PEARSON

Data Structures and Their Algorithms

数据结构与算法

[美] Harry R. Lewis, Larry Denenberg 著
肖依 李志刚 陈涛 译

中国电力出版社
CHINA ELECTRIC POWER PRESS

图书在版编目（CIP）数据

数据结构与算法 /（美）路易斯，（美）丹尼伯格著；肖侬，李志刚，陈涛译. —北京：中国电力出版社，2010.7
ISBN 978-7-5123-0545-8

Ⅰ. ①数… Ⅱ. ①路… ②丹… ③肖… ④李… ⑤陈… Ⅲ. ①数据结构 ②算法分析 Ⅳ. ①TP311.12

中国版本图书馆 CIP 数据核字（2010）第 111493 号

中国电力出版社出版、发行
（北京市东城区北京站西街 19 号 100005 http://www.cepp.sgcc.com.cn）
印刷厂印刷
各地新华书店经售
*
2012 年 4 月第一版 2012 年 4 月北京第一次印刷
毫米× 毫米 16 开本 印张 千字
定价 **49.00** 元

前　言

就像所有的工程活动一样，计算机编程兼有技术与科学的成分。建一座桥或者编一段程序，都需要熟悉相关领域中关于整体设计的一些已有技术方法。如何在多种可用的技术中作出巧妙地选择，需要了解决定其性能和经济效用的数学原理。本书涉及数字电子计算机中数据的组织、重组、移动、使用和提取等操作方法，及相关的数学分析。该学科就像静力学和动力学是机械工程的核心一样，是计算机编程艺术的理论基础。

本书所选的主题基于以下几个朴素的原则。第一，本书只讲解实用的技术，而忽略一些理论上非常虽然出色、但不太实用的算法。第二，本书既包含经典的方法，也包括最近发现的方法；这种选择是基于内在的简便性、广泛的应用性和潜在的使用性等标准，而不是无遗漏地包含前人书籍的目录。例如，第 6 章中集合的列表和树的实现，以及包括静态数据的最优二分搜索树的构建算法，也包括比较新的动态数据的跳跃表结构。在其他章节中会讲述伸展树、可扩充散列、网格文件和其他优秀的算法。第三，所有讲述的方法基本上都会对应的分析。本书的一个主要的目的就是介绍一些相对简练和非技术化的算法分析，但同时也能体现出其重要的性能特征。就像在机械工程这门课程中，扩展性是重要的一课：同样满足某种规模结构的方法，当规模扩大十倍以后不再适合。

本书忽略了一些无关紧要的语法细节。本书的主题是算法，而不是针对某一具体编程语言的算法实现，因此采用程序员容易读懂的伪代码来描述算法。我们假设本书的读者起码学过如 Pascal 或者 C 等计算机编程语言，能够没有困难地将伪代码翻译成标识符声明、begin-end 块之类的计算机语言。为了简化在动态树算法中的一个最复杂的编码问题——搜索过程中如何更改已遍历节点的指针——引入定位格（locatives），一种新的编程工具。我们能够用不到一页的纸张伪代码准确和完整地描述每一个算法。

同样，本书在细致地分析算法的同时，避免使用大学二年级学生都没有掌握的数学工具。对数、指数和几何级数的和在很多分析里面起到关键作用，那么在第 1 章中给出了这些概念的一些基础的例子。古典概率论推理同样是基础，本书对此概念也进行了介绍。另一方面，在很少几个地方用到了微分学（积分几乎没有用到），那么没有学过微积分的读者可以跳过这些分析直接看结论。

每一章的后面都附有练习题和参考文献。这些练习题的各个部分分别对应该章节中的主要内容。这些问题从使用小的数据集合直接模拟算法，到需要将文章忽略的一些问题补充完整或者设计、分析新的扩展数据结构和算法。参考文献所应用的资料都有重要的历史意义，或者是对该问题有很好的总结。

第 13 章是对数据结构设计和分析的综合，给出了一些开放，自由的练习。这些问题有的是一两页纸的文卷；其他可能需要一个学期的编程实践。它们的共同点是不针对具体的数据结构，而是一些有关存在多种设计数据结构方法的计算情形的问题，也不能通过清晰

的数学分析来选择使用那一种方式。我们希望通过这些练习，学生们可以获得一些有效的计算方法的工程经验。

致谢

在此我们向多年来为本书的写作过程中提出建议和指正的朋友致以谢意。Paul Bamberg、Mihaly Gereb、Victor Milenkovic、Bernard Moret 和 Henry Shapiro 使用该书的草稿授课，并给予我们宝贵的反馈意见。感谢 Danny Krizanc 指出了快速排序分析中的一处错误，同时感谢 Bob Sedgewick 根据该建议作出了修改。Marty Tompa 仔细阅读了该书草稿最后的版本，并帮助修改了很多错误。Mike Karr 对方位格提出了有益的批评。Bill Gasarch 和 Victor Milenkovic 补充了大量的练习和参考文献。David Johnson 帮我们解决了一个内存管理方面的问题。Stephen Gildea 是我们的舞蹈顾问（dance consultant）。BBN Communications 为本书的第二作者提供了良好的支持环境。Chiron Inc.的 Joe Snowden 能够迅速而且专业的解决排版方面的问题。本书根据哈佛大学讲授的 Computer Science 124（原先是 Applied Mathematics 119）写作完成。一大批天才的教师多年来为我们对如何表述这些材料的认识作出了贡献，同时给出了很多习题。在这些教师当中有 David Albert、Jeff Baron、Mark Berman、Marshall Brinn、David Frankel、Adam Gottlieb、Abdelsalam Heddaya、Kevin Knight、Joe Marks、Mike Massimilla、Marios Mavronicolas、Ted Nesson、Julia Shaffner、Ra'ad Siraj、Dan Winkler，以及 Michael Yampol，感谢他们。特别感谢 Alex Lewin 细致的校正。有位匿名的审稿人对几个分析提供了宝贵的改进意见。

