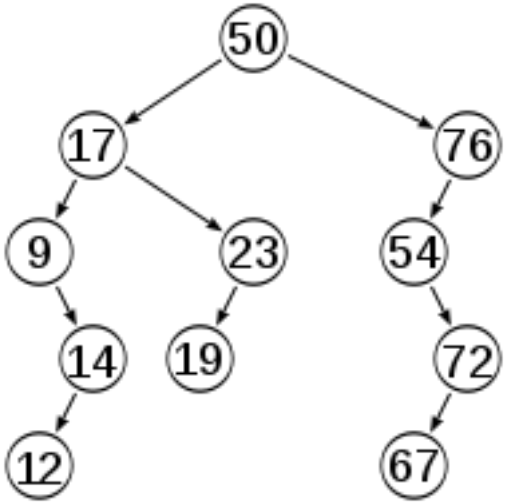
CSE 373
Data Structures and Algorithms

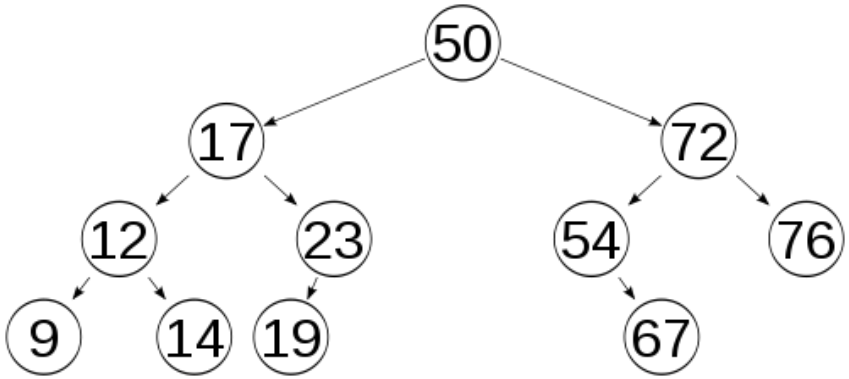
Lecture 11: Trees III (AVL Trees)

Balanced Tree

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



unbalanced tree



balanced tree

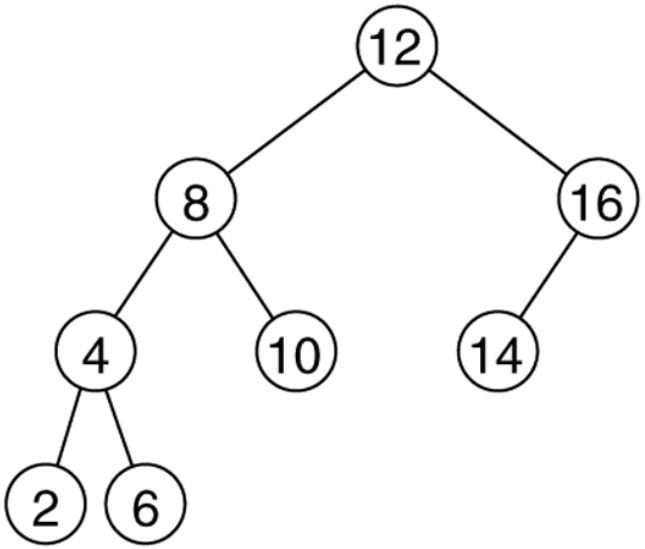
AVL trees

- **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
  - invented in 1962 by two mathematicians (Adelson-Velskii and Landis)
  - one of several auto-balancing trees (others in book)
  - 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

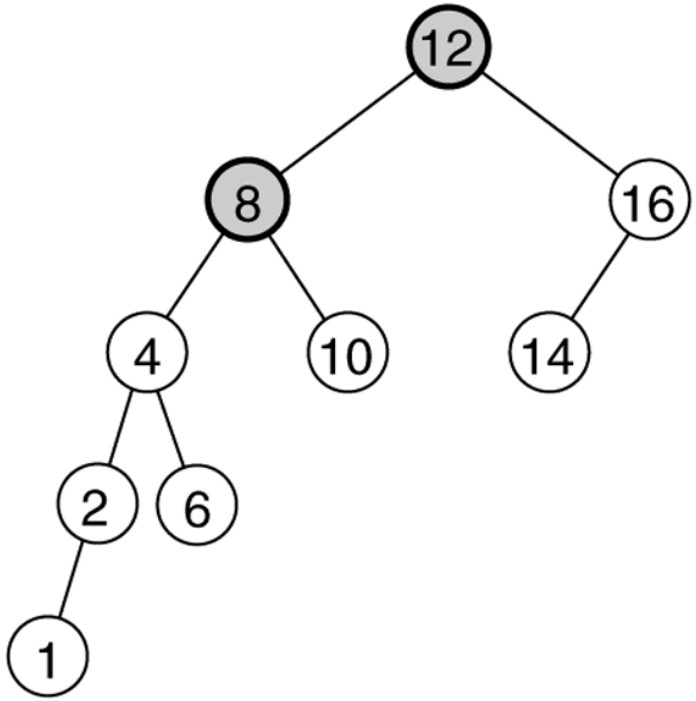
- **balance factor**, for a tree node *n* :
  - height of *n*'s right subtree minus height of *n*'s left subtree
  - $BF_n = Height_{n.right} - Height_{n.left}$
  - start counting heights at n

