Indian Institute of Information Technology Kottayam

AVL Trees

By

Arun Cyril Jose

AVL Tree

- **Self-balancing binary search tree** invented by G.M. Adelson-Velsky and E.M. Landis in 1962.
- Heights of the two sub-trees of a node may differ by at most one.
- Also known as a height-balanced tree.

- Takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case.

- Structure is similar to Binary Search Tree.

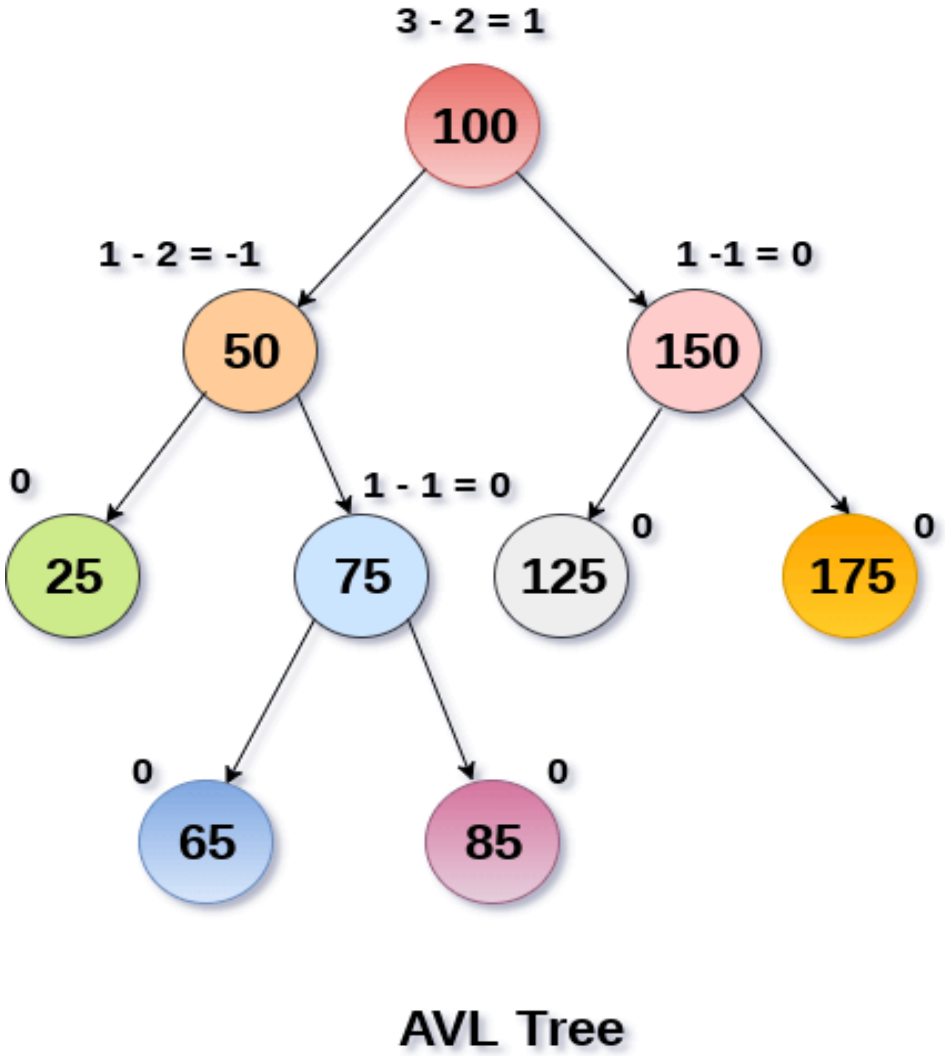
AVL Tree

- Stores an additional variable called the *BalanceFactor*.
- Calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.
- $Balance\ factor = Height\ (left\ sub\text{-}tree) - Height\ (right\ sub\text{-}tree)$
- A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced.
- Node with any other balance factor indicates an *unbalanced tree* and requires rebalancing to be an AVL tree.

AVL Tree

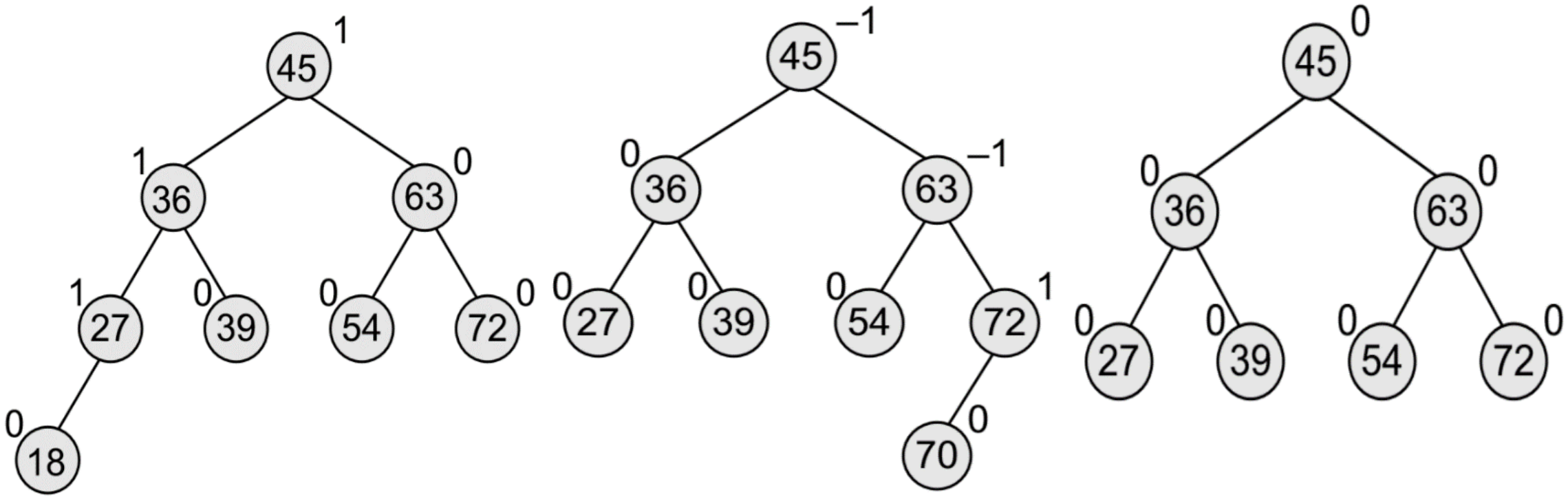
- If the balance factor of a node is 1, then height of the left sub-tree is one level higher than that of the right sub-tree. Called as a **left-heavy tree**.
- If the balance factor of a node is 0, then height of the left sub-tree is equal to the height of the right sub-tree.
- If the balance factor of a node is –1, then height of the left sub-tree is one level lower than that of the right sub-tree. Called as a **right-heavy tree**.

AVL Tree



AVL Tree

AVL Tree


45
1
1
36
0
63
1
27
0
39
0
54
0
72
0
18
45
−1
0
36
−1
63
0
27
0
39
0
54
1
72
0
70
45
0
0
36
0
63
0
27
0
39
0
54
0
72

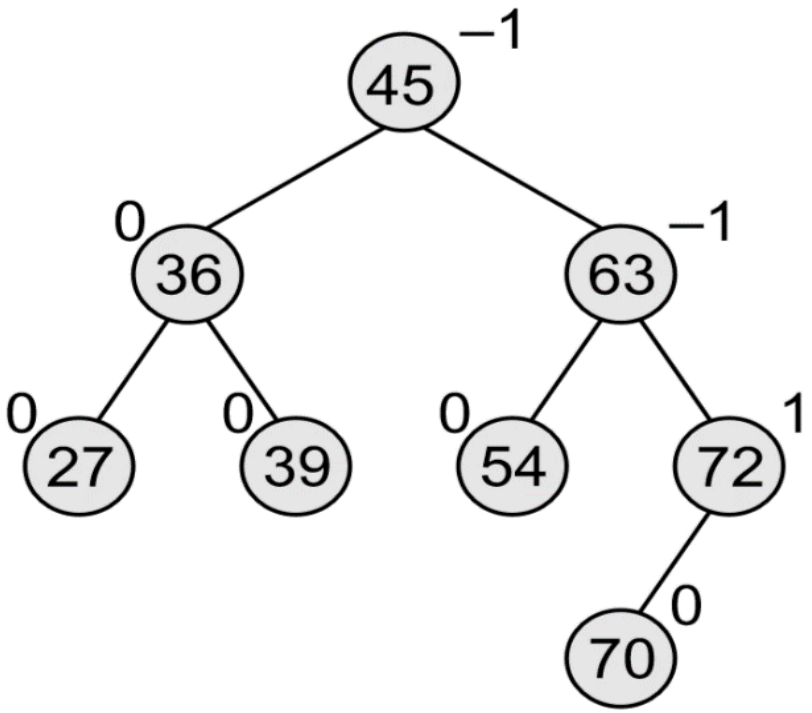
Searching in AVL Tree

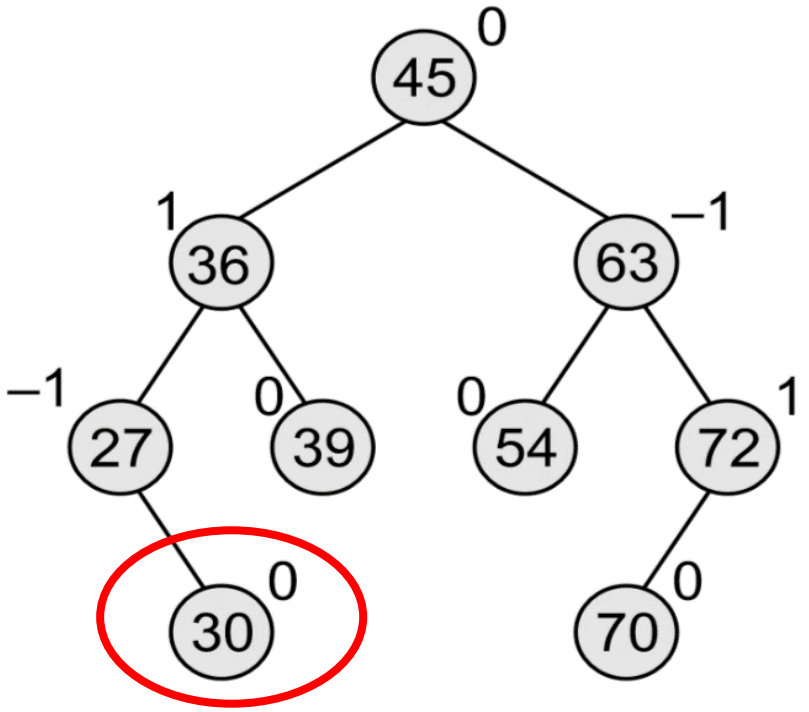
- Searching in an AVL tree is performed exactly same as that in a binary search tree.

Insertion of a new node into AVL Tree

- Same way as it is done in a binary search tree.
- New node is always inserted as the leaf node. It is usually followed by an additional step of rotation to restore the balance of the tree.
- If insertion of the new node does not disturb the balance factor then rotations are not required.
- New node is inserted as the leaf node, so it will always have a balance factor of zero.
- How about the balance factors of the nodes that lie in the path between the root of the tree and the newly inserted node?

