UVA DATA SCIENCE

Foundation of Computer Science for Data Science

AVL Trees

Mai Dahshan

October 9, 2024

Learning Objectives

- Understand the concept of AVL trees, their properties, and how they are organized
- Searching, inserting and deleting in AVL trees
- Analyzing AVL trees time complexity

Motivation

- Balancing binary tree is hard to when we allow dynamic insert and remove.
  - We want a tree that has the following properties
    - Tree height = O(log(N))
    - allows dynamic insert and remove with O(log(N)) time complexity.
  - The AVL tree is one of this kind of trees.

AVL Trees

the height of an empty sub tree is -1

- AVL Trees are a form of balanced binary search trees
- Has an additional ***height constraint***:
  - For each node x in the tree, Height(x.left) differs from Height(x.right) by at most 1
- Named after the initials of their inventors
  - Adelson-Velskii and Landis
  - guarantees O(log n) worst-case for any sequence of insert and delete operations.

AVL Trees

- To be an AVL tree, must **always:**
    - Be a ***binary search tree***
    - Satisfy the ***height constraint***

AVL Trees

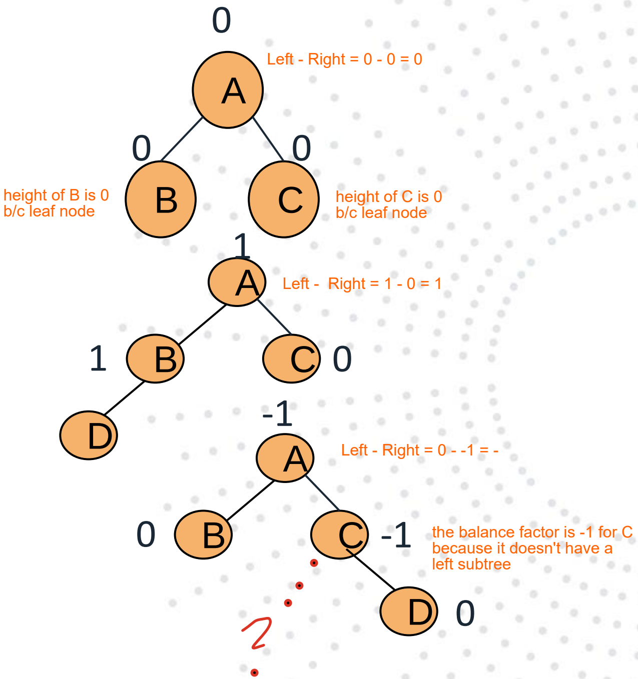
- AVL Trees aim for the next thing:
  - For any node, the heights of its two children can differ by at most 1
  - Define height of empty subtree as -1
  - Height of a tree = 1 + max{height-left, height-right}

Balance Factor

- To check the **balance constraint**, we have to know the ***height h*** of each node
- The balance factor bf(x) = ~~h(x.right) – h(x.left)~~ h(x.left) - h(x.right)
  - bf(x) values -1, 0, and 1 are allowed.
  - If bf(x) < -1 or bf(x) > 1 then tree is **NOT AVL**

Balance Factor

- If the balance factor is **0**, the node is perfectly balanced (i.e., the left and right subtrees have the same height)
- If the balance factor is **+1**, the left subtree is one level taller than the right subtree
- If the balance factor is **-1**, the right subtree is one level taller than the left subtree

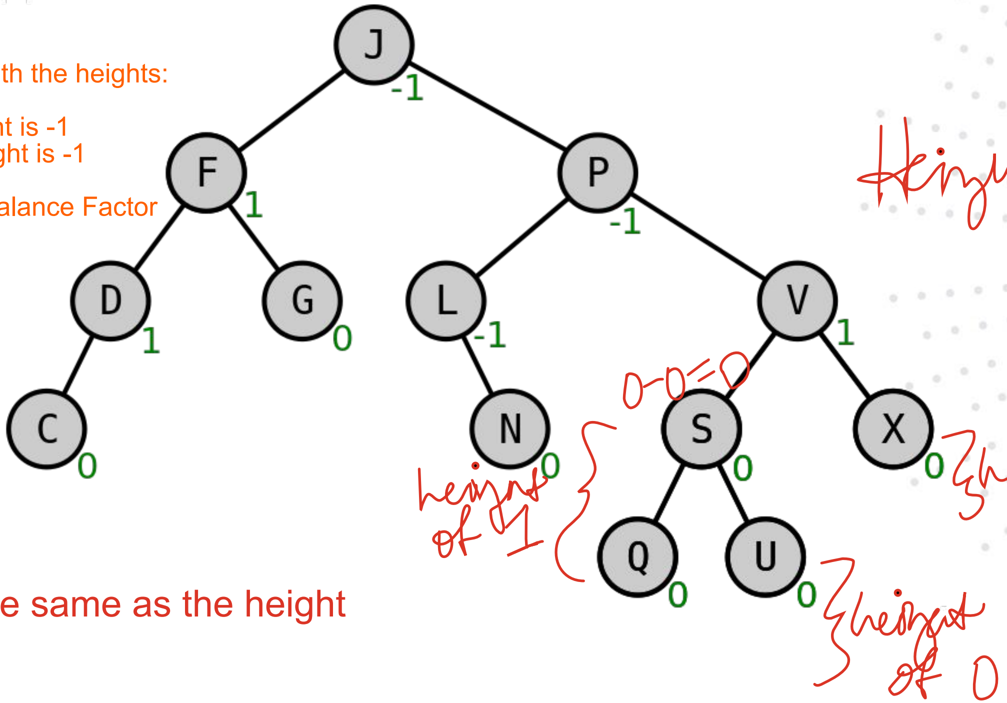


Balance Factor of Tree - Example

You calculate the balance factor with the heights:

*if you don't have a left node, height is -1
*if you don't have a right node, height is -1

Height of Left - Height of Right = Balance Factor



Height L - Height R

The balance factor is not the same as the height

Balance Violation Conditions

- What if condition violated after a node insertion or deletion?
  - We need to rebalance the tree
  - The entire tree will be rebalanced
- Violation cases at node k (deepest node)
  - Case 1: An insertion into left subtree of left child of k
  - Case 2: An insertion into right subtree of left child of k
  - Case 3: An insertion into left subtree of right child of k
  - Case 4: An insertion into right subtree of right child of k

Balance Condition

- The balance condition can be violated sometimes
  - Do something to fix it : rotations
  - After rotations, the balance of the whole tree is maintained
- Cases 1 and 4 equivalent
  - Single rotation to rebalance
- Cases 2 and 3 equivalent
  - Double rotation to rebalance

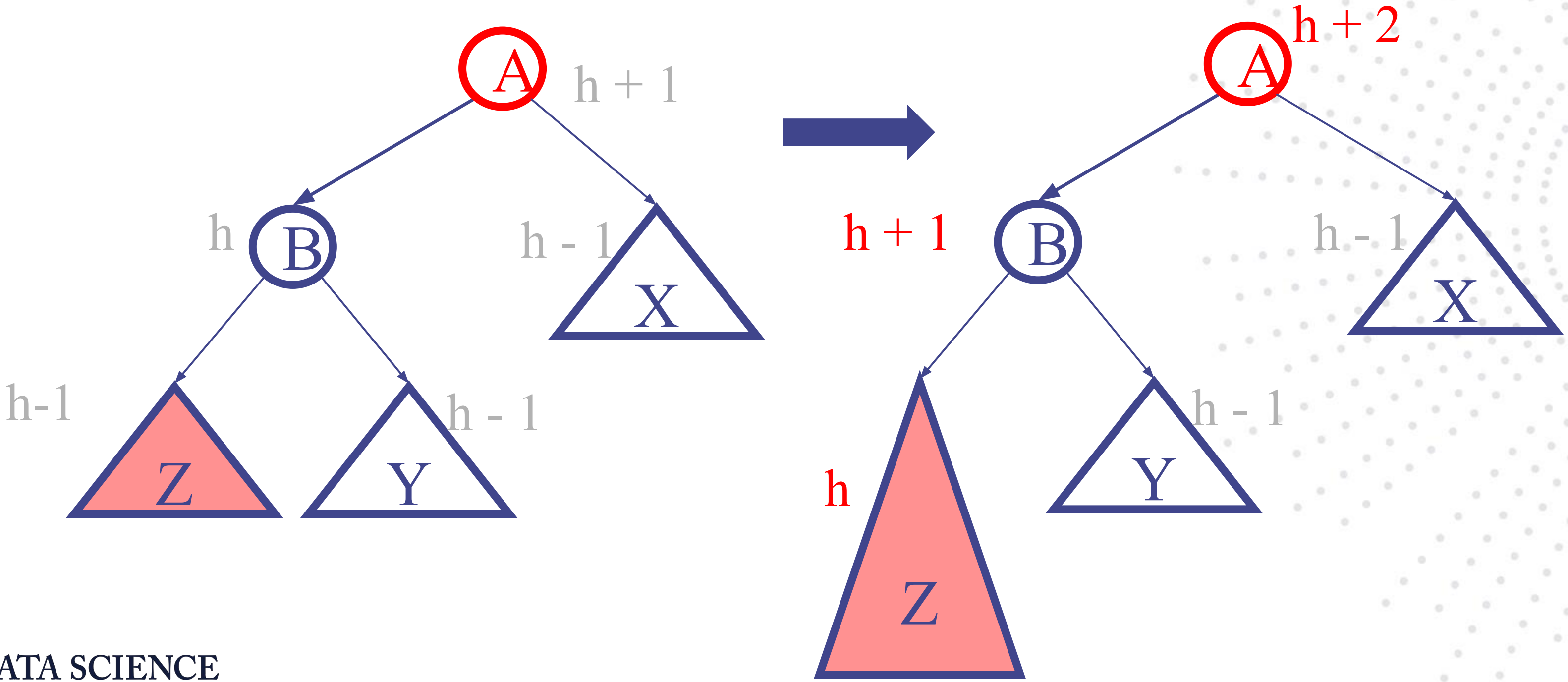
Insert into AVL Tree

- **Step 1**: Do the insertion just like with BST
- **Step 2**: Check the violation case and perform the tree rotation
- Assume node *thisNode* is the deepest node that violates the balance constraint and must be rebalanced.

