AVL Trees

1

Objectives

- To know what an AVL tree is (§26.1).
- To understand how to rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§26.2).
- To know how to design the AVLTree class (§26.3).
- To insert elements into an AVL tree (§26.4).
- To implement node rebalancing (§26.5).
- To delete elements from an AVL tree (§26.6).
- To implement the AVLTree class (§26.7).
- To test the AVLTree class (§26.8).
- To analyze the complexity of search, insert, and delete operations in AVL trees (§26.9).

Why AVL Tree?

The search, insertion, and deletion time for a binary tree is dependent on the height of the tree. In the worst case, the height is O(n). If a tree is *perfectly balanced*, i.e., a complete binary tree, its height is . Can we maintain a perfectly balanced tree? Yes. But it will be costly to do so. The compromise is to maintain a well-balanced tree, i.e., the heights of two subtrees for every node are about the same.

What is an AVL Tree?

AVL trees are well-balanced. AVL trees were invented by two Russian computer scientists G. M. Adelson-Velsky and E. M. Landis in 1962. In an AVL tree, the difference between the heights of two subtrees for every node is 0 or 1. It can be shown that the maximum height of an AVL tree is O(logn).

AVL Tree Animation

https://liveexample.pearsoncmg.com/dsanimation/AVLTree.html
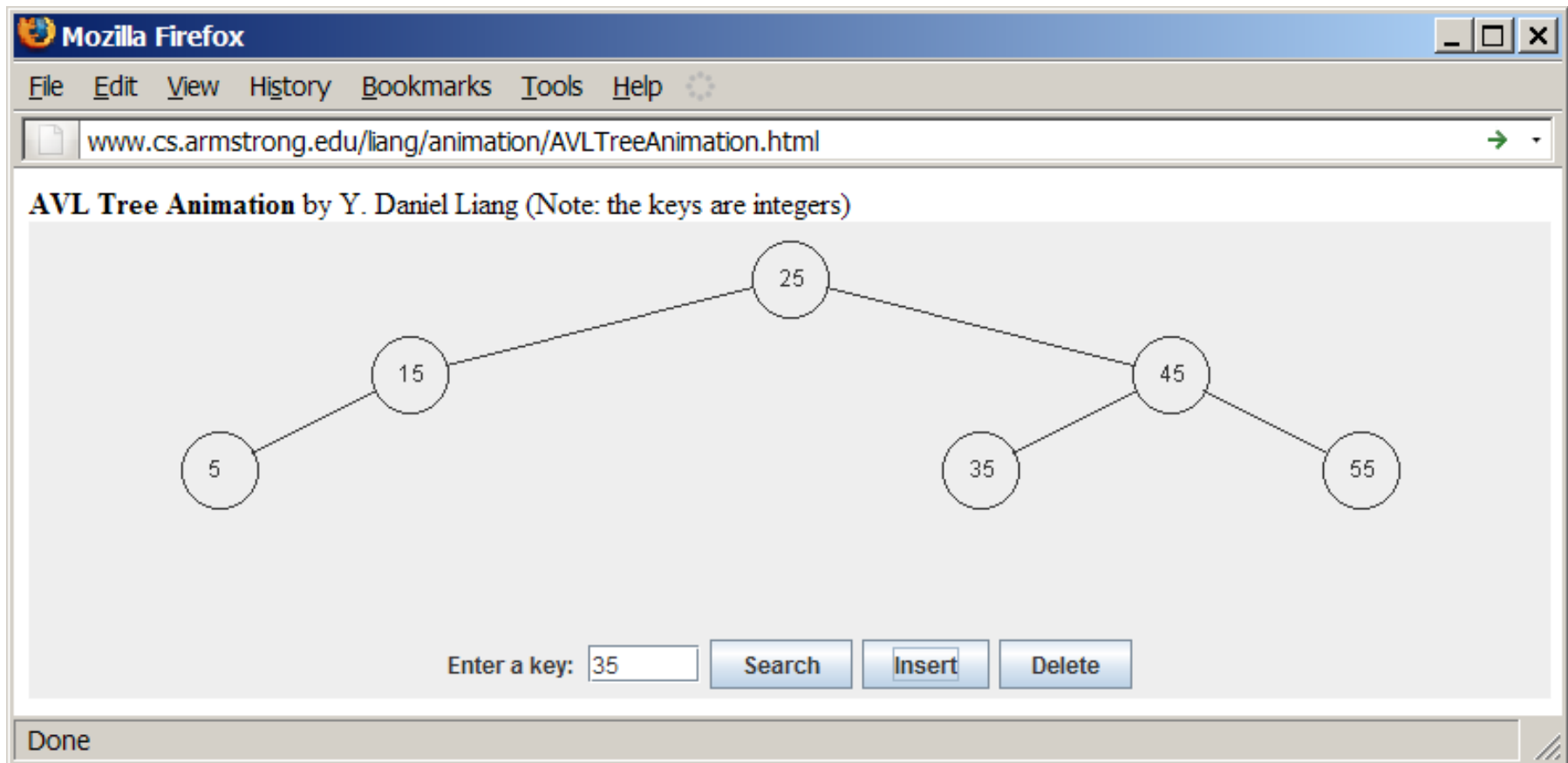
Balance Factor/Left-Heavy/Right-Heavy

