同样，有可能一种支持 Map 的操作但不保证有序排列的数据结构可能运行得更快，因为它要做的工作更少。第 5 章将探索这种可能性，并讨论隐藏在另一种 Map 实现背后的想法，这种实现叫做 HashMap。HashMap 将实现的运行时间从 4 秒减少到 3 秒。

小结

我们已经看到树在操作系统、编译器设计、以及查找中的应用。表达式树是更一般结构即所谓分析树(parse tree)的一个小例子，分析树是编译器设计中的核心数据结构。分析树不是二叉树，而是表达式树相对简单的扩充(不过，建立分析树的算法却并不这么简单)。

查找树在算法设计中是非常重要的。它们几乎支持所有有用的操作，而其对数平均开销很小。查找树的非递归实现多少要快一些，但是递归实现更巧妙、更精彩，而且更易于理解和除错。查找树的问题在于，其性能严重地依赖于输入，而输入却是随机的。如果情况不是这样，则运行时间会显著增加，查找树会成为昂贵的链表。

我们见到了处理这个问题的几种方法。AVL 树要求所有节点的左子树与右子树的高度相差最多是 1。这就保证了树不至于太深。不改变树的操作(但插入操作改变树)都可以使用标准二叉查找树的程序。改变树的操作必须将树恢复。这多少有些复杂，特别是在删除的情况。我们叙述了在以 $O(\log N)$的时间插入后如何将树恢复。

我们还考察了伸展树。伸展树中的节点可以达到任意深度，但是在每次访问之后树又以多少有些神秘的方式被调整。实际效果是，任意连续 M 次操作花费$O(M \log N)$时间，它与平衡树花费的时间相同。

与 2 路树或二叉树不同，B 树是平衡 M 路树，它能很好地适应磁盘操作的情况；一种特殊情形是 2-3 树($M=3$)，它是实现平衡查找树的另一种方法。

在实践中，所有平衡树方案的运行时间对于插入和删除操作(除查找稍微快一些外)都不如简单二叉查找树省时(差一个常数因子)，但这一般说来是可以接受的，它防止轻易得到最坏情形的输入。第 12 章将讨论某些另外的查找树数据结构并给出一些详细的实现方法。

最后注意：通过将一些元素插入到查找树然后执行一次中序遍历，我们得到的是排过顺序的元素。这给出排序的一种 $O(N \log N)$算法，如果使用任何成熟的查找树则它就是最坏情形的界。我们将在第 7 章看到一些更好的方法，不过，这些方法的时间界都不可能更低。

练习

问题 4.1～4.3 参考图 4-69 中的树。

4.1 对于图 4-69 中的树：

a. 哪个节点是根？

b. 哪些节点是树叶？

4.2 对于图 4-69 中树上的每一个节点：

a. 指出它的父节点。

b. 列出它的儿子。

c. 列出它的兄弟。

d. 计算它的深度。

e. 计算它的高度。

4.3 图 4-69 中树的深度是多少？