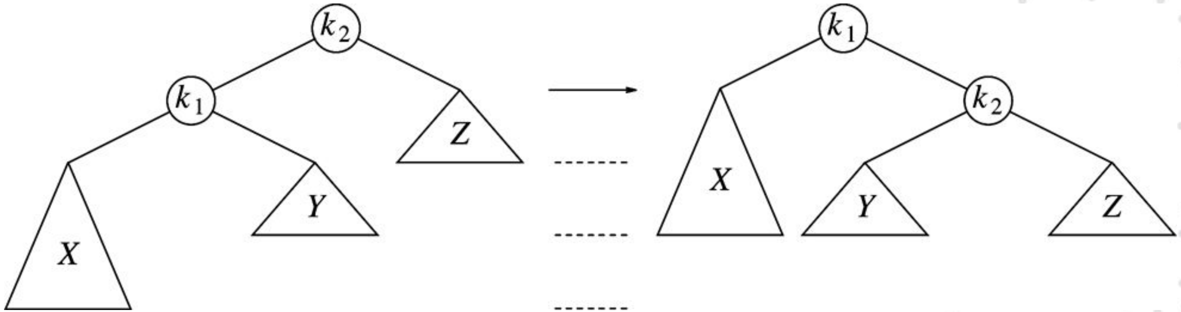
AVL Tree Visualization: https://visualgo.net/en/bst

Single Rotation (Case 1)

- Case 1: The left subtree of the left child of A violates the property

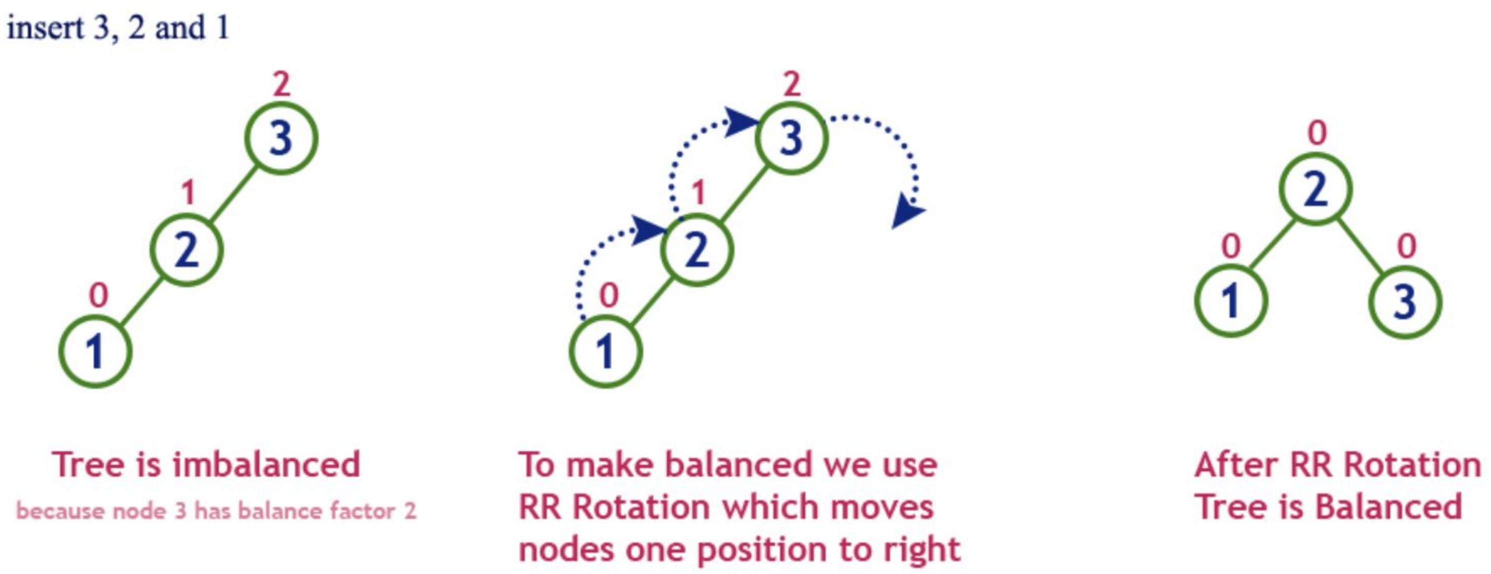


Single Rotation (Case 1)

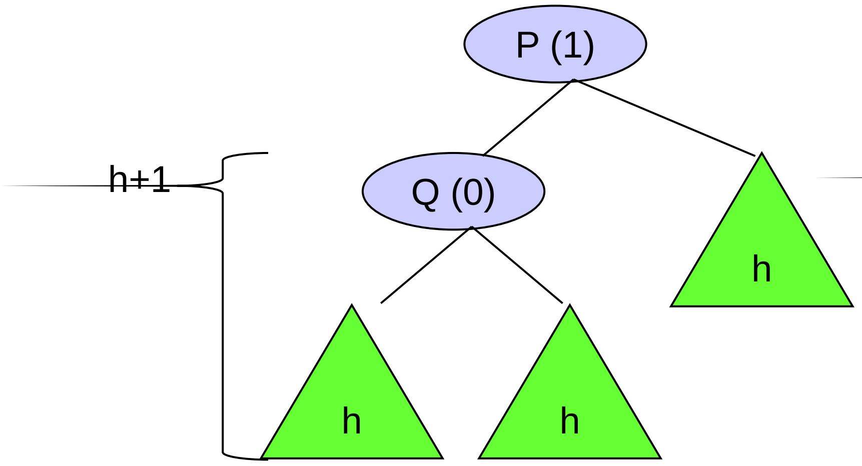


- Replace node k_2 by node k_1
- Set node k_2 to be right child of node k_1
- Set subtree Y to be left child of node k_2
- Case 4 is similar

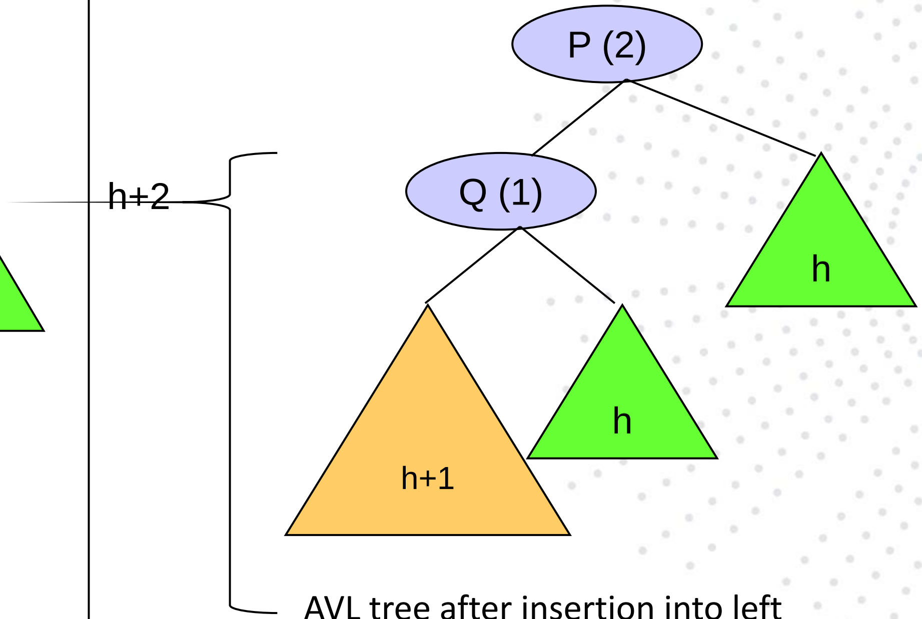
Single Rotation (Case 1)



Single Rotation (Case 1)

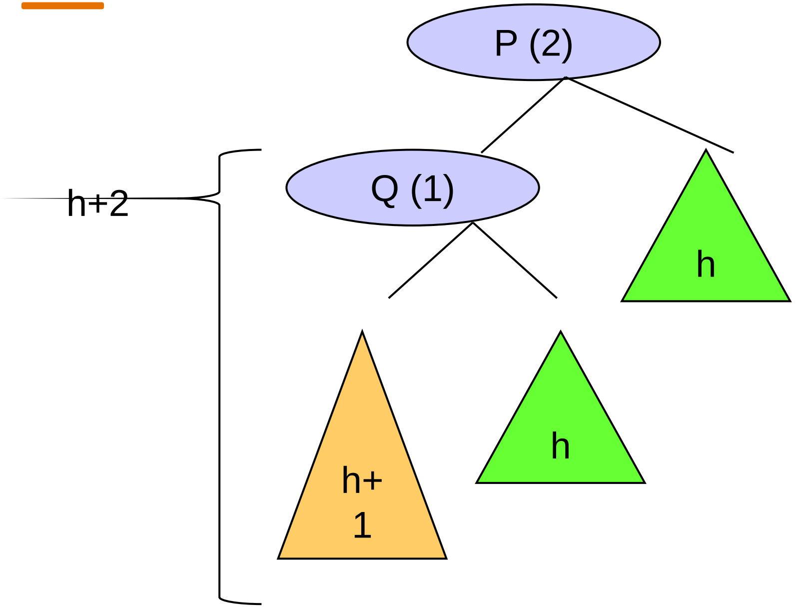


Initial AVL tree before insertion

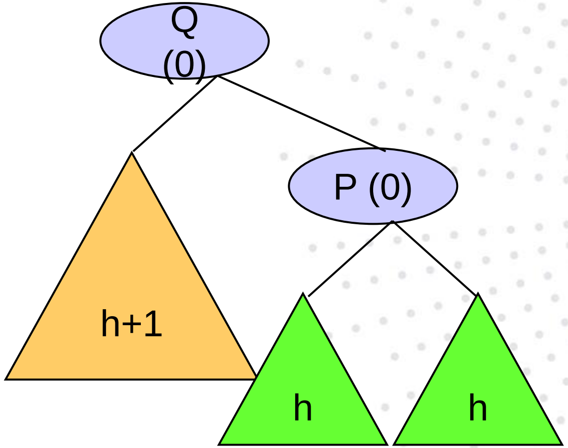


AVL tree after insertion into left subtree of left child Q

Single Rotation (Case 1)

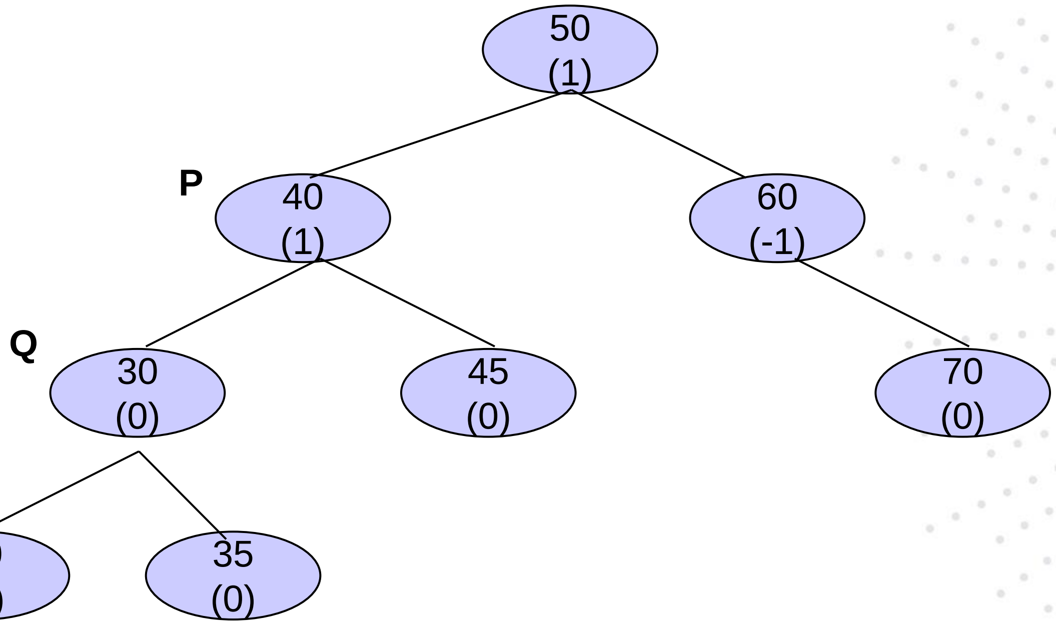


imbalance in an AVL tree caused by the insertion of a new node into the left subtree of the left child of P



