CSE 373: Data Structures and Algorithms

Lecture 8: Trees II (AVL Trees)

Introduction

<u>Observation</u>: the shallower the BST the better

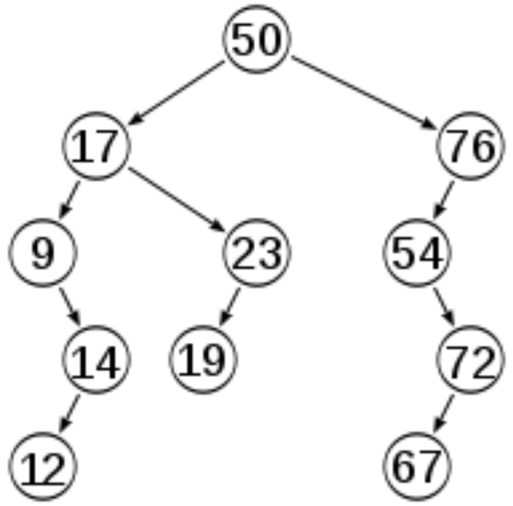
- For a BST with *n* nodes
  - Average case height is $\Theta(\log n)$
  - Worst case height is $\Theta(n)$
- Simple cases such as insert(1, 2, 3, ..., *n*) lead to the worst case scenario: height $\Theta(n)$

<u>Strategy</u>: Don't let the tree get lopsided

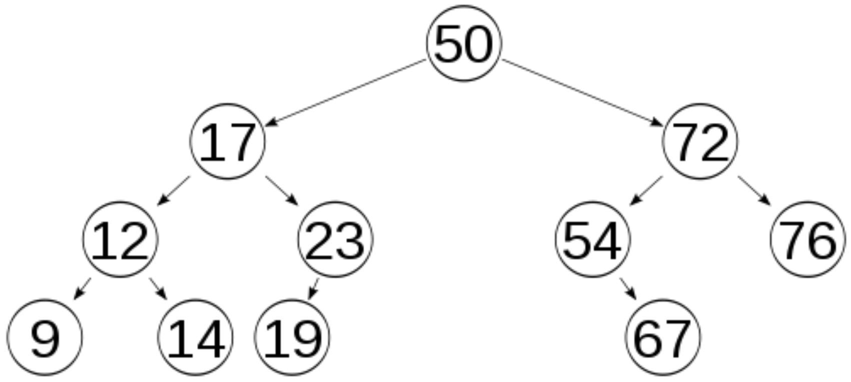
- Constantly monitor balance for each subtree
- Rebalance subtree before going too far astray

Balanced Tree

- **Balanced Tree**: a tree in which heights of subtrees are approximately equal



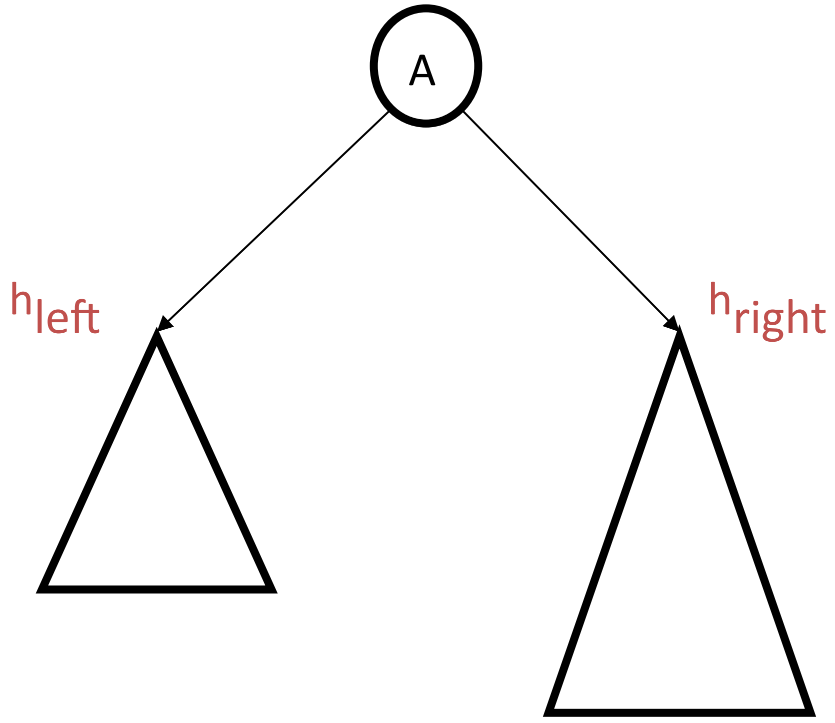
unbalanced tree



balanced tree

Tree height calculation

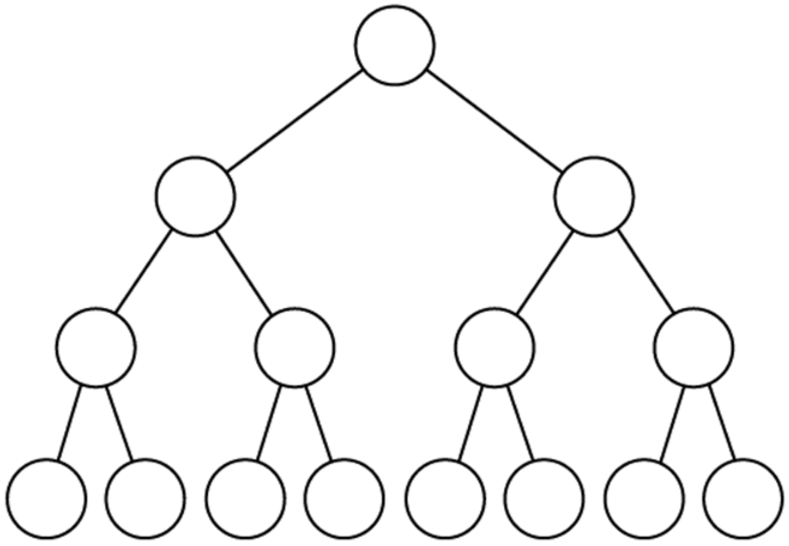
- Height is max number of edges from root to leaf
  - height(null) = -1
  - height(1) = 0
  - height(A)?
    - Hint: it's recursive!

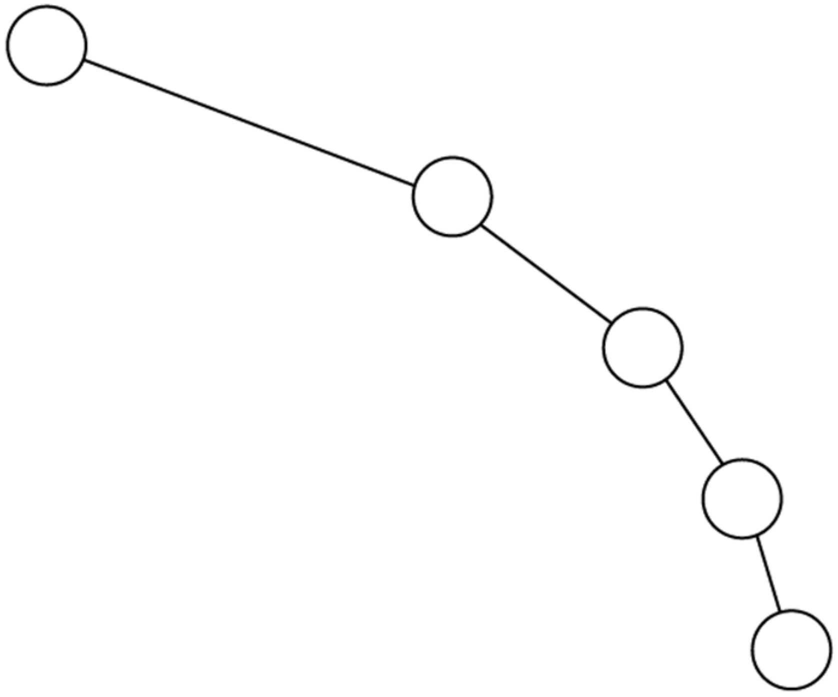


Tree balance and height

(a) The balanced tree has a height of: ____________

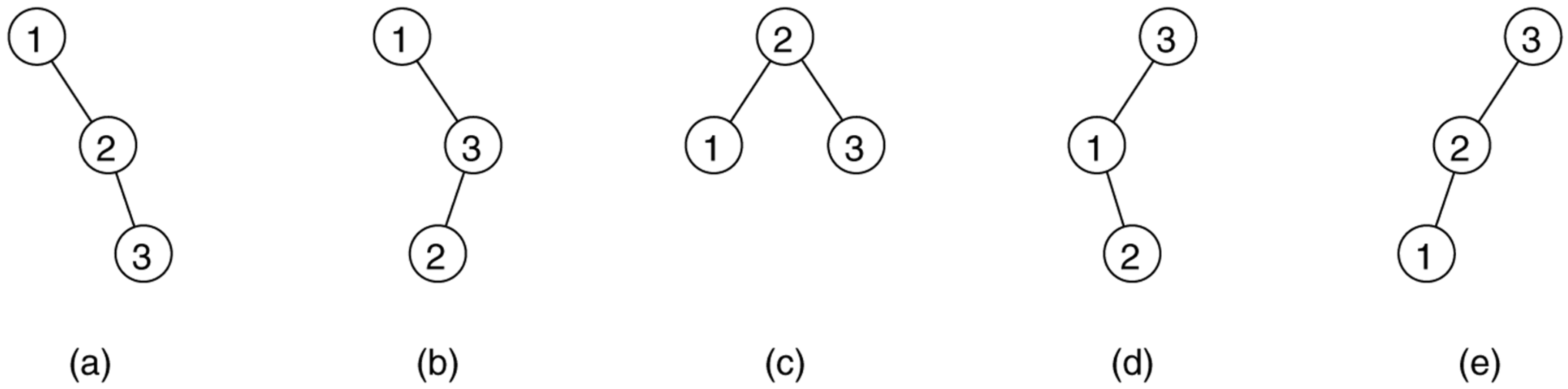
(b) The unbalanced tree has a height of:____________


(a)


(b)

Tree balance: probabilities

binary search trees resulting from adding a random permutation of 1, 2, and 3:



- Which is most likely to occur, given random input?
- Which input orderings are "bad" or "good"?

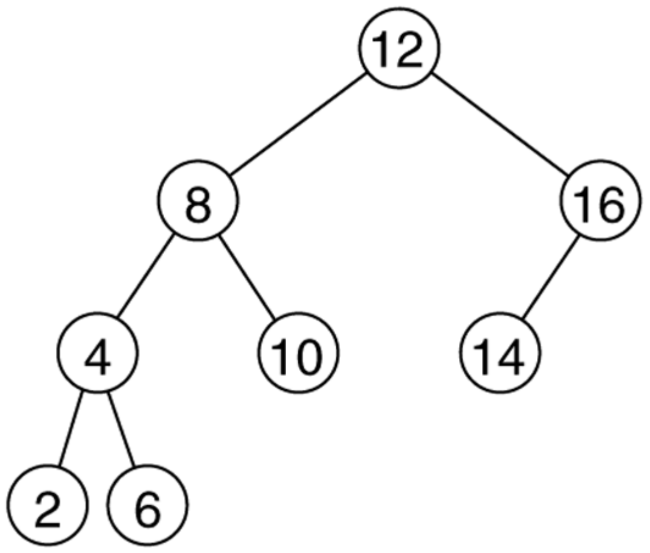
AVL trees

- **AVL tree**: a binary search tree that uses modified add and remove operations to stay balanced as items are added to and remove from it
  - specifically, maintains a balance factor of each node of 0, 1, or -1
    - i.e. no node's two child subtrees differ in height by more than 1
  - invented in 1962 by two Russian mathematicians (Adelson-Velskii and Landis)
  - one of several auto-balancing trees (others in book)

