CHAPTER 26

AVL Trees

Objectives

- To know what an AVL tree is (§26.1).
- To understand how to rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§26.2).
- To design the `AVLTree` class by extending the `BST` class (§26.3).
- To insert elements into an AVL tree (§26.4).
- To implement tree rebalancing (§26.5).
- To delete elements from an AVL tree (§26.6).
- To implement the `AVLTree` class (§26.7).
- To test the `AVLTree` class (§26.8).
- To analyze the complexity of search, insertion, and deletion operations in AVL trees (§26.9).

26.1 Introduction

AVL Tree is a balanced binary search tree.

Key Point

perfectly balanced tree
well-balanced tree

Chapter 25 introduced binary search trees. The search, insertion, and deletion times for a binary tree depend on the height of the tree. In the worst case, the height is $O(n)$. If a tree is *perfectly balanced*—that is, a complete binary tree—its height is $\log n$. Can we maintain a perfectly balanced tree? Yes, but doing so will be costly. The compromise is to maintain a *well-balanced tree*—that is, the heights of every node's two subtrees are about the same. This chapter introduces AVL trees. Web Chapters 40 and 41 will introduce 2–4 trees and red–black trees.

AVL tree
$O(\log n)$

AVL trees are well balanced. AVL trees were invented in 1962 by two Russian computer scientists, G. M. Adelson-Velsky and E. M. Landis (hence the name *AVL*). In an AVL tree, the difference between the heights of every node's two subtrees is **0** or **1**. It can be shown that the maximum height of an AVL tree is $O(\log n)$.

balance factor
balanced
left-heavy
right-heavy

The process for inserting or deleting an element in an AVL tree is the same as in a binary search tree, except that you may have to rebalance the tree after an insertion or deletion operation. The *balance factor* of a node is the height of its right subtree minus the height of its left subtree. For example, the balance factor for the node 87 in Figure 26.1a is **0**, for the node 67 is **1**, and for the node 55 is **−1**. A node is said to be *balanced* if its balance factor is **−1**, **0**, or **1**. A node is considered *left-heavy* if its balance factor is **−1** or less, and *right-heavy* if its balance factor is **+1** or greater.

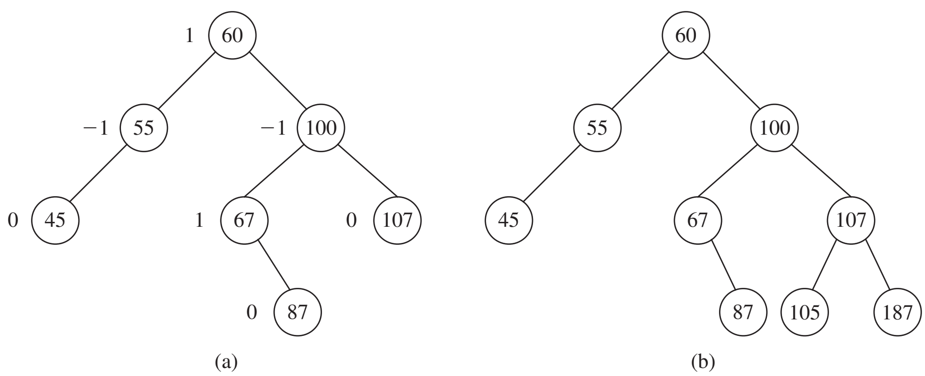


FIGURE 26.1 A balance factor determines whether a node is balanced.

AVL tree animation on Companion Website

Pedagogical Note
For an interactive GUI demo to see how an AVL tree works, go to liveexample.pearsoncmg.com/dsanimation/AVLTreeeBook.html, as shown in Figure 26.2.

26.2 Rebalancing Trees

After inserting or deleting an element from an AVL tree, if the tree becomes unbalanced, perform a rotation operation to rebalance the tree.

Key Point

rotation

If a node is not balanced after an insertion or deletion operation, you need to rebalance it. The process of rebalancing a node is called *rotation*. There are four possible rotations: LL, RR, LR, and RL.

LL rotation
LL imbalance

LL rotation: An *LL imbalance* occurs at a node **A**, such that **A** has a balance factor of **−2** and a left child **B** with a balance factor of **−1** or **0**, as shown in Figure 26.3a. This type of imbalance can be fixed by performing a single right rotation at **A**, as shown in Figure 26.3b.

RR rotation
RR imbalance

RR rotation: An *RR imbalance* occurs at a node **A**, such that **A** has a balance factor of **+2** and a right child **B** with a balance factor of **+1** or **0**, as shown in Figure 26.4a. This type of imbalance can be fixed by performing a single left rotation at **A**, as shown in Figure 26.4b.

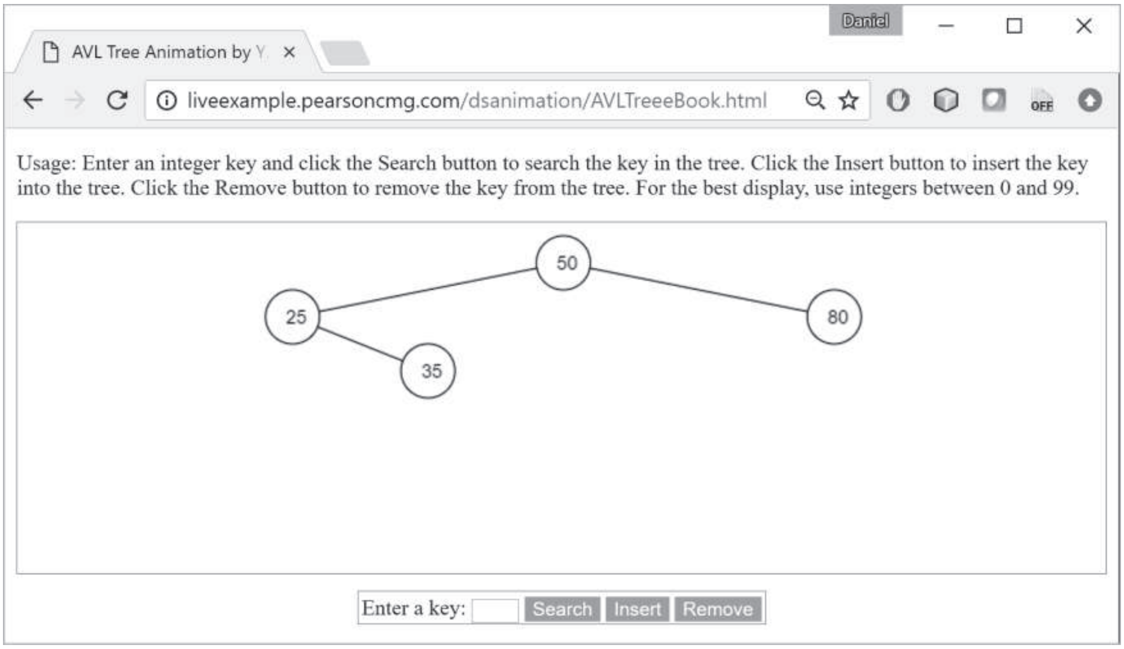


FIGURE 26.2 The animation tool enables you to insert, delete, and search elements. *Source:* Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

