Trees

sushmakadge@somaiya.edu

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

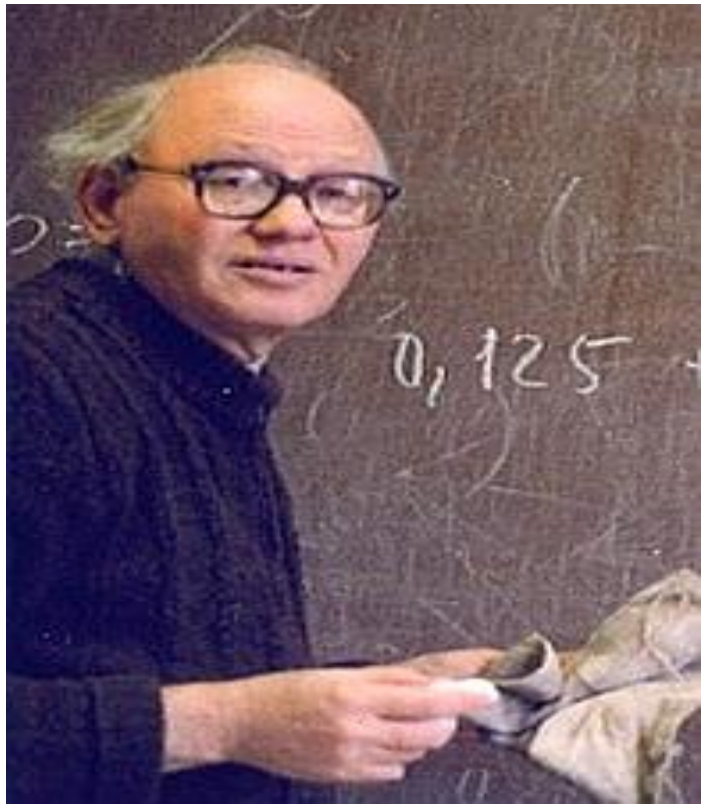
Outline

- Tree – concept
- General tree
- Types of trees
- Binary tree: representation, operation
- Binary tree traversal
- Binary search tree
- BST- The data structure and implementation
- Threaded binary trees.
- Search Trees –
  - AVL tree, Multiway Search Tree, B Tree, B+ Tree, and Trie,
- Applications/Case study of trees.
- Summary
- Queries?

Somaiya TRUST

AVL Tree

- The tree is named AVL in honour of its inventors.
- AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962.



AVL Tree

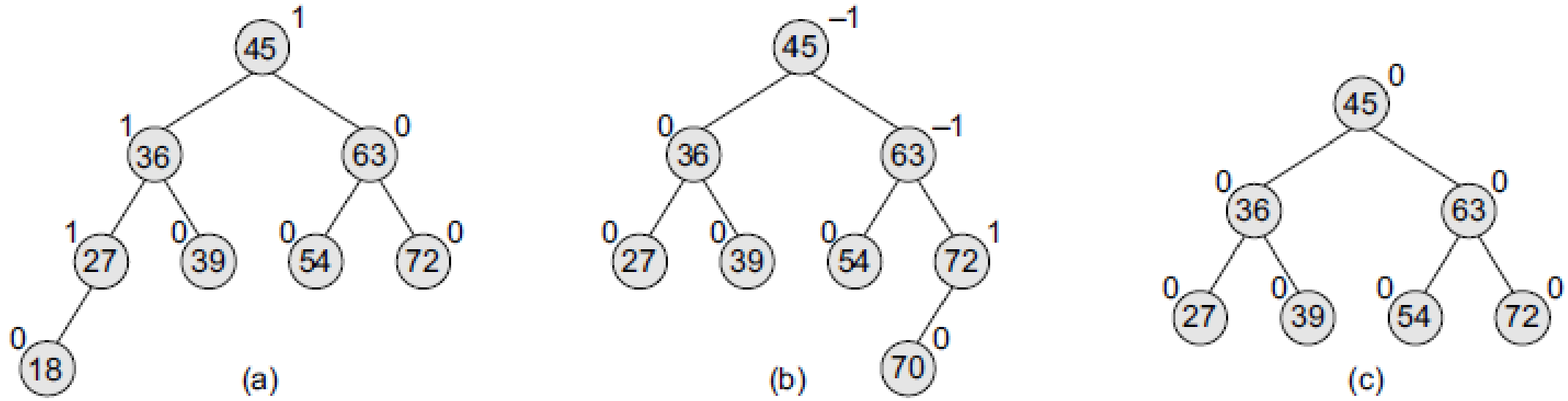
- In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the AVL tree is also known as a height-balanced tree.
- The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the BalanceFactor.
- 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.
- A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced.
- *Balance factor = Height (left sub-tree) – Height (right sub-tree)*

AVL Tree

- A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.
- 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 therefore called as a *left-heavy tree*.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-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 therefore called as a *right-heavy tree*.

AVL Tree

(a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree



AVL Tree

- The trees given in Fig. are typical candidates of AVL trees because the balancing factor of every node is either 1, 0, or –1.
- However, insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, rebalancing of the tree may have to be done.
- The tree is rebalanced by performing rotation at the critical node.
- There are four types of rotations:
- LL rotation, RR rotation, LR rotation, and RL rotation.
- The type of rotation that has to be done will vary depending on the particular situation.

Operations on AVL Trees

- ***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.
- Since the operation does not modify the structure of the tree, no special provisions are required.

Inserting a New Node in an AVL Tree

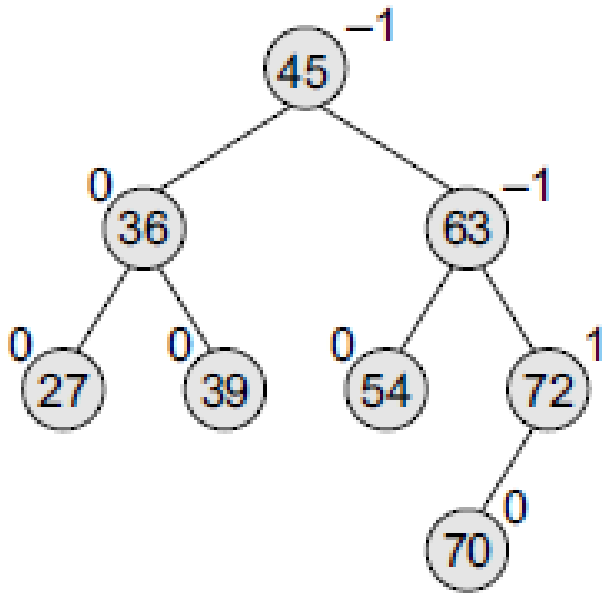
- Insertion in an AVL tree is also done in the same way as it is done in a binary search tree.
- In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation.
- Rotation is done to restore the balance of the tree.
- 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 required.
- During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node.

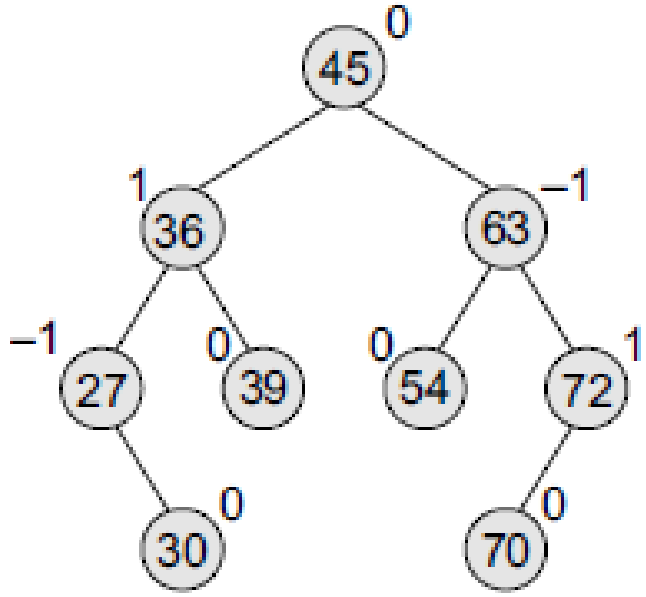
Inserting a New Node in an AVL Tree

- The possible changes which may take place in any node on the path are as follows:
- Initially, the node was either left- or right-heavy and after insertion, it becomes balanced.
- Initially, the node was balanced and after insertion, it becomes either left- or right-heavy.
- Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a *critical node*.