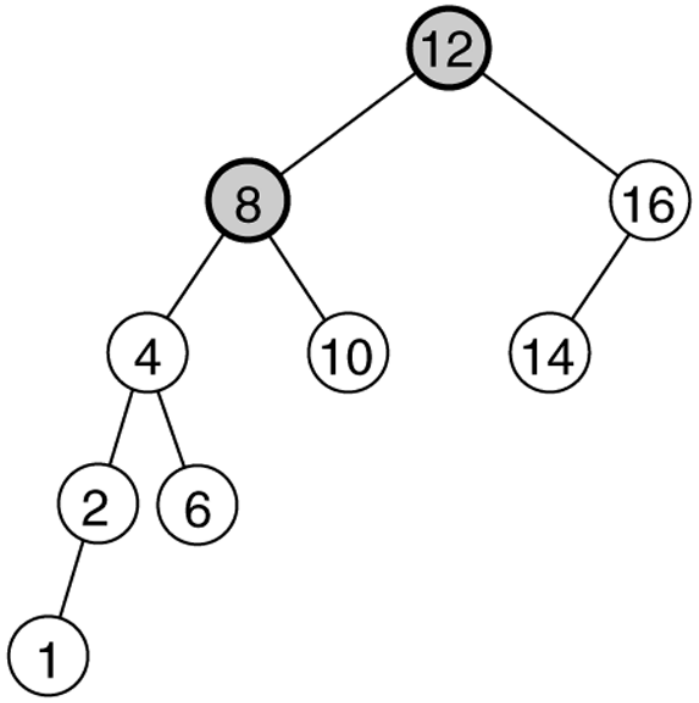
- **balance factor**, for a tree node *n* :
  - height of *n*'s right subtree minus height of *n*'s left subtree
  - $BF_n = Height_{n.right} - Height_{n.left}$
  - start counting heights at n

AVL tree examples

Two binary search trees:
(a) an AVL tree
(b) not an AVL tree (unbalanced nodes are darkened)

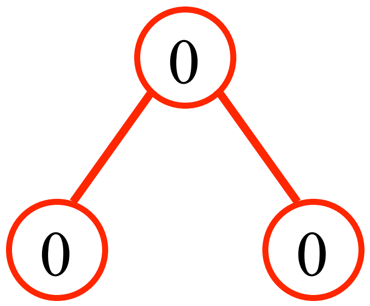


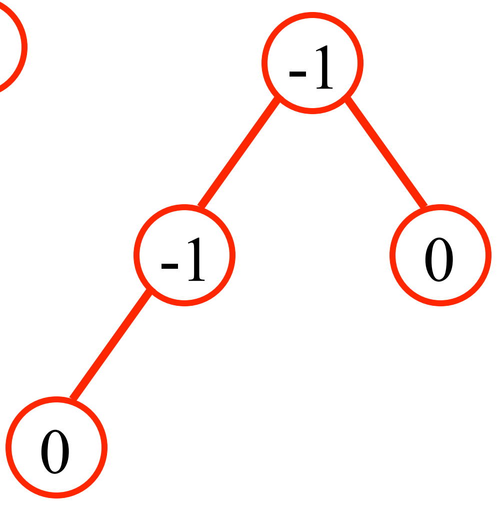
(a)

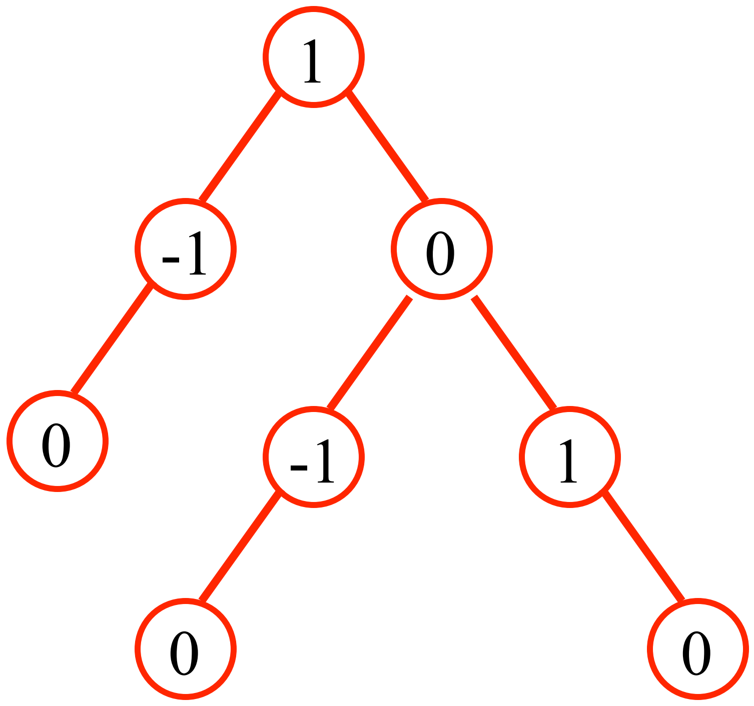


(b)

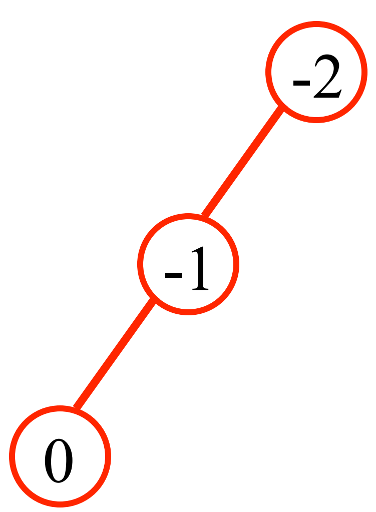
More AVL tree examples

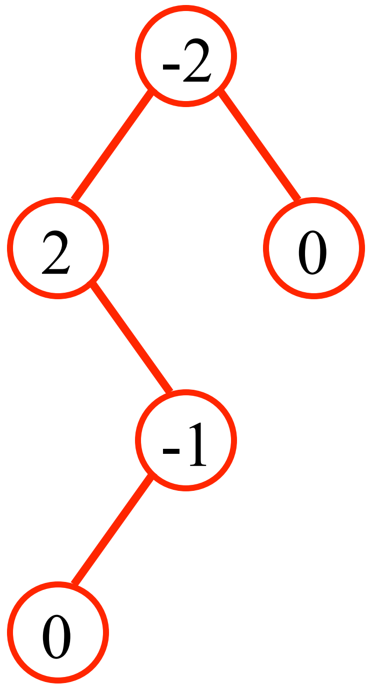


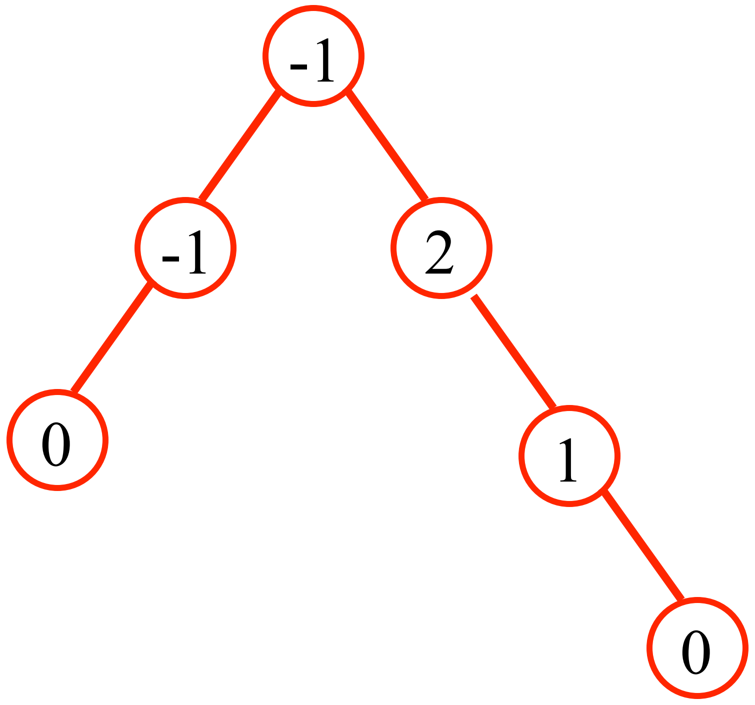




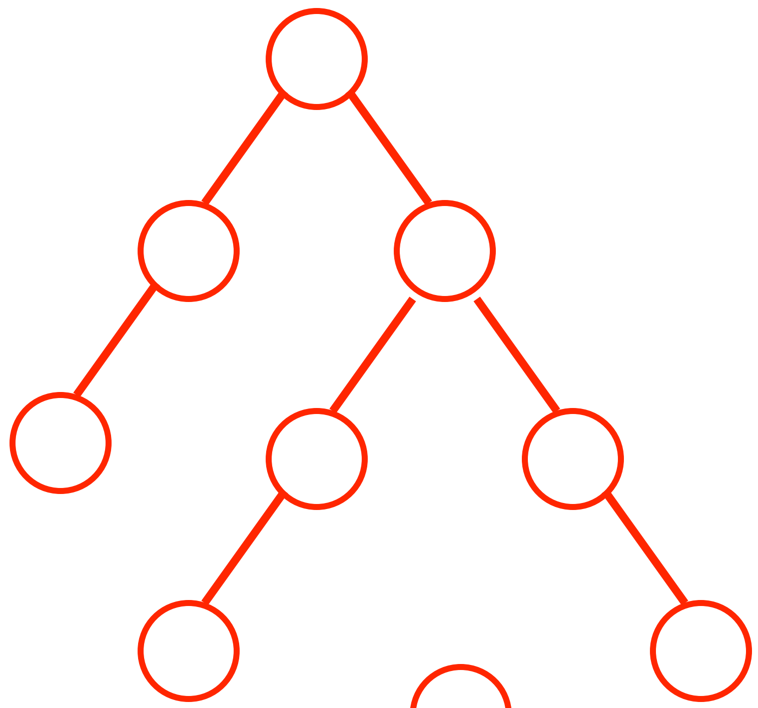
Not AVL tree examples

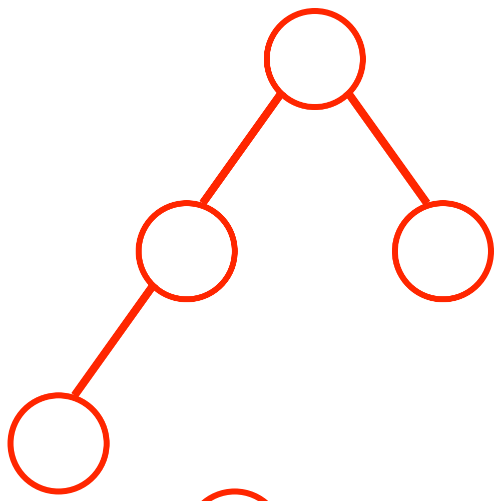


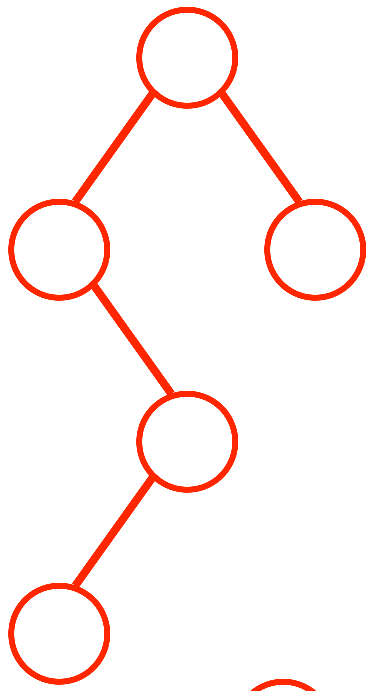


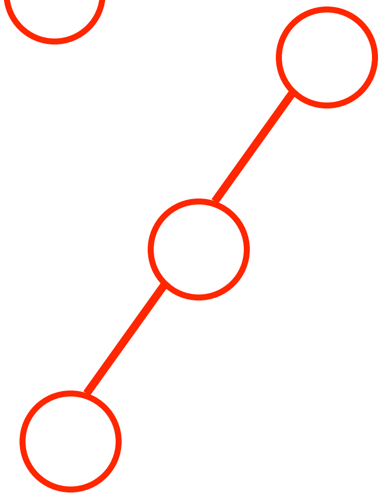


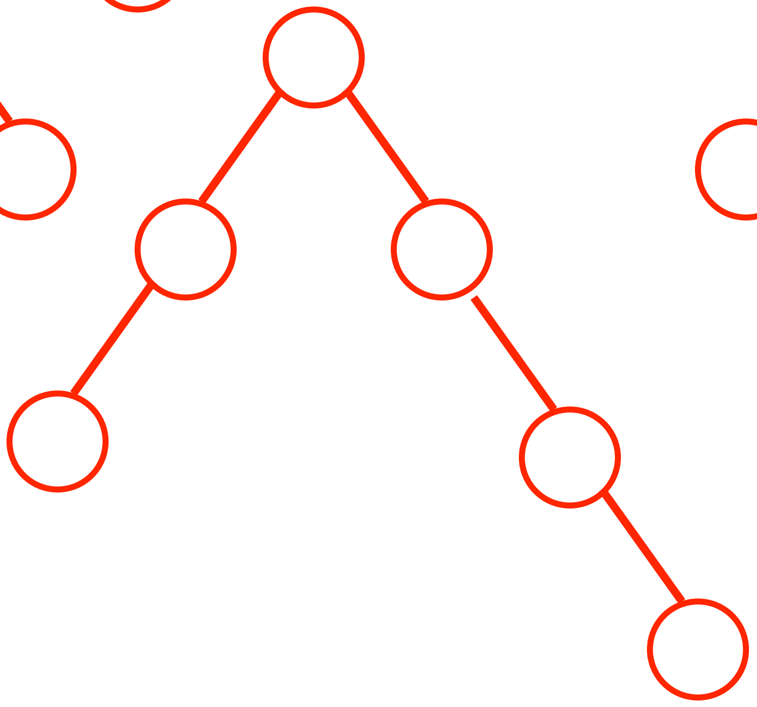
Which are AVL trees?

