This equation is solved fairly easily (see Exercise 13.12 on page 411) to obtain

$$f(n) = \sum_{k=1}^{n} \frac{2k-1}{k(k+1)} \tag{13.8}$$

This result can be simplified (see Exercise 13.13 on page 411) to

$$f(n) = 2H_{n+1} - 3 + \frac{1}{n+1} \tag{13.9}$$

Combining this equation with Equation (13.6) yields the final result in Formula 13.3 on page 385.

13.5 AVL Trees

An AVL tree[2] is a binary search tree that maintains its balance by forcing the two subtrees at any node to have nearly the same height. This is done by rotating subtrees whenever an imbalance occurs.

For example, suppose we build a BST by inserting the keys 20, 30, 40, 50, 60, and 70, in that order. Without any extra balancing, the resulting BST would be a linear list. But instead, suppose that we perform a rotation about a node whenever an insertion causes one of that node's subtrees to be more than one level deeper than the other subtree. With this input sequence, that will first happen after 40 is inserted. (See Figure 13.10.)

Before the rotation, node x has a (NIL) left subtree of height -1 and a right subtree of height 1. After the rotation, both subtrees have height 0.

Continuing the insertions, we have another correctable imbalance after 60 is inserted. (See Figure 13.11.)

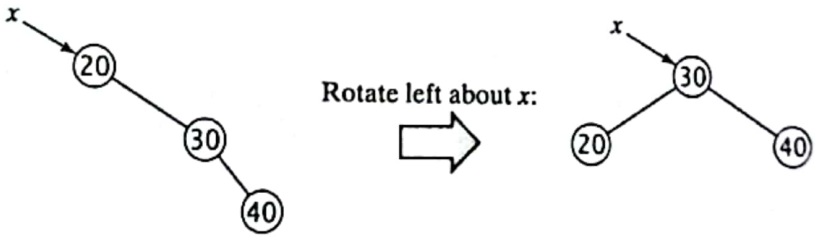


FIGURE 13.10 The AVL Rotate Left algorithm.

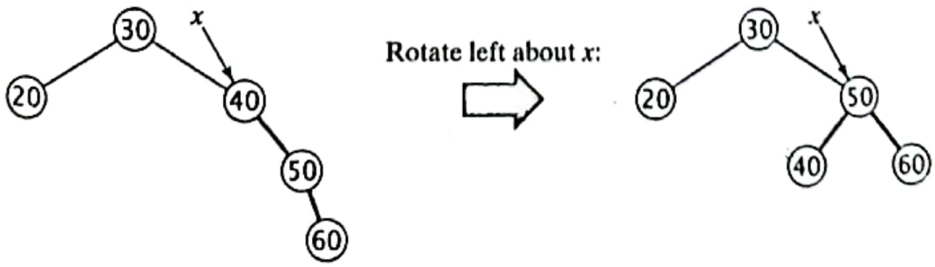


FIGURE 13.11 The AVL Rotate Left algorithm.

[2] The name comes from the two inventors of this data structure: G. M. Adel'son-Velskii and Y. M. Landis.

Scanned with CamScanner

Again, we rotate a three-generation node triple (node–parent–grandparent), pivoting this time about the grandparent $x = (40)$. The path that connects the triple is marked by darker lines.

Inserting 70 requires another rotation, as shown in Figure 13.12.

On this third rotation, the pivot node x is the root again: Before the rotation, x's left subtree has height 0 and its right subtree has height 2; afterwards, both have height 1.

On that third rotation, notice how the node (40) shifts from being the left child of the right child of x to being the right child of the left child of x. This is necessary to maintain the required *BST property* that an inorder traversal will visit the keys in ascending order: Node (40) must remain between (30) and (50).

Figure 13.13 shows the general action of the *rotate left* operation.

The subtree B gets shifted from node (b) to node (a). Notice that the rotation will reduce the height of the subtree rooted at x, as long as the heights of subtrees A and B are no greater than those of C and D.

Continuing our example, suppose that 66 is inserted next, as in Figure 13.14.

The imbalance occurs at node $x = [60]$. Since the parity (left or right) of the child $y = [70]$ is the opposite of that of its new child [66], a double rotation is required. First, a right, two-node mini-rotation is made at y to restore the parity (both right). Then a left rotation is made about node x.

