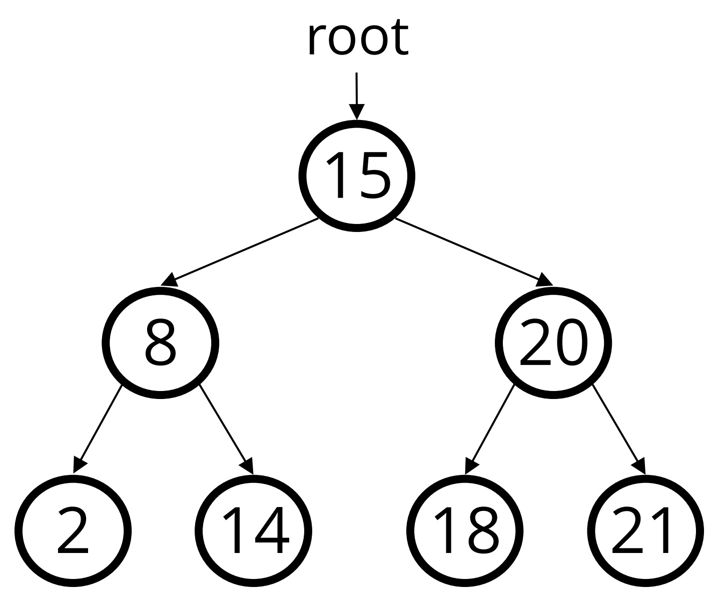
COSC 222 Data Structure

Tree Balancing
AVL Trees

Some height numbers

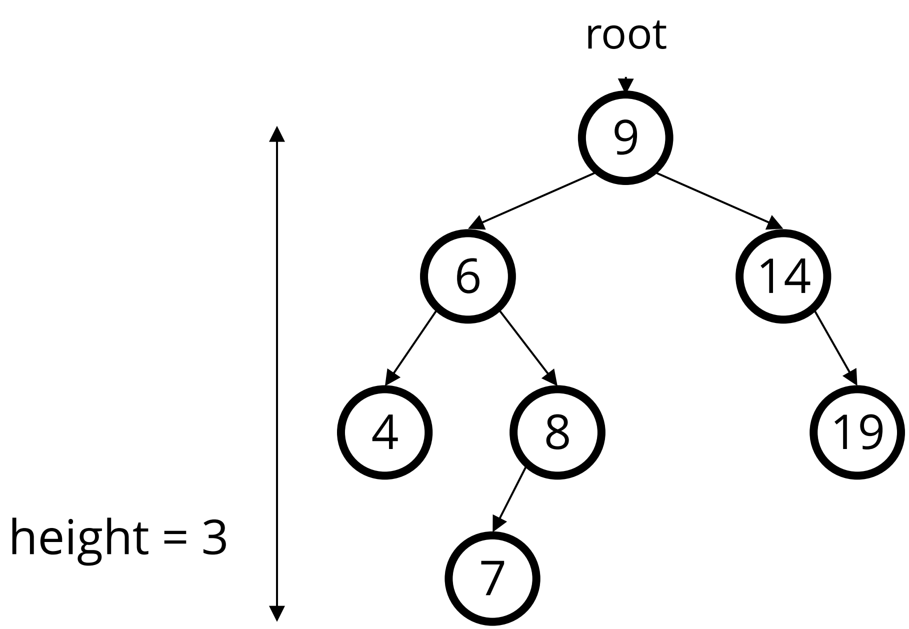
- Observation: The shallower the BST the better.
  - Average case height is O(log N)
  - Worst case height is O(N)
  - Simple cases such as adding (1, 2, 3, ..., N), or the opposite order, lead to the worst case scenario: height O(N).

- For binary tree of height h:
  - max # of leaves: 2^h
  - max # of nodes: $2^{h+1} - 1$
  - min # of leaves: 1
  - min # of nodes: h+1



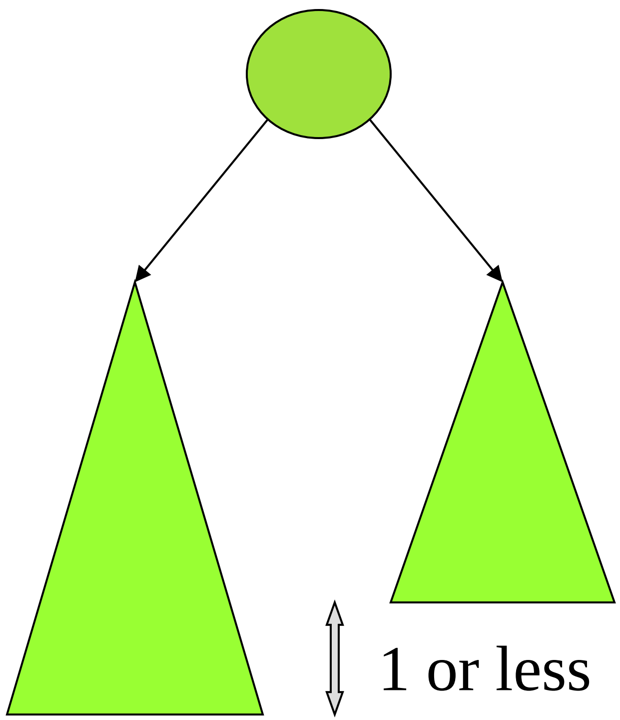
A good tree

- Balanced tree: One whose subtrees differ in height by at most 1 and are themselves balanced.
  - The runtime of adding/searching a BST is related to height
  - A balanced tree of N nodes has a height of ~ $\log_2$ N.



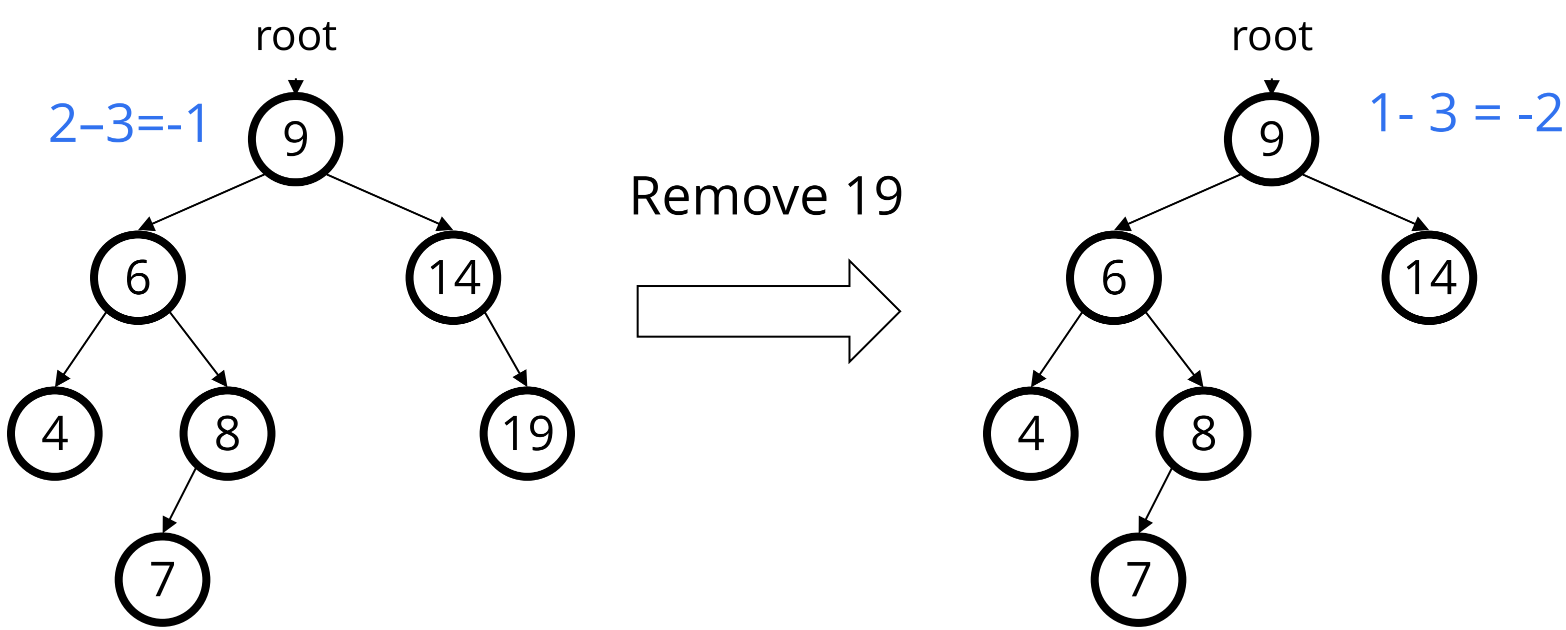
AVL trees

- G. M. **A**delson-**V**elskii and E. M. **L**andis, 1962
- AVL tree: a binary search tree that uses modified add and remove operations to stay balanced as its elements change



AVL trees

- When nodes are added to / removed from the tree
  - if the tree becomes **unbalanced**
  - **repair** the tree until balance is restored.



Balance factor

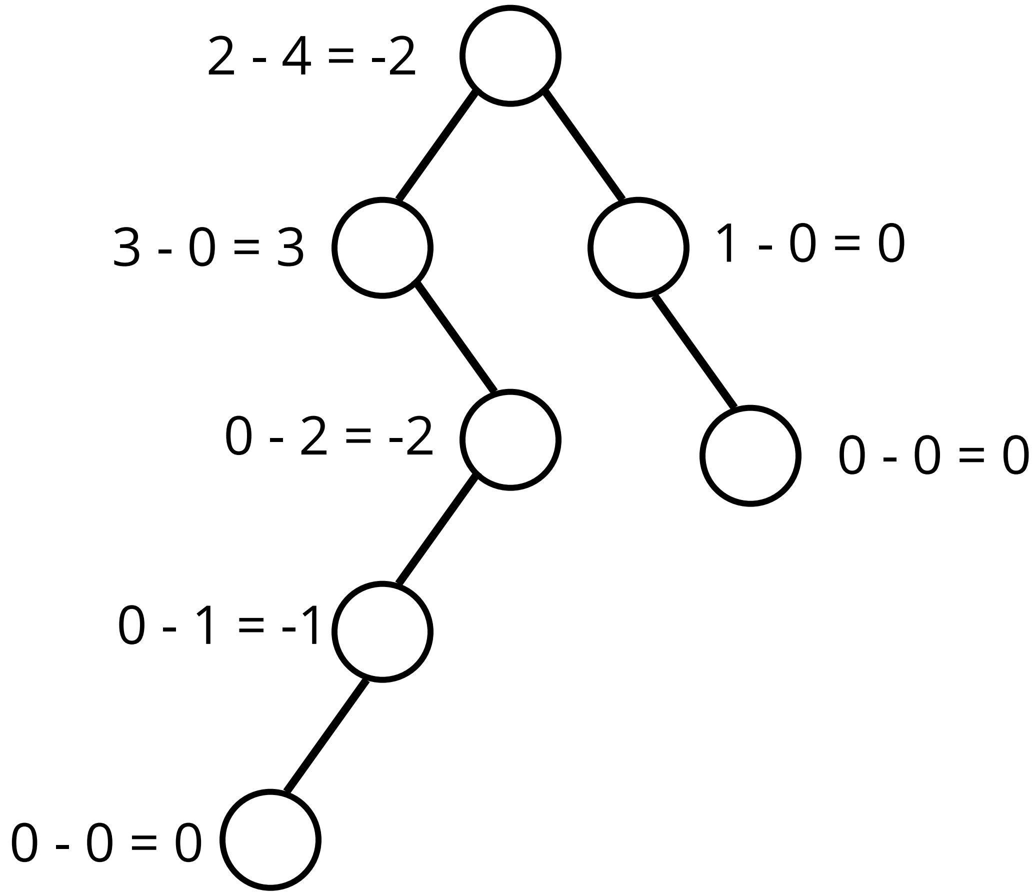
- **Balance Factor (BF)**, for a tree node T :

  = height of T's right subtree minus height of T's left subtree.

