- 放松边 4→5 并将顶点 5 加入队列。放松失效的边 4→7。
- 放松边 5→1 并将顶点 1 加入队列。放松失效的边 5→4 和 5→7。
- 放松无效的边 1→3。队列为空。

在这个例子中，最短路径树就是一条从顶点 0 到顶点 1 的路径。从顶点 4、5 和 1 指出的所有边都被放松了两次。对照这个例子重读命题 X 的证明能够帮助你更好的理解这个算法。

4.4.6.8 负权重环的检测

实现 BellmanFordSP 会检测负权重环来避免陷入无限的循环中。我们也可以将这段检测代码独立出来使得用例可以检查并得到负权重环。因此我们为表 4.4.4 中的 API 添加以下方法请见表 4.4.8。

表 4.4.8 为处理负权重环扩展最短路径的 API

`boolean hasNegativeCycle()`	是否含有负权重环
`Iterable<DirectedEdge> negativeCycle()`	得到负权重环（如果没有则返回 null）

实现这些方法并不困难，如以下代码所示。在 BellmanFordSP 的构造函数运行之后，命题 Y 说明在将所有边放松 V 轮之后当且仅当队列非空时有向图中才存在从起点可达的负权重环。如果是这样，edgeTo[] 数组所表示的子图中必然含有这个负权重环。因此，要实现 negativeCycle()，会根据 edgeTo[] 中的边构造一幅加权有向图并在该图中检测环。我们会使用并修改 4.2 节中的 DirectedCycle 类来在加权有向图中寻找环（请见练习 4.4.12）。这种检查的成本分为以下几个部分。

- 添加一个变量 cycle 和一个私有函数 findNegative-Cycle()。如果找到负权重环，该方法会将 cycle 的值设为含有环中所有边的一个迭代器（如果没有找到则设为 null）。
- 每调用 V 次 relax() 方法后即调用 findNegativeCycle() 方法。

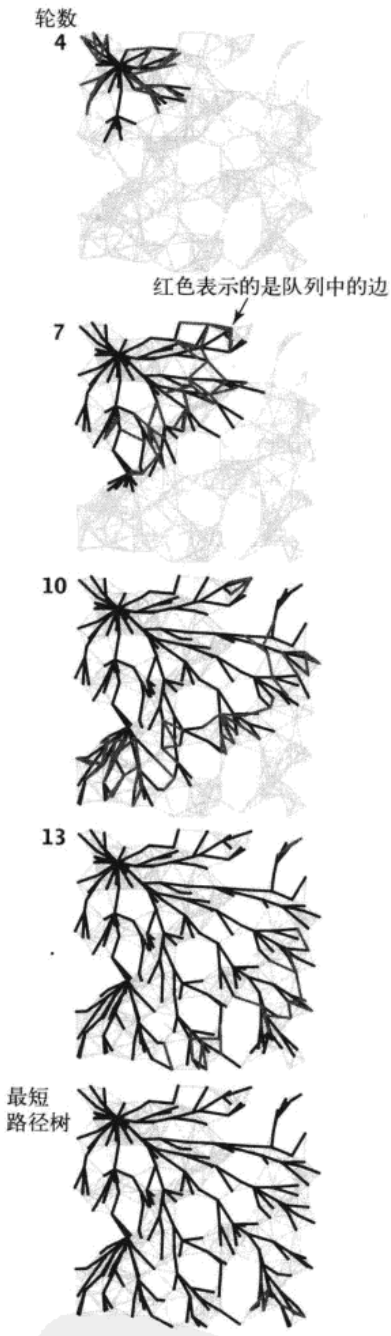


图 4.4.23 Bellman-Ford 算法（250 个顶点）（另见彩插）

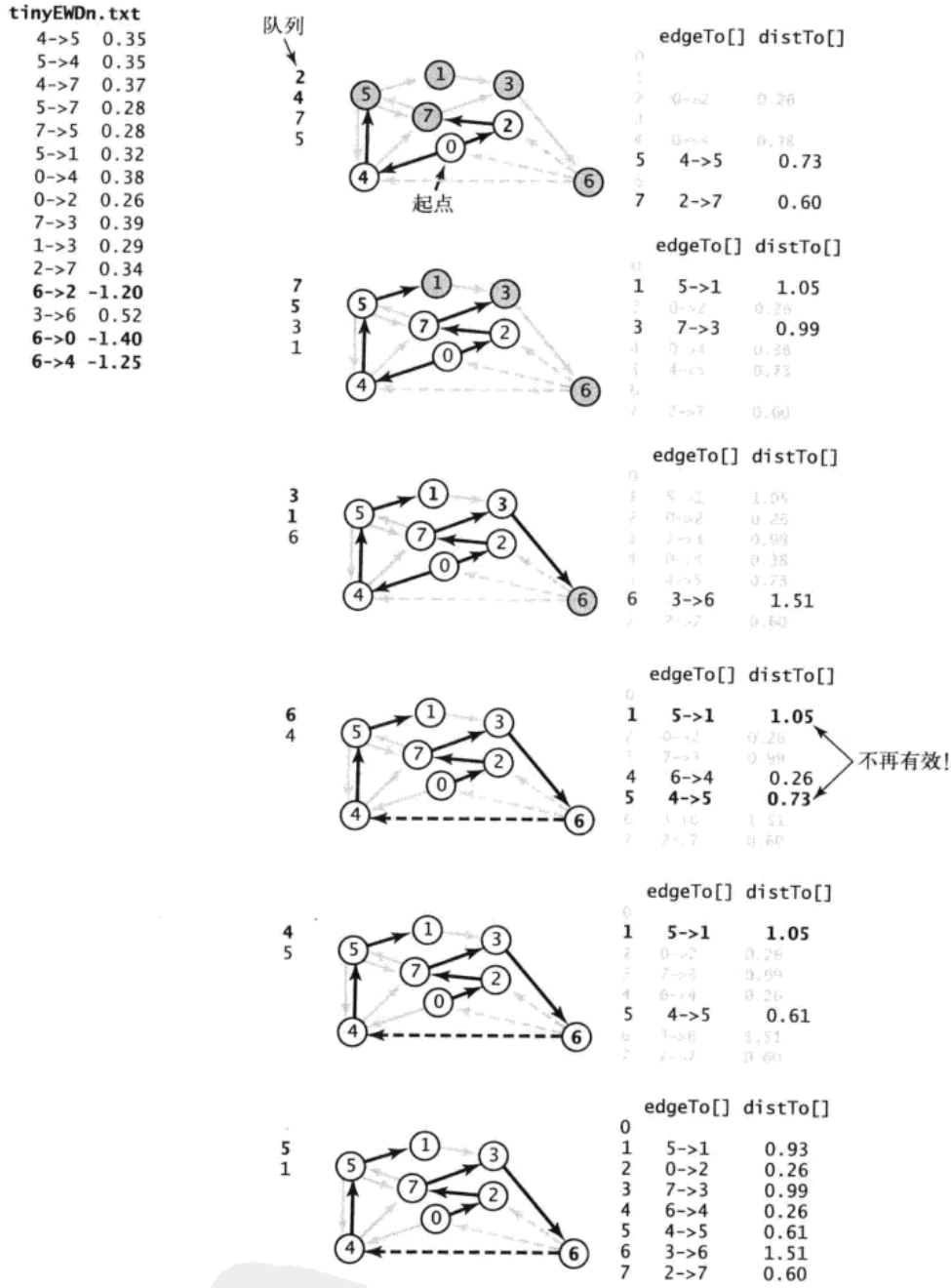


图 4.4.24 Bellman-Ford 算法的轨迹（图中含有负权重边）（另见彩插）

这种方法能够保证构造函数中的循环必然会终止。另外，用例可以调用 `hasNegativeCycle()` 来判断是否存在从起点可达的负权重环（并用 `negativeCycle()` 来获取这个环）。要在任意有向图中检测负权重环的存在只需稍作扩展即可（请见练习 4.4.43）。

图 4.4.25 是 Bellman-Ford 算法在一幅含有负权重环的有向图中的运行轨迹。头两轮放松操作与处理 tinyEWDn.txt 时是一样的。在第三轮中，算法在放松了边 7→3 和 5→1 并将顶点 3 和 1 加入队列后开始放松负权重边 5→4。在这次放松操作中算法发现了一个负权重环 4→5→4。它将 5→4 加入最短路径树中并在 edgeTo[] 中将环和起点隔离开来。从这时开始，算法沿着环继续运行并会减少到达所遇到的所有顶点的距离，直至检测到环的存在，此时队列非空。环被保存在 edgeTo[] 中，findNegativeCycle() 会在其中找到它。

```
private void findNegativeCycle()
{
   int V = edgeTo.length;
   EdgeWeightedDigraph spt;
   spt = new EdgeWeightedDigraph(V);
   for (int v = 0; v < V; v++)
      if (edgeTo[v] != null)
         spt.addEdge(edgeTo[v]);

   EdgeWeightedCycleFinder cf;
   cf = new EdgeWeightedCycleFinder(spt);

   cycle = cf.cycle();
}

public boolean hasNegativeCycle()
{  return cycle != null;  }

public Iterable<Edge> negativeCycle()
{  return cycle;  }
```

Bellman-Ford算法的负权重环检测方法

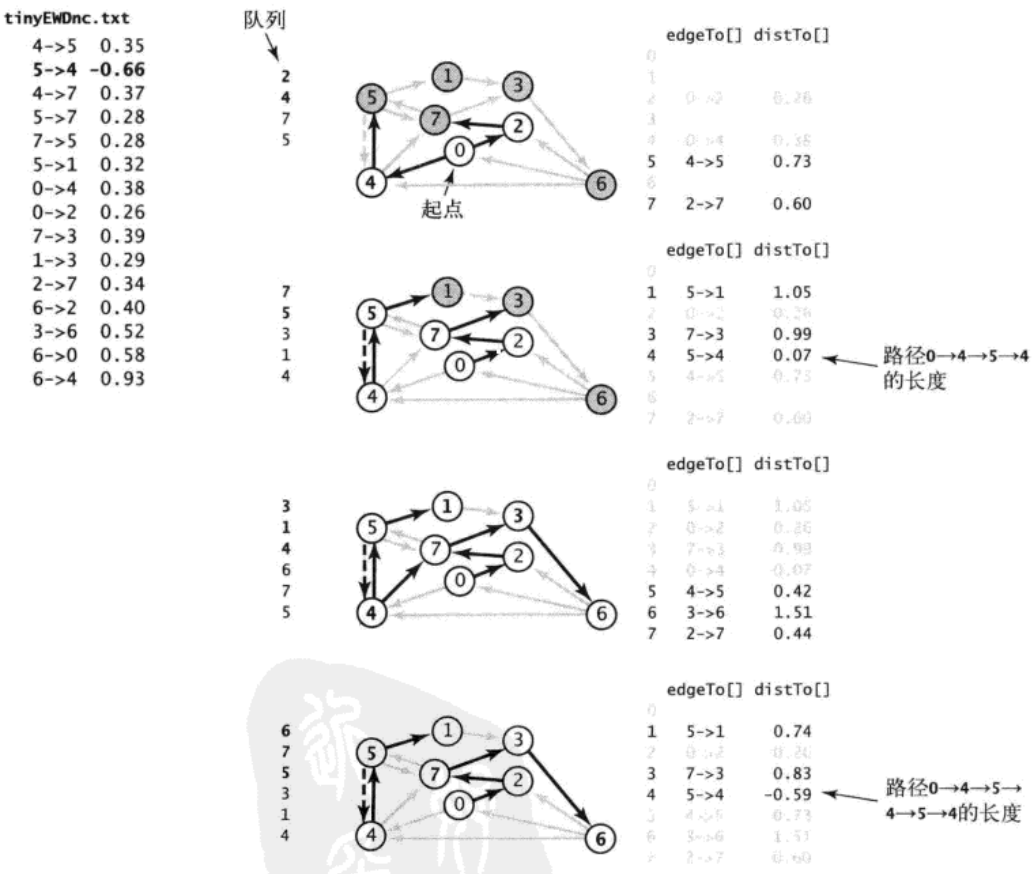


图 4.4.25 Bellman-Ford 算法的轨迹（含有负权重环的图，另见彩插）

675 ~ 678

4.4.6.9 套汇

假设有一个基于商品贸易的金融交易市场。以下框注显示的是示例文件 rates.txt 的内容，你可以在任意货币兑换比例的表格中找到类似的内容。文件的第一行是货币的种类数 V，接下来的每一行都对应一种货币，开头是该货币的名称，紧接着是它和其他货币兑换的汇率。简单起见，这个例子中只包含了能够在现代市场中进行交易的数百种货币中的五种：美元（USD）、欧元（EUR）、英镑（GBP）、瑞士法郎（CHF）和加元（CAD）。第 s 行的第 t 个数字表示一个汇率，即购买一个单位的第 s 行的货币需要多少个单位的第 t 行的货币。例如，这张表告诉我们，1000 美元能够购买 741 欧元。这张表格等价于一幅完全的加权有向图，顶点对应着货币，边则对应着汇率。权重为 x 的边 s→t 表示从货币 s 到货币 t 的汇率为 x。这张图中的路径则表示多次兑换。例如，将权重为 y 的边 t→u 和刚才的边结合起来就得到了一条路径 s→t→u，即一个单位的货币 s 可以兑换为 xy 个单位的货币 u。比如，欧元可以兑换得到 1012.206=741*1.366 加元。注意，这比直接用美元兑换的汇率更高。你可能会以为 xy 总是应该等于边 s→u 的权重，但这张表格所表示的金融系统非常复杂，并不总是能够保证这种一致性。因此，找到所有从 s 到 u 的路径中所有边的权重之积最大者就是我们最感兴趣的问题。一种更有趣的情况是，所有边的权重之积小于从终点指向起点的边的权重。在这个示例中，假设边 u→s 的权重为 z 且 xyz>1。那么环 s→t→u→s 就能够用一个单位的货币 s 得到多于一个单位（xyz）的货币 s。换句话说，将货币 s 兑换为 t、u 并最后再兑换为 s 就可以得到 100(xyz-1) 的利润。例如，如果将 1012.206 加元重新兑换为美元，可以得到 1012.206*0.995=1007.14497 美元，也就是得到了 7.14497 美元的利润。这看起来似乎不多，但一个外汇交易商可能会用一百万美元并在每分钟都进行一遍这样的交易，也就是说他每分钟的利润将超过 7000 美元，或者说每小时的利润超过 420 000 美元！这就是套汇交易的一个例子，请见图 4.4.26。如果没有外力的限制，比如手续费或是交易金额上限，交易商可以从其中获取无限的利润。即使是在现实世界中的这些限制下，套汇的利润仍然是非常高的。这个问题和最短路径问题有什么关系呢？要回答这个问题非常简单。

```
% more rates.txt
5
USD  1      0.741  0.657  1.061  1.005
EUR  1.349  1      0.888  1.433  1.366
GBP  1.521  1.126  1      1.614  1.538
CHF  0.942  0.698  0.619  1      0.953
CAD  0.995  0.732  0.650  1.049  1
```

命题 Z。套汇问题等价于加权有向图中的负权重环的检测问题。

证明。取每条边权重的自然对数并取反，这样在原始问题中所有边的权重之积的计算就转化为了新图中所有边的权重之和的计算。任意权重之积 $w_1w_2\cdots w_k$ 即对应 $-\ln(w_1)-\ln(w_2)-\cdots-\ln(w_k)$ 之和。转换后边的权重可能为正也可能为负。一条从 v 到 w 的路径表示将货币 v 兑换为货币 w，图中的任意负权重环都是一次套汇的好机会（请见图 4.4.27）。

在这个示例中，货币可以任意兑换，因此有向图是完全的，任意负权重环都是从任意顶点可达的。在一般的商品交易中，有些边可能并不存在，因此需要练习 4.4.43 所述的只有一个参数的构造函数。目前没有已知的寻找最佳套汇机会（图中负权重最小的环）的高效算法（图的规模不需要很大就能使所需的计算量超过计算机的承受能力），但找出任意套汇机会的最快算法仍然是很重要的——在第二快的算法找到任何套汇机会之前，使用这种算法的商人很可能已经可以系统地排除许多不佳的

套汇机会了。

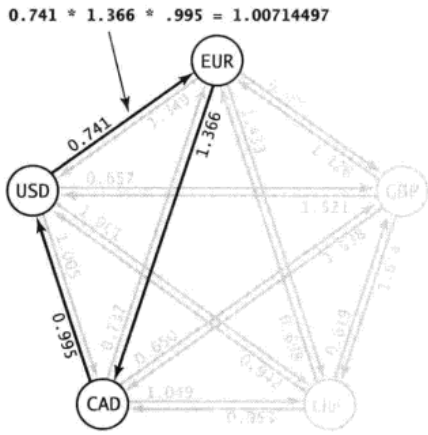


图 4.4.26 一次套汇机会

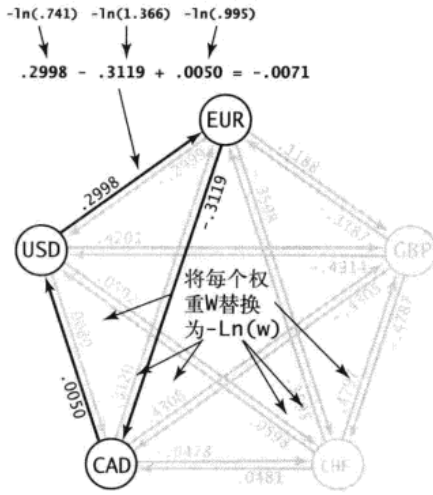


图 4.4.27 一个负权重环就表示了一次套汇的机会

货币兑换中的套汇

```
public class Arbitrage
{
   public static void main(String[] args)
   {
      int V = StdIn.readInt();
      String[] name = new String[V];
      EdgeWeightedDigraph G = new EdgeWeightedDigraph(V);
      for (int v = 0; v < V; v++)
      {
         name[v] = StdIn.readString();
         for (int w = 0; w < V; w++)
         {
            double rate = StdIn.readDouble();
            DirectedEdge e = new DirectedEdge(v, w, -Math.log(rate));
            G.addEdge(e);
         }
      }

      BellmanFordSP spt = new BellmanFordSP(G, 0);
      if (spt.hasNegativeCycle())
      {
         double stake = 1000.0;
         for (DirectedEdge e : spt.negativeCycle())
         {
            StdOut.printf("%10.5f %s", stake, name[e.from()]);
            stake *= Math.exp(-e.weight());
            StdOut.printf("= %10.5f %s\n", stake, name[e.to()]);
         }
      }
      else StdOut.println("No arbitrage opportunity");
   }
}
```

这段代码调用了 BellmanFordSP 类来寻找汇率表中的套汇机会。它首先使用完全有向图表示汇率表，然后用 Bellman-Ford 算法来寻找图中的负权重环。

```
% java Arbitrage < rates.txt
1000.00000 USD =  741.00000 EUR
 741.00000 EUR = 1012.20600 CAD
1012.20600 CAD = 1007.14497 USD
```

命题 Z 的证明即使在没有套汇机会的情况下仍然有用，因为它将货币兑换问题转化为了一个最短路径问题。因为对数函数是单调的（且会对计算的结果取反），当边的权重之和最小时汇率之积正好最大。尽管边的权重可正可负，从 v 到 w 的最短路径仍然是将货币 v 兑换为货币 w 的最好方法。

679 ~ 681

4.4.7 展望

表 4.4.9 总结了本节中我们所学习到的各种最短路径算法的重要性质。在这些算法中进行选择的第一个条件是问题所涉及的有向图的基本性质。它含有负权重的边吗？它含有环吗？它含有负权重的环吗？除了这些基本性质之外，加权有向图的特性多种多样，因此在有多个合适的选择时就需要通过实验找出最佳的算法。

表 4.4.9 最短路径算法的性能特点

算法	局限	路径长度的比较次数（增长的数量级）		所需空间	优势
		一般情况	最坏情况		
Dijkstra 算法（即时版本）	边的权重必须为正	$E\log V$	$E\log V$	V	最坏情况下仍有较好的性能
拓扑排序	只适用于无环加权有向图	$E+V$	$E+V$	V	是无环图中的最优算法
Bellman-Ford 算法（基于队列）	不能存在负权重环	$E+V$	VE	V	适用领域广泛

历史资料

自 20 世纪 50 年代以来，最短路径算法就已经被深入地研究并被广泛应用了。计算最短路径的 Dijkstra 算法的历史和计算最小生成树的 Prim 算法的历史背景相似（并且也相关）。Dijkstra 算法既指的是按照顶点距离起点的远近顺序构造最短路径树的算法，也指的是该算法的实现，（它也是最适合用临接矩阵表示的算法。），因为 Dijkstra 在 1959 年的一篇论文中发表了上述观点（并且证明了这种方法同样也可以用来计算最小生成树）。稀疏图算法的性能改进来自于之后对优先队列实现的改进，不仅仅针对最短路径问题。这其中最重要的是 Dijkstra 算法性能的改进。（例如，使用斐波那契堆后最坏情况下的复杂度可以提高到 $E+V\log V$）。实践证明 Bellman-Ford 算法十分有效并且应用领域广泛，特别是处理一般性的加权有向图。由于 Bellman-Ford 算法计算普通应用的运行时间常常是线性的，因此在最坏情况下它的运行时间是 VE。最坏情况下的运行时间为线性级别的稀疏图的最短路径算法是一个仍在研究之中的问题。Bellman-Ford 算法最早由 L.Ford 和 R.Bellman 发表于 20 世纪 50 年代。尽管我们已经看到许多其他的图算法性能得到了大幅改进，但是处理含有负权重边（但不含负权重环）的且在最坏情况下性能更好的有向图算法还没有出现。

682 ~ 683

答疑

问 为什么要分别为无向图、有向图、加权无向图、加权有向图定义不同的数据类型?

答 这么做是为了使用例代码更清晰，同时也是为了更加简洁和高效地实现没有权重的图。在需要处理各种图的应用或系统中，软件工程中的标准做法就是先定义一种抽象数据结构并根据它衍生出其他抽象数据结构，也就是4.1节中学习的无向图Graph，4.2节中学习的有向图Digraph，4.3节中学习的加权无向图EdgeWeightedGraph，或是在本节中学习的加权有向图EdgeWeightedDigraph。

问 如何在（加权）无向图中找到最短路径?

答 对于边的权重均为正的图，Dijkstra算法可以解决这个问题。只需根据给定的EdgeWe-ightedGraph构造一幅EdgeWeightedDigraph（无向图中的每条边都对应着有向图中的两条方向不同的边）并执行Dijkstra算法即可。如果边的权重可能为负，高效的算法也是存在的，但它们比Bellman-Ford算法更复杂。

684

练习

4.4.1 真假判断：将每条边的权重都加上一个常数不会改变单点最短路径问题的答案。

4.4.2 为EdgeWeightedDigraph类实现toString()方法。

4.4.3 为稠密图实现一种使用邻接矩阵表示法（用二维数组保存边的权重，请参考练习4.3.9）的EdgeWeightedDigraph类。忽略平行边。

4.4.4 从tinyEWD.txt中（请见图4.4.4）删去顶点7并给出加权有向图中以顶点0为起点的最短路径树，使用父链接数组表示这棵树。将图中所有边的方向反转并回答相同的问题。

4.4.5 在tinyEWD.txt中（请见图4.4.4）改变边0→2的方向。画出该加权有向图中以顶点2为起点的两棵不同的最短路径树。

4.4.6 给出用即时版本的Dijkstra算法计算练习4.4.5所定义的图的最短路径树的轨迹。

4.4.7 实现DijkstraSP的另一个版本，支持一个方法来返回一幅加权有向图中从s到t的另一条最短路径。（如果从s到t的最短路径只有一条则返回null。）

4.4.8 一幅有向图的直径指的是连接任意两个顶点的所有最短路径中的最大长度。编写一个DijkstraSP的用例，找出边的权重非负的给定EdgeWeightedDigraph图的直径。

4.4.9 表4.4.10来自于一张很早以前出版的公路地图，它显示的是城市之间的最短路径的长度。这张表中有一个错误。改正这个错误并新建一张表来说明最短路径是哪条。

4.4.10 将练习4.4.4中定义的有向图看作无向图，该无向图中的每条边对应有向图中的两条方向不同但权重相同的边。为对应的有向图回答练习4.4.6中的问题。

4.4.11 使用1.4节中的内存使用模型评估用EdgeWeightedDigraph表示一幅含有V个顶点和E条边的图所需的内存。

4.4.12 修改4.2节中的DirectedCycle类和Topological类，使之使用本节中的EdgeWeightedDigraph类和DirectedEdge类的API并实现EdgeWeightedCycleFinder类和EdgeWeightedTopological类。

4.4.13 从tinyEWD.txt中（请见图4.4.4）删去边5→7，用Dijkstra算法计算所得的有向图的最短路径树并按照正文中的样式给出算法的轨迹。

表 4.4.10

	普罗维登斯	威斯特里	新伦敦	诺威治
普罗维登斯	—	53	54	48
威斯特里	53	—	18	101
新伦敦	54	18	—	12
诺威治	48	101	12	—

4.4.14 给出使用 4.4.6.1 节和 4.4.6.2 节的两种尝试处理图 4.4.19 的 tinyEWDn.txt 所得到的路径。

4.4.15 如果从顶点 s 到 v 的路径上存在一个负权重环，调用 Bellman-Ford 算法的 `pathTo(v)` 方法会发生什么？

4.4.16 假设用 `EdgeWeightedGraph` 中的每条边 `Edge` 都替换为两条（两个方向各一条）`DirectedEdge` 的方式将 `EdgeWeightedGraph` 类转化为 `EdgeWeightedDigraph` 类（如答疑中关于 Dijkstra 算法的部分所述）然后再使用 Bellman-Ford 算法处理它。说明为什么这种方法大错特错。

4.4.17 在 Bellman-Ford 算法中如果一个顶点在同一轮中被两次加入队列会发生什么？

解答：算法所需的运行时间将会达到指数级。例如，描述一幅边的权重全部为 –1 的加权有向完全图中 Bellman-Ford 算法的执行情况。

4.4.18 编写一个 CPM 的用例来打印出所有的关键路径。

4.4.19 找出正文中的例子里权重最低的环（即最佳套汇机会）。

4.4.20 从网上或者报纸上找到一张汇率表并用它构造一张套汇表。*注意*：不要使用根据若干数据计算得出的汇率表，它们的精度有限。*附加题*：从汇率市场上赚点外快！

4.4.21 用 Bellman-Ford 算法计算练习 4.4.5 中的加权有向图的最短路径树并按照正文中的样式给出算法的轨迹。

685 ≀ 687

提高题

4.4.22 *顶点的权重*。证明，要得到顶点也有非负权重的加权有向图中的最短路径（路径的权重为路径上的顶点权重之和），可以通过构造一幅只有边含有权重的加权有向图解决。

4.4.23 *给定两点的最短路径*。设计并实现一份 API，使用 Dijkstra 算法的改进版本解决加权有向图中给定两点的最短路径问题。

4.4.24 *多起点最短路径*。设计并实现一份 API，使用 Dijkstra 算法解决加权有向图中的多起点最短路径问题，其中边的权重均为正：给定一组起点，找到相应的最短路径森林并实现一个方法为用例返回从任意起点到达每个顶点的最短路径。*提示*：添加一个伪顶点和从该顶点指向每个起点的一条权重为零的边，或者在初始化时将所有起点加入优先队列并将它们在 `distTo[]` 中对应的值均设为 0。

4.4.25 *两个顶点集合之间的最短路径*。给定一幅边的权重均为正的有向图和两个没有交集的顶点集 S 和 T，找到从 S 中的任意顶点到达 T 中的任意顶点的最短路径。你的算法在最坏情况下所需的时间应该与 $E\log V$ 成正比。

4.4.26 稠密图中的单点最短路径。实现另一个版本的 Dijkstra 算法，使之能够在与 V^2 成正比的时间内在一幅稠密的加权有向图中计算出给定顶点的最短路径树。请使用邻接矩阵法表示稠密图（请参考练习 4.4.3 和练习 4.3.29）。

4.4.27 欧拉图中的最短路径。已知图中的顶点均在平面上，修改 API 以提高 Dijkstra 算法的性能。

4.4.28 有向无环图中的最长路径。重新实现 AcyclicLP 类，根据命题 T 解决加权有向无环图中的最长路径问题。

4.4.29 一般最优性。完成命题 W 的证明，说明如果存在从 s 到 v 的有向路径且从 s 到 v 的任意路径上的所有顶点都不在任意负权重环上，那么必然存在一条从 s 到 v 的最短路径（提示：参考命题 P）。

4.4.30 含有负权重环的图中的任意顶点对之间的最短路径。参考 4.4.4.3 节框注“任意顶点对之间的最短路径”所实现的不含负权重环的图中任意顶点对之间的最短路径问题并设计一份 API。使用 Bellman-Ford 算法的一个变种来确定权重数组 pi[]，使得对于任意边 v→w，边的权重加上 pi[v] 和 pi[w] 之差的和非负。然后更新所有边的权重，使得 Dijkstra 算法可以在新图中找出所有的最短路径。

688

4.4.31 线图中任意顶点对之间的最短路径。给定一幅加权线图（无向连通图，除了两个端点度数为 1 之外所有顶点的度数为 2），给出一个算法在线性时间内对图进行预处理并在常数时间内返回任意两个顶点之间的最短路径。

4.4.32 启发式的父结点检查。修改 Bellman-Ford 算法，仅当顶点 v 在最短路径树中的父结点 edgeTo[v] 目前不在队列中时才访问 v。Cherkassky、Goldberg 和 Radzik 在实践中发现这种启发式的做法十分有帮助。证明这种方法能够正确的计算出最短路径且在最坏情况下的运行时间和 EV 成正比。

4.4.33 网格图中的最短路径。给定一个 $N \times N$ 的正整数矩阵，找到从 (0,0) 到 $(N-1, N-1)$ 的最短路径，路径的长度即为路径中所有正整数之和。在只能向右和向下移动的限制下重新解答这个问题。

4.4.34 单调最短路径。给定一幅加权有向图，找出从 s 到其他每个顶点的单调最短路径。如果一条路径上的所有边的权重是严格单调递增或递减的，那么这条路径就是单调的。这样的路径应该是简单的（不包含重复顶点）。提示：按照权重的升序放松所有边并找到一条最佳路径；然后按照权重的降序放松所有边再找到另一条最佳路径。

4.4.35 双调最短路径。给定一幅有向图，找到从 s 到其他每个顶点的双调最短路径（如果存在）。如果从 s 到 t 的路径上存在一个中间顶点 v 使得从 s 到 v 中的所有边的权重均严格单调递增且从 v 到 t 中的所有边的权重均严格单调递减，那么这就是一条双调路径。这样的路径应该是简单的（不包含重复顶点）。

4.4.36 邻居顶点。编写一个 SP 的用例，找出一幅给定加权有向图中和一个给定顶点的距离在 d 之内的所有顶点。你的算法所需的运行时间应该与由这些顶点和依附于它们的边组成的子图的大小以及 V（用于初始化数据结构）中的较大者成正比。

689

4.4.37 关键边。给出一个算法来找到给定的加权有向图中的一条边，删去这条边使得给定的两个顶点之间的最短距离的增加值最大。

4.4.38 敏感度。给定一幅加权有向图和一对顶点 s 和 t，编写一个 SP 的用例对该图中的所有边进行敏感度分析：计算一个 $V \times V$ 的布尔矩阵，对于任意的 v 和 w，当 v→ w 为加权有向图中的一条边且增加 v→w 的权重不会增加从 s 到 t 的最短路径的权重时，v 行 w 列的值为 true，否则为 false。

4.4.39 延时 Dijkstra 算法的实现。根据正文实现 Dijkstra 算法的延时版本。

4.4.40 瓶颈最短路径树。请证明一幅无向图中的一棵最小生成树等价于该图中的一棵瓶颈最短路径树：对于任意一对顶点 v 和 w，该树都含有一条连接它们的路径且其中的最长边是所有连接两点的路径中最短的。

4.4.41 双向搜索。基于算法 4.9 的代码为给定两点的最短路径问题实现一个类，但在初始化时将起点和终点都加入优先队列。这么做会使最短路径树从两个顶点同时开始生长，你的主要任务是决定两棵树相遇时应该怎么办。

4.4.42 最坏情况（Dijkstra 算法）。找出含有 V 个顶点和 E 条边的一组图，使得 Dijkstra 算法处理它们所需的运行时间为最坏情况。

4.4.43 负权重环的检测。假设为算法 4.11 加入了一个构造函数，它和已有的构造函数的区别仅在于不需要第二个参数并将 `distTo[]` 中的所有元素初始化为 0。证明，如果用例调用的是这个构造函数，那么当且仅当图中含有一个负权重环时，`hasNegativeCycle()` 才会返回 `true`。（`negativeCycle()` 会返回那个负权重环。）

解答：向原图添加一个新的起点以及从该起点指向所有其他顶点的权重为 0 的边。在一轮放松之后，`distTo[]` 中的所有元素的值均会变为 0，从新起点开始寻找一个负权重环和在原图中寻找负权重环是等价的。

4.4.44 最坏情况（Bellman-Ford 算法）。找出一组图，使得算法 4.11 的运行时间与 VE 成正比。 690

4.4.45 快速 Bellman-Ford 算法。对于边的权重为整数且绝对值不大于某个常数的特殊情况，给出一个解决一般的加权有向图中的单点最短路径问题的算法，其所需的运行时间低于线性对数级别。

4.4.46 动画。编写一段程序将 Dijkstra 算法用动画表现出来。 691

实验题

4.4.47 随机加权有向稀疏图。修改你为练习 4.3.34 给出的解答，随机指定每条边的方向。

4.4.48 随机加权有向欧拉图。修改你为练习 4.3.35 给出的解答，随机指定每条边的方向。

4.4.49 随机加权有向网格图。修改你为练习 4.3.36 给出的解答，随机指定每条边的方向。

4.4.50 负权重边 I。修改你的随机加权有向图生成器，通过调整比例将边的权重控制在在 x 和 y 之间（x 和 y 都在 –1 和 1 之间）。

4.4.51 负权重边 II。修改你的随机加权有向图生成器，将固定比例（此值由用例指定）的边的权重取反来生成负权重的边。

4.4.52 负权重边 III。编写一段程序，调用你的加权有向图生成器，尽可能为大范围的 V 和 E 值生成多幅加权有向图，保证图中大部分边的权重为负且只有若干个负权重环。 692

测试所有的算法并研究所有图的模型的所有参数是不现实的。请为下面的每一道题都编写一段程序来处理从输入得到的任意图。这段程序可以调用上面的任意生成器并对相应的图模型进行实验。你可以根据上次实验的结果自己作出判断来选择不同实验。陈述结果以及由此得出的任何结论。

4.4.53 预测。请估计你的计算机和程序系统使用 Dijkstra 算法在 10 秒钟之内能够计算出图中所有的最短路径的图的最大规模，其中 $E=10V$，误差在 10 倍以内。

4.4.54 延时的代价。对于各种图的模型，运行实验并根据经验比较 Dijkstra 算法的延时版本和即时版本的性能差异。

4.4.55 Johnson 算法。使用一个 d 向堆实现优先队列。对于各种加权有向图的模型，找到 d 的最优值。

4.4.56 套汇模型。实现一个模型来生成随机的套汇问题。目标是尽量生成与练习 4.4.20 中相似表格。

4.4.57 最后期限限制下的并行任务调度模型。实现一个模型来生成随机的最后期限限制下的并行任务调度问题。目标是尽量生成复杂但可以解决的问题。

693



第5章 字 符 串

我们通过交流成串的字符进行沟通，所以无数的重要而熟悉的应用软件都是基于字符串处理的。本章中，我们会考察一些经典算法，解决以下应用领域背后的计算问题。

信息处理。当你根据一个给定的关键字搜索网页时，就是在使用一个字符串处理应用程序。在现代世界中，可以说所有的信息都是用一系列字符串表示的，而对它们进行处理的都是非常重要的字符串处理应用程序。

基因组学。计算生物学家的一项工作就是根据密码子将 DNA 转换为由 4 个碱基（A、C、T 和 G）组成的（非常长的）字符串。近些年来人类构建起来的庞大的基因数据库已经能够描述各种活体器官，因此字符串处理已经成为了现在计算生物学研究的基石。

通信系统。无论你是在发送短信、电子邮件或是下载电子书，都是在将字符串从一个地方传送到另一个地方。以此为目标的字符串处理应用程序是字符串处理算法开发的源动力。

编程系统。程序是由字符串组成的。编译器、解释器等其他能够将程序转换为机器指令的软件都是使用复杂的字符串处理技术的重要应用软件。事实上，所有的书面语言都是由字符串表达的。另外，开发字符串处理算法的另一个动力来源在于形式语言理论，它研究的是对不同类型的字符串集合的描述。

这几个非常有意义的示例说明了字符串处理算法的重要性和应用领域的多样性。

694 ~ 695

本章的结构如下：在介绍了字符串的基本性质以后，我们会在 5.1 节和 5.2 节中再次遇到第 2 章和第 3 章学过的排序和查找 API。当使用字符串作为键时，能够利用键的特殊性质的算法将比之前学习过的算法更快更灵活。在 5.3 节中，我们会学习*子字符串查找算法*，包括由 Knuth、Morris 和 Pratt 发明的一个著名的算法。在 5.4 节中会介绍*正则表达式*，它是*模式匹配*问题的基础，是一个一般化了的子字符串查找问题，也是搜索工具 grep 的核心。这些经典的算法的基础是两个基本概念，分别叫做*形式语言*和*确定有限状态自动机*。5.5 节主要介绍了一个重要应用：*数据压缩*，即尝试将一个字符串的长度缩短到最小程度。

5.0.1 游戏规则

为了简洁高效，我们将使用 Java 的 `String` 类来表示字符串，但我们将有意识地尽量少使用该类的方法以使算法能够适用于其他字符串数据类型以及其他编程语言。我们已经在 1.2 节中详细介绍过各种字符串，这里简要回顾一下它们最主要的性质。

字符。`String` 是由一系列字符组成的。字符的类型是 `char`，可能有 2^{16} 个值。数十年以来，程序员的注意力都局限于 7 位 ASCII 码（请见表 5.5.4）或是 8 位扩展 ASCII 码表示的字符，但许多现代的应用程序都已经需要使用 16 位 Unicode 编码了。

不可变性。`String` 对象是不可变的，因此可以将它们用于赋值语句、作为函数的参数或是返

回值，而不用担心它们的值会发生变化。

索引。我们最常完成的操作就是从某个字符串中提取一个特定的字符，即 Java 的 String 类的 charAt() 方法。我们希望 charAt() 方法能够在常数时间内完成，就好像字符串是保存在一个 char[] 数组中一样。根据第 1 章中的讨论，这种期望是非常合理的。

长度。在 Java 中，String 类型的 length() 方法实现了获取字符串的长度的操作。同样，我们也希望 lengt() 方法能够在常数时间内完成，这也是合情合理的，尽管在某些编程环境中实现这一点并不容易。

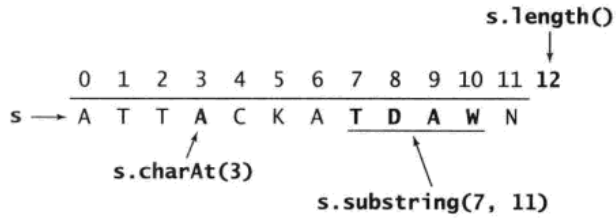


图 5.0.1 String 类型的基本常数时间操作

子字符串。Java 的 substring() 方法实现了提取特定的子字符串的操作。同样，我们也希望这个方法能够在常数时间内完成，Java 的标准实现也做到了这一点。*如果你还不熟悉 substring() 方法和为什么它只需要常数时间，请务必重新阅读 1.2 节中讨论的 Java 字符串的标准实现（请见表 1.2.7 和图 1.4.10）。*

字符串的连接。在 Java 中通过将一个字符串追加到另一个字符串的末尾创建一个新字符串的操作是一个内置的操作（使用“+”运算符），所需的时间与结果字符串的长度成正比。例如，我们会避免将字符一个一个地追加到字符串中，因为在 Java 里这个过程所需的时间将会是平方级别的（为此 Java 提供了一个 StringBuilder 类，请见图 5.0.1）。

字符数组。Java 的 String 类显然并不是一个原始数据类型。Java 的标准实现提供了刚才提到的几个操作以供客户端程序调用。但与之相反，我们将要学习的许多算法都能够处理字符串的低级表示，比如 char 类型的数组，而且许多字符串的用例程序也更愿意使用这种表示，因为它消耗的空间更小，访问所需的时间更少。在我们将要学习的几个算法中，将字符串从一种表示转换成另一种表示的代价甚至比算法的运行成本更高。如表 5.0.1 所示，处理这两种表示所用的代码的差别是很小的（substring() 方法比较复杂，此处省略），所以无论使用哪种表示方式都不会影响读者对算法的理解。

表 5.0.1 在 Java 中表示字符串的两种方法

操 作	字符数组	Java 字符串
声明	Char[] a	String s
根据索引访问字符	a[i]	s.charAt(i)
获取字符串长度	a.length	s.length()
表示方法转换	a=s.toCharArray();	s=new String(a);

理解这些操作的运行效率是理解许多字符串处理算法效率的关键部分。并不是所有编程语言实现的 String 类都能有这样的性能。例如，查找子字符串和获取字符串长度的操作在 C 语言中所需的时间就与字符串中的字符数量成正比。修改我们的算法并使之适用于这样的编程语言是完全可以的（实现一个类似 Java 的 String 类的抽象数据类型），但这也意味着不同的挑战和机遇。

696
l
697

在正文中，我们主要会使用 String 数据类型。我们会经常调用通过索引访问字符串中的字符操作和获取字符串长度的操作，有时会使用提取子字符串或是连接字符串的操作。我们还会在

本书的网站上提供相应的使用 char 数组的代码。在性能优先的应用场景中，用例在这两种表示方法之间权衡的常常是访问字符的成本（在一般的 Java 实现中，a[i] 很可能比 s.charAt(i) 要快很多）。

5.0.2 字母表

一些应用程序可能会对字符串的字母表作出限制。在这些应用中，可能常常会需要一个 API 如表 5.0.2 所示的 Alphabet 类。

表 5.0.2 字母表的 API

public class **Alphabet**	
Alphabet(String s)	根据 s 中的字符创建一张新的字母表
char toChar(int index)	获取字母表中索引位置的字符
int toIndex(char c)	获取 c 的索引，在 0 到 R–1 之间
boolean contains(char c)	c 在字母表之中吗
int R()	基数（字母表中的字符数量）
int lgR()	表示一个索引所需的位数
int[] toIndices(String s)	将 s 转换为 R 进制的整数
String toChars(int[] indices)	将 R 进制的整数转换为基于该字母表的字符串

这份 API 定义了一个构造函数，它用一个含有 R 个字符的字符串参数指定了字母表。API 定义了 toChar() 方法和 toIndex() 方法来在字符和 0 到 R–1 之间的整型值进行转换（常数时间）。它还包含了 contains() 方法来检查给定的字符是否存在于字母表中。方法 R() 和 lgR() 用来获取字母表中的字符数以及表示它们所需的位数。toIndices() 方法和 toChars() 方法能够将由字母表中的字符组成的字符串与 int 数组相互转换。方便起见，下面的表格显示了各种内置的字母表，你可以通过类似 Alphabet.UNICODE 的方式来访问它们。Alphabet 的实现很简单，我们将它留作练习（请见 5.1.12）。我们会在表 5.0.3 后面的框注“Alphabet 类的典型用例”来展示一个它的用例。

表 5.0.3 标准字母表

名 称	R()	lgR()	字 符 集
BINARY	2	1	01
DNA	4	2	ACTG
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	ASCII 字符集
EXTENDED_ASCII	256	8	扩展 ASCII 字符集
UNICODE16	65536	16	Unicode 字符集

```
public class Count
{
   public static void main(String[] args)
   {
      Alphabet alpha = new Alphabet(args[0]);
      int R = alpha.R();
      int[] count = new int[R];

      String s = StdIn.readAll();
      int N = s.length();
      for (int i = 0; i < N;  i++)
         if (alpha.contains(s.charAt(i)))
            count[alpha.toIndex(s.charAt(i))]++;

      for (int c = 0; c < R; c++)
         StdOut.println(alpha.toChar(c)
                        + " " + count[c]);

   }
}
```

```
% more abra.txt
ABRACADABRA!

% java Count ABCDR < abra.
txt
A 5
B 2
C 1
D 1
R 2
```

Alphabet 类的典型用例

字符索引数组。我们使用 Alphabet 类的一个最重要的原因是字符索引的数组能够提高算法的效率。在这个数组中，用字符作为索引来获取与之相关联的信息。如果要使用 Java 的 String 类，那就必须使用一个大小为 65 536 的数组；有了 Alphabet 类，则只需要使用一个字母表大小的数组即可。我们将要学习的一些算法能够产生大量的此类数组。在这种情况下，大小为 655 36 的数组是不可接受的。例如前面框注中的 Count 类，它从命令行接受一个字符串并在标准输出上打印输入的每个字符串的出现频率。Count 中用来保存出现频率的 count[] 数组就是一个字符索引数组的示例。你可能会认为数组的计算有些繁琐，但实际上它是 5.1 节介绍的一系列快速排序算法的基础。

698 ≀ 699

数字。你可以从几个标准的 Alphabet 类的示例中看到，我们经常要处理字符串形式的数字。toIndices() 方法能够将任意基于给定的 Alphabet 类的 String 转换为一个 R 进制的数字，用一个元素均在 0 到 R–1 之间的 int[] 数组表示。在某些情况下，一开始就进行这样的转换可以使代码更简洁，因为任意数字都能作为一个字符串索引数组中的索引。例如，如果我们已知输入中仅含有字母表中的字母，那就可以将 Count 中的内循环替换为下面这段更加简洁的代码：

```
int[] a = alpha.toIndices(s);
for (int i = 0; i < N; i++)
   count[a[i]]++;
```

```
% more pi.txt
3141592653
5897932384
6264338327
9502884197
... [π的100 000位]

% java Count 0123456789 < pi.txt
0 9999
1 10137
2 9908
3 10026
4 9971
5 10026
6 10028
7 10025
8 9978
9 9902
```

其中，我们将 R 称为基数，即进制数。我们介绍的几种算法也常常被称为“基数”方法，因为它们一次只处理一位数。

尽管使用 Alphabet 这样的数据类型能够为字符串处理算法带来许多好处（特别是对于较小的字母表），但是本书中并没有实现基于通用字母表 Alphabet 类得到的字符串类型，这是因为：

- 大多数程序使用的都是 String 类型；
- 将字符串转化为索引或是由索引得到字符串常常会落入内循环中，这会大幅降低实现的性能；
- 这会使代码更加复杂，也更加难以理解。

700 ≀ 701

因此我们仍然会使用 String 类，在代码中使用常数 R = 256 并在分析中将 R 作为参数。在适当的时候我们会讨论通用字母表的性能。本书的网站提供了基于 Alphabet 类的各种算法的完整实现。

5.1 字符串排序

对于许多排序应用，决定顺序的键都是字符串。本节中，我们将会考察能够利用字符串的特殊性质将字符串键排序的方法，它们将比第 2 章学过的通用排序方法效率更高。

我们将学习两类完全不同的字符串排序方法。它们都是为程序员服务了几十年的强大方法。

第一类方法会从右到左检查键中的字符。这种方法一般被称为低位优先（LSD）的字符串排序。使用*数字*（digit）代替*字符*（character）的原因要追溯到相同方法在各种数字类型中的应用。如果将一个字符串看作一个 256 进制的数字，那么从右向左检查字符串就等价于先检查数字的最低位。这种方法最适合用于键的长度都相同的字符串排序应用。

第二类方法会从左到右检查键中的字符，首先查看的是最高位的字符。这些方法通常称为高位优先（MSD）的字符串排序——本节将会学习两种此类算法。高位优先的字符串排序的吸引人之处在于，它们不一定需要检查所有的输入就能够完成排序。高位优先的字符串排序和快速排序类似，因为它们都会将需要排序的数组切分为独立的部分并递归地用相同的方法处理子数组来完成排序。它们的区别之处在于高位优先的字符串排序算法在切分时仅使用键的第一个字符，而快速排序的比较则会涉及键的全部。要学习的第一种方法会将相同字符的键划入同一个切分，第二种方法则总会产生三个切分，分别对应被搜索键的第一个字符小于、等于或大于切分键的第一个字符的情况。

在分析字符串排序算法时，字母表的大小是一个重要的因素。尽管我们的重点是基于扩展的 ASCII 字符集的字符串（R=256），但也会分析来自较小字母表的字符串（例如基因序列）和来自较大字母表的字符串（例如含有 65 536 个字符的 Unicode 字母表，它是自然语言编码的国际标准）。 702

5.1.1 键索引计数法

作为热身，我们先学习一种适用于小整数键的简单排序方法。这种叫做*键索引计数*的方法本身就很实用，同时也是本节中将要学习的两三种字符串排序算法的基础。

老师在统计学生的分数时可能会遇到以下数据处理问题。学生被分为若干组，标号为 1、2、3 等。在某些情况下，我们希望将全班同学按组分类。因为组的编号是较小的整数，使用键索引计数法来排序是很合适的，请见图 5.1.1。为了说明这种方法，假设数组 `a[]` 中的每个元素都保存了一个名字和一个组号，其中组号在 0 到 `R-1` 之间，代码 `a[i].key()` 会返回指定学生的组号。这种方法有 4 个步骤，我们会依次讲解。

输入		排序结果（按组别排序）	
姓名	组号		
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

↑
键为较小的整数

图 5.1.1 适于使用键索引计数法的典型情况

5.1.1.1 频率统计

第一步就是使用 `int` 数组 `count[]` 计算每个键出现的频率。对于数组中的每个元素，都使用它的键访问 `count[]` 中的相应元素并将其加 1。如果键为 `r`，则将 `count[r+1]` 加 1。（为什么需要加 1？这么做的原因到下一步你就会明白了。）在图 5.1.2

的例子中，首先将 count[3] 加 1，因为 Anderson 在第二组中，然后会将 count[4] 加 2，因为 Brown 和 Davis 都在第三组中，如此继续。注意，count[0] 的值总是 0，在这个示例中 count[1] 的值也为 0（第零组中没有学生）。

5.1.1.2 将频率转换为索引

接下来，我们会使用 count[] 来计算每个键在排序结果中的起始索引位置。在这个示例中，因为第一组中有 3 个人，第二组中有 5 个人，因此第三组中的同学在排序结果数组中的起始位置为 8。一般来说，任意给定的键的起始索引均为所有较小的键所对应的出现频率之和。对于每个键值 r，小于 r+1 的键的频率之和为小于 r 的键的频率之和加上 count[r]，因此从左向右将 count[] 转化为一张用于排序的索引表是很容易的（请见图 5.1.3）。

```
for (i = 0; i < N; i++)
  count[a[i].key() + 1]++;
```

		count[]					
		0	1	2	3	4	5
		0	0	0	0	0	0
Anderson	2	0	0	0	**1**	0	0
Brown	3	0	0	0	1	**1**	0
Davis	3	0	0	0	1	**2**	0
Garcia	4	0	0	0	1	2	**1**
Harris	1	0	**1**	1	2	1	
Jackson	3	0	1	1	**3**	1	
Johnson	4	0	1	1	3	**2**	
Jones	3	0	1	1	**4**	2	
Martin	1	0	**2**	1	4	2	
Martinez	2	0	2	**2**	4	2	
Miller	2	0	2	**3**	4	2	
Moore	1	0	**3**	3	4	2	
Robinson	2	0	3	**4**	4	2	
Smith	4	0	3	4	4	**3**	
Taylor	3	0	3	4	**5**	3	
Thomas	4	0	3	4	5	**4**	
Thompson	4	0	3	4	5	**5**	
White	2	0	3	**5**	5	5	
Williams	3	0	3	5	**6**	5	
Wilson	4	0	3	5	6	**6**	

总是0

第三组的总人数

图 5.1.2 计算出现频率

```
for (int r = 0; r < R; r++)
  count[r+1] += count[r];
```

	count[]					
r	0	1	2	3	4	5
0	0	0	3	5	6	6
1	0	**0**	3	5	6	6
2	0	0	**3**	5	6	6
3	0	0	3	**8**	6	6
4	0	0	3	8	**14**	6
5	0	0	3	8	14	**20**
	0	0	3	8	14	20

总是0

组号小于3的总人数（第三组在输出中的起始索引）

图 5.1.3 将频率转换为起始索引

5.1.1.3 数据分类

在将 count[] 数组转换为一张索引表之后，将所有元素（学生）移动到一个辅助数组 aux[] 中以进行排序。每个元素在 aux[] 中的位置是由它的键（组别）对应的 count[] 值决定，在移动之后将 count[] 中对应元素的值加 1，以保证 count[r] 总是下一个键为 r 的元素在 aux[] 中的索引位置。这个过程只需遍历一遍数据即可产生排序结果，如图 5.1.4 所示。注意：在我们的一个应用中，这种实现方式的稳定性是很关键的——键相同的元素在排序后会被聚集到一起，但相对顺序没有变化，请见图 5.1.5。

```
for (int i = 0; i < N; i++)
   aux[count[a[i].key()]++] = a[i];
```

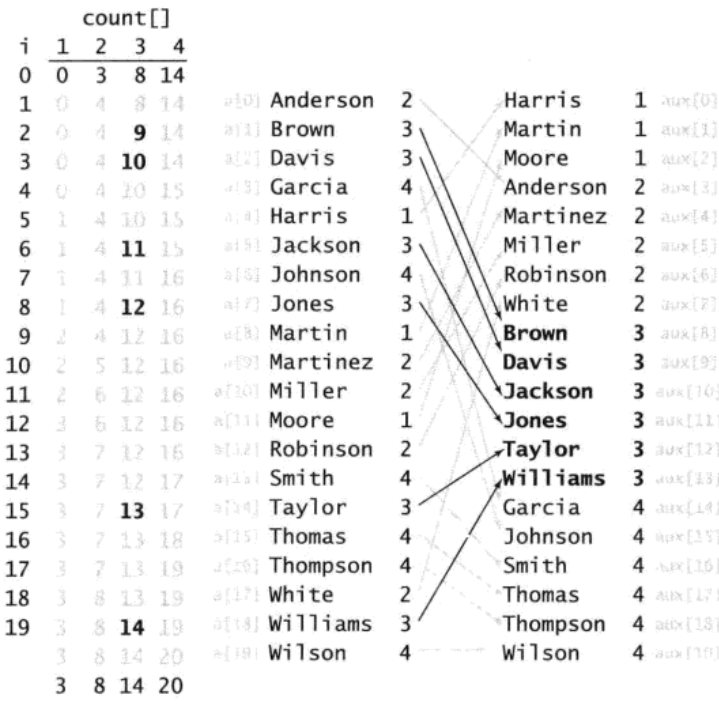


图 5.1.4 将数据分类（键为 3 的条目均突出显示）

703 ~ 704

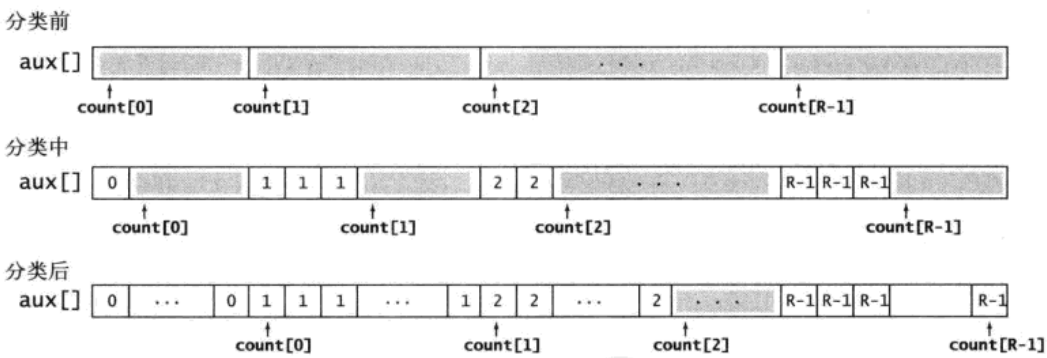


图 5.1.5 键索引计数法（分类阶段）

5.1.1.4 回写

因为我们在将元素移动到辅助数组的过程中完成了排序，所以最后一步就是将排序的结果复制回原数组中。

命题A。键索引计数法排序N个键为0到R–1之间的整数的元素需要访问数组$11N+4R+1$次。

证明。根据代码可得，初始化数组会访问数组$N+R+1$次。在第一次循环中，N个元素均会使计数器的值加1（访问数组$2N$次）；第二次循环会进行R次加法（访问数组$2R$次）；第三次循环会使计数器的值增大N次并移动N次数据（访问数组$3N$次）；第四次循环会移动数据N次（访问数组$2N$次）。所有的移动操作都维护了等键元素的相对顺序。

