数据结构与算法（Java 描述）

邓俊辉　著

机械工业出版社

目录

前言

关于计算机教育规范，最有权威、影响最大的莫过于 ACM 与 IEEE-CS 联合制订的 Computing Curricula。比如，Computing Curricula 1991（CC1991）就曾对我国高校的计算机教育产生深刻的影响。 正在制订中的 CC2001（后改称 CC2004）认为，随着近年来计算概念的快速发展，计算学科已经发展成为一个内涵繁杂的综合性学科，至少可以划分为计算机工程（CE）、计算机科学（CS）、信息系统（IS）、信息技术（IT）和软件工程（SE）等五个领域，而且不同领域的人才所应具备的知识结构与能力侧重也不尽相同。尽管如此，从目前已经完成的部分来看，数据结构与算法在各领域的知识体系中仍然占据着重要的位置。比如在 CE 分卷（草案）中，Programming Fundame ntals 共 39 个核心学时，其中 Algorithms and Problem Solving 占 8 个学时，Data Structures 占 13 个学时；在 CS 分卷（正式稿）中，Programming Fundamentals 共 38 个核心学时，其中 Algorithms and Problem Solving 占 6 个学时，Data Structures 占 14 个学时。

为什么数据结构与算法长期以来会一直受到如此重视？数据结构和算法是一对孪生兄弟，数据结构研究的问题包括数据在计算机中存储、组织、传递和转换的过程及方法，这些也是构成与支撑算法的基础。我们在编写计算机程序来解决应用问题时，最终都要归结并落实到这两个问题上。正因为此，N. Wirth 早在上世纪 70 年代就曾指出“程序 = 数据结构 + 算法”。近年来，随着面向对象技术的广泛应用，从数据结构的定义、分类、组成，到设计、实现与分析的模式与方法都有了长足的发展，现代数据结构更加注重和强调数据结构的整体性、通用性、复用性、简洁性和安全性。

为遵循上述原则，本书选择 Java 作为描述语言，因为相对于其它语言，Java 语言比较完整、彻底地体现了面向对象的设计思想，这也是目前国际上同行们的一个主要取向。不过，关于 Java 语言本身的介绍与交待，本书并未花费多少笔墨，而是直接假定读者已经熟练掌握了该语言。这既能缩减本书篇幅，也符合此领域教学今后的规范。比如，CC2001 的 CS 分卷将 Data Structures and Algorithms 课程编号为 CS103，并明确指出该课程是 Programming Fundamentals (CS101)和 The Object-Oriented Paradigm (CS102)的后续课程。

在内容选取、剪裁与体例结构上，作者力图突破现成的模式。这里并未对各种数据结构面面俱到，而是通过分类和讲解典型结构，力图使读者形成对数据结构的宏观认识；结合各部分的具体内容，书中都穿插了大量的问题，把更多的思考空间留给读者。

根据内容侧重，本书分为以下具体部分。

第一章是全书的基础，重点在于介绍算法与数据结构的关系，以及算法时间、空间复杂度的概念及其度量方法。

第二至四章覆盖了基本的数据结构，既有传统的栈与队列，也有更为抽象和通用的向量和列表。面向对象技术在现代数据结构理论中的重要地位在此得到体现，我们利用接口实现数据结构的封装，抽象出位置的概念，在向量和列表的基础上定义并实现序列结构，引入迭代器概念并针对上述结构具体实现。通过结合数组与链表结构的优点，第四章自然地导出了树结构的概念、实现及算法。

第五、六和七章介绍了若干高级数据结构。通过在一般性队列中引入优先级概念，导出了优先队列结构；面向实际问题中的查找需求，导出了映射和词典结构；针对全序集的查找问题，引入了查找数结构，并结合 **AVL** 树、伸展树和 **B-**树的实例介绍了平衡查找树结构的原理及其实现。面向对象的思想在这里继续得到贯彻，普遍采用了抽象、封装及继承等技术。

这一部分的侧重点逐渐转向算法，并结合具体问题介绍其应用与实现，包括堆结构的生成及调整算法、**Huffman** 编码树算法、散列算法以及二路或多路平衡查找树的生成、插入、删除算法。在这三章中，读者也将对算法复杂度的各种分析方法有所了解。

第八、九和十章的论述重点完全转向算法。第八章对此前各章中陆续介绍过的排序算法作了归纳与分类，着重介绍了归并排序算法与快速排序算法，并给出了此类算法的复杂度下界。结合串结构的应用，第九章着重讨论了串匹配问题，从蛮力算法出发，采用不同的启发式加速策略，分别介绍了 **KMP** 算法及 **BM** 算法。

关于图结构，第十章将重点放在相关算法上，并力图使初学者从各种图算法中梳理出清晰的脉络。为此，这里尝试通过一条主线——遍历——将各种算法串接起来。在这里，面向对象技术的魅力再次得以展现：基于统一的图遍历算法模版，分别实现了深度优先、广度优先和最佳优先三大类遍历算法。实际上，这三类算法本身仍然以模版形式实现，包括边分类、可达分量、连通分量、最短路径以及最小生成树在内的各种具体算法，都进而基于这三个模版分别得到了实现。

书中涉及的所有代码，都符合 **J2SDK-1.4.1** 规范，并构成一个名为 **dsa**（**Data Structures & Algorithms**）的包；所有类之间的扩展、继承关系，都在书后的“**DSA** 类关系图”部分统一介绍。

为获取本书中的代码及相关教学资料，欢迎访问：http://thudsa.3322.org/~deng/ds/dsaj/。

从本书的筹划、撰写、审订到定稿发行，其间跨越三个年头，在此漫长的过程中，我的父母和妻子给了我巨大的支持，就这个意义而言，他们也是本书的完成者。年幼的女儿总是以她独有的方式，不时将我从写作和调试的苦海中解救出来，她让我体会了禁用 **PowerOff** 按钮的妙用（尽管除了拔除接头外我还没有找到更好的办法，使得在她恶作剧地按下 **Reset** 按钮后依然能够继续写作）。我的确需要感谢她，她让我养成了及时备份的良好习惯，更使我意识到在离开电脑后自己依然能够而且更好地思考。

经过在清华大学的多年执教，我日益深刻地体会到教育的伟大和神圣，是我的同事、我的学生使我懂得了如何忘却劳动的艰辛，并从教育的过程中获得乐趣。

我还要特别感谢机械工业出版社的温莉芳女士，在我一次次地推迟交稿时，她表现出的耐心与理解都令我既钦佩又惭愧，正是这非凡的宽容激励着我不断追求完美。

虽反复斟酌，在本书即将出版之际，内心依然惶恐。“丑媳妇终要见公婆”，恳请读者批评指正。

邓俊辉

2005 年仲夏初笔

2005 年岁末定稿

第一章

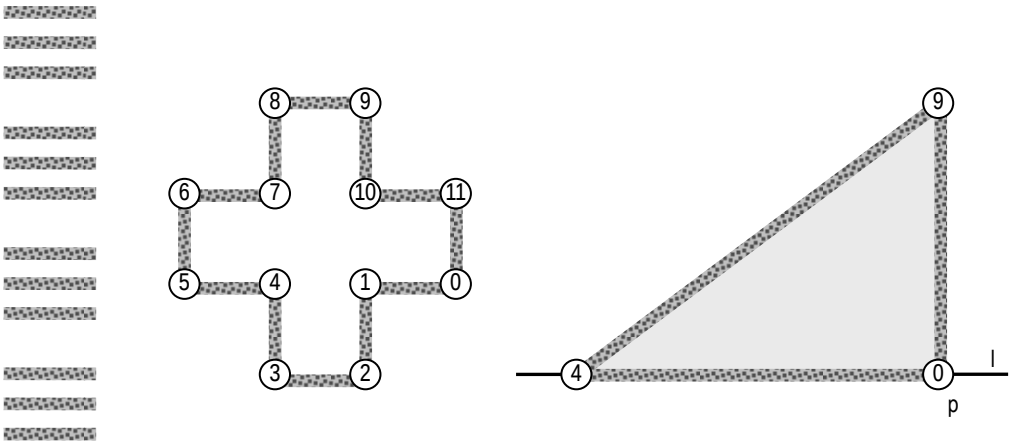
算法及其复杂度

§1.1 计算机与算法

现代意义上的电子计算机诞生于上世纪中叶，然而实际上，从人类在地球上出现之日起，计算与计算机都一直伴随在我们的周围。打结的绳子[一]，刻痕的石头，都是人类的计算工具。下面我们就来看几个例子。

1.1.1 过指定垂足的直角边

在规划和实施复杂而规模浩大的土木工程的过程中，古代埃及人逐渐归纳并掌握了一整套基本方法。比如，早在公元前 2000 年，他们就已经知道了如下实际问题的解决方法：通过直线上给定的一点，作该直线的垂线。



图一.1 公元前 2000 年古埃及人使用的绳索计算机及其算法[二]

具体方法可以描述如下：

```
算法：RightAngle(l, p)
输入：直线l及其上面的一点p
输出：经过p、垂直于l的一条直线
```

[一] 易系辞云：上古结绳而治，后世圣人，易之以书契。

[二] 不难看出，其原理就是勾股定理的逆命题。这一定理最早记载于《周髀算经》(成书于公元前 100~200 年)，后由赵爽给出了初等而严格的证明(公元 220 年前后)。关于这一定理，《周髀算经》记载了商高向周公的一段解释：故禹之所以治天下者，此数之所由生也。也就是说，早在大禹（生于公元前 2297 年 6 月 6 日，卒于公元前 2198 年 8 月）治理黄河时，中国人就已经掌握了这一定理并应用于社会生产实际，比古埃及人的记载要早 100 年。

```
{
  1．取12段等长的绳索，首尾依次联结成一个环; //联结处称作“结”，按顺时针方向编号

  2．将0号结固定于给定的点p处;
  3．第一个奴隶牵动4号结，将绳索沿直线l方向拉直;
  4．另一个奴隶牵动9号结，将绳索尽可能地拉直;
  5．记录下由0号和9号结确定的直线;
}
```

算法一.1 过直线上给定点作直角的算法

从外表看，由古埃及奴隶与绳索组成这一简易工具与现代电子计算机相去甚远，然而就其本质而言，二者却有着很多相似之处。首先，它们都明确地定义了问题输入的形式与要求。其次，基于若干基本的操作（比如取等长绳索、联结绳索、将绳结固定在指定点以及拉直绳索等），具体的操作方法与规程也可以明白无误地得到描述。最后也是最重要的，只要按照这种方法来进行操作，对指定问题的任何输入，它们都能在有限的时间内给出相应的解答。因此从这个意义上讲，将四千年前的这一计算工具称作“绳索计算机”，一点也不过份[1]。

1.1.2 三等分线段

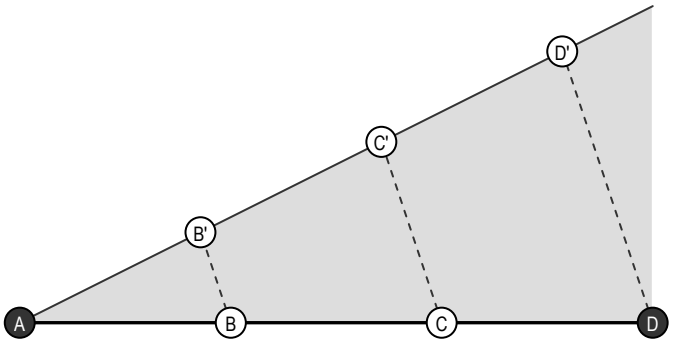
欧几里德几何是现代公理系统的鼻祖。如果从计算机的角度来看，直尺和圆规就相当于我们今天的计算机，而为了解决某一特定几何问题而设计的一套几何作图流程，则相当于今天的一个算法或程序。比如典型的三等分线段过程，就可以描述为如下算法：

```
算法: Tripartition(AB)
输入: 线段AB
输出: 通过线段AB上的两点C和D，将其三等分
{
  从A发出一条与AB不重合的射线ρ;
  在ρ上任取三点C'、D'和B'，使得|AC'| = |C'D'| = |D'B'| = |AB'|/3;
  联接B'B;
  经过D'做B'B的平行线，交AB于D;
  经过C'做B'B的平行线，交AB于C;
}
```

算法一.2 三等分给定线段的算法

如 图一.2 所示的，就是 算法一.2 的一次实际执行结果。

[1] 事实上，这类计算机的功能极其强大，某些方面甚至超过了现代计算机。



图一.2 古希腊人的尺规计算机

当然，算法一.2 中涉及到的操作并不都是基本的，比如，“经过直线外一点作其平行线”本身就是一个几何问题。幸运的是，这些问题都可以借助相应的更为基本的算法来解决（请读者根据初中平面几何的知识给出具体的描述），相对 算法一.2，这些算法的作用相当于“子程序”。

1.1.3 排序

我们再来看另一个算法实例。

据统计，计算机中 80%的计算工作，都可以归结为同一类操作：按照某种次序，将给定的一组元素顺序排列——即排序（Sorting）。比如，将 n 个整数排成一个非降序列。

■ 排序器接口

排序算法种类繁多，在 第八章中我们将专门讨论这一问题。为了遵循面向对象的规范，本书中涉及的排序算法都符合如 代码一.1 所示的排序器接口：

```
/*
 * 排序器接口
 */

package dsa;

public interface Sorter {
  public void sort(Sequence s);
}
```

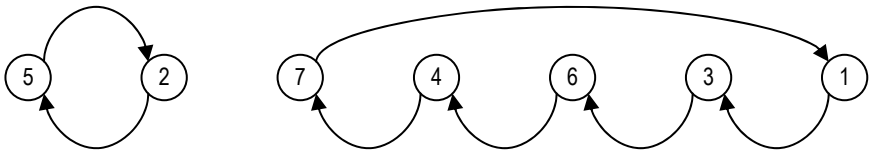
代码一.1 排序器接口

这里，我们将输入统一表示为Sequence类，第 §3.3 节将详细介绍该类的定义及实现。

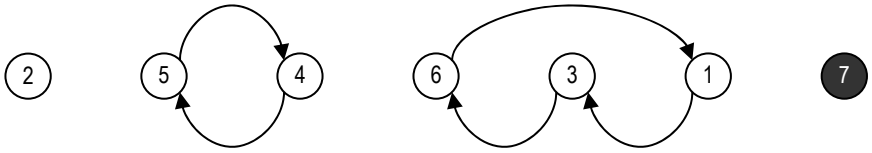
■ 起泡排序

在由一组元素组成的序列 A[0..n-1]中，若某一对相邻元素 A[i-1]和 A[i]满足 A[i-1] ≤ A[i]，我们就称它们是顺序的；否则是逆序的。于是，排序操作的目的，就是通过元素位置的交换，使得所有相邻元素都是顺序的，即对任意的 0 < i < n，都有 A[i-1] ≤ A[i]。

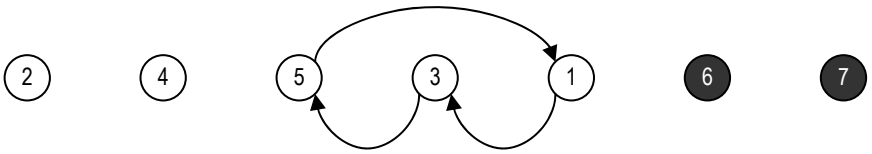
由上述分析可以立即得出如下算法：从前向后逐对检查相邻元素，一旦发现逆序，就交换其位置。这一过程，称作一趟扫描交换。在 图一.3 中，对于如(a)所示的由 7 个整数组成的序列A[0..6]，经过一趟扫描交换之后的结果如(b)所示。



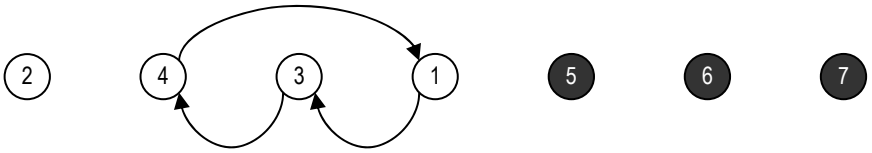
(a) 初始序列



(b)经过一趟扫描交换



(c)经过两趟扫描交换



(d)经过三趟扫描交换

(e)经过四趟扫描交换

(f)经过五趟扫描交换

(g)经过六趟扫描交换

图一.3 对七个整数做起泡排序

经过这样的一趟扫描，如果序列仍未达到完全有序，则可以再次对其进行一趟扫描交换，所得结果如(c)所示。事实上，我们将不断对序列进行扫描交换，直到最终其中不再含有逆序的相邻元素，如(g)所示。

起泡排序的过程可以描述为 算法一.3：

```
算法：Bubblesort(S[], n)
输入：n个元素组成的一个序列S[]，每个元素由[0..n-1]之间的下标确定，元素之间可以比较大小
输出：重新调整S[]中元素的次序，使得它们按照非降次序排列
{
  从S[0]和S[1]开始，依次检查每一对相邻的元素；
  只要它们位置颠倒，则交换其位置；
  反复执行上述操作，直到每一对相邻元素的次序都符合要求；
}
```

算法一.3 起泡排序算法

■ 实现

基于如 代码一.1 所定义的排序器接口，可以实现如 代码一.2 所示的起泡排序器：

```
/*
 * 起泡排序算法
 */

package dsa;

public class Sorter_Bubblesort implements Sorter {
  private Comparator C;

  public Sorter_Bubblesort()
  { this(new ComparatorDefault()); }

  public Sorter_Bubblesort(Comparator comp)
  { C = comp; }

  public void sort(Sequence S) {
    int n = S.getSize();
    for (int i=0; i<n; i++)
    for (int j=0; j<n-i-1; j++)
      if (0 < C.compare(S.getAtRank(j), S.getAtRank(j+1))) {
        Object temp = S.getAtRank(j);
        S.replaceAtRank(j, S.getAtRank(j+1));
        S.replaceAtRank(j+1, temp);
      }
  }
}
```

代码一.2 起泡排序器

■ 正确性

关于起泡排序算法，很自然的一个问题是，需要经过多少趟扫描转换才能完成排序？我们甚至可以怀疑，经过有限趟扫描转换之后，是否总是能够完成排序？也就是说，它是不是一个名副其实的算法。

从 图一.3 不难看出，每经过一趟扫描，尽管并不能保证序列达到完全有序，我们却总是能够在某些方面取得一些进展。如(b)所示，经过第一趟扫描之后，序列中的最大元素必然就位，而且在此后的各轮扫描交换中，该元素将绝不会参与任何交换。因此，经过一趟扫描交换之后，我们可以只关注前面的n-1 个元素——也就是说，问题的规模减少了一个单位。实际上，这一结论对后续的每一趟扫描交换都成立。由此，根据数学归纳法可以得出如下结论：

引理一.1 利用起泡排序算法对长度为 n 的序列进行排序，至多经过 n 轮扫描交换，所有元素都将就位，即实现完全有序。

1.1.4 算法的定义

上面的三个例子，都可以称作算法。那么，究竟什么是算法呢？

所谓算法，就是在特定计算模型下，在信息处理过程中为了解决某一类问题而设计的一个指令序列。比如，对于“过直线上某一点作垂直线”这一问题，绳索及奴隶就构成了一个特定的计算模型，古埃及人基于这一模型所设计的 算法一.1，就是在这一计算模型下解决这一问题的一个算法。

更准确地，一个算法还必须具备以下要素：

- 输入：待处理的信息，即对具体问题的描述。比如，对于上述三个例子来说，输入分别是“任意给定的直线以及其上的一点”、“任意给定的一条线段”以及“由 n 个可比较元素组成的序列”。
- 输出：经过处理之后得到的信息，即问题的答案。比如，对于上述三个例子来说，输出分别是我们所要得到的“垂直线”、“三等分点”以及“完全有序的序列”。
- 确定性：任一算法都可以描述为由若干种基本操作组成的序列。在垂直线算法中，“取等长绳索”、“联结绳索”、“将绳结固定于一点”、“沿特定方向拉直绳索”等操作都属于基本操作。在三等分线段算法中，基本操作就是欧氏作图法所允许的所有尺规操作。而在起泡排序算法中，基本操作就是图灵机所允许的各种操作：“读取某一元素的内容”、“比较两个元素的大小”以及“修改某一元素的内容”等等。
- 可行性：在相应的计算模型中，每一基本操作都可以实现，且能够在常数时间内完成。
- 有穷性：对于任何输入，按照算法，经过有穷次基本操作都可以得到正确的输出。

§1.2 算法性能的分析与评价

1.2.1 三个层次

相信本书的读者，都已学习并掌握了至少一种程序设计语言，比如 Java。学习程序设计语言的目的，就是学会如何编写合法的程序。这里所说的“合法”，指的是合乎该语言的语法，从而保证所编写的程序能够顺利通过编译器生成可执行代码，或者能够通过解释器解释执行。从利用计算机解决实际问题的角度来看，这只是第一个层次，当然也是最基本的要求。

对于算法而言，前面提到的有穷性应该是一项起码的要求，然而遗憾的是，合法的程序却未必满足有穷性——相信大多数读者都曾编写过导致无穷循环或者递归溢出的合法程序。

实际上，我们不仅需要确定算法对任何输入都能够在有穷次操作后终止并输出结果，而且希望等待的时间尽可能地短。为此，我们首先需要确立一种尺度，以度量不同算法的效率（执行速度）；另外，我们还需要学习如何设计和使用适宜的数据结构，编写出效率高、能够处理大规模数据的程序。这些都属于第二层次的问题。前一问题将通过对算法分析理论的学习来解决，后一问题则需要通过对算法设计技巧以及相应的数据结构的学习来解决，这些也正是本书的主题。

如果将借助计算机解决实际问题比作书法，那么在第一层次上就相当于学习点、横、竖、撇和捺等基本笔画，而第二层次则相当于学习如何将不同的笔画按照一定的间架结构组成有意义的不同汉字。然而这还远远不够，作为一幅完整的书法作品，还需要在汉字之间形成大小、粗细、疏密、浓淡、奇正及枯润等方面的搭配与呼应，也就是要讲求章法。在利用计算机解决实际问题的过程中，同样存在这样一个更高层次的问题：倘若软件的规模大到任何个人或少数人都不足以开发出来地步，而且在其生命期内软件也需要很多人的协作才能得以维护，则需要进一步考虑一些更为全局性的问题，这些问题都属于软件工程学的范畴，本书将不做深入介绍。

1.2.2 时间复杂度及其度量

我们首先来解决第二层次的前一个问题：如何度量一个算法的执行速度并评价其效率？具体地，对于不同的输入，算法需要运行多少时间才能得出结果？

- 问题规模、运行时间及时间复杂度

直接回答上述问题并非易事，原因在于，即使是同一算法，针对不同的输入运行的时间并不相同。以排序问题为例，输入序列的规模、组成和次序都不是确定的，这些因素都会影响到排序算法的运行时间。在所有这些因素中，输入的规模是最重要的一个。还是以排序问题为例，如果是操作系统对某一文件夹内的文件按名字排序，则输入的规模通常不超过 100，故可以在瞬间完成排序；反之，若需要对全中国人口普查的数据进行排序，则输入的规模将高达 10^9，此时若采用起泡算法，恐怕需要经过数月甚至数年才能完成排序。

因此，为了简化分析，我们通常只考虑输入规模这一主要因素。如此一来，本节开头所提出的问题就转化为：针对不同规模的输入，算法的执行时间各是多少？如果将某一算法为了处理规模为 n 的问题所需的时间记作 **T**(n)，那么随着问题规模 n 的增长，运行时间 **T**(n)将如何增长？我们将 **T**(n)称作算法的时间复杂度。

■ 渐进复杂度及大 $\mathcal{O}$ 记号

新的问题依然不好回答，原因在于，同一规模的输入仍不确定，通常都有很多个，算法对它们进行处理时所需的时间也不尽相同。仍以排序问题为例，有 n 个元素组成的输入序列有 n!种。因此严格说来，上面对 **T**(n)的定义并不成其为定义（not well-defined）。如果仍然希望籍此度量算法的执行速度，那么在规模为 n 的所有输入中，我们究竟应该以哪一个输入所对应的时间作为 **T**(n)呢？

幸运的是，在评价算法的运行时间时，我们往往可以忽略其在处理小规模问题时的性能，转而关注其在处理足够大规模问题时的性能，即所谓的渐进复杂度（Asmpototic complexity）。原因不难理解，小规模的问题所需的处理时间相对更少，不同算法在效率方面的差异并不明显；只有在处理大规模的问题时，这方面的差异才有质的区别。

另外，通常我们也不需要知道 **T**(n)的确切大小，而只需要对其上界作出估计。比如说，如果存在正常数 a、N 和一个函数 f(n)，使得对于任何 n > N，都有

$$\mathbf{T}(n) < a \times f(n)$$

我们就可以认为在 n 足够大之后，f(n)给出了 **T**(n)的一个上界。对于这种情况，我们记之为

$$\mathbf{T}(n) = \mathcal{O}(f(n))$$

这里的 $\mathcal{O}$ 称作“大 $\mathcal{O}$ 记号（Big-$\mathcal{O}$ notation）”。

■ 机器差异与基本操作

在实际计算环境中，如上定义的 **T**(n)依然无法度量。即便是同一算法、同一输入，在不同的硬件平台上、使用不同的操作系统所需要的计算时间都不相同。然而实际上，无论在何种计算环境中，每一次基本操作都可以在常数时间内完成，因此如果根据算法所需执行的基本操作次数来表示 **T**(n)，就可以更加客观地反映算法的效率。

以第 1.1.3 节介绍的起泡排序算法为例，如果将该算法为了处理长度为n的序列所需的时间记作 **T**(n)，**T**(n)的上界是多少呢？根据上面的分析，我们只需估计出该算法所需执行的基本操作次数。

从 算法一.3 和 代码一.2 都可以看出，该算法由内、外两层循环组成。内循环从前向后依次检查各相邻的元素对，如有必要，则交换逆序的元素对，因此在每一轮内循环中，至多需要扫描和比较n-1 对元素，至多需要交换n-1 对元素。无论是比较元素大小还是交换元素的位置，都属于基本操作，故每一轮内循环至多需要执行 2(n-1)次基本操作。

另外，根据 引理一.1，外循环至多执行n-1 轮。因此，总共需要执行的基本操作不超过 $2(n-1)^2$ 次。若取a=2 和$f(n)=n^2$，则有

$$\mathbf{T}(n) < 2 \times n^2$$

也就是说

$$\mathbf{T}(n) = \mathcal{O}(n^2)$$

以上可以看出，大 $\mathcal{O}$ 记号实质上是对算法执行效率的一种保守估计——对于规模为 n 的任意输入，算法的运行时间都不会超过 $\mathcal{O}(f(n))$。

■ 算法运行时间的实测统计

上面对起泡排序算法的分析，是估计算法时间复杂度的最直接的方法，我们在本书中还将看到更多这样的例子。然而有些算法的时间复杂度极难从理论上作出分析，此时我们可以采用实验的方法，随机选择足够多规模不同的输入，通过实测统计得出运行时间随输入规模而增长的趋势。

■ 大Ω记号

如果存在正常数 a、N 和一个函数 g(n)，使得对于任何 n > N，都有

$$\mathbf{T}(n) > a \times g(n)$$

我们就可以认为在 n 足够大之后，g(n)给出了 **T**(n)的一个下界。对于这种情况，我们记之为

$$\mathbf{T}(n) = \Omega(g(n))$$

这里的Ω称作“大Ω记号（Big-Ω notation）”。

大Ω记号与大 *O* 记号正好相反，它是对算法执行效率的一种乐观估计——对于规模为 n 的任意输入，算法的运行时间都不会低于Ω(g(n))。

■ Θ记号

如果存在正常数 a<b、N 和一个函数 h(n)，使得对于任何 n > N，都有

$$a \times h(n) < \mathbf{T}(n) < b \times h(n)$$

我们就可以认为在 n 足够大之后，h(n)给出了 **T**(n)的一个确界。对于这种情况，我们记之为

$$\mathbf{T}(n) = \Theta(h(n))$$

Θ记号是对算法执行效率的一种准确估计——对于规模为 n 的任意输入，算法的运行时间都与Θ(h(n))同阶。

1.2.3 空间复杂度

衡量算法性能的另一个重要方面，就是算法所需使用的存储空间量，即算法空间复杂度。显然，对于同样的输入规模，在时间复杂度相同的前提下，我们希望算法所占用的空间越少越好。为此，我们可以借助第 1.2.2 节所引入的各种记号来度量算法的空间复杂度，在此不再赘述。

不过，在通常情况下，我们将更多地分析和讨论算法的时间复杂度，甚至只关注时间复杂度。之所以能够这样做，是基于以下事实：

观察结论一.1 就渐进复杂度的意义而言，在任何一个算法的任何一次运行过程中，其实际占用的存储空间都不会多于其间执行的基本操作次数。

实际上，每次基本操作只会涉及到常数规模的空间。于是，纵然每次基本操作所占用的存储空间都是新开辟的，所需的空间总量也不过与基本操作的次数同价。从这个意义上说，时间复杂度本身就是空间复杂度的一个上界。

当然，空间复杂度本身也有其存在的意义，尤其是在对空间效率非常在乎的应用场合，或者是当问题的输入规模极为庞大时。在本书后续的章节中，我们将结合一些实际的问题就此进行讨论。

§1.3 算法复杂度及其分析

我们不仅要了解算法复杂度的度量规则，更要学会如何对各个具体算法的复杂度进行分析。在第 1.2.2 节所引入的度量算法复杂度的三种记号中，大O记号是最基本的，也是最常用到的。按照渐进复杂度的思想，可以将算法的复杂度按照高低划分为若干典型的级别。

1.3.1 $O(1)$——取非极端元素

首先来看这样一个问题：给定整数子集S, $+\infty > |S| = n \geq 3$，从中找出一个元素$a \in S$，使得$a \neq \max(S)$且$a \neq \min(S)$。也就是说，在最大、最小者之外，取出任意一个数。这一问题可以用 算法一.4 解决：

```
算法：NonextremeElement(S[], n)
输入：由n个整数构成的集合S
输出：其中的任一非极端元素
{
  任取的三个元素x, y, z ∈ S; //既然S是集合，这三个元素必互异
  通过比较，找出其中的最小者min{x, y, z}和最大者max{x, y, z};
  输出最小、最大者之外的那个元素;
}
```

算法一.4　取非极端元素

算法一.4 的正确性不言而喻，但它需要运行多少时间呢？与输入的规模n有何联系？

我们注意到，既然 S 是有限集，故其中的最大、最小元素各有且仅有一个。因此，无论 S 的规模有多大，在前三个元素 S[0]、S[1]和 S[2]中，必包含至少一个非极端元素。于是，我们可以取 x = S[0]、y = S[1]和 z = S[2]，这只需执行三次基本操作，耗费$O(3)$时间。接下来，为了确定这三个元素的大小次序，我们最多需要做三次比较（请读者自己给出证明），也是$O(3)$时间。最后，输出居中的那个元素只需$O(1)$时间。

综合起来，算法一.4 的运行时间为：

$$T(n) = O(3) + O(3) + O(1) = O(7) = O(1)$$

也就是说，算法一.4 具有常数的时间复杂度。

1.3.2 $O(\log n)$——进制转换

考虑如下问题：给定任一十进制整数，将其转换为三进制表示。比如

$$23_{(10)} = 212_{(3)}$$

$$101_{(10)} = 10202_{(3)}$$

这一问题可以由 算法一.5 解决：

```
算法：BaseConversion(n)

输入：十进制整数n

输出：n的三进制表示
{
  不断循环，直到n = 0 {
    输出 n mod 3; //取模
    令 n = n/3; //整除
  }
}
```