  - BF(T) = **Height(T.right) - Height(T.left)**
  - Each node of a tree compute BF

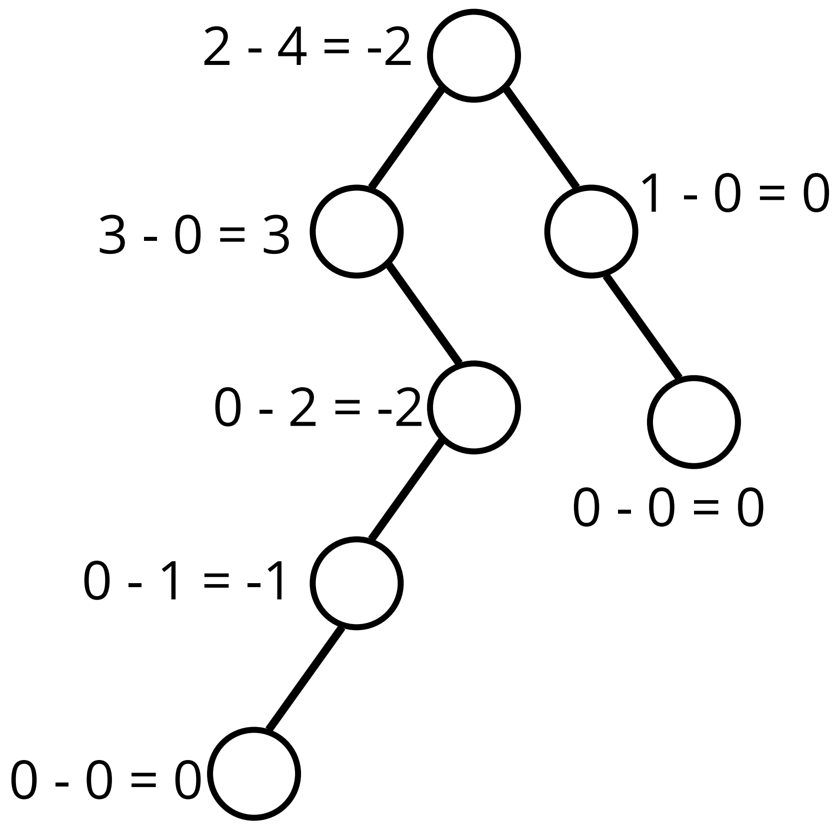
Balance factor

- Balance factor in each node:

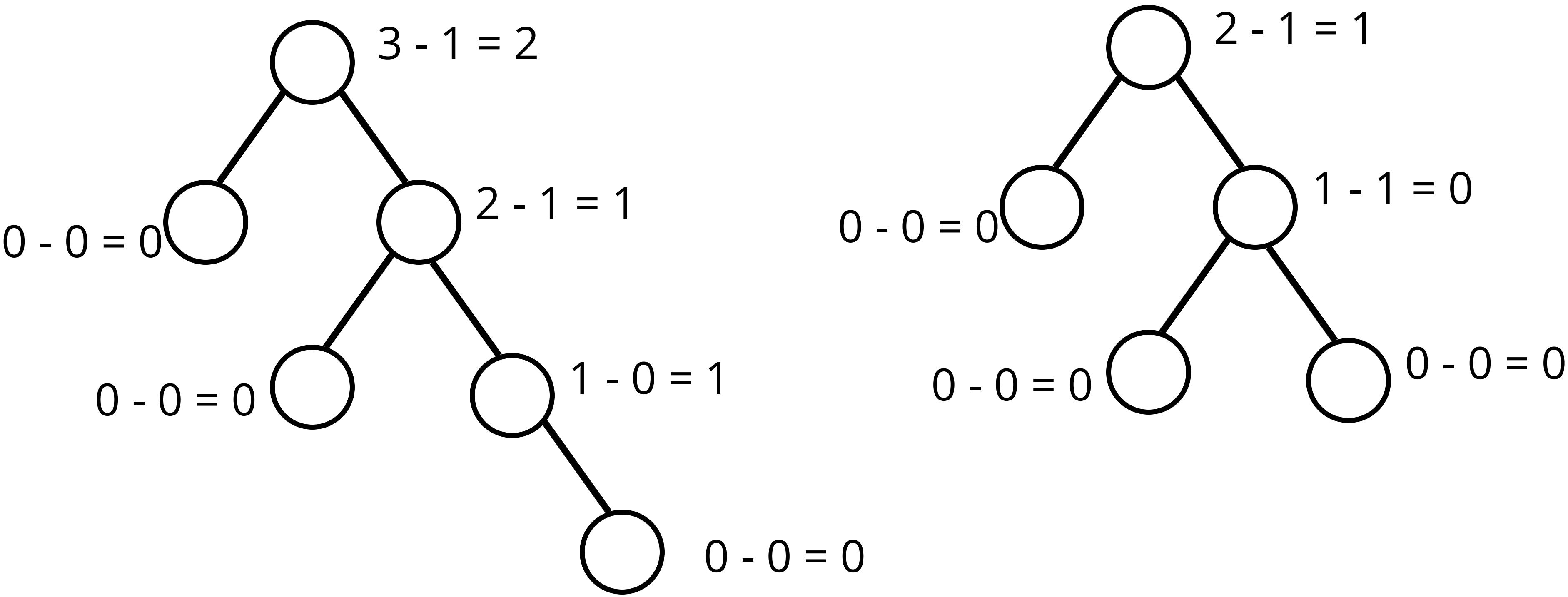


Balance factor

- an AVL tree maintains a balance factor in each node of 0, 1, or -1
  i.e. no node's two child subtrees differ in height by more than 1

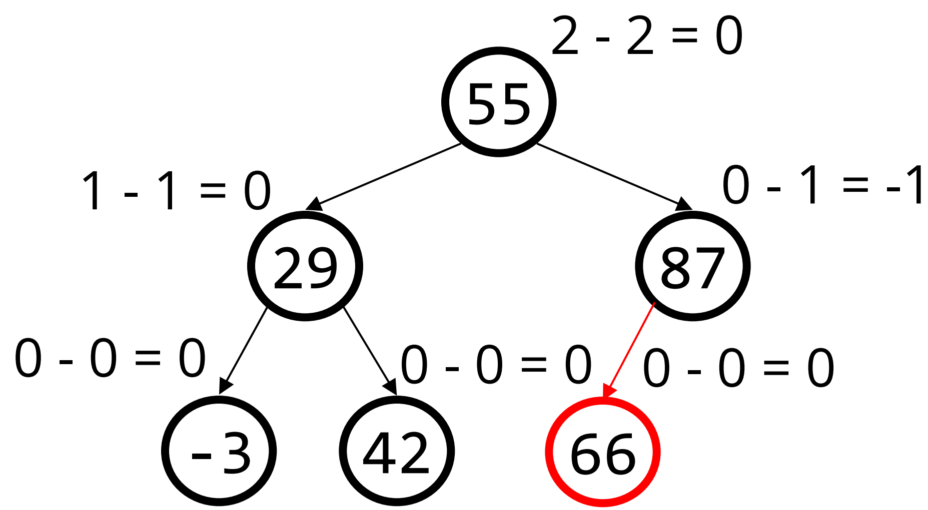


Balance factor



AVL add operation

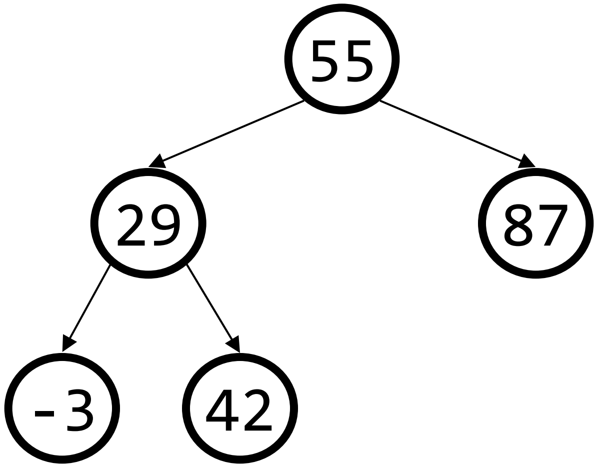
- For all AVL operations, we assume the tree was balanced before the operation began.
- Adding a new node begins the same as with a typical BST, traversing left and right to find the proper location and attaching the new node.

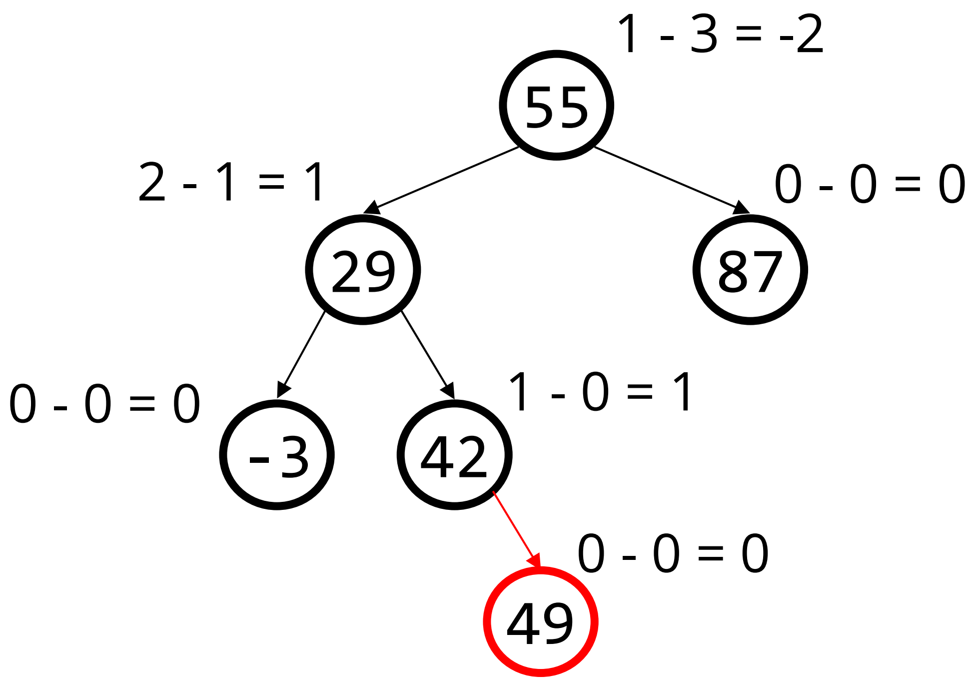


AVL add operation

- But adding a new node may unbalance the tree by 1:

add(49)

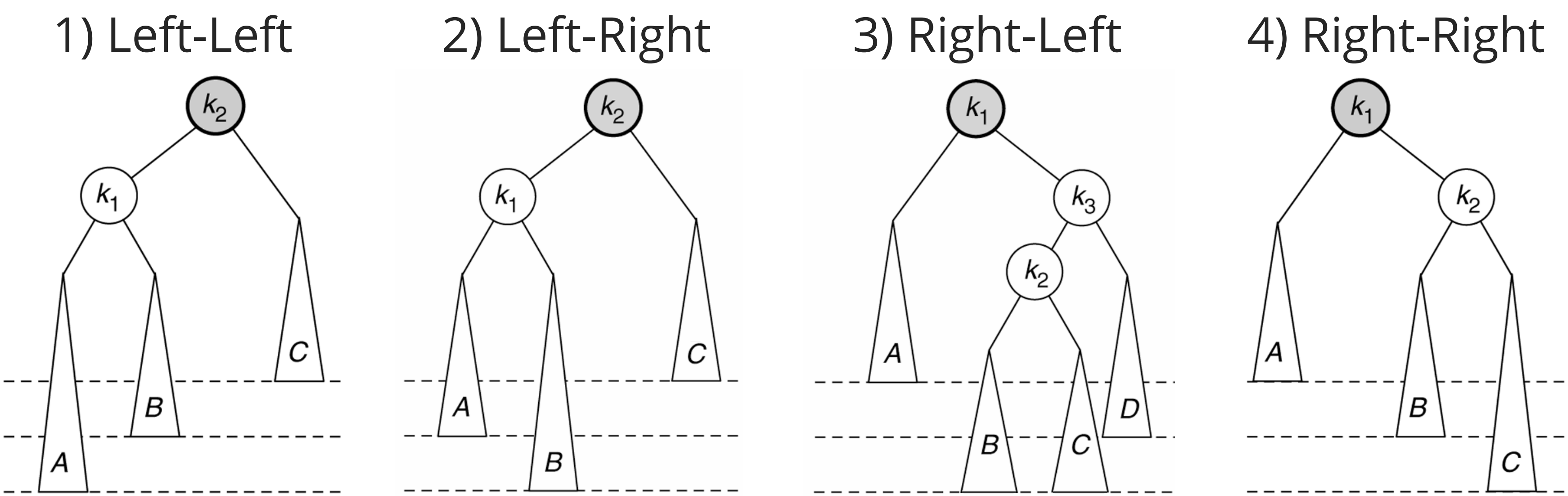




AVL tree insert

- Consider the highlighted node k_2 / k_1 that become unbalanced
  - The new offending node could be in one of the four following grandchild subtrees:

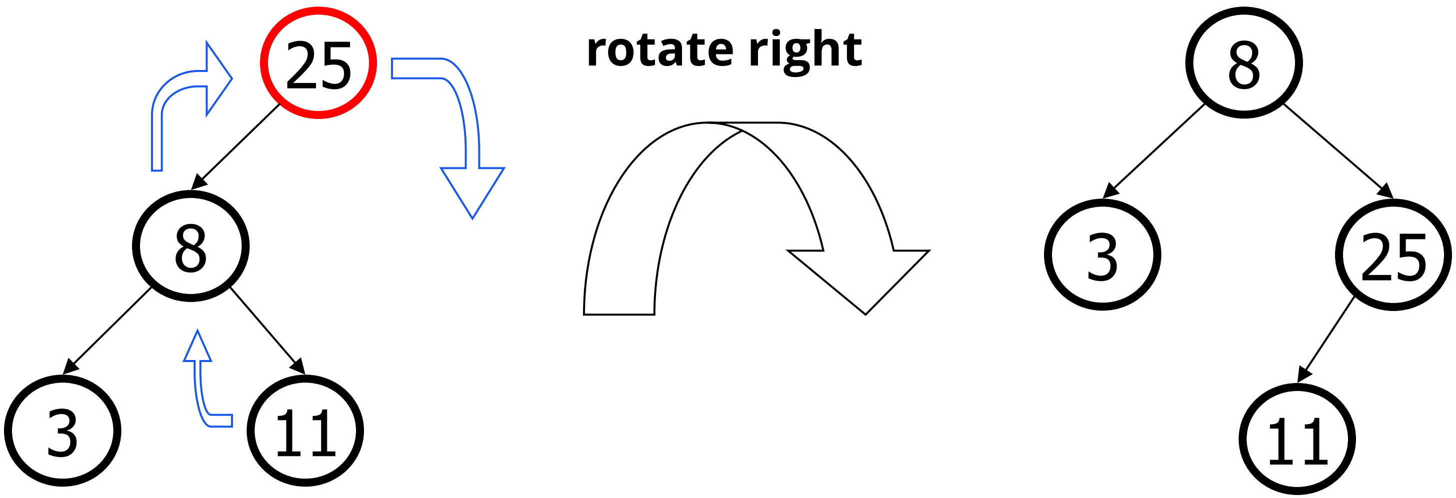
1. *Left-left case*: **left** subtree of the **left** child of b.
2. *Left-right case:* **right** subtree of the **left** child of b.
3. *Right-left case:* **left** subtree of the **right** child of b.
4. *Right-right case:* **right** subtree of the **right** child of b.



AVL tree insert

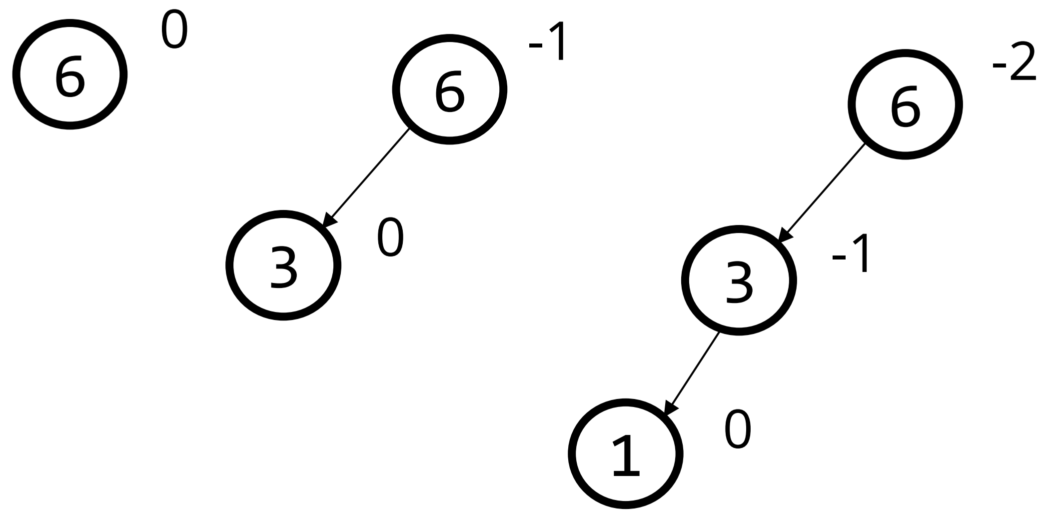
Idea:

- 1 & 4 are solved by a single rotation.
- 2 & 3 are solved by a double rotation.



Case #1: Example

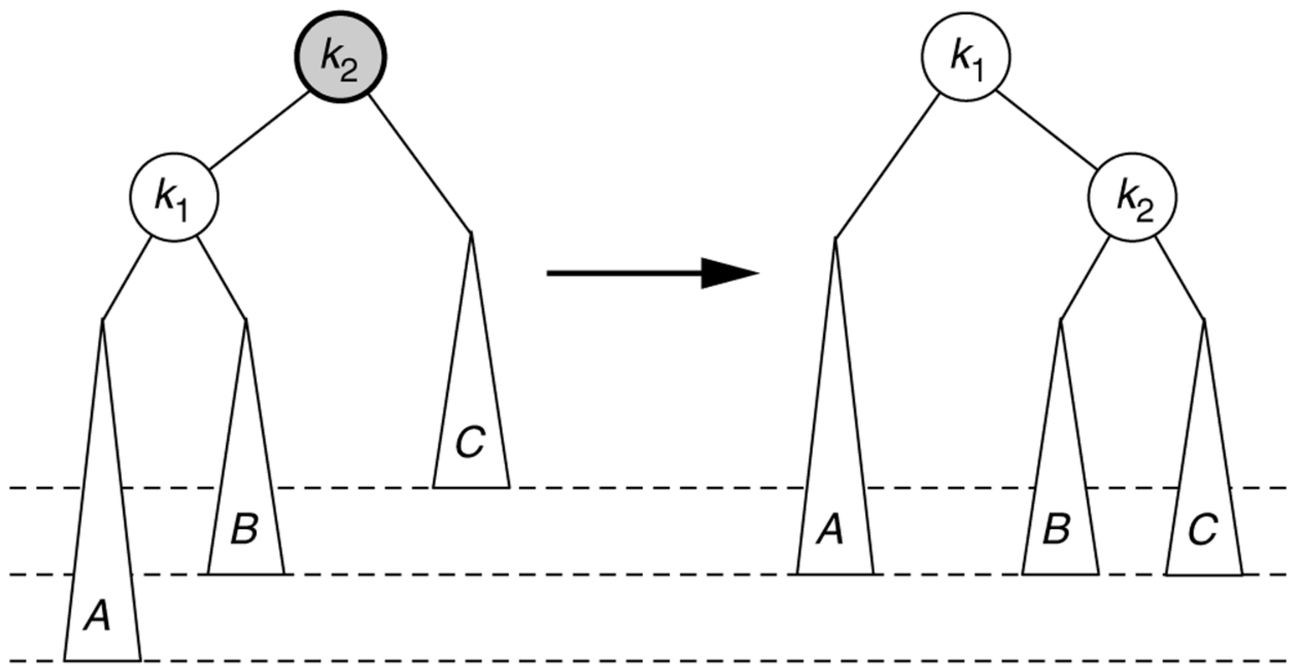
- Insert(6)
- Insert(3)
- Insert(1)



- Third insertion violates balance property
  - happens to be at the root
- What is the only way to fix this?

The general right rotation (left-left case)

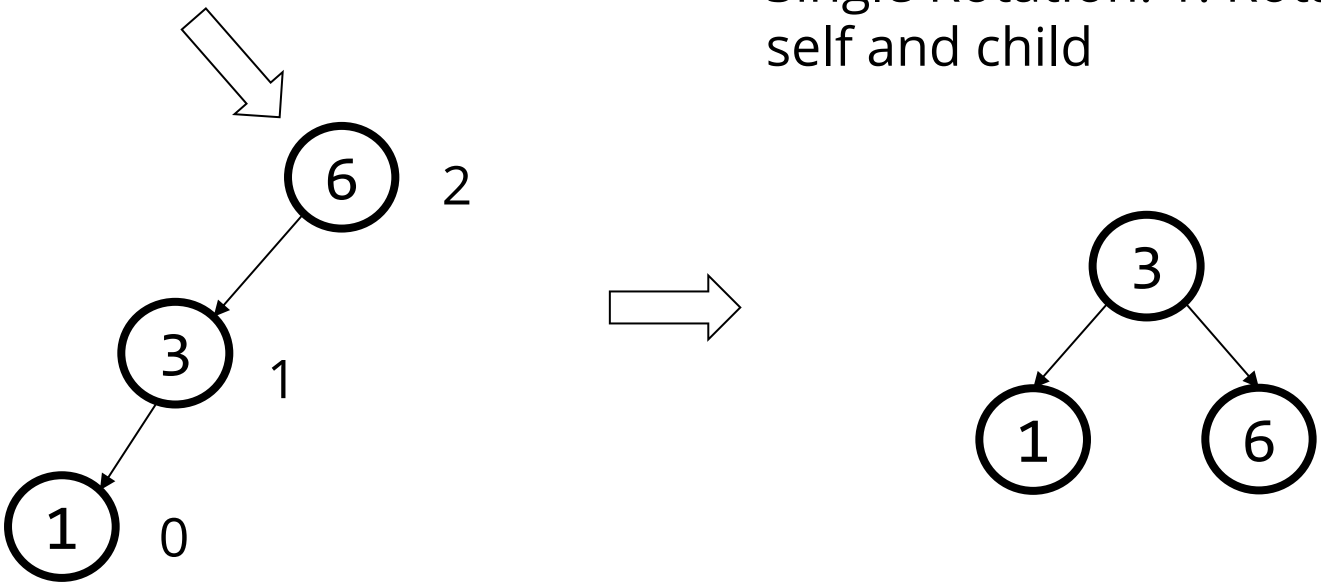
- right rotation (clockwise):
  - left child k1 becomes parent
  - original parent k2 demoted to right
  - k1's original right subtree B (if any) is attached to k2 as left subtree



Fix: Apply “Single Rotation”

