# Balanced Binary Search Trees: AVL Trees

Michael Hahsler

With figures from:
M.A. Weiss, Data Structures and Algorithm Analysis,4th edition

# Binary Search Trees

**Definition:** Binary tree + order property.

**Order property**: *For every node in the tree, the items in each left subtree are smaller than the node and the items in the right subtree are larger. This requires that a total order over nodes is defined.*

**Time complexity**

- The depth of a binary search tree $d$ leads to $O(d)$ operations (for all but deleting and copying the whole tree).
- The average tree depth $d$ is $O(\log\ N)$ under the assumption that all insertion sequences are equally likely. $O(\log\ N)$ is the result of the observation that every additional level in a tree roughly doubles the number of nodes.

**Problems:** Situations with close to the worst-case time complexity of $O(N)$ are likely to happen:

- Items are inserted in (almost) sorted order.
- Deletions often replace a node with a node for the right subtree, resulting in an unbalanced tree that is left heavy!

# AVL Trees

An AVL (Adelson-Velskii and Landis) tree is an autobalancing binary search tree with the following **balance condition**:

*For every node in the tree, the height of the left and the right subtree can differ by at most 1.*

Missing nodes have height = -1

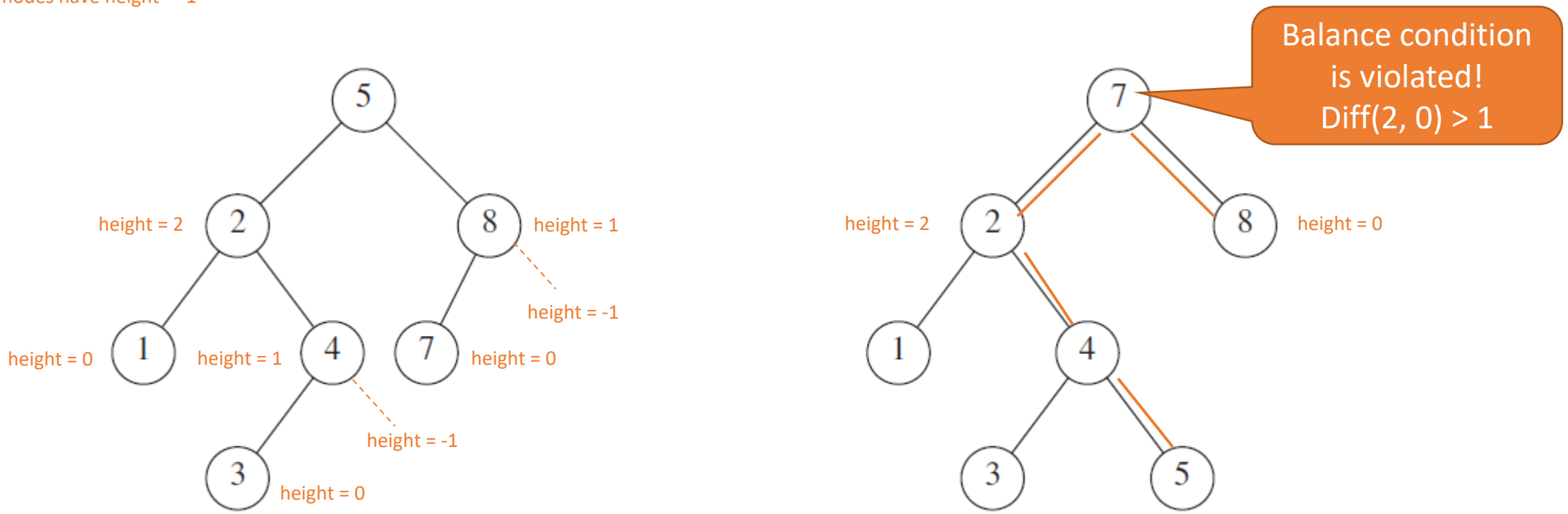



**Figure 4.32** Two binary search trees. Only the left tree is AVL.

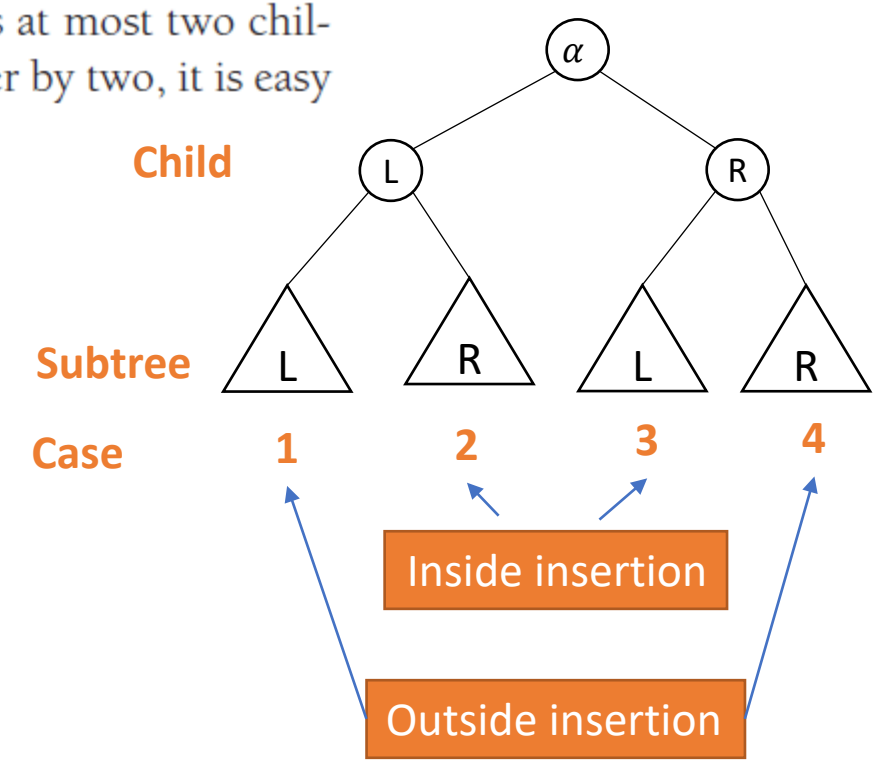
# Rebalancing

Only necessary locally (in the subtree) where the tree gets changed by an insertion (or deletion). If we store and update height information in the nodes, then detection is easy!

Cases:

Let us call the node that must be rebalanced $\alpha$. Since any node has at most two children, and a height imbalance requires that $\alpha$'s two subtrees' heights differ by two, it is easy to see that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of $\alpha$
2. An insertion into the right subtree of the left child of $\alpha$
3. An insertion into the left subtree of the right child of $\alpha$
4. An insertion into the right subtree of the right child of $\alpha$



