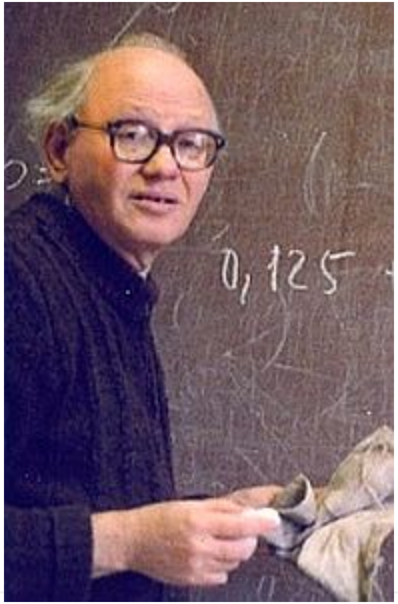
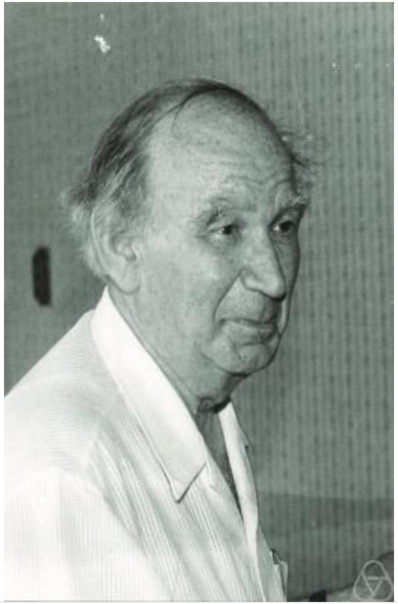
Lecture 32
AVL Trees

December 07, 2021
Tuesday

NAMED AFTER INVENTORS

- Invented by Georgy Adelson-Velsky and Evgenii Landis in 1962





The AVL Tree Data Structure

An AVL tree is a self-balancing binary search tree.

Structural properties

1. Binary tree property (same as BST)
2. Order property (same as for BST)
3. Balance condition:

    balance of every node is between -1 and 1

    where balance(node) = height(node.left) – height(node.right)

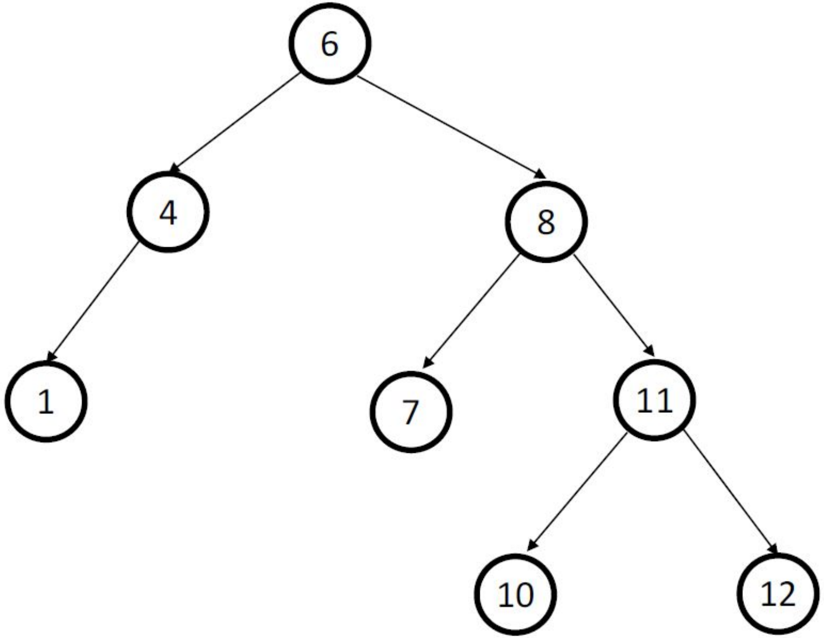
Example #1: Is this an AVL Tree?

Balance Condition:

balance of every node is between -1 and 1

where balance(node) =

height(node.left) – height(node.right)



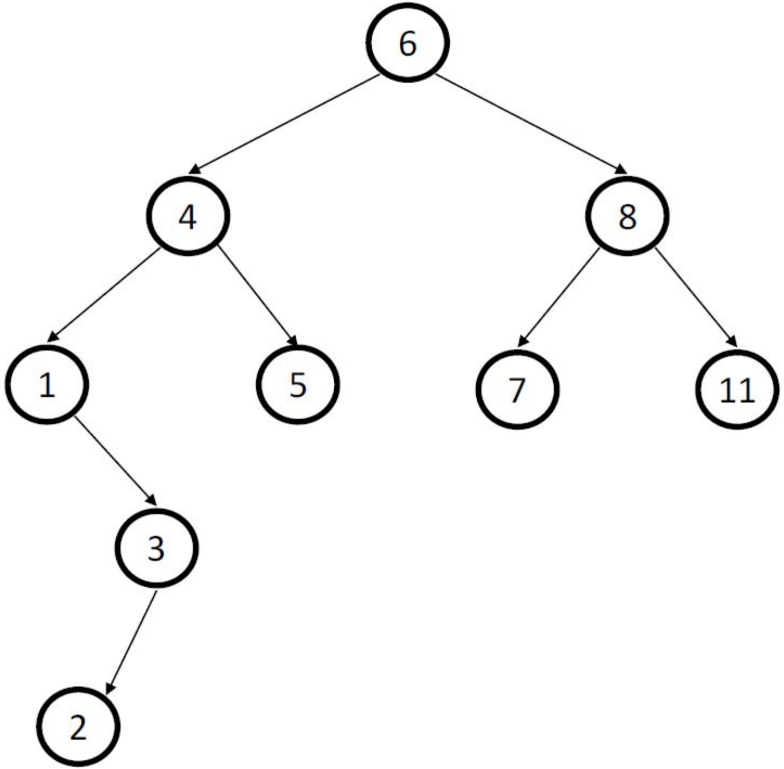
Example #1: Is this an AVL Tree?

