CSE 373: Data Structures and Algorithms

Lecture 11: Trees III

AVL Tree Motivation

Observation: the shallower the BST the better

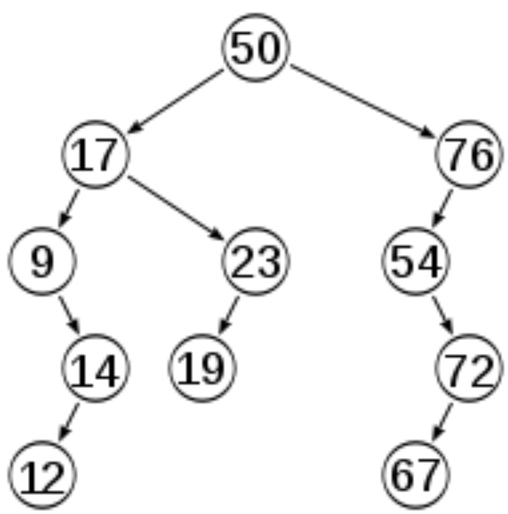
- For a BST with n nodes
  - Average case height is $\Theta(\log n)$
  - Worst case height is $\Theta(n)$
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario: height $\Theta(n)$

Strategy: Don't let the tree get lopsided

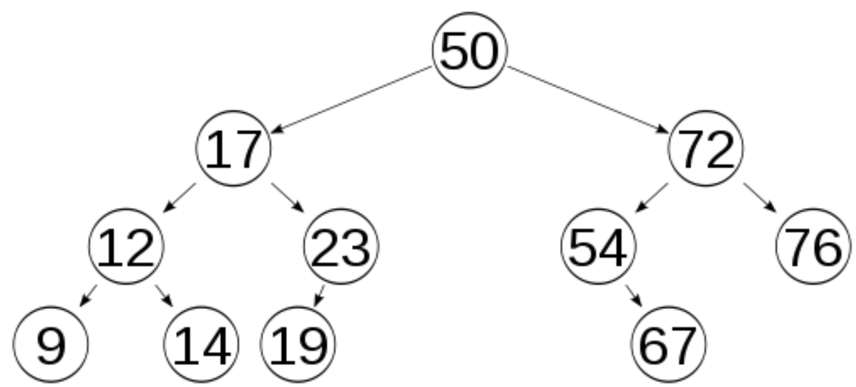
- Constantly monitor balance for each subtree
- Rebalance subtree before going too far astray

Balanced Tree

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



unbalanced tree



balanced tree

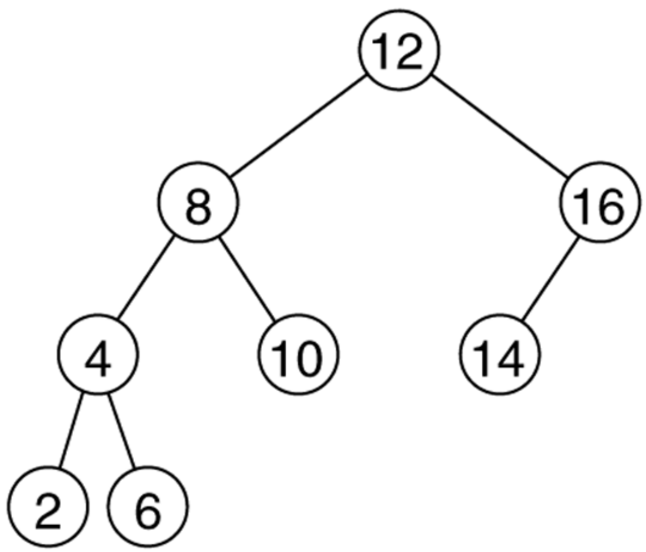
AVL trees

- **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)

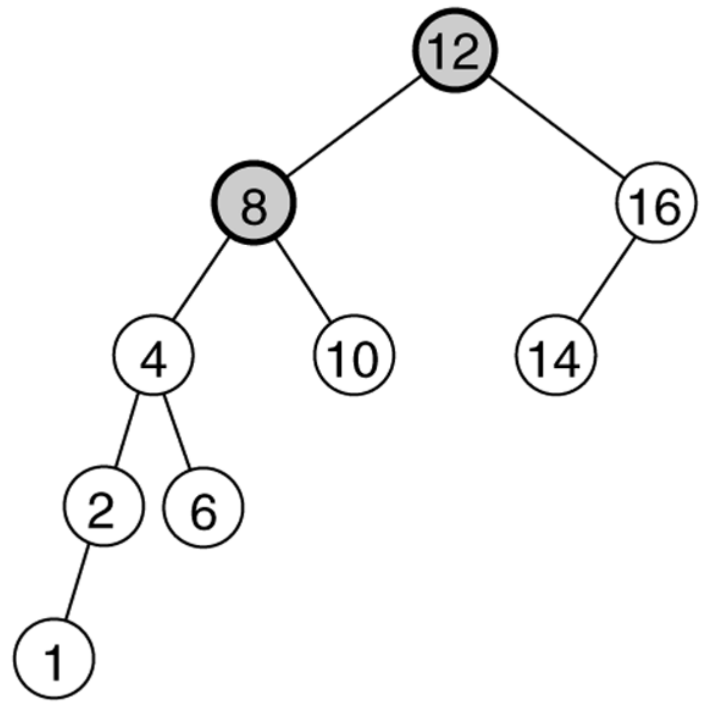
- **balance factor**, for a tree node *n* :
  - height of *n*'s right subtree minus height of *n*'s left subtree
  - $BF_n = Height_{n.right} - Height_{n.left}$
  - start counting heights at n

