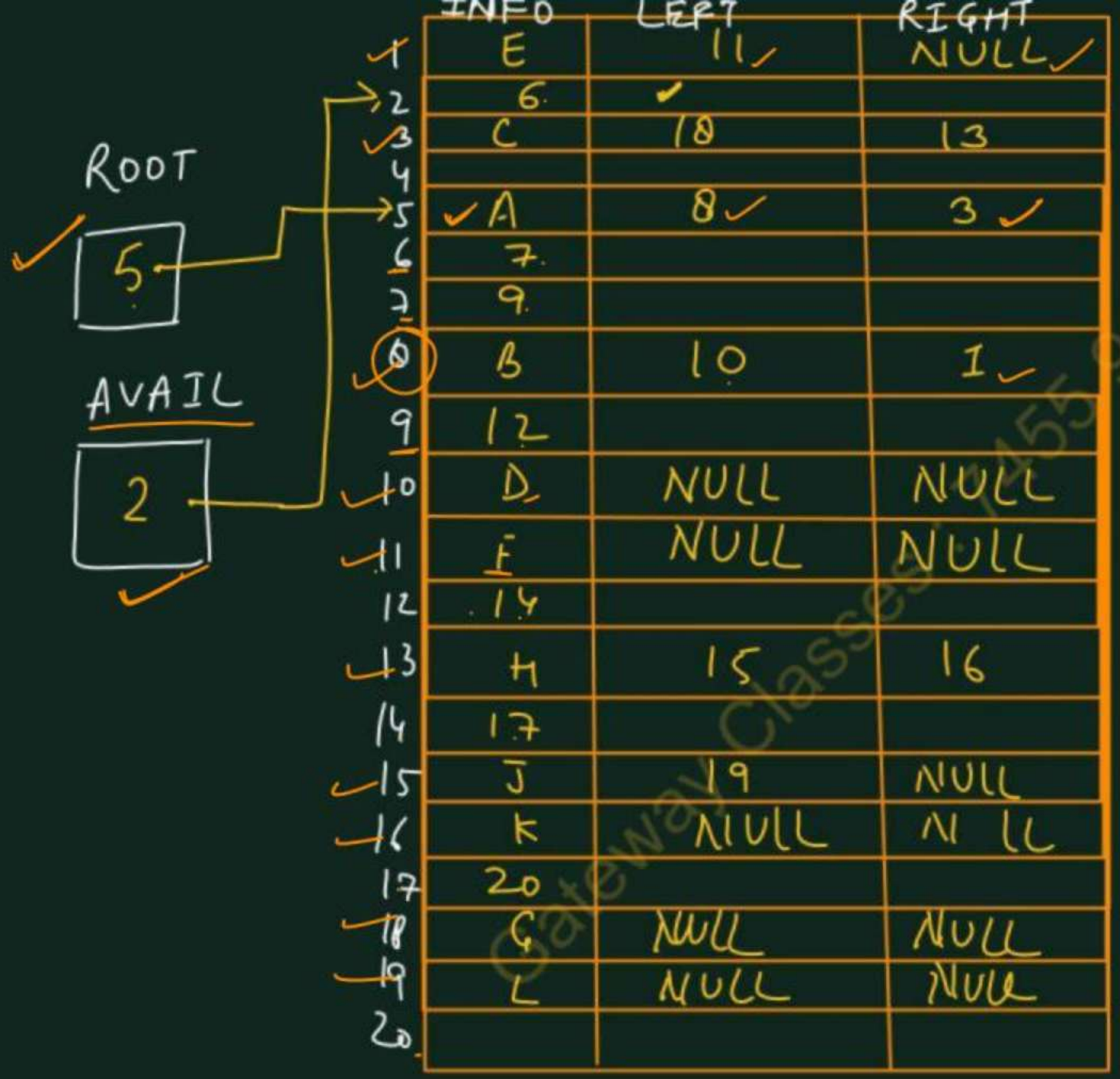


0, 1, 2



leaf node

:- NULL

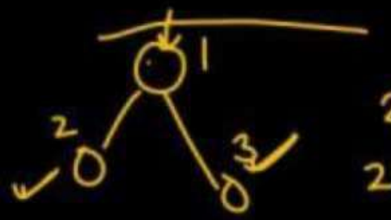


Avail

Root

2. Sequential Representation of Binary Trees -

- Suppose T is a binary tree that is complete or nearly complete.
- Then there is an efficient way of maintaining T in memory called the *sequential representation* of T.



$2 \times 1 = 2$

$2 \times 1 + 1 = 3$

- This representation uses only a single linear array TREE as follows: -

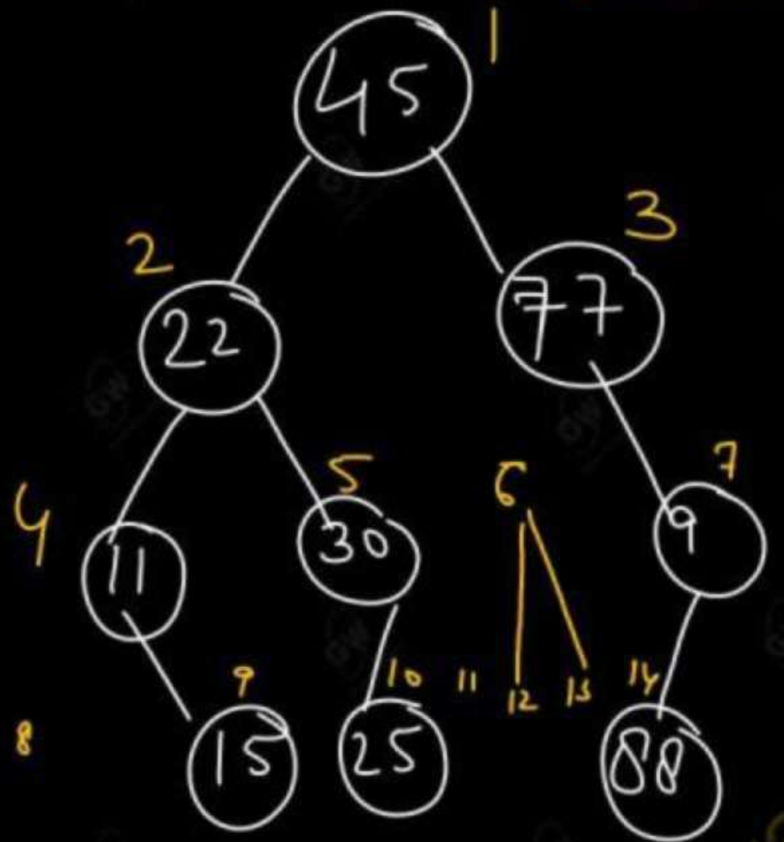
  (a) The root R of T is stored in TREE [1].

  (b) If a node N occupies TREE [K], then its left child is stored in TREE [2K] and its right child is stored in TREE [2 * K + 1].

2 4 6 9 14

- Again, NULL is used to indicate an empty subtree.
- In particular, TREE [1] = NULL indicates that the tree is empty.
- The sequential representation of the binary tree T is represented in the next figures:-

SEQUENTIAL REPRESENTATION OF BINARY TREE



Binary Tree

1	45
2	22
3	77
4	11
5	30
6	NULL
7	9
8	NULL
9	15
10	25
11	
12	
13	
14	88
15	
16	
⋮	
29	

TRAVERSING BINARY TREES

- There are three standard ways of traversing a binary tree T with root R.
- These three algorithms, called preorder, inorder and postorder, are as follows:

- **Preorder:**
  (1) Process the root R.
  (2) Traverse the left subtree of R in preorder.
  (3) Traverse the right subtree of R in preorder.

- **Inorder:**
  (1) Traverse the left subtree of R in inorder.
  (2) Process the root R.
  (3) Traverse the right subtree of R in inorder.

- **Postorder:**
  (1) Traverse the left subtree of R in postorder.
  (2) Traverse the right subtree of R in postorder.
  (3) Process the root R.

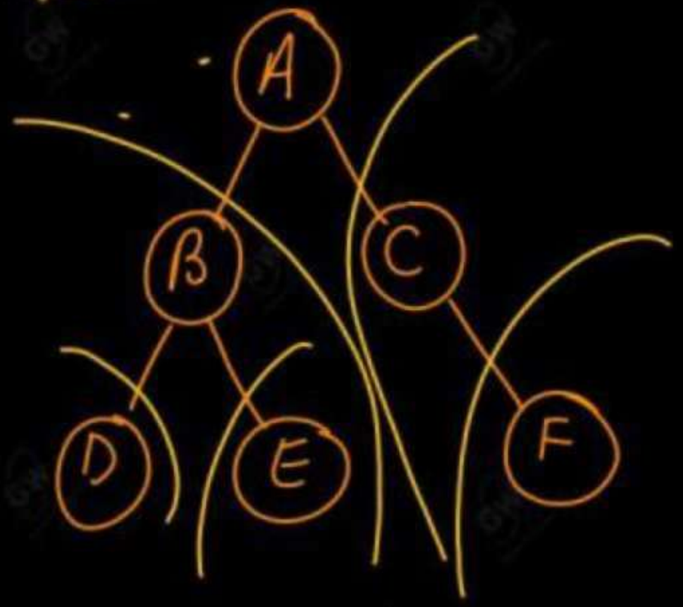
1. Preorder Traversal
2. Inorder
3. Postorder

Preorder — Root left right

Inorder — left Root Right

Postorder — left Right Root

Example:-



Root left Right

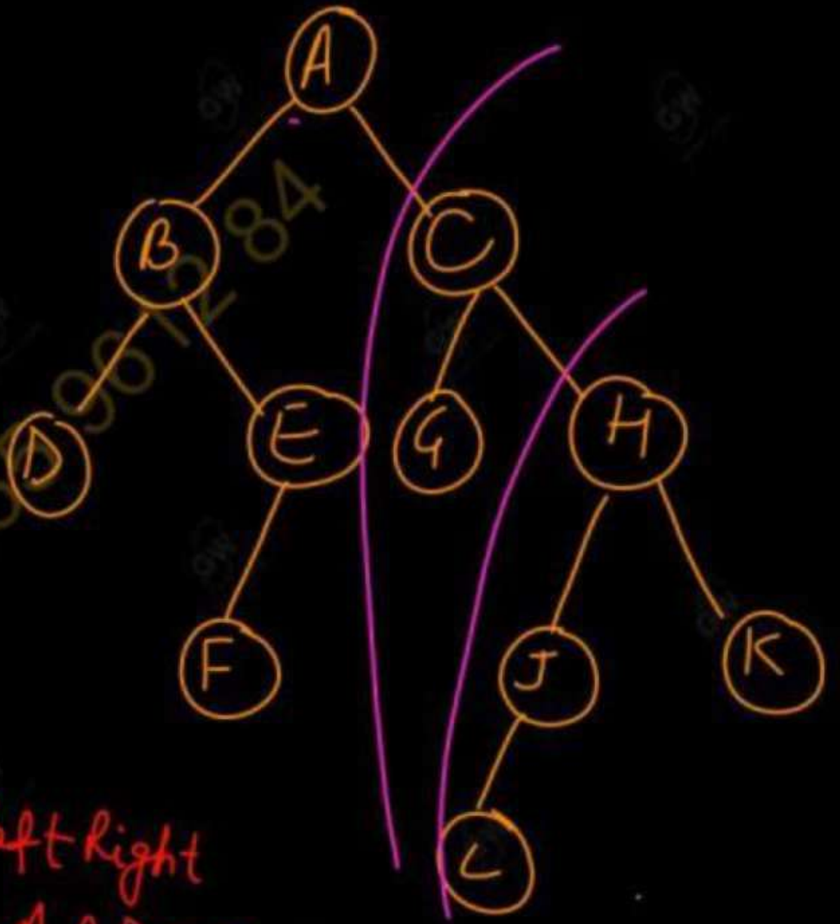
1) PREORDER :- A B D E C F

Left Root Right

2) IN ORDER :- D B E A C F

Left Right Root

3) POSTORDER :- D E B F C A



Root Left Right

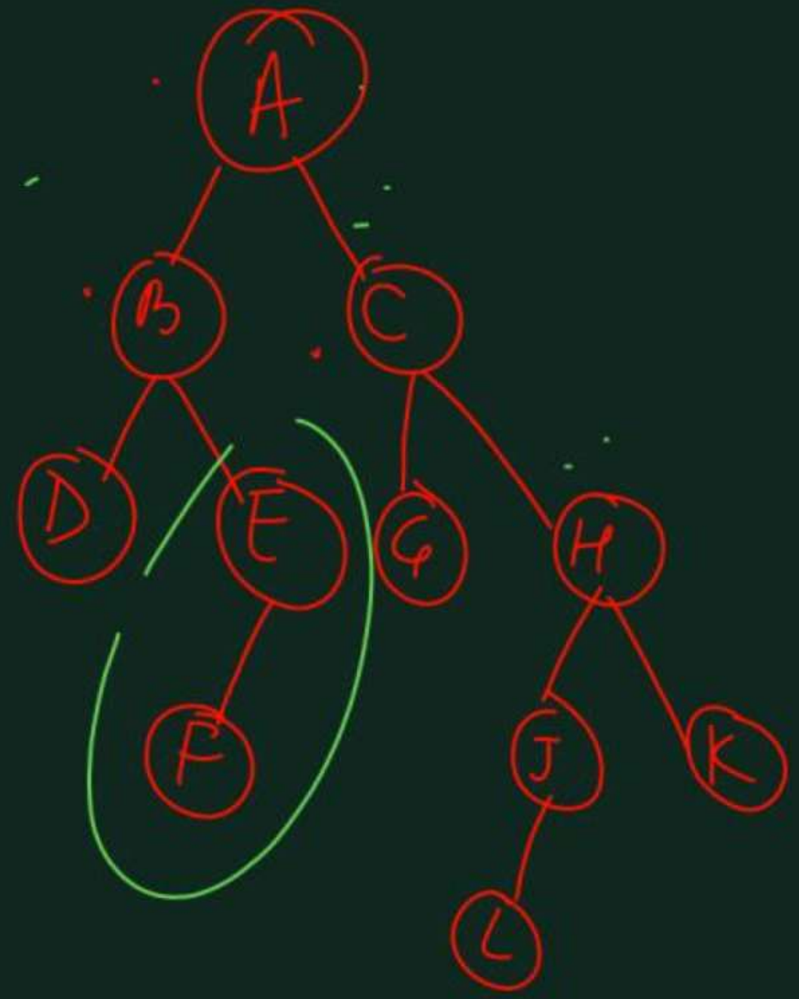
PRE :- A B D E F C G H J L K

Left Root Right

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

Left Right Root

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



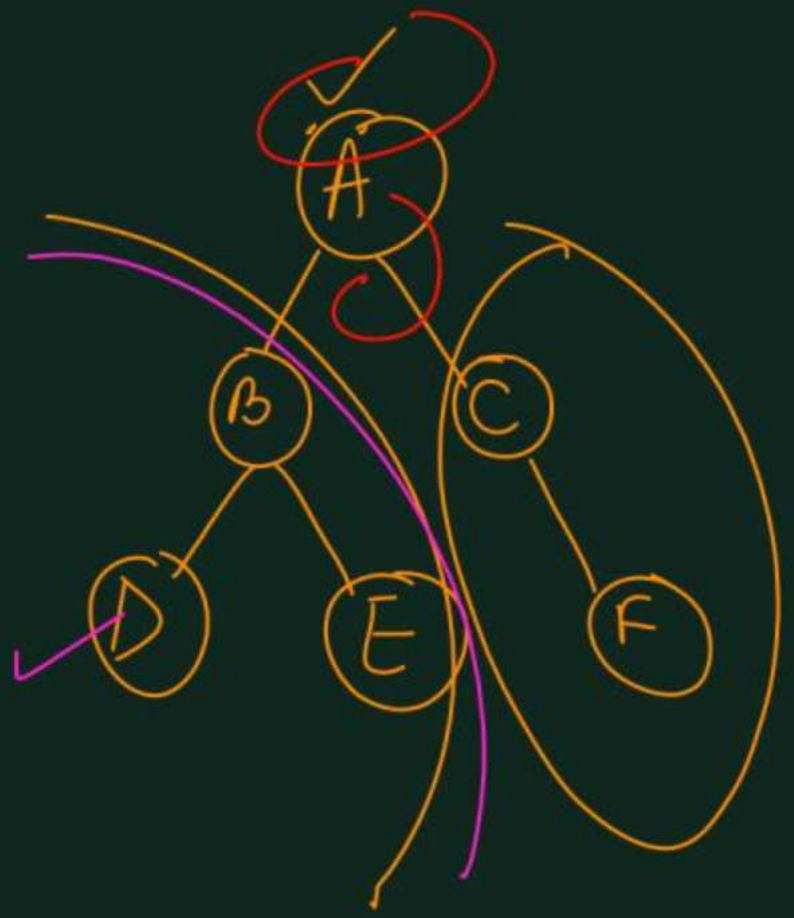
Preorder A B D E F C G H J L K

Inorder:- left Root Right

D B F E A G C L J H K

Postorder:- left Right Root

D F E B G L J K H A



Preorder :- Root Left Right

A B D E C F

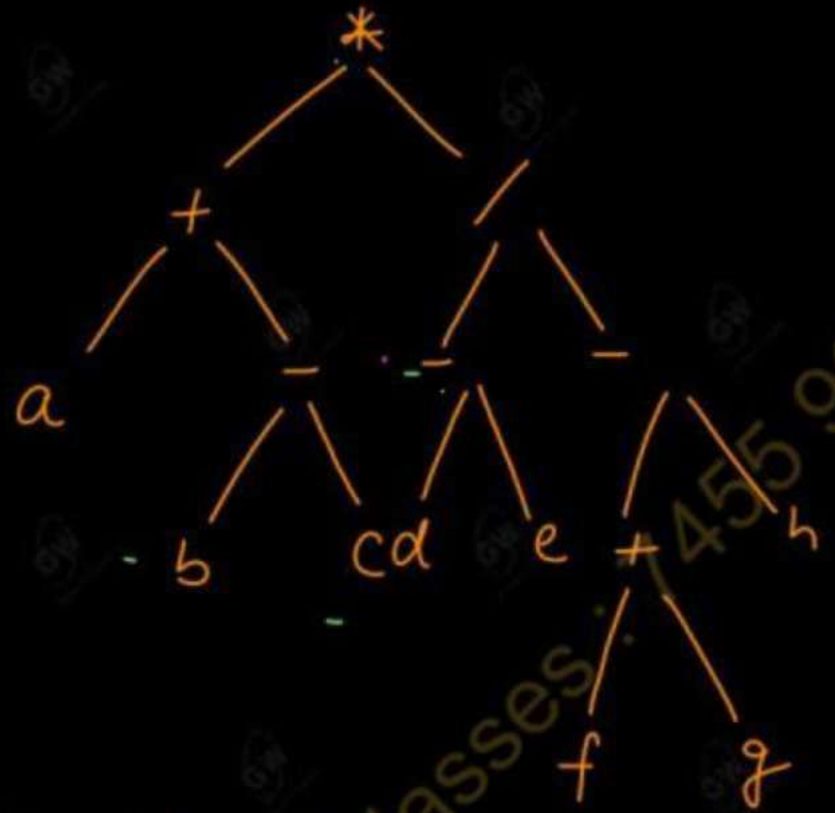
Inorder :- Left Root Right

D B E A C F

Postorder :- Left Right Root

D E B F C A

Example:-

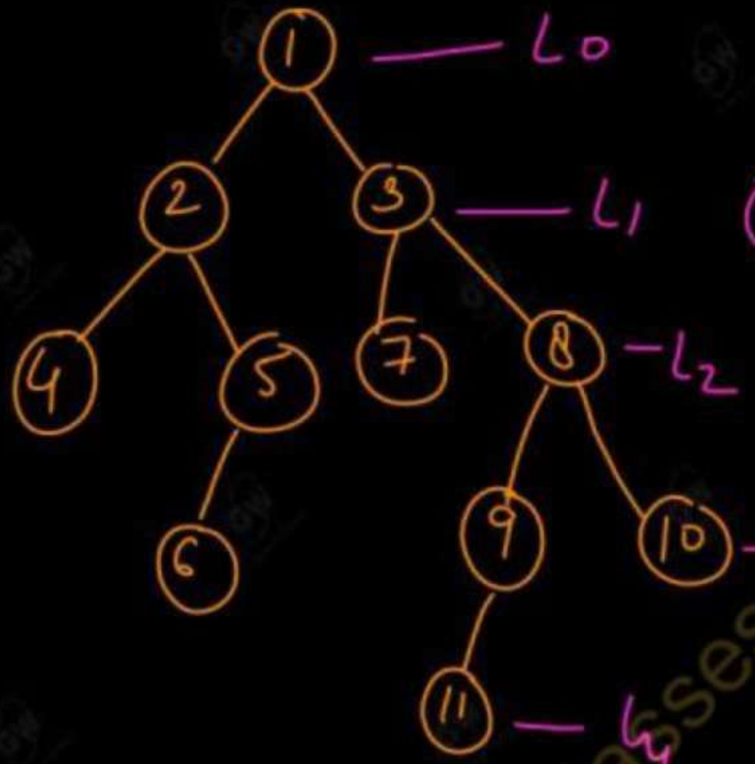


PRE :- Root, Left, Right * + a - b c / - d e - + f g h

IN :- Left Root Right a + b - c * d - e / f + g - h

POST :- Left Right Root a b c - + d e - f g + h - / *

Example:-



① PRE – 1 2 4 5 6 3 7 8 9 11 10

② IN – Left Root Right 4 2 6 5 1 7 3 11 9 8 10

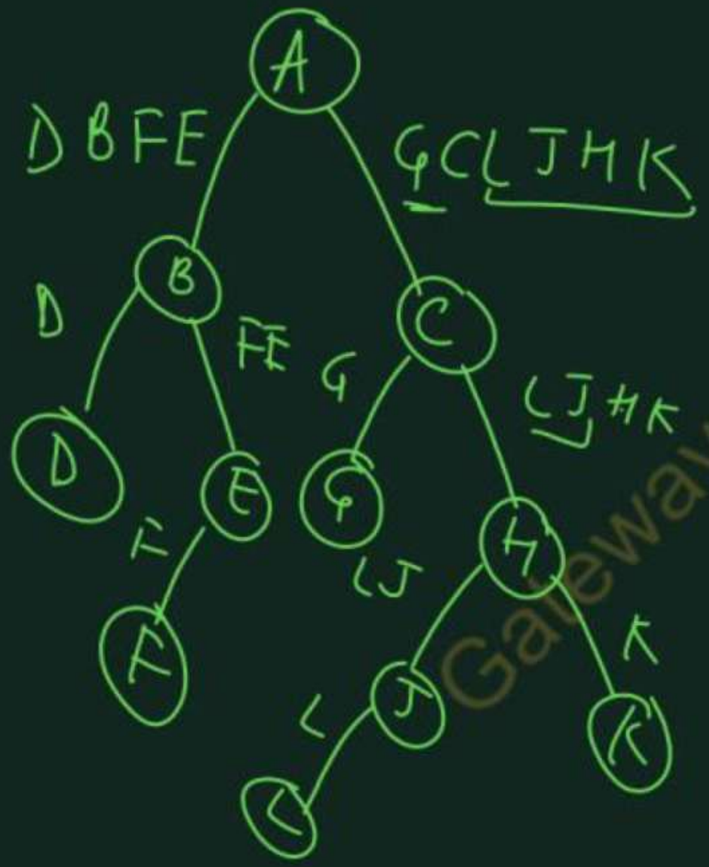
③ POST – Left Right Root 4 6 5 2 7 11 9 10 8 3 1

④ LEVEL BY LEVEL – 1, 2, 3, 4, 5, 7, 8, 6, 9, 10, 11

Example - Draw T.

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

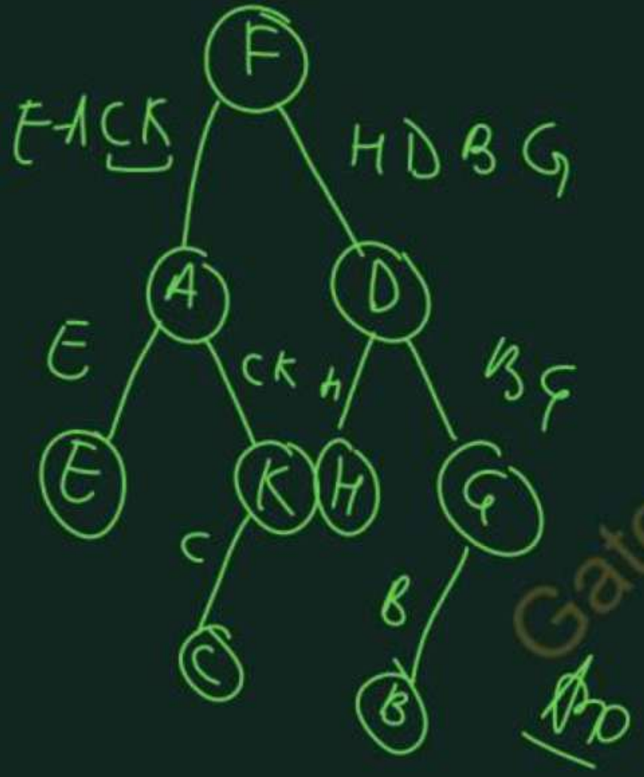
Postorder :- D F E B G L J K H C A

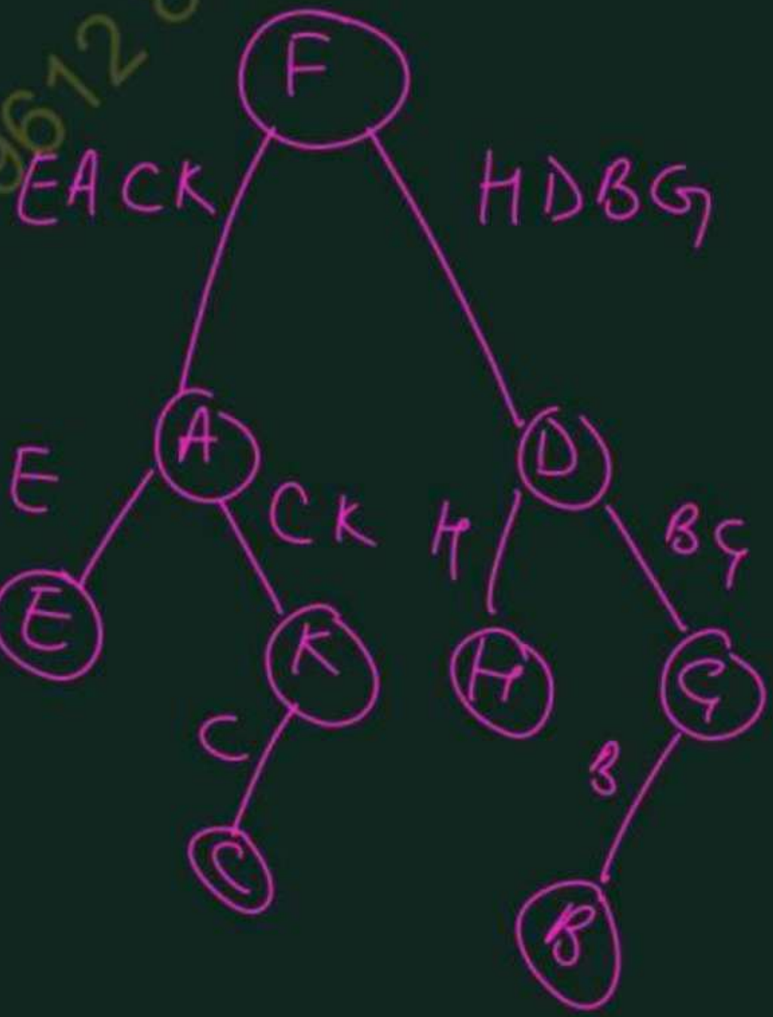


Example:- Draw the tree T.

Inorder:- E A C K F H D B G

Preorder:- F A E K C D H G B





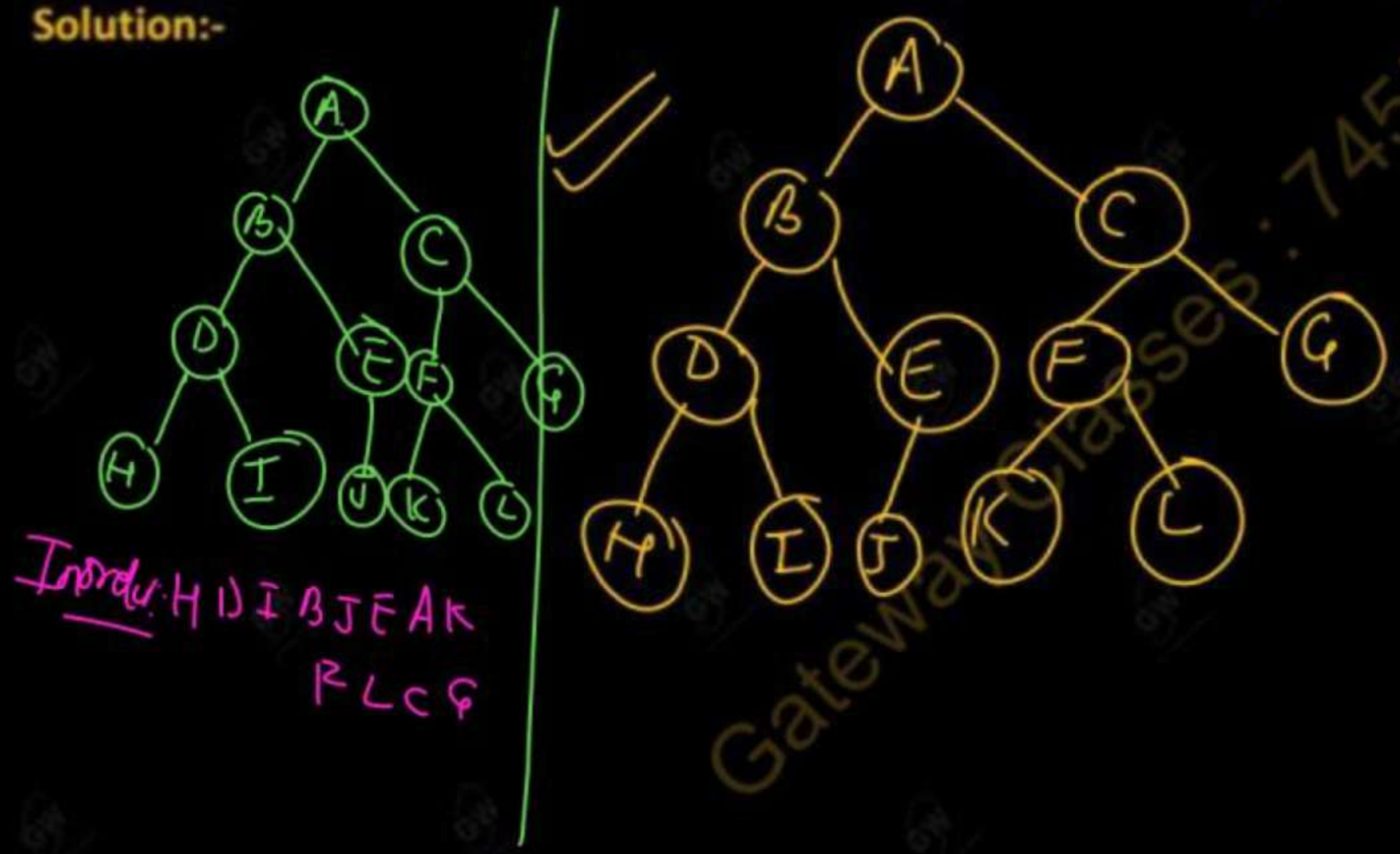
Creation of Binary Tree using Preorder and Postorder Traversal

Example 1:- Construct a binary tree for the following preorder and post order and write its in order traversal.

Preorder : A B D H I E J C F K L G (Root, left, Right)

Post order : H I D J E B K L F C G A (left, Right, Root)

Solution:-



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

PREORDER :-

A B D H I E J C F K L G

INORDER :- left Root Right

H D I B J E A K F L C G

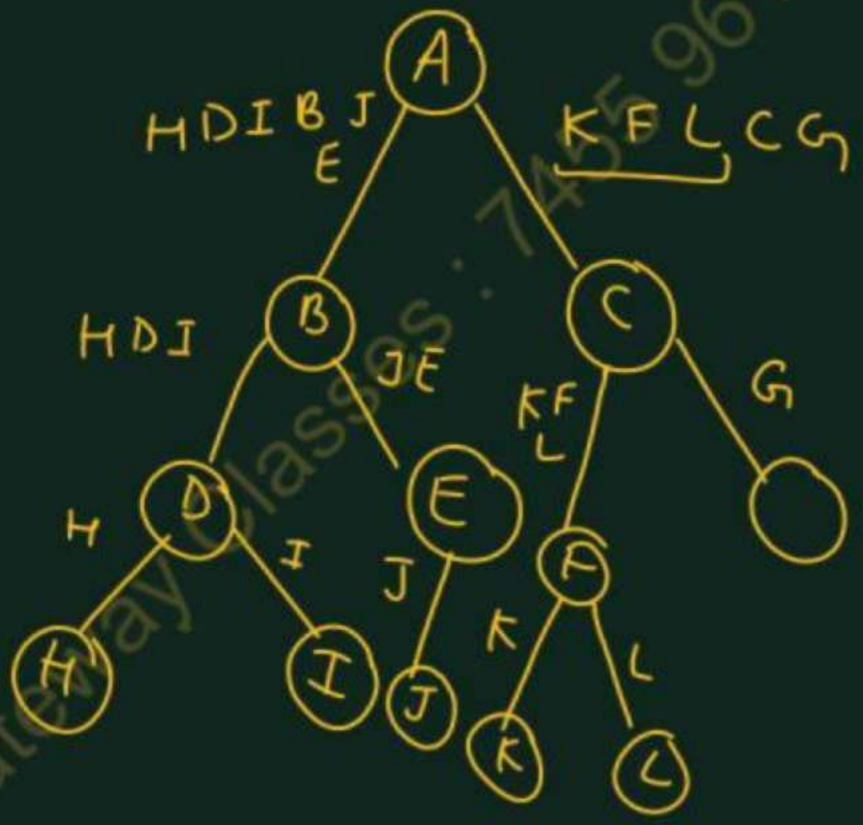
Inorder: · H D I B J E A K F L C G

Preorder _____

Postorder _____

PREORDER :- A B D H I E J C F K L G

INORDER :- H D I B J E A K F L C G



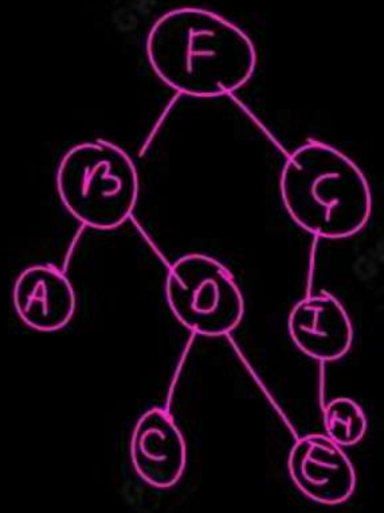
Example 2:- Construct a binary tree for the following preorder and post order and write its in order traversal.

Preorder : F B A D C E G I H

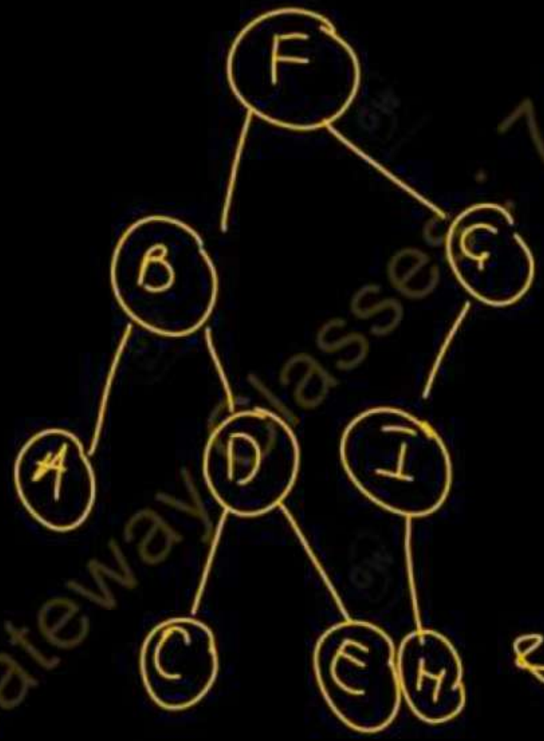
Post order : A C E D B H I G F

match.

Solution:-



Ans



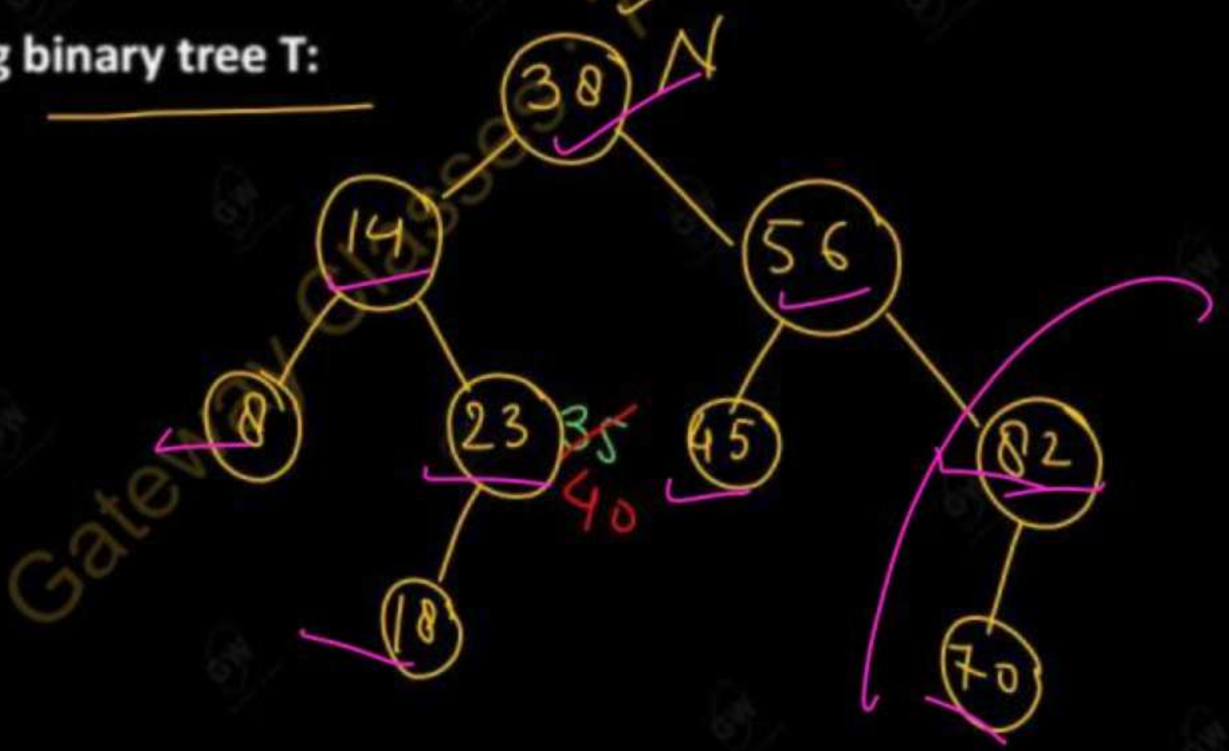
Preorder = ?
Inorder = ?

Binary Tree

match.

BINARY SEARCH TREES

- Suppose T is a binary tree.
- Then T is called a binary search tree (or binary sorted tree) if each node N of T has the following property:
- The value at N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.
- Consider the following binary tree T:



- T is a binary search tree; that is, every node N in T exceeds every number in its left subtree and is less than every number in its right subtree.
- Suppose the 23 were replaced by 35. Then T would still be a binary search tree.
- On the other hand, suppose the 23 were replaced by 40. Then T would not be a binary search tree, since the 38 would not be greater than the 40 in its left subtree.
- The definition of a binary search tree given in this section assumes that all the node values are distinct. No duplicacy | No redundancy
- There is an analogous definition of a binary search tree which admits duplicates, that is, in which each node N has the following property: -
- The value at N is greater than every value in the left subtree of N and is less than or equal to every value in the right subtree of N.

