CS 300 Data Structures

Balanced Binary Search Tree – AVL Trees

Fall 2021

Bellevue College

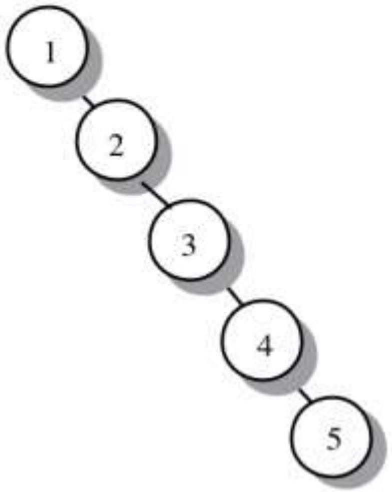
AVL Trees

- An AVL tree is a binary search tree with a *balance* condition.
- AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
- AVL tree *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a height close to the minimum.

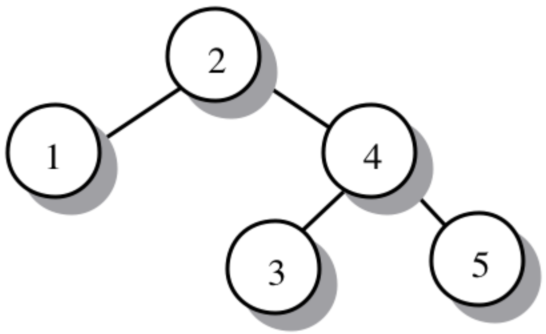
Definition:

An AVL tree is a binary search tree such that
for any node in the tree, the height of the left and
right subtrees can differ by at most 1.

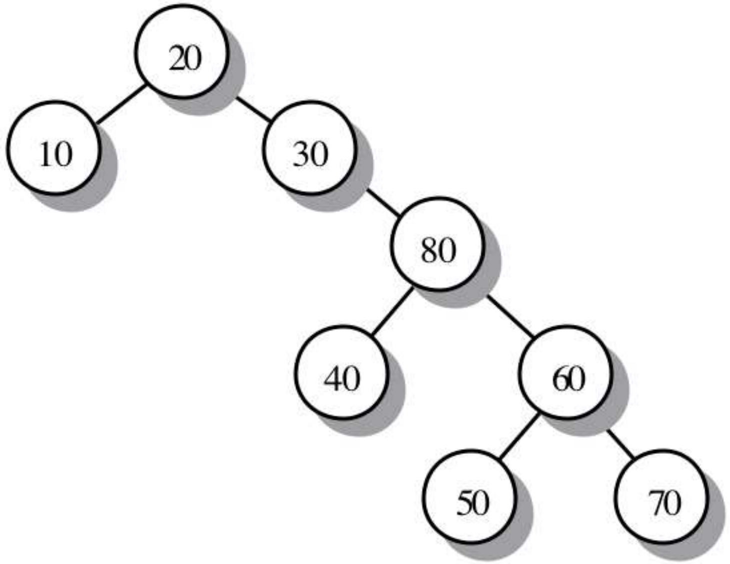
Binary Search Tree



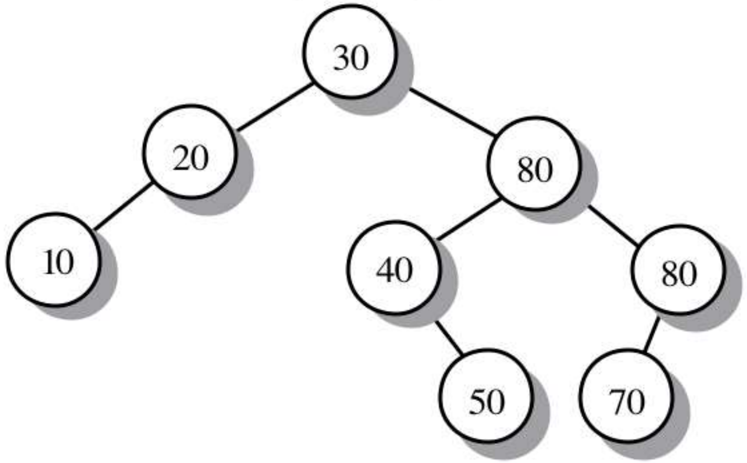
AVL Tree



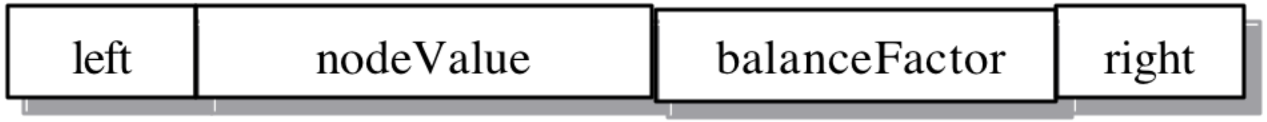
Binary Search Tree



AVL Tree



AVL Tree Node



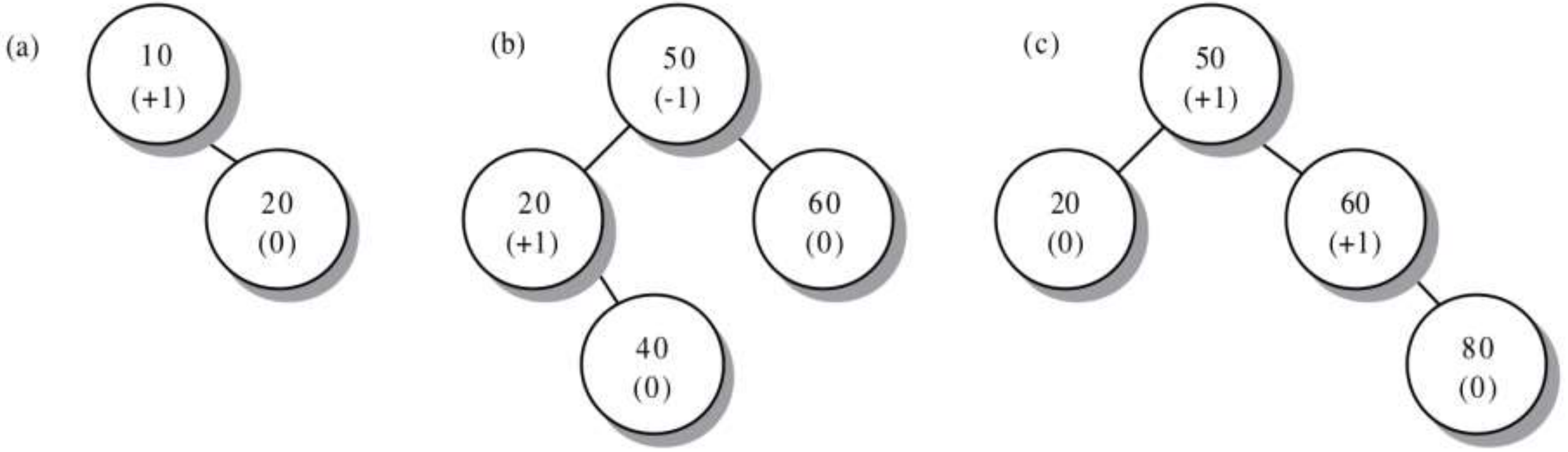
avlTreeNode

```
balanceFactor = height(right-subtree) - height(left-subtree)
```

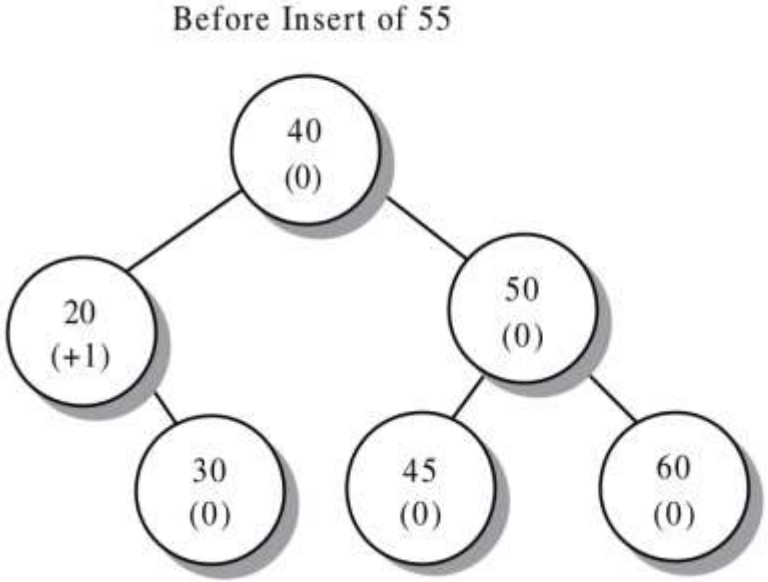
An AVL tree is a binary search tree in which the balance-factor of each node is in the range -1 to 1.

