CS-2001
DATA STRUCTURE

Dr. Hashim Yasin

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

AVL TREE

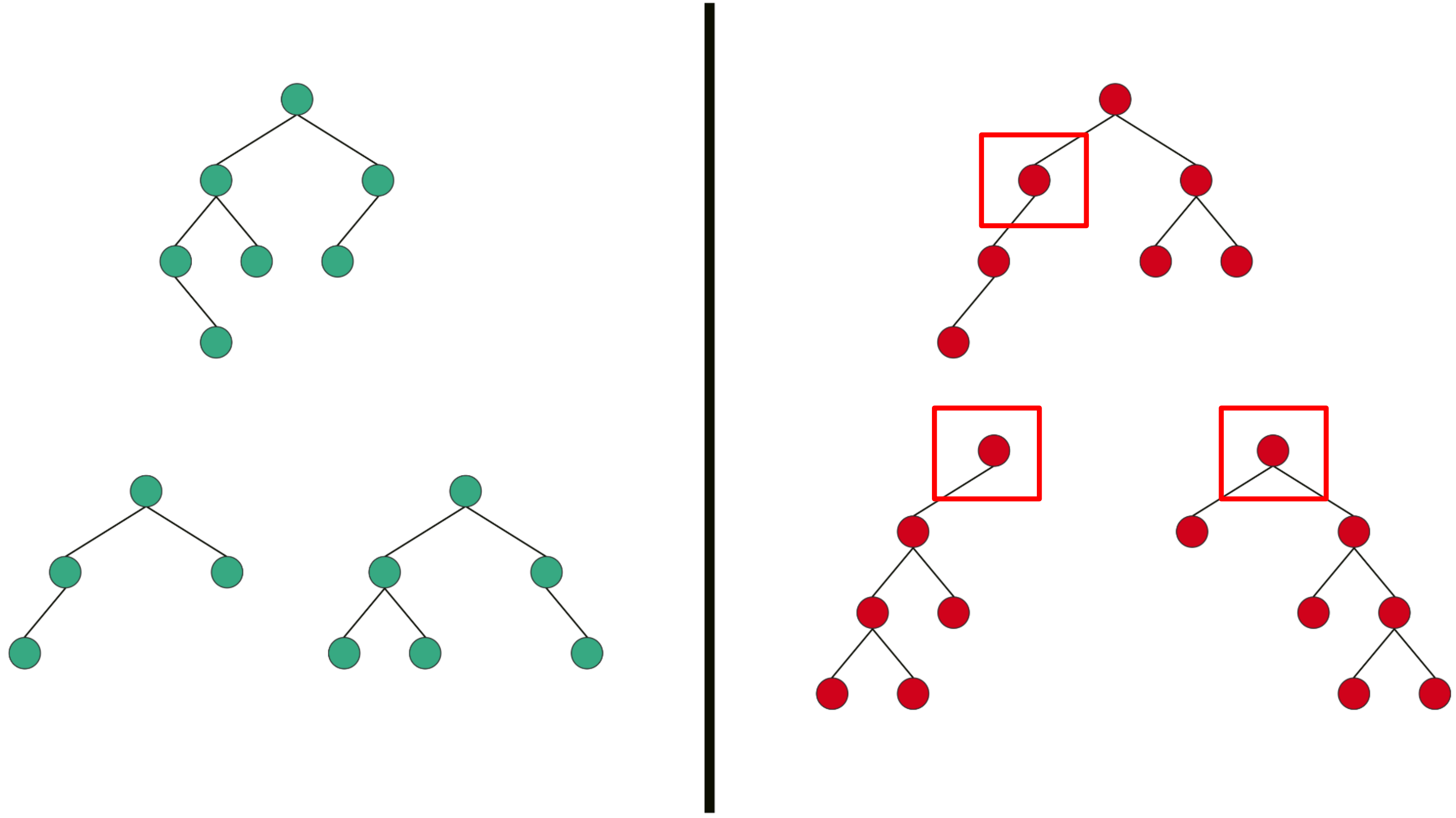
Balanced Binary Trees

Balanced Binary Tree

- is a Binary tree in which height of the left and the right sub-trees of every node may differ by at most 1.

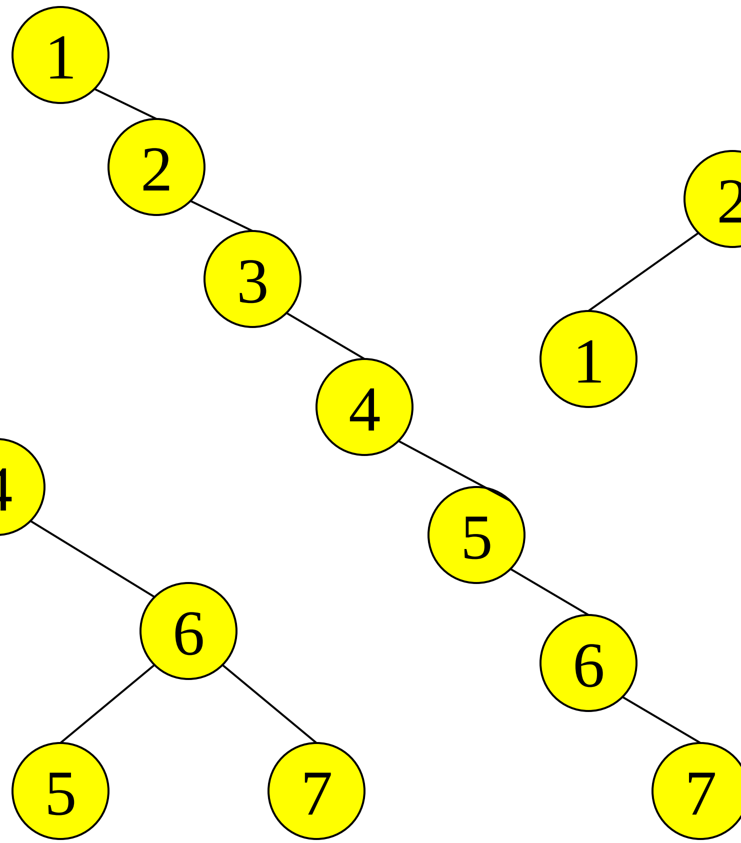
  - For every node, heights of left and right subtree can differ by no more than 1

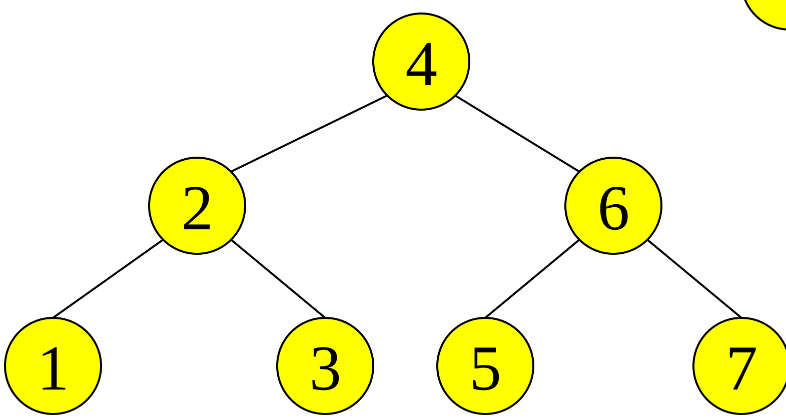
Balanced Binary Trees

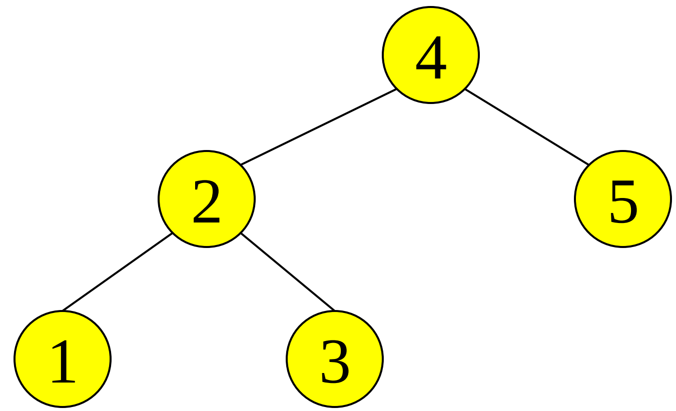


Valid and Invalid Structure of Balanced Binary Tree

Balanced and Unbalanced BST



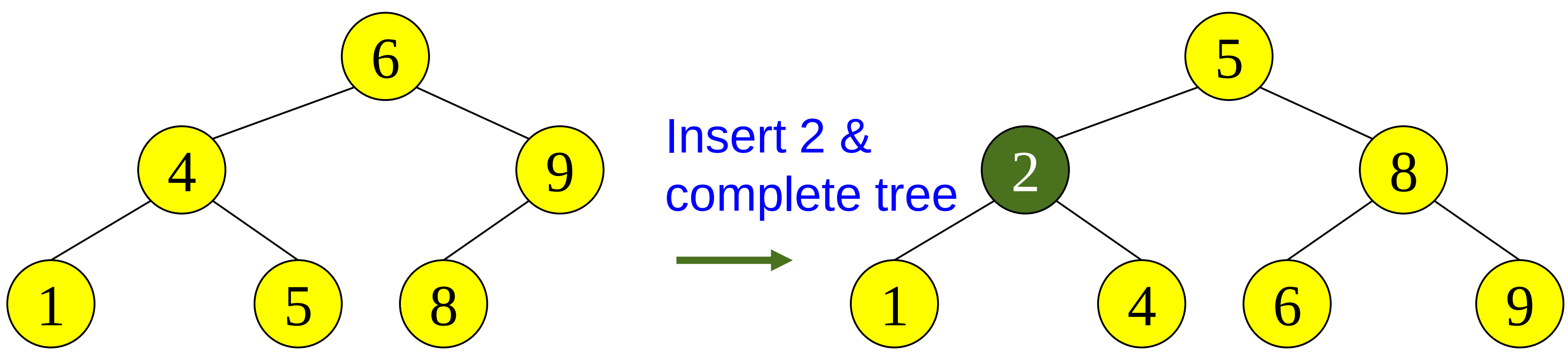




Is this “balanced”?