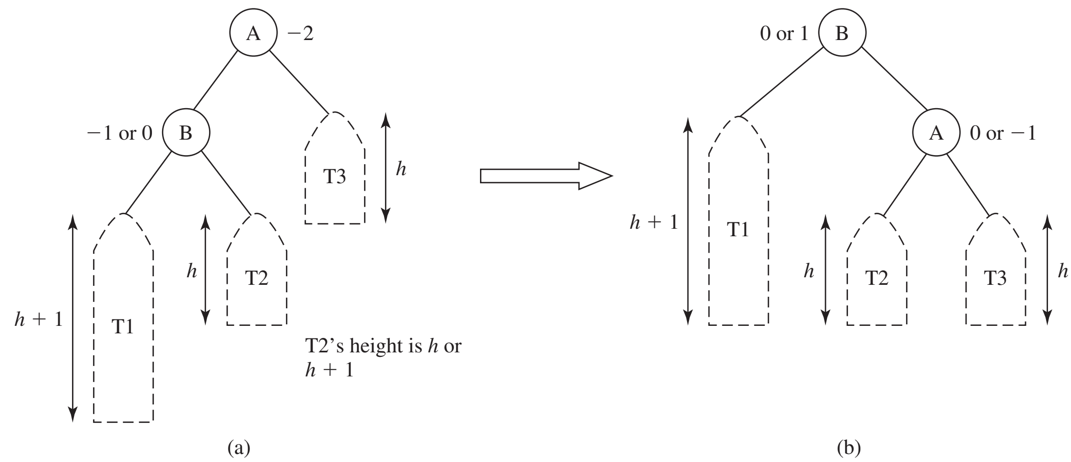


FIGURE 26.3 An LL rotation fixes an LL imbalance.

LR rotation: An *LR imbalance* occurs at a node **A**, such that **A** has a balance factor of **−2** and a left child **B** with a balance factor of **+1**, as shown in Figure 26.5a. Assume **B**'s right child is **C**. This type of imbalance can be fixed by performing a double rotation (first a single left rotation at **B**, then a single right rotation at **A**), as shown in Figure 26.5b.

LR rotation
LR imbalance

RL rotation: An *RL imbalance* occurs at a node **A**, such that **A** has a balance factor of **+2** and a right child **B** with a balance factor of **−1**, as shown in Figure 26.6a. Assume **B**'s left child is **C**. This type of imbalance can be fixed by performing a double rotation (first a single right rotation at **B**, then a single left rotation at **A**), as shown in Figure 26.6b.

RL rotation
RL imbalance

Check Point

26.2.1 What is an AVL tree? Describe the following terms: balance factor, left-heavy, and right-heavy.

26.2.2 Show the balance factor of each node in the trees shown in Figure 26.1.

26.2.3 Describe LL rotation, RR rotation, LR rotation, and RL rotation for an AVL tree.

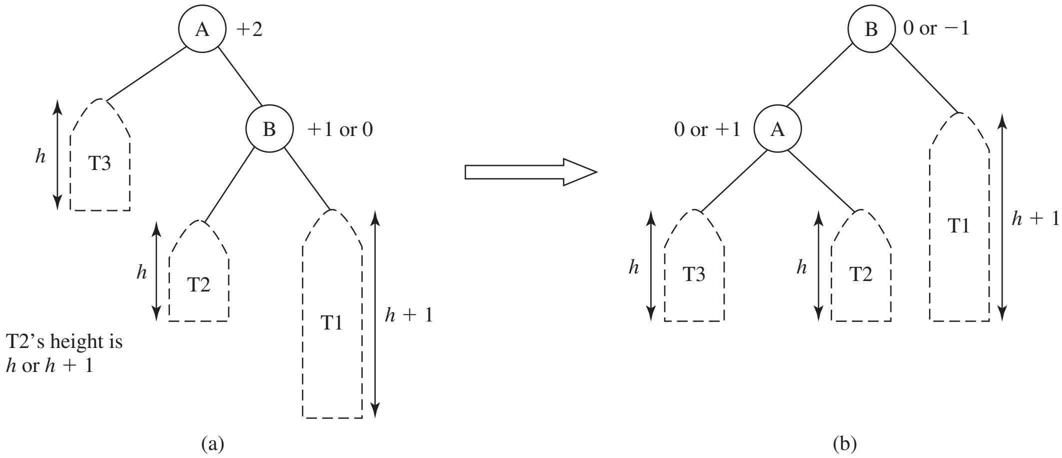


FIGURE 26.4 An RR rotation fixes an RR imbalance.

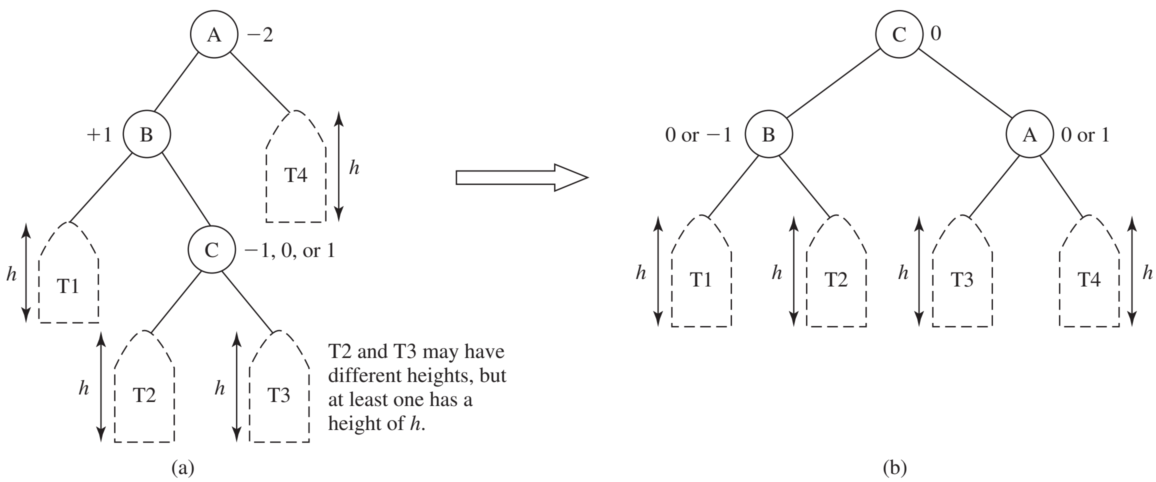


FIGURE 26.5 An LR rotation fixes an LR imbalance.

