AVL Trees
(Just for fun)

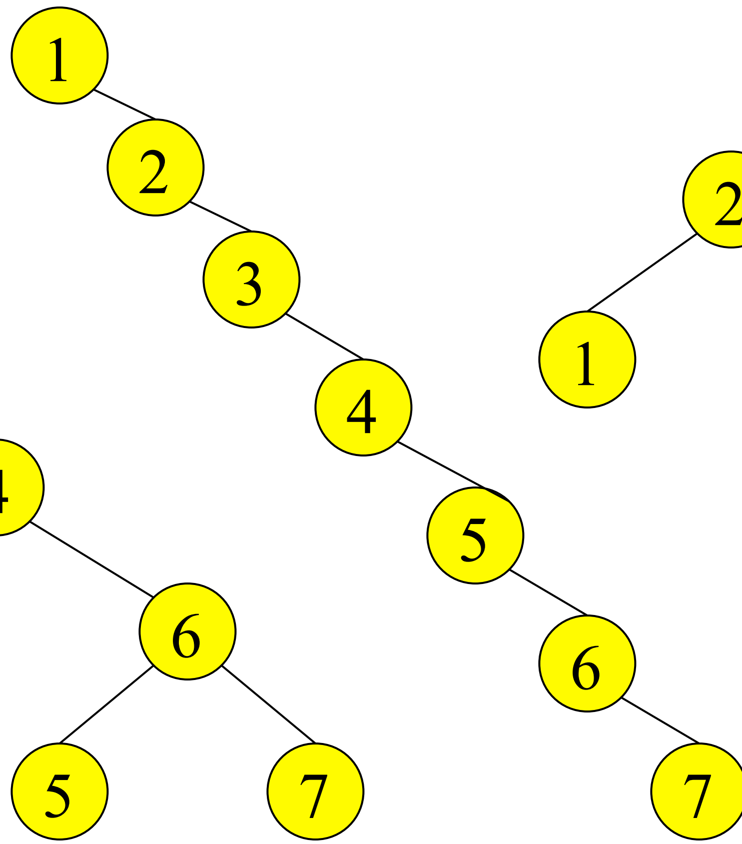
Binary Search Tree - Best Time

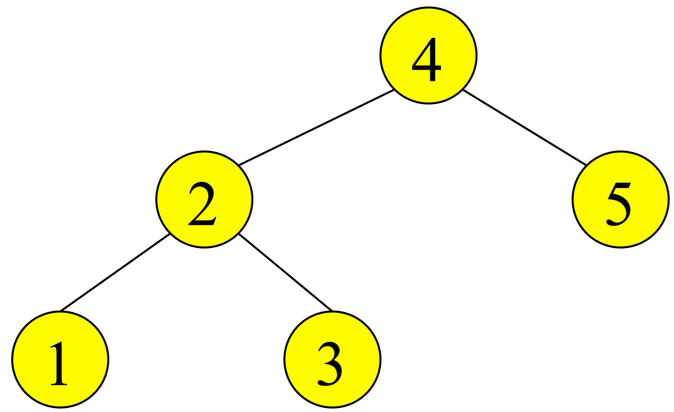
- All BST operations are O(d), where d is tree depth
- minimum d is $d = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - › What is the best case tree?
  - › What is the worst case tree?
- So, best case running time of BST operations is O(log N)

