数据结构与算法分析

左元

2023 年 5 月 6 日

目录

第一部分　基础知识 **5**

第一章　数学知识复习 **7**
1.1　单调性 7
1.2　取整 7
1.3　指数 8
1.4　对数 8
1.5　级数 9
1.6　模运算 11
1.7　常用的数学证明方法 11
1.7.1　归纳法 11
1.7.2　反证法 13

第二章　算法在计算中的作用 **14**
2.1　算法 14
2.2　作为一种技术的算法 17

第三章　算法基础 **20**
3.1　插入排序 20
3.2　分析算法 24
3.3　如何刻画运行时间 29
3.3.1　大 O 表示法, 大 Ω 表示法和大 Θ 表示法 30
3.3.2　渐进表示法: 形式化定义 32
3.4　斐波那契数列和递归 35
3.5　二分查找 38
3.6　设计算法 40
3.6.1　分治法 40
3.6.2　分析归并排序算法 44

第四章 力扣算法题选讲 **50**

第二部分 排序和顺序统计量 **51**

第五章 堆排序 **52**
5.1 堆 (Heap) 52
5.2 保持堆的性质 53
5.3 建堆 54
5.4 堆排序算法 56

第六章 快速排序 **59**
6.1 快速排序的描述 59
6.2 快速排序的性能 62
6.3 随机化版本的快速排序 62
6.4 快速排序分析 63

第七章 基于比较的排序算法的下界 **64**

第八章 计数排序 **65**

第九章 中位数和顺序统计量 **66**
9.1 最小值和最大值 66
9.2 期望为线性时间的快速选择算法 66

第三部分 数据结构 **69**

第十章 基本数据结构 **70**
10.1 基于数组实现的数据结构: 数组, 矩阵, 栈和队列 70
10.1.1 数组 70
10.1.2 矩阵 72
10.1.3 栈 72
10.1.4 队列 74
10.2 链表 76
10.3 指针和对象的实现 80
10.4 有根树的表示方式 80

第十一章 哈希表 **82**

11.1 直接寻址表 82
11.2 哈希表 83
11.3 哈希函数 83
11.4 开放定址 83
11.5 实际中的考虑 83

第十二章 二叉搜索树 **84**
12.1 什么是二叉搜索树 84
12.2 二叉搜索树的查找 86
12.3 二叉搜索树的插入和删除 88

第十三章 红黑树 **91**
13.1 红黑树的性质 91
13.2 旋转 95
13.3 插入 98
13.4 删除 106
13.5 红黑树的测试和漂亮打印 114

第十四章 B 树 **116**
14.1 B 树的定义 120
14.2 B 树的基本操作 120
14.3 从 B 树中删除关键字 120

第四部分 图算法 **124**

第十五章 图的表示 **126**

第十六章 广度优先搜索 **130**

第十七章 深度优先搜索 **134**

第十八章 拓扑排序 **136**

第五部分 高级设计和分析技术 **137**

第十九章 组合搜索问题 **138**

第二十章 动态规划 **139**

第二十一章 贪心算法 **140**

第二十二章 字符串匹配和确定性有限状态自动机 **141**

第二十三章 正则表达式和非确定性有限状态自动机 **142**

第一部分

基础知识

这一部分将引导读者开始思考算法的设计和分析问题, 简单介绍算法的表达方法, 将在本书中用到的一些设计策略, 以及算法分析中用到的许多基本思想. 本课程后面的内容都是建立在这些基础知识之上的.

第二章是对算法及其在现代计算系统中地位的一个综述. 本章给出了算法的定义和一些算法的例子. 此外, 本章还说明了算法是一项技术, 就像快速的硬件、图形用户界面、面向对象系统和网络一样.

在第三章中, 我们给出了书中的第一批算法, 它们解决的是对 n 个数进行排序的问题. 这些算法是用一种伪代码形式给出的, 这种伪代码尽管不能直接翻译为任何常规的程序设计语言, 但是足够清晰地表达了算法的结构, 以便任何一位能力比较强的程序员都能用自己选择的语言将算法实现出来. 我们分析的排序算法是插入排序, 它采用了一种增量式的做法; 另外还分析了归并排序, 它采用了一种递归技术, 称为 “分治法”. 尽管这两种算法所需的运行时间都随 n 的值而增长, 但增长的速度是不同的. 我们在第 2 章分析了这两种算法的运行时间, 并给出了一种有用的 “渐进” 表示法来表达这些运行时间.

第 3 章给出了这种表示法的准确定义, 称为渐近表示. 在第 3 章的一开始, 首先定义几种渐近符号, 它们主要用于表示算法运行时间的上界和下界. 第 3 章余下的部分主要给出了一些数学表示方法. 这一部分的作用更多的是为了确保读者所用的记号能与本书的记号体系相匹配, 而不是教授新的数学概念.

第一章　数学知识复习

本章回顾一些以前学习过的数学知识.

1.1　单调性

如果 $m \le n \implies f(m) \le f(n)$, 则函数 $f(n)$ 是**单调递增**的. 类似地, 如果 $m \le n \implies f(m) \ge f(n)$, 则函数 $f(n)$ 是**单调递减的**. 如果 $m < n \implies f(m) < f(n)$, 则函数 $f(n)$ 是**严格递增**的. 如果 $m < n \implies f(m) > f(n)$, 则函数 $f(n)$ 是**严格递减**的.

1.2　取整

对任意实数 x, 我们用 $\lfloor x \rfloor$ 表示小于或者等于 x 的最大整数 (读作 "x 的向下取整"), 并用 $\lceil x \rceil$ 表示大于或者等于 x 的最小整数 (读作 "x 的向上取整"). 对所有实数 x,

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1 \tag{1.1}$$

对任意整数 n,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

1.3　指数

$$x^a x^b = x^{a+b}$$
$$\frac{x^a}{x^b} = x^{a-b}$$
$$(x^a)^b = x^{ab}$$
$$x^n + x^n = 2x^n \neq x^{2n}$$
$$2^n + 2^n = 2^{n+1}$$

1.4　对数

在计算机科学中, 除非有特别的声明, 否则所有的对数都是**以 2 为底**的.

定义 1.1. $x^a = b$当且仅当 $\log_x b = a$.

由该定义可以得到几个方便的等式.

定理 1.1. $\log_a b = \frac{\log_c b}{\log_c a}; a, b, c > 0, a \neq 1, c \neq 1$

证明. 令 $x = \log_c b, y = \log_c a$, 以及$z = \log_a b$. 此时由对数的定义, $c^x = b, c^y = a$以及$a^z = b$, 联合这三个等式则产生 $(c^y)^z = c^x = b$. 因此, $x = yz$, 这意味着 $z = x/y$, 定理得证.

定理 1.2. $\log ab = \log a + \log b; a, b > 0$

证明. 令 $x = \log b, y = \log a$, 以及$z = \log ab$. 此时由于我们假设默认的底为 $2, 2^x = a, 2^y = b$, 及 $2^z = ab$, 联合最后的三个等式则有 $2^x 2^y = 2^z = ab$. 因此 $x + y = z$, 定理得证.

其它一些有用的公式如下, 它们都能够用类似的方法推导.

- $\log \frac{a}{b} = \log a - \log b$
- $\log (a^b) = b \log a$
- $\log x < x$对所有的$x > 0$成立
- $\log 1 = 0, \log 2 = 1, \log 1024 = 10, \log 1048576 = 20$

1.5　级数

最容易记忆的公式是

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

和

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

在第二个公式中, 如果 $0 < A < 1$, 则

$$\sum_{i=0}^{n} a^i \leq \frac{1}{1 - a}$$

当 n 趋近于 ∞ 时, 上面的和趋近于 $1/(1 - a)$, 这些公式是**几何级数**公式.

我们可以用下面的方法推导关于 $\sum_{i=0}^{\infty} a^i (0 < a < 1)$ 的公式. 令 s 是其和. 此时

$$s = 1 + a + a^2 + a^3 + a^4 + a^5 + \cdots$$

于是

$$as = a + a^2 + a^3 + a^4 + a^5 + \cdots$$

如果我们将这两个方程相减 (这种运算只允许对收敛级数进行), 等号右边的所有的项相消, 只留下 1:

$$s - as = 1$$

即

$$s = \frac{1}{1 - a}$$

可以用相同的方法计算 $\sum_{i=1}^{\infty} i/2^i$, 它是一个经常出现的和. 我们写成

$$s = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \cdots$$

用 2 乘之得到

$$2s = 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} + \frac{6}{2^5} + \cdots$$

将这两个方程相减得到

$$s = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots$$

因此, $s = 2$.

算法分析中另一种常用类型的级数是**算术级数**. 任何这样的级数都可以从基本公式计算它的值.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

例如, 为求出和 $2 + 5 + 8 + \cdots + (3k-1)$, 将其改写为 $3(1 + 2 + 3 + \cdots + k) - (1 + 1 + 1 + \cdots + 1)$. 显然, 它就是 $3k(k+1)/2 - k$. 另一种记忆的方法则是将第一项与最后一项相加 (和为 $3k+1$), 第二项与倒数第二项相加 (和也是 $3k+1$), 等等. 由于有 $k/2$ 个这样的数对, 因此总和就是 $k(3k+1)/2$, 这与前面的答案相同.

还有下面两个不太常见的公式

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{|k+1|} \quad \text{当} k \neq -1$$

当 $k = -1$ 时, 后一个公式不成立. 此时, 我们需要下面的公式. 这个公式在计算机科学中的使用频率要比在其它数学学科中高很多. H_N 叫做**调和级数**, 求和结果叫做**调和和**. 下面近似式中的误差趋近于 $\gamma \approx 0.57721566$, 成为**欧拉常数** (Euler's constant).

$$h_n = \sum_{i=1}^{n} \frac{1}{i} \approx \log_e n$$

以下两个公式是常见的代数运算的结果:

$$\sum_{i=1}^{n} f(n) = nf(n)$$

$$\sum_{i=n_0}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{n_0-1} f(i)$$

1.6 模运算

模运算也就是求余运算. 如果 n 整除 $a-b$, 那么就说 a 与 b 模 n 同余, 记为 $a \equiv b \pmod{n}$. 直观地看, 这意味着无论是 a 还是 b 被 n 去除, 所得余数都是相同的. 于是, $81 \equiv 61 \equiv 1 \pmod{10}$. 如同等号的情形一样, 则 $a+c \equiv b+c \pmod{n}$ 以及 $ad \equiv bd \pmod{n}$.

1.7 常用的数学证明方法

在数据结构与算法分析中, 最常用的证明方法有两种:

- 归纳法
- 反证法

1.7.1 归纳法

由归纳法进行的证明有两个标准的部分.

1. 证明**基准情形** (base case), 就是确定定理对于某个 (某些) 小的 (通常是退化的) 值的正确性.
2. 进行**归纳假设** (inductive hypothesis). 一般来说, 它指的是假设定理对直到某个有限的数 k 的所有的情况都是成立的. 然后使用这个假设证明定理对下一个值 (通常是 $k+1$) 也是成立的. 至此定理得证 (在 k 是有限的情形下).

举个例子, 我们证明一下斐波那契数, $F_0 = 1, F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, \cdots, F_i = F_{i-1} + F_{i-2}$, 满足对 $i \geq 1$, 有 $F_i < (5/3)^i$. 为了证明这个不等式, 我们首先验证定理对简单的情形成立. 容易验证 $F_1 = 1 < 5/3$ 及 $F_2 = 2 < 25/9$. 这就证明了基准情形. 假设定理对于 $i = 1, 2, \cdots, k$ 成立. 这就是归纳假设. 为了证明定理, 我们需要证明 $F_{k+1} < (5/3)^{k+1}$. 根据定义可得

$$F_{k+1} = F_k + F_{k-1}$$

将归纳假设用于等号右边, 得到

$$\begin{aligned}
F_{k+1} &< (5/3)^k + (5/3)^{k-1} \\
&= (3/5)(5/3)^{k+1} + (3/5)^2(5/3)^{k+1} \\
&= (3/5)(5/3)^{k+1} + (9/25)(5/3)^{k+1} \\
&= (3/5 + 9/25)(5/3)^{k+1} \\
&= (24/25)(5/3)^{k+1} \\
&< (5/3)^{k+1}
\end{aligned}$$

这就完成了证明.

我们再来举一个例子, 我们将建立以下定理.

定理 1.3. 如果 $N \geq 1$, 则 $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$

证明. 用数学归纳法证明. 对于基准情形, 容易看到, 当 $N = 1$ 时定理成立. 对于归纳假设, 设定理对 $1 \leq k \leq N$ 成立. 我们将在该假设下证明定理对于 $N+1$ 也是成立的. 我们有

$$\sum_{i=1}^{N+1} i^2 = \sum_{i=1}^{N} i^2 + (N+1)^2$$

应用归纳假设得到

$$\begin{aligned}
\sum_{i=1}^{N+1} i^2 &= \frac{N(N+1)(2N+1)}{6} + (N+1)^2 \\
&= (N+1)[\frac{N(2N+1)}{6} + (N+1)] \\
&= (N+1)\frac{(2N^2+7N+6)}{6} \\
&= \frac{(N+1)(N+2)(2N+3)}{6}
\end{aligned}$$

因此

$$\sum_{i=1}^{N+1} i^2 = \frac{(N+1)[(N+1)+1][2(N+1)+1]}{6}$$

定理得证.

1.7.2 反证法

反证法证明是通过假设定理不成立, 然后证明该假设导致某个已知的性质不成立, 从而证明原假设是错误的. 一个经典的例子是证明存在无穷多个素数. 为了证明这个结论, 我们假设定理不成立. 于是, 存在某个最大的素数 P_k. 令 $P_1, P_2, \cdots, P_k$ 是依序排列的所有素数并考虑

$$N = P_1 P_2 P_3 \cdots P_k + 1$$

显然, N 是比 P_k 大的数, 根据假设 N 不是素数. 可是, $P_1, P_2, \cdots, P_k$ 都不能整除 N, 因为除得的结果总有余数 1. 这就产生了一个矛盾, 因为每一个整数或者是素数, 或者是素数的乘积. 因此, P_k 是最大素数的原假设是不成立的, 定理得证.

再举一个例子, 如何证明 $\sqrt{2}$ 是无理数. 我们采用反证法证明. 假设 $\sqrt{2}$ 是有理数, 那么 $\sqrt{2}$ 可以写成最简分数形式 n/d.

两边同时平方, 得 $2 = n^2/d^2$, 有 $2d^2 = n^2$. 可以知道 n 是 2 的倍数. 所以 n^2 一定是 4 的倍数. 而 $2d^2 = n^2$, 可以知道 $2d^2$ 是 4 的倍数, 所以 d^2 是 2 的倍数. 因此 d 是 2 的倍数.

所以, 分子和分母都有公因子 2, 这与 n/d 是最简分数形式相矛盾. 因此, $\sqrt{2}$ 一定是无理数. 定理得证.

第二章　算法在计算中的作用

什么是算法？为什么算法值得研究？相对于计算机中使用的其他技术来说算法的作用是什么？本章我们将回答这些问题.

2.1　算法

非形式地说, **算法** (algorithm) 就是任何经过良好的定义的计算过程, 这个过程取某个值或值的集合作为**输入**并在有限的时间内产生某个值或值的集合作为**输出**. 这样算法就是把输入转换成输出的计算步骤的一个序列.

我们也可以把算法看成是用于求解良说明的**计算问题**的工具. 一般来说, 问题陈述说明了期望的输入/输出关系. 算法则描述一个特定的计算过程来实现该输入/输出关系.

例如, 我们可能需要把一个数列排成非递减序. 实际上, 这个问题经常出现, 并且为引入许多标准的设计技术和分析工具提供了足够的理由. 下面是我们关于**排序问题**的形式化定义.

输入: N 个数的一个序列 $\langle a_1, a_2, \ldots, a_n\rangle$.

输出: 输入序列的一个排列 $\langle a'_1, a'_2, \ldots, a'_n\rangle$, 满足$\langle a'_0 \leq a'_1 \leq \cdots a'_n\rangle$.

例如, 给定输入序列 $\langle 31, 41, 59, 26, 41, 58\rangle$, 排序算法将返回序列 $\langle 26, 31, 41, 41, 58, 59\rangle$ 作为输出. 这样的输入序列称为排序问题的一个**实例** (instance). 一般来说, 问题实例由计算该问题解所必需的 (满足问题陈述中强加的各种约束的) 输入组成.

因为许多程序使用排序作为一个中间步, 所以排序是计算机科学中的一个基本操作. 因此, 已有许多好的排序算法供我们任意使用. 对于给定应用, 哪个算法最好依赖于以下因素: 将被排序的项数, 这些项已被稍微排序的程度, 关于项值的可能限制, 计算机的体系结构, 以及将使用的存储设备的种类 (主存, 磁盘或者磁带).

若对每个输入实例算法都以正确的输出**停机** (halt)–在有限的时间内结束,

则称该算法是正确的, 并称正确的算法解决了给定的计算问题. 不正确的算法对某些输入实例可能根本不停机, 也可能以不正确的回答停机. 与人们期望的相反, 不正确的算法只要其错误率可控有时可能是有用的. 但是通常我们只关心正确的算法.

算法可以用自然语言来说明, 也可以说明成计算机程序, 甚至说明成硬件设计. 唯一的要求是这个说明必须精确描述所要遵循的计算过程.

算法解决哪种问题

排序绝不是已开发算法的唯一计算问题, 算法的实际应用无处不在, 包括以下例子:

- 人类基因工程已经取得重大进展, 其目标是识别人类 DNA 中的所有 10 万个基因, 确定构成人类 DNA 的 30 亿个化学基对的序列, 在数据库中存储这类信息并为数据分析开发工具. 这些工作都需要复杂的算法. 虽然对涉及的各种问题的求解超出了本课程的范围, 但是求解这些生物问题的许多方法采用了本课程多章内容的思想, 从而使得科学家能够有效地使用资源以完成任务. 因为可以从实验技术中提取更多的信息, 所以既能节省人和机器的时间又能节省金钱.

- 互联网使得全世界的人都能快速地访问与检索大量信息. 借助于一些聪明的算法, 互联网上的网站能够管理和处理这些海量数据. 必须使用算法的问题示例包括为数据传输寻找好的路由, 使用一个搜索引擎来快速地找到特定信息所在的网页.

- 电子商务使得货物与服务能够以电子方式洽谈与交换, 并且它依赖于像信用卡号, 密码和银行结单这类个人信息的保密性. 电子商务中使用的核心技术包括公钥密码与数字签名, 它们以数值算法和数论为基础.

虽然这些例子的一些细节已超出本书的范围, 但是我们确实说明了一些适用于这些问题和问题领域的基本技术. 我们还说明如何求解许多具体问题, 包括以下问题:

- 给定一张交通图, 上面标记了每对相邻十字路口之间的距离, 我们希望确定从一个十字路口到另一个十字路口的最短道路. 即使不允许穿过自身的道路, 可能路线的数量也会很大. 在所有可能路线中, 我们如何选择哪一条是最短的? 这里首先把交通图 (它本身就是实际道路的一个模型) 建模为一个图, 然后寻找图中从一个顶点到另一个顶点的最短路径.

- 给定两个有序的符号序列 $X = \langle x_1, x_2, \ldots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \ldots, y_n \rangle$, 求出 X 和 Y 的最长公共子序列. X 的子序列就是去掉一些元素 (可能是所

有, 也可能一个没有) 后的 X. 例如, $\langle A, B, C, D, E, F, G\rangle$ 的一个子序列是 $\langle B, C, E, G\rangle$. X 和 Y 的最长公共子序列的长度度量了这两个序列的相似程序. 例如, 若两个序列是 DNA 链中的基对, 则当它们具有长的公共子序列时我们认为它们是相似的. 若 X 有 m 个符号且 Y 有 n 个符号, 则 X 和 Y 分别有 2 个可能的子序列. 除非 m 和 n 很小, 否则选择 X 和 Y 的所有可能子序列做匹配将花费使人望而却步多的时间. 我们将使用**动态规划**技术来解决这个问题.

- 给定一个依据部件库的机械设计, 其中每个部件可能包含其他部件的实例, 我们需要依次列出这些部件, 以使每个部件出现在使用它的任何部件之前. 若该设计由 n 个部件组成, 则存在 $n!$ 种可能的顺序, 其中 $n!$ 表示阶乘函数. 因为阶乘函数甚至比指数函数增长还快, (除非我们只有几个部件, 否则) 先生成每种可能的顺序再验证按该顺序每个部件出现在使用它的部件之前, 是不可行的. 我们将使用**拓扑排序**算法来解决这个问题.

虽然这些问题的列表还远未穷尽, 但是它们却展示了许多有趣的算法问题所共有的两个特征:

1. 存在许多候选解, 但绝大多数候选解都没有解决手头的问题. 寻找一个真正的解或一个最好的解可能是一个很大的挑战.

2. 存在实际应用. 在上面所列的问题中, 最短路径问题提供了最易懂的例子. 一家运输公司 (如公路运输或铁路运输公司) 对如何在公路或铁路网中找出最短路径, 有着经济方面的利益, 因为采用的路径越短, 其人力和燃料的开销就越低. 互联网上的一个路由结点为了快速地发送一条消息可能需要寻找通过网络的最短路径. 希望从北京开车去上海的人可能想从一个恰当的网站寻找开车方向, 或者开车时她可能使用其 GPS.

数据结构

本课程也包含几种数据结构. 数据结构是一种存储和组织数据的方式, 旨在便于访问和修改. 没有一种单一的数据结构对所有用途均有效, 所以重要的是知道几种数据结构的优势和局限.

技术

虽然可以把本课程当做一本有关算法的 “菜谱” 来使用, 但是也许在某一天你会遇到一个问题, 一时无法很快找到一个已有的算法来解决它. 本课程将教你一些算法设计与分析的技术, 以便你能自行设计算法, 证明其正确性和理解其效率. 不同的章介绍算法问题求解的不同方面.

2.2　作为一种技术的算法

假设计算机是无限快的并且计算机存储器是免费的, 你还有什么理由来研究算法吗? 即使只是因为你还想证明你的解法会终止并以正确的答案终止, 那么回答也是肯定的.

如果计算机无限快, 那么用于求解某个问题的任何正确的方法都行. 也许你希望你的实现在好的软件工程实践的范围内 (例如, 你的实现应该具有良好的设计与文档), 但是你最常使用的是最容易实现的方法.

当然, 计算机也许是快的, 但它们不是无限快. 存储器也许是廉价的, 但不是免费的. 所以计算时间是一种有限资源, 存储器中的空间也一样. 你应该明智地使用这些资源, 在时间或空间方面有效的算法将帮助你这样使用资源.

效率

为求解相同问题而设计的不同算法在效率方面常常具有显著的差别. 这些差别可能比由于硬件和软件造成的差别要重要得多.

作为一个例子, 第三章将介绍两个用于排序的算法. 第一个称为插入排序, 为了排序 n 个项, 该算法所花时间大致等于 $c_1 n^2$, 其中 c_1 是一个不依赖于 n 的常数. 也就是说, 该算法所花时间大致与 n^2 成正比. 第二个称为归并排序, 为了排序 n 个项, 该算法所花时间大致等于 $c_2 n \lg n$, 其中 $\lg n$ 代表 $\log_2 n$ 且 c_2 是另一个不依赖于 n 的常数. 与归并排序相比, 插入排序通常具有一个较小的常数因子, 所以 $c_1 < c_2$. 我们将看到就运行时间来说, 常数因子可能远没有对输入规模 n 的依赖性重要. 把插入排序的运行时间写成 $c_1 n \cdot n$ 并把归并排序的运行时间写成 $c_2 n \cdot \lg n$. 这时就运行时间来说, 插入排序有一个因子 n 的地方归并排序有一个因子 $\lg n$, 后者要小得多. (例如, 当 $n = 1000$ 时, $\lg n$ 大致为 10, 当 n 等于 100 万时, $\lg n$ 大致仅为 20.) 虽然对于小的输入规模, 插入排序通常比归并排序要快, 但是一旦输入规模 n 变得足够大, 归并排序 $\lg n$ 对 n 的优点将足以补偿常数因子的差别. 不管 c_1 比 c_2 小多少, 总会存在一个交叉点, 超出这个点, 归并排序更快.

作为一个具体的例子, 我们让运行插入排序的一台较快的计算机 (计算机 A) 与运行归并排序的一台较慢的计算机 (计算机 B) 竞争. 每台计算机必须排序一个具有 1000 万个数的数组.(虽然 1000 万个数似乎很多, 但是, 如果这些数是 8 字节的整数, 那么输入将占用大致 80MB, 即使一台便宜的便携式计算机的存储器也能多次装入这么多数.) 假设计算机 A 每秒执行百亿条指令 (快于写本书时的任何单台串行计算机), 而计算机 B 每秒仅执行 1000 万条指令, 结果计算机 A 就纯计算能力来说比计算机 B 快 1000 倍. 为使差别更具戏剧性, 假设世上最巧妙的程序员为计算机 A 用机器语言编码插入排序, 并且为了排序 n 个数, 结

果代码需要 $2n^2$ 条指令. 进一步假设仅由一位水平一般的程序员使用某种带有一个低效编译器的高级语言来实现归并排序, 结果代码需要 $50n \lg n$ 条指令. 为了排序 1000 万个数, 计算机 A 需要

$$\frac{2 \cdot (10^7)^2 \text{条指令}}{10^10 \text{条指令/秒}} = 20000 \text{秒 (多于 5.5 小时)}$$

而计算机 B 需要

$$\frac{50 \cdot 10^7 \lg 10^7 \text{条指令}}{10^10 \text{条指令/秒}} \approx 1163 \text{秒 (少于 20 分钟)}$$

通过使用一个运行时间增长较慢的算法, 即使采用一个较差的编译器, 计算机 B 比计算机 A 还快 17 倍! 当我们排序 1 亿个数时, 归并排序的优势甚至更明显: 这时插入排序需要 23 天多, 而归并排序不超过 4 小时. 一般来说, 随着问题规模的增大, 归并排序的相对优势也会增大.

算法与其他技术

上面的例子表明我们应该像计算机硬件一样把算法看成是一种技术. 整个系统的性能不但依赖于选择快速的硬件而且还依赖于选择有效的算法. 正如其他计算机技术正在快速推进一样, 算法也在快速发展.

你也许想知道相对其他先进的计算机技术 (如以下列出的), 算法对于当代计算机是否真的那么重要:

- 先进的计算机体系结构与制造技术
- 易于使用, 直观的图形用户界面 (GUI)
- 面向对象的系统
- 集成的万维网技术
- 有线与无线网络的快速组网

回答是肯定的. 虽然某些应用在应用层不明确需要算法内容 (如某些简单的基于万维网的应用), 但是许多应用确实需要算法内容. 例如, 考虑一种基于万维网的服务, 它确定如何从一个位置旅行到另一个位置. 其实现依赖于快速的硬件, 一个图形用户界面, 广域网, 还可能依赖于面向对象技术. 然而, 对某些操作, 如寻找路线 (可能使用最短路径算法), 描绘地图, 插入地址, 它还是需要算法.

而且, 即使是那些在应用层不需要算法内容的应用也高度依赖于算法. 该应用依赖于快速的硬件吗? 硬件设计用到算法. 该应用依赖于图形用户界面吗? 任何图形用户界面的设计都依赖于算法. 该应用依赖于网络吗? 网络中的路由高

度依赖于算法. 该应用采用一种不同于机器代码的语言来书写吗? 那么它被某个编译器, 解释器或汇编器处理过, 所有这些都广泛地使用算法. 算法是当代计算机中使用的大多数技术的核心.

进一步, 随着计算机能力的不断增强, 我们使用计算机来求解比以前更大的问题. 正如我们在上面对插入排序与归并排序的比较中所看到的, 正是在较大问题规模时, 算法之间效率的差别才变得特别显著.

是否具有算法知识与技术的坚实基础是区分真正熟练的程序员与初学者的一个特征. 使用现代计算技术, 如果你对算法懂得不多, 你也可以完成一些任务, 但是, 如果有一个好的算法背景, 那么你可以做的事情就多得多.

第三章　算法基础

本章将要介绍一个贯穿本书的框架, 后续的算法设计与分析都是在这个框架中进行的. 这一部分内容基本上是独立的, 但也有对第 3 章和第 4 章中一些内容的引用.

首先, 我们考察求解第 1 章中引入的排序问题的插入排序算法. 我们定义一种对于已经编写过计算机程序的读者来说应该熟悉的 “伪代码”, 并用它来表明我们将如何说明算法. 然后, 在说明了插入排序算法后, 我们将证明该算法能正确地排序并分析其运行时间. 这种分析引入了一种记号, 该记号关注时间如何随着将被排序的项数而增加. 在讨论完插入排序之后, 我们引入用于算法设计的分治法并使用这种方法开发一个称为归并排序的算法. 最后, 我们分析归并排序的运行时间.

3.1　插入排序

我们的第一个算法 (插入排序) 求解第二章中引入的排序问题:

输入: n 个数的一个序列 $\langle a_0, a_1, \ldots, a_{n-1}\rangle$.

输出: 输入序列的一个排列 $\langle a'_0, a'_1, \ldots, a'_{n-1}\rangle$, 满足 $\langle a'_0 \le a'_1 \le \cdots \le a'_{n-1}\rangle$.

我们希望排序的数也称为 **key**. 虽然概念上我们在排序一个序列, 但是输入是以 n 个元素的数组的形式出现的.

我们首先介绍插入排序, 对于少量元素的排序, 它是一个有效的算法. 插入排序的工作方式像许多人排序一手扑克牌. 开始时, 我们的左手为空并且桌子上的牌面向下. 然后. 我们每次从桌子上拿走一张牌并将它插入左手中正确的位置. 为了找到一张牌的正确位置, 我们从右到左将它与已在手中的每张牌进行比较. 拿在左手上的牌总是排序好的, 原来这些牌是桌子上牌堆中顶部的牌.

插入排序程序接收一个参数: 一个参数是数组 A, 包含了将被排序的数组元素. 数组元素占据 $A[0]$ 到 $A[n-1]$ 的位置, 我们表示为 $A[0:n-1]$. 当插入排序程序执行完毕时, 数组 $A[0:n-1]$ 包含了已经排过序的原始数组中的所有元

素.

