220
~
221

union-find 的成本模型。在研究实现 union-find 的 API 的各种算法时，我们统计的是数组的访问次数（访问任意数组元素的次数，无论读写）。

1.5.2 实现

我们将讨论三种不同的实现，它们均根据以触点为索引的 id[] 数组来确定两个触点是否存在于相同的连通分量中。

1.5.2.1 quick-find 算法

一种方法是保证当且仅当 id[p] 等于 id[q] 时 p 和 q 是连通的。换句话说，在同一个连通分量中的所有触点在 id[] 中的值必须全部相同。这意味着 connected(p, q) 只需要判断 id[p] == id[q]，当且仅当 p 和 q 在同一连通分量中该语句才会返回 true。为了调用 union(p, q) 确保这一点，我们首先要检查它们是否已经存在于同一个连通分量之中。如果是我们就不需要采取任何行动，否则我们面对的情况就是 p 所在的连通分量中的所有触点的 id[] 值均为同一个值，而 q 所在的连通分量中的所有触点的 id[] 值均为另一个值。要将两个分量合二为一，我们必须将两个集合中所有触点所对应的 id[] 元素变为同一个值，如表 1.5.2 所示。为此，我们需要遍历整个数组，将所有和 id[p] 相等的元素的值变为 id[q] 的值。我们也可以将所有和 id[q] 相等的元素的值变为 id[p] 的值——两者皆可。根据上述文字得到的 find() 和 union() 的代码简单明了，如下面的代码框所示。图 1.5.3 显示的是我们的开发用例在处理测试数据 tinyUF.txt 时的完整轨迹。

```
public int find(int p)
{  return id[p];  }

public void union(int p, int q)
{  // 将p和q归并到相同的分量中
   int pID = find(p);
   int qID = find(q);

   // 如果p和q已经在相同的分量之中则不需要采取任何行动
   if (pID == qID) return;

   // 将p的分量重命名为q的名称
   for (int i = 0; i < id.length; i++)
      if (id[i] == pID) id[i] = qID;
   count--;
}
```

quick-find

表 1.5.2 quick-find 概览

find() 方法正在检查id[5]和id[9]

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	8	1	1	1	8	8

union() 方法需要要将所有的1修改为8

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	8	1	1	1	8	8
		8	**8**	**8**	8	8	**8**	**8**	**8**	8	8

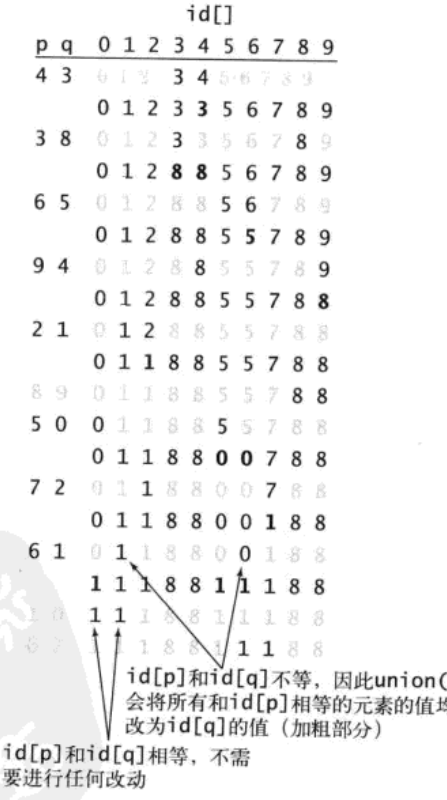


图 1.5.3 quick-find 的轨迹

1.5.2.2 quick-find 算法的分析

find() 操作的速度显然是很快的，因为它只需要访问 id[] 数组一次。但 quick-find 算法一般无法处理大型问题，因为对于每一对输入 union() 都需要扫描整个 id[] 数组。

命题 F。在 quick-find 算法中，每次 find() 调用只需要访问数组一次，而归并两个分量的 union() 操作访问数组的次数在 $(N+3)$ 到 $(2N+1)$ 之间。

证明。由代码马上可以知道，每次 connected() 调用都会检查 id[] 数组中的两个元素是否相等，即会调用两次 find() 方法。归并两个分量的 union() 操作会调用两次 find()，检查 id[] 数组中的全部 N 个元素并改变它们中 1 到 $N-1$ 个元素的值。

假设我们使用 quick-find 算法来解决动态连通性问题并且最后只得到了一个连通分量，那么这至少需要调用 $N-1$ 次 union()，即至少 $(N+3)(N-1) \sim N^2$ 次数组访问——我们马上可以猜想动态连通性的 quick-find 算法是平方级别的。将这种分析推广我们可以得到，quick-find 算法的运行时间对于最终只能得到少数连通分量的一般应用是平方级别的。在计算机上用倍率测试可以很容易验证这个猜想（指导性的例子请见练习 1.5.23）。现代计算机每秒钟能够执行数亿甚至数十亿条指令，因此如果 N 较小的话这个成本并不是很明显。但是在现代应用中我们也很可能需要处理几百万甚至数十亿的触点和连接，例如我们的测试文件 largeUF.txt。如果你还不相信并且觉得自己的计算机足够快，请使用 quick-find 算法找出 largeUF.txt 中所有整数对所表示的连通分量的数量。结论无可争议，使用 quick-find 算法解决这种问题是不可行的，我们需要寻找更好的算法。

222 ~ 223

1.5.2.3 quick-union 算法

我们要讨论的下一个算法的重点是提高 union() 方法的速度，它和 quick-find 算法是互补的。它也基于相同的数据结构——以触点作为索引的 id[] 数组，但我们赋予这些值的意义不同，我们需要用它们来定义更加复杂的结构。确切地说，每个触点所对应的 id[] 元素都是同一个分量中的另一个触点的名称（也可能是它自己）——我们将这种联系称为**链接**。在实现 find() 方法时，我们从给定的触点开始，由它的链接得到另一个触点，再由这个触点的链接到达第三个触点，如此继续跟随着链接直到到达一个根触点，即链接指向自己的触点（你将会看到，这样一个触点必然存在）。当且仅当分别由两个触点开始的这个过程到达了同一个根触点时它们存在于同一个连通分量之中。为了保证这个过程的有效性，我们需要 union(p, q) 来保证这一点。它的实现很简单：我们由 p 和 q 的链接分别找到它们的根触点，然后只需将一个根触点链接到另一个即可将一个分量重命名为另一个分量，因此这个算法叫做 quick-union。和刚才一样，无论是重命名含有 p 的分量还是重命名含有 q 的分量都可以，右侧的这段实现重命名了 p 所在的分量。图 1.5.5 显示了 quick-union 算法在处理 tinyUF.txt 时的轨迹。图 1.5.4 能够很好地说明图 1.5.5（见 1.5.2.4 节）中的轨迹，我们接下来要讨论的就是它。

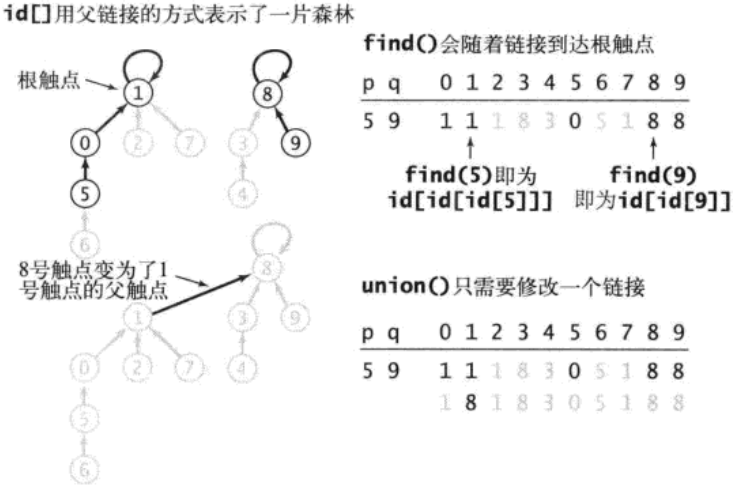
```
private int find(int p)
{  // 找出分量的名称
   while (p != id[p]) p = id[p];
   return p;
}

public void union(int p, int q)
{  // 将p和q的根节点统一
   int pRoot = find(p);
   int qRoot = find(q);
   if (pRoot == qRoot) return;

   id[pRoot] = qRoot;

   count--;
}
```

quick-union



224

图 1.5.4 quick-union 算法概述

1.5.2.4 森林的表示

quick-union 算法的代码很简洁，但有些难以理解。用节点（带标签的圆圈）表示触点，用从一个节点到另一个节点的箭头表示链接，由此得到数据结构的图像表示使我们理解算法的操作变得相对容易。我们的得到的结构是树——从技术上来说，`id[]` 数组用父链接的形式表示了一片森林。为了简化图表，我们常常会省略链接的箭头（因为它们的指向全部朝上）和树的根节点中指向自己的链接。tinyUF.txt 的 `id[]` 数组所对应的森林如图 1.5.5 所示。无论我们从任何触点所对应的节点开始跟随链接，最终都将达到含有该节点的树的根节点。可以用归纳法证明这个性质的正确性：在数组被初始化之后，每个节点的链接都指向它自己；如果在某次 `union()` 操作之前这条性质成立，那么操作之后它必然也成立。因此，quick-union 中的 `find()` 方法能够返回根节点所对应的触点的名称（这样 `connected()` 才能够判定两个触点是否在同一棵树中）。这种表示方法对于这个问题很实用，因为当且仅当两个触点存在于相同的分量之中时它们对应的节点才会在同一棵树中。另外，构造树并不困难：quick-union 中 `union()` 的实现只用了一条语句就将一个根节点变为另一个根节点的父节点，从而归并了两棵树。

1.5.2.5 quick-union 算法的分析

quick-union 算法看起来比 quick-find 算法更快，因为它不需要为每对输入遍历整个数组。但它能够快多少呢？分析 quick-union 算法的成本比分析 quick-find 算法的成本更困难，因为这依赖于输入的特点。在最好的情况下，`find()` 只需要访问数组一次就能够得到一个触点所在的分量的标识符；而在最坏情况下，这需要 $2N - 1$ 次数组访问，如图 1.5.6 中的 0 触点（这个估计是较为保守的，因为 `while` 循环中经过编译的代码对 `id[p]` 的第二次引用一般都不会访问数组）。由此我们不难构造一个最佳情况的输入使得解决动态连通性问题的用例的运行时间是线性级别的；另一方面，我们也可以构造一个最坏情况的输入，此时它的运行时间是平方级别的（请见图 1.5.6 和下面的命题 G）。幸好我们不需要面对分析 quick-union 算法的问题，我们也不会仔细对比 quick-union 算法和 quick-find 算法的性能，因为我们下面将会学习一种比两者的效率都高得多的算法。目前，我们可以将 quick-union 算法看做是 quick-find 算法的一种改良，因为它解决了 quick-find 算法中最主要的问题（`union()` 操作总是线性级别的）。对于一般的输入数据这个变化显然是一次改进，但 quick-union 算法仍然存在问题，我们不能保证在所有情况下它都能比 quick-find 算法快得多（对于某些输入，quick-union 算法并不比 quick-find 算法快）。

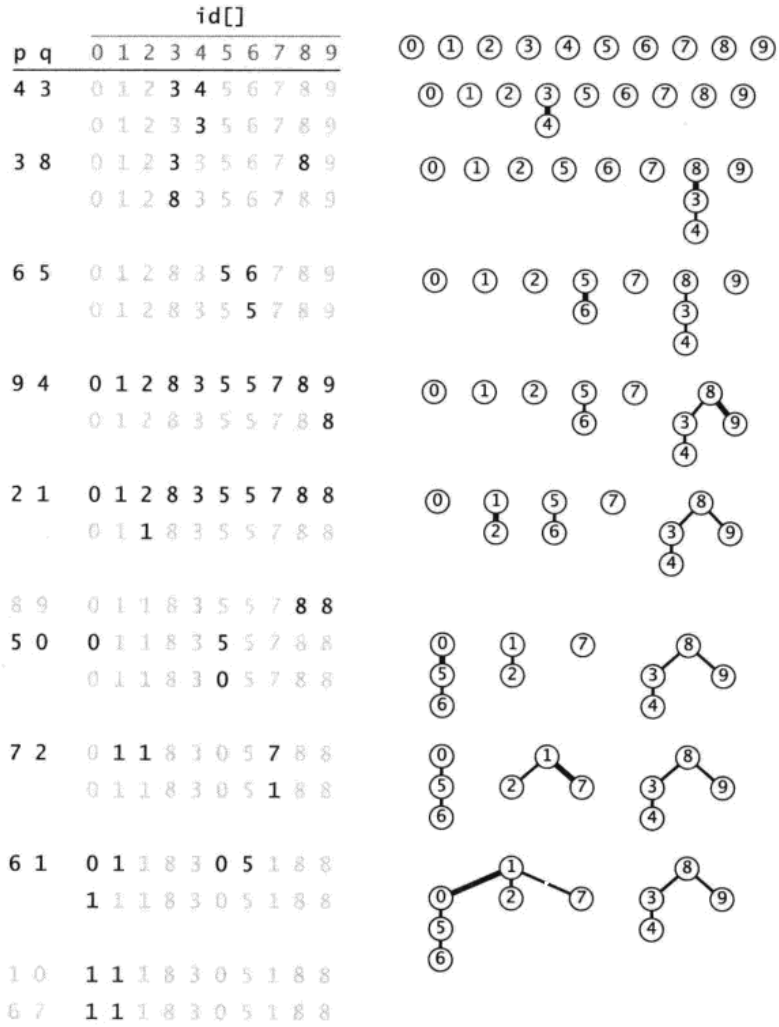


图 1.5.5 quick-union 算法的轨迹（以及相应的森林）

定义。一棵树的大小是它的节点的数量。树中的一个节点的深度是它到根节点的路径上的链接数。树的高度是它的所有节点中的最大深度。

命题 G。quick-union 算法中的 `find()` 方法访问数组的次数为 1 加上给定触点所对应的节点的深度的两倍。`union()` 和 `connected()` 访问数组的次数为两次 `find()` 操作（如果 `union()` 中给定的两个触点分别存在于不同的树中则还需要加 1）。

证明。请见代码。

225 ≀ 226

同样，假设我们使用 quick-union 算法解决了动态连通性问题并最终只得到了一个分量，由命题 G 我们马上可以知道算法的运行时间在最坏情况下是平方级别的。假设输入的整数对是有序的

0-1、0-2、0-3 等，N–1 对之后我们的 N 个触点将全部处于相同的集合之中且由 quick-union 算法得到的树的高度为 N–1，其中 0 链接到 1，1 链接到 2，2 链接到 3，如此下去（请见图 1.5.6）。由命题 G 可知，对于整数对 0 i，union() 操作访问数组的次数为 $2i$+2（触点 0 的深度为 i，触点 i 的深度为 0）。因此，处理 N 对整数所需的所有 find() 操作访问数组的总次数为 $2(1+2+\cdots+N) \sim N^2$。

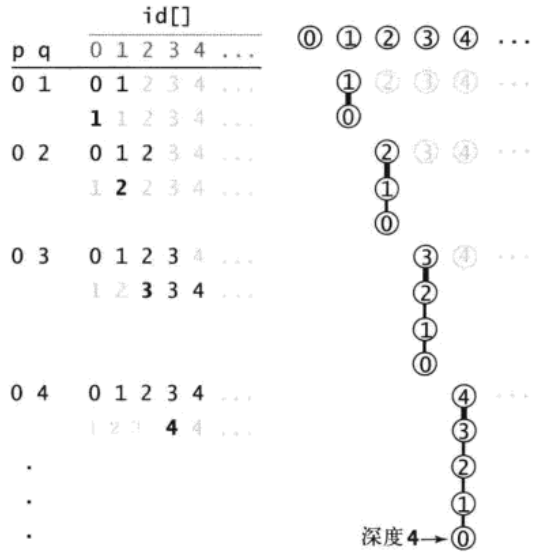


图 1.5.6 quick-union 算法的最坏情况

1.5.2.6 加权 quick-union 算法

幸好，我们只需简单地修改 quick-union 算法就能保证像这样的糟糕情况不再出现。与其在 union() 中随意将一棵树连接到另一棵树，我们现在会记录每一棵树的大小并总是将较小的树连接到较大的树上。这项改动需要添加一个数组和一些代码来记录树中的节点数，如算法 1.5 所示，但它能够大大改进算法的效率。我们将它称为*加权* quick-union 算法（如图 1.5.7 所示）。该算法在处理 tinyUF.txt 时构造的森林如图 1.5.8 中左侧的图所示。即使对于这个较小的例子，该算法构造的树的高度也远远小于未加权的版本所构造的树的高度。

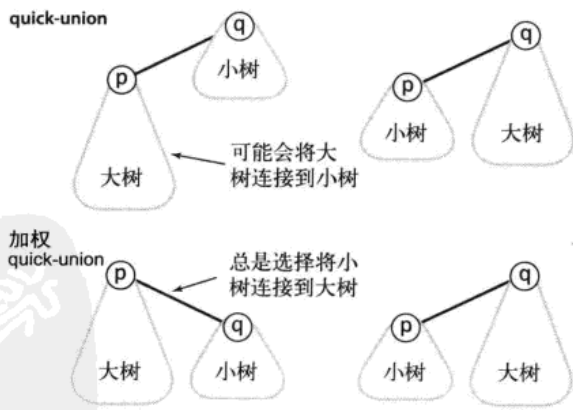


图 1.5.7 加权 quick-union

1.5.2.7 加权 quick-union 算法的分析

图 1.5.8 显示了加权 quick-union 算法的最坏情况。其中将要被归并的树的大小总是相等的（且总是 2 的幂）。这些树的结构看起来很复杂，但它们均含有 2^n 个节点，因此高度都正好是 n。另外，当我们归并两个含有 2^n 个节点的树时，我们得到的树含有 2^{n+1} 个节点，由此将树的高度增加到了 $n+1$。由此推广我们可以证明加权 quick-union 算法能够保证对数级别的性能。加权 quick-union 算法的实现如算法 1.5 所示。

```
% java WeightedQuickUnionUF < mediumUF.txt
528 503
548 523
...
3 components

% java WeightedQuickUnionUF < largeUF.txt
786321 134521
696834 98245
...
6 components
```

227

算法 1.5（续） union-find 算法的实现（加权 quick-union 算法）

```
public class WeightedQuickUnionUF
{
   private int[] id;      // 父链接数组（由触点索引）
   private int[] sz;      // （由触点索引的）各个根节点所对应的分量的大小
   private int count;     // 连通分量的数量
   public WeightedQuickUnionUF(int N)
   {
      count = N;
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
      sz = new int[N];
      for (int i = 0; i < N; i++) sz[i] = 1;
   }
   public int count()
   {  return count;  }
   public boolean connected(int p, int q)
   {  return find(p) == find(q);  }
   private int find(int p)
   {  // 跟随链接找到根节点
      while (p != id[p]) p = id[p];
      return p;
   }
   public void union(int p, int q)
   {
      int i = find(p);
      int j = find(q);
      if (i == j) return;
      // 将小树的根节点连接到大树的根节点
      if   (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
      else                 { id[j] = i; sz[i] += sz[j]; }
      count--;
   }
}
```

根据正文所述的森林表示方法这段代码很容易理解。我们加入了一个由触点索引的实例变量数组 sz[]，这样 union() 就可以将小树的根节点连接到大树的根节点。这使得算法能够处理规模较大的问题。

228

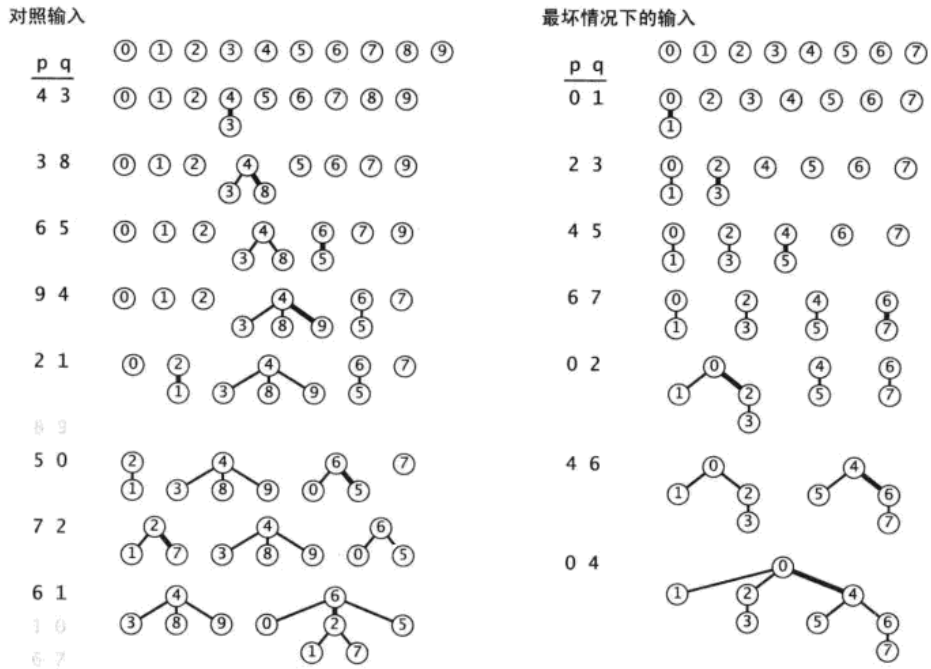


图 1.5.8 加权 quick-union 算法的轨迹（森林）

命题 H。对于 N 个触点，加权 quick-union 算法构造的森林中的任意节点的深度最多为 $\lg N$。

证明。我们可以用归纳法证明一个更强的命题，即森林中大小为 k 的树的高度最多为 $\lg k$。在原始情况下，当 k 等于 1 时树的高度为 0。根据归纳法，我们假设大小为 i 的树的高度最多为 $\lg i$，其中 $i<k$。设 $i \leq j$ 且 $i+j=k$，当我们将大小为 i 和大小为 j 的树归并时，quick-union 算法和加权 quick-union 算法中触点与深度示例如图 1.5.9 所示。小树中的所有节点的深度增加了 1，但它们现在所在的树的大小为 $i+j=k$，而 $1+\lg i=\lg(i+i) \leq \lg(i+j)=\lg k$，性质成立。

229

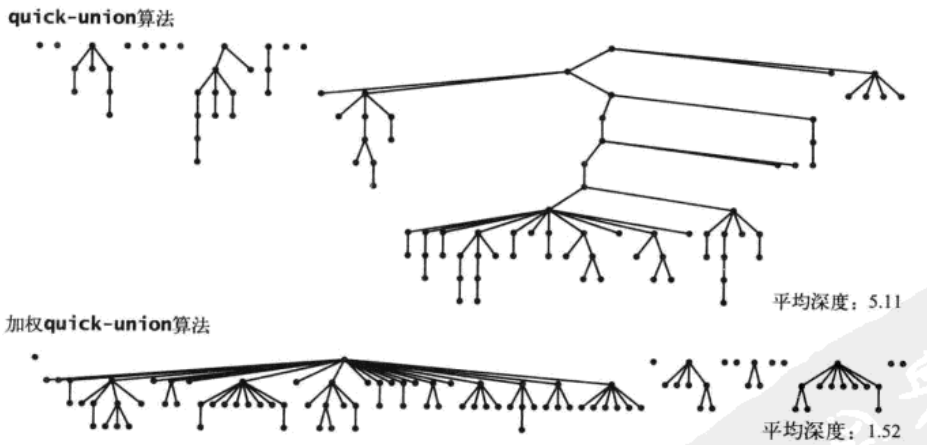


图 1.5.9 quick-union 算法与加权 quick-union 算法的对比（100 个触点，88 次 union() 操作）

推论。对于加权 quick-union 算法和 N 个触点，在最坏情况下 `find()`、`connected()` 和 `union()` 的成本的增长数量级为 $\log N$。

证明。在森林中，对于从一个节点到它的根节点的路径上的每个节点，每种操作最多都只会访问数组常数次。

对于动态连通性问题，命题 H 和它的推论的实际意义在于加权 quick-union 算法是三种算法中唯一可以用于解决大型实际问题的算法。加权 quick-union 算法处理 N 个触点和 M 条连接时最多访问数组 $cM\lg N$ 次，其中 c 为常数。这个结果和 quick-find 算法（以及某些情况下的 quick-union 算法）需要访问数组至少 MN 次形成了鲜明的对比。因此，有了加权 quick-union 算法我们就能保证能够在合理的时间范围内解决实际中的大规模动态连通性问题。只需要多写几行代码，我们所得到的程序在处理实际应用中的大型动态连通性问题时就会比简单的算法快数百万倍。

图 1.5.9 显示的是一个含有 100 个触点的例子。从图中我们可以很明显地看到，加权 quick-union 算法中远离根节点的节点相对较少。事实上，只含有一个节点的树被归并到更大的树中的情况很常见，这样该节点到根节点的距离也只有一条链接而已。针对大规模问题的经验性研究告诉我们，加权 quick-union 算法在解决实际问题时一般都能在常数时间内完成每个操作（如表 1.5.3 所示）。我们可能很难找到比它效率更高的算法了。 230

表 1.5.3 各种 union-find 算法的性能特点

算　法	存在 N 个触点时成本的增长数量级（最坏情况下）		
	构造函数	`union()`	`find()`
quick-find 算法	N	N	1
quick-union 算法	N	树的高度	树的高度
加权 quick-union 算法	N	$\lg N$	$\lg N$
使用路径压缩的加权 quick-union 算法	N	非常非常地接近但仍没有达到 1（均摊成本）（请见练习 1.5.13）	
理想情况	N	1	1

1.5.2.8 最优算法

我们可以找到一种能够保证在常数时间内完成各种操作的算法吗？这个问题非常困难并且困扰了研究者们许多年。在寻找答案的过程中，大家研究了 quick-union 算法和加权 quick-union 算法的各种变体。例如，下面这种路径压缩方法很容易实现。理想情况下，我们希望每个节点都直接链接到它的根节点上，但我们又不想像 quick-find 算法那样通过修改大量链接做到这一点。我们接近这种理想状态的方式很简单，就是在检查节点的同时将它们直接链接到根节点。这种方法乍一看很激进，但它的实现非常容易，而且这些树并没有阻止我们进行这种修改的特殊结构：如果这么做能够改进算法的效率，我们就应该实现它。要实现路径压缩，只需要为 `find()` 添加一个循环，将在路径上遇到的所有节点都直接链接到根节点。我们所得到的结果是几乎完全扁平化的树，它和 quick-find 算法理想情况下所得到的树非常接近。这种方法即简单又有效，但在实际情况下已经不太可能对加权 quick-union 算法继续进行任何改进了（请见练习 1.5.24）。对该情况的理论研究结果非常复杂也值得我们注意：*路径压缩的加权 quick-union 算法是最优的算法，但并非所有操作都能在常数时间内完成*。也就是说，使用路径压缩的加权 quick-union 算法的每个操作在在最坏情况下（即均摊后）

都不是常数级别的，而且不存在其他算法能够保证 union-find 算法的所有操作在均摊后都是常数级别的（在非常一般的 cell probe 模型之下）。使用路径压缩的加权 quick-union 算法已经是我们对于
231 这个问题能够给出的最优解了。

1.5.2.9 均摊成本的图像

与对其他任何数据结构实现的讨论一样，我们应该按照 1.4 节中的讨论在实验中用典型的用例验证我们对算法性能的猜想。图 1.5.10 详细显示了我们的动态连通性问题的开发用例在使用各种算法处理一份含有 625 个触点的样例数据（mediumUF.txt）时的性能。绘制这种图像很简单（请见练习 1.5.16）：在处理第 i 个连接时，用一个变量 `cost` 记录其间访问数组（`id[]` 或 `sz[]`）的次数，并用一个变量 `total` 记录到目前为止数组访问的总次数。我们在 `(i, cost)` 处画一个灰点，在 `(i, total/i)` 处画一个红点，红点表示的是每个操作的平均成本，即均摊成本。图像能够帮助我们更好地理解算法的行为。对于 quick-find 算法，每次 `union()` 操作都至少访问数组 625 次（每归并一个分量还要加 1，最多再加 625），每次 `connected()` 操作都访问数组 2 次。一开始，大多数连接都会产生一个 `union()` 调用，因此累计平均值徘徊在 625 左右；后来，大多数连接产生的 `connected()` 调用会跳过 `union()`，因此累计平均值开始下降，但仍保持了相对较高的水平（能够产生大量 `connected()` 调用并跳过 `union()` 的输入性能要好得多，例子请见练习 1.5.23）。对于 quick-union 算法，所有的操作在初始阶段访问数组的次数都不多；到了后期，树的高度成为一个重要因素，均摊成本的增长很明显。对于加权 quick-union 算法，树的高度一直很小，没有任何昂贵的操作，均摊成本也很低。这些实验验证了我们的结论，显然非常有必要实现加权 quick-union 算法，在解决实际问题时已经没有多少进一步改
232 进的空间了。

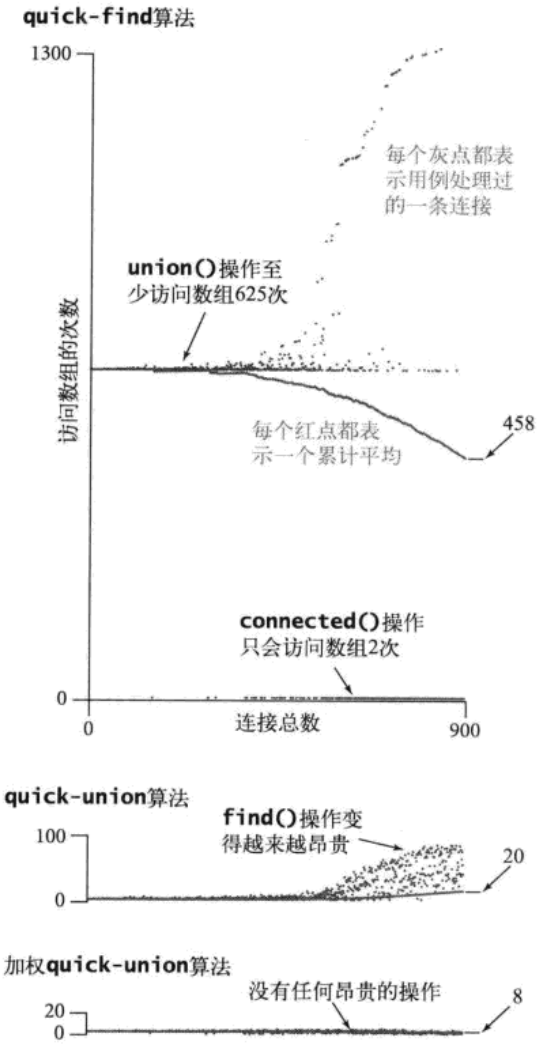


图 1.5.10 所有操作的总成本
（625 个触点，另见彩插）

1.5.3 展望

直观感觉上，我们学习的每种 UF 的实现都改进了上一个版本的实现，但这个过程并不突兀，因为我们可以总结学者们对这些算法多年的研究。我们很明确地说明了问题，解决方法的实现也很

简单，因此可以用经验性的数据评估各个算法的优劣。另外，还可以通过这些研究验证将算法的性能量化的数学结论。只要可能，我们在本书中研究各种基础问题时都会遵循类似于本节中讨论 union-find 问题时的基本步骤，在这里我们要再次强调它们。

- 完整而详细地定义问题，找出解决问题所必需的基本抽象操作并定义一份 API。
- 简洁地实现一种初级算法，给出一个精心组织的开发用例并使用实际数据作为输入。
- 当实现所能解决的问题的最大规模达不到期望时决定改进还是放弃。
- 逐步改进实现，通过经验性分析或（和）数学分析验证改进后的效果。
- 用更高层次的抽象表示数据结构或算法来设计更高级的改进版本。
- 如果可能尽量为最坏情况下的性能提供保证，但在处理普通数据时也要有良好的性能。
- 在适当的时候将更细致的深入研究留给有经验的研究者并继续解决下一个问题。

我们从 union-find 问题中可以看到，算法设计在解决实际问题时能够为程序的性能带来惊人的提高，这种潜力使它成为热门研究领域。还有什么其他类型的设计行为可能将成本降为原来的数百万甚至数十亿分之一呢？

设计高效的算法是一种很有成就感的智力活动，同时也能够产生直接的实际效益。正如动态连通性问题所示，为解决一个简单的问题我们学习了许多算法，它们不但有用有趣，也精巧而引人入胜。我们还将遇到许多新颖独特的算法，它们都是人们在数十年以来为解决许多实际问题而发明的。随着计算机算法在科学和商业领域的应用范围越来越广，能够使用高效的算法来解决老问题并为新问题开发有效的解决方案也越来越重要了。

233

答疑

问　我希望为 API 添加一个 `delete()` 方法来允许用例删除连接。能够给我一些建议吗？

答　目前还没有人能够发明既能处理删除操作而又和本节中所介绍的算法同样简单而高效的算法。这个主题在本书中会反复出现。在我们讨论的一些数据结构中删除比添加要困难得多。

问　cell-probe 模型是什么？

答　它是一种计算模型，其中我们只会记录对随机内存的访问，内存大小足以保存所有输入且假设其他操作均没有成本。

234

练习

1.5.1　使用 quick-find 算法处理序列 9-0 3-4 5-8 7-2 2-1 5-7 0-3 4-2 。对于输入的每一对整数，给出 `id[]` 数组的内容和访问数组的次数。

1.5.2　使用 quick-union 算法（请见 1.5.2.3 节代码框）完成练习 1.5.1。另外，在处理完输入的每对整数之后画出 `id[]` 数组表示的森林。

1.5.3　使用加权 quick-union 算法（请见算法 1.5）完成练习 1.5.1。

1.5.4　在正文的加权 quick-union 算法示例中，对于输入的每一对整数（包括对照输入和最坏情况下的输入），给出 `id[]` 和 `sz[]` 数组的内容以及访问数组的次数。

1.5.5　在一台每秒能够处理 10^9 条指令的计算机上，估计 quick-find 算法解决含有 10^9 个触点和 10^6 条连接的动态连通性问题所需的最短时间（以天记）。假设内循环 `for` 的每一次迭代需要执行 10 条机器指令。

1.5.6 使用加权 quick-union 算法完成练习 1.5.5。

1.5.7 分别为 quick-find 算法和 quick-union 算法实现 QuickFindUF 类和 QuickUnionUF 类。

1.5.8 用一个反例证明 quick-find 算法中的 union() 方法的以下直观实现是错误的：

```
public void union(int p, int q)
{
   if (connected(p, q)) return;
   // 将 p 的分量重命名为 q 的分量
   for (int i = 0; i < id.length; i++)
       if (id[i] == id[p]) id[i] = id[q];
   count--;
}
```

1.5.9 画出下面的 id[] 数组所对应的树。这可能是加权 quick-union 算法得到的结果吗？解释为什么不可能，或者给出能够得到该数组的一系列操作。

i	0	1	2	3	4	5	6	7	8	9
id[i]	1	1	3	1	5	6	1	3	4	5

235

1.5.10 在加权 quick-union 算法中，假设我们将 id[find(p)] 的值设为 q 而非 id[find(q)]，所得的算法是正确的吗？

答：是，但这会增加树的高度，因此无法保证同样的性能。

1.5.11 实现加权 quick-find 算法，其中我们总是将较小的分量重命名为较大的分量的标识符。这种改变会对性能产生怎样的影响？

236

提高题

1.5.12 使用路径压缩的 quick-union 算法。根据路径压缩修改 quick-union 算法（请见 1.5.2.3 节），在 find() 方法中添加一个循环来将从 p 到根节点的路径上的每个触点都连接到根节点。给出一列输入，使该方法能够产生一条长度为 4 的路径。注意：该算法的所有操作的均摊成本已知为对数级别。

1.5.13 使用路径压缩的加权 quick-union 算法。修改加权 quick-union 算法（算法 1.5），实现如练习 1.5.12 所述的路径压缩。给出一列输入，使该方法能够产生一棵高度为 4 的树。注意：该算法的所有操作的均摊成本已知被限制在反 Ackermann 函数的范围之内，且对于实际应用中可能出现的所有 N 值均小于 5。

1.5.14 根据高度加权的 quick-union 算法。给出 UF 的一个实现，使用和加权 quick-union 算法相同的策略，但记录的是树的高度并总是将较矮的树连接到较高的树上。用算法证明 N 个触点的树的高度不会超过其大小的对数级别。

1.5.15 二项树。请证明，对于加权 quick-union 算法，在最坏情况下树中每一层的节点数均为二项式系数。在这种情况下，计算含有 $N=2^n$ 个节点的树中节点的平均深度。

1.5.16 均摊成本的图像。修改你为练习 1.5.7 给出的实现，绘出如正文所示的均摊成本的图像。

1.5.17 随机连接。设计 UF 的一个用例 ErdosRenyi，从命令行接受一个整数 N，在 0 到 N-1 之间产生随机整数对，调用 connected() 判断它们是否相连，如果不是则调用 union() 方法（和我们的开发用例一样）。不断循环直到所有触点均相互连通并打印出生成的连接总数。将你的程序打包成一个接受参数 N 并返回连接总数的静态方法 count()，添加一个 main() 方法从命令行接受 N，调用 count() 并打印它的返回值。

1.5.18 随机网格生成器。编写一个程序 RandomGrid，从命令行接受一个 int 值 N，生成一个 $N \times N$ 的网格中的所有连接。它们的排列随机且方向随机（即 (p q) 和 (q p) 出现的可能性是相等的），将这个结果打印到标准输出中。可以使用 RandomBag 将所有连接随机排列（请见练习 1.3.34），并使用如右下所示的 Connection 嵌套类来将 p 和 q 封装到一个对象中。将程序打包成两个静态方法：generate()，接受参数 N 并返回一个连接的数组；main()，从命令行接受参数 N，调用 generate()，遍历返回的数组并打印出所有连接。 237

1.5.19 动画。编写一个 RandomGrid（请见练习 1.5.18）的用例，和我们的开发用例一样使用 UnionFind 来检查触点的连通性并在处理的同时用 StdDraw 将它们绘出。

1.5.20 动态生长。使用链表或大小可变的数组实现加权 quick-union 算法，去掉需要预先知道对象数量的限制。为 API 添加一个新方法 newSite()，它应该返回一个类型为 int 的标识符。

```
private class Connection
{
   int p;
   int q;

   public Connection(int p, int q)
   {  this.p = p; this.q = q;  }
}
```

封装连接的嵌套类 238

实验题

1.5.21 Erdös-Renyi 模型。使用练习 1.5.17 的用例验证这个猜想：得到单个连通分量所需生成的整数对数量为 ~1/2Nln$N$。

1.5.22 Erdös-Renyi 模型的倍率实验。开发一个性能测试用例，从命令行接受一个 int 值 T 并进行 T 次以下实验：使用练习 1.5.17 的用例生成随机连接，和我们的开发用例一样使用 UnionFind 来检查触点的连通性，不断循环直到所有触点均相互连通。对于每个 N，打印出 N 值和平均所需的连接数以及前后两次运行时间的比值。使用你的程序验证正文中的猜想：quick-find 算法和 quick-union 算法的运行时间是平方级别的，加权 quick-union 算法则接近线性级别。

1.5.23 在 Erdös-Renyi 模型下比较 quick-find 算法和 quick-union 算法。开发一个性能测试用例，从命令行接受一个 int 值 T 并进行 T 次以下实验：使用练习 1.5.17 的用例生成随机连接。保存这些连接并和我们的开发用例一样分别用 quick-find 算法和 quick-union 算法检查触点的连通性，不断循环直到所有触点均相互连通。对于每个 N，打印出 N 值和两种算法的运行时间的比值。

1.5.24 适用于 Erdös-Renyi 模型的快速算法。在练习 1.5.23 的测试中增加加权 quick-union 算法和使用路径压缩的加权 quick-union 算法。你能分辨出这两种算法的区别吗？

1.5.25 随机网格的倍率测试。开发一个性能测试用例，从命令行接受一个 int 值 T 并进行 T 次以下实验：使用练习 1.5.18 的用例生成一个 N×N 的随机网格，所有连接的方向随机且排列随机。和我们的开发用例一样使用 UnionFind 来检查触点的连通性，不断循环直到所有触点均相互连通。对于每个 N，打印出 N 值和平均所需的连接数以及前后两次运行时间的比值。使用你的程序验证正文中的猜想：quick-find 算法和 quick-union 算法的运行时间是平方级别的，加权 quick-union 算法则接近线性级别。注意：随着 N 值加倍，网格中触点的数量会乘 4，因此平方级别的算法的运行时间会变为原来的 16 倍，线性级别的算法的运行时间则变为原来的 4 倍。 239

1.5.26 Erdös-Renyi 模型的均摊成本图像。开发一个用例，从命令行接受一个 int 值 N，在 0 到 N–1 之间产生随机整数对，调用 connected() 判断它们是否相连，如果不是则调用 union() 方法（和我们的开发用例一样）。不断循环直到所有触点均相互连通。按照正文的样式将所有操作的均摊成本绘制成图像。 240



第2章 排 序

排序就是将一组对象按照某种逻辑顺序重新排列的过程。比如，信用卡账单中的交易是按照日期排序的——这种排序很可能使用了某种排序算法。在计算时代早期，大家普遍认为30%的计算周期都用在了排序上。如果今天这个比例降低了，可能的原因之一是如今的排序算法更加高效，而并非排序的重要性降低了。现在计算机的广泛使用使得数据无处不在，而整理数据的第一步通常就是进行排序。所有的计算机系统都实现了各种排序算法以供系统和用户使用。

即使你只是使用标准库中的排序函数，学习排序算法仍然有三大实际意义：

- 对排序算法的分析将有助于你全面理解本书中比较算法性能的方法；
- 类似的技术也能有效解决其他类型的问题；
- 排序算法常常是我们解决其他问题的第一步。

更重要的是这些算法都很经典、优雅和高效。

排序在商业数据处理和现代科学计算中有着重要的地位，它能够应用于事物处理、组合优化、天体物理学、分子动力学、语言学、基因组学、天气预报和很多其他领域。其中一种排序算法（快速排序，见2.3节）甚至被誉为20世纪科学和工程领域的十大算法之一。

241 ~ 243

在本章中我们将学习几种经典的排序算法，并高效地实现了“优先队列”这种基础数据类型。我们将讨论比较排序算法的理论基础并在本章结尾总结若干排序算法和优先队列的应用。

2.1 初级排序算法

作为对排序算法领域的第一次探索，我们将学习两种初级的排序算法以及其中一种的一个变体。深入学习这些相对简单的算法的原因在于：第一，我们将通过它们熟悉一些术语和简单的技巧；第二，这些简单的算法在某些情况下比我们之后将会讨论的复杂算法更有效；第三，以后你会发现，它们有助于我们改进复杂算法的效率。

2.1.1 游戏规则

我们关注的主要对象是重新排列数组元素的算法，其中每个元素都有一个主键。排序算法的目标就是将所有元素的主键按照某种方式排列（通常是按照大小或是字母顺序）。排序后索引较大的主键大于等于索引较小的主键。元素和主键的具体性质在不同的应用中千差万别。在 Java 中，元素通常都是对象，对主键的抽象描述则是通过一种内置的机制（请见 2.1.1.4 节中的 Comparable 接口）来完成的。

“排序算法类模版”中的 Example 类展示了我们的习惯约定：我们会将排序代码放在类的 sort() 方法中，该类还将包含辅助函数 less() 和 exch()（可能还有其他辅助函数）以及一个示例用例 main()。Example 类还包含了一些早期调试使用的代码：测试用例 main() 将标准输入得到的字符串排序，并用私有方法 show() 打印字符数组的内容。我们还会在本章中遇到各种用于比较不同算法并研究它们的性能的测试用例。为了区别不同的排序算法，我们为相应的类取了不同的名字，用例可以根据名字调用不同的实现，例如 Insertion.sort()、Merge.sort()、Quick.sort() 等。

大多数情况下，我们的排序代码只会通过两个方法操作数据：less() 方法对元素进行比较，exch() 方法将元素交换位置。exch() 方法的实现很简单，通过 Comparable 接口实现 less() 方法也不困难。将数据操作限制在这两个方法中使得代码的可读性和可移植性更好，更容易验证代码的正确性、分析性能以及排序算法之间的比较。在学习具体的排序算法实现之前，我们先讨论几个对于所有排序算法都很重要的问题。 244

排序算法类的模板

```
public class Example
{
   public static void sort(Comparable[] a)
   {  /* 请见算法2.1、算法2.2、算法2.3、算法2.4、算法2.5或算法2.7*/ }

   private static boolean less(Comparable v, Comparable w)
   {  return v.compareTo(w) < 0;  }

   private static void exch(Comparable[] a, int i, int j)
   {  Comparable t = a[i]; a[i] = a[j]; a[j] = t;  }

   private static void show(Comparable[] a)
   {  // 在单行中打印数组
      for (int i = 0; i < a.length; i++)
         StdOut.print(a[i] + " ");
      StdOut.println();
   }
   public static boolean isSorted(Comparable[] a)
   {  // 测试数组元素是否有序
      for (int i = 1; i < a.length; i++)
         if (less(a[i], a[i-1]))  return false;
```

```
% more tiny.txt
S O R T E X A M P L E

% java Example < tiny.txt
A E E L M O P R S T X
```

```
        return true;
    }

    public static void main(String[]
args)
    {  // 从标准输入读取字符串，将它们排序并输出
       String[] a = In.readStrings();
       sort(a);
       assert isSorted(a);
       show(a);
    }
}
```

```
% more words3.txt
bed bug dad yes zoo ... all bad yet

% java Example < words.txt
all bad bed bug dad ... yes yet zoo
```

这个类展示的是数组排序实现的框架。对于我们学习的每种排序算法，我们都会为这样一个类实现一个 sort() 方法并将 Example 改为算法的名称。测试用例会将标准输入得到的字符串排序，但是这段
245 代码使我们的排序方法适用于任意实现了 Comparable 接口的数据类型。

2.1.1.1 验证

无论数组的初始状态是什么，排序算法都能成功吗？谨慎起见，我们会在测试代码中添加一条语句 assert isSorted(a); 来确认排序后数组元素都是有序的。尽管一般都会测试代码并从数学上证明算法的正确性，但在实现每个排序算法时加上这条语句仍然是必要的。需要注意的是，如果我们只使用 exch() 来交换数组的元素，这个测试就足够了。当我们直接将值存入数组中时，这条语句无法提供足够的保证（例如，把初始输入数组的元素全部置为相同的值也能通过这个测试）。

2.1.1.2 运行时间

我们还要评估算法的性能。首先，要计算各个排序算法在不同的随机输入下的基本操作的次数（包括比较和交换，或者是读写数组的次数）。然后，我们用这些数据来估计算法的相对性能并介绍在实验中验证这些猜想所使用的工具。对于大多数实现，代码风格一致会使我们更容易作出对性能的合理猜想。

排序成本模型。在研究排序算法时，我们需要计算比较和交换的数量。对于不交换元素的算法，我们会计算访问数组的次数。

2.1.1.3 额外的内存使用

排序算法的额外内存开销和运行时间是同等重要的。排序算法可以分为两类：除了函数调用所需的栈和固定数目的实例变量之外无需额外内存的*原地排序算法*，以及需要额外内存空间来存储另一份数组副本的其他排序算法。

2.1.1.4 数据类型

我们的排序算法模板适用于任何实现了 Comparable 接口的数据类型。遵守 Java 惯例的好处是很多你希望排序的数据都实现了 Comparable 接口。例如，Java 中封装数字的类型 Integer 和 Double，以及 String 和其他许多高级数据类型（如 File 和 URL）都实现了 Comparable 接口。因此你可以直接用这些类型的数组作为参数调用我们的排序方法。例如，右上方的代码使用了快速排序（请见 2.3 节）来对 N 个随机的 Double 数据进
246 行排序。

```
Double a[] = new Double[N];
for (int i = 0; i < N; i++)
  a[i] = StdRandom.uniform();
Quick.sort(a);
```

将N个随机值的数组排序

在创建自己的数据类型时，我们只要实现 Comparable 接口就能够保证用例代码可以将其排序。要做到这一点，只需要实现一个 compareTo() 方法来定义目标类型对象的自然次序，如右侧的 Date 数据类型所示（参见表 1.2.12）。

对于 v<w、v=w 和 v>w 三种情况，Java 的习惯是在 v.compareTo(w) 被调用时分别返回一个负整数、零和一个正整数（一般是 –1、0 和 1）。为了节约篇幅，我们接下来用 v>w 来表示 v.compareTo(w)>0 这样的代码。一般来说，如果 v 和 w 无法比较或者两者之一是 null，v.compareTo(w) 将会抛出一个异常。此外，compareTo() 必须实现一个完整的比较序列，即：

- 自反性，对于所有的 v，v=v；
- 反对称性，对于所有的 v<w 都有 v>w，且 v=w 时 w=v；
- 传递性，对于所有的 v、w 和 x，如果 v<=w 且 w<=x，则 v<=x。

```
public class Date implements Comparable<Date>
{
   private final int day;
   private final int month;
   private final int year;

   public Date(int d, int m, int y)
   {  day = d; month = m; year = y; }

   public int day()   {  return day;    }
   public int month() {  return month;  }
   public int year()  {  return year;    }

   public int compareTo(Date that)
   {
      if (this.year  > that.year ) return +1;
      if (this.year  < that.year ) return -1;
      if (this.month > that.month) return +1;
      if (this.month < that.month) return -1;
      if (this.day   > that.day  ) return +1;
      if (this.day   < that.day  ) return -1;
      return 0;
   }

   public String toString()
   { return month + "/" + day + "/" + year; }
}
```

定义一个可比较的数据类型

从数学上来说这些规则都很标准和自然，遵守它们应该不难。总之，compareTo() 实现了我们的主键抽象——它给出了实现了 Comparable 接口的任意数据类型的对象的大小顺序的定义。需要注意的是 compareTo() 方法不一定会用到进行比较的实例的所有实例变量，毕竟数组元素的主键很可能只是每个元素的一小部分。

本章剩余篇幅将会讨论对一组自然次序的对象进行排序的各种算法。为了比较和对照各种算法，我们会检查它们的许多性质，包括在各种输入下它们比较和交换数组元素的次数以及额外内存的使用量。通过这些我们能够对它们的性能作出猜想，而这些猜想在过去的数十年间已经在无数的计算机上被验证过了。所有的实现都是需要通过检验的，所以我们也会讨论相关的工具。在研究经典的选择排序、插入排序、希尔排序、归并排序、快速排序和堆排序之后，我们将在 2.5 节讨论一些实际的应用和问题。 247

2.1.2 选择排序

一种最简单的排序算法是这样的：首先，找到数组中最小的那个元素，其次，将它和数组的第一个元素交换位置（如果第一个元素就是最小元素那么它就和自己交换）。再次，在剩下的元素中找到最小的元素，将它与数组的第二个元素交换位置。如此往复，直到将整个数组排序。这种方法叫做选择排序，因为它在不断地选择剩余元素之中的最小者。

如算法 2.1 所示，选择排序的内循环只是在比较当前元素与目前已知的最小元素（以及将当前索引加 1 和检查是否代码越界），这已经简单到了极点。交换元素的代码写在内循环之外，每次交换都能排定一个元素，因此交换的总次数是 N。所以算法的时间效率取决于比较的次数。

命题A。对于长度为 N 的数组，选择排序需要大约 $N^2/2$ 次比较和 N 次交换。

证明。可以通过算法的排序轨迹来证明这一点。我们用一张 $N \times N$ 的表格来表示排序的轨迹（见算法2.1下部的表格），其中每个非灰色字符都表示一次比较。表格中大约一半的元素不是灰色的——即对角线和其上部分的元素。对角线上的每个元素都对应着一次交换。通过查看代码我们可以更精确地得到，0到 $N-1$ 的任意 i 都会进行一次交换和 $N-1-i$ 次比较，因此总共有 N 次交换以及 $(N-1)+(N-2)+\cdots+2+1=N(N-1)/2 \sim N^2/2$ 次比较。

总的来说，选择排序是一种很容易理解和实现的简单排序算法，它有两个很鲜明的特点。

运行时间和输入无关。为了找出最小的元素而扫描一遍数组并不能为下一遍扫描提供什么信息。这种性质在某些情况下是缺点，因为使用选择排序的人可能会惊讶地发现，一个已经有序的数组或是主键全部相等的数组和一个元素随机排列的数组所用的排序时间竟然一样长！我们将会看到，其他算法会更善于利用输入的初始状态。

数据移动是最少的。每次交换都会改变两个数组元素的值，因此选择排序用了 N 次交换——交换次数和数组的大小是线性关系。我们将研究的其他任何算法都不具备这个特征（大部分的增长数量级都是线性对数或是平方级别）。

248

算法2.1 选择排序

```
public class Selection
{
   public static void sort(Comparable[] a)
   {  // 将a[]按升序排列
      int N = a.length;                    // 数组长度
      for (int i = 0; i < N; i++)
      {  // 将a[i]和a[i+1..N]中最小的元素交换
         int min = i;                      // 最小元素的索引
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min])) min = j;
         exch(a, i, min);
      }
   }
   // less()、exch()、isSorted()和main()方法见“排序算法类模板”
}
```

该算法将第 i 小的元素放到 a[i] 之中。数组的第 i 个位置的左边是 i 个最小的元素且它们不会再被访问。

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	**A**	M	P	L	E
1	4	A	O	R	T	**E**	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	**E**
3	9	A	E	E	T	O	X	S	M	P	**L**	R
4	7	A	E	E	L	O	X	S	**M**	P	T	R
5	7	A	E	E	L	M	X	S	**O**	P	T	R
6	8	A	E	E	L	M	O	S	X	**P**	T	R
7	10	A	E	E	L	M	O	P	X	S	T	**R**

算法在黑色的元素中查找最小值

加粗的元素都是**a[min]**

PDG



选择排序的轨迹（每次交换后的数组内容）

249

2.1.3 插入排序

通常人们整理桥牌的方法是一张一张的来，将每一张牌插入到其他已经有序的牌中的适当位置。在计算机的实现中，为了给要插入的元素腾出空间，我们需要将其余所有元素在插入之前都向右移动一位。这种算法叫做插入排序，实现请见算法 2.2。

与选择排序一样，当前索引左边的所有元素都是有序的，但它们的最终位置还不确定，为了给更小的元素腾出空间，它们可能会被移动。但是当索引到达数组的右端时，数组排序就完成了。

和选择排序不同的是，插入排序所需的时间取决于输入中元素的初始顺序。例如，对一个很大且其中的元素已经有序（或接近有序）的数组进行排序将会比对随机顺序的数组或是逆序数组进行排序要快得多。

命题 B。对于随机排列的长度为 N 且主键不重复的数组，平均情况下插入排序需要 ~ $N^2/4$ 次比较以及 ~ $N^2/4$ 次交换。最坏情况下需要 ~ $N^2/2$ 次比较和 ~ $N^2/2$ 次交换，最好情况下需要 $N-1$ 次比较和 0 次交换。

证明。和命题 A 一样，通过一个 $N \times N$ 的轨迹表可以很容易就得到交换和比较的次数。最坏情况下对角线之下所有的元素都需要移动位置，最好情况下都不需要。对于随机排列的数组，在平均情况下每个元素都可能向后移动半个数组的长度，因此交换总数是对角线之下的元素总数的二分之一。

比较的总次数是交换的次数加上一个额外的项，该项为 N 减去被插入的元素正好是已知的最小元素的次数。在最坏情况下（逆序数组），这一项相对于总数可以忽略不计；在最好情况下（数组已经有序），这一项等于 $N-1$。

插入排序对于实际应用中常见的某些类型的非随机数组很有效。例如，正如刚才所提到的，想想当你用插入排序对一个有序数组进行排序时会发生什么。插入排序能够立即发现每个元素都已经在合适的位置之上，它的运行时间也是线性的（对于这种数组，选择排序的运行时间是平方级别的）。对于所有主键都相同的数组也会出现相同的情况（因此命题 B 的条件之一就是主键不重复）。

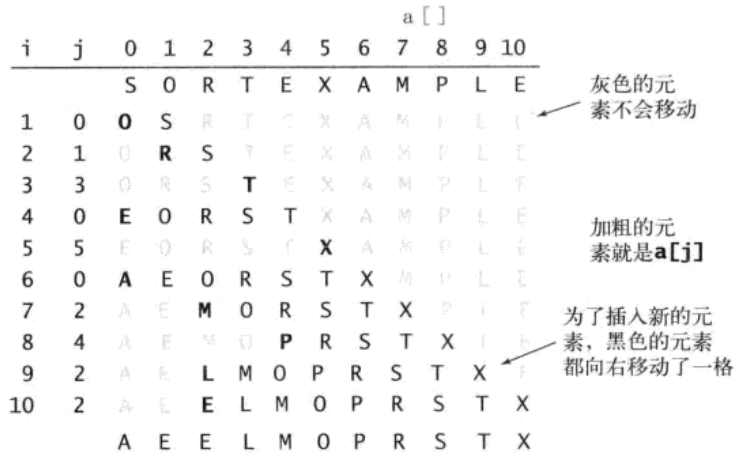
250

算法 2.2 插入排序

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {  // 将a[]按升序排列
      int N = a.length;
      for (int i = 1; i < N; i++)
      {  // 将 a[i] 插入到 a[i-1]、a[i-2]、a[i-3]...之中
         for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
```