AVL tree examples

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

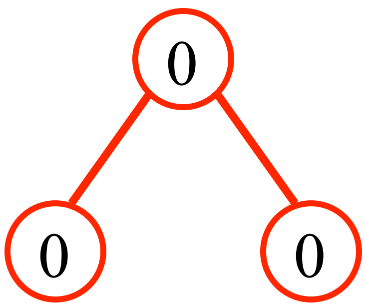


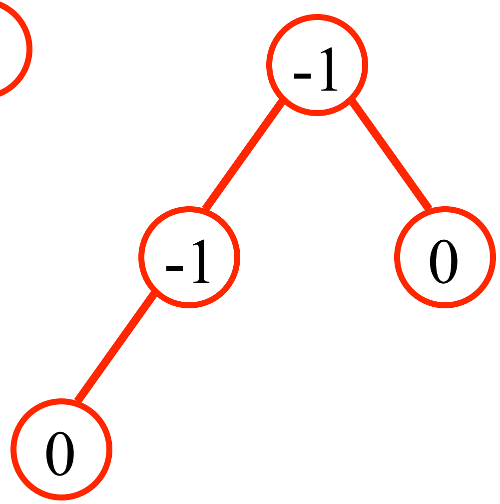
(a)

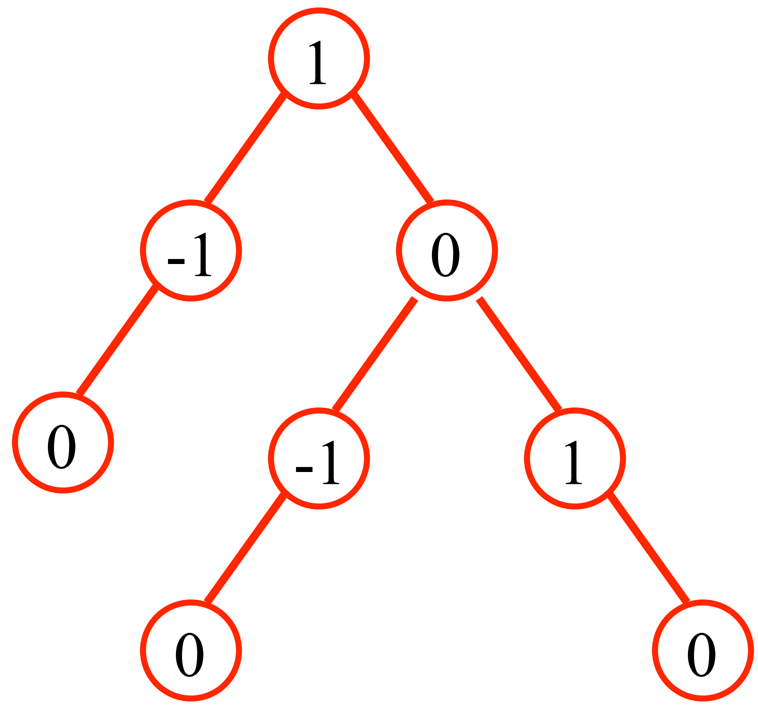


(b)