Binary Search Tree - Worst Time

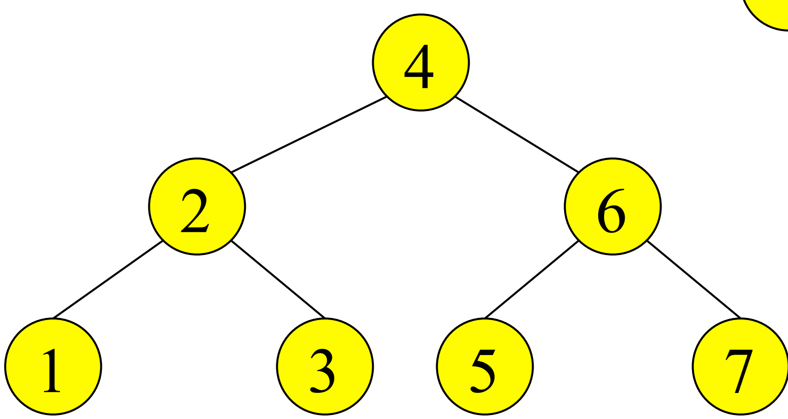
- Worst case running time is O(N)
  - › What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - › Problem: Lack of "balance":
    - compare depths of left and right subtree
  - › Unbalanced degenerate tree

Balanced and unbalanced BST





Is this “balanced”?



Approaches to balancing trees

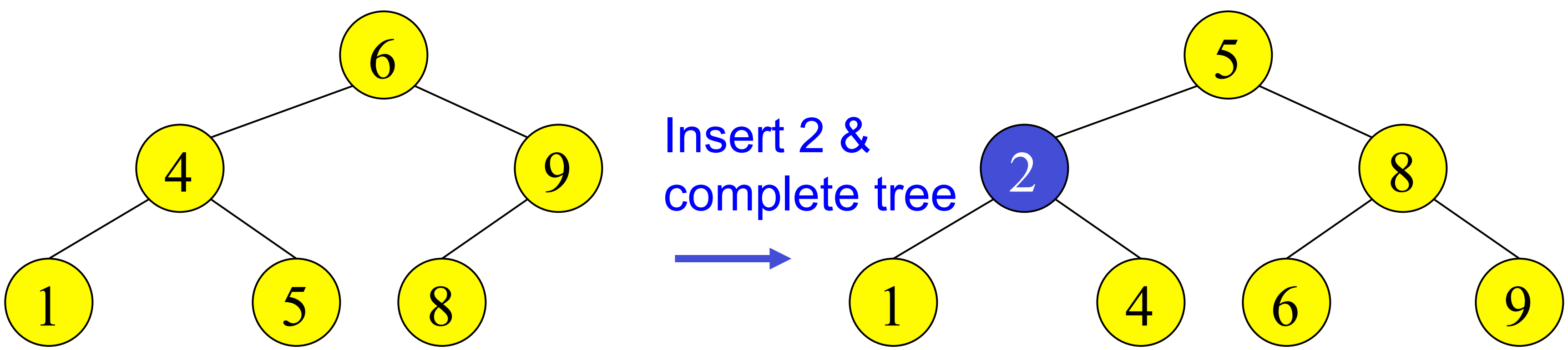
- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

Perfect Balance

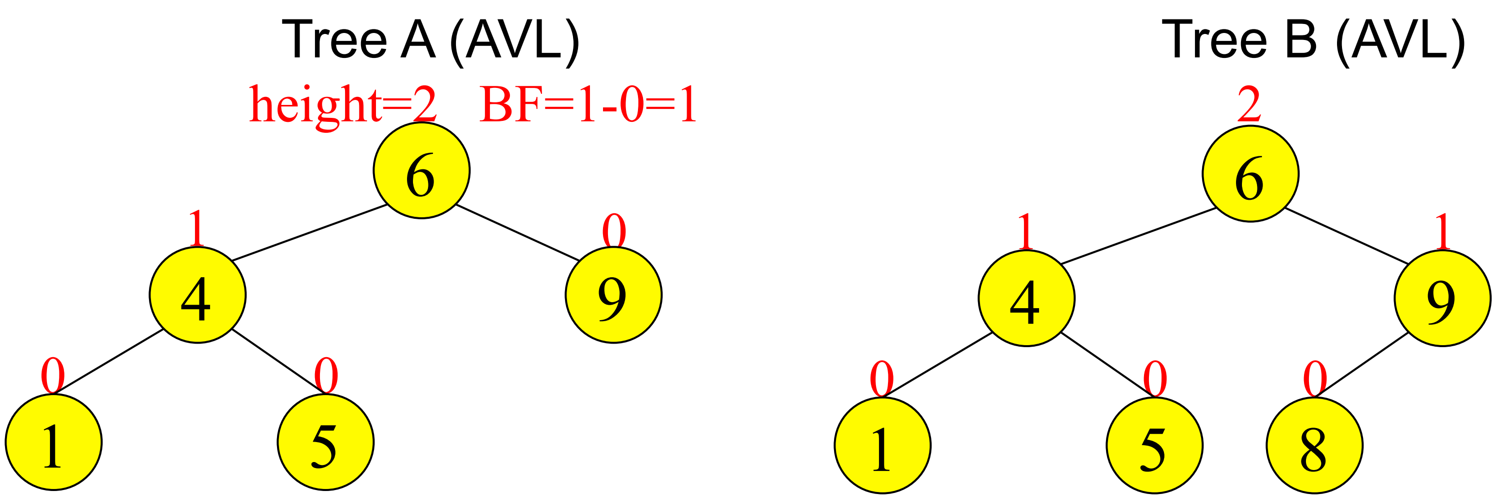
- Want a complete tree after every operation
  - › tree is full except possibly in the lower right
- This is expensive
  - › For example, insert 2 in the tree on the left and then rebuild as a complete tree