```
                exch(a, j, j-1);
            }
        }
        // less()、exch()、isSorted()和main()方法见“排序算法类模板”
    }
```

对于0到N-1之间的每一个i，将a[i]与a[0]到a[i-1]中比它小的所有元素依次有序地交换。在索引i由左向右变化的过程中，它左侧的元素总是有序的，所以当i到达数组的右端时排序就完成了。



i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	**O**	S									
2	1		**R**	S								
3	3				**T**							
4	0	**E**	O	R	S	T						
5	5					**X**						
6	0	**A**	E	O	R	S	T	X				
7	2			**M**	O	R	S	T	X			
8	4				**P**	R	S	T	X			
9	2			**L**	M	O	P	R	S	T	X	
10	2			**E**	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

插入排序的轨迹（每次插入后的数组内容）

251

我们要考虑的更一般的情况是部分有序的数组。倒置指的是数组中的两个顺序颠倒的元素。比如EXAMPLE中有11对倒置：E-A、X-A、X-M、X-P、X-L、X-E、M-L、M-E、P-L、P-E以及L-E。如果数组中倒置的数量小于数组大小的某个倍数，那么我们说这个数组是部分有序的。下面是几种典型的部分有序的数组：

- 数组中每个元素距离它的最终位置都不远；
- 一个有序的大数组接一个小数组；
- 数组中只有几个元素的位置不正确。

插入排序对这样的数组很有效，而选择排序则不然。事实上，当倒置的数量很少时，插入排序很可能比本章中的其他任何算法都要快。

命题C。插入排序需要的交换操作和数组中倒置的数量相同，需要的比较次数大于等于倒置的数量，小于等于倒置的数量加上数组的大小再减一。

证明。每次交换都改变了两个顺序颠倒的元素的位置，相当于减少了一对倒置，当倒置数量为0时，排序就完成了。每次交换都对应着一次比较，且1到N-1之间的每个i都可能需要一次额外的比较（在a[i]没有达到数组的左端时）。

要大幅提高插入排序的速度并不难，只需要在内循环中将较大的元素都向右移动而不总是交换两个元素（这样访问数组的次数就能减半）。我们把这项改进留做一个练习（请见练习2.1.25）。

总的来说，插入排序对于部分有序的数组十分高效，也很适合小规模数组。这很重要，因为这些类型的数组在实际应用中经常出现，而且它们也是高级排序算法的中间过程。我们会在学习高级排序算法时再次接触到插入排序。

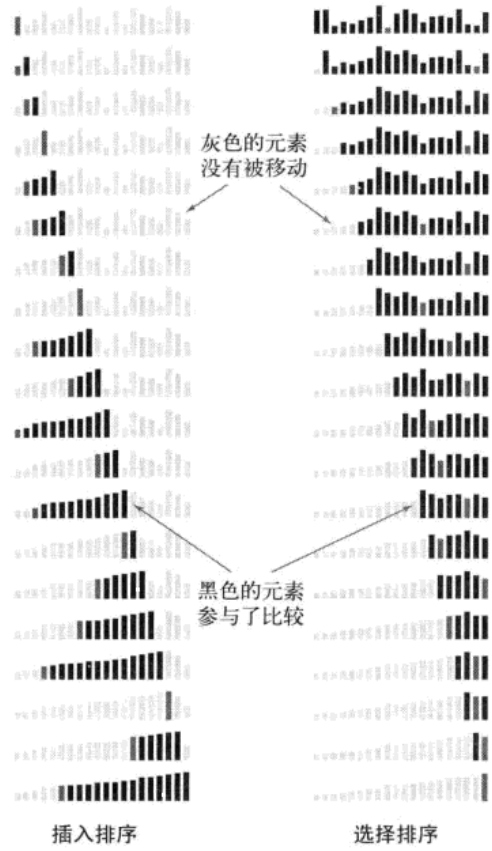


图 2.1.1 初级排序算法的可视轨迹图（另见彩插）

2.1.4 排序算法的可视化

在本章中我们会使用一种简单的图示来帮助我们说明排序算法的性质。我们没有使用字母、数字或是单词这样的键值来跟踪排序的进程，而使用了棒状图，并以它们的高矮来排序。这种表示方法的好处是能够使排序过程一目了然。

如图 2.1.1 所示，插入排序不会访问索引右侧的元素，而选择排序不会访问索引左侧的元素。另外，在这种可视化的轨迹图中可以看到，因为插入排序不会移动比被插入的元素更小的元素，它所需的比较次数平均只有选择排序的一半。

用我们的 StdDraw 库画出一张可视轨迹图并不比追踪一次算法的运行轨迹难多少。将 `Double` 值排序，并在适当的时候指示算法调用 `show()` 方法（和追踪算法的轨迹时一样），然后开发一个使用 StdDraw 来绘制棒状图而不是打印结果的 `show()` 方法。最复杂的部分是设置 y 轴的比例以使轨迹的线条符合预期的顺序。请通过练习 2.1.18 来更好地理解可视轨迹图的价值和使用。

将轨迹变成动画，理解起来就更加简单，这样可以看到动态演化到有序状态的过程。产生轨迹动画的过程本质上和上一段所描述的相同，但不需要担心 y 轴的问题（只需每次擦除窗口中的内容并重绘棒状图即可）。尽管我们无法在书中展现这些动画，它们对于理解算法的工作原理也很有帮助，你能通过练习 2.1.17 体会这一点。

252
~
253

2.1.5 比较两种排序算法

现在我们已经实现了两种排序算法，我们很自然地想知道选择排序（算法 2.1）和插入排序（算法 2.2）哪种更快。这个问题在学习算法的过程中会反复出现，也是本书的重点之一。我们已经在第 1 章中讨论过一些基本的概念，这里我们第一次用实践说明我们解决这个问题的办法。一般来说，根据 1.4 节所介绍的方法，我们将通过以下步骤比较两个算法：

- 实现并调试它们；
- 分析它们的基本性质；
- 对它们的相对性能作出猜想；

❑ 用实验验证我们的猜想。

这些步骤都是经过时间检验的科学方法，只是现在是运用在算法研究之上。

现在，算法 2.1 和算法 2.2 表示已经实现了第一步，命题 A、命题 B 和命题 C 组成了第二步，下面的性质 D 将是第三步，之后“比较两种排序算法”的 SortCompare 类将会完成第四步。这些行为都是紧密相关的。

在这些简洁的步骤之下是大量的算法实现、调试分析和测试工作。每个程序员都知道只有经过长期的调试和改进才能得到这样的代码，每个数学家都知道正确分析的难度，每个科学家也都知道从提出猜想到设计并执行实验来验证它们是多么费心。只有研究那些最重要的算法的专家才会经历完整的研究过程，但每个使用算法的程序员都应该了解算法的性能特性背后的科学过程。

实现了算法之后，下一步我们需要确定一个适当的输入模型。对于排序，命题 A、命题 B 和命题 C 用到的自然输入模型假设数组中的元素随机排序，且主键值不会重复。对于有很多重复主键的应用来说，我们需要一个更加复杂的模型。

如何估计插入排序和选择排序在随机排序数组下的性能呢？通过算法 2.1 和算法 2.2 以及命题 A、命题 B 和命题 C 可以发现，对于随机排序数组，两者的运行时间都是平方级别的。也就是说，在这种输入下插入排序的运行时间和 N^2 乘以一个小常数成正比，选择排序的运行时间和 N^2 乘以另一个小常数成比例。这两个常数的值取决于所使用的计算机中比较和交换元素的成本。对于许多数据类型和一般的计算机，可以假设这些成本是相近的（但我们也会看到一些大不相同的例外）。因此我们直接得出了以下猜想。

254

性质 D。对于随机排序的无重复主键的数组，插入排序和选择排序的运行时间是平方级别的，两者之比应该是一个较小的常数。

例证。这个结论在过去的半个世纪中已经在许多不同类型的计算机上经过了验证。在 1980 年本书第 1 版完成之时插入排序就比选择排序快一倍，现在仍然是这样，尽管那时这些算法将 10 万条数据排序需要几个小时而现在只需要几秒钟。在你的计算机上插入排序也比选择排序快一些吗？可以通过 SortCompare 类来检测。它会使用由命令行参数指定的排序算法名称所对应的 sort() 方法进行指定次数的实验（将指定大小的数组排序），并打印出所观察到的各种算法的运行时间的比例。

为了证明这一点，我们用 SortCompare（见“比较两种排序算法”）来做几次实验。我们使用 Stopwatch 来计时，右侧的 time() 函数的任务是调用本章中的几种简单排序算法。

随机数组的输入模型由 SortCompare 类中的 timeRandomInput() 方法实现。这个方法会生成随机的 Double 值，将它们排

```
public static double time(String alg, Comparable[] a)
{
   Stopwatch timer = new Stopwatch();
   if (alg.equals("Insertion")) Insertion.sort(a);
   if (alg.equals("Selection")) Selection.sort(a);
   if (alg.equals("Shell"))     Shell.sort(a);
   if (alg.equals("Merge"))     Merge.sort(a);
   if (alg.equals("Quick"))     Quick.sort(a);
   if (alg.equals("Heap"))      Heap.sort(a);
   return timer.elapsedTime();
}
```

针对给定输入，为本章中的一种排序算法计时

序，并返回指定次测试的总时间。使用 0.0 至 1.0 之间的随机 Double 值比使用类似于 StdRandom.shuffle() 的库函数更简单有效，因为这样几乎不可能产生相等的主键值（请见练习 2.5.31）。如第 1 章中所讨论的，用命令行参数指定重复次数的好处是能够运行大量的测试（测试次数越多，每遍测试所需的平均时间就越接近于真实的平均数据）并且能够减小系统本身的影响。你应该在自己的计算机上用 SortCompare 进行实验，来了解关于插入排序和选择排序的结论是否成立。

比较两种排序算法

```
public class SortCompare
{
   public static double time(String alg, Double[] a)
   {  /* 请见前面的正文 */  }

   public static double timeRandomInput(String alg, int N, int T)
   {  // 使用算法alg将T个长度为N的数组排序
      double total = 0.0;
      Double[] a = new Double[N];
      for (int t = 0; t < T; t++)
      {  // 进行一次测试（生成一个数组并排序）
         for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform();
         total += time(alg, a);
      }
      return total;
   }

   public static void main(String[] args)
   {
      String alg1 = args[0];
      String alg2 = args[1];
      int N = Integer.parseInt(args[2]);
      int T = Integer.parseInt(args[3]);
      double t1 = timeRandomInput(alg1, N, T); // 算法1的总时间
      double t2 = timeRandomInput(alg2, N, T); // 算法2的总时间
      StdOut.printf("For %d random Doubles\n    %s is", N, alg1);
      StdOut.printf(" %.1f times faster than %s\n", t2/t1, alg2);
   }
}
```

这个用例会运行由前两个命令行参数指定的排序算法，对长度为 N（由第三个参数指定）的 Double 型随机数组进行排序，元素值均在 0.0 到 1.0 之间，重复 T 次（由第四个参数指定），然后输出总运行时间的比例。

```
% java SortCompare Insertion Selection 1000 100
For 1000 random Doubles
  Insertion is 1.7 times faster than Selection
```

255 ~ 256

我们故意将性质 D 描述得不够明确——没有说明那个小常量的值，以及对比较和交换的成本相近的假设，这样性质 D 才能广泛适用于各种情况。可能的话，我们会尽量用这样的语言来抓住我们所研究的每个算法的性能的本质。如第 1 章中讨论的那样，我们提出的每个性质都需要在特定的场景中进行科学测试，也许还需要用一个基于相关命题（数学定理）的猜想进行补充。

对于实际应用，还有一个很重要的步骤，*那就是用实际数据在实验中验证我们的猜想*。我们会在2.5节和练习中再考虑这一点。在这种情况下，当主键有重复或是排列不随机，性质D就可能会不成立。可以使用 `StdRandom.shuffle()` 来将一个数组打乱，但有大量重复主键的情况则需要更加细致的分析。

我们对算法分析的讨论是抛砖引玉，而非盖棺定论。如果你想到了关于算法性能的其他问题，可以用 `SortCompare` 等工具来研究它，后面的练习为你提供了许多机会。

插入排序和选择排序的性能比较就讨论到这里，还存在许多比它们快成千上万倍的算法，我们对此会更感兴趣。当然，仍然有必要学习这些初级算法，因为：

- 它们帮助我们建立了一些基本的规则；
- 它们展示了一些性能基准；
- 在某些特殊情况下它们也是很好的选择；
- 它们是开发更强大的排序算法的基石。

因此，不止是排序，对于本书中的每个问题我们都会沿用这种方式，首先学习的就是最初级的相关算法。`SortCompare` 这样的程序对于这种渐进式的算法研究十分重要。每一步，我们都能用这
257 类程序来了解新的或是改进后的算法的性能是否产生了预期的进步。

2.1.6 希尔排序

为了展示初级排序算法性质的价值，接下来我们将学习一种基于插入排序的快速的排序算法。对于大规模乱序数组插入排序很慢，因为它只会交换相邻的元素，因此元素只能一点一点地从数组的一端移动到另一端。例如，如果主键最小的元素正好在数组的尽头，要将它挪到正确的位置就需要 $N-1$ 次移动。希尔排序为了加快速度简单地改进了插入排序，交换不相邻的元素以对数组的局部进行排序，并最终用插入排序将局部有序的数组排序。

希尔排序的思想是使数组中任意间隔为h的元素都是有序的。这样的数组被称为*h有序数组*。换句话说，一个*h有序数组*就是h个互相独立的有序数组编织在一起组成的一个数组（见图2.1.2）。在进行排序时，如果h很大，我们就能将元素移动到很远的地方，为实现更小的h有序创造方便。用这种方式，对于任意以1结尾的h序列，我们都能够将数组排序。这就是希尔排序。算法2.3的实现使用了序列 $1/2(3^k-1)$，从 $N/3$ 开始递减至1。我们把这个序列称为*递增序列*。算法2.3实时计算了它的*递增序列*，另一种方式是将递增序列存储在一个数组中。

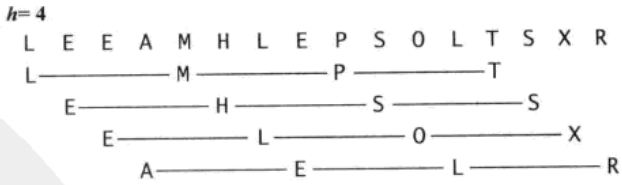


图2.1.2 一个h有序数组即一个由h个有序子数组组成的数组

实现希尔排序的一种方法是对于每个h，用插入排序将h个子数组独立地排序。但因为子数组是相互独立的，一个更简单的方法是在h-子数组中将每个元素交换到比它大的元素之前去（将比它大的元素向右移动一格）。只需要在插入排序的代码中将移动元素的距离由1改为h即可。这样，希尔排序的实现就转化为了一个类似于插入排序但使用不同增量的过程。

希尔排序更高效的原因是它权衡了子数组的规模和有序性。排序之初，各个子数组都很短，排

序之后子数组都是部分有序的，这两种情况都很适合插入排序。子数组部分有序的程度取决于递增序列的选择。透彻理解希尔排序的性能至今仍然是一项挑战。实际上，算法 2.3 是我们唯一无法准确描述其对于乱序的数组的性能特征的排序方法。

算法 2.3 希尔排序

```
public class Shell
{
   public static void sort(Comparable[] a)
   {  // 将a[]按升序排列
      int N = a.length;
      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...
      while (h >= 1)
      {  // 将数组变为h有序
         for (int i = h; i < N; i++)
         {  // 将a[i]插入到a[i-h], a[i-2*h], a[i-3*h]... 之中
            for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }
         h = h/3;
      }
   }

   // less()、exch()、isSorted()和main()方法见“排序算法类模板”

}
```

如果我们在插入排序（算法 2.2）中加入一个外循环来将 h 按照递增序列递减，我们就能得到这个简洁的希尔排序。增幅 h 的初始值是数组长度乘以一个常数因子，最小为 1。

```
% java SortCompare Shell Insertion 100000 100
For 100000 random Doubles
  Shell is 600 times faster than Insertion
```

输入	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

希尔排序的轨迹（每遍排序后的数组内容）

258 ≀ 259

如何选择递增序列呢？要回答这个问题并不简单。算法的性能不仅取决于 h，还取决于 h 之间的数学性质，比如它们的公因子等。有很多论文研究了各种不同的递增序列，但都无法证明某个序列是“最好的”。算法 2.3 中递增序列的计算和使用都很简单，和复杂递增序列的性能接近。但可以证明复杂的序列在最坏情况下的性能要好于我们所使用的递增序列。更加优秀的递增序列有待我们去发现。

和选择排序以及插入排序形成对比的是，希尔排序也可以用于大型数组。它对任意排序（不一定是随机的）的数组表现也很好。实际上，对于一个给定的递增序列，构造一个使希尔排序运行缓慢的

数组并不容易。希尔排序的轨迹如图 2.1.3 所示，可视轨迹如图 2.1.4 所示。

输入	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	**P**	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	**L**	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	**E**
4-sort	**L**	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	**S**	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	**O**	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	**R**	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	**T**	E	X	A	M	S	L	E
	L	**E**	E	L	P	H	O	R	T	S	X	A	M	S	L	E
	L	E	E	L	P	H	O	R	T	S	**X**	A	M	S	L	E
	L	E	E	**A**	P	H	O	L	T	S	X	R	M	S	L	E
	L	E	E	A	**M**	H	O	L	P	S	X	R	T	S	L	E
	L	E	E	A	M	H	O	L	P	S	X	R	T	**S**	L	E
	L	E	E	A	M	H	**L**	L	P	S	**O**	R	T	S	X	E
	L	E	E	A	M	H	L	**E**	P	S	O	L	T	S	X	R
1-sort	**E**	L	E	A	M	H	L	E	P	S	O	L	T	S	X	R
	E	**E**	L	A	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	**M**	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	**H**	L	M	L	E	P	S	O	L	T	S	X	R
	A	E	E	H	L	**L**	M	E	P	S	O	L	T	S	X	R
	A	E	E	**E**	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	**P**	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	**S**	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	**O**	P	S	L	T	S	X	R
	A	E	E	E	H	L	L	**L**	M	O	P	S	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	**T**	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	**S**	T	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	S	**T**	X	R
	A	E	E	E	H	L	L	L	M	O	P	**R**	S	S	T	X
结果	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

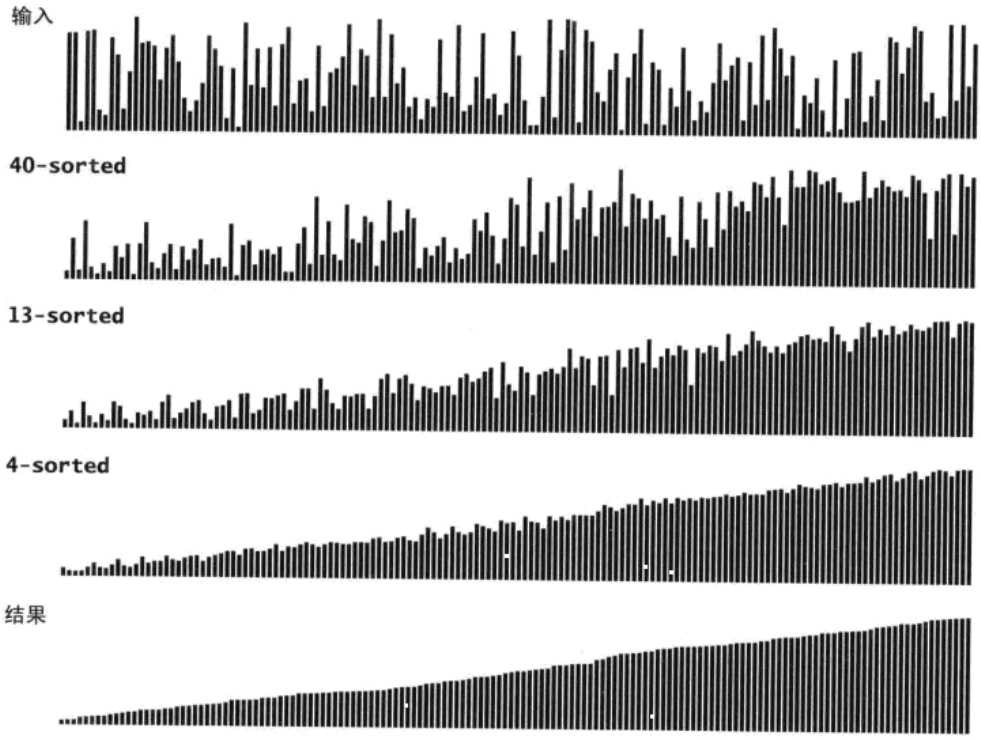
260

图 2.1.3 希尔排序的详细轨迹（各种插入）

通过 SortCompare 可以看到，希尔排序比插入排序和选择排序要快得多，并且数组越大，优势越大。在继续学习之前，请在你的计算机上用 SortCompare 比较一下希尔排序和插入排序以及选择排序的性能，数组的大小按照 2 的幂次递增（见练习 2.1.27）。你会看到希尔排序能够解决一些初级排序算法无能为力的问题。这个例子是我们第一次用实际应用说明一个贯穿本书的重要理念：*通过提升速度来解决其他方式无法解决的问题是研究算法的设计和性能的主要原因之一。*

研究希尔排序性能需要的数学论证超出了本书范围。如果你不相信，可以从证明下面这一点开始：*当一个“h 有序”的数组按照增幅 k 排序之后，它仍然是“h 有序”的。*至于算法 2.3 的性能，目前最重要的结论是它的运行时间达不到平方级别。例如，已知在最坏的情况下算法 2.3 的比较次数和 $N^{3/2}$ 成正比。有意思的是，由插入排序到希尔排序，一个小小的改变就突破了平方级别的运行时间

的屏障。这正是许多算法设计问题想要达到的目标。



261

图 2.1.4 希尔排序的可视轨迹

在输入随机排序数组的情况下，我们在数学上还不知道希尔排序所需要的平均比较次数。人们发明了很多递增序列来渐进式地改进最坏情况下所需的比较次数（$N^{4/3}$, $N^{5/4}$, $N^{6/5}\cdots$），但这些结论大多只有学术意义，因为对于实际应用中的 N 来说它们的递增序列的生成函数（以及与 N 乘以一个常数因子）之间的区别并不明显。

在实际应用中，使用算法 2.3 中的递增序列基本就足够了（或者是本节最后的练习中提供的一个递增序列，它可能可以将性能改进 20% ~ 40%）。另外，很容易就能验证下面这个猜想。

性质 E。 使用递增序列 1, 4, 13, 40, 121, 364…的希尔排序所需的比较次数不会超出 N 的若干倍乘以递增序列的长度。

例证。 记录算法 2.3 中比较的数量并将其除以使用的序列长度是一道简单的练习（请见练习 2.1.12）。大量的实验证明平均每个增幅所带来的比较次数约为 $N^{1/5}$，但只有在 N 很大的时候这个增长幅度才会变得明显。这个性质似乎也和输入模型无关。

有经验的程序员有时会选择希尔排序，因为对于中等大小的数组它的运行时间是可以接受的。它的代码量很小，且不需要使用额外的内存空间。在下面的几节中我们会看到更加高效的算法，但除了对于很大的 N，它们可能只会比希尔排序快两倍（可能还达不到），而且更复杂。如果你需要解决一个排序问题而又没有系统排序函数可用（例如直接接触硬件或是运行于嵌入式系统中的代码），可以先用希尔排序，然后再考虑是否值得将它替换为更加复杂的排序算法。

262

答疑

问 排序看起来是个很简单的问题，我们用计算机不是可以做很多更有意思的事情吗？

答 也许吧，但快速的排序算法才使得那些更有意思的事情成为可能。在2.5节以及全书的其他章节你都可以找到很多这样的例子。排序算法今天仍然值得我们学习是因为它易于理解，你能从中领会到许多精妙之处。

问 为什么有这么多排序算法？

答 原因之一是许多排序算法的性能都和输入模型有很大的关系，因此不同的算法适用于不同应用场景中的不同输入。例如，对于部分有序和小规模的数组应该选择插入排序。其他限制条件，例如空间和重复的主键，也都是需要考虑的因素。我们将会在2.5节中再次讨论这个问题。

问 为什么要使用 `less()` 和 `exch()` 这些不起眼的辅助函数？

答 它们抽象了所有排序算法都会用到的共同操作，这种抽象使得代码更便于理解。而且它们增强了代码的可移植性。例如，算法2.1和算法2.2中的大部分代码在其他几种编程语言中也是可以执行的。即使是在Java中，只要将 `less()` 实现为 `v < w`，这些算法的代码就可以将不支持 `Comparable` 接口的基本数据类型排序了。

问 当我运行 `SortCompare` 时，每次的结果都不一样（而且和书上的也不相同），为什么？

答 对于初学者，你的计算机和我们的计算机不同，操作系统、Java运行时环境等都不一样。这些不同可能导致算法代码生成的机器码不同。每次运行所得结果不同的原因可能在于当时运行的其他程序或是很多其他原因。大量的重复实验可以淡化这种干扰，我们的经验是现如今算法性能的微小差异很难观察。这就是我们要关注较大差异的原因。

263

练习

2.1.1 按照算法2.1所示轨迹的格式给出选择排序是如何将数组 E A S Y Q U E S T I O N 排序的。

2.1.2 在选择排序中，一个元素最多可能会被交换多少次？平均可能会被交换多少次？

2.1.3 构造一个含有 N 个元素的数组，使选择排序（算法2.1）运行过程中 `a[j] < a[min]`（由此 `min` 会不断更新）成功的次数最大。

2.1.4 按照算法2.2所示轨迹的格式给出插入排序是如何将数组 E A S Y Q U E S T I O N 排序的。

2.1.5 构造一个含有 N 个元素的数组，使插入排序（算法2.2）运行过程中内循环（`for`）的两个判断结果总是假。

2.1.6 在所有的主键都相同时，选择排序和插入排序谁更快？

2.1.7 对于逆序数组，选择排序和插入排序谁更快？

2.1.8 假设元素只可能有三种值，使用插入排序处理这样一个随机数组的运行时间是线性的还是平方级别的？或是介于两者之间？

2.1.9 按照算法2.3所示轨迹的格式给出希尔排序是如何将数组 E A S Y S H E L L S O R T Q U E S T I O N 排序的。

2.1.10 在希尔排序中为什么在实现 `h` 有序时不使用选择排序？

2.1.11 将希尔排序中实时计算递增序列改为预先计算并存储在一个数组中。

2.1.12 令希尔排序打印出递增序列的每个元素所带来的比较次数和数组大小的比值。编写一个测试用例对随机 `Double` 数组进行希尔排序，验证该值是一个小常数，数组大小按照10的幂次递增，

不小于 100。 264

提高题

2.1.13 *纸牌排序。*说说你会如何将一副扑克牌按花色排序（花色顺序是黑桃、红桃、梅花和方片），限制条件是所有牌都是背面朝上排成一列，而你一次只能翻看两张牌或者交换两张牌（保持背面朝上）。

2.1.14 *出列排序。*说说你会如何将一副扑克牌排序，限制条件是只能查看最上面的两张牌，交换最上面的两张牌，或是将最上面的一张牌放到这摞牌的最下面。

2.1.15 *昂贵的交换。*一家货运公司的一位职员得到了一项任务，需要将若干大货箱按照发货时间摆放。比较发货时间很容易（对照标签即可），但将两个货箱交换位置则很困难（移动麻烦）。仓库已经快满了，只有一个空闲的仓位。这位职员应该使用哪种排序算法呢？

2.1.16 *验证。*编写一个 `check()` 方法，调用 `sort()` 对任意数组排序。如果排序成功而且数组中的所有对象均没有被修改则返回 `true`，否则返回 `false`。不要假设 `sort()` 只能通过 `exch()` 来移动数据，可以信任并使用 `Arrays.sort()`。

2.1.17 *动画。*修改插入排序和选择排序的代码，使之将数组内容绘制成正文中所示的棒状图。在每一轮排序后重绘图片来产生动画效果，并以一张“有序”的图片作为结束，即所有圆棒均已按照高度有序排列。*提示：*使用类似于正文中的用例来随机生成 `Double` 值，在排序代码的适当位置调用 `show()` 方法，并在 `show()` 方法中清理画布并绘制棒状图。

2.1.18 *可视轨迹。*修改你为上一题给出的解答，为插入排序和选择排序生成和正文中类似的可视轨迹。*提示：*使用 `setYscale()` 函数是一个明智的选择。*附加题：*添加必要的代码，与正文中的图片一样用红色和灰色强调不同角色的元素。

2.1.19 *希尔排序的最坏情况。*用 1 到 100 构造一个含有 100 个元素的数组并用希尔排序和递增序列 1 4 13 40 对其排序，使比较的次数尽可能多。

2.1.20 *希尔排序的最好情况。*最好情况是什么？证明你的结论。 265

2.1.21 *可比较的交易。*用我们的 `Date` 类（请见 2.1.1.4 节）作为模板扩展你的 `Transaction` 类（请见练习 1.2.13），实现 `Comparable` 接口，使交易能够按照金额排序。

解答：

```
public class Transaction implements Comparable<Transaction>
{
   ...
   private final double amount;
   ...
   public int compareTo(Transaction that)
   {
      if (this.amount > that.amount) return +1;
      if (this.amount < that.amount) return -1;
      return 0;
   }
   ...
}
```

2.1.22 *事务排序测试用例。*编写一个 `SortTransaction` 类，在静态方法 `main()` 中从标准输入读取一系列事务，将它们排序并在标准输出中打印结果（请见练习 1.3.17）。

解答：

```
public class SortTransactions
{
   public static Transaction[] readTransactions()
   {  // 请见练习 1.3.17  }
   public static void main(String[] args)
   {
      Transaction[] transactions = readTransactions();
      Shell.sort(transactions);
      for (Transaction t : transactions)
         StdOut.println(t);
   }
}
```

266

实验题

2.1.23 纸牌排序。请几位朋友分别将一副扑克牌排序（见练习 2.1.13）。仔细观察并记录他们所使用的方法。

2.1.24 插入排序的哨兵。在插入排序的实现中先找出最小的元素并将其置于数组的最左边，这样就能去掉内循环的判断条件 j>0。使用 SortCompare 来评估这种做法的效果。注意：这是一种常见的规避边界测试的方法，能够省略判断条件的元素通常被称为哨兵。

2.1.25 不需要交换的插入排序。在插入排序的实现中使较大元素右移一位只需要访问一次数组（而不用使用 exch()）。使用 SortCompare 来评估这种做法的效果。

2.1.26 原始数据类型。编写一个能够处理 int 值的插入排序的新版本，比较它和正文中所给出的实现（能够隐式地用自动装箱和拆箱转换 Integer 值并排序）的性能。

2.1.27 希尔排序的用时是次平方级的。在你的计算机上用 SortCompare 比较希尔排序和插入排序以及选择排序。测试数组的大小按照 2 的幂次递增，从 128 开始。

2.1.28 相等的主键。对于主键仅可能取两种值的数组，评估和验证插入排序和选择排序的性能，假设两种主键值出现的概率相同。

2.1.29 希尔排序的递增序列。通过实验比较算法 2.3 中所使用的递增序列和递增序列 1，5，19，41，109，209，505，929，2161，3905，8929，16 001，36 289，64 769，146 305，260 609（这是通过序列 $9\times4^k-9\times2^k+1$ 和 $4^k-3\times2^k+1$ 综合得到的）。可以参考练习 2.1.11。

2.1.30 几何级数递增序列。通过实验找到一个 t，使得对于大小为 $N=10^6$ 的任意随机数组，使用递增序列 1，$\lfloor t\rfloor$，$\lfloor t^2\rfloor$，$\lfloor t^3\rfloor$，$\lfloor t^4\rfloor$，…的希尔排序的运行时间最短。给出你能找到的三个最佳 t 值以及相应的递增序列。

267

以下练习描述的是各种用于评估排序算法的测试用例。它们的作用是用随机数据帮助你增进对性能特性的理解。随着命令行指定的实验次数的增大，可以和 SortCompare 一样在它们中使用 time() 函数来得到更精确的结果。在以后的几节中我们会使用这些练习来评估更加复杂的算法。

2.1.31 双倍测试。编写一个能够对排序算法进行双倍测试的用例。数组规模 N 的起始值为 1000，排序后打印 N、估计排序用时、实际排序用时以及在 N 增倍之后两次用时的比例。用这段程序验证在随机输入模型下插入排序和选择排序的运行时间都是平方级别的。对希尔排序的性能作出猜想并验证你的猜想。

2.1.32 运行时间曲线图。编写一个测试用例，使用 StdDraw 在各种不同规模的随机输入下将算法的平均运行时间绘制成一张曲线图。可能需要添加一两个命令行参数，请尽量设计一个实用的工具。

2.1.33 分布图。对于你为练习 2.1.33 给出的测试用例，在一个无穷循环中调用 sort() 方法将由第三个

命令行参数指定大小的数组排序，记录每次排序的用时并使用 StdDraw 在图上画出所有平均运行时间，应该能够得到一张运行时间的分布图。

2.1.34 *罕见情况。*编写一个测试用例，调用 sort() 方法对实际应用中可能出现困难或极端情况的数组进行排序。比如，数组可能已经是有序的，或是逆序的，数组的所有主键相同，数组的主键只有两种值，大小为 0 或是 1 的数组。

2.1.35 *不均匀的概率分布。*编写一个测试用例，使用非均匀分布的概率来生成随机排列的数据，包括：

- 高斯分布；
- 泊松分布；
- 几何分布；
- 离散分布（一种特殊情况请见练习 2.1.28）。

评估并验证这些输入数据对本节讨论的算法的性能的影响。 268

2.1.36 *不均匀的数据。*编写一个测试用例，生成不均匀的测试数据，包括：

- 一半数据是 0，一半是 1；
- 一半数据是 0，1/4 是 1，1/4 是 2，以此类推；
- 一半数据是 0，一半是随机 int 值。

评估并验证这些输入数据对本节讨论的算法的性能的影响。

2.1.37 *部分有序。*编写一个测试用例，生成部分有序的数组，包括：

- 95% 有序，其余部分为随机值；
- 所有的元素和它们的正确位置的距离都不超过 10；
- 5% 的元素随机分布在整个数组中，剩下的数据都是有序的。

评估并验证这些输入数据对本节讨论的算法的性能的影响。

2.1.38 *不同类型的元素。*编写一个测试用例，生成由多种数据类型元素组成的数组，元素的主键值随机，包括：

- 每个元素的主键均为 String 类型（至少长 10 个字符），并含有一个 double 值；
- 每个元素的主键均为 double 类型，并含有 10 个 String 值（每个都至少长 10 个字符）；
- 每个元素的主键均为 int 类型，并含有一个 int[20] 值

评估并验证这些输入数据对本节讨论的算法的性能的影响。 269

2.2 归并排序

在本节中我们所讨论的算法都基于*归并*这个简单的操作，即将两个有序的数组归并成一个更大的有序数组。很快人们就根据这个操作发明了一种简单的递归排序算法：*归并排序*。要将一个数组排序，可以先（递归地）将它分成两半分别排序，然后将结果归并起来。你将会看到，归并排序最吸引人的性质是它能够保证将任意长度为 N 的数组排序所需时间和 $N\log N$ 成正比；它的主要缺点则是它所需的额外空间和 N 成正比。简单的归并排序如图 2.2.1 所示。



图 2.2.1 归并排序示意图

2.2.1 原地归并的抽象方法

实现归并的一种直截了当的办法是将两个不同的有序数组归并到第三个数组中，两个数组中的元素应该都实现了 Comparable 接口。实现的方法很简单，创建一个适当大小的数组然后将两个输入数组中的元素一个个从小到大放入这个数组中。

但是，当用归并将一个大数组排序时，我们需要进行很多次归并，因此在每次归并时都创建一个新数组来存储排序结果会带来问题。我们更希望有一种能够在原地归并的方法，这样就可以先将前半部分排序，再将后半部分排序，然后在数组中移动元素而不需要使用额外的空间。你可以先停下来想想应该如何实现这一点，乍一看很容易做到，但实际上已有的实现都非常复杂，尤其是和使用额外空间的方法相比。

尽管如此，将原地归并抽象化仍然是有帮助的。与之对应的是我们的方法签名 merge(a, lo, mid, hi)，它会将子数组 a[lo..mid] 和 a[mid+1..hi] 归并成一个有序的数组并将结果存放在 a[lo..hi] 中。下面的代码只用几行就实现了这种归并。它将涉及的所有元素复制到一个辅助数组中，再把归并的结果放回原数组中。实现的另一种方法请见练习 2.2.10。

270

原地归并的抽象方法

```
public static void merge(Comparable[] a, int lo, int mid, int hi)
{  // 将a[lo..mid] 和 a[mid+1..hi] 归并
   int i = lo, j = mid+1;

   for (int k = lo; k <= hi; k++)  // 将a[lo..hi]复制到aux[lo..hi]
      aux[k] = a[k];

   for (int k = lo; k <= hi; k++)  // 归并回到a[lo..hi]
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi )              a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
}
```

该方法先将所有元素复制到 aux[] 中，然后再归并回 a[] 中。方法在归并时（第二个 for 循环）进行了 4 个条件判断：左半边用尽（取右半边的元素）、右半边用尽（取左半边的元素）、右半边

的当前元素小于左半边的当前元素（取右半边的元素）以及右半边的当前元素大于等于左半边的当前元素（取左半边的元素）。

		a[]												aux[]									
	k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9
输入		E	E	G	M	R	A	C	E	R	T			-	-	-	-	-	-	-	-	-	-
复制		E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	A	C	E	R	T
												0	5										
	0	**A**										0	6	E	E	G	M	R	**A**	C	E	R	T
	1	A	**C**									0	7	E	E	G	M	R		**C**	E	R	T
	2	A	C	**E**								1	7	**E**	E	G	M	R			**E**	R	T
	3	A	C	E	**E**							2	7		**E**	G	M	R			**E**	R	T
	4	A	C	E	E	**E**						2	8			**G**	M	R			**E**	R	T
	5	A	C	E	E	E	**G**					3	8			**G**	M	R				**R**	T
	6	A	C	E	E	E	G	**M**				4	8				**M**	R				**R**	T
	7	A	C	E	E	E	G	M	**R**			5	8					**R**				**R**	T
	8	A	C	E	E	E	G	M	R	**R**		5	9									**R**	T
	9	A	C	E	E	E	G	M	R	R	**T**	6	10										**T**
归并结果		A	C	E	E	E	G	M	R	R	T												

原地归并的抽象方法的轨迹 271

2.2.2 自顶向下的归并排序

算法 2.4 基于原地归并的抽象实现了另一种递归归并，这也是应用高效算法设计中分治思想的最典型的一个例子。这段递归代码是归纳证明算法能够正确地将数组排序的基础：如果它能将两个子数组排序，它就能够通过归并两个子数组来将整个数组排序。

算法 2.4 自顶向下的归并排序

```
public class Merge
{
   private static Comparable[] aux;       // 归并所需的辅助数组

   public static void sort(Comparable[] a)

   {
      aux = new Comparable[a.length];     // 一次性分配空间
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {  // 将数组a[lo..hi]排序
      if (hi <= lo) return;
      int mid = lo + (hi - lo)/2;
      sort(a, lo, mid);          // 将左半边排序
      sort(a, mid+1, hi);        // 将右半边排序
      merge(a, lo, mid, hi);  // 归并结果（代码见“原地归并的抽象方法”）
   }
}
```

要对子数组 a[lo..hi] 进行排序，先将它分为 a[lo..mid] 和 a[mid+1..hi] 两部分，分别通过递归调用将它们单独排序，最后将有序的子数组归并为最终的排序结果。

	a[]															
lo hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

自顶向下的归并排序中归并结果的轨迹

要理解归并排序就要仔细研究该方法调用的动态情况，如图 2.2.2 中的轨迹所示。要将 a[0..15] 排序，sort() 方法会调用自己将 a[0..7] 排序，再在其中调用自己将 a[0..3] 和 a[0..1] 排序。在将 a[0] 和 a[1] 分别排序之后，终于才会开始将 a[0] 和 a[1] 归并（简单起见，我们在轨迹中把对单个元素的数组进行排序的调用省略了）。第二次归并是 a[2] 和 a[3]，然后是 a[0..1] 和 a[2..3]，以此类推。从这段轨迹可以看到，sort() 方法的作用其实在于安排多次 merge() 方法调用的正确顺序。后面几节还会用到这个发现。

这段代码也是我们分析归并排序的运行时间的基础。因为归并排序是算法设计中分治思想的典型应用，我们会详细对它进行分析。

我们也可以通过图 2.2.3 所示的树状图来理解命题 F。每个结点都表示一个 sort() 方法通过 merge() 方法归并而成的子数组。这棵树正好有 n 层。对于 0 到 n–1 之间的任意 k，自顶向下的第 k 层有 2^k 个子数组，每个数组的长度为 2^{n-k}，归并最多需要 2^{n-k} 次比较。因此每层的比较次数为 $2^k \times 2^{n-k}=2^n$，n 层总共为 $n2^n=N\lg N$。

```
        sort(a, 0, 15)
将左半   sort(a, 0, 7)
部分排序   sort(a, 0, 3)
             sort(a, 0, 1)
               merge(a, 0, 0, 1)
             sort(a, 2, 3)
               merge(a, 2, 2, 3)
             merge(a, 0, 1, 3)
           sort(a, 4, 7)
             sort(a, 4, 5)
               merge(a, 4, 4, 5)
             sort(a, 6, 7)
               merge(a, 6, 6, 7)
             merge(a, 4, 5, 7)
           merge(a, 0, 3, 7)
将右半   sort(a, 8, 15)
部分排序   sort(a, 8, 11)
             sort(a, 8, 9)
               merge(a, 8, 8, 9)
             sort(a, 10, 11)
               merge(a, 10, 10, 11)
             merge(a, 8, 9, 11)
           sort(a, 12, 15)
             sort(a, 12, 13)
               merge(a, 12, 12, 13)
             sort(a, 14, 15)
               merge(a, 14, 14,15)
             merge(a, 12, 13, 15)
           merge(a, 8, 11, 15)
归并结果 merge(a, 0, 7, 15)
```

图 2.2.2 自顶向下的归并排序的调用轨迹

命题 F。对于长度为 N 的任意数组，自顶向下的归并排序需要 $\frac{1}{2}N\lg N$ 至 $N\lg N$ 次比较。

证明。令 $C(N)$ 表示将一个长度为 N 的数组排序时所需要的比较次数。我们有 $C(0)=C(1)=0$，对于 $N>0$，通过递归的 sort() 方法我们可以由相应的归纳关系得到比较次数的上限：

$$C(N) \leq C(\lfloor N/2 \rfloor) + C(\lceil N/2 \rceil) + N$$

右边的第一项是将数组的左半部分排序所用的比较次数，第二项是将数组的右半部分排序所用的比较次数，第三项是归并所用的比较次数。因为归并所需的比较次数最少为 $\lfloor N/2 \rfloor$，比较次数的下限是：

$$C(N) \geq C(\lfloor N/2 \rfloor) + C(\lceil N/2 \rceil) + \lfloor N/2 \rfloor$$

当 N 为 2 的幂（即 $N=2^n$）且等号成立时我们能够得到一个解。首先，因为 $\lfloor N/2 \rfloor = \lceil N/2 \rceil = 2^{n-1}$，可以得到：

$$C(2^n) = 2C(2^{n-1}) + 2^n$$

将两边同时除以 2^n 可得：

$$C(2^n)/2^n = C(2^{n-1}))/2^{n-1} + 1$$

用这个公式替换右边的第一项，可得：

$$C(2^n)/2^n = C(2^{n-2}))/2^{n-2} + 1 + 1$$

将上一步重复 $n-1$ 遍可得：

$$C(2^n)/2^n = C(2^0)/2^0 + n$$

将两边同时乘以 2^n 就可以解得：

$$C(N) = C(2^n) = n2^n = N\lg N$$

对于一般的 N，得到的准确值要更复杂一些。但对比较次数的上下界不等式使用相同的方法不难证明前面所述的对于任意 N 的结论。这个结论对于任意输入值和顺序都成立。

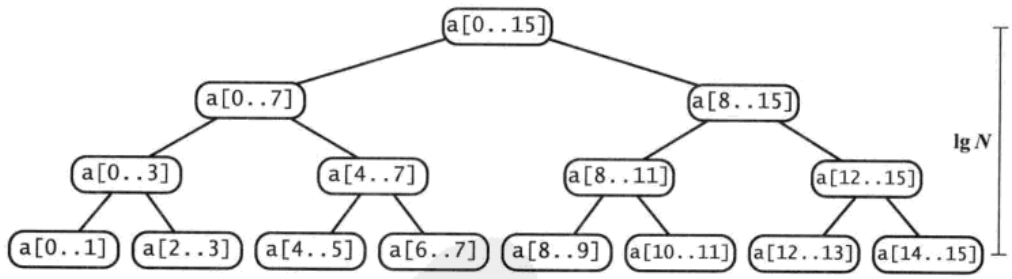


图 2.2.3 $N=16$ 时归并排序中子数组的依赖树

272 ≀ 274

命题 G。对于长度为 N 的任意数组，自顶向下的归并排序最多需要访问数组 $6N\lg N$ 次。

证明。每次归并最多需要访问数组 $6N$ 次（$2N$ 次用来复制，$2N$ 次用来将排好序的元素移动回去，另外最多比较 $2N$ 次），根据命题 F 即可得到这个命题的结果。

命题F和命题G告诉我们归并排序所需的时间和$N\lg N$成正比。这和2.1节所述的初级排序方法不可同日而语，它表明我们只需要比遍历整个数组多个对数因子的时间就能将一个庞大的数组排序。可以用归并排序处理数百万甚至更大规模的数组，这是插入排序或者选择排序做不到的。归并排序的主要缺点是辅助数组所使用的额外空间和N的大小成正比。另一方面，通过一些细致的思考我们还能够大幅度缩短归并排序的运行时间。

2.2.2.1 对小规模子数组使用插入排序

用不同的方法处理小规模问题能改进大多数递归算法的性能，因为递归会使小规模问题中方法的调用过于频繁，所以改进对它们的处理方法就能改进整个算法。对排序来说，我们已经知道插入排序（或者选择排序）非常简单，因此很可能在小数组上比归并排序更快。和之前一样，一幅可视轨迹图能够很好地说明归并排序的行为方式。图2.2.4中的可视轨迹图显示的是改良后的归并排序的所有操作。使用插入排序处理小规模的子数组（比如长度小于15）一般可以将归并排序的运行时间缩短10%～15%（请见练习2.2.23）。

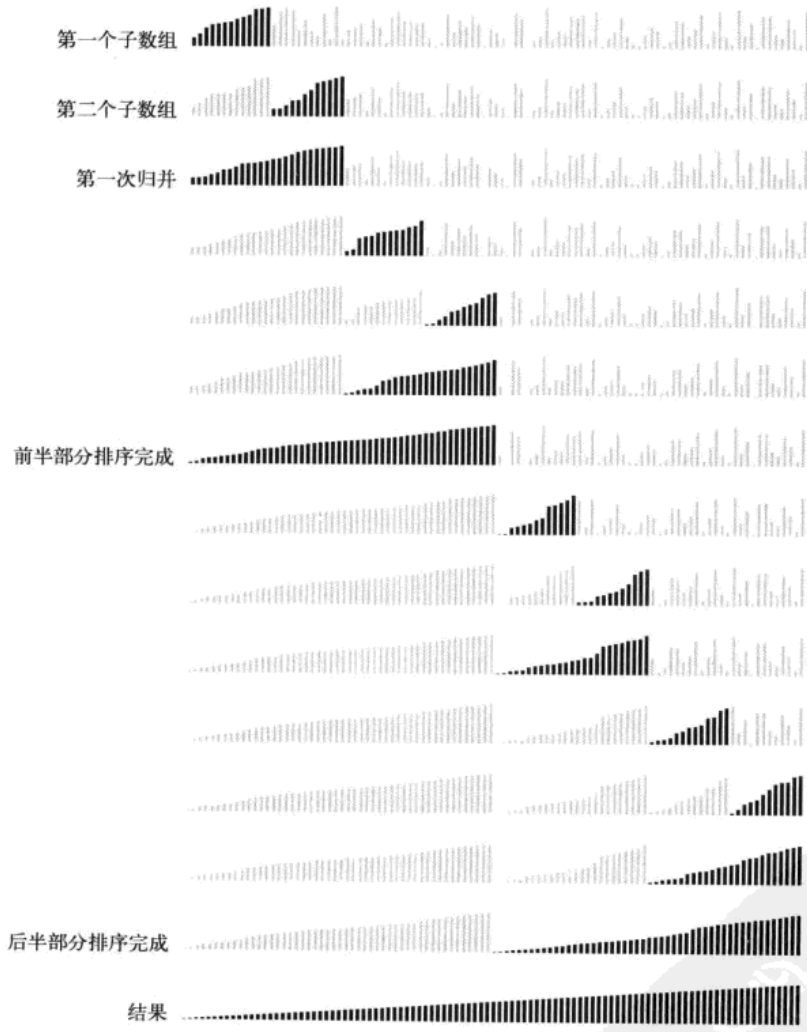


图2.2.4 改进了小规模子数组排序方法后的自顶向下的归并排序的可视轨迹

2.2.2.2 测试数组是否已经有序

我们可以添加一个判断条件，如果 a[mid] 小于等于 a[mid+1]，我们就认为数组已经是有序的并跳过 merge() 方法。这个改动不影响排序的递归调用，但是任意有序的子数组算法的运行时间就变为线性的了（请见练习 2.2.8）。

2.2.2.3 不将元素复制到辅助数组

我们可以节省将数组元素复制到用于归并的辅助数组所用的时间（但空间不行）。要做到这一点我们要调用两种排序方法，一种将数据从输入数组排序到辅助数组，一种将数据从辅助数组排序到输入数组。这种方法需要一些技巧，我们要在递归调用的每个层次交换输入数组和辅助数组的角色（请见练习 2.2.11）。

275 ~ 276

这里我们要重新强调第 1 章中提出的一个很容易遗忘的要点。在每一节中，我们会将书中的每个算法都看做某种应用的关键。但在整体上，我们希望学习的是为每种应用找到最合适的算法。我们并不是在推荐读者一定要实现所提到的这些改进方法，而是提醒大家不要对算法初始实现的性能盖棺定论。研究一个新问题时，最好的方法是先实现一个你能想到的最简单的程序，当它成为瓶颈的时候再继续改进它。实现那些只能把运行时间缩短某个常数因子的改进措施可能并不值得。你需要用实验来检验一项改进，正如本书中所有练习所演示的那样。

对于归并排序，刚才列出的三个建议都很容易实现且在应用归并排序时是十分有吸引力的——比如本章最后讨论的情况。

2.2.3 自底向上的归并排序

递归实现的归并排序是算法设计中分治思想的典型应用。我们将一个大问题分割成小问题分别解决，然后用所有小问题的答案来解决整个大问题。尽管我们考虑的问题是归并两个大数组，实际上我们归并的数组大多数都非常小。实现归并排序的另一种方法是先归并那些微型数组，然后再成对归并得到的子数组，如此这般，直到我们将整个数组归并在一起。这种实现方法比标准递归方法所需要的代码量更少。首先我们进行的是两两归并（把每个元素想象成一个大小为 1 的数组），然后是四四归并（将两个大小为 2 的数组归并成一个有 4 个元素的数组），然后是八八的归并，一直下去。在每一轮归并中，最后一次归并的第二个子数组可能比第一个子数组要小（但这对 merge() 方法不是问题），如果不是的话所有的归并中两个数组大小都应该一样，而在下一轮中子数组的大小会翻倍。此过程的可视轨迹如图 2.2.5 所示。

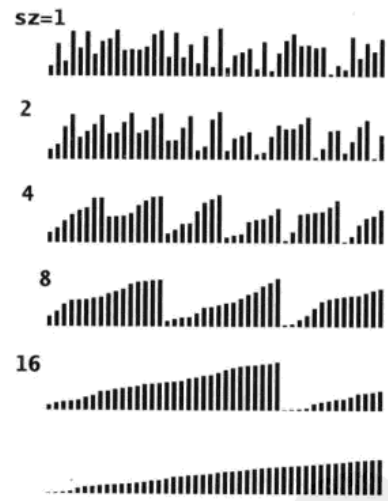


图 2.2.5 自底向上的归并排序的可视轨迹

277

自底向上的归并排序算法的实现如下。

自底向上的归并排序

```
public class MergeBU
{
   private static Comparable[] aux;      // 归并所需的辅助数组
```

```
   // merge()方法的代码请见"原地归并的抽象方法"
   public static void sort(Comparable[] a)
   {  //  进行lgN次两两归并
      int N = a.length;
      aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)        // sz子数组大小
         for (int lo = 0; lo < N-sz; lo += sz+sz) // lo:子数组索引
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

自底向上的归并排序会多次遍历整个数组，根据子数组大小进行两两归并。子数组的大小 sz 的初始值为1，每次加倍。最后一个子数组的大小只有在数组大小是 sz 的偶数倍的时候才会等于 sz（否则它会比 sz 小）。

	a[i]														
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
sz = 1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, **0**, 0, **1**)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, **2**, 2, **3**)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, **4**, 4, **5**)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, **6**, 6, **7**)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, **8**, 8, **9**)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, **10**, 10, **11**)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, **12**, 12, **13**)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, **14**, 14, **15**)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
merge(a, **0**, 1, **3**)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, **4**, 5, **7**)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, **8**, 9, **11**)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, **12**, 13, **15**)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
merge(a, **0**, 3, **7**)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, **8**, 11, **15**)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																
merge(a, **0**, 7, **15**)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

278

自底向上的归并排序的归并结果

命题 H。对于长度为 N 的任意数组，自底向上的归并排序需要 $1/2N\lg N$ 至 $N\lg N$ 次比较，最多访问数组 $6N\lg N$ 次。

证明。处理一个数组的遍数正好是 $\lfloor \lg N \rfloor$（即 $2^n \leq N < 2^{n+1}$ 中的 n）。每一遍会访问数组 $6N$ 次，比较次数在 $N/2$ 和 N 之间。

当数组长度为 2 的幂时，自顶向下和自底向上的归并排序所用的比较次数和数组访问次数正好相同，只是顺序不同。其他时候，两种方法的比较和数组访问的次序会有所不同（请见练习 2.2.5）。

自底向上的归并排序比较适合用链表组织的数据。想象一下将链表先按大小为 1 的子链表进行排序，然后是大小为 2 的子链表，然后是大小为 4 的子链表等。这种方法只需要重新组织链表链接就能将链表原地排序（不需要创建任何新的链表结点）。

用自顶向下或是自底向上的方式实现任何分治类的算法都很自然。归并排序告诉我们，当能够用其中一种方法解决一个问题时，你都应该试试另一种。你是希望像 Merge.sort() 中那样化整为

零（然后递归地解决它们）的方式解决问题，还是希望像 MergeBU.sort() 中那样循序渐进地解决问题呢？

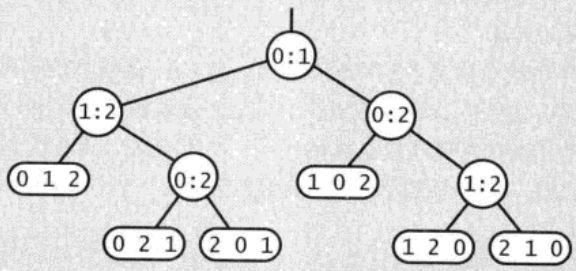
2.2.4 排序算法的复杂度

学习归并排序的一个重要原因是它是证明计算复杂性领域的一个重要结论的基础，而计算复杂性能够帮助我们理解排序自身固有的难易程度。计算复杂性在算法设计中扮演着非常重要的角色，而这个结论正是和排序算法的设计直接相关的，因此接下来我们就要详细地讨论它。

研究复杂度的第一步是建立一个计算模型。一般来说，研究者会尽量寻找一个和问题相关的最简单的模型。对排序来说，我们的研究对象是基于比较的算法，它们对数组元素的操作方式是由主键的比较决定的。一个基于比较的算法在两次比较之间可能会进行任意规模的计算，但它只能通过主键之间的比较得到关于某个主键的信息。因为我们局限于实现了 Comparable 接口的对象，本章中的所有算法都属于这一类（注意，我们忽略了访问数组的开销）。在第 5 章中，我们会讨论不局限于 Comparable 元素的算法。 279

命题 I。没有任何基于比较的算法能够保证使用少于 $\lg(N!) \sim N\lg N$ 次比较将长度为 N 的数组排序。

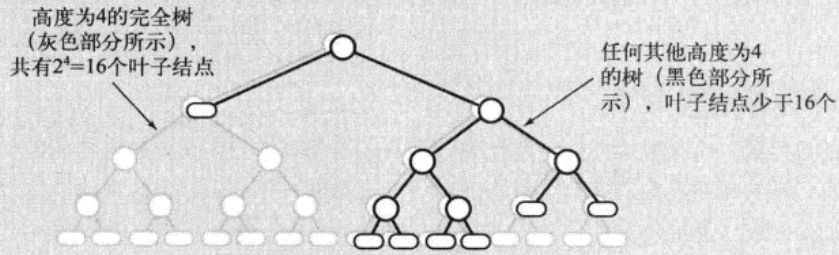
证明。首先，假设没有重复的主键，因为任何排序算法都必须能够处理这种情况。我们使用二叉树来表示所有的比较。树中的结点要么是一片叶子 $(i_0\ i_1\ i_2\ \ldots\ i_{N-1})$，表示排序完成且原输入的排列顺序是 $a[i_0], a[i_1], \ldots, a[i_{N-1}]$，要么是一个内部结点 $(i:j)$，表示 a[i] 和 a[j] 之间的一次比较操作，它的左子树表示 a[i] 小于 a[j] 时进行的其他比较，右子树表示 a[i] 大于 a[j] 时进行的其他比较。从根结点到叶子结点每一条路径都对应着算法在建立叶子结点所示的顺序时进行的所有比较。例如，这是一棵 N=3 时的比较树：



