CS3500: Object-Oriented Design
Spring 2014

Class 16
3.11.2014

Northeastern University
College *of* Computer and Information Science

Today...

- Assignment
- Visitor pattern
- Approaches to Balanced Trees

Assignment 8

- Due: Friday, March 14, 2014 at 11:59pm
- Visitor pattern

Visitor Pattern

Visitor Pattern

(From http://www.ccs.neu.edu/course/cs2510h/dpc.pdf. Version: April 8, 2013. Page 177.)

The visitor pattern is a general design pattern that allows you to seperate [sic] data from functionality in an object-oriented style.

For instance, suppose you want to develop a library of ranges. Users of your library are going to want a bunch of different methods, and in principal, you can't possibly know or want to implement all of them. On the other hand, you may not want to expose the implementation details of how you chose to represent ranges.

The visitor pattern can help—it requires you to implement one method which accepts what we call a "visitor" that is then exposed to a view of the data. Any computation over shapes can be implemented as a visitor, so this one method is universal—no matter how people want to use your library, this one method is enough to ensure they can write whatever computation they want. What's better is that even if you change the representation of ranges, so long as you provide the same "view" of the data, everything will continue to work.

Visitor Pattern: Intent

[Gamma et al.]

"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, Reading, MA, 1995.

When to Use Visitor Pattern

[Gamma et al.]

- Many classes of objects

- Distinct and unrelated operations to perform on object

- Object structures rarely change but add new operations often

Benefits of Using Visitor Pattern

[Gamma et al.]

- Adding new operations is easy
- Gathers related operations and separate unrelated ones
- Accumulating state

Visitor in Java

- Visitor implementation
  - visit method
- ADT implementation
  - accept method

Double Dispatch

- Client code
  - `o.accept(v)`
- Implementation of accept
  - `v.visit(n)`

Visitor with StackInt

```
/**
 * An interface for a visitor that performs on
 * ints
 *
 * @author Jessica Young Schmidt
 */
public interface IntVisitor {

    /**
     * Performs some arbitrary operation with a
     * int as parameter, and returns a
     * int as its result.
     *
     * @param x
     *              int to perform action on
     * @return result of performing action
     */
    public int visit(int x);
}
```

```
    /**
     * @param v
     *                  IntVisitor to use to
     *                  traverse the StackInt
     * @return a StackInt of the results
     *              returned by the IntVisitor
     *              when it visits the elements
     *              of this StackInt. (because
     *              we want each subclass to
     *              have this method)
     */
    public StackInt accept(IntVisitor v)
```

ICE: Implement Visitor

- visitor Height implements RBTVisitor such that computes the length of the longest path in the tree
- visitor CountNodes implements RBTVisitor such that returns an ArrayList with three elements—index 0: number of non-empty nodes, index 1: number of non-empty black nodes, index 2: number of non-empty red nodes