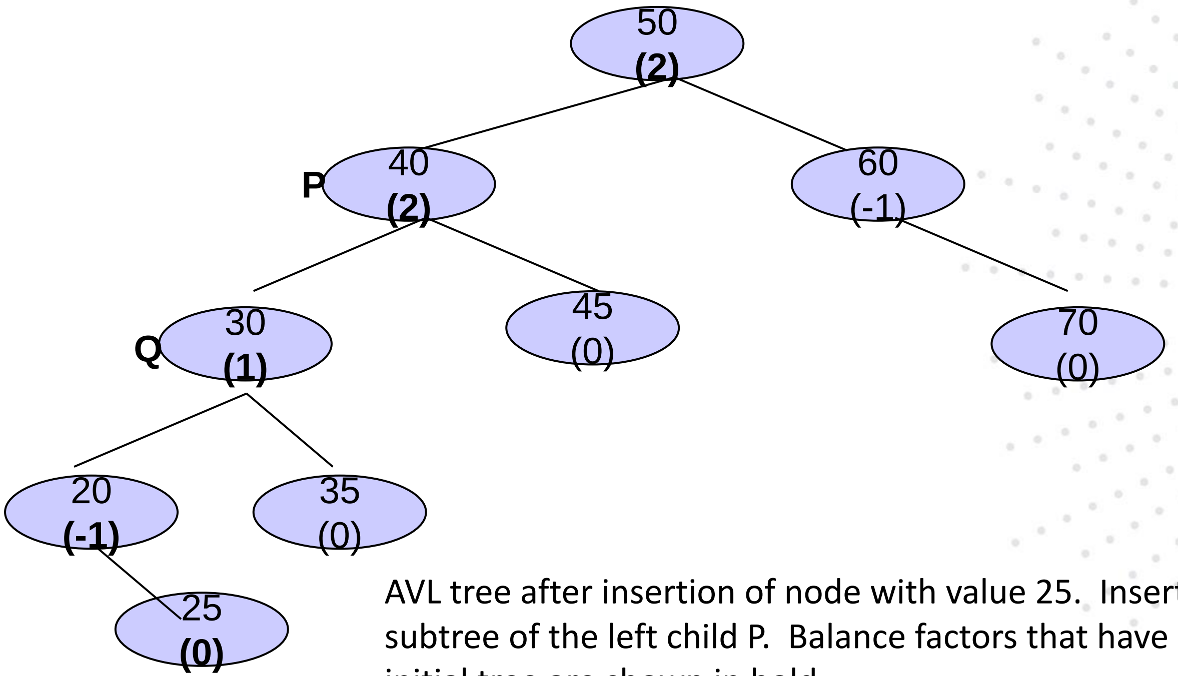
AVL tree after a single **right rotation** of Q about P. Balance has been restored to the tree.

Single Rotation (Case 1) - Example



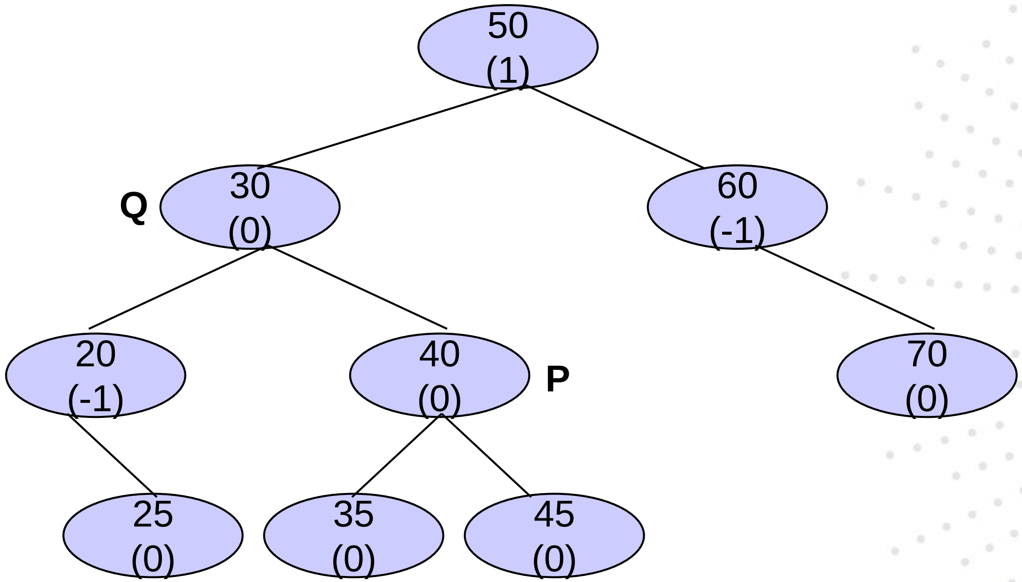
Initial AVL Tree

Single Rotation (Case 1) - Example



AVL tree after insertion of node with value 25. Insertion in the left subtree of the left child P. Balance factors that have changed from the initial tree are shown in bold.

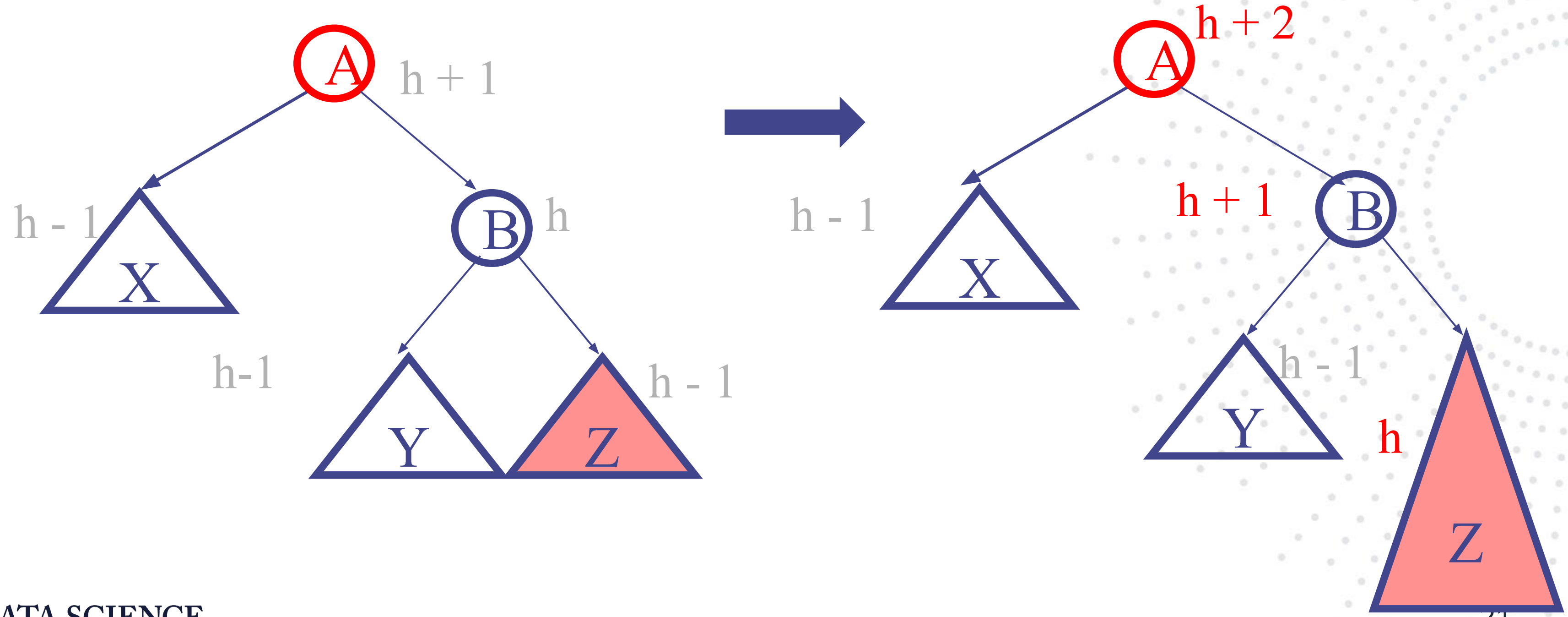
Single Rotation (Case 1) - Example



AVL tree after a single right rotation of 30 about 40.
Note that this balances the tree all the way to the root.

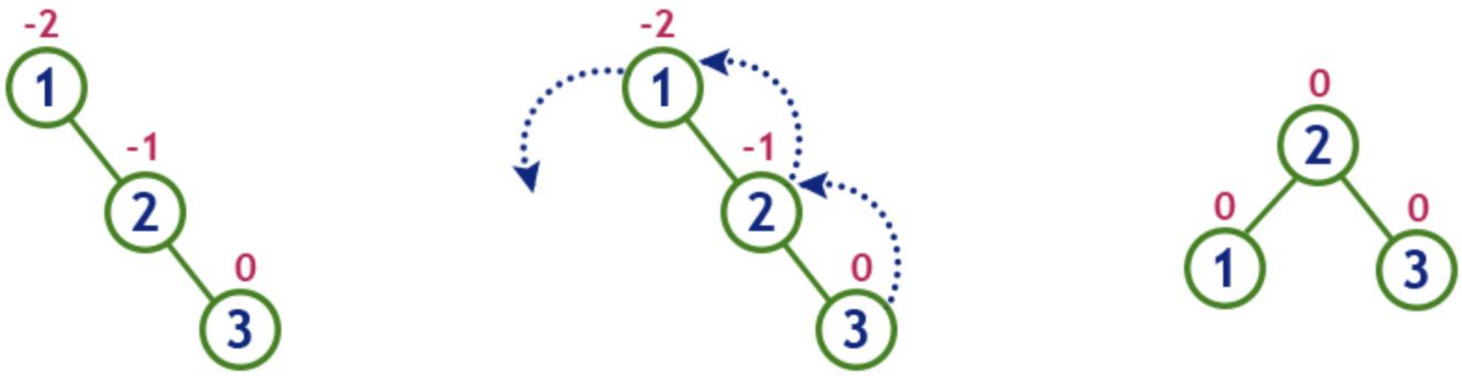
Single Rotation (Case 4)

- Case 2:The right subtree of the right child of X violates the property



Single Rotation (Case 4)