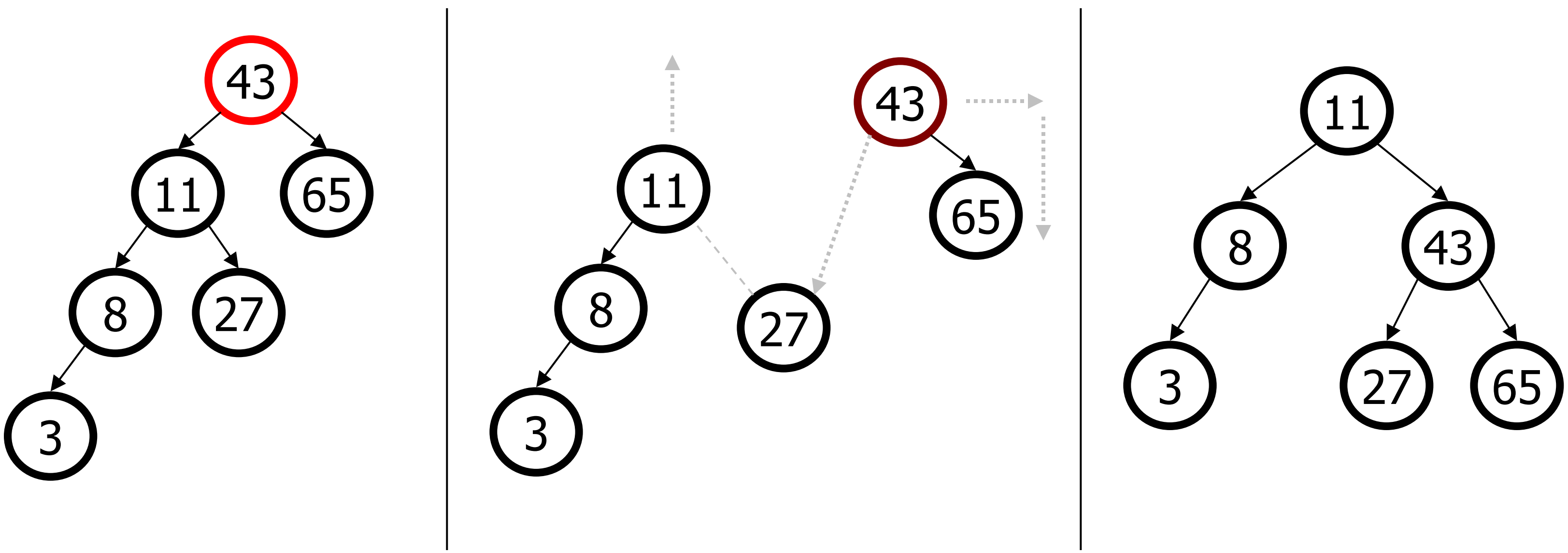
AVL Property violated here

Single Rotation: 1. Rotate between self and child



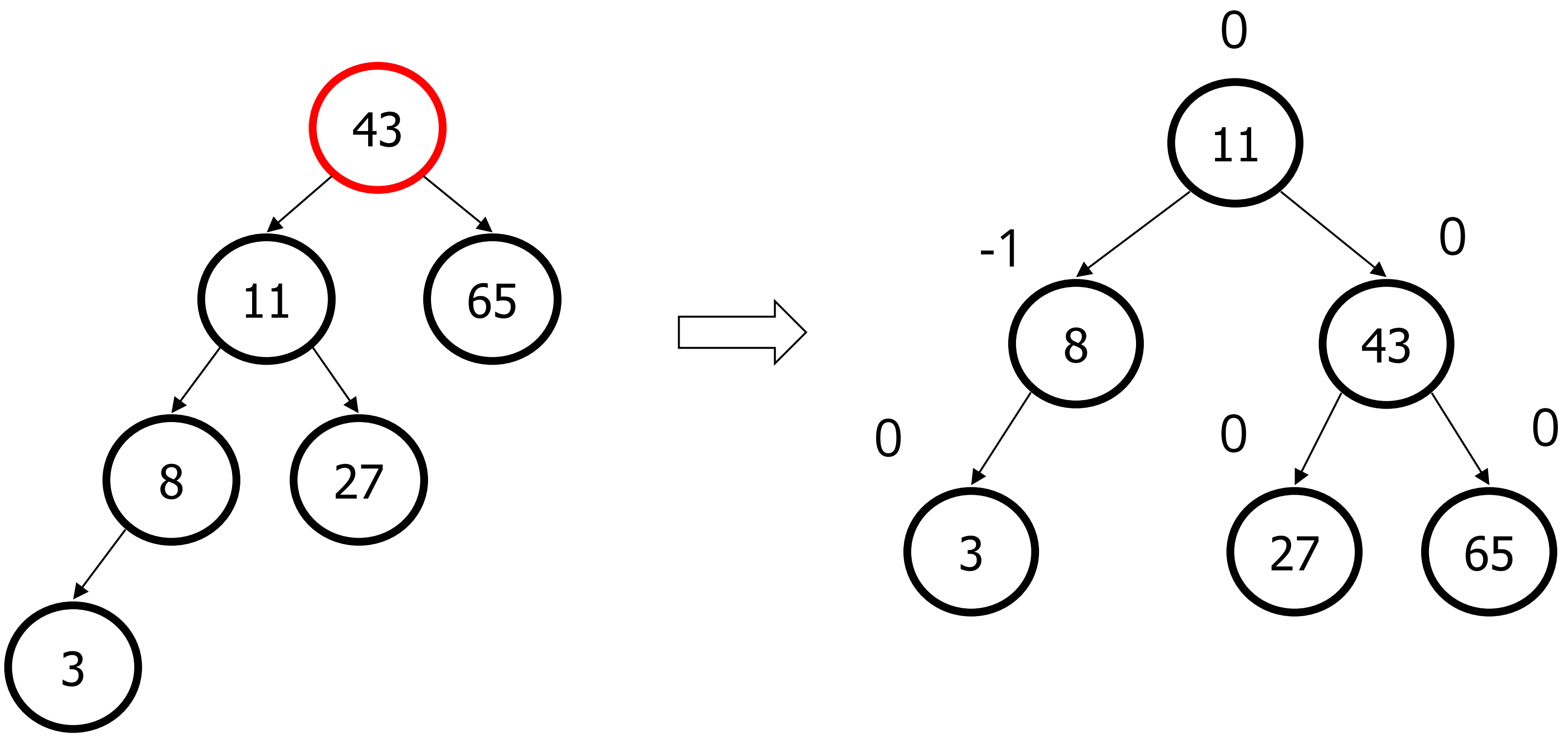
Right rotation steps

1. Detach left child (11)'s right subtree (27) *(don't lose it!)*
2. Consider left child (11) be the new parent.
3. Attach old parent (43) onto right of new parent (11).
4. Attach new parent (11)'s old right subtree (27)
   as left subtree of old parent (43).



Right rotation example

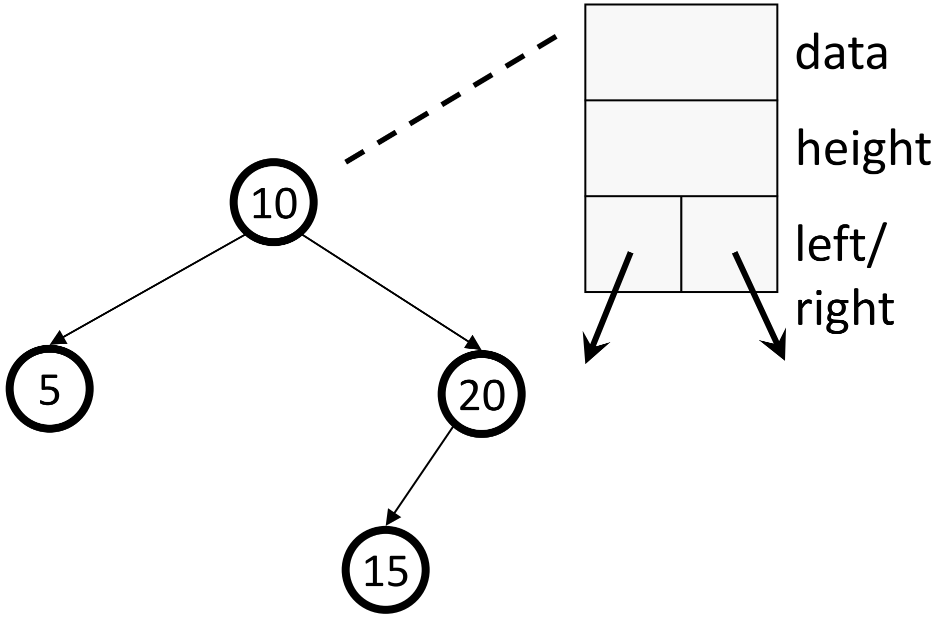
- What is the balance factor of the nodes after rotation?



Tracking subtree height

- Many of the AVL tree operations depend on height.
- Height can be computed recursively by walking the tree.
- Instead, each node can keep track of its subtree height as a field:

```
private class TreeNode {
  private int data;
  private int height;
  private TreeNode left;
  private TreeNode right;
}
```



Right rotation code

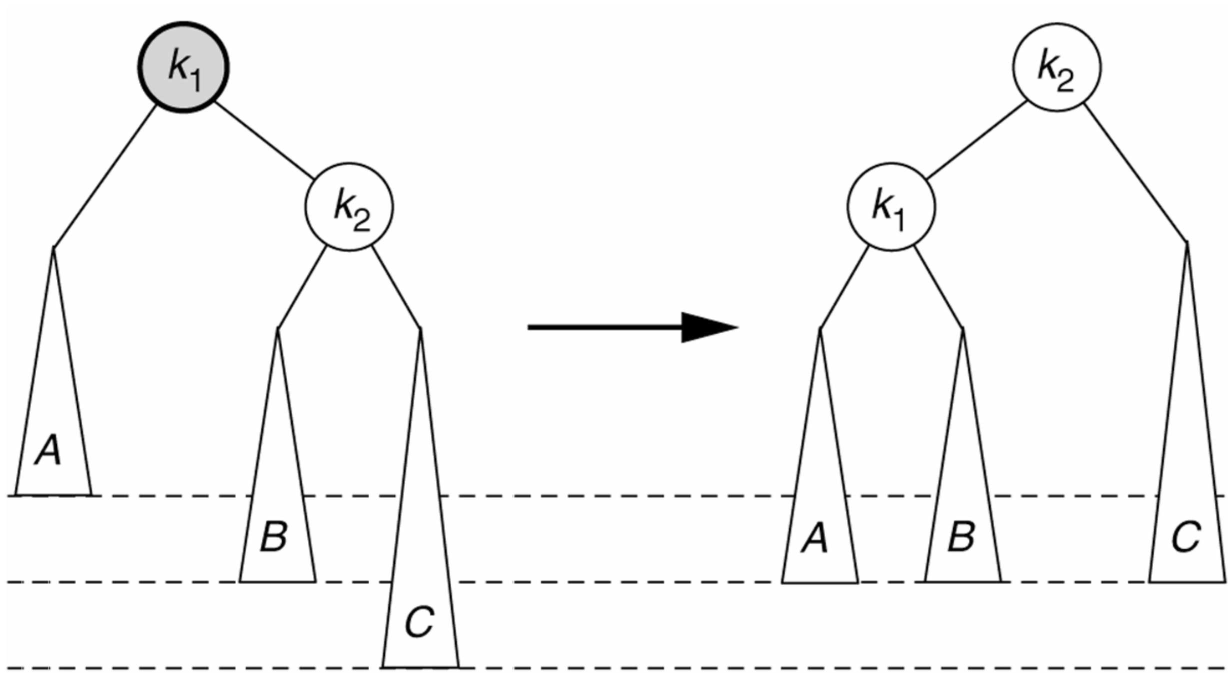
```
private TreeNode rightRotate(TreeNode oldParent) {
    // 1. detach left child's right subtree
    TreeNode orphan = oldParent.left.right;

    // 2. consider left child to be the new parent
    TreeNode newParent = oldParent.left;

    // 3. attach old parent onto right of new parent
    newParent.right = oldParent;

    // 4. attach new parent's old right subtree as
    //    left subtree of old parent
    oldParent.left = orphan;

    oldParent.height = height(oldParent);  // update nodes'
    newParent.height = height(newParent);  // height values

    return newParent;
}
```

Right rotation code

```
private int height(TreeNode node) {
    if (node == null) {
        return 0;
    }
    int left  = (node.left  == null) ? 0 : node.left.height;
    int right = (node.right == null) ? 0 : node.right.height;
    return 1 + Math.max(left, right);
}
```

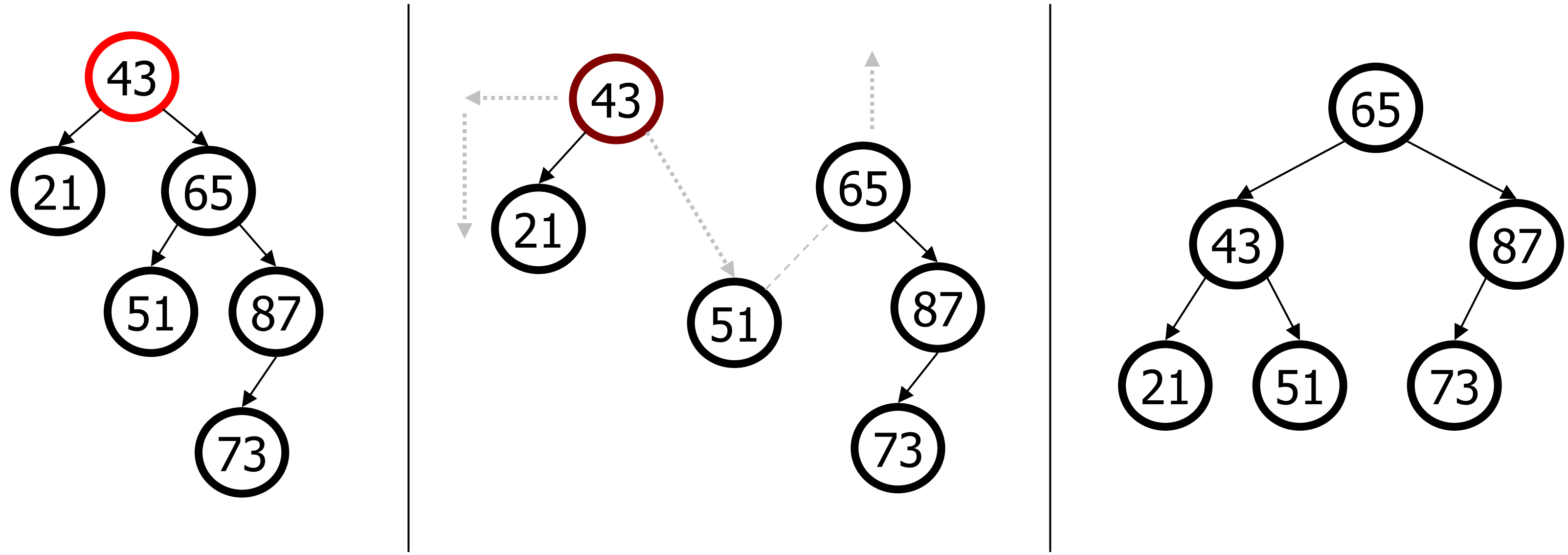
The general right-right case (left rotation)

- **left rotation** (counter-clockwise):
  - right child k_2 becomes parent
  - original parent k_1 demoted to left
  - k_2's original left subtree B (if any) is attached to k_1 as left subtree



Left rotation steps

1. Detach right child (65)'s left subtree (51) *(don't lose it!)*
2. Consider right child (65) be the new parent.
3. Attach old parent (43) onto left of new parent (65).
4. Attach new parent (65)'s old left subtree (51)
   as right subtree of old parent (43).

