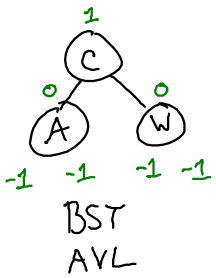
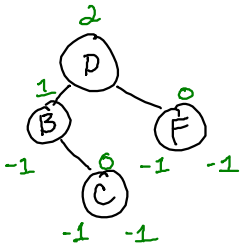
BST —— binary tree where, for all nodes, left desc. have lesser key and right desc. have greater key

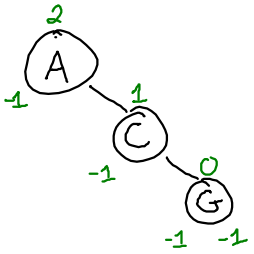
AVL —— BST where, for all nodes, height of left subtree and height of right subtree differ by at most one



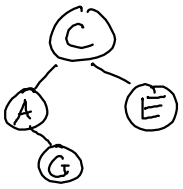
BST
AVL



AVL



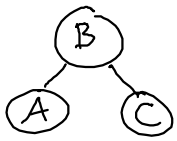
BST, not AVL



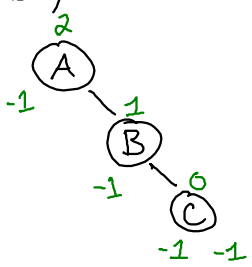
not BST

when a BST is an AVL tree, height of tree is $O(\log n)$

B, A, C



A, B, C



AVL

insertInSubtree (node, key, value)

If node == null :

    Return new Node (key , value)

Else If key < node.key :

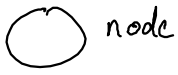
    node.left ← insertInSubtree (node.left, key, value)

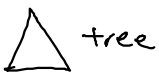
    node ← rebalance (node)

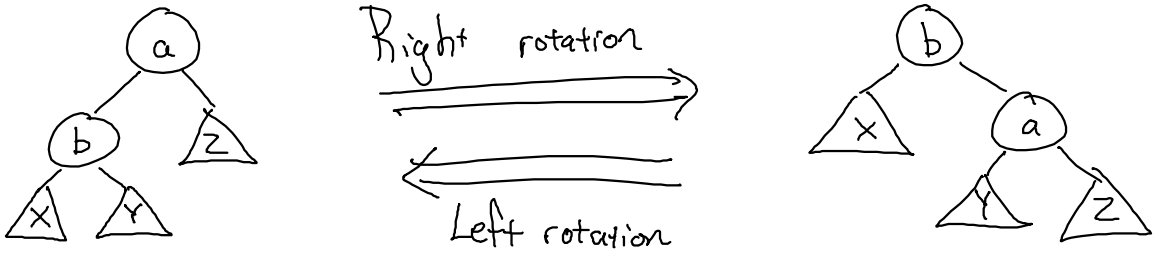
    recalcHeight(node)

    Return node

Tree Rotation

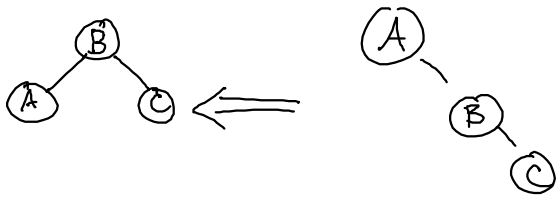




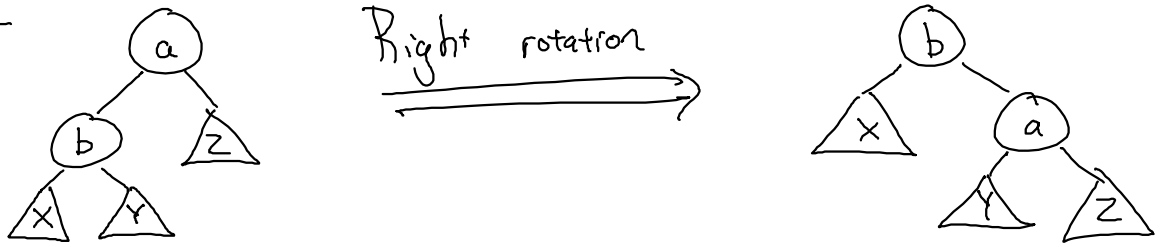



```
Function rotateRight(node):
    a ← node
    b ← node.left
    X ← b.left
    Y ← b.right
    Z ← a.right
    a.left ← Y
    b.right ← a
    Return b
EndFunction
```

O(1)