Balance Condition:

balance of every node is between -1 and 1

where balance(node) =

height(node.left) – height(node.right)



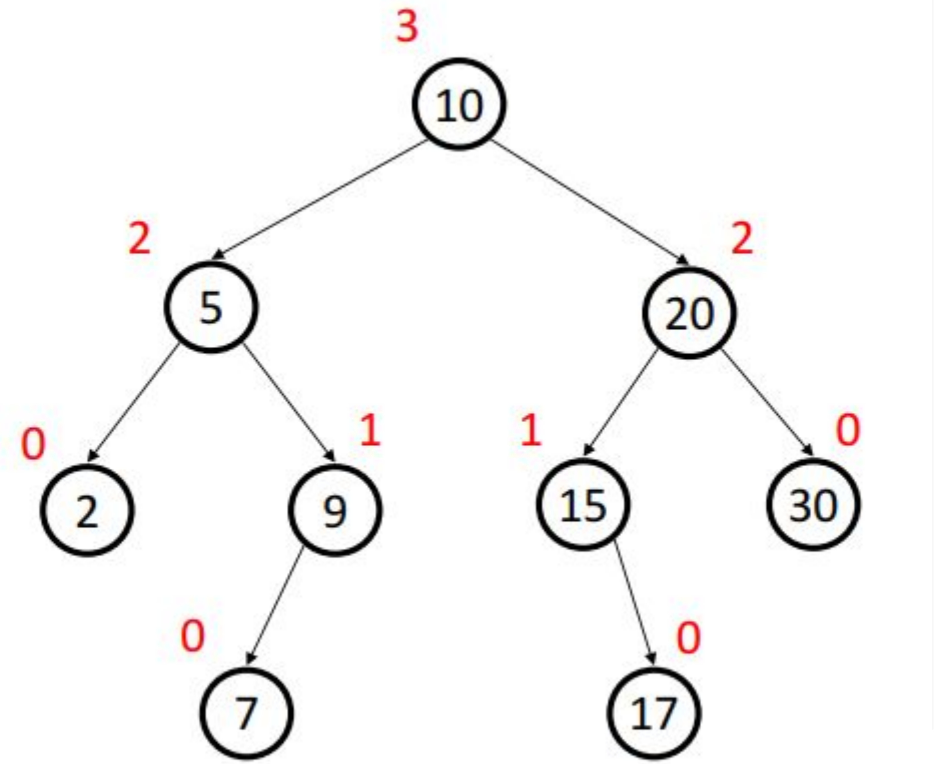
Example #1: Is this an AVL Tree?

Balance Condition:

balance of every node is between -1 and

where balance(node) =

height(node.left) – height(node.right)



AVL TREE & PERFECTLY BALANCED TREE

- First Technique guarantees perfectly balanced tree
- DSW algorithm guarantees perfectly balanced tree.
- AVL does not guarantee perfectly balanced tree.

PERFECTLY BALANCED BINARY TREE

Bounds of the AVL Tree

- $\lg(n + 1) \leq h < 1.44\lg(n + 2) - 0.328$

The worst case search requires O(lg n) comparisons for perfectly balanced tree. For the same height AVL

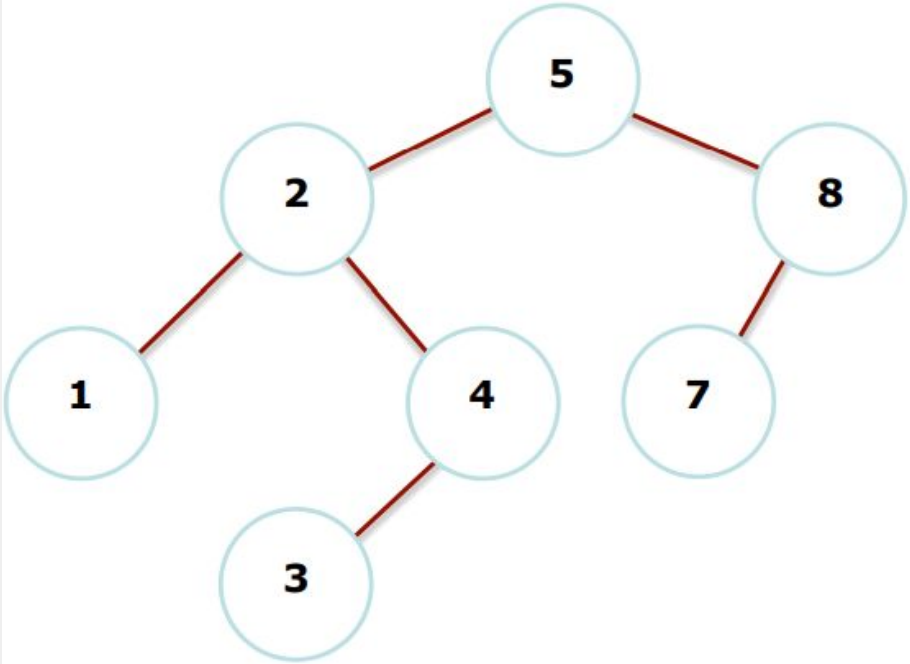
$$h = \lceil \lg(n + 1) \rceil$$

Therefore, the search time **in the worst case in an AVL tree is 44% worse** than in the best case tree configuration.

INSERTION IN AVL

Height information is kept for each node, and the height is almost log N in practice.

- When we insert into an AVL tree, we have to **update the balancing factor** back up the tree .
- We also have to **maintain the AVL property** - tricky? Think about **inserting 6** into the tree: this **would upset the balance at node 8.**



UPDATING THE BALANCE FACTOR

Please refer to the Chapter 06. Binary Trees

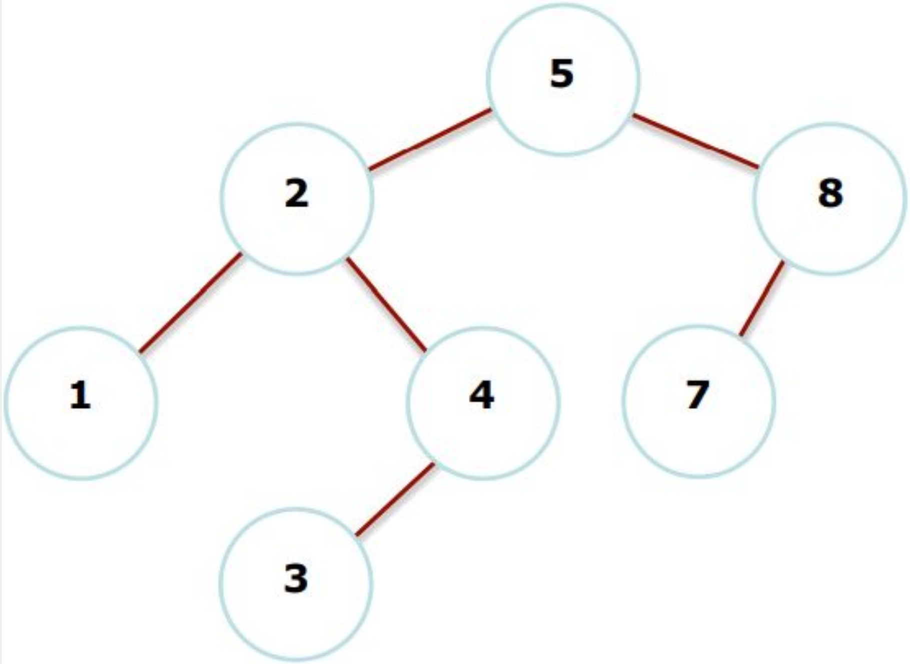
Page 258

ROTATION

A simple modification of the tree, called **rotation**, can restore the AVL property.