AVL tree examples

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

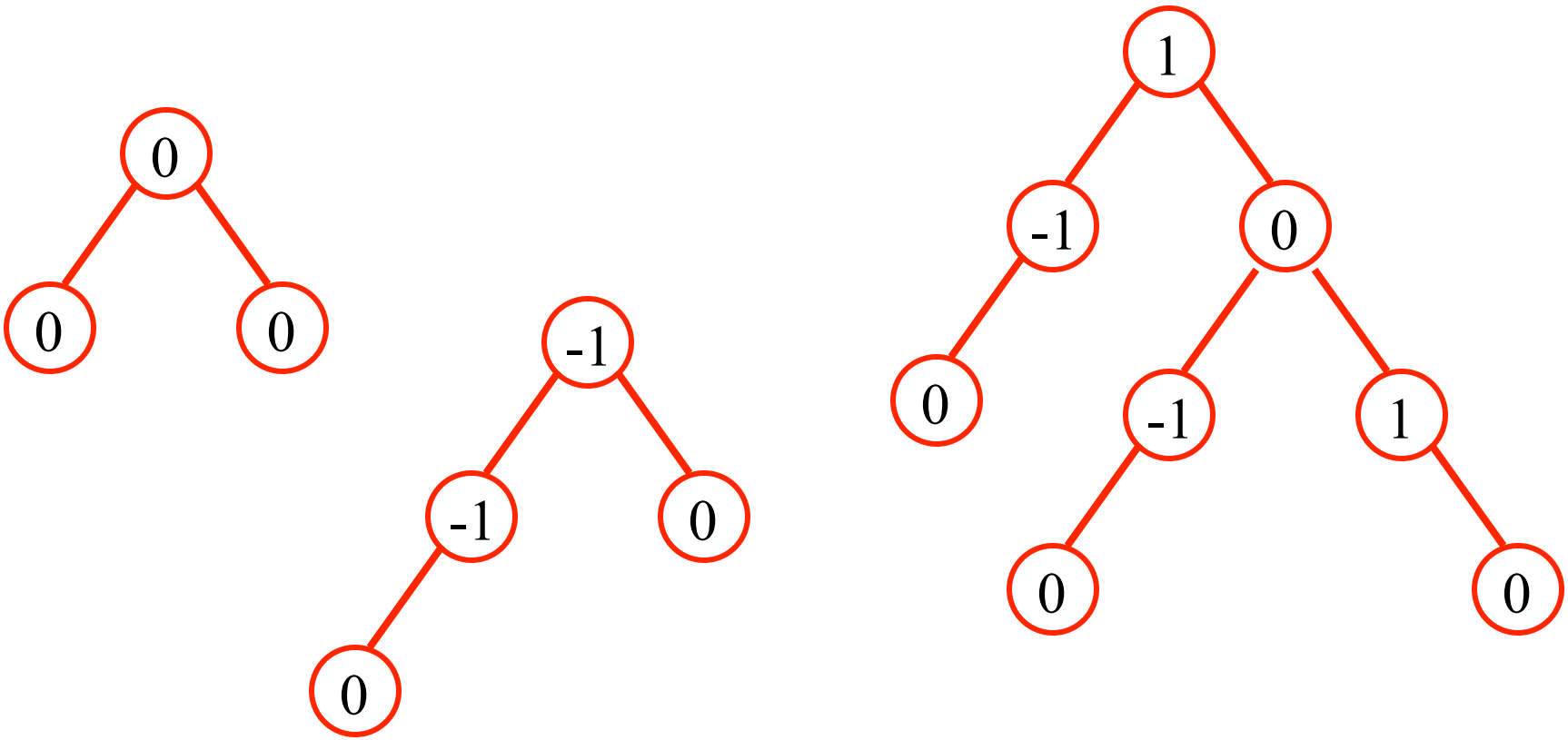


(a)

