

**AVLTREE**

(ADELSON-VELSKY AND LANDIS)

*OR*

**SELF-BALANCING BINARY**  
**SEARCH TREE**

*OR*

**HEIGHT BALANCED TREE**

# AVL TREES

- AVL tree is a ***self-balancing binary search tree*** in which the heights of the two sub-trees of a node may differ by at most one.
- Because of this property, AVL tree is also known as a ***height-balanced tree***.
- The key advantage of using an AVL tree is that it takes  $O(\log n)$  time to perform search, insertion and deletion operations in average case as well as worst case (because the height of the tree is limited to  $O(\log n)$ ).
- The structure of an AVL tree is same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the ***Balance Factor***.

# AVL TREES

- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

***Balance factor = Height (left sub-tree) – Height (right sub-tree)***

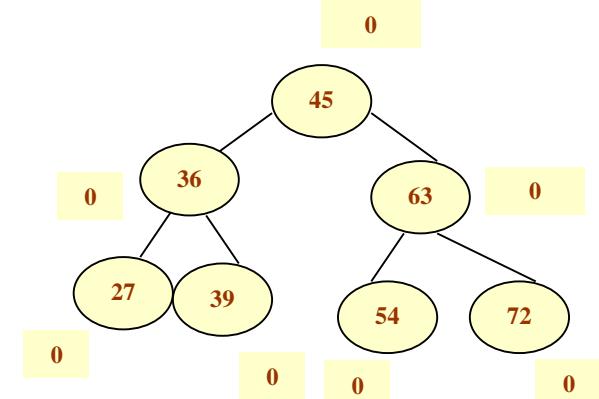
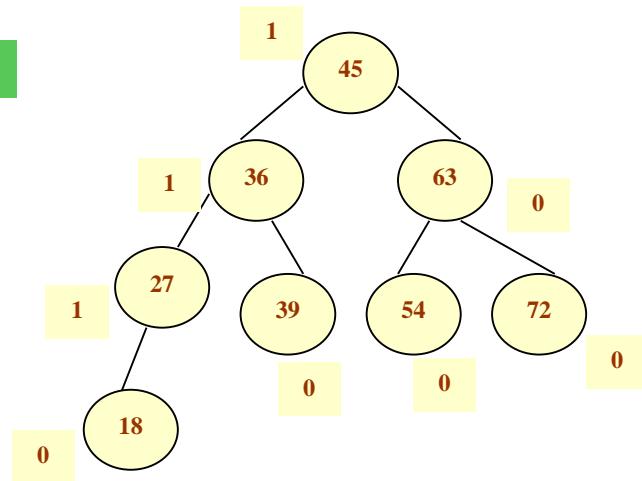
A binary search tree in which every node has a balance factor of **-1, 0 or 1** is said to be **height balanced**. A node with any **other balance factor** is considered to be **unbalanced and requires rebalancing**.

# AVL TREES

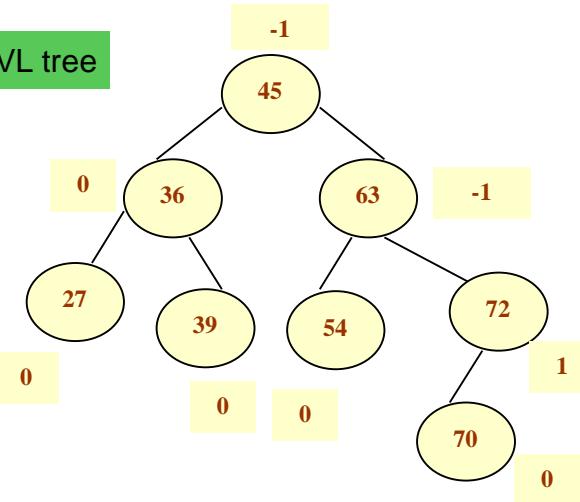
- If the **balance factor of a node is 1**, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is called ***Left-heavy tree.***
- If the **balance factor of a node is 0**, then it means that the height of the left sub-tree is equal to the height of its right sub-tree. Such a tree is called ***Balanced tree.***
- If the **balance factor of a node is -1**, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is called ***Right-heavy tree.***

# AVL TREES

Left heavy AVL tree



Right heavy AVL tree



# AVL TREE

Various operation can be performed on AVL Tree

- Search
- Insertion
- Deletion

# SEARCHING FOR A NODE IN AN AVL TREE

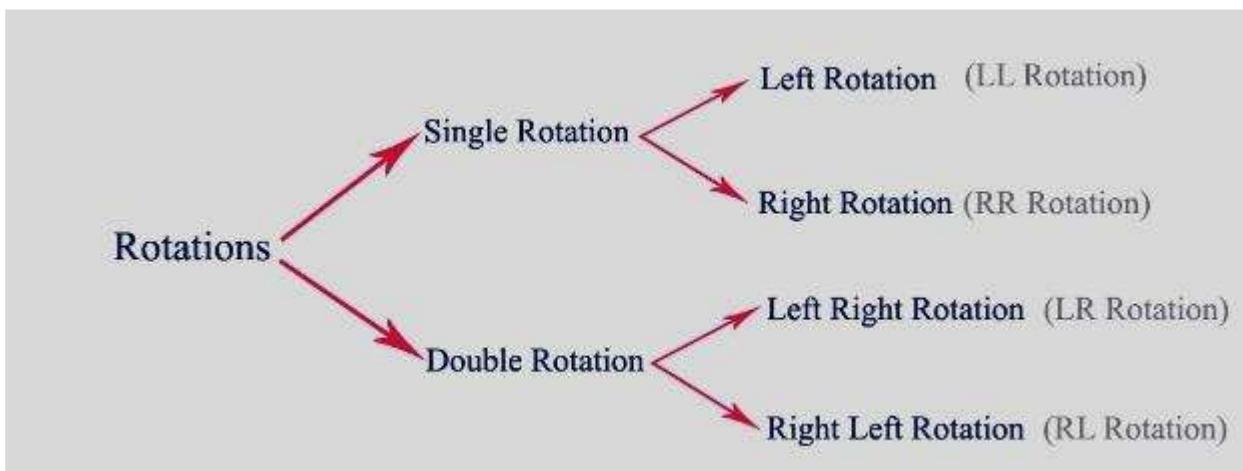
- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Because of the height-balancing of the tree, the search operation takes **O(log n)** time to complete.
- Since the operation does not modify the structure of the tree, no special provisions need to be taken.