AVL - Good but not Perfect Balance

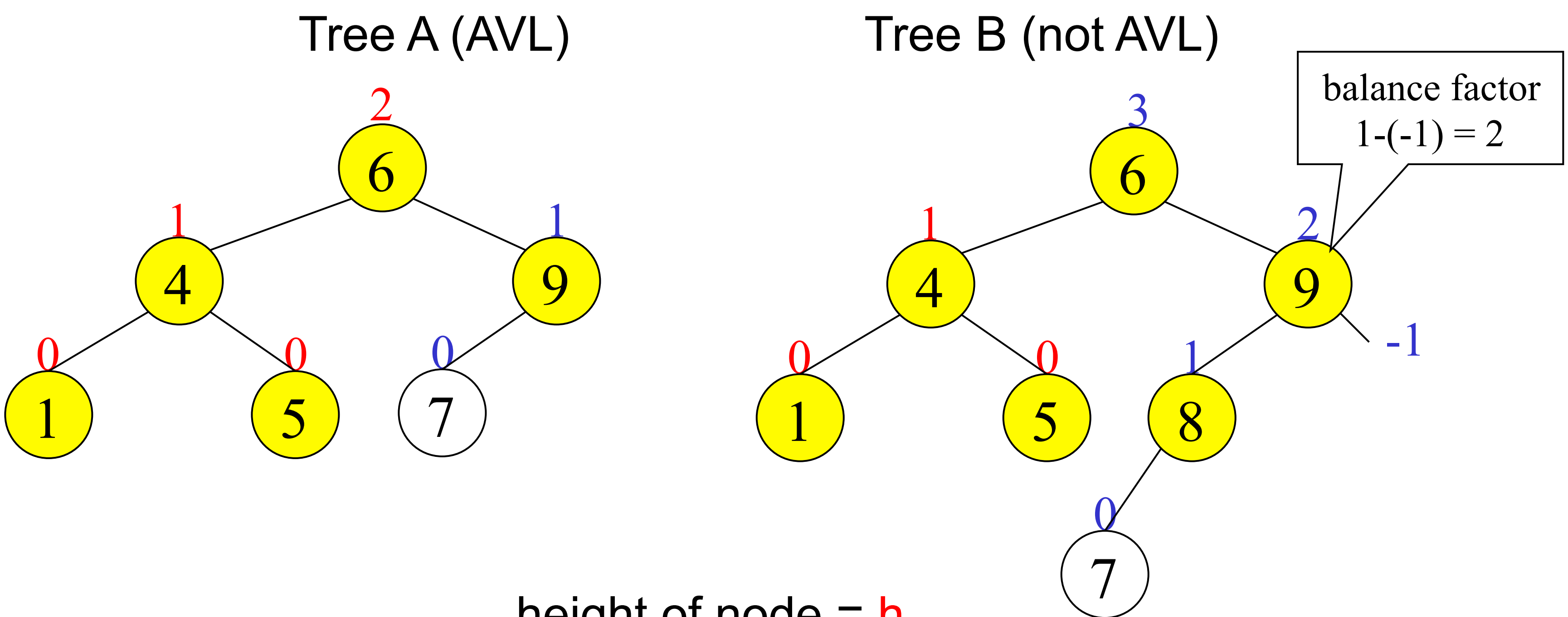
- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - › height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - › For every node, heights of left and right subtree can differ by no more than 1
  - › Store current heights in each node