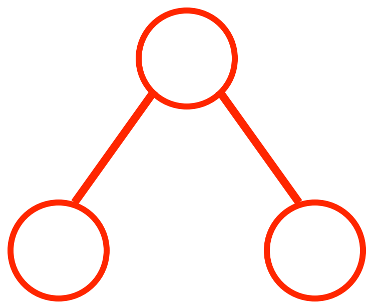




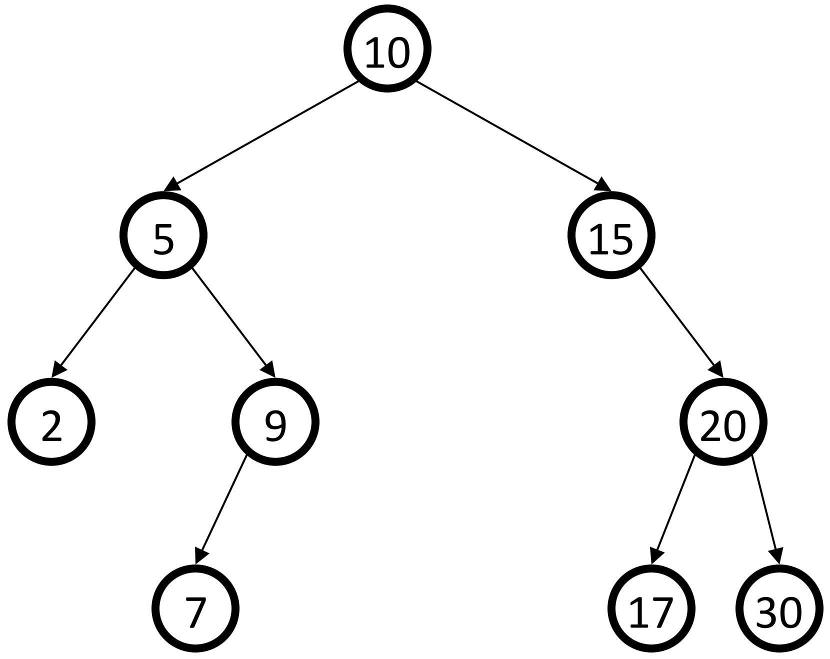








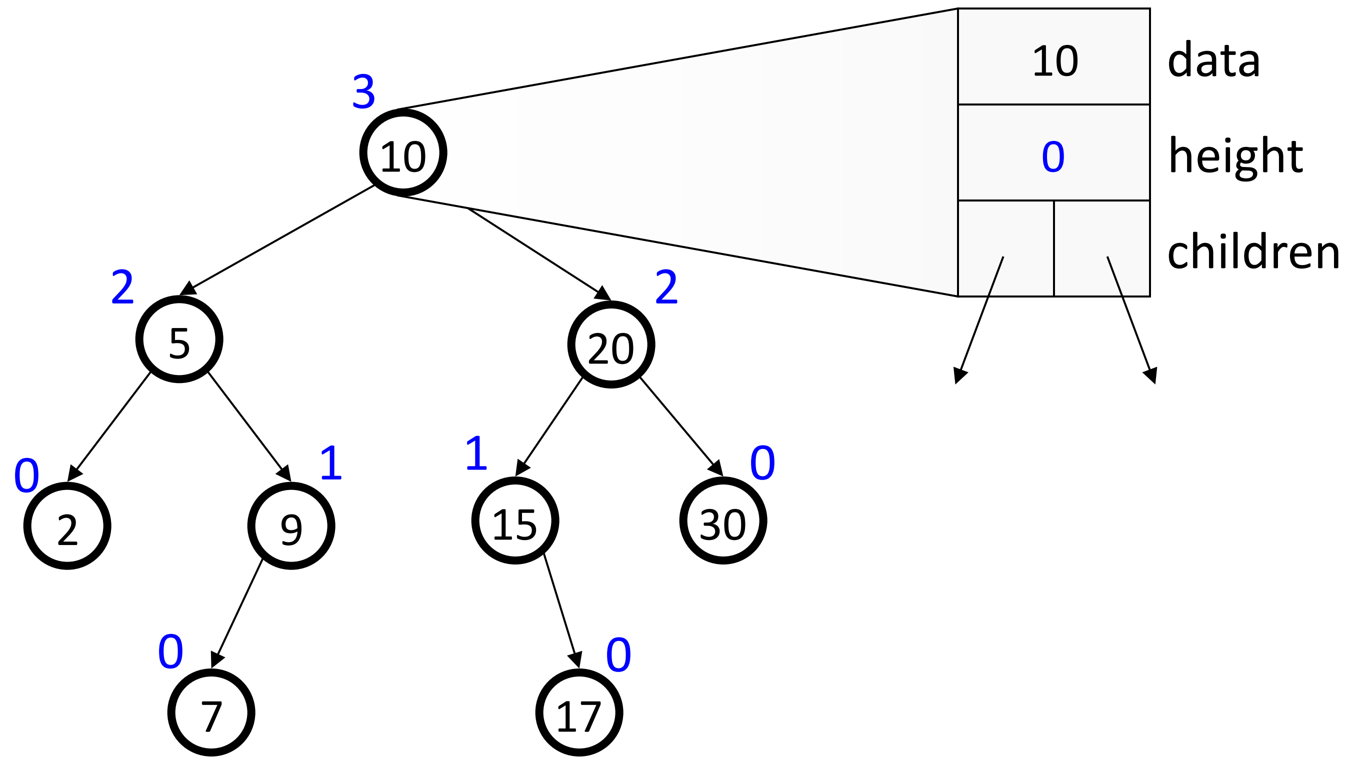
Testing the Balance Property



We need to be able to:

1. Track Balance
2. Detect Imbalance
3. Restore Balance

Tracking Balance

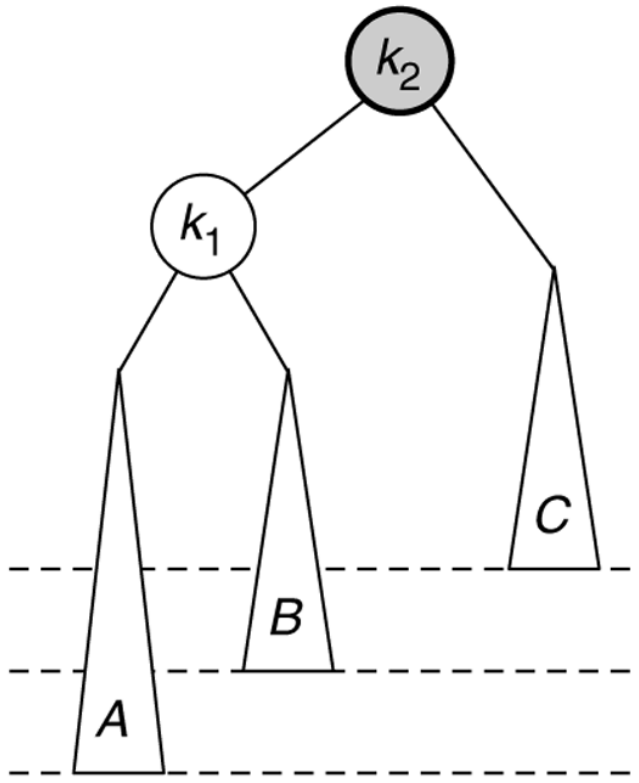


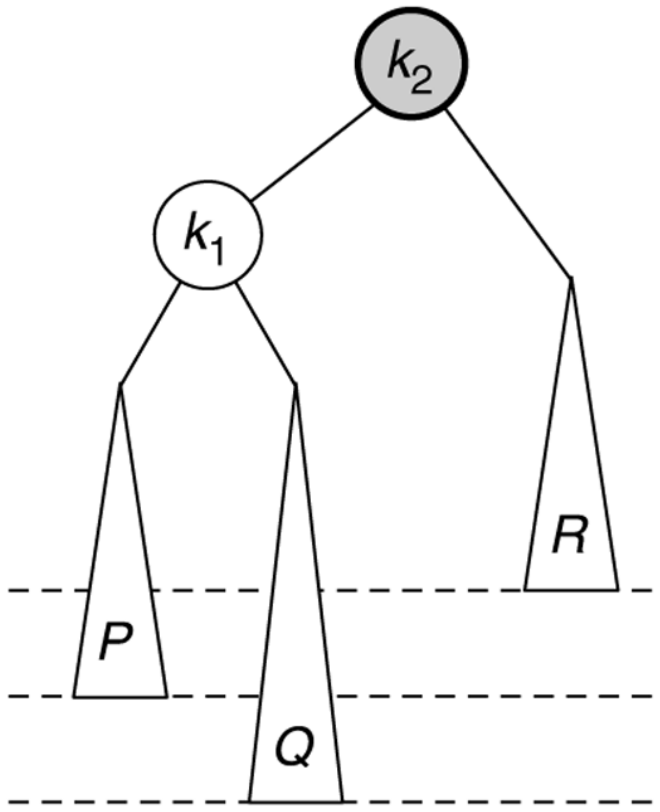
AVL Trees: search, insert, remove

- AVL search:
  - Same as BST search.
- AVL insert:
  - Same as BST insert, *except* you need to check your balance and may need to "fix" the AVL tree after the insert.
- AVL remove:
  - Remove it, check your balance, and fix it.