# Single Rotation

Insertion in X

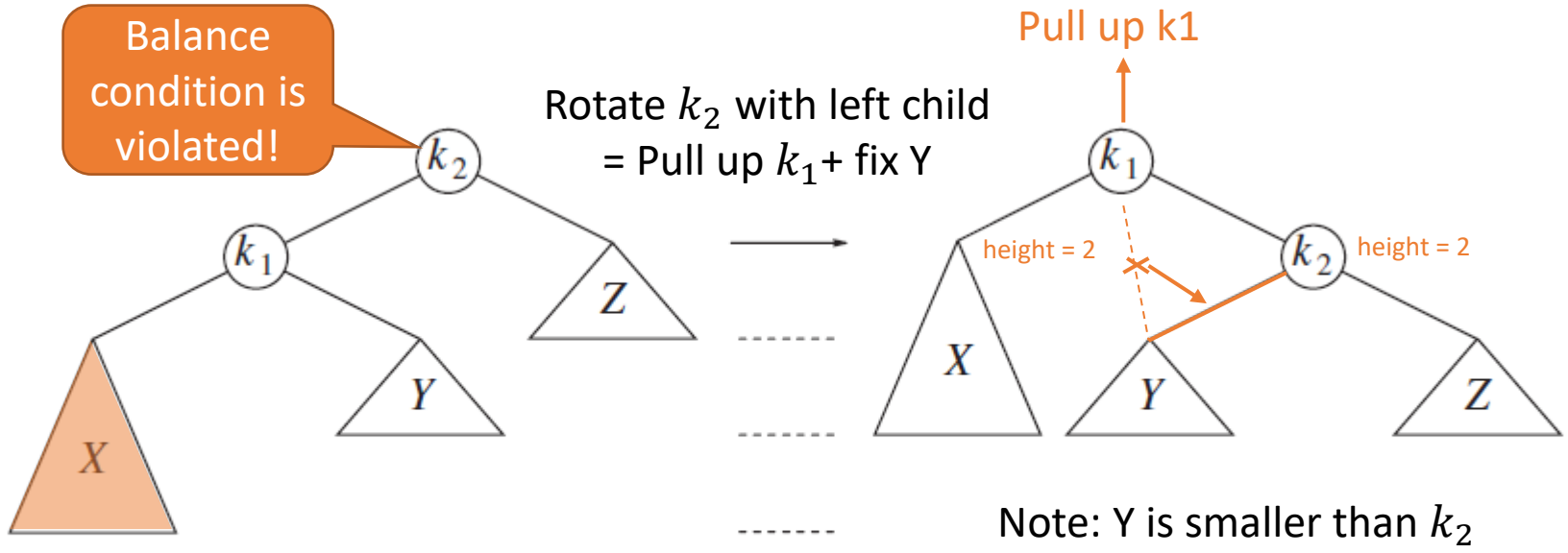



**Figure 4.34** Single rotation to fix case 1: lifts up X by one level.

Insertion in Z

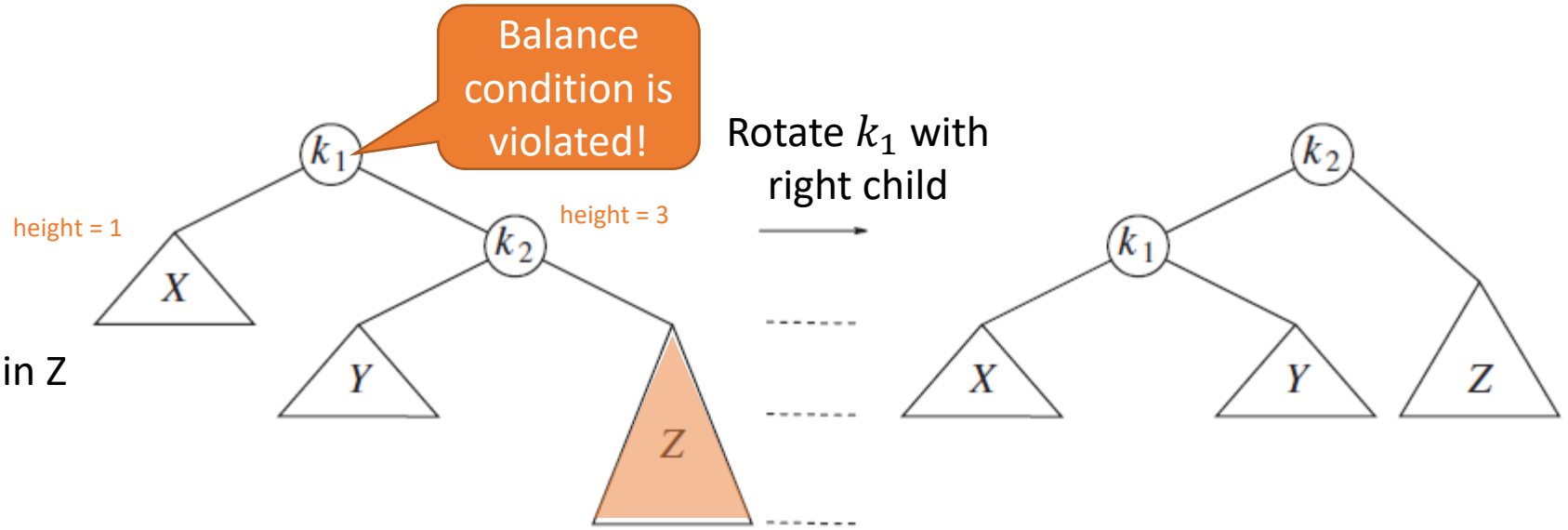



**Figure 4.36** Single rotation fixes case 4 : lifts up Z by one level.

**Outside Insertion**

These cases are perfectly symmetric!