程序 3.1: 插入排序

```java
public class InsertionSortAlgorithm {
    public static void InsertionSort(int[] A) {
        for (int i = 1; i < A.length; i++) {
            int key = A[i];
            // 将 A[i] 插入到已经排好序的子数组 A[0:i-1]
            int j = i - 1;
            while (j > -1 && A[j] > key) {
                A[j + 1] = A[j];
                j = j - 1;
            }
            A[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] A = {5,2,4,6,1,3};
        InsertionSort(A);
        for (var i : A)
            System.out.println(i);
    }
}
```

循环不变量与插入排序的正确性

图3.2表明对 $A = \langle 5, 2, 4, 6, 1, 3\rangle$ 数组插入排序算法如何工作. 下标 i 指出正被插入到手中的 "当前牌". 在 **for** 循环 (循环变量为 i) 的每次迭代的开始, 包含元素 $A[0 : i—1]$ 的**子数组**构成了当前排序好的左手中的牌, 剩余的子数组 $A[i + 1 : n - 1]$ 对应于仍在桌子上的牌堆. 事实上, 元素 $A[0 : i—1]$ 就是原来在位置 0 到 $i—1$ 的元素, 但现在已按序排列. 我们把 $A[0 : i - 1]$ 的这些性质形式地表示为一个**循环不变量**:

在每次 **for** 迭代 (3-13 行) 开始时, 子数组 $A[0 : i - 1]$ 包括了最开始在 $A[0 : i - 1]$ 中的数组元素, 但是已经经过了排序.

循环不变量主要用来帮助我们理解算法的正确性. 关于循环不变量. 我们必须证明三条性质:

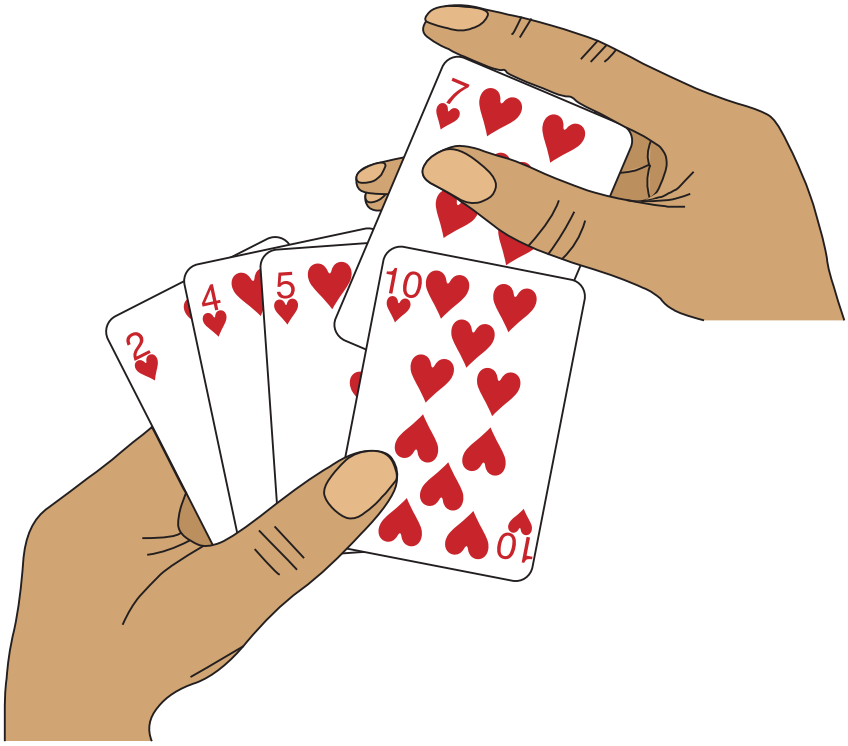


图 3.1: 插入排序的纸牌示意图

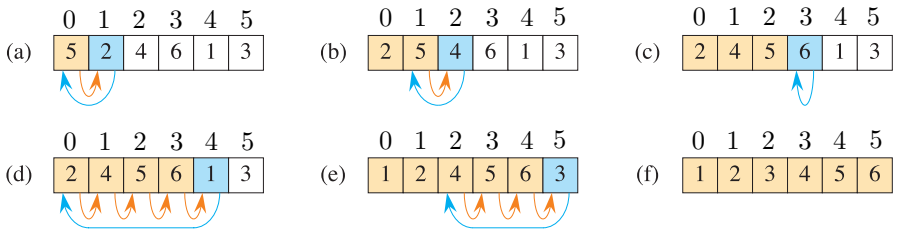


图 3.2: 插入排序程序 $InsertionSort(A)$ 的工作过程. A 最开始包含序列 $\langle 5, 2, 4, 6, 1, 3\rangle$. 数组索引位于数组上方. 数组元素在方格中. **(a)-(e)** 是 **for** 循环 3-13 行的每一次迭代. 在每次迭代时, 蓝色方格里的是 $key = A[i]$, 代码的第 7 行会将 key 和之前的浅黄色的方格中的数进行比较. 橙色的箭头表示比 key 大的数值向右移动的过程 (第 8 行), 蓝色的箭头表示 key 的移动 (第 11 行). **(f)** 最终排好序的数组.

初始化: 循环的第一次迭代之前. 它为真.

保持: 如果循环的某次迭代之前它为真. 那么下次迭代之前它仍为真.

终止: 在循环终止时. 不变量为我们提供一个有用的性质. 该性质有助于证明算法是正确的.

当前两条性质成立时. 在循环的每次迭代之前循环不变量为真.(当然, 为了证明循环不变量在每次迭代之前保持为真, 我们完全可以使用不同于循环不变量本身的其它已证实的事实.) 注意, 这类似于数学归纳法, 其中为了证明某条性质成立, 需要证明一个**基准情况**和一个**归纳步骤**. 这里, 证明第一次迭代之前不变量成立对应于基准情况, 证明从一次迭代到下一次迭代不变量成立对应于归纳步骤.

第三条性质也许是最重要的. 因为我们将使用循环不变量来证明程序的正确性. 通常, 我们和导致循环终止的条件一起使用循环不变量, 终止性不同于我们通常使用数学归纳法的做法, 在归纳法中, 归纳步骤是无限地使用的, 这里当循环终止时, 停止 “归纳”.

让我们看看对于插入排序, 如何证明这些性质成立.

初始化: 首先证明在第一次循环迭代之前 (当 $i = 1$ 时), 循环不变量成立. 所以子数组 $A[0 : i - 1]$ 仅由单个元素 $A[0]$ 组成, 实际上就是 $A[0]$ 中原来的元素. 而且该子数组是排序好的 (当然很平凡). 这表明第一次循环迭代之前循环不变量成立.

保持: 其次处理第二条性质: 证明每次迭代保持循环不变量. 非形式化地,

for 循环体的第 6-10 行将 $A[i-1], A[i—2], A[i-3]$ 等向右移动一个位置, 直到找到 $A[i]$ 的适当位置, 第 11 行将 $A[i]$ 的值插入该位置. 这时子数组 $A[0:i]$ 由原来在 $A[0:i]$ 中的元素组成, 但已按序排列. 那么对 **for** 循环的下一次迭代增加 i 将保持循环不变量.

第二条性质的一种更形式化的处理要求我们对第 7-10 行的 **while** 循环给出并证明一个循环不变量. 然而, 这里我们不愿陷入形式主义的困境, 而是依赖以上非形式化的分析来证明第二条性质对外层循环成立.

终止: 最后研究在循环终止时发生了什么. 导致 **for** 循环终止的条件是 $i > n-1$. 因为每次循环迭代 i 增加 1, 那么循环终止时, 必有 $i = n$. 在循环不变量的表述中将 i 用 n 代替, 我们有: 子数组 $A[0:n-1]$ 由原来在 $A[0:n-1]$ 中的元素组成, 但已按序排列. 注意到, 子数组 $A[0:n-1]$ 就是整个数组, 我们推断出整个数组已排序. 因此算法正确.

在本章后面以及其他章中, 我们将采用这种循环不变量的方法来证明算法的正确性.

3.2　分析算法

分析算法的结果意味着预测算法需要的资源. 虽然有时我们主要关心像内存, 通信带宽或计算机硬件这类资源, 但是通常我们想度量的是计算时间. 一般来说, 通过分析求解某个问题的几种候选算法, 我们可以选出一种最有效的算法. 这种分析可能指出不止一个可行的候选算法, 但是在这个过程中, 我们往往可以抛弃几个较差的算法.

在能够分析一个算法之前, 我们必须有一个要使用的实现技术的模型, 包括描述所用资源及其代价的模型. 对本教程的大多数章节, 我们假定一种通用的单处理器计算模型**随机访问机** (random-access machine, RAM) 来作为我们的实现技术, 算法可以用计算机程序来实现. 在 RAM 模型中, 指令一条接一条地执行, 没有并发操作. 严格地说, 我们应该精确地定义 RAM 模型的指令及其代价. 然而, 这样做既乏味又对算法的设计与分析没有多大意义. 我们还要注意不能滥用 RAM 模型. 例如, 如果一台 RAM 有一条排序指令, 会怎样呢? 这时, 我们只用一条指令就能排序. 这样的 RAM 是不现实的, 因为真实的计算机并没有这样的指令. 所以, 我们的指导性意见是真实计算机如何设计, RAM 就如何设计. RAM 模型包含真实计算机中常见的指令: 算术指令 (如加法, 减法, 乘法, 除法, 取余, 向下取整, 向上取整), 数据移动指令 (加载, 存储, 复制) 和控制指令 (条件与无条件转移, 子程序调用与返回). 每条这样的指令所需时间都为常量.

RAM 模型中的数据类型有整数型和浮点实数型. 虽然在本教程中, 我们一

般不关心精度, 但是在某些应用中, 精度是至关重要的. 我们还对每个数据字的规模假定一个范围. 例如, 当处理规模为 n 的输入时, 我们一般假定对某个大于等于 1 的常量 c, 整数由 $c\log_2 n$ 位来表示. 我们要求 $c \geq 1$, 这样每个字都可以保存 n 的值, 从而使我们能索引单个输入元素. 我们限制 c 为常量, 这样字长就不会任意增长.(如果字长可以任意增长, 我们就能在一个字中存储巨量的数据, 并且其上的操作都在常量时间内进行, 这种情况显然不现实.)

真实的计算机包含一些上面未列出的指令, 这些指令代表了 RAM 模型中的一个灰色区域. 例如, 指数运算是一条常量时间的指令吗? 一般情况下不是; 当 x 和 y 都是实数时, 计算立需要若干条指令。然而, 在受限情况下, 指数运算又是一个常量时间的操作. 许多计算机都有 “左移” 指令, 它在常量时间内将一个整数的各位向左移 k 位. 在大多数计算机中, 将一个整数的各位向左移一位等价于将该整数乘以 2, 结果将一个整数的各位向左移 k 位等价于将该整数乘以 2^k. 所以, 只要 k 不大于一个计算机字中的位数, 这样的计算机就可以由一条常量时间的指令来计算 2^k 即将整数 1 向左移 k 位. 我们尽量避免 RAM 模型中这样的灰色区域, 但是, 当 k 是一个足够小的正整数时, 我们将把 2^k 的计算看成一个常量时间的操作.

在 RAM 模型中, 我们并不试图对当代计算机中常见的内存层次进行建模. 也就是说, 我们没有对高速缓存和虚拟内存进行建模. 几种计算模型试图解释内存层次的影响, 对真实计算机上运行的真实程序, 这种影响有时是重大的. 本教程中的一些问题考查了内存层次的影响, 但是本教程的大部分分析将不考虑这些影响. 与 RAM 模型相比, 包含内存层次的模型要复杂得多, 所以可能难于使用. 此外, RAM 模型分析通常能够很好地预测实际计算机上的性能.

采用 RAM 模型即使分析一个简单的算法也可能是一个挑战. 需要的数学工具可能包括组合数学, 概率论, 代数技巧, 以及识别一个公式中最有意义的项的能力. 因为对每个可能的输入, 算法的行为可能不同, 所以我们需要一种方法来以简单的, 易于理解的公式的形式总结那样的行为.

即使我们通常只选择一种机器模型来分析某个给定的算法, 在决定如何表达我们的分析时仍然面临许多选择. 我们想要一种表示方法, 它的书写和处理都比较简单, 并能够表明算法资源需求的重要特征, 同时能够抑制乏味的细节.

插入排序算法的分析

过程3.1需要的时间依赖于输入: 排序 1000 个数比排序 3 个数需要更长的时间. 此外, 依据它们已被排序的程度, 3.1可能需要不同数量的时间来排序两个具有相同规模的输入序列. 一般来说, 算法需要的时间与输入的规模同步增长, 所以通常把一个程序的运行时间描述成其输入规模的函数. 为此, 我们必须更仔细地定义术语 “运行时间” 和 “输入规模”.

输入规模的最佳概念依赖于研究的问题. 对许多问题, 如排序, 最自然的量度是输入中的项数, 例如, 待排序数组的规模 n. 对其他许多问题, 如两个整数相乘, 输入规模的最佳量度是用通常的二进制记号表示输入所需的总位数. 有时, 用两个数而不是一个数来描述输入规模可能更合适. 例如, 若某个算法的输入是一个图, 则输入规模可以用该图中的顶点数和边数来描述. 对于研究的每个问题, 我们将指出所使用的输入规模量度.

一个算法在特定输入上的运行时间是指执行的基本操作数或步数. 定义“步”的概念以便尽量独立于机器是方便的. 目前, 让我们采纳以下观点, 执行每行伪代码需要常量时间. 虽然一行与另一行可能需要不同数量的时间, 但是我们假定第 i 行的每次执行需要时间 c_i, 其中 c_i 是一个常量. 这个观点与 RAM 模型是一致的, 并且也反映了伪代码在大多数真实计算机上如何实现气在下面的讨论中, 我们由繁到简地改进插入排序程序的运行时间的表达式, 最初的公式使用所有语句代价 c_i, 而最终的记号则更加简明, 更容易处理, 简单得多. 这种较简单的记号比较易于用来判定一个算法是否比另一个更有效.

我们首先给出过程 3.1 中, 每条语句的执行时间和执行次数. 对 $i = 2, 3, \ldots, n$, 假设 t_i 表示对那个值 i 第 5 行执行 **while** 循环测试的次数. 当一个 **for** 或 **while** 循环按通常的方式 (即由于循环头中的测试) 退出时, 执行测试的次数比执行循环体的次数多 1 . 我们假定注释是不可执行的语句, 所以它们不需要时间.

```
1  public static void InsertionSort(int[] A) {                代价      次数
2      for (int i = 1; i < A.length; i++) {                   c_1       n
3          int key = A[i];                                    c_2       n − 1
4          // 将 A[i] 插入到已经排好序的子数组 A[0:i-1]       0         n − 1
5          int j = i - 1;                                     c_4       n − 1
6          while (j > -1 && A[j] > key) {                     c_5       ∑_{i=2}^{n} t_i
7              A[j + 1] = A[j];                               c_6       ∑_{i=2}^{n} t_i − 1
8              j = j - 1;                                     c_7       ∑_{i=2}^{n} t_i − 1
9          }
10         A[j + 1] = key;                                    c_8       n − 1
11     }
12 }
```

该算法的运行时间是执行每条语句的运行时间之和. 需要执行 c_i 步且执行 n 次的一条语句将贡献 $c_i n$ 给总运行时间. 为计算在具有 n 个值的输入上 3.1 的运行时间 $T[n]$, 我们将代价与次数列对应元素之积求和, 得:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1)$$

即使对给定规模的输入, 一个算法的运行时间也可能依赖于给定的是该规模下的哪个输入. 例如, 在 3.1 中, 若输入数组已排好序, 则出现最佳运行情况. 在这种情况下, 每次执行第 5 行时, **key** 的值–最开始在 $A[i]$ 的值–已经大于等于 $A[1 : i-1]$ 中的所有值, 直到第 5-7 行的 **while** 循环退出. 所以对于 $i = 2, 3, \ldots, n$, 有 $t_i = 1$, 那么最好情况的运行时间如下:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned} \tag{3.1}$$

我们可以把该运行时间表示为 $an + b$, 其中常量 a 和 b 依赖于语句代价 c_k(其中 $a = c_1 + c_2 + c_4 + c_5 + c_8$以及$b = c_2 + c_4 + c_5 + c_8$). 因此, 它是 n 的**线性函数**.

若输入数组已反向排序, 即按递减序排好序, 则导致最坏运行情况. 我们必须将每个元素 $A[i]$ 与整个已排序子数组 $A[1 : i—1]$ 中的每个元素进行比较, 所以对 $i = 2, 3, \ldots, n$, 有 $t_i = i$.(这个过程会发现每次执行第 5 行时, 有 $A[j] > key$, 以及只有当 j 到达 0 时, **while** 才会退出.) 注意到

$$\begin{aligned} \sum_{i=2}^{n} i &= (\sum_{i=1}^{n} i) - 1 \\ &= \frac{n(n+1)}{2} - 1 \end{aligned}$$

以及

$$\begin{aligned} \sum_{i=2}^{n} (i-1) &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n-1)}{2} \end{aligned}$$

我们发现在最坏情况下, 3.1 的运行时间为

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
\tag{3.2}
$$

我们可以把该最坏情况运行时间表示为 $an^2 + bn + c$, 其中常量 a, b 和 c 又依赖于语句代价 c_k. 因此, 它是 n 的**二次函数**.

最坏情况与平均情况分析

在分析插入排序时, 我们既研究了最佳情况, 其中输入数组已排好序, 又研究了最坏情况, 其中输入数组已反向排好序. 然而, 在本教程的余下部分中, 我们往往集中于只求**最坏情况运行时间**, 即对规模为 n 的任何输入, 算法的最长运行时间. 下面给出这样做的三点理由:

- 一个算法的最坏情况运行时间给出了任何输入的运行时间的一个上界. 知道了这个界, 就能确保该算法绝不需要更长的时间. 我们不必对运行时间做某种复杂的猜测并可以期望它不会变得更坏.
- 对某些算法, 最坏情况经常出现. 例如, 当在数据库中检索一条特定信息时, 若该信息不在数据库中出现, 则检索算法的最坏情况会经常出现. 在某些应用中, 对缺失信息的检索可能是频繁的.
- "平均情况" 往往与最坏情况大致一样差. 假定随机选择 n 个数并应用插入排序. 需要多长时间来确定在子数组 $A[1 : i-1]$ 的什么位置插入元素 $A[i]$? 平均来说, $A[1 : j-1]$ 中的一半元素小于 $A[i]$, 一半元素大于 $A[i]$. 所以, 平均来说, 我们检查子数组 $A[1 : i-1]$ 的一半, 那么 t_i 大约为 $i/2$. 导致的平均情况运行时间结果像最坏情况运行时间一样, 也是输入规模的一个二次函数.

增长量级

我们使用某些简化的抽象来使过程 3.1 的分析更加容易. 首先, 通过使用常量 c_k 表示这些代价来忽略每条语句的实际代价. 其次, 注意到这些常量也提供了比我们真正需要的要多的细节: 把最坏情况运行时间表示为 $an^2 + bn + c$, 其中常量 a, b 和 c 依赖于语句代价 c_k. 这样, 我们不但忽略实际的语句代价, 而且也忽略抽象的代价 c_k.

现在我们做出一种更简化的抽象: 即我们真正感兴趣的运行时间的增长率或增长量级. 所以我们只考虑公式中最重要的项 (例如, an^2), 因为当 n 的值很大时, 低阶项相对来说不太重要. 我们也忽略最重要的项的常系数, 因为对大的输入, 在确定计算效率时常量因子不如增长率重要. 对于插入排序, 当我们忽略低阶项和最重要的项的常系数时, 只剩下最重要的项中的因子 n^2. 举个例子, 假设一个算法的输入规模是 n, 而在某一台特定的机器上面需要执行的时间是 $n^2/100 + 100n + 17$ 微秒. 尽管 n^2 的系数 $1/100$ 相比 n 的系数 100 小了 4 个数量级, 但是当 n 超过 $10,000$ 时, $n^2/100$ 将会越来越拉大和 $100n$ 的差距. 尽管 $10,000$ 看起来是个很大的输入规模, 实际上相对于实际情况, 可以说这个数量是不值一提的. 很多显示的问题, 有着大得多的输入数据规模.

为了强调运行时间的增长量级, 我们使用一个特殊的希腊字母 Θ(theta). 例如插入排序的最坏情况运行时间, 我们写作 $\Theta(n^2)$. 插入排序的最好情况运行时间我们写作 $\Theta(n)$. 现在, 我们可以把 $\Theta(f(n))$ 表示法理解为: 当 n 很大时和 $f(n)$ 成比例增长. 所以 $\Theta(n^2)$ 的意思是: 当 n 很大时, 和 n^2 成比例增长. $\Theta(n)$ 的意思是: 当 n 很大时, 和 n 成比例增长. 后面我们会形式化的讨论 Θ 表示法.

如果一个算法的最坏情况运行时间具有比另一个算法更低的增长量级, 那么我们通常认为前者比后者更有效. 由于常量因子和低阶项, 对于小的输入, 运行时间具有较高增长量级的一个算法与运行时间具有较低增长量级的另一个算法相比, 其可能需要较少的时间. 但对于足够大的输入规模, 如果一个算法的最坏情况运行时间是 $\Theta(n^2)$, 那么会比最坏情况运行时间是 $\Theta(n^3)$ 的算法所花的时间要少. 无论 Θ 表示法中隐藏的常数是多少, 总会有一个数 n_0, 对于输入规模 $n \geq n_0$, 在最坏情况下, $\Theta(n^2)$ 的算法会打败 $\Theta(n^3)$ 的算法.

3.3　如何刻画运行时间

第 2 章中定义的算法运行时间的增长量级简单地刻画了算法效率, 并且还允许我们比较可选算法的相对性能. 一旦输入规模 n 变得足够大, 最坏情况运行时间为 $\Theta(n \lg n)$ 的归并排序将战胜最坏情况运行时间为 $\Theta(n^2)$ 的插入排序. 正如我们在第 2 章中对插入排序所做的, 虽然有时我们能够确定一个算法的精确运行时间, 但是通常并不值得花力气来计算它以获得多余的精度. 对于足够大的输入, 精确运行时间中的倍增常量和低阶项被输入规模本身的影响所支配.

当输入规模足够大，使得只有运行时间的增长量级有关时, 我们要研究算法的**渐近**效率. 也就是说, 我们关心当输入规模无限增加时, 在极限中, 算法的运行时间如何随着输入规模的变大而增加. 通常, 渐近地更有效的某个算法对除很小的输入外的所有情况将是最好的选择.

本章给出几种标准方法来简化算法的渐近分析. 下一节首先定义几类 “渐近记号”, 其中, 我们已经见过的一个例子是 Θ 表示法. 然后, 我们给出贯穿本教程使用的几种表示法约定. 最后, 我们回顾一下在算法分析中常见的若干函数的行为.

3.3.1　大 O 表示法, 大 Ω 表示法和大 Θ 表示法

当我们分析插入排序的最坏情况运行时间时, 我们最开始得到的结果是如下的复杂表达式:

$$\begin{aligned}(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2})n^2+(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8)n\\ -(c_2+c_4+c_5+c_8).\end{aligned}$$

我们丢弃了低阶项 $(c_1+c_2+c_4+c_5/2-c_6/2-c_7/2+c_8)n$ 和 $c_2+c_4+c_5+c_8$, 然后还忽略了 n^2 的系数 $c_5/2+c_6/2+c_7/2$. 这样只剩下了因子 n^2, 所以我们将运行时间用 Θ 表示法写成了 $\Theta(n^2)$.

Θ 表示法并不是唯一的 “渐进表示法”. 这一节, 我们将看看其它的渐进表示法. 我们先来从直觉上研究一下这些表示法, 下一节再给出渐进表示法的形式化定义.

大 O 表示法

大 O 表示法刻画了函数的渐进行为的**上界**. 换句话说, 就是一个函数基于函数中的高阶项, 它的增长速度不会快过某个特定的增长率. 举个例子, 函数 $7n^3+100n^2-20n+6$ 的高阶项是 $7n^3$, 所以我们说这个函数的增长率是 n^3. 由于函数的增长率不会快过 n^3, 所以我们可以写作 $O(n^3)$. 我们也可以将 $7n^3+100n^2-20n+6$ 写作 $O(n^4)$. 为什么呢? 因为函数的增长率肯定比 n^4 慢啊. 我们还可以写成 $O(n^5), O(n^6)$ 等等. 更一般地, 可以写成 $O(n^c)$, 只要 $c\geq 3$.

大 Ω 表示法

大 Ω 表示法刻画的是函数的渐进行为的**下界**. 换句话说, 就是一个函数基于函数中的高阶项, 它的增长率不会慢于某个特定的增长率. 因为函数 $7n^3+100n^2-20n+6$ 的高阶项至少增长的和 n^3 一样快, 所以这个函数是 $\Omega(n^3)$. 当然也可以写作 $\Omega(n^2)$ 和 $\Omega(n)$. 更一般地, 可以写作 $\Omega(n^c)$, 只要 $c\leq 3$.

大 Θ 表示法

大 Θ 表示法刻画的是一个函数的渐进行为的**紧界**. 也就是说一个函数基于函数中的高阶项, 精确地按照某个增长率增长.

如果我们能确定一个函数即是 $O(f(n))$ 又是 $\Omega(f(n))$, 那么函数一定是 $\Theta(f(n))$. 例如, $7n^3+100n^2-20n+6$ 既是 $O(n^3)$ 又是 $\Omega(n^3)$, 所以函数一

定是 $\Theta(n^3)$.

例子：插入排序

让我们重新来看一下插入排序的最坏情况运行时间为什么可以写作渐进表示法 $\Theta(n^2)$, 当然这次我们不做繁琐的计算. 下面是插入排序的代码:

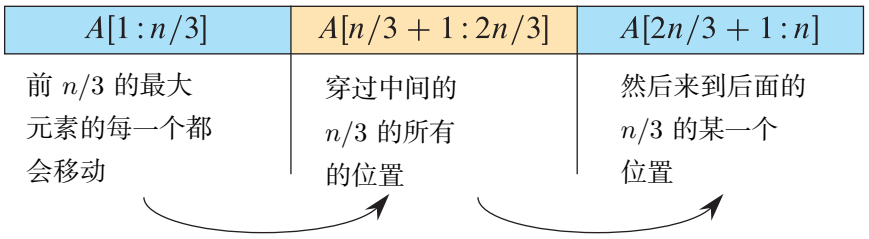
```
1  public static void InsertionSort(int[] A) {                代价    次数
2      for (int i = 1; i < A.length; i++) {                   c1      n
3          int key = A[i];                                    c2      n − 1
4          // 将 A[i] 插入到已经排好序的子数组 A[0:i-1]       0       n − 1
5          int j = i - 1;                                     c4      n − 1
6          while (j > -1 && A[j] > key) {                     c5      ∑_{i=2}^{n} t_i
7              A[j + 1] = A[j];                               c6      ∑_{i=2}^{n} t_i − 1
8              j = j - 1;                                     c7      ∑_{i=2}^{n} t_i − 1
9          }
10         A[j + 1] = key;                                    c8      n − 1
11     }
12 }
```

从上面的程序中可以观察出什么呢? 代码中有嵌套循环. 无论数组的有序程度如何, 外层 **for** 循环执行 $n-1$ 次. 内层的 **while** 循环的循环次数取决于数组元素的有序程度. 循环变量 j 从 $i-1$ 开始, 然后每次迭代时减去 1, 知道 j 到达 0 或者 $A[j] \leq key$. 对于一个给定的值 i, **while** 循环的迭代次数从 0 次到 $i-1$ 次都有可能. 每次迭代执行时, **while** 循环体 (6-7 行) 需要常数时间.

这些观察可以归纳出对于任意情形的 **Insertion-Sort** 程序, 运行时间是 $O(n^2)$, 这就给了我们针对所有情形的输入都正确的估计. 运行时间由内层循环主要决定. 由于 $n-1$ 次外层循环的每一次, 内层循环的迭代次数都是最多执行 $i-1$ 次, 而 i 的最大值是 n, 所以最内层循环的循环体 (6-7 行) 的执行次数最多是 $(n-1)(n-1)$ 次. 显然小于 n^2 次. 而循环体 (6-7 行) 需要常数时间, 执行次数是 n^2, 所以插入排序的总时间是 $O(n^2)$.

如果有一点创造力的话, 我们会发现 **Insertion-Sort** 程序在最坏情况的运行时间是 $\Omega(n^2)$. 我们说一个算法的最坏情况的运行时间是 $\Omega(n^2)$ 的意思是, 对于超过某个阈值的输入 n, 至少**存在**一个输入规模 n, 算法至少需要 cn^2 的运行时间, c 是某个正常数. 这并不是说算法对于所有的输入规模都需要至少 cn^2 的时间.

让我们现在来看一下插入排序 **Insertion-Sort** 的最坏情况运行时间为什么是 $\Omega(n^2)$. 对于数组中的一个值, 伪代码的第 6 行最终会将这个值移动到最终的正确的位置. 事实上, 如果对于一个元素, 从开始处于的位置需要向右移动 k 个位置才能到达正确的位置, 那么伪代码第 6 行必须执行 k 次. 如图3.3.1所示,



让我们假设 n 是 3 的整数倍, 所以我们可以将 A 分成三组. 再假设在最开始, 数组中 $n/3$ 个最大的元素为数组中前 $n/3$ 个位置. 一旦数组完成排序, 最大的这 $n/3$ 个元素将位于数组中的 $A[2n/3+1:n]$. 也就是说最大的这 $n/3$ 个数中的每一个都会穿越中间的 $A[n/3+1:2n/3]$ 这 $n/3$ 个位置. $n/3$ 个最大的元素中的每一个都需要穿越中间的 $n/3$ 个位置, 每次走一步, 那么第 6 行程序最少需要执行 $n/3$ 次. 对于这个例子, 至少有 $n/3$ 的数组元素移动了至少 $n/3$ 步, 那么插入排序在这个例子上的最坏情况下的运行时间和 $(n/3)(n/3) = (n^2/9)$ 成比例, 所以可以写成 $\Omega(n^2)$.

因为我们已经看到插入排序在最坏情况下对于**所有**输入规模的运行时间都是 $O(n^2)$, 同时至少**存在一个**输入使得插入排序在最坏情况下的运行时间是 $\Omega(n^2)$. 上界和下界的常数因子可以直接忽略掉. 经过分析发现我们并没有证明对于所有的输入个规模插入排序的运行时间是 $\Theta(n^2)$, 事实上, 我们之前已经分析过了, 插入排序在最好情况下运行时间是 $\Theta(n)$.