Insertion of a new node into AVL Tree

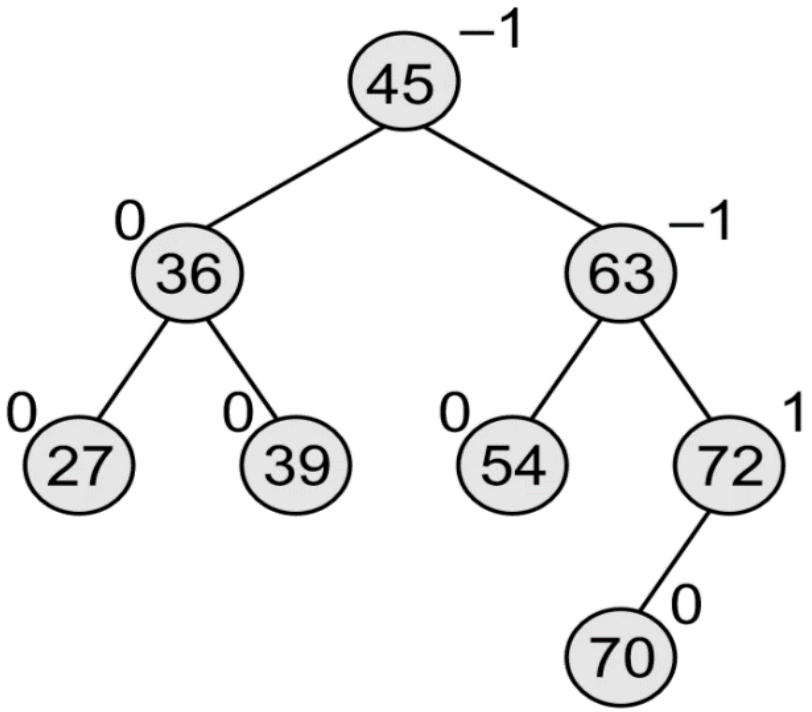


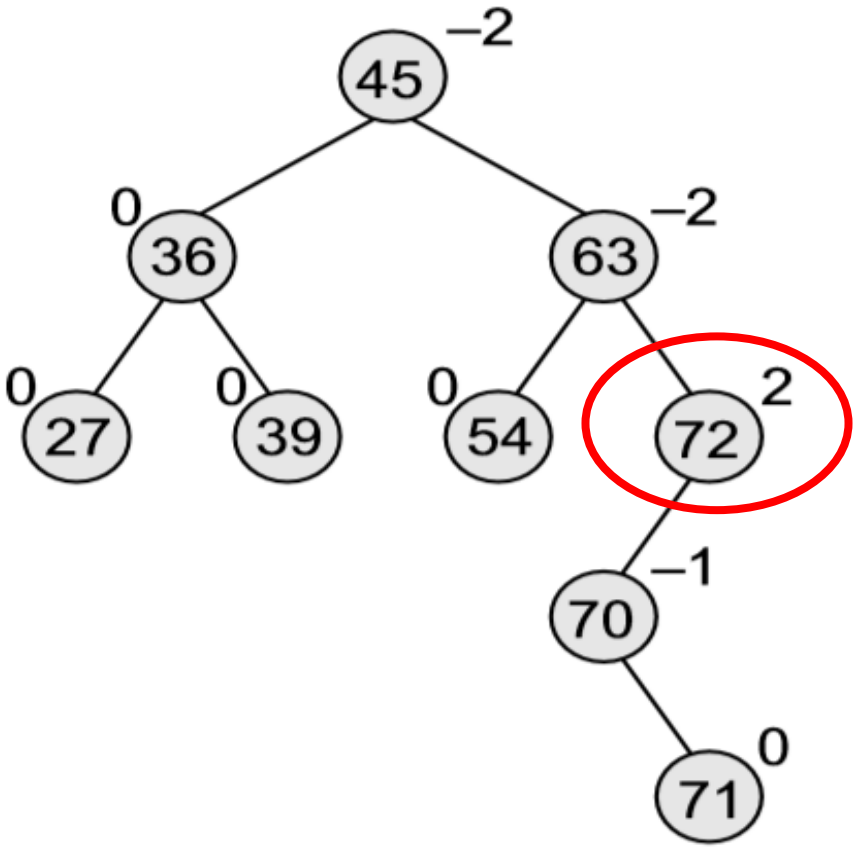


Insertion of a new node into AVL Tree

- After insertion these things can happen:
- 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*.

Insertion of a new node into AVL Tree

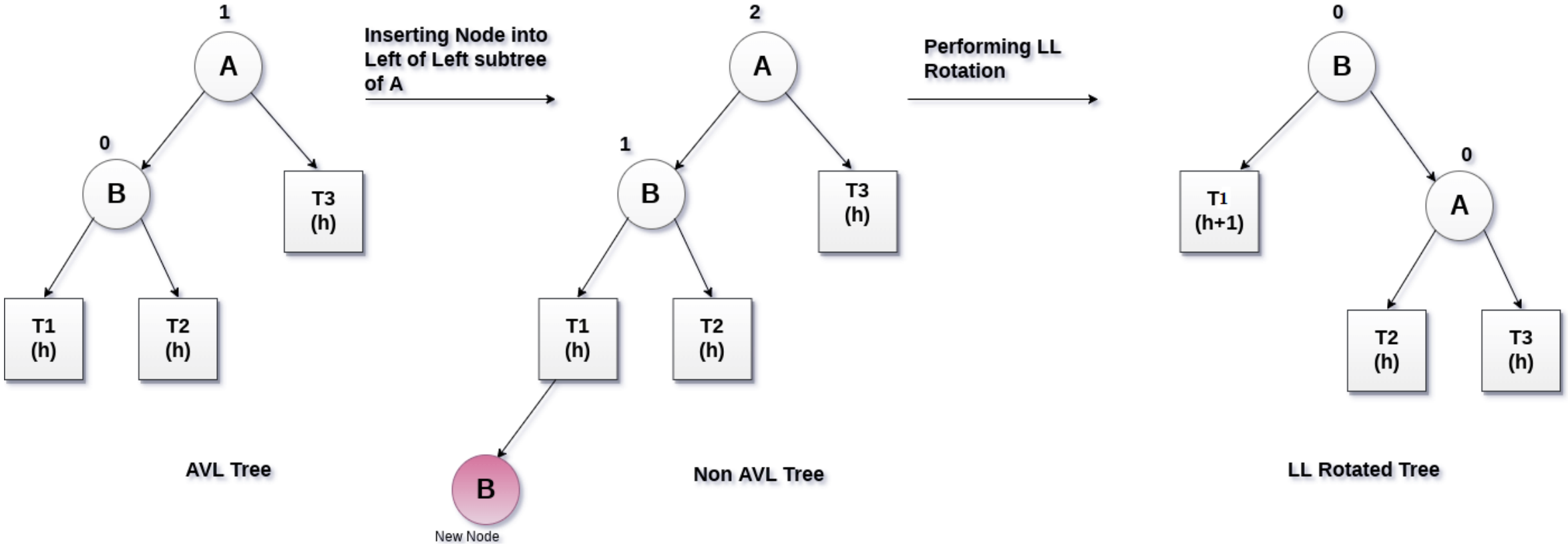




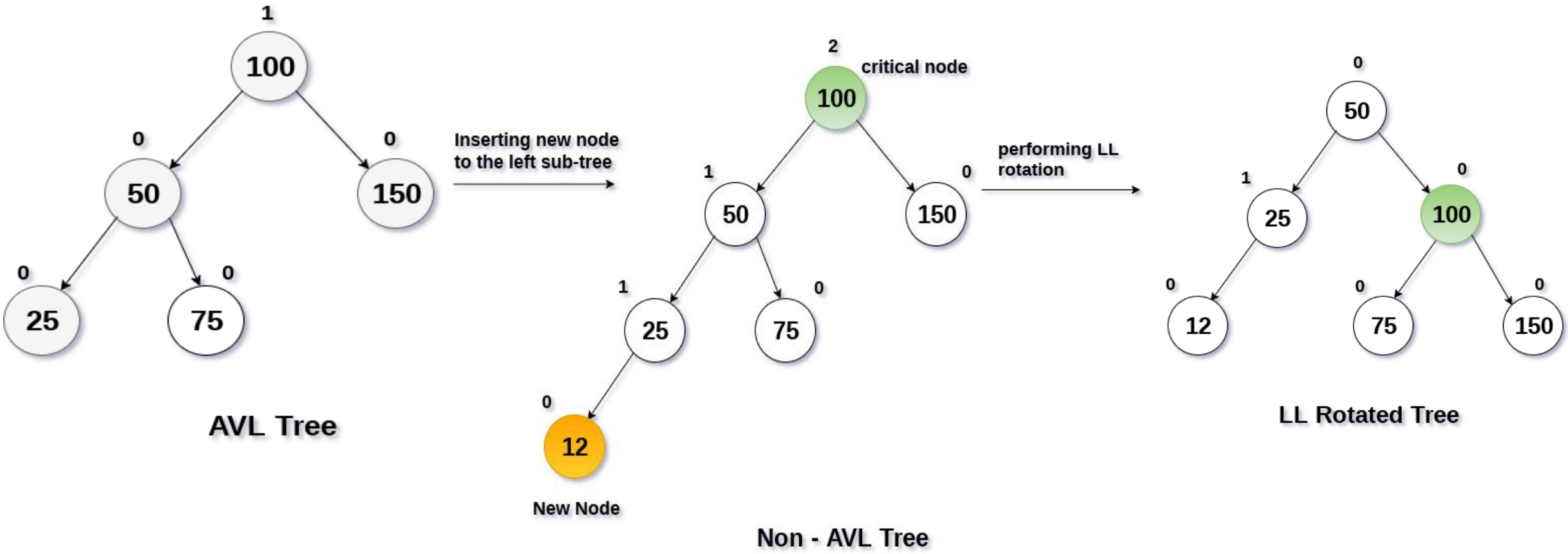
Insertion of a new node into AVL Tree

- Determine which type of rotation has to be done.
- Four types of rebalancing rotations are there:
- **LL Case**: The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
- **RR Case**: The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
- **LR Case**: The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
- **RL Case**: The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