Inserting a New Node in an AVL Tree

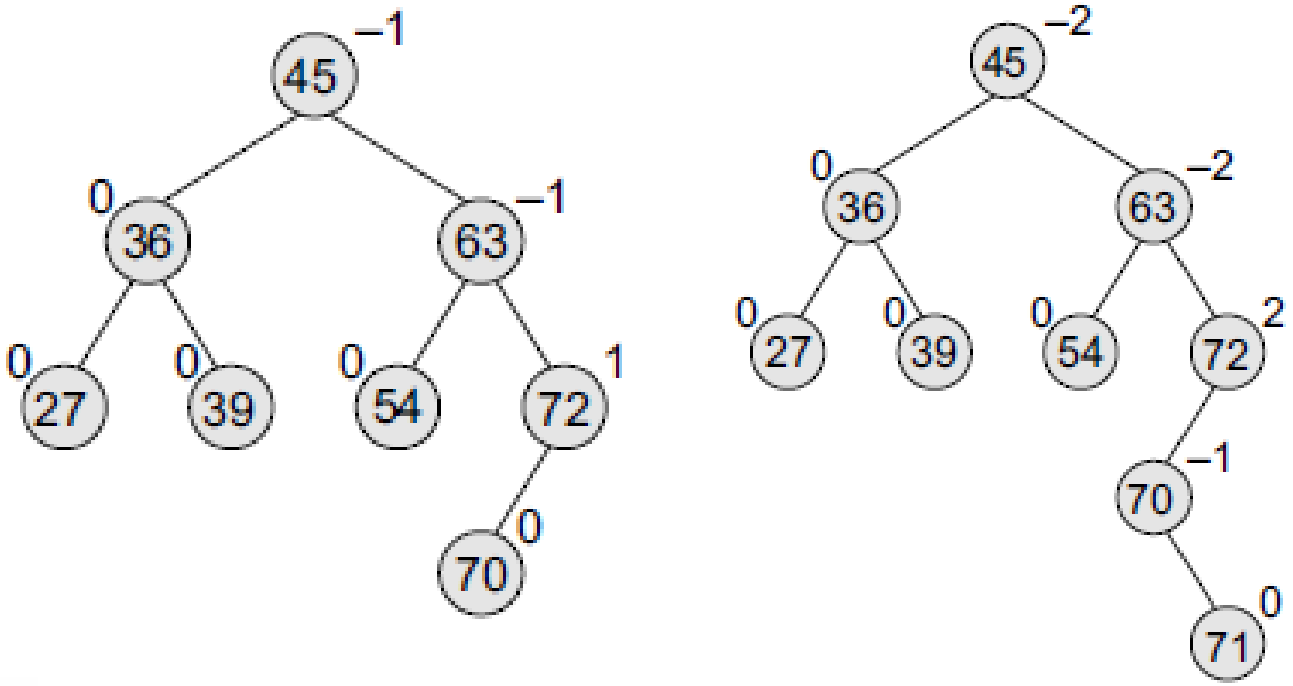
- Consider the AVL tree given in Fig. If we insert a new node with the value 30, then the new tree will still be balanced and no rotations will be required in this case. The tree after inserting node 30.





Inserting a New Node in an AVL Tree

- Let us take another example to see how insertion can disturb the balance factors of the nodes and how rotations are done to restore the AVL property of a tree. After inserting a new node with the value 71, the new tree will be as shown in Fig. b



Inserting a New Node in an AVL Tree

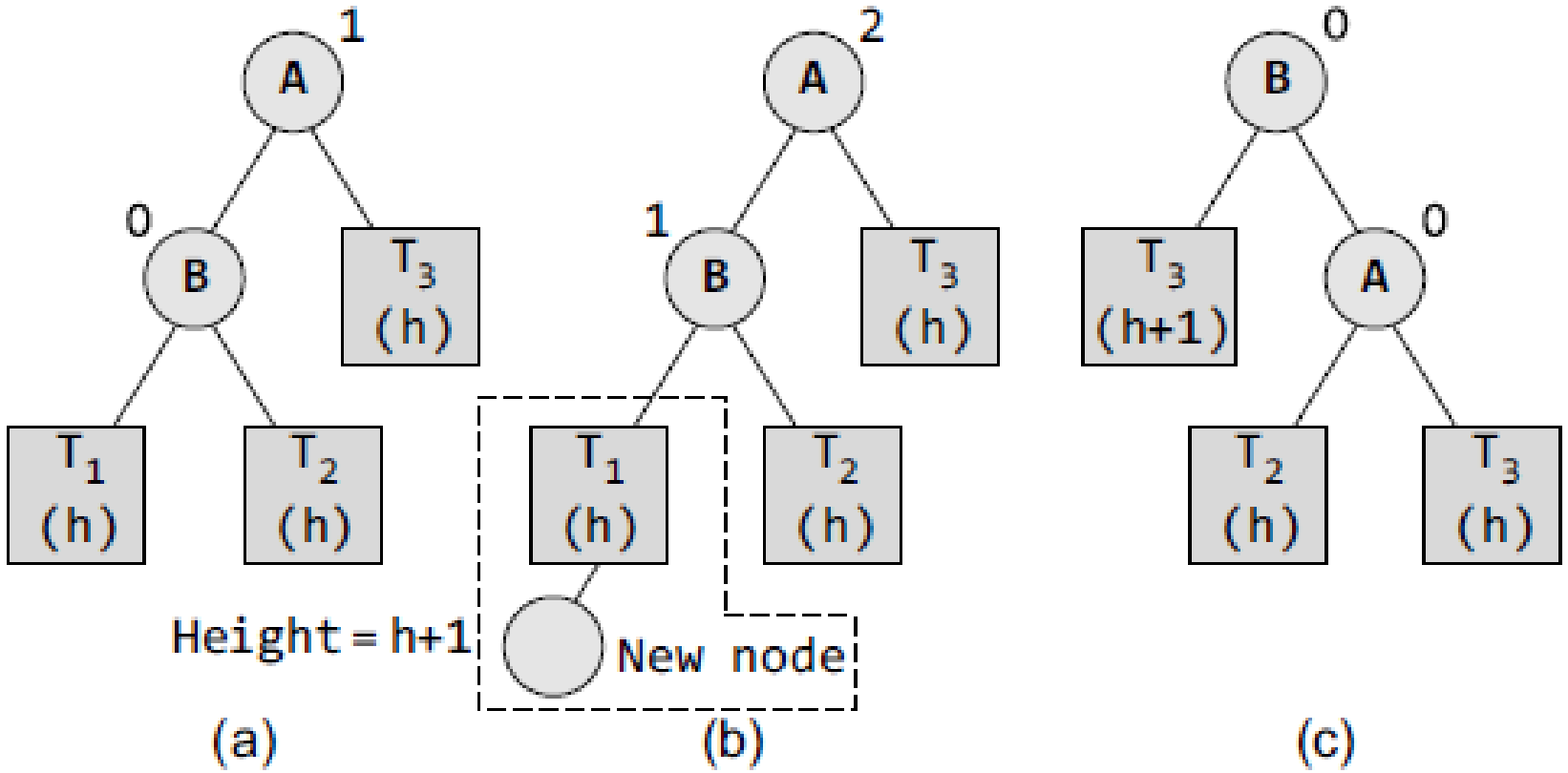
- Note that there are three nodes in the tree that have their balance factors 2, –2, and –2, thereby disturbing the *AVLness* of the tree.
- So, here comes the need to perform rotation.
- To perform rotation, our first task is to find the critical node.
- Critical node is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither –1, 0, nor 1. Critical node is 72.
- The second task in rebalancing the tree is to determine which type of rotation has to be done.
- There are four types of rebalancing rotations and application of these rotations depends on the position of the inserted node with reference to the critical node.

Inserting a New Node in an AVL Tree

- The **four categories of rotations** are:
- *LL rotation :* The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
- *RR rotation :* The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
- *LR rotation :* The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
- *RL rotation :* The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

Inserting a New Node in an AVL Tree

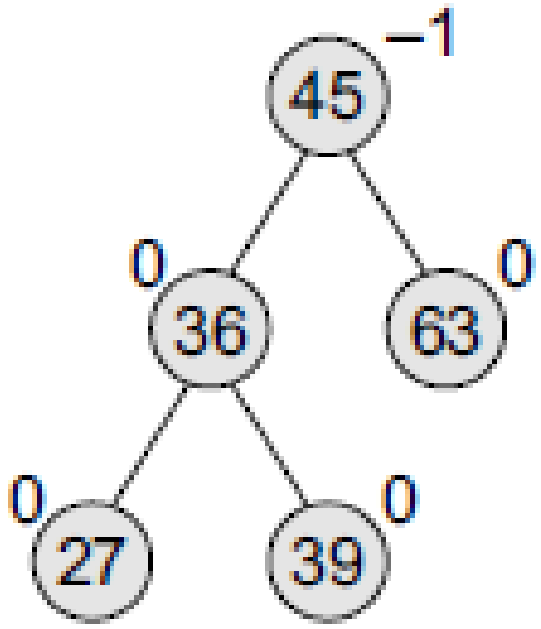
- ***LL Rotation:*** First, we will see where and how LL rotation is applied.

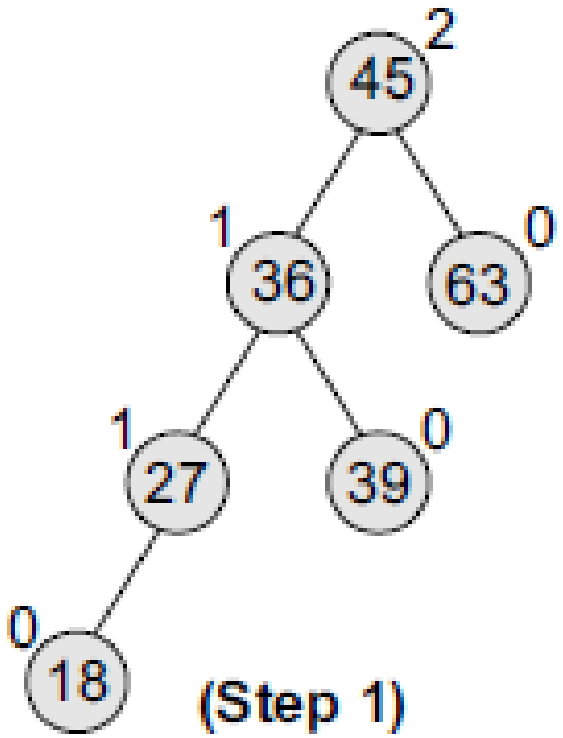


While rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

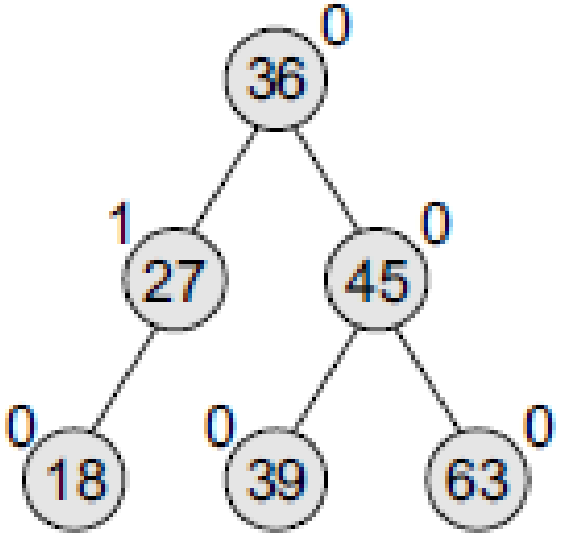
Inserting a New Node in an AVL Tree

- Consider the AVL tree given in Fig. and insert 18 into it.





