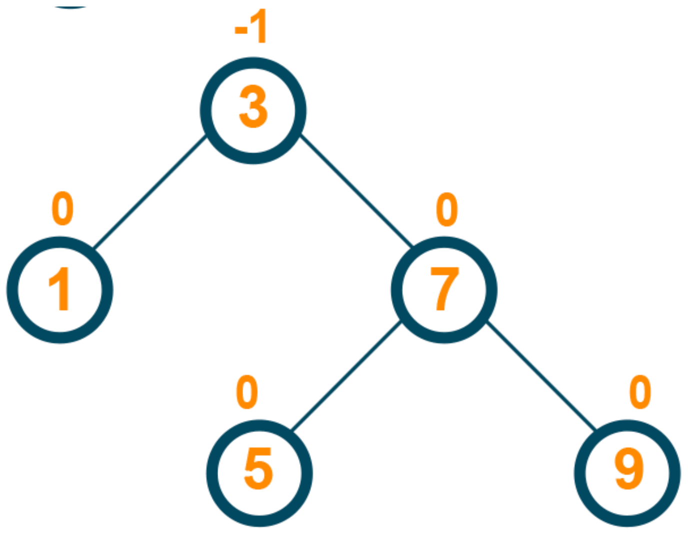
Data Structures and Algorithms

Lecture 30: AVL Trees - Introduction and Insertion

AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

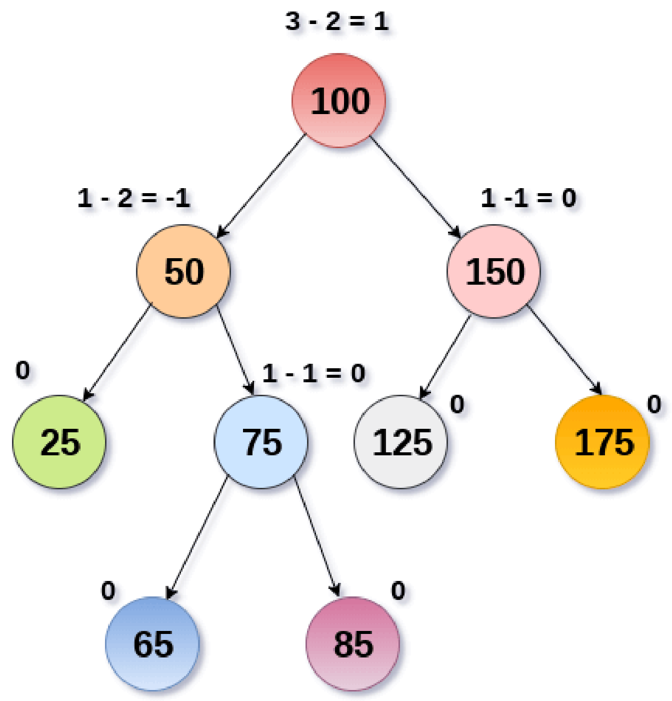


AVL Tree

- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- Balance Factor (k) = height (left(k)) - height (right(k))
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