The process for inserting or deleting an element in an AVL tree is the same as in a regular binary search tree. The difference is that you may have to rebalance the tree after an insertion or deletion operation. The *balance factor* of a node is the height of its right subtree minus the height of its left subtree. A node is said to be *balanced* if its balance factor is -1, 0, or 1. A node is said to be *left-heavy* if its balance factor is -1. A node is said to be *right-heavy* if its balance factor is +1.

Balancing Trees

If a node is not balanced after an insertion or deletion operation, you need to rebalance it. The process of rebalancing a node is called a *rotation*. There are four possible rotations.

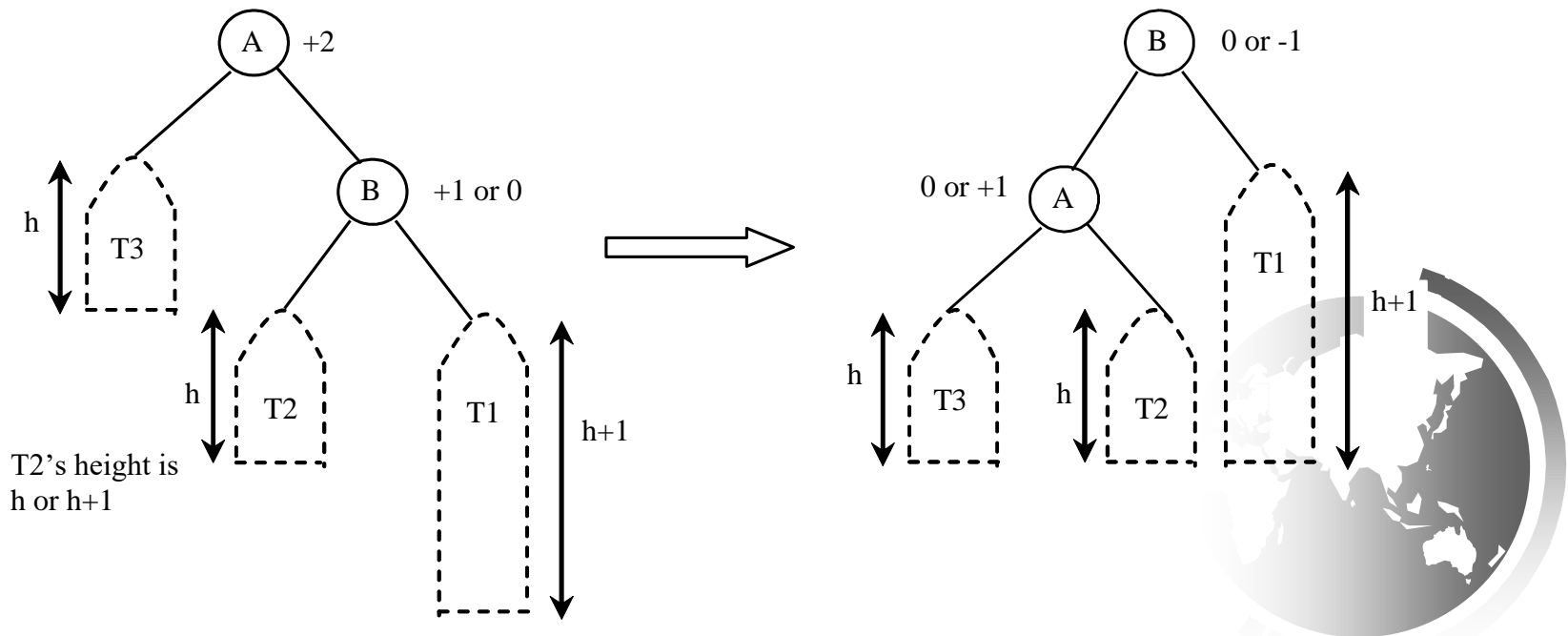# LL imbalance and LL rotation

LL Rotation: An *LL imbalance* occurs at a node A such that A has a balance factor -2 and a left child B with a balance factor -1 or 0. This type of imbalance can be fixed by performing a single right rotation at A.
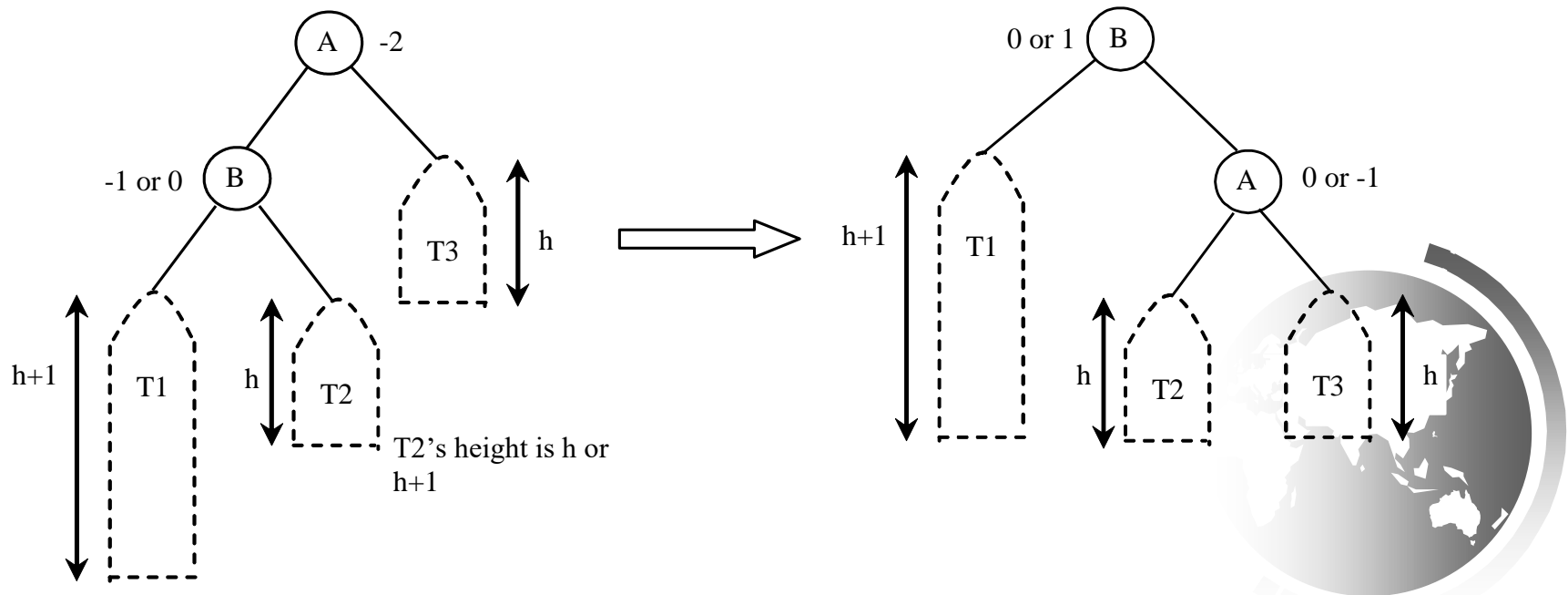
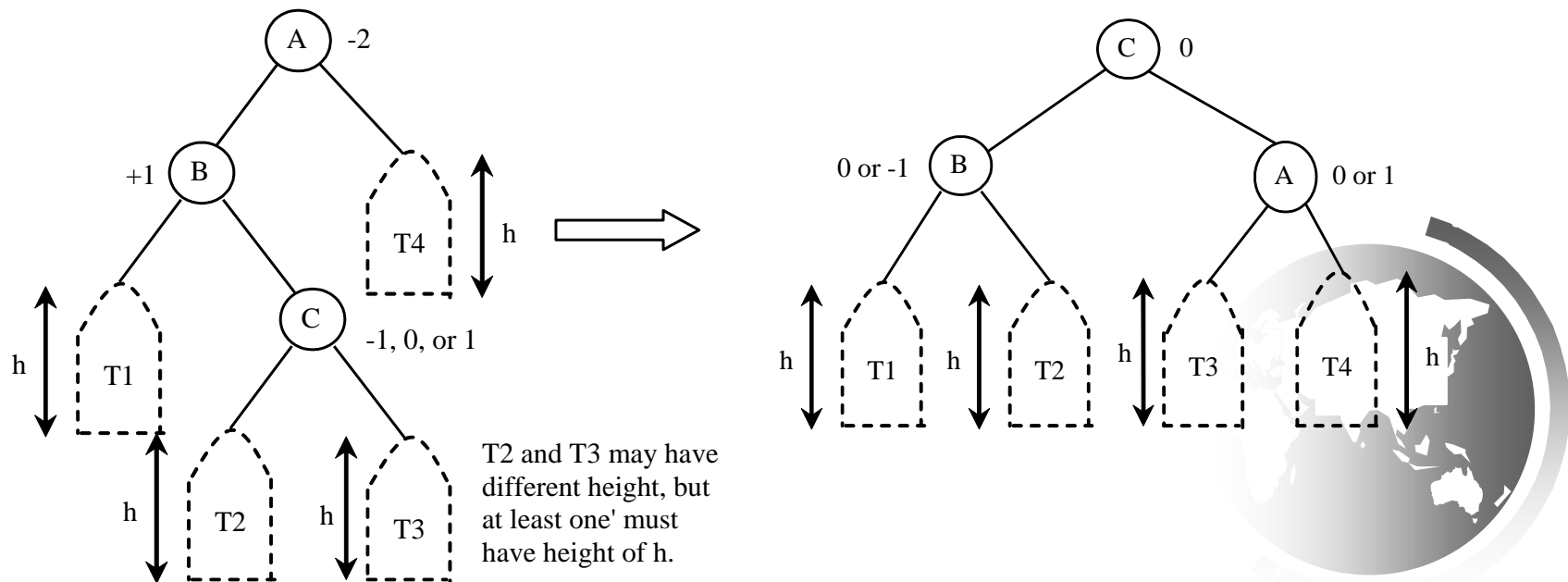# RR imbalance and RR rotation