Problem cases for AVL insert

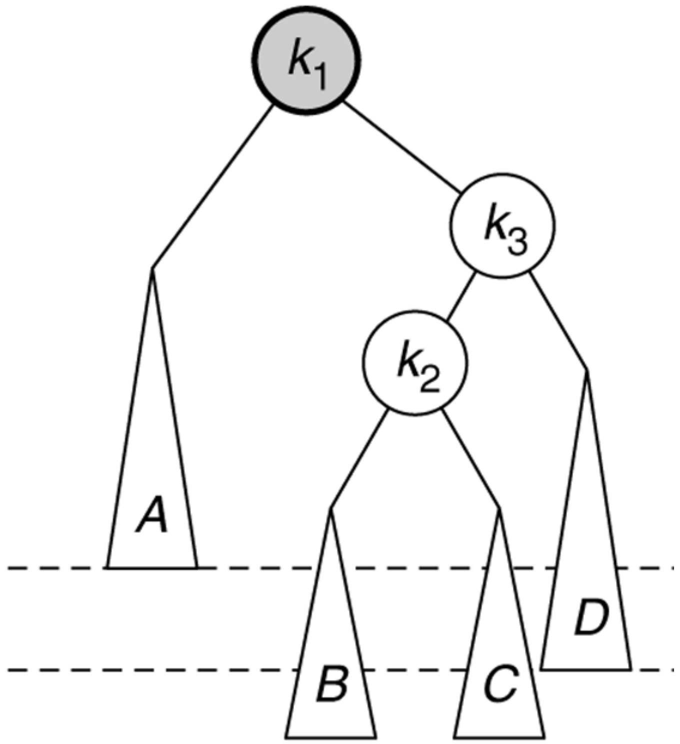
1. LL Case: insertion into left subtree of node's left child
2. LR Case: insertion into right subtree of node's left child

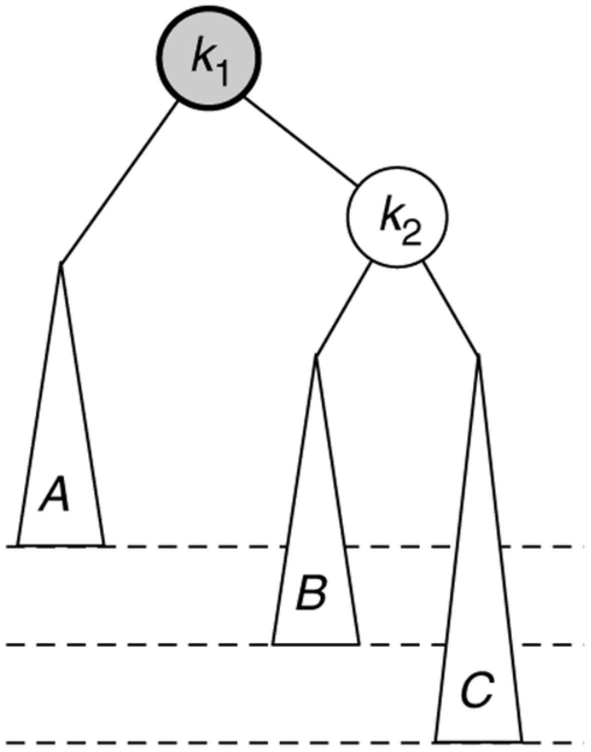




Problem cases for AVL insert, cont.

3. RL Case: insertion into left subtree of node's right child
4. RR Case: insertion into right subtree of node's right child





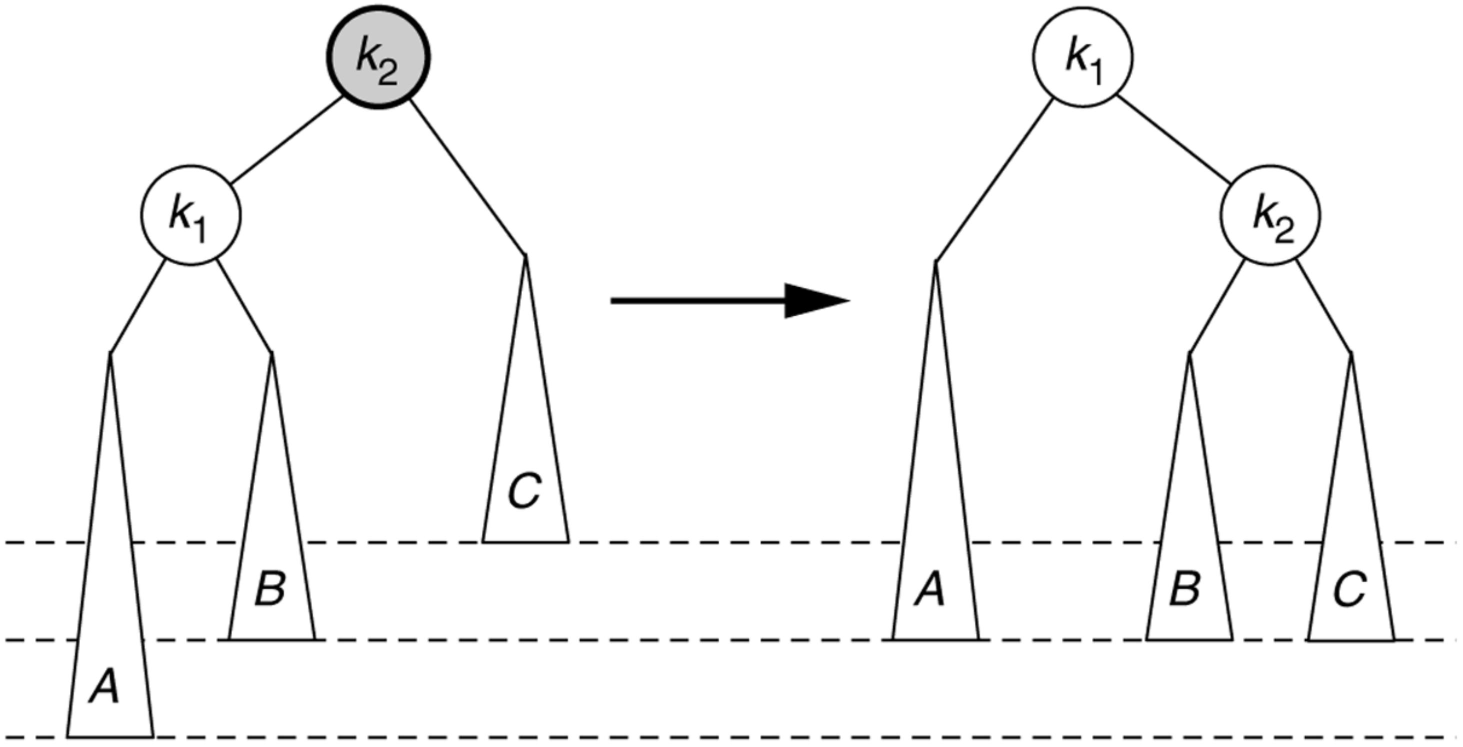
Maintaining Balance

- Maintain balance using *rotations*
  - The idea: reorganize the nodes of an unbalanced subtree until they are balanced, by "rotating" a trio of parent - leftChild - rightChild

- Maintaining balance will result in searches (`contains`) that take O(log *n*)

Right rotation to fix Case 1 (LL)

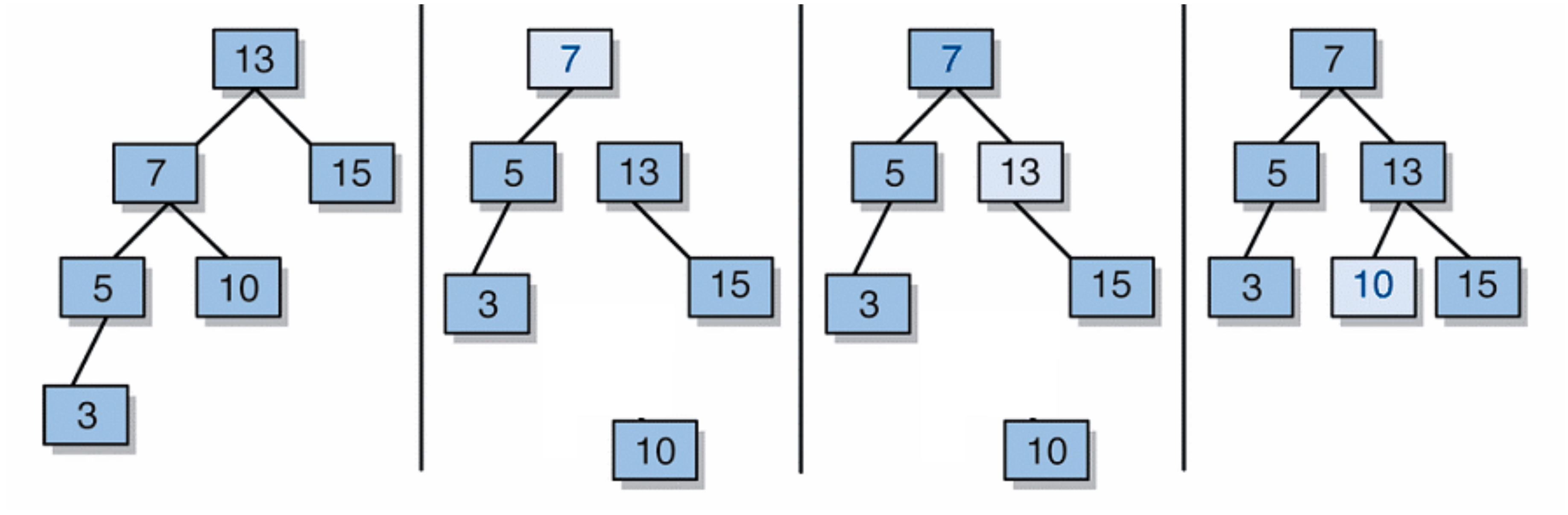
right rotation (clockwise): left child becomes parent; original parent demoted to right