SEARCHING AND INSERTING IN BINARY SEARCH TREES

- Suppose T is a binary search tree.
- Let us discuss the basic operations of searching and inserting with respect to T.
- In fact, the searching and inserting will be given by a single search and insertion algorithm.
- The operation of deleting is treated separately.
- Traversing in T is the same as traversing in any binary tree.

- Suppose an ITEM of information is given.
- The following algorithm finds the location of ITEM in the binary search tree T, or inserts ITEM as a new node in its appropriate place in the tree.

(a) Compare ITEM with the root node N of the tree.

(i) If ITEM <N, proceed to the left child of N.

(ii) If ITEM>N, proceed to the right child of N.

(b) Repeat Step (a) until one of the following occurs:

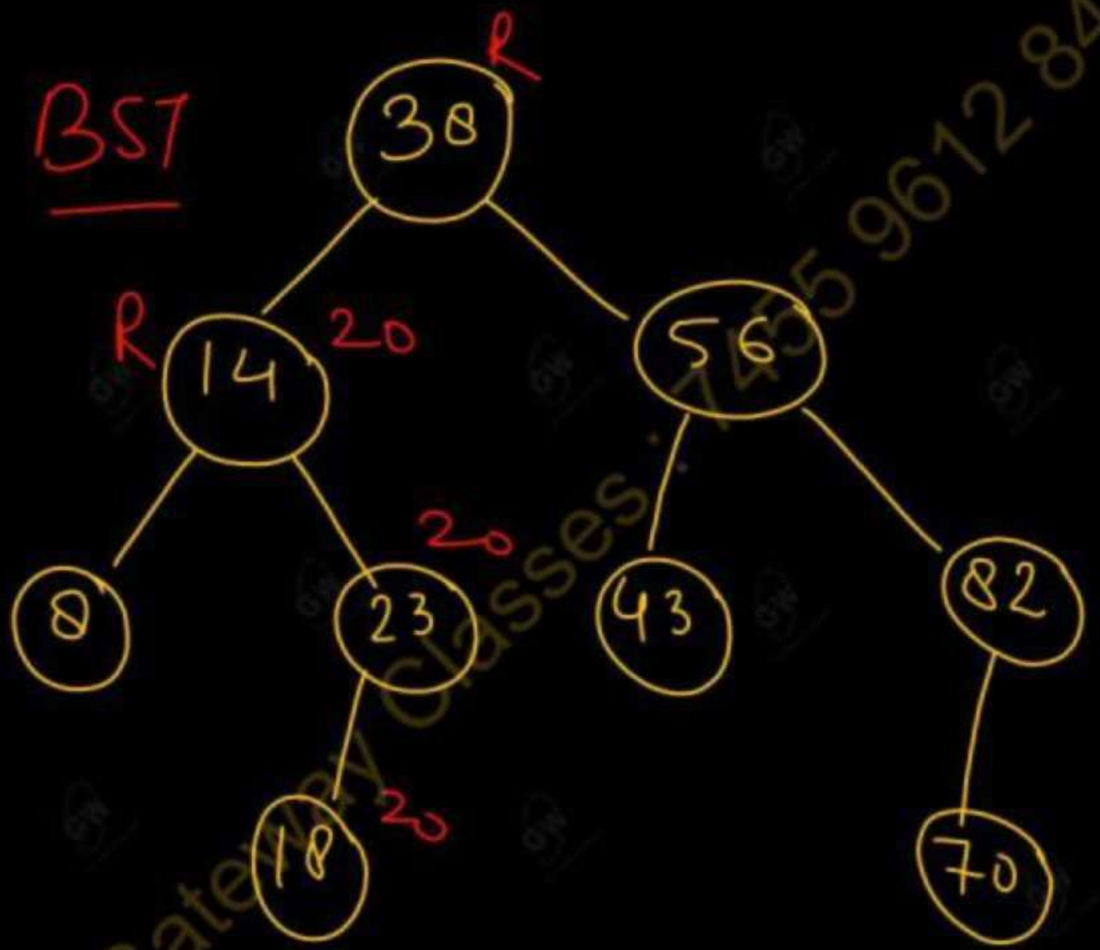
(i) We meet a node N such that ITEM N. In this case the search is successful.

(ii) We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

- In other words, proceed from the root R down through the tree T until finding ITEM in T or inserting ITEM as a terminal node in T.

EXAMPLE -

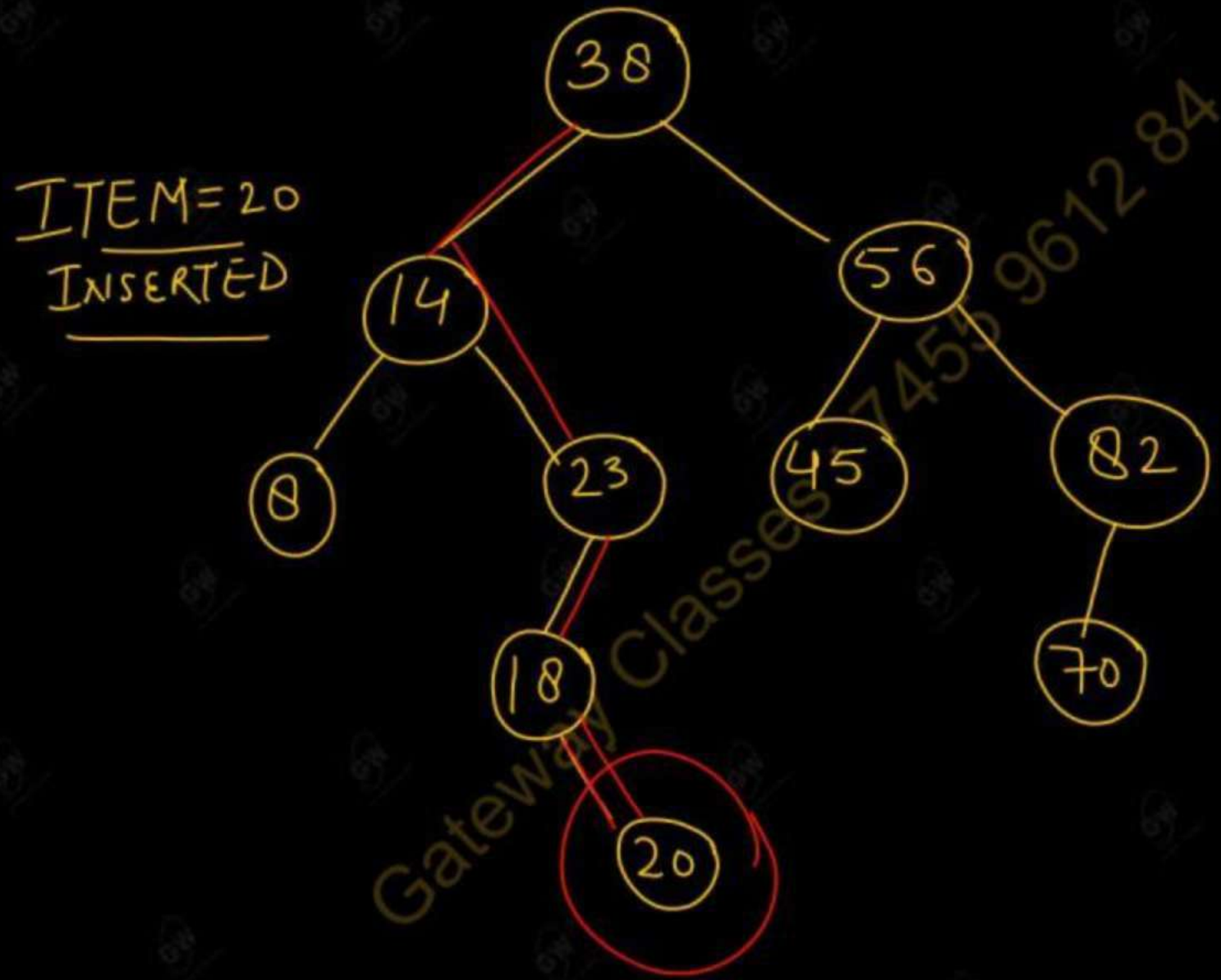
(a) Again consider the following binary search tree T.



- Suppose ITEM = 20 is given.
- Simulating the above algorithm, we obtain the following steps:

1. Compare ITEM = 20 with the root, 38, of the tree T. Since 20 < 38, proceed to the left child of 38, which is 14.
2. Compare ITEM = 20 with 14. Since 20 > 14, proceed to the right child of 14, which is 23.
3. Compare ITEM = 20 with 23. Since 20 < 23, proceed to the left child of 23, which is 18.
4. Compare ITEM = 20 with 18. Since 20 > 18 and 18 does not have a right child, insert 20 as the right child of 18.

- The next tree shows the new tree with ITEM = 20 inserted.
- The shaded edges indicate the path down through the tree during the algorithm.



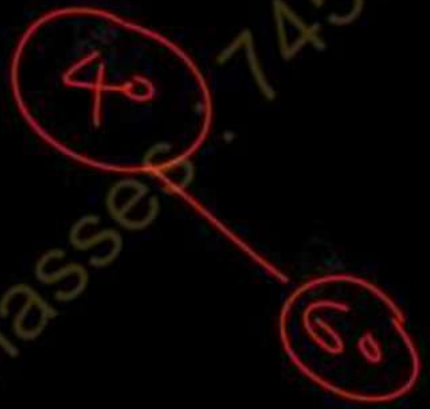
EXAMPLE :- Suppose the following six numbers are inserted in order into an empty binary search tree: 40, 60, 50, 33, 55, 11 .

Draw each step and final BST. (Pre, Post, inorder)

Solution : -

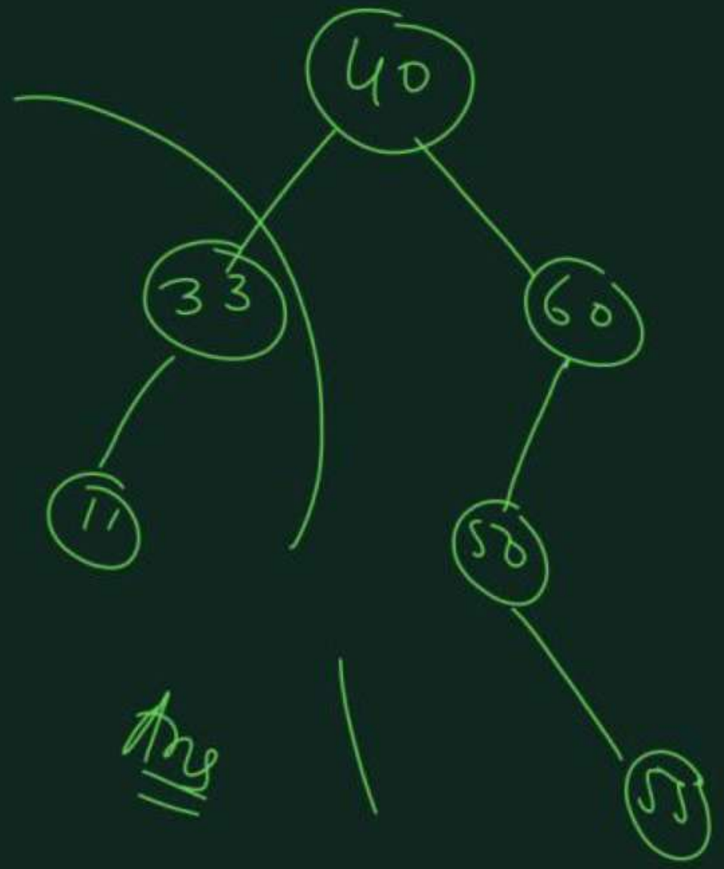
(i) item 40

(ii) item = 60



(iii) item = 50

40, 60, 50, 33, 55, 11



Ans

Preorder - Root, left, Right

40, 33, 11, 60, 50, 55

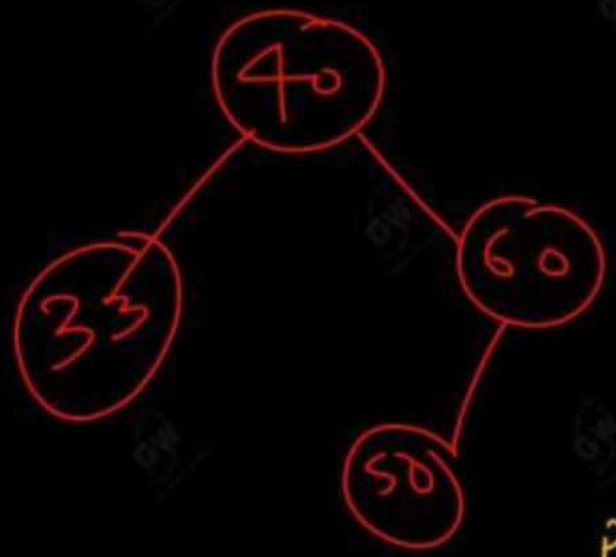
Inorder - left Root Right

11, 33, 40, 50, 55, 60

Postorder:- left Right Root

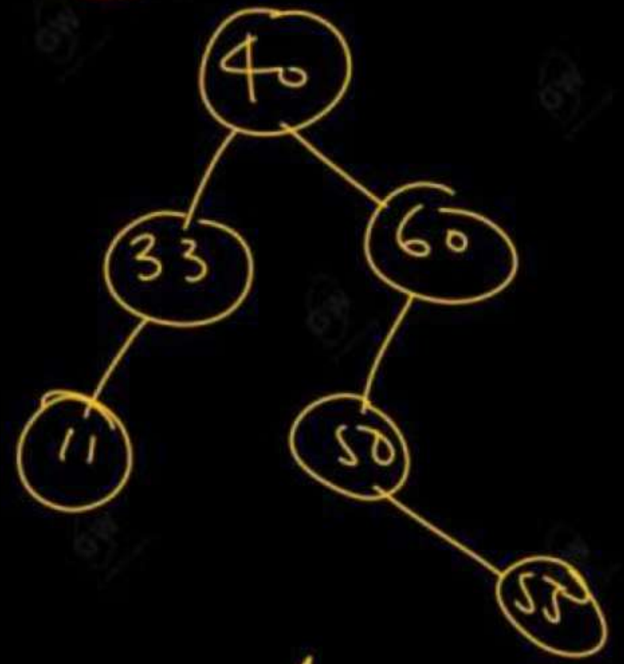
11, 33, 55, 50, 60, 40 Ans

(V) 11 = 33



(VI) item = 55

(VII) item = 11



PRE :- Root Left Right

In = Left Root Right

11, 33, 40, 50, 55, 60

40 33 11 60 50 55

Post :- Left Right Root

11, 33, 55, 50, 60, 40

Ans

- The formal presentation of our search and insertion algorithm will use the following procedure, which finds the locations of a given ITEM and its parent.
- The procedure traverses down the tree using the pointer PTR and the pointer SAVE for the parent node.

Procedure: FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

A binary search tree T is in memory and an ITEM of information is given.

This procedure finds the location LOC of ITEM in T and also the location PAR of the parent of ITEM.

There are three special cases:

(i) LOC = NULL and PAR = NULL will indicate that the tree is empty.

(ii) LOC != NULL and PAR = NULL will indicate that ITEM is the root of T.

(iii) LOC = NULL and PAR != NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.

1. [Tree empty?]

If ROOT= NULL, then: Set LOC:= NULL and PAR = NULL, and Return.

2. [ITEM at root?]

If ITEM=INFO[ROOT], then: Set LOC: = ROOT and PAR: = NULL, and Return.

3. [Initialize pointers PTR and SAVE.]

If ITEM < INFO[ROOT], then:

Set PTR = LEFT[ROOT] and SAVE:=ROOT.

Else:

Set PTR = RIGHT[ROOT] and SAVE:= ROOT.

[End of If structure.]

4. Repeat Steps 5 and 6 while PTR ≠ NULL;

5. [ITEM found?]

If ITEM=INFO[PTR], then: Set LOC: = PTR and PAR: = SAVE, and Return.

6. If ITEM < INFO[PTR], then:

Set SAVE: = PTR and PTR = LEFT[PTR].

Else:

Set SAVE: = PTR and PTR: =RIGHT[PTR]. [End of If structure.]

[End of Step 4 loop.]

7. [Search unsuccessful.] Set LOC:= NULL and PAR :=SAVE.

8. Exit.

Observe that, in Step 6, we move to the left child or the right child according to whether ITEM < INFO[PTR] or ITEM> INFO[PTR].

The formal statement of our search and insertion algorithm is as follows.

Algorithm: INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

A binary search tree T is in memory and an ITEM of information is given.

This algorithm finds the location LOC of ITEM in T or adds ITEM as a new node in T at location LOC.

1. Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR). [Procedure Find]
2. If LOC != NULL, then Exit.
3. [Copy ITEM into new node in AVAIL list.]

(a) If AVAIL = NULL, then: Write: OVERFLOW, and Exit.

(b) Set NEW: =AVAIL, AVAIL: =LEFT [AVAIL] and INFO[NEW]=ITEM.

(c) Set LOC:= NEW, LEFT[NEW]:= NULL and RIGHT|NEW] := NULL,

4. [Add ITEM to tree.]

If PAR =NULL, then: Set ROOT:= NEW

Else if ITEM < INFO[PAR], then:

Set LEFT [PAR]:= NEW

Else:

Set RIGHT [PAR]:= NEW

[End of If structure.]

5. Exit.

Observe that, in Step 4, there are three possibilities:

(1) The tree is empty,

(2) ITEM is added as a left child and

(3) ITEM is added as a right child.

DELETING IN A BINARY SEARCH TREE

- Suppose T is a binary search tree, and suppose an ITEM of information is given.
- This section gives an algorithm which deletes ITEM from the tree T
- The deletion algorithm first uses Procedure FIND to find the location of the node N which contains ITEM and also the location of the parent node P(N).
- The way N is deleted from the tree depends primarily on the number of children of node N.
- There are three cases:

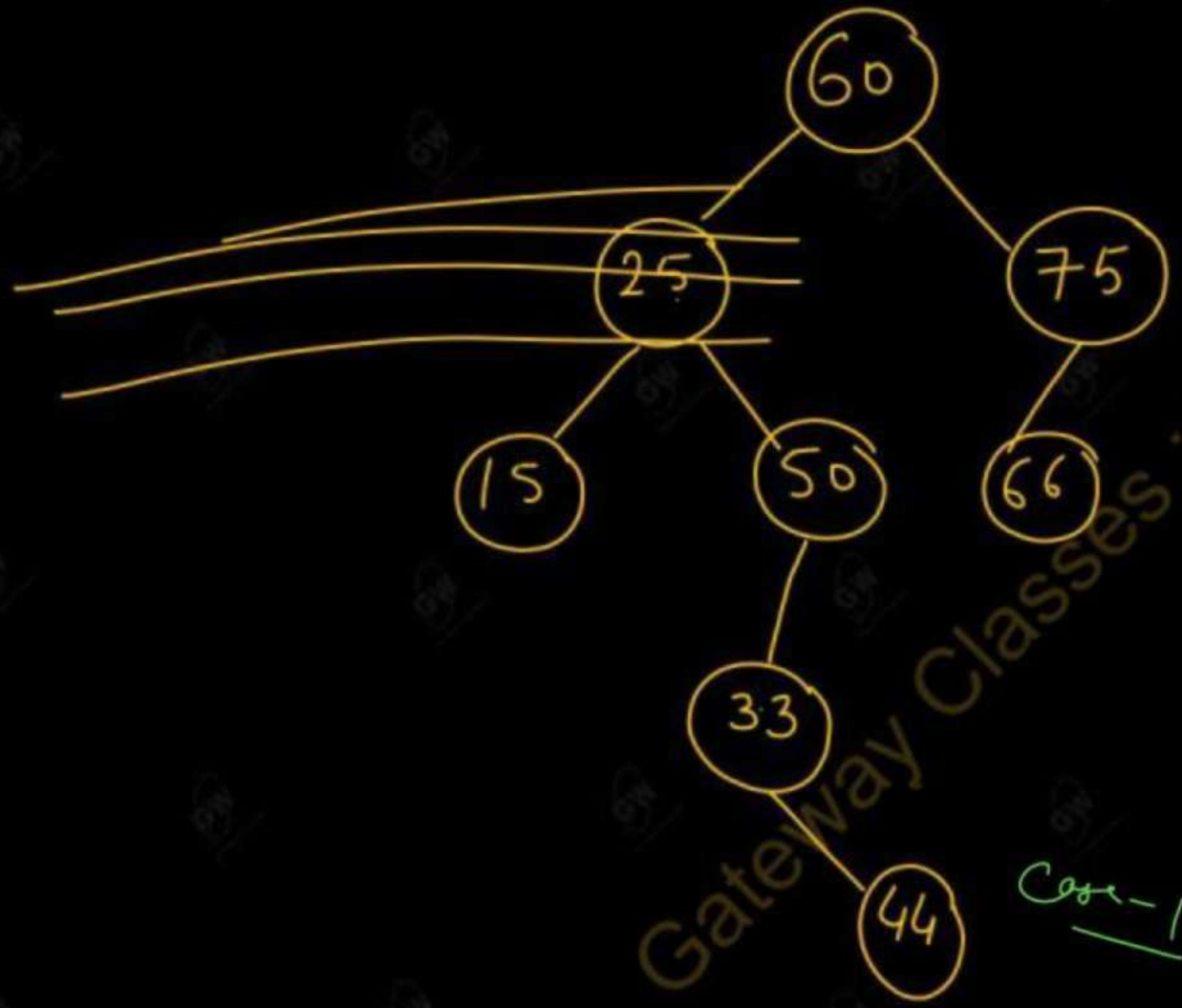
Case 1. N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the null pointer.

Case 2. N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

Case 3. N has two children. Let S(N) denote the inorder successor of N. (The reader can verify that S(N) does not have a left child.) Then N is deleted from T by first deleting S(N) from T (by using Case 1 or Case 2) and then replacing node N in T by the node S(N).

- Observe that the third case is much more complicated than the first two cases.
- In all three cases, the memory space of the deleted node N is returned to the AVAIL list.

EXAMPLE :- Consider the following binary search tree:-



BST

Inorder :-

N

L N(N) R

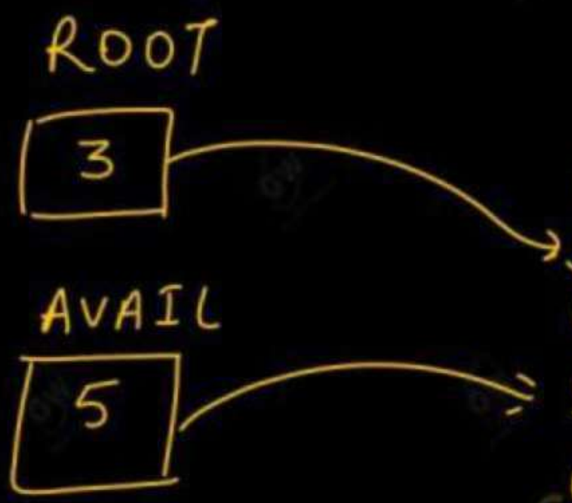
15, 25, 33, 44, 50, 60, 66, 75

Inordertraversal

15, 25, 33, 44, 50, 60, 66, 75

Case-1

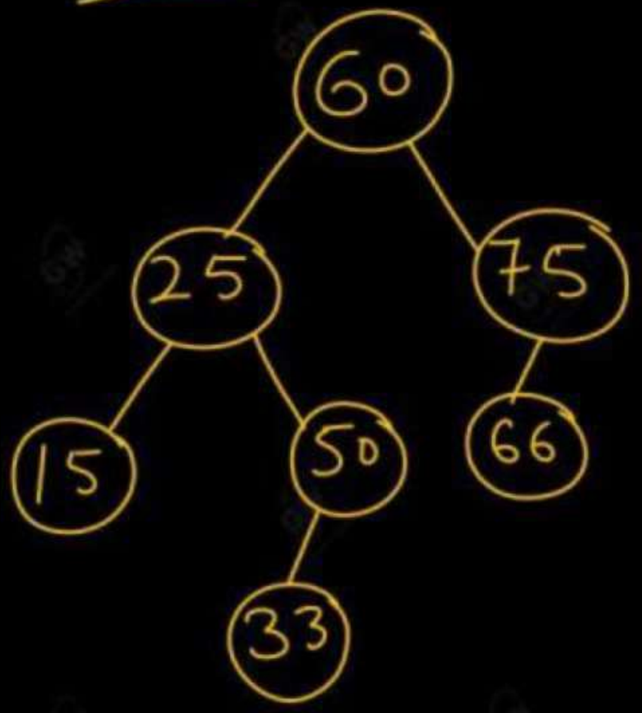
Suppose T appears in memory as per the following:



	INFO	LEFT	RIGHT
1	33	0 NULL	9
2	25	8	10
3	60	2	7
4	66	0	0 NULL
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

LINKED REPRESENTATION

(a) Suppose we delete node 44 from the previous binary search tree. Note that node 44 has no children. So after deletion 44, the binary search tree and its linked representation will be as follows:



NODE 44 is Deleted

ROOT 3

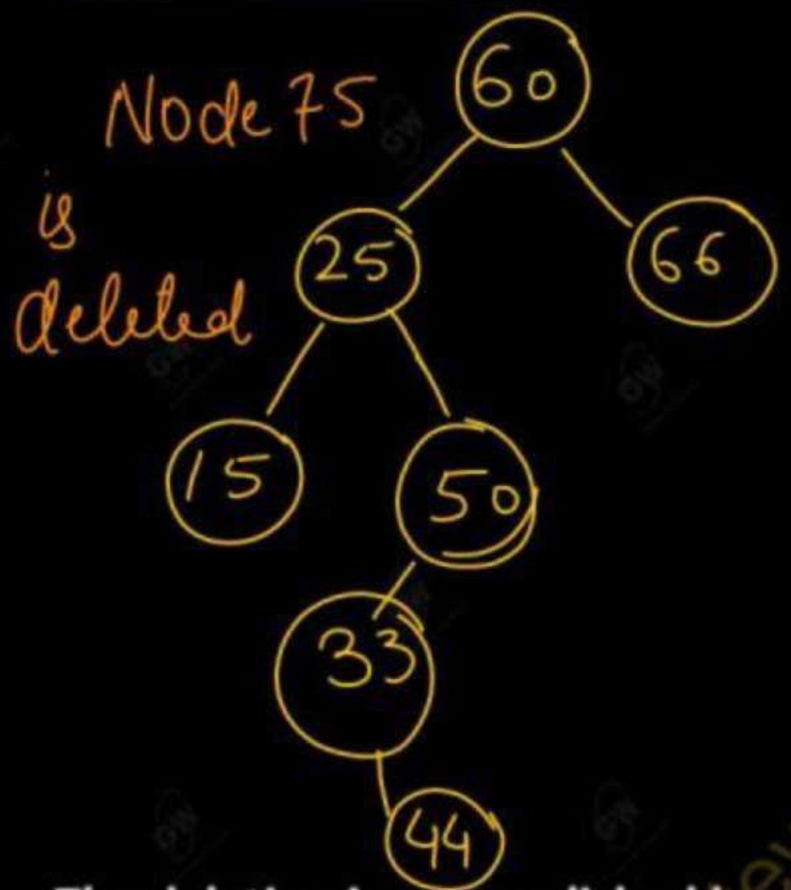
AVAIL 9

	INFO	LEFT	RIGHT
1	33	0	0
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9		5	
10	50	1	0

LINKED REPRESENTATION

The deletion is accomplished by simply assigning NULL to the parent node, 33, (The shading indicates the changes.)

(b) Suppose we delete node 75 from the previous binary tree instead of node 44. Note that node 75 has only one child. The following binary search tree pictures the tree after 75 is deleted.



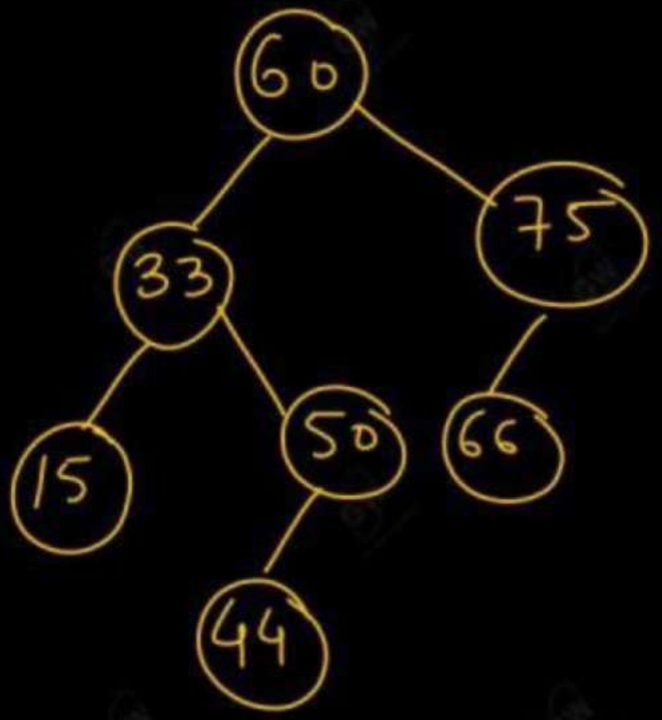
ROOT
3
AVAIL
7

	INFO	LEFT	RIGHT
1	33	0	9
2	25	0	10
3	60	2	4
4	66	0	0
5		6	
6		0 NULL	
7		5	
8	15	0	0
9	44	0	0
10	50	1	0

LINKED REPRESENTATION

The deletion is accomplished by changing the right pointer of the parent node 60, which originally pointed to 75, so that it now points to node 66, the only child of 75. (The shading indicates the changes.)

(c) Suppose we delete node 25 from the previous binary tree instead of node 44 or node 75. Note that node 25 has two children. Also observe that node 33 is the inorder successor of node 25. The following figure shows the tree after 25 is deleted with its linked representation.



Node 25 is deleted

ROOT

AVAIL

2

	INFO	LEFT	RIGHT
1	33	8	10
2		5	
3	60	1	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	9	0

LINKED REPRESENTATION

The deletion is accomplished by first deleting 33 from the tree and then replacing node 25 by node 33. We emphasize that the replacement of node 25 by node 33 is executed in memory only by changing pointers, not by moving the contents of a node from one location to another. Thus 33 is still the value of INFO[1].

- Our deletion algorithm will be stated in terms of Procedures CASEA AND CASEB.
- The first procedure refers to Cases 1 and 2, where the deleted node N does not have two children; and the second procedure refers to Case 3, where N does have two children.
- There are many subcases. which reflect the fact that N may be a left child, a right child or the root. Also, in Case 2, N may have a left child or a right child.
- Procedure CASEB treats the case that the deleted node N has two children.
- We note that the inorder successor of N can be found by moving to the right child of N and then moving repeatedly to the left until meeting a node with an empty left subtree.

Procedure CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure deletes the node N at location LOC, where N does not have two children.

The pointer PAR gives the location of the parent of N or else PAR = NULL indicates that N is the root node.

The pointer CHILD gives the location of the only child of N, or else CHILD = NULL indicates N has no children.

1. [Initializes CHILD.]

If LEFT [LOC] = NULL and RIGHT[LOC] =NULL, then:

Set CHILD: =NULL,

Else if LEFT [LOC] != NULL, then:

Set CHILD: = LEFT[LOC]

Else

Set CHILD: =RIGHT[LOC]

[End of If structure.]

2. If PAR != NULL, then:

If LOC = LEFT [PAR] then:

Set LEFT [PAR]= CHILD

Else:

Set RIGHT [PAR]: CHILD.

[End of If structure.]

Else

Set ROOT: =CHILD.

[End of If structure.]

3. Return.

Procedure CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure will delete the node N at location LOC, where N has two children.

The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node.

The pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

1. [Find SUC and PARSUC.]

(a) Set PTR = RIGHT [LOC] and SAVE: = LOC.

(b) Repeat while LEFT[PTR] != NULL:

Set SAVE:= PTR and PTR = LEFT [PTR].

[End of loop.]

(c) Set SUC:= PTR and PARSUC:= SAVE.

2. [Delete inorder successor, using Procedure CASEA]

Call CASEA (INFO, LEFT, RIGHT, ROOT, SUC, PARSUC).

3. [Replace node N by its inorder successor.]

(a) If PAR != NULL, then:

If LOC = LEFT [PAR], then:

Set LEFT[PAR]:= SUC.

Else:

Set RIGHT [PAR]:= SUC. [End of If structure.]

Else:

Set ROOT:= SUC. [End of If structure.]

(b) Set LEFT[SUC]:= LEFT[LOC] and

RIGHT[SUC]:= RIGHT[LOC].

4. Return.

We can now formally state our deletion algorithm, using Procedures CASEA and CASEB as building blocks.

Algorithm : DEL (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

A binary search tree T is in memory, and an ITEM of information is given. This algorithm deletes ITEM from the tree.

1. [Find the locations of ITEM and its parent, using Procedure FIND]

Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).

2. [ITEM in tree?]

If LOC = NULL, then: Write: ITEM not in tree, and Exit.

3. [Delete node containing ITEM.]

If RIGHT [LOC] != NULL and LEFT [LOC] != NULL, then:

Call CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR).

Else:

Call CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR). [End of If structure.]

4. [Return deleted node to the AVAIL list.]

Set LEFT [LOC] := AVAIL and AVAIL : = LOC.

5. Exit.

Example :- Suppose the following numbers are inserted in an empty Binary Search Tree -

40, 25, 70, 22, 85, 60, 80, 90, 10, 30

(i) Draw the BST, T.

(ii) Find preorder, inorder and postorder traversal of T.

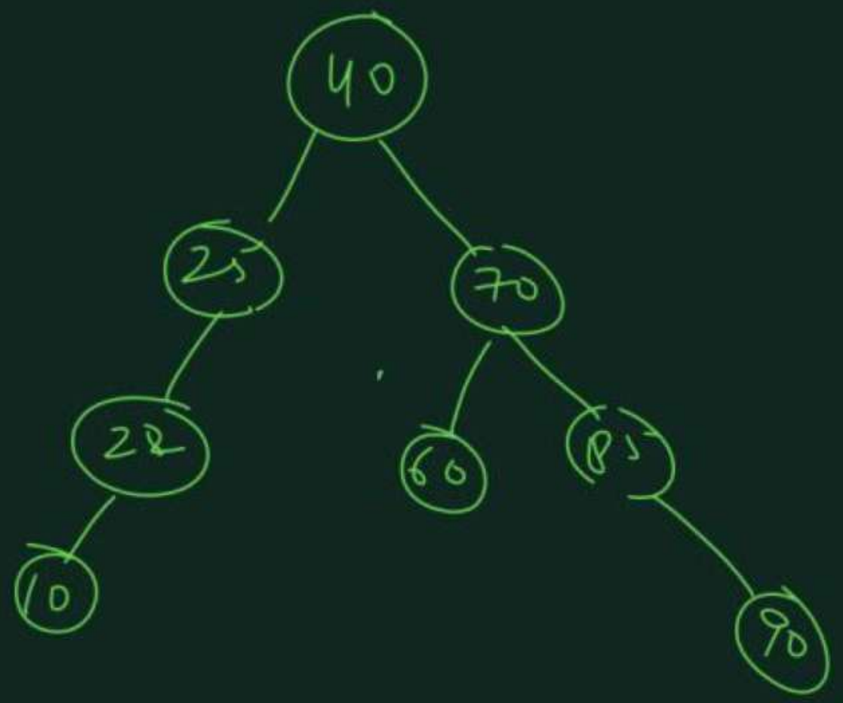
(iii) Apply the following operations on T.

(a) Delete node 30.

(b) Delete node 80.

(c) Delete node 40.

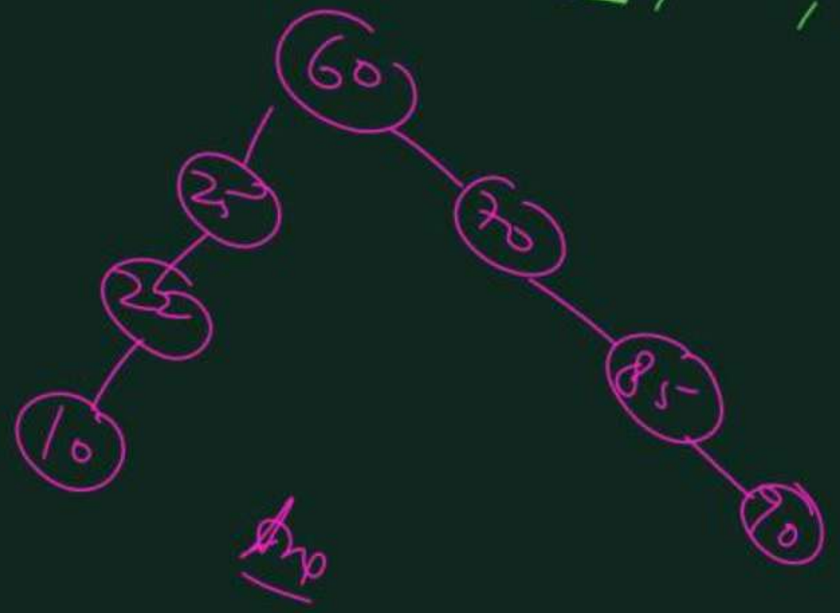
40, 25, 70, 22, 85, 60, 80, 90, 10, 30



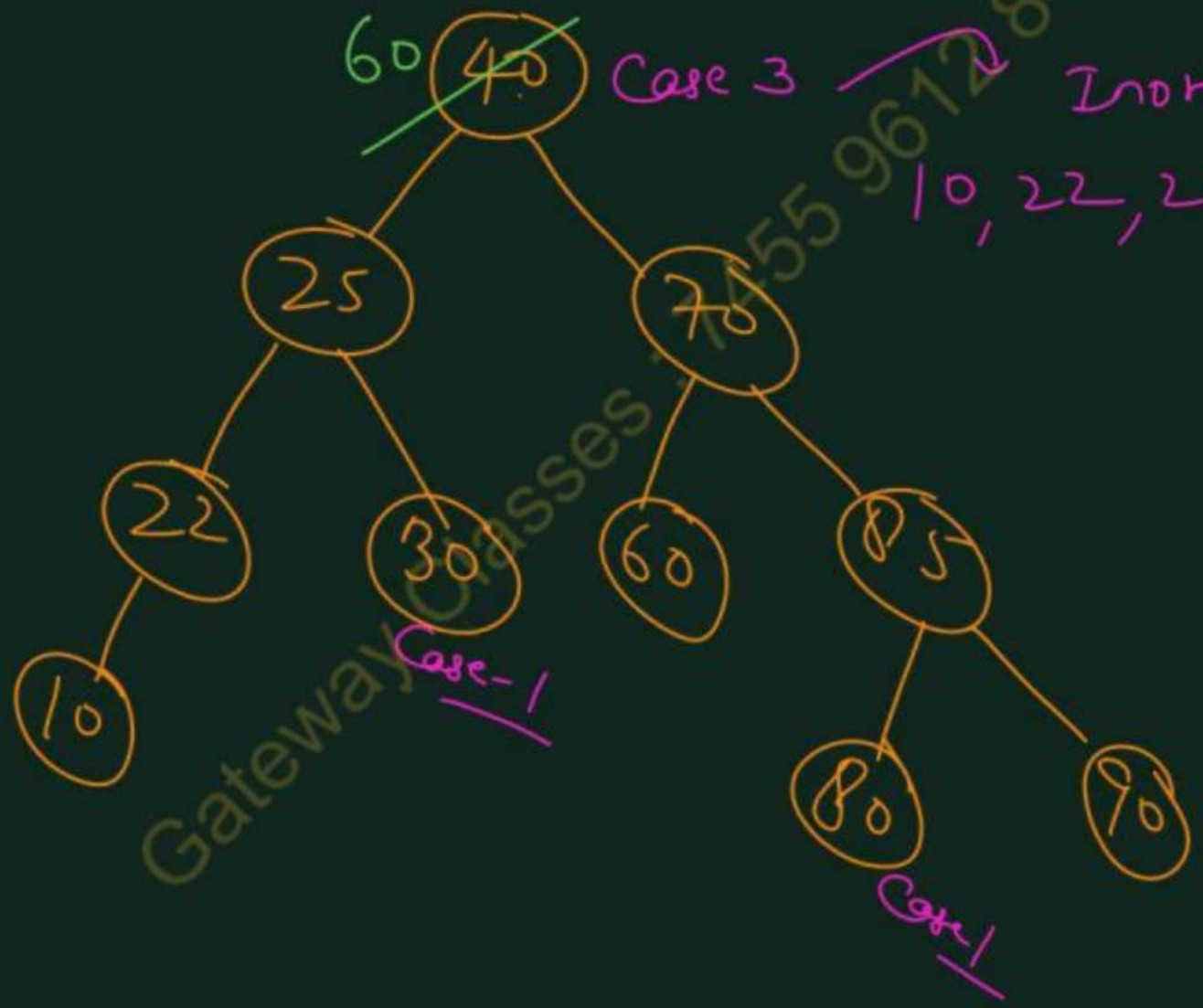
Inorder Traversal

N S(N)

10, 22, 25, 40, 60, 70, 85, 90



40, 25, 70, 22, 85, 60, 80, 90, 10, 30



Inorder

10, 22, 25, 30, 40, 60, 70, 80, 85, 90

N S(N)

PATH LENGTHS; HUFFMAN'S ALGORITHM

- We know that an extended *binary* tree or 2-*tree* is a binary tree *T* in which each node has either 0 or 2 children.
- The nodes with 0 children are called external nodes, and the nodes with 2 children are called internal nodes.
- The following tree shows a 2-tree where the internal nodes are denoted by circles and the external nodes are denoted by squares.

