AVL Trees

AVL Trees

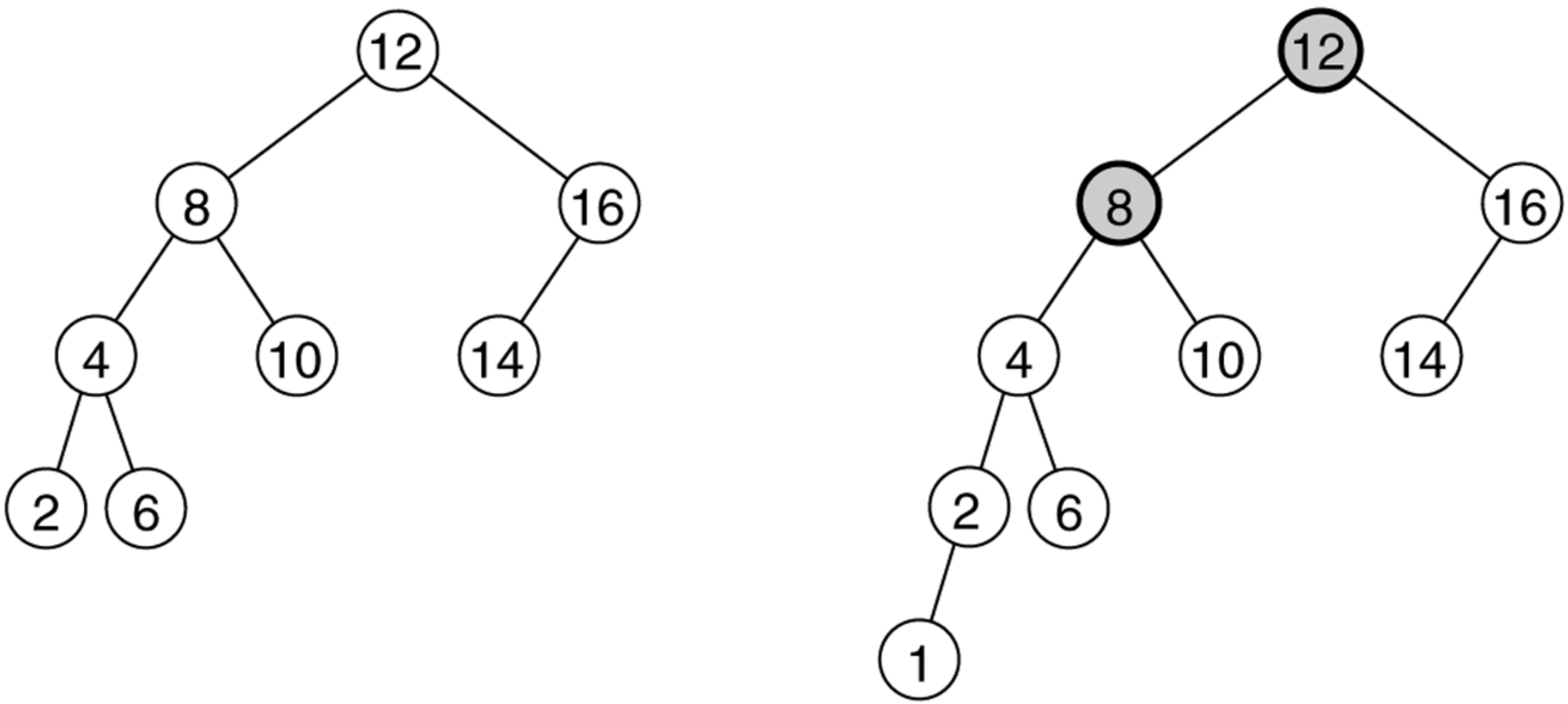
- An AVL tree is a binary search tree with a *balance* condition.
- AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
- AVL tree *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a height close to the minimum.

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.

Figure 19.21

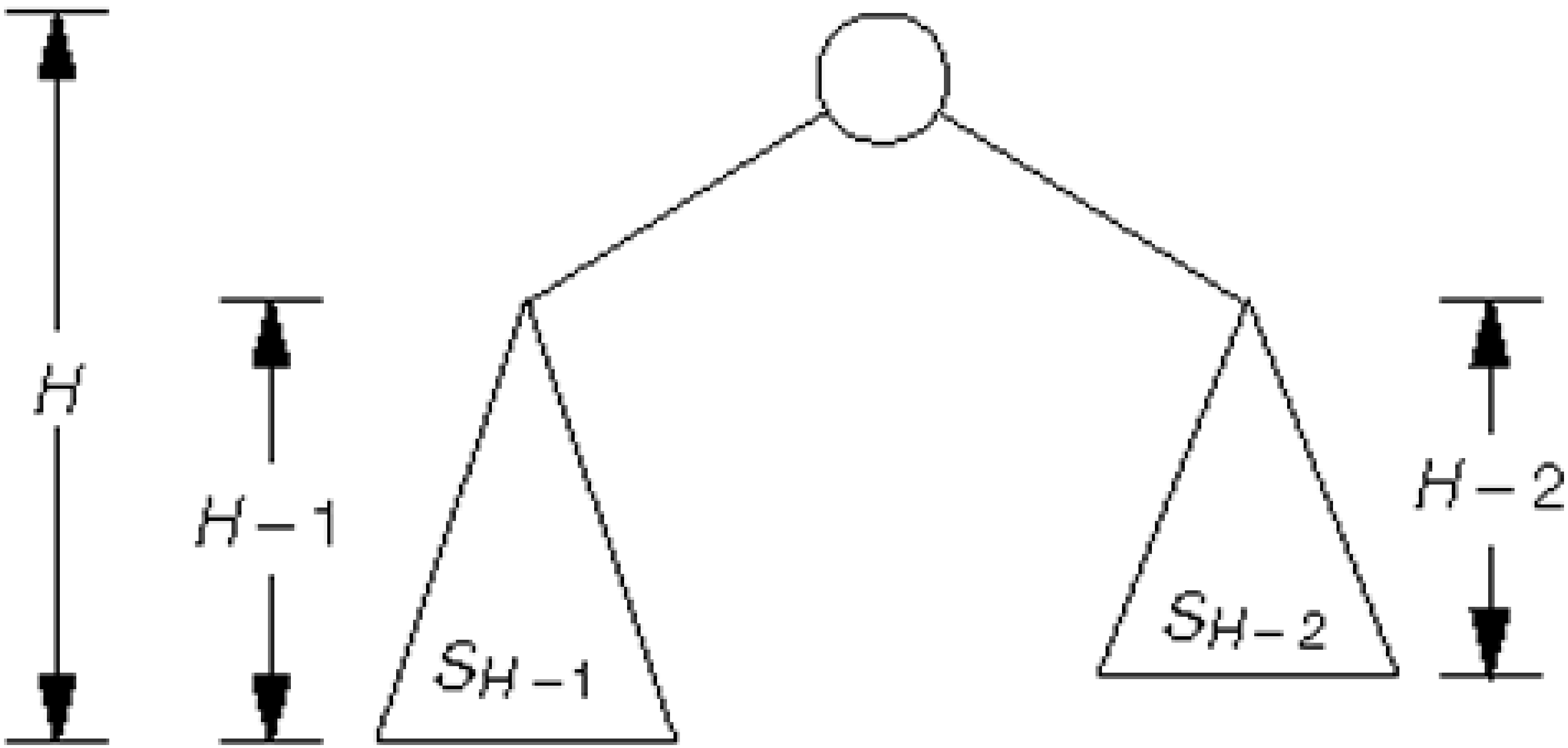
Two binary search trees: (a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)