insert 1, 2 and 3

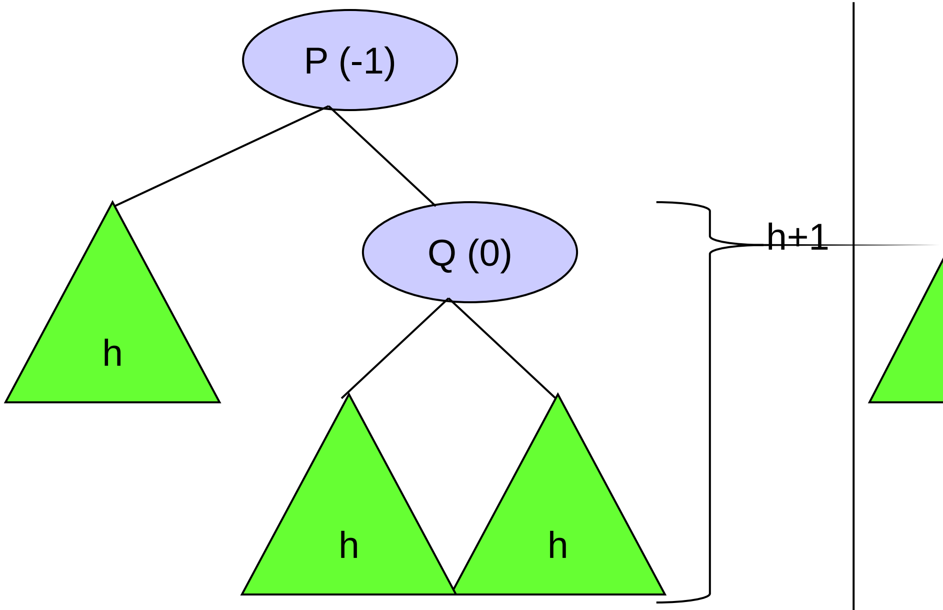


Tree is imbalanced

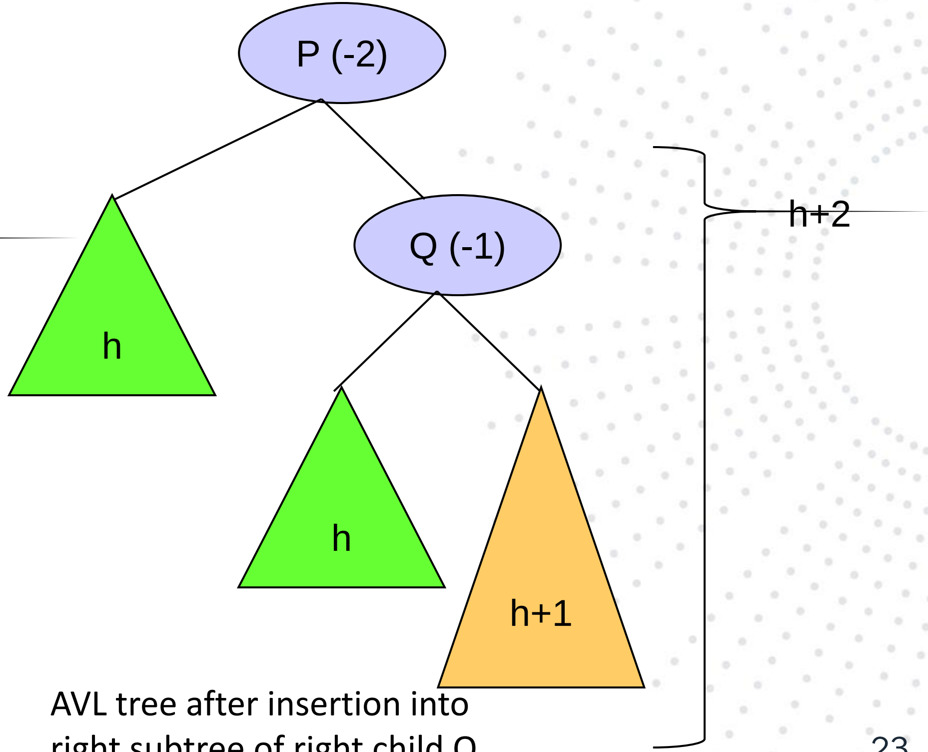
To make balanced we use LL Rotation which moves nodes one position to left

After LL Rotation Tree is Balanced

Single Rotation (Case 4)

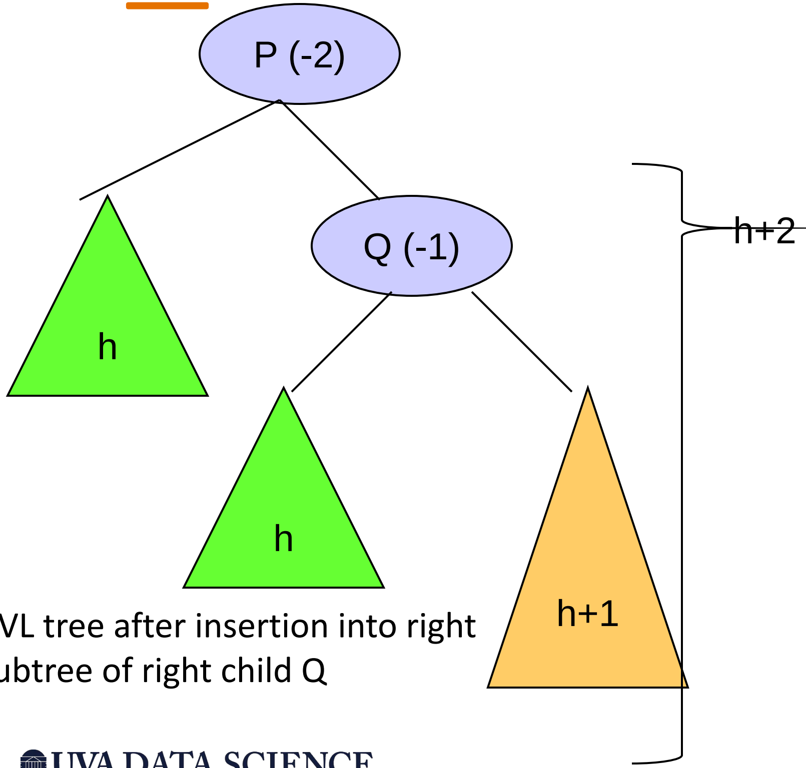


Initial AVL tree before insertion

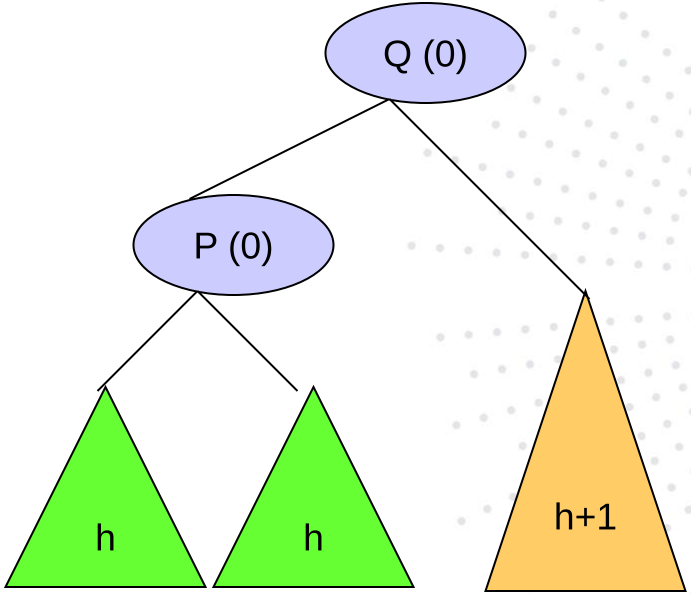


AVL tree after insertion into right subtree of right child Q

Single Rotation (Case 4)

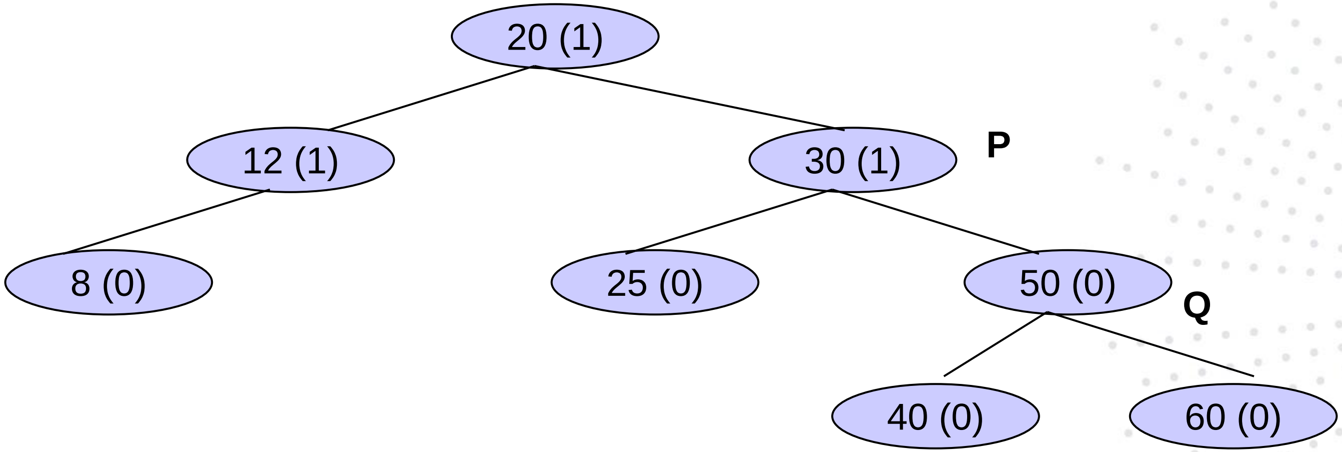


AVL tree after insertion into right subtree of right child Q



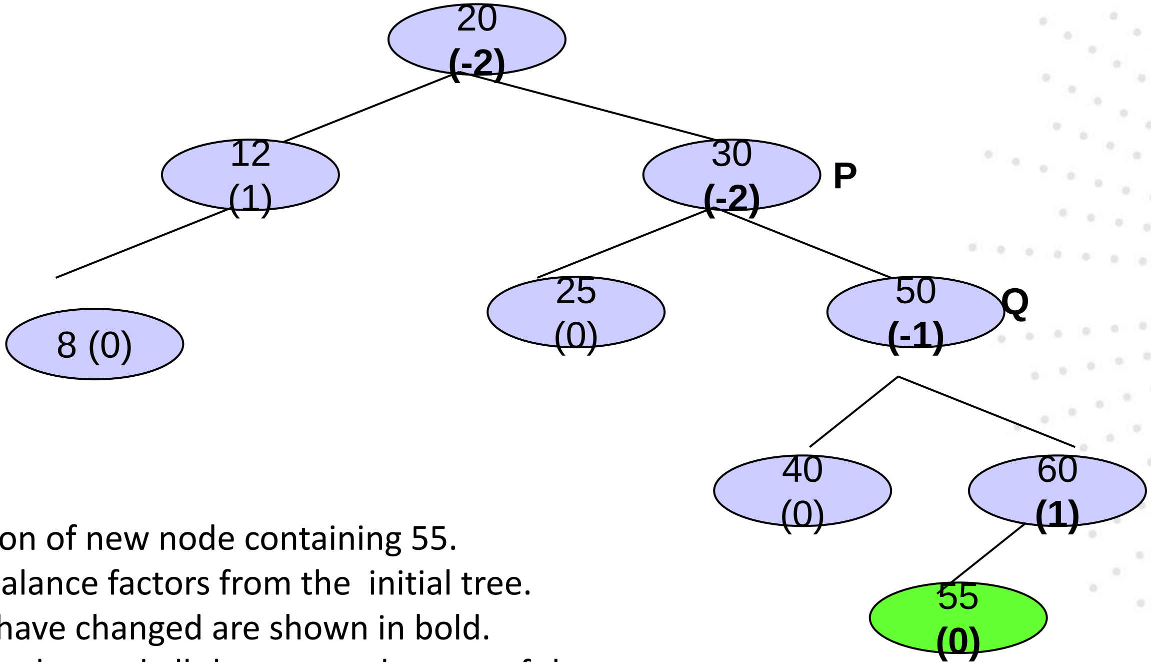
AVL tree after single **left rotation** of Q about P. Balance has been restored in the tree.

Single Rotation (Case 4) - Example



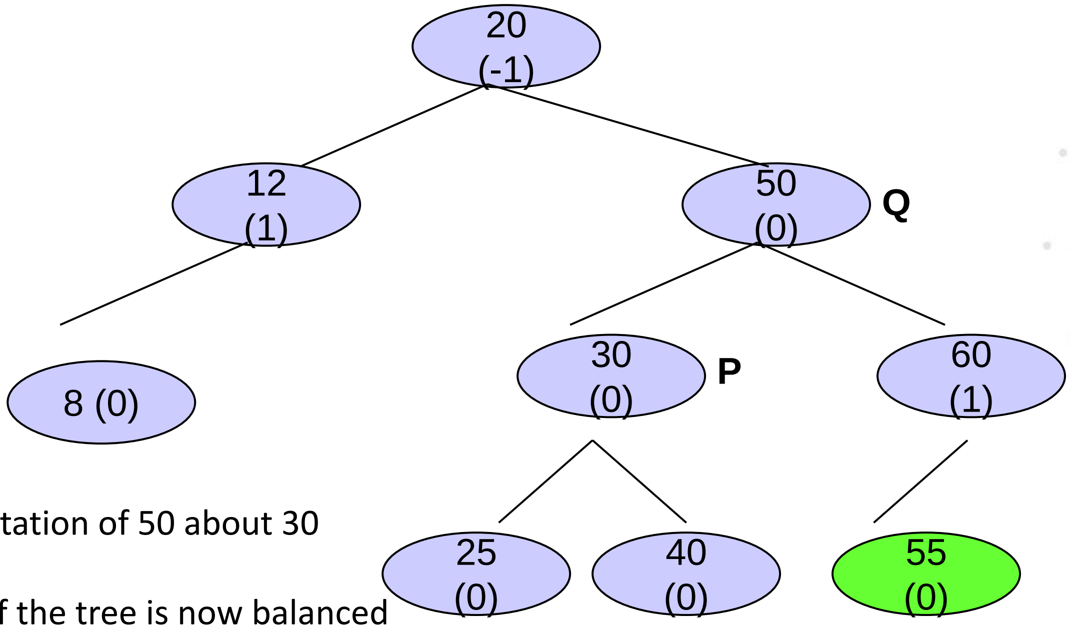
Initial AVL tree

Single Rotation (Case 4) - Example



AVL tree after insertion of new node containing 55.
Note the change in balance factors from the initial tree.
Balance factors that have changed are shown in bold.
Notice that they have changed all the way to the root of the tree.