BST

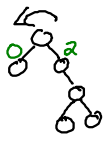


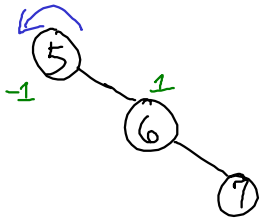
Rotations preserve BST invariant

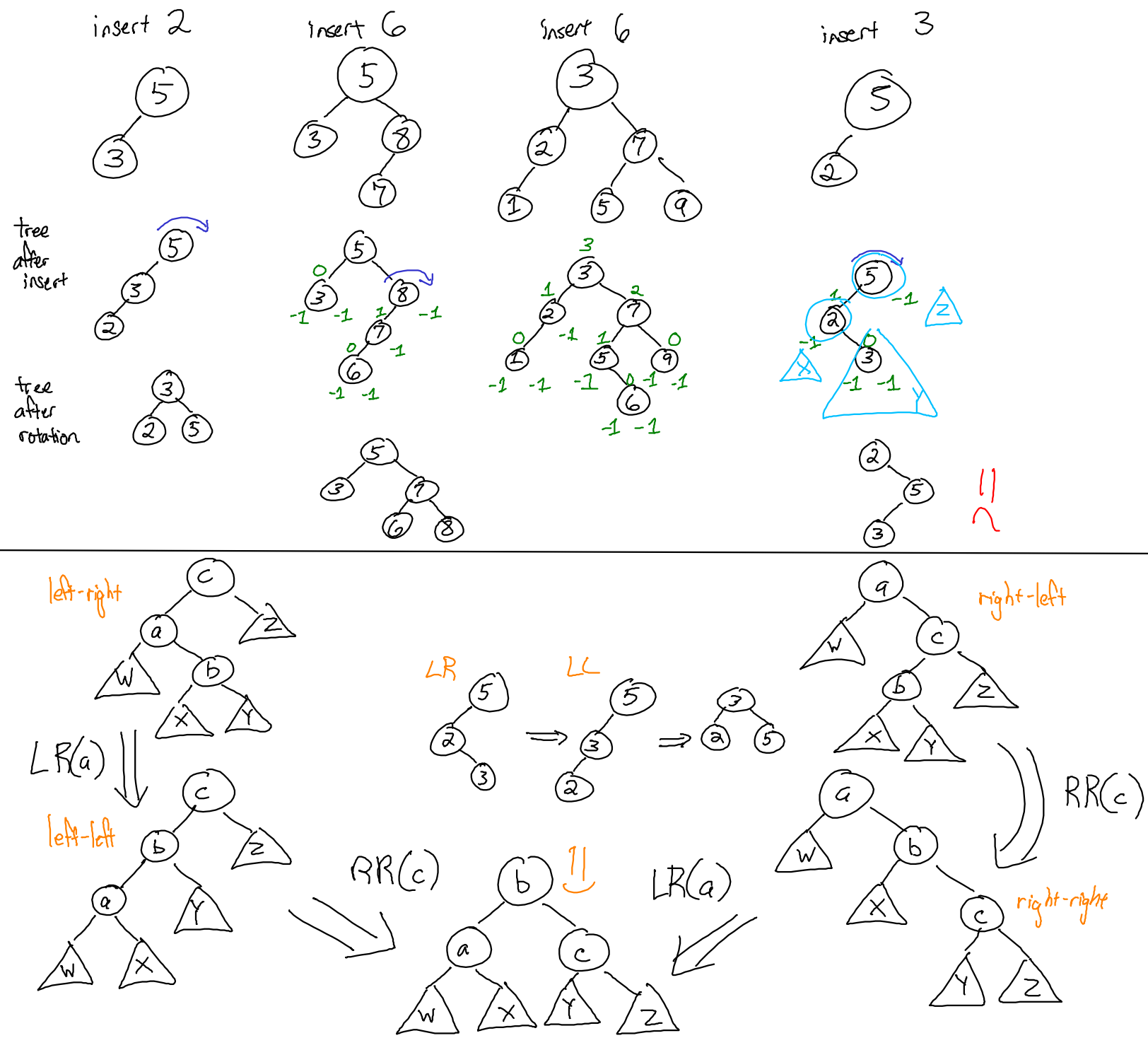
b < a
X < a
Y < a
a < Z
X < b
b < Y

!! not this one

```
Function rebalance(node):                          right-heavy
    If node.left.height + 1 < node.right.height:
        node ← rotateRight(node)                   left-heavy
    Else If node.left.height > node.right.height + 1:
        node ← rotateLeft(node)
    EndIf
EndFunction
```








```
Function rebalance(node):
    If node.left.height + 1 < node.right.height:
        If node.left.left.height < node.left.right.height:
            node.left ← rotateLeft(node.left)
        EndIf
        node ← rotateRight(node)
    Else If node.left.height > node.right.height + 1:
        ...
```