AVL Tree

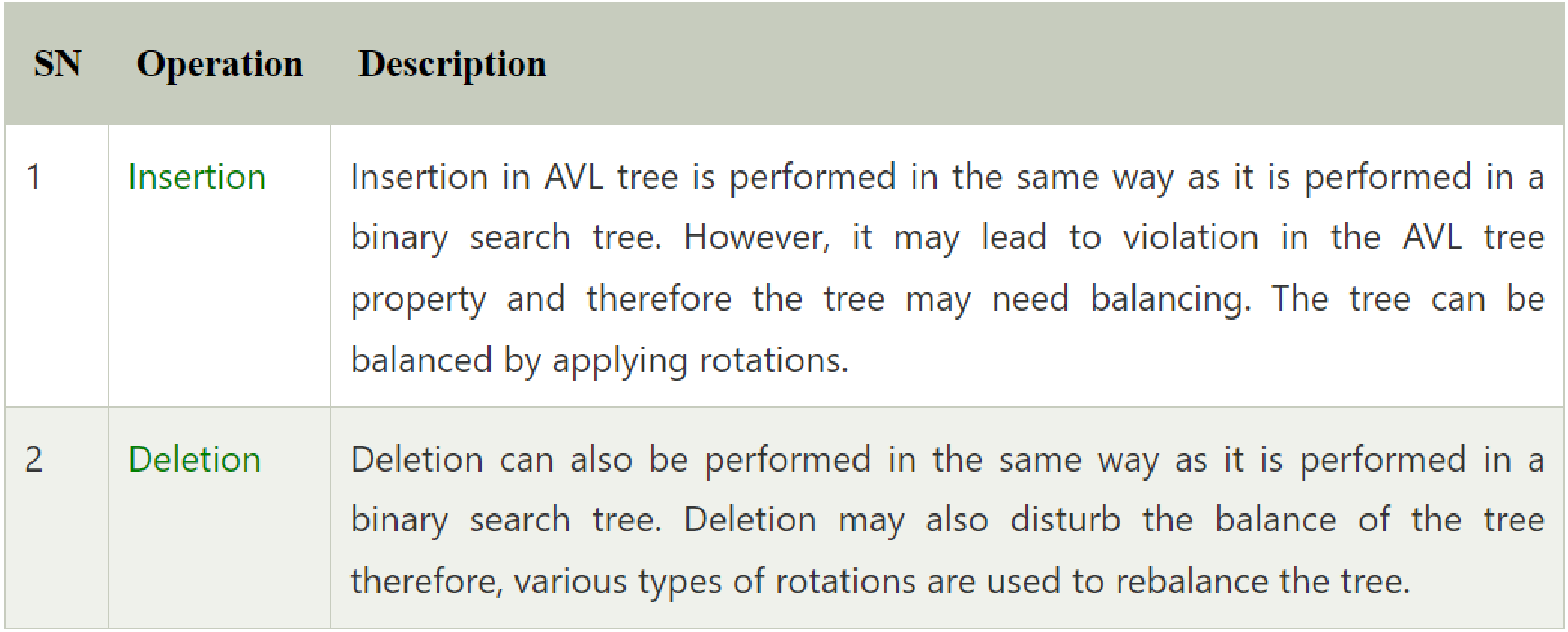
An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Operations on AVL tree

Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they 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.

Importance of AVL Tree

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree 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.

AVL Rotations

Rotation is the process of moving nodes either to left or to right to make the tree balanced. There are basically four types of rotations which are as follows:

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

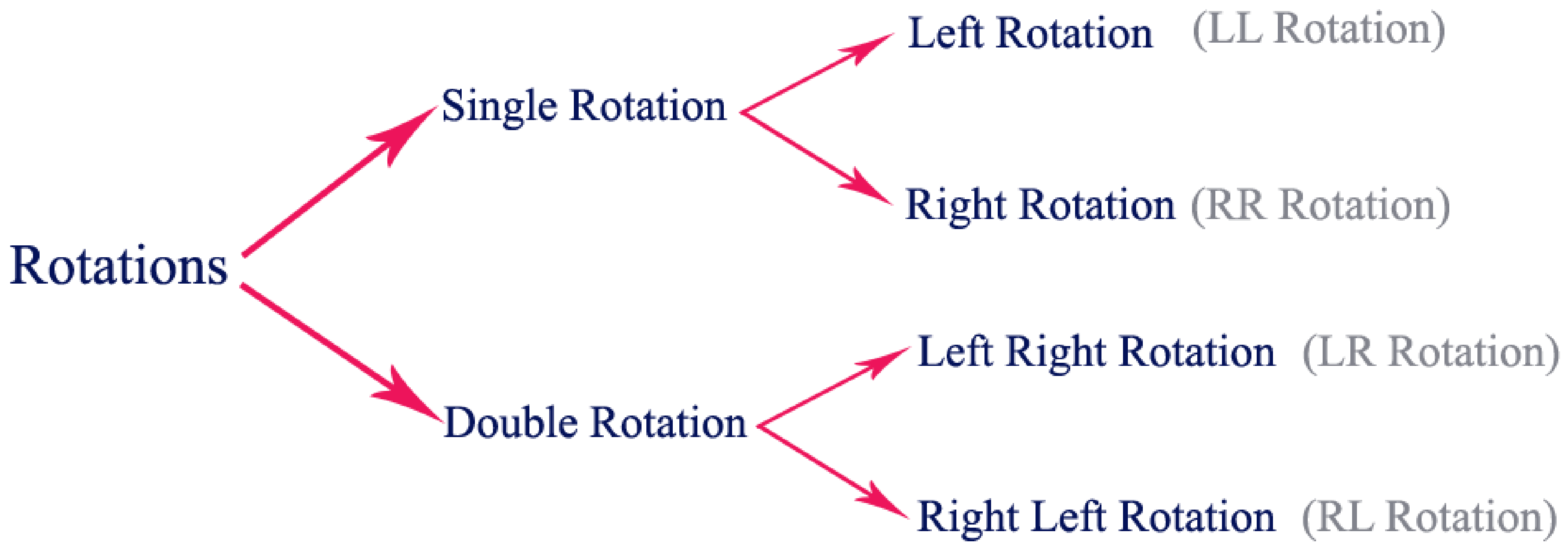
L R rotation : Inserted node is in the right subtree of left subtree of A