$L_I = 0+1+1+2+2+3+$

$= 9$

$L_E = L_I + 2n$

$= 9 + 2 \times 6$

$= 9 + 12 = 21$

$21 = 21$

$L_E = 2+2+3+4+4+3+3$

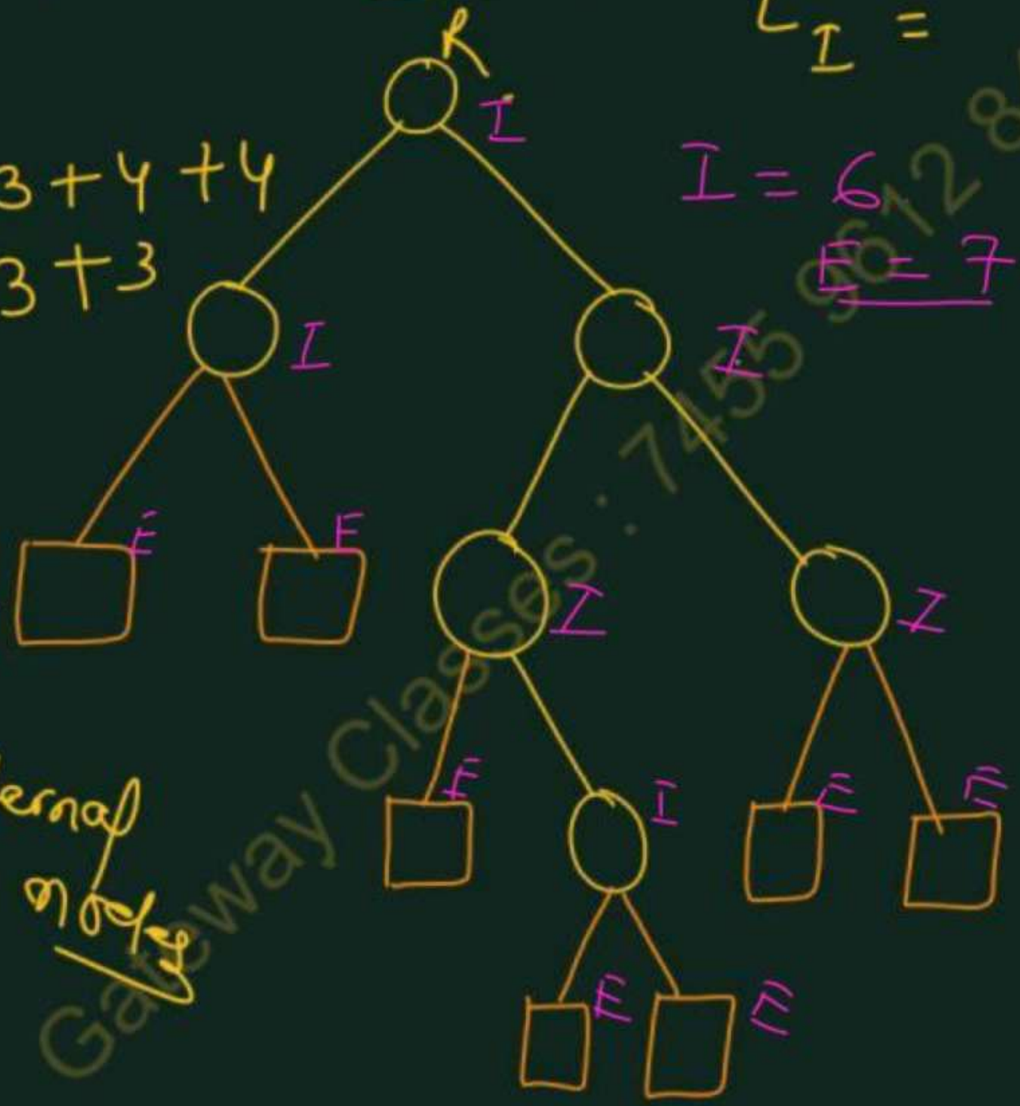
$= 21$

$L_E = \underline{2+2+3+4} + \underline{4+3+3}$

$= \underline{21}$

External nodes

2 - TREE



$L_I =$

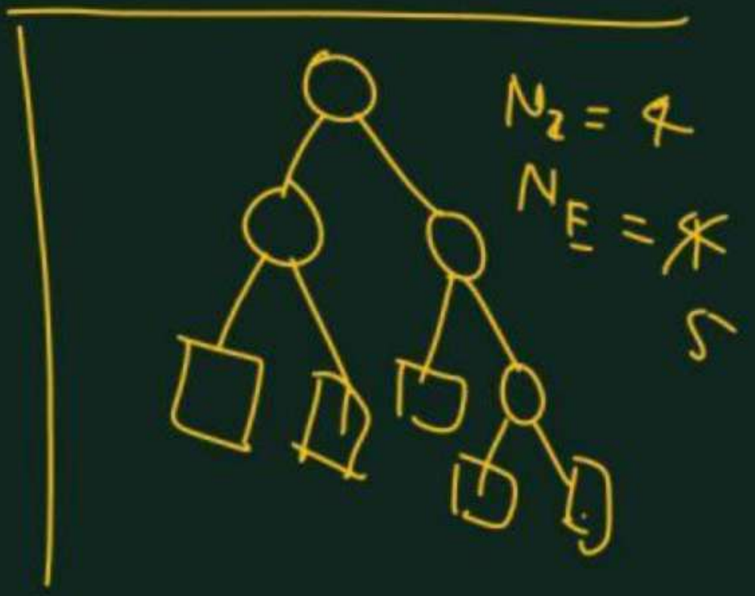
$I = 6$

$E = 7$

$N_E = N_I + 1$

$7 = 6 + 1$

$7 = 7$ Ans



- In any 2-tree, the number N_E of external nodes is 1 more than the number N_I of internal nodes, that is:

$$N_E = N_I + 1$$

- For example, for the previous 2 - tree, $N_I = 6$, and $N_E = N_I + 1 = 7$
- The running time of the algorithm may depend on the lengths of the paths in the tree.
- Therefore, we define the external *path* length L_E of a 2-tree *T* to be the sum of all path lengths summed over each path from the root *R* of *T* to an external node.
- The internal *path* length L_I of *T* is defined, using internal nodes instead of external nodes.
- For the tree in the previous figure -
  - External Path Length $L_E = 2 + 2 + 3 + 4 + 4 + 3 + 3 = 21$ and
  - Internal Path Length $L_I = 0 + 1 + 1 + 2 + 3 + 2 = 9$

- Observe that $L_I + 2n = 9 + 2.6 = 9 + 12 = 21 = L_E$

where n = 6 is the number of internal nodes. In fact, the formula

$$L_E = L_I + 2n$$

is true for any 2-tree with n internal nodes.

$L_E = 21$

$L_I = 9$

$21 = 9 + 2 \times 6$

$21 = 9 + 12$

$21 = 21$

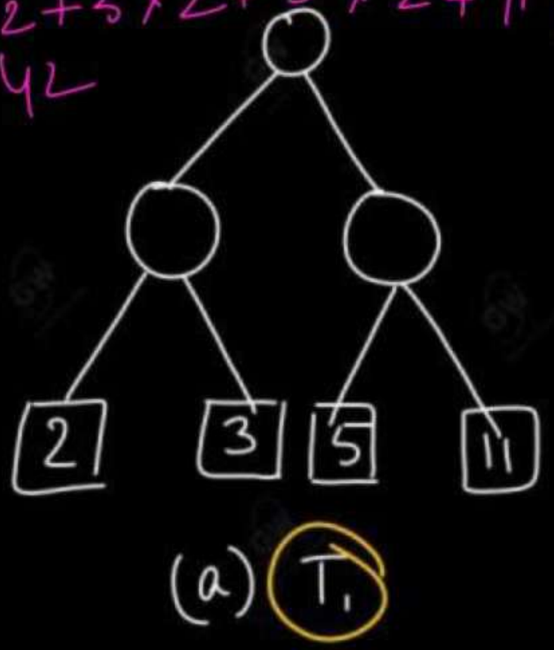
- Suppose T is a 2-tree with n external nodes, and suppose each of the external nodes assigned a is (nonnegative) weight.
- The (external) weighted path length P of the tree T is defined to be the sum of the weighted path lengths; i.e..

$$P = W_1L_1 + W_2L_2 + \cdots W_nL_n$$

where W_1, and L_1 denote, respectively, the weight and path length of an external node N_1.

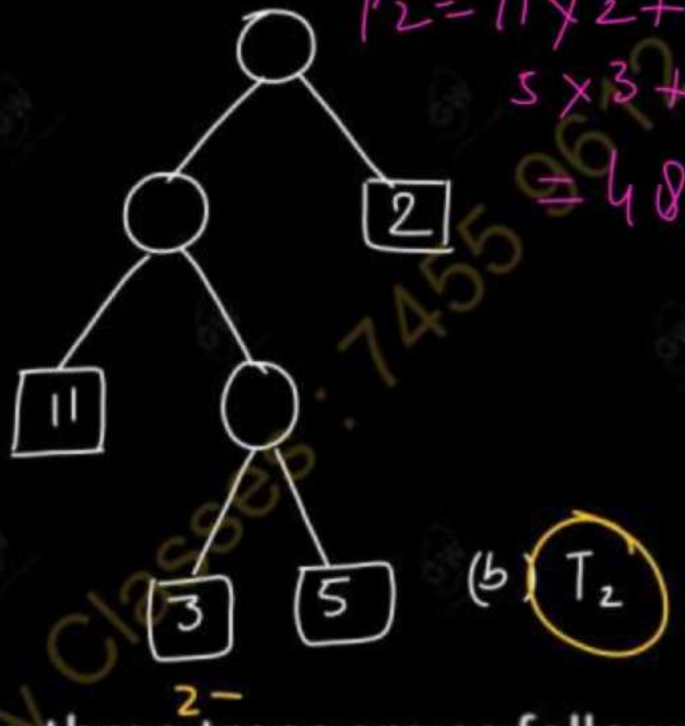
EXAMPLE : Consider following figures for three 2-trees, T_1, T_2 and T_3, each having external nodes with weights 2, 3, 5 and 11.

$P_1 = 2\times2+3\times2+5\times2+11\times2$
$= 42$

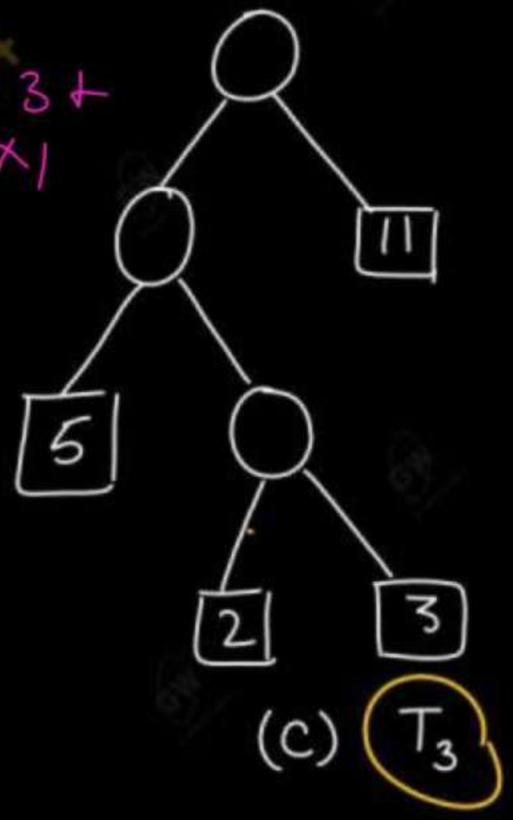


(a) T_1

$P_2 = 11\times2+3\times3+5\times3+2\times1$
$= 48$



(b) T_2



(c) T_3

The weighted path lengths of the three trees are as follows:

$$P_1 = 2 * 2 + 3 * 2 + 5 * 2 + 11 * 2 = 42$$

$$P_2 = 11 * 2 + 3 * 3 + 5 * 3 + 2 * 1 = 48$$

$$P_3 = 5 * 2 + 2 * 3 + 3 * 3 + 11 * 1 = 36$$

- The general problem that we want to solve is as follows.
- Suppose a list of n weights is given:

$$W_1 , W_2 , \ldots \ldots \ldots \ldots , W_n$$

- Among all the 2-trees with n external nodes and with the given a weights, find a tree T with a minimum - weighted path length.
- Huffman gave an algorithm, to find such a tree T.

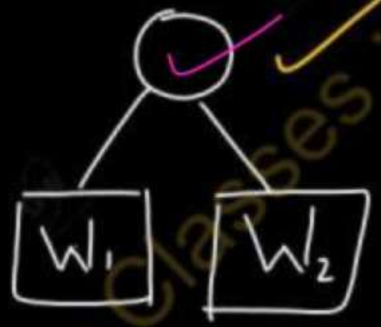
a tree with min weighted Path length.

Huffman's Algorithm: Suppose W_1 and W_2 are two minimum weights among the n given weights $W_1, W_2 \ldots W_n$. Find a tree T which gives a solution for the n-1 weights

$$W_1 + W_2, W_3, W_4 \ldots W_n$$

Then, in the tree T', replace the external node

$W_1 + W_2$ by the subtree



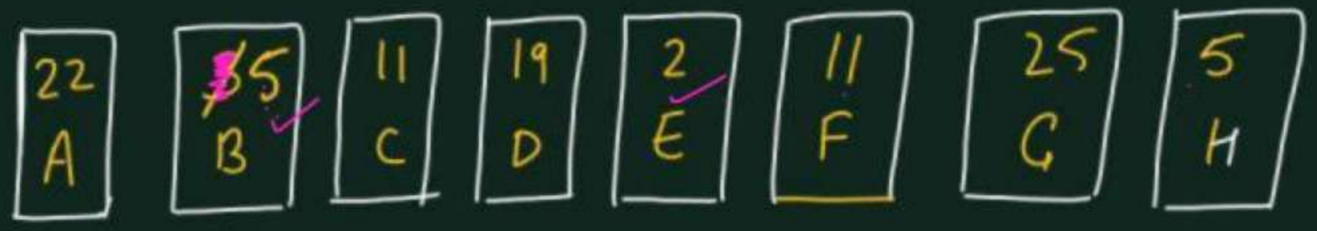
The new 2-tree T is the desired solution.

EXAMPLE : -

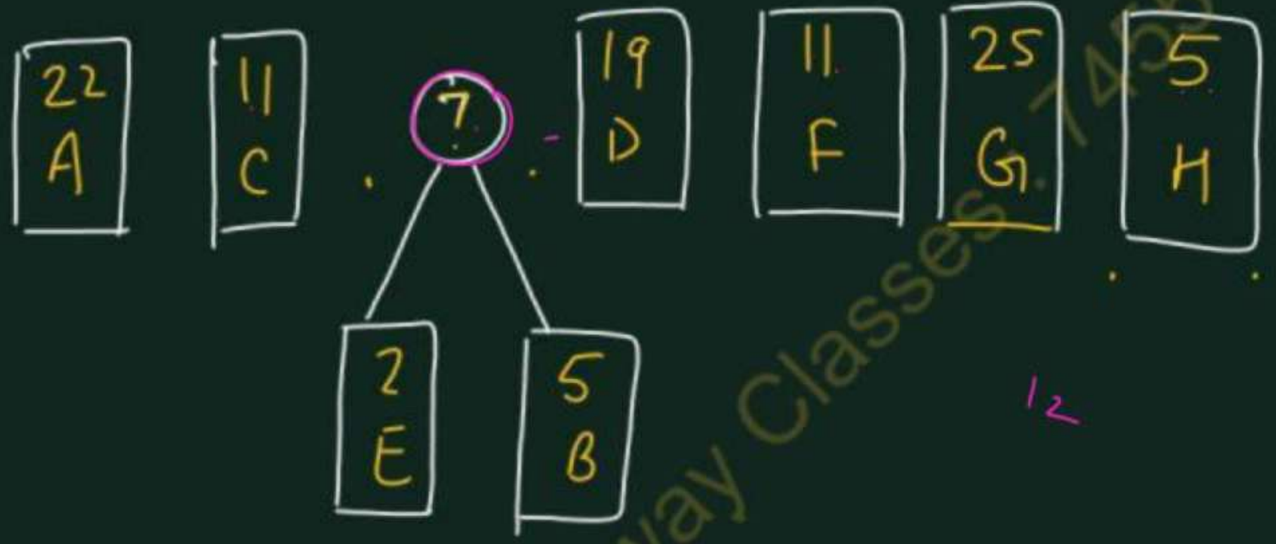
Suppose A, B, C, D, E, F, G and H are 8 data items, and suppose they are assigned weights as follows:

Data item :	A	B	C	D	E	F	G	H
Weight	22	5	11	19	2	11	25	5

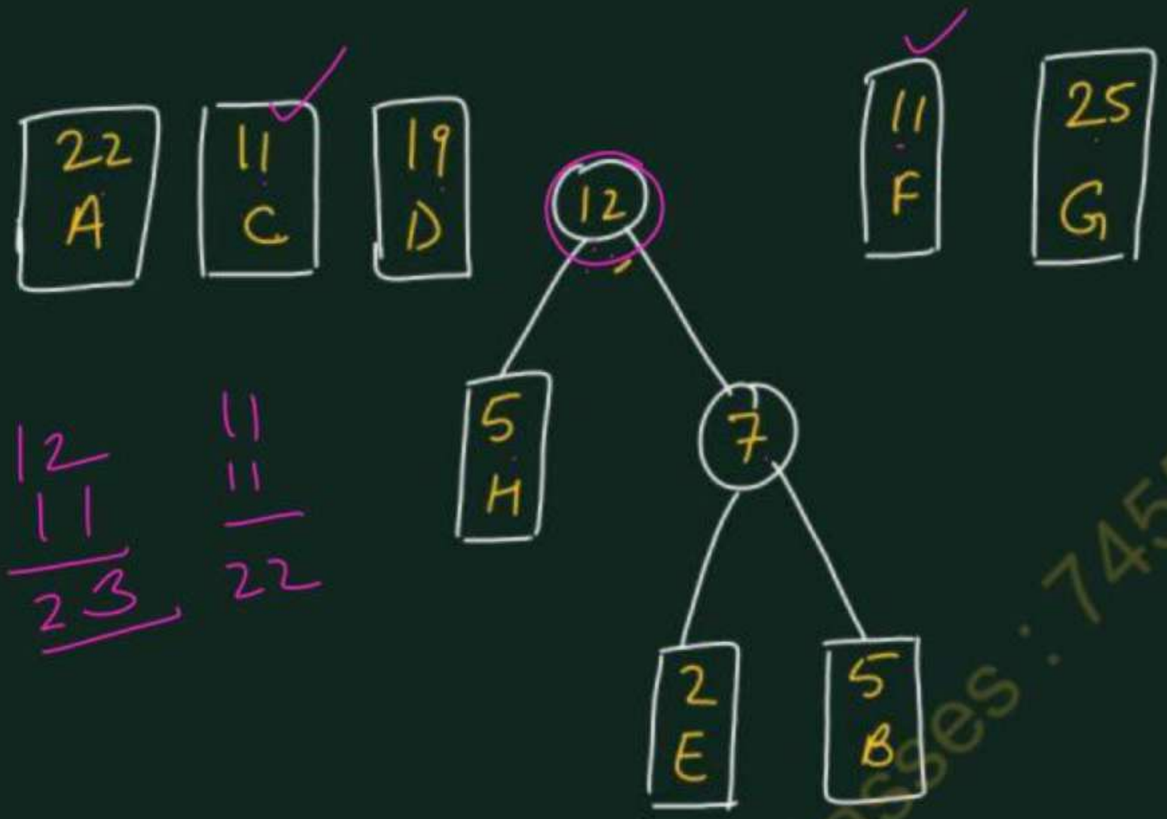
The following figures from (a) through (h) shows how to construct the tree T with minimum-weighted path length using the above data and Huffman's algorithm.



(a)



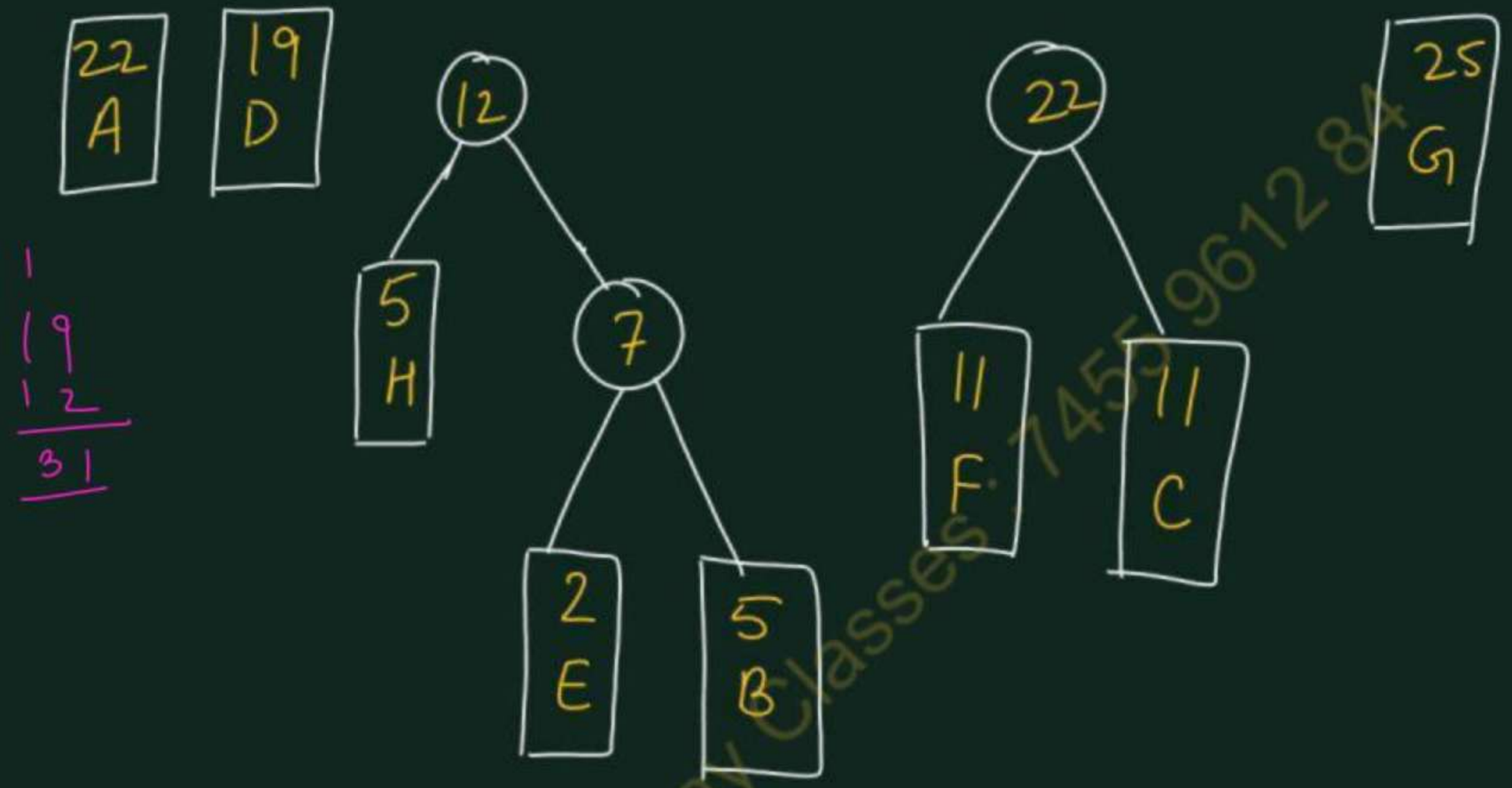
b)



12
11
23

11
11
22

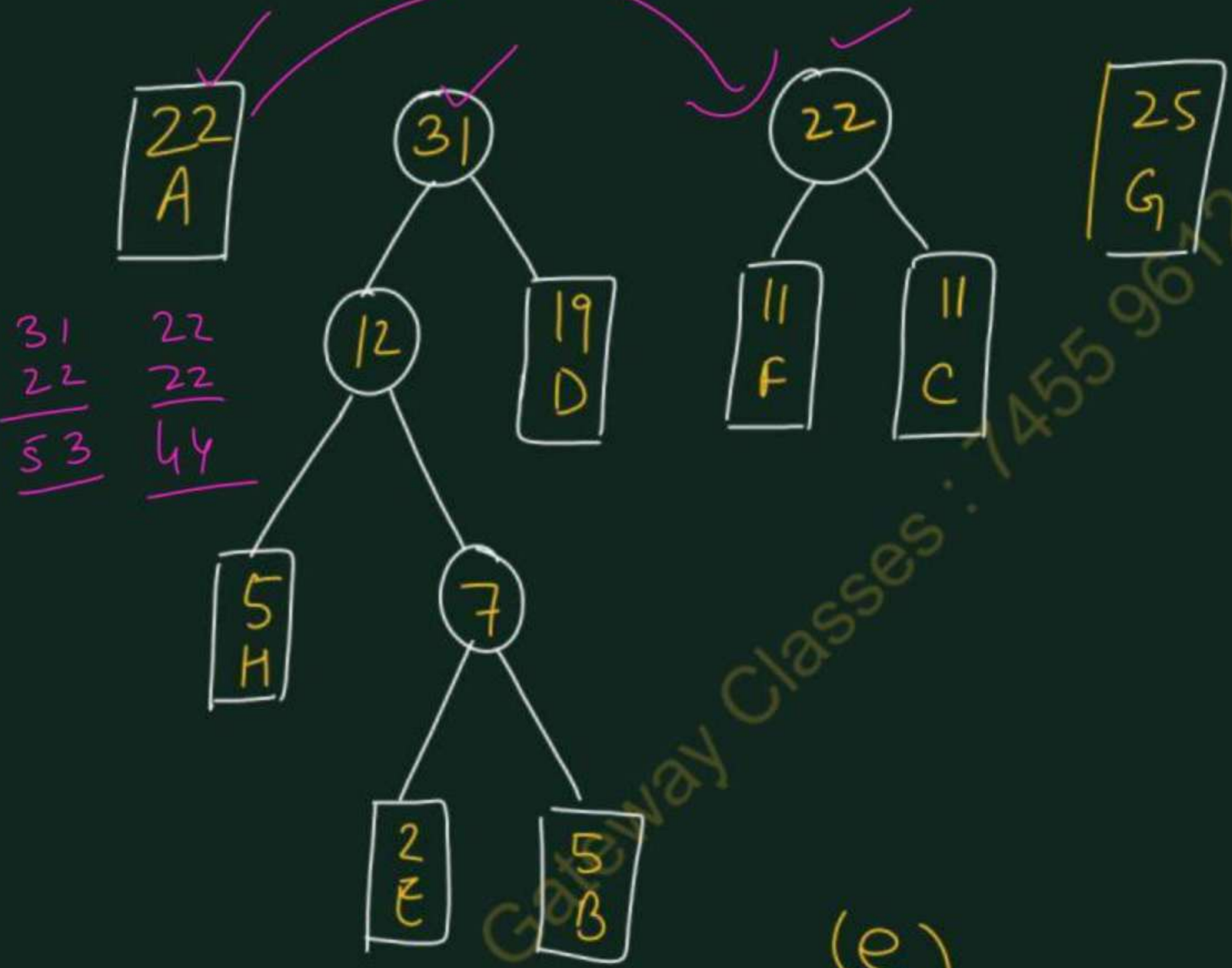
(c)



1
19
12
31

12
22
34
12
19
31

(d)

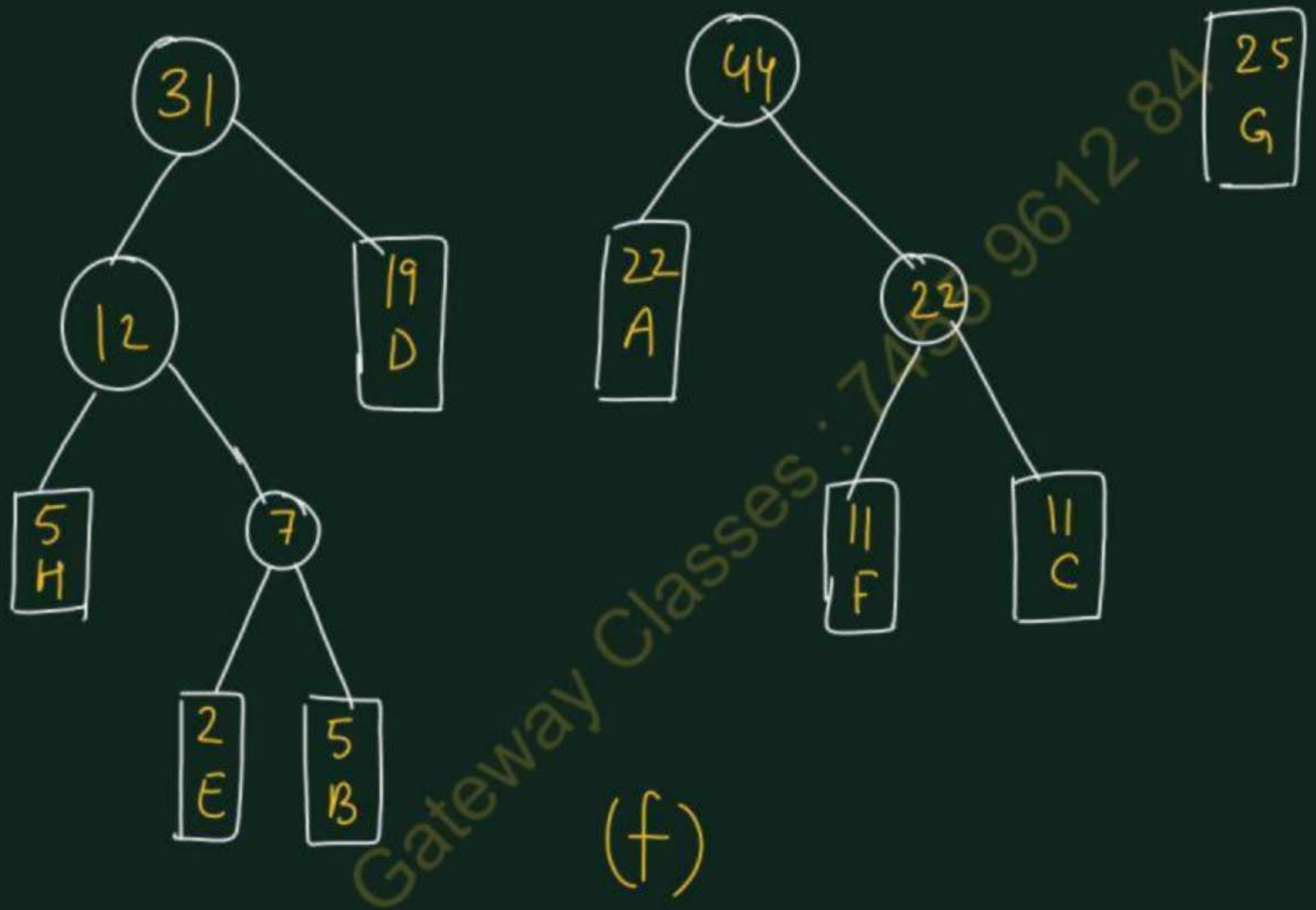


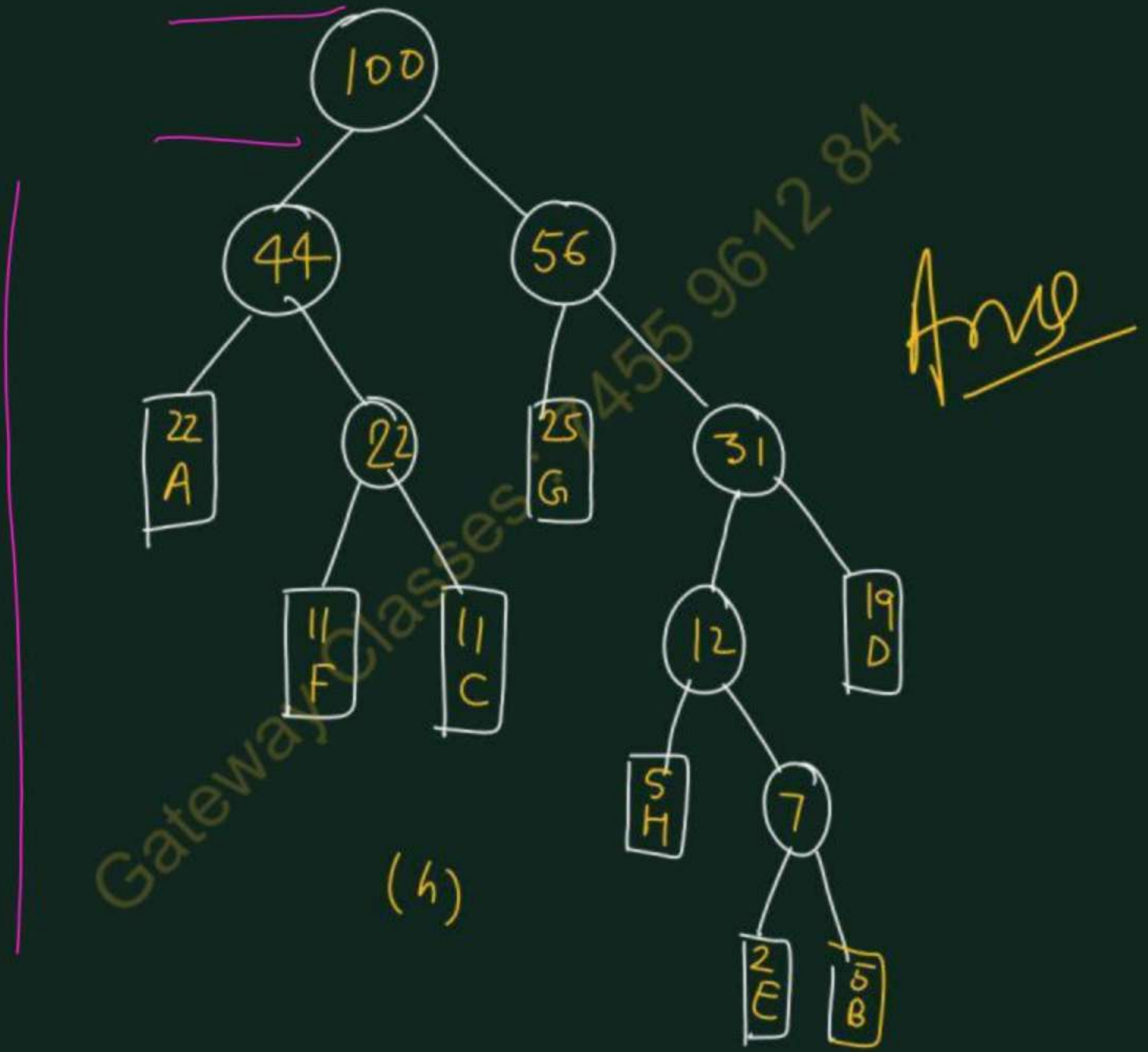
(e)

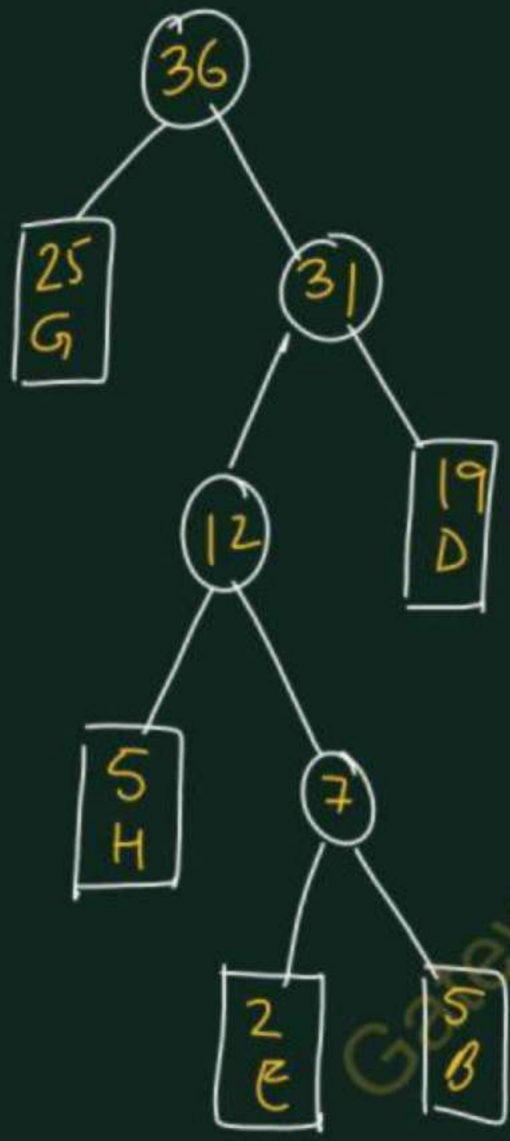
31
22
53

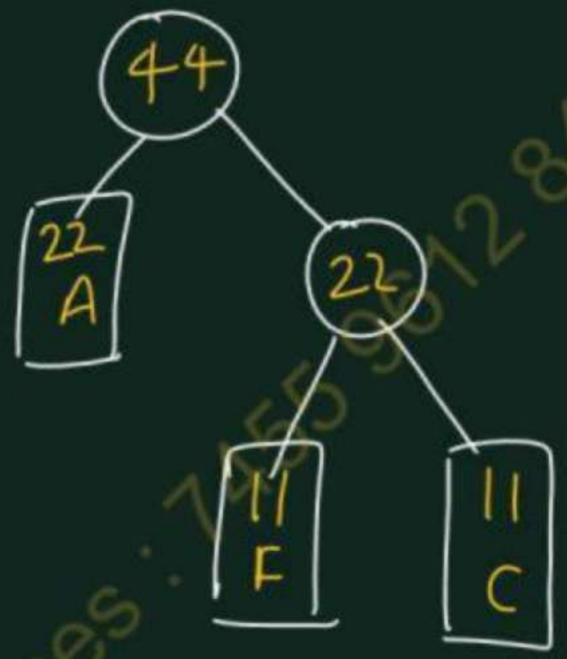
22
25
47

22
22
44









(g)

Question:- What happens if the binary search tree is left-oriented or right-oriented? Explain the problem and give the solution.

left skewed skewed

Solution:-

- In a binary search tree (BST), the orientation of the tree refers to the arrangement of nodes concerning their parent nodes.
- The terms "left-oriented" and "right-oriented" indicate the direction in which child nodes are positioned concerning their parent nodes.

Left-Oriented Binary Search Tree:

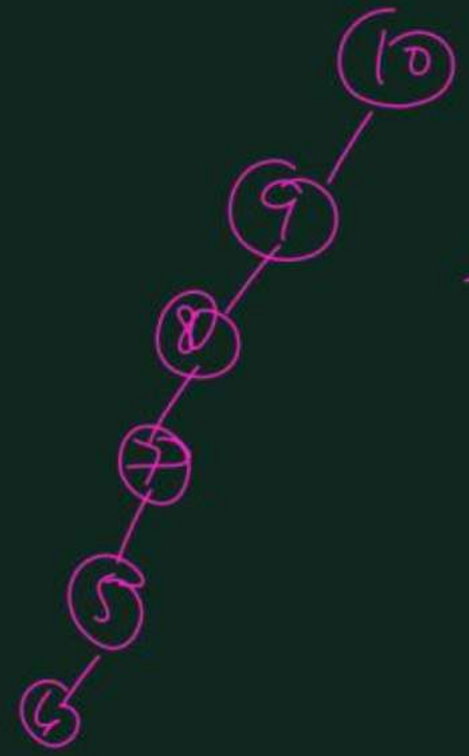
Problem:

- In a left-oriented BST, each node has only a left child, and there are no right children.
- This leads to an imbalanced tree where the height of the tree could be significantly greater on the left side compared to the right side.