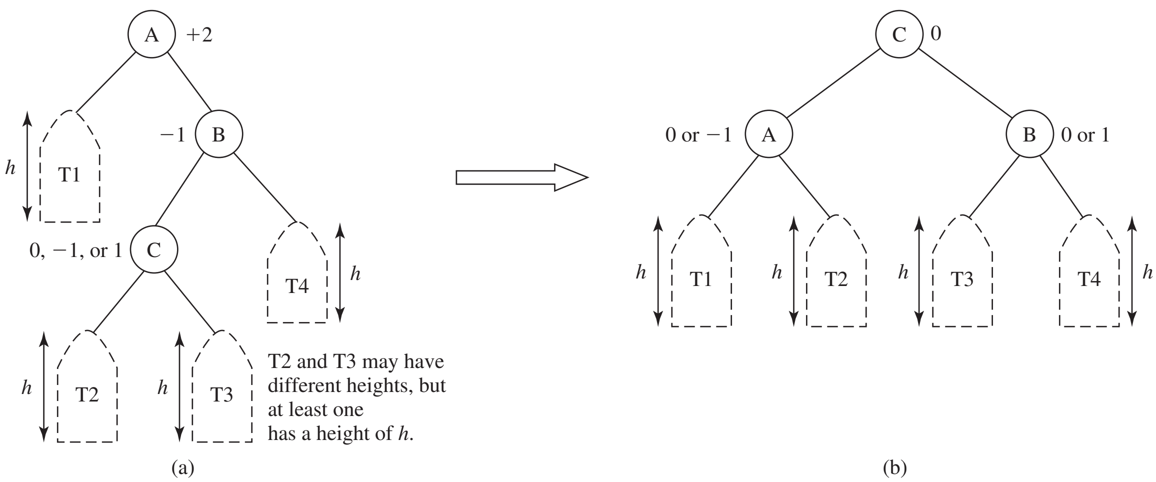


FIGURE 26.6 An RL rotation fixes an RL imbalance.

26.3 Designing Classes for AVL Trees

Since an AVL tree is a binary search tree, **AVLTree** *is designed as a subclass of* **BST**.

Key Point

An AVL tree is a binary tree, so you can define the **AVLTree** class to extend the **BST** class, as shown in Figure 26.7. The **BST** and **TreeNode** classes were defined in Section 25.2.5.

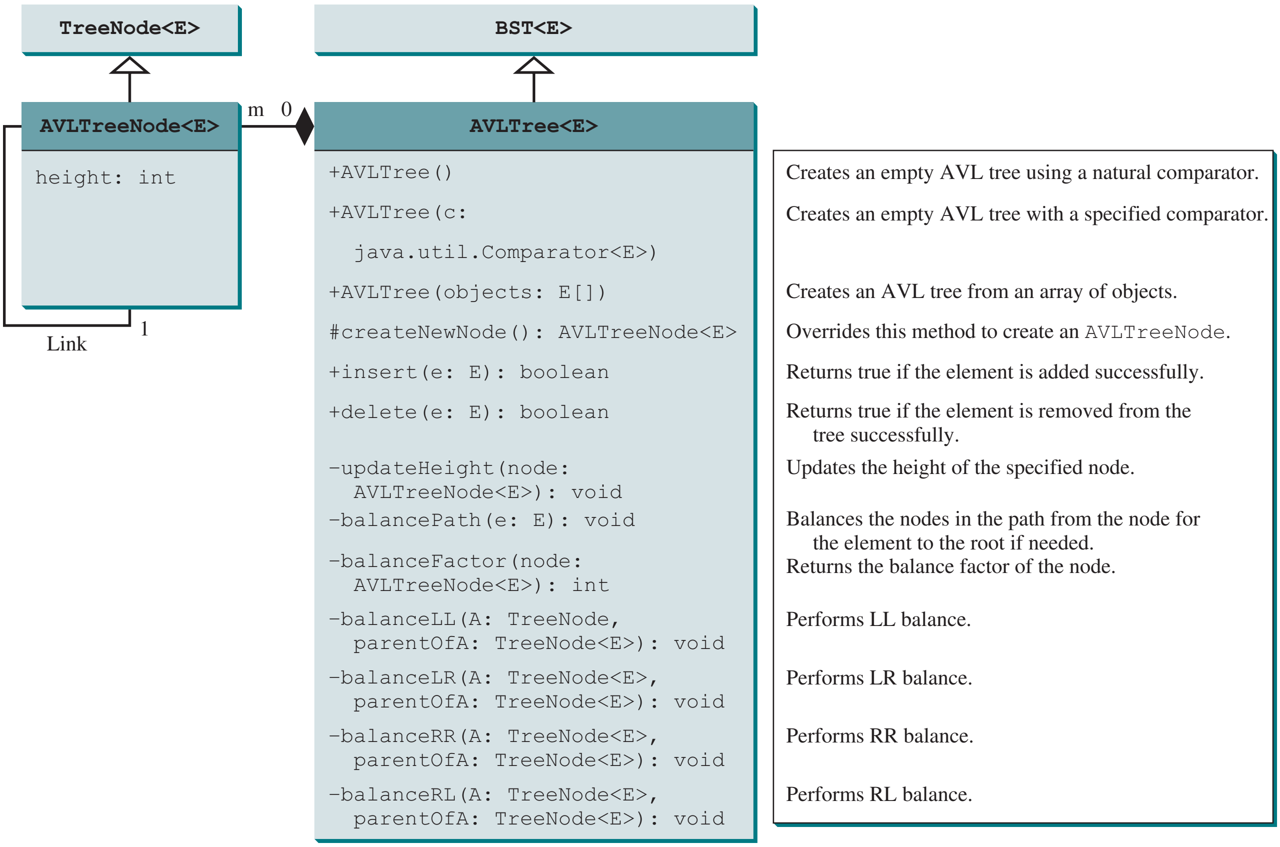


FIGURE 26.7 The **AVLTree** class extends **BST** with new implementations for the **insert** and **delete** methods.

In order to balance the tree, you need to know each node's height. For convenience, store the height of each node in **AVLTreeNode** and define **AVLTreeNode** to be a subclass of **BST.TreeNode**. Note that **TreeNode** is defined as a static inner class in **BST**. **AVLTreeNode** will be defined as a static inner class in **AVLTree**. **TreeNode** contains the data fields **element**, **left**, and **right**, which are inherited by **AVLTreeNode**. Thus, **AVLTreeNode** contains four data fields, as shown in Figure 26.8.

AVLTreeNode

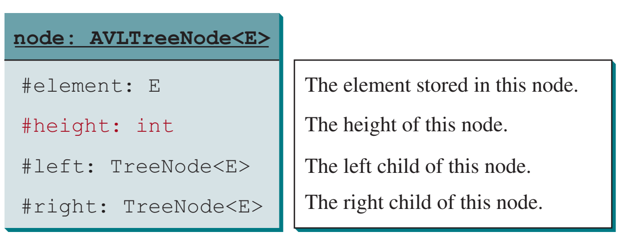


FIGURE 26.8 An **AVLTreeNode** contains the protected data fields **element**, **height**, **left**, and **right**.