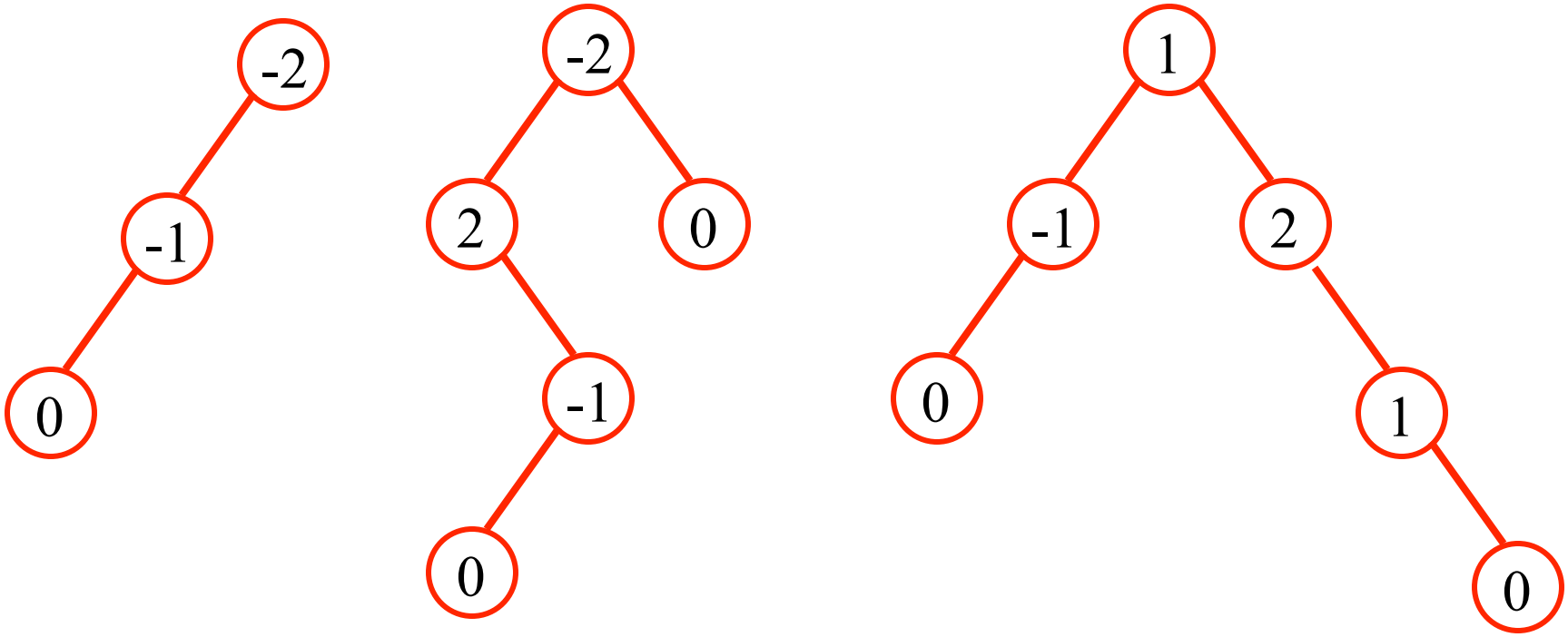


(b)

