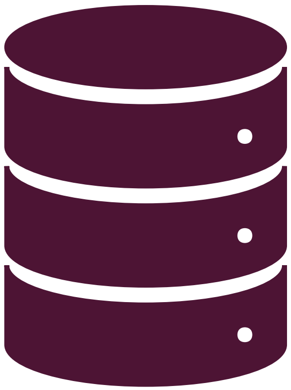


DATA STRUCTURES

WENYE LI

CUHK-SZ

AVL TREE

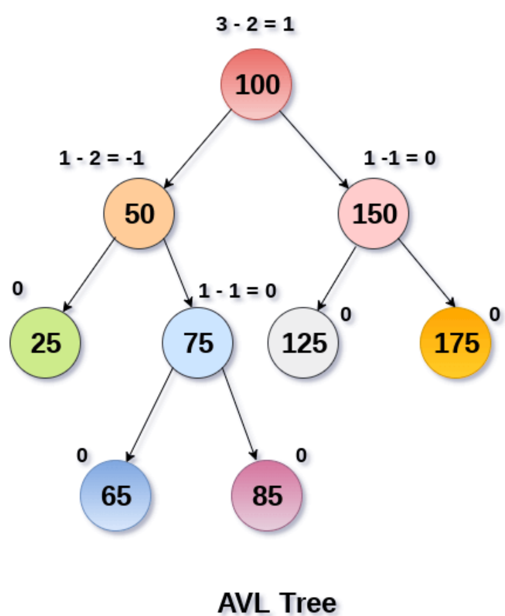
- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962.

- Height: the length of the longest path from a node to a leaf.
  - All leaves have a height of 0
  - An empty tree has a height of –1

- AVL Tree: **height balanced binary search tree** in which **each node is associated with a balance factor** which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

- A tree is **balanced** if **balance factor of each node is in between -1 to 1**, otherwise, unbalanced.

BALANCE FACTOR



- **Balance Factor (k) = height (left(k)) - height (right(k))**
- balance factor of any node is 0: left sub-tree and right sub-tree contain equal height.
- balance factor of any node is -1: left sub-tree is one level lower than right sub-tree.

WHY AVL TREE

- AVL tree controls the height of the BST by not letting it to be skewed.
- The time taken for all operations in a BST of height h is O(h).
  - However, it can be extended to O(n) if the BST becomes skewed (i.e. worst case).
  - By limiting this height to log n, AVL tree imposes an upper bound on each operation to be O(log n) where n is the number of nodes.

- Note: we will explain the O() notation later.

OPERATIONS ON AVL TREE

- AVL tree is also a binary search tree
  - All operations are performed in the same way as they are performed in a BST.
  - Searching and traversing do not lead to the violation in property of AVL tree.
  - **Insertion** and **deletion** can violate this property and therefore, need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