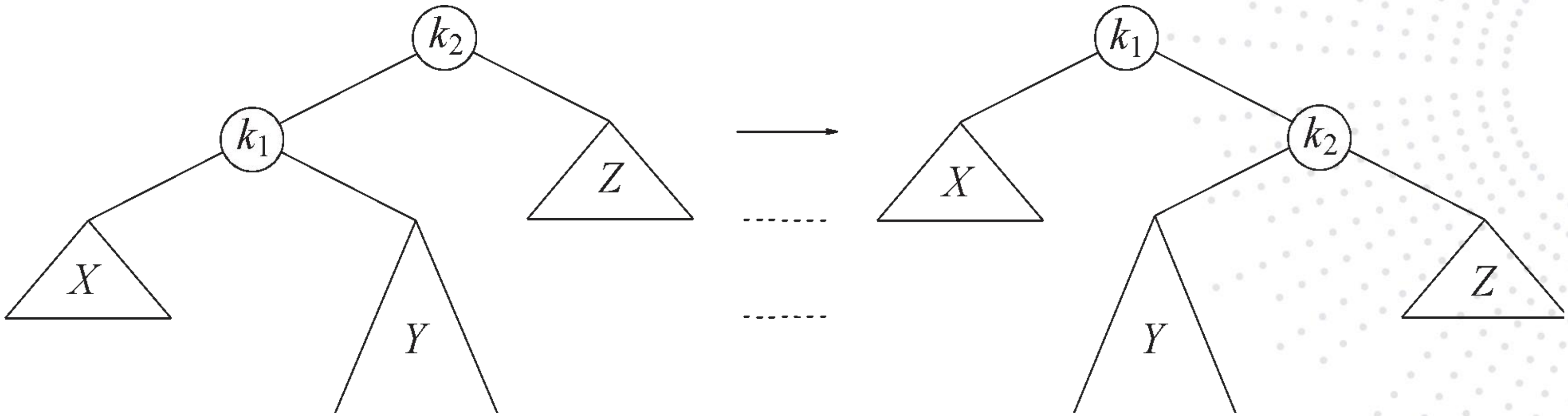
Single Rotation (Case 4) - Example



AVL tree after left rotation of 50 about 30
(Q about P)
Note that the root of the tree is now balanced

Case 2 and Case 3

- Single rotation can fail in case (2):
  - Insert into right subtree of left child or vice versa

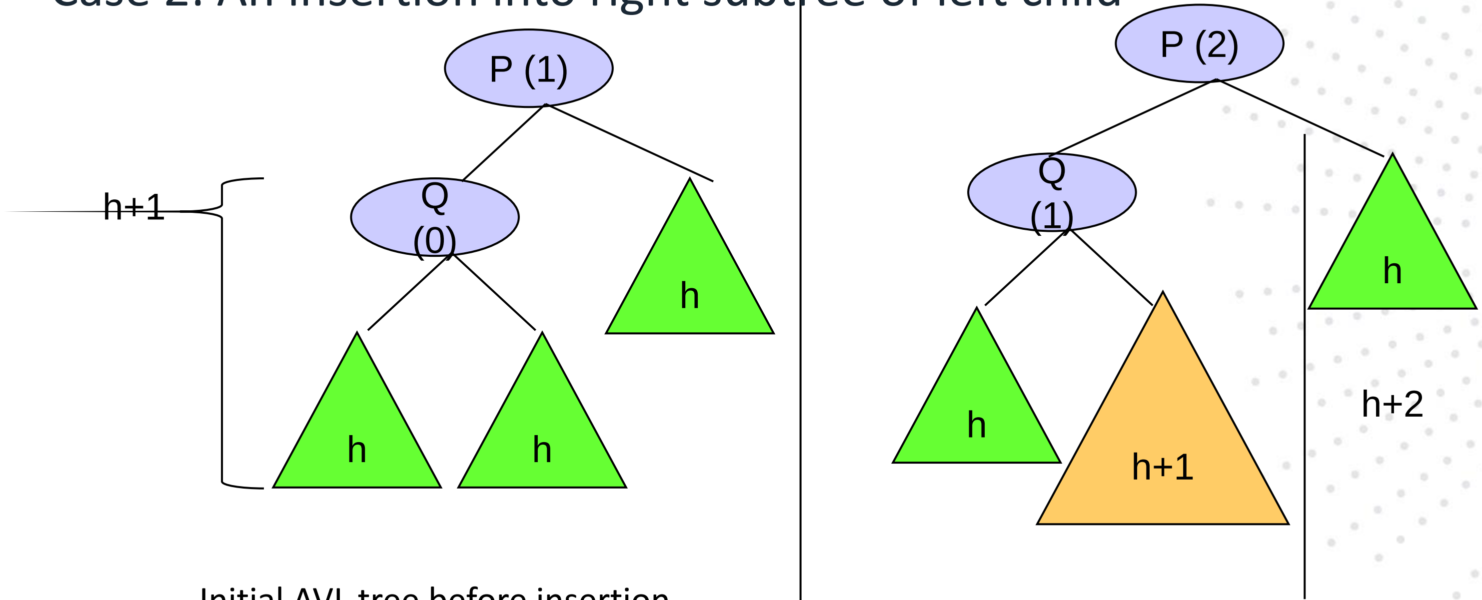


Case 2 and Case 3

- For case 2 and 3
- After single rotation, k_1 still not balanced
- Double rotations needed for case 2 and case 3

Double Rotation (Case 2)

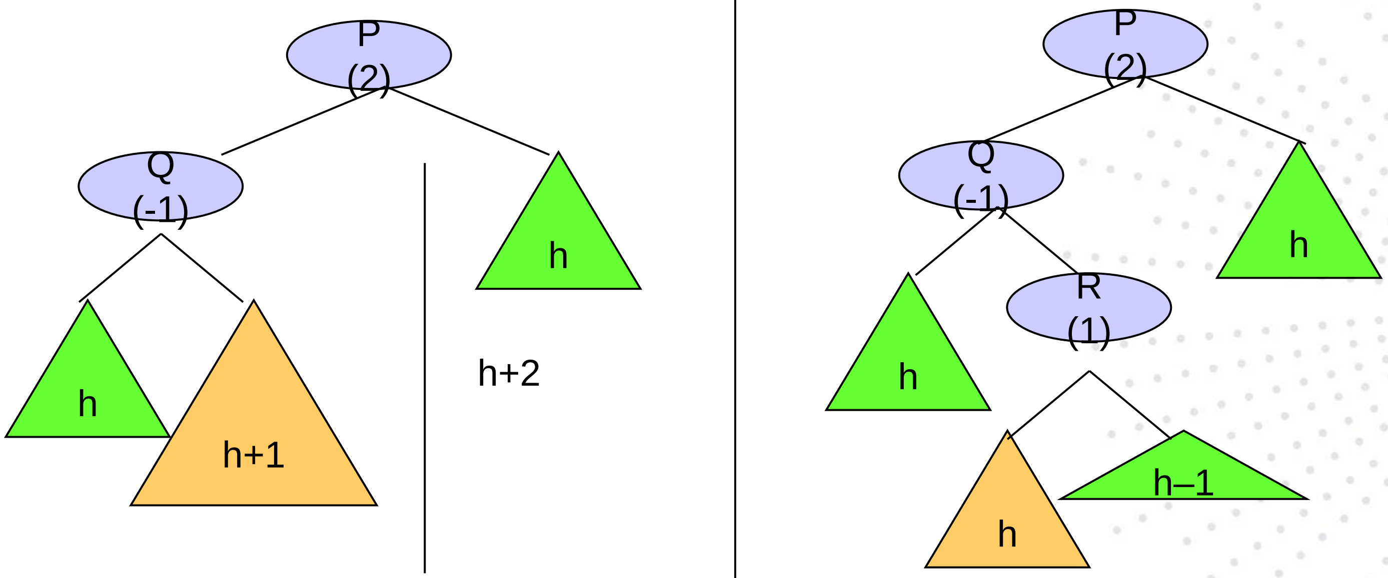
- Case 2: An insertion into right subtree of left child



Initial AVL tree before insertion

AVL tree after insertion into right subtree of left child Q

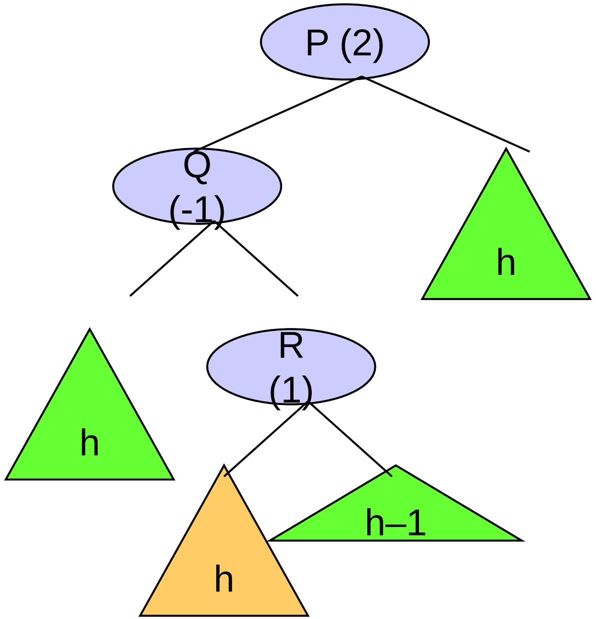
Double Rotation (Case 2)



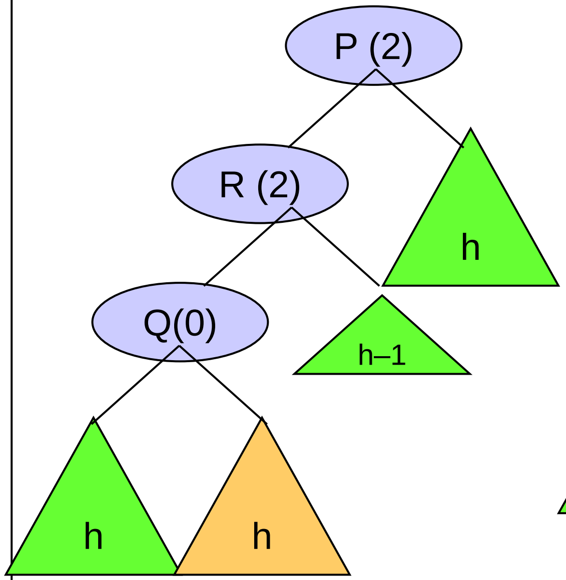
AVL tree after insertion into right subtree of left child Q

Assuming insertion was in the left subtree of the right child of Q (call this node R)