算法一.5 三进制转换

以 $101_{(10)}$为例。第一轮循环，输出 101 mod 3 = 2，n = 100/3 = 33；第二轮循环，输出 33 mod 3 = 0，n = 33/3 = 11；第三轮循环，输出 11 mod 3 = 2，n = 11/3 = 3；第四轮循环，输出 3 mod 3 = 0，n = 3/3 = 1；第五轮循环，输出 1 mod 3 = 1，n = 1/3 = 0，至此算法结束。请注意，以上各个数位是按照从低到高的次序输出的，所以转换后的结果应该是 $10202_{(3)}$。

实际上，对 算法一.5 稍作修改，即可推广至任意进制，请读者自行完成这一任务。

我们以整数n的大小作为输入规模，来分析 算法一.5 的运行时间。该算法由若干次循环构成，每一轮循环内部，都只需进行两次基本操作（取模、整除）。为了确定需要进行的循环轮数，我们可以注意到以下事实：每经过一轮循环，n都至少减少至 1/3。于是，至多经过 $1+\lfloor \log_3 n \rfloor$次循环，即可减小至 0。

也可以从另一个角度来解释这一结果。该算法的任务是依次给出三进制表示的各个数位，其中的每一轮循环，都恰好给出其中的一个数位。因此，总共需要进行的循环轮数，应该恰好等于 n 的三进制表示的位数，即 $1+\lfloor \log_3 n \rfloor$。因此，该算法需要运行 $\mathcal{O}(2\times(1+\lfloor \log_3 n \rfloor)) = \mathcal{O}(\log_3 n)$时间。

鉴于大 $\mathcal{O}$ 记号的性质，我们通常会忽略对数函数的常底数。比如这里的底数为常数 3，故通常将上述复杂度记作 $\mathcal{O}(\log n)$。请读者根据大 $\mathcal{O}$ 记号的定义证明：就渐进复杂度的意义而言，对数函数的常底数具体是多少是无所谓的。此时，我们称这类算法具有对数的时间复杂度，

1.3.3 $\mathcal{O}(n)$——数组求和

考虑如下问题：给定n个整数，计算它们的总和。这一问题可以由 算法一.6 解决。

```
算法：Sum(A[], n)
输入：由n个整数组成的数组A[]
输出：A[]中所有元素的总和
{
  令s = 0;
  对于每一个A[i]，i = 0, 1, …, n-1
    令s = s + A[i];
```

```
  输出s;
}
```

算法一.6 计算数组元素总和

算法一.6 的正确性一目了然，它需要运行多少时间呢？

首先，对s的初始化需要$O(1)$时间。算法的主体部分是一个循环，每一轮循环中只需进行一次累加运算，这属于基本操作，可以在$O(1)$时间内完成。每经过一轮循环，都对一个元素进行累加，故总共需要做n轮循环。因此，算法一.6 的运行时间为

$$O(1) + O(1)\times n = O(n+1) = O(n)$$

我们称这类算法具有线性的时间复杂度。这类问题的另一个例子是最大元素问题：在 n 个整数中找出最大者。请读者自行设计一个需要运行线性时间的算法，并对其复杂度做一分析。

1.3.4 $O(n^2)$——起泡排序

还记得第 1.1.3 节中的起泡排序算法吗？让我们回过头来更为仔细地分析一下 算法一.3 的时间复杂度。

根据 引理一.1，为了对n个整数排序，该算法的外循环最多需要做n轮。经过第i轮循环，元素S[n-i-1]必然就位，i = 0, 1, …, n-1。在第i轮外循环中，内循环需要做n-i-1 轮。在每一轮内循环中，需要做一次比较操作，另外至多需要做三次赋值操作，这些都属于基本操作，可以在$O(4)$的时间内完成。因此，该算法总共需要的运行时间为：

$$\mathbf{T}(n) = \sum_{i=0}^{n-1}(n-i-1)\times O(4) = O(2n(n-1)) = O(2n^2 - 2n)$$

鉴于大 O 记号的特性，低次项可以忽略，常系数可以简化为 1，故再次得到

$$\mathbf{T}(n) = O(n^2)$$

我们称这类算法具有平方时间复杂度。对于其它一些算法，n 的次数可能更高，但只要其次数为常数，我们都统称之为“多项式时间复杂度”。

另外，对于任意的n > 0，都存在规模为n的输入序列，使得 算法一.3 的确需要运行$\Omega(n^2)$时间才能完成对该序列的排序（请读者自行构造这样的实例），故这一复杂度上界是紧的。既然如此，更准确地，我们应该记之为$\mathbf{T}(n) = \Theta(n^2)$。

1.3.5 $O(2^r)$——幂函数

再来考虑幂函数的计算问题：给定非负整数 r，计算 2^r。对于这类数值计算问题，我们更倾向于用输入数值（r）的二进制位数（$n = 1 + \lfloor \log_2 r \rfloor$）做为输入规模。

为了解决这一问题，一种直截了当的方法可以描述为 算法一.7：

```
算法: PowerBruteForce(r)
```

```
输入：非负整数r
输出：幂2^r
{
  power = 1;

  while (0 < r--)

    power = power * 2;
  return power;
}
```

算法一.7 计算幂函数的蛮力算法

算法一.7 总共需要做r次迭代，每次迭代只涉及常数次基本操作，故总共需要运行$O(r)$时间。按照如上定义，问题的输入规模为n，故有$O(r) = O(2^n)$。我们称这样的算法具有指数的时间复杂度。

从常数、对数、线性到平方时间复杂度，算法的效率不断下降，但就实际应用而言，这类算法的效率还在允许的范围内。然而，在多项式时间复杂度与指数时间复杂度之间，却有着一道巨大的鸿沟，通常我们都认为，指数复杂度的算法无法应用于实际问题之中，它们不是有效的算法，甚至不能称作算法。

如幂函数计算之类的问题，实际上还存着更高效的算法（请读者自行设计、实现并分析）。然而绝大多数的实际问题，都不存在多项式时间的算法，也就是说，这类问题的任何一个算法，都至少需要运行指数的时间，甚至无穷的时间。

§1.4 计算模型

1.4.1 可解性

现代意义上的电子计算机所对应的计算模型，就是所谓的图灵机[1]。关于这一模型，图灵本人的一项重要研究成果就是：绝大多数能够用这一模型描述的问题，根本无法用这一模型解决，比如著名的“停机问题”。实际上，在任何足够复杂的计算模型中，这一现象都是普遍存在的。比如在欧氏的尺规作图模型中，就存在大量非常简单明了、但无法解决的问题——倍方问题、三等分角问题…等等。关于这方面的讨论，属于可计算性理论（Theory of computability）的范畴，因此本书只讨论能够借助图灵机模型解决的那些问题。

[1] 由其发明者、著名计算机科学家图灵（Turing, 1912~1954）得名，还有几种与之等价的计算模型，比如λ-演算（lambda calculus）等。

1.4.2 有效可解

更严格地说，本书的讨论范围仅限于那些利用图灵机模型能够有效解决的问题。这里所谓的“有效”，具体来说就是指存在某一算法，能够在多项式时间以内解决这一问题。反之，若某问题的任一算法都具有不低于指数的复杂度，则不是有效可解的。

1.4.3 下界

不难理解，能够有效解决的问题可能存在不止一个多项式复杂度的算法，而且即便是从渐进时间复杂度的角度讲，不同算法所对应的时间复杂度也不尽相同。以第 1.1.3 节引入的排序问题为例，尽管 算法一.3 是解决这一问题的有效算法，但其时间复杂度为$\Theta(n^2)$；然而，在 第五章和 第八章中我们还将介绍若干种不同的排序算法，它们的运行时间仅为$\mathcal{O}(n\log n)$。那么，是否还存在复杂度更低的排序算法呢？答案是否定的。

实际上，在任何一种特定计算模型下，对于任一可有效解决的问题，任何算法的时间复杂度都不可能低于某一范围，我们称之为该问题在这一计算模型下的复杂度下界，或简称该问题的下界。反过来，一旦找到某个算法能够渐进地在这样多的时间内解决该问题，我们就称此算法为该问题的一个最优算法。

比如，针对“取非极端元素”的问题，第 1.3.1 节给出了一个常数时间复杂度的算法，这也是该问题复杂度的下界——即便是从n个整数中随意取出一个元素，也需要这么多时间。从这个意义上说，算法一.4 已经是最优的了。又如，针对“进制转换”问题，第 1.3.2 节给出了一个对数时间复杂度的算法，这也是该问题复杂度的下界——即便是直接将输入整数的三进制的各个数位输出，也需要$\mathcal{O}(\log n)$的时间。因此，算法一.5 也已经是最优的了。再如，针对“数组求和”问题，第 1.3.3 节给出了一个线性时间复杂度的算法，类似地，这也是该问题的复杂度下界——为了得到其总和，我们至少需要访问每个元素一次，仅仅这项工作就需要$\mathcal{O}(n)$时间。因此，算法一.6 也是该问题的一个最优算法。

当然，绝大多数问题的下界并不是这样的显而易见，需要借助一些数学技巧才能证明，在第 §8.3 节中，我们将看到这样的一个实例。

§1.5 递归

递归是高级程序设计语言的一个重要特征，它允许程序中的函数或过程自我调用。比如在 Java 中，当某个方法调用自己时，我们就称之为递归调用（Recursive call ）。还有另一种形式的递归：方法 M 直接调用的虽然是其它方法，但是最终会间接导致 M 被调用。本节所要讨论的主题，就是如何按照递归的模式设计出高效的算法。得益于这一模式，我们可以对实际问题中反复出现的结构和形式加以利用。在描述算法的时候，只要以递归的方式来描述与刻画这些反复出现的结构和形式，就能够避开各种复杂的具体情况、分支以及嵌套的循环。从而更加简捷地描述和实现算法，减少代码量，同时算法的可读性更强，算法的效率也很高。此外，递归也有助于定义具有重复或相似结构的对象。

比如，在现代操作系统中，文件系统的目录结构就是递归定义的。具体来说，每个文件系统都有一个最顶层的目录，其中可以包含若干文件和下一层的子目录；在每个子目录中，也同样可能包含若干文件和再下一层的子目录；如此下去。通过如此的递归定义，文件系统中的目录就可以嵌套任意多层（只要系统的存储资源足以支持）。

再如，现代程序设计语言的语法本身也多以递归形式定义。例如，Java 语言中的参数表就可以递归地定义为：

```
argument-list:
  argument
  argument-list, argument
```

也就是说，每一参数表要么由单个参数构成，要么由另一个参数表和一个参数构成（其间用一个逗号分隔）。这等同于说，每个参数表都是由一系列以逗号分隔的参数构成的。类似地，算术表达式也可以通过基本元素（变量和常量）以及算术表达式本身递归地定义出来，请读者自行给出具体定义。

图一.4 俄罗斯套娃

在艺术及自然科学的各个领域，递归的实例举不胜举。比如俄罗斯套娃（Russian Matryoshka dolls），就是艺术领域中最经典的递归实例：所有娃娃都用木材做成，内部中空；每个娃娃里面藏有一个个头更小的娃娃。

在本节中，我们将从对递归的一般性介绍入手，循序渐进地讲解各种形式的递归（包括线性递归、二分递归、多分支递归等），以及如何通过递归跟踪或递推方程来分析递归算法的复杂度。

1.5.1 线性递归

线性递归是最简单的递归形式，这类方法的每个实例只能递归地调用自己至多一次。比如，有的时候，某些算法问题的输入可以分解为第一个（或最后一个）元素以及剩余的部分，而且剩余的部分又具有与原输入相同的结构，此时就可以利用线性递归。

■ 数组元素的递归求和

这里，我们再次以第 1.3.3 节（12 页）中曾讨论过的数组求和问题为例，介绍如何采用线性递归的策略，对数组A[]中的n个整数进行求和。为此，我们可以注意到：若n=1，则数组的总和就是A[0]；

否则，数组的总和就是前n-1 个整数（A[0..n-2]）的和，再加上A[]的最后一个元素（A[n-1]）。于是，我们可以利用 算法一.8 来解决这一问题。

```
算法：LinearSum(A, n)
输入：包含至少n个整数的数组A，n≥1
输出：A中前n个元素之和
{
  if (n=1)    return A[0];

  else        return LinearSum(A, n-1)+A[n-1];

}
```

算法一.8　通过线性递归计算数组元素之和

从这个例子我们可以看出递归方法的一条重要性质——经过有限的时间后，它必须能够终止。在此，我们需要首先对 n=1 的情况做非递归处理，以保证该方法必然终止。递归方法的另一个特性是，每次递归调用时采用的参数（n-1），都要小于前一次的参数（n），也就是说，随着递归的深入，调用的参数将单调递减。因此，无论最初的输入 n 有多大，递归调用的总次数都是有限的，这也意味着，算法的执行迟早将会终止。最后一次递归调用被称作递归的“基底”，简称“递归基”。当执行到递归基时，算法将执行非递归的计算（返回 A[0]）。一般而言，线性递归式算法都具有如下形式：

- 检测递归基。首先要检测是否到达递归基，也就是最基本、最简单的情况，在这些平凡情况下无需做进一步递归调用。递归基可能包括多种这样的平凡情况；但至少要有一种，而且必须保证这类情况迟早会出现，否则就会导致递归溢出。在 算法一.8 中，通过检查n是否已经减小到 1 来确定是否执行递归基。读者可以自己验证一下，如果缺少这一句检查，会有什么后果。
- 递归处理。如果尚未遇到平凡的情况，则执行一次递归调用。通常，递归调用有多种可能，此时需要经过进一步的检测以判断具体应按何种方式做递归调用。同样地，任一可能的递归调用都必须最终导向某一平凡情况，以保证算法必然终止。

■ 通过递归实现数组倒置

所谓“倒置”，就是将数组A[]中n个元素的前后次序颠倒过来。借助线性递归可以解决这一问题，为此需要注意到如下事实：为了得到数组的倒置，可以先将其首、末元素互换，然后对除这两个元素以外的部分递归地实施倒置。具体的算法如 算法一.9 所示，可以通过ReverseArray(A, 0, n-1)来启动该算法。

```
算法：ReverseArray(A, lo, hi)
输入：数组A，非负整数lo和hi
输出：A[lo..hi]的次序被倒置
{
  if (lo<hi) {
    Swap(A[lo], A[hi]);
    ReverseArray(A, lo+1; hi-1);
```

```
  }
}
```

算法一.9 数组倒置的递归算法

请注意，这一算法的递归基有两种情况：lo = hi（数组长度为奇数时）或 lo > hi（数组长度为偶数时）。长度为奇（偶）数的数组，必然终止于前（后）一种平凡情况，因此算法必会终止。

■ 重新定义问题，以便递归实现

在设计递归算法时，应该从各种角度来考虑对问题的划分，重新对问题做出定义，使得分解后的子问题与原问题具有相同的结构。以上面的 ReverseArray 算法为例，在引入参数 lo 和 hi 后，对数组 A[]及其子数组的递归调用，都可以统一相同的语法形式。当然，该算法的启动形式将不再是 ReverseArray(A)，而应统一为 ReverseArray(A, 0, n-1)。要是很难找出递归算法所需的这种重复性结构，你可以从简单而具体的例子入手，进而找到问题的新定义。

■ 基于线性递归的幂计算

下面，我们重新讨论第 1.3.5 节中的幂函数的计算问题：计算 2 的r次幂，即power(2, r) = 2^r，其中r为非负整数，其二进制表示由$n = 1 + \lfloor \log_2 r \rfloor$个比特位组成。按照线性递归的思想，不难对该函数重定义如下：

$$\text{power}(2, r) = \begin{cases} 1 & \text{若 } r=0 \\ 2\cdot\text{power}(2, r-1) & \text{否则} \end{cases}$$

由此可以直接得到一个递归算法（请读者自行给出），该算法需要做$O(r)$次递归调用才能计算出power(2, r)，就其复杂度而言，与 算法一.7 完全一样。

然而实际上，按照该函数其它形式的递归定义，完全可以更快速地完成计算。下面就是一例：

$$\text{power}(2, r) = \begin{cases} 1 & \text{若 } r=0 \\ 2\cdot\text{power}(2, (r-1)/2)^2 & \text{若 } r>0 \text{且为奇数} \\ \text{power}(2,r/2)^2 & \text{若 } r>0 \text{且为偶数} \end{cases}$$

比如：

$2^4 = (2^2)^2$

$2^5 = 2\times(2^2)^2$

$2^6 = (2^3)^2 = (2\times2^2)^2$

$2^7 = 2\times(2^3)^2 = 2\times(2\times2^2)^2$

……

由此可以直接得到 算法一.10：

```
算法：Power(r)
输入：整数r≥0
输出：幂2^r
{
  if (r=0)      return 1;

  if (r是奇数)   return 2·sqr(Power((r-1)/2));
  else          return sqr(Power(r/2));

}
```

算法一.10 幂函数的线性递归实现

不难看出，每经过一次递归调用，指数 r 都会递减至少一半。因此，只需 $\mathcal{O}(\log r)$次递归调用，即可得到计算结果。与前面提到的 $\mathcal{O}(r)$复杂度相比，几乎降低了一个线性因子，前后两个算法的效率有天壤之别。

■ 尾递归

以上可以看出，利用递归可以编写出简洁而优美的算法。然而，这些优点并非没有代价，为此计算机通常需要使用更多的空间、需要花费额外的时间以跟踪递归调用的过程。在对运行速度要求极高、对存储空间需要精打细算时，往往需要将算法的递归版本改写成非递归的版本。

通常，我们都是利用后面第 §2.1 节将要介绍的栈结构来完成从递归版本到非递归版本的转化，不过在某些特定的条件下，这种转化却可以简明而有效地完成——比如，原算法属于所谓的“尾递归”形式时。在线性递归算法中，若递归调用恰好出现在算法的最后一次操作，我们就称之为尾递归（Tail recursion）。比如 算法一.9 的最后一次操作，是对去除了首、末元素之后的子数组进行递归倒置，这就属于典型的尾递归。

请注意，即使递归调用语句出现在方法体的最后一行，也不见得就是尾递归。严格地说，只有当该方法的任何一次执行（当然，除平凡的递归基外）都以这一递归调用结束时，才属于尾递归。比如，尽管 算法一.8 算法中的最后一行是递归调用，但它却不属于尾递归——请注意，真正的最后一次操作是加法。

只要将尾递归改写为迭代形式，就可以将其转化为功能相同的非递归算法。下面的 算法一.11 给出了一个迭代形式的算法，只要调用IterativeReverseArray(A, 0, n-1)，就能与 算法一.9 一样地完成对数组的倒置。请比较这两个版本之间的差别。

```
算法：IterativeReverseArray(A, lo, hi)
输入：数组A，非负整数lo和hi
输出：A[lo..hi]的次序被倒置
{
  while (lo<hi) {
    Swap(A[lo], A[hi]);
    lo++;
```

```
    hi--;
  }
}
```

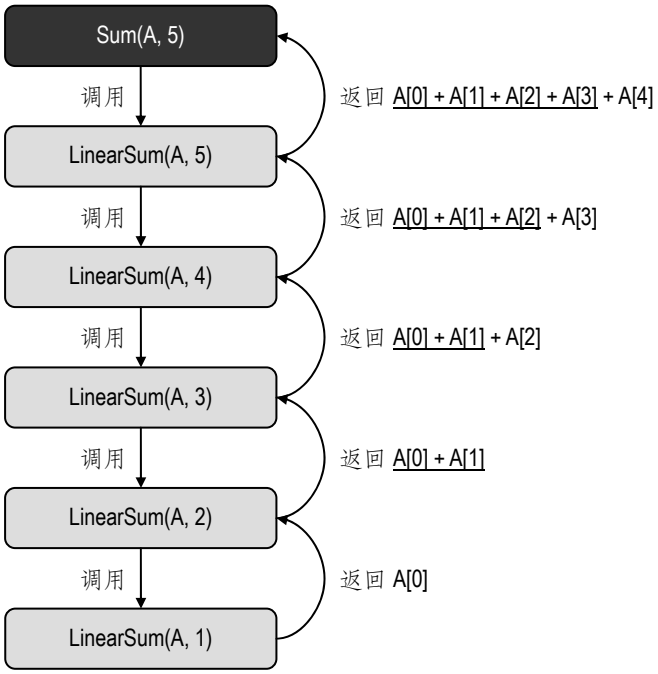
算法一.11 数组倒置的迭代实现

1.5.2 递归算法的复杂度分析

■ 递归跟踪法

递归跟踪（Recursion trace）是一种直观的、可视的分析方法，它可以帮助我们分析递归算法的运行时间。所谓递归跟踪法，就是将递归方法的执行过程表示为图形的形式：方法的每一实例都对应于一个方框，其中注明了该实例调用的参数；若方法实例 M 调用方法实例 N，则在 M 与 N 对应的方框之间添加一条有向联线，指明调用与被调用的关系。

下面，就利用这一方法对LinearSum算法做一分析。按照上述约定，对 算法一.8 的递归跟踪可以表示为如 图一.5 所示的形式。



图一.5 对LinearSum(A, 5)的递归跟踪

其中每个方框分别对应于一次递归调用，方框中还标明了相应的调用参数。每发生一次递归调用，我们就从当前位置向下引出一个箭头，指向新的方法实例所对应的方框。我们从中可以看出算法执行的整个过程：首先对参数 n 进行调用，再转向对参数 n-1 的调用，再转向对参数 n-2 的调用，…，直到最终的参数 1。最后一次递归调用完成之后，将和值 A[0]返回给对参数 2 的调用；累加上 A[1]之后，再返回给对参数 3 的调用；累加上 A[2]之后，继续返回给对参数 4 的调用；…；如此不断返回，直到最终返回给对参数 n 的调用；最后，累加上 A[n-1]之后，就得到了整个数组的总和。

从 图一.5 可以清楚地看出，算法的运行时间实质消耗于各方法实例的非递归部分（如果忽略系统为实现递归而消耗的时间）。在每一次调用中，这部分处理（在此前的部分和的基础上，累计下一整数）均只需常数时间。对于规模为n的数组，递归的深度显然为n，因此算法LinearSum需要运行$n \times \mathcal{O}(1) = \mathcal{O}(n)$时间。

那么，这一算法的空间复杂度又是多少呢？不难看出，当到达最后一次（即最深的那次）递归调用时，算法占用的空间量达到最大。确切的说，此时占用的空间总量，等于所有方框（方法实例）各自所占空间的总和。每个方框对应的空间消耗量显然都是常数，故该算法的空间复杂度正比于递归的深度，为 $n \times \mathcal{O}(1) = \mathcal{O}(n)$。

请读者运用递归跟踪法，自行对 算法一.9 和 算法一.10 的复杂度进行分析。

■ 递推方程法

对递归算法做复杂度分析的另一种常用方法，就是递推方程（Recurrence equation）法。与递归跟踪法相反，这种方法并不直观，而是通过对递归的模式进行归纳从而导出关于复杂度函数的递推方程，递归方程的解将给出算法的复杂度。这一方法的思路与微分方程法极其相似：对于一些极为复杂的函数，我们并不能立即给出其显式的表示，但是它们的微分形式却可能会满足相对简单的方程，通过这种微分方程，我们可以最终导出原函数的显式表示。通常，微分方程的解并不唯一，除非给定一些边界条件。类似地，为了使复杂度函数的递推方程能够给出确定的解，也需要给定某些边界条件——这一任务由递归基完成。

还是以 算法一.8 为例，我们将LinearSum()在处理规模为n的数组时所需的运行时间记作$\mathbf{T}(n)$。该算法的思路可以理解成：为了求解问题LinearSum(A, n)，我们递归地求解问题LinearSum(A, n-1)，然后再累计上A[n-1]。因此，求解问题LinearSum(A, n)所需的时间，应该等于求解LinearSum(A, n-1)的时间，再加上一次加法所需的时间，即

$$\mathbf{T}(n) = \mathbf{T}(n-1) + \mathcal{O}(1) = \mathbf{T}(n-1) + c_1，c_1 \text{为常数}$$

另一方面，当抵达递归基时，求解问题 LinearSum(A, 1)只需常数时间，即

$$\mathbf{T}(1) = \mathcal{O}(1) = c_2，c_2 \text{为常数}$$

联立以上两个方程，可以解得：

$$\mathbf{T}(n) = c_1(n-1) + c_2 = c_1 n + (c_2 - c_1) = \mathcal{O}(n)$$

与递归跟踪法殊途同归。

1.5.3 二分递归

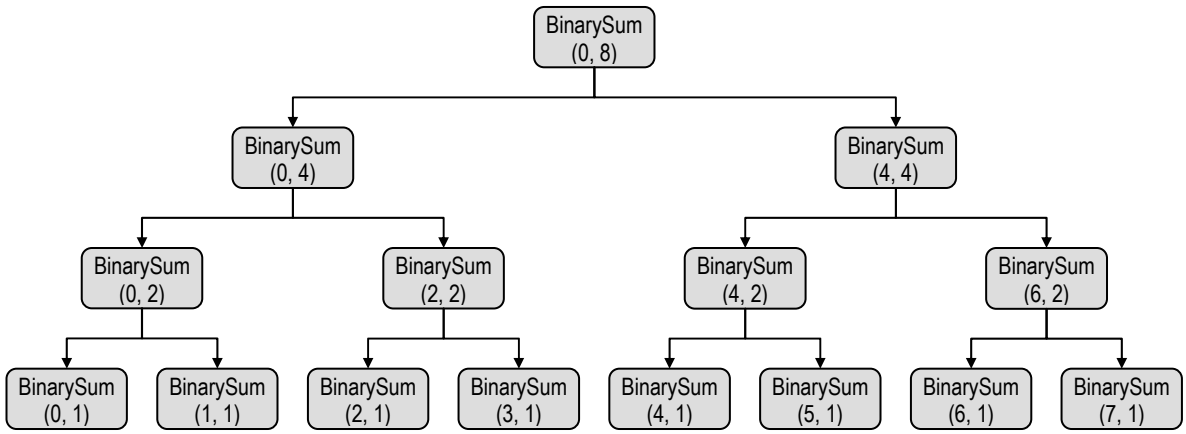
■ 通过二分递归对数组元素求和

有的时候，算法需要将一个大问题分解为两个子问题，然后分别通过递归调用来求解，这种情况称作二分递归（Binary recursion ）。下面，我们就采用二分递归策略来进行数组求和。新算法的思路是：以居中的元素为界，将数组一分为二；递归地对这两个子数组分别求和，它们相加就得到了原数组的总和。具体的方法可以描述为 算法一.12，算法入口的调用形式为BinarySum(A, 0, n)。

```
算法：BinarySum(A, i, n)
输入：数组A，非负整数i和正整数n
输出：A中从位置i起连续n个元素之和
{
  if (n<=1) return A[i];
  else      return BinarySum(A, i, ⌈n/2⌉) + BinarySum(A, i+⌈n/2⌉, ⌊n/2⌋);
}
```

算法一.12 数组求和的二分递归实现

该算法的正确性不言而喻。为了分析该算法的复杂度，不妨假定n是 2 的幂，即$n = 2^k$。图一.6给出了对BinarySum(A, 0, 8)执行过程的递归跟踪。每个框中都标出了对应的i和n值，也就是子数组的起始位置及其长度。可以看出，按照调用的关系及次序，该方法的所有实例构成了一个层次结构（即 第四章将要介绍的树形结构）。沿着这个层次结构，每下降一层，每个实例BinarySum(i, n)都会分裂为一对实例BinarySum(i, n/2)和BinarySum(i+n/2, n/2)——也就是说，每经过一次递归调用，n的数值就会减半。因此，递归的深度（即任一时刻活跃的方法实例的总数）不会超过 $1+\log_2 n$。这说明除了数组本身，该算法只需要$\mathcal{O}(\log n)$的附加空间。考虑到 算法一.8 中的LinearSum算法需要占用$\mathcal{O}(n)$空间，应该说新算法有很大改进。与 算法一.8 相同地，这里的每个方框（方法实例）都只需要常数时间。鉴于这里共有 2n-1 个方框，故BinarySum算法的时间复杂度为$\mathcal{O}(2n-1) = \mathcal{O}(n)$，与LinearSum算法相同。



图一.6 对BinarySum(A, 0, 8)的递归跟踪

■ 通过二分递归计算 Fibonacci 数

下面讨论如何计算第 k 个 Fibonacci 数。你应该记得，Fibonacci 数就是以递归形式定义的：

$$Fib(n) = \begin{cases} 0 & \text{若} n=0 \\ 1 & \text{若} n=1 \\ Fib(n-1)+Fib(n-2) & \text{若} n\geq 2 \end{cases}$$

根据这一定义，可以直接导出算法BinaryFib，如 算法一.13 所示：

```
算法：BinaryFib(k)
输入：非负整数k
输出：Fib(k)
{
  if (k<=1) return k;
  else      return BinaryFib(k-1) + BinaryFib(k-2);
}
```

算法一.13 通过二分递归计算Fibonacci数

不幸的是，尽管表面看来 Fibonacci 数的这一定义与二分递归十分相似，但在这里采用这一策略的效率却极低。实际上，上述算法需要运行 $\mathcal{O}(2^k)$的时间才能计算出第 k 个 Fibonacci 数，也就是说，这是一个指数复杂度的算法，在实际环境中毫无用处。

为了说明这一点，我们将 BinaryFib(k)的运行时间记作 T(k)。根据算法的流程，为了计算 BinaryFib(k)，需要花费 T(k-1)的时间计算 BinaryFib(k-1)，再花费 T(k-2)的时间计算 BinaryFib(k-2)，最后用一个单位的时间将它们累加起来。于是我们就可以得到如下递推式：

$$T(k) = \begin{cases} 1 & \text{若} k \le 1 \\ T(k-1) + T(k-2) + 1 & \text{否则} \end{cases}$$

如果令 S(k) = (T(k) + 1)/2，则有

$$S(k) = \begin{cases} 1 & \text{若} k \le 1 \\ S(k-1) + S(k-2) & \text{否则} \end{cases}$$

我们发现，数列 S(k)的递推形式与 Fib(k)完全一致，只是递归基不同：

$$S(0) = (T(0)+1)/2 = 1 = Fib(1)$$

$$S(1) = (T(1)+1)/2 = 1 = Fib(2)$$

由此可知：

$$S(n) = Fib(n+1)$$

$$T(n) = 2*S(n) - 1 = 2*Fib(n+1) - 1 = \frac{2}{\sqrt{5}} \times \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right] - 1$$

$$T(n) = \mathcal{O}(2^n)$$

■ 通过线性递归计算 Fibonacci 数

上述算法的效率之所以如此低下，是因为其中有大量重复进行的递归调用——只需画出该算法的递归跟踪，即可验证这一结论。我们之所以会想到通过二分递归来解决这一问题，是因为被其外表蒙骗了——Fib(k)是由 Fib(k-1)和 Fib(k)决定的。然而实际上，Fibonacci 数的本质却是线性递归，因此二分递归并不适用于这一问题。