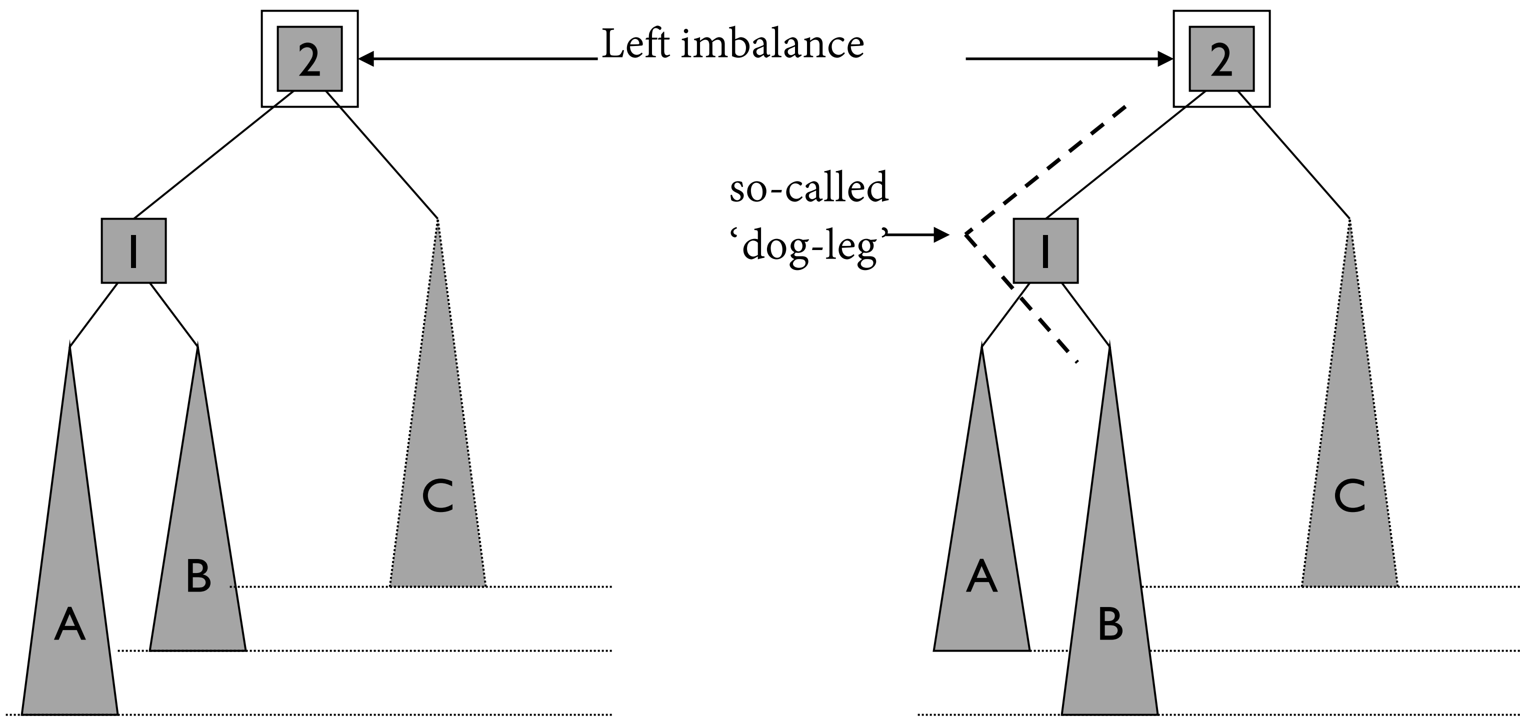
ADDITION

- We perform rotation in AVL tree only when Balance Factor is other than -1, 0, and 1.
- Four types of rotations:
  - **L L** rotation: Inserted node is in the left subtree of left subtree of A
  - **R R** rotation : Inserted node is in the right subtree of right subtree of A
  - **L R** rotation : Inserted node is in the right subtree of left subtree of A
  - **R L** rotation : Inserted node is in the left subtree of right subtree of A
  - (where node A is the node whose balance Factor is other than -1, 0, 1.)
  - The first two rotations LL and RR are single rotations.
  - The next two rotations LR and RL are double rotations.

- For a tree to be unbalanced, minimum height must be at least 2

IMBALANCE

- Left-left (right-right)
- Left-right (right-left)



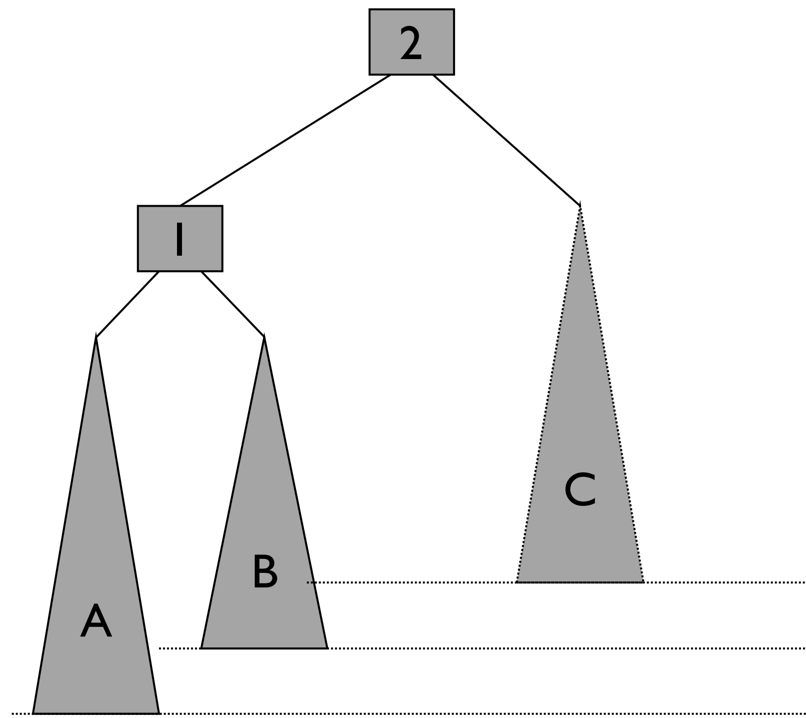
There are no other possibilities for the left (or right) subtree

LOCALISING THE PROBLEM

- Two principles:
  - Imbalance will only occur on the path from the inserted node to the root (only these nodes have had their subtrees altered - local problem)
  - Rebalancing should occur at the deepest unbalanced node (local solution too)

LEFT(LEFT) IMBALANCE (AND RIGHT(RIGHT) IMBALANCE)

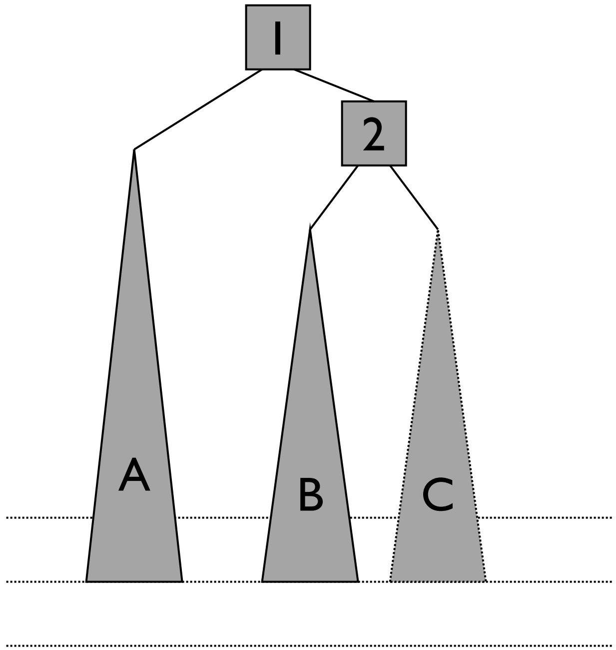
- Note the levels



- B and C have the same height
- A is one level higher
- Therefore make 1 the new root, 2 its right child and B and C the subtrees of 2