# INSERTING A NODE IN AN AVL TREE

- Insertion is also done in the same way as it is done in case of a binary search tree.
- Like in binary search tree, the new node is **always inserted as the leaf node**.
- But the step of insertion is usually followed by an additional step of rotation.
- **Rotation is done to restore the balance of the tree.** However, if insertion of the new node does not disturb the balance factor, that is, **if the balance factor of every node is still -1, 0 or 1, then rotations are not needed.**
- **The nodes whose balance factors will change are those which lie on the path between the root of the tree and the newly inserted node.**

# AVL TREE ROTATIONS

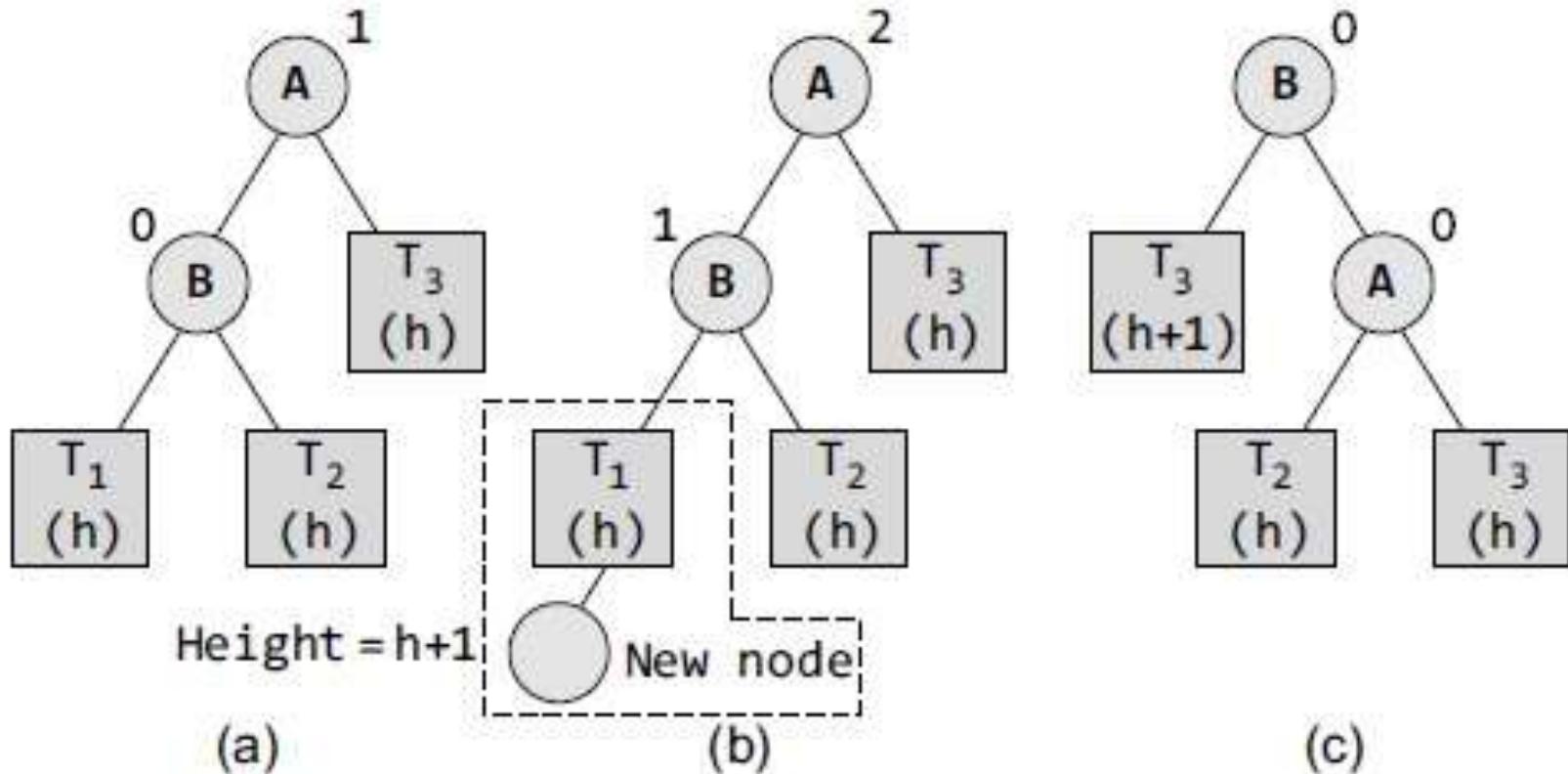
- Rotation is the process of moving the nodes to either left or right to make tree balanced.
- To perform rotation, our first work is to find the ***critical node***.
- ***Critical node*** is the nearest ancestor node on the path from the root to the inserted node whose balance factor is neither -1, 0 nor 1.
- The second task is to determine which type of rotation has to be done.
- There are four types of rebalancing rotations