键索引计数法是一种对于小整数键排序非常有效却常常被忽略的排序方法。理解它的工作原理是理解字符串排序的第一步。命题A意味着键索引计数法突破了$N\log N$的排序算法运行时间下限（之前已经证明过）。它是怎么做到的呢？2.2节中的命题I证明的是所需的比较次数的下限（只能通过compareTo()访问数据）——键索引计数法不需要比较（它只通过key()方法访问数据）。只要当R在N的一个常数因子范围之内，它都是一个线性时间级别的排序方法。

```
int N = a.length;

String[] aux = new String[N];
int[] count = new int[R+1];

// 计算出现频率
for (int i = 0; i < N; i++)
   count[a[i].key() + 1]++;
// 将频率转换为索引
for (int r = 0; r < R; r++)
   count[r+1] += count[r];
// 将元素分类
for (int i = 0; i < N; i++)
   aux[count[a[i].key()]++] = a[i];
// 回写
for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

键索引计数法（a[].key()为[0,R)之间的一个整数）

5.1.2 低位优先的字符串排序

我们学习的第一个字符串排序算法叫做低位优先（LSD）的字符串排序。考虑以下应用：假设有一位工程师架设了一个设备来记录给定时间段内某条忙碌的高速公路上所有车辆的车牌号，他希望知道总共有多少辆不同的车辆经过了这段高速公路。根据2.1节你可以知道，解决这个问题的一种简单方法就是将所有车牌号排序，然后遍历并找出所有不同的车牌号的数量，如Dedup所示（请见3.5.2.1节框注“Dedup过滤器”）。车牌号由数字和字母混合组成，因此一般都将它们表示为字符串。在最简单的情况中（例如图5.1.6所示的加利福尼亚州的车牌号），这些字符串的长度都是相同的。这种情况在排序应用中很常见——比如电话号码、银行账号、IP地址等都是典型的定长字符串。

输入	排序结果
4PGC938	1ICK750
2IYE230	1ICK750
3CIO720	1OHV845
1ICK750	1OHV845
1OHV845	1OHV845
4JZY524	2IYE230
1ICK750	2RLA629
3CIO720	2RLA629
1OHV845	3ATW723
1OHV845	3CIO720
2RLA629	3CIO720
2RLA629	4JZY524
3ATW723	4PGC938

↑ 键的长度均相同

图5.1.6 适于使用低位优先的字符串排序算法的典型情况

将此类字符串排序可以通过键索引计数法来完成，如算法5.1（LSD）和其下方的例子所示。如果字符串的长度均为W，那就从右向左以每个位置的字符作为键，用键索引计数法将字符串排序W遍。乍一看你很难相信这种方法能够产生一个有序的数组——事实上，除非键索引计数法是稳定的，否则这种方法是行不通的。在研究以下证明时请记住这一点并参考后面的示例。

命题 B。低位优先的字符串排序算法能够稳定地将定长字符串排序。

证明。由命题 A 可知，该命题完全依赖于键索引计数法的实现是稳定的。在将它们的最后 i 个字符作为键（用稳定的方式）进行排序之后，可以知道，任意两个键在数组中的顺序都是正确的（只考虑这些字符）。要么因为它们的倒数第 i 个字符不同，所以排序方法已经将它们的顺序摆放正确；要么它们的倒数第 i 个字符相同，所以由于排序的稳定性它们仍然有序（由归纳法可知，对于 i-1 这一点仍然正确）。

705 ~ 706

算法 5.1 低位优先的字符串排序

```
public class LSD
{
   public static void sort(String[] a, int W)
   {  // 通过前W个字符将a[]排序
      int N = a.length;
      int R = 256;
      String[] aux = new String[N];

      for (int d = W-1; d >= 0; d--)
      { // 根据第d个字符用键索引计数法排序

         int[] count = new int[R+1];      // 计算出现频率
         for (int i = 0; i < N; i++)
             count[a[i].charAt(d) + 1]++;

         for (int r = 0; r < R; r++)      // 将频率转换为索引
            count[r+1] += count[r];

         for (int i = 0; i < N; i++)      // 将元素分类
            aux[count[a[i].charAt(d)]++] = a[i];

         for (int i = 0; i < N; i++)      // 回写
            a[i] = aux[i];
       }
    }
}
```

要将每个元素均为含有 W 个字符的字符串数组 a[] 排序，要进行 W 次键索引计数排序：从右向左，以每个位置的字符为键排序一次。

输入(W=7)	d = 6	d = 5	d = 4	d = 3	d = 2	d = 1	d = 0	输出
4PGC938	2IYE230	3CIO720	2IYE230	2RLA629	1ICK750	3ATW723	1ICK750	1ICK750
2IYE230	3CIO720	3CIO720	4JZY524	2RLA629	1ICK750	3CIO720	1ICK750	1ICK750
3CIO720	1ICK750	3ATW723	2RLA629	4PGC938	4PGC938	3CIO720	1OHV845	1OHV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	1OHV845	1ICK750	1OHV845	1OHV845
1OHV845	3CIO720	2RLA629	3CIO720	1ICK750	1OHV845	1ICK750	1OHV845	1OHV845
4JZY524	3ATW723	2RLA629	3CIO720	1ICK750	1OHV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CIO720	3CIO720	4JZY524	2RLA629	2RLA629
3CIO720	1OHV845	4PGC938	1ICK750	3CIO720	3CIO720	1OHV845	2RLA629	2RLA629
1OHV845	1OHV845	1OHV845	1ICK750	1OHV845	2RLA629	1OHV845	3ATW723	3ATW723
1OHV845	1OHV845	1OHV845	1OHV845	1OHV845	3RLA629	1OHV845	3CIO720	3CIO720
2RLA629	4PGC938	1OHV845	1OHV845	1OHV845	3ATW723	4PGC938	3CIO720	3CIO720
2RLA629	2RLA629	1ICK750	1OHV845	3ATW723	4JZY524	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

707

证明该命题的另一种方法是向前看：如果有两个键，它们中还没有被检查过的字符都是完全相同的，那么键的不同之处就仅限于已经被检查过的字符。因为两个键已经被排序过，所以出于稳定性它们将一直保持有序。另外，如果还没被检查过的部分是不同的，那么已经被检查过的字符对于两者的最终顺序没有意义，之后的某轮处理会根据更高位字符的不同修正这对键的顺序。

老式的卡片打孔排序机使用的就是低位优先的基数排序法。这类机器开发于20世纪初期，比用计算机处理商业数据的时代早了数十年。这种机器能够根据卡片上被选定列中孔的模式将一组卡片分别放入10个盒子中。如果多个数字被打在这组卡片的多个列上，操作员将所有卡片排序的方法就是先根据最右边的数字排序，然后将所有卡片按照顺序叠好并再次根据倒数第二个数字排序，如此这般直到排序第一个数字为止。将所有已被排序的卡片按顺序再次叠放就是一个稳定的过程，键索引计数法模仿了这个过程。在整个20世纪70年代，这个版本的低位优先基数排序法不仅在商业领域非常重要，许多严谨的程序员（和学生！）也使用它，因为他们需要将程序保存在打了孔的卡片上（每张卡片上一行）并且会在一组完整表示某个程序的卡片的最后几列打上序号，这样即使卡片散乱之后也能将它们重新按顺序排列。这也是一种将扑克牌排序的简洁方法：将所有牌（按大小）分成13堆，按顺序从13堆排中抽取同种花色的扑克牌，最后将13堆排（按花色）变为4堆。分牌的过程是稳定的，因此花色中的牌也是有序的，所以按照花色将这4堆牌合并即可得到一副已排序的扑克牌，请见图5.1.7。

在许多字符串排序的应用中（甚至对于某些州的车牌号），键的长度可能互不相同。改进后的低位优先的字符串排序是可以适应这些情况的，但我们将这个任务留作练习，因为下面将学习两种专门处理变长键排序的算法。

从理论上说，低位优先的字符串排序的意义重大，因为它是一种适用于一般应用的线性时间排序算法。无论N有多大，它都只遍历W次数据。具体描述如下。

命题B（续）。对于基于R个字符的字母表的N个以长为W的字符串为键的元素，低位优先的字符串排序需要访问~$7WN+3WR$次数组，使用的额外空间与$N+R$成正比。

证明。该方法等价于进行W轮键索引计数法，但是aux[]只会被初始化一次。根据前面的代码和命题A即可得到算法访问数组和使用空间的总数。



图5.1.7 用低位优先的字符串排序算法将一副扑克牌排序

对于典型的应用，R远小于N，因此命题B说明算法的总运行时间与WN成正比。N个长为W的字符串的输入总共含有WN个字符，因此低位优先的字符串排序的运行时间与输入的规模成正比。

5.1.3 高位优先的字符串排序

要实现一个通用的字符串排序算法（字符串的长度不一定相同），我们应该考虑从左向右遍历所有字符。我们知道，以 a 开头的字符串应该排在以 b 开头的字符串前面，等等。实现这种思想的一个很自然方法就是一种递归算法，被称为高位优先（MSD）的字符串排序，请见图 5.1.8。首先用键索引计数法将所有字符串按照首字母排序，然后（递归地）再将每个首字母所对应的子数组排序（忽略首字母，因为每一类中的所有字符串的首字母都是相同的）。和快速排序一样，高位优先的字符串排序会将数组切分为能够独立排序的子数组来完成排序任务，但它的切分会为每个首字母得到一个子数组，而不是像快速排序中那样产生固定的两个或三个切分，请见图 5.1.9。

♣J ♥6 ♦A ♥A ♠K ♥J ♦Q ♣6 ♠J ♣A ♦9 ♥9 ♦8 ♠9 ♣K ♦4 ♠5 ♣Q ♥3 ♠2 ♣10 ♣9 ♥7 ♣4 ♥4 ♦10 ♠A ♦5 ♠3 ♥8 ♣2 ♦K ♠4 ♣7 ♥Q ♦J ♠6 ♣3 ♠7 ♠8 ♠10 ♦3 ♥10 ♦7 ♠Q ♥2 ♦2 ♣5 ♥K ♥5 ♦6 ♣8

♠K ♠J ♠9 ♠5 ♣2 ♠A ♠3 ♠4 ♠6 ♠7 ♠8 ♣10 ♠Q ♥6 ♥A ♥J ♥9 ♥3 ♥7 ♥4 ♥8 ♥Q ♥10 ♥2 ♥K ♦5 ♦A ♦Q ♦9 ♦8 ♦4 ♦10 ♦5 ♦K ♦J ♦3 ♦7 ♦2 ♦6 ♣J ♣6 ♣A ♣K ♣Q ♣10 ♣9 ♣4 ♣2 ♣7 ♣3 ♣5 ♣8

♠A ♠2 ♠3 ♠4 ♠5 ♠6 ♠7 ♠8 ♠9 ♠10 ♠J ♠Q ♠K ♥A ♥2 ♥3 ♥4 ♥5 ♥6 ♥7 ♥8 ♥9 ♥10 ♥J ♥Q ♥K ♦A ♦2 ♦3 ♦4 ♦5 ♦6 ♦7 ♦8 ♦9 ♦10 ♦J ♦Q ♦K ♣A ♣2 ♣3 ♣4 ♣5 ♣6 ♣7 ♣8 ♣9 ♣10 ♣J ♣Q ♣K

图 5.1.8 用高位优先的字符串排序算法将一副扑克牌排序

5.1.3.1 对字符串末尾的约定

在高位优先的字符串排序算法中，要特别注意到达字符串末尾的情况。在排序中，合理的做法是将所有字符都已被检查过的字符串所在的子数组排在所有子数组的前面，这样就不需要递归地将该子数组排序，请见图 5.1.10。为了简化这两步计算，我们使用了一个接受两个参数的私有方法 `toChar()` 来将字符串中字符索引转化为数组索引，当指定的位置超过了字符串的末尾时该方法返回 –1。然后将所有返回值加 1，得到一个非负的 `int` 值并用它作为 `count[]` 的索引。这种转换意味着字符串中的每个字符都可能产生 R+1 中不同的值：0 表示字符串的结尾，1 表示字母表的第一个字符，2 表示字母表的第二个字符，等等。因为键索引计数法本来就需要一个额外的位置，所以使用代码 `int count[] = new int[R+2];` 创建记录统计频率的数组（将所有值设为 0）。注意：某些编程语言，特别是 C 和 C++，已经约定了字符串结束的表示方法，因此对于这类语言本节的代码需要进行相应的调整。

708 ≀ 710

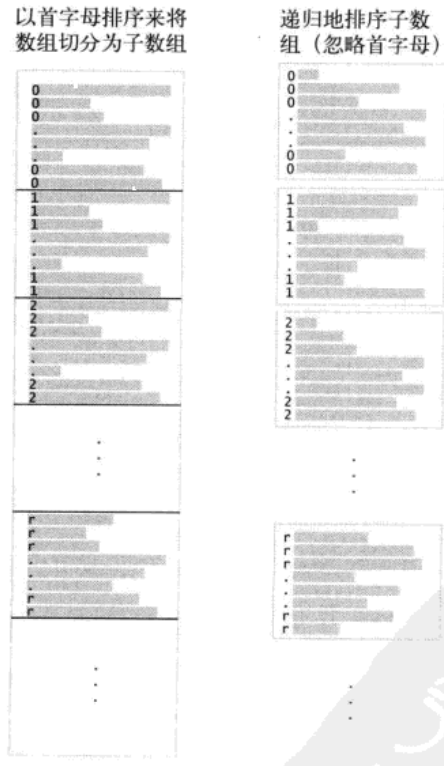


图 5.1.9 高位优先的字符串排序的示意图

有了这些预备知识，就会知道算法 5.2 实现高位优先的字符串排序算法所需的新代码其实并不多。增加了一个条件语句以在子数组较小时切换插入排序，（这里使用的是一个特殊版本的插

入排序，我们会在稍后考察。）还添加了一个键索引计数法的循环来完成递归调用。从表 5.1.1 可知，count[] 数组中的值（在统计频率、转换为索引并将数据分类之后）正是将每个字符所对应的子数组（递归地）排序时所需要的值。

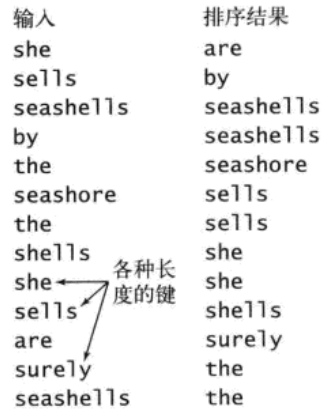


图 5.1.10 适于使用高位优先的字符串排序的典型情况

5.1.3.2 指定的字母表

高位优先的字符串排序的成本与字母表中的字符数量有很大关系。我们可以很容易地令排序算法修接受一个 Alphabet 对象作为参数，以改进基于较小的字母表的字符串排序程序的性能。完成这一点需要进行如下改动：

- 在构造函数中用一个 alpha 对象保存字母表；
- 在构造函数中将 R 设为 alpha.R()；
- 在 charAt() 方法中将 s.charAt(d) 替换为 alpha.toIndex(s.charAt(d))。

711

表 5.1.1 高位优先的字符串排序中 count[] 数组的意义

第 d 个字符排序的完成阶段	ccunt[r] 的值				
	r=0	r=1	r 在 2 与 R-1 之间	r=R	r=R+1
频率统计	0（未使用）	长度为 d 的字符串数量	第 d 个字符的索引值是 r-2 的字符串的数量		
将频率转化为索引	长度为 d 的字符串的子数组的起始索引	第 d 个字符的索引值是 r-1 的字符串的子数组的起始索引			未使用
数据分类	第 d 个字符的索引值为 r 的字符串的子数组的起始索引			未使用	
	1+ 长度为 d 的字符串的子数组的结束索引	1+ 第 d 个字符串的索引值是 r-1 的字符串的子数组的结束索引			未使用

在本节的示例中，字符串都是由小写字母组成的。扩展低位优先的字符串排序算法以支持这种特性也很简单，但带来的性能提升一般比高位优先的字符串排序小得多。

算法 5.2 高位优先的字符串排序

```
public class MSD
{
   private static int R = 256;         // 基数
   private static final int M = 15;    // 小数组的切换阈值
   private static String[] aux;        // 数据分类的辅助数组
   private static int charAt(String s, int d)

   {  if (d < s.length()) return s.charAt(d); else return -1;  }

   public static void sort(String[] a)
   {
      int N = a.length;
      aux = new String[N];
      sort(a, 0, N-1, 0);
   }

   private static void sort(String[] a, int lo, int hi, int d)
   {  // 以第d个字符为键将a[lo]至a[hi]排序
      if (hi <= lo + M)
      {  Insertion.sort(a, lo, hi, d); return;  }
```

```
        int[] count = new int[R+2];          // 计算频率
        for (int i = lo; i <= hi; i++)
           count[charAt(a[i], d) + 2]++;

        for (int r = 0; r < R+1; r++)        // 将频率转换为索引
           count[r+1] += count[r];

        for (int i = lo; i <= hi; i++)       // 数据分类
           aux[count[charAt(a[i], d) + 1]++] = a[i];

        for (int i = lo; i <= hi; i++)       // 回写
           a[i] = aux[i - lo];

        // 递归的以每个字符为键进行排序
        for (int r = 0; r < R; r++)
           sort(a, lo + count[r], lo + count[r+1] - 1, d+1);

     }

}
```

在将一个字符串数组 a[] 排序时，首先根据它们的首字母用键索引计数法进行排序，然后（递归地）根据子数组中的字符串的首字母将子数组排序。 712

算法 5.2 中的代码的简洁令人刮目相看，它隐藏了一些非常复杂的计算。花些时间深入研究图 5.1.11 所示的算法顶层调用轨迹和图 5.1.12 中递归调用的轨迹以确保你理解了这个算法的精妙之处，这些时间不会白花。在这段轨迹中，小数组的插入排序切换阈值（M）为 0，因此你可以看到完整的排序过程。在这个例子中，字符串来自于 Alphabet.LOWERCASE，其中 R=26。一般的应用使用的大都是 R=256 的 Alphabet.EXTENDED_ASCII，或是 R=65 536 的 Alphabet.UNICODE。对于较大的字母表，高位优先的排序算法虽然简单但可能会很危险——如果使用不当，它可能会消耗令人无法承受的时间和空间。在仔细研究它的性能特点之前，我们要先讨论三个在任何应用中都必须解决的重要的问题（这些问题曾在第 2 章中讨论过）。

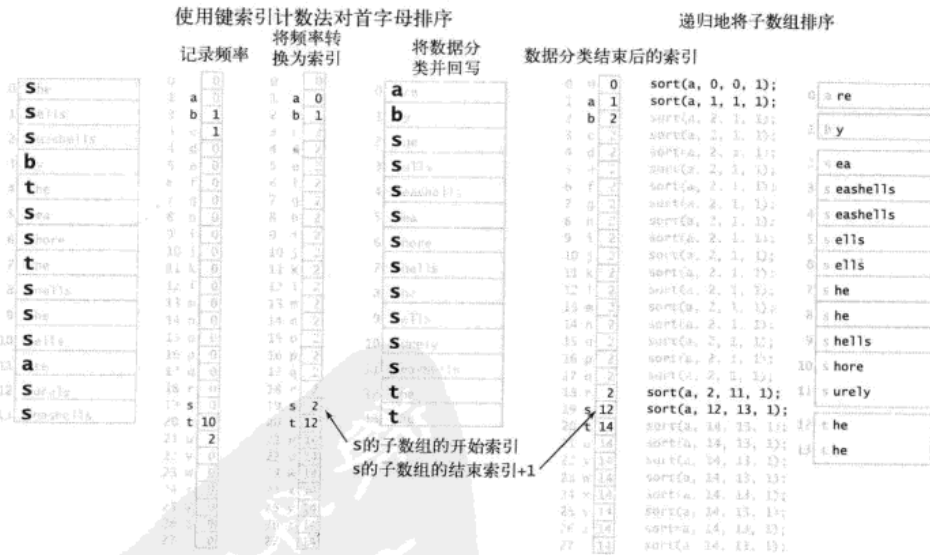


图 5.1.11 高位优先的字符串排序：sort(a, 0, 14, 0) 的顶层轨迹

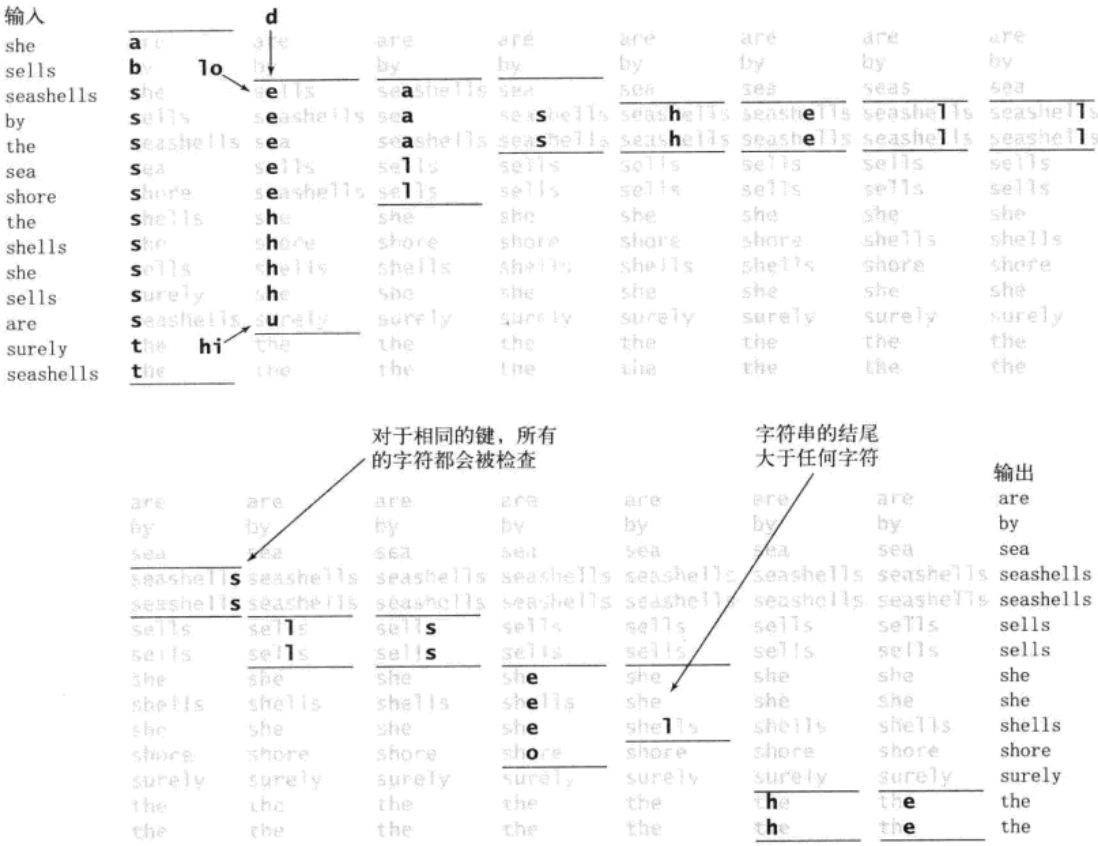


图 5.1.12 高位优先的字符串排序的递归调用轨迹（小数组不会切换到插入排序，大小为 0 和 1 的子数组已被省略）

5.1.3.3 小型子数组

高位优先的字符串排序的基本思想是很有效的：在一般的应用中，只需检查若干个字符就能完成所有字符串的排序。换句话说，这种方法能够快速地将需要排序的数组切分为较小的数组。但这种切分也是一把双刃剑：我们肯定会需要处理大量微型数组，因此必须快速处理它们。*小型子数组对于高位优先的字符串排序的性能至关重要。*我们在其他递归排序算法中也遇到过这种情况（快速排序和归并排序），但小数组对于高位优先的字符串排序的影响尤其强烈。例如，假设你需要将数百万个不同的 ASCII 字符串（R=256）排序且不会对小数组进行特殊处理。每个字符串最终都会产生一个只含有它自己的子数组，因此你需要将数百万个大小为 1 的子数组排序。但每次排序都需要将 `count[]` 的 258 个元素初始化为 0 并将它们都转化为索引。这种代价比排序的其他部分要高很多。在使用 Unicode 时（R=655 36），排序过程可能会减慢上千倍。事实上，正因为如此，许多使用排序但考虑不周的程序在从 ASCII 切换到 Unicode 后运行时间从几分钟暴涨到几个小时。然而，将小数组切换到插入排序对于高位优先的字符串排序算法是必须的。为了避免重复检查已知相同的字符所带来的成本，我们使用了后面框注“对前 d 个字符均相同的字符串执行插入排序”中给出的一个版本的插入排序。它接受一个额外的参数 d 并假设所有需要排序的字符串的前 d 个字符都是相同的。这段代码的效率取决于 `substring()` 方法所需的时间是否为常数。和快速排序以及归并排序一样，

713
~
714

一个较小的转换阈值就能将性能提高很多，但对于高位优先的字符串排序算法它节约的时间是非常可观的。图 5.1.13 显示了一个典型应用中的实验结果。在长度小于等于 10 时将子数组切换到插入排序能够将运行时间降低为原来的十分之一。

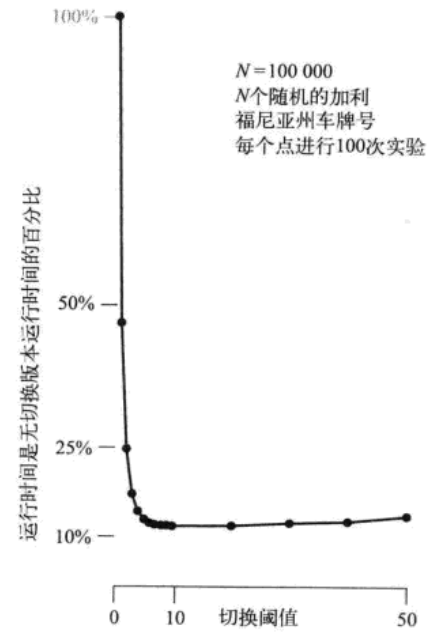


图 5.1.13 高位优先的字符串排序算法 中切换小型子数组的排序方法的实际效果

5.1.3.4 等值键

高位优先的字符串排序中的第二个陷阱是，对于含有大量等值键的子数组的排序会较慢。如果相同的子字符串出现得过多，切换排序方法条件将不会出现，那么递归方法就会检查所有相同键中的每一个字符。另外，键索引计数法无法有效判断字符串中的字符是否全部相同：它不仅需要检查每个字符和移动每个字符串，还需要初始化所有的频率统计并将它们转换为索引等。因此，高位优先的字符串排序的最坏情况就是所有的键均相同。大量含有相同前缀的键也会产生同样的问题，这在一般的应用场景中是很常见的。

5.1.3.5 额外空间

为了进行切分，高位优先的算法使用了两个辅助数组：一个用来将数据分类的临时数组（aux[]）和一个用来保存将会被转化为切分索引的统计频率的数组（count[]）。aux[] 的大小为 N 且可以在递归方法 sort() 外创建。如果牺牲稳定性，则可以去掉 aux[] 数组（请见练习 5.1.17），但它并不是高位优先的字符串排序算法在实际应用中所关注的内容。相反，count[] 所需的空间才是主要问题（因为它不能在递归方法 sort() 之外创建），如下文的命题 D 所述。

```
public static void sort(String[] a, int lo, int hi, int d)
{  // 对前d个字符排序，从a[lo]到a[hi]

   for (int i = lo; i <= hi; i++)
      for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
         exch(a, j, j-1);
}

private static boolean less(String v, String w, int d)
{  return v.substring(d).compareTo(w.substring(d)) < 0;  }
```

对前 d 个字符均相同的字符串执行插入排序

5.1.3.6 随机字符串模型

为了研究高位优先的字符串排序算法的性能，我们使用了一个随机字符串模型，其中每个字符串都（独立的）由随机字符组成，长度没有限制。这实际上排除了出现较长的等值键的情况，因为它们出现的几率非常小。高位优先的字符串排序算法在这个模型中的表现和随机定长键模型中的表现类似，也和它在一般的真实数据中的性能类似。我们将会看到，在这三种情况中，高位优先的字符串排序算法通常都只需要检查每个键开头的若干个字符即可。

5.1.3.7 性能

高位优先的字符串排序算法的性能取决于数据。对于基于比较的方法，我们主要关注的是键的顺序；对于高位优先的字符串排序算法，键的顺序并不重要，我们关注的是键所对应的值，请见图 5.1.14。

- 对于随机输入，高位优先的字符串排序算法只会检查足以区别字符串所需的字符。相对于输入数据中的字符总数，算法的运行时间是亚线性的（它只会检查输入字符中的一小部分）。
- 对于非随机的输入，高位优先的字符串排序算法可能仍然是亚线性的，但需要检查的字符可能比随机情况下更多。特别是对于相等的键，它需要检查它们的所有字符，所以当存在大量等值键时它所需的运行时间是接近线性的。
- 在最坏情况下，高位优先的字符串排序算法会检查所有键中的所有字符，所以相对于数据中的所有字符它所需的运行时间是线性的（和低位优先的字符串排序算法相同）。最坏情况下的输入中所有的字符串均相同。

随机字符串（亚线性时间）	非随机字符串且有重复（接近线性时间）	最坏情况（线性时间）
1E**IO402**	a**re**	1DNB377
1H**YL490**	b**y**	1DNB377
1R**OZ572**	sea	1DNB377
2H**XE734**	seashells	1DNB377
2I**YE230**	seashells	1DNB377
2X**OR846**	sells	1DNB377
3CD**B573**	sells	1DNB377
3CV**P720**	she	1DNB377
3I**GJ319**	she	1DNB377
3K**NA382**	shel**ls**	1DNB377
3T**AV879**	sho**re**	1DNB377
4C**QP781**	s**urely**	1DNB377
4Q**GI284**	the	1DNB377
4Y**HV229**	the	1DNB377

图 5.1.14 高位优先的字符串排序算法的字符检查情况

716

某些应用程序所处理的键和随机字符串模型能很好匹配，而有些则含有很多重复的键或是较长的公共前缀，这种情况下排序所需的时间和最坏情况接近。比如，在我们的车牌号处理应用程序中这两种极端情况都可能出现：如果工程师选取一条繁忙的州际公路一小时的数据，那么数据中的重复项会很少，符合随机模型；如果取的是一条乡间小道一个星期的数据，那么数据中肯定会有大量的重复项，算法的性能将会和最坏情况类似。

作为提示以及对为何该证明已经超出了本书的范围的说明，我在这里提醒大家注意，命题的结论和键的长度是无关的。事实上，随机字符串模型所允许的键长接近无限。两个键之间有任意多的字符相吻合，这个可能性不是零，但这个可能性非常小，在估计性能时可以将其忽略。

由以上讨论可以知道，检查的字符数量并不是高位优先的字符串排序算法性能的全部。我们还需要考虑统计字符的出现频率以及将频率转化为索引所需要的时间和空间。

命题 C。要将基于大小为 R 的字母表的 N 个字符串排序，高位优先的字符串排序算法平均需要检查 $N\log_R N$ 个字符。

简略证明。我们希望子数组的大小几乎都是相同的，因此递推关系 $C_N=RC_{N/R}+N$ 可以近似地描述算法的性能并得到命题所述的结果。它也是第 2 章中快速排序性能证明的一般化证明。另一方面，这种描述并不完全准确，因为 N/R 并不一定能够得到整数，子数组的大小相同也仅是平均而言（而且在现实中键的长度是有限的）。这些因素对高位优先的字符串排序算法的影响比对标准快速排序算法的影响小，因此算法运行时间中的最大项就是这个递推关系的答案。这个问题的详细证明是算法分析中的经典例子，最早由 Knuth 完成于 20 世纪 70 年代早期。

命题 D。要将基于大小为 R 的字母表的 N 个字符串排序，高位优先的字符串排序算法访问数组的次数在 $8N+3R$ 到 $7wN+3wR$ 之间，其中 w 是字符串的平均长度。

证明。由代码、命题 A 和命题 B 可得，在最好情况下高位优先的排序算法只需遍历数据一轮；而在最坏情况下，它和低位优先的字符串排序算法的性能类似。 717

当 N 较小时，R 是主要因子。尽管对总成本的精确分析是困难而复杂的，但你只需考虑无重复键的情况下所有较小的子数组就可以估计出该成本的实际效果。在不为较小的子数组切换排序方法的情况下，每个键都会产生一个单独的子数组，因此仅为处理这些子数组就需要访问 NR 次数组。如果为小于 M 的数组切换排序方法，将会有 N/M 个大小为 M 的子数组，因此等于是在用 $NM/4$ 次比较换取 NR/M 次数组访问，这说明应该选择与 R 的平方根成正比的 M。

命题 D（续）。要将基于大小为 R 的字母表的 N 个字符串排序，最坏情况下高位优先的字符串排序算法所需的空间与 R 乘以最长的字符串的长度之积成正比（再加上 N）。

证明。`count[]` 数组必须在 `sort()` 中创建，因此空间需求的总量与 R 和递归的深度之积成正比（再加上辅助数组的大小 N）。准确地说，递归的深度即最长字符串的长度，也就是两个或多个被排序的字符串的公共前缀的长度。

正如刚才所讨论的，相等的键使得递归的深度和键的长度成正比。由命题 D 马上可以推论出，在用高位优先的字符串排序算法将基于大型字母表的长字符串排序时，它很有可能消耗过多的时间或者空间，特别是在已知可能出现较长的等值键的情况下。例如，如果使用的是 `Alphabet.UNICODE` 且某些字符串中公共前缀的长度超过 1000 个字符，那么 `MSD.sort()` 将需要为超过 6500 万个计数器元素分配空间！

在将长字符串排序时，令高位优先的字符串排序算法发挥出最大效率的主要挑战在于处理数据中的非随机因素。一般来说，一些键可能存在较长的公共部分，或者部分键的取值范围有限。比如，在处理学生信息的应用程序中，数据的键可能是毕业年份（4 个字节，但只有 4 种可能的值），州名（可能需要 10 个字节，但只有 50 种可能的值），性别（1 个字节，2 种值）以及学生的姓名（和随机字符串最接近，但有可能很长，字母出现频率的分布并不均匀且当该栏长度固定时字符串的末尾会被添加许多空格）。这些限制使得高位优先的字符串排序算法会产生许多空子数组。下面我们将学习一种能够漂亮地解决这个问题的算法。 718

5.1.4 三向字符串快速排序

我们也可以根据高位优先的字符串排序算法改进快速排序，根据键的首字母进行三向切分，仅在中间子数组中的下一个字符（因为键的首字母都与切分字符相等）继续递归排序。这个算法的实现并不困难，请见算法 5.3：我们只是为算法 2.5 中的递归方法添加了一个参数来保存当前的切分字母并令三向切分的代码使用该字符，然后递归适当修正方法，请见图 5.1.15。

尽管排序的方式有所不同，但三向字符串快速排序根据的仍然是键的首字母并使用递归方法将其余部分的键排序。对于字符串的排序，这个方法比普通的快速排序和高位优先的字符串排序更友好。实际上，它就是这两种算法的结合。

三向字符串快速排序只将数组切分为三部分，因此当相应的高位优先的字符串排序产生的非空切分较多时，它需要移动的数据量就会变大，因为它需要进行一系列的三向切分才能取得多向切分的效果。但是，高位优先的字符串排序可能会创建大量（空）子数组，而三向字符串快速排序的切分总是只有三个。因此三向字符串快速排序能够很好处理等值键、有较长公共前缀的键、取值范围较小的键和小数组——所有高位优先的字符串排序算法不善长的各种情况，请见图 5.1.16。特别重要的一点是，这种切分方法能够适应键的不同部分的不同结构。和快速排序一样，三向字符串快速排序也不需要额外的空间(递归所需的隐式栈除外)，这是它相比高位优先的字符串排序的一大优点，后者在统计频率和使用辅助数组时都需要空间。

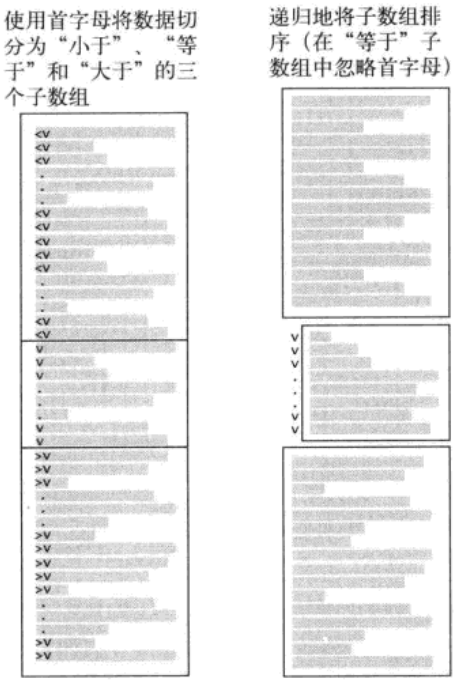


图 5.1.15 三向字符串快速排序的示意图



图 5.1.16 适于使用三向字符串快速排序的典型情况

图 5.1.17 显示了 Quick3string 在处理样例数据时产生的所有递归调用。每个子数组都正好只用了三个递归调用就完成了排序，只是省略了中间子数组中到达（相等的）字符串的结尾时的递归调用。

和以前一样，在实际应用中下列对算法 5.3 的标准改进都是很值得考虑的。

5.1.4.1 小型子数组

在所有的递归算法中，我们都可以通过对小型子数组进行特殊处理来提高效率。这里使用的是 5.1.2.3 框注中的“对前 d 个字符均相同的字符串执行插入排序”中的插入排序，它能够跳过已知相等的字符。这项修改带来的改进会很明显，尽管它在三向字符串排序的重要性远不如它在高位优先的字符串排序的重要性高。

5.1.4.2 有限的字母表

为了处理特殊的字母表，可以为所有方法添加一个 Alphabet 类型的参数 alpha 并在 charAt() 方法中将 s.charAt(d) 替换为 alpha.toIndex(s.charAt(d))。在这里，这么做并不能得到什么收

益，相反添加这段代码可能会大幅降低算法的运行速度，因为它在内循环之中。

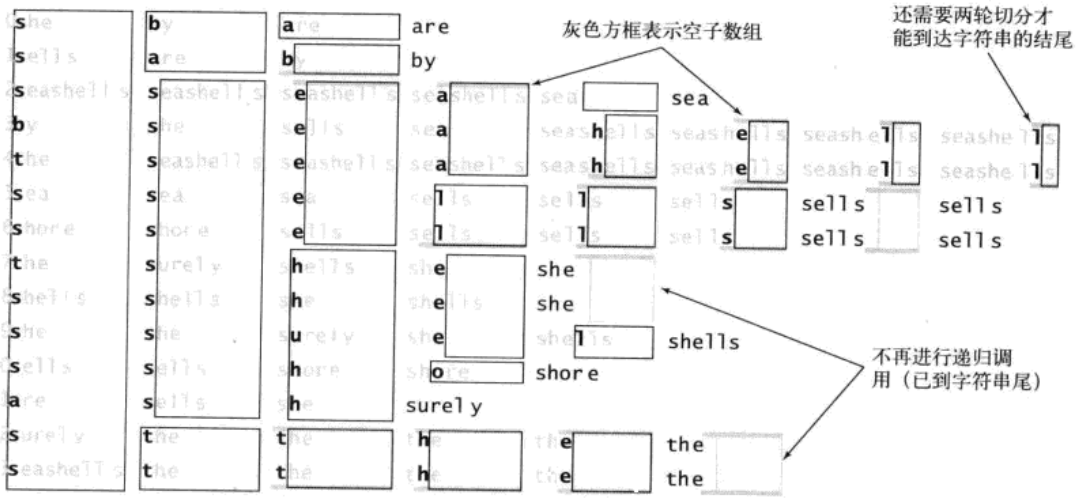


图 5.1.7 三向字符串快速排序的递归调用轨迹（不在子数组较小时切换排序方法）

算法 5.3 三向字符串快速排序

```
public class Quick3string
{
   private static int charAt(String s, int d)
   {  if (d < s.length()) return s.charAt(d); else return -1;  }
   public static void sort(String[] a)
   {  sort(a, 0, a.length - 1, 0);  }
   private static void sort(String[] a, int lo, int hi, int d)
   {
      if (hi <= lo) return;
      int lt = lo, gt = hi;
      int v = charAt(a[lo], d);
      int i = lo + 1;
      while (i <= gt)
      {
         int t = charAt(a[i], d);
         if      (t < v) exch(a, lt++, i++);
         else if (t > v) exch(a, i, gt--);
         else            i++;

      }
      // a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi]
      sort(a, lo, lt-1, d);
      if (v >= 0) sort(a, lt, gt, d+1);
      sort(a, gt+1, hi, d);
   }
}
```

在将字符串数组 a[] 排序时，根据它们的首字母进行三向切分，然后（递归地）将得到的三个子数组排序：一个含有所有首字母小于切分字符的字符串子数组，一个含有所有首字母等于切分字符的字符串的子数组（排序时忽略它们的首字母），一个含有所有首字母大于切分字符的字符串的子数组。

719 ≀ 721

5.1.4.3 随机化

和快速排序一样，最好在排序之前将数组打乱或是将第一个元素和一个随机位置的元素交换以得到一个随机的切分元素。这么做主要是为了预防数组已经有序或是接近有序的最坏情况。

对于字符串类型的键，标准的快速排序以及第 2 章中的其他排序方法实际上都是高位优先类的字符串排序算法，这是因为 `String` 类的 `compareTo()` 方法是从左到右访问字符串中的所有字符的。也就是说，`compareTo()` 在首字母不同时只会访问首字母，在首字母相同且第二个字母不同时只会访问它们的前两个字母，等等。例如，如果所有字符串的首字母均不相同，标准的排序算法只会检查这些首字母，这就自动实现了一些我们希望对高位优先的字符串排序算法的改进。三向字符串排序背后的核心思想是对首字母相同的键采取特殊的策略。实际上你可以把算法 5.3 看作对标准快速排序的改进，使之能够记录已知相同的多个开头字母。在较小的子数组中，排序所需的大多数比较都已经完成，其中的字符串很可能含有多个相同的开头字母。标准的方法在每次比较时仍然需要扫描整个字符串，但三向字符串快速排序则可以避免这一点。

5.1.4.4 性能

考虑字符串键都很长的情况（简单起见，长度均相同）且键前面的大半部分首字母均相同。在这种情况下，标准快速排序的性能与字符串的长度乘以 $2N\ln N$ 成正比，而三向字符串排序的运行时间则与 N 乘以字符串的长度（需要发现所有的相同开头字母）再加上 $2N\ln N$ 次比较（对剩下的较短部分进行排序）的和成正比。也就是说，三向字符串快速排序所需比较的字符最多比普通的快速排
722 序少 $2\ln N$ 个。实际排序应用中处理的键和这个例子类似的情况也并不少见。

命题 E。要将含有 N 个随机字符串的数组排序，三向字符串快速排序平均需要比较字符 ~ $2N\ln N$ 次。

证明。我们可以用两种方式来理解这个结论。首先，将这个方法看作在快速排序中用首字母切分并（递归地）调用相同的方法将子数组排序，那么它所需的操作数量和普通的快速排序相同就一点也不奇怪了——但这只是比较单个字符所需的操作，而非比较整个键所需的次数。其次，可以将这个方法看作用快速排序代替了键索引计数法，根据命题 D，我们预计的运行时间为 $N\log_R N$ 与 $2\ln N$ 的积，这是因为快速排序需要 $2R\ln R$ 步来将 R 个字符排序，而对于相同的字符串，高位优先的字符串排序算法只需要 R 步。这里就不给出完整的证明了。

我们曾在 5.1.3.7 节强调过，随机字符串模型是很有用的，但要预测实际情况下算法的性能还需要更仔细的分析。研究者已经对这个算法进行了深入的研究并已经证明在非常一般的假设下，其他算法最多比三向字符串快速排序快常数级别（以比较的字符数量衡量）。它的应用非常广泛，因为三向字符串快速排序的性能并不直接取决于字母表的大小。

5.1.4.5 举例：网站日志

作为三向字符串快速排序鹤立鸡群的一个示例，我们来考察一个现代系统中的典型数据处理任务。假设你架设了一个网站并希望分析它产生的流量。你可以从系统管理员那里得到网站的所有活动，每项活动的信息中都含有发起者的域名。例如，本书网站上的 web.log.txt 文件中包含的就是该网站一个星期中的所有活动。为什么三向字符串快速排序能够有效处理这种文件呢？因为排序结果中许多字符串都有
713 很长的公共前缀，而这种算法不会重复检查它们。

5.1.5 字符串排序算法的选择

我们很自然会对这里的字符串排序算法和第 2 章中的通用排序算法的对比感兴趣。表 5.1.2 总

结了本节所讨论过的字符串排序算法的重要特征（快速排序、归并排序和三向快速排序的数据来自第 2 章，以供比较）。

表 5.1.2 各种字符串排序算法的性能特点

算 法	是否稳定	原地排序	在将基于大小为 R 的字母表的 N 个字符串排序的过程中调用 charAt() 方法次数的增长数量级（平均长度为 w，最大长度为 W）		优势领域
			运行时间	额外空间	
字符串的插入排序	是	是	N 到 N^2 之间	1	小数组或是已经有序的数组
快速排序	否	是	$N\log^2 N$	$\log N$	通用排序算法，特别适合用于空间不足的情况
归并排序	是	否	$N\log^2 N$	N	稳定的通用排序算法
三向快速排序	否	是	N 到 $N\log N$ 之间	$\log N$	大量重复键
低位优先的字符串排序	是	否	NW	N	较短的定长字符串
高位优先的字符串排序	是	否	N 到 Nw 之间	$N+WR$	随机字符串
三向字符串快速排序	否	是	N 到 Nw 之间	$W+\log N$	通用排序算法，特别适合用于含有较长公共前缀的字符串

和第 2 章一样，根据具体的算法和数据将这些增长数量级乘以适当的常数就可以估计出程序所需的运行时间。

我们已经看到过许多示例和练习中的许多示例，不同的情况需要用不同的算法和参数来处理。在专家的指导下（现在也许就是你），在特定的场景下算法的性能也许能够得到大幅度提高。 724

答疑

问 Java 系统的排序使用了这些方法来处理 String 对象吗？

答 没有，但 Java 的标准实现中的字符串比较非常快，它使得标准排序的性能与本节中讨论的这些算法不相上下。

问 那么，我只需要使用系统排序来处理 String 类型的键就可以了吗？

答 在 Java 中可能是这样的。当然如果你要处理的字符串非常多或者需要一个极快的算法，就可能需要用 char 数组代替 String 对象并使用基数排序算法。

问 表 5.1.2 中的 $\log^2 N$ 是怎么回事？

答 说明这些算法中的大多数比较都是在含有长度约为 $\log N$ 的公共前缀的字符串之间进行的。最近的一些研究通过详细的数学分析也证明了随机字符串也满足这一性质（参见本书网站）。 725

练习

5.1.1 实现一种排序算法，首先统计不同键的数量，然后使用一个符号表来实现键索引计数法并将数组排序。（这种方法不适用于不同键的数量很大的情况）。

5.1.2 给出使用低位优先的字符串排序算法处理下面这些键的轨迹：no is th ti fo al go pe to co to th ai of th pa。

5.1.3 给出使用高位优先的字符串排序算法处理下面这些键的轨迹：no is th ti fo al go pe to co to th ai of th pa。

5.1.4 给出使用三向字符串快速排序算法处理下面这些键的轨迹：no is th ti fo al go pe to co to th ai of th pa。

5.1.5 给出使用高位优先的字符串排序算法处理下面这些键的轨迹：now is the time for all good people to come to the aid of。

5.1.6 给出使用三向字符串快速排序算法处理下面这些键的轨迹：now is the time for all good people to come to the aid of。

5.1.7 用一个 Queue 对象的数组实现键索引计数法。

5.1.8 对于一个含有 N 个键 a, aa, aaa, aaaa, ... 的文件，给出高位优先的字符串排序和三向字符串快速排序所检查的字符数量。

5.1.9 实现能够处理变长字符串的低位优先的字符串排序算法。

5.1.10 要将 N 个定长字符串排序（长度均为 W），在最坏情况下三向字符串快速排序总共需要检查多少个字符？

726

提高题

5.1.11 *队列排序*。按照以下方法使用队列实现高位优先的字符串排序：为每个盒子[①]设置一个队列。在第一次遍历所有元素时，将每个元素根据首字母插入到适当的队列中。然后，将每个子列表排序并合并所有队列得到一个完整的排序结果。注意，在这种方法中 count[] 数组不需要在递归方法内创建。

5.1.12 *字母表*。实现 5.0.2 节给出的 Alphabet 类的 API 并用它实现能够处理任意字母表的低位优先的和高位优先的字符串排序算法。

5.1.13 *混合排序*。利用标准的高位优先的字符串排序的多向切分优势处理大型数组，利用三向字符串快速排序能够避免产生大量空子数组的特点处理小型数组。研究这种想法的可行性。

5.1.14 *数组排序*。编写一个方法，使用三向字符串快速排序处理以整型数组作为键的情况。

5.1.15 *亚线性排序*。编写一个处理 int 值的排序算法，遍历数组两遍，第一遍根据所有键的高 16 位进行低位优先的排序，第二遍进行插入排序。

5.1.16 *链表排序*。编写一个排序算法，接受一条以 String 为键值参数的结点链表并重新按顺序排列所有结点（返回一个指向键值最小的结点的指针）。使用三向字符串快速排序。

5.1.17 *原地键索引计数法*。实现一个仅使用常数级别的额外空间的键索引计数法。证明你的实现是稳定的或者提供一个反例。

727

实验题

5.1.18 *随机小数键*。编写一个静态方法 randomDecimalKeys，接受整型参数 N 和 W 并返回一个含有 N 个字符串的数组，每个字符串都是一个含有 W 位数的小数。

① 参见老式卡片打孔排序机。——译者注

5.1.19 *随机的加利福尼亚州车牌号*。编写一个静态方法 `randomPlatesCA`，接受一个整型参数 `N` 并返回一个含有 `N` 个字符串的数组，每个字符串都是与本节的示例类似的加利福尼亚州的车牌号。

5.1.20 *随机定长单词*。编写一个静态方法 `randomFixedLengthWords`，接受整型参数 `N` 和 `W` 并返回一个含有 `N` 个字符串的数组，每个字符串都基于英文字母表且长度为 `W`。

5.1.21 *随机元素*。写一个静态方法 `randomItems`，接受整型参数 `N` 并返回一个含有 `N` 个字符串的数组，每个字符串的长度均在 15 到 30 之间且由三个部分组成：第一个部分含有 4 个字符，来自于 10 个固定的字符串；第二个部分含有 10 个字符，来自于 50 个固定的字符串；第三个部分含有 1 个字符，来自于 2 个固定的字符串；第四个部分长 15 个字节，值为长度在 4 到 15 之间且向左对齐的随机字符串。

5.1.22 *运行时间*。使用多种键生成器比较高位优先的字符串排序与三向字符串快速排序的运行时间。对于定长的键，在比较中加入低位优先的字符串排序算法。

5.1.23 *数组访问*。使用多种键生成器比较高位优先的字符串排序与三向字符串快速排序的数组访问次数。对于定长的键，在比较中加入低位优先的字符串排序算法。

5.1.24 *被访问的最靠右的字符*。使用多种键生成器比较高位优先的字符串排序与三向字符串快速排序能够访问到的最靠右的字符的位置。

728

5.2 单词查找树

和排序一样，我们也可以利用字符串的性质开发比第3章中介绍的通用算法更有效的查找算法，以便用于以字符串作为被查找的键的一般应用程序。

具体来说，本节中所讨论的算法在一般应用场景中（甚至对于巨型的符号表）都能够取得以下性能：

- 查找命中所需的时间与被查找的键的长度成正比；
- 查找未命中只需检查若干个字符。

仔细思考过后你会发现，这样的性能是相当惊人的。它们是算法研究的最高成就之一，也是建成现今能够便捷、快速地访问海量信息所依赖的基础设施的重要因素。更重要的是，我们可以扩展符号表的API，添加基于字符的用于处理字符串类型的键的操作（但不必为所有Comparable类型的键都添加类似操作）。它们在实际应用中非常强大并实用，如表5.2.1所示。

表5.2.1 以字符串为键的符号表的API

public class	**StringST<Value>**	
	StringST()	创建一个符号表
void	put(String key, Value val)	向表中插入键值对（如果值为null则删除键key）
Value	get(String key)	键key所对应的值（如果键不存在则返回null）
void	delete(String key)	删除键key（和它的值）
boolean	contains(String key)	表中是否保存着key的值
boolean	isEmpty()	符号表是否为空
String	longestPrefixOf(String s)	s的前缀中最长的键
Iterable<String>	keysWithPrefix(String s)	所有以s为前缀的键
Iterable<String>	keysThatMatch(String s)	所有和s匹配的键（其中“.”能够匹配任意字符）
int	size()	键值对的数量
Iterable<String>	keys()	符号表中的所有键

729
~
730

这份API与第3章中所介绍的符号表API有以下不同：

- 将泛型的Key的类型换成了具体的类型String；
- 添加了3个方法：longestPrefixOf()、keysWithPrefix()和keysThatMatch()。

本节仍然遵守第3章中实现符号表时的几个基本约定（不接受重复键或空键，值不能为空）。

从对字符串的排序算法中可以看到，指定字符串的字母表常常是十分重要的。对小型字母表的简单而高效的实现不适用于大型字母表，这是因为后者消耗的空间太多。在这种情况下，应该添加一个构造函数，允许用例指定所使用的字母表。我们会在本节稍后讨论这个构造函数的实现，但目前暂时没有在API中列出它，因为要将精力集中在字符串类型的键上。

下面我们用she sells sea shells by the sea shore这几个键作为示例描述以下3个新方法。

- longestPrefixOf()接受一个字符串参数并返回符号表中该字符串的前缀中最长的键。对于以上所有键，longestPrefixOf("shell")的结果是she，longestPrefixOf("shellsort")的结果是shells。
- keysWithPrefix()接受一个字符串参数并返回符号表中所有以该字符串作为前缀的键。对于以上所有键，keysWithPrefix("she")的结果是she和shells，keysWithPrefix ("se")

的结果是 sells 和 sea。

- keysThatMatch() 接受一个字符串参数并返回符号表中所有和该字符串匹配的键，其中参数字符串中的点（“.”）可以匹配任何字符。对于以上所有键，keysThatMatch(".he") 的结果是 she 和 the，keysThatMatch("s..") 的结果是 she 和 sea。

在见过这些基本的符号表方法后，我们将详细讨论这些操作的的实现和应用。这些特别的操作是字符串类型的键所可能进行的操作中的代表操作，我们将会在练习中讨论其他可能的操作。

为了突出中心思想，本节的重点是 put()、get() 和新增的几个方法；（和第 3 章一样）使用了 contains() 和 isEmpty() 的默认实现，并将 size() 和 delete() 的实现留作练习。因为字符串都是 Comparable 的，所以可以在 API 中包含第 3 章有序符号表 API 中的各种有序性操作（非常值得这样做）。我们将它们的实现（大多都很简单）留作练习并放在了本书的网站上。 731

5.2.1 单词查找树

本节中，我们要学习一种叫做*单词查找树*的数据结构。它由字符串键中的所有字符构造而成，允许使用被查找键中的字符进行查找。它的英文单词 trie 来自于 E.Fredkin 在 1960 年玩的一个文字游戏，因为这个数据结构的作用是取出（retrieval）数据，但发音为 try 是为了避免与 tree 相混淆。我们首先会描述单词查找树的基本性质，包括查找和插入算法，然后详细学习它的数据表示方法和 Java 实现。

5.2.1.1 基本性质

和各种查找树一样，单词查找树也是由链接的结点所组成的数据结构，这些链接可能为空，也可能指向其他结点。每个结点都只可能有一个指向它的结点，称为它的*父结点*（只有一个结点除外，即*根结点*，没有任何结点指向根结点）。每个结点都含有 R 条链接，其中 R 为字母表的大小。单词查找树一般都含有大量的空链接，因此在绘制一棵单词查找树时一般会忽略空链接。尽管链接指向的是结点，但是也可以看作链接指向的是另一棵单词查找树，它的根结点就是被指向的结点。每条链接都对应着一个字符——因为每条链接都只能指向一个结点，所以可以用链接所对应的字符标记被指向的结点（根结点除外，因为没有链接指向它）。每个结点也含有一个相应的值，可以是空也可以是符号表中的某个键所关联的值。具体来说，我们将每个键所关联的值保存在该键的最后一个字母所对应的结点中。我们应该记住非常重要的一点：*值为空的结点在符号表中没有对应的键，它们的存在是为了简化单词查找树中的查找操作*。一棵单词查找树的例子如图 5.2.1 所示。

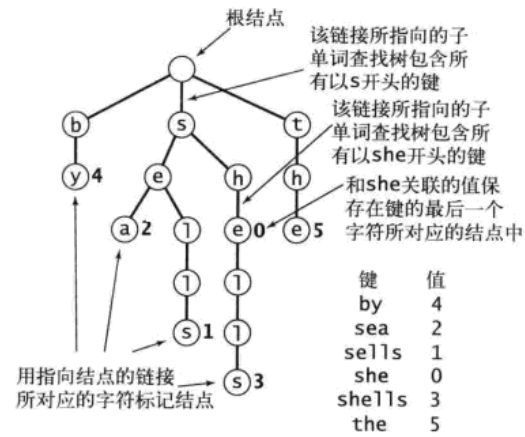


图 5.2.1 单词查找树详解

5.2.1.2 单词查找树中的查找操作

在单词查找树中查找给定字符串键所对应的值是一个很简单的过程，它是以被查找的键中的字符为导向的。单词查找树中的每个结点都包含了下一个可能出现的所有字符的链接。从根结点开始，首先经过的是键的首字母所对应的链接；在下一个结点中沿着第二个字符所对应的链接继续前进； 732

在第二个结点中沿着第三个字符所对应的链接向前，如此这般直到到达键的最后一个字母所指向的结点或是遇到了一条空链接。这时可能会出现以下 3 种情况（示例请见图 5.2.2）。

- 键的尾字符所对应的结点中的值非空（如图 5.2.2 中查找 shells 和 she 的示例）。这是一次命中的查找——键所对应的值就是键的尾字符所对应的结点中保存的值。
- 键的尾字符所对应的结点中的值为空（如图 5.2.2 中查找 shell 的示例）。这是一次未命中的查找——符号表中不存在被查找的键。
- 查找结束于一条空链接（如图 5.2.2 中查找 shore 的示例）。这也是一次未命中的查找。

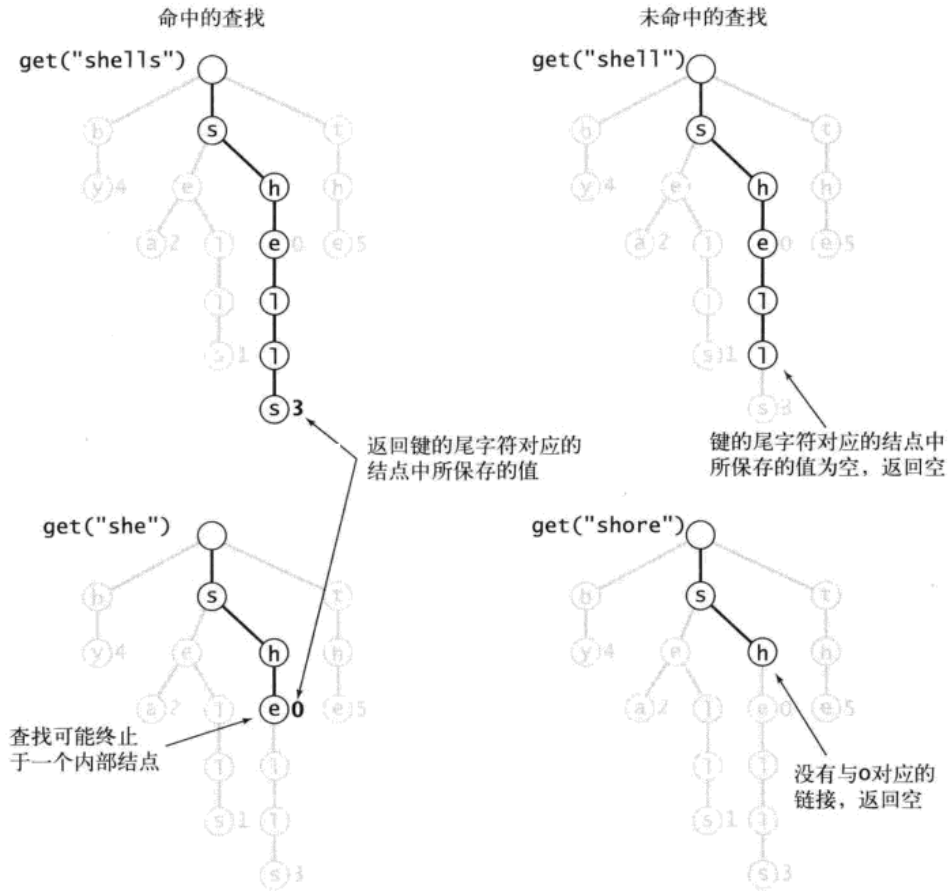


图 5.2.2 单词查找树的查找示例

733 在所有的情况中，执行查找的方式就是在单词查找树中从根结点开始检查某条路径上的所有结点。

5.2.1.3 单词查找树中的插入操作

和二叉查找树一样，在插入之前要进行一次查找：在单词查找树中意味着沿着被查找的键的所有字符到达树中表示尾字符的结点或者一个空链接。此时可能会出现以下两种情况。

- 在到达键的尾字符之前就遇到了一个空链接。在这种情况下，字符查找树中不存在与键的尾字符对应的结点，因此需要为键中还未被检查的每个字符创建一个对应的结点并将键的值保存到最后一个字符的结点中。

- 在遇到空链接之前就到达了键的尾字符。在这种情况下，和关联性数组一样，将该结点的值设为键所对应的值（无论该值是否为空）。

在所有情况下，我们都会检查键中的每个字符并为它们在树中创建一个对应的结点。在使用第3章中的标准索引用例处理输入 `she sells sea shells by the sea shore` 时所构造的单词查找树如图 5.2.3 所示。

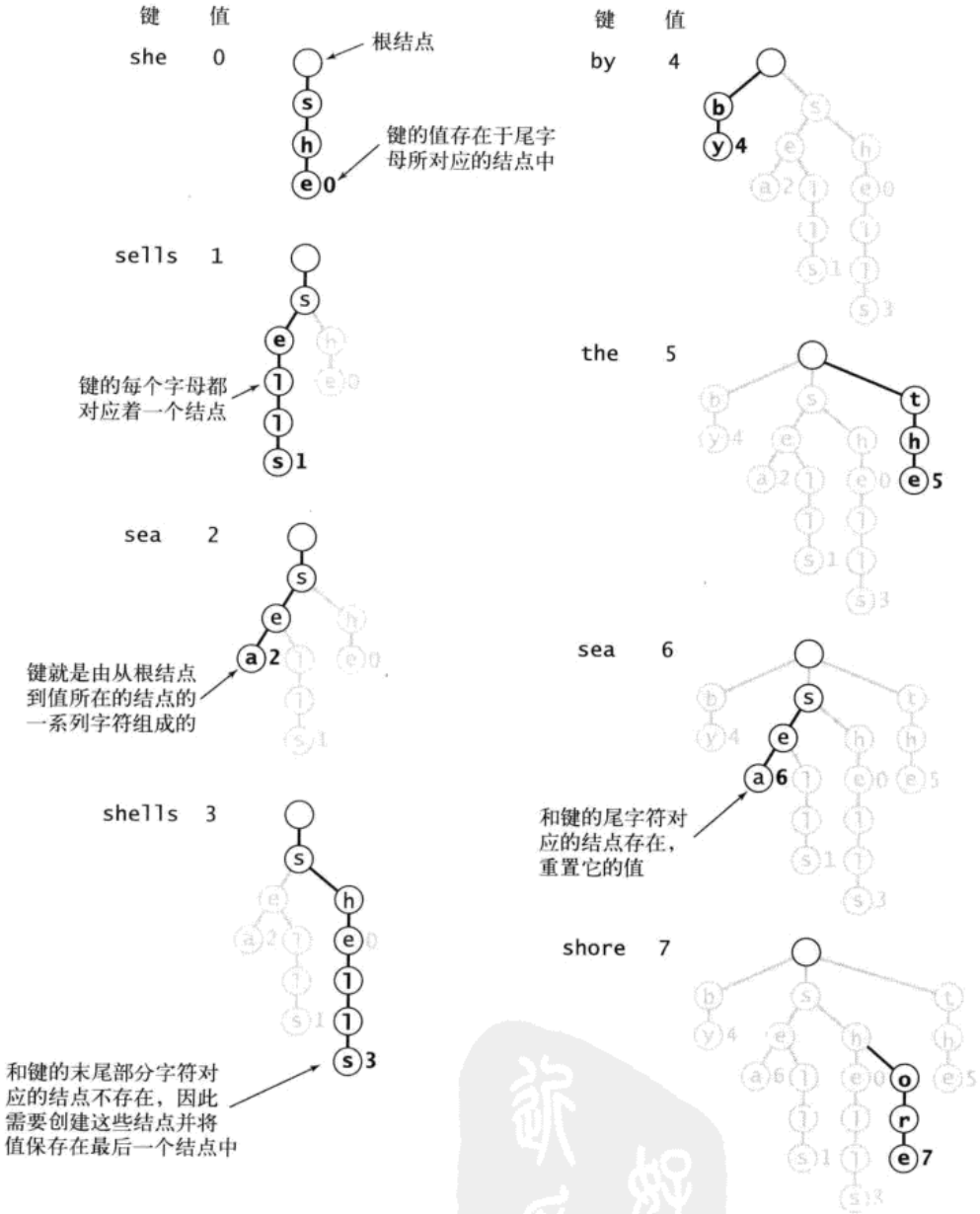


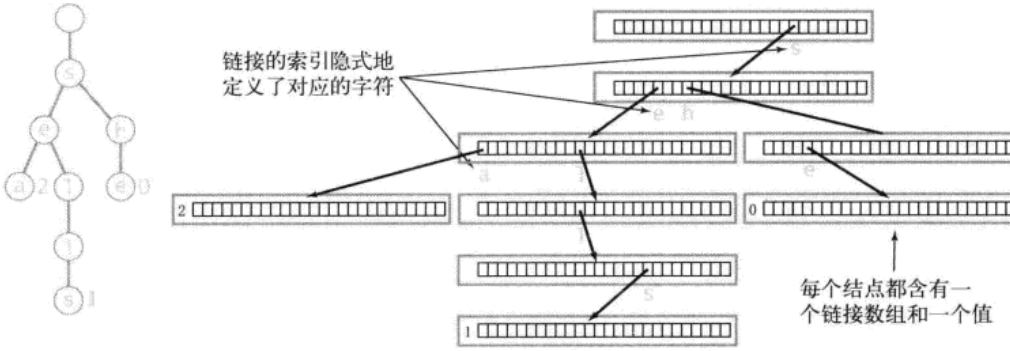
图 5.2.3 标准索引用例中单词查找树的构造轨迹

5.2.1.4 结点的表示

在本节开头提到过，我们为单词查找树所绘出的图像和在程序中构造的数据结构并不完全一致，因为我们没有画出空链接。将空链接考虑进来将会突出单词查找树的以下重要性质：

- ❑ 每个结点都含有 R 个链接，对应着每个可能出现的字符；
- ❑ 字符和键均隐式地保存在数据结构中。

例如，在图 5.2.4 中的单词查找树中，所有的键均由小写字母组成，每个结点都含有一个值和 26 个链接。第一条链接指向的子单词查找树中的所有键的首字母都是 a，第二条链接指向的子单词查找树中的所有键的首字母都是 b，等等。



734
⌇
735

图 5.2.4 单词查找树的表示（R=26）

在单词查找树中，键是由从根结点到含有非空值的结点的路径所隐式表示的。例如，在单词查找树中，字符串 sea 所关联的值是 2，因为根结点中的第 19 条链接（指向由所有以 s 开头的键组成的子单词查找树）非空，下一个结点中的第 5 条链接（指向由所有以 se 开头的键组成的子单词查找树）非空，第三个结点中的第 1 条链接（指向由所有以 sea 开头的键组成的子单词查找树）的值为 2。数据结构既没有保存字符串 sea 也没有保存字符 s、e 和 a。事实上，数据结构不会存储任何字符串或字符，它保存了链接数组和值。因为参数 R 的作用的重要性，所以将基于含有 R 个字符的字母表的单词查找树称为 R 向单词查找树。

有了这些预备知识之后，算法 5.4 实现的符号表 TrieST 就很容易理解了。它也使用了类似于第 3 章介绍的查找树使用的递归方法。它的私有 Node 类用实例变量 val 保存键相关联的值并用数组 next[] 保存所有指向其他 Node 对象的引用。这些递归方法的实现非常简洁，值得仔细研究。下面，我们将讨论接受一个 Alphabet 对象作为参数的构造函数和 size()、keys()、longestPrefixOf()、keysWithPrefix()、keysThatMatch() 和 delete() 方法的实现。理解这些递归方法也并不困难，只是每个方法都会比前一个稍加复杂。