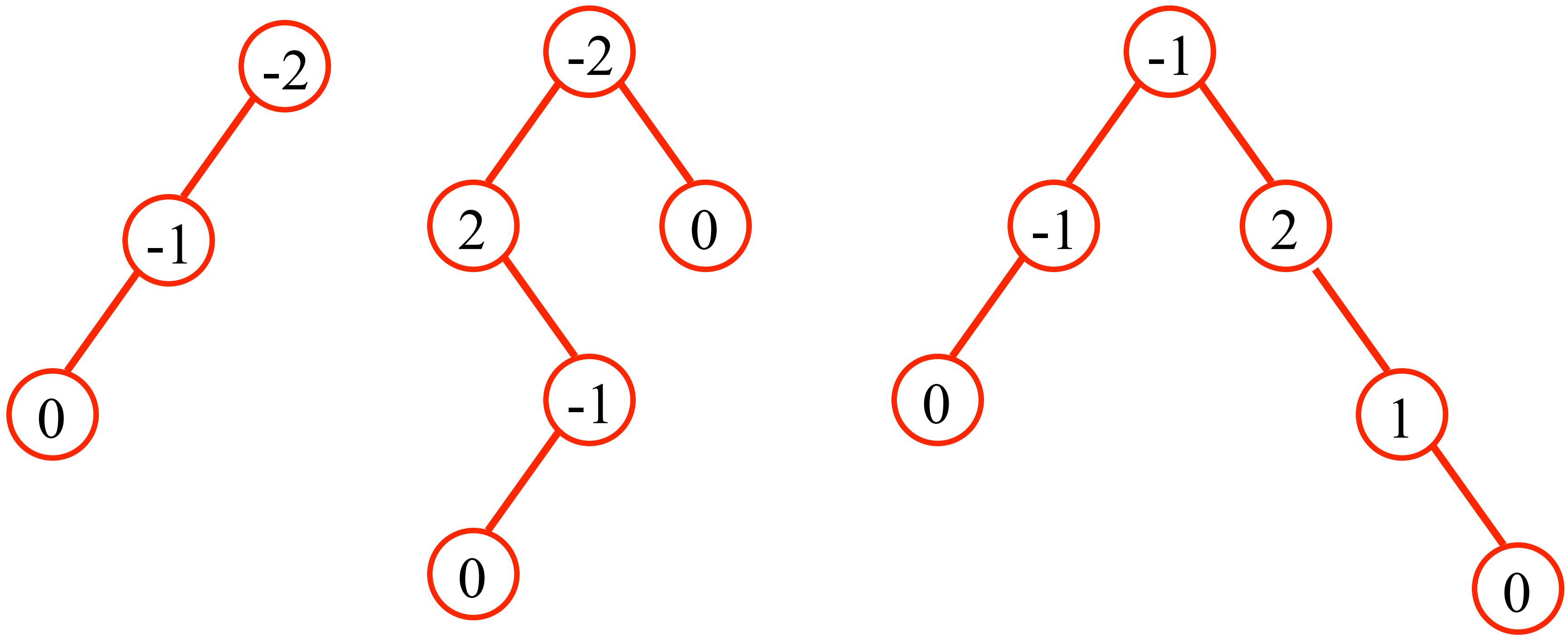
More AVL tree examples



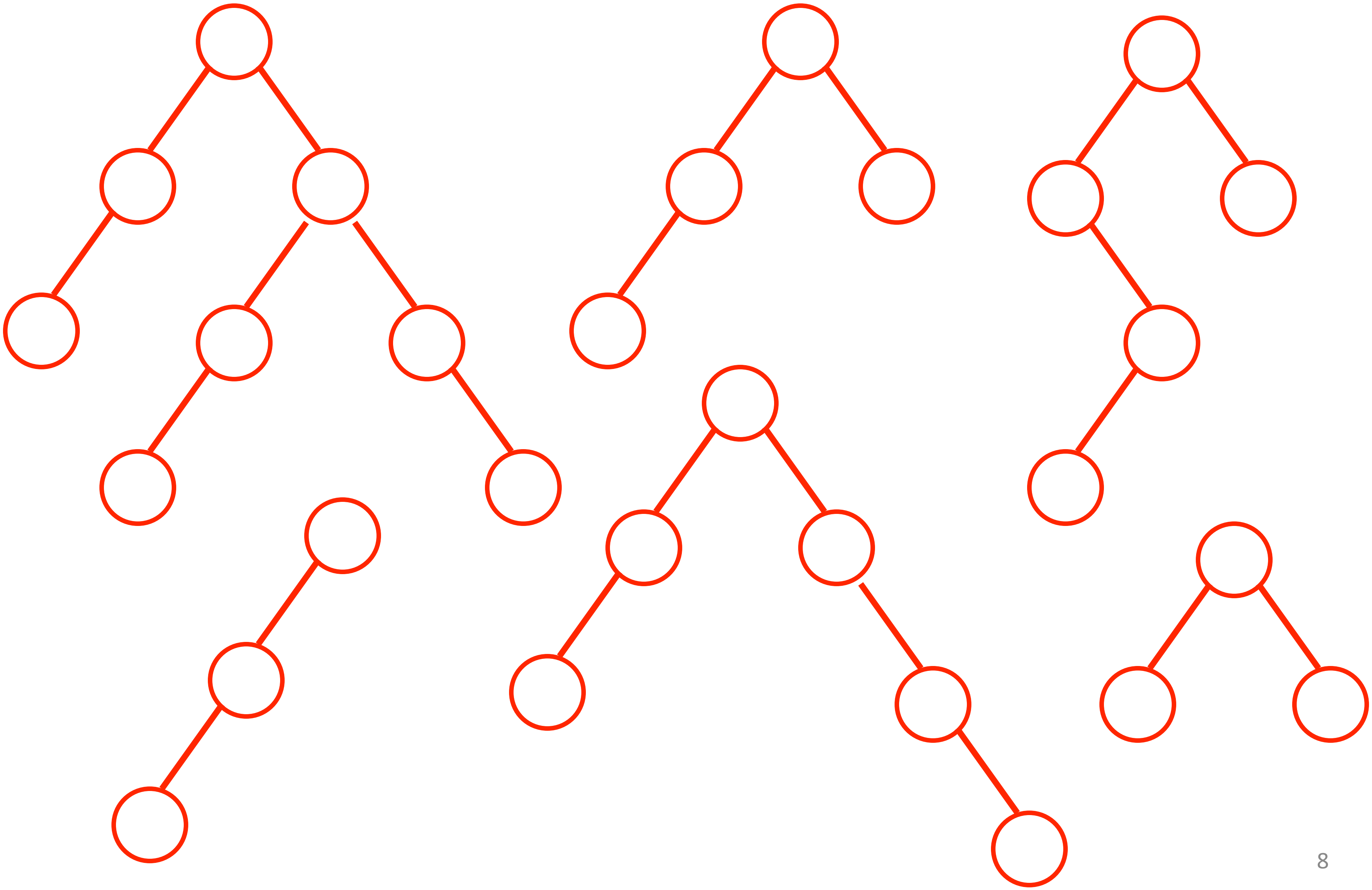




Not AVL tree examples



Which are AVL trees?