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.

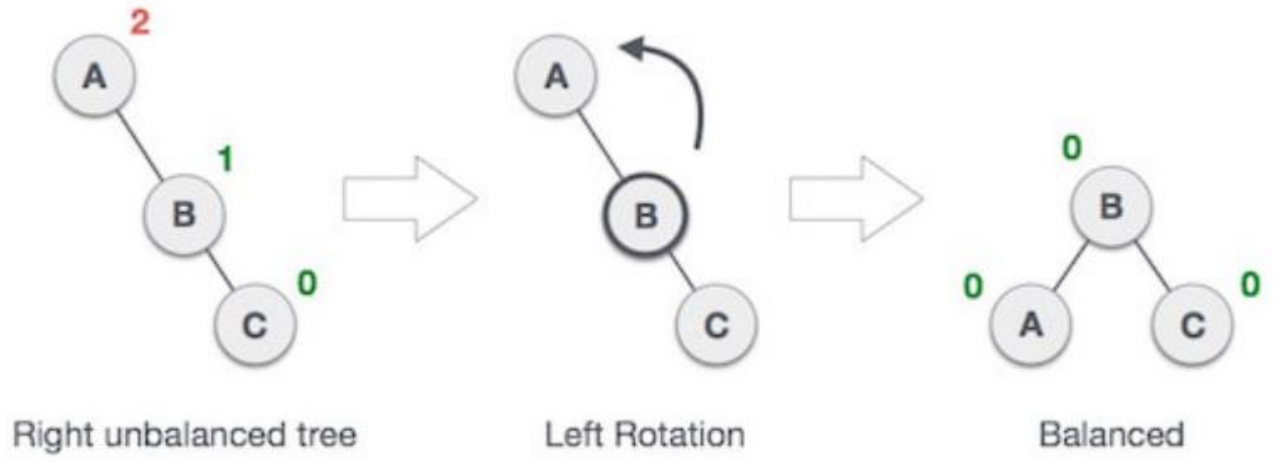
The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations.

AVL Rotations


Rotations
Single Rotation
Double Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