Perfect Balance

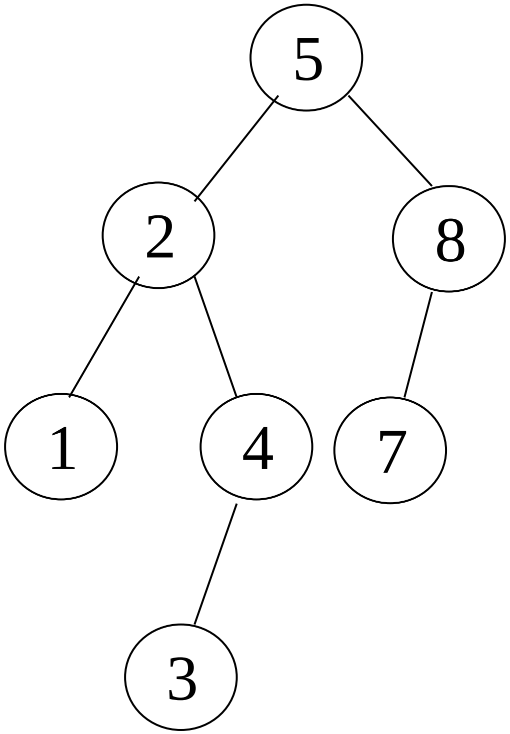
- We want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



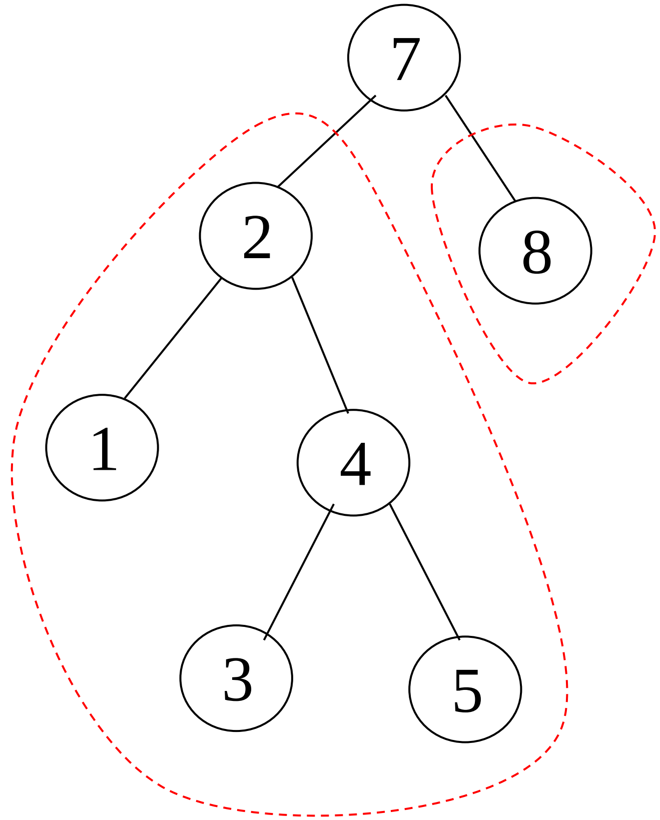
AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees.
- An AVL tree has **balance factor** calculated at every node.
  - For every node, heights of left and right subtree can differ by no more than 1

AVL Trees



An AVL Tree



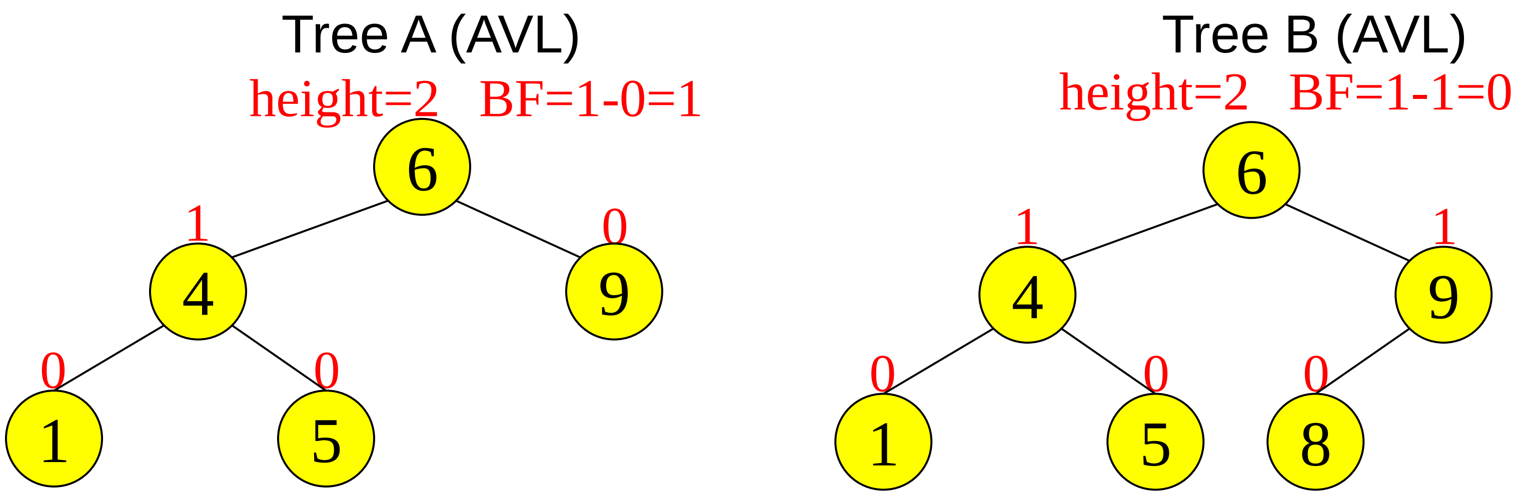
Not an AVL Tree

AVL Trees

Balancing Factor:

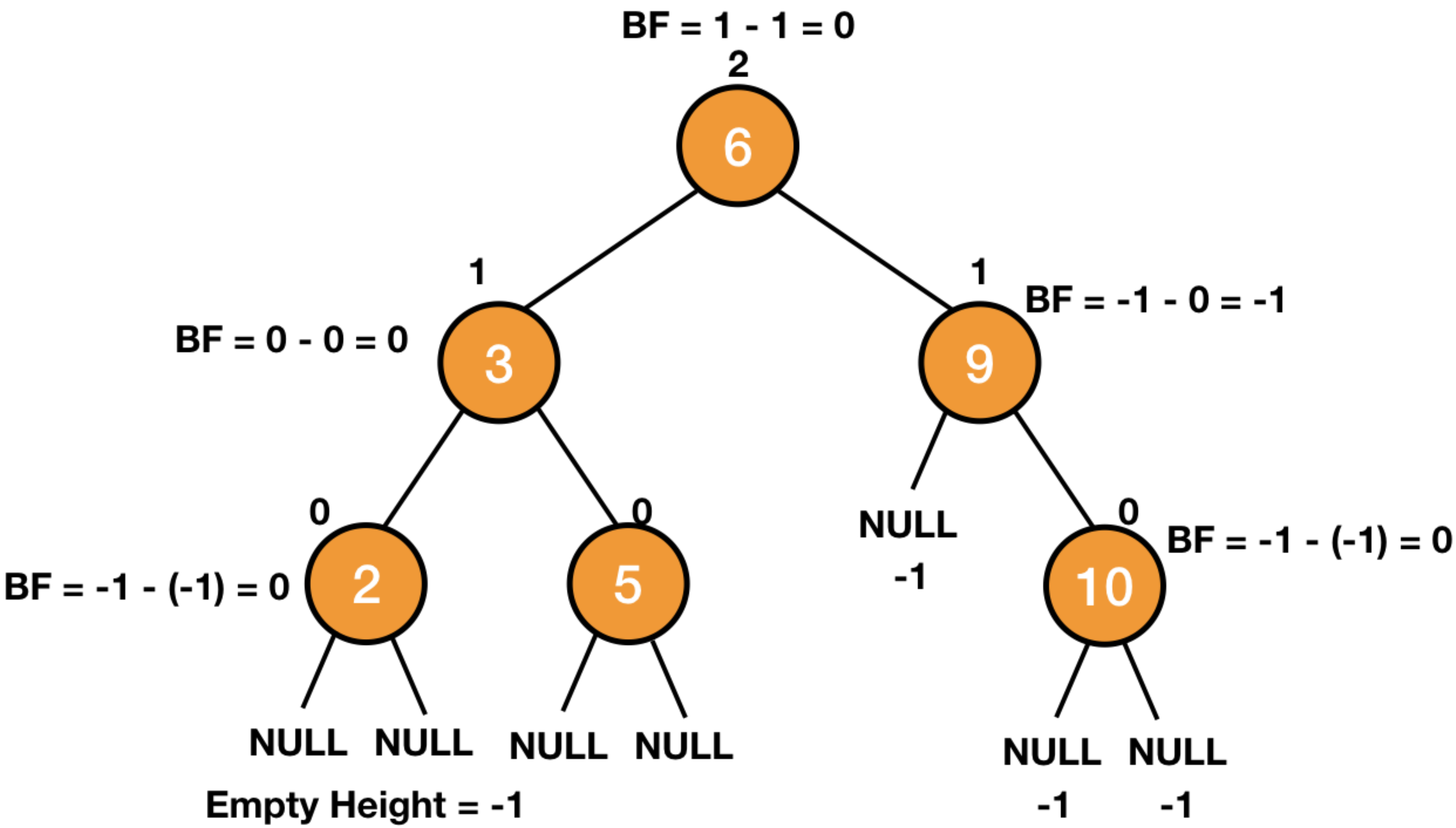
- The height of the left subtree minus the height of the right subtree of a node is called the *balance of the node(Balancing Factor)*.
  - For an AVL tree, the Balance Factors (BF) of the nodes are always -1, 0 or 1.
  - BF= height(left sub-tree) – height(right sub-tree)
- The height of an empty tree is defined to be 0.