The general action of the compound *rotate right–left* operation is shown in Figure 13.15.

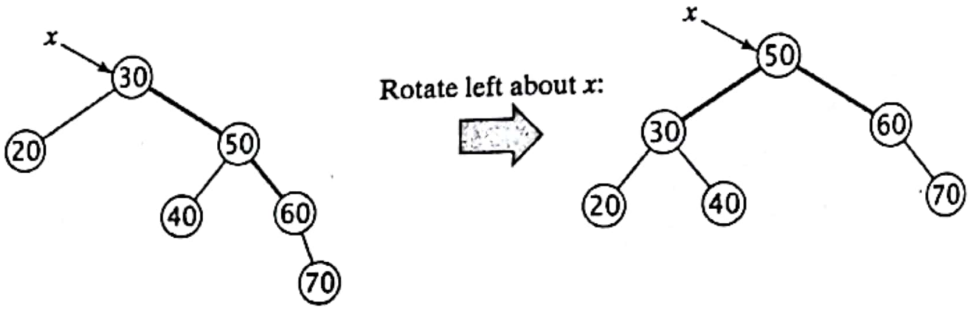


FIGURE 13.12 The AVL Rotate Left algorithm.

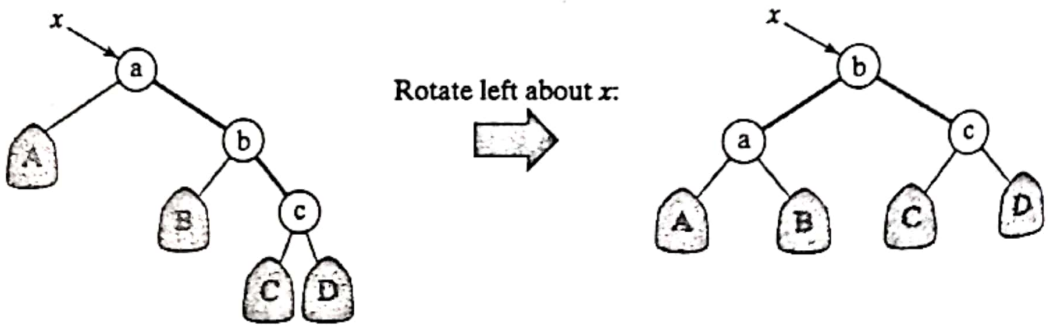


FIGURE 13.13 The AVL Rotate Left algorithm.

Scanned with CamScanner

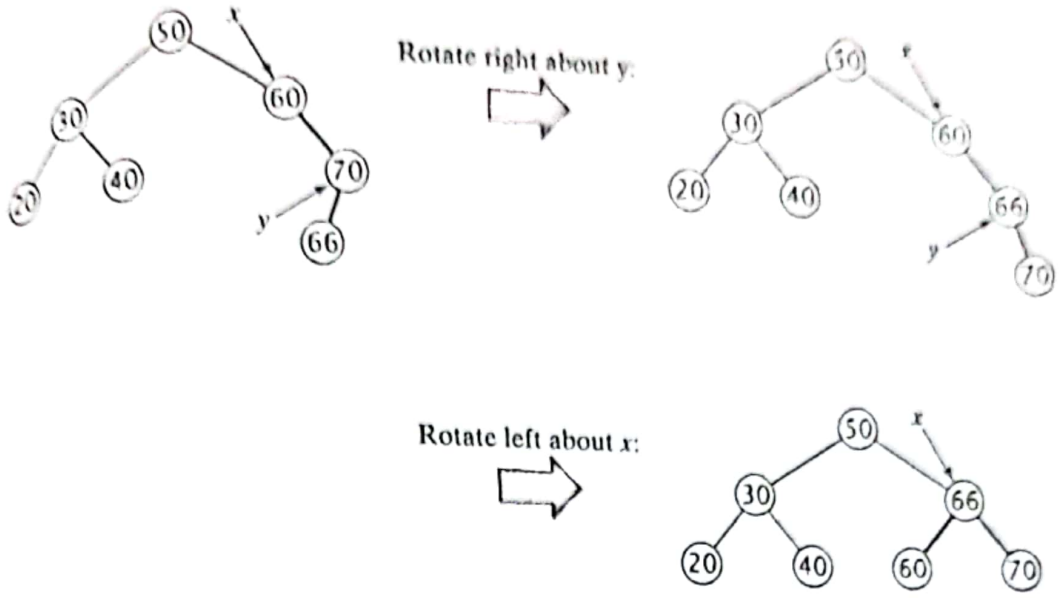


FIGURE 13.14 The AVL Double Rotation algorithm.