(a)

(b)

Figure 19.22
Minimum tree of height *H*



Properties

- The depth of a typical node in an AVL tree is very close to the optimal *log N*.
- Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.
- An update (insert or remove) in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.
- After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

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*.

Single Rotation

- A single rotation switches the roles of the parent and child while maintaining the search order.
- Single rotation handles the outside cases (i.e. 1 and 4).
- We rotate between a node and its child.
  - Child becomes parent. Parent becomes right child in case 1, left child in case 4.
- The result is a binary search tree that satisfies the AVL property.

Figure 19.23

Single rotation to fix case 1: Rotate right

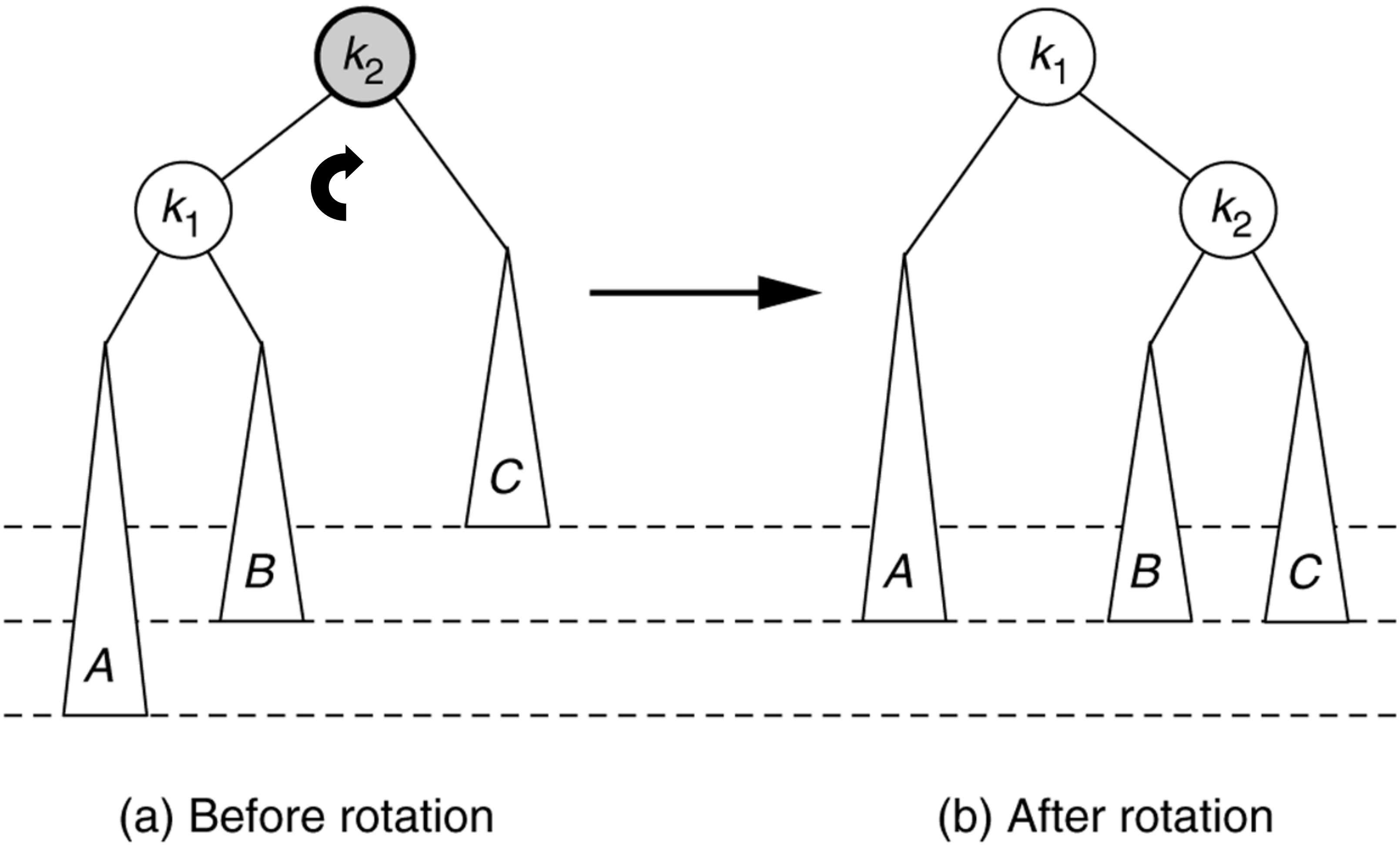
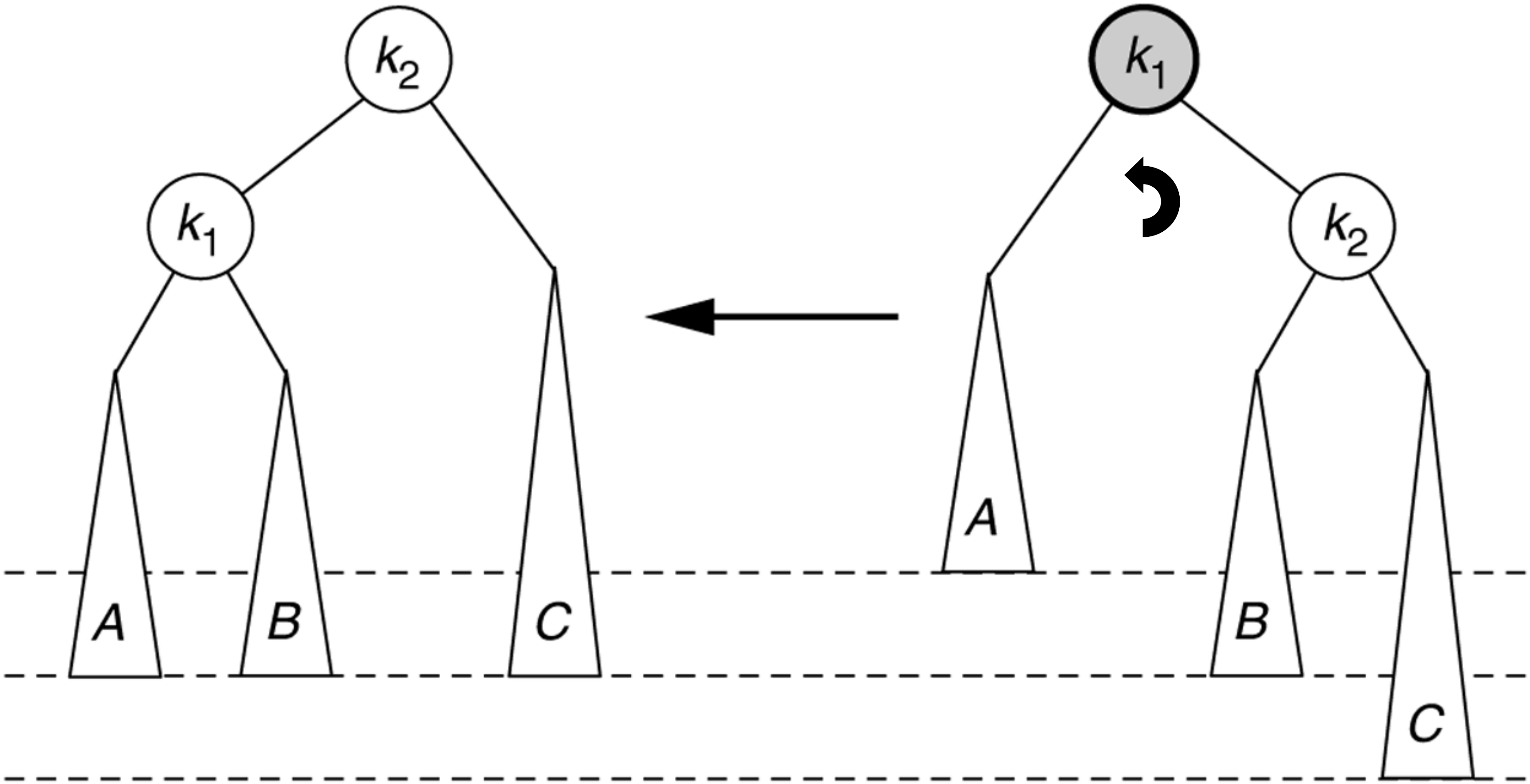