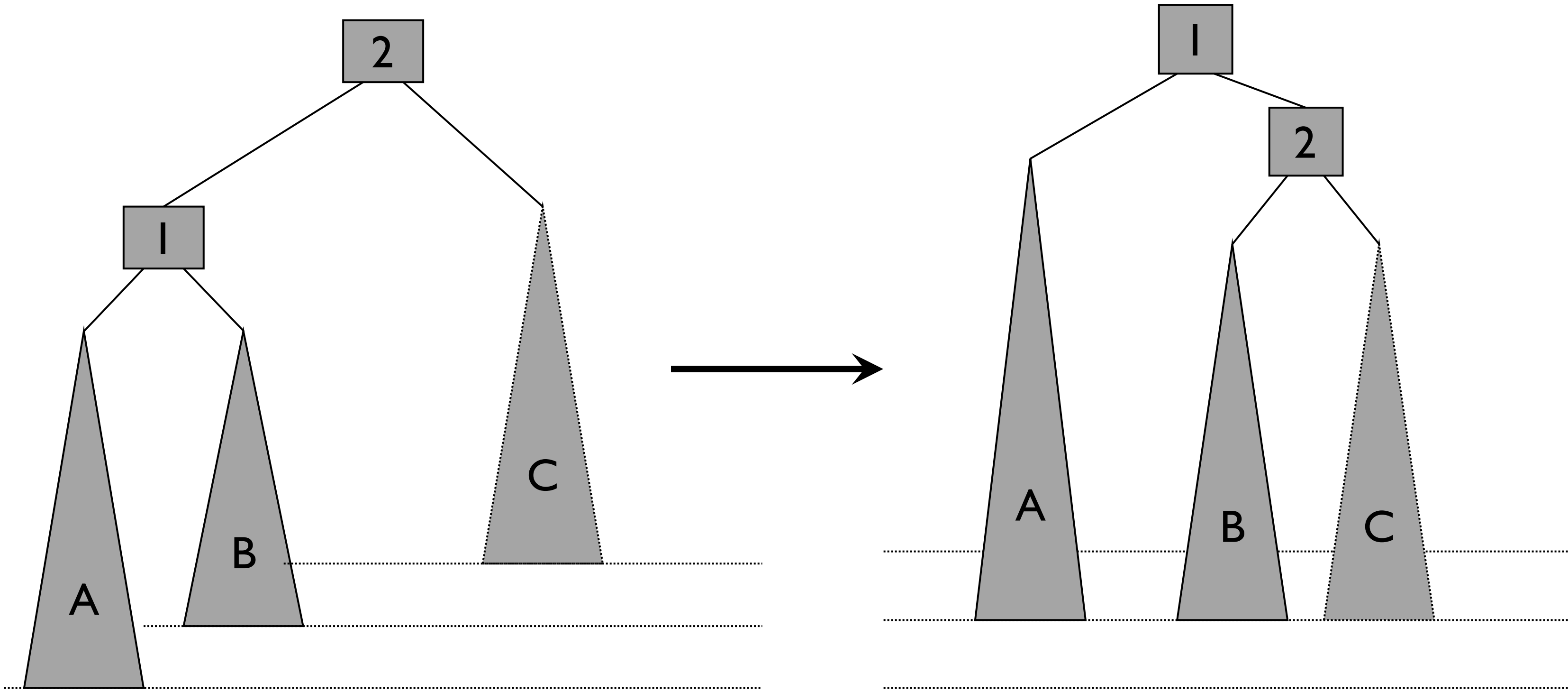
LEFT(LEFT) IMBALANCE (AND RIGHT(RIGHT) IMBALANCE)

- Note the levels



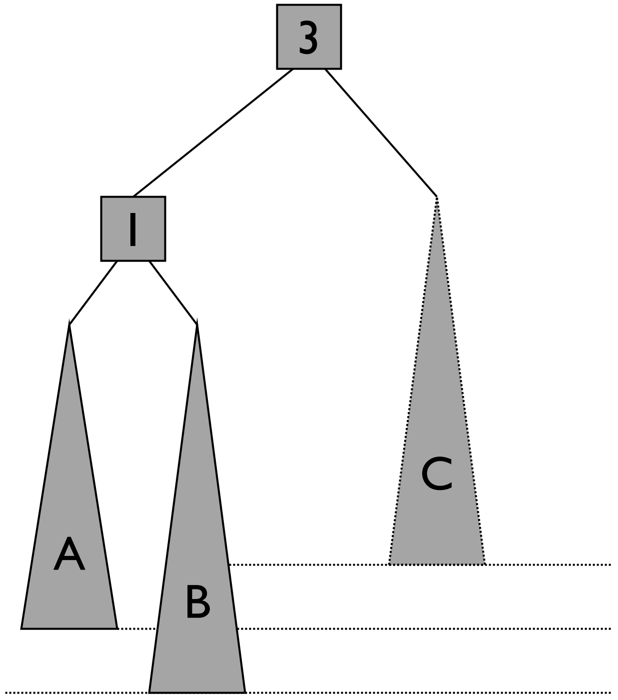
- B and C have the same height
- A is one level higher
- Therefore make 1 the new root, 2 its right child and B and C the subtrees of 2
- Result: a more balanced and legal AVL tree

SINGLE ROTATION



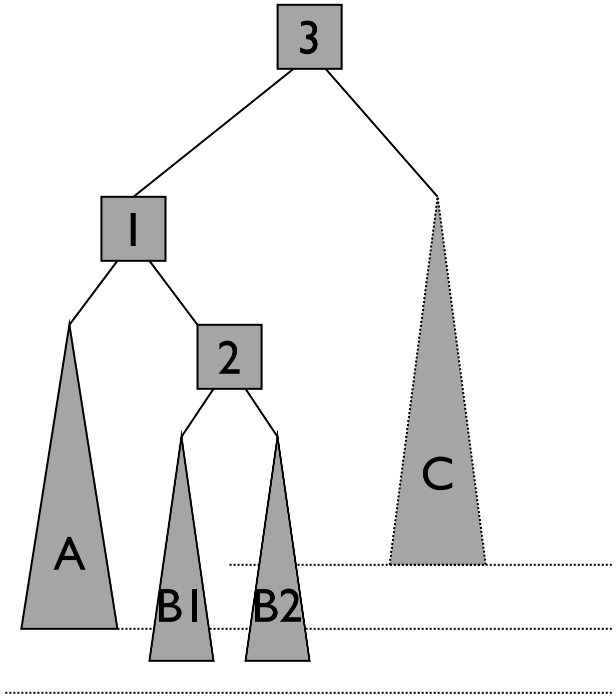
LEFT(RIGHT) IMBALANCE
(AND RIGHT(LEFT) IMBALANCE)

- Can't use the left-left balance trick - because now it's the middle subtree, i.e. B, that's too deep.
- Instead consider what's inside B...



