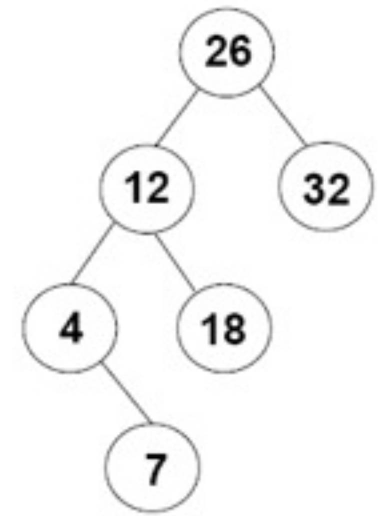
AVL Trees

EECS 233

Height and Balance of Trees

- The height of a tree is the length of the longest path from the root node to a leaf node.
  - What is the height of the tree with root=12?
  - The height of a tree with only one node (a leaf)?

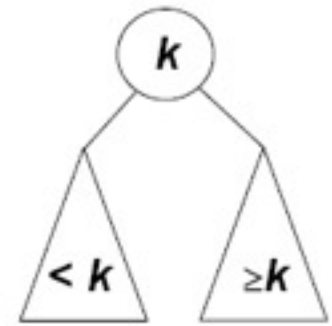


- Define empty tree to have height -1

- *balance* of a tree (with node N as the root): *balance*(N) = height(N's right subtree) – height(N's left subtree)
  - balance(node 26) = 0 – 2 = –2
  - Balance(node 4) = 0 – (–1) = 1

AVL Trees
(Adelson-Velsky & Landis'62)

- An AVL tree is a variant of a binary search tree that takes special measures to ensure that the tree is balanced.
  - Binary search tree
  - Balanced: balance of all nodes are -1, 0, or 1

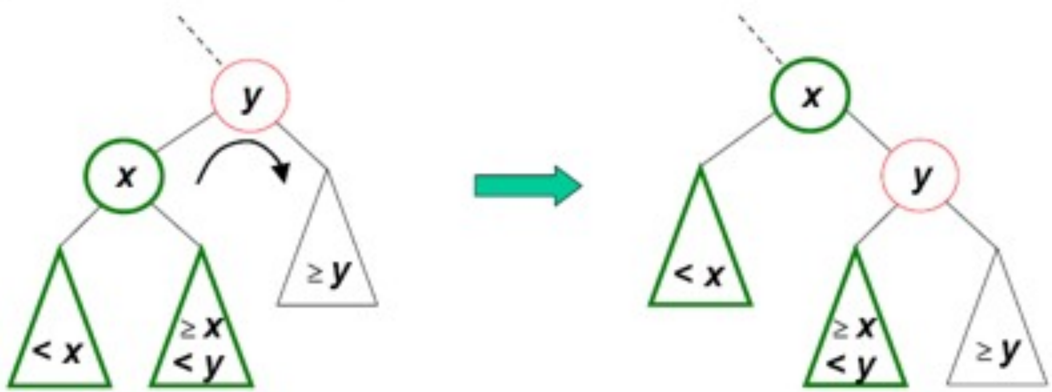


- If a newly inserted node would cause the tree to go out of balance, the nodes are rearranged to restore balance.

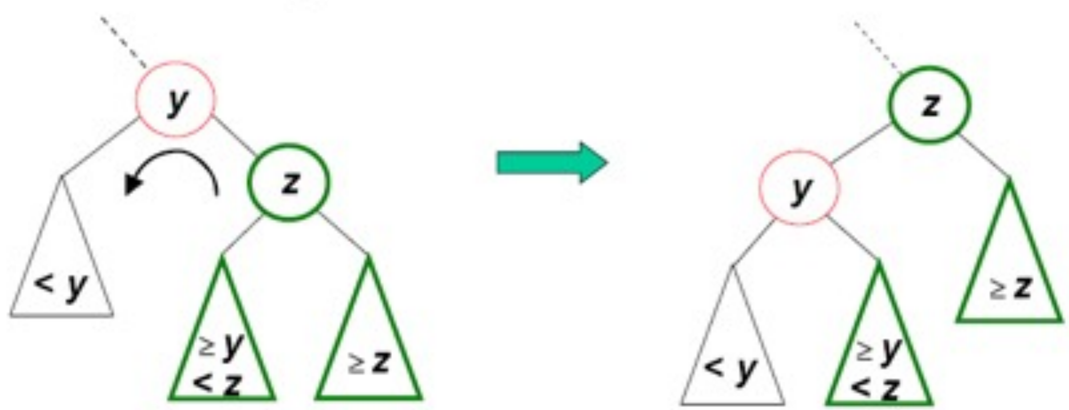
- Challenge: the steps taken to restore balance must:
  - maintain the search-tree inequalities
  - have a worst-case time complexity of $O(\log n)$

Rotation Operations

- A rotation rearranges the nodes in a tree while maintaining the search-tree inequalities.
  - *Right rotation* on (around) y:

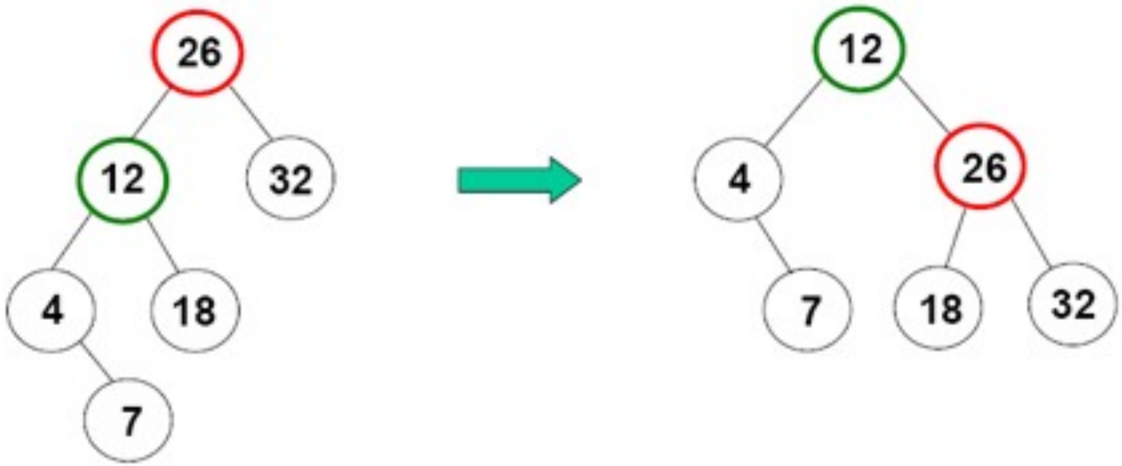


  - *Left rotation* on y:

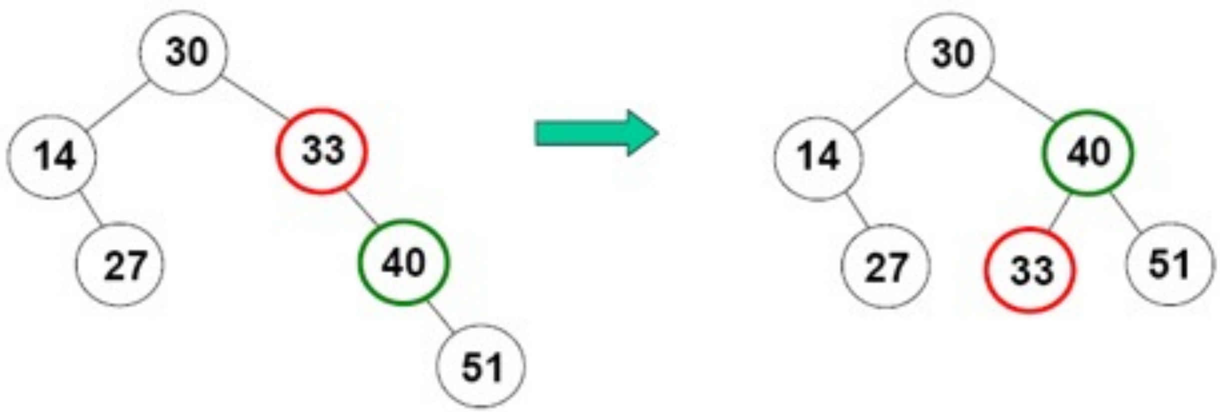


Example Rotations

- Right rotation on node 26



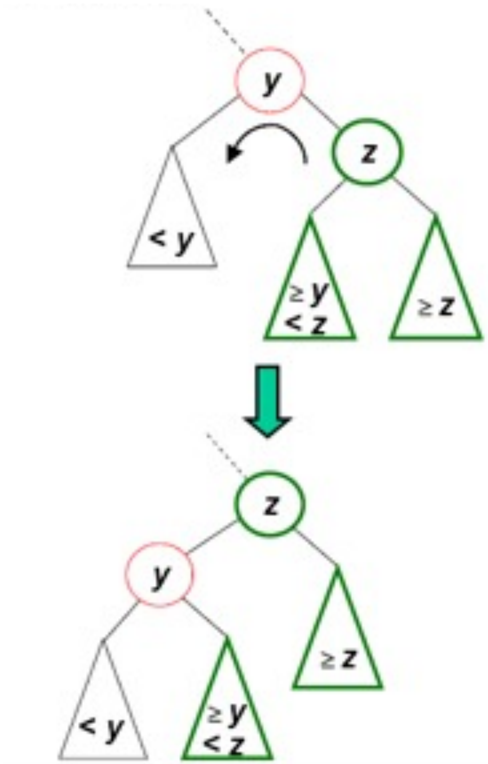
- Left rotation on node 33



Implementation of Rotations

- AVL Tree Representation in Java

```
public class AVLTree {
        private class Node {
                private int key;
                private String data;
                private Node left;              // reference to left child
                private Node right;             // reference to right child
                private Node parent;            // reference to parent node
                private int balance;            // balance value of the node
                …
        }
        private Node root;
        …
}
```



- assume each node has a reference to its parent, i.e., define left, right, and parent references in each node

Implementation of Rotations

- A rotation involves just rearranging pointers/references.
- Example: a left rotation.

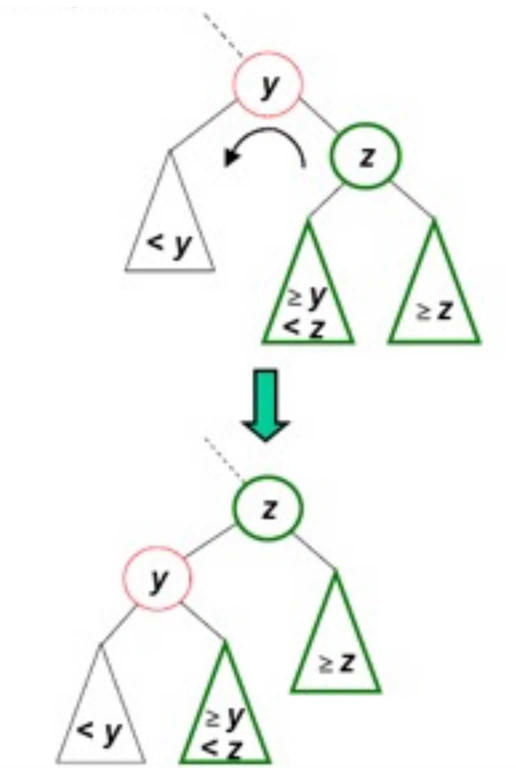
private void LeftRotation(Node y)
{

Involves changing child/parent variables:

- **right child and parent of y,**
- **right child of parent of y**
- **left child and parent of z,**
- **parent of left child of z**

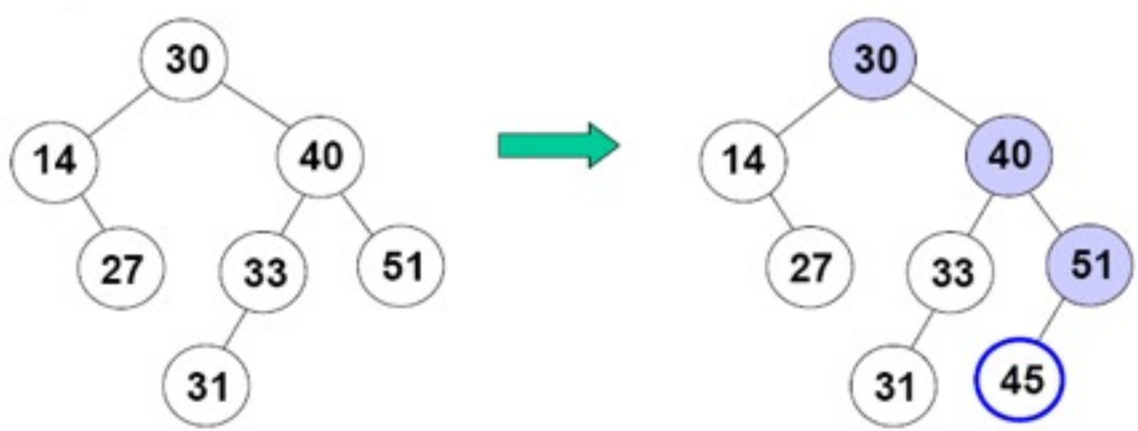
}

Constant-time complexity!



