AVL Trees

Michael Hahsler

With figures from Weiss:
Data Structures and Algorithms

Binary Search Trees

Definition: In a binary search tree, all items in each left subtree are smaller than the items in the right subtree.

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. Remember, $O(\log\ N)$ means that the problem size is halved with each step.

Problem: The assumption of $O(\log\ N)$ average running time is only true if no deletions are used! 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 AVL (Adelson-Velskii and Landis) tree is a 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.

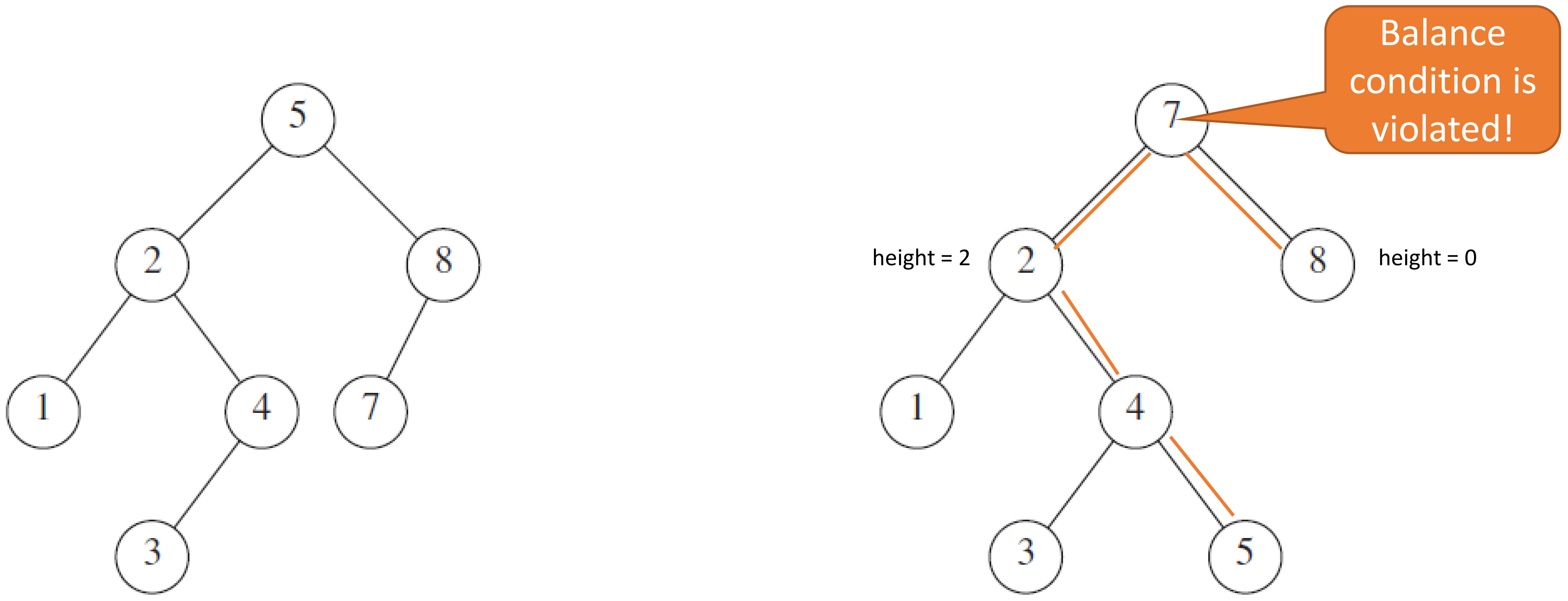


Figure 4.32 Two binary search trees. Only the left tree is AVL.