在写作遇到困难的时候 Marlyn McGrath Lewis 给我们无限的鼓励，并给出了智慧的建议使得该工作得以顺利完成。

本书的排版使用的是 Donald Knuth 的 TEX，它保证了本书编排工作的顺利进行。

目　录

前　言
第 1 章　导言……1
1.1　程序设计：一项工程活动……1
1.2　计算机科学背景……3
1.3　数学背景……12
习题……26
参考文献……31
第 2 章　算法分析……33
2.1　算法的属性……33
2.2　精确分析与渐进分析……35
2.3　算法范例……42
习题……46
参考文献……50
第 3 章　线性表……52
3.1　线性表操作……52
3.2　线性表的基本表示法……53
3.3　栈和递归……58
3.4　线性表的遍历表示法……60
3.5　双向链表……62
习题……65
参考文献……68
第 4 章　树……70
4.1　基本的定义……70
4.2　几种特殊的树……74
4.3　树的操作和遍历……75
4.4　树的实现……79
4.5　树遍历和扫描的实现……82
小结……92
习题……93
参考文献……96
第 5 章　数组与字符串……97
5.1　抽象数据类型的数组……97

5.2 数组的连续表示法 99
5.3 稀疏数组 103
5.4 字符串的表示法 107
5.5 字符串搜索 115
习题 123
参考文献 129
第 6 章 集合的表和树实现 131
6.1 抽象数据型集合和字典 131
6.2 无序表 133
6.3 有序表 136
6.4 二分搜索树 145
6.5 静态二分搜索树 150
习题 156
参考文献 162
第 7 章 动态字典的树结构 165
7.1 AVL 树 165
7.2 2-3 树和 B-树 173
7.3 自调节二元搜索树 184
习题 190
参考文献 193
第 8 章 数据集合 195
8.1 位向量 195
8.2 TRIE 和数字搜索树 197
8.3 散列技术 201
8.4 可扩展散列 211
8.5 散列函数 214
习题 219
参考文献 223
第 9 章 特殊操作集合 224
9.1 优先级队列 224
9.2 带合并的不相交集合 231
9.3 范围搜索 238
习题 248
参考文献 253
第 10 章 内存管理 255
10.1 内存管理问题 255
10.2 单一长度记录 257
10.3 变长记录的紧凑 265
10.4 变长块池的管理 267

10.5 伙伴系统 ……274
习题……276
参考文献……279
第 11 章 排序 ……282
11.1 排序算法的种类……282
11.2 插入排序和希尔排序 ……283
11.3 选择排序和堆排序 ……286
11.4 快速排序……288
11.5 信息理论的下界 ……292
11.6 数字排序……293
11.7 外部排序……297
11.8 中值的查找……304
习题……306
参考文献……311
第 12 章 图……314
12.1 图及其表示……314
12.2 图搜索算法……320
12.3 图的贪婪算法……328
12.4 所有顶点对之间的最小代价路径 ……334
12.5 网络流（Network Flow） ……336
习题……345
参考文献……350
第 13 章 数据结构工程……353
习题……365
参考文献……365
附录 A 定位格 ……367
习题……370

第 1 章

导　　言

1.1　程序设计：一项工程活动

程序是解决一种问题的方案。该问题可能是具体明确的，例如，将 1～100 所有整数的平方根精确到 10 个小数位；也可能是大而模糊的，例如，开发一种计算机印刷系统。大而模糊的问题可以通过分解成小而明确的问题来解决。如在上面的计算机印刷系统中，单词遇到跨行的时候需要确定连字符的位置就是一个小问题。本书的宗旨是对足够明确的问题进行程序设计，这些问题可以用简单的语言来描述，并且可以很容易地判断什么才是它的一个解决方案，而且在对一些更复杂的问题进行程序设计时它们经常会出现。

即使用简单的语言就可以精确描述的问题也会有很多可能的解决方案。虽然通过改变变量的名称，如将 FORTRAN 翻译为 Pascal 或者其他类似方法可以得到“不同的”程序。通过运用不同的方式方法，却可以得到更不一样的解决方案。例如，在一个有序的单词表中找出单词 *K*，可以给出下面三种方式：

A. 从表头开始遍历，将表中的每一个单词和 *K* 进行比较，直到找到 *K* 或者到表尾。

显然该方式没有利用表的有序性。下面是一个更聪明的改进。

B. 像（A）一样从表头开始遍历。当找到 *K* 或找到一个应该在 *K* 后面出现的单词，或者直到表尾停止搜索。

这个方案通过改变程序的结束条件减少了方法（A）中一些不必要的工作。例如，如果要寻找单词 aardvark，使用（B）方法我们就不必在单词表的后面继续寻找。当然除此之外还有更好的方法。