In the `BST` class, the `createNewNode()` method creates a `TreeNode` object. This method is overridden in the `AVLTree` class to create an `AVLTreeNode`. Note the return type of the `createNewNode()` method in the `BST` class is `TreeNode`, but the return type of the `createNewNode()` method in the `AVLTree` class is `AVLTreeNode`. This is fine, since `AVLTreeNode` is a subclass of `TreeNode`.

createNewNode()

Searching for an element in an `AVLTree` is the same as searching in a binary search tree, so the `search` method defined in the `BST` class also works for `AVLTree`.

The `insert` and `delete` methods are overridden to insert and delete an element and perform rebalancing operations if necessary to ensure that the tree is balanced.

Check Point

26.3.1 What are the data fields in the `AVLTreeNode` class?

26.3.2 True or false: `AVLTreeNode` is a subclass of `TreeNode`.

26.3.3 True or false: `AVLTree` is a subclass of `BST`.

26.4 Overriding the `insert` Method

Key Point

Inserting an element into an AVL tree is the same as inserting it to a BST, except that the tree may need to be rebalanced.

A new element is always inserted as a leaf node. As a result of adding a new node, the heights of the new leaf node's ancestors may increase. After inserting a new node, check the nodes along the path from the new leaf node up to the root. If an unbalanced node is found, perform an appropriate rotation using the algorithm in Listing 26.1.

LISTING 26.1 Balancing Nodes on a Path

get the path · update node height · get parent node · is balanced? · LL rotation · LR rotation · RR rotation · RL rotation

```
balancePath(E e) {
  Get the path from the node that contains element e to the root,
    as illustrated in Figure 26.9;
  for each node A in the path leading to the root {
    Update the height of A;
    Let parentOfA denote the parent of A,
      which is the next node in the path, or null if A is the root;

    switch (balanceFactor(A)) {
      case -2:  if balanceFactor(A.left) == -1 or 0
                  Perform LL rotation; // See Figure 26.3
                else
                  Perform LR rotation; // See Figure 26.5
                break;
      case +2:  if balanceFactor(A.right) == +1 or 0
                  Perform RR rotation; // See Figure 26.4
                else
                  Perform RL rotation; // See Figure 26.6
    } // End of switch
  } // End of for
} // End of method
```

The algorithm considers each node in the path from the new leaf node to the root. Update the height of the node on the path. If a node is balanced, no action is needed. If a node is not balanced, perform an appropriate rotation.

Check Point

26.4.1 For the AVL tree in Figure 26.1a, show the new AVL tree after adding element `40`. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

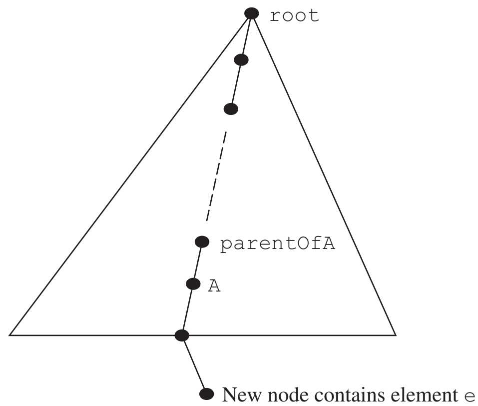


FIGURE 26.9 The nodes along the path from the new leaf node may become unbalanced.

26.4.2 For the AVL tree in Figure 26.1a, show the new AVL tree after adding element **50**. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.4.3 For the AVL tree in Figure 26.1a, show the new AVL tree after adding element **80**. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.4.4 For the AVL tree in Figure 26.1a, show the new AVL tree after adding element **89**. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.5 Implementing Rotations

An unbalanced tree becomes balanced by performing an appropriate rotation operation.

Key Point

Section 26.2, Rebalancing Trees, illustrated how to perform rotations at a node. Listing 26.2 gives the algorithm for the LL rotation, as illustrated in Figure 26.3.

LISTING 26.2 LL Rotation Algorithm

```
balanceLL(TreeNode A, TreeNode parentOfA) {
  Let B be the left child of A.

  if (A is the root)
    Let B be the new root
  else {
    if (A is a left child of parentOfA)
      Let B be a left child of parentOfA;
    else
      Let B be a right child of parentOfA;
  }

  Make T2 the left subtree of A by assigning B.right to A.left;
  Make A the right child of B by assigning A to B.right;
  Update the height of node A and node B;
} // End of method
```

left child of A

reconnect B's parent

move subtrees

adjust height

Note the height of nodes **A** and **B** can be changed, but the heights of other nodes in the tree are not changed. You can implement the RR, LR, and RL rotations in a similar manner.

Check Point

26.5.1 Use Listing 26.2 as a template to describe the algorithms for implementing the RR, LR, and RL rotations.

26.6 Implementing the `delete` Method

Key Point

Deleting an element from an AVL tree is the same as deleting it from a BST, except that the tree may need to be rebalanced.

As discussed in Section 25.3, Deleting Elements from a BST, to delete an element from a binary tree, the algorithm first locates the node that contains the element. Let `current` point to the node that contains the element in the binary tree and `parent` point to the parent of the `current` node. The `current` node may be a left child or a right child of the `parent` node. Two cases arise when deleting an element.

***Case 1*:** The `current` node does not have a left child, as shown in Figure 25.10a. To delete the `current` node, simply connect the `parent` node with the right child of the `current` node, as shown in Figure 25.10b.

The height of the nodes along the path from the `parent` node up to the `root` may have decreased. To ensure that the tree is balanced, invoke

```
balancePath(parent.element); // Defined in Listing 26.1
```

***Case 2*:** The `current` node has a left child. Let `rightMost` point to the node that contains the largest element in the left subtree of the `current` node and `parentOfRightMost` point to the parent node of the `rightMost` node, as shown in Figure 25.12a. The `rightMost` node cannot have a right child, but may have a left child. Replace the element value in the `current` node with the one in the `rightMost` node, connect the `parentOfRightMost` node with the left child of the `rightMost` node, and delete the `rightMost` node, as shown in Figure 25.12b.

The height of the nodes along the path from `parentOfRightMost` up to the root may have decreased. To ensure the tree is balanced, invoke

```
balancePath(parentOfRightMost); // Defined in Listing 26.1
```

Check Point

26.6.1 For the AVL tree in Figure 26.1a, show the new AVL tree after deleting element `107`. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.6.2 For the AVL tree in Figure 26.1a, show the new AVL tree after deleting element `60`. What rotation do you perform in order to rebalance the tree? Which node was unbalanced?

