19.4 – AVL Trees

1. One technique for enforcing balance in the tree is to require that the height of the left and right subtrees for any node differ by no more than 1. An *AVL Tree* is defined to be a binary search tree with this balance property. In the work that follows, the height of an empty subtree is defined to be -1.

2. Example:

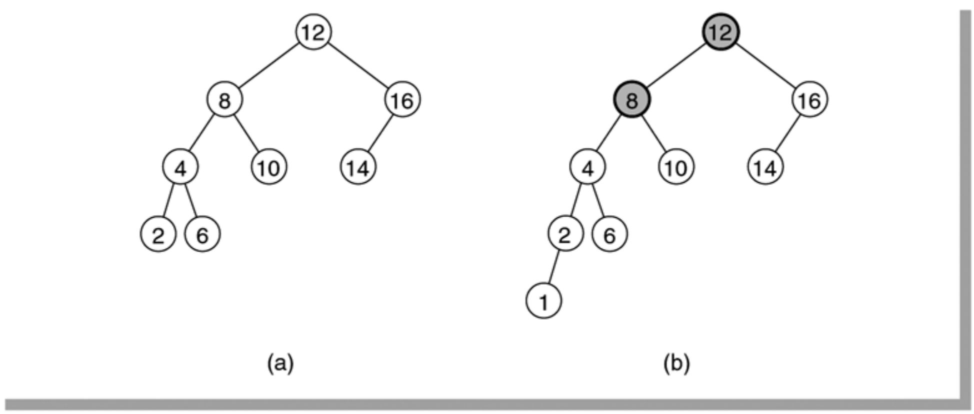


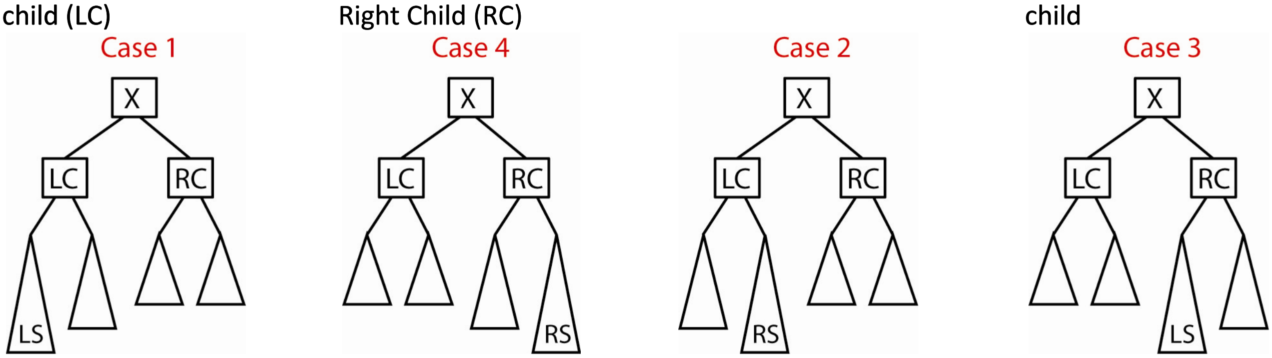
figure 19.21

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

3. Note in figure 19.21a above, that *insert(1)*, using the BST algorithm (Section 19.1) will destroy the AVL property as shown in figure 19.21b above. Thus, we will have to modify the insert (and remove) algorithms so that they don't destroy the AVL property. The following algorithm, called *single rotation*, finds the deepest node that violates the property (node 8 in figure 19.21b above) and rebalances the tree from there. It can be proven that this rebalancing guarantees that the entire tree satisfies the AVL property.

4. Consider inserting a node into an AVL Tree. Suppose a height imbalance of 2 results at some deepest node, X. Thus, X needs to be rebalanced. Assuming an AVL tree (e.g. balance condition met) existed before the insertion, relative to X, we can define these 4 places where the insertion took place:

Left subtree (LS) of Left child (LC)	Right subtree (RS) of Right Child (RC)	Right subtree of Left child	Left subtree of Right child