Node Heights



height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

Node Heights after Insert 7

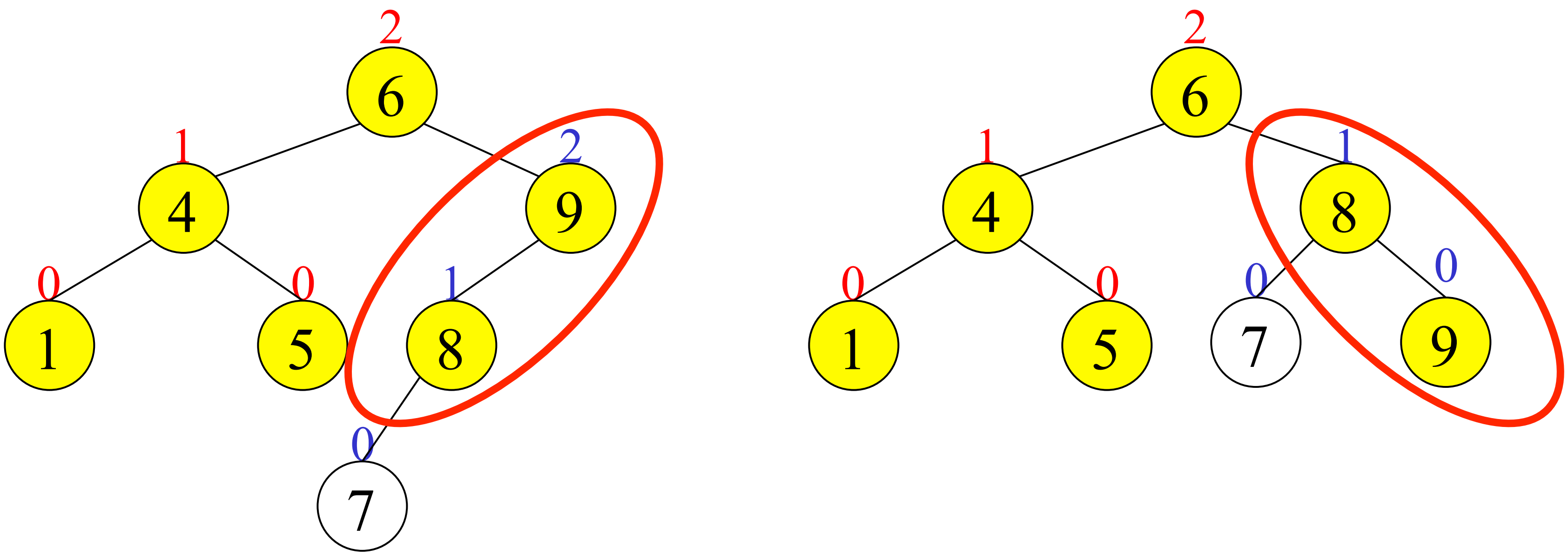


height of node = h
balance factor = $h_{left}-h_{right}$
empty height = -1

Insert and Rotation 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
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation in an AVL Tree

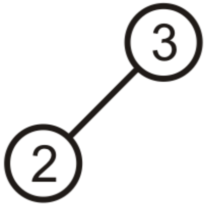


Balanced BST and AVL Trees

- The principle of operation is remarkably simple; let's look at this "prototypical" trick to maintain balance:
  - Let's add 3, 2, 1 (in that order) to a binary search tree:

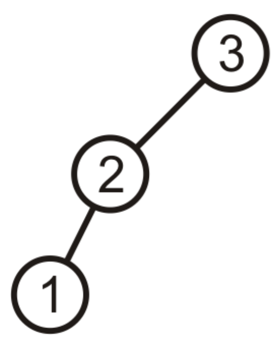
Balanced BST and AVL Trees

- The principle of operation is remarkably simple; let's look at this “prototypical” trick to maintain balance:
  - Let's add 3, 2, 1 (in that order) to a binary search tree:



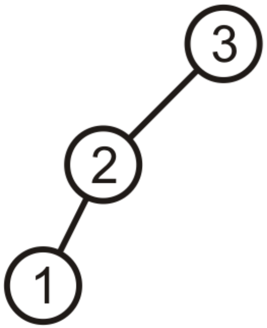
Balanced BST and AVL Trees

- Inserting 1 causes the tree to become imbalanced at node 3 (left sub-tree has height 1, and right sub-tree has height ... ?? you tell me?)



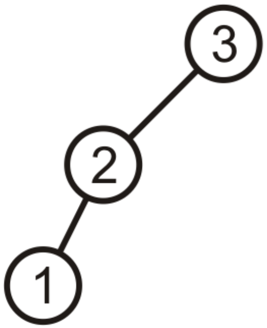
Balanced BST and AVL Trees

- Inserting 1 causes the tree to become imbalanced at node 3 (left sub-tree has height 1, and right sub-tree has height ... ?? you tell me?) — we recall from the first class on trees, that by convention, an empty tree has height −1



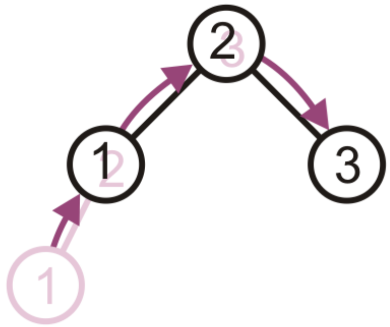
Balanced BST and AVL Trees

- So, we rotate it towards the right:



Balanced BST and AVL Trees

- So, we rotate it towards the right:



Balanced BST and AVL Trees

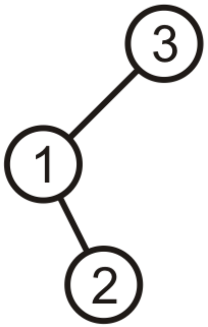
- If we had inserted the sequence 3, 2, 1, the situation would have been essentially the same; the right sub-tree would be the deeper one in that case, so we rotate towards the left in that case.

Balanced BST and AVL Trees

- Inserting the sequence 3, 1, 2, however, does make a significant difference — we end up with the following tree:

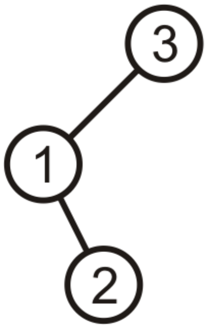
Balanced BST and AVL Trees

- Inserting the sequence 3, 1, 2, however, does make a significant difference — we end up with the following tree:



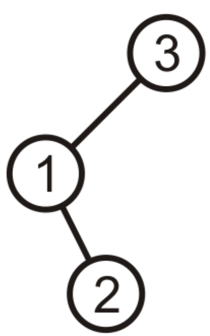
Balanced BST and AVL Trees

- Inserting the sequence 3, 1, 2, however, does make a significant difference — we end up with the following tree:
  - Clearly, we can't rotate to balance (right? *why?*)