Figure 19.26

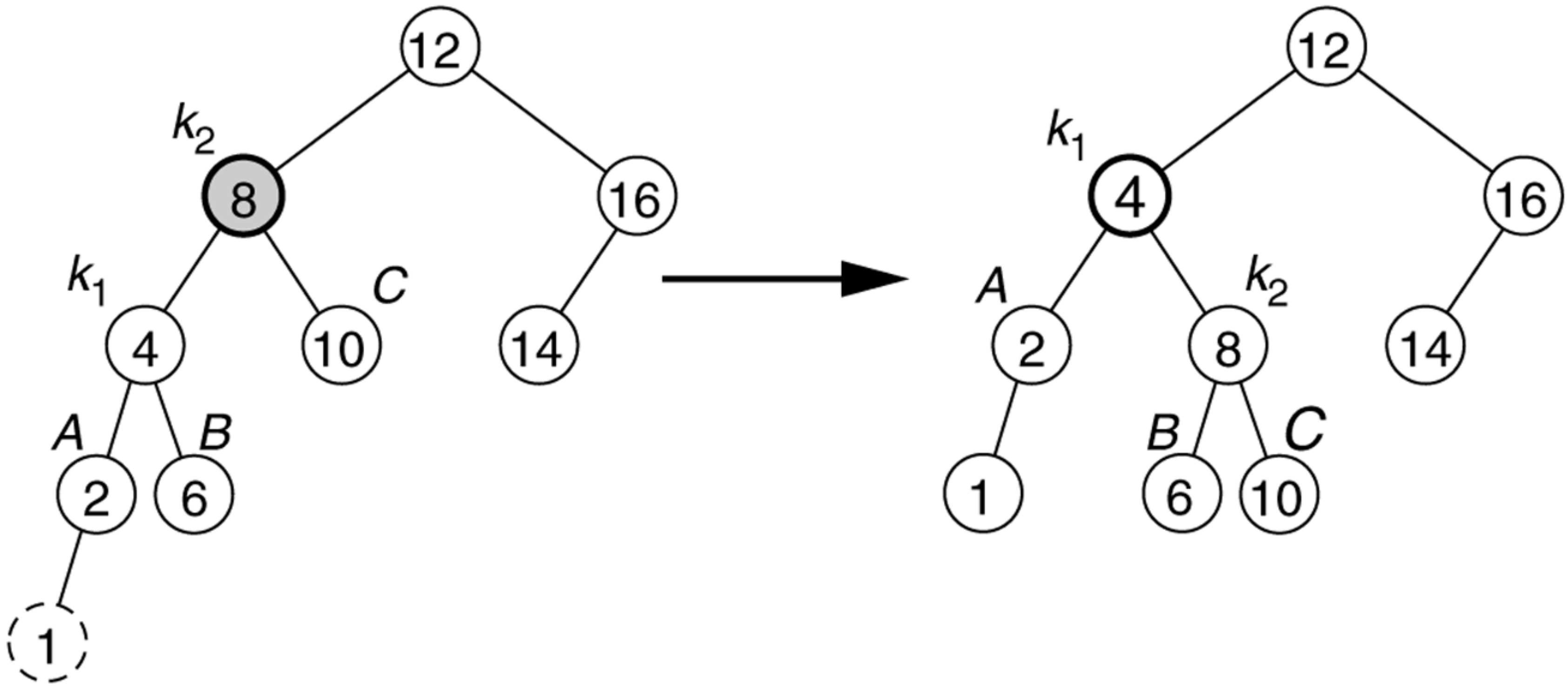
Symmetric single rotation to fix case 4 : Rotate left