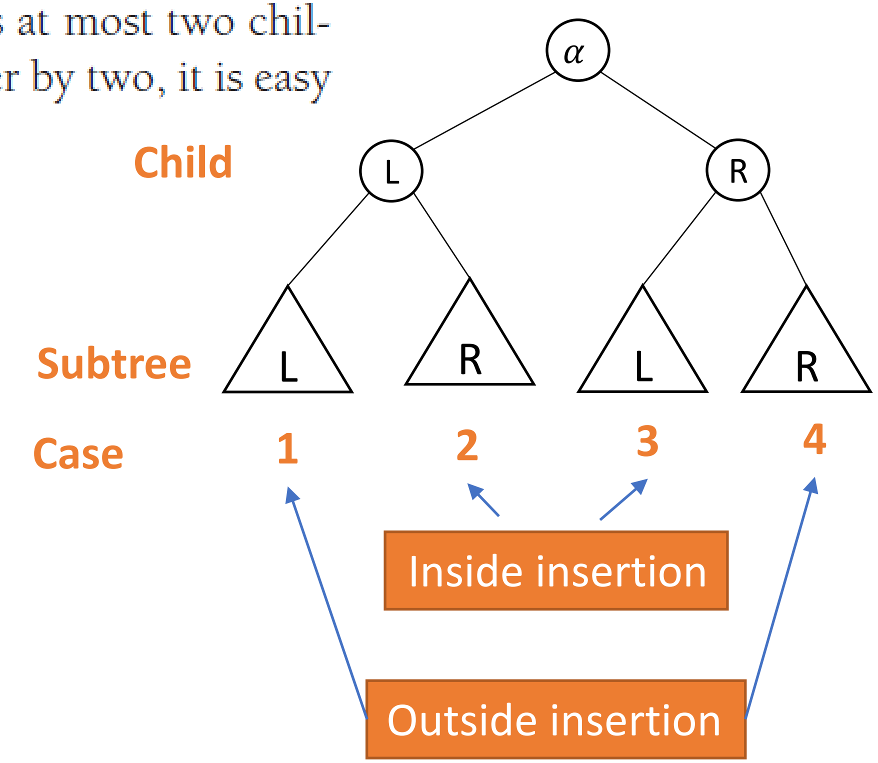
Rebalancing

Only necessary locally 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 α. Since any node has at most two children, and a height imbalance requires that α'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 α
2. An insertion into the right subtree of the left child of α
3. An insertion into the left subtree of the right child of α
4. An insertion into the right subtree of the right child of α