我们从来没有明确地构造这棵树——它只是用来描述算法中的比较的一个数学工具。

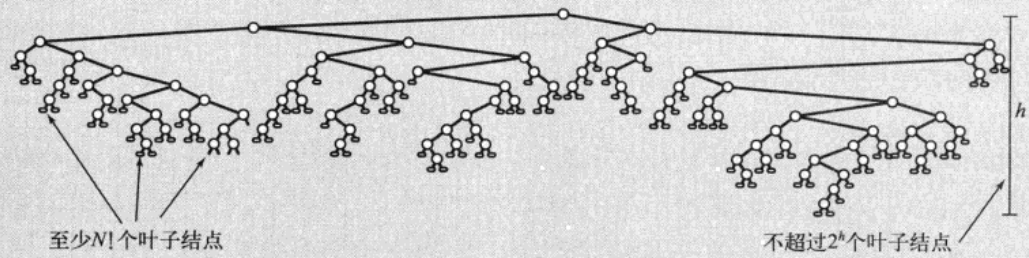
从比较树观察得到的第一个重要结论是这棵树应该至少有 $N!$ 个叶子结点，因为 N 个不同的主键会有 $N!$ 种不同的排列。如果叶子结点少于 $N!$，那肯定有一些排列顺序被遗漏了。算法对于那些被遗漏的输入肯定会失败。

从根结点到叶子结点的一条路径上的内部结点的数量即是某种输入下算法进行比较的次数。我们感兴趣的是这种路径能有多长（也就是树的高度），因为这也就是算法比较次数的最坏情况。二叉树的一个基本的组合学性质就是高度为 h 的树最多只可能有 2^h 个叶子结点，拥有 2^h 个结点的树是完美平衡的，或称为完全树。下图所示的就是一个 h=4 的例子。 280



结合前两段的分析可知，任意基于比较的排序算法都对应着一棵高 h 的比较树（如下图所示），其中：

$$N! \leqslant \text{叶子结点的数量} \leqslant 2^h$$



h 的值就是最坏情况下的比较次数，因此对不等式的两边取对数即可得到任意算法的比较次数至少是 $\lg N!$。根据斯特灵公式对阶乘函数的近似（见表 1.4.6）可得 $\lg N! \sim N \lg N$。

这个结论告诉了我们在设计排序算法的时候能够达到的最佳效果。例如，如果没有这个结论，我们可能会去尝试设计一个在最坏情况下比较次数只有归并排序的一半的基于比较的算法。命题 I 中的下限告诉我们这种努力是没有意义的——这样的算法不存在。这是一个重要结论，适用于任何我们能够想到的基于比较的算法。

命题 H 表明归并排序在最坏情况下的比较次数为 $\sim N \lg N$。这是其他排序算法复杂度的上限，也
281 就是说更好的算法需要保证使用的比较次数更少。命题 I 说明没有任何排序算法能够用少于 $\sim N \lg N$ 次比较将数组排序，这是其他排序算法复杂度的下限。也就是说，即使是最好的算法在最坏的情况下也至少需要这么多次比较。将两者结合起来也就意味着：

命题 J。归并排序是一种渐进最优的基于比较排序的算法。

证明。更准确地说，这句话的意思是，归并排序在最坏情况下的比较次数和任意基于比较的排序算法所需的最少比较次数都是 $\sim N \lg N$。命题 H 和命题 I 证明了这些结论。

需要强调的是，和计算模型一样，我们需要精确地定义最优算法。例如，我们可以严格地认为仅仅只需要 $\lg N!$ 次比较的算法才是最优的排序算法。我们不这么做的原因是，即使对于很大的 N，这种算法和（比如说）归并排序之间的差异也并不明显。或者我们也可以放宽最优的定义，使之包含任意在最坏情况下的比较次数都在 $N \lg N$ 的某个常数因子范围之内的排序算法。我们不这么做的原因是对于很大的 N，这种算法和归并排序之间的差距还是很明显的。

计算复杂度的概念可能会让人觉得很抽象，但解决可计算问题内在困难的基础性研究则不管怎

么说都是非常必要的。而且，在适用的情况下，关键在于计算复杂度会影响优秀软件的开发。首先，准确的上界为软件工程师保证性能提供了空间。很多例子表明，平方级别排序的性能低于线性排序。其次，准确的下界可以为我们节省很多时间，避免因不可能的性能改进而投入资源。

但归并排序的最优性并不是结束，也不代表在实际应用中我们不会考虑其他的方法了，因为本节中的理论还是有许多局限性的，例如：

- 归并排序的空间复杂度不是最优的；
- 在实践中不一定会遇到最坏情况；
- 除了比较，算法的其他操作（例如访问数组）也可能很重要；
- 不进行比较也能将某些数据排序。

因此在本书中我们还将继续学习其他一些排序算法。 282

答疑

问 归并排序比希尔排序快吗？

答 在实际应用中，它们的运行时间之间的差距在常数级别之内（希尔排序使用的是像算法 2.3 中那样的经过验证的递增序列），因此相对性能取决于具体的实现。

```
% java SortCompare Merge Shell 100000
For 100000 random Double values
    Merge is 1.2 times faster than Shell
```

理论上来说，还没有人能够证明希尔排序对于随机数据的运行时间是线性对数级别的，因此存在平均情况下希尔排序的性能的渐进增长率[①]更高的可能性。在最坏情况下，这种差距的存在已经被证实了，但这对实际应用没有影响。

问 为什么不把数组 `aux[]` 声明为 `merge()` 方法的局部变量？

答 这是为了避免每次归并时，即使是归并很小的数组，都创建一个新的数组。如果这么做，那么创建新数组将成为归并排序运行时间的主要部分（请见练习 2.2.26）。更好的解决方案是将 `aux[]` 变为 `sort()` 方法的局部变量，并将它作为参数传递给 `merge()` 方法（为了简化代码我们没有在例子中这么做，请见练习 2.2.9）。

问 当数组中存在重复的元素时归并排序的表现如何？

答 如果所有的元素都相同，那么归并排序的运行时间将是线性的（需要一个额外的测试来避免归并已经有序的数组）。但如果有多个不同的重复值，这样做的性能收益就不是很明显了。例如，假设输入数组的 N 个奇数位上的元素都是同一个值，另外 N 个偶数位上的元素都是另一个值，此时算法的运行时间是线性对数的（这样的数组和所有元素都不重复的数组满足了相同的循环条件），而非线性的。 283

练习

2.2.1 按照本节开头所示轨迹的格式给出原地归并的抽象 `merge()` 方法是如何将数组 `A E Q S U Y E I N O S T` 排序的。

① 即运行时间的近似函数。——译者注

2.2.2 按照算法 2.4 所示轨迹的格式给出自顶向下的归并排序是如何将数组 E A S Y Q U E S T I O N 排序的。

2.2.3 用自底向上的归并排序解答练习 2.2.2。

2.2.4 是否当且仅当两个输入的子数组都有序时原地归并的抽象方法才能得到正确的结果？证明你的结论，或者给出一个反例。

2.2.5 当输入数组的大小 N=39 时，给出自顶向下和自底向上的归并排序中各次归并子数组的大小及顺序。

2.2.6 编写一个程序来计算自顶向下和自底向上的归并排序访问数组的准确次数。使用这个程序将 N=1 至 512 的结果绘成曲线图，并将其和上限 $6N\lg N$ 比较。

2.2.7 证明归并排序的比较次数是单调递增的（即对于 $N>0$，$C(N+1)>C(N)$）。

2.2.8 假设将算法 2.4 修改为：只要 `a[mid] <= a[mid+1]` 就不调用 `merge()` 方法，请证明用归并排序处理一个已经有序的数组所需的比较次数是线性级别的。

2.2.9 在库函数中使用 `aux[]` 这样的静态数组是不妥当的，因为可能会有多个程序同时使用这个类。实现一个不用静态数组的 `Merge` 类，但也不要将 `aux[]` 变为 `merge()` 的局部变量（请见本节的答
284 疑部分）。提示：可以将辅助数组作为参数传递给递归的 `sort()` 方法。

提高题

2.2.10 *快速归并。*实现一个 `merge()` 方法，按降序将 `a[]` 的后半部分复制到 `aux[]`，然后将其归并回 `a[]` 中。这样就可以去掉内循环中检测某半边是否用尽的代码。*注意：*这样的排序产生的结果是不稳定的（请见 2.5.1.8 节）。

2.2.11 *改进。*实现 2.2.2 节所述的对归并排序的三项改进：加快小数组的排序速度，检测数组是否已经有序以及通过在递归中交换参数来避免数组复制。

2.2.12 *次线性的额外空间。*用大小 M 将数组分为 N/M 块（简单起见，设 M 是 N 的约数）。实现一个归并方法，使之所需的额外空间减少到 $\max(M, N/M)$：(i) 可以先将一个块看做一个元素，将块的第一个元素作为块的主键，用选择排序将块排序；(ii) 遍历数组，将第一块和第二块归并，完成后将第二块和第三块归并，等等。

2.2.13 *平均情况的下限。*请证明任意基于比较的排序算法的预期比较次数至少为 $\sim N\lg N$（假设输入元素的所有排列的出现概率是均等的）。*提示：*比较次数至少是比较树的外部路径的长度（根结点到所有叶子结点的路径长度之和），当树平衡时该值最小。

2.2.14 *归并有序的队列。*编写一个静态方法，将两个有序的队列作为参数，返回一个归并后的有序队列。

2.2.15 *自底向上的有序队列归并排序。*用下面的方法编写一个自底向上的归并排序：给定 N 个元素，创建 N 个队列，每个队列包含其中一个元素。创建一个由这 N 个队列组成的队列，然后不断用练习 2.2.14 中的方法将队列的头两个元素归并，并将结果重新加入到队列结尾，直到队列的队列只剩下一个元素为止。

2.2.16 *自然的归并排序。*编写一个自底向上的归并排序，当需要将两个子数组排序时能够利用数组中已经有序的部分。首先找到一个有序的子数组（移动指针直到当前元素比上一个元素小为止），
285 然后再找出另一个并将它们归并。根据数组大小和数组中递增子数组的最大长度分析算法的运行时间。

2.2.17 *链表排序。*实现对链表的自然排序（这是将链表排序的最佳方法，因为它不需要额外的空间，且运行时间是线性对数级别的）。

2.2.18 *打乱链表*。实现一个分治算法，使用线性对数级别的时间和对数级别的额外空间随机打乱一条链表。

2.2.19 *倒置*。编写一个线性对数级别的算法统计给定数组中的“倒置”数量（即插入排序所需的交换次数，请见 2.1 节）。这个数量和 *Kendall tau* 距离有关，请见 2.5 节。

2.2.20 *间接排序*。编写一个不改变数组的归并排序，它返回一个 `int[]` 数组 `perm`，其中 `perm[i]` 的值是原数组中第 *i* 小的元素的位置。

2.2.21 *一式三份*。给定三个列表，每个列表中包含 *N* 个名字，编写一个线性对数级别的算法来判定三份列表中是否含有公共的名字，如果有，返回第一个被找到的这种名字。

2.2.22 *三向归并排序*。假设每次我们是把数组分成三个部分而不是两个部分并将它们分别排序，然后进行三向归并。这种算法的运行时间的增长数量级是多少？ 286

实验题

2.2.23 *改进*。用实验评估正文中所提到的归并排序的三项改进（请见练习 2.2.11）的效果，并比较正文中实现的归并和练习 2.2.10 所实现的归并之间的性能。根据经验给出应该在何时为子数组切换到插入排序。

2.2.24 *改进的有序测试*。在实验中用大型随机数组评估练习 2.2.8 所做的修改的效果。根据经验用 *N*（被排序的原始数组的大小）的函数描述条件语句（`a[mid] <= a[mid+1]`）成立（无论数组是否有序）的平均次数。

2.2.25 *多向归并排序*。实现一个 *k* 向（相对双向而言）归并排序程序。分析你的算法，估计最佳的 *k* 值并通过实验验证猜想。

2.2.26 *创建数组*。使用 `SortCompare` 粗略比较在你的计算机上在 `merge()` 中和在 `sort()` 中创建 `aux[]` 的性能差异。

2.2.27 *子数组长度*。用归并将大型随机数组排序，根据经验用 *N*（某次归并时两个子数组的长度之和）的函数估计当一个子数组用尽时另一个子数组的平均长度。

2.2.28 *自顶向下与自底向上*。对于 $N=10^3$、10^4、10^5 和 10^6，使用 `SortCompare` 比较自顶向下和自底向上的归并排序的性能。

2.2.29 *自然的归并排序*。对于 $N=10^3$、10^6 和 10^9，类型为 `Long` 的随机主键数组，根据经验给出自然的归并排序（请见练习 2.2.16）所需要的遍数。*提示*：不需要实现这个排序（甚至不需要生成所有完整的 64 位主键）也能完成这道练习。 287

2.3 快速排序

本节的主题是快速排序，它可能是应用最广泛的排序算法了。快速排序流行的原因是它实现简单、适用于各种不同的输入数据且在一般应用中比其他排序算法都要快得多。快速排序引人注目的特点包括它是原地排序（只需要一个很小的辅助栈），且将长度为 N 的数组排序所需的时间和 $N\lg N$ 成正比。我们已经学习过的排序算法都无法将这两个优点结合起来。另外，快速排序的内循环比大多数排序算法都要短小，这意味着它无论是在理论上还是在实际中都要更快。它的主要缺点是非常脆弱，在实现时要非常小心才能避免低劣的性能。已经有无数例子显示许多种错误都能致使它在实际中的性能只有平方级别。幸好我们将会看到，由这些错误中学到的教训也大大改进了快速排序算法，使它的应用更加广泛。

2.3.1 基本算法

快速排序是一种分治的排序算法。它将一个数组分成两个子数组，将两部分独立地排序。快速排序和归并排序是互补的：归并排序将数组分成两个子数组分别排序，并将有序的子数组归并以将整个数组排序；而快速排序将数组排序的方式则是当两个子数组都有序时整个数组也就自然有序了。在第一种情况中，递归调用发生在处理整个数组之前；在第二种情况中，递归调用发生在处理整个数组之后。在归并排序中，一个数组被等分为两半；在快速排序中，切分（partition）的位置取决于数组的内容。快速排序的大致过程如图 2.3.1 所示。



288

图 2.3.1 快速排序示意图

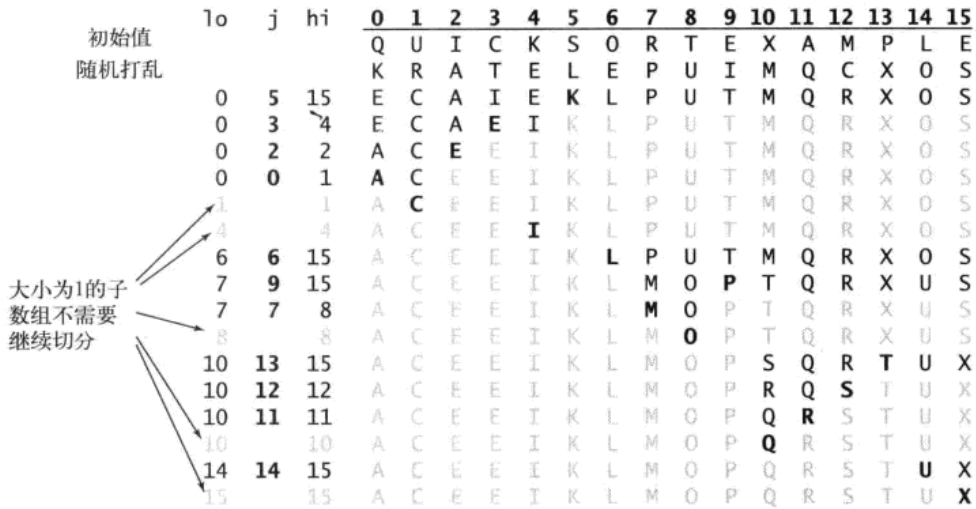
快速排序的实现过程如算法 2.5 所示。

算法 2.5 快速排序

```
public class Quick
{
   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);            // 消除对输入的依赖
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);  // 切分（请见“快速排序的切分”）
      sort(a, lo, j-1);              // 将左半部分a[lo .. j-1]排序
      sort(a, j+1, hi);              // 将右半部分a[j+1 .. hi]排序
   }
}
```

快速排序递归地将子数组 a[lo..hi] 排序，先用 partition() 方法将 a[j] 放到一个合适位置，然后再用递归调用将其他位置的元素排序。



289

该方法的关键在于切分，这个过程使得数组满足下面三个条件：

- 对于某个 j，a[j] 已经排定；
- a[lo] 到 a[j-1] 中的所有元素都不大于 a[j]；
- a[j+1] 到 a[hi] 中的所有元素都不小于 a[j]。

我们就是通过递归地调用切分来排序的。

因为切分过程总是能排定一个元素，用归纳法不难证明递归能够正确地将数组排序：如果左子数组和右子数组都是有序的，那么由左子数组（有序且没有任何元素大于切分元素）、切分元素和右子数组（有序且没有任何元素小于切分元素）组成的结果数组也一定是有序的。算法 2.5 就是实现了这个思路的一个递归程序。它是一个随机化的算法，因为它在将数组排序之前会将其随机打乱。我们这么做的原因是希望能够预测（并依赖）该算法的性能特性，之后我们会详细讨论。

要完成这个实现，需要实现切分方法。一般策略是先随意地取 a[lo] 作为*切分元素*，即那个将会被排定的元素，然后我们从数组的左端开始向右扫描直到找到一个大于等于它的元素，再从数组的右端开始向左扫描直到找到一个小于等于它的元素。这两个元素显然是没有排定的，因此我们交换它们的位置。如此继续，我们就可以保证左指针 i 的左侧元素都不大于切分元素，右指针 j 的右侧元素都不小于切分元素。当两个指针相遇时，我们只需要将切分元素 a[lo] 和左子数组最右侧的元素（a[j]）交换然后返回 j 即可。切分方法的大致过程如图 2.3.2 所示。

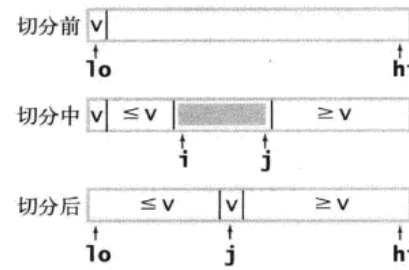


图 2.3.2 快速排序的切分示意图

这段快速排序的实现代码中还有几个细节问题值得一提，因为它们都可能导致实现错误或是影响性能，我们会在下面讨论。本节稍后我们会研究算法的三个高层次的改进。

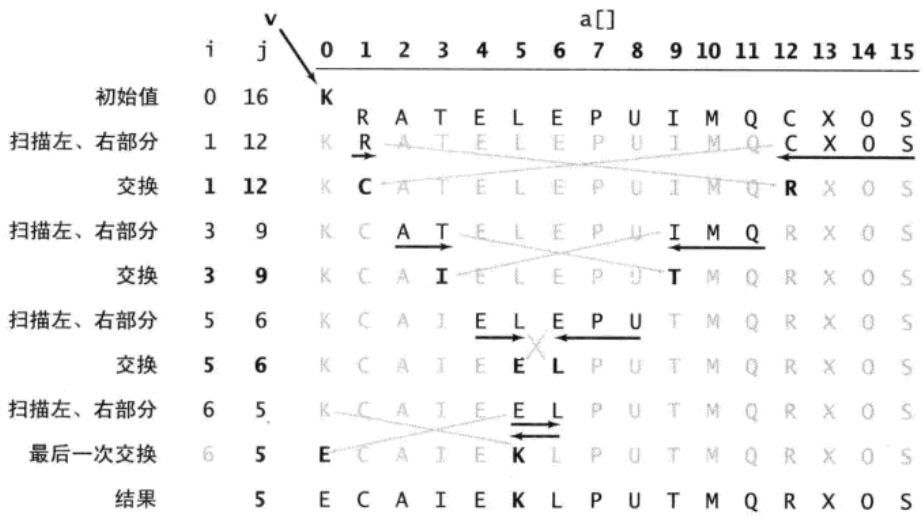
290

快速排序的切分的实现如下所示。

快速排序的切分

```
private static int partition(Comparable[] a, int lo, int hi)
{  // 将数组切分为a[lo..i-1], a[i], a[i+1..hi]
   int i = lo, j = hi+1;            // 左右扫描指针
   Comparable v = a[lo];            // 切分元素
   while (true)
   {  // 扫描左右，检查扫描是否结束并交换元素
      while (less(a[++i], v)) if (i == hi) break;
      while (less(v, a[--j])) if (j == lo) break;
      if (i >= j) break;
      exch(a, i, j);
   }
   exch(a, lo, j);       // 将v = a[j]放入正确的位置
   return j;             // a[lo..j-1] <= a[j] <= a[j+1..hi] 达成
}
```

这段代码按照 a[lo] 的值 v 进行切分。当指针 i 和 j 相遇时主循环退出。在循环中，a[i] 小于 v 时我们增大 i，a[j] 大于 v 时我们减小 j，然后交换 a[i] 和 a[j] 来保证 i 左侧的元素都不大于 v，j 右侧的元素都不小于 v。当指针相遇时交换 a[lo] 和 a[j]，切分结束（这样切分值就留在 a[j] 中了）。



切分轨迹（每次交换前后的数组内容）

291

2.3.1.1 原地切分

如果使用一个辅助数组，我们可以很容易实现切分，但将切分后的数组复制回去的开销也许会使我们得不偿失。一个初级 Java 程序员甚至可能会将空数组创建在递归的切分方法中，这会大大降低排序的速度。

2.3.1.2 别越界

如果切分元素是数组中最小或最大的那个元素，我们就要小心别让扫描指针跑出数组的边界。partition() 实现可进行明确的检测来预防这种情况。测试条件（j == lo）是冗余的，因为切分

元素就是 a[lo]，它不可能比自己小。数组右端也有相同的情况，它们都是可以去掉的（请见练习 2.3.17）。

2.3.1.3 保持随机性

数组元素的顺序是被打乱过的。因为算法 2.5 对所有的子数组都一视同仁，它的所有子数组也都是随机排序的。这对于预测算法的运行时间很重要。保持随机性的另一种方法是在 partition() 中随机选择一个切分元素。

2.3.1.4 终止循环

有经验的程序员都知道保证循环结束需要格外小心，快速排序的切分循环也不例外。正确地检测指针是否越界需要一点技巧，并不像看上去那么容易。一个最常见的错误是没有考虑到数组中可能包含和切分元素的值相同的其他元素。

2.3.1.5 处理切分元素值有重复的情况

如算法 2.5 所示，左侧扫描最好是在遇到大于等于切分元素值的元素时停下，右侧扫描则是遇到小于等于切分元素值的元素时停下。尽管这样可能会不必要地将一些等值的元素交换，但在某些典型应用中，它能够避免算法的运行时间变为平方级别（请见练习 2.3.11）。稍后我们会讨论另一种可以更好地处理含有大量重复值的数组的方法。

2.3.1.6 终止递归

有经验的程序员还知道保证递归总是能够结束也是需要小心的，快速排序也不例外。例如，实现快速排序时一个常见的错误就是不能保证将切分元素放入正确的位置，从而导致程序在切分元素正好是子数组的最大或是最小元素时陷入了无限的递归循环之中。 292

2.3.2 性能特点

数学上已经对快速排序进行了详尽的分析，因此我们能够精确地说明它的性能。大量经验也证明了这些分析，它们是算法调优时的重要工具。

快速排序切分方法的内循环会用一个递增的索引将数组元素和一个定值比较。这种简洁性也是快速排序的一个优点，很难想象排序算法中还能有比这更短小的内循环了。例如，归并排序和希尔排序一般都比快速排序慢，其原因就是它们还在内循环中移动数据。

快速排序另一个速度优势在于它的比较次数很少。排序效率最终还是依赖切分数组的效果，而这依赖于切分元素的值。切分将一个较大的随机数组分成两个随机子数组，而实际上这种分割可能发生在数组的任意位置（对于元素不重复的数组而言）。下面我们来分析这个算法，看看这种方法和理想方法之间的差距。

快速排序的最好情况是每次都正好能将数组对半分。在这种情况下快速排序所用的比较次数正好满足分治递归的 $C_N=2C_{N/2}+N$ 公式。$2C_{N/2}$ 表示将两个子数组排序的成本，N 表示用切分元素和所有数组元素进行比较的成本。由归并排序的命题 F 的证明可知，这个递归公式的解 $C_N \sim N\lg_N$。尽管事情并不总会这么顺利，但平均而言切分元素都能落在数组的中间。将每个切分位置的概率都考虑进去只会使递归更加复杂、更难解决，但最终结果还是类似的。我们对快速排序的信心来自于这个结论的证明。如果你不喜欢数学公式，可以跳过这个证明，相信它即可；如果你喜欢，你会发现它很有趣。

命题K。将长度为N的无重复数组排序，快速排序平均需要~$2N\ln N$次比较（以及1/6的交换）。

证明。令C_N为将N个不同元素排序平均所需的比较次数。显然$C_0=C_1=0$，对于$N>1$，由递归
293 程序可以得到以下归纳关系：

$$C_N=N+1+(C_0+C_1+\cdots+C_{N-2}+C_{N-1})/N+(C_{N-1}+C_{N-2}+\cdots+C_0)/N$$

第一项是切分的成本（总是$N+1$），第二项是将左子数组（长度可能是0到$N-1$）排序的平均成本，第三项是将右子数组(长度和左子数组相同)排序的平均成本。将等式左右两边乘以N并整理各项得到：

$$NC_N=N(N+1)+2(C_0+C_1+\cdots+C_{N-2}+C_{N-1})$$

将该等式减去$N-1$时的相同等式可得：

$$NC_N-(N-1)C_{N-1}=2N+2C_{N-1}$$

整理等式并将两边除以$N(N+1)$可得：

$$C_N/(N+1)=C_{N-1}/N+2/(N+1)$$

归纳法推导可得：

$$C_N\sim 2(N+1)(1/3+1/4+\cdots+1/(N+1))$$

括号内的量是曲线$2/x$下从3到N的离散近似面积加一，积分得到$C_N\sim 2N\ln N$。注意到$2N\ln N\approx 1.39N\lg N$，也就是说平均比较次数只比最好情况多39%。
要得到命题中的交换次数需要一个类似（但更加复杂的）分析。

在实际应用中，当数组元素可能重复时，精确的分析会相当复杂，但不难证明即使存在重复的元素，平均比较次数也不会大于C_N（在2.3.3.3节中我们会改进快速排序在这种情况下的性能）。

尽管快速排序有很多优点，它的基本实现仍有一个潜在的缺点：在切分不平衡时这个程序可能会极为低效。例如，如果第一次从最小的元素切分，第二次从第二小的元素切分，如此这般，每次调用只会移除一个元素。这会导致一个大子数组需要切分很多次。我们要在快速排序前将数组随机排序的
294 主要原因就是要避免这种情况。它能够使产生糟糕的切分的可能性降到极低，我们就无需为此担心了。

命题L。快速排序最多需要约$N^2/2$次比较，但随机打乱数组能够预防这种情况。

证明。根据刚才的证明，在每次切分后两个子数组之一总是空的情况下，比较次数为：

$$N+(N-1)+(N-2)+\cdots+2+1=(N+1)N/2$$

这不仅说明算法所需的时间是平方级别的，也显示了算法所需的空间是线性的，而这对于大数组来说是不可接受的。但是（经过一些复杂的工作）通过扩展对一般情况的分析我们可以得到比较次数的标准差约为$0.65N$。因此，随着N的增大，运行时间会趋近于平均数，且不可能与平均数偏差太大。例如，对于一个有100万个元素的数组，由Chebyshev不等式可以粗略地估计出运行时间是平均所需时间的10倍的概率小于0.000 01（且真实的概率还要小得多）。对于大数组，运行时间是平方级别的概率小到可以忽略不计（请见练习2.3.10）。例如，快速排序所用的比较次数和插入排序或者选择排序一样多的概率比你的电脑在排序时被闪电击中的概率都要小得多！



总的来说，可以肯定的是对于大小为 N 的数组，算法 2.5 的运行时间在 $1.39N\lg N$ 的某个常数因子的范围之内。归并排序也能做到这一点，但是快速排序一般会更快（尽管它的比较次数多 39%），因为它移动数据的次数更少。这些保证都来自于数学概率，你完全可以相信它。

2.3.3 算法改进

快速排序是由 C.A.R Hoare 在 1960 年发明的，从那时起就有很多人在研究并改进它。改进快速排序总是那么吸引人，发明更快的排序算法就好像是计算机科学届的“老鼠夹子”，而快速排序就是夹子里的那块奶酪。几乎从 Hoare 第一次发表这个算法开始，人们就不断地提出各种改进方法。并不是所有的想法都可行，因为快速排序的平衡性已经非常好，改进所带来的提高可能会被意外的副作用所抵消。但其中一些，也是我们现在要介绍的，非常有效。 295

如果你的排序代码会被执行很多次或者会被用在大型数组上（特别是如果它会被发布成一个库函数，排序的对象数组的特性是未知的），那么下面所讨论的这些改进意见值得你参考。需要注意的是，你需要通过实验来确定改进的效果并为实现选择最佳的参数。一般来说它们能将性能提升 20% ~ 30%。

2.3.3.1 切换到插入排序

和大多数递归排序算法一样，改进快速排序性能的一个简单办法基于以下两点：

- 对于小数组，快速排序比插入排序慢；
- 因为递归，快速排序的 `sort()` 方法在小数组中也会调用自己。

因此，在排序小数组时应该切换到插入排序。简单地改动算法 2.5 就可以做到这一点：将 `sort()` 中的语句

```
if (hi <= lo) return;
```

替换成下面这条语句来对小数组使用插入排序：

```
if (hi <= lo + M) { Insertion.sort(a, lo, hi); return; }
```

转换参数 `M` 的最佳值是和系统相关的，但是 5 ~ 15 之间的任意值在大多数情况下都能令人满意（请见练习 2.3.25）。

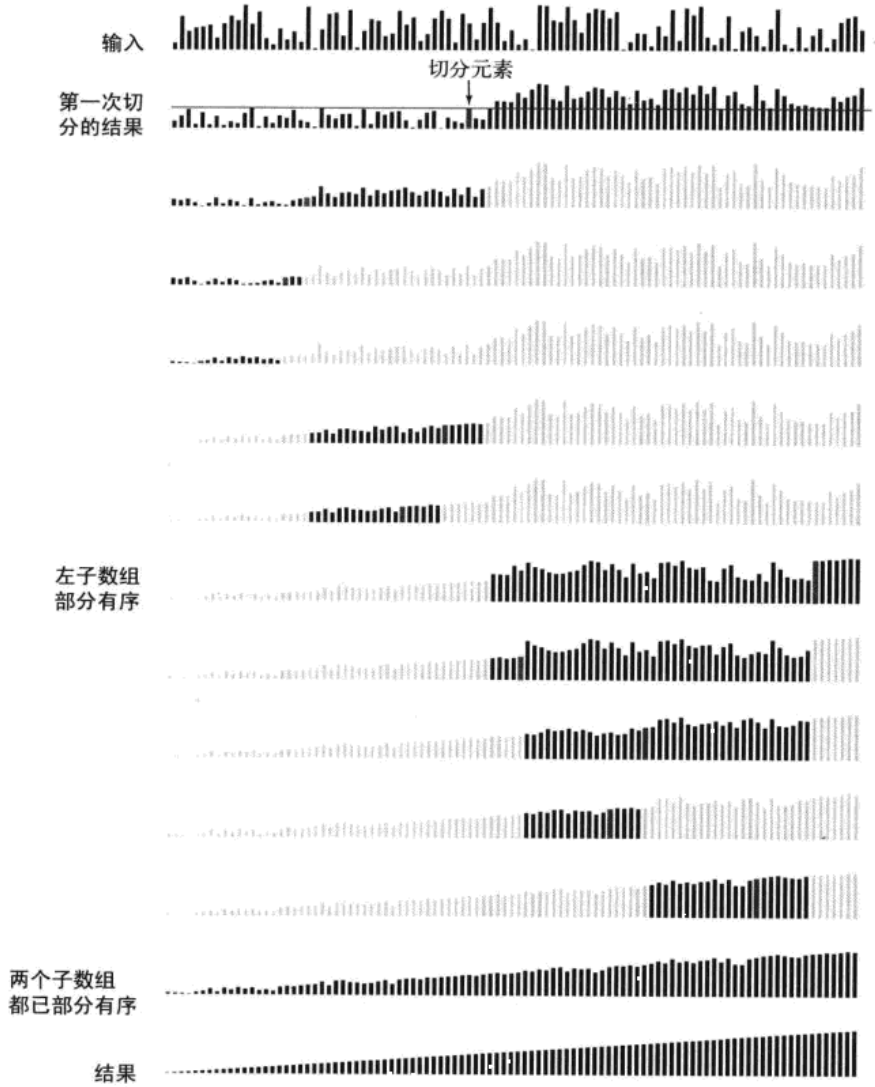
2.3.3.2 三取样切分

改进快速排序性能的第二个办法是使用子数组的一小部分元素的中位数来切分数组。这样做得到的切分更好，但代价是需要计算中位数。人们发现将取样大小设为 3 并用大小居中的元素切分的效果最好（请见练习 2.3.18 和练习 2.3.19）。我们还可以将取样元素放在数组末尾作为“哨兵”来去掉 `partition()` 中的数组边界测试。使用三取样切分的快速排序轨迹如图 2.3.3 所示。

2.3.3.3 熵最优的排序

实际应用中经常会出现含有大量重复元素的数组，例如我们可能需要将大量人员资料按照生日排序，或是按照性别区分开来。在这些情况下，我们实现的快速排序的性能尚可，但还有巨大的改进空间。例如，一个元素全部重复的子数组就不需要继续排序了，但我们的算法还会继续将它切分为更小的数组。在有大量重复元素的情况下，快速排序的递归性会使元素全部重复的子数组经常出现，这就有很大的改进潜力，将当前实现的线性对数级的性能提高到线性级别。 296

一个简单的想法是将数组切分为三部分，分别对应小于、等于和大于切分元素的数组元素。这种切分实现起来比我们目前使用的二分法更复杂，人们为解决它想出了许多不同的办法。这也是 E. W. Dijkstra 的荷兰国旗问题引发的一道经典的编程练习，因为这就好像用三种可能的主键值将数组排序一样，这三种主键值对应着荷兰国旗上的三种颜色。



297

图 2.3.3 使用了三取样切分和插入排序转换的快速排序（另见彩插）

Dijkstra 的解法如“三向切分的快速排序”中极为简洁的切分代码所示。它从左到右遍历数组一次，维护一个指针 lt 使得 a[lo..lt-1] 中的元素都小于 v，一个指针 gt 使得 a[gt+1..hi] 中的元素都大于 v，一个指针 i 使得 a[lt..i-1] 中的元素都等于 v，a[i..gt] 中的元素都还未确定，如图 2.3.4 所示。一开始 i 和 lo 相等，我们使用 Comparable 接口（而非 less()）对 a[i] 进行三向比较来直接处理以下情况：

- a[i] 小于 v，将 a[lt] 和 a[i] 交换，将 lt 和 i 加一；
- a[i] 大于 v，将 a[gt] 和 a[i] 交换，将 gt 减一；
- a[i] 等于 v，将 i 加一。

这些操作都会保证数组元素不变且缩小 gt-i 的值（这样循环才会结束）。另外，除非和切分元素相等，其他元素都会被交换。

20 世纪 70 年代，快速排序发布不久后这段代码就出现了，但它并没有流行开来，因为在数组中重复元素不多的普通情况下它比标准的二分法多使用了很多次交换。90 年代，J. Bently 和 D. McIlroy 找到一个聪明的方法解决了这个问题（请见练习 2.3.22），使得三向切分的快速排序比归并排序和其他排序方法在包括重复元素很多的实际应用中更快。之后，J. Bently 和 R. Sedgewick 证明了这一点，我们会在下面讨论。

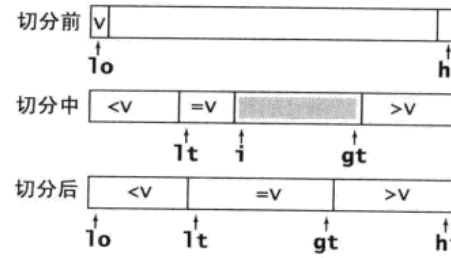


图 2.3.4 三向切分的示意图

但我们已经证明过归并排序是最优的。如何才能突破它的下界？这个问题的答案在于 2.2 节的命题 I 讨论的是对任意输入的最差性能，而我们目前在讨论时已经知道输入数组的一些信息了。对于含有以任意概率分布的重复元素的输入，归并排序无法保证最佳性能。 298

三向切分的快速排序的实现如下所示。

三向切分的快速排序

```
public class Quick3way
{

   private static void sort(Comparable[] a, int lo, int hi)
   {  // 调用此方法的公有方法sort()请见算法2.5
      if (hi <= lo) return;
      int lt = lo, i = lo+1, gt = hi;
      Comparable v = a[lo];
      while (i <= gt)
      {
         int cmp = a[i].compareTo(v);
         if      (cmp < 0) exch(a, lt++, i++);
         else if (cmp > 0) exch(a, i, gt--);
         else              i++;
      }  // 现在 a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi]成立
      sort(a, lo, lt - 1);
      sort(a, gt + 1, hi);
   }
}
```

这段排序代码的切分能够将和切分元素相等的元素归位，这样它们就不会被包含在递归调用处理的子数组之中了。对于存在大量重复元素的数组，这种方法比标准的快速排序的效率高得多（请见正文）。

三向分切的快速排序的可视轨迹如图 2.3.5 所示。

			v	a[]									
lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	8	B	B	R	R	R	W	B	R	R	W	W	W
2	5	7	B	B	R	R	R	R	B	R	W	W	W	W
2	6	7	B	B	R	R	R	R	B	R	W	W	W	W
3	7	7	B	B	B	R	R	R	R	R	W	W	W	W
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W

三向切分的轨迹（每次迭代循环之后的数组内容） 299

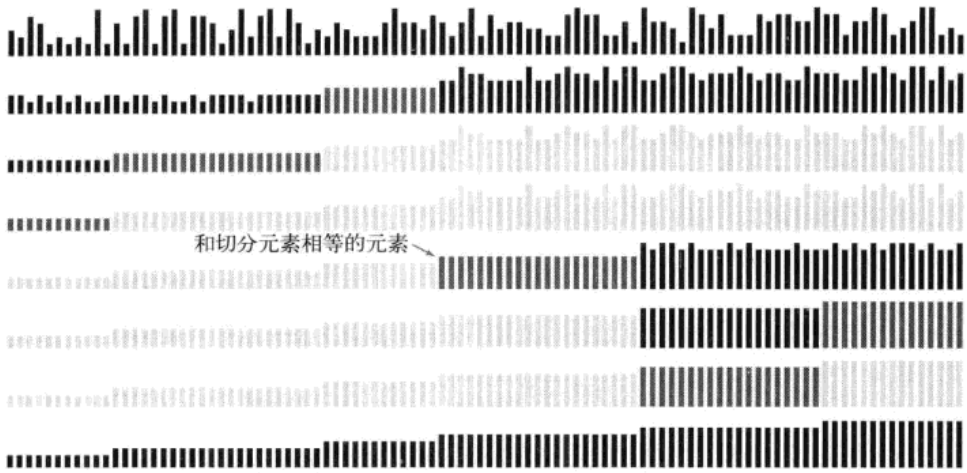


图 2.3.5 三向切分的快速排序的可视轨迹（另见彩插）

例如，对于只有若干不同主键的随机数组，归并排序的时间复杂度是线性对数的，而三向切分快速排序则是线性的。从上面的可视轨迹就可以看出，主键值数量的 N 倍是运行时间的一个保守的上界。

这些准确的结论来自于对主键概率分布的分析。给定包含 k 个不同值的 N 个主键，对于从 1 到 k 的每个 i，定义 f_i 为第 i 个主键值出现的次数，p_i 为 f_i/N，即为随机抽取一个数组元素时第 i 个主键值出现的概率。那么所有主键的香农信息量（对信息含量的一种标准的度量方法）可以定义为：

$$H=-(p_1\lg p_1+p_2\lg p_2+\cdots+p_k\lg p_k)$$

给定任意一个待排序的数组，通过统计每个主键值出现的频率就可以计算出它包含的信息量。值得一提的是，可以通过这个信息量得出三向切分的快速排序所需要的比较次数的上下界。

命题 M。不存在任何基于比较的排序算法能够保证在 $NH-N$ 次比较之内将 N 个元素排序，其中 H 为由主键值出现频率定义的香农信息量。

300

略证。将 2.2 节的命题 I 中下界的证明（相对简单地）一般化即可证明该结论。

命题 N。对于大小为 N 的数组，三向切分的快速排序需要 $\sim(2\ln 2)NH$ 次比较。其中 H 为由主键值出现频率定义的香农信息量。

略证。将命题 K 中快速排序的普通情况的分析（相对困难地）通用化即可证明该结论。在所有主键都不重复的情况下，它比最优解所需比较多 39%（但仍在常数因子的范围之内）。

请注意，当所有的主键值均不重复时有 $H=\lg N$（所有主键的概率均为 $1/N$），这和 2.2 节的命题 I 以及命题 K 是一致的。三向切分的最坏情况正是所有主键均不相同。当存在重复主键时，它的性能就会比归并排序好得多。更重要的是，这两个性质一起说明了三向切分是信息量最优的，即对于任意分布的输入，最优的基于比较的算法平均所需的比较次数和三向切分的快速排序平均所需的

比较次数相互处于常数因子范围之内。

对于标准的快速排序，随着数组规模的增大其运行时间会趋于平均运行时间，大幅偏离的情况非常罕见，因此可以肯定三向切分的快速排序的运行时间和输入的信息量的 N 倍是成正比的。在实际应用中这个性质很重要，因为对于包含大量重复元素的数组，它将排序时间从线性对数级降低到了线性级别。这和元素的排列顺序没有关系，因为算法会在排序之前将其打乱以避免最坏情况。元素的概率分布决定了信息量的大小，没有基于比较的排序算法能够用少于信息量决定的比较次数完成排序。这种对重复元素的适应性使得三向切分的快速排序成为排序库函数的最佳算法选择——需要将包含大量重复元素的数组排序的用例很常见。

经过精心调优的快速排序在绝大多数计算机上的绝大多数应用中都会比其他基于比较的排序算法更快。快速排序在今天的计算机业界中的广泛应用正是因为我们讨论过的数学模型说明了它在实际应用中比其他方法的性能更好，而近几十年的大量实验和经验也证明了这个结论。

在第 5 章中我们会发现，这些并不是快速排序发展的终点，因为有人研究出了完全不需要比较的排序算法！但快速排序的另一个版本在那个环境下仍然是最棒的，和这里一样。 301

答疑

问 有没有将数组平分的办法，而不是根据切分元素的最后位置来切分数组？

答 这个问题困扰了专家们十多年。这和用数组的中位数切分的想法类似。我们在 2.5.3.4 节中讨论了寻找中位数的问题。在线性时间内找到是可能的，但用现有的算法（基于快速排序的切分），这么做的代价远远超过将数组平分而节省的 39%。

问 随机地将数组打乱似乎占了排序用时的一大部分，这么做值得吗？

答 值得。这能够防止出现最坏情况并使运行时间可以预计。Hoare 在 1960 年提出这个算法的时候就推荐了这种方法——它是一种（也是第一批）偏爱随机性的算法。

问 为什么都将注意力放在重复元素上？

答 这个问题直接影响到实际应用中的性能。它曾被忽略了数十年，结果是一些老的实现对含有大量重复元素的数组排序时用时超过平方级别，这在实际应用中肯定出现过。像算法 2.5 等较好的实现对于这种数组的复杂度是线性对数级别的，但在很多情况下，如本节最后将其改进为信息量最佳的线性级别是很值得的。 302

练习

2.3.1 按照 `partition()` 方法的轨迹的格式给出该方法是如何切分数组 `E A S Y Q U E S T I O N` 的。

2.3.2 按照本节中快速排序所示轨迹的格式给出快速排序是如何将数组 `E A S Y Q U E S T I O N` 排序的（出于练习的目的，可以忽略开头打乱数组的部分）。

2.3.3 对于长度为 N 的数组，在 `Quick.sort()` 执行时，其最大的元素最多会被交换多少次？

2.3.4 假如跳过开头打乱数组的操作，给出六个含有 10 个元素的数组，使得 `Quick.sort()` 所需的比较次数达到最坏情况。

2.3.5 给出一段代码将已知只有两种主键值的数组排序。

2.3.6 编写一段代码来计算 C_N 的准确值，在 N=100、1000 和 10 000 的情况下比较准确值和估计值 $2N\ln N$ 的差距。

2.3.7 在使用快速排序将 N 个不重复的元素排序时，计算大小为 0、1 和 2 的子数组的数量。如果你喜欢数学，请推导；如果你不喜欢，请做一些实验并提出猜想。

2.3.8 `Quick.sort()` 在处理 N 个全部重复的元素时大约需要多少次比较？

2.3.9 请说明 `Quick.sort()` 在处理只有两种主键值的数组时的行为，以及在处理只有三种主键值的数组时的行为。

2.3.10 Chebyshev 不等式表明，一个随机变量的标准差距离均值大于 k 的概率小于 $1/k^2$。对于 N=100 万，用 Chebyshev 不等式计算快速排序所使用的比较次数大于 1000 亿次的概率（$0.1N^2$）。

2.3.11 假如在遇到和切分元素重复的元素时我们继续扫描数组而不是停下来，证明使用这种方法的快速排序在处理只有若干种元素值的数组时的运行时间是平方级别的。

303

2.3.12 按照代码所示轨迹的格式给出信息量最佳的快速排序第一次是如何切分数组 `B A B A B A B A C A D A B R A` 的。

2.3.13 在最佳、平均和最坏情况下，快速排序的*递归深度*分别是多少？这决定了系统为了追踪递归调用所需的栈的大小。在最坏情况下保证递归深度为数组大小的对数级的方法请见练习 2.3.20。

2.3.14 证明在用快速排序处理大小为 N 的不重复数组时，比较第 i 大和第 j 大元素的概率为 $2/(j-i)$，并用该结论证明命题 K。

304

提高题

2.3.15 *螺丝和螺帽。*(G. J. E. Rawlins) 假设有 N 个螺丝和 N 个螺帽混在一堆，你需要快速将它们配对。一个螺丝只会匹配一个螺帽，一个螺帽也只会匹配一个螺丝。你可以试着把一个螺丝和一个螺帽拧在一起看看谁大了，但不能直接比较两个螺丝或者两个螺帽。给出一个解决这个问题的有效方法。

2.3.16 *最佳情况* 编写一段程序来生成使算法 2.5 中的 `sort()` 方法表现最佳的数组（无重复元素）：数组大小为 N 且不包含重复元素，每次切分后两个子数组的大小最多差 1（子数组的大小与含有 N 个相同元素的数组的切分情况相同）。（对于这道练习，我们不需要在排序开始时打乱数组。）*以下练习描述了快速排序的几个变体。它们每个都需要分别实现，但你也很自然地希望使用* `SortCompare` *进行实验来评估每种改动的效果。*

2.3.17 *哨兵。*修改算法 2.5，去掉内循环 `while` 中的边界检查。由于切分元素本身就是一个哨兵（`v` 不可能小于 `a[lo]`），左侧边界的检查是多余的。要去掉另一个检查，可以在打乱数组后将数组的最大元素放在 `a[length-1]` 中。该元素永远不会移动（除非和相等的元素交换），可以在所有包含它的子数组中成为哨兵。*注意：*在处理内部子数组时，右子数组中最左侧的元素可以作为左子数组右边界的哨兵。

2.3.18 *三取样切分。*为快速排序实现正文所述的三取样切分（参见 2.3.3.2 节）。运行双倍测试来确认这项改动的效果。

2.3.19 *五取样切分。*实现一种基于随机抽取子数组中 5 个元素并取中位数进行切分的快速排序。将取样元素放在数组的一侧以保证只有中位数元素参与了切分。运行双倍测试来确定这项改动的效果，并和标准的快速排序以及三取样切分的快速排序（请见上一道练习）进行比较。*附加题：*找到一种对于任意输入都只需要少于 7 次比较的五取样算法。

305

2.3.20 *非递归的快速排序。*实现一个非递归的快速排序，使用一个循环来将弹出栈的子数组切分并将结果子数组重新压入栈。*注意：*先将较大的子数组压入栈，这样就可以保证栈最多只会有 $\lg N$ 个元素。

2.3.21 *将重复元素排序的比较次数的下界。*完成命题 M 的证明的第一部分。参考命题 I 的证明并注意当有 k 个主键值时所有元素存在 $N!/f_1!f_2!\cdots f_k!$ 种不同的排列，其中第 i 个主键值出现的频率为 f_i（即 Np_i，按照命题 M 的记法），且 $f_1+\cdots+f_k=N$。

2.3.22 *快速三向切分。*（J. Bently，D. McIlroy）用将重复元素放置于子数组两端的方式实现一个信息量最优的排序算法。使用两个索引 p 和 q，使得 a[lo..p-1] 和 a[q+1..hi] 的元素都和 a[lo] 相等。使用另外两个索引 i 和 j，使得 a[p..i-1] 小于 a[lo]，a[j+i..q] 大于 a[lo]。在内循环中加入代码，在 a[i] 和 v 相当时将其与 a[p] 交换（并将 p 加 1），在 a[j] 和 v 相等且 a[i] 和 a[j] 尚未和 v 进行比较之前将其与 a[q] 交换。

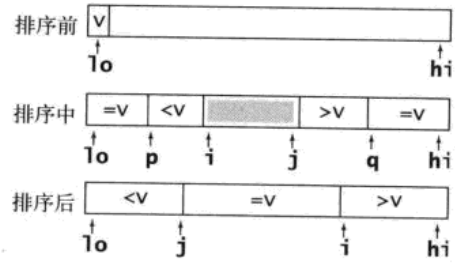


图 2.3.6 Bently-McIlroy 三向切分

添加在切分循环结束后将和 v 相等的元素交换到正确位置的代码，如图 2.3.6 所示。*请注意：*这里实现的代码和正文中给出的代码是等价的，因为这里额外的交换用于和切分元素相等的元素，而正文中的代码将额外的交换用于和切分元素不等的元素。

2.3.23 *Java 的排序库函数。*在练习 2.3.22 的代码中使用 Tukey's ninther 方法来找出切分元素——选择三组，每组三个元素，分别取三组元素的中位数，然后取三个中位数的中位数作为切分元素，且在排序小数组时切换到插入排序。

2.3.24 *取样排序。*（W. Frazer，A. McKellar）实现一个快速排序，取样大小为 2^k-1。首先将取样得到的元素排序，然后在递归函数中使用样品的中位数切分。分为两部分的其余样品元素无需再次排序并可以分别应用于原数组的两个子数组。这种算法被称为*取样排序*。 306

实验题

2.3.25 *切换到插入排序。*实现一个快速排序，在子数组元素少于 M 时切换到插入排序。用快速排序处理大小 N 分别为 10^3、10^4、10^5 和 10^6 的随机数组，根据经验给出使其在你的计算环境中运行速度最快的 M 值。将 M 从 0 变化到 30 的每个值所得到的平均运行时间绘成曲线。*注意：*你需要为算法 2.2 添加一个需要三个参数的 sort() 方法以使 Insertion.sort(a, lo, hi) 将子数组 a[lo..hi] 排序。

2.3.26 *子数组的大小。*编写一个程序，在快速排序处理大小为 N 的数组的过程中，当子数组的大小小于 M 时，排序方法需要切换为插入排序。将子数组的大小绘制成直方图。用 $N=10^5$，$M=10$、20 和 50 测试你的程序。

2.3.27 *忽略小数组。*用实验对比以下处理小数组的方法和练习 2.3.25 的处理方法的效果：在快速排序中直接忽略小数组，仅在快速排序结束后运行一次插入排序。*注意：*可以通过这些实验估计出电脑的缓存大小，因为当数组大小超出缓存时这种方法的性能可能会下降。

2.3.28 *递归深度。*用经验性的研究估计切换阈值为 M 的快速排序在将大小为 N 的不重复数组排序时的平均递归深度，其中 $M=10$、20 和 50，$N=10^3$、10^4、10^5 和 10^6。

2.3.29 *随机化。*用经验性的研究对比随机选择切分元素和正文所述的一开始就将数组随机化这两种策略的效果。在子数组大小为 M 时进行切换，将大小为 N 的不重复数组排序，其中 $M=10$、20 和 50，$N=10^3$、10^4、10^5 和 10^6。

2.3.30 *极端情况*。用初始随机化和非初始随机化的快速排序测试练习 2.1.35 和练习 2.1.36 中描述的大型非随机数组。在将这些大数组排序时，乱序对快速排序的性能有何影响？

2.3.31 *运行时间直方图*。编写一个程序，接受命令行参数 N 和 T，用快速排序对大小为 N 的随机浮点数数组进行 T 次排序，并将所有运行时间绘制成直方图。令 $N=10^3$、10^4、10^5 和 106，为了使曲线更平滑，T 值越大越好。这个练习最关键的地方在于找到适当的比例绘制出实验结果。

307

2.4 优先队列

许多应用程序都需要处理有序的元素，但不一定要求它们全部有序，或是不一定要一次就将它们排序。很多情况下我们会收集一些元素，处理当前键值最大的元素，然后再收集更多的元素，再处理当前键值最大的元素，如此这般。例如，你可能有一台能够同时运行多个应用程序的电脑（或者手机）。这是通过为每个应用程序的事件分配一个优先级，并总是处理下一个优先级最高的事件来实现的。例如，绝大多数手机分配给来电的优先级都会比游戏程序的高。

在这种情况下，一个合适的数据结构应该支持两种操作：*删除最大元素*和*插入元素*。这种数据类型叫做*优先队列*。优先队列的使用和队列（删除最老的元素）以及栈（删除最新的元素）类似，但高效地实现它则更有挑战性。

在本节中，简单地讨论优先队列的基本表现形式（其一或者两种操作都能在线性时间内完成）之后，我们会学习基于*二叉堆*数据结构的一种优先队列的经典实现方法，用数组保存元素并按照一定条件排序，以实现高效地（对数级别的）*删除最大元素*和*插入元素*操作。

优先队列的一些重要的应用场景包括模拟系统，其中事件的键即为发生的时间，而系统需要按照时间顺序处理所有事件；任务调度，其中键值对应的优先级决定了应该首先执行哪些任务；数值计算，键值代表计算错误，而我们需要按照键值指定的顺序来修正它们。在第 6 章中我们会学习一个具体的例子，展示优先队列在粒子碰撞模拟中的应用。

通过插入一列元素然后一个个地删掉其中最小的元素，我们可以用优先队列实现排序算法。一种名为*堆排序*的重要排序算法也来自于基于堆的优先队列的实现。稍后在本书中我们会学习如何用优先队列构造其他算法。在第 4 章中我们会看到优先队列如何恰到好处地抽象若干重要的图搜索算法；在第 5 章中，我们将使用本节所示的方法开发出一种数据压缩算法。这些只是优先队列作为算法设计工具所起到的举足轻重的作用的一部分例子。 308

2.4.1 API

优先队列是一种*抽象数据类型*（请见 1.2 节），它表示了一组值和对这些值的操作，它的抽象层使我们能够方便地将应用程序（用例）和我们将在本节中学习的各种具体实现隔离开来。和 1.2 节一样，我们会详细定义一组应用程序编程接口（API）来为数据结构的用例提供足够的信息（参见表 2.4.1）。优先队列最重要的操作就是*删除最大元素*和*插入元素*，所以我们会把精力集中在它们身上。删除最大元素的方法名为 `delMax()`，插入元素的方法名为 `insert()`。按照惯例，我们只会通过辅助函数 `less()` 来比较两个元素，和排序算法一样。如果允许重复元素，最大表示的是所有最大元素之一。为了将 API 定义完整，我们还需要加入构造函数（和我们在栈以及队列中使用的类似）和一个*空队列测试方法*。为了保证灵活性，我们在实现中使用了泛型，将实现了 `Comparable` 接口的数据的类型作为参数 `Key`。这使得我们可以不必再区别元素和元素的键，对数据类型和算法的描述也将更加清晰和简洁。例如，我们将用“最大元素”代替“最大键值”或是“键值最大的元素”。

表 2.4.1 泛型优先队列的 API

public class	**MaxPQ**<Key extends Comparable<Key>>	
	MaxPQ()	创建一个优先队列
	MaxPQ(int max)	创建一个最大容量为 max 的优先队列

（续）

public class	MaxPQ<Key extends Comparable<Key>>	
	MaxPQ(Key[] a)	用 a[] 中的元素创建一个优先队列
void	Insert(Key v)	向优先队列中插入一个元素
Key	max()	返回最大元素
Key	delMax()	删除并返回最大元素
boolean	isEmpty()	返回队列是否为空
309 int	size()	返回优先队列中的元素个数

为了用例代码的方便，API 包含的三个构造函数使得用例可以构造指定大小的优先队列（还可以用给定的一个数组将其初始化）。为了使用例代码更加清晰，我们会在适当的地方使用另一个类 MinPQ。它和 MaxPQ 类似，只是含有一个 delMin() 方法来删除并返回队列中键值最小的那个元素。MaxPQ 的任意实现都能很容易地转化为 MinPQ 的实现，反之亦然，只需要改变一下 less() 比较的方向即可。

优先队列的调用示例

为了展示优先队列的抽象模型的价值，考虑以下问题：输入 N 个字符串，每个字符串都对映着一个整数，你的任务就是从中找出最大的（或是最小的）M 个整数（及其关联的字符串）。这些输入可能是金融事务，你需要从中找出最大的那些；或是农产品中的杀虫剂含量，这时你需要从中找出最小的那些；或是服务请求、科学实验的结果，或是其他应用。在某些应用场景中，输入量可能非常巨大，甚至可以认为输入是无限的。解决这个问题的一种方法是将输入排序然后从中找出 M 个最大的元素，但我们已经说明输入将会非常庞大。另一种方法是将每个新的输入和已知的 M 个最大元素比较，但除非 M 较小，否则这种比较的代价会非常高昂。只要我们能够高效地实现 insert() 和 delMin()，下面的**优先队列用例**中调用了 MinPQ 的 TopM 就能使用优先队列解决这个问题，这就是本节中我们的目标。在现代基础性计算环境中超大的输入 N 非常常见，这些实现使我们能够解决以前缺乏足够资源去解决的问题，如表 2.4.2 所示。