(a) After rotation (b) Before rotation

Figure 19.25

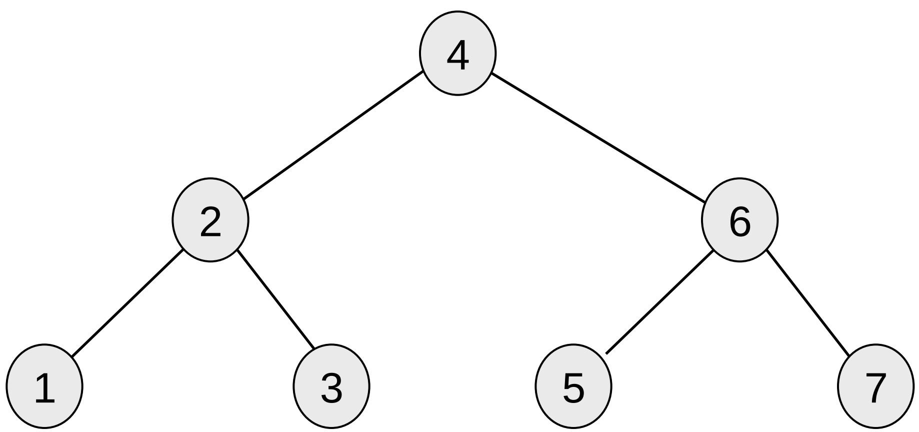
Single rotation fixes an AVL tree after insertion of 1.



(a) Before rotation (b) After rotation

Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:



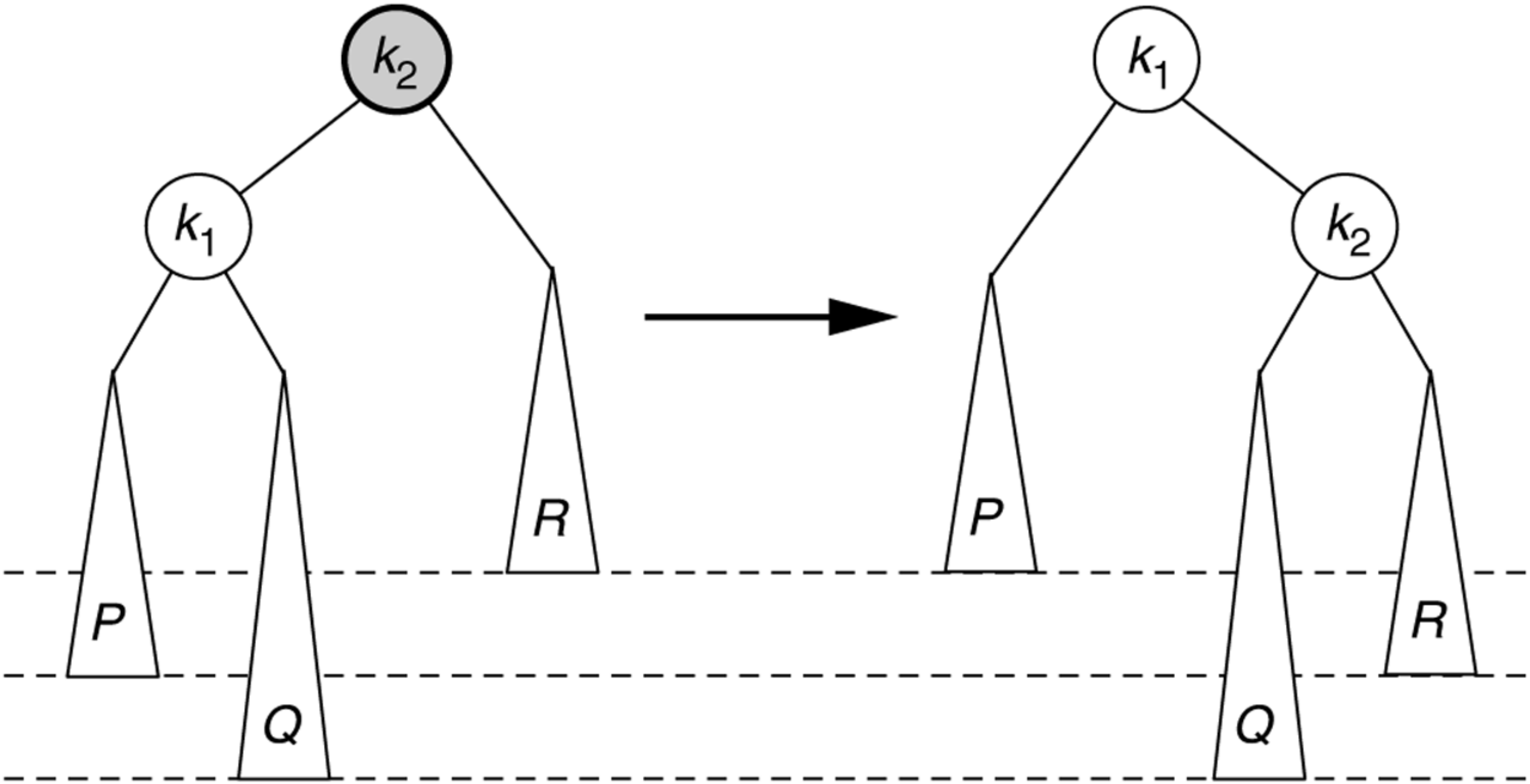
Analysis

- One rotation suffices to fix cases 1 and 4.
- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.
- Therefore it is enough to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- Thus the rotation takes O(1) time.
- Hence insertion is O(logN)

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.