C. 从单词表的中间开始查找。如果 *K* 恰好在单词表的中间，那么工作便结束了；否则，通过比较 *K* 和中间的这个单词，来决定 *K* 到底是位于单词表的上半部分还是下半部

分。然后对选定的二分之一单词表重复以上过程。在接下来的迭代过程里用同样的方式来搜索四分之一表、八分之一表……。当找到 K 时或者一直将迭代过程缩小到没有发现为止。

（C）方法被称成为“**二分搜索法**”，是三种方法里面最快的（当然它也是最不易于编程的。事实上，上面的叙述忽略了很多重要的细节，例如，怎样才能确定一个长度为 10 的表的“中间位置”？）。第 6 章中会详细讨论“二分搜索法”，不过可以通过上面的例子得出几点启示。首先，（A）、（B）和（C）是解决同一个问题的三个不同算法。它们都不是程序，描述它们的语言也就不是程序设计语言。但是任何程序员都会理解这些描述。同时他们也知道用 FORTRAN 和 Pascal 实现的比如方法（C）的程序体现了同一个算法，而用 Pascal 实现的方法（A）和方法（C）则使用了完全不同的算法。

算法是用来解决一种问题的计算方法。本书的目标是向你介绍一些在计算机程序设计过程中一再遇到的问题的最重要算法，并且教你在需要选择的时候如何选择算法（你经常会碰到这样的事情）。

我们选择某个算法而非其他算法，或者因为该算法总是比较快，或者是通常比较快，或者使用较少的内存。也有可能是因为该算法易于编写程序；或者该算法比较通用，可以适应以后问题发生变化的可能性。对于本书而言，我们更多的是强调算法的速度和使用内存的情况。

当然我们不可能通过编写一个程序，以它的运行速度来确定算法的速度。通过这种方式得到的数字过于依赖程序员的水平和所使用计算机的速度，因为计算机有通用和专用之分。所以我们会用更抽象更数学的方式来处理该问题。比如上面的单词表的长度为 n，那么方法（A）的时间开销和 n 成正比；单词表的长度增长一倍，那么算法的时间开销大体上是原来的两倍。而方法（C）的时间开销最坏的情况也就是和 n 以 2 为底的对数成正比（也就是按照 C 方法来划分表，一直到成为单个元素为止）。

第 2 章将会详细介绍有关“算法分析”的内容，下面给出一些简单的启示。利用数学工具来分析所考虑的算法，是因为正确的数学工具可以支持该算法的所有应用。为了开发这些数学工具，需要提出我们要理解的场景的数学模型。例如，为了得出方法（A）所需时间与单词表的长度成正比，需要假设从表中读取每一个元素的时间是一样的。一般对很多实现表的方式来说这个结论都是成立的，所以从一个比较粗略的假设中可以得出通用性。

程序设计既不是纯科学也不是纯数学而是一项工程活动。在编写程序的时候不能忽视实际问题的繁杂细节，而且也不会有唯一的答案。工程师是基于对几个可选方案结果的理解作设计决定的。这种理解是从有关规律的知识，特别是那些用数学术语陈述的知识得来的，这些知识涵盖了很广泛的情形。如果要建造一座桥，工程师通过对环境归纳出相应的参数（长度、载重等），利用一般原理刻画不同种类桥的特性，决定在河流的特定位置修建一座什么样的桥。工程师同时也要运用从工程建造过程中积累的智慧经验。程序员也应该这样来思考问题，既需要理解影响算法性能的一般原理，也需要从实现算法的过程中获得智慧。

1.2　计算机科学背景

冯・诺依曼结构计算机的存储器和数据

在本书讨论算法的过程中涉及的计算机称为“冯・诺依曼”机器*。这样的计算机包含一个单独的处理器，该处理器与内存相连接。存储器为二进制的，即最终由单独的比特组成，由比特再组织成更大的单位或者单元。一个单元可能包含一个整数、字符、浮点数，或者是其他数据类型的元素。在本书的术语中，单元的大小依赖于所存储数据的类型，因此单元与字节（byte）或者字（word）等无关。而且连续的单元可以组合在一起存储几个数据项，这样的数据项组合在一起称作**记录**（record）。记录其实是包含一个逻辑上结构化的对象的内存单元。记录中的单个组件称为记录的**字段**（field）。比如图 1.1 所示的一个记录包括一个人的姓、名、身高以及体重等；其姓和名各占 16 字节，身高（单位：英尺）和体重（单位：磅）各占 4 字节；总计该记录的长度为 40 字节。

每个存储单元都有一个用数字表示的**地址**，处理器通过该地址提取内存单元的一个数据（a datum）。尽可能用最小的内存单元对内存进行编址，比如说 8 比特。在图 1.1 的例子中，从 1240 号地址开始按字节划分此记录；在 1280 号地址以前结束。