3.3.2　渐进表示法: 形式化定义

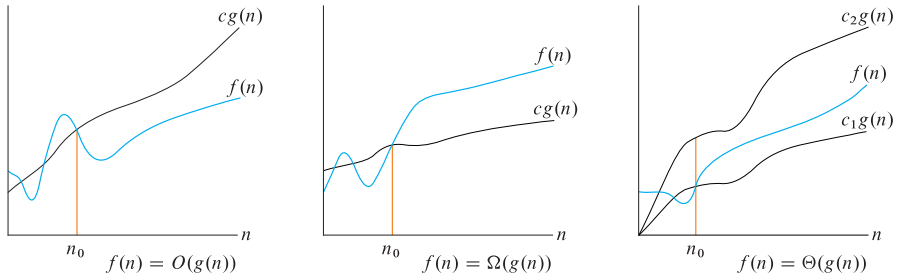
非正式地研究了一番渐进表示法, 我们现在来给出它们的形式化定义. 渐进表示法用来描述一个算法的渐进运行时间. 渐进表示法函数的定义域是自然数集合 $\mathbb{N} = \{0, 1, 2, \cdots\}$. 这些表示法适合用来表示运行时间函数 $T(n)$.

大 O 表示法

大 O 表示法表示**渐进上界**, 给定一个函数 $g(n)$, 我们定义 $O(g(n))$ 如下

定义 3.1. $O(g(n)) = \{f(n)$: *存在一个正的常数 c 和 n_0 使得对于所有 $n \geq n_0$, 有 $0 \leq f(n) \leq cg(n)$*$\}$.

上面的定义是说对于足够大的规模 n, 如果存在常数 c 使得 $f(n) \leq cg(n)$, 那么 $f(n)$ 就属于 $O(g(n))$ 这个集合. 图3.3.2(a) 展示了大 O 表示法背后的直



觉. 从图里可以看到对于所有在 n_0 右边的值 n, $f(n)$ 都在 $cg(n)$ 底下.

$f(n)$ 属于 $O(g(n))$ 集合, 虽然可以写成 $f(n) \in O(g(n))$, 但我们一般写作 $f(n) = O(g(n))$.

我们举个例子来说明使用大 O 表示法时, 丢弃低阶项和忽略系数的合理性. 我们会展示为什么 $4n^2+100n+500 = O(n)$, 即使低阶项的系数远大于高阶项的系数. 我们需要找到 c 和 n_0, 来使得对于所有的 $n \geq n_0$ 有 $4n^2 + 100n + 500 \leq cn^2$. 将不等式两边同时除以 n^2 得到 $4 + 100/n + 500/n^2 \leq c$. 我们可以找到很多符合条件的 c 和 n_0. 例如: $n_0 = 1, c = 604$, $n_0 = 10, c = 19$, 以及 $n_0 = 100, c = 5.05$ 都符合条件.

我们还可以使用大 O 表示法的定义来说明 $n^3 - 100n^2$ 不属于 $O(n^2)$ 集合, 即使 n^2 的系数是一个很大的负数. 使用反证法, 如果我们有 $n^3 - 100n^2 = O(n^2)$, 那么一定存在 c 和 n_0 使得对于所有的 $n \geq n_0$ 有 $n^3 - 100n^2 \leq cn^2$. 我们再次将不等式两边同时除以 n^2, 得到 $n - 100 \leq c$. 无论怎么选择 c, 对于 $n > c + 100$ 不等式都不成立.

大 Ω 表示法

大 O 表示法提供了一个函数的渐进上界. 那么大 Ω 表示法就给出了一个函数的**渐进下界**. 对于一个给定的函数 $g(n)$, 我们给出 $\Omega(g(n))$ 的形式化定义

定义 3.2. $\Omega(g(n)) = \{f(n)$: *存在正的常数 c 和 n_0 使得对于所有的 $n \geq n_0$ 有* $0 \leq cg(n) \leq f(n)\}$.

图3.3.2(b) 展示了大 Ω 表示法背后的直觉. 对于所有在 n_0 右边的 n, $f(n)$ 在 $cg(n)$ 的上面.

我们已经证明了 $4n^2+100n+500 = O(n)$, 现在让我们来证明 $4n^2+100n+500 = \Omega(n)$. 我们需要找到 c 和 n_0 使得对于所有 $n \geq n_0$ 有 $4n^2+100n+500 \geq cn^2$. 显然 $c = 4, n_0 = 1$ 就符合条件.

我们还可以证明 $n^2/100 - 100n - 500 = \Omega(n^2)$. 实际上 $c = 0.0089, n_0 = 100,000$ 就符合条件.

大 Θ 表示法

我们使用 Θ 来表示**渐进紧界**. 对于 $g(n)$ 函数, 我们将 $\Theta(g(n))$ 定义为如下集合

定义 3.3. $\Theta(g(n)) = \{f(n)$: 存在正的常数 c_1, c_2 和 n_0 使得对于所有的 $n \geq n_0$ 有 $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$.

图3.3.2(c) 展示了大 Θ 表示法背后的直觉. 对于所有在 n_0 右边的 n, $f(n)$ 在 $c_1 g(n)$ 的上面, 且在 $c_2 g(n)$ 的下面. 实际上说明 $f(n)$ 和 $g(n)$ 的增长率是相同的, 仅仅可能差了常数因子.

现在我们有以下定理

定理 3.1. 对于两个函数 $f(n)$ 和 $g(n)$, $f(n) = \Theta(g(n))$ 当且仅当 $f(n) = O(g(n))$ 且 $f(n) = \Omega(g(n))$.

总结

我们需要掌握一些法则来加快我们判断一个程序的运行时间的速度.

法则 1

如果 $T_1(n) = O(f(n)), T_2(n) = O(g(n))$, 那么

1. $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$
2. $T_1(n) \cdot T_2(n) = O(f(N) \cdot g(n))$

法则 2

如果 $T(n)$ 是一个 k 次多项式, 则 $T(n) = \Theta(n^k)$.

法则 3

对任意常数 $k, \lg^k n = O(n)$. 也就是说对数增长的非常缓慢.

这些信息足以按照增长率对大部分常见的函数进行分类.

有几点需要注意. 首先, 将常数或者低阶项放进大 O 是非常坏的习惯. 不要写成 $T(n) = O(2n^2)$ 或者 $T(n) = O(n^2 + n)$. 在这两种情况下, 正确的形式是 $T(n) = O(n^2)$. 也就是说, 在需要大 O 表示法的任何分析中, 各种简化都是可能发生的. 低阶项一般可以被忽略, 而常数也可以丢弃. 此时, 要求的精度是很粗糙的.

第二, 我们总能够通过计算极限 $\lim_{n\to\infty} f(n)/g(n)$ 来确定两个函数 $f(n)$ 和 $g(n)$ 的相对增长率. 例如如果极限是 $c \neq 0$: 这意味着 $f(n) = \Theta(g(n))$.

使用这种方法几乎总能算出相对增长率, 不过有些复杂. 通常, 两个函数 $f(n)$ 和 $g(n)$ 之间的关系用简单的代数方法就能得到. 例如, 如果 $f(n) = n \lg n$ 和 $g(n) = n^{1.5}$, 那么为了确定 $f(n)$ 和 $g(n)$ 哪个增长得更快, 实际上就是确定 $\lg n$ 和 $n^{0.5}$ 哪个增长得更快. 这与确定 $\lg^2 n$ 和 n 哪个增长更快是一样的, 而后者是个简单的问题. 因为我们已经知道, n 的增长要快于 lg 的任意的幂. 因此, $g(n)$ 的增长要快于 $f(n)$ 的增长.

另外, 在风格上还应该注意: 不要写成 $f(n) \leq O(g(n))$, 因为定义已经隐含有不等式了. 写成 $f(n) \geq O(g(n))$ 是错误的, 它没有意义.

根据上面的结论, 可以知道一些常见的计算结果:

- $O(1) + O(\lg n) = O(\lg n)$
- $O(\lg n) + O(n) = O(n)$
- $O(n) + O(n \lg n) = O(n \lg n)$
- $O(n \lg n) + O(n^2) = O(n^2)$
- $nO(n) = O(n^2)$
- ……

3.4　斐波那契数列和递归

程序 3.2: 斐波那契数列的递归实现

```
1  public class RecursiveFibAlgorithm {
2      public long RecursiveFib(int n) {
3          if (n == 0 || n == 1)
4              return n;
5          else
6              return RecursiveFib(n - 1) + RecursiveFib(n - 2);
7      }
8
9      public static void main(String[] args) {
10         System.out.println(RecursiveFib(5));
11     }
12 }
```

程序 3.3: 斐波那契数列的缓存实现

```
1  public class CacheFibAlgorithm {
2      public static int MAXN = 92;
3      public static int UNKNOWN = -1;
4      public static long F[MAXN + 1];
5
6      public static long CacheFib(int n) {
7          if (F[n] == UNKNOWN)
8              F[n] = CacheFib(n - 1) + CacheFib(n - 2);
9          return F[n];
10     }
11
12     public static long CacheFibDriver(int n) {
13         int i;
14
15         F[0] = 0;
16         F[1] = 1;
17
18         for (i = 2; i <= n; i++)
19             F[i] = UNKNOWN;
20
21         return CacheFib(n);
22     }
23
24     public static void main(String[] args) {
25         System.out.println(CacheFibDriver(5));
26     }
27 }
```

程序 3.4: 斐波那契数列的动态规划实现

```
1  public class DpFibAlgorithm {
2      public static long DpFib(int n) {
3          int i;
4          long F[MAXN + 1];
5
```

```
        F[0] = 0;
        F[1] = 1;

        for (i = 2; i <= n; i++)
            F[i] = F[i-1] + F[i-2];

        return F[n];
    }

    public static void main(String[] args) {
        System.out.println(DpFib(5));
    }
}
```

程序 3.5: 斐波那契数列的终极实现

```
public class UltimateFibAlgorithm {
    public static long UltimateFib(int n) {
        int i;
        long back2 = 0; back1 = 1;
        long next;

        if (n == 0) return 0;

        for (i = 2; i < n, i++) {
            next = back1 + back2;
            back2 = back1;
            back1 = next;
        }

        return back1 + back2;
    }

    public static void main(String[] args) {
        System.out.println(UltimateFib(5));
    }
```

```
21 }
22
```

3.5　二分查找

程序 3.6: 二分查找的递归实现

```
1  public class RecursiveBinarySearchAlgorithm {
2      public static int RecursiveBinarySearch(int[] A, int t, int
       ↪  low, int high) {
3          if (low > high) return -1;
4          int mid = (low + high) / 2;
5          if (t == A[mid])
6              return mid;
7          else if (t > A[mid])
8              return RecursiveBinarySearch(A, t, mid + 1, high);
9          else
10             return RecursiveBinarySearch(A, t, low, mid - 1);
11     }
12
13     public static void main(String[] args) {
14         int[] A = {1,2,3,4,5,6,7,8,9};
15         System.out.println(RecursiveBinarySearch(A, 9, 0, 8));
16     }
17 }
```

$$T(n) = \begin{cases} \Theta(1) & \text{如果 } n = 1, \\ T((n-1)/2) + \Theta(1) & \text{如果 } n > 1 \end{cases}$$

推导过程如下, 在最坏情况下, 目标数据不在数组中, 那么下面的不等式成立, 因为数组越大, 最坏情况下的运行时间肯定越长.

$$T(n) = T((n-1)/2) + \Theta(1) \leq T(n/2) + \Theta(1)$$

我们假设 $n = 2^m$

$$
\begin{aligned}
T(n) &\leq T(n/2) + \Theta(1) \\
&= T(2^{m-1}) + 1 \cdot \Theta(1) \\
&= T(2^{m-2}) + 2 \cdot \Theta(1) \\
&= T(2^{m-3}) + 3 \cdot \Theta(1) \\
&= \cdots \\
&= T(2^{m-m}) + m \cdot \Theta(1) \\
&= (1+m)\Theta(1) \\
&= \Theta(m+1) \\
&= \Theta(m) \\
&= \Theta(\lg n)
\end{aligned}
$$

所以二分查找在最坏情况下, 时间复杂度是 $\Theta(\lg n)$.

在最好情况下, 要查找的数位于数组中心, 那么一次就可以命中. 所以最好情况下的运行时间是 $\Theta(1)$.

程序 3.7: 二分查找的迭代实现

```
1  public class IterativeBinarySearchAlgorithm {
2      public static int IterativeBinarySearch(int[] A, int t) {
3          int low = A[0];
4          int high = A[A.length - 1];
5          while (low <= high) {
6              int mid = (low + high) / 2;
7              if (t == A[mid])
8                  return mid;
9              else if (t > A[mid])
10                 low = mid + 1;
11             else
12                 high = mid - 1;
13         }
14         return -1;
15     }
16
17     public static void main(String[] args) {
```

```
18          int[] A = {1,2,3,4,5,6,7,8,9};
19          System.out.println(IterativeBinarySearch(A, 9));
20      }
21  }
```

3.6　设计算法

我们可以选择使用的算法设计技术有很多. 插入排序使用了**增量**方法: 在排序子数组 $A[1:i-1]$ 后, 将单个元素 $A[i]$ 插入子数组 $A[1:i]$ 的适当位置.

本节我们考查另一种称为 “分治法” 的设计方法. 第 4 章将更深入地探究该方法. 我们将用分治法来设计一个排序算法, 该算法的最坏情况运行时间比插入排序要少得多. 分治算法的优点之一是, 通过使用第 4 章介绍的技术往往很容易确定其运行时间.

3.6.1　分治法

许多有用的算法在结构上是递归的: 为了解决一个给定的问题, 算法一次或多次递归地调用其自身以解决紧密相关的若干子问题. 这些算法典型地遵循分治法的思想: 将原问题分解为几个规模较小但类似于原问题的子问题, 递归地求解这些子问题, 然后再合并这些子问题的解来建立原问题的解.

分治模式在每层递归时都有三个步骤:

分解原问题为若干子问题, 这些子问题是原问题的规模较小的实例.

解决这些子问题, 递归地求解各子问题. 然而, 若子问题的规模足够小, 则直接求解.

合并这些子问题的解成原问题的解.

归并排序算法完全遵循分治模式. 直观上其操作如下:

分解: 分解待排序的 n 个元素的序列成各具 $n/2$ 个元素的两个子序列.

解决: 使用归并排序递归地排序两个子序列.

合并: 合并两个已排序的子序列以产生已排序的答案.

当待排序的序列长度为 1 时, 递归 “开始回升”, 在这种情况下不要做任何工作, 因为长度为 1 的每个序列都已排好序.

归并排序算法的关键操作是 “合并” 步骤中两个已排序序列的合并. 我们通过调用一个辅助过程 $Merge(A,p,q,r)$ 来完成合并, 其中 A 是一个数组, p, q 和 r 是数组下标，满足 $p \le q < r$. 该过程假设子数组 $A[p:q]$ 和 $A[q+1:r]$ 都已

排好序. 它合并这两个子数组形成单一的已排好序的子数组并代替当前的子数组 $A[p:r]$.

为了理解合并程序的工作原理, 让我们回到玩扑克牌的例子, 假设桌上有两堆牌面朝上的牌, 每堆都已排序, 最小的牌在顶上. 我们希望把这两堆牌合并成单一的排好序的输出堆, 牌面朝下地放在桌上. 我们的基本步骤包括在牌面朝上的两堆牌的顶上两张牌中选取较小的一张, 将该牌从其堆中移开 (该堆的顶上将显露一张新牌) 并牌面朝下地将该牌放置到输出堆. 重复这个步骤, 直到一个输入堆为空, 这时, 我们只是拿起剩余的输入堆并牌面朝下地将该堆放置到输出堆.

让我们思考一下合并两堆已经排好序的纸牌需要多长时间. 每个基本步骤都只花费常数时间, 因为我们做的只是比较两张最上面的纸牌. 如果我们准备合并的两堆纸牌, 每堆纸牌各有 $n/2$ 张纸牌, 那么基本步骤的次数至少是 $n/2$ 次, 至多是 n 次. 所以每个基本步骤花费常数时间, 而基本步骤的操作次数在 $n/2$ 和 n 之间. 所以我们可以说合并所需要的时间大概和 n 是成比例增长的. 也就是说, 合并需要 $\Theta(n)$ 的时间.

让我们深入一下细节, 实际上合并程序以如下过程执行: 将两个子数组 $A[p:q]$ 和 $A[q+1:r]$ 分别拷贝到临时数组 L 和 R, 然后将 L 和 R 中的元素合并回 $A[p:r]$. 程序中的第 1 行和第 2 行计算 $A[p:q]$ 的长度 n_L, 还有 $A[q+1:r]$ 的长度 n_R. 第 3 行创建两个空数组: 长度为 n_L 的 $L[0:n_L-1]$ 数组和长度为 n_R 的 $R[0:n_R-1]$ 数组. 第 4-5 行的 **for** 循环将子数组 $A[p:q]$ 拷贝到数组 L, 第 6-7 行的 **for** 循环将子数组 $A[q+1:r]$ 拷贝到数组 R.

第 8-18 行, 如图3.6.1所示, 执行了基本步骤. 第 12-18 行的 **while** 循环反复识别 L 和 R 中还没有拷贝到 $A[p:r]$ 的元素中的最小值, 然后将最小值拷贝到 $A[p:r]$ 中. 如注释所示, 索引 k 给出了 A 中待填充的位置, 索引 i 和 j 分别给出了 L 和 R 各自的剩下的最小元素的位置. 最终, L 和 R 的所有元素都会被拷贝回 $A[p:r]$, 然后循环终结. 如果循环终结是因为 R 中所有的元素都被拷贝回去了, 那么, 由于 j 等于 n_R, 而 i 仍然小于 n_L, 所以一些 L 中的元素还没有被拷贝回去, 而这些元素的值是目前 L 和 R 中最大的值. 在这种情况下, 20-23 行的代码将 L 中剩余的元素拷贝回 $A[p:r]$ 的最后面的几个位置. 因为 j 等于 n_R, 24-27 行的 **while** 循环执行 0 次. 如果 12-18 行的 **while** 循环因为 i 等于 n_L 而终结, 那么 L 中所有的元素都已经拷贝回 $A[p:r]$ 中了, 那么 24-27 行的 **while** 循环将会把 R 中剩下的元素拷贝到 $A[p:r]$ 的末尾处.

可以看到合并程序的执行时间是 $\Theta(n), n = r-p+1$, 注意到 1-3 的每一行代码和 8-10 的每一行代码都只花费常数时间, 而 4-7 行的 **for** 循环花费 $\Theta(n_L+n_R) = \Theta(n)$ 时间. 为了计算 12-18, 20-23 和 24-27 这 3 个 **while** 循环的时间, 只需要看一下代码就会发现这些循环的每一次迭代做的事情只是将 L 或

者 R 中的一个元素拷贝回 A 数组. 所以这 3 个循环的总的迭代次数是 n. 由于 3 个循环的每次迭代都花费常数时间, 所以总时间花费 $\Theta(n)$.

现在我们可以把过程合并程序作为归并排序算法中的一个子程序来用. 下面的过程 $MergeSort(A,p,r)$ 排序子数组 $A[p:r]$ 中的元素. 若 $p=r$, 则该子数组最多有一个元素, 所以已经排好序. 否则, 分解步骤简单地计算一个下标 q, 将 $A[p:r]$ 分成两个子数组 $A[p:q]$ 和 $A[q+1:r]$, 前者包含 $\lceil n/2 \rceil$ 个元素，后者包含 $\lfloor n/2 \rfloor$ 个元素. 初始调用 $MergeSort(A,1,n)$ 对整个数组 $A[1:n]$ 排序.

图3.6.1展示了归并排序在 $n=8$ 的数组上的完整执行过程, 将分解和合并的每一步都展示了出来. 归并排序递归地将数组分解成了只包含 1 个元素的子数组. 合并的步骤会将单元素子数组的对进行合并, 来生成包含 2 个元素的子数组, 然后继续合并出包含 4 个元素的子数组, 然后最终合并成 8 个元素的子数组. 如果 n 不是 2 的整数次幂, 那么分解出来的最终子数组的长度可能会相差 1(例如, 分解一个长度为 7 的子数组, 那么一个子数组长度为 4, 一个子数组长度为 3). 无论要合并的子数组的长度如何, 合并 n 个元素的时间是 $\Theta(n)$.

程序 3.8: 归并排序中的合并过程

```
1  public static void Merge(int[] A, int p, int q, int r) {
2      int nL = q - p + 1;
3      int nR = r - q;
4
5      int[] L = new int[nL];
6      int[] R = new int[nR];
7
8      for (int i = 0; i < nL; i++) {
9          L[i] = A[p + i];
10     }
11     for (int j = 0; j < nR; j++) {
12         R[j] = A[q + j + 1];
13     }
14
15     int i = 0;
16     int j = 0;
17     int k = p;
18
19     while (i < nL && j < nR) {
20         if (L[i] <= R[j]) {
```

```
            A[k] = L[i];
            i = i + 1;
        } else {
            A[k] = R[j];
            j = j + 1;
        }
        k = k + 1;
    }

    while (i < nL) {
        A[k] = L[i];
        i = i + 1;
        k = k + 1;
    }

    while (j < nR) {
        A[k] = R[j];
        j = j + 1;
        k = k + 1;
    }
}
```

程序 3.9: 归并排序的主程序

```
public static void MergeSort(int[] A, int p, int r) {
    if (p >= r) return;
    int q = (p + r) / 2;
    MergeSort(A, p, q);
    MergeSort(A, q + 1, r);
    Merge(A, p, q, r);
}
```

程序 3.10: 归并排序的测试程序

```
public class MergeSortAlgorithm {
    public static void Merge(int[] A, int p, int q, int r) {
```

```
        // 程序见3.8
    }

    public static void MergeSort(int[] A, int p, int r) {
        // 程序见3.9
    }

    public static void main(String[] args) {
        int[] A = {12,3,7,9,14,6,11,2};
        MergeSort(A, 0, 7);
        for (var i : A)
            System.out.println(i);
    }
}
```

3.6.2　分析归并排序算法

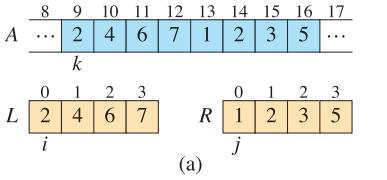
当一个算法包含对其自身的递归调用时, 我们往往可以用**递归方程**或**递归式**来描述其运行时间, 该方程根据在较小输入上的运行时间来描述在规模为 n 的问题上的总运行时间. 然后, 我们可以使用数学工具来求解该递归式并给出算法性能的界.

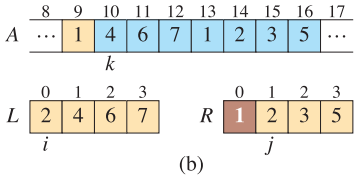
分治算法运行时间的递归式来自基本模式的三个步骤. 就像我们分析插入排序时那样, 我们假设 $T(n)$ 是规模为 n 的一个问题的最坏情况下的运行时间. 若问题规模足够小, 如对某个常量 $n_0, n < n_0$, 则直接求解需要常量时间, 我们将其写作 $\Theta(1)$. 假设把原问题分解成 a 个子问题, 每个子问题的规模是原问题的 $1/b$.(对归并排序, a 和 b 都为 2). 为了求解一个规模为 n/b 的子问题, 需要 $T(n/b)$ 的时间, 所以需要 $aT(n/b)$ 的时间来求解 a 个子问题. 如果分解问题成子问题需要时间 $D(n)$, 合并子问题的解成原问题的解需要时间 $C(n)$, 那么得到递归式:

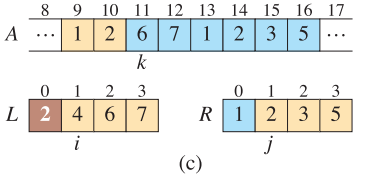
$$T(n) = \begin{cases} \Theta(1), & \text{如果} n < n_0 \\ D(n) + aT(n/b) + C(n), & \text{其他情况} \end{cases}$$

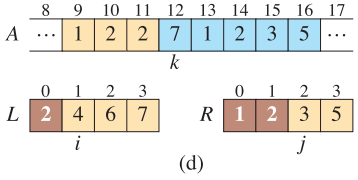
归并排序的分析

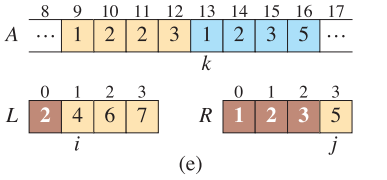
下面我们分析建立归并排序 n 个数的最坏情况运行时间 $T(n)$ 的递归式. 归并排序一个元素需要常量时间. 当有 $n > 1$ 个元素时, 我们分解运行时间如下:

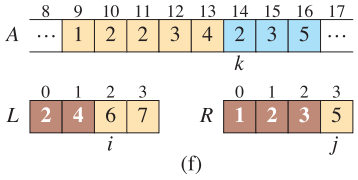

A … 2 4 6 7 1 2 3 5 …
k
L 2 4 6 7
i
R 1 2 3 5
j
(a)

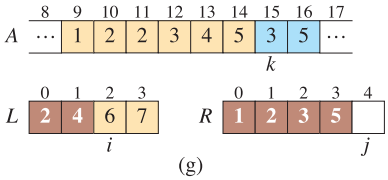

A … 1 4 6 7 1 2 3 5 …
k
L 2 4 6 7
i
R 1 2 3 5
j
(b)

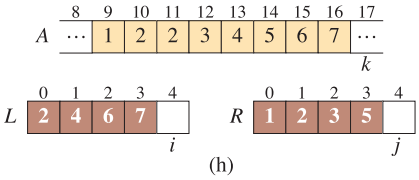

A … 1 2 6 7 1 2 3 5 …
k
L 2 4 6 7
i
R 1 2 3 5
j
(c)

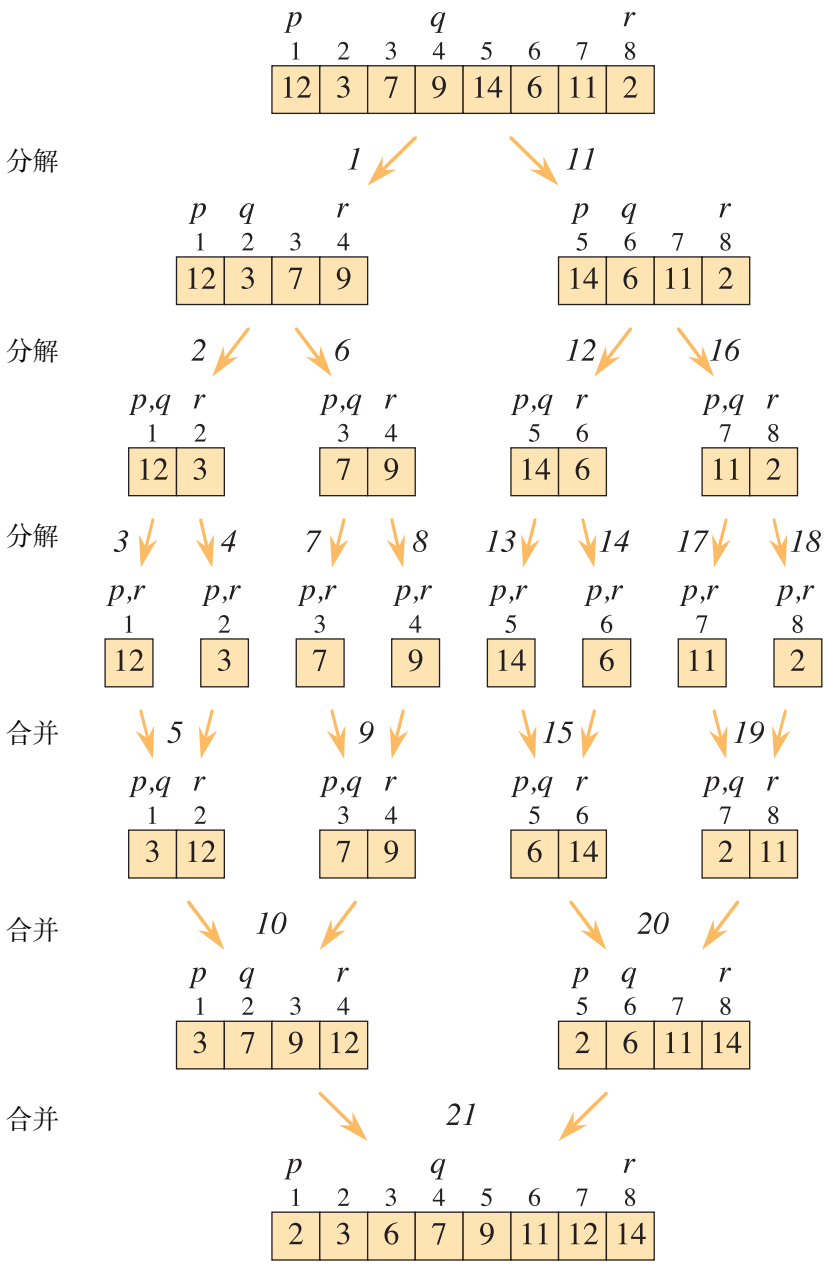

A … 1 2 2 7 1 2 3 5 …
k
L 2 4 6 7
i
R 1 2 3 5
j
(d)


A … 1 2 2 3 1 2 3 5 …
k
L 2 4 6 7
i
R 1 2 3 5
j
(e)


A … 1 2 2 3 4 2 3 5 …
k
L 2 4 6 7
i
R 1 2 3 5
j
(f)


A … 1 2 2 3 4 5 3 5 …
k
L 2 4 6 7
i
R 1 2 3 5
j
(g)


A … 1 2 2 3 4 5 6 7 …
k
L 2 4 6 7
i
R 1 2 3 5
j
(h)


p q r
1 2 3 4 5 6 7 8
12 3 7 9 14 6 11 2
分解
1
11
p q r
1 2 3 4
12 3 7 9
p q r
5 6 7 8
14 6 11 2
分解
2
6
12
16
p,q r
1 2
12 3
p,q r
3 4
7 9
p,q r
5 6
14 6
p,q r
7 8
11 2
分解
3
4
7
8
13
14
17
18
p,r
1
12
p,r
2
3
p,r
3
7
p,r
4
9
p,r
5
14
p,r
6
6
p,r
7
11
p,r
8
2
合并
5
9
15
19
p,q r
1 2
3 12
p,q r
3 4
7 9
p,q r
5 6
6 14
p,q r
7 8
2 11
合并
10
20
p q r
1 2 3 4
3 7 9 12
p q r
5 6 7 8
2 6 11 14
合并
21
p q r
1 2 3 4 5 6 7 8
2 3 6 7 9 11 12 14

分解: 分解步骤仅仅计算子数组的中间位置, 需要常量时间, 因此, $D(n) = \Theta(1)$.

解决: 我们递归地求解两个规模均为 $n/2$ 的子问题, 将贡献 $2T(n/2)$ 的运行时间.