表 2.4.2 从 N 个输入中找到最大的 M 个元素所需成本

示 例	增长的数量级	
	时 间	空 间
排序算法的用例	$N\log N$	N
调用初级实现的优先队列	NM	M
调用基于堆实现的优先队列	$N\log M$	M

一个优先队列的用例

```
public class TopM
{
   public static void main(String[] args)
   {  // 打印输入流中最大的M行
      int M = Integer.parseInt(args[0]);
      MinPQ<Transaction> pq = new MinPQ<Transaction>(M+1);
      while (StdIn.hasNextLine())
      {  // 为下一行输入创建一个元素并放入优先队列中
         pq.insert(new Transaction(StdIn.readLine()));
         if (pq.size() > M)
```

```
            pq.delMin();      // 如果优先队列中存在M+1个元素则删除其中最小的元素
        } // 最大的M个元素都在优先队列中

        Stack<Transaction> stack = new Stack<Transaction>();
        while (!pq.isEmpty()) stack.push(pq.delMin());
        for (Transaction t : stack) StdOut.println(t);
    }
}
```

从命令行输入一个整数 M 以及一系列字符串，每一行表示一个事务。这段代码调用了 MinPQ 并会打印数字最大的 M 行。它用到了 Transaction 类（请见表 1.2.6，练习 1.2.19 和练习 2.1.21），构造了一个用数字作为键的优先队列。当优先队列的大小超过 M 时就删掉其中最小的元素。所有事务输入完毕之后程序会从优先队列中按递减顺序打印出最大的 M 个事务。这段代码相当于将所有事务放入一个栈，遍历栈以颠倒它们的顺序并按照增序将它们打印出来。

```
% more tinyBatch.txt
Turing      6/17/1990   644.08
vonNeumann  3/26/2002  4121.85
Dijkstra    8/22/2007  2678.40
vonNeumann  1/11/1999  4409.74
Dijkstra   11/18/1995   837.42
Hoare       5/10/1993  3229.27
vonNeumann  2/12/1994  4732.35
Hoare       8/18/1992  4381.21
Turing      1/11/2002    66.10
Thompson    2/27/2000  4747.08
Turing      2/11/1991  2156.86
Hoare       8/12/2003  1025.70
vonNeumann 10/13/1993  2520.97
Dijkstra    9/10/2000   708.95
Turing     10/12/1993  3532.36
Hoare       2/10/2005  4050.20
```

```
% java TopM 5 < tinyBatch.txt
Thompson    2/27/2000  4747.08
vonNeumann  2/12/1994  4732.35
vonNeumann  1/11/1999  4409.74
Hoare       8/18/1992  4381.21
vonNeumann  3/26/2002  4121.85
```

2.4.2 初级实现

我们在第 1 章中讨论过的 4 种基础数据结构是实现优先队列的起点。我们可以使用有序或无序的数组或链表。在队列较小时，大量使用两种主要操作之一时，或是所操作元素的顺序已知时，它们十分有用。因为这些实现相对简单，我们在这里只给出文字描述并将实现代码作为练习（请见练习 2.4.3）。

2.4.2.1 数组实现（无序）

或许实现优先队列的最简单方法就是基于 2.1 节中下压栈的代码。insert() 方法的代码和栈的 push() 方法完全一样。要实现删除最大元素，我们可以添加一段类似于选择排序的内循环的代码，将最大元素和边界元素交换然后删除它，和我们对栈的 pop() 方法的实现一样。和栈类似，我们也可以加入调整数组大小的代码来保证数据结构中至少含有四分之一的元素而又永远不会溢出。

310 ≀ 311

2.4.2.2 数组实现（有序）

另一种方法就是在 insert() 方法中添加代码，将所有较大的元素向右边移动一格以使数组保持有序（和插入排序一样）。这样，最大的元素总会在数组的一边，优先队列的删除最大元素操作就和栈的 pop() 操作一样了。

2.4.2.3 链表表示法

和刚才类似，我们可以用基于链表的下压栈的代码作为基础，而后可以选择修改 pop() 来找到并返回最大元素，或是修改 push() 来保证所有元素为逆序并用 pop() 来删除并返回链表的首元素（也就是最大的元素）。

使用无序序列是解决这个问题的惰性方法，我们仅在必要的时候才会采取行动（找出最大元素）；使用有序序列则是解决问题的积极方法，因为我们会尽可能未雨绸缪（在插入元素时就保持列表有序），使后续操作更高效。

实现栈或是队列与实现优先队列的最大不同在于对性能的要求。对于栈和队列，我们的实现能够在常数时间内完成所有操作；而对于优先队列，我们刚刚讨论过的所有初级实现中，插入元素和删除最大元素这两个操作之一在最坏情况下需要线性时间来完成（如表 2.4.3 所示）。我们接下来要讨论的基于数据结构堆的实现能够保证这两种操作都能更快地执行。

表 2.4.3 优先队列的各种实现在最坏情况下运行时间的增长数量级

数据结构	插入元素	删除最大元素
有序数组	N	1
无序数组	1	N
堆	$\log N$	$\log N$
理想情况	1	1

在一个优先队列上执行的一系列操作如表 2.4.4 所示。

表 2.4.4 在一个优先队列上执行的一系列操作

操作	参数	返回值	大小	内容（无序）	内容（有序）
插入元素	P		1	**P**	**P**
插入元素	Q		2	P **Q**	P **Q**
插入元素	E		3	P Q **E**	**E** P Q
删除最大元素		**Q**	2	P E	E P
插入元素	X		3	P E **X**	E P **X**
插入元素	A		4	P E X **A**	**A** E P X
插入元素	M		5	P E X A **M**	A E **M** P X
删除最大元素		**X**	4	P E M A	A E M P
插入元素	P		5	P E M A **P**	A E M P **P**
插入元素	L		6	P E M A P **L**	A E **L** M P
插入元素	E		7	P E M A P L **E**	A E **E** L M
312 删除最大元素		**P**	6	E E M A P L	A E E L M

2.4.3 堆的定义

数据结构二叉堆能够很好地实现优先队列的基本操作。在二叉堆的数组中，每个元素都要保证大于等于另两个特定位置的元素。相应地，这些位置的元素又至少要大于等于数组中的另两个元素，以此类推。如果我们将所有元素画成一棵二叉树，将每个较大元素和两个较小的元素用边连接就可以很容易看出这种结构。

定义。当一棵二叉树的每个结点都大于等于它的两个子结点时，它被称为堆有序。

相应地，在堆有序的二叉树中，每个结点都小于等于它的父结点（如果有的话）。从任意结点向上，我们都能得到一列非递减的元素；从任意结点向下，我们都能得到一列非递增的元素。特别地：

命题 O。根结点是堆有序的二叉树中的最大结点。

证明。根据树的性质归纳可得。

二叉堆表示法

如果我们用指针来表示堆有序的二叉树，那么每个元素都需要三个指针来找到它的上下结点（父结点和两个子结点各需要一个）。但如图 2.4.1 所示，如果我们使用完全二叉树，表达就会变得特别方便。要画出这样一棵完全二叉树，可以先定下根结点，然后一层一层地由上向下、从左至右，在每个结点的下方连接两个更小的结点，直至将 N 个结点全部连接完毕。完全二叉树只用数组而不需要指针就可以表示。具体方法就是将二叉树的结点按照层级顺序放入数组中，根结点在位置 1，它的子结点在位置 2 和 3，而子结点的子结点则分别在位置 4、5、6 和 7，以此类推。

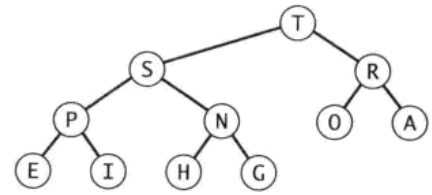


图 2.4.1 一棵堆有序的完全二叉树

313

定义。二叉堆是一组能够用堆有序的完全二叉树排序的元素，并在数组中按照层级储存（不使用数组的第一个位置）。

（简单起见，在下文中我们将二叉堆简称为堆）在一个堆中，位置 k 的结点的父结点的位置为 $\lfloor k/2 \rfloor$，而它的两个子结点的位置则分别为 $2k$ 和 $2k+1$。这样在不使用指针的情况下（我们在第 3 章中讨论二叉树时会用到它们）我们也可以通过计算数组的索引在树中上下移动：从 a[k] 向上一层就令 k 等于 k/2，向下一层则令 k 等于 2k 或 2k+1。

用数组（堆）实现的完全二叉树的结构是很严格的，但它的灵活性已经足以让我们高效地实现优先队列。用它们我们将能实现对数级别的插入元素和删除最大元素的操作。利用在数组中无需指针即可沿树上下移动的便利和以下性质，算法保证了对数复杂度的性能。

命题 P。一棵大小为 N 的完全二叉树的高度为 $\lfloor \lg N \rfloor$。

证明。通过归纳很容易可以证明这一点，且当 N 达到 2 的幂时树的高度会加 1。

堆的表示如图 2.4.2 所示。

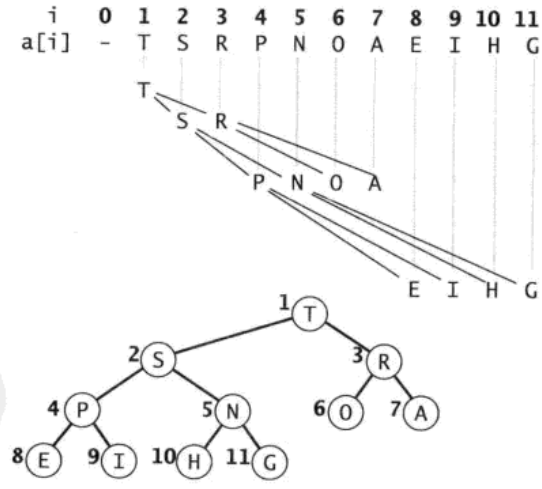


图 2.4.2 堆的表示

314

2.4.4 堆的算法

我们用长度为 $N+1$ 的私有数组 pq[] 来

表示一个大小为 N 的堆，我们不会使用 pq[0]，堆元素放在 pq[1] 至 pq[N] 中。在排序算法中，我们只通过私有辅助函数 less() 和 exch() 来访问元素，但因为所有的元素都在数组 pq[] 中，我们在 2.4.4.2 节中会使用更加紧凑的实现方式，不再将数组作为参数传递。堆的操作会首先进行一些简单的改动，打破堆的状态，然后再遍历堆并按照要求将堆的状态恢复。我们称这个过程叫做*堆的有序化*（reheapifying）。

```
private boolean less(int i, int j)
{  return pq[i].compareTo(pq[j]) < 0;  }

private void exch(int i, int j)
{  Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;  }
```

堆实现的比较和交换方法

堆实现的比较和交换方法如右上方的代码框所示。

在有序化的过程中我们会遇到两种情况。当某个结点的优先级上升（或是在堆底加入一个新的元素）时，我们需要由下至上恢复堆的顺序。当某个结点的优先级下降（例如，将根结点替换为一个较小的元素）时，我们需要由上至下恢复堆的顺序。首先，我们会学习如何实现这两种辅助操作，然后再用它们实现插入元素和删除最大元素的操作。

2.4.4.1 由下至上的堆有序化（上浮）

如果堆的有序状态因为某个结点变得比它的父结点更大而被打破，那么我们就需要通过交换它和它的父结点来修复堆。交换后，这个结点比它的两个子结点都大（一个是曾经的父结点，另一个比它更小，因为它是曾经父结点的子结点），但这个结点仍然可能比它现在的父结点更大。我们可以一遍遍地用同样的办法恢复秩序，将这个结点不断向上移动直到我们遇到了一个更大的父结点。只要记住位置 k 的结点的父结点的位置是 $\lfloor k/2 \rfloor$，这个过程实现起来很简单。swim() 方法中的循环可以保证只有位置 k 上的结点大于它的父结点时堆的有序状态才会被打破。因此只要该结点不再大于它的父结点，堆的有序状态就恢复了。至于方法名，当一个结点太大的时候它需要浮（swim）到堆的更高层。由下至上的堆有序化的实现代码如右上方所示。

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k/2, k);
      k = k/2;
   }
}
```

由下至上的堆有序化（上浮）的实现

图 2.4.3 展示的是由下至上的堆有序化示意图。

2.4.4.2 由上至下的堆有序化（下沉）

如果堆的有序状态因为某个结点变得比它的两个子结点或是其中之一更小了而被打破了，那么我们可以通过将它和它的两个子结点中的较大者交换来恢复堆。交换可能会在子结点处继续打破堆的有序状态，因此我们需要不断地用相同的方式将其修复，将结点向下移动直到它的子结点都比它更小或是到达了堆的底部。由位置为 k 的结点的子结点位于 $2k$ 和 $2k+1$ 可以直接得到对应的代码。至于方法名，由上至下的堆有序化的示意图及实现代码分别见图 2.4.4 和下页的代码框。当一个结点太小的时候它需要沉（sink）到堆的更低层。

如果我们把堆想象成一个严密的黑社会组织，每个子结点都表示一个下属（父结点则表示它的直接上级），那么这些操作就可以得到很有趣的解释。swim() 表示一个很有能力的新人加入组织并被逐级提升（将能力不够的上级踩在脚下），直到他遇到了一个更强的领导。sink() 则类似于整个社团的领导退休并被外来者取代之后，如果他的下属比他更厉害，他们的角色就会交换，这种交换会持续下去直到他的能力比其下属都强为止。这些理想化的情景在现实生活中可能很罕见，但它们能够帮助你理解堆的这些基本行为。

sink() 和 swim() 方法是高效实现优先队列 API 的基础，原因如下（具体的实现请见算法 2.6）。

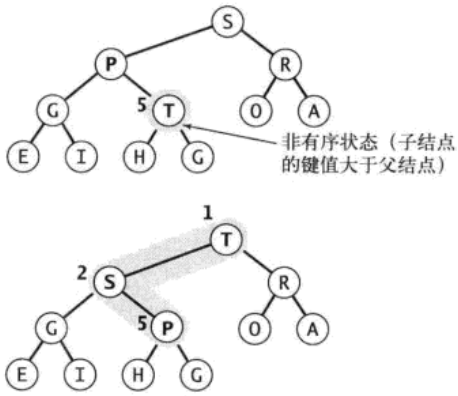


图 2.4.3 由下至上的堆有序化（上浮）

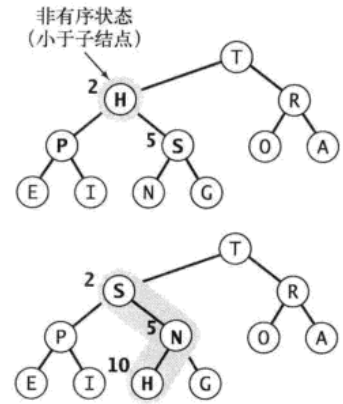


图 2.4.4 由上至下的堆有序化（下沉）

插入元素。我们将新元素加到数组末尾，增加堆的大小并让这个新元素上浮到合适的位置（如图 2.4.5 左半部分所示）。

删除最大元素。我们从数组顶端删去最大的元素并将数组的最后一个元素放到顶端，减小堆的大小并让这个元素下沉到合适的位置（如图 2.4.5 右半部分所示）。

算法 2.6 解决了我们在本节开始时提出的一个基本问题：它对优先队列 API 的实现能够保证插入元素和删除最大元素这两个操作的用时和队列的大小仅成对数关系。

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

由上至下的堆有序化（下沉）的实现

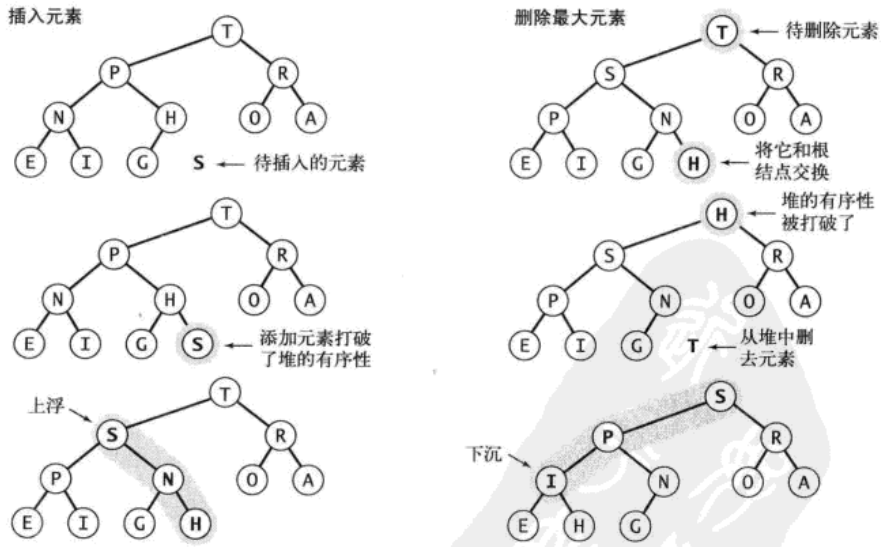


图 2.4.5 堆的操作

315～317

算法 2.6 基于堆的优先队列

```
public class MaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;                  // 基于堆的完全按二叉树
   private int N = 0;                 // 存储于pq[1..N]中，pq[0]没有使用

   public MaxPQ(int maxN)
   {  pq = (Key[]) new Comparable[maxN+1];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public int size()
   {  return N;  }

   public void insert(Key v)
   {
      pq[++N] = v;
      swim(N);
   }

   public Key delMax()

   {
      Key max = pq[1];                // 从根结点得到最大元素
      exch(1, N--);                   // 将其和最后一个结点交换
      pq[N+1] = null;                 // 防止越界
      sink(1);                        // 恢复堆的有序性
      return max;
   }

   // 辅助方法的实现请见本节前面的代码框
   private boolean less(int i, int j)
   private void exch(int i, int j)
   private void swim(int k)
   private void sink(int k)
}
```

优先队列由一个基于堆的完全二叉树表示，存储于数组 pq[1..N] 中，pq[0] 没有使用。在 insert() 中，我们将 N 加一并把新元素添加在数组最后，然后用 swim() 恢复堆的秩序。在 delMax() 中，我们从 pq[1] 中得到需要返回的元素，然后将 pq[N] 移动到 pq[1]，将 N 减一并用 sink() 恢复堆的秩序。同时我们还将不再使用的 pq[N+1] 设为 null，以便系统回收它所占用的空间。和以前一样（请见 1.3 节），
318 这里省略了动态调整数组大小的代码。其他的构造函数请见练习 2.4.19。

命题 Q。对于一个含有 N 个元素的基于堆的优先队列，插入元素操作只需不超过（$\lg N+1$）次比较，删除最大元素的操作需要不超过 $2\lg N$ 次比较。

证明。由命题 P 可知，两种操作都需要在根结点和堆底之间移动元素，而路径的长度不超过 $\lg N$。对于路径上的每个结点，删除最大元素需要两次比较（除了堆底元素），一次用来找出较大的子结点，一次用来确定该子结点是否需要上浮。

对于需要大量混杂的插入和删除最大元素操作的典型应用来说，命题 Q 意味着一个重要的性能突破，总结请见表 2.4.3。使用有序或是无序数组的优先队列的初级实现总是需要线性时间来完成其

中一种操作，但基于堆的实现则能够保证在对数时间内完成它们。这种差别使得我们能够解决以前无法解决的问题。

2.4.4.3 多叉堆

基于用数组表示的完全三叉树构造堆并修改相应的代码并不困难。对于数组中 1 至 N 的 N 个元素，位置 k 的结点大于等于位于 $3k-1$、$3k$ 和 $3k+1$ 的结点，小于等于位于 $\lfloor (k+1)/3 \rfloor$ 的结点。甚至对于给定的 d，将其修改为任意的 d 叉树也并不困难。我们需要在树高（$\log_d N$）和在每个结点的 d 个子结点找到最大者的代价之间找到折中，这取决于实现的细节以及不同操作的预期相对频繁程度。

堆上的优先队列操作如图 2.4.6 所示。

2.4.4.4 调整数组大小

我们可以添加一个没有参数的构造函数，在 `insert()` 中添加将数组长度加倍的代码，在 `delMax()` 中添加将数组长度减半的代码，就像在 1.3 节中的栈那样。这样，算法的用例就无需关注各种队列大小的限制。当优先队列的数组大小可以调整、队列长度可以是任意值时，**命题 Q** 指出的对数时间复杂度上限就只是针对一般性的队列长度 N 而言了（请见练习 2.4.22）。

2.4.4.5 元素的不可变性

优先队列存储了用例创建的对象，但同时假设用例代码不会改变它们（改变它们就可能打破堆的有序性）。我们可以将这个假设转化为强制条件，但程序员通常不会这么做，因为增加代码的复杂性会降低性能。

2.4.4.6 索引优先队列

在很多应用中，允许用例引用已经进入优先队列中的元素是有必要的。做到这一点的一种简单方法是给每个元素一个*索引*。另外，一种常见的情况是用例已经有了总量为 N 的多个元素，而且可能还同时使用了多个（平行）数组来存储这些元素的信息。此时，其他无关的用例代码可能已经在使用一个整数索引来引用这些元素了。这些考虑引导我们设计了表 2.4.5 中的 API。

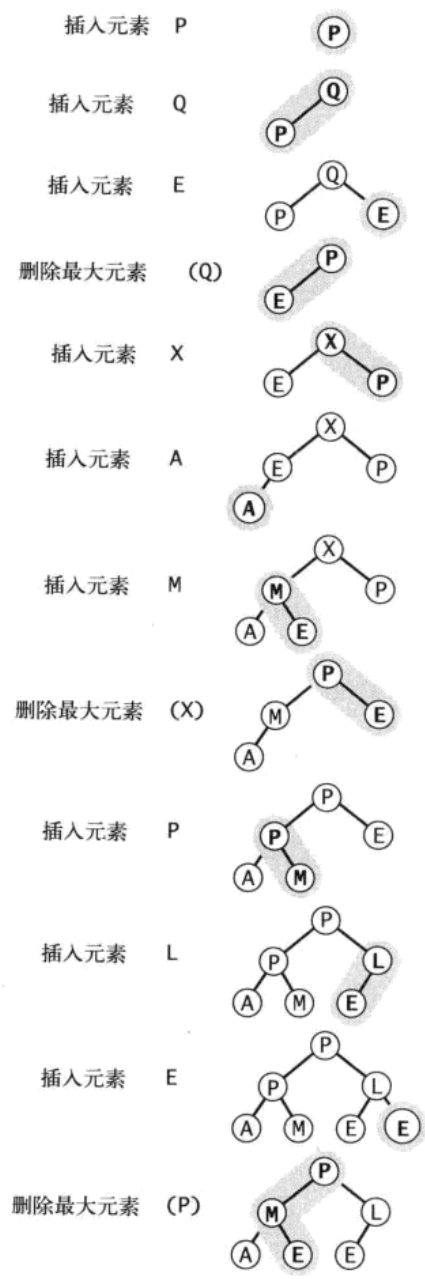


图 2.4.6 在堆上的优先队列操作

表 2.4.5 关联索引的泛型优先队列的 API

public class	**IndexMinPQ**<Item extends Comparable<Item>>	
	IndexMinPQ(int maxN)	创建一个最大容量为 maxN 的优先队列，索引的取值范围为 0 至 maxN-1
void	insert(int k, Item item)	插入一个元素，将它和索引 k 相关联
void	change(int k, Item item)	将索引为 k 的元素设为 item
boolean	contains(int k)	是否存在索引为 k 的元素

（续）

public class	IndexMinPQ<Item extends Comparable<Item>>	
void	delete(int k)	删去索引 k 及其相关联的元素
Item	min()	返回最小元素
int	minIndex()	返回最小元素的索引
int	delMin()	删除最小元素并返回它的索引
boolean	isEmpty()	优先队列是否为空
int	size()	优先队列中的元素数量

319 ~ 320

理解这种数据结构的一个较好方法是将它看成一个能够快速访问其中最小元素的数组。事实上它还要更好——它能够快速访问数组的一个特定子集中的最小元素（指所有被插入的元素）。换句话说，可以将名为 pq 的 IndexMinPQ 类优先队列看做数组 pq[0..N-1] 中的一部分元素的代表。将 pq.insert(k, item) 看做将 k 加入这个子集并使 pq[k] = item，pq.change(k, item) 则代表令 pq[k]=item。这两种操作没有改变其他操作所依赖的数据结构，其中最重要的就是 delMin()（删除最小元素并返回它的索引）和 change()（改变数据结构中的某个元素的索引——即 pq[i]=item）。这些操作在许多应用中都很重要并且依赖于对元素的引用（索引）。练习 2.4.33 说明了如何用较少的代码将算法 2.6 扩展为极高效的索引优先队列。一般来说，当堆发生变化时，我们会用下沉（元素减小时）或上浮（元素变大时）操作来恢复堆的有序性。在这些操作中，我们可以用索引查找元素。能够定位堆中的任意元素也使我们能够在 API 中加入一个 delete() 操作。

命题 Q（续）。在一个大小为 N 的索引优先队列中，插入元素（insert）、改变优先级（change）、删除（delete）和删除最小元素（remove the minimum）操作所需的比较次数和 $\log N$ 成正比（如表 2.4.6 所示）。

证明。已知堆中所有路径最长即为 $\sim \lg N$，从代码中很容易得到这个结论。

表 2.4.6 含有 N 个元素的基于堆的索引优先队列所有操作在最坏情况下的成本

操 作	比较次数的增长数量级
insert()	$\log N$
change()	$\log N$
contains()	1
delete()	$\log N$
min()	1
minIndex()	1
delMin()	$\log N$

这段讨论针对的是找出最小元素的队列；和以前一样，我们也在本书网站上实现了一个找出最大元素的版本 IndexMaxPQ。

2.4.4.7 索引优先队列用例

下面的用例调用了 IndexMinPQ 的代码 Multiway 解决了多向归并问题：它将多个有序的输入流归并成一个有序的输出流。许多应用中都会遇到这个问题。输入可能来自于多种科学仪器的输出（按时间排序），或是来自多个音乐或电影网站的信息列表（按名称或艺术家名字排序），或是商业交易（按账号或时间排序），或者其他。如果有足够的空间，你可以把它们简单地读入一个数组

并排序，但如果用了优先队列，无论输入有多长你都可以把它们全部读入并排序。 321

使用优先队列的多向归并

```
public class Multiway
{
   public static void merge(In[] streams)
   {
      int N = streams.length;
      IndexMinPQ<String> pq = new IndexMinPQ<String>(N);

      for (int i = 0; i < N; i++)
         if (!streams[i].isEmpty())
             pq.insert(i, streams[i].readString());

      while (!pq.isEmpty())
      {
         StdOut.println(pq.min());
         int i = pq.delMin();

         if (!streams[i].isEmpty())
             pq.insert(i, streams[i].readString());
      }
   }

   public static void main(String[] args)
   {
      int N = args.length;
      In[] streams = new In[N];
      for (int i = 0; i < N; i++)
          streams[i] = new In(args[i]);
      merge(streams);
   }
}
```

这段代码调用了 IndexMinPQ 来将作为命令行参数输入的多行有序字符串归并为一行有序的输出（请见正文）。每个输入流的索引都关联着一个元素（输入中的下个字符串）。初始化之后，代码进入一个循环，删除并打印出队列中最小的字符串，然后将该输入的下一个字符串添加为一个元素。为了节约，下面将所有的输出排在了一行——实际输出应该是一个字符串一行。

```
% more m1.txt
A B C F G I I Z
% more m2.txt
B D H P Q Q
% more m3.txt
A B E F J N

% java Multiway m1.txt m2.txt m3.txt
A A B B B C D E F F G H I I J N P Q Q Z
```

322

2.4.5 堆排序

我们可以把任意优先队列变成一种排序方法。将所有元素插入一个查找最小元素的优先队列，然后再重复调用删除最小元素的操作来将它们按顺序删去。用无序数组实现的优先队列这么做相当于进行一次插入排序。用基于堆的优先队列这样做等同于哪种排序？一种全新的排序方法！下面我们就用堆来实现一种经典而优雅的排序算法——堆排序。

堆排序可以分为两个阶段。在堆的构造阶段中，我们将原始数组重新组织安排进一个堆中；然后在下沉排序阶段，我们从堆中按递减顺序取出所有元素并得到排序结果。为了和我们已经学习过的代码保持一致，我们将使用一个面向最大元素的优先队列并重复删除最大元素。为了排序的需要，

我们不再将优先队列的具体表示隐藏，并将直接使用 swim() 和 sink() 操作。这样我们在排序时就可以将需要排序的数组本身作为堆，因此无需任何额外空间。

2.4.5.1 堆的构造

由 N 个给定的元素构造一个堆有多难？我们当然可以在与 $N\log N$ 成正比的时间内完成这项任务，只需从左至右遍历数组，用 swim() 保证扫描指针左侧的所有元素已经是一棵堆有序的完全树即可，就像连续向优先队列中插入元素一样。一个更聪明更高效的办法是从右至左用 sink() 函数构造子堆。数组的每个位置都已经是一个子堆的根结点了，sink() 对于这些子堆也适用。如果一个结点的两个子结点都已经是堆了，那么在该结点上调用 sink() 可以将它们变成一个堆。这个过程会递归地建立起堆的秩序。开始时我们只需要扫描数组中的一半元素，因为我们可以跳过大小为 1 的子堆。最后我们在位置 1 上调用 sink() 方法，扫描结束。在排序的第一阶段，堆的构造方法和我们的想象有所不同，因为我们的目标是构造一个堆有序的数组并使最大元素位于数组的开头（次大的元素在附近）而非构造函数结束的末尾。

命题 R。用下沉操作由 N 个元素构造堆只需少于 $2N$ 次比较以及少于 N 次交换。

证明。观察可知，构造过程中处理的堆都较小。例如，要构造一个 127 个元素的堆，我们会处理 32 个大小为 3 的堆，16 个大小为 7 的堆，8 个大小为 15 的堆，4 个大小为 31 的堆，2 个大小为 63 的堆和 1 个大小为 127 的堆，因此（最坏情况下）需要 $32\times1+16\times2+8\times3+4\times4+2\times5+1\times6=120$ 次交换（两倍于比较）。完整证明请见练习 2.4.20。

323

堆排序的实现过程如算法 2.7 所示。

算法 2.7 堆排序

```
public static void
sort(Comparable[] a)
{
   int N = a.length;
   for (int k = N/2; k >= 1;
k--)
      sink(a, k, N);
   while (N > 1)
   {
      exch(a, 1, N--);
      sink(a, 1, N);
   }
}
```

这段代码用 sink() 方法将 a[1] 到 a[N] 的元素排序（sink() 被修改过，以 a[] 和 N 作为参数）。for 循环构造了堆，然后 while 循环将最大的元素 a[1] 和 a[N] 交换并修复了堆，如此重复直到堆变空。将 exch() 和 less() 的实现中的索引减一即可得到和其他排序算法一致的实现（将 a[0] 至 a[N-1] 排序）。堆排序具体流程示意图显示在图 2.4.7 中。

324

		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
初始值			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	**L**	X	A	M	P	**E**	E
11	4		S	O	R	**T**	L	X	A	M	P	E	E
11	3		S	O	**X**	T	L	**R**	A	M	P	E	E
11	2		S	**T**	X	**P**	L	R	A	**M**	**O**	E	E
11	1		**X**	T	**S**	P	L	R	A	M	O	E	E
堆有序			X	T	S	P	L	R	A	M	O	E	E
10	1		**T**	**P**	S	**O**	L	R	A	M	**E**	E	**X**
9	1		**S**	P	**R**	O	L	**E**	A	M	E	**T**	X
8	1		**R**	P	**E**	O	L	**E**	A	M	**S**	T	X
7	1		**P**	**O**	E	**M**	L	E	A	**R**	S	T	X
6	1		**O**	**M**	E	**A**	L	E	**P**	R	S	T	X
5	1		**M**	**L**	E	A	**E**	**O**	P	R	S	T	X
4	1		**L**	**E**	E	A	**M**	O	P	R	S	T	X
3	1		**E**	**A**	E	**L**	M	O	P	R	S	T	X
2	1		**E**	A	**E**	L	M	O	P	R	S	T	X
1	1		**A**	**E**	E	L	M	O	P	R	S	T	X
排序结果			A	E	E	L	M	O	P	R	S	T	X

堆排序的轨迹（每次下沉后的数组内容）

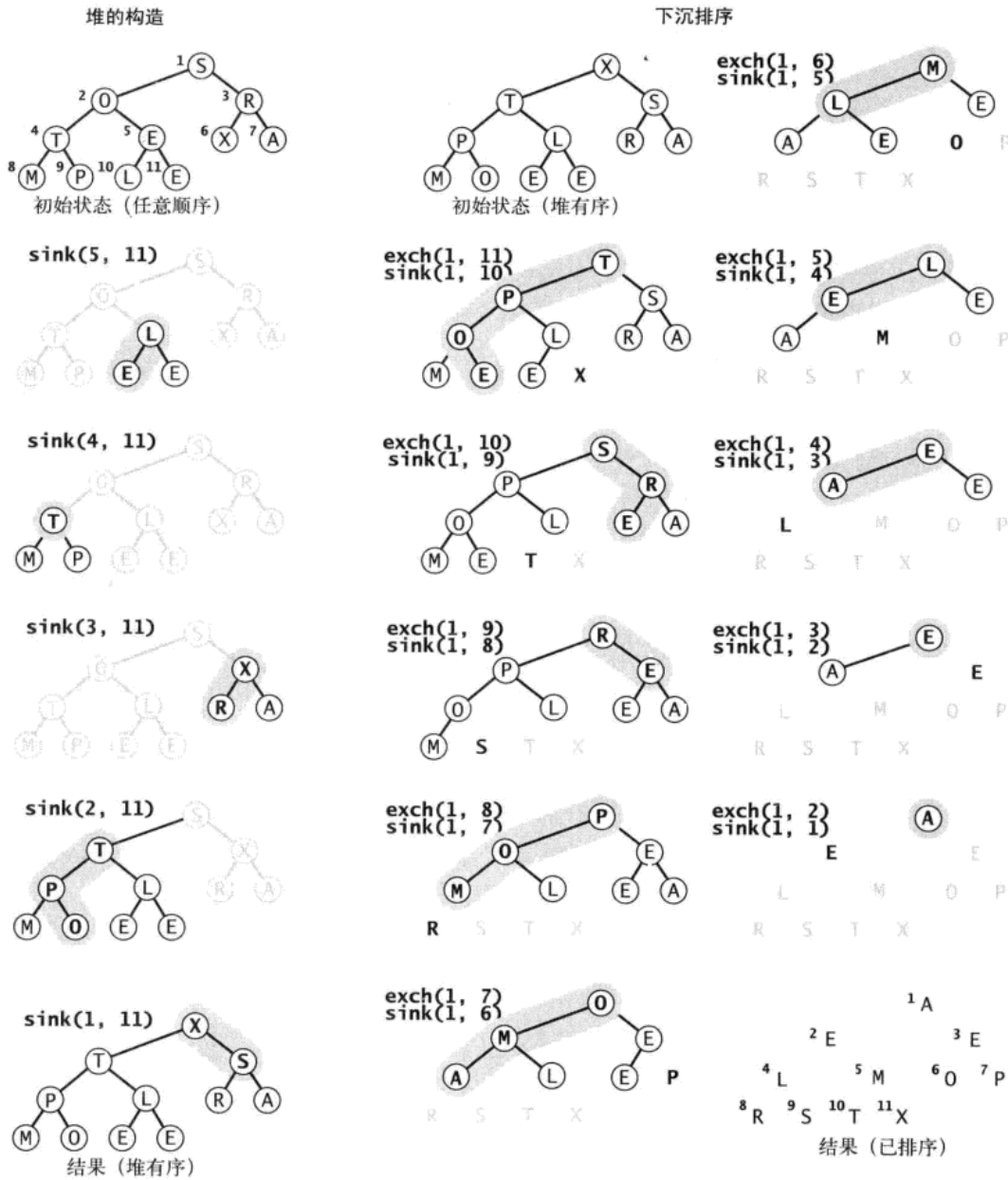


图 2.4.7 堆排序：堆的构造（左）和下沉排序（右） 325

2.4.5.2 下沉排序

堆排序的主要工作都是在第二阶段完成的。这里我们将堆中的最大元素删除，然后放入堆缩小后数组中空出的位置。这个过程和选择排序有些类似（按照降序而非升序取出所有元素），但所需的比较要少得多，因为堆提供了一种从未排序部分找到最大元素的有效方法。

命题S。将N个元素排序，堆排序只需少于$(2N\lg N+2N)$次比较（以及一半次数的交换）。

证明。$2N$项来自于堆的构造（见命题R）。$2N\lg N$项来自于每次下沉操作最大可能需要$2\lg N$次比较（见命题P与命题Q）。

算法2.7完整地实现了这些思想，也就是经典的堆排序算法。它的发明人是J. W. J. Williams，并由R. W. Floyd在1964年改进。尽管这段程序中循环的任务各不同（第一段循环构造堆，第二段循环在下沉排序中销毁堆），它们都是基于`sink()`方法。我们将该实现和优先队列的API独立开来是为了突出这个排序算法的简洁性（`sort()`方法只需8行代码，`sink()`函数8行），并使其可以嵌入其他代码之中。

和以前一样，通过研究可视轨迹（如图2.4.8所示）我们可以深入了解算法的操作。一开始算法的行为似乎杂乱无章，因为随着堆的构建较大的元素都被移动到了数组的开头，但接下来算法的行为看起来就和选择排序一模一样了（除了它比较的次数少得多）。

和我们学过的其他算法一样，很多人都研究过许多改进基于堆的优先队列的实现和堆排序的方法。我们这里简要地看看其中之一。

2.4.5.3 先下沉后上浮

大多数在下沉排序期间重新插入堆的元素会被直接加入到堆底。Floyd在1964年观察发现，我们正好可以通过免去检查元素是否到达正确位置来节省时间。在下沉中总是直接提升较大的子结点直至到达堆底，然后再使元素上浮到正确的位置。这个想法几乎可以将比较次数减少一半——接近了归并排序所需的比较次数（随机数组）。这种方法需要额外的空间，因此在实际应用中只有当比较操作代价较高时才有用（例如，当我们在将字符串或者其他键值较长类型的元素进行排序时）。

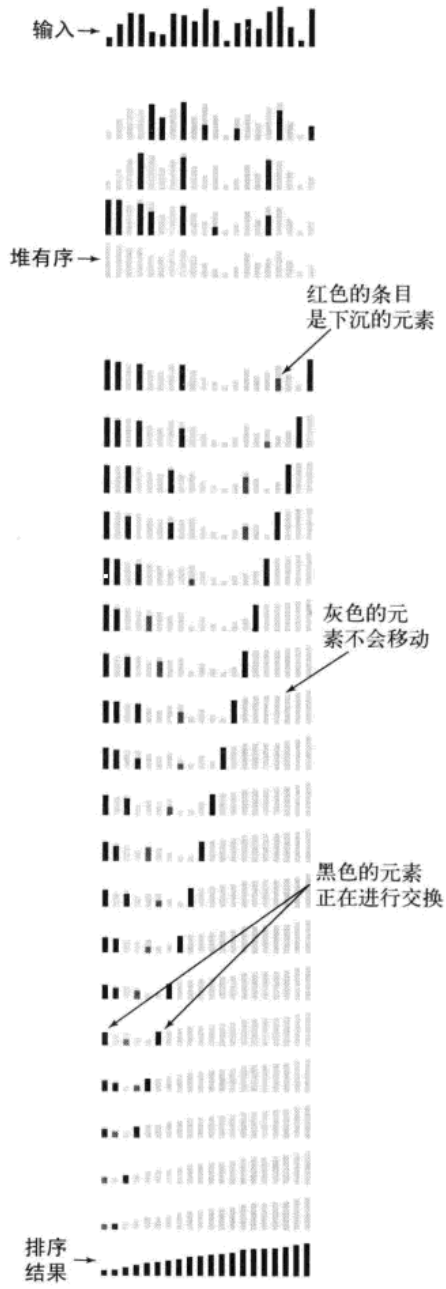


图2.4.8 堆排序的可视轨迹（另见彩插）

堆排序在排序复杂性的研究中有着重要的地位，因为它是我们所知的唯一能够同时最优地利用空间和时间的方法——在最坏的情况下它也能保证使用~ $2N\lg N$次比较和恒定的额外空间。当空间十分紧张的时候（例如在嵌入式系统或低成本的移动设备中）它很流行，因为它只用几行就能实现（甚至机器码也是）较好的性能。但现代系统的许多应用很少使用它，因为它无法利用缓存。数组元素很少和相邻的其他元素进行比较，因此缓存未命中的次数要远远高于大多数比较都在相邻元素间进行的算法，如快速排序、归并排序，甚至是希尔排序。

另一方面，用堆实现的优先队列在现代应用程序中越来越重要，因为它能在插入操作和删除最大元素操作混合的动态场景中保证对数级别的运行时间。我们会在本书后续章节见到更多的例子。 326 ~ 327

答疑

问 我还是不明白优先队列是做什么用的。为什么我们不直接把元素排序然后再一个个地引用有序数组中的元素？

答 在某些数据处理的例子里，比如 TopM 和 Multiway，总数据量太大，无法排序（甚至无法全部装进内存）。如果你需要从 10 亿个元素中选出最大的十个，你真的想把一个 10 亿规模的数组排序吗？但有了优先队列，你就只用一个能存储十个元素的队列即可。在其他的例子中，我们甚至无法同时获取所有的数据，因此只能先从优先队列中取出并处理一部分，然后再根据结果决定是否向优先队列中添加更多的数据。

问 为什么不像我们在其他排序算法中那样使用 Comparable 接口，而在 MaxPQ 中使用泛型的 Item 呢？

答 这么做的话 delMax() 的用例就需要将返回值转换为某种具体的类型，比如 String。一般来说，应该尽量避免在用例中进行类型转换。

问 为什么在堆的表示中不使用 a[0] ？

答 这么做可以稍稍简化计算。实现从 0 开始的堆并不困难，a[0] 的子结点是 a[1] 和 a[2]，a[1] 的子结点是 a[3] 和 a[4]，a[2] 的子结点是 a[5] 和 a[6]，以此类推。但大多数程序员更喜欢我们的简单方法。另外，将 a[0] 的值用作哨兵（作为 a[1] 的父结点）在某些堆的应用中很有用。

问 在我看来，在堆排序中构造堆时，逐个向堆中添加元素比 2.4.5.1 节中描述的由底向上的复杂方法更简单。为什么要这么做？

答 对于一个排序算法来说，这么做能够快上 20%，而且所需的代码更少（不会用到 swim() 函数）。理解算法的难度并不一定与它的简洁性或者效率相关。

问 如果我去掉 MaxPQ 的实现中的 extends Comparable<Key> 这句话会怎样？

答 和以前一样，回答这类问题的最简单的办法就是你自己直接试试。如果这么做 MaxPQ 会报出一个编译错误：

```
MaxPQ.java:21: cannot find symbol
symbol  : method compareTo(Item)
```

Java 这样告诉你它不知道 Item 对象的 compareTo() 方法，因为你没有声明 Item extends Comparable<Item>。 328

练习

2.4.1 用序列 P R I O * R * * I * T * Y * * * Q U E * * * U * E （字母表示插入元素，星号表示删除最大元素）操作一个初始为空的优先队列。给出每次删除最大元素返回的字符。

2.4.2 分析以下说法：要实现在常数时间找到最大元素，为何不用一个栈或队列，然后记录已插入的最大元素并在找出最大元素时返回它的值？

2.4.3 用以下数据结构实现优先队列，支持插入元素和删除最大元素的操作：无序数组、有序数组、无序链表和链表。将你的 4 种实现中每种操作在最坏情况下的运行时间上下限制成一张表格。

2.4.4 一个按降序排列的数组也是一个面向最大元素的堆吗？

2.4.5 将EASYQUESTION顺序插入一个面向最大元素的堆中，给出结果。

2.4.6 按照练习2.4.1的规则，用序列P R I O * R * * I * T * Y * * * Q U E * * * U * E操作一个初始为空的面向最大元素的堆，给出每次操作后堆的内容。

2.4.7 在堆中，最大的元素一定在位置1上，第二大的元素一定在位置2或者3上。对于一个大小为31的堆，给出第k大的元素可能出现的位置和不可能出现的位置，其中k=2、3、4（设元素值不重复）。

2.4.8 回答上一道练习中第k小元素的可能和不可能的位置。

2.4.9 给出A B C D E五个元素可能构造出来的所有堆，然后给出A A A B B这五个元素可能构造出来的所有堆。

2.4.10 假设我们不想浪费堆有序的数组pq[]中的那个位置，将最大的元素放在pq[0]，它的子结点放在pq[1]和pq[2]，以此类推。pq[k]的父结点和子结点在哪里？

329 2.4.11 如果你的应用中有大量的插入元素的操作，但只有若干删除最大元素操作，哪种优先队列的实现方法更有效：堆、无序数组、有序数组？

2.4.12 如果你的应用场景中大量的找出最大元素的操作，但插入元素和删除最大元素操作相对较少，哪种优先队列的实现方法更有效：堆、无序数组、有序数组？

2.4.13 想办法在sink()中避免检查j < N。

2.4.14 对于没有重复元素的大小为N的堆，一次删除最大元素的操作中最少要交换几个元素？构造一个能够达到这个交换次数的大小为15的堆。连续两次删除最大元素呢？三次呢？

2.4.15 设计一个程序，在线性时间内检测数组pq[]是否是一个面向最小元素的堆。

2.4.16 对于N=32，构造数组使得堆排序使用的比较次数最多以及最少。

2.4.17 证明：构造大小为k的面向最小元素的优先队列，然后进行N–k次替换最小元素操作（删除最小元素后再插入元素）后，N个元素中的前k大元素均会留在优先队列中。

2.4.18 在MaxPQ中，如果一个用例使用insert()插入了一个比队列中的所有元素都大的新元素，随后立即调用delMax()。假设没有重复元素，此时的堆和进行这些操作之前的堆完全相同吗？进行两次insert()（第一次插入一个比队列所有元素都大的元素，第二次插入一个更大的元素）操作接两次delMax()操作呢？

2.4.19 实现MaxPQ的一个构造函数，接受一个数组作为参数。使用正文2.4.5.1节中所述的自底向上的方法构造堆。

330 2.4.20 证明：基于下沉的堆构造方法使用的比较次数小于$2N$，交换次数小于N。

提高题

2.4.21 *基础数据结构*。说明如何使用优先队列实现第1章中的栈、队列和随机队列这几种数据结构。

2.4.22 *调整数组大小*。在MaxPQ中加入调整数组大小的代码，并和命题Q一样证明对于一般性长度为N的队列其数组访问的上限。

2.4.23 *Multiway的堆*。只考虑比较的成本且假设找到t个元素中的最大者需要t次比较，在堆排序中使用t向堆的情况下找出使比较次数$N\lg N$的系数最小的t值。首先，假设使用的是一个简单通用的sink()方法；其次，假设Floyd方法在内循环中每轮可以节省一次比较。

2.4.24 *使用链接的优先队列*。用堆有序的二叉树实现一个优先队列，但使用链表结构代替数组。每个结点都需要三个链接：两个向下，一个向上。你的实现即使在无法预知队列大小的情况下也能保证优先队列的基本操作所需的时间为对数级别。

2.4.25 计算数论。编写程序 CubeSum.java，在不使用额外空间的条件下，按大小顺序打印所有 a^3+b^3 的结果，其中 a 和 b 为 0 至 N 之间的整数。也就是说，不要全部计算 N^2 个和然后排序，而是创建一个最小优先队列，初始状态为 $(0^3, 0, 0),(1^3, 1, 0),(2^3, 2, 0),\cdots,(N^3, N, 0)$。这样只要优先队列非空，删除并打印最小的元素 (i^3+j^3, i, j)。然后如果 $j<N$，插入元素 $(i^3+(j+1)^3, i, j+1)$。用这段程序找出 0 到 10^6 之间所有满足 $a^3+b^3=c^3+d^3$ 的不同整数 a,b,c,d。

2.4.26 无需交换的堆。因为 sink() 和 swim() 中都用到了初级函数 exch()，所以所有元素都被多加载并存储了一次。回避这种低效方式，用插入排序给出新的实现（请见练习 2.1.25）。

2.4.27 找出最小元素。在 MaxPQ 中加入一个 min() 方法。你的实现所需的时间和空间都应该是常数。

2.4.28 选择过滤。编写一个 TopM 的用例，从标准输入读入坐标 (x, y, z)，从命令行得到值 M，然后打印出距离原点的欧几里德距离最小的 M 个点。在 $N=10^8$ 且 $M=10^4$ 时，预计程序的运行时间。 331

2.4.29 同时面向最大和最小元素的优先队列。设计一个数据类型，支持如下操作：插入元素、删除最大元素、删除最小元素（所需时间均为对数级别），以及找到最大元素、找到最小元素（所需时间均为常数级别）。提示：用两个堆。

2.4.30 动态中位数查找。设计一个数据类型，支持在对数时间内插入元素，常数时间内找到中位数并在对数时间内删除中位数。提示：用一个面向最大元素的堆再用一个面向最小元素的堆。

2.4.31 快速插入。用基于比较的方式实现 MinPQ 的 API，使得插入元素需要 ~ loglog$N$ 次比较，删除最小元素需要 ~2log$N$ 次比较。提示：在 swim() 方法中用二分查找来寻找祖先结点。

2.4.32 下界。请证明，不存在一个基于比较的对 MinPQ 的 API 的实现能够使得插入元素和删除最小元素的操作都保证只使用 ~Nloglog$N$ 次比较。

2.4.33 索引优先队列的实现。按照 2.4.4.6 节的描述修改算法 2.6 来实现索引优先队列 API 中的基本操作：使用 pq[] 保存索引，添加一个数组 keys[] 来保存元素，再添加一个数组 qp[] 来保存 pq[] 的逆序——qp[i] 的值是 i 在 pq[] 中的位置（即索引 j，pq[j]=i）。修改算法 2.6 的代码来维护这些数据结构。若 i 不在队列之中，则总是令 qp[i] = −1 并添加一个方法 contains() 来检测这种情况。你需要修改辅助函数 exch() 和 less()，但不需要修改 sink() 和 swim()。 332

部分答案：

```
public class IndexMinPQ<Key extends Comparable<Key>>
{
   private int N;           // PQ中的元素数量
   private int[] pq;        // 索引二叉堆，由1开始
   private int[] qp;        // 逆序：qp[pq[i]] = pq[qp[i]] = i
   private Key[] keys;      // 有优先级之分的元素
   public IndexMinPQ(int maxN)
   {
      keys = (Key[]) new Comparable[maxN + 1];
      pq   = new int[maxN + 1];
      qp   = new int[maxN + 1];
      for (int i = 0; i <= maxN; i++) qp[i] = -1;
   }

   public boolean isEmpty()
   {  return N == 0;  }

   public boolean contains(int k)
   {  return qp[k] != -1;  }
```

```
   public void insert(int k, Key key)
   {
      N++;
      qp[k] = N;
      pq[N] = k;
      keys[k] = key;
      swim(N);
   }

   public Key min()
   {  return keys[pq[1]];  }

   public int delMin()
   {
      int indexOfMin = pq[1];
      exch(1, N--);
      sink(1);
      keys[pq[N+1]] = null;
      qp[pq[N+1]] = -1;
      return indexOfMin;
   }
}
```

333

2.4.34 索引优先队列的实现（附加操作）。向练习 2.4.33 的实现中添加 minIndex()、change() 和 delete() 方法。

解答：

```
public int minIndex()
{  return pq[1];  }

public void change(int k, Key Key)
{
   keys[k] = key;
   swim(qp[k]);
   sink(qp[k]);
}

public void delete(int k)
{
   exch(k, N--);
   swim(qp[k]);
   sink(qp[k]);
   keys[pq[N+1]] = null;
   qp[pq[N+1]] = -1;
}
```

2.4.35 离散概率分布的取样。编写一个 Sample 类，其构造函数接受一个 double 类型的数组 p[] 作为参数并支持以下操作：random()——返回任意索引 i 及其概率 p[i]/T（T 是 p[] 中所有元素之和）；change(i, v)——将 p[i] 的值修改为 v。提示：使用完全二叉树，每个结点对应一个权重 p[i]。在每个结点记录其下子树的权重之和。为了产生一个随机的索引，取 0 到 T 之间的一个随机数并根据各个结点的权重之和来判断沿着哪条子树搜索下去。在更新 p[i] 时，同时更新从根结点到 i 的路径上的所有结点。不要像堆的实现那样显式使用指针。

334

实验题

2.4.36 性能测试 I。编写一个性能测试用例，用插入元素操作填满一个优先队列，然后用删除最大元素操作删去一半元素，再用插入元素操作填满优先队列，再用删除最大元素操作删去所有元素。用一列随机的长短不同的元素多次重复以上过程，测量每次运行的用时，打印平均用时或是将其绘制成图表。

2.4.37 性能测试 II。编写一个性能测试用例，用插入元素操作填满一个优先队列，然后在一秒钟之内尽可能多地连续反复调用删除最大元素和插入元素的操作。用一列随机的长短不同的元素多次重复以上过程，将程序能够完成的删除最大元素操作的平均次数打印出来或是绘成图表。

2.4.38 练习测试。编写一个练习用例，用算法 2.6 中实现的优先队列的接口方法处理实际应用中可能出现的高难度或是极端情况。例如，元素已经有序、元素全部逆序、元素全部相同或是所有元素只有两个值。

2.4.39 构造函数的代价。对于 $N=10^3$、10^6 和 10^9，根据经验判断堆排序时构造堆占总耗时的比例。

2.4.40 Floyd 方法。根据正文中 Floyd 的先沉后浮思想实现堆排序。对于 $N=10^3$、10^6 和 10^9 大小的随机不重复数组，记录你的程序所使用的比较次数和标准实现所使用的比较次数。

2.4.41 Multiway 堆。根据正文中的描述实现基于完全堆有序的三叉树和四叉树的堆排序。对于 $N=10^3$、10^6 和 10^9 大小的随机不重复数组，记录你的程序所使用的比较次数和标准实现所使用的比较次数。

2.4.42 堆的前序表示。用前序法而非级别表示一棵堆有序的树，并基于此实现堆排序。对于 $N=10^3$、10^6 和 10^9 大小的随机不重复数组，记录你的程序所使用的比较次数和标准实现所使用的比较次数。

335

2.5 应用

排序算法和优先队列在许多场景中有着广泛的应用。本节中我们将简要地浏览一遍这些应用，研究如何能让我们已经学习过的高效算法在这些应用中大展身手，然后讨论一下应该如何使用我们的排序和优先队列的代码。

排序如此有用的一个主要原因是，在一个有序的数组中查找一个元素要比在一个无序的数组中查找简单得多。人们用了一个多世纪发现在一本按姓氏排序的电话黄页中查找某个人的电话号码最容易。现在，数字音乐作家们将歌曲文件按照作家名或是歌曲名排序，搜索引擎按照搜索结果的重要性的高低显示结果，电子表格按照某一列的排序结果显示所有栏，矩阵处理工具将一个对称矩阵的真实特征值按照降序排列，等等。只要队列是有序的，很多其他任务也更容易完成，比如在本书最后的有序索引中查找某项，或是从一列长长的邮件列表或者投票人列表或者网站列表中删去重复项，或是在统计学计算中剔除异常值、查找中位数或者计算比例。

在许多看似无关的领域中，排序其实仍然是一个重要的子问题。数据压缩、计算机图形学、计算生物学、供应链管理、组合优化、社会选择和投票等，不一而足。我们在本章中学习的算法也在开发本书其他章节的强大算法的过程中起到了关键作用。

通用排序算法是最重要的，因此我们首先会考虑一些在构建适用于多种情况的排序算法时需要注意的实际问题。虽然部分话题只适用于 Java，但每个问题都仍然是所有系统需要解决的。

我们的主要目的是为了说明，尽管我们所学习的各种算法的思想相对简单，但它们的适用领域仍然广泛。经过验证的各种排序算法的应用列表很长，我们在这里只会涉及其中的一小部分，一些是科学领域的，一些是算法领域的，还有一些是商业领域的。在练习中你们还能找到更多例子，本书的网站上还有更多。另外，为了更好的说明问题，后续章节还会不时地引用本章的内容！

336

2.5.1 将各种数据排序

我们的实现的排序对象是由实现了 Comparable 接口的对象组成的数组。Java 的约定使得我们能够利用 Java 的回调机制将任意实现了 Comparable 接口的数据类型排序。如 2.1 节所述，实现 Comparable 接口只需要定义一个 compareTo() 函数并在其中定义该数据类型中的大小关系。我们的代码直接能够将 String、Integer、Double 和一些其他例如 File 和 URL 类型的数组排序，因为它们都实现了 Comparable 接口。同一段代码能够适应所有这些类型的数据是非常方便的，但一般的应用程序中需要排序的数据类型都是应用程序自己定义的。相应，在自定义的数据类型中实现一个 compareTo() 方法也是很常见的，这样就实现了 Comparable 接口，也就使得这种数据类型可以被排序了（也可以用其构造优先队列）。

2.5.1.1 交易事务

排序算法的一种典型应用就是商业数据处理。例如，设想一家互联网商业公司为每笔交易记录都保存了所有的相关信息，包括客户名、日期、金额等。如今，一家成功的商业公司需要能够处理数百万的这种交易数据。如我们在练习 2.1.21 中看到的，一种合适的方法是将交易记录按金额大小排序，我们在类的定义中实现一个恰当的 compareTo() 方法就可以做到这一点。这样我们在处理 Transaction 类型的数组 a[] 时就可以先将其排序，比如这样 Quick.sort(a)。我们的排序算法对 Transaction 类型一无所知，但 Java 的 Comparable 接口使我们可以为该类型定义大小关系，

这样我们的任意排序算法都能够用于 Transaction 对象了。或者我们也可以令 Transaction 对象按照日期排序（如下面的代码所示），将 compareTo() 方法实现为比较 Date 字段。因为 Date 对象本身也实现了 Comparable 接口，我们可以直接调用它的 compareTo() 方法而不用自己实现了。将这种类型按照用户名排序也是合理的。使算法的用例能够灵活地用不同的字段排序则是我们在稍后将要面对的另一项有趣的挑战。