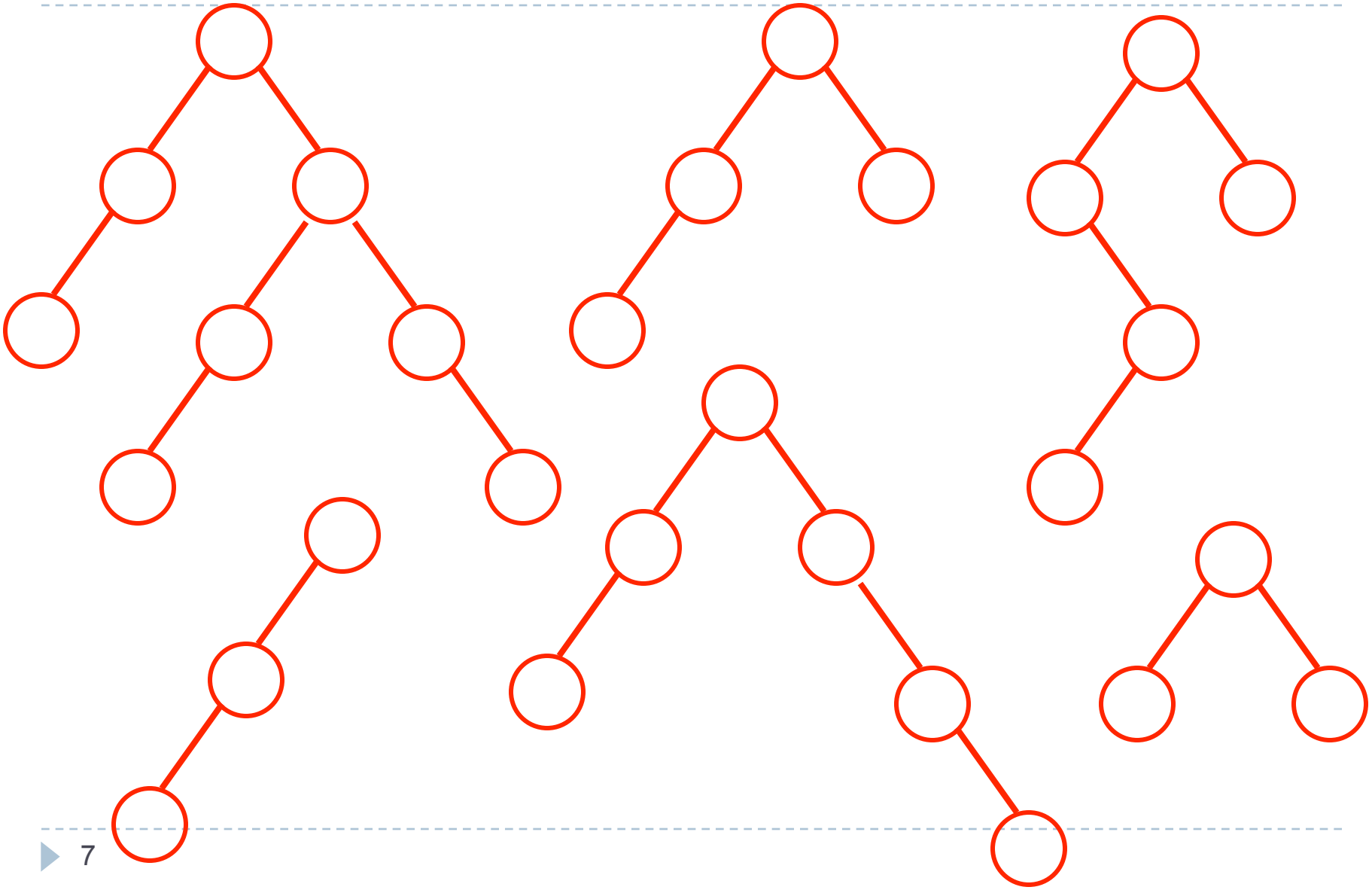
More AVL tree examples



Not AVL tree examples



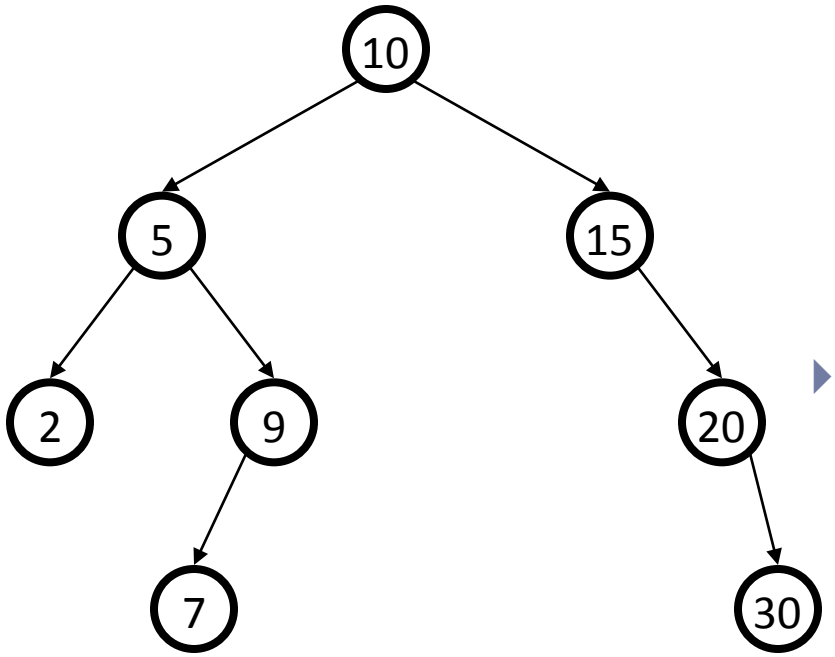
Which are AVL trees?



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.

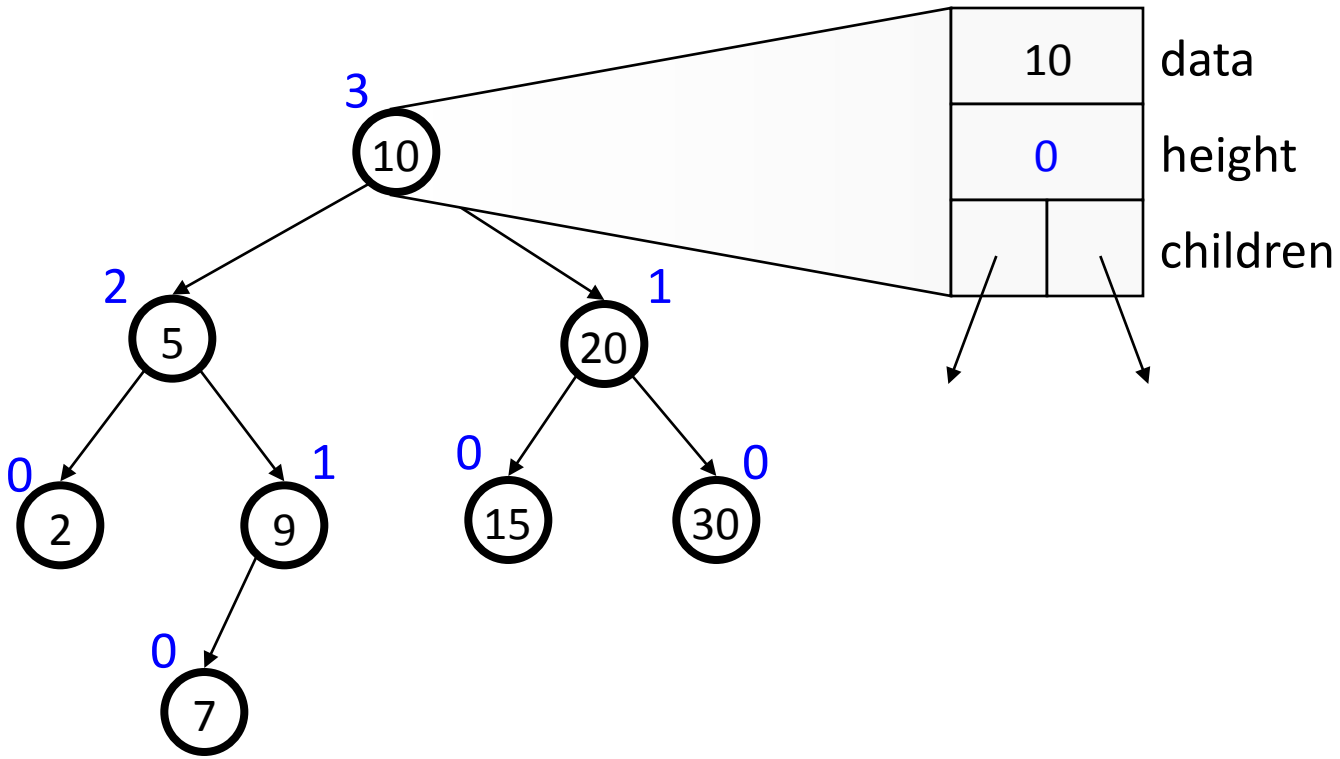
Testing the Balance Property



- We need to be able to:
  1. Track Balance Factor
  2. Detect Imbalance
  3. Restore Balance