26.6.3 For the AVL tree in Figure 26.1a, show the new AVL tree after deleting element `55`. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

26.6.4 For the AVL tree in Figure 26.1b, show the new AVL tree after deleting elements `67` and `87`. What rotation did you perform in order to rebalance the tree? Which node was unbalanced?

26.7 The `AVLTree` Class

Key Point

The `AVLTree` class extends the `BST` class to override the `insert` and `delete` methods to rebalance the tree if necessary.

Listing 26.3 gives the complete source code for the `AVLTree` class.

LISTING 26.3 AVLTree.java

extends BST
no-arg constructor

```
public class AVLTree<E> extends BST<E> {
  /** Create an empty AVL tree using a natural comparator*/
  public AVLTree() { // super() is implicitly called
```

```java
  }

  /** Create a BST with a specified comparator */
  public AVLTree(java.util.Comparator<E> c) {
    super(c);
  }

  /** Create an AVL tree from an array of objects */
  public AVLTree(E[] objects) {
    super(objects);
  }

  @Override /** Override createNewNode to create an AVLTreeNode */
  protected AVLTreeNode<E> createNewNode(E e) {
    return new AVLTreeNode<E>(e);
  }

  @Override /** Insert an element and rebalance if necessary */
  public boolean insert(E e) {
    boolean successful = super.insert(e);
    if (!successful)
      return false; // e is already in the tree
    else {
      balancePath(e); // Balance from e to the root if necessary
    }

    return true; // e is inserted
  }

  /** Update the height of a specified node */
  private void updateHeight(AVLTreeNode<E> node) {
    if (node.left == null && node.right == null) // node is a leaf
      node.height = 0;
    else if (node.left == null) // node has no left subtree
      node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;
    else if (node.right == null) // node has no right subtree
      node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;
    else
      node.height = 1 +
        Math.max(((AVLTreeNode<E>)(node.right)).height,
        ((AVLTreeNode<E>)(node.left)).height);
  }

  /** Balance the nodes in the path from the specified
   * node to the root if necessary
   */
  private void balancePath(E e) {
    java.util.ArrayList<TreeNode<E>> path = path(e);
    for (int i = path.size() - 1; i >= 0; i--) {
      AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));
      updateHeight(A);
      AVLTreeNode<E> parentOfA = (A == root) ? null :
        (AVLTreeNode<E>)(path.get(i - 1));

      switch (balanceFactor(A)) {
        case -2:
          if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) {
            balanceLL(A, parentOfA); // Perform LL rotation
          }
          else {
```

constructor with a comparator

constructor for arrays

create AVL tree node

override insert

balance tree

update node height

balance nodes

get path

consider a node

update height

get parentOfA

left-heavy

LL rotation

LR rotation
right-heavy
RR rotation
RL rotation
get balance factor
LL rotation
update height
LR rotation

```
            balanceLR(A, parentOfA); // Perform LR rotation
          }
          break;
        case +2:
          if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) {
            balanceRR(A, parentOfA); // Perform RR rotation
          }
          else {
            balanceRL(A, parentOfA); // Perform RL rotation
          }
      }
    }
  }

  /** Return the balance factor of the node */
  private int balanceFactor(AVLTreeNode<E> node) {
    if (node.right == null) // node has no right subtree
      return -node.height;
    else if (node.left == null) // node has no left subtree
      return +node.height;
    else
      return ((AVLTreeNode<E>)node.right).height -
        ((AVLTreeNode<E>)node.left).height;
  }

  /** Balance LL (see Figure 26.3) */
  private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.left; // A is left-heavy and B is left-heavy

    if (A == root) {
      root = B;
    }
    else {
      if (parentOfA.left == A) {
        parentOfA.left = B;
      }
      else {
        parentOfA.right = B;
      }
    }

    A.left = B.right; // Make T2 the left subtree of A
    B.right = A; // Make A the left child of B
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
  }

  /** Balance LR (see Figure 26.5) */
  private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.left; // A is left-heavy
    TreeNode<E> C = B.right; // B is right-heavy

    if (A == root) {
      root = C;
    }
    else {
      if (parentOfA.left == A) {
        parentOfA.left = C;
      }
      else {
```

```
        parentOfA.right = C;
      }
    }

    A.left = C.right; // Make T3 the left subtree of A
    B.right = C.left; // Make T2 the right subtree of B
    C.left = B;
    C.right = A;

    // Adjust heights
```

update height

```
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
    updateHeight((AVLTreeNode<E>)C);
  }

  /** Balance RR (see Figure 26.4) */
```

RR rotation

```
  private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy

    if (A == root) {
      root = B;
    }
    else {
      if (parentOfA.left == A) {
        parentOfA.left = B;
      }
      else {
        parentOfA.right = B;
      }
    }

    A.right = B.left; // Make T2 the right subtree of A
    B.left = A;
```

update height

```
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
  }

  /** Balance RL (see Figure 26.6) */
```

RL rotation

```
  private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {
    TreeNode<E> B = A.right; // A is right-heavy
    TreeNode<E> C = B.left; // B is left-heavy

    if (A == root) {
      root = C;
    }
    else {
      if (parentOfA.left == A) {
        parentOfA.left = C;
      }
      else {
        parentOfA.right = C;
      }
    }

    A.right = C.left; // Make T2 the right subtree of A
    B.left = C.right; // Make T3 the left subtree of B
    C.left = A;
    C.right = B;

    // Adjust heights
```

update height

```
    updateHeight((AVLTreeNode<E>)A);
    updateHeight((AVLTreeNode<E>)B);
    updateHeight((AVLTreeNode<E>)C);
  }

  @Override /** Delete an element from the binary tree.
   * Return true if the element is deleted successfully
   * Return false if the element is not in the tree */
```

override delete

```
  public boolean delete(E element) {
    if (root == null)
      return false; // Element is not in the tree

    // Locate the node to be deleted and also locate its parent node
    TreeNode<E> parent = null;
    TreeNode<E> current = root;
    while (current != null) {
      if (c.compare(element, current.element) < 0) {
        parent = current;
        current = current.left;
      }
      else if (c.compare(element, current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        break; // Element is in the tree pointed by current
    }

    if (current == null)
      return false; // Element is not in the tree

    // Case 1: current has no left children (See Figure 23.6)
    if (current.left == null) {
      // Connect the parent with the right child of the current node
      if (parent == null) {
        root = current.right;
      }
      else {
        if (c.compare(element, parent.element) < 0)
          parent.left = current.right;
        else
          parent.right = current.right;

        // Balance the tree if necessary
```