Single Rotation

Insertion in X

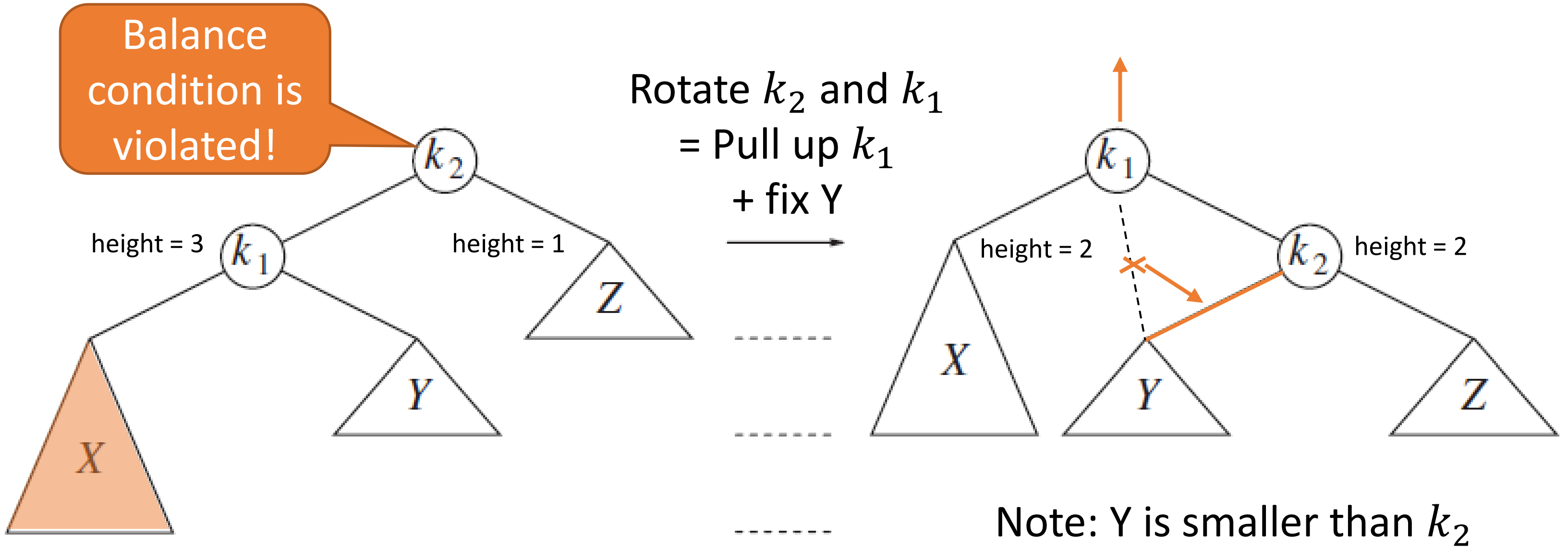


Figure 4.34 Single rotation to fix case 1: lifts up X by one level.

Insertion in Z

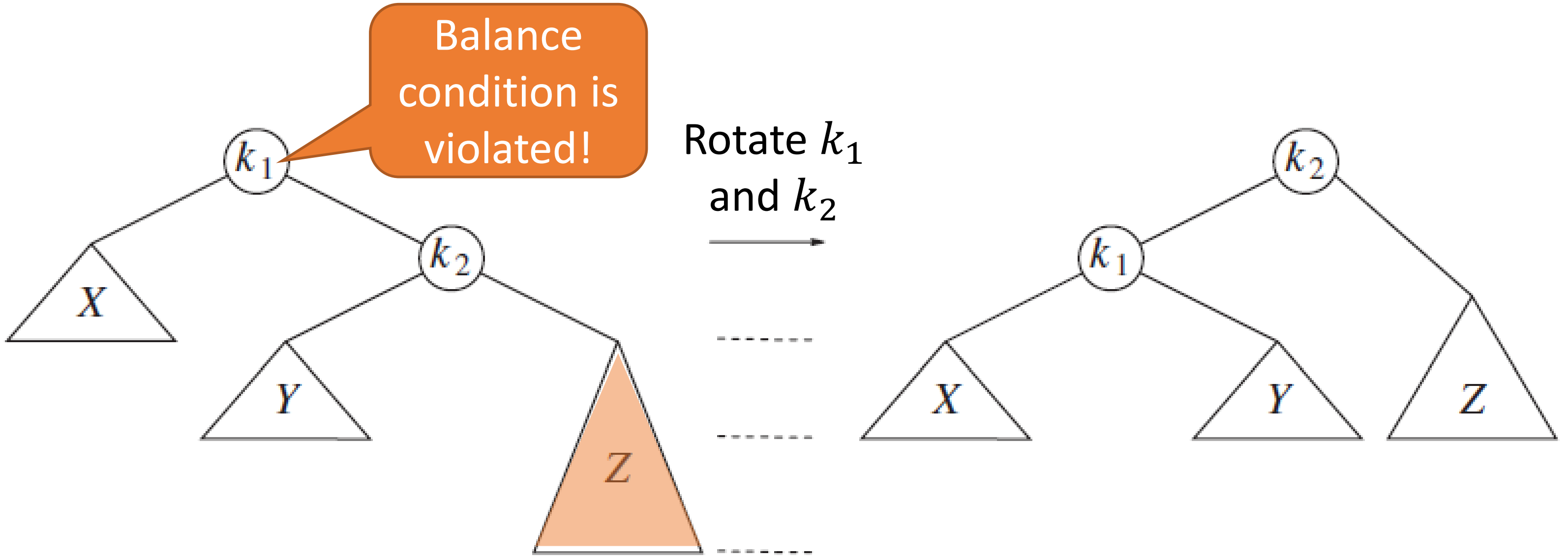


Figure 4.36 Single rotation fixes case 4 : lifts up Z by one level.

These cases a perfectly symmetric!