AVL Tree Node cont.

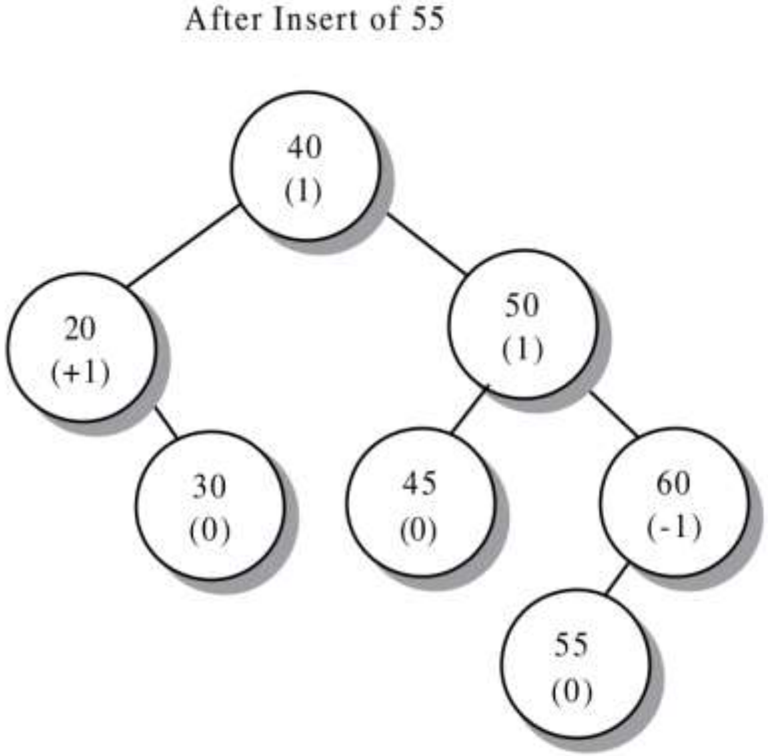
-1: height of the left subtree is one greater than the right subtree.
0: height of the left and right subtrees are equal.
+1: height of the right subtree is one greater than the left subtree.



Insert



Before Insert of 55
40
(0)
20
(+1)
50
(0)
30
(0)
45
(0)
60
(0)



After Insert of 55
40
(1)
20
(+1)
50
(1)
30
(0)
45
(0)
60
(-1)
55
(0)

Rebalancing

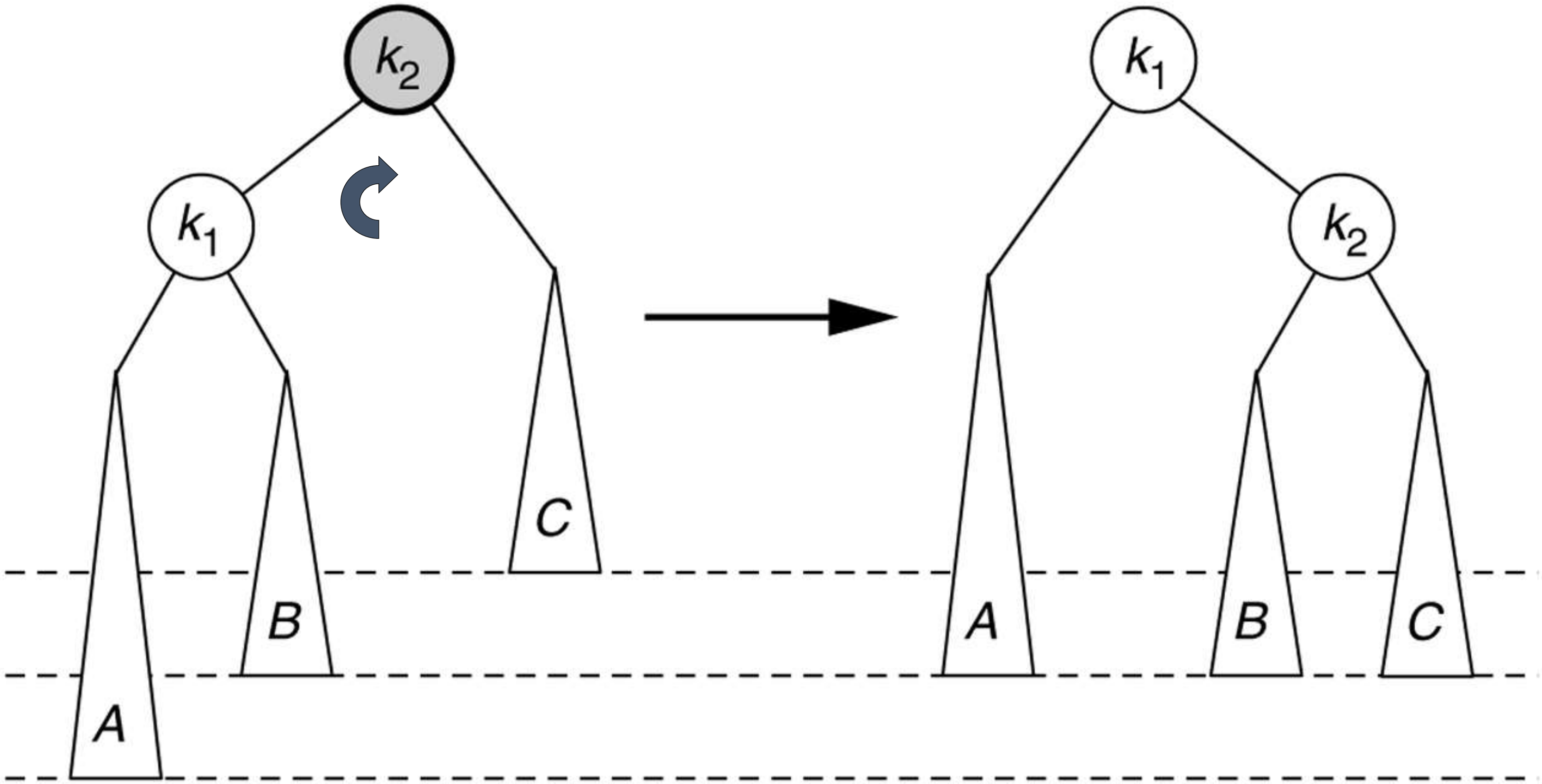
- Suppose the node to be rebalanced is X. There are 4 cases that we might have to fix (two are the mirror images of the other two):
  1. An insertion in the left subtree of the left child of X,
  2. An insertion in the right subtree of the left child of X,
  3. An insertion in the left subtree of the right child of X, or
  4. An insertion in the right subtree of the right child of X.
- Balance is restored by tree *rotations*.