# Example:

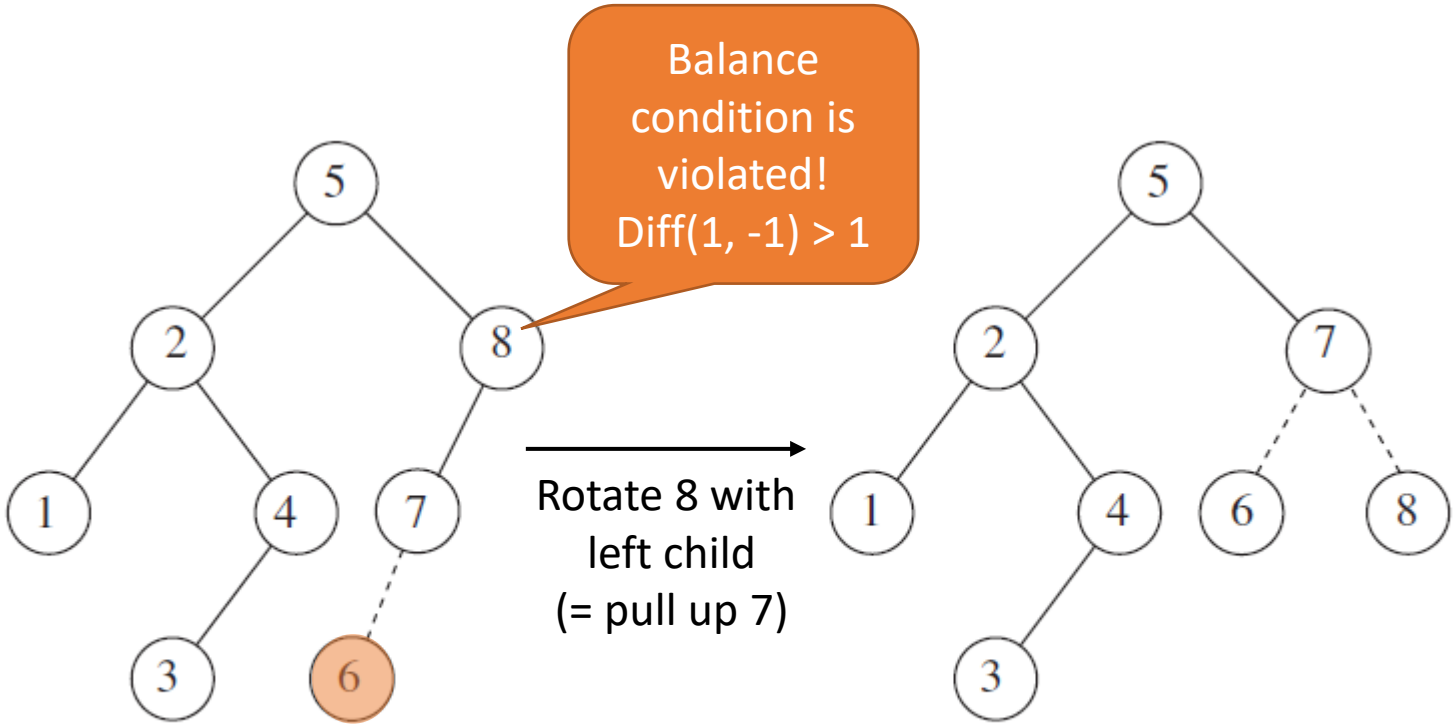



**Figure 4.35** AVL property destroyed by insertion of 6, then fixed by a single rotation

Note: There is no right subtree of 7 (Y) to fix since 7 has only a single child.