Homework 19.8

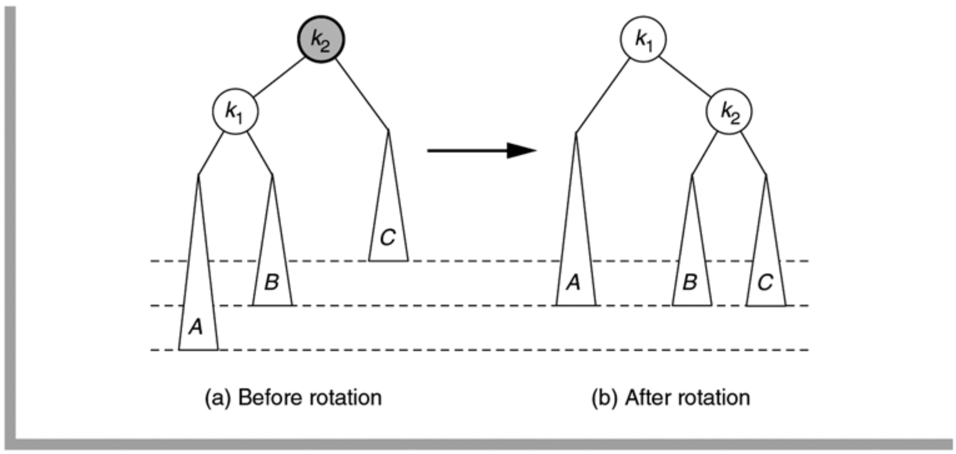
1. Describe the 4 cases that can result when inserting a new node into an AVL tree.

19.4 – AVL Trees, Single Rotation

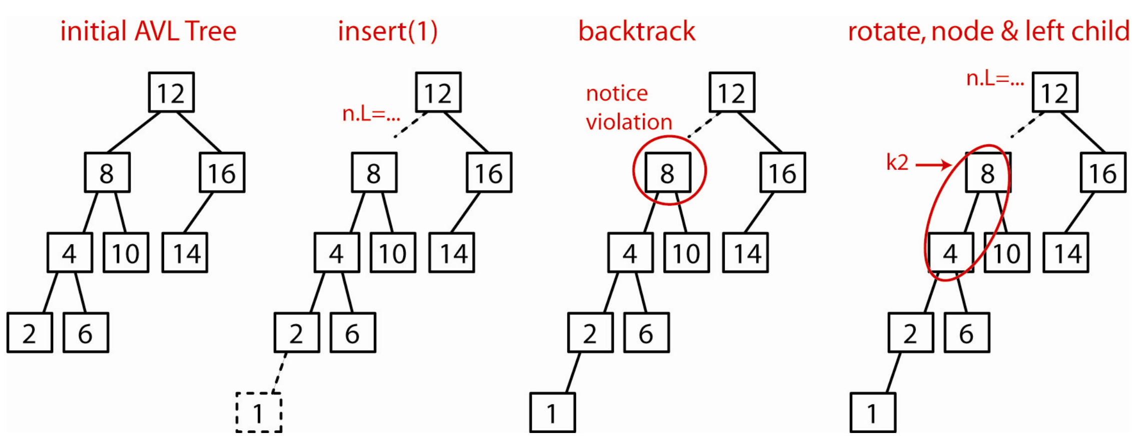
1. Consider Case 1. In figure 19.23a below, node *k2* plays the role of *X* in preceding figure, the deepest node where the imbalance is observed, and *k1* is the left child of *k2*. The idea is to rotate *k1* and *k2* clockwise making *k2* the right subtree of *k1* and making *B* the left subtree of *k2*. It is easy to verify that this approach works. First, *k2* is larger than *k1*, thus *k2* can be the right child of *k1*. Second, all nodes in B are between *k1* and *k2* which is still true when *B* becomes *k2*'s left child.