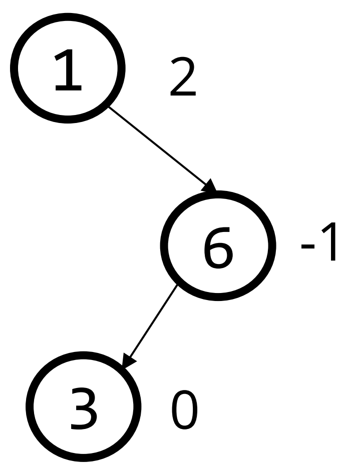


Left rotation code

```
private TreeNode leftRotate(TreeNode oldParent) {
    // 1. detach right child's left subtree
    TreeNode orphan = oldParent.right.left;

    // 2. consider right child to be the new parent
    TreeNode newParent = oldParent.right;

    // 3. attach old parent onto left of new parent
    newParent.left = oldParent;

    // 4. attach new parent's old left subtree as
    //    right subtree of old parent
    oldParent.right = orphan;

    oldParent.height = height(oldParent);  // update nodes'
    newParent.height = height(newParent);  // height values

    return newParent;
}
```

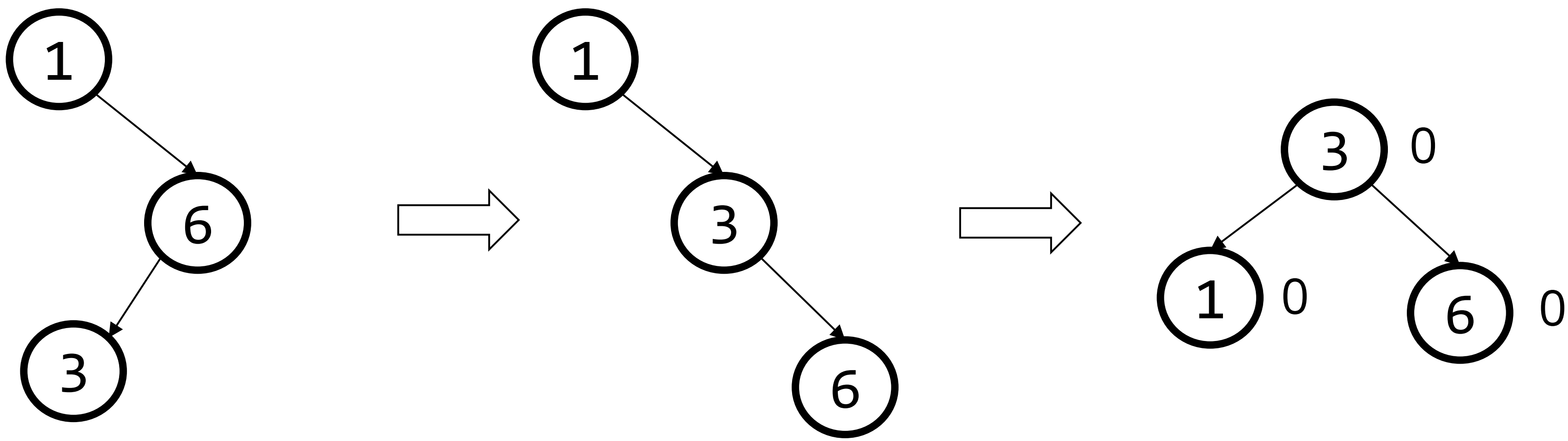
Double rotation

- Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree
- Simple example: insert(1), insert(6), insert(3)



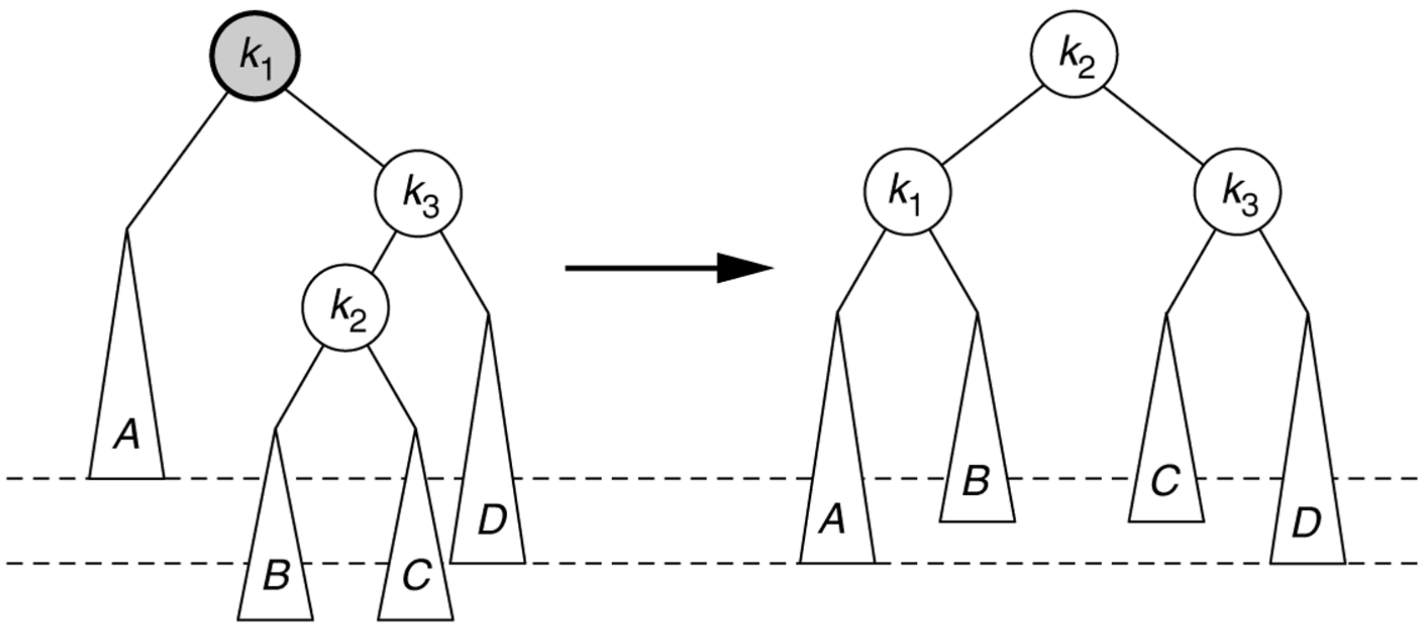
Double rotation

- Double rotation:
  - Rotate problematic child and grandchild
  - Then rotate between self and new child



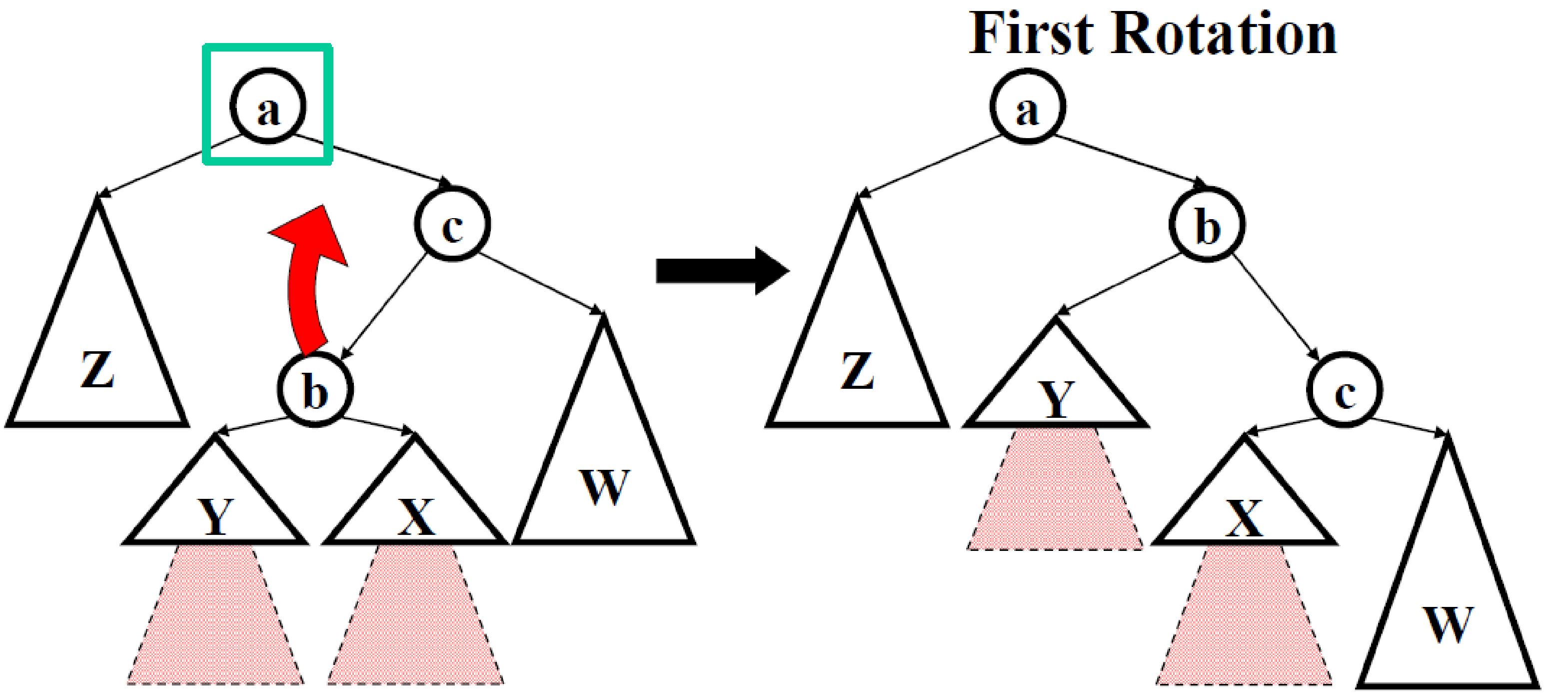
Right-left double rotation

- **right-left double rotation**: *(fixes Case 3 (RL))*
  - 1) right-rotate k_1's right child ... reduces Case 3 into Case 4
  - 2) left-rotate k_1 to fix Case 4



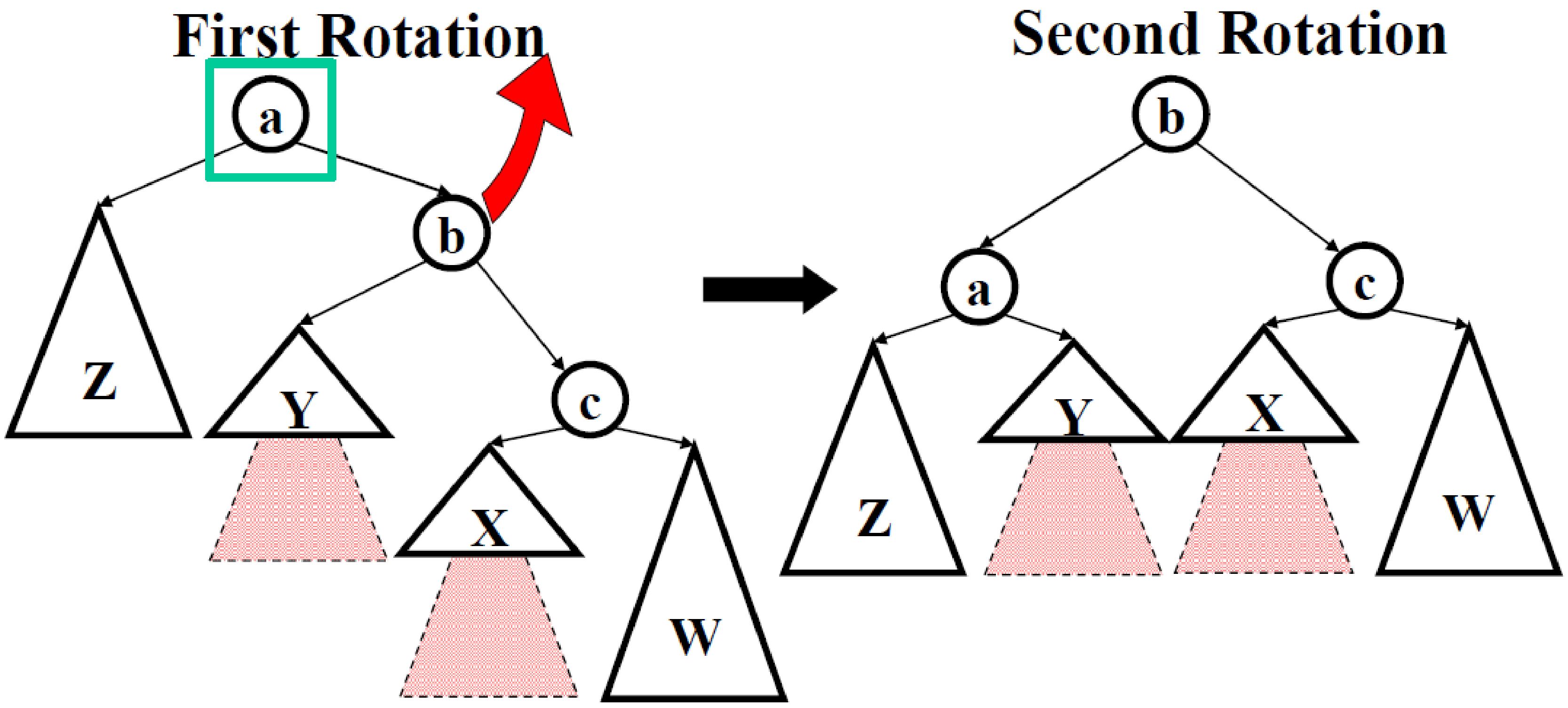
Double Rotation (right-left)