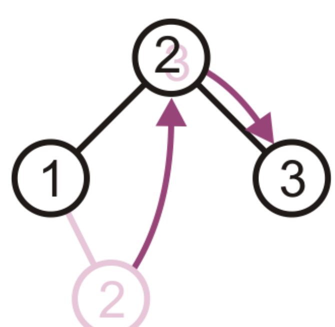
Balanced BST and AVL Trees

- However, we can do a double rotation:
  - First, rotate towards the left the sub-tree 1-2, and then we're in the exact same previous case (so we rotate towards the right)



Balanced BST and AVL Trees

- However, we can do a double rotation:
  - First, rotate towards the left the sub-tree 1-2, and then we're in the exact same previous case (so we rotate towards the right)
  - The “net” effect is the following:

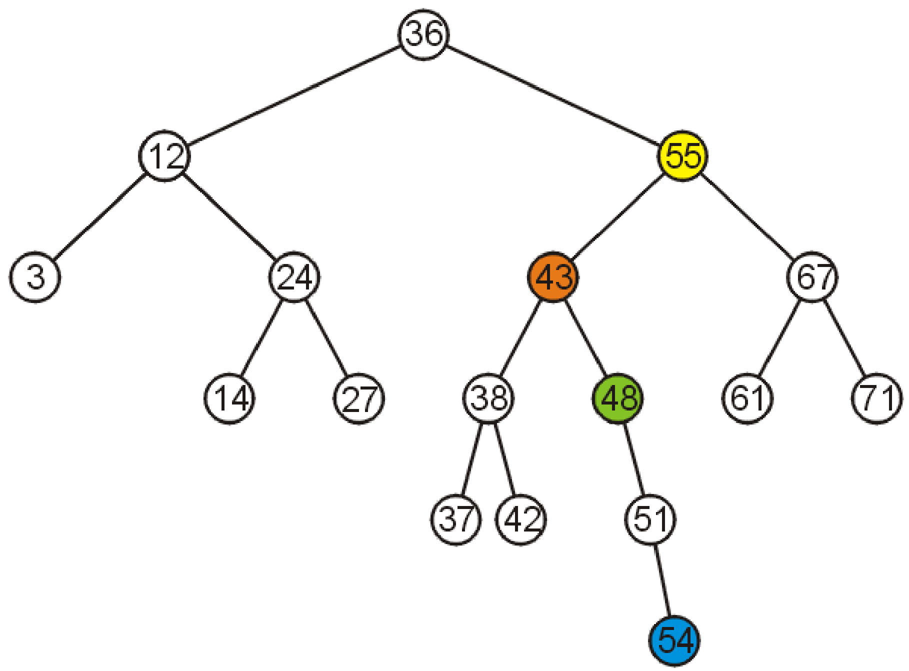


Balanced BST and AVL Trees

- In other words, this tells us that whenever an imbalance is created, we only need to address it at the deepest level where there is imbalance.
  - Things were originally balanced everywhere.
  - Fixing something at a node with depth *d* fixes fixes any imbalance in any node above it.
    - Since nothing else changed in the tree, it must be the case that fixing the imbalance at the deepest node where imbalance is present must be sufficient.