1. RR Rotation

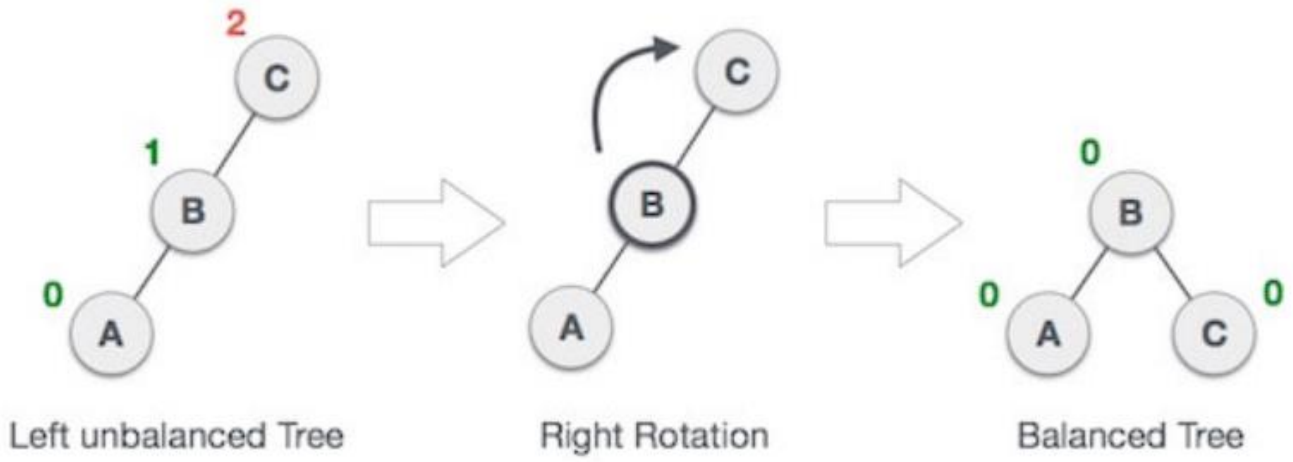
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

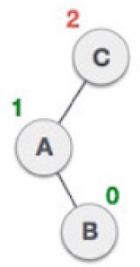
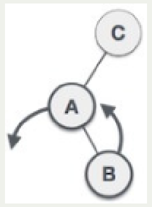
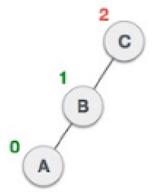
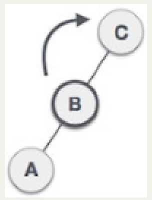
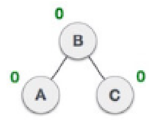


In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

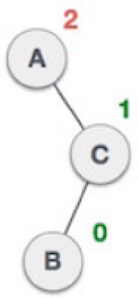
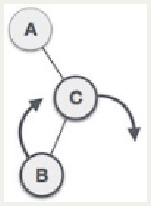
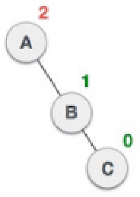
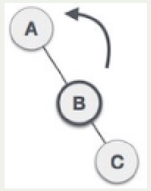
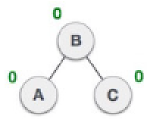
3. LR Rotation

	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

RL rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

4. RL Rotation

	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

AVL Tree - Insertion

In AVL Tree, a new node is always inserted as a leaf node.

Step 1- Insert the new element into the tree using Binary Search Tree insertion logic.

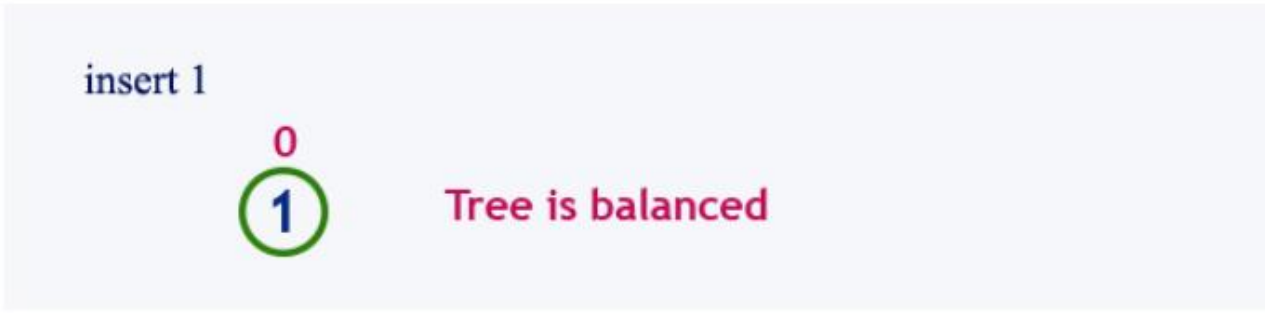
Step 2 - After insertion, check the Balance Factor of every node.