```
public int size()
{  return size(root);  }

private int size(Node x)
{
   if (x == null) return 0;

   int cnt = 0;
   if (x.val != null) cnt++;
   for (char c = 0; c < R; c++)
      cnt += size(next[c]);

   return cnt;
}
```

单词查找树的延时递归方法size()

5.2.1.5 大小

和第 3 章中的二叉查找树一样，size() 方法的实现有以下 3 种显而易见的选择。

- ❑ 即时实现：用一个实例变量 N 保存键的数量。

- 更加即时的实现：用结点的实例变量保存子单词查找树中键的数量，在递归的 put() 和 delete() 方法调用之后更新它们。
- 延时递归实现：如上页框注“单词查找树的延时递归方法 size()”所示。它会遍历单词查找树中的所有结点并记录非空值结点的总数。

和二叉查找树一样，延时实现很有指导意义但是应该尽量避免，因为它会给用例造成性能上的问题。我们会在练习中讨论它的即时实现。 736

算法 5.4 基于单词查找树的符号表

```
public class TrieST<Value>
{
   private static int R = 256; // 基数
   private Node root;          // 单词查找树的根结点

   private static class Node
   {
      private Object val;
      private Node[] next = new Node[R];
   }

   public Value get(String key)
   {
      Node x = get(root, key, 0);
      if (x == null) return null;
      return (Value) x.val;
   }

   private Node get(Node x, String key, int d)
   {  // 返回以x作为根结点的子单词查找树中与key相关联的值
      if (x == null) return null;
      if (d == key.length()) return x;
      char c = key.charAt(d); // 找到第d个字符所对应的子单词查找树
      return get(x.next[c], key, d+1);
   }

   public void put(String key, Value val)
   {  root = put(root, key, val, 0);  }

   private Node put(Node x, String key, Value val, int d)
   {  // 如果key存在于以x为根结点的子单词查找树中则更新与它相关联的值
      if (x == null) x = new Node();
      if (d == key.length()) {  x.val = val; return x; }
      char c = key.charAt(d); // 找到第d个字符所对应的子单词查找树
      x.next[c] = put(x.next[c], key, val, d+1);
      return x;
   }
}
```

这份代码使用 R 向单词查找树实现了符号表。我们会在下面的几页中讨论表 5.2.1 中字符串符号表 API 中新增的方法。我们很容易通过修改这段代码来处理特殊字母表中的键（请见 5.2.1.8 节）。因为 Java 不支持泛型数组，所以 Node 中的值的类型必须是 Object，可以在 get() 中将值的类型转换为 Value。 737

5.2.1.6 查找所有键

因为字符和键是被隐式地表示在单词查找树中，所以使用例能够遍历符号表的所有键就变得有些困难。在二叉查找树中，我们将所有字符串键保存在一个队列（Queue）里。但对于单词查找树，不仅要能够在数据结构中找到这些键，还需要显式地表示它们。我们用一个类似于 size() 的私有递归方法 collect() 来完成这个任务，它维护了一个字符串用来保存从根结点出发的路径上的一系列字符。每当我们在 collect() 调用中访问一个结点时，方法的第一个参数就是该结点，第二个参数则是和该结点相关联的字符串（从根结点到该结点的路径上的所有字符）。在访问一个结点时，如果它的值非空，我们就将和它相关联的字符串加入队列之中，然后（递归地）访问它的链接数组所指向的所有可能的字符结点。在每次调用之前，都将链接对应的字符附加到当前键的末尾作为调用的参数键。用这个 collect() 方法为 API 中的 keys() 和 keysWithPrefix() 方法收集符号表中所有的键。要实现 keys() 方法，可以以空字符串作为参数调用 keysWithPrefix() 方法。要实现 keysWithPrefix() 方法，可以先调用 get() 找出给定前缀所对应的单词查找树（如果不存在则返回 null），再使用 collect() 方法完成任务。图 5.2.5 显示了 collect() 方法（或者说 keysWithPrefix("") 调用）在一棵单词查找树中的轨迹，它给出了每次调用 collect() 方法时第二个参数的值和队列的内容。图 5.2.6 显示了 keysWithPrefix("sh") 的运行过程。

```
public Iterable<String> keys()
{  return keysWithPrefix("");  }

public Iterable<String>
keysWithPrefix(String pre)
{
   Queue<String> q = new Queue<String>();
   collect(get(root, pre, 0), pre, q);
   return q;
}

private void collect(Node x, String pre,
                     Queue<String> q)
{
   if (x == null) return;
   if (x.val != null) q.enqueue(pre);
   for (char c = 0; c < R; c++)
      collect(x.next[c], pre + c, q);
}
```

收集一棵单词查找树中的所有键

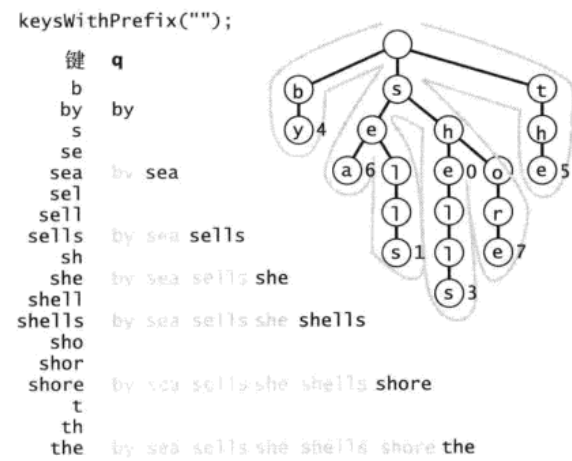


图 5.2.5 收集一棵单词查找树中的所有键的轨迹

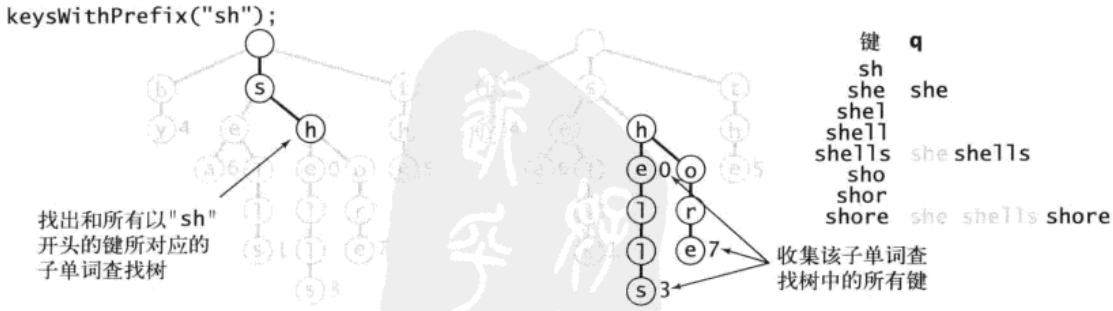


图 5.2.6 单词查找树中的前缀匹配

5.2.1.7 通配符匹配

我们可以用一个类似的过程实现 keysThatMatch()，但需要为 collect() 方法添加一个参数来指定匹配的模式。如果模式中含有通配符，就需要用递归调用处理所有的链接，否则就只需要处理模式中指定字符的链接即可，如下方的框注所示。你还可以注意到，这里不需要考虑长度超过模式字符串的键。

```
public Iterable<String> keysThatMatch(String pat)
{
   Queue<String> q = new Queue<String>();
   collect(root, "", pat, q);
   return q;
}
public void collect(Node x, String pre, String pat, Queue<String> q)
{
   int d = pre.length();
   if (x == null) return;
   if (d == pat.length() && x.val != null) q.enqueue(pre);
   if (d == pat.length()) return;
   char next = pat.charAt(d);
   for (char c = 0; c < R; c++)
      if (next == '.' || next == c)
         collect(x.next[c], pre + c, pat, q);
}
```

单词查找树中的通配符匹配

5.2.1.8 最长前缀

为了找到给定字符串的最长键前缀，就需要使用一个类似于 get() 的递归方法。它会记录查找路径上所找到的最长键的长度（将它作为递归方法的参数在遇到值非空的结点时更新它）。查找会在被查找的字符串结束或是遇到空链接时终止，请见图 5.2.7。

```
public String longestPrefixOf(String s)
{
   int length = search(root, s, 0, 0);
   return s.substring(0, length);
}
private int search(Node x, String s, int d, int length)
{
   if (x == null) return length;
   if (x.val != null) length = d;
   if (d == s.length()) return length;
   char c = s.charAt(d);
   return search(x.next[c], s, d+1, length);
}
```

对给定字符串的最长前缀进行匹配

5.2.1.9 删除操作

从一棵单词查找树中删去一个键值对的第一步是，找到键所对应的结点并将它的值设为空（null）。如果该结点含有一个非空的链接指向某个子结点，那么就不需要在进行其他操作了。如果它的所有链接均为空，那就需要从数据结构中删去这个结点。如果删去它使得它的父结点的所有链接也均为空，就需要继续删除它的父结点，依此类推。如下面框注中的实现所示，根据标准递归流程，这项操作所需的代码极少：在递归删除了某个结点 x 之后，如果该结点的值和所有的链接均为空则返回 null，否则返回 x，请见图 5.2.8。

```
public void delete(String key)
{  root = delete(root, key, 0);  }

private Node delete(Node x, String key, int d)
{
   if (x == null) return null;
   if (d == key.length())
      x.val = null;
   else
   {
      char c = key.charAt(d);
      x.next[c] = delete(x.next[c], key, d+1);
   }
   if (x.val != null) return x;
   for (char c = 0; c < R; c++)
      if (x.next[c] != null) return x;
   return null;
}
```

从单词查找树中删除一个键（和它相关联的值）

5.2.1.10 字母表

和以前一样，算法 5.4 处理的是 Java 的 String 类型的键，但将它修改为处理由任意字母表得到的键也很容易。

- 实现一个构造函数，接受一个 Alphabet 对象作为参数，将一个 Alphabet 类型的实例变量设为该参数的值并将实例变量 R 的值设为字母表中字母的个数。
- 在 get() 和 put() 中使用 Alphabet 类的 toIndex() 方法，将字符串中的字符转化为 0 到 R–1 之间的索引值。
- 使用 Alphabet 类的 toChar() 方法，将 0 到 R–1 之间的索引值转化为字符型（char）的值。get() 和 put() 方法不需要进行此操作，但它在 keys()、keysWithPrefix() 和 keysThatMatch() 方法的实现中很重要。

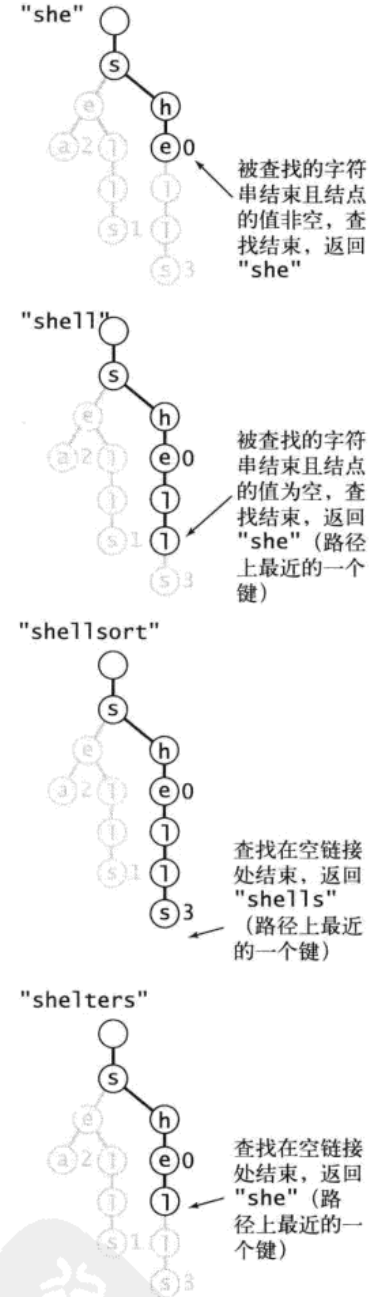


图 5.2.7 longestPrefixOf() 方法的各种可能情况

经过这些修改，如果已知所有键仅来自于一个小型的字母表，那可以节省相当大的空间（在每个结点中仅使用 R 条链接），代价是字母和索引相互转化所需要的时间。

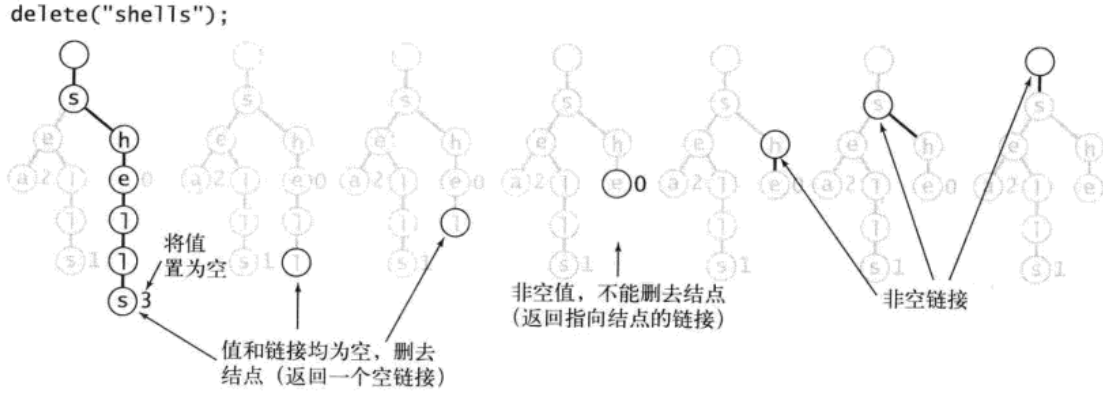


图 5.2.8 从单词查找树中删除一个键（和它相关联的值）

739
↓
741

框注“从单词查找树中删除一个键（和它相关联的值）”就是字符串符号表 API 的一个简洁而完整的实现，它适用于各种实际应用场景。本节的练习讨论了它的几种变化和扩展。下面我们要讨论单词查找树的基本性质和限制条件。

5.2.2 单词查找树的性质

和以前一样，我们希望知道在一般的应用程序中使用单词查找树所需要的时间和空间。单词查找树已经被分析和研究得很透彻了，它的基本性质也比较容易理解和应用。

命题 F。单词查找树的链表结构（形状）和键的插入或删除顺序无关：对于任意给定的一组键，其单词查找树都是唯一的。

证明。由数学归纳法很容易通过子单词查找树证明这个结论。

这个基本的结论是单词查找树的一个特殊性质：我们目前已经学过的所有其他结构的查找树的构造都不仅和键的集合有关，而且还取决于这些键的插入顺序。

5.2.2.1 最坏情况下查找和插入操作的时间界限

在单词查找树中找到给定键的值要花多长时间？对于二叉查找树、散列表和第 3 章中所介绍的其他算法，都需要使用数学分析来回答这个问题。但是对于单词查找树，这个问题很简单。

命题 G。在单词查找树中查找一个键或是插入一个键时，访问数组的次数最多的键的长度加 1。

证明。由代码可知，`put()` 和 `get()` 方法的递归实现都带有一个参数 `d`。它的初始值为 0，每次调用时都会加 1，当长度等于键的长度时递归调用停止。

从理论角度来说，命题 G 意味着单词查找树对于命中的查找是最理想的——我们不能奢望查找所需的时间比与被查找的键的长度成正比更好。无论使用的是什么算法和数据结构，在检查完要查找的键中的所有字符之前都是无法判断是否已找到该键。从实际角度来说，这个保证也很重要，因为它和符号表中键的数量无关：当我们在处理类似于车牌号码的 7 个字符的键时，可以知道查找或插入操作最多只需要检查 8 个结点；当我们在处理 20 个字符的数字账号时，最多只需要检查 21 个结点就可以完成查找或插入操作。

5.2.2.2 查找未命中的预期时间界限

假设我们正在单词查找树中查找一个键，发现根结点中与被查找键的第一个字符所对应的链接为空。此时只检查了一个结点就知道了该键不存在于表中。这种情况是很常见的：单词查找树的最重要的性质之一就是未命中的查找一般都只需要检查很少的几个结点。如果假设键都来自于随机字
742 符串模型（R 中的所有不同字符出现的几率均相同），可以证明以下结论。

命题 H。 字母表的大小为 R，在一棵由 N 个随机键构造的单词查找树中，未命中查找平均所需检查的结点数量为 $\sim\log_R N$。

简略证明（写给熟悉概率分析的读者）。所有的 N 个键都与一个随机的查找键的前 t 个字符中至少有一个字符不同的概率为 $(1-R^{-t})^N$。用 1 减去它即可得到单词查找树中至少有一个键和被查找键的前 t 个字符都相匹配的概率。也就是说，$1-(1-R^{-t})^N$ 的查找操作至少需要比较 t 个字符的概率。在概率分析中，对于 t=0,1,2…，一个整数随机变量大于 t 的概率之和就是该随机变量的平均值。因此，查找的平均成本为：

$$1-(1-R^{-1})^N+1-(1-R^{-2})^N+\cdots+1-(1-R^{-t})^N+\cdots$$

根据基本的近似公式 $(1-1/x)^x\sim e^{-1}$，查找的平均成本的近似函数为：

$$1-(1-e^{-N/R^1})+1-(1-e^{-N/R^2})+\cdots+(1-e^{-N/R^t})+\cdots$$

当 R^t 远小于 N 时，相对应的约 $\ln_R N$ 项的值非常接近于 1；当 R^t 远大于 N 时，所对应的所有的项的值均极为接近于 0；当 $R^t\approx N$ 时，所对应的项不多且它们的值均在 0 和 1 之间。因此，它的总和约为 $\log_R N$。

从实际角度来说，该命题说明的最重要的一点就是，查找未命中的成本与键的长度无关。例如，它说明在一棵由 100 万个随机键构造出的单词查找树中，未命中的查找也只需要检查 3~4 个结点，无论这些键是含有 7 个数字的车辆牌照还是 20 个数字的账号。虽然在实际应用中真正的随机键是不
743 可能出现的，但该模型能够描述一般应用场景中单词查找树算法对键的处理方式，上述猜想是合理的。事实上，这种行为方式在实际应用中十分常见而且也是单词查找树得到广泛应用的一个重要原因。

5.2.2.3 空间

一棵单词查找树需要多少空间？回答这个问题（了解可用的空间有多少）是有效使用单词查找树的关键。

命题 I。 一棵单词查找树中的链接总数在 RN 到 RNw 之间，其中 w 为键的平均长度。

证明。 在单词查找树中，每个键都有一个对应的结点保存着它关联的值，同时每个结点也含有 R 条链接，因此链接总数至少有 RN 条。如果所有的键的首字母均不相同，那么每个键中的每个字母都有一个对应的结点，因此链接总数应该等于 R 乘以所有键中的字符总数，即 RNw。

表 5.2.2 说明了我们所讨论的一些典型的应用场景所需的空间成本。它说明了单词查找树中的一些经验性的规律。

- 当所有键均较短时，链接的总数接近于 RN；
- 当所有键均较长时，链接的总数接近于 RNw；

❑ 因此，缩小 R 能够节省大量的空间。

这张表传递出的另一条更加微妙的信息是，在实际应用中采用单词查找树之前了解将要被插入的所有键的性质是非常重要的。

表 5.2.2 典型的单词查找树的空间需求

应　用	典型的键	平均长度 w	字母表大小 R	100 万个键所构造的单词查找树中的链接总数
加利福尼亚州的车牌号	4PGC938	7	256	2 亿 5 千 6 百万
数字账号	02400019992993299111	20	256 10	40 亿 2 亿 5 千 6 百万
URL	www.cs.princeton.edu	28	256	40 亿
文本处理	seashells	11	256	2 亿 5 千 6 百万
基因组数据中的蛋白质	ACTGACTG	8	256 4	2 亿 5 千 6 百万 40 亿

5.2.2.4 单向分支

长键在单词查找树中占用了大量空间的主要原因是，树中的长键通常都有一条长长的“尾巴”，其中每个结点都只含有一条指向下一个结点的链接（因此都含有 R–1 条空链接）。这种情况并不难纠正（请见练习 5.2.11 和图 5.2.9）。单词查找树的内部也可能存在单向的分支。例如，两个长键可能只有最后一个字符不同。解决这种情况要更加困难一些（请见练习 5.2.12）。这些修改能够使得单词查找树的空间消耗比已经讨论过的简单实现缩小许多，但它们对于实际应用场景基本不起作用。下面我们将学习降低单词查找树的空间消耗的另一种方式。

我们的底线是：*不要使用算法 5.4 处理来自于大型字母表的大量长键*。它所构造的单词查找树所需要的空间与 R 和所有键的字符总数之积成正比。但是，如果你能够负担得起这么庞大的空间，单词查找树的性能是无可匹敌的。

```
put("shells", 1);
put("shellfish", 2);
```

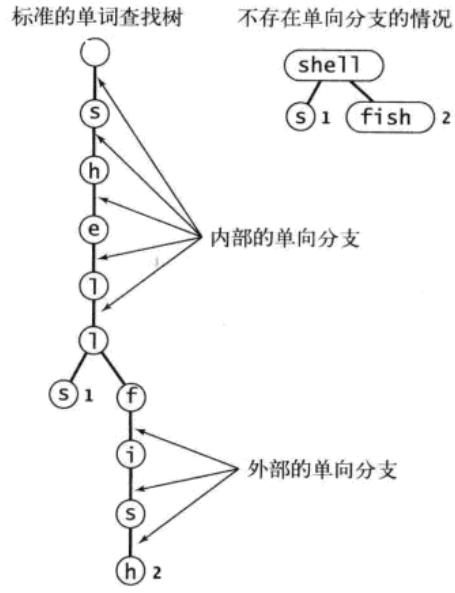


图 5.2.9 消除单词查找树中的单向分支

744
~
745

5.2.3 三向单词查找树

为了避免 R 向单词查找树过度的空间消耗，我们现在来学习另一种数据的表示方法：三向单词查找树（TST）。在三向单词查找树中，每个结点都含有一个字符、三条链接和一个值。这三条链接分别对应着当前字母小于、等于和大于结点字母的所有键。在算法 5.4 的 R 向单词查找树中，树的结点含有 R 条链接，每个非空链接的索引隐式地表示了它所对应的字符。在等价的三向单词查找树中，字符是显式地保存在结点中的——只有在沿着中间链接前进时才会根据字符找到表中的键，请见图 5.2.10。

查找与插入操作

用三向单词查找树实现符号表 API 中的查找和插入操作很简单。在查找时，我们首先比较键的首字母和根结点的字母。如果键的首字母较小，就选择左链接；如果较大，就选择右链接；如果相等，

则选择中链接。然后，递归地使用相同的算法。如果遇到了一个空链接或者当键结束时结点的值为空，那么查找未命中；如果键结束时结点的值非空则查找命中。在插入一个新键时，首先进行查找，然后和在单词查找树一样，在树中补全键末尾的所有结点。算法 5.5 给出了这些方法的实现细节。

这种实现方式等价于将 R 向单词查找树中的每个结点实现为以非空链接所对应的字符作为键的二叉查找树。不同的是，算法 5.4 使用的是由键索引的数组。图 5.2.10 显示了一棵单词查找树和与它相对应的三向单词查找树。按照第 3 章中所述的二叉查找树和其他排序算法之间的对应关系来看，我们可以发现三向单词查找树与三向字符串快速排序之间的对应关系与二叉查找树与快速排序以及单词查找树与高位优先的排序之间的对应关系是一样的。图 5.1.12 和图 5.1.17 分别显示了高位优先的字符串排序和三向字符串快速排序的递归调用结构，它们与图 5.2.10 中由同一组键所构造的单词查找树和三向单词查找树正好完全对应。单词查找树中的链接所占用的空间即为高位优先的字符串排序中的计数器所占用的空间。三向分支为两者都提供了一个非常有效的解决方案，请见图 5.2.11 和图 5.2.12。

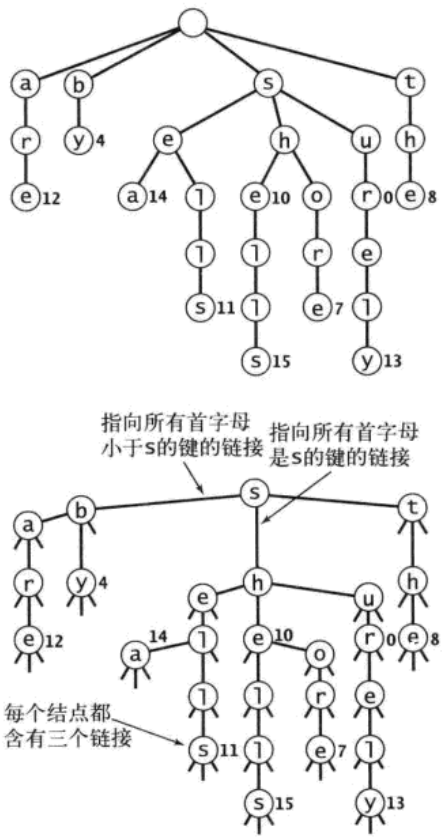


图 5.2.10 一棵单词查找树所对应的三向单词查找树

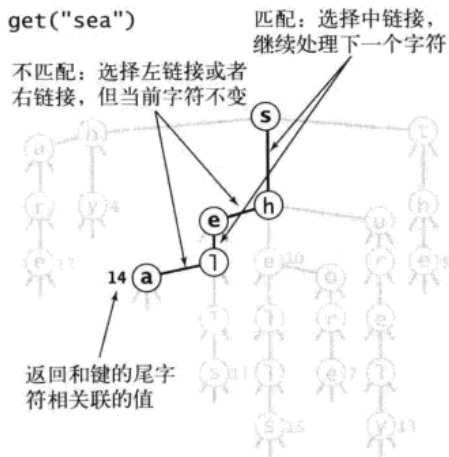


图 5.2.11 三向单词查找树中的查找示例

算法 5.5 基于三向单词查找树的符号表

```
public class TST<Value>
{
   private Node root;                  // 树的根结点
   private class Node
   {
      char c;                          // 字符
      Node left, mid, right;           // 左中右子三向单词查找树
      Value val;                       // 和字符串相关联的值
   }
```

```
   public Value get(String key)  // 和单词查找树相同（请见算法5.4）

   private Node get(Node x, String key, int d)
   {
      if (x == null) return null;
      char c = key.charAt(d);
      if      (c < x.c) return get(x.left,  key, d);
      else if (c > x.c) return get(x.right, key, d);
      else if (d < key.length() - 1)
                        return get(x.mid,   key, d+1);
      else return x;
   }

   public void put(String key, Value val)
   {  root = put(root, key, val, 0);  }

   private Node put(Node x, String key, Value val, int d)
   {
      char c = key.charAt(d);
      if (x == null) { x = new Node(); x.c = c; }
      if      (c < x.c) x.left  = put(x.left,  key, val, d);
      else if (c > x.c) x.right = put(x.right, key, val, d);
      else if (d < key.length() - 1)
                        x.mid   = put(x.mid,   key, val, d+1);
      else x.val = val;
      return x;
   }
}
```

这段实现使用含有一个 char 类型的值 c 和三条链接的结点构建了三向单词查找树，其中子树的键的首字母分别小于（左子树）、等于（中子树）和大于（右子树）c。

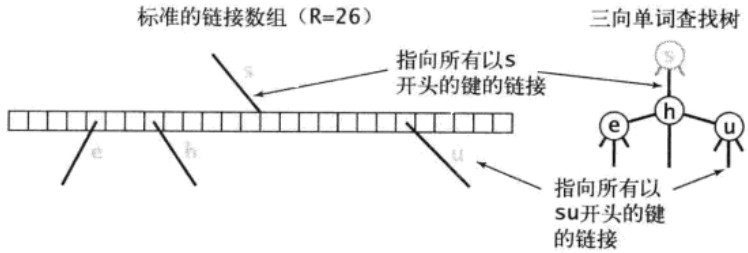


图 5.2.12 单词查找树结点示例

746
~
748

5.2.4 三向单词查找树的性质

三向单词查找树是 *R* 向单词查找树的紧凑表示，但两种数据结构的性质截然不同。这其中最重要的不同可能在于命题 A 对于三向单词查找树不再成立：和其他所有二叉查找树一样，每个单词查找树结点的二叉查找树表示也取决于键的插入顺序。

5.2.4.1 空间

三向单词查找树最重要的性质就是每个结点只含有三个链接，因此三向单词查找树所需要空间远小于对应的单词查找树。

命题 J。由 N 个平均长度为 w 的字符串构造的三向单词查找树中的链接总数在 $3N$ 到 $3Nw$ 之间。

证明。同命题 I。

三向单词查找树实际使用的内存空间一般都低于由每个字符三个链接得到的上界，因为有相同前缀的键会共享树中的高层结点。

5.2.4.2 查找成本

要计算三向单词查找树中查找（和插入）操作的成本，需要将它所对应的单词查找树中的查找成本乘以遍历每个结点的二叉查找树所需的成本。

命题 K。在一棵由 N 个随机字符串构造的三向单词查找树中，查找未命中平均需要比较字符 ~ $\ln N$ 次。除 ~ $\ln N$ 次外，一次插入或命中的查找会比较一次被查找的键中的每个字符。

证明。由代码我们马上可以得到插入和查找命中的成本。查找未命中的成本的证明和命题 H 的简略证明相同。假设在查找路径上除了常数个结点（高层的几个）之外的其他所有结点均为由 R 个字符值随机构造的二叉查找树，且树的平均路径长度为 $\ln R$，因此将时间成本 $\log_R N = \ln N / \ln R$ 乘以 $\ln R$。

在最坏情况下，一个结点可能变成一个完全的 R 向结点，不平衡且像一条链表一样展开，因此需要乘以一个系数 R。一般的情况下，在第一层（因为根结点类似于一棵由 R 个不同的值组成的随机二叉查找树）甚至是其下的几层（如果键存在公共的前缀且前缀之后的字符最多可能有 R 种不同的取值）那么进行字符比较的次数将是 $\ln R$ 或者更少，之后对于大多数字符也只需进行几次比较（因为指向大多数单词查找树结点的非空链接的分布十分稀疏）。未命中的查找一般都需要若干次字符
749 比较并结束于单词查找树高层的某个空链接。在命中的查找中，被查找的键中的每个字符都需要并且只需要一次比较，因为它们大多数都是单词查找树底部的单向分支上的结点。

5.2.4.3 字母表

使用三向单词查找树的最大好处是它能够很好地适应实际应用中可能出现的被查找键的不规则性。需要特别注意到的是，不应该按照用例提供的字母表构造字符串，这对于单词查找树很关键。这主要会产生两点影响。首先，实际应用中的键都来自于大型字母表，而且字符集中的各个字符的使用是非常不均衡的。有了三向单词查找树，我们可以使用 256 个字符的 ASCII 编码或者 65 536 个字符的 Unicode 编码，而不必担心 256 向分支或者 65 536 向分支带来的巨大开销，也不必判断哪些才是相关的字符集。非罗马字母表的 Unicode 字符串中可能含有上千种字符——三向单词查找树特别适合于可能含有此类字符的 Java 标准 `String` 类型的键。其次，实际应用程序中的键常常有着类似的结构，这在不同的应用之中可能不同。键的一部分可能只会使用字母，而另一部分可能只会使用数字。在加利福尼亚州的车牌号的例子中，第二、三、四个字符都是大写字母（R=26），而其他字符都是数字（R=10）。在这种键构造的三向单词查找树中，一部分结点会被表示为 10 结点的二叉查找树（键的数字部分），另一部分结点会被表示为 26 结点的二叉查找树（键的字母部分）。这种结构的生成是自动的，无需对键进行特别分析。

5.2.4.4 前缀匹配、查找所有键和通配符匹配

因为三向单词查找树也是单词查找树，前文中单词查找树的 `longestPrefixOf()`、`keys()`、

keysWithPrefix() 和 keyThatMatch() 方法的实现可以很容易移植过来。这个练习能够加深你对单词查找树和三向单词查找树的理解（请见练习 5.2.9）。和查找操作一样，这里也存在空间和时间的交换（使用线性级别的内存空间，但每个字符的比较次数需要乘以 $\ln R$）。

5.2.4.5 删除操作

三向单词查找树中的 delete() 方法要更复杂一些。从本质上来说，每个将被删除的字符都属于一棵二叉查找树。在单词查找树中，只需将链接数组中和该字符对应的元素置为空即可删去它的链接。在三向单词查找树中，需要用在二叉查找树中删除结点的方法来删去与该字符对应的结点。

5.2.4.6 混合三向单词查找树

简单改进一下基于三向单词查找树的查找方式：使用一个大型显式的多向根结点。实现它最简单的办法就是维护一张含有 R 棵三向单词查找树的表，每一棵都对应着键的首字母的一种可能的值。如果 R 不大，那可以使用键的头两个字母（表的大小变为 R^2）。这种方法有效的前提是键的首字母的分布必须均匀。这样得到的混合查找算法和人们在电话黄页中查找姓名的行为很相似。查找的第一步是进行多向判断（“让我们来看看，它的首字母是‘A’”），接下来可能是某种双向判断（“它在‘Andrews’之前，但在‘Aitken’之后”），然后就是一系列字符匹配（“‘Algonquin’，……没有，‘Algorithms’不在列表之中，因为没有以‘Algor’开头的单词！”）。这些程序可能是查找字符串类型的键的最快算法。 750

5.2.4.7 单向分支

和单词查找树一样，我们也可以通过将键的尾字母变为叶子结点并在内部结点中消除单向分支来提高三向单词查找树的空间利用率。

命题 L。由 N 个随机字符串构造的混合结点进行了 R' 向分支且不含有外部单向分支的三向单词查找树中，一次插入或查找操作平均需要进行约 $\ln N - t\ln R$ 次字符比较。

证明。这些粗略的估计也可以由命题 K 的证明得到。假设在查找路径上除了常数个结点（高层的几个）之外的其他所有结点均为由 R 个字符值组成的二叉查找树，因此需要将时间成本乘以 $\ln R$。

尽管将算法调优至最佳性能是一个非常大的诱惑，我们不应该忘记三向单词查找树最吸引人的特点，那就是不必担心对特定应用场景的依赖，即使是在没有调优的情况下也能提供不错的性能。 751

5.2.5 应该使用字符串符号表的哪种实现

和字符串排序一样，我们自然也想对比一下已经学习过的字符串查找方法和第 3 章中学习的通用方法。表 5.2.3 总结了已讨论过的各种算法的重要性质（二叉查找树、红黑树和散列表的条目来自第 3 章，作为比较之用）。对于特定的应用场景，这些条目有指导意义，但并非绝对的结论，因为在研究符号表实现的过程中发现许多因素（例如键的性质和混合操作的顺序）都会产生影响。

如果空间足够，R 向单词查找树的速度是最快的，能够在常数次字符比较内完成查找。对于大型字母表，R 向单词查找树所需的空间可能无法满足时，三向单词查找树是最佳的选择，因为它对“字符”比较次数是对数级别的比较，而二叉查找树中键的比较次数是对数级别的。散列表也是很有竞争力的，但如前文所述，它不支持有序性的符号表操作，也不支持扩展的字符类 API 操作，例如前

752 缀或通配符匹配。

表 5.2.3 各种字符串查找算法的性能特点

算法（数据结构）	处理由大小为 R 的字母表构造的 N 个字符串（平均长度为 w）的增长数量级		优 点
	未命中查找检查的字符数量	内存使用	
二叉树查找 (BST)	$c_1(\lg N)^2$	$64N$	适用于随机排列的键
2-3 树查找（红黑树）	$c_2(\lg N)^2$	$64N$	有性能保证
线性探测法（并行数组）	w	$32N$~$128N$	内置类型 缓存散列值
字典树查找（R 向单词查找树）	$\log_R N$	$(8R+56)N$~$(8R+56)Nw$	适用于较短的键和较小的字母表
字典树查找（三向单词查找树）	$1.39\lg N$	$64N$~$64Nw$	适用于非随机的键

答疑

问 Java 的系统排序方法使用了本节介绍的方法来查找 String 类型的键吗？

753 答 没有。

练习

5.2.1 将以下键按照顺序插入一棵 R 向空单词查找树之中并画出结果（忽略空链接）：no is th ti fo al go pe to co to th ai of th pa。

5.2.2 将以下键按照顺序插入一棵空三向单词查找树之中并画出结果（忽略空链接）：no is th ti fo al go pe to co to th ai of th pa。

5.2.3 将以下键按照顺序插入一棵 R 向空单词查找树之中并画出结果（忽略空链接）：now is the time for all good people to come to the aid of。

5.2.4 将以下键按照顺序插入一棵空三向单词查找树之中并画出结果（忽略空链接）：now is the time for all good people to come to the aid of。

5.2.5 给出非递归版本的 TrieST 和 TST。

5.2.6 对于 StringSET 数据类型，实现以下 API，如表 5.2.4 所示。

表 5.2.4 字符串集合的数据类型的 API

public class **StringSET**		
	StringSET()	创建一个字符串的集合
void	add(String key)	将 key 添加到集合中
void	delete(String key)	从集合中删除 key
boolean	contains(String key)	key 是否存在于集合中
boolean	isEmpty()	集合是否为空
int	size()	集合中的键的数量
String	toString()	对象的字符串表示

754

提高题

5.2.7 三向单词查找树中的空字符串。三向单词查找树（TST）的代码未能正确处理空字符串。说明原因并给出修正方案。

5.2.8 单词查找树的有序性操作。为 TrieST 实现 floor()、ceiling()、rank() 和 select() 方法（来自第 3 章标准有序性符号表的 API）。

5.2.9 三向单词查找树的扩展操作。为三向单词查找树实现 keys() 和本节所介绍的几种扩展操作：longestPrefixOf()、keysWithPrefix() 和 keysThatMatch()。

5.2.10 size() 方法。为 TrieST 和 TST 实现最为即时的 size() 方法（在每个结点中保存子树中的键的总数）。

5.2.11 外部单项分支。为 TrieST 和 TST 添加消除外部单向分支的代码。

5.2.12 内部单项分支。为 TrieST 和 TST 添加消除内部单向分支的代码。

5.2.13 R^2 向分支的根结点的三向单词查找树。如正文所述，为 TST 添加代码，在前两层结点中实现多向分支。

5.2.14 长度为 L 的唯一子字符串。编写一个 TST 的用例，从标准输入读取文本并计算其中长度为 L 的唯一子字符串的数量。例如，如果输入为 cgcgggcgcg，那么长度为 3 的唯一子字符串就有 5 个：cgc、cgg、gcg、ggc 和 ggg。提示：使用字符串方法 substring(i,i+L) 来提取第 i 个子字符串并将它插入到一张符号表中。

5.2.15 唯一子字符串。编写一个 TST 的用例，从标准输入读取文本并计算其中任意长度的唯一子字符串的数量。后缀树能够高效完成这个任务——请见第 6 章。

5.2.16 文档的相似性。编写一个 TST 的静态方法用例，接受一个 int 值 L 和两个文件名作为命令行参数并计算两份文档的“L- 相似性”：各个频率向量之间的欧拉距离，其中频率向量为各个长度为 3 的子字符串（trigram）的出现次数除以所有长度为 3 的子字符串的总数。给出一个静态方法 main()，接受一个 int 值 L 作为命令行参数，从标准输入中获取一系列文件名并打印出一个矩阵，以显示所有文档之间的 L- 相似性。 755

5.2.17 拼写检查。编写一个 TST 的用例 SpellChecker，从命令行接受一个英语字典文件作为参数，然后从标准输入读取一个字符串并打印所有不在字典中的单词。请使用字符串集合数据类型。

5.2.18 白名单。编写一个 TST 的用例，解决 1.1 节和 3.5 节中介绍并讨论过的（请见 3.5.2.2 节）白名单问题。

5.2.19 随机电话号码。编写一个 TrieST 的用例（R=10），从命令行接受一个 int 值 N 并打印出 N 个形如 (xxx) xxx-xxxx 的随机电话号码。使用符号表避免出现重复的号码。使用本书网站上的 AreaCodes.txt 来避免打印出不存在的区号。

5.2.20 是否含有前缀。为 StringSET 类（请见练习 5.2.6）添加一个方法 containsPrefix()，接受一个字符串 s 作为输入，如果集合中存在某个以 s 作为前缀的字符串时返回 true。

5.2.21 子字符串匹配。给定一列（短）字符串，你的任务是找到所有含有用户所寻找的字符串 s 的字符串。为此任务设计一份 API 并给出一个 TST 用例来实现这个 API。提示：将每个单词的所有后缀（例如：string, tring, ring, ing, ng, g）插入到 TST 中。

5.2.22 打字的猴子。假设有一只会打字的猴子，它打出每个字母的概率为 p，结束一个单词的概率为 1–26p。编写一个程序，计算产生各种长度的单词的概率分布。其中如果 "abc" 出现了多次，只计算一次。 756

实验题

5.2.23 *重复元素（再续）*。使用 StringSET（请见练习 5.2.6）代替 HashSET 重新完成练习 3.5.30，比较两种方法的运行时间。然后使用 dedup 为 $N=10^7$、10^8 和 10^9 运行实验，用随机 long 型字符串重复实验并讨论结果。

5.2.24 *拼写检查器*。使用本书网站上的 dictionary.txt 文件和 3.5.2.2 节中的 BlackFilter 用例重新完成练习 3.5.31 并打印出一个文本文件中所有拼错的单词。用该用例处理 war.txt 文件，比较 TrieST 和 TST 的性能并讨论结果。

5.2.25 *字典*。重新完成练习 3.5.32：在一个需要高性能的场景中研究一个类似于 LookupCSV 的用例的性能（使用 TrieST 和 TST）。确切地说，设计一个查询生成器来取代从标准输入接受命令，对大量输入和大量查询进行性能测试。

5.2.26 *索引*。重新完成练习 3.5.33：在一个需要高性能的场景中研究一个类似于 LookupIndex 的用例的性能（使用 TrieST 和 TST）。确切地说，设计一个查询生成器来取代从标准输入接受命令，对大量输入和大量查询进行性能测试。

757

5.3 子字符串查找

字符串的一种基本操作就是子字符串查找：给定一段长度为 N 的文本和一个长度为 M 的模式（pattern）字符串，在文本中找到一个和该模式相符的子字符串请见图 5.3.1。解决该问题的大部分算法都可以很容易地扩展为找出文本中所有和该模式相符的子字符串、统计该模式在文本中的出现次数、或者找出上下文（和该模式相符的子字符串周围的文字）的算法。



图 5.3.1 子字符串的查找

758

当你在文本编辑器或是浏览器中查找某个单词时，就是在查找子字符串。事实上，该问题的原始动机就是为了支持这种查找操作。字符串查找的另一个经典应用是在截获的通信内容中寻找某种重要的模式。一位军队将领感兴趣的可能是在截获的文本中寻找和“拂晓进攻”类似的字句。一名黑客感兴趣的可能是在内存中查找与“Password:”相关的内容。在今天的世界中，我们经常在互联网的海量信息中查找字符串。

为了更好地理解算法，请记住模式相对于文本是很短的（M 可能等于 100 或者 1000），而文本相对于模式是很长的（N 可能等于 100 万或者 10 亿）。在字符串查找中，一般会对模式进行预处理来支持在文本中的快速查找。

字符串查找是一个很有趣而且也很经典的问题：人们发明了几个截然不同（且令人惊讶的）算法，它们不仅产生了一系列能够实际应用的查找方法，而且也展示了许多重要的算法设计技巧。

5.3.1 历史简介

我们将要学习的几种算法有一段有趣的历史。我们在这里进行总结并帮助大家对它们的地位有一个正确的认识。

子字符串查找有一个简单而使用广泛的暴力算法。虽然它在最坏情况下的运行时间与 MN 成正比，但是在处理许多应用程序中的字符串时（除了一些变态的情况之外），它的实际运行时间一般与 $M+N$ 成正比。另外，它很好地利用了大多数计算机系统中标准的结构特性，因此即使是更加巧妙的算法也很难超越它经过优化后的版本的性能。

在 1970 年，S.Cook 在理论上证明了一个关于某种特定类型的抽象计算机的结论。这个结论暗示了一种在最坏情况下用时也只是与 $M+N$ 成正比的解决子字符串查找问题的算法。D.E.Knuth 和 V.R.Pratt 改进了 Cook 用来证明定理的框架（并非为实际应用所设计）并将它提炼为一个相对简单而实用的算法。这看起来是一个鲜有但令人满意的将理论结果（意外的）立刻转化为实际应用的例子。但实际上，J.H.Morris 在实现一个文本编辑器时，为了解决某个棘手的问题（他希望能够在文本中避免“回退”）也发明了几乎相同的算法。殊途同归的两种方式得到了同一种算法，这说明它是这个问题的一种基础的解决方案。

Knuth、Morris 和 Pratt 直到 1976 年才发表了他们的算法。在这段时间里，R.S.Boyer 和 J.S.Moore（以及 R.W.Gosper 独立地）发明了一种在许多应用程序中都非常快的算法，该算法一般只会检查文本字符串中的一部分字符。许多文本编辑器都使用了这个算法，以显著降低字符串查找的响应时间。

Knuth-Morris-Pratt 算法和 Boyer-Moore 算法都需要对模式字符串进行复杂的预处理，这个过程十分晦涩而且也限制了它们的应用范围。（事实上，有位系统程序员觉得 Morris 算法实在是太难懂了，就干脆用暴力算法代替了。）

在 1980 年，M.O.Rabin 和 R.M.Karp 使用散列开发出了一种与暴力算法几乎一样简单但运行时间与 $M+N$ 成正比的概率极高的算法。另外，它们的算法还可以扩展到二维的模式和文本中，这使得它比其他算法更适用于图像处理。

这段历史说明人们在不断地研究更好的算法。事实上大家都认为，这个经典问题还将会有很大
759 的发展。

5.3.2 暴力子字符串查找算法

子字符串查找的一个最显而易见的方法就是在文本中模式可能出现匹配的任何地方检查匹配是否存在。如左侧框注所示的 search() 方法就是在文本字符串 txt 中查找模式字符串 pat 第一次出现的位置。这段程序使用了一个指针 i 跟踪文本，一个指针 j 跟踪模式。对于每个 i，代码首先将 j 重置为 0 并不断将它增大，直至找到了一个不匹配的字符或是模式结束（j==M）为止，请见图 5.3.2。如果在模式字符串结束之前文本字符串就已经结束了（i==N-M+1），那么就没有找到匹配：模式字符串在文本中不存在。我们约定在不匹配时返回 N 的值。

```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
   for (int i = 0; i <= N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt.charAt(i+j) != pat.charAt(j))
            break;
      if (j == M) return i;  // 找到匹配
   }
   return N;                 // 未找到匹配
}
```

暴力子字符串查找

在典型的字符串处理应用程序中，索引 j 增长的机会很少，因此该算法的运行时间与 N 成正比。绝大多数比较在比较第一个字符时就会产生不匹配。例如，假设你在这一段文字之中查找 pattern 这个模式字符串。在找到模式字符串的第一次匹配之前共有 191 个单词，其中只有 7 个的首字母是 p（且没有以 pa 开头的单词）。因此字符比较的总次数为 191+7，也就是说文本中每个字符平均需要比较 1.036 次。从另一个方面来说，没人能够保证算法总是如此高效。例如，模式字符串可能以一连串的 A 开头。如果是这样且文本也包含含有一大串 A
760 的字符串，那么字符串的查找就可能会很慢。

命题 M。在最坏情况下，暴力子字符串查找算法在长度为 N 的文本中查找长度为 M 的模式需要 $\sim NM$ 次字符比较，请见图 5.3.3。

证明。一种最坏的情况是文本和模式都是一连串的 A 接一个 B。那么，对于 $N-M+1$ 个可能的匹配位置，模式中的所有字符都需要和文本比对，总成本为 $M(N-M+1)$。一般来说 M 远小于 N，因此总成本为 $\sim NM$。

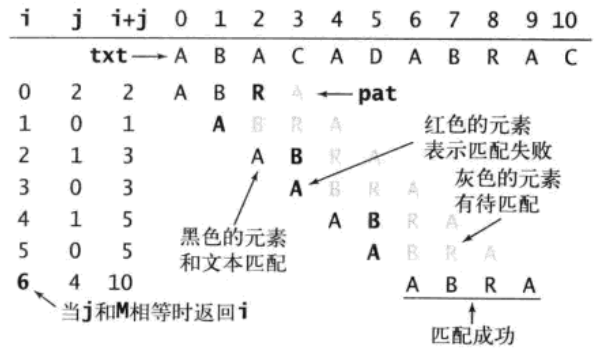


图 5.3.2 暴力子字符串查找（另见彩插）

这种奇怪的字符串不太可能出现在英文文本之中，但在其他应用场景中是完全可能的（例如二进制文本），因此我们需要更好的算法。

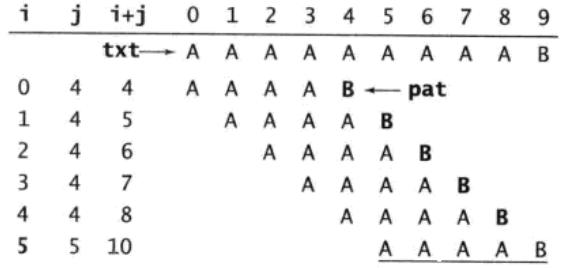


图 5.3.3 暴力子字符串查找（最坏情况）

下方框注所示的该算法的另一种实现是有指导意义的。和以前一样，程序使用了一个指针 i 跟踪文本，一个指针 j 跟踪模式。在 i 和 j 指向的字符相匹配时，代码进行的字符比较和上一个实现相同。请注意，这段代码中的 i 值相当于上一段代码中的 i+j：它指向的是文本中已经匹配过的字符序列的*末端*（i 以前指向的是这个序列的*开头*）。如果 i 和 j 指向的字符不匹配了，那么需要回退这两个指针的值：将 j 重新指向模式的开头，将 i 指向本次匹配的开始位置的下一个字符。