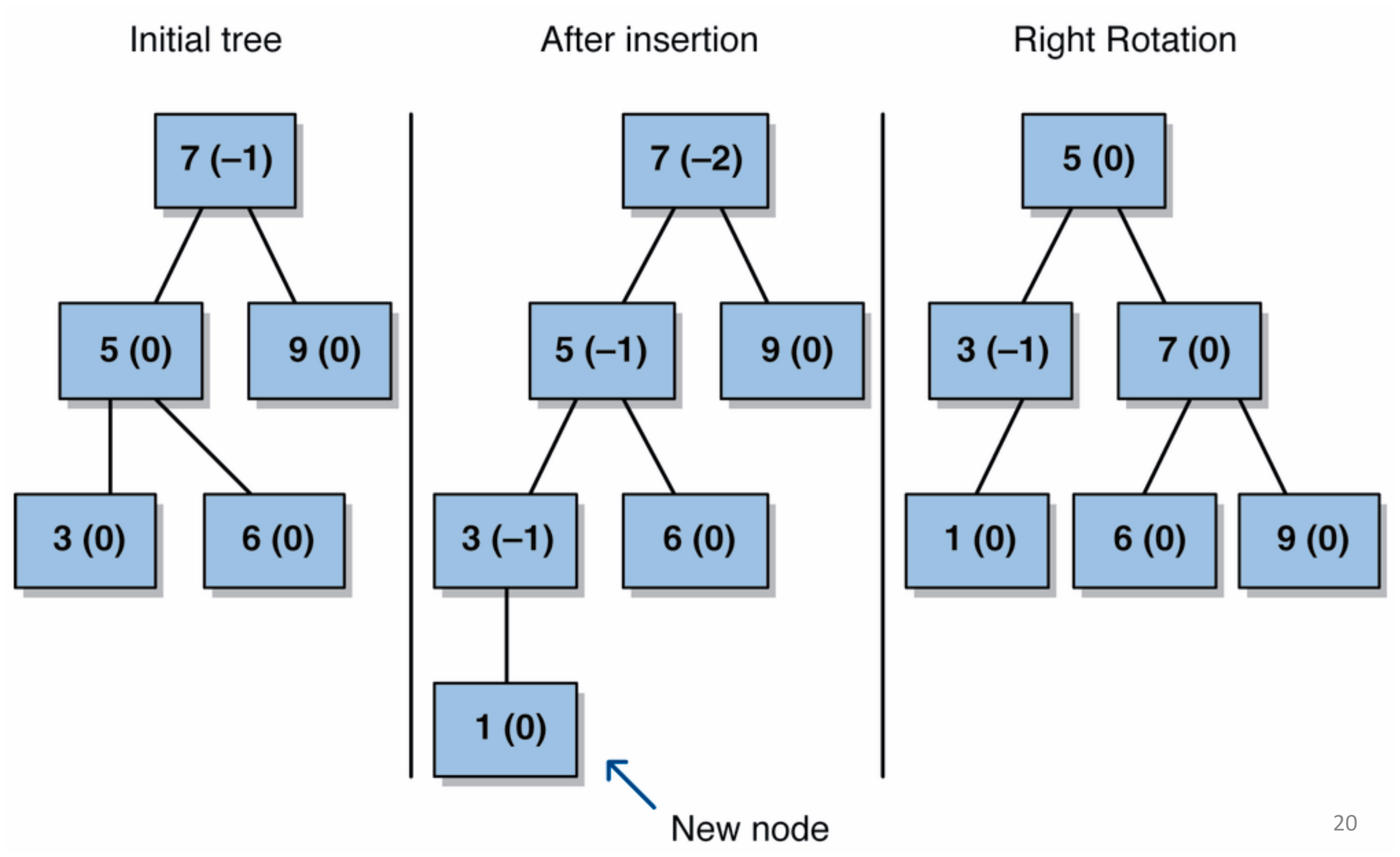
(a) Before rotation (b) After rotation

Right rotation, steps

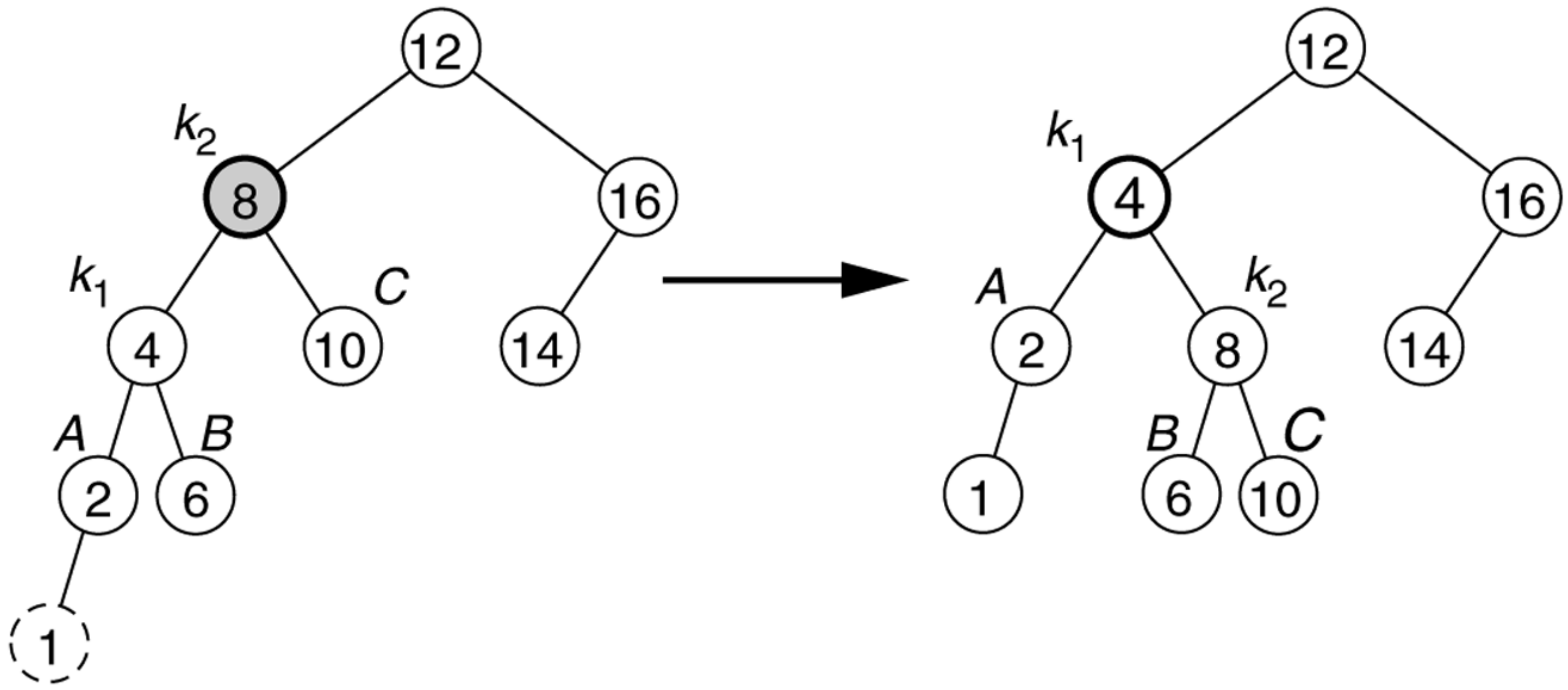
1. detach left child (7)'s right subtree (10) (*don't lose it!*)
2. consider left child (7) be the new parent
3. attach old parent (13) onto right of new parent (7)
4. attach old left child (7)'s old right subtree (10) as left subtree of new right child (13)



Right rotation example



Right rotation example



(a) Before rotation

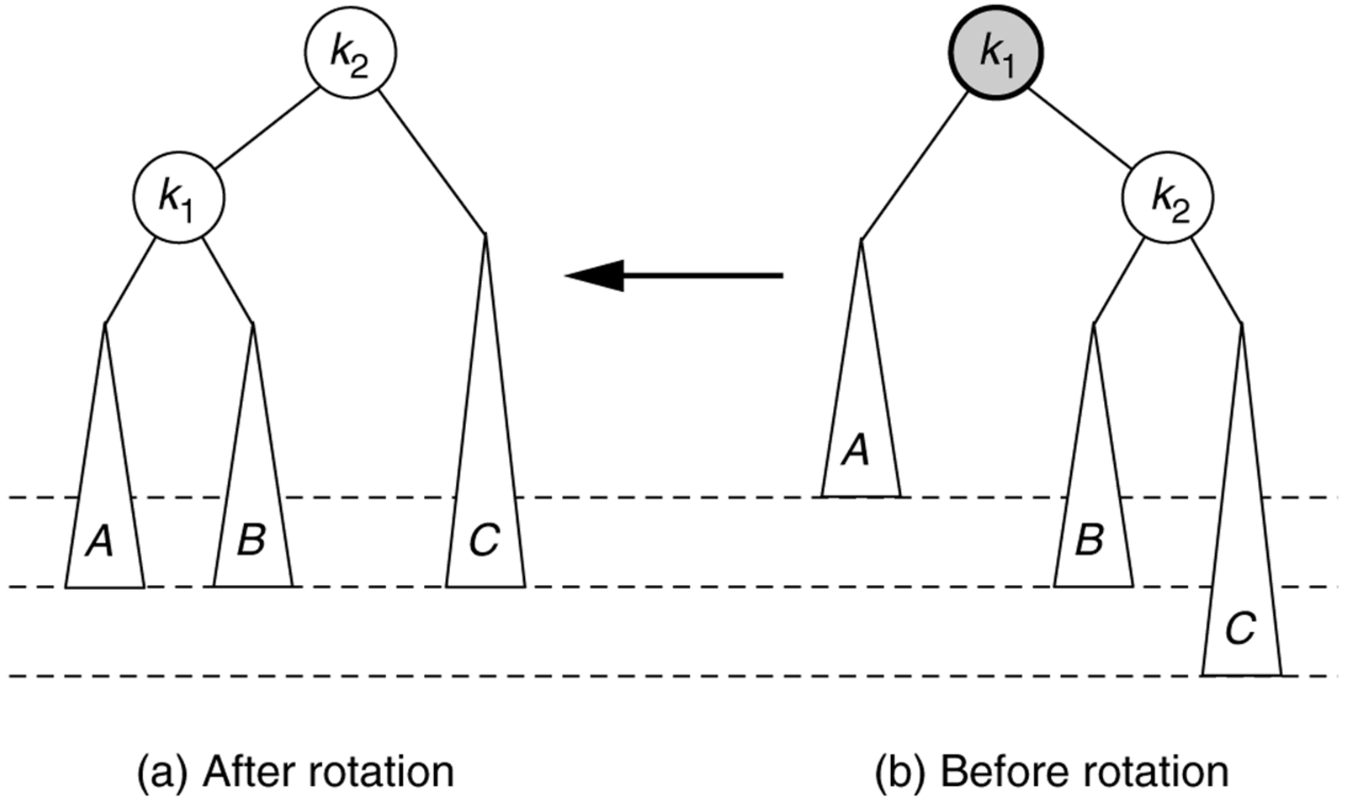
(b) After rotation

Code for right rotation

```
private StringTreeNode rightRotate(StringTreeNode parent) {
    // 1. detach left child's right subtree
    StringTreeNode leftright = parent.left.right;

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

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

    // 4. attach old left child's old right subtree as
    //    left subtree of new right child
    newParent.right.left = leftright;

    parent.height = computeHeight(parent);
    newParent.height = computeHeight(newParent);

    return newParent;
}
```

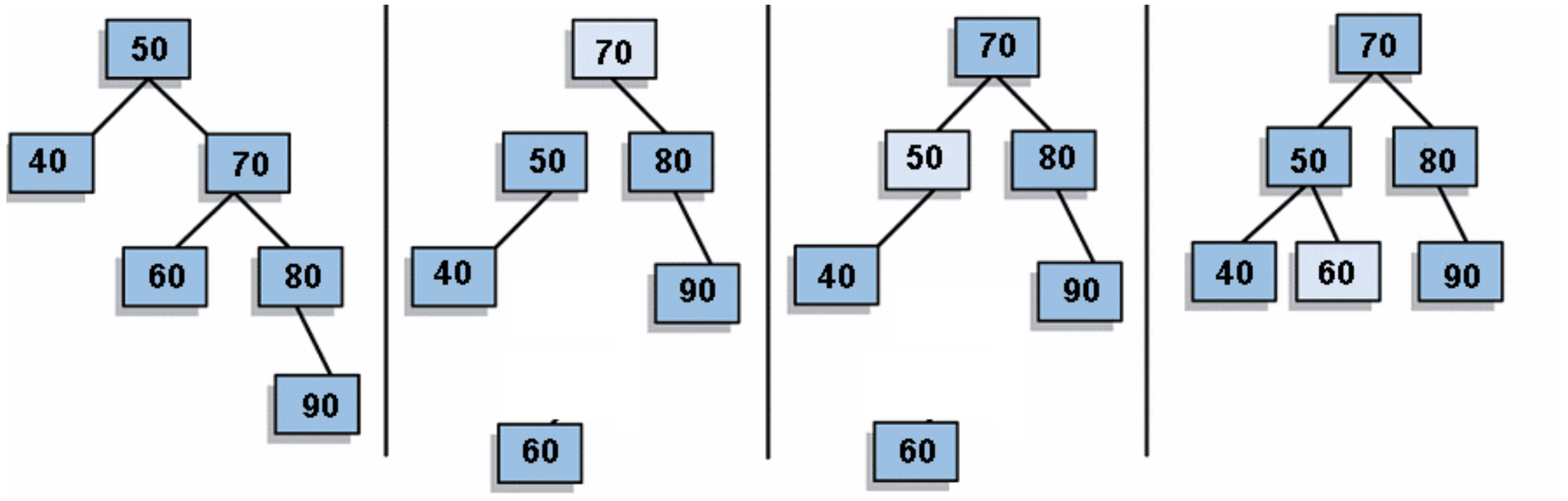
Left rotation to fix Case 4 (RR)

left rotation (counter-clockwise): right child becomes parent; original parent demoted to left



Left rotation, steps

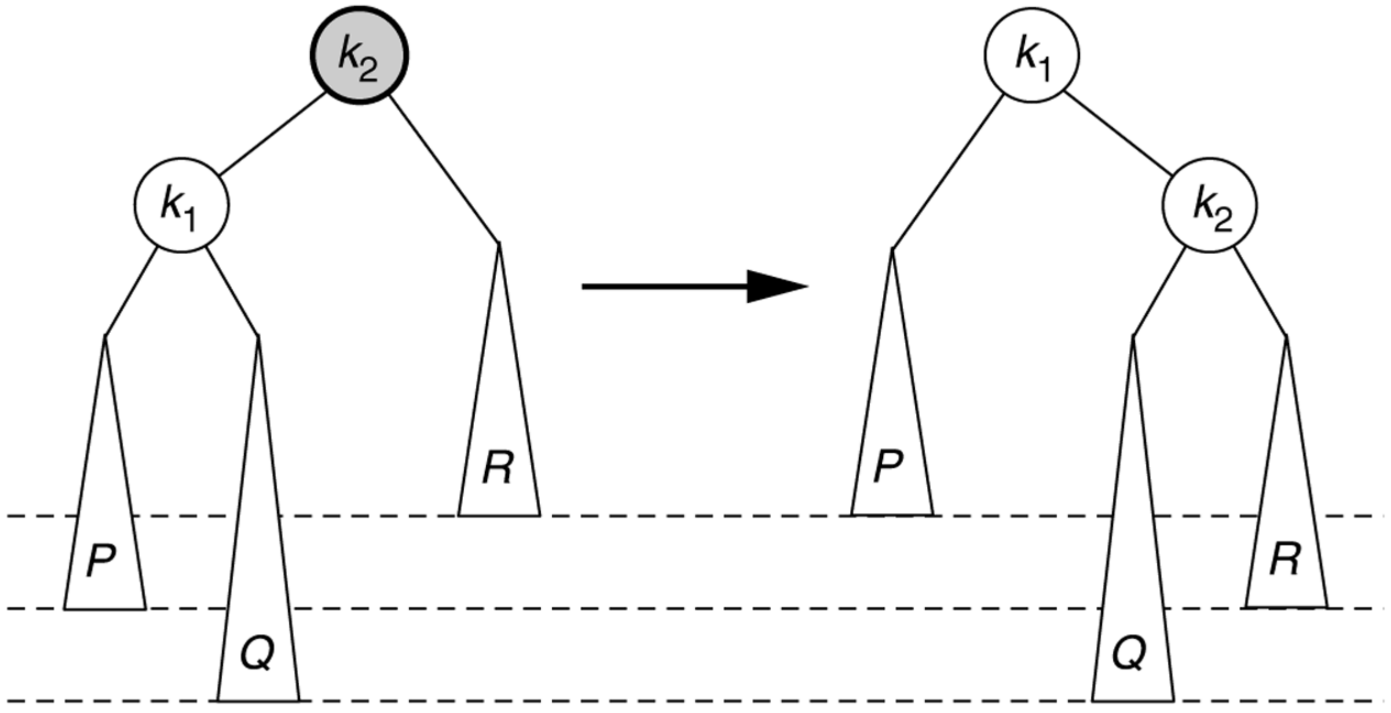
1. detach right child (70)'s left subtree (60) (*don't lose it!*)
2. consider right child (70) be the new parent
3. attach old parent (50) onto left of new parent (70)
4. attach old right child (70)'s old left subtree (60) as right subtree of new left child (50)



Problem: Cases 2, 3

a single right rotation does not fix Case 2!