Balancing Operations: Rotations

- Case 1 and case 4 are symmetric and requires the same operation for balance.
  - Cases 1,4 are handled by *single rotation.*
- Case 2 and case 3 are symmetric and requires the same operation for balance.
  - Cases 2,3 are handled by *double rotation.*

Figure 19.23

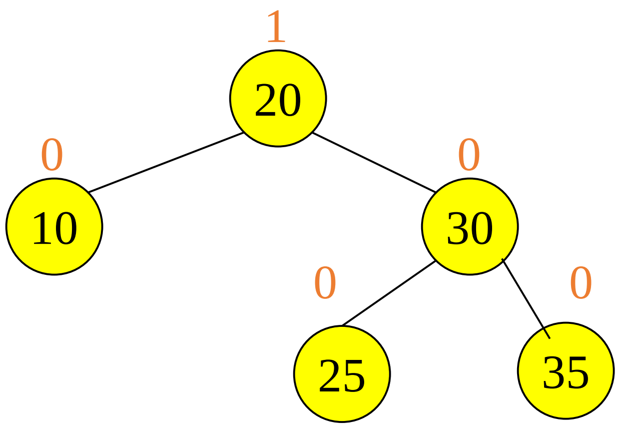
Single rotation to fix case 1: Rotate right



(a) Before rotation

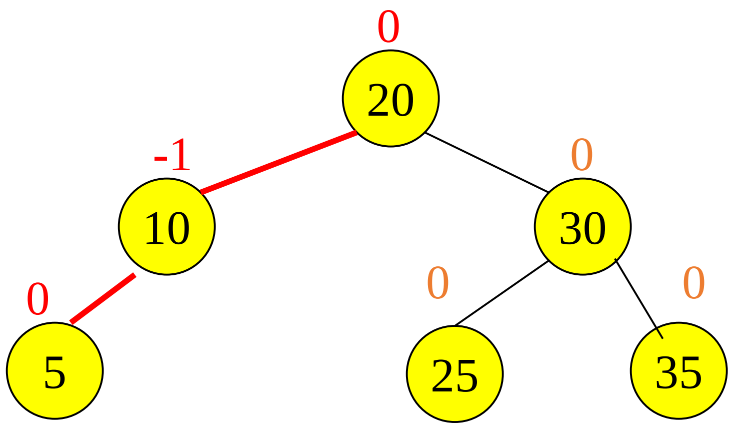
(b) After rotation

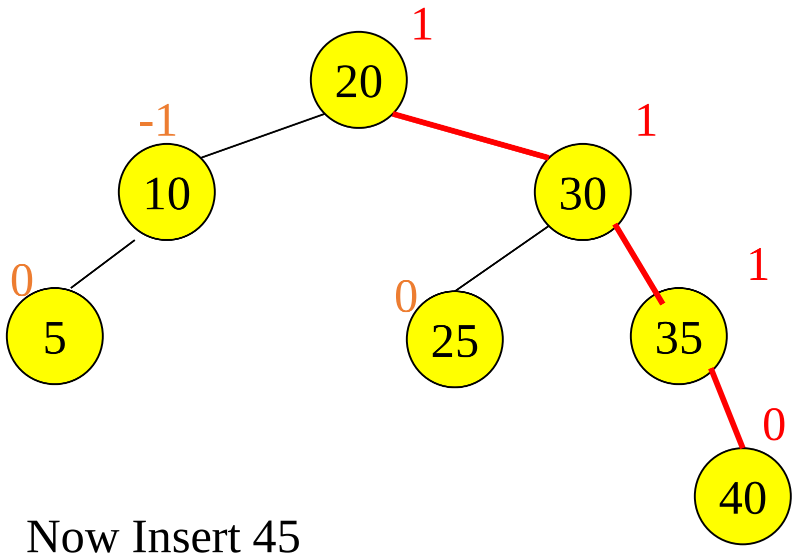
Single Rotation…



Insert 5, 40, 45

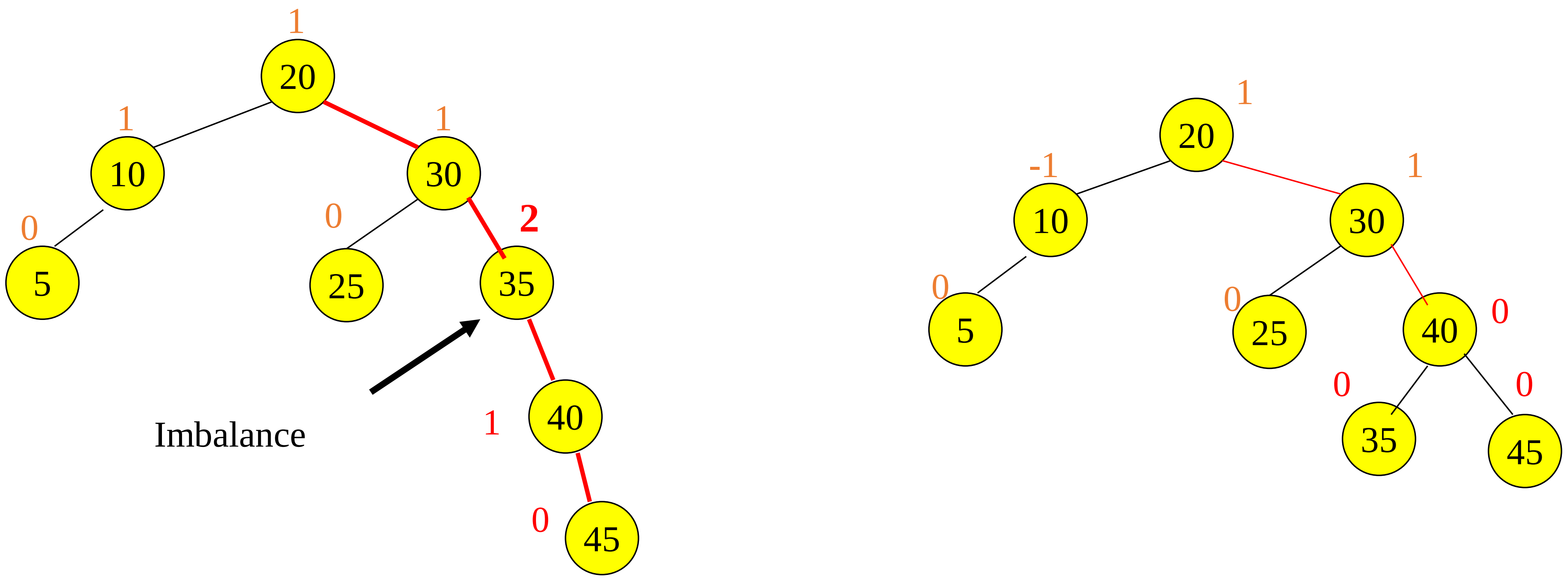
Single Rotation…





Now Insert 45

Single rotation (outside case)