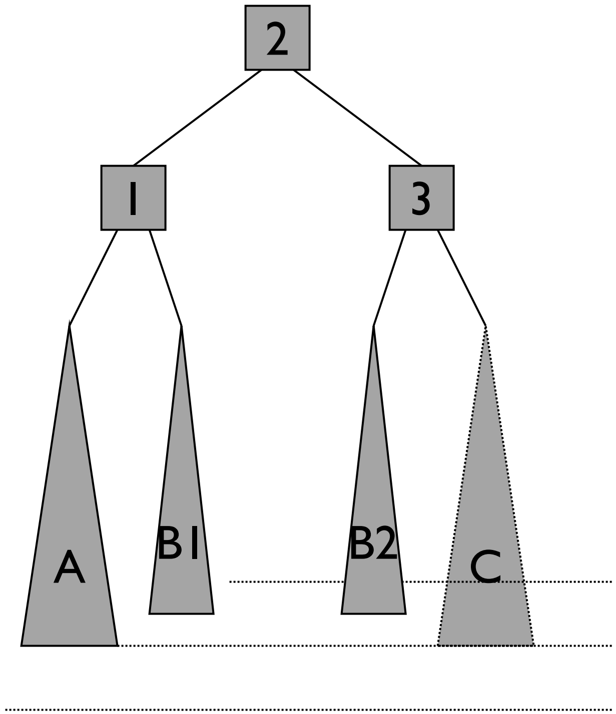
LEFT(RIGHT) IMBALANCE (AND RIGHT(LEFT) IMBALANCE)

- B will have two subtrees containing at least one item
- We do not know which is too deep - set them both to 0.5 levels below subtree A

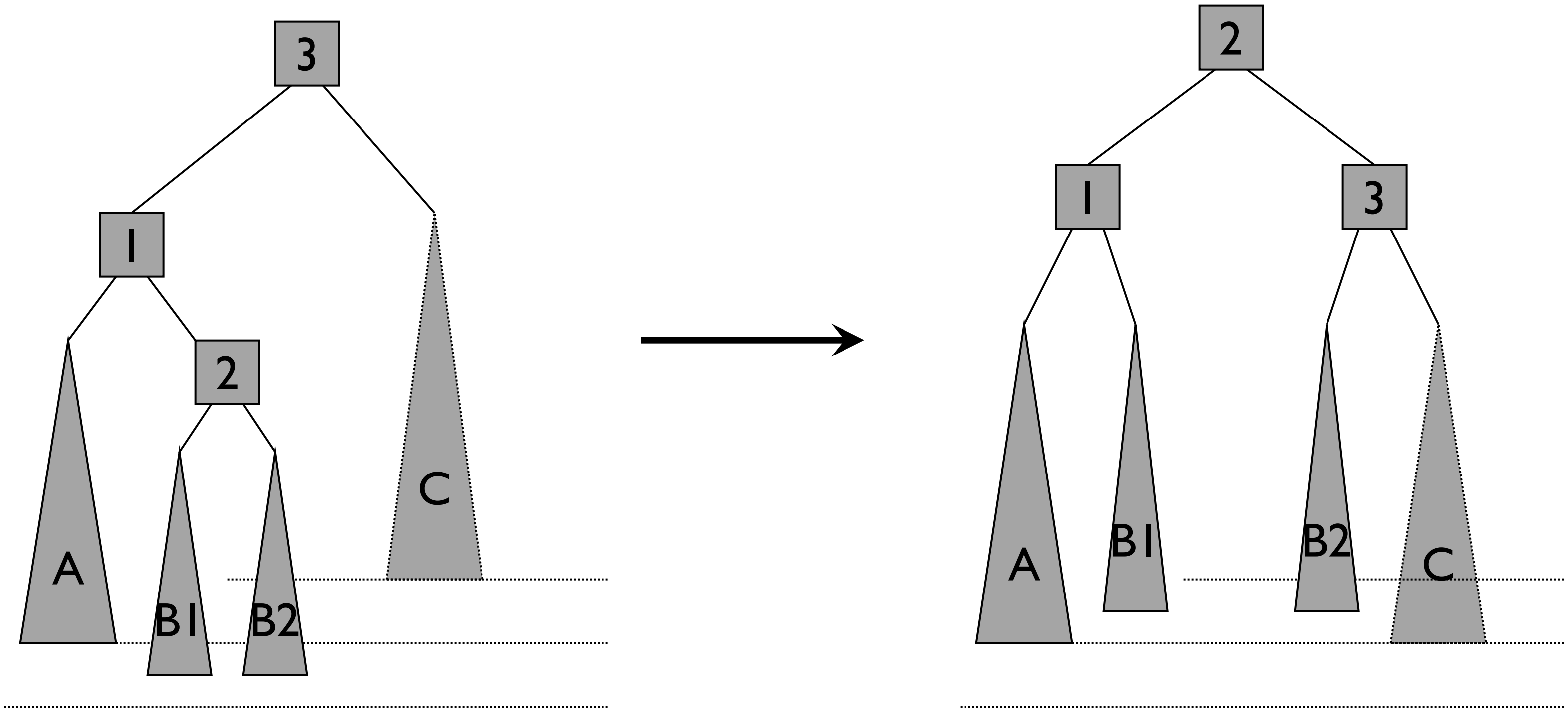


LEFT(RIGHT) IMBALANCE (AND RIGHT(LEFT) IMBALANCE)

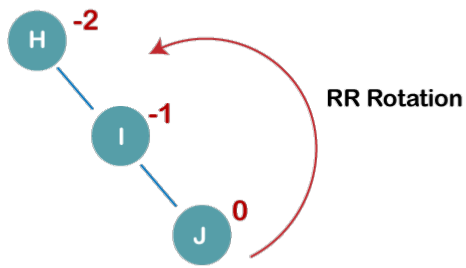
- Neither 1 nor 3 worked as root node so make 2 the root
- Rearrange the subtrees in the correct order
- No matter how deep B1 or B2 (+/- 0.5 levels) we get a legal AVL tree again