如果一系列的存储单元类型相同，有固定的间隔且在内存中连续，我们称这样组合在以一起的存储单元为表：存储单元 C_0、C_1、…、C_{n-1} 的地址按照单元的长度连续增加（图 1.2）。所以假设 X 为表头所在的地址，c 是一个单元的长度，那么 C_i 的地址为 $X+c\cdot i$。如图 1.2 所示，$c=40$、$X=1240$，那么 C_i 的地址为 $1240+40\cdot i$。

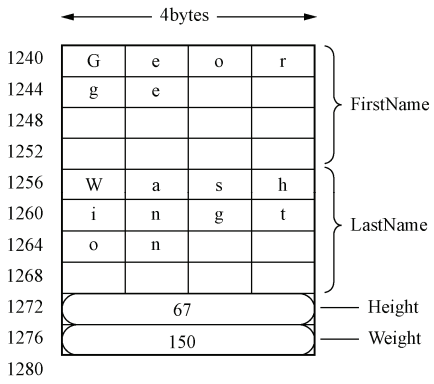

图 1.1　一个数据记录在内存中的布局。该记录有四个字段，分别有 16、16、4 和 4 个字节

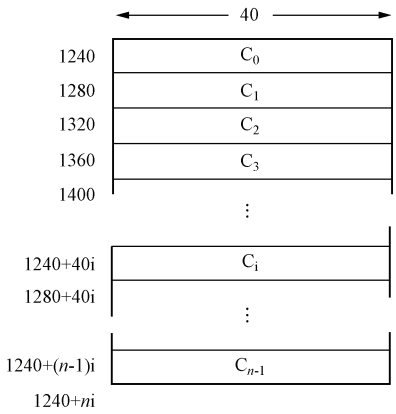

图 1.2　表在存储器中的布局

对于给定的记录来说，字段的位置可以通过字段的大小和距记录头的距离得到。如图 1.1 所示，该记录的地址为 X，那么：

- FirstName 字段的长度为 16 字节，地址为 X。

* 实际上到目前为止，所有数字计算机都是冯・诺依曼结构的机器。最近几年里出现了一些非冯・诺依曼结构的机器，比如有些机器可以拥有一些或者上千个处理器，它们分散在不同的地方且以复杂的方式连接。设计这些机器的程序需要一种新的算法思想（见本章末的参考文献）。

- LastName 字段的长度为 16 字节，地址为 $X+16$。
- Height 字段的长度为 4 字节，地址为 $X+32$。
- Weight 字段的长度为 4 字节，地址为 $X+36$。

读取位于地址 X 的某个具体的记录字段，可以通过 FirstName(X)、LastName(X)来表示。如本例所示，也可以为该记录定义一个比如 Person 的类型名称。

本书所讲的存储器都是随机访问的，即对任何地址单元取、存数据的时间都相同，与地址无关（虽然说读取或者存储较大的数据或记录，有可能比较小的时间要长）。对于上面的例子，在 C_0、C_1、…、C_{n-1} 所组成的表中访问第 i 个记录的时间是个定值，与 i 无关。

地址作为一个数字而言，当然也可以存储到内存中。存储其他单元地址的单元称为其他单元的引用或者指针。图 1.3 中利用箭头来指示一个单元和存储其地址的单元的关系。

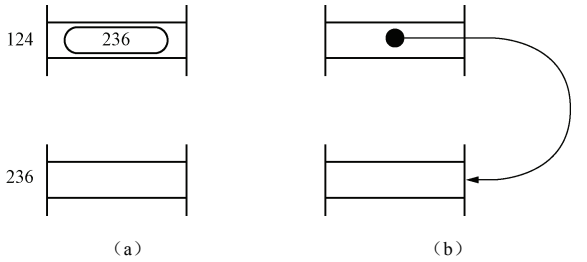

图 1.3　地址和指针

（a）计算机内的情况：地址为 124 的单元存储着数 236；
（b）逻辑表示：124 处单元包含的数字作为一个地址指向位于地址 236 处的单元

指针为在存储器内建立具有复杂记录间内部引用的结构提供了机会。例如，图 1.4 所示的存储结构称作**单向链表**（singly linked list）。单向链表中的每个项都有一个或者多个字段用来存储数据（图 1.4 中的 Info 字段），并且还有一个 Next 字段用来存储一个地址。这整个记录结构称作一个节点（Node）。虽然各个记录中保存的地址并不是连续的或者符合同样的模式，但是所有节点在逻辑上组织成一个顺序列表，因为可以从第一个节点开始，

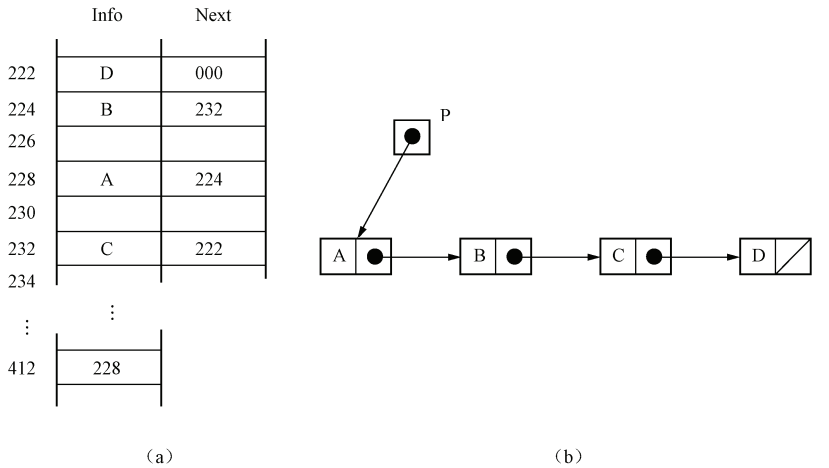

图 1.4　链表

（a）内部表示；（b）图释。在该例中，A 表示左边内存地址为 000 的单元，也就是右面斜线所指的单元。右边称为 P 的单元是指针变量，位于单元 412 处，并且指向链表的首部地址 228