# Single Rotation Fails

Insertion in Y makes it too deep

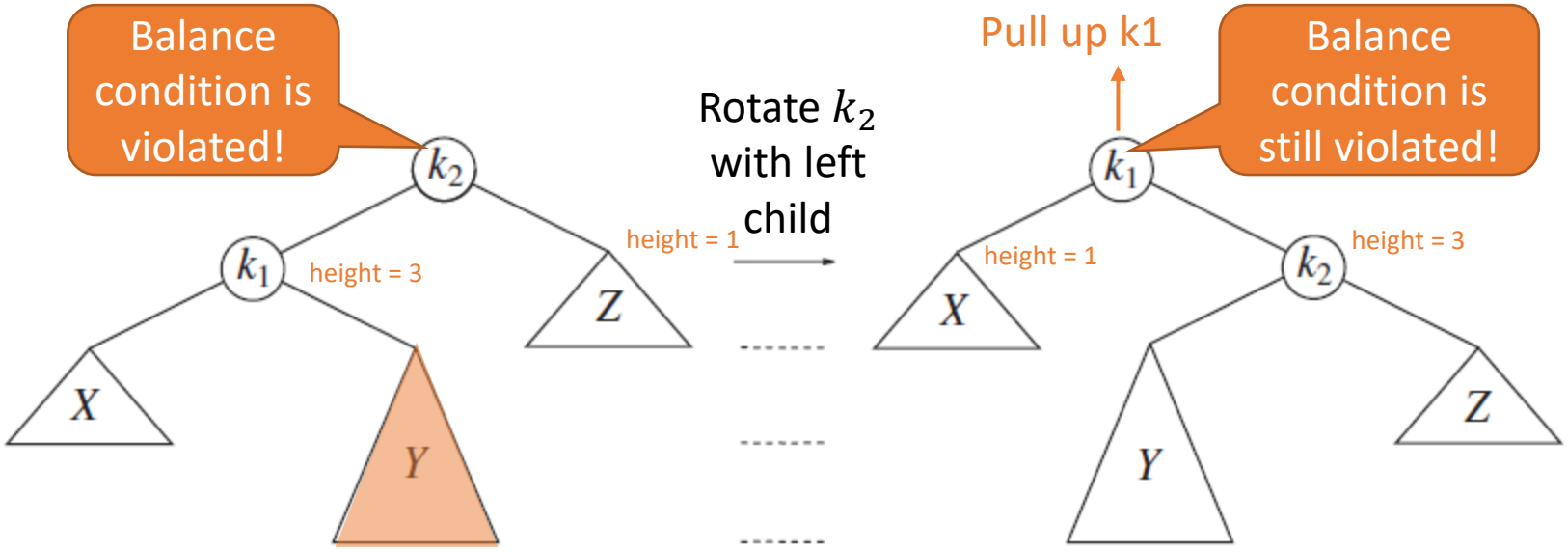
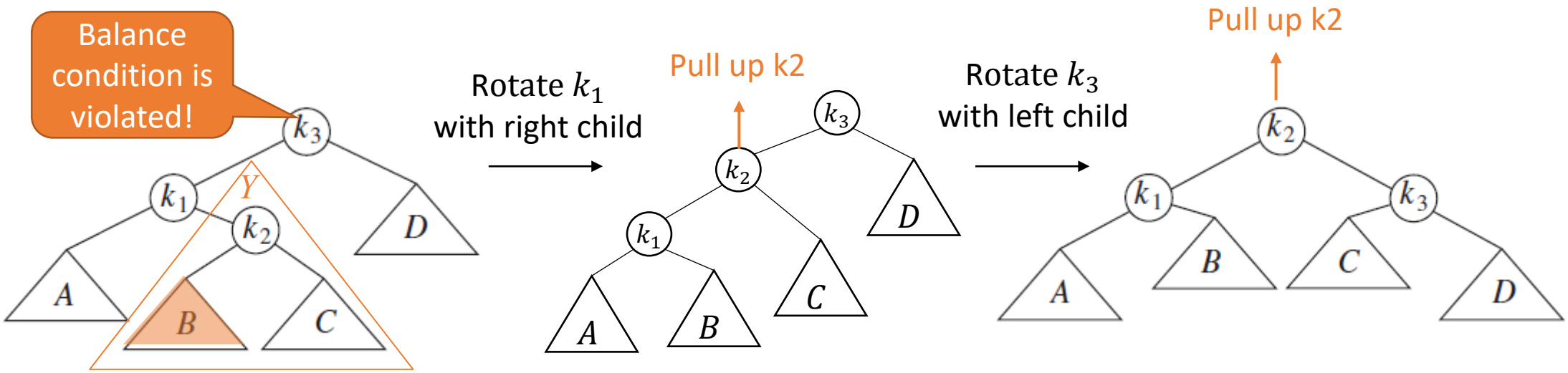