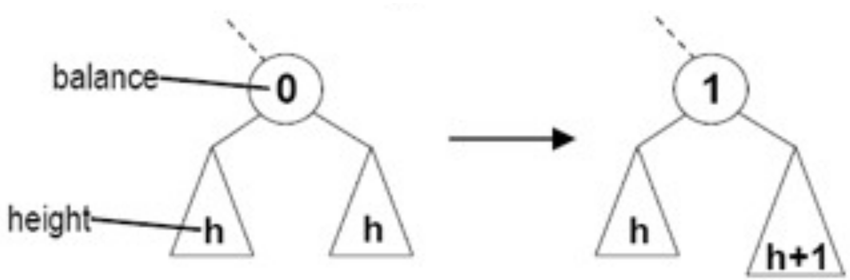
Insertion into AVL Trees

- Remember two new fields:
  - a reference/pointer to the node's parent (null for the root)
  - the node's balance (an integer)
- We begin by inserting the node as in a binary search tree.
- An insertion can only affect the height values and balance values in the new node's ancestors.



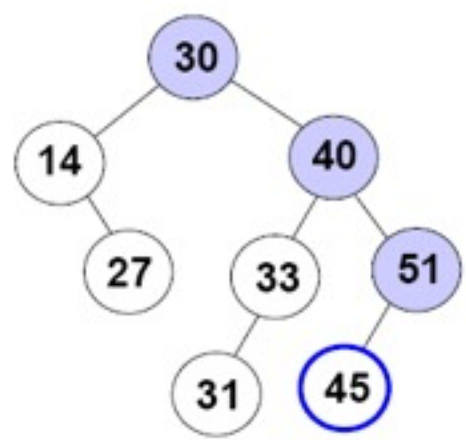
Change in Balance: Case 1

- When an insertion causes the subtree of node N to increase in height, there are three cases
- Case 1: N's balance goes from 0 to +/–1.



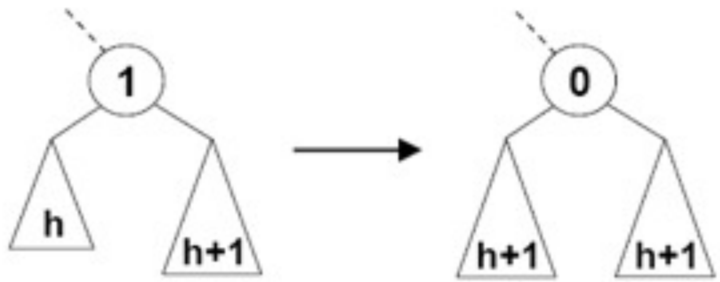
Example: node 51

- Node N's subtree is still within AVL rules
- The height of N has increased
- So the balance of N's parent *will* change
- Need to check if N's parent satisfies the AVL rule.
- The balance of other ancestors may also be affected



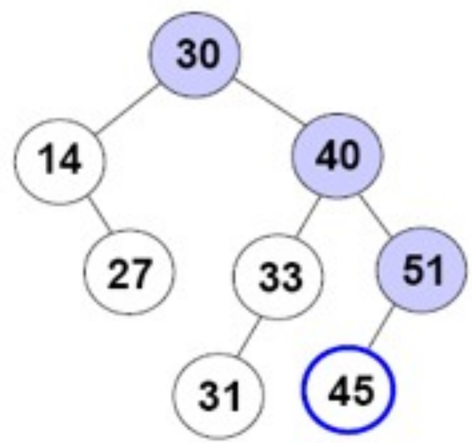
Change in Balance: Case 2

- Case 2: N's balance goes from +/–1 to 0.



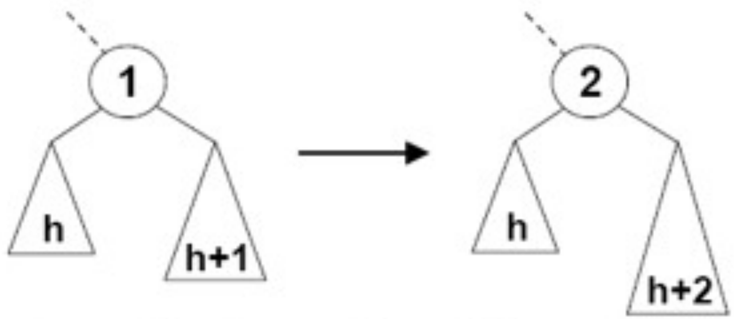
Example: node 40 at right

The height of the subtree of which N is the root has *not* changed, so we don't need to look at its ancestors



Change in Balance: Case 3

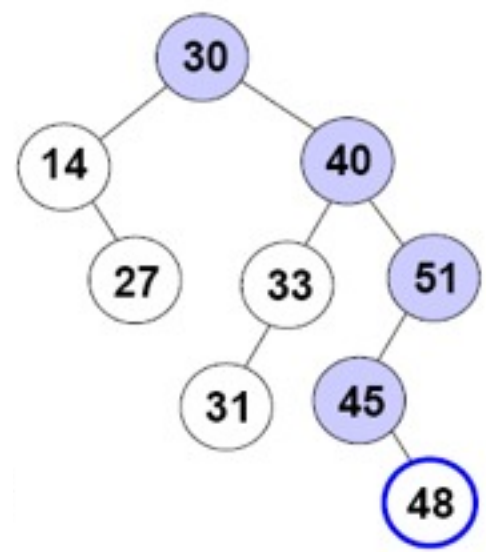
- Case 3. N's balance goes from 1 to 2 or from -1 to -2.



Example: after inserting 48 in the previous tree, which node has a balance of +2 or –2?