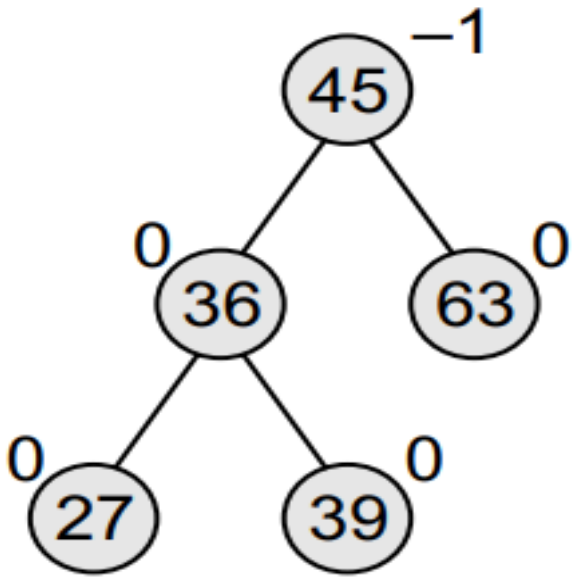
Insertion into AVL Tree: LL Case

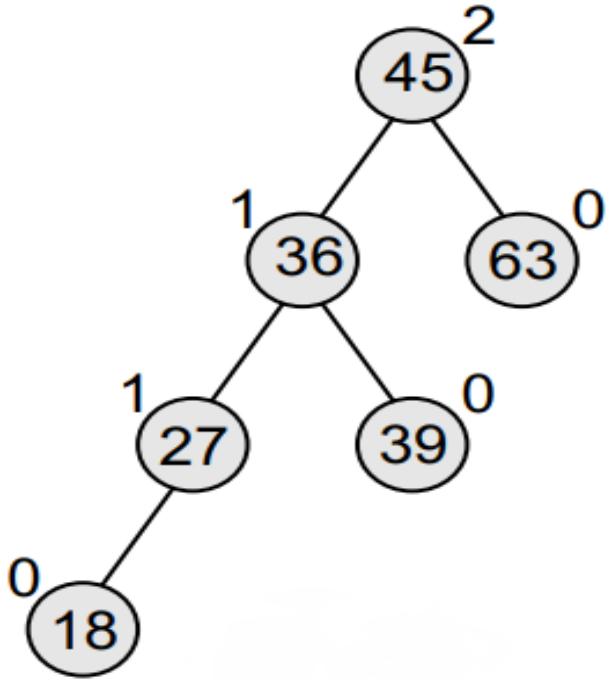


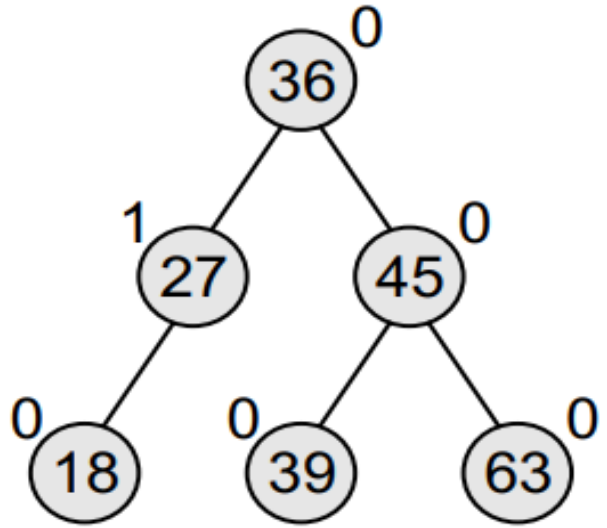
Insertion into AVL Tree: LL Case



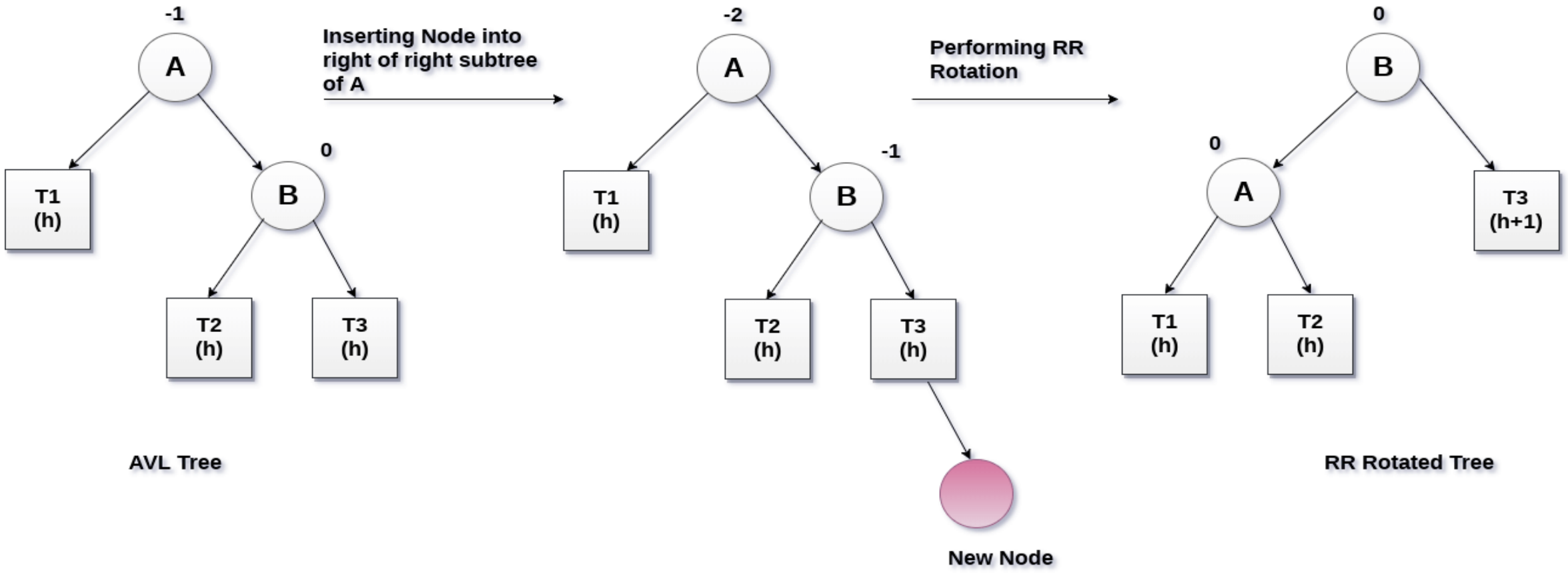
Insertion into AVL Tree: LL Case



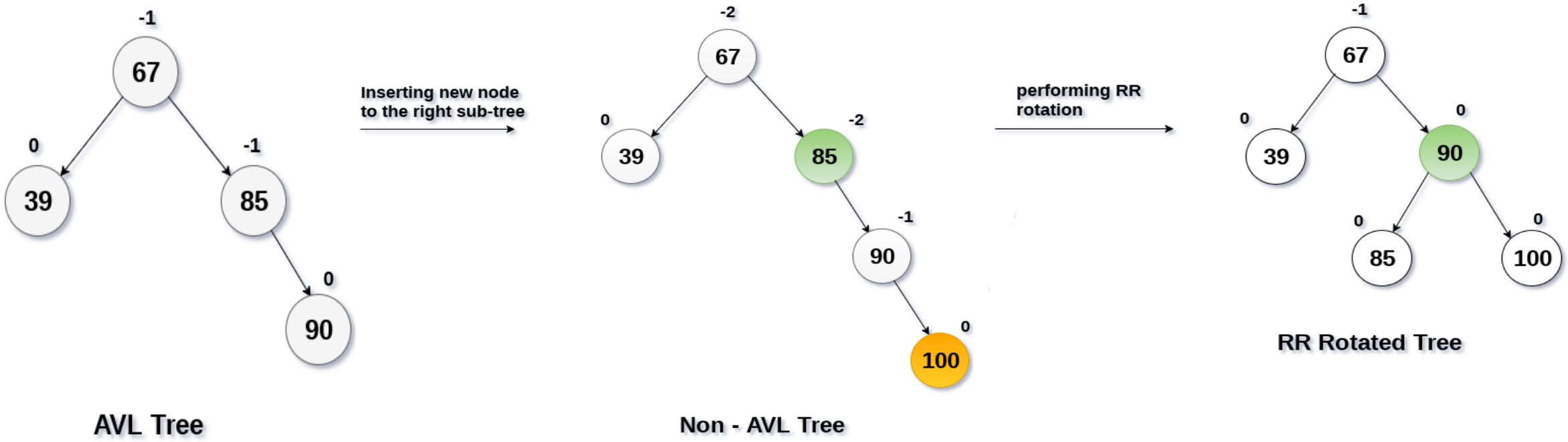




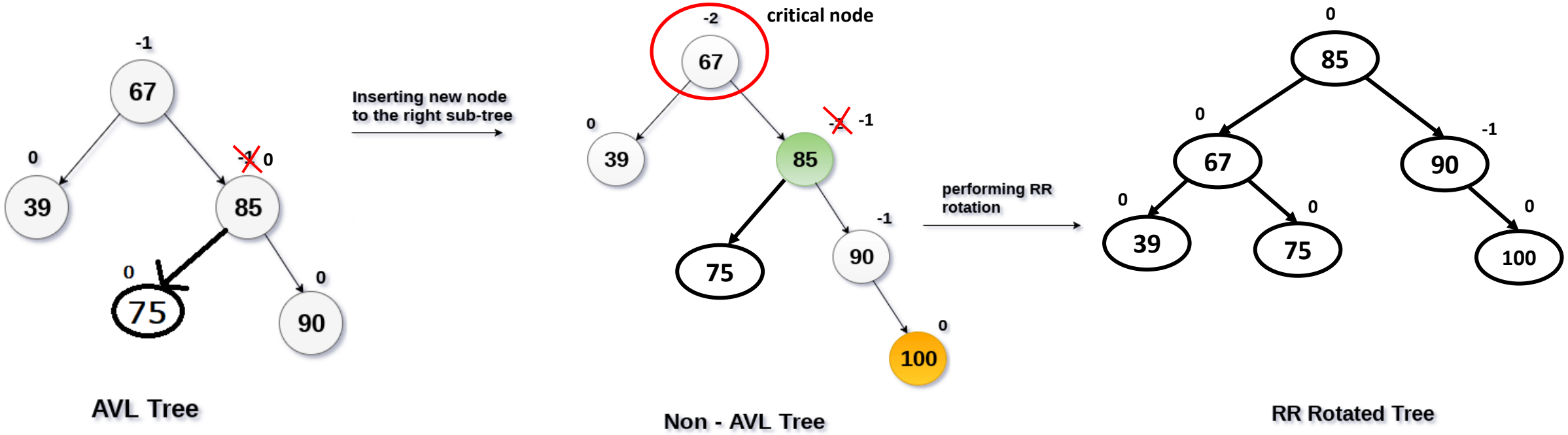
Insertion into AVL Tree: RR Case