通过它的 Next 字段中的地址访问下一个节点，通过下一个节点的 Next 字段中的地址访问下一个节点的下一个节点，依次类推。列表的末尾可以在其 Next 字段中放置一个特殊地址 Λ；这些都绘在图 1.4 中，用一条斜线表示末尾节点的 Next 字段，而其他节点中的 Next 字段中放置黑点，用一个箭头指向下一个节点。在图 1.4（b）中有两种方格：列表记录中的 Info 字段（图中包含 A、B、C、D 的方格）和指针变量 *P*（指向指针列表头的方格）。

单链表和表（table）一样，也可以表示具有相同数据类型的一系列数据项。不过这种数据表示方式对存储器的使用并不经济，因为每个节点必须具有一个指针字段指向下一个节点。而且它也没有表的一个良好性质：通过有序的索引读取表中的一个单元，并且时间开销与索引无关；一般，搜索单链表中的一个节点，需要从列表头开始遍历所有该节点以前的节点。不过从另一方面来说，像插入记录或者删除记录这样的操作，对表来说就需要移动或者重排很多数据，在链表中只需要移动一对指针即可。正是因为链接结构能够如此好地支持动态结构的重新组织，才成为很多有效率的算法的核心。链表的另一个优点是，事先可以不知道存储器的使用量，而表就需要根据最大长度预分配存储器。

用 *p* 表示存储一个指针所需要的比特数；那么单链表中每个记录都有 *p* 个比特的额外开销。大多数情况下，指针字段并不需要存储完整的机器地址。如果数据结构中的所有记录都在一个长度为 *n* 的表中，并且知道起始地址，那么只要存储一个范围在 0 到 *n*－1 索引就可以访问某个单元，同理对于一般地指针来说也只需要存储少量的比特达到该目的。不过通过这样的方式得到的好处，在由索引计算完整地址的算术计算、考虑记录所在的特定表的基址的时候会抵消一部分。

作为一般的问题，数据结构的设计过程中经常会涉及到折中或者**权衡**（tradeoffs）的问题：虽然我们常常希望某个数据结构在不同方式中都是优越的，但是这并不能同时实现，因此我们常常牺牲某些特征来换取其他几种较好的特征。比如，使用表索引替代指针用速度换取存储器的使用，用表来替代链表减少了内存的使用但降低了插入与删除的速度。

程序中的符号

大多数程序都是用高级语言写的。高级语言比起低级的机器语言或者汇编语言，在算法描述上有很多优点。高级程序语言在讨论数据集合时可以将它们看成一个整体，并不需要知道它们在存储器中是如何表示的。比如，在 Pascal 语言中，二维实数数组 *A:array*[1..10，1..10]由 100 个实数组成，按某种方式存放在内存中。对于 Pascal 程序员来说并不需要知道这些数据是如何在内存中存放的，我们只需要确保每次访问，比如说访问 *A*[5，7]的时候都能得到同样元素即可，虽然不需要得到相同的值。不过，如果需要详细地考虑算法的性能，我们有可能需要能够对存储器的底层组织有更紧凑的控制，而不能仅仅限于高级语言所规定的语法。基于这个原因，需要严格区分**数据类型**（data type）和**数据结构**（data structure）这两个概念。数据类型是程序语言中的概念，而数据结构是计算机存储器的一个逻辑组织，通常指的是存储单元地址的使用模式。

procedure *SinglyLinkedInsert*(**pointer** *P*,*Q*):
{Insert the cell to which *P* points just after the cell to which *Q* points}
　　Next(*P*)←**Next**(*Q*)
　　Next(*Q*)←*P*

算法 1.1　在单链表中插入一个节点

随着表达方式的增加，高级程序语言也表现出一些不足。像 Pascal 这样的语言会有“强类型”，它的意思是每一个变量和每一个数据对象都有一个数据类型，当且仅当它们具有相同的数据类型时一个值能赋给一个变量。某些算法有可能在底层操作数据的表示，或者是在不同的时间对相同的存储单元赋予不同类型的数据对象，像 Pascal 这样的语言不能有效地实现这种情况。另一个就是有关算法操作地址的问题。某些语言根本没有地址或者“指针”数据类型；某些语言有“指针”类型，但是强化了指针与所指向的数据类型的关系（因此，指向一个记录的指针和指向该记录第一个组件的指针是不同类型的对象，虽然它们指向的是同一个机器地址）。