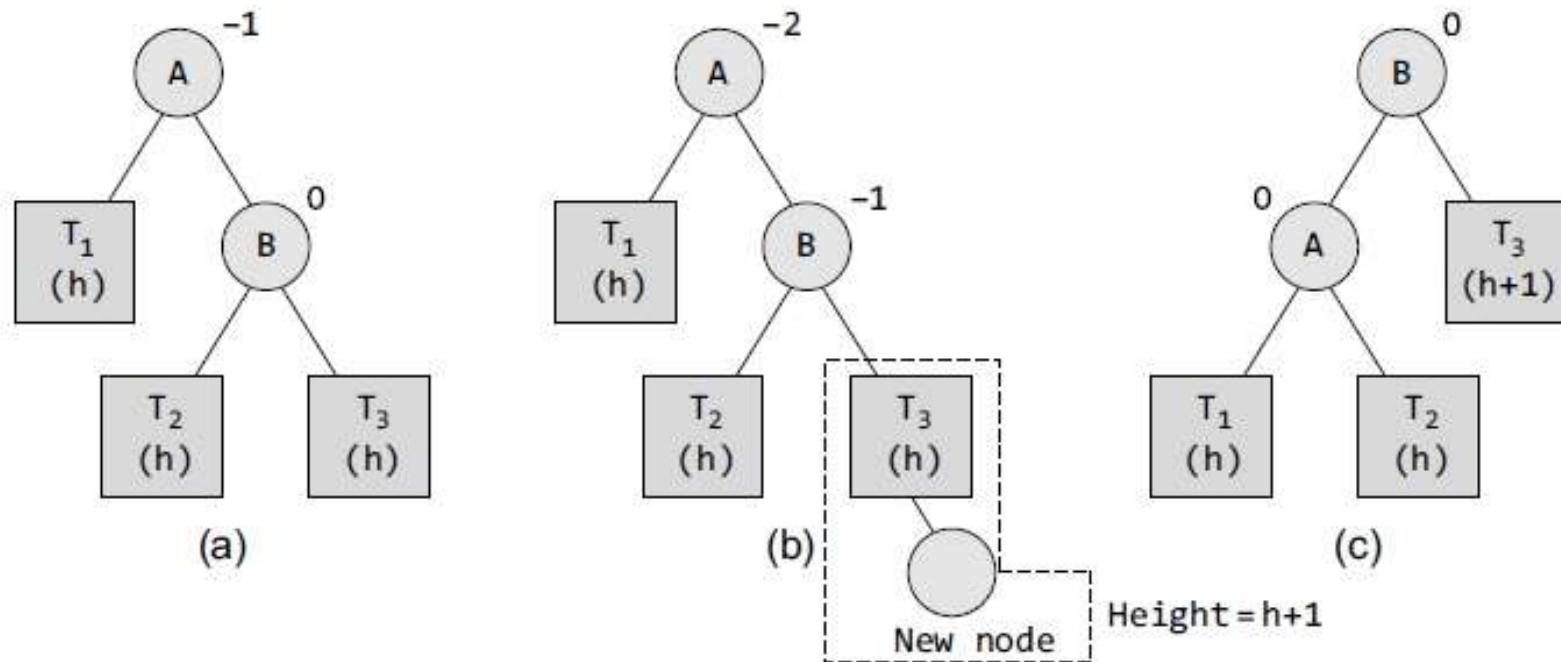
# AVL TREE ROTATIONS

- ***LL rotation:*** the new node is inserted in the **left** sub-tree of the **left child of the critical node**
- ***RR rotation:*** the new node is inserted in the **right** sub-tree of the **right child of the critical node**
- ***LR rotation:*** the new node is inserted in the **right** sub-tree of the **left child of the critical node**
- ***RL rotation:*** the new node is inserted in the **left** sub-tree of the **right child of the critical node**