- After insertion, **only nodes on the path from the insertion might have their balance altered,** because only those nodes had their subtrees altered.

- We will re-balance as we follow the path up to the root updating balancing information.



IMBALANCE WITH INSERTION

An AVL tree can become out of balance in four situations, however, two are symmetrical. Inserting a node in the

1. **Right subtree of the right child (Left-Left).**
    a. **Left subtree of the left child (Right-Right).**
2. **Left subtree of the right child (Left-Right).**
    a. **Right subtree of the left child (Right-Left).**

RESTORING AVL PROPERTY

Outside cases require Single Rotation

1. **Right subtree of the right child (Left-Left).**
    a. **Left subtree of the left child (Right-Right).**

Inside cases require Double Rotation

2. **Left subtree of the right child (Left-Right).**
    a. **Right subtree of the left child (Right-Left).**

SELF-BALANCING SEARCH TREES

There are many different implementations of self-balancing search trees
(e.g. Red-black trees, AVL trees, B-tree, 2-3-4 trees)

Today, we'll go over a key primitive that's used in these implementations:
ROTATIONS

Note: going forward, we're going to focus rotations for on BINARY search trees (BSTs).

ROTATIONS

IDEA: locally rebalance a node's subtree in O(1) time while maintaining BST property

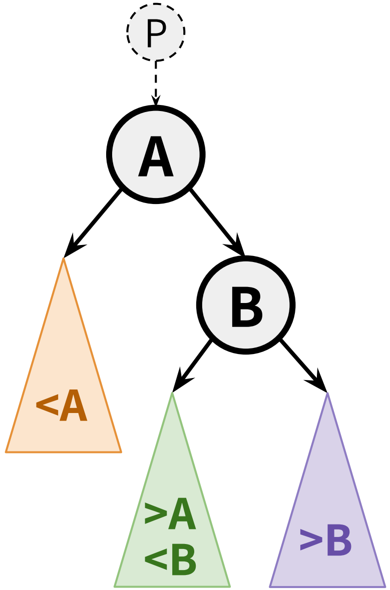
LEFT ROTATION

RIGHT ROTATION

ROTATIONS

IDEA: locally rebalance a node's subtree in O(1) time while maintaining BST property

LEFT ROTATION

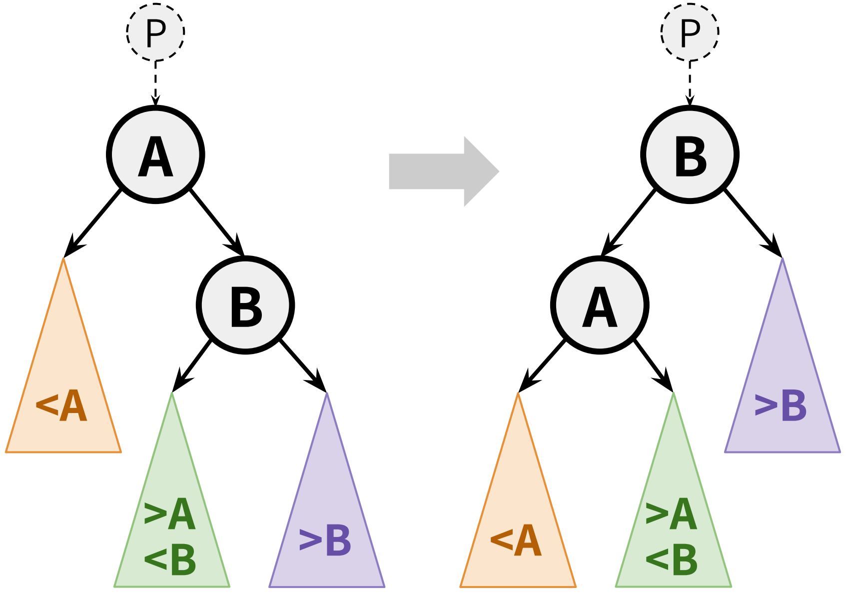


RIGHT ROTATION

ROTATIONS

IDEA: locally rebalance a node's subtree in O(1) time while maintaining BST property

LEFT ROTATION

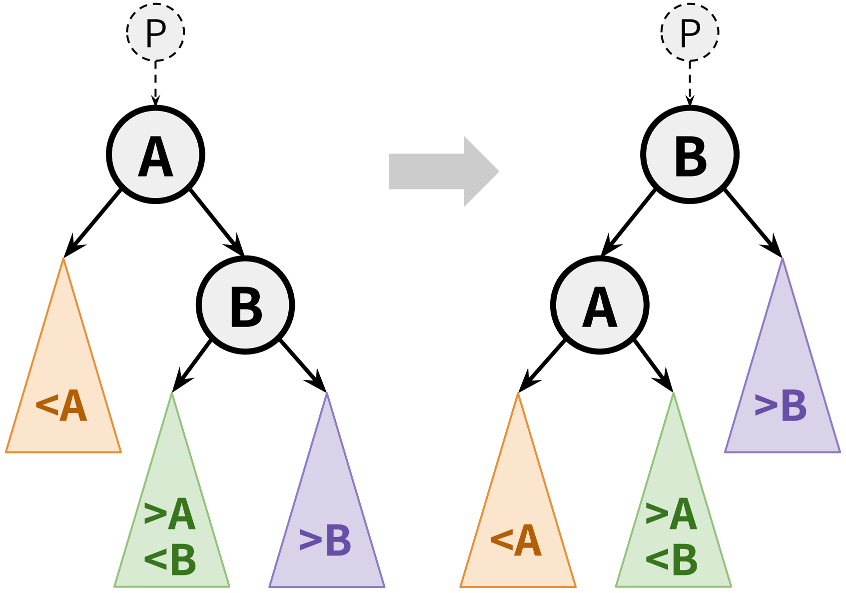


RIGHT ROTATION

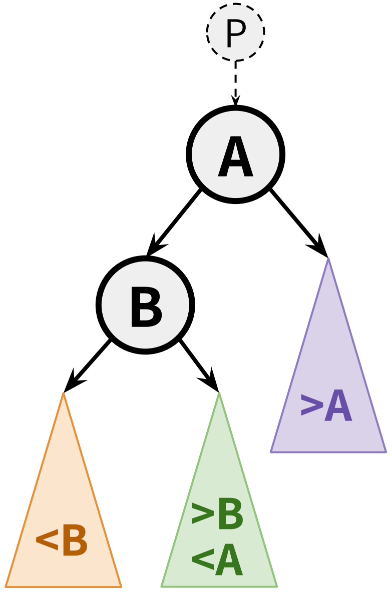
ROTATIONS

IDEA: locally rebalance a node's subtree in O(1) time while maintaining BST property