- First Rotation



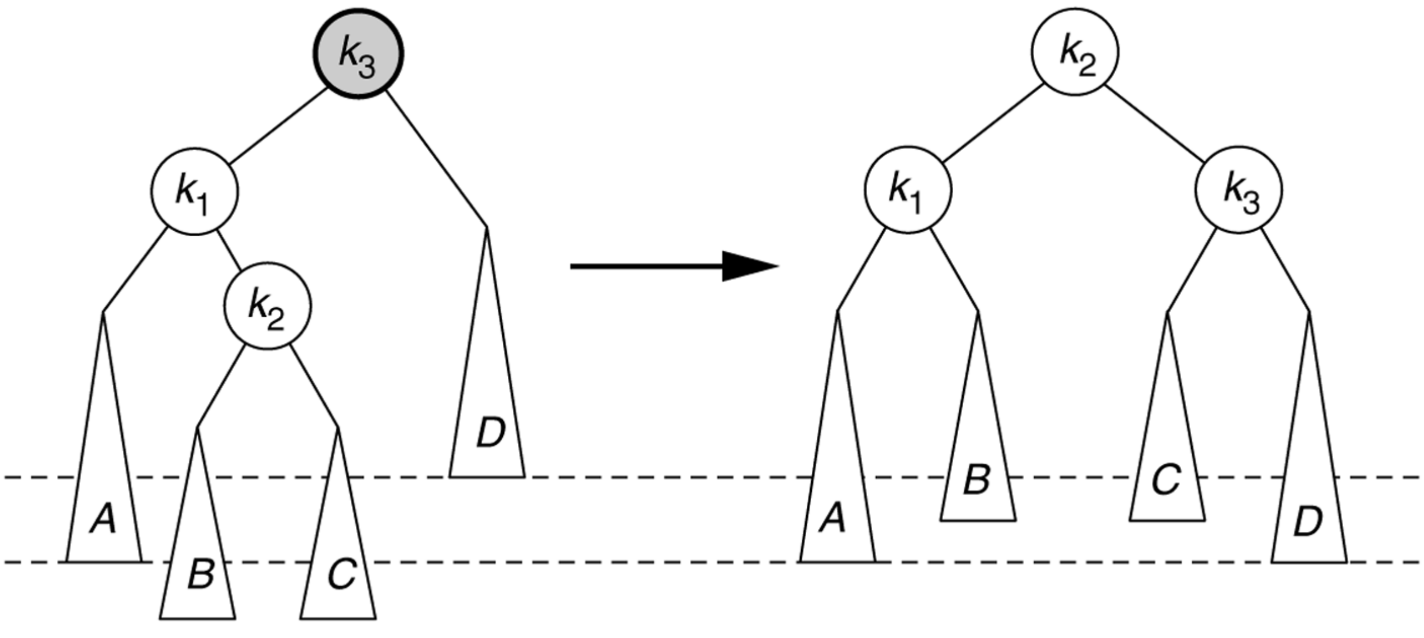
Double Rotation (right-left)

- Second Rotation



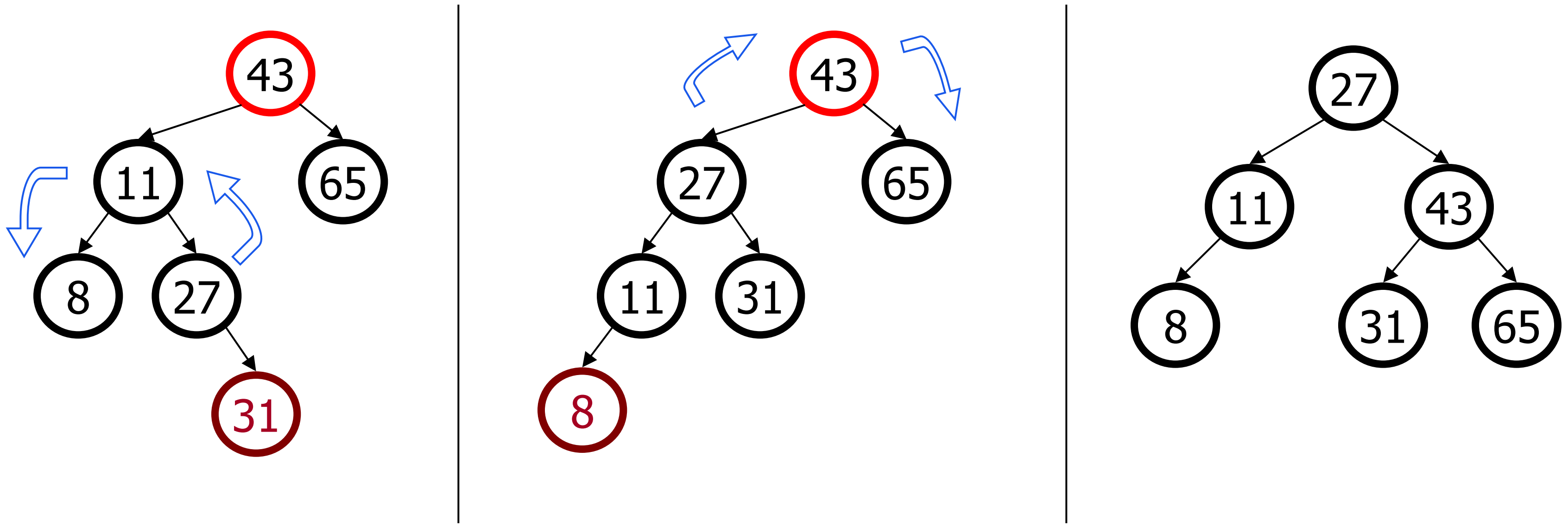
Left-right double rotation

- **left-right double rotation**: *(fixes Case 2 (LR))*
  - 1) left-rotate k_3's left child … reduces Case 2 into Case 1
  - 2) right-rotate k_3 to fix Case 1



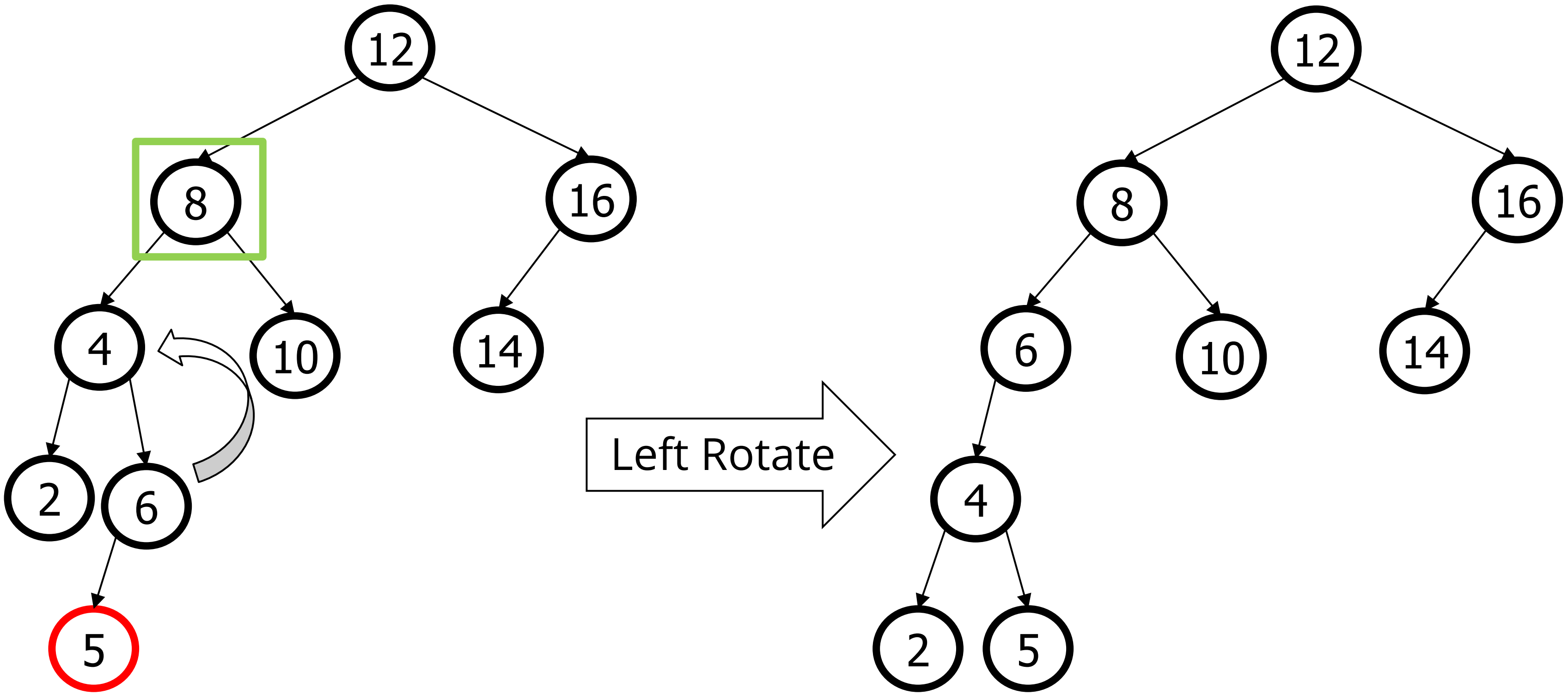
Left-right rotation steps

1. Left-rotate the overall parent's left child (11).
   - This reduces Case 2 (LR) to Case 1 (LL).
2. Right-rotate the overall parent (43).
   - This repairs Case 1 to be balanced.