```
public static int search(String pat, String txt)
{
   int j, M = pat.length();
   int i, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
   {
      if (txt.charAt(i) == pat.charAt(j)) j++;
      else { i -= j; j = 0;  }
   }
   if (j == M) return i - M;  // 找到匹配
   else        return N;  // 未找到匹配
}
```

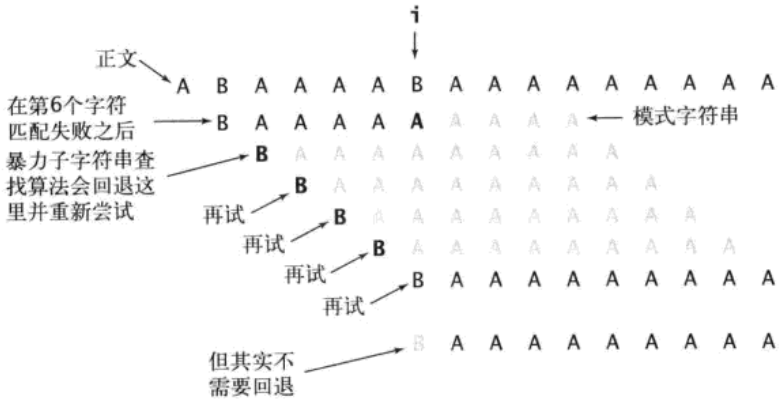
暴力子字符匹配算法的另一种实现（显式回退）

5.3.3 Knuth-Morris-Pratt 子字符串查找算法

Knuth、Morris 和 Pratt 发明的算法的基本思想是当出现不匹配时，就能知晓一部分文本的内容（因为在匹配失败之前它们已经和模式相匹配）。我们可以利用这些信息避免将指针回退到所有这些已知的字符之前。

举一个具体的例子。假设字母表中只有两个字符，查找的模式字符串为 B A A A A A A A A A 。现在，假设已经匹配了模式中的 5 个字符，第 6 个字符匹配失败。当发现不匹配的字符时，可以知道文本中的前 6 个字符肯定是 B A A A A B（前 5 个匹配，第 6 个失败），文本指针现在指向的是末尾的字符 B。你可以观察到，这里不需要回退文本指针 i，因为正文中的前 4 个字符都是 A，均与模式的第一个字符不匹配。另外，i 当前指向的字符 B 和模式的第一个字符相匹配，所以可以直接将 i 加 1，以比较文本中的下一个字符和模式中的第二个字符。这说明，对于这个模式，可以将暴力子字符串查找算法实现中的 else 语句替换为 j=1（且并不将 i 加 1）。因为循环中 i 的值并未变化，这种方法最多只会进行 N 次字符比较。这次特殊变化的实际影响仅限于这种特殊情况，但这种想法是值得思考的——Knuth-Morris-Pratt 算法正是这种情况的一般化。令人惊讶的是，在匹配失败时总是能够将 j 设为某个值以使 i 不回退，请见图 5.3.4。

在匹配失败时，如果模式字符串中的某处可以和匹配失败处的正文相匹配，那么就不应该完全跳过所有已经匹配的所有字符。例如，当在文本 A A B A A B A A A A 中查找模式 A A B A A A 时，我们首先会在模式的第 5 个字符处发现匹配失败，但是应该在第 3 个字符处继续查找，否则就会错过已经匹配的部分。KMP 算法的主要思想是提前判断如何重新开始查找，而这种判断只取决于模式本身。



762

图 5.3.4 文本字符串的指针在子字符串查找中的回退

5.3.3.1 模式指针的回退

在 KMP 子字符串查找算法中，不会回退文本指针 i，而是使用一个数组 dfa[][] 来记录匹配失败时模式指针 j 应该回退多远。对于每个字符 c，在比较了 c 和 pat.charAt(j) 之后，dfa[c][j] 表示的是应该和下个文本字符比较的模式字符的位置。在查找中，dfs[txt.charAt(i)][j] 是在比较了 txt.charAt(i) 和 pat.charAt(j) 之后应该和 txt.charAt(i+1) 比较的模式字符位置。在匹配时会继续比较下一个字符，因此 dfa[pat.charAt(j)][j] 总是 j+1。在不匹配时，不仅可以知道 txt.charAt(i) 的字符，也可以知道正文中的前 j-1 个字符，它们就是模式中的前 j-1 个字符。对于每个字符 c，你可以将这个过程想象为首先将模式字符串的一个副本覆盖在这 j 个字符

之上（模式中的前 j-1 个字符以及字符 c——需要判断的是当这些字符就是 txt.charAt(i-j+1..i) 时应该怎么办），然后从左向右滑动这个副本直到所有重叠的字符都相互匹配（或者没有相匹配的字符）时才停下来。这将指明模式字符串中可能产生匹配的下一个位置。和 txt.charAt(i+1)(dfa[txt.charAt(i)][j]) 比较的模式字符的索引正是重叠字符的数量，请见图 5.3.5。

5.3.3.2 KMP 查找算法

只要计算出了 dfa[][] 数组，就得到了后面框注所示的子字符串查找算法：当 i 和 j 所指向的字符匹配失败时（从文本的 i-j+1 处开始检查模式的匹配情况），模式可能匹配的下一个位置应该从 i-dfa[txt.charAt(i)][j] 处开始。按照算法，从该位置开始的 dfa[txt.charAt(i)][j] 个字符和模式的前 dfa[txt.charAt(i)][j] 个字符应该相同，因此无需回退指针 i，只需要将 j 设为 dfa[txt.charAt(i)][j] 并将 i 加 1 即可，这正是当 i 和 j 所指向的字符匹配时的行为。

5.3.3.3 DFA 模拟

说明这个过程的一种较好的方法是使用确定有限状态自动机（DFA）。事实上，由它的名字你也可以看出，dfa[][] 数组定义的正是一个确定有限状态自动机。图 5.3.6 显示确定有限状态自动机是由状态（数字标记的圆圈）和转换（带标签的箭头）组成的。模式中的每个字符都对应着一个状态，每个此类状态能够转换为字母表中的任意字符。对于子字符串查找问题，在我们所考虑的 DFA 中，这些转换中只有一条是匹配转换（从 j 到 j+1，标签为 pat.charAt(j)），其他的都是非匹配转换（指向左侧）。所有状态都和字符的比较相对应，每个状态都表示一个模式字符串

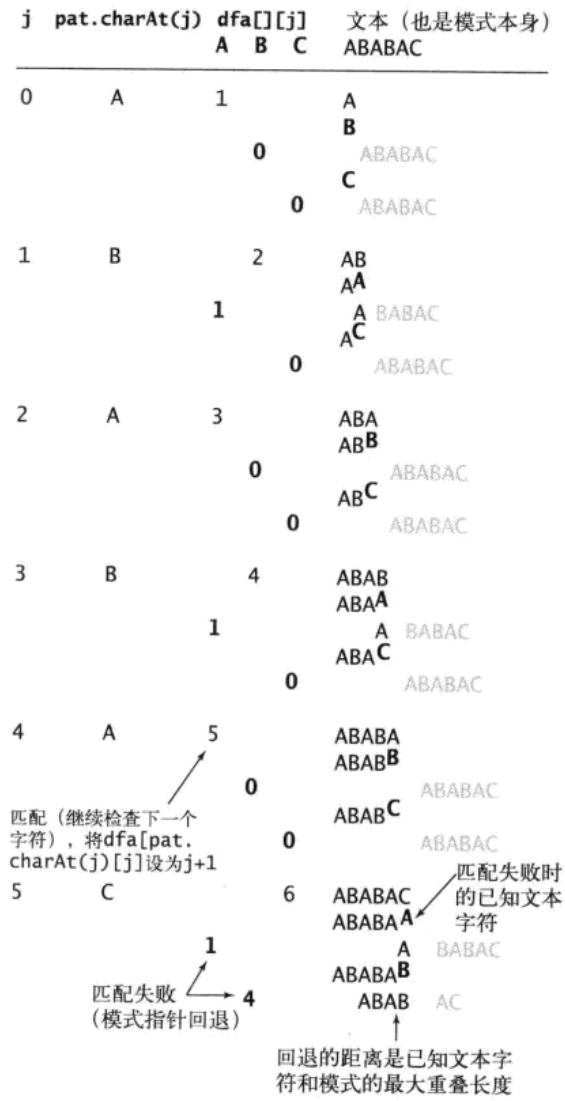


图 5.3.5 KMP 子字符串查找算法在处理 A B A B A C 时模式指针的回退

```
public int search(String txt)
{  // 模拟DFA处理文本txt时的操作
   int i, j, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
      j = dfa[txt.charAt(i)][j];
   if (j == M) return i - M;  // 找到匹配
   else        return N;      // 未找到匹配
}
```

KMP子字符串查找算法（DFA模拟）

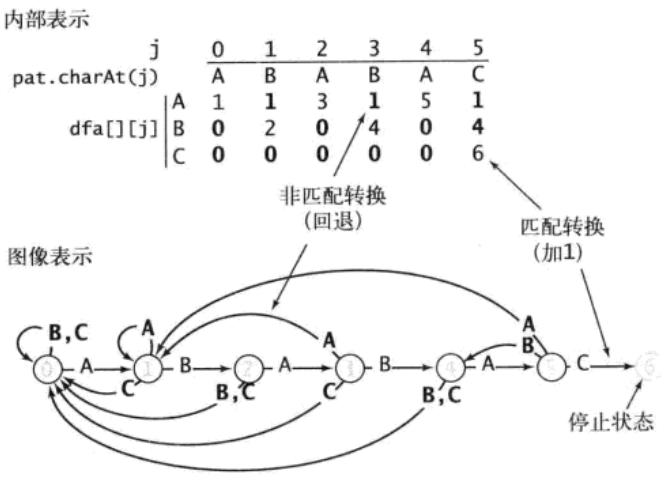


图 5.3.6 和模式字符串 A B A B A C 对应的确定有限状态自动机

的索引值。当我们在标记为 j 的状态中检查文本中的第 i 个字符时，自动机的行为是这样的："沿着转换 dfa[txt.charAt(i)][j] 前进并继续检查下一个字符（将 i 加 1）。"对于一个匹配的转换，就向右移动一位，因为 dfa[txt.charAt(i)][j] 的值总是 j+1；对于一个非匹配转换，就在向左移动。自动机每次从左向右从文本中读取一个字符并移动到一个新的状态。我们还包含了一个不会进行任何转换的停止状态 M。自动机从状态 0 开始：如果自动机到达了状态 M，那么就在文本中找到了和模式相匹配的一段子字符串（我们称这种情况为确定有限状态自动机识别了该模式）；如果自动机在文本结束时都未能到达状态 M，那么就可以知道文本中不存在匹配该模式的子字符串。每个模式字符串都对应着一个自动机（由保存了所有转换的 dfa[][] 数组表示）。KMP 的字符串查找方法 search() 只是一段模拟自动机运行的 Java 程序。

763
~
764

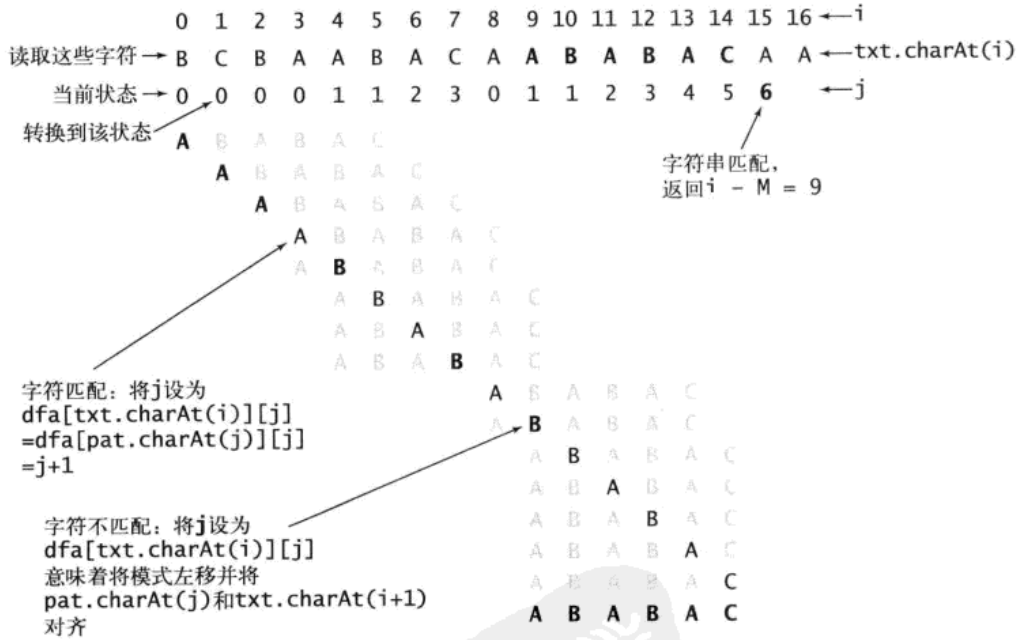


图 5.3.7 KMP 子字符串查找算法处理 A B A B A C 时的轨迹（DFA 模拟）

要体验在 DFA 中的子字符串查找操作，你可以先想象一下它所完成的两件最简单的任务。在查找过程的开始，从文本的开头进行查找，起始状态为 0。它停留在 0 状态并扫描文本，直到找到

一个和模式的首字母相同的字符。这时它移动到下一个状态并开始运行。在这个过程的最后，当它找到一个匹配时，它会不断地匹配模式中的字符与文本，自动机的状态会不断前进直到状态 M。图 5.3.7 所示的轨迹给出了 DFA 运行的一个典型例子。每次匹配都会将 DFA 带向下一个状态（等价于增大模式字符串的指针 j）；每次匹配失败都会使 DFA 回到较早前的状态（等价于将模式字符串的指针 j 变为一个较小的值）。正文指针 i 是从左向右前进的，一次一个字符，但索引 j 会在 DFA 的指导下在模式字符串中左右移动。

5.3.3.4 构造 DFA

现在你应该已经明白了 DFA 的原理，接下来解决 KMP 算法的关键问题：如何计算给定模式相对应的 dfa[][] 数组？意外的是，这个问题的答案仍然是 DFA 本身！ Knuth、Morris 和 Pratt 发明了这种巧妙（但也相当复杂）的构造方式。当在 pat.charAt(j) 处匹配失败时，希望了解的是，如果回退了文本指针并在右移一位之后重新扫描已知的文本字符，DFA 的状态会是什么？我们其实并不想回退，只是想将 DFA 重置到适当的状态，就好像已经回退过文本指针一样。 765

这里的关键在于需要重新扫描的文本字符正是 pat.charAt(1) 到 pat.charAt(j-1) 之间，忽略了首字母是因为模式需要右移一位，忽略了最后一个字符是因为匹配失败。这些模式中的字符都是已知的，因此对于每个可能匹配失败的位置都可以预先找到重启 DFA 的正确状态。图 5.3.8 显示了示例中的各种可能性。请务必理解这个概念。

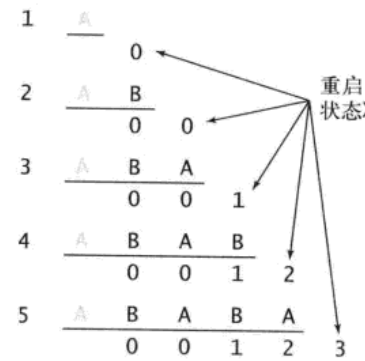


图 5.3.8 计算模式 A B A B A C 的重启状态的 DFA 模拟

DFA 应该如何处理下一个字符？和回退时的处理方式相同，除非在 pat.charAt(j) 处匹配成功，这时 DFA 应该前进到状态 j+1。例如，对于 A B A B A C，要判断在 j=5 时匹配失败后 DFA 应该怎么做。通过 DFA 可以知道完全回退之后算法会扫描 B A B A 并达到状态 3，因此可以将 dfa[][3] 复制到 dfa[][5] 并将 C 所对应的元素的值设为 6，因为 pat.charAt(5) 是 C（匹配）。因为在计算 DFA 的第 j 个状态时只需要知道 DFA 是如何处理前 j-1 个字符的，所以总能从尚不完整的 DFA 中得到所需的信息。

计算中最后一个关键细节是，你可以观察到在处理 dfa[][] 的第 j 列时维护重启位置 X 很容易。因为 X<j，所以可以由已经构造的 DFA 部分来完成这个任务——X 的下一个值是 dfa[pat.charAt(j)][X]。继续上一段中的例子，将 X 的值更新为 dfa['C'][3]=0（但我们不会使用这个值，因为 DFA 的构造已经完成了）。

由以上的讨论可以得到右侧框注这段短小精悍的代码来构造给定模式的 DFA。对于每个 j，它将会：

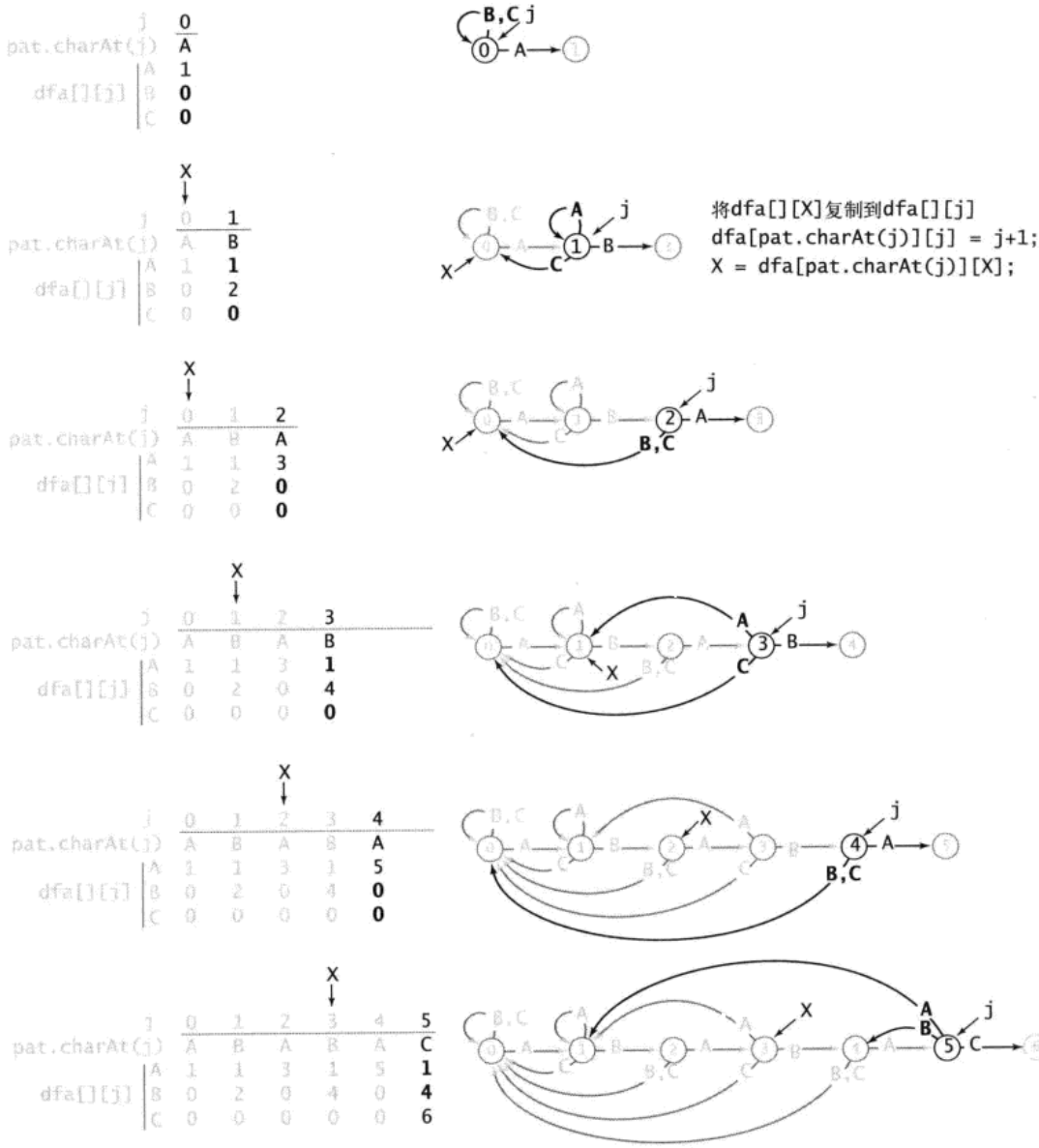
- 将 dfa[][X] 复制到 dfa[][j]（对于匹配失败的情况）；
- 将 dfa[pat.charAt(j)][j] 设为 j+1（对于匹配成功的情况）；
- 更新 X。

```
dfa[pat.charAt(0)][0] = 1;
for (int X = 0, j = 1; j < M; j++)
{  // 计算dfa[][j]
   for (int c = 0; c < R; c++)
      dfa[c][j] = dfa[c][X];
   dfa[pat.charAt(j)][j] = j+1;

   X = dfa[pat.charAt(j)][X];
}
```

KMP子字符串查找算法中DFA的构造 766

图 5.3.9 显示了这段代码处理样例输入的轨迹。为了确保你能完全理解它，请完成练习 5.3.2 和练习 5.3.3。



767 图 5.3.9 KMP 子字符串查找算法中模式 A B A B A C 的 DFA 的构造

算法 5.6 Knuth-Morris-Pratt 字符串查找算法

```
public class KMP
{
   private String pat;
   private int[][] dfa;
   public KMP(String pat)
   {  // 由模式字符串构造DFA
```

```
        this.pat = pat;
        int M = pat.length();
        int R = 256;
        dfa = new int[R][M];
        dfa[pat.charAt(0)][0] = 1;
        for (int X = 0, j = 1; j < M; j++)
        {  // 计算dfa[][j]
           for (int c = 0; c < R; c++)
              dfa[c][j] = dfa[c][X];              // 复制匹配失败情况下的值
           dfa[pat.charAt(j)][j] = j+1;           // 设置匹配成功情况下的值
           X = dfa[pat.charAt(j)][X];             // 更新重启状态
        }
     }
     public int search(String txt)
     {  // 在txt上模拟DFA的运行
        int i, j, N = txt.length(), M = pat.length();
        for (i = 0, j = 0; i < N && j < M; i++)
           j = dfa[txt.charAt(i)][j];
        if (j == M) return i - M;  // 找到匹配（到达模式字符串的结尾）
        else        return N;      // 未找到匹配（到达文本字符串的结尾）
     }
     public static void main(String[] args)
     // 请见表5.3.1
  }
```

```
% java KMP AACAA AABRAACADABRAACAADABRA
text:    AABRAACADABRAACAADABRA
pattern:             AACAA
```

该 Knuth-Morris-Pratt 子字符串查找算法的实现的构造函数根据模式字符串构造了一个确定有限状态自动机，使用 search() 方法在给定文本字符串中查找模式字符串。它和暴力子字符串查找算法的功能相同，但带适合查找自我重复性的模式字符串。

算法 5.6 实现了表 5.3.1 所示的 API。

768

表 5.3.1 子字符串查找的 API

public class KMP	
KMP(String pat)	根据模式字符串 pat 创建一个 DFA
int search(String txt)	在 txt 中找到 pat 的出现位置

你可以在下页框注中看到 KMP 的一个典型的测试用例。KMP 的构造函数会根据模式字符串创建一个 DFA 并用 search() 方法中在给定的文本中查找该模式字符串。

命题 N。对于长度为 M 的模式字符串和长度为 N 的文本，Knuth-Morris-Pratt 字符串查找算法访问的字符不会超过 $M+N$ 个。

证明。由代码可以马上得到，在计算 dfa[][] 时，算法会访问模式字符串中的每个字符一次，在 search() 方法中会访问文本中的每个字符（最坏情况下）一次。

我们还需要引入另一个参数，即字母表的大小 R，所以构造 DFA 所需的总时间（和空间）将与 MR 成正比。如果在构造 DFA 时为每个状态设置一个匹配转换和一个非匹配转换（而非指向每个可能出现的字符的多个转换），那么也可以去掉参数 R，但构造过程会更加复杂一些。

```
public static void main(String[] args)
{
   String pat = args[0];
   String txt = args[1];
   KMP kmp = new KMP(pat);
   StdOut.println("text:    " + txt);
   int offset = kmp.search(txt);
   StdOut.print("pattern: ");
   for (int i = 0; i < offset; i++)
      StdOut.print(" ");
   StdOut.println(pat);
}
```

KMP子字符串查找算法的测试用例

769

KMP 算法为最坏情况提供的线性级别运行时间保证是一个重要的理论成果。在实际应用中，它比暴力算法的速度优势并不十分明显，因为极少有应用程序需要在重复性很高的文本中查找重复性很高的模式。但该方法的一个优点是不需要在输入中回退。这使得 KMP 子字符串查找算法更适合在长度不确定的输入流（例如标准输入）中进行查找，需要回退的算法在这种情况下则需要复杂的缓冲机制。但其实当回退很容易时，还可以比 KMP 快得多。下面，我们来学习一种利用回退来获取巨大性能收益的算法。

5.3.4 Boyer-Moore 字符串查找算法

当可以在文本字符串中回退时，如果可以从*右向左*扫描模式字符串并将它和文本匹配，那么就能得到一种非常快的字符串查找算法。例如，在查找子字符串 B A A B B A A时，如果匹配了第七个和第六个字符，但在第 5 个字符处匹配失败，那马上就可以将模式向右移动 7 个位置并继续检查文本中的第 14 个字符。这是因为部分匹配找到了 X A A 而 X 不是 B，而这 3 个连续的字符在模式中是唯一的。一般来说，模式的结尾部分也可能出现在文本的其他位置，因此和 Knuth-Morris-Pratt 算法一样，也需要一个记录重启位置的数组。本节不会再次详细介绍它的构造方法，因为它和 Knuth-Morris-Pratt 算法中的实现很相似。这里将讨论 Boyer 和 Moore 给出的另一种从右向左扫描模式字符串的更有效的方法。

和 KMP 子字符串查找算法的实现一样，我们会根据匹配失败时文本和模式中的字符来决定下一步的行动。而预处理步骤的目的在于判断对于文本中可能出现的每一个字符，在匹配失败时算法应该怎么办。将这个想法变为现实就可以得到一种高效实用的子字符串查找算法。

5.3.4.1 启发式的处理不匹配的字符

请看图 5.3.10，它显示了在文本 F I N D I N A H A Y S T A C K N E E D L E 中查找模式 N E E D L E 的过程。因为是从右向左与模式进行匹配，所以首先会比较模式字符串中的 E 和文本中的 N（位置为 5 的字符）。因为 N 也出现在了模式字符串中，所以将模式字符串向右移动 5 个位置，将文本中的字符 N 和模式字符串中（最左侧）的 N 对齐。然后比较模式字符串最右侧的 E 和文本中的 S（位置在第 10 个字符），匹配失败。但因为 S *不包含在*模式字符串中，所以可以将模式字符串向右移动 6 个位置。此时模式字符串最右侧的 E 和文本中位置为 16 的 E 相匹配，但我们发现文本的下一个（位置为 15 的）字符为 N，匹配再次失败。于是和第一次一样，将模式字符串再次向右移动 5 个位置。最后，从位置 20 处开始从右向左扫描，发现文本中含有与模式匹配的子字符串。这种方法找到匹配位置仅用了 4 次字符比较（以及 6 次比较来验证匹配）！

```
i   j   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
  文本→ F I N D I N A H A Y S  T  A  C  K  N  E  E  D  L  E
0   5     N E E D L E ←模式字符串
5   5           N E E D L E
11  4                       N  E  E  D  L  E
15  0                          N  E  E  D  L  E
返回i=15
```

图 5.3.10 从右向左的（Boyer-Moore）子字符串查找中的启发式地处理不匹配的字符

770

5.3.4.2 起点

要实现启发式的处理不匹配的字符，我们使用数组 right[] 记录字母表中的每个字符在模式中出现的最靠右的地方（如果字符在模式中不存在则表示为 –1）。这个值揭示了如果该字符出现在文本中且在查找时造成了一次匹配失败，应该向右跳跃多远。要将 right[] 数组初始化，首先将所有元素的值设为 –1，然后对于 0 到 M–1 的 j，将 right[pat.charAt(j)] 设为 j，如图 5.3.11 对模式 N E E D L E 的处理所示。

		N	E	E	D	L	E	
c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	**3**	3	3	3
E	-1	-1	**1**	**2**	2	2	**5**	5
...								-1
L	-1	-1	-1	1	-1	**4**	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	**0**	0	0	0	0	0	0
...								-1

图 5.3.11 Boyer-Moore 算法中的跳跃表的计算

5.3.4.3 子字符串的查找

在计算完 right[] 数组之后，算法 5.7 的实现就很简单了。我们用一个索引 i 在文本中从左向右移动，用另一个索引 j 在模式中从右向左移动。内循环会检查正文和模式字符串在位置 i 是否一致。如果从 M–1 到 0 的所有 j，txt.charAt(i+j) 都和 pat.charAt(j) 相等，那么就找到了一个匹配。否则匹配失败，就会遇到以下三种情况。

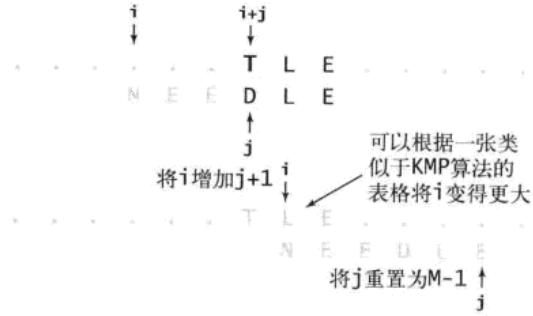


图 5.3.12 启发式的处理不匹配的字符（不匹配的字符不包含在模式字符串中）

- 如果造成匹配失败的字符不包含在模式字符串中，将模式字符串向右移动 j+1 个位置（即将 i 增加 j+1）。小于这个偏移量只可能使该字符与模式中的某个字符重叠。事实上，这次移动也会将模式字符串前面一部分已知的字符和模式结尾的一部分已知字符对齐。通过预先计算一张类似于 KMP 算法的表格，还可以将 i 值变得更大（请见图 5.3.12）。
- 如果造成匹配失败的字符包含在模式字符串中，那就可以使用 right[] 数组来将模式字符串和文本对齐，使得该字符和它在模式字符串中出现的最右位置相匹配。和刚才一样，小于这个偏移量只可能使该字符和模式中的与它无法匹配的字符（比它出现的最右位置更靠右的字符）重叠。我们可以用一张类似于 KMP 算法的表格将 i 变得更大，如图 5.3.13 所示。

771

- 如果这种方式无法增大 i，那就直接将 i 加 1 来保证模式字符串至少向右移动了一个位置。图 5.3.13 下方的例子说明了这种情况。

算法 5.7 简明地实现了这个过程。请注意，使用 –1 表示 right[] 数组中相应字符不包含在模式字符串中，这个约定能够将前两种情况合并（将 i 增大 j-right[txt.charAt(i+j)]）。

完整的 Boyer-Moore 算法预计算了模式字符串与自身的不匹配情况（和 KMP 算法的方式类似[①]）并为最坏情况提供了线性级别的运行时间保证（而算法 5.7 在最坏情况下的运行时间与 NM 成正比——请见练习 5.3.19）。我们在这里省略了算法的计算，因为在一般的应用程序中对不匹配字符的启发式处理已经可以控制算法的性能。

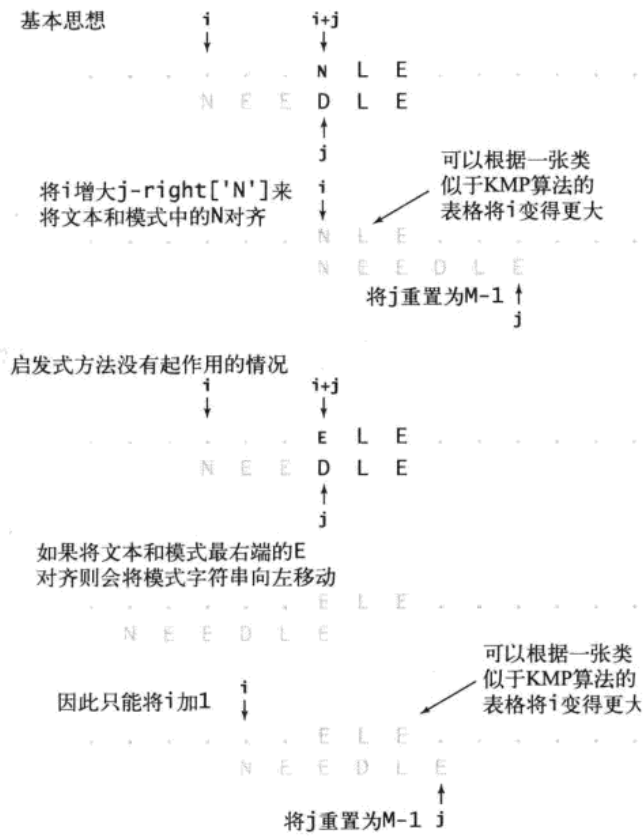


图 5.3.13 启发式的处理不匹配的字符（不匹配的字符包含在模式字符串中）

命题 O。在一般情况下，对于长度为 N 的文本和长度为 M 的模式字符串，使用了 Boyer-Moore 的子字符串查找算法通过启发式处理不匹配的字符需要 ~N/M 次字符比较。

讨论。我们可以用各种随机字符串模型证明该结论，但这些模型一般都不太可能在实际情况中出现，因此这里省略了证明的细节。在许多实际应用场景中，模式字符串中仅含有字母表中的若干字符是很常见的，因此几乎所有的比较都会使算法跳过 M 个字符，这样就得到了以上结论。

算法 5.7 Boyer-Moore 字符串匹配算法（启发式地处理不匹配的字符）

```
public class BoyerMoore
{
   private int[] right;
   private String pat;
   BoyerMoore(String pat)
   {  // 计算跳跃表
      this.pat = pat;
      int M = pat.length();
      int R = 256;
      right = new int[R];
```

① 即跳跃表。——译者注

```
      for (int c = 0; c < R; c++)
         right[c] = -1;                  // 不包含在模式字符串中的字符的值为-1
      for (int j = 0; j < M; j++)        // 包含在模式字符串中的字符的值为
         right[pat.charAt(j)] = j;       // 它在其中出现的最右位置
   }

   public int search(String txt)
   {  // 在txt中查找模式字符串
      int N = txt.length();
      int M = pat.length();
      int skip;
      for (int i = 0; i <= N-M; i += skip)
      {  // 模式字符串和文本在位置i匹配吗?
         skip = 0;
         for (int j = M-1; j >= 0; j--)
            if (pat.charAt(j) != txt.charAt(i+j))
            {
               skip = j - right[txt.charAt(i+j)];
               if (skip < 1) skip = 1;
               break;
            }
         if (skip == 0) return i;          // 找到匹配
      }
      return N;                            // 未找到匹配
   }

   public static void main(String[] args)  // 请见表5.3.1
}
```

这段子字符串查找算法的实现的构造函数根据模式字符串构造了一张每个字符在模式中出现的最右位置的表格。查找算法会从右向左扫描模式字符串，并在匹配失败时通过跳跃将文本中的字符和它在模式字符串中出现的最右位置对齐。

772 ~ 773

5.3.5 Rabin-Karp 指纹字符串查找算法

M.O.Rabin 和 R.A.Karp 发明了一种完全不同的基于散列的字符串查找算法。我们需要计算模式字符串的散列函数，然后用相同的散列函数计算文本中所有可能的 M 个字符的子字符串散列值并寻找匹配。如果找到了一个散列值和模式字符串相同的子字符串，那么再继续验证两者是否匹配。这个过程等价于将模式保存在一张散列表中，然后在文本的所有子字符串中进行查找。但不需要为散列表预留任何空间，因为它只会含有一个元素。根据这段描述直接实现的算法将会比暴力子字符串查找算法慢很多（因为计算散列值将会涉及字符串中的每个字符，成本比直接比较这些字符要高得多）。Rabin 和 Karp 发明了一种能够在常数时间内算出 M 个字符的子字符串散列值的方法（需要预处理），这样就得到了在实际应用中的运行时间为线性级别的字符串查找算法。

5.3.5.1 基本思想

长度为 M 的字符串对应着一个 R 进制的 M 位数。为了用一张大小为 Q 的散列表来保存这种类型的键，需要一个能够将 R 进制的 M 位数转化为一个 0 到 Q-1 之间的 int 值散列函数。除留余数法（请见 3.4 节）是一个很好的选择：将该数除以 Q 并取余。在实际应用中会使用一个随机的素数 Q，在不溢出的情况下选择一个尽可能大的值。（因为我们并不会真的需要一张散列表。）理解这个方法最简单的办法就是取一个较小的 Q 和 R=10 的情况，如下所示。要在文本 3 1 4 1 5 9 2 6 5 3 5 8

9 7 9 3中找到模式2 6 5 3 5，首先要选择散列表的大小Q（在这个例子中是997），则散列值为26535 % 997 = 613，然后计算文本中所有长度为5个数字的子字符串的散列值并寻找匹配。在这个例子中，在找到613的匹配之前，得到的散列值分别为508、201、715、971、442和929，请见图5.3.14。

```
          pat.charAt(j)
j     0  1  2  3  4
      2  6  5  3  5  % 997 = 613

                          txt.charAt(i)
i     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
      3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
0     3  1  4  1  5  % 997 = 508
1        1  4  1  5  9  % 997 = 201
2           4  1  5  9  2  % 997 = 715
3              1  5  9  2  6  % 997 = 971
4                 5  9  2  6  5  % 997 = 442
5                    9  2  6  5  3  % 997 = 929   匹配
6 ←返回i=6              2  6  5  3  5  % 997 = 613
```

图5.3.14 Rabin-Karp字符串查找算法的基本思想

5.3.5.2 计算散列函数

对于5位的数值，只需使用int值即可完成所有所需的计算。但如果M是100或者1000怎么办？这里使用的是Horner方法，它和3.4节中见过的用于字符串和其他多值类型的键的计算方法非常相似，代码如下面框注所示。这段代码计算了用char值数组表示的R进制的M位数的散列函数，所需时间与M成正比。（将M作为参数传递给该方法，这样就可以将它同时用于模式字符串和正文。）对于这个数中的每一位数字，将散列值乘以R，加上这个数字，除以Q并取其余数。例如，这样计算示例模式字符串散列值的过程如图5.3.15所示。我们也可以用同样的方法计算文本中的子字符串散列值，但这样一来字符串查找算法的成本就将是对文本中的每个字符进行乘法、加法和取余计算的成本之和。在最坏情况下这需要NM次操作，相对于暴力子字符串查找算法来说并没有任何改进。

```
private long hash(String key, int M)
{  // 计算key[0..M-1]的散列值
   long h = 0;
   for (int j = 0; j < M; j++)
      h = (R * h + key.charAt(j)) % Q;
   return h;
}
```

Horner方法，用于除留余数法计算散列值

```
          pat.charAt(j)
i     0  1  2  3  4
      2  6  5  3  5
0     2  % 997 = 2          R      Q
1     2  6  % 997 = (2*10 + 6) % 997 = 26
2     2  6  5  % 997 = (26*10 + 5) % 997 = 265
3     2  6  5  3  % 997 = (265*10 + 3) % 997 = 659
4     2  6  5  3  5  % 997 = (659*10 + 5) % 997 = 613
```

图5.3.15 使用Horner方法计算模式字符串的散列值

5.3.5.3 关键思想

Rabin-Karp 算法的基础是对于所有位置 i，高效计算文本中 i+1 位置的子字符串散列值。这可以由一个简单的数学公式得到。我们用 t_i 表示 txt.charAt(i)，那么文本 txt 中起始于位置 i 的含有 M 个字符的子字符串所对应的数即为：

$$x_i=t_iR^{M-1}+t_{i+1}R^{M-2}+\cdots+t_{i+M-1}R^0$$

假设已知 $h(x_i)=x_i \bmod Q$。将模式字符串右移一位即等价于将 x_i 替换为：

$$x_{i+1}=(x_i-t_iR^{M-1})R+t_{i+M}$$

即将它减去第一个数字的值，乘以 R，再加上最后一个数字的值。现在，关键的一点在于不需要保存这些数的值，而只需要保存它们除以 Q 之后的余数。取余操作的一个基本性质是如果在每次算术操作之后都将结果除以 Q 并取余，这等价于在完成了所有算术操作之后再将最后的结果除以 Q 并取余。曾经在用 Horner 方法（请见 3.1.1.4 节）实现除留余数法时利用过这个性质。这么做的结果就是无论 M 是 5、100 还是 1000，都可以在常数时间内高效地不断向右一格一格地移动。

774
↓
775

5.3.5.4 实现

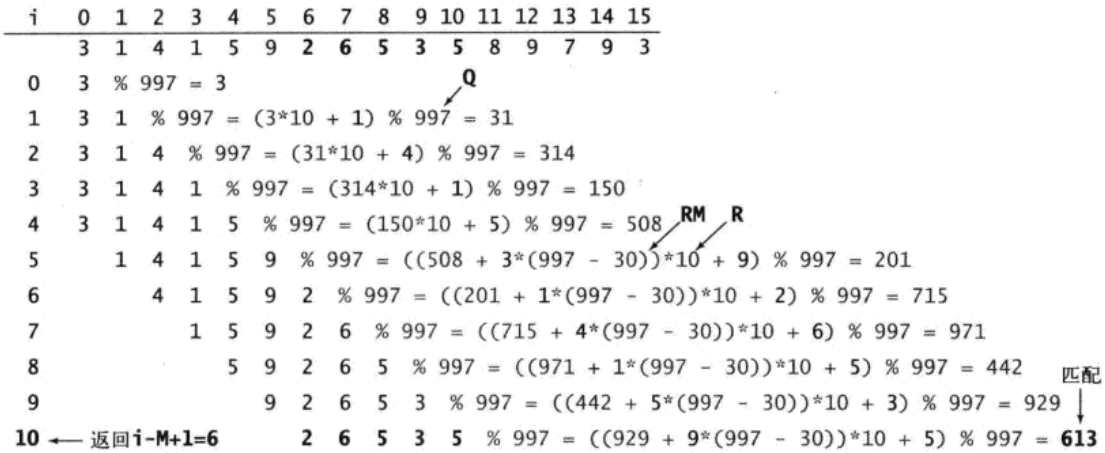
根据以上讨论可以立即得到算法 5.8 中对该子字符串查找算法的实现。构造函数为模式字符串计算了散列值 patHash 并在变量 RM 中保存了 $R^{M-1} \bmod Q$ 的值。hashSearch() 方法开头计算了文本的前 M 个字母的散列值并将它和模式字符串的散列值进行比较。如果未能匹配，它将会在文本中继续前进，用以上讨论的方法计算由位置 i 开始的 M 个字符的散列值，将它保存在 txtHash 变量中并将每个新的散列值和 patHash 进行比较，请见图 5.3.16 和图 5.3.17。（在 txtHash 的计算中，额外加上了一个 Q 来保证所有的数均为正，这样取余操作才能够得到预期的结果。）



图 5.3.16 Rabin-Karp 字符串查找算法中的关键计算（在文本中右移一位）

5.3.5.5 小技巧：用蒙特卡洛法验证正确性

在文本 txt 中找到散列值与模式字符串相匹配的一个 M 个字符的子字符串之后，你可能会逐个比较它们的字符以确保得到了一个匹配而非相同的散列值。我们不会这么做，因为这需要回退文本指针。作为替代，这里将散列表的“规模” Q 设为任意大的一个值，因为我们并不会真构造一张散列表而只是希望用模式字符串验证是否会产生冲突。我们会取一个大于 10^{20} 的 long 型值，使得一个随机键的散列值与模式字符串冲突的概率小于 10^{-20}。这是一个极小的值。如果它还不够小，你可以将这种方法运行两遍，这样失败的几率将会小于 10^{-40}。这是蒙特卡洛算法一种著名早期应用，它既能够保证运行时间，失败的概率又非常小。检查匹配的其他方法可能很慢（性能有很小的概率相当于暴力算法）但能够确保正确性。这种算法被称为拉斯维加斯算法。



776

图 5.3.17 Rabin-Karp 子字符串查找算法举例

算法 5.8 Rabin-Karp 指纹字符串查找算法

```
public class RabinKarp
{
   private String pat;        // 模式字符串（仅拉斯维加斯算法需要）
   private long patHash;      // 模式字符串的散列值
   private int M;             // 模式字符串的长度
   private long Q;            // 一个很大的素数
   private int R = 256;       // 字母表的大小
   private long RM;           // R^(M-1) % Q

   public RabinKarp(String pat)
   {
      this.pat = pat;         // 保存模式字符串（仅拉斯维加斯算法需要）
      this.M = pat.length();
      Q = longRandomPrime();           // 请见练习5.3.33
      RM = 1;
      for (int i = 1; i <= M-1; i++)   // 计算R^(M-1) % Q
         RM = (R * RM) % Q;            //   用于减去第一个数字时的计算
      patHash = hash(pat, M);
   }
   public boolean check(int i)  // 蒙特卡洛算法（请见正文）
   {  return true;  }  //   对于拉斯维加斯算法，检查模式与txt(i..i-M+1)的匹配
   private long hash(String key, int M)
   // 请见正文
   private int search(String txt)
   {  // 在文本中查找相等的散列值
      int N = txt.length();
      long txtHash = hash(txt, M);
      if (patHash == txtHash&&ckeck(0)) return 0;   // 一开始就匹配成功
      for (int i = M; i < N; i++)
      {  // 减去第一个数字，加上最后一个数字，再次检查匹配
```

```
            txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
            txtHash = (txtHash*R + txt.charAt(i)) % Q;
            if (patHash == txtHash)
              if (check(i - M + 1)) return i - M + 1; // 找到匹配

        }
        return N;                                    // 未找到匹配
    }
}
```

该字符串查找算法的基础是散列。它在构造函数中计算了模式字符串的散列值并在文本中查找该散列值的匹配。 777

命题 P。 使用蒙特卡洛算法的 Rabin-Karp 子字符串查找算法的运行时间是线性级别的且出错的概率极小。使用拉斯维加斯算法的 Rabin-Karp 子字符串查找算法能够保证正确性且性能极其接近线性级别。

讨论。 因为我们不需要实际创建一张散列表，使用非常大的 Q 几乎不可能发生散列值冲突。Rabin 和 Karp 证明了只要选择了适当的 Q 值，随机字符串产生散列碰撞的概率为 1/Q。这意味着对于这些变量实际可能出现的值，字符串不匹配时散列值也不会匹配，散列值匹配时字符串才会匹配。理论上来说，文本中的某个子字符串可能会在与模式不匹配的情况下产生散列冲突，但在实际应用中使用该算法寻找匹配是可靠的。

如果你对概率论（或者我们使用的随机字符串模型以及生成随机数字的代码）并不是很有信心，那么可以在 `check()` 方法中添加检查文本子字符串和模式是否匹配的代码。这将把算法 5.8 变成拉斯维加斯版本（请见练习 5.3.12）。如果你再添加一个方法来检查这段代码是否真正被执行过，随着时间的推移你就会逐渐相信概率论的证明了。

Rabin-Karp 字符串查找算法也称为*指纹字符串查找算法*，因为它只用了极少量信息就表示了（可能非常大的）模式字符串并在文本中寻找它的指纹（散列值）。算法的高效性来自于对指纹的高效计算和比较。 778

5.3.6 总结

表 5.3.2 总结了我们已经讨论过的各种子字符串查找算法。尽管常常出现多个算法都能完成相同的任务的情况，但它们都各有特点：暴力查找算法的实现非常简单且在一般的情况下都工作良好；（Java 的 `String` 类型的 `indexOf()` 方法使用的就是暴力子字符串查找算法。）Knuth-Morris-Pratt 算法能够保证线性级别的性能且不需要在正文中回退；Boyer-Moore 算法的性能在一般情况下都是亚线性级别（可能是线性级别的 M 倍）；Rabin-Karp 算法是线性级别。每种算法也各有缺点：暴力查找算法所需的时间可能和 MN 成正比；Knuth-Morris-Pratt 算法和 Boyer-Moore 算法都需要额外的内存空间；Rabin-Karp 算法的内循环很长（若干次算术运算，而其他算法都只需要比较字符）。这些特点都总结在了表 5.3.2 中。

（续）

表 5.3.2 各种字符串查找算法的实现的成本总结

算 法	版 本	操作次数		在文本中回退	正确性	额外的空间需求
		最坏情况	一般情况			
暴力算法	—	MN	$1.1N$	是	是	1
Knuth-Morris-Pratt 算法	完整的 DFA（算法 5.6）	$2N$	$1.1N$	否	是	MR
	仅构造不匹配的状态转换	$3N$	$1.1N$	否	是	M
	完整版本	$3N$	N/M	是	是	R
Boyer-Moore 算法	启发式的查找不匹配的字符（算法 5.7）	MN	N/M	是	是	R
Rabin-Karp 算法*	蒙特卡洛算法（算法 5.8）	$7N$	$7N$	否	是*	1
779	拉斯维加斯算法	$7N$*	$7N$	是	是	1

* 概率保证，需要使用均匀和独立的散列函数。

答疑

问 子字符串查找问题看起来并没有什么实际用处，我们真的需要理解这些复杂的算法吗？

答 这个……Boyer-Moore 算法能够将速度提高 M 倍，在实际应用当中还是相当强大的。另外，能够处理流输入（无需回退）的性质也给 KMP 算法和 Rabin-Karp 算法带来了许多应用。除了这些直接的实际应用之外，这些算法也为我们介绍了抽象自动机和随机性在算法设计领域的应用。

问 为什么不能通过将所有字符都转换为二进制数并处理二进制的文本来简化问题呢？

780 答 这种方法并没有什么效果，因为字符的边界处可能产生错误的匹配。

练习

5.3.1 使用算法 5.6 相同的 API，开发一个暴力子字符串查找算法的实现 `Brute`。

5.3.2 在 Knuth-Morris-Pratt 算法中，给出模式 `A A A A A A A A A A` 的 `dfa[][]` 数组，按照正文中的样式画出 DFA。

5.3.3 在 Knuth-Morris-Pratt 算法中，给出模式 `A B R A C A D A B R A` 的 `dfa[][]` 数组，按照正文中的样式画出 DFA。

5.3.4 编写一个方法，接受一个字符串 `txt` 和一个整数 `M` 作为参数，返回字符串中 `M` 个连续的空格第一次出现的位置，如果不存在则返回 `txt.length`。估计你的方法在一般的文本中和在最坏情况下所需的字符比较次数。

5.3.5 开发一个暴力子字符串查找算法的实现 `BruteForceRL`，从右向左匹配模式字符串（算法 5.7 的简化版本）。

5.3.6 给出算法 5.7 的构造函数计算模式 `A B R A C A D A B R A` 所得到的 `right[]` 数组。

5.3.7 为暴力子字符串查找算法的实现添加一个 `count()` 方法，统计模式字符串在文本中的出现次数，再添加一个 `searchAll()` 方法来打印出所有出现的位置。

5.3.8 为 KMP 类添加一个 `count()` 方法来统计模式字符串的在文本中的出现次数，再添加一个 `searchAll()` 方法来打印出所有出现的位置。

5.3.9 为 BoyerMoore 类添加一个 count() 方法来统计模式字符串的在文本中的出现次数，再添加一个 searchAll() 方法来打印出所有出现的位置。

5.3.10 为 RabinKarp 类添加一个 count() 方法来统计模式字符串的在文本中的出现次数，再添加一个 searchAll() 方法来打印出所有出现的位置。

5.3.11 为算法 5.7 实现的 Boyer-Moore 算法构造一个最坏情况下的输入（说明它的运行时间不是线性级别的）。

5.3.12 为 RabinKarp 类（算法 5.8）的 check() 方法中添加代码，将它变为使用拉斯维加斯算法的版本（检查给定位置的文本和模式字符串是否匹配）。

5.3.13 在算法 5.7 实现的 Boyer-Moore 算法中，证明当 c 为模式字符串中的最后一个字符时，能够将 right[c] 设为 c 在模式字符串中的倒数第二次出现的位置。 781

5.3.14 使用 char[] 代替 String 来表示文本和模式字符串，给出本节中的各种子字符串查找算法的实现。

5.3.15 设计一个从右向左扫描模式字符串的暴力子字符串查找算法。

5.3.16 按照正文中轨迹的样式显示暴力子字符串查找算法在处理以下模式和文本时的轨迹。

a. 模式：AAAAAAAB 文本：AAAAAAAAAAAAAAAAAAAAAAAAB

b. 模式：ABABABAB 文本：ABABABABAABABABABAAAAAAAA

5.3.17 为以下模式字符串画出 KMP 算法的 DFA。

a. AAAAAAB

b. AACAAAB

c. ABABABAB

d. ABAABAAABAAAB

e. ABAABCABAABCB

5.3.18 假设模式字符串和文本都是由大小为 R（不小于 2）的字母表随机生成的字符串。证明暴力算法预期的字符比较次数为 $(N-M+1)(1-R^{-M})/(1-R^{-1}) \leq 2(N-M+1)$。

5.3.19 构造一个使 Boyer-Moore 算法（仅使用对不匹配字符的启发式查找）性能低下的样例输入。

5.3.20 如何修改 Rabin-Karp 算法才能够判定 k 个模式（假设它们的长度全部相同）中的任意子集出现在文本之中？

解答：计算所有 k 个模式字符串的散列值并将散列值保存在一个 StringSET（请见练习 5.2.6）对象中。

5.3.21 如何修改 Rabin-Karp 算法来查找中间字符为“通配符”（能够匹配任意字符的符号）的模式字符串？ 782

5.3.22 如何修改 Rabin-Karp 算法来在 $N \times N$ 的文本中查找一个 $H \times V$ 的模式？

5.3.23 编写一个程序，一次读入字符串中的一个字符并立即判断当前字符串是否为回文。提示：使用 Rabin-Karp 的散列思想。 783

提高题

5.3.24 找出所有子字符串。为我们学习过的 4 种字符串查找算法添加一个 findAll() 方法，返回一个 Iterable<Integer> 对象使得用例能够遍历文本中模式字符串出现的所有位置。

5.3.25 流输入。为 KMP 类添加一个 search() 方法，接受一个 In 类型的变量作为参数，在不使用其他任何实例变量的条件下在指定的输入流中查找模式字符串。为 RabinKarp 类也添加一个类似的方法。

5.3.26 回环变位。编写一个程序，对于给定的两个字符串，检查它们是否互为对方的回环变位。例如 example 和 ampleex。

5.3.27 串联重复查找。在字符串 s 中，基础字符串 b 的串联重复就是连续将 b 至少重复两遍（无重叠）的一个子字符串。开发并实现一个线性时间的子字符串查找算法，接受给定的字符串 b 和 s，返回 s 中 b 的最长串联重复的起始位置。例如，当 b 为 "abcd" 而 s 为 "abcabcababcababcababcab" 时，你的程序应该返回 3。

5.3.28 暴力子字符串查找算法中的缓冲区。向你为练习 5.3.1 给出的解答中添加一个 search() 方法，接受一个（In 类型的）输入流作为参数并在给定的输入流查找模式字符串。注意：你需要维护一个至少能够保存输入流的前 M 个字符的缓冲区。面临的挑战是要编写高效的代码为任意输入流初始化、更新和清理缓冲区。

5.3.29 Boyer-Moore 算法中的缓冲区。为算法 5.7 添加一个 search() 方法，接受一个（In 类型的）输入流作为参数并在给定的输入流中查找模式字符串。

5.3.30 二维查找。实现另一个版本的 Rabin-Karp 算法，在二维文本中查找模式，假设模式和文本都是由字符组成的矩形。

5.3.31 随机模式。在一段给定的文本中查找一个长度为 100 的随机模式字符串需要多少次字符比较？

答：一次也不用。以下方法就可以有效的完成这个任务：

```
public boolean search(char[] txt)
{  return false; }
```

因为一个长度为 100 的随机模式字符串出现在任何文本中的概率之低足以让我们认为它是 0。

784 5.3.32 唯一的子字符串。使用 Rabin-Karp 算法的思想完成练习 5.2.14。

5.3.33 随机素数。为 RabinKarp 类（算法 5.8）实现 longRandomPrime() 方法。提示：随机的 n 位数字是素数的概率与 $1/n$ 成正比。

5.3.34 直线型代码。[①] Java 的虚拟机（以及计算机上的汇编语言）支持一种 goto 指令，它使我们能够将查找“嵌入”到机器代码中，如下方的程序所示（这段程序等价于在 KMP 算法中用 KMPdfa 数组模拟模式的 DFA 的运行，但效率要高的多）。为了避免在每次增大 i 时检查是否已经到达文本的结尾，假设文本的最后 M 个字符就是模式字符串本身。