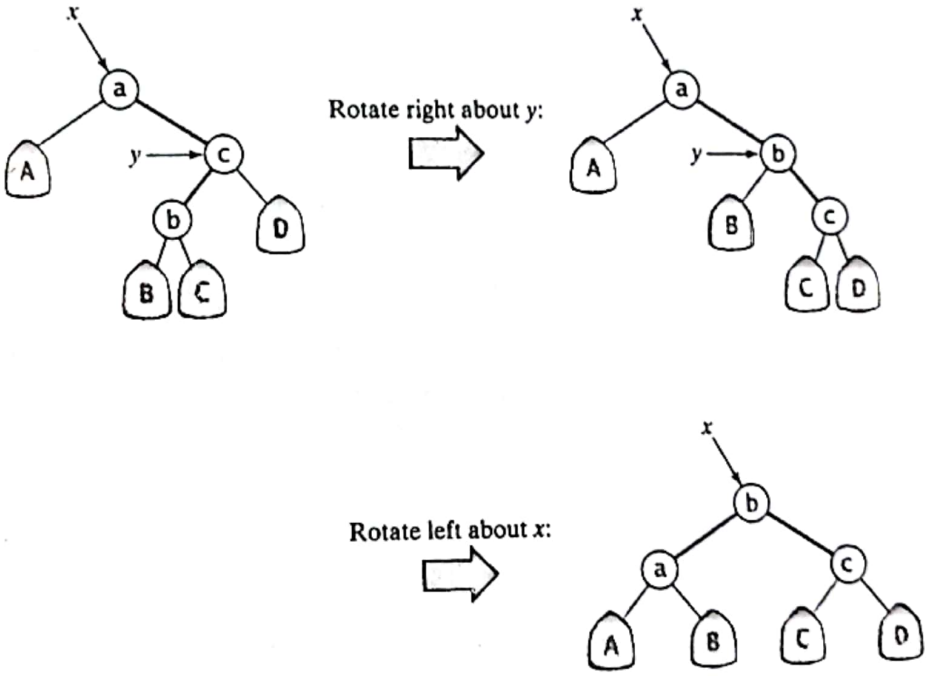


FIGURE 13.15 The AVL Double Rotation algorithm.

The decision on whether to make a simple rotation or a compound rotation is based upon which pattern is causing the imbalance. The four possible patterns are shown in Figure 13.16 on page 390.

Balance control is maintained in AVL trees by enforcing the *AVL criterion*: At each node, the heights of the two subtrees cannot differ by more than 1. If we denote those two heights by h_L and h_R, then the AVL criterion states that $| h_L - h_R | < 2$, which means that the difference can be only -1, 0, or 1. Since this criterion is defined entirely in terms of the subtree heights, an AVL tree is also called a *height-balanced tree*.