Node Heights



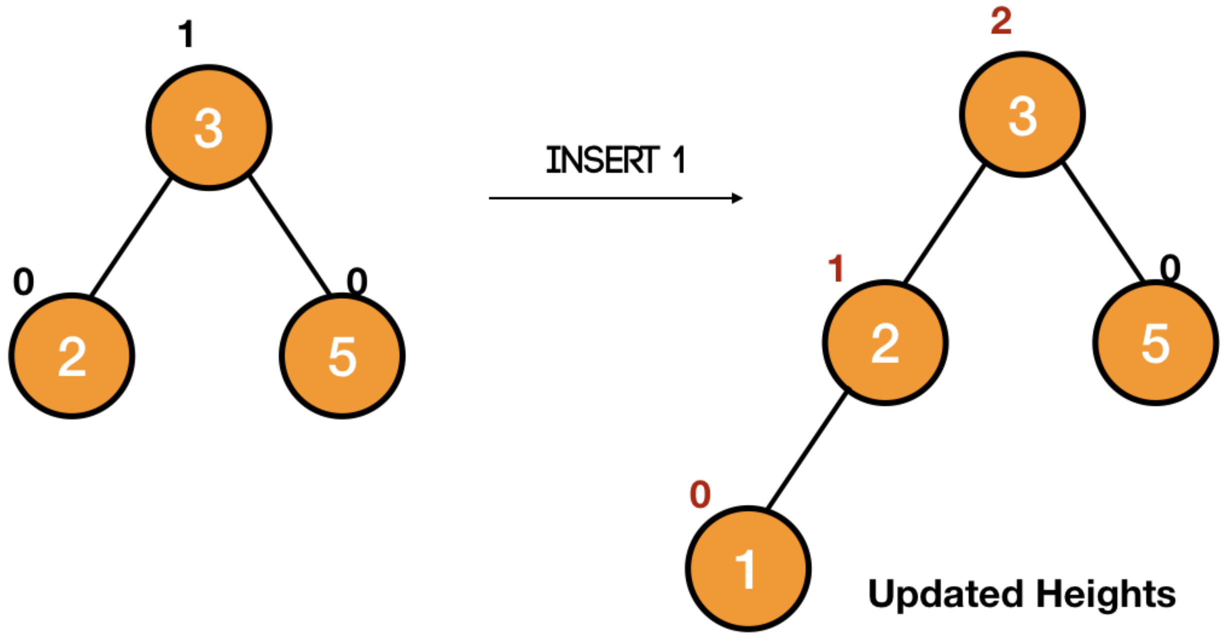
height of node = h
balance factor = h_{left}-h_{right}
empty height = 0

AVL Trees


BF = 1 - 1 = 0
2
6
1
1
BF = -1 - 0 = -1
BF = 0 - 0 = 0
3
9
0
0
0
NULL
-1
BF = -1 - (-1) = 0
BF = -1 - (-1) = 0
2
5
10
NULL NULL NULL NULL
NULL NULL
Empty Height = -1
-1 -1

AVL Tress

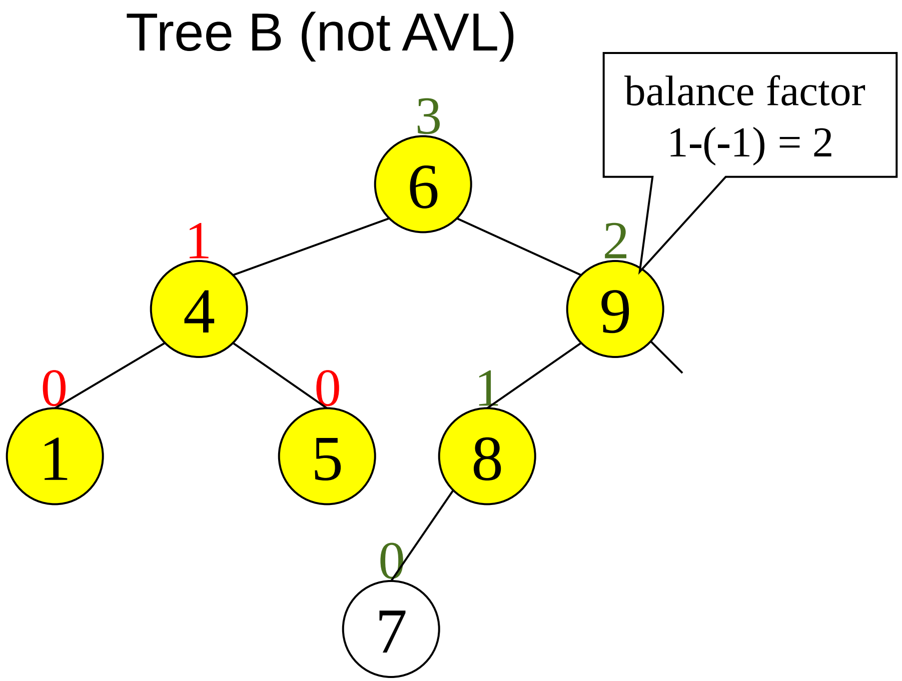
- Given an AVL tree, if insertions or deletions are performed, the AVL tree *may not* remain height balanced.



AVL Tress

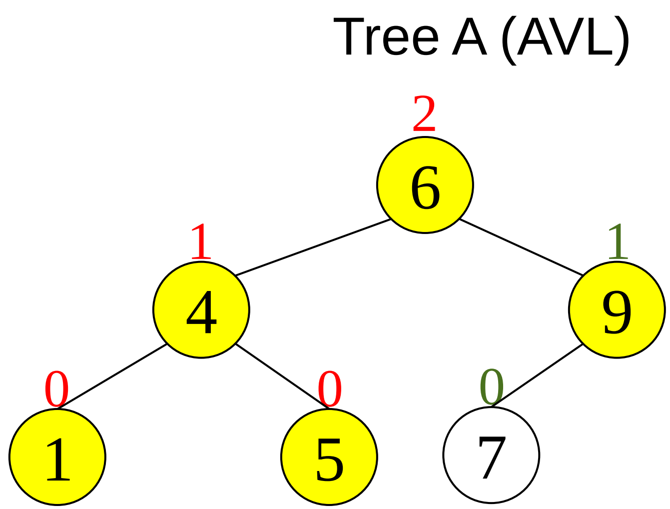
- Given an AVL tree, if insertions or deletions are performed, the AVL tree *may not* remain height balanced.

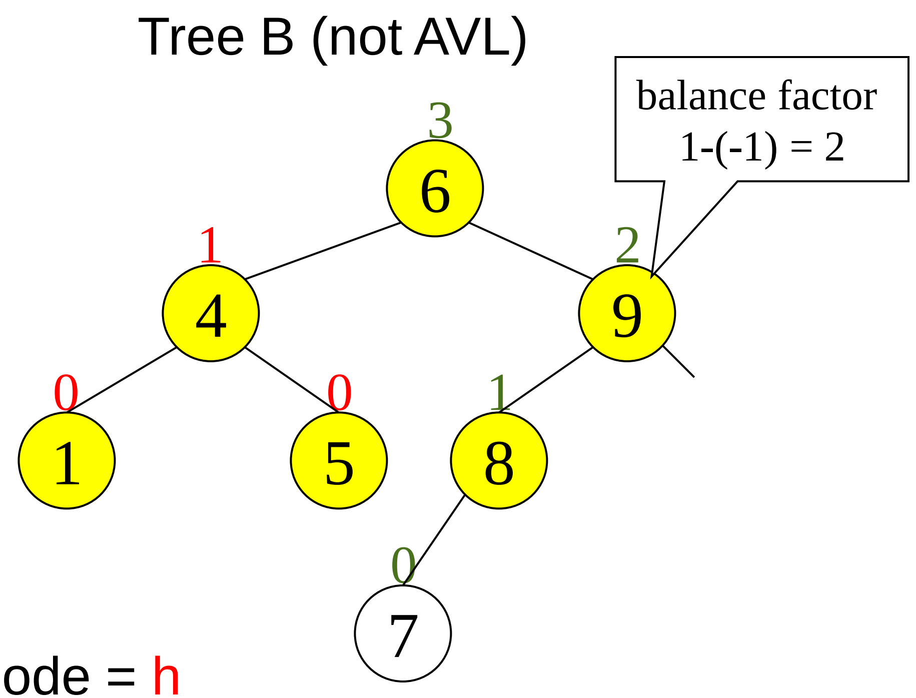
For Example: After Insertion 7, *the AVL tree becomes height unbalanced.*