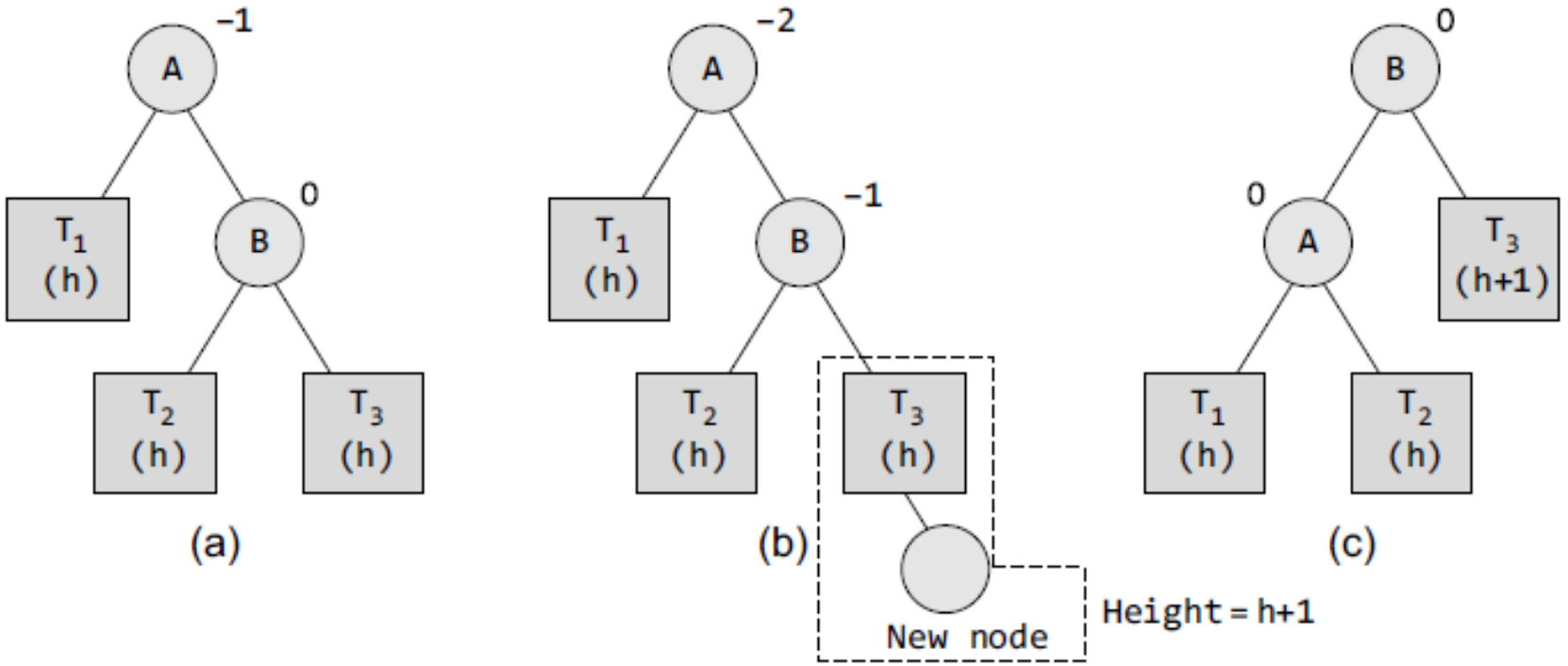
(Step 1)



(Step 2)

Inserting a New Node in an AVL Tree

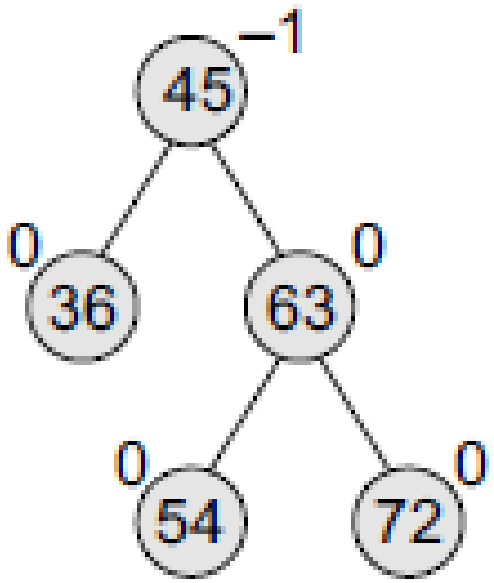
- ***RR Rotation:*** First, we will see where and how RR rotation is applied.

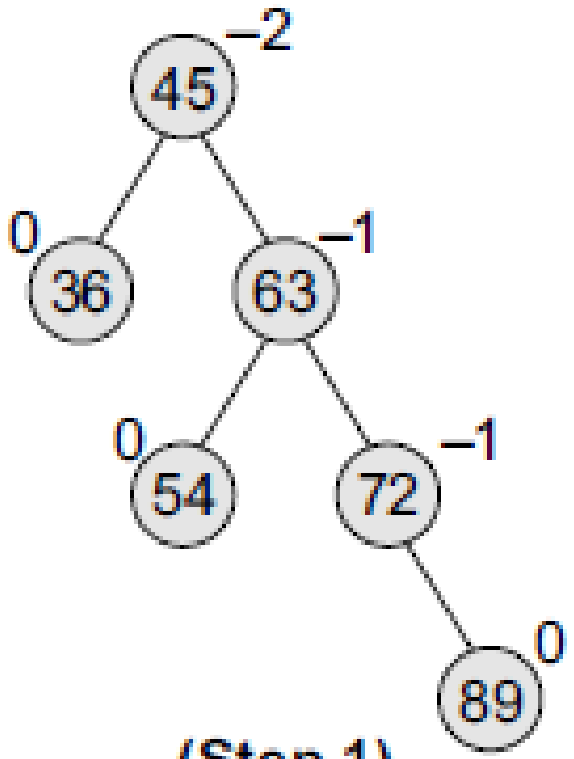


While rotation, node B becomes the root, with A and T3 as its left and right child. T1 and T2 become the left and right sub-trees of A.

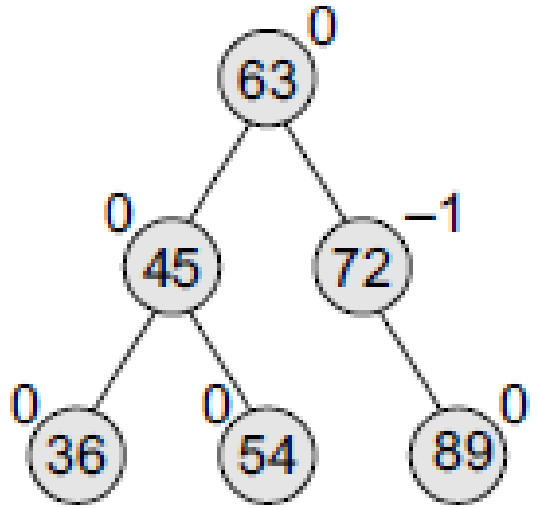
Inserting a New Node in an AVL Tree

- Consider the AVL tree given in Fig. and insert 89 into it.