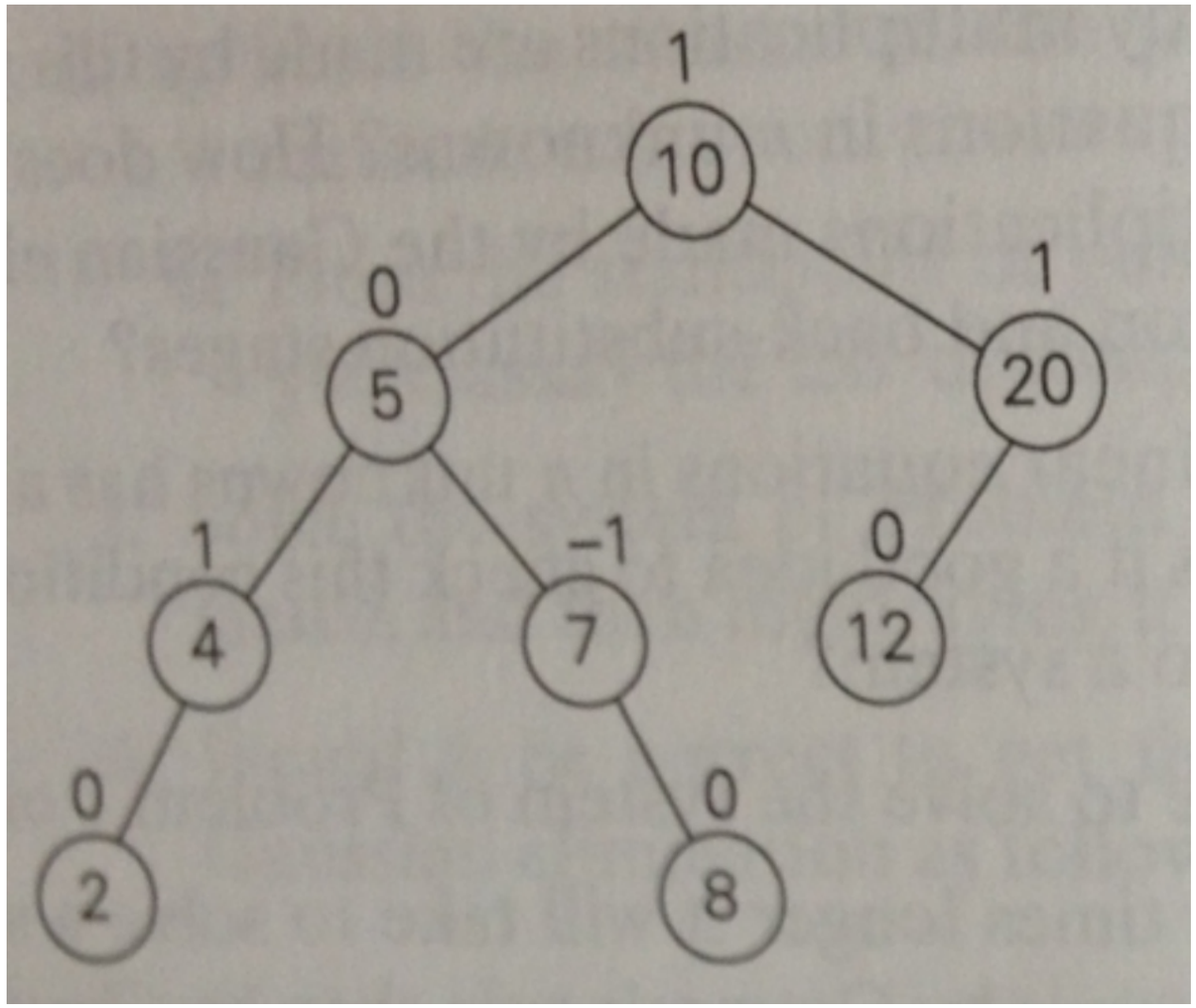
Other Approaches to Balanced Trees

- AVL Trees
- 2-3 Trees

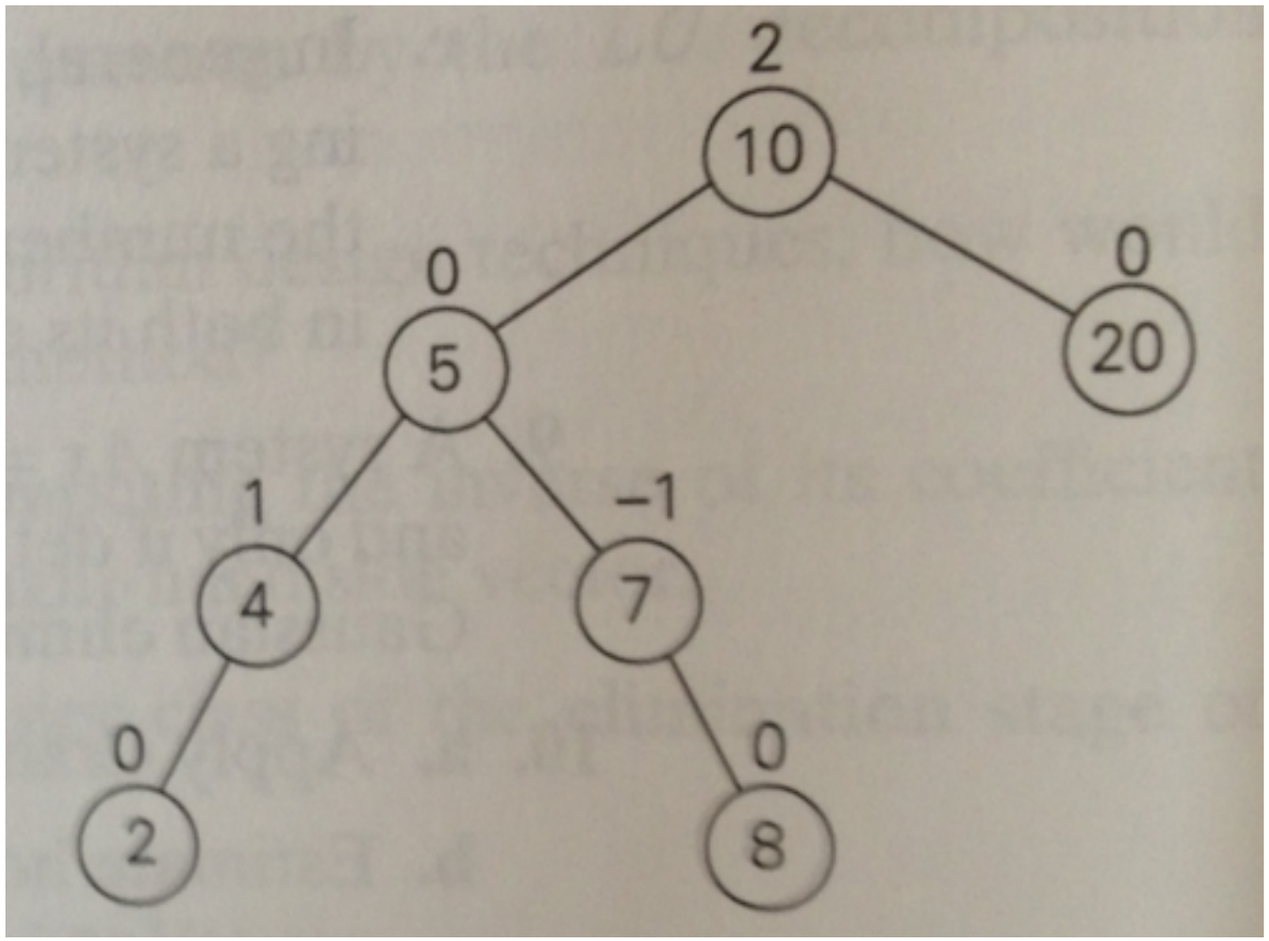
AVL Trees

[Levitin]

An *AVL tree* is a binary search tree in which the *balance factor* of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or -1. (The height of the empty tree is defined as -1.)