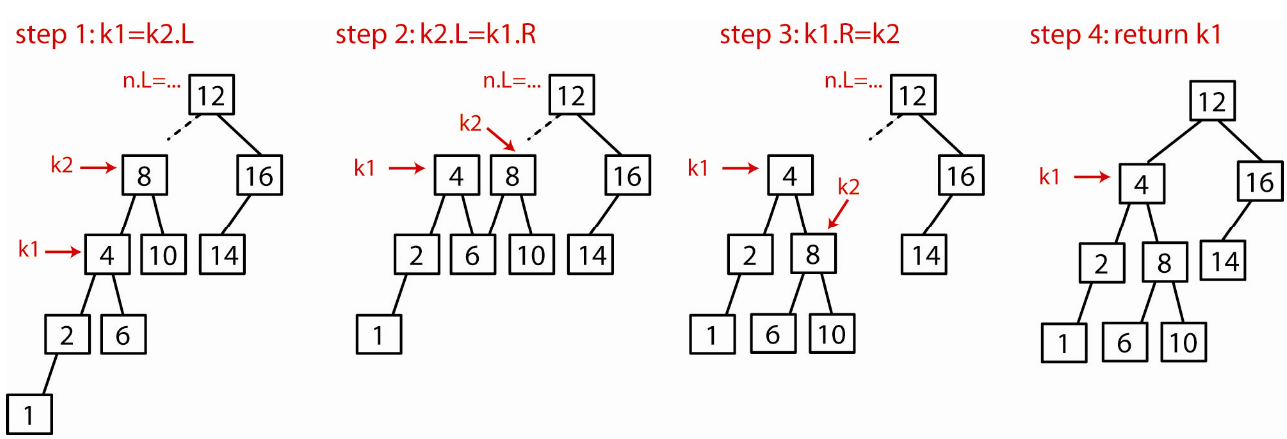
figure 19.23

Single rotation to fix case 1



2. Consider an example of the Case 1 algorithm which is shown in 4 steps below:





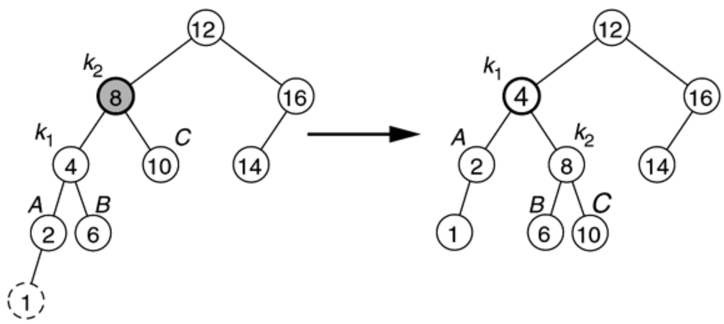
3. Implementation of the algorithm for Case 1:

figure 19.24

Pseudocode for a single rotation (case 1)