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

right skewed BST

10 9 8 7 5 4 3 2 1

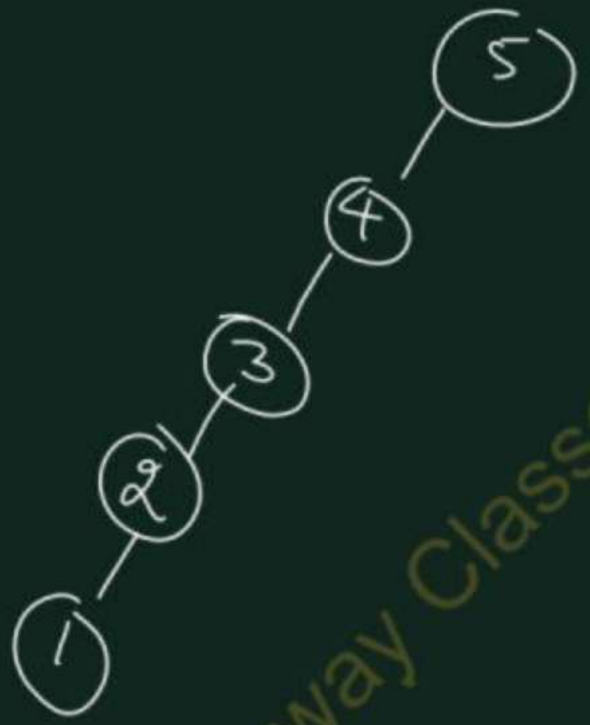


left skewed BST

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

right oriented
Binary search Tree
or
Binary ordered tree

5, 4, 3, 2, 1



left ori - BST

- As a result, the time complexity of basic operations like search, insert, and delete can degrade to O(n) in the worst case, where n is the number of nodes.

Solution:

- To maintain the efficiency of a BST, it is crucial to keep the tree balanced.
- Various balancing techniques, such as AVL trees or Red-Black trees, can be employed to ensure that the height of the tree remains logarithmic, providing efficient search, insert, and delete operations (O(log n)).

Right-Oriented Binary Search Tree:

Problem:

- Similar to a left-oriented tree, a right-oriented BST has nodes with only right children, leading to an imbalanced tree with a skewed height on the right side.
- This can also result in inefficient search and other operations.

Solution:

- The solution is again to maintain balance in the tree.
- Balancing techniques like AVL trees or Red-Black trees can be applied to ensure that the tree remains balanced, and the time complexity of operations remains efficient.
- The main issue with left-oriented or right-oriented binary search trees is that they can become highly unbalanced, leading to poor performance in terms of time complexity for basic operations.
- AVL trees, Red-Black trees, and other self-balancing binary search tree structures are commonly used to address these issues and maintain the desired logarithmic time complexity.

AVL Tree / HBT

- The first balanced binary search tree was the AVL tree (named after its discoveries, Adelson Velskii and Landis). Balanced BST
- The AVL tree is a binary search tree that has an additional balance condition.
- The idea is to require that the left and right sub-trees have the same height.
- Recursion dictates that this idea applies to all nodes in the tree, since each node is itself a root of some sub-tree.
- This balance condition ensures that the depth of the tree is logarithmic, but it is too restrictive because it is too difficult to insert new items while maintaining balance.
- Thus the AVL trees uses a notion of balance that is somewhat weaker but still strong enough to guarantee logarithmic depth.

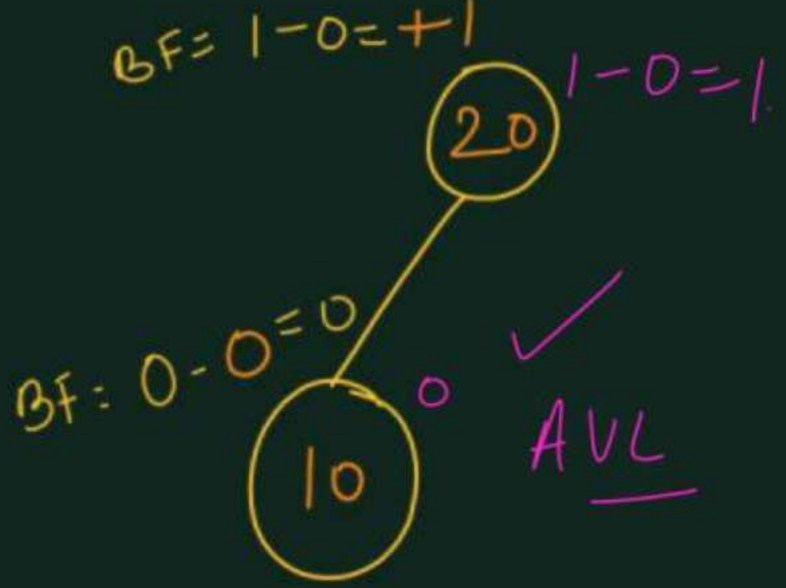
BALANCE FACTOR

search

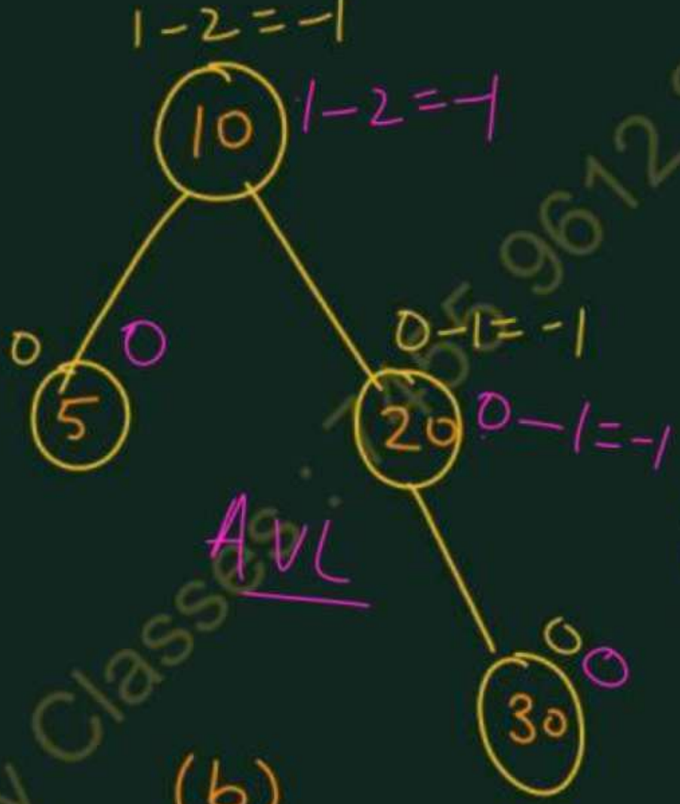
- To implement an AVL tree each node must contain a balance factor, which indicates its states of Balance relative to its sub-trees.
- If balance is defined by -

  Balance Factor = Height of left sub-tree - Height of right sub-tree = { -1 , 0, 1}

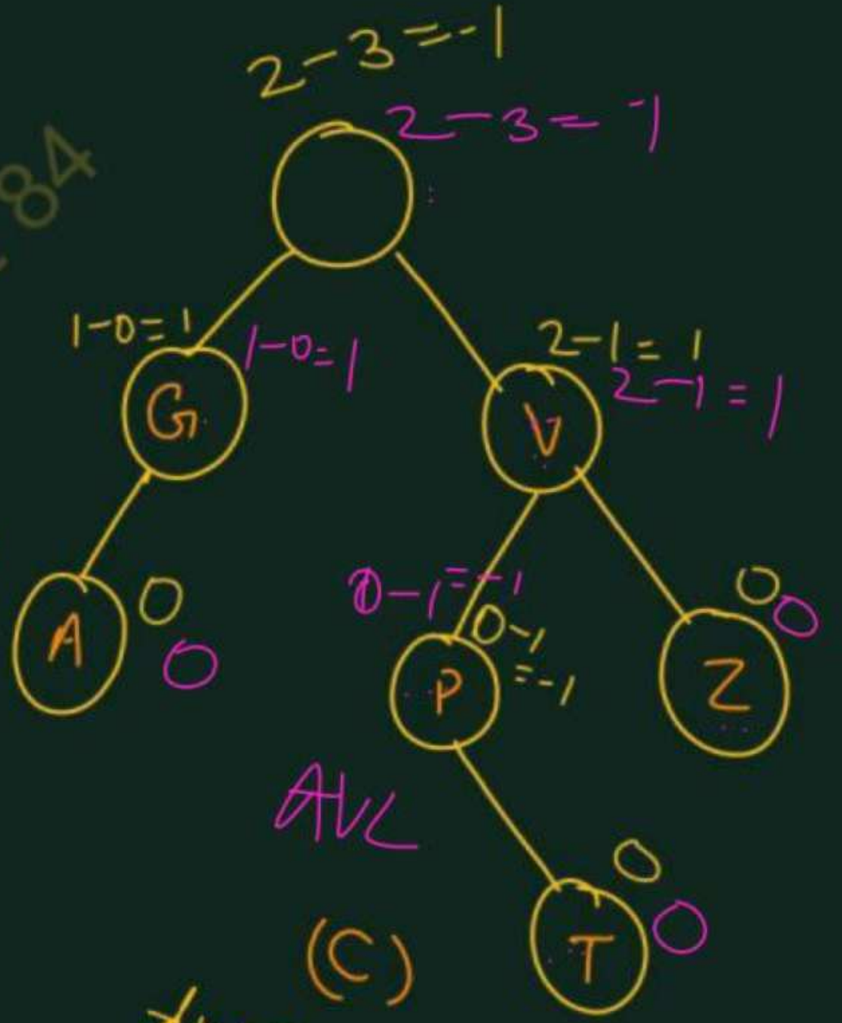
- Then the balance factors in a balanced tree can have values of -1, 0 or 1.
- In figure, numbers placed near the nodes indicate balance factors.



(a) AVL & Search Tree



(b) AVL



(C) AVL Tree

- For an AVL tree, the value of the balance factor of any node is -1, 0, or 1.
- If it is other than these three values then the tree is not balanced or it is not an AVL tree.
- If the value of the balance factor of any node is -1, then the height of the right sub-tree of that node is one more than the height of its left sub-tree.
- If the value of the balance factor of any node is 0 then the height of its left and right sub-tree is exactly the same.
- If the value of the balance factor of any node is 1 then the height of the left subtree of that node is one more than the height of its right sub-tree.

a, b ⇒ single

AVL Tree Rotations Insertion

(4)

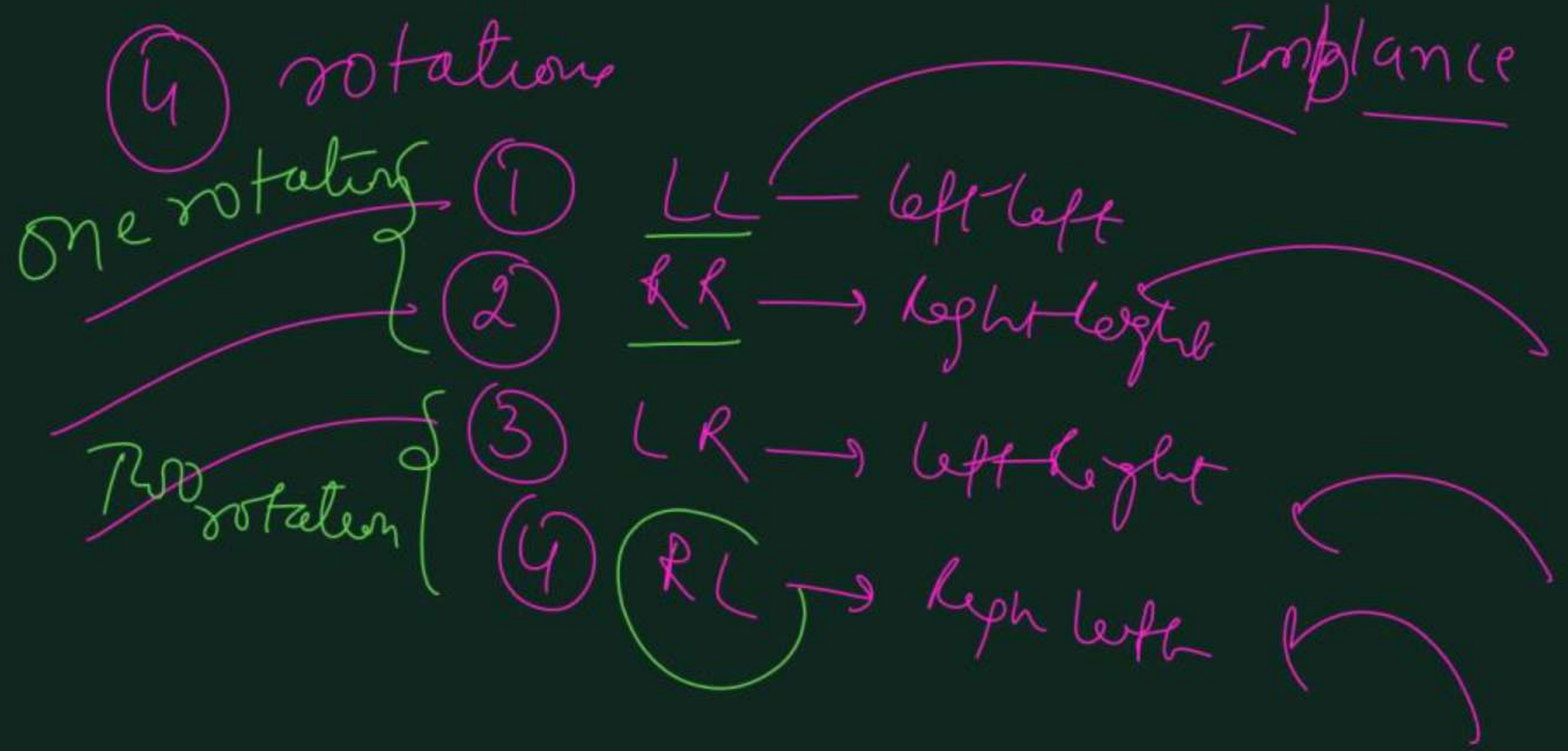
In order to balance a tree, there are four cases of rotations, such as:

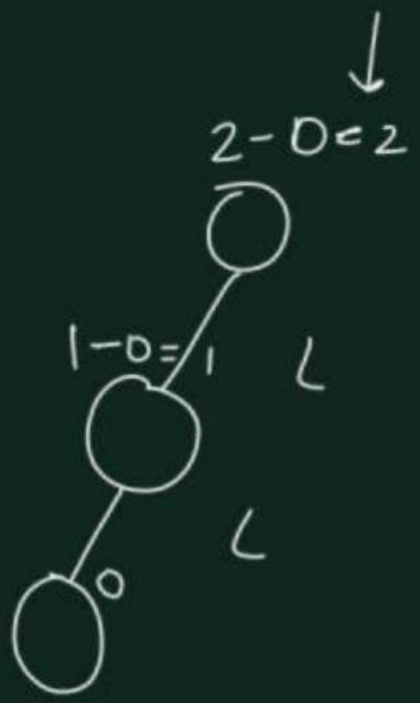
a) **LL Rotation :** Inserted node is in the left sub-tree of left sub-tree of node.

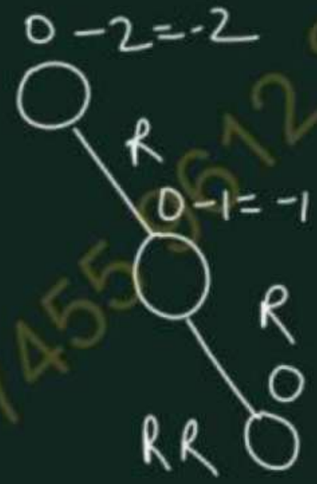
b) **RR Rotation :** Inserted node is in the right sub-tree of right sub-tree of node.

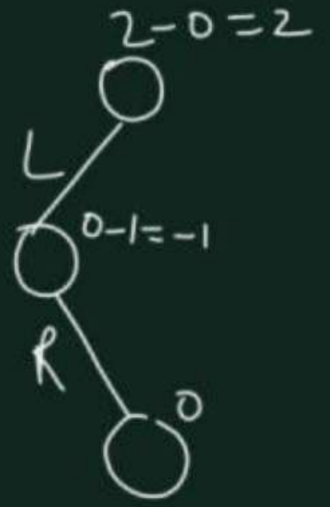
c) **LR Rotation :** Inserted node is in the right sub-tree of left sub-tree of node.

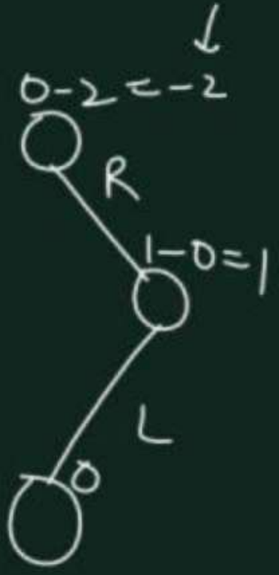
d) **RL Rotation :** Inserted node is in the left sub-tree of right sub-tree of node.


4 rotations
Imbalance
one rotation
2 rotation
1
LL
left left
2
RR
right right
3
LR
left right
4
RL
right left



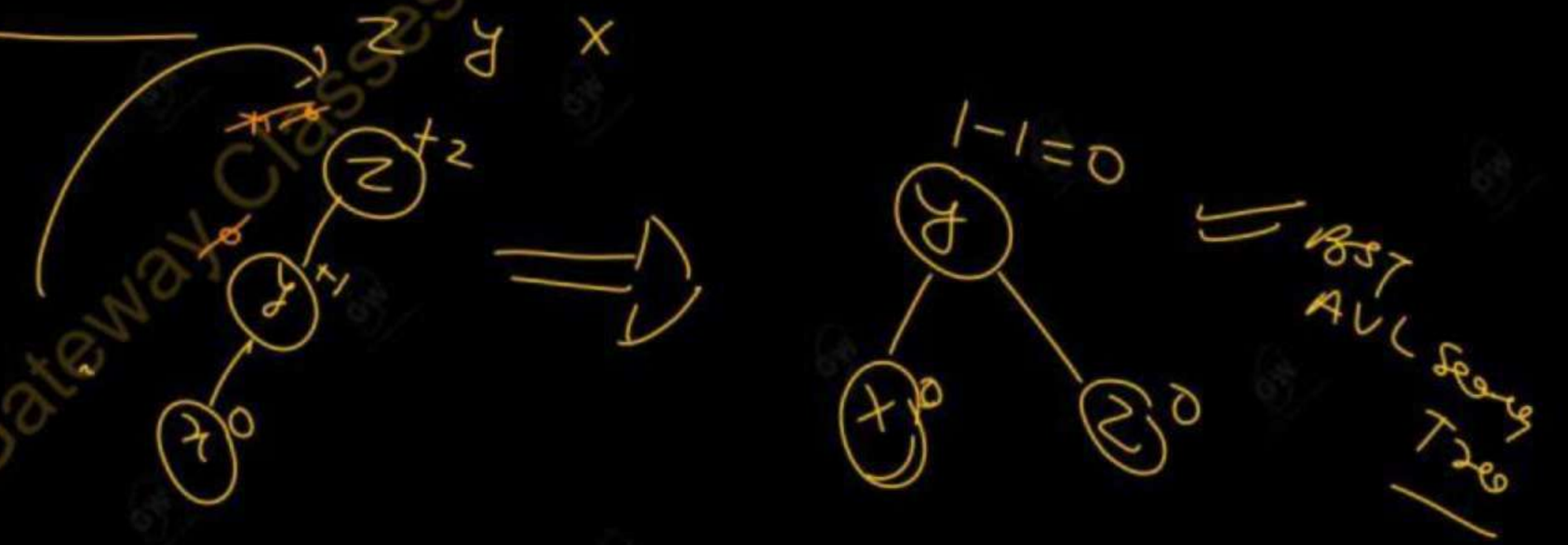


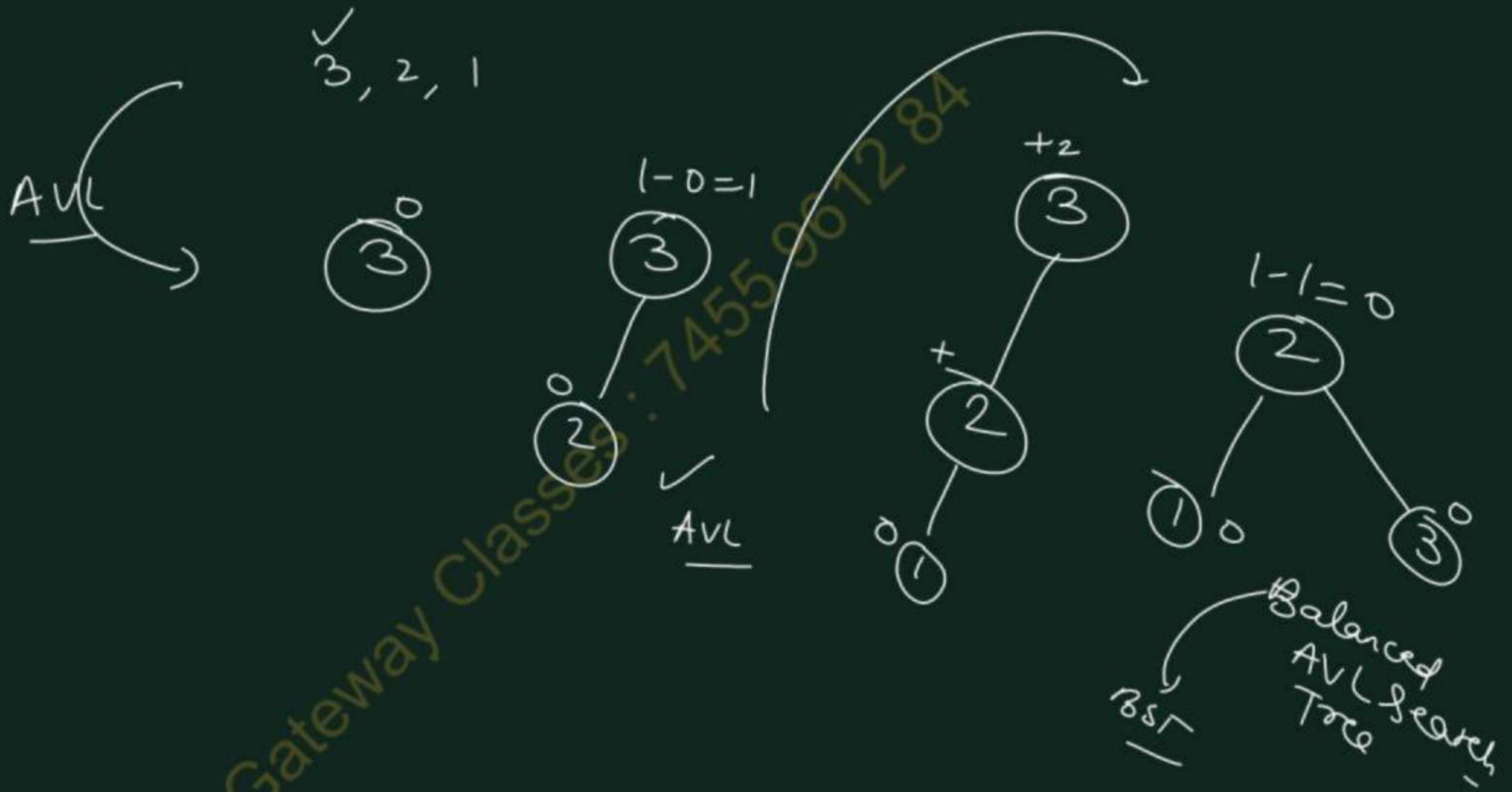




(a) LL Rotation:

- The new node x is inserted in the left sub-tree of left sub-tree of A whose balance factor becomes +2 after insertion.
- To rebalance the search tree, it is rotated so as to allow B to be the root with B_L and A as B_L is the left sub-tree and A is the right sub-tree and B_R and A_R to be the left and right sub-trees of A.





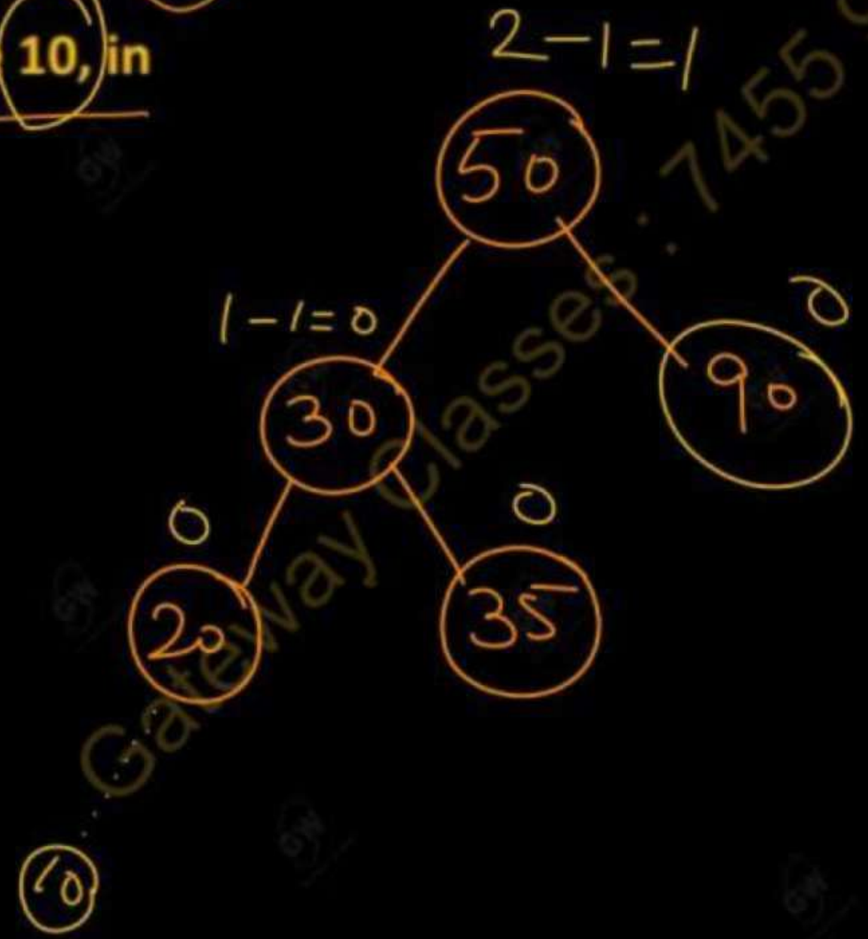
(Here A_R: Right subtree of A, B_L Left subtree of B, B_R: Right sub-tree of B)

i.e., in LL Rotation

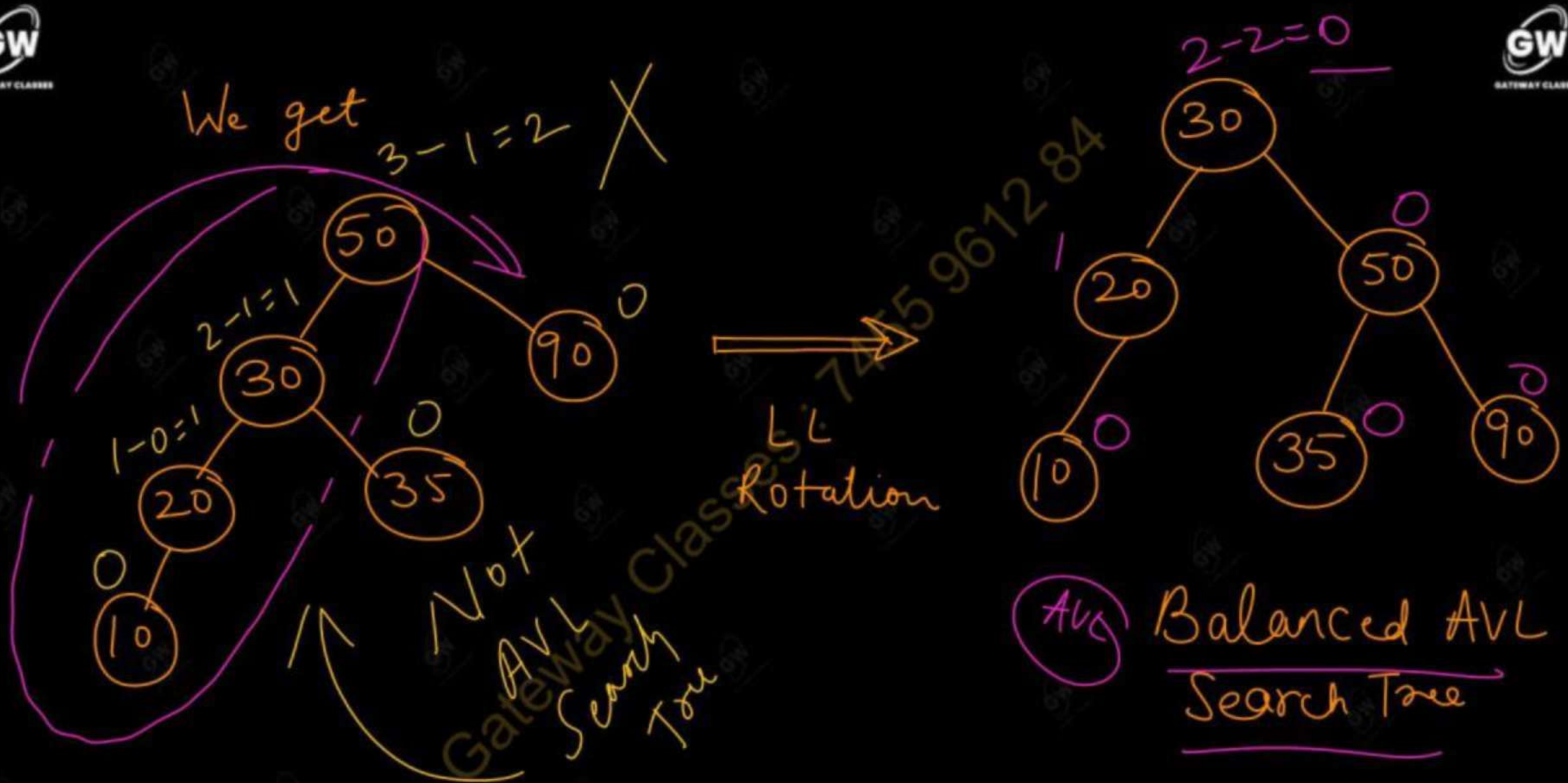
Left (A) ← Right (B)

Right (B) ← A

Example: Insert node 10, in



AVL Tree


We get
3-1=2
X
50
30
90
20
35
10
2-1=1
1-0=1
0
0
0
Not AVL Search Tree
LL Rotation
2-2=0
30
20
50
10
35
90
1
0
0
0
0
AVL
Balanced AVL Search Tree
Gateway Classes 7455 9612 84

(b) RR Rotation

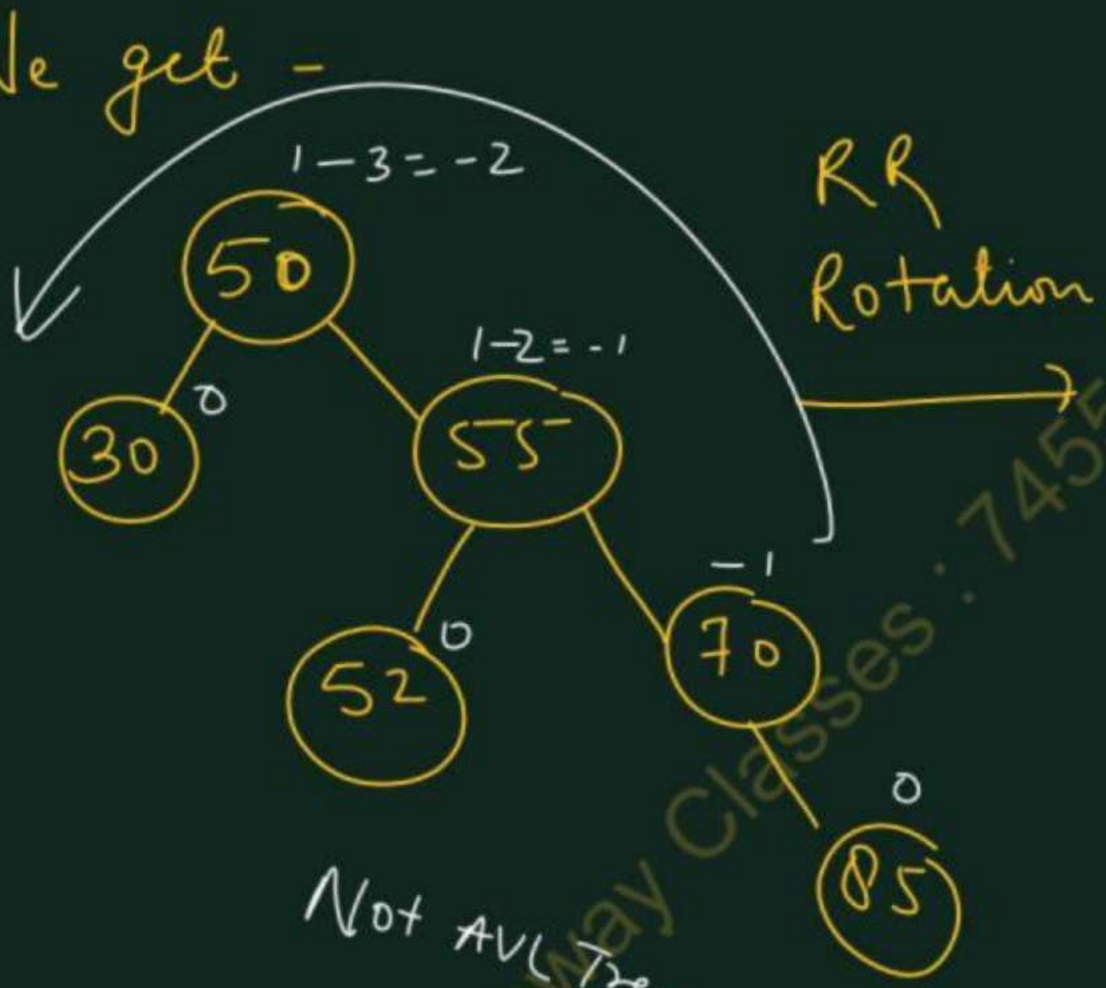
- Unbalance occurred due to the insertion in the right sub-tree of the right child of the pivot node.
- So, in this rotation new node is inserted at the right sub-tree of the right sub-tree of root node (pivot node).
- The next figure illustrates the balancing of an AVL search tree using RR rotation.
- Here the new node x is in the right sub-tree of A.
- The re-balancing rotation pushes B upto the root with A as its left child and B_R as its right sub-tree and A_L and B_L as the left and right sub-trees of A.
- That is, in RR rotation-

  Right (A) ← left (B)

  Left (B) ← A

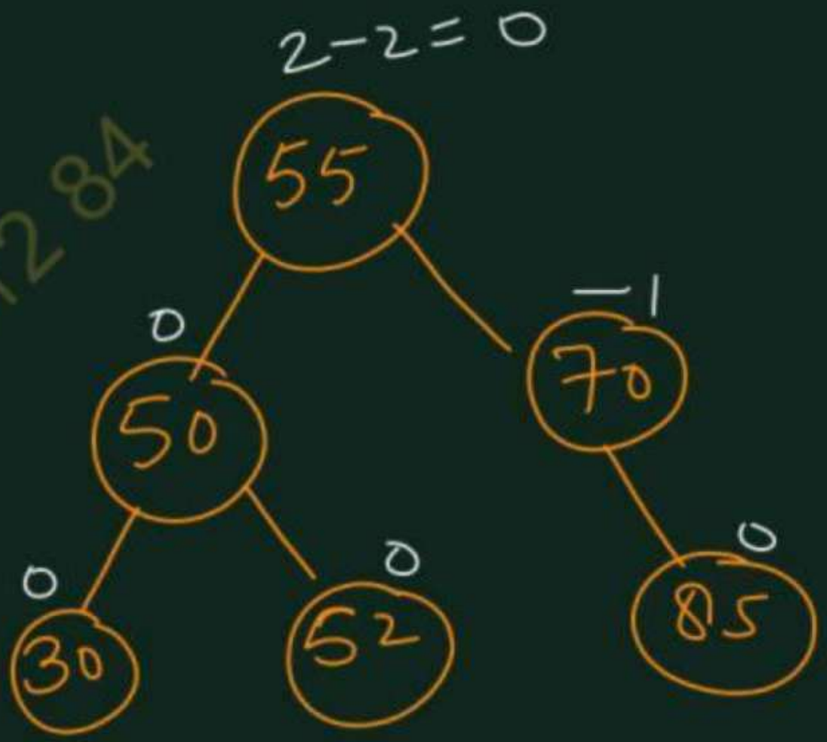
Example: Insert node 85 in.

We get -



RR Rotation

Not AVL Tree

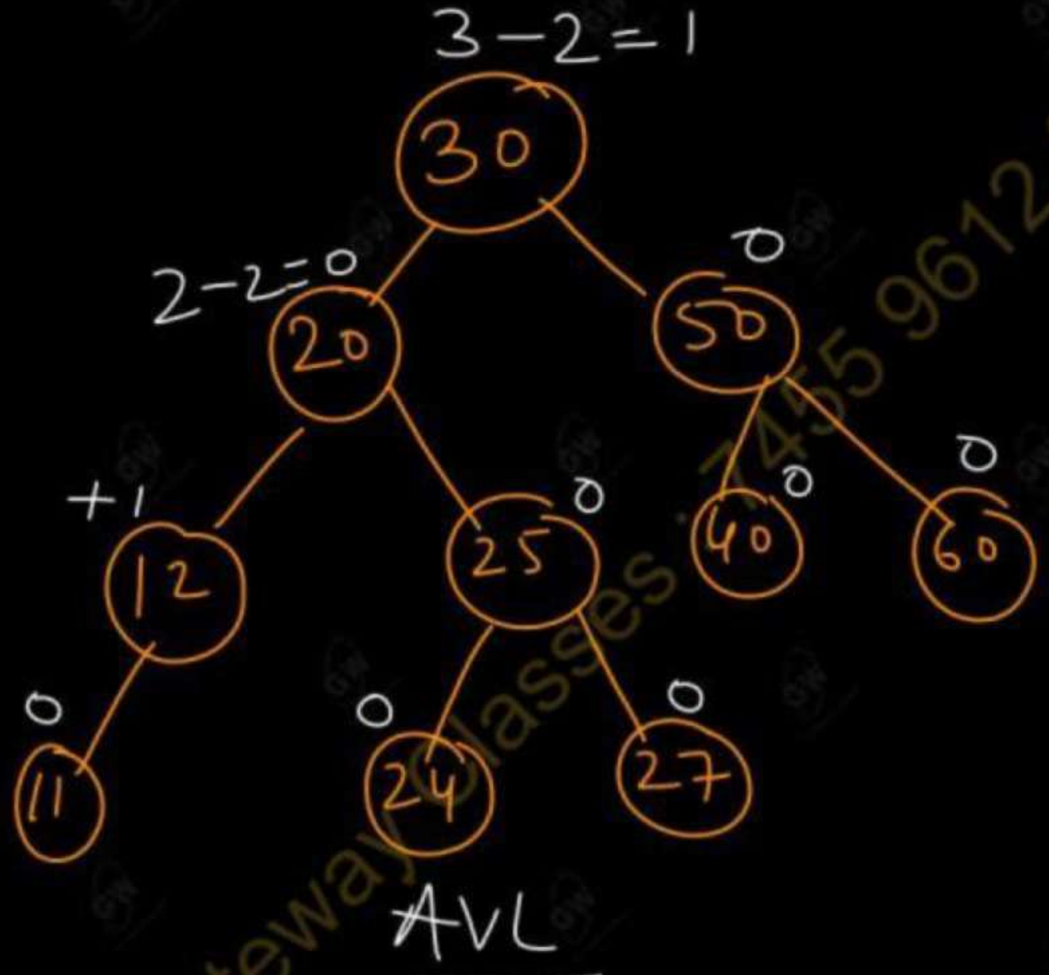


Balanced AVL earch Tree

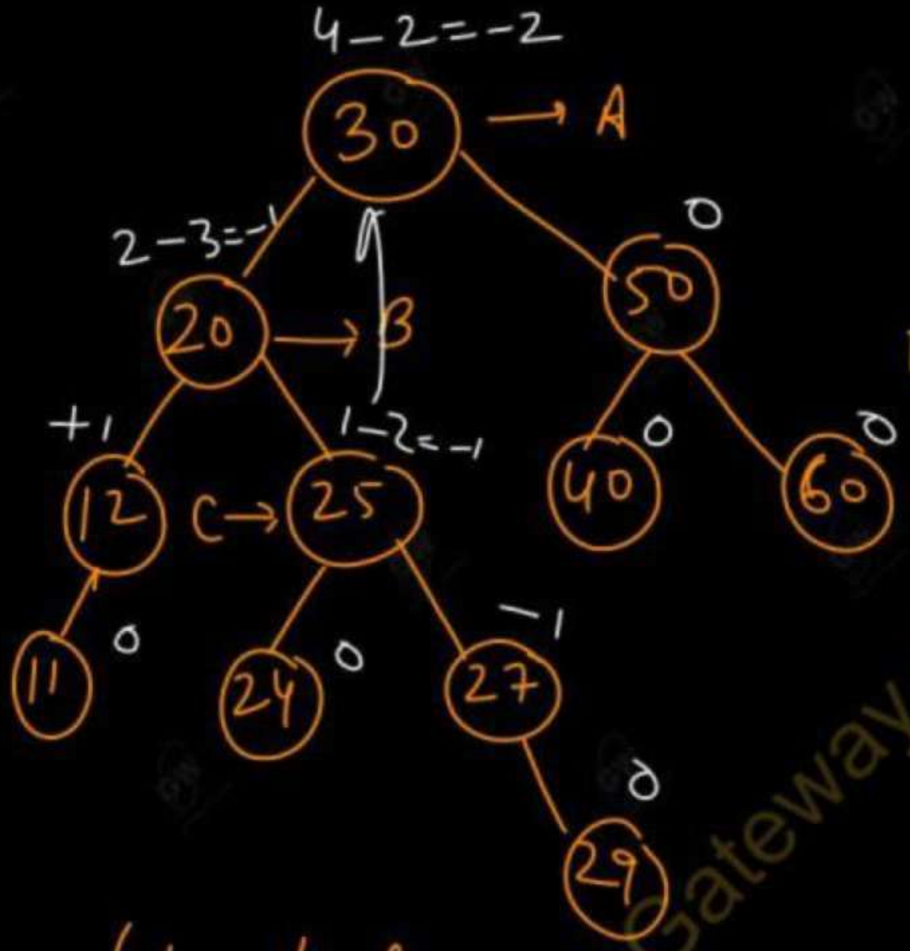
(c) LR Rotation

- In this, unbalance occurred due to the insertion in the right sub-tree of the left child of the root (pivot) node.
- So, this is known as left to right insertion.
- LR rotation involves two rotations for the manipulation in pointers.