不过，为了应用线性递归，我们需要对Fibonacci数给出新的递归定义。比如，在这里我们可以定义一个递归函数，来计算一对相邻的Fibonacci数(Fib(k), Fib(k-1))，此时约定Fib(-1) = 0。于是，就得到了如 算法一.14 所示的线性递归算法：

```
算法: LinearFibonacci(k)
输入: 非负整数k
输出: 相邻的一对Fibonacci数(Fib(k), Fib(k-1))
{
  if (k<=1)
    return (k, 0);
  else {
    (i, j) = LinearFibonacci(k-1);

    return (i+j, i);

  }
}
```

算法一.14 通过线性递归计算Fibonacci数

采用这一算法计算 Fib(n)，每次递归调用后参数 n 都会减一，因此即使记入最开始的那次调用，该方法总共只也不过被调用 n 次。因此，该算法的时间复杂度为 $O(n)$，较之二分递归算法的 $O(2^n)$ 有天壤之别（建议读者通过实验体验二者之间的巨大差别）。

比较 Fibonacci 数的这两种算法可以看出，为了确保二分递归算法的效率，必须保证分解出来的每对子问题之间是相互独立的，即它们各自的计算没有重复和冗余；即使不能彻底做到这样，也应尽可能减少不必要的重复计算。

减少重复计算的一种直接而常用技巧，就是把子问题的计算结果记录下来，此后若再次遇到相同的子问题，就可以根据记录直接获得结果，而不必重新计算——实际上，这也就是动态规划（Dynamic programming）的核心思想。

1.5.4 多分支递归

有的时候，一个问题可能需要分解为不止两个子问题，此时就要采用多分支递归（Multiple recursion）。这类递归的一类典型运用，就是在求解组合游戏问题时，枚举各种可能的排列。比如下面就是一些典型的和式游戏（Summation puzzle）：

pot + pan = bib

dog + cat = pig

tom + jerry = jacky

tommy + jerry = happy

boy + girl = baby

这类游戏的要求，就是给其中的字母赋予互异的数字（即 0, 1, …, 9），使得和式能够成立[1]。一种直观的方法，就是不断尝试各种可能的赋值（对应于字母的排列），从中找出答案。

当可能的排列总数不是很大时，可以借助计算机枚举出所有可能的排列，进而通过检测找出答案。算法一.15 所示的，就是一个这样的算法。这是一个通用的算法，它可以枚举出由字母表U中各字母构成的所有无重复排列。不同的实际问题，只需编写出相应的Test方法即可。

```
算法：PuzzleSolve(k, S, U)
输入：  非负整数k——表示尚未启用的字母个数
        序列S——由已经启用的字母构成

        集合U——由尚未启用的字母组成

输出：枚举出由U中字母所能组合出的所有序列
{
  for (all e∈U) { //依次尝试各个字母
    U = U\{e}; //启用字母e
    S = S + e; //将e接到序列S的尾部
    k--; //尚未启用的字母个数减一
    if (k==0) { //若所有字母都已启用，则
      if (true==Test(S)) //检查S是否为一个可行解。若是，则
        return "Solution found:" + S; //报告结果
    } else //否则
      PuzzleSolve(k, S, U); //继续递归
    //至此，已经生成并检查过一个全排列，故需回溯。为此，
    S = S - e; //将末尾的字母删除，
    U = U∪{e}; //将它放回至集合U中，并
    k++; //尚未启用的字母个数加一
  }
}
```

算法一.15 利用多分支递归解答组合游戏：枚举出所有可能的组合，逐一测试

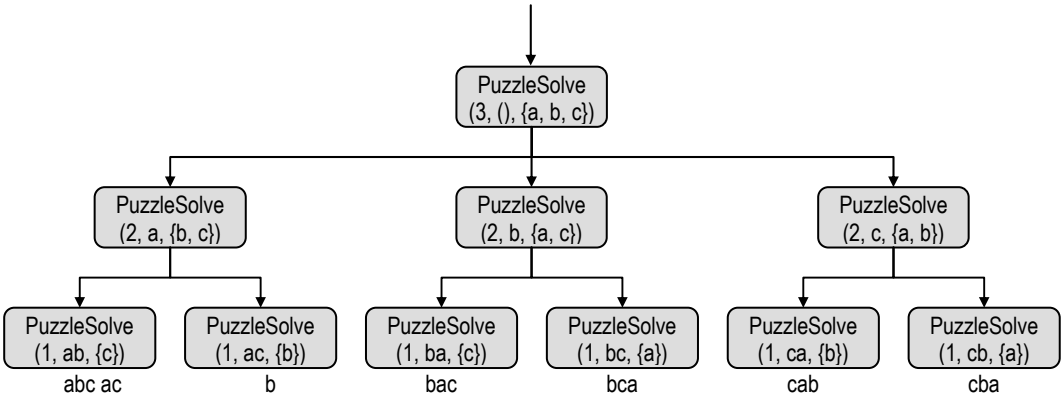
这一通用算法的思路是：为了生成长度为 k 的所有排列，只需

1. 递归地生成长度为 k-1 的所有排列；然后
2. 将某一尚未启用的字母添加到尾部。

为了避免字母被重复启用，算法使用集合 U 来动态地记录当前尚未启用的字母。以上面的和式游戏为例，初始的 U = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}。

[1] 比如，上面最后那个和式的一个解就是：a = 4，b = 9，g = 8，i = 5，l = 0，o = 3，r = 6，y = 2。

如 图一.7 所示，就是执行PuzzleSolve(3, "", {a, b, c})后的递归跟踪。从中可以看出，由这三个字母构成的所有排列是如何被枚举出来的。请注意，由于这里共有三个字母，所以初始的方法实例会做三次递归调用，下一层的每一方法实例会做两次递归调用，再下一层的每一方法实例会做一次递归调用。一般地，对于规模为m的字母表U，PuzzleSolve(3, "", U)将会生成m×(m-1)×(m-2)×…×2×1 = m!个全排列——这正是我们所需要的。



图一.7 运行PuzzleSolve(3, "", {a, b, c})后的递归跟踪，生成的排列标在对应方框的下面

第二章

栈与队列

在各式各样的数据结构中，栈与队列也许是最简单、最基本的，但它们绝对是最重要的。这两种基本数据结构的应用非常广泛，也是复杂数据结构的基础。比如，如今计算机的 CPU，往往会在内部用硬件微指令形式直接实现若干种数据结构，而栈与队列则是其中最常见。此外，现代计算环境的很多重要特性，也都建立在这些数据结构之上，Java 的实时运行环境——Java 虚拟机（JVM, Java Virtual Machine）——就是这方面的一个典型例子。

再如，Java 类集框架（Java Collections Framework）为我们提供了一个专门针对栈的内建类，同时还支持对栈与队列的各种操作，由此足见这两种数据结构的重要地位。不过，我们在此不仅要学习如何使用 Java 内部提供的这些数据结构，更重要的是学会独立编写数据结构。为此，我们将以栈与队列为例，介绍如何“从头开始”设计简单的数据结构。这里所要介绍的一些原则，也同样适用于设计更为复杂的、Java 类集框架没有提供的数据结构。

在本章中，我们首先按照一般的形式给出栈与队列的抽象数据类型定义，然后介绍它们的两种实现方法：数组及链表。数组实现采用“基于下标访问”（Index-based access）的模式，而链表实现则是基于“节点”（Node）或“位置”（Position）的概念。我们将看到，无论是哪种实现方式，栈与队列的每一基本操作都可以在常数时间内完成。

为了讲解如何使用栈与队列，我们将具体介绍它们的几个应用实例，包括它们在 Java 虚拟机模型中的应用。我们还将介绍栈与队列的一种推广——双端队列——及其利用双向链表的实现方法。

§2.1 栈

栈是存放对象的一种特殊容器，在插入与删除对象时，这种结构遵循后进先出（Last-in-first-out, LIFO）的原则——也就是说，对象可以任意插入栈中，但每次取出的都是此前插入的最后一个对象。比如一摞椅子（如 图二.1 所示），只能将最顶端的椅子移出，也只能将新椅子放到最顶端——这两种操作分别称作入栈（Push）和退栈（Pop）。

图二.1 由椅子构成的栈

栈是最基本的数据结构之一，在实际应用中几乎无所不在。例如，网络浏览器会将用户最近访问过的地址组织为一个栈：用户每访问一个新页面，其地址就会被存放至栈顶；而用户每次按下“Back”按钮，最后一个被记录下的地址就会被清除掉。再如，当今主流的文本编辑器大都支持编辑操作的历史记录功能：用户的编辑操作会被依次记录在一个栈中；一旦出现误操作，用户只需按下“Undo”按钮，即可撤销最近一次操作并回到此前的编辑状态。

2.1.1 栈 ADT

作为一种抽象数据类型，栈必须支持以下方法：

表二.1 栈ADT支持的操作

操作方法	功能描述
push(x)：	将对象 x 压至栈顶 输入：一个对象 输出：无
pop()：	若栈非空，则将栈顶对象移除，并将其返回 否则，报错 输入：无 输出：对象

此外，还可以定义如下的方法：

表二.2 栈ADT支持的其它操作

操作方法	功能描述
getSize()：	返回栈内当前对象的数目 输入：无 输出：非负整数
isEmpty()：	检查栈是否为空 输入：无 输出：布尔标志
top()：	若栈非空，则返回栈顶对象（但并不移除） 否则，报错 输入：无 输出：栈顶对象

表二.3 给出了从一个空栈开始，在依次执行一系列操作的过程中，栈中内容的相应变化。

表二.3 栈操作实例

操作方法	输出	栈结构组成（右侧为栈顶）
push(5) -		(5)
push(3) -		(5, 3)
pop() 3		(5)
push(7) -		(5, 7)
pop() 7		(5)
top() 5		(5)
pop() 5		()
pop()	空栈错	()
isEmpty() true		()
push(9) -		(9)
push(7) -		(9, 7)

操作方法	输出	栈结构组成（右侧为栈顶）
push(3)	-	(9, 7, 3)
push(5)	-	(9, 7, 3, 5)
getSize()	4	(9, 7, 3, 5)
pop()	5	(9, 7, 3)
push(8)	-	(9, 7, 3, 8)
pop()	8	(9, 7, 3)
pop() 3		(9, 7)

■ St ack 接口

由于其重要性，在 Java 的 java.util 包中已经专门为栈结构内建了一个类——java.util.Stack。任何 Java 对象都可以作为该内建类的栈元素，同时该类还提供了多种方法：push()、pop()、peek()（功能等价于 top()）、getSize()以及 empty()（功能等价于 isEmpty()）。在遇到空栈时，方法 pop()和 peek()都会报意外错 ExceptionStackEmpty。这个内建类使用起来很方便，不过，为了达到学习的效果，我们还是需要对其中若干最基本的问题进行讨论，并亲自动手来设计并实现栈结构。

Java抽象数据类型的实现过程，通常可以分为两步。首先，要给出其应用程序接口定义（Application programming interface, API），简称接口（Interface）。接口的作用，就是明确ADT所支持方法的名称、声明及调用的形式。此外，还要针对各种可能出现的错误条件，定义相应的意外。例如 代码二.1 中定义的ExceptionStackEmpty的意外——在对空栈应用pop()或top()方法时，这一意外就会被抛出，以报告错误。

```
/*
 * 当试图对空栈应用pop或top方法时，本意外将被抛出
 */

package dsa;

public class ExceptionStackEmpty extends RuntimeException {
  public ExceptionStackEmpty(String err) {
    super(err);
  }
}
```

代码二.1 在试图对空栈应用pop或top方法时将被抛出的意外

以下 代码二.2 给出了栈ADT的完整Java接口。可以看出，这个接口非常通用，按照该接口的定义，属于任何类（哪怕是异构类）的对象，都可以作为栈的成员。

```
/*
 * 栈接口
 */

package dsa;
```

```
public interface Stack {

  public int getSize();//返回栈中元素数目

  public boolean isEmpty();//判断栈是否为空
  public Object top() throws ExceptionStackEmpty;//取栈顶元素（但不删除）
  public void push (Object ele);//入栈
  public Object pop() throws ExceptionStackEmpty;//出栈
}
```

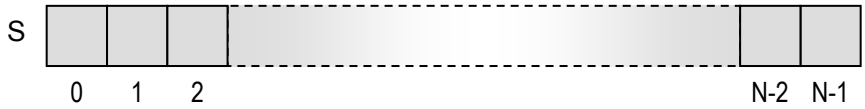
代码二.2 Stack 接口

要使栈 ADT 真正有用，还需要定义出一个具体的类，来实现与该 ADT 接口中定义的各个方法。接下来，我们就给出 Stack 接口的一个简单实现。

2.1.2 基于数组的简单实现

■ 思路

为了实现栈接口，我们可以用一个数组来存放其中的元素。具体来说，就是使用一个容量为 N 的数组 S，再加上一个变量 top 来只是当前栈顶的位置。



图二.2 利用数组实现栈

由于Java数组的元素都是从 0 开始编号，所以top必须初始化为-1；反过来，只要top = -1 ，就说明栈为空。另一方面，由于数组的容量固定，因此有可能会出现栈溢出的情况。为此，我们专门引入一个新的意外ExceptionStackFull（如 代码二.3 所示），以指示这类错误。需要注意的是，这一例外并非栈ADT本身的要求，而只是针对数组实现而设置的。

```
/*
 * 当试图对满栈应用push方法时，本意外将被抛出
 */

package dsa;

public class ExceptionStackFull extends RuntimeException {
  public ExceptionStackFull(String err) {
    super(err);
  }
}
```

代码二.3 在试图对满栈应用push方法时将被抛出的意外

■ Jav a 实现

基于数组的栈实现细节，如 代码二.4 所示。

```
/*
 * 借助定长数组实现Stack接口
 */

package dsa;

public class Stack_Array implements Stack {
  public static final int CAPACITY = 1024;//数组的默认容量
  protected int capacity;//数组的实际容量
  protected Object[] S;//对象数组
  protected int top = -1;//栈顶元素的位置

//按默认容量创建栈对象
  public Stack_Array()
  { this(CAPACITY); }

//按指定容量创建栈对象
  public Stack_Array(int cap) {
    capacity = cap;
    S = new Object[capacity];
  }

//获取栈当前的规模
  public int getSize()
  { return (top + 1); }

//测试栈是否为空
  public boolean isEmpty()
  { return (top < 0); }

//入栈
  public void push(Object obj) throws ExceptionStackFull {
    if (getSize() == capacity)
      throw new ExceptionStackFull("意外: 栈溢出");
    S[++top] = obj;
  }

//取栈顶元素
  public Object top() throws ExceptionStackEmpty {
    if (isEmpty())
      throw new ExceptionStackEmpty("意外: 栈空");
    return S[top];
  }

//出栈
```

```
  public Object pop() throws ExceptionStackEmpty {

    Object elem;

    if (isEmpty())
      throw new ExceptionStackEmpty("意外：栈空");
    elem = S[top];
    S[top--] = null;
    return elem;
  }
}
```

代码二.4 借助一个容量为N的数组实现栈

■ 正确性分析

根据各方法的定义，上述基于数组栈实现的正确性显而易见。不过，其中 pop()方法的实现还是值得推敲的。

你可能注意到，在弹出栈顶之后，将原栈顶 S[top]置为 null 的操作似乎是多余的——即使省略这一步，该方法依然符合 ADT 定义的要求。不过就 Java 语言而论，在实现该方法时若果真准备省略这一步，我们就必须对利弊做一权衡。这涉及到 Java 的内存垃圾回收机制（Garbage collection mechanism）——Java 会自动监测存放各对象的内存区域，一旦发现某个对象不再被任何活跃的对象引用，就会回收其占用的空间。假设原栈顶为 e = S[top]，在应用 pop()方法时将 S[top]置为 null，实际上就是告诉系统：该栈元素不再保留指向对象 e 的一个引用。于是，只要不再有任何活跃的引用指向 e，该对象占用的内存空间就会被当作垃圾被回收。

■ 复杂度分析

若基于数组实现栈，各方法的时间复杂度可以归纳为 表二.4。事实上，其中每一方法都只需执行常数次算术运算、比较和赋值。虽然pop()和top()都会调用一次isEmpty()，但后者也只需常数时间。因此，如此实现的Stack ADT，所有方法都只需运行常数时间。若数组的容量为N，则空间复杂度为$\mathcal{O}(N)$——无论实际使用了多少单元，这个复杂度在创建栈时就已将注定了。

表二.4 基于数组实现的栈，各方法的时间复杂度

操作方法	时间复杂度
getSize()	$\mathcal{O}(1)$
isEmpty()	$\mathcal{O}(1)$
top()	$\mathcal{O}(1)$
push()	$\mathcal{O}(1)$
pop()	$\mathcal{O}(1)$

请注意，这里使用了一个常量 CAPACITY 来指定数组的默认容量。在编译之前，可以根据实际需要修改 CAPACITY，即可调整数组的容量。

■ 缺陷

基于数组的 Stack 接口实现简单而有效，因而被广泛应用于很多场合。然而，这一实现的不足也十分明显：其内部数组的容量是事先固定的。一方面，很多的时候需要大容量的栈，事先确定的容量难以保证足够大，因此在实际运行中很可能会造成栈溢出；反过来，在一些应用问题中，小容量的栈足以满足要求，此时固定的容量又会造成存储空间的浪费。因此，有必要对这一实现进行改进。

在本章稍后，我们将介绍栈的其它实现，这些实现可以根据运行过程中的实际需要，动态地调整栈的规模。当然，如果能够在事先估计出栈实际所需的空间量，用数组实现也未尝不可。

■ 强制转换

上面，我们通过定义Stack接口并进而以Java类形式实现这一接口，完成了对栈ADT的实现。这一实现形式的一个突出优点在于，它使得我们能够用栈来存放来自于不同类的通用对象（参见 代码二.2 和 代码二.4），无论是Integer对象、Student对象还是Planet对象，Stack_Array都可以存放。

鉴于通用栈的成员都被 Java 统一地视作 Object 类的对象，为了保证数据的一致性，我们在使用这种栈时需要格外仔细。当然，插入操作不会有任何问题——因为 Java 的每个类都是 Object 类的子类。然而反过来，在从栈中取出对象（执行 top()或 pop()操作）时，无论该对象具体属于哪个类，我们得到的却总是一个 Object 对象的引用。如果对被取出对象实施的操作与其具体所属的类有关，就必须将取出的对象从通用类强制转换为特定的类。

■ 借助栈进行数组倒置

根据后进先出的原则，在任意时刻，自栈顶到栈底，栈数组中的元素必然按照入栈次序逆序排列。利用这一特性，可以逆向访问数组的成员。由此，我们可以得到另一非递归算法，以解决第 1.5.1 节中的数组倒置问题。算法的思想非常简单：将数组的所有元素依次压入栈中，然后不断退栈，并将弹出的元素依次填入数组。该算法的Java实现如 代码二.5 所示。

```
public static Integer[] reverse(Integer[] a) {
  Stack_Array S = new Stack_Array(a.length);
  Integer[] b = new Integer[a.length];
  for (int i=0; i<a.length; i++)    S.push(a[i]); //所有元素顺序入栈
  for (int i=0; i<a.length; i++)    b[i] = (Integer) (S.pop());  //逆序退栈
  return b;
}
```

代码二.5 利用栈实现对数组的倒置

正如此前所讨论的，该算法中从栈中弹出的每一元素都是属于 Object 类，因此在将它们填入原数组之前，需要将其强制转换为 Integer 类。

栈的应用十分广泛，它在 代码二.5 中的应用只不过是一个具体而微的例子。实际上，在Java语言本身的实现中，栈这一数据结构也扮演着重要的角色。

2.1.3 Java 虚拟机中的栈

Java 的每一个程序都要被编译为一个二进制指令序列，这些指令可以在一个特定的计算模型——Java 虚拟机（Java Virtual Machine, JVM）——上执行。就 Java 语言自身的定义而言，JVM 起着关键性的作用。正是由于可以将 Java 源程序编译为 JVM 的可执行代码，而不是在某一特定 CPU 上直接支持的可执行代码，才使得 Java 程序可以运行于任何计算机——只要该计算机上配有一个 JVM 解释器。十分有趣的是，对于 JVM 的定义而言，栈这一数据结构也是至关重要的。

■ Java 方法栈

任一运行中的Java程序（更准确地说，应该是运行中的Java线程）都会配备一个私有的栈，称作Java方法栈（Java method stack）或简称Java栈（Java stack），用来记录各个方法在被调用过程中的局部变量等重要信息（参见 图二.3）。

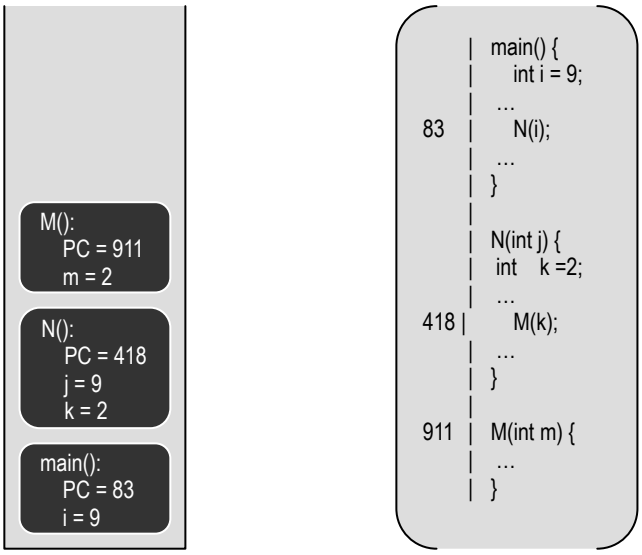
具体来说，在 Java 程序的执行期间，JVM 会维护一个栈，其中的元素分别是当前活跃的某个方法实例的描述符，称作帧（Frame）。比如，若方法 N 调用了方法 M，则在 M 的这个实例所对应的帧中，记录了该实例的调用参数以及其中的局部变量，还有关于 N 的信息，以及在 M 结束时应该返回给 N 的东西。

JVM 还设置了一个称作程序计数器（Program counter）的变量 PC，负责记录程序在 JVM 中运行到的当前位置。当方法 N 要调用方法 M 时，程序计数器当前的数值就会被存放在 N 的实例所对应的帧中——这样，待 M 执行完毕后，JVM 才能知道应该返回到什么位置并继续执行下去。Java 栈中最顶层的帧，总是对应于当前正在执行的方法实例。其余的各帧，分别对应于某个被挂起、尚未执行完的方法。居于栈底的那一帧对应于 main 方法，其余各帧自栈底向上按照被调用的次序顺序排列。每当有一个方法被调用，其对应的帧就会被压入栈中；一旦当前方法实例运行终止，程序计数器就会恢复为该帧中先前保存的位置，然后该帧出栈，控制权转交给新栈顶所对应的方法实例。

JVM 还会通过 Java 栈将参数传递给被调用的方法。具体来说，Java 是按照“值传递”（Call-by-value）的方式传递参数的。也就是说，以参数形式传递给被调用方法的，总是变量或表达式当前的实际值。

对于属于基本类型（比如int或float）的变量x，x的当前值就是其算数值。假设方法N调用方法M时，如果x作为参数传递给M，那么x的当前值就会被复制到M所对应帧中的某个局部变量（参见 图二.3）。需要指出的是，此后即使方法M修改了该局部变量的数值，x在N中的数值也决不会受到影响。

不过，倘若 x 是指向某对象 o 的一个引用，那么 x 的当前值就是对象 o 的内存地址。因此，如果将 x 作为参数传递给方法 N，实际上传递的是对象 o 的内存地址。这个地址被复制给 N 的某个局部变量 y 之后，y 也将和 x 一样地指向对象 o。



图二.3 Java方法栈的实例：方法main调用方法N，方法N再调用方法M

因此，一旦方法 M 通过引用 y 改变了对象 o 的状态，也就是修改了该引用所指的对象。同理，即使方法 M 将 y 指向其它对象，x 依然还是指向对象 o。

JVM 正是这样地借助方法栈来调用方法并传递参数。无独有偶，方法栈并非 Java 首次采用，实际上，多数现代程序语言（C、C++等）都采用了这一机制。

■ 递归的实现

借助栈来实现方法调用的优点之一，就是支持程序的递归——正如第 §1.5 节所介绍的那样，Java允许方法直接或间接地自我调用。鉴于递归程序的简洁性和高效性，现代程序设计语言几乎都采用了这一模式。有趣的是，尽管诸如Cobol和Fortran之类的早期程序语言一开始并没有采用栈来实现过程调用，但在它们后来的新版本中，也引入了栈结构来实现其过程调用。

如果对递归程序的运行进行跟踪，那么每一个方框都分别对应于 Java 方法栈中的某一帧。而且，在任一时刻，Java 方法栈中的所有帧，自底向上依次对应于从方法 main 到当前活跃方法的一条调用路径。

为了更好地说明Java运行栈支持递归方法的机制，我们来考察另一经典的递归函数：阶乘Fac(n) = n!。该函数的Java实现如 代码二.6 所示：

```
/*
 * 阶乘函数
 */
public static long factorial(1ong n) {
  if (n <= 1)   return 1;
  else          return n*factorial(n-1);
}
```

代码二.6 阶乘函数的递归实现

首次调用方法 factorial()时，对应的帧中设有一个局部变量存放 n。接下来，该方法通过递归调用自己以计算出(n-1)!，相应地，新的一帧会被压入方法栈中。类似地，该方法会继续递归调用自己，以计算出(n-2)!。这一过程不断进行下去，在经过 n 次这样的递归调用之后，最后一次调用将是 factorial(1)，此时遇到递归基，故该方法不再继续递归下去，而是返回具体的数值 1。请注意，在同一运行栈中，同一方法 factorial()可能会对应于很多帧（准确地说，至多不超过 n 帧）。每一帧都保存了各自的参数 n 以及返回值。最后，首次被调用的那个方法将执行完毕，并返回(n-1)!；调用它的方法将把这一返回值乘上 n 以得到 n!，然后返回给用户。

■ 操作数栈

实际上，JVM 对栈的应用并不仅限于方法栈。在对算术表达式（如((a+b)*(c-d))/e 之类）进行求值时，JVM 也使用了另一个栈——操作数栈（Operand st ack）。以“a+b”之类的简单二元操作为例，为了计算其值，首先将 a 和 b 依次压入栈中，然后执行一条专门的指令——该指令要求将栈顶的两个元素弹出，对它们实施加法，并将结果重新压入栈中。

关于操作数栈，这里不准备深入讨论，我们只需了解，对于 Java 虚拟机而言，操作数栈和 Java 方法栈都是不可或缺的重要组成部分。

2.1.4 栈应用实例

接下来我们讨论栈结构的两个具体应用：表达式中的括号匹配以及 HTML 文件中的标志匹配。

■ 括号匹配算法

算术表达式中可能出现很多对相互匹配的符号，比如：

- 小括号："("和")"
- 大括号："{"和"}"
- 方括号："["和"]"
- 下取整括号："⌊"和"⌋"
- 上取整括号："⌈"和"⌉"

所谓括号匹配，就是要求上述左、右括号成对匹配，而且其层次嵌套关系也必须合理。比如：

- ()({}){([()[]])}：匹配
- ([]{)：不匹配（右大括号缺失）
-)([()]{}：不匹配（第一对括号左右颠倒）
- ([())]{}：不匹配（括号虽然能够两两成对，但嵌套关系紊乱）

匹配关系不难理解，我们将其严格的定义作为作业留给读者。

下面，我们将借助一个栈结构 S，通过对算术表达式自左向右的一遍扫描，检查其中的括号是否匹配。

假设算术表达式为 X = "$x_0x_1x_2…x_{n-1}$"，其中 x_i 可以是括号、常数、变量名或者算术运算符。我们依次检查 X 中的各个符号，非括号的符号都可以忽略。若遇到左括号，则将其压入栈 S 中；若遇到右括号，则将栈顶符号弹出并与该右括号对比。如果发现某对括号不匹配，或者遇到右括号时栈

为空，或者整个表达式扫描过后栈非空，都可以断定括号不匹配。在按照以上规则扫描完所有字符后，若栈为空，则说明括号是匹配的。如果按照前面对栈的实现，每一 push()和 pop()操作都只需常数时间，因此对于长度为 n 的算术表达式，上述算法需要运行 $\mathcal{O}(n)$的时间。

该算法的伪代码描述如 算法二.1 所示：

```
算法：ParenMatch(X, n)
输入：长度为n数组X，对应于某一算术表达式
输出：判断X中的括号是否匹配
{
  初始化栈S;
  for (i=0 to n-1) //依次扫描X中的各个符号
    if (X[i]为左括号) //若遇到左括号，则
      S.push(X[i]); //入栈
    else if (X[i]为右括号) { //若遇到右括号，则
      if (S.isEmpty()) //若此时栈为空，则
        return false; //不用检查后续字符，即可报告“不匹配”
      if (S.pop()与X[i]不匹配) //否则，若弹出的栈顶与当前括号不匹配，则
        return false; //报告“不匹配”
    }
  //至此，整个表达式已经扫描完毕
  if (S.isEmpty())  return true; //若此时栈空，则报告“匹配”
  else              return false; //否则，报告“不匹配”
}
```

算法二.1　算术表达式的括号匹配算法

■ HTML　文档的标志匹配

HTML 是 Internet 超文本的标准格式。按照规范，HTML 文档由不同的标志（Tag）划分为不同的部分与层次。与括号类似，这些标志也需成对出现，对于名为<myTag>的起始标志，相应的结束标志为</myTag>。常用的 HTML 标志有：

- <bod　y>和</body>：文档体
- <h1>　和</h1>：节的头部
- 　和 ：居中对齐
- <lef　t>和</left>：左对齐
- 　和 ：段落
- 　和：编号列表
- 　和：列表项
- ……

尽管很多浏览器有很强的容错、纠错能力，但最好还是按照 HTML 的规范保证标记的匹配。

```
<body>
 <left>
<h1>数据结构与算法</h1>
 <p>
  程序=数据结构+算法
 </p>
 <ol>
  <li>线性结构</li>
  <li>半线性结构</li>
  <li>非线性结构</li>
 </ol>
 </left>
</body>
```

数据结构与算法

程序=数据结构+算法

1. 线性结构
2. 半线性结构
3. 非线性结构

图二.4 HTML 标志示例：HTML文档（左）及其浏览效果（右）

其实，只要对 算法二.1 稍作修改，即可用来对HTML文档进行标志匹配。代码二.7 给出了该算法的完整Java实现。

```
/*
 * HTML文档标志匹配算法
 */
import java.util.StringTokenizer;
import dsa.Stack;
import dsa.Stack_List;
import java.io.*;

//检查HTML文档的括号匹配
public class HTML {

//嵌套类，存放HTML标志
  public static class Tag {
    String name;//标志名
    boolean opening;//起始标志
    public Tag() {//默认构造方法
      name = "";
      opening = false;
    }
    public Tag(String nm, boolean type) {//构造方法
      name = nm;
      opening = type;
    }
    public boolean isOpening() { return opening; }//判断是否起始标志
    public String getName() { return name; }//返回标志名称
  }//class TAG

//通过缩格显示匹配标志的层次
  private void indent(int level) {
    for (int k=0; k<level; k++) System.out.print("\t|");
  }
```