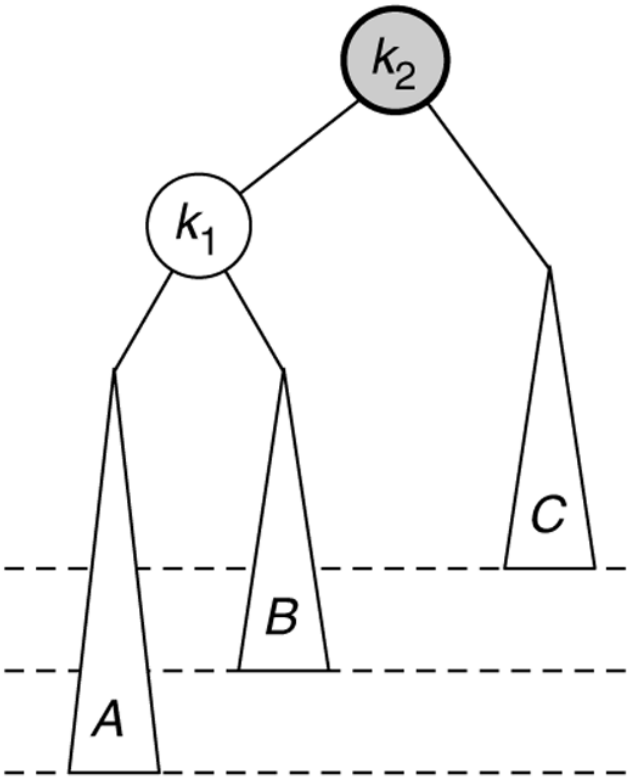
- How do we accomplish each step?

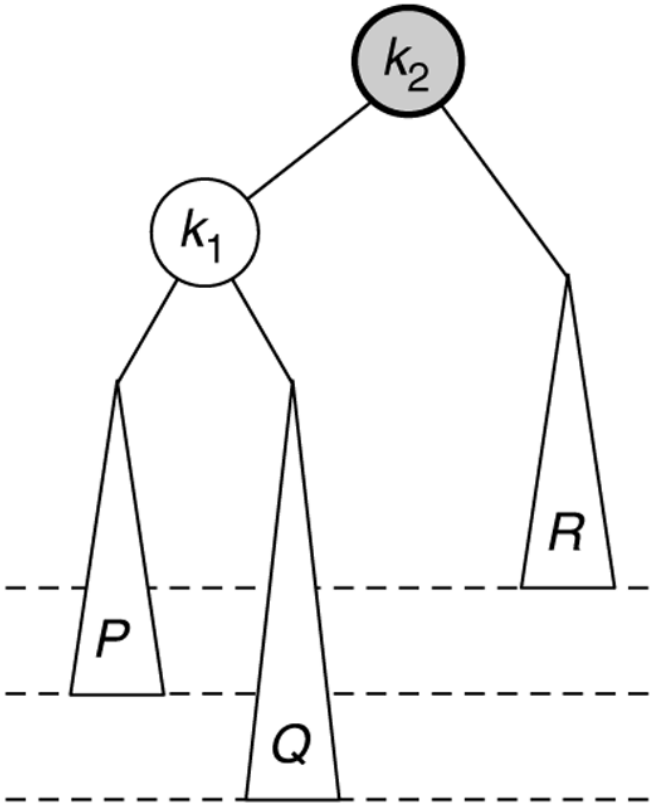
Tracking Balance



Problem Cases for AVL insert

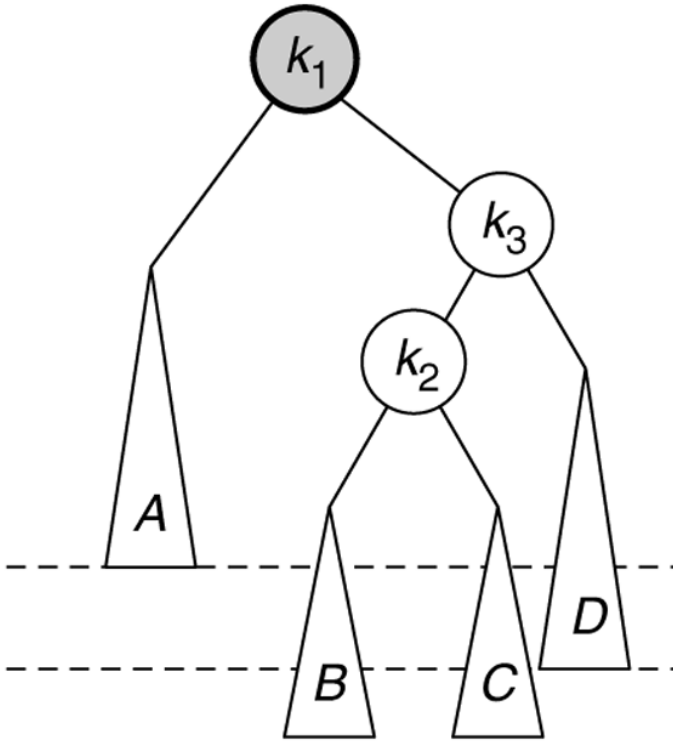
1. LL Case: insertion into left subtree of node's left child
2. LR Case: insertion into right subtree of node's left child