Example: Consider the following AVL tree which is height balanced:

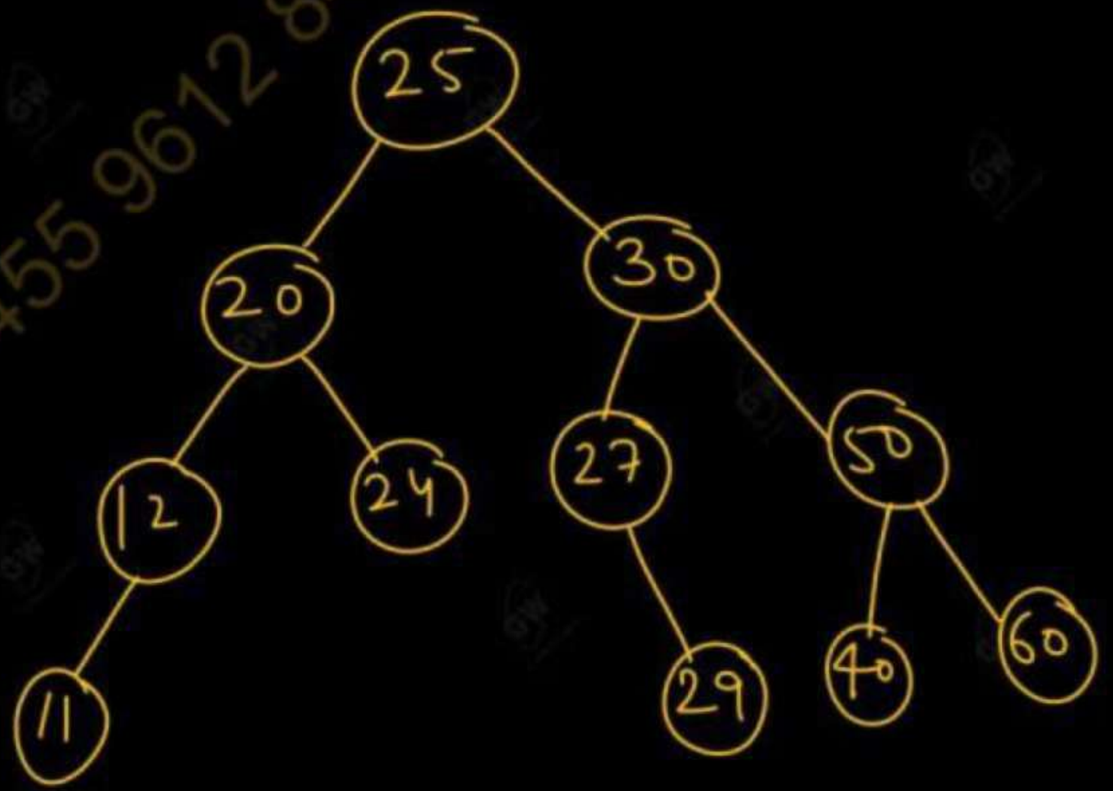


- Insertion of the value 29 makes the tree unbalanced as shown in Fig (a).
- Now AVL rotation has to be performed to make the tree balanced.



(Unbalanced BST ~~AVL~~ Tree)

LR Rotation

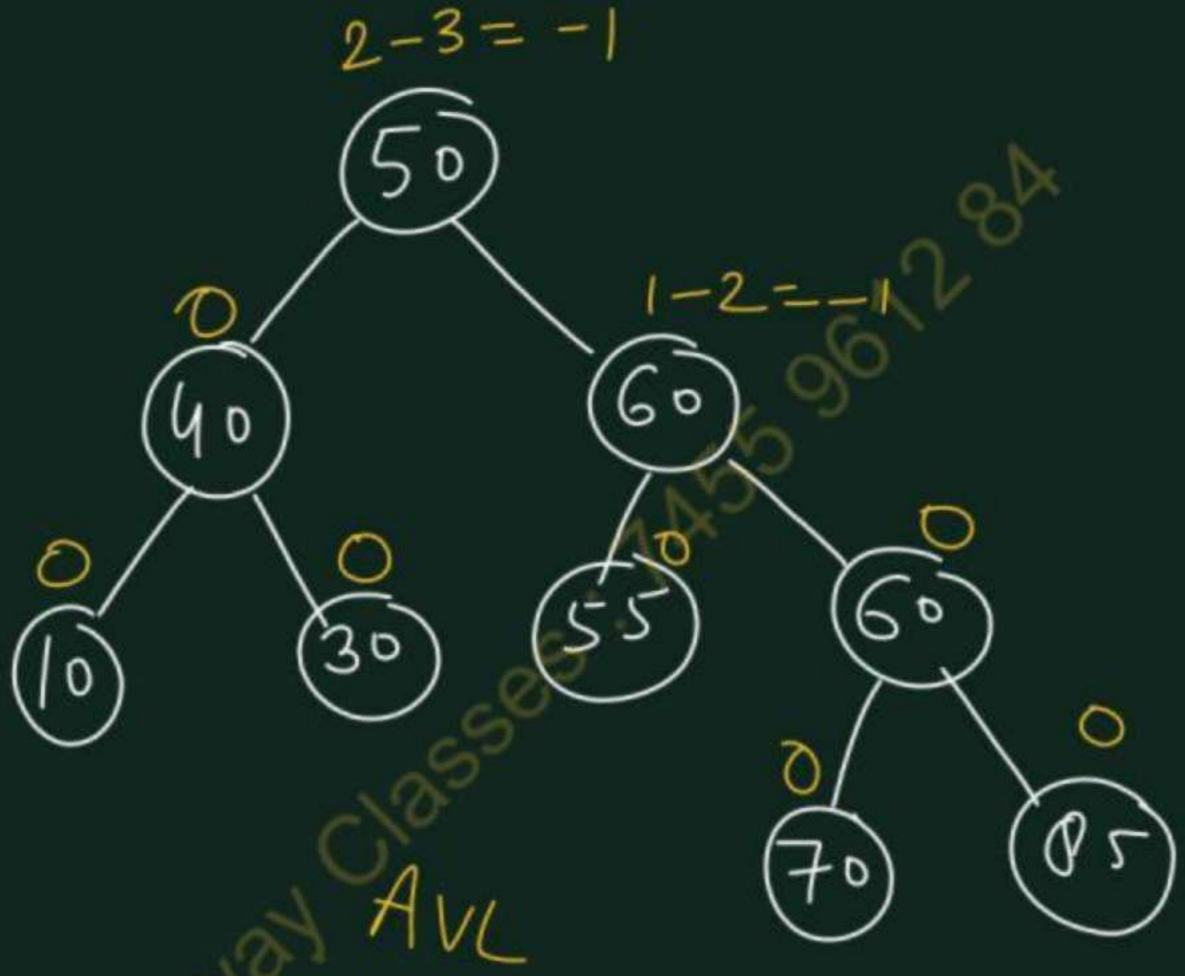


{ After Rotation Balanced AVL TREE }

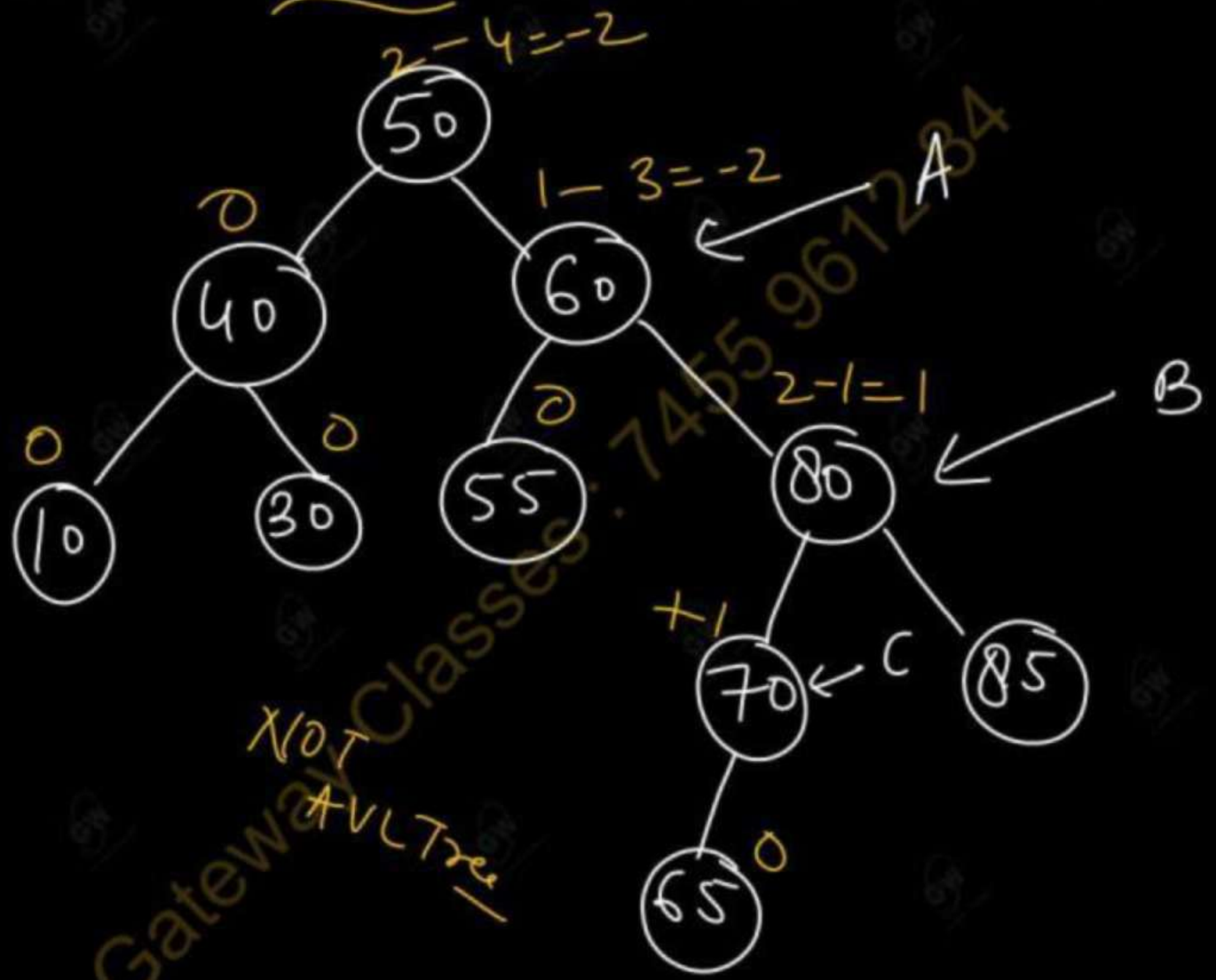
(d) RL Rotation:

- In this, unbalance occurred due to the insertion in the left sub-tree of the right child of the r (pivot) node. This is known as right-to-left insertion.
- RL-Rotation is the mirror image of LR-Rotation. Next figures, illustrates the unbalance appearance and its removal due to the insertion.

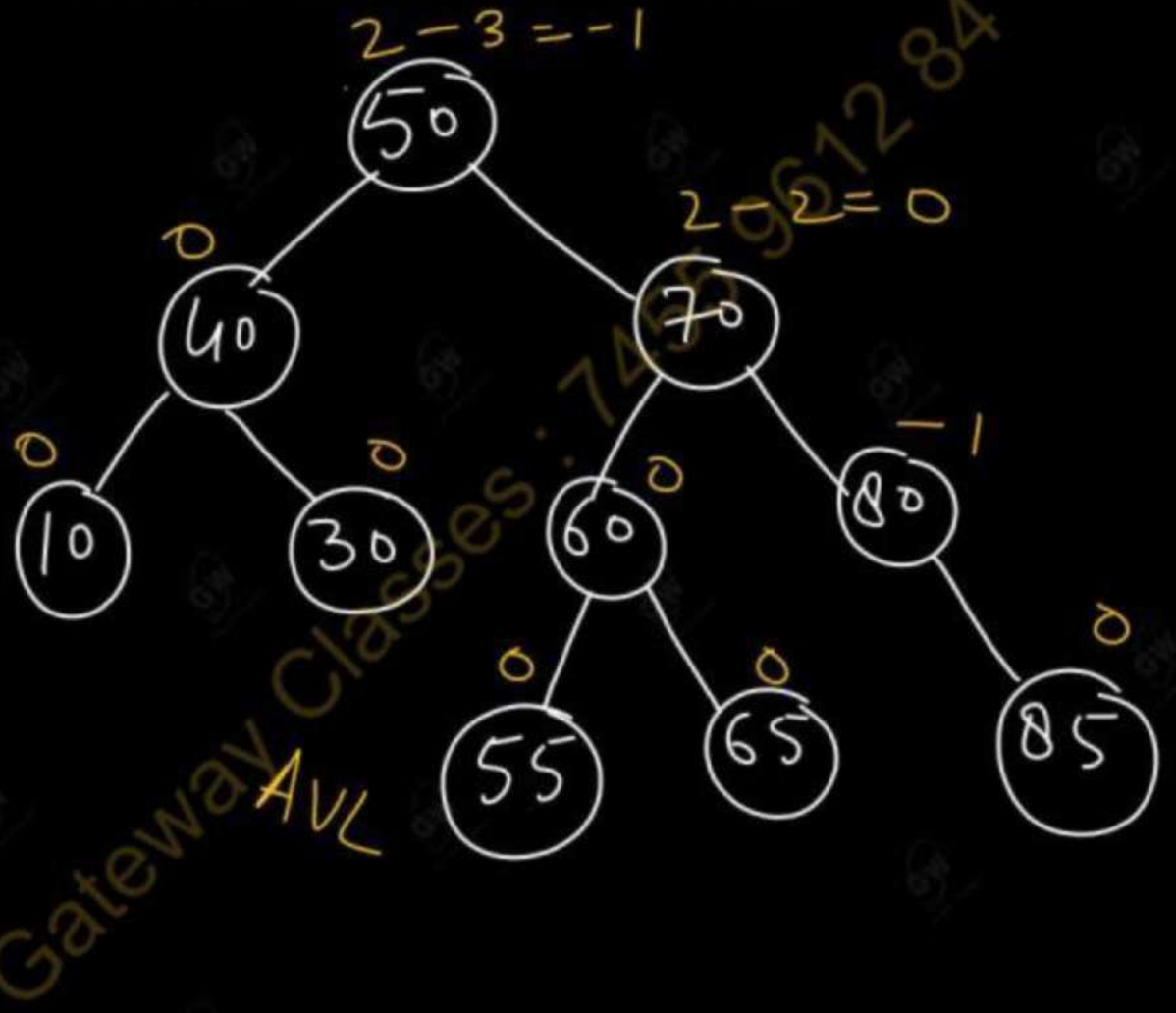
- Here, there are two rotations for the manipulations of pointers. They are the following:
- **Rotation 1:** The right sub-tree (C_R) of the left child (C) of the right child (B) of root node (A) becomes the left sub-tree of B and the right child (B) of the root node becomes the right child of C.
- **Rotation 2:** The left sub-tree (C_L) of the left child (C) of the right child (B) of the root node becomes the right sub-tree of A.
- **Example:** Consider the AVL tree as in next figure –



❑ This is height unbalanced. When 65 is inserted into this tree it becomes



- Now the AVL tree is unbalanced.
- To make it balanced, do the AVL rotations (RL rotation).



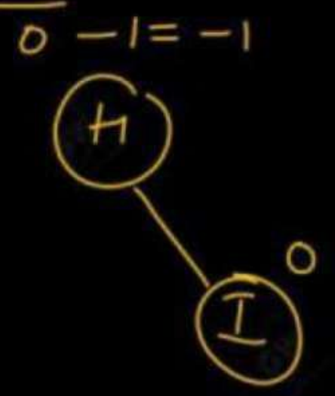
- Now the tree has become a height balance tree.

Example: - Create an AVL tree from the given set of values:

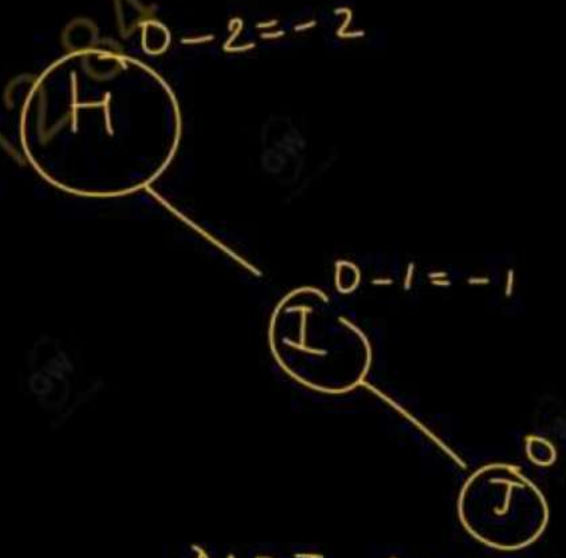
H, I, J, B, A, E, C, F, D, G, K, L

Solution:

AVL

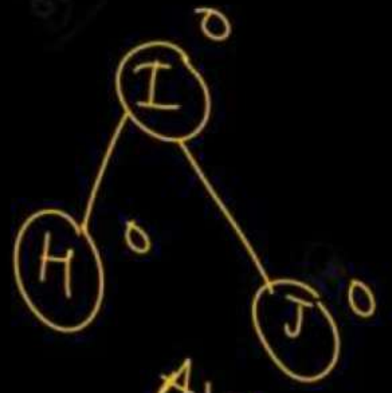


AVL

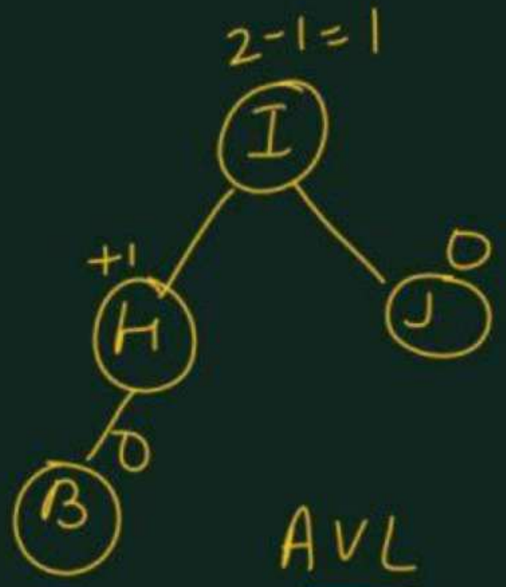


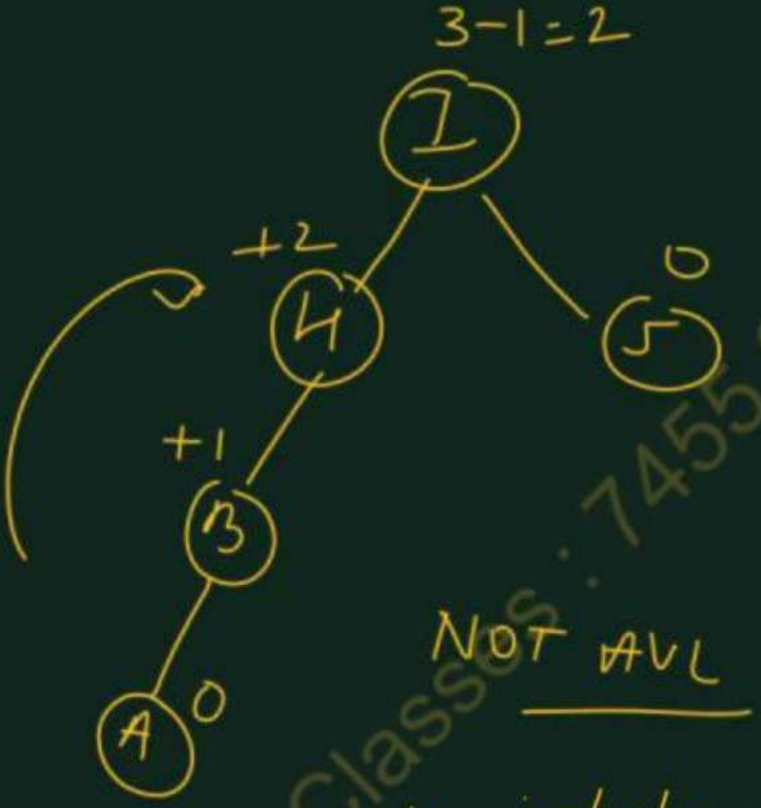
NOT AVL Tree

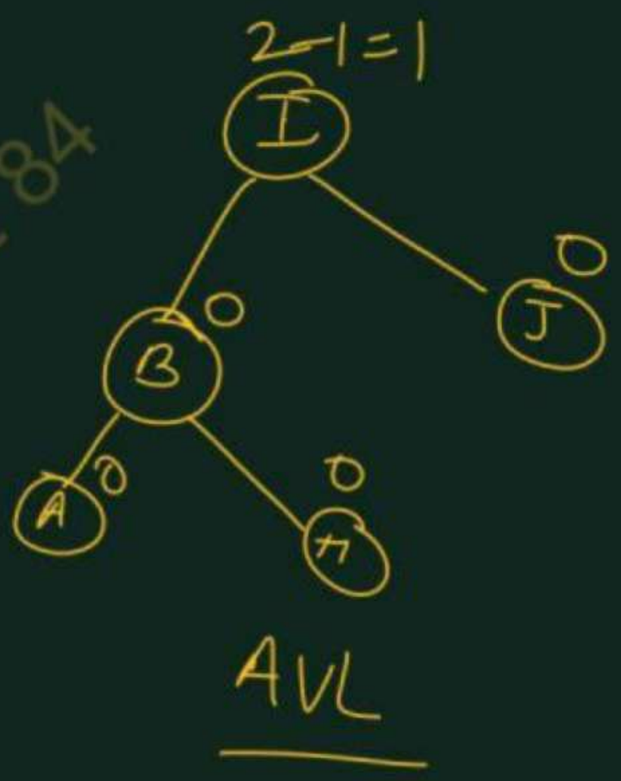
RR imbalance (Perform left rotation)

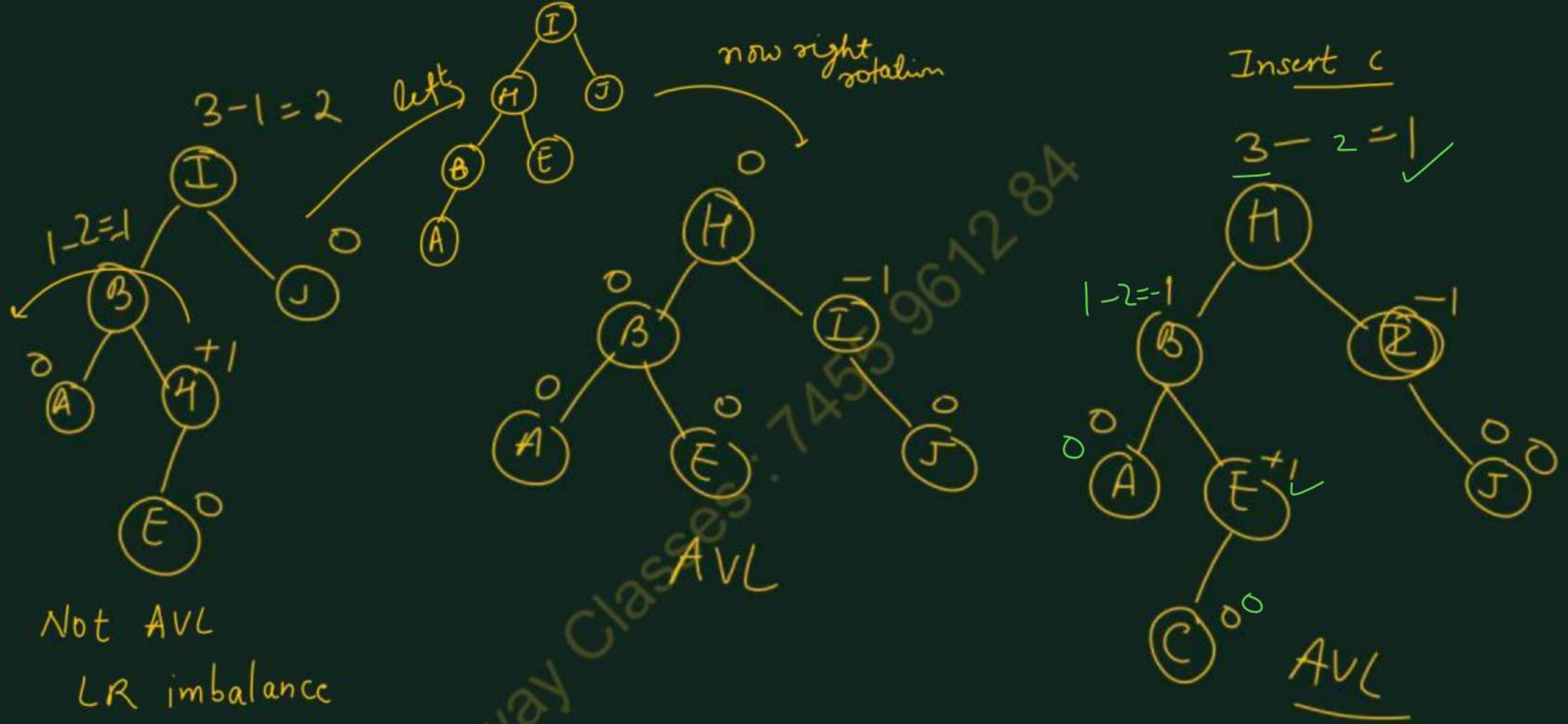


AVL

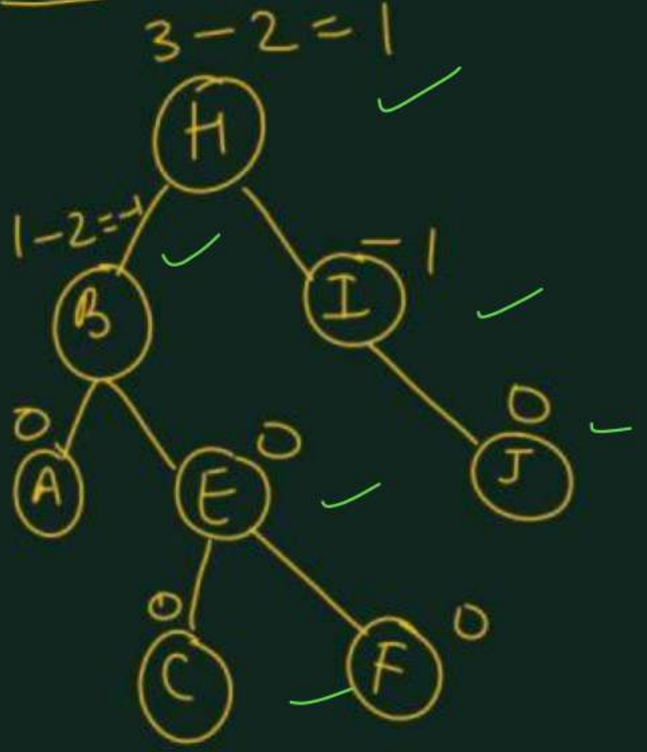






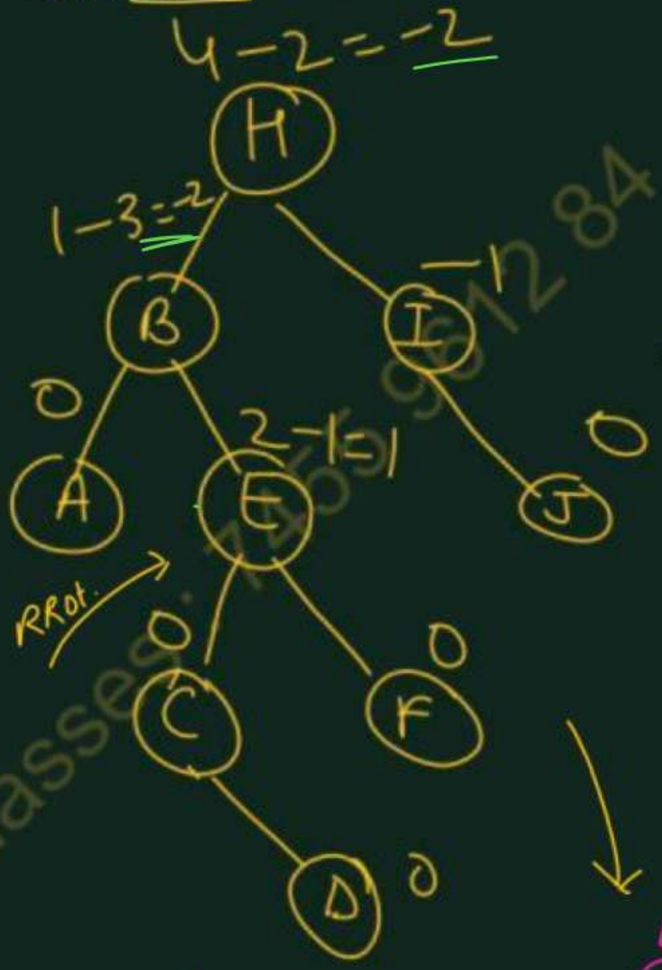


Insert F



AVL

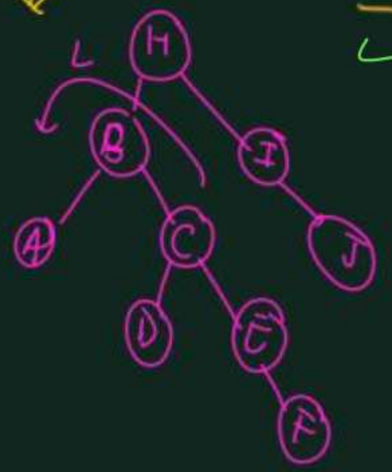
Insert D

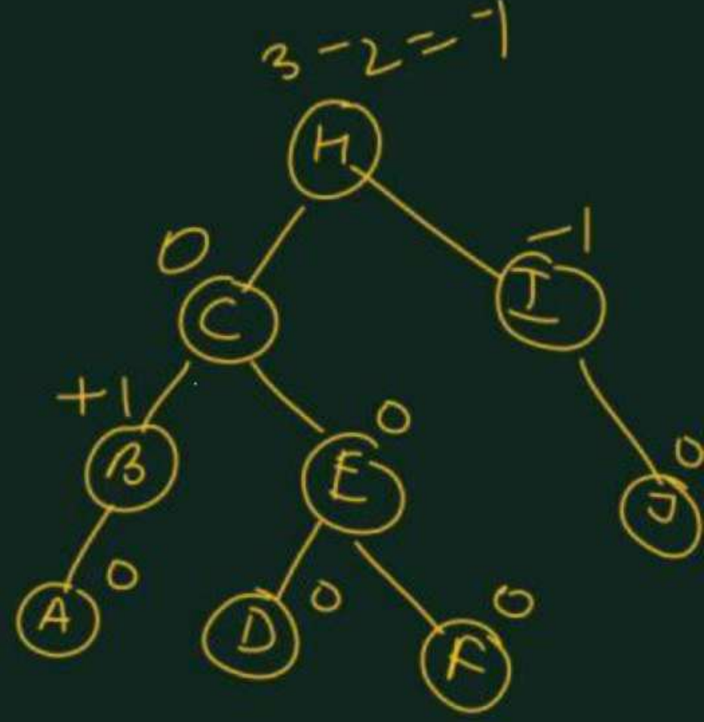


RL imbalance

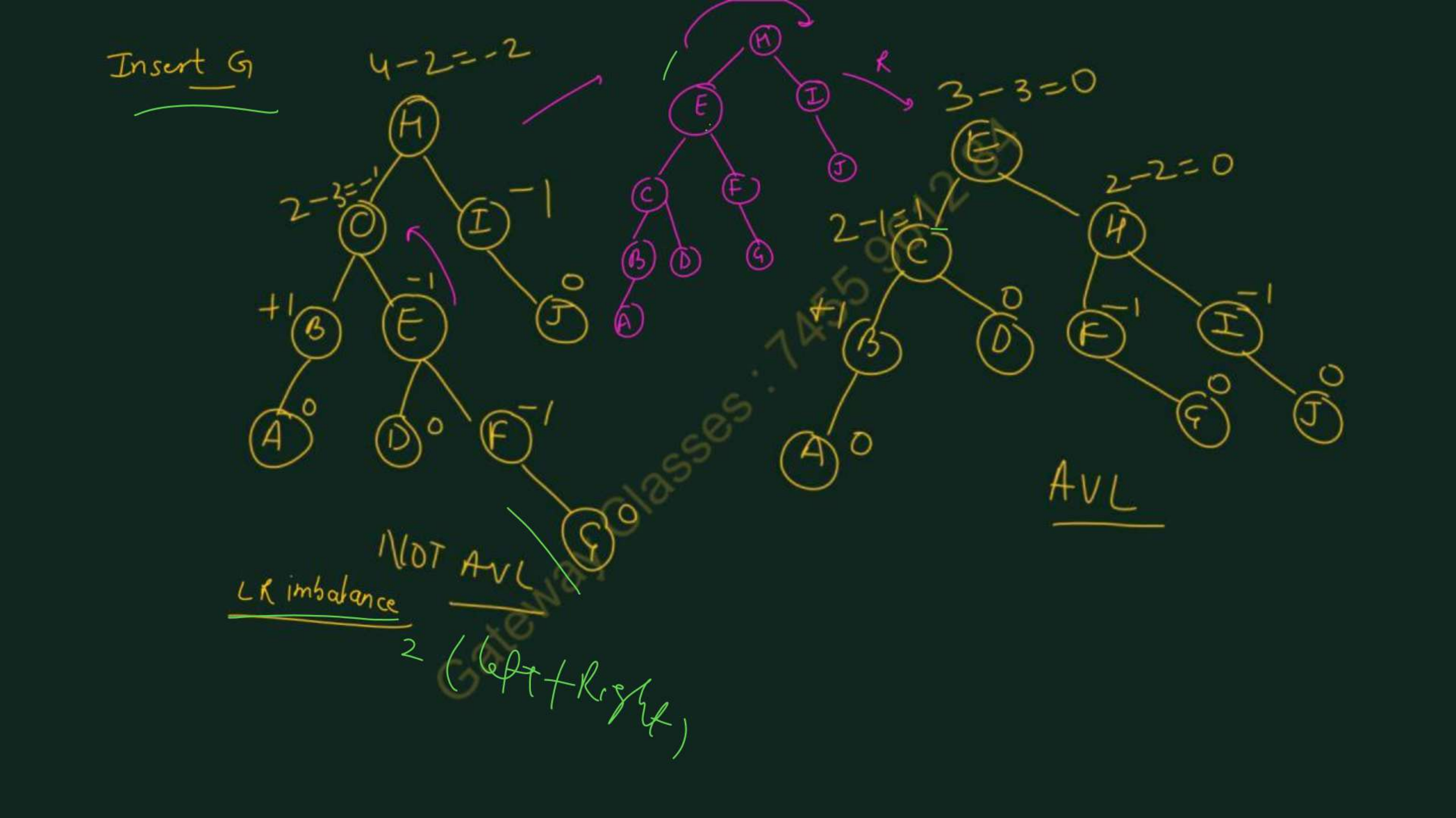
RL rotation

2 (Right + left)