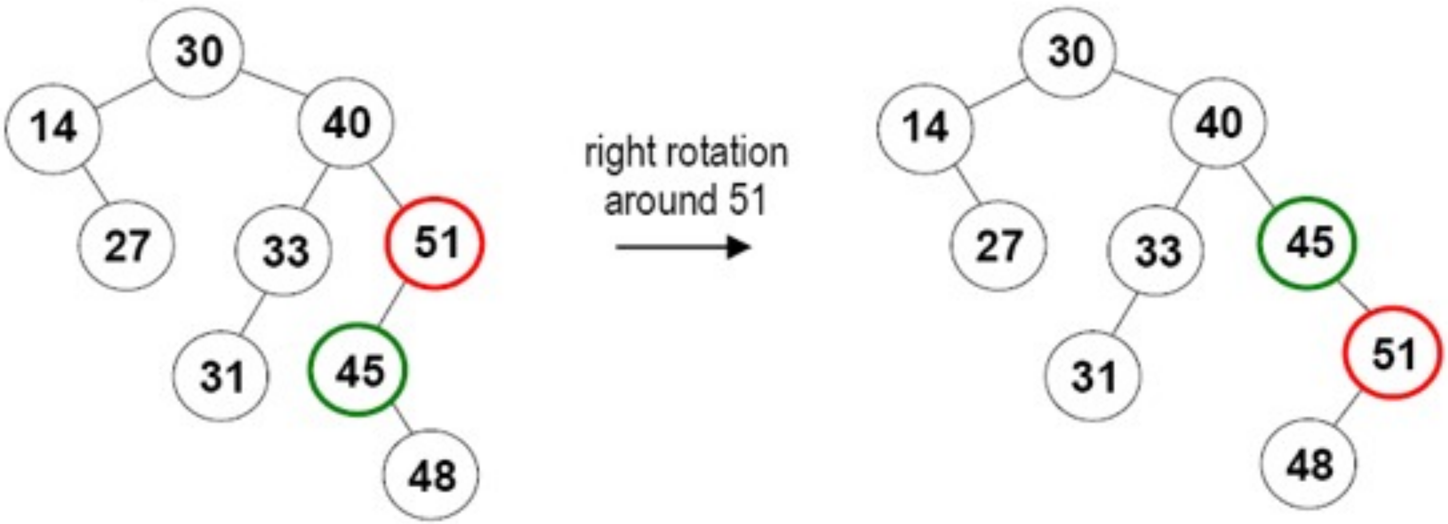
we need to rebalance the tree using rotations

A good thing: the rotations will restore the height that the rotated subtree had before the insertion, and thus N's ancestors' balances won't change.



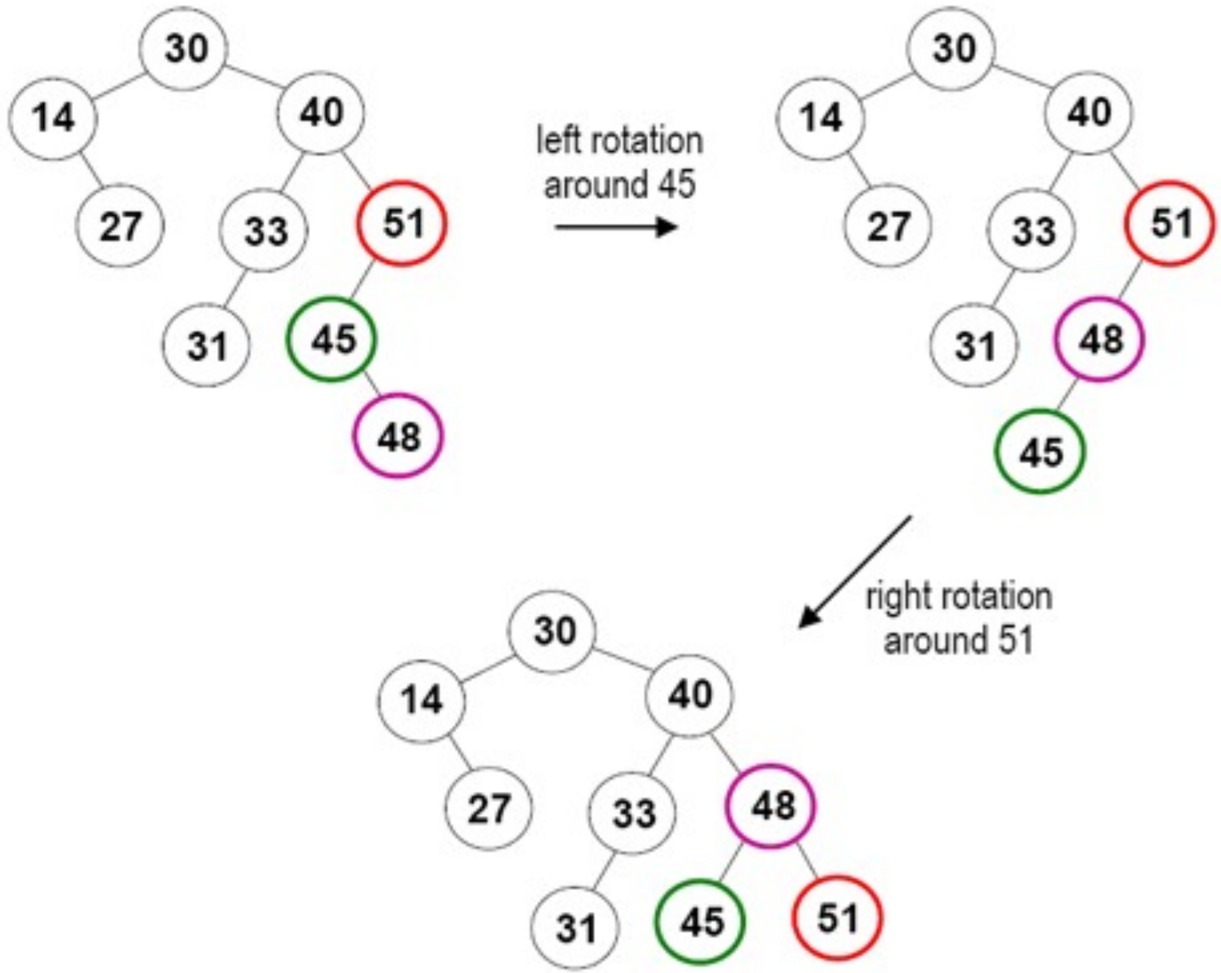
A Puzzle?

- A single rotation doesn't always rebalance the tree.