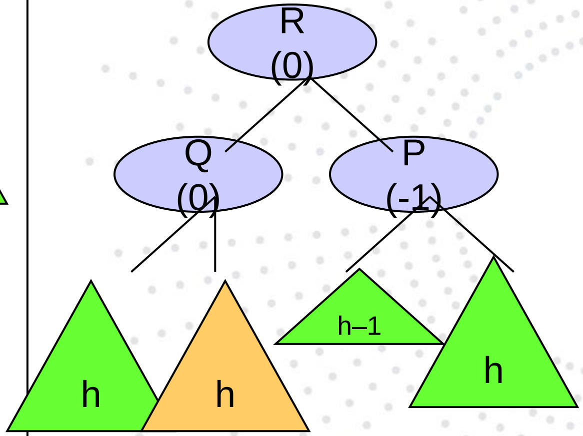
Double Rotation (Case 2)



insertion caused
imbalanced tree

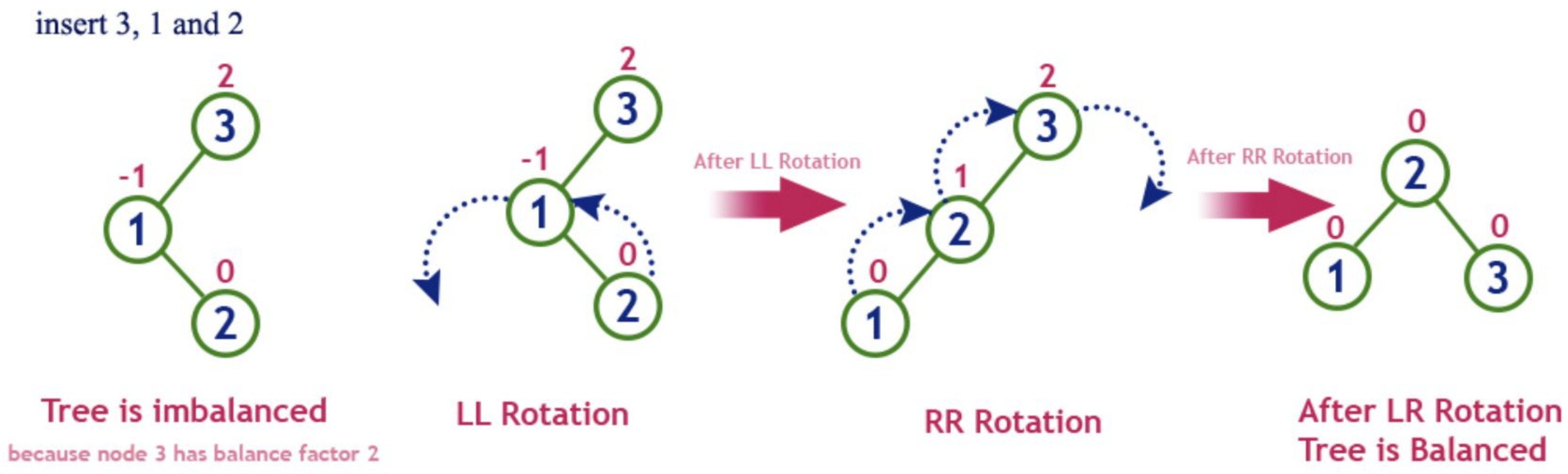


After left rotation of R
about Q. Note tree is still
imbalanced, now both at
R and P.

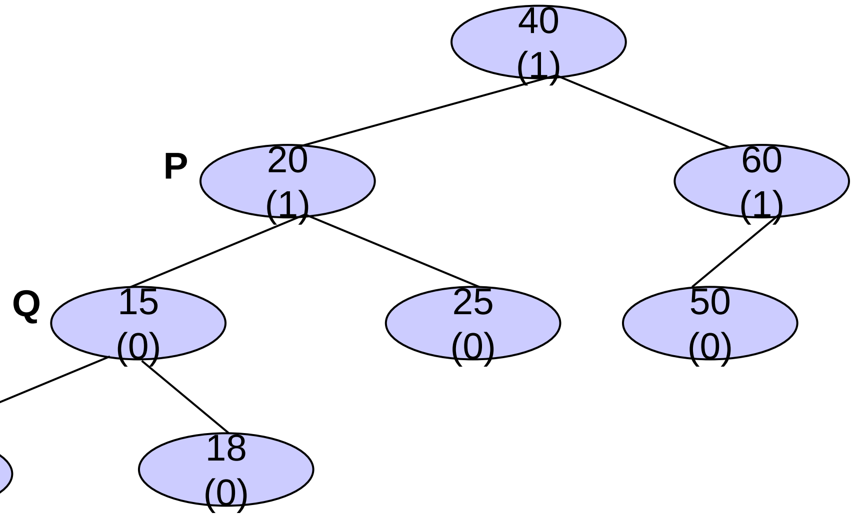


After right rotation of Q
about R. Note tree is now
balanced.

Double Rotation (Case 2)

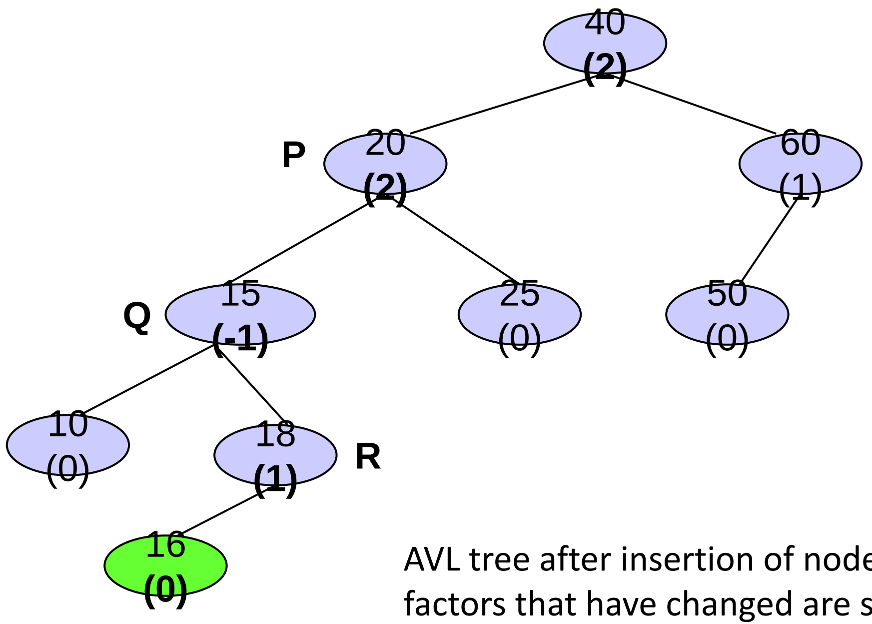


Double Rotation (Case 2) - Example



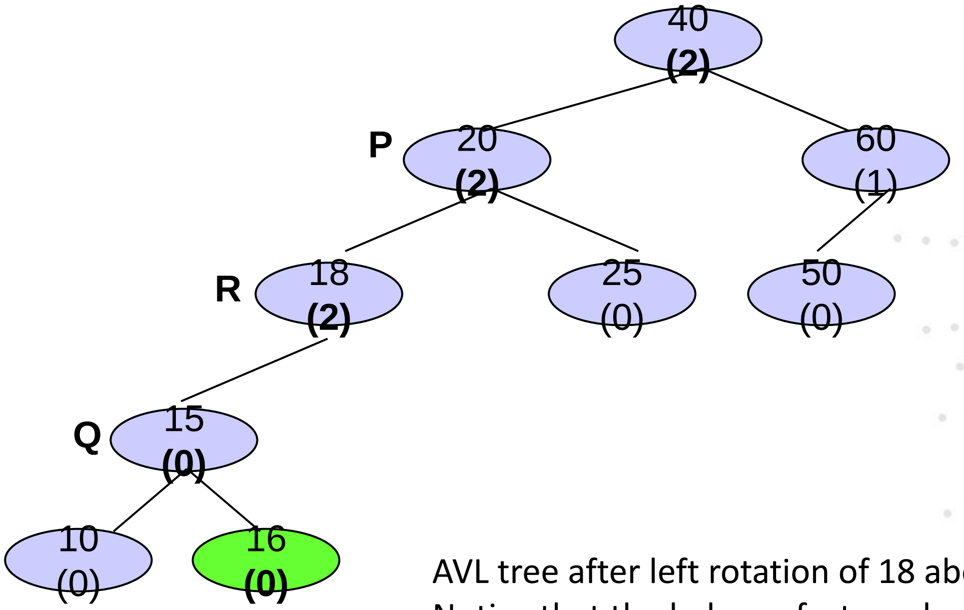
Initial AVL Tree

Double Rotation (Case 2) - Example



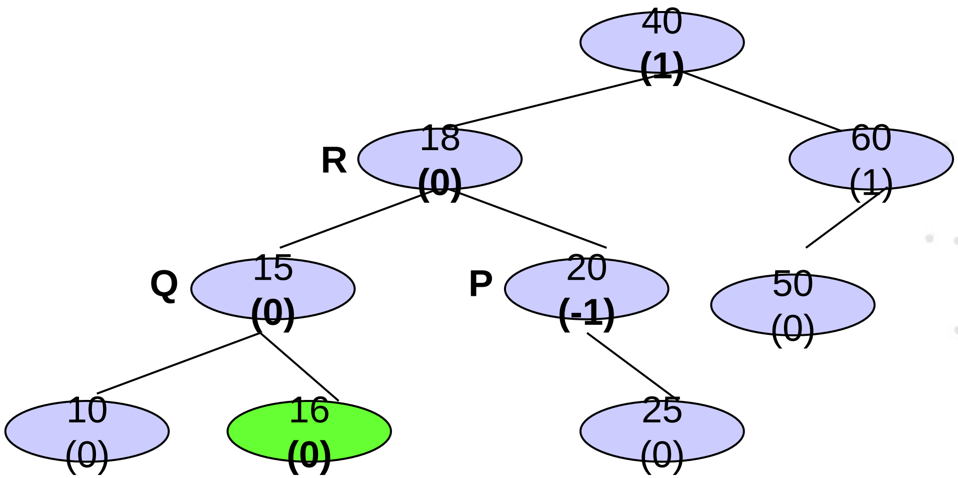
AVL tree after insertion of node with value 16. Balance factors that have changed are shown in bold.

Double Rotation (Case 2) - Example



AVL tree after left rotation of 18 about 15 (R about Q). Notice that the balance factors clearly indicate the tree is not balanced.