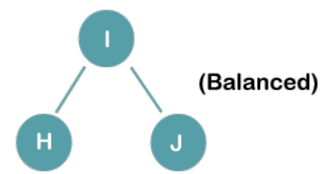


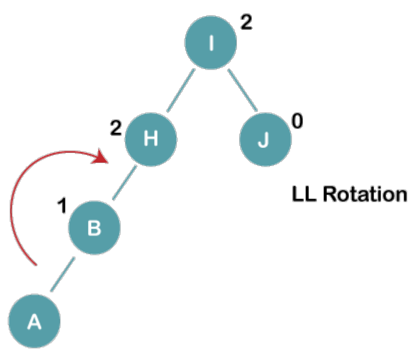
DOUBLE ROTATION

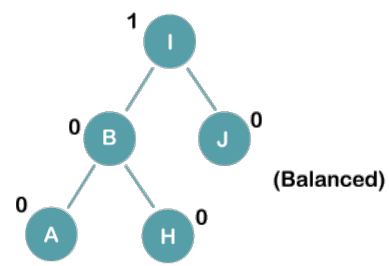


EXAMPLE: H, I, J, B, A, E, C, F, D, G, K, L

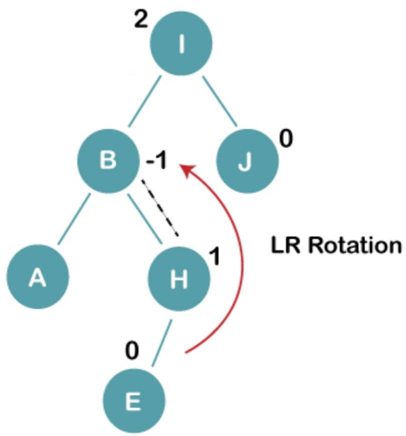


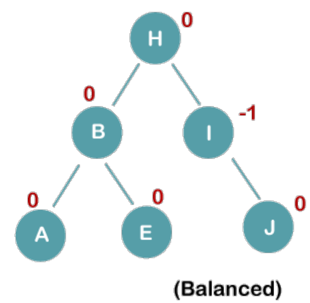


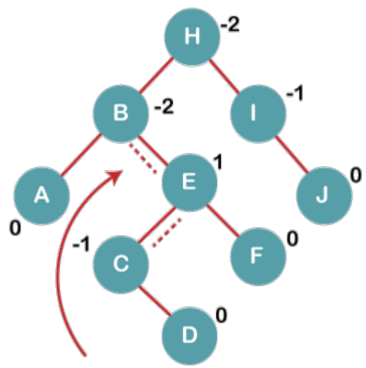


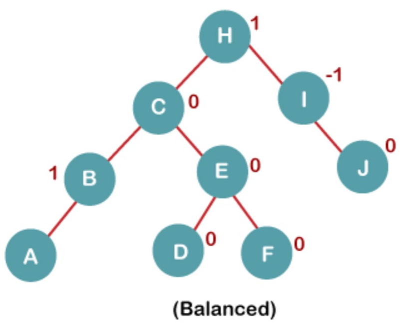


EXAMPLE: H, I, J, B, A, E, C, F, D, G, K, L

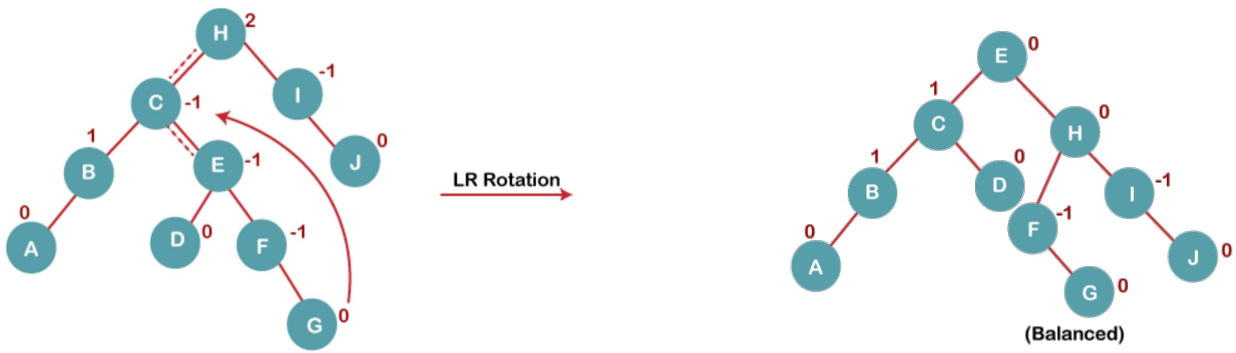




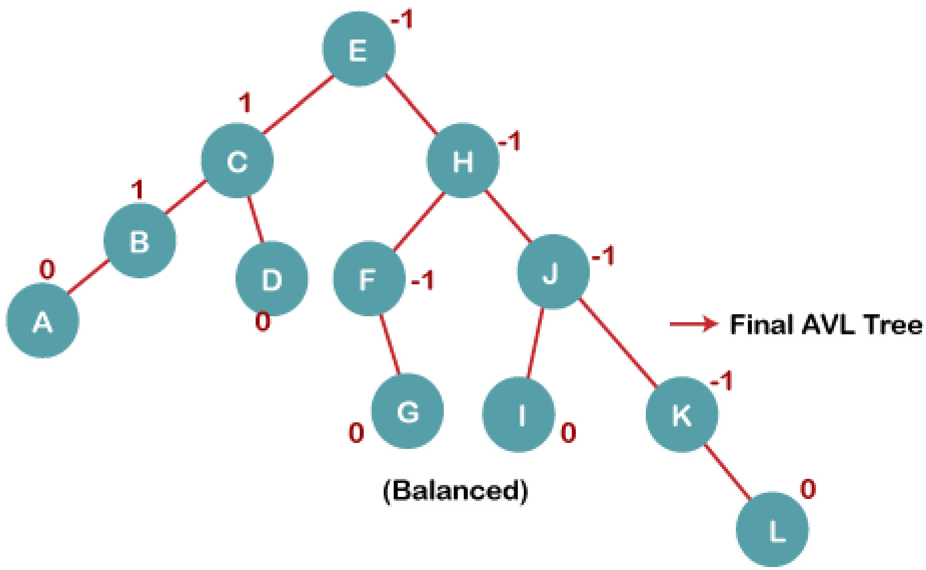