合并: 我们已经注意到在一个具有 n 个元素的子数组上过程 **Merge** 需要 $\Theta(n)$ 的时间, 所以 $C(n) = \Theta(n)$.

当为了分析归并排序而把函数 $D(n)$ 与 $C(n)$ 相加时, 我们是在把一个 $\Theta(n)$ 函数与另一个 $\Theta(1)$ 函数相加. 相加的和是 n 的一个线性函数. 也就是说当 n 足够大时, $\Theta(n) + \Theta(1) = \Theta(n)$, 也就是说 $\Theta(1)$ 相比 $\Theta(n)$ 可以忽略掉. 把它与来自 “解决” 步骤的项 $2T(n/2)$ 相加, 将给出归并排序的最坏情况运行时间 $T(n)$ 的递归式:

$$T(n) = \begin{cases} \Theta(1), & \text{如果} n = 1 \\ 2T(n/2) + \Theta(n), & \text{如果} n > 1 \end{cases}$$

由于 $\Theta(1)$ 是常数时间, 所以我们可以假设 $\Theta(1) = c_1$, 而 $\Theta(n) = c_2 n$. 那么上面的递归式可以化简成如下:

$$T(n) = \begin{cases} c_1, & \text{如果} n = 1 \\ 2T(n/2) + c_2 n, & \text{如果} n > 1 \end{cases}$$

我们在分析运行时间时, 经常把常数 c_1 化简为 1, 将 $c_2 n$ 化简为 n, 这样更方便我们分析, 所以上面的式子进一步化简成了如下:

$$T(n) = \begin{cases} 1, & \text{如果} n = 1 \\ 2T(n/2) + n, & \text{如果} n > 1 \end{cases}$$

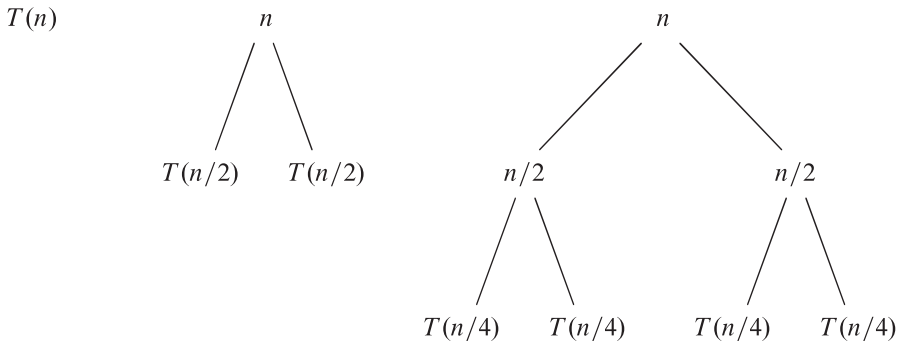
递归式的求解可以从图3.6.2中看出来.

不过我们还是进行一下完整的推导, 首先假设

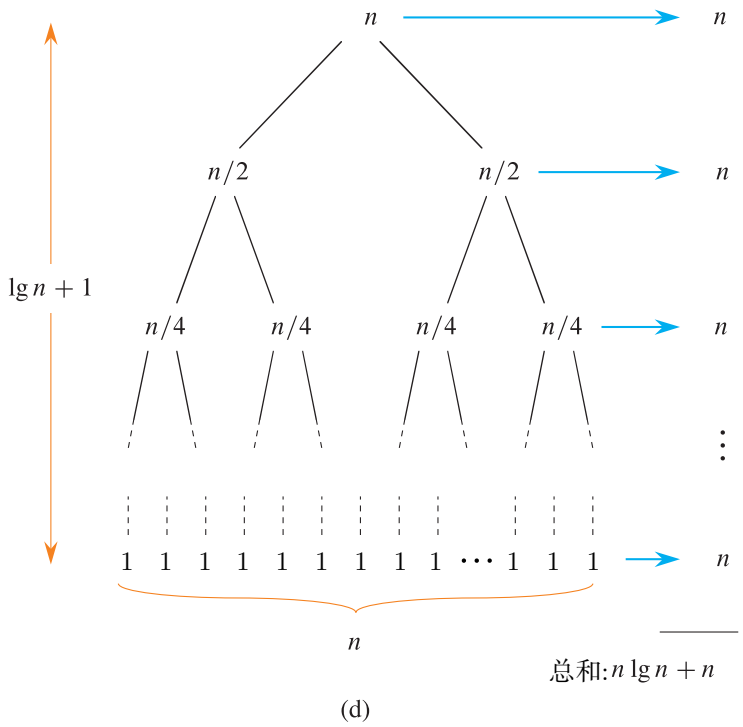
$$n = 2^m$$

那么根据替换法则, 有如下

$$\begin{aligned} T(n) &= T(2^m) \\ T(n/2) &= T(2^{m-1}) \\ T(n/4) &= T(2^{m-2}) \end{aligned}$$


T(n)
n
T(n/2)
T(n/2)
n
n/2
n/2
T(n/4)
T(n/4)
T(n/4)
T(n/4)

(a)　(b)　(c)


n
n
n/2
n/2
n
lg n + 1
n/4
n/4
n/4
n/4
n
1 1 1 1 1 1 1 1 1 ··· 1 1 1
n
n
总和:n lg n + n

(d)

所以我们可以开始推导了

$$
\begin{aligned}
T(2^m) &= 2^1 T(2^{m-1}) + 1 \cdot 2^m \\
&= 2(2T(2^{m-2}) + 2^{m-1}) + 2^m \\
&= 2^2 T(2^{m-2}) + 2 \cdot 2^m \\
&= 2^2(2T(2^{m-3}) + 2^{m-2}) + 2 \cdot 2^m \\
&= 2^3 T(2^{m-3}) + 3 \cdot 2^m \\
&= \dots \\
&= 2^m T(2^{m-m}) + m \cdot 2^m \\
&= 2^m T(1) + m \cdot 2^m \\
&= n + \lg nn \\
&= \lg nn \\
&= \Theta(\lg nn)
\end{aligned}
$$

第四章　力扣算法题选讲

第二部分

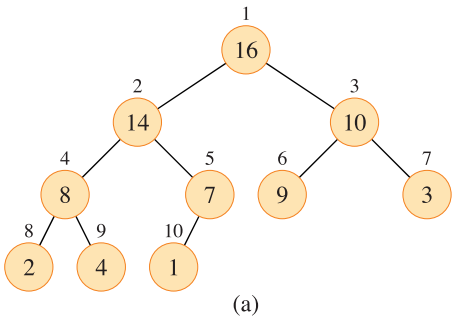
排序和顺序统计量

第五章　堆排序

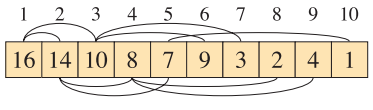
5.1　堆 (Heap)

程序 5.1: 使用数组实现堆如何计算某个元素的左孩子, 右孩子和父亲结点

```
1  public int parent(int i) {
2      return (i - 1) / 2;
3  }
4
5  public int left(int i) {
6      return 2 * i + 1;
7  }
8
9  public int right(int i) {
10     return 2 * i + 2;
11 }
```



(a)



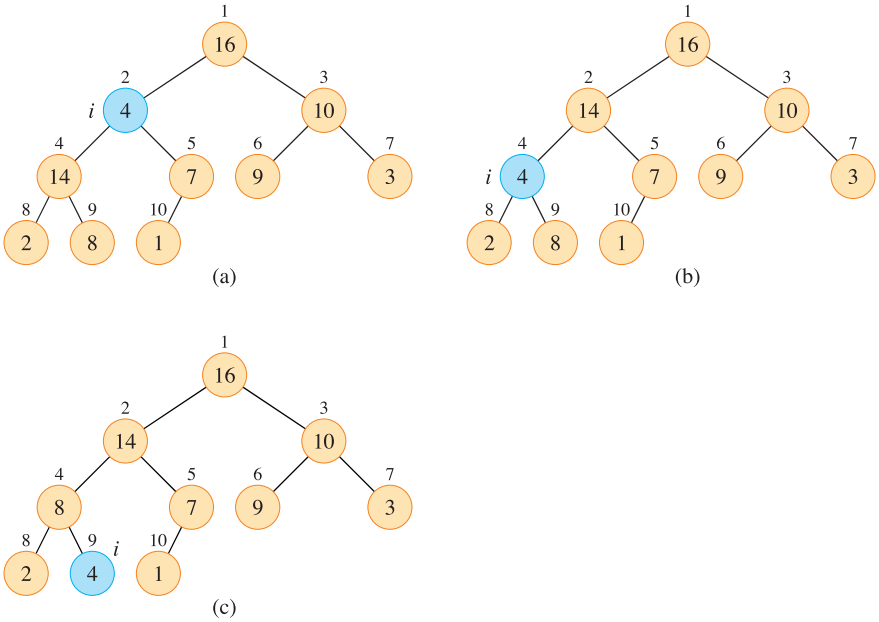
(b)

5.2　保持堆的性质

程序 5.2: 保持堆的性质

```
public static class Heap {
    public int[] A;
    public int heapSize;

    public Heap(int[] a) {
        A = a;
    }

    public int parent(int i) {/* 代码见5.1 */}

    public int left(int i) {/* 代码见5.1 */}

    public int right(int i) {/* 代码见5.1 */}


    public void swap(int i, int j) {
        int tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
    }

    public void MaxHeapify(int i) {
        int l = left(i);
        int r = right(i);
        int largest;
        if (l < heapSize && A[l] > A[i])
            largest = l;
        else
            largest = i;
        if (r < heapSize && A[r] > A[largest])
            largest = r;
        if (largest != i) {
            swap(i, largest);
            MaxHeapify(largest);
```

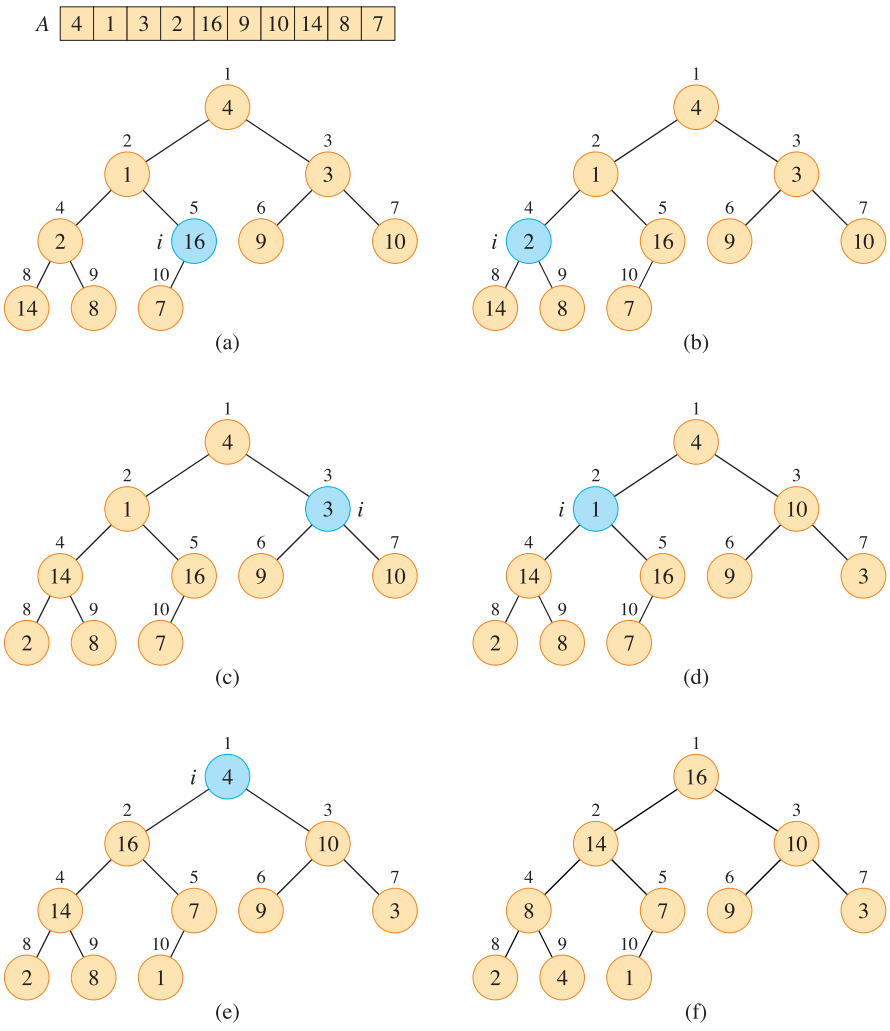
```
35          }
36      }
37  }
38
39
```



5.3　建堆

程序 5.3: 建堆

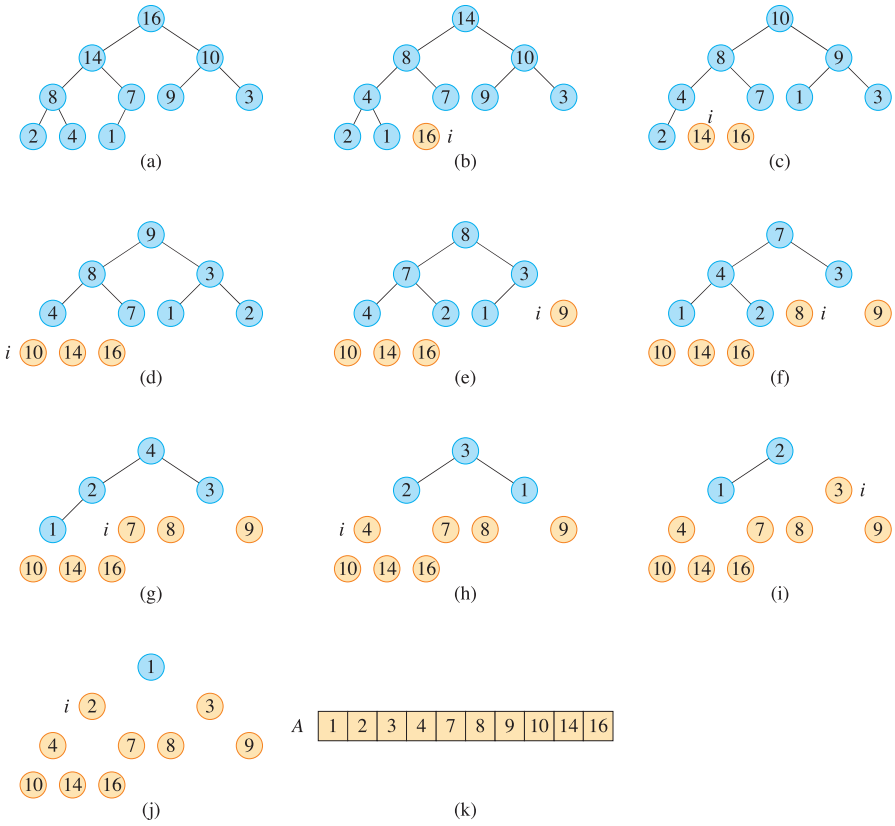
```
1  public void BuildMaxHeap(int n) {
2      heapSize = n;
3      for (int i = n / 2; i >= 0; i--)
4          MaxHeapify(i);
5  }
```

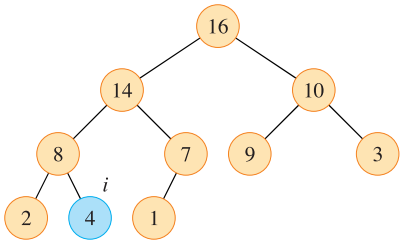

A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

5.4　堆排序算法

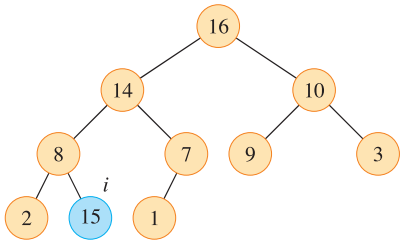
程序 5.4:

```java
public class InsertionSortAlgorithm {
    public static void InsertionSort(int[] A) {
        for (int i = 1; i < A.length; i++) {
            int key = A[i];
            // 将 A[i] 插入到已经排好序的子数组 A[0:i-1]
            int j = i - 1;
            while (j > -1 && A[j] > key) {
                A[j + 1] = A[j];
                j = j - 1;
            }
            A[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] A = {5,2,4,6,1,3};
        InsertionSort(A);
        for (var i : A)
            System.out.println(i);
    }
}
```

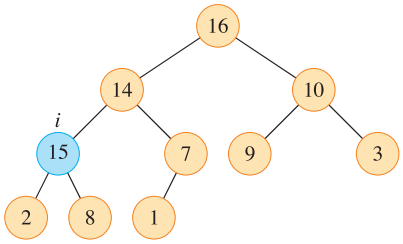

16 14 10 8 7 9 3 2 4 1
(a)
14 8 10 4 7 9 3 2 1 16 i
(b)
10 8 9 4 7 1 3 2 i 14 16
(c)
9 8 3 4 7 1 2 i 10 14 16
(d)
8 7 3 4 2 1 i 9 10 14 16
(e)
7 4 3 1 2 8 i 9 10 14 16
(f)
4 2 3 1 i 7 8 9 10 14 16
(g)
3 2 1 i 4 7 8 9 10 14 16
(h)
2 1 3 i 4 7 8 9 10 14 16
(i)
1 i 2 3 4 7 8 9 10 14 16
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)


16
14
10
8
7
9
3
i
2
4
1

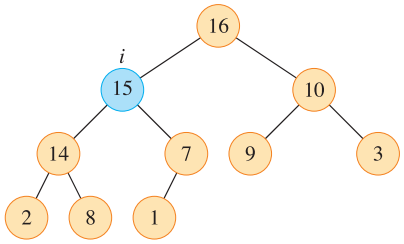
(a)


16
14
10
8
7
9
3
i
2
15
1

(b)


16
14
10
i
15
7
9
3
2
8
1

(c)


16
i
15
10
14
7
9
3
2
8
1

(d)

第六章　快速排序

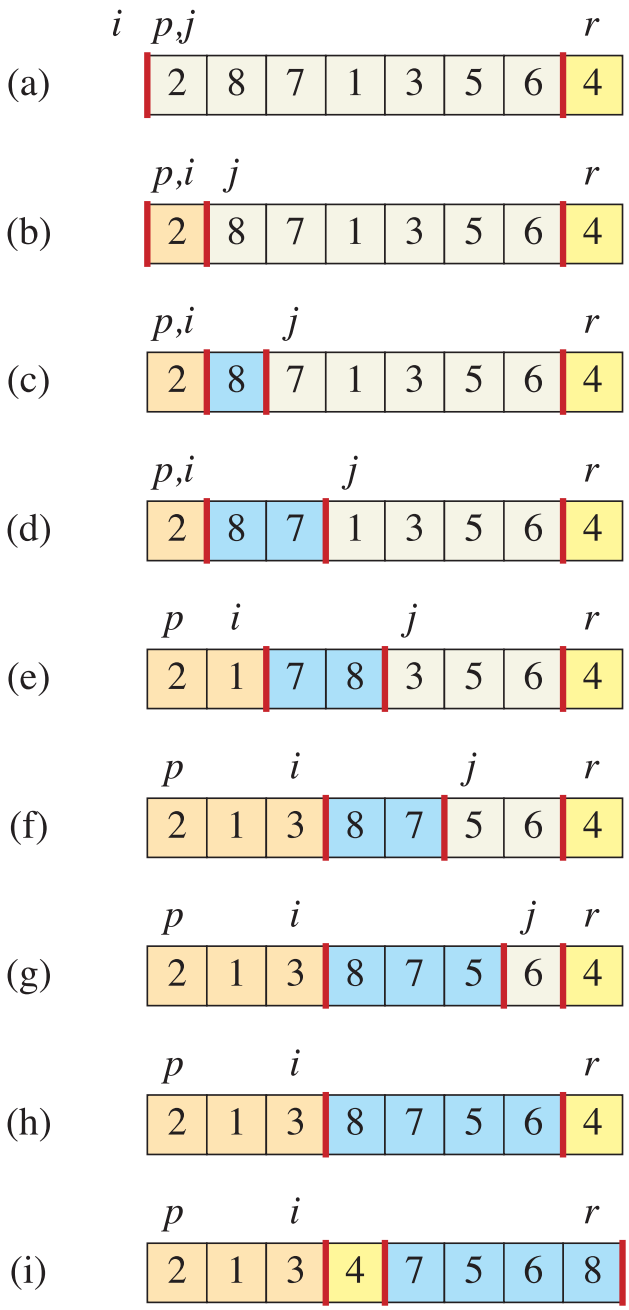
6.1　快速排序的描述

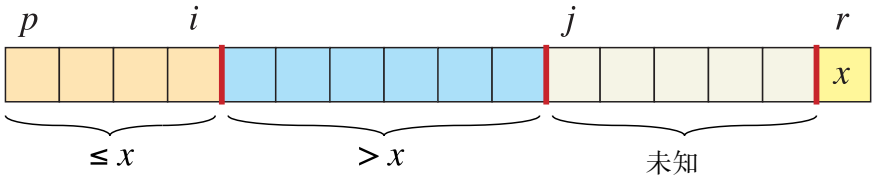
程序 6.1: 快速排序的主程序

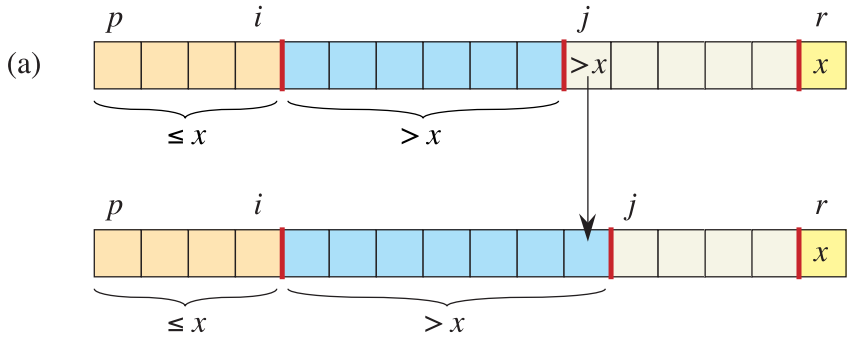
```
public static void QuickSort(int[] A, int p, int r) {
    if (p < r) {
        var q = Partition(A, p, r);
        QuickSort(A, p, q - 1);
        QuickSort(A, q + 1, r);
    }
}
```

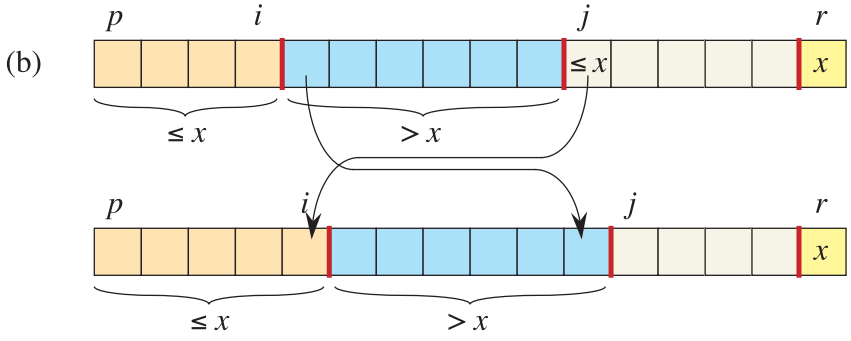
程序 6.2: 快速排序的分割过程

```
public class RecursiveFibAlgorithm {
    public long RecursiveFib(int n) {
        if (n == 0 || n == 1)
            return n;
        else
            return RecursiveFib(n - 1) + RecursiveFib(n - 2);
    }

    public static void main(String[] args) {
        System.out.println(RecursiveFib(5));
    }
}
```


i p,j r
(a) 2 8 7 1 3 5 6 4
p,i j r
(b) 2 8 7 1 3 5 6 4
p,i j r
(c) 2 8 7 1 3 5 6 4
p,i j r
(d) 2 8 7 1 3 5 6 4
p i j r
(e) 2 1 7 8 3 5 6 4
p i j r
(f) 2 1 3 8 7 5 6 4
p i j r
(g) 2 1 3 8 7 5 6 4
p i r
(h) 2 1 3 8 7 5 6 4
p i r
(i) 2 1 3 4 7 5 6 8


p
i
j
r
x
≤ x
> x
未知


(a)
p
i
j
r
> x
x
≤ x
> x
p
i
j
r
x
≤ x
> x


(b)
p
i
j
r
≤ x
x
≤ x
> x
p
i
j
r
x
≤ x
> x

6.2　快速排序的性能

6.3　随机化版本的快速排序

程序 6.3: 随机化版本的快速排序

```
1  import java.util.Random;
2
3  public class RandomizedQuickSortAlgorithm {
4      public static void swap(int[] A, int i, int j) {
5          var tmp = A[i];
6          A[i] = A[j];
7          A[j] = tmp;
8      }
9
10     public static int Partition(int[] A, int p, int r) {
11         var x = A[r];
12         var i = p - 1;
13         for (int j = p; j < r; j++) {
14             if (A[j] < x) {
15                 i++;
16                 swap(A, i, j);
17             }
18         }
19         swap(A, i + 1, r);
20         return i + 1;
21     }
22
23     public static int RandomizedPartition(int[] A, int p, int r)
       ↪ {
24         Random random = new Random();
25         int i = random.nextInt(r - p) + p;
26         swap(A, i, r);
27         return Partition(A, p, r);
28     }
29     public static void RandomizedQuickSort(int[] A, int p, int r)
       ↪ {
```

```
30          if (p < r) {
31              int q = RandomizedPartition(A, p, r);
32              RandomizedQuickSort(A, p, q - 1);
33              RandomizedQuickSort(A, q + 1, r);
34          }
35      }
36
37      public static void main(String[] args) {
38          int[] A = {2,8,7,1,3,5,6,4};
39          RandomizedQuickSort(A, 0, 7);
40          for (var i : A)
41              System.out.println(i);
42      }
43  }
```

6.4　快速排序分析

第七章　基于比较的排序算法的下界

第八章　计数排序

程序 8.1: 计数排序

```java
public class CountingSortAlgorithm {
    public static void CountingSort(int[] A, int[] B, int k) {
        var C = new int[k + 1];
        for (int i = 0; i < A.length; i++)
            C[A[i]] = C[A[i]] + 1;
        for (int j = 1; j < k + 1; j++)
            C[j] = C[j] + C[j - 1];
        for (int i = A.length - 1; i >= 0; i--) {
            B[C[A[i]] - 1] = A[i];
            C[A[i]] = C[A[i]] - 1;
        }
    }

    public static void main(String[] args) {
        int[] A = {2,5,3,0,2,3,0,3};
        int[] B = new int[A.length];
        int k = 5;
        CountingSort(A, B, k);
        for (var i : B)
            System.out.println(i);
    }
}
```

第九章　中位数和顺序统计量

9.1　最小值和最大值

程序 9.1: 求数组最小值

```java
public class MinimumValueAlgorithm {
    public static int Minimum(int[] A) {
        int min = A[0];
        for (int i = 1; i < A.length; i++)
            if (min > A[i])
                min = A[i];
        return min;
    }

    public static void main(String[] args) {
        int[] A = {4,1,3,2,16,9,10,14,8,7};
        System.out.println(Minimum(A));
    }
}
```

9.2　期望为线性时间的快速选择算法

程序 9.2: 快速选择算法

```java
import java.util.Random;

public class RandomizedSelectAlgorithm {
    public static void swap(int[] A, int i, int j) {
```

```
5           var tmp = A[i];
6           A[i] = A[j];
7           A[j] = tmp;
8       }
9
10      public static int Partition(int[] A, int p, int r) {
11          var x = A[r];
12          var i = p - 1;
13          for (int j = p; j < r; j++) {
14              if (A[j] < x) {
15                  i++;
16                  swap(A, i, j);
17              }
18          }
19          swap(A, i + 1, r);
20          return i + 1;
21      }
22      public static int RandomizedPartition(int[] A, int p, int r)
        ↪ {
23          Random random = new Random();
24          int i = random.nextInt(r - p) + p;
25          swap(A, i, r);
26          return Partition(A, p, r);
27      }
28      public static int RandomizedSelect(int[] A, int p, int r, int
        ↪ i) {
29          if (p == r) return A[p];
30          int q = RandomizedPartition(A, p, r);
31          int k = q - p + 1;
32          if (i == k)
33              return A[q];
34          else if (i < k)
35              return RandomizedSelect(A, p, q- 1, i);
36          else
37              return RandomizedSelect(A, q + 1, r, i - k);
38      }
39
```

```
40      public static void main(String[] args) {
41          int[] A = {6,19,4,12,14,9,15,7,8,11,3,13,2,5,10};
42          int result = RandomizedSelect(A, 0, 14, 5);
43          System.out.println(result);
44      }
45  }
```

第三部分

数据结构

第十章　基本数据结构

10.1　基于数组实现的数据结构：数组，矩阵，栈和队列

10.1.1　数组

程序 10.1: 动态数组的实现

```
import java.util.Random;

public class RandomizedQuickSortAlgorithm {
    public static void swap(int[] A, int i, int j) {
        var tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
    }

    public static int Partition(int[] A, int p, int r) {
        var x = A[r];
        var i = p - 1;
        for (int j = p; j < r; j++) {
            if (A[j] < x) {
                i++;
                swap(A, i, j);
            }
        }
        swap(A, i + 1, r);
        return i + 1;
```

```
    }

    public static int RandomizedPartition(int[] A, int p, int r)
    ↪ {
        Random random = new Random();
        int i = random.nextInt(r - p) + p;
        swap(A, i, r);
        return Partition(A, p, r);
    }
    public static void RandomizedQuickSort(int[] A, int p, int r)
    ↪ {
        if (p < r) {
            int q = RandomizedPartition(A, p, r);
            RandomizedQuickSort(A, p, q - 1);
            RandomizedQuickSort(A, q + 1, r);
        }
    }

    public static void main(String[] args) {
        int[] A = {2,8,7,1,3,5,6,4};
        RandomizedQuickSort(A, 0, 7);
        for (var i : A)
            System.out.println(i);
    }
}
```

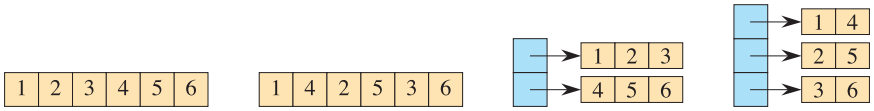
我们可以测试一下:

程序 10.2: 动态数组的测试程序

```
public class MyArrayListTest {
    public static void main(String[] args) {
        var arr = new MyArrayList<String>();
        arr.add("algorithm");
        System.out.println(arr.get(0));
        arr.remove(0);
        System.out.println(arr.get(0));
```

```
    }
}
```