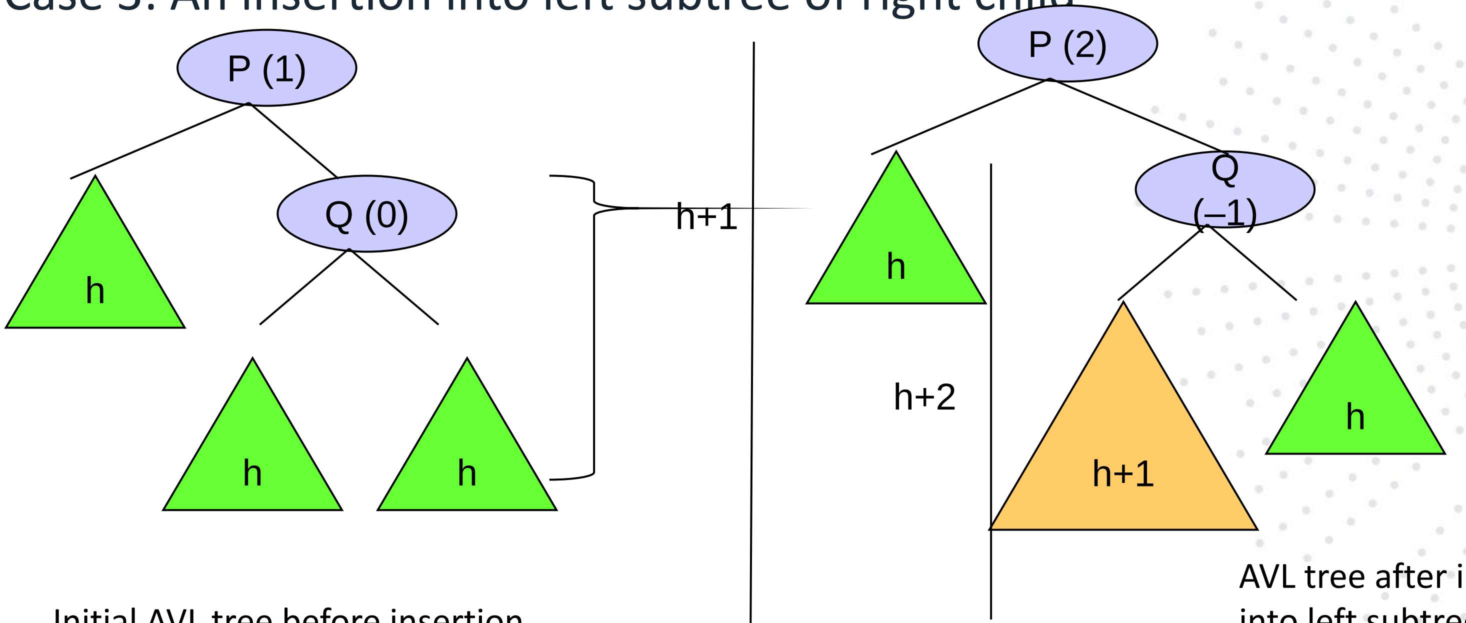
Double Rotation (Case 2) - Example



AVL tree after right rotation of 18 about 10 (R about P). Tree is now balanced.

Double Rotation (Case 3)

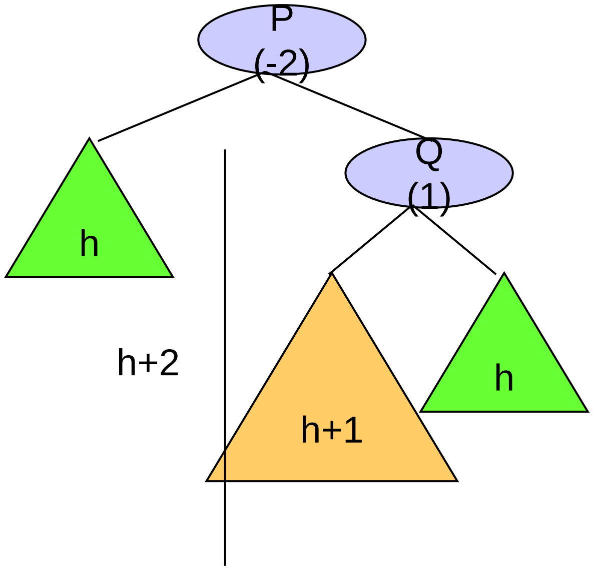
- Case 3: An insertion into left subtree of right child



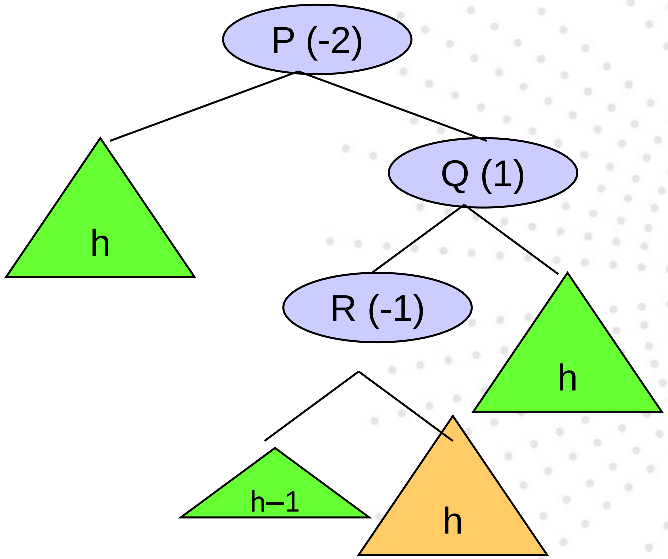
Initial AVL tree before insertion

AVL tree after insertion into left subtree of right child Q

Double Rotation (Case 3)

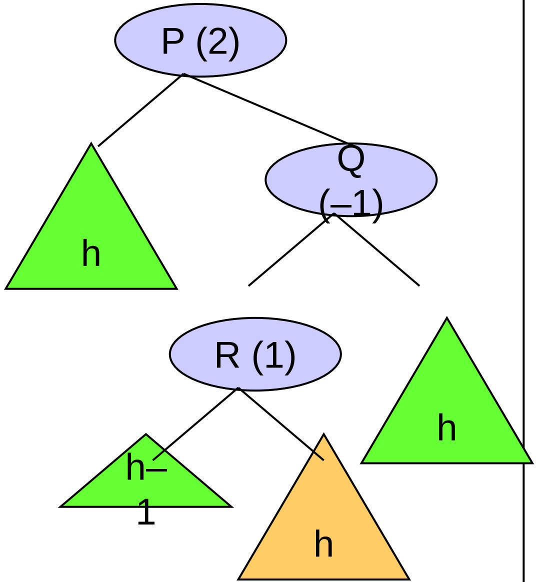


AVL tree after insertion into left subtree of right child Q

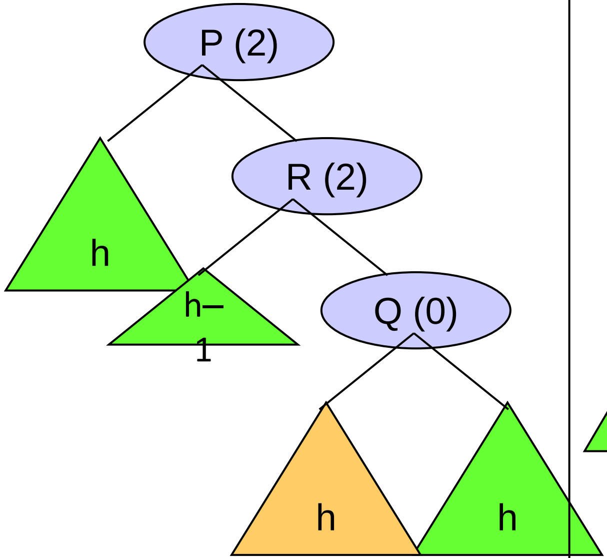


Assuming insertion was in the right subtree of the left child of Q (call this node R)

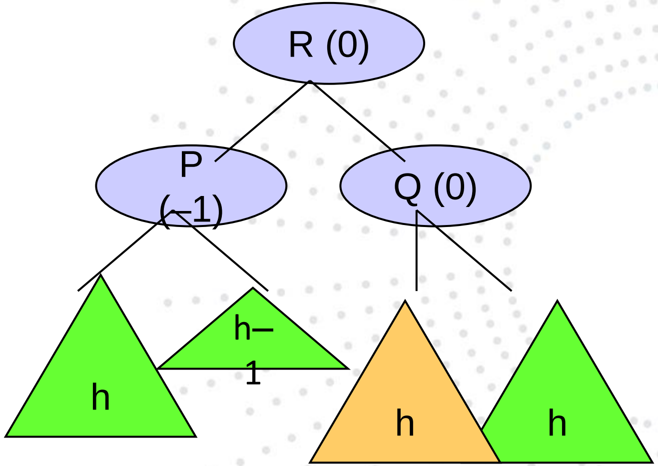
Double Rotation (Case 3)



imbalanced tree

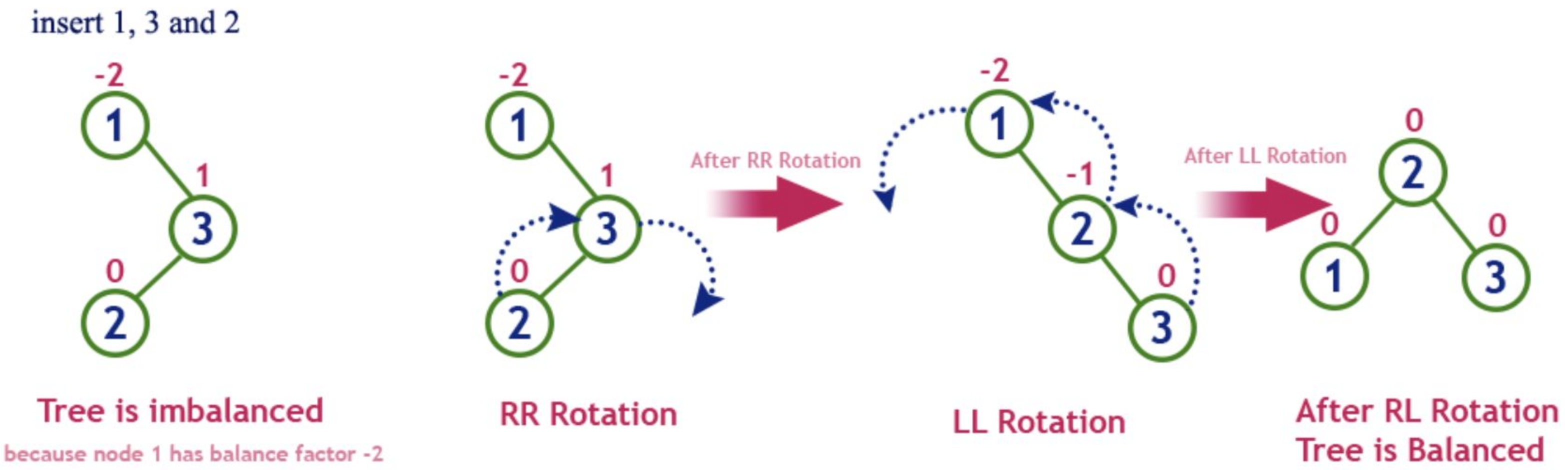


After right rotation of R about Q

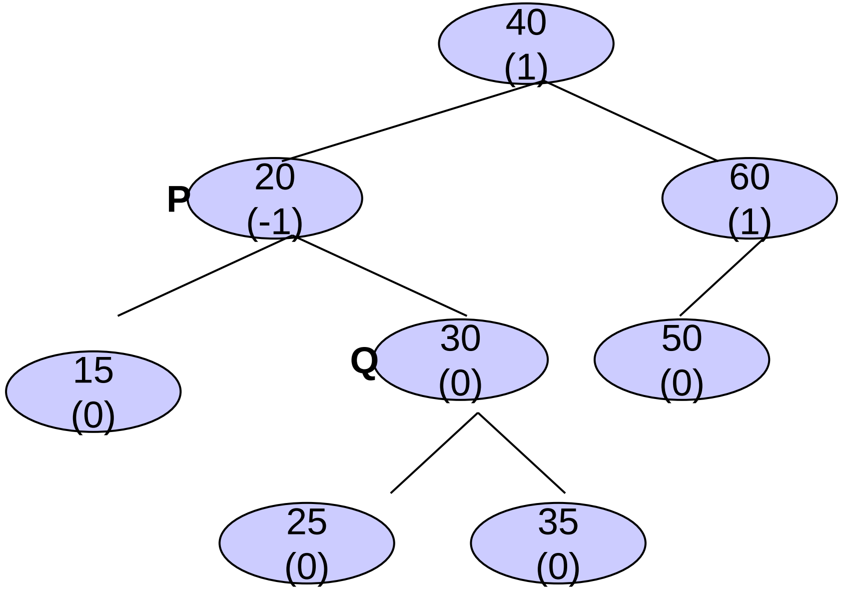


After left rotation of R about P. The complete set of rotations has been a right followed by a left or a RL double rotation.