EXAMPLE: H, I, J, B, A, E, C, F, D, G, K, L



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

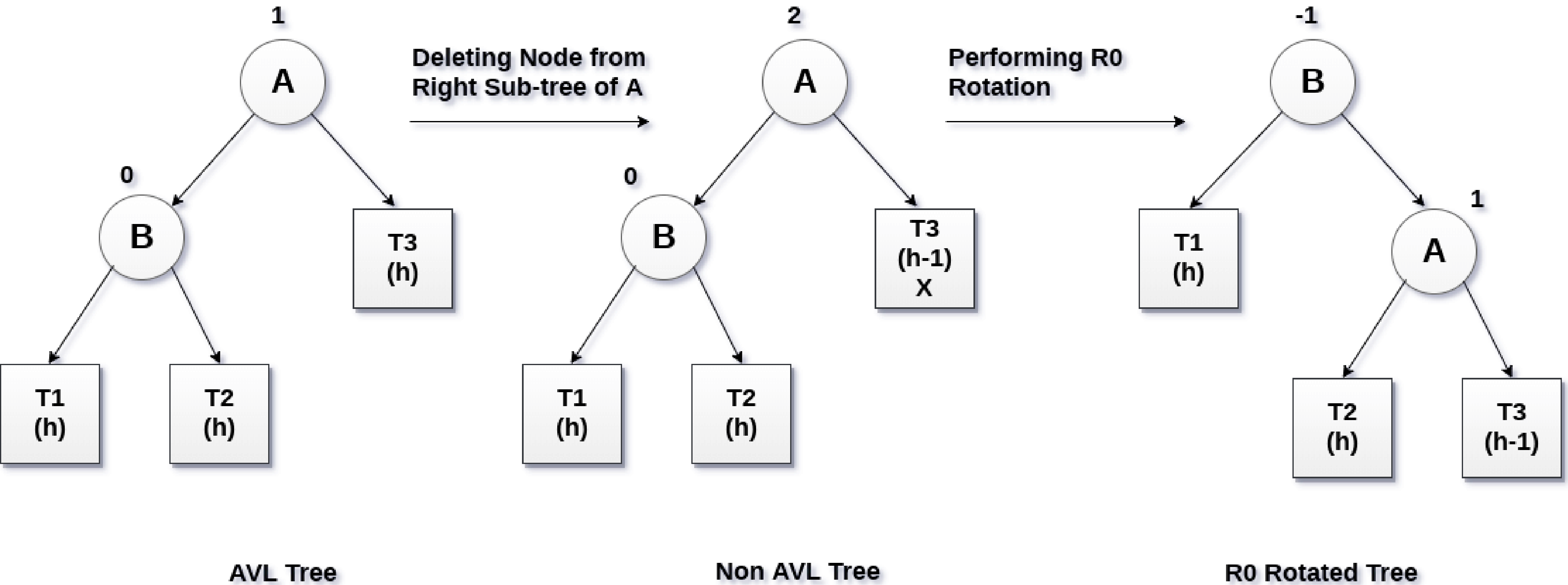


DELETION

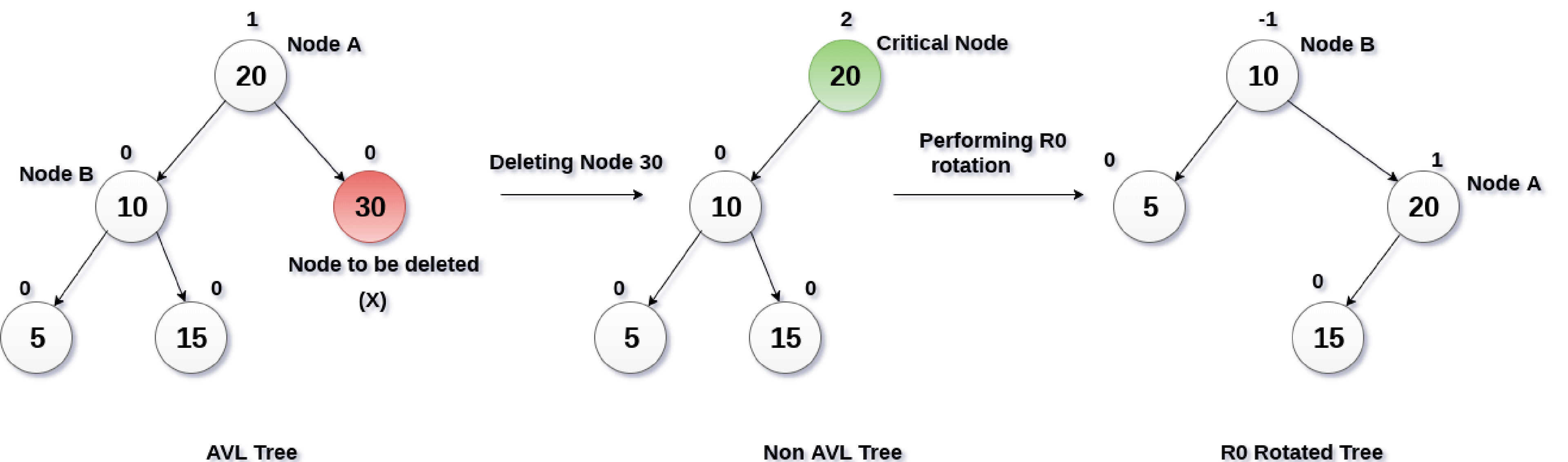
- Deletion may disturb the balance factor of an AVL tree
  - We need to perform rotations. The two types of rotations are L rotation and R rotation.
  - If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied.
  - If the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

- Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations.