AVL Tree Height

- The height of an AVL Tree is $\Theta(\log n)$
- Justification: Find n(h): the minimum number of nodes in an AVL tree of height h
  - n(0) = 1 and n(1) = 2
  - For h >= 2, an AVL tree of height h contains the root node, an AVL subtree of height h - 1 and an AVL subtree of height h - 1 or h - 2
    - $n(h) = 1 + n(h - 1) + n(h - 2)$
      - Solving this recurrence leads to $\Theta(\log n)$

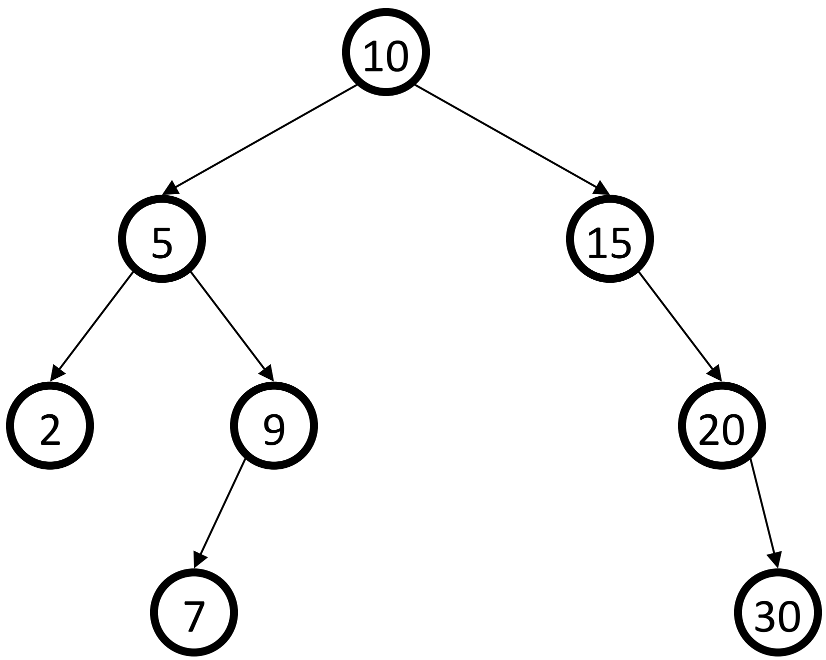
AVL Trees: search, insert, remove

- 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.

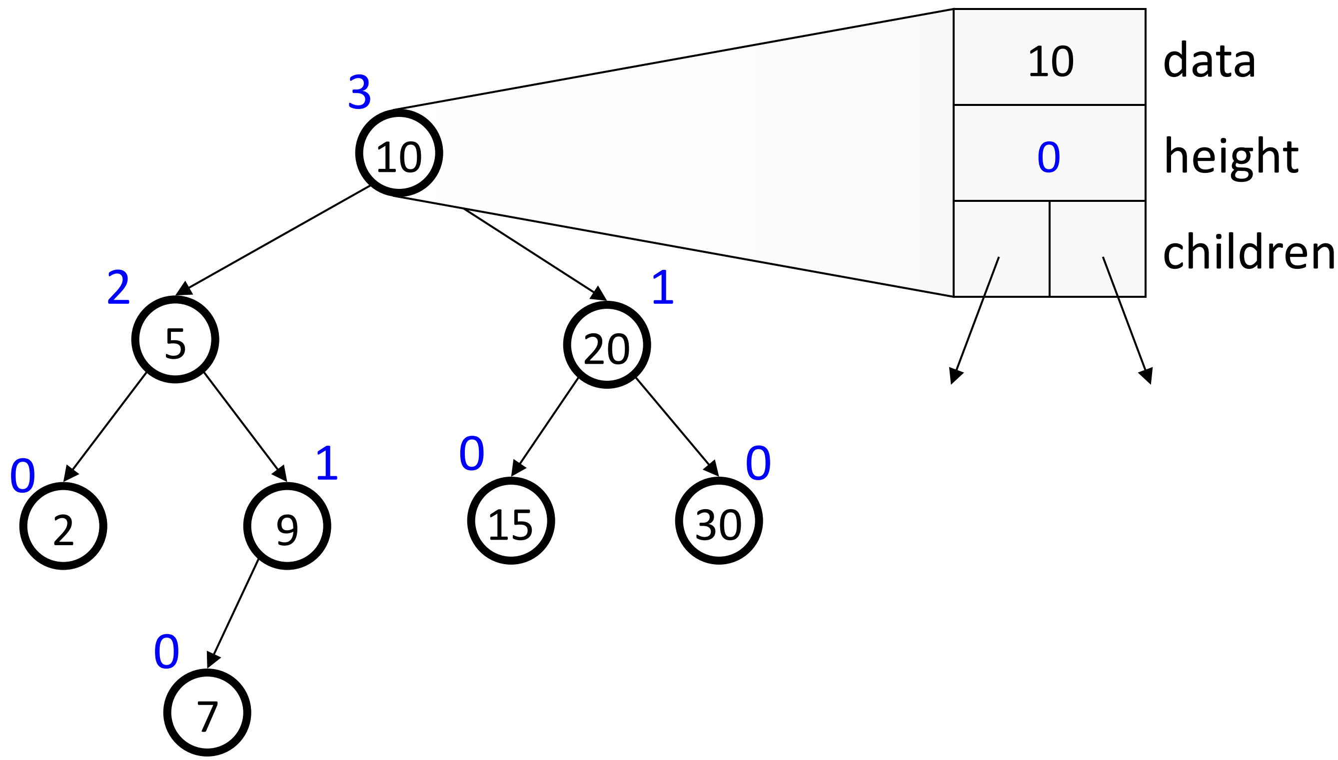
Testing the Balance Property



We need to be able to:

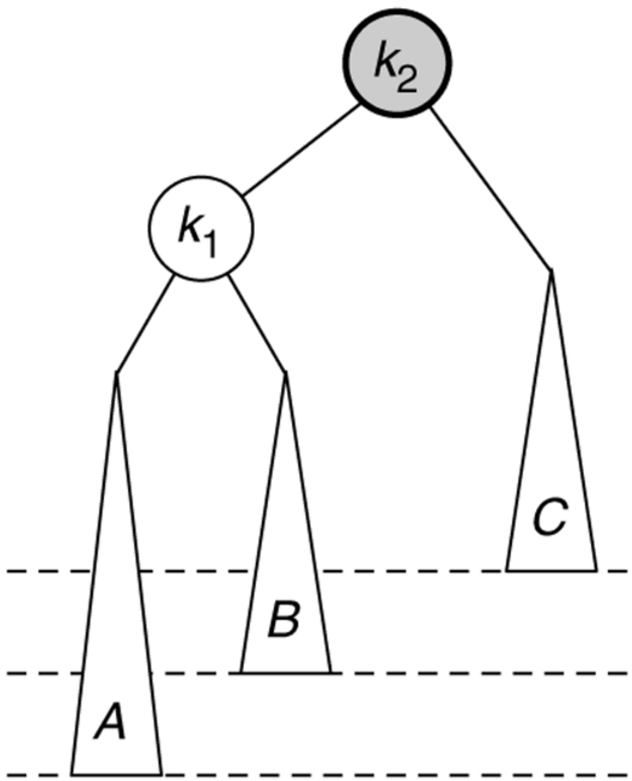
1. Track Balance Factor
2. Detect Imbalance
3. Restore Balance

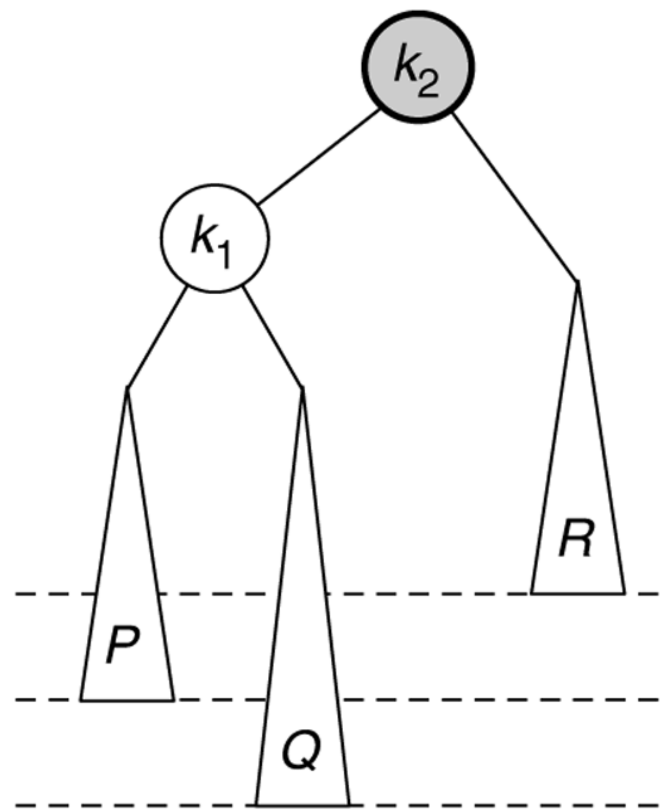
Tracking Balance



Problem cases for AVL insert

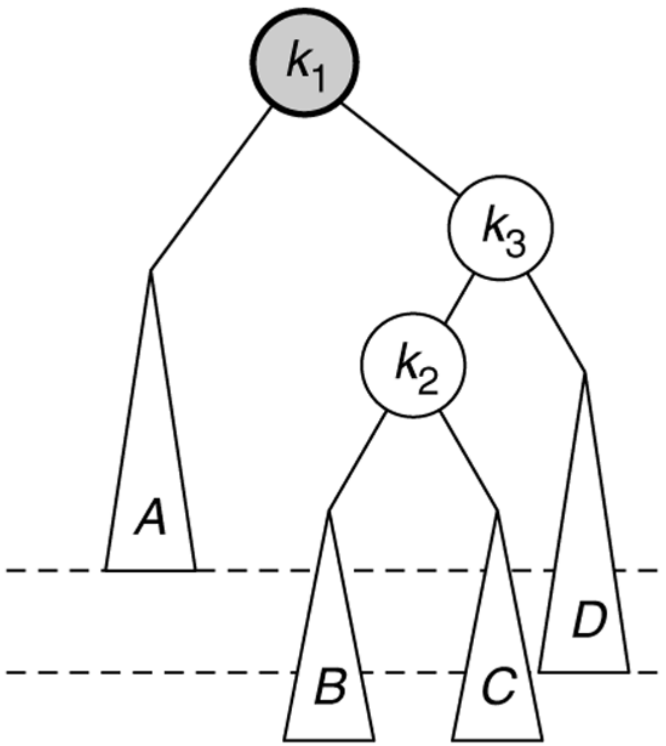
1. LL Case: insertion into left subtree of node's left child
2. LR Case: insertion into right subtree of node's left child