balance nodes

```
        balancePath(parent.element);
      }
    }
    else {
      // Case 2: The current node has a left child
      // Locate the rightmost node in the left subtree of
      // the current node and also its parent
      TreeNode<E> parentOfRightMost = current;
      TreeNode<E> rightMost = current.left;

      while (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.right; // Keep going to the right
      }

     // Replace the element in current by the element in rightMost
```

```
244         current.element = rightMost.element;
245
246         // Eliminate rightmost node
247         if (parentOfRightMost.right == rightMost)
248           parentOfRightMost.right = rightMost.left;
249         else
250           // Special case: parentOfRightMost is current
251           parentOfRightMost.left = rightMost.left;
252
253         // Balance the tree if necessary
254         balancePath(parentOfRightMost.element);
255       }
256
257       size--;
258       return true; // Element inserted
259     }
260
261     /** AVLTreeNode is TreeNode plus height */
262     protected static class AVLTreeNode<E> extends BST.TreeNode<E> {
263       protected int height = 0; // New data field
264
265       public AVLTreeNode(E o) {
266         super(o);
267       }
268     }
269 }
```

balance nodes

inner AVLTreeNode class

node height

The **AVLTree** class extends **BST**. Like the **BST** class, the **AVLTree** class has a no-arg constructor that constructs an empty **AVLTree** (lines 3 and 4) using a natural comparator, a constructor that constructs an empty AVLTree (lines 7–9) with a specified comparator, and a constructor that creates an initial **AVLTree** from an array of elements (lines 12–14).

constructors

The **createNewNode()** method defined in the **BST** class creates a **TreeNode**. This method is overridden to return an **AVLTreeNode** (lines 17–19).

The **insert** method in **AVLTree** is overridden in lines 22–31. The method first invokes the **insert** method in **BST**, then invokes **balancePath(e)** (line 27) to ensure that the tree is balanced.

insert

The **balancePath** method first gets the nodes on the path from the node that contains element **e** to the root (line 51). For each node in the path, update its height (line 58), check its balance factor (line 58), and perform appropriate rotations if necessary (lines 59–73).

balancePath

Four methods for performing rotations are defined in lines 90–187. Each method is invoked with two **TreeNode** arguments—**A** and **parentOfA**—to perform an appropriate rotation at node **A**. How each rotation is performed is illustrated in Figures 26.3–26.6. After the rotation, the heights of nodes **A**, **B**, and **C** are updated (lines 107, 134, 157, and 184).

rotations

The **delete** method in **AVLTree** is overridden in lines 192–259. The method is the same as the one implemented in the **BST** class, except that you have to rebalance the nodes after deletion in two cases (lines 228, 254).

delete

Check Point

26.7.1 Why is the **createNewNode** method defined protected? When is it invoked?

26.7.2 When is the **updateHeight** method invoked? When is the **balanceFactor** method invoked? When is the **balancePath** method invoked? Will the program work if you replace the break in line 61 in the **AVLTree** class with a return and add a return at line 69?

26.7.3 What are data fields in the **AVLTree** class?

26.7.4 In the **insert** and **delete** methods, once you have performed a rotation to balance a node in the tree, is it possible there are still unbalanced nodes?

26.8 Testing the AVLTree Class

Key Point

This section gives an example of using the **AVLTree** *class.*

Listing 26.4 gives a test program. The program creates an **AVLTree** initialized with an array of the integers **25**, **20**, and **5** (lines 4 and 5), inserts elements in lines 9–18, and deletes elements in lines 22–28. Since **AVLTree** is a subclass of **BST** and the elements in a **BST** are iterable, the program uses a foreach loop to traverse all the elements in lines 33–35.

Listing 26.4 TestAVLTree.java

create an AVLTree
insert 34
insert 50
insert 30
insert 10
delete 34
delete 30
delete 50
delete 5
foreach loop

```
1  public class TestAVLTree {
2    public static void main(String[] args) {
3      // Create an AVL tree
4      AVLTree<Integer> tree = new AVLTree<Integer>(new Integer[]{25,
5        20, 5});
6      System.out.print("After inserting 25, 20, 5:");
7      printTree(tree);
8
9      tree.insert(34);
10     tree.insert(50);
11     System.out.print("\nAfter inserting 34, 50:");
12     printTree(tree);
13
14     tree.insert(30);
15     System.out.print("\nAfter inserting 30");
16     printTree(tree);
17
18     tree.insert(10);
19     System.out.print("\nAfter inserting 10");
20     printTree(tree);
21
22     tree.delete(34);
23     tree.delete(30);
24     tree.delete(50);
25     System.out.print("\nAfter removing 34, 30, 50:");
26     printTree(tree);
27
28     tree.delete(5);
29     System.out.print("\nAfter removing 5:");
30     printTree(tree);
31
32     System.out.print("\nTraverse the elements in the tree: ");
33     for (int e: tree) {
34       System.out.print(e + " ");
35     }
36   }
37
38   public static void printTree(BST tree) {
39     // Traverse tree
40     System.out.print("\nInorder (sorted): ");
41     tree.inorder();
42     System.out.print("\nPostorder: ");
43     tree.postorder();
44     System.out.print("\nPreorder: ");
45     tree.preorder();
46     System.out.print("\nThe number of nodes is " + tree.getSize());
```

```
      System.out.println();
    }
  }
```