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 */
static BinaryNode rotateWithLeftChild( BinaryNode k2 )
{
    BinaryNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}
```

4. Same example, using figure 19.23's notation.

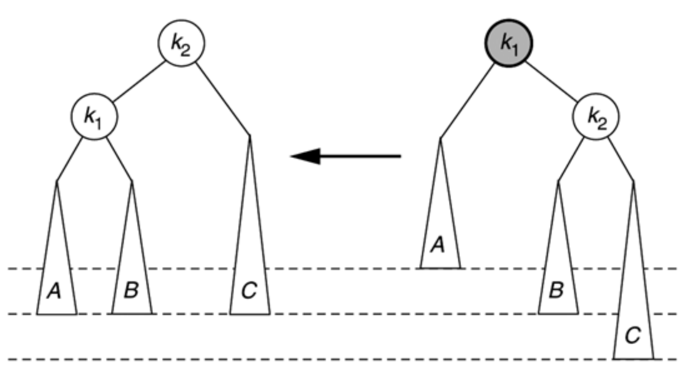


(a) Before rotation (b) After rotation

figure 19.25

Single rotation fixes an AVL tree after insertion of 1.

5. Consider Case 4 which is a mirror image of Case 1. In figure 19.26b below, node *k1* plays the role of *X*, the deepest node where the imbalance is observed, and *k2* is the right child. The idea is to rotate *k1* and *k2* counter-clockwise *k1* the left child of *k2* and making *B* the right child of *k1.* It is easy to verify that this approach works. First, *k1* is smaller than *k2*, thus it can be the left child of *k2*. Second, all nodes in *B* are between *k1* and *k2* which is still true when *B* becomes *k1*'s right child.

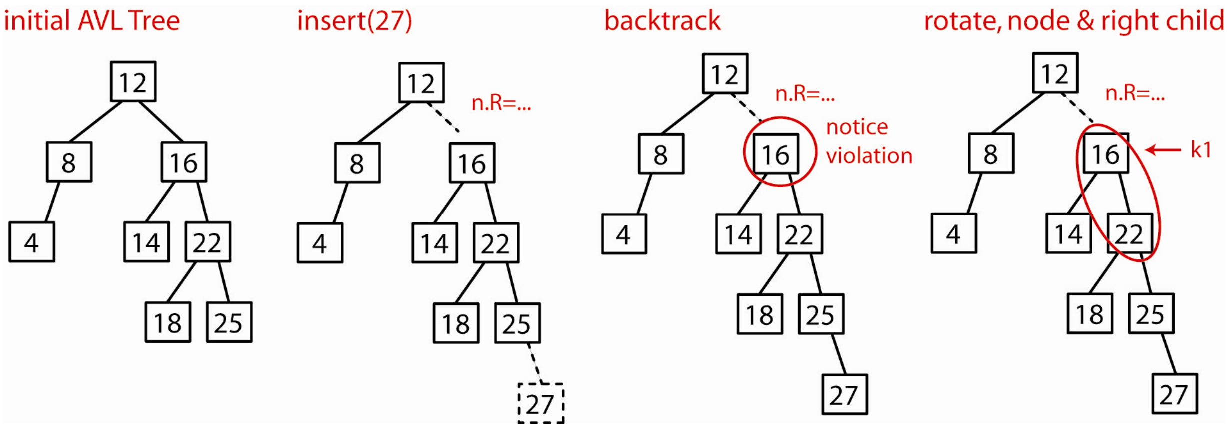


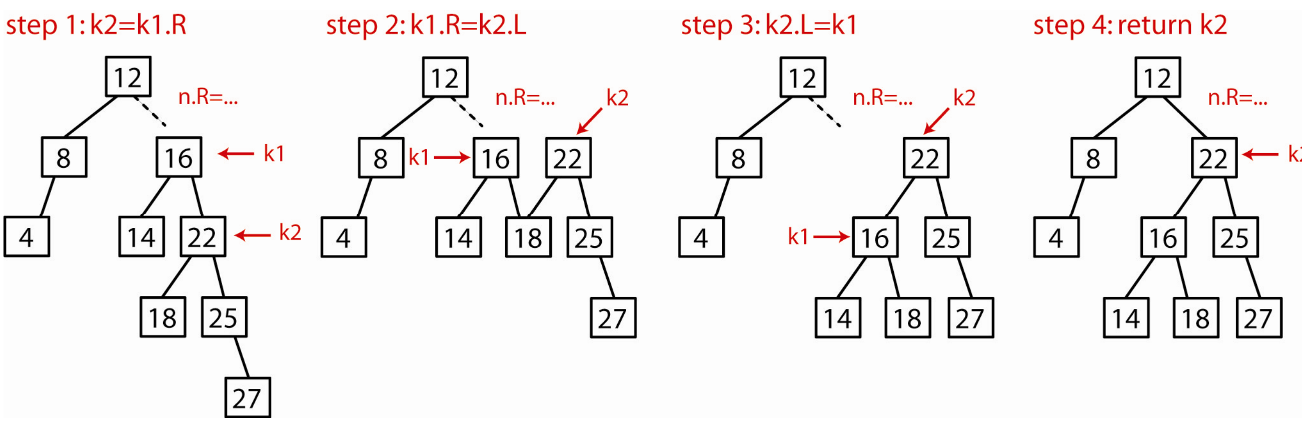
(a) After rotation (b) Before rotation

figure 19.26