```
    int i = -1;
sm: i++;
s0: if (txt[i]) != 'A' goto sm;
s1: if (txt[i]) != 'A' goto s0;
s2: if (txt[i]) != 'B' goto s0;
s3: if (txt[i]) != 'A' goto s2;
s4: if (txt[i]) != 'A' goto s0;
s5: if (txt[i]) != 'A' goto s3;
    return i-8;
```

处理模式字符串A A B A A A 的直线型代码

在这段代码中 goto 的标签与 dfa[] 数组完全一一对应。编写一个静态方法，接受一个模式作为参数，产生一段类似的直线型代码来查找给定的模式。

5.3.35 二进制字符串中的 Boyer-Moore 算法。启发式处理不匹配的字符对于二进制字符串并没有什么作用，因为匹配失败的可能字符只有两种（而且它们都非常可能出现在模式字符串中）。编写一个适用于二进制字符串的子字符串查找类，它应该能够将多个位组合成可以被算法 5.7 处理的“字符”。注意：如果你每次都取 b 位，那么需要一个含有 2^b 个元素的 right[] 数组。b 的值不能太大，以保证 right[] 数组不会太大；也不能太小，以使文本中大多数 b 位字符不太可能出现在模式中——模式中含有 M–b+1 种不同的 b 位字符（从第 1 到第 M–b+1 位的每个位置上各有一个），

① 译法参考《代码大全》，第二版第14章。——译者注

因此 $M-b+1$ 远小于 2^b。例如，如果你选择的 b 使得 2^b 约等于 $\lg(4M)$，那么 `right[]` 数组中超过四分之三的元素的值都将是 -1。但不要让 b 小于 $M/2$，否则当模式字符串横跨两个 b 位字符时你完全可能会漏掉它。 785

实验题

5.3.36 随机文本。编写一个程序，接受整型参数 `M` 和 `N`，生成一个长度为 `N` 的随机二进制文本字符串，计算该字符串的最后 `M` 位在整个字符串中的出现次数。注意：不同的 `M` 值适用的方法可能不同。

5.3.37 随机文本的 KMP 算法。编写一个用例，接受整型参数 `M`、`N` 和 `T` 并运行以下实验 `T` 遍：随机生成一个长度为 `M` 的模式字符串和一段长度为 `N` 的文本，记录使用 `KMP` 算法在文本中查找该模式时比较字符的次数。修改 `KMP` 类的实现来记录比较次数并打印出重复 `T` 次之后的平均比较次数。

5.3.38 随机文本的 Boyer-Moore 算法。对于 Boyer-Moore 算法完成上一道练习。

5.3.39 运行时间。编写一段程序，用本节学习的 4 种算法在《双城记》（tale.txt）中查找以下字符串并记录时间：

```
it is a far far better thing that i do than i have ever done
```

讨论你的结果在何种程度上验证了正文对这几种算法的性能猜想。 786

5.4 正则表达式

在许多应用程序中，我们在查找子字符串时并没有被查找模式的完整信息。文本编辑器的用户可能希望仅指定模式的一部分，或是指定某种能够匹配若干个不同单词的模式，或是指定几种可以任意匹配的不同模式。例如，生物学家可能希望在基因组序列中寻找满足特定条件的基因。本节中，我们将会学习如何高效地完成这种类型的模式匹配。

5.3 节中的算法完全依赖指定完整的模式字符串，因此需要寻找不同的方法。本节将会学习的一些基本工具能够构造一个非常强大的字符串查找程序，它能够在长度为 N 的文本中匹配长度为 M 的复杂模式。在最坏情况下，它所需的时间和 MN 成正比，而在一般的应用程序中还会快得多。

首先，我们需要一种描述模式的方法，即一种严谨的说明上述“部分子字符串的查找问题”的方式。这份说明必须含有一些比 5.3 节中使用的“检查文本字符串的第 i 个字符和模式字符串的第 j 个字符是否匹配”更加强大的原始操作。为此，我们使用*正则表达式*。它能够用自然、简单而强大的 3 种操作组合来描述模式。

程序员使用正则表达式的历史已经有数十年了。随着网络搜索的爆炸性增长，它们的使用变得更加广泛。本节开始会讨论几个应用程序。这不仅是为了让你感受它的用途和功能，也是为了让你对它的基本性质更加熟悉。

和 5.3 节中的 KMP 算法一样，本节也将使用一种能够在文本中查找模式的抽象自动机来描述这 3 种基本的操作。模式匹配算法同样会构造一个这样的自动机并模拟它的运行。当然，这种模式匹配自动机比 KMP 算法的 DFA 更加复杂，但不会超出你的想象。

你将会看到，我们为模式匹配问题给出的解答和计算机科学中最基础的问题有着紧密的联系。例如，我们在程序中用于完成给定模式下的字符串查找任务的算法和 Java 系统中用来将 Java 程序转化为计算机上的机器语言的算法很相似。我们还会遇到*非确定性*这个概念。它在人们对高效算法的追求中起到了关键的作用（请见第 6 章）。

787 ~ 788

5.4.1 使用正则表达式描述模式

我们的重点是模式的描述，它由 3 种基本操作和作为操作数的字符组成。这里，我们用*语言*指代一个字符串的集合（可能是无限的），用*模式*指代一种语言的详细说明。我们将要学习的规则和大家都很熟悉的算术表达式中的规则十分类似。

5.4.1.1 连接操作

第一种基本操作就是 5.3 节中使用过的连接操作。当我们写出 AB 时，就指定了一种语言 {AB}。它含有一个由两个字符组成的字符串，由 A 和 B 连接而成。

5.4.1.2 或操作

第二种基本操作可以在模式中指定多种可能的匹配。如果我们在两种选择之间指定了一个*或*运算符，那么它们都将属于同一种语言。我们用竖线符号“|”表示这个操作。例如，A|B 指定的语言是 {A,B}，A|E|I|O|U 指定的语言是 {A,E,I,O,U}。连接操作的优先级高于*或*操作，因此 AB|BCD 指定的语言是 {AB,BCD}。

5.4.1.3 闭包操作

第三种基本操作可以将模式的部分重复任意的次数。模式的*闭包*是由将模式和自身连接任意多次（包括零次）而得到的所有字符串所组成的语言。我们将“*”标记在需要被重复的模式之后，以表示闭包。闭包操作的优先级高于连接操作，因此 AB* 指定的语言由一个 A 和 0 个或多个 B 的字

符串组成，而 A*B 指定的语言由 0 个或多个 A 和一个 B 的字符串组成。空字符串的记号是 ϵ，它存在于所有文本字符串之中（包括 A*）。

5.4.1.4 括号

我们使用括号来改变默认的优先级顺序。例如，C(AC|B)D 指定的语言是 {CACD,CBD}，(A|C)((B|C)D) 指定的语言是 {ABD,CBD,ACD,CCD}，(AB)* 指定的语言是由将 AB 连接任意多次得到的所有字符串和空字符串组成的 {ϵ,AB,ABAB,...}

这些简单的例子已经可以写出虽然复杂但却清晰而完整的描述某种语言的正则表达式了（示例请见表 5.4.1）。某些语言可能可以用其他方式简单表述，但找到这些简单的方法可能会比较困难。例如，表格的最后一行中的正则表达式指定的就是 (A|B)* 的一个只含有偶数个 B 的子集。

表 5.4.1 正则表达式举例

正则表达式	匹配的字符串	不匹配的字符串
(A|B)(C|D)	AC AD BC BD	其他所有字符串
A(B|C)*D	AD ABD ACD ABCCBD	BCD ADD ABCBC
A*|(A*BA*BA*)*	AAA BBAABB BABAAA	ABA BBB BABBAAA

789

正则表达式都是非常简单的形式语言对象，甚至比你在小学里学到的算术表达式更简单。我们将会利用它的简洁性开发小巧而高效的算法来处理它们。首先给出如下正式定义。

定义。一个正则表达式可以是：

- 空字符串 ϵ；
- 单个字符；
- 包含在括号中的另一个正则表达式；
- 两个或多个连接起来的正则表达式；
- 由或运算符分隔的两个或多个正则表达式；
- 由闭包运算符标记的一个正则表达式。

这段定义描述了正则表达式的语法，说明了怎样才是一个合法的正则表达式。在本节中对给定正则表达式的非形式化的描述是它的语义。作为复习，我们要继续在形式定义中对它们进行总结。

定义（续）。每个正则表达式表示的都是一个字符串的集合，它们的定义如下所述。

- 空正则表达式表示的字符串的集合为空，含有 0 个元素。
- 一个字符表示的字符串的集合含有一个元素，即该字符本身。
- 一个由括号和包含在其中的正则表达式组成的正则表达式表示的字符串的集合与括号内的正则表达式相同。
- 由两个正则表达式连接起来的正则表达式表示的字符串的集合为这两个正则表达式分别表示的字符串集合的叉乘。（按照正则表达式中指定的顺序，由一个字符串集合中的元素和另一个字符串集合中的元素相连接所能够组合而成的所有字符串。）
- 由或运算符连接的两个正则表达式所表示的字符串的集合为两个正则表达式所分别表示的字符串集合的并集。

❑ 由一个正则表达式的闭包所表示的字符串的集合由∈（空字符串）或将被修饰的正则表达式所表示的字符串集合重复任意次所得到的所有字符串所组成。

一般来说，给定正则表达式所描述的语言可能非常庞大，甚至是无限的。描述一种语言可以有许多中不同的方法，我们必须尝试给出最简洁的模式，就像在不断地尝试写出简洁的程序和实现高效的算法一样。

790

5.4.2 缩略写法

一般的应用程序都在基本规则的基础上增加了各种额外的规则，以力求简洁地描述实际应用中所需要的语言。从理论角度来看，它们都只是涉及多个操作数的一系列操作的缩略写法；从实际角度来看，它们是对基本操作的实用扩展，以便能够写出小巧的模式。

5.4.2.1 字符集描述符

只用一个或几个字符来直接表示一个字符集时常能够带来方便。点“.”是一个能够表示任意字符的通配符。包含在方括号中的一系列字符表示这些字符中的任意一个。这一系列字符可以由一个范围来表示。如果开头字符为“^”，这个方括号表示的就是任意非该括号内的字符。这些记法都是一系列或操作的简写，请见表5.4.2。

表5.4.2 字符集描述符

名 称	记 法	举 例
通配符	.	A.B
指定的集合	包含在[]中的字符	[AEIOU]*
范围集合	包含在[]中，由“-”分隔	[A-Z] [0-9]
补集	包含在[]中，首字母为“^”	[^AEIOU]*

5.4.2.2 闭包的简写

闭包运算符表示将它的操作数复制任意多次。在实际应用中，我们希望能够灵活指定重复的次数，或者是次数的范围。我们用“+”（加号）表示至少复制一次，“?”（问号）表示重复0次或1次，用写在“{}”（花括号）内的数或者范围来指定重复的次数。和刚才一样，这些记法也是一系列基本的连接、或和闭包操作的简写，请见表5.4.3。

表5.4.3 闭包的简写（指定操作数的重复次数）

选 项	记 法	举 例	原始写法	语言中的字符串	不在语言中的字符串
至少重复1次	+	(AB)+	(AB)(AB)*	AB ABABAB	∈ BBBAAA
重复0或1次	?	(AB)?	∈|AB	∈ AB	所有其他字符串
重复指定次数	由{}指定次数	(AB){3}	(AB)(AB)(AB)	ABABAB	所有其他字符串
重复指定范围的次数	由{}指定范围	(AB){1-2}	(AB)|(AB)(AB)	AB ABAB	所有其他字符串

5.4.2.3 转义序列

某些字符，例如“\”、“.”、“|”、“*”、“(”和“)”，都是用来构造正则表达式的元字符。我们使用以反斜杠开头的转义序列来将元字符和字母表中的字符区别开来。一个转义序列可以是一个“\”加上单个元字符（这就表示这个字符本身）。例如，“\\”表示的就是“\”。其他转义

序列表示了特殊字符和空白字符。例如，“\t”表示一个制表符，“\n”表示一个换行符，“\s”表示任意空白字符。 791

5.4.3 正则表达式的实际应用

实际应用已经证明了正则表达式善于描述与语言有关的内容。因此，正则表达式使用广泛，这方面的研究也比较深入。为了让你能在熟悉正则表达式的同时向你展示一些它的用途，在讨论正则表达式的模式匹配算法之前先给出一些实际应用的例子。正则表达式在计算机科学理论中也起到了重要的作用。在本书中完整说明它的应用范围不切实际，但会在适当的地方提到相关的理论成果。

5.4.3.1 子字符串查找

我们的总体目标是开发一种算法，能够判定给定子字符串是否包含在给定正则表达式所描述的字符串集合之中。如果文本包含在模式所描述的语言之中，就称文本和模式*相匹配*。正则表达式的模式匹配一般化了 5.3 节中的子字符串查找问题。准确地说，要在一段文本 `txt` 中查找一个子字符串 `pat`，就是检查 `txt` 是否存在于模式“`.*pat.*`”所描述的语言之中。

5.4.3.2 合法性检查

在使用互联网时，你常常会遇到正则表达式。当你在某个商业网站上输入一个日期或是账号时，输入处理程序会检查输入的格式是否正确。进行这类检查的一种方式是用代码检查所有可能出现的情况：如果你应该输入一个金额（美元），代码就会检查第一个字符是否是“$”，而且“$”之后的字符是否是一组数字，等等。更好的办法是定义一个正则表达式来描述所有合法的输入。之后，检查用户的输入是否合法就完全是模式匹配问题了：输入是否包含在正则表达式所描述的语言之中吗？随着这种检查的广泛应用，使用正则表达式进行常见检查的库在互联网上已经随处可见，请见表 5.4.4。一般来说，相比一个能够检查所有情况的程序，正则表达式是对所有有效字符串的集合更加准确和精炼的表达。

表 5.4.4 正则表达式的典型应用（简化版本）

应用场景	正则表达式	匹　配
字符串查找	`.*NEEDLE.*`	`A HAYSTACK NEEDLE IN`
电话号码	`\([0-9]{3}\)\ [0-9]{3}-[0-9]{4}`	`(800) 867-5309`
Java 标识符	`[$_A-Za-z][$_A-Za-z0-9]*`	`Pattern_Matcher`
基因组	`gcg(cgg|agg)*ctg`	`gcgaggaggcggcggctg`
电子邮件地址	`[a-z]+@([a-z]+\.)+(edu|com)`	`rs@cs.princeton.edu`

792

5.4.3.3 程序员的工具箱

正则表达式模式匹配的起源是 Unix 的命令 `grep`，它会打印出和给定正则表达式匹配的所有输入行。这个工具是数代程序员的无价之宝，而正则表达式也已经被内置于许多现代编程系统之中，从 `awk` 和 `emacs`，到 Perl、Python 和 Javascript。例如，某个目录中含有许多 .java 文件，而你希望知道哪些文件使用了 `StdIn`。这条命令可以很快给出答案：

```
% grep StdIn *.java
```

它会打印出每个文件中与“`*.StdIn.*`”匹配的每一行代码。

5.4.3.4 基因组

生物学家也会使用正则表达式来研究重要的科学问题。例如，人类的基因序列的某个区域可以用正则表达式 `gcg(cgg)*ctg` 描述，其中模式 `cgg` 的重复次数在不同的个体之间有很大区别。人们

已知某种能够造成智力障碍和其他一些症状的基因疾病和该模式的高重复次数有关。

5.4.3.5 搜索

互联网搜索引擎都支持正则表达式，但可能不是非常完整。一般来说，如果你希望通过“|”指定其他的匹配模式或者通过“*”产生重复，它都能做到。

5.4.3.6 正则表达式的可能性

理论计算机科学的第一堂入门课程就是找出正则表达式所能够指定的语言集合。例如，你可能会感到意外的是，正则表达式能够实现取余操作：例如 (0 | 1(01*0)*1)* 描述的所有由 0 和 1 组成的字符串都是 3 的倍数的二进制表示！也就是说，11、110、1001 和 1100 都在这个语言之中，而 10、1011 和 10000 都不在。

5.4.3.7 局限

并不是所有的语言都可以用正则表达式定义。一个令人深思的示例就是不存在能够描述所有合法正则表达式字符串的集合的正则表达式。这个示例的简单版本包括无法使用正则表达式检查括号是否匹配完整以及检查字符串中的 A 和 B 的数量是否一样多。

这些例子都只是冰山一角。正则表达式是计算性基础设施中非常实用的一部分，对于帮助我们理解计算的本质起到了重要的作用。和 KMP 算法一样，下面将要描述的算法也是在探索这个理论过程中的副产品。

793

5.4.4 非确定有限状态自动机

我们可以将 Knuth-Morris-Pratt 算法看作一台由模式字符串构造的能够扫描文本的有限状态自动机。对于正则表达式，我们要将这个思想推而广之。

KMP 的有限状态自动机会根据文本中的字符改变自身的状态。当且仅当自动机达到停止状态时它才找到了一个匹配。算法本身就是模拟这种自动机，这种自动机的运行很容易模拟的原因是因为它是*确定性的*：每种状态的转换都完全由文本中的字符所决定。

要处理正则表达式，就需要一种更加强大的抽象自动机。因为或操作的存在，自动机无法仅根据一个字符就判断出模式是否出现；事实上，因为闭包的存在，自动机甚至无法知道需要检查多少字符才会出现匹配失败。为了克服这些困难，我们需要*非确定性*的自动机：当面对匹配模式的多种可能时，自动机能够“猜出”正确的转换！你也许会认为这种能力是不可能的，但你会看到，编写一个程序来构造*非确定有限状态自动机*（NFA）并有效模拟它的运行是很简单的。正则表达式模式匹配程序的总体结构和 KMP 算法的总体结构几乎相同：

- 构造和给定正则表达式相对应的非确定有限状态自动机；
- 模拟 NFA 在给定文本上的运行轨迹。

Kleene *定理*是理论计算机科学中的一个重要结论，它证明了对于任意正则表达式都存在一个与之对应的非确定有限状态自动机（反之亦然）。我们会学习该定理的证明并演示如何将任意正则表达式转变为一台非确定有限状态自动机，然后模拟 NFA 的运行轨迹来完成模式匹配任务。

在学习如何构造模式匹配的 NFA 之前，先来看一个示例，它说明了 NFA 的性质和操作。请看图 5.4.1，它所显示的 NFA 是用来判断一段文本是否包含在正则表达式 ((A*B|AC)D) 所描述的语言之中。如这个示例所示，我们所定义的 NFA 有着以下特点。

- 长度为 M 的正则表达式中的每个字符在所对应的 NFA 中都有且只有一个对应的状态。NFA 的起始状态为 0 并含有一个（虚拟的）接受状态 M。
- 字母表中的字符所对应的状态都有一条从它指出的边，这条边指向模式中的下一个字符所对

应的状态（图中的黑色的边）。

- 元字符“(”、“)”、“|”和“*”所对应的状态至少含有一条指出的边（图中的红色的边），这些边可能指向其他的任意状态。
- 有些状态有多条指出的边，但一个状态只能有一条指出的黑色边。

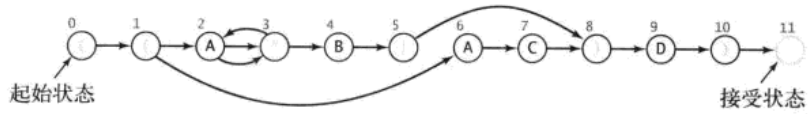


图 5.4.1 模式 ((A*B|AC)D) 所对应的 NFA（另见彩插）

我们约定将所有的模式都包含在括号中，因此 NFA 中的第一个状态对应的是左括号，而最后一个状态对应的是右括号（并能够转换为接受状态）。

和 5.3 节中的 DFA 一样，在 NFA 中也是从状态 0 开始读取文本中的第一个字符。NFA 在状态的转换中有时会从文本中读取字符，从左向右一次一个。但它和 DFA 有着一些基本的不同：

- 在图中，字符对应的是结点而不是边；
- NFA 只有在读取了文本中的所有字符之后才能识别它，而 DFA 并不一定需要读取文本中的全部内容就能够识别一个模式。

这些不同并不是关键——我们选择的是最适合研究的算法的自动机版本。

现在的重点是检查文本和模式是否匹配——为了达到这个目标，自动机需要读取所有文本并到达它的接受状态。在 NFA 中从一个状态转移到另一个状态的规则也与 DFA 不同——在 NFA 中状态的转换有以下两种方式，请见图 5.4.2。

- 如果当前状态和字母表中的一个字符相对应且文本中的当前字符和该字符相匹配，自动机可以扫过文本中的该字符并（由黑色的边）转换到下一个状态。我们将这种转换称为*匹配转换*。
- 自动机可以通过红色的边转换到另一个状态而不扫描文本中的任何字符。我们将这种转换称为 ϵ- *转换*，也就是说它所对应的“匹配”是一个空字符串 ϵ。

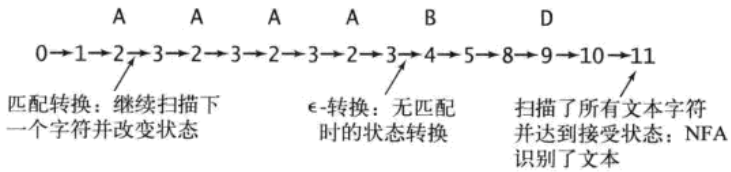


图 5.4.2 找到与 ((A*B | AC)D)NFA 相匹配的模式（另见彩插）

794 ~ 795

例如，假设输入为 A A A A B D 并启动正则表达式 ((A*B|AC)D) 所对应的自动机（起始状态为 0）。图 5.4.2 显示的一系列状态转换最终到达了接受状态。这一系列的转换说明输入文本是属于正则表达式所描述的字符串的集合之中的——即文本和模式相匹配。按照 NFA 方式，我们称该 NFA 识别了这段文本。

图 5.4.3 的例子说明了即使对于类似于 A A A A B D 这种 NFA 本应该能够识别的输入文本，也可以找到一个使 NFA 停滞的状态转换序列。例如，如果 NFA 选择在扫描完所有 A 之前就转换到状态 4，它就无法再继续前进了，因为离开状态 4 的唯一办法是匹配 B。这两个例子说明了这种自动机的不确定性。在扫描了一个 A 并到达状态 3 之后，NFA 面临着两个选择：它可以转换到状态 4，或者回到状态 2。这次选择或者会使它最终达到接受状态（如第一个例子所示）或者进入停滞（如第二个例子所示）。NFA 在状态 1 时也需要进行选择（是否由 ϵ- 转换到达状态 2 或者状态 6）。

这个例子说明了 NFA 和 DFA 之间的关键区别：因为在 NFA 中离开一个状态的转换可能有多种，因此从这种状态可能进行的转换是不确定的——即使不扫描任何字符，它在不同的时间所进行的状态转换也可能是不同的。要使这种自动机的运行有意义，所设想的 NFA 必须能够猜测对于给定的文本进行哪种转换（如果有的话）才能最终到达接受状态。换句话说，当且仅当一个 NFA 从状态 0 开始从头读取了一段文本中的所有字符，进行了一系列状态转换并最终到达了接受状态时，则称该 NFA 识别了一个文本字符串。相反，当且仅当对于一个 NFA 没有任何匹配转换和 ϵ- 转换的序列能够扫描所有文本字符并到达接受状态时，则称该 NFA 无法识别这段文本字符串。

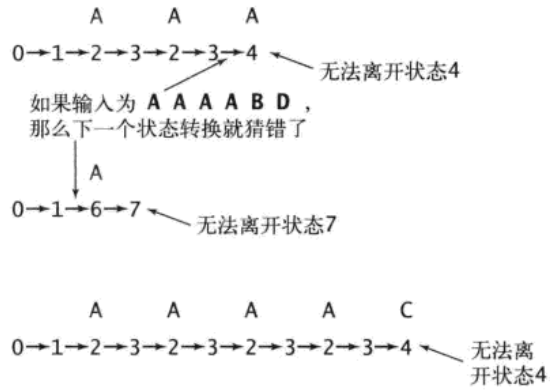


图 5.4.3 使得 ((A*B|AC)D) 的 NFA 进入停滞的状态转换序列

和 DFA 一样，这里列出所有状态的转换即可跟踪 NFA 处理文本字符串的轨迹。任意类似的结束于最终状态的转换序列都能证明某个自动机识别了某个字符串（也可能有其他的证明）。但对于一段给定的文本，应该如何找到这样一个序列呢？对于另一段给定的文本我们应该如何证明不存在这样一个序列呢？这些问题的答案比你想象的要简单，即系统地尝试所有的可能性！

796

5.4.5 模拟 NFA 的运行

存在能够猜测到达接受状态所需的状态转换自动机的设想就好像能够写出解决任意问题的程序一样：这看起来很荒谬。经过仔细思考，你会发现这个任务从概念上来说并不困难：我们可以检查所有可能的状态转换序列，只要存在能够到达接受状态的序列，我们就会找到它。

5.4.5.1 自动机的表示

首先，需要能够表示 NFA。选择很简单：正则表达式本身已经给出了所有状态名（0 到 M 之间的整数，其中 M 为正则表达式的长度）。用 char 数组 re[] 保存正则表达式本身，这个数组也表示了匹配的转换（如果 re[i] 存在于字母表中，那么就存在一个从 i 到 i+1 的匹配转换）。ϵ- 转换最自然的表示方法当然是有向图——它们都是连接 0 到 M 之间的各个顶点的有向边（图 5.4.4 中的红色边）。因此，我们用有向图 G 表示所有 ϵ- 转换。在讨论模拟的过程之后将讨论由给定正则表达式构建有向图的任务。对于上面的例子，它的有向图含有以下 9 条边：

0→1 1→2 1→6 2→3 3→2 3→4 5→8 8→9 10→11

5.4.5.2 NFA 的模拟与可达性

为了模拟 NFA 的运行轨迹，我们会记录自动机在检查当前输入字符时可能遇到的所有状态的集合。这里，关键的计算是我们已经熟悉并在算法 4.4 中解决的多点可达性问题。我们会查找所有从状态 0 通过 ϵ- 转换可达的状态来初始化这个集合。对于集合中的每个状态，检查它是否可能与第一个输入字符相匹配。检查并匹配之后就得到了 NFA 在匹配第一个字符之后可能到达的状态的集合。这里还需要向该集合中加入所有从该集合中的任意状态通过 ϵ- 转换可以到达的其他状态。有了这个匹配了第一个字符之后可能到达的所有状态的集合，ϵ- 转换有向图中的多点可达性问题的答案就

是可能匹配第二个输入字符的状态集合。例如，在示例 NFA 中初始状态集合为 {0,1,2,3,4,6}，如果第一个输入字符为 A，那么 NFA 通过匹配转换可能到达的状态是 {3,7}，然后它可能进行 3 到 2 或 3 到 4 的 ∈- 转换，因此可能与第二个字符匹配的状态集合为 {2,3,4,7}。重复这个过程直到文本结束可能得到两种结果：

- 可能到达的状态集合中含有接受状态；
- 可能到达的状态集合中不含有接受状态。

第一种结果说明存在某种转换序列使 NFA 到达接受状态。第二种结果说明对于该输入 NFA 总是会停滞，导致匹配失败。使用我们已经实现了的 SET 数据类型和用于在有向图中解决多点可达性问题的 DirectedDFS 类，下面的 NFA 模拟代码只是翻译了刚才的描述。你可以用图 5.4.4 检查你对这段代码的理解，它显示了样例输入的完整轨迹。 797

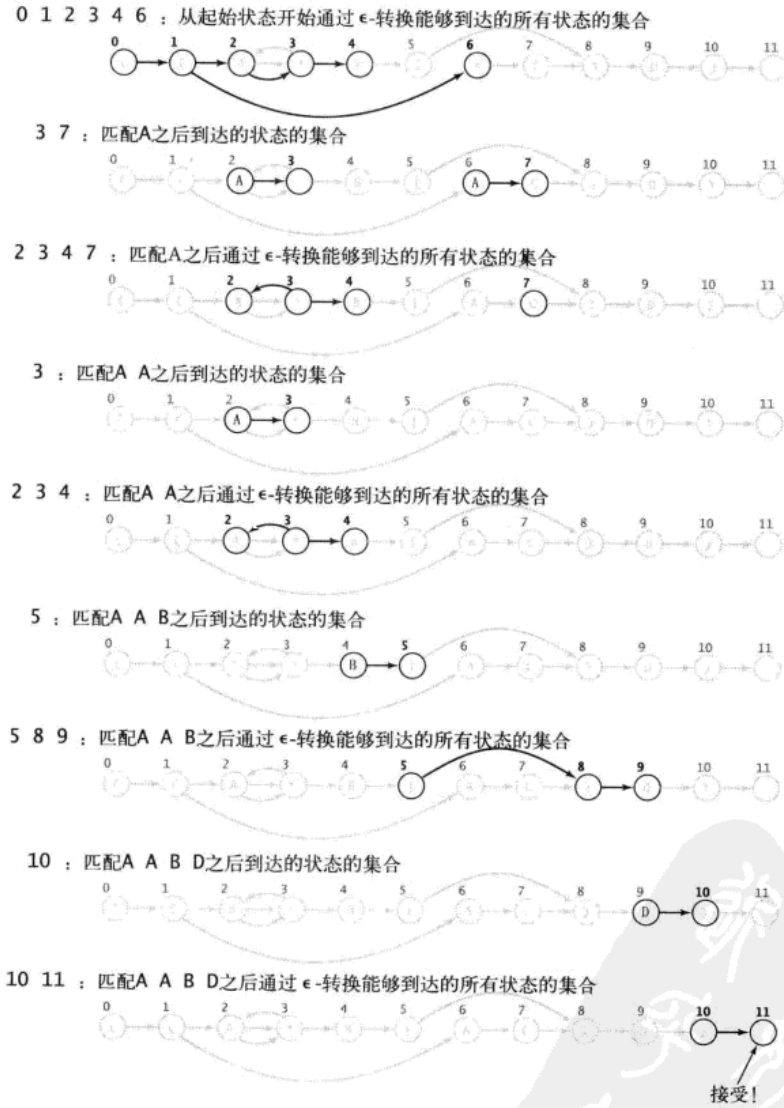


图 5.4.4 对 ((A*B|AC)D) 的 NFA 处理输入 A A B D 的模拟

命题Q。判定一个长度为 M 的正则表达式所对应的NFA能否识别一段长度为 N 的文本所需的时间在最坏情况下和 MN 成正比。

证明。对于长度为 N 的文本中的每个字符，我们都会遍历一个大小不超过 M 的状态集合并在 ϵ- 转换的有向图中进行深度优先搜索。下面即将学习的自动机的构造可以证明该有向图中的边数不会超过 $2M$ 条，因此每次深度优先搜索在最坏情况下的运行时间与 M 成正比。

请仔细思考一下这个不同寻常的结果。它在最坏情况下的成本为文本和模式的长度之积，这个成本和5.3节开始时学习的最坏情况下寻找固定子字符串的初级算法的成本竟然是相同的！

```
public boolean recognizes(String txt)
{  // NFA是否能够识别文本txt?
   Bag<Integer> pc = new Bag<Integer>();
   DirectedDFS dfs = new DirectedDFS(G, 0);
   for (int v = 0; v < G.V(); v++)
      if (dfs.marked(v)) pc.add(v);

   for (int i = 0; i < txt.length(); i++)
   {  // 计算txt[i+1]可能到达的所有NFA状态
      Bag<Integer> match = new Bag<Integer>();
      for (int v : pc)
         if (v < M)
            if (re[v] == txt.charAt(i) || re[v] == '.')
               match.add(v+1);
      pc = new Bag<Integer>();
      dfs = new DirectedDFS(G, match);
      for (int v = 0; v < G.V(); v++)
         if (dfs.marked(v)) pc.add(v);

   }
   for (int v : pc) if (v == M) return true;
   return false;
}
```

使用NFA模拟的模式匹配

798
~
799

5.4.6 构造与正则表达式对应的NFA

根据正则表达式和大家所熟悉的算术表达式的相似性，你肯定不会惊讶于将正则表达式转化为NFA的过程在某种程度上类似于1.3节中使用Dijkstra的双栈算法对表达式求值的过程。这两个过程的不同之处在于：

- 正则表达式中的连接操作并没有运算符；
- 正则表达式的闭包（*）是一个一元运算符；
- 正则表达式只有一个二元运算符，即或（|）。

我们不会在两者的不同和相似之处深究，而是会学习一种为正则表达式量身定做的实现。例如，这里只需要一个栈，而不是两个。

根据上一小节开头讨论的NFA表示，这里只需要构造一个由所有 ϵ- 转换组成的有向图G。正则表达式本身和本节开头学习过的形式定义足以提供所需的所有信息。根据Dijkstra的算法，我们会使用一个栈来记录所有左括号和或运算符的位置。

5.4.6.1 连接操作

对于 NFA，连接操作是最容易实现的了。状态的匹配转换和字母表中的字符的对应关系就是连接操作的实现。

5.4.6.2 括号

我们要将正则表达式字符串中所有左括号的索引压入栈中。每当我们遇到一个右括号，我们最终都会用后文所述的方式将左括号从栈中弹出。和 Dijkstra 算法一样，栈可以很自然地处理嵌套的括号。

5.4.6.3 闭包操作

闭包运算符（*）只可能出现在 (i) 单个字符之后（此时将在该字符和“*”之间添加相互指向的两条 ϵ- 转换），或者是 (ii) 右括号之后，此时将在对应的左括号（即栈顶元素）和“*”之间添加相互指向的两条 ϵ- 转换。

5.4.6.4 “或”表达式

在形如 (A|B) 的正则表达式中，A 和 B 也都是正则表达式。我们的处理方式是添加两条 ϵ- 转换：一条从左括号所对应的状态指向 B 中的第一个字符所对应的状态，另一条从“|”字符所对应的状态指向右括号所对应的状态。将正则表达式字符串中“|”运算符的索引（以及如上文所述的左括号的索引）压入栈中，这样在到达右括号时这些所需信息都会在栈的顶部。这些 ϵ- 转换使得 NFA 能够在这两者之间进行选择。此时并没有像平常一样添加一条从“|”运算符所对应的状态到下一个字符所对应的状态的 ϵ- 转换——NFA 离开“或”运算符的唯一方式就是通过某种状态转换到达右括号所对应的状态。

800

这些简单的规则足以构造任意复杂的正则表达式所对应的 NFA。算法 5.9 实现了这些规则。它的构造函数创建了给定正则表达式所对应的 ϵ- 转换有向图。该算法处理样例的轨迹如图 5.4.7 所示。图 5.4.5、图 5.4.6 和练习中给出了一些其他的例子，我们也希望你自己通过更多的示例加深对这个过程的理解。为了实现的简洁和清晰，我们将一些实现细节（处理元字符、字符集描述符、闭包的缩略写法和多向“或”运算等）留做了练习（请见练习 5.4.16 和练习 5.4.21）。在没有这些扩展的情况，NFA 构造过程所需的代码非常少，是我们所见过的最巧妙的算法之一。

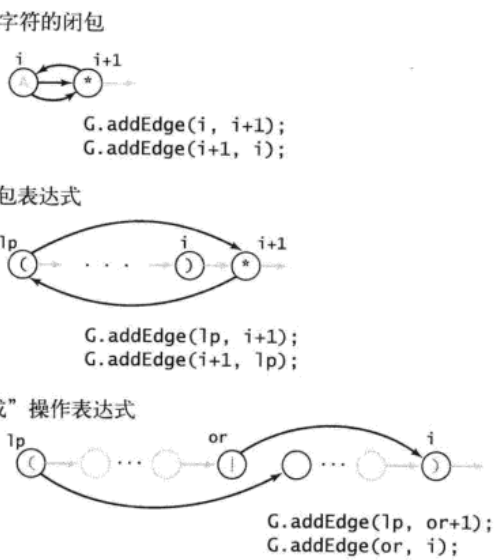


图 5.4.5 NFA 的构造规则

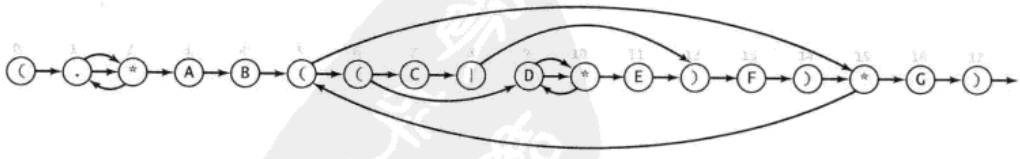


图 5.4.6 模式 (.*AB((C|D*E)F)*G) 所对应的 NFA

801

算法 5.9 正则表达式的模式匹配（grep）

```
public class NFA
{
   private char[] re;              // 匹配转换
   private Digraph G;              // epsilon转换
   private int M;                  // 状态数量

   public NFA(String regexp)
   {  // 根据给定的正则表达式构造NFA
      Stack<Integer> ops = new Stack<Integer>();
      re = regexp.toCharArray();
      M = re.length;
      G = new Digraph(M+1);

      for (int i = 0; i < M; i++)
      {
         int lp = i;
         if (re[i] == '(' || re[i] == '|')
            ops.push(i);
         else if (re[i] == ')')
         {
            int or = ops.pop();
            if (re[or] == '|')
            {
               lp = ops.pop();
               G.addEdge(lp, or+1);
               G.addEdge(or, i);
            }
            else lp = or;
         }
         if (i < M-1 && re[i+1] == '*')  // 查看下一个字符
         {
            G.addEdge(lp, i+1);
            G.addEdge(i+1, lp);
         }
         if (re[i] == '(' || re[i] == '*' || re[i] == ')')
            G.addEdge(i, i+1);
      }
   }
   public boolean recognizes(String txt)
   // NFA是否能够识别文本txt？（请见5.4.5.2节框注“使用NFA模拟的模式匹配”）
}
```

802 该构造函数根据给定的正则表达式构造了对应的 NFA 的 ϵ- 转换有向图。

命题 R。 构造和长度为 M 的正则表达式相对应的 NFA 所需的时间和空间在最坏情况下与 M 成正比。

证明。 对于长度为 M 的正则表达式中的每个字符，最多会添加三条 ϵ- 转换并可能执行一到两次栈操作。

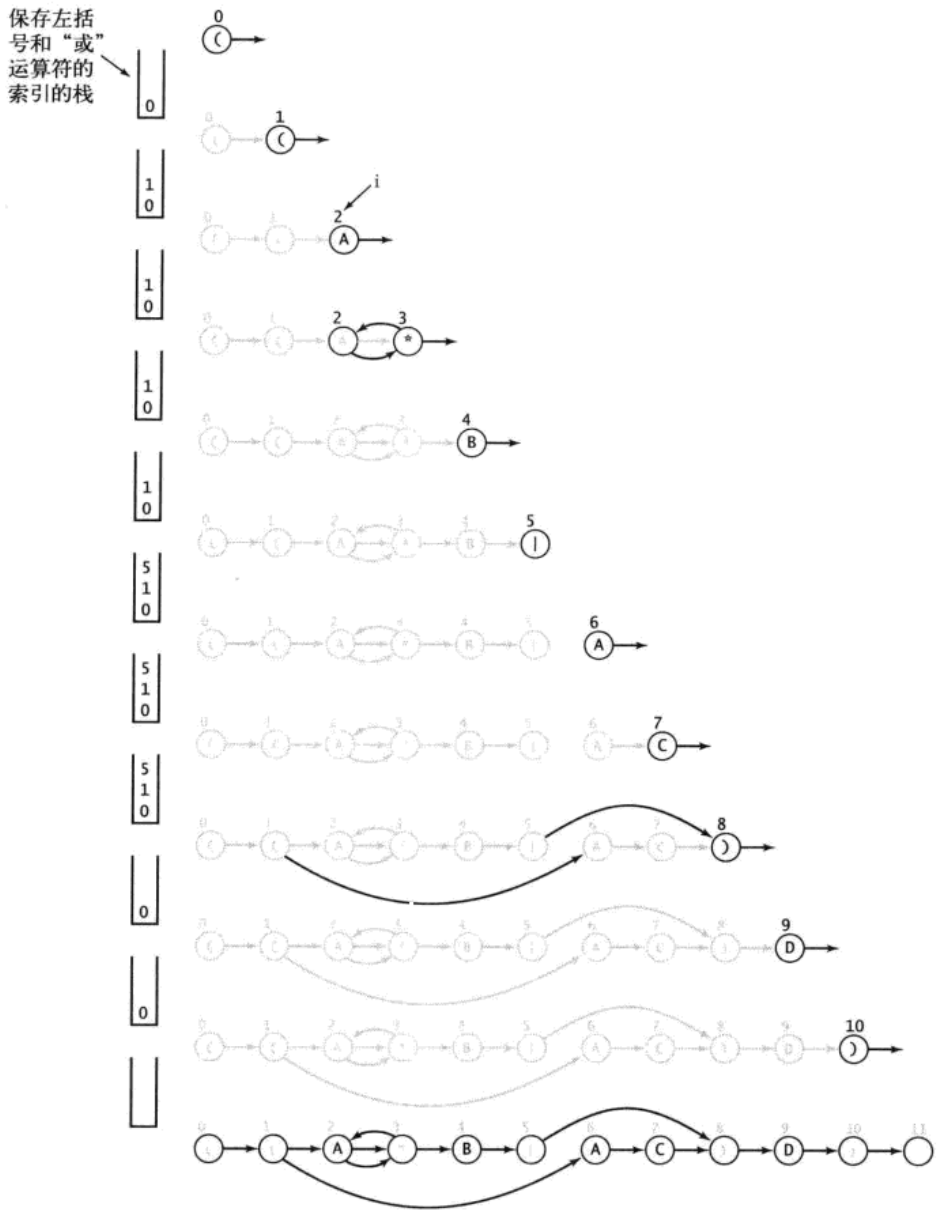


图 5.4.7 构造正则表达式 ((A*B|AC)D) 所对应的 NFA

模式匹配的经典用例 GREP 的代码如后面框注所示。它接受一个正则表达式为参数并能够打印出标准输入中含有属于正则表达式所描述的语言的子字符串的所有行。这个程序是 Unix 早期实现中的一项特性并已经成为数代程序员不可缺少的工具。

```
public class GREP
{
   public static void main(String[] args)
   {
      String regexp = "(.*" + args[0] + ".*)";
      NFA nfa = new NFA(regexp);
      while (StdIn.hasNextLine())
      {
         String txt = StdIn.hasNextLine();
         if (nfa.recognizes(txt))
            StdOut.println(txt);
      }
   }
}
```

```
% more tinyL.txt
AC
AD
AAA
ABD
ADD
BCD
ABCCBD
BABAAA
BABBAAA

% java GREP "(A*B|AC)D" < tinyL.
txt
ABD
ABCCBD

% java GREP StdIn < GREP.java
     while (StdIn.hasNextLine())
        String txt = StdIn.
hasNextLine();
```

803
~
804

经典的一般正则表达式模式匹配(GREP)NFA的用例

答疑

问 空（null）和∈有什么区别?

答 前者表示一个空集，后者表示一个空字符串。你可以构造一个只有一个元素∈的集合，而显然这个集合不是空集（null）。

805

练习

5.4.1 给出能够描述含有以下字符的所有字符串的正则表达式：

- ❑ 4个连续的A
- ❑ 最多4个的连续的A
- ❑ 1到4个连续的A

5.4.2 用自然语言简略的描述以下正则表达式：

a. .*

b. A.*A | A

c. .*ABBABBA.*

d. .* A.*A.*A.*A.*

5.4.3 一个使用M个或运算符且不使用闭包的正则表达式最多能够描述多少个不同的字符串？（可以使用连接操作和括号。）

5.4.4 画出模式(((A|B)*|CD*|EFG)*)*所对应的NFA。

5.4.5 画出练习5.4.4的NFA的∈-转换有向图。

5.4.6 对于输入A B B A C E F G E F G C A A B，给出练习5.4.4的NFA中每次匹配转换和∈-转换之后可达的状态集合。

5.4.7 将5.4.6.4节框注“经典的一般正则表达式模式匹配（GREP）NFA的用例”中的GREP修改为GREPmatch，将模式用括号包裹起来但不在模式两端加上“.*”。这样程序就只会打出属于给定

正则表达式所描述的语言的输入行字符串。给出以下命令的结果。

a. `% java GREPmatch "(A|B)(C|D)" < tinyL.txt`

b. `% java GREPmatch "A(B|C)*D" < tinyL.txt`

c. `% java GREPmatch "(A*B|AC)D" < tinyL.txt`

5.4.8 用正则表达式描述以下二进制字符串的集合。

a. 含有至少 3 个连续的 1

b. 含有子字符串 110

c. 含有子字符串 1101100

d. 不含有子字符串 110

806

5.4.9 用一个正则表达式描述至少含有两个 0 但不含有任何连续的 0 的二进制字符串。

5.4.10 用正则表达式描述以下二进制字符串的集合。

a. 至少含有 3 个字符，且第三个字符为 0

b. 字符串中的 0 的个数为 3 的倍数

c. 起止字符相同

d. 长度为奇数

e. 首字母为 0 且长度为奇数，或者首字母为 1 且长度为偶数

f. 长度在 1 到 3 之间

5.4.11 对于以下正则表达式，计算有多少个长度正好为 1000 的二进制字符串和它们匹配。

a. `0(0 | 1)*1`

b. `0*101*`

c. `(1 | 01)*`

5.4.12 为以下应用写出 Java 的正则表达式。

a. 电话号码，例如 (609) 555-1234

b. 社会保险号，例如 123-45-6789

c. 日期，例如 December 31, 1999

d. 形如 a.b.c.d 的 IP 地址，其中每个字符都表示着一个可能是 1 位、2 位或者 3 位的数字，例如 196.26.155.241

e. 车牌号，前 4 个字符为数字，最后 2 个字符为大写字母

807

提高题

5.4.13 *有难度的正则表达式。*使用二值字母表的正则表达式描述以下字符串的集合。

a. 除了 11 和 111 的所有字符串

b. 奇数位数字为 1 的所有字符串

c. 至少含有两个 0 和至多含有一个 1 的所有字符串

d. 不存在连续两个 1 的所有字符串

5.4.14 *二进制数的可整除性。*使用正则表达式描述以下二进制字符串使得其对应的整数能够满足以下条件。

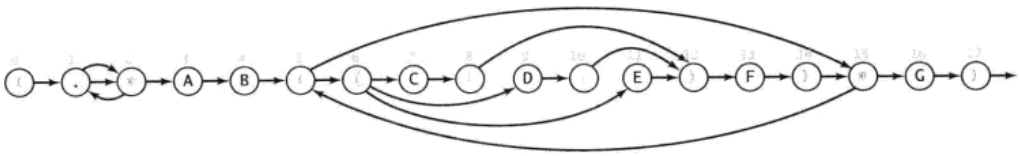
a. 被 2 整除

b. 被 3 整除

c. 被 123 整除

5.4.15 *单层正则表达式*。构造一个 Java 的正则表达式来描述所有二值字母表的合法正则表达式字符串的集合，字符串不含有嵌套的括号。例如，`(.*1)*` 和 `(1.*0)*` 都是这个语言中的字符串，但 `(1(0` 或者 `1)1)*` 不是。

5.4.16 *多向“或”运算*。为 NFA 实现多向“或”运算。代码为模式 `(.*AB((C|D|E)F)*G)` 生成的自动机应该如图 5.4.8 所示。



808

图 5.4.8 模式 `(.*AB((C|D|E)F)*G)` 所对应的 NFA

5.4.17 *通配符*。为 NFA 添加处理通配符的能力。

5.4.18 *至少重复一次*。为 NFA 添加处理闭包的“+”运算符的能力。

5.4.19 *指定重复次数*。为 NFA 添加处理指定重复次数的能力。

5.4.20 *范围描述符*。为 NFA 添加处理指定重复范围的能力。

5.4.21 *补集*。为 NFA 添加处理补集描述符的能力。

5.4.22 *证明*。开发一个新版本的 NFA，使它能够打印一份证明，指出给定字符串包含在 NFA 能够识别的语言之中（即终止于接受状态的一系列状态转换）。

809

5.5 数据压缩

这个世界充满了数据，而能够有效表达数据的算法在现代计算机基础架构中有着重要的地位。压缩数据的原因主要有两点：节省保存信息所需的空间和节省传输信息所需的时间。尽管科技在发展，但是这两点的重要性并没有发生变化，如今任何需要更大存储空间或是长时间等待下载任务完成的人都会意识到数据压缩的重要性。

当你在处理数字图像、声音、电影和其他各种数据时，就已经在与数据压缩打交道了。我们将会学习的算法之所以能够节省空间，是因为大多数数据文件都有很大的冗余：例如，文本文件中有些字符序列的出现频率远高于其他字符串；用来将图片编码的位图文件中可能有大片的同质区域；保存数字图像、电影、声音等其他类似信号的文件都含有大量重复的模式。

我们将会讨论广泛应用的一种初级的算法和两种高级的算法。这些算法的压缩效果可能有所不同，取决于输入的特征。文本数据一般都能节省 20% ~ 50% 的空间，某些情况下能够达到 50% ~ 90%。你将会看到，任何数据压缩算法的效果都十分依赖于输入的特征。注意：本书中，我们在提到性能的时候一般指的都是时间；而对于数据压缩，性能指代的是算法的压缩率，当然也会考虑压缩的用时。

从另一方面来说，现在的数据压缩技术并没有以前那么重要了，因为计算机的存储设备的成本已经大幅度降低，普通用户拥有的存储空间比以前要多得多。但是，现在数据压缩技术也比任何时候都更重要，因为现在存储的数据更多了，因此数据压缩能够节省的空间也就更大了。事实上，随着互联网的出现，数据压缩得到了更加广泛的应用，因为它是减少传输大量数据所需时间的最经济的办法。

数据压缩有着丰富的历史积淀（我们只会作简要的介绍），而它在未来世界中扮演的角色将会更加重要。所有人都能从数据压缩算法的学习中得到益处，因为这些算法都非常经典、优雅、有趣而高效。

810

5.5.1 游戏规则

现代计算机系统中处理的所有类型的数据都有一个共同点：它们最终都是用二进制表示的。我们可以将它们都看成一串比特（或者字节）的序列。简单起见，本节中使用比特流这个术语表示比特的序列，用字节流这个术语表示可以看作固定大小的字节序列的比特序列。比特流或字节流可以是保存在计算机中的文件，也可以是互联网上传输的一条消息。

基础模型

数据压缩的基础模型非常简单（请见图 5.5.1）。它由两个主要的部分组成，两者都是一个能够读写比特流的黑盒子：

- 压缩盒，能够将一个比特流 B 转化为压缩后的版本 C(B)；
- 展开盒，能够将 C(B) 转化回 B。

如果使用 |B| 表示比特流中比特的数量的话，我们感兴趣的是将 |C(B)|/|B| 最小化，这个值被称为压缩率。

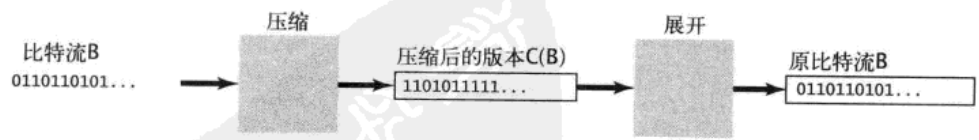


图 5.5.1 数据压缩的基础模型

这种模型叫做*无损压缩模型*——保证不丢失任何信息，即压缩和展开之后的比特流必须和原始的比特流完全相同。许多种类型的文件都会用到无损压缩，例如数值数据或者可执行的代码。对于某些类型的文件（例如图像、视频和音乐），有损的压缩方法也是可以接受的，此时解码器所产生的输出只是与原输入文件近似。有损压缩算法的评价标准不仅是压缩率，还包括主观的质量感受。在本书中不会讨论有损压缩算法。

5.5.2 读写二进制数据

完整描述计算机上信息的编码方式取决于系统，这超出了本书的讨论范围。但我们可以通过几个基本的假设和两个简单的 API 来将实现与这些细节隔离开来。BinaryStdIn 和 BinaryStdOut 这两份 API 来自于我们一直在使用的 StdIn 和 StdOut，但它们的作用是读取和写入比特，而 StdIn 和 StdOut 面向的是由 Unicode 编码的字符流。StdOut 上的一个 int 值是一串字符（它的十进制表示）；
811 BinaryStdOut 上的一个 int 值是一串比特（它的二进制表示）。

5.5.2.1 二进制的输入输出

今天，大多数系统的输入输出系统，包括 Java，都是基于 8 位的字节流，因此我们的 API 也许应该读写字节流，以和原始数据类型内部表示的输入输出格式相匹配，将 8 位的 char 编码为 1 个字节，16 位的 short 编码为 2 个字节，32 位的 int 编码为 4 个字节，等等。因为比特流是数据压缩的主要抽象层次，这就需要更进一步，允许用例读写单个的比特以及原始类型的数据。我们的目标是尽量减少用例需要进行的类型转换并按照操作系统的要求表示数据。表 5.5.1 中的 API 从标准输入中读取比特流。

表 5.5.1 从标准输入读取比特流的静态方法的 API

public class	**BinaryStdIn**	
boolean	readBoolean()	读取 1 位数据并返回一个 boolean 值
char	readChar()	读取 8 位数据并返回一个 char 值
char	readChar(int r)	读取 r（1~16）位数据并返回一个 char 值
[适用于 byte（8 位）、short（16 位）、int（32 位）以及 long 和 double（64 位）的类似方法]		
boolean	isEmpty()	比特流是否为空
void	close()	关闭比特流

和 StdIn 明显不同的是，这份抽象 API 的一个关键特性在于*标准输入中的数据并不一定是与字节边界对齐的*。如果输入流只含有一个字节，用例可以一个比特一个比特地调用 8 次 readBoolean() 方法读取它。虽然 close() 方法并不十分重要，但为了能够终止输入，用例应该使用 close() 方法表示不会再读取任何数据。和 StdIn 与 StdOut 一样，使用表 5.5.2 中的补充 API 来向标准输出写入比特流。

表 5.5.2 向标准输出中写入比特流的静态方法的 API

public class	**BinaryStdOut**	
void	write(boolean b)	写入指定的比特
void	write(char c)	写入指定的 8 位字符
void	write(char c, int r)	写入指定字符的低 r（1~16）位
[适用于 byte（8 位）、short（16 位）、int（32 位）以及 long 和 double（64 位）的类似方法]		
812 void	close()	关闭比特流

对于输出，close() 方法就很重要了：用例必须使用 close() 方法保证之前调用 write() 方法处理的所有数据都写入比特流，比特流的最后一个字节必须用 0 补齐以保证和文件系统的兼容性。StdIn 与 StdOut 有 In 与 Out 这两份 API 与之关联，这里也通过 BinaryIn 和 BinaryOut 直接使用二进制编码的文件。

5.5.2.2 举例

以下是一个简单的示例，假设你用一个数据结构将日期表示为 3 个 int 值（月、日、年）。使用 StdOut 将这些值以 12/31/1999 的格式输出需要 10 个字符，也就是 80 位。如果用 BinaryStdOut 直接输出这些值则需要 96 位（每个 int 值 32 位）；如果用 byte 值来表示月和日，用 short 值表示年，输出将只有 32 位。如果使用 BinaryStdOut，可以只用 4 位、5 位和 12 位的 3 个域，输出总共 21 位，请见图 5.5.2（实际上是 24 位，因为文件必须是完整的 8 位字节，因此 close() 方法会在末尾添加三个 0 位。）注意：这是最粗糙的数据压缩方式。

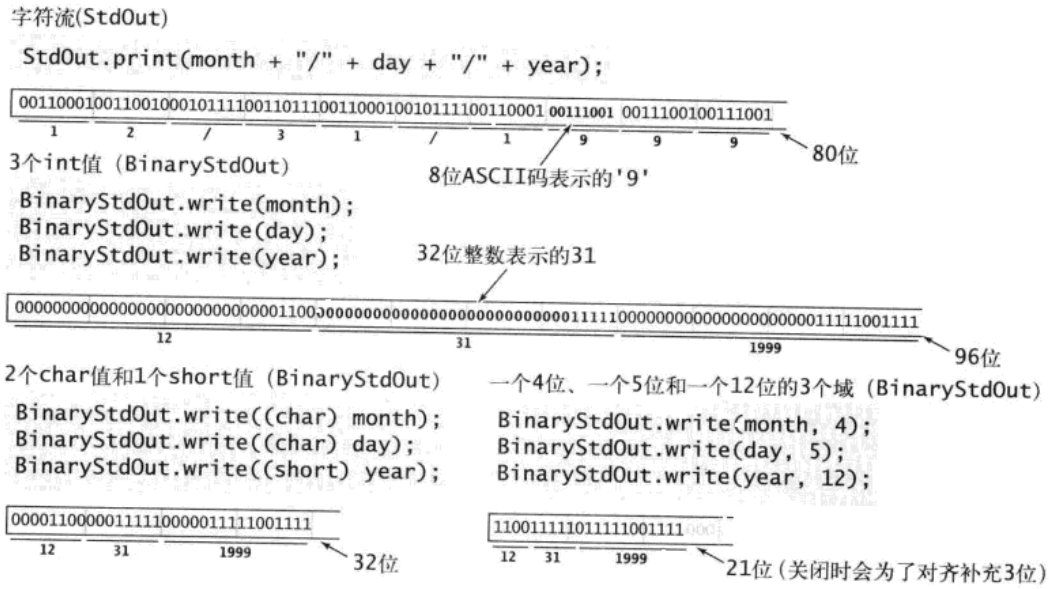


图 5.5.2 向标准输出中写入一个日期的 4 种方法

5.5.2.3 二进制转储

在调试的时候，我们应该如何检查比特流或者字节流的内容呢？早期的程序员面临着这个问题，因为当时寻找 bug 的唯一方式就是检查内存中的每个比特。*转储*（dump）这个词从计算机的早期一直沿用下来，表示的是比特流的一种可供人类阅读的形式。如果你试图用一个编辑器来打开一个二进制文件，或者用文本方式察看一个二进制文件的内容（或者运行一个使用 BinaryStdOut 的程序），那会看到一团乱码，内容取决于使用的系统。BinaryStdIn 可以避开对系统的依赖性，允许我们编写自己的程序来将比特流转化为标准工具能够处理的内容。例如，下页框注所示的程序 BinaryDump 调用了 BinaryStdIn，将标准输入中的比特按照 0 和 1 的形式打印出来。在处理小规模输入时这个程序是一个很有用的调试工具。类似的工具 HexDump 可以将数据组织成 8 位的字节并将它打印为各表示 4 位的两个十六进制数。用例 PictureDump 可以用 Picture 对象表示比特，其中白色像素表示 0，黑色像素表示 1。你可以从本书的网站上下载 BinaryDump、HexDump 和 PictureDump，请见图 5.3.3。我们一般会用管道和重定向等方式在命令行处理二进制文件，将编码器的输出通过管道传递

813

给 BinaryDump、HexDump 或者 PictureDump，或者将它重定向到一个文件之中。