LEFT ROTATION



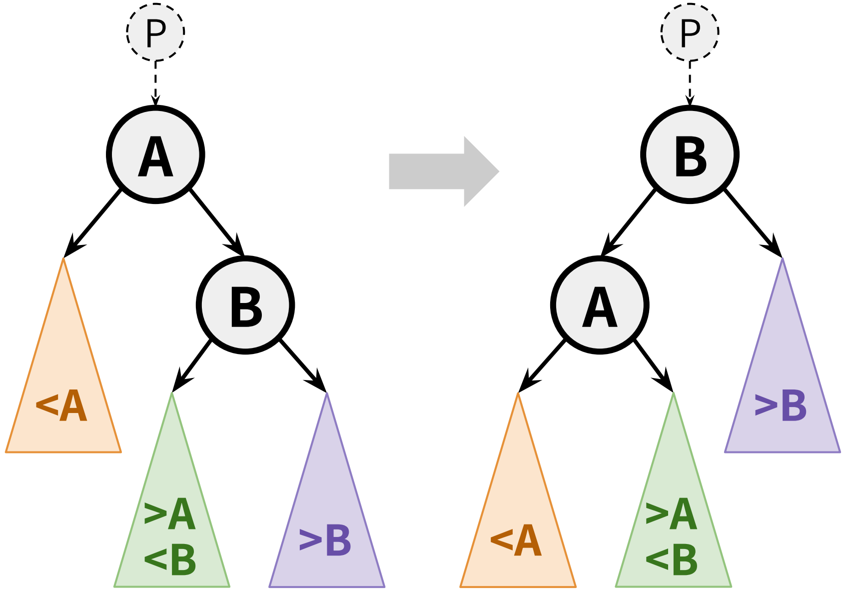
RIGHT ROTATION



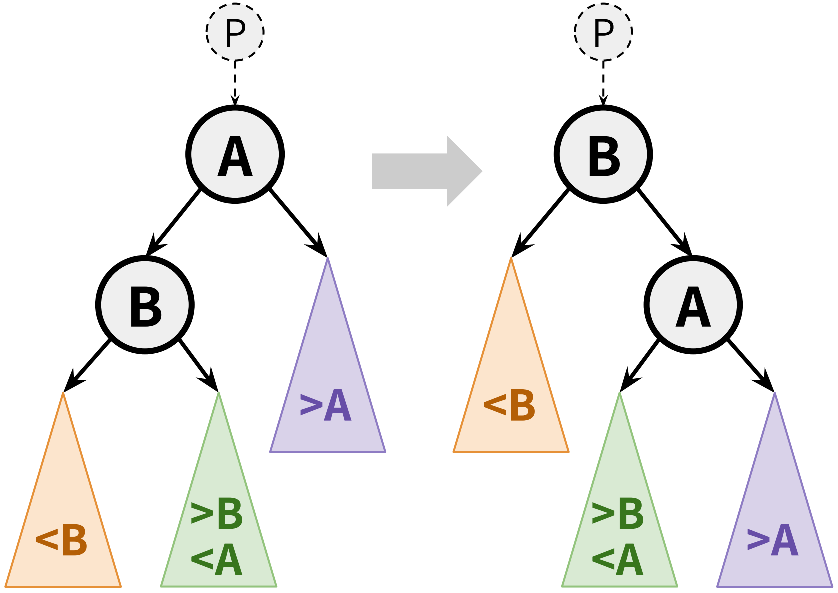
ROTATIONS

IDEA: locally rebalance a node's subtree in O(1) time while maintaining BST property