```
public int compareTo(Transaction that)
{  return this.when.compareTo(that.when);  }
```

将交易记录按照日期排序的compareTo()方法

337

2.5.1.2 指针排序

我们使用的方法在经典教材中被称为*指针排序*，因为我们只处理元素的引用而不移动数据本身。在其他编程语言例如 C 和 C++ 之中，程序员需要明确地指出操作的是数据还是指向数据的指针，而在 Java 中，指针操作是隐式的。除了原始数字类型之外，我们操作的总是数据的引用（指针），而非数据本身。指针排序增加了一层间接性，因为数组保存的是待排序的对象的引用，而非对象本身。我们会简要讨论一些相关的问题。对于多个引用数组，我们可以将同一组数据的不同部分按照多种方式排序（可能会用到下面提到的多键）。

2.5.1.3 不可变的键

如果在排序后用例还能够修改键值，那么数组就很可能不再是有序的了。类似，优先队列在用例能够修改键值的情况下也不太可能正常工作。在 Java 中，可以用不可变的数据类型作为键来避免这个问题。大多数你可能用作键的数据类型，例如 String、Integer、Double 和 File 都是不可变的。

2.5.1.4 廉价的交换

使用引用的另一个好处是我们不必移动整个元素。对于元素大而键小的数组来说这带来的节约是巨大的，因为比较只需要访问元素的一小部分，而排序过程中大部分元素都不会被访问到。对于几乎任意大小的元素，使用引用使得在一般情况下交换的成本和比较的成本几乎相同（代价是需要额外的空间存储这些引用）。如果键值很长，那么交换的成本甚至会低于比较的成本。研究将数字排序的算法性能的一种方法就是观察其所需的比较和交换总数，因为这里隐式地假设了比较和交换的成本是相同的。由此得出的结论则适用于 Java 中的许多应用，因为我们都是在将引用排序。

2.5.1.5 多种排序方法

在很多应用中我们都希望根据情况将一组对象按照不同的方式排序。Java 的 Comparator 接口允许我们在一个类之中实现多种排序方法。它只有一个 compare() 方法来比较两个对象。如果一种数据类型实现了这个接口，我们可以像 2.5.1.6 节中的例子那样将另一个实现了 Comparator 接口的对象传递给 sort() 方法（sort() 再将其传递给 less()）。Comparator 接口允许我们为任意数据类型定义任意多种排序方法。用 Comparator 接口来代替 Comparable 接口能够更好地将数据类型的定义和两个该类型的对象应该如何比较的定义区分开来。事实上，比较两个对象的确可以有多种标准，Comparator 接口使得我们能够在其中进行选择。例如，想在忽略大小写的情况下将字符串数组 a[] 排序，可以使用 Java 的 String 类型中定义的 CASE_INSENSITVE_

338

ORDER 比较器并调用 Insertion.sort(a, String.CASE_INSENSITIVE_ORDER)。你也知道，精确定义的字符串排序规则十分复杂，而各种自然语言又差异很大，所以 Java 的 String 类型含有很多比较器。

2.5.1.6 多键数组

一般在应用程序中，一个元素的多种属性都可能被用作排序的键。在交易的例子中，有时可能需要将交易按照客户排序（例如，找出每个客户进行的所有交易）；有时又可能需要按照金额排序（例如，需要找出交易金额较高的交易）；有时还可能用另一个属性来排序。要实现这种灵活性，Comparator 接口正合适。我们可以定义多种比较器，如 2.5.1.7 节展示的 Transaction 类的另一种实现那样。在这样定义之后，要将 Transaction 对象的数组按照时间排序可以调用：

```
Insertion.sort(a, new Transaction.WhenOrder())
```

或者这样来按照金额排序：

```
Insertion.sort(a, new Transaction.HowMuchOrder())
```

sort() 方法在每次比较中都会回调 Transaction 类中用例指定的 compare() 方法。为了避免每次排序都创建一个新的 Comparator 对象，我们使用了 public final 来定义这些比较器（代码如下，就像 Java 定义的 CASE_INSENSITIVE_ORDER 一样）。

```
public static void sort(Object[] a, Comparator c)
{
   int N = a.length;
   for (int i = 1; i < N; i++)
      for (int j = i; j > 0 && less(c, a[j], a[j-1]); j--)
         exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w)
{  return c.compare(v, w) < 0;  }

private static void exch(Object[] a, int i, int j)
{  Object t = a[i]; a[i] = a[j]; a[j] = t; }
```

339

使用了Comparator的插入排序

2.5.1.7 使用比较器实现优先队列

比较器的灵活性也可以用在优先队列上。我们可以按照以下步骤来扩展算法 2.6 的标准实现来支持比较器：

- 导入 java.util.Comparator；
- 为 MaxPQ 添加一个实例变量 comparator 以及一个构造函数，该构造函数接受一个比较器作为参数并用它将 comparator 初始化；
- 在 less() 中检查 comparator 属性是否为 null（如果不是的话就用它进行比较）。

实现代码如下：

```
import java.util.Comparator;

public class Transaction
{
   ...
   private final String who;
   private final Date when;
   private final double amount;
   ...
   public static class WhoOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {  return v.who.compareTo(w.who);  }
   }

   public static class WhenOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {  return v.when.compareTo(w.when);  }
   }

   public static class HowMuchOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {
         if (v.amount < w.amount) return -1;
         if (v.amount > w.amount) return +1;
         return 0;
      }
   }
}
```

使用了Comparator的插入排序

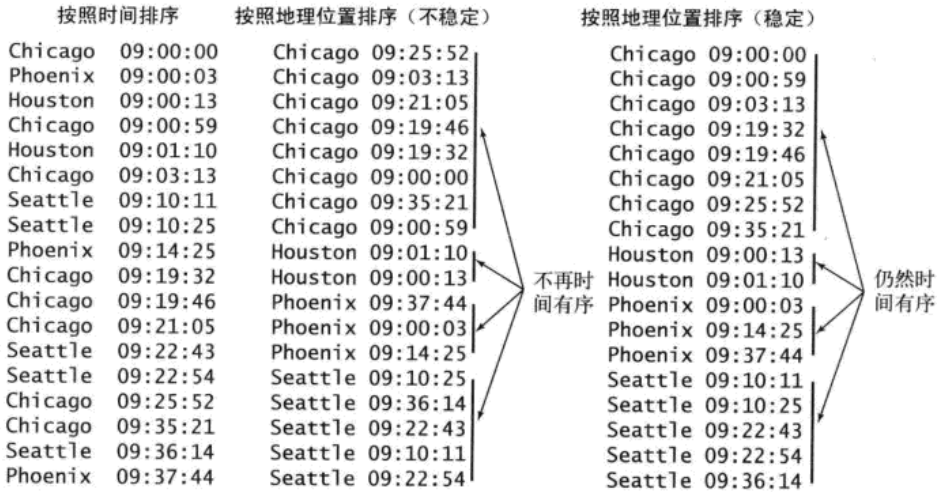
例如，修改后可以使用 Transaction 的多种字段构造不同的优先队列，分别按照时间、地点、账号排序。如果你在 MinPQ 中去掉了 Key extends Comparable<Key> 这句话，甚至可以支持尚未定义过比较方法的键。

340

2.5.1.8 稳定性

如果一个排序算法能够保留数组中重复元素的相对位置则可以被称为是稳定的。这个性质在许多情况下很重要。例如，考虑一个需要处理大量含有地理位置和时间戳的事件的互联网商业应用程序。首先，我们在事件发生时将它们挨个存储在一个数组中，这样在数组中它们已经是按照时间顺序排好了的。现在假设在进一步处理前将按照地理位置切分。一种简单的方法是将数组按照位置排序。如果排序算法不是稳定的，排序后的每个城市的交易可能不会再是按照时间顺序排列的了。很多情况下，不熟悉排序稳定性的程序员在第一次遇见这种情形时会惊讶于不稳定的排序算法似乎把数据弄得一团糟。在本章中，我们学习过的一部分算法是稳定的（插入排序和归并排序），但很多不是（选择排序、希尔排序、快速排序和堆排序）。有很多办法能够将任意排序算法变成稳定的（请见练习 2.5.18），但一般只有在稳定性是必要的情况下稳定的排序算法才有优势。人们很容易觉得算法具有稳定性是理所当然的，但事实上没有任何实际应用中常见的方法不是用了大量额外的时间和空间才做到了这一点（研究人员开发了这样的算法，但应用程序员发现它们太复杂了，无法使用）。

从另一个键上排序的稳定性如图 2.5.1 所示。



341

图 2.5.1 从另一个键上排序的稳定性

2.5.2 我应该使用哪种排序算法

在本章中我们学习了许多种排序算法，这个问题就变得很自然了。排序算法的好坏很大程度上取决于它的应用场景和具体实现，但我们也学习了一些通用的算法，它们能在很多情况下达到和最佳算法接近的性能。

表 2.5.1 总结了在本章中我们学习过的排序算法的各种重要性质。除了希尔排序（它的复杂度只是一个近似）、插入排序（它的复杂度取决于输入元素的排列情况）和快速排序的两个版本（它们的复杂度和概率有关，取决于输入元素的分布情况）之外，将这些运行时间的增长数量级乘以适当的常数就能够大致估计出其运行时间。这里的常数有时和算法有关（比如堆排序的比较次数是归并排序的两倍，且两者访问数组的次数都比快速排序多得多），但主要取决于算法的实现、Java 编译器以及你的计算机，这些因素决定了需要执行的机器指令的数量以及每条指令所需的执行时间。最重要的是，因为这些都是常数，你能通过较小的 N 得到的实验数据和我们的标准双倍测试来推测较大的 N 所需的运行时间。

表 2.5.1 各种排序算法的性能特点

算 法	是否稳定	是否为原地排序	将 N 个元素排序的复杂度		备 注
			时间复杂度	空间复杂度	
选择排序	否	是	N^2	1	
插入排序	是	是	介于 N 和 N^2 之间	1	取决于输入元素的排列情况
希尔排序	否	是	$N\log N$? $N^{6/5}$?	1	
快速排序	否	是	$N\log N$	$\lg N$	运行效率由概率提供保证
三向快速排序	否	是	介于 N 和 $N\log N$ 之间	$\lg N$	运行效率由概率保证，同时也取决于输入元素的分布情况
归并排序	是	否	$N\log N$	N	
堆排序	否	是	$N\log N$	1	

342

性质 T。快速排序是最快的通用排序算法。

例证。自从数十年前快速排序发明以来，它在无数计算机系统中的无数实现已经证明了这一点。总的来说，快速排序之所以最快是因为它的内循环中的指令很少（而且它还能利用缓存，因为它总是顺序地访问数据），所以它的运行时间的增长数量级为 $\sim cN\lg N$，而这里的 c 比其他线性对数级别的排序算法的相应常数都要小。在使用三向切分之后，快速排序对于实际应用中可能出现的某些分布的输入变成线性级别的了，而其他的排序算法则仍然需要线性对数时间。

因此，在大多数实际情况中，快速排序是最佳选择。当然，面对多种排序方法和各式计算机及系统，这么一句干巴巴的话很难让人信服。例如，我们已经见过一个明显的例外：如果稳定性很重要而空间又不是问题，归并排序可能是最好的。我们会在第 5 章中见到更多例外。有了 SortCompare 这样的工具，再加上一点时间和努力，你能够更仔细地比较这些算法的性能并实现我们讨论过的各种改进方案，详见本节最后的若干练习。也许证明性质 T 的最好方式正如这里所说，在运行时间至关重要的任何排序应用中认真地考虑使用快速排序。

2.5.2.1 将原始类型数据排序

一些性能优先的应用的重点可能是将数字排序，因此更合理的做法是跳过引用直接将原始数据类型的数据排序。例如，想想将一个 double 类型的数组和一个 Double 类型的数组排序的差别。对于前者我们可以直接交换这些数并将数组排序；而对于后者，我们交换的是存储了这些数字的 Double 对象的引用。如果我们只是在将一大组数排序的话，跳过引用可以为我们节省存储所有引用所需的空间和通过引用来访问数字的成本，更不用说那些调用 compareTo() 和 less() 方法的开销了。把 Comparable 接口替换为原始数据类型名，重定义 less() 方法或者干脆将调用 less() 的地方替换为 a[i] < a[j] 这样的代码，我们就能得到可以将原始数据类型的数据更快地排序的各种算法（请见练习 2.1.26）。

2.5.2.2 Java 系统库的排序算法

为了演示表 2.5.1 所示的数据，这里我们考虑 Java 系统库中的主要排序方法 java.util.Arrays.sort()。根据不同的参数类型，它实际上代表了一系列排序方法：

343

- 每种原始数据类型都有一个不同的排序方法；
- 一个适用于所有实现了 Comparable 接口的数据类型的排序方法；
- 一个适用于实现了比较器 Comparator 的数据类型的排序方法。

Java 的系统程序员选择对原始数据类型使用（三向切分的）快速排序，对引用类型使用归并排序。这些选择实际上也暗示着用速度和空间（对于原始数据类型）来换取稳定性（对于引用类型），如刚才讨论的那样。

我们讨论过的这些算法和思想是包括 Java 的许多现代系统的核心组成部分。当为实际应用开发 Java 程序时，你会发现 Java 的 Arrays.sort() 实现（可能再加上你自己实现的 compareTo() 或者 compare()）已经基本够用了，因为它使用的三向快速排序和归并排序都是经典。

在本书中我们一般都会使用我们自己的 Quick.sort() 或者 Merge.sort()（在稳定性比空间更重要时）。你也可以使用 Arrays.sort()，或者在特殊的情况下使用其他排序算法。

2.5.3 问题的归约

使用排序算法来解决其他问题的思想是算法设计领域的基本技巧——*归约*的一个例子。因为归

约十分重要，我们会在第6章详细讨论它，同时研究几个具体实例。*归约*指的是为解决某个问题而发明的算法正好可以用来解决另一种问题。应用程序员对于归约的概念已经很熟悉了（无论是否明确地知道这一点）——每次你在使用解决问题B的方法来解决问题A时，你都是在将A归约为B。实际上，实现算法的一个目标就是使算法的适用性尽可能广泛，使得问题的归约更简单。作为例子，我们先看看几个简单的排序问题。很多这种问题都以算法测验的形式出现，而解决它们的第一想法往往是平方级别的暴力破解。但很多情况下如果先将数据排序，那么解决剩下的问题就只需要线性级别的时间了，这样归约后的运行时间的增长数量级就由平方级别降低到了线性对数级别。

2.5.3.1 找出重复元素：

在一个Comparable对象的数组中是否存在重复元素？有多少重复元素？哪个值出现得最频繁？对于小数组，用平方级别的算法将所有元素互相比较一遍就足以解答这些问题。但这么做对于
344 大数组行不通。但有了排序，你就能在线性对数的时间内回答这些问题：首先将数组排序，然后遍历有序的数组，记录连续出现的重复元素即可。例如，下面就是一段统计数组中不重复的元素个数的代码。只要稍稍修改这段代码你就能回答上面的问题，还可以打印所有不同元素的值、所有重复元素的值，等等，即使数组很大也无妨。

2.5.3.2 排名

```
Quick.sort(a);
int count = 1; // 假设a.length > 0.
for (int i = 1; i < a.length; i++)
   if (a[i].compareTo(a[i-1]) != 0)
      count++;
```

统计a[]中不重复元素的个数

一组*排列*（或是排名）就是一组N个整数的数组，其中0到N–1的每个数都只出现一次。两个排列之间的Kendall tau *距离*就是在两组数列中顺序不同的数对的数目。例如，0 3 1 6 2 5 4和1 0 3 6 4 2 5之间的Kendall tau距离是4，因为0-1、3-1、2-4、5-4这4对数字在两组排列中的相对顺序不同，但其他数字的相对顺序都是相同的。这种统计方法的应用十分广泛。在社会学中它被用于研究社会选择和投票理论，在分子生物学中被用于使用基因表达图谱比较基因，在网络中被用于搜索引擎结果的排名，等等。某个排列和标准排列（即每个元素都在正确位置上的排列）的Kendall tau距离就是其中逆序数对的数量。根据插入排序设计一个平方级别的算法来计算它并不困难（请回想2.1节中的命题C）。高效地计算Kendall tau距离可以留给已经熟悉那些经典的排序算法的程序员（或者学生）作为一个有趣的练习（请见练习2.5.19）。

2.5.3.3 优先队列

在2.4节中我们已经见过两个被归约为优先队列操作的问题的例子。一个是2.4.2.1节中的TopM，它能够找到输入流中M个最大的元素；另一个是2.4.4.7节中的Multiway，它能够将M个输入流归并为一个有序的输出流。这两个问题都可以轻易用长度为M的优先队列解决。

2.5.3.4 中位数与顺序统计

一个和排序有关但又不需要完全排序的重要应用就是找出一组元素的*中位数*（中间值，它不大于一半的元素又不小于另一半元素）。查找中位数在统计学计算和许多数据处理的应用程序中都很常见。它是一种特殊的*选择*：找到一组数中的第k小的元素（如下页代码所示）。“选择”在处理实验数据和其他数据中应用广泛，使用中位数和其他*顺序统计*来切分一个数组也很常见。一般，我
345 们只需要处理一个很大的数组中的一小部分，在这种情况下，一个程序可以选择，比如将前10%的元素完全排序即可。2.4节中我们的TopM用优先队列为无界限输入解决了这个问题。除了TopM，另一种选择是直接将数组中的元素排序。在调用Quick.sort(a)之后，数组中的k个最小的元素就是数组的前k个元素，其中k小于数组长度。但这种方法需要调用排序，所以运行时间的增长数

量级是线性对数的。

还有更好的办法吗？当 k 很小或者很大时找出数组中的 k 个最小值都很简单，但当 k 和数组大小成一定比例时这个任务就变得比较困难了，比如找到中位数（$k=N/2$）。让人惊讶的是其实上面的 select() 方法能够在线性时间内解决这个问题（这个实现需要在用例中进行类型转换；去掉这个限制的代码请见本书的网站）。为了完成这个任务，select() 用两个变量 hi 和 lo 来限制含有要选择的 k 元素的子数组，并用快速排序的切分法来缩小子数组的范围。请回想 partition() 方法，它会将数组的 a[lo] 至 a[hi] 重新排列并返回一个整数 j 使得 a[lo..j-1] 小于等于 a[j] 且 a[j+1..hi] 大于等于 a[j]。那么，如果 k = j，问题就解决了。如果 k < j，我们就需要切分左子数组（令 hi = j-1）；如果 k > j，我们则需要切分右子数组（令 lo = j+1）。这个循环保证了数组中 lo 左边的元素都小于等于 a[lo..hi]，而 hi 右边的元素都大于等于 a[lo..hi]。我们不断地切分直到子数组中只含有第 k 个元素，此时 a[k] 含有最小的（k+1）个元素，a[0] 到 a[k-1] 都小于等于 a[k]，而 a[k+1] 及其后的元素都大于等于 a[k]。至于为何这个算法是线性级别的，是因为 346
假设每次都正好将数组二分，那么比较的总次数为（$N+N/2+N/4+N/8\cdots$），直到找到第 k 的元素，这个和显然小于 $2N$。和快速排序一样，这里也需要一点数学知识来得到比较的上界，它比快速排序略高。这个算法和快速排序的另一个共同点是这段分析依赖于使用随机的切分元素，因此它的性能保证也来自于概率。

```
public static Comparable
select(Comparable[] a, int k)
{
   StdRandom.shuffle(a);
   int lo = 0, hi = a.length - 1;
   while (hi > lo)
   {
      int j = partition(a, lo, hi);
      if      (j == k)  return a[k];
      else if (j > k)   hi = j - 1;
      else if (j < k)   lo = j + 1;
   }
   return a[k];
}
```

找到一组数中的第k小元素

用快速排序的切分来查找中位数的可视轨迹如图 2.5.2 所示。

命题 U。平均来说，基于切分的选择算法的运行时间是线性级别的。

证明。该命题的分析和快速排序的命题 K 的证明类似，但要复杂得多。结论就是算法的平均比较次数为 $\sim 2N+2k\ln(N/k)+2(N-k)\ln(N/(N-k))$，这对于所有合法的 k 值都是线性的。例如，这个公式说明找到中位数（$k=N/2$）平均需要 $\sim(2+2\ln 2)N$ 次比较。注意，最坏的情况下算法的运行时间仍然是平方级别的，但与快速排序一样，将数组乱序化可以有效防止这种情况出现。

设计一个能够保证在最坏情况下也只需要线性比较次数的算法是计算复杂性领域的一个经典问题，但到目前为止仍然没有一个能够实用的算法。 347

2.5.4 排序应用一览

排序的直接应用极为普遍和广泛，无法一一列举。你可以将歌曲按照曲名或是歌手排序，将邮件按照时间或是发件人排序（或者来电按照时间或来电者排序），将照片按照日期排序。大学会将学生的账户按照姓名或是 ID 排序。信用卡公司会将上百万甚至上亿的交易按照日期或是金额排序。科学家会将实验数据按照时间或其他标准排序来精确地模拟现实世界，从粒子或者天体的运动，到物质的结构，到社会中的人际关系。实际上，很难找到和排序无关的任何计算性应用！为了更好地

说明这一点，我们在这一小节中举几个比应用归约更加复杂的例子，其中几个我们会在本书的其他章节更加详细地研究。

2.5.4.1 商业计算

世界已经被信息的海洋所淹没。政府组织、金融机构和商业公司都依赖排序来管理大量的信息。无论这些信息是按照名字或者数字排序的账号、按照日期或者金额排序的交易、按照邮编或者地址排序的邮件、按照名称或者日期排序的文件等，处理这些数据必然需要排序算法。一般这些信息都会存储在大型的数据库里，能够按照多个键排序以提高搜索效率。一个普遍使用的有效方法是先收集新的信息并添加到数据库，将其按感兴趣的键排序，然后将每个键的排序结果归并到已存在的数据库中。从计算机发明的早期开始，我们学习过的这些方法就已经被用来构建庞大的基础数据，处理它们的方法则是所有这些商业活动的基石。今天，我们能够按部就班地处理上百万甚至上亿大小的数组——没有线性对数级别的排序算法也就没法将它们排序，进一步处理这些数据也会极端困难，甚至是不可能的。

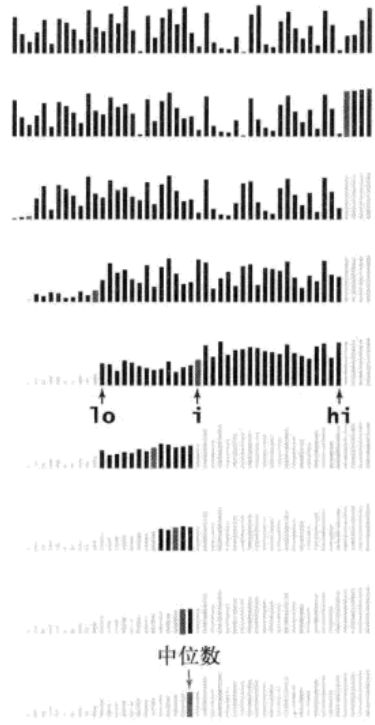


图 2.5.2 用切分找出中位数（另见彩插）

2.5.4.2 信息搜索

有序的信息确保我们可以用经典的*二分查找法*（见第 1 章）来进行高效的搜索。你会看到许多其他种类的查询也可以用相同的方式完成。有多少元素小于给定的元素？有哪些在给定的范围之内？在第 3 章中我们不但会解答这些问题，还会具体学习排序算法和二分查找的各种扩展，使得我们能够用删除和插入的混合操作解答这些问题，并保证所有操作的对数级别的性能。

348

2.5.4.3 运筹学

*运筹学*指的是研究数学模型并将其应用于问题解决和决策的领域。在本书中我们会看到若干运筹学和算法研究的关系的例子。这里我们先来看排序算法在运筹学的经典问题——调度中的应用。假设我们需要完成 N 个任务，第 j 个任务需要耗时 t_j 秒。我们需要在完成所有任务的同时尽量确保客户满意，将每个任务的平均完成时间最小化。按照*最短优先*的原则，只要我们将任务按照处理时间升序排列就可以达到目标。因此我们可以将任务按照耗时排序，或是将它们插入到一个最小优先队列中。如果加上其他各种限制，我们可以得到不同的调度问题，这在工业界的应用中很常见，也被很好地研究过。另一个例子是*负载均衡*问题。假设我们有 M 个相同的处理器以及 N 个任务，我们的目标是用尽可能短的时间在这些处理器上完成所有的任务。这个问题是 NP- 困难的（请见第 6 章），因此我们实际上不可能算出一种最优的方案。但一种较优调度方法是*最大优先*。我们将任务按照耗时降序排列，将每个任务依次分配给当前可用的处理器。要实现这种算法，我们先要逆序排列这些任务，然后为 M 个处理器维护一个优先队列，每个元素的优先级就是对应的处理器上运行的任务的耗时之和。每一步中，我们都删去优先级最低的那个处理器，将下一个任务分配给这个处理器，然后再将它重新插入优先队列。

2.5.4.4 事件驱动模拟

很多科学上的应用都涉及模拟，用大量计算来将现实世界的某个方面建模以期能够更好地理解它。在计算机发明之前，科学家们除了构建数学模型之外别无选择，而现在计算机模型很好地补充

了这些数学模型。逼真地模拟现实世界是很有挑战的，而使用正确的算法使得我们能够在有限的时间内完成这些模拟，而不是无奈地接受不精确的实验结果或是无尽地等待计算的完成。我们会在第6章中展示能够说明这一点的一个具体例子。

2.5.4.5 数值计算

在科学计算中，精确度非常重要（我们距离真正的答案有多远），特别是当我们在计算机中使用的只是真正的实数的近似值——浮点数来进行上百万次计算的时候。一些数值计算算法使用优先队列和排序来控制计算中的精确度。例如，在求曲线下区域的面积时，数值积分的一个方法就是使 349
用一个优先队列存储一组小间隔中每段的近似精确度。积分的过程就是删去精确度最低的间隔并将其分为两半（这样两半都能变得更加精确），然后将两半都重新加入优先队列。如此这般，直到达到预期的精确程度。

2.5.4.6 组合搜索

人工智能领域一个解决“疑难杂症”的经典范式就是定义一组状态、由一组状态演化到另一组状态可能的步骤以及每个步骤的优先级，然后定义一个起始状态和目标状态（也就是问题的解决办法）。著名的 A* 算法的解决办法就是将起始状态放入优先队列中，然后重复下面的方法直到到达目的地：删去优先级最高的状态，然后将能够从该状态在一步之内达到的所有状态全部加入优先队列（除了刚刚删去的那个状态之外）。和事件驱动模拟一样，这个过程简直就是为优先队列量身定做的。它将问题的解决转化为了定义一个适当的优先级函数问题。例子请见练习 2.5.32。

除了这些直接应用之外（我们只说了很小的一部分而已），排序和优先队列在算法设计领域也是很重要的抽象概念，因此本书会经常用到它们。下面我们举了一些本书后续内容中的应用作为例子，它们都依赖于本章中的排序算法和优先队列数据类型的高效实现。

- **Prim 算法和 Dijkstra 算法**

它们都是第 4 章中的经典算法。第 4 章的主题是图的处理算法，图是由结点和连接两个结点的边组成的一种重要的基础模型。图算法的基石就是图的搜索，也就是一个结点一个结点地查找，优 350
先队列在其中扮演了重要的角色。

- **Kruskal 算法**

这是图中的加权图的另一个经典算法，其中边的处理顺序取决于它的权重。算法的运行时间是由排序所需的时间决定的。

- **霍夫曼压缩**

这是一个经典的数据压缩算法。它处理的数据中的每个元素都有一个小整数作为权重，而处理的过程就是将权重最小的两个元素归并成一个新元素，并将其权重相加得到新元素的权重。使用优先队列可以立即实现这个算法。其他几种数据压缩算法也是基于排序的。

- **字符串处理**

字符串处理算法在现代密码学和基因组学中起着关键性的作用。它们也常常依赖于排序算法（一般都会使用第 5 章中所讨论的特殊的字符串排序算法）。例如，在第 6 章中我们在学习找出给定字符串中的最长重复子字符串算法时会先将字符串的后缀排序。 351

答疑

问 Java 的系统库中有优先队列这种数据类型吗？

答 有，请见 `java.util.PriorityQueue`。 352

练习

2.5.1 在下面这段 String 类型的 compareTo() 方法的实现中，第三行对提高运行效率有何帮助？

```
public int compareTo(String that)
{
   if (this == that) return 0;  // 这一行
   int n = Math.min(this.length(), that.length());
   for (int i = 0; i < n; i++)
   {
      if      (this.charAt(i) < that.charAt(i)) return -1;
      else if (this.charAt(i) > that.charAt(i)) return +1;
   }
   return this.length() - that.length();
}
```

2.5.2 编写一段程序，从标准输入读入一列单词并打印出其中所有由两个单词组成的组合词。例如，如果输入的单词为 after、thought 和 afterthought，那么 afterthought 就是一个组合词。

2.5.3 找出下面这段账户余额 Balance 类的实现代码的错误。为什么 compareTo() 方法对 Comparable 接口的实现有缺陷？

```
public class Balance implements Comparable<Balance>
{
   ...
   private double amount;
   public int compareTo(Balance that)
   {
      if (this.amount < that.amount - 0.005) return
-1;
      if (this.amount > that.amount + 0.005) return
+1;
      return 0;
   }
   ...
}
```

说明如何修正这个问题。

2.5.4 实现一个方法 String[] dedup(String[] a)，返回一个有序的 a[]，并删去其中的重复元素。

2.5.5 说明为何选择排序是不稳定的。

2.5.6 用递归实现 select()。

2.5.7 用 select() 找出 N 个元素中的最小值平均大约需要多少次比较？

2.5.8 编写一段程序 Frequency，从标准输入读取一列字符串并按照字符串出现频率由高到低的顺序打印出每个字符串及其出现次数。

2.5.9 为将右侧所示的数据排序编写一个新的数据类型。

2.5.10 创建一个数据类型 Version 来表示软件的版本，例如 115.1.1、115.10.1、115.10.2。为它实现 Comparable 接口，其中 115.1.1 的版本低于 115.10.1。

输入DJIA每天的成交量

```
 1-Oct-28      3500000
 2-Oct-28      3850000
 3-Oct-28      4060000
 4-Oct-28      4330000
 5-Oct-28      4360000
...
30-Dec-99    554680000
31-Dec-99    374049984
 3-Jan-00    931800000
 4-Jan-00   1009000000
 5-Jan-00   1085500032
 ...
```

输出

```
19-Aug-40 130000
26-Aug-40 160000
24-Jul-40 200000
10-Aug-42 210000
23-Jun-42 210000
...
23-Jul-02 2441019904
17-Jul-02 2566500096
15-Jul-02 2574799872
19-Jul-02 2654099968
24-Jul-02 2775559936
```

2.5.11 描述排序结果的一种方法是创建一个保存 0 到 a.length–1 的排列 p[]，使得 p[i] 的值为 a[i] 元素的最终位置。用这种方法描述插入排序、选择排序、希尔排序、归并排序、快速排序和堆排序对一个含有 7 个相同元素的数组的排序结果。

353 ⌇ 354

提高题

2.5.12 调度。编写一段程序 SPT.java，从标准输入中读取任务的名称和所需的运行时间，用 2.5.4.3 节所述的最短处理时间优先的原则打印出一份调度计划，使得任务完成的平均时间最小。

2.5.13 负载均衡。编写一段程序 LPT.java，接受一个整数 M 作为命令行参数，从标准输入中读取任务的名称和所需的运行时间，用 2.5.4.3 节所述的最长处理时间优先原则打印出一份调度计划，将所有任务分配给 M 个处理器并使得所有任务完成所需的总时间最少。

2.5.14 逆域名排序。为域名编写一个数据类型 Domain 并为它实现一个 compareTo() 方法，使之能够按照逆向的域名排序。例如，域名 cs.princeton.edu 的逆是 edu.princeton.cs。这在网络日志处理时很有用。提示：使用 s.split("\\.") 将域名用点分为若干部分。编写一个 Domain 的用例，从标准输入读取域名并将它们按照逆域名有序地打印出来。

2.5.15 垃圾邮件大战。在非法垃圾邮件之战的伊始，你有一大串来自各个域名（也就是电子邮件地址中 @ 符号后面的部分）的电子邮件地址。为了更好地伪造回信地址，你应该总是从相同的域中向目标用户发送邮件。例如，从 wayne@cs.princeton.edu 向 rs@cs.princeton.edu 发送垃圾邮件就很不错。你会如何处理这份电子邮件列表来高效地完成这个任务呢？

2.5.16 公正的选举。为了避免对名字排在字母表靠后的候选人的偏见，加州在 2003 年的州长选举中将所有候选人按照以下字母顺序排列：

R W Q O J M V A H B S G Z X N T C I E K U P D Y F L

创建一个遵守这种顺序的数据类型并编写一个用例 California，在它的静态方法 main() 中将字符串按照这种方式排序。假设所有字符串全部都是大写的。

2.5.17 检测稳定性。扩展练习 2.1.16 中的 check() 方法，对指定数组调用 sort()，如果排序结果是稳定的则返回 true，否则返回 false。不要假设 sort() 只会使用 exch() 移动数据。

355

2.5.18 强制稳定。编写一段能够将任意排序方法变得稳定的封装代码，创建一种新的数据类型作为键，将键的原始索引保存在其中，并在调用 sort() 之后再根据保存的索引恢复键的原始顺序。

2.5.19 Kendall tau 距离。编写一段程序 KendallTau.java，在线性对数时间内计算两组排列之间的 Kendall tau 距离。

2.5.20 空闲时间。假设有一台计算机能够并行处理 N 个任务。编写一段程序并给定一系列任务的起始时间和结束时间，找出这台机器最长的空闲时间和最长的繁忙时间。

2.5.21 多维排序。编写一个 Vector 数据类型并将 d 维整型向量排序。排序方法是先按照一维数字排序，一维数字相同的向量则按照二维数字排序，再相同的向量则按照三维数字排序，如此这般。

2.5.22 股票交易。投资者对一只股票的买卖交易都发布在电子交易市场中。他们会指定最高买入价和最低卖出价，以及在该价位买卖的笔数。编写一段程序，用优先队列来匹配买家和卖家并用模拟数据进行测试。可以使用两个优先队列，一个用于买家一个用于卖家，当一方的报价能够和另一方的一份或多份报价匹配时就进行交易。

2.5.23 选择的取样：实验使用取样来改进 select() 函数的想法。提示：使用中位数可能并不总是有效。

2.5.24 稳定的优先队列。实现一个稳定的优先队列（将重复的元素按照它们被插入的顺序返回）

2.5.25 平面上的点。为表 1.2.3 的 Point2D 类型编写三个静态的比较器，一个按照 x 坐标比较，一个按照 y 坐标比较，一个按照点到原点的距离进行比较。编写两个非静态的比较器，一个按照两点到第三点的距离比较，一个按照两点相对于第三点的幅角比较。

356 2.5.26 简单多边形。给定平面上的 N 个点，用它们画出一个多边形。提示：找到 y 坐标最小的点 p，在有多个最小 y 坐标的点时取 x 坐标最小者，然后将其他点按照以 p 为原点的幅角大小的顺序依次连接起来。

2.5.27 平行数组的排序。在将平行数组排序时，可以将索引排序并返回一个 index[] 数组。为 Insertion 添加一个 indirectSort() 方法，接受一个 Comparable 的对象数组 a[] 作为参数，但它不会将 a[] 中的元素重新排列，而是返回一个整形数组 index[] 使得 a[index[0]] 到 a[index[N-1]] 正好是升序的。

2.5.28 按文件名排序。编写一个 FileSorter 程序，从命令行接受一个目录名并打印出按照文件名排序后的所有文件。提示：使用 File 数据类型。

2.5.29 按大小和最后修改日期将文件排序。为 File 数据类型编写比较器，使之能够将文件按照大小、文件名或最后修改日期将文件升序或者降序排列。在程序 LS 中使用这些比较器，它接受一个命令行参数并根据指定的顺序列出目录的内容。例如，"-t" 指按照时间戳排序。支持多个选项以消除排序位次相同者，同时必须确保排序的稳定性。

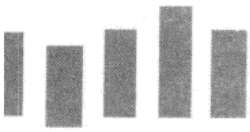
2.5.30 Boerner 定理。真假判断：如果你先将一个矩阵的每一列排序，再将矩阵的每一行排序，所有的
357 列仍然是有序的。证明你的结论。

实验题

2.5.31 重复元素。编写一段程序，接受命令行参数 M、N 和 T，然后使用正文中的代码进行 T 遍实验：生成 N 个 0 到 M–1 间的 int 值并计算重复值的个数。令 T=10，N=10^3、10^4、10^5 和 10^6 以及 M=N/2、N 和 2N。根据概率论，重复值的个数应该约为 $(1-e^{-\alpha})$，其中 α=N/M。打印一张表格来确认你的实验验证了这个公式。

2.5.32 8 字谜题。8 字谜题是 S. Ioyd 于 19 世纪 70 年代发明的一个游戏。游戏需要一个三乘三的九宫格，其中八格中填上了 1 到 8 这 8 个数字，一格空着。你的目标就是将所有的格子排序。可以将一个格子向上下或者左右移动（但不能是对角线方向）到空白的格子中。编写一个程序用 A* 算法解决这个问题。先用到达九宫格的当前位置所需的步数加上错位的格子数量作为优先级函数（注意，步数至少大于等于错位的格子数）。尝试用其他函数代替错位的格子数量，比如每个格子距离它的正确位置的曼哈顿距离，或是这些距离的平方之和。

2.5.33 随机交易。开发一个接受参数 N 的生成器，根据你能想到的任意假设条件生成 N 个随机的 Transaction 对象（请见练习 2.1.21 和练习 2.1.22）。对于 N=10^3、10^4、10^5 和 10^6，比较用希尔排序、
358 归并排序、快速排序和堆排序将 N 个交易排序的性能。



第3章 查 找

现代计算机和网络使我们能够访问海量的信息。高效检索这些信息的能力是处理它们的重要前提。本章描述的都是数十年来在广泛应用中经过实践检验的经典查找算法。没有这些算法，现代信息世界的基础计算设施都无从谈起。

我们会使用*符号表*这个词来描述一张抽象的表格，我们会将信息（*值*）存储在其中，然后按照指定的*键*来搜索并获取这些信息。键和值的具体意义取决于不同的应用。符号表中可能会保存很多键和很多信息，因此实现一张高效的符号表也是一项很有挑战性的任务。

符号表有时被称为*字典*，类似于那本将单词的释义按照字母顺序排列起来的历史悠久的参考书。在英语字典里，键就是单词，值就是单词对应的定义、发音和词源。符号表有时又叫做*索引*，即书本最后将术语按照字母顺序列出以方便查找的那部分。在一本书的索引中，键就是术语，而值就是书中该术语出现的所有页码。

在说明了基本的 API 和两种重要的实现之后，我们会学习用三种经典的数据类型来实现高效的符号表：二叉查找树、红黑树和散列表。在总结中我们会看到它们的若干扩展和应用，它们的实现都有赖于我们在本章中将会学到的高效算法。

359
~
361

3.1 符号表

符号表最主要的目的就是将一个键和一个值联系起来。用例能够将一个键值对插入符号表并希望在之后能够从符号表的所有键值对中按照键直接找到相对应的值。本章会讲解多种构造这样的数据结构的方法，它们不光能够高效地插入和查找，还可以进行其他几种方便的操作。要实现符号表，我们首先要定义其背后的数据结构，并指明创建并操作这种数据结构以实现插入、查找等操作所需的算法。

查找在大多数应用程序中都至关重要，许多编程环境也因此将符号表实现为高级的抽象数据结构，包括 Java——我们会在 3.5 节中讨论 Java 的符号表实现。表 3.1.1 给出的例子是在一些典型的应用场景中可能出现的键和值。我们马上会看到一些参考性的用例，3.5 节的目的就是向你展示如何在程序中有效地使用符号表。本书中我们还会在其他算法中使用符号表。

定义。符号表是一种存储键值对的数据结构，支持两种操作：**插入**（put），即将一组新的键值对存入表中；**查找**（get），即根据给定的键得到相应的值。

表 3.1.1 典型的符号表应用

应 用	查找的目的	键	值
字典	找出单词的释义	单词	释义
图书索引	找出相关的页码	术语	一串页码
文件共享	找到歌曲的下载地址	歌曲名	计算机 ID
账户管理	处理交易	账户号码	交易详情
网络搜索	找出相关网页	关键字	网页名称
编译器	找出符号的类型和值	变量名	类型和值

362

3.1.1 API

符号表是一种典型的*抽象数据类型*（请见第 1 章）：它代表着一组定义清晰的值以及相应的操作，使得我们能够将类型的实现和使用区分开来。和以前一样，我们要用应用程序编程接口（API）来精确地定义这些操作（如表 3.1.2 所示），为数据类型的实现和用例提供一份“契约”。

表 3.1.2 一种简单的泛型符号表 API

public class	**ST<Key, Value>**	
	ST()	创建一张符号表
void	put(Key key, Value, val)	将键值对存入表中（若值为空则将键 key 从表中删除）
Value	get(Key key)	获取键 key 对应的值（若键 key 不存在则返回 null）
void	delete(Key key)	从表中删去键 key（及其对应的值）
boolean	contains(Key key)	键 key 在表中是否有对应的值
boolean	isEmpty()	表是否为空
int	size()	表中的键值对数量
Iterable<Key>	keys()	表中的所有键的集合

在查看用例代码之前，为了保证代码的一致、简洁和实用，我们要先说明具体实现中的几个设计决策。

3.1.1.1 泛型

和排序一样，在设计方法时我们没有指定处理对象的类型，而是使用了泛型。对于符号表，我们通过明确地指定查找时键和值的类型来区分它们的不同角色，而不是像 2.4 节的优先队列那样将键和元素本身混为一谈。在考虑了这份基本的 API 后（例如，这里没有说明键的有序性），我们会用 Comparable 的对象来扩展典型的用例，这也会为数据类型带来许多新的方法。

3.1.1.2 重复的键

我们的所有实现都遵循以下规则：

- 每个键只对应着一个值（表中不允许存在重复的键）；
- 当用例代码向表中存入的键值对和表中已有的键（及关联的值）冲突时，新的值会替代旧的值。

这些规则定义了*关联数组的抽象形式*。你可以将符号表想象成一个数组，键即索引，值即数组的元素。在一个一般的数组中，键就是整型的索引，我们用它来快速访问数组的内容；在一个 363
关联数组（符号表）中，键可以是任意类型，但我们仍然可以用它来快速访问数组的内容。一些编程语言（非 Java）直接支持程序员使用 st[key] 来代替 st.get(key)，st[key]=val 来代替 st.put(key, val)，其中 key（键）和 val（值）都可以是任意类型的对象。

3.1.1.3 空（null）键

键不能为空。和 Java 中的许多其他机制一样，使用空键会产生一个运行时异常（请见本节答疑的第三条）。

3.1.1.4 空（null）值

我们还规定不允许有空值。这个规定的直接原因是在我们的 API 定义中，当键不存在时 get() 方法会返回空，这也意味着任何不在表中的键关联的值都是空。这个规定产生了两个（我们所期望的）结果：第一，我们可以用 get() 方法是否返回空来测试给定的键是否存在于符号表中；第二，我们可以将空值作为 put() 方法的第二个参数存入表中来实现删除，也就是 3.1.1.5 节的主要内容。

3.1.1.5 删除操作

在符号表中，删除的实现可以有两种方法：*延时删除*，也就是将键对应的值置为空，然后在某个时候删去所有值为空的键；或是*即时删除*，也就是立刻从表中删除指定的键。刚才已经说过，put(key, null) 是 delete(key) 的一种简单的（延时型）实现。而实现（即时型）delete() 就是为了替代这种默认的方案。在我们的符号表实现中不会使用默认的方案，而在本书的网站上 put() 实现的开头有这样一句防御性代码：

```
if (val == null) { delete(key); return; }
```

这保证了符号表中任何键的值都不为空。为了节省版面我们没有在本书中附上这段代码（我们也不会在调用 put() 时使用 null）。

3.1.1.6 便捷方法

为了用例代码的清晰，我们在 API 中加入了 contains() 和 isEmpty() 方法，它们的实现如表 3.1.3 所示，只需要一行。

表 3.1.3 默认实现

方 法	默认实现
void delete(Key key)	put(key, null);
boolean contains(key)	return get(key) != null;
boolean isEmpty()	return size() == 0;

为节省篇幅，我们不想重复这些代码，但我们约定它们存在于所有符号表 API 的实现中，用例
364 程序可以自由使用它们。

3.1.1.7 迭代

为了方便用例处理表中的所有键值，我们有时会在 API 的第一行加上 `implements Interable<Key>` 这句话，强制所有实现都必须包含 `iterator()` 方法来返回一个实现了 `hasNext()` 和 `next()` 方法的迭代器，如 1.3 节的栈和队列所述。但是对于符号表我们采用了一个更简单的方法。我们定义了 `keys()` 方法来返回一个 `Interable<Key>` 对象以方便用例遍历所有的键。这么做是为了和以后的有序符号表的所有方法保持一致，使得用例可以遍历表的键集的一个指定的部分。

3.1.1.8 键的等价性

要确定一个给定的键是否存在于符号表中，首先要确立对象等价性的概念。我们在 1.2.5.8 节深入讨论过这一点。在 Java 中，按照约定所有的对象都继承了一个 `equals()` 方法，Java 也为它的标准数据类型例如 `Integer`、`Double` 和 `String` 以及一些更加复杂的类型，如 `File` 和 `URL`，实现了 `equals()` 方法——当使用这些数据类型时你可以直接使用内置的实现。例如，如果 `x` 和 `y` 都是 `String` 类型，当且仅当 `x` 和 `y` 的长度相同且每个位置上的字母都相同时，`x.equals(y)` 返回 `true`。而自定义的键则需要如 1.2 节所述重写 `equals()` 方法。你可以参考我们为 `Date` 类型（请见 1.2.5.8 节）实现的 `equals()` 方法为你自己的数据类型实现 `equals()` 方法。和 2.4.4.5 节中讨论
365 的优先队列一样，最好使用不可变的数据类型作为键，否则表的一致性是无法保证的。

3.1.2 有序符号表

典型的应用程序中，键都是 `Comparable` 的对象，因此可以使用 `a.compareTo(b)` 来比较 `a` 和 `b` 两个键。许多符号表的实现都利用了 `Comparable` 接口带来的键的有序性来更好地实现 `put()` 和 `get()` 方法。更重要的是在这些实现中，我们可以认为符号表都会保持键的有序并大大扩展它的 API，根据键的相对位置定义更多实用的操作。例如，假设键是时间，你可能会对最早的或是最晚的键或是给定时间段内的所有键等感兴趣。在大多数情况下用实现 `put()` 和 `get()` 方法背后的数据结构都不难实现这些操作。于是，对于 `Comparable` 的键，在本章中我们实现了表 3.1.4 中的 API。

表 3.1.4 一种有序的泛型符号表的 API

public class ST<Key extends Comparable<key>, Value>		
	ST()	创建一张有序符号表
void	put(Key key, Value, val)	将键值对存入表中（若值为空则将键 key 从表中删除）
Value	get(Key key)	获取键 key 对应的值（若键 key 不存在则返回空）
void	delete(Key key)	从表中删去键 key（及其对应的值）
boolean	contains(Key key)	键 key 是否存在于表中
boolean	isEmpty()	表是否为空
int	size()	表中的键值对数量
Key	min()	最小的键
Key	max()	最大的键
Key	floor(Key key)	小于等于 key 的最大键
Key	ceiling(Key key)	大于等于 key 的最小键
int	rank(Key key)	小于 key 的键的数量
Key	select(int k)	排名为 k 的键

（续）

public class ST<Key extends Comparable<key>, Value>		
void	deleteMin()	删除最小的键
void	deleteMax()	删除最大的键
int	size(Key lo, Key hi)	[lo..hi] 之间键的数量
Iterable<Key>	keys(Key lo, Key hi)	[lo..hi] 之间的所有键，已排序
Iterable<Key>	keys()	表中的所有键的集合，已排序

366

只要你见到类的声明中含有泛型变量 Key extends Comparable<Key>，那就说明这段程序是在实现这份 API，其中的代码依赖于 Comparable 的键并且实现了更加丰富的操作。上面所有这些操作一起为用例定义了一个*有序符号表*。

3.1.2.1 最大键和最小键

对于一组有序的键，最自然的反应就是查询其中的最大键和最小键。我们在 2.4 节讨论优先队列时已经遇到过这些操作。在有序符号表中，我们也有方法删除最大键和最小键（以及它们所关联的值）。有了这些，符号表就具有了类似于 2.4 节中 IndexMinPQ() 的能力。主要的区别在于优先队列中可以存在重复的键但符号表中不行，而且有序符号表支持的操作更多。

3.1.2.2 向下取整和向上取整

对于给定的键，向下取整（floor）操作（找出小于等于该键的最大键）和向上取整（ceiling）操作（找出大于等于该键的最小键）有时是很有用的。这两个术语来自于实数的取整函数（对一个实数 x 向下取整即为小于等于 x 的最大整数，向上取整则为大于等于 x 的最小整数）。

3.1.2.3 排名和选择

检验一个新的键是否插入合适位置的基本操作是*排名*（rank，找出小于指定键的键的数量）和*选择*（select，找出排名为 k 的键）。要测试一下你是否完全理解了它们的作用，请确认对于 0 到 size()-1 的所有 i 都有 i==rank(select(i))，且所有的键都满足 key==select(rank(key))。2.5 节中我们在学习排序时已经遇到过对这两种操作的需求了。对于符号表，我们的挑战是在实现插入、删除和查找的同时快速实现这两种操作。

有序符号表的操作示例如表 3.1.5 所示。

表 3.1.5 有序符号表的操作示例

	键	值
min() →	**09:00:00**	Chicago
	09:00:03	Phoenix
	09:00:13	**Houston**
get(09:00:13) →	09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00) →	**09:03:13**	Chicago
	09:10:11	Seattle
select(7) →	**09:10:25**	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00) →	**09:21:05**	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00) →	**09:35:21**	Chicago
	09:36:14	Seattle
max() →	**09:37:44**	Phoenix

size(09:15:00, 09:25:00) = 5
rank(09:10:25) = 7

367

3.1.2.4 范围查找

给定范围内（在两个给定的键之间）有多少键？是哪些？在很多应用中能够回答这些问题并接受两个参数的 size() 和 keys() 方法都很有用，特别是在大型数据库中。能够处理这类查询是有序符号表在实践中被广泛应用的重要原因之一。

3.1.2.5 例外情况

当一个方法需要返回一个键但表中却没有合适的键可以返回时，我们约定抛出一个异常（另一

种合理的方法是在这种情况下返回空）。例如，在符号表为空时，min()、max()、deleteMin()、deleteMax()、floor() 和 ceiling() 都会抛出异常，当 k<0 或 k>=size() 时 select(k) 也会抛出异常。

3.1.2.6 便捷方法

在基础 API 中我们已经见过了 contains() 和 isEmpty() 方法，为了用例的清晰我们又在 API 中添加了一些冗余的方法。为了节约版面，除非特别声明，我们约定所有有序符号表 API 的实现都含有如表 3.1.6 所示的方法。

表 3.1.6 有序符号表中冗余有序性方法的默认实现

方 法	默认的实现
`void deleteMin()`	`delete(min());`
`void deleteMax()`	`delete(max());`
`int size(Key lo, Key hi)`	`if (hi.compareTo(lo) < 0)` `  return 0;` `else if (contains(hi))` `  return rank(hi) - rank(lo) + 1;` `else` `  return rank(hi) - rank(lo);`
`Iterable<Key> keys()`	`return keys(min(), max());`

3.1.2.7 （再谈）键的等价性

Java 的一条最佳实践就是维护所有 Comparable 类型中 compareTo() 方法和 equals() 方法的一致性。也就是说，任何一种 Comparable 类型的两个值 a 和 b 都要保证 (a.compareTo(b)==0) 和 a.equals(b) 的返回值相同。为了避免任何潜在的二义性，我们不会在有序符号表的实现中使用 equals() 方法。作为替代，我们只会使用 compareTo() 方法来比较两个键，即我们用布尔表达式 a.compareTo(b)==0 来表示“a 和 b 相等吗？”。一般来说，这样的比较都代表着在符号表中
368 的一次成功查找（找到了 b）。和排序算法一样，Java 为许多经常作为键的数据类型提供了标准的 compareTo() 方法，为你自定义的数据类型实现一个 compareTo() 方法也不困难（参见 2.5 节）。

3.1.2.8 成本模型

无论我们是使用 equals() 方法（对于符号表的键不是 Comparable 对象而言）还是 compareTo() 方法（对于符号表的键是 Comparable 对象而言），我们使用比较一词来表示将一个符号表条目和一个被查找的键进行比较操作。在大多数的符号表实现中，这个操作都出现在内循环。在少数的例外中，我们则会统计数组的访问次数。

查找的成本模型。在学习符号表的实现时，我们会统计比较的次数（等价性测试或是键的相互比较）。在内循环不进行比较（极少）的情况下，我们会统计数组的访问次数。

符号表实现的重点在于其中使用的数据结构和 get()、put() 方法。在下文中我们不会总是给出其他方法的实现，因为将它们作为练习能够更好地检验你对实现背后的数据结构的理解程度。为了区别不同的实现，我们在特定的符号表实现的类名前加上了描述性前缀。在用例代码中，除非我们想使用一个特定的实现，我们都会使用 ST 表示一个符号表实现。在本章和其他章节中，经过学习和讨论过大量符号表的使用和实现后你会慢慢地理解这些 API 的设计初衷。同时我们也会在答疑
369 和练习中讨论算法设计时的更多选择。

3.1.3 用例举例

虽然我们会在 3.5 节中详细说明符号表的更多应用，在学习它的实现之前我们还是应该先看看如何使用它。相应地我们这里考察两个用例：一个用来跟踪算法在小规模输入下的行为测试用例，和一个用来寻找更高效的实现的性能测试用例。

3.1.3.1 行为测试用例

为了在小规模的输入下跟踪算法的行为，我们用以下测试用例测试我们对符号表的所有实现。这段代码会从标准输入接受多个字符串，构造一张符号表来将 i 和第 i 个字符串相关联，然后打印符号表。在本书中我们假设所有的字符串都只有一个字母。一般我们会使用 "S E A R C H E X A M P L E"。按照我们的约定，用例会将键 S 和 0，键 R 和 3 关联起来，等等。但 E 的值是 12（而非 1 或者 6），A 的值为 8（而非 2），因为我们的关联型数组意味着每个键的值取决于最近一次 put() 方法的调用。对于符号表的简单实现（无序），用例的输出中键的顺序是不确定的（这和具体实现有关）；对于有序符号表，用例应该将键按顺序打印出来。这是一种索引用例，它是我们将在 3.5 节中讨论的一种重要的符号表应用的一个特殊情况。

测试用例的实现代码如下所示。测试用例的键、值及输出如图 3.1.1 所示。

```
public static void main(String[] args)
{
   ST<String, Integer> st;
   st = new ST<String, Integer>();

   for (int i = 0; !StdIn.isEmpty(); i++)
   {
      String key = StdIn.readString();
      st.put(key, i);
   }

   for (String s : st.keys())
      StdOut.println(s + " " + st.get(s));
}
```

简单的符号表测试用例

键	S	E	A	R	C	H	E	X	A	M	P	L	E
值	0	1	2	3	4	5	6	7	8	9	10	11	12

简单符号表的（一种可能的）输出		有序符号表的输出	
L	11	A	8
P	10	C	4
M	9	E	12
X	7	H	5
H	5	L	11
C	4	M	9
R	3	P	10
A	8	R	3
E	12	S	0
S	0	X	7

图 3.1.1 测试用例的键、值和输出

370

3.1.3.2 性能测试用例

FrequencyCounter 用例会从标准输入中得到的一列字符串并记录每个（长度至少达到指定的阈值）字符串的出现次数，然后遍历所有键并找出出现频率最高的键。这是一种字典，我们会在 3.5 节中更加详细地讨论这种应用。这个用例回答了一个简单的问题：哪个（不小于指定长度的）单词在一段文字中出现的频率最高？在本章中，我们会用这个用例以及三段文字来进行性能测试：狄更斯的《双城记》中的前五行（tinyTale.txt），《双城记》全书（tale.txt），以及一个知名的叫做 Leipzig Corpora Collection 的数据库（leipzig1M.txt），内容为一百万条随机从网络上抽取的句子。例如，这是 tinyTale.txt 的内容：

```
% more tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
```

小型测试输入

这段文字共有60个单词，去掉重复的单词还剩20个，其中4个出现了10次（频率最高）。对于这段文字，FrequencyCounter可能会打印出it、was、the或者of中的某一个单词（具体会打印出哪一个取决于符号表的具体实现），以及它出现的频率10。表3.1.7总结了大型测试输入流的性质。

表3.1.7 大型测试输入流的性质

	TinyTale.txt		tale.txt		leipzig1M.txt	
	单词数	不同的单词数	单词数	不同的单词数	单词数	不同的单词数
所有单词	60	20	135 635	10 679	21 191 455	534 580
长度大于等于8的单词	3	3	14 350	5 737	4 239 597	299 593
长度大于等于10的单词	2	2	4 582	2 260	1 610 829	165 555

371

FrequencyCounter用例实现过程如下所示。

符号表的用例

```
public class FrequencyCounter
{
   public static void main(String[] args)
   {
      int minlen = Integer.parseInt(args[0]);   // 最小键长
      ST<String, Integer> st = new ST<String, Integer>();
      while (!StdIn.isEmpty())
      {  // 构造符号表并统计频率
         String word = StdIn.readString();
         if (word.length() < minlen) continue;  // 忽略较短的单词
         if (!st.contains(word)) st.put(word, 1);
         else                    st.put(word, st.get(word) + 1);
      }
      // 找出出现频率最高的单词
      String max = " ";
```