Symmetric single rotation to fix case 4

6. Consider an example of the Case 4 algorithm which is shown in 4 steps below:





7. Implementation of the algorithm for Case 4:

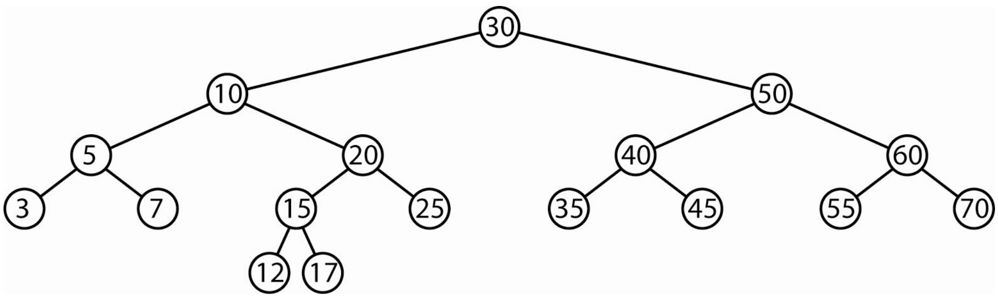
```
/**
 * Rotate binary tree node with right child.
 * For AVL trees, this is a single rotation for case 4.
 */
static BinaryNode rotateWithRightChild( BinaryNode k1 )
{
    BinaryNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}
```

figure 19.27

Pseudocode for a single rotation (case 4)

Homework 19.9

1. Show the AVL tree that results when 11 is inserted.



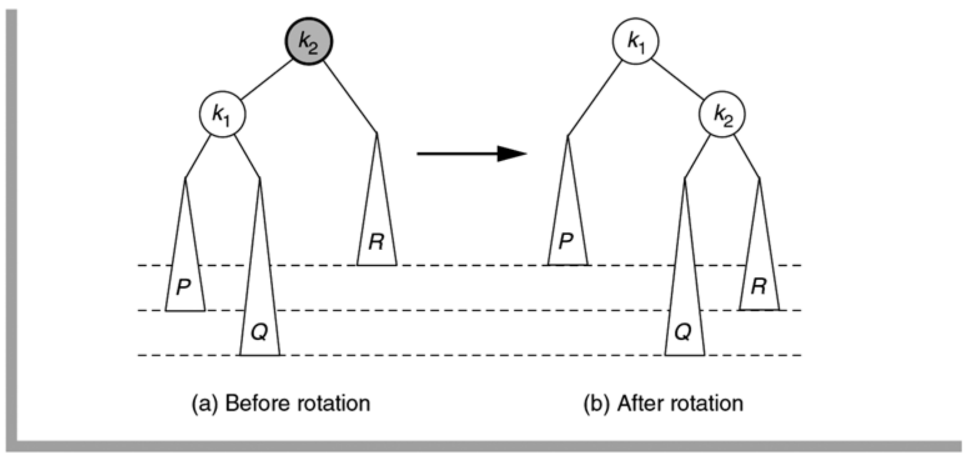
2. Using the tree that results from problem 1, show the AVL tree that results when 27 is inserted.

19.4 – AVL Trees, Double Rotation

1. Consider Case 2. A rotation as described above, does not work.

figure 19.28

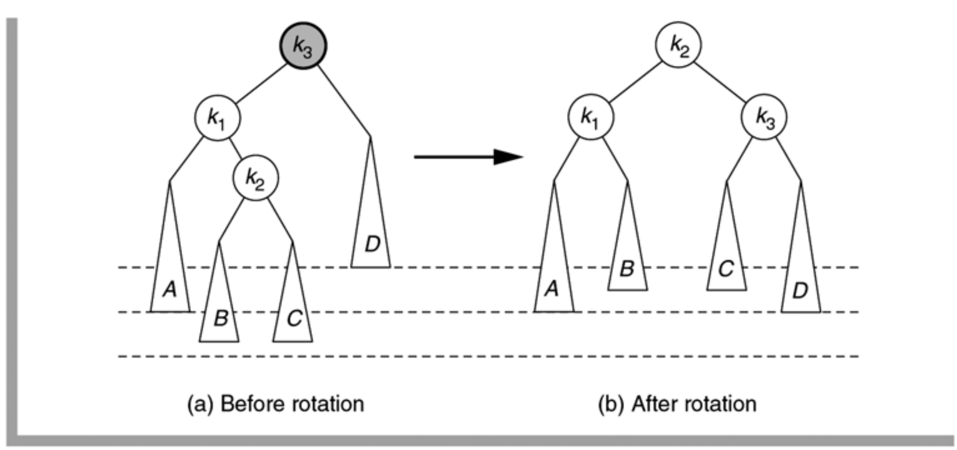
Single rotation does not fix case 2.



2. Consider Case 2 again. A *double rotation* does work. Since k1<k2<k3, the nodes can be rearranged so that k1 and k3 are the left and right, respectively, subtrees of k2. Since