LEFT ROTATION

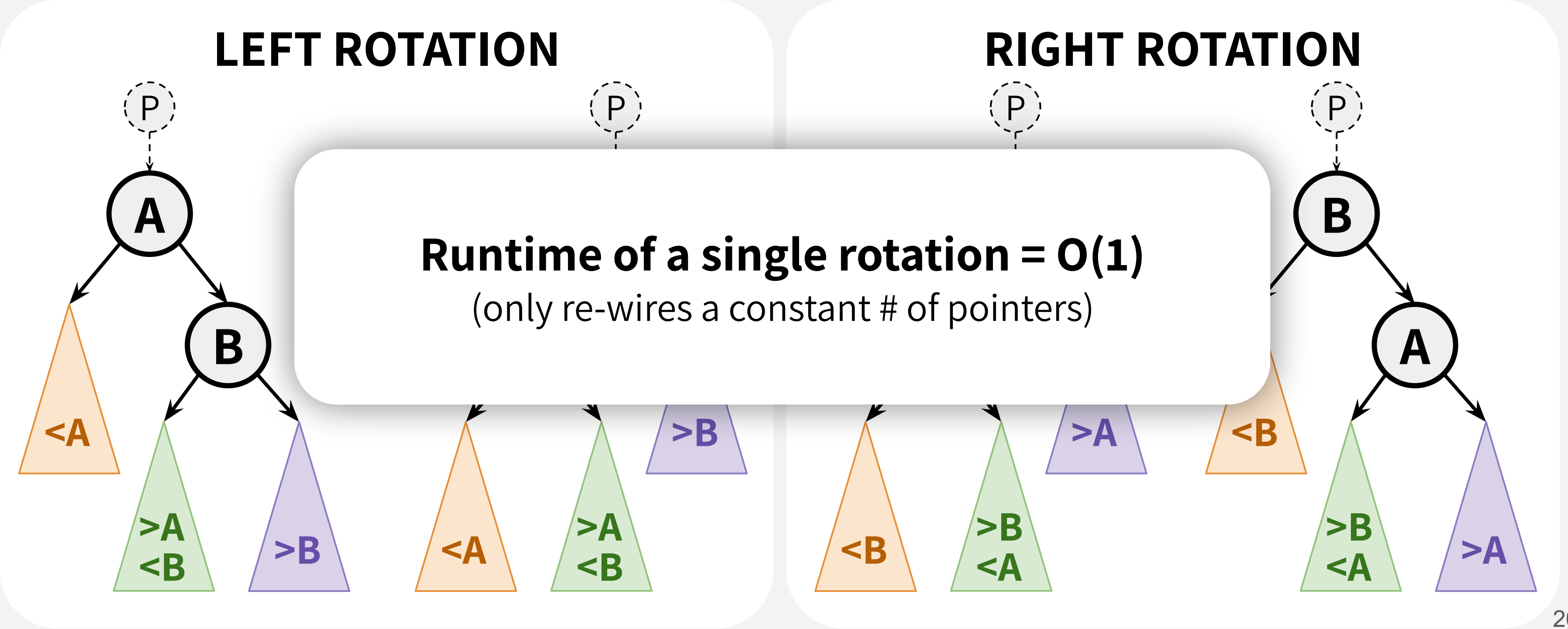


RIGHT ROTATION



ROTATIONS

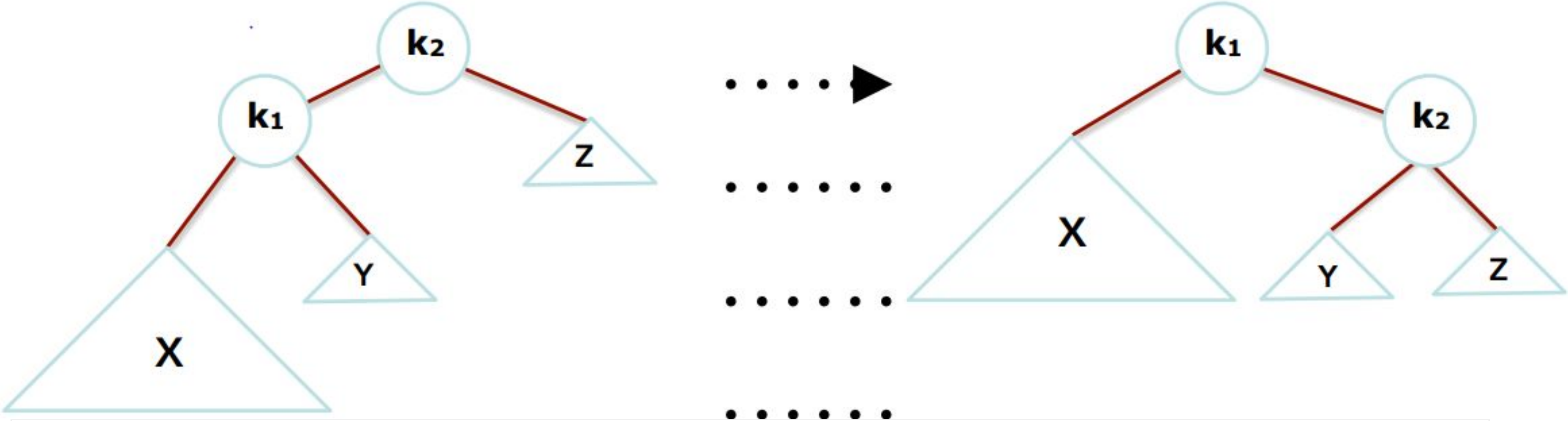
IDEA: locally rebalance a node's subtree in O(1) time while maintaining BST property



Runtime of a single rotation = O(1)

(only re-wires a constant # of pointers)

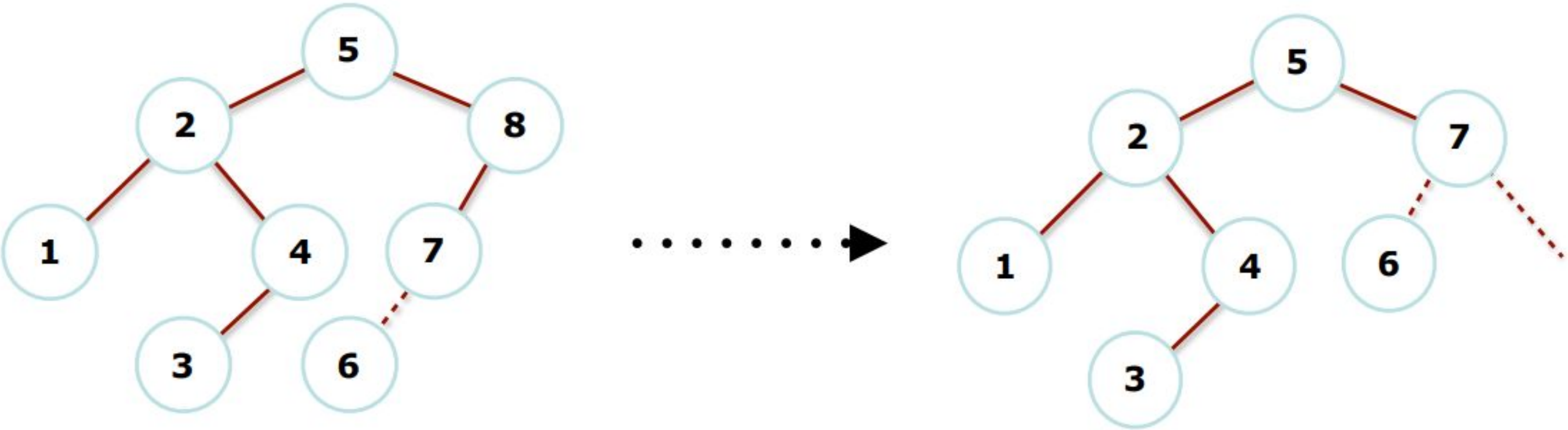
RESTORING AVL PROPERTY



k_2 violates the AVL property, as **X** has grown to be **2 levels deeper than Z**. **Y** cannot be at the same level as **X** because **k_2** would have been out of balance before the insertion.

We would like to move X up a level and Z down a level (fine, but not strictly necessary).

RESTORING AVL PROPERTY

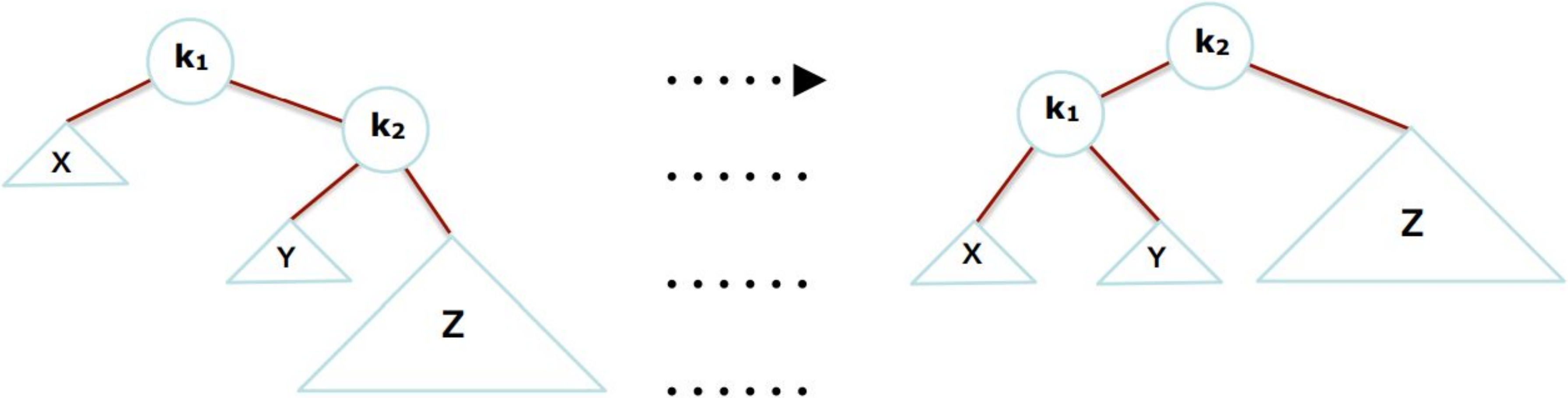


Which node is imbalance?