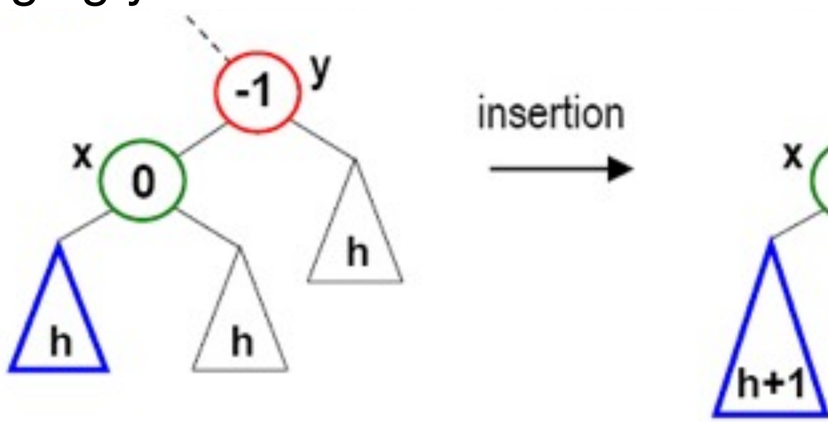
Double Rotations

- Instead, we perform two rotations (a *double rotation*):

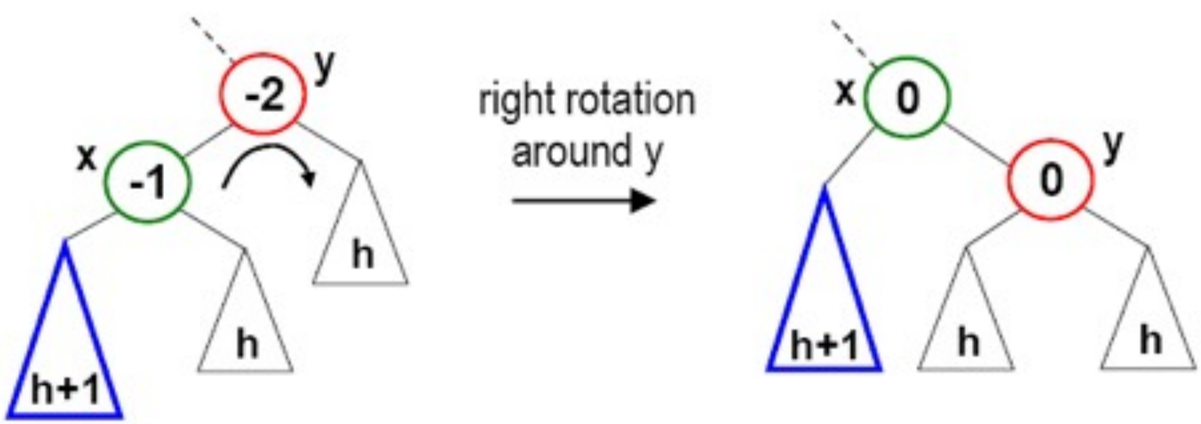


Rebalance an AVL Tree (1)

- Case 3a: we've added a node to the left subtree of y's left child, x, bringing y's balance to -2.

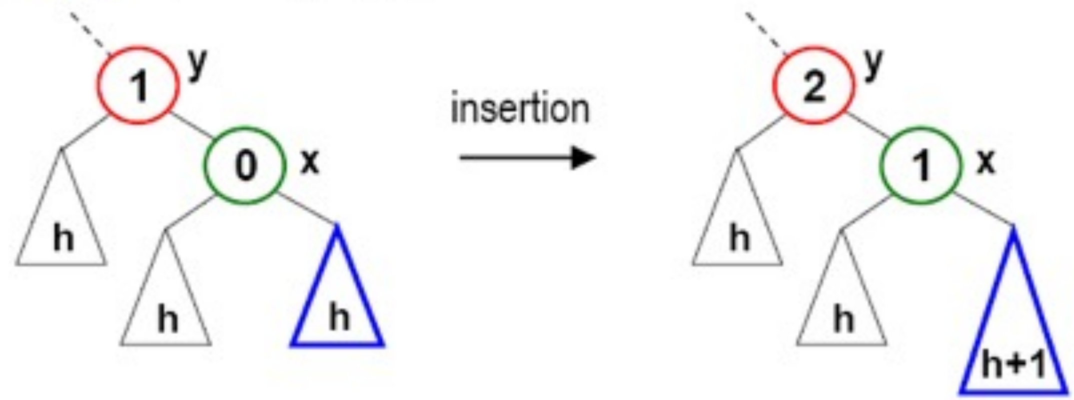


  - rebalance by performing a single right rotation around y:

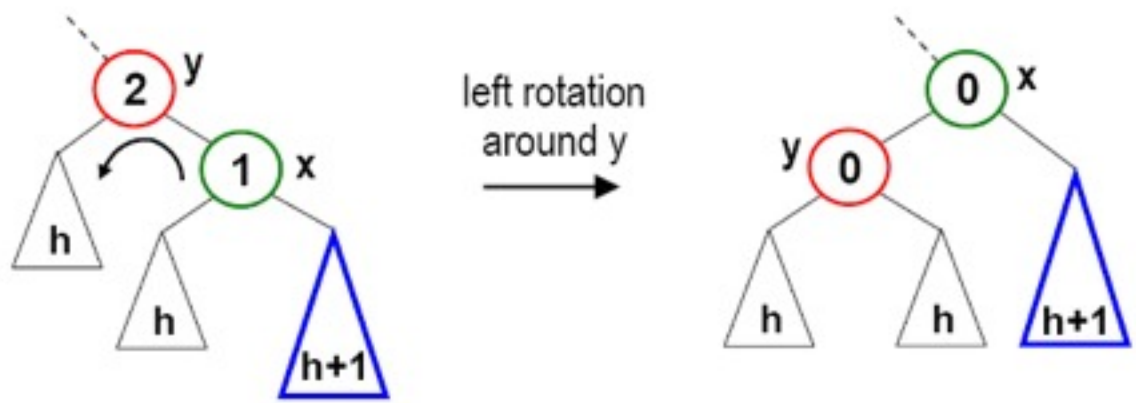


Rebalance an AVL Tree (2)

- Case 3b: we've added a node to the right subtree of y's right child, x, bringing y's balance to +2. (*symmetric to case 3a*)