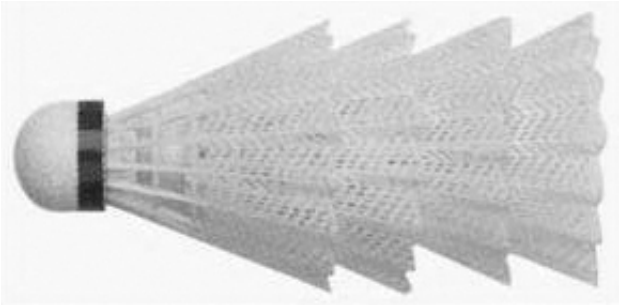
```
//检查每个起始标志是否都对应于一个结束标志

  public boolean isHTMLMatched(Tag[] tag) {

    int level = 0;//标志的层次
    Stack S = new Stack_List();//存放标志的栈
    for (int i=0; (i<tag.length) && (tag[i] != null); i++) {//逐一检查各标志
      if (tag[i].isOpening()) {//若遇到起始标志，则
        S.push(tag[i].getName());//压之入栈
        indent(level++);  System.out.println("\t┌"+tag[i].getName());
      } else {//否则，即当前标志为结束标志，故
        if (S.isEmpty())//若此时栈空，则
          return false;//报告"不匹配"
        if (!((String) S.pop()).equals(tag[i].getName()))
                                  //否则，若当前标志与弹出的标志不匹配，则
          return false;//报告"不匹配"
        indent(--level);  System.out.println("\t└"+tag[i].getName());
      }//else
    }//for
  //至此，已扫描完整个文档
    if (S.isEmpty())//若此时栈为空，则
      return true;//报告"匹配"
    else//否则
      return false;//报告"不匹配"
  }//isHTMLMatched

  public final static int CAPACITY = 1000;//数组的最大容量
//从HTML文档中提取标志，依次存入数组
  public Tag[] parseHTML(BufferedReader r) throws IOException {
    String line;//文档中的一行
    boolean inTag = false;//标志：当前是否扫描到标志
    Tag[] tag = new Tag[CAPACITY];//存放标志的数组
    int count = 0;//标志的计数器
    while ((line = r.readLine()) != null) {//依次读入文档的各行
      StringTokenizer st = new StringTokenizer(line,"<> \t",true);//标志的特征为尖括号
      while (st.hasMoreTokens()) {
        String token = (String) st.nextToken();
        if (token.equals("<"))//若扫描到'<'，说明遇到了下一标志，故
          inTag = true;//将当前状态设为"正在扫描标志"
        else if (token.equals(">"))//若扫描到'>'，说明上一标志扫描完毕，故
          inTag = false;//将当前状态设为"处于标志之外"
        else if (inTag) {       //若正在扫描一个标志
          if ( (token.length() == 0) || (token.charAt(0) != '/') )
                                    //若是起始标志，则
```

```
            tag[count++] = new Tag(token, true);//加入之
          else//否则
            tag[count++] = new Tag(token.substring(1), false);

                                          //加入一个结束标志（跳过首字符'/'）

        }//请注意：所有非标志部分均被忽略了
      }//while
    }//while
    return tag;//返回标志数组
  }//parseHTML

//测试用main方法
  public static void main(String[] args) throws IOException {
    BufferedReader stdr = new BufferedReader(new InputStreamReader(System.in));//标准
输入
    HTML tagChecker = new HTML();
    if (tagChecker.isHTMLMatched(tagChecker.parseHTML(stdr)))
      System.out.println("该文件符合HTML的标志匹配");
    else
      System.out.println("该文件不符合HTML的标志匹配");
  }
}
```

代码二.7 HTML 文档标志匹配算法的完整Java实现

§2.2 队列

与栈一样，队列也是最基本的数据结构之一。队列也是对象的一种容器，其中对象的插入和删除遵循“先进先出”（First-In-First-Out, FIFO）的原则——也就是说，每次删除的只能是最先插入的对象。因此，我们可以想象成将对象追加在队列的后端，而从其前端摘除对象。就这一性质而言，队列与栈堪称“孪生兄弟”。比如，在银行等待接受服务时，顾客们就会排成一个队列——最先到达者优先得到服务。再如，用球桶来乘放羽毛球时，所有的球也排成一个队列——你总是从一端将球取出，而从另一端把球放入。



图二.5 成筒的羽毛球就是队列的一个模型：球总是从一端取出，从另一端插入

2.2.1 队列 ADT

队列的抽象数据类型就是一个容器，其中的对象排成一个序列，我们只能访问和取出排在最前端（Front）的对象，只能在队列的尾部（Rear）插入新对象。正是按照这一规则，才能保证最先被插入的对象首先被删除（FIFO）。

队列 ADT 首先支持下面的两个基本方法：

表二.5 队列ADT支持的操作

操作方法	功能描述
enqueue(x)：	将元素 x 加到队列末端 输入：一个对象 输出：无
dequeue()：	若队列非空，则将队首元素移除，并将其返回 否则，报错 输入：无 输出：对象

此外，与栈 ADT 类似地，队列 ADT 还支持如下的方法：

表二.6 队列ADT支持的其它操作

操作方法	功能描述
getSize()：	返回队列中当前包含的元素数目 输入：无 输出：非负整数
isEmpty()：	检查队列是否为空 输入：无 输出：布尔标志
front()：	若队列非空，则返回队首元素（但并不移除） 否则，报错 输入：无 输出：队头对象

表二.7 给出了从一个空队列开始，在依次执行一系列操作的过程中，队列中内容的相应变化。

表二.7 队列操作实例

操作	输出	队列（左侧为队头）
enqueue(9) -		{9}
enqueue(3) -		{9, 3}
dequeue() 9		{3}
enqueue(5) -		{3, 5}
dequeue() 3		{5}
front() 5		{5}
dequeue() 5		{}
dequeue()	空队错	{}
isEmpty() true		{}
enqueue(8) -		{8}
enqueue(4) -		{8, 4}
getSize() 2		{8, 4}
enqueue(11)	-	{8, 4, 11}
enqueue(9)	-	{8, 4, 11, 9}
dequeue() 8		{4, 11, 9}

队列的应用十分广泛，无论是商店、剧院、机场还是银行、医院，凡是按照“先到的客户优先接受服务”原则的场合，都可以利用队列这一数据结构来编排相应的服务。

■ Queue 接口

代码二.8 给出了队列ADT的一个Java接口。这是一个通用的接口，来自任何类的对象，都可以作为队列的成员。因此与栈一样，在从队列中取出一个对象后，也需要对其进行强制转换。

```
/*
 * 队列接口
 */

package dsa;

public interface Queue {
  public int getSize();//返回队列中元素数目
  public boolean isEmpty();//判断队列是否为空
  public Object front()//取队首元素（但不删除）
  throws ExceptionQueueEmpty;
  public void enqueue (Object obj)
  throws ExceptionQueueFull;//入队
  public Object dequeue()//出队
  throws ExceptionQueueEmpty;
```

```
  public void Traversal();//遍历

}
```

代码二.8 Queue 接口

可以看出，这里的 getSize()和 isEmpty()方法与栈的对应方法完全相同。这两个方法加上 front()方法都属于所谓的访问式方法（Accessor method ）——这类方法仅仅读取队列的信息，但不会修改队列的内容。

2.2.2 基于数组的实现

■ 顺序数组

借助一个定长数组 Q 来存放对象，即可简单地实现队列。那么，为了符合 FIFO 准则，应该如何表示和记录队列中各对象的次序呢？

一种自然的办法就是仿照栈的实现，以 Q[0]作为队首，其它对象顺序往后存放。然而如此一来，每次首元素出队之后，都需要将后续的所有元素向前顺移一个单元——若队长为 n，这项工作需要 $\mathcal{O}(n)$时间，因此效率很低。

■ 循环数组

为了避免数组的整体移动，可以引入如下两个变量 f 和 r：

- f：始终等于 Q 的首元素在数组中的下标，即指向下次出队元素的位置
- r：始终等于 Q 的末元素的下标加一，即指向下次入队元素的位置

一开始，f = r = 0，此时队空。每次有对象入队时，将其存放于 Q[r]，然后 r 加一，以指向下一单元。对称地，每次有对象出队之后，也将 f 加一，指向新的队首元素。这样，对 front()、enqueue()和 dequeue()方法的每一次调用都只需常数时间。

然而，这还不够。细心的读者或许已经注意到，按照上述约定，在队列的生命期内，f 和 r 始终在单调增加。因此，若队列数组的容量为 N，则在经过 N 次入队操作后，r 所指向的单元必然超出数组的范围；在经过 N 次出队操作后，f 所指向的单元也会出现类似的问题。

解决上述问题的一种简便方法，就是在每次 f 或 r 加一后，都要以数组的长度做取模运算，以保证其所指单元的合法性。就其效果而言，这就相当于把数组的头和尾相联，构成一个环状结构。

基于上述构想，可以得到如 代码二.9 所示的算法：

```
/*
 * 借助定长循环数组实现Queue接口
 */

package dsa;
```

```
public class Queue_Array implements Queue {
  public static final int CAPACITY = 1000;//数组的默认容量
  protected int capacity;//数组的实际容量

  protected Object[] Q;//对象数组

  protected int f = 0;//队首元素的位置
  protected int r= 0;//队尾元素的位置

//构造方法（空队列）
  public Queue_Array()
  { this(CAPACITY); }

//按指定容量创建对象
  public Queue_Array(int cap)
  { capacity = cap; Q = new Object[capacity]; }

//查询当前队列的规模
  public int getSize()
  { return (capacity-f+r) % capacity; }

//判断队列是否为空
  public boolean isEmpty()
  { return (f == r);  }

//入队
  public void enqueue(Object obj) throws ExceptionQueueFull {
    if (getSize() == capacity-1)
      throw new ExceptionQueueFull("Queue overflow.");
    Q[r] = obj;
    r = (r+1) % capacity;
  }

//出队
  public Object dequeue() {
    Object elem;
    if (isEmpty())
      throw new ExceptionQueueEmpty("意外: 队列空");
    elem = Q[f];
    Q[f] = null;
    f = (f+1) % capacity;
    return elem;
  }

//取（并不删除）队首元素
  public Object front() throws ExceptionQueueEmpty {
    if (isEmpty())
      throw new ExceptionQueueEmpty("意外: 队列空");
    return Q[f];
```

```
  }

//遍历（不属于ADT）
  public void Traversal() {
    for (int i = f; i < r; i++)
      System.out.print(Q[i]+" ");

    System.out.println();

  }
}
```

代码二.9 借助循环数组实现队列

■ 队空与队满

请注意，这里采用的依然是定长数组，故与栈一样，有可能出现空间溢出的情况。一旦空间溢出，需要给出 ExceptionQueueFull 意外错。反过来，对于空队列的出队操作也需要禁止，这里将报 ExceptionQueueEmpty 意外错。那么，如何判断这两种情况呢？其中的一些细节值得讨论。

我们注意到，当队列中不含任何对象时，必有 f = r。然而，反之却不然。

试考虑如下情况：在数组中只剩下一个空闲单元（此时有 f ≡ (r+1) mod N）时，需要插入一个对象。若则按照上面的 enqueue()算法，插入后有 f = r，但事实上此时的队列已满。如果根据 “f = r” 判断队列为空，则尽管队列中含有元素，但出队操作却无法进行；反过来，尽管数组空间已满，却还能插入新元素（原有的元素将被覆盖掉）。

为了解决这一问题，一种简便易行的方法就是禁止队列的实际规模超过N-1。请注意在 代码二.9 中的enqueue()算法的实现：一旦队列规模将要超过这一范围，即报ExceptionQueueFull意外错。

■ 性能分析

不难看出，如上实现的队列，每一方法都只需要执行常数次算术运算、比较及赋值操作，因此它们的运行时间都是 $\mathcal{O}(1)$。

表二.8 基于数组实现的队列，各方法的时间复杂度

操作方法	时间复杂度
getSize()	$\mathcal{O}(1)$
isEmpty()	$\mathcal{O}(1)$
front()	$\mathcal{O}(1)$
enqueue()	$\mathcal{O}(1)$
dequeue()	$\mathcal{O}(1)$

与栈类似，这种实现的空间复杂度取决于数组的规模，即 $O(N)$。请注意，无论队列的实际规模有多大，这一复杂度都在事先就注定了，因此，在实际应用中通常效率很低——除非像下面这个例子那样，能够保证队列的规模基本不变。

2.2.3 队列应用实例

■ 循环分配器

队列结构很适于用来实现循环分配器：按照环形次序反复循环，为共享某一资源的一群客户（比如共享一个 CPU 的多个应用程序）做资源分配。

借助一个队列Q，就可以一个实现如 算法二.2 所示的循环分配器。

```
算法：RoundRobin
{
    e = Q.dequeue();
    Serve(e);
    Q.enqueue(e);
}
```

算法二.2 利用队列结构实现的循环分配器

■ Josephus 环

孩提时的你是否玩过“烫手山芋”游戏：一群小孩围成一圈，有一个刚出锅的山芋在他们之间传递。其中一个孩子负责数数，每数一次，拿着山芋的孩子就把山芋转交给右边的邻居。一旦数到某个特定的数，拿着山芋的孩子就必须退出，然后重新数数。如此不断，最后剩下的那个孩子就是幸运者。

通常，数数的规则总是从 1 开始，数到 k 时让拿着山芋的孩子出列，然后重新从 1 开始。Josephus 问题可以表述为：n 个孩子玩这个游戏，最后的幸运者是谁？

为了解答这个问题，我们可以利用队列结构来表示围成一圈的n个孩子。一开始，假定对应于队列首节点的那个孩子拿着山芋。然后，按照游戏的规则，把“土豆”向后传递到第k个孩子（交替进行k次dequeue()和k次enqueue()操作），并让她出队（dequeue()）。如此不断迭代，直到队长（getSize()）为 1。该算法的具体实现如 代码二.10 所示：

```
/*
 * Josephus环
 */
import dsa.*;
import java.io.*;

//模拟Josephus环
public class Josephus {
```

```
//利用队列结构模拟Josophus环
  public static Object Josephus(Queue Q, int k) {
    if (Q.isEmpty()) return null;
    while (Q.getSize() > 1) {//不断迭代
      Q.Traversal();//显示当前的环
      for (int i=0; i < k; i++)//将山芋向前传递k次

        Q.enqueue(Q.dequeue());

      Object e = Q.dequeue();//拿着山芋的孩子退出
      System.out.println("\n\t" + e + "退出");
    }
    return Q.dequeue();//最后剩下的那个孩子
  }

//将一组对象组织为一个队列
  public static Queue buildQueue(Object a[]) {
    Queue Q = new Queue_Array();
    for (int i=0; i<a.length; i++)
      Q.enqueue(a[i]);
    return Q;
  }

//测试用main方法
  public static void main(String[] args) {
    String[] kid = {"Alice",  "Bob",    "Cindy",  "Doug",   "Ed",
                    "Fred",   "Gene",   "Hope",   "Irene",  "Jack",
                    "Kim",    "Lance",  "Mike",   "Nancy",  "Ollie"};
    System.out.println("最终的幸运者是" + Josephus(buildQueue(kid), 5));
  }
}
```

代码二.10 利用队列结构模拟Josephus环

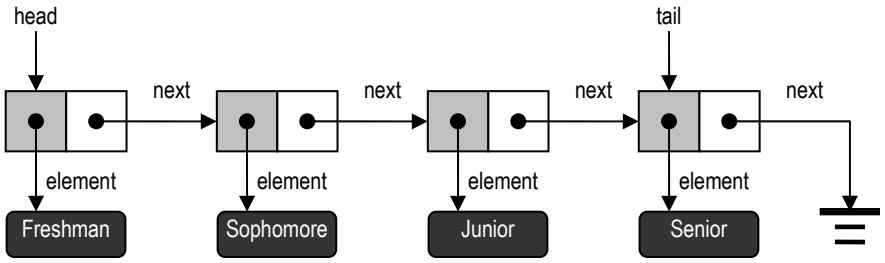
该算法每经过一次迭代，就有一个孩子出列，因此总共需要做 n-1 次迭代。每次迭代中，需要顺序访问 k 个孩子，因此总的时间复杂度为 $O(nk)$。

§2.3 链表

前面我们介绍了栈与队列的 ADT，并利用数组加以实现。遗憾的是，尽管这种实现简单明了，但由于数组长度必须固定，在空间效率及适应性方面还存在不足。本节将介绍一种基于链表的实现，以消除上述缺陷。

2.3.1 单链表

所谓链表（Linked list），就是按线性次序排列的一组数据节点。如 图二.6 所示，每个节点都是一个对象，它通过一个引用element指向对应的数据元素，同时还通过一个引用next指向下一节点。



图二.6 单链表结构

节点间这种相互指向的关系，乍看起来有循环引用之嫌，然而实际上却完全可行，而且不难实现。每个节点的 next 引用都相当于一个链接或指针，指向另一节点。借助于这些 next 引用，我们可以从一个节点移动至其它节点。链表的第一个和最后一个节点，分别称作链表的首节点（Head）和末节点（Tail）。末节点的特征是，其 next 引用为空。如此定义的链表，称作单链表（Singly linked list）。

与数组类似，单链表中的元素也具有一个线性次序——若 P 的 next 引用指向 S，则 P 就是 S 的直接前驱，而 S 是 P 的直接后继。与数组不同的是，单链表的长度不再固定，而是可以根据实际需要不断变化。如此一来，包含 n 个元素的单链表只需占用 $\mathcal{O}(n)$空间——这要比定长数组更为灵活。

■ Jav a 实现

为了实现Java的单链表结构，我们定义如 代码二.11 所示的Node类。

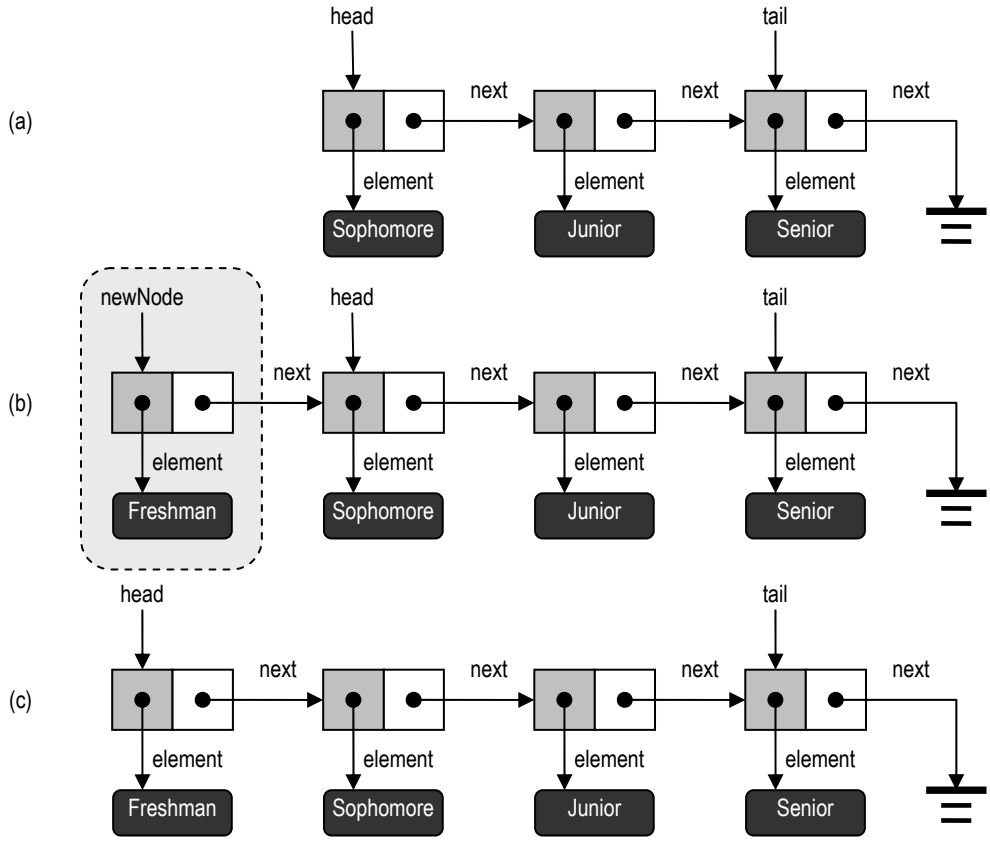
```
/*
 * 单链表节点类
 */

package dsa;

public class Node implements Position {
  private Object element;//数据对象
  private Node next;//指向后继节点

/***************************** 构造函数 *****************************/
  public Node()
  { this(null, null); }//指向数据对象、后继节点的引用都置空

  public Node(Object e, Node n)
  { element = e;  next = n; }//指定数据对象及后继节点

/***************************** Position接口方法 *****************************/
//返回存放于该位置的元素
```

```
  public Object getElem() { return element; }

//将给定元素存放至该位置，返回此前存放的元素
  public Object setElem(Object e)
  { Object oldElem = element; element = e;  return oldElem; }



/**************************** 单链表节点方法 ****************************/
//取当前节点的后继节点
  public Node getNext()
  { return next; }


//修改当前节点的后继节点
  public void setNext(Node newNext)
  { next = newNext; }
}
```

代码二.11 单链表节点的实现

■ 首节点的插入与删除

对于如 图二.7(a)所示的单链表，为了插入一个节点，我们首先要创建该节点，然后将其next引用指向当前的首节点（图二.7(b)），然后将队列的head引用指向新插入的节点（图二.7(c)）。



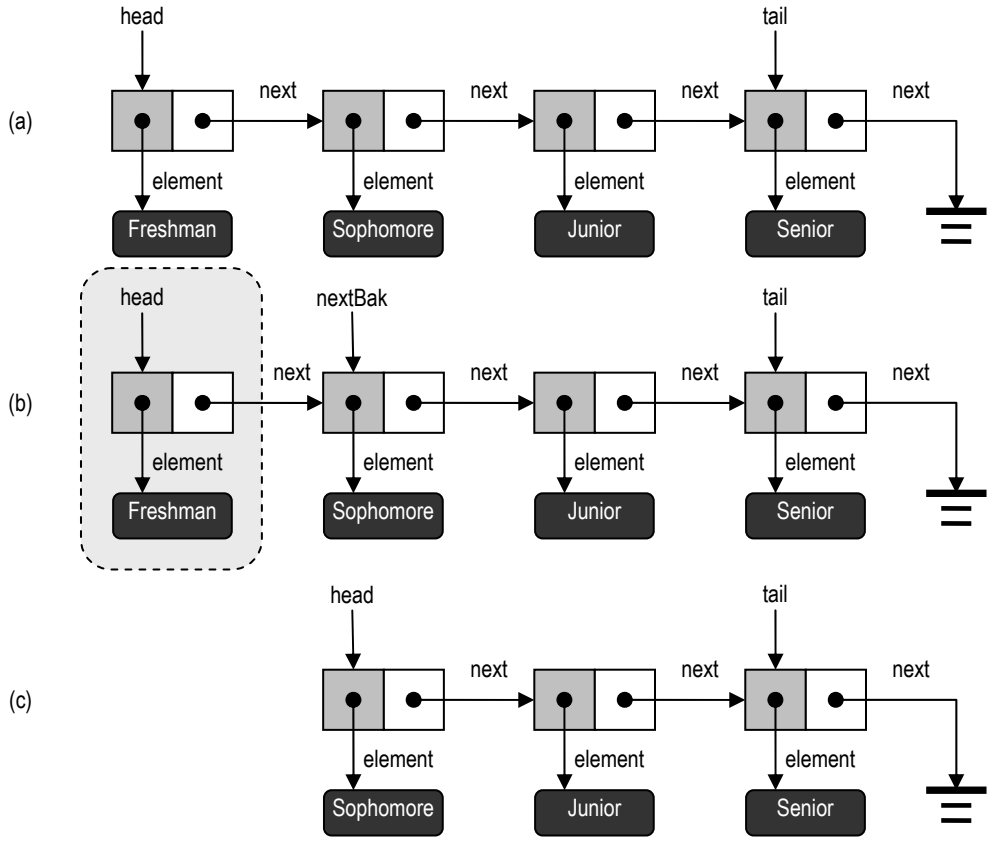
图二.7 在单链表的前端插入节点

请读者自行核对，即使单链表为空，上述操作依然正确。当然，在这种情况下，还需要同时更新 tail 引用，令其指向新插入的节点。

上述过程只涉及常数次基本操作，故可以在 $\mathcal{O}(1)$时间完成。

■ 首节点的删除

反之，对于如 图二.8(a)所示的单链表，为了删除首节点，我们首先将首节点的next引用复制一份，然后才删除该节点（图二.8(b)），最后将表头引用设置为先前复制的引用（图二.8(c)）。



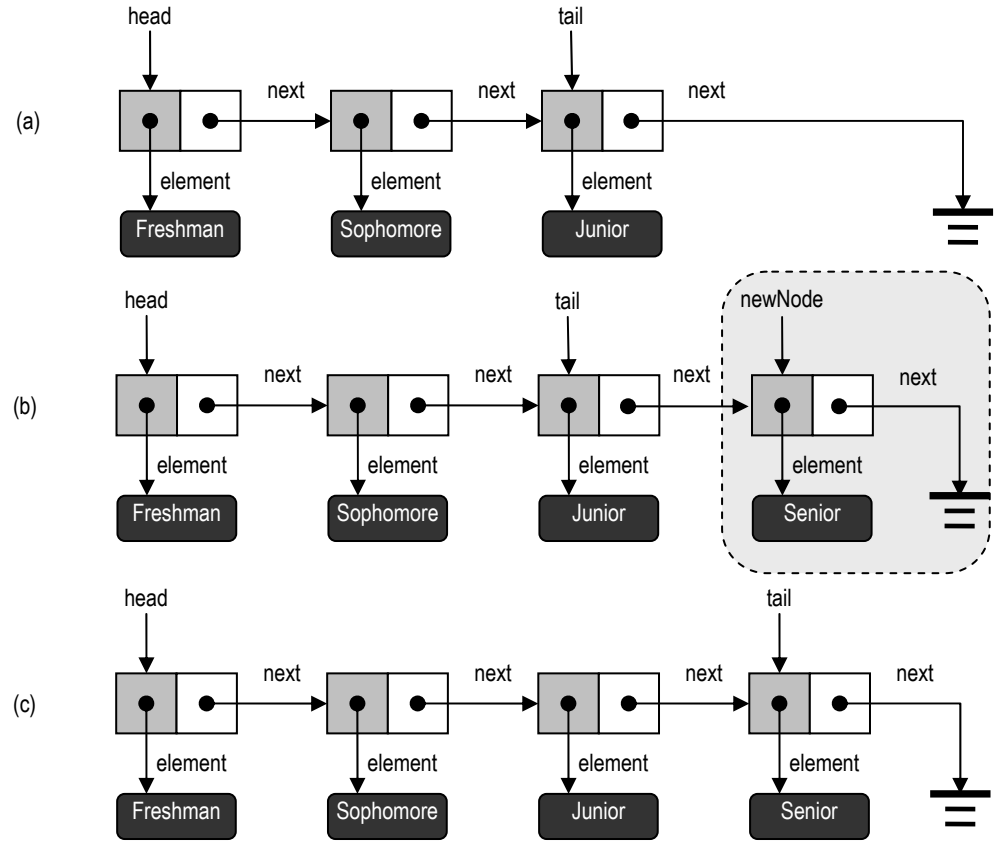
图二.8 删除单链表的首节点

请读者自行核对，即使单链表只含一个节点，上述操作依然正确。当然，在这种情况下，也需要同时更新 tail 引用，将其置为 null。

同样地，上述过程只涉及常数次基本操作，故可以在 $\mathcal{O}(1)$时间完成。

■ 末节点的插入

假定我们借助一个引用tail始终指向的末节点，则在表尾插入新节点也只需$\mathcal{O}(1)$时间。对于如 图二.9(a)所示的单链表，为了将一个新节点作为末节点插入，我们首先要创建该节点，将其next引用置为空，并将当前末节点的next引用指向该新节点（图二.9(b)），最后将tail引用指向新节点（图二.9(c)）。需要特别提醒的是，上述操作的次序很有讲究，不能随意调换。



图二.9 在单链表的后端插入节点

请读者再次自行验证，即使是空的单链表，上述操作依然可行。实际上，对于空的单链表而言，插入的节点既是末节点，也是首节点。因此同样地，在这种情况下，也需要同时更新 head，令其指向新插入的节点。

同样地，上述过程只涉及常数次基本操作，故可以在 $O(1)$时间完成。

■ 末节点的删除

请读者参照前面的介绍，自行给出末节点的删除方法。

需要特别指出的是，即使我们始终通过一个 tail 引用指向当前的末节点，末节点的删除操作也不能在 $O(1)$时间内完成。其原因在于，只有在找到末节点的直接前驱节点之后，才能对表尾节点实施删除操作。然而，为此我们不得不从链表的前端开始逐一检查各个节点——这需要 $O(n)$的时间。

2.3.2 基于单链表实现栈

接下来让我们看看，如何利用单链表结构来实现栈与队列。

由于栈的操作只限于栈顶元素，而单链表只有对首元素才能在$O(1)$时间内完成插入和删除，故这里把单链表的首节点作为栈顶，其余元素依次排列。此外，为了保证getSize()方法也能够在$O(1)$时间内完成，还需借助一个实例变量来动态记录栈中元素的数目。具体的实现如 代码二.12 所示。

```
/*
 * 基于单链表实现栈结构
```

```
 */
```

```
package dsa;

public class Stack_List implements Stack {
  protected Node top;//指向栈顶元素
  protected int size;//栈中元素的数目

//构造方法（空栈）
  public Stack_List()
  { top = null; size = 0; }

//查询当前栈的规模
  public int getSize()
  { return size; }

//判断是否栈空
  public boolean isEmpty()
  { return (top == null) ? true : false; }

//压栈
  public void push(Object elem) {
    Node v = new Node(elem, top);//创建一个新节点，将其作为首节点插入
    top = v;//更新首节点引用
    size++;//更新规模记录
  }

//读取（但不删除）栈顶
  public Object top() throws ExceptionStackEmpty {
    if (isEmpty())
      throw new ExceptionStackEmpty("意外：栈空");
    return top.getElem();
  }

//弹出栈顶
  public Object pop() throws ExceptionStackEmpty {
    if (isEmpty())
      throw new ExceptionStackEmpty("意外：栈空");
    Object temp = top.getElem();
    top = top.getNext();//更新首节点引用
    size--;//更新规模记录
    return temp;
  }
}
```

代码二.12 基于单链表的栈实现

若栈中实际共有 n 个元素，则除常数个实例变量外，只需保存 n 个节点。由于每个节点只占用 $O(1)$空间，故上述实现只占用 $O(n)$的空间。于是，占用的空间量将取决于栈的实际规模，而不再一成不变。相对于利用数组的实现，这一实现的另一优点在于，无需针对溢出问题而刻意去设置一个意外错。当然，溢出错误仍然会得到处理，只不过交由 JRE 来处理。这不仅统一和简化了错误的处理，而且更重要的是，出现栈溢出错误的可能性大大降低——此时，只有在 JRE 没有剩余空间时才会出现栈溢出的现象；而在基于数组的实现中，只要超过数组的长度就会报栈溢出。这两种处理方法有着本质的区别，前者中 JRE 报告的 OutOfMemoryError 属于错误（Error），而后者报告的 ExceptionStackFull 属于意外（Exception）。

这里设置了一个实例变量 top，指向表中的首节点。在新元素 e 入栈时，只需创建一个以 e 为数据的节点 v，并将 v 作为首节点插入。反过来，在退栈时，可以直接摘除首节点，并返回其数据。综上可见，所有插入、删除操作都可以在表的前端进行。根据前面关于单链表结构的分析结论，这些操作都可以在 $O(1)$时间内完成，这一效率与基于数组的栈实现相同。