```
        st.put(max, 0);
        for (String word : st.keys())
           if (st.get(word) > st.get(max))
              max = word;
        StdOut.println(max + " " + st.get(max));
     }
  }
```

这个符号表的用例统计了标准输入中各个单词的出现频率，然后将频率最高的单词打印出来。命令行参数指定了表中的键的最短长度。

```
% java FrequencyCounter 1 < tinyTale.txt
it 10
% java FrequencyCounter 8 < tale.txt
business 122
% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

372

研究符号表处理大型文本的性能要考虑两个方面的因素：首先，每个单词都会被作为键进行搜索，因此处理性能和输入文本的单词总量必然有关；其次，输入的每个单词都会被存入符号表（输入中不重复单词的总数也就是所有键都被插入以后符号表的大小），因此输入流中不同的单词的总数也是相关的。我们需要这两个量来估计 FrequencyCounter 的运行时间（作为开始，请见练习 3.1.6）。我们会在学习了一些算法之后再回头说明一些细节，但你应该对类似这样的符号表应用的需求有一个大致的印象。例如，用 FrequencyCounter 分析 leipzig1M.txt 中长度不小于 8 的单词意味着，在一个含有数以千计的键值对的符号表中进行上百万次的查找，而互联网中的一台服务器可能需要在含有上百万个键值对的表中处理上亿的交易。

这个用例和所有这些例子都提出了一个简单的问题：我们的实现能够在一张用多次 get() 和 put() 方法构造出的巨型符号表中进行大量的 get() 操作吗？如果我们的查找操作不多，那么任意实现都能够满足需要。但没有一个高效的符号表作为基础是无法使用 FrequencyCounter 这样的程序来处理大型问题的。FrequencyCounter 是一种极为常见的应用的代表，它的这些特性也是许多其他符号表应用的共性：

- 混合使用查找和插入的操作；
- 大量的不同键；
- 查找操作比插入操作多得多；
- 虽然不可预测，但查找和插入操作的使用模式并非随机。

我们的目标就是实现一种符号表来满足这些能够解决典型的实际问题的用例的需要。

下面，我们将会学习两种初级的符号表实现并通过 FrequencyCounter 分别评估它们的性能。在之后的几节中，你会学习一些经典的实现，即使对于庞大的输入和符号表它们的性能仍然非常优秀。

373

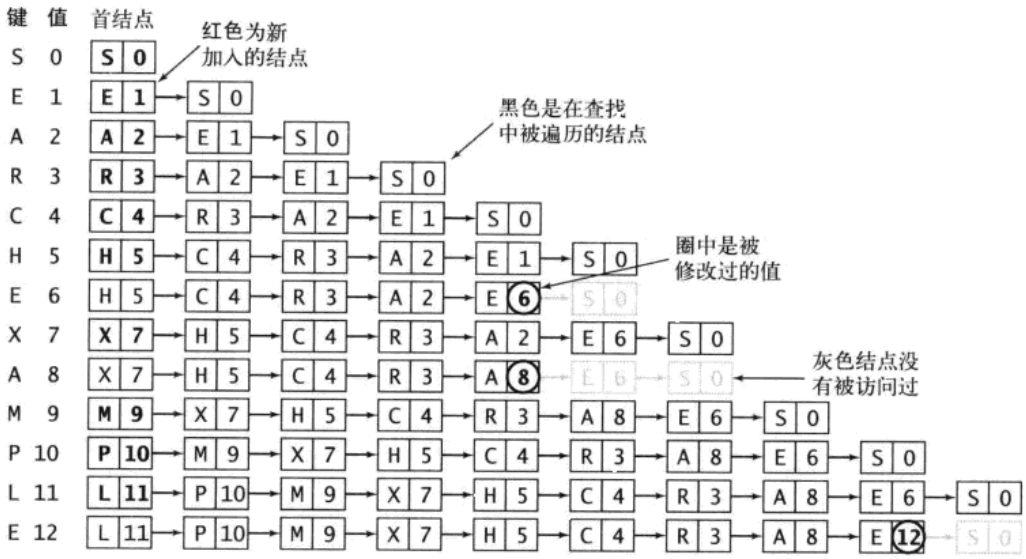
3.1.4 无序链表中的顺序查找

符号表中使用的数据结构的一个简单选择是链表，每个结点存储一个键值对，如算法 3.1 中的代码所示。get() 的实现即为遍历链表，用 equals() 方法比较需被查找的键和每个结点中

的键。如果匹配成功我们就返回相应的值，否则我们返回 null。put() 的实现也是遍历链表，用 equals() 方法比较需被查找的键和每个结点中的键。如果匹配成功我们就用第二个参数指定的值更新和该键相关联的值，否则我们就用给定的键值对创建一个新的结点并将其插入到链表的开头。这种方法也被称为顺序查找：在查找中我们一个一个地顺序遍历符号表中的所有键并使用 equals() 方法来寻找与被查找的键匹配的键。

算法 3.1（SequentialSearchST）用链表实现了符号表的基本 API，我们在第 1 章中的基础数据结构中学习过它。这里我们将 size()、keys() 和即时型的 delete() 方法留做练习。这些练习能够巩固并加深你对链表和符号表的基本 API 的理解。

这种基于链表的实现能够用于和我们的用例类似的、需要大型符号表的应用吗？我们已经说过，分析符号表算法比分析排序算法更困难，因为不同的用例所进行的操作序列各不相同。对于 FrequencyCounter，最常见的情形是虽然查找和插入的使用模式是不可预测的，但它们的使用肯定不是随机的。因此我们主要研究最坏情况下的性能。为了方便，我们使用命中表示一次成功的查找，未命中表示一次失败的查找。使用基于链表的符号表的索引用例的轨迹如图 3.1.2 所示。



374

图 3.1.2 使用基于链表的符号表的索引用例的轨迹（另见彩插）

算法 3.1 顺序查找（基于无序链表）

```
public class SequentialSearchST<Key, Value>
{
   private Node first;          // 链表首结点
   private class Node
   {  // 链表结点的定义
      Key key;
      Value val;
```

```
      Node next;
      public Node(Key key, Value val, Node next)
      {
         this.key  = key;
         this.val  = val;
         this.next = next;
      }
   }
   public Value get(Key key)
   {  // 查找给定的键，返回相关联的值
      for (Node x = first; x != null; x = x.next)
         if (key.equals(x.key))
            return x.val;     // 命中
      return null;             // 未名中
   }
   public void put(Key key, Value val)
   {  // 查找给定的键，找到则更新其值，否则在表中新建结点
      for (Node x = first; x != null; x = x.next)
         if (key.equals(x.key))
         {  x.val = val; return;  }       // 命中，更新
      first = new Node(key, val, first); // 未命中，新建结点
   }
}
```

符号表的实现使用了一个私有内部 Node 类来在链表中保存键和值。get() 的实现会顺序地搜索链表查找给定的键（找到则返回相关联的值）。put() 的实现也会顺序地搜索链表查找给定的键，如果找到则更新相关联的值，否则它会用给定的键值对创建一个新的结点并将其插入到链表的开头。size()、keys() 和即时型的 delete() 方法的实现留做练习。

375

命题 A。在含有 N 对键值的基于（无序）链表的符号表中，未命中的查找和插入操作都需要 N 次比较。命中的查找在最坏情况下需要 N 次比较。特别地，向一个空表中插入 N 个不同的键需要 ~ $N^2/2$ 次比较。

证明。在表中查找一个不存在的键时，我们会将表中的每个键和给定的键比较。因为不允许出现重复的键，每次插入操作之前我们都需要这样查找一遍。

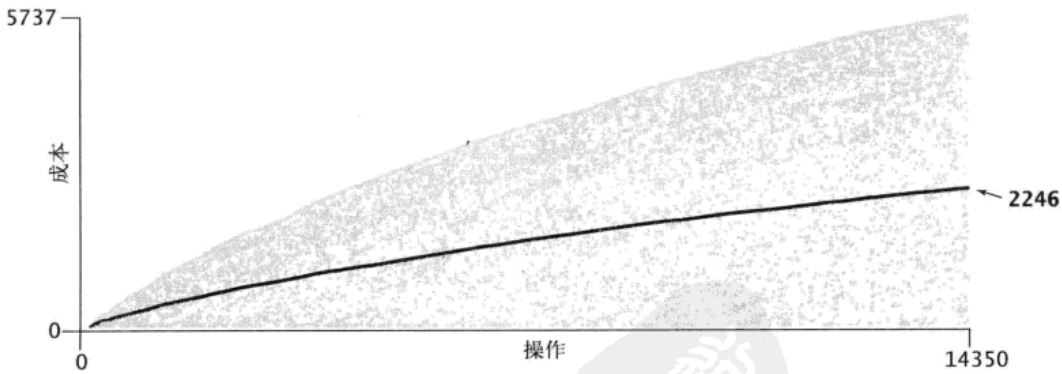
推论。向一个空表中插入 N 个不同的键需要 ~ $N^2/2$ 次比较。

查找一个已经存在的键并不需要线性级别的时间。一种度量方法是查找表中的每个键，并将总时间除以 N。在查找表中的每个键的可能性都相同的情况下时，这个结果就是一次查找平均所需的比较数。我们将它称为随机命中。尽管符号表用例的查找模式不太可能是随机的，这个模型也总能适应得很好。我们很容易就可以得到随机命中所需的平均比较次数为 $\sim N/2$：算法 3.1 中的 get() 方法查找第一个键需要 1 次比较，查找第二个键需要 2 次比较，如此这般，平均比较次数为 $(1+2+\cdots+N)/N=(N+1)/2\sim N/2$。

这些分析完全证明了基于链表的实现以及顺序查找是非常低效的，无法满足 FrequencyCounter 处理庞大输入问题的需求。比较的总次数和查找次数与插入次数的乘积成正比。对于《双城记》这个数字大于 10^9，而对于 Leipzig Corpora 数据库这个数字大于 10^{14}。

按照惯例，为了验证分析结果我们需要进行一些实验。这里我们用 FrequencyCounter 以及命令行参数 8 来分析 tale.txt。这将需要 14 350 次 put()（已经说过，输入中的每个单词都需要一次 put() 操作来更新它的出现频率，contains() 方法的调用是可以避免的，这里忽略了它的成本）。符号表将包含 5737 个键，也就是说大约三分之一的操作都将表增大了，其余操作为查找。为了将性能可视化我们使用了 VisualAccumulator（请见表 1.2.14）将每次 put() 操作转换为两个点：对于第 i 次 put() 操作，我们会在横坐标为 i，纵坐标为该次操作所进行的比较次数的位置画一个灰点，以及横坐标为 i，纵坐标为前 i 次 put() 操作累计所需的平均比较次数的位置画一个黑点，如图 3.1.3 所示。和所有科学实验数据一样，这其中包含了很多信息供我们研究（这张图含有 14 350 个灰点和 14 350 个黑点）。这里，我们的主要兴趣在于这张表证实了我们关于 put() 平均需要访问半条链表的猜想。虽然实际的数据比一半稍少，但对这个事实（以及图表曲线的形状）最好的解释应该是应用的特性，而非算法（请见练习 3.1.36）。

尽管某个具体用例的性能特点可能是复杂的，但只要使用我们准备的文本或者随机有序输入以及我们在第 1 章中介绍的 DoublingTest 程序，我们还是能够轻松估计出 FrequencyCounter 的性能并测试验证的。我们将这些测试留给练习和接下来将要学习的更加复杂的实现。如果你并不觉得我们需要更快的实现，请一定完成这些练习！（或者用 FrequencyCounter 调用 SequentialSearchST 来处理 leipzig1M.txt！）



376
↓
377

图 3.1.3 使用 SequentialSearchST，运行 java FrequencyCounter 8 < tale.txt 的成本

3.1.5 有序数组中的二分查找

下面我们要学习有序符号表 API 的完整实现。它使用的数据结构是一对平行的数组，一个存储

键一个存储值。算法 3.2（BinarySearchST）可以保证数组中 Comparable 类型的键有序，然后使用数组的索引来高效地实现 get() 和其他操作。

这份实现的核心是 rank() 方法，它返回表中小于给定键的键的数量。对于 get() 方法，只要给定的键存在于表中，rank() 方法就能够精确地告诉我们在哪里能够找到它（如果找不到，那它肯定就不在表中了）。

对于 put() 方法，只要给定的键存在于表中，rank() 方法就能够精确地告诉我们到哪里去更新它的值，以及当键不在表中时将键存储到表的何处。我们将所有更大的键向后移动一格来腾出位置（从后向前移动）并将给定的键值对分别插入到各自数组中的合适位置。结合我们测试用例的轨迹来研究 BinarySearchST 也是学习这种数据结构的好方法。

这段代码为键和值使用了两个数组（另一种方式请见练习 3.1.12）。和我们在第 1 章中对泛型的栈和队列的实现一样，这段代码也需要创建一个 Key 类型的 Comparable 对象的数组和一个 Value 类型的 Object 对象的数组，并在构造函数中将它们转化回 Key[] 和 Value[]。和以前一样，我们可以动态调整数组，使得用例无需担心数组大小（请注意，你会发现这种方法对于大数组实在是太慢了）。

使用基于有序数组的符号表实现的索引用例的轨迹如表 3.1.8 所示。

表 3.1.8 使用基于有序数组的符号表实现的索引用例的轨迹

键	值
S	0
E	1
A	2
R	3
C	4
H	5
E	6
X	7
A	8
M	9
P	10
L	11
E	12

keys[]

N	0	1	2	3	4	5	6	7	8	9
1	**S**									
2	**E**	S								
3	**A**	E	S							
4	A	E	**R**	S						
5	A	**C**	E	R	S					
6	A	C	E	**H**	R	S				
6	A	C	E	H	R	S				
7	A	C	E	H	R	S	**X**			
7	A	C	E	H	R	S	X			
8	A	C	E	H	**M**	R	S	X		
9	A	C	E	H	M	**P**	R	S	X	
10	A	C	E	H	**L**	M	P	R	S	X
10	A	C	E	H	L	M	P	R	S	X
	A	C	E	H	L	M	P	R	S	X

vals[]

N	0	1	2	3	4	5	6	7	8	9
1	**0**									
2	**1**	0								
3	**2**	1	0							
4	2	1	**3**	0						
5	2	**4**	1	3	0					
6	2	4	1	**5**	3	0				
6	2	4	**6**	5	3	0				
7	2	4	6	5	3	0	**7**			
7	**8**	4	6	5	3	0	7			
8	8	4	6	5	**9**	3	0	7		
9	8	4	6	5	9	**10**	3	0	7	
10	8	4	6	5	**11**	9	10	3	0	7
10	8	4	**12**	5	11	9	10	3	0	7
	8	4	12	5	11	9	10	3	0	7

黑色加粗的是被插入的元素

灰色的是没有被移动过的元素

黑色的是向右移动过的元素

圈中的是修改过的值

378

算法 3.2 二分查找（基于有序数组）

```
public class BinarySearchST<Key extends Comparable<Key>, Value>
{
   private Key[] keys;
   private Value[] vals;
   private int N;
   public BinarySearchST(int capacity)
   {  // 调整数组大小的标准代码请见算法1.1
      keys = (Key[]) new Comparable[capacity];
      vals = (Value[]) new Object[capacity];
```

```
   }
   public int size()
   {  return N;  }
   public Value get(Key key)
   {
      if (isEmpty()) return null;
      int i = rank(key);
      if (i < N && keys[i].compareTo(key) == 0) return vals[i];
      else                                      return null;
   }

   public int rank(Key key)
   // 请见算法3.2（续1）

   public void put(Key key, Value val)
   {  // 查找键，找到则更新值，否则创建新的元素
      int i = rank(key);
      if (i < N && keys[i].compareTo(key) == 0)
      {  vals[i] = val; return;  }
      for (int j = N; j > i; j--)
      {  keys[j] = keys[j-1]; vals[j] = vals[j-1];  }
      keys[i] = key; vals[i] = val;
      N++;
   }

   public void delete(Key key)
   // 该方法的实现请见练习3.1.16

}
```

这段符号表的实现用两个数组来保存键和值。和1.3节中基于数组的栈一样，put()方法会在插入
379 新元素前将所有较大的键向后移动一格。这里省略了调整数组大小部分的代码。

3.1.5.1 二分查找

我们使用有序数组存储键的原因是，第1章中作为例子出现的经典二分查找法能够根据数组的索引大大减少每次查找所需的比较次数。我们会使用有序索引数组来标识被查找的键可能存在的子数组的大小范围。在查找时，我们先将被查找的键和子数组的中间键比较。如果被查找的键小于中间键，我们就在左子数组中继续查找，如果大于我们就在右子数组中继续查找，否则中间键就是我们要找的键。算法3.2（续1）中实现rank()方法的代码使用了刚才讨论的二分查找法。这个实现值得我们仔细研究。作为开始，我们来看看这段等价的递归代码。

```
public int rank(Key key, int lo, int hi)
{
   if (hi < lo) return lo;
   int mid = lo + (hi - lo) / 2;
   int cmp = key.compareTo(keys[mid]);
   if      (cmp < 0)
        return rank(key, lo, mid-1);
   else if (cmp > 0)
        return rank(key, mid+1, hi);
   else return mid;
}
```

递归的二分查找

调用这里的rank(key, 0, N-1)所进行的比较和调用算法3.2（续1）的实现所进行的比较完全相同。但如1.1节中讨论的，这个版本更好地暴露了算法的结构。递归的rank()保留了以下性质：

- 如果表中存在该键，rank()应该返回该键的位置，也就是表中小于它的键的数量；

❑ 如果表中不存在该键，rank() 还是应该返回表中小于它的键的数量。

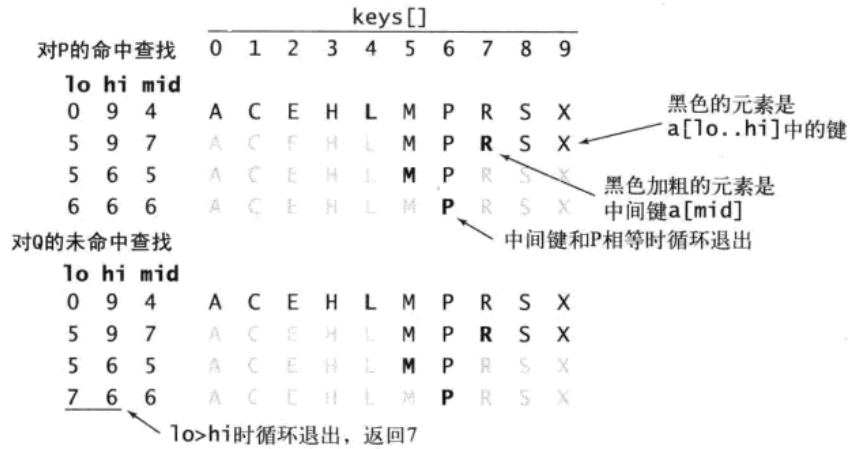
好好想想算法 3.2（续 1）中非递归的 rank() 为什么能够做到这些（你可以证明两个版本的等价性，或者直接证明非递归版本中的循环在结束时 lo 的值正好等于表中小于被查找的键的键的数量），所有程序员都能从这些思考中有所收获。（提示：lo 的初始值为 0，且永远不会变小）

算法 3.2（续 1） 基于有序数组的二分查找（迭代）

```
public int rank(Key key)
{
   int lo = 0, hi = N-1;
   while (lo <= hi)
   {
      int mid = lo + (hi - lo) / 2;
      int cmp = key.compareTo(keys[mid]);
      if      (cmp < 0) hi = mid - 1;
      else if (cmp > 0) lo = mid + 1;
      else return mid;

   }
   return lo;
}
```

该方法实现了正文所述的经典算法来计算小于给定键的键的数量。它首先将 key 和中间键比较，如果相等则返回其索引；如果小于中间键则在左半部分查找；大于则在右半部分查找。



在有序数组中使用二分法查找排名的轨迹

算法 3.2（续 2）基于二分查找的有序符号表的其他操作

```
public Key min()
{  return keys[0];  }

public Key max()
{  return keys[N-1];  }

public Key select(int k)
{  return keys[k];  }
```

```
public Key ceiling(Key key)
{
   int i = rank(key);
   return keys[i];
}

public Key floor(Key key)
// 请见练习3.1.17

public Key delete(Key key)
// 请见练习3.1.16

public Iterable<Key> keys(Key lo, Key hi)
{
   Queue<Key> q = new Queue<Key>();
   for (int i = rank(lo); i < rank(hi); i++)
   q.enqueue(keys[i]);
   if (contains(hi))
      q.enqueue(keys[rank(hi)]);
   return q;
}
```

这些方法，以及练习 3.1.16 和练习 3.1.17，组成了我们对使用二分查找的有序符号表的完整实现。`min()`、`max()` 和 `select()` 方法都很简单，只需按照给定的位置从数组中返回相应的值即可。`rank()` 方法实现了二分查找，是其他方法的基石。`floor()` 和 `delete()` 方法虽然也不难，但稍微复杂一些，在此留做练习。

3.1.5.2 其他操作

因为键被保存在有序数组中，算法 3.2（续 2）中和顺序有关的大多数操作都一目了然。例如，调用 `select(k)` 就相当于返回 `keys[k]`。我们将 `delete()` 和 `floor()` 留做练习。你应该研究一下 `ceiling()` 和带两个参数的 `keys()` 方法的实现，并完成练习来巩固和加深你对有序符号表的 API 及其实现的理解。

380 ~ 382

3.1.6 对二分查找的分析

`rank()` 的递归实现还能够让我们立即得到一个结论：二分查找很快，因为递归关系可以说明算法所需比较次数的上界。

命题 B。 在 N 个键的有序数组中进行二分查找最多需要（$\lg N+1$）次比较（无论是否成功）。

证明。 这里的分析和对归并排序的分析（第 2 章的命题 F）类似（但相对简单）。令 $C(N)$ 为在大小为 N 的符号表中查找一个键所需进行的比较次数。显然我们有 $C(0)=0$，$C(1)=1$，且对于 $N>0$ 我们可以写出一个和递归方法直接对应的归纳关系式：

$$C(N) \leqslant C(\lfloor N/2 \rfloor)+1$$

无论查找会在中间元素的左侧还是右侧继续，子数组的大小都不会超过 $\lfloor N/2 \rfloor$，我们需要一次比较来检查中间元素和被查找的键是否相等，并决定继续查找左侧还是右侧的子数组。当 N 为 2 的幂减 1 时（$N=2^n-1$），这种递推很容易。首先，因为 $\lfloor N/2 \rfloor=2^{n-1}-1$，所以我们有：

$$C(2^n-1) \leqslant C(2^{n-1}-1)+1$$

用这个公式代换不等式右边的第一项可得：

$$C(2^n-1) \leqslant C(2^{n-2}-1)+1+1$$

将上面这一步重复 $n-2$ 次可得：

$$C(2^n-1) \leqslant C(2^0)+n$$

最后的结果即：

$$C(N)=C(2^n) \leqslant n+1<\lg N+1$$

对于一般的 N，确切的结论更加复杂，但不难通过以上论证推广得到（请见练习 3.1.20）。二分查找所需时间必然在对数范围之内。

刚才给出的实现中，ceiling() 只是调用了一次 rank()，而接受两个参数的默认 size() 方法调用了两次 rank()，因此这份证明也保证了这些操作（包括 floor()）所需的时间最多是对数级别的（min()、max() 和 select() 操作所需的时间都是常数级别的）。

尽管能够保证查找所需的时间是对数级别的，BinarySearchST 仍然无法支持我们用类似 FrequencyCounter 的程序来处理大型问题，因为 put() 方法还是太慢了。二分查找减少了比较的次数但无法减少运行所需时间，因为它无法改变以下事实：在键是随机排列的情况下，构造一个基于有序数组的符号表所需要访问数组的次数是数组长度的平方级别（在实际情况下键的排列虽然不是随机的，但仍然很好地符合这个模型）。BinarySearchST 的操作的成本如表 3.1.9 所示。

命题 B（续）。向大小为 N 的有序数组中插入一个新的元素在最坏情况下需要访问~$2N$ 次数组，因此向一个空符号表中插入 N 个元素在最坏情况下需要访问~N^2 次数组。

证明。同命题 A。

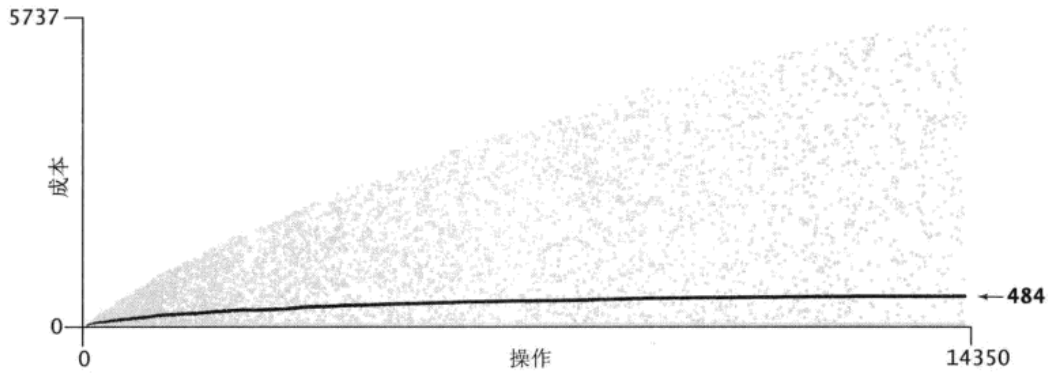
表 3.1.9 BinarySearchST 的操作的成本

方　法	运行所需时间的增长数量级
put()	N
get()	$\log N$
delete()	N
contains()	$\log N$
size()	1
min()	1
max()	1
floor()	$\log N$
ceiling()	$\log N$
rank()	$\log N$
select()	1
deleteMin()	N
deleteMax()	1

383

对于含有 10^4 个不同键的《双城记》，构建符号表需要访问数组约 10^8 次；而对于含有 10^6 个不同键的 Leipzig 项目则需要访问数组 10^{11} 次。虽然现代计算机可勉强实现，但这样的成本还是过高了。

回头看看 FrequencyCounter 在参数为 8 时 put() 操作的性能，我们可以看到平均情况下的比较次数（包括访问数组的次数）从 SequentialSearchST 的 2246 次降低到了 BinarySearchST 的 484 次（如图 3.1.4 所示）。这比我们在分析中预测的还要更好，额外的部分可能能够再次通过应用的性质得到解释（请见练习 3.1.36）。这次改进令人印象深刻，但你会看到，我们还能做得更好。



384 图 3.1.4 使用 BinarySearchST，运行 java FrequencyCounter 8 < tale.txt 的成本

3.1.7 预览

一般情况下二分查找都比顺序查找快得多，它也是众多实际应用程序的最佳选择。对于一个静态表（不允许插入）来说，将其在初始化时就排序是值得的，如第 1 章中的二分查找所示（请见表 1.2.15）。即使查找前所有的键值对已知（这在应用程序中是一种常见的情况），为 BinarySearchST 添加一个能够初始化并将符号表排序的构造函数也是有意义的（请见练习 3.1.12）。当然，二分查找也不适合很多应用。例如，它无法处理 Leipzig Corpora 数据库，因为查找和插入操作是混合进行的，而且符号表也太大了。如我们所强调的那样，现代应用需要同时能够支持高效的查找和插入两种操作的符号表实现。也就是说，我们需要在构造庞大的符号表的同时能够任意插入（也许还有删除）键值对，同时也要能够完成查找操作。

表 3.1.10 给出了本节中介绍的符号表的初级实现的性能特点。表中给出的是总成本中的最高级项（对于二分查找是数组的访问次数，对于其他则是比较次数），即运行时间的增长数量级。

表 3.1.10 简单的符号表实现的成本总结

算法（数据结构）	最坏情况下的成本（N 次插入后）		平均情况下的成本（N 次随机插入后）		是否高效地支持有序性相关的操作
	查 找	插 入	查 找	插 入	
顺序查找（无序链表）	N	N	$N/2$	N	否
二分查找（有序数组）	$\lg N$	$2N$	$\lg N$	N	是

核心的问题在于我们能否找到能够同时保证查找和插入操作都是对数级别的算法和数据结构。答案是令人兴奋的“可以”！这个答案也正是本章的重点所在。和第 2 章讨论的高效排序算法一样，能够高效地查找和插入的符号表是算法领域对世界最重要的贡献之一，也是我们今天能够享受的丰富计算性基础设施的开发基础。

我们如何能够实现这个目标呢？要支持高效的插入操作，我们似乎需要一种链式结构。但单链接的链表是无法使用二分查找法的，因为二分查找的高效来自于能够快速通过索引取得任何子数组
385 的中间元素（但得到一条链表的中间元素的唯一方法只能是沿链表遍历）。为了将二分查找的效率和链表的灵活性结合起来，我们需要更加复杂的数据结构。能够同时拥有两者的就是二叉查找树，它也是我们下面两节的主题。我们会将散列表留到 3.4 节中讨论。

在本章中我们会学习 6 种符号表的实现，这里我们先给出一个简单的预览。表 3.1.11 包含一系列数据结构以及它们适用和不适用于某个应用场景的原因，按照我们学习它们的先后顺序排列。

表 3.1.11 符号表的各种实现的优缺点

使用的数据结构	实 现	优 点	缺 点
链表（顺序查找）	SequentialSearchST	适用于小型问题	对于大型符号表很慢
有序数组（二分查找）	BinarySearchST	最优的查找效率和空间需求，能够进行有序性相关的操作	插入操作很慢
二叉查找树	BST	实现简单，能够进行有序性相关的操作	没有性能上界的保证 链接需要额外的空间
平衡二叉查找树	RedBlackBST	最优的查找和插入效率，能够进行有序性相关的操作	链接需要额外的空间
散列表	SeparateChainHashST LinearProbingHashST	能够快速地查找和插入常见类型的数据	需要计算每种类型的数据的散列无法进行有序性相关的操作链接和空结点需要额外的空间

在学习中我们会仔细了解每种算法和实现的各种性质，这里的简单特性是为了帮助你在学习它们的同时能够从全局的高度来理解它们。一句话，我们有若干种高效的符号表实现，它们能够并且已经被应用于无数程序之中了。

386

答疑

问 为什么符号表不像 2.4 节中优先队列那样使用一个 Comparable 的 Item 类型，而是对于键和值使用不同的数据类型？

答 这的确是一种可行的办法。这两者代表了将键和值关联起来的两种不同方式——我们可以构造一种将键包含在其中的数据结构来隐式关联键值或是显式地将键和值区分开来。对于符号表，我们选择突出关联数组的抽象形式。同时也请注意，符号表的用例在查找时只会指定一个键，而非一个键值对。

问 为什么要用 equals()？为什么不一直使用 compareTo()？

答 并不是所有的数据产生的键值对都能够进行比较，尽管有时候将它们保存在符号表可以。举一个比较极端的例子，你可能会用一幅照片或者一首歌作为键，但没法比较它们，只能知道它们是否相等（也要花点儿工夫）。

问 为什么键的值不能为空（null）？

答 因为我们会用 Key 调用 compareTo() 或者 equals() 方法，因此我们假设它是一个 Object。但是当 a 为 null 时 a.compareTo(b) 会抛出一个空指针异常。如果能消除这种可能性，用例的代码能够更简单。

问 为什么不和排序一样使用一个类似于 less() 的方法？

答 在符号表中等价性比较特殊，因此我们还需要一个方法来测试等价性。为了避免增加本质上功能相同的方法，我们使用了 Java 内置的 equals() 和 compareTo()。

问 在 BinarySearchST 中的类型转换之前，为什么不将 key[] 和 val[] 一样声明为 Object[]（而是 Comparable[]）？

答 问得好。如果你这么做，你会得到一个 ClassCastException，因为键只能是 Comparable 的（以保证 key[] 中的元素都有 compareTo() 方法）。因此将 key[] 声明为 Comparable[] 是必需的。深
387 入程序语言的设计细节来解释这里的原因可能会有些跑题。在本书所有使用泛型的 Comparable 对象和数组的代码中我们都会照此办理。

问 如果我们需要将多个值关联到同一个键怎么办？例如，如果我们在应用程序中用 Date 日期作为键，那不会需要处理重复的键吗？

答 可能会，也可能不会。例如，两列火车不可能同时在同一条轨道上到达同一个车站（但它们可以在不同的铁轨上同时到站）。处理这种情形有两个办法：用其他信息来消除重复或者使用 Queue 类型来存储所有有相同键的值。我们会在 3.5 节中详细讨论符号表的应用。

问 3.1.7 节中将表预排序的想法看起来是个好主意，为什么把它留作一道练习（请见练习 3.1.12）？

答 的确，在某些应用中它确实是最佳的选择。但在一个希望实现快速查找的数据结构中为了“图方便”而加入一个低效的插入方法会变成一个性能陷阱，因为一个普通用例可能会在一张很大的表中混合使用查找和插入操作却发现运行所需的时间是平方级别的。这种陷阱太常见了，因此当你使用他人开发的软件，尤其是接口繁多时，你应该加倍小心。当对象含有大量“便捷”方法而导致到处都是性能陷阱，而用例却可能认为所有的方法都同样高效时，这个问题就非常严重了。Java 的
388 ArrayList 类就是这样的一个例子（请见练习 3.5.27）。

练习

3.1.1 编写一段程序，创建一张符号表并建立字母成绩和数值分数的对应关系，如下表所示。从标准输入读取一系列字母成绩，计算并打印 GPA（字母成绩对应的分数的平均值）。

A+	A	A–	B+	B	B–	C+	C	C–	D	F
4.33	4.00	3.67	3.33	3.00	2.67	2.33	2.00	1.67	1.00	0.00

3.1.2 开发一个符号表的实现 ArrayST，使用（无序）数组来实现我们的基本 API。

3.1.3 开发一个符号表的实现 OrderedSequentialSearchST，使用有序链表来实现我们的有序符号表 API。

3.1.4 开发抽象数据类型 Time 和 Event 来处理表 3.1.5 中的例子中的数据。

3.1.5 实现 SequentialSearchST 中的 size()、delete() 和 keys() 方法。

3.1.6 用输入中的单词总数 W 和不同单词总数 D 的函数给出 FrequencyCounter 调用的 put() 和 get() 方法的次数。

3.1.7 对于 N=10、10^2、10^3、10^4、10^5 和 10^6，在 N 个小于 1000 的随机非负整数中 Frequency Counter 平均能够找到多少个不同的键？

3.1.8 在《双城记》中，使用频率最高的长度大于等于 10 的单词是什么？

3.1.9 在 FrequencyCounter 中添加追踪 put() 方法的最后一次调用的代码。打印出最后插入的那个单词以及在此之前总共从输入中处理了多少个单词。用你的程序处理 tale.txt 中长度分别大于等于 1、8 和 10 的单词。

3.1.10 给出用 SequentialSearchST 将键 E A S Y Q U E S T I O N 插入一个空符号表的过程的轨迹。一共进行了多少次比较？

3.1.11 给出用 BinarySearchST 将键 E A S Y Q U E S T I O N 插入一个空符号表的过程的轨迹。一

共进行了多少次比较?

3.1.12 修改 BinarySearchST，用一个 Item 对象的数组而非两个平行数组来保存键和值。添加一个构造函数，接受一个 Item 的数组为参数并将其归并排序。 389

3.1.13 对于一个会随机混合进行 10^3 次 put() 和 10^6 次 get() 操作的应用程序，你会使用本节中的哪种符号表的实现？说明理由。

3.1.14 对于一个会随机混合进行 10^6 次 put() 和 10^3 次 get() 操作的应用程序，你会使用本节中的哪种符号表的实现？说明理由。

3.1.15 假设在一个 BinarySearchST 的用例程序中，查找操作的次数是插入操作的 1000 倍。当分别进行 10^3、10^6 和 10^9 次查找时，请估计插入操作在总耗时中的比例。

3.1.16 为 BinarySearchST 实现 delete() 方法。

3.1.17 为 BinarySearchST 实现 floor() 方法。

3.1.18 证明 BinarySearchST 中 rank() 方法的实现的正确性。

3.1.19 修改 FrequencyCounter，打印出现频率最高的所有单词，而非其中之一。*提示*：请用 Queue。

3.1.20 补全命题 B 的证明（证明 N 的一般情况）。*提示*：先证明 $C(N)$ 的单调性，即对于所有的 $N>0$，$C(N) \leqslant C(N+1)$ 390

提高题

3.1.21 *内存使用*。基于 1.4 节中的假设，对于 N 对键值比较 BinarySearchST 和 SequentialSearchST 的内存使用情况。不需要记录键值本身占用的内存，只统计它们的引用。对于 BinarySearchST，假设数组大小可以动态调整，数组中被占用的空间比例为 25% ～ 100%。

3.1.22 *自组织查找*。自组织查找指的是一种能够将数组元素重新排序使得被访问频率较高的元素更容易被找到的查找算法。请修改你为练习 3.1.2 给出的答案，在每次查找命中时：将被找到的键值对移动到数组的开头，将所有中间的键值对向右移动一格。这个启发式的过程被称为*前移编码*。

3.1.23 *二分查找的分析*。请证明对于大小为 N 的符号表，一次二分查找所需的最大比较次数正好是 N 的二进制表示的位数，因为右移一位的操作会将二进制的 N 变为二进制的 $[N/2]$。

3.1.24 *插值法查找*。假设符号表的键支持算术操作（例如，它们可能是 Double 或者 Interger 类型的值）。编写一个二分查找来模拟查字典的行为，例如当单词的首字母在字母表的开头时我们也会在字典的前半部分进行查找。具体来说，设 k_{lo} 为符号表的第一个键，k_{hi} 为符号表的最后一个键，当要查找 k_x 时，先和 $\lfloor (k_x-k_{lo})/(k_{hi}-k_{lo}) \rfloor$ 进行比较，而非取中间元素。用 SearchCompare[①] 调用 FrequencyCounter 来比较你的实现和 BinarySearchST 的性能。

3.1.25 *缓存*。因为默认的 contains() 的实现中调用了 get()，所以 FrequencyCounter 的内循环会将同一个键查找两三遍：

```
if (!st.contains(word)) st.put(word, 1);
else                    st.put(word, st.get(word) + 1);
```

为了能够提高这样的用例代码的效率，我们可以用一种叫*缓存*的技术手段，即将访问最频繁的键的位置保存在一个变量中。修改 SequentialSearchST 和 BinarySearchST 来实现这个点子。 391

① SearchCompare应该是一个类似于SortCompare的类，但实际上正文中并没有任何关于这个SearchCompare类的内容。——译者注

3.1.26 基于字典的频率统计。修改 FrequencyCounter，接受一个字典文件作为参数，统计标准输入中出现在字典中的单词的频率，并将单词和频率打印为两张表格，一张按照频率高低排序，一张按照字典顺序排序。

3.1.27 小符号表。假设一段 BinarySearchST 的用例插入了 N 个不同的键并会进行 S 次查找。当构造表的成本和所有查找的总成本相同时，给出 S 的增长数量级。

3.1.28 有序的插入。修改 BinarySearchST，使得插入一个比当前所有键都大的键只需要常数时间（这样在构造符号表时有序地使用 put() 插入键值对就只需要线性时间了）

3.1.29 测试用例。编写一段测试代码 TestBinarySearch.java 用来测试正文中 min()、max()、floor()、ceiling()、select()、rank()、deleteMin()、deleteMax() 和 keys() 的实现。可以参考 3.1.3.1 节的索引用例，添加代码使其在适当的情况下接受更多的命令行参数。

3.1.30 验证。向 BinarySearchST 中加入断言（assert）语句，在每次插入和删除数据后检查算法的有效性和数据结构的完整性。例如，对于每个索引必有 i==rank(select(i)) 且数组应该总是有序的。

392

实验题

3.1.31 性能测试。编写一段性能测试程序，先用 put() 构造一张符号表，再用 get() 进行访问，使得表中的每个键平均被命中 10 次，且有大致相同次数的未命中访问。键为长度从 2 到 50 不等的随机字符串。重复这样的测试若干遍，记录每遍的运行时间，打印平均运行时间或将它们绘制成图。

3.1.32 练习。编写一段练习程序，用困难或者极端的但在实际应用中可能出现的情况来测试我们的有序符号表 API。一些简单的例子包括有序的键列、逆序的键列、所有键全部相同或者只含有两种不同的值。

3.1.33 自组织查找。编写一段程序调用自组织查找的实现（请见练习 3.1.22），用 put() 构造一个大小为 N 的符号表，然后根据预先定义好的概率分布进行 $10N$ 次命中查找。对于 $N=10^3$、10^4、10^5 和 10^6，用这段程序比较你在练习 3.1.22 中的实现和 BinarySearchST 的运行时间，在预定义的概率分布中查找命中第 i 小的键的概率为 $1/2^i$。

3.1.34 Zipf 法则。用命中第 i 小的键的概率为 $1/(iH_N)$ 的分布重新完成上一道练习，其中 H_N 为调和级数（请见表 1.4.6）。这种分布被称为 Zipf 法则。比较前移编码和上一道练习中的在特定分布下的最优安排，该安排将所有键按升序排列（即按照它们的期望频率的降序排列）。

3.1.35 性能验证 I。用各种不同的 N 运行双倍测试，取《双城记》的前 N 个单词，验证 FrequencyCounter 在使用 SequentialSearchST 时所需的运行时间是 N 的平方级别的猜想。

3.1.36 性能验证 II。解释 FrequencyCounter 在使用 BinarySearchST 时比使用 SequentialSearchST 时的性能提高程度好于预期的原因。

3.1.37 put/get 的比例。当 FrequencyCounter 使用 BinarySearchST 在 100 万个长度为 M 位的随机整数中统计每个值的出现频率时，根据经验判断 BinarySearchST 中 put() 操作和 get() 操作的耗时比，其中 M=10、20 和 30。再统计 tale.txt 并评估耗时比，并比较两次的结果。

393

3.1.38 均摊成本图。修改 FrequencyCounter、SequentialSearchST 和 BinarySearchST，统计计算中每次 put() 操作的成本并生成类似本节所示的图。

3.1.39 *实际耗时。*修改 FrequencyCounter，用 Stopwatch 和 StdDraw 绘图，其中 x 轴为 get() 和 put() 的调用次数之和，y 轴为总运行时间，每次调用时就根据已运行时间画一个点。分别用 SequentialSearchST 和 BinarySearchST 处理《双城记》并讨论运行的结果。*注意*：曲线中突然的跳跃可能是缓存导致的，这已经超出了这个问题的讨论范围。

3.1.40 *二分查找的临界点。*找出使用二分查找比顺序查找要快 10 000 倍和 1000 倍的 N 值。分析并预测 N 的大小并通过实验验证它。

3.1.41 *插值查找的临界点。*找出使用插值查找比二分查找要快 1 倍、2 倍和 10 倍的 N 值，其中假设所有键为随机的 32 位整数（请见练习 3.1.24）。分析并预测 N 的大小并通过实验验证它。 394

3.2 二叉查找树

在本节中我们将学习一种能够将链表插入的灵活性和有序数组查找的高效性结合起来的符号表实现。具体来说，就是使用每个结点含有两个链接（链表中每个结点只含有一个链接）的二叉查找树来高效地实现符号表，这也是计算机科学中最重要的算法之一。

首先，我们需要定义一些术语。我们所使用的数据结构由结点组成，结点包含的链接可以指向空（null）或者其他结点。在二叉树中，每个结点只能有一个父结点指向自己（只有一个例外，也就是根结点，它没有父结点），而且每个结点都只有左右两个链接，分别指向自己的左子结点和右子结点（如图 3.2.1 所示）。尽管链接指向的是结点，但我们可以将每个链接看做指向了另一棵二叉树，而这棵树的根结点就是被指向的结点。因此我们可以将二叉树定义为一个空链接，或者是一个有左右两个链接的结点，每个链接都指向一棵（独立的）子二叉树。在二叉查找树中，每个结点还包含了一个键和一个值，键之间也有顺序之分以支持高效的查找。

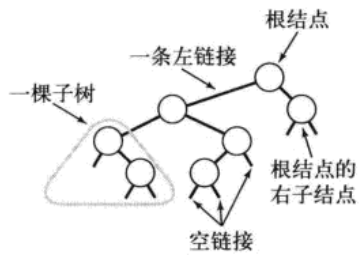


图 3.2.1 详解二叉树

定义。一棵二叉查找树（BST）是一棵二叉树，其中每个结点都含有一个 Comparable 的键（以及相关联的值）且每个结点的键都大于其左子树中的任意结点的键而小于右子树的任意结点的键。

我们在画出二叉查找树时会将键写在结点上。我们使用“A 是 E 的左子结点”的说法用键指代结点。我们用连接结点的线表示链接，并将键对应的值写在结点旁边（若值不确定则省略）。除了空结点只表示为向下的一条线段以外，每个结点的链接都指向它下方的结点。和以前一样，我们在例子中只会使用索引测试用例生成的单个字母作为键，如图 3.2.2 所示。

395
~
396

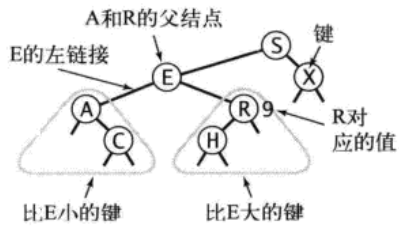


图 3.2.2 详解二叉查找树

3.2.1 基本实现

算法 3.3 定义了二叉查找树（BST）的数据结构，我们会在本节中用它实现有序符号表的 API。首先我们要研究一下这个经典的数据类型，以及与它的特点紧密相关的 get()（查找）和 put()（插入）方法的实现。

3.2.1.1 数据表示

和链表一样，我们嵌套定义了一个私有类来表示二叉查找树上的一个结点。每个结点都含有一个键、一个值、一条左链接、一条右链接和一个结点计数器（有需要时我们会在图中将结点计数器的值写在结点上方）。左链接指向一棵由小于该结点的所有键组成的二叉查找树，右链接指向一棵由大于该结点的所有键组成的二叉查找树。变量 N 给出了以该结点为根的子树的结点总数。你将会看到，它简化了许多有序符号表的操作的实现。算法 3.3 中实现的私有方法 size() 会将空链接的值当作 0，这样我们就能保证以下公式对于二叉树中的任意结点 x 总是成立。

```
size(x) = size(x.left) + size(x.right) + 1
```

一棵二叉查找树代表了一组键（及其相应的值）的集合，而同一个集合可以用多棵不同的二叉查找树表示（如图 3.2.3 所示）。如果我们将一棵二叉查找树的所有键投影到一条直线上，保证一个结点的左子树中的键出现在它的左边，右子树中的键出现在它的右边，那么我们一定可以得到一条有序的键列。我们会利用二叉查找树的这种天生的灵活性，用多棵二叉查找树表示同一组有序的键来实现构建和使用二叉查找树的高效算法。

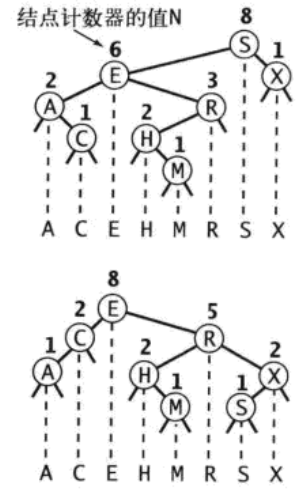


图 3.2.3 两棵能够表示同一组键的二叉查找树

3.2.1.2 查找

一般来说，在符号表中查找一个键可能得到两种结果。如果含有该键的结点存在于表中，我们的查找就命中了，然后返回相应的值。否则查找未命中（并返回 `null`）。根据数据表示的递归结构我们马上就能得到，在二叉查找树中查找一个键的递归算法：如果树是空的，则查找未命中；如果被查找的键和根结点的键相等，查找命中，否则我们就（递归地）在适当的子树中继续查找。如果被查找的键较小就选择左子树，较大则选择右子树。算法 3.3（续 1）中递归的 `get()` 方法完全实现了这段算法。它的第一个参数是一个结点（子树的根结点），第二个参数是被查找的键。代码会保证只有该结点所表示的子树才会含有和被查找的键相等的结点。和二分查找中每次迭代之后查找的区间就会减半一样，在二叉查找树中，随着我们不断向下查找，当前结点所表示的子树的大小也在减小（理想情况下是减半，但至少会有一个结点）。当找到一个含有被查找的键的结点（命中）或者当前子树变为空（未命中）时这个过程才会结束。从根结点开始，在每个结点中查找的进程都会递归地在它的一个子结点上展开，因此一次查找也就定义了树的一条路径。对于命中的查找，路径在含有被查找的键的结点处结束。对于未命中的查找，路径的终点是一个空链接，如图 3.2.4 所示。

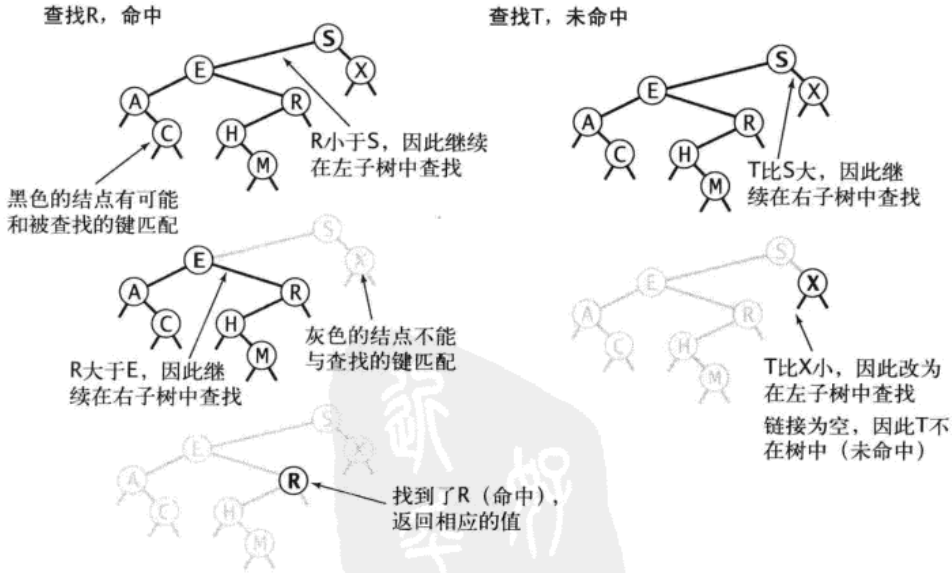


图 3.2.4 二叉查找树中的查找命中（左）和未命中（右）

算法 3.3 基于二叉查找树的符号表

```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;                  // 二叉查找树的根结点

   private class Node
   {
      private Key key;                 // 键
      private Value val;               // 值
      private Node left, right;        // 指向子树的链接
      private int N;                   // 以该结点为根的子树中的结点总数

      public Node(Key key, Value val, int N)
      {  this.key = key; this.val = val; this.N = N; }
   }

   public int size()
   {  return size(root);  }

   private int size(Node x)
   {
      if (x == null) return 0;
      else           return x.N;
   }

   public Value get(Key key)
   // 请见算法3.3（续1）

   public void put(Key key, Value val)
   // 请见算法3.3（续1）

   // max()、min()、floor()、ceiling()方法请见算法3.3（续2）
   // select()、rank()方法请见算法3.3（续3）
   // delete()、deleteMin()、deleteMax()方法请见算法3.3（续4）
   // keys()方法请见算法3.3（续5）

}
```

这段代码用二叉查找树实现了有序符号表的 API，树由 Node 对象组成，每个对象都含有一对键值、两条链接和一个结点计数器 N。每个 Node 对象都是一棵含有 N 个结点的子树的根结点，它的左链接指向一棵由小于该结点的所有键组成的二叉查找树，右链接指向一棵由大于该结点的所有键组成的二叉查找树。root 变量指向二叉查找树的根结点 Node 对象（这棵树包含了符号表中的所有键值对）。本节会陆续给出其他方法的实现。