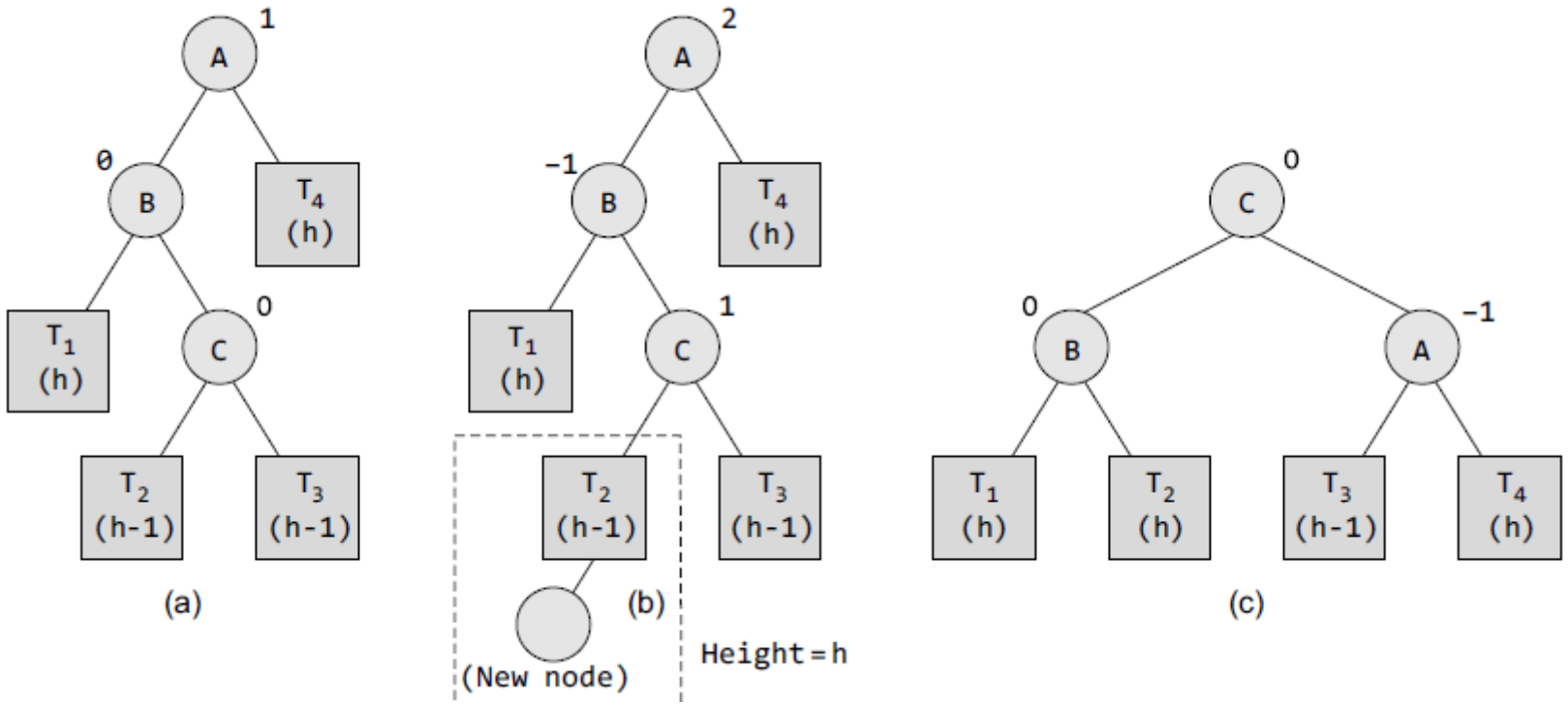
(Step 1)



(Step 2)

Inserting a New Node in an AVL Tree

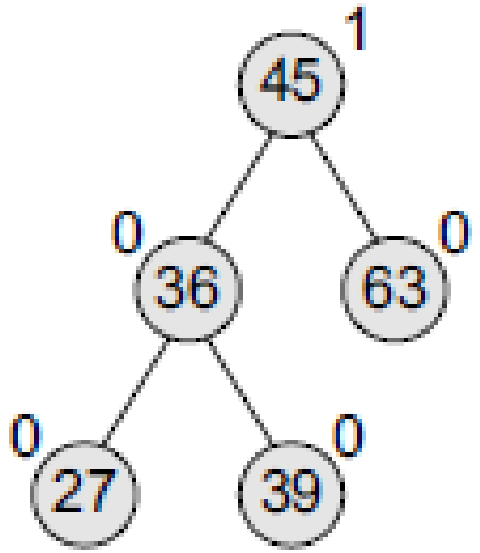
- ***LR Rotations :*** Lets see how LR rotation is done to rebalance the tree.



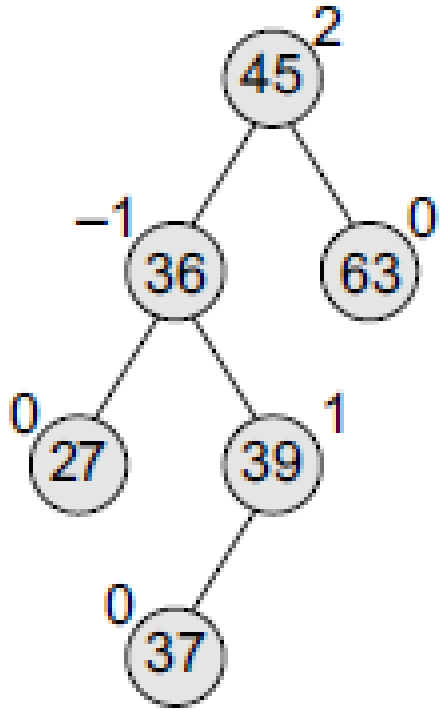
While rotation, node C becomes the root, with B and A as its left and right children. Node B has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node A.

Inserting a New Node in an AVL Tree

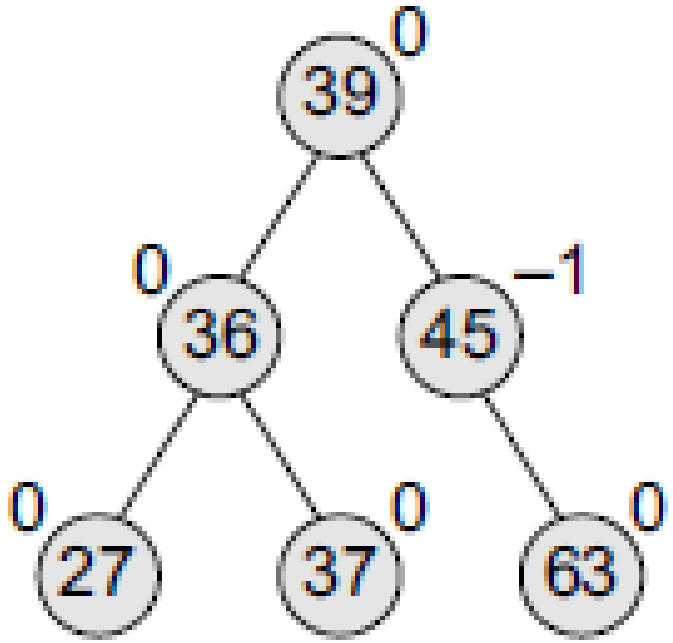
- Consider the AVL tree given in Fig. and insert 37 into it.



(Step 1)

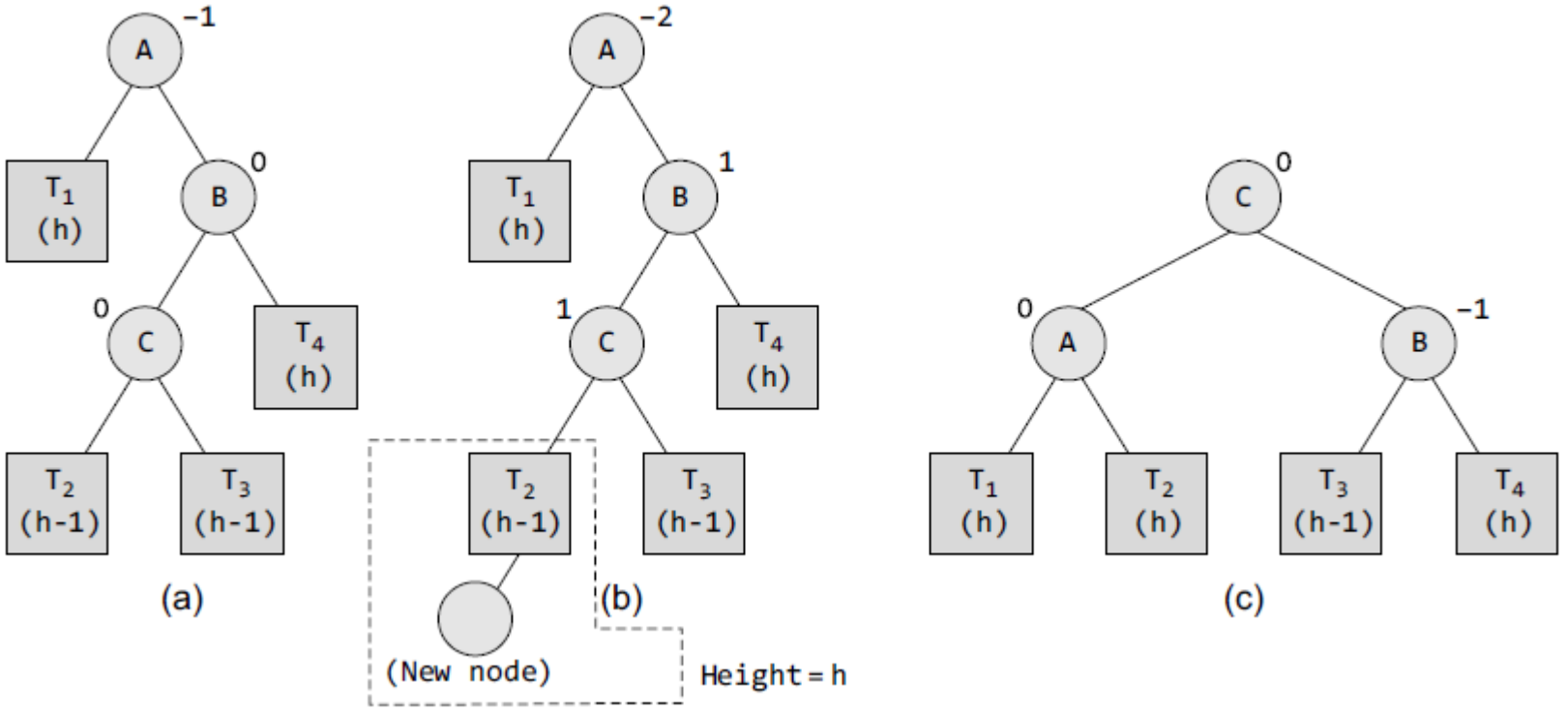


(Step 2)