Scanned with CamScanner

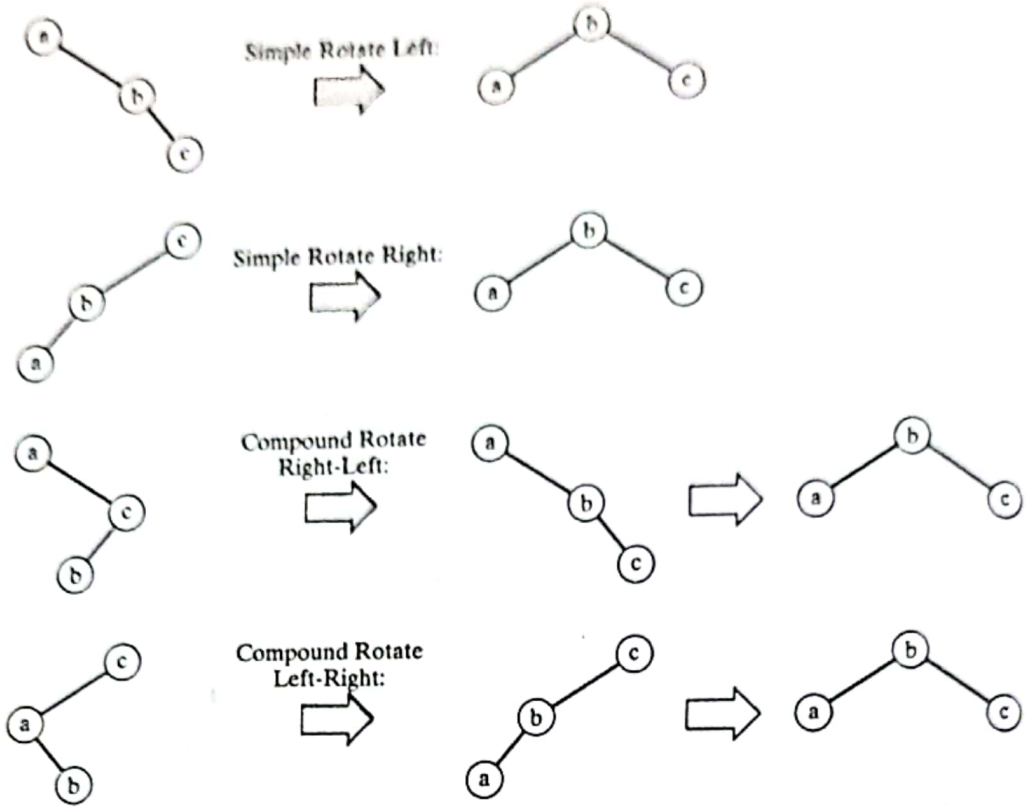


FIGURE 13.16 The four AVL Rotation patterns.

Figure 13.17 shows two binary search trees. The number above each node is the height of the subtree rooted at that node. The tree on the top is not an AVL tree because it has some nodes where the AVL criterion fails. For example, at node (22), the left subtree has height 0 and the right subtree has height 2.

13.6 Implementing AVL Trees

Listing 13.2 presents a Java class for AVL trees:

The class defines a `static` object named `NIL` (line 5) which represents the unique (`final`) empty tree. This uniqueness constraint is enforced by defining the no-argument constructor to be `private`; it is used only once, within the class, to define `NIL`, and cannot be used anywhere else. This constant `NIL` tree object has `key` = 0, `height` = –1, and both child pointers pointing to the object itself. A node is a leaf if and only if both of its children are `NIL`. The `NIL` is used in place of the `null` reference. This use allows AVL trees to be built and processed with no occurrences of the `null` reference, and thus no chance of generating a `NullPointerException`.

The class also defines a `private` three-argument constructor at line 45. It is used exclusively by the two rotate methods. It should not be allowed to be used outside the class, because a client could inadvertently use it to build an instance of the `AVLTree` class that is not an AVL tree.

Scanned with CamScanner

LISTING 13.2: An AVLTree Class

```
public class AVLTree {
  private int key, height;
  private AVLTree left, right;

  public static final AVLTree NIL = new AVLTree();

  public AVLTree(int key){
    this.key = key;
    left = right = NIL;
  }

  public boolean add(int key) {
    int oldSize = size();
    grow(key);
    return size()>oldSize;
  }

  public AVLTree grow(int key) {
    if (this == NIL) return new AVLTree(key);
    if (key == this.key) return this;  // prevent key duplication
    if (key < this.key) left = left.grow(key);
    else right = right.grow(key);
    rebalance();
    height = 1 + Math.max(left.height,right.height);
    return this;
  }

  public int size() {
    if (this == NIL) return 0;
    return 1 + left.size() + right.size();
  }

  public String toString() {
    if (this == NIL) return "";
    return left + " " + key + " " + right;
  }

  private AVLTree() {  // constructs the empty tree
    left = right = this;
    height = -1;
  }

  private AVLTree(int key, AVLTree left, AVLTree right) {
    this.key = key;
    this.left = left;
    this.right = right;
    height = 1 + Math.max(left.height,right.height);
  }
```