**Figure 4.37** Single rotation fails to fix case 2 : does not lift up Y!

We need a double rotation



# Double Rotation

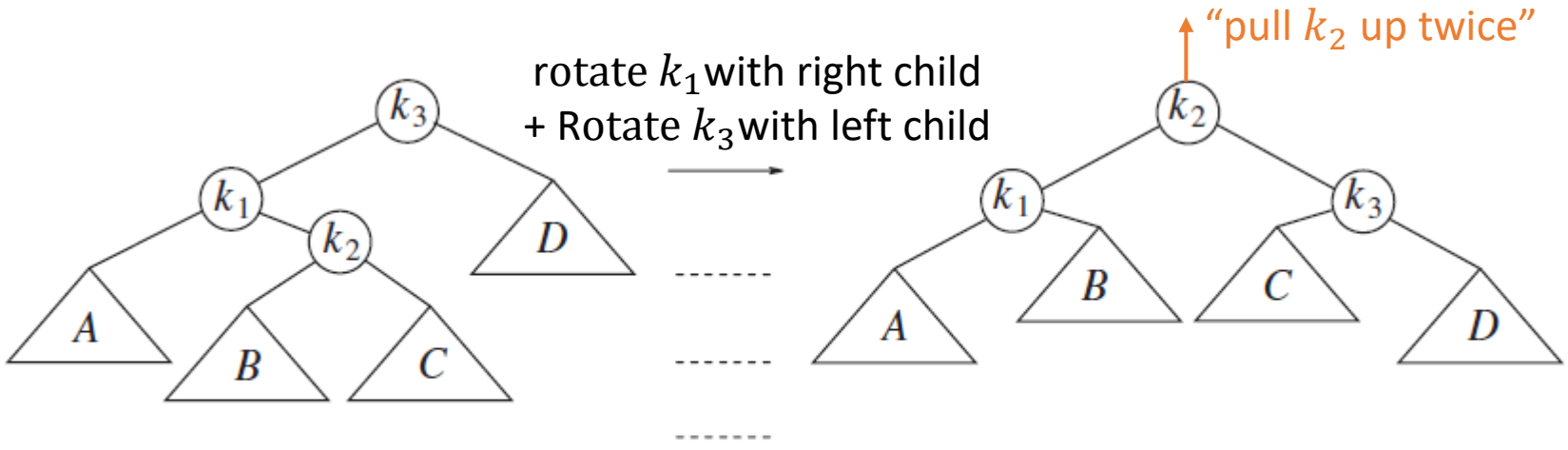



**Figure 4.38** Left–right double rotation to fix case 2

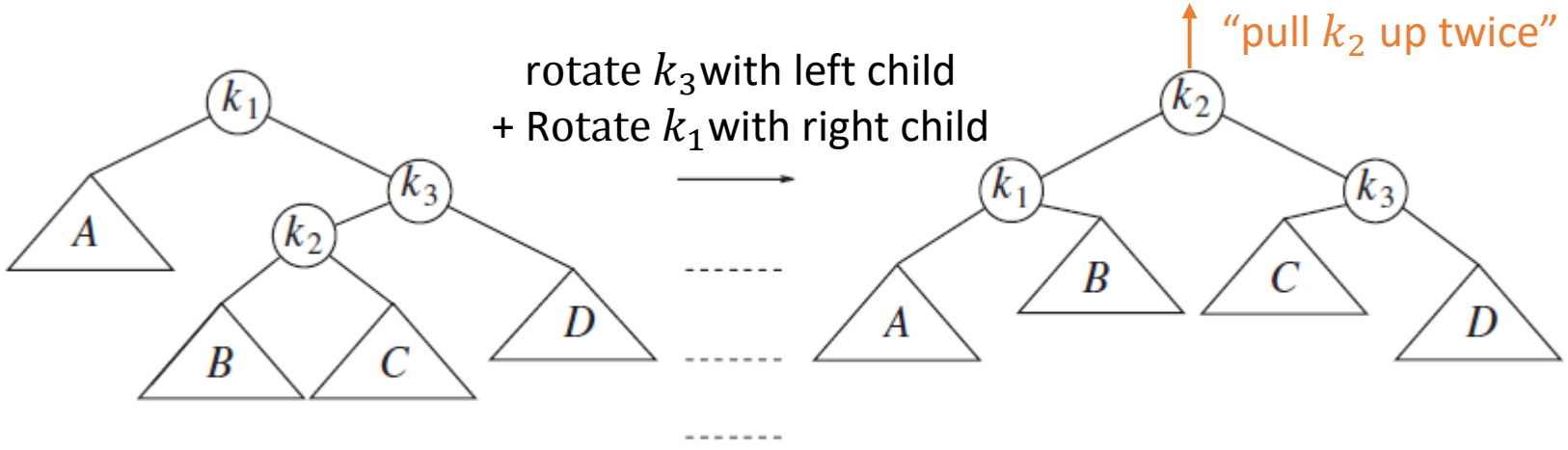



**Figure 4.39** Right–left double rotation to fix case 3

**Inside Insertion**

These cases are perfectly symmetric!