```
public class BinaryDump
{
   public static void main(String[] args)
   {
      int width = Integer.parseInt(args[0]);
      int cnt;
      for (cnt = 0; !BinaryStdIn.isEmpty(); cnt++)
      {
         if (width == 0) continue;
         if (cnt != 0 && cnt % width == 0)
            StdOut.println();
         if (BinaryStdIn.readBoolean())
              StdOut.print("1");
         else StdOut.print("0");
      }
      StdOut.println();
      StdOut.println(cnt + " bits");
   }
}
```

将比特流打印在标准输出上（字符形式）

标准字符流

```
% more abra.txt
ABRACADABRA!
```

用0和1表示的比特流

```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
96位
```

用十六进制数字表示的比特流

```
%  java HexDump 4 < abra.txt
41 42 52 41
43 41 44 41
42 52 41 21
96位
```

用Picture对象中的像素表示的比特流

```
% java PictureDump 16 6 < abra.txt
```



96位

图 5.5.3 查看比特流的 4 种方法

814

5.5.2.4 ASCII 编码

当你使用 HexDump 查看一个含有 ASCII 编码的字符的比特流的内容时，最好参考图 5.5.4。对于给定的两个十六进制数字，用第一个数字表示行、第二个数字表示列即可找到它所表示的字符。例如，31 表示“1”，4A 表示“J”，等等。这张表适用于 7 位 ASCII 码，因此第一个十六进制数字必须是小于等于 7 的。以 0 或者 1 开头的数（以及 20 和 7F）对应的都是无法打印出来的控制字符。许多控制

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

图 5.5.4 十六进制编码和 ASCII 字符的转换表

字符都是为了控制打字机时代的物理设备而遗留下来的产物。我们在这张表中突出了一些你可能在转储中已经见过的字符。例如，SP 是空格符，NUL 是空字符，LF 是换行符，CR 是回车。

总之，在处理数据压缩问题时，除了标准输入输出之外还要能够处理二进制编码的数据。BinaryStdIn 和 BinaryStdOut 提供了我们所需要的方法。它们能够在用例中区分为文件存储和数据传输而输出的信息（供其他程序使用）和为打印而输出的信息（供人类阅读）。 815

5.5.3 局限

为了更好地理解数据压缩算法，你需要了解它们的一些局限性。研究人员已经为此打下了完整而重要的理论基础，本节的最后会简要讨论，但现在我们先来探讨几个方便入门的结论。

5.5.3.1 通用数据压缩

在已经学习了许多重要问题的算法之后，你可能会认为我们的目标是通用性的数据压缩算法，即一个能够缩小任意比特流的算法。但与之相反，我们定下的目标更加朴素，因为通用性的数据压缩是不可能存在的，请见图 5.5.5。

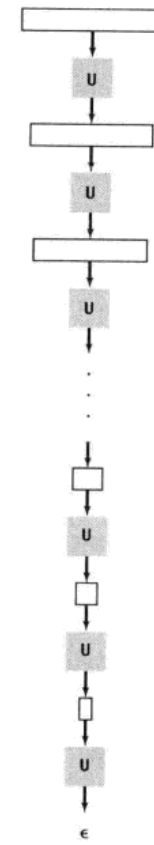


图 5.5.5 是否存在通用数据压缩

命题 S。不存在能够压缩任意比特流的算法。

证明。我们来看两种有见地的证明。第一种采用的是反证法：假设存在一个能够压缩任意比特流的算法，那么也就可以用它压缩它自己的输出以得到一段更短的比特流，循环往复直到比特流的长度为 0！能够将任意比特流的长度压缩为 0 显然是荒谬的，因此存在能够压缩任意比特流的算法的假设也是错误的。

第二种证明方法基于统计：假设有一种算法能够对所有长度为 1000 位的比特流进行无损压缩，那么每一种能够被压缩的比特流都对应着一段较短且不同的比特流。但长度小于 1000 位的比特流一共只有 $1+2+4+\cdots+2^{998}+2^{999}=2^{1000}-1$ 种，而长度为 1000 位的比特流一共有 2^{1000} 种，因此该算法不可能压缩所有长度为 1000 的比特流。如果我们声明更多的条件，那么这段证明会更有说服力。例如，继续假设算法的目标是取得大于 50% 的压缩率，那么显然所有长度为 1000 位的比特流中的压缩成功率将只有 $1/2^{500}$！

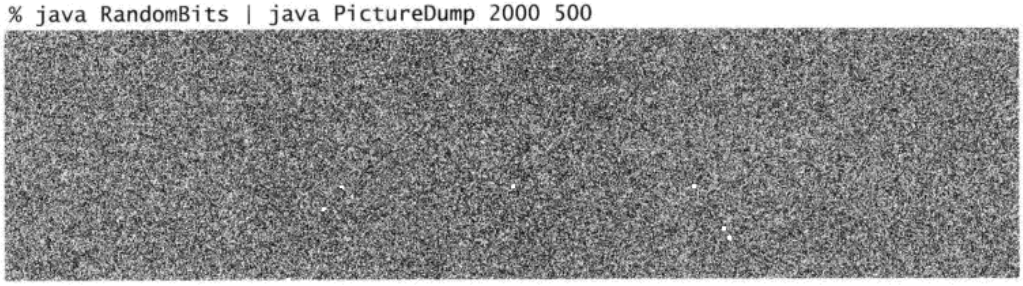
换句话说，对于任意数据压缩算法，将长度为 1000 位的随机比特流压缩为一半的概率最多为 $1/2^{500}$。当遇到一种新的无损压缩算法时，我们可以肯定它是无法大幅度压缩随机比特流的。抛弃对压缩随机比特流的幻想是理解数据压缩的起点。虽然我们会经常处理数百万至数十亿比特长度的字符串，但处理过的数据总量只是这种字符串总数的九牛一毛，所以不必为这个理论结果而沮丧。事实上，经常被处理的比特字符串都是非常有规律的，在压缩时可以利用这一点。 816

5.5.3.2 不可判定性

请见图 5.5.6，它是一条上百万位的字符串。这个字符串看起来很随机，所以你不太可能为它找到一个无损压缩算法。但有一种方法只用几千个比特就可以表示这个字符串，因为它是通过右下

框注中的程序生成的。（这个程序是伪随机数生成器的一个示例，和 Java.Math.random() 方法一样。）通过用 ASCII 文本编写生成程序来进行压缩、通过读取并运行该程序来展开被压缩字符串的压缩算法能够取得 0.3% 的压缩率，这是非常难以超越的。（我们还能够降低这个比例，只要该程序再输出更多比特即可。）压缩这个文件最好的方法就是找出创造这些数据的程序。这个例子并不像它看起来那么深奥：当你在压缩一段视频或是一本通过扫描而数字化的旧书或是互联网上的无数其他类型的文件时，你都在寻找创造这个文件的程序。在意识到我们处理的大部分数据都是由某种程序产生的之后，我们才能发现计算理论中的一些深刻的问题并理解数据压缩所面临的挑战。例如，可以证明最优数据压缩（找到能够产生给定字符串的最短程序）是一个不可判定的问题：我们不但不可能找到能够压缩任意比特流的算法，也不可能找到最佳的压缩算法！

```
% java RandomBits | java PictureDump 2000 500
```



1 000 000位

图 5.5.6 一个难以压缩的文件：100 万（伪）随机比特

这些局限性所带来的实际影响要求无损压缩算法必须尽量利用被压缩的数据流中的*已知*结构。我们将会依次讨论 4 种方法来处理具备以下结构特点的数据：

- 小规模的字母表；
- 较长的连续相同的位或字符；
- 频繁使用的字符；
- 较长的连续重复的位或字符。

如果你已知给定的比特流中具有以上一种或多种特点，那么就能够通过将要学习的 4 种方法将它压缩；如果不知道给定比特流具有的特点，也可以用它们碰碰运气，因为你的数据结构也许并不是那么明显，而这些方法的适用性很广。你将会看到，每种方法都有多个参数和变种，并且可以为特定的比特流调优以达到最佳的压缩率。第一个和最后一个示例是为了帮助你了解数据的结构，接下来我们会学习一个方法来压缩示例数据。

817
≀
818

```
public class RandomBits
{
   public static void main(String[] args)
   {
      int x = 11111;
      for (int i = 0; i < 1000000; i++)
      {
         x = x * 314159 + 218281;
         BinaryStdOut.write(x > 0);
      }
      BinaryStdOut.close();
   }
}
```

"被压缩后的"一段上百万比特的数据流

5.5.4 热身运动：基因组

在讨论更加复杂的数据压缩问题之前，我们先来处理一个初级的（但也十分重要的）数据压缩任务。我们在这个例子中会介绍一些约定，它们将适用于本节中的所有实现。

5.5.4.1 基因数据

作为数据压缩的第一个示例，请看下面这个字符串：

ATAGATGCATAGCGCATAGCTAGATGTGCTAGCAT

如果使用标准的 ASCII 编码（每个字符 1 个字节，8 位），这个字符串的比特流长度为 8 × 35=280 位。这种字符串在现代生物学中非常重要，因为生物学家用字母 A、C、T 和 G 来表示生物体的 DNA 中的四种碱基。基因就是一条碱基的序列。科学家认识到理解基因的性质是理解它们在活体器官中如何作用的关键，包括生命、死亡和疾病。许多生物的基因现在都是已知的，而一些科学家正在编写程序来分析这些序列的结构。

5.5.4.2 双位编码压缩

基因的一个简单性质是，它由 4 种不同的字符组成。这些字符可以用两个比特编码，如右侧的 compress() 方法所示。尽管我们知道输入流是由字符组成的，但是仍然可以使用 BinaryStdIn 来读取这些输入以和标准的数据压缩模型保持一致（从比特流到比特流）。我们在压缩后的文件中记录了被编码的字符数量，这样即使最后一位并没有和字节对齐，解码也能够顺利进行。因为它能够将一个 8 位的字符转换为一个双位编码，且只在最后附加 32 位用于记录总长度，上方程序的压缩率会随着压缩字符的增多越来越接近 25%。

```
public static void compress()
{
   Alphabet DNA = new Alphabet("ACTG");
   String s = BinaryStdIn.readString();
   int N = s.length();
   BinaryStdOut.write(N);
   for (int i = 0; i < N; i++)
   {  // 将字符用双位编码代码表示
      int d = DNA.toIndex(s.charAt(i));
      BinaryStdOut.write(d, DNA.lgR());
   }
   BinaryStdOut.close();
}
```

基因数据的压缩方法

```
public static void expand()
{
   Alphabet DNA = new Alphabet("ACTG");
   int w = DNA.lgR();
   int N = BinaryStdIn.readInt();
   for (int i = 0; i < N; i++)
   {  // 读取2比特，写入一个字符
      char c = BinaryStdIn.readChar(w);
      BinaryStdOut.write(DNA.toChar(c));
   }
   BinaryStdOut.close();
}
```

基因数据的展开方法

5.5.4.3 双位编码展开

右边框注中的 expand() 方法能够将这个 compress() 方法产生的比特流展开。和压缩时一样，该方法会按照数据压缩的基础模型读取一个比特流并输出一个比特流。它输出的比特流和原始输入相同。

相同的方法也适用于其他字母表大小固定的字符串，但我们将它的推广留作（简单的）习题（请见练习 5.5.25）。

这些方法和数据压缩的基础模型并不完全一致，因为编码后的比特流中并没有包含将其解码所需的所有信息。由 A、C、T、G 4 个字母组成的字母表只是两个方法之间的约定。这种约定在基因组这种应用中是合理的，因为这些编码会被大量复用。但在其他的场景中，字母表也可能需要包含在被编码的信息中（请见练习 5.5.25）。在比较数据压缩的方法时我们通常都要计入这些成本。

在基因组学的早期，分析一段染色体序列是一个漫长而艰苦的任务，因此已知的序列都相对较短，科学家可以用标准的 ASCII 编码来存储和交换它们。现在，这个实验流程的效率已经大大提高了，已知

的基因组的数量非常多而且都很长（人类的基因组长度超过 10^{10} 比特）。用这些简单的方法就能节省75%的空间已经非常可观了。还有继续压缩的余地吗？这是一个非常有趣的问题，因为这是一个科学问题：继续压缩的潜力意味着这些数据中还存在着某种结构，而现代基因组学的重点就是希望从基因数据中发现更多的结构。我们将会学习的一些标准数据压缩方法对于（经过双位编码压缩后的）基因数据并没有什么效果，和处理随机数据类似。

小型测试用例（264位）

```
% more genomeTiny.txt
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC

java BinaryDump 64 < genomeTiny.txt
0100000101010100010000010100011101000001010101000100011101000011
0100000101010100010000010100011101000011010001110100001101000001
0101010001000001010001110100001101010100010000010100011101000001
0101010001000111010101000100011101000011010101000100000101000111
01000011
264位

% java Genome - < genomeTiny.txt
?? ←—— 在标准输出上无法看到比特流

% java Genome - < genomeTiny.txt | java BinaryDump 64
0000000000000000000000000000100001001000110010110100100011101110100
1000110110001100101110110110001101000000
104位

% java Genome - < genomeTiny.txt | java HexDump 8
00 00 00 21 23 2d 23 74
8d 8c bb 63 40
104位

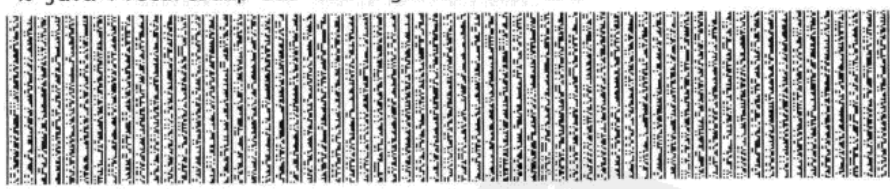
% java Genome - < genomeTiny.txt > genomeTiny.2bit
% java Genome + < genomeTiny.2bit
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC

% java Genome - < genomeTiny.txt | java Genome +
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

压缩–展开循环得到了原始输入

一个真实的病毒（50 000位）

```
% java PictureDump 512 100 < genomeVirus.txt
```



```
50 000位

% java Genome - < genomeVirus.txt | java PictureDump 512 25
```



```
12 536位
```

图 5.5.7 使用双位编码压缩和展开基因组序列

我们将 compress() 和 expand() 作为静态方法和一个简单的用例打包在一个相同的类中，如框注代码所示。为了测试你对游戏规则的理解和我们用于数据压缩的基本工具，请研究图 5.5.7 中的各种命令。它们调用了 Genome.compress() 和 Genome.expand() 来处理样本数据（以及输出）。

```
public class Genome
{
   public static void compress()
   // 请见正文

   public static void expand()
   // 请见正文

   public static void main(String[] args)
   {
      if (args[0].equals("-")) compress();
      if (args[0].equals("+")) expand();
   }
}
```

数据压缩方法的打包方式

819 ≀ 821

5.5.5 游程编码

比特流中最简单的冗余形式就是一长串重复的比特。下面我们学习一种经典的*游程编码*（Run-Length Encoding）来利用这种冗余压缩数据。例如，请看下面这条 40 位长的字符串：

```
0000000000000000111111100000001111111111
```

该字符串含有 15 个 0，然后是 7 个 1，然后是 7 个 0，然后是 11 个 1，因此我们可以将该比特字符串编码为 15，7，7，11。所有的比特字符串都是由交替出现的 0 和 1 组成的，因此我们只需要将游程的长度编码即可。在这个例子中，如果用 4 位表示长度并以连续的 0 作为开头，那么就可以得到一个 16 位长的字符串（15=1111，7=0111，7=0111，11=1011）：

```
1111011101111011
```

压缩率为 16/40=40%。为了将这里的描述转化成一种有效的数据压缩方法，我们需要解决以下几个问题。

- 应该使用多少比特来记录游程的长度？
- 当某个游程的长度超过了能够记录的最大长度时怎么办？
- 当游程的长度所需的比特数小于记录长度的比特数时怎么办？

我们感兴趣的主要是含有的短游程相对较少的长比特流，因此这些问题的回答是：

- 游程长度应该在 0 到 255 之间，使用 8 位编码；
- 在需要的情况下使用长度为 0 的游程来保证所有游程的长度均小于 256；
- 我们也会将较短的游程编码，虽然这样做有可能使输出变得更长。

这些决定非常容易实现而且对于实际应用中经常出现的几种比特流十分有效。它们不适用于含有大量短游程的输入——只有在游程的长度大于将它们用二进制表示所需的长度时才能节省空间。

5.5.5.1 位图

作为游程编码效果的一个示例，这里探讨*位图*。它

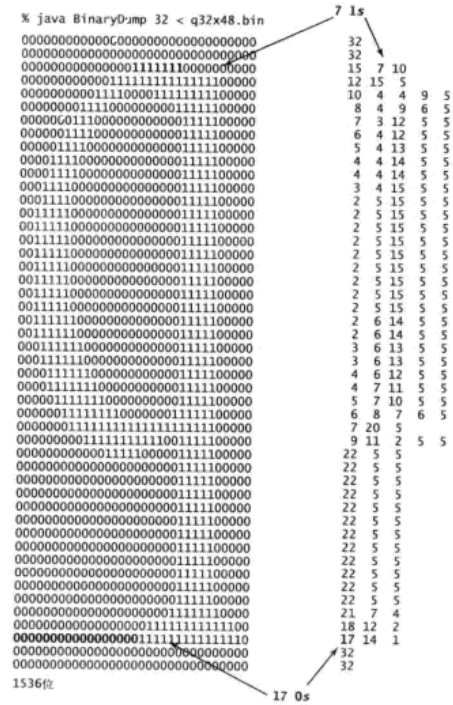


图 5.5.8 一幅典型的位图，每行的游程编码如右所示

被广泛用于保存图像和扫描文档。简单起见，我们将二进制位图数据组织为将像素按行排列的比特流。我们可以用 PictureDump 查看位图的内容。用程序将为“截屏”或是“扫描文档”所定义的多种常见的无损图像格式转化为位图十分简单（请见练习 5.5.*x*）。这里用来展示游程编码的效果的示例来自本书的图像：一个字符“q”（各种分辨率）。我们的重点是一幅 32×48 像素的截图的二进制转储，如图 5.5.8 所示，每行的右侧为该行的游程编码。因为每行的开始和结束都是 0，所以每行的游程数量都是奇数。因为一行的结束之后就是另一行的开始，所以比特流中相对应的游程的长度就是每一行的最后一个游程的长度和下一行的第一个游程的长度之和（全部为 0 的行则应该继续相加）。

5.5.5.2 实现

由刚才给出的非正式描述可以立即得到右边框注中的 compress() 和 expand() 方法。和以前一样，expand() 的实现相对简单：读取一个游程的长度，将当前比特按照长度复制并打印，转换当前比特然后继续，直到输入结束。compress() 方法也很简单。对于输入，它进行了以下操作：

- 读取一个比特；
- 如果它和上一个比特不同，写入当前的计数值并将计数器归零；
- 如果它和上一个比特相同且计数器已经到达最大值，则写入计数值，再写入一个 0 计数值，然后将计数器归零；
- 增加计数器的值。

当输入流结束时，写入计数值（最后一个游程的长度）并结束。

```
public static void expand()
{
   boolean b = false;
   while (!BinaryStdIn.isEmpty())
   {
      char cnt = BinaryStdIn.readChar();
      for (int i = 0; i < cnt; i++)
         BinaryStdOut.write(b);
      b = !b;
   }
   BinaryStdOut.close();
}
public static void compress()
{
   char cnt = 0;
   boolean b, old = false;
   while (!BinaryStdIn.isEmpty())
   {
      b = BinaryStdIn.readBoolean();
      if (b != old)
      {
         BinaryStdOut.write(cnt);
         cnt = 0;
         old = !old;
      }
      else
      {
         if (cnt == 255)
         {
            BinaryStdOut.write(cnt);
            cnt = 0;
            BinaryStdOut.write(cnt);
         }
      }
      cnt++;
   }
   BinaryStdOut.write(cnt);
   BinaryStdOut.close();
}
```

游程编码的压缩和展开方法

5.5.5.3 提高位图的分辨率

游程编码广泛用于位图的主要原因是，随着分辨率的提高它的效果也会大大的提高。证明这一点很简单。假设将上一个例子中的分辨率提高一倍，则很容易得到：

- 总比特数变为了原来的 4 倍；
- 游程的数量变为约原来的 2 倍；
- 游程的长度变为约原来的 2 倍；
- 压缩后的比特数量变为约原来的 2 倍；
- 因此，压缩率变成了原来的一半！

未使用游程编码时，当分辨率提高一倍时图像所需空间变为原来的 4 倍；使用了游程编码后，当分辨率提高一倍时压缩后的比特流的长度仅变为了原来的一倍。也就是说，随着所需空间的增大，压缩比和分辨率成反比。例如，我们的字母“q”（在低分辨率时）的压缩率为 74%；如果将分辨率提高到 64×96，压缩比就下降为

37%。我们从图 5.5.9 中 PictureDump 的输出中可以明显看出这个变化。高分辨率的字符图像所需的空间是低分辨率字符图像的 4 倍（两个维度上的长度均加倍），但压缩后的版本所需的空间仅为原来的 2 倍（只在一个维度上增倍）。如果继续将分辨率提高到 128 × 192（接近于打印所需的分辨率），压缩比则会下降到 18%（请见练习 5.5.5）。

822 ~ 824

小型测试用例（40位）

```
% java BinaryDump 40 < 4runs.bin
0000000000000001111111000000011111111111
40位

% java RunLength - < 4runs.bin | java HexDump
0f 07 07 0b
32位          压缩比32/40=80%

% java RunLength - < 4runs.bin | java RunLength + | java BinaryDump 40
0000000000000001111111000000011111111111 ←—压缩-展开得到了原始输入
40位
```

ASCII文本（96位）

```
% java RunLength - < abra.txt | java HexDump 24
01 01 05 01 01 01 04 01 02 01 01 01 02 01 02 01 05 01 01 01 04 02 01 01
05 01 01 01 03 01 03 01 05 01 01 01 04 01 02 01 01 01 02 01 02 01 05 01
02 01 04 01
416位     ←— 压缩比416/96=433% —— 请勿使用游程编码来处理ASCII文本!
```

一幅位图（1536位）

```
% java RunLength - < q32x48.bin > q32x48.bin.rle
% java HexDump 16 < q32x48.bin.rle
4f 07 16 0f 0f 04 04 09 0d 04 09 06 0c 03 0c 05
0b 04 0c 05 0a 04 0d 05 09 04 0e 05 09 04 0e 05
08 04 0f 05 08 04 0f 05 07 05 0f 05 07 05 0f 05
07 05 0f 05 07 05 0f 05 07 05 0f 05 07 05 0f 05
07 05 0f 05 07 05 0f 05 07 06 0e 05 07 06 0e 05
08 06 0d 05 08 06 0d 05 09 06 0c 05 09 07 0b 05
0a 07 0a 05 0b 08 07 06 0c 14 0e 0b 02 05 11 05
05 05 1b 05 1b 05 1b 05 1b 05 1b 05 1b 05 1b 05
1b 05 1b 05 1b 05 1b 05 1a 07 16 0c 13 0e 41
1144位     ←— 压缩比1144/1536=74%
```

一幅分辨率更高的位图（6144位）

```
% java BinaryDump 0 < q64x96.bin
6144位
% java RunLength - < q64x96.bin | java BinaryDump 0
2296位     ←—  压缩比2296/6144=37%
```

```
% java PictureDump 32 48 < q32x48.bin
```



```
1536位
% java PictureDump 32 36 < q32x48.rle.bin
```



```
1144位

% java PictureDump 64 96 < q64x96.bin
```



```
6144位

% java PictureDump 64 36 < q64x96.rle.bin
```



```
2296位
```

图 5.5.9 使用游程编码压缩和展开比特流

游程编码在许多场景中非常有效，但在许多情况下我们希望压缩的比特流并不含有较长的游程（例如典型的英文文档）。下面我们来学习两种适用于多种类型的文件压缩算法。它们的应用非常广泛，在从网络上下载文件时很可能就用到了它们。

825

5.5.6 霍夫曼压缩

我们现在来学习一种能够大幅压缩自然语言文件空间（以及许多其他类型文件）的数据压缩技术。它的主要思想是放弃文本文件的普通保存方式：不再使用7位或8位二进制数表示每一个字符，而是用较少的比特表示出现频率高的字符，用较多的比特表示出现频率低的字符。

为了说明这个概念，先来看一个简单的示例。假设需要将字符串A B R A C A D A B R A !编码。由7位ASCII字符编码我们可以得到比特字符串：

```
1000001100001010100101000001100001111000001-
1000100100000110000101010010100000010100001.
```

要将这段比特字符串解码，只需每次读取7位并根据图5.5.4的ASCII编码表将它转换为字符。在这种标准的编码下，只出现了一次的D和出现了5次的A所需的比特数是一样的。霍夫曼压缩的思想是通过用较少的比特表示出现频繁的字符而用较多的比特表示偶尔出现的字符来节省空间，这样字符串所使用的总比特数就会降低。

5.5.6.1 变长前缀码

和每个字符所相关联的编码都是一个比特字符串，就好像有一个以字符为键、比特字符串为值的符号表一样。我们可以试着将最短的比特字符串赋予最常用的字符，将A编码为0、B编码为1、R为00、C为01、D为10、！为11。这样A B R A C A D A B R A !的编码就是0 1 00 0 01 0 10 0 1 00 0 11。这种表示方法只用了17位，而7位的ASCII编码则用了77位。但这种表示方法并不完整，因为它需要空格来区分字符。如果没有空格，比特字符串就会变成这个样子：

```
01000010100100011
```

它也可以被解码为C R R D D C R C B或是其他字符串。但17位加上10个分隔符也比标准的编码要紧凑的多了，没有用于编码的比特字符不会在这条消息中出现。*如果所有字符编码都不会成为其他字符编码的前缀，那么就不需要分隔符了。*下一步我们就要做到这一点。含有这种性质的编码规则叫做*前缀码*。刚才我们给出的编码并不是前缀码，因为A的编码0就是R的编码00的前缀。例如，如果我们将A编码为0、B为1111、C为110、D为100、R为1110、!为101，那么将以下长为30的比特字符串解码的方式就只有A B R A C A D A B R A !一种了：

```
011111110011001000111111100101
```

所有的前缀码的解码方式都和它一样，是*唯一的*（不需要任何分隔符），因此前缀码被广泛应用于实际生产之中。注意，像7位ASCII编码这样的定长编码也是前缀码。

5.5.6.2 前缀码的单词查找树

表示前缀码的一种简便方法就是使用单词查找树（请见5.2节）。事实上，任意含有M个空链接的单词查找树都为M个字符定义了一种前缀码方法：我们将空链接替换为指向叶子结点（含有两个空链接的结点）的链接，每个叶子结点都含有一个需要编码的字符。这样，每个字符的编码就是从根结点到该结点的路径表示的比特字符串，其中左链接表示0，右链接表示1。例如，图5.5.10显示了字符串A B R A C A D A B R A !中的字符的两种前缀码方式。上方的例子就是我们刚才提到的编码方式，下方的编码得到的比特字符串为：

```
11000111101011100110001111101
```

该字符串只有29位，比上一种少1位。是否存在能够压缩得更多的单词查找树呢？我们如何才能找到压缩率最高的前缀码？实际上，这些问题都有一个优雅的解。有一种算法能够为任意字符

串构造一棵能够将比特流最小化的单词查找树。为了公平比较各种编码，还需要计算编码本身所需的空间，因为没有它就无法将字符串解码。你会看到，编码的方式是和字符串相关的。寻找最优前缀码的通用方法是D.Huffman在1952年发现的(当时他还是个学生！)，因此被称为霍夫曼编码。

5.5.6.3 概述

使用前缀码进行数据压缩需要经过5个主要步骤。我们将待编码的比特流看作一个字节流并按照以下方式使用前缀码：

- ❑ 构造一棵编码单词查找树；
- ❑ 将该树以字节流的形式写入输出以供展开时使用；
- ❑ 使用该树将字节流编码为比特流。

在展开时需要：

- ❑ 读取单词查找树（保存在比特流的开头）；
- ❑ 使用该树将比特流解码。

为了帮助你更好地理解和领会这个过程，我们将按照难度逐个考察这些步骤。

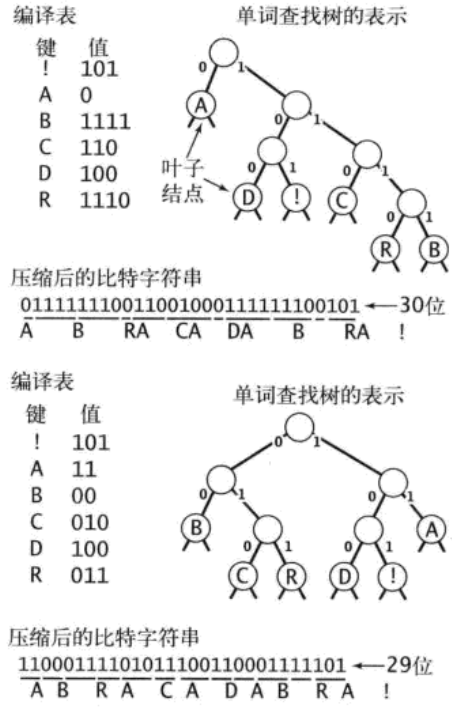


图 5.5.10 两种不同的前缀码

826 ≀ 827

5.5.6.4 单词查找树的结点

我们首先遇到的是如后面框注所示的Node类。它和我们曾经用来构造二叉树和单词查找树的嵌套类相似：每个Node对象都含有指向其他Node对象的`left`和`right`引用，这定义了单词查找树的结构。每个Node对象还包含一个实例变量`freq`，构造函数会用到它。另一个实例变量`ch`用于表示叶子结点中需要被编码的字符。

```
private static class Node implements Comparable<Node>
{  // 霍夫曼单词查找树中的结点
   private char ch;   // 内部结点不会使用该变量
   private int freq;  // 展开过程不会使用该变量
   private final Node left, right;

   Node(char ch, int freq, Node left, Node right)
   {
      this.ch    = ch;
      this.freq  = freq;
      this.left  = left;
      this.right = right;
   }

   public boolean isLeaf()
   {  return left == null && right == null;  }

   public int compareTo(Node that)
   {  return this.freq - that.freq;  }

}
```

单词查找树的结点表示

```
public static void expand()
{
   Node root = readTrie();
   int N = BinaryStdIn.readInt();
   for (int i = 0; i < N; i++)
   {  // 展开第i个编码所对应的字母
      Node x = root;
      while (!x.isLeaf())
         if (BinaryStdIn.readBoolean())
               x = x.right;
         else x = x.left;
      BinaryStdOut.write(x.ch);
   }
   BinaryStdOut.close();
}
```

前缀码的展开（解码）

5.5.6.5 使用前缀码展开

有了定义前缀码的单词查找树，扩展被编码的比特流就简单了。后面框注中的 expand() 方法实现了这个过程。在从标准输入中使用后文所述的 readTrie() 方法读取了单词查找树之后，用它将比特流的其余部分展开：根据比特流的输入从根结点开始向下移动（读取一个比特，如果为 0 则移动到左子结点，如果为 1 则移动到右子结点）。当遇到叶子结点后，输出该结点的字符并重新回到根结点。如果你仔细研究这个方法在图 5.5.11 中的小型前缀码示例中的表现，就能够理解这个过程。例如，在解码比特流 011111001011... 时，从根结点开始，因为第一个比特是 0，所以移动到左子结点，输出 A；回到根结点，向右子结点移动 3 次，然后输出 B；回到根结点，向右子结点移动两次，左子结点移动 1 次，输出 R；如此往复。展开的简单性也是前缀码，特别是霍夫曼压缩算法流行的原因之一。

828

5.5.6.6 使用前缀码压缩

在压缩时，我们使用单词查找树定义的编码来构造编译表，如后面框注中的 buildCode() 方法所示。该方法短小而优雅，其巧妙之处值得仔细研究。对于任意单词查找树，它都能产生一张将树中的字符和比特字符串（用由 0 和 1 组成的 String 字符串表示）相对应的编译表。编译表就是一张将每个字符和它的比特字符串相关联的符号表：为了提升效率，我们使用了一个由字符索引的数组 st[] 而非普通的符号表，因为字符的数量并不多。在构造该符号表时，buildCode() 递归遍历整棵树并为每个结点维护了一条从根结点到它的路径所对应的二进制字符串（0 表示左链接，1 表示右链接）。每当到达一个叶子结点时，算法就将结点的编码设为该二进制字符串。编译表建立之后，压缩就很简单了，只需在其中查找输入字符所对应的编码即可。使用后面框注中的编码压缩 A B R A C A D A B R A !，首先写入 0（A 的编码），然后是 111（B 的编码），然后是 110（R 的编码），等等。框注中的这一段代码完成的任务是查找输入的每个字符所对应的编码 String 对象，将 char 数组中字符转化为 0 和 1 的值并写入输出的比特字符串中。

编译表

键	值
!	1010
A	0
B	111
C	1011
D	100
R	110



图 5.5.11 一种霍夫曼编码

```
private static String[] buildCode(Node root)
{  // 使用单词查找树构造编译表
   String[] st = new String[R];
   buildCode(st, root, "");
   return st;
}
private static void buildCode(String[] st, Node x, String s)
{  // 使用单词查找树构造编译表（递归）
   if (x.isLeaf())
   {  st[x.ch] = s; return; }
   buildCode(st, x.left,  s + '0');
   buildCode(st, x.right, s + '1');
}
```

通过前缀码字典查找树构建编译表

```
for (int i = 0; i < input.length; i++)
{
   String code = st[input[i]];
   for (int j = 0; j < code.length(); j++)
      if (code.charAt(j) == '1')
           BinaryStdOut.write(true);
      else BinaryStdOut.write(false);
}
```

使用编译表的压缩

5.5.6.7 单词查找树的构造

作为描述过程的参考，图 5.5.12 展示了为以下输入构造一棵霍夫曼单词查找树的过程：

```
it was the best of times it was the
worst of times
```

我们将需要被编码的字符放在叶子结点中并在每个结点中维护了一个名为 freq 的实例变量来表示以它为根结点的子树中的所有字符出现的频率。构造的第一步是创建一片由许多只有一个结点（即叶子结点）的树所组成的森林。每棵树都表示输入流中的一个字符，每个结点中的 freq 变量的值都表示了它在输入流中的出现频率。在我们的例子中，输入含有 8 个 t，5 个 e，11 个空格等（特别提示：为了得到这些频率，需要读取整个输入流——霍夫曼编码是一个两轮算法，因为需要再次读取输入流才能压缩它）。接下来自底向上根据频率构造这棵编码的单词查找树。在构造时将它看作一棵结点中含有频率信息的二叉树；在构造后，我们才将它看作一棵用于编码的单词查找树。构造过程如下：首先找到两个频率最小的结点，然后创建一个以二者为子结点的新结点（新结点的频率值为它的两个子结点的频率值之和）。这个操作会将森林中树的数量减一。然后不断重复这个过程，找到森林中的两棵频率最小的树并用相同的方式创建一个新的结点。用优先队列能够轻易实现这个过程，如左下框注的 buildTrie 方法所示。（为了说明这个过程，图 5.5.12 中的所有单词查找树是有序的。）随着这个过程的继续，我们构造的单词查找树将越来越大，而森林中的树会越来越少（每一步都会删除两棵树，添加一棵新树）。最终，所有的结点会被合并为一棵单独的单词查找树。这棵树中的叶子结点为所有待编码的字符和它们在输入中出现的频率，每个非叶子结点中的频率值为它的两个子结点之和。频率较低的结点会被安排在树的底层，而高频率的结点则会被安排在根结点附近的地方。根结点的频率值等于输入中的字符数量。因为这是一棵二叉树且字符仅存在于叶子结点中，所以就定义了这些字符的前缀码。使用 buildCode() 方法为这个示例构造的编译表（如图 5.5.13 的右侧所示），得到了以下输出：

```
private static Node buildTrie(int[] freq)
{
   // 使用多棵单结点树初始化优先队列
   MinPQ<Node> pq = new MinPQ<Node>();
   for (char c = 0; c < R; c++)
      if (freq[c] > 0)
         pq.insert(new Node(c, freq[c], null, null));

   while (pq.size() > 1)
   {  // 合并两棵频率最小的树
      Node x = pq.delMin();
      Node y = pq.delMin();
      Node parent = new Node('\0', x.freq + y.freq, x, y);
      pq.insert(parent);
   }
   return pq.delMin();
}
```

构造一棵霍夫曼编码单词查找树

```
10111110100101101110001111110010000110101100-
01001110100111100001111101111010000100011011-
11101001011011100011111100100001001000111010-
01001110100111100001111101111010000100101010.
```

这个比特字符串长176位，相比用标准的8位ASCII编码得到的51个字符的408位编码节省了57%（没有计算构造编码的开销，下面马上讨论）。另外，因为它是一个霍夫曼编码，所以不存在其他能够用更少的比特将输入编码的前缀码了。

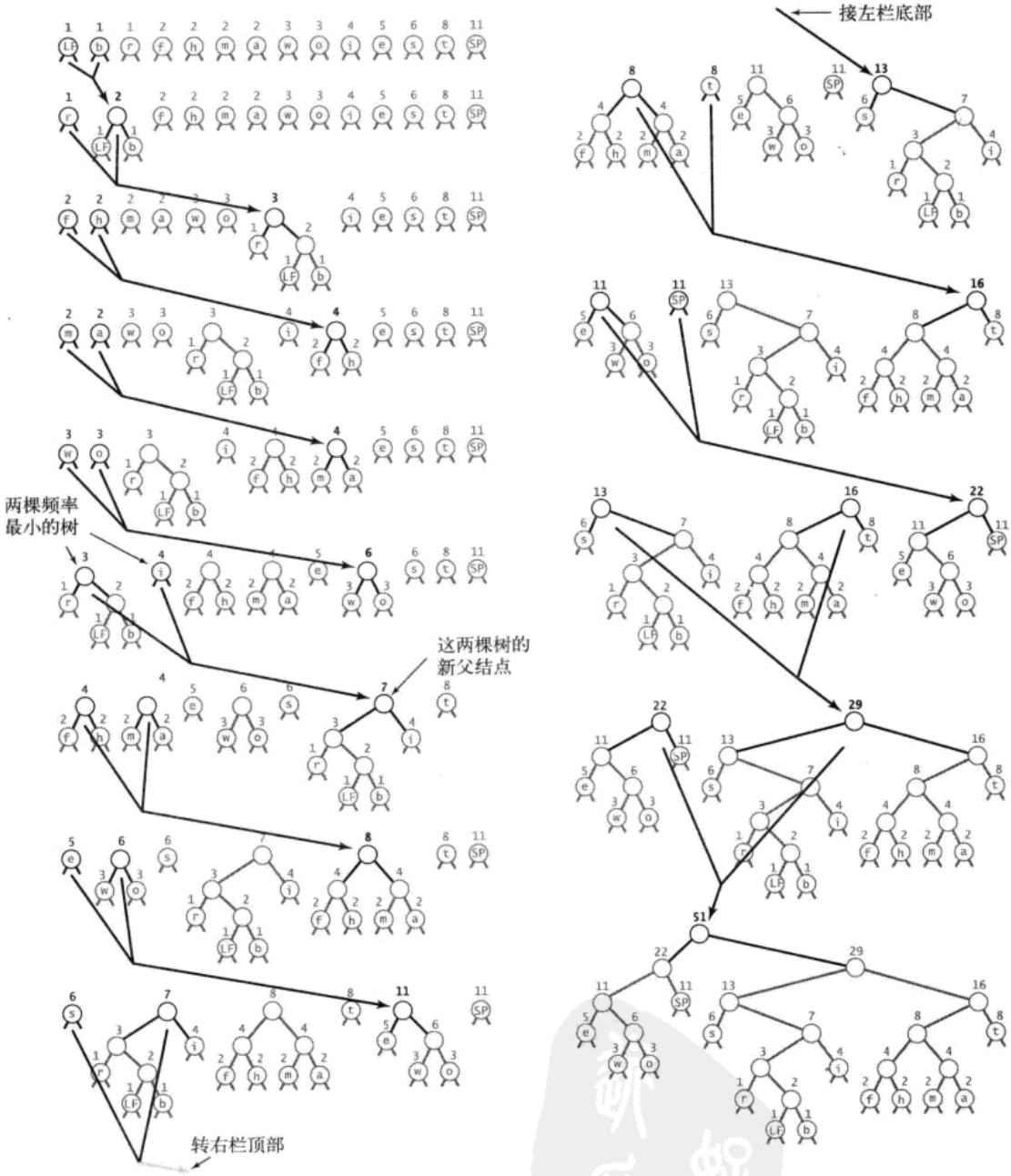


图 5.5.12 构造一棵霍夫曼编码单词查找树

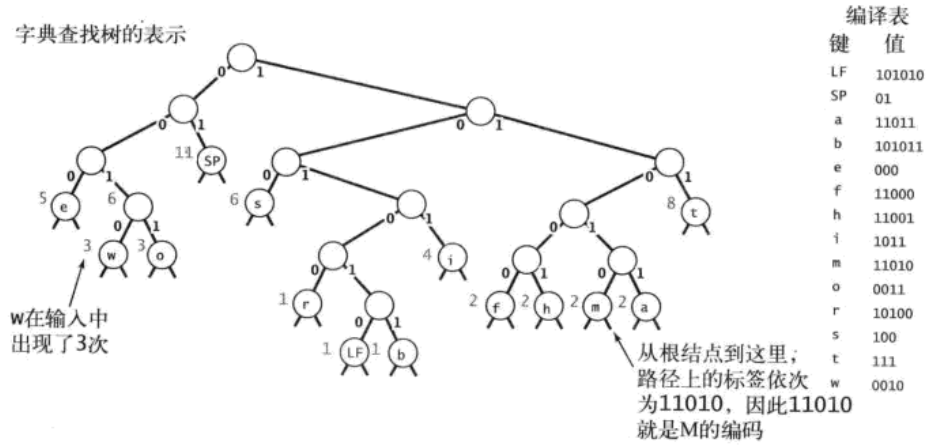


图 5.5.13 字符串“it was the best of times it was the worst of times LF”的霍夫曼编码

5.5.6.8 最优性

我们已经看到，在树中高频率的字符比低频率的字符离根结点更近，因此编码所需的比特更少，所以这种编码的方式更好。但为什么这是一种最优的前缀码呢？要回答这个问题，首先要定义树的加权外部路径长度这个概念，它是所有叶子结点的权重（频率）和深度（请见 1.5.2.5 节）之积的和。

829 ≀ 832

命题 T。对于任意前缀码，编码后的比特字符串的长度等于相应单词查找树的加权外部路径长度。

证明。每个叶子结点的深度就是将该叶子结点的字符编码所需的比特数。因此，加权外部路径长度就是编码后的比特字符串的长度：它等于所有字符的出现次数和字符的编码长度之积的和。

在示例中，有一个叶子结点的距离为 2（SP，出现频率为 11），三个距离为 3（e、s 和 t，总频率为 19），三个距离为 4（w、o 和 i，总频率为 10），五个距离为 5（r、f、h、m 和 a，总频率为 9），两个距离为 6（LF 和 b，总频率为 2），因此综合为 $2\times11+3\times19+4\times10+5\times9+6\times2=176$。这与输出的比特字符串的长度预期相等。

命题 U。给定一个含有 r 个符号的集合和它们的频率，霍夫曼算法所构造的前缀码是最优的。

证明。数学归纳法。假设霍夫曼编码对于任意规模小于 r 的符号集合都是最优的。设 T_H 是用霍夫曼算法计算并编码符号集和相应的频率 $(s_1,f_1),\cdots,(s_r,f_r)$ 所得到的输出，并用 $W(T_H)$ 表示输出的总长度（单词查找树的加权外部路径长度）。假设 (s_i,f_i) 和 (s_j,f_j) 是最先被选中的两个符号，那么算法接下来将计算 (s_i,f_i) 和 (s_j,f_j) 被 (s^*,f_{i+j}) 替代后的 $r-1$ 个符号的集合的编码以输出 T_H^*，其中 s^* 表示深度为 d 的某个叶子结点中的新符号。可以注意到：

$$W(T_H)=W(T_H^*)-d(f_i+f_j)+(d+1)(f_i+f_j)=W(T_H^*)+(f_i+f_j)$$

现在，假设 $(s_1,f_1),\cdots,(s_r,f_r)$ 有一棵最优的高度为 h 的单词查找树 T。注意，(s_i,f_i) 和 (s_j,f_j) 的深度必然都是 h（否则将它们和深度为 h 的结点交换就可以得到一棵加权外部路径长度更小的单词查找树）。另外，通过将 (s_j,f_j) 和 (s_k,f_k) 的兄弟结点交换可以假设 (s_k,f_k) 和 (s_j,f_j) 是兄弟结点。现在，考虑将它们的父结点替换为 (s^*,f_i+f_j) 所得到的树 T^*。注意（用同样的方法可以得到）$W(T)=W(T^*)+(f_i+f_j)$。

根据归纳法，T_H^* 是最优的，即 $W(T_H^*)\leqslant W(T^*)$。因此有：

$$W(T_H)=W(T_H^*)+(f_i+f_j)\leqslant W(T^*)+(f_i+f_j)=W(T)$$

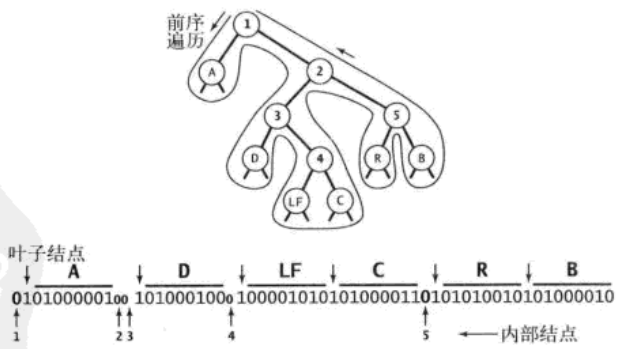
833 因为 T 是最优的，等号必然成立，因此 T_H 也是最优的。

每当一个结点被选中时，也可能有若干个结点和它的权重相同。霍夫曼算法并没有说明如何区别它们，也没有说明应该如何确定子结点的左右位置。不同的选择会得到不同的霍夫曼编码，但用它们将信息编码所得到的比特字符串在所有前缀码中都是最优的。

5.5.6.9 写入和读取单词查找树

我们已经强调过，图 5.5.13 中所显示出的空间节约并不准确，因为没有单词查找树被压缩的比特流是无法被解码的。所以，我们必须将输出比特字符串中的单词查找树的成本考虑进来。对于较长的输入，这个成本相对较小。但为了保证数据压缩流程的完整，必须在压缩时将树写入比特流并在展开时读取它。怎样才能将一棵单词查找树编码为比特流并展开它呢？其实，只要基于单词查找树的前序遍历，这两个任务都只需要很简单的递归即可完成。下面框注中的 writeTrie() 方法会按照前序遍历单词查找树：当它访问的是一个内部结点时，它会写入一个比特 0；当它访问的是一个叶子结点时，它会写入一个比特 1，紧接着是该叶子结点中字符的 8 位 ASCII 编码。A B R A C A D A B R A! 的霍夫曼树的比特字符串编码如图 5.5.14 所示。第一位是 0，对应着根结点；下一个遇到是含有 A 的叶子结点，因此下一位为 1，紧接着是 01000001，即“A”的 8 位 ASCII 编码。下两位均为 0，因为遇到的都是两个内部结点，等等。相应的 readTrie() 如框注所示。它从比特字符串中重新构造了单词查找树：首先读取一个比特以得到当前结点的类型，如果是叶子结点（比特为 1）那么就读取字符的编码并创建一个叶子结点；如果是内部结点（比特为 0）那么就创建一个内部结点并（递归地）继续构造它的左右子树。请一定要理解这些方法：它们的简洁性有时是有欺骗性的。

```
private static void
writeTrie(Node x)
{  // 输出单词查找树的比特字符串
   if (x.isLeaf())
   {
      BinaryStdOut.write(true);
      BinaryStdOut.write(x.ch);
      return;
   }
   BinaryStdOut.write(false);
   writeTrie(x.left);
   writeTrie(x.right);
}
```



834 将单词查找树写为比特字符串

图 5.5.14 使用前序遍历将一棵单词查找树编码为比特流

```
private static Node readTrie()
{
   if (BinaryStdIn.readBoolean())
      return new Node(BinaryStdIn.readChar(), 0, null, null);
   return new Node('\0', 0, readTrie(), readTrie());
}
```

从比特流的前序表示中重建单词查找树

5.5.6.10 霍夫曼压缩的实现

算法 5.10 加上之前讨论过的 buildCode()、buildTrie()、readTrie() 和 write-Trie()（以及一开始展示的 expand() 方法），就是霍夫曼压缩算法的完整实现。为了展开前文对算法的概述，我们将需要压缩的比特流看作 8 位编码的 Char 值流并将它按照如下方法压缩：

- 读取输入；
- 将输入中的每个 char 值的出现频率制成表格；
- 根据频率构造相应的霍夫曼编码树；
- 构造编译表，将输入中的每个 char 值和一个比特字符串相关联；
- 将单词查找树编码为比特字符串并写入输出流；
- 将单词总数编码为比特字符串并写入输出流；
- 使用编译表翻译每个输入字符。

要展开一条编码过的比特流，步骤如下：

- 读取单词查找树（编码在比特流的开头）；
- 读取需要解码的字符数量；
- 使用单词查找树将比特流解码。

霍夫曼压缩算法含有 4 个递归方法处理单词查找树，整个压缩过程需要 7 步，是我们学习的较为复杂的算法之一，请见图 5.5.15。但因为效率高，它也是应用最广泛的算法之一。 835

算法 5.10 霍夫曼压缩

```
public class Huffman
{
   private static int R = 256;   // ASCII字母表
   // Node内部类请见5.5.6.4节框注“单词查找树的结点表示”
   // 其他辅助方法和expand()方法请见正文

   public static void compress()
   {
      // 读取输入
      String s = BinaryStdIn.readString();
      char[] input = s.toCharArray();
      // 统计频率
      int[] freq = new int[R];
      for (int i = 0; i < input.length; i++)
         freq[input[i]]++;
      // 构造霍夫曼编码树
      Node root = buildTrie(freq);
      // （递归地）构造编译表
      String[] st = new String[R];
```

```
        buildCode(st, root, "");

        // (递归地)打印解码用的单词查找树
        writeTrie(root);

        // 打印字符总数
        BinaryStdOut.write(input.length);

        // 使用霍夫曼编码处理输入
        for (int i = 0; i < input.length; i++)
        {
           String code = st[input[i]];
           for (int j = 0; j < code.length(); j++)
           if (code.charAt(j) == "1")
                BinaryStdOut.write(true);
           else BinaryStdOut.write(false);
        }
        BinaryStdOut.close();
     }
  }
```

836 这段霍夫曼编码算法的实现构造了一棵清晰的编码单词查找树并使用了前文所述的各种辅助方法。

测试用例(96位)

```
% more abra.txt
ABRACADABRA!

% java Huffman - < abra.txt | java BinaryDump 60
010100000100101000100010000101010100001101010100101010000100
000000000000000000000000000011000111110010110100011111010100
120位      ←— 压缩率120/96=125%，原因是字典查找树需要59位，字符总数需要32位
```

正文中的例子(408位)

```
% more tinytinyTale.txt
it was the best of times it was the worst of times

% java Huffman - < tinytinyTale.txt | java BinaryDump 64
000101100101010111011111011011111001000000010111001100101110010010
000010101011000101011010010001011001101011010000101101101101100
011011101000000000000000000000000000011001110111110100101101110
011111100100001101011000100111010011110000111110111101000010001
011111010010110111000111111001000010010001110100100111010011110
001111101111010000100101010000000
352位      ←— 压缩率352/408=86%，尽管字典查找树占用了137位，字符总数占用了32位

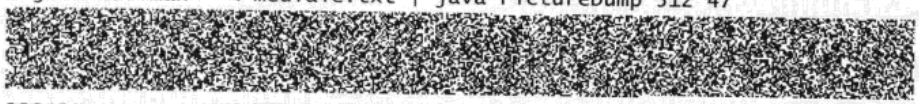
% java Huffman - < tinytinyTale.txt | java Huffman +
it was the best of times it was the worst of times
```

《双城记》的第一章

```
% java PictureDump 512 90 < medTale.txt
```



图 5.5.15 使用霍夫曼编码压缩和展开字节流

```
45056位
% java Huffman - < medTale.txt | java PictureDump 512 47

23912位      ←— 压缩率23912/45056=53%

《双城记》全文
% java BinaryDump 0 < tale.txt
5812552位

% java Huffman - < tale.txt > tale.txt.huf
% java BinaryDump 0 < tale.txt.huf
3043928位      ←— 压缩率3043928/5812552=52%
```

图 5.5.15 使用霍夫曼编码压缩和展开字节流（续） 837

霍夫曼压缩算法流行的一个原因是，不仅对于自然语言文本，它对各种类型的文件都有效果。我们在编写方法的代码时十分小心，以保证它能够正确处理 8 位字符可能表示的任意 8 位值。换句话说，我们可以将它应用于任何字节流。对于我们在本节中讨论过的其他几种类型的文件，图 5.5.16 显示了这些例子，说明了霍夫曼压缩与定长编码以及游程编码相比仍然十分具有竞争力，尽管这些算法是为某些类型的文件专门设计的。理解霍夫曼编码在这些领域的优越性能是十分有帮助的。对于基因组数据，霍夫曼压缩实际上发现了双位编码。因为 4 种字符的出现频率基本相同，因此霍夫曼编码树是平衡的，每个字符分配到的都是一个两位的编码。在游程编码的示例中，00000000 和 11111111 都可能是出现最频繁的字符，因此它们的编码可能只有 2 ~ 3 位，这样就能够大幅度地压缩输入数据。

```
病毒（50 000位）
% java Genome  - < genomeVirus.txt | java PictureDump 512 25

12536位
% java Huffman - < genomeVirus.txt | java PictureDump 512 25

12576 bits ←— 霍夫曼编码只比自定义的双位编码多使用了40个比特

位图（1536位）
% java RunLength - < q32x48.bin | java BinaryDump 0
1144 位

% java Huffman   - < q32x48.bin | java BinaryDump 0
816 位     ←— 霍夫曼压缩算法比自定义算法使用的比特数少29%

更高分辨率的位图（6144位）
% java RunLength - < q64x96.bin | java BinaryDump 0
2296 位

% java Huffman   - < q64x96.bin | java BinaryDump 0
2032 位     ←— 对于更高的分辨率，差距缩小到11%
```

图 5.5.16 用霍夫曼编码压缩和展开基因组和位图数据 838

除了霍夫曼压缩算法，另一种值得一提的选择是20世纪70年代末至80年代初由A.Lempel、J.Ziv和T.Welch发明的一种算法。它的应用也非常广泛，因为它的实现简单，而且也适用于多种类型的文件。

这种算法的基本思想和霍夫曼编码的基本思想相反。霍夫曼算法是为输入中的定长模式产生一张变长的编码编译表，但这种方法是为输入中的变长模式生成一张定长的编码编译表。这种方法的另一种令人惊讶的特性在于，和霍夫曼编码不同，输出中不需要附上这张编译表。

5.5.6.11 LZW压缩算法

为了说明这种算法的基本思想，先来看一个数据压缩的示例。假设需要读取一列由7位ASCII编码的字符组成的输入流并将它们写为一条8位字节的输出流。（在实际应用中使用的参数值一般都会更大——实现中使用的是8位的输入和12位的输出。）我们将输入字节称为字符，输入的字节序列称为字符串，输出字节称为编码，尽管这些术语在其他情况下的意义有所不同。LZW压缩算法的基础是维护一张字符串键和（定长）编码的编译表。在符号表中将128个单字符键的值初始化为8位编码，即在每个字符的7位值前添加一个0。为了简单明了，用十六进制数字来表示编码的值，这样ASCII的A的编码即为41，R的编码为52，等等。我们将80保留为文件结束的标志并将其余的编码值（81 ~ FF）分配给在输入中遇到的各种子字符串，即从81开始不断为新键赋予更大的编码值。为了压缩数据，只要输入还未结束，就会不断进行以下操作：

- 找出未处理的输入在符号表中最长的前缀字符串s；
- 输出s的8位值（编码）；
- 继续扫描s之后的一个字符c；
- 在符号表中将s+c（连接s和c）的值设为下一个编码值。

在后面的几步中，我们需要继续查看输入中的下一个字符才能构造字典中的下一个条目，因此将这个字符c称为前瞻（lookahead）字符。现在，当用尽了编码值（将FF赋予了某个字符串）之后暂时只能停止向符号表中添加新的条目——我们会在稍后讨论其他策略。

839

5.5.6.12 LZW压缩举例

下表所示的是LZW算法压缩样例输入A B R A C A D A B R A B R A B R A的详细过程。对于前7个字符，匹配的最长前缀仅为1个字符，因此输出这些字符所对应的编码，并将编码81到87和产生的7个两个字符的字符串相关联。然后我们发现AB匹配了输入的前缀（于是输出81并将ABR添加到符号表中），然后是RA（输出83并添加RAB），BR（输出82并添加BRA）和ABR（输出88并添加ABRA），最后只剩下A（输出41，请见图5.5.17）。