Node Heights

Node Heights after Insert 7:



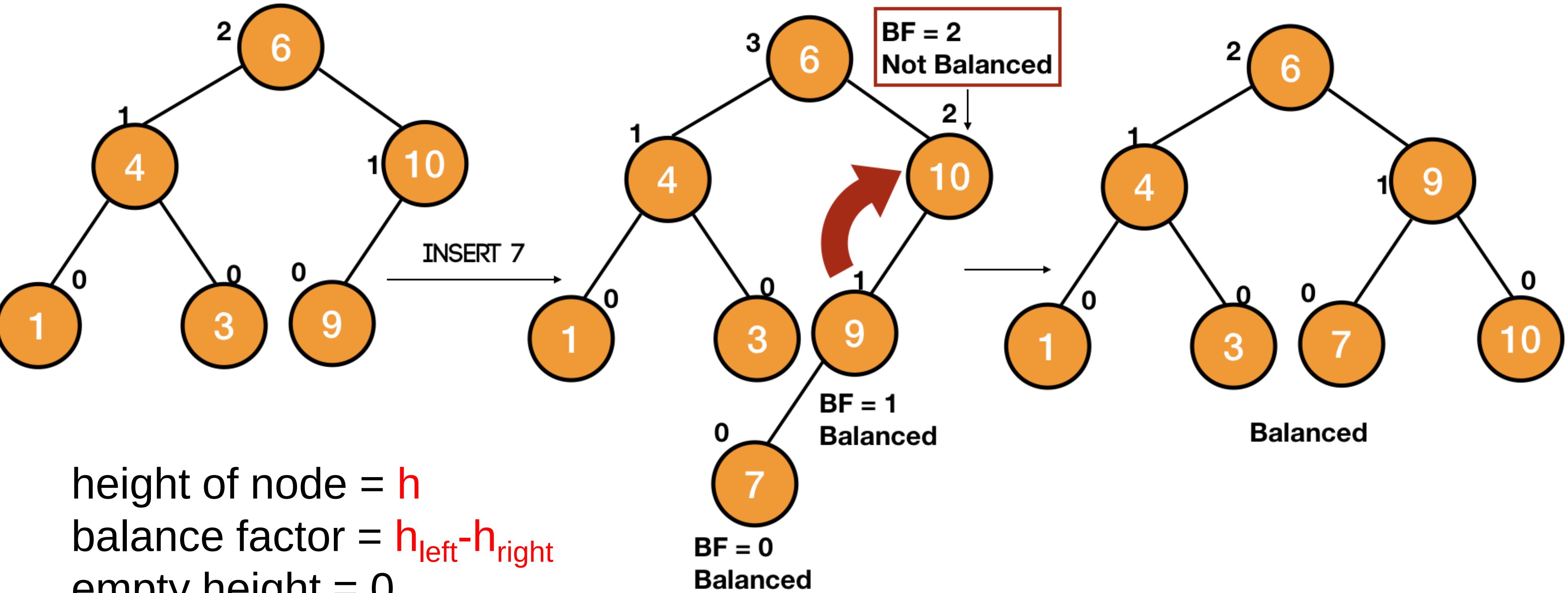


height of node = h

balance factor = h_{left}-h_{right}

empty height = 0

Node Heights



height of node = h
balance factor = h_{left}-h_{right}
empty height = 0

AVL Trees

To maintain the height balanced property of the AVL tree after insertion or deletion, it is necessary to perform a *transformation* on the tree so that,

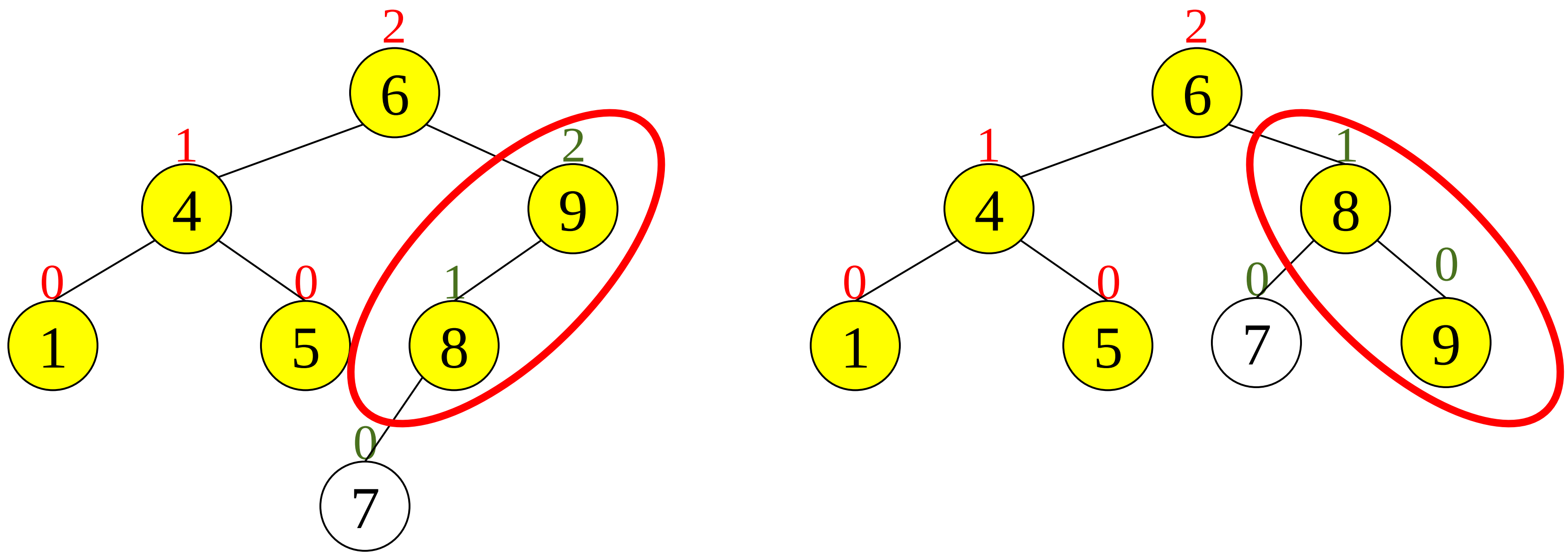
(1) the *in-order traversal of the transformed tree is the same as for the original tree* (i.e., the new tree remains a binary search tree).

(2) the tree after transformation is height-balanced.

Insertion in AVL Trees

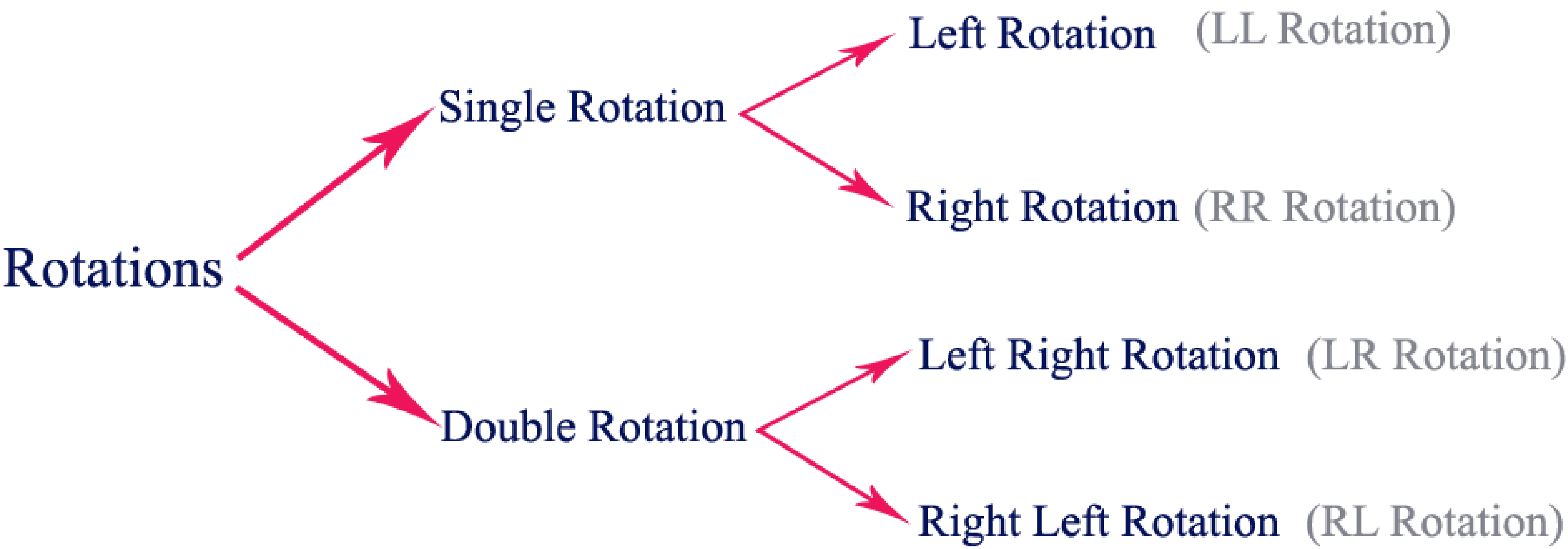
- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - Follow the path up to the root, find the first node (i.e., deepest) whose new balance violates the AVL condition. Call this node *a*
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Insertion in AVL Trees