# Implementation

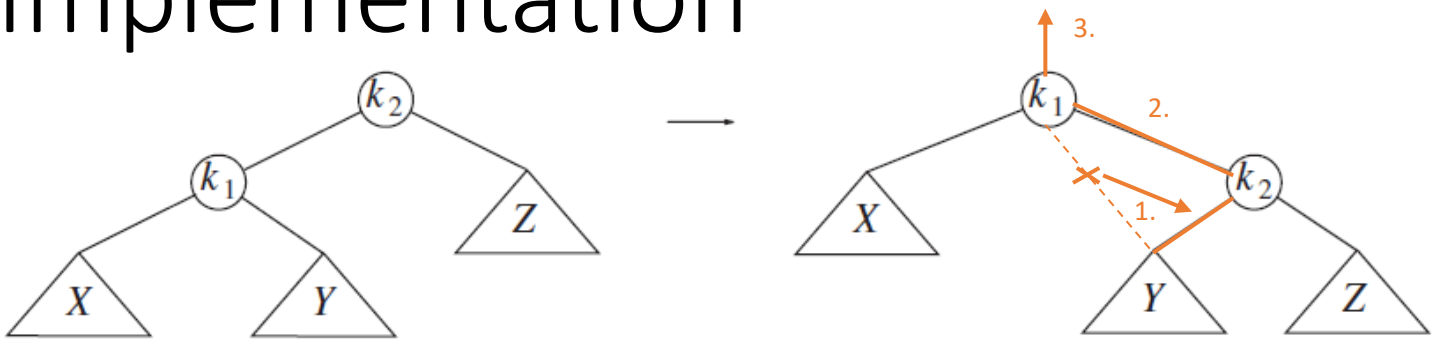



**Figure 4.43** Single rotation

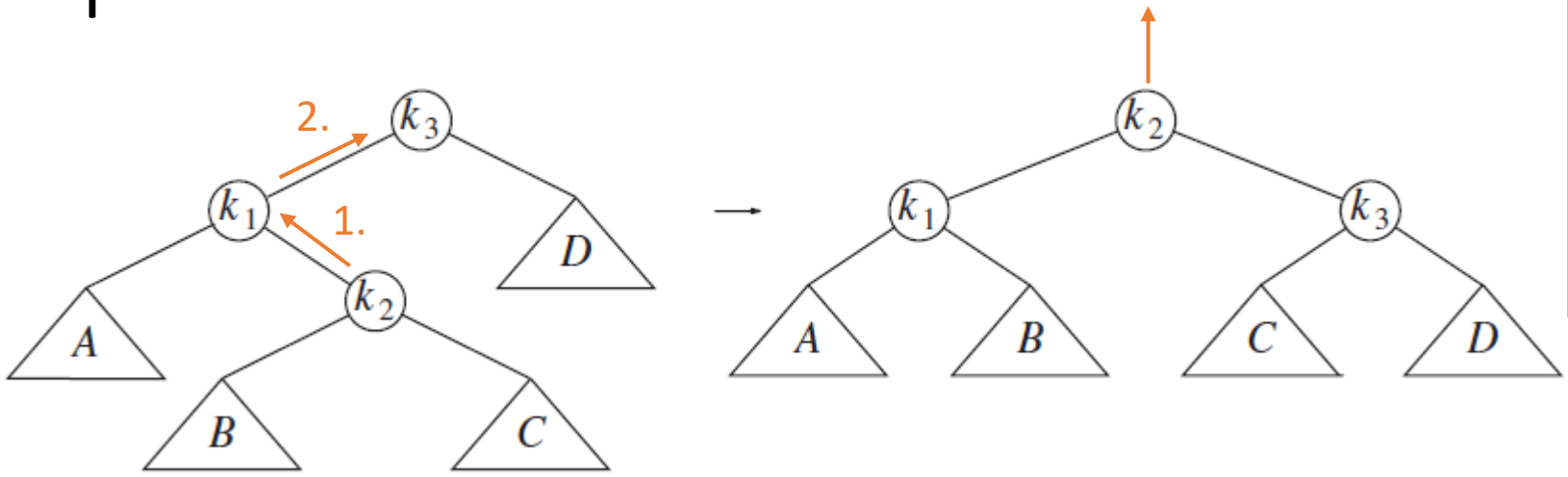
```
1   struct AvlNode
2   {
3       Comparable element;
4       AvlNode   *left;
5       AvlNode   *right;
6       int       height;
7   }
```

```
1   /**
2    * Rotate binary tree node with left child.
3    * For AVL trees, this is a single rotation for case 1.
4    * Update heights, then set new root.
5    */
6   void rotateWithLeftChild( AvlNode * & k2 )
7   {
8       AvlNode *k1 = k2->left;  // find k1
9       k2->left = k1->right;    // 1. move Y to k2
10      k1->right = k2;          // 2. add k2 to k1
11      k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12      k1->height = max( height( k1->left ), k2->height ) + 1;   // update height
13      k2 = k1;                 // 3. replace the subtree with k1 (note the reference passed on)
14  }
```

**Figure 4.44** Routine to perform single rotation

rotateWithRightChild() is similar with right and left switched.

# Implementation




```
1   struct AvlNode
2   {
3       Comparable element;
4       AvlNode   *left;
5       AvlNode   *right;
6       int       height;
7   }
```

```
1   /**
2    * Double rotate binary tree node: first left child
3    * with its right child; then node k3 with new left child.
4    * For AVL trees, this is a double rotation for case 2.
5    * Update heights, then set new root.
6    */
7   void doubleWithLeftChild( AvlNode * & k3 )
8   {
9       rotateWithRightChild( k3->left );     // 1. swap k1 (which is k3->left) and k2
10      rotateWithLeftChild( k3 );            // 2. swap k2 and k3
11  }
```

**Figure 4.46** Routine to perform double rotation

doubleWithRightChild() is similar with right and left switched.