图5.5.17 LZW算法压缩A B R A C A D A B R A B R A B R A

输入为 17 个 7 位的 ASCII 字符，总共 119 位；输出为 12 个 8 位的编码，总共 96 位——压缩比为 82%，即使这只是个很小的例子。

5.5.6.13 LZW 的单词查找树

LZW 压缩算法含有两种符号表操作：

- 找到输入和符号表的所有键的最长前缀匹配；
- 将匹配的键和前瞻字符相连得到一个新键，将新键和下一个编码关联并添加到符号表中。

5.2 节中介绍的单词查找树数据结构完全是为这些操作量身定做的。对于上一个示例，它的单词查找树表示如图 5.5.18 所示。要查找最长前缀匹配，从根结点开始遍历树，按照结点的标签和输入字符匹配；在添加一个新编码时，先创建一个用新编码和前瞻字符标记的结点并将它和查找结束的结点相关联。在实践中，为了节省空间我们使用的是 5.2 节中介绍的三向单词查找树。值得一提的是这里对单词查找树的使用与霍夫曼编码的不同：对于霍夫曼编码，使用单词查找树是因为任意编码都不会是其他编码的前缀；但对于 LZW 算法，使用单词查找树是因为每个由输入字符串得到的键的前缀也都是符号表中的一个键。

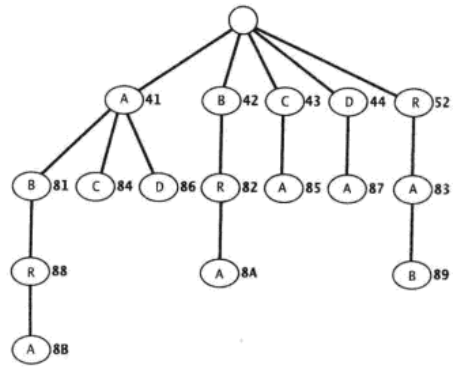


图 5.5.18 LZW 算法的编译表的单词查找树表示

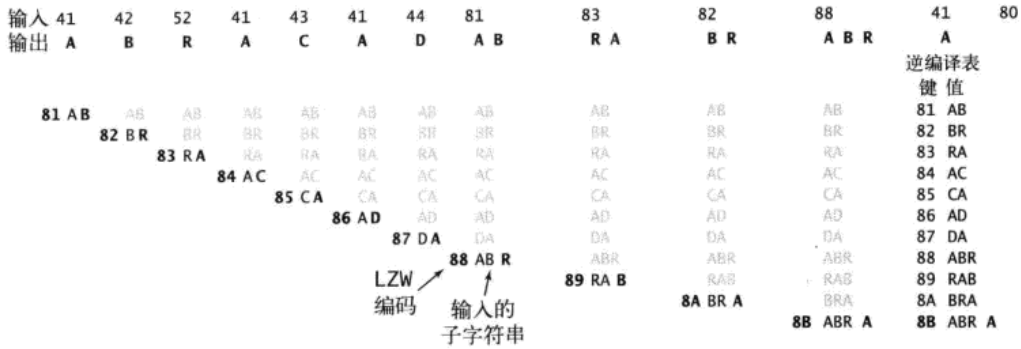
840

5.5.6.14 LZW 压缩的展开

如示例所示，LZW 压缩的展开所需的输入是一系列 8 位编码，而输出则是一个 7 位 ASCII 字符组成的字符串。在展开时，我们会维护一张关联字符串和编码值的符号表（这张表的逆表是压缩时所用的符号表）。在这张表中加入 00 到 7F 和所有单个 ASCII 字符的字符串的关联条目，将第一个未关联的编码值设为 81（80 保留为文件结尾的标记），将保存了当前字符串的变量 val 设为含有第一个字符的字符串，在遇到编码 80（文件结束）之前不断进行以下操作：

- 输出当前字符串 val；
- 从输入中读取一个编码 x；
- 在符号表中将 s 设为和 x 相关联的值；
- 在符号表中将下一个未分配的编码值设为 val+c，其中 c 为 s 的首字母；
- 将当前字符串 val 设为 s。

这个过程比压缩更加复杂，原因来自于前瞻字符：需要读取下一个编码来得到和它相关联的字符串的首字母，这使得整个过程不同步。对于前 7 个编码，只需要在符号表中查找并输出相应的字符，然后多读取一个字符并在符号表中添加一个两个字符的字符串的条目。这和之前是相同的。然后读到 81（输出 AB 并向符号表中添加 ABR），然后是 83（输出 RA 并添加 RAB），82（输出 BR 并添加 BRA），88（输出 ABR 并添加 ABRA），然后只剩下 41。最终会遇到文件结束的标记 80（因此输出 A）。这个过程结束后，就已经如期写出了原始的输入，并且构造了一张和压缩时相同的符号表（只是键和值的位置对调了，请见图 5.5.19）。注意，我们也可以使用一个简单的字符串数组来表示符号表，索引为编码。



841

图 5.5.19 LZW 算法对 41 42 52 41 43 41 44 81 83 82 88 41 80 的展开

算法 5.11 LZW 算法的压缩

```
public class LZW
{
   private static final int R = 256;        // 输入字符数
   private static final int L = 4096;       // 编码总数=2^12
   private static final int W = 12;         // 编码宽度

   public static void compress()
   {
      String input = BinaryStdIn.readString();
      TST<Integer> st = new TST<Integer>();

      for (int i = 0; i < R; i++)
          st.put( "" + (char) i, i);
      int code = R+1;  // R为文件结束(EOF)的编码

      while (input.length() > 0)
      {
         String s = st.longestPrefixOf(input); // 找到匹配的最长前缀
         BinaryStdOut.write(st.get(s), W);     // 打印出s的编码

         int t = s.length();
         if (t < input.length() && code < L)   // 将s加入符号表

             st.put(input.substring(0, t + 1), code++);
         input = input.substring(t);           // 从输入中读取s
      }

      BinaryStdOut.write(R, W);                // 写入文件结束标记
      BinaryStdOut.close();
   }

   public static void expand()
   // 请见算法5.11(续)
}
```

Lempel-Ziv-Welch 数据压缩算法的这份实现的输入为 8 位的字节流，输出为 12 位编码，适用于任意大小的文件。对于较小的样例输入，它所产生的编码和在正文中所讨论的类似：单字符的编码的开头为 0，其他编码从 100 开始。

```
% more abraLZW.txt
ABRACADABRABRABRA

% java LZW - < abraLZW.txt | java HexDump 20
04 10 42 05 20 41 04 30 41 04 41 01 10 31 02 10 80 41 10 00
160位
```

842

5.5.6.15 特殊情况

在刚才描述的过程中，存在这一个小小的问题。常常只有基于以上描述实现了这个过程的同学（以及有经验的程序员！）才能发现它。这个问题就是前瞻过程所得到的字符可能和当前子字符串的开头字符相同，如图 5.5.20 所示。在这个例子中，输入字符串：

ABABABA

如图 5.5.20 上方所示，被压缩得到的输出编码为：

41 42 81 83 80

在展开时，首先会得到编码 41 并输出 A，然后读取 42 得到前瞻字符并将 AB 和 81 插入符号表；输出 42 所对应的 B，读取 81 得到前瞻字符并将 BA 和 82 插入符号表；输出 81 所对应的 AB。到目前为止事情进展得不错。但当我们接下来取得了编码 83 并希望得到前瞻字符时，就被卡住了，因为读取编码所要补全的符号表条目正是 83！幸运的是，检查（只有在读取的编码和需要完成的编码条目相同时才会出现）并修正（此时，前瞻字符必然是当前字符串的首字母，因为它就是下个将被输出的字符）这种情况并不困难。在这个例子中，前瞻字符必然是 A（ABA 的首字母）。因此，下一个被输出的字符串和符号表中 83 到值都是 ABA。

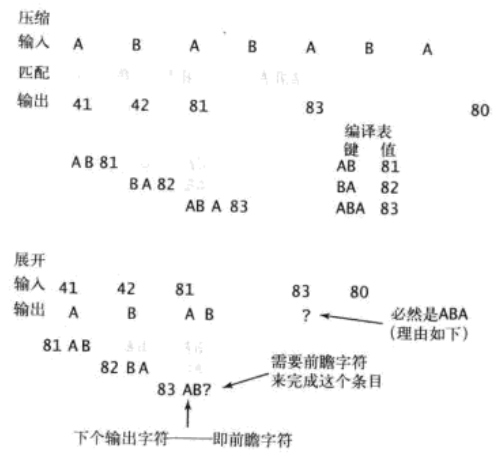


图 5.5.20 LZW 算法的扩展：特殊情况

5.5.6.16 实现

经过这些描述之后，实现 LZW 编码就很简单了，如算法 5.11 所示（expand() 方法的实现请见算法 5.11（续））。这段实现接受 8 位字节流作为输入（因此能压缩任意文件，而不仅仅是字符串），并产生 12 位编码的输出流（因此字典会非常大，压缩率也会更好）。这些值指定在（final 修饰的）实例变量 R、L 和 W 中。在 compress() 方法中使用了一棵三向单词查找树（请见 5.2 节）来表示编译表（利用单词查找树来支持高效的 longestPrefixOf() 操作），在 expand() 方法中使用了一个字符串数组来表示逆向编译表。这样，compress() 和 expand() 方法的代码就不完全与正文中的描述一一对应了。这些方法非常高效。对于某些文件，我们还可以通过在编译表满时将其清空并重用全部编码来改进它们。这些改进以及评估它们的性能所需的实验都留作本节最后的练习。

843

算法 5.11（续） LZW 算法的展开

```
public static void expand()
{
   String[] st = new String[L];

   int i; // 下一个待补全的编码值

   for (i = 0; i < R; i++)              // 用字符初始化编译表
      st[i] = "" + (char) i;
   st[i++] = " ";  // （未使用）文件结束标记(EOF)的前瞻字符

   int codeword = BinaryStdIn.readInt(W);
   String val = st[codeword];
   while (true)
   {
      BinaryStdOut.write(val);          // 输出当前子字符串
      codeword = BinaryStdIn.readInt(W);
      if (codeword == R) break;
      String s = st[codeword];          // 获取下一个编码
      if (i == codeword)                // 如果前瞻字符不可用
         s = val + val.charAt(0);       //    根据上一个字符串的首字母得到编码的字符串
      if (i < L)
         st[i++] = val + s.charAt(0); // 为编译表添加新的条目
      val = s;                          // 更新当前编码
   }

   BinaryStdOut.close();

}
```

这段代码实现了 Lempel-Ziv-Welch 算法的展开。展开比压缩更加复杂，因为需要从下一个编码中获取前瞻字符，并且存在前瞻字符可能不可用的复杂情况（请见正文）。

```
% java LZW - < abraLZW.txt | java LZW +
ABRACADABRABRABRA

% more ababLZW.txt
ABABABA

% java LZW - < ababLZW.txt | java LZW +
ABABABA
```

和以前一样，请花一点时间仔细研究程序和图 5.5.21 给出的 LZW 算法压缩的实例。十几年以来，它已经被证明为是一个多用途高效率的压缩算法。

病毒（50 000位）

```
% java Genome - < genomeVirus.txt | java PictureDump 512 25
```



```
12 536位

% java LZW - < genomeVirus.txt | java PictureDump 512 36
```



```
18 232位    ←— 效果不如双位编码，因为重复数据很少
```

位图（6144位）

```
% java RunLength - < q64x96.bin | java BinaryDump 0
2296位

% java LZW - < q64x96.bin | java BinaryDump 0
2824位      ←— 效果不如游程编码，因为文件太小
```

《双城计》全文（5 812 552位）

```
% java BinaryDump 0 < tale.txt
5 812 552位

% java Huffman - < tale.txt | java BinaryDump 0
3 043 928位

% java LZW - < tale.txt | java BinaryDump 0
2 667 952位  ←— 压缩率2667952/5812552 = 46%（已知最好成绩）
```

图 5.5.21 采用 12 位编码的 LZW 算法对各种文件的压缩和展开

844 ? 845

答疑

问 为什么需要 BinaryStdIn 和 BinaryStdOut？

答 这是在便利性和效率之间作出的一个平衡。StdIn 每次能够处理 8 位数据，而 BinaryStdIn 必须处理每一位数据。大多数应用程序处理的都是字节流，但数据压缩是个例外。

问 为什么需要 close() 方法？

答 有这个要求的是因为标准输出流是一个字节流，因此 BinaryStdOut 需要知道何时将最后一个字节对齐并输出。

问 能够将 StdIn 和 BinaryStdIn 混用吗？

答 最好不要这样。因为它们都和系统以及具体的实现有关，谁也不知道会出现什么情况。我们的实现会抛出一个异常。但从另一方面来说，混用 StdOut 和 BinaryStdOut 没有问题（我们的代码就这么使用的）。

问 为什么在 Huffman 类中 Node 类是静态的？

答 我们将所有数据压缩算法都组织成了静态方法的集合，而没有实现任何数据结构。

问 我能保证数据压缩算法至少不会将比特流还长吗？

答 你可以直接把输入复制到输出，但仍然需要某种标记来说明不需要使用任何标准的数据压缩方法就可以使用它。某些商业数据压缩程序有时会作出这种保证，但实际上这种保证很脆弱并且远远不具备通用性。事实上，大多数数据压缩算法甚至都做不到我们对命题 S 的第一种证明方法的第二步：极少有算法能够进一步压缩其自身产生的比特字符串。

846

练习

5.5.1 请看下表所示的4种变长编码。哪些编码是无前缀的？哪些编码的解码方式是唯一的？对于解码方式唯一的编码，请给出1000000000000的编码结果。

符号	编码1	编码2	编码3	编码4
A	0	0	1	1
B	100	1	01	01
C	10	00	001	001
D	11	11	0001	000

5.5.2 给出一个非前缀码但解码方式又是唯一的编码。

答：任意无后缀的编码都是解码方式唯一的编码。

5.5.3 给出一个即非前缀码又非后缀码且解码方式唯一的编码。

答：{0011, 011, 11, 1110} 或 {01, 10, 011, 110}

5.5.4 {01, 1001, 1011, 111, 1110} 和 {01, 1001, 1011, 111, 1110} 的解码方式是唯一的吗？如果不是，找出一条可以用两种方式解码的字符串。

5.5.5 使用 RunLength 处理本书网站上的文件 q128x192.bin。被压缩后的文件含有多少比特？

5.5.6 将 N 个符号 a 编码需要多少比特（作为 N 的函数）？N 个序列 abc 呢？

5.5.7 给出用游程编码、霍夫曼编码、LZW 编码压缩字符串 a,aa,aaa,aaaa,...（含有 N 个 a 的字符串）的结果，以 N 的函数表示压缩比。

5.5.8 给出用游程编码、霍夫曼编码、LZW 编码压缩字符串 ab,abab,ababab,abababab,...（将 ab 重复 N 次得到的字符串）的结果，以 N 的函数表示压缩比。

5.5.9 估计游程编码、霍夫曼编码和 LZW 编码处理长度为 N 的随机 ASCII 字符串（任意位置都有独立均等的几率出现任意字符）的压缩比。

5.5.10 按照正文中的示意图的样式显示使用 Huffman 处理字符串 it was the age of foolishness 时霍夫曼编码树的构造过程。压缩后的比特流需要多少比特？

847

5.5.11 如果所有字符均来自一个只有两个字符的字母表，该字符串的霍夫曼编码将会是什么？给出这样的一个长度为 N 的字符串，使得霍夫曼编码得到的结果最长。

5.5.12 假设所有符号出现的概率均为2的负若干次方，描述相应的霍夫曼编码。

5.5.13 假设所有符号出现的概率均相等，描述相应的霍夫曼编码。

5.5.14 假设需要编码的所有字符的出现频率均不相同。此时的霍夫曼编码树是唯一的吗？

5.5.15 只需扩展霍夫曼算法即可有效地将双位字符编码（使用四向树[①]）。这么做的主要优点和缺点是什么？

5.5.16 以下输入经过 LZW 编码后的结果是什么？

a. TOBEORNOTTOBE

b. YABBADABBADABBADOO

c. AAAAAAAAAAAAAAAAAAAA

① 每个结点都含有4条链接。——译者注

5.5.17 总结 LZW 编码中需要特别注意的情况。

解答：每当遇到形如 cScSc 的字符串时都会出现这种情况，其中 c 是一个符号而 S 是一个字符串，字典中已经含有 cS 但没有 cSc。

5.5.18 设 F_k 是第 k 个斐波那契数。假设有一个符号序列，其中第 k 个符号的频率为 F_k。注意，$F_1+F_2+\cdots+F_N=F_{N+2}-1$。给出相应的霍夫曼编码。提示：最长编码的长度为 $N-1$。

5.5.19 证明，对于给定的 N 个符号的集合，至少存在 2^{N-1} 种不同的霍夫曼编码。

5.5.20 给出一种霍夫曼编码，使得输出中的 0 的出现频率比 1 要高得多。

答：如果字符 A 出现了 100 万次而 B 只出现了一次，那么将 A 的编码设为 0，B 的编码设为 1 即可。 848

5.5.21 请证明在任意霍夫曼编码中，最长的两个编码的长度必然是相等的。

5.5.22 请证明霍夫曼编码的以下性质：如果符号 i 的出现频率大于符号 j，那么符号 i 的编码长度将会小于等于符号 j 的编码长度。

5.5.23 如果将用霍夫曼编码得到的字符串看作由 5 位字符组成的字符流并继续用霍夫曼编码处理它，结果将会是什么？

5.5.24 按照正文中示意图的样式显示使用 LZW 编码处理以下字符串时所构造的编码树以及整个压缩和展开的过程。

```
it was the best of times it was the worst of times
```

849

提高题

5.5.25 *定长定宽的编码*。实现一个使用定长编码的 RLE 类来压缩不同字符较少的 ASCII 字节流，将编码输出为比特流的一部分。在 `compress()` 方法用一个 `alpha` 字符串保存输入中所有不同的字母，用它得到一个 `Alphabet` 对象以供 `compress()` 方法使用。将 `alpha` 字符串（8 位编码再加上它的长度）添加到压缩后的比特流的开头。修改 `expand()` 方法，在展开之前先读取它的字母表。

5.5.26 *重建 LZW 字典*。修改 LZW 算法，当字典饱和时将其清空。这种方式适合某些应用程序，因为它能更好地适应输入中的字符变化。

5.5.27 *较长的重复*。估计游程编码、霍夫曼编码和 LZW 编码处理长度为 $2N$ 的一条字符串的压缩率，该字符串由长度为 N 的一条随机 ASCII 字符串（请见练习 5.5.9）重复而成。 850

第6章 背 景

在现代社会中，计算机设备无处不在。在过去的几十年中，我们世界中的电子设备还是一片空白，但现在它们已经成为数十亿人日常必备的工具。今天的手机甚至都比30年前只有少数人才有权使用的超级计算机强大若干个数量级。这些设备高效工作的背后都离不开算法，而其中的一些算法本书中也有所讨论。这是为什么呢？因为适者生存。可扩展的（线性的和线性对数级别的）算法是这个过程的核心并证明了高效算法的重要性。20世纪60年代和70年代的一些研究者用这些算法为我们的今天打下了基础。他们知道，可扩展的算法是未来的关键，而过去几十年的发展也证明了这一点。现在，基础设施已经完备，人们已经开始利用它们达到各种目的。正如B.Chazelle所说，20世纪是方程的世纪，但21世纪是算法的世纪。

本书中讨论的基础算法只是一个开始。当算法能够成为大学中的一门独立学科时，这一天就快要到来了（也许已经来了）。在商业应用、科学计算、工程、运筹学和其他无数有待人们探索的领域中，高效的算法都能使原来不可能解决的问题得到解决。本书的重点是学习重要而实用的算法。在本章中，我们会沿着这条路继续讨论几个示例，它们能够说明已经学过的一些算法在高级实践情景中的作用。（还包括一些学习算法的方法。）为了说明算法的影响范围，我们首先列出算法的几个重要的应用领域，然后详细讨论几个有代表性的示例并介绍算法的相关理论来说明应用的深度。不过对于这本大厚书来说，在最后涉及的这两个主题都是介绍性的，并不全面，实际生活中还有许多同样广泛的领域、同样重要的应用场景、同样有影响力的具体问题。

851
~
853

商业应用

互联网的出现加强了算法在商业应用软件中的核心地位。人们经常使用的所有应用都得益于我们已经学过的许多经典算法：

- 基础设施（操作系统、数据库、通信）；
- 应用程序（电子邮件、文档处理、数码照片）；
- 出版（书籍、杂志、网络内容）；
- 网络（无线网络、社交网络、互联网）；
- 交易处理（金融、零售、网络搜索）。

本章中将会讨论一个有代表性的示例，即B-树。它是为20世纪60年代的大型机发明的一种复杂的数据结构，但今天它仍然是现代数据库系统的基础结构。此外，还将讨论用于文本索引的后缀数组。

科学计算

自从冯·诺依曼在1950年发明了归并排序之后，算法在科学计算领域逐渐起到了重要的作用。今天的科学家需要处理大量的实验数据。他们在同时使用数学模型和计算模型来理解自然世界，包括：

- ❑ 数学计算（多项式、矩阵、微分方程）；
- ❑ 数据处理（实验结果和观测资料，特别是基因组学）；
- ❑ 计算模型和模拟。

这些任务都可能需要大量复杂的海量数据计算。在科学计算领域，本章中会详细讨论的一个经典示例就是事件驱动模拟问题。它的思想是维护一个复杂的真实世界的模型并根据时间控制模型中发生的变化。这种基础方法有着非常多的应用。此外还将讨论一个基因计算领域的基础数据处理问题。

工程学

现代工程学的基础是技术，而现代技术的基础是计算机。因此，算法能够发挥重要作用的方面包括：

- ❑ 数学计算和数据处理；
- ❑ 计算机辅助设计和生产；
- ❑ 基于算法的工程设计（网络、控制系统）；
- ❑ 图像和其他医学系统。

854

工程师和科学家使用的许多工具和方法都是相同的。例如，科学家用计算模型和模拟来理解自然世界；而工程师用计算模型和模拟来设计、建造并控制他们所制造的各种产品。

运筹学

运筹学领域的研究者和实践者开发了各种数学模型并用它们解决了许多问题，包括：

- ❑ 任务调度；
- ❑ 决策；
- ❑ 资源分配。

4.4节中的最短路径问题就是一个经典的运筹学问题。本章会再次讨论它并介绍最大流量问题。我们会展示规约的重要性并讨论它对于问题解决（problem-solving）的通用模型的影响，特别是对运筹学中核心的线性规划模型的影响。

算法在计算机科学的各个子领域中都有着重要的地位，它的应用领域包括，但绝对不局限于：

- ❑ 计算几何；
- ❑ 密码学；
- ❑ 数据库；
- ❑ 编程语言与系统；
- ❑ 人工智能。

在所有领域中，说明问题并找到有效算法和数据结构来解决问题都是非常重要的。我们已经学过的部分算法是可以直接使用的。更重要的是，本书的核心内容，也就是设计、实现和分析算法的一般方法在所有这些领域中都已经被成功地验证过。这种效应已经从计算机科学扩散到了许多其他领域，包括体育、音乐、语言学、金融、神经科学，等等。

我们现在已经学习了许多重要且实用的算法，那么理解它们之间的相互关系就变得很必要了。在本章的（也是本书的！）结尾我们会简要介绍计算理论，重点是不可解性（intractability）和P=NP?这个问题。它们仍然是理解实践中遇到的各种问题的关键。

855

6.0.1 事件驱动模拟

我们的第一个示例是一个基础的科学应用：按照弹性碰撞的原理模拟粒子系统的运动。科学家

通过这个系统可以理解和预测物理系统的性质。这个模型可以模拟气体中分子的运动、化学反应的动态过程、原子扩散、最密堆积问题（sphere packing）、行星的环的稳定性、某些元素的相变、一维自引力体系前向阵面传播技术等许多问题。它可应用的范围从分子运动中的微小亚原子粒子到天体物理学中巨大的星体对象。

讨论这个问题需要一些高中物理知识、一些软件工程的知识和一些算法知识。我们把大部分和物理有关的内容留作练习，而主要关注使用基础的算法工具（基于堆的优先队列），以处理它的一个实际应用，将不可能的计算变为可能。

6.0.1.1 刚性球体模型

首先介绍一个理想模型，它描述的是原子和分子在含有以下性质的容器中的运动：

- ❑ 运动的粒子与墙以及互相之间的碰撞是弹性的；
- ❑ 每个粒子都是一个已知位置、速度、质量和直径的球体；
- ❑ 不存在其他外力。

这个简单的模型在统计力学这个既与宏观现象（例如温度和压力）有关又与微观现象（例如单个原子和分子的运动）有关的学科中十分重要。麦克斯维尔和玻尔兹曼使用这个模型得到了由温度的函数表示的相互碰撞的分子的速度分布，爱因斯坦用这个模型解释了花粉颗粒在水中的布朗运动。不存在其他外力的假设意味着粒子在碰撞之前是在做匀速直线运动。我们也可以通过添加其他作用力来扩展这个模型。例如，如果加上摩擦力和自旋，那就可以更加准确地描述一些熟悉的物理运动，例如台球桌上的台球。

6.0.1.2 时间驱动模拟

我们的主要目标是维持这个模型，即希望能够记录所有粒子在任意时间内的位置和速度。为此，需要计算：在给定了时刻 t 时的所有粒子的位置和速度后，再给出 dt 时间之后，即未来的时间点 $t+dt$ 时它们的位置和速度。如果所有粒子互相之间以及和墙的距离都很远，那么计算就很简单了：因为粒子的轨迹是一条直线，所以只需要用粒子的速度就可以更新它的位置。这个问题的挑战在于要考虑碰撞情况。一种解决方法叫做时间驱动模拟（请见图 6.0.1），它基于使用固定长度的 dt。在每次更新时，我们都需要检查所有粒子对，判定它们是否可能相遇，然后还原它们的第一次碰撞。此时，我们将会更新两个粒子的速度以反映出碰撞的结果（计算方法会稍后讨论）。在粒子数量很多时，这种方式的计算量非常大：如果 dt 是以秒计（一般为一秒的若干分之一），它模拟 N 个粒子的系统一秒钟的运动所需的时间与 N^2/dt 成正比。这种成本太昂贵了（比平方级别的算法更高）——在一般的应用中，N 都会非常大而 dt 会非常小。dt 的问题在于如果它太小，计算量就太高，但如果它太大，那就可能错过许多次碰撞，请见图 6.0.2。

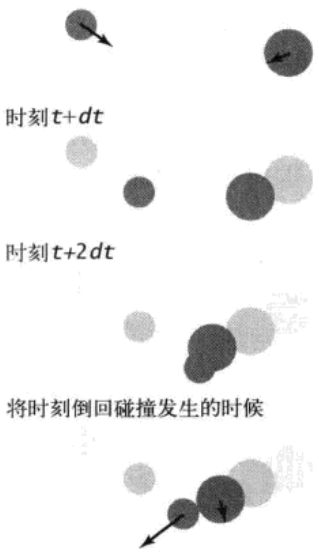


图 6.0.1 以时间作为驱动的模拟

6.0.1.3 事件驱动模拟

另一种方法是仅关注碰撞发生的时间点，重点关注下一次碰撞（因为在此之前由速度计算得到的所有粒子的位置都是有效的）。因此，我们可以使用一个优先队列来记录所有事件。事件是未来的某个时间的一次潜在的碰撞，可能发生在两个粒子之间，也可能发生在粒子和墙之间。和每个事

件相关联的优先级就是它发生的时间，因此当从优先队列中删去优先级最低的元素时，就会得到下一次潜在的碰撞。

6.0.1.4　碰撞预测

我们如何才能识别潜在的碰撞呢？粒子的速度正好提供了这个必要的信息。例如，假设在单位空间中，在时刻 t 有一个半径为 s 速度为 (v_x, v_y) 的粒子位于 (r_x, r_y)。假设墙位于 x=1 处，高度 y 在 0 到 1 之间。我们感兴趣的是运动的横向分量，因此注意力集中在位置的 x 分量 r_x 和速度的 x 分量 v_x 上。如果 v_x 是负数，那么粒子的轨迹不会与墙体相交，但如果 v_x 是正数，那就存在一个粒子和墙的潜在碰撞。将例子和墙的间距 $(1–s–r_x)$ 除以速度的 x 分量 (v_x)，就可以得到粒子和墙的碰撞时间为 $dt=(1–s–r_x)/v_x$ 个时间单位之后，此时粒子的位置将为 $(1–s, r_y+v_y dt)$，除非它在之前又撞上了其他某个粒子或者墙，请见图 6.0.3。因此，我们就可以向优先队列中插入一个优先级为 $t+dt$ 的条目（以及一些描述该示例和墙的碰撞事件的信息）。墙体的碰撞预测计算都是类似的（请见练习 6.1）。两个粒子之间的碰撞也是类似的，但更加复杂一些。不过你会注意到这种计算得到的预测结果通常是不会碰撞（比如粒子正在向墙体的反方向移动，或者两个粒子的运动方向相反）——这种情况下就不需要向优先队列中插入任何东西。为了处理另一种典型情况，也就是预测到的碰撞距现在的时间太远时，就需要一个 `limit` 参数来指定有效的时间段，这样就可以忽略时间晚于 `limit` 发生的所有事件了。

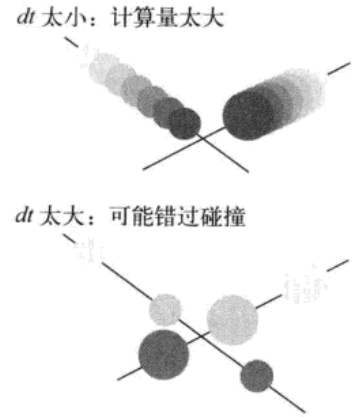


图 6.0.2　驱动模拟的主要问题

856 ~ 857

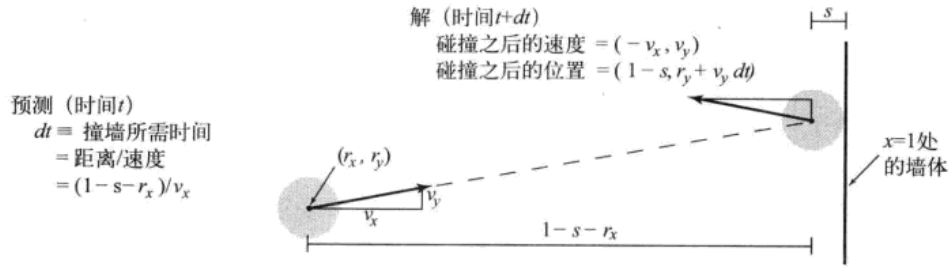


图 6.0.3　预测并解决粒子和墙体的一次碰撞

6.0.1.5　碰撞计算

当发生碰撞时，我们需要使用物理公式来进行计算，以描述一个粒子在和另一个粒子或者墙体发生刚性碰撞时的行为。在示例中，墙体遇到了一面竖墙。如果发生碰撞，粒子的速度将会从 (v_x, v_y) 变为 $(-v_x, v_y)$，请见图 6.0.4。其他墙体的碰撞和它类似。两个粒子的碰撞也是类似的，在物理上这是不严密的，但要更加复杂一些（请见练习 6.1）。

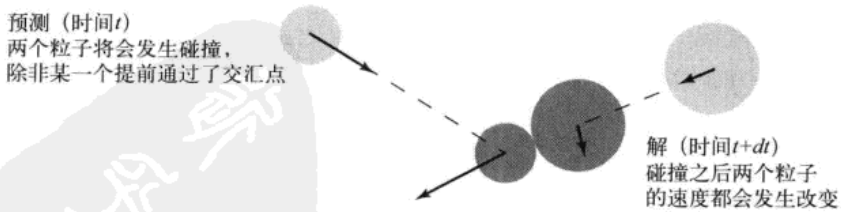


图 6.0.4　预测并计算粒子和墙体的一次碰撞

858

6.0.1.6 排除无效事件。

预测的许多碰撞实际上都不会发生，因为它们被其他的碰撞打断了，请见图 6.0.5。为了处理这种情况，我们为每个粒子维护一个实例变量来记录和它有关的碰撞数量。当从优先队列中取出一个事件来处理时，我们会检查该事件所涉及粒子的碰撞计数器在事件被创建后是否已经更新。这是排除无效碰撞的延时方法：当某个粒子参与了一次碰撞时，我们不会删除优先队列中和该粒子有关的其他碰撞（尽管这些碰撞事件现在都已经无效了），而是会在之后遇到它们时直接将其忽略请见图 6.0.6。另一种即时的方式是立刻从优先队列中删除所有与参与当前事件的粒子相关的其他事件，然后再计算这些粒子的新潜在碰撞事件。这种方式需要的优先队列更加复杂（需要实现删除操作，请见图 6.0.7）。

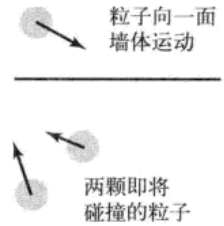


图 6.0.5 可预测的事件

以上讨论了一些预备知识，这些都是对按照物理定律进行弹性碰撞的运动粒子执行事件驱动模拟所必备的。相应的软件架构会将实现封装在 3 个类中：一个 `Particle` 数据类型，封装了所有和粒子有关的计算；一个 `Event` 数据类型来预测事件；一个它们的用例 `CollisionSystem` 类用来完成模拟。模拟的核心是一个含有所有事件的 `MinPQ` 优先队列，按照时间排序。下面看一下 `Particle`、`Event` 和 `CollisionSystem` 的实现。

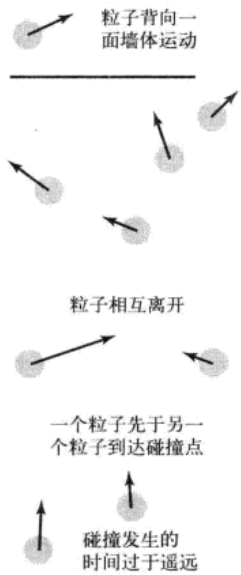


图 6.0.6 可预测的不可能发生的事件

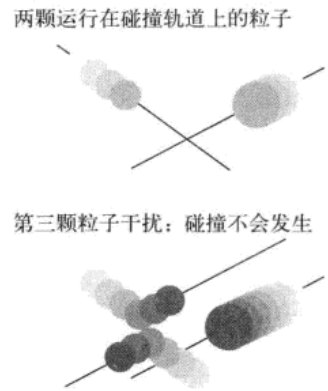


图 6.0.7 一次失效的事件

859

6.0.1.7 粒子

练习 6.1 基于牛顿的运动学定律给出了粒子数据类型的实现要点。模拟用例应该能够移动粒子、画出粒子并进行若干和碰撞相关的计算，如表 6.0.1 中的 API 所示。

表 6.0.1 运动的粒子对象的 API

public class **Particle**	
Particle()	在单位空间中创造一个新的随机粒子

（续）

public class **Particle**	
Particle(double rx, double ry, double vx, double vy, double s, double mass)	用给定的位置、速度、半径和质量创建一个粒子
void draw()	画出粒子
void move(double dt)	根据时间的流逝 dt 改变粒子的位置
int count()	该粒子所参与的碰撞总数
double timeToHit(Particle b)	距离该粒子和粒子 b 碰撞所需的时间
double timeToHitHorizontalWall()	距离该粒子和水平的墙体碰撞所需的时间
double timeToHitVerticalWall()	距离该粒子和垂直的墙体碰撞所需的时间
double bounceOff(Particle b)	碰撞后该粒子的速度
double bounceOffHorizontalWall()	碰撞水平墙体后该粒子的速度
double bounceOffVerticalWall()	碰撞垂直墙体后该粒子的速度

当粒子不在碰撞轨道上时（这是很常见的），3 个 timeToHit*() 的方法都会返回 Double.POSITIVE_INFINITY。这些方法可以帮助预测给定粒子在未来的所有碰撞，将在 limit 时间内发生的碰撞事件插入优先队列。在处理两颗粒子相撞的事件时，使用 bounceOff() 方法计算两颗粒子在碰撞之后的速度。bounceOff*() 方法用于处理粒子和墙体之间的碰撞事件。

6.0.1.8 事件

我们将应该放入优先队列中的所有对象信息封装在一个私有类之中（各种事件）。实例变量 time 记录的是事件的预计发生时间，实例变量 a 和 b 保存的是和该事件相关的粒子。这里有 3 种不同类型的事件：粒子和垂直墙体碰撞、粒子和水平墙体碰撞、粒子和粒子碰撞。为了平滑动态地显示运动中的粒子，我们添加了第 4 种类型的事件，即重绘事

```
private class Event implements Comparable<Event>
{
   private final double time;
   private final Particle a, b;
   private final int countA, countB;

   public Event(double t, Particle a, Particle b)
   {  // 创造一个发生在时间t且与a和b相关的新事件
      this.time = t;
      this.a    = a;
      this.b    = b;
      if (a != null) countA = a.count(); else countA = -1;
      if (b != null) countB = b.count(); else countB = -1;
   }

   public int compareTo(Event that)
   {
      if      (this.time < that.time) return -1;
      else if (this.time > that.time) return +1;
      else return 0;
   }

   public boolean isValid()
   {
      if (a != null && a.count() != countA) return false;
      if (b != null && b.count() != countB) return false;
      return true;
   }
}
```

粒子模拟的事件类

件。它的作用是将所有粒子在它们的当前位置画出。为了使 Event 的实现能够表示这 4 种类型的事件，允许粒子的值为空（null）：

- a 和 b 均不为空：粒子与粒子碰撞；
- a 非空而 b 为空：粒子 a 和垂直墙体的碰撞；
- a 为空而 b 非空：粒子 b 和水平墙体的碰撞；
- a 为空而 b 非空：重绘事件（画出所有粒子）。

860 ~ 861

尽管没有完全遵循面向对象编程的原则，但这些约定能够得到简洁的用例代码。它的实现如后面框注所示。

Event 类型实现中的第二个技巧是，它维护了两个实例变量 countA 和 countB，以记录事件创建时每个粒子所参与的碰撞事件数量。如果在将事件从优先队列中取出时该值没有发生变化，那么就可以继续模拟这个事件的发生。但如果在这个事件进入优先队列和离开优先队列的这段时间内任何计数器发生了变化，这个事件就失效了，那就可以忽略它。方法 isValid() 支持用例代码检查这种情况。

6.0.1.9 模拟器代码

有了封装在 Particle 类和 Event 类中的运算，实际模拟所需的代码非常少，如 CollisionSystem 的实现所示（请见框注“基于事件模拟互相碰撞的粒子（框架）”和框注“基于事件模拟互相碰撞的粒子（主循环）”）。大多数运算都封装在右侧框注所示的 predictCollision() 方法中。这个方法会计算与粒子 a 有关的所有潜在碰撞（可能是和另一个粒子，也可能是和一面墙）并将相应的事件加入优先队列中。

```
private void predictCollisions(Particle a, double limit)
{
   if (a == null) return;
   for (int i = 0; i < particles.length; i++)
   {  // 将与particles[i]发生碰撞的事件插入pq中
      double dt = a.timeToHit(particles[i]);
      if (t + dt <= limit)
         pq.insert(new Event(t + dt, a, particles[i]));
   }
   double dtX = a.timeToHitVerticalWall();
   if (t + dtX <= limit)
      pq.insert(new Event(t + dtX, a, null));
   double dtY = a.timeToHitHorizontalWall();
   if (t + dtY <= limit)
      pq.insert(new Event(t + dtY, null, a));
}
```

预测其他粒子的碰撞事件

模拟的核心是框注“基于事件模拟互相碰撞的粒子（主循环）”中的 simulate() 方法。我们会调用 predictCollision() 方法来初始化每个粒子，将所有粒子和墙体以及粒子和粒子之间的潜在碰撞加入优先队列中，然后进入事件驱动模拟的主循环，它的任务包括：

- 取出即将发生的事件（时间为 *t* 的优先级最小的事件）；
- 如果事件无效，将它忽略；
- 按照直线运动轨迹使所有粒子运动到时间 *t*；
- 更新所有参与碰撞的粒子速度；
- 使用 predictCollision() 方法来预测参与碰撞的粒子在未来可能发生的碰撞，并向优先队列中插入相应的事件。

这个模拟过程可以作为计算系统中的各种有趣性质的基础，如练习所示。例如，我们所感兴趣的一种基本性质是所有粒子向墙体所施加的压力。计算这种压力的一种方法是记录墙体和粒子碰撞的次数和动量（根据粒子的质量和速度计算这个值很简单），这样就很容易得到它们的总量。温度性质的计算也是类似的。

862

基于事件模拟互相碰撞的粒子（框架）

```
public class CollisionSystem
{
   private class Event implements Comparable<Event>
   {  /* 请见正文 *//  }

   private MinPQ<Event> pq;          // 优先队列
   private double t  = 0.0;          // 模拟时钟
   private Particle[] particles;     // 粒子数组

   public CollisionSystem(Particle[] particles)
   {  this.particles  = particles;  }

   private void predictCollisions(Particle a, double limit)
   {  /* 请见正文 */  }

   public void redraw(double limit, double Hz)
   { // 重绘事件：重新画出所有粒子
     StdDraw.clear();
     for(int i = 0; i < particles.length; i++) particles[i].draw();
     StdDraw.show(20);
     if (t < limit)
        pq.insert(new Event(t + 1.0 / Hz, null, null));
   }

   public void simulate(double limit, double Hz)
   {  /* 请见后面的主循环代码*/  }

   public static void main(String[] args)
   {
      StdDraw.show(0);
      int N = Integer.parseInt(args[0]);
      Particle[] particles = new Particle[N];

      for (int i = 0; i < N; i++)
         particles[i] = new Particle();
      CollisionSystem system = new CollisionSystem(particles);
      system.simulate(10000, 0.5);
   }
}
```

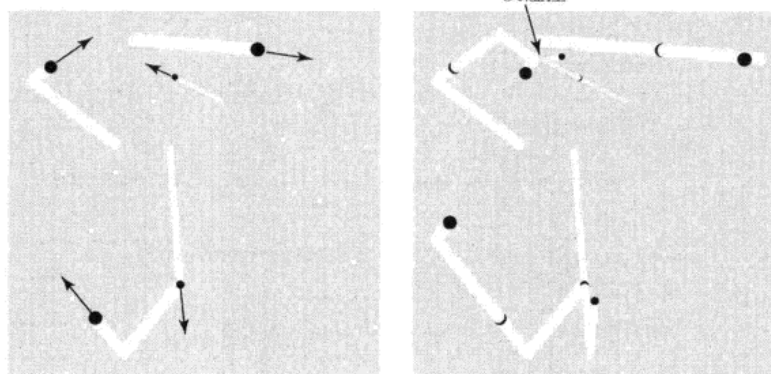
该类使用了优先队列来模拟粒子系统随着时间的运动。测试用例 `main()` 接受命令行参数 N，创造了 N 个随机粒子并创建了含有所有粒子的 `CollisionSystem`，然后调用 `simulate()` 方法模拟系统的演化。其中的实例变量分别保存了模拟所需的优先队列、当前时间和所有粒子。 863

基于事件模拟互相碰撞的粒子（主循环）

```
public void simulate(double limit, double Hz)
{
   pq = new MinPQ<Event>();
   for (int i = 0; i < particles.length; i++)
      predictCollisions(particles[i], limit);
   pq.insert(new Event(0, null, null));  // 添加重绘事件
   while (!pq.isEmpty())
   {  // 处理一个事件
```

```
            Event event = pq.delMin();
            if (!event.isValid()) continue;
            for (int i = 0; i < particles.length; i++)
               particles[i].move(event.time - t); // 更新粒子的位置
            t = event.time;                       //   和时间
            Particle a = event.a, b = event.b;
            if      (a != null && b != null) a.bounceOff(b);
            else if (a != null && b == null) a.bounceOffHorizontalWall();
            else if (a == null && b != null) b.bounceOffVerticalWall();
            else if (a == null && b == null) redraw(limit, Hz);
            predictCollisions(a, limit);
            predictCollisions(b, limit);
         }
      }
```

% java CollisonSystem 5



该方法是事件驱动模拟的主要部分。首先，我们用所有粒子预测的所有未来碰撞初始化优先队列。然后，主循环从队列中取出一个事件，更新时间和粒子的位置，并在处理碰撞后向队列中加入由此产生的所有新的潜在碰撞。

864

6.0.1.10 性能

如本小节的开头所述，我们对于事件驱动模拟的主要兴趣在于避免时间驱动模拟的内循环所必须的大量计算。

命题 A。 对 N 个能够相互碰撞的粒子系统，基于事件的模拟在初始化时最多需要 N^2 次优先队列操作，在碰撞时最多需要 N 次优先队列操作（且对于每个无效的事件都需要一次额外的操作）。

证明。 请见代码。

如果使用 2.4 节中优先队列的标准实现，我们能够保证优先队列的每次操作都是对数级别的，因此每次碰撞所需的时间是线性对数级别的。这样，才有可能模拟大量的粒子。

事件驱动模拟已经被应用于无数需要对运动中的物理对象建模的其他领域，例如分子学、天体物理学和机器人技术。这些应用可能会用其他实体，或是三维空间，或是其他作用力等许多种方法扩展这个模型。每种扩展都会为计算带来新的挑战。这种事件驱动的方式得到的模拟比其他方法更加健壮、准确和高效，而基于堆的优先队列的效率使不可能完成的计算成为了可能。

模拟在科学和工程的各个领域都是帮助研究者理解自然世界中各种性质的重要工具。它的应用

从制造业、生物学、金融领域到复杂的工程结构，数不胜数。对于它们其中的一大部分应用，基于堆的优先队列数据类型或是高效的排序算法能够使模拟的质量和范围大有改观。 865

6.0.2 B-树

在第3章中我们已经看到，能够快速访问大量数据中的特定元素的算法对于实际应用有着重要意义。例如在巨型数据集中，查找是一项非常重要的操作，该操作在许多计算场景中会消耗掉大部分资源。随着互联网的进步，某项任务访问到的信息可能非常庞大——我们的挑战在于在其中进行有效地查找。在本小节中，我们将介绍一种3.3节的平衡树算法的扩展。它支持对保存在磁盘或者网络上的符号表进行外部查找，这些文件可能比我们以前考虑的输入要大的多（以前的输入能够保存在内存中）。现代软件系统正在淡化本地文件和网页之间的区别，这些内容也可能保存在一台远程计算机上，因此我们可以找到的信息实际上近似于无限。令人惊讶的是，我们将要学习的算法只需使用4 ~ 5个指向一小块数据的引用即可有效支持在含有数百亿或者更多元素的符号表中进行查找和插入操作。

6.0.2.1 成本模型

数据存储的机制多种多样且在不断前进，因此我们将使用一个能够抓住本质的简单模型。这里用*页*表示一块连续的数据，用*探查*表示访问一个页。假设访问一页需要将它的内容读入本地内存，因此之后的访问就可以相对高效。一个页可能是本地计算机上的一个文件，也可能是远程计算机上的一张网页，也可能是服务器上的某个文件的一部分，等等。我们的目标是实现能够仅用极少次数的探查即可找到任意给定键的查找算法。我们不想假设页的具体大小或者一次探查（对于远程设备显然需要通信）所需时间与随后访问块中内容（显然这发生在本地处理器上）所需时间的比例。在一般情况下，这些值的数量级可能是100、1000或者10 000。我们不需要更精确的值，因为在我们感兴趣的范围内，算法对这些值的不同并不非常敏感。

B-树的成本模型。我们使用页的访问次数（无论读写）作为外部查找算法的成本模型。

6.0.2.2 B-树

它是对3.3节所述的2-3树数据结构的扩展。关键的不同在于：我们不会将数据保存在树中，而是会构造一棵由键的副本组成的树，每个副本都关联着一条链接。这种方式能够更加方便地将索引和符号表本身分开，就像一本实体书中的索引一样。和2-3树一样，我们限制了每个结点中能够含有的“键-链接”对的上下数量界限：选择一个参数 M（一般都是一个偶数）并构造一棵多向 866
树，每个结点最多含有 $M-1$ 对键和链接（假设 M 足够小，使得每个 M 向结点都能够存放在一个页中），最少含有 $M/2$ 对键和链接（以提供足够多的分支来保证查找路径较短）。根结点是个例外，它可以含有少于 $M/2$ 对键和链接，但也不能少于2对。这种树被Bayer和McCreight在1970年命名为B-树。他们是最早使用多向平衡树进行外部查找的研究者。有些人也用B-树这个术语来描述Bayer和McCreight发明的算法所构造的数据结构。本节用它泛指所有基于固定页大小的多向平衡查找树的数据结构。我们用 M 阶的B-树来指定 M 的值。在一棵4阶B-树中，每个结点都含有至少2对至多3对键-链接；在一棵6阶B-树中请见图6.0.8，每个结点都至少含有3对至多5对键-接（根结点除外，它可以只含有2对键与链接），等等。对于较大的 M 根结点是个例外的原因，在学习构造算法的细节时你就会明白了。

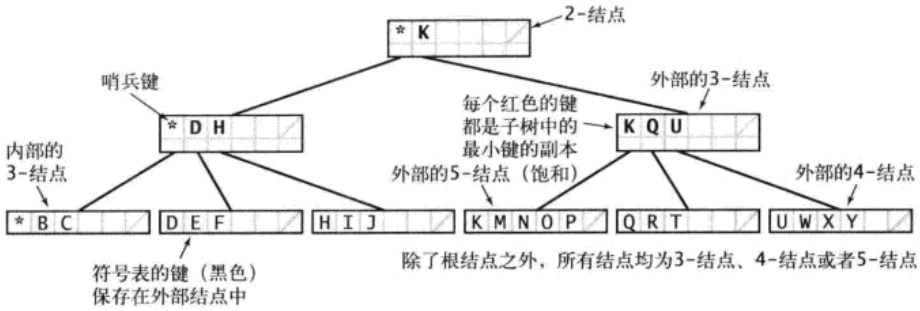


图 6.0.8 详解用一棵 B- 树表示的键集（M=6）

6.0.2.3 约定

为了说明基本的流程，我们先讨论（有序）（集合）SET 的一个实现（只有键没有值）。将它扩展得到一个能够将键和值相关联的符号表实现是一个很好的练习（请见练习 6.16）。我们的目标是为一个巨大的键集实现 `add()` 和 `contains()` 方法。使用有序集的原因是我们希望将查找树推广，而这依赖于键的有序性。扩展实现来支持其他有序性操作也是十分有益的练习。外部查找的应用常常会将索引和数据隔离。对于 B- 树，我们通过使用以下两种不同类型的结点做到这一点。

- 内部结点：含有与页相关联的键的副本。
- 外部结点：含有指向实际数据的引用。

内部结点中的每个键都与一个结点相关联，以此结点为根的子树中，所有的键都大于等于与此
867 结点关联的键，但小于原内部结点中更大的键（如果存在的话）。为了方便这里使用了一个特殊的哨兵键，它小于其他所有键。一开始 B- 树只含有一个根节点，而根结点在初始化时仅含有该哨兵键。符号表不含有重复键，但我们会（在内部结点中）使用键的多个副本来引导查找。（在示例中，所有键都是单个字母并使用小于所有字母的“*”作为哨兵键。）这些约定能够一定程度上简化代码，并且说明了另一种在内部结点中将所有数据和链接混合的便利（而且是广泛使用的）方式，就像其他查找树一样。

6.0.2.4 查找和插入

B- 树中查找的基础是在可能含有被查找键的唯一子树中进行递归搜索。当且仅当被查找的键包含在集合中时，每次查找便会结束于一个外部结点。在内部结点中遇到被查找的键的副本时就判断查找命中并结束，但总会找到相应的外部结点，因为这么做可以简化将 B- 树扩展为有序符号表的实现（当 M 很大时这种情况很少出现）。举一个具体的例子：假设有一棵 6 阶 B- 树，该树由多个含有 3 对键 – 链接的 3– 结点、含有 4 对键 – 链接的 4– 结点和含有 5 对键 – 链接的 5– 结点以及一个 2– 根结点组成，请见图 6.0.9。在查找时，从根结点开始，根据被查找的键选择当前结点中的适当区间并根据适当的链接从一个结点移动到下一个结点。最终，查找过程会到达树底的一个含有键的页。如果被查找的键在该页中，查找命中并结束；如果不在，则查找未命中。和 2–3 树一样，要在树的底部插入一个新键，可以使用递归代码。如果空间不足，那么可以允许被插入的结点暂时“溢出”（变成一个 6– 结点），并在递归调用后向上不断分裂 6– 结点。如果根结点也变成了 6– 结点，则可以将它分裂成连接了两个 3– 结点的 2– 结点；对于树的其他位置，我们将 6– 结点结点的父 k– 结点变为连接着两个 3– 结点的 (k+1)– 结点。将上文中的 3 替换成 M/2，6 替换成 M，即可得到 M 阶 B- 树中的查找和插入操作的方法，请见图 6.0.10。定义如下所示。

定义。一棵 M 阶 B- 树（M 为正偶数）或者仅是一个外部 k- 结点（含有 k 个键和相关信息的树），或者由若干内部 k– 结点（每个结点都含有 k 个键和 k 条链接，链接指向的子树表示了键之间的间隔区域）组成。它的结构性质如下：从根结点到每个外部结点的路径长度均相同（完美平衡）；对于根结点，k 在 2 到 M–1 之间，对于其他结点 k 在 M/2 到 M–1 之间。

868

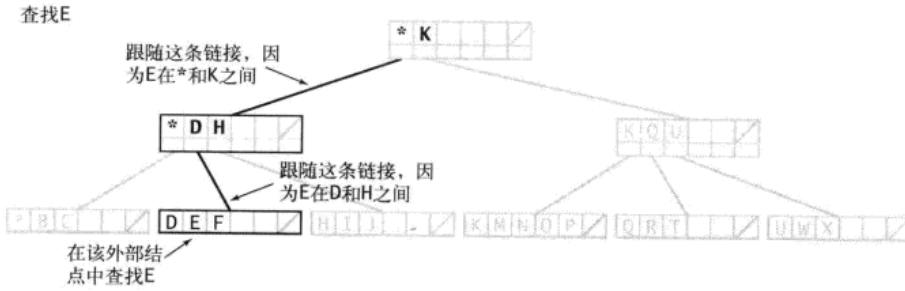


图 6.0.9　在由 B- 树表示的键集中进行查找（M=6）

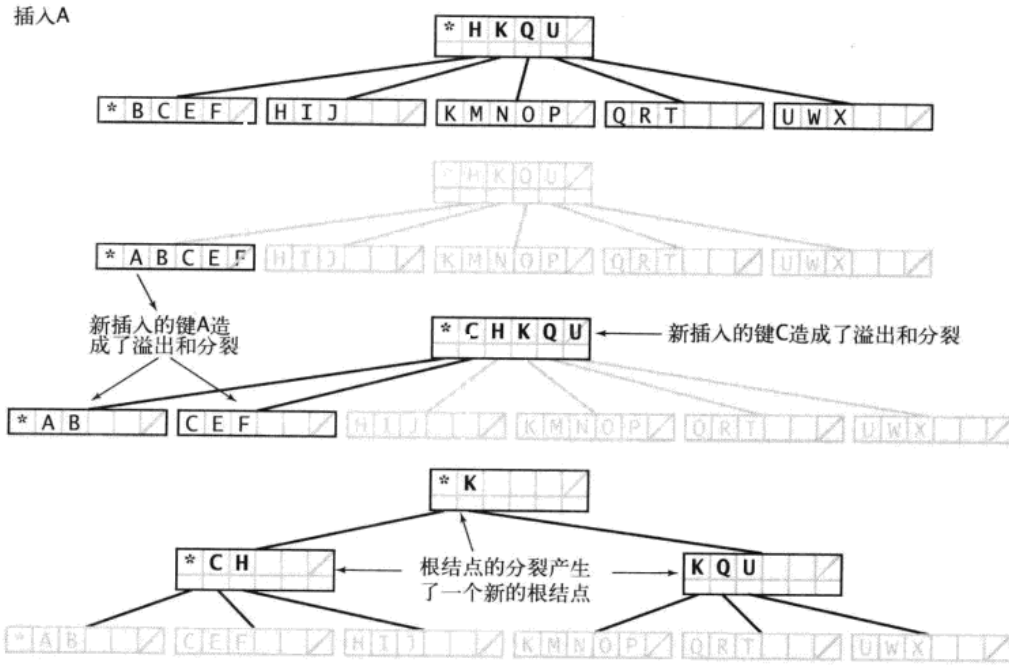


图 6.0.10　向由 B- 树表示的键集中插入一个新键

869

6.0.2.5　数据表示

按照刚才的讨论，我们在选择 B- 树结点的表示方法上有很大的自由度。我们将这些选择封装在一个 Page API 中（请见表 6.0.2）。它可以关联键与指向 Page 对象的链接，支持检测页是否溢出、分裂页并区分内部页和外部页的操作。你可以将 Page 看作一张符号表，但是是保存在外部介质上的（本地或是网络上的文件）。API 中的“打开”（open）和“关闭”（close）方法指的是将外部页读入内存和将内存内容写回外部页（如果需要的话）的过程。add() 方法是为内部页准备

的，它是一个符号表操作，会将给定页和以该页为根结点的子树中的最小键关联起来。add() 和 contains() 方法是为外部页准备的，和 SET 中相应的方法类似。在所有实现中，最重要的方法都是 split()。在分裂一张饱和页时，split() 方法会将排序后位置正好大于 $M/2$ 的键移动到一个新的 Page 对象中，并返回该对象的引用。练习 6.15 讨论了使用 BinarySearchST 对 Page 的一种实现。这种方法将 B-树实现在了内存中，和其他查找树的实现一样。在某些系统中，这种外部查找的实现可能已经足够了，因为虚拟内存系统会处理磁盘访问。更加贴近实际的实现可能包含与硬件相关的代码来读取和写入页的内容。练习 6.19 会鼓励你用网页实现 Page。这里不会讨论这些细节，而强调的重点是 B-树的概念能够广泛用于各种场景之中。