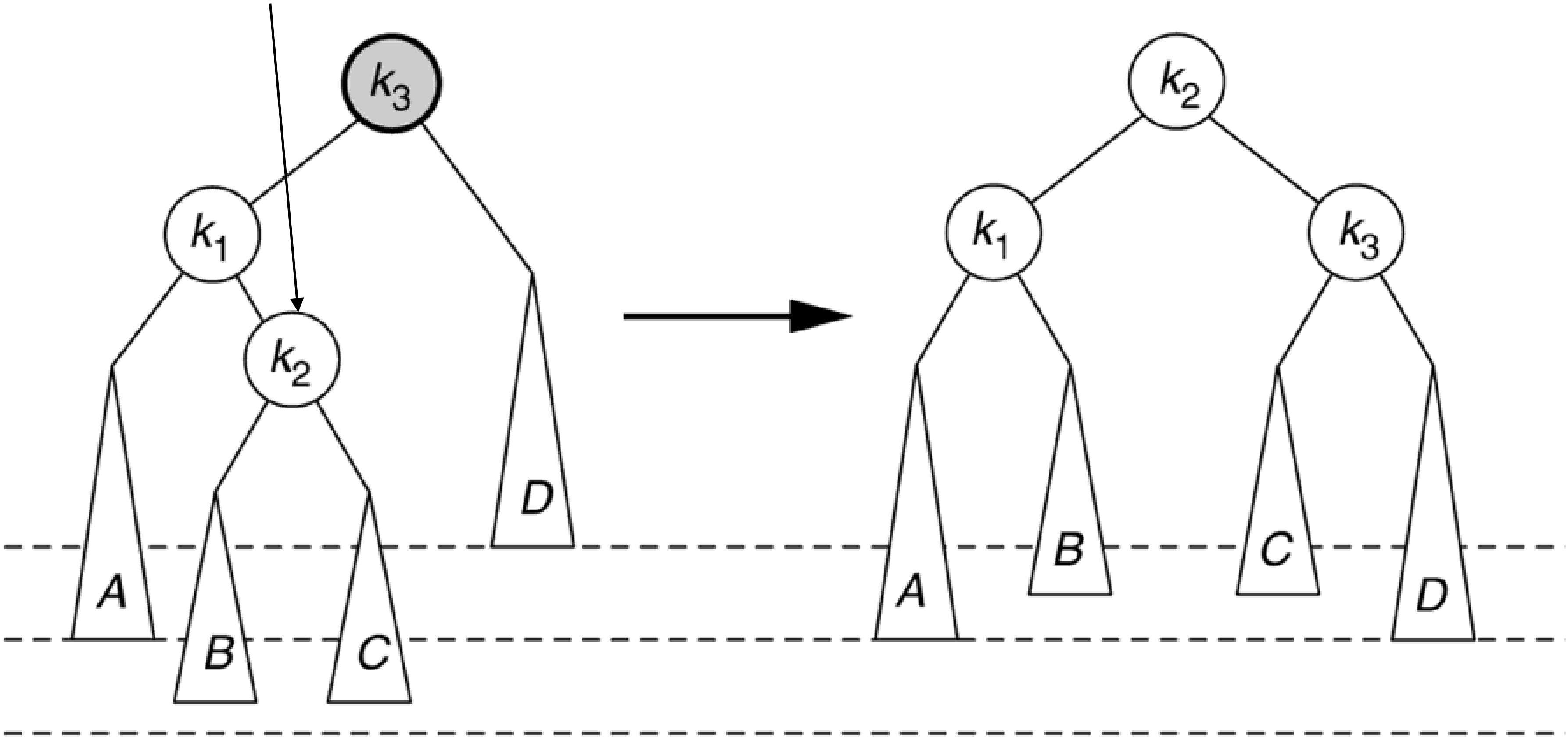


Double Rotation

- Single rotation does not fix the inside cases (2 and 3).
- These cases require a *double* rotation, involving three nodes and four subtrees.

Left–right double rotation to fix case 2

Lift this up:
first rotate left between (k_1,k_2),
then rotate right betwen (k_3,k_2)



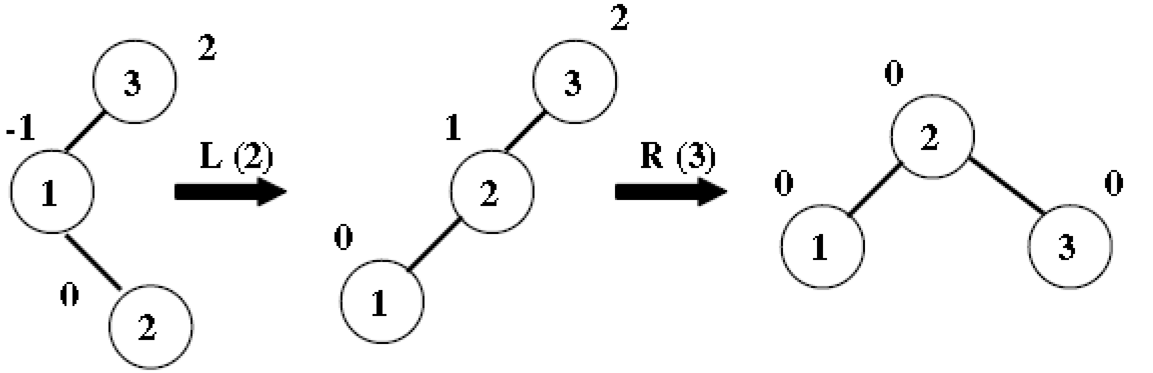
(a) Before rotation

(b) After rotation

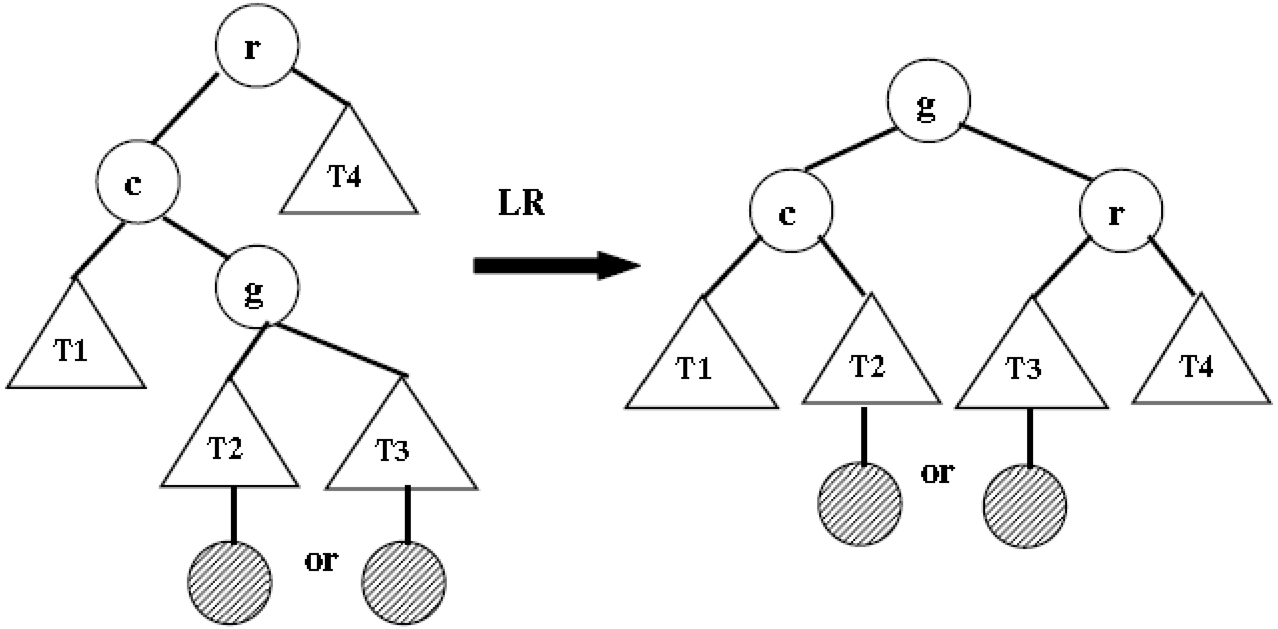
Left-Right Double Rotation

- A left-right double rotation is equivalent to a sequence of two single rotations:
  - 1st rotation on the original tree:
    a *left* rotation between X's left-child and grandchild
  - 2nd rotation on the new tree:
    a *right* rotation between X and its new left child.