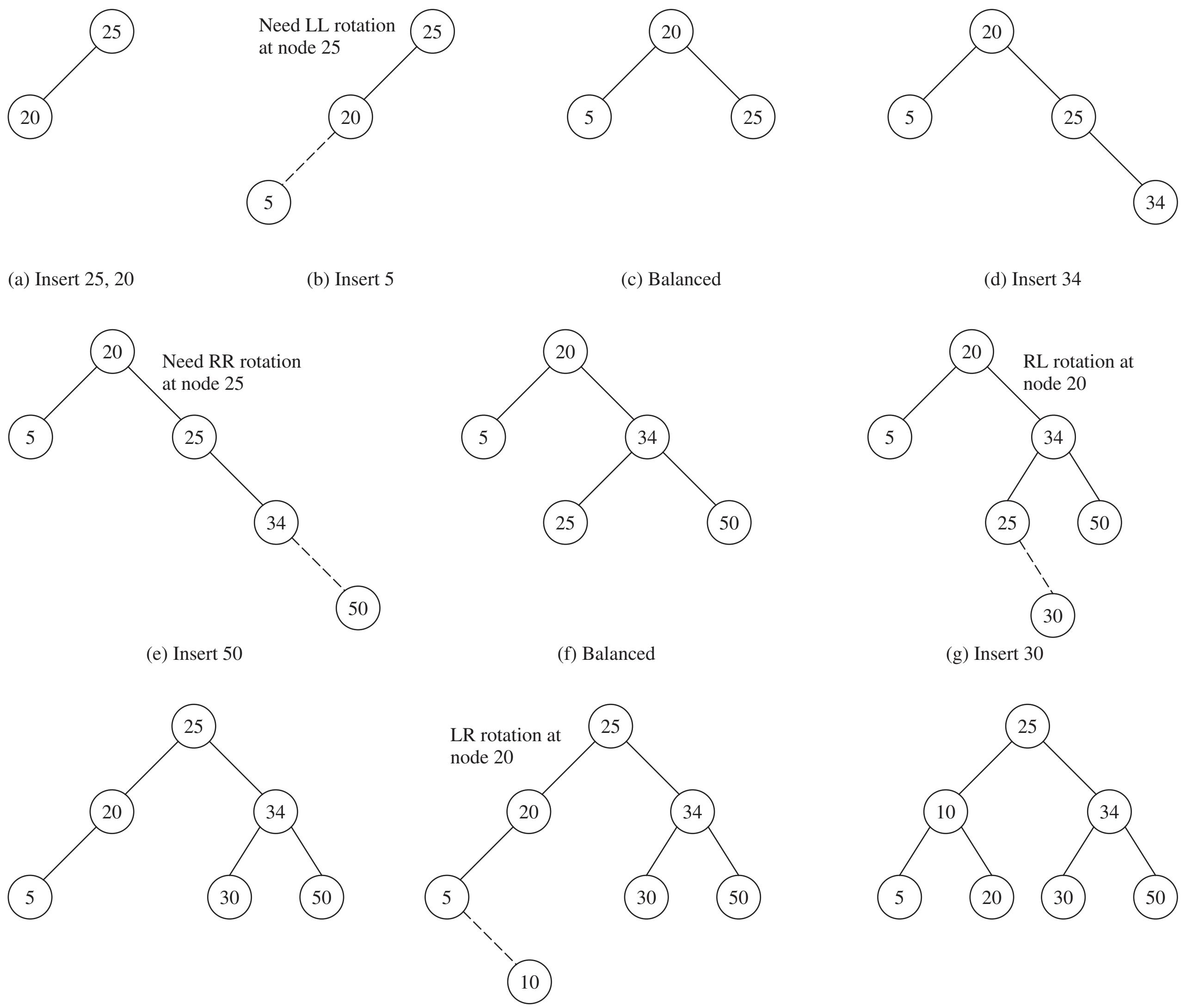
```
After inserting 25, 20, 5:
Inorder (sorted): 5 20 25
Postorder: 5 25 20
Preorder: 20 5 25
The number of nodes is 3

After inserting 34, 50:
Inorder (sorted): 5 20 25 34 50
Postorder: 5 25 50 34 20
Preorder: 20 5 34 25 50
The number of nodes is 5

After inserting 30
Inorder (sorted): 5 20 25 30 34 50
Postorder: 5 20 30 50 34 25
Preorder: 25 20 5 34 30 50
The number of nodes is 6

After inserting 10
Inorder (sorted): 5 10 20 25 30 34 50
Postorder: 5 20 10 30 50 34 25
Preorder: 25 10 5 20 34 30 50
The number of nodes is 7

After removing 34, 30, 50:
Inorder (sorted): 5 10 20 25
Postorder: 5 20 25 10
Preorder: 10 5 25 20
The number of nodes is 4

After removing 5:
Inorder (sorted): 10 20 25
Postorder: 10 25 20
Preorder: 20 10 25
The number of nodes is 3
Traverse the elements in the tree: 10 20 25
```

Figure 26.10 shows how the tree evolves as elements are added to the tree. After **25** and **20** are added, the tree is as shown in Figure 26.10a. **5** is inserted as a left child of **20**, as shown in Figure 26.10b. The tree is not balanced. It is left-heavy at node **25**. Perform an LL rotation to result in an AVL tree as shown in Figure 26.10c.

After inserting **34**, the tree is as shown in Figure 26.10d. After inserting **50**, the tree is as shown in Figure 26.10e. The tree is not balanced. It is right-heavy at node **25**. Perform an RR rotation to result in an AVL tree as shown in Figure 26.10f.

After inserting **30**, the tree is as shown in Figure 26.10g. The tree is not balanced. Perform an RL rotation to result in an AVL tree as shown in Figure 26.10h.

After inserting **10**, the tree is as shown in Figure 26.10i. The tree is not balanced. Perform an LR rotation to result in an AVL tree as shown in Figure 26.10j.



(a) Insert 25, 20 (b) Insert 5 (c) Balanced (d) Insert 34

(e) Insert 50 (f) Balanced (g) Insert 30

(h) Balanced (i) Insert 10 (j) Balanced

FIGURE 26.10 The tree evolves as new elements are inserted.

Figure 26.11a shows how the tree evolves as elements are deleted. After deleting **34**, **30**, and **50**, the tree is as shown in Figure 26.11b. The tree is not balanced. Perform an LL rotation to result in an AVL tree as shown in Figure 26.11c.

After deleting **5**, the tree is as shown in Figure 26.11d. The tree is not balanced. Perform an RL rotation to result in an AVL tree as shown in Figure 26.11e.

Check Point

26.8.1 Show the change of an AVL tree when inserting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, **6** into the tree, in this order.

26.8.2 For the tree built in the preceding question, show its change after **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, **6** are deleted from the tree in this order.

26.8.3 Can you traverse the elements in an AVL tree using a foreach loop?

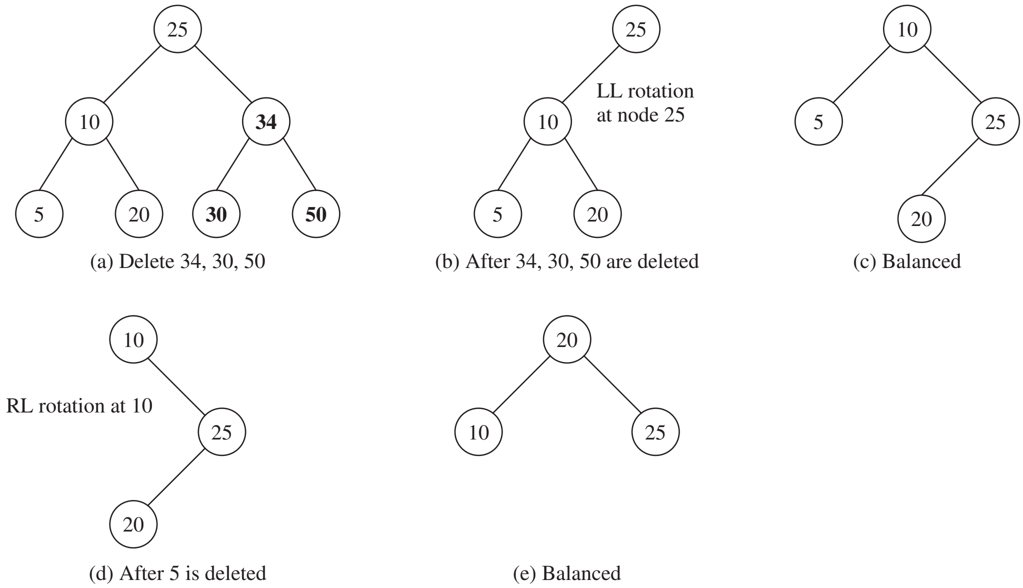


FIGURE 26.11 The tree evolves as elements are deleted from the tree.