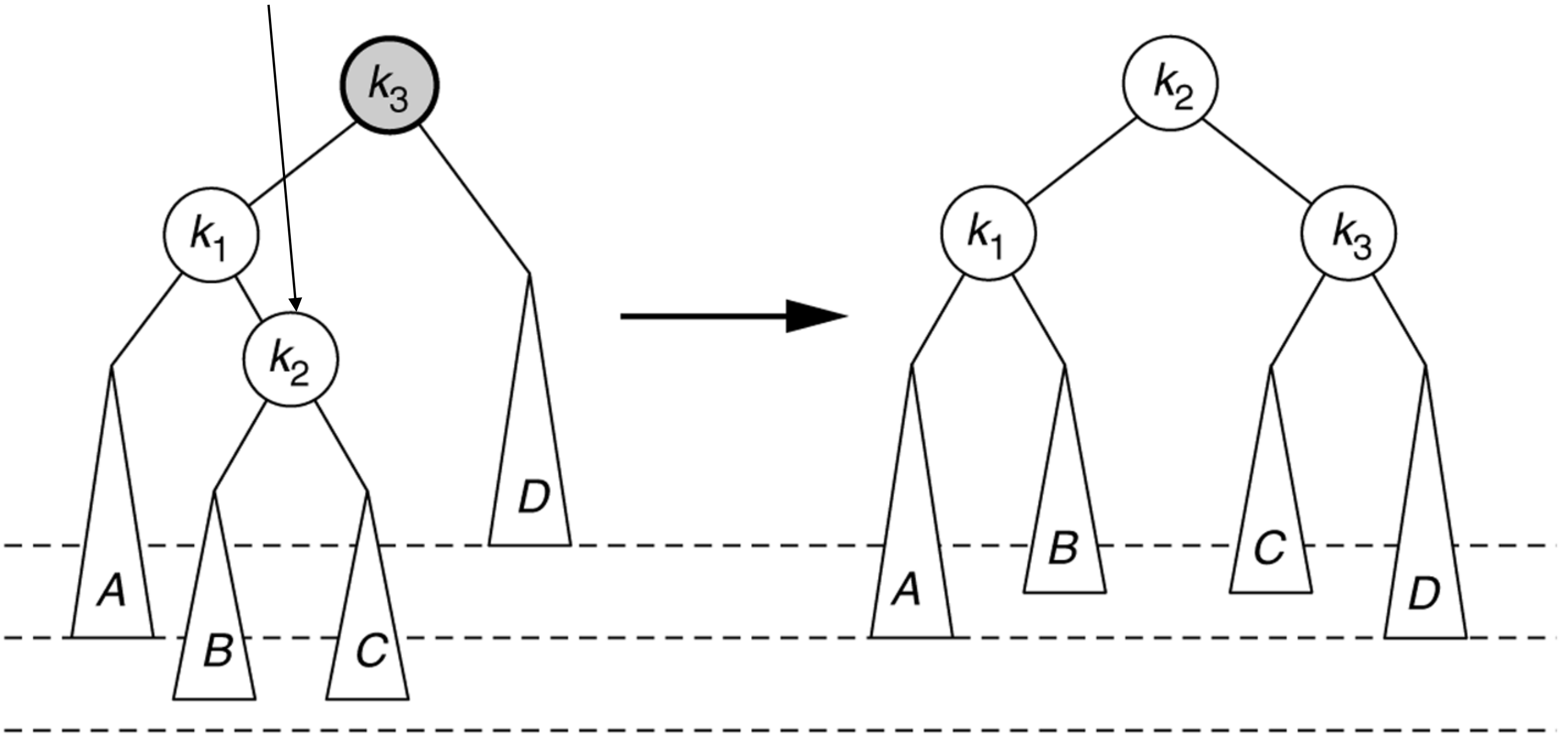
Figure 19.28
Single rotation **does not** fix case 2.



(a) Before rotation (b) After rotation

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

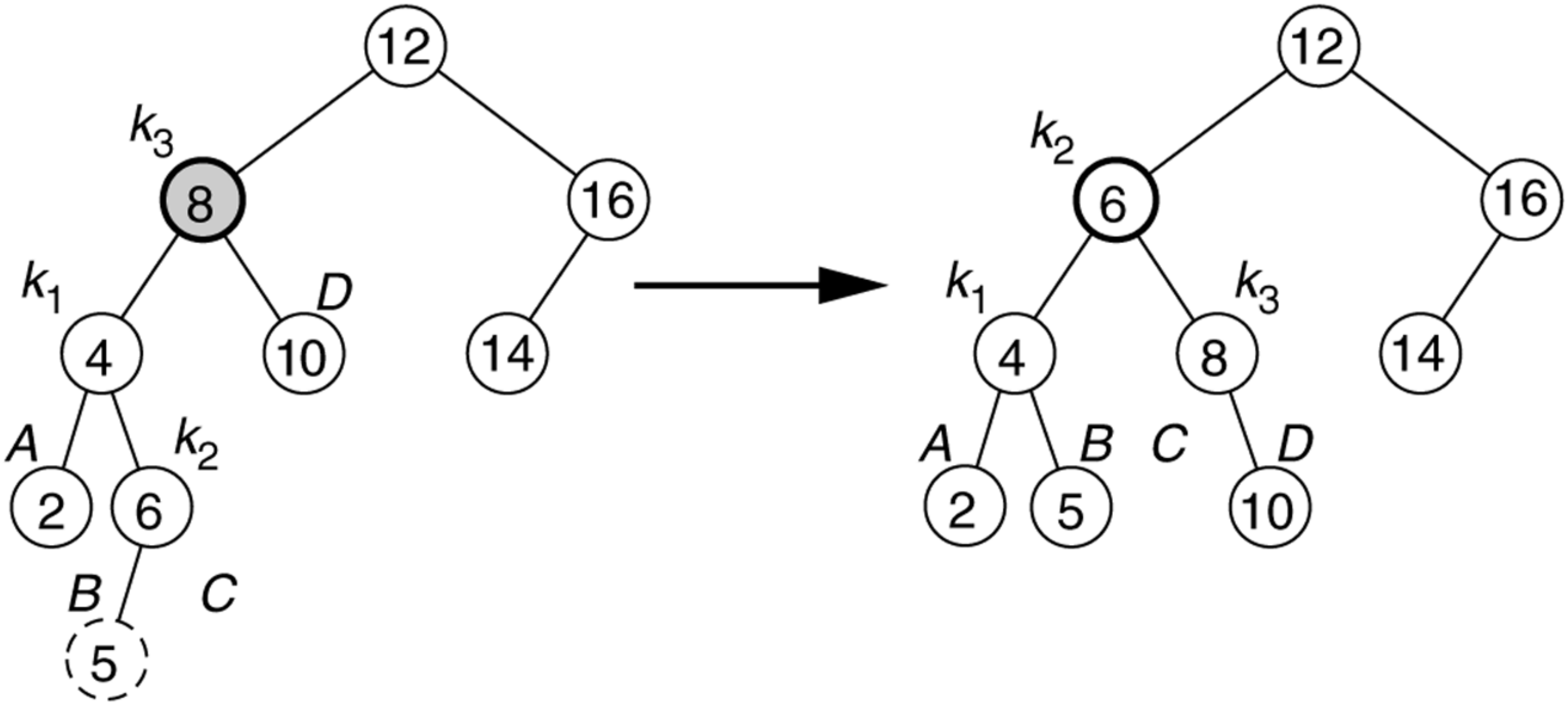
(b) After rotation

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.