算法 3.3（续 1）的实现过程如下所示。

算法 3.3（续 1） 二叉查找树的查找和排序方法的实现

```
public Value get(Key key)
{  return get(root, key);  }
private Value get(Node x, Key key)
{  // 在以x为根结点的子树中查找并返回key所对应的值;
   // 如果找不到则返回null
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
```

```
    if      (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
}

public void put(Key key, Value val)
{  // 查找key，找到则更新它的值，否则为它创建一个新的结点
   root = put(root, key, val);
}

private Node put(Node x, Key key, Value val)
{
   // 如果key存在于以x为根结点的子树中则更新它的值；
   // 否则将以key和val为键值对的新结点插入到该子树中
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = put(x.left,  key, val);
   else if (cmp > 0) x.right = put(x.right, key, val);
   else x.val = val;
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

这段代码实现了有序符号表 API 中的 put() 和 get() 方法，它们的递归实现也是本章稍后将会讨论的其他几种实现的模板。每个方法的实现既可以看做是实用的代码，也可以看做是之前讨论的递推猜想的证明。

3.2.1.3 插入

算法 3.3（续 1）中的查找代码几乎和二分查找的一样简单，这种简洁性是二叉查找树的重要特性之一。而二叉查找树的另一个更重要的特性就是插入的实现难度和查找差不多。当查找一个不存在于树中的结点并结束于一条空链接时，我们需要做的就是将链接指向一个含有被查找的键的新结点（详见图 3.2.5）。算法 3.3（续 1）中递归的 put() 方法的实现逻辑和递归查找很相似：如果树是空的，就返回一个含有该键值对的新结点；如果被查找的键小于根结点的键，我们会继续在左子树中插入该键，否则在右子树中插入该键。

3.2.1.4 递归

这些递归实现值得我们花点儿时间去理解其中的运行细节。可以将递归调用前的代码想象成沿着树向下走：它会将给定的键和每个结点的键相比较并根据结果向左或者向右移动到下一个结点。然后可以将递归调用后的代码想象成沿着树向上爬。对于 get() 方法，这对应着一系列的返回指令（return），但是对于 put() 方法，这意味着重置搜索路径上每个父结点指向子结点的链接，并增加路径上每个结点中的计数器的值。在一棵简单的二叉查找树中，唯一的新链接就是在最底层指向新结点的链接，重置更上层的链接

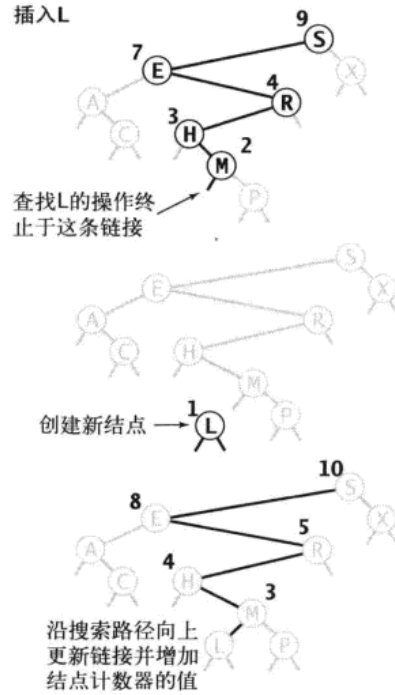
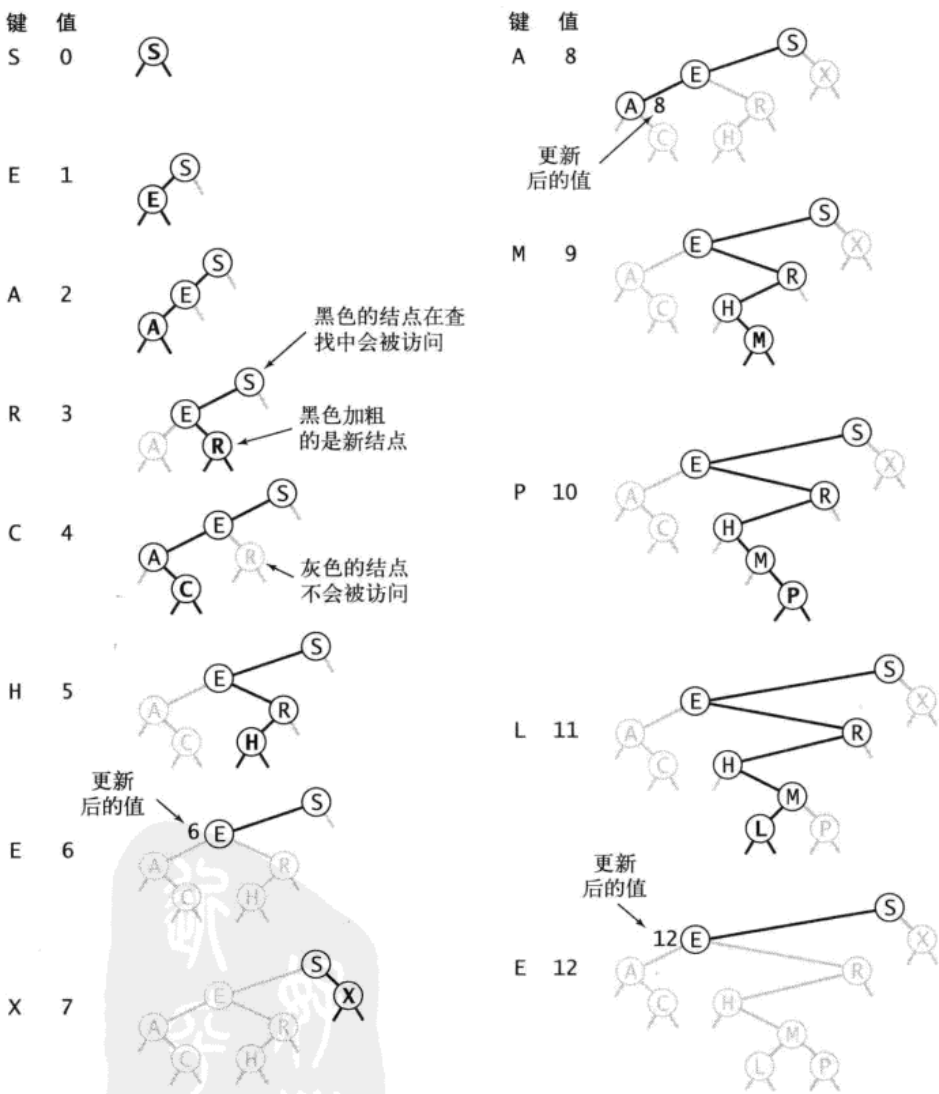


图 3.2.5 二叉查找树的插入操作

397 ≀ 400

可以通过比较语句来避免。同样，我们只需要将路径上每个结点中的计数器的值加 1，但我们使用了更加通用的代码，使之等于结点的所有子结点的计数器之和加 1。在本节和下一节中，我们会学习一些更加高级但原理相同的算法，但它们在搜索路径上需要改变的链接更多，也需要适应性更强的代码来更新结点计数器。基本的二叉查找树的实现常常是非递归的（请见练习 3.2.12）——我们在实现中使用了递归，一来是为了便于读者理解代码的工作方式，二来也是为学习更加复杂的算法做准备。

图 3.2.6 是对我们的标准索引用例轨迹的一份详细的研究，它向你展示了二叉树是如何生长的。新结点会连接到树底层的空链接上，树的其他部分则不会改变。例如，第一个被插入的键就是根结点，第二个被插入的键是根结点的两个子结点之一，以此类推。因为每个结点都含有两个链接，树会逐渐长大而不是萎缩。不仅如此，因为只有查找或者插入路径上的结点才会被访问，所以随着树的增长，
401 被访问的结点数量占树的总结点数的比例也会不断的降低。



402 图 3.2.6 使用二叉查找树的标准索引用例的轨迹

3.2.2 分析

使用二叉查找树的算法的运行时间取决于树的形状，而树的形状又取决于键被插入的先后顺序。在最好的情况下，一棵含有 N 个结点的树是完全平衡的，每条空链接和根结点的距离都为～ lg$N$。在最坏的情况下，搜索路径上可能有 N 个结点。如图 3.2.7 所示。但在一般情况下树的形状和最好情况更接近。

对于很多应用来说，图 3.2.8 所示的简单模型都是适用的：我们假设键的分布是（均匀）随机的，或者说它们的插入顺序是随机的。对这个模型的分析而言，二叉查找树和快速排序几乎就是“双胞胎”。树的根结点就是快速排序中的第一个切分元素（左侧的键都比它小，右侧的键都比它大），而这对于所有的子树同样适用，这和快速排序中对子数组的递归排序完全对应。这使我们能够分析得到二叉查找树的一些性质。

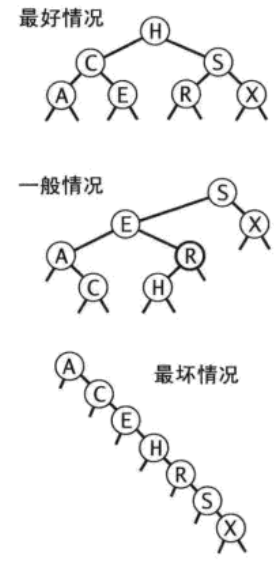


图 3.2.7 二叉查找树的可能形状

命题 C。 在由 N 个随机键构造的二叉查找树中，查找命中平均所需的比较次数为～ $2\ln N$（约 $1.39\lg N$）。

证明。 一次结束于给定结点的命中查找所需的比较次数为查找路径的深度加 1。如果将树中的所有结点的深度加起来，我们就能够得到一棵树的**内部路径长度**。因此，在二叉查找树中的平均比较次数即为平均内部路径长度加 1。我们可以使用 2.3 节的命题 K 的证明得到它：令 C_N 为由 N 个随机排序的不同键构造得到的二叉查找树的内部路径长度，则查找命中的平均成本为 $(1+C_N/N)$。我们有 $C_0=C_1=0$，且对于 $N>1$ 我们可以根据二叉查找树的递归结构直接得到一个归纳关系式：

$$C_N=N-1+(C_0+C_{N-1})/N+(C_1+C_{N-2})/N+\cdots+(C_{N-1}+C_0)/N$$

其中 $N-1$ 这一项表示根结点使得树中的所有 $N-1$ 个非根结点的路径上都加了 1。表达式的其他项代表了所有子树，它们的计算方法和大小为 N 的二叉查找树的方法相同。整理表达式后我们会发现，这个归纳公式和我们在 2.3 节中为快速排序得到的公式几乎完全相同，因此我们同样可以得到 $C_N\sim 2N\ln N$。

命题 D。 在由 N 个随机键构造的二叉查找树中插入操作和查找未命中平均所需的比较次数为～ $2\ln N$（约 $1.39\lg N$）。

证明。 插入操作和查找未命中平均比查找命中需要一次额外的比较。这一点由归纳法不难得到（请见练习 3.2.16）。

403

命题 C 说明在二叉查找树中查找随机键的成本比二分查找高约 39%。命题 D 说明这些额外的成本是值得的，因为插入一个新键的成本是对数级别的——这是基于二分查找的有序数组所不具备的灵活性，因为它的插入操作所需访问数组的次数是线性级别的。和快速排序一样，比较次数的标准差很小，因此 N 越大这个公式越准确。

实验

我们的随机键模型和典型的符号表使用情况是否相符？按照惯例，这个问题的答案需要具体问题具体分析，因为在不同的应用场景中性能的差别可能很大。幸好，对于大多数用例，这个模型都能很好地适应。

作为例子，我们研究用 FrequencyCounter 处理长度大于等于 8 的单词时 put() 操作的成本。从图 3.2.9 可以看到，每次操作的平均成本从 BinarySearchST 的 484 次数组访问降低到了二叉查找树的 13 次，这也再次验证了理论模型所预测的对数级别的性能。根据命题 C 和命题 D，这个数值的合理大小应该是符号表大小的自然对数的两倍左右，因为对于一个几乎充满的符号表，大多数操作都是查找。这个预测至少有以下不准确性：

- 很多操作都是在较小的符号表中进行的；
- 键不随机；
- 符号表可能太小，近似值 $2\ln N$ 不准确。

无论如何，通过表 3.2.1 你都能看到，对于 FrequencyCounter 这个预测的误差只有若干次比较。事实上，大多数误差都能通过对近似值的数学表达式的改进得到解释（请见练习 3.2.35）。

404

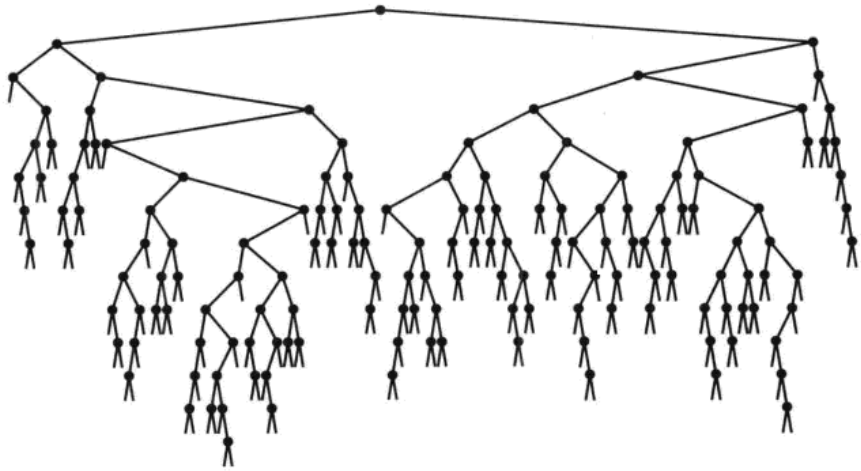


图 3.2.8 一棵典型的二叉查找树，由 256 个随机键组成

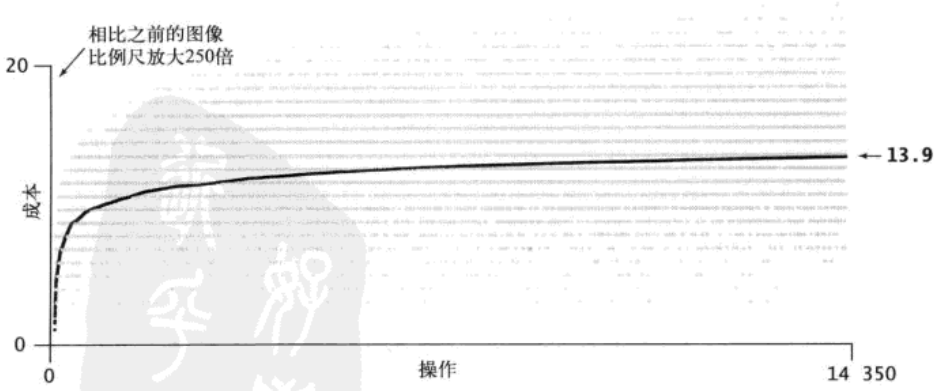


图 3.2.9 使用二叉查找树，运行 java FrequencyCounter 8 < tale.txt 的成本

表 3.2.1 使用二叉查找树的 FrequencyCounter 的每次 put() 操作平均所需的比较次数

	tale.txt				leipzig1M.txt			
	单词数	不同单词数	比较次数		单词数	不同单词数	比较次数	
			模型预测	实际次数			模型预测	实际次数
所有单词	135 635	10 679	18.6	17.5	21 1914 55	534 580	23.4	22.1
长度大于等于 8 的单词	14 350	5 737	17.6	13.9	4 239 597	299 593	22.7	21.4
长度大于等于 10 的单词	4 582	2 260	15.4	13.1	1 610 829	165 555	20.5	19.3

405

3.2.3 有序性相关的方法与删除操作

二叉查找树得以广泛应用的一个重要原因就是它能够保持键的有序性，因此它可以作为实现有序符号表 API（请见 3.1.2 节）中的众多方法的基础。这使得符号表的用例不仅能够通过键还能通过键的相对顺序来访问键值对。下面，我们要研究有序符号表 API 中各个方法的实现。

3.2.3.1 最大键和最小键

如果根结点的左链接为空，那么一棵二叉查找树中最小的键就是根结点；如果左链接非空，那么树中的最小键就是左子树中的最小键。这不仅描述了算法 3.3（续 2）中 min() 方法的递归实现，同时也递推地证明了它能够在二叉查找树中找到最小的键。简单的循环也能等价实现这段描述，但为了保持一致性我们使用了递归。我们可以让递归调用返回键 Key 而非结点对象 Node，但我们后面还会用到这方法来找出含有最小键的结点。找出最大键的方法也是类似的，只是变为查找右子树而已。

3.2.3.2 向上取整和向下取整

如果给定的键 key 小于二叉查找树的根结点的键，那么小于等于 key 的最大键（floor）一定在根结点的左子树中；如果给定的键 key 大于二叉查找树的根结点，那么只有当根结点右子树中存在小于等于 key 的结点时，小于等于 key 的最大键才会出现在右子树中，否则根结点就是小于等于 key 的最大键。这段描述说明了 floor() 方法的递归实现，同时也递推地证明了它能够计算出预期的结果。将“左”变为“右”（同时将小于变为大于）就能够得到 ceiling() 的算法。向上取整函数的计算如图 3.2.10 所示。

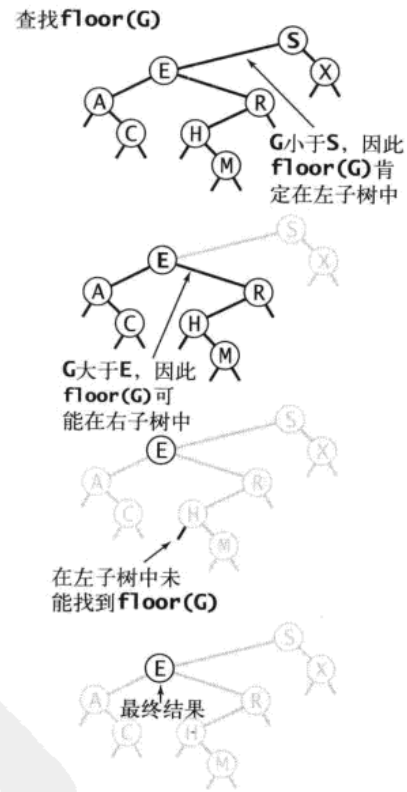


图 3.2.10 计算 floor() 函数

3.2.3.3 选择操作

二叉查找树中的选择操作和 2.5 节中我们学习过的基于切分的数组选择操作类似。我们在二叉查找树的每个结点中维护的子树结点计数器变量 N 就是用来支持此操作的。

406

算法 3.3（续 2） 二叉查找树中 max()、min()、floor()、ceiling() 方法的实现

```
public Key min()
{
   return min(root).key;
}
private Node min(Node x)
{
   if (x.left == null) return x;
   return min(x.left);
}
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if (cmp == 0) return x;
   if (cmp < 0)  return floor(x.left, key);
   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;
}
```

每个公有方法都对应着一个私有方法，它接受一个额外的链接作为参数指向某个结点，通过正文中描述的递归方法查找返回 null 或者含有指定 Key 的结点 Node。max() 和 ceiling() 的实现分别与 min() 和 floor() 方法基本相同，只是将代码中的 left 和 right（以及≥和≤）调换而已。

407

假设我们想找到排名为 k 的键（即树中正好有 k 个小于它的键）。如果左子树中的结点数 t 大于 k，那么我们就继续（递归地）在左子树中查找排名为 k 的键；如果 t 等于 k，我们就返回根结点中的键；如果 t 小于 k，我们就（递归地）在右子树中查找排名为（k–t–1）的键。和刚才一样，这段描述既说明了 select() 方法的递归实现同时也证明了它的正确性，此过程如图 3.2.11 所示。

3.2.3.4 排名

rank() 是 select() 的逆方法，它会返回给定键的排名。它的实现和 select() 类似：如果给定的键和根结点的键相等，我们返回左子树中的结点总数 t；如果给定的键小于根结点，我们会返回该键在左子树中的排名（递归计算）；如果给定的键大于根结点，我们会返回 t+1（根结点）加上它在右子树中的排名（递归计算）。

二叉查找树中选择和排名操作的实现如算法 3.3（续 3）所示。

算法 3.3（续 3） 二叉查找树中 select() 和 rank() 方法的实现

```
public Key select(int k)
{
   return select(root, k).key;
}
private Node select(Node x, int k)
```

```
{   // 返回排名为k的结点
    if (x == null) return null;
    int t = size(x.left);
    if      (t > k) return select(x.left,  k);
    else if (t < k) return select(x.right, k-t-1);
    else            return x;
}
public int rank(Key key)
{  return rank(key, root);  }
private int rank(Key key, Node x)
{  // 返回以x为根结点的子树中小于x.key的键的数量
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) return rank(key, x.left);
   else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
   else              return size(x.left);
}
```

这段代码使用了和我们已经在本章中学习过的其他实现中一样的递归模式实现了 select() 和 rank() 方法。它依赖于本节开始处给出的 size() 方法来统计每个结点以下的子结点总数。

3.2.3.5 删除最大键和删除最小键

二叉查找树中最难实现的方法就是 delete() 方法，即从符号表中删除一个键值对。作为热身运动，我们先考虑 deleteMin() 方法（删除最小键所对应的键值对），如图 3.2.12 所示。和 put() 一样，我们的递归方法接受一个指向结点的链接，并返回一个指向结点的链接。这样我们就能够方便地改变树的结构，将返回的链接赋给作为参数的链接。对于 deleteMin()，我们要不断深入根结点的左子树中直至遇见一个空链接，然后将指向该结点的链接指向该结点的右子树（只需要在递归调用中返回它的右链接即可）。此时已经没有任何链接指向要被删除的结点，因此它会被垃圾收集器清理掉。我们给出的标准递归代码在删除结点后会正确地设置它的父结点的链接并更新它到根结点的路径上的所有结点的计数器的值。deleteMax() 方法的实现和 deleteMin() 完全类似。

3.2.3.6 删除操作

我们可以用类似的方式删除任意只有一个子结点（或者没有子结点）的结点，但应该怎样删除一个拥有两个子结点的结点呢？删除之后我们要处理两棵子树，但被删除结点的父结点只有一条空出来的链接。T. Hibbard 在 1962 年提出了解决这个难题的第一个方法，在删除结点 x 后用它的后继结点填



图 3.2.11 二叉查找树中的 select() 操作

补它的位置。因为 x 有一个右子结点，因此它的后继结点就是其右子树中的最小结点。这样的替换仍然能够保证树的有序性，因为 x.key 和它的后继结点的键之间不存在其他的键。我们能够用 4 个简单的步骤完成将 x 替换为它的后继结点的任务（具体过程如图 3.2.13 所示）：

- 将指向即将被删除的结点的链接保存为 t；
- 将 x 指向它的后继结点 min(t.right)；
- 将 x 的右链接（原本指向一棵所有结点都大于 x.key 的二叉查找树）指向 deleteMin(t.right)，也就是在删除后所有结点仍然都大于 x.key 的子二叉查找树；
- 将 x 的左链接（本为空）设为 t.left（其下所有的键都小于被删除的结点和它的后继结点）。

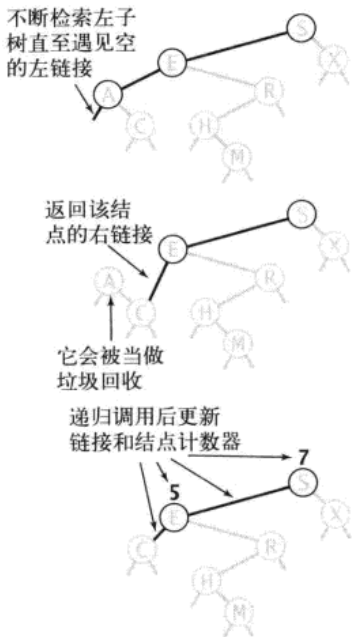


图 3.2.12 删除二叉查找树中的最小结点

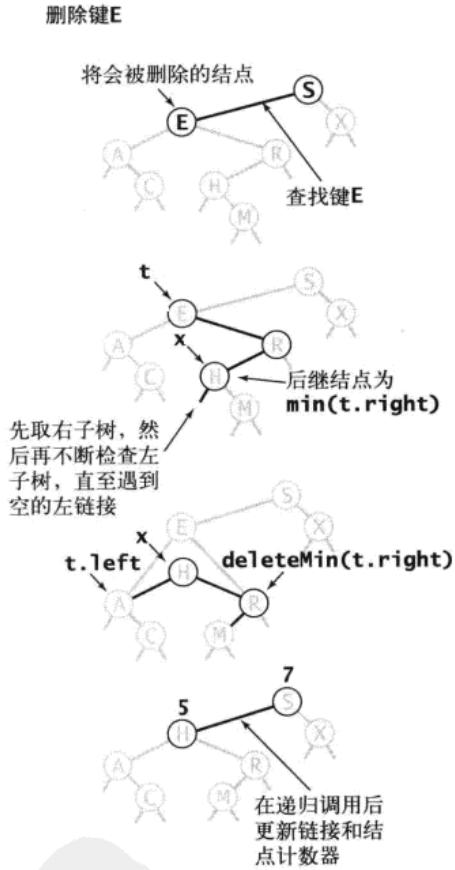


图 3.2.13 二叉查找树中的删除操作

在递归调用后我们会修正被删除的结点的父结点的链接，并将由此结点到根结点的路径上的所有结点的计数器减 1（这里计数器的值仍然会被设为其所有子树中的结点总数加一）。尽管这种方法能够正确地删除一个结点，它的一个缺陷是可能会在某些实际应用中产生性能问题。这个问题在于选用后继结点是一个随意的决定，且没有考虑树的对称性。可以使用它的前继结点吗？实际上，前继结点和后继结点的选择应该是随机的。详细讨论请见练习 3.2.42。

408 ≀ 410

二叉查找树中删除操作的实现如算法 3.3（续 4）所示。

算法 3.3（续 4） 二叉查找树的 delete() 方法的实现

```
public void deleteMin()
{
   root = deleteMin(root);
}

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}

public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else
   {
      if (x.right == null) return x.left;
      if (x.left == null) return x.right;
      Node t = x;
      x = min(t.right);  // 请见算法3.3（续2）
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

如前文所述，这段代码实现了 Hibbard 的二叉查找树中对结点的即时删除。delete() 方法的代码很简洁，但不简单。也许理解它的最好办法就是读懂正文中的讲解，试着自己实现它并对比自己的代码和这段代码。一般情况下这段代码的效率不错，但对于大规模的应用来说可能会有一点问题（请见练习 3.2.42）。deleteMax() 的实现和 deleteMin() 类似，只需将左改为右即可。 411

3.2.3.7 范围查找

要实现能够返回给定范围内键的 keys() 方法，我们首先需要一个遍历二叉查找树的基本方法，叫做*中序遍历*。要说明这个方法，我们先看看如何能够将二叉查找树中的所有键按照顺序打印出来。要做到这一点，我们应该先打印出根结点的左子树中的所有键（根据二叉查找树的定义它们应该都小于根结点的键），然后打印出根结点的键，最后打印出根结点的右子树中的所有键（根据二叉查找树的定义它们应该都大于根结点的键），如右侧的代码所示。

```
private void print(Node x)
{
   if (x == null) return;
   print(x.left);
   StdOut.println(x.key);
   print(x.right);
}
```

按顺序打印二叉查找树中的所有键

和以前一样，刚才的描述也递推地证明了这段代码能够顺序打印树中的所有键。为了实现接受两

个参数并能够将给定范围内的键返回给用例的 keys() 方法，我们可以修改一下这段代码，将所有落在给定范围以内的键加入一个队列 Queue 并跳过那些不可能含有所查找键的子树。和 BinarySearchST 一样，用例不需要知道我们使用 Queue 来收集符合条件的键。我们使用什么数据结构来实现 Iterable<Key> 并不重要，用例只要能够使用 Java 的 foreach 语句遍历返回的所有键就可以了。

二叉查找树的范围查找操作的实现如算法 3.3（续 5）所示。

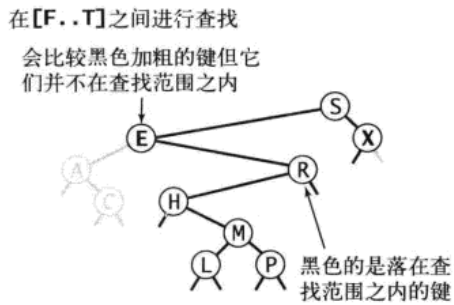
算法 3.3（续 5） 二叉查找树的范围查找操作

```
public Iterable<Key> keys()
{  return keys(min(), max());  }

public Iterable<Key> keys(Key lo, Key hi)
{
    Queue<Key> queue = new Queue<Key>();
    keys(root, queue, lo, hi);
    return queue;
}

private void keys(Node x, Queue<Key> queue, Key lo, Key hi)
{
   if (x == null) return;
   int cmplo = lo.compareTo(x.key);
   int cmphi = hi.compareTo(x.key);
   if (cmplo < 0) keys(x.left, queue, lo, hi);
   if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
   if (cmphi > 0) keys(x.right, queue, lo, hi);
}
```

为了确保以给定结点为根的子树中所有在指定范围之内的键加入队列，我们会（递归地）查找根结点的左子树，然后查找根结点，然后（递归地）查找根结点的右子树。



二叉查找树的范围查找

3.2.3.8 性能分析

二叉查找树中和有序性相关的操作的效率如何？要研究这个问题，我们首先要知道树的高度（即树中任意结点的最大深度）。给定一棵树，树的高度决定了所有操作在最坏情况下的性能（范围查找除外，因为它的额外成本和返回的键的数量成正比）。

命题 E。在一棵二叉查找树中，所有操作在最坏情况下所需的时间都和树的高度成正比。

证明。树的所有操作都沿着树的一条或两条路径行进。根据定义，路径的长度不可能大于树的高度。

我们估计树的高度（即最坏情况下的成本）将会大于我们在 3.2.2 节中定义的平均内部路径长度（这个平均值已经包含了所有较短的路径），但会高多少呢？也许在你看来这个问题和命题 C 和命题 D 解答的问题类似，但它的解答其实要困难得多，完全超出了本书的范畴。1979 年，J. Robson 证明了随机键构造的二叉查找树的平均高度为树中结点数的对数级别，随后 L. Devroye 证明了对于足够大的 N，这个值趋近于 $2.99\lg N$。因此，如果我们的应用中的插入操作能够适用于这个随机模型，我们距离实现一个支持对数级别的所有操作的符号表的目标就已经不远了。我们可以认为随机构造的树中的所有路径长度都小于 $3\lg N$，但如果构造树的键不是随机的怎么办？在下一节中你会看到在实际应用中这个问题其实没有意义，因为还有平衡二叉查找树，它能保证无论键的插入顺序如何，树的高度都将是总键数的对数。

412 ~ 413

总的来说，二叉查找树的实现并不困难，且当树的构造和随机模型近似时在各种实际应用场景中它都能进行快速地查找和插入。对于我们的例子（以及其他许多实际应用场景）来说，二叉查找树将不可能完成的任务变为可能。另外，许多程序员都偏爱基于二叉查找树的符号表的原因是它还支持高效的 `rank()`、`select()`、`delete()` 以及范围查找等操作。但同时，正如我们所强调过的，在某些场景中二叉查找树在最坏情况下的恶劣性能仍然是不可接受的。二叉查找树的基本实现的良好性能依赖于其中的键的分布足够随机以消除长路径。对于快速排序，我们可以先将数组打乱；而对于符号表的 API，我们无能为力，因为符号表的用例控制着各种操作的先后顺序。但最坏情况在实际应用也有可能出现——用例将所有键按照顺序或者逆序插入符号表就会增加这种情况出现的概率，而在没有明确的警告来避免这种行为时有些用例肯定会尝试这么做。这就是我们寻找更好的算法和数据结构的主要原因，这些算法和数据结构我们会在下一节学习。

本书中简单的符号表实现的成本列在表 3.2.2 中。

表 3.2.2 简单的符号表实现的成本总结

算法（数据结构）	最坏情况下的运行时间的增长数量级（N 次插入之后）		平均情况下的运行时间的增长数量级（N 次插入随机键之后）		是否支持有序性相关的操作
	查找	插入	查找命中	插入	
顺序查询（无序链表）	N	N	$N/2$	N	否
二分查找（有序数组）	$\lg N$	N	$\lg N$	$N/2$	是
二叉树查找（二叉查找树）	N	N	$1.39\lg N$	$1.39\lg N$	是

414

答疑

问 我见过二叉查找树，但它的实现没有使用递归。这两种方式各有哪些优缺点？

答 一般来说，递归的实现更容易验证其正确性，而非递归的实现效率更高。在练习 3.2.13 中你需要用另一种方法实现 `get()`，你可能会注意到性能上的改进。如果树不是平衡的，函数调用的栈的深度

可能会成为递归实现的一个问题。我们使用递归的一个主要原因是使读者能够轻松过渡到下一节中的平衡二叉查找树，而且递归版本显然更易于实现和调试。

问 维护 Node 对象中的结点计数器似乎需要很多代码，这有必要吗？为什么不只用一个变量来保存整棵树中的结点总数来实现用例中的 size() 方法？

答 rank() 和 select() 方法需要知道每个结点所代表的子树中的结点总数。如果你不需要实现这些操作，可以去掉这个变量以简化代码（请见练习 3.2.12）。要保证所有结点中的计数器的正确性的确很容易出错，但这个值在调试中同样有用。你也可以用递归的方法实现用例中的 size() 函数，但这样统计所有结点的运行时间可能是线性的。这十分危险，因为如果不知道这么一个简单的操作会如此耗时，用例的性能可能会变得很差。

415

练习

3.2.1 将 E A S Y Q U E S T I O N 作为键按顺序插入一棵初始为空的二叉查找树中（方便起见设第 i 个键对应的值为 i），画出生成的二叉查找树。构造这棵树需要多少次比较？

3.2.2 将 A X C S E R H 作为键按顺序插入将会构造出一棵最坏情况下的二叉查找树结构，最下方的结点的两个链接全部为空，其他结点都含有一个空链接。用这些键给出构造最坏情况下的树的其他 5 种排列。

3.2.3 给出 A X C S E R H 的 5 种能够构造出最优二叉查找树的排列。

3.2.4 假设某棵二叉查找树的所有键均为 1 至 10 的整数，而我们要查找 5。那么以下哪个不可能是键的检查序列？

a. 10, 9, 8, 7, 6, 5

b. 4, 10, 8, 7, 5, 3

c. 1, 10, 2, 9, 3, 8, 4, 7, 6, 5

d. 2, 7, 3, 8, 4, 5

e. 1, 2, 10, 4, 8, 5

3.2.5 假设已知某棵二叉查找树中的每个结点的查找频率，且我们可以以任意顺序用它们构造一棵树。我们是应该按照查找频率的顺序由高到低或是由低到高将它们插入，还是用其他某种顺序？证明你的结论。

3.2.6 为二叉查找树添加一个方法 height() 来计算树的高度。实现两种方案：一种使用递归（用时为线性级别，所需空间和树高成正比），一种模仿 size() 在每个结点中添加一个变量（所需空间为线性级别，查询耗时为常数）。

3.2.7 为二叉查找树添加一个方法 avgCompares() 来计算一棵给定的树中的一次随机命中查找平均所需的比较次数（即树的内部路径长度除以树的大小再加 1）。实现两种方案：一种使用递归（用时为线性级别，所需空间和树高成正比），一种模仿 size() 在每个结点中添加一个变量（所需空间为线性级别，查询耗时为常数）。

3.2.8 编写一个静态方法 optCompares()，接受一个整型参数 N 并计算一棵最优（完美平衡的）二叉查找树中的一次随机查找命中平均所需的比较次数，如果树中的链接数量为 2 的幂，那么所有的空链接都应该在同一层，否则则分布在最底部的两层中。

416

3.2.9 对于 N=2、3、4、5 和 6，画出用 N 个键可能构造出的所有不同形状的二叉查找树。

3.2.10 编写一个测试用例 TestBST.java 来测试正文中 min()、max()、floor()、ceiling()、

select()、rank()、delete()、deleteMin()、deleteMax() 和 keys() 方法的实现。可以参考 3.1.3.1 节的标准索引用例，使它接受其他合适的命令行参数。

3.2.11 高度为 N 且含有 N 个结点的二叉树能有多少种形状？使用 N 个不同的键能有多少种不同的方式构造一棵高度为 N 的二叉查找树？（参考练习 3.2.2）

3.2.12 实现一种二叉查找树，舍弃 rank() 和 select() 方法并且不在 Node 对象中使用计数器。

3.2.13 为二叉查找树实现非递归的 put() 和 get() 方法。

部分解答，以下是 get() 方法的实现：

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if (cmp == 0) return x.val;
      else if (cmp < 0) x = x.left;
      else if (cmp > 0) x = x.right;
   }
   return null;
}
```

put() 的实现更复杂一些，因为它需要保存一个指向底层结点的链接，以便使之成为新结点的父结点。你还需要额外遍历一遍查找路径来更新所有的结点计数器以保证结点插入的正确性。因为在性能优先的实现中查找的次数比插入多得多，有必要使用这段 get() 代码，而相应的 put() 实现则无关紧要。 417

3.2.14 实现非递归的 min()、max()、floor()、ceiling()、rank() 和 select() 方法。

3.2.15 对于右下方的二叉查找树，给出计算下列方法的过程中结点的访问序列。

a. floor("Q")

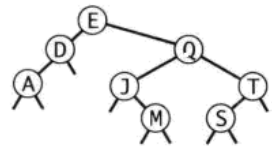
b. select(5)

c. ceiling("Q")

d. rank("J")

e. size("D", "T")

f. keys("D", "T")



3.2.16 设一棵树的外部路径长度为从根结点到空链接的所有路径上的结点总数。证明对于大小为 N 的任意二叉树，其外部路径长度和内部路径长度之差为 $2N$（可以参考命题 C）

3.2.17 从练习 3.2.1 构造的二叉查找树中将所有键按照插入顺序逐个删除并画出每次删除所得到的树。

3.2.18 从练习 3.2.1 构造的二叉查找树中将所有键按照字母顺序逐个删除并画出每次删除所得到的树。

3.2.19 从练习 3.2.1 构造的二叉查找树中逐次删除树的根结点并画出每次删除所得到的树。

3.2.20 请证明：对于含有 N 个结点的二叉查找树，接受两个参数的 size() 方法所需的运行时间最多为树高的倍数加上查找范围内的键的数量。

3.2.21 为二叉查找树添加一个 randomKey() 方法来在和树高成正比的时间内从符号表中随机返回一个键。

3.2.22 请证明：若一棵二叉查找树中的一个结点有两个子结点，那么它的后继结点不会有左子结点，前继结点不会有右子结点。

3.2.23 delete() 方法符合交换律吗？（先删除 x 后删除 y 和先删除 y 后删除 x 能够得到相同的结果吗？）

418 3.2.24 请证明：使用基于比较的算法构造一棵二叉查找树所需的最小比较次数为 $\lg(N!)$~$N\lg N$。

提高题

3.2.25 完美平衡。编写一段程序，用一组键构造一棵和二分查找等价的二叉查找树。也就是说，在这棵树中查找任意键所产生的比较序列和在这组键中使用二分查找所产生的比较序列完全相同。

3.2.26 准确的概率。计算用 N 个随机的互不相同的键构造出练习 3.2.9 中的每一棵树的概率。

3.2.27 内存使用。基于 1.4 节的假设，对于 N 对键值比较二叉查找树和 BinarySearchST 以及 SequentialSearchST 的内存使用情况。不需要记录键值本身占用的内存，只统计它们的引用。用图精确描述一棵以 String 为键、Integer 为值的二叉查找树（比如 FrequencyCounter 构造的那种）的内存使用情况，然后估计 FrequencyCounter 在使用二叉查找树处理《双城记》时树的内存使用情况（精确到字节）。

3.2.28 缓存。修改二叉查找树的实现，将最近访问的结点 Node 保存在一个变量中，这样 get() 或 put() 再次访问同一个键时就只需要常数时间了（参考练习 3.1.25）。

3.2.29 二叉树检查。编写一个递归的方法 isBinaryTree()，接受一个结点 Node 为参数。如果以该结点为根的子树中的结点总数和计数器的值 N 相符则返回 true，否则返回 false。注意：这项检查也能保证数据结构中不存在环，因此这的确是一棵二叉树！

3.2.30 有序性检查。编写一个递归的方法 isOrdered()，接受一个结点 Node 和 min、max 两个键作为参数。如果以该结点为根的子树中的所有结点都在 min 和 max 之间，min 和 max 的确分别是树中的最小和最大的结点且二叉查找树的有序性对树中的所有键都成立，返回 true，否则返回 false。

3.2.31 等值键检查。编写一个方法 hasNoDuplicates()，接受一个结点 Node 为参数。如果以该结点为根的二叉查找树中不含有等值的键则返回 true，否则返回 false。假设树已经通过了前几道练习的检查。

3.2.32 验证。编写一个方法 isBST()，接受一个结点 Node 为参数。若该结点是一个二叉查找树的根结点则返回 true，否则返回 false。提示：这个任务比看起来要困难，它和你调用前三题中各个

419 方法的顺序有关。

解答：

```
private boolean isBST()
{
   if (!isBinaryTree(root)) return false;
   if (!isOrdered(root, min(), max())) return false;
   if (!hasNoDuplicates(root)) return false;
   return true;
}
```

3.2.33 选择/排名检查。编写一个方法，对于 0 到 size()-1 之间的所有 i，检查 i 和 rank(select(i)) 是否相等，并检查二叉查找树中的的任意键 key 和 select(rank(key)) 是否相等。

3.2.34 线性符号表。你的目标是实现一个扩展的符号表 ThreadST，支持以下两个运行时间为常数的操作：

```
Key next(Key key)，key 的下一个键（若 key 为最大键则返回空）
Key prev(Key key)，key 的上一个键（若 key 为最小键则返回空）
```

要做到这一点需要在结点中增加 pred 和 succ 两个变量来保存结点的前继和后继结点，并相应修改 put()、deleteMin()、deleteMax() 和 delete() 方法来维护这两个变量。

3.2.35 *改进的分析*。为了更好地解释正文表格中的试验结果请改进它的数学模型。证明随着 N 的增大，在一棵随机构造的二叉查找树中，一次命中查找所需的平均比较次数会趋近于 $\text{limit}(2\ln N)+2\gamma-3\approx 1.39\lg N-1.85$，其中 $\gamma=0.57721\cdots$，即欧拉常数。*提示*：参考 2.3 节中对快速排序的分析，$1/x$ 的积分趋近于 $\ln N+\gamma$。

3.2.36 *迭代器*。能否实现一个非递归版本的 keys() 方法，其使用的额外空间和树的高度成正比（和查找范围内的键的多少无关）？

3.2.37 *按层遍历*。编写一个方法 printLevel()，接受一个结点 Node 作为参数，按照层级顺序打印以该结点为根的子树（即按每个结点到根结点的距离的顺序，同一层的结点应该按从左至右的顺序）。*提示*：使用队列 Queue。

3.2.38 *绘图*。为二叉查找树添加一个方法 draw()，按照正文中的样式将树绘制出来。*提示*：在结点中用变量保存坐标并用递归的方法设置这些变量。 420 ~ 421

实验题

3.2.39 *平均情况*。用经验数据评估在一棵由 N 个随机结点构造的二叉查找树中，一次命中的查找和未命中的查找平均所需的比较次数的平均差和标准差，其中 $N=10^4$、10^5 和 10^6，重复实验 100 遍。将你的实验结果和练习 3.2.35 给出的计算平均比较次数的公式进行对比。

3.2.40 *树的高度*。用经验数据评估一棵由 N 个随机结点构造的二叉查找树的平均高度，其中 $N=10^4$、10^5 和 10^6，重复实验 100 遍。将你的试验结果和正文中给出的估计值 $2.99\lg N$ 进行对比。

3.2.41 *数组表示*。开发一个二叉查找树的实现，用三个数组表示一棵树（预先分配为构造函数中所指定的最大长度）：一个数组用来保存键，一个数组用来保存左链接的索引，一个数组用来保存右链接的索引。比较你的程序和标准实现的性能。

3.2.42 *Hibbard 删除方法的性能问题*。编写一个程序，从命令行接受一个参数 N 并构造一棵由 N 个随机键生成的二叉查找树，然后进入一个循环。在循环中它先删除一个随机键（delete(select(StdRandom.uniform(N)))），然后再插入一个随机键，如此循环 N^2 次。循环结束后，计算并打印树的内部平均路径长度（内部路径长度除以 N 再加 1）。对于 $N=10^2$、10^3 和 10^4，运行你的程序来验证一个有些违反直觉的假设：这个过程会增加树的平均路径长度，增加的长度和 N 的平方根成正比。使用能够随机选择前继或后继结点的 delete() 方法重复这个实验。

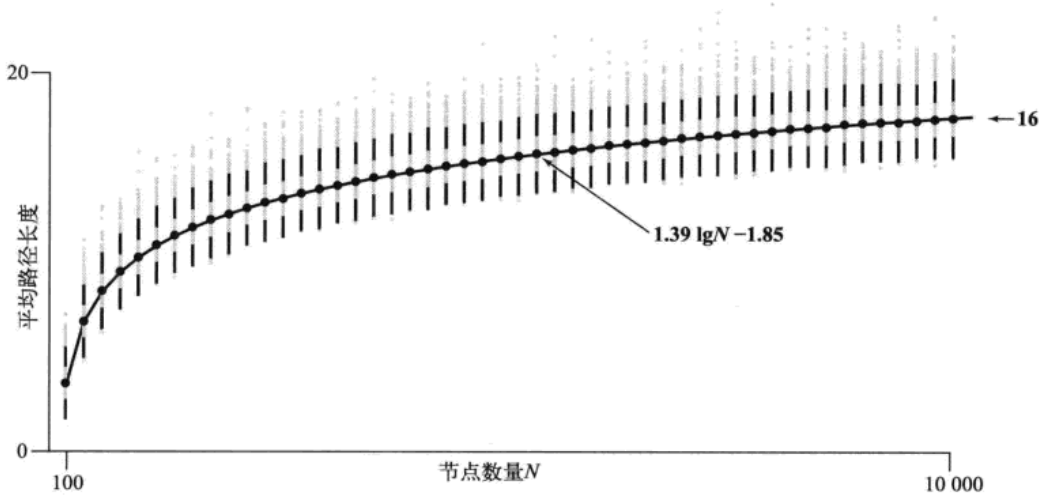
3.2.43 *put()/get() 方法的比例*。用经验数据评估当使用 FrequencyCounter 来统计 100 万个随机整数中每个数的出现频率时，二叉查找树中 put() 方法和 get() 方法所消耗的时间的比例。

3.2.44 *绘制成本图*。改造二叉查找树的实现来绘制本节所示的那种能够显示计算中每次 put() 操作成本的图。

3.2.45 *实际耗时*。改造 FrequencyCounter，使用 Stopwatch 和 StdDraw 绘图，其中 x 轴表示 get() 和 put() 调用的总数，y 轴为总运行时间，每次调用之后即在当前运行时间处绘制一个点。使用 SequentialSearchST 和你的程序处理《双城记》，再用 BinarySearchST 处理一遍，最后用二叉查找树处理一遍，然后讨论运行的结果。*注意*：曲线中突然的跳跃可能是*缓存*导致的，这已经超出了这个问题的讨论范围（请见练习 3.1.39）。 422

3.2.46 二叉查找树的临界点。使用随机 double 值作为键，分别找出使得二叉查找树的符号表比二分查找要快 10、100 倍和 1000 倍的 N 值。分析并预测 N 的大小并通过实验验证它。

3.2.47 平均查找耗时。用实验研究和计算在一棵由 N 个随机结点构造的二叉查找树中到达任意结点的平均路径长度（内部路径长度除以 N 再加 1）的平均差和标准差，对于 100 到 10 000 之间的每个 N 重复实验 1000 遍。将结果绘制成和图 3.2.14 相似的一张 Tufte 图，并画上函数 $1.39\lg N-1.85$ 的曲线（请见练习 3.2.35 和练习 3.2.39）。



423 图 3.2.14 一棵随机构造的二叉查找树中由根到达任意结点的平均路径长度（另见彩插）

3.3 平衡查找树

我们在前面几节中学习过的算法已经能够很好地用于许多应用程序中，但它们在最坏情况下的性能还是很糟糕。在本节中我们会介绍一种二分查找树并能保证无论如何构造它，它的运行时间都是对数级别的。理想情况下我们希望能够保持二分查找树的平衡性。在一棵含有 N 个结点的树中，我们希望树高为 $\sim\lg N$，这样我们就能保证所有查找都能在 $\sim\lg N$ 次比较内结束，就和二分查找一样（请见命题 B）。不幸的是，在动态插入中保证树的完美平衡的代价太高了。在本节中，我们稍稍放松完美平衡的要求并将学习一种能够保证符号表 API 中所有操作（范围查找除外）均能够在对数时间内完成的数据结构。

3.3.1 2-3 查找树

为了保证查找树的平衡性，我们需要一些灵活性，因此在这里我们允许树中的一个结点保存多个键。确切地说，我们将一棵标准的二叉查找树中的结点称为 2- 结点（含有一个键和两条链接），而现在我们引入 3- 结点，它含有两个键和三条链接。2- 结点和 3- 结点中的每条链接都对应着其中保存的键所分割产生的一个区间。

定义。一棵 2-3 查找树或为一棵空树，或由以下结点组成：

- 2- 结点，含有一个键（及其对应的值）和两条链接，左链接指向的 2-3 树中的键都小于该结点，右链接指向的 2-3 树中的键都大于该结点。
- 3- 结点，含有两个键（及其对应的值）和三条链接，左链接指向的 2-3 树中的键都小于该结点，中链接指向的 2-3 树中的键都位于该结点的两个键之间，右链接指向的 2-3 树中的键都大于该结点。

和以前一样，我们将指向一棵空树的链接称为空链接。2-3 查找树如图 3.3.1 所示。

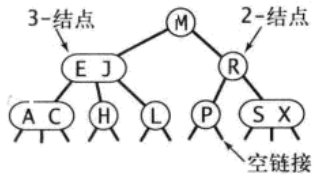


图 3.3.1 2-3 查找树示意图

一棵完美平衡的 2-3 查找树中的所有空链接到根结点的距离都应该是相同的。简洁起见，这里我们用 2-3 树指代一棵完美平衡的 2-3 查找树（在其他情况下这个词应该表示一种更一般的结构）。稍后我们将会学习定义并高效地实现 2- 结点、3- 结点和 2-3 树的基本操作。现在先假设我们已经能够自如地操作它们并来看看应该如何将它们用作查找树。 424

3.3.1.1 查找

将二叉查找树的查找算法一般化我们就能够直接得到 2-3 树的查找算法。要判断一个键是否在树中，我们先将它和根结点中的键比较。如果它和其中任意一个相等，查找命中；否则我们就根据比较的结果找到指向相应区间的链接，并在其指向的子树中递归地继续查找。如果这是个空链接，查找未命中。具体查找过程如图 3.3.2 所示。

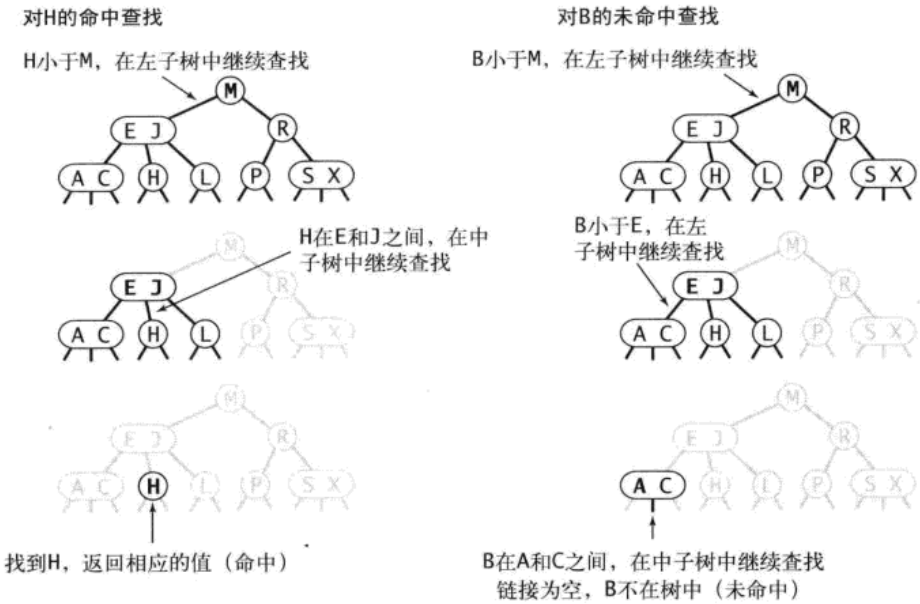


图 3.3.2 2-3 树中的查找命中（左）和未命中（右）

3.3.1.2 向 2- 结点中插入新键

要在 2-3 树中插入一个新结点，我们可以和二叉查找树一样先进行一次未命中的查找，然后把新结点挂在树的底部。但这样的话树无法保持完美平衡性。我们使用 2-3 树的主要原因就在于它能够在插入后继续保持平衡。如果未命中的查找结束于一个 2- 结点，事情就好办了：我们只要把这个 2- 结点替换为一个 3- 结点，将要插入的键保存在其中即可（如图 3.3.3 所示）。如果未命中的查找结束于一个 3- 结点，事情就要麻烦一些。

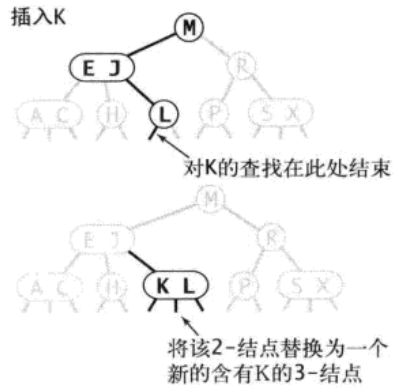


图 3.3.3 向 2- 结点中插入新的键

425

3.3.1.3 向一棵只含有一个 3- 结点的树中插入新键

在考虑一般情况之前，先假设我们需要向一棵只含有一个 3- 结点的树中插入一个新键。这棵树中有两个键，所以在它唯一的结点中已经没有可插入新键的空间了。为了将新键插入，我们先临时将新键存入该结点中，使之成为一个 4- 结点。它很自然地扩展了以前的结点并含有 3 个键和 4 条链接。创建一个 4- 结点很方便，因为很容易将它转换为一棵由 3 个 2- 结点组成的 2-3 树，其中一个结点（根）含有中键，一个结点含有 3 个键中的最小者（和根结点的左链接相连），一个结点含有 3 个键中的最大者（和根结点的右链接相连）。这棵树既是一棵含有 3 个结点的二叉查找树，同时也是一棵完美平衡的 2-3 树，因为其中所有的空链接到根结点的距离都相等。插入前树的高度为 0，插入后树的高度为 1。这个例子很简单但却值得学习，它说明了 2-3 树是如何生长的，如图 3.3.4 所示。

3.3.1.4 向一个父结点为 2- 结点的 3- 结点中插入新键

作为第二轮热身，假设未命中的查找结束于一个 3- 结点，而它的父结点是一个 2- 结点。在这种情况下我们需要在维持树的完美平衡的前提下为新键腾出空间。我们先像刚才一样构造一个临时

的 4- 结点并将其分解，但此时我们不会为中键创建一个新结点，而是将其移动至原来的父结点中。你可以将这次转换看成将指向原 3- 结点的一条链接替换为新父结点中的原中键左右两边的两条链接，并分别指向两个新的 2- 结点。根据我们的假设，父结点中是有空间的：父结点是一个 2- 结点（一个键两条链接），插入之后变为了一个 3- 结点（两个键 3 条链接）。另外，这次转换也并不影响（完美平衡的）2-3 树的主要性质。树仍然是有序的，因为中键被移动到父结点中去了；树仍然是完美平衡的，插入后所有的空链接到根结点的距离仍然相同。请确认你完全理解了这次转换——它是 2-3 树的动态变化的核心，其过程如图 3.3.5 所示。

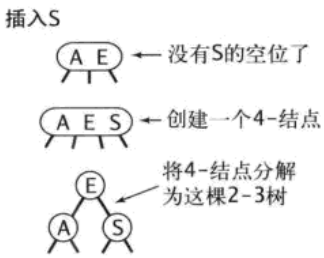


图 3.3.4 向一棵只含有一个 3- 结点的树中插入新键

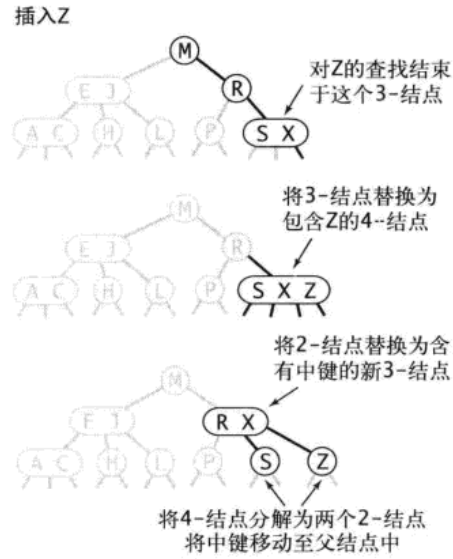


图 3.3.5 向一个父结点为 2- 结点的 3- 结点中插入新键

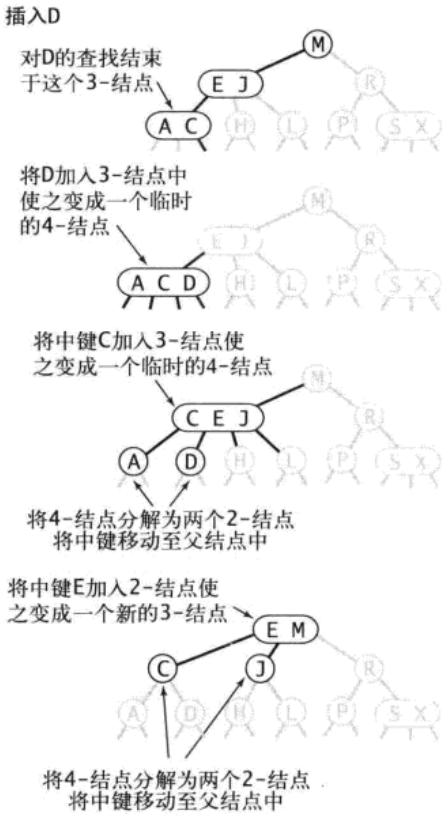
426

3.3.1.5 向一个父结点为 3- 结点的 3- 结点中插入新键

现在假设未命中的查找结束于一个父结点为 3- 结点的结点。我们再次和刚才一样构造一个临时的 4- 结点并分解它，然后将它的中键插入它的父结点中。但父结点也是一个 3- 结点，因此我们再用这个中键构造一个新的临时 4- 结点，然后在这个结点上进行相同的变换，即分解这个父结点并将它的中键插入到它的父结点中去。推广到一般情况，我们就这样一直向上不断分解临时的 4- 结点并将中键插入更高层的父结点，直至遇到一个 2- 结点并将它替换为一个不需要继续分解的 3- 结点，或者是到达 3- 结点的根。该过程如图 3.3.6 所示。

3.3.1.6 分解根结点

如果从插入结点到根结点的路径上全都是 3- 结点，我们的根结点最终变成一个临时的 4- 结点。此时我们可以按照向一棵只有一个 3- 结点的树中插入新键的方法处理这个问题。我们将临时的 4- 结点分解为 3 个 2- 结点，使得树高加 1，如图 3.3.7 所示。请注意，这次最后的变换仍然保持了树的完美平衡性，因为它变换的是根结点。





427 图 3.3.6 向一个父结点为 3- 结点的 3- 结点中插入新键

图 3.3.7 分解根结点

3.3.1.7 局部变换

将一个 4- 结点分解为一棵 2-3 树可能有 6 种情况，都总结在了图 3.3.8 中。这个 4- 结点可能是根结点，可能是一个 2- 结点的左子结点或者右子结点，也可能是一个 3- 结点的左子结点、中子结点或者右子结点。2-3 树插入算法的根本在于这些变换都是*局部的*：除了相关的结点和链接之外不必修改或者检查树的其他部分。每次变换中，变更的链接数量不会超过一个很小的常数。需要特别指出的是，不光是在树的底部，树中的*任何地方*只要符合相应的模式，变换都可以进行。每个变换都会将 4- 结点中的一个键送入它的父结点中，并重构相应的链接而不必涉及树的其他部分。

3.3.1.8 全局性质

这些*局部*变换不会影响树的*全局*有序性和平衡性：任意空链接到根结点的路径长度都是相等的。作为参考，图 3.3.9 所示的是当一个 4- 结点是一个 3- 结点的中子结点时的完整变换情况。如果在变换之前根结点到所有空链接的路径长度为 h，那么变换之后该长度仍然为 h。所有的变换都具有这个性质，即使是将一个 4- 结点分解为两个 2- 结点并将其父结点由 2- 结点变为 3- 结点，或是由 3- 结点变为一个临时的 4- 结点时也是如此。当根结点被分解为 3 个 2- 结点时，所有空链接到根结点的路径长度才会加 1。如果你还没有完全理解，请完成练习 3.3.7。它要求你为其他的 5 种情况画出图 3.3.8 的扩展图来证明这一点。理解所有局部变换都不会影响整棵树的有序性和平衡性是理解这个算法的关键。

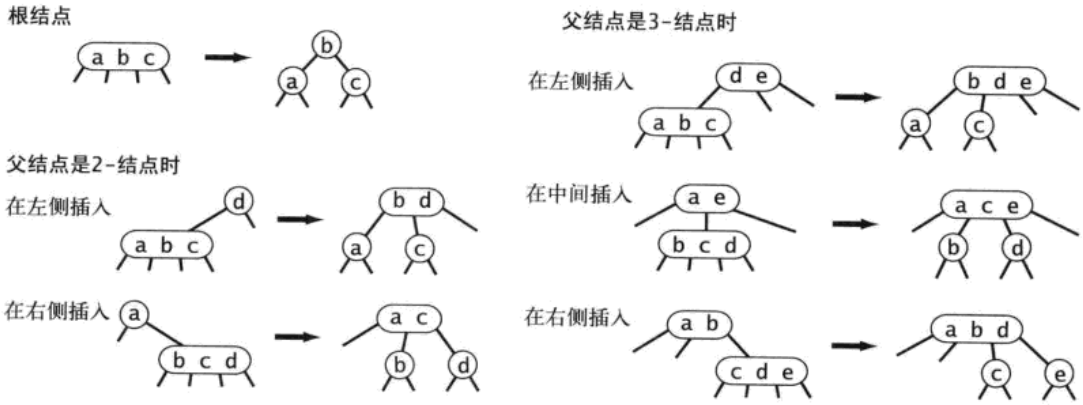


图 3.3.8 在一棵 2-3 树中分解一个 4- 结点的情况汇总

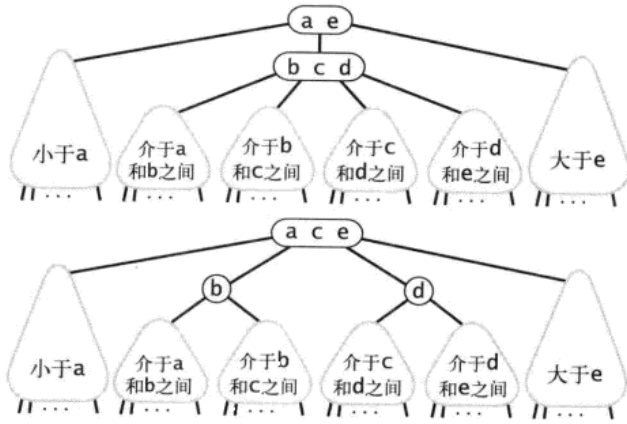


图 3.3.9 4- 结点的分解是一次局部变换，不会影响树的有序性和平衡性 428

和标准的二叉查找树由上向下生长不同，2-3 树的生长是由下向上的。如果你花点时间仔细研究一下图 3.3.10，就能很好地理解 2-3 树的构造方式。它给出了我们的标准索引测试用例中产生的一系列 2-3 树，以及一系列由同一组键按照升序依次插入到树中时所产生的所有 2-3 树。还记得在二叉查找树中，按照升序插入 10 个键会得到高度为 9 的一棵最差查找树吗？如果使用 2-3 树，树的高度是 2。

以上的文字已经足够为我们定义一个使用 2-3 树作为数据结构的符号表的实现了。2-3 树的分析和二叉查找树的分析大不相同，因为我们主要感兴趣的是最坏情况下的性能，而非一般情况（这种情况下我们会用随机键模型分析预期的性能）。在符号表的实现中，一般我们无法控制用例会按照什么顺序向表中插入键，因此对最坏情况的分析是唯一能够提供性能保证的办法。

命题 F。在一棵大小为 N 的 2-3 树中，查找和插入操作访问的结点必然不超过 $\lg N$ 个。

证明。一棵含有 N 个结点的 2-3 树的高度在 $\lfloor \log_3 N \rfloor = \lfloor (\lg N)/(\lg 3) \rfloor$（如果树中全是 3- 结点）和 $\lfloor \lg N \rfloor$（如果树中全是 2- 结点）之间（请见练习 3.3.4）。

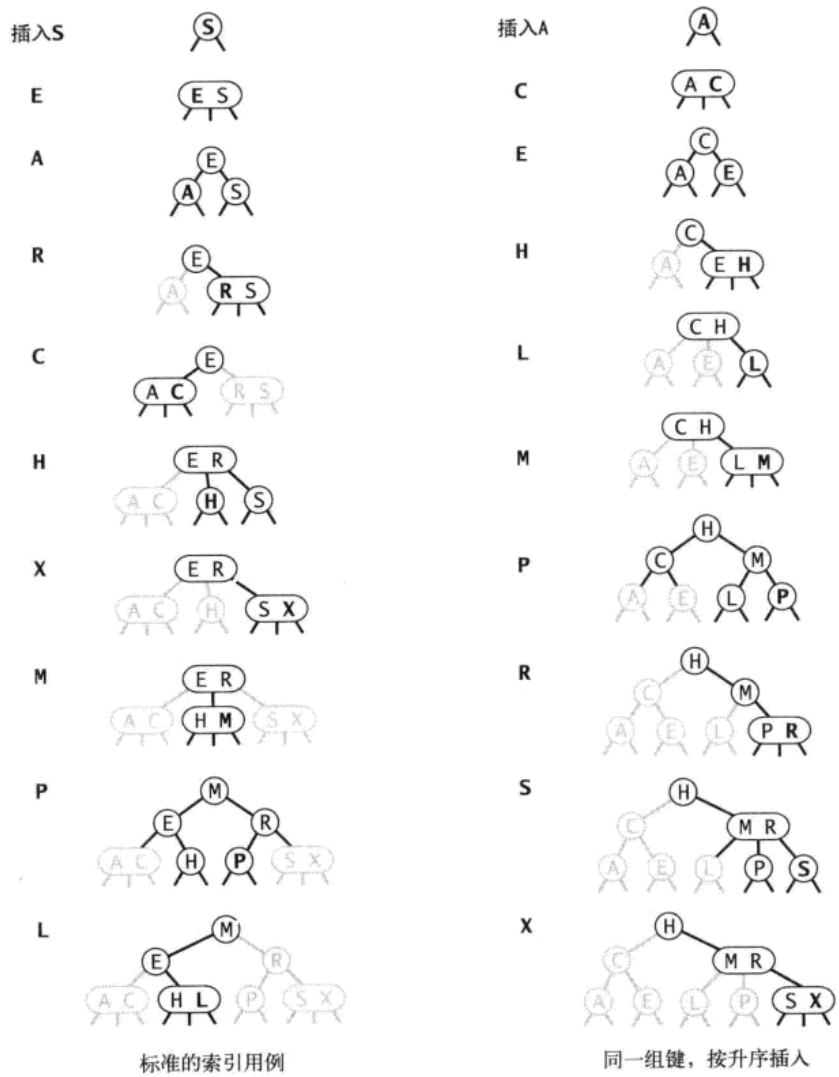


图 3.3.10 2-3 树的构造轨迹

因此我们可以确定 2-3 树在最坏情况下仍有较好的性能。每个操作中处理每个结点的时间都不会超过一个很小的常数，且这两个操作都只会访问一条路径上的结点，所以任何查找或者插入的成本都肯定不会超过对数级别。通过对比图 3.3.11 中的 2-3 树和表 3.2.1 中由相同的键构造的二叉查找树你也可以看到，完美平衡的 2-3 树要平展得多。例如，含有 10 亿个结点的一棵 2-3 树的高度仅在 19 到 30 之间。我们最多只需要访问 30 个结点就能够在 10 亿个键中进行任意查找和插入操作，这是相当惊人的。



图 3.3.11 由随机键构造的一棵典型的 2-3 树

但是，我们和真正的实现还有一段距离。尽管我们可以用不同的数据类型表示 2- 结点和 3- 结点并写出变换所需的代码，但用这种直白的表示方法实现大多数的操作并不方便，因为需要处理的情况实在太多。我们需要维护两种不同类型的结点，将被查找的键和结点中的每个键进行比较，将链接和其他信息从一种结点复制到另一种结点，将结点从一种数据类型转换到另一种数据类型，等等。实现这些不仅需要大量的代码，而且它们所产生的额外开销可能会使算法比标准的二叉查找树更慢。平衡一棵树的初衷是为了消除最坏情况，但我们希望这种保障所需的代码能够越少越好。幸运的是你将看到，我们只需要一点点代价就能用一种统一的方式完成所有变换。

429 ≀ 431

3.3.2 红黑二叉查找树

上文所述的 2-3 树的插入算法并不难理解，现在我们会看到它也不难实现。我们要学习一种名为*红黑二叉查找树*的简单数据结构来表达并实现它。最后的代码量并不大，但理解这些代码是如何工作的以及为什么能够工作却需要一番仔细的探究。

3.3.2.1 替换 3- 结点

红黑二叉查找树背后的基本思想是用标准的二叉查找树（完全由 2- 结点构成）和一些额外的信息（替换 3 - 结点）来表示 2-3 树。我们将树中的链接分为两种类型：*红链接*将两个 2- 结点连接起来构成一个 3- 结点，*黑链接*则是 2-3 树中的普通链接。确切地说，我们将 3- 结点表示为由一条*左斜*的红色链接（两个 2- 结点其中之一是另一个的左子结点）相连的两个 2- 结点，如图 3.3.12 所示。这种表示法的一个优点是，我们无需修改就可以直接使用标准二叉查找树的 `get()` 方法。对于任意的 2-3 树，只要对结点进行转换，我们都可以立即派生出一棵对应的二叉查找树。我们将用这种方式表示 2-3 树的二叉查找树称为*红黑二叉查找树*（以下简称为*红黑树*）。

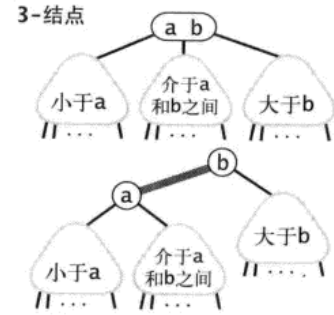


图 3.3.12 由一条红色左链接相连的两个 2- 结点表示一个 3- 结点（另见彩插）

3.3.2.2 一种等价的定义

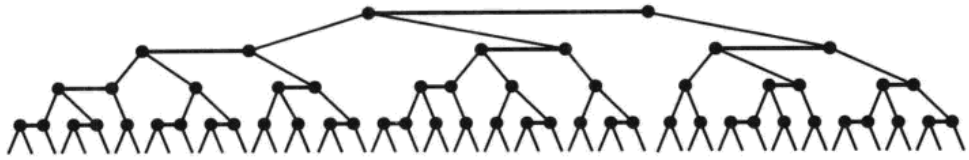
红黑树的另一种定义是含有红黑链接并满足下列条件的二叉查找树：

- 红链接均为左链接；
- 没有任何一个结点同时和两条红链接相连；
- 该树是完美黑色平衡的，即任意空链接到根结点的路径上的黑链接数量相同。

满足这样定义的红黑树和相应的 2-3 树是一一对应的。

3.3.2.3 一一对应

如果我们将一棵红黑树中的红链接画平，那么所有的空链接到根结点的距离都将是相同的（如图 3.3.13 所示）。如果我们将由红链接相连的结点合并，得到的就是一棵 2-3 树。相反，如果将一棵 2-3 树中的 3- 结点画作由红色左链接相连的两个 2- 结点，那么不会存在能够和两条红链接相连的结点，且树必然是完美黑色平衡的，因为黑链接即 2-3 树中的普通链接，根据定义这些链接必然是完美平衡的。无论我们选择用何种方式去定义它们，红黑树都既是二叉查找树，也是 2-3 树，如图 3.3.14 所示。因此，如果我们能够在保持一一对应关系的基础上实现 2-3 树的插入算法，那么我们就能够将两个算法的优点结合起来：二叉查找树中简洁高效的查找方法和 2-3 树中高效的平衡插入算法。

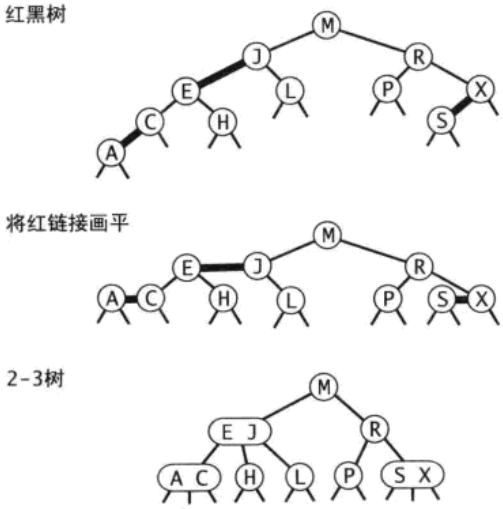


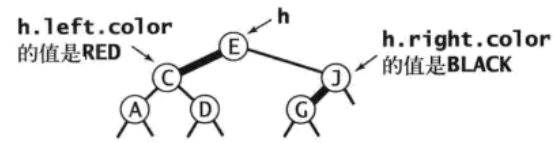
432

图 3.3.13 将红链接画平时，一棵红黑树就是一棵 2-3 树（另见彩插）

3.3.2.4 颜色表示

方便起见，因为每个结点都只会有一条指向自己的链接（从它的父结点指向它），我们将链接的颜色保存在表示结点的 Node 数据类型的布尔变量 color 中。如果指向它的链接是红色的，那么该变量为 true，黑色则为 false。我们约定空链接为黑色。为了代码的清晰我们定义了两个常量 RED 和 BLACK 来设置和测试这个变量。我们使用私有方法 isRed() 来测试一个结点和它的父结点之间的链接的颜色。当我们提到一个结点的颜色时，我们指的是指向该结点的链接的颜色，反之亦然。颜色表示的代码实现如图 3.3.15 所示。