RR Rotation: An *RR imbalance* occurs at a node A such that A has a balance factor +2 and a right child B with a balance factor +1 or 0. This type of imbalance can be fixed by performing a single left rotation at A.

LR imbalance and LR rotation

LR Rotation: An *LR imbalance* occurs at a node A such that A has a balance factor -2 and a left child B with a balance factor +1. Assume B's right child is C. This type of imbalance can be fixed by performing a double rotation (first a single left rotation at B and then a single right rotation at A).

T2 and T3 may have different height, but at least one' must have height of h.
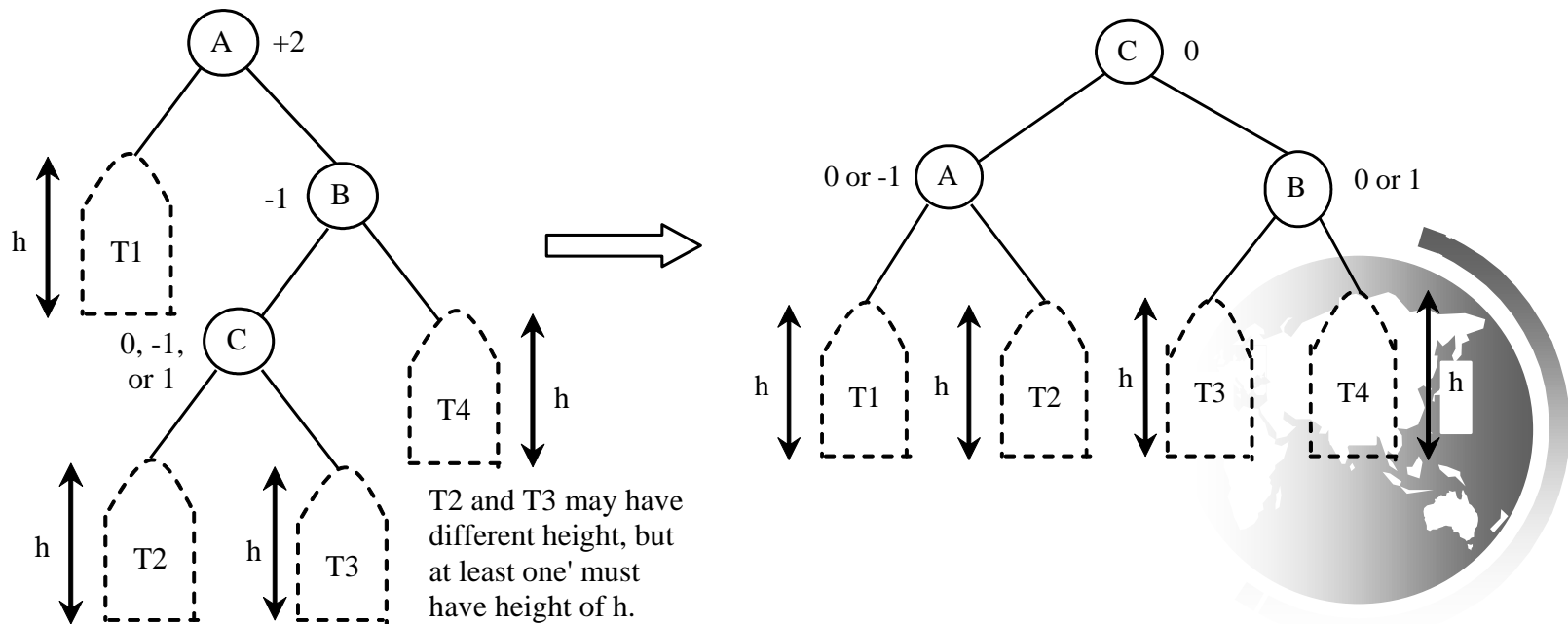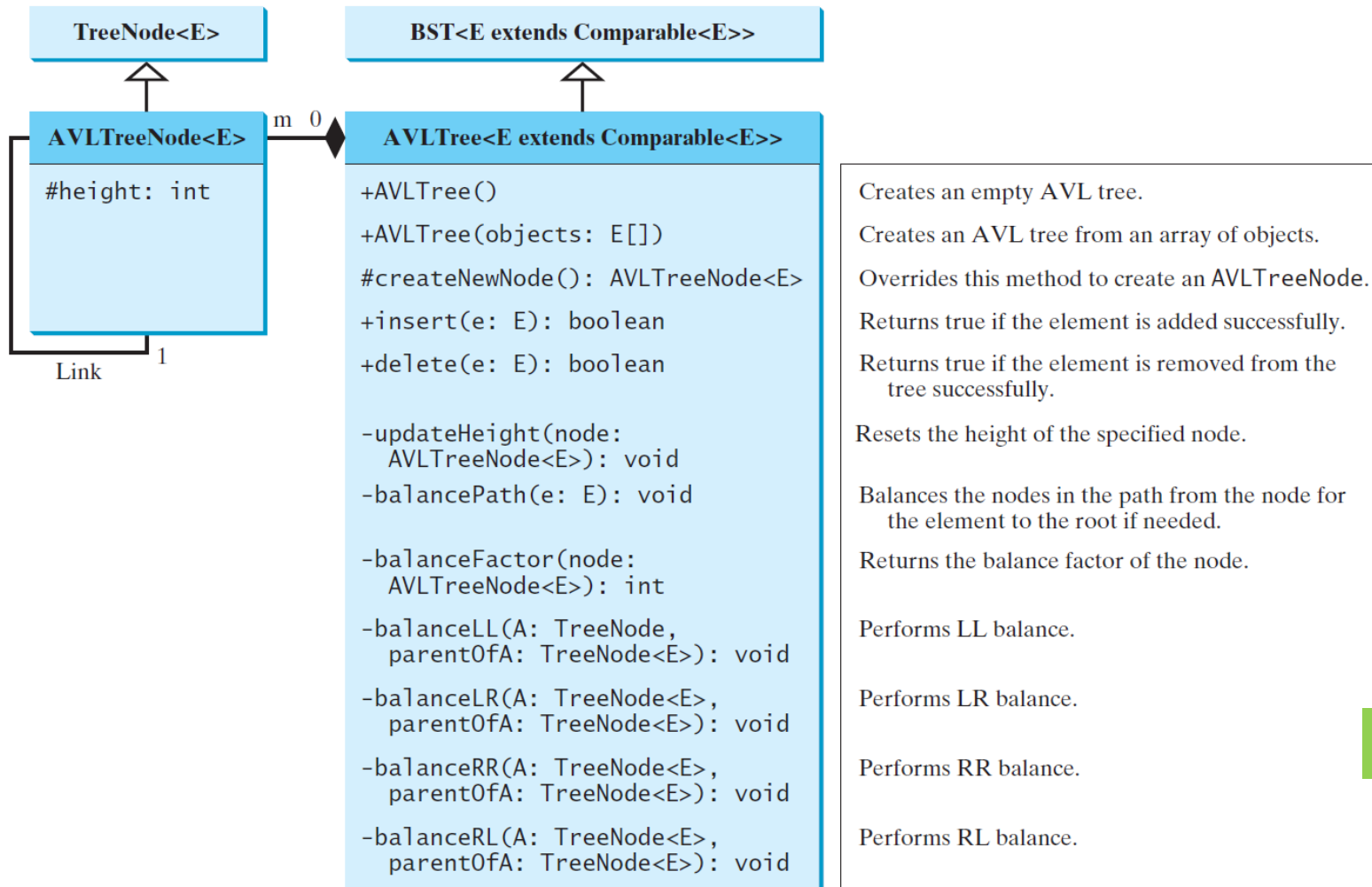
RL imbalance and RL rotation