2.3.3 基于单链表实现队列

与栈一样，我们也可以借助单链表来实现队列ADT。同样地，出于效率方面的考虑，我们将以单链表的首（末）节点作为队列的首（末）节点——这样，可以回避单链表在尾部进行删除操作时效率低下的缺陷。此外，还需要两个实例变量分别指示表的首、末节点。具体的实现如 代码二.13 所示：

```
/*
 * 基于单链表实现队列结构
 */

package dsa;

public class Queue_List implements Queue {
  protected Node head;//指向表首元素
  protected Node tail;//指向表末元素
  protected int size;//队列中元素的数目

//构造方法（空队列）
  public Queue_List()
  { head = tail = null; size = 0; }

//查询当前队列的规模
  public int getSize()
  { return size; }

//判断队列是否为空
  public boolean isEmpty()
  { return (0 == size) ? true : false; }

//入队
  public void enqueue(Object obj) {
    Node node = new Node();
```

```
    node.setElem(obj);
    node.setNext(null);//新节点作为末节点插入

    if (0 == size)  head = node;//若此前队列为空，则直接插入

    else            tail.setNext(node);//否则，将新节点接至队列末端
    tail = node;//更新指向末节点引用
    size++;//更新规模
  }

//出队
  public Object dequeue() throws ExceptionQueueEmpty {
    if (0 == size)
      throw new ExceptionQueueEmpty("意外：队列空");
    Object obj = head.getElem();
    head = head.getNext();
    size--;
    if (0 == size)  tail = null;//若队列已空，须将末节点引用置空
    return obj;
  }

//取（并不删除）队首元素
  public Object front() throws ExceptionQueueEmpty {
    if (isEmpty())
      throw new ExceptionQueueEmpty("意外：队列空");
    return head.getElem();
  }

//遍历（不属于ADT）
  public void Traversal() {
    Node  p = head;
    while (null != p) {
      System.out.print(p.getElem()+" ");
      p = p.getNext();
    }
  }
}
```

代码二.13 基于单链表的队列实现

与基于单链表实现的栈结构同理，如上实现的各个队列 ADT 方法都可以在 $\mathcal{O}(1)$时间内完成。而且，这里同样无需限定队列的最大规模。当然，受单链表结构的限制，我们仍然需要对各种特殊情况（如队空时）专门处理。

§2.4 位置

抽象出位置（Position）这一概念，使我们既能够保持链表结构高效性，而又不致违背面向对象的设计原则。

2.4.1 位置 ADT

所谓“位置”，就是支持以下方法的一种数据类型：

表二.9 位置ADT支持的方法

操作方法	功能描述
getElem():	返回存放于当前位置的元素 输入：无 输出：对象
setElem(e):	将元素 e 放入当前位置，并返回此处原先存放的元素 输入：一个元素 输出：一个元素

按线性次序排列的一组位置，就构成了一个列表。在列表中，各个位置都是相对而言的——相对于它的前、后邻居。在列表中，除第一个（最后一个）位置外，每个位置都有唯一的前驱（后继）位置。如果位置 p 对应于列表中的某个元素 e，那么即使 e 在列表中的次序改变了，位置 p 也依然不变。只有在 e 被显式地删除之后，位置 p 才会相应地被销毁。此外，无论元素 e 被替换还是与另一元素互换，位置 p 都不会改变。正是由于位置的这种特性，我们才能够基于位置的概念，为列表定义出一套以位置对象为参数、返回位置对象的方法。

2.4.2 位置 ADT 接口

代码二.14 给出了位置ADT的Java接口。

```
/*
 * 位置ADT接口
 */

package dsa;

public interface Position {
  public Object getElem();//返回存放于该位置的元素
  public Object setElem(Object e);//将给定元素存放至该位置，返回此前存放的元素
}
```

代码二.14 位置ADT的Java接口

下面，我们将按照Position接口实现双向链表结构。在第 §3.2 节中，我们还将按照Position接口实现列表结构。

§2.5 双端队列

本节介绍队列的一种变型——双端队列（Double-ended queue），简称为Deque[㈠]。顾名思义，也就是前端与后端都支持插入和删除操作的队列。

2.5.1 双端队列的 ADT

相比于栈和队列，双端队列的抽象数据类型要复杂很多，其中基本的方法如下：

表二.10 双端队列ADT支持的基本操作

操作方法	功能描述
insertFirst(x):	将对象 x 作为首元素插入 输入：一个对象 输出：无
insertLast(x):	将对象 x 作为末元素插入 输入：一个对象 输出：无
removeFirst():	若队列非空，则将首元素删除，并将其内容返回 否则，报错 输入：无 输出：对象
removeLast():	若队列非空，则将末元素删除，并将其内容返回 否则，报错 输入：无 输出：对象

此外，还可能支持以下方法：

表二.11 双端队列ADT支持的附加操作

操作方法	功能描述
first():	若队列非空，则返回首元素的内容 否则，报错 输入：无 输出：对象
last():	若队列非空，则返回末元素的内容 否则，报错 输入：无 输出：对象

㈠ d eque 的读音与标准队列 ADT 中的方法 dequeue 雷同（都是[di:kju]）。为了以示区别，通常将 Deque 读作“[dek]”（与 deck 同音）。

操作方法	功能描述
getSize():	报告队列中的元素数目 输入：无 输出：非负整数
isEmpty():	判断队列是否为空 输入：无 输出：布尔量

表二.12 给出了从一个空双端队列开始，在依次执行一系列操作的过程中，队列中内容的相应变化。

表二.12 双端队列操作实例

操作	输出	双端队列（左侧为队列前端）
insertFirst(9) -		{9}
insertFirst(5) -		{5, 9}
removeFirst() 5		{9}
insertLast(3) -		{9, 3}
removeFirst() 9		{3}
removeLast() 3		{}
removeFirst()	空队错	{}
isEmpty() true		{}
insertLast(8) -		{8}
insertLast(6) -		{8, 6}
insertFirst(3)	-	{3, 8, 6}
getSize()	3	{3, 8, 6}

2.5.2 双端队列的接口

双端队列接口如 代码二.15 所示：

```
/*
 * 双端队列接口
 */

package dsa;

public interface Deque {
  public int getSize();//返回队列中元素数目
  public boolean isEmpty();//判断队列是否为空
  public Object first() throws ExceptionQueueEmpty;//取首元素（但不删除）
  public Object last() throws ExceptionQueueEmpty;//取末元素（但不删除）
  public void insertFirst(Object obj);//将新元素作为首元素插入
  public void insertLast(Object obj);//将新元素作为末元素插入
```

```
  public Object removeFirst() throws ExceptionQueueEmpty;//删除首元素

  public Object removeLast() throws ExceptionQueueEmpty;//删除末元素

  public void Traversal();//遍历
}
```

代码二.15 双端队列接口

2.5.3 双向链表

第 2.3.1 节曾经指出，我们需要花费$\mathcal{O}(n)$的时间才能从单链表中删除末节点。因此，若依然采用单链表来实现双端队列，将不可能同时支持对首、末节点的快速删除。为此，我们需要对单链表结构进行必要的扩充，以保证无论是哪一端的删除操作，都能够在$\mathcal{O}(1)$的时间内完成。本节将要介绍的双向链表就能够满足这一要求。顾名思义，这类链表中的每一节点不仅配有next引用，同时还有一个prev引用，指向其直接前驱节点（没有前驱时为null）。代码二.16 给出了双向链表节点类型的具体实现：

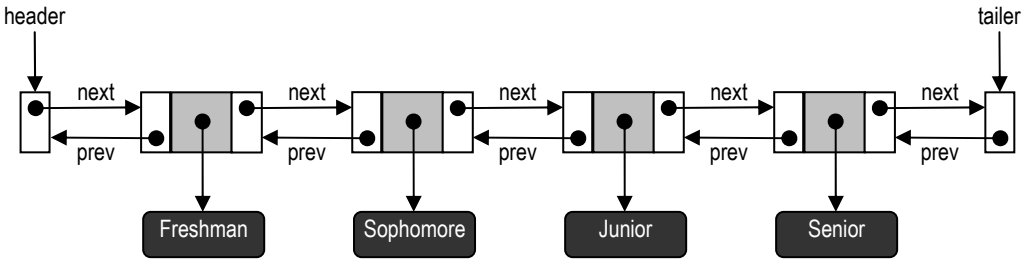
```
/*
 * 基于位置接口实现的双向链表节点类
 */

package dsa;

public class DLNode implements Position {
  private Object element;//数据对象
  private DLNode prev;//指向前驱节点
  private DLNode next;//指向后继节点

/*************************** 构造函数 ***************************/
  public DLNode()
  { this(null, null, null); }

  public DLNode(Object e, DLNode p, DLNode n)
  { element = e; prev = p; next = n; }
  //注意三个参数的次序：数据对象、前驱节点、后继节点

/*************************** Position接口方法 ***************************/
//返回存放于该位置的元素
  public Object getElem()
  { return element; }

//将给定元素存放至该位置，返回此前存放的元素
  public Object setElem(Object e)
  { Object oldElem = element; element = e;  return oldElem; }
```

```
/*************************** 双向链表节点方法 ***************************/
//找到后继位置
  public DLNode getNext()

  { return next; }


//找到前驱位置
  public DLNode getPrev()
  { return prev; }

//修改后继位置
  public void setNext(DLNode newNext)
  { next = newNext; }

//修改前驱位置
  public void setPrev(DLNode newPrev)
  { prev = newPrev; }
}
```

代码二.16 双向链表节点类型的实现

2.5.4 基于双向链表实现的双端队列

■ 哨兵



图二.10 两端附有哨兵节点的双向链表

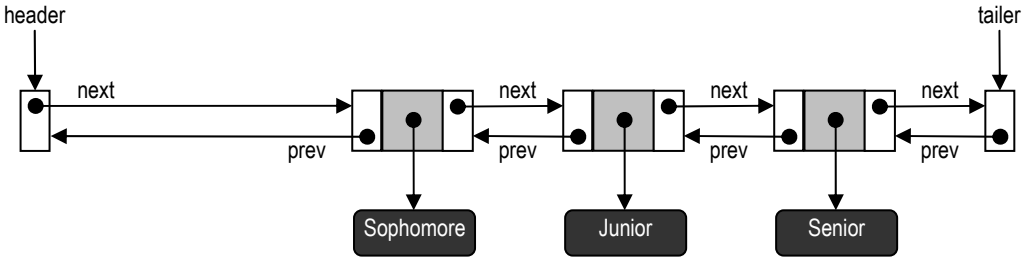
在基于NLNode类实现双向链表的时候，为了使编程更加简洁，通常我们都要在最前端和最后端各设置一个哑元节点（Dummy node ）。这两个节点分别称作头节点（Header node ）和尾节点（Trailer node）[1]，起哨兵（Sentinel）的作用。也就是说，它们并不存储任何实质的数据对象，头

[1] 关于哨兵节点与存储实质内容的末端节点，命名方法十分混乱，在中文教材中尤其如此。本书试图对头、尾节点与首、末节点给出明确的定义，希望能够消除混淆，简化表述。

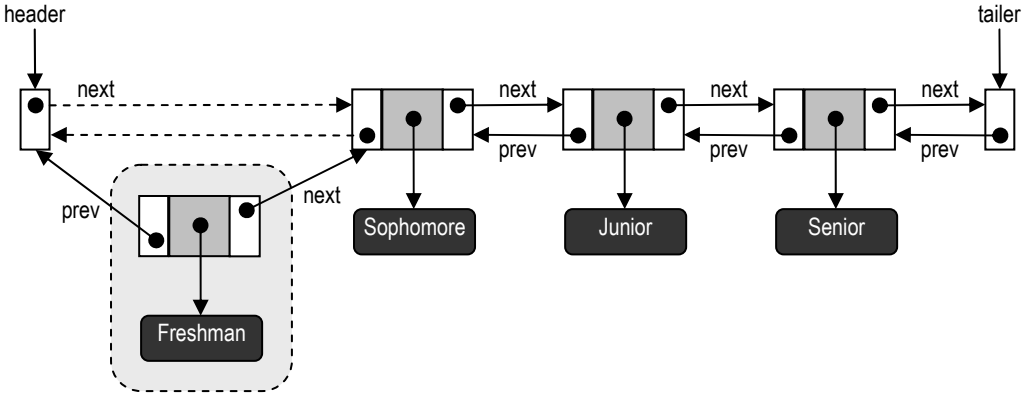
（尾）节点的next（prev）引用指向首（末）节点，而prev（next）引用为空。如此构成的双向链表结构，如 图二.10 所示。

- 首、末节点的插入

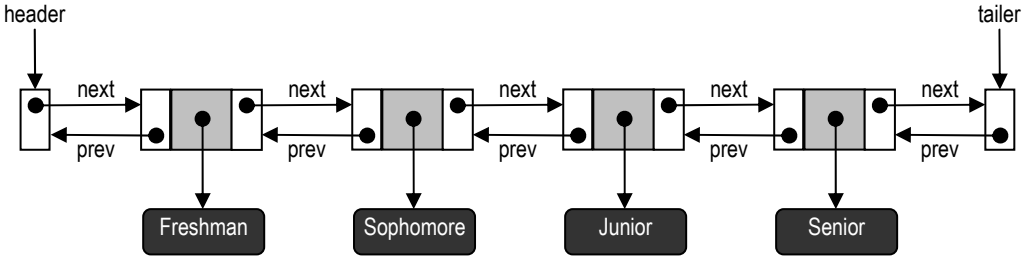
对双端队列的插入操作，与单链表类似。图二.11 给出了在链表前端插入节点的过程。



(a) 原始链表



(b) 生成新节点，将其接入队列前端



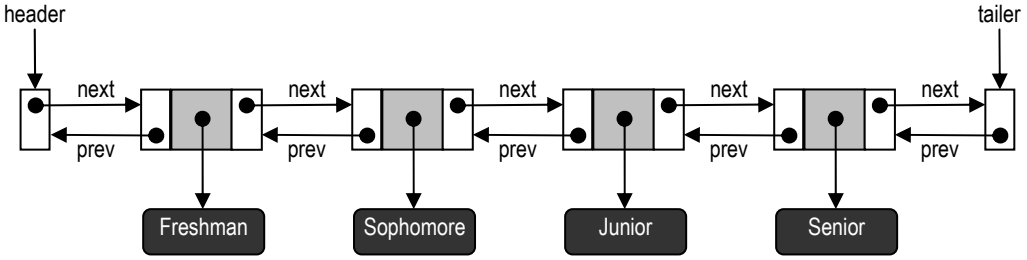
(c) 将头节点的 next 引用、首节点的 prev 引用指向新节点

图二.11 利用附设的哨兵，在双向链表的前端插入新节点

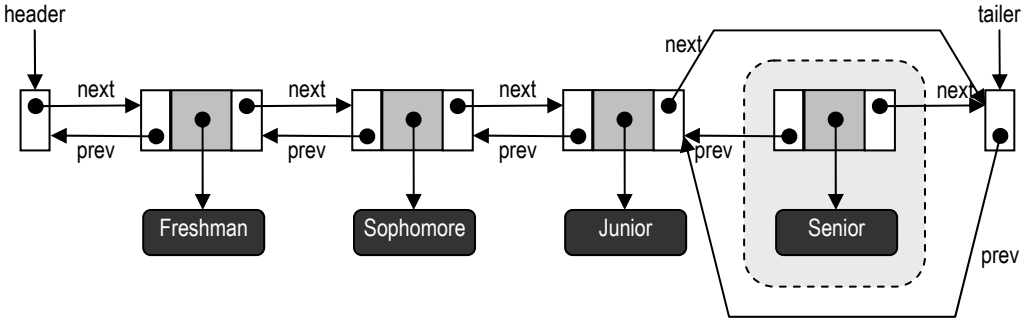
在双向链表的后端插入新节点的过程是对称的，请读者自行给出。

- 首、末节点的删除

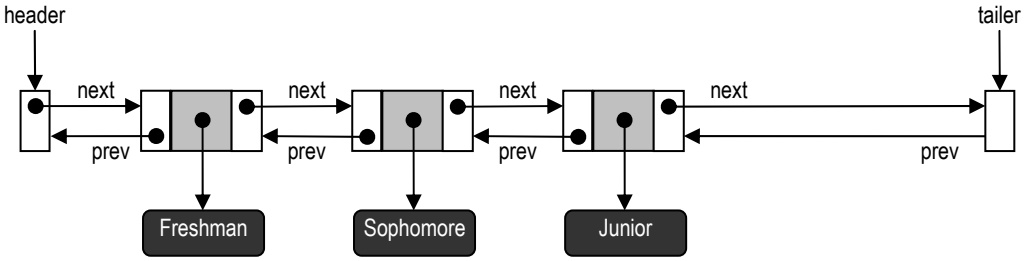
更重要的是，我们可以在$O(1)$时间内删除双向链表任何一端的节点。如 图二.12 所示的，就是删除末节点的过程。首节点的删除过程是对称的，请读者自行给出。



(a) 原始链表



(b) 将新的末节点的 next 引用指向尾节点，尾节点的 prev 引用指向新的末节点

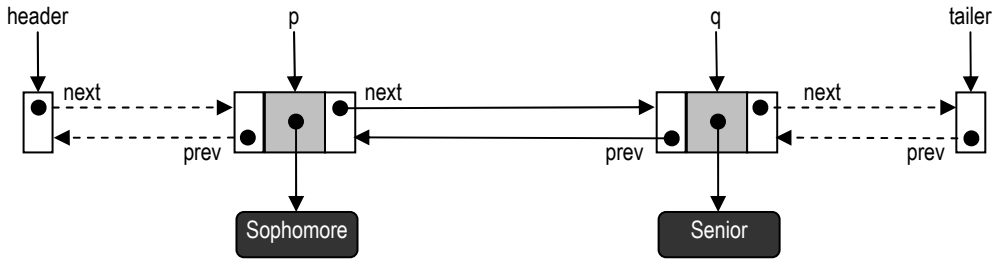


(c) 原先的末节点将有系统回收

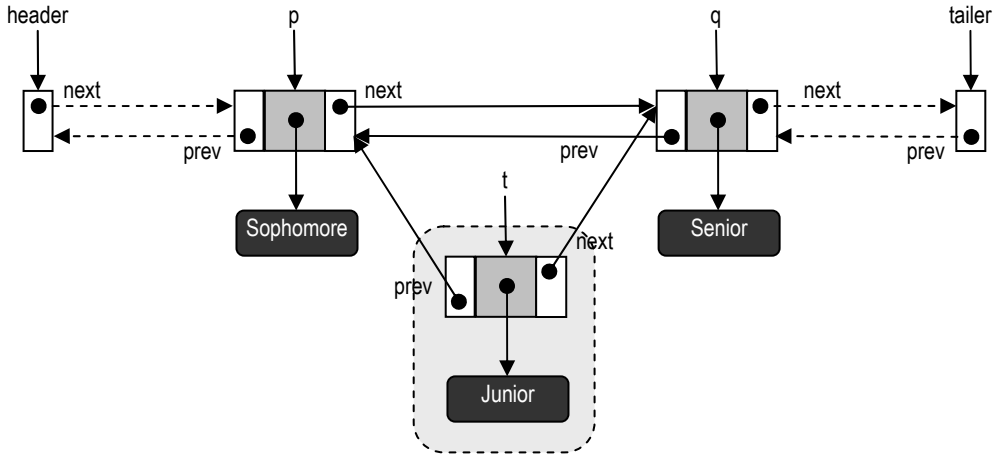
图二.12 利用附设的哨兵，在双向链表的后端删除节点

需要特别指出的是，与单链表不同，这里能够在常数时间内删除末节点。之所以能够这样，是因为可以直接通过末节点的 prev 引用找到它的直接前驱（新的末节点），而无需遍历整个链表。

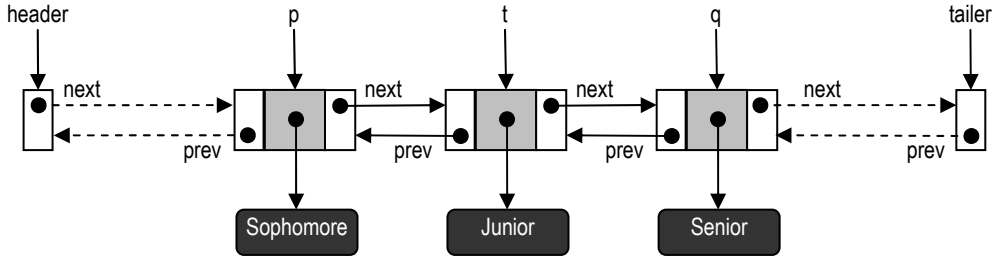
■ 一般节点的插入与删除



(a) 原始链表



(b) 创建新节点 t，令其 prev 和 next 引用分别指向 p 和 q



(c) 令 p 的 next 引用和 q 的 prev 引用指向 t

图二.13 在节点p和q之间插入新节点

一般地，若要在相邻的一对节点 p 和 q 之间插入新元素 e，我们只需生成一个新节点 t，将 e 存放至其中，然后将 t 的 prev 和 next 引用分别指向 p 和 q，然后让 p（q）的 next（prev）引用指向 t。

反过来。若要删除节点 t，我们可以找到其直接前驱节点 p 和直接后继节点 q，然后将 p（q）的 next（prev）引用指向 q（p）。对于节点 t，我们无需做任何处理——既然它不再被引用，就将被内存回收器回收。请读者自行画出具体过程。

需要特别指出的是，p 和 q 都有可能是哨兵节点。请读者自行验证：即使 p 或（和）q 是头（尾）节点，上述操作过程依然正确。

■ 复杂度分析

基于双向链表实现的双端队列中，各方法的运行时间可以归纳为 表二.13。

表二.13 基于双向链表实现的双端队列，各方法的时间复杂度

操作方法	时间复杂度
getSize()	$O(1)$
isEmpty()	$O(1)$
first()	$O(1)$
last()	$O(1)$
insertFirst()	$O(1)$
insertLast()	$O(1)$
removeFirst()	$O(1)$
removeLast()	$O(1)$

总而言之，利用双向链表，可以使双端队列的每一方法都能在常数时间内完成。

■ Jav a 实现

具体的实现如 代码二.17 所示。请读者自行验证：即使是对于空队列，这里对以上ADT方法的实现依然正确。

```
/*
 * 基于双向链表实现双端队列结构
 */

package dsa;

public class Deque_DLNode implements Deque {
  protected DLNode header;//指向头节点（哨兵）
  protected DLNode trailer;//指向尾节点（哨兵）
  protected int size;//队列中元素的数目

//构造函数
  public Deque_DLNode() {
    header = new DLNode();
    trailer = new DLNode();
    header.setNext(trailer);
    trailer.setPrev(header);
    size = 0;
  }

//返回队列中元素数目
  public int getSize()
  { return size; }
```

```
//判断队列是否为空

  public boolean isEmpty()
  { return (0 == size) ? true : false; }

//取首元素（但不删除）
  public Object first() throws ExceptionQueueEmpty {
    if (isEmpty())
      throw new ExceptionQueueEmpty("意外: 双端队列为空");
    return header.getNext().getElem();
  }

//取末元素（但不删除）
  public Object last() throws ExceptionQueueEmpty {
    if (isEmpty())
      throw new ExceptionQueueEmpty("意外: 双端队列为空");
    return  trailer.getPrev().getElem();
  }

//在队列前端插入新节点
  public void insertFirst(Object obj) {
    DLNode  second = header.getNext();
    DLNode  first = new DLNode(obj, header, second);
    second.setPrev(first);
    header.setNext(first);
    size++;
  }

//在队列后端插入新节点
  public void insertLast(Object obj) {
    DLNode  second = trailer.getPrev();
    DLNode  first = new DLNode(obj, second, trailer);
    second.setNext(first);
    trailer.setPrev(first);
    size++;
  }

//删除首节点
  public Object removeFirst() throws ExceptionQueueEmpty {
    if (isEmpty())
      throw new ExceptionQueueEmpty("意外: 双端队列为空");
    DLNode  first = header.getNext();
    DLNode  second = first.getNext();
    Object  obj = first.getElem();
    header.setNext(second);
    second.setPrev(header);
    size--;
    return(obj);
  }
```

```
//删除末节点
  public Object removeLast() throws ExceptionQueueEmpty {

    if (isEmpty())

      throw new ExceptionQueueEmpty("意外: 双端队列为空");
    DLNode  first = trailer.getPrev();
    DLNode  second = first.getPrev();
    Object  obj = first.getElem();
    trailer.setPrev(second);
    second.setNext(trailer);
    size--;
    return(obj);
  }

//遍历
  public void Traversal() {
    DLNode  p = header.getNext();
    while (p != trailer) {
      System.out.print(p.getElem()+" ");
      p = p.getNext();
    }
    System.out.println();
  }
}
```

代码二.17 基于双向链表实现双端队列结构

第三章

向量、列表与序列

所谓序列（Sequence），就是依次排列的多个对象。比如，每一计算机程序都可以看作一个序列，它由一系列依次排列的指令组成，正是指令之间的次序决定了程序的具体功能。因此，所谓序列，就是一组对象之间的后继与前驱关系。在实际问题中，序列可以用来实现很多种数据结构，因此被认为是数据结构设计的基础。

本章将介绍两种典型的序列：向量（Vector）和列表（List）。它们都由一组按线性次序排列的元素组成，并支持若干个访问、插入和删除元素的方法。二者之间的区别，在于它们所各自支持的ADT方法。第二章所介绍的栈、队列以及双端队列，都可以看作带有附加限制的序列：插入、删除和访问操作只限于首元素的，就是栈；删除和访问操作只限于首元素、插入操作只限于末元素的，就是队列；前、后端都支持插入、删除和访问操作的，就是双端队列。作为更一般性的结构，序列依然保持了这些结构的一条重要性质：其中各元素的逻辑次序与其数值无关，只取决于在序列结构生命期内所执行过的具体ADT操作。

对数组结构进行抽象与扩展之后，就可以得到向量结构，因此向量也称作数组列表（Array list）。向量提供一些访问方法，使得我们可以通过下标直接访问序列中的元素，也可以将指定下标处的元素删除，或将新元素插入至指定下标。为了与通常数组结构的下标（Index）概念区分开来，我们通常将序列的下标称为秩（Rank）。

与向量相对应地，列表ADT则是对链表结构的抽象。列表提供的访问、更新方法，按照面向对象的规范对列表的节点对象进行了封装，称作位置（Position）。这里，我们沿用第 §2.4 节所提供的一种面向对象式方法，以指示各元素的存放“位置”。

最后，我们将给出完整的序列 ADT，它将向量与列表的 ADT 统一起来，并给出序列的两种（基于数组和基于双向链表）基本实现方法。我们还将对这两种实现的性能做一比较，并讨论如何做出权衡。本章的讨论还涉及到迭代器（Iterator）的设计模式，并给出其具体实现。

§3.1 向量与数组

假定集合 S 由 n 个元素组成，它们按照线性次序存放，于是我们就可以直接访问其中的第一个元素、第二个元素、第三个元素……。也就是说，通过[0, n-1]之间的每一个整数，都可以直接访问到唯一的元素 e，而这个整数就等于 S 中位于 e 之前的元素个数——在此，我们称之为该元素的秩（Rank）。不难看出，若元素 e 的秩为 r，则只要 e 的直接前驱（或直接后继）存在，其秩就是 r-1（或 r+1）。这一定义与 Java、C++之类的程序语言中关于数组元素的编号规则是一致的。

支持通过秩直接访问其中元素的序列，称作向量（Vector）或数组列表（Array list）。实际上，秩这一直观概念的功能非常强大——它可以直接指定插入或删除元素的位置。

3.1.1 向量 ADT

向量 ADT 定义了如下方法：

表三.1 向量ADT支持的操作

操作方法	功能描述
getSize()：	报告向量中的元素数目 输入：无 输出：非负整数
isEmpty()：	判断向量是否为空 输入：无 输出：布尔量
getAtRank(r)：	若 0 ≤ r < getSize()，则返回秩为 r 的那个元素 否则，报错 输入：一个整数 输出：对象
replaceAtRank(r, e)：	若 0 ≤ r < getSize()，则将秩为 r 的元素替换为 e，并返回原来的元素 否则，报错 输入：一个整数和一个对象 输出：对象
insertAtRank(r, e)：	若 0 ≤ r ≤ getSize()，则将 e 插入向量中，作为秩为 r 的元素（原秩不小于 r 的元素顺次后移）；并返回该元素 否则，报错 输入：一个整数和一个对象 输出：对象
removeAtRank(r)：	若 0 ≤ r < getSize()，则删除秩为 r 的那个元素并返回之（原秩大于 r 的元素顺次前移） 否则，报错 输入：一个整数 输出：对象

请注意其中的 insertAtRank()方法，该方法之所以要返回被插入的元素，是为了使程序员可以链式编程。

一种直截了当的方法就是采用数组来实现向量：下标为r的数组项，就对应于秩为r的元素。当然，还有很多种其它的实现形式。之所以要提出秩的概念，就是为了使我们不用了解序列的具体实现，就可以通过广义的“下标”直接访问序列中对应的元素。如 表三.2 所示，随着序列的更新，其中元素的秩也会不断变化。

表三.2 向量操作实例：从空向量开始，依次执行一系列操作的相应结果

操作	输出	向量组成（自左向右）
insertAtRank(0, 9)	9	{9}
insertAtRank(0, 4)	4	{4, 9}
getAtRank(1) 9		{4, 9}
insertAtRank(2, 5)	5	{4, 9, 5}
getAtRank(3)	越界错	{4, 9, 5}
insertAtRank(3, 7)	越界错	{4, 9, 5}
removeAtRank(1) 9		{4, 5}

操作	输出	向量组成（自左向右）
insertAtRank(1, 3)	5	{4, 3, 5}
removeAtRank(3)	越界错	{4, 3, 5}
insertAtRank(1, 2)	2	{4, 2, 3, 5}
replaceAtRank(2, 7)	3	{4, 2, 7, 5}

■ 意外错

当作为参数的秩越界时，对应的意外错为ExceptionBoundaryViolation（代码三.1）。

```
/*
 * 当作为参数的数组下标、向量的秩或列表的位置越界时，本意外将被抛出
 */

package dsa;

public class ExceptionBoundaryViolation extends RuntimeException {
  public ExceptionBoundaryViolation(String err) {
    super(err);
  }
}
```

代码三.1 ExceptionBoundaryViolation意外错

■ 向量接口

代码五.12 给出了向量的接口定义：

```
/*
 * 向量接口
 */

package dsa;

public interface Vector {
//返回向量中元素数目
  public int getSize();

//判断向量是否为空
  public boolean isEmpty();

//取秩为r的元素
  public Object getAtRank(int r)
  throws ExceptionBoundaryViolation;

//将秩为r的元素替换为obj
  public Object replaceAtRank(int r, Object obj)
```

```
  throws ExceptionBoundaryViolation;


//插入obj，作为秩为r的元素；返回该元素
  public Object insertAtRank(int r, Object obj)
  throws ExceptionBoundaryViolation;

//删除秩为r的元素
  public Object removeAtRank(int r)
  throws ExceptionBoundaryViolation;
}
```

