个具有高度 $h-2$。因此，

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

回顾下，在下标为 i 处的斐波那契数字可以使用递推关系 $F(i) = F(i-1) + F(i-2)$。因此，函数 $G(h)$ 实质上和 $F(i)$ 一样。可以证明

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

这里 n 是树中的结点数。因此，一个 AVL 树的高度为 $O(\log n)$。

search、insert 以及 delete 方法只涉及树中沿着一条路径上的结点。updateHeight 和 balanceFactor 方法对于路径上的结点来说执行常量时间。balance Path 方法对于路径上的结点来说也执行常量时间。因此，search、insert 以及 delete 方法的时间复杂度为 $O(\log n)$。

复习题

26.22 对于具有 3 个结点、5 个结点以及 7 个结点的 AVL 树来说，最大 / 最小高度是多少？

26.23 如果一棵 AVL 树高度为 3，该树可以具有的最大结点数目为多少？该树可以具有的最小结点数目为多少？

26.24 如果一棵 AVL 树高度为 4，该树可以具有的最大结点数目为多少？该树可以具有的最小结点数目为多少？

关键术语

AVL tree（高度平衡二叉树）
balance factor（平衡因子）
left-heavy（左偏重）
LL rotation（LL 旋转）
LR rotation（LR 旋转）
perfectly balanced tree（完全平衡树）
right-heavy（右偏重）
RL rotation（RL 旋转）
rotation（旋转）
RR rotation（RR 旋转）
well-balanced tree（高度平衡树）

本章小结

1. AVL 树是高度平衡二叉树。在一棵 AVL 树中，每个结点的两个子树的高度差为 0 或者 1。
2. 在一棵 AVL 树中插入或者删除元素的过程和在常规的二叉查找树中是一样的。不同之处在于可能需要在插入或者删除后重新平衡该树。
3. 插入和删除引起的树的不平衡，通过不平衡结点处的子树的旋转重新获得平衡。
4. 一个结点的重新平衡的过程称为*旋转*。有 4 种可能的旋转：LL *旋转*、LR *旋转*、RR *旋转*、RL *旋转*。
5. AVL 树的高度为 $O(\log n)$。因此，search、insert 以及 delete 方法的时间复杂度为 $O(\log n)$。

测试题

回答位于网址 www.cs.armstrong.edu/liang/intro10e/quiz.html 的本章测试题。

编程练习题

*26.1 （图形化显示 AVL 树）编写一个程序，显示一棵 AVL 树，每个结点处显示平衡因子。

26.2 （比较性能）编写一个测试程序，随机产生 500 000 个数字，并将它们插入 BST 中，然后重新打乱这 500 000 个数字然后执行一次查找，再次打乱数字并从树中删除。编写另外一个测试程序，为 AVLTree 执行同样的操作。比较两个程序的执行时间。

***26.3 （AVL 树的动画）编写一个程序，实现 AVL 树的 insert、delete 以及 search 方法的动画，如

图 26-1 所示。

**26.4 (BST 的父引用) 假定定义在 BST 中的 TreeNode 类包含了指向结点的父结点的引用，如编程练习题 25.15 所示。实现 AVLTree 类来支持这个改变。编写一个测试程序，添加数字 1，2，…，100 到该树中并显示所有叶子结点的路径。

**26.5 (第 k 小的元素) 可以通过中序遍历在 $O(n)$ 的时间内找到 BST 中第 k 小的元素。对于一棵 AVL 树而言，可以在 $O(\log n)$ 时间内找到。为了做到这点，在 AVLTreeNode 中添加一个命名为 size 的新的数据域，存储以该结点为根结点的子树的结点数。注意，一个结点 v 比它的两个子结点的大小的和多 1。图 26-12 显示了一棵 AVL 树，以及树中每个结点的 size 值。

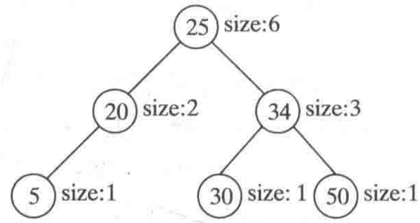

图 26-12 AVLTreeNode 中的 size 数据域存储以该结点为根结点的子树的结点数

AVLTree 类中，添加以下方法，返回树中第 k 小的元素。

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

如果 k < 1 或者 k > the size of the tree（树的大小），方法返回 null。使用递归方法 find(k, root) 实现该方法，递归方法返回以指定根元素的树中的第 k 小的元素。让 A 和 B 作为该根元素的左子结点和右子结点。假设树不为空，并且 $k \leqslant root.size$，可以如下递归定义 find(k,root)：

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

修改 AVLTree 树中的 insert 和 delete 方法，为每个结点中的 size 属性设置正确的值。insert 和 delete 方法仍然为 $O(\log n)$ 时间。find(k) 方法可以在 $O(\log n)$ 时间内执行。因此，可以在一棵 AVL 树中在 $O(\log n)$ 时间内找到第 k 小的元素。

使用下面的主方法来测试你的程序：

```
import java.util.Scanner;

public class Exercise26_05 {
  public static void main(String[] args) {
  AVLTree<Double> tree = new AVLTree<>();
  Scanner input = new Scanner(System.in);
  System.out.print("Enter 15 numbers: ");
  for (int i = 0; i < 15; i++) {
    tree.insert(input.nextDouble());
  }

  System.out.print("Enter k: ");
  System.out.println("The " + k + "th smallest number is " +
    tree.find(k));
  }
}
```

**26.6 (测试 AVLTree) 设计和编写一个完整的测试程序，测试程序清单 26-4 中的 AVLTree 类是否满足所有要求。