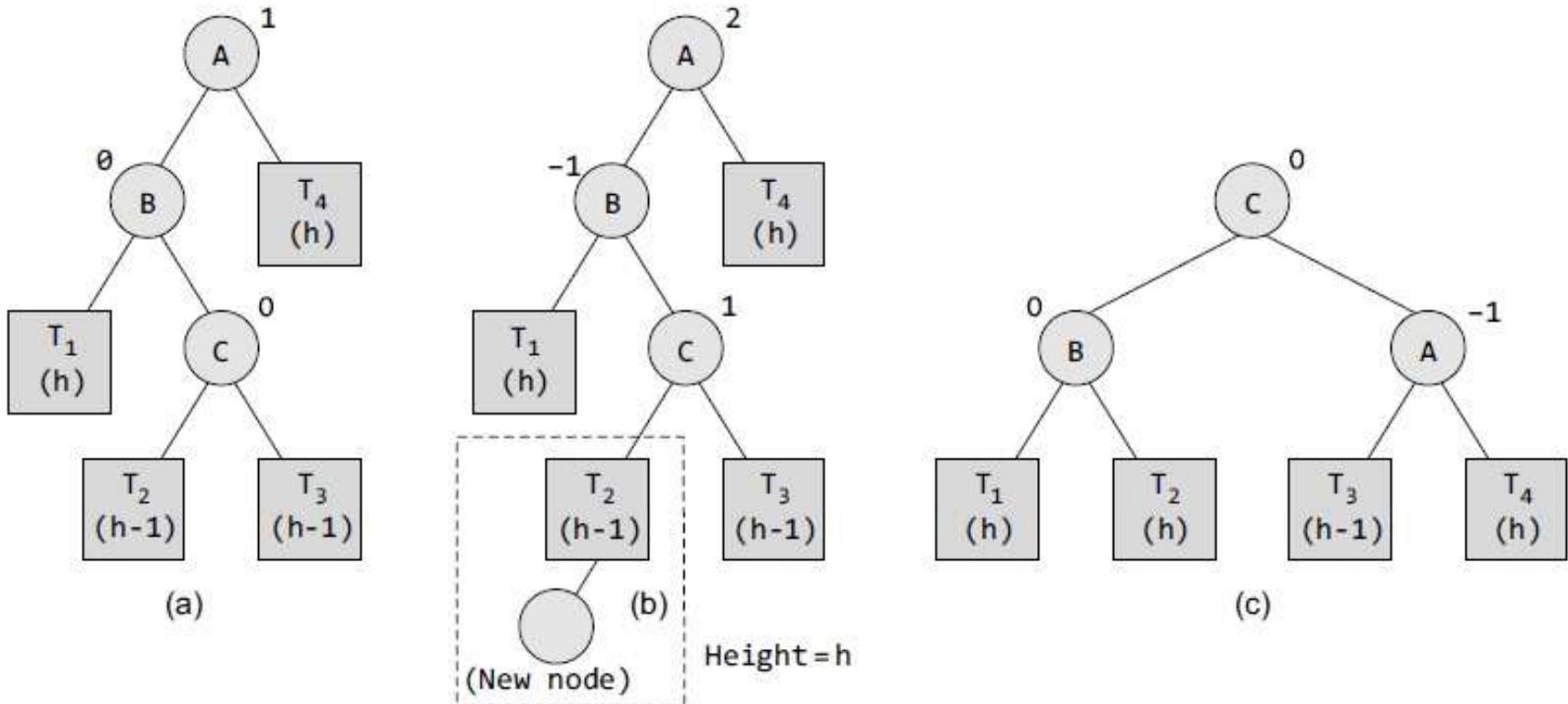
# AVL TREE ROTATIONS - LL



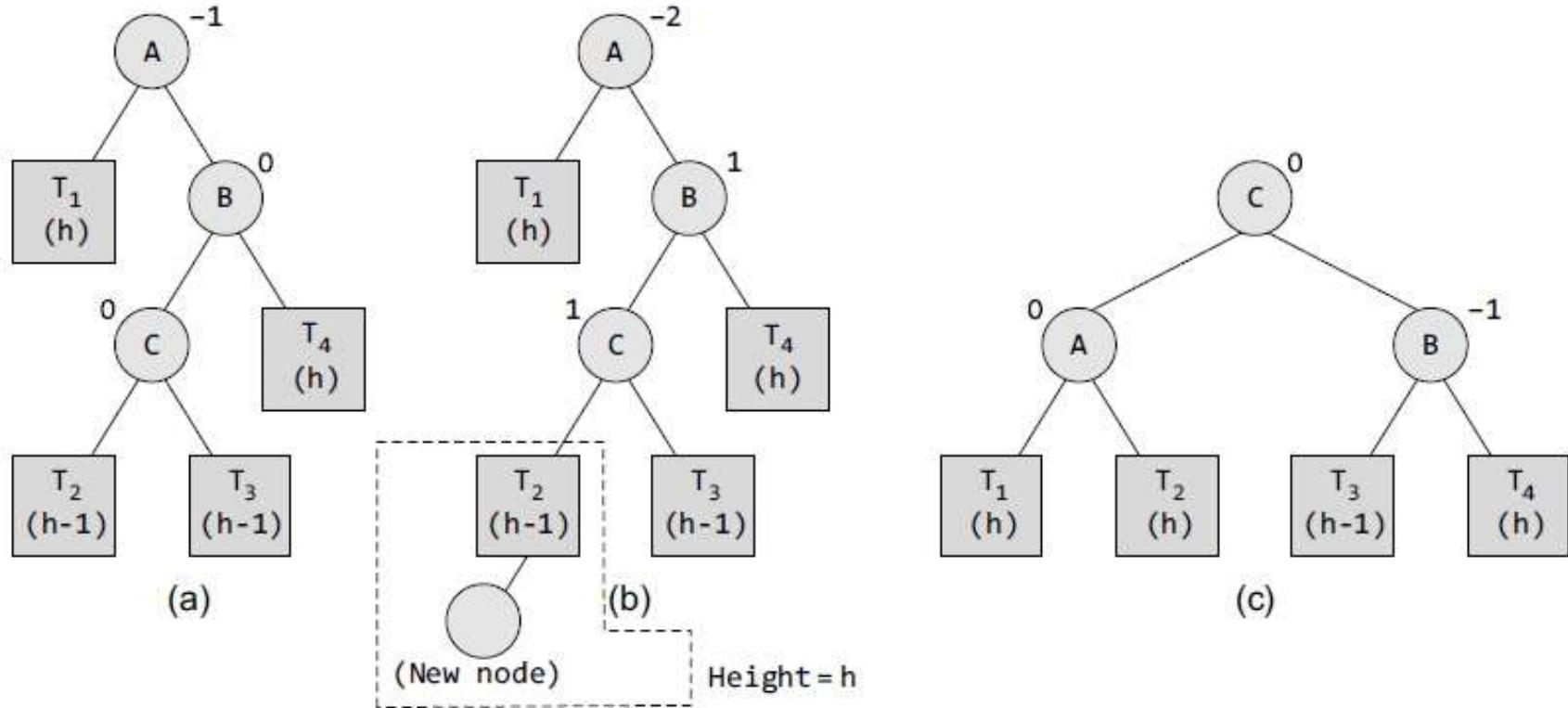
# AVL TREE ROTATIONS - RR



# AVL TREE ROTATIONS - LR



# AVL TREE ROTATIONS - RL

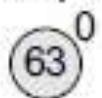


# EXAMPLE - INSERTION

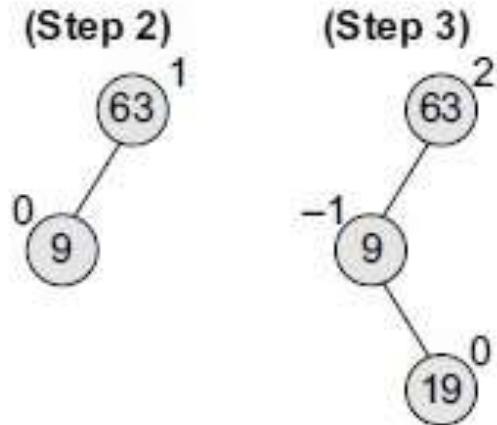
Construct an AVL tree by inserting the following elements in the given order.

**63, 9, 19, 27, 18, 108, 99, 81**

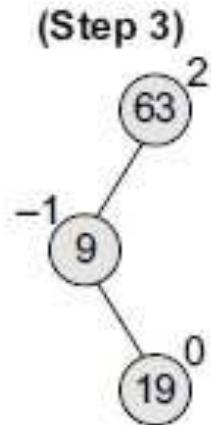
(Step 1)



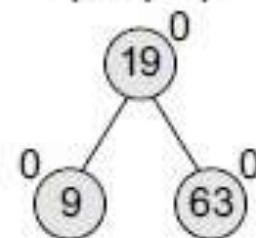
(Step 2)



(Step 3)



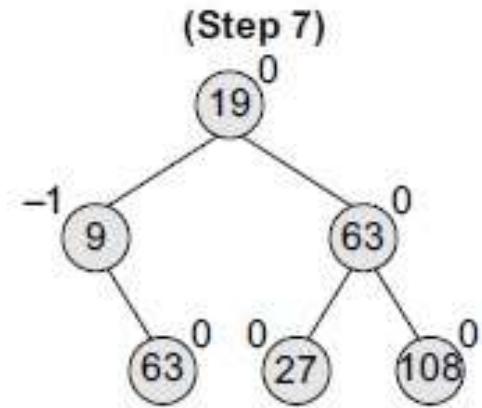
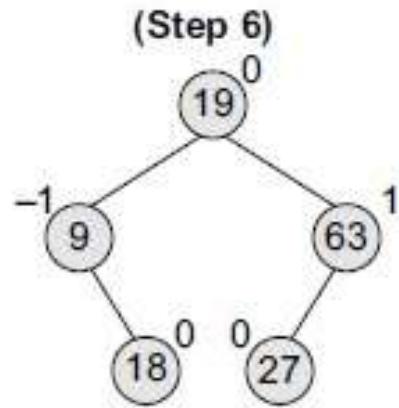
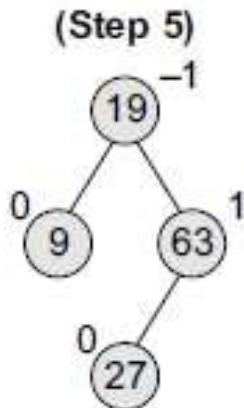
After LR Rotation  
(Step 4)



# EXAMPLE - INSERTION

Construct an AVL tree by inserting the following elements in the given order.

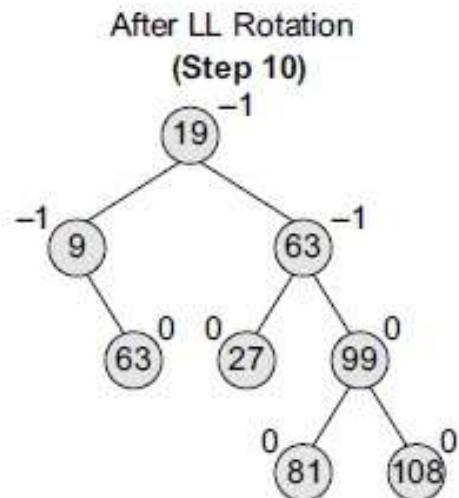
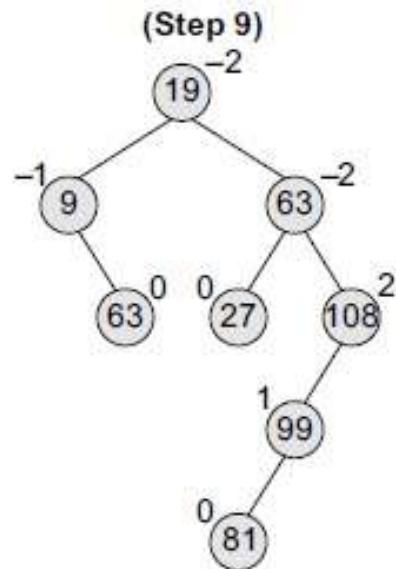
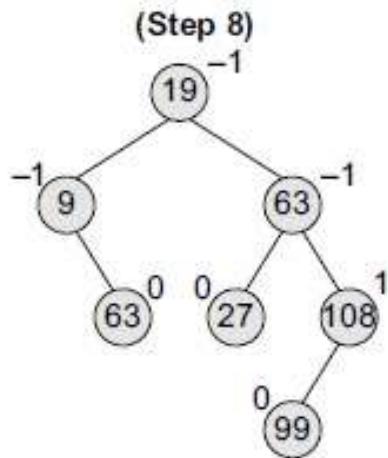
63, 9, 19, **27**, 18, 108, 99, 81