Example:

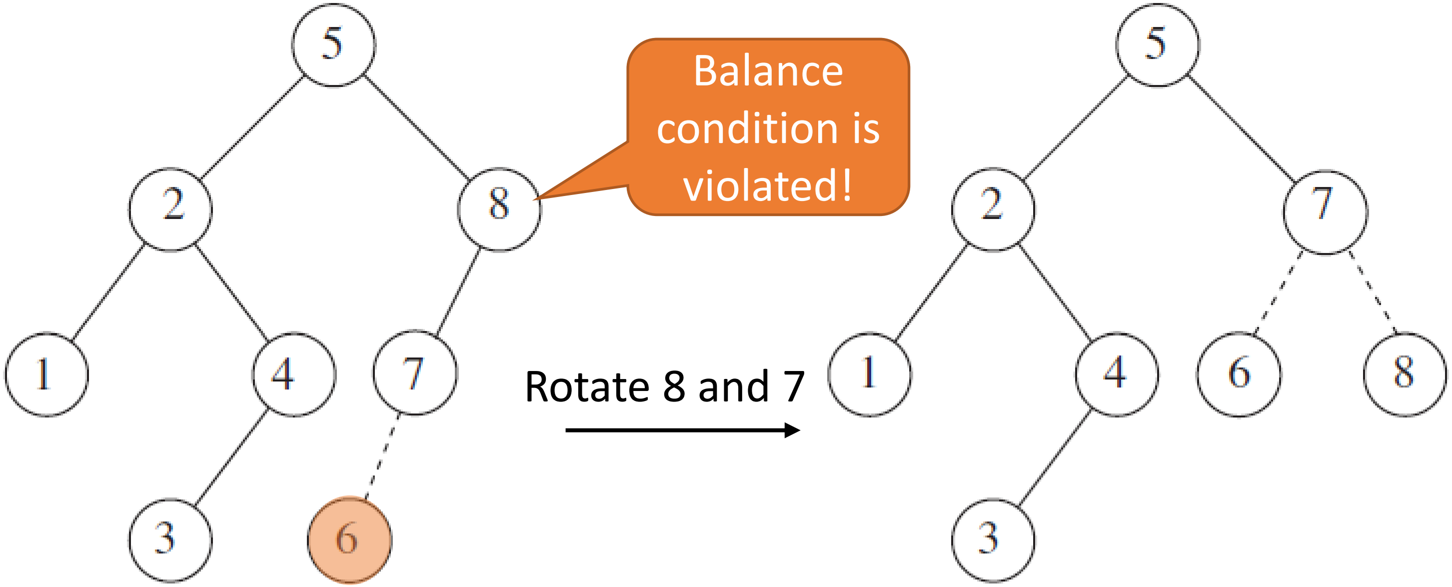


Figure 4.35 AVL property destroyed by insertion of 6, then fixed by a single rotation

Note: There is no Y to fix since 7 has only a single child.