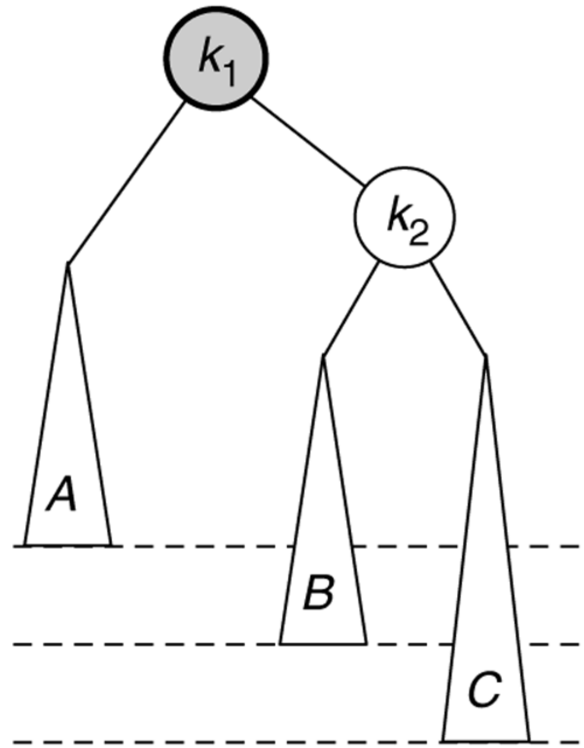




Problem cases for AVL insert, cont.

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





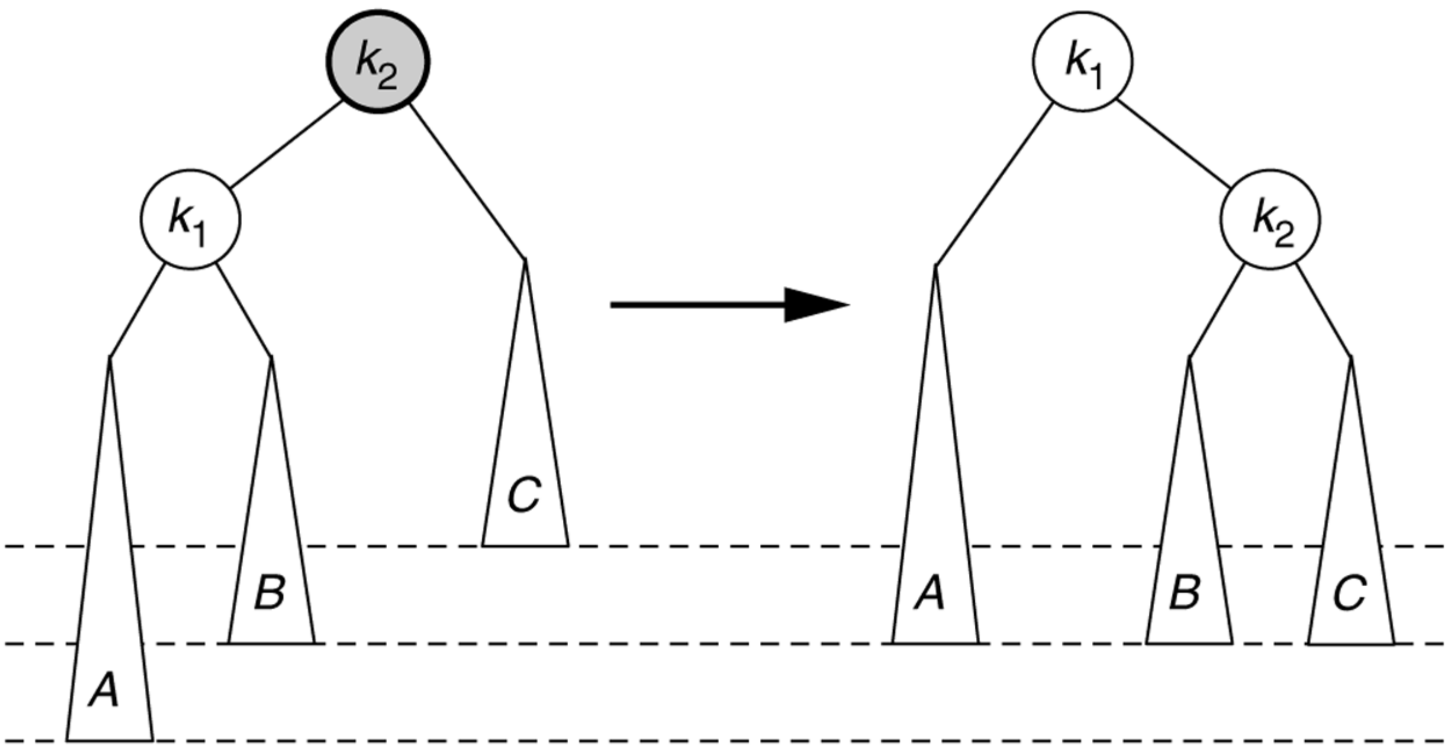
Maintaining Balance

- Maintain balance using *rotations*
  - The idea: locally 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 $\Theta(\log n)$