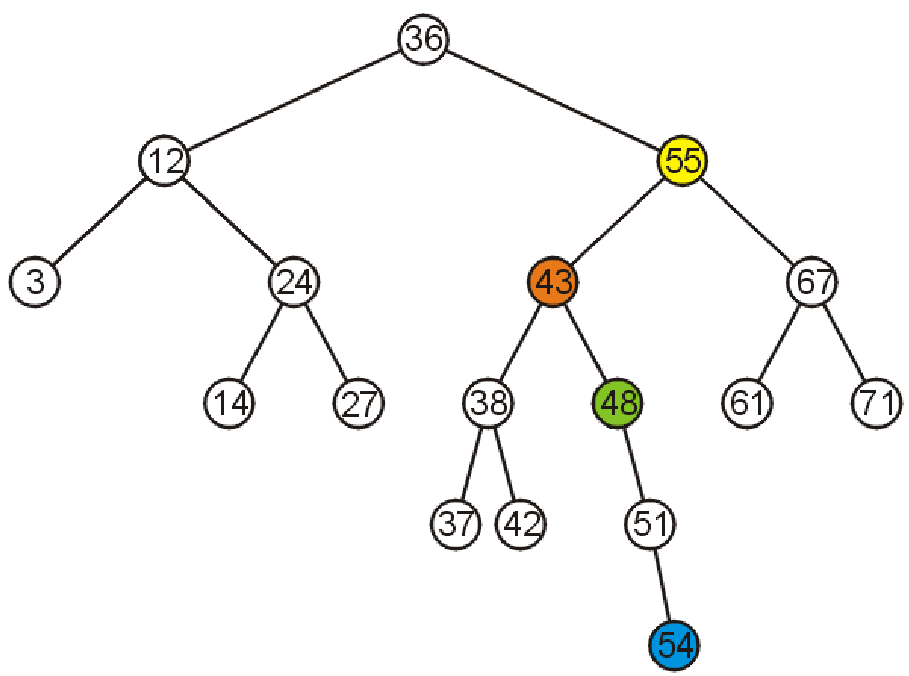
Balanced BST and AVL Trees

- For example, the just added 54 causes imbalance at the root, at 55, and at 48:



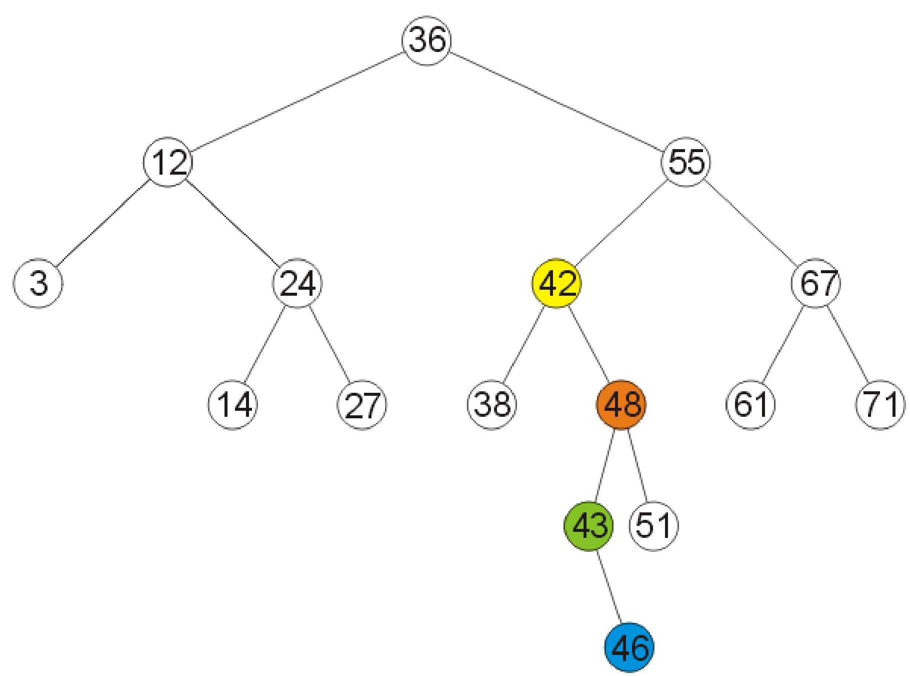
Balanced BST and AVL Trees

- However, it's pretty clear that a rotation around 48 fixes the imbalance at all points!