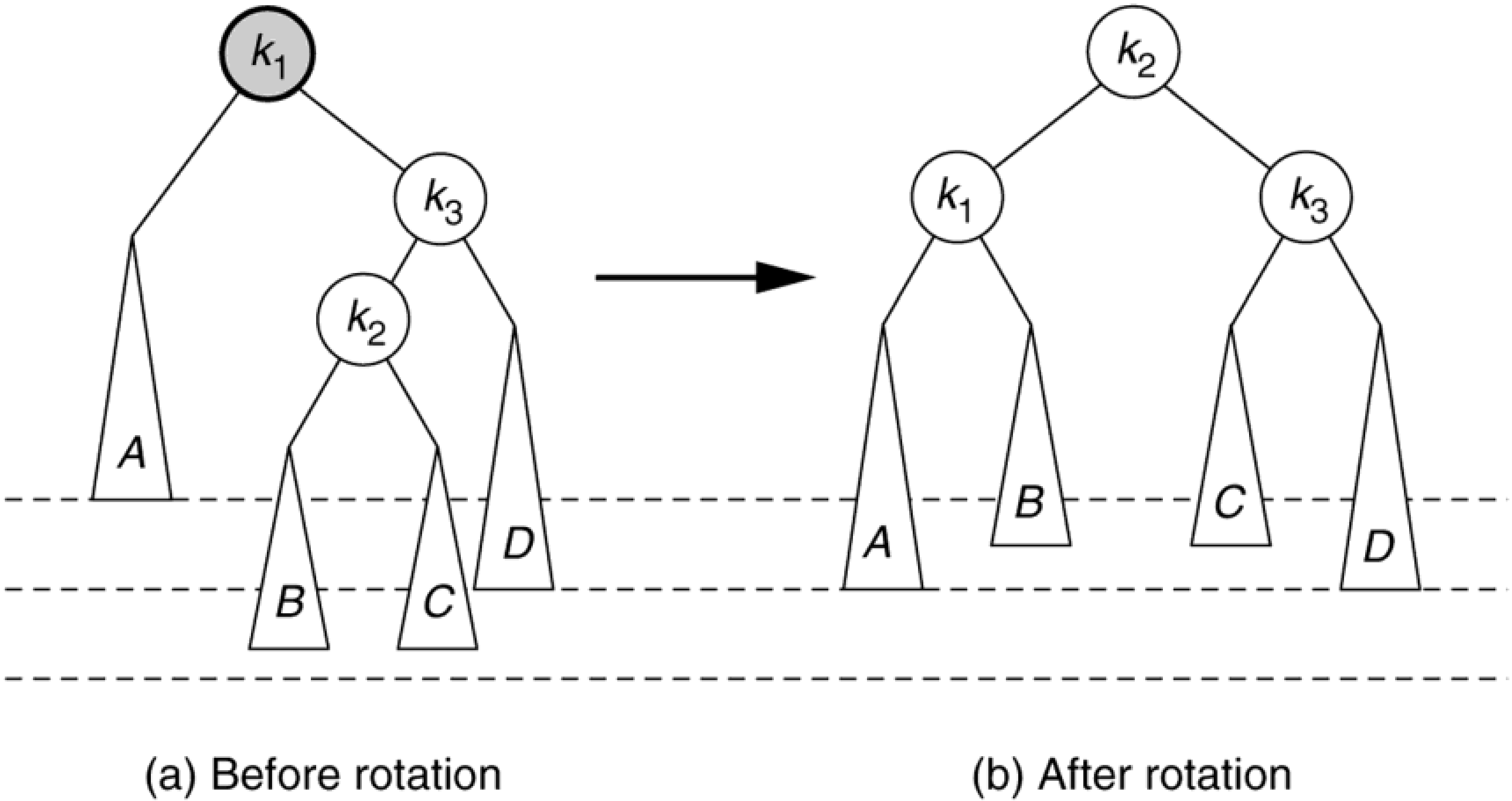
Example



- General form: A shaded node is the last node inserted. It can be either in the left sub-tree or in the right sub-tree of the root's grandchild.



Right–Left double rotation to fix case 3.