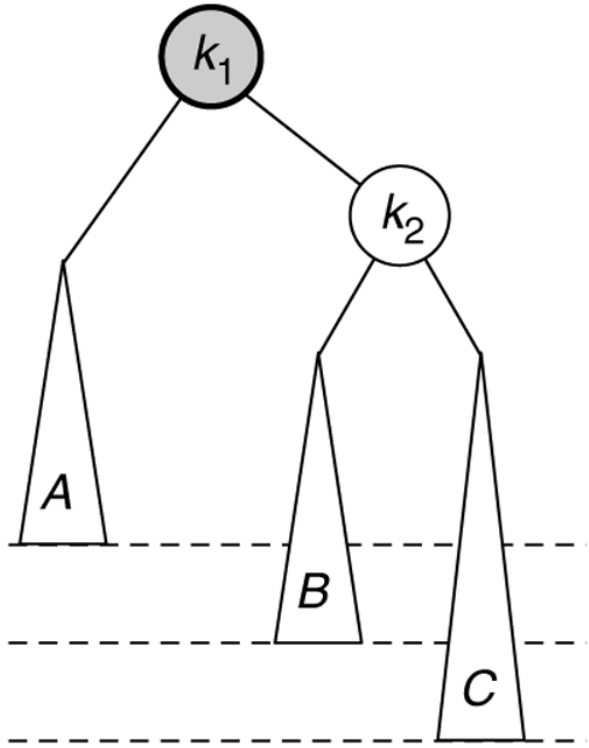




Problem Cases for AVL insert (cont'd)

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





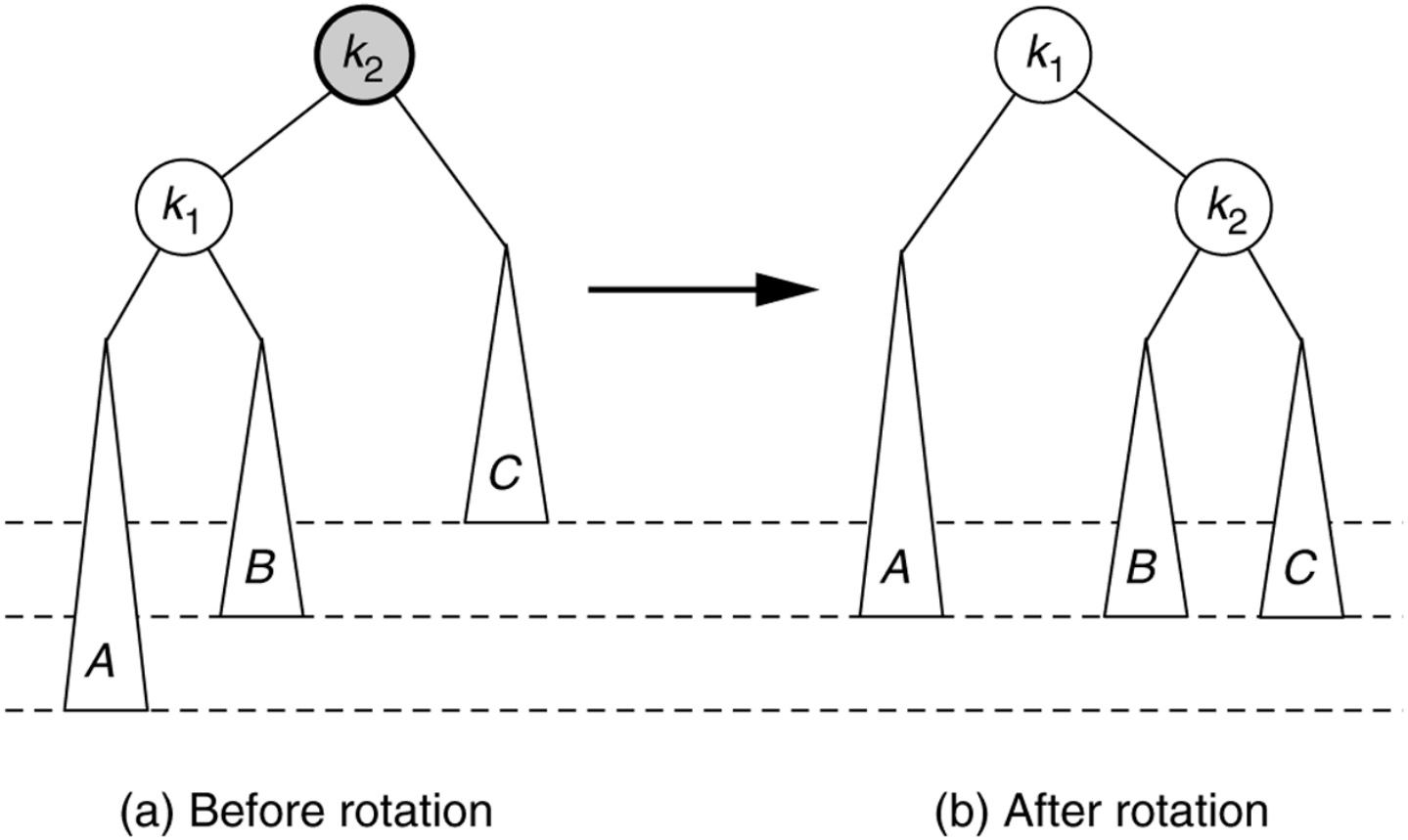
Maintaining Balance

- Maintain balance using *rotations*
  - The idea: locally reorganize the nodes of an unbalanced subtree until they are balanced, by "rotating" a trio of parent, left child, and right child