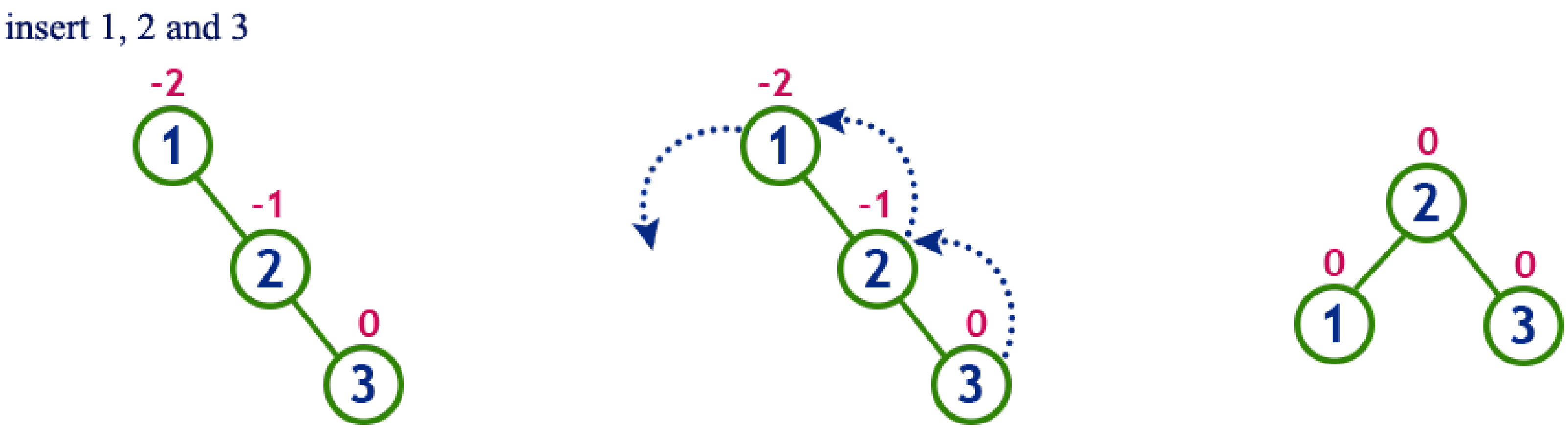
AVL TREE ROTATION

AVL Tree … Rotations



LL Rotation

- In LL Rotation, every node moves *one position to **<u>left</u>** from the current position*.



Tree is imbalanced

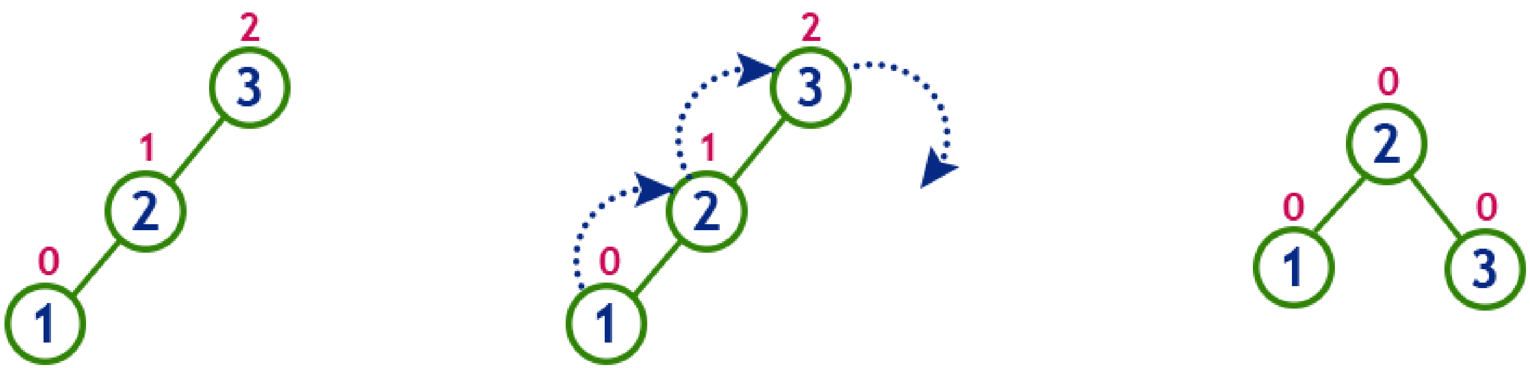
To make balanced we use LL Rotation which moves nodes one position to left

After LL Rotation Tree is Balanced

RR Rotation

- In RR Rotation, every node moves *one position to* ***<u>right</u>*** *from the current position*.

insert 3, 2 and 1



Tree is imbalanced

because node 3 has balance factor 2

To make balanced we use RR Rotation which moves nodes one position to right

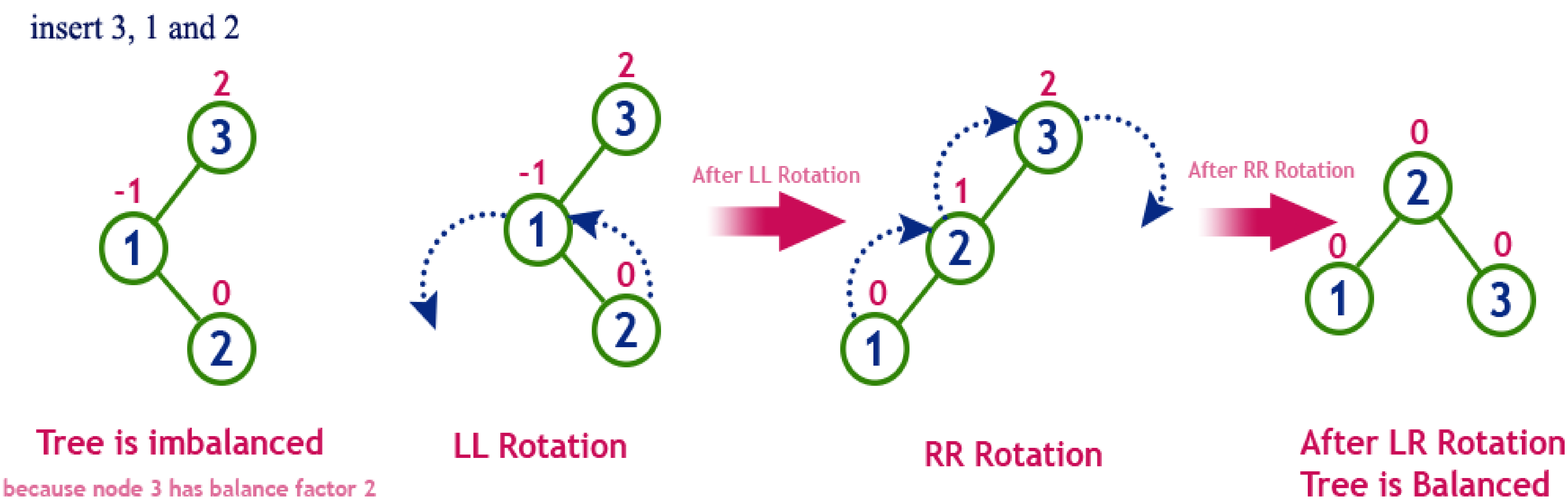
After RR Rotation Tree is Balanced

LR Rotation

- The LR Rotation is a *sequence of single left rotation followed by a single right rotation*.

- In LR Rotation, at first,
  - every node moves one position to the left and
  - one position to right from the current position.

LR Rotation



insert 3, 1 and 2
2
3
-1
1
0
2
Tree is imbalanced
because node 3 has balance factor 2
2
3
-1
1
0
2
LL Rotation
After LL Rotation
2
3
1
2
0
1
RR Rotation
After RR Rotation
0
2
0
1
0
3
After LR Rotation
Tree is Balanced

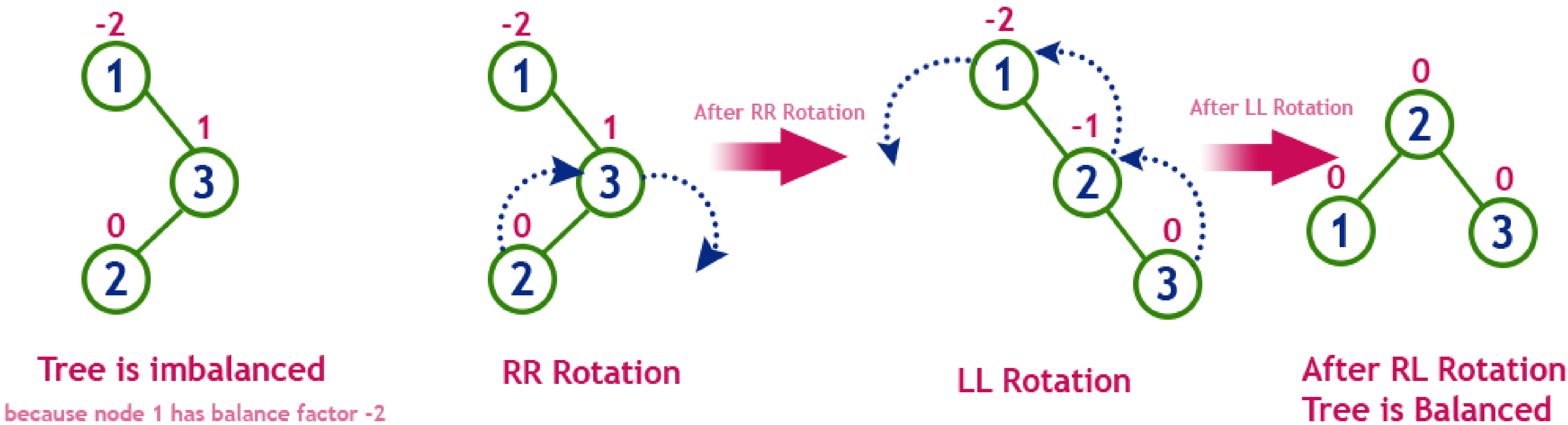
RL Rotation

- The RL Rotation is *sequence of single right rotation followed by single left rotation*.