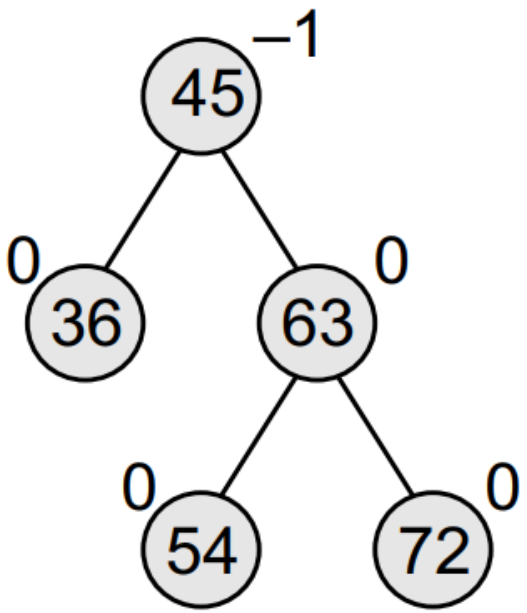
Insertion into AVL Tree: RR Case

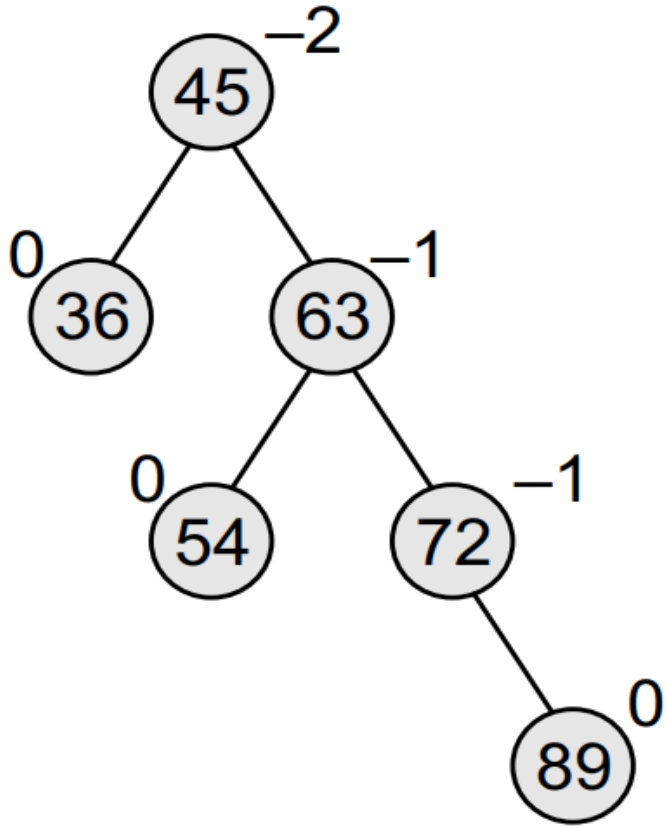


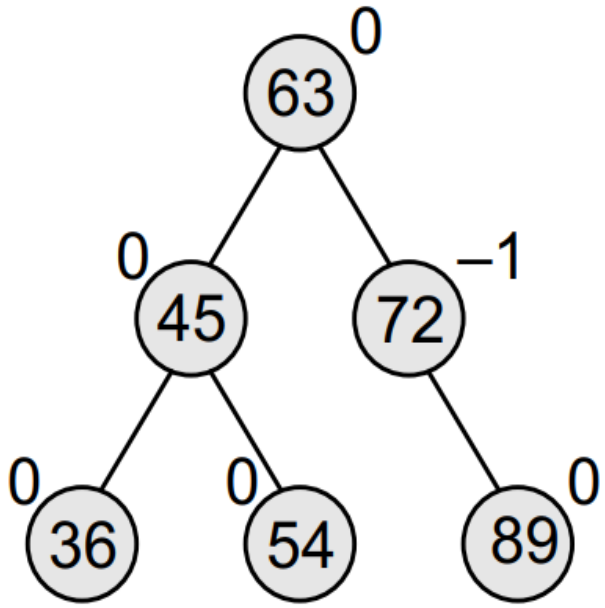
Insertion into AVL Tree: RR Case



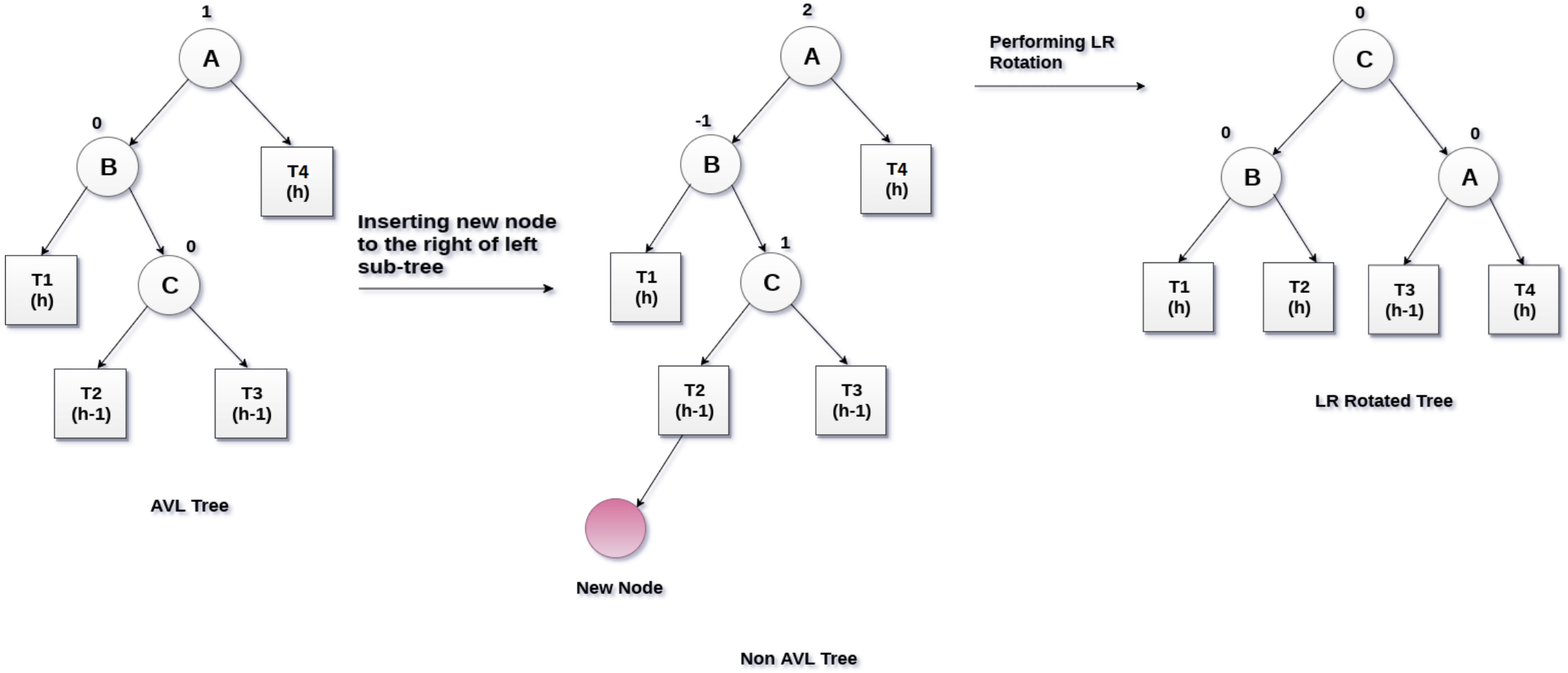
Insertion into AVL Tree: RR Case



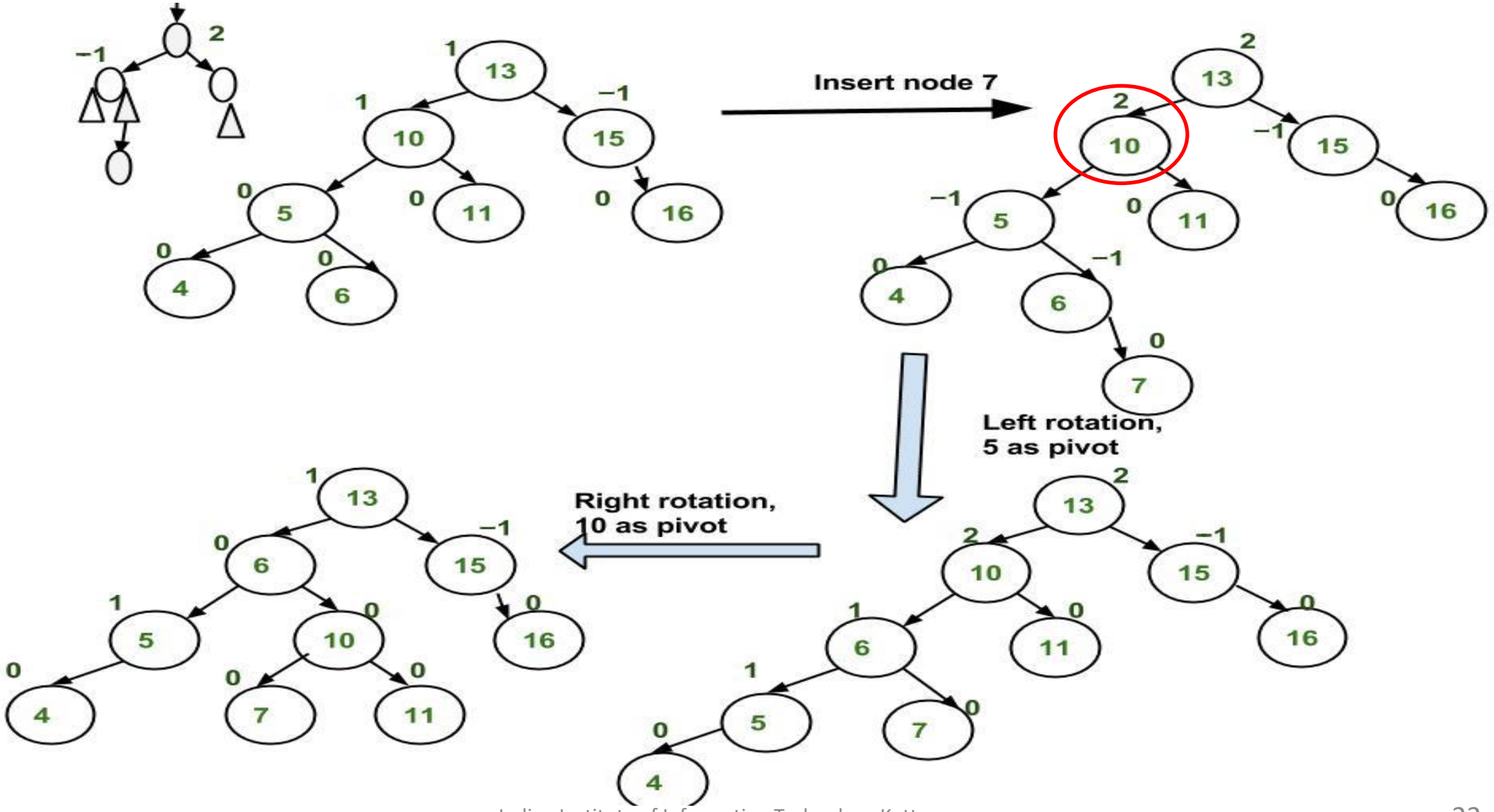




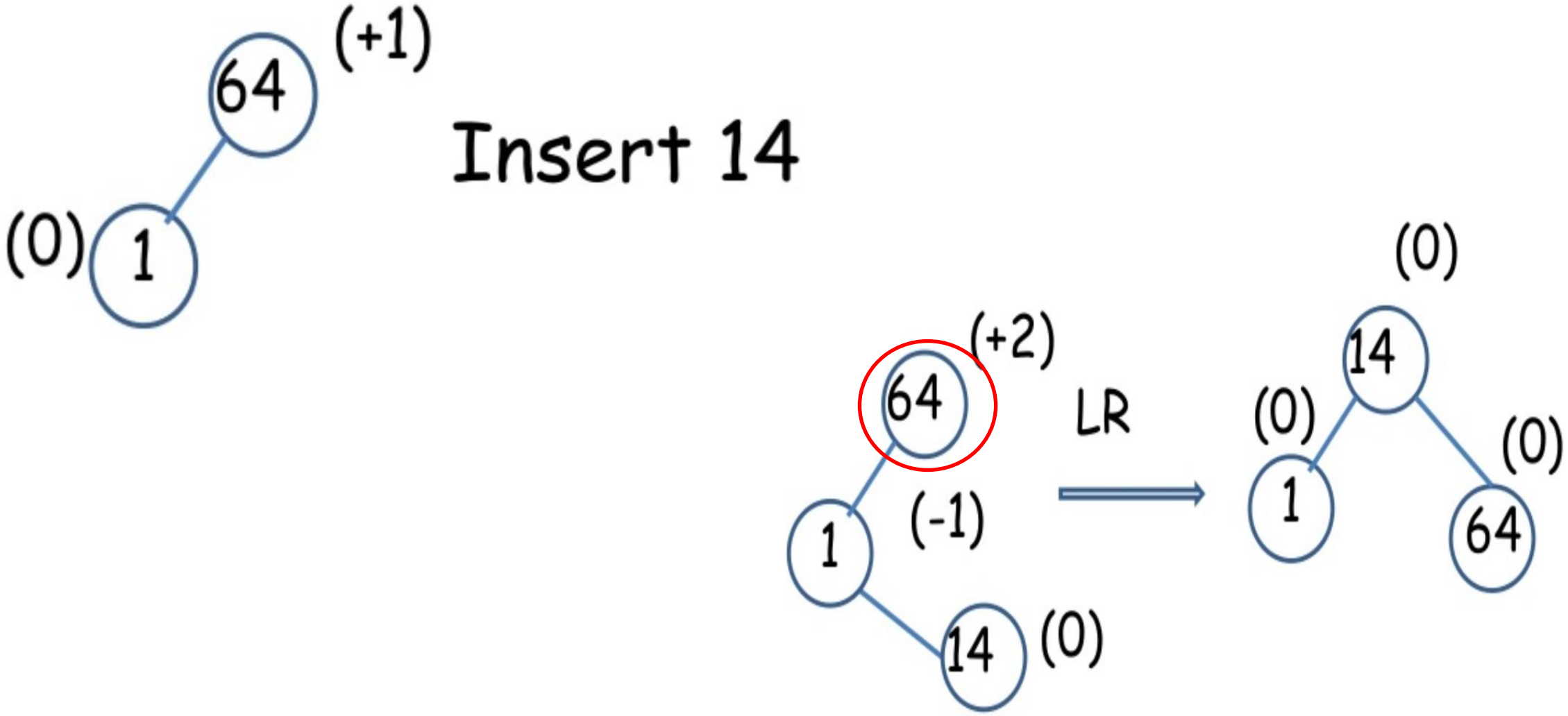
Insertion into AVL Tree: LR Case



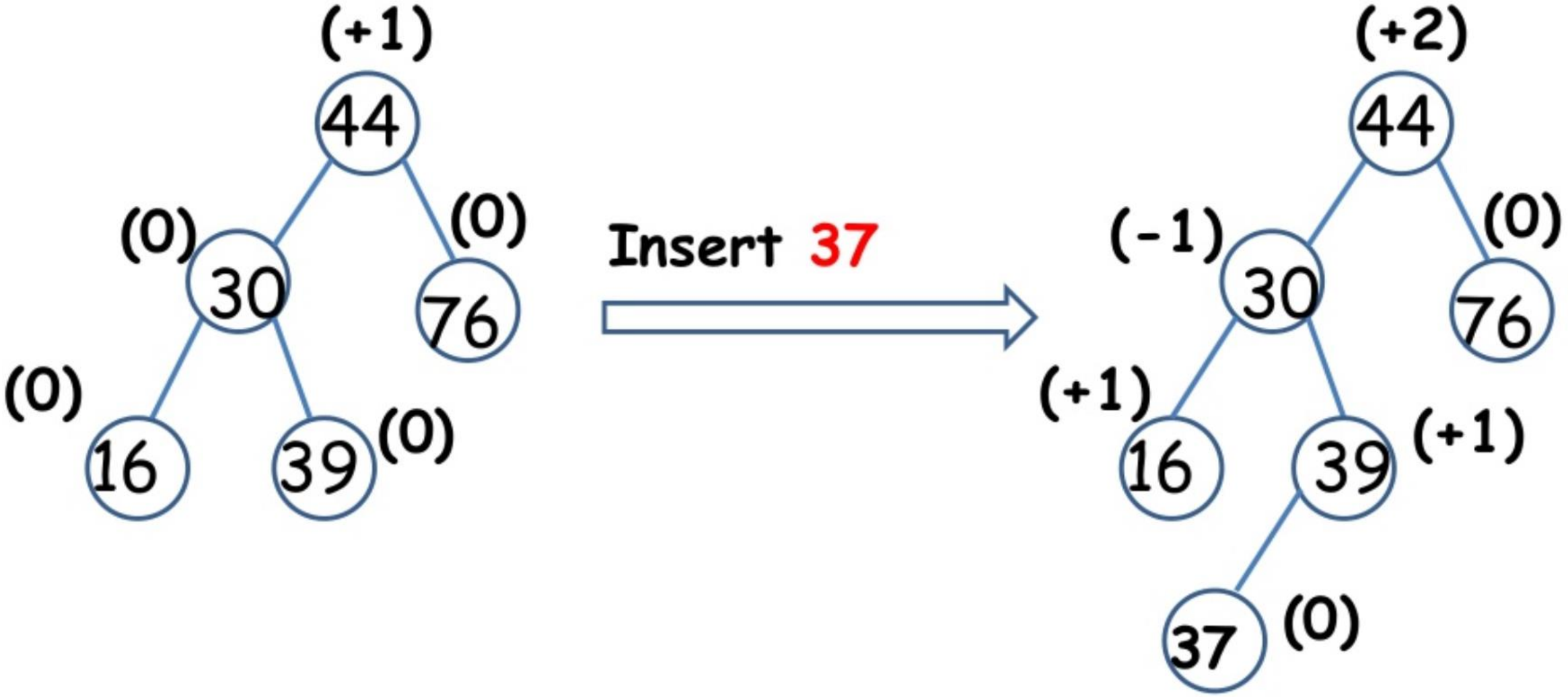