代码三.2 向量接口

■ 利用向量实现双端队列

向量提供的以上方法似乎不多，但如 表三.3 所示，通过这些方法，完全可以实现第 2.5.1 节定义的双端队列ADT中的所有方法。

表三.3 利用向量提供的方法实现双端队列

双端队列的方法	借助向量方法的实现
getSize() getSiz	e()
isEmpty() isEmpty()	
first() getAtRank(0)	
last() getAtRank(getS	ize()-1)
insertFirst(e) insertAtRank(0,	e)
insertLast(e) insertAt	Rank(getSize(), e)
removeFirst() removeAtR	ank(0)
removeLast() remove	AtRank(getSize()-1)

3.1.2 基于数组的简单实现

基于数组，可以直接实现向量 ADT。我们借用一个数组 A[]，其中 A[i]分别存放一个引用，指向秩为 i 的向量元素。为此，A[]的容量 N 需要足够大，还需要设置一个实例变量 n 指示向量的实际规模。

代码三.3 给出了具体的实现细节。

```
/*
 * 基于数组的向量实现
 */

package dsa;

public class Vector_Array implements Vector {
```

```
  private final int N = 1024;//数组的容量
  private int n = 0;//向量的实际规模

  private Object[] A;//对象数组


//构造函数
  public Vector_Array() {
    A = new Object[N];
    n = 0;
  }

//返回向量中元素数目
  public int getSize() { return n; }

//判断向量是否为空
  public boolean isEmpty() { return (0 == n) ? true : false; }

//取秩为r的元素
  public Object getAtRank(int r)//O(1)
  throws ExceptionBoundaryViolation {
    if (0 > r || r >= n)  throw new ExceptionBoundaryViolation("意外: 秩越界");
    return A[r];
  }

//将秩为r的元素替换为obj
  public Object replaceAtRank(int r, Object obj)
  throws ExceptionBoundaryViolation {
    if (0 > r || r >= n)  throw new ExceptionBoundaryViolation("意外: 秩越界");
    Object bak = A[r];
    A[r] = obj;
    return bak;
  }

//插入obj，作为秩为r的元素；返回该元素
  public Object insertAtRank(int r, Object obj)
  throws ExceptionBoundaryViolation {
    if (0 > r || r > n) throw new ExceptionBoundaryViolation("意外: 秩越界");
    if (n >= N) throw new ExceptionBoundaryViolation("意外: 数组溢出");
    for (int i=n; i>r; i--) A[i] = A[i-1];//后续元素顺次后移
    A[r] = obj;//插入
    n++;//更新当前规模
    return obj;
  }

//删除秩为r的元素
  public Object removeAtRank(int r)
  throws ExceptionBoundaryViolation {
```

```
    if (0 > r || r >= n)  throw new ExceptionBoundaryViolation("意外: 秩越界");
    Object bak = A[r];
    for (int i=r; i<n; i++) A[i] = A[i+1];//后续元素顺次前移
    n--;//更新当前规模

    return bak;

  }
}
```

代码三.3 基于数组的向量实现

■ 性能分析

这一实现所占用的空间量取决于向量的最大规模，即数组的容量$\mathcal{O}(N)$。与基于数组实现的其它结构一样，这一复杂度与向量的实际规模无关。

基于数组实现的向量中，各方法的运行时间可以归纳为 表三.4。

表三.4 基于数组实现的向量，各方法的时间复杂度

操作方法	时间复杂度
getSize()	$\mathcal{O}(1)$
isEmpty()	$\mathcal{O}(1)$
getAtRank()	$\mathcal{O}(1)$
replaceAtRank()	$\mathcal{O}(1)$
insertAtRank()	$\mathcal{O}(n)$
removeAtRank()	$\mathcal{O}(n)$

其中插入（删除）操作之所以需要耗费线性时间，是因为需要将操作位置的所有后继元素向后（向前）顺次移动。这样的效率实在难以令人满意，因此我们将在后面讨论其它的实现。

3.1.3 基于可扩充数组的实现

第 3.1.2 节给出的向量实现，有个很大的缺陷——数组容量N固定。一方面，在向量规模很小时，预留这么多的空间实属浪费；反过来，当向量规模超过N时，即使系统有足够的空间资源，也会因意外错而崩溃。幸好，有一个简易的方法可以克服这一缺陷。

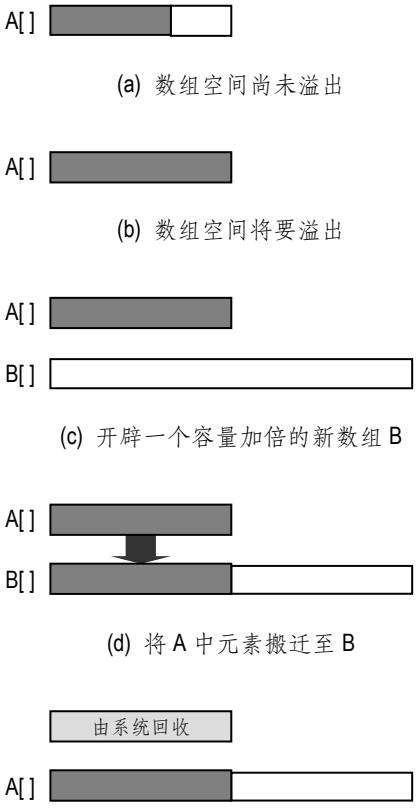
我们希望向量能够根据实际需要，动态地扩充数组的容量。当然，Java 语言本身并不支持这一功能，与多数程序语言一样，Java 中数组的容量都是固定的。我们的策略是，一旦数组空间溢出（即$n \geq N$），insertAtRank()方法就会做如下处理：

```
1. 开辟一个容量为2N的新数组B
2. 将A[]中各元素搬迁至B[]，即B[i]=A[i]，i=0, …, N-1
```

3．将A替换为B，即令A=B

此后，insertAtRank()就可以将新元素插入至扩容后的数组了。

实际上，这就是所谓的可扩充数组（Extendable arra y）策略。就像蝉一样，每当生长到足够大时，就会蜕去原来的外壳，换上一身更大的外壳。具体过程如 图三.1 所示。不必担心原先的数组，它占用的空间将被内存回收器自动回收。



图三.1 可扩充数组的溢出处理

代码五.12 给出了这一策略的具体实现。

```
/*
 * 基于可扩充数组的向量实现
 */

package dsa;

public class Vector_ExtArray implements Vector {
  private int N = 8;//数组的容量，可不断增加
  private int n;//向量的实际规模
  private Object A[];//对象数组

//构造函数
```

```
  public Vector_ExtArray()  { A = new Object[N];  n = 0; }

//返回向量中元素数目
  public int getSize() { return n; }


//判断向量是否为空

  public boolean isEmpty() { return (0 == n) ? true : false; }

//取秩为r的元素
  public Object getAtRank(int r)
  throws ExceptionBoundaryViolation {
    if (0 > r || r >= n)  throw new ExceptionBoundaryViolation("意外: 秩越界");
    return A[r];
  }

//将秩为r的元素替换为obj
  public Object replaceAtRank(int r, Object obj)
  throws ExceptionBoundaryViolation {
    if (0 > r || r >= n)  throw new ExceptionBoundaryViolation("意外: 秩越界");
    Object bak = A[r];
    A[r] = obj;
    return bak;
  }

//插入obj，作为秩为r的元素；并返回该元素
  public Object insertAtRank(int r, Object obj)
  throws ExceptionBoundaryViolation {
    if (0 > r || r > n) throw new ExceptionBoundaryViolation("意外: 秩越界");
    if (N <= n) {//空间溢出的处理
      N *= 2;
      Object  B[] = new Object[N];//开辟一个容量加倍的数组
      for (int i=0; i<n; i++) B[i] = A[i];//A[]中内容复制至B[]
      A = B;//用B替换A（原A[]将被自动回收）
    }
    for (int i=n; i>r; i--) A[i] = A[i-1];//后续元素顺次后移
    A[r] = obj;//插入
    n++;//更新当前规模
    return obj;
  }

//删除秩为r的元素
  public Object removeAtRank(int r)
  throws ExceptionBoundaryViolation {
    if (0 > r || r >= n)  throw new ExceptionBoundaryViolation("意外: 秩越界");
    Object bak = A[r];
    for (int i=r; i<n-1; i++) A[i] = A[i+1];//后续元素顺次前移
```

```
    n--;//更新当前规模
    return bak;
  }
}
```

代码三.4 基于可扩充数组的向量实现

■ 时间复杂度的分摊分析

与简单的数组实现相比，基于可扩充数组的实现可以更高效地利用存储空间。另外，只要系统还有可利用的空间，向量的最大规模将不再受限于数组的容量。然而，我们也需要为此付出代价——每次扩容时，需要花费额外的时间以将原数组的内容复制到新数组中。准确地说，为了将数组的容量由 N 扩至 2N，需要花费 $\mathcal{O}(N)$的时间。

如此看来，上述扩充数组的策略似乎效率很低。然而幸运的是，这不过是一种错觉。事实上，每经过一次扩充，数组的容量都会加倍，此后，至少要再经过 N 次插入操作，才有可能需要再次扩容。也就是说，随着向量规模的增加，需要进行扩容的概率会急剧下降。下面，我们就对此做一严格的分析。

不过，扩容操作的执行并不确定，因此无法采用通常的方法来度量和分析其复杂度。为此，我们可以引入分摊复杂度的概念。所谓分摊运行时间（Amortized running time），就是指在连续执行的足够多次操作中，每次操作所需的平均运行时间。

需要强调的是: 这里分摊时间(Amortized time) 与平均时间(Average r unning time) 有本质的区别。一个算法的平均运行时间，指的是对于出现概率符合某种分布的所有输入，算法所需运行时间的平均值，因此也称作期望运行时间（Expected runnin g time）。而这里的分摊运行时间，指的是在反复使用某一算法及其数据结构的过程中，连续的足够多次运行所需时间的平均值。

以这里的可扩充数组为例，我们需要考察对这一结构的连续 n 次操作，将每次操作所需的时间累计起来，然后除以 n，只要 n 足够大，这个平均时间就是分摊运行时间。

定理三.1 基于可扩充数组实现的向量，每次数组扩容的分摊运行时间为 $\mathcal{O}(1)$。

〖证明〗

假定数组的初始容量为常数 N。这里的目标是估计复杂度的上界，故不妨设向量的初始规模也为 N，我们来考察此后的连续 n 次 insertAtRank()方法调用。

定义如下函数:

$$\begin{cases} V(n) = \text{连续插入n个元素后向量的规模} \\ A(n) = \text{连续插入n个元素后数组的容量} \\ T(n) = \text{为连续插入n个元素而花费于扩容的时间} \end{cases}$$

首先，易见 V(n) = n + N。

另外，数组的容量为不小于 V(n)的、形如 $2^m N$ 的整数，其中 $m = \lceil \log_2(1+n/N) \rceil \geq 0$。由于可以将 N 视作常数，故有:

$$A(n) = 2^mN = N \times 2^{\lceil \log_2(1+n/N) \rceil} = \mathcal{O}(n)$$

为证明定理的结论，只需证明 $T(n) = \mathcal{O}(A(n))$。为此可以采用数学归纳法。

首先考虑最平凡的情况：$n = 0$，即尚未插入任何元素的最初状态。此时有 $T(n) = 0$，$A(n) = N$，命题成立。

假定命题对任何 $n \le (2^m-1)N$ 都成立，现考虑所有的 $(2^m-1)N < n \le (2^{m+1}-1)N$。在这期间的 2^mN 次插入中，只有第一次（$n = (2^m-1)N+1$）需要将数组的容量由 2^mN 扩至 $2^{m+1}N$，为此需要花费 $\mathcal{O}(2^mN)$ 的时间。于是，所有 n 次插入操作花费于扩容的时间为

$$T(n) = T((2^m-1)N+1) \le \mathcal{O}(2^mN) + \mathcal{O}(2^mN) = \mathcal{O}(2^{m+1}N) = A(n)$$

因此，分摊到这连续的 n 次操作，每次操作平均所需的运行时间为

$$T(n) / n = A(n)/n = \mathcal{O}(n) / n = \mathcal{O}(1)$$

证毕。 ❑

根据 定理三.1，动态扩充数组容量的策略不仅可行，而且就分摊复杂度而言，效率也足以令人满意。

读者可能已经发现，如果简单地按照平均复杂度进行估计，在最坏情况下，对可扩充数组单次操作需要 $\mathcal{O}(n)$ 的时间。由此可见，分摊分析的方法能够更为精细、准确和客观地评价算法和数据结构的性能。

这一方法应用的条件是，最坏情况虽然会出现，但在实际的足够多次连续的操作过程中，出现最坏情况的可能性远远低于我们的直觉。就以可扩充数组为例，从 定理三.1 的证明中可以看出，在任何一个实际的可扩充数组的生命期内，随着其容量的增加，溢出的可能性将急剧下降。

在后续的章节中，我们还会多次使用分摊分析的方法。

3.1.4 java.util.ArrayList 类和 java.util.Vector 类

Java本身也提供了与向量ADT功能类似的两个类：java.util.ArrayList和java.util.Vector。而且，这两个类的标准实现都采用了第 3.1.3 节中介绍的策略——根据实际需要，动态扩充数组的容量。

表三.5 向量ADT与java.util.ArrayList类的对比

向量 ADT 中的方法	java.util.ArrayList 类的方法
getSize()	
isEmpty()	
getAtRank() get()	
replaceAtRank() s	et()
insertAtRank() add()	
removeAtRank() remove()	

表三.5 将这两个类与第 3.1.3 节中的向量ADT做了比较。 可以看出，Java为这两个类采用了更为简短的方法名，虽然这使得程序的编写更为简洁，但反过来却降低了程序的可读性。

除了我们在向量ADT中定义的方法，java.util.ArrayList类和java.util.Vector类还提供了其它方面的支持。比如java.util.Vector类设有一个名为capacityIncrement的参数，通过这一参数，程序员可以指定每次对数组进行扩容的增量。该参数默认为 0，对应的策略与第 3.1.3 节相同——一旦数组溢出，就将其容量加倍。若参数capacityIncrement为正整数，则在数组溢出时，只追加capacityIncrement个单元。虽然这个参数为程序员提供了选择的可能，但在具体使用时我们必须格外小心。实际上，在绝大多数情况下都应该使用默认值；否则，很有可能会使时间复杂度陡增。读者可以自行分析和证明：通常情况下，若将capacityIncrement设为正整数，则在连续n次操作过程中消耗于数组扩容的时间有可能会高达$\Omega(n^2)$——也就是说，分摊下来每一操作平均需要$\mathcal{O}(n)$时间！

§3.2 列表

3.2.1 基于节点的操作

试考察一个（单向或双向）链表 S。如果直接照搬秩的概念，对链表的访问速度会很慢——为了在链表结构中确定特定元素的秩，我们不得不顺着元素间的 next（或 prev）引用，从前端（双向链表也可以从后端）开始逐一扫描各个元素，直到发现目标元素。在最坏情况下，这需要线性的时间。请读者证明：即使考虑所有元素等概率出现的情况，每次操作的平均运行时间也是 $\mathcal{O}(n)$。

实际上，除了通过秩，我们还可以通过其它途径来确定元素在序列中的位置。我们希望能够为序列扩充若干新的方法，使之能够直接以某些节点做为参数，找到所需的节点并对其实施操作。比方说，若能够将方法 removeAtRank(r)替换为新方法 removeAtNode(v)，即可直接删除节点 v 所对应的元素。只要能够以节点做为参数，就可以在 $\mathcal{O}(1)$的时间内直接确定目标节点的存放位置，进而修改其前驱与后继的 next 和 prev 引用，以完成对该元素的删除。类似地，若能够将 insertAtRank(r, obj)方法替换为新方法 insertAfterNode(v, obj)，即可将 obj 插入到节点 v 的后继位置。总之，只要能够像这样地以节点作为参数，就可以在 $\mathcal{O}(1)$时间内确定目标节点的存储位置，进而完成相应的操作。

3.2.2 由秩到位置

为了在列表 ADT 中增添上述基于节点的操作，必须首先回答这样一个问题：需要将多少有关列表具体实现的信息暴露给程序员？自然，最好是既能够利用单链表或双向链表，同时又能够将其中的细节封装起来。同样地，尽管如此实现的列表的确为程序员提供了直接访问和修改内部数据的可能（比如 next 和 prev 引用），出于安全性的考虑，我们还是不鼓励他们去使用这些底层的功能。

为此，在下面将要给出的列表ADT中，我们对链表结构做了抽象处理。实际上，我们可以借助第 §2.4 节介绍的“位置”概念来实现这一抽象。在经过抽象化处理后，在统一的列表ADT下可以有多种实现形式，而且元素的存储形式也不尽相同，就这个意义而言，位置（Position）的概念就是对元素的不同形式的抽象和统一，也是对列表元素之间相对“位置”的形式化刻画。正是在这种抽象的基础上，我们才可以安全地为列表扩充一整套基于节点的操作。按照这一思路，依然可以将列表看作是存放元素的容器，其中的元素各有自己的存放位置，我们只是将它们的位置组织为一个线性的结构。

3.2.3 列表 ADT

在引入位置概念对列表“节点”进行封装之后，就可以得到一个新的序列 ADT，称作列表（List）。该 ADT 支持对列表 S 的以下方法：

表三.6 列表ADT支持的方法

操作方法	功能与描述
first():	若 S 非空，则给出其中第一个元素的位置 否则，报错 输入：无 输出：位置
last():	若 S 非空，则给出其中最后一个元素的位置 否则，报错 输入：无 输出：位置
getPrev(p):	若 p 不是第一个位置，则给出 S 中 p 的前驱所在的位置 否则，报错 输入：位置 输出：位置
getNext(p):	若 p 不是最后一个位置，则给出 S 中 p 的前驱所在的位置 否则，报错 输入：位置 输出：位置

通过以上方法，才可以谈论并使用相对于列表前端或末端的位置概念，并且顺次访问列表的各个元素。尽管这一位置概念可以直观地与列表中的节点对应起来，但一定要注意：这里并没有通过什么引用直接指向各节点对象。此外，若以位置作为参数传递给某个列表方法，就必须保证该位置的确属于该列表。

- 修改列表的方法

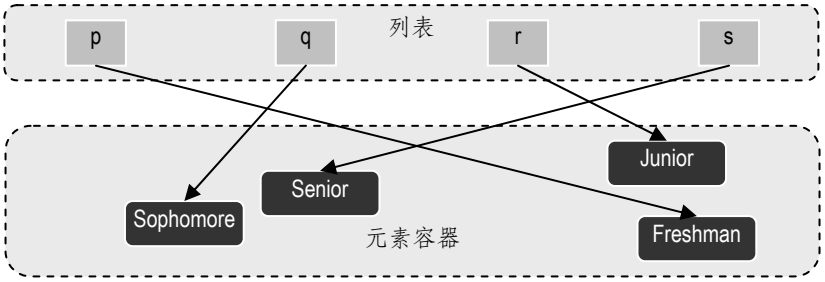
除了通常都提供的 getSize()和 isEmpty()以及上面给出的方法，为了对列表做修改与更新，列表 ADT 还需支持以下方法。这些方法的共同点在于，它们都以位置为参数或（和）返回值。

表三.7 列表支持的动态修改操作

操作方法	功能与描述
replace(p, e):	将处于位置 p 处的元素替换为 e，并返回被替换的元素 输入：一个位置和一个对象 输出：对象
insertFirst(e):	将元素 e 作为第一个元素插入 S 中，并返回 e 的位置 输入：一个对象 输出：位置
insertLast(e):	将元素 e 作为最后一个元素插入 S 中，并返回 e 的位置 输入：一个对象 输出：位置

操作方法	功能与描述
insertBefore(p, e):	将元素 e 插入于 S 中位置 p 之前，并返回 e 的位置 输入：一个位置和一个对象 输出：位置
insertAfter(p, e):	将元素 e 插入于 S 中位置 p 之后，并返回 e 的位置 输入：一个位置和一个对象 输出：位置
remove(p):	删除 S 中位置 p 处的元素 输入：一个位置 输出：对象

如 图三.2 所示，有了列表ADT，我们就可以从位置的角度来看待一组按序存放的对象，而不必考虑它们的位置的具体表示。



图三.2 列表中各位置的次序：p → q → r → s

有的读者或许已经注意到，在上述列表 ADT 的定义中，有些操作的功能是重复的。比如，insertBefore(first(), e) 的效果与 insertFirst(e)完全相同，insertAfter(last(), e) 的效果与 insertLast(e) 也完全相同。之所以允许这类功能的冗余，是为了增加代码的可读性。

还需注意的是，当作为参数传递给上述方法的位置不合法时，需要做报错处理。位置 p 非法，有以下可能：

- null == p
- p 已被删除
- p 是另一列表中的位置
- 在调用方法 getPrev(p)时，p 是列表中的第一个位置
- 在调用方法 getNext(p)时，p 是列表中的最后一个位置

表三.8 给出了从一个空列表开始，在依次执行一系列操作的过程中，列表中内容的相应变化。（这里用p_1、p_2、p_3、p_4 和p_5 分别表示各个位置）

表三.8 列表操作实例

操作方法	输出	列表结构组成（自左向右）
insertFirst(9) p	$_1$(9) (9)	
insertAfter(p_1, 5)	p_2(5) (9,	5)
insertBefore(p_2, 7)	p_3(7)	(9, 7, 5)

操作方法	输出	列表结构组成（自左向右）
insertFirst(3) p	$_4$(3)	(3, 9, 7, 5)
getPrev(p_3) p	$_1$(9)	(3, 9, 7, 5)
last() p	$_2$(5)	(3, 9, 7, 5)
remove(p_4)	3	(9, 7, 5)
replace(p_3, 6)	7	(9, 6, 5)
insertAfter(first(), 2)	p_5(2)	(9, 2, 7, 5)
getPrev(p_1)	非法位置错	(9, 2, 7, 5)
getNext(p_2)	非法位置错	(9, 2, 7, 5)

■ 意外错

当作为参数的位置越界时，意外错ExceptionBoundaryViolation（代码三.1）将会被扔出。

当作为参数的位置非法（例如为null或不属于任何列表）时，意外错ExceptionPositionInvalid（代码五.12）将会被扔出。

```
/*
 * 当作为参数的数组下标、向量的秩或列表的位置非法时，本意外将被抛出
 */

package dsa;

public class ExceptionPositionInvalid extends RuntimeException {
  public ExceptionPositionInvalid(String err) {
    super(err);
  }
}
```

代码三.5 Ex ceptionPositionInvalid意外错

■ 列表 ADT 接口

列表ADT的Java接口如 代码三.6 所示。

```
/*
 * 列表ADT接口
 */

package dsa;

public interface List {
//查询列表当前的规模
  public int getSize();

//判断列表是否为空
  public boolean isEmpty();
```

```
//返回第一个元素（的位置）
  public Position first();

//返回最后一个元素（的位置）
  public Position last();
```

```
//返回紧接给定位置之后的元素（的位置）
  public Position getNext(Position p)
  throws ExceptionPositionInvalid, ExceptionBoundaryViolation;

//返回紧靠给定位置之前的元素（的位置）
  public Position getPrev(Position p)
  throws ExceptionPositionInvalid, ExceptionBoundaryViolation;

//将e作为第一个元素插入列表
  public Position insertFirst(Object e);

//将e作为最后一个元素插入列表
  public Position insertLast(Object e);

//将e插入至紧接给定位置之后的位置
  public Position insertAfter(Position p, Object e)
  throws ExceptionPositionInvalid;

//将e插入至紧靠给定位置之前的位置
  public Position insertBefore(Position p, Object e)
  throws ExceptionPositionInvalid;

//删除给定位置处的元素，并返回之
  public Object remove(Position p)
  throws ExceptionPositionInvalid;

//删除首元素，并返回之
  public Object removeFirst();

//删除末元素，并返回之
  public Object removeLast();

//将处于给定位置的元素替换为新元素，并返回被替换的元素
  public Object replace(Position p, Object e)
  throws ExceptionPositionInvalid;

//位置迭代器
  public Iterator positions();

//元素迭代器
```

```
  public Iterator elements();
}
```

代码三.6 列表ADT的Java接口

■ 迭代器

你或许注意到了上述ADT中的positions()和elements()方法，它们分别是位置和元素的迭代器，对此我们将在第 §3.4 节详细讨论。

3.2.4 基于双向链表实现的列表

在第 2.5.3 节中，为了实现双向链表结构，我们曾经基于位置ADT实现了双向链表节点类型DLNode（代码二.16）。在那里，每一节点对应于一个位置，而getElem()方法只需返回该节点所保存的元素。也就是说，节点本身就担当了位置的角色：在内部，链表将其视为节点；而从外部看来，它们都是一般意义上的位置。在内部，每一节点v都拥有prev和next引用，分别指向v的直接前驱与直接后继（注意：只要在前端、后端分别设置头、尾哨兵节点，它们就必然存在）。不过，同样地按照封装的思想，那里并没有直接使用变量prev和next，而是通过方法getPrev()、getNext()、setPrev()和setNext()间接地对它们进行访问或修改。如此实现的位置ADT，不仅完全符合面向对象的规范，而且时间、空间复杂度均不会增加。

下面，我们将利用 DLNode 类来实现列表 ADT。给定列表 S 中的任一位置 p，我们只要将 p“展开”，即可进而取出其对应的节点 v——具体来说，就是将 p 强制转换为节点 v。反之亦然，比如，给定任一节点 v，我们都可以利用 v.getNext()来实现方法 getNext(p)（当然，若 v 恰好是末节点，则需报错）。

在给出完整的列表 ADT 实现之前，让我们以 insertBefore(p, e)方法和 remove(p)为例，对各方法的算法实现做一介绍。

■ insertBefore(p, e)算法

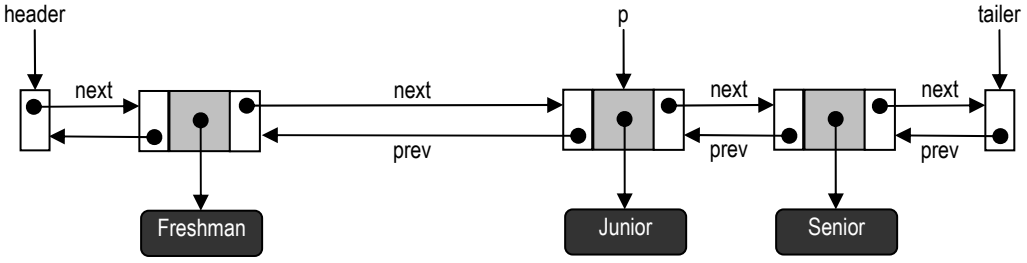
首先，为了将元素e作为p的直接前驱插入列表，我们可以首先创建一个节点v来存放e，然后将v插入至要求的位置，最后调整其直接前驱与后继的next和prev引用。具体的过程可以描述为 算法三.1。

```
算法：insertBefore(p, e)
输入：位置p和元素e
输出：将e插入至p之前后，返回插入的位置
{
  创建一个新节点v;
  v.setElement(e);//用v来存放e
  v.setNext(p);//以p为v的直接后继
  v.setPrev(p.getPrev);//以p的直接前驱为v的直接前驱
  (p.getPrev()).setNext(v);//p的直接前驱应该以v为直接后继
```

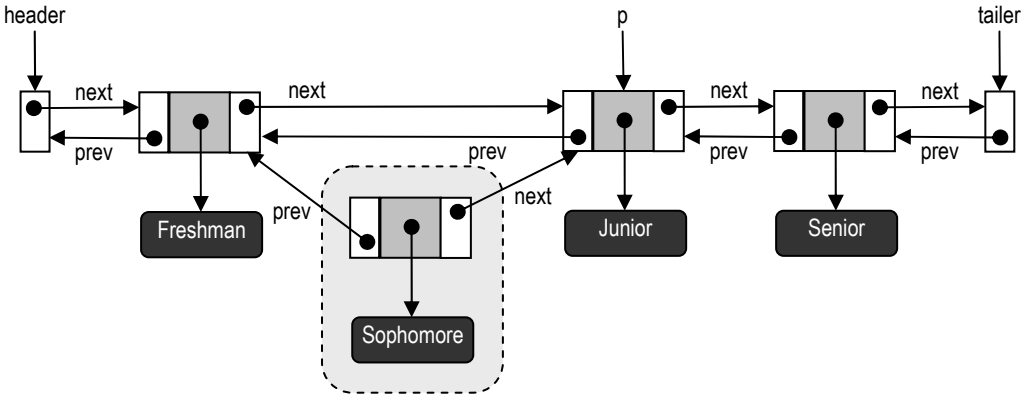
```
  p.setPrev(v);//作为v的直接后继，p应该以v为直接前驱
  return((Position)v);//v作为一个位置被返回
}
```

算法三.1 insertBefore() 算法

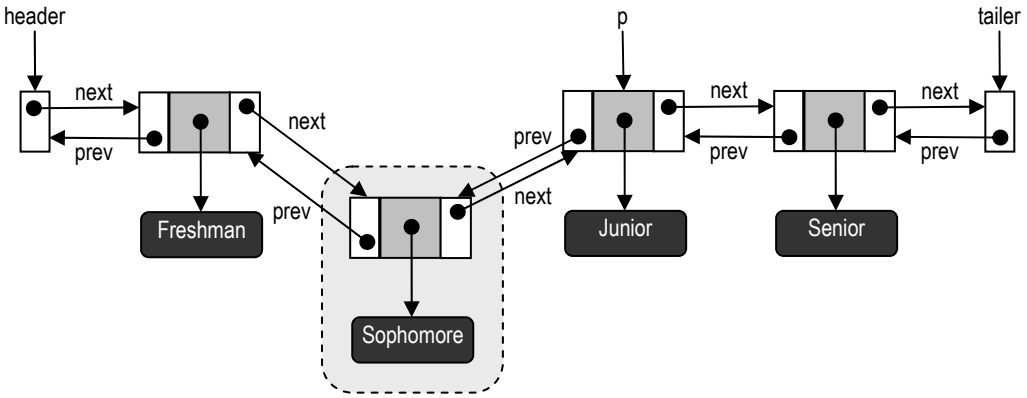
图三.3 则给出了一次insertBefore()操作实例。



(a) 准备在原列表的节点"Junior"之前插入新节点"Sophomore"



(b) 生成新节点以存放新元素，并将其链入列表



(c) 将新节点直接前驱的 next 引用以及直接后继的 prev 引用指向新节点

图三.3 在p = "Junior"之前插入新元素"Sophomore"

方法 insertAfter()、insertFirst()和 insertLast()都可以仿照这一算法来实现。

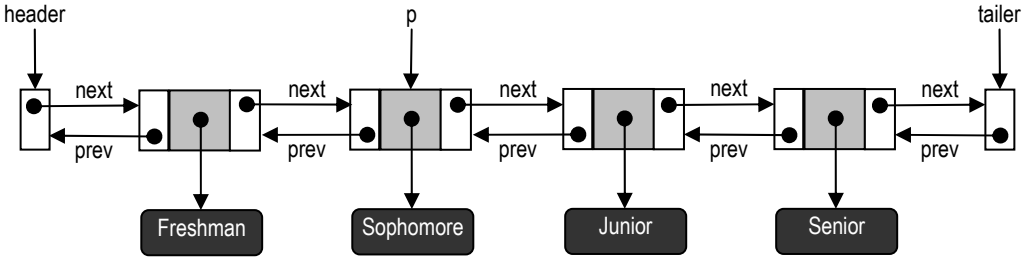
■ remove(p) 算法

下面再介绍方法remove(p)的实现。为了删除存放于位置p的元素e，我们可以将p的直接前驱与直接后继相互链接起来，从而将p剔除出去。一旦p不再被任何引用指向，内存回收器就会自动将其空间回收。具体的算法如 算法三.2 所示。