RL Rotation: An *RL imbalance* occurs at a node A such that A has a balance factor +2 and a right child B with a balance factor -1. Assume B's left child is C. This type of imbalance can be fixed by performing a double rotation (first a single right rotation at B and then a single left rotation at A).

Designing Classes for AVL Trees

An AVL tree is a binary tree. So you can define the AVLTree class to extend the BST class.

AVLTree

TreeNode<E>

BST<E extends Comparable<E>>

AVLTreeNode<E>

#height: int

Link

1

m   0

AVLTree<E extends Comparable<E>>

+AVLTree()

+AVLTree(objects: E[])

#createNewNode(): AVLTreeNode<E>

+insert(e: E): boolean

+delete(e: E): boolean

-updateHeight(node:
    AVLTreeNode<E>): void
-balancePath(e: E): void

-balanceFactor(node:
    AVLTreeNode<E>): int
-balanceLL(A: TreeNode,
    parentOfA: TreeNode<E>): void

-balanceLR(A: TreeNode<E>,
    parentOfA: TreeNode<E>): void

-balanceRR(A: TreeNode<E>,
    parentOfA: TreeNode<E>): void

-balanceRL(A: TreeNode<E>,
    parentOfA: TreeNode<E>): void

Creates an empty AVL tree.

Creates an AVL tree from an array of objects.

Overrides this method to create an AVLTreeNode.

Returns true if the element is added successfully.

Returns true if the element is removed from the tree successfully.

Resets the height of the specified node.

Balances the nodes in the path from the node for the element to the root if needed.

Returns the balance factor of the node.

Performs LL balance.

Performs LR balance.

Performs RR balance.

Performs RL balance.

TestAVLTree

Run