- In RL Rotation, at first
  - every node moves one position to right and
  - one position to left from the current position.

RL Rotation

insert 1, 3 and 2

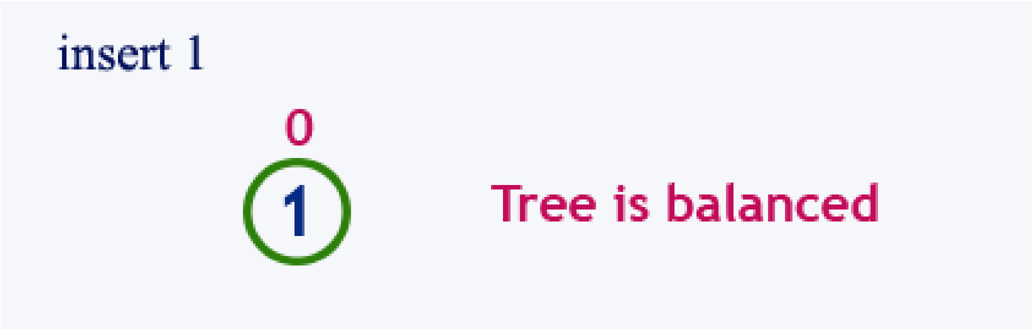


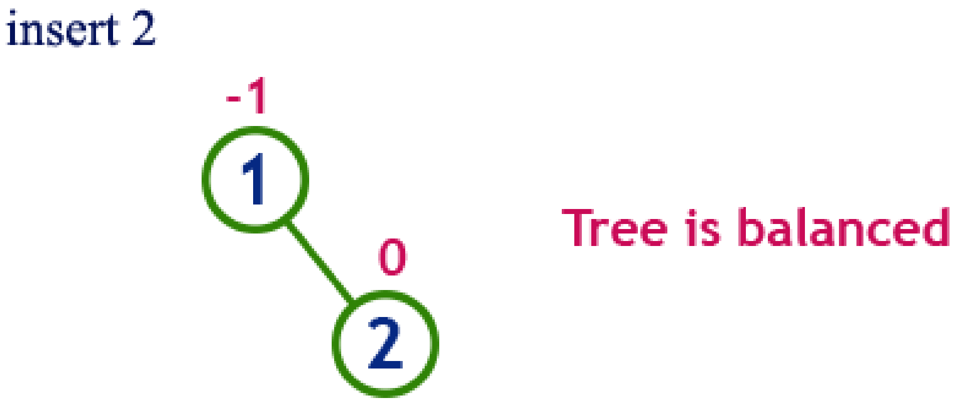
EXAMPLE

Example

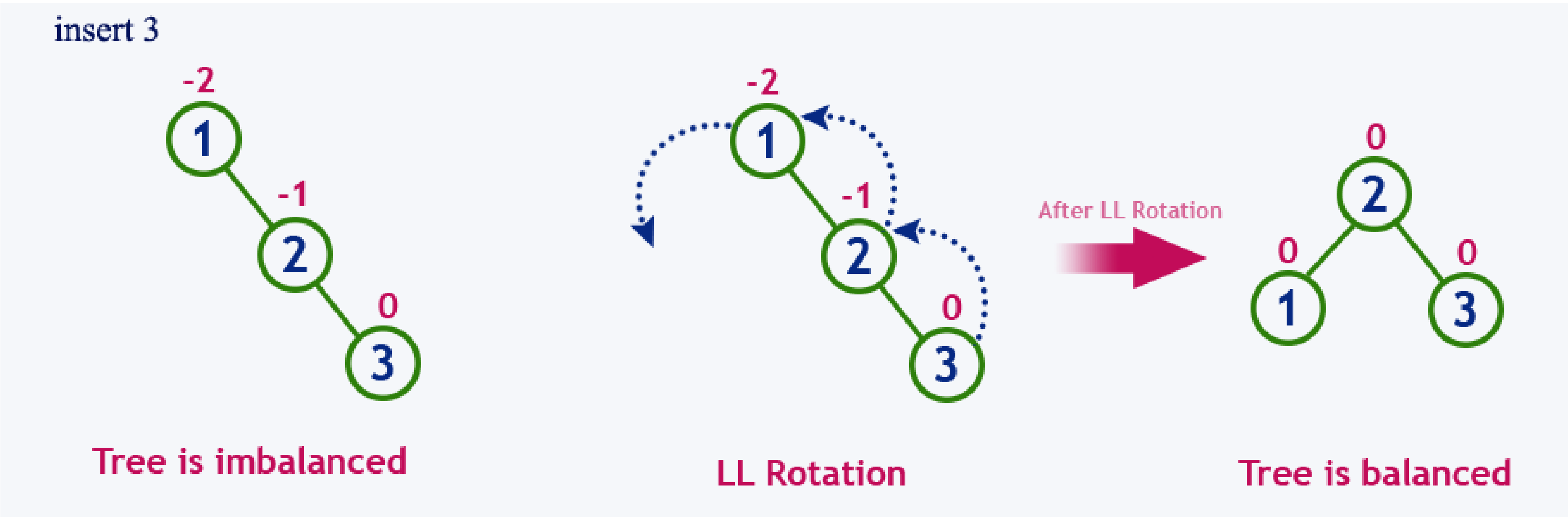
Construct an AVL Tree by inserting numbers from 1 to 8.