all elements in B are between k1 and k2 they remain that way when we make k1's right child be B. Similarly, since all elements in C are between k2 and k3, we can make C k3's left child.

figure 19.29

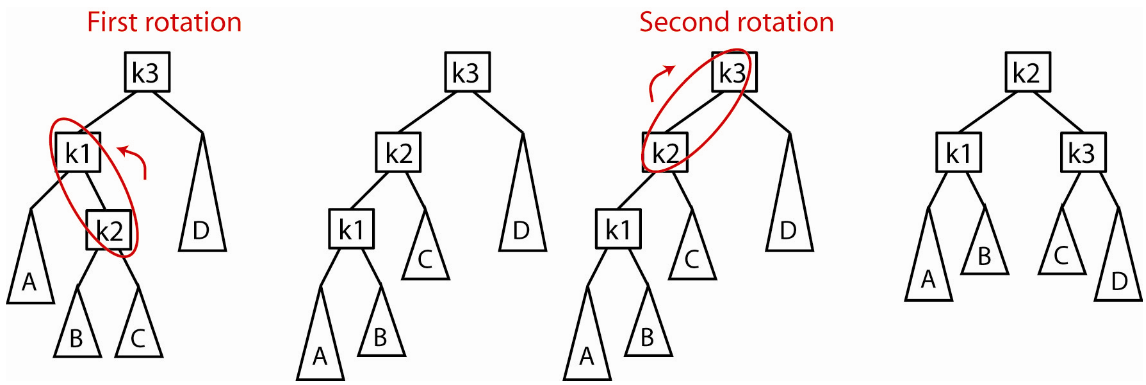
Left–right double rotation to fix case 2



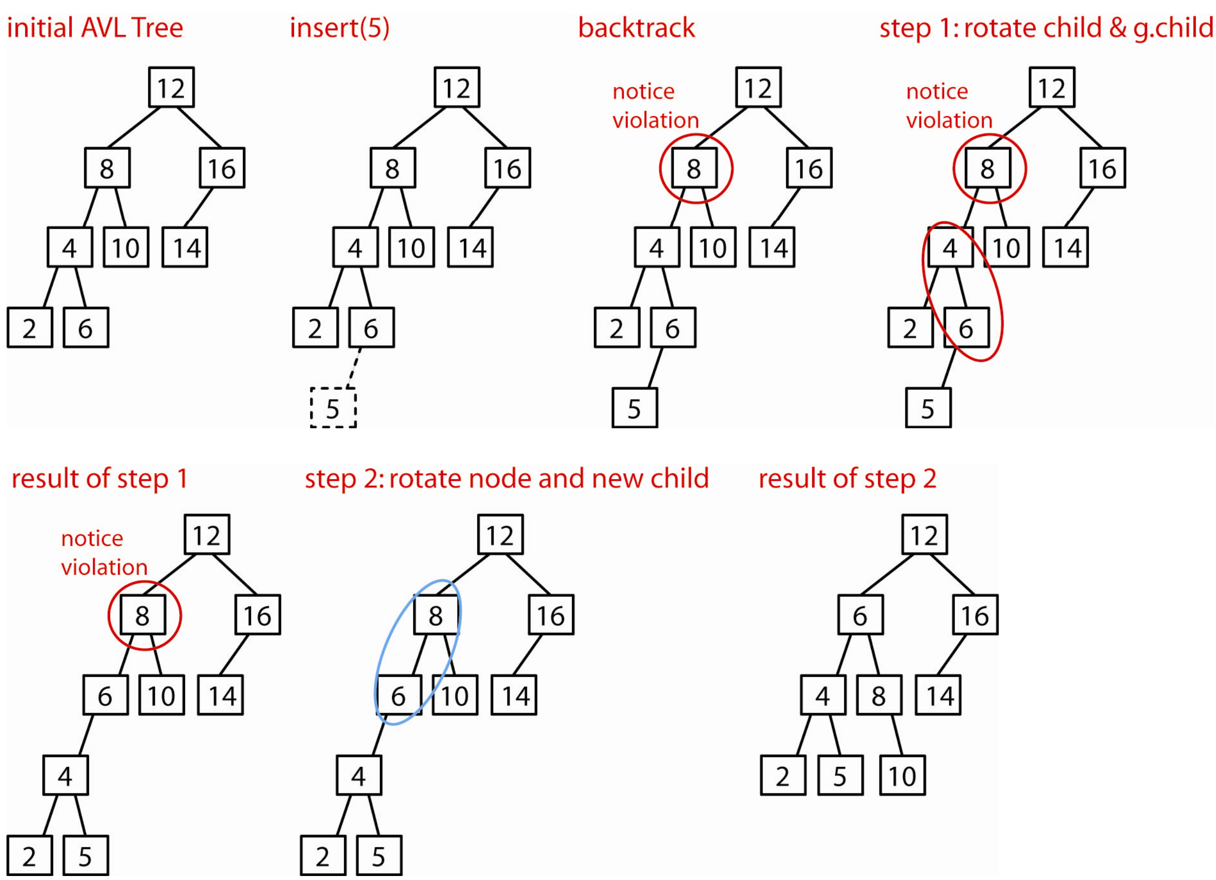
3. The double rotation can be seen as two single rotations as described for cases 1 and 4:

    1. Rotate X's child and grandchild
    2. Rotate X and its new child

    In figure 19.29 above, first rotate k1 and k2 counter-clockwise, then rotate k2 and k3 clockwise:



5. Example:



6. Same example using figure from text:

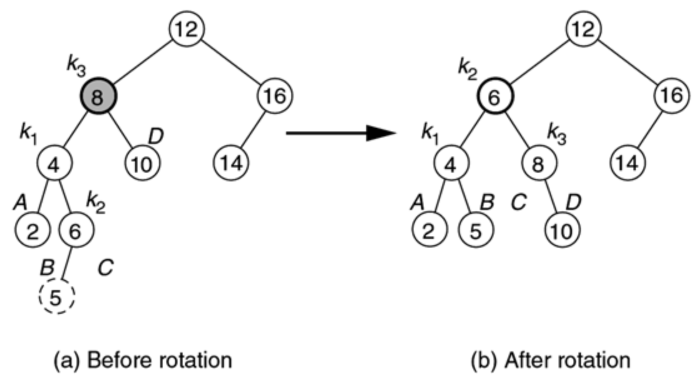


figure 19.30

Double rotation fixes AVL tree after the insertion of 5.

7. Case 3 is a mirror image of Case 2. Here, we rotate k2 and k3 clockwise then rotate k1 and k2 counter-clockwise.

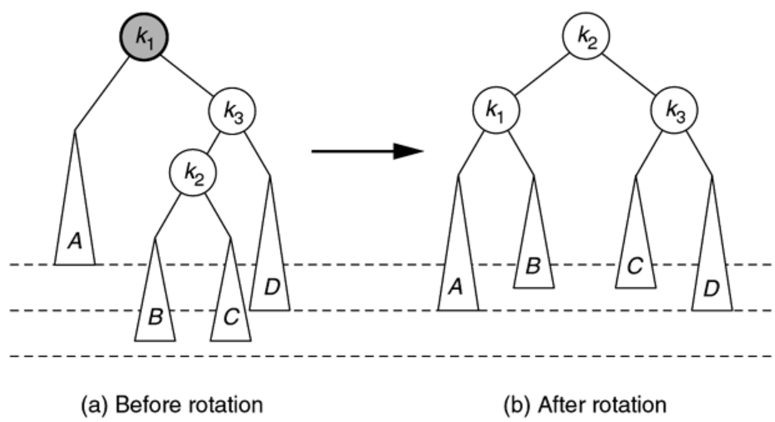


figure 19.31

Right–Left double rotation to fix case 3.

8. Java implementation of Case 2 and Case 3

figure 19.32

Pseudocode for a double rotation (case 2)