Inserting a New Node in an AVL Tree

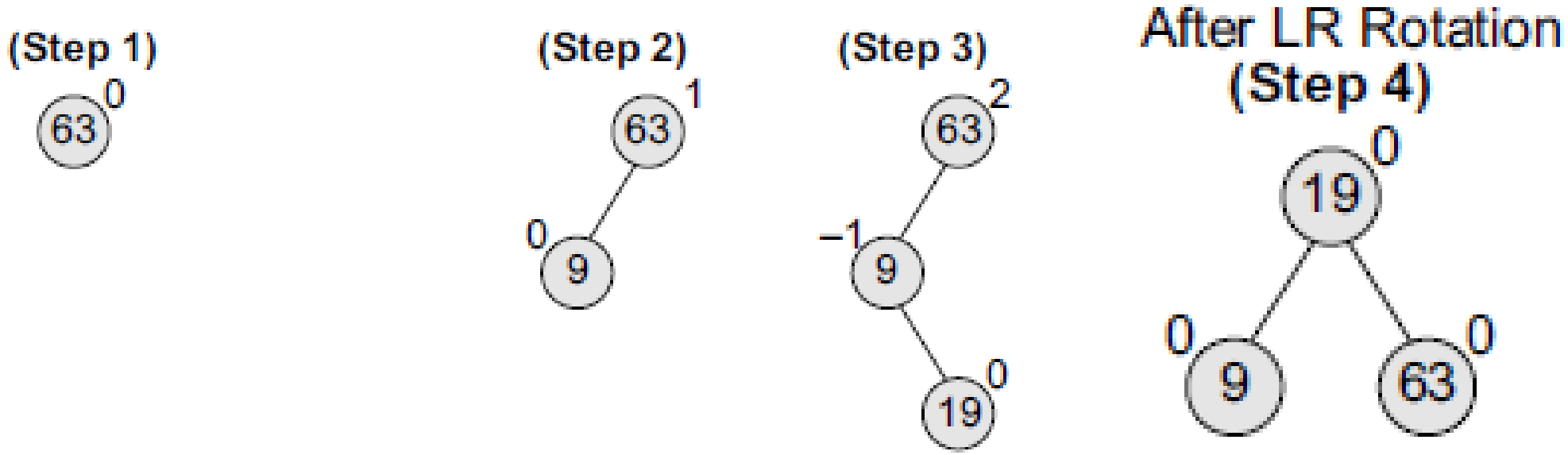
- ***RL Rotations :*** Lets see how RL rotation is done to rebalance the tree.



While rotation, node C becomes the root, with A and B as its left and right children. Node A has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node B.

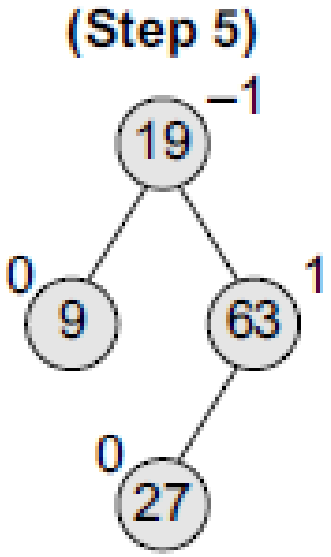
Inserting a New Node in an AVL Tree

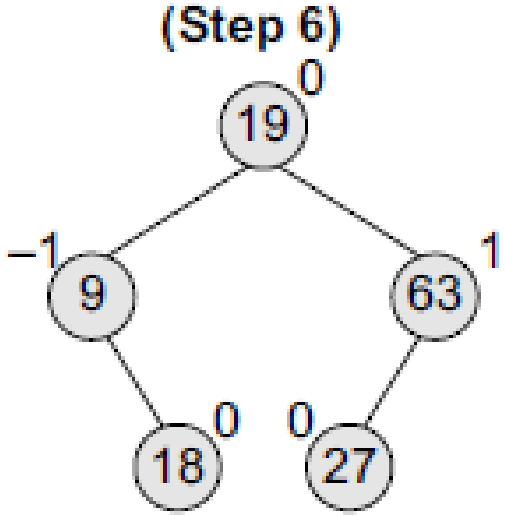
- Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

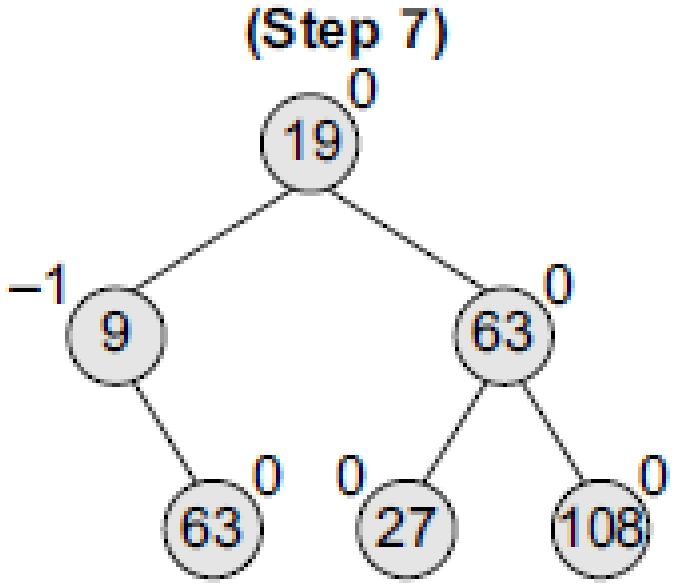


Inserting a New Node in an AVL Tree

- Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

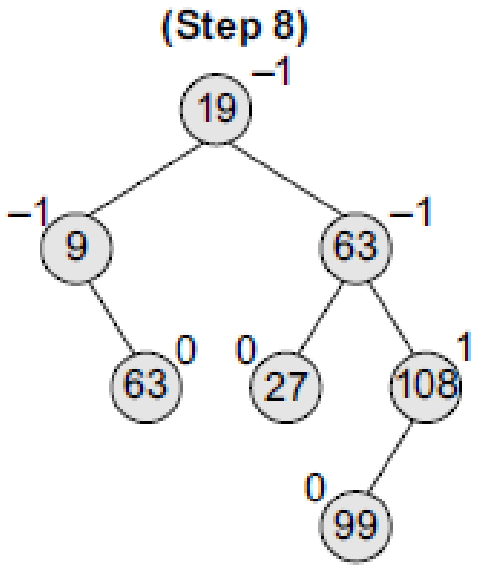


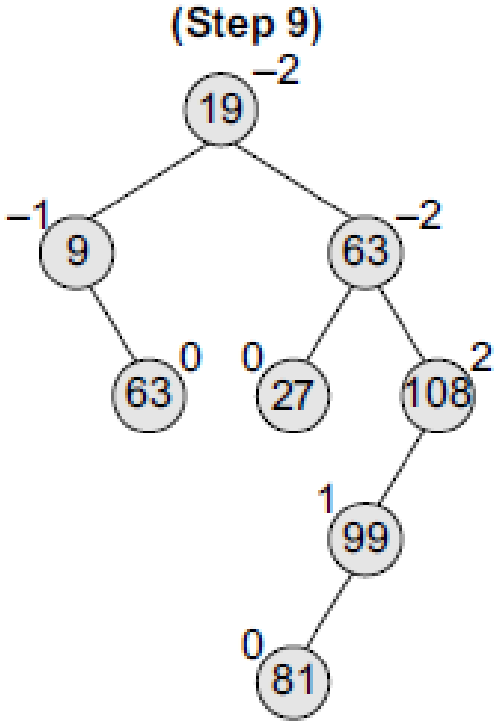


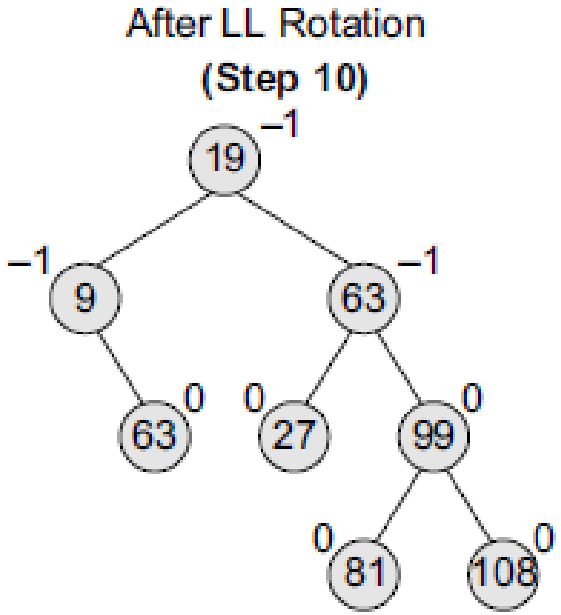


Inserting a New Node in an AVL Tree

- Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.