Example 1

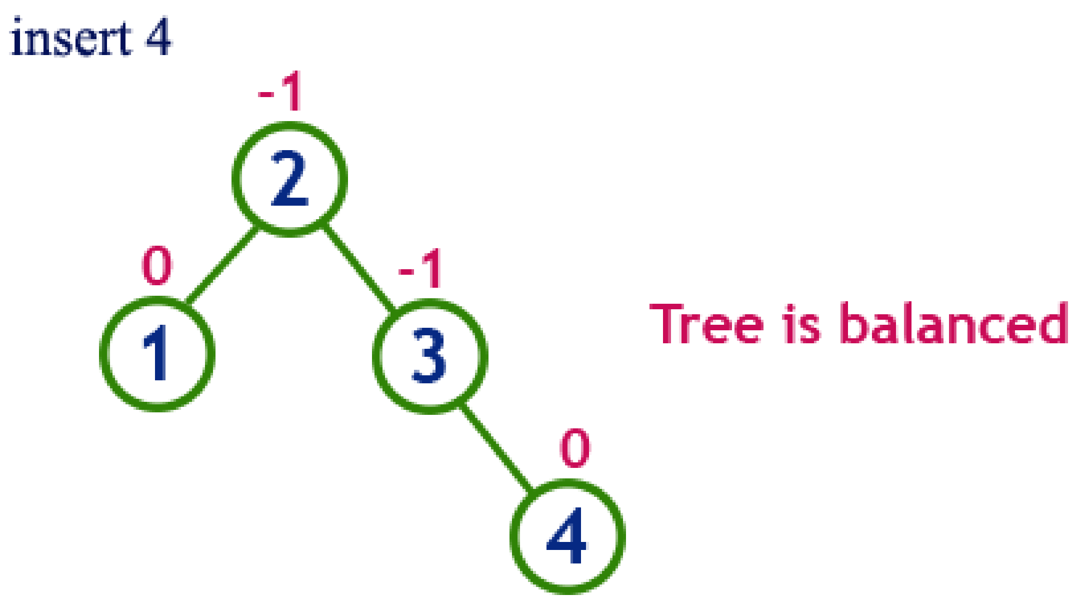




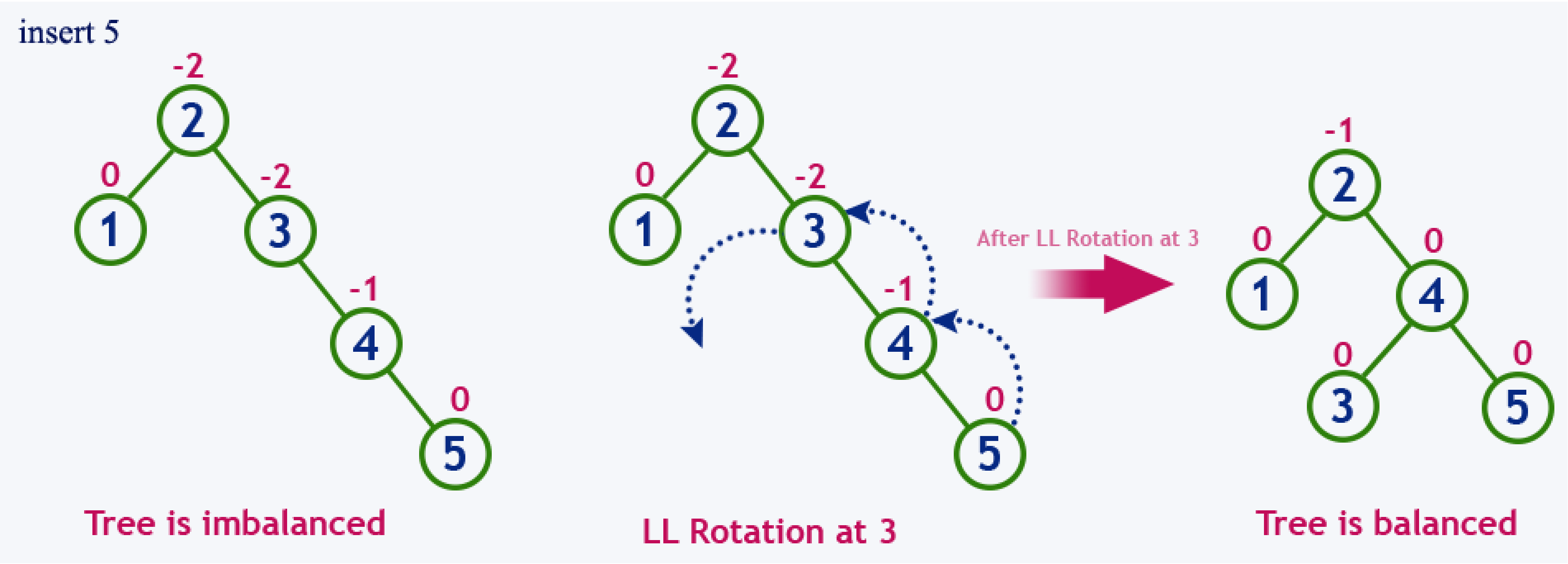
Example 1


insert 3
-2
1
-1
2
0
3
Tree is imbalanced
-2
1
-1
2
0
3
LL Rotation
After LL Rotation
0
2
0
1
0
3
Tree is balanced

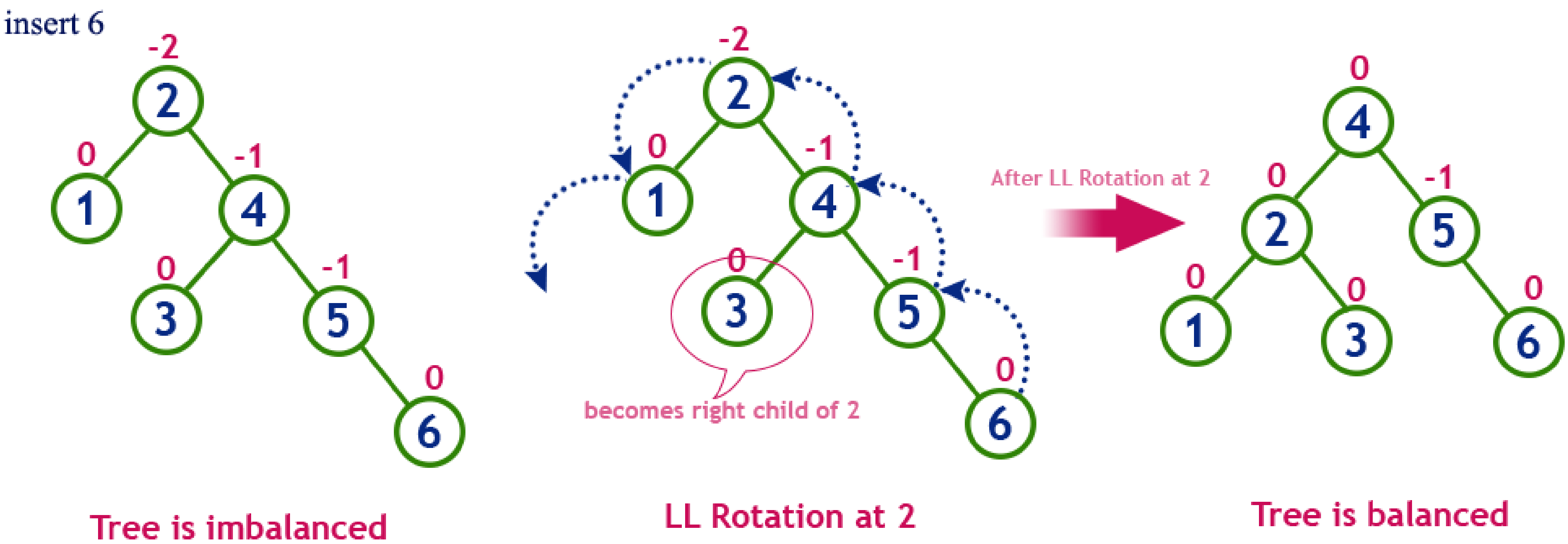
Example 1



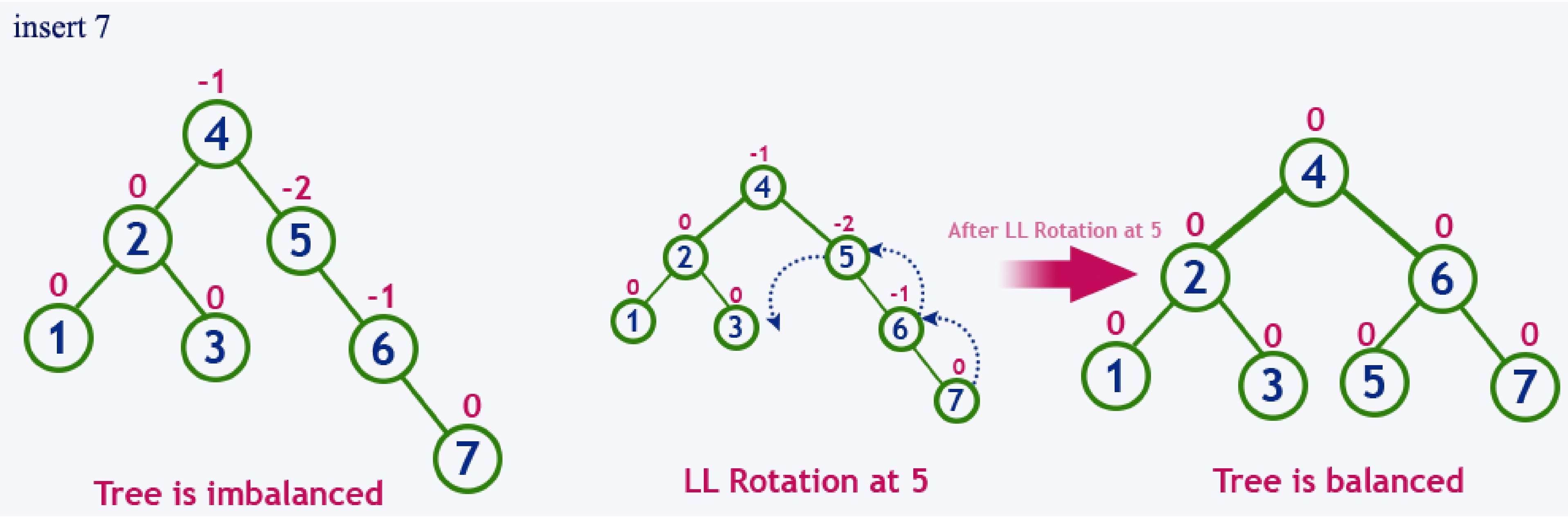
Example 1


insert 5
-2
2
0
1
-2
3
-1
4
0
5
Tree is imbalanced
-2
2
0
1
-2
3
-1
4
0
5
LL Rotation at 3
After LL Rotation at 3
-1
2
0
1
0
4
0
3
0
5
Tree is balanced

Example 1

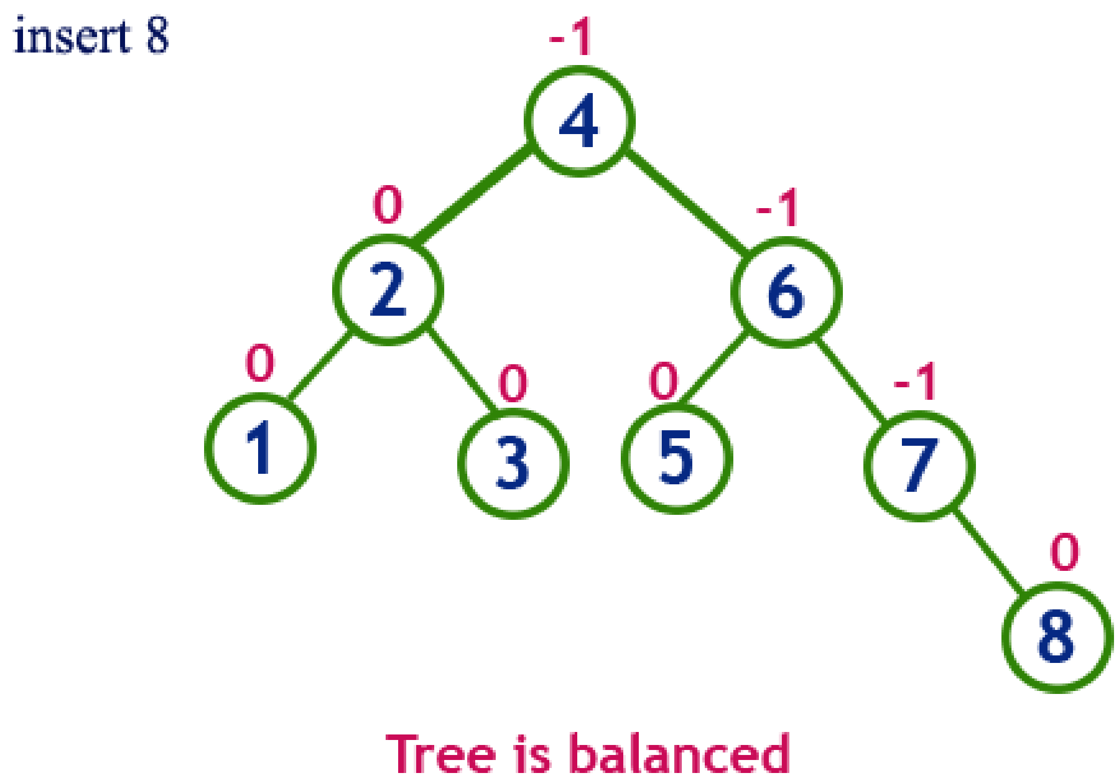

insert 6
-2
2
0
1
-1
4
0
3
-1
5
0
6
Tree is imbalanced
-2
2
0
1
-1
4
0
3
-1
5
0
6
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
0
4
0
2
-1
5
0
1
0
3
0
6
Tree is balanced