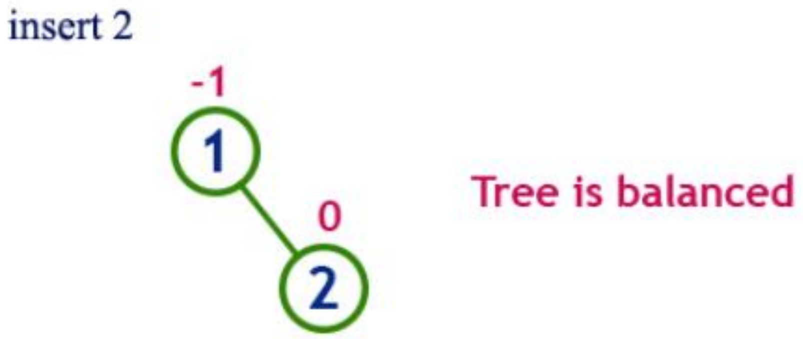
Step 3 - If the Balance Factor of every node is 0 or 1or -1then go for next operation.

Step 4 - If the Balance Factor of any node is other than 0 or 1or -1then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

AVL Tree – Insertion Example

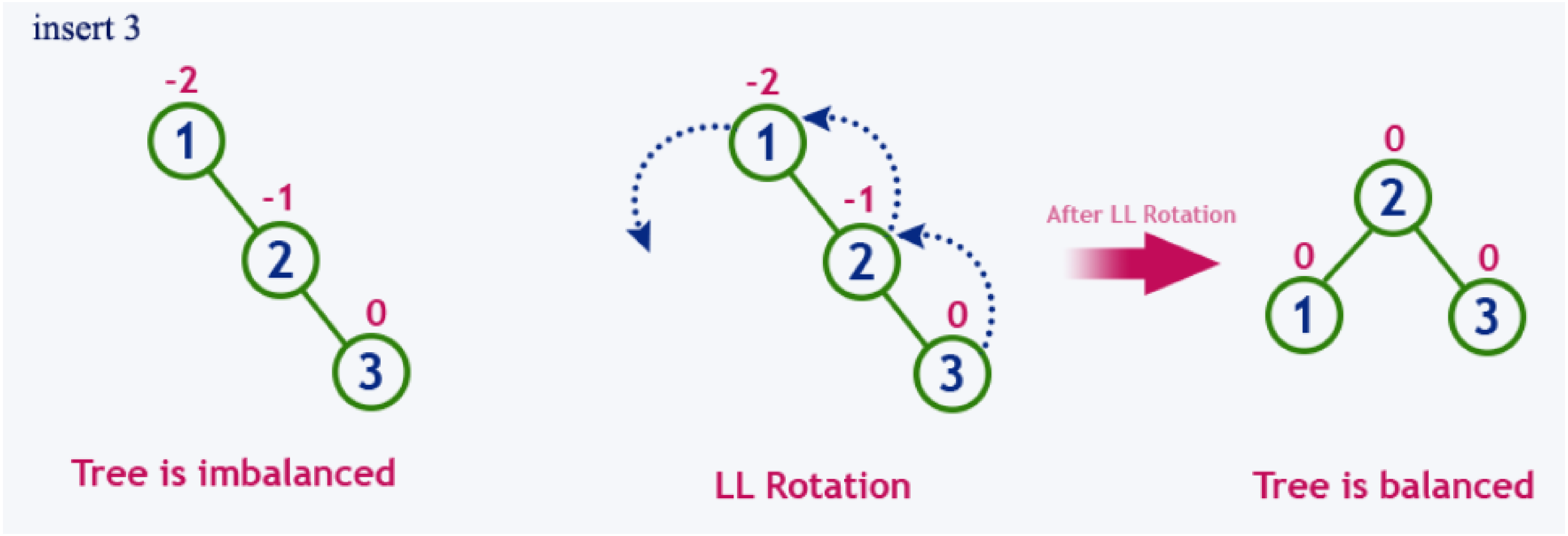
Let us Construct AVL Tree by inserting number from 1 to 8