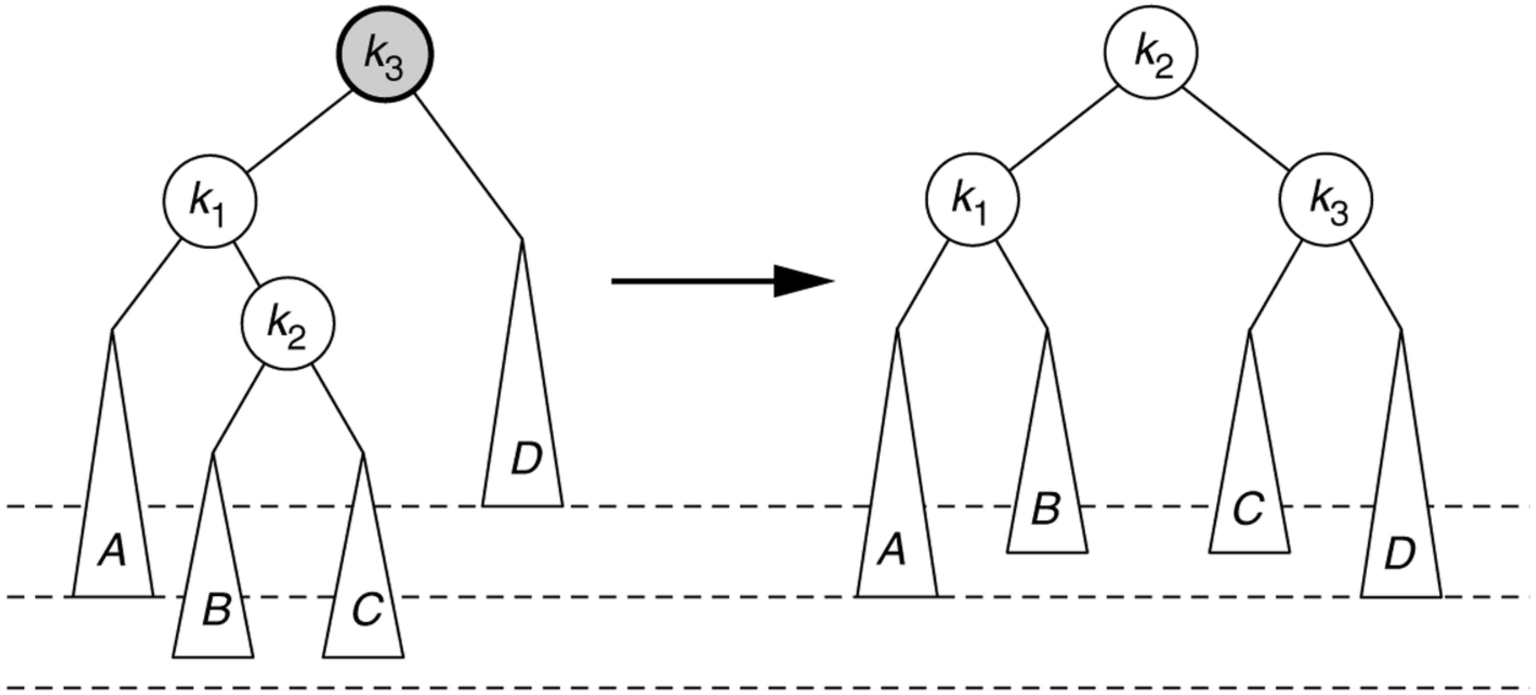
a single left rotation also does not fix Case 3



(a) Before rotation (b) After rotation

Left-right rotation for Case 2

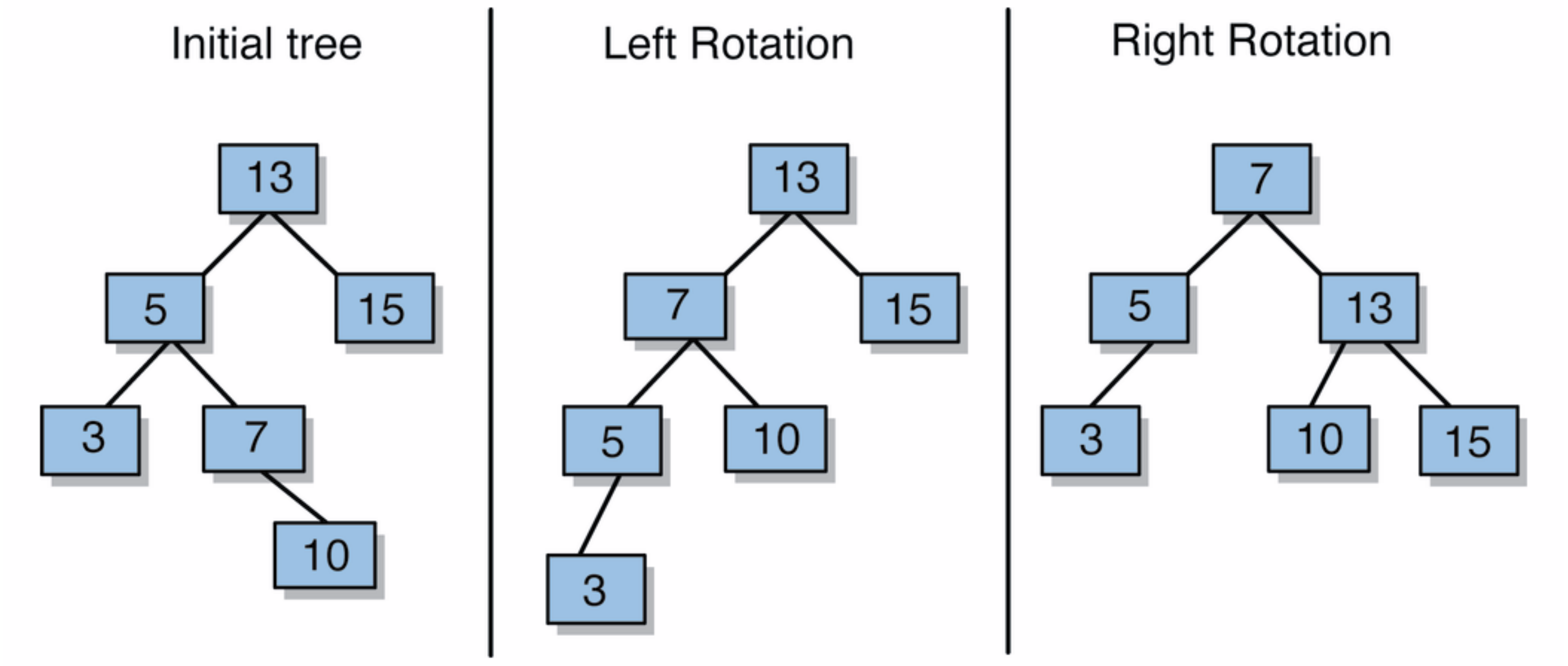
left-right double rotation: a left rotation of the left child, followed by a right rotation at the parent



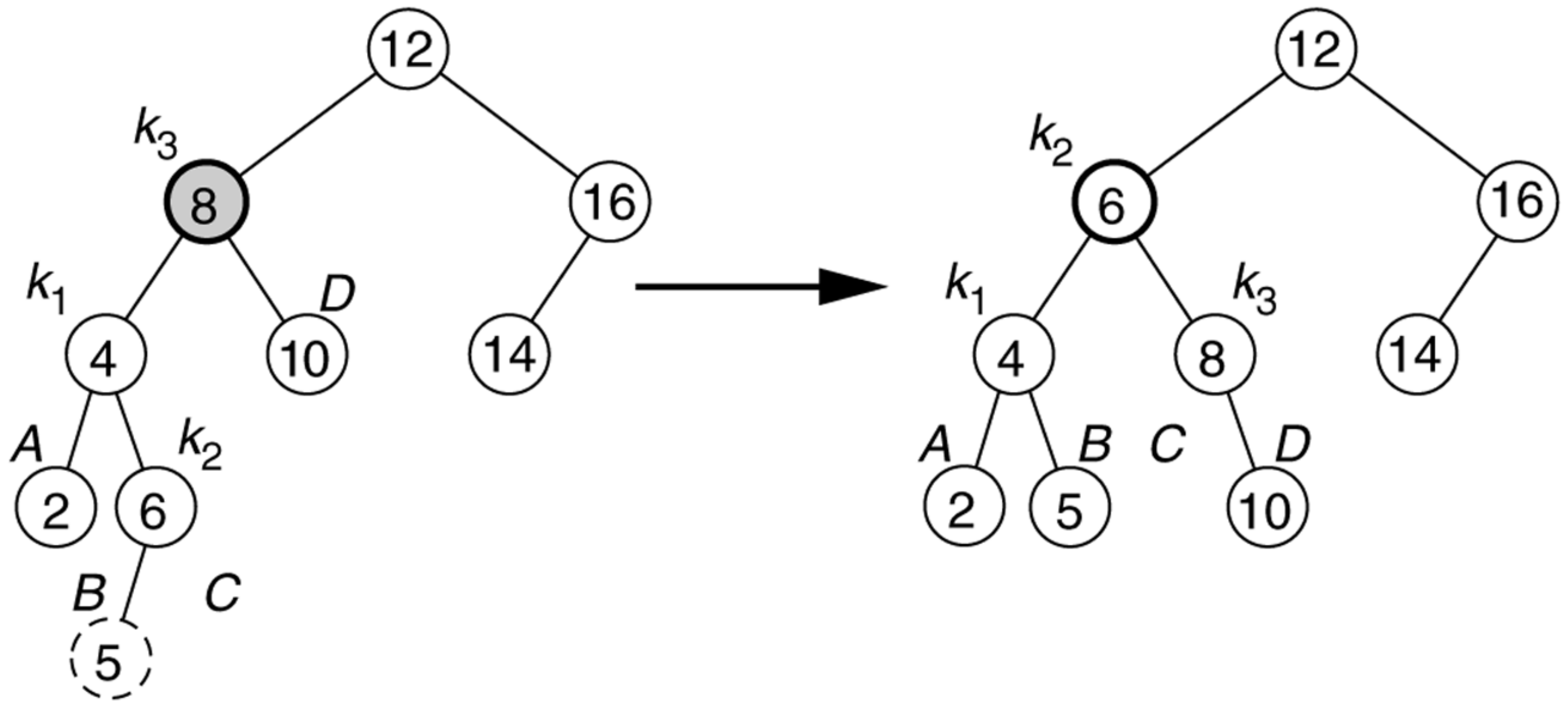
(a) Before rotation (b) After rotation

Left-right rotation, steps

1. perform left-rotate on left child
2. perform right-rotate on parent (current node)