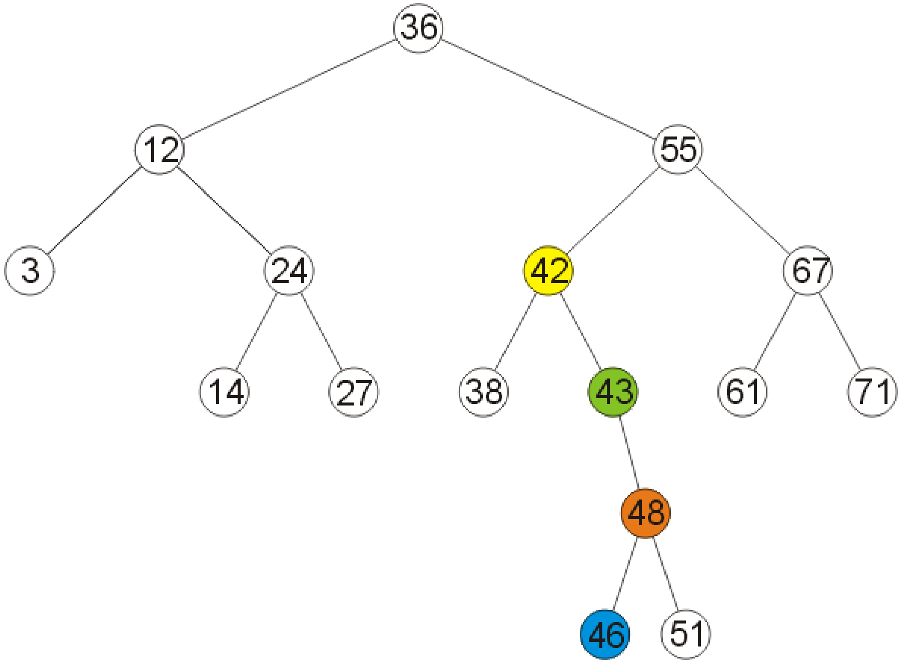
Balanced BST and AVL Trees

- An example: insertion of 46 causes imbalance at nodes 42, 55, and 36 — we address the imbalance at 42 (the deepest node):



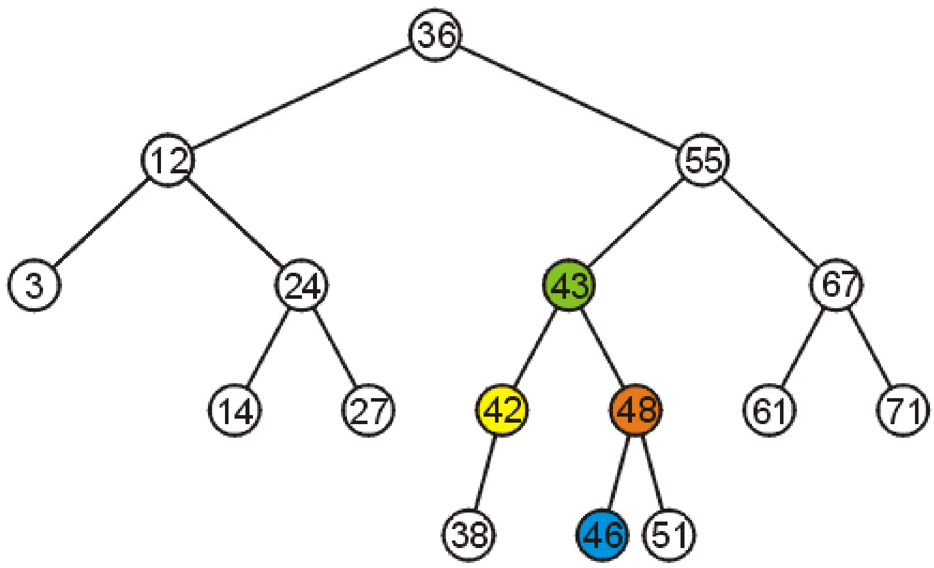
Balanced BST and AVL Trees

- First, rotate towards the right around 43 (detaching 46 and reattaching it as 48's left sub-tree):



Balanced BST and AVL Trees

- Then rotate towards the left around 42:



Balanced BST and AVL Trees

- We observe a piece of good news: these rotations take $\Theta(1)$, and the insertion takes $\Theta(h) = \Theta(\log n)$, so we're in good shape: insertion (including maintaining balance) takes $\Theta(\log n)$
- We'll also see that removals, though a bit more complicated (and more inefficient), also take $\Theta(\log n)$.