Which rotation is required to restore the AVL Property

Write the steps of Rotation on Your notebook.

RESTORING AVL PROPERTY



Which node is imbalance?

Which rotation is required to restore the AVL Property

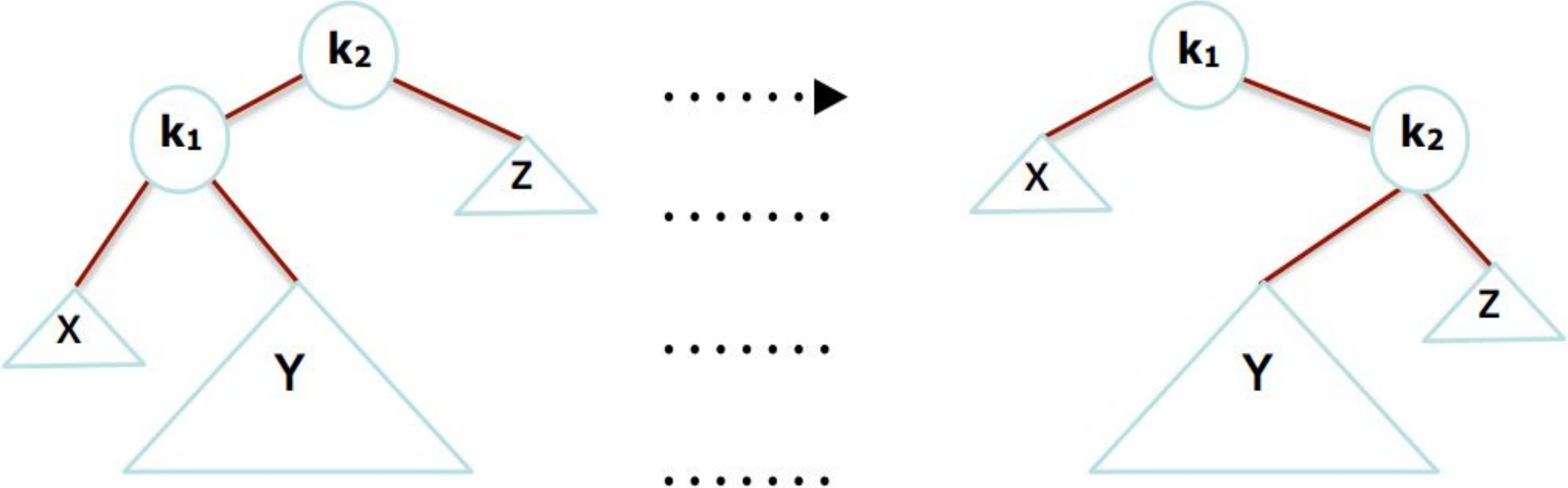
Write the steps of Rotation on Your notebook.

AVL VISUALIZATION

Insert 3, 2, 1, 4, 5, 6, 7

http://www.cs.usfca.edu/~galles/visualization/AVLtree.html

AVL INNER CASE WITH SINGLE ROTATION

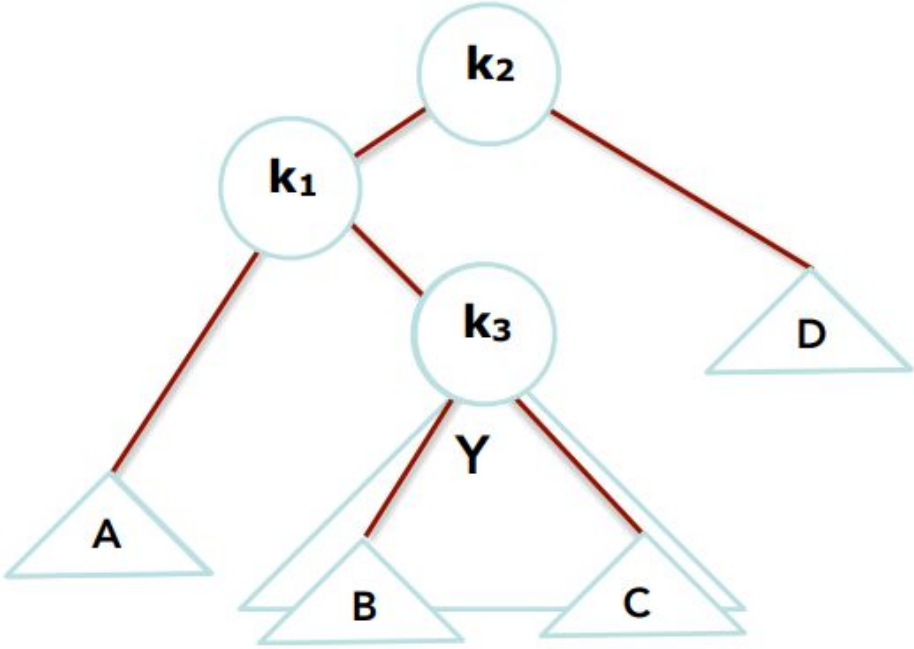


Single Rotation doesn’t work for right-left, left-right cases!

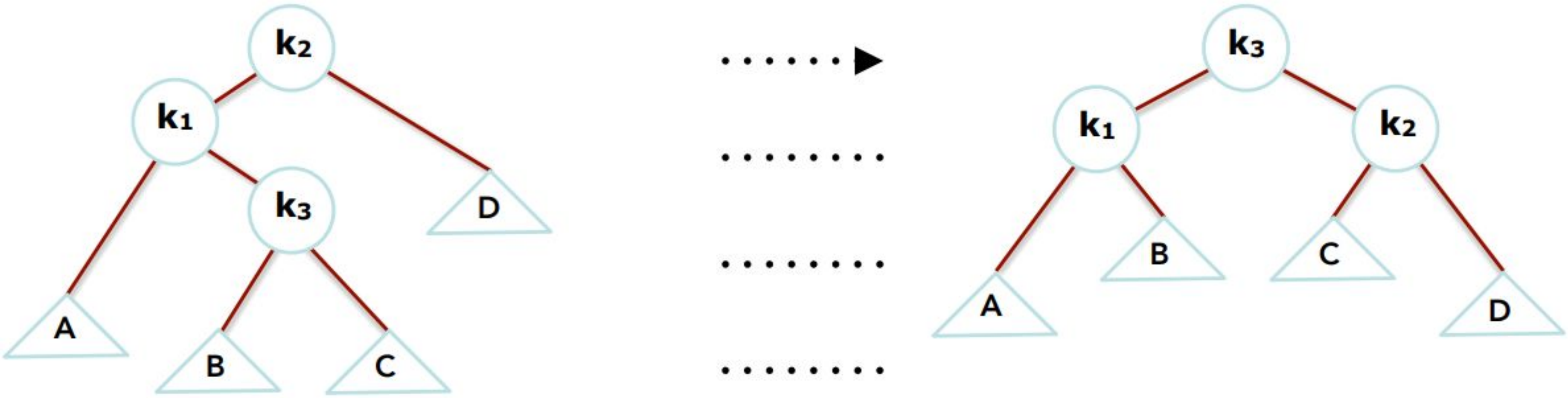
DOUBLE ROTATION

You can think of double rotation as one complex rotation or Two Simple Single Rotations.