Insertion into AVL Tree: LR Case



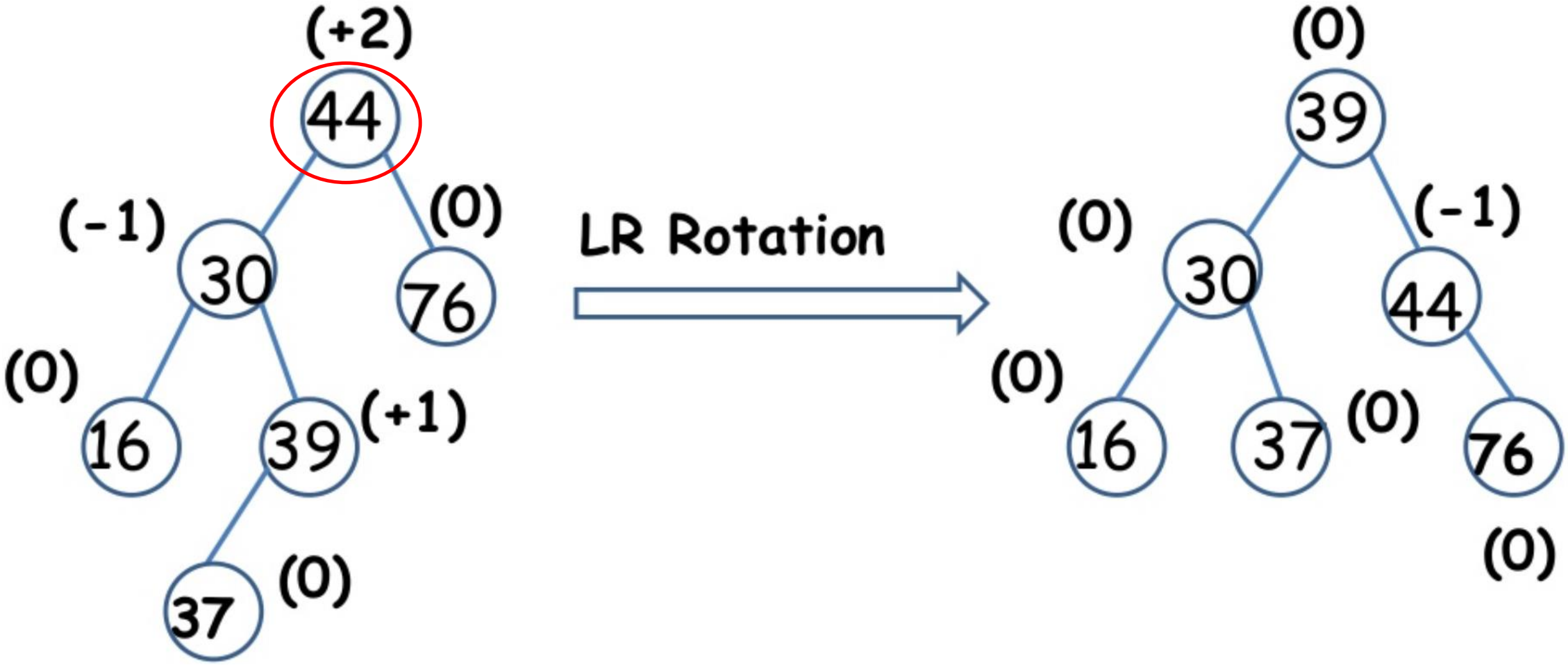
Insertion into AVL Tree: LR Case



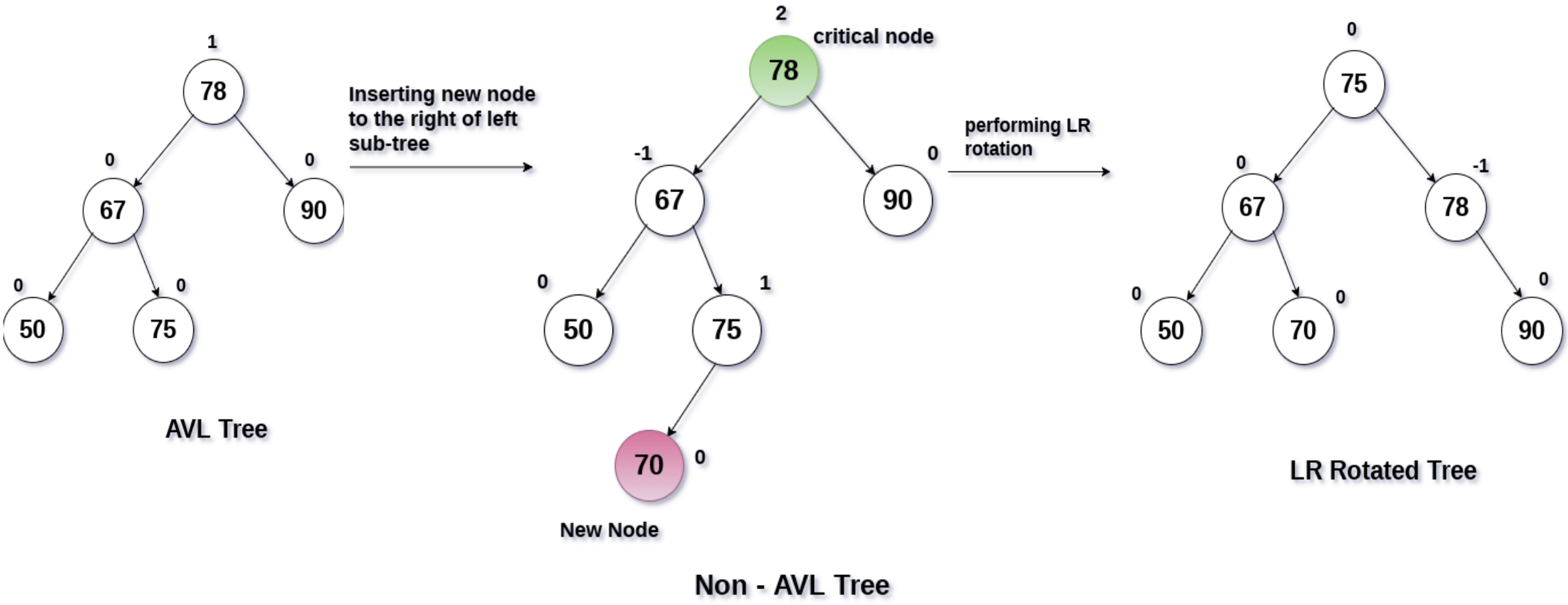
Insertion into AVL Tree: LR Case



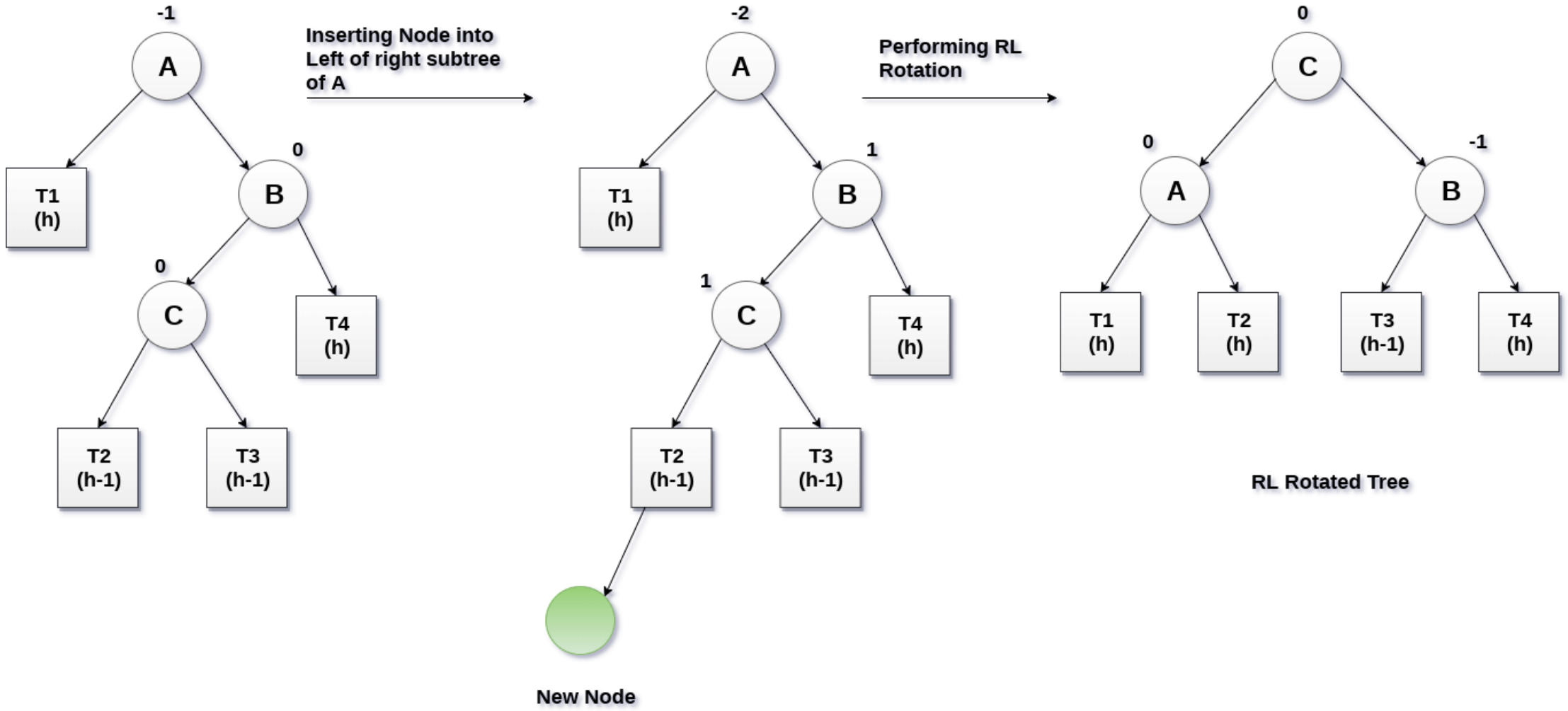
Insertion into AVL Tree: LR Case



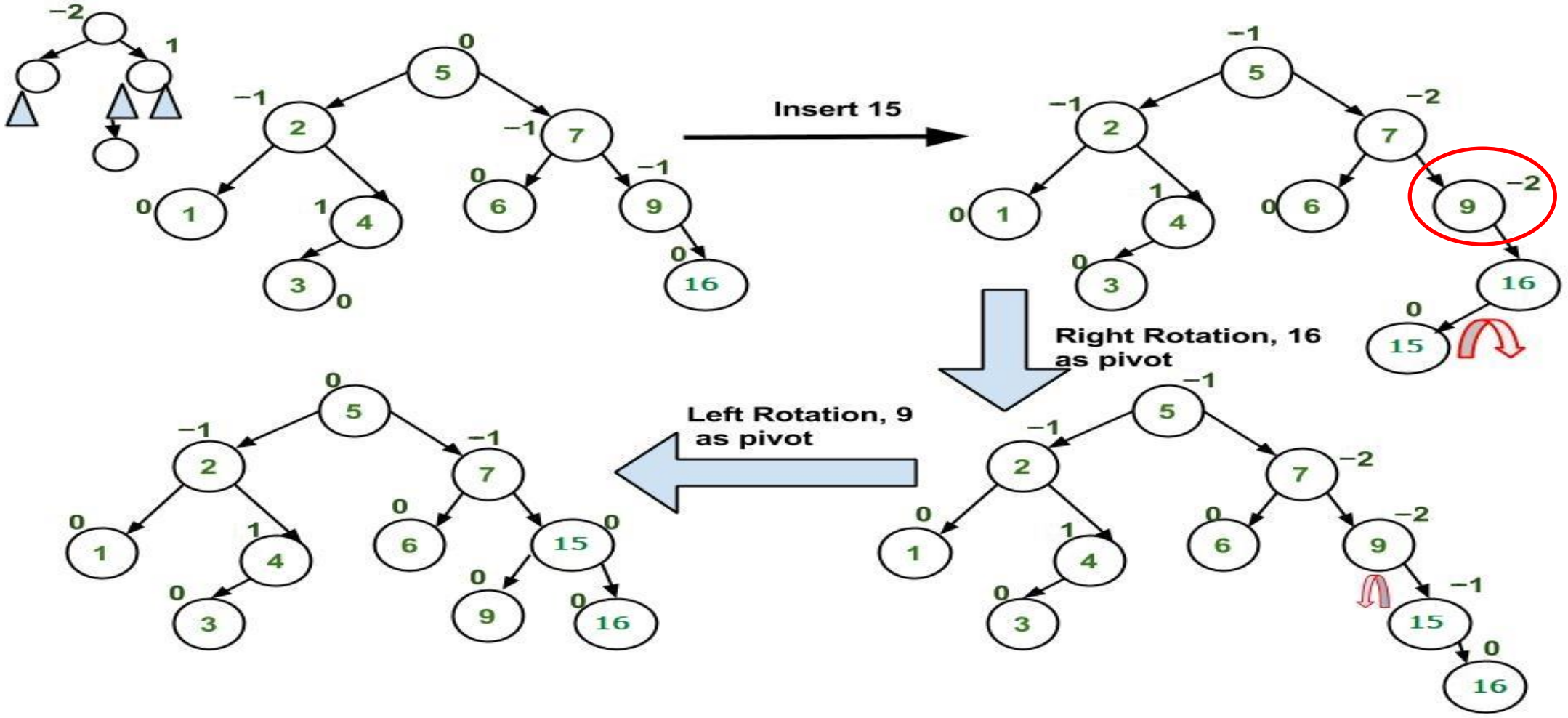
Insertion into AVL Tree: LR Case



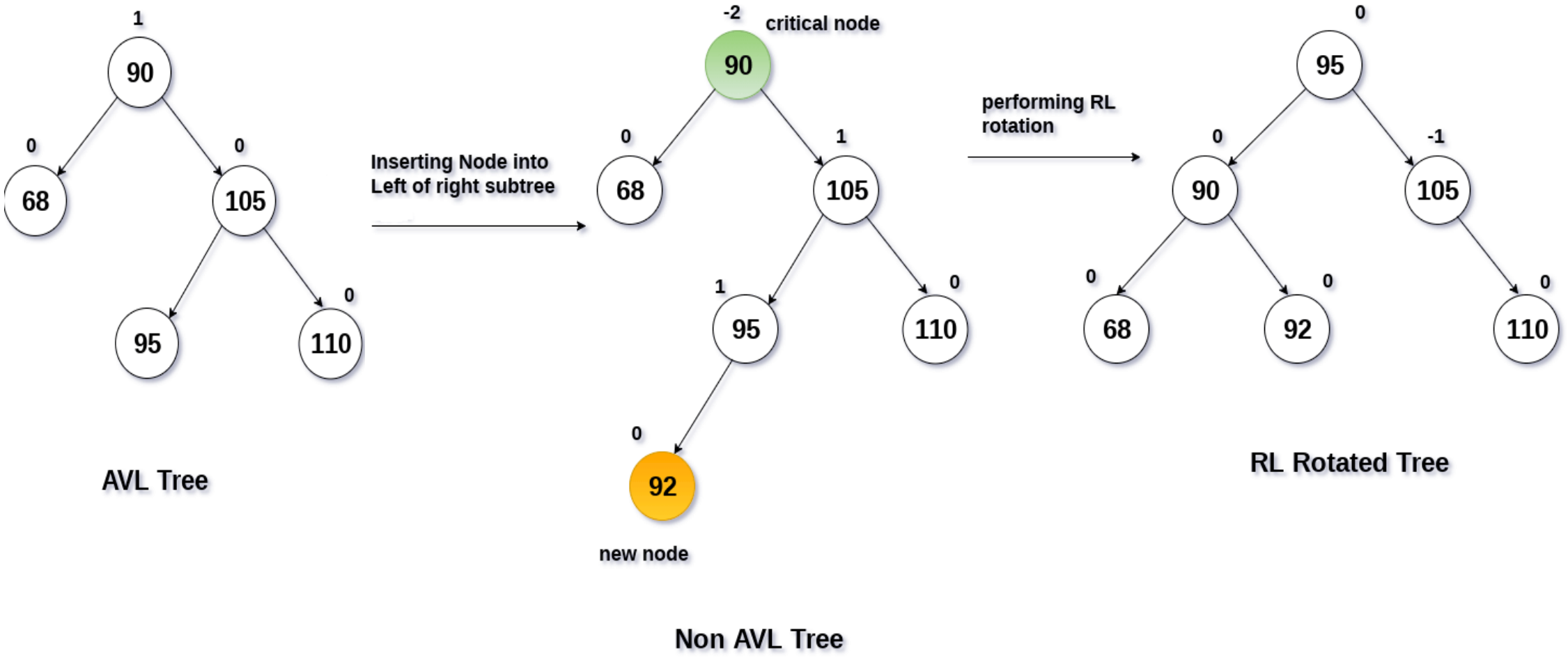