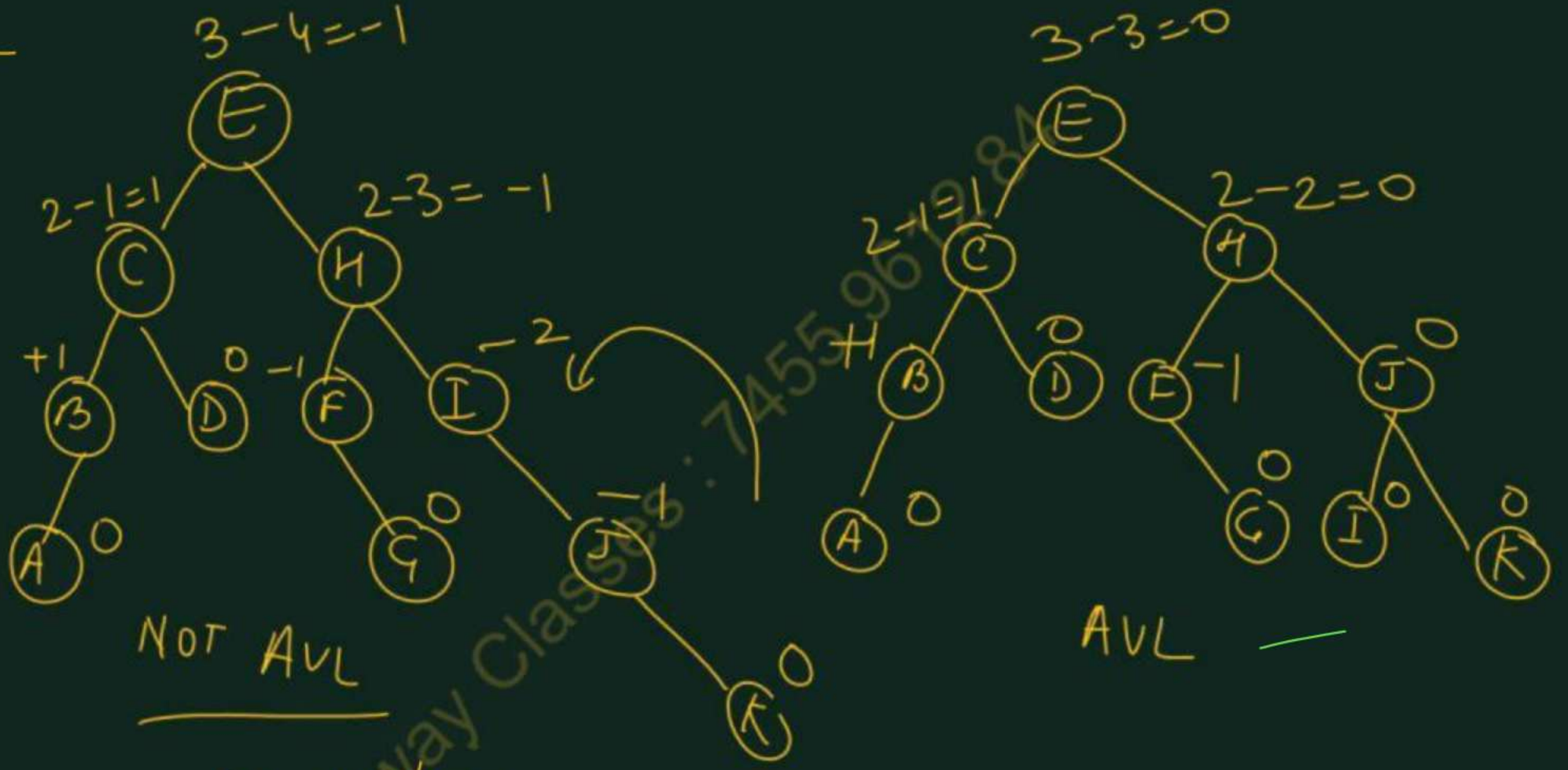




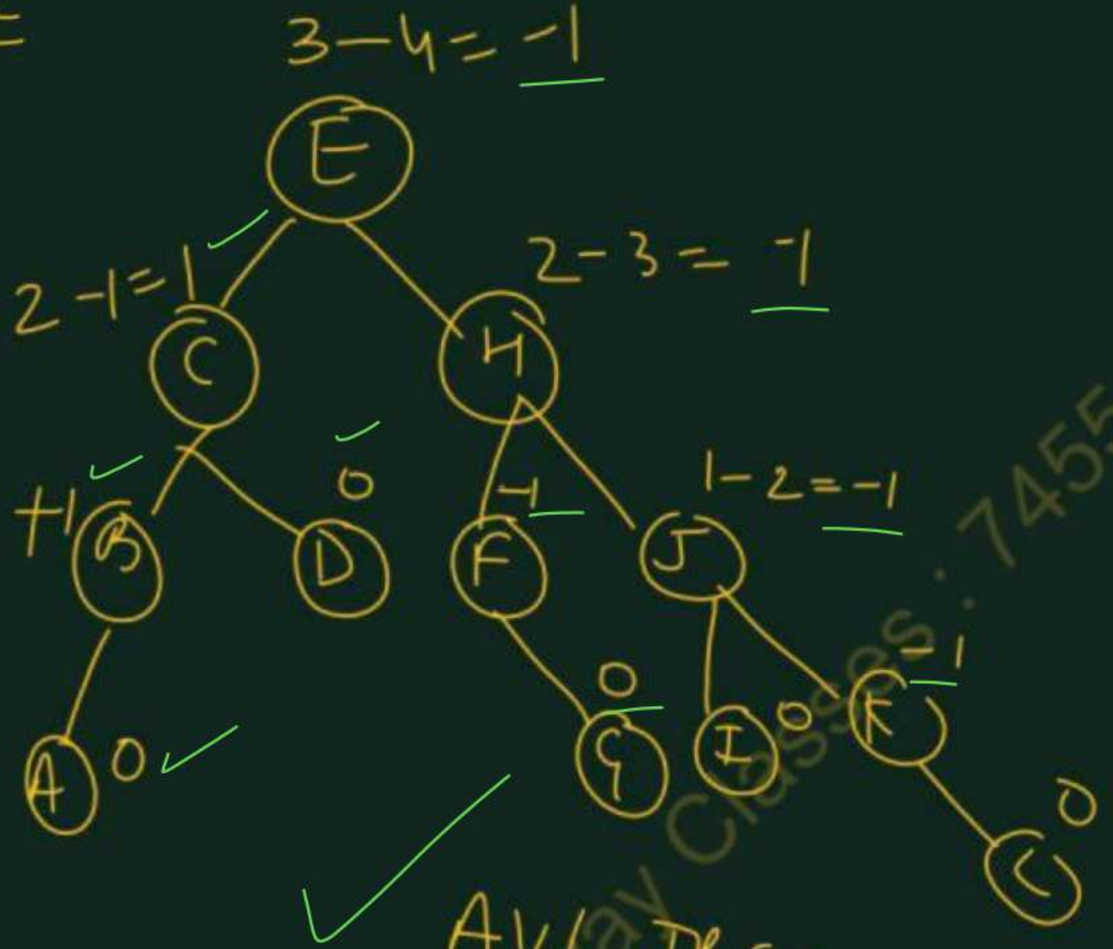
AVL


Insert G
4-2=-2
2-3=-1
-1
+1
-1
0
-1
0
NOT AVL
LR imbalance
2 (left + Right)
R
3-3=0
2-2=0
2-1=1
+1
0
-1
-1
0
0
AVL

Insert K



Insert L

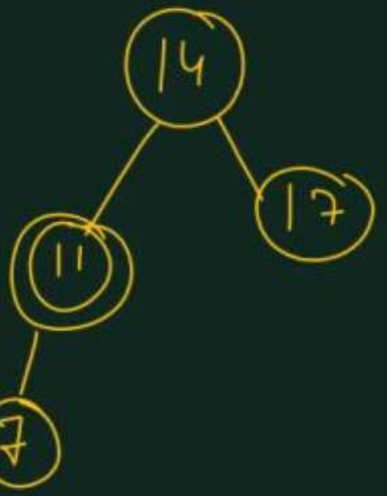


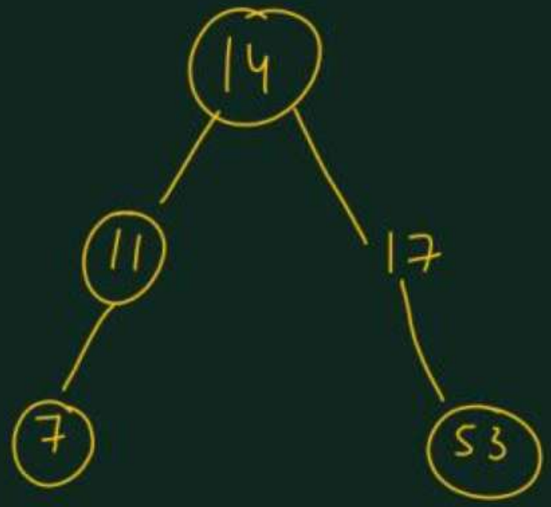
AVL TREE

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

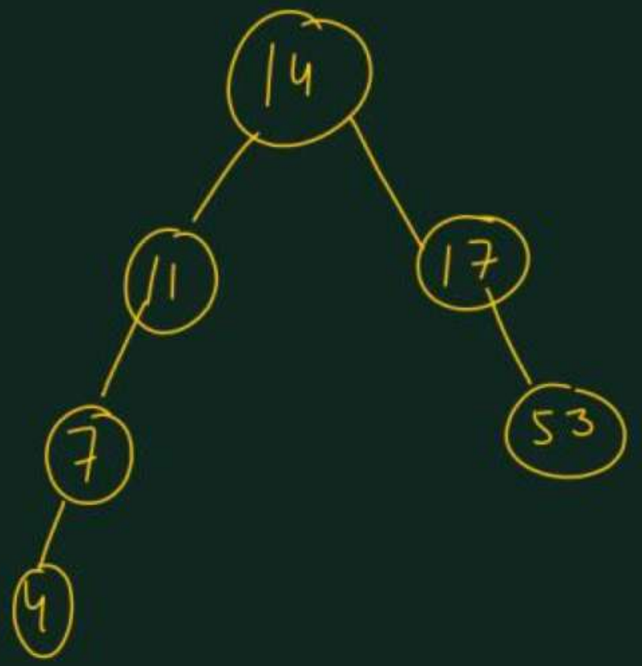
A

B





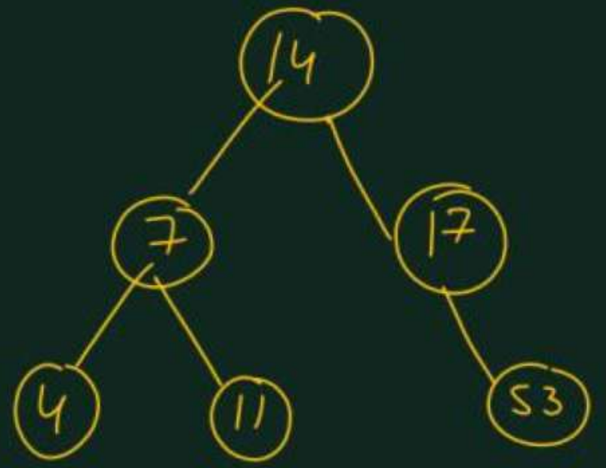
4



LL Imbalance

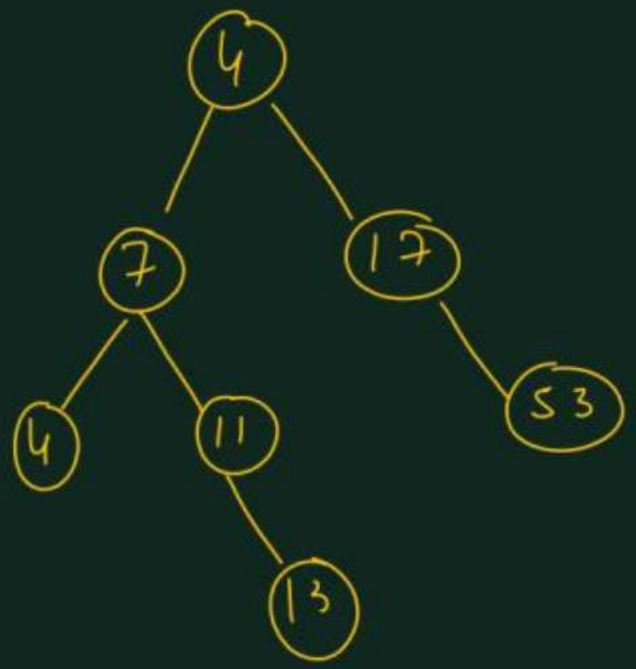
rotate right

⇌

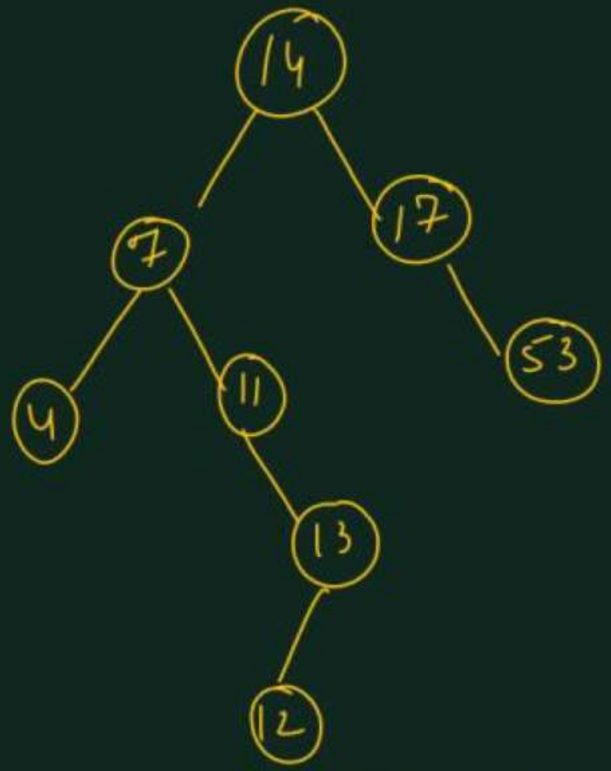


AVL

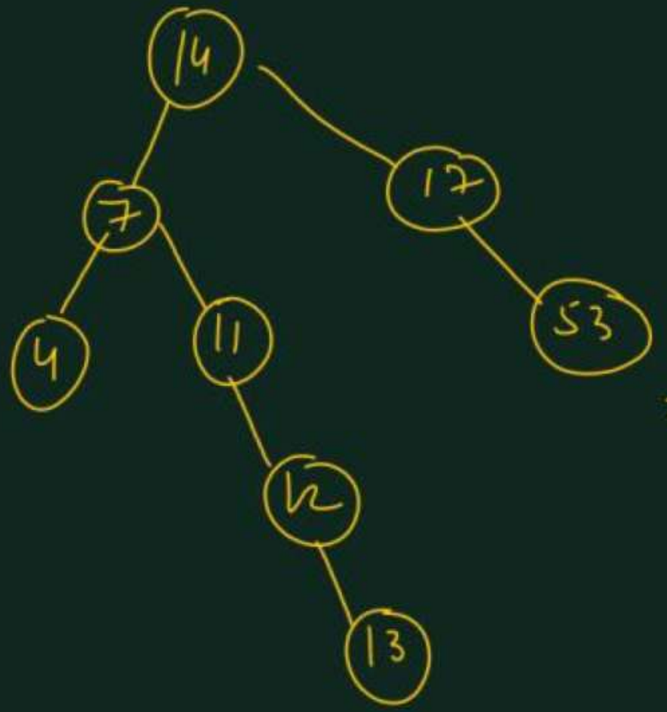
13

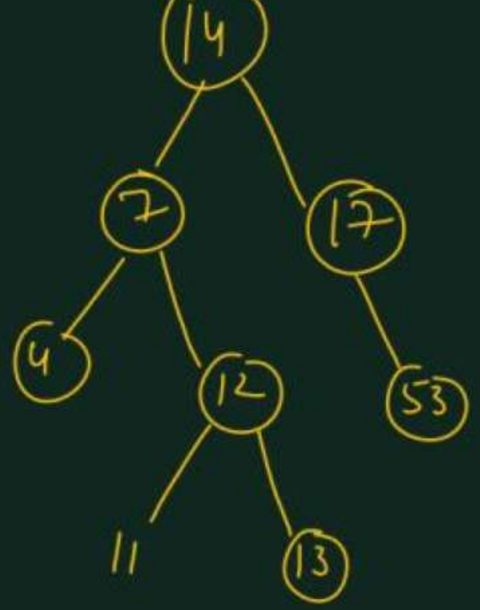


12

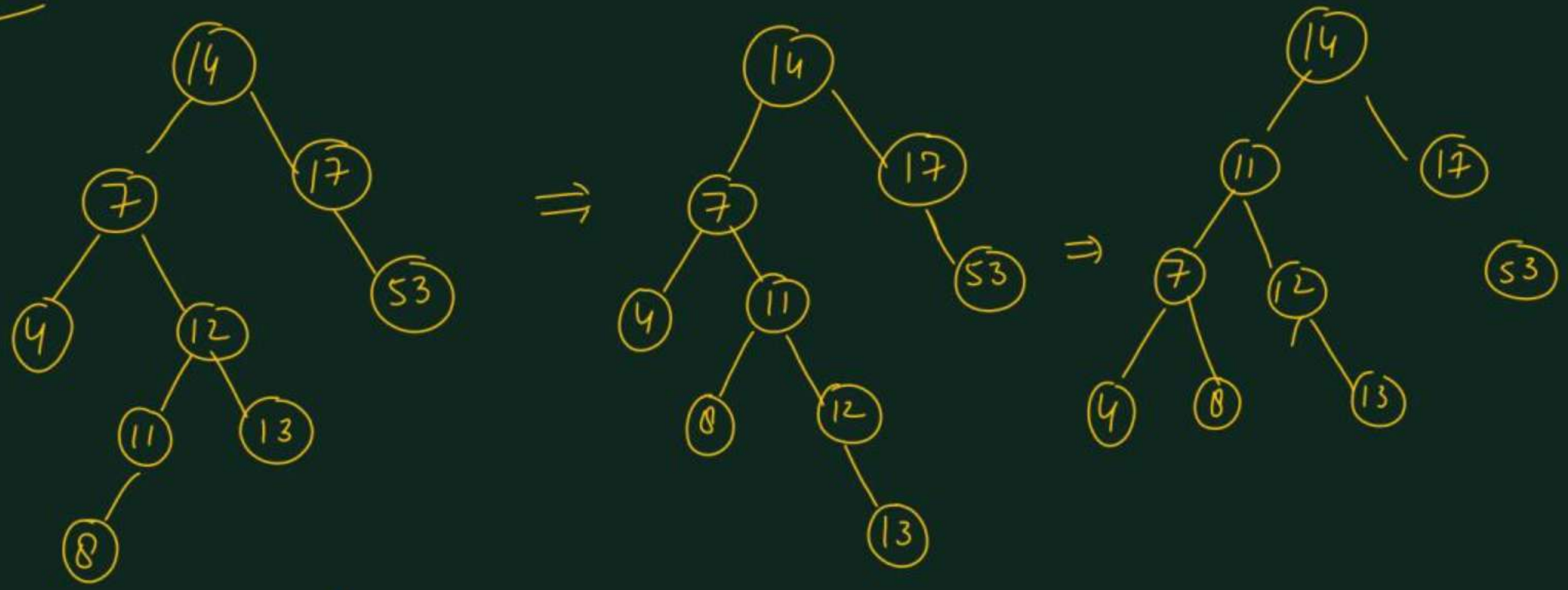


RL imbalance
right + left





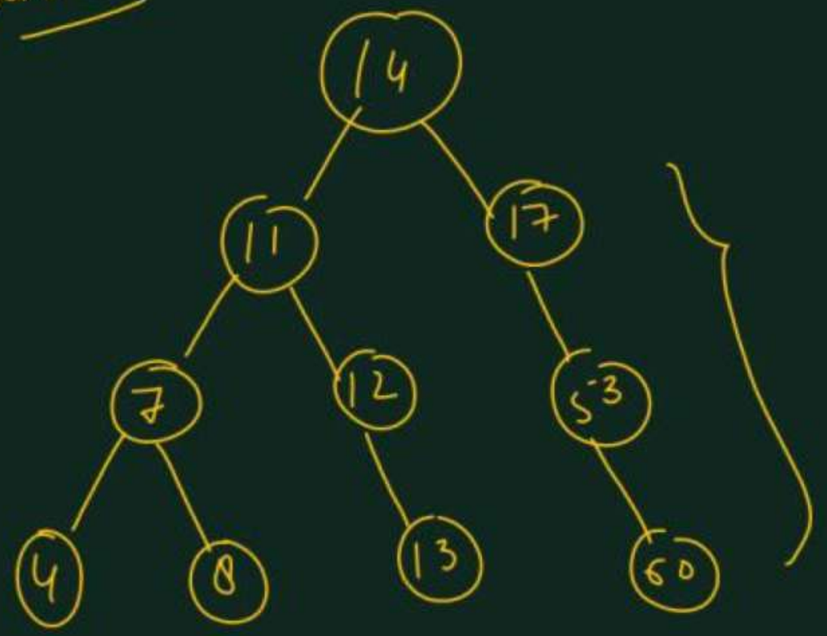
Insert 8



RL rotation

right + left

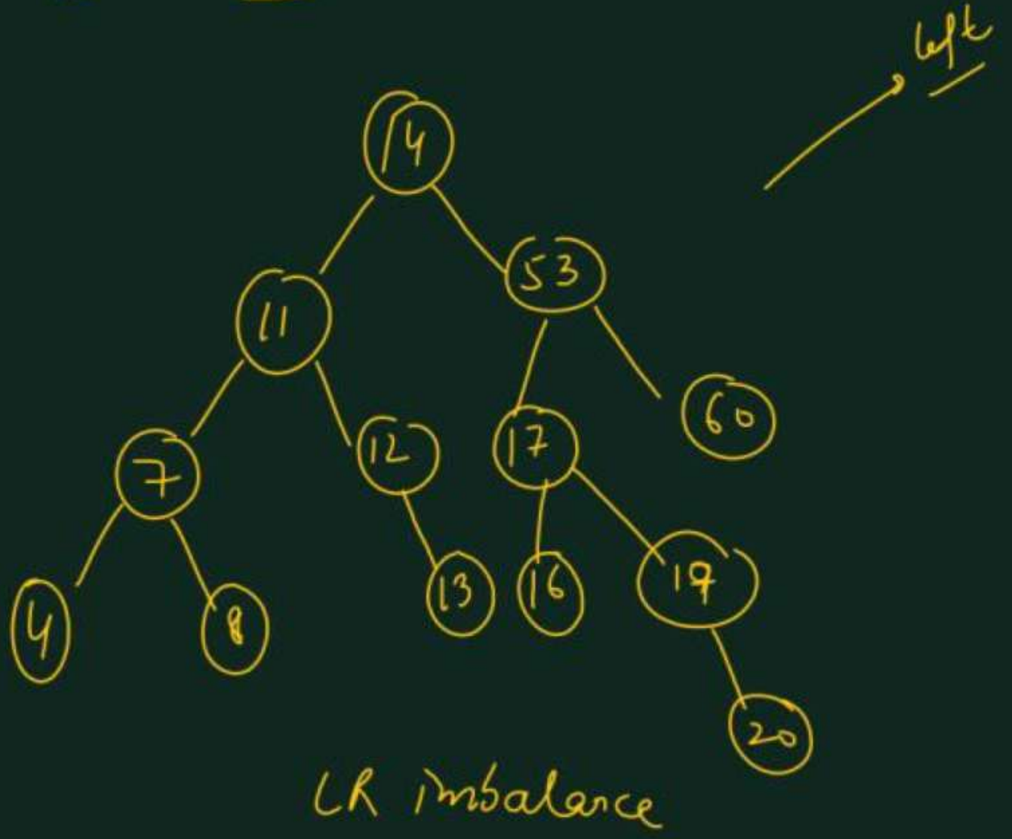
Insert 60



RR imbalance

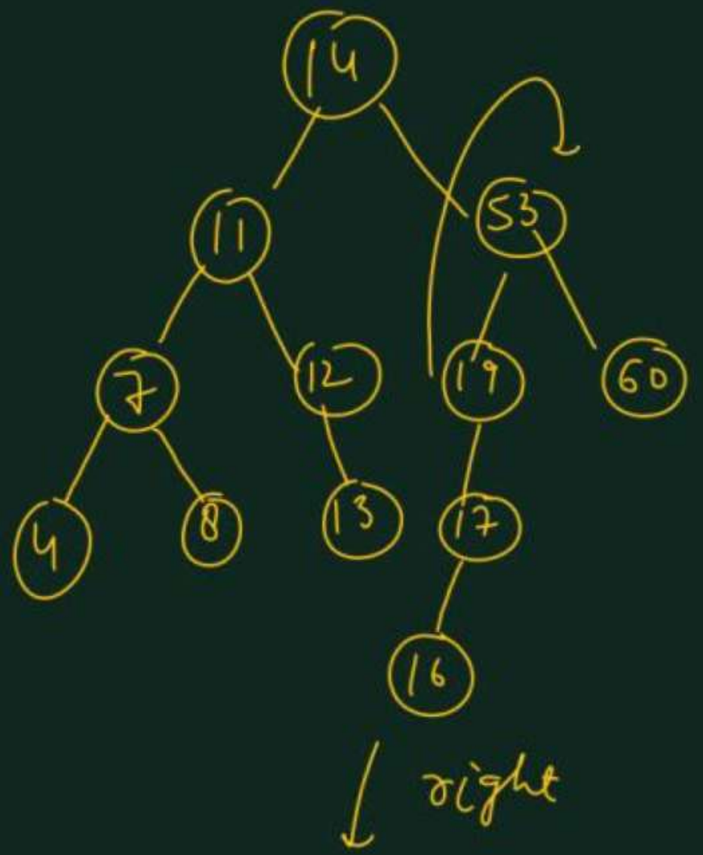
insert 19

Insert 20



left

LR Imbalance



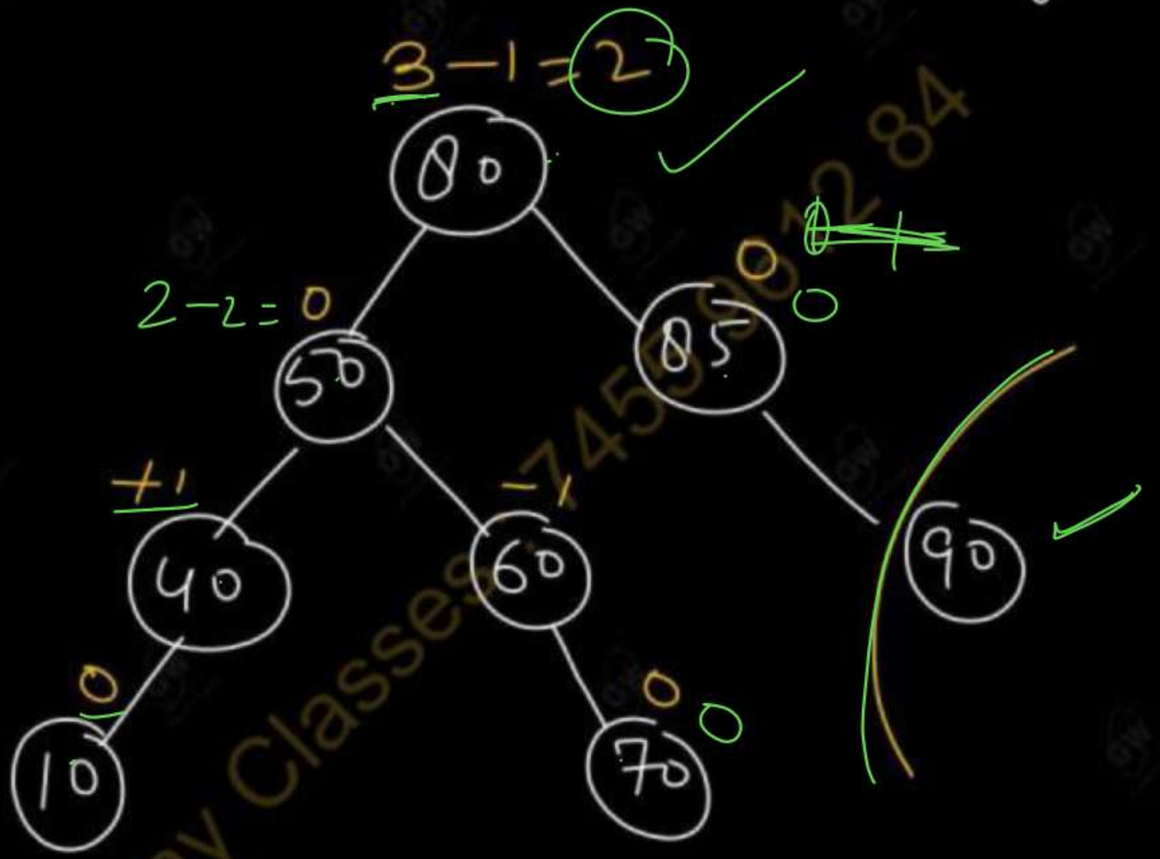
right

DELETION IN AN AVL TREE

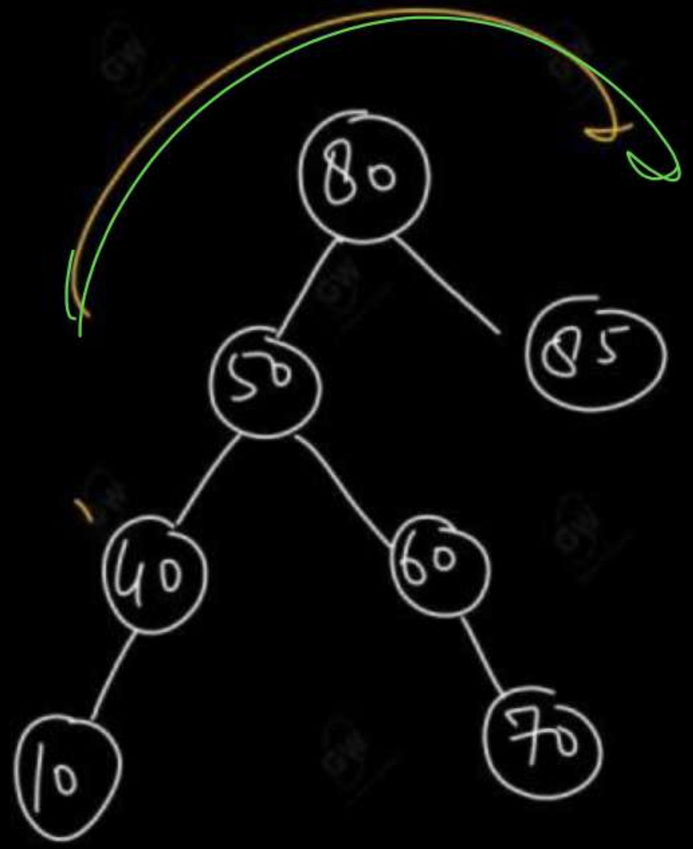
- The deletion of a node from an AVL tree is exactly the same as the deletion of a node from the BST.
- The sequence of steps to be followed in deleting a node from an AVL tree is as follows:

1. Initially, the AVL tree is searched to find the node to be deleted.
2. The procedure used to delete a node in an AVL tree is the same as deleting the node in the binary search tree.
3. After deletion of the node, check the balance factor of each node.
4. Rebalance the AVL tree if the tree is unbalanced. For this, AVL rotations are used.

Example: Delete 90 from the AVL search tree shown in the following figure.

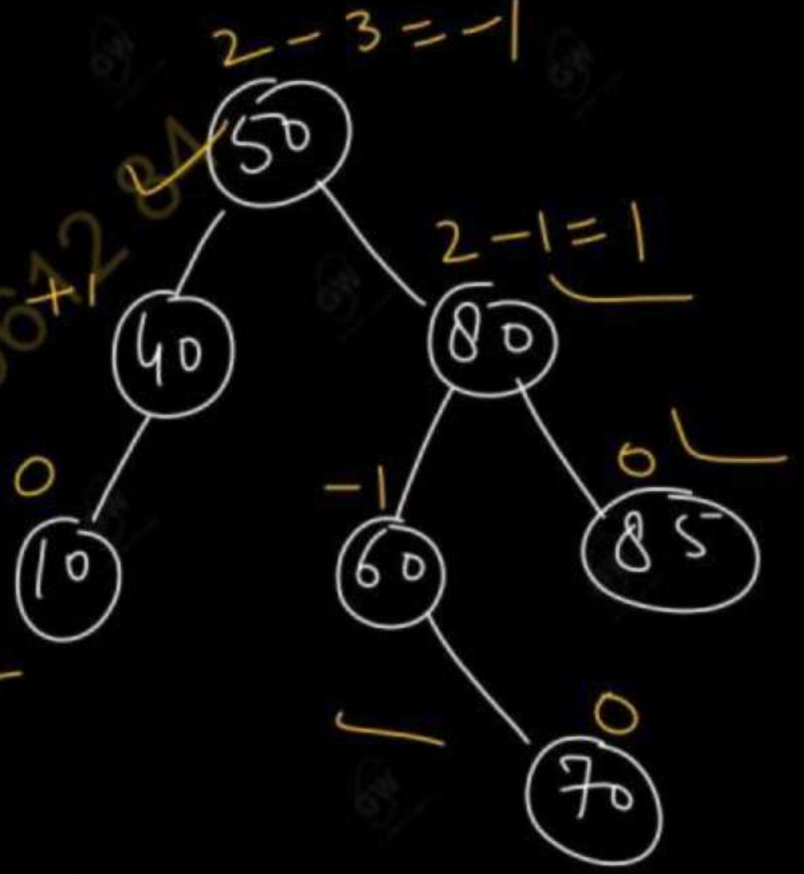


After deletion of 90, we get unbalanced AVL tree, as



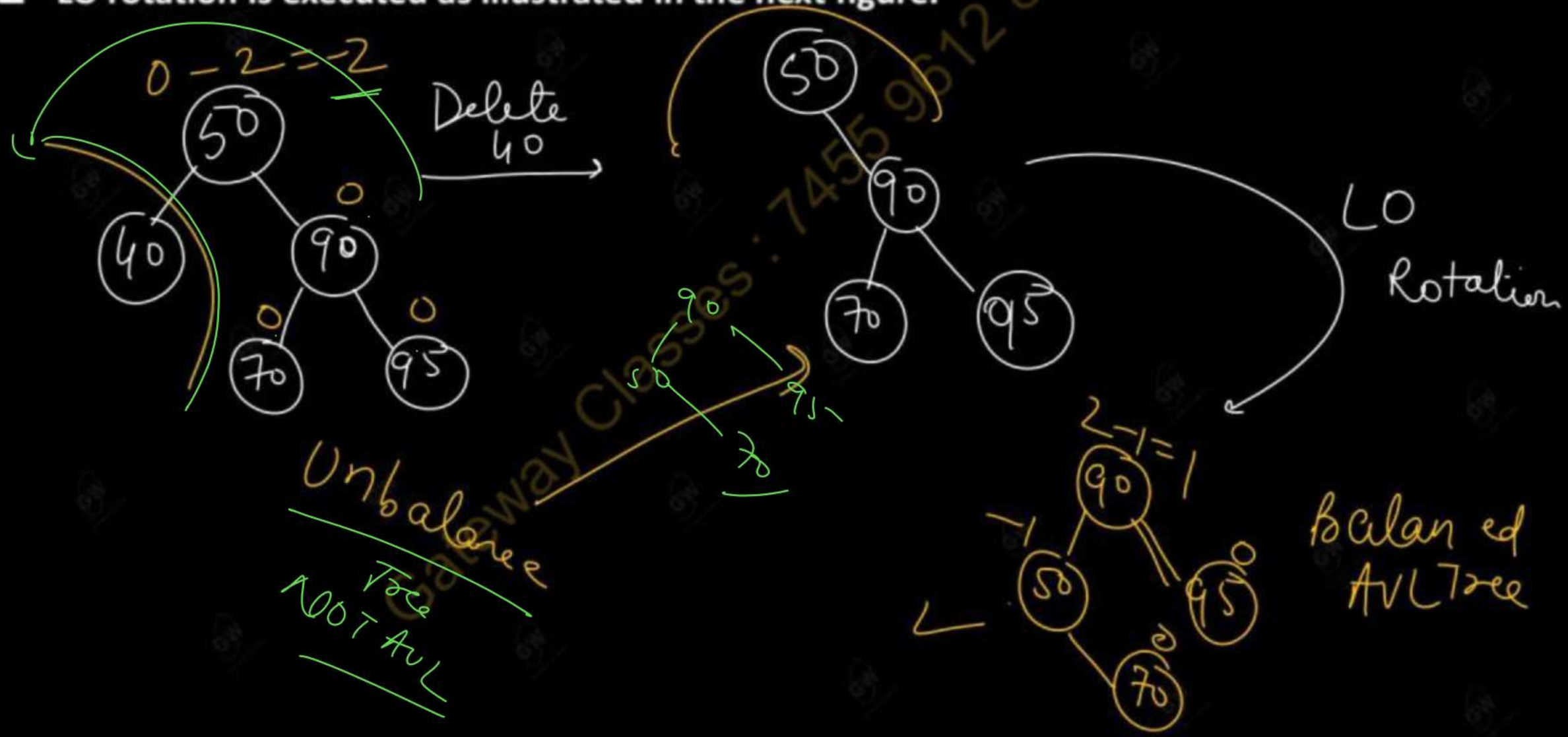
RO

Rotation

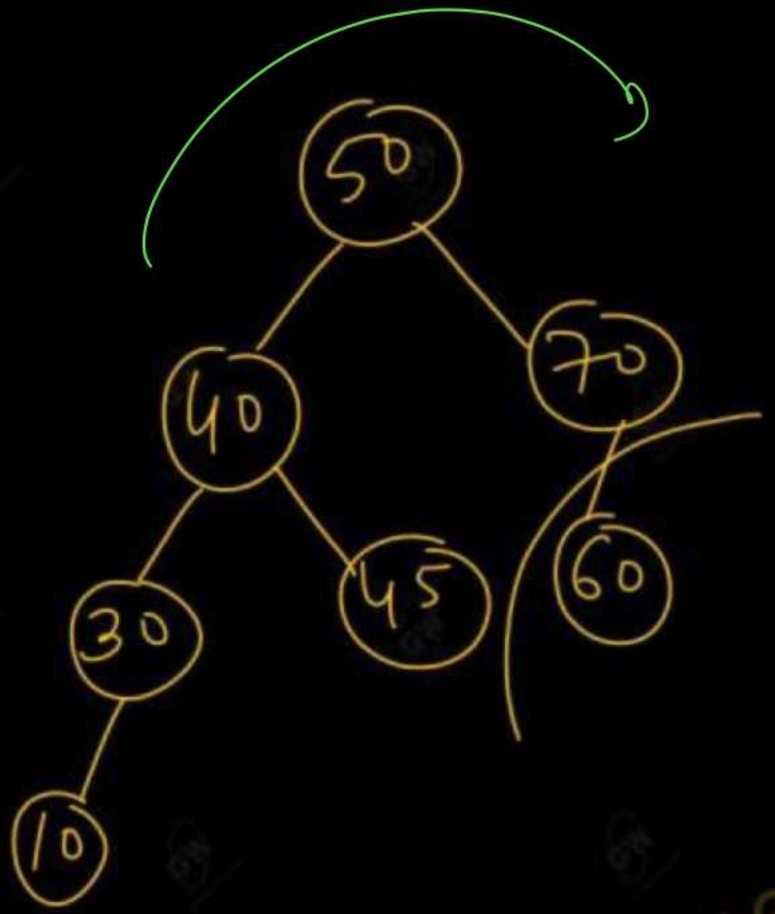


Balanced AVL Tree after RO Rotation

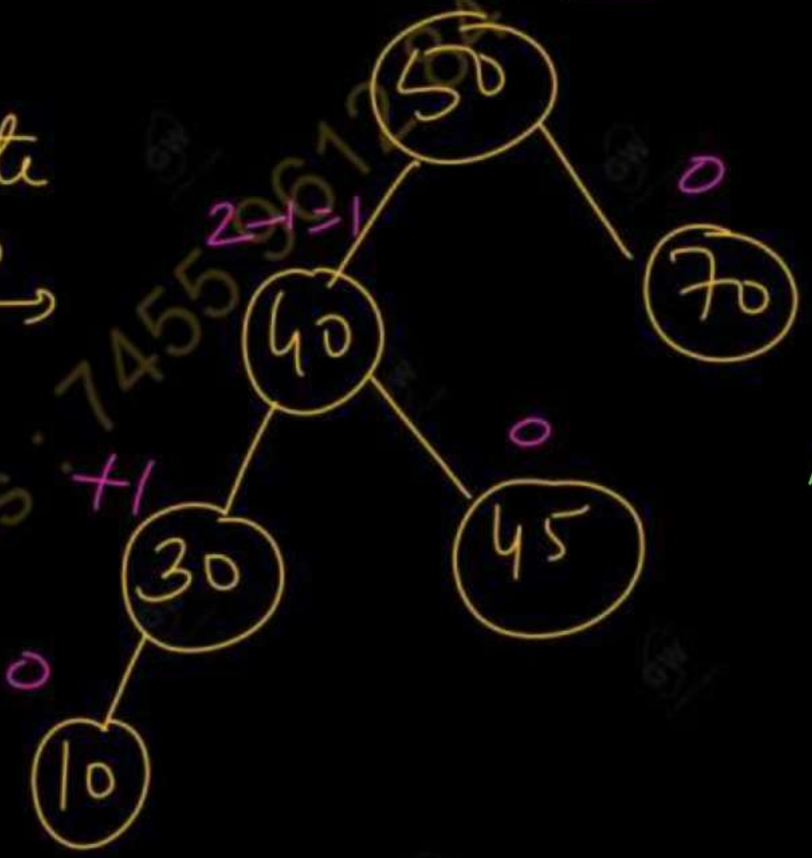
- Similarly LO rotation is executed, when balance factor of B is zero and B is the root of right sub-tree of A.
- LO rotation is executed as illustrated in the next figure.