# EXAMPLE - INSERTION

Construct an AVL tree by inserting the following elements in the given order.

**63, 9, 19, 27, 18, 108, 99, 81**



# HOW TO PRESENT IN EXAM

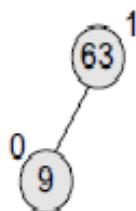
## AVL TREE

INSERT 63, 9, 19, 27, 18, 108, 99, 81

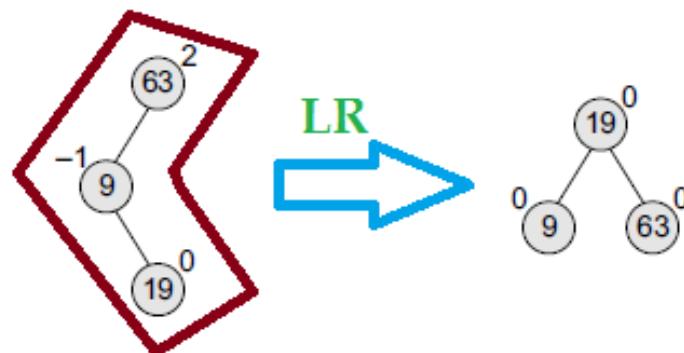
STEP 1 : INSERT 63



STEP 2 : INSERT 9



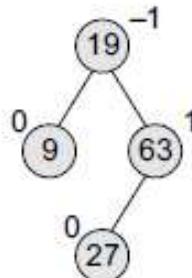
STEP 3 : INSERT 19



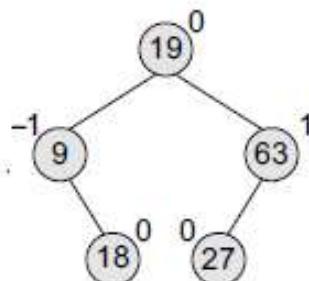
# HOW TO PRESENT IN EXAM

INSERT 63, 9, 19, 27, 18, 108, 99, 81

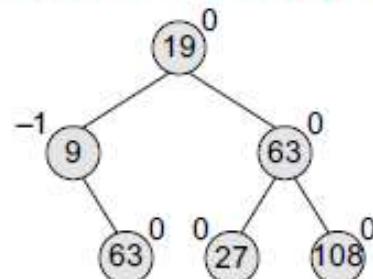
**STEP 4 : INSERT 27**



**STEP 5 : INSERT 18**



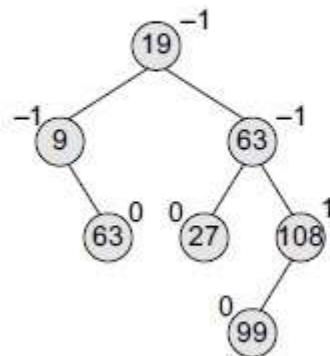
**STEP 6 : INSERT 108**



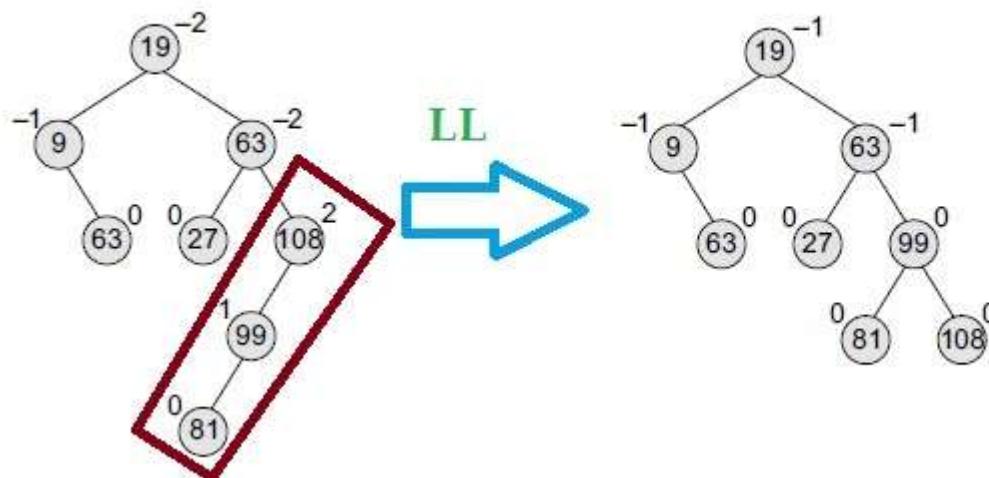
# HOW TO PRESENT IN EXAM

INSERT 63, 9, 19, 27, 18, 108, 99, 81

**STEP 7 : INSERT 99**



**STEP 8 : INSERT 81**



# DELETING A NODE FROM AN AVL TREE

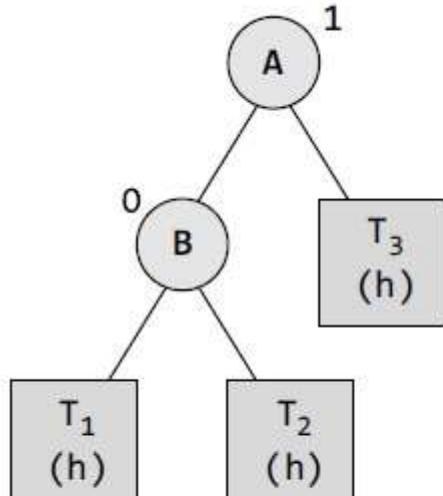
- Deletion of a node in an AVL tree is similar to that of binary search trees.
- But deletion may disturb the AVLness of the tree, so to re-balance the AVL tree we need to perform rotations.
- There are **two classes of rotation** that can be performed on an AVL tree after deleting a given node: **R rotation and L rotation**.
- If the node to be deleted is present in the left sub-tree of the critical node, then **L rotation** is applied else if node is in the right sub-tree, **R rotation** is performed.
- Further there are **three categories of L and R rotations**. The variations of L rotation are: **L-1, L0 and L1 rotation**. Correspondingly for R rotation, there are **R0, R-1 and R1** rotations.
- **L0, L1 and L-1 rotation**