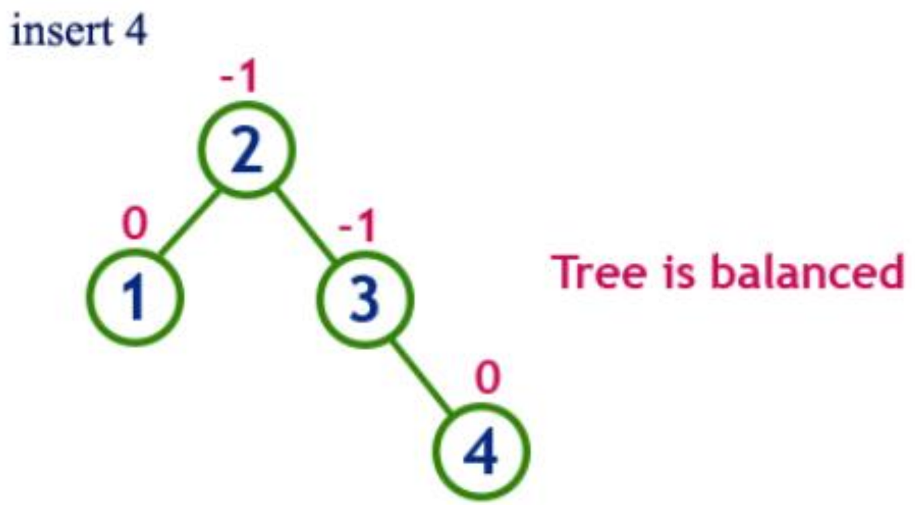
AVL Tree – Insertion Example

Let us Construct AVL Tree by inserting number from 1 to 8



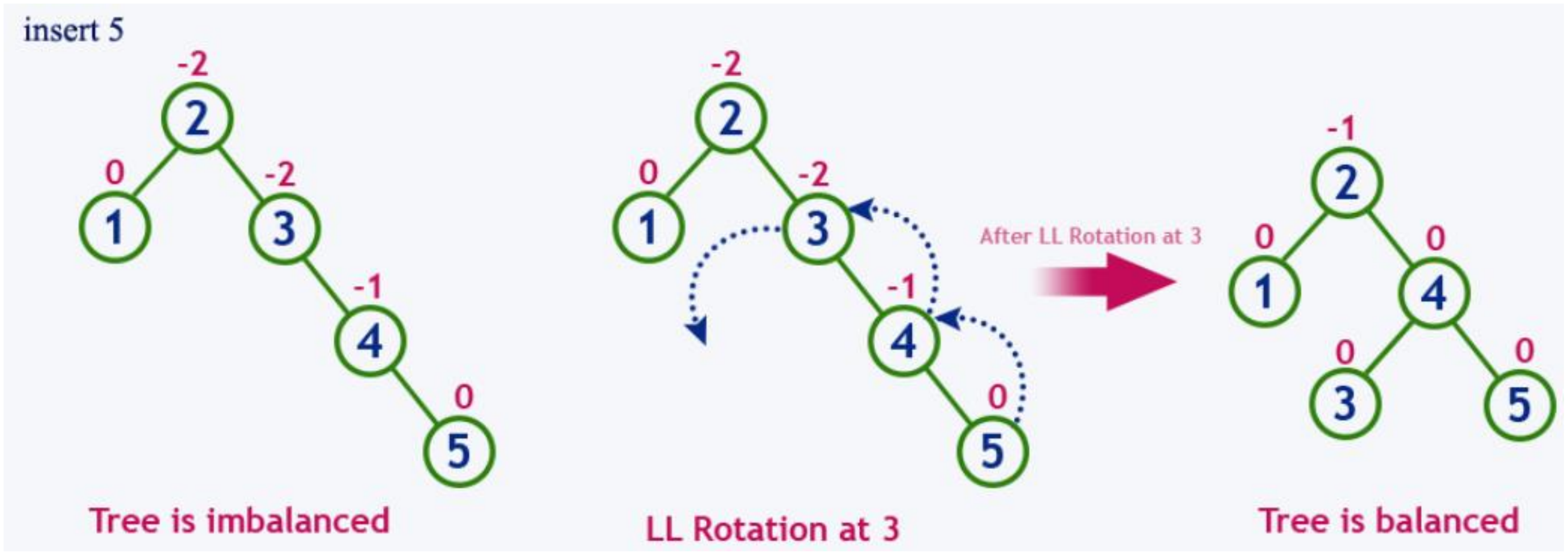
AVL Tree – Insertion Example

Let us Construct AVL Tree by inserting number from 1 to 8