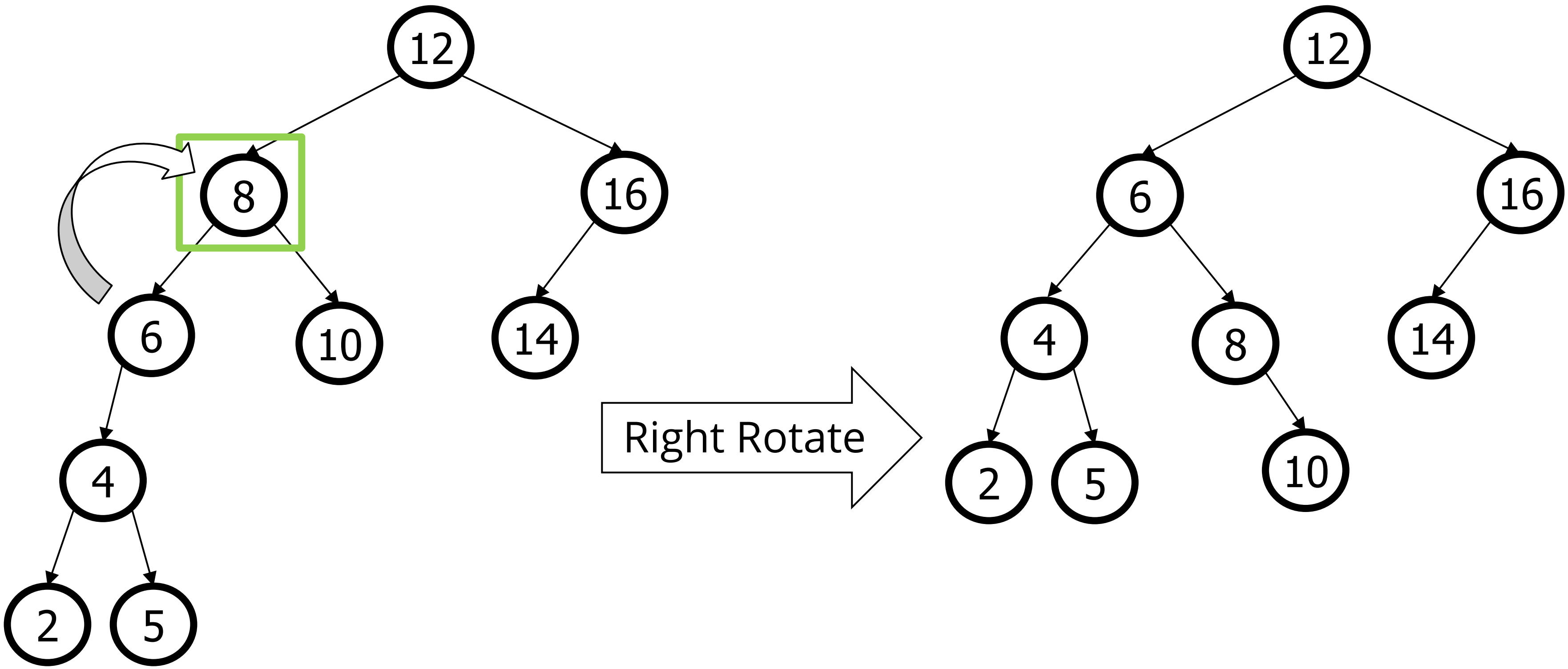
Left-right rotation example

First rotation - Left-rotate the overall parent's left child



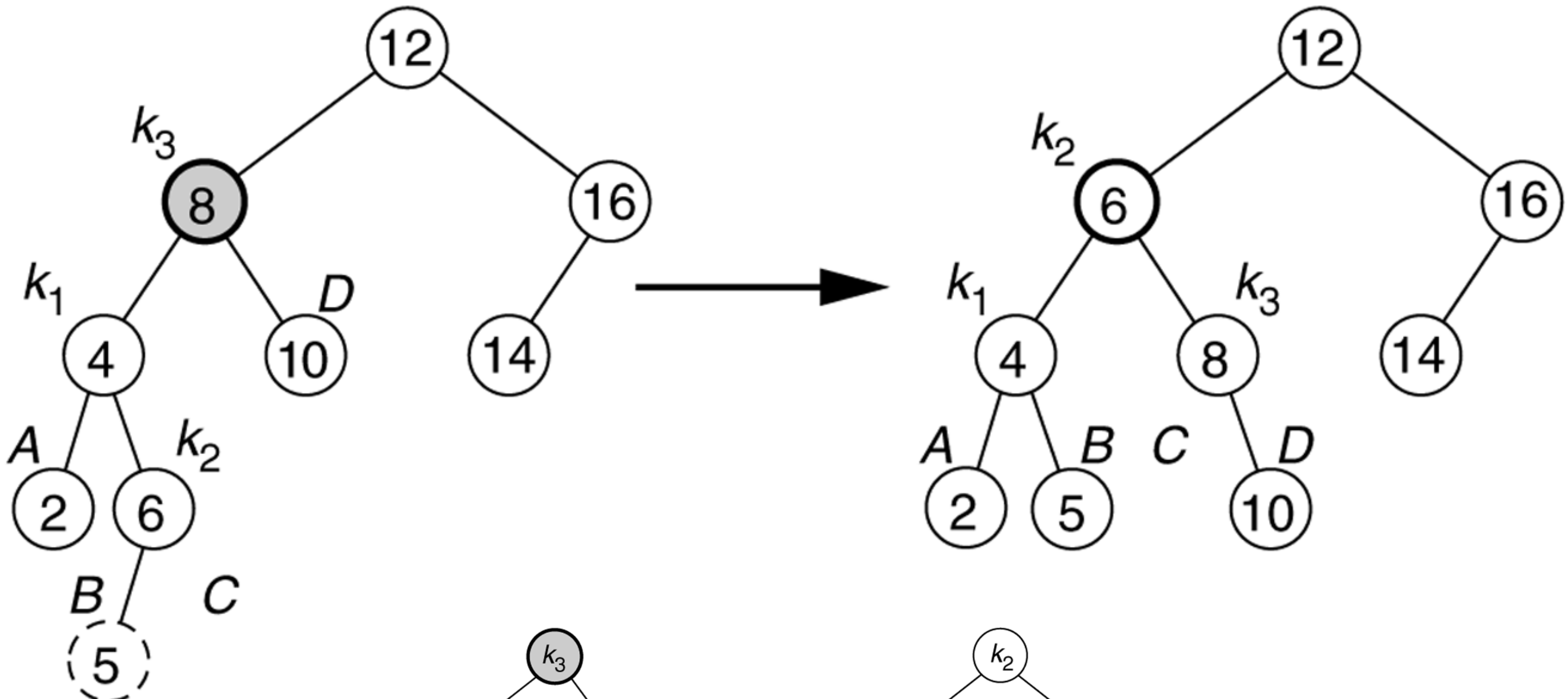
Left-right rotation example

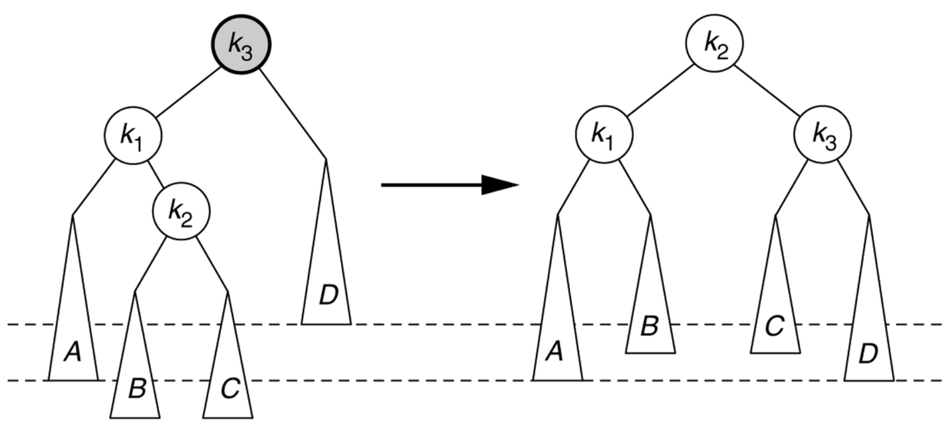
Second rotation - Right-rotate the overall parent



Left-right rotation example

- What is the balance factor of k_1, k_2, k_3 before and after rotating?





Implementing add

- Perform normal BST add. But as recursive calls return, update each node's height from new leaf back up to root.
  - If a node's balance factor becomes +/- 2, rotate to rebalance it.

- How do you know which of the four Cases you are in?
  - Current node BF < -1 $\rightarrow$ LL or LR

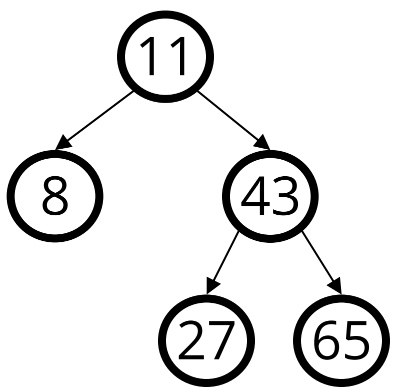
    look at current node's left child BF.
    - left child BF < 0 $\rightarrow$ *fix with R rotation*
    - left child BF > 0 $\rightarrow$ *fix with LR rotations*
  - Current node BF > 1 $\rightarrow$ RL or RR.

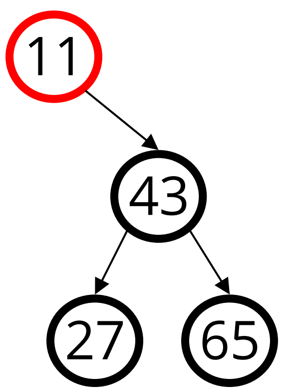
    look at current node's right child BF.
    - right child BF < 0 $\rightarrow$ *fix with RL rotations*
    - right child BF > 0 $\rightarrow$ *fix with L rotation*

AVL remove

- Removing from an AVL tree can also unbalance the tree.
  - Similar cases as with adding: LL, LR, RL, RR
  - Can be handled with the same remedies: rotate R, LR, RL, L

`remove(8)`



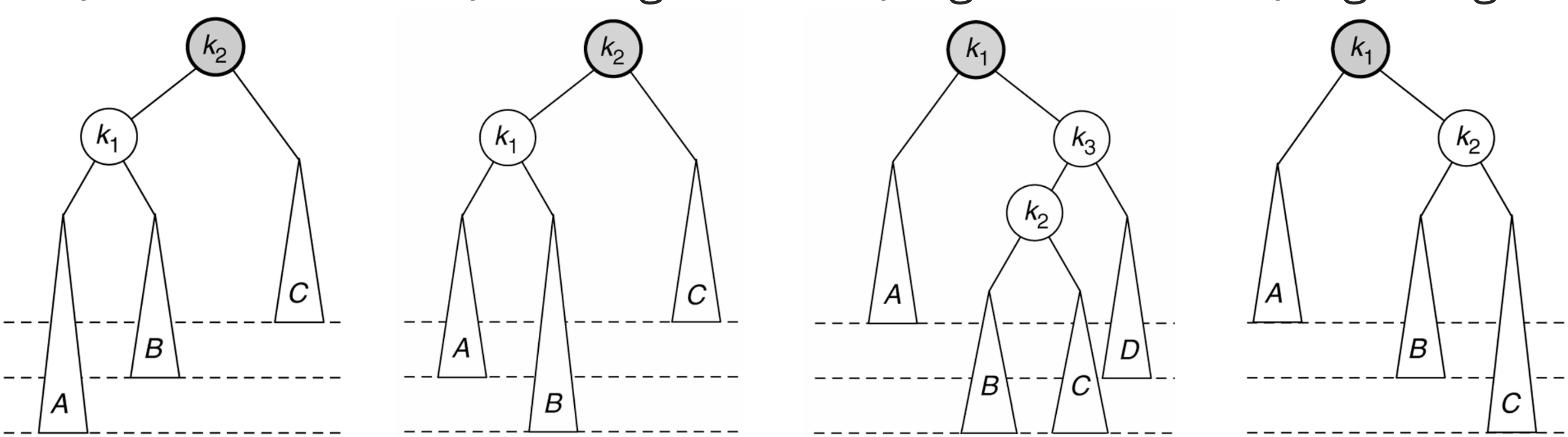


AVL tree insert

- When an add makes a node imbalanced, the child on the imbalanced side always has a balance factor of either -1 or 1.

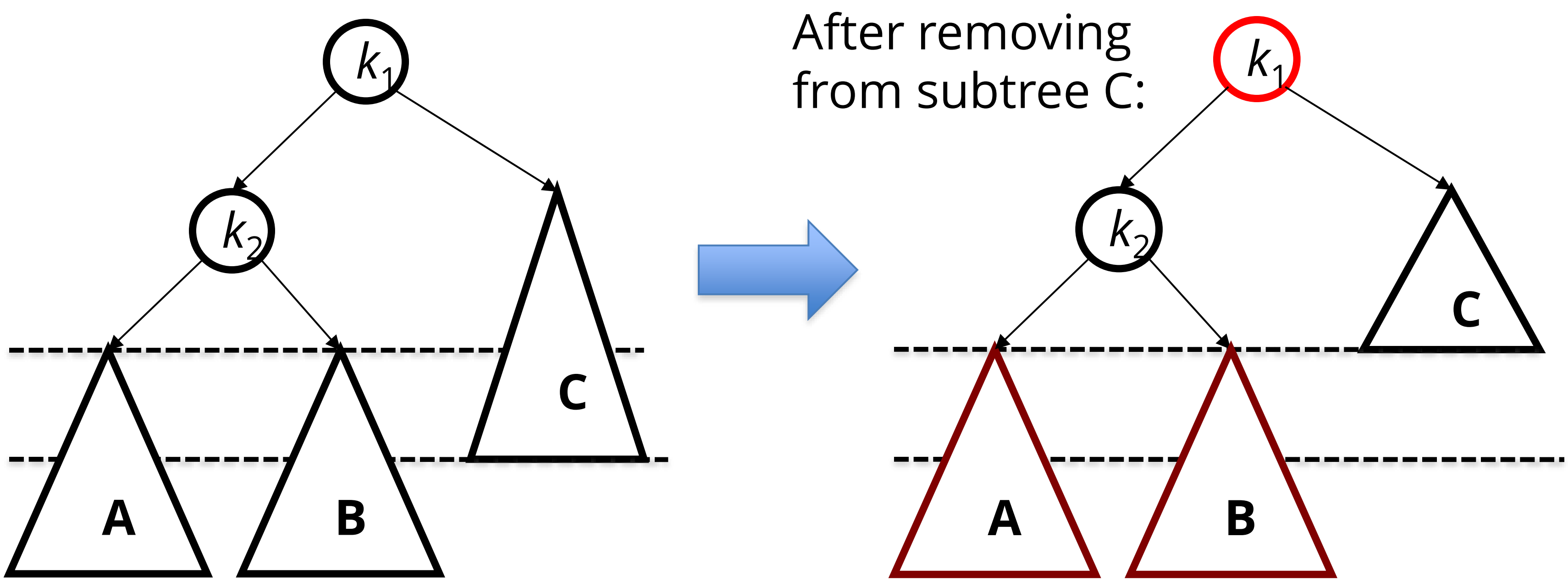
1. *Left-left case*: **left** subtree of the **left** child of b.
2. *Left-right case:* **right** subtree of the **left** child of b.
3. *Right-left case:* **left** subtree of the **right** child of b.
4. *Right-right case:* **right** subtree of the **right** child of b.