Single Rotation Fails

Insertion in Y makes it too deep

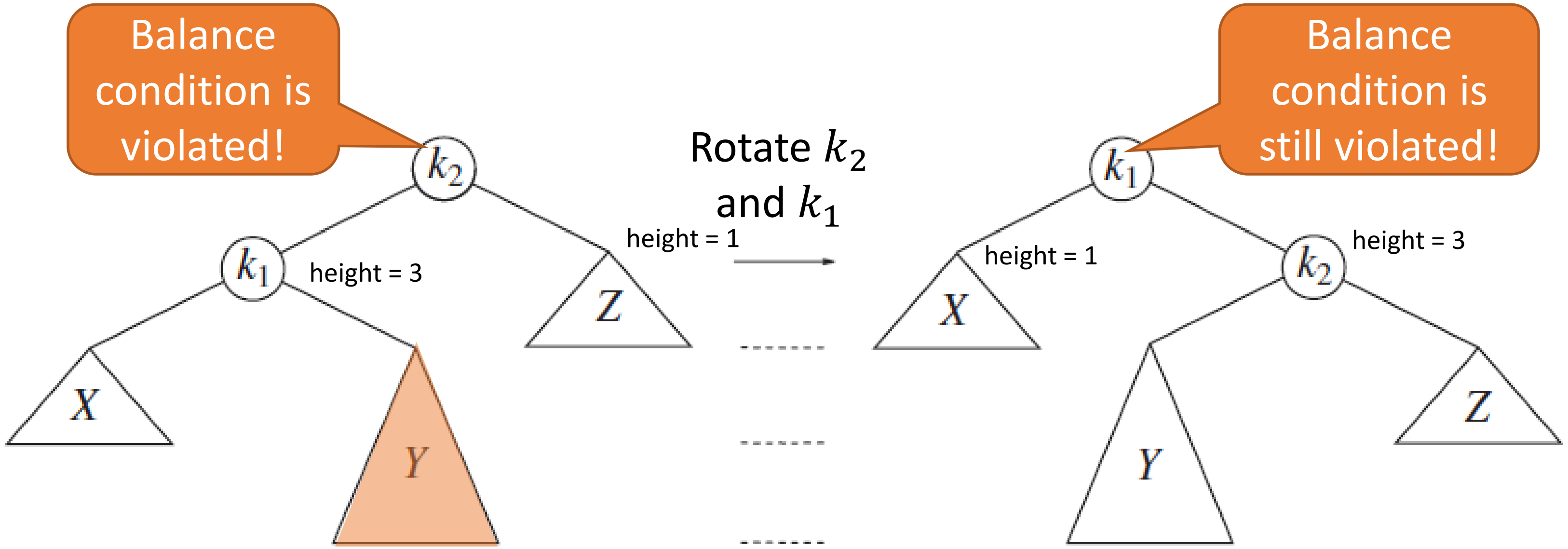
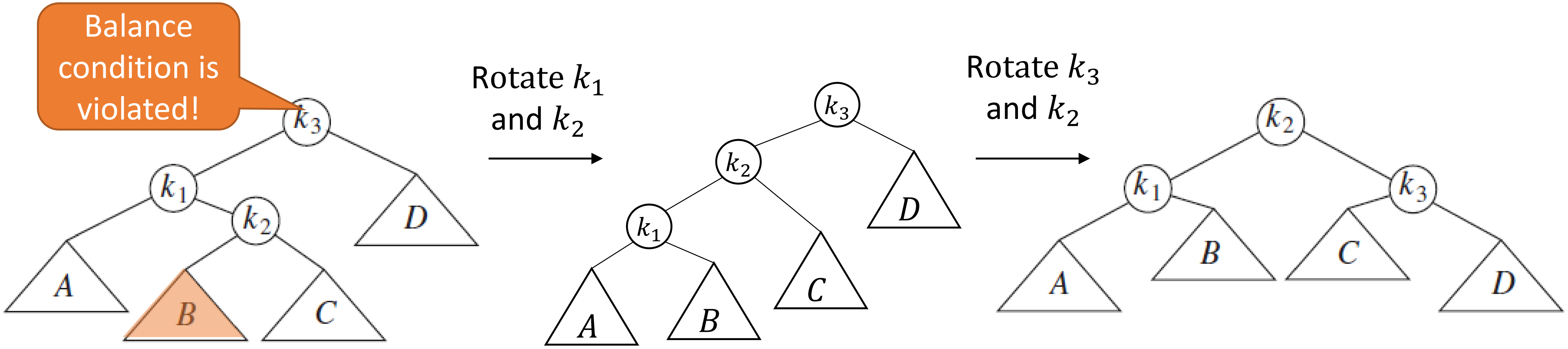


Figure 4.37 Single rotation fails to fix case 2 : does not lift up Y!



Left-right double rotation: lifts up B by one level.