Figure 19.30

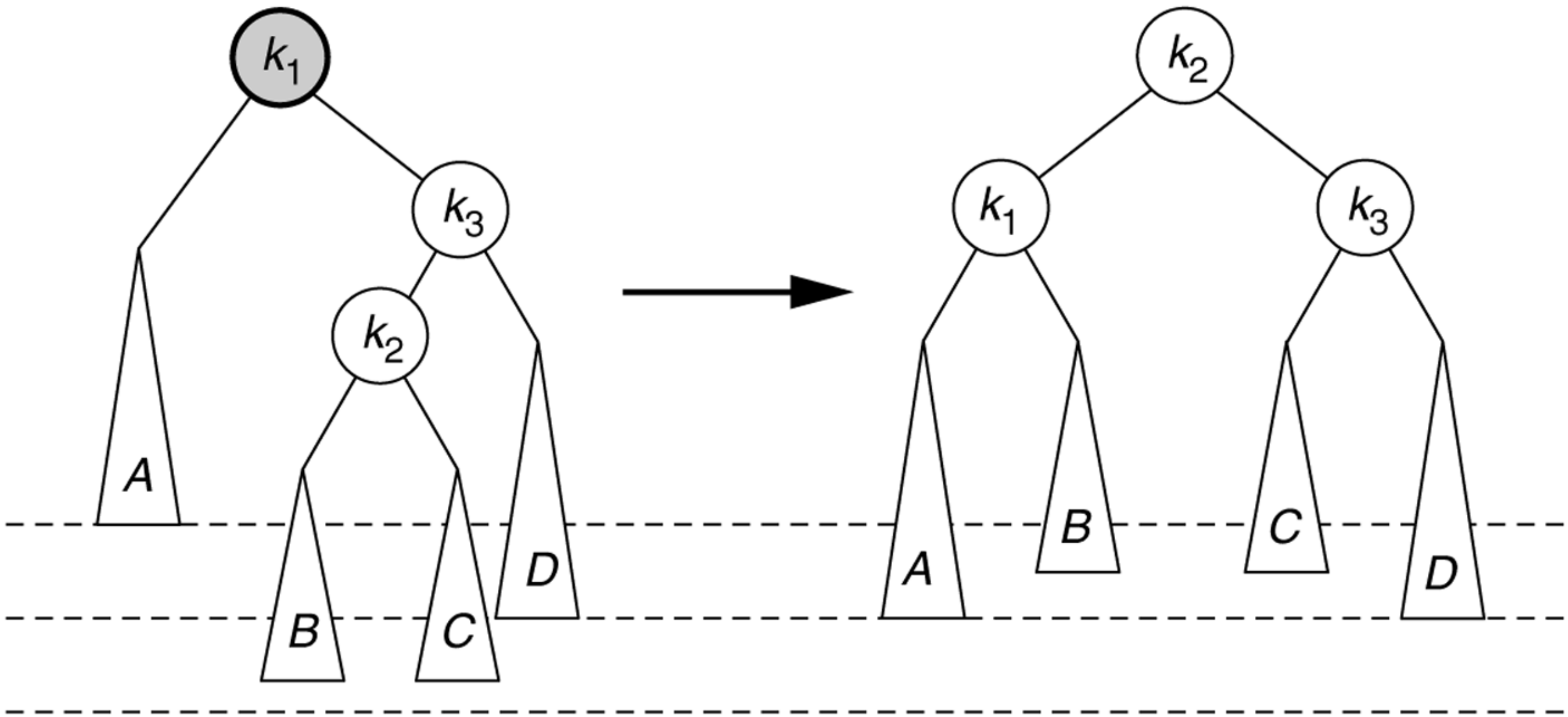
Double rotation fixes AVL tree after the insertion of 5.



(a) Before rotation

(b) After rotation

Right–Left double rotation to fix case 3.

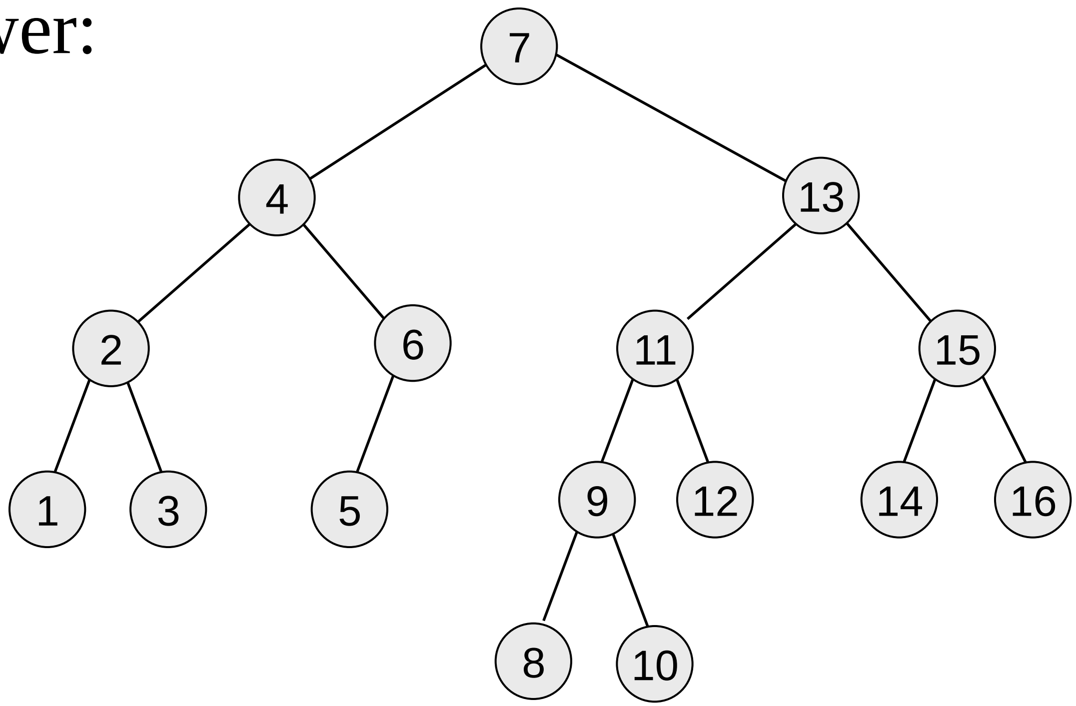


(a) Before rotation

(b) After rotation

Example

- Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the previous tree obtained in the previous single rotation example.
- Answer:



Node declaration for AVL trees

```
template <class Comparable>
class AvlTree;

template <class Comparable>
class AvlNode
{
   Comparable element;
   AvlNode   *left;
   AvlNode   *right;
   int       height;

   AvlNode( const Comparable & theElement, AvlNode *lt,
            AvlNode *rt, int h = 0 )
     : element( theElement ), left( lt ), right( rt ),
                height( h ) { }
   friend class AvlTree<Comparable>;
};
```

Height

```
template class <Comparable>
int AvlTree<Comparable>::height(
    AvlNode<Comparable> *t) const
{
    return t == NULL ? -1 : t->height;
}
```

Single right rotation

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
template <class Comparable>
void AvlTree<Comparable>::rotateWithLeftChild(
  AvlNode<Comparable> * & k2 ) const
{
   AvlNode<Comparable> *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;
}
```

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 Comparable>
void AvlTree<Comparable>::doubleWithLeftChild(
  AvlNode<Comparable> * & k3 ) const
{
   rotateWithRightChild( k3->left );
   rotateWithLeftChild( k3 );
}
```

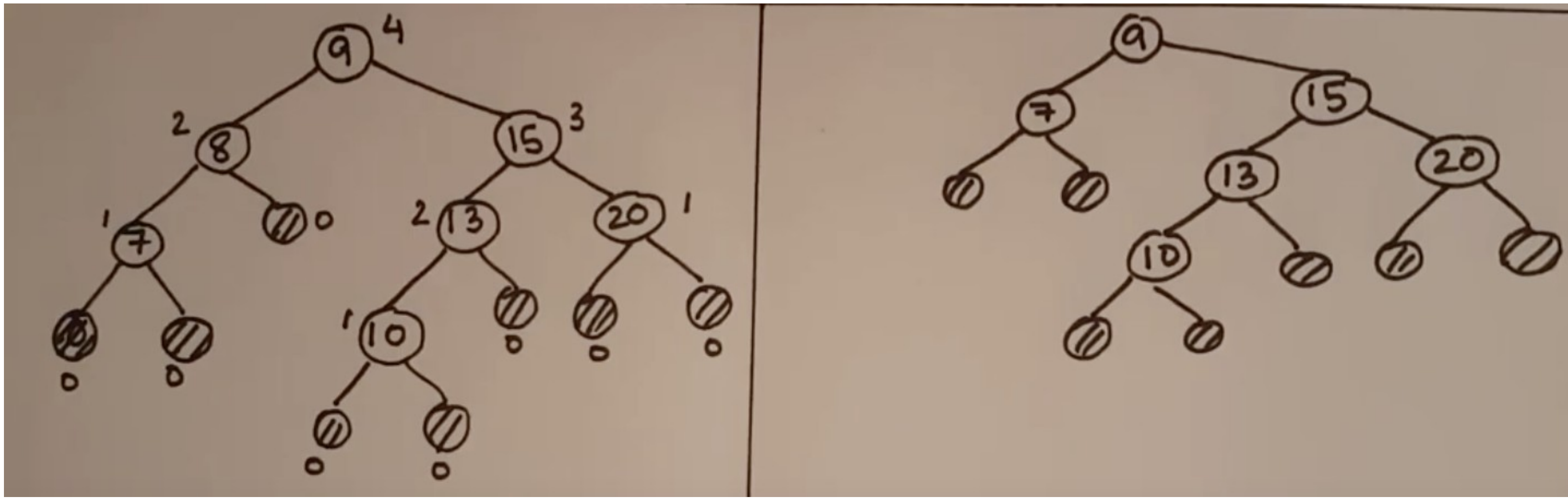
```
/* Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 */
template <class Comparable>
void AvlTree<Comparable>::insert( const Comparable & x, AvlNode<Comparable> * & t