26.9 AVL Tree Time Complexity Analysis

Key Point

Since the height of an AVL tree is O(log n), the time complexity of the **`search`**, **`insert`**, *and* **`delete`** *methods in* **`AVLTree`** *is O(log n).*

tree height

The time complexity of the **`search`**, **`insert`**, and **`delete`** methods in **`AVLTree`** depends on the height of the tree. We can prove that the height of the tree is $O(\log n)$.

Let $G(h)$ denote the minimum number of nodes in an AVL tree with height h. For the definition of the height of a binary tree, see Section 25.2. Obviously, $G(0)$ is 1 and $G(1)$ is 2. The minimum number of nodes in an AVL tree with height $h \geq 2$ must have two minimum subtrees: one with height $h - 1$ and the other with height $h - 2$. Thus,

$$G(h) = G(h - 1) + G(h - 2) + 1$$

Recall that a Fibonacci number at index i can be described using the recurrence relation $F(i) = F(i - 1) + F(i - 2)$. Therefore, the function $G(h)$ is essentially the same as $F(i)$. It can be proven that

$$h < 1.4405 \log(n + 2) - 1.3277$$

where n is the number of nodes in the tree. Hence, the height of an AVL tree is $O(\log n)$.

The **`search`**, **`insert`**, and **`delete`** methods involve only the nodes along a path in the tree. The **`updateHeight`** and **`balanceFactor`** methods are executed in a constant time for each node in the path. The **`balancePath`** method is executed in a constant time for a node in the path. Thus, the time complexity for the **`search`**, **`insert`**, and **`delete`** methods is $O(\log n)$.

Check Point

26.9.1 What is the maximum/minimum height for an AVL tree of 3 nodes, 5 nodes, and 7 nodes?

26.9.2 If an AVL tree has a height of 3, what maximum number of nodes can the tree have? What minimum number of nodes can the tree have?

26.9.3 If an AVL tree has a height of 4, what maximum number of nodes can the tree have? What minimum number of nodes can the tree have?

Key Terms

AVL tree 996
balance factor 996
left-heavy 996
LL rotation 996
LR rotation 997
perfectly balanced tree 996
right-heavy 996
RL rotation 997
rotation 996
RR rotation 996
well-balanced tree 996

Chapter Summary

1. An *AVL tree* is a *well-balanced* binary tree. In an AVL tree, the difference between the heights of two subtrees for every node is `0` or `1`.

2. The process for inserting or deleting an element in an AVL tree is the same as in a binary search tree. The difference is that you may have to rebalance the tree after an insertion or deletion operation.

3. Imbalances in the tree caused by insertions and deletions are rebalanced through subtree rotations at the node of the imbalance.

4. The process of rebalancing a node is called a *rotation*. There are four possible rotations: *LL rotation*, *LR rotation*, *RR rotation*, and *RL rotation*.

5. The height of an AVL tree is $O(\log n)$. Therefore, the time complexities for the `search`, `insert`, and `delete` methods are $O(\log n)$.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

***26.1** (*Display AVL tree graphically*) Write a program that displays an AVL tree along with its balance factor for each node.

26.2 (*Compare performance*) Write a test program that randomly generates 500,000 numbers and inserts them into a `BST`, reshuffles the 500,000 numbers and performs a search, and reshuffles the numbers again before deleting them from the tree. Write another test program that does the same thing for an `AVLTree`. Compare the execution times of these two programs.

*****26.3** (*AVL tree animation*) Write a program that animates the AVL tree `insert`, `delete`, and `search` methods, as shown in Figure 26.2.

****26.4** (*Parent reference for `BST`*) Suppose the `TreeNode` class defined in `BST` contains a reference to the node's parent, as shown in Programming Exercise 25.15. Implement the `AVLTree` class to support this change. Write a test program that adds numbers `1`, `2`, . . . , `100` to the tree and displays the paths for all leaf nodes.

****26.5** (*The* k*th smallest element*) You can find the kth smallest element in a BST in $O(n)$ time from an inorder iterator. For an AVL tree, you can find it in $O(\log n)$ time. To achieve this, add a new data field named `size` in `AVLTreeNode` to store the number of nodes in the subtree rooted at this node. Note the size of a node v is one more than the sum of the sizes of its two children. Figure 26.12 shows an AVL tree and the **size** value for each node in the tree.

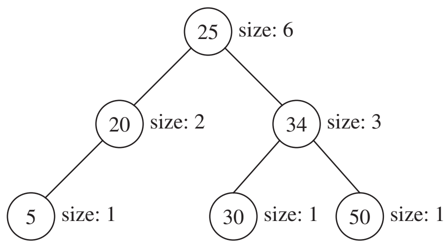


FIGURE 26.12 The **size** data field in **AVLTreeNode** stores the number of nodes in the subtree rooted at the node.

In the **AVLTree** class, add the following method to return the kth smallest element in the tree:

```
public E find(int k)
```

The method returns **null** if **k < 1** or **k > the size of the tree**. This method can be implemented using the recursive method **find(k, root)**, which returns the kth smallest element in the tree with the specified root. Let **A** and **B** be the left and right children of the root, respectively. Assuming the tree is not empty and $k \leq root.size$, **find(k, root)** can be recursively defined as follows:

$$find(k, root) = \begin{cases} root.element, \text{ if } A \text{ is null and } k \text{ is } 1; \\ B.element, \text{ if } A \text{ is null and } k \text{ is } 2; \\ find(k, A), \text{ if } k <= A.size; \\ root.element, \text{ if } k = A.size + 1; \\ find(k - A.size - 1, B), \text{ if } k > A.size + 1; \end{cases}$$

Modify the **insert** and **delete** methods in **AVLTree** to set the correct value for the **size** property in each node. The **insert** and **delete** methods will still be in $O(\log n)$ time. The **find(k)** method can be implemented in $O(\log n)$ time. Therefore, you can find the kth smallest element in an AVL tree in $O(\log n)$ time.

Test your program using the code at https://liveexample.pearsoncmg.com/test/Exercise26_05.txt.

****26.6** (*Closest pair of points*) Section 22.8 introduced an algorithm for finding a closest pair of points in O(nlogn) time using a divide-and-conquer approach. The algorithm was implemented using recursion with a lot of overhead. Using an AVL tree, you can solve the same problem in O(nlogn) time. Implement the algorithm using an **AVLTree**.

****26.7** (*Test AVL tree*) Define a new class named **MyBST** that extends the **BST** class with the following method:

```
// Returns true if the tree is an AVL tree
public boolean isAVLTree()
```

Use https://liveexample.pearsoncmg.com/test/Exercise26_07.txt to test your code.