Double Rotation

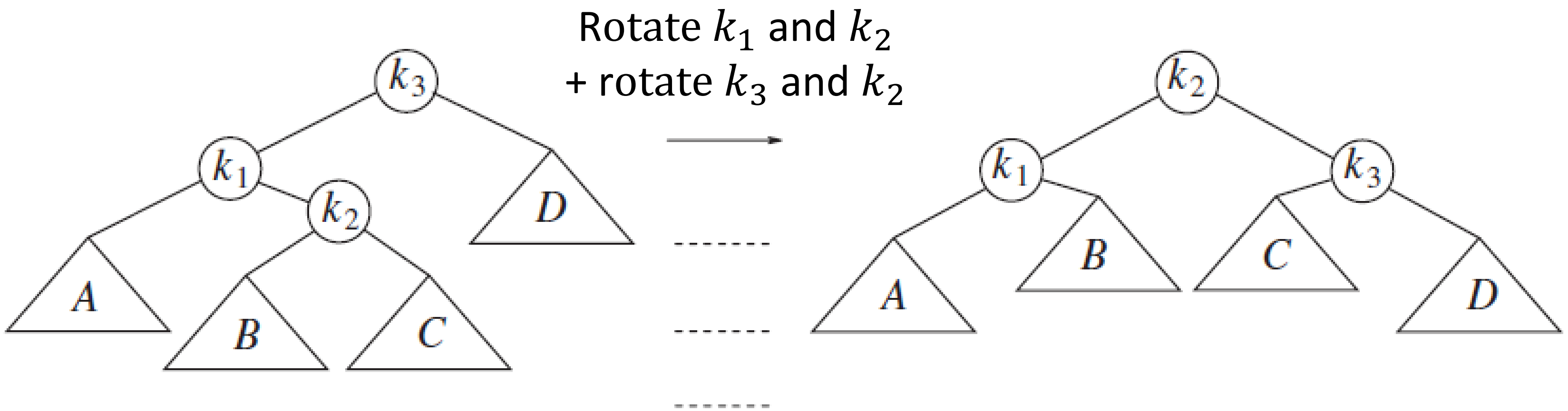


Figure 4.38 Left–right double rotation to fix case 2

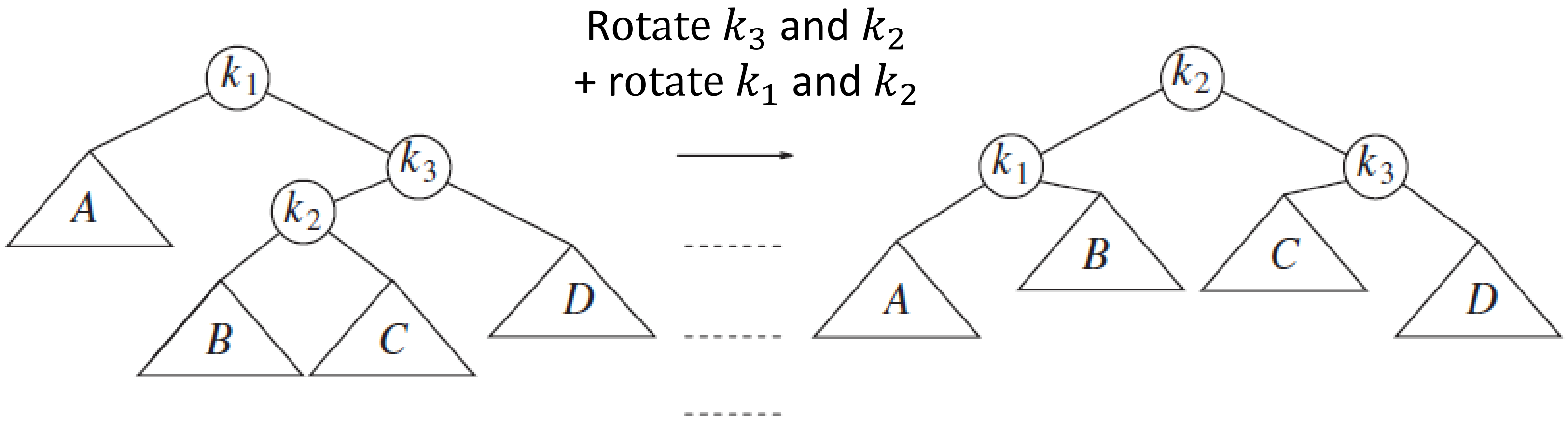


Figure 4.39 Right–left double rotation to fix case 3

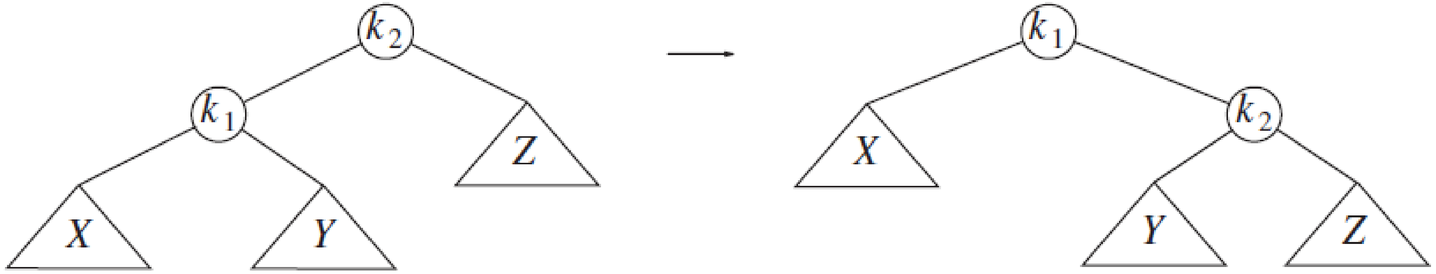


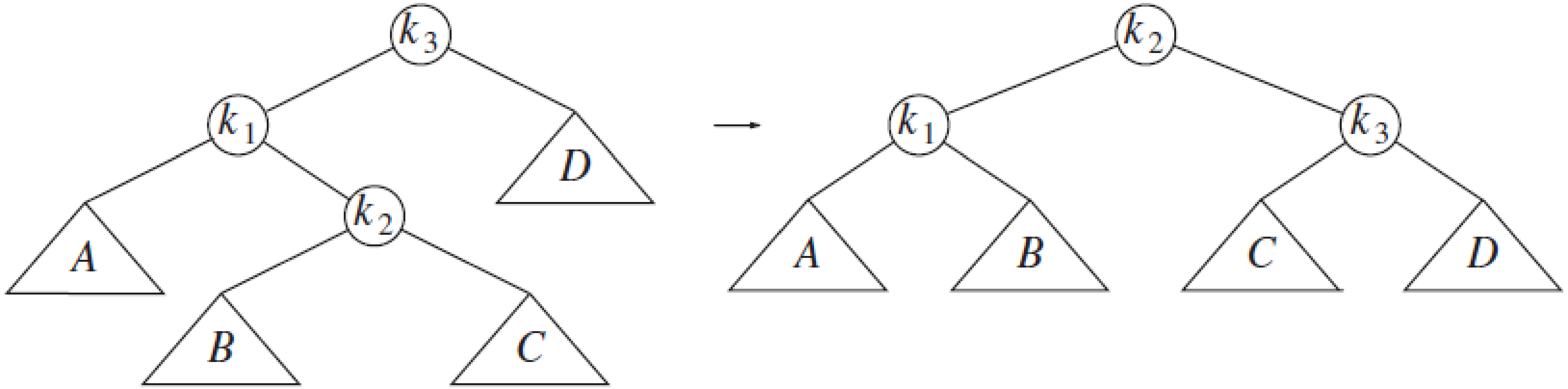
Figure 4.43 Single rotation

```
struct AvlNode
{
    Comparable element;
    AvlNode   *left;
    AvlNode   *right;
    int        height;
}
```

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
void rotateWithLeftChild( AvlNode * & k2 )
{
    AvlNode *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
    k1->height = max( height( k1->left ), k2->height ) + 1;
    k2 = k1;
}
```

Figure 4.44 Routine to perform single rotation

rotateWithRightChild() is similar with right and left switched.




```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
void doubleWithLeftChild( AvlNode * & k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

Figure 4.46 Routine to perform double rotation

doubleWithRightChild() is similar with right and left switched.