   ) const
{
   if( t == NULL )
     t = new AvlNode<Comparable>( x, NULL, NULL );
   else if( x < t->element )
   {
     insert( x, t->left );
     if( height( t->left ) - height( t->right ) == 2 )
       if( x < t->left->element )
           rotateWithLeftChild( t );
       else
           doubleWithLeftChild( t );
   }
   else if( t->element < x )
   {
       insert( x, t->right );
       if( height( t->right ) - height( t->left ) == 2 )
          if( t->right->element < x )
             rotateWithRightChild( t );
          else
             doubleWithRightChild( t );
    }
    else
       ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

AVL Tree -- Deletion

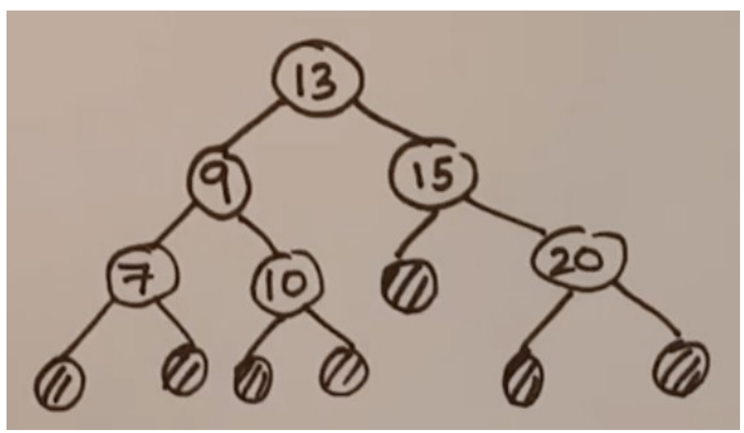
- Deletion is more complicated.
- We may need more than one rebalance on the path from deleted node to root.
- Deletion is O(logN)

Example

Delete 8. First regular BST deletion.

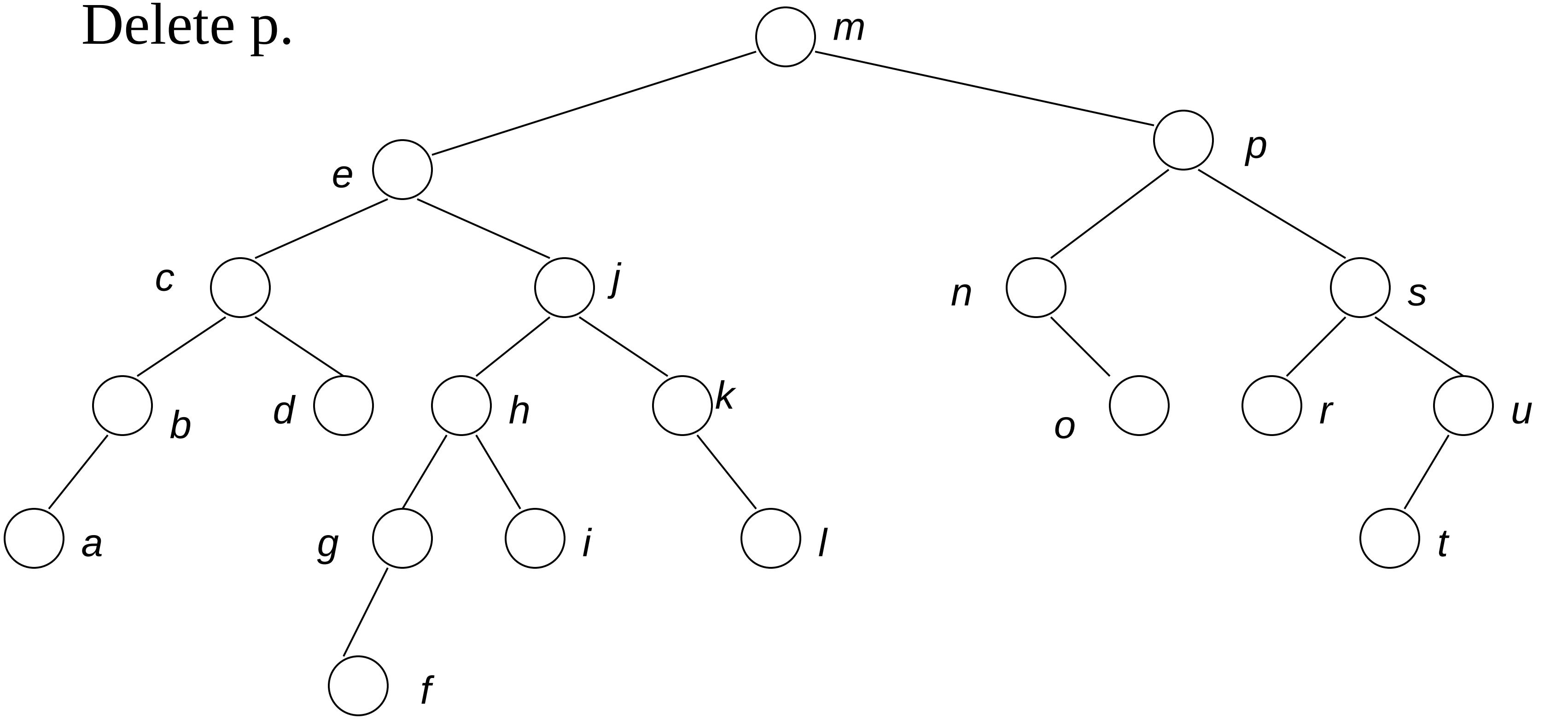


Then recovery. Case 3: RL.



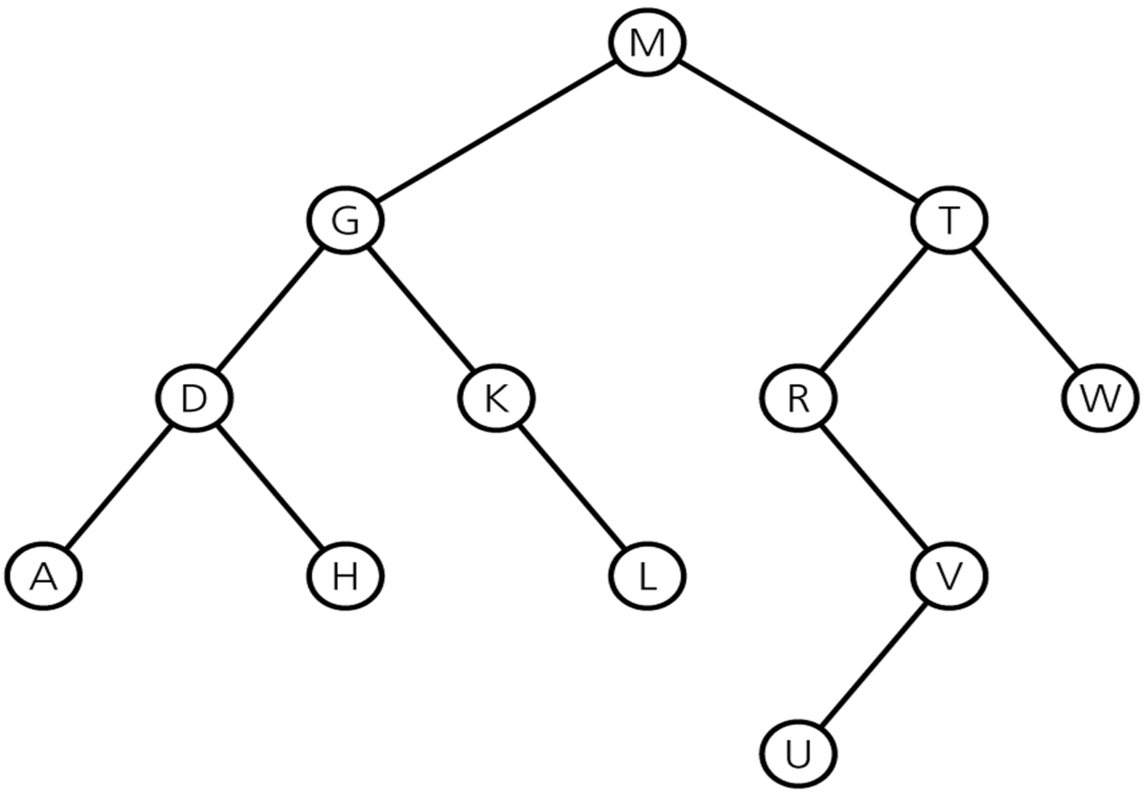
Example

Delete p.



Exercises

1. What are preorder, postorder and inorder traversals of the following binary tree.



2. Assume that the *inorder* traversal of a binary tree is

C G A H F D E I B J

and its *postorder* traversal is

G C H F A I E J B D

Draw this binary tree.