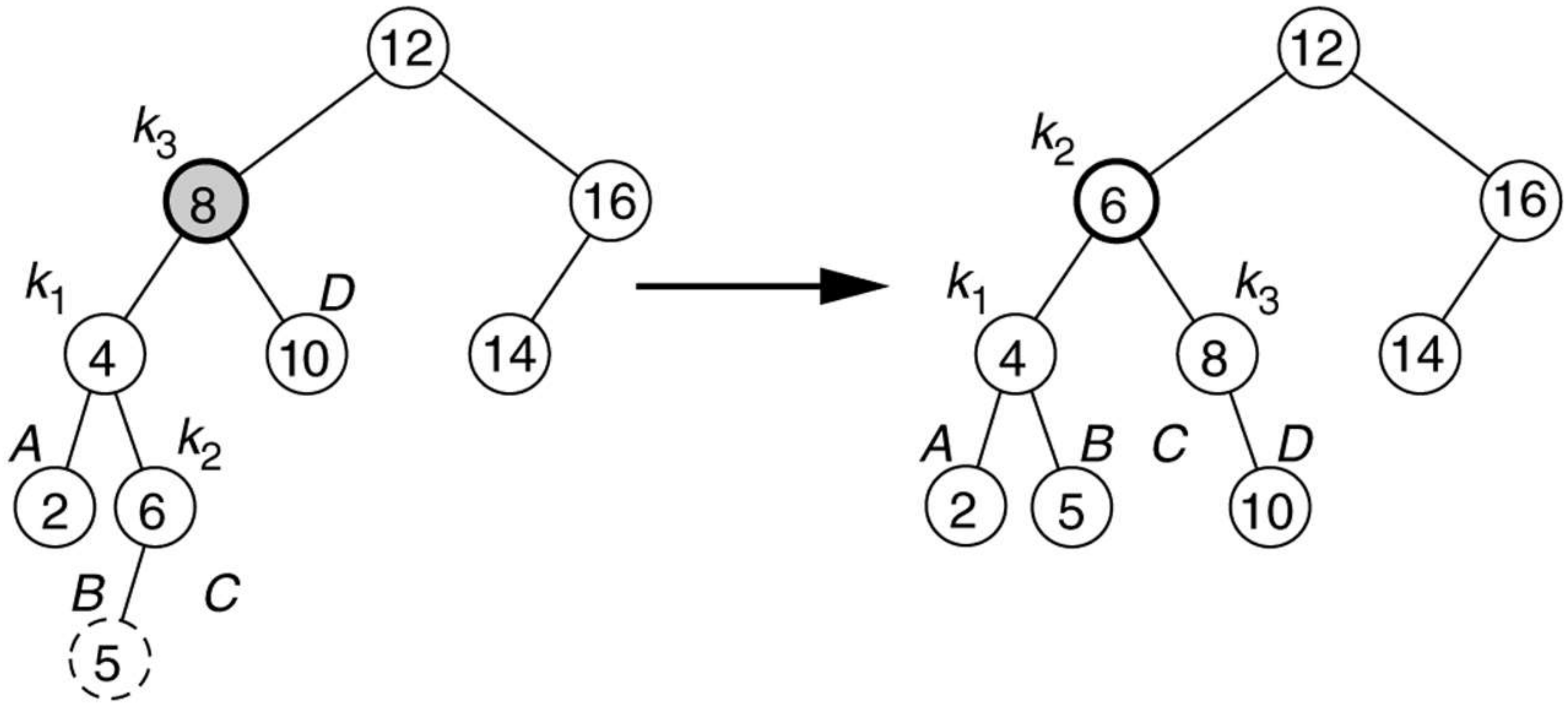
Example-1

6 6

8 7 7

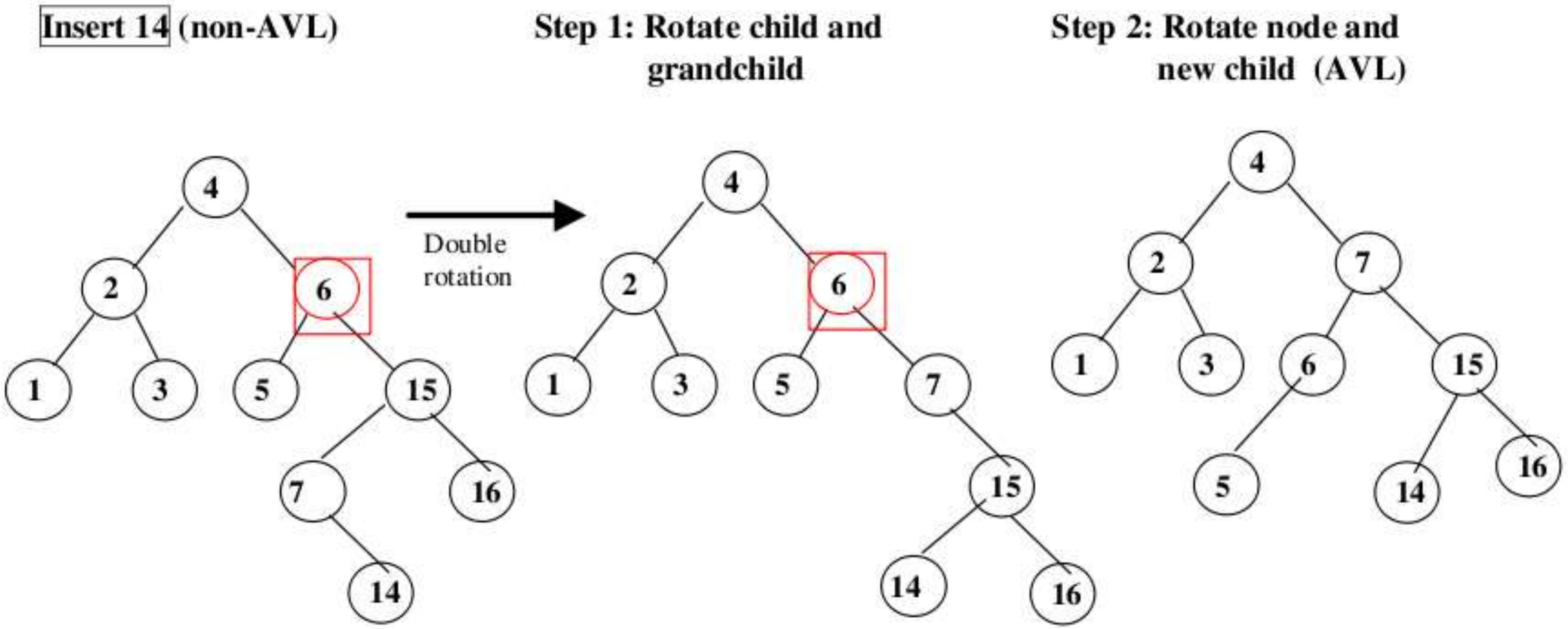
7 8 6 8

Figure 19.30
Double rotation fixes AVL tree after the insertion of 5.