Left-right rotation example

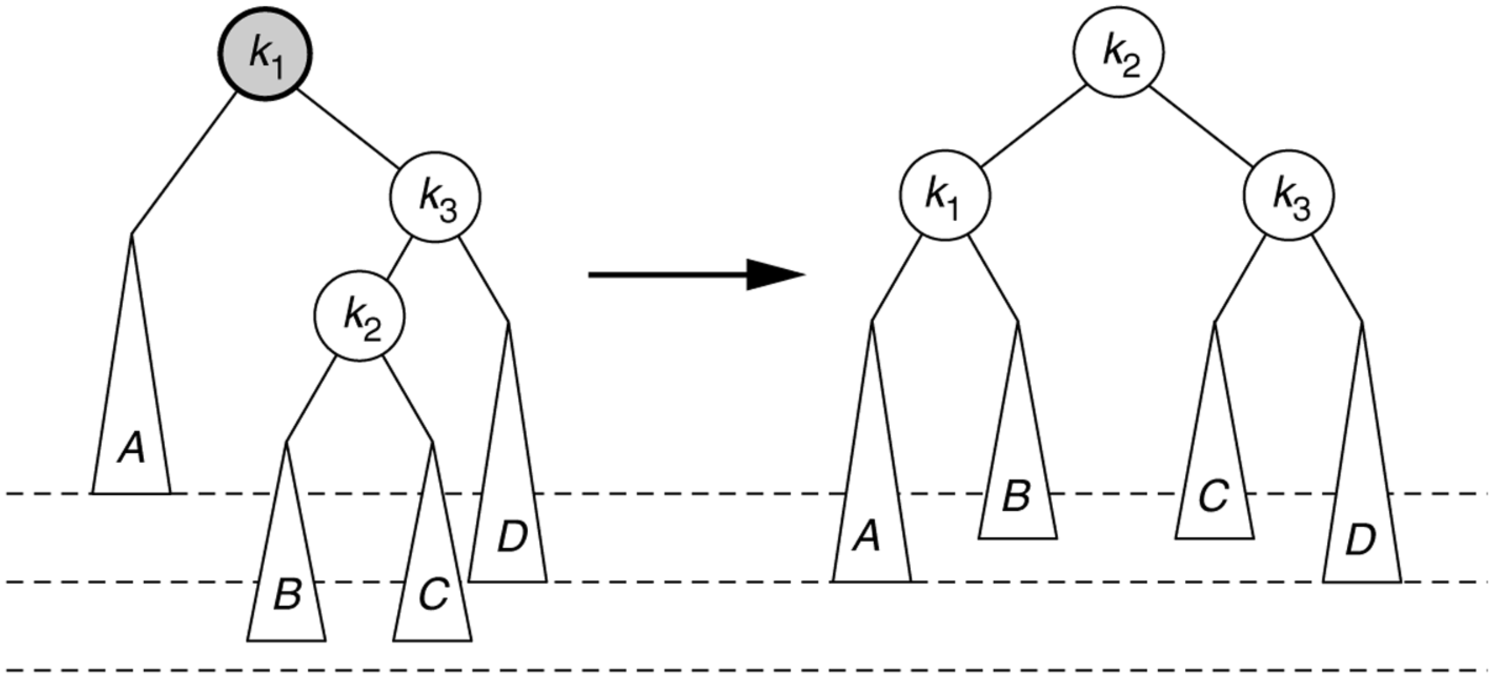


(a) Before rotation

(b) After rotation

Right-left rotation for Case 3

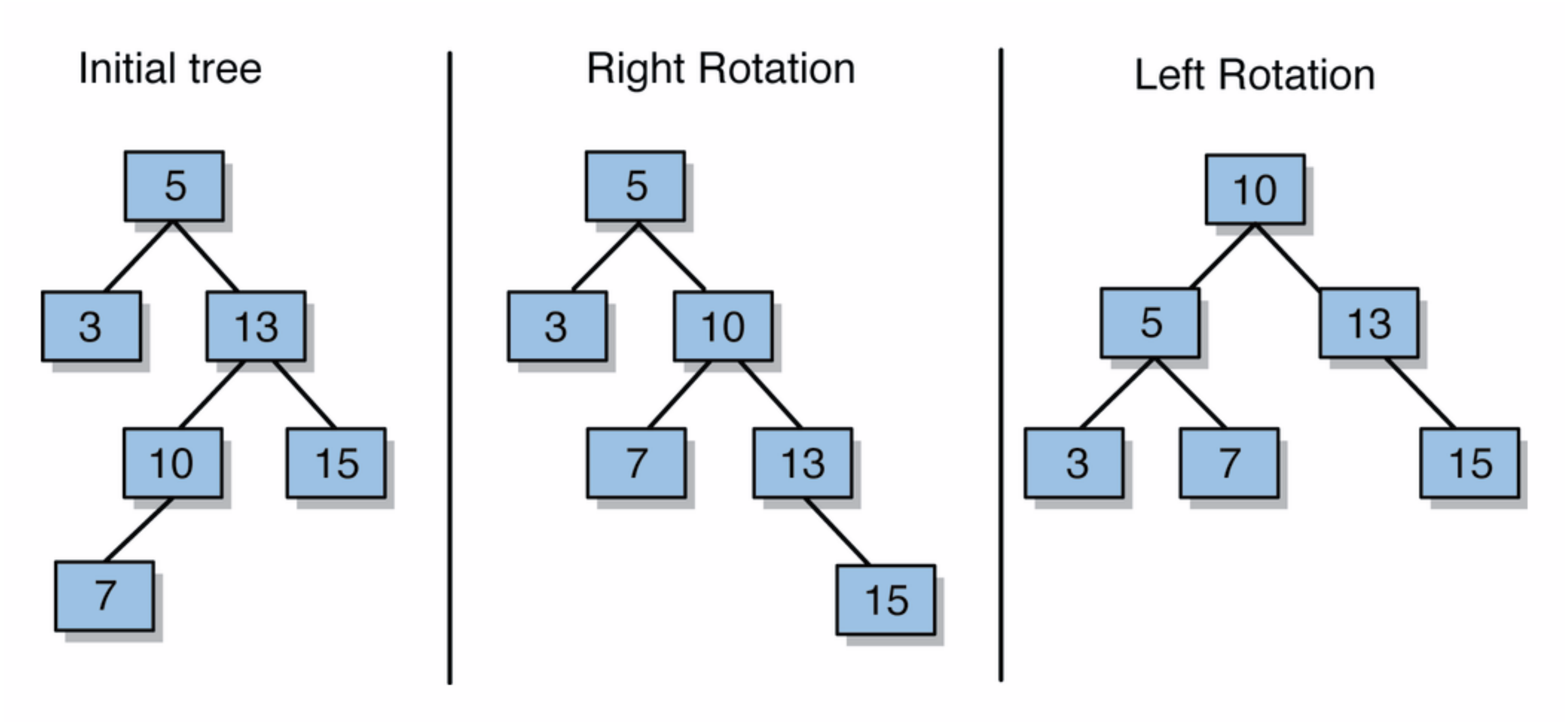
right-left double rotation: a right rotation of the right child, followed by a left rotation at the parent



(a) Before rotation (b) After rotation

Right-left rotation, steps

1. perform right-rotate on right child
2. perform left-rotate on parent (current node)



AVL tree practice problem

- Draw the AVL tree that would result if the following numbers were added in this order to an initially empty tree:
  - 40, 70, 90, 80, 30, -50, 10, 60, 40, -70, 20, 35, 37, 32, 38, 39

- Then give the following information about the tree:
  - size
  - height
  - balance factor at each node

Implementing AVL add

- After normal BST add, update heights from new leaf up towards root
  - If balance factor changes to > +1 or < -1, then use rotation(s) to rebalance
- Let *n* be the first unbalanced node found
  - Case 1: *n* has balance factor -2 and *n*'s left child has balance factor of –1
    - fixed by performing **right-rotation** on *n*
  - Case 2: *n* has balance factor -2 and *n*'s left child has balance factor of 1
    - fixed by perform **left-rotation** on *n*'s left child, then **right-rotation** on *n* (left-right double rotation)

AVL add, cont'd

- Case 3: *n* has balance factor 2 and *n*'s right child has balance factor of –1
  - fixed by perform **right-rotation** on *n*'s right child, then **left-rotation** on *n* (right-left double rotation)
- Case 4: *n* has balance factor 2 and *n*'s right child has balance factor of 1
  - fixed by performing **left-rotation** on *n*

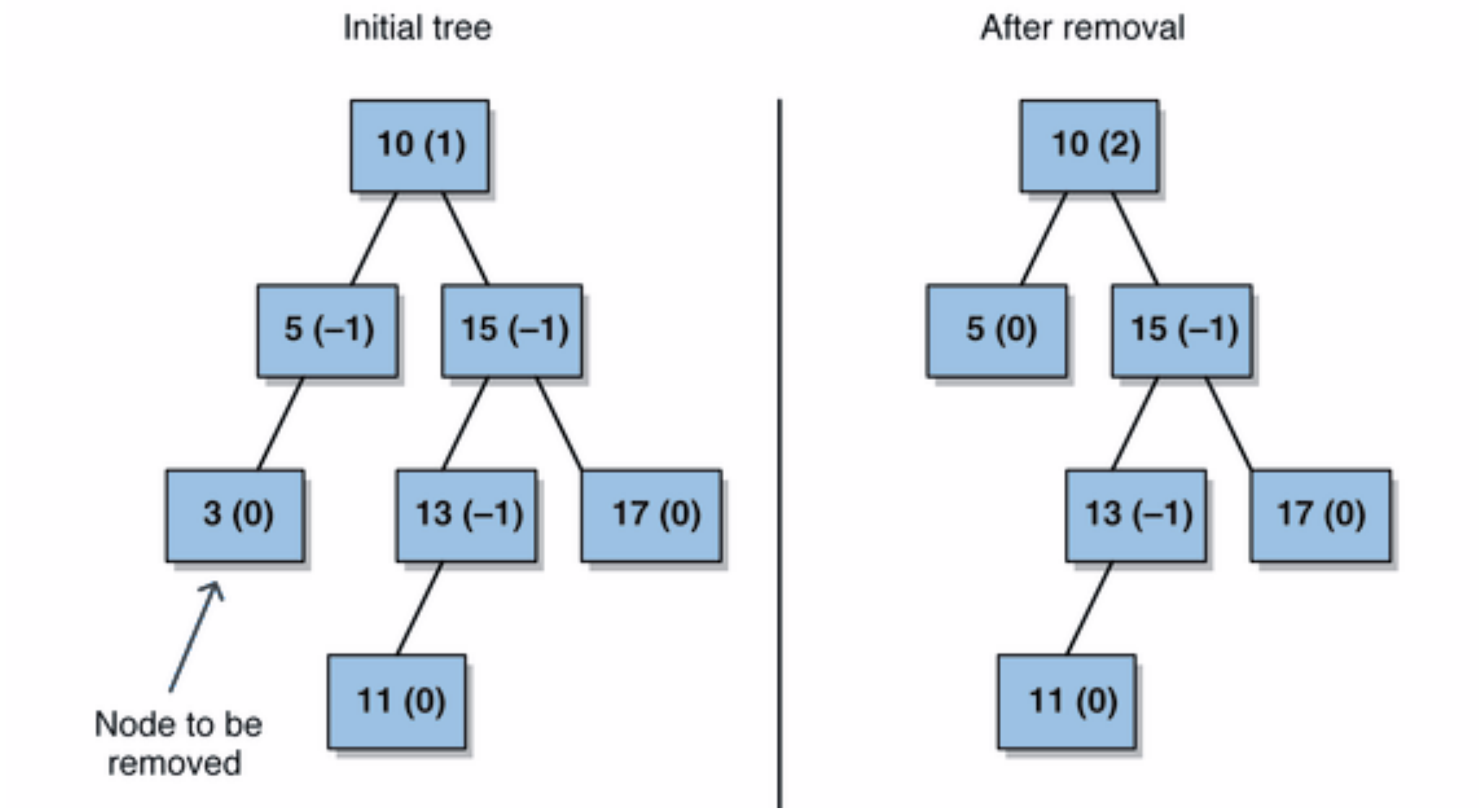
- After rebalancing, continue up the tree updating heights
  - What if *n*'s child has balance factor 0?
  - What if another imbalance occurs higher up?