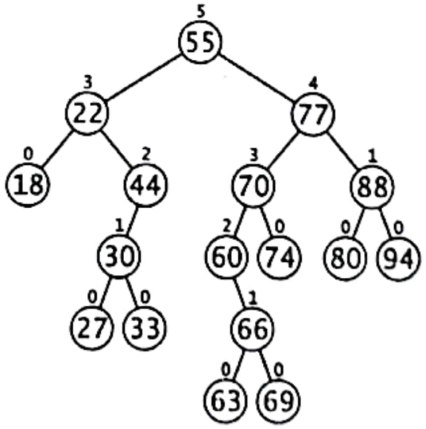
Scanned with CamScanner

```
    private void rebalance() {
      if (right.height > left.height+1) {
        if (right.left.height > right.right.height) right.rotateRight(
        rotateLeft();
      }
      else if (left.height > right.height+1) {
        if (left.right.height > left.left.height) left.rotateLeft();
        rotateRight();
      }
    }

    private void rotateLeft() {
      left = new AVLTree(key,left,right.left);
      key = right.key;
      right = right.right;
    }

    private void rotateRight() {
      right = new AVLTree(key,left.right,right);
      key = left.key;
      left = left.left;
    }
  }
```

This is not an AVL tree:



This is an AVL tree:

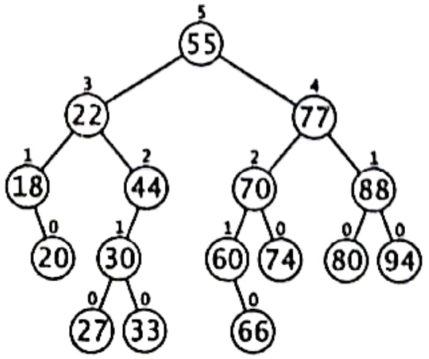


FIGURE 13.17 The AVL criterion.

Scanned with CamScanner

The only public constructor is the one-argument version (line 7) that builds singleton AVL trees. It is used, along with the add() method to build trees, like this:

```
AVLTree tree = new AVLTree(20);
tree.add(30);
tree.add(40);
tree.add(50);
```

The add() method returns true if it succeeds in adding the given key. That will always happen unless the key is already in the table. The method calls the recursive grow() method to do its work.

The grow() method uses the BST property to locate where to insert the new key. This is the same algorithm that the BST ADT uses. Since only AVL trees that already have at least one element can call add(), we know that the condition (this==NIL) at line 19 will be true only on recursive calls, in which case the new subtree will be attached either to a left or right field (at line 21 or line 22). After that, the subtree is rebalanced, if necessary, and its height field is recomputed.

The other two public methods, size() and toString(), are also recursive.

The rebalance() method implements the algorithms described in Section 13.5. The four cases described on page 390 are handled by the four if statements in this method. For example, if the condition (right.height>left.height+1) at line 51 is true, then the height of the right subtree is at least 2 more than that of the left subtree, so rotateLeft() is called to rebalance. Whether a simple or a compound rotation is performed depends upon the condition (right.left.height>right.right.height) at line 52. This is illustrated in Figure 13.18. When the key 35 is inserted, the subtree rooted at (33) becomes imbalanced: Its left subtree has height 1, while its right subtree has height 3. Moreover, this is a "dog leg" imbalance: The right subtree of (33) is the larger (right.height>left.height+1), but the left subtree of its child (44) is the larger (right.left.height>right.right.height). So the call right.rotateRight() at line 52 executes before the call rotateLeft() at line 53.

The rotateLeft() and rotateRight() methods are short and efficient. To avoid losing the current node (located by the immutable "this" reference), the contents of that node's child is copied into it, after reproducing that node's contents in a new AVLTree object. The third step deletes the only reference to that child node, which results in it being "garbage collected" by the run-time system.

For example, consider the final rotateLeft() action, pivoting about node (40), as shown in Figure 13.18. Figure 13.19 illustrates the execution of lines 62–64 of Listing 13.2. Line 62 invokes the private three-argument constructor (defined at line 43) to create a subtree rooted at a duplicate copy of node (33) using the existing subtrees A and B as its left and right subtrees. This new subtree is then used to replace the existing left subtree A as the subtree of the old node (33) referenced by "this". Line 63 copies the key 40 from the "this.right" node to the "this" node. Line 64 slides the subtree rooted at (44) up to becoming the "this.right" node, replacing the redundant node (40) there. Since this.right was the only reference to that node, it will now be deleted by the system's "garbage collector." So there is no net change in the number of nodes. we have

```
62      left = new AVLTree(key,left,right.left);
63      key = right.key;
64      right = right.right;
```