```
/**
 * 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.
 */
static BinaryNode doubleRotateWithLeftChild( BinaryNode k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}
```

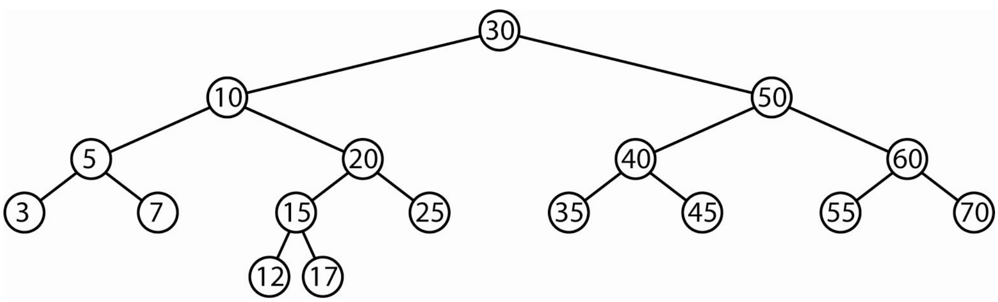
figure 19.33

Pseudocode for a double rotation (case 3)

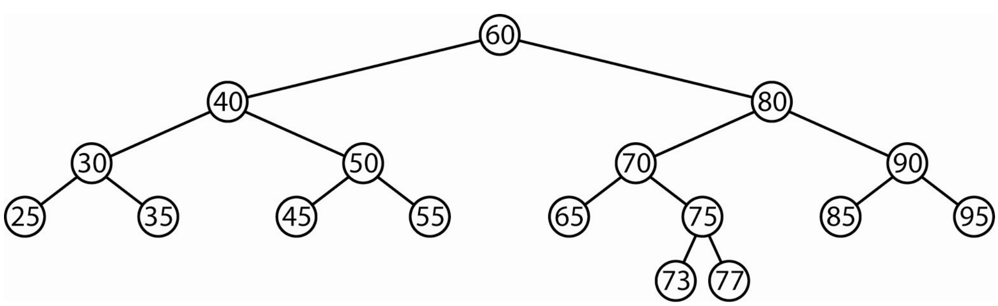
```
/**
 * Double rotate binary tree node: first right child
 * with its left child; then node k1 with new right child.
 * For AVL trees, this is a double rotation for case 3.
 */
static BinaryNode doubleRotateWithRightChild( BinaryNode k1 )
{
    k1.right = rotateWithLeftChild( k1.right );
    return rotateWithRightChild( k1 );
}
```

Homework 19.10

1. Show the AVL tree that results when 18 is inserted.



2. Using the tree that results from problem 1, show the AVL tree that results when 19 is inserted.
3. Show the AVL tree that results when 72 is inserted.



4. Using the tree that results from problem 3, show the AVL tree that results when 78 is inserted.
5. Problem 19.3, p. 765 (skip probability part)
6. Problem 19.5, p.765 (skip red-black tree part).

19.4 – AVL Trees, Rotation Summary

1. Summary of rotation: Consider the path from the deepest node where the imbalance is first detected to the grandchild of this node with greatest height. Call the nodes on this path: *grandparent, parent, node:*

   a. Call the node where the imbalance is detected the ***grandparent***
   b. Call the grandchild with greatest height (where the imbalance occurs), ***node***
   c. Call node's parent, ***parent***

   An algorithm for rotation:

```
if node is left child and parent is left child          // case 1
   rotate( parent, grandparent, CW )

else if node is right child and parent is left child   // case 2
   rotate( node, parent, CCW )
   rotate( node, grandparent, CW )

else if node is left child and parent is right child   // case 3
   rotate( node, parent, CW )
   rotate( node, grandparent, CCW )

else if node is right child and parent is right child  // case 4
   rotate( parent, grandparent, CCW )
```

2. Several notes:

   a. To properly connect the tree, we would also need to keep track of the great-grandparent.

   b. We must also keep track of height information, and update it correctly.

   c. We have not considered the remove algorithm.

19.4 – AVL Trees, insert method

1. The basic idea of a recursive *insert* algorithm is that we recursively insert the element into the appropriate subtree. If the insertion does not cause the subtree's height to change, we are done. Otherwise, if an imbalance of 2 is detected then we do either a single or double rotation.

```java
private AvlNode<T> insert( T x, AvlNode<T> t )
{
    if( t == null )
        t = new AvlNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
   {
        t.left = insert( x, t.left );
        if( height( t.left ) - height( t.right ) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0 )
    {
        t.right = insert( x, t.right );
        if( height( t.right ) - height( t.left ) == 2 )
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );
    }
    else
    ;  // Duplicate; do nothing
    t.height = max( height( t.left ), height( t.right ) ) + 1;
    return t;
}
```

Note that the rotate algorithms would be required to update the height information.

```java
private static AvlNode 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;
    return k1;
}
```

A more efficient approach than the recursive insert is to use an iterative algorithm.