1) Left-Left 2) Left-Right 3) Right-Left 4) Right-Right



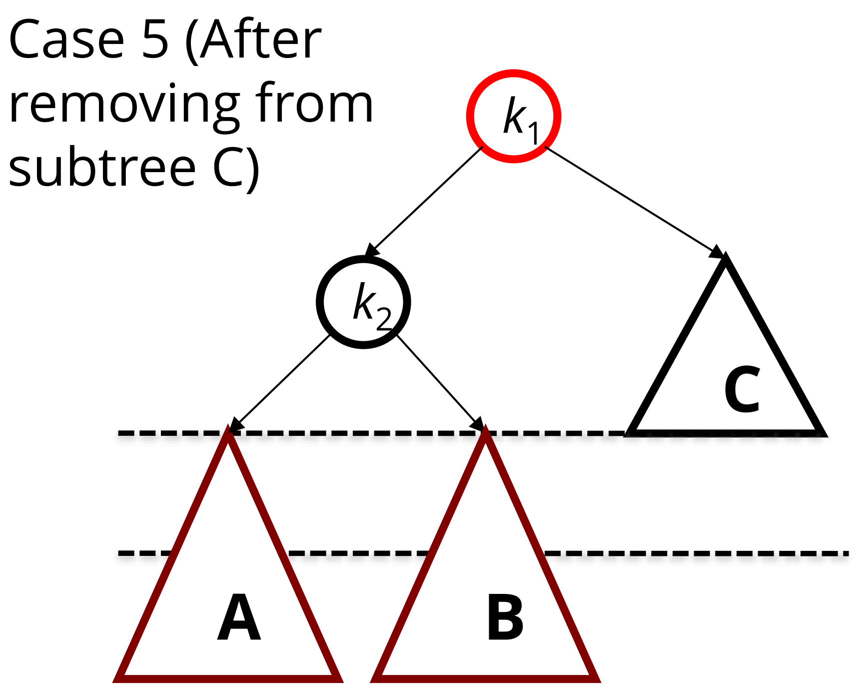
Remove extra cases

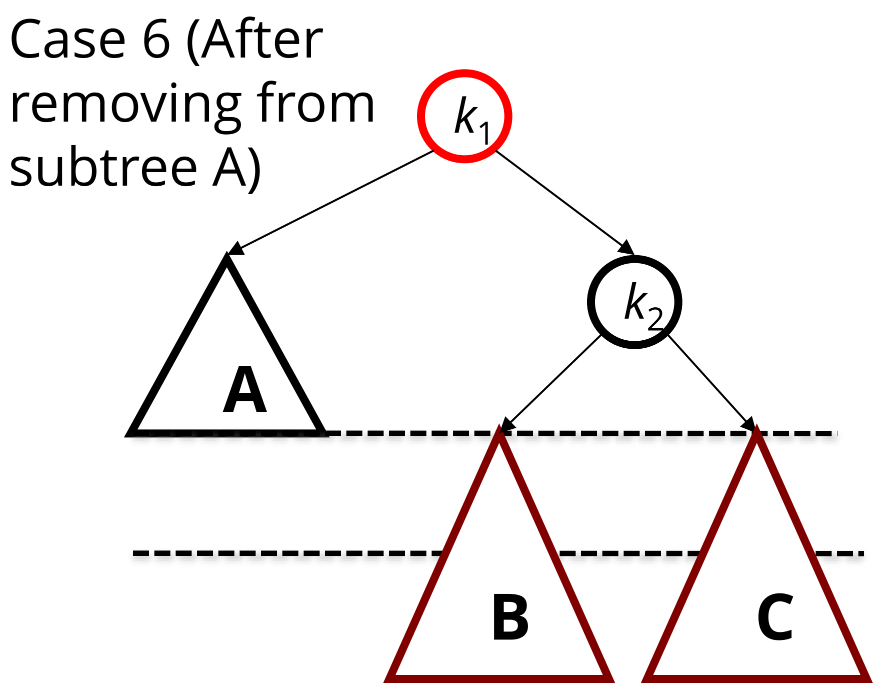
- AVL remove has 2 more cases beyond the 4 from adding:
  - In these cases, the offending subtree has a balance factor of 0. (The cause of imbalance is *both* LL *and* LR relative to k_1 below.)



Labeling the extra cases

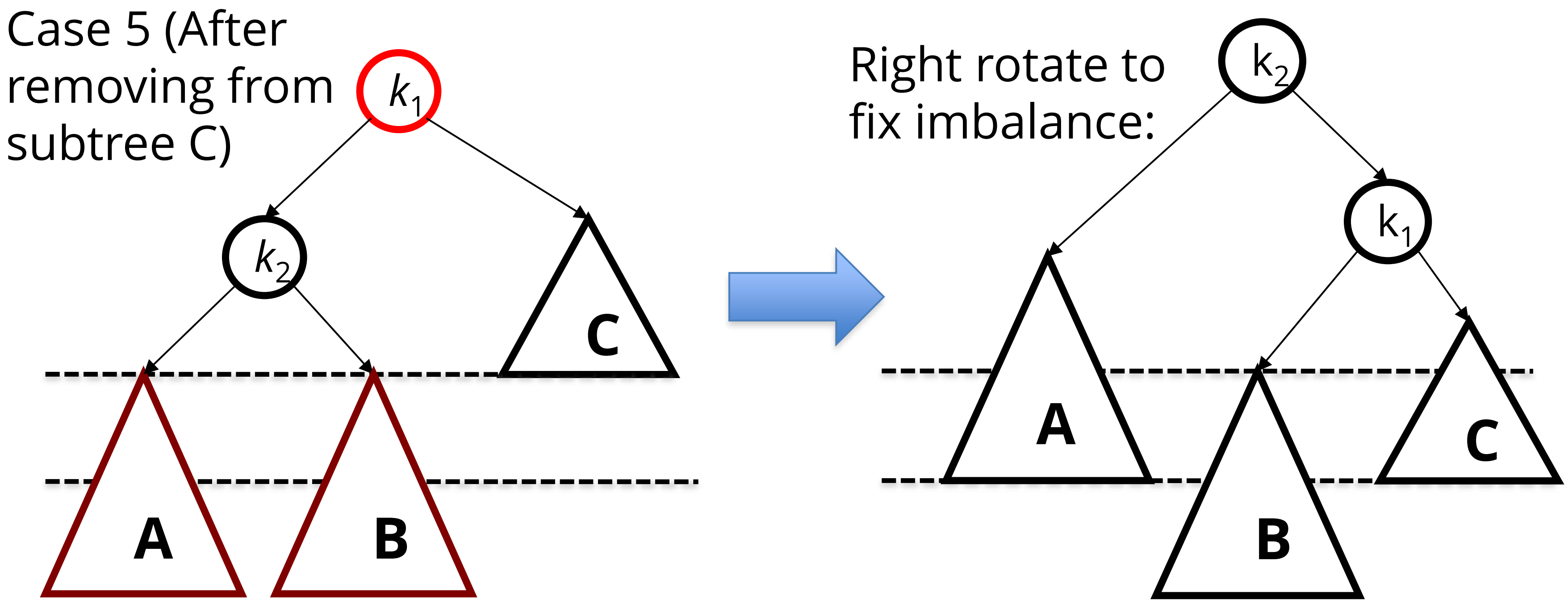
- Let's label these two new cases of remove imbalance:
  - *Case 5:* Problem is in both the LL and LR subtrees of the parent.
  - *Case 6:* Problem is in both the RL and RR subtrees of the parent.





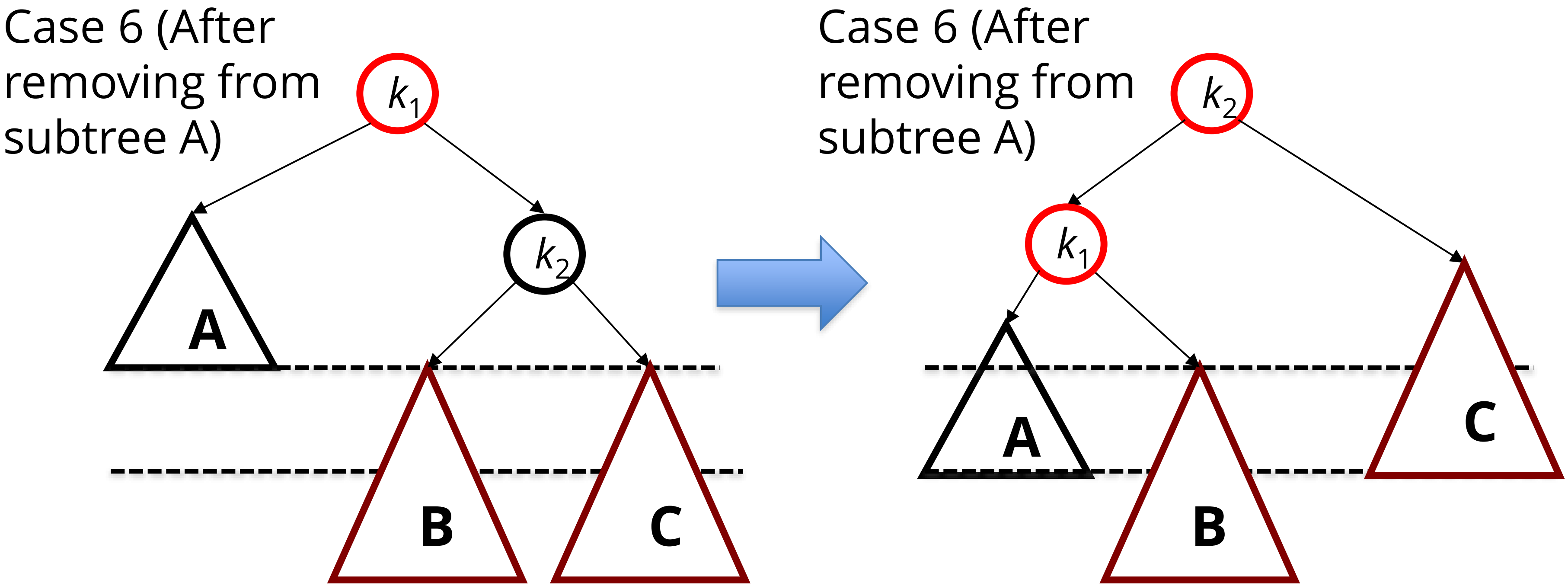
Fixing remove cases

- Each of these new cases can be fixed through a single rotation:
  - To fix Case 5, we right rotate *(left-both)*



Labeling the extra cases

- Each of these new cases can be fixed through a single rotation:
  - To fix Case 6, we left rotate *(right-both)*



Implementing remove

- Perform normal BST remove. But as recursive calls return, update each node's height from new leaf back up to root.
  - If a node's balance factor becomes +/- 2, rotate to rebalance it.
  - Current node BF < -1 $\rightarrow$ LL or LR or L-both.

    look at current node's left child BF.
    - left child BF < 0 $\rightarrow$ *fix with R rotation*
    - left child BF > 0 $\rightarrow$ *fix with LR rotations*
    - left child BF = 0 $\rightarrow$ *fix with R rotation*
  - Current node BF > 1 $\rightarrow$ RL or RR or R-both

    look at current node's right child BF.
    - right child BF < 0 $\rightarrow$ *fix with RL rotations*
    - right child BF > 0 $\rightarrow$ *fix with L rotation*
    - right child BF = 0 $\rightarrow$ *fix with L rotation*

Questions?