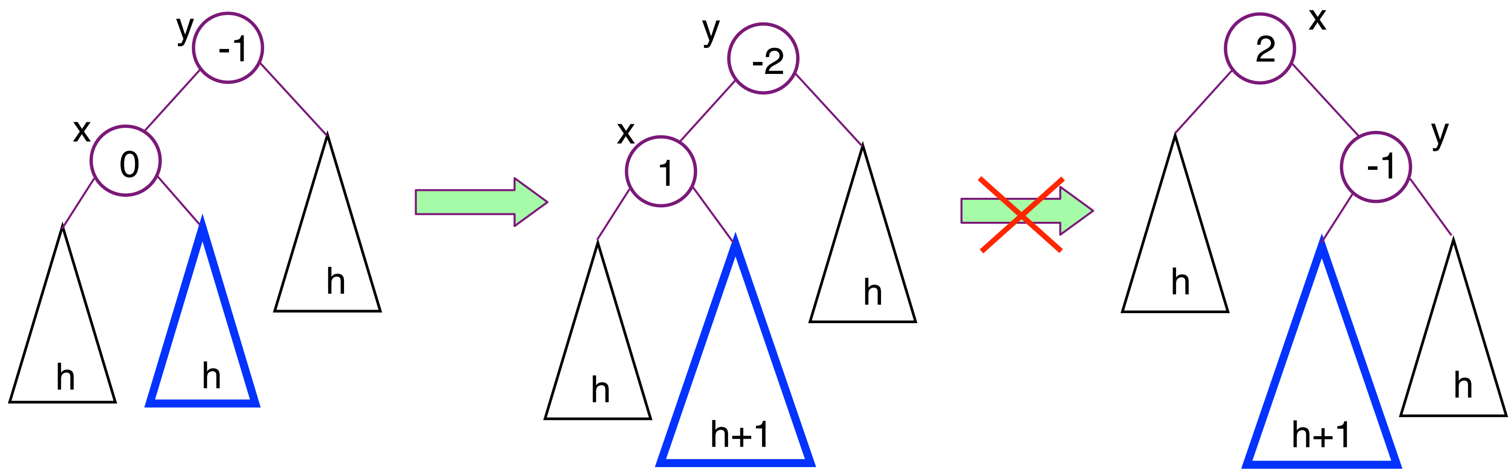


  - rebalance by performing a single left rotation around y:



Rebalance an AVL Tree (3)

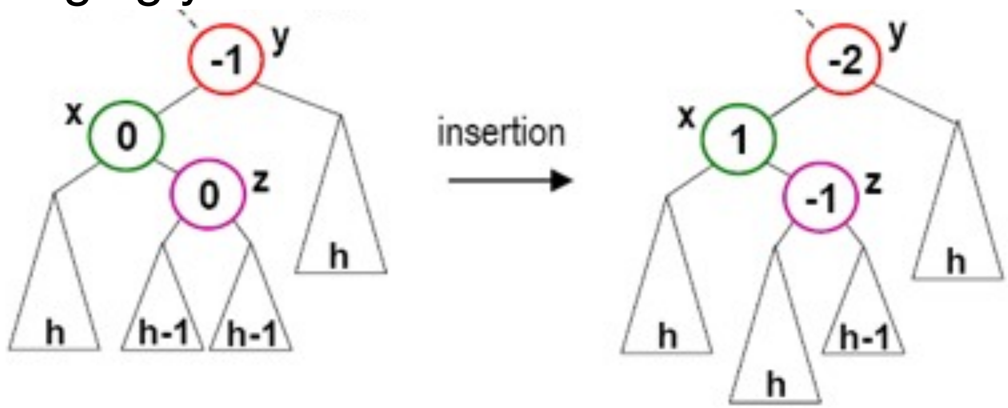
- Case 3c: we've added a node to the right subtree of y's left child, x, bringing y's balance to -2.



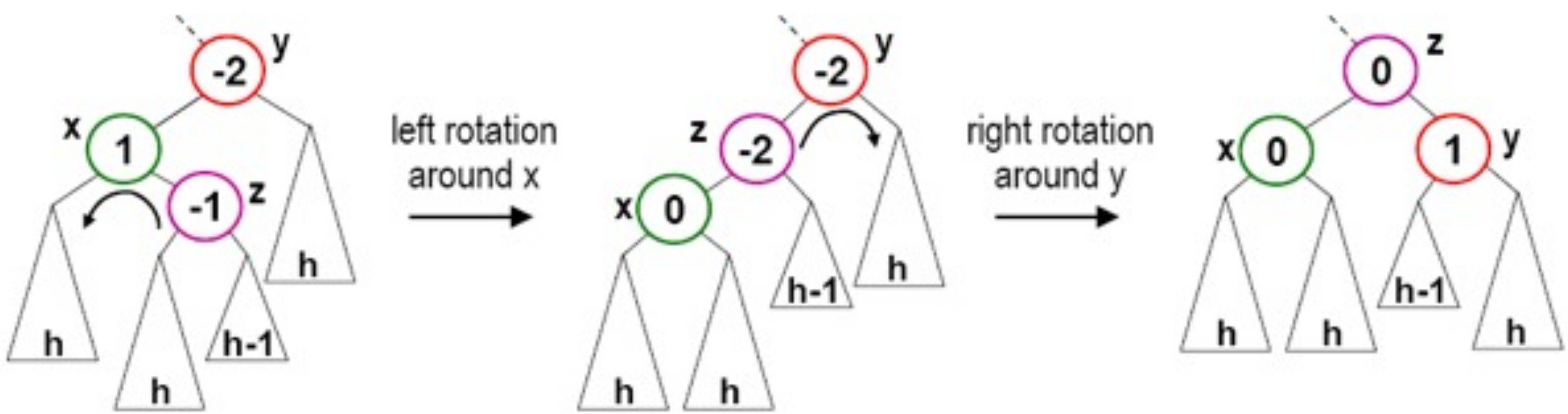
  - Single rotation does not help

Rebalance an AVL Tree (3)

- Case 3c: we've added a node to the right subtree of y's left child, x, bringing y's balance to -2.

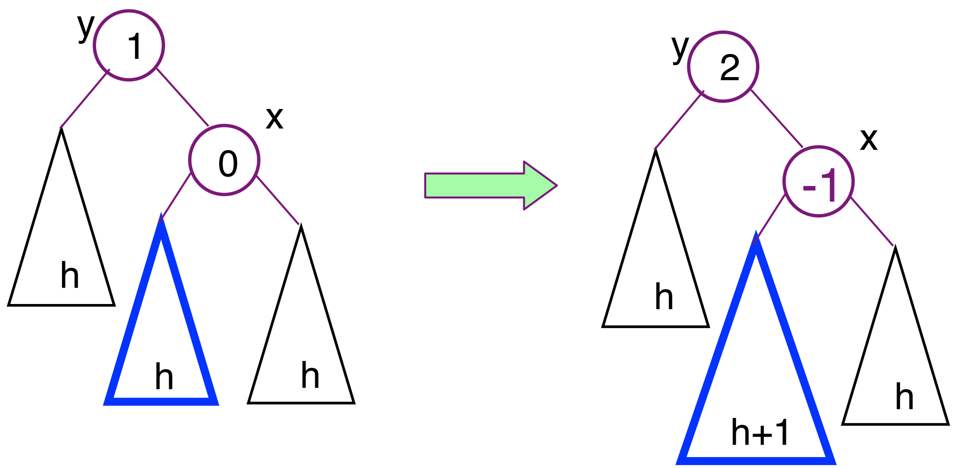


  - rebalance by performing a left-right double rotation: left around x, right around y:



Rebalance an AVL Tree (4)

- Case 3d: we've added a node to the left subtree of y's right child, x, bringing y's balance to +2. (*symmetric to case 3c; can you draw pictures to show how it works?*)

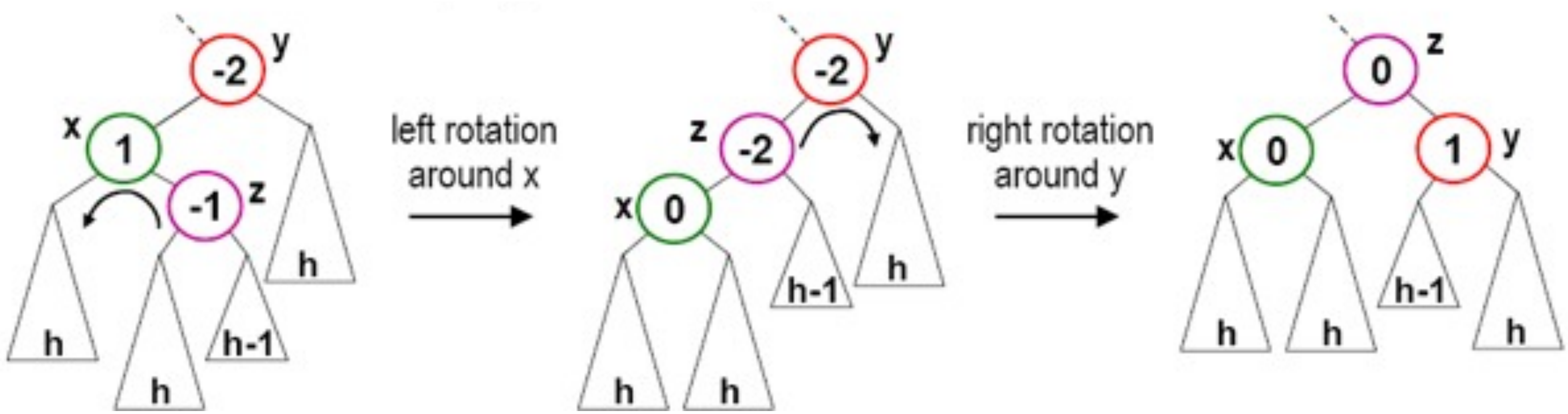


  - rebalance by performing a right-left double rotation: right around x, left around y

Implementation of Double Rotations

```
private void leftRightRotation(Node y)
{
        leftRotation(y.left);
        rightRotation(y);

}
```

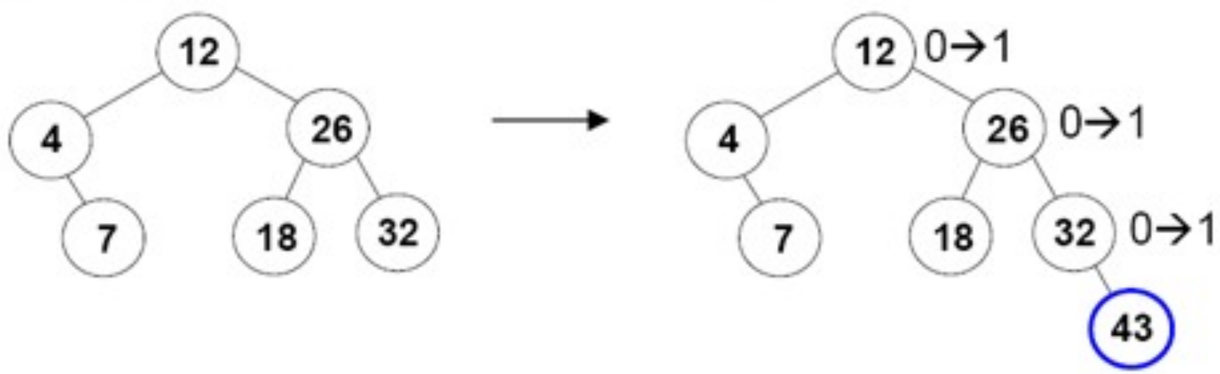


Complete Insertion Method

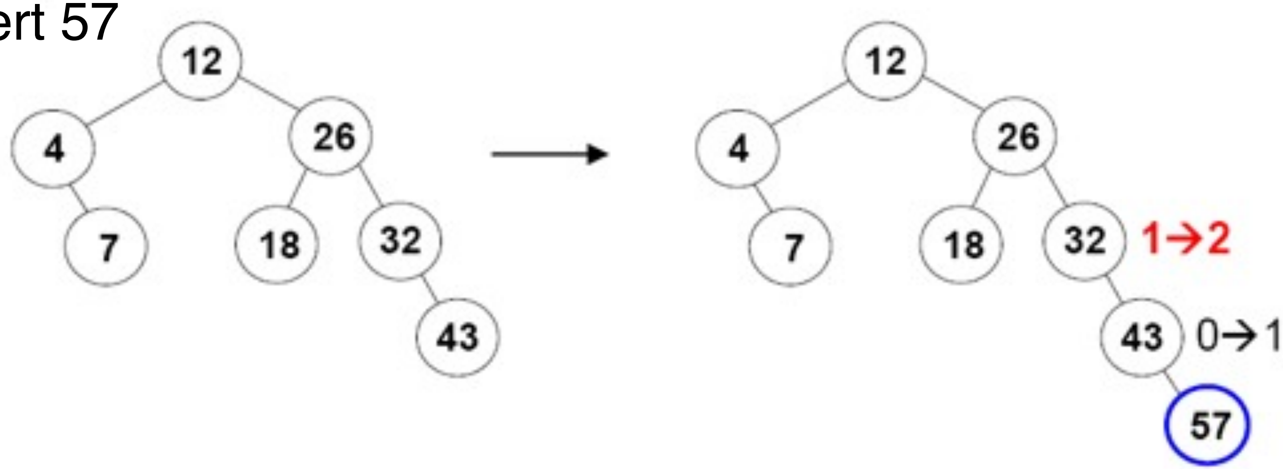
- Insert the new node N as in a binary search tree.
- Use parent references to follow the path from N back to the root:
  - if an ancestor's balance was 0, it will now be +/–1 (case 1), continue up the path to the root
  - if an ancestor's balance was +/–1, we have two cases:
    - ✓ it is now 0 (case 2): stop
    - ✓ it is now +/–2, and we need to perform 1 or 2 rotations to rebalance the tree (cases 3a – 3d), depending on where N is inserted (left-left, right-right, left-right, right-left).
    - ✓ in either case, stop (no need to go any further up the tree)