```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;           // 键
   Value val;         // 相关联的值
   Node left, right; // 左右子树
   int N;             // 这棵子树中的结点总数
   boolean color;     // 由其父结点指向它的链接的颜色


   Node(Key key, Value val, int N, boolean color)
   {
      this.key   = key;
      this.val   = val;
      this.N     = N;
      this.color = color;
   }
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

图 3.3.14 红黑树和 2-3 树的一一对应关系（另见彩插） 图 3.3.15 红黑树的结点表示（另见彩插）

3.3.2.5 旋转

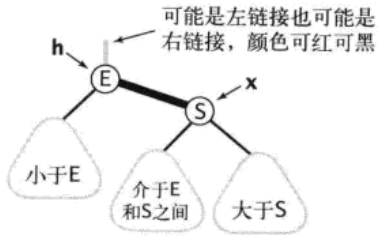
在我们实现的某些操作中可能会出现红色右链接或者两条连续的红链接，但在操作完成前这些情况都会被小心地旋转并修复。旋转操作会改变红链接的指向。首先，假设我们有一条红色的右链接需要被转化为左链接（请见图 3.3.16）。这个操作叫做左旋转，它对应的方法接受一条指向红黑树中的某个结点的链接作为参数。假设被指向的结点的右链接是红色的，这个方法会对树进行必要的调整并返回一个指向包含同一组键的子树且其左链接为红色的根结点的链接。如果你对照图示中

433 调整前后的情况逐行阅读这段代码，你会发现这个操作很容易理解：我们只是将用两个键中的较小

者作为根结点变为将较大者作为根结点。实现将一个红色左链接转换为一个红色右链接的一个*右旋转*的代码完全相同，只需要将 `left` 换成 `right` 即可（如图 3.3.17 所示）。

3.3.2.6 在旋转后重置父结点的链接

无论左旋转还是右旋转，旋转操作都会返回一条链接。我们总是会用 `rotateRight()` 或 `rotateLeft()` 的返回值重置父结点（或是根结点）中相应的链接。返回的链接可能是左链接也可能是右链接，但是我们总会将它赋予父结点中的链接。这个链接可能是红色也可能是黑色——`rotateLeft()` 和 `rotateRight()` 都通过将 `x.color` 设为 `h.color` 保留它原来的颜色。这可能会产生两条连续的红链接，但我们的算法会继续用旋转操作修正这种情况。例如，代码 `h = rotateLeft(h);` 将旋转结点 h 的红色右链接，使得 h 指向了旋转后的子树的根结点（组成该子树中的所有键和旋转前相同，只是根结点发生了变化）。这种简洁的代码是我们使用递归实现二叉查找树的各种方法的主要原因。你会看到，它使得旋转操作成为了普通插入操作的一个简单补充。




```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

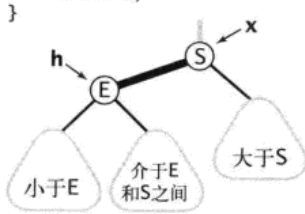
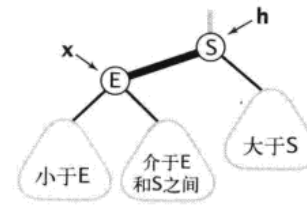


图 3.3.16 左旋转 h 的右链接（另见彩插）




```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

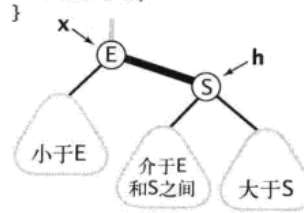


图 3.3.17 右旋转 h 的左链接（另见彩插）

434

在插入新的键时我们可以使用旋转操作帮助我们保证 2-3 树和红黑树之间的一一对应关系，因为旋转操作可以保持红黑树的两个重要性质：*有序性*和*完美平衡性*。也就是说，我们在红黑树中进行旋转时无需为树的有序性或者完美平衡性担心。下面我们来看看应该如何使用旋转操作来保持红黑树的另外两个重要性质（不存在两条连续的红链接和不存在红色的右链接）。我们先用一些简单的情况热热身。

3.3.2.7 向 2- 结点中插入新键

一棵只含有一个键的红黑树只含有一个 2- 结点。插入另一个键之后，我们马上就需要将它们旋转。如果新键小于老键，我们只需要新增一个红色的结点即可，新的红黑树和单个 3- 结点完全等价。如果新键大于老键，那么新增的红色结点将会产生一条红色的右链接。我们需要使用 `root = rotateLeft(root);` 来将其旋转为红色左链接并修正根结点的链接，插入操作才算完成。两种情

况的结果均为一棵和单个3-结点等价的红黑树，其中含有两个键，一条红链接，树的黑链接高度为1，如图3.3.18所示。

3.3.2.8 向树底部的2-结点插入新键

用和二叉查找树相同的方式向一棵红黑树中插入一个新键会在树的底部新增一个结点（为了保证有序性），但总是用红链接将新结点和它的父结点相连。如果它的父结点是一个2-结点，那么刚才讨论的两种处理方法仍然适用。如果指向新结点的是父结点的左链接，那么父结点就直接成为了一个3-结点；如果指向新结点的是父结点的右链接，这就是一个错误的3-结点，但一次左旋转就能够修正它，如图3.3.19所示。

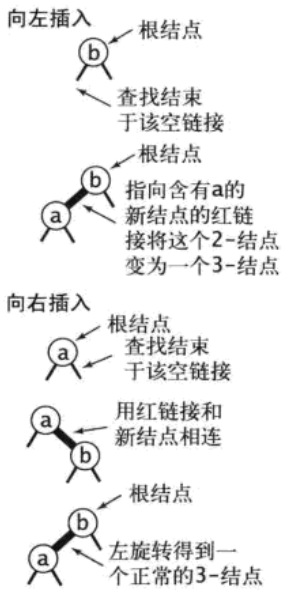


图3.3.18 向单个2-结点中插入一个新键（另见彩插）

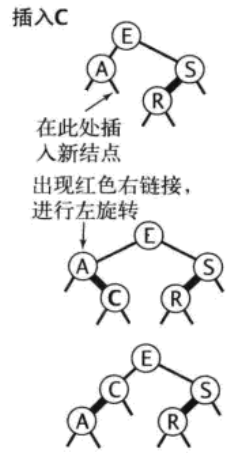


图3.3.19 向树底部的2-结点插入一个新键（另见彩插）

3.3.2.9 向一棵双键树（即一个3-结点）中插入新键

这种情况又可分为三种子情况：新键小于树中的两个键，在两者之间，或是大于树中的两个键。每种情况中都会产生一个同时连接到两条红链接的结点，而我们的目标就是修正这一点。

- 三者中最简单的情况是新键大于原树中的两个键，因此它被连接到3-结点的右链接。此时树是平衡的，根结点为中间大小的键，它有两条红链接分别和较小和较大的结点相连。如果我们将两条链接的颜色都由红变黑，那么我们就得到了一棵由三个结点组成、高为2的平衡树。它正好能够对应一棵2-3树，如图3.3.20（左）。其他两种情况最终也会转化为这种情况。

435

- 如果新键小于原树中的两个键，它会被连接到最左边的空链接，这样就产生了两条连续的红链接，如图3.3.20（中）。此时我们只需要将上层的红链接右旋转即可得到第一种情况（中值键为根结点并和其他两个结点用红链接相连）。
- 如果新键介于原树中的两个键之间，这又会产生两条连续的红链接，一条红色左链接接一条红色右链接，如图3.3.20（右）。此时我们只需要将下层的红链接左旋转即可得到第二种情况（两条连续的红色左链接）。

总的来说，我们通过 0 次、1 次和 2 次旋转以及颜色的变化得到了期望的结果。在 2-3 树中，请确认你完全理解了这些转换，它们是红黑树的动态变化的关键。

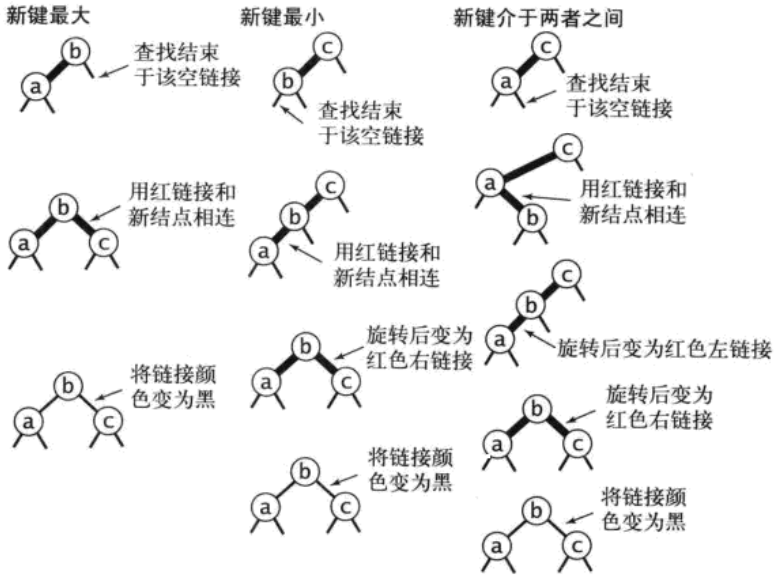
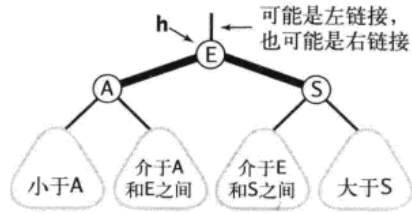


图 3.3.20 向一棵双键树（即一个 3- 结点）中插入一个新键的三种情况（另见彩插）

3.3.2.10 颜色转换

如图 3.3.21 所示，我们专门用一个方法 `flipColors()` 来转换一个结点的两个红色子结点的颜色。除了将子结点的颜色由红变黑之外，我们同时还要将父结点的颜色由黑变红。这项操作最重要的性质在于它和旋转操作一样是局部变换，不会影响整棵树的黑色平衡性。根据这一点，我们马上能够在下面完整地实现红黑树。




```
void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

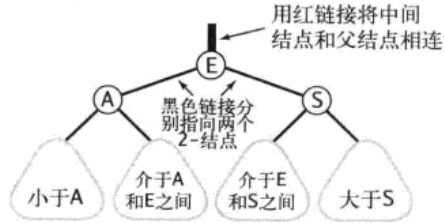


图 3.3.21 分解 4- 结点的同时转换链接的颜色（另见彩插）

3.3.2.11 根结点总是黑色

在 3.3.29 所述的情况中，颜色转换会使根结点变为红色。这也可能出现在很大的红黑树中。严格地说，红色的根结点说明根结点是一个 3- 结点的一部分，但实际情况并不是这样。因此我们在每次插入后都会将根结点设为黑色。注意，每当根结点由红变黑时树的黑链接高度就会加 1。

3.3.2.12 向树底部的 3- 结点插入新键

现在假设我们需要在树的底部的一个 3- 结点下加入一个新结点。前面讨论过的三种情况都会出现，如图 3.3.22 所示。指向新结点的链接可能是 3- 结点的右链接（此时我们只需要转换颜色即可），或是左链接（此时我们需要进行右旋转然后再转换颜色），或是中链接（此时我们需要先左旋转下层链接然后右旋转上层链接，最后再转换颜色）。颜色转换

会使到中结点的链接变红，相当于将它送入了父结点。这意味着在父结点中继续插入一个新键，我们也会继续用相同的办法解决这个问题。

3.3.2.13 将红链接在树中向上传递

2-3 树中的插入算法需要我们分解 3- 结点，将中间键插入父结点，如此这般直到遇到一个 2- 结点或是根结点。我们所考虑过的所有情况都正是为了达成这个目标：每次必要的旋转之后我们都会进行颜色转换，这使得中结点变红。在父结点看来，处理这样一个红色结点的方式和处理一个新插入的红色结点完全相同，即继续把红链接转移到中结点上去。图 3.3.23 中总结的三种情况显示了在红黑树中实现 2-3 树的插入算法的关键操作所需的步骤：要在一个 3- 结点下插入新键，先创建一个临时的 4- 结点，将其分解并将红链接由中间键传递给它的父结点。重复这个过程，我们就能将红链接在树中向上传递，直至遇到一个 2- 结点或者根结点。

总之，只要谨慎地使用左旋转、右旋转和颜色转换这三种简单的操作，我们就能够保证插入操作后红黑树和 2-3 树的一一对应关系。在沿着插入点到根结点的路径向上移动时在所经过的每个结点中顺序完成以下操作，我们就能完成插入操作：

- 如果右子结点是红色的而左子结点是黑色的，进行左旋转；
- 如果左子结点是红色的且它的左子结点也是红色的，进行右旋转；
- 如果左右子结点均为红色，进行颜色转换。

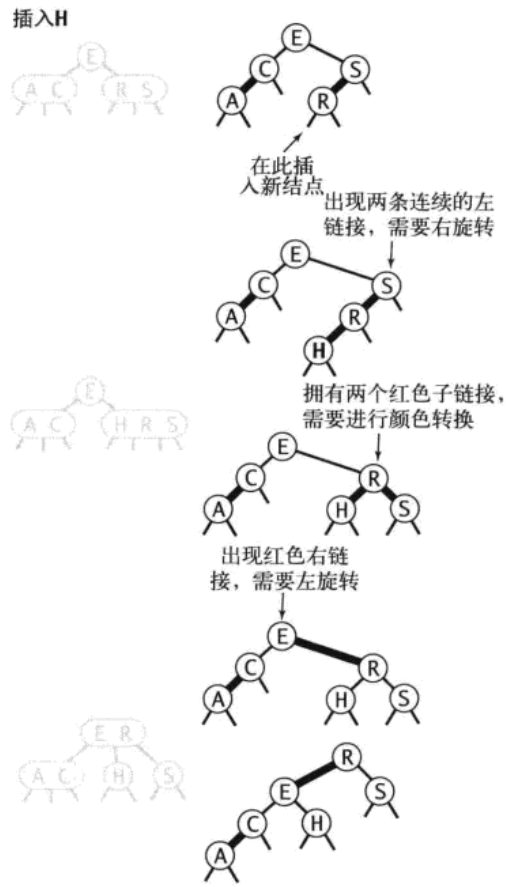


图 3.3.22 向树底部的 3- 结点插入一个新键（另见彩插）

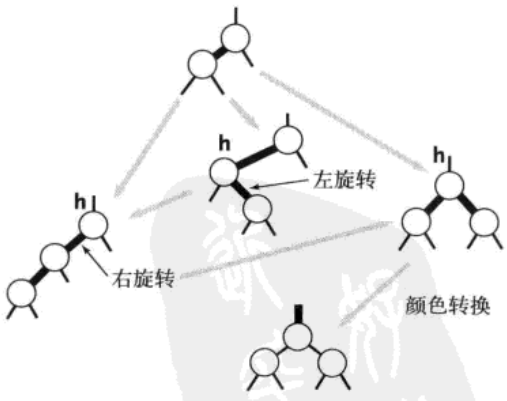


图 3.3.23 红黑树中红链接向上传递（另见彩插）

你应该花点时间确认以上步骤处理了前文描述的所有情况。请注意，第一个操作表示将一个 2- 结点变为一个 3- 结点和插入的新结点与树底部的 3- 结点通过它的中链接相连的两种情况。

3.3.3 实现

因为保持树的平衡性所需的操作是由下向上在每个所经过的结点中进行的，将它们植入我们已有的实现中十分简单：只需要在递归调用之后完成这些操作即可，如算法 3.4 所示。上一段中列出的三种操作都可以通过一个检测两个结点的颜色的 `if` 语句完成。尽管实现所需的代码量很小，

但如果没有我们学习过的两种抽象数据结构（2-3 树和红黑树）作为铺垫，这段实现仍然会非常难以理解。在检查了三到五个结点的颜色之后（也许还需要进行一两次旋转以及颜色转换），我们就可以得到一棵近乎完美平衡的二叉查找树。

图 3.3.24 给出了使用我们的标准索引测试用例进行测试的轨迹和用同一组键按照升序构造一棵红黑树的测试轨迹。仅从红黑树的三种标准操作的角度分析这些例子对我们理解问题很有帮助，之前我们也是这样做的。另一个基本练习是检查它们和 2-3 树的一一对应关系（可以对比图 3.3.10 中由同一组键构造的 2-3 树）。在两种情况中你都能通过思考将 P 插入红黑树所需的转换来检验你对算法的理解程度（请见练习 3.3.12）。

436 ~ 438

算法 3.4 红黑树的插入算法

```
public class RedBlackBST<Key extends Comparable<Key>, Value>
{

   private Node root;

   private class Node // 含有color变量的Node对象（请见3.3.2.4节）

   private boolean isRed(Node h)     // 请见3.3.2.4节
   private Node rotateLeft(Node h)   // 请见图3.3.16
   private Node rotateRight(Node h) // 请见图3.3.17
   private void flipColors(Node h)   // 请见图3.3.21

   private int size()                // 请见算法3.3

   public void put(Key key, Value val)
   {  // 查找key，找到则更新其值，否则为它新建一个结点
      root = put(root, key, val);
      root.color = BLACK;
   }

   private Node put(Node h, Key key, Value val)
   {
      if (h == null)  // 标准的插入操作，和父结点用红链接相连
         return new Node(key, val, 1, RED);

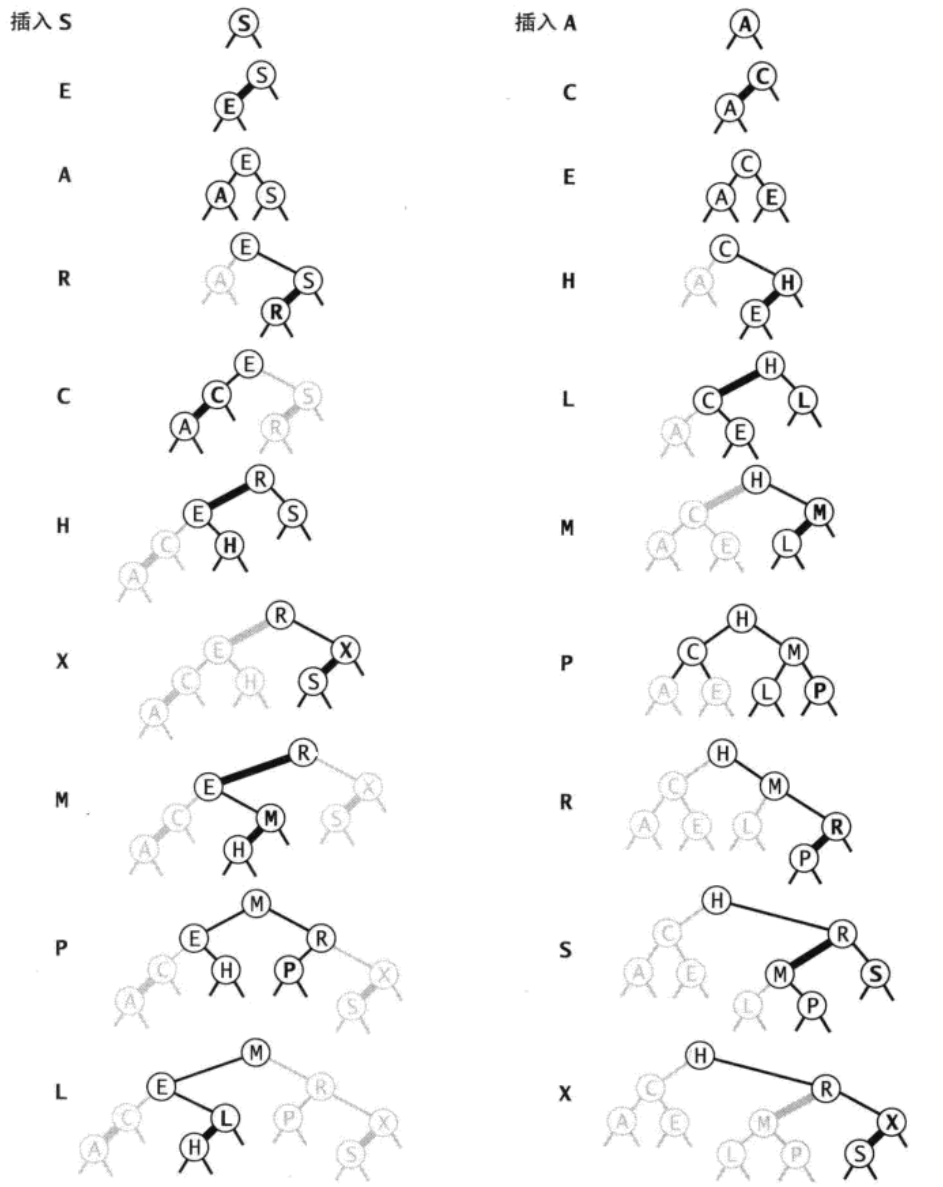
      int cmp = key.compareTo(h.key);
      if      (cmp < 0) h.left  = put(h.left,  key, val);
      else if (cmp > 0) h.right = put(h.right, key, val);
      else h.val = val;

      if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h);
      if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
      if (isRed(h.left) && isRed(h.right))      flipColors(h);

      h.N = size(h.left) + size(h.right) + 1;
      return h;
   }
}
```

除了递归调用后的三条 if 语句，红黑树中 put() 的递归实现和二叉查找树中 put() 的实现完全相同。它们在查找路径上保证了红黑树和 2-3 树的一一对应关系，使得树的平衡性接近完美。第一条 if 语句会将任意含有红色右链接的 3- 结点（或临时的 4- 结点）向左旋转；第二条 if 语句会将临时的 4- 结点中两条连续红链接中的上层链接向右旋转；第三条 if 语句会进行颜色转换并将红链接在树中向上传递（详情请见正文）。

439



440

图 3.3.24 红黑树的构造轨迹（另见彩插）

3.3.4 删除操作

算法 3.4 中的 put() 方法是本书中最复杂的实现之一，而红黑树的 deleteMin()、deleteMax() 和 delete() 的实现更麻烦，我们将它们的完整实现留做练习，但这里仍然需要学习它们的基本原理。要描述删除算法，首先我们要回到 2-3 树。和插入操作一样，我们也可以定义一系列局部变换来在删除一个结点的同时保持树的完美平衡性。这个过程比插入一个结点更加复杂，因为我们不仅要在（为了删除一个结点而）构造临时 4- 结点时沿着查找路径向下进行变换，还要在分解

遗留的 4- 结点时沿着查找路径向上进行变换（同插入操作）。

3.3.4.1 自顶向下的 2-3-4 树

作为第一轮热身，我们先学习一个沿查找路径既能向上也能向下进行变换的稍简单的算法：2-3-4 树的插入算法，2-3-4 树中允许存在我们以前见过的 4- 结点。它的插入算法沿查找路径向下进行变换是为了保证当前结点不是 4- 结点（这样树底才有空间来插入新的键），沿查找路径向上进行变换是为了将之前创建的 4- 结点配平，如图 3.3.25 所示。向下的变换和我们在 2-3 树中分解 4- 结点所进行的变换完全相同。如果根结点是 4- 结点，我们就将它分解成三个 2- 结点，使得树高加 1。在向下查找的过程中，如果遇到一个父结点为 2- 结点的 4- 结点，我们将 4- 结点分解为两个 2- 结点并将中间键传递给它的父结点，使得父结点变为一个 3- 结点；如果遇到一个父结点为 3- 结点的 4- 结点，我们将 4- 结点分解为两个 2- 结点并将中间键传递给它的父结点，使得父结点变为一个 4- 结点；我们不必担心会遇到父结点为 4- 结点的 4- 结点，因为插入算法本身就保证了这种情况不会出现。到达树的底部之后，我们也只会遇到 2- 结点或者 3- 结点，所以我们可以插入新的键。要用红黑树实现这个算法，我们需要：

- 将 4- 结点表示为由三个 2- 结点组成的一棵平衡的子树，根结点和两个子结点都用红链接相连；
- 在向下的过程中分解所有 4- 结点并进行颜色转换；
- 和插入操作一样，在向上的过程中用旋转将 4- 结点配平[①]。

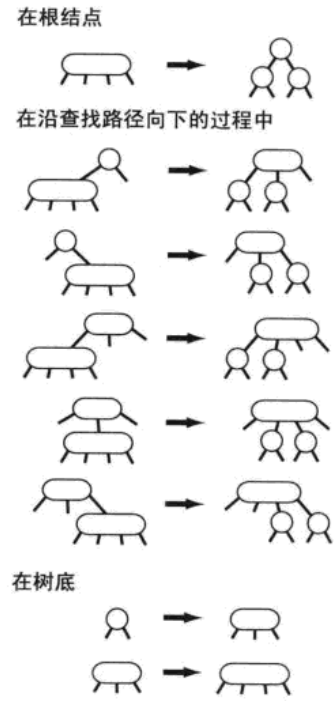


图 3.3.25 自顶向下的 2-3-4 树的插入算法中的变换

441

令人惊讶的是，你只需要移动算法 3.4 的 put() 方法中的一行代码就能实现 2-3-4 树中的插入操作：将 colorFlip() 语句（及其 if 语句）移动到递归调用之前（null 测试和比较操作之间）。在多个进程可以同时访问同一棵树的应用中这个算法优于 2-3 树，因为它操作的总是当前结点的一个或两个链接。我们下面要讲的删除算法和它的插入算法类似，而且也适用于 2-3 树。

3.3.4.2 删除最小键

在第二轮热身中我们要学习 2-3 树中删除最小键的操作。我们注意到从树底部的 3- 结点中删除键是很简单的，但 2- 结点则不然。从 2- 结点中删除一个键会留下一个空结点，一般我们会将它替换为一个空链接，但这样会破坏树的完美平衡性。所以我们需要这样做：为了保证我们不会删除一个 2- 结点，我们沿着左链接向下进行变换，确保当前结点不是 2- 结点（可能是 3- 结点，也可能是临时的 4- 结点）。首先，根结点可能有两种情况。如果根是 2- 结点且它的两个子结点都是 2- 结点，我们可以直接将这三个结点变成一个 4- 结点；否则我们需要保证根结点的左子结点不是 2- 结点，如有必要可以从它右侧的兄弟结点“借”一个键来。以上情况如图 3.3.26 所示。在沿着左链接向下的过程中，保证以下情况之一成立：

- 如果当前结点的左子结点不是 2- 结点，完成；
- 如果当前结点的左子结点是 2- 结点而它的亲兄弟结点不是 2- 结点，将左子结点的兄弟结点中的一个键移动到左子结点中；

① 因为4-结点可以存在，所以可以允许一个结点同时连接到两条链接。——译者注

- 如果当前结点的左子结点和它的亲兄弟结点都是2-结点，将左子结点、父结点中的最小键和左子结点最近的兄弟结点合并为一个4-结点，使父结点由3-结点变为2-结点或者由4-结点变为3-结点。

在遍历的过程中执行这个过程，最后能够得到一个含有最小键的3-结点或者4-结点，然后我们就可以直接从中将其删除，将3-结点变为2-结点，或者将4-结点变为3-结点。然后我们再回头向上分解所有临时的4-结点。

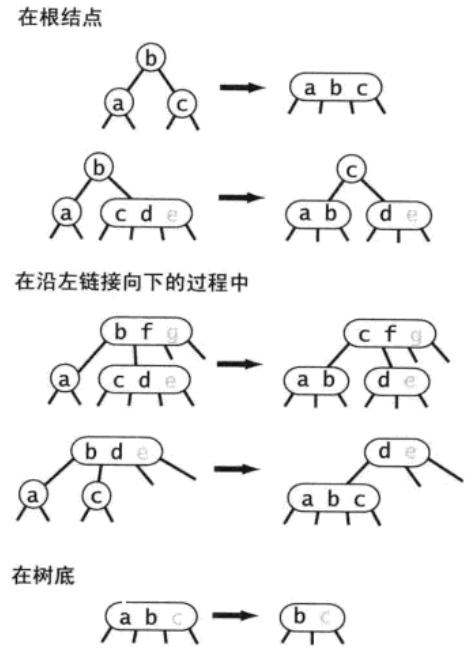


图 3.3.26 删除最小键操作中的变换

3.3.4.3 删除操作

在查找路径上进行和删除最小键相同的变换同样可以保证在查找过程中任意当前结点均不是2-结点。如果被查找的键在树的底部，我们可以直接删除它。如果不在，我们需要将它和它的后继结点交换，就和二叉查找树一样。因为当前结点必然不是2-结点，问题已经转化为在一棵根结点不是2-结点的子树中删除最小的键，我们可以在这棵子树中使用前文所述的算法。和以前一样，删除之后我们需要向上回溯并分解余下的4-结点。

本节末尾的练习中有几道是关于这些删除算法的例子和实现的。有兴趣理解或实现删除算法的读者应该掌握这些练习中的细节。对算法研究感兴趣的读者应该认识到这些方法的重要性，因为这是我们见过的第一种能够同时实现高效的查找、插入和删除操作的符号表实现。下面我们将会验证这一点。

442 ~ 443

3.3.5 红黑树的性质

研究红黑树的性质就是要检查对应的2-3树并对相应的2-3树进行分析的过程。我们的最终结论是*所有基于红黑树的符号表实现都能保证操作的运行时间为对数级别*（范围查找除外，它所需的额外时间和返回的键的数量成正比）。我们重复并强调这一点是因为它十分重要。

3.3.5.1 性能分析

首先，无论键的插入顺序如何，红黑树都几乎是完美平衡的（请见图3.3.27）。这从它和2-3树的一一对应关系以及2-3树的重要性质可以得到。

命题G。一棵大小为N的红黑树的高度不会超过$2\lg N$。

简略的证明。红黑树的最坏情况是它所对应的2-3树中构成最左边的路径结点全部都是3-结点而其余均为2-结点。最左边的路径长度是只包含2-结点的路径长度（$\sim \lg N$）的两倍。要按照某种顺序构造一棵平均路径长度为$2\lg N$的最差红黑树虽然可能，但并不容易。如果你喜欢数学，你也许会喜欢在练习3.3.24中探究这个问题的答案。

这个上界是比较保守的。使用随机的键序列和典型应用中常见的键序列进行的实验都证明，在一棵大小为 N 的红黑树中一次查找所需的比较次数约为（$1.00\lg N-0.5$）。另外，在实际情况下你不太可能遇到比这个数字高得多的平均比较次数，如表 3.3.1 所示。

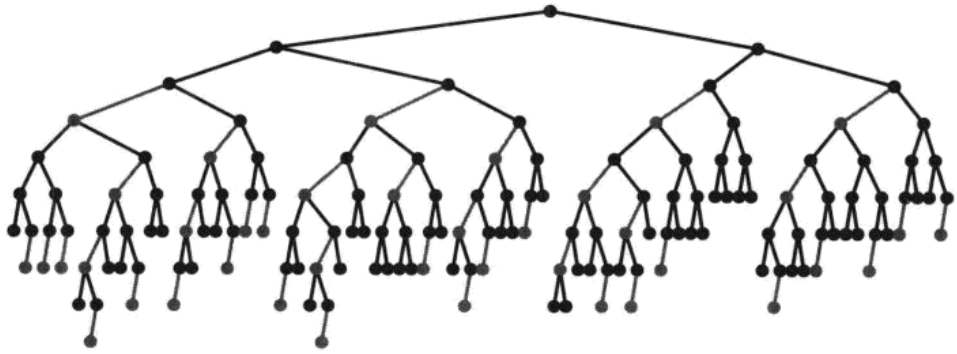


图 3.3.27 使用随机键构造的典型红黑树，没有画出空链接（另见彩插） 444

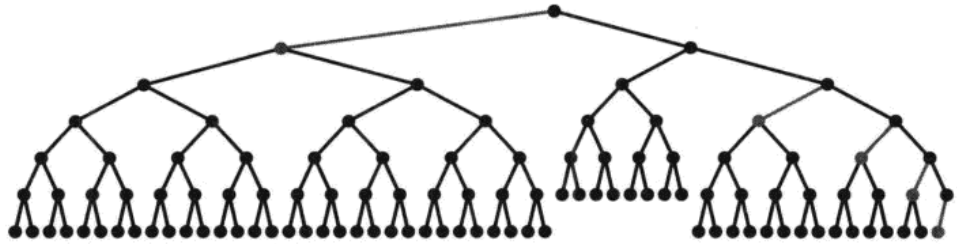
表 3.3.1 使用 RedBlackBST 的 FrequencyCounter 的每次 put() 操作平均所需的比较次数

	tale.txt				leipzig1M.txt			
	单词数	不同单词数	比较次数		单词数	不同单词数	比较次数	
			模型预测	实际次数			模型预测	实际次数
所有单词	135 635	10 679	13.6	13.5	21 191 455	534 580	19.4	19.1
长度大于等于 8 的单词	14 350	5 737	12.6	12.1	4 239 597	299 593	18.7	18.4
长度大于等于 10 的单词	4 582	2 260	11.4	11.5	1 610 829	165 555	17.5	17.3

命题 H。一棵大小为 N 的红黑树中，根结点到任意结点的平均路径长度为 ~ $1.00\lg N$。

例证。和典型的二叉查找树（例如表 3.2.1 中所示的树）相比，一棵典型的红黑树的平衡性是很好的，例如图 3.3.27 所示（甚至是图 3.3.28 中由升序键列构造的红黑树）。表 3.3.1 显示的数据表明 FrequencyCounter 在运行中构造的红黑树的路径长度（即查找成本）比初等二叉查找树低 40% 左右，和预期相符。自红黑树的发明以来，无数的实验和实际应用都印证了这种性能改进。

以使用 FrequencyCounter 在处理长度大于等于 8 的单词时 put() 操作的成本为例，我们可以看到平均成本降低得更多（如图 3.3.29 所示）。这又一次验证了理论模型所预测的对数级别的运行时间，只不过这次的惊喜比二叉查找树的小，因为性质 G 已经向我们保证了这一点。节约的总成本低于在查找上节约的 40% 的成本，因为除了比较我们也统计了旋转和颜色变换的次数。



445 图 3.3.28 使用升序键列构造的一棵红黑树，没有画出空链接（另见彩插）

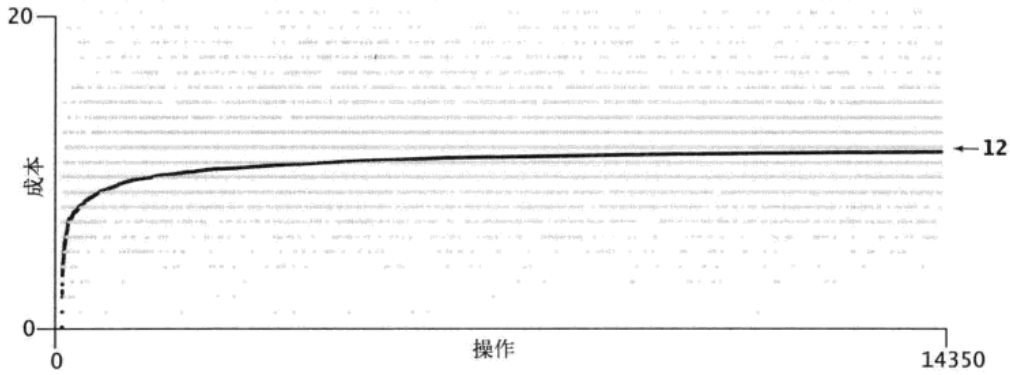


图 3.3.29 使用 RedBlackBST，运行 java FrequencyCounter 8 < tale.txt 的成本

红黑树的 get() 方法不会检查结点的颜色，因此平衡性相关的操作不会产生任何负担；因为树是平衡的，所以查找比二叉查找树更快。每个键只会被插入一次，但却可能被查找无数次，因此最后我们只用了很小的代价（和二分查找不同，我们可以保证插入操作是对数级别的）就取得了和最优情况近似的查找时间（因为树是接近完美平衡的，且查找过程中不会进行任何平衡性的操作）。查找的内循环只会进行一次比较并更新一条链接，非常简短，和二分查找的内循环类似（只有比较和索引运算）。这是我们见到的第一个能够保证对数级别的查找和插入操作的实现，它的内循环更紧凑。它通过了各种应用的考验，包括许多库实现。

3.3.5.2 有序符号表 API

红黑树最吸引人的一点是它的实现中最复杂的代码仅限于 put()（和删除）方法。二叉查找树中的查找最大和最小键、select()、rank()、floor()、ceiling() 和范围查找方法不做任何变动即可继续使用，因为红黑树也是二叉查找树而这些操作也不会涉及结点的颜色。算法 3.4 和这些方法（以及删除方法）一起完整地实现了我们的有序符号表 API。这些方法都能从红黑树近乎完美的平衡性中受益，因为它们最多所需的时间都和树高成正比。因此命题 G 和命题 E 一起保证了所
446 有操作的运行时间是对数级别的 。

命题 I。在一棵红黑树中，以下操作在最坏情况下所需的时间是对数级别的：查找（get()）、插入（put()）、查找最小键、查找最大键、floor()、ceiling()、rank()、select()、删除最小键（deleteMin()）、删除最大键（deleteMax()）、删除（delete()）和范围查询（range()）。

证明。我们已经讨论过 put()、get() 和 delete() 方法。对于其他方法，代码可以从 3.2 节中照搬（它们不涉及结点颜色）。命题 G 和命题 E 可以保证算法是对数级别的，所有操作在所经过的结点上只会进行常数次数的操作也说明了这一点。

各种符号表实现的性能总结如表 3.3.2 所示。

表 3.3.2 各种符号表实现的性能总结

算法（数据结构）	最坏情况下的运行时间的增长数量级（N 次插入之后）		平均情况下的运行时间的增长数量级（N 次插入之后）		是否支持有序性相关的操作
	查找	插入	查找	插入	
顺序查询（无序链表）	N	N	$N/2$	N	否
二分查找（有序数组）	$\lg N$	N	$\lg N$	$N/2$	是
二叉树查找（BST）	N	N	$1.39\lg N$	$1.39\lg N$	是
2-3 树查找（红黑树）	$2\lg N$	$2\lg N$	$1.00\lg N$	$1.00\lg N$	是

想想看，这样的保证是一个非凡的成就。在信息世界的汪洋大海中，表的大小可能上千亿，但我们仍能够确保在几十次比较之内就完成这些操作。 447

答疑

问 为什么不允许存在红色右链接和 4- 结点？

答 它们都是可用的，并且已经应用了几十年了。在练习中你会遇到它们。只允许红色左链接的存在能够减少可能出现的情况，因此实现所需的代码会少得多。

问 为什么不在 Node 类型中使用一个 Key 类型的数组来表示 2- 结点、3- 结点和 4- 结点？

答 问得好。这正是我们在 B- 树（请见第 6 章）的实现中使用的方案，它的每个结点中可以保存更多的键。因为 2-3 树中的结点较少，数组所带来的额外开销太高了。

问 在分解一个 4- 结点时，我们有时会在 rotateRight() 中将右结点的颜色设为 RED（红）然后立即在 flipColors() 中将它的颜色变为 BLACK（黑）。这不是浪费时间吗？

答 是的，有时我们还会不必要地反复改变中结点的颜色。从整体来看，多余的几次颜色变换和将所有方法的运行时间的增长数量级从线性级别提升到对数级别不是一个级别的。当然，在有性能要求的应用中，你可以将 rotateRight() 和 flipColors() 的代码在所需要的地方展开来消除那些额外的开销。我们在删除中也会使用这两个方法。在能够保证树的完美平衡的前提下，它们更加容易使用、理解和维护。 448

练习

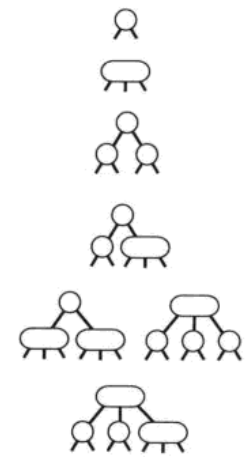
3.3.1 将键 E A S Y Q U T I O N 按顺序插入一棵空 2-3 树并画出结果。

3.3.2 将键 Y L P M X H C R A E S 按顺序插入一棵空 2-3 树并画出结果。

3.3.3 使用什么顺序插入键 S E A C H X M 能够得到一棵高度为 1 的 2-3 树？

3.3.4 证明含有 N 个键的 2-3 树的高度在 $\lfloor \log 3N \rfloor$ 即 $0.63 \lg N$（树完全由 3- 结点组成）和 $\lfloor \lg N \rfloor$（树完全由 2- 结点组成）之间。

3.3.5 右图显示了 N=1 到 6 之间大小为 N 的所有不同的 2-3 树（无先后次序）。请画出 N=7、8、9 和 10 的大小为 N 的所有不同的 2-3 树。

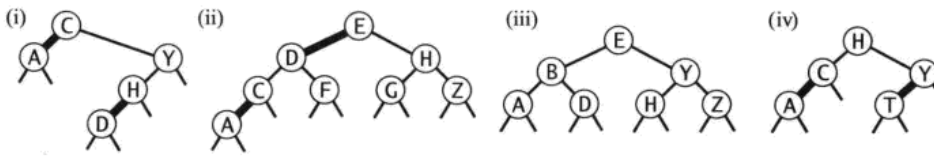


3.3.6 计算用 N 个随机键构造练习 3.3.5 中每棵 2-3 树的概率。

3.3.7 以图 3.3.8 为例为图中的其他 5 种情况画出相应的示意图。

3.3.8 画出使用三个 2- 结点和红链接一起表示一个 4- 结点的所有可能方法（不一定只能使用红色左链接）。

3.3.9 下图中哪些是红黑树（粗的链接为红色）？



3.3.10 将含有键 E A S Y Q U T I O N 的结点按顺序插入一棵空红黑树并画出结果。

449 3.3.11 将含有键 Y L P M X H C R A E S 的结点按顺序插入一棵空红黑树并画出结果。

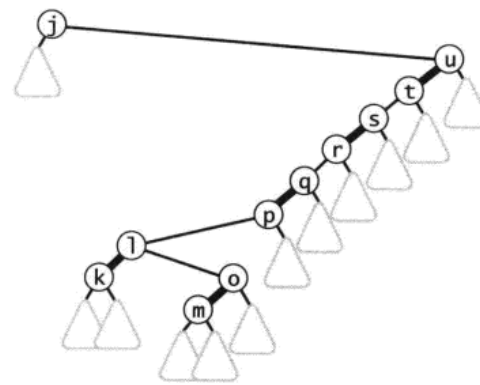
3.3.12 在我们的标准索引测试用例中插入键 P 并画出插入的过程中每次变换（颜色转换或是旋转）后的红黑树。

3.3.13 真假判断：如果你按照升序将键顺序插入一棵红黑树中，树的高度是单调递增的。

3.3.14 用字母 A 到 K 按顺序构造一棵红黑树并画出结果，然后大致说明在按照升序插入键来构造一棵红黑树的过程中发生了什么（可以参考正文中的图例）。

3.3.15 在键按照降序插入红黑树的情况下重新回答上面两道练习。

3.3.16 向右图所示的红黑树（黑色加粗部分的链接为红色）中插入 n 并画出结果（图中只显示了插入时的查找路径，你的解答中只需包含这些结点即可）。



3.3.17 随机生成两棵均含有 16 个结点的红黑树。画出它们（手绘或者代码绘制均可）并将它们和使用同一组键构造的（非平衡的）二叉查找树进行比较。

3.3.18 对于 2 到 10 之间的 N，画出所有大小为 N 的不同红黑树（请参考练习 3.3.5）。

3.3.19 每个结点只需要 1 位来保存结点的颜色即可表示 2- 结点、3- 结点和 4- 结点。使用二叉树，我们

在每个结点需要几位信息才能表示 5- 结点、6- 结点、7- 结点和 8- 结点？

3.3.20 计算一棵大小为 N 且完美平衡的二叉查找树的内部路径长度，其中 N 为 2 的幂减 1。

3.3.21 基于你为练习 3.2.10 给出的答案编写一个测试用例 TestRB.java。

3.3.22 找出一组键的序列使得用它顺序构造的二叉查找树比用它顺序构造的红黑树的高度更低，或者证明这样的序列不存在。 450

提高题

3.3.23 *没有平衡性限制的 2-3 树。*使用 2-3 树（不一定平衡）作为数据结构实现符号表的基本 API。树中的 3- 结点中的红链接可以左斜也可以右斜。树底部的 3- 结点和新结点通过黑色链接相连。实验并估计随机构造的这样一棵大小为 N 的树的平均路径长度。

3.3.24 *红黑树的最坏情况。*找出如何构造一棵大小为 N 的最差红黑树，其中从根结点到几乎所有空链接的路径长度均为 $2\lg N$。

3.3.25 *自顶向下的 2-3-4 树。*使用平衡 2-3-4 树作为数据结构实现符号表的基本 API。在树的表示中使用红黑链接并实现正文所述的插入算法，其中在沿查找路径向下的过程中分解 4- 结点并进行颜色转换，在回溯向上的过程中将 4- 结点配平。

3.3.26 *自顶向下一遍完成。*修改你为练习 3.3.25 给出的答案，不使用递归。在沿查找路径向下的过程中分解并平衡 4- 结点（以及 3- 结点），最后在树底插入新键即可。

3.3.27 *允许红色右链接。*修改你为练习 3.3.25 给出的答案，允许红色右链接的存在。

3.3.28 *自底向上的 2-3-4 树。*使用平衡 2-3-4 树作为数据结构实现符号表的基本 API。在树的表示中使用红黑链接并用和算法 3.4 相同的递归方式实现自底向上的插入。你的插入方法应该只需要分解查找路径底部的 4- 结点（如果有的话）。

3.3.29 *最优存储。*修改 RedBlackBST 的实现，用下面的技巧实现无需为结点颜色的存储使用额外的空间：要将结点标记为红色，只需交换它的左右链接。要检测一个结点是否是红色，检测它的左子结点是否大于它的右子结点。你需要修改一些比较语句来适应链接的交换。这个技巧将变量的比较变成了键的比较，显然成本会更高，但它说明在需要的情况下这个变量是可以被删掉的。

3.3.30 *缓存。*修改 RedBlackBST 的实现，将最近访问的结点 Node 保存在一个变量中，这样 get() 或 put() 在再次访问同一个键时就只需要常数时间了（请参考练习 3.1.25）。 451

3.3.31 *树的绘制。*为 RedBlackBST 添加一个 draw() 方法，像正文一样绘制出红黑树。

3.3.32 *AVL 树。*AVL 树是一种二叉查找树，其中任意结点的两棵子树的高度最多相差 1（最早的平衡树算法就是基于使用旋转保持 AVL 树中子树高度的平衡）。证明将其中由高度为偶数的结点指向高度为奇数的结点的链接设为红色就可以得到一棵（完美平衡的）2-3-4 树，其中红色链接可以是右链接。*附加题：*使用 AVL 树作为数据结构实现符号表的 API。一种方法是在每个结点中保存它的高度并在递归调用后使用旋转来根据需要调整这个高度；另一种方法是在树的表示中使用红黑链接并使用类似练习 3.3.39 和练习 3.3.40 的 moveRedLeft() 和 moveRedRight() 的方法。

3.3.33 *验证。*为 RedBlackBST 实现一个 is23() 方法来检查是否存在同时和两条红链接相连的结点和红色右链接，以及一个 isBalanced() 方法来检查从根结点到所有空链接的路径上的黑链接的数量是否相同。将这两个方法和练习 3.2.32 的 isBST() 方法结合起来实现一个 isRedBlackBST() 来检查一棵树是否是红黑树。