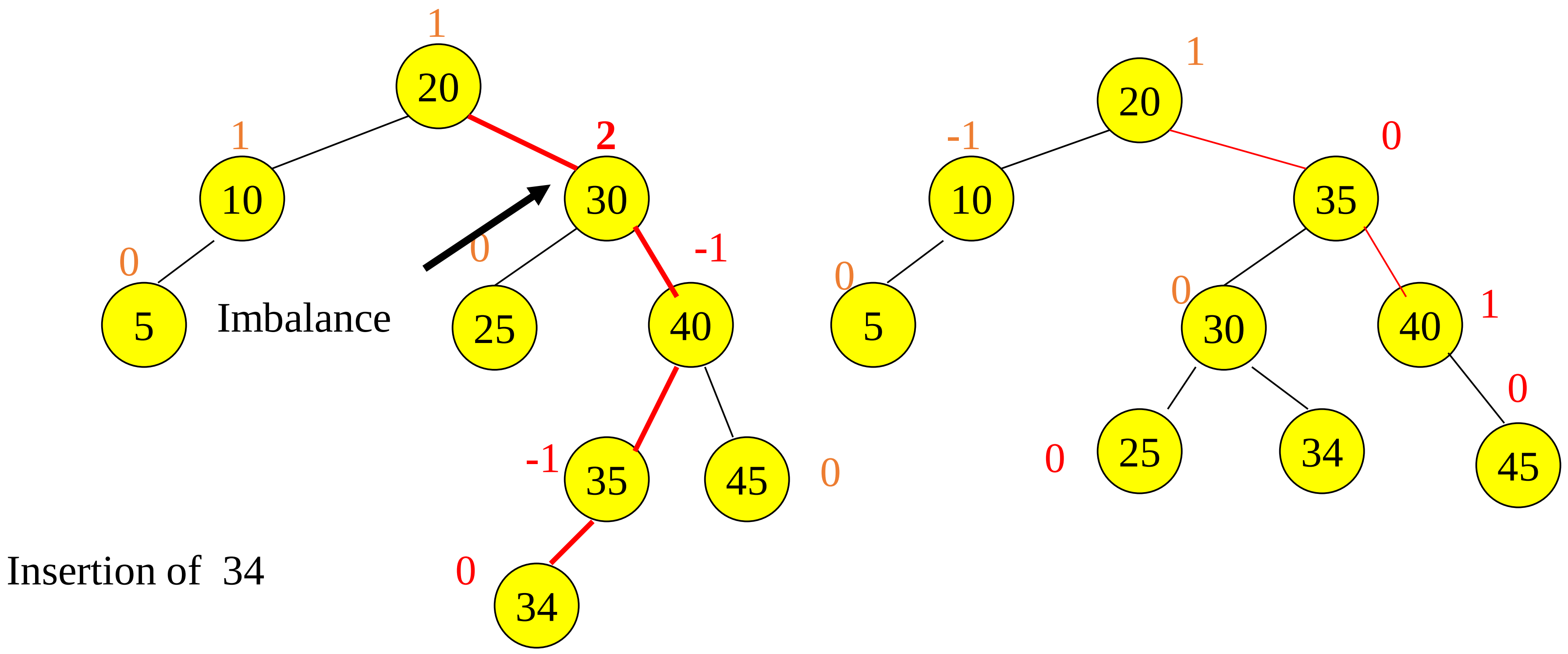


(a) Before rotation (b) After rotation

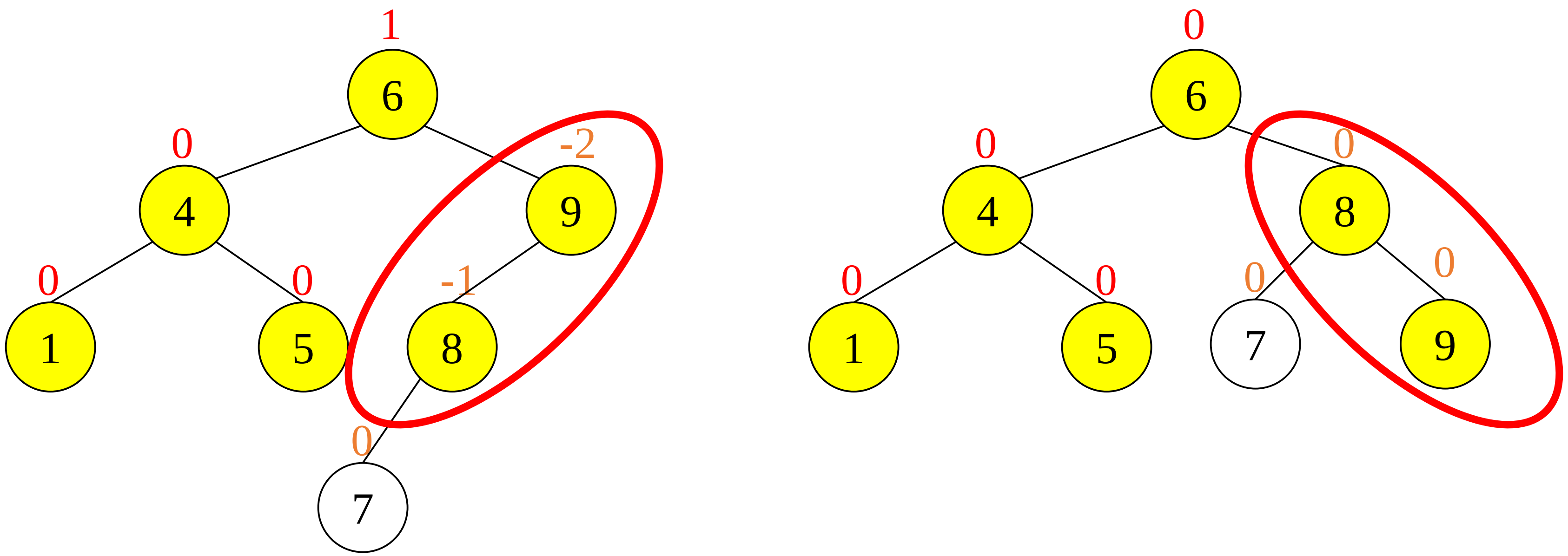
Example-2

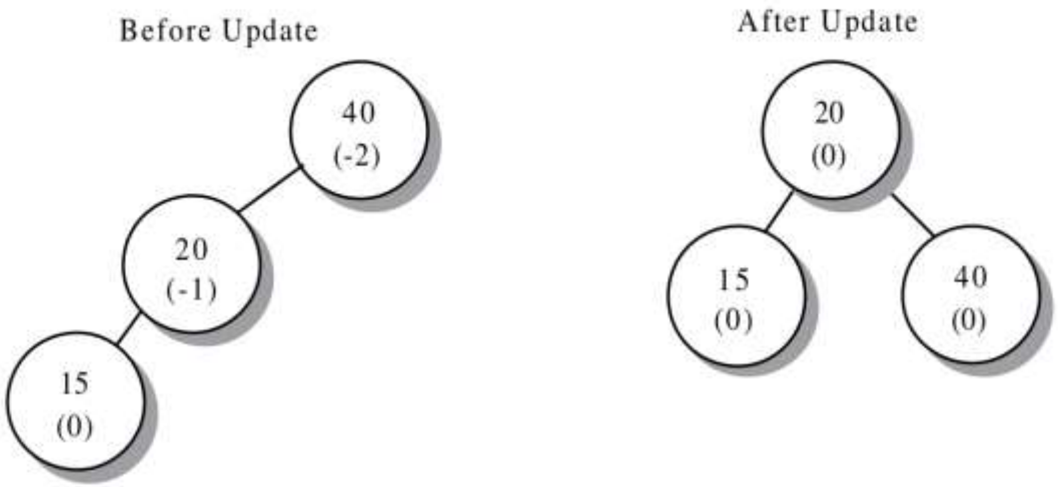

Insert 14 (non-AVL)
Double
rotation
Step 1: Rotate child and
grandchild
Step 2: Rotate node and
new child (AVL)
4
2
6
1
3
5
15
7
16
14
4
2
6
1
3
5
7
15
14
16
4
2
7
1
3
6
15
5
14
16

Double rotation (inside case)

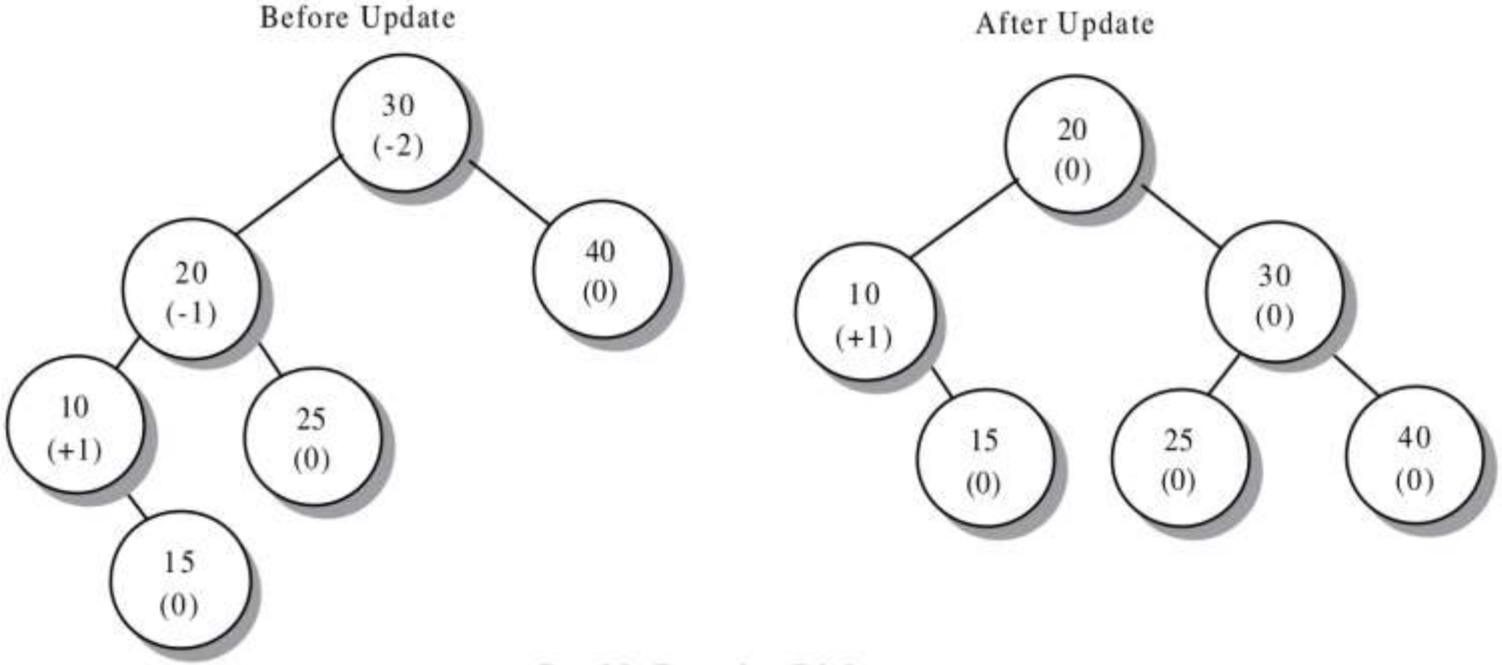


inside or outside?