**(Mirror image of R0, R1 and R-1 rotation )**

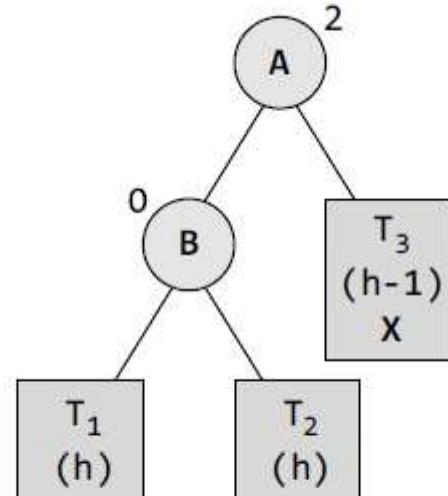
# DELETING A NODE FROM AN AVL TREE

## *R0 Rotation*

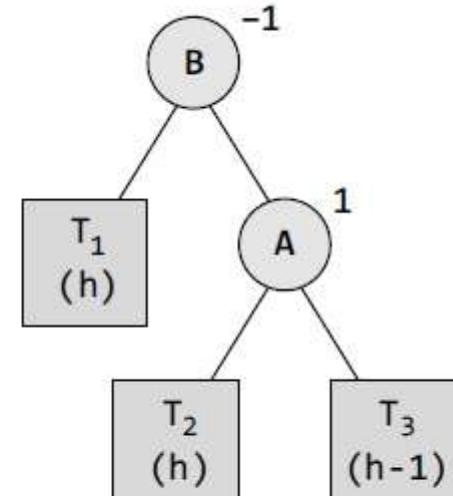
- Let B be the root of the left sub-tree of A (critical node).
- R0 rotation is applied if the balance factor of B is 0.



(a)



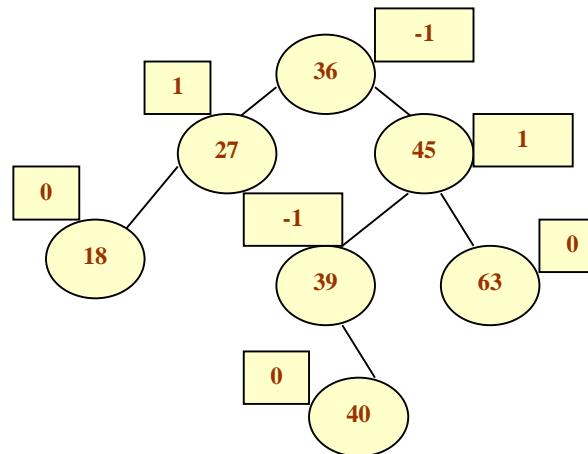
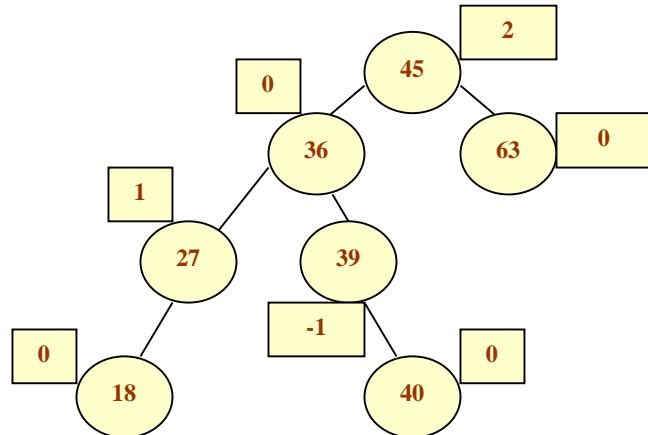
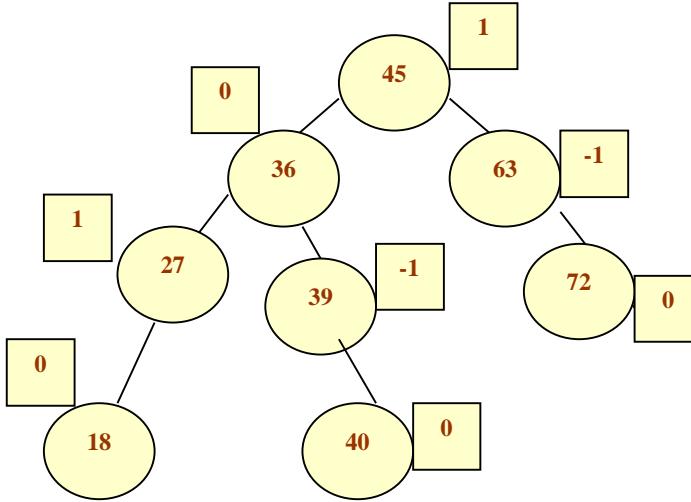
(b)



(c)

# DELETING A NODE FROM AN AVL TREE

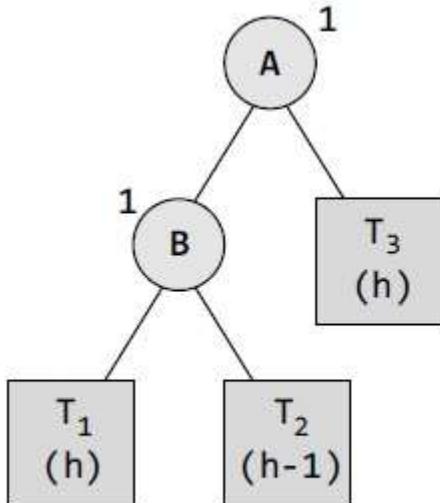
Consider the AVL tree given below and delete 72 from it.