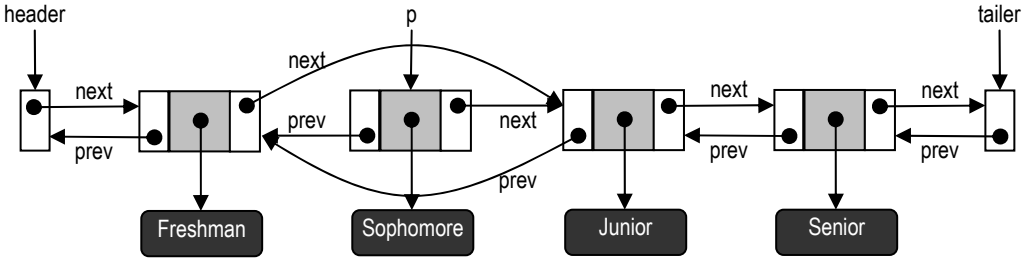
```
算法：remove(p)
输入：位置p
输出：删除存放于位置p处的元素e，并返回该元素
{
  bak = p.element;//复制待删除的元素
  (p.getNext()).setPrev(p.getPrev);//令p的直接后继以p的直接前驱为直接前驱
  (p.getPrev()).setNext(p.getNext);//令p的直接前驱以p的直接后继为直接后继
  p.setNext(null);//切断p的后继引用
  p.setPrev(null);//切断p的前躯引用
  return bak;//返回先前保留的备份
}
```

算法三.2 remove() 算法

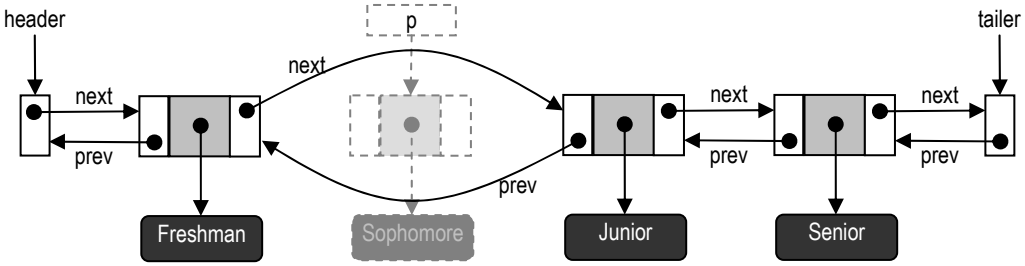
图三.4 则给出了remove()操作的一个实例。



(a) 原列表，待删除节点位置由 p 指示



(b) 将待删除节点的直接前驱与直接后继相互链接，从而剔除该节点



(c) 剔除出来的节点被内存回收器回收

图三.4 删除存放"Sophomore"的节点

得益于哨兵节点的设置，我们的算法可以不用考虑各种退化情况。请读者自行验证：无论是对于第一个位置、最后一个位置还是空列表，上述算法都能正确运行。

■ 算法效率

不难看出，基于双向链表实现的所有列表 ADT 方法，都可以在常数时间内完成——这已经是最优的了，不可能指望有比这更好的结果。

■ List_DLNod e 类——List 接口的实现

现在，我们已经可以给出基于双向链表的列表的完整实现，如 代码三.7 所示：

```
/*
 * 基于双向链表实现列表结构
 */
```

```
package dsa;

public class List_DLNode implements List {
  protected int numElem;//列表的实际规模
  protected DLNode header, trailer;//哨兵：首节点+末节点

//构造函数
  public List_DLNode() {
    numElem = 0;//空表
    header = new DLNode(null, null, null);//首节点
    trailer = new DLNode(null, header, null);//末节点
    header.setNext(trailer);//首、末节点相互链接
  }

/***************************** 辅助方法 *****************************/
//检查给定位置在列表中是否合法，若是，则将其转换为*DLNode
  protected DLNode checkPosition(Position p) throws ExceptionPositionInvalid {
    if (null == p)
      throw new ExceptionPositionInvalid("意外：传递给List_DLNode的位置是null");
    if (header == p)
      throw new ExceptionPositionInvalid("意外：头节点哨兵位置非法");
    if (trailer == p)
      throw new ExceptionPositionInvalid("意外：尾结点哨兵位置非法");
    DLNode temp = (DLNode)p;
    return temp;
  }

/***************************** ADT方法 *****************************/
//查询列表当前的规模
  public int getSize() { return numElem; }

//判断列表是否为空
  public boolean isEmpty() { return (numElem == 0); }

//返回第一个元素（的位置）
  public Position first() throws ExceptionListEmpty {
    if (isEmpty())
      throw new ExceptionListEmpty("意外：列表空");
    return header.getNext();
  }

//返回最后一个元素（的位置）
  public Position last()  throws ExceptionListEmpty {
    if (isEmpty())
      throw new ExceptionListEmpty("意外：列表空");
```

```
    return trailer.getPrev();
  }

//返回紧靠给定位置之前的元素（的位置）

  public Position getPrev(Position p)
  throws ExceptionPositionInvalid, ExceptionBoundaryViolation {
    DLNode v = checkPosition(p);
    DLNode prev = v.getPrev();
    if (prev == header)
      throw new ExceptionBoundaryViolation("意外：企图越过列表前端");
    return prev;
  }

//返回紧接给定位置之后的元素（的位置）
  public Position getNext(Position p)
  throws ExceptionPositionInvalid, ExceptionBoundaryViolation {
    DLNode v = checkPosition(p);
    DLNode next = v.getNext();
    if (next == trailer)
      throw new ExceptionBoundaryViolation("意外：企图越过列表后端");
    return next;
  }

//将e插入至紧靠给定位置之前的位置
  public Position insertBefore(Position p, Object element)
  throws ExceptionPositionInvalid {
    DLNode v = checkPosition(p);
    numElem++;
    DLNode newNode = new DLNode(element, v.getPrev(), v);
    v.getPrev().setNext(newNode);
    v.setPrev(newNode);
    return newNode;
  }

//将e插入至紧接给定位置之后的位置
  public Position insertAfter(Position p, Object element)
  throws ExceptionPositionInvalid {
    DLNode v = checkPosition(p);
    numElem++;
    DLNode newNode = new DLNode(element, v, v.getNext());
    v.getNext().setPrev(newNode);
    v.setNext(newNode);
    return newNode;
  }

//将e作为第一个元素插入列表
  public Position insertFirst(Object e) {
    numElem++;
    DLNode newNode = new DLNode(e, header, header.getNext());
```

```
    header.getNext().setPrev(newNode);
    header.setNext(newNode);
    return newNode;
  }

//将e作为最后一个元素插入列表

  public Position insertLast(Object e) {
    numElem++;
    DLNode newNode = new DLNode(e, trailer.getPrev(), trailer);
    if (null == trailer.getPrev())  System.out.println("!!!Prev of trailer is
NULL!!!");
    trailer.getPrev().setNext(newNode);
    trailer.setPrev(newNode);
    return newNode;
  }

//删除给定位置处的元素，并返回之
  public Object remove(Position p)
  throws ExceptionPositionInvalid {
    DLNode v = checkPosition(p);
    numElem--;
    DLNode vPrev = v.getPrev();
    DLNode vNext = v.getNext();
    vPrev.setNext(vNext);
    vNext.setPrev(vPrev);
    Object vElem = v.getElem();
  //将该位置（节点）从列表中摘出，以便系统回收其占用的空间
    v.setNext(null);
    v.setPrev(null);
    return vElem;
  }

//删除首元素，并返回之
  public Object removeFirst()
  { return remove(header.getNext()); }

//删除末元素，并返回之
  public Object removeLast()
  { return remove(trailer.getPrev()); }

//将处于给定位置的元素替换为新元素，并返回被替换的元素
  public Object replace(Position p, Object obj)
  throws ExceptionPositionInvalid {
    DLNode v = checkPosition(p);
    Object oldElem = v.getElem();
    v.setElem(obj);
    return oldElem;
  }
```

```
//位置迭代器
  public Iterator positions()
  { return new IteratorPosition(this); }

//元素迭代器
  public Iterator elements()
  { return new IteratorElement(this); }

}
```

代码三.7 Lis t_DLNode类——基于双向链表实现列表ADT

§3.3 序列

本节将定义并实现一个通用的序列 ADT，从而将向量 ADT 与列表 ADT 中的所有方法集成起来——也就是说，这一 ADT 将同时支持以秩或位置来访问其中的元素。因此，这种数据结构将适用于解决更多的实际问题。

3.3.1 序列 ADT

如上所言，序列ADT将同时支持向量ADT（第 §3.1 节）与列表ADT（第 §3.2 节）中的所有方法，此外，还支持以下两个方法——正是借助这两个方法，我们才得以将秩和位置的概念联系起来：

表三.9 序列ADT支持的操作

操作方法	功能描述
rank2Pos(r):	若 0 ≤ r < getSize()，则返回秩为 r 的元素所在的位置；否则，报错 输入：一个（作为秩的）整数 输出：位置
pos2Rank(p):	若 p 是序列中的合法位置，则返回存放于 p 处的元素的秩；否则，报错 输入：一个位置 输出：（作为秩的）整数

- 通过多重继承定义序列 ADT

我们可以采用Java的多重继承技术，通过已有的向量ADT和列表ADT来定义序列ADT，从而使之继承这两个父ADT的所有方法。具体的定义如 代码三.8 所示：

```
/*
 * 序列接口
 */
```

```
package dsa;

public interface Sequence extends Vector, List {
//若0 <= r < getSize()，则返回秩为r的元素所在的位置；否则，报错
  public Position rank2Pos(int r)
  throws ExceptionBoundaryViolation;

//若p是序列中的合法位置，则返回存放于p处的元素的秩；否则，报错
  public int pos2Rank(Position p)
  throws ExceptionPositionInvalid;

}
```

代码三.8 通过多重继承定义的序列ADT

3.3.2 基于双向链表实现序列

实现序列最自然、最直接的方式，就是利用双向链表。这样，来自列表 ADT 的每个方法都依然可以保持原先 $\mathcal{O}(1)$的时间复杂度。当然，来自向量 ADT 的方法尽管也可以借助双向链表来实现，但其效率却会受到影响。实际上，如果希望保持原列表 ADT 中各方法的高效率（即通过位置类来确定操作元素的位置），就不可能显式地在序列中保留和维护各元素的秩。为了完成诸如 getAtRank(r) 之类的操作，我们不得不从列表的某一端起逐一扫描各个元素，直到发现秩为 r 的元素。于是，在最坏情况下，这些操作的时间复杂度将注定为$\Omega(n)$。

具体的实现如 代码三.9 所示。

```
/*
 * 基于双向链表实现序列
 */

package dsa;

public class Sequence_DLNode extends List_DLNode implements Sequence {
//检查秩r是否在[0, n)之间
  protected void checkRank(int r, int n)
  throws ExceptionBoundaryViolation {
    if (r < 0 || r >= n)
      throw new ExceptionBoundaryViolation("意外：非法的秩" + r + "，应该属于[0, " + n +")");
  }

//若0 <= r < getSize()，则返回秩为r的元素所在的位置；否则，报错--O(n)
  public Position rank2Pos(int r) throws ExceptionBoundaryViolation {
    DLNode node;
    checkRank(r, getSize());
    if (r <= getSize()/2) {//若秩较小，则
      node = header.getNext( );//从前端开始
```

```
      for (int i=0; i<r; i++) node = node.getNext();//逐一扫描
    } else {//若秩较大，则
      node = trailer.getPrev();//从后端开始
      for (int i=1; i<getSize()-r; i++) node = node.getPrev();//逐一扫描
    }
    return node;
  }

//若p是序列中的合法位置，则返回存放于p处的元素的秩；否则，报错--O(n)
  public int pos2Rank(Position p) throws ExceptionPositionInvalid {
    DLNode  node = header.getNext();
    int   r = 0;

    while (node != trailer) {

      if (node == p)  return(r);
      node = node.getNext();  r++;
    }
    throw new ExceptionPositionInvalid("意外：作为参数的位置不属于序列");
  }

//取秩为r的元素--O(n)
  public Object getAtRank(int r) throws ExceptionBoundaryViolation {
    checkRank(r, getSize());
    return rank2Pos(r).getElem();
  }

//将秩为r的元素替换为obj--O(n)
  public Object replaceAtRank(int r, Object obj) throws ExceptionBoundaryViolation {
    checkRank(r, getSize());
    return replace(rank2Pos(r), obj);
  }

//插入obj，作为秩为r的元素--O(n)；返回该元素
  public Object insertAtRank(int r, Object obj) throws ExceptionBoundaryViolation {
    checkRank(r, getSize()+1);
    if (getSize() == r) insertLast(obj);
    else                insertBefore(rank2Pos(r), obj);
    return obj;
  }

//删除秩为r的元素--O(n)
  public Object removeAtRank(int r) throws ExceptionBoundaryViolation {
    checkRank(r, getSize());
    return remove(rank2Pos(r));
  }
}
```

代码三.9 基于双向链表实现的序列

当 r ∉ [0, n)时，其中的 checkRank(r)方法会扔出 ExceptionBoundaryViolation 意外。

rank2Pos(r)做了很小的优化：首先判断秩为 r 的元素究竟处于序列的前半段还是后半段，然后分别从前端或后端开始扫描。然而正如此前所分析的那样，在最坏情况下，方法 rank2Pos(r)的时间复杂度依然为 Ω(n)。由于 getAtRank(r)、replaceAtRank(r, obj)、insertAtRank(r, obj)和 removeAtRank(int r)等方法都利用了 rank2Pos(r)方法，故它们的时间复杂度也是 $\mathcal{O}(n)$。

3.3.3 基于数组实现序列

与链表相对称地，我们也可以利用数组来实现序列——将序列 S 中的各个元素分别存放到数组 A[]对应的单元中。具体地，每一位置对象除了设有一个引用指向数组 A[]，同时还保存一个下标 i。这就使得 getElem(p)方法可以直接返回 A[i]。这种实现也有不足：A[]中的单元无法直接得到各自对应的位置。比如，在执行 insertFirst()操作时，我们将无法通知 S 中相关的那些位置，以便它们将各自的秩加一（你应该还记得，序列中的位置总是相对于其前后邻居而言的，而不是它们的秩）。因此，为了利用数组来实现一个通用的序列结构，我们只能另想办法。

一种可行的办法是：不再将S中的元素直接存放在A[]的各单元中，而是代之以另一种位置对象；而真正的元素，则被存放到各个位置中。如 图三.5 所示，每个这样的位置对象p不仅存有某一下标i，而且还存放了与p对应的元素e。

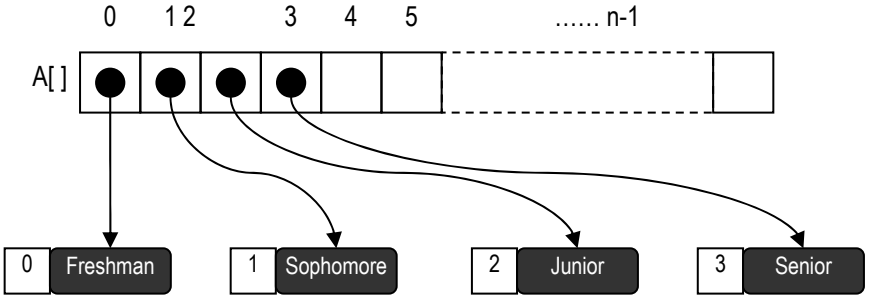


图 三.5 基于数组实现的序列

采用这种数据结构，在插入或删除操作之后，我们只需扫描一遍数组，即可找到需要修正秩的那些位置，并将其秩加一。

具体实现，请读者自行完成。

■ 性能分析

按照上述方式实现序列，一旦序列有所变动，就需要将有关的位置对象顺次移动，而且在最坏情况下需要移动 $\mathcal{O}(n)$个对象，因此 insertFirst()、insertBefore()、insertAfter()和 remove()方法都需要耗费 $\mathcal{O}(n)$时间。其它那些基于位置的操作则只需要 $\mathcal{O}(1)$的时间。

§3.4 迭代器

在对向量、列表和序列进行处理（比如，查找某一特定的元素）时，一种典型的操作就是依次访问或修改其中的各个元素。迭代器是软件设计的一种模式，它是对“逐一访问所有元素”这类操作的抽象。

迭代器本身也是一个序列S，在任何时候，迭代器中都有唯一的当前元素。迭代器还必须提供某种机制，使得我们可以不断转向S中的下一元素，并将其置为新的当前元素。与第 §3.2 节所介绍的位置ADT相比，迭代器是对前者的扩展与推广。实际上，一个位置本身已经可以被看作是一个迭代器了，只不过它还不能不断更新。总之，所谓迭代器，就是对一组对象之间的位置、直接后继等关系的集成。

3.4.1 简单迭代器的 ADT

迭代器 ADT 必须支持以下两个方法：

表三.10 迭代器ADT支持的操作

操作方法	功能描述
hasNext():	检查迭代器中是否还有剩余的元素 输入：无 输出：布尔标志
getNext():	返回迭代器中的下一元素 输入：无 输出：对象

请注意，在对序列进行遍历的过程中，迭代器 ADT 始终维护了一个“当前”元素。只要迭代器非空，那么对 getNext()方法的首次调用就会给出最初始的当前元素。

迭代器提供了一种统一的机制，无论对象之间的具体组织形式如何，它都可以从一组对象中为我们逐一列举出每个元素。对于向量、列表或序列之类具有线性次序的结构，迭代器还必须能够按照这一次序给出其中的元素。

■ Jav a 中的简单迭代器

Java 已经通过 java.util.Iterator 接口提供了一个迭代器。这一接口还有一个额外的功能——一旦转向新的对象，就将此前的对象从集合中删除。这一通过迭代器删除对象的方式值得商榷，毕竟它有悖于面向对象的原则，因此这一功能是否真正实现是可以选择的。

稍早前，Java 还曾提供过另一个 java.util.Enumeration 接口，其中对应的方法名分别为 hasMoreElements()和 nextElement()。

■ 在列表和其它 ADT 中引入迭代器

为了支持上述对数据结构遍历的统一机制，对象集合的 ADT 必须提供下面的方法：

表三.11 对象集合ADT支持的操作

操作方法	功能描述
elements():	返回集合中所有元素的一个迭代器 输入：无 输出：迭代器

比如，java.util.Vector就支持这一方法，只不过其返回值是一个Enumeration对象。代码三.10给出了java.util.Iterator的应用实例，这个例子利用iterator()方法，将指定的java.util.Vector向量中的所有元素逐一打印输出。这里的iterator()方法与对象集合ADT中的elements()方法功能类似，只不过返回的是一个java.util.Iterator。

```
public static void printVector(java.util.Vector vec) {
  java.util.Iterator  it = vec.iterator();
  while (it.hasNext())
    System.out.println(it.getNext());

}
```

代码三.10 Ja va迭代器的应用实例：打印输出向量中的所有元素

对于列表、序列之类支持位置概念的 ADT，我们还提供如下方法：

表三.12 支持位置的ADT需要提供的方法

操作方法	功能描述
positions():	返回集合中所有位置的一个迭代器 输入：无 输出：迭代器

当我们需要逐一枚举某一容器中的所有位置时，利用这一方法将再好不过。

此前，我们已经在向量ADT（第 3.1.1 节）和列表ADT（第 3.2.3 节）中提供了positions()和elements()方法，之所以这么做，正是为了确保列表能够提供对上述功能的支持。

3.4.2 迭代器接口

综上所述，Iterator接口可以描述为 代码三.11。

```
/*
 * 迭代器ADT接口
 */

package dsa;

public interface Iterator {
  boolean hasNext();//检查迭代器中是否还有剩余的元素
```

```
  Object  getNext();//返回迭代器中的下一元素
}
```

代码三.11 迭代器ADT接口

3.4.3 迭代器的实现

■ 利用快照建立迭代器

实现迭代器的一种直接办法，就是给容器中的所有元素照张“快照”，并用某一数据结构将其记录下来，当然，这种数据结构必须支持后续对这些元素的遍历操作。比如，可以借助栈结构来存放“快照”——将所有元素（的引用）压入某一栈中。于是，!isEmpty()方法就等效于 hasNext()方法，而 pop()操作则等效于 getNext()操作。如果元素之间已经定义了某种次序，我们可以按照这一次序的相反次序来压栈，这样就可以保证此后按照指定的次序遍历所有的元素。同理，也可以借助队列结构来实现迭代器，当然，此时直接让所有元素按照原次序入队即可。显然，这两种实现方式都需要对所有元素访问一遍，因此若原容器的规模为 n，则 positions()和 elements()方法均需要 $\mathcal{O}(n)$时间。

■ IteratorPositi on()的实现

在 代码三.12 中，我们采用另一种方式——借助列表类本身——来实现IteratorPosition()。这种实现只需保留并跟踪迭代器的当前元素，因此无论是构造方法还是hasNext()和getNext()方法，都可以在$\mathcal{O}(1)$时间内完成。

```
/*
 * 基于列表实现的位置迭代器
 */

package dsa;

public class IteratorPosition implements Iterator {
  private List list;//列表
  private Position nextPosition;//当前（下一个）位置

//默认构造方法
  public IteratorPosition() { list = null; }

//构造方法
  public IteratorPosition(List L) {
    list = L;
    if (list.isEmpty())//若列表为空，则
      nextPosition = null;//当前位置置空
    else//否则
      nextPosition = list.first();//从第一个位置开始
  }
```

```
//检查迭代器中是否还有剩余的位置
  public boolean hasNext() { return (nextPosition != null); }

//返回迭代器中的下一位置
  public Object getNext() throws ExceptionNoSuchElement {
    if (!hasNext()) throw new ExceptionNoSuchElement("意外: 没有下一位置");
    Position currentPosition = nextPosition;
    if (currentPosition == list.last())//若已到达尾位置，则
      nextPosition = null;//不再有下一个位置
    else//否则
      nextPosition = list.getNext(currentPosition);//转向下一位置
    return currentPosition;
  }
}
```

代码三.12 基于列表实现位置迭代器

■ IteratorElement() 的实现

按照类似的思路，也可以实现IteratorElement，参见 代码三.13。

```
/*
 * 基于列表实现的元素迭代器
 */

package dsa;

public class IteratorElement implements Iterator {
  private List list;//列表
  private Position nextPosition;//当前（下一个）元素的位置

//默认构造方法
  public IteratorElement() { list = null; }

//构造方法
  public IteratorElement(List L) {
    list = L;
    if (list.isEmpty())//若列表为空，则
      nextPosition = null;//当前元素置空
    else//否则
      nextPosition = list.first();//从第一个元素开始
  }

//检查迭代器中是否还有剩余的元素
  public boolean hasNext() { return (null != nextPosition); }
```

```
//返回迭代器中的下一元素
  public Object getNext() throws ExceptionNoSuchElement {
    if (!hasNext()) throw new ExceptionNoSuchElement("意外：没有下一元素");
    Position currentPosition = nextPosition;
    if (currentPosition == list.last())//若已到达尾元素，则
      nextPosition = null;//不再有下一元素
    else//否则
      nextPosition = list.getNext(currentPosition);//转向下一元素
    return currentPosition.getElem();
  }
}
```

代码三.13 基于列表实现元素迭代器

3.4.4 Java 中的列表及迭代器

在使用迭代器的过程中，如果原容器中的内容正在被（比如另一个线程）修改，就很可能会造成危险的后果。若需在容器中的某一“位置”进行插入、删除或替换之类的操作，最好是通过一个位置对象来指明。实际上，java.util.Iterator 的大多数实现都提供了故障快速修复（Fail-fast）的机制——在利用迭代器遍历某一容器的过程中，一旦发现该容器的内容有所改变，迭代器就会抛出 ConcurrentModificationException 意外错并立刻退出。

java.util 包中 LinkedList 类的 API 接口，并没有向程序员提供位置这一概念。实际上，Java 对 LinkedList 对象的访问与更新并不是通过秩完成的，而是更倾向于另一种方式——通过 listIterator() 方法，创建一个 ListIterator 迭代器。该迭代器不仅支持向前、向后的遍历，同时还支持对局部内容的更新。按照这一迭代器的习惯，一个位置要么紧靠在首元素之前，要么夹在相邻的两个元素之间，要么紧跟在末元素之后。如此定义的位置很像屏幕上的光标（Cursor），后者总是夹在一对相邻的字符之间。具体来说，java.util.ListIterator 接口由以下方法组成：

表三.13 java.ut il.ListIterator接口定义的操作

操作方法	功能描述
add(e)：	在迭代器的当前位置插入元素 e
hasNext()：	当迭代器的当前位置之后存在某个元素时，返回 true；否则，返回 false
hasPrevious()：	当迭代器的当前位置之前存在某个元素时，返回 true；否则，返回 false
previous()：	返回紧靠于当前位置之前的元素 e，并将当前位置移至紧靠于 e 之前
getNext()：	返回紧跟于当前位置之后的元素 e，并将当前位置移至紧跟于 e 之后
nextIndex()：	返回下一元素的下标（秩）
previousIndex()：	返回前一元素的下标（秩）
set(e)：	将由刚才的 getNext()或 previous()操作所返回的元素替换为 e
remove()：	删除由刚才的 getNext()或 previous()操作所返回的元素

在 Java 中，可以通过多个迭代器同时对同一链表进行遍历。不过，正如上面所提到的，一旦其中某个迭代器修改了链表的内容，所有的迭代器都会成为非法的。

■ j ava.util.List 接口及其实现

java.util.List 接口所提供的功能，与我们的序列 ADT 类似，只不过 java.util 中的 ArrayList 类和 Vector 类都是基于数组实现的，而 LinkedList 类则是基于链表实现的。当然，这两种实现各有利弊，在解决实际问题时，我们需要在二者之间做一权衡。此外，Java 也通过迭代器提供了一些功能，它们与我们的序列 ADT 利用位置所实现的功能相仿。

第四章

树

在 第二章和 第三章中，我们已经学习了一些数据结构，根据其实现方式，这些数据结构可以划分为两种类型：基于数组的实现与基于链表的实现。正如我们已经看到的，就其效率而言，这两种实现方式各有长短。具体来说，基于数组实现的结构允许我们通过下标或秩，在常数的时间内找到目标对象，并读取或更新其内容。然而，一旦需要对这类结构进行修改，那么无论是插入还是删除，都需要耗费线性的时间。反过来，基于链表实现的结构允许我们借助引用或位置对象，在常数的时间内插入或删除元素；但是为了找出居于特定次序的元素，我们不得不花费线性的时间对整个结构进行遍历查找。能否将这两类结构的优点结合起来，并回避其不足呢？本章将要介绍的树结构，将正面回答这一问题。

实际上，从更广的角度来看，此前介绍的结构都属于所谓的线性结构（Linear structures），亦即，在其中各元素之间存在一个自然的线性次序。树结构则不然，其中的元素之间并不存在天然的直接后继或直接前驱关系，因此属于非线性结构（Non-linear structures ）。不过，正如我们马上就要看到的，只要附加上某种约束（比如遍历），也可以在树结构中的元素之间确定某种线性次序，因此也有人称之为半线性结构（Semi-linear structures）。

无论如何，随着从线性结构转入树结构，我们的思维方式也将有个飞跃。只有在完成思维方式的这一转换之后，我们才能够更加有效地解决更多的基本算法问题。以 第七章将要介绍的平衡二分查找树为例，如果其中包含n个元素，则每一次查找、更新、插入或删除操作都可以在$\mathcal{O}(\log n)$[1]的时间内完成，而每次遍历都可以在$\mathcal{O}(n)$的时间内完成。

就其内部组成而言，树是一种分层结构。树结构之所以在算法理论与实际应用中始终都扮演着最关键的角色，并且有着不计其数的变种，其实并不足为怪——层次化的概念几乎蕴含于所有事物之中，乃是它们的本质属性之一。从文件系统、Internet 的域名系统、数据库系统到人类社会系统，层次结构无所不在。

§4.1 术语及性质

尽管树不是线性结构，但是得益于其层次化的特点，我们还是可以说“树的某一元素是另一元素‘直接上邻’”，也可以说“某一元素是另一元素的‘直接下邻’之一”。这些关系可以形象地用“父亲”、“孩子”、“祖先”和“后代”等术语来描述，下面我们首先对此做一介绍。

[1] 不难证明，对任意的 c>0，都有 $\log n = \mathcal{O}(n^c)$。也就是说，就多项式而言 $\mathcal{O}(\log n)$与 $\mathcal{O}(1)$可以无限接近，而 $\mathcal{O}(\log n)$与 $\mathcal{O}(n)$相比几乎提高了一个线性因子。因此就这一点而言，树结构的确能够将数组和链表的优点结合起来。

4.1.1 节点的深度、树的深度与高度

作为一种抽象数据类型，树可以用来对一组元素进行层次化的组织。本书所谈论的树，其实不过是一种特殊的图结构（参见 第十章），因此，树中的元素也称作节点（Node）。此外，按照如下规则，树中的每个节点v都被赋予了一个特殊的指标——深度，记作depth(v)：

定义四.1 在树结构中，

⓪ **每个节点的深度都是一个非负整数；**

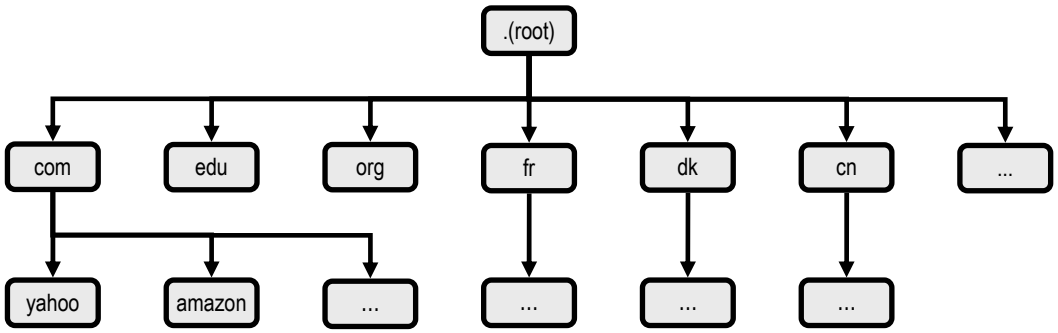
① **深度为 0 的节点有且仅有一个，称作树根（Root）；**

② **对于深度为 k (k≥1)的每个节点 u，都有且仅有一个深度为 k-1 的节点 v 与之对应，称作 u 的父亲（Parent）或父节点。**

定义四.2 若节点 v 是节点 u 的父亲，则 u 称作 v 的孩子（Child），并在二者之间建立一条树边（Edge）。

请注意，尽管每个节点至多只有一个父亲，但却可能有多个孩子。同一节点的孩子互称“兄弟”（Sibling）。

另外，联接于父、子节点之间的边实际上是有方向的——从父亲指向孩子。



图四.1 Inter net的域名分布可以描述为一棵树

如 图四.1 所示，我们通常按照深度自上而下画出各个节点，并通过直线、折线或曲线将存在父子关系的节点联接起来。从这一形式看，这种结构的确像一棵上下颠倒、自上向下生长的树，故此得名。