10.1.2　矩阵



(a)　(b)　(c)　(d)

10.1.3　栈

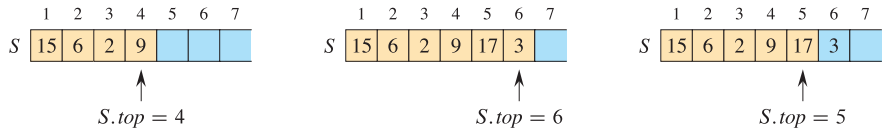
程序 10.3: 使用数组实现栈

```
public class MyStack<T> {
    private int top;
    private int size;
    private T[] S;

    public MyStack(int size) {
        this.size = size;
        S = (T[])new Object[size];
        top = -1;
    }

    public boolean isEmpty() {
        if (top == -1) return true;
        else return false;
    }

    public void push(T x) {
        if (top == size - 1)
            throw new RuntimeException(" 栈溢出");
        else {
```

```
21              S[++top] = x;
22          }
23      }
24
25      public T pop() {
26          if (isEmpty())
27              throw new RuntimeException(" 不能对空栈执行弹出操作");
28          else {
29              return S[top--];
30          }
31      }
32  }
```

程序 10.4: 栈的测试程序

```
1   public class MyStackTest {
2       public static void main(String[] args) {
3           var stack = new MyStack<Integer>(2);
4           stack.push(1);
5           stack.push(2);
6           System.out.println(stack.pop());
7           stack.push(3);
8       }
9   }
```



(a)　(b)　(c)

10.1.4　队列

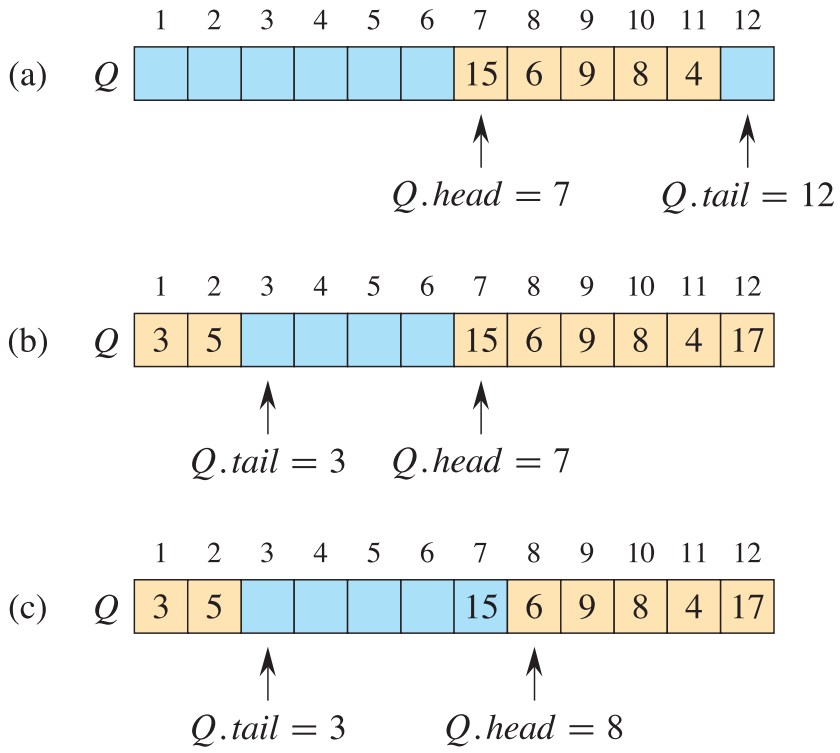
程序 10.5: 使用数组实现队列

```
public class MyQueue<T> {
    private int head;
    private int tail;
    private int size;
    private T[] Q;

    public MyQueue(int size) {
        this.size = size;
        head = tail = 0;
        Q = (T[])new Object[size];
    }

    public void Enqueue(T x) {
        Q[tail] = x;
        if (tail == size - 1)
            tail = 0;
        else
            tail++;
    }

    public T Dequeue() {
        T x = Q[head];
        if (head == size - 1)
            head = 0;
        else
            head++;
        return x;
    }
}
```

程序 10.6: 环形队列测试程序

```
public class MyQueueTest {
    public static void main(String[] args) {
```

```
        var queue = new MyQueue<Integer>(10);
        queue.Enqueue(10);
        queue.Enqueue(20);
        System.out.println(queue.Dequeue());
        System.out.println(queue.Dequeue());
    }
}
```



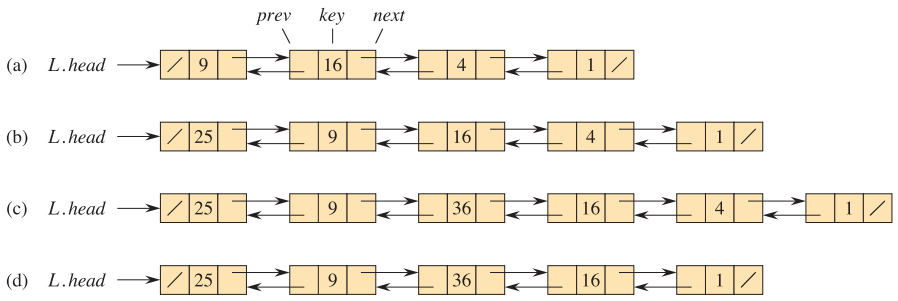
10.2　链表

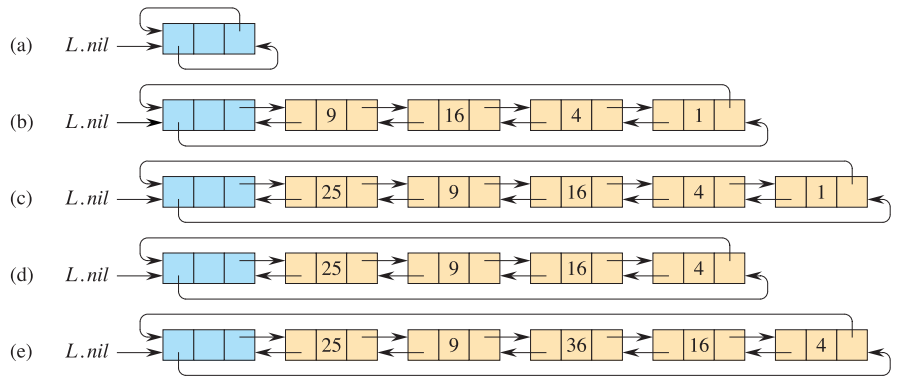
程序 10.7: 双向链表的实现

```
public class MyLinkedList<T> {
    private int theSize;
    private int modCount = 0;
    private Node<T> beginMarker;
    private Node<T> endMarker;

    private static class Node<T> {
        public T data;
        public Node<T> prev;
        public Node<T> next;

        public Node(T d, Node<T> p, Node<T> n) {
            data = d;
            prev = p;
            next = n;
        }
    }

    public MyLinkedList() {
        doClear();
    }

    public void clear() {
        doClear();
    }

    private void doClear() {
        beginMarker = new Node<T>(null, null, null);
        endMarker = new Node<T>(null, beginMarker, null);
        beginMarker.next = endMarker;

        theSize = 0;
        modCount++;
    }
```

```
35
36      public int size() {
37          return theSize;
38      }
39
40      public boolean isEmpty() {
41          return size() == 0;
42      }
43
44      public boolean add(T x) {
45          add(size(), x);
46          return true;
47      }
48
49      public void add(int idx, T x) {
50          addBefore(getNode(idx, 0, size()), x);
51      }
52
53      public T get(int idx) {
54          return getNode(idx).data;
55      }
56
57      public T set(int idx, T newVal) {
58          Node<T> p = getNode(idx);
59          T oldVal = p.data;
60          p.data = newVal;
61          return oldVal;
62      }
63
64      public T remove(int idx) {
65          return remove(getNode(idx));
66      }
67
68      private void addBefore(Node<T> p, T x) {
69          Node<T> newNode = new Node<>(x, p.prev, p);
70          newNode.prev.next = newNode;
71          p.prev = newNode;
```

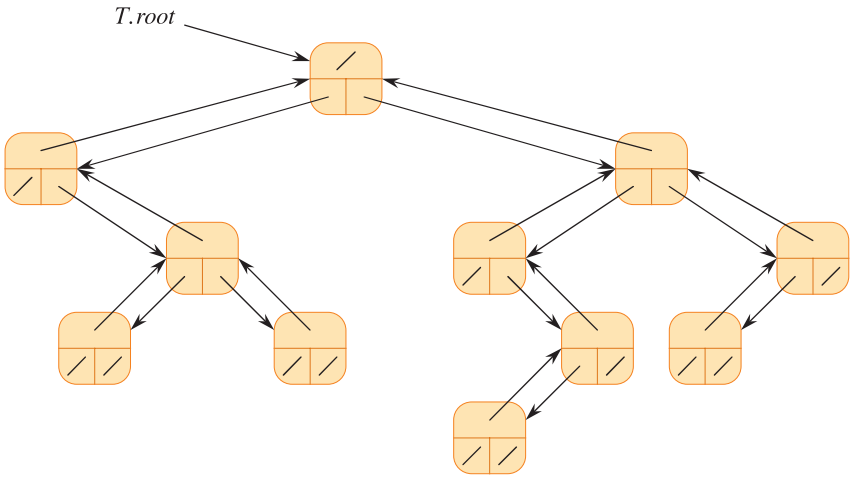
```
72          theSize++;
73          modCount++;
74      }
75
76      private T remove(Node<T> p) {
77          p.next.prev = p.prev;
78          p.prev.next = p.next;
79          theSize--;
80          modCount++;
81
82          return p.data;
83      }
84
85      private Node<T> getNode(int idx) {
86          return getNode(idx, 0, size() - 1);
87      }
88
89      private Node<T> getNode(int idx, int lower, int upper) {
90          Node<T> p;
91
92          if (idx < lower || idx > upper)
93              throw new IndexOutOfBoundsException();
94
95          if (idx < size() / 2) {
96              p = beginMarker.next;
97              for (int i = 0; i < idx; i++)
98                  p = p.next;
99          } else {
100             p = endMarker;
101             for (int i = size(); i > idx; i--)
102                 p = p.prev;
103         }
104
105         return p;
106     }
107 }
```

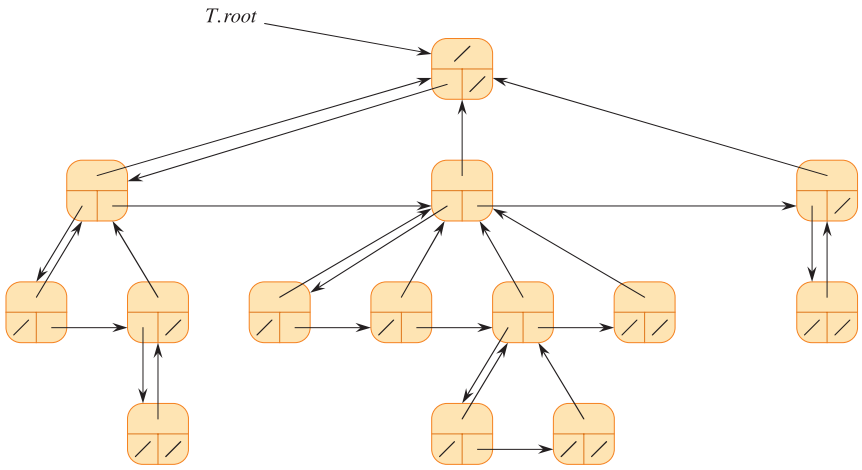

prev
key
next
(a) L.head
9
16
4
1
(b) L.head
25
9
16
4
1
(c) L.head
25
9
36
16
4
1
(d) L.head
25
9
36
16
1


(a) L.nil
(b) L.nil
9
16
4
1
(c) L.nil
25
9
16
4
1
(d) L.nil
25
9
16
4
(e) L.nil
25
9
36
16
4

10.3　指针和对象的实现

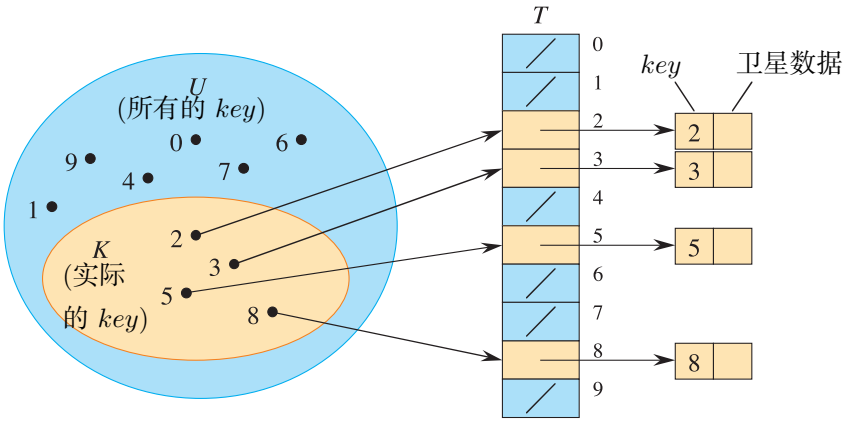
10.4　有根树的表示方式




T.root

第十一章　哈希表

11.1　直接寻址表



程序 11.1:

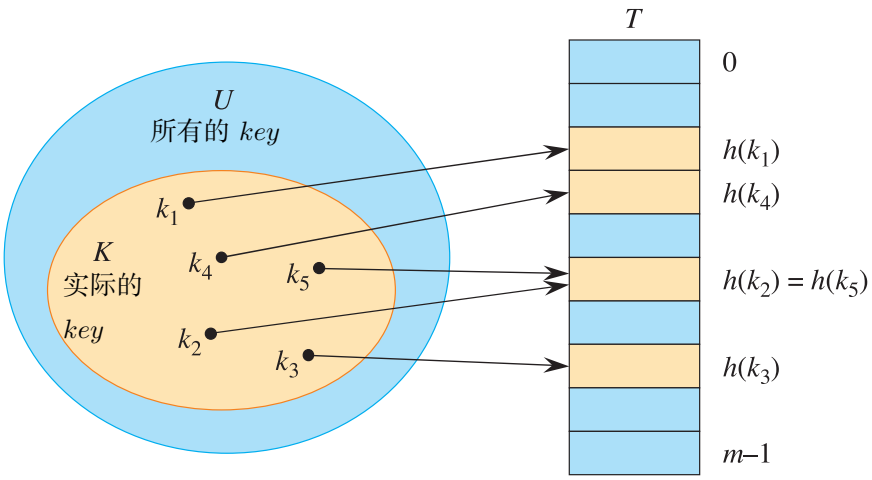
```
public class TestMyArrayList {
    public static void main(String[] args) {
        var arr = new MyArrayList<String>();
        arr.add("algorithm");
        System.out.println(arr.get(0));
        arr.remove(0);
        System.out.println(arr.get(0));
    }
}
```

11.2　哈希表

11.3　哈希函数

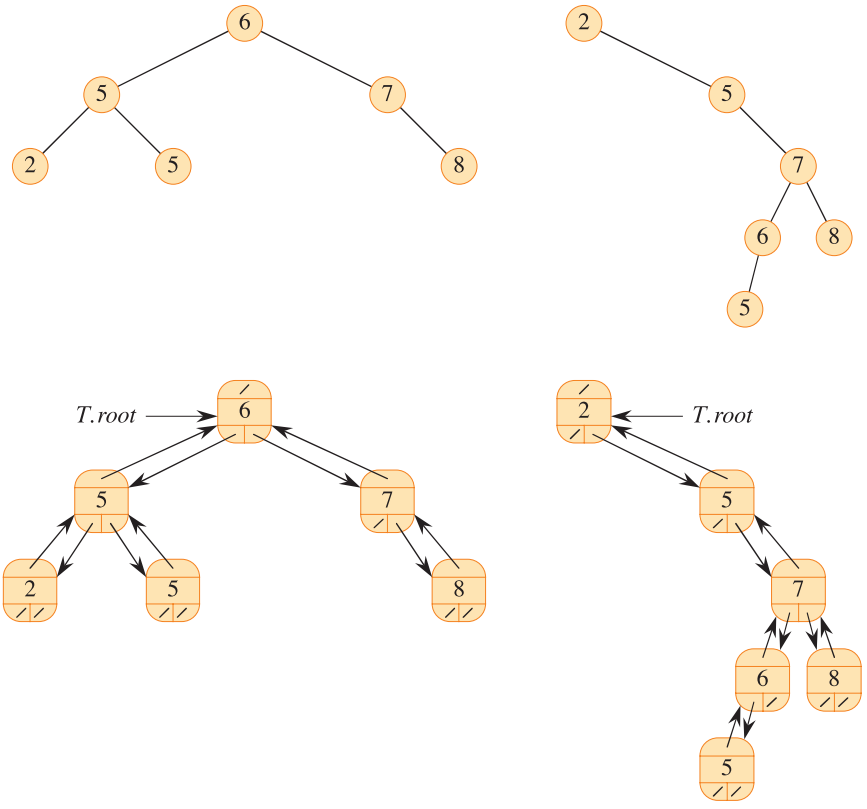
11.4　开放定址

11.5　实际中的考虑



第十二章　二叉搜索树

12.1　什么是二叉搜索树

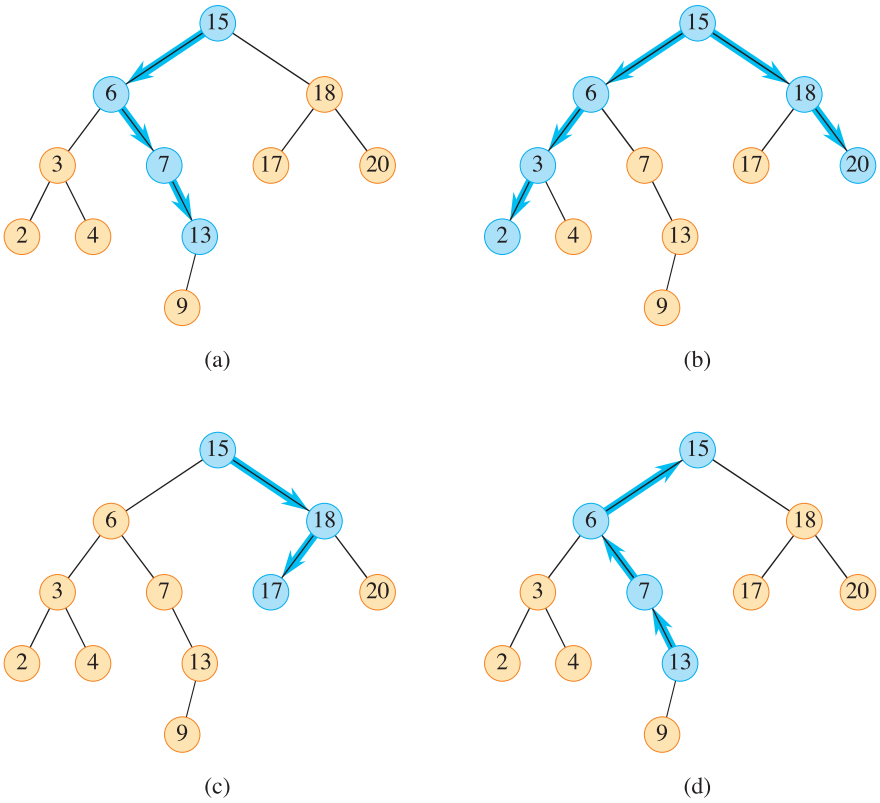


(a)　(b)

程序 12.1:

```
public class TreeWalkAlgorithm {
    public static class TreeNode {
        public int key;
        public TreeNode left;
        public TreeNode right;

        public TreeNode(int key) {
            this.key = key;
        }
    }

    public void InOrderTreeWalk(TreeNode x) {
        if (x != null) {
            InorderTreeWalk(x.left);
            System.out.println(x.key);
            InorderTreeWalk(x.right);
        }
    }

    public void PreOrderTreeWalk(TreeNode x) {
        if (x != null) {
            System.out.println(x.key);
            PreOrderTreeWalk(x.left);
            PreOrderTreeWalk(x.right);
        }
    }

    public void PostOrderTreeWalk(TreeNode x) {
        if (x != null) {
            PostOrderTreeWalk(x.left);
            PostOrderTreeWalk(x.right);
            System.out.println(x.key);
        }
    }
}
```

12.2　二叉搜索树的查找



程序 12.2:

```
1  public class TreeSearchAlgorithm {
2      public static int TreeSearch(TreeNode x, int k) {
3          if (x == null || k == x.key)
4              return x;
5          if (k < x.key)
6              return TreeSearch(x.left, k);
7          else
8              return TreeSearch(x.right, k);
9      }
```

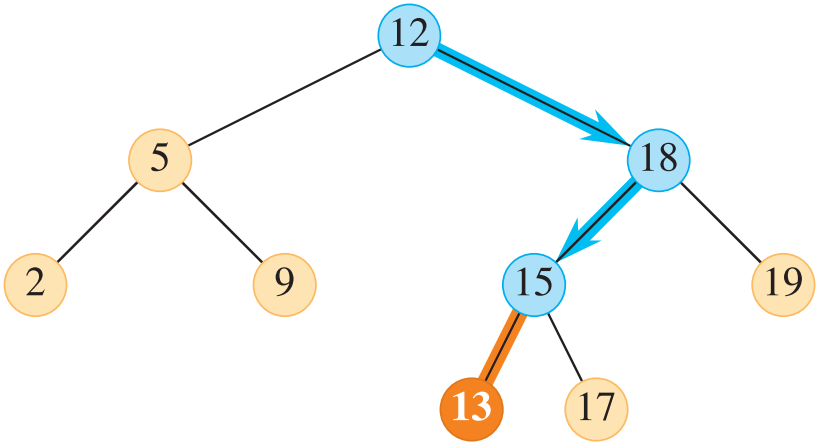
```
10
11      public static int IterativeTreeSearch(TreeNode x, int k) {
12          while (x != null && k != x.key) {
13              if (k < x.key)
14                  x = x.left;
15              else
16                  x = x.right;
17          }
18
19          return x;
20      }
21
22      public static TreeMinimum(TreeNode x) {
23          while (x.left != null) x = x.left;
24          return x;
25      }
26
27      public static TreeMaximum(TreeNode x) {
28          while (x.right != null) x = x.right;
29          return x;
30      }
31
32      public static TreeSuccessor(TreeNode x) {
33          if (x.right != null)
34              return TreeMinimum(x.right);
35          else {
36              y = x.p;
37              while (y != null && x == y.right) {
38                  x = y;
39                  y = y.p;
40              }
41              return y;
42          }
43      }
44  }
```

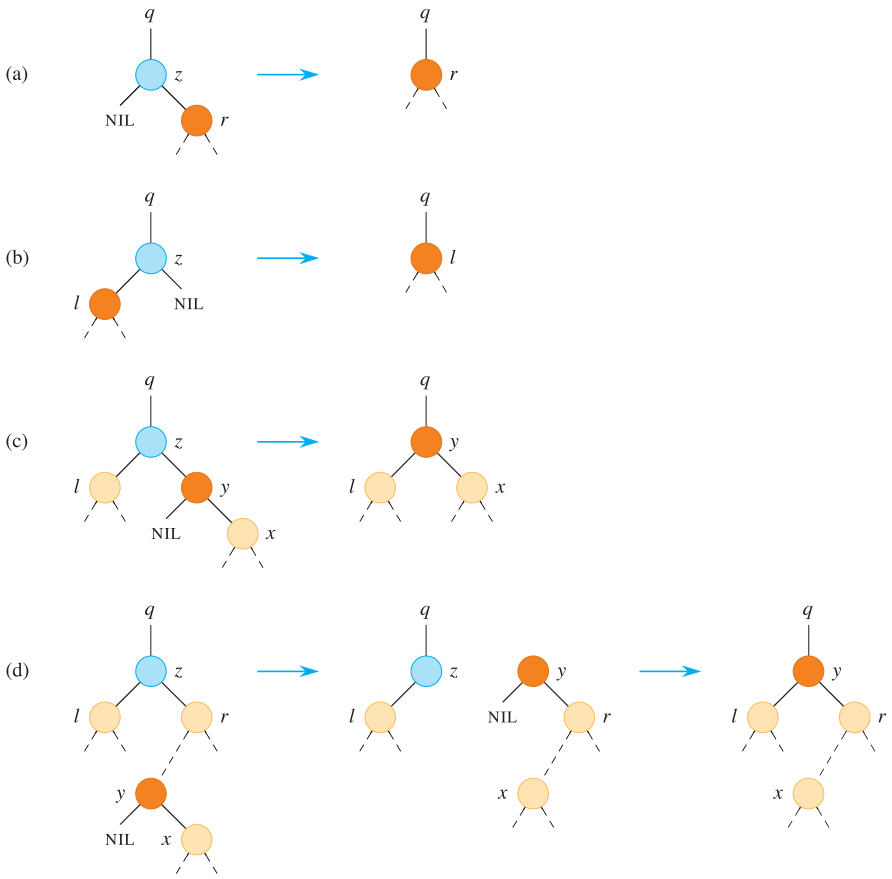
12.3　二叉搜索树的插入和删除

程序 12.3:

```
1  public class TreeWalkAlgorithm {
2      public static class TreeNode {
3          public int key;
4          public TreeNode left;
5          public TreeNode right;
6
7          public TreeNode(int key) {
8              this.key = key;
9          }
10     }
11
12     public void InOrderTreeWalk(TreeNode x) {
13         if (x != null) {
14             InorderTreeWalk(x.left);
15             System.out.println(x.key);
16             InorderTreeWalk(x.right);
17         }
18     }
19
20     public void PreOrderTreeWalk(TreeNode x) {
21         if (x != null) {
22             System.out.println(x.key);
23             PreOrderTreeWalk(x.left);
24             PreOrderTreeWalk(x.right);
25         }
26     }
27
28     public void PostOrderTreeWalk(TreeNode x) {
29         if (x != null) {
30             PostOrderTreeWalk(x.left);
31             PostOrderTreeWalk(x.right);
32             System.out.println(x.key);
33         }
34     }
```

```
35 }
```




(a)
q
z
NIL
r
q
r
(b)
q
z
l
NIL
q
l
(c)
q
z
l
y
NIL
x
q
y
l
x
(d)
q
z
l
r
y
NIL
x
q
z
l
y
NIL
r
x
q
y
l
r
x

第十三章　红黑树

第 十二 章介绍了一棵高度为 h 的二叉搜索树, 它可以支持任何一种基本动态集合操作, 如 $SEARCH$, $PREDECESSOR$, $SUCCESSOR$, $MINIMUM$, $MAXIMUM$, $INSERT$ 和 $DELETE$ 等, 其时间复杂度均为 $O(h)$. 因此, 如果搜索树的高度较低时, 这些集合操作会执行得较快. 然而, 如果树的高度较高时, 这些集合操作可能并不比在链表上执行得快. 红黑树 (Red-Black Tree) 是许多 "平衡" 搜索树中的一种, 可以保证在最坏情况下基本动态集合操作的时间复杂度为 $O(\lg n)$.

13.1　红黑树的性质

红黑树是一棵二叉搜索树, 它在每个结点上增加了一个存储位来表示结点的颜色, 可以是 RED 或 $BLACK$. 通过对任何一条从根到叶子的简单路径上各个结点的颜色进行约束, 红黑树确保没有一条路径会比其他路径长出 2 倍, 因而是近似于**平衡的**.

树中每个结点包含 5 个属性: $color, key, left, right$ 和 p. 如果一个结点没有子结点或父结点, 则该结点相应指针属性的值为 **NIL**. 我们可以把这些 **NIL** 视为指向二叉搜索树的叶结点 (外部结点) 的指针, 而把带 key 的结点视为树的内部结点.

结点定义如代码13.1所示.

程序 13.1: 红黑树结点的定义

```
1 public class RBTreeNode {
2     public int key; // 结点中保存的数据
3     public RBTreeNode left; // 指向左孩子的引用
4     public RBTreeNode right; // 指向右孩子的引用
```

```
5     public RBTreeNode p; // 指向父亲结点的引用
6     public int color; // 红色是 1, 黑色是 0.
7 }
```

一棵红黑树是满足下面**红黑性质**的二叉搜索树:

1. 每个结点或是红色的, 或是黑色的.
2. 根结点是黑色的.
3. 每个叶结点 (**NIL**) 是黑色的.
4. 如果一个结点是红色的, 则它的两个子结点都是黑色的.
5. 对每个结点, 从该结点到其所有后代叶结点的简单路径上, 均包含相同数目的黑色结点.

图 13.1(a) 显示了一个红黑树的例子.

为了便于处理红黑树代码中的边界条件, 使用一个哨兵来代表 **NIL** (参见 10.2 节). 对于一棵红黑树 T, 哨兵 $T.nil$ 是一个与树中普通结点有相同属性的对象. 它的 $color$ 属性为 $BLACK$, 而其他属性 $p, left, right$ 和 key 可以设为任意值. 如图 13.1(b) 所示, 所有指向 **NIL** 的指针都用指向哨兵 $T.nil$ 的指针替换.

使用哨兵后, 就可以将结点 x 的 **NIL** 孩子视为一个普通结点, 其父结点为 x. 尽管可以为树内的每一个 **NIL** 新增一个不同的哨兵结点, 使得每个 **NIL** 的父结点都有良好的定义, 但这种做法会浪费空间. 取而代之的是, 使用一个哨兵 $T.nil$ 来代表所有的 **NIL**: 所有的叶结点和根结点的父结点. 哨兵的属性 $p, left, right$ 和 key 的取值并不重要, 尽管为了方便起见可以在程序中设定它们. 红黑树的 Java 类见代码13.2.

程序 13.2: 红黑树的 Java 类

```
1 public class RBTree {
2     // 颜色的枚举定义
3     public static final int RED = 1;
4     public static final int BLACK = 0;
5 
6     public RBTreeNode root; // 根节点的引用
7     public RBTreeNode nil;  // nil 结点
```

```
8
9       public RBTree() {
10          nil = new RBTreeNode();
11          nil.color = 0; // T.nil 结点是黑色的.
12          nil.left = null;
13          nil.right = null;
14          root = nil;
15      }
16  }
```

我们通常将注意力放在红黑树的内部结点上, 因为它们存储了 key 的值. 在本章的后面部分, 所画的红黑树都忽略了叶结点, 如图 13.1(c) 所示.