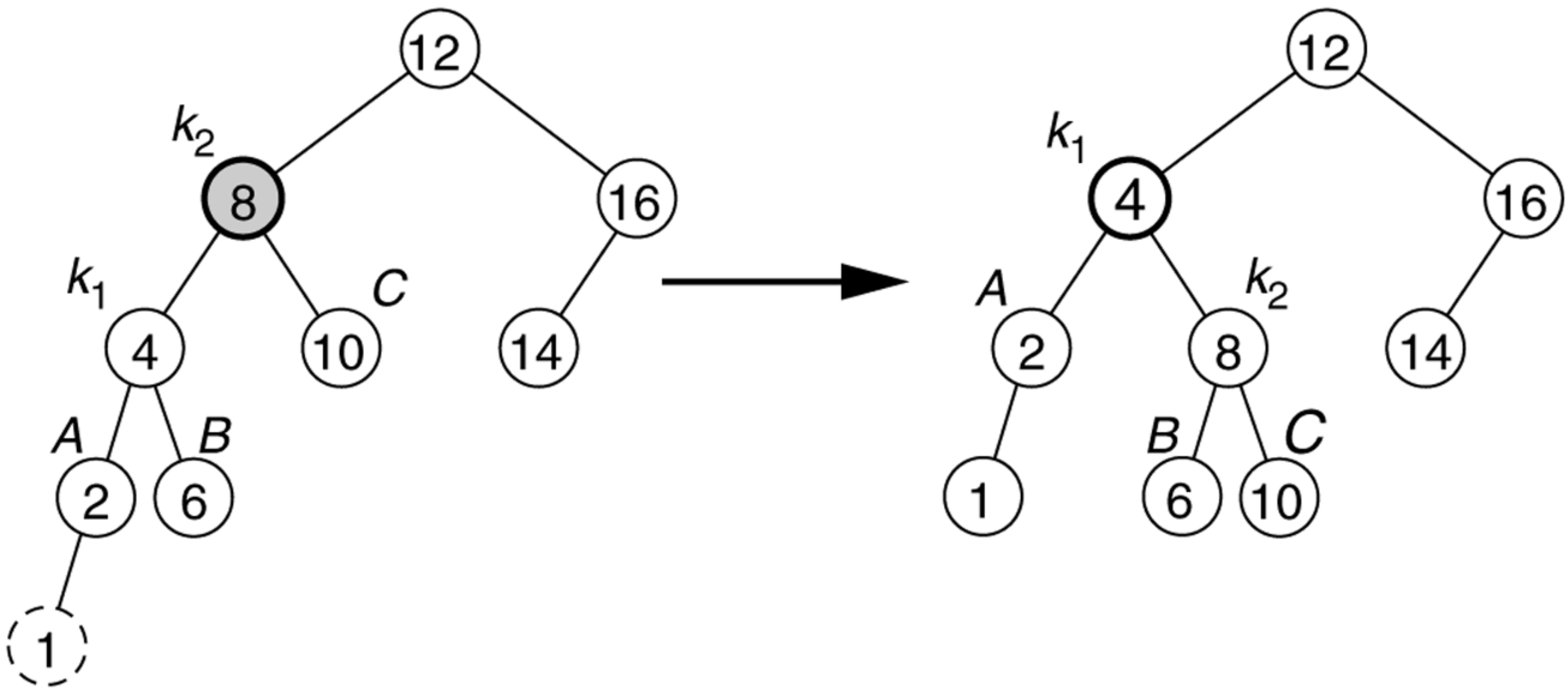
Right rotation to fix Case 1 (LL)