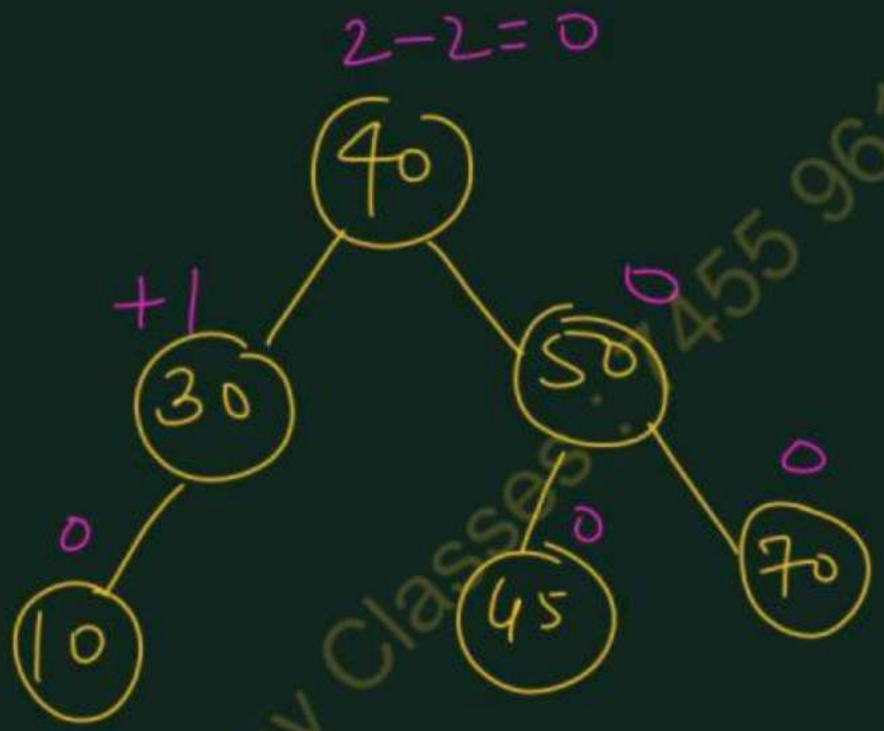


Example –



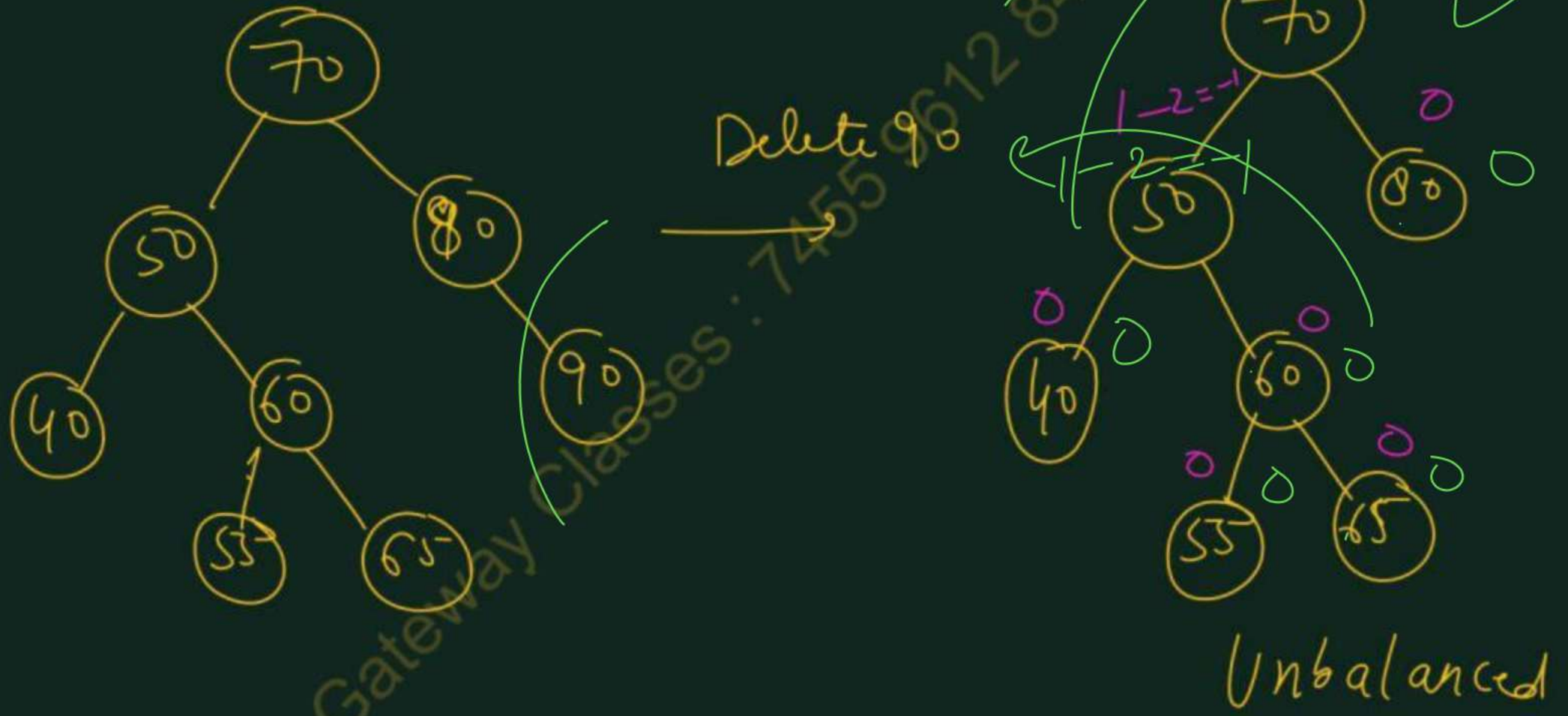
Delete 60



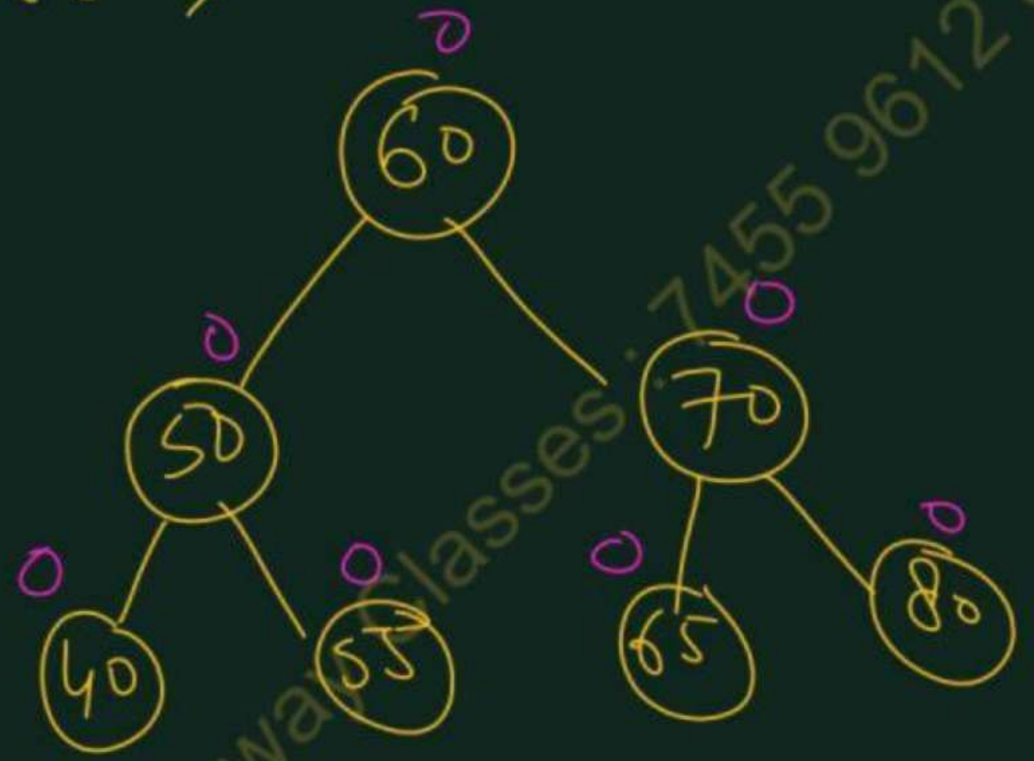


Balanced AVL Tree

Example –



By R(-1) rotation, We get -



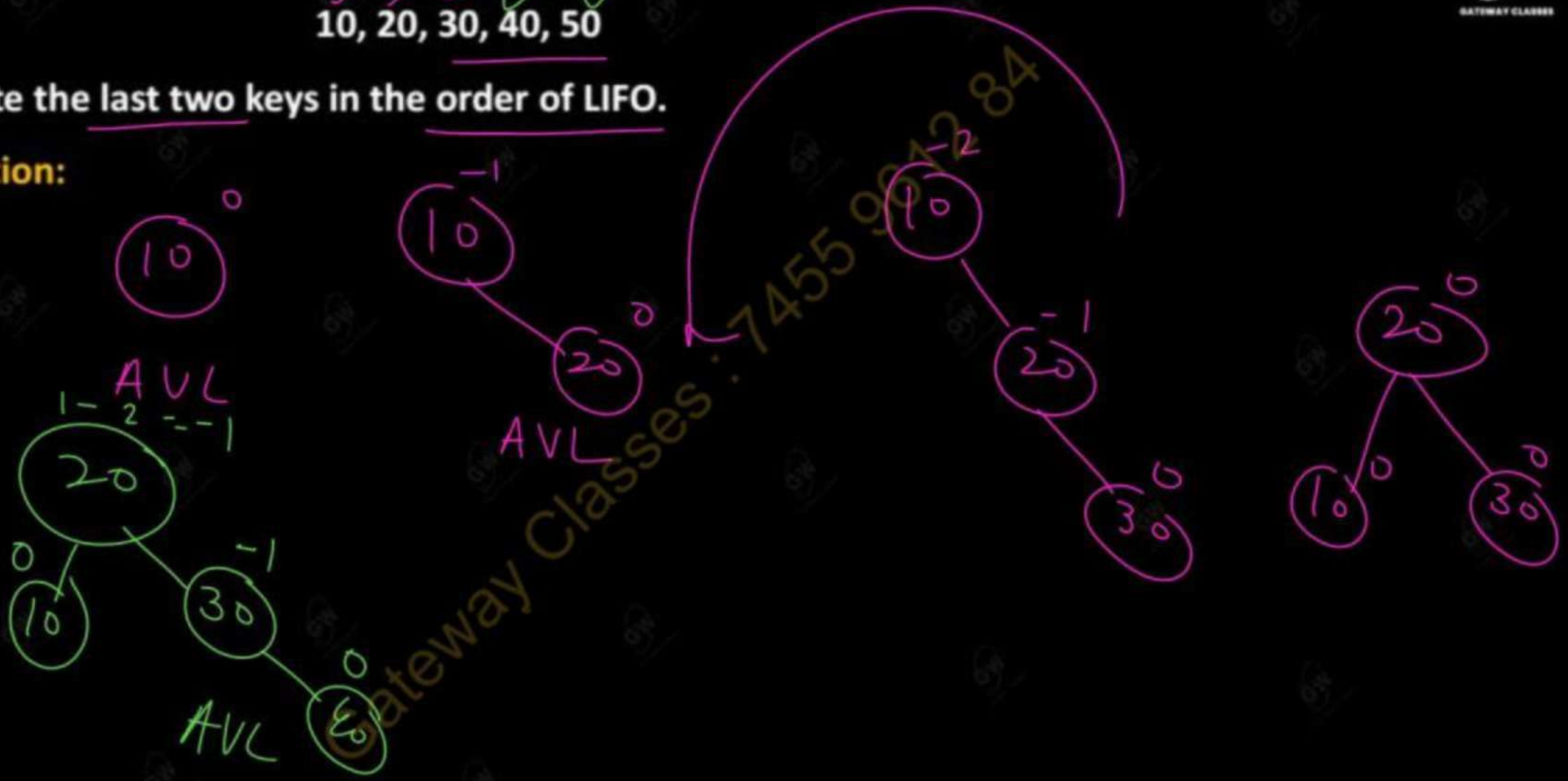
Balanced AVL Search Tree

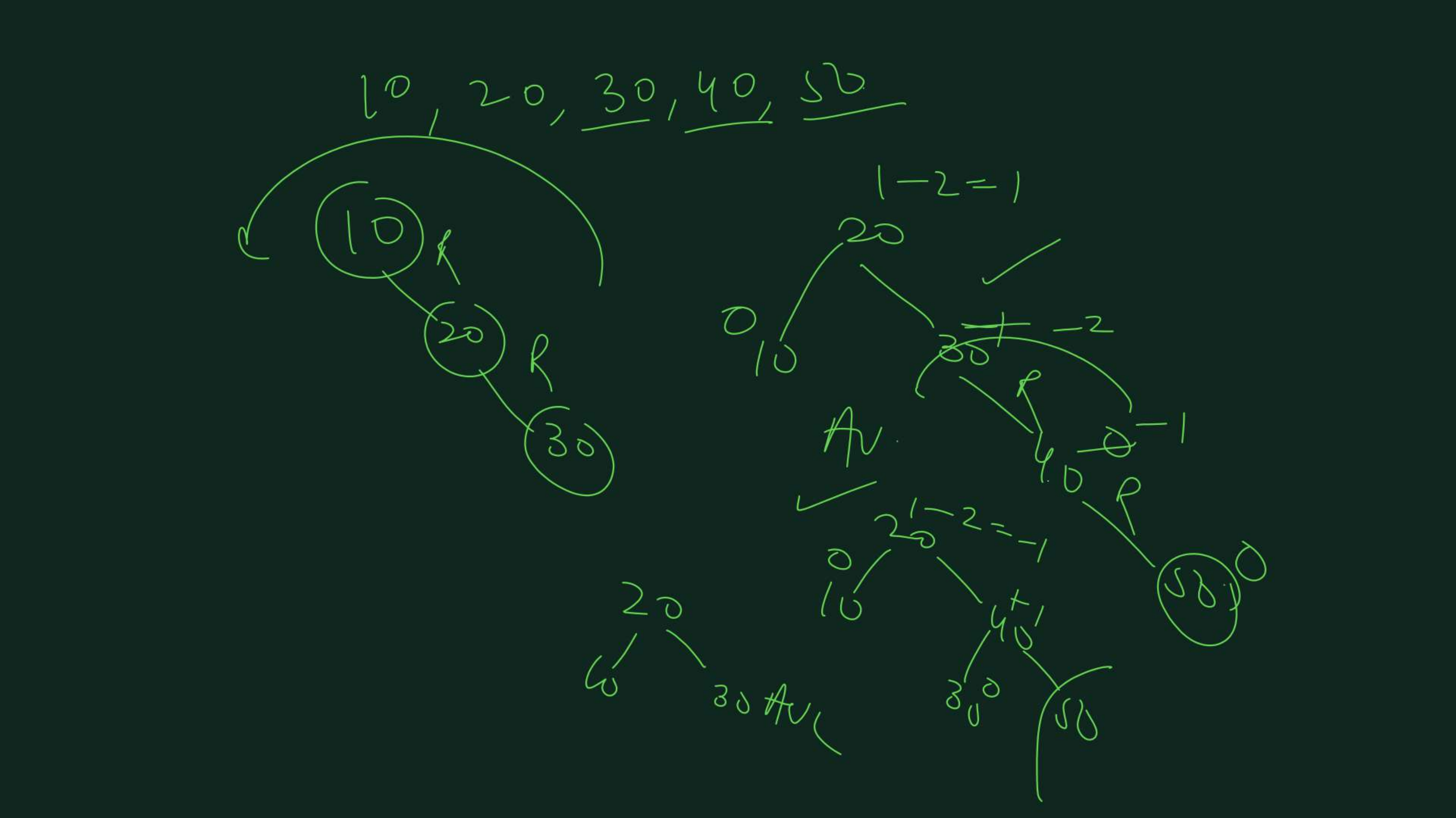
Example: Insert the following keys in order shown to construct an AVL tree

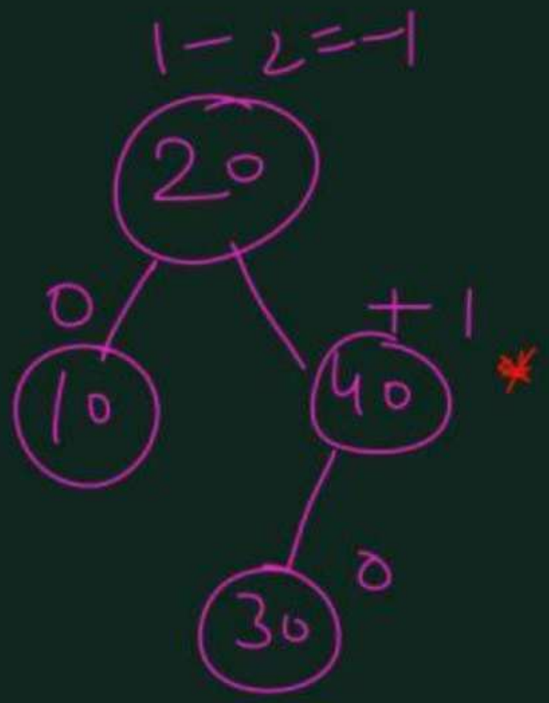
10, 20, 30, 40, 50

Delete the last two keys in the order of LIFO.

Solution:




10, 20, 30, 40, 50
10
R
20
R
30
1-2=1
20
10
-2
30
R
AV
-1
40
R
1-2=-1
20
10
+1
40
30
50
50
20
40
30 AVL



AVL

AVL T

B-Tree

Q.8 Construct a B tree of order 4 using following elements:

K, U, W, C, M, P, Y, A, E, Q, X, D, H, V, F, J, I, B, S, T

2020-21, 7 marks

Q.9 (i) Insert the following keys into an initially empty B tree of order 5

a, g, f, b, k, d, h, m, j, e, s, l, r, x, c, l, n, t, u, p

(ii) What will be the resultant B-tree after deleting keys j , t and d in sequence?

2021-22, 10 marks

Q.10 What is B-tree? Write the various properties of B-tree. Show the results of inserting the keys F, S, Q, K, C, L, H, T, V, W, R, N, P, A, B in order in to an empty B-tree of order 5.

2022-23, 10 marks

B Tree

The B-tree of order n can be defined as-

1. A B-tree is a balanced m-way tree.
2. A B-tree is also known as balanced sorted tree.
3. It find its use in external sorting.
4. It is not a binary tree.
5. To reduce disk accesses, several conditions of the three must be true.

(i) the height of the tree must be kept to a minimum.

(ii) There must be no empty sub-trees above the leaves of the tree.

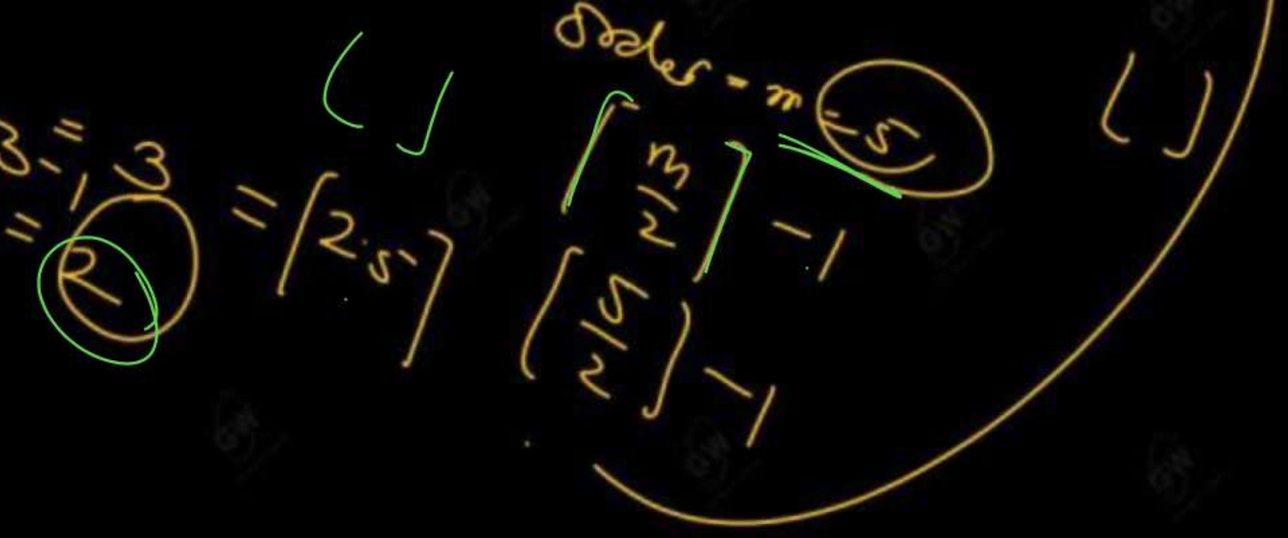
(iii) The leaves of the tree must all be on the same level; and

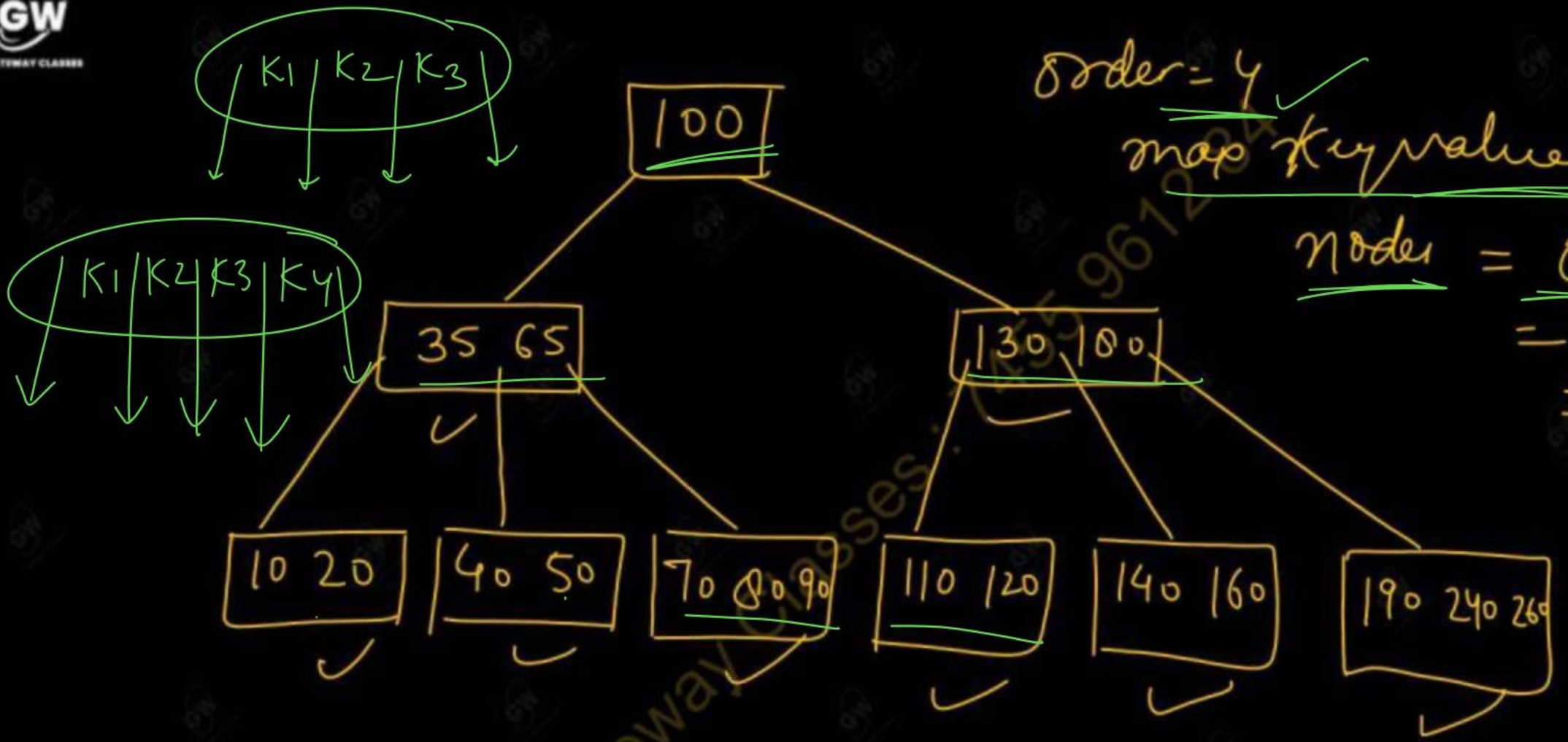
(iv) All nodes except the leaves must have at least some minimum number of children.

B tree has the following property:

1. All leaf nodes will be at same level.
2. Every node has maximum m children where m is order of b tree.
3. Every node has maximum m-1 keys.
4. Every node has minimum; root node = 1 key and all other node = Ceil (m/2) – 1 key. For example order = 5 the except root node all other node must have minimum 2 keys.
5. Keys in non leaf node will divide the left and right sub tree where value of left subtree keys will be less and the value of the right sub tree keys will be more than that particular key.

Let us take a B-tree of order 4:





Order = 4

max key values in a node = Order - 1

= m - 1

= 4 - 1

= 3

- Here we can see all leaf nodes are at same level.
- All non leaf nodes have no empty sub tree and they have keys 1 less than number of their children.

Insertion in B-Tree

Algorithm: Insertion _B-Tree

- Insertion will be always in the leaf node only.
- The insertion of a key in a B-tree requires first traversal in B-Tree.
- Through traversal it will find that key to be inserted is already existing or node.
- Suppose key does not exist in tree then through traversal it will reach leaf node.
- Now we have two cases for inserting the key:

1. Node is not full
2. Node is already full

- If the leaf node in which the key is to be inserted is not full, then the insertion is done in the node. A node is said to be full if it contains a maximum of m - 1 keys, where m is the order of the B-Tree.

- If the node is full, then insert the key in order into the existing set of keys in the node, split the node at its median into two nodes at the same level, pushing the median element up by one level.
- It is to be noted that the split nodes are half full.
- Accommodate the median element in the parent node if it is not full.
- Otherwise repeat the same procedure and this may even call for rearrangement of the keys in the root node or the formation of a new root itself.
- Thus since the leaf node are all at same level, the tree grows upward.

Example: Consider building a B tree of degree 4 that is balanced 4-way tree where each node can hold three data values and have four branches. Suppose it needs to contain the following values:

1, 5, 6, 2, 8, 11, 13, 18, 20, 7, 9

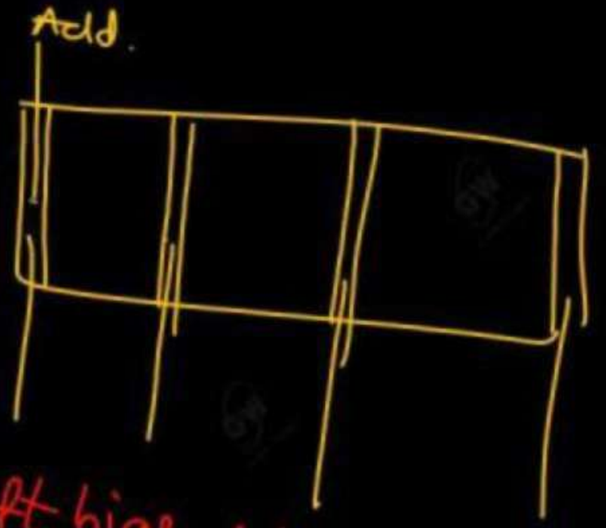
order = 4
max no. of keys in a node = 4 − 1 = 3

Solution :-

$m = 4$

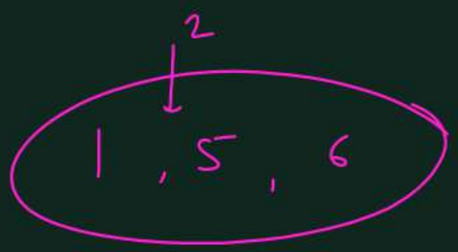
no of keys in a node = order − 1 = 4 − 1 = 3

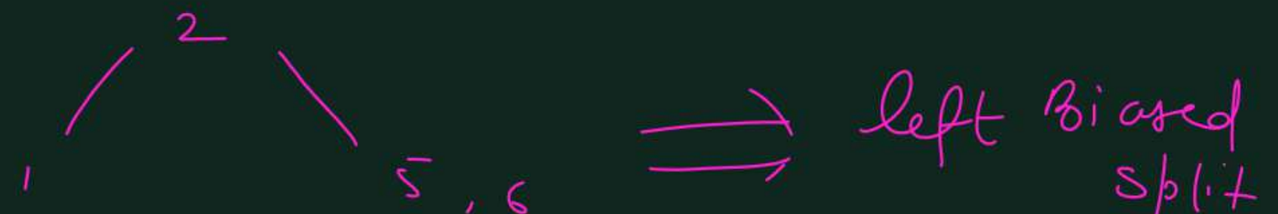
SPLIT

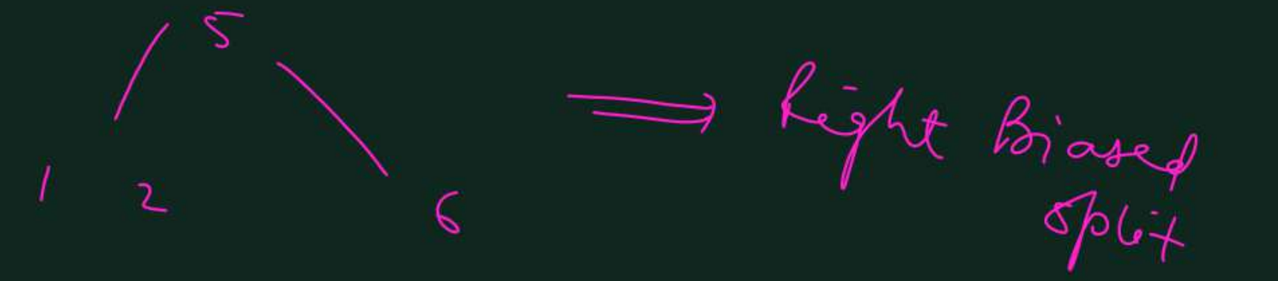


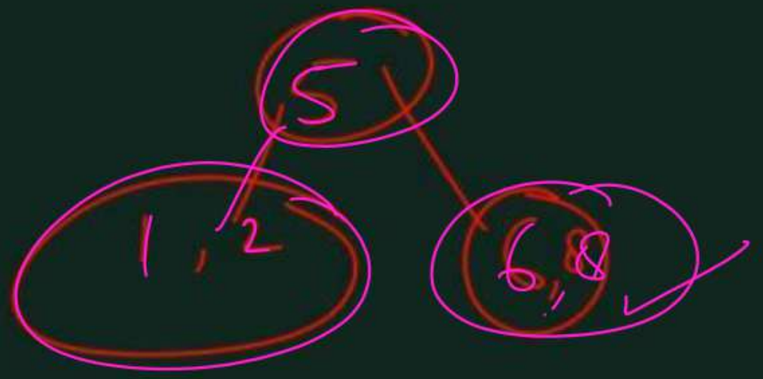
left biased tree splitting

right biased


2
1, 5, 6

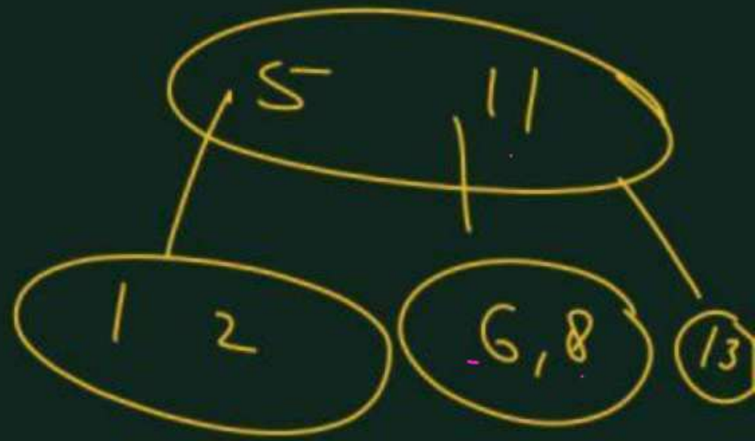

2
1
5, 6
left Biased split

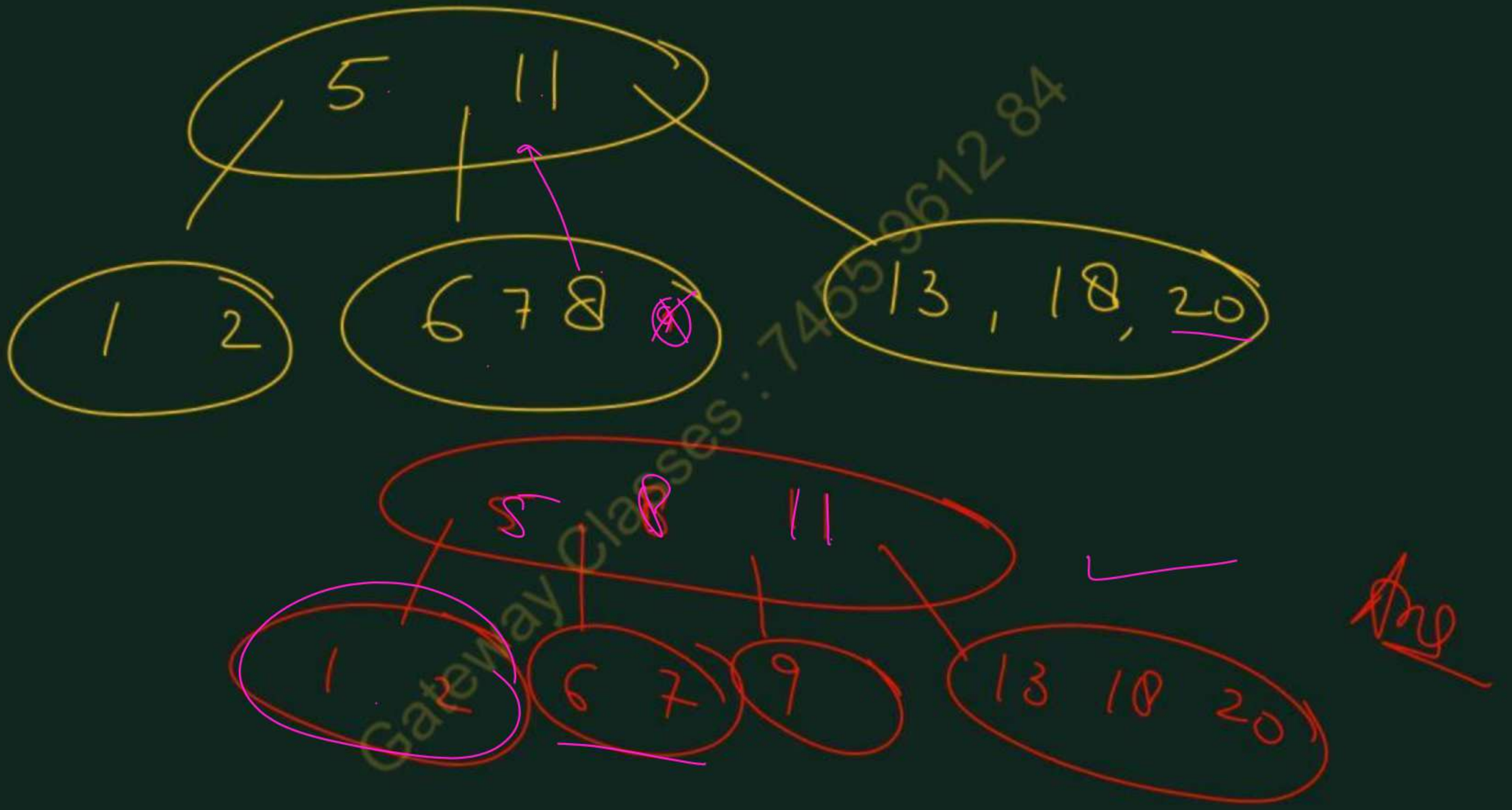

5
1 2
6
Right Biased split



right biased

13





order = 5
max = 4
min = ceil($\frac{5}{2}$) − 1 = 3 − 1 = 2

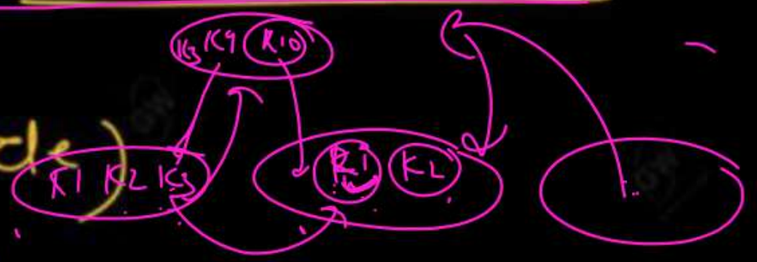
order = 4
max = 3
min = ceil($\frac{4}{2}$) − 1 = 2 − 1 = 1

Deletion in B-tree

- Deletion of key also requires first traversal in B-tree and after reaching on particular node, two cases may occur-

(i) Node is leaf node (Last node, External node)

(ii) Node is non leaf node (Internal node)

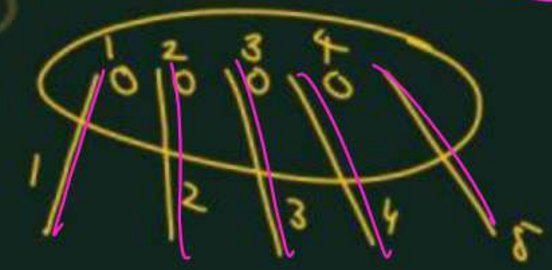


- For the first case suppose node has more than minimum number of keys then it can be easily deleted. But suppose it has only minimum number of keys then first we will see the number of keys in adjacent leaf node if it has more than minimum number of keys then first key of the adjacent node will go to the parent node and the key in parent node which partitioning will be combined together in one node. Suppose now parent has only less than the minimum number of keys then the same thing will be repeated until it will get the node which has more than the minimum number of keys.