从某个结点 x 出发 (不含该结点) 到达一个叶结点的任意一条简单路径上的黑色结点个数称为该结点的黑高 (black-height) , 记为 $bh(x)$. 根据性质 5, 黑高的概念是明确定义的, 因为从该结点出发的所有下降到其叶结点的简单路径的黑结点个数都相同. 于是定义红黑树的黑高为其根结点的黑高.

下面的引理说明了为什么红黑树是一种好的搜索树.

引理 13.1. 一棵有 n 个内部结点的红黑树的高度至多为 $2\lg(n+1)$.

证明. 先证明以任一结点 x 为根的子树中至少包含 $2^{bh(x)}-1$ 个内部结点. 要证明这点, 对 x 的高度进行归纳. 如果 x 的高度为 0, 则 x 必为叶结点 ($T.nil$), 且以 x 为根结点的子树至少包含 $2^{bh(x)}-1=2^0-1=0$ 个内部结点. 对于归纳步骤, 考虑一个高度为正值且有两个子结点的内部结点 x. 每个子结点有黑高 $bh(x)$ 或 $bh(x)-1$, 其分别取决于自身的颜色是红还是黑. 由于 x 子结点的高度比 x 本身的高度要低, 可以利用归纳假设得出每个子结点至少有 $2^{bh(x)-1}$ 个内部结点的结论. 于是, 以 x 为根的子树至少包含 $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$ 个内部结点, 因此得证.

为完成引理的证明, 设 h 为树的高度. 根据性质 4, 从根到叶结点 (不包括根结点) 的任何一条简单路径上都至少有一半的结点为黑色. 因此, 根的黑高至少为 $h/2$; 于是有

$$n \geq 2^{h/2}-1$$

把 1 移到不等式的左边, 再对两边取对数, 得到 $\lg(n+1) \geq h/2$, 或者 $h \leq 2\lg(n+1)$. □

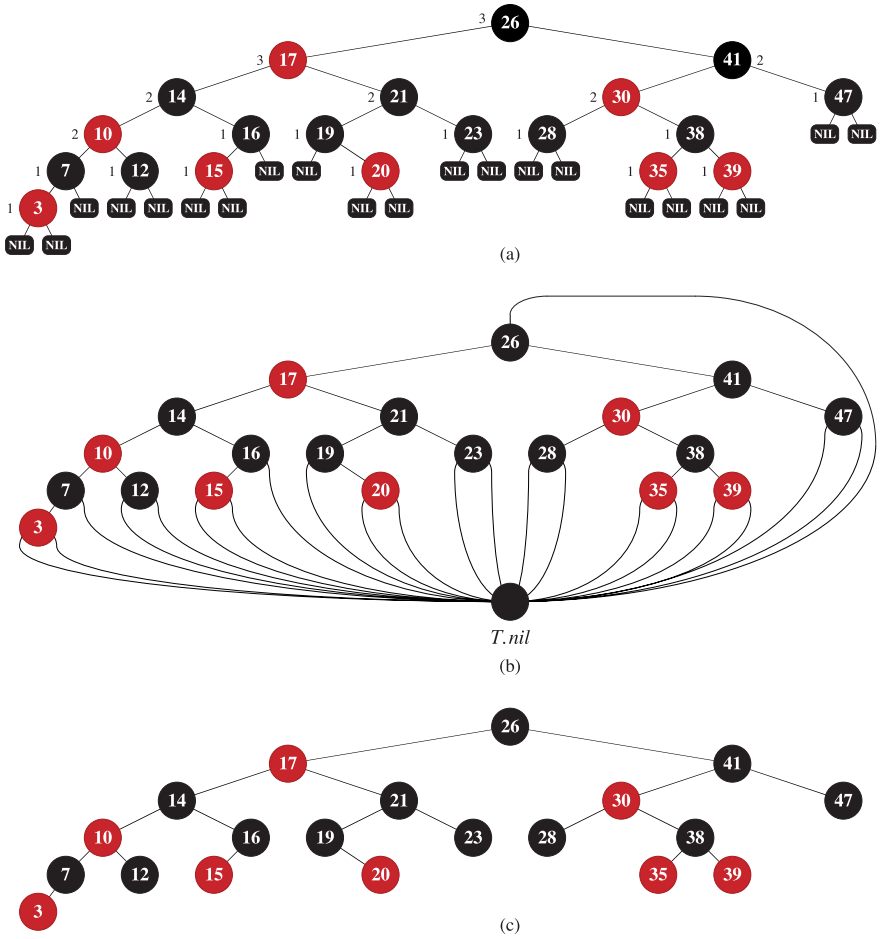


图 13.1: 一棵红黑树, 其中黑结点涂黑, 红结点以浅阴影表示. 在一棵红黑树内, 每个结点或红或黑, 红结点的两个子结点都是黑色, 且从每个结点到其后代叶结点的每条简单路径上, 都包含相同数目的黑结点. (a) 每个标为 **NIL** 的叶结点都是黑的. 每个非 **NIL** 结点都标上它的黑高; **NIL** 的黑高为 0. (b) 同样的这棵红黑树, 不是用一个个的 **NIL** 表示, 而用一个总是黑色的哨兵 $T.nil$ 来代替, 它的黑高也被省略. 根结点的父结点也是这个哨兵. (c) 同样的这棵红黑树, 其叶结点与根结点的父结点全部被省略. 本章的其余部分也采用这种画图方式

由该引理可知, 动态集合操作 $SEARCH$, $MINIMUM$, $MAXIMUM$, $SUCCESSOR$ 和 $PREDECESSOR$ 可在红黑树上在 $O(\lg n)$ 时间内执行, 因为这些操作在一棵高度为 h 的二叉搜索树上的运行时间为 $O(h)$(参见第 十二 章), 而任何包含 n 个结点的红黑树又都是高度为 $O(\lg n)$ 的二叉搜索树.(当然, 在第 十二 章的算法中, **NIL** 的引用必须用 $T.nil$ 来代替.) 虽然当给定一棵红黑树作为输入时, 第 十二 章的算法 $TreeInsert$ 和 $TreeDelete$ 的运行时间为 $\lg n$, 但是这两个算法并不直接支持动态集合操作 $INSERT$ 和 $DELETE$, 因为它们并不能保证被这些操作修改过的二叉搜索树仍是红黑树. 那么如何在时间 $O(\lg n)$ 内支持这两个操作呢, 我们将在 13.3 节和 13.4 节中介绍.

13.2　旋转

搜索树操作 $TreeInsert$ 和 $TreeDelete$ 在含 n 个关键字的红黑树上, 运行花费时间为 $O(\lg n)$. 由于这两个操作对树做了修改, 结果可能违反 13.1 节中列出的红黑性质. 为了维护这些性质, 必须要改变树中某些结点的颜色以及指针结构.

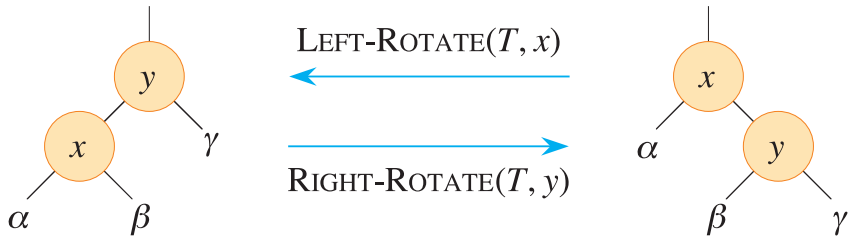


图 13.2: 二叉搜索树上的旋转操作. 操作 $LeftRotate(T, x)$ 通过改变常数数目的指针, 可以将右边两个结点的结构转变成左边的结构. 左边的结构可以使用相反的操作 $RightRotate(T, y)$ 来转变成右边的结构. 字母 α, β 和 γ 代表任意的子树. 旋转操作保持了二叉搜索树的性质: α 的 key 在 $x.key$ 之前, $x.key$ 在 β 的 key 之前, γ 的 key 在 $y.key$ 之前, $y.key$ 在 y 的 key 之前

指针结构的修改是通过**旋转** (rotation) 来完成的, 这是一种能保持二叉搜索树性质的搜索树局部操作. 图 13.2 中给出了两种旋转: 左旋和右旋. 当在某个结点 x 上做左旋时, 假设它的右孩子为 y 而不是 $T.nil$; x 可以为其右孩子不是 $T.nil$ 结点的树内任意节点. 左旋以 x 到 y 的链为 “支轴” 进行. 它使 y 成为该

子树新的根节点, x 成为 y 的左孩子, y 的左孩子成为 x 的右孩子.

在 $LeftRotate$ 的代码13.3中, 假设 $x.right \neq T.nil$ 且根结点的父结点为 $T.nil$.

程序 13.3: 左旋的实现

```
public void LeftRotate(RBTreeNode x) {
    var y = x.right;
    x.right = y.left;
    if (y.left != nil)
        y.left.p = x;
    y.p = x.p;
    if (x.p == nil)
        root = y;
    else if (x == x.p.left)
        x.p.left = y;
    else
        x.p.right = y;
    y.left = x;
    x.p = y;
}
```

图 13.2 给出了一个 $LeftRotate$ 操作修改二叉搜索树的例子. $RightRotate$ 操作的代码是对称的. $LeftRotate$ 和 $RightRotate$ 都在 $O(1)$ 时间内运行完成. 在旋转操作中只有指针改变, 其他所有属性都保持不变.

程序 13.4: 右旋的实现

```
public void RightRotate(RBTreeNode x) {
    var y = x.left;
    x.left = y.right;
    if (y.right != nil)
        y.right.p = x;
    y.p = x.p;
    if (x.p == nil)
        root = y;
    else if (x == x.p.right)
        x.p.right = y;
```

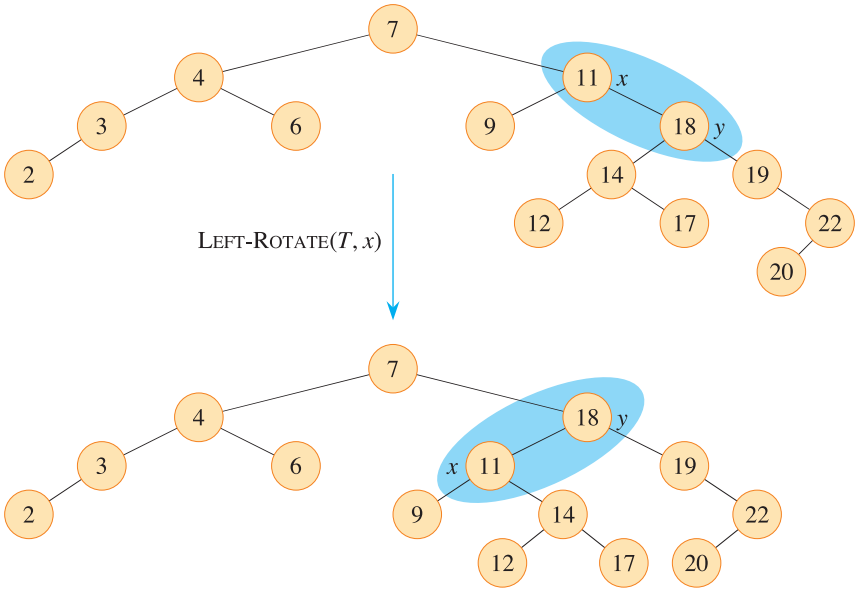


图 13.3: 左旋操作修改二叉搜索树的例子. 输入的树和修改过的树进行中序遍历, 产生相同的关键字值列表

```
    else
        x.p.left = y;
    y.right = x;
    x.p = y;
}
```

13.3　插入

我们可以在 $O(\lg n)$ 时间内完成向一棵含 n 个结点的红黑树中插入一个新结点. 为了做到这一点, 利用 $TreeInsert$ 过程 (参见 12.3 节) 的一个略作修改的版本来将结点 z 插入树 T 内, 就好像 T 是一棵普通的二叉搜索树一样, 然后将 z 着为红色. 为保证红黑性质能继续保持, 我们调用一个辅助程序 $RBInsertFixup$ 来对结点重新着色并旋转. 调用 $RBInsert(T, z)$ 在红黑树 T 内插入结点 z, 假设 z 的 key 属性已被事先赋值.

程序 13.5: 红黑树的插入程序

```
public void RBInsert(int key) {
    var z = new RBTreeNode();
    z.key = key;
    // 首先进行普通的二叉搜索树插入操作
    var x = root;
    var y = nil;
    while (x != nil) {
        y = x;
        if (z.key < x.key)
            x = x.left;
        else
            x = x.right;
    }

    // y 是 x 的父结点
    z.p = y;
    if (y == nil)
        root = z;
    else if (z.key < y.key)
```

```
20          y.left = z;
21      else
22          y.right = z;
23
24      // 初始化新结点的一些属性
25      z.left = nil;
26      z.right = nil;
27      z.color = RED; // 新结点必须是红色
28
29      // 修复红黑树
30      RBInsertFixup(z);
31  }
```

过程 $TreeInsert$ 和 $RBInsert$ 之间有 4 处不同. 第一, $TreeInsert$ 内的所有 $null$ 都被 $T.nil$ 代替. 第二, $RBInsert$ 的第 14-15 行置 $z.left$ 和 $z.right$ 为 $T.nil$, 以保持合理的树结构. 第三, 在第 16 行将 z 着为红色. 第四, 因为将 z 着为红色可能违反其中的一条红黑性质, 所以在 $RBInsert$ 的第 17 行中调用 $RBInsertFixup$(T, z) 来保持红黑性质.

程序 13.6: 红黑树插入的修复程序

```
1   public void RBInsertFixup(RBTreeNode z) {
2       RBTreeNode y;
3       while (z.p.color == RED) {
4           if (z.p == z.p.p.left) {
5               y = z.p.p.right;
6               if (y.color == RED) {
7                   z.p.color = BLACK;
8                   y.color = BLACK;
9                   z.p.p.color = RED;
10                  z = z.p.p;
11              } else {
12                  if (z == z.p.right) {
13                      z = z.p;
14                      LeftRotate(z);
15                  }
```

```
16                  z.p.color = BLACK;
17                  z.p.p.color = RED;
18                  RightRotate(z.p.p);
19              }
20          } else {
21              y = z.p.p.left;
22              if (y.color == RED) {
23                  z.p.color = BLACK;
24                  y.color = BLACK;
25                  z.p.p.color = RED;
26                  z = z.p.p;
27              } else {
28                  if (z == z.p.left) {
29                      z = z.p;
30                      RightRotate(z);
31                  }
32                  z.p.color = BLACK;
33                  z.p.p.color = RED;
34                  LeftRotate(z.p.p);
35              }
36          }
37      }
38      root.color = BLACK;
39  }
```

为了理解 $RBInsertFixup$ 过程如何工作, 把代码分为三个主要的步骤. 首先, 要确定当结点 z 被插入并着为红色后, 红黑性质中有哪些不能继续保持. 其次, 应分析第 1-15 行中 **while** 循环的总目标. 最后, 要分析 **while** 循环体中的三种情况, 看看它们是如何完成目标的. 图 13.4 给出一个范例, 显示在一棵红黑树上 $RBInsertFixup$ 如何操作.

在调用 $RBInsertFixup$ 操作时, 哪些红黑性质可能会被破坏呢? 性质 1 和性质 3 继续成立, 因为新插入的红结点的两个子结点都是哨兵 T.nil. 性质 5, 即从一个指定结点开始的每条简单路径上的黑结点的个数都是相等的, 也会成立, 因为结点 z 代替了 (黑色) 哨兵, 并且结点 z 本身是有哨兵孩子的红结点. 这样来看, 仅可能被破坏的就是性质 2 和性质 4, 即根结点需要为黑色以及一个红结点不能有红孩子. 这两个性质可能被破坏是因为 z 被着为红色. 如果 z 是根结

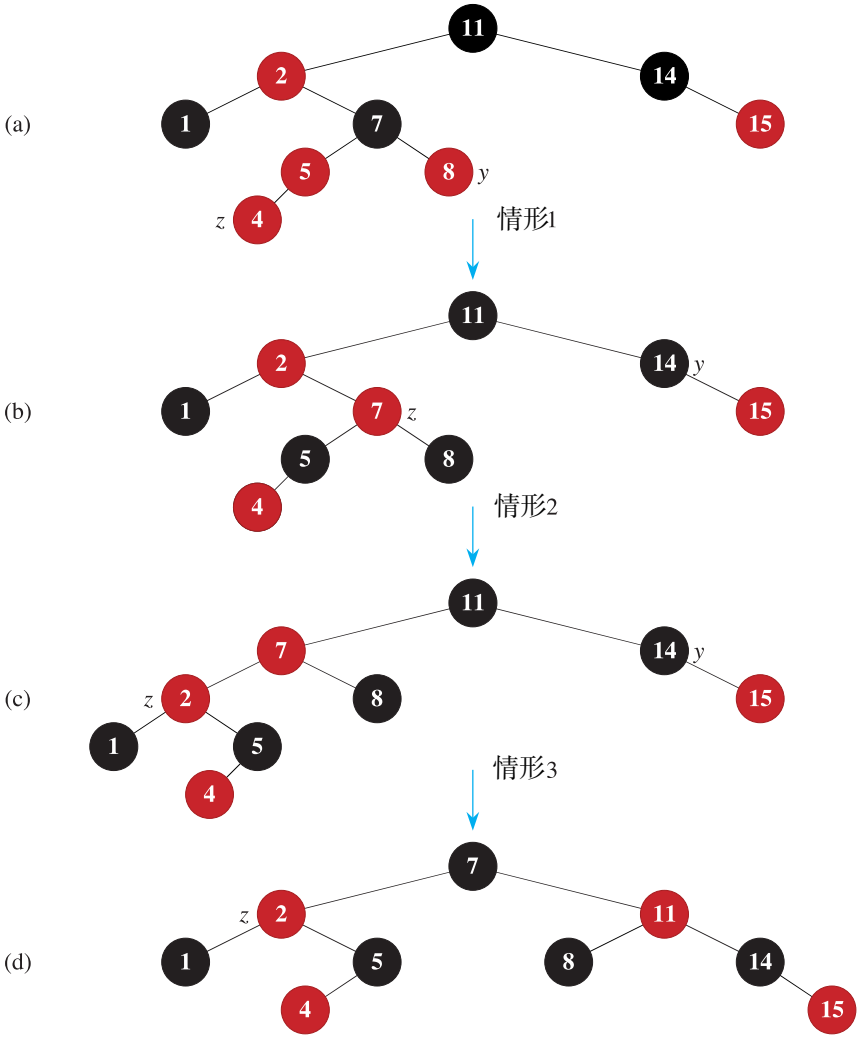


图 13.4: $RBInsertFixup$ 操作. (a) 插入后的结点 z. 由于 z 和它的父结点 $z.p$ 都是红色的, 所以违反了性质 4. 由于 z 的叔结点 y 是红色的, 可以应用程序中的情况 1. 结点被重新着色, 并且指针 z 沿树上升, 所得的树如 (b) 所示. 再一次 z 及其父结点又都为红色, 但 z 的叔结点 y 是黑色的. 因为 z 是 $z.p$ 的右孩子, 可以应用情况 2. 在执行 1 次左旋之后, 所得结果树见 (c). 现在, z 是其父结点的左孩子, 可以应用情况 3. 重新着色并执行一次右旋后得 (d) 中的树, 它是一棵合法的红黑树

点, 则破坏了性质 2; 如果 z 的父结点是红结点, 则破坏了性质 4. 图 13.4(a) 显示在插入结点 z 之后性质 4 被破坏的情况.

第 1-15 行中的 **while** 循环在每次迭代的开头保待下列 3 个部分的不变式:

1. 结点 z 是红结点.
2. 如果 $z.p$ 是根结点, 则 $z.p$ 是黑结点.
3. 如果有任何红黑性质被破坏, 则至多只有一条被破坏, 或是性质 2, 或是性质 4. 如果性质 2 被破坏, 其原因为 z 是根结点且是红结点. 如果性质 4 被破坏, 其原因为 z 和 $z.p$ 都是红结点.

第 3 部分处理红黑性质的破坏, 相比第 1 部分和第 2 部分来说, 显得更是 $RBInsertFixup$ 保持红黑性质的中心内容, 我们以此来理解代码中的各种情形. 由于将注意力集中在结点 z 以及树中靠近它的结点上, 所以有助于从第 1 部分得知 z 为红结点. 当在第 2, 3, 7, 8, 13 和 14 行中引用 $z.p.p$ 时, 我们使用 b 部分来表明它的存在.

需要证明在循环的第一次迭代之前循环不变量为真, 每次迭代都保持这个循环不变量成立, 并且在循环终止时, 这个循环不变量会给出一个有用的性质.

先从初始化和终止的不变式证明开始. 然后, 依据细致地考察循环体如何工作, 来证明循环在每次迭代中都保持这个循环不变量. 同时, 还要说明循环的每次迭代会有两种可能的结果: 或者指针 z 沿着树上移, 或者执行某些旋转后循环终止.

初始化: 在循环的第一次迭代之前, 从一棵正常的红黑树开始, 并新增一个红结点 z.

要证明当 $RBInsertFixup$ 被调用时, 不变式的每个部分都成立.

a. 当调用 $RBInsertFixup$ 时, z 是新增的红结点.

b. 如果 $z.p$ 是根, 那么 $z.p$ 开始是黑色的, 且在调用 $RBInsertFixup$ 之前保持不变.

c. 注意到在调用 $RBInsertFixup$ 时, 性质 1, 性质 3 和性质 5 成立.

如果违反了性质 2, 则红色根结点一定是新增结点 z, 它是树中唯一的内部结点. 因为 z 的父结点和两个子结点都是黑色的哨兵, 没有违反性质 4. 这样, 对性质 2 的违反是整棵树中唯一违反红黑性质的地方.

如果违反了性质 4, 则由于 z 的子结点是黑色哨兵, 且该树在 z 加入之前没有其他性质的违反, 所以违反必然是因为 z 和 $z.p$ 都是红色的. 而且, 没有其他红黑性质被违反.

终止: 循环终止是因为 z.p 是黑色的.(如果 z 是根结点, 那么 z.p 是黑色哨兵 T.nil.) 这样, 树在循环终止时没有违反性质 4. 根据循环不变量, 唯一可能不

成立的是性质 2. 第 16 行恢复这个性质, 所以当 $RBInsertFixup$ 终止时, 所有的红黑性质都成立.

保持: 实际需要考虑 while 循环中的 6 种情况, 而其中三种与另外三种是对称的. 这取决于第 2 行中 z 的父结点 z.p 是 z 的祖父结点 z.p.p 的左孩子, 还是右孩子. 我们只给出 z.p 是左孩子时的代码. 根据循环不变量的 b 部分, 如果 z.p 是根结点, 那么 z.p 是黑色的, 可知结点 z.p.p 存在. 因为只有在 z.p 是红色时才进入一次循环迭代, 所以 z.p 不可能是根结点. 因此, z.p.p 存在.

情况 1 和情况 2, 情况 3 的区别在于 z 父亲的兄弟结点 (或称为 “叔结点”) 的颜色不同. 第 3 行使 y 指向 z 的叔结点 z.p.p.right, 在第 4 行测试 y 的颜色. 如果 y 是红色的, 那么执行情况 1. 否则, 控制转向情况 2 和情况 3 上. 在所有三种情况中, z 的祖父结点 z.p.p 是黑色的, 因为它的父结点 z.p 是红色的, 故性质 4 只在 z 和 z.p 之间被破坏了.

情况 1: z 的叔结点 y 是红色的

图 13.3 显示了情况 1(第 5-8 行) 的情形, 这种情况在 z.p 和 y 都是红色时发生. 因为 z.p.p 是黑色的, 所以将 z.p 和 y 都着为黑色, 以此解决 z 和 z.p 都是红色的问题, 将 z.p.p 着为红色以保持性质 5. 然后, 把 z.p.p 作为新结点 z 来重复 while 循环. 指针 z 在树中上移两层.

现在, 证明情况 1 在下一次循环迭代的开头会保持这个循环不变量. 用 z 表示当前迭代中的结点 z, 用 z'=z.p.p 表示在下一次迭代第 1 行测试时的结点 z.

a. 因为这次迭代把 $z.p.p$ 着为红色, 结点 z' 在下次迭代的开始是红色的.

b. 在这次迭代中结点 $z'.p$ 是 $z.p.p.p$, 且这个结点的颜色不会改变. 如果它是根结点, 则在此次迭代之前它是黑色的, 且它在下次迭代的开头仍然是黑色的.

c. 我们已经证明情况 1 保待性质 5, 而且它也不会引起性质 1 或性质 3 的破坏.

如果结点 z' 在下一次迭代开始时是根结点, 则在这次迭代中悄况 1 修正了唯一被破坏的性质 4. 由于 z' 是红色的而且是根结点, 所以性质 2 成为唯一被违反的性质, 这是由 z' 导致的.

如果结点 z' 在下一次迭代开始时不是根结点, 则情况 1 不会导致性质 2 的破坏. 情况 1 修正了在这次迭代的开始唯一违反的性质 4. 然后它把 z' 着为红色而 z'.p 不变. 如果 z'.p 是黑色的, 则没有违反性质 4. 如果 z'.p 是红色的, 则把 z' 着为红色会在 z' 与 z'.p 之间造成性质 4 的违反.

情况 2: z 的叔结点 y 是黑色的且 z 是一个右孩子

情况 3: z 的叔结点 y 是黑色的且 z 是一个左孩子

在情况 2 和情况 3 中, z 的叔结点 y 是黑色的. 通过 z 是 z.p 的右孩子还是左孩子来区别这两种情况. 第 10 11 行构成了情况 2, 它和情况 3 一起显示在图

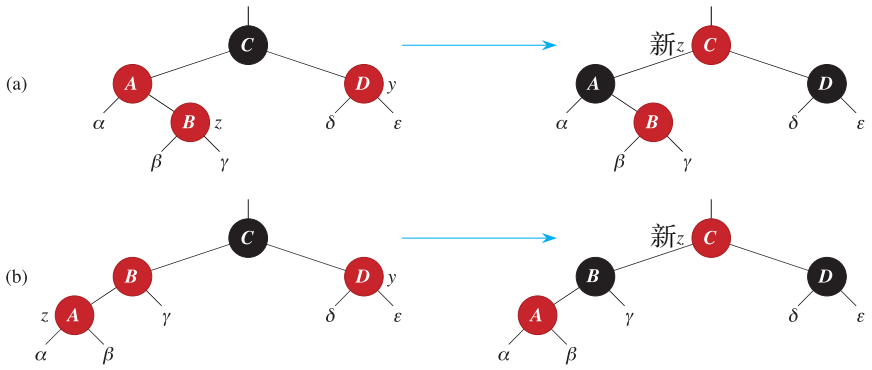


图 13.5: 过程 $RBInsertFixup$ 中的情况 1. 性质 4 被违反, 因为 z 和它的父结点 z.p 都是红色的. 无论 z 是一个右孩子 (图 (a)) 还是一个左孩子 (图 (b)), 都同样处理. 每一棵子树 $\alpha, \beta, \gamma, \theta$和$\epsilon$ 都有一个黑色根结点, 而且具有相同的黑高. 情况 1 的代码改变了某些结点的颜色, 但保持了性质 5: 从一个结点向下到一个叶结点的所有简单路径都有相同数目的黑结点. while 循环将结点 z 的祖父 z.p.p 作为新的 z 以继续迭代. 现在性质 4 的破坏只可能发生在新的红色结点 z 和它的父结点之间, 条件是如果父结点也为红色的

13.6 中. 在情况 2 中, 结点 z 是它的父结点的右孩子. 可以立即使用一个左旋来将此情形转变为情况 3(第 12-14 行), 此时结点 z 为左孩子. 因为 z 和 z.p 都是红色的, 所以该旋转对结点的黑高和性质 5 都无影响. 无论是直接进入情况 2, 还是通过情况 3 进入情况 2, z 的叔结点 y 总是黑色的, 因为否则就要执行情况 1. 此外, 结点 z.p.p 存在, 因为已经推断在执行第 2 行和第 3 行时该结点存在, 且在第 10 行将 z 往上移一层, 然后在第 11 行将 z 往下移一层之后, z.p.p 的身份保持不变. 在情况 3 中, 改变某些结点的颜色并做一次右旋, 以保待性质 5. 这样, 由于在一行中不再有两个红色结点, 所有的处理到此完毕. 因为此时 z.p 是黑色的, 所以无需再执行一次 while 循环.

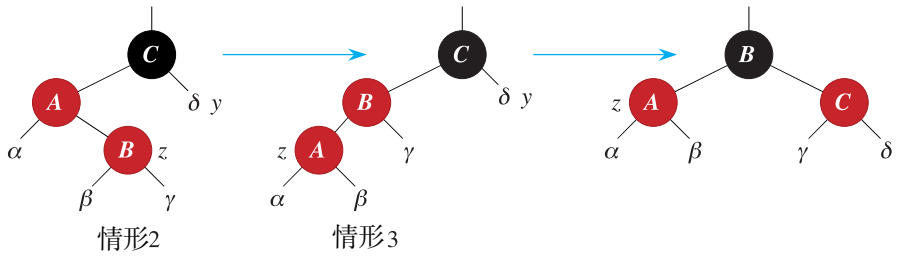


图 13.6: 过程 $RBInsertFixup$ 中的情况 2 和情况 3. 如同情况 1, 由于 z 和它的父结点 z.p 都是红色的, 性质 4 在情况 2 或情况 3 中会被破坏. 每一棵子树 α, β, γ 和 δ 都有一个黑色根结点 (α, β 和 γ 是由性质 4 而来, δ 也有黑色根结点, 因为否则将导致情况 1), 而且具有相同的黑高. 通过左旋将情况 2 转变为情况 3, 以保待性质 5, 从一个结点向下到一个叶结点的所有简单路径都有相同数目的黑结点. 情况 3 引起某些结点颜色的改变, 以及一个同样为了保待性质 5 的右旋. 然后 while 循环终止, 因为性质 4 已经得到了满足: 一行中不再有两个红色结点

现在来证明情况 2 和情况 3 保持了循环不变量. (正如已经讨论的, z.p 在第 1 行中下一次测试会是黑色, 循环体不会再次执行.)

a. 情况 2 让 z 指向红色的 z.p. 在情况 2 和情况 3 中 z 或 z 的颜色都不再改变.

b. 情况 3 把 z.p 着成黑色, 使得如果 z.p 在下一次迭代开始时是根结点, 则它是黑色的.

c. 如同情况 1, 性质 1 、性质 3 和性质 5 在情况 2 与情况 3 中得以保持.

由于结点 z 在情况 2 和情况 3 中都不是根结点, 所以性质 2 没有被破坏. 情况 2 和情况 3 不会引起性质 2 的违反, 因为唯一着为红色的结点在情况 3 中通过旋转成为一个黑色结点的子结点.