Code for AVL add

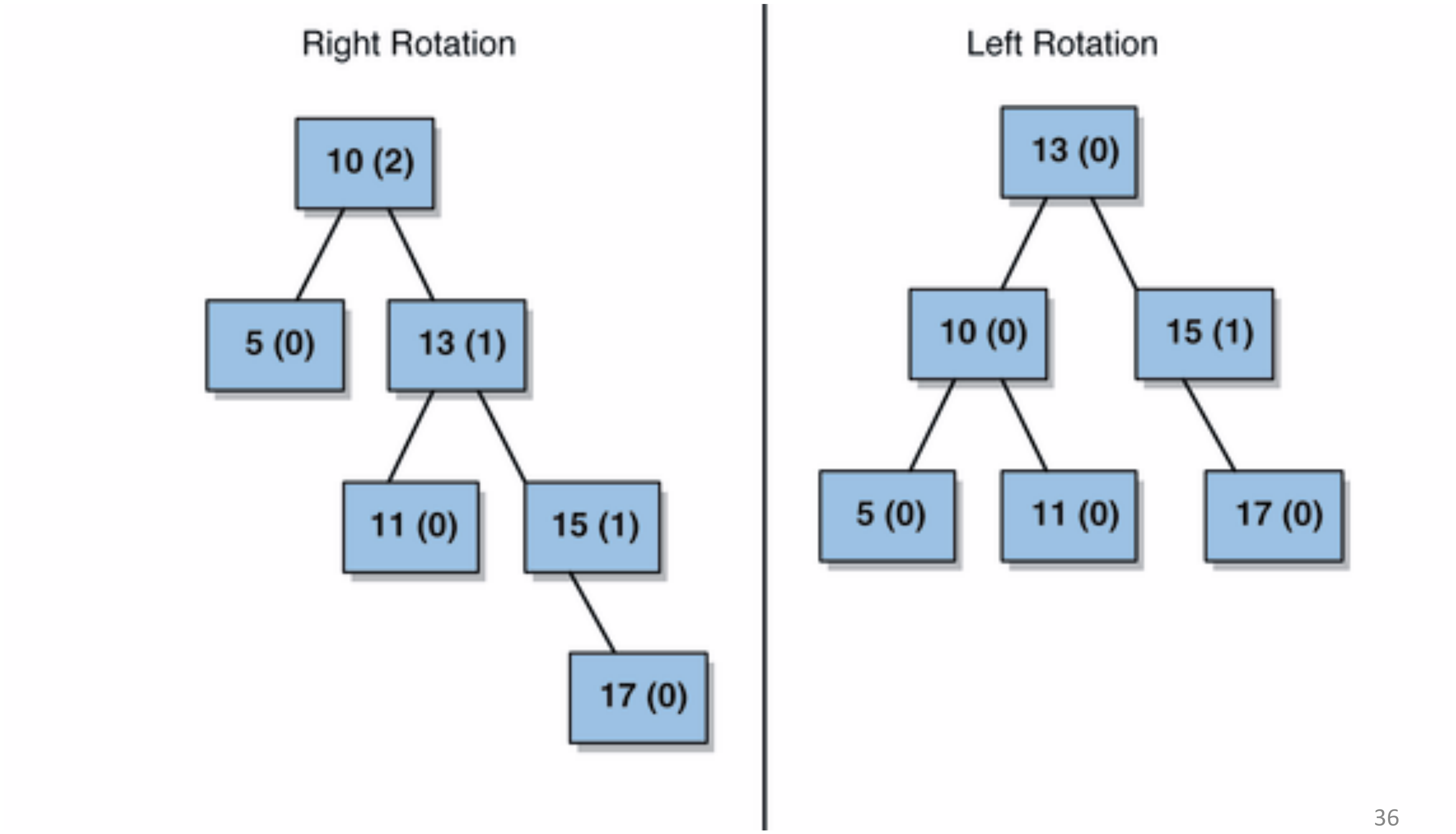
```
protected StringTreeNode add(StringTreeNode node, String value) {
    node = super.add(node, value);
    node.height = computeHeight(node);
    node = rebalance(node);

    return node;
}

protected StringTreeNode rebalance(StringTreeNode node) {
    int bf = balanceFactor(node);
    if (bf < -1) {
        if (balanceFactor(node.left) < 0) {   // case 1 (LL)
            node = rightRotate(node);
        } else {                              // case 2 (LR)
            node.left = leftRotate(node.left);
            node = rightRotate(node);
        }
    } else if (bf > 1) {
       // take care of symmetric cases
       //     case 3 (RL)
       //     case 4 (RR)
    }
}
```

Problems for AVL remove

removal from AVL tree can also unbalance the tree



Right-left rotation on remove



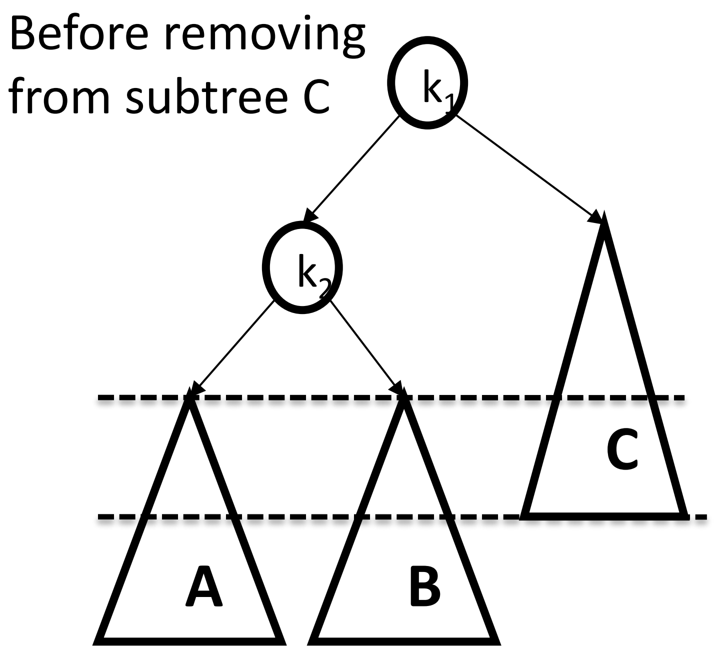
AVL remove, cont'd

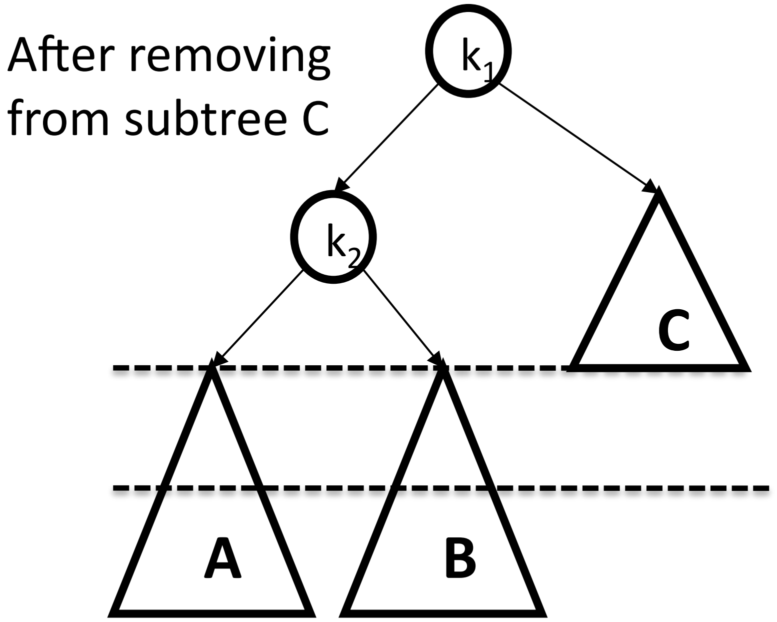
1. perform normal BST remove (with replacement of node to be removed with its successor)
2. update heights from successor node location upwards towards root
   - if balance factor changes to +2 or -2, then use rotation(s) to rebalance

- remove has the same 4 cases (and fixes) as insert
  - are there any additional cases?
- After rebalancing, continue up the tree updating heights; must continue checking for imbalances in balance factor, and rebalancing if necessary
  - Are all cases handled?

Additional AVL Remove Cases

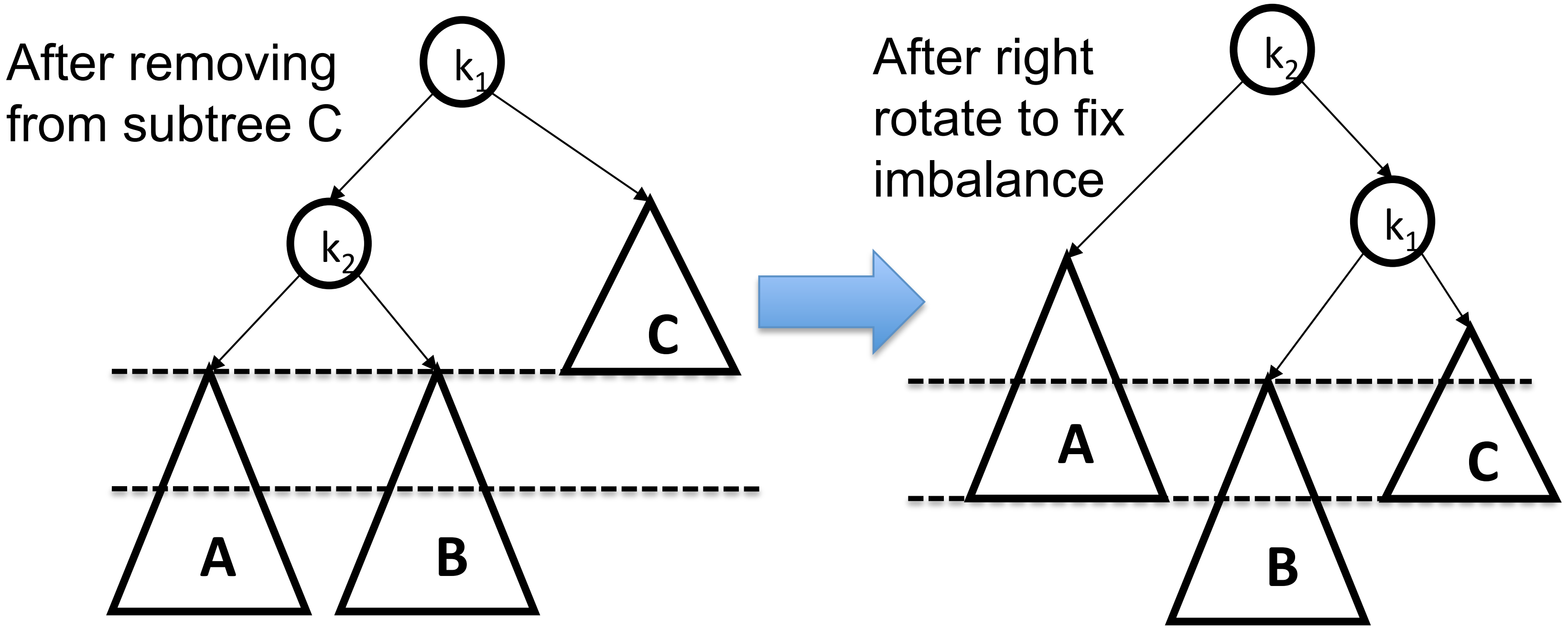
- Two additional cases cause AVL tree to become unbalanced on remove
- In these cases, a node (e.g., k_1 below) violates balance condition after removing from one of its subtrees when its other subtree has a balance factor of 0
  - these cases do not occur for insertion: when insertion causes a tree to have a balance factor of 2 or -2, the child containing the subtree where the insertion occurred either has a balance factor of -1 or 1





Fixing AVL Remove Cases

- Each of these cases can be fixed through a single rotation
  - If remove from right subtree of node creates imbalance and left subtree has balance factor of 0 we right rotate (shown below)
  - If remove from left subtree of node creates imbalance and right subtree has balance factor of 0 we left rotate (symmetric case)


After removing from subtree C
k_1
k_2
A
B
C
After right rotate to fix imbalance
k_2
k_1
A
B
C