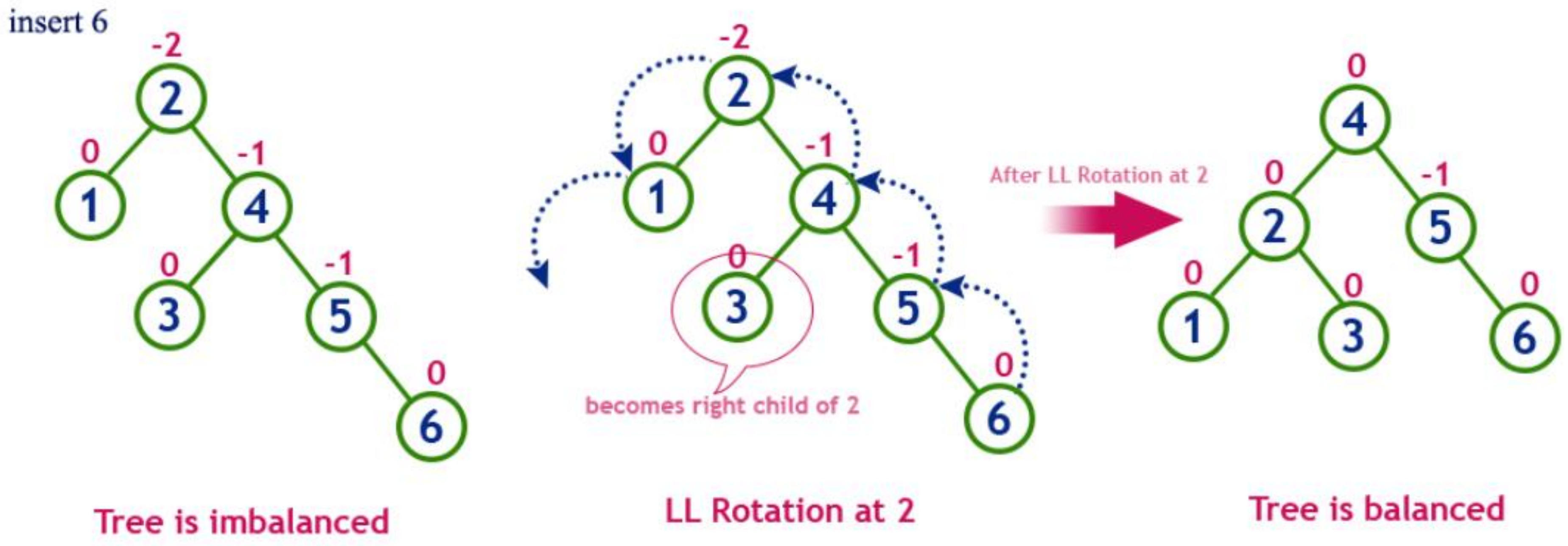
AVL Tree – Insertion Example

Let us Construct AVL Tree by inserting number from 1 to 8



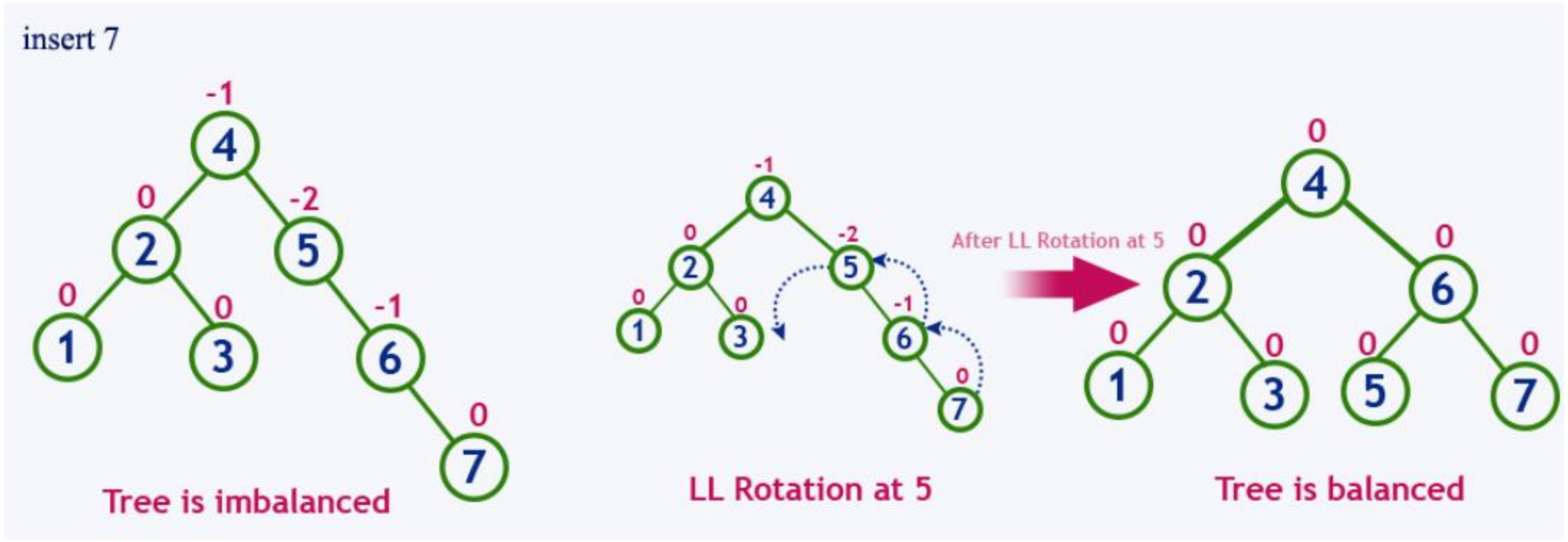
AVL Tree – Insertion Example

Let us Construct AVL Tree by inserting number from 1 to 8