Scanned with CamScanner

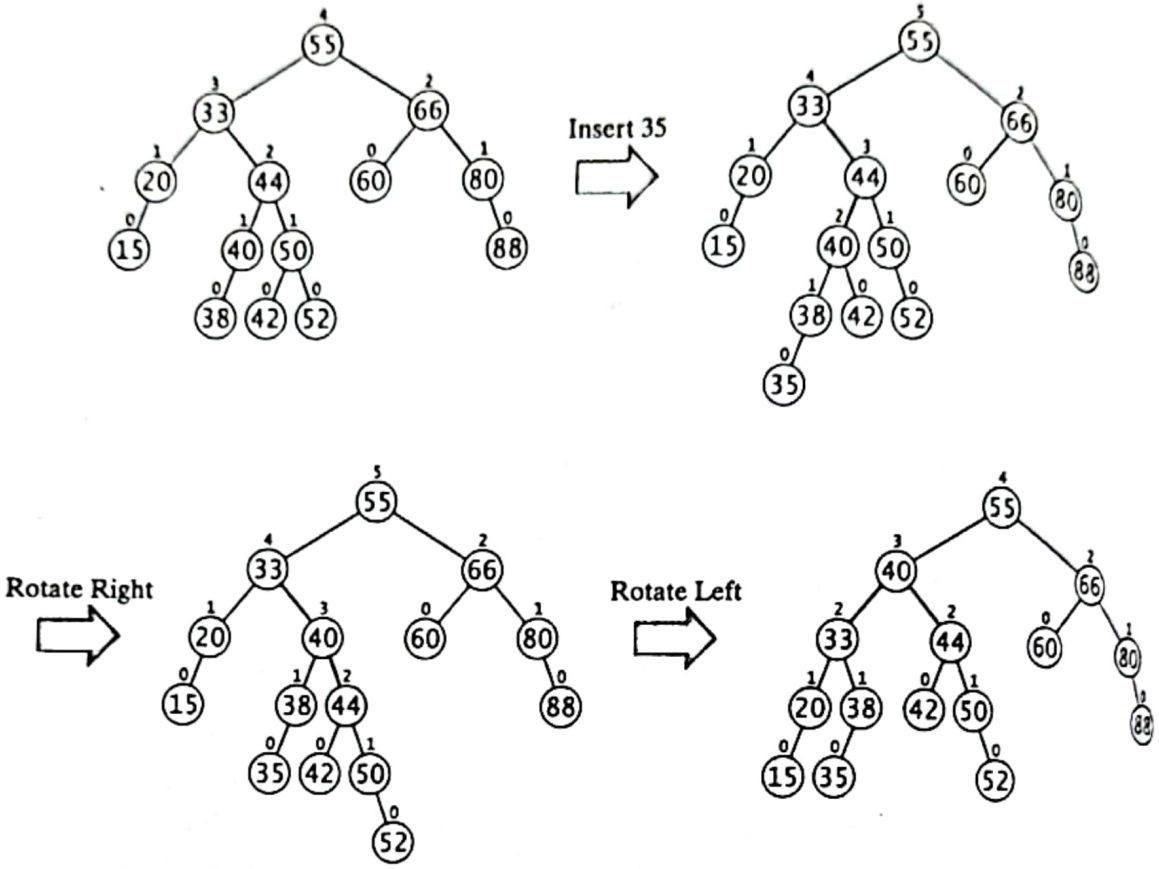


FIGURE 13.18 AVL Insert with rotations.