Example 1



insert 7
-1
4
0
2
-2
5
0
1
0
3
-1
6
0
7
Tree is imbalanced
-1
4
0
2
-2
5
0
1
0
3
-1
6
0
7
LL Rotation at 5
After LL Rotation at 5
0
4
0
2
0
6
0
1
0
3
0
5
0
7
Tree is balanced

Example 1

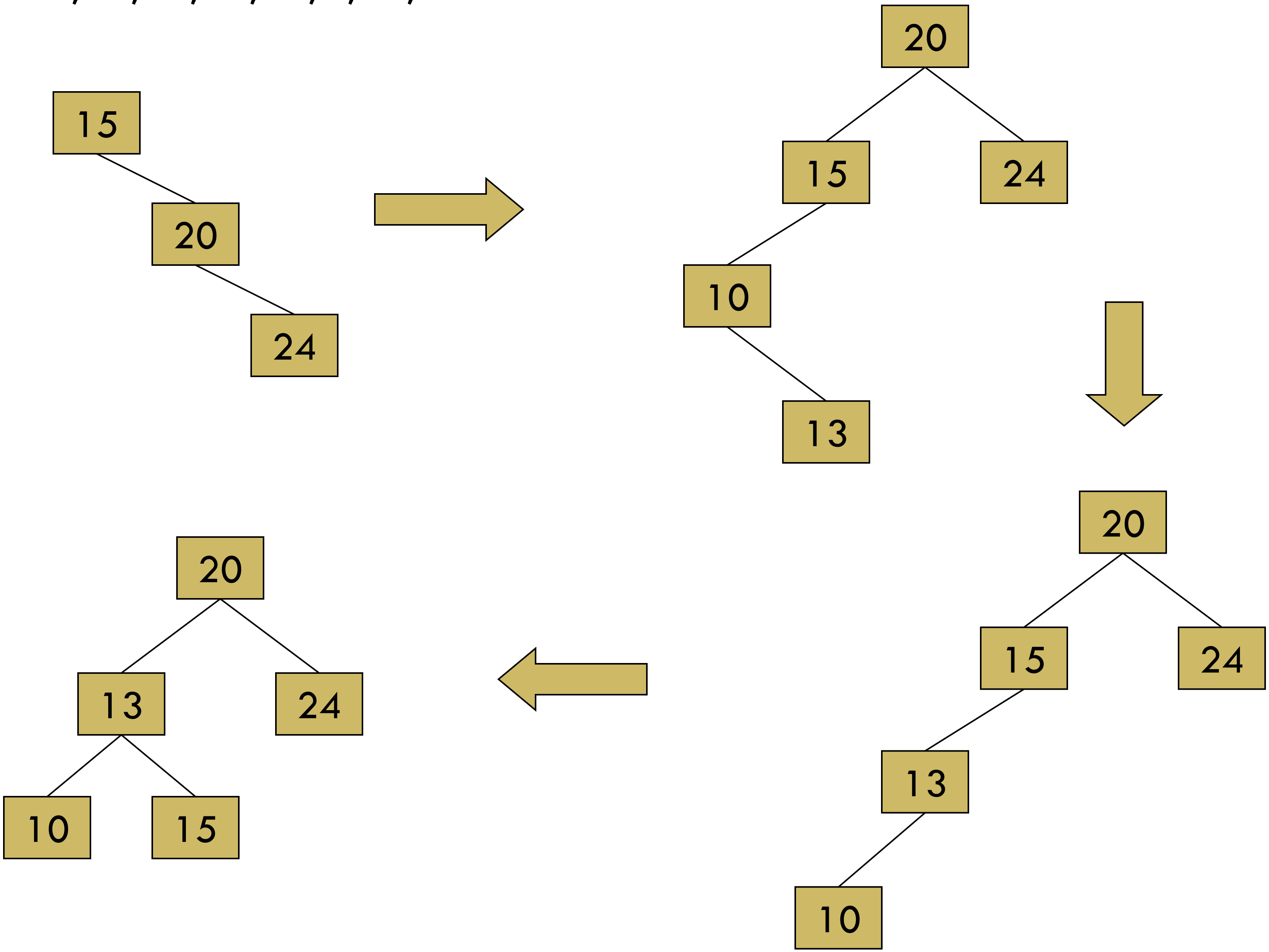


Example 2

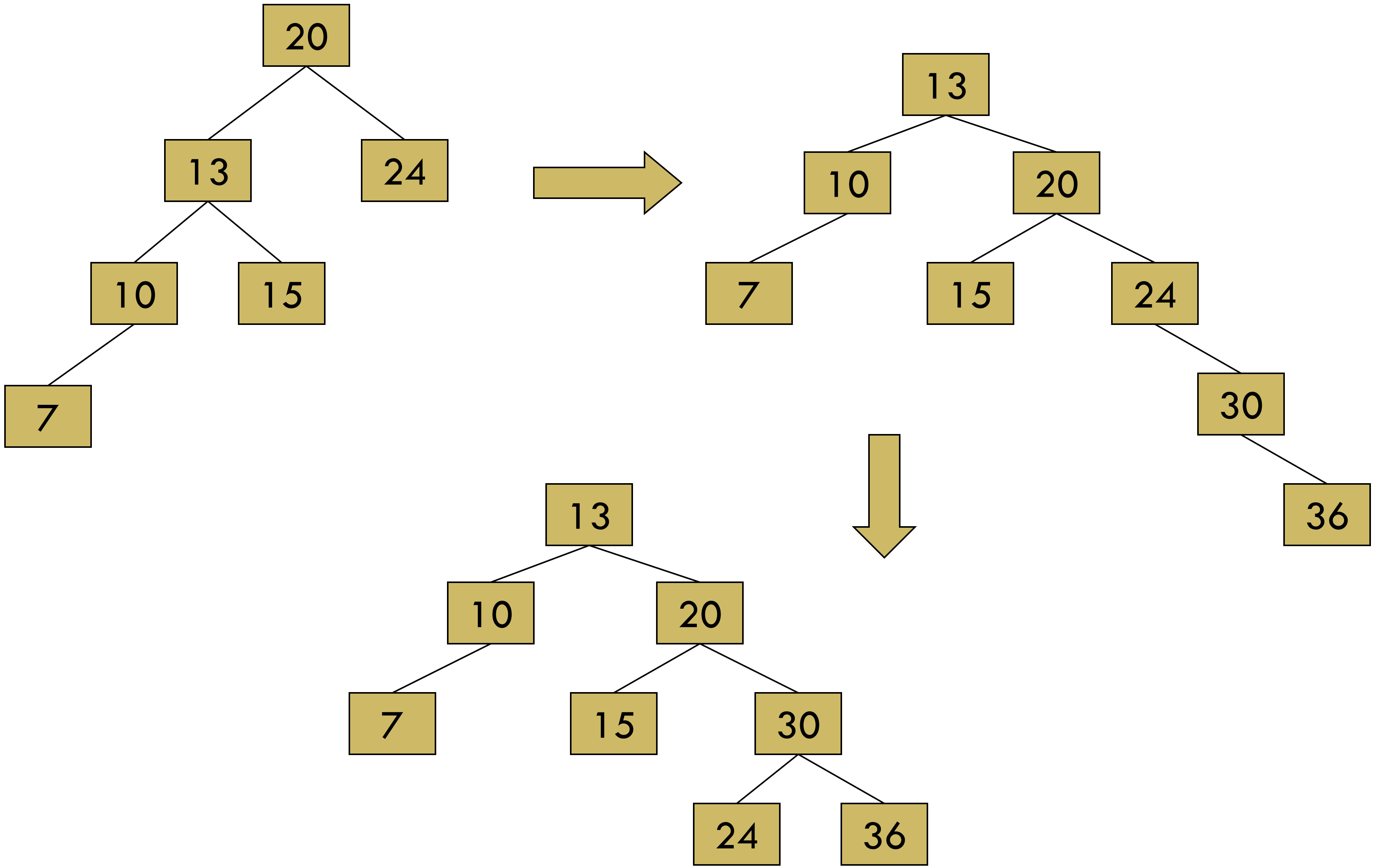
- Build an AVL tree with the following values:

 15, 20, 24, 10, 13, 7, 30, 36

15, 20, 24, 10, 13, 7, 30, 36



15, 20, 24, 10, 13, 7, 30, 36



Reading Materials

- Schaum's Outlines: Chapter # 7
- D. S. Malik: Chapter # 11
- Nell Dale: Chapter # 8
- Allen Weiss: Chapter # 4
- Tenebaum: Chapter # 5