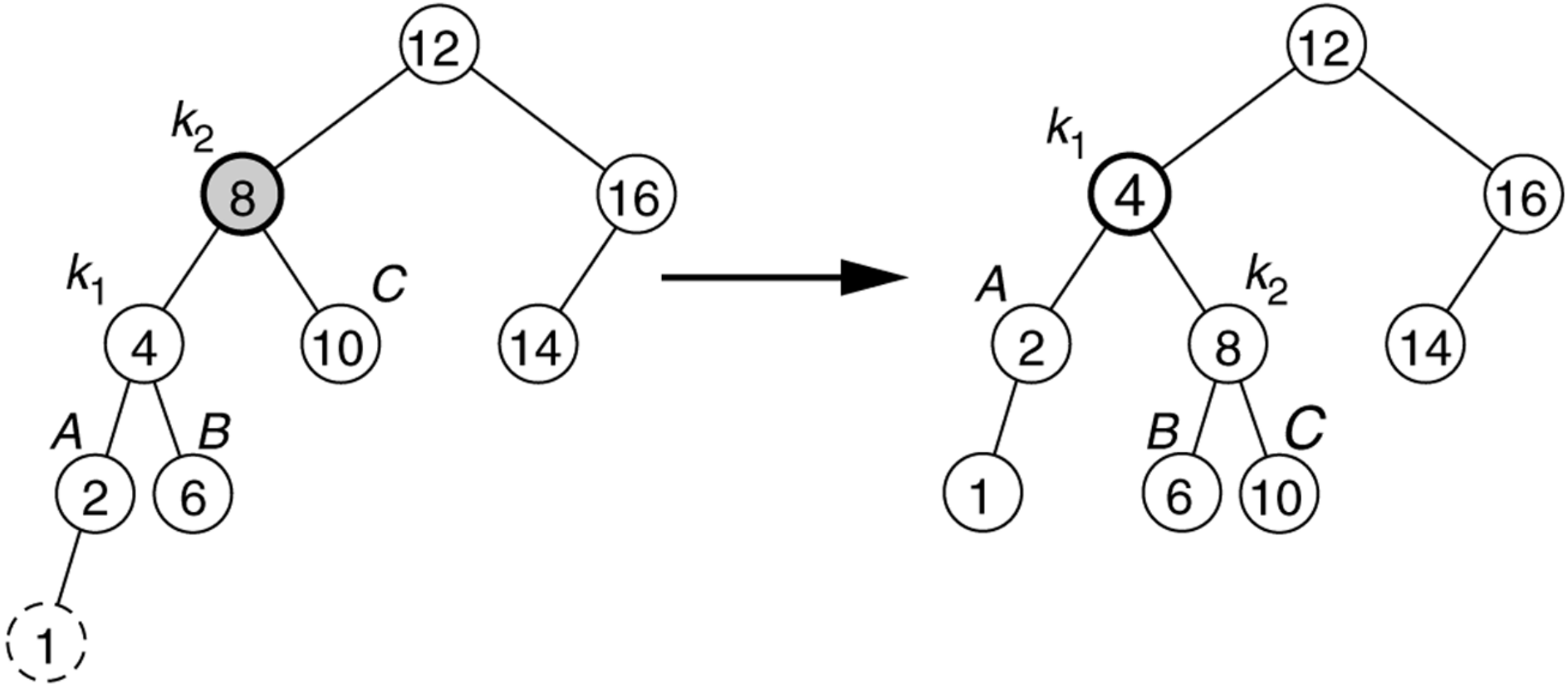
- Maintaining balance will result in searches (`contains`) that take $\Theta(\log n)$

Right rotation to fix Case 1 (LL)