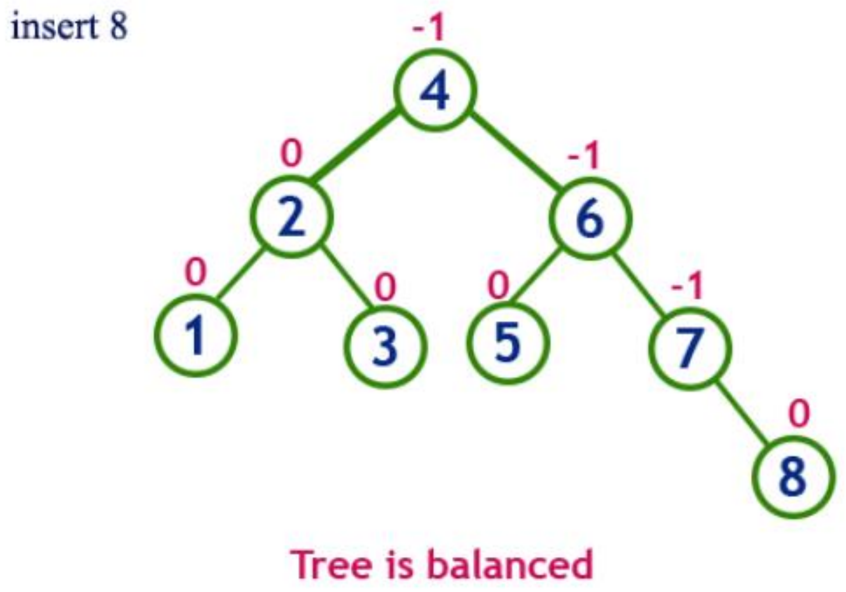
AVL Tree – Insertion Example

Let us Construct AVL Tree by inserting number from 1 to 8



AVL Tree – Insertion Example

Let us Construct AVL Tree by inserting number from 1 to 8


insert 8
-1
4
0
2
-1
6
0
1
0
3
0
5
-1
7
0
8
Tree is balanced