Order = m (5)

(i) a node will contain max elements / key = Order - 1 = m - 1 = 5 - 1 = 4

(ii) max no. of children for a node = order = 5

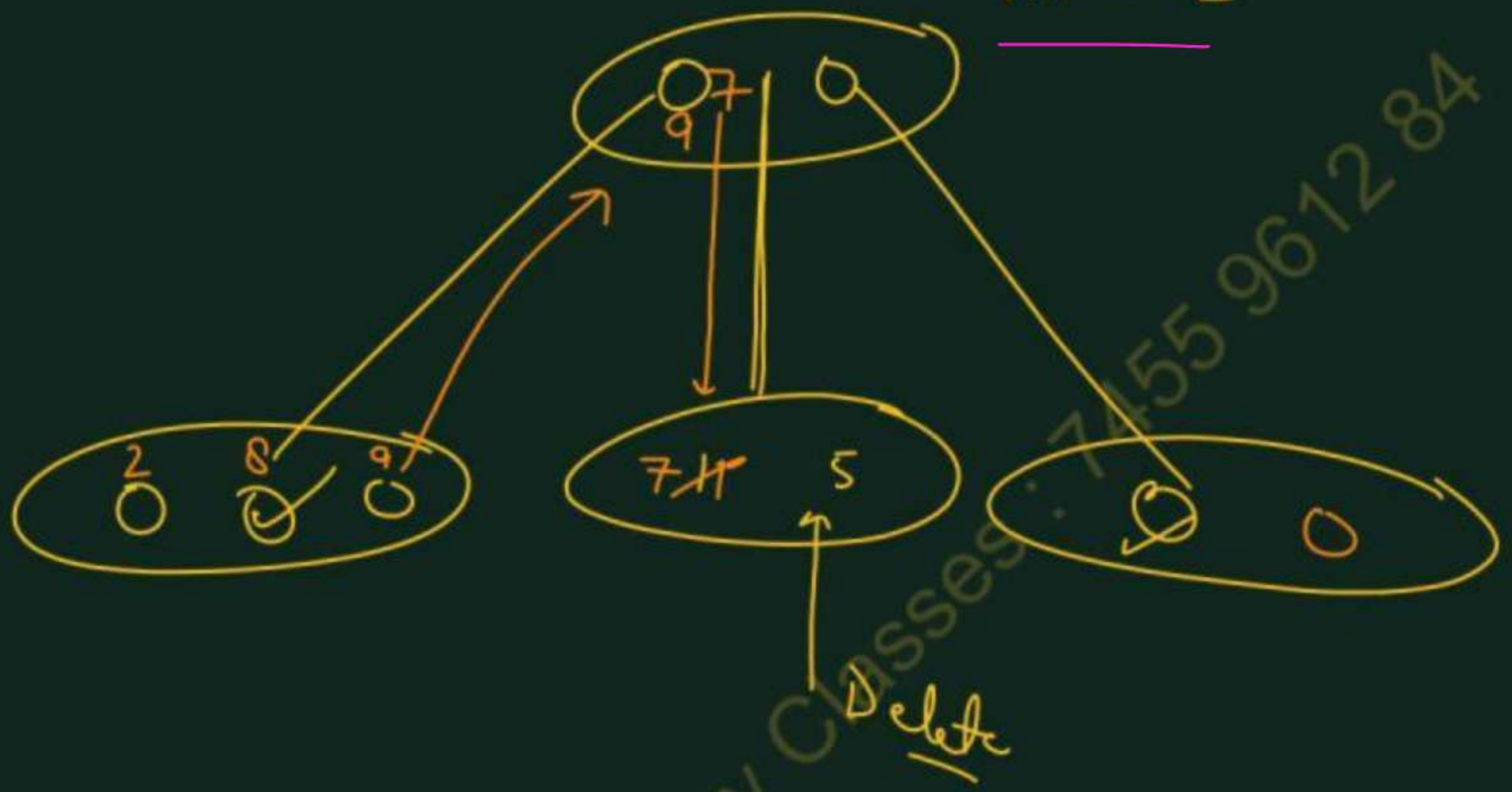


(iii) B-Tree minimum number of elements | keys in node

$$= \text{Ceil}\left[\frac{\text{order}}{2}\right] - 1$$

$$= \left\lceil \frac{5}{2} \right\rceil - 1 = [2.5] - 1$$

$$= 3 - 1 = 2$$

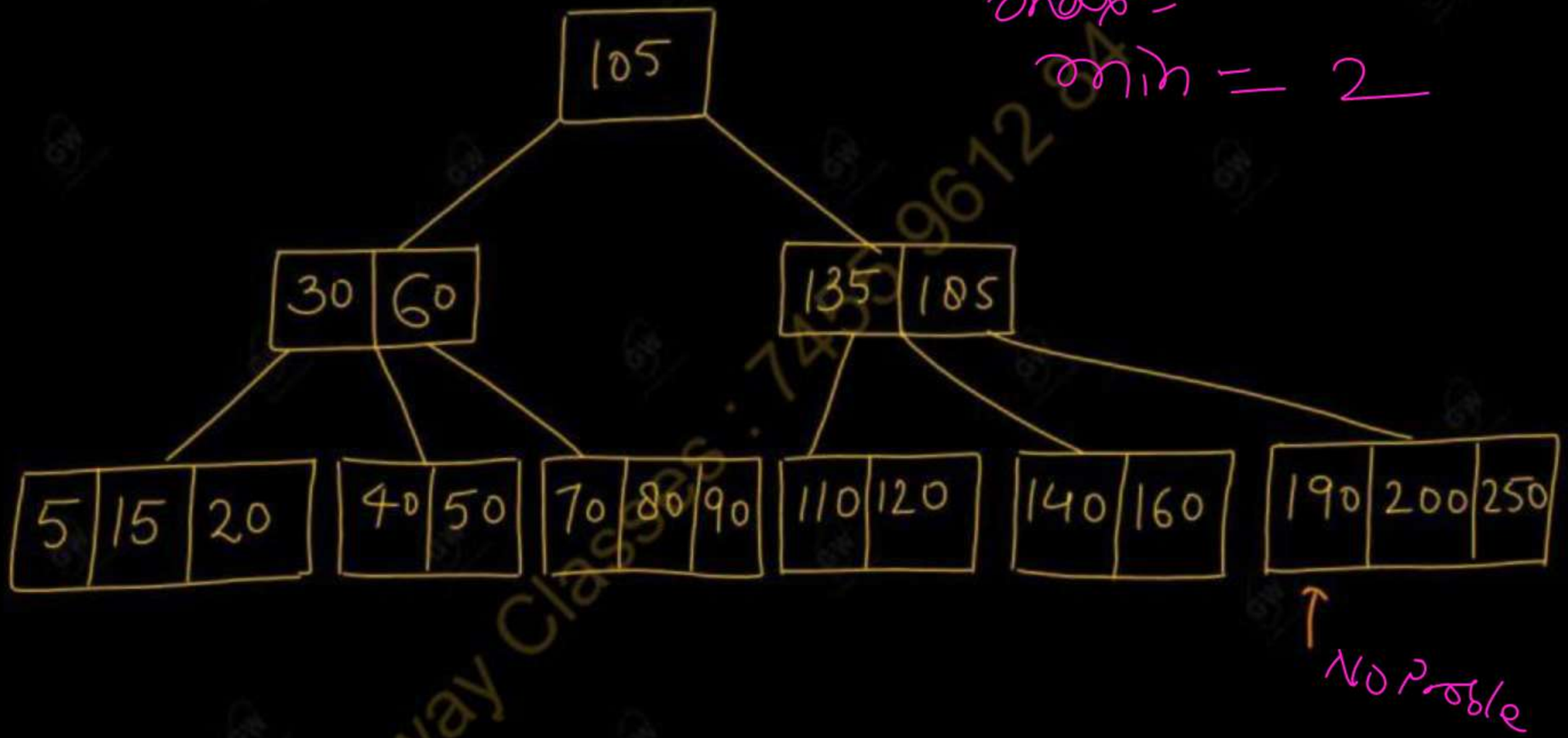


- For the second case key will be deleted and its predecessor or successor key will come on its place. Suppose both nodes predecessor and successor key have minimum number of keys then the nodes of predecessor and successor keys will be combined.
- Thus while removing a key from a leaf node, if the node contains more than the minimum number of keys, then key can be easily removed. If the leaf node contains just the minimum number of elements, then borrow for an element from either the left sibling node or right sibling node to fill the vacancy to make there minimum number of nodes. If the left sibling node has more than the minimum number of keys, pull the largest key up into the parent node and move down the intervening entry from the parent node to the leaf node where the key is to be deleted. Otherwise, pull the smallest key of the right sibling node to the parent node and move down the intervening parent element to the leaf node.

- If both the sibling nodes have only minimum number of entries, then create a new leaf node out of the two leaf nodes and the intervening element of the parent node, ensuring that the total number does not exceed the maximum limit for a node. If while borrowing the intervening element from the parent node, it leaves the number of keys in the parent node to be below the minimum number, then we propagate the process upwards ultimately resulting in a reduction of height of the B-tree.

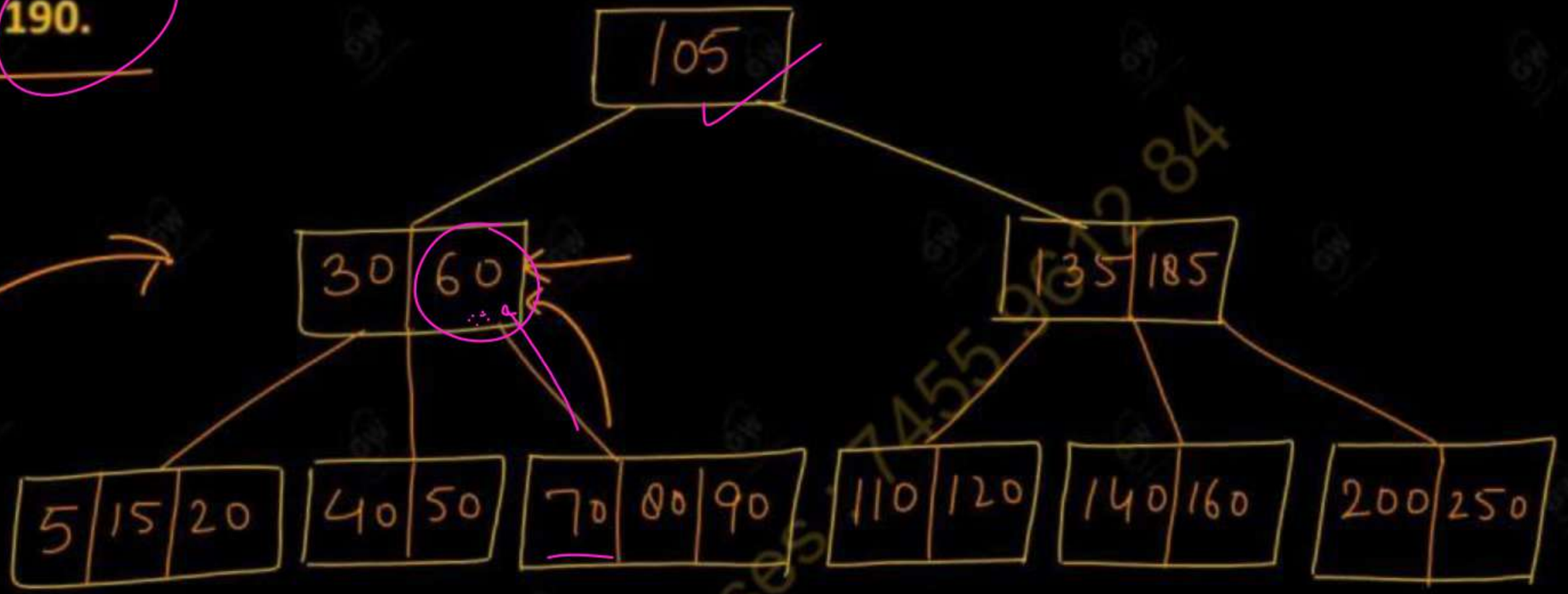
Example:- Let us take a B-tree of order 5.

order = 5
max =
min = 2



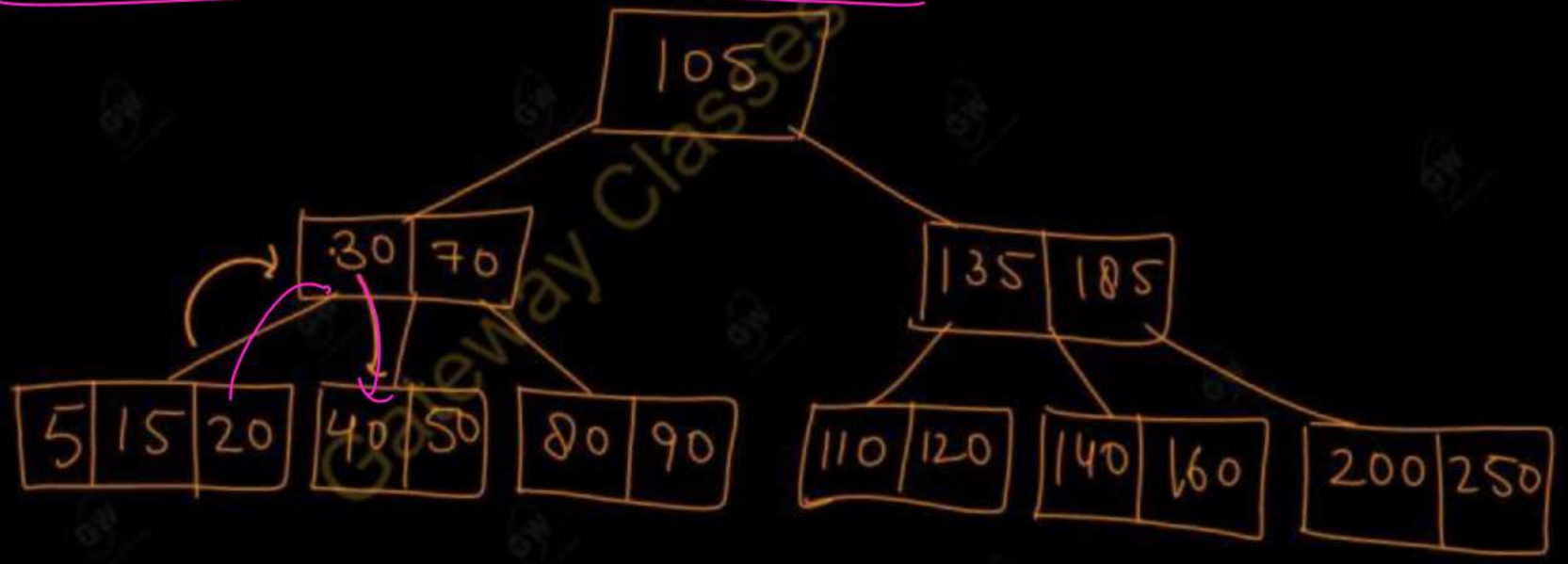
- Order 5 means node will have maximum number of keys = 4 (5-1)
- And node will have minimum number of keys = $\lceil \frac{5}{2} \rceil - 1 = 2$ {except the root}

Delete 190.

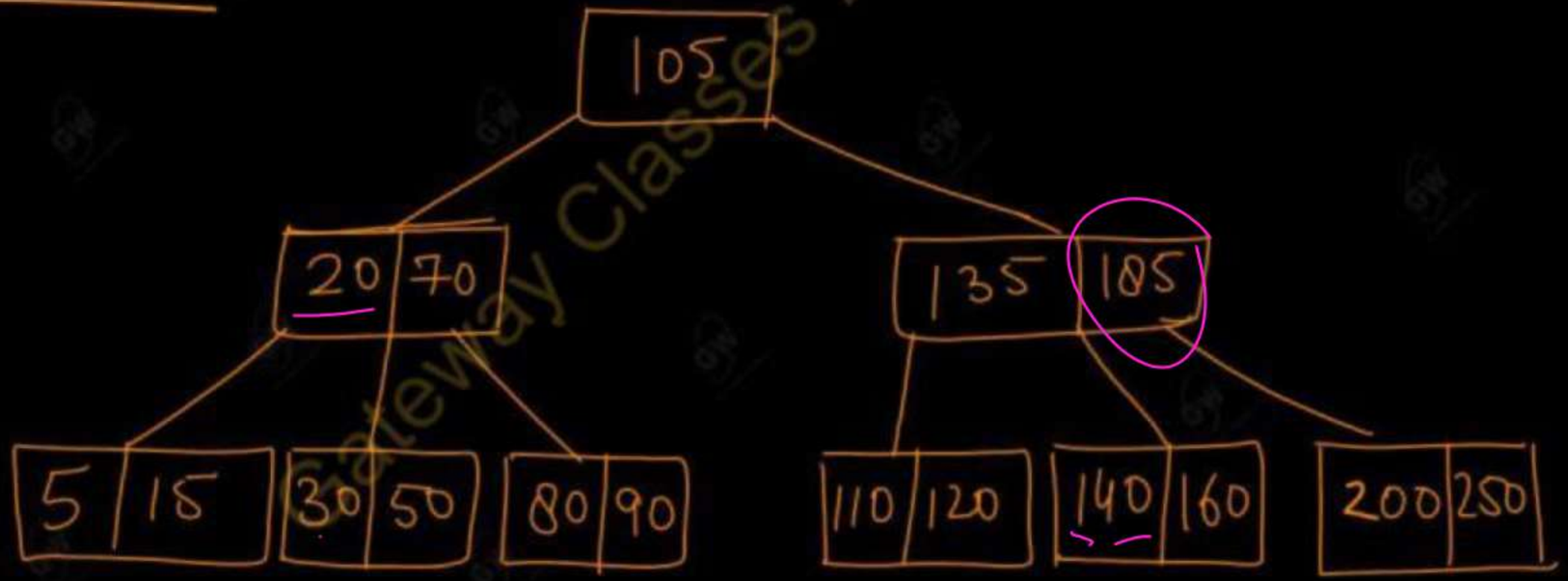


- Here 190 is in leaf node and after deleting 190 the node will have minimum number of keys so it can be deleted easily from the leaf node.
- After deletion of 190, the B tree will be as follows-

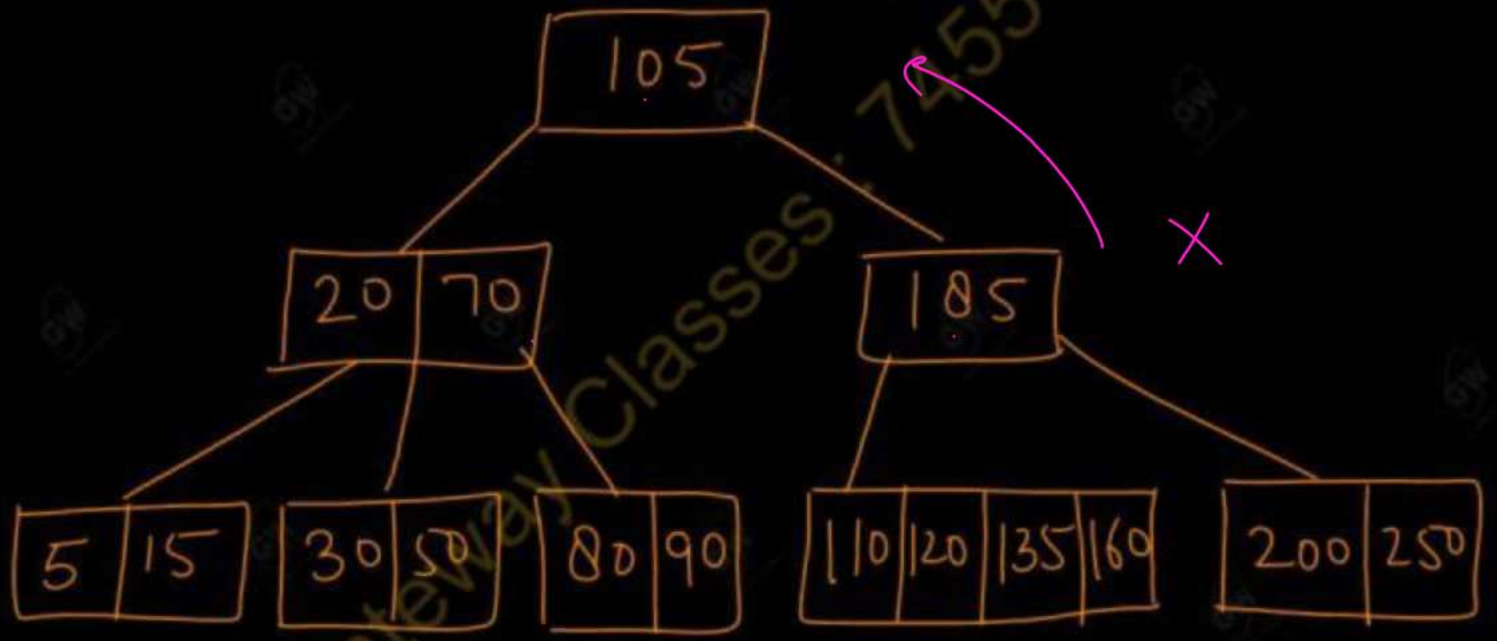
Delete 60 - Here 60 is non leaf node. This node is having minimum number of keys i.e. 2 and after deletion it will violate the rule of B tree so we have to do something to maintain minimum number of keys in this node. So first 60 will be deleted from the node and then the minimum element of right node will come in that node, that element is 70. After moving 70 to upward node, still it will maintain the minimum number of keys in a node. After this adjustment tree will be as follows:



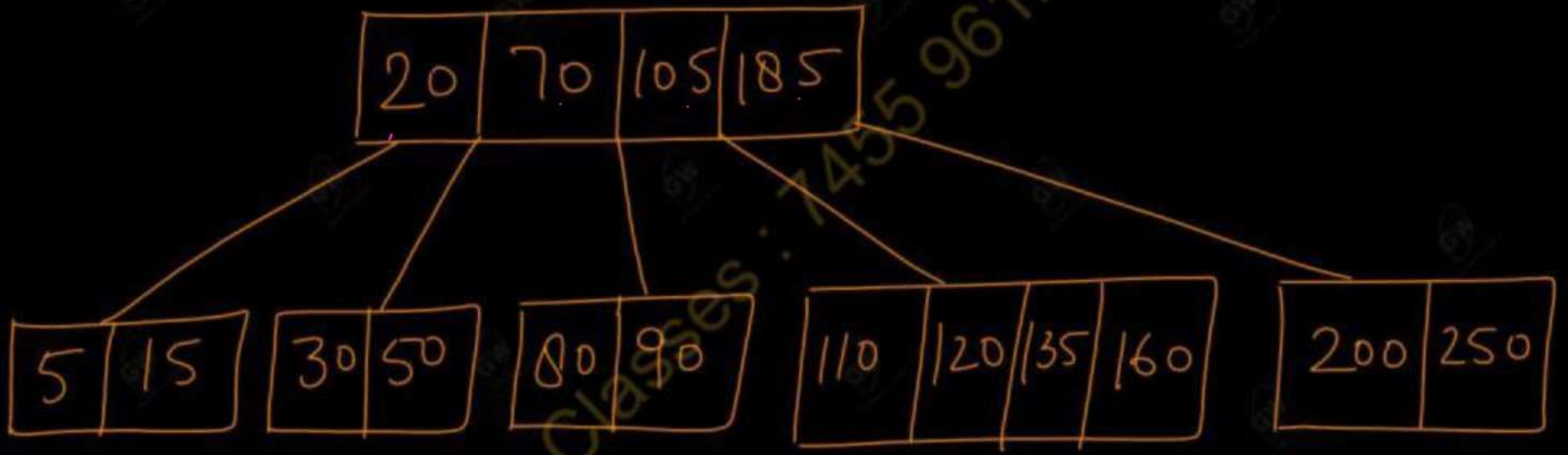
Delete 40 - Here 40 is in leaf node. This node is having minimum number of keys i.e. 2 and after deletion it will violate the rule of B tree so we have to do something to maintain minimum number of keys in this node. So first 40 will be deleted from the leaf node and then the maximum key of the left side node will come in the parent node and the minimum key od parent node will come in leaf node. After this adjustment the B-tree will be as follows:



Delete 140 - Here 140 is in leaf node. This node is having minimum number of keys i.e. 2 and after deletion it will violate the rule of B tree so we have to do something to maintain minimum number of keys in this node. Its siblings and parent node is also have minimum number of keys. So we have to merge two nodes as shown in this tree.

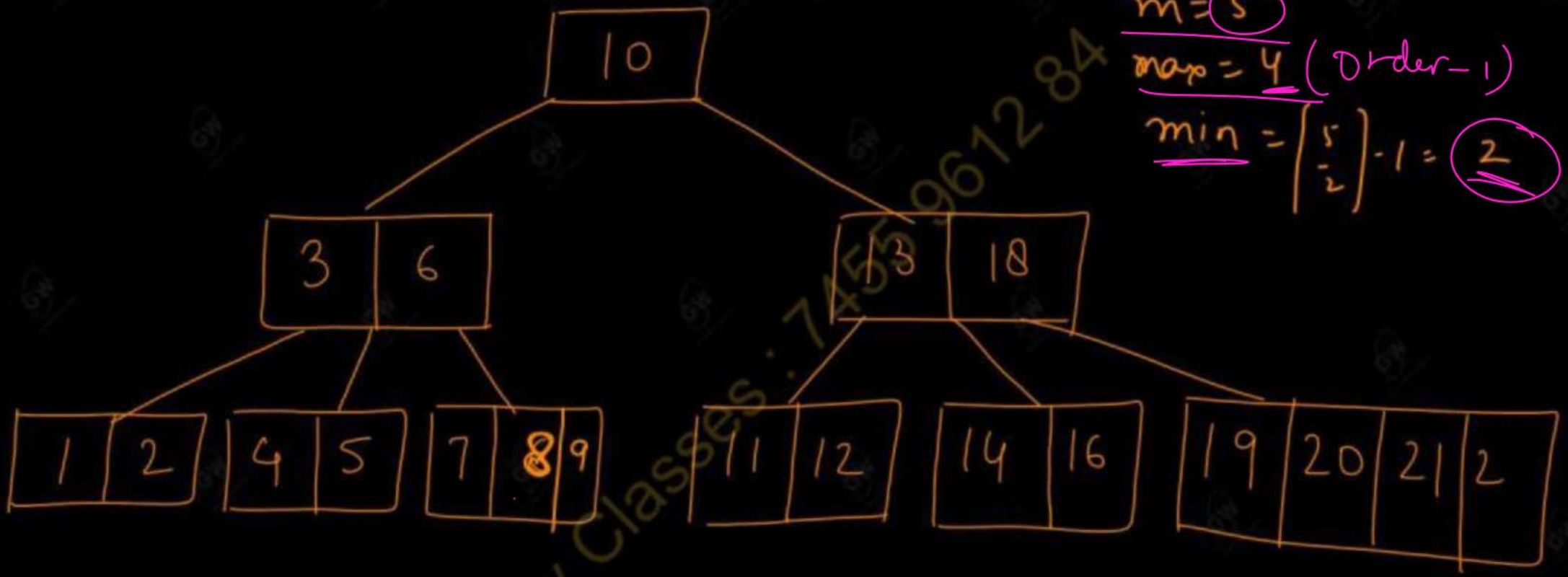


But at least two keys must be in child node (except root node) so again we have to merge the nodes. This is shown in the following figure.



This is the final B-tree after performing all deletions.

Example:- Consider the following B-tree of order 5:



$m = 5$

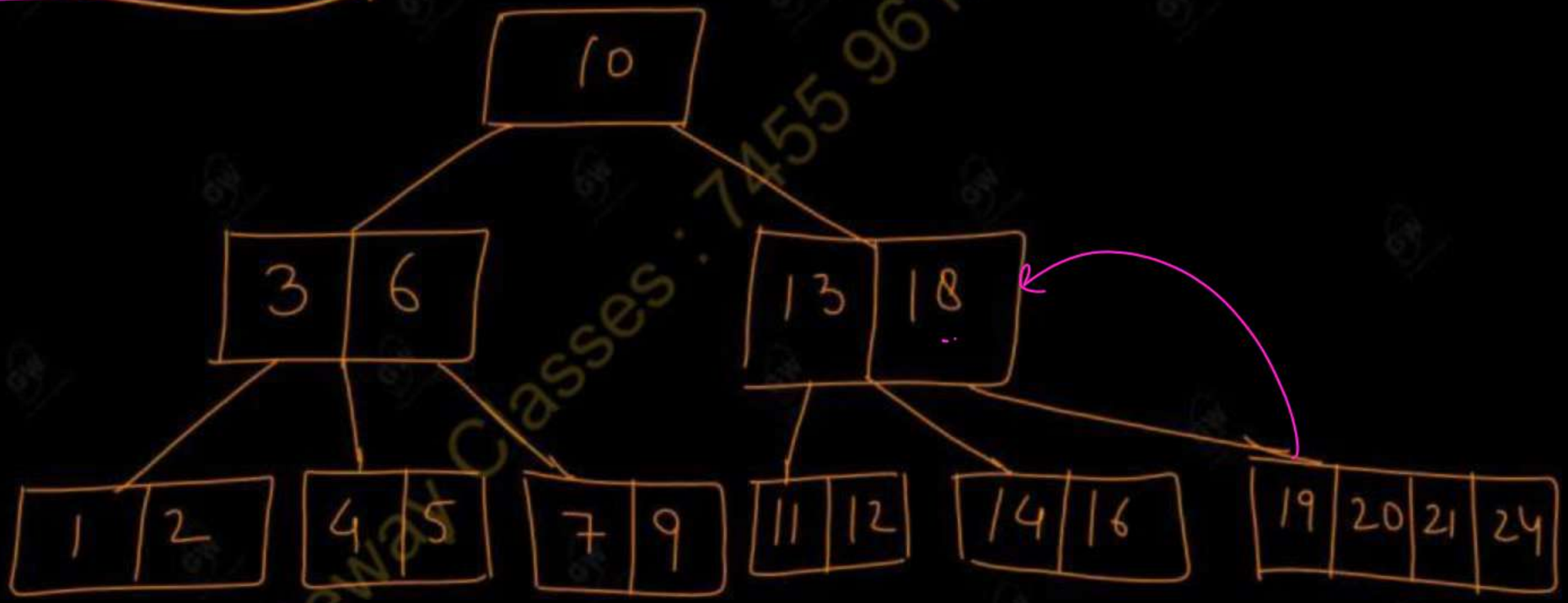
$max = 4$ (Order − 1)

$min = \lceil \frac{5}{2} \rceil - 1 = 2$

- Delete 8, 18, 16 and 4.

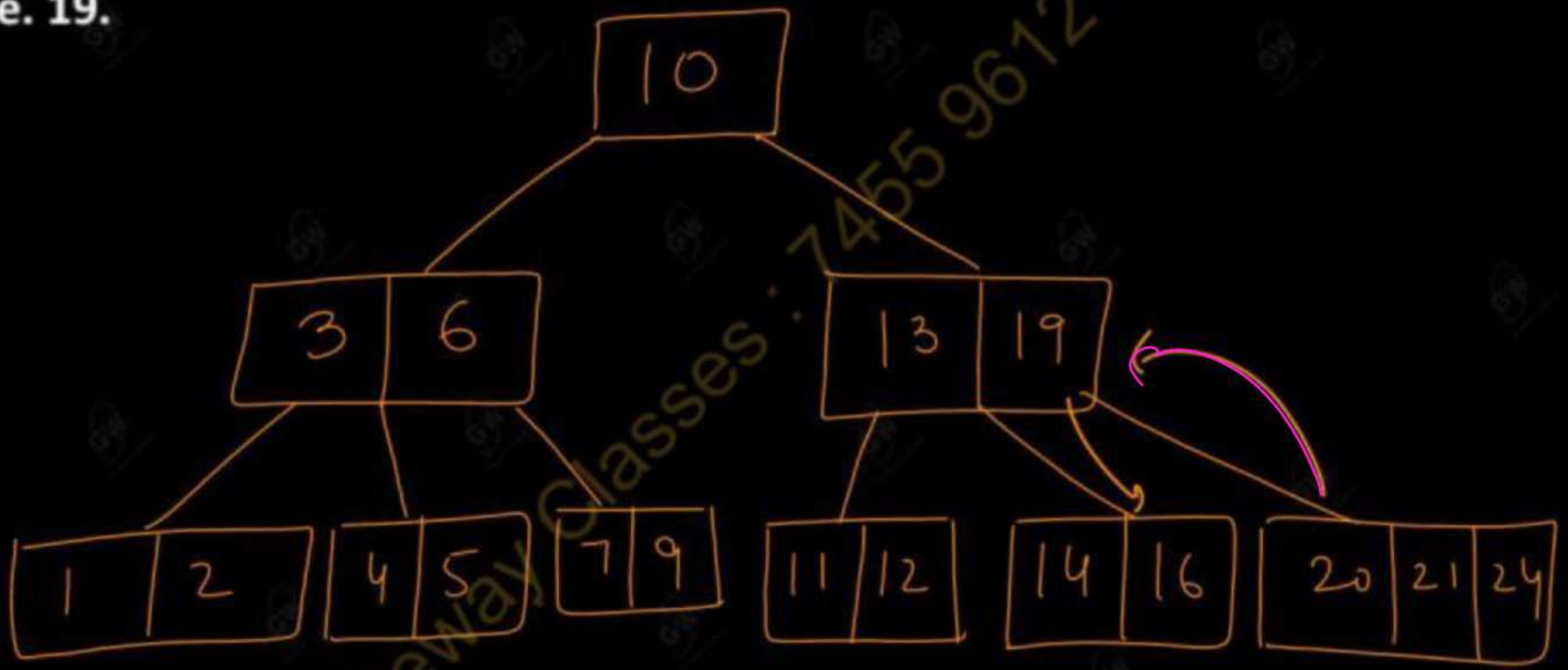
Delete 8:

- Deletion of 8 is from the leaf node with more than the minimum number of elements and hence leaves no problem.



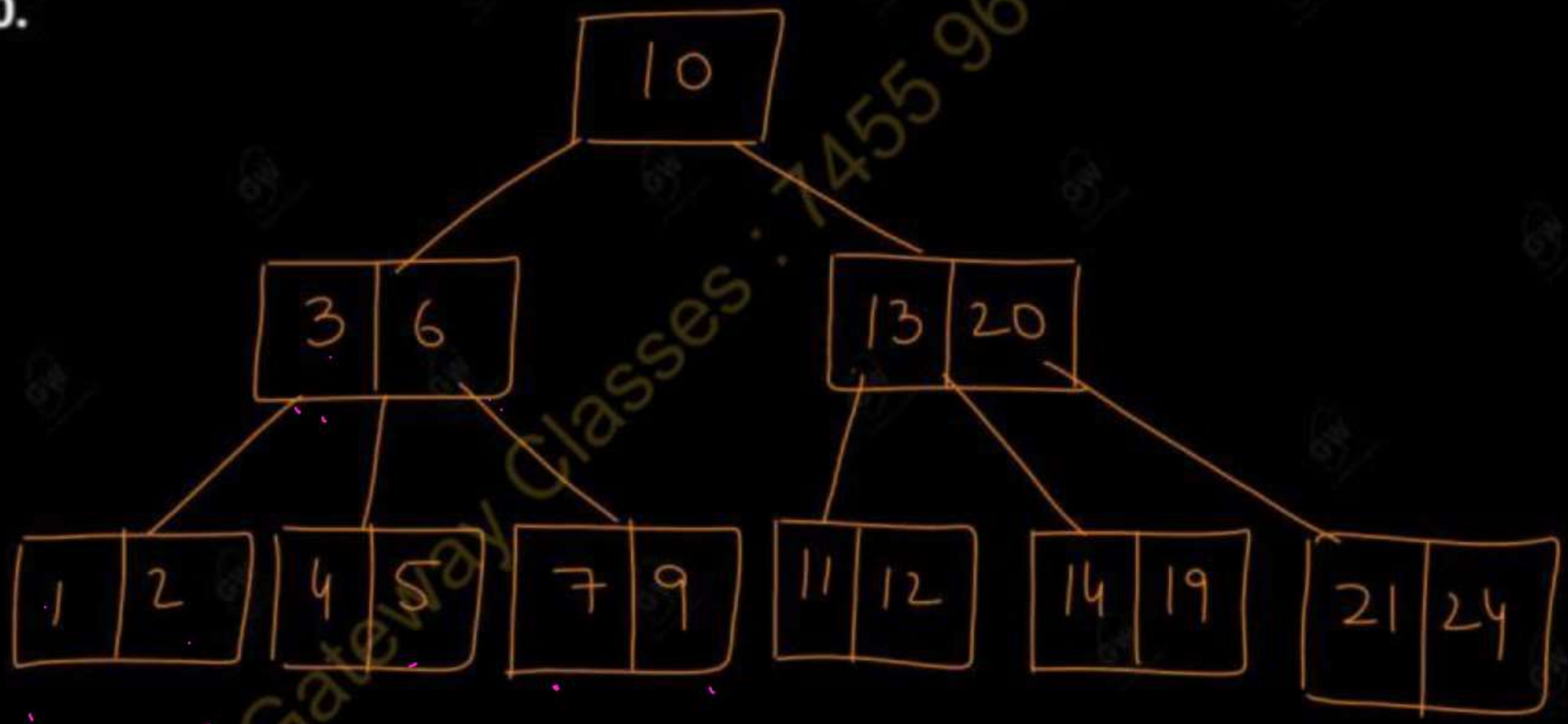
Delete 18:

- Deletion of 18 is from the non leaf node and therefore immediate successor is placed at 18, i.e. 19.



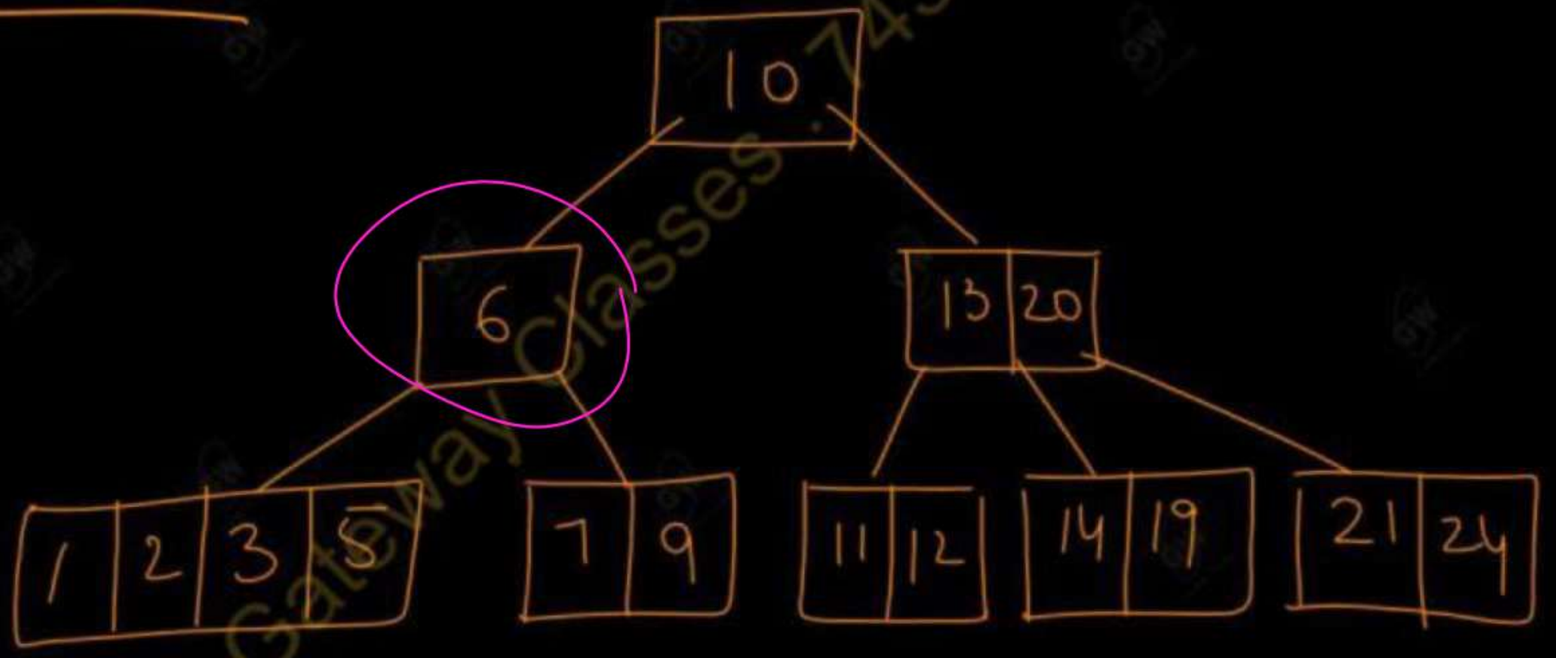
Delete 16:

- Deletion of 16 leaves its nodes with too few elements.
- The element 19 from the parent node is therefore brought down and replaced by the element 20.

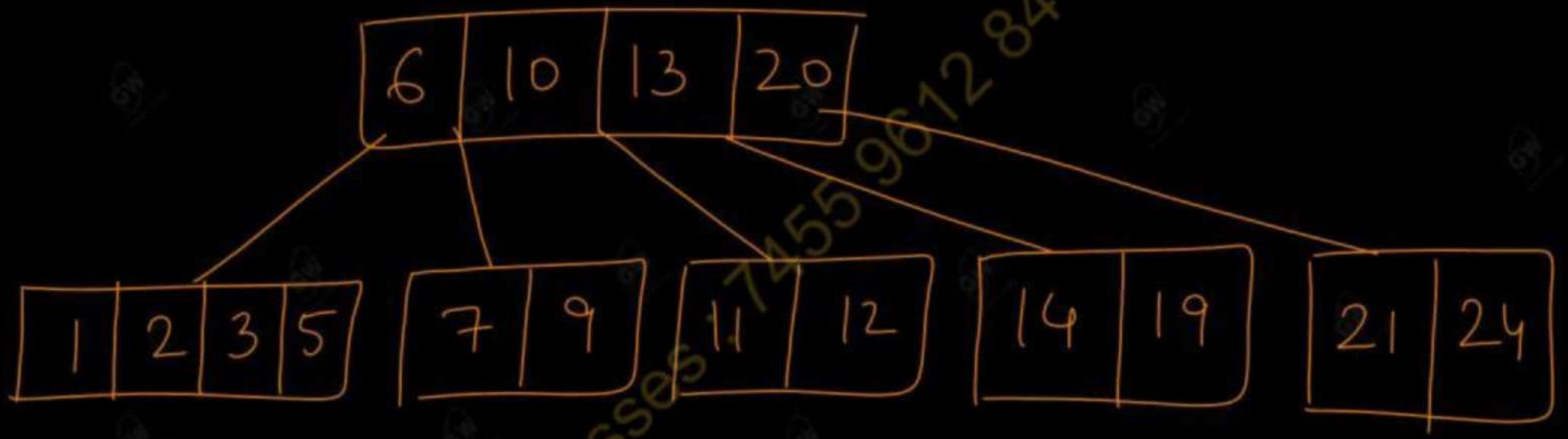


Delete 4:

- Deletion of 4 leaves the node with less than minimum number of elements and neither of its sibling nodes can spare an element.
- The node therefore is combined with one of the sibling and with the median element from the parent node.



- But at least two elements should be in the child of root so it will go in root node.



- It reduces the height of the tree also.

Download Gateway Classes Application
From Google Play store
Link in Description

Next Lecture:- Unit – 5, Graph

Thank You