我们采用了一些折中的符号来描述算法。使用 $T[a..b]$表示索引从 a 到 b（都是整数）的表。$T[i]$表示表 T 的第 i 个元素，其中 $a \leqslant i \leqslant b$。一般假设表占用连续的存储单元。索引方式同表一样的数组，会在第 5 章详细讨论。数组不需要关心条目是如何存储的，也不需要关心访问元素的时间。

同样使用高级符号表示记录类型及其字段，在变量和适当类型的值之间自由地使用赋值符号(←)。如果 P 是一个记录的指针，该记录有一个名为 F 的字段，那么符号 $F(P)$表示 P 所指向的记录的 F 字段。比如，算法 1.1 将 P 指针指向的节点插入到 Q 指针指向的节点的后面（图 1.5）。

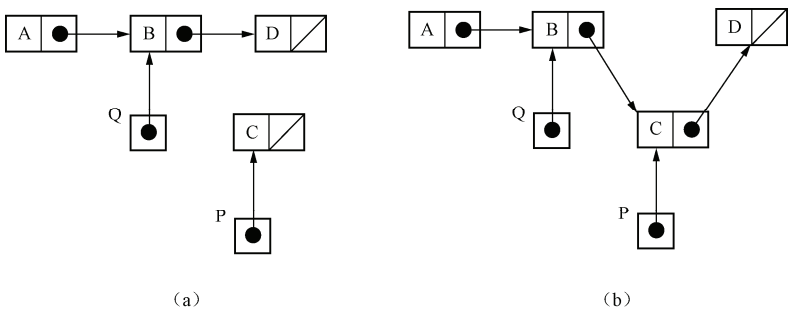

图 1.5　在 Q 所指向的节点处插入一个指针 P 指向的新节点

对赋值符号进行扩展，使用“列向量”符号表示对几个变量同时进行赋值。比如，符号

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \leftarrow \begin{pmatrix} Y \\ Z \\ X \end{pmatrix}$$

表示将左边三个变量 X、Y 和 Z 的值进行“旋转”；X 被赋以原来 Y 的值，Y 被赋以原来 Z 的值，Z 被赋以原来 X 的值。$\begin{pmatrix} X \\ Y \end{pmatrix} \leftarrow \begin{pmatrix} Y \\ X \end{pmatrix}$常简写为 $X \leftrightarrow Y$，即互换 X 和 Y 的值。对于大多数的程序语言来说，这种赋值方式是不能直接进行的，还需要引入一个“临时”变量，其唯一的目的在于使两个或更多的赋值能够依次进行。

有时为了方便会反复引入其他符号。不过，我们试图定义尽可能少的必要的符号；如果用语言可以更容易表达某件事情，我们就用语言表达而不用特殊的符号，因为作为一个有经验的程序员能够理解如何通过选定的程序语言表达相同的语法。

我们对于算法的控制部分，采用“if…then…”和“if…then…else…”或者是“while…do…”和“for…do…”循环语句等。像“repeat forever…”的循环体执行次数是不能确定的；在循环体里面应该包含某些类似于从子程序（subroutine）中返回的语句，才能使得该循环停止。我们舍弃了 Pascal 语句中的 begins 和 ends，而是用缩排和行首空格表示语句的组合。我们也把每一个子程序都看成一个过程（没有返回值的子程序）或者一个函数（有返回值的子程序）。用 return 语句表示一个过程立即返回，而 return x 则表示一个函数返回一个值 x。如果某个子程序可能在其他地方被调用，那么对它进行命名并且在第一行列出它的参数，前面再冠以 procedure 或者 function。在函数定义结尾处的前一行给出函数的返回值。解释性的注释用{}括起来。比如算法 1.2 就是第一页中算法(A)的规范版本。

```
function SequentialSearch(table T[0..n−1], key K): integer
{Return position of K in table T, if it is present, otherwise−1}
    for i from 0 to n−1 do
        if T[i] = K then return i
    return−1
```

算法 1.2　在表 $T[0..n-1]$中顺序查找键 K。

我们的算法能够处理常用的基本数据类型（atomic data type），比如说整型（integers）和布尔型（boolean），还有基本数据类型构成的表。指针类型的值为地址。像上面提到的一样，在某些特殊情况下数据类型也不是特别重要，那么就可以使用一个通用的名字：键（key）。在某些高级语言中，比如 Pascal，键必须是一种具体的数据类型，比如整型；在其他一些语言中有可能将顺序搜索（sequentialsearch）作为一个通用的函数，适用于任意数据类型。对于本书所用的符号，我们假设有经验的程序员能够转换为程序，但是并不能自动转换。

形容“条件 1 and 条件 2”的布尔表达式，只有当“条件 1”和“条件 2”同时为真，原表达式才为真。不过如果“条件 1”为假，那么“条件 2”的判断就“短路”了，即不用再对“条件 2”进行判断。因此可以写一个类似“if $P \neq \Lambda$ and $F(P) \neq \Lambda$ then…”的条件语句，可以确定如果 P 的确为 Λ，那么就不必再查看 P 的 F 字段了。（C 和 Lisp 语言对布尔表达式采用“短路”机制，但 Pascal 没有）。相似地，对于条件语句“条件 1 or 条件 2”，如果“条件 1”为真，那么就不必再判断“条件 2”了。