情况 2 和情况 3 修正了对性质 4 的违反, 也不会引起对其他红黑性质的违反.

证明了循环的每一次迭代都会保持循环不变量之后, 也就证明了 $RBInsert-Fixup$ 能够正确地保持红黑性质.

分析

$RBInsert$ 的运行时间怎样呢? 由于一棵有 n 个结点的红黑树的高度为 $O(\lg n)$, 因此 $RBInsert$ 的第 1 16 行要花费 $O(\lg n)$ 时间. 在 $RBInsertFixup$ 中, 仅当情况 1 发生, 然后指针 z 沿着树上升 2 层, while 循环才会重复执行. 所以 while 循环可能被执行的总次数为 $O(\lg n)$. 因此, $RBInsert$ 总共花费 $O(\lg n)$ 时间. 此外, 该程序所做的旋转从不超过 2 次, 因为只要执行了情况 2 或情况 3, while 循环就结束了.

13.4　删除

与 n 个结点的红黑树上的其他基本操作一样, 删除一个结点要花费 $O(\lg n)$ 时间. 与插入操作相比, 删除操作要稍微复杂些.

从一棵红黑树中删除结点的过程是基于 $TreeDelete$ 过程而来的. 首先, 需要特别设计一个供 $TreeDelete$ 调用的子过程 $Transplant$, 并将其应用到红黑树上:

程序 13.7:

```
1  public void RBTransplant(RBTreeNode u, RBTreeNode v) {
2      if (u.p == nil)
3          root = v;
4      else if (u == u.p.left)
5          u.p.left = v;
6      else
7          u.p.right = v;
8      v.p = u.p;
9  }
10
11 public RBTreeNode TreeMinimum(RBTreeNode x) {
12     while (x.left != nil) {
13         x = x.left;
14     }
```

```
15      return x;
16  }
```

过程 $RBTransplant$ 与 $Transplant$ 有两点不同. 首先, 第 1 行引用哨兵 $T.nil$ 而不是 NIL. 其次, 第 6 行对 $v.p$ 的赋值是无条件执行: 即使 v 指向哨兵, 也要对 $v.p$ 赋值. 实际上, 当 $v = T.nil$ 时, 也能给 $v.p$ 赋值.

过程 $RBDelete$ 与 $TreeDelete$ 类似, 只是多了几行代码. 多出的几行代码记录结点 y 的踪迹, y 有可能导致红黑性质的破坏. 当想要删除结点 z, 且此时 z 的子结点少于 2 个时, z 从树中删除, 并让 y 成为 z. 当 z 有两个子结点时, y 应该是 z 的后继, 并且 y 将移至树中的 z 位置. 在结点被移除或者在树中移动之前, 必须记住 y 的颜色, 并且记录结点 x 的踪迹, 将 x 移至树中 y 的原来位置, 因为结点 x 也可能引起红黑性质的破坏. 删除结点 z 之后, $RBDelete$ 调用一个辅助过程 $RBDeleteFixup$, 该过程通过改变颜色和执行旋转来恢复红黑性质.

程序 13.8: 红黑树的结点删除程序

```
1  public void RBDelete(int key) {
2      RBDeleteHelper(root, key);
3  }
4
5  public void RBDeleteHelper(RBTreeNode node, int key) {
6      RBTreeNode z = nil;
7      while (node != nil) {
8          if (node.key == key) z = node;
9          if (node.key <= key) node = node.right;
10         else node = node.left;
11     }
12     var y = z;
13     var yOriginalColor = y.color;
14
15     RBTreeNode x;
16     if (z.left == nil) {
17         x = z.right;
18         RBTransplant(z, z.right);
19     } else if (z.right == nil) {
20         x = z.left;
```

```
21          RBTransplant(z, z.left);
22      } else {
23          y = TreeMinimum(z.right);
24          yOriginalColor = y.color;
25          x = y.right;
26          if (y != z.right) {
27              RBTransplant(y, y.right);
28              y.right = z.right;
29              y.right.p = y;
30          } else {
31              x.p = y;
32          }
33          RBTransplant(z, y);
34          y.left = z.left;
35          y.left.p = y;
36          y.color = z.color;
37      }
38      if (yOriginalColor == BLACK)
39          RBDeleteFixup(x);
40  }
```

虽然 *RBDelete* 包含的伪代码行数几乎是 *TreeDelete* 的 2 倍, 但这两个过程具有相同的基本结构. 在 *RBDelete* 中能够找到 *TreeDelete* 的每一行语句 (其中 NIL 被替换成了 $T.nil$, 而调用 *Transplant* 换成了调用 *RBTransplant*), 其执行的条件相同.

下面是两个过程之间的其他区别:

- 始终维持结点 y 为从树中删除的结点或者移至树内的结点. 当 z 的子结点少于 2 个时, 第 1 行将 y 指向 z, 并因此要移除. 当 z 有两个子结点时, 第 9 行将 y 指向 z 的后继, 这与 *TreeDelete* 相同, y 将移至树中 z 的位置.

- 由于结点 y 的颜色可能改变, 变量 y-original-color 存储了发生改变前的 y 颜色. 第 2 行和第 10 行在给 y 赋值之后, 立即设置该变量. 当 z 有两个子结点时, 则 y != z 且结点 y 移至红黑树中结点之的原始位置; 第 20 行给 y 赋予和 z 一样的颜色. 我们需要保存 y 的原始颜色, 以在 *RBDelete* 结束时测试它; 如果它是黑色的, 那么删除或移动 y 会引起红黑性质的破坏.

- 正如前面讨论过的, 我们保存结点 x 的踪迹, 使它移至结点 y 的原始位置

上. 第 4, 7 和 11 行的赋值语句令 x 或指向 y 的唯一子结点或指向哨兵 T.nil(如果 y 没有子结点).(回忆一下 12. 3 节 y 没有左孩子的情形.)

- 因为结点 x 移动到结点 y 的原始位置, 属性 x.p 总是被设置指向树中 y 父结点的原始位置, 甚至当 x 是哨兵 T.nil 时也是这样. 除非 z 是 y 的原始父结点 (该情况只在之有两个孩子且它的后继 y 是 z 的右孩子时发生), 否则对 x.p 的赋值在 RB-TRANSPLANT 的第 6 行.(注意到, 在第 5, 8 或 14 行调用 RB-TRANSPLANT 时, 传递的第 2 个参数与 x 相同.) 然而, 当 y 的原父结点是 z 时, 我们并不想让 x.p 指向 y 的原始父结点, 因为要在树中删除该结点. 由于结点 y 将在树中向上移动占据 z 的位置, 第 13 行将 x.p 设置为 y, 使得 x.p 指向 y 父结点的原始位置, 甚至当 x=T.nil 时也是这样.

- 最后, 如果结点 y 是黑色, 就有可能已经引入了一个或多个红黑性质被破坏的情况, 所以在第 22 行调用 $RBDeleteFixup$ 来恢复红黑性质. 如果 y 是红色, 当 y 被删除或移动时, 红黑性质仍然保持, 原因如下:

  1. 树中的黑高没有变化.
  2. 不存在两个相邻的红结点. 因为 y 在树中占据了 z 的位置, 再考虑到 z 的颜色, 树中 y 的新位置不可能有两个相邻的红结点. 另外, 如果 y 不是 z 的右孩子, 则 y 的原右孩子 x 代替 y. 如果 y 是红色, 则 x 一定是黑色, 因此用 x 替代 y 不可能使两个红结点相邻.
  3. 如果 y 是红色, 就不可能是根结点, 所以根结点仍旧是黑色.

如果结点 y 是黑色的, 则会产生三个问题, 可以通过调用 $RBDeleteFixup$ 进行补救. 第一, 如果 y 是原来的根结点, 而 y 的一个红色的孩子成为新的根结点, 这就违反了性质 2. 第二, 如果 x 和 $x.p$ 是红色的, 则违反了性质 4. 第三, 在树中移动 y 将导致先前包含 y 的任何简单路径上黑结点个数少 1. 因此, y 的任何祖先都不满足性质 5. 改正这一问题的办法是将现在占有 y 原来位置的结点 x 视为还有一重额外的黑色. 也就是说, 如果将任意包含结点 x 的简单路径上黑结点个数加 1, 则在这种假设下, 性质 5 成立. 当将黑结点 y 删除或移动时, 将其黑色 “下推” 给结点 x. 现在问题变为结点 x 可能既不是红色, 又不是黑色, 从而违反了性质 1. 现在的结点 x 是双重黑色或者红黑色, 这就分别给包含 x 的简单路径上黑结点数贡献了 2 或 1. x 的 $color$ 属性仍然是 RED(如果 x 是红黑色的) 或者 $BLACK$(如果 x 是双重黑色的). 换句话说, 结点额外的黑色是针对 x 结点的, 而不是反映在它的 $color$ 属性上的.

现在我们来看看过程 $RBDeleteFixup$ 是如何恢复搜索树的红黑性质的.

程序 13.9: 红黑树删除结点后的修复程序

```
public void RBDeleteFixup(RBTreeNode x) {
    RBTreeNode w;
    while (x != root && x.color == BLACK) {
        if (x == x.p.left) {
            w = x.p.right;
            if (w.color == RED) {
                w.color = BLACK;
                x.p.color = RED;
                LeftRotate(x.p);
                w = x.p.right;
            }
            if (w.left.color == BLACK && w.right.color == BLACK)
            ↪ {
                w.color = RED;
                x = x.p;
            } else {
                if (w.right.color == BLACK) {
                    w.left.color = BLACK;
                    w.color = RED;
                    RightRotate(w);
                    w = x.p.right;
                }
                w.color = x.p.color;
                x.p.color = BLACK;
                w.right.color = BLACK;
                LeftRotate(x.p);
                x = root;
            }
        } else {
            w = x.p.left;
            if (w.color == RED) {
                w.color = BLACK;
                x.p.color = RED;
                RightRotate(x.p);
                w = x.p.left;
```

```
35              }
36              if (w.right.color == BLACK && w.left.color == BLACK)
                ↪  {
37                  w.color = RED;
38                  x = x.p;
39              } else {
40                  if (w.left.color == BLACK) {
41                      w.right.color = BLACK;
42                      w.color = RED;
43                      LeftRotate(w);
44                      w = x.p.left;
45                  }
46                  w.color = x.p.color;
47                  x.p.color = BLACK;
48                  w.left.color = BLACK;
49                  RightRotate(x.p);
50                  x = root;
51              }
52          }
53      }
54      x.color = BLACK;
55  }
```

过程 *RBDeleteFixup* 恢复性质 1, 性质 2 和性质 4. 练习 13.4-1 和 13.4-2 要求读者说明这个过程是如何恢复性性质 2 和性质 4 的, 因此, 本节的其余部分将专注千性质 1. 第 1-22 行中 **while** 循环的目标是将额外的黑色沿树上移, 直到:

- x 指向红黑结点, 此时在第 23 行中, 将 x 着为 (单个) 黑色.
- x 指向根结点, 此时可以简单地 “移除” 额外的黑色.
- 执行适当的旋转和重新着色, 退出循环.

在 while 循环中, x 总是指向一个具有双重黑色的非根结点. 在第 2 行中要判断 x 是其父结点 x.p 的左孩子还是右孩子.(已经给出了 x 为左孩子时的代码; x 为右孩子的第 22 行的代码是对称的.) 保持指针 w 指向 x 的兄弟. 由于结点 x 是双重黑色的, 故 w 不可能是 T.nil, 因为否则, 从 x.p 至 (单黑色) 叶子 w 的简单路径上的黑结点个数就会小于从 x.p 到 x 的简单路径上的黑结点数.

图 13-7 给出了代码中的 4 种情况. 在具体研究每一种情况之前, 先看看如何证实每种情况中的变换保待性质 5. 关键思想是在每种情况中, 从子树的根 (包括根) 到每棵子树 $\alpha, \beta, \cdots, \gamma$ 之间的黑结点个数 (包括 x 的额外黑色) 并不被变换改变. 因此, 如果性质 5 在变换之前成立, 那么变换之后也仍然成立. 举例说明, 图 13-7(a) 说明了情况 1, 在变换前后, 根结点至子树 α 或 β 之间的黑结点数都是 3.(再次记住, 结点 x 增加了额外一重黑色.) 类似地, 在变换前后根结点至子树 γ, δ, ϵ和ζ 中的任何一个之间的黑结点数都是 2. 在图 13-7(b) 中, 计数时还要包括所示子树的根结点的 *color* 属性的值 c, 它或是 RED 或是 $BLACK$. 如果定义 $count(RED) = O$ 以及 $count(BLACK) = 1$, 那么变换前后根结点至 α 的黑结点数都为 $2 + count(c)$. 在此情况下, 变换之后新结点 x 具有 *color* 属性值 c, 但是这个结点的颜色是红黑 (如果 $c = RED$) 或者双重黑色的 (如果 $c = BLACK$). 其他情况可以类似地加以验证.

情况 1: x 的兄弟结点 w 是红色的

情况 1(见 $RBDeleteFixup$ 的第 5-8 行和图 13-7(a)) 发生在结点 x 的兄弟结点 w 为红色时. 因为 w 必须有黑色子结点, 所以可以改变 w 和 x.p 的颜色, 然后对 x.p 做一次左旋而不违反红黑树的任何性质. 现在, x 的新兄弟结点是旋转之前 w 的某个子结点, 其颜色为黑色. 这样, 就将情况 1 转换为情况 2, 3 或 4 处理.

当结点 w 为黑色时, 属于情况 2, 3 和 4; 这些情况是由 w 的子结点的颜色来区分的.

情况 2: x 的兄弟结点 w 是黑色的, 而且 w 的两个子结点都是黑色的

在情况 2(见 $RBDeleteFixup$ 的第 10-11 行和图 13-7(6)) 中, w 的两个子结点都是黑色的. 因为 w 也是黑色的, 所以从 x 和 w 上去掉一重黑色, 使得 x 只有一重黑色而 w 为红色. 为了补偿从 x 和 w 中去掉的一重黑色, 在原来是红色或黑色的 x.p 上新增一重额外的黑色. 通过将 x.p 作为新结点 x 来重复 while 循环. 注意到, 如果通过情况 1 进入到情况 2, 则新结点 x 是红黑色的, 因为原来的 x.p 是红色的. 因此, 新结点 x 的 color 属性值 c 为 RED, 并且在测试循环条件后循环终止. 然后, 在第 23 行中将新结点 x 着为 (单一) 黑色.

情况 3: X 的兄弟结点 w 是黑色的, w 的左孩子是红色的, w 的右孩子是黑色的

情况 3(见第 13-16 行和图 13-7(c)) 发生在 w 为黑色且其左孩子为红色, 右孩子为黑色时. 可以交换 w 和其左孩子 w.left 的颜色, 然后对 w 进行右旋而不违反红黑树的任何性质. 现在 x 的新兄弟结点 w 是一个有红色右孩子的黑色结点, 这样我们就将情况 3 转换成了情况 4.

情况 4: X 的兄弟结点 w 是黑色的, 且 w 的右孩子是红色的

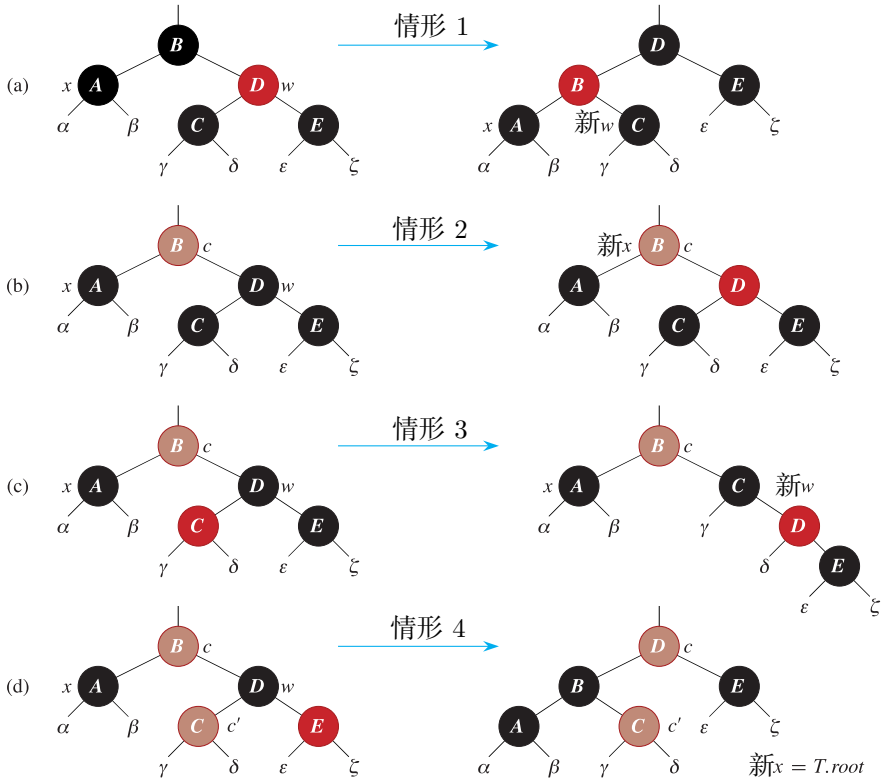


图 13.7: 过程 *RBDeleteFixup* 中 while 循环的各种情况. 加黑的结点 color 属性为 BLACK, 深阴影的结点 color 属性为 RED, 浅阴影的结点 color 属性用 c 和 c' 表示, 它既可为 RED 也可为 BLACK. 字母 $\alpha, \beta, \cdots, \zeta$ 代表任意的子树. 在每种情况中, 通过改变某些结点的颜色及/或进行一次旋转, 可以将左边的结构转化为右边的结构. x 指向的任何结点都具有额外的一重黑色而成为双重黑色或红黑色. 只有情况 2 引起循环重复. (a) 通过交换结点 B 和 D 的颜色以及执行一次左旋, 可将情况 1 转化为情况 2, 3 或 4. (b) 在情况 2 中, 在将结点 D 着为红色, 并将 x 设为指向结点 B 后, 由指针 x 所表示的额外黑色沿树上升. 如果通过情况 1 进入情况 2, 则 while 循环结束, 因为新的结点 x 是红黑的, 因此其 color 属性 c 是 RED. (c) 通过交换结点 C 和 D 的颜色并执行一次右旋, 可以将情况 3 转换成情况 4. (d) 在情况 4 中, 通过改变某些结点的颜色并执行一次左旋 (不违反红黑性质), 可以将由 x 表示的额外黑色去掉, 然后循环终止

情况 4(见第 17-21 行和图 13-7(d)) 发生在结点 x 的兄弟结点 w 为黑色且 w 的右孩子为红色时. 通过进行某些颜色修改并对 x.p 做一次左旋, 可以去掉 x 的额外黑色, 从而使它变为单重黑色, 而且不破坏红黑树的任何性质. 将 x 设置为根后, 当 while 循环测试其循环条件时, 循环终止.

分析

$RBDelete$ 的运行时间怎样呢? 因为含 n 个结点的红黑树的高度为 $O(\lg n)$, 不调用 $RBDeleteFixup$ 时该过程的总时间代价为 $O(\lg n)$. 在 $RBDeleteFixup$ 中, 情况 1, 3 和 4 在各执行常数次数的颜色改变和至多 3 次旋转后便终止. 情况 2 是 while 循环可以重复执行的唯一情况, 然后指针 x 沿树上升至多 $O(\lg n)$ 次, 且不执行任何旋转. 所以, 过程 $RBDeleteFixup$ 要花费 $O(\lg n)$ 时间, 做至多 3 次旋转, 因此 $RBDelete$ 运行的总时间为 $O(\lg n)$.

13.5　红黑树的测试和漂亮打印

程序 13.10: 打印红黑树的结构的代码

```
1  public void printHelper(RBTreeNode node, String indent, boolean
   ↪  last) {
2      if (node != nil) {
3          System.out.print(indent);
4          if (last) {
5              System.out.print("R----");
6              indent += "     ";
7          } else {
8              System.out.print("L----");
9              indent += "|    ";
10         }
11
12         var color = node.color == RED ? "RED" : "BLACK";
13         System.out.println(node.key + "(" + color + ")");
14         printHelper(node.left, indent, false);
15         printHelper(node.right, indent, true);
16     }
17 }
18
19 public void prettyPrint() {
```

```
20      printHelper(root, "", true);
21  }
```

红黑树的测试程序:

程序 13.11: 红黑树的测试程序

```
1   public class RBTreeTest {
2       public static void main(String[] args) {
3           var t = new RBTree();
4           t.RBInsert(18);
5           t.RBInsert(5);
6           t.RBInsert(8);
7           t.RBInsert(15);
8           t.RBInsert(17);
9           t.RBInsert(25);
10          t.RBInsert(40);
11          t.RBInsert(80);
12          t.RBDelete(25);
13          t.prettyPrint();
14      }
15  }
```

第十四章　B 树

B 树是为磁盘或其他直接存取的辅助存储设备而设计的一种平衡搜索树. B 树类似于红黑树, 但它们在降低磁盘 I/O 操作数方面要更好一些. 许多数据库系统使用 B 树或者 B 树的变种 (如 B+ 树) 来存储信息.

B 树与红黑树的不同之处在于 B 树的结点可以有很多孩子, 从数个到数千个. 也就是说, 一个 B 树的 “分支因子” 可以相当大, 尽管它通常依赖于所使用的磁盘单元的特性. B 树类似于红黑树, 就是每棵含有 n 个结点的 B 树的高度为 $O(\lg n)$. 然而, 一棵 B 树的严格高度可能比一棵红黑树的高度要小许多, 这是因为它的分支因子, 也就是表示高度的对数的底数可以非常大. 因此, 我们也可以使用 B 树在时间 $O(\lg n)$ 内完成一些动态集合的操作.

B 树以一种自然的方式推广了二叉搜索树. 图14.1给出了一棵简单的 B 树. 如果 B 树的一个内部结点 x 包含 $x.n$ 个关键字, 那么结点 x 就有 $x.n+1$ 个孩子. 结点 x 中的关键字就是分隔点, 它把结点 x 中所处理的关键字的属性分隔为 $x.n+1$ 个子域, 每个子域都由 x 的一个孩子处理. 当在一棵 B 树中查找一个关键字时, 基于对存储在 x 中的 $x.n$ 个关键字的比较, 做出一个 $(x.n+1)$ 路的选择. 叶结点的结构与内部结点的结构不同, 14.1节将讨论这些差别.

14.1节给出 B 树的精确定义, 并证明了 B 树的高度仅随它所包含的结点数按对数增长. 14.2节介绍如何在 B 树中查找和插入一个关键字, 14.3节讨论删除操作. 然而, 在开始之前, 需要弄清楚为什么针对磁盘设计的数据结构不同于针对随机访问的主存所设计的数据结构.

辅存上的数据结构

计算机系统利用各种技术来提供存储能力. 一个计算机系统的主存 (primary memory 或 main memory) 通常由硅存储芯片组成. 这种技术每位的存储代价一般要比磁存储技术 (如磁带或磁盘) 高不止一个数量级. 许多计算机系统还有基于磁盘的辅存 (secondary storage); 这种辅存的容量通常要比主存的容量高出至少两个数量级.

图14.2是一个典型的磁盘驱动器. 驱动器由一个或多个盘片 (platter) 组成,

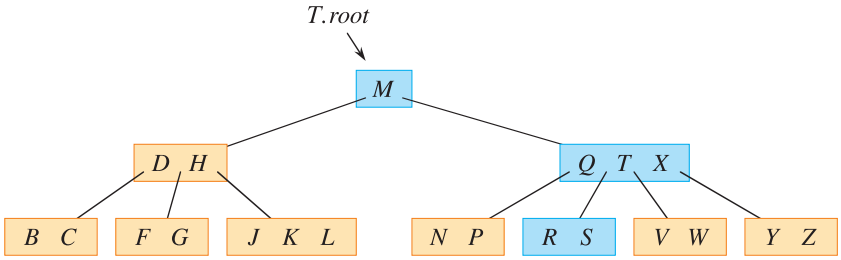


图 14.1: 一棵关键字为英语中辅音字母的 B 树. 一个内部结点 x 包含 $x.n$ 个关键字以及 $x.n+1$ 个孩子, 所有叶结点处于树中相同的深度. 浅阴影的结点是在查找字母 R 时检查过的结点

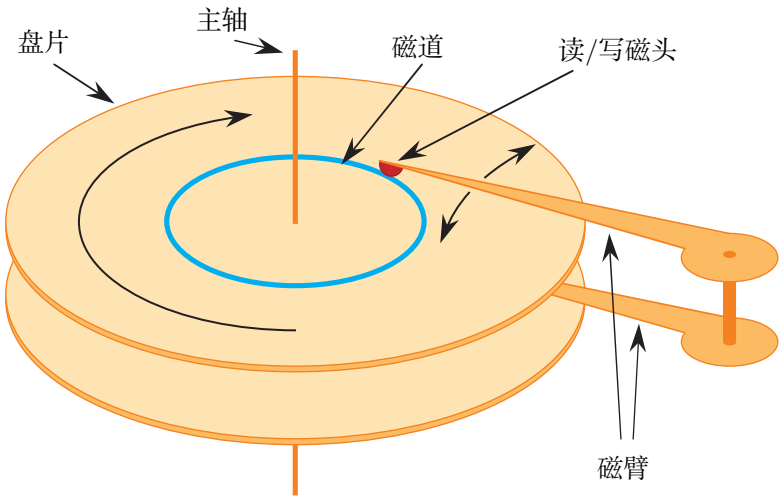


图 14.2: 一个典型的磁盘驱动器. 它包括了一个或多个绕主轴旋转的盘片 (这里画出的是两个). 每个盘片通过磁臂末端的磁头来读写. 这些磁臂围绕着一个共同的旋转轴旋转. 当读/写磁头静止时, 由它下方经过的磁盘表面就是一个磁道

它们以一个固定的速度绕着一个共同的主轴 (spindle) 旋转. 每个盘的表面覆盖着一层可磁化的物质. 驱动器通过磁臂 (arm) 末尾的磁头 (head) 来读/写盘片. 磁臂可以将磁头向主轴移近或移远. 当一个给定的磁头处于静止时, 它下面经过的磁盘表面称为一个磁道 (track). 多个盘片增加的仅仅是磁盘驱动器的容量, 而不影响性能.

尽管磁盘比主存便宜并且具有更多的容量, 但是它们比主存慢很多, 因为它们有机械运动的部分. 磁盘有两个机械运动的部分: 盘片旋转和磁臂移动. 在撰写本书时, 商用磁盘的旋转速度是 5400-15000 转/分钟 (RPM). 通常看到的 15000RPM 的速度是用于服务器级的驱动器上, 7200RPM 的速度用于台式机的驱动器上, 5400RPM 的速度用于笔记本的驱动器上. 尽管 7200RPM 看上去很快, 但是旋转一圈需要 8.33ms, 比硅存储的常见存取时间 50ns 要高出 5 个数量级. 也就是说, 如果不得不等待一个磁盘旋转完整的一圈, 让一个特定的项到达读/写磁头下方, 在这个时间内, 我们可能存取主存超过 100000 次. 平均来讲, 只需等待半圈, 但硅存储存取时间和磁盘存储存取时间的差距仍然是巨大的. 移动磁臂也要耗费时间. 在撰写本书时, 商用磁盘的平均存取时间在 8-11ms 范围内.

为了摊还机械移动所花费的等待时间, 磁盘会一次存取多个数据项而不是一个. 信息被分为一系列相等大小的在柱面内连续出现的位页面 (page), 并且每个磁盘读或写一个或多个完整的页面. 对于一个典型的磁盘来说, 一页的长度可能为 $2^{11} \sim 2^{14}$ 字节. 一旦读/写磁头正确定位, 并且盘片已经旋转到所要页面的开头位置, 对磁盘的读或写就完全电子化了 (除了磁盘的旋转外), 磁盘能够快速地读或写大量的数据.

通常, 定位到一页信息并将其从磁盘里读出的时间要比对读出信息进行检查的时间要长得多. 因此, 本章将对运行时间的两个主要组成成分分别加以考虑:

- 磁盘存取次数.
- CPU(计算) 时间.

我们使用需要读出或写入磁盘的信息的页数来衡量磁盘存取次数. 注意到, 磁盘存取时间并不是常量–它依赖于当前磁道和所需磁道之间的距离以及磁盘的初始旋转状态. 但是, 我们仍然使用读或写的页数作为磁盘存取总时间的主要近似值.

在一个典型的 B 树应用中, 所要处理的数据量非常大, 以至于所有数据无法一次装入主存. B 树算法将所需页面从磁盘复制到主存, 然后将修改过的页面写回磁盘. 在任何时刻, B 树算法都只需在主存中保持一定数量的页面. 因此, 主存的大小并不限制被处理的 B 树的大小.

用以下的伪代码来对磁盘操作进行建模. 设 x 为指向一个对象的指针. 如果该对象正在主存中, 那么可以像平常一样引用该对象的各个属性: 如 $x.key$. 然而, 如果 x 所指向的对象驻留在磁盘上, 那么在引用它的属性之前, 必须先执行 DISK-READ(x), 将该对象读入主存中. (假设如果 x 已经在主存中, 那么 DISK-READ(x) 不需要磁盘存取, 即它是个空操作.) 类似地, 操作 DISK-WRITE(x) 用来保存对象 x 的属性所做的任何修改. 也就是说, 一个对象操作的典型模式如下:

在任何时刻, 这个系统可以在主存中只保持有限的页数. 假定系统不再将被使用的页从主存中换出; 后面的 B 树算法会忽略这一点.

由于在大多数系统中, 一个 B 树算法的运行时间主要由它所执行的 DISK-READ 和 DISK-WRITE 操作的次数决定, 所以我们希望这些操作能够读或写尽可能多的信息. 因此, 一个 B 树结点通常和一个完整磁盘页一样大, 并且磁盘页的大小限制了一个 B 树结点可以含有的孩子个数.

对存储在磁盘上的一棵大的 B 树, 通常看到分支因子在 50 ~ 2000 之间, 具体取决于一个关键字相对于一页的大小. 一个大的分支因子可以大大地降低树的高度以及查找任何一个关键字所需的磁盘存取次数. 图14.3显示的是一棵分支因子为 1001, 高度为 2 的 B 树, 它可以存储超过 10 亿个关键字. 不过, 由于根结点可以持久地保存在主存中, 所以在这棵树中查找某个关键字至多只需两次磁盘存取.