Deleting a Node from an AVL Tree

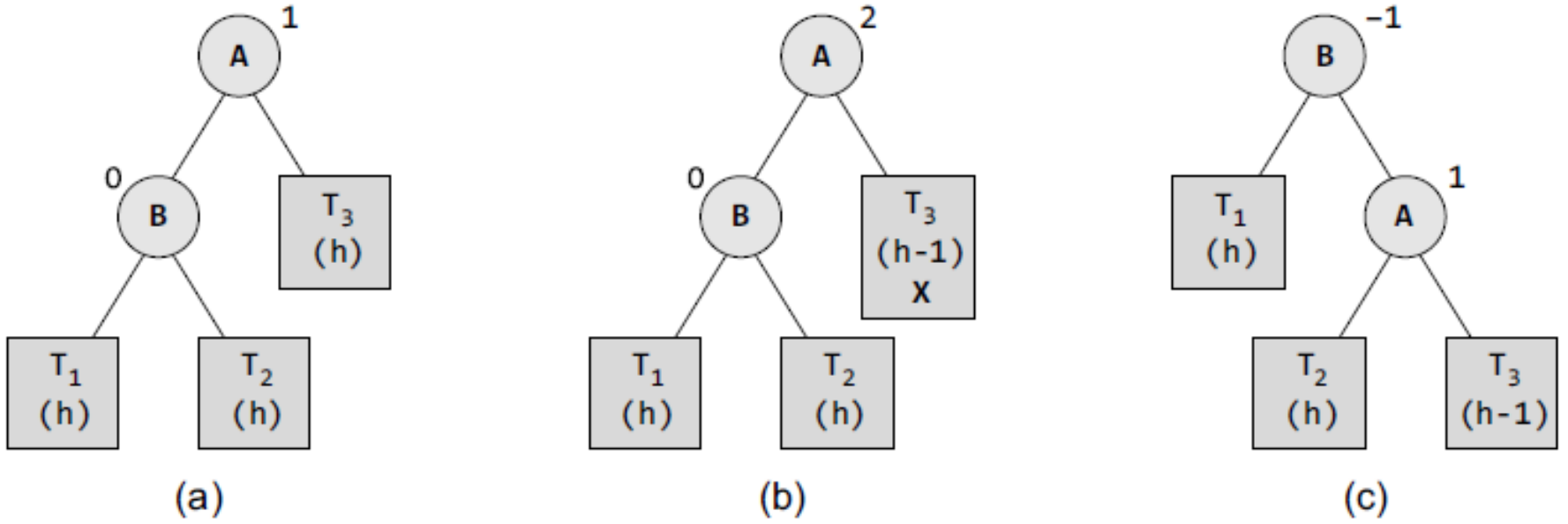
- Deletion of a node in an AVL tree is similar to that of binary search trees. Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree, we need to perform rotations.
- There are two classes of rotations that can be performed on an AVL tree after deleting a given node. These rotations are R rotation and L rotation.
- On deletion of node X from the AVL tree, if node A becomes the critical node (closest ancestor node on the path from X to the root node that does not have its balance factor as 1, 0, or –1), then
- the type of rotation depends on whether X is in the left sub-tree of A or in its right sub-tree.

Deleting a Node from an AVL Tree

- If the node to be deleted is present in the left sub-tree of A, then L rotation is applied, else if X 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.
- We will discuss only R rotation. L rotations are the mirror images of R rotations.

Deleting a Node from an AVL Tree

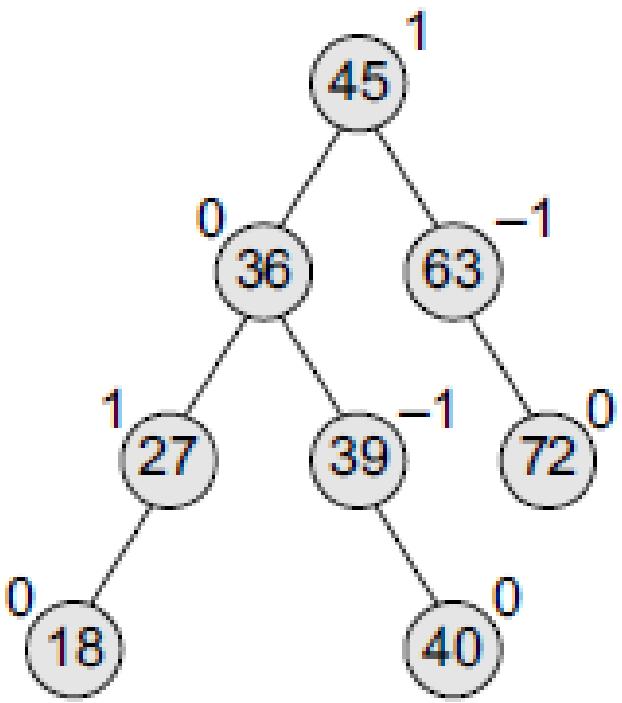
- ***R0 Rotation :*** Let B be the root of the left or right sub-tree of A (critical node). R0 rotation is applied if the balance factor of B is 0.

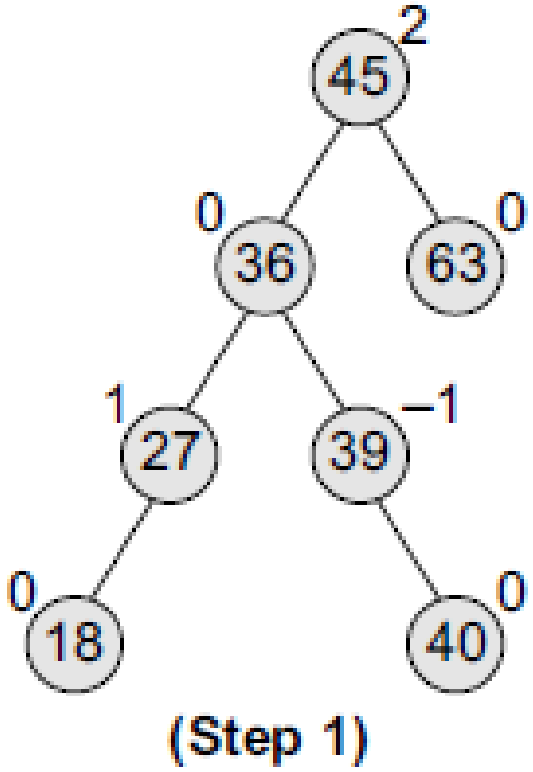


During the process of rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

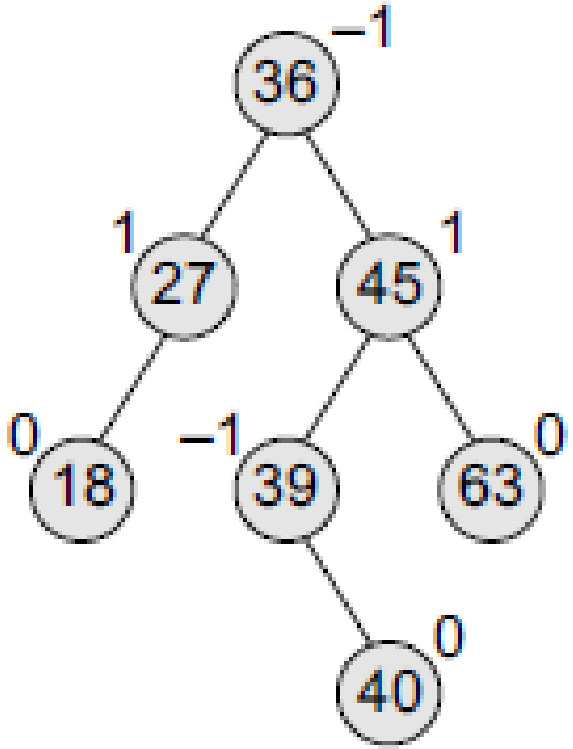
Deleting a Node from an AVL Tree

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





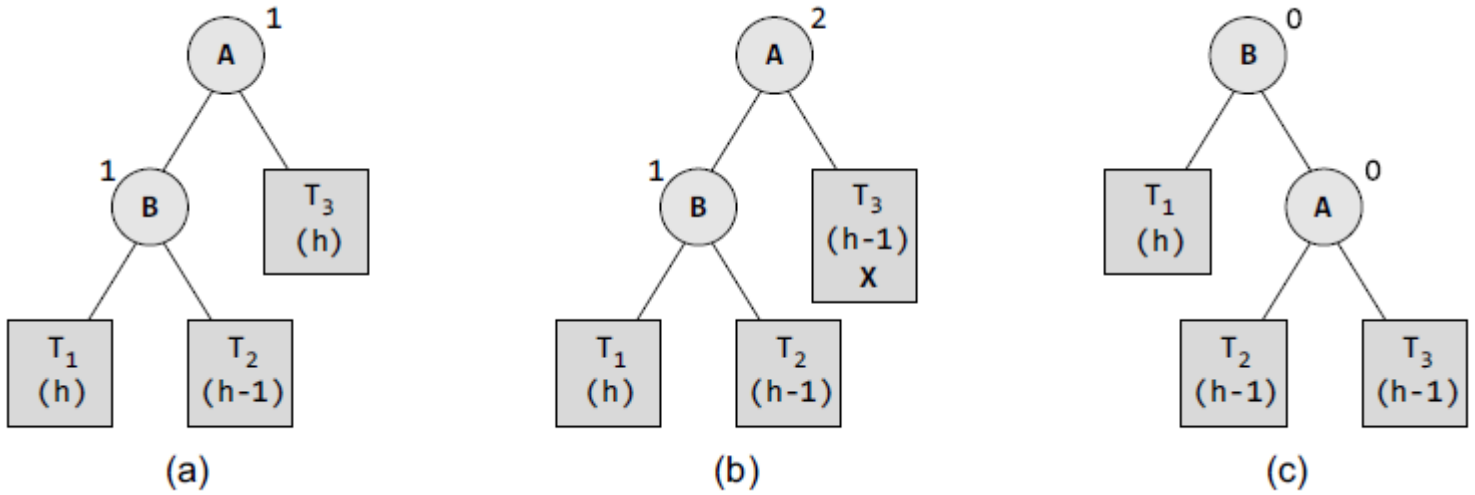
(Step 1)



(Step 2)

Deleting a Node from an AVL Tree

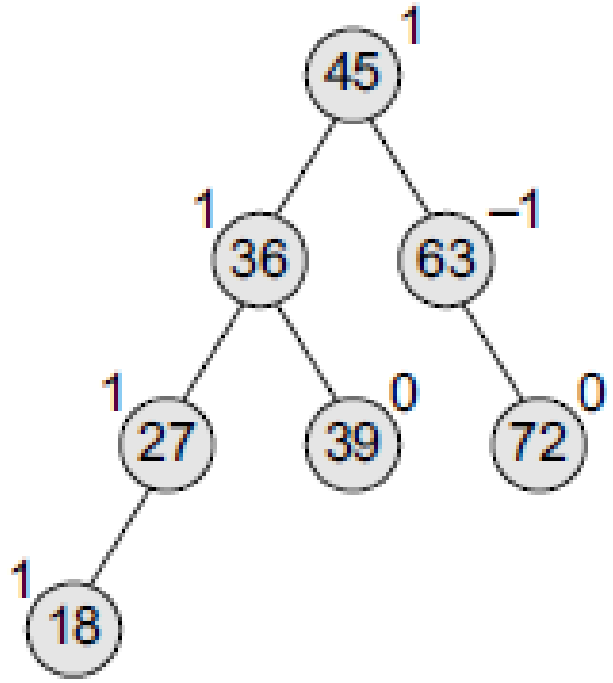
- ***R1 Rotation :*** Let B be the root of the left or right sub-tree of A (critical node). R1 rotation is applied if the balance factor of B is 1.
- Observe that R0 and R1 rotations are similar to LL rotations; the only difference is that R0 and R1 rotations yield different balance factors.