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



(a) Before rotation (b) After rotation

Right rotation example

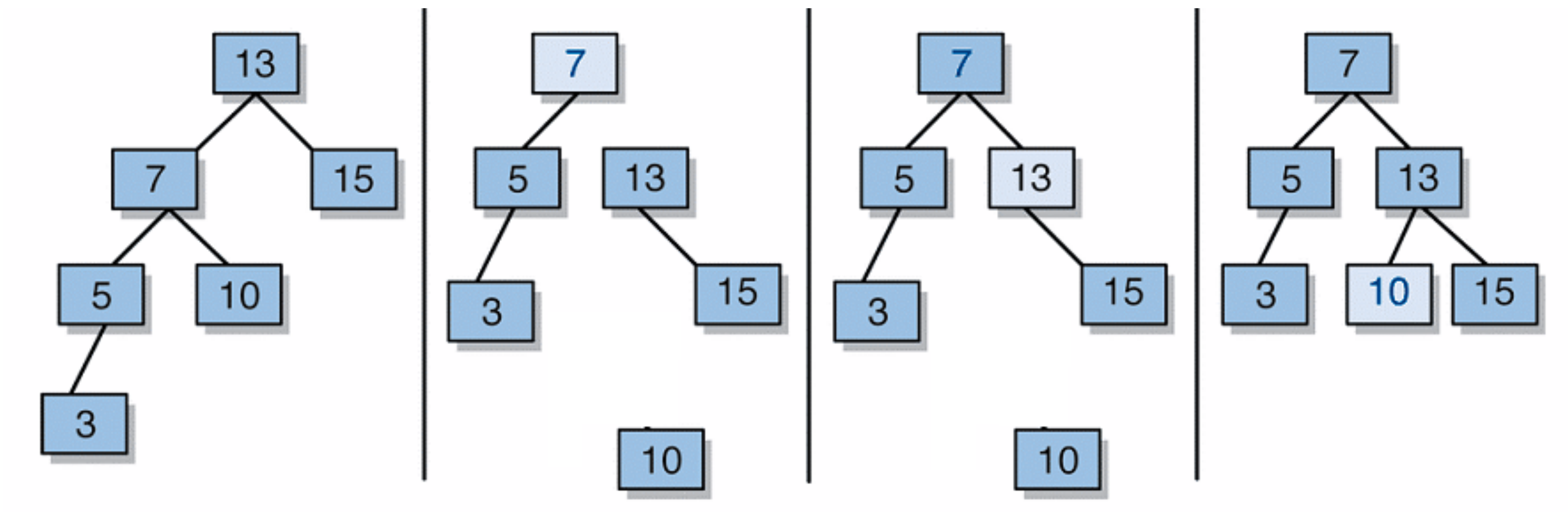


(a) Before rotation

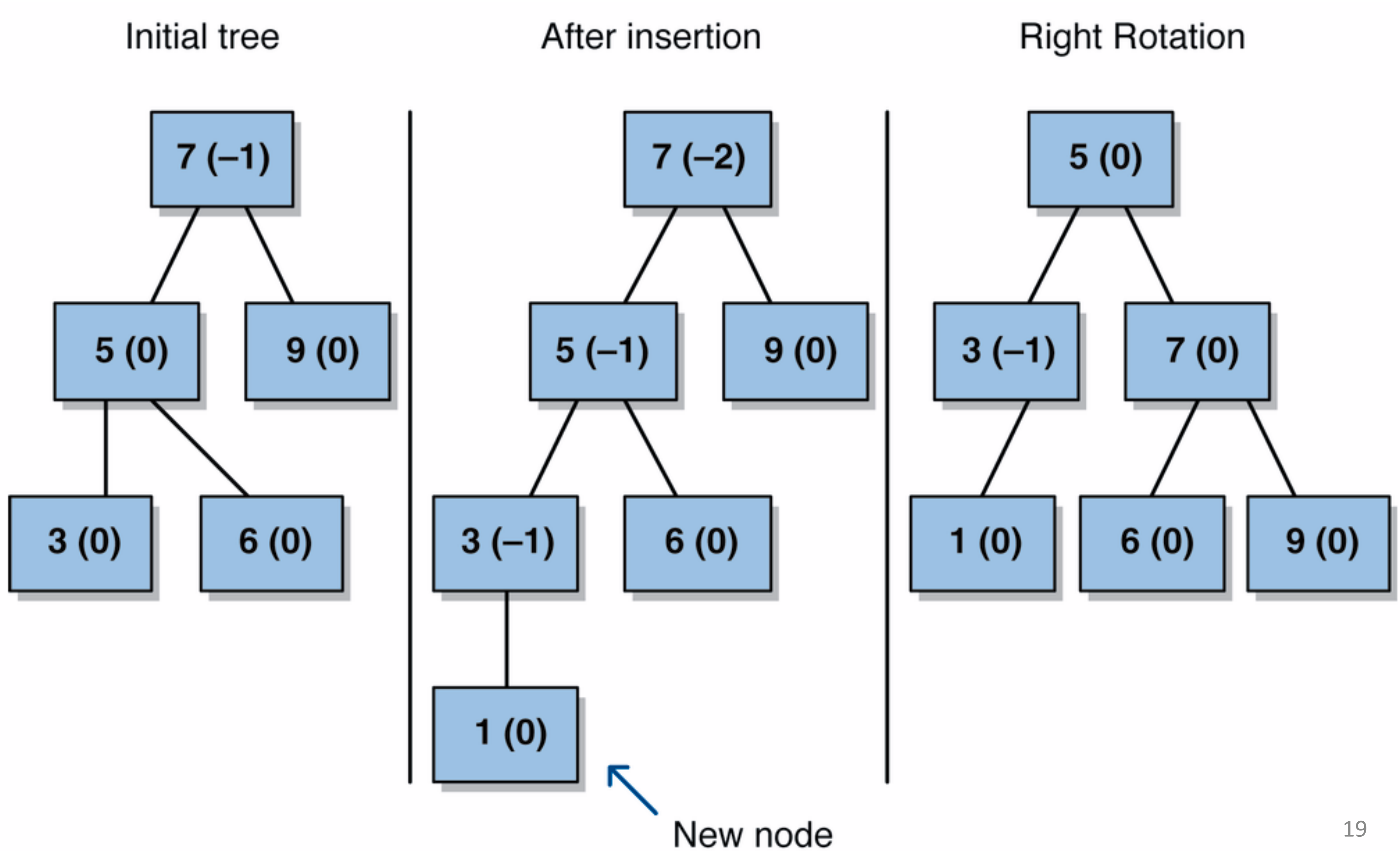
(b) After rotation

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



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;
}
```