Insertion into AVL Tree: RL Case



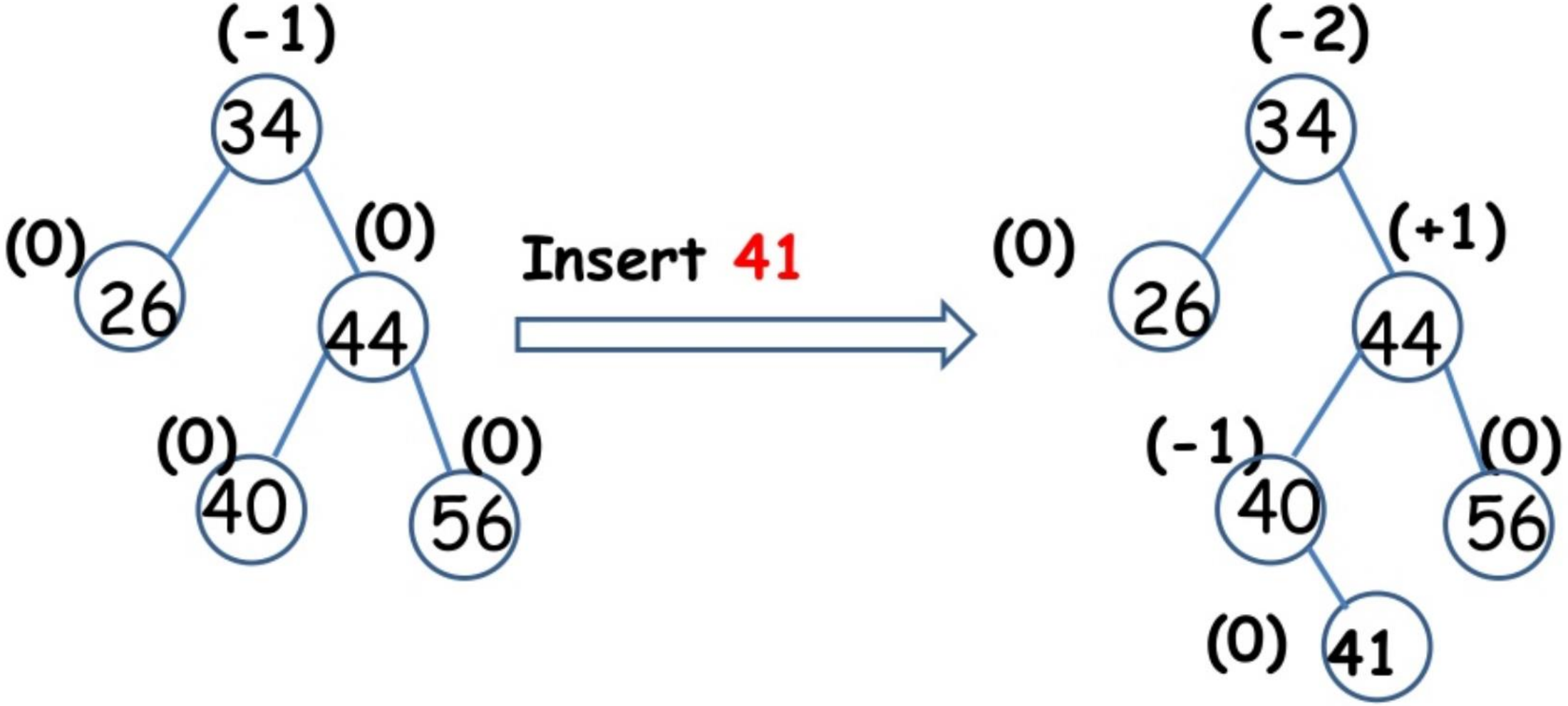
Insertion into AVL Tree: RL Case



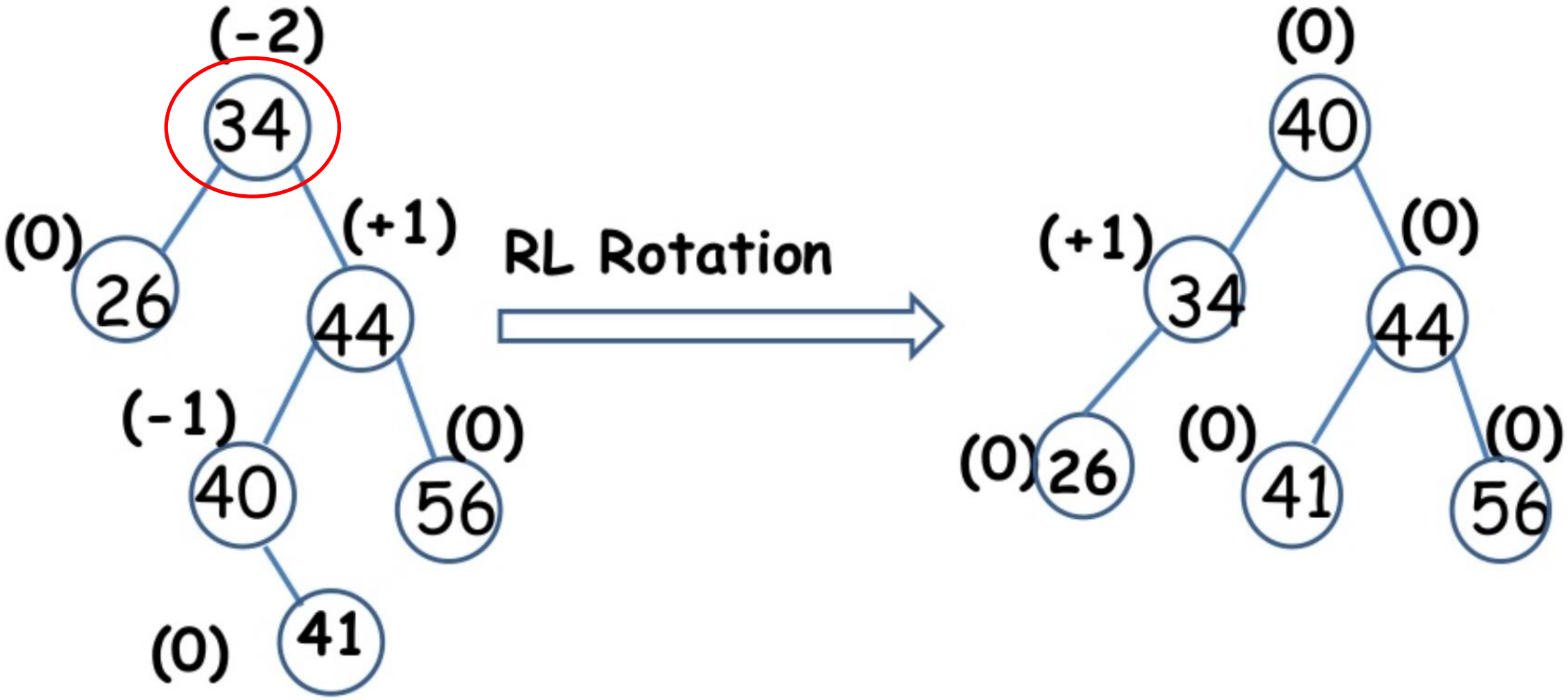
Insertion into AVL Tree: RL Case



Insertion into AVL Tree: RL Case



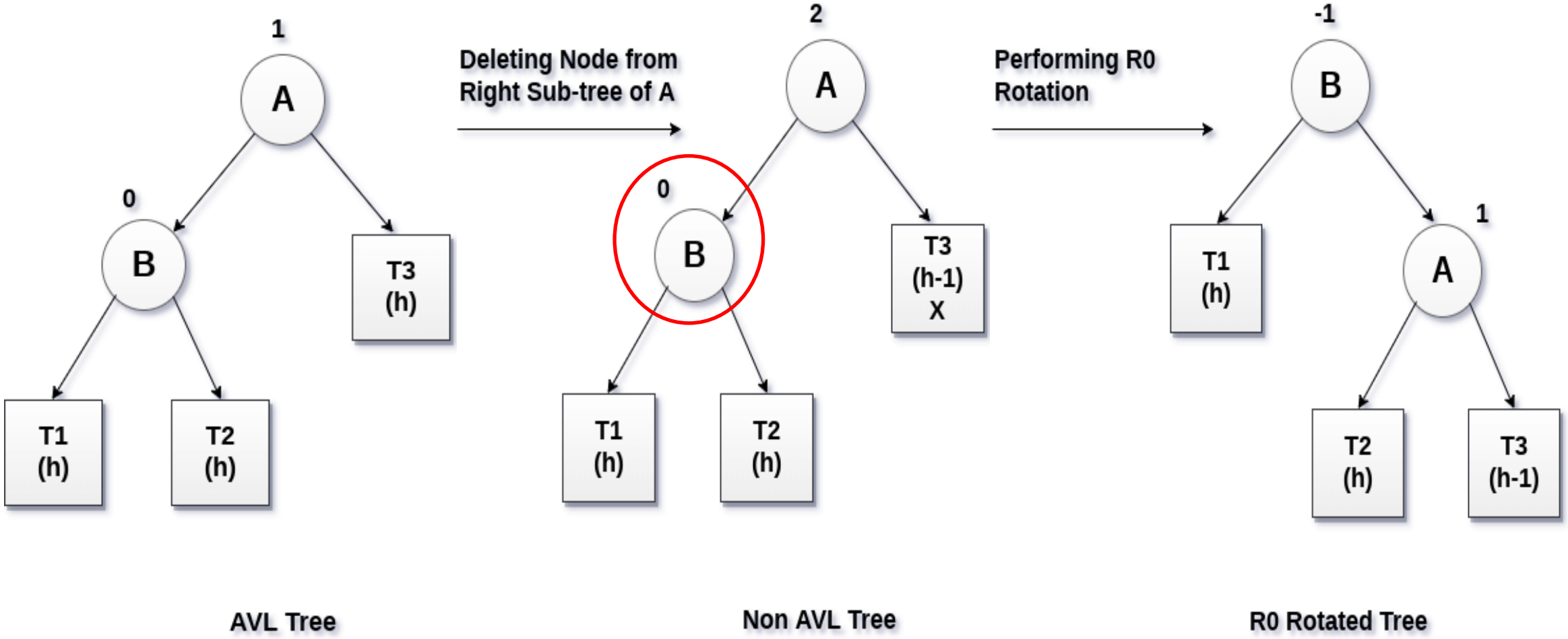
Insertion into AVL Tree: RL Case



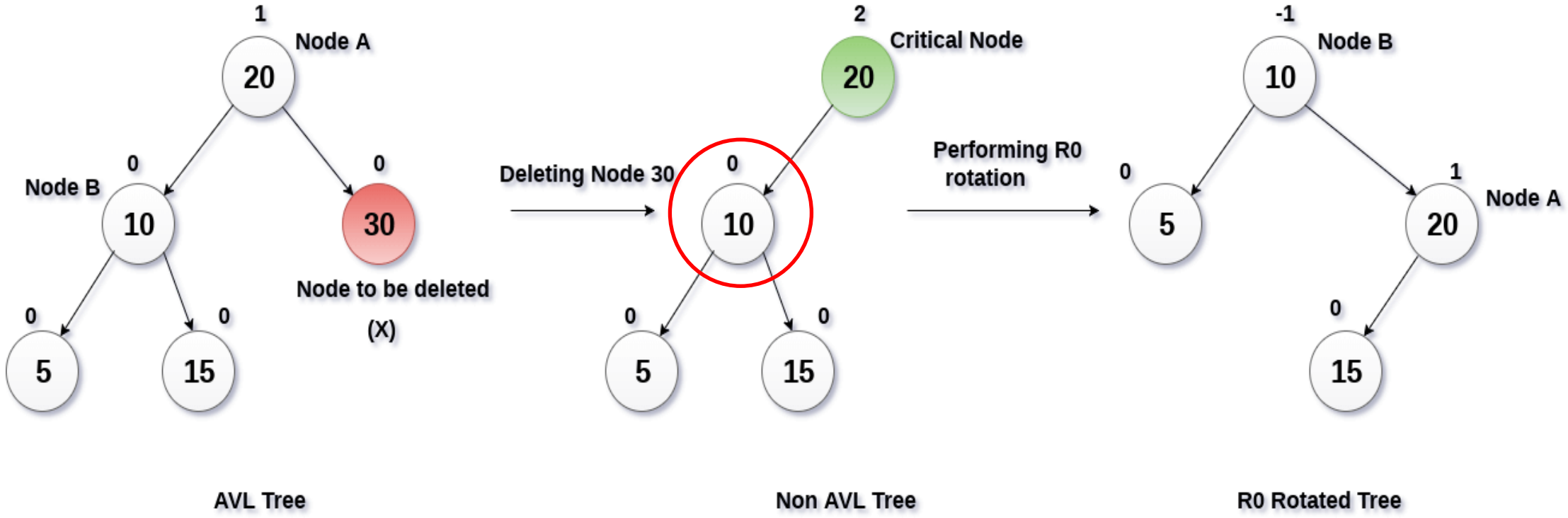
Deletion from AVL Tree

- Similar to deletion from binary tree.
- Deletion may disturb the AVL-ness of the tree, so to rebalance the AVL tree, we need to perform rotations.
- Two classes of rotations: **R** rotation and **L** rotation.
- On deletion of node X from the AVL tree, if node A becomes the critical node, then the type of rotation depends on whether X is in the left sub-tree of A or in its right sub-tree.
- If 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.