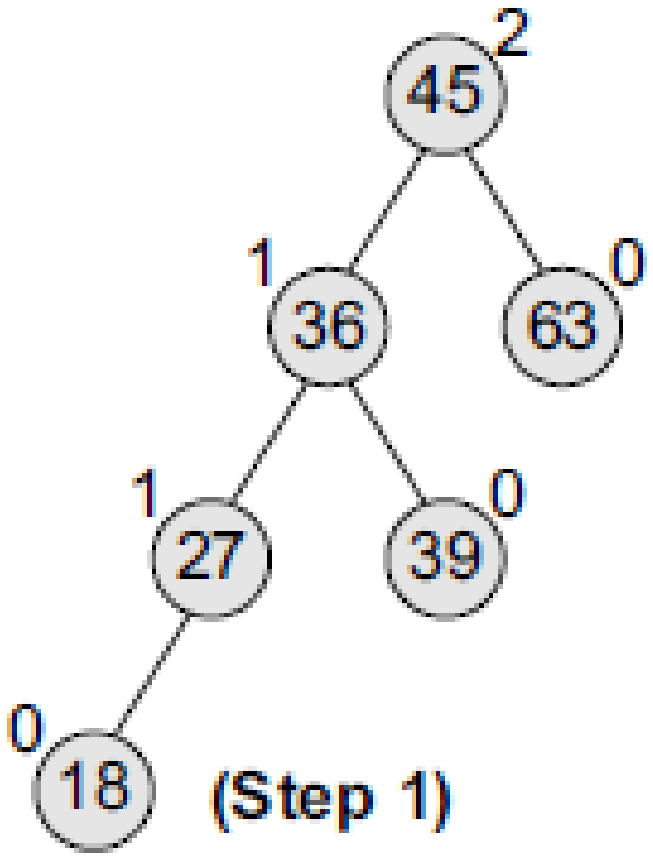


During the process of rotation, node B becomes the root, with T1 and A as its left and right children. T2 and T3 become the left and right sub-trees of A.

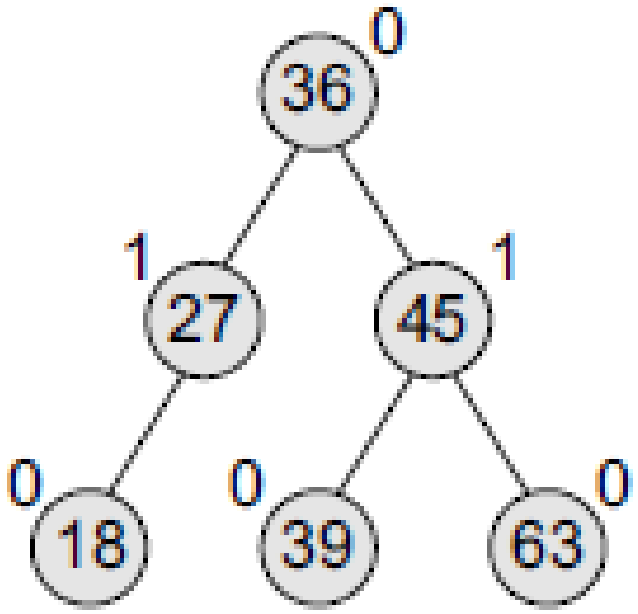
Deleting a Node from an AVL Tree

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





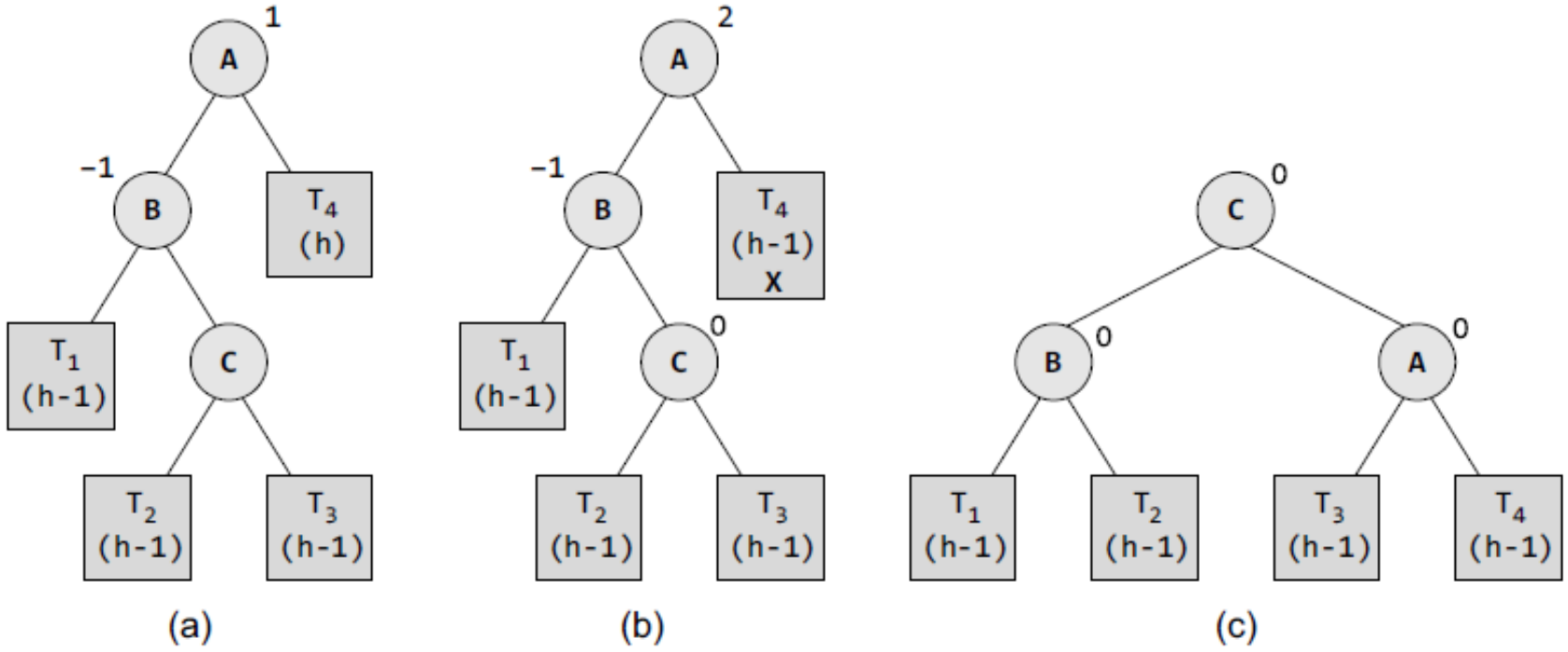
(Step 1)



(Step 2)

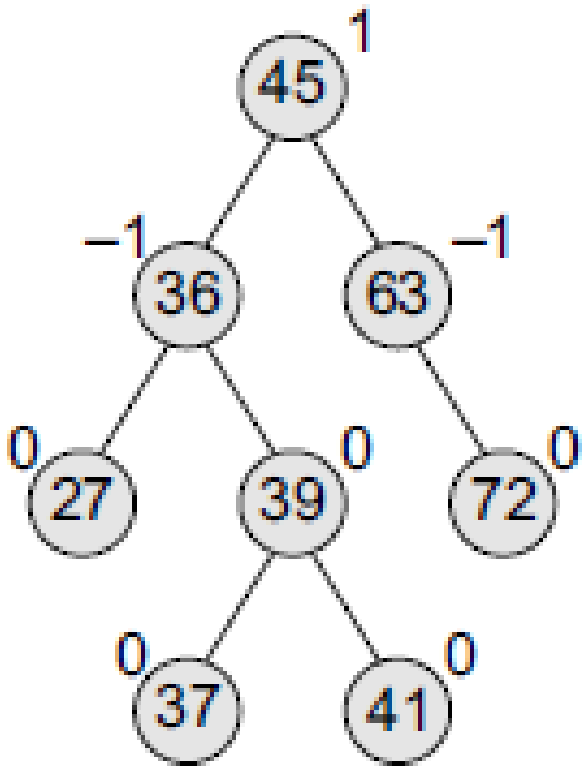
Deleting a Node from an AVL Tree

- ***R–1 Rotation :*** Let B be the root of the left or right sub-tree of A (critical node). R–1 rotation is applied if the balance factor of B is –1. Observe that R–1 rotation is similar to LR rotation. This is illustrated in Fig.