子程序可以调用自身，称此调用为“递归”（recursive）。递归可以让程序解释起来比较清楚，而且有很多有效的算法是用递归描述的。不过递归程序实现过程中会有一些隐含的代价。特别是在递归调用的过程中需要一个“栈”（stack）来保存变量和参数的值；因为“栈”对程序员来说是透明的，在编码的过程中不需要考虑它，在程序的运行过程中也不需要考虑“栈”占用了大量的存储空间。在 58 页中会继续讨论这个问题。算法 1.3 是二分搜索的递归描述［第 2 页的算法（C）］，是程序中的符号的另外一个例子。

```
founction BinarySearch(table T[a..b], key K: integer
{Return position of K in sorted table T, if it is present, otherwise−1}
    if a > b then return−1
    middle ← ⌊(a+b)/2⌋
    if K = T [middle] then
```

```
        return middle
    else if K < T [middle] then
        return BinarySearch (T [a..middle－1], K)
    else {K > T [middle]}
        return BinarySearch(T [middle＋1..b], K )
```

算法 1.3 在有序表 $T[a..b]$中用二分法查找键 K。

算法 1.3 的调用规则和顺序搜索算法的描述有点不同。因为希望能够将表的任意下界 a 和上界 b 的索引作为一个参数，那么就将该上下界作为表描述的一部分（返回－1 则表示搜索失败，a 和 b 都是非负整数）。表的下界索引有可能超过上界索引，在这种情况下表中没有任何元素。事实上，如果在表中没有找到相应的项，那么就说二分搜索算法搜索了一个 $T[a..a-1]$表，这种情况下递归结束。

本书还引入了一个很有用的符号 $\lfloor x \rfloor$，表示对 x 进行向下取整；比如 $\lfloor 3.4 \rfloor=3$, $\lfloor 3 \rfloor=3$, $\lfloor -3.4 \rfloor=-4$ *。通过这个符号就可以解决表 $T[0..9]$的“中间”元素的问题了；根据算法，$T[0..9]$的中间元素为$\lfloor (0+9)/2 \rfloor$，即元素 4。如果 $T[4]$不是所要找的 K，那么递归调用 *BinarySearch*，输入或者为 $T[0..3]$或者为 $T[5..9]$。

如果某个结构在创建以后，数据量可以增加或者减少，那么就说这样的数据结构是**动态的**（dynamic）；反之如果除了重新创建以外，该结构的数据量不能改变就称其为**“静态的”**（static）。因此链表是动态的，而表则是静态的。处理像链表这样的动态结构的时候，假设存在一个 *NewCell* 程序可以灵活地按需分配任意类型的新单元。把所需的类型作为一个参数；那么 *NewCell*（Node）返回一个存放 Node 的内存块的地址。很多程序语言都提供这种内存管理组件或者程序（比如，Pascal 的 *new* 和 C 的 *malloc*）。在实际情况中，这些程序分配一些当然可以用尽的有限“存储池”块。在描述算法的时候忽略可以这个问题，不过在第 10 章中详细讨论了内存分配中的问题。

```
function NewNode(key K, pointer P): pointer
{Return address of a new cell of type Node containing key K and pointer P}
    Q ← NewCell(Node)
    Key(Q) ← K
    Next(Q)← P
    return Q
```

算法 1.4 创建一个类型为 Node 的新链表单元，并初始化它的两个字段。

定位格

许多需要修改链接结构的算法必须处理这个问题：指针一旦被读过之后就来不及更改它的值了；唯一能修改的只是指针所指向的单元的值。我们现在解释一下这个问题，重新看看前面对链表进行插入操作的例子。其困难在于：“不能在一个项的前面进行插入操作，只能在这个项的后面插入”。具体说来，假设每个记录都有两个字段，Key 字段（Key field）

* 相对的，还有另一个符号$\lceil x \rceil$，$\lceil x \rceil$的向上取整，比如$\lceil 3.4 \rceil=4$,$\lceil 3 \rceil=3$,$\lceil -3.4 \rceil=-3$。

包含一些有序的数据类型比如数字和字符串，Next 字段保存了下一个记录的地址。程序 *NewNode(K,P)*创建一个新的节点，并分别设置 Key 和 Next 字段为 *K* 和 *P*。

在链表插入算法中，变量 *list* 指向链表中的第一个记录；如果链表为空，则 *list*＝Λ。我们希望链表是一个有序链表（那么查找时间就会降低）。函数 *LLInsert* 将 Key 值 *K* 作为参数，并对链表进行修改，增加一个包含 Key 值为 *K* 的新节点。如果链表中已经存在一个这样的节点的话，函数什么也不做；否则，通过调用 *NewNode* 创建一个新的节点，并将该节点放到链表中的适当位置确保修改过的链表仍然能够按序排列。最简单的方式是用一个指针 *P* 访问连续的链表单元来搜索列表；如果 *P* 最后能够指向一个 Key 为 *K* 的记录，那么函数返回。不过如果在 *P* 变为 Λ 之前都没有找到 *K*，或者找到了一个 Key 值在 *K* 以后的记录，则表明 *K* 不在表中。现在如果要插入 *K*，就需要插在 *P* 前一轮的位置处，通常的方法是用另一个变量 *S* 保存 *P* 的前一轮的值（算法 1.5）。

```
procedure LLInsert(key K):
{Insert a cell containing key K in list if none exists already}
    S ← Λ
    P ← list
    while P≠Λ and Key(P)<K do
        S ← P
        P ← Next(P)
    if P≠Λ and Key(P)=K then return
    if S=Λ then                          {Put K at the beginning of the list}
        list ← NewNode(K, list)
else                                     {Insert K after some key already in the list}
        Next(S) ←NewNode(K, P)
```