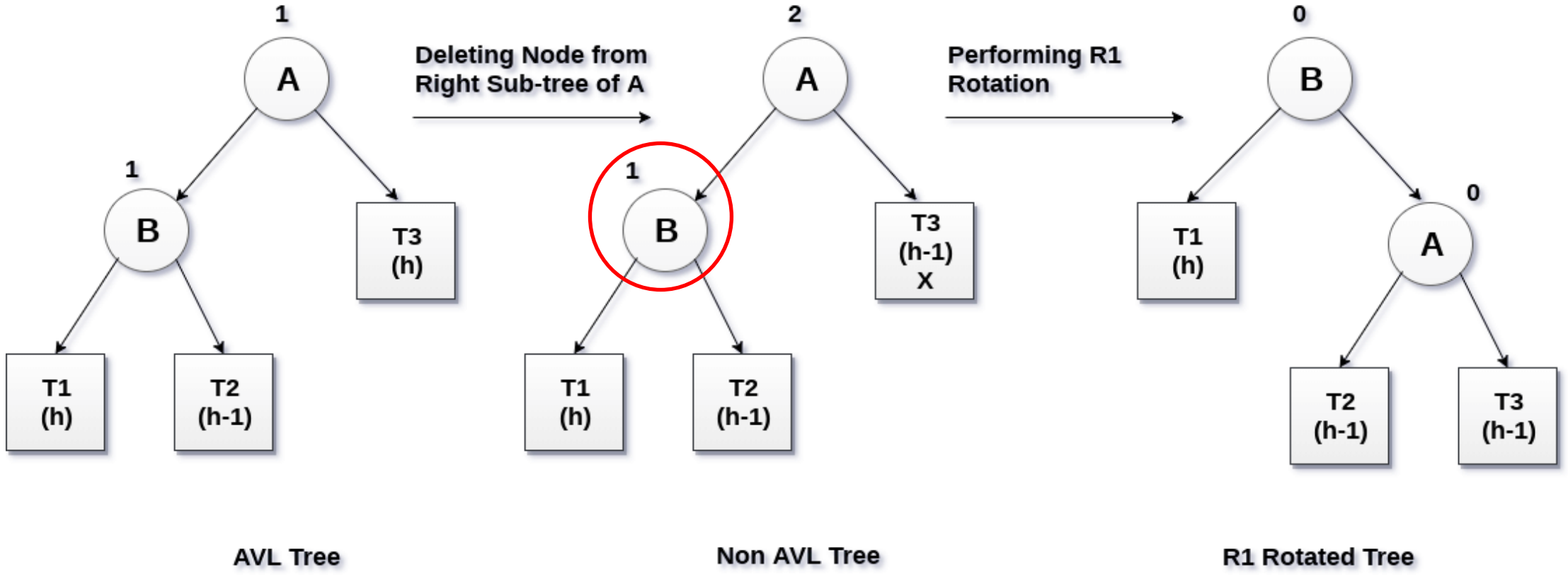
Deletion from AVL Tree: R0 Rotation



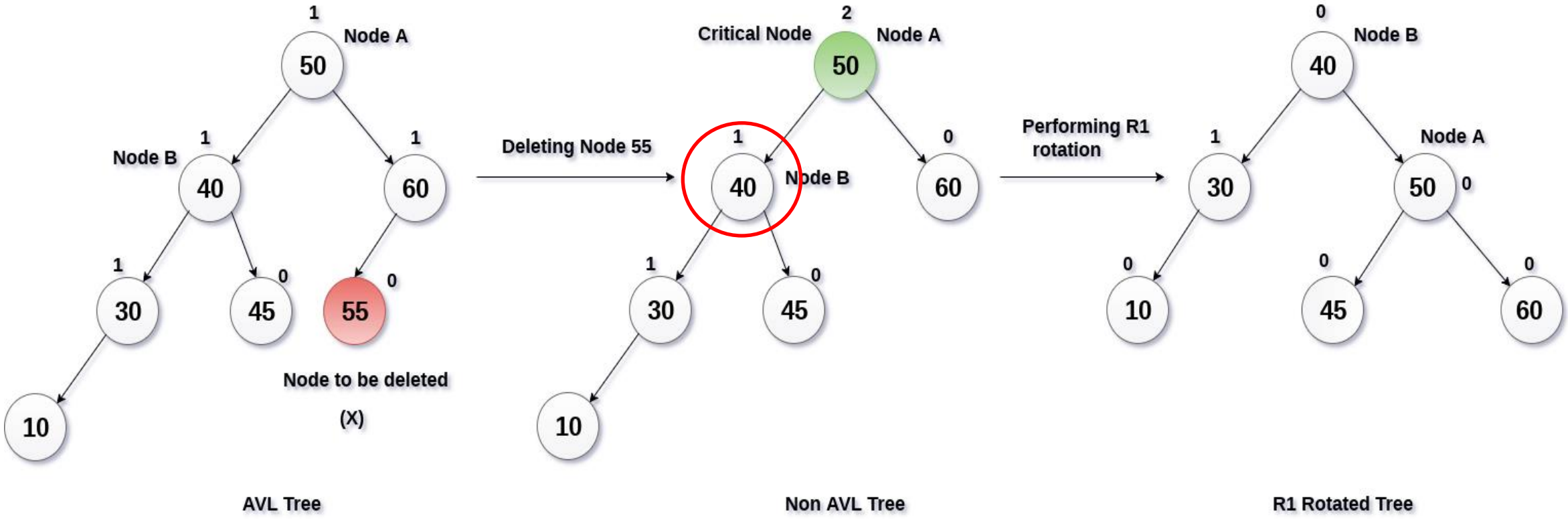
Deletion from AVL Tree: R0 Rotation



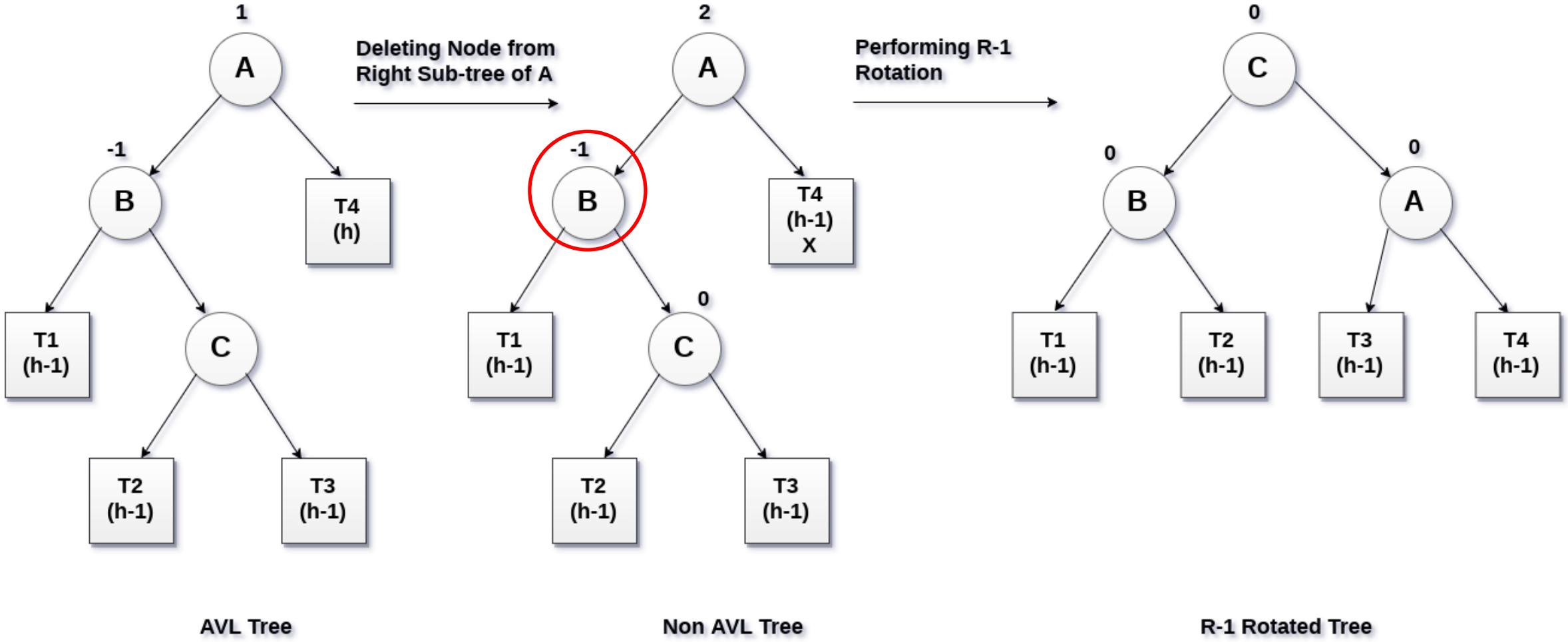
Deletion from AVL Tree: R1 Rotation



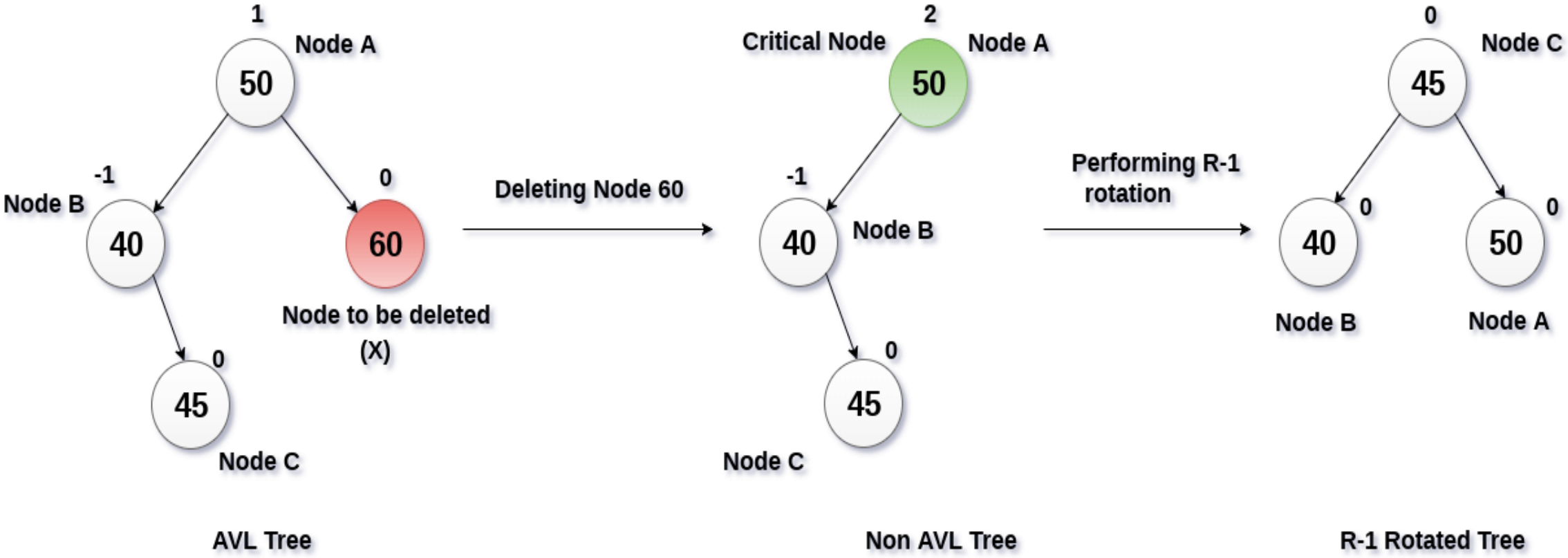
Deletion from AVL Tree: R1 Rotation



Deletion from AVL Tree: R-1 Rotation



Deletion from AVL Tree: R-1 Rotation



Deletion from AVL Tree: R-1 Rotation