AVL Tree



Not AVL Tree

Figures from Levitin

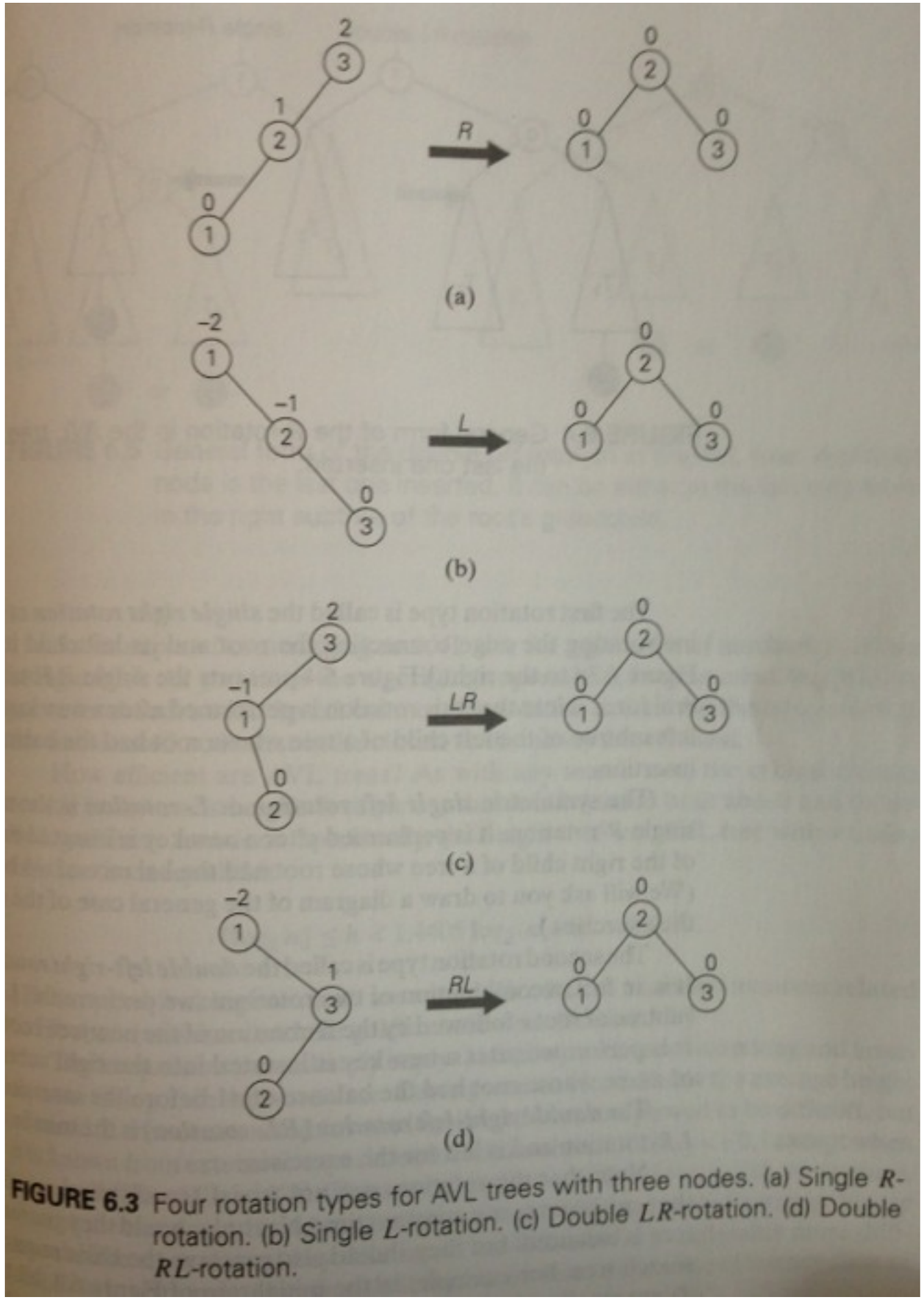


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

Figure from Levitin

2-3 Trees

[Levitin]

- A *2-3 tree* is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.
- A *2-node* contains a single key K and has two children.
- A *3-node* contains two ordered keys K1 and K2 (K1 < K2) and has three children.
- A 2-3 tree is always *height-balanced*

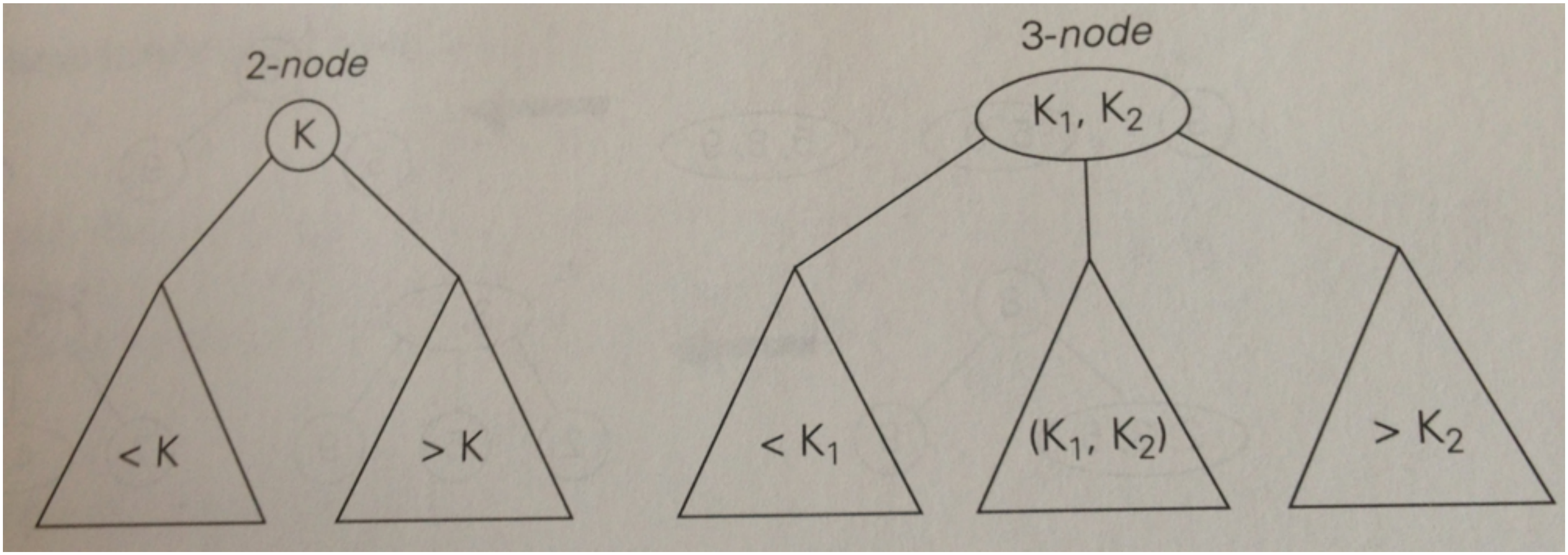


Figure from Levitin

Inserting into 2-3 Trees
[Levitin]

- Always insert a new key K at a leaf except for the empty tree.
- The appropriate leaf is found by performing a search of K.
- If the leaf in question is a 2-node, we insert K there as with the first or the second key, depending on whether K is smaller or larger than the node's old key.
- If the leaf is a 3-node, we split the leaf in two: the smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest is put in the second leaf, while the middle key is promoted to the old leaf's parent. (If the leaf happens to be the tree's root, a new root is created to accept the middle key.)
- Note that promotion of a middle key to its parent can cause the parent's overflow (if it was a 3-node) and hence can lead to several node splits along the chain of the leaf's ancestors.