- **Instead of three subtrees**, we can view the **tree as four subtrees, connected by three nodes**



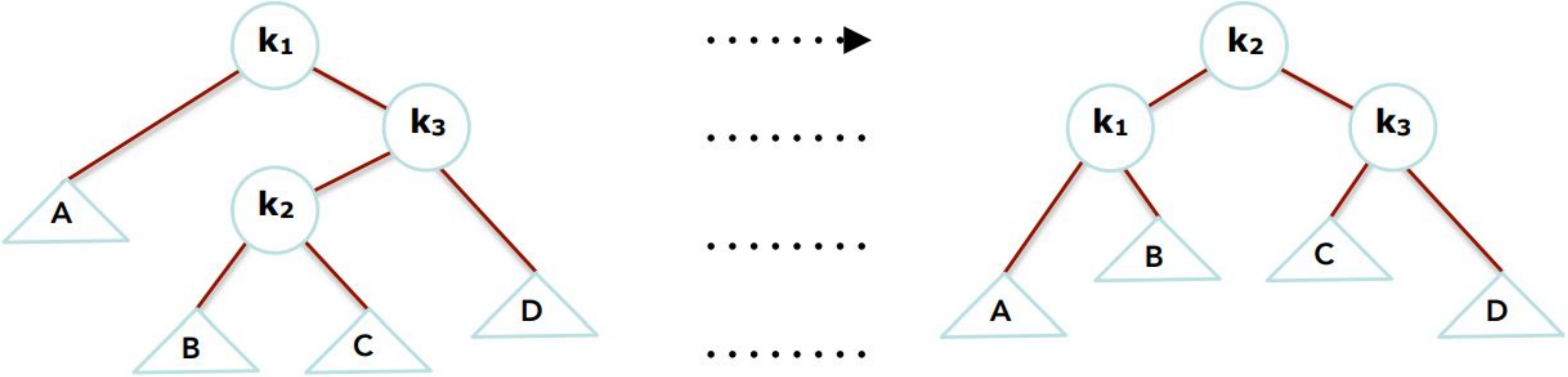
DOUBLE ROTATION



For Left-Right Case:

- **Left Rotation** Followed by a **Right Rotation** will restore the AVL Property.

DOUBLE ROTATION



For Right-Left Case:

- **Right Rotation** Followed by a **Left Rotation** will restore the AVL Property.

DELETION IN AVL

Deletion may be more time-consuming than insertion.

First, we apply deleteByCopying() to delete a node.

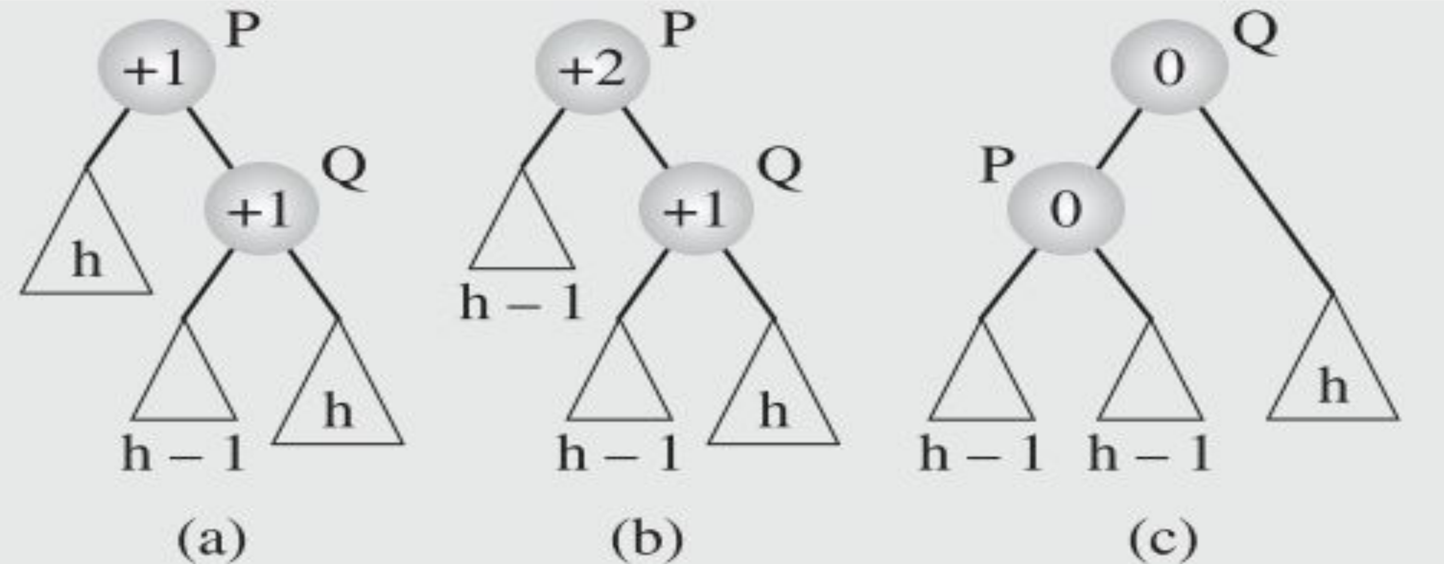
- This technique allows us to reduce the problem of deleting a node with two descendants to deleting a node with at most one descendant.
- After a node has been deleted from the tree, balance factors are updated from the parent of the deleted node up to the root.