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



Right rotation example

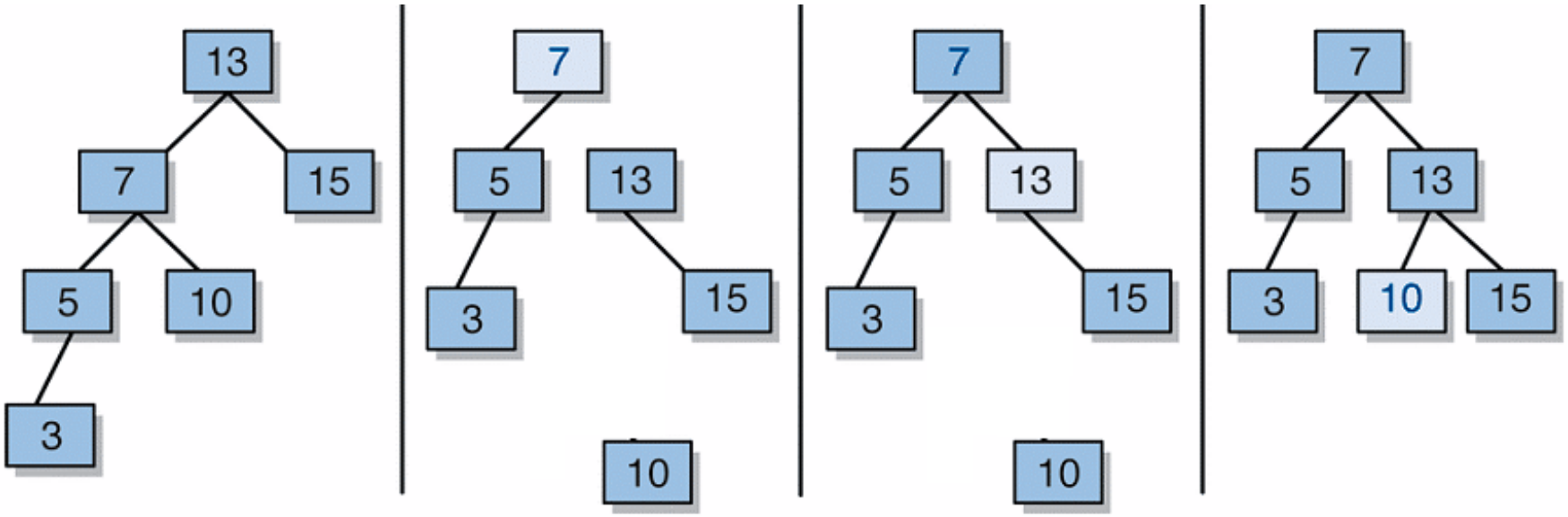


(a) Before rotation

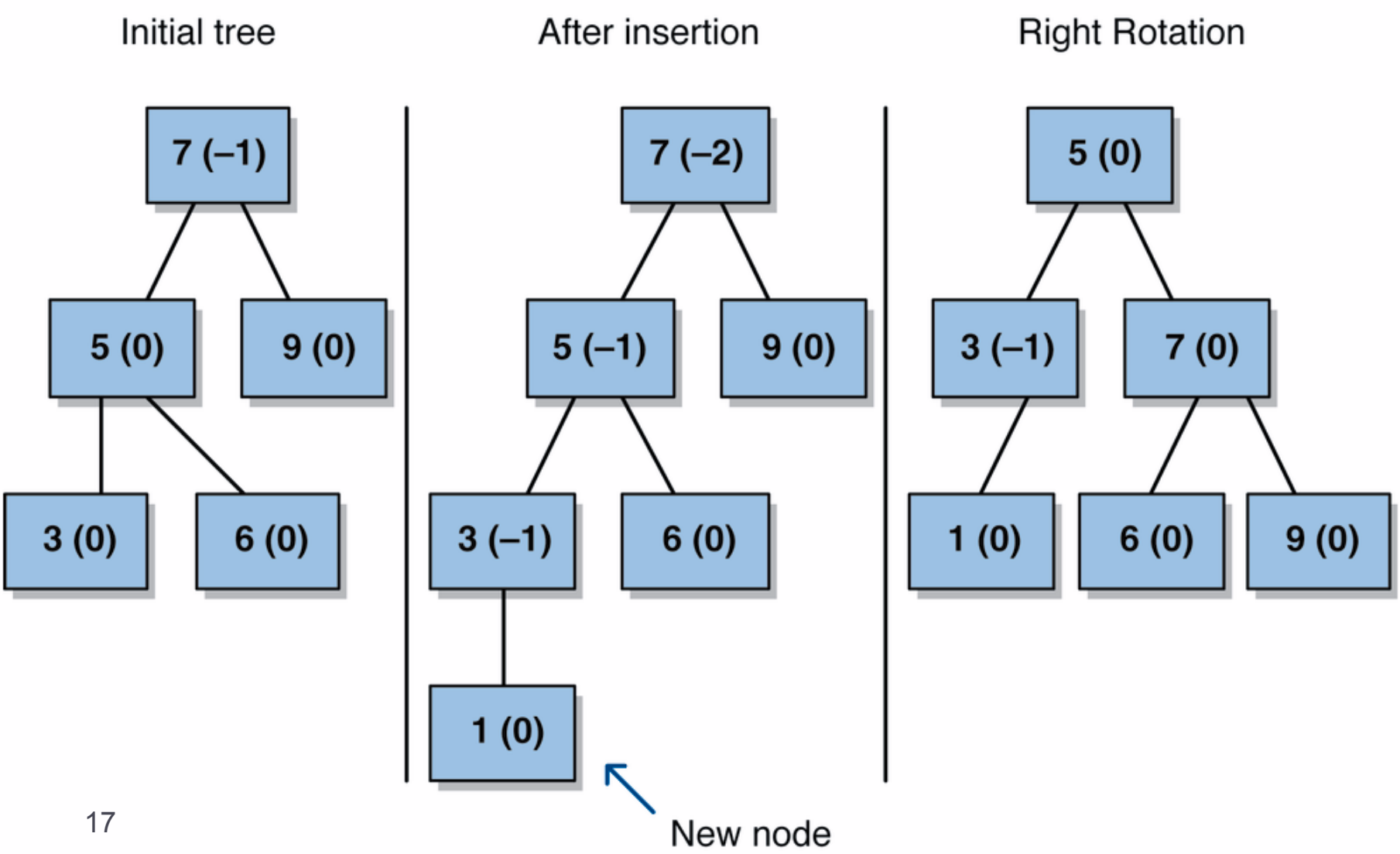
(b) After rotation

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



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;
}
```