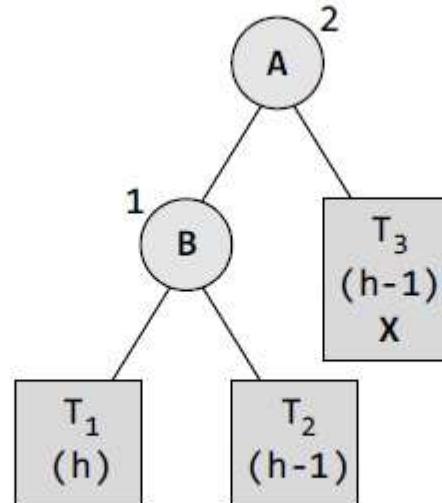
# DELETING A NODE FROM AN AVL TREE

## R1 Rotation

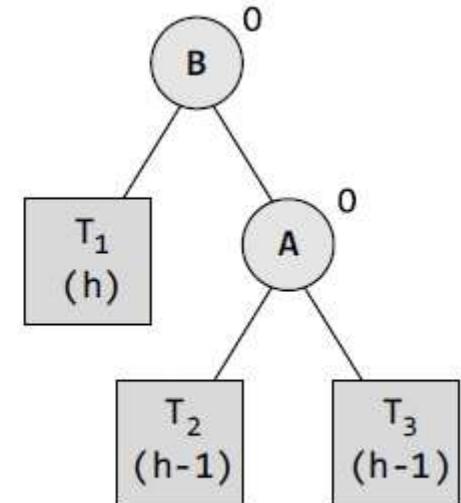
- Let B be the root of the left sub-tree of the critical node.
- R1 rotation is applied if the balance factor of B is 1.



(a)



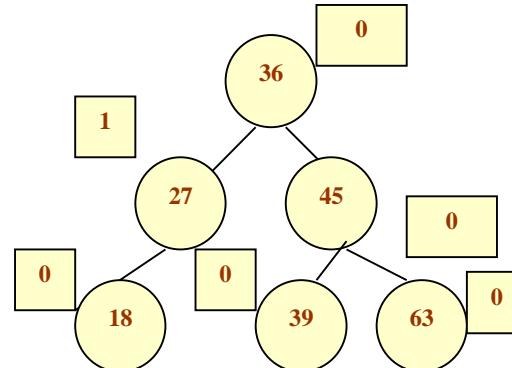
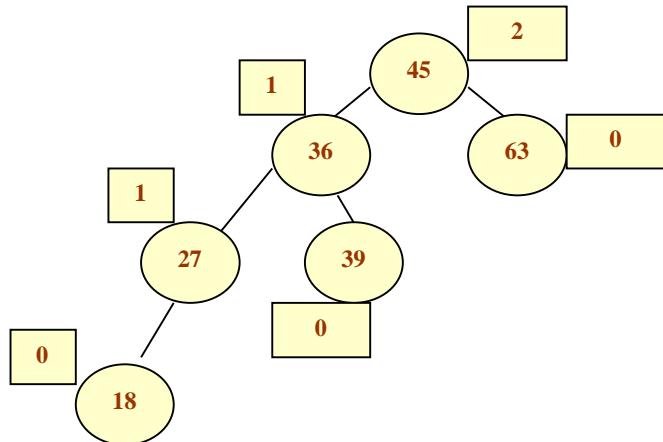
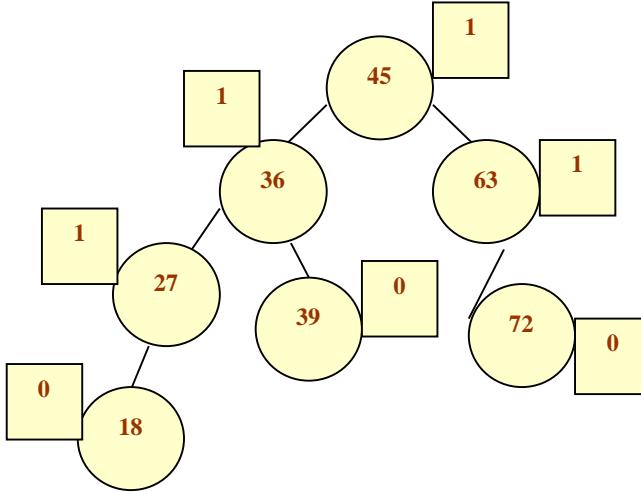
(b)



(c)

# DELETING A NODE FROM AN AVL TREE

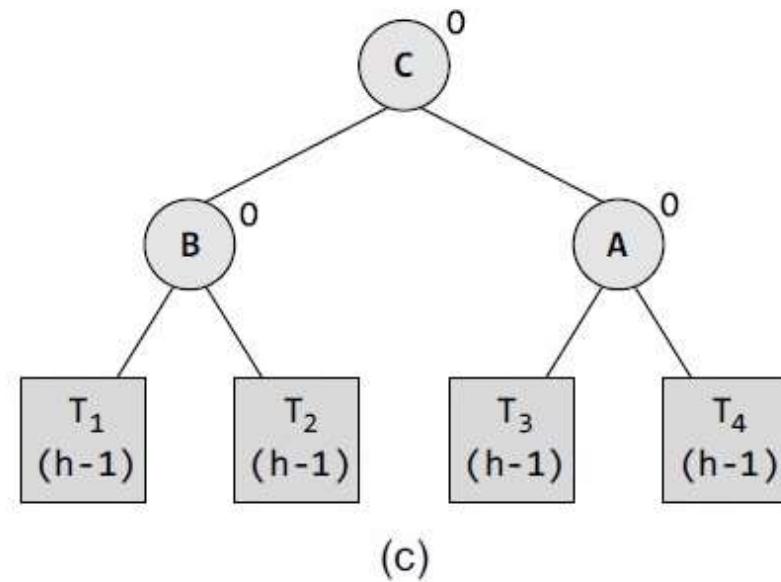
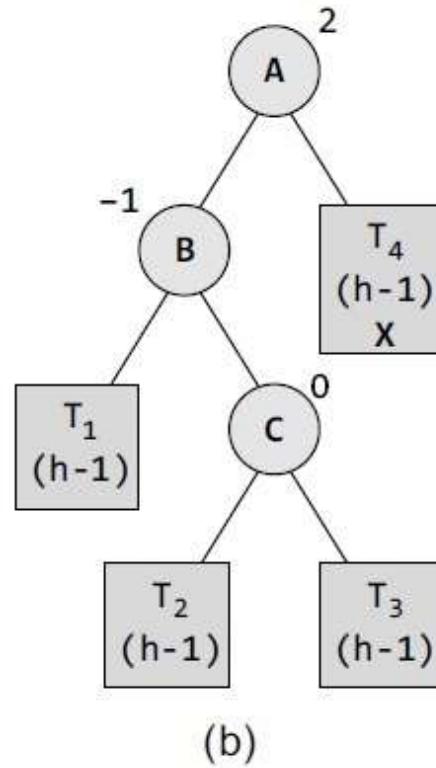
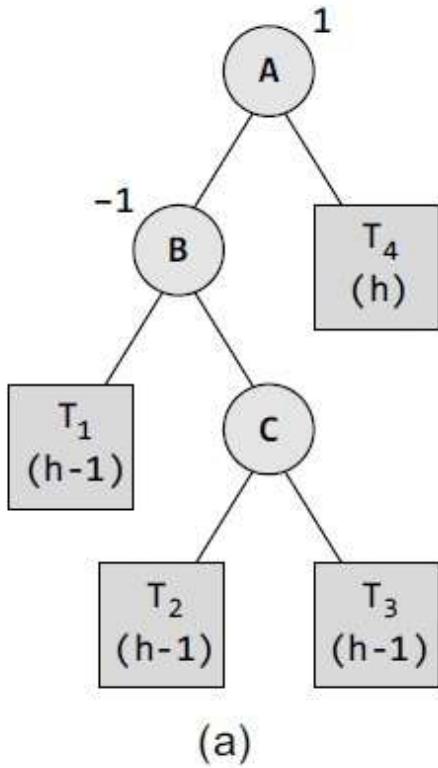
- Consider the AVL tree given below and delete 72 from it.