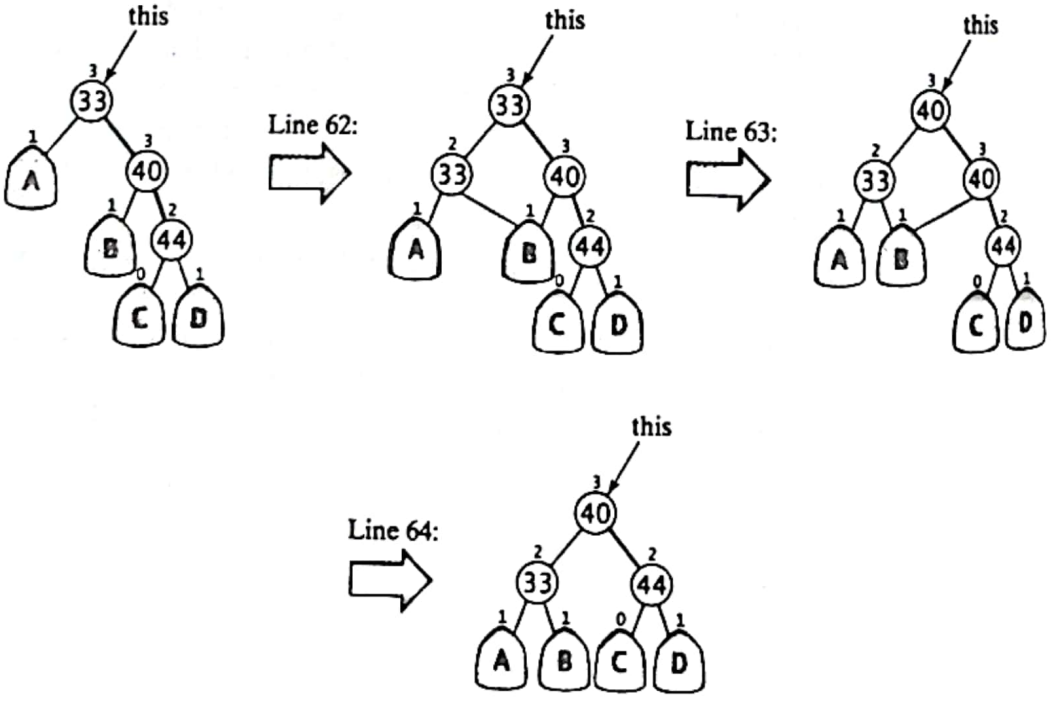


FIGURE 13.19 AVL rotation.

Scanned with CamScanner

Note that the height of the new node created by the three-argument constructor at line 62 has its height set automatically by that constructor at line 47. And the height of the "this" node is then reset by the grow() method at line 24:

```
height = 1 + Math.max(left.height,right.height);
```

13.7 Fibonacci Trees

Since AVL trees are balanced binary search trees, we know that they will be at least as efficient as unconstrained binary search trees:

$$B(n) = \Theta(\lg n)$$
$$A(n) = \Theta(\lg n)$$
$$W(n) = \Theta(n)$$

But the whole point of balancing them is to reduce that worst-case time. How much better is this improvement?

The answer comes from a special kind of AVL tree, called a *Fibonacci tree*. These are built using the same recursive idea that generates the Fibonacci numbers. (See Appendix C.) We shall see that these trees are the worst-case AVL trees and that their analysis proves that general AVL trees perform in logarithmic time, even in the worst case:

$$W(n) = \Theta(\lg n)$$

To understand the definition of the Fibonacci trees, imagine trying to build worst-case AVL trees. That means that, for each height h, we want to find the binary tree structures that have the fewest elements and still could be AVL trees.

At height $h = 0$, there is no choice: The singleton tree is the only possibility. Call it T_0.

At height $h = 1$, there are three binary trees, all of which could be AVL trees. (See Figure 13.20.) Two of these have size $n = 2$, and one has size $n = 3$. Since we want the ones with the smallest sizes, we have a choice of two trees. They are mirror opposites, so it doesn't matter which one we choose. We therefore choose the first one and define it as the Fibonacci tree of height $h = 1$, and call it T_1.

There are 21 binary trees of height $h = 2$. (See Exercise 13.6 on page 411.) Among them all but six be AVL trees. These 15 trees have size n in the range $4 \le n \le 7$. So the minimal AVL trees of height $h = 2$ have size $n = 4$. The four of those are shown in Figure 13.21 on page 396. We could define any one of these to be "the" Fibonacci tree of height $h = 2$. We choose the first one and call it T_2, as shown in Figure 13.22 on page 396.

We could define any one of the trees in Figure 13.21 as the Fibonacci tree of height $h = 2$. We choose the first one and call it T_2.

Note that T_2 can be built by attaching T_1 and T_0 to a root node as its left and right subtrees. Similarly, we define T_3 to be the binary tree obtained by attaching T_2 and T_1 to a root node as left and right subtrees.

FIGURE 13.20 Binary trees of height 1.

Scanned with CamScanner