DELETION IN AVL

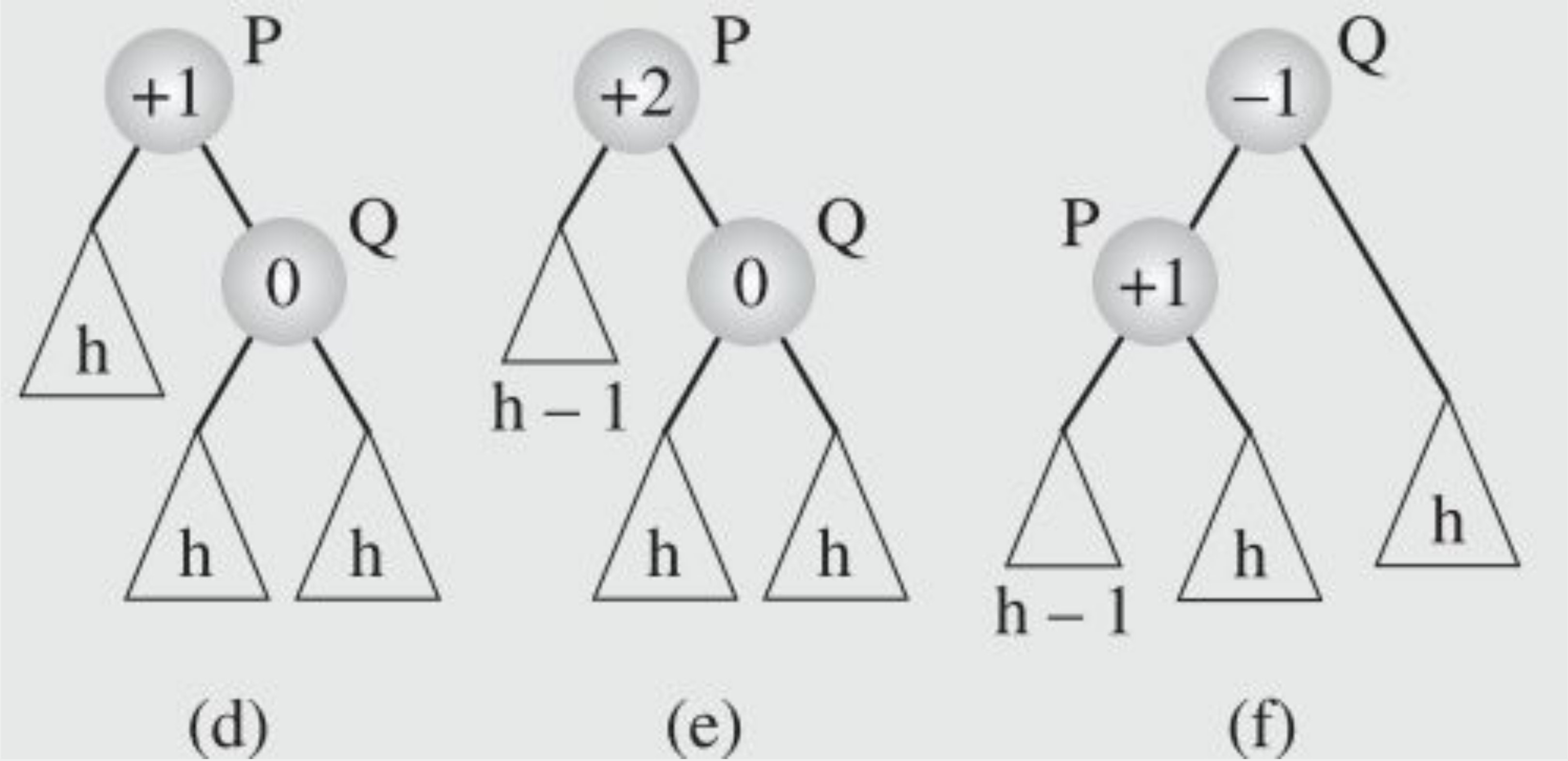
- Importantly, **the rebalancing does not stop** after the first node P is found for which the balance factor would become ±2, as is the case with insertion.
- Hence, deletion leads to at most O(lg n) rotations, because in the worst case
  - Every node on the path from the deleted node to the root may require rebalancing.
- Deletion of a node may improve the balance factor of its parent from ∓1 to 0 can also make grandparent from ∓1 to ∓2

CASE I - P = 1 & Q = 1

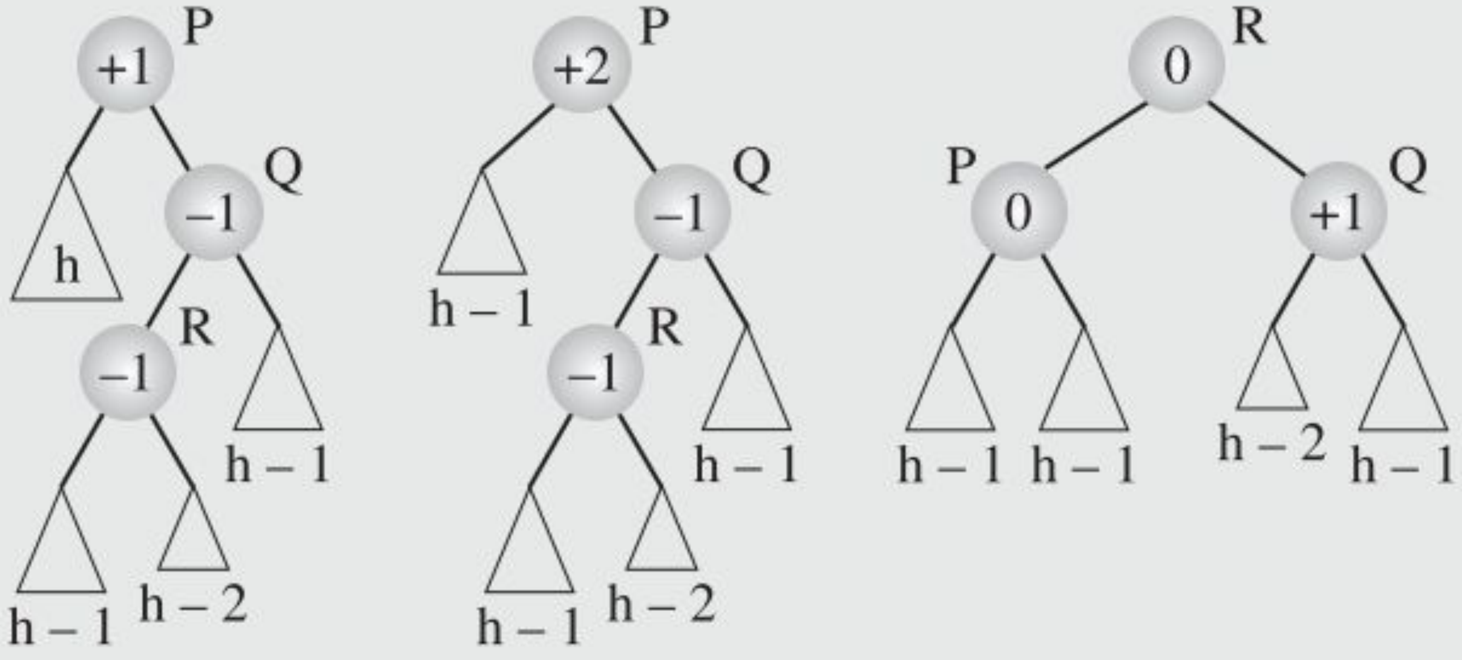
- There are 4 cases (along with 4 symmetric) which leads to immediate rotation. In each of these cases we assume that left child of node P is deleted.



CASE II - P = 1 & Q = 0



CASE III



(g) (h) (i)

CASE IV