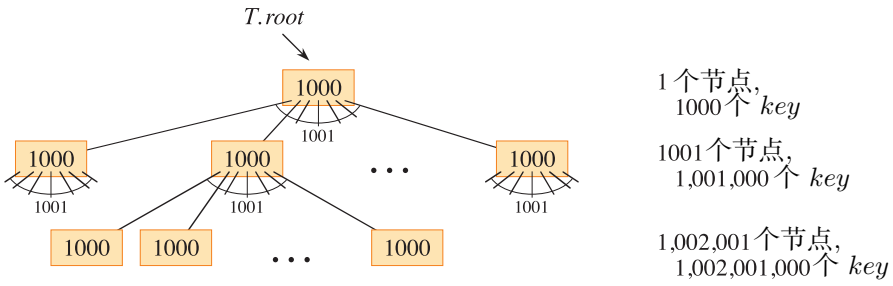
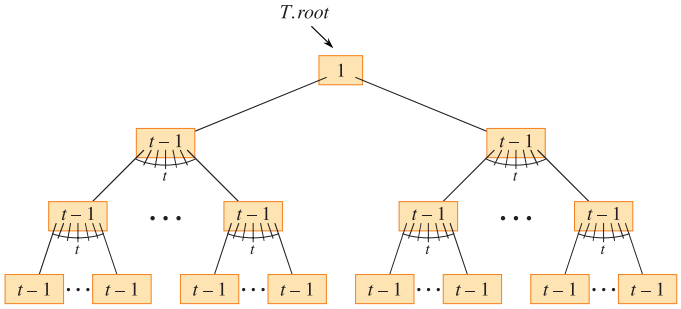


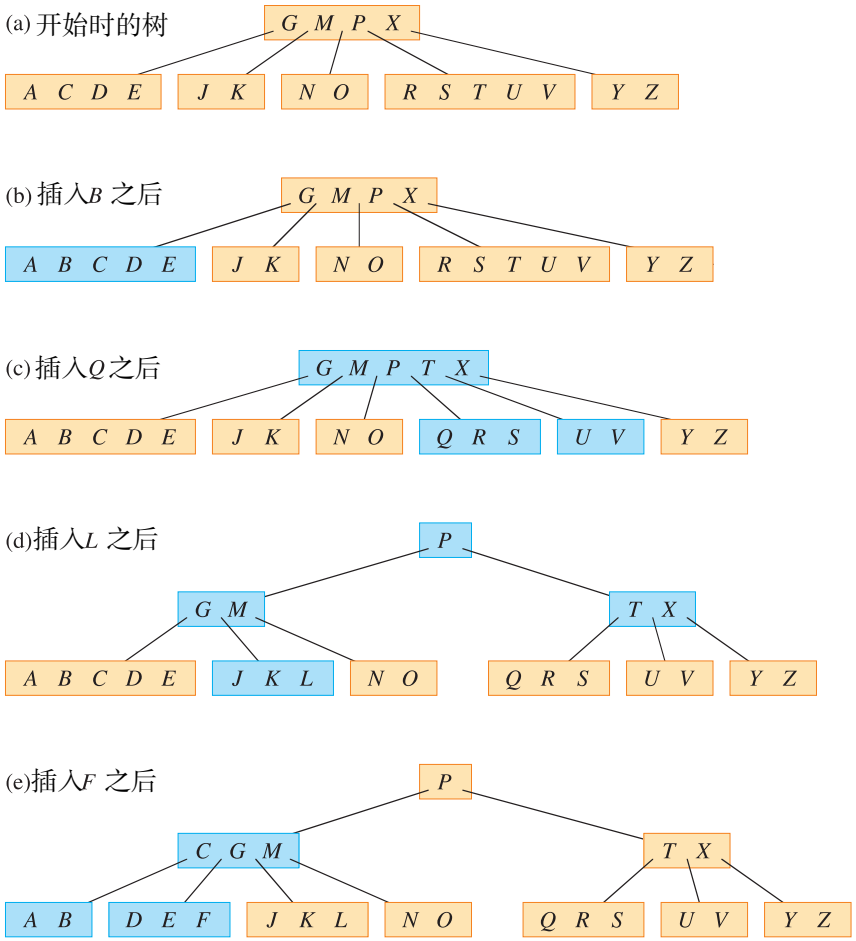
图 14.3: 一棵高度为 2 的 B 树包含 10 亿多个关键字. 显示在每个结点 x 内的是 $x.n$, 表示 x 中关键字个数. 每个内部结点及叶结点包含 1000 个关键字. 这棵 B 树在深度 1 上有 1001 个结点, 在深度 2 上有超过 100 万个叶结点

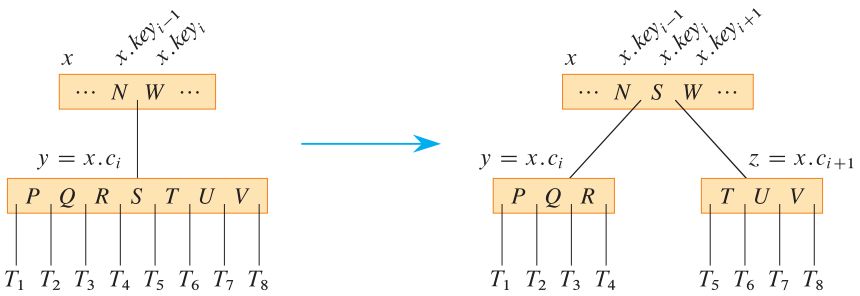
14.1　B 树的定义

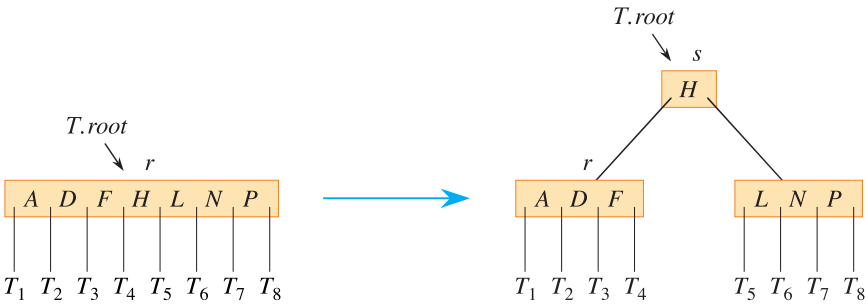


14.2　B 树的基本操作

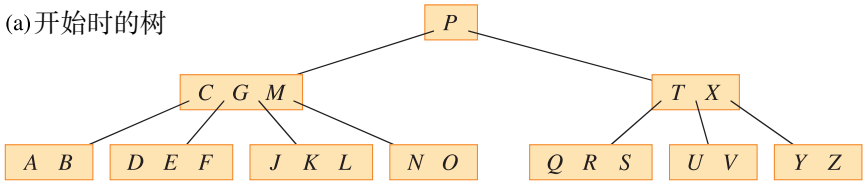
14.3　从 B 树中删除关键字


(a) 开始时的树
G M P X
A C D E
J K
N O
R S T U V
Y Z
(b) 插入B 之后
G M P X
A B C D E
J K
N O
R S T U V
Y Z
(c) 插入Q之后
G M P T X
A B C D E
J K
N O
Q R S
U V
Y Z
(d)插入L 之后
P
G M
T X
A B C D E
J K L
N O
Q R S
U V
Y Z
(e)插入F 之后
P
C G M
T X
A B
D E F
J K L
N O
Q R S
U V
Y Z

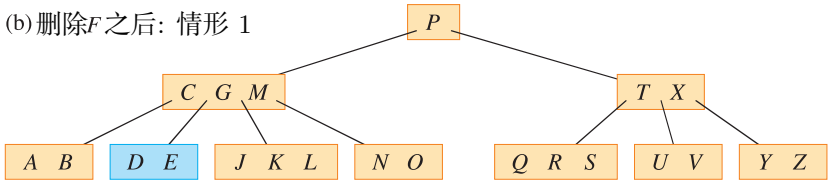

x
x.key_{i-1}
x.key_i
··· N W ···
y = x.c_i
P Q R S T U V
T1 T2 T3 T4 T5 T6 T7 T8
x
x.key_{i-1}
x.key_i
x.key_{i+1}
··· N S W ···
y = x.c_i
z = x.c_{i+1}
P Q R
T U V
T1 T2 T3 T4 T5 T6 T7 T8



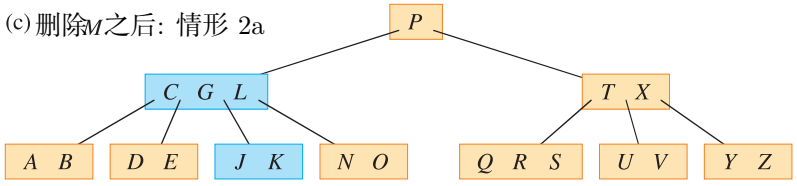
(a)开始时的树



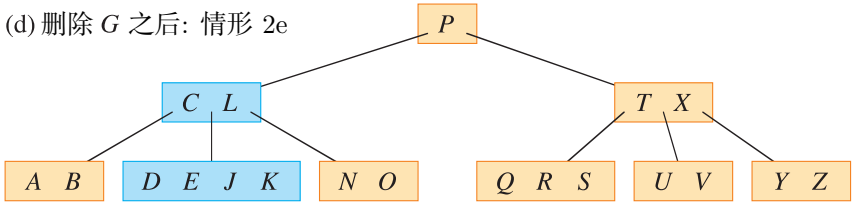
(b)删除F之后：情形 1



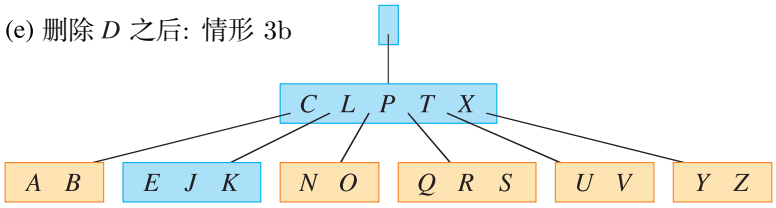
(c)删除M之后：情形 2a



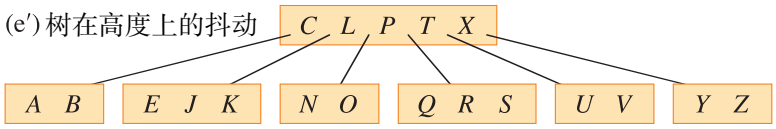
(d) 删除 G 之后: 情形 2e



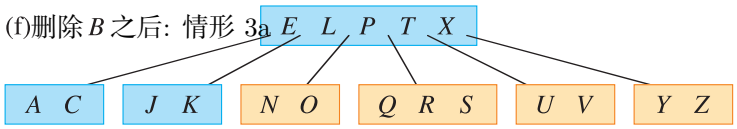
(e) 删除 D 之后: 情形 3b



(e′) 树在高度上的抖动



(f)删除 B 之后: 情形 3a



第四部分

图算法

本部分将介绍图的表示和图的搜索. 图的搜索指的是系统化地跟随图中的边来访问图中的每个结点. 图搜索算法可以用来发现图的结构. 许多的图算法在一开始都会先通过搜索来获得图的结构, 其他的一些图算法则是对基本的搜索加以优化. 可以说, 图的搜索技巧是整个图算法领域的核心.

22.1 节对图的两种最常见的计算机表示法进行讨论. 这两种表示法分别是邻接链表和邻接矩阵. 22.2 节讲解一种简单的图搜索算法, 称为广度优先搜索, 并演示如何创建一棵广度优先树. 22.3 节讲解深度优先搜索, 同时对此种搜索所访问的结点之间的次序进行讨论, 并对这方面的一些标准结果进行证明. 22.4 节给出深度优先搜索的一个实际应用: 有向无环图中的拓扑排序. 22.5 节则讨论深度优先搜索的另一个实际应用: 在有向图中计算强连通分量.

第十五章　图的表示

对于图 $G = (V, E)$, 可以用两种标准表示方法表示. 一种表示法将图作为邻接链表的组合, 另一种表示法则将图作为邻接矩阵来看待. 两种表示方法都既可以表示无向图, 也可以表示有向图. 邻接链表因为在表示稀疏图 (边的条数 $|E|$ 远远小于 $|V|^2$ 的图) 时非常紧凑而成为通常的选择. 本书给出的多数图算法都假定作为输入的图是以邻接链表方式进行表示的. 不过, 在稠密图 ($|E|$ 接近 $|V|^2$ 的图) 的情况下，我们可能倾向于使用邻接矩阵表示法. 另外, 如果需要快速判断任意两个结点之间是否有边相连, 可能也需要使用邻接矩阵表示法.

对于图 $G = (V, E)$ 来说, 其邻接链表表示由一个包含 $|V|$ 条链表的数组 Adj 所构成, 每个结点有一条链表. 对于每个结点 $u \in E$, 邻接链表 $Adj[u]$ 包含所有与结点 u 之间有边相连的结点 v, 即 $Adj[u]$ 包含图 G 中所有与 u 邻接的结点 (也可以说, 该链表里包含指向这些结点的指针). 由于邻接链表代表的是图的边, 在代码里, 我们将数组 Adj 看做是图的一个属性, 就如我们将边集合 E 看做是图的属性一样. 因此, 在代码里, 我们将看到 $G.Adj[u]$ 这样的表示. 图 15.1(a) 给出的是一个无向图, 图 15.1(b) 给出的是图 15.1(a) 的邻接链表表示. 类似地, 图 15.2(b) 给出的是图 15.2(a) 的有向图的邻接链表表示.

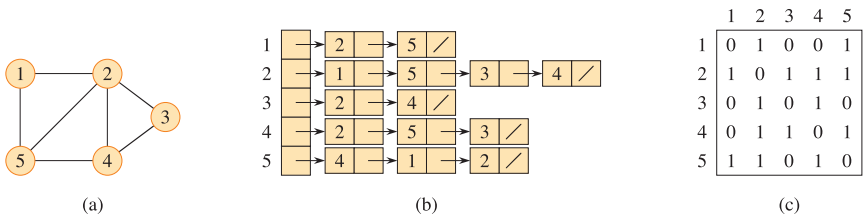
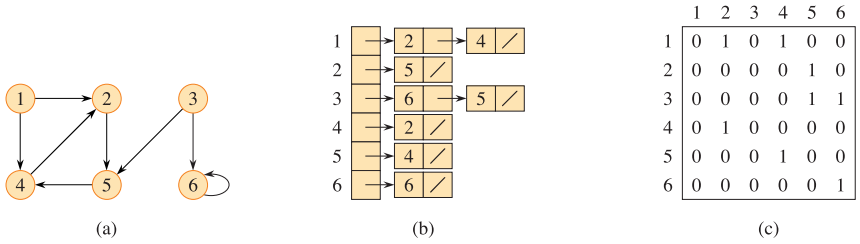


图 15.1: 无向图的两种表示. (a) 一个有 5 个结点和 7 条边的无向图 G. (b)G 的邻接链表表示. (c)G 的邻接矩阵表示

如果 G 是一个有向图, 则对于边 (u,v) 来说, 结点 v 将出现在链表 $Adj[u]$ 里, 因此, 所有邻接链表的长度之和等于 $|E|$. 如果 G 是一个无向图, 则对于边 (u,v) 来说, 结点 v 将出现在链表 $Adj[u]$ 里, 结点 u 将出现在链表 $Adj[v]$ 里, 因此, 所有邻接链表的长度之和等于 $2|E|$. 但不管是有向图还是无向图, 邻接链表表示法的存储空间需求均为 $\Theta(V+E)$, 这正是我们所希望的数量级.



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

图 15.2: 有向图的两种表示. (a) 一个有 6 个结点和 8 条边的有向图 G. (b)G 的邻接链表表示. (c)G 的邻接矩阵表示

邻接链表的一个潜在缺陷是无法快速判断一条边 (u,v) 是否是图中的一条边, 唯一的办法是在邻接链表 $Adj[u]$ 里面搜索结点 v. 邻接矩阵表示则克服了这个缺陷, 但付出的代价是更大的存储空间消耗 (存储空间的渐近数量级更大).

对千邻接矩阵表示来说, 我们通常会将图 G 中的结点编为 $1,2,\ldots,|V|$, 这种编号可以是任意的. 在进行此种编号之后, 图 G 的邻接矩阵表示由一个 $|V| \times |V|$ 的矩阵 $A=(a_{ij})$ 予以表示, 该矩阵满足下述条件:

$$a_{ij} = \begin{cases} 1 & \text{如果}(i,j) \in E \\ 0 & \text{其它} \end{cases}$$

图 15.1(c) 和图 15.2(c) 分别给出的是图 15.1(a) 的无向图和图 15.2(a) 的有向图的邻接矩阵表示. 不管一个图有多少条边, 邻接矩阵的空间需求皆为 $\Theta(V^2)$.

从图 15.1(c) 可以看到, 无向图的邻接矩阵是一个对称矩阵. 由于在无向图中, 边 (u,v) 和边 (v,u) 是同一条边, 无向图的邻接矩阵 A 就是自己的转置, 即 $A=A^T$. 在某些应用中, 可能只需要存放对角线及其以上的这部分邻接矩阵 (即半个矩阵), 从而将图存储空间需求减少几乎一半.

虽然邻接链表表示法和邻接矩阵表示法在渐近意义下至少是一样空间有效

的, 但邻接矩阵表示法更为简单, 因此在图规模比较小时, 我们可能更倾向于使用邻接矩阵表示法. 而且, 对于无向图来说, 邻接矩阵还有一个优势: 每个记录项只需要 1 位的空间.

程序 15.1: 无向图的邻接表存储方式

```java
import java.util.HashMap;
import java.util.List;

public class UndirectedGraphAdjacencyList {
    public static void main(String[] args) {
        var graph = new HashMap<Integer, List<Integer>>();
        graph.put(1, List.of(2, 5));
        graph.put(2, List.of(1, 5, 3, 4));
        graph.put(3, List.of(2, 4));
        graph.put(4, List.of(2, 5, 3));
        graph.put(5, List.of(4, 1, 2));

        System.out.println(graph.get(1).contains(2));
    }
}
```

程序 15.2: 无向图的邻接矩阵存储方式

```java
public class UndirectedGraphMatrix {
    public static void main(String[] args) {
        int[][] graph = {
                {0,1,0,0,1},
                {1,0,1,1,1},
                {0,1,0,1,0},
                {0,1,1,0,1},
                {1,1,0,1,0}
        };

        System.out.println(graph[0][1] == 1);
    }
}
```

程序 15.3: 有向图的邻接表存储方式

```java
import java.util.HashMap;
import java.util.List;

public class DirectedGraphAdjacencyList {
    public static void main(String[] args) {
        var graph = new HashMap<Integer, List<Integer>>();

        graph.put(1, List.of(2, 4));
        graph.put(2, List.of(5));
        graph.put(3, List.of(6, 5));
        graph.put(4, List.of(2));
        graph.put(5, List.of(4));
        graph.put(6, List.of(6));

        System.out.println(graph.get(1).contains(2));
    }
}
```

程序 15.4: 有向图的邻接矩阵存储方式

```java
public class DirectedGraphMatrix {
    public static void main(String[] args) {
        int[][] graph = {
                {0,1,0,1,0,0},
                {0,0,0,0,1,0},
                {0,0,0,0,1,1},
                {0,1,0,0,0,0},
                {0,0,0,1,0,0},
                {0,0,0,0,0,1}
        };

        System.out.println(graph[0][1] == 1);
    }
}
```

第十六章　广度优先搜索

广度优先搜索是最简单的图搜索算法之一, 也是许多重要的图算法的原型.

给定图 $G=(V,E)$ 和一个可以识别的源结点 s, 广度优先搜索对图 G 中的边进行系统性的探索来发现可以从源结点 s 到达的所有结点. 该算法能够计算从源结点 s 到每个可到达的结点的距离 (最少的边数), 同时生成一棵 "广度优先搜索树". 该树以源结点 s 为根结点, 包含所有可以从 s 到达的结点. 对于每个从源结点 s 可以到达的结点 v, 在广度优先搜索树里从结点 s 到结点 v 的简单路径所对应的就是图 G 中从结点 s 到结点 v 的 "最短路径", 即包含最少边数的路径. 该算法既可以用于有向图, 也可以用于无向图.

广度优先搜索之所以如此得名是因为该算法始终是将已发现结点和未发现结点之间的边界, 沿其广度方向向外扩展. 也就是说, 算法需要在发现所有距离源结点 s 为 K 的所有结点之后, 才会发现距离源结点 s 为 $k+1$ 的其他结点.

为了跟踪算法的进展, 广度优先搜索在概念上将每个结点涂上白色, 灰色或黑色. 所有结点在一开始的时候均涂上白色. 在算法推进过程中, 这些结点可能会变成灰色或者黑色. 在搜索过程中, 第一次遇到一个结点就称该结点被 "发现", 此时该结点的颜色将发生改变. 因此, 凡是灰色和黑色的结点都是已被发现的结点. 但广度优先搜索对灰色和黑色结点加以区别, 以确保搜索按照广度优先模式进行推进. 如果边 $(u,v)\in E$ 且结点 u 是黑色, 则结点 v 既可能是灰色也可能是黑色. 也就是说, 所有与黑色结点邻接的结点都已经被发现. 对于灰色结点来说, 其邻接结点中可能存在未被发现的白色结点. 灰色结点所代表的就是已知和未知两个集合之间的边界.

在执行广度优先搜索的过程中将构造出一棵广度优先树. 一开始, 该树仅含有根结点, 就是源结点 s. 在扫描已发现结点 u 的邻接链表时, 每当发现一个白色结点 v, 就将结点 v 和边 (u,v) 同时加入该棵树中. 在广度优先树中, 称结点 u 是结点 v 的前驱或者父结点. 由于每个结点最多被发现一次, 它最多只有一个父结点. 广度优先树中的祖先和后代关系皆以相对于根结点 s 的位置来进行定义: 如果结点 u 是从根结点 s 到结点 v 的简单路径上的一个结点, 则结点 u 是

结点 v 的祖先, 结点 v 是结点 u 的后代.

在下面给出的广度优先搜索过程 BFS 中, 假定输入图 $G = (V, E)$ 是以邻接链表所表示的. 该算法为图中每个结点赋予了一些额外的属性: 我们将每个结点 u 的颜色存放在属性 $u.color$ 里, 将 u 的前驱结点存放在属性 $u.\pi$ 里. 如果 u 没有前驱结点 (例如, 如果 $u = s$ 或者结点 u 尚未被发现), 则 $u.\pi = NIL$. 属性 $u.d$ 记录的是广度优先搜索算法所计算出的从源结点 s 到结点 u 之间的距离. 该算法使用一个先进先出的队列 Q 来管理灰色结点集.

程序 16.1:

```
lass Vertex {
    int color;
    int d;
    Vertex prev;
}

class Graph {
    List<Vertex> V;
    Map<Vertex, List<Vertex>> Adj;
}


int WHITE = 0;
int GRAY  = 1;
int BLACK = 2;

public static void BFS(Graph G, Vertex s) {
    for (Vertex u : G.V.remove(s)) {
        u.color = WHITE;
        u.d = Integer.MAX_VALUE;
        u.pi = null;
    }
    s.color = GRAY;
    s.d = 0;
    s.pi = null;
    var Q = new ArrayList<Vertex>();
    Q.add(s);
```

```
    while (Q.size() != 0) {
        var u = Q.remove(0);
        // 搜索 u 的所有邻居
        for (var v : G.Adj[u]) {
            // v 被发现了吗?
            if (v.color == WHITE) {
                v.color = GRAY;
                v.d = u.d + 1;
                v.pi = u;
                // 将 v 入队列
                Q.add(v);
            }
        }
        // u 被遍历到了
        u.color = BLACK;
    }
}
```

程序 16.2:

```
oid PrintPath(Graph G, Vertex s, Vertex v) {
    if (v == s)
        System.out.println(s);
    else if (v.prev == null)
        System.out.println(" 不存在从" + s + " 到" + v +
        ↪  " 的路径");
    else {
        PrintPath(G, s, v.prev);
        System.out.println(v);
    }
}
```

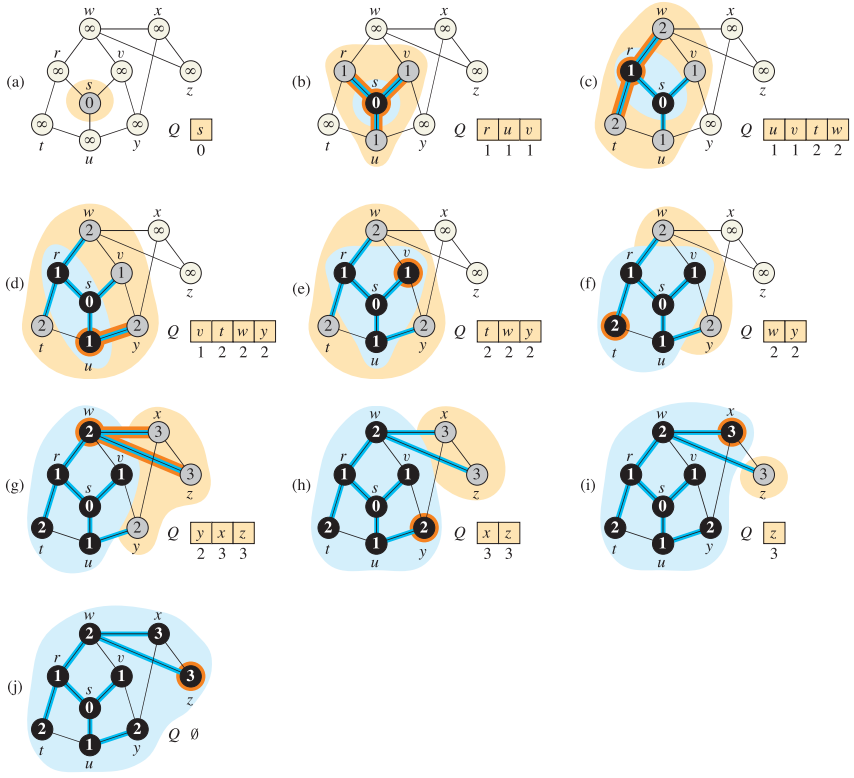
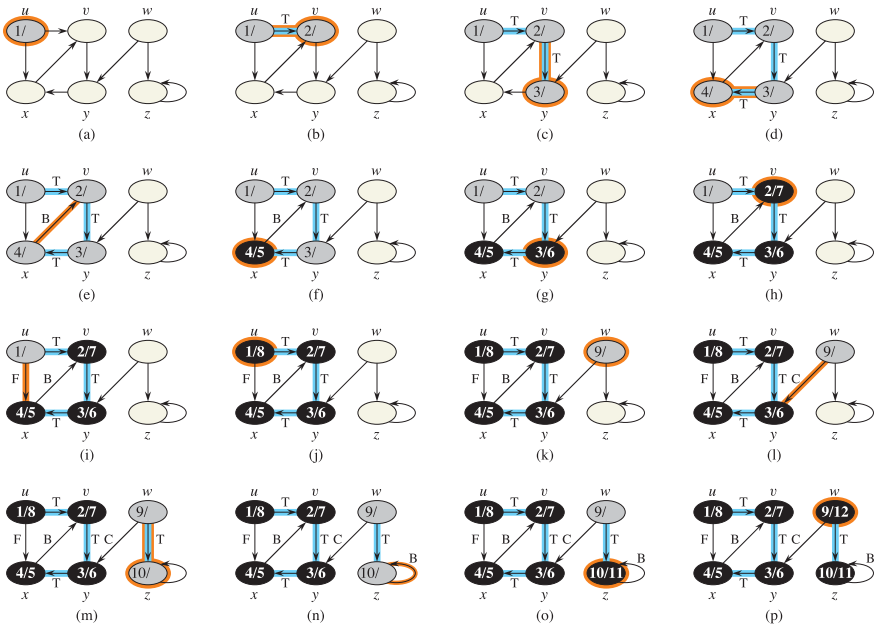


图 16.1: BFS 在无向图上的运行过程. 添加了阴影的边是被 BFS 发现的边. 每个结点 u 里面记录的是 $u.d$ 的值. 途中还给出了在算法第 10-18 行的 while 循环每次开始时的队列 Q 的内容. 结点距离标记在队列相应结点的下方

第十七章　深度优先搜索

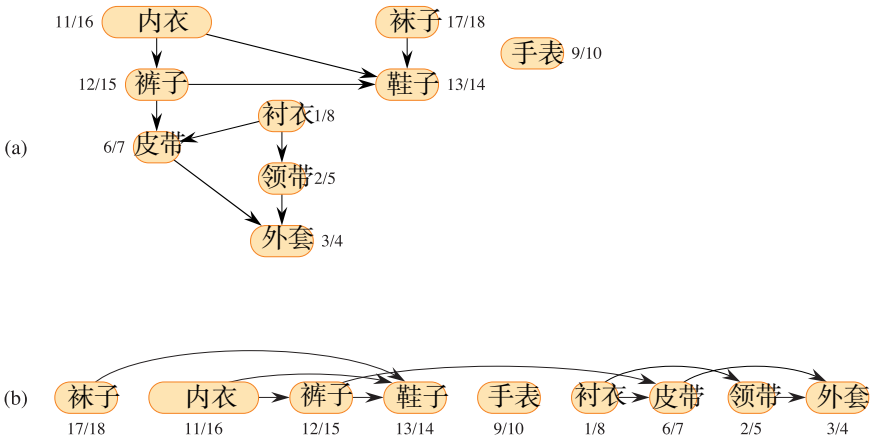


程序 17.1:

```
nt time;

void DFS(Graph G) {
    for (var u : G.V) {
        u.color = WHITE;
        u.prev = null;
```

```
    }
    time = 0;
    for (var u : G.V) {
        if (u.color == WHITE)
            DfsVisit(G, u);
    }
}

void DfsVisit(Graph G, Vertex u) {
    time = time + 1;
    u.d = time;
    u.color = GRAY;
    for (var v : G.Adj[u]) {
        if (v.color == WHITE) {
            v.prev = u;
            DfsVisit(G, v);
        }
    }
    time = time + 1;
    u.f = time;
    u.color = BLACK;
}
```

第十八章　拓扑排序


(a)
11/16
内衣
袜子
17/18
手表
9/10
12/15
裤子
鞋子
13/14
衬衣
1/8
6/7
皮带
领带
2/5
外套
3/4
(b)
袜子
17/18
内衣
11/16
裤子
12/15
鞋子
13/14
手表
9/10
衬衣
1/8
皮带
6/7
领带
2/5
外套
3/4

第五部分

高级设计和分析技术

第十九章　组合搜索问题

第二十章　动态规划

第二十一章　贪心算法

第二十二章　字符串匹配和确定性有限状态自动机

第二十三章　正则表达式和非确定性有限状态自动机