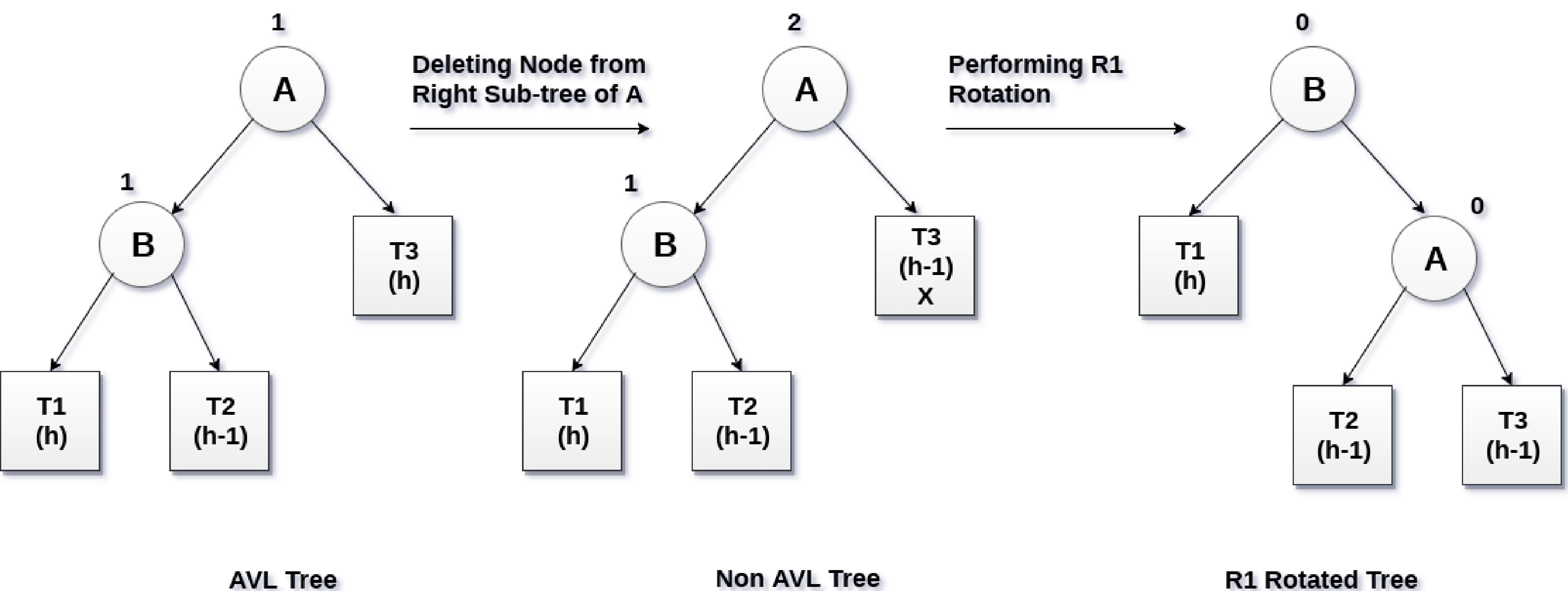
R0 ROTATION
(NODE B HAS BALANCE FACTOR 0)



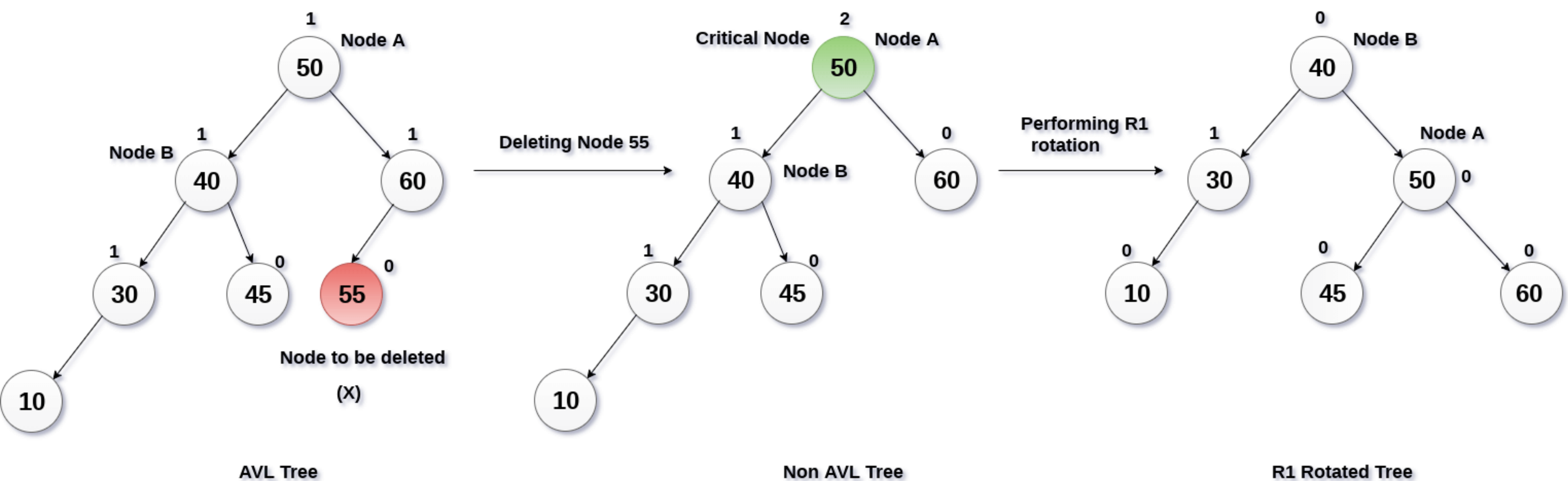
R0 ROTATION
(NODE B HAS BALANCE FACTOR 0)