Examples of Insertion

- Insert 43. No balancing is required

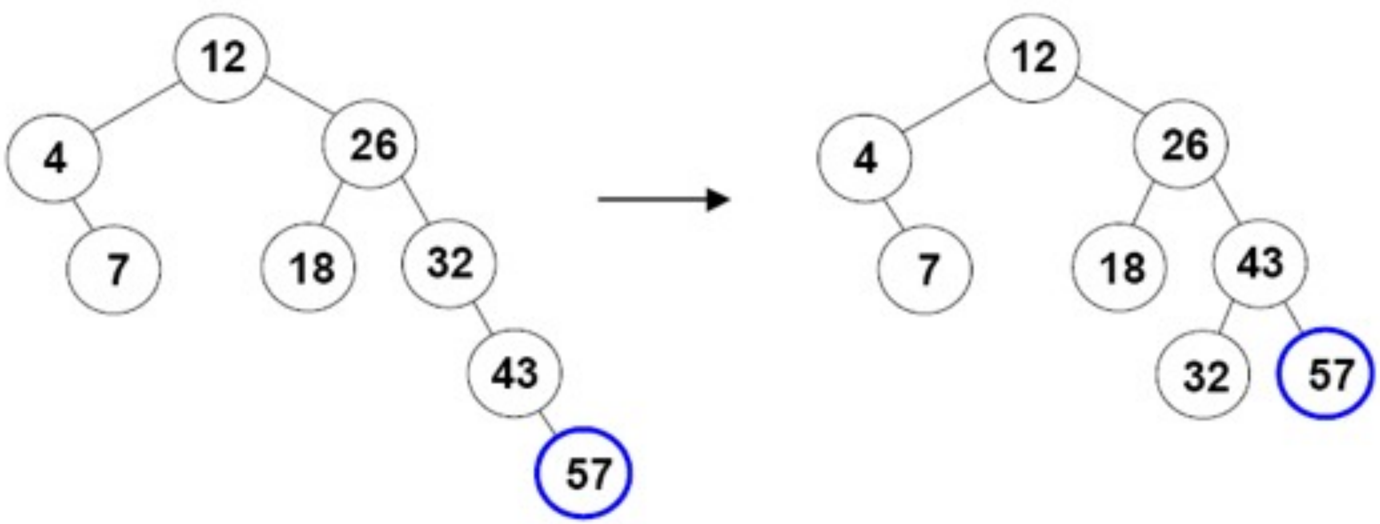


- Insert 57



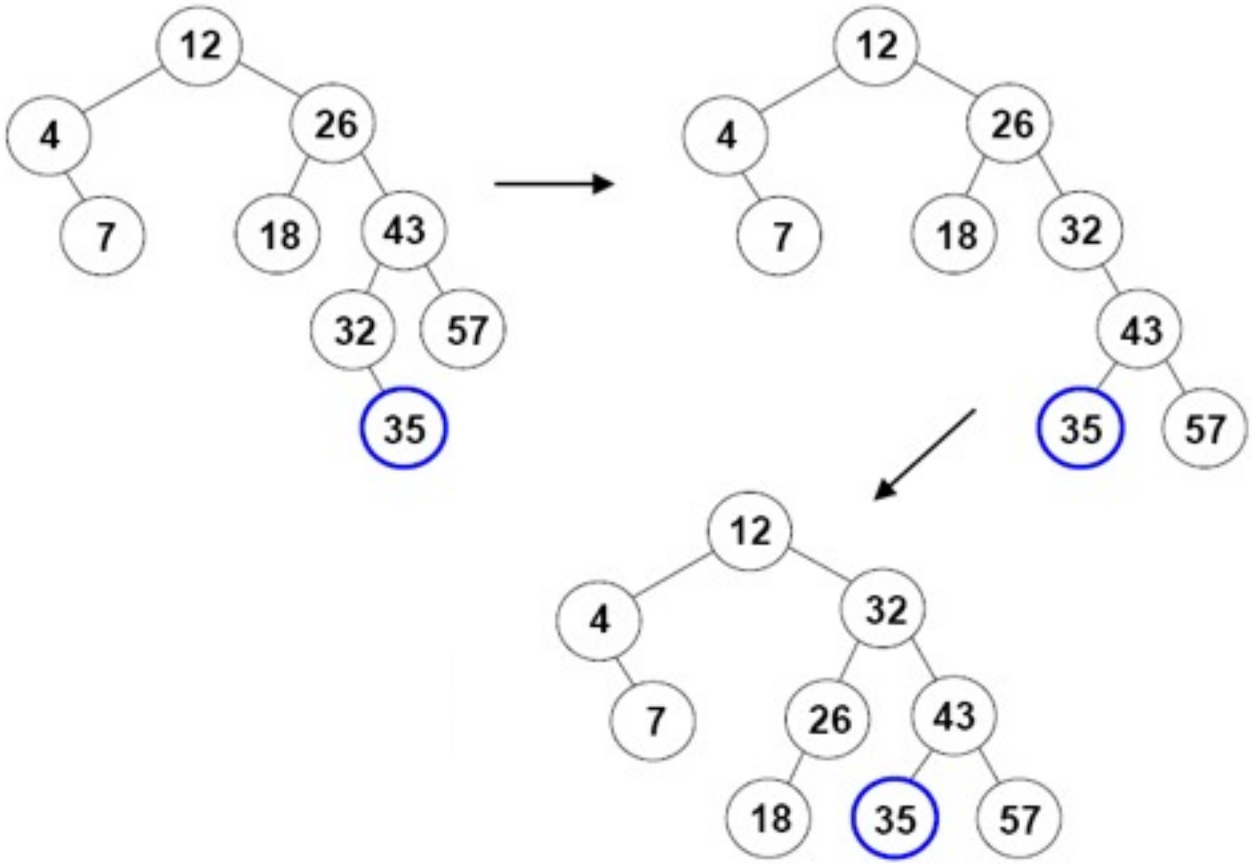
Examples of Insertion

- 32's balance is too big. Since we added a node to the right subtree of its right child, this is case 3b.
  - we rebalance using a single left rotation about 32:
  - we don't need to go further up the tree (the balances of 26 and 12 are unchanged)



Examples of Insertion

- Insert 35
- Node 26's balance is too big. What case is this?
- We rebalance using a right-left double rotation.



An Exercise

- Insert 15, 13, 14