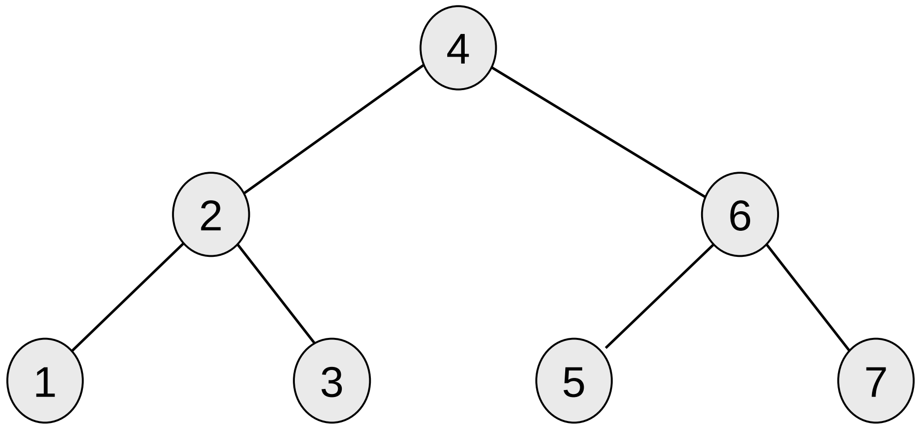
Single Rotation Right



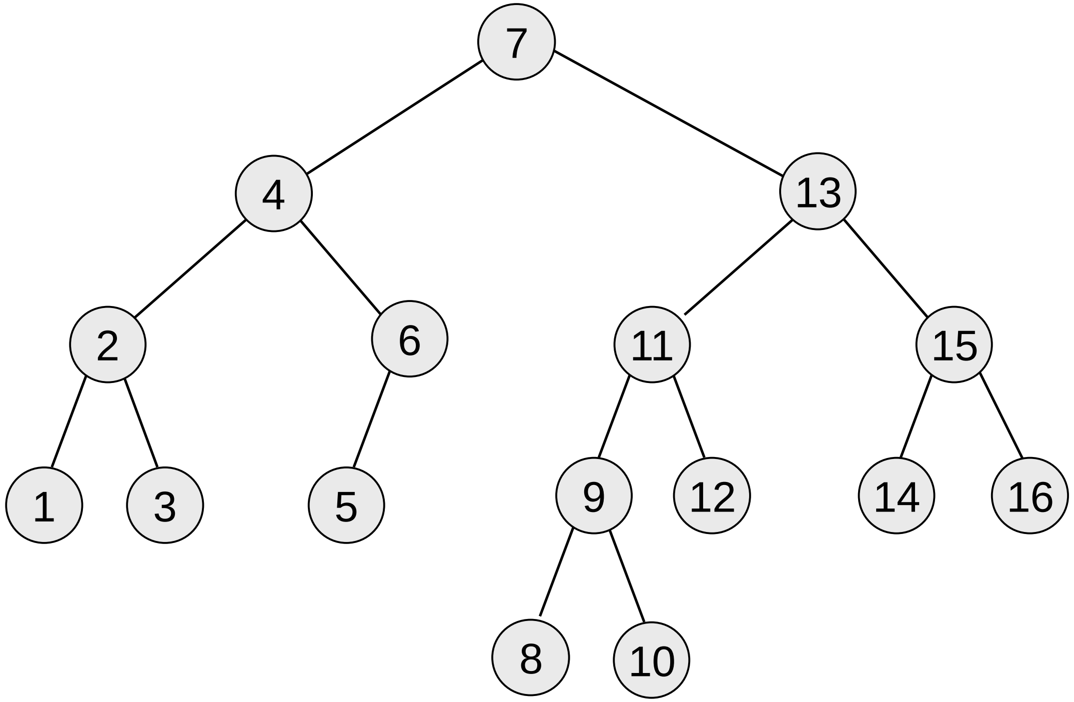
Double Rotation Right

Example-3

- Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the following tree.



Answer



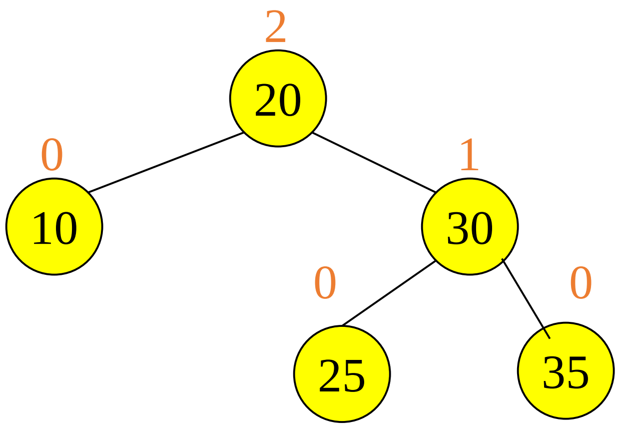
Node declaration for AVL trees

```
template <class T>
struct AvlNode
{
   T data;
   AvlNode   *left;
   AvlNode   *right;
   int       balanceFactor;
};
```

Single right rotation

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

Single Rotation…



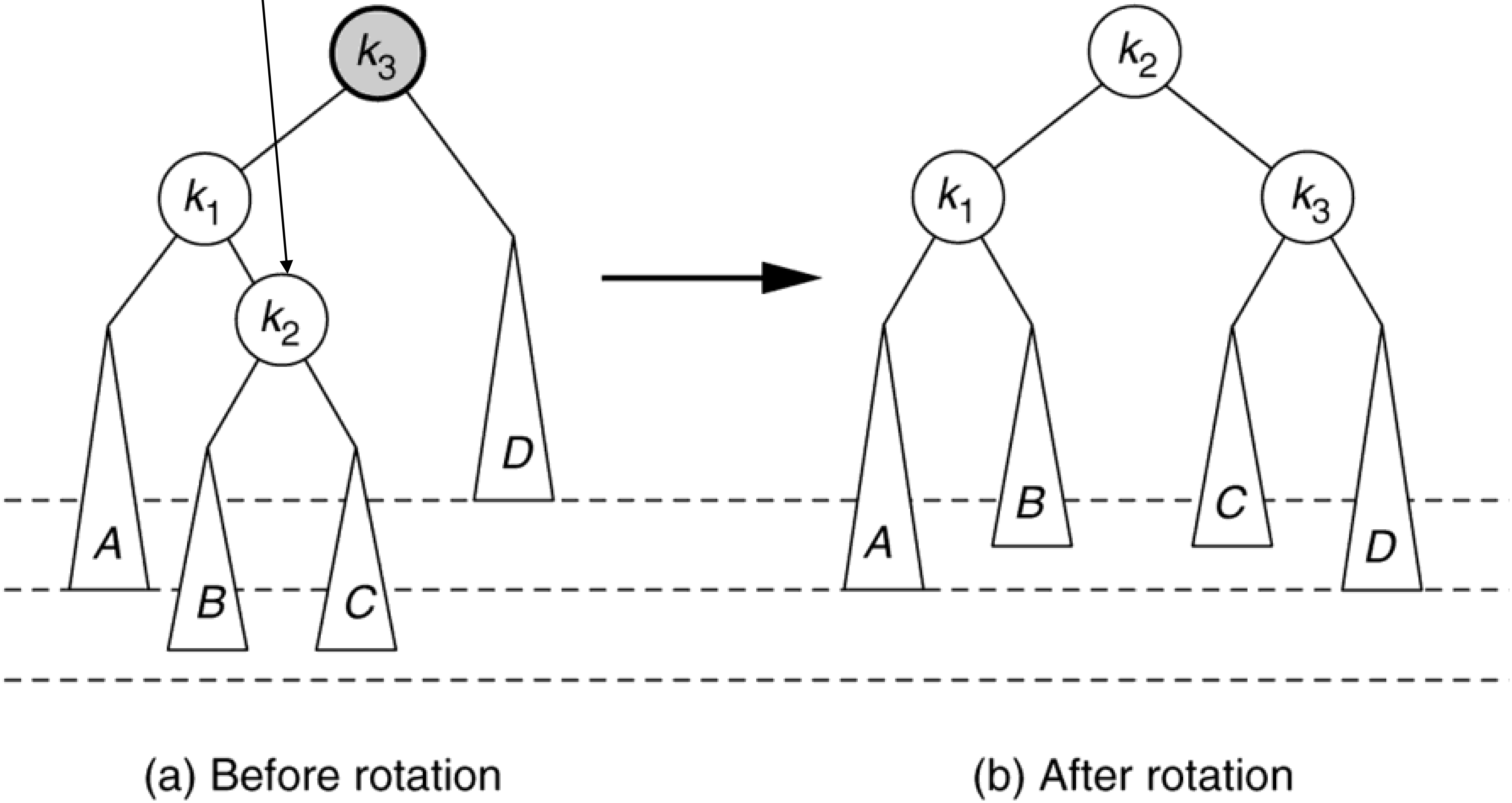
Insert 5, 40, 45

Double Rotation

- Single rotation does not fix the inside cases (2 and 3).
- These cases require a *double* rotation, involving three nodes and four subtrees.

Left–right double rotation to fix case 2

Lift this up:
first rotate left between (k_1,k_2),
then rotate right betwen (k_3,k_2)



Double Rotation

```
/**
 * 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.
 */
template <class T>
void doubleWithLeftChild( AvlNode<T> *& k3 )
{
   rotateWithRightChild( k3->left );
   rotateWithLeftChild( k3 );
}
```

```
/* private function to insert into a subtree.
 * x is the item to insert; t is the node that roots the tree.
 */
template <class T>
void insert(AvlNode<T> *& t, T& item )
{
   if( t == NULL )
     t = new AvlNode<T>(item);
   else if( item < t->data)
   {
     insert( item, t->left );
     if( balanceFactor == 2 )
       if( item < t->left->data)
           rotateWithLeftChild( t );  // case 1
       else
           doubleWithLeftChild( t );  // case 2
   }
   else if( item > t->data)
   {
       insert( item, t->right );
       if( balanceFactor == 2 )
          if( item> t->right->data)
             rotateWithRightChild( t );  // case 4
          else
             doubleWithRightChild( t );  // case 3
    }
    else
       ;  // Duplicate; do nothing
    t->balanceFactor = height(t->right)-height(t->right ));
}
```