算法 1.5　在有序链表中插入一个新键值。全局变量 list 保存了链表的首地址。

撇开使用了两个指针变量这个因素，算法 1.5 还有其他两个原因不太令人满意。首先，最后的 if 语句对两个并列的情况进行了不同的编码；仅仅为了确定 *K* 成为链表的头，那么对每次插入都做检查是很麻烦的。其次，算法的代码中包含一个对全局变量 *list* 的引用；这个变量是不能作为一个参数进行传递的，因为它的值可能会改变*。结果就是，如果某个程序中有几个链表，要不就是每个链表有独立的插入过程，要不就是就用一个通用的插入程序，使用一个很复杂的类型变量——“指向链表单元的指针的指针”。

还有其他两种解决该问题的常见方式。创建一个“哑元”节点（dummy）或者叫“头”节点（header）；这种节点不包含 Key 值，而且它的 Next 字段指向真正的链表头。因此，一个不包含 key 的链表仅仅有一个头节点。使用这种方式，算法 1.5 中最后的 if 语句的分支就可以合在一起了。但是仍然需要两个指针，或者等价的字段引用。作为选择，如果程序语言支持，那么算法就可以明确变量 *list* 中的地址和记录中 Next 字段中的地址。对 Pascal 语言来说这是做不到的；C 语言可以通过“取址”（&）和“取内容”（*）操作做到，但是程序会显得很混乱。

为了使得在对这样的算法编码的时候显得更简单一些，本书使用一个新的数据类型

* 在 Pascal 语言中，*list* 可以作为一个 var 参数传递。

“locative”。在多数情况下，locative 很像平常的变量；如果 P 是一个指向某个链表的一个节点的 locative，那么就可以通过 Key(P)和 Next(P)获取其中的 Key 和 Next 字段。不过，当对一个 locative 赋值的时候，不仅需要记住它的值而且需要记住保存这个值的内存位置。比如，假设 P 是一个当前值为 1000 的 locative，表示一个链表节点的地址；假设该节点的 Next 字段中存放的地址为 1002，1002 地址存放的值为 400。那么赋值 $P \leftarrow \text{Next}(P)$ 将值 400 赋给 P，但是还需要记住这个值是从 1002 地址来的。称此与 P 相关的第二条信息为 P 的 locative 值；在本例中，P 的一般值（ordinary value）为 400，P 的 locative 值为 1002。

```
procedure LLInsert(key K, locative P):
{Insert a cell containing key K in list P if none exists already}
    while P≠Λ and Key(P)<K do
        P ← Next(P)
    if P≠Λ and Key(P)=K then return
    P ⇐ NewNode(K, P)
```

算法 1.6 利用定位格在有序链表中插入一个新关键字值。变量 list 指向链表的首部，并且作为该过程的第二个参数进行传递。

只有一个构造使用 locative 值。赋值 $P \Leftarrow Q$ 将 Q 的值赋给 P *的 locative 值处的内存位置*，其中 P 是一个 locative。通过算法 1.6 可以看出其用途，该算法用 locative 对算法 1.5 重新编码。算法 1.6 的实质是对整个链表运行 P，只负责搜索不负责插入。当需要插入操作的时候，P 的 locative 值负责修改相应的单元。

注意在算法 1.6 中链表头的地址作为参数接收，而不是作为一个全局变量接收的。这也说明了 locative 的另一个特点：locative 可以作为函数调用或者过程调用的形参，如同普通赋值中实参到形参一样来确定 locative 值和一般值。比如，如果 *LLInsert*(*K*,*list*)调用算法 1.6，效果如同算法体中一开始就有一个赋值 $P \leftarrow list$；即 P 的一般值变为 list 的值，P 的 locative 值变为 list 的地址。用这种方式，不管 P 的 locative 值是 list 变量的地址还是某个链表单元 Next 字段中的地址，最终的赋值 $P \Leftarrow Q$ 都会产生正确的结果。

Locative 还有三点更重要的性质。第一，如果一个 locative 赋值给另一个 locative，或者作为子程序的形参传递，那么一般值和 locative 值都会从一个传给另一个。第二，任何形如 $P \Leftarrow Q$ 的赋值，都会将 P 的一般值转变为 Q 的一般值，其中 Q 是一个 locative。第三，如果赋值给 locative 赋值运算符 ⇐ 的不是一个 locative，那么它就与一个普通赋值运算符 ← 一样。

虽然 locative 可能看起来并不太熟悉，不过编程的时候可以很容易将其转换为很多传统的高级语言，只利用指向变量的指针，而不关联两个 locative 值。附录 A 详细讨论了 locative 的语义和实现。

抽象数据类型

在完成任务之前清楚地理解任务是程序设计的一个基本原则。这条原则的一个平常例子就是选择数据表示方式的时候：在选择数据表示之前判定对数据的操作。