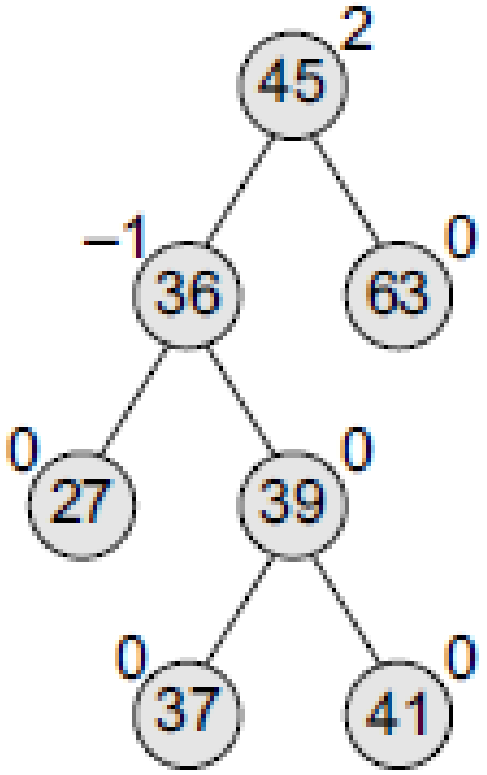


Deleting a Node from an AVL Tree

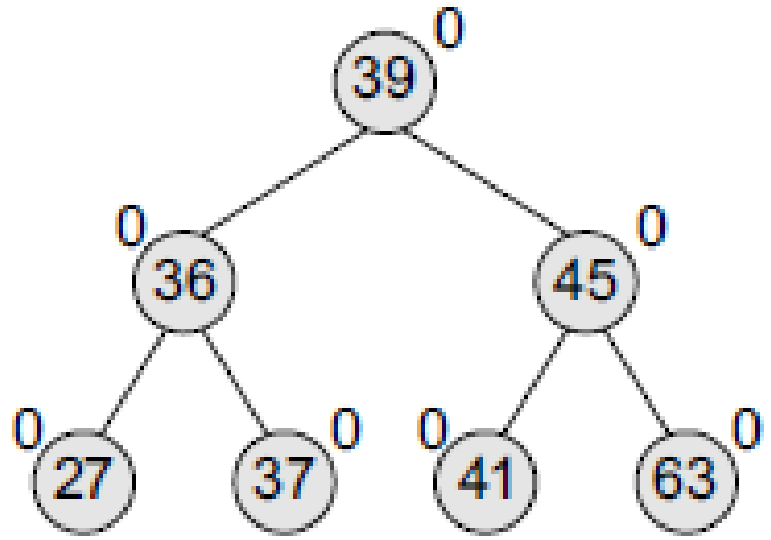
- Consider the AVL tree given in Fig. and delete 72 from it.



(Step 1)

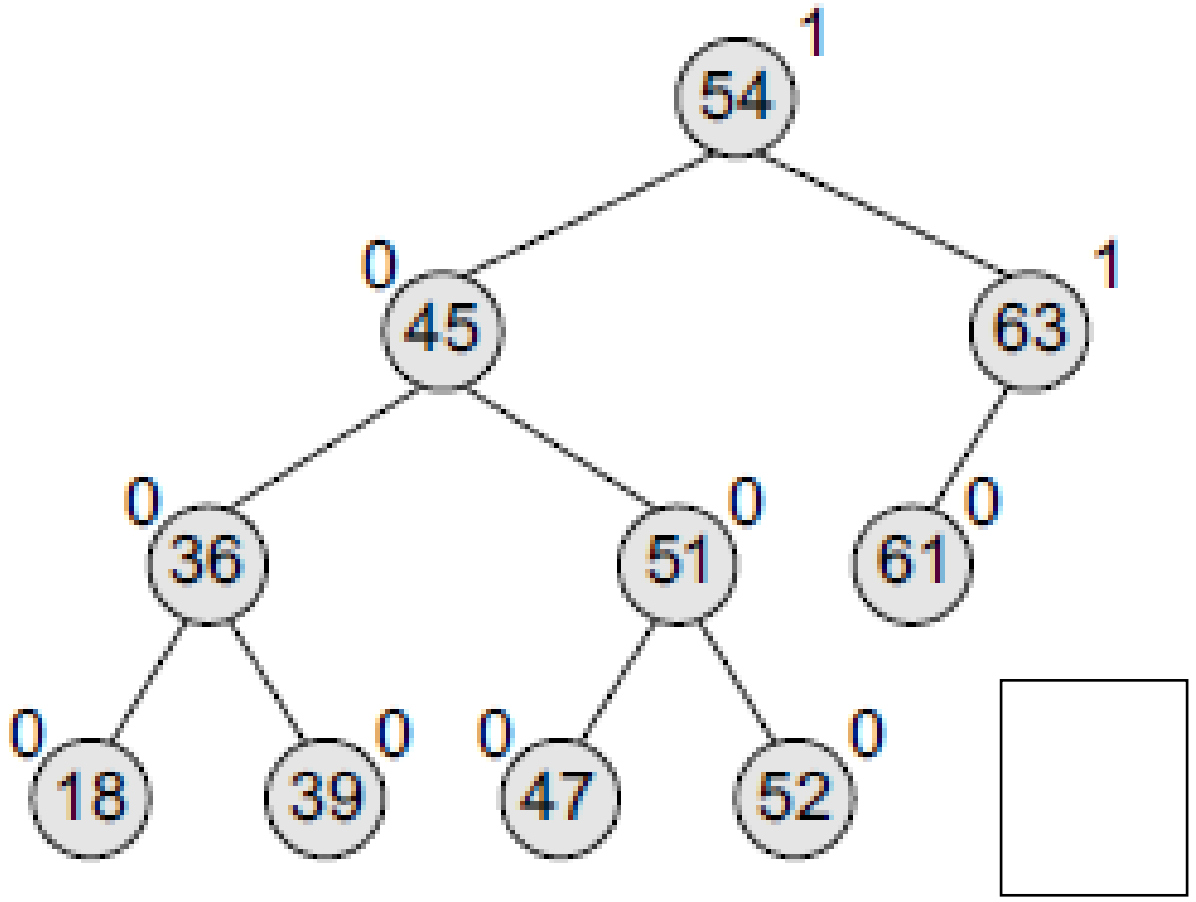


(Step 2)

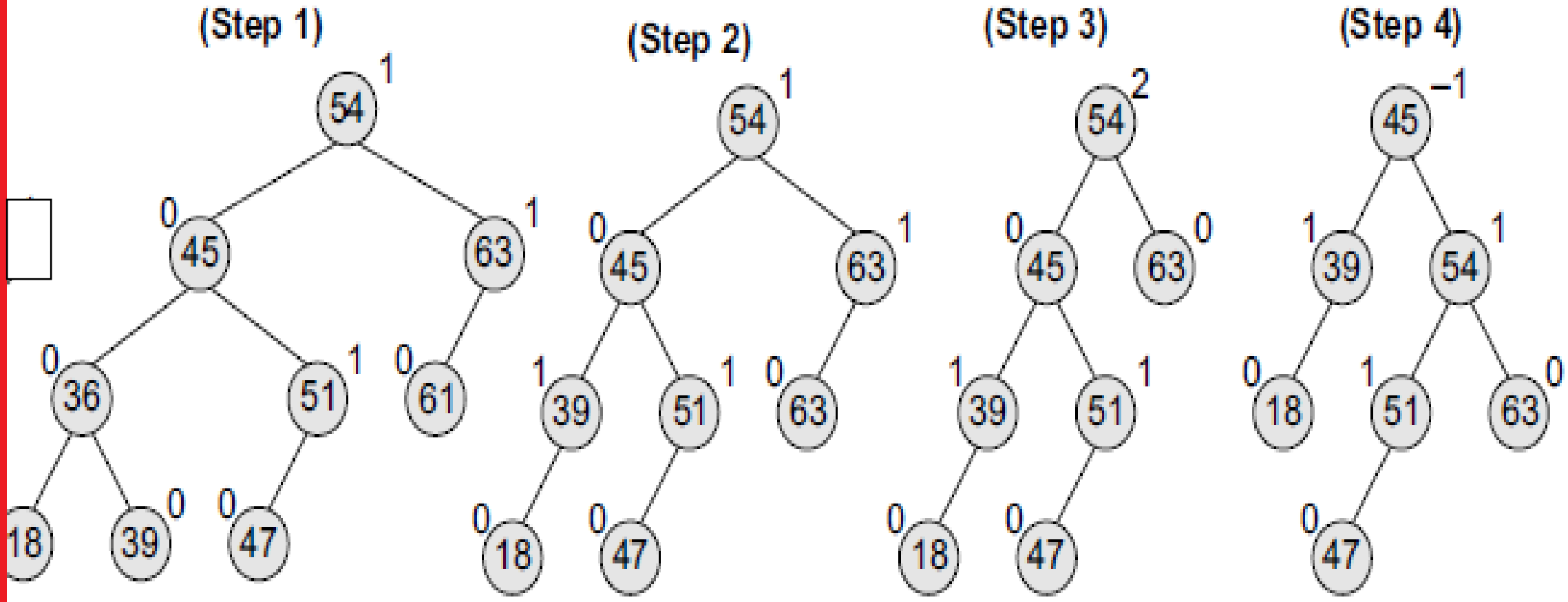


Deleting a Node from an AVL Tree

- Delete nodes 52, 36, and 61 from the AVL tree given in Fig.



Deleting a Node from an AVL Tree



Pros and Cons of AVL Trees

- Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of `insert` and `delete`

- Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. If *amortized* logarithmic time is enough, use splay trees (also in the text, not covered in this class)