# DELETING A NODE FROM AN AVL TREE

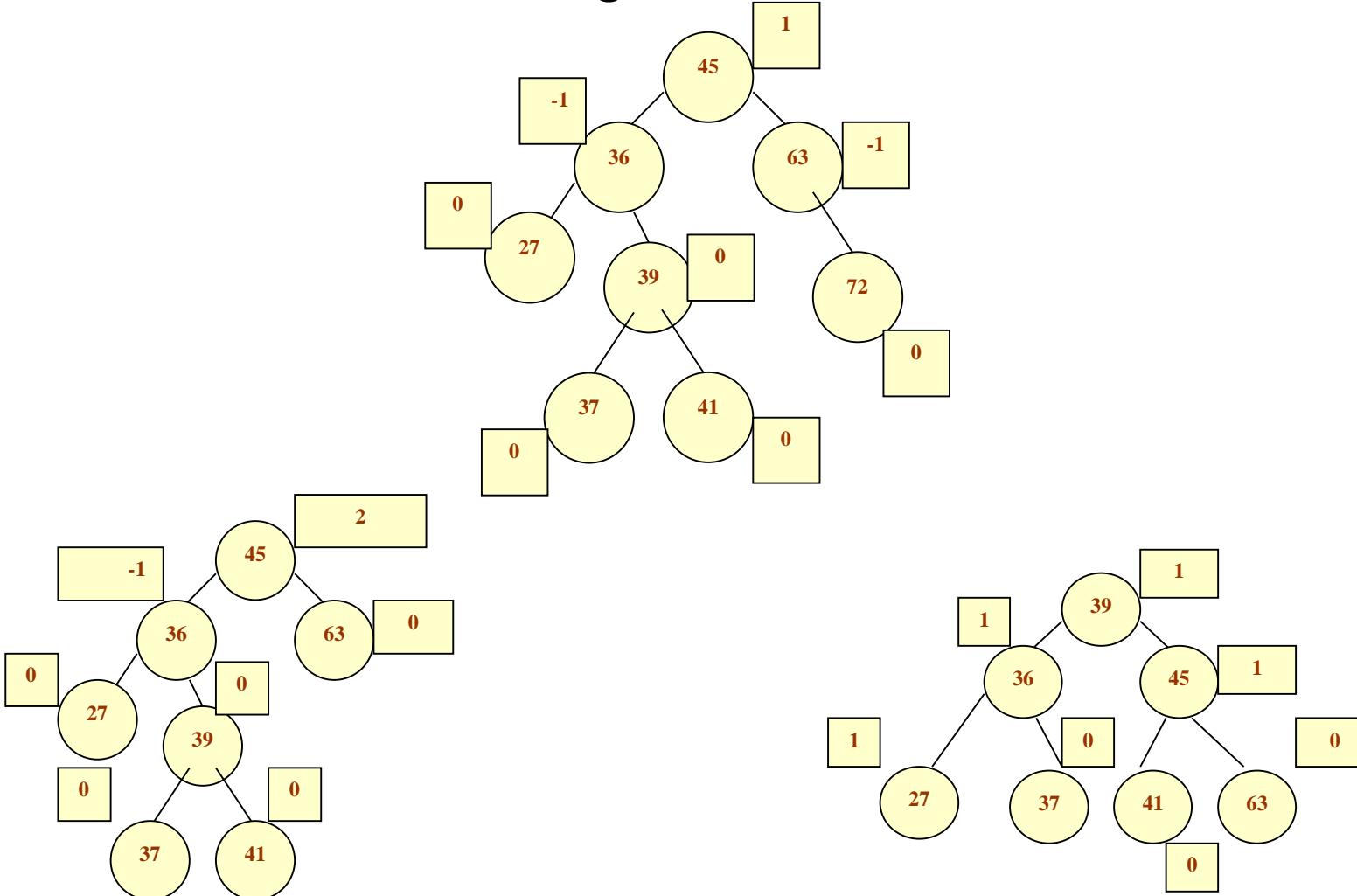
## R-1Rotation

- Let B be the root of the left sub-tree of the critical node.
- R-1 rotation is applied if the balance factor of B is -1.



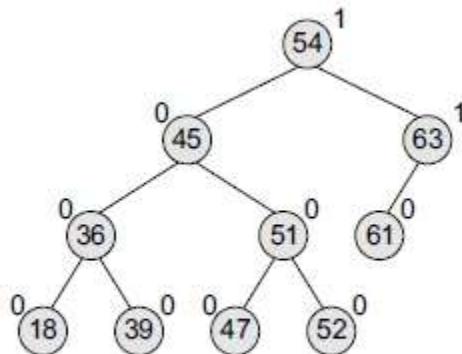
# DELETING A NODE FROM AN AVL TREE

- Consider the AVL tree given below and delete 72 from it.

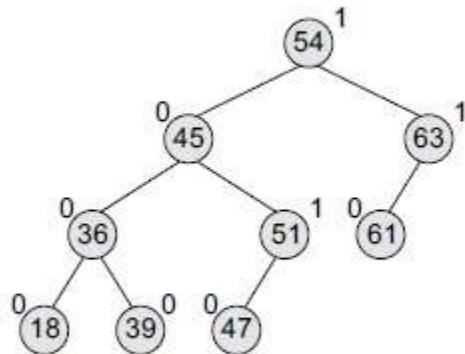


# HOW TO PRESENT IN EXAM

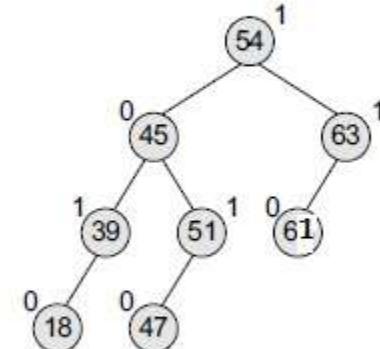
DELETE 52, 36, 61



STEP 1 : DELETE 52



STEP 2 : DELETE 36



STEP 3 : DELETE 61