Double Rotation (Case 3)

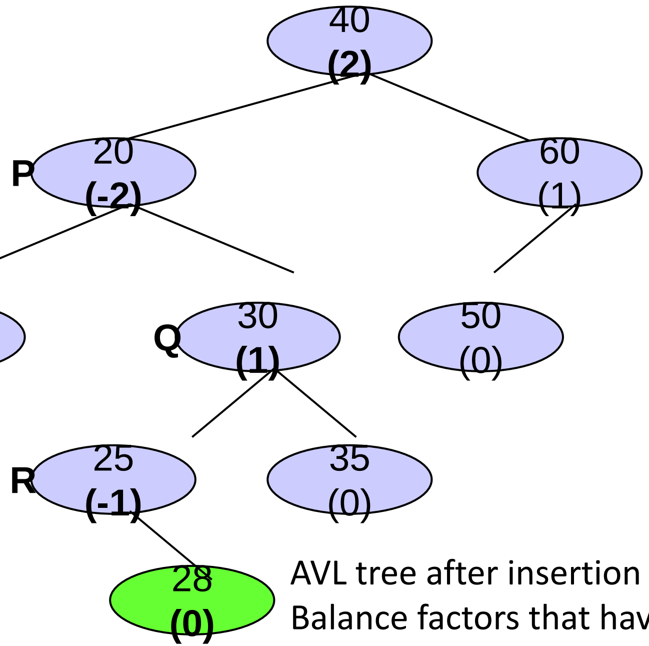


Double Rotation (Case 3) - Example



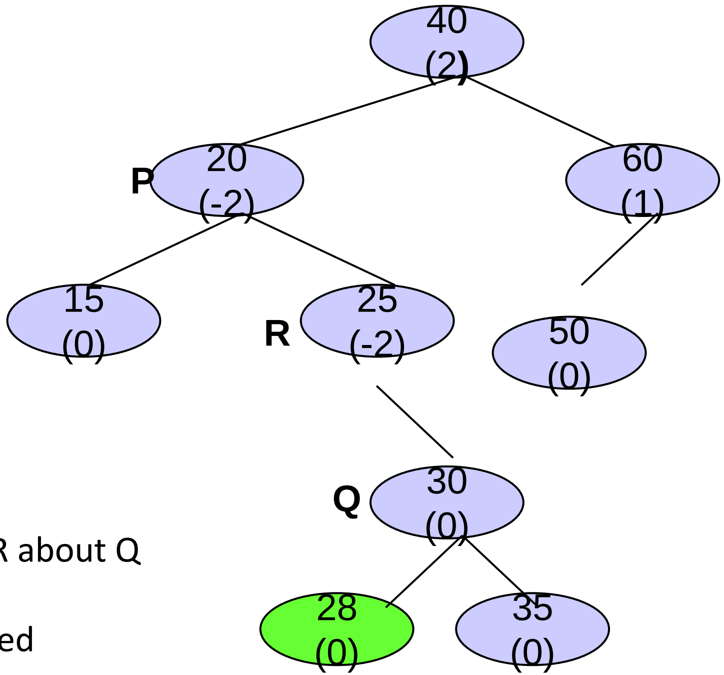
Initial AVL Tree

Double Rotation (Case 3) - Example



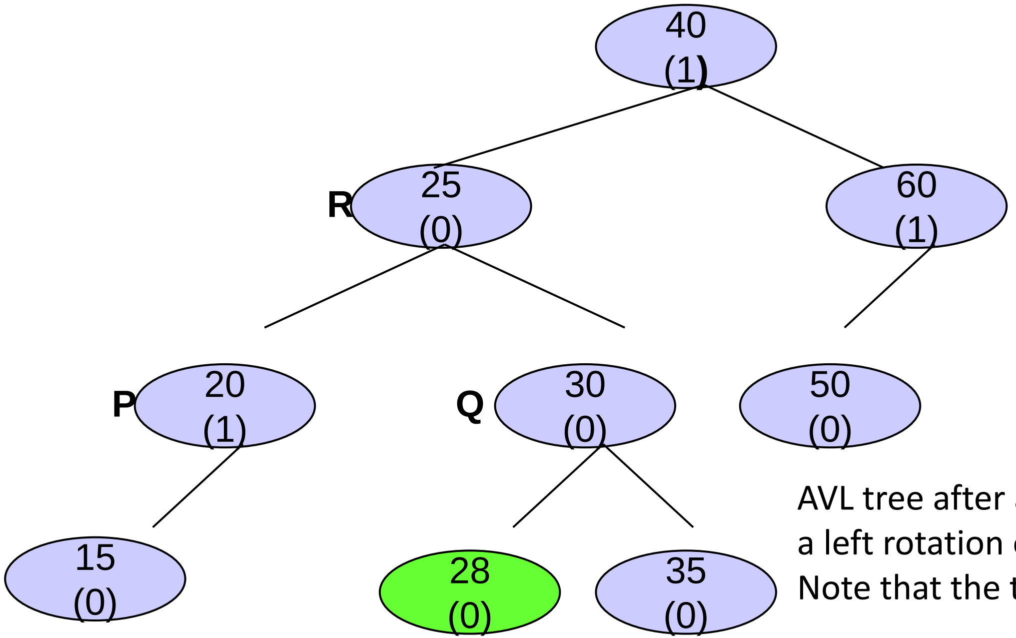
AVL tree after insertion of node with value 28
Balance factors that have changed from the initial tree are shown in bold

Double Rotation (Case 3) - Example



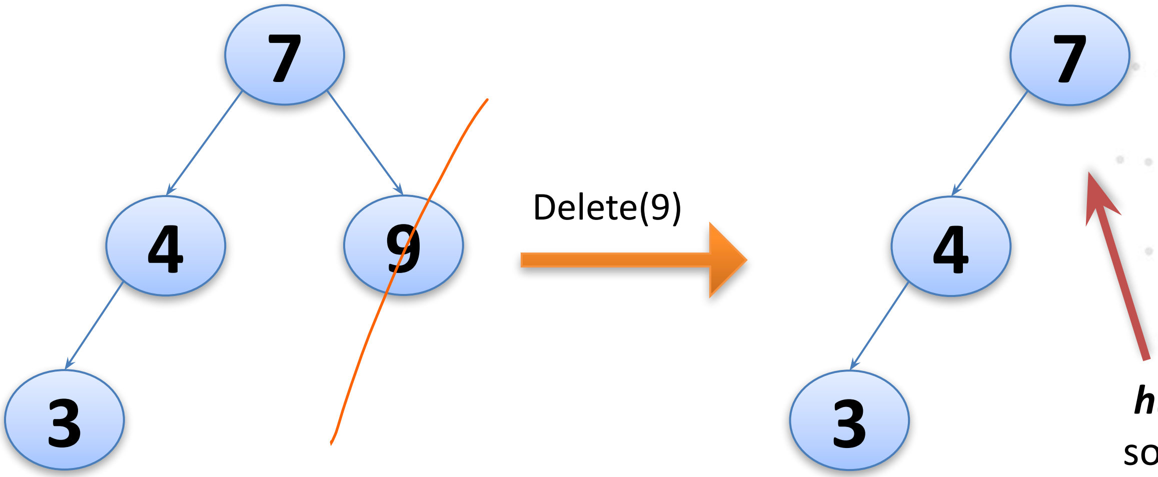
AVL tree after right rotation of R about Q (25 about 30).
Note that the tree is not balanced after this first rotation.

Double Rotation (Case 3) - Example



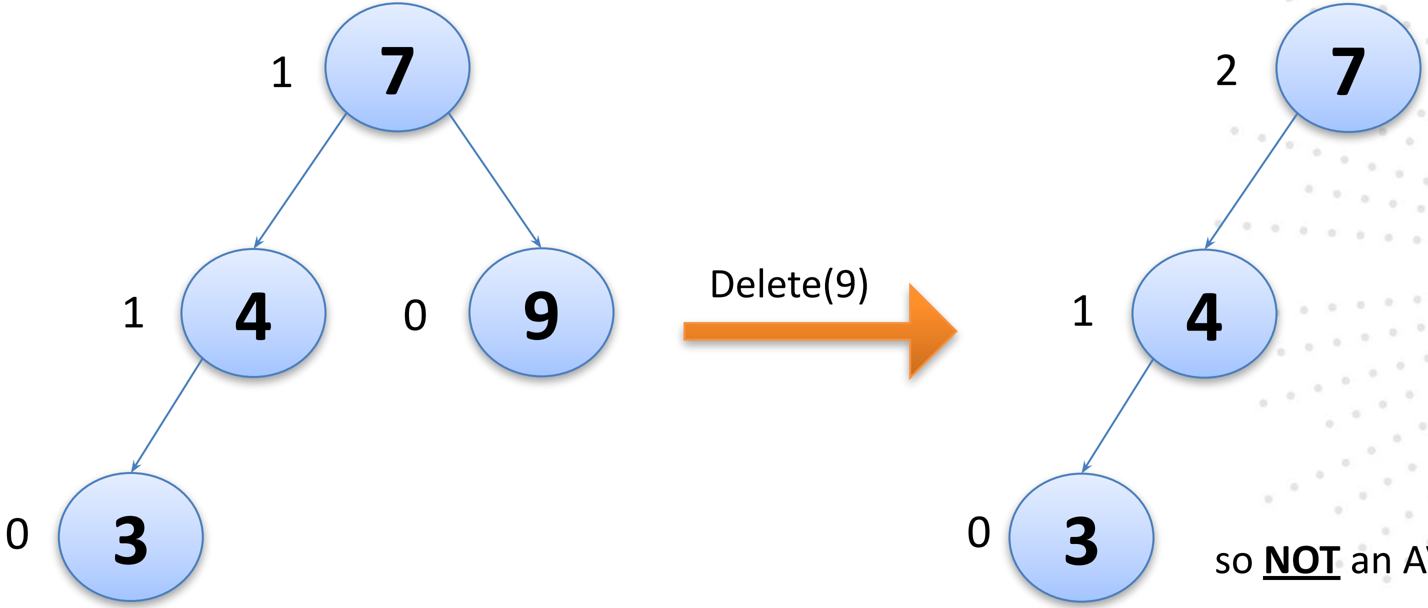
AVL tree after a second rotation, this one a left rotation of R about P (25 about 20). Note that the tree is now balanced.

Delete node from tree



h(left) > h(right)+1
so **<u>NOT</u>** an AVL tree!

Delete node from tree



so **<u>NOT</u>** an AVL tree!

Delete node from tree

- We are starting to see what our delete algorithm must look like.
- **Goal:** if tree is AVL before Delete, then tree is AVL after Delete.
- **Step 1:** do BST delete.
  - This maintains the *BST property*,
    but can BREAK the *balance factors of ancestors!*
- **Step 2:** fix the balance constraint.
  - Do something that maintains the BST property,
    but fixes any balance factors that are < -1 or > 1.

Motivation

- What bad values can bf(x) take on?
  - Delete can reduce a subtree's height by 1.
  - So, it might increase or decrease h(x.right) – h(x.left) by 1.
  - So, bf(x) might increase or decrease by 1.
  - This means:
    - if bf(x) = 1 before Delete, it might become 2. **BAD.**
    - If bf(x) = -1 before Delete, it might become -2. **BAD.**
    - If bf(x) = 0 before Delete, then it is still -1, 0 or 1. **OK.**

Summary

- An AVL Tree can perform the following operations in worst-case time O(log n) each:
  - Insert
  - Delete
  - Find
  - Find Min, Find Max
  - Find Successor, Find Predecessor