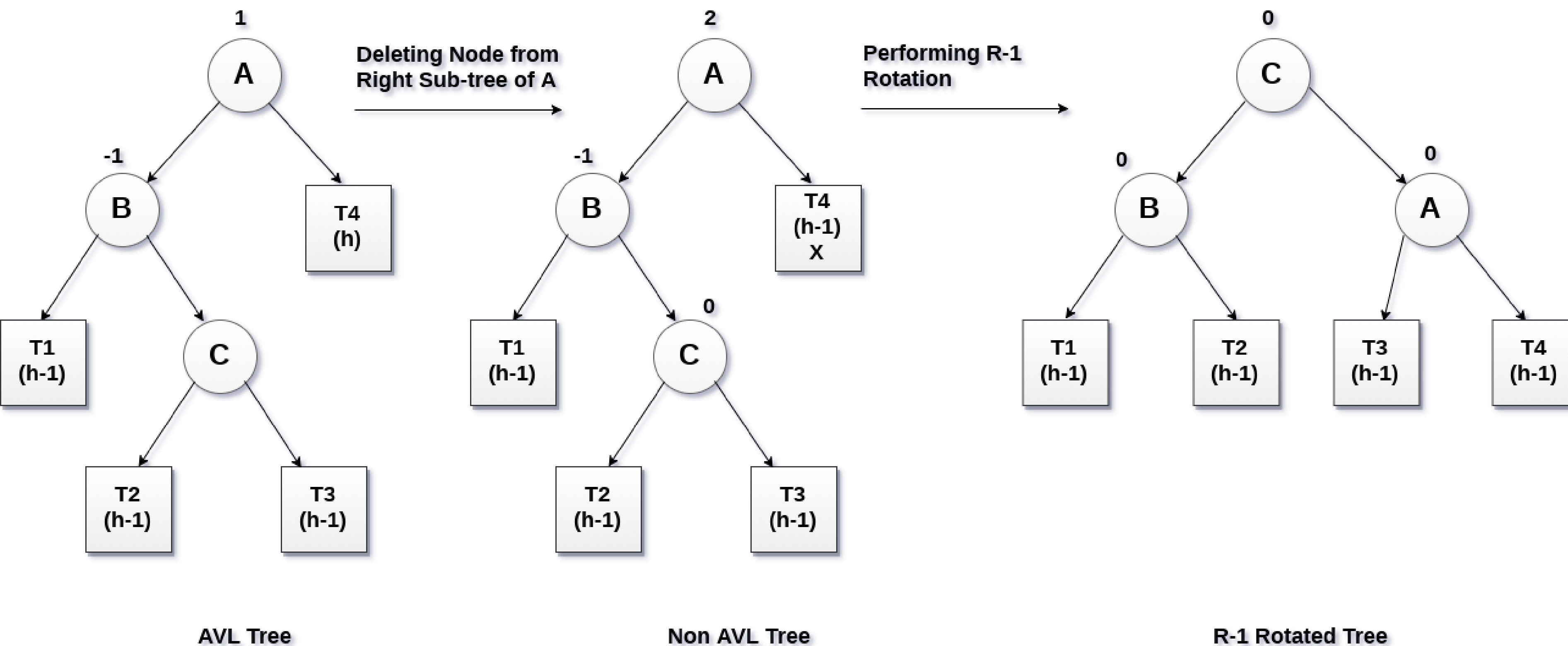
R1 ROTATION
(NODE B HAS BALANCE FACTOR 1)



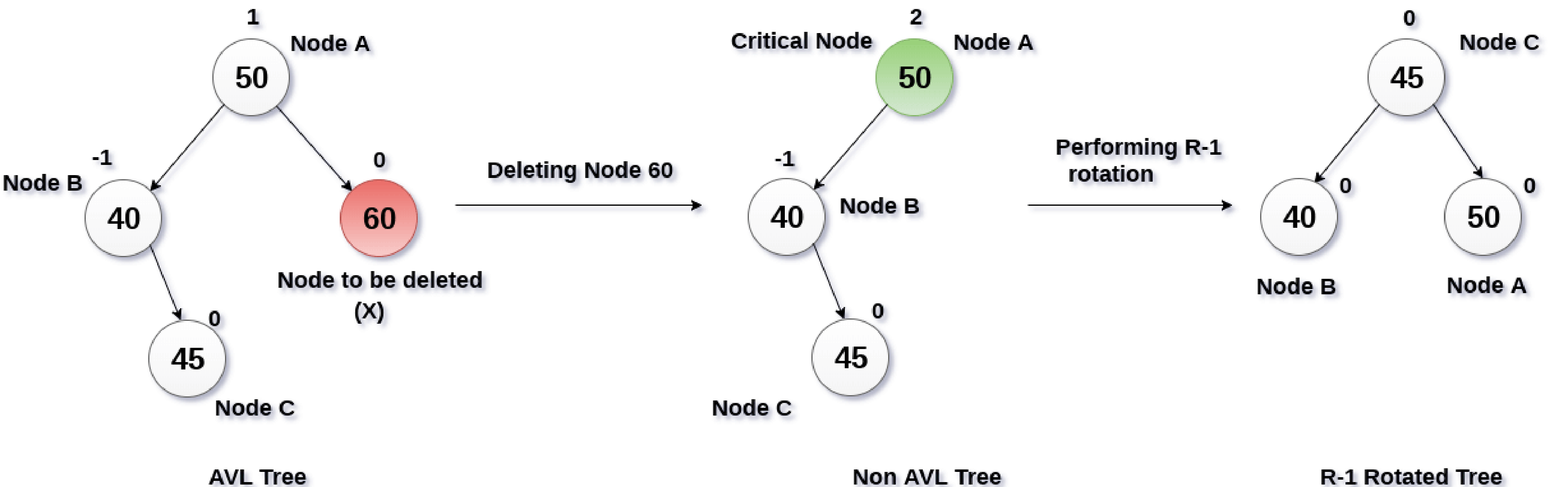
R1 ROTATION
(NODE B HAS BALANCE FACTOR 1)



R-1 ROTATION
(NODE B HAS BALANCE FACTOR -1)



R-1 ROTATION
(NODE B HAS BALANCE FACTOR -1)



THANKS