6.0.2 B- 树的页的 API

public class	Page<Key>	
	Page(boolean bottom)	创建并打开一个页
void	close()	关闭页
void	add(Key key)	将键插入（外部的）页中
void	add(Page p)	打开 p，向这个（内部）页中插入一个条目并将 p 和 p 中的最小键相关联
boolean	isExternal()	这是一个外部页吗
boolean	contains(Key key)	键 key 在页中吗
Page	next(Key key)	可能含有键 key 的子树
boolean	isFull()	页是否已经溢出
Page	split()	将较大的中间键移动到一个新页中
Iterable<Key>	keys()	页中所有键的迭代器

870

在这些准备之后，后面框注“B- 树集合的实现”的 BTreeSET 就很简单了。它用递归实现了 contains() 方法，接受一个 Page 对象作为参数并处理了以下 3 种情况。

- 如果当前页是外部页且键在该页中，返回 true。
- 如果当前页是外部页且键不在该页中，返回 false。
- 否则，递归地在可能含有该键的子树中查找。

我们用相同的递归结构实现了 add() 方法，只是在没有找到该键的时候将它插入到了树底部的页中，然后分裂回溯过程中所遇到的所有饱和结点，请见图 6.0.11。

6.0.2.6 性能

B- 树最重要的性质就是，在实际应用中对于适当的参数 M，查找的成本是常数级别的。

命题 B。含有 N 个元素的 M 阶 B- 树中的一次查找或插入操作需要 $\log_M N \sim \log_{M/2} N$ 次探查——在实际情况下这基本是一个常数。

证明。因为树中的所有内部结点（非根结点也非外部结点的所有结点）的形成都是由含有 M 个键的饱和结点分裂得到的且大小只可能增长（当它的子结点分裂时），所以其中的链接数总是在 $M/2$ 到 $M-1$ 之间。在最好的情况下，这些结点能够形成一棵 $M-1$ 向的完全树，由此马上就可以得到命题中所述的上下界。在最坏情况下，根结点只含有两个链接并分别指向两棵 $M/2$ 向的完全树。将对数的底设为 M 可以得到一个非常小的数——例如，当 M 为 1000 且 N 小于 625 亿时，树的高度小于 4。

在一般情况下，我们可以将根结点保存在内存中，这样可以将探查次数减 1。在磁盘和网络中进行查找时，应该在开始大量查找前显示地完成这一步。在带有缓存的虚拟内存中，应该将根结点放在最快的缓存中，因为它是访问最频繁的结点。

6.0.2.7 空间需求

在实际应用中，我们对 B- 树使用的空间也很感兴趣。由页的构造可知，它们至少都是半满的。在最坏的情况下，B- 树所需的空间是所有键占用的实际空间的一倍再加上链接所需的空间。对于随机键，A.Yao 在 1979 年（使用超出了本书范围的数学方法）证明了结点中平均含有 $M\ln 2$ 个键，因此浪费的空间约占 44%。和其他查找算法一样，这个随机模型也很好地预测了在实际应用中所观察到的键的分布。 871

算法 6.12 B- 树集合的实现

```
public class BTreeSET<Key extends Comparable<Key>>
{
   private Page root = new Page(true);

   public BTreeSET(Key sentinel)
   {  put(sentinel);  }

   public boolean contains(Key key)
   {  return contains(root, key);  }

   private boolean contains(Page h, Key key)
   {
      if (h.isExternal()) return h.contains(key);
      return contains(h.next(key), key);
   }

   public void add(Key key)
   {
      put (root, key);
      if (root.isFull())
      {
         Page lefthalf = root;
         Page righthalf = root.split();
         root = new Page(false);
         root.put(lefthalf);
         root.put(righthalf);
      }
   }

   public void add(Page h, Key key)
   {
      if (h.isExternal()) {  h.put(key); return;  }

      Page next = h.next(key);
      put(next, key);
      if (next.isFull())
         h.put(next.split());
      next.close();
   }
}
```

如正文所述，这段代码实现了多向平衡查找树（B- 树）。它在查找时使用了 Page 数据类型来将键和可能含有该键的子树相关联，并通过检测键的溢出和分裂结点的方法完成了插入操作，请见图 6.0.11。 872

命题 B 的影响之巨大，值得我们思考。你会猜到某种查找算法只需 4 ~ 5 次访问即可搜索你能够想象的最大文件吗？ B- 树的应用十分广泛，就是因为它能够实现这一点。在实践中，主要的挑战是在实现时尽量保证 B- 树中结点所需的空间，但随着大部分设备上的存储空间的增长，这已经不算什么问题了。

基本 B- 树抽象的许多变种都很容易理解。一类变化是尽可能在内部结点中保存更多的页引用以节省时间，这样可以使分支增多并将树更加扁平化。另一类变化是在分裂前将兄弟结点合并以提高存储的使用效率。对算法的变种以及参数的选择应该适应于具体的设备和应用。尽管提高的效率也仅限于常数因子的范围之内，但对于巨型符号表或是大量事物处理需求来说，这样的改进也有着重要的意义，这也是为什么 B- 树如此高效的原因。

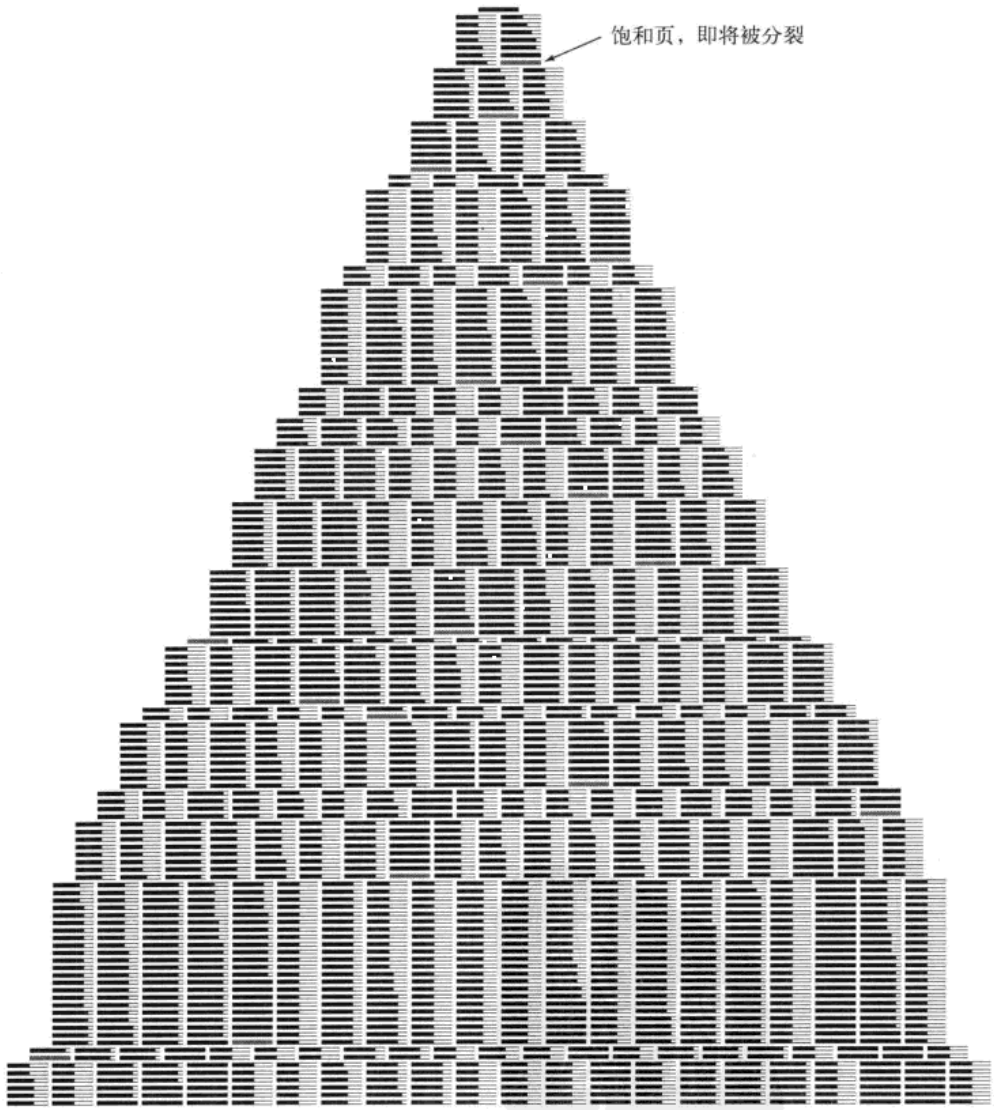


图 6.0.11 构造一棵庞大的 B- 树

6.0.3 后缀数组

字符串处理的高效算法在科学计算和商业应用中都有着重要的地位。从搜索互联网文本信息到科学家为了揭开生命的秘密而努力研究的庞大基因数据库，21 世纪中基于字符串的计算机应用在大规模增长。和以前一样，许多经典的算法都十分有效，但人们也发明了一些很好的新算法。下面，我们将介绍能够支持这些算法的一种数据结构和一份 API。首先，我们来看一个典型的（而且是经典的）字符串处理问题。

6.0.3.1 最长重复子字符串

在给定的字符串中，至少出现了两次的最长子字符串是什么？例如，在字符串 "to be or not to be" 中，最长重复子字符串就是 "to be"。你觉得应该怎样解决这个问题呢？你能在长度为数百万个字符的字符串中找出它的最长重复子字符串吗？这个问题的说明很简单，应用也很多，包括数据压缩、密码学和计算机辅助音乐分析等。例如，开发大型软件系统中的一种常见技术叫做*代码重构*。程序员经常会通过复制粘贴代码从原有的程序生成新的程序。对于开发了很长时间的一大段程序，将不断重复出现的代码转化为函数调用能够使程序更加容易理解和维护。我们可以通过在程序中寻找最长重复子字符串做到这一点。这个问题的另一个应用是计算生物学。在给定的基因中存在大量相同的片段吗？同样，这个问题背后的本质也是找出字符串中的最长重复子字符串。科学家一般更关心细节（事实上，重复子字符串的意义正是科学家所希望理解的），但这个问题显然比寻找简单的最长重复子字符串更难以回答。

6.0.3.2 暴力解法

作为热身，考虑以下这个简单的任务：给定两个字符串，找到它们的最长公共前缀（两者的前缀字符串中的相同且最长者）。例如，acctgttaac 和 accgttaa 的最长公共前缀是 acc。右边框注中的代码是我们解决更加复杂问题的起点：它所需的时间和相匹配的子字符串长度成正比。现在，我们应该如何在给定的字符串中找到最长重复子字符串呢？根据 lcp()，马上可以得到下面这种暴力解法：将一个字符串中起始位置为 i 的子字符串与另一个字符串中起始位置为 j 的子字符串相比较，记录匹配的最长子字符串。这段代码不适合处理长字符串，因为它的运行时间至少是字符串长度的*平方*级别：不同的子字符串对 i 和 j 的数量为 $N(N-1)/2$，因此这种方式调用 lcp() 的次数将会是~ $N^2/2$。用这种方法处理含有上百万个字符的碱基对序列将会调用几百亿次 lcp()，显然这是不可行的。

```
private static int lcp(String s, String t)
{
   int N = Math.min(s.length(), t.length());
   for (int i = 0; i < N; i++)
      if (s.charAt(i) != t.charAt(i)) return i;
   return N;
}
```

两个字符串的最长公共前缀

875

6.0.3.3 后缀排序

下面这种巧妙的方法用一种出人意料的方式利用排序算法高效地找出了字符串中的最长重复子字符串：用 Java 的 substring() 方法创建一个由字符串 s 的所有后缀字符串（由字符串的所有位置开始得到的后缀字符串）组成的数组，然后将该数组排序，请见图 6.0.12。算法的关键在于原字符串的每个子字符串都是数组中的某个后缀字符串的前缀。在排序之后，最长重复子字符串会出现在数组中的相邻位置。因此，只需要遍历排序后的数组一遍即可在相邻元素中找到最长的公共前缀。这种方法比暴力方法有效得多。但在实现和分析它之前，我们先介绍

后缀排序的另一种应用。

6.0.3.4 定位字符串

当需要在大量文本中寻找某个特定的子字符串时（例如，当你在使用文本编辑器或是在浏览网页时），你就是在进行一次子字符串查找，即5.3节中讨论过的问题。对于这个问题，我们假设文本比要查找的字符串庞大得多，并将注意力集中在查找字符串的预处理上，以保证能够在任意给定的文本中高效的找到该子字符串。当在浏览器中输入要查找的关键字时，就是在进行一次字符串键查找，即5.2节的主题。搜索引擎必然已经预先计算得到了一张索引表，因为它不可能即时地根据输入的关键字扫描互联网中的所有页面。根据3.5节的讨论（请见3.5.4节框注“文件索引”的FileIndex），理想情况下最好有一张反向索引符号表将每个被查找的字符串和所有含有它的网页关联起来——在符号表的每个条目中，键即为被查找的字符串，而值则为一组指针，请见图6.0.13（每个指针都含有能够定位该键在互联网上具体位置所需的信息——这可以是一个网页的URL加上键的出现位置的偏移量。）在实际应用中，这样的符号表会非常非常大，因此搜索引擎会使用各种复杂的算法来缩小它的体积。一种方法是将网页按照重要程度排序（可以使用3.5.5节讨论的PageRank算法）并只选择排序等级较高的网页而非全部网页。另一种减小符号表大小的方法是将多个关键词（以空格分隔）作为预处理得到的索引表的键并和URL关联。那么，当你查找一个关键词时，搜索引擎可以通过索引找到含有被查找的键（即关键词）的（相对重要的）网页，并在该页面中使用字符串查找来定位关键词。使用这种方法时，如果文本含有的是“everything”而你要找的是“thing”，那可能会找不到。对于某些应用，构造一个能够帮助我们找出文本中的任意子字符串的索引是值得的。这么做可能是为了对一本非常重要的文学作品进行语言学研究，或是为了找出可能成为许多科学家研究对象的某段碱基对序列，或者找出访问量很大的网页。同样，在理想情况下，索引表应该将文本字符串的所有子字符串分别和它们的出现位置关联起来，如图6.0.14所示。这种方法的问题显然是子字符串的总数太大，在符号表中为每个子字符串创建一个条目不现实。（一段含有N个字符的文本含有$N(N+1)/2$个子字符串。）图6.0.14中的符号表需要含有b、be、bes、best、best o、best of、e、es、est、est o、est of、s、st、st o、st of、t、t o、t of、o、of和许许多多其他子字符串的条目。这次我们也可以用后缀排序的方法解决这个问题，就像3.1节中用二分查找对符号表的第一次实现一样。我们可以将N个后缀作为键，以这些键（后缀）创建一个有序的数组并使用二分查找法搜索数组，比较被查找的键和所有后缀，请见图6.0.15。

```
输入字符串
   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
   a a c a a g t t t a c a a g c

所有后缀字符串
 0 aacaagtttacaagc
 1 acaagtttacaagc
 2 caagtttacaagc
 3 aagtttacaagc
 4 agtttacaagc
 5 gtttacaagc
 6 tttacaagc
 7 ttacaagc
 8 tacaagc
 9 acaagc
10 caagc
11 aagc
12 agc
13 gc
14 c

排序后的后缀字符串
 0 aacaagtttacaagc
11 aagc
 3 aagtttacaagc
 9 acaagc
 1 acaagtttacaagc
12 agc
 4 agtttacaagc
14 c
10 caagc
 2 caagtttacaagc
13 gc
 5 gtttacaagc
 8 tacaagc
 7 ttacaagc
 6 tttacaagc

最长重复子字符串
     1          9
   a acaag   ttt acaag   c
```

图6.0.12 使用后缀排序计算最长重复子字符串

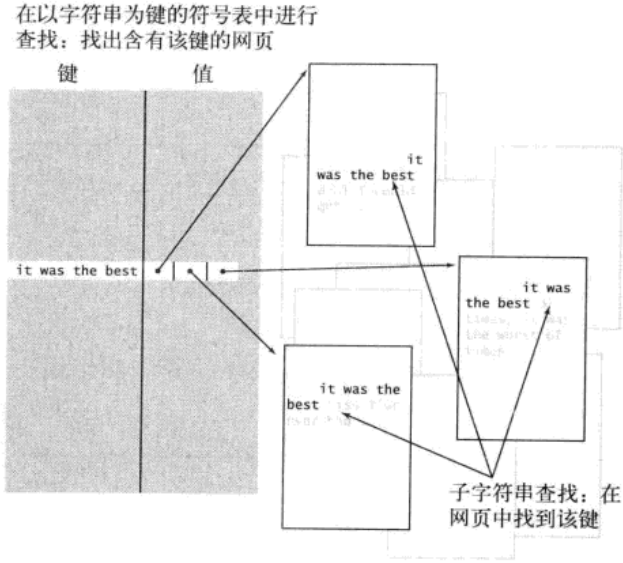


图 6.0.13　理想化的一次典型的网络搜索

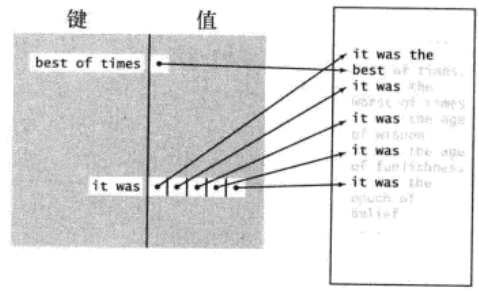


图 6.0.14　理想化的一张文本字符串索引表



图 6.0.15　后缀数组中的二分查找

876
~
878

6.0.3.5　API 及其用例

为了解决这两个问题，我们给出了以下 API。它含有构造函数、length() 方法、select() 和 index() 方法分别给出了有序后缀数组中给定位置的后缀和它的索引值、lcp() 方法会返回每个后缀和它在数组中的前一个后缀的最长公共前缀、rank() 方法能够给出小于给定键的后缀数量。（自

从第1章中第一次学习二分查找后就一直在使用它。）我们用*后缀数组*表示有序后缀字符串列表的这种抽象数据结构，但实际使用的并不一定是字符串数组，如表6.0.3所示。

表6.0.3 后缀数组的API

public class	SuffixArray	
	SuffixArray(String text)	为文本text构造后缀数组
int	length()	文本text的长度
String	select(int i)	后缀数组中的第i个元素（i在0到*N*–1之间）
int	index(int i)	select(i)的索引（i在0到*N*–1之间）
int	lcp(int i)	select(i)和select(i–1)的最长公共前缀的长度（i在1到*N*–1之间）
int	rank(String key)	小于键key的后缀数量

在右边框注所示的例子中，select(9)的结果是“as the best of times...”、index(9)的值是4、lcp(20)的值是10（因为“it was the best of times...”和“it was the”的公共前缀“it was the”的长度为10）、rank(“th”)的值是30。注意，select(rank(key))是有序后缀数组中第一个以key为前缀的后缀字符串，键key在正文中出现的其他位置都在后缀数组中紧跟着该条目（请见图6.0.15）。使用这份API可以立即写出框注中的代码。LRS类（见本页框注）会为标准输入得到的文本构造后缀数组，并根据扫描数组所得的最大lcp()值找出文本中的最长重复子字符串。KWIC类（见下页框注）会为命令行参数指定的文本构造后缀数组，从标准输入接受查询并打印出被查询的子字符串在文本中的上下文（该字符串的前后若干个字符）。KWIC这个名字表示的是上下文中的关键词（keyword-in-context）查找，最早出现在20世纪60年代。这些典型的字符串处理应用代码的简洁和高效令人赞叹。这也说明了精心设计API的重要性（以及简单而巧妙的思想的影响力）。

```
public class LRS
{
   public static void main(String[] args)
   {
      String text = StdIn.readAll();
      int N = text.length();
      SuffixArray sa = new SuffixArray(text);
      String lrs = "";
      for (int i = 1; i < N; i++)
      {
         int length = sa.lcp(i);
         if (length > lrs.length())
            lrs = sa.select(i).substring(0, length);
      }
      StdOut.println(lrs);
   }
}
```

最长重复子字符串算法的用例

```
% 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

% java LRS < tinyTale.txt
st of times it was the
```

879
↓
880

```
public class KWIC
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int context = Integer.parseInt(args[1]);

      String text = in.readAll().replaceAll("\\s+", " ");;
      int N = text.length();
      SuffixArray sa = new SuffixArray(text);

      while (StdIn.hasNextLine())
      {
         String q = StdIn.readLine();
         for (int i = sa.rank(q); i < N && sa.select(i).startsWith(q); i++)
         {
            int from = Math.max(0, sa.index(i) - context);
            int to   = Math.min(N-1, from + q.length() + 2*context);
            StdOut.println(text.substring(from, to));
         }
         StdOut.println();
      }
   }
}
```

上下文中的关键词的索引用例

```
% java KWIC tale.txt 15
search
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
 dispersing in search of other carri
n that bed and search the straw hold


better thing
t is a far far better thing that i do than
 some sense of better things else forgotte
was capable of better things mr carton ent
```

881

6.0.3.6 实现

算法 6.13 中的代码简洁明了地实现了 SuffixArry 的 API。它的实例变量包括一个字符串数组和（为了节省代码）一个表示数组长度的的变量 N（既是字符串的长度也是它的后缀字符串数量）。类的构造函数会构造后缀数组并将它排序，因此 `select(i)` 只需返回 `suffixes[i]` 即可。`index()` 的实现也只要一行代码，但稍微复杂一点，因为后缀字符串的长度就说明了它的起始位置。长度为 N 的后缀字符串的起始位置为 0，长度为 N–1 的后缀字符串的起始位置为 1，长度为 N–2 的后缀字符串的起始位置为 2，依此类推。因此 `index(i)` 的返回值即为 N-`suffixes[i].length()`。由 6.0.3.2 节中的静态 `lcp()` 方法可以很容易得到这里的 `lcp()` 方法的实现，`rank()` 方法与 3.1.5 节“算法 3.2（续）”中基于二分查找的符号表的实现也基本相同。同样，实现的简洁与优雅并不能掩盖这是一种复杂的算法，它解决了如最长重复子字符串这种其他方法无法解决的重要问题。

6.0.3.7 性能

后缀排序算法的效率取决于 Java 的子字符串提取操作使用的内存空间，它是一个常数 —— 每个子字符串都是由标准对象、指向原字符串的指针和它的长度组成的。因此，索引的大小和字符串的长度是线性关系。这让人有些意外，因为所有子字符串中的字符总数为~ $N^2/2$，即字符串长度的平方级别。另外，这种平方级别的性能也会大大影响子字符串数组的排序成本。我们要记住的重要一点是，这种方法对长字符串有效的原因在于 Java 的字符串表示方法：当交换两个字符串时，实际交换的仅仅是对它们的引用，而非字符串本身。虽然当两个字符串有很长的公共前缀时比较它们的成本与它们的长度成正比，但在一般的应用场景下，大多数比较都只需要检查几个字符。如果是这样的话，后缀数组的排序时间就是线性对数的。例如，在许多应用中，随机字符串模型都是合理的。

命题 C。使用三向字符串快速排序，构造长度为 N 的随机字符串的后缀数组，平均所需的空间与 N 成正比，字符比较次数与 ~ $2N\ln N$ 成正比。

讨论。后缀数组的空间需求很明显，但它所需的时间来自于 P.Jaquet 和 W.Szpankowski 的一份艰深而复杂的研究成果。他们证明了将所有后缀排序的成本渐进于将 N 个随机字符串排序的成
882 本（请见 5.1.4.4 节中的命题 E）。

算法 6.13 后缀数组（初级实现）

```
public class SuffixArray
{
   private final String[] suffixes;  // 后缀数组
   private final int N;              // 字符串（和数组）的长度

   public SuffixArray(String s)
   {
      N = s.length();
      suffixes = new String[N];
      for (int i = 0; i < N; i++)
         suffixes[i] = s.substring(i);
      Quick3way.sort(suffixes);
   }

   public int length()          { return N; }
   public String select(int i) { return suffixes[i]; }
   public int index(int i)      { return N - suffixes[i].length(); }

   // 请见6.0.3.2节框注“两个字符串的最长公共前缀”
   public int lcp(int i)
   {  return lcp(suffixes[i], suffixes[i-1]); }
   public int rank(String key)
   {  // 二分查找
      int lo = 0, hi = N - 1;
      while (lo <= hi)
      {
         int mid = lo + (hi - lo) / 2;
         int cmp = key.compareTo(suffixes[mid]);
         if      (cmp < 0) hi = mid - 1;
         else if (cmp > 0) lo = mid + 1;
         else return mid;
      }
```

```
        return lo;
    }
}
```

SuffixArray API 的实现效率取决于 Java 的 String 类的不可改变性，这种性质使得子字符串实际上都是引用，提取子字符串只需常数时间（请见正文）。

883

6.0.3.8　改进的实现

SuffixArray 的初级实现在最坏情况下的性能很糟。例如，如果所有的字符都相同，后缀数组的排序会检查每个后缀字符串中的每个字符，所需的时间为平方级别。对于我们用作示例的碱基对序列字符串或是自然语言的文本字符串，这可能不是问题，但算法对于含有一大串相同字符的文本可能会很慢。此外，查找最长重复子字符串所需的时间可能会是子字符串长度的平方级别，因为重复的子字符串的所有前缀都会被检查（请见图 6.0.16）。对于《双城记》来说这不是问题，因为其中最长的重复子字符串为：

```
"s dropped because it would have
 been a bad thing for me in a
 worldly point of view i"
```

只有 84 个字符。然而，对于经常含有很长的重复部分的碱基对序列来说，这就是一个严重的问题了。如何避免查找重复子字符串时出现的这种平方级别运算呢？幸运的是，P.Weiner 在 1973 年的研究显示我们可以保证在线性时间内解决最长重复子字符串问题。Weiner 算法的基础是构造一棵后缀字符串树（即一棵由所有后缀字符串组成的字典查找树）。如果在每个字符处使用多个链接，后缀树在解决许多实际问题时会消耗非常大的空间，这又推动了后缀数组的发展。在 20 世纪 90 年代，U.Manber 和 E.Myers 演示了一种构造后缀数组的线性对数级别的算法，以及一个同时完成预处理和对后缀数组排序以支持常数时间的 lcp() 方法。之后人们又发明了若干线性时间的后缀排序算法。经过一些改造，Manber-Myers 算法的实现也能够支持两个参数的 lcp() 方法，以在常数时间内找出给定的但不一定是相邻的两个后缀之间的最长公共前缀。这也是对初级实现的一项重大改进。这些结果非常令人惊讶，因为它们所达到的效率远远超出了人们的预期。

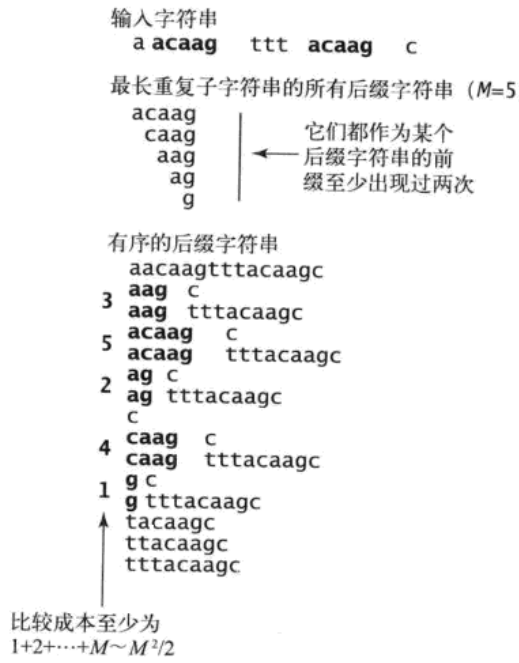


图 6.0.16　查找最长重复子字符串的成本是重复子字符串长度的平方级别

884

命题 D。使用后缀数组，我们可以在线性时间内解决后缀排序和最长重复子字符串问题。

证明。解决这些问题的优美算法已经超出了本书的范畴，但你在本书的网站上可以找到线性时间的 SuffixArray 的构造函数和常数时间的 lcp() 方法的实现。

885

基于这些思想的 SuffixArray 实现足以高效解决许多字符串处理问题，而且用例代码非常简单，如我们的 LRS 和 KWIC 例子所示。

后缀数组是自 20 世纪 60 年代解决 KWIC 索引的单词查找树以来数十年研究积累的成果。我们讨论的很多种算法都是许多研究者在几十年的实践中发明的，这些问题包括将《牛津英语大词典》搬上互联网、第一代搜索引擎以及人类基因组测序，等等。这完全说明了算法的设计和分析的重要性。

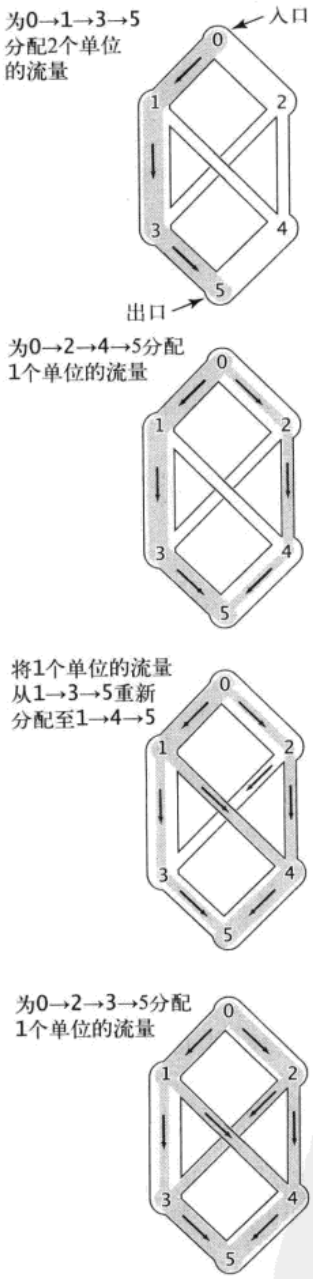


图 6.0.17 为输油网络分配流量

6.0.4 网络流算法

下面我们将讨论一种图的模型，它的成功之处不仅在于为我们提供了能够轻松描述解决实际问题的模型，而且使用这些模型我们能得到许多高效的算法来解决问题。我们将要讨论的解决方案说明了两种特定需求之间的矛盾，即具有广泛适用性的需求与能够解决特殊问题的需求。网络流算法研究的迷人之处在于它紧凑优雅的实现几乎能够同时达到这两个目标。你将会看到，我们的实现非常易懂而且能够保证运行时间与网络大小成正比。

网络流问题的经典解决方案和第 4 章中介绍的那些图算法紧密相关。基于已有的工具，我们可以编写非常精炼的程序来解决它们。我们已经在许多问题中看到，良好的算法和数据结构能够大幅减少解决问题所需的时间。人们还在积极研究该领域中更好的算法和数据结构并不断地发明新的方法。

6.0.4.1 物理模型

首先用一个理想化的物理模型来介绍几个直观的概念。请想象一组相互连接大小不一的输油管道，在连接处装有能够控制原油流向的开关，如图 6.0.17 所示。

我们还假设这个输油网只有一个入口（比如一处油田）和一个出口（比如一个大型的炼油厂），所有的输油管最终都会和它们相连。在每个结点处，原油流入量和流出量都会达到的平衡。我们用相同的单位衡量流量和管道的输送能力（例如，加仑每秒）。如果在每个开关处都有流入管道的总流量和流出管道的总流量相等，那么问题就不存在了：只需要将所有输油管充满即可。否则，虽然并不是所有管道都是饱和的，但原油仍然会根据各个关节处的开关设置在网络中流动，并将在关节处满足一个*局部平衡条件*：流入结点的流量等于流出结点的流量，请见图 6.0.18。

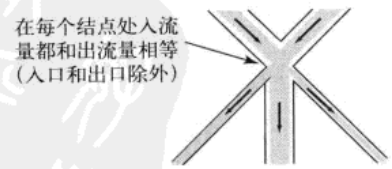


图 6.0.18 流量网络中的局部平衡

例如，如图 6.0.17 所示，一开始操作员可能会将原油的路径设为 0→1→3→5，这条路线能够输送 2 个单位的流量，然后再打开 0→2→4→5 这条路径上的开关，又可以输送 1 个单位的流量。因为 0→1、2→4 和 3→5 都已经饱和，已经无法直接将更多的原油从 0 输送到 5。但如果调整 1 处的开关将 1→4 充满，那么就又可以在 3→5 空出足够的空间使得 0→2→3→5 可以再增加 1 个单位的流量。即使是这样一个简单的网络，找到能够使得流量最大化的开关配置也并不容易；而对于更加复杂的网络，我们感兴趣的显然是下面这个问题：怎样配置所有开关才能使从入口到出口的流量最大化？我们可以直接用只含有一个起点和一个终点的加权有向图构造出这个问题的模型。图中的边对应的是输油管道，顶点对应的是配有能够控制原油走向和流量的开关结点，边的权重对应的是管道的容量，请见图 6.0.19。我们假设边是有向的，即原油在每个管道中都只能朝着一个方向流动。每条管道中都流动着一定量的原油，流量小于等于管道的容量，而每个顶点都需要满足流入量和流出量相等。这种抽象的流量网络是一个能够解决问题的实用模型，它能够直接应用于许多场景，而间接适用的则更多。我们有时会用原油流过管道的方式直观地说明一些基本的概念，但这里的讨论同样适用于物流分配的通道等情况。鉴于我们在各种最短路径算法中对“距离”概念的用法，在必要的时候会抛弃图的所有物理意义，因为我们讨论的所有定义、性质和算法所基于的抽象模型并不一定遵守物理定律。事实上，人们对网络流问题的主要兴趣在于许多其他问题都能转化为这个模型，下一个小节中将会详述。

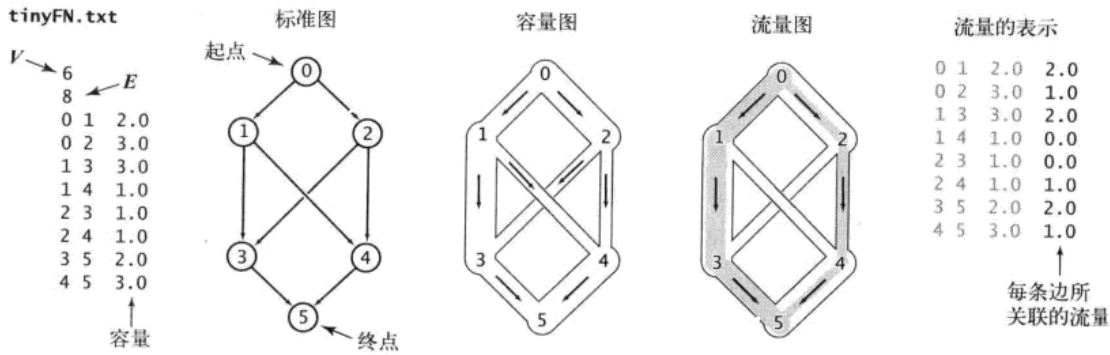


图 6.0.19 网络流问题详解

6.0.4.2 定义

因为它广泛的应用性，我们需要用精确的语言说明刚才介绍的通俗的概念和术语。

定义。一个*流量网络*是一张边的权重（这里称为*容量*）为正的加权有向图。一个 *st-* *流量网络*有两个已知的顶点，即起点 s 和终点 t。

有时我们会认为某些边的容量是无限的，或者说是没有容量限制的。这表示不会将其中的流量和它的容量进行比较，或者它的容量必然比所有流量都大。我们将流向一个顶点的总流量（所有指向该顶点的边中的流量之和）称为该顶点的*流入量*，流出一个顶点的总流量（由该顶点指出的所有边中的流量之和）称为该顶点的*流出量*，而两者之差（流入量减去流出量）则为称为该顶点的*净流量*。为了简化讨论，我们假设没有从 t 指出的边或是指向 s 的边。

定义。 *st-* 流量网络中的 *st-* 流量配置是由一组和每条边相关联的值组成的集合，这个值被称为边的流量。如果所有边的流量均小于边的容量且满足每个顶点的局部平衡（即净流量均为零，s 和 t 除外），那么就称这种流量配置方案是可行的。

我们将终点的流入量称为 *st-* 流量的值。命题 C 将会证明这个值和起点的流出量是相等的。有了这些定义，就能够正式地描述这个基本问题了。

最大 *st-* 流量。 给定一个 *st-* 流量网络，找到一种 *st-* 流量配置，使得从 s 到 t 的流量最大化。

为了简洁，我们将这样的流量配置称为最大流量，那么在网络中寻找这种配置的问题就是一个最大流量问题。在某些应用中，只需要知道最大流量的值即可，但一般情况下人们还是希望知道达到该值的具体流量配置（各条边的流量值）。

887
~
888

```
private boolean localEq(FlowNetwork G, int v)
{  // 检查每个顶点v的局部平衡
   double EPSILON = 1E-11;
   double netflow = 0.0;
   for (FlowEdge e : G.adj(v))
      if (v == e.from()) netflow -= e.flow();
      else               netflow += e.flow();
   return Math.abs(netflow) < EPSILON;
}
private boolean isFeasible(FlowNetwork G)
{
   // 确认每条边的流量非负且不大于边的容量
   for (int v = 0; v < G.V(); v++)
      for (FlowEdge e : G.adj(v))
         if (e.flow() < 0 || e.flow() > e.cap())
            return false;
   // 检查每个顶点的局部平衡
   for (int v = 0; v < G.V(); v++)
      if (v !=s && v != t && !localEq(v))
         return false;
   return true;
}
```

检查流量网络中的一种流量配置是否可行

6.0.4.3 API

表 6.0.4 和表 6.0.5 所示的 FlowEdge 和 FlowNetwork 简单扩展了第 3 章中相应 API。我们将会在 6.0.4.6 节学习 FlowEdge 的一种实现，它的基础是 4.3.2 节中的 WeightedEdge 类并添加了一个实例变量来保存边的流量。流量是有方向的，但 FlowEdge 的基类并不是 WeightedDirectedEdge，因为它还需要解决下面将要描述的一个更加抽象的剩余网络问题。我们需要使每条边都出现在它的两个顶点的邻接表中才能实现剩余网络。剩余网络能够增减流量并检测一条边是否已经饱和（无法再增大流量）或者是否为空（无法再减小流量）。这些抽象是通过 residualCapacity() 和 addResidualFlow() 方法实现的，我们将在之后讨论它们。FlowNetwork 的实现与 4.3.2 节中 WeightedEdge 的实现基本相同，因此这里将它省略。为了简化文件格式，我们约定起点的编号为 0，

终点的编号为 V-1，请见图6.0.20。有了这些API之后最大流量算法的目标就很明确了：构造一个网络，计算所有边中保存流量的实例变量的值并使得网络中的流量最大化。框注所示的是检验一个流量配置方案是否可行的用例代码，一般会将这种检查作为最大流量算法的最后一步。 889

表 6.0.4　流量网络中的边的 API

public class	**FlowEdge**	
	FlowEdge(int v, int w, double cap)	
int	from()	这条边的起始顶点
int	to()	这条边的目的顶点
int	other(int v)	边的另一个顶点
double	capacity()	边的容量
double	flow()	边中的流量
double	residualCapacityTo(int v)	v 的剩余容量
double	addFlowTo(int v, double delta)	将 v 的流量增加 delta
String	toString()	对象的字符串表示

表 6.0.5　流量网络的 API

public class	**FlowNetwork**	
	FlowNetwork(int V)	创建一个含有 V 个顶点的空网络
	FlowNetwork(In in)	从输入流中构造流量网络
int	V()	顶点总数
int	E()	边的总数
void	addEdge(FlowEdge e)	向流量网络中添加边 e
Iterable<FlowEdge>	adj(int v)	从 v 指出的边
Iterable<FlowEdge>	edges()	流量网络中的所有边
String	toString()	对象的字符串表示

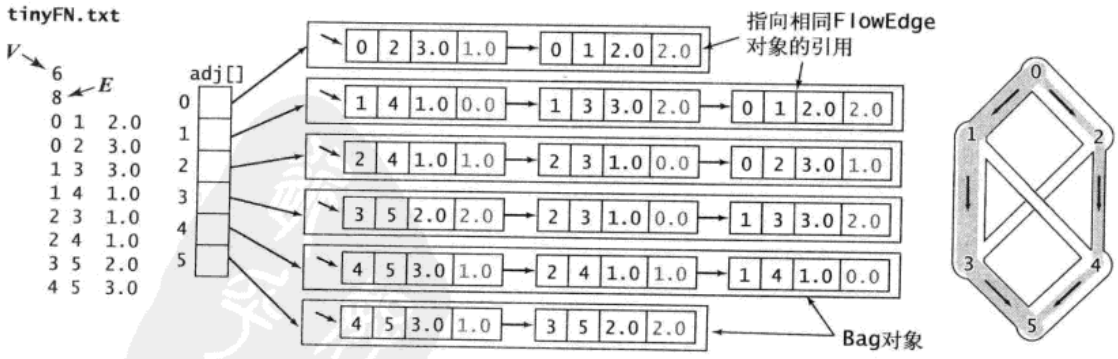


图 6.0.20　流量网络的表示 890

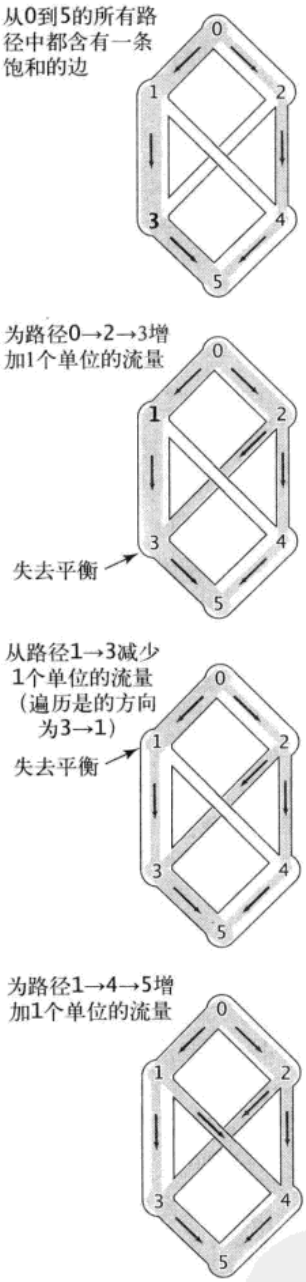


图 6.0.21 一条增广路径 (0 → 2 → 3 → 1 → 4 → 5)

6.0.4.4 Ford-Fulkerson 算法

在 1962 年，L.R.Ford 和 D.R.Fulkerson 发明了一种解决最大流量问题的有效方法。它是一种沿着由起点到终点的路径逐步增加流量的通用方法，因此它也是同类算法的基础。在经典文献中它被称为 Ford-Fulkerson *算法*，但它也被称为*增广路径算法*。考虑一个 *st*-流量网络中的任意一条从起点到终点的有向路径。假设 x 为该路径上的所有边中未使用容量的最小值。那么只需将所有边的流量增大 x 即可将网络中的总流量至少增大 x。反复这个过程，就得到了第一种计算网络中的流量分配方法：找到另一条路径，增大路径中的流量，如此反复，直到所有从起点到终点的路径上至少有一条边是饱和的。（这样在这条路径上就无法继续增大流量了。）这种方法在某些情况下能够计算出网络中的最大流量，但在有些情况下不行，图 6.0.17 就是这类情况。为了改进算法使之总是能够找到最大流量，就要用另一种更加通用的方式增大网络中的流量，即将依据变为网络所对应的*无向图*中从起点到终点的路径。在这样的路径中，当沿着路径从起点向终点前进时，经过某条边时的方向可能和流量的方向相同，那这条边即为*正向边*；也可能和流量的方向相反，那这条边即为*逆向边*。现在，对于任意非饱和正向边和非空逆向边，我们可以通过增加正向边的流量和降低逆向边的流量来增加网络中的总流量。流量的增量受路径上的所有正向边的未使用容量最小值和所有逆向边的流量的限制。这样的一条路径被称为增广路径，比如图 6.0.21。在新的流量配置中，路径中至少有一条正向边达到了饱和，或是至少有一条逆向边为空。以上所述的过程就是经典的 Ford-Fulkerson 算法（增广路径算法）的基础。我们将它总结如下。

> **Ford-Fulkerson 最大流量算法。**网络中的初始流量为零，沿着任意从起点到终点（且不含有饱和的正向边或是空逆向边）的增广路径增大流量，直到网络中不存在这样的路径为止。

令人惊讶的是（在关于流量性质的一定技术性限制之下），无论我们如何选择路径，该方法总能找出最大流量。如同 4.3 节中讨论的贪心最小生成树算法和 4.4 节中讨论的通用最短路径算法一样，它的意义在于证明了所有同类算法的正确性。我们可以用任何方法选择路径。人们发明了多种算法来计算增广路径的序列，以计算最大流量。这些算法的不同之处在于它们得到的增广路径数量和得到每条路径的成本，但它们实现的都是 Ford-Fulkerson 算法并能够找到网络的最大流量。

6.0.4.5 最大流 – 最小切分[①]定理

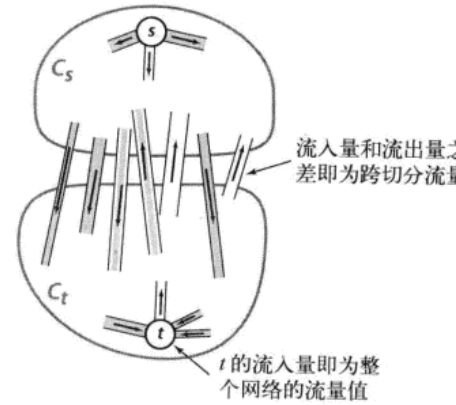
为了证明 Ford-Fulkerson 算法的任意实现所计算得到的流量确实是最大流量，需要证明一个

① 也有时译为“最大流–最小割”。——编者注

叫做*最大流 - 最小切分*的关键定理。理解这个定理是理解所有网络流算法中最重要的一步。顾名思义，定理的基础是网络中的流量和切分的关系，因此需要先定义和切分有关的名词。回顾4.3节，图的*切分*是将所有顶点分为两个不相交的集合，而一条横切边则是连接分别存在于两个集合中的两个顶点的一条边。对于流量网络，我们将它们的定义提炼如下。

定义。*st- 切分*是一个将顶点 *s* 和顶点 *t* 分配于不同集合中的切分。

在一个 *st-* 切分中，每条横切边要么是一条由含有 *s* 的集合指向含有 *t* 的集合的 *st-* 边，要么是一条反方向的 *ts-* 边。有时我们将 *st-* 边的集合称为一个*切分集*。在流量网络中，一个 *st-* 切分的容量为该切分的 *st-* 边的容量之和，*st-* 切分的*跨切分流量*（flow across）是切分的所有 *st-* 边的流量之和与所有 *ts-* 边的流量之和的差。在网络中删去 *st-* 切分的所有 *st-* 边（即切分集）将会切断所有从 *s* 到 *t* 的路径。而重新添加其中的任意一条边都会得到一条从 *s* 到 *t* 的路径。切分能够抽象许多应用。比如我们的原油流量模型，切分会从入口流向出口的原油完全切断。如果将切分的容量看作这么做的成本，那么切断流量的最有效方法是解决以下问题。



最小 *st-* 切分。给定一个 *st-* 网络，找到容量最小的 *st-* 切分。简单起见，我们将这样的切分称为*最小切分*，而将在网络中找到它的问题称为*最小切分问题*。

最小切分问题的定义中并没有提到流量，而且这些定义似乎和增广路径算法无关。从表面上来看，计算最小切分（得到一组边）似乎比计算最大流量（为所有的边赋权值）更容易。但实际上，最大流量和最小切分问题是紧密相关的。增广路径算法本身就是证明。流量和切分的以下基本关系即可证明 *st-* 流量网络中的局部平衡即意味着整个网络的全局平衡（推论一），并且可以得到任意 *st-* 流量值的上界（推论二）。

命题 E。对于任意 *st-* 流量网络，每种 *st-* 切分中的跨切分流量都和总流量的值相等。

证明。设 C_s 为含有顶点 *s* 的集合，C_t 为含有顶点 *t* 的集合。对 C_t 使用归纳法：当 C_t 仅含有 *t* 时该命题成立，若将一个顶点由 C_s 移动到 C_t，则该结点处的局部平衡意味着可以一直保持该性质。因此，通过移动顶点可以得到任意 *st-* 切分。

推论。*s* 的流出量等于 *t* 的流入量（即 *st-* 流量网络的值）。

证明。令 *Cs* 为 {*s*} 即可。

推论。*st-* 流量网络的值不可能超过任意 *st-* 切分的容量。

891 ~ 893

命题F（最大流量-最小切分定理）。令f为一个st-流量网络，以下三种条件是等价的：

i 存在某个st-切分，其容量和f的流量相等；

ii f达到了最大流量；

iii f中已经不存在任何增广路径。

证明。根据命题E的推论，我们可以由条件i得到条件ii。因为增广路径的存在意味着存在某个流量更大的网络配置，这与f的最大性相冲突，因此由条件ii也可以得到条件iii。

但还需要证明条件iii和条件i等价。令C_s为由s通过所有不含有任何饱和正向边或空逆向边的无向路径可达的所有顶点组成的集合，令C_t为其余的顶点的集合。t必然存在于C_t中，因此(C_s,C_t)为一个st-切分。它的切分集完全由饱和正向边和空逆向边组成。该切分的跨切分流量和它的容量相等（因为所有正向边都是饱和的，而所有逆向边都是空的），即等于网络中的总流量（由命题E可得）。

推论（完整性）。当所有容量均为整数时，存在一个整数值的最大流量，而Ford-Fulkerson算法能够找出这个最大值。

证明。每条增广路径都会将总流量增大某个正整数值（正向边中未使用容量的最小值和逆向边的容量都是正整数）。

即使所有边的容量均为整数，我们也可以设计出能够达到最大流量的非整数配置，但这里不需要考虑这样的配置。从理论角度来说，下面的意见是很重要的：我们已经演示过并且实际情况也需要允许容量和流量可以为实数，但它会导致一些异常情况。例如，已知Ford-Fulkerson算法在原则上可能得到无穷多的增广路径以至于无法收敛到某种最大流量的配置。我们讨论的这个版本总是可以收敛的，即使是实数值的容量和流量也不例外。无论我们用什么方法寻找增广路径，无论我们找到了什么样的路径，最后总是能够得到一种不存在任何增广路径的流量配置，即最大流量的配置。

894

6.0.4.6 剩余网络

通用的Ford-Fulkerson算法并没有指定寻找增广路径的方法。如何才能找到不含有饱和正向边和空逆向边的路径呢？为此，我们给出如下定义。

定义。给定某个st-流量网络和其st-流量配置，这种配置下的剩余网络中的顶点和原网络相同。原网络中的每条边都对应着剩余网络中的1～2条边。它的定义如下：对于原网络中的每条从顶点v到w的边e，令f_e表示它的流量、c_e表示它的容量。如果f_e为正，将边w→v加入剩余网络且容量为f_e；如果f_e小于c_e，将边v→w加入剩余网络且容量为c_e-f_e。

如果从v到w的边e为空（即f_e为0），剩余网络中就只有一条容量为c_e的边v→w与之对应；如果该边饱和（即f_e等于c_e），剩余网络就只有一条容量为f_e的边w→v与之对应；如果它既不为空，也不饱和，那么剩余网络中将含有相应容量的v→w和w→v。请见图6.0.22。

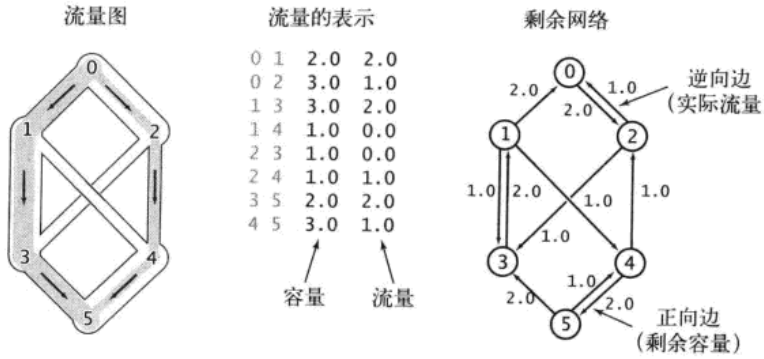


图 6.0.22 网络流问题详解

乍一看，剩余网络有些让人困惑，因为与流量对应的边的方向却和流量本身相反。正向边表示的是剩余的容量（即如果选择从这条边通行所能增长的流量）；逆向边表示了实际流量（即如果选择从这条边通行将会减少的流量）。后面框注中的代码给出了在 FlowEdge 类中实现剩余网络这种抽象所需的方法。通过这些实现，虽然该算法处理的是剩余网络，但它实际上是在检查所有剩余的容量并（通过边的引用）修正流量配置。

895

流量网络中的边（剩余网络）

```
public class FlowEdge
{
   private final int v;                          // 边的起点
   private final int w;                          // 边的终点
   private final double capacity;                // 容量
   private double flow;                          // 流量

   public FlowEdge(int v, int w, double capacity)
   {
      this.v = v;
      this.w = w;
      this.capacity = capacity;
      this.flow = 0.0;
   }

   public int from()          {  return v;          }
   public int to()            {  return w;          }
   public double capacity()   {  return capacity;   }
   public double flow()       {  return flow;       }

   public int other(int vertex)
   // 同Edge类

   public double residualCapacityTo(int vertex)
   {
      if      (vertex == v) return flow;
      else if (vertex == w) return capacity- flow;

      else throw new RuntimeException("Inconsistent edge");

   }

   public void addResidualFlowTo(int vertex, double delta)
   {
      if      (vertex == v) flow -= delta;
```

```
        else if (vertex == w) flow += delta;
        else throw new RuntimeException("Inconsistent edge");
    }
    public String toString()
    {  return String.format("%d->%d %.2f %.2f", v, w, capacity, flow);  }
}
```

这里的 FlowEdge 类的基础是 4.4 节中对加权边的 DirectedEdge 类的实现（请见 4.4.2 节框注“加权有向边的数据类型”），它添加了一个实例变量 flow 和两个方法来实现了剩余网络。

896

我们可以使用 from() 和 other() 方法处理两个方向的边：e.other(v) 可以返回 e 的两个顶点中和 v 相对的另一个顶点。residualCapacityTo() 和 residualFlowTo() 方法实现了剩余网络。剩余网络使得我们可以通过图中的搜索算法寻找增广路径，这是因为在剩余网络中所有从起点到终点的路径都是原流量网络中的一条增广路径。沿着增广路径增大流量意味着修改剩余网络。例如，至少有一条路径上的边变得饱和或变为空，因此在剩余网络中至少有一条边将会改变方向或者消失。（我们使用的是抽象的剩余网络，因此只会检查正容量，不需要实际插入或删除边。）

```
private boolean hasAugmentingPath(FlowNetwork G, int s, int t)
{
   marked = new boolean[G.V()];  // 标记路径已知的顶点
   edgeTo = new FlowEdge[G.V()]; // 路径上的最后一条边
   Queue<Integer> q = new Queue<Integer>();
   marked[s] = true;             // 标记起点
   q.enqueue(s);                 // 并将它入列
   while (!q.isEmpty())
   {
      int v = q.dequeue();
      for (FlowEdge e : G.adj(v))
      {
         int w = e.other(v);
         if (e.residualCapacityTo(w) > 0 && !marked[w])
         {                         // （在剩余网络中）对于任意一条连接到一个未
                                      被标记的顶点的边
            edgeTo[w] = e;         // 保存路径上的最后一条边
            marked[w] = true;      // 标记w，因为路径现在是已知的了
            q.enqueue(w);          // 将它入列
         }
      }
   }
   return marked[t];
}
```

在剩余网络中通过广度优先搜索寻找增广路径

6.0.4.7 最短增广路径算法

对 Ford-Fulkerson 算法最简单的实现可能就是最短增广路径算法了（最短指的是路径长度最小，而非流量或是容量）。J.Edmonds 和 R.Karp 在 1972 年发明了这个算法。这里，增广路径的查找等价于剩余网络中的广度优先搜索（BFS），如 4.1 节所述。你也可以将 hasAugmentingPath() 的实现与广度优先搜索实现的算法 4.2 比较一下。（剩余网络是有向图，因此这实际上是一个

有向图处理算法。）这个方法为完整实现剩余网络的算法 6.14 打下了基础，它非常简洁。为了方便，我们将这个方法称为最短增广路径的最大流量算法。它处理样例数据的详细轨迹如图 6.0.23 所示。 897

算法 6.14 最短增广路径的 Ford-Fulkerson 最大流量算法。

```
public class FordFulkerson
{
   private boolean[] marked;     // 在剩余网络中是否存在从s到v的路径?
   private FlowEdge[] edgeTo;    // 从s到v的最短路径上的最后一条边
   private double value;         // 当前最大流量
   public FordFulkerson(FlowNetwork G, int s, int t)
   {  // 找出从s到t的流量网络G的最大流量配置
      while (hasAugmentingPath(G, s, t))
      {  // 利用所有存在的增广路径
         // 计算当前的瓶颈容量
         double bottle = Double.POSITIVE_INFINITY;
         for (int v = t; v != s; v = edgeTo[v].other(v))
            bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));
         // 增大流量
         for (int v = t; v != s; v = edgeTo[v].other(v))
            edgeTo[v].addResidualFlowTo(v, bottle);

         value += bottle;
      }
   }

   public double value()         {  return value;      }
   public boolean inCut(int v)   {  return marked[v];  }

   public static void main(String[] args)
   {
      FlowNetwork G = new FlowNetwork(new In(args[0]));
      int s = 0, t = G.V() - 1;
      FordFulkerson maxflow = new FordFulkerson(G, s, t);

      StdOut.println("Max flow from " + s + " to " + t);
      for (int v = 0; v < G.V(); v++)
         for (FlowEdge e : G.adj(v))
            if ((v == e.from()) && e.flow() > 0)
               StdOut.println("   " + e);
      StdOut.println("Max flow value = " +  maxflow.value());

   }
}
```

这段 Ford-Fulkerson 算法的实现会在剩余网络中寻找最短增广路径，找出路径上的瓶颈容量并增大该路径上的流量，如此往复直至不再存在从起点到终点的增广路径为止。 898