定义四.3 树中所有节点的最大深度，称作树的深度或高度。

后面我们将会看到，从对最坏时间复杂度的衡量来看，树的深度是一项极为重要的指标。实际上，贯穿整个 第七章的一条主线，就是如何控制查找树的深度。

利用数学归纳法不难证明：

观察结论四.1 树中节点的数目，总是等于边数加一。

观察结论四.1 非常重要，它告诉我们：就渐进复杂度而言，树中边的总数与节点的总数相当。正是基于这一事实，在对涉及树结构的有关算法做复杂度分析时，我们可以用节点的数目来度量树结构本身的存储空间复杂度。

4.1.2 度、内部节点与外部节点

定义四.4 任一节点的孩子数目，称作它的“度”（Degree）。

请注意，节点的父亲不计入其度数。

定义四.5 至少拥有一个孩子的节点称作“内部节点”（Internal node）；没有任何孩子的节点则称作“外部节点”（External node）或“叶子”（Leaf）。

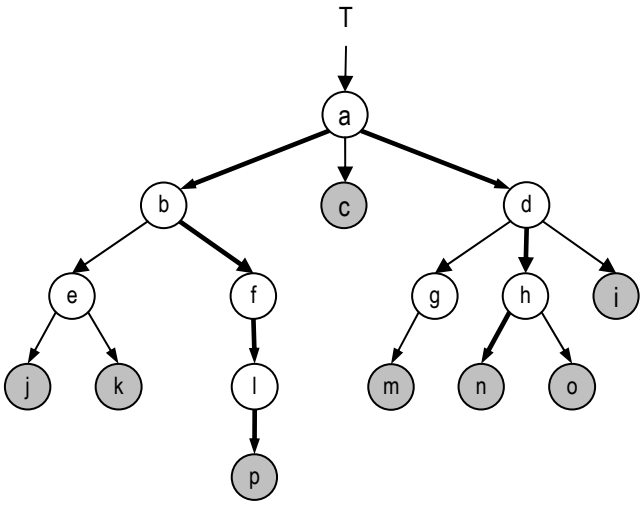
也就是说，一个节点是叶子，当且仅当它的度数为零。

4.1.3 路径

定义四.6 由树中 k+1 节点通过树边首尾衔接而构成的序列$\{ (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \mid k \geq 0\}$，称作树中长度为 k 的一条路径（Path）。

注意，这里忽略了边的方向。

特别地，由单个节点、零条边构成的路径也是合法的，其长度为 0。



图四.2 任意节点v和u之间都存在唯一的通路

以 图四.2 为例，{ (p, l), (l, f), (f, b), (b, a), (a, d), (d, h), (h, n) }构成了一条联接于节点p和n之间、长度为 7 的路径。

从 图四.2 也不难发现：

观察结论四.2 树中任何两个节点之间都存在唯一的一条路径。

这就意味着，树既是连通的，同时又不致出现环路。

特别地，在树根与每个节点之间，也存在唯一的一条路径。

根据前面关于节点深度的定义，不难得出如下结论：

观察结论四.3 若 v 是 u 的父亲，则 depth(v) + 1 = depth(u)。

由此，根据数学归纳法易知：

推论四.1 从树根通往任一节点的路径长度，恰好等于该节点的深度。

4.1.4 祖先、后代、子树和节点的高度

基于父子关系，我们还可以在节点之间递归地定义出“祖先/后代关系”如下：

定义四.7

0. **每个节点都是自己的“祖先”（Ancestor），也是自己的“后代”（Descendent）；**
1. **若 v 是 u 的父节点的祖先，则 v 也是 u 的祖先；**
2. **若 u 的父节点是 v 的后代，则 u 也是 v 的后代。**

请读者自己证明：树根节点是所有节点的祖先。

定义四.8 除节点本身以外的祖先（后代），称作真祖先（后代）。

还是以 图四.2 为例，节点a和b是f的真祖先，而l和p都是f的真后代，同时f既是自己的祖先也是自己的后代；根节点a是所有节点的祖先。

观察结论四.4 任一节点 v 的深度，等于其真祖先的数目。

观察结论四.5 任一节点 v 的祖先，在每一深度上最多只有一个。

否则，从节点v将有两条路径通向根节点，这与 观察结论四.2 相悖。

然而反过来，在同一深度上，某一节点的后代却有可能存在多个甚至多个。例如在 图四.2 中，节点g、h和i深度相同，它们都是节点d的后代。

定义四.9 树 T 中每一节点 v 的所有后代也构成一棵树，称作 T 的“以 v 为根的子树（Subtree）”。

需要特别指出的是，空节点（null）本身也构成一棵树，称作“空树”（Empty tree）。空树虽然不含任何节点，但却是任何树的（平凡）子树。

在上下文不致混淆的情况下，为使叙述简化，我们可将“以 v 为根的子树”直接称作“子树 v”。

定义四.10 若子树 v 的深度（高度）为 h，则称 v 的高度为 h，记作 height(v) = h。

特别地，根节点的高度就是整棵树的深度（高度）。

请读者自行证明以下事实：

观察结论四.6 对于叶子节点 u 的任何祖先 v，必有 $\text{depth}(v) + \text{height}(v) \geq \text{depth}(u)$。

在上下文不致混淆的情况下，我们可以用各节点中存放的具体数值或内容来直接指代该节点。以 图四.1 为例，我们可以说“节点yahoo是节点com的孩子”，或者“节点yahoo与节点amazon互为兄弟”，或者“节点com是节点amazon的祖先”，或者“子树com由节点com、yahoo、amazon等等组成”。

4.1.5 共同祖先及最低共同祖先

定义四.11在树 T 中，若节点 u 和 v 都是节点 a 的后代，则称节点 a 为节点 u 和 v 的共同祖先（Common ancestor）。

不难看出，根节点是所有节点的共同祖先，由此可知：

观察结论四.7 每一对节点至少存在一个共同祖先。

通常，一对节点可以有多个共同祖先。当然，根据 观察结论四.5，每一对节点在每一深度上至多只有一个共同祖先，我们更加关心的是其中深度最大的那个。

定义四.12 在一对节点 u 和 v 的所有共同祖先中，深度最大者称为它们的最低共同祖先（Lowerest common ancestor），记作 lca(u, v)。

与上面同理可知：

观察结论四.8 每一对节点的最低共同祖先必存在且唯一。

4.1.6 有序树、m 叉树

定义四.13 在树 T 中，若在每个节点的所有孩子之间都可以定义某一线性次序，则称 T 为一棵“有序树（Ordered tree）”。

对于有序树，我们可以明确定义每个节点的第一个孩子、第二个孩子、第三个孩子…。在画一棵有序树时，通常都按照这一次序自左向右地画出同一节点的所有孩子。

比如，每本书的大纲结构都对应于一棵树。该树的树根对应于整本书，其中的层次自上而下依次为章、节和小节，而每段文字、每张插图、各个表格和每个公式都可以表示为叶子节点。由于每一章（节）下属的所有节（小节）之间都有一个清晰的线性次序，所以这是一棵不折不扣的有序树。请读者参照目录，绘制出与本书大纲结构相对应的一棵有序树。

定义四.14 每个内部节点均为 m 度的有序树，称作 m 叉树。

其中，每个内部节点的 m 个孩子都可以依次从 1 到 m 编号。在画出这类树时，通常的习惯就是按照这种编号，将 m 个孩子自左向右地排列。

4.1.7 二叉树

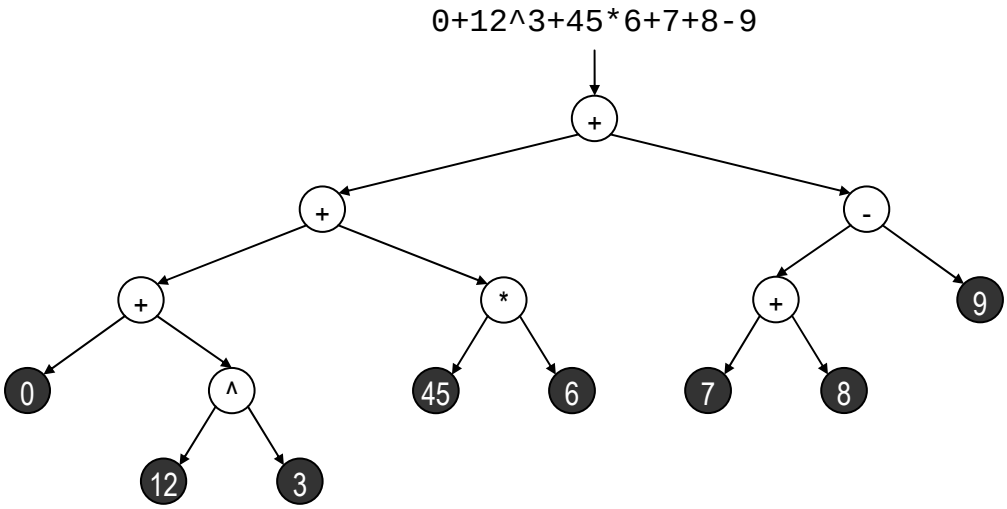
定义四.15 每个节点均不超过 2 度的有序树，称作二叉树（Binary tree）。

二叉树是最简单的非平凡 m 叉树，但却是讨论最多、应用最广的。

在二叉树中，每个节点的孩子可以用左、右区分，分别称作左孩子和右孩子。如果左、右孩子同时存在，则左孩子的次序优先于右孩子。

以节点 v 的左（右）孩子为根的子树，称作 v 的左（右）子树。

定义四.16 不含 1 度节点的二叉树，称作真二叉树（Proper bina ry tree ），否则称作非真二叉树（Improper binary tree）。



图四.3 表达式树：利用二叉树描述算术表达式

运用数学归纳法易证：

观察结论四.9 在二叉树中，深度为 k 的节点不超过 2^k 个。

由此可得如下的进一步结论：

推论四.2 高度为 h 的二叉树最多包含 2^{h+1}-1 个节点。

反之，

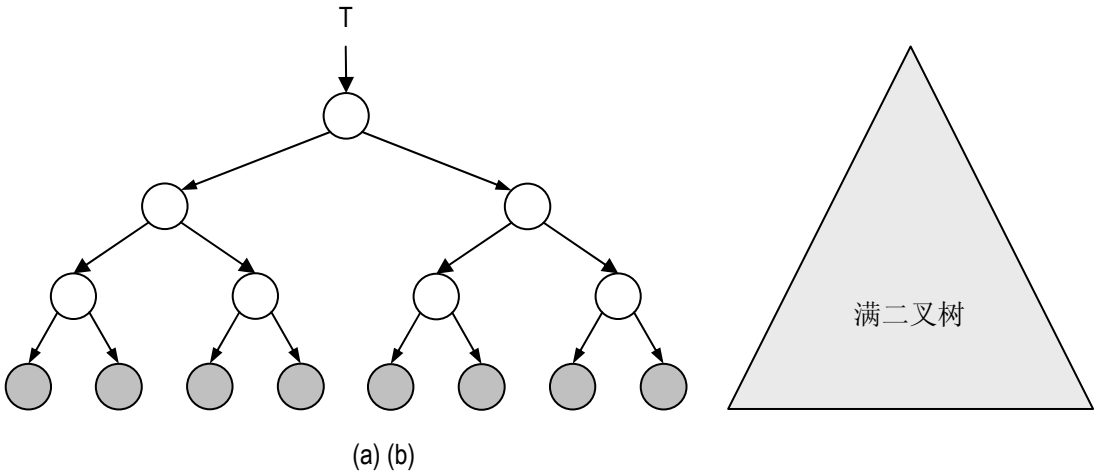
推论四.3 由 n 个节点构成的二叉树，高度至少为$\lfloor \log_2 n \rfloor$。

下面这一事实留给读者证明：

观察结论四.10在二叉树中，叶子总是比 2 度节点多一个。

4.1.8 满二叉树与完全二叉树

定义四.17 若二叉树 T 中所有叶子的深度完全相同，则称之为满二叉树（Full binary tree）。



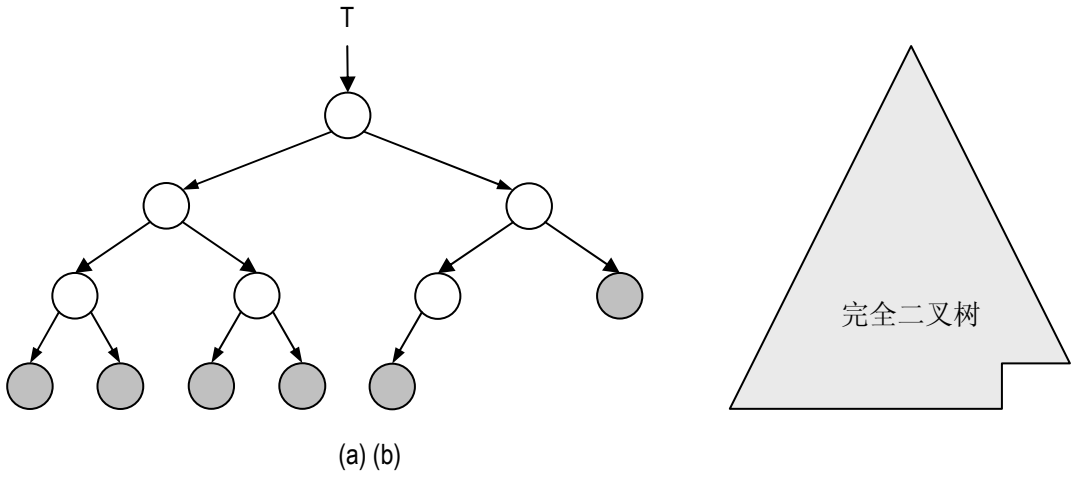
图四.4 (a) 高度为3的满二叉树，以及(b)满二叉树的宏观特征

如 图四.4 所示，就其宏观结构来看，满二叉树相当于一个完整的等腰三角形，所有叶子沿底边分布。

观察结论四.11 高度为 h 的二叉树是满的，当且仅当它拥有 2^h 匹叶子、2^{h+1}-1 个节点。

定义四.18 若在一棵满二叉树中，从最右侧起将相邻的若干匹叶子节点摘除掉，则得到的二叉树称作完全二叉树（Complete binary tree）。

完全二叉树的实例及其宏观结构，如 图四.5 所示。



图四.5 (a) 高度为3的完全二叉树，以及(b)完全二叉树的宏观特征

引理四.1 由 n 个节点构成的完全二叉树，高度 $h = \lfloor \log_2 n \rfloor$。

〖证明〗

任取由 n 个节点构成的一棵完全二叉树 T，设其高度为 h。则其中高度小于 h 的所有节点必然构成一棵满树，而且最底层的（叶子）节点至少有一个、至多有 2^h 个，故有

$$(2^h - 1) + 1 \leq n \leq (2^h - 1) + 2^h$$

$$2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$$

即

$$h \leq \log_2 n < h+1$$

也就是说，$\log_2 n \in [h, h+1)$，即 $h = \lfloor \log_2 n \rfloor$ ❑

结合 推论四.3，可以进而得到如下结论：

推论四.4 在由固定数目的节点所组成的所有二叉树中，完全二叉树的高度最低。

§4.2 树抽象数据类型及其实现

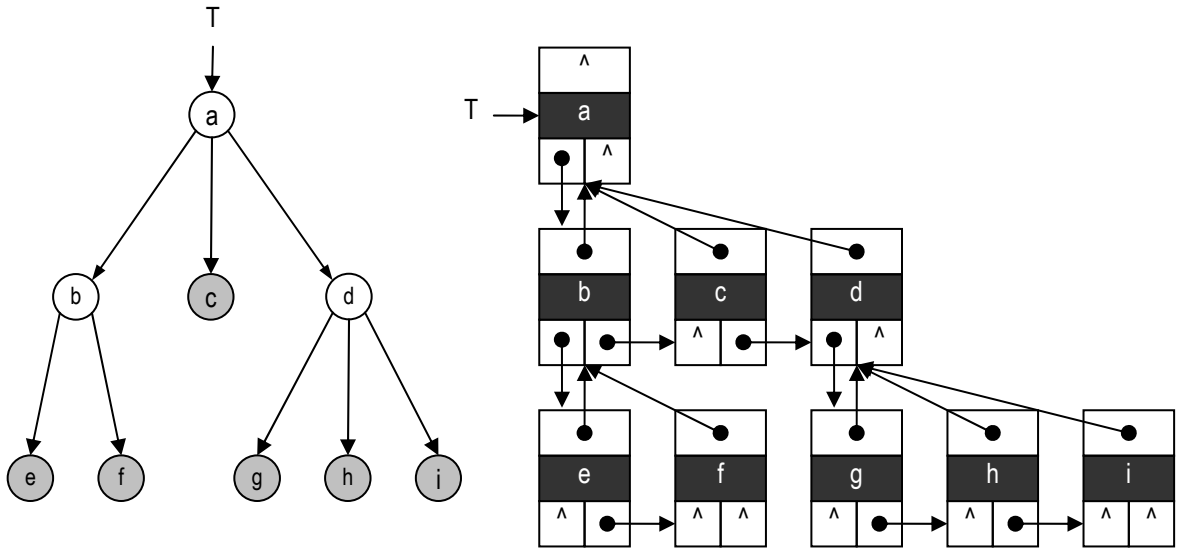
在 第二章、第三章中，我们的注意力主要集中于数据结构的实现，从树结构开始，我们讨论的重点将逐渐转入算法，因此将不再拘泥于面向对象的规范，而将更多的笔墨转向算法的实现及分析。

4.2.1 父亲-长子-弟弟”模型

根据树的定义，每个节点的所有后代均构成了一棵子树，故从数据类型的角度来看，树、子树以及树节点都是等同的。这里，将它们统一为一个类：Tree。

图四.6 基于“父亲-长子-弟弟”模型的树节点结构

这里将采用“父亲-长子-弟弟”模型来定义树ADT。如 图四.6 所示，在每个节点中除数据项外，还设有三个引用，分别指向该节点的父亲、长子和最大弟弟节点（若不存在，则为null）。



图四.7 基于“父亲-长子-弟弟”模型的树结构

如 图四.7 所示的就是一棵树，及其基于“父亲-长子-弟弟”模型的表示。

接下来，我们由“父亲-长子-弟弟”模型定义出树 ADT，然后给出其具体实现，并对相关算法逐一分析。

4.2.2 树 ADT

树抽象类型要支持以下的基本方法：

表四.1 树ADT支持的操作

操作方法	功能描述
getElement():	返回存放于当前节点处的对象 输入：无 输出：对象
setElement(e):	将对象 e 存入当前节点，并返回其中此前所存的内容 输入：一个对象 输出：对象
getParent():	返回当前节点的父节点 输入：无 输出：树节点
getFirstChild():	返回当前节点的长子 输入：无 输出：树节点
getNextSibling():	返回当前节点的最大弟弟 输入：无 输出：树节点

就对树的更新操作而言，不同的应用问题会要求树结构提供不同方法。这方面的差异太大，我们无法在树 ADT 中定义出通用的更新操作。在后续章节的讨论中，我们将结合各种应用问题，陆续给出一些具体的更新操作的实现。

4.2.3 树的 Java 接口

根据上述规范，代码四.1 给出了树ADT的Java接口：

```
/*
 * 树ADT接口
 */

package dsa;

public interface Tree {
//返回当前节点中存放的对象
  public Object getElem();
//将对象obj存入当前节点，并返回此前的内容
  public Object setElem(Object obj);

//返回当前节点的父节点
  public TreeLinkedList getParent();

//返回当前节点的长子
  public TreeLinkedList getFirstChild();

//返回当前节点的最大弟弟
  public TreeLinkedList getNextSibling();

//返回当前节点后代元素的数目，即以当前节点为根的子树的规模
  public int getSize();

//返回当前节点的高度
  public int getHeight();

//返回当前节点的深度
  public int getDepth();
}
```

代码四.1　树ADT的Java接口

4.2.4 基于链表实现树

如 代码四.2 所示，我们首先给出基于链表实现的树结构，其中相关的算法将陆续在后面讨论。

```
/*
 * 基于链表实现树结构
```

```
 */
```

```
package dsa;

public class TreeLinkedList implements Tree {
  private Object element;//树根节点
  private TreeLinkedList parent, firstChild, nextSibling;//父亲、长子及最大的弟弟

//（单节点树）构造方法
  public TreeLinkedList()
  { this(null, null, null, null); }

//构造方法
  public TreeLinkedList(Object e, TreeLinkedList p, TreeLinkedList c, TreeLinkedList
s) {
    element = e;
    parent = p;
    firstChild = c;
    nextSibling = s;
  }

/*---------- Tree接口中各方法的实现 ----------*/
//返回当前节点中存放的对象
  public Object getElem()
  { return element; }


//将对象obj存入当前节点，并返回此前的内容
  public Object setElem(Object obj)
  { Object bak = element; element = obj;  return bak; }

//返回当前节点的父节点；对于根节点，返回null
  public TreeLinkedList getParent()
  { return parent; }

//返回当前节点的长子；若没有孩子，则返回null
  public TreeLinkedList getFirstChild()
  { return firstChild; }

//返回当前节点的最大弟弟；若没有弟弟，则返回null
  public TreeLinkedList getNextSibling()
  { return nextSibling; }

//返回当前节点后代元素的数目，即以当前节点为根的子树的规模
  public int getSize() {
    int size = 1;//当前节点也是自己的后代
    TreeLinkedList subtree = firstChild;//从长子开始
```

```
    while (null != subtree) {//依次
      size += subtree.getSize();//累加
      subtree = subtree.getNextSibling();//所有孩子的后代数目

    }

    return size;//即可得到当前节点的后代总数
  }

//返回当前节点的高度
  public int getHeight() {
    int height = -1;
    TreeLinkedList subtree = firstChild;//从长子开始
    while (null != subtree) {//依次
      height = Math.max(height, subtree.getHeight());//在所有孩子中取最大高度
      subtree = subtree.getNextSibling();
    }
    return height+1;//即可得到当前节点的高度
  }

//返回当前节点的深度
  public int getDepth() {
    int depth = 0;
    TreeLinkedList  p = parent;//从父亲开始
    while (null != p) {//依次
      depth++;  p = p.getParent();//访问各个真祖先
    }
    return depth;//真祖先的数目，即为当前节点的深度
  }
}
```

代码四.2 基于链表的树实现

§4.3 树的基本算法

本节将详细分析第 4.2.4 节中所给出的相关算法。

4.3.1 getSize()——统计（子）树的规模

观察结论四.12一棵树的规模，等于根节点下所有子树规模之和再加一，也等于根节点的后代总数。

基于这一事实，该算法首先通过 firstChild 引用找出根节点的长子，并沿着 nextSibling 引用顺次找到其余的孩子，递归地统计出各子树的规模。最后，只要将所有子树的规模累加起来，再计入根节点本身，就得到了整棵树的规模。当遇到没有任何孩子的节点（即原树的叶子）时，递归终止。

如果不计入递归调用，该算法在每个节点上只需花费常数时间，因此若树的规模为 n，则总的时间复杂度为 $O(n)$。读者也许注意到了，实际上，这一算法也能够在 $O(n)$时间内统计出树中所有子树的规模。

4.3.2 getHeight()——计算节点的高度

另一基本操作是计算以v为根的子树高度。根据第 4.1.4 节的定义，不难得出如下推论：

推论四.5 ① 若 u 是 v 的孩子，则 height(v) ≥ height(u) + 1；

② $\text{height}(v) = 1 + \max_{u\text{是}v\text{的孩子}} \text{height}(u)$ 。

因此，算法 getHeight(v)也是首先通过 firstChild 引用找出根节点的长子，并沿着 nextSibling 引用顺次找到其余的孩子，递归地计算出各子树的高度。最后，只要找出所有子树的最大高度，再计入根节点本身，就得到了根节点的高度（即树高）。请读者自行分析递归终止的条件。

仿照对 getSize()方法的分析可知，getHeight(v)算法的运行时间也是 $O(n)$，其中 n 为 v 的后代总数（即树的规模）。实际上，花费这样多的时间，同样可以统计出所有子树的高度。

4.3.3 getDepth()——计算节点的深度

我们也时常需要计算给定节点v在树中的深度。根据第 4.1.1 节的定义，不难得出如下推论：

推论四.6 若 u 是 v 的孩子，则 depth(u) = depth(v) + 1。

根据这一事实，读者不难自行给出该算法的递归实现。不过，由于这是典型的尾递归（参见第 1.5.1 节），所以根据 观察结论四.4，这里直接改写为迭代形式。

具体地，算法 getDepth(v)将从 v 的父亲开始，沿着 parent 指针不断上移，直到深度为 0 的树根。在这个过程中所遇到的每个节点，都是 v 的真祖先；反之，在这一过程中，v 的每一真祖先迟早都会被找到。因此，根据总共上移的层数，就可以得到 v 在整棵树中的深度。

由于该算法只需访问 v 的所有真祖先，而且在每个节点只需 $O(1)$时间，故其复杂度为 $O(\text{depth}(v))$。在最坏情况下，由 n 个节点组成的树的高度可以达到 n。此时，若针对最底层的叶子调用 getDepth()算法，将需要 $O(n)$的时间。

4.3.4 前序、后序遍历

所谓树的遍历（Traversal），就是按照某种次序访问树中的节点，且每个节点恰好访问一次。也就是说，按照被访问的次序，可以得到由树中所有节点排成的一个序列。

在本节中，我们将分别介绍两种最基本的树遍历算法——前序遍历（Preoder trave rsal）和后序遍历（Postorder traver sal）。这两种遍历算法都是递归定义的，只是其中对“次序”的定义略有不同。

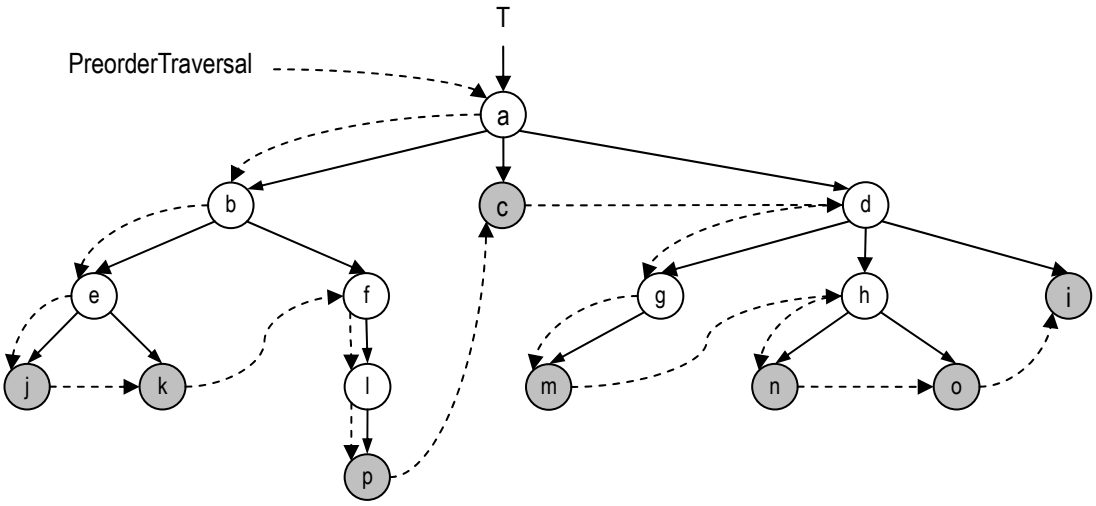
对任一（子）树的前序遍历，将首先访问其根节点，然后再递归地对其下的各棵子树进行前序遍历。对于同一根节点下的各棵子树，遍历的次序通常是任意的；但若换成有序树，则可以按照兄弟间相应的次序对它们实施遍历。由前序遍历生成的节点序列，称作前序遍历序列。

前序遍历的过程可以描述为 算法四.1：

```
算法：PreorderTraversal(v)
输入：树节点v

输出：v所有后代的前序遍历序列

{
  if (null != v) {
    首先访问并输出v;
    for (u = v.getFirstChild(); null != u; u = u.getNextSibling()) //依次
      PreorderTraversal(u);//前序遍历v的各棵子树
  }
}
```

算法四.1　前序遍历算法PreorderTraversal()

如 图四.8 所示的是前序遍历的一个实例：



图四.8　树的前序遍历序列：{a, b, e, j, k, f, l, p, c, d, g, m, h, n, o, i}

对称地，对任一（子）树的后序遍历将首先递归地对根节点下的各棵子树进行后序遍历，最后才访问根节点。由后序遍历生成的节点序列，称作后序遍历序列。

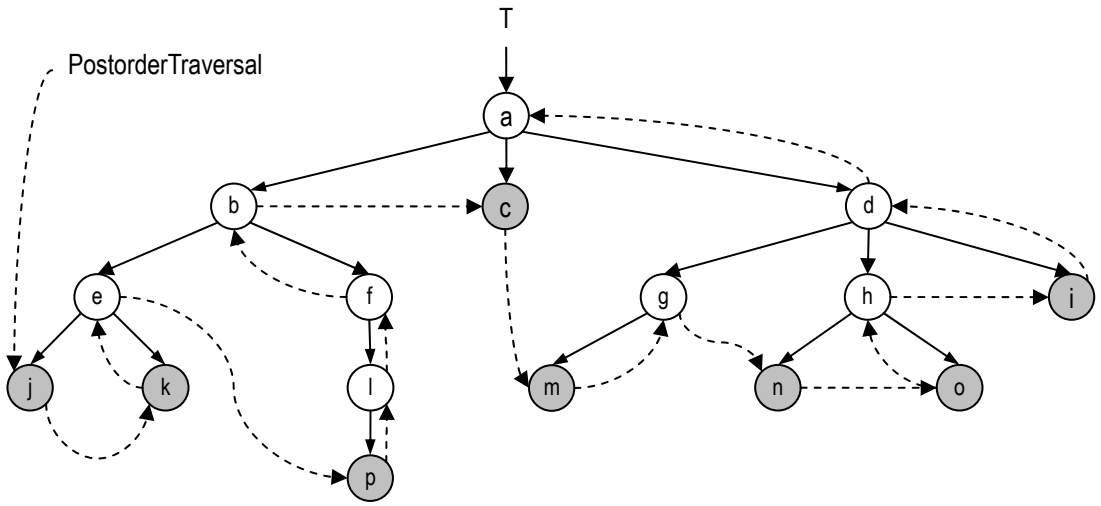
后序遍历过程可以描述为 算法四.2：

```
算法：PostorderTraversal(v)
输入：树节点v
输出：v所有后代的后序遍历序列
```

```
{
  if (null != v) {
    for (u = v.getFirstChild(); null != u; u = u.getNextSibling()) //依次
      PostorderTraversal(u);//后序遍历v的各棵子树
    当所有后代都访问过后，最后才访问并输出节点v;
  }
}
```

如 图四.9 所示的是后序遍历的一个实例：

算法四.2 后序遍历算法PostorderTraversal()



图四.9 树的后序遍历序列：{ j, k, e, p, l, f, b, c, m, g, n, o, h, i, d, a }

4.3.5 层次遍历

除了上述两种最常见的遍历算法，还有其它一些遍历算法，层次遍历（Traversal by level ）算法就是其中的一种。在这种遍历中，各节点被访问的次序取决于它们各自的深度，其策略可以总结为“深度小的节点优先访问”。对于同一深度的节点，访问的次序可以是随机的，通常取决于它们的存储次序，即首先访问由firstChild指定的长子，然后根据nextSibling确定后续节点的次序。当然，若是有序树（定义四.13，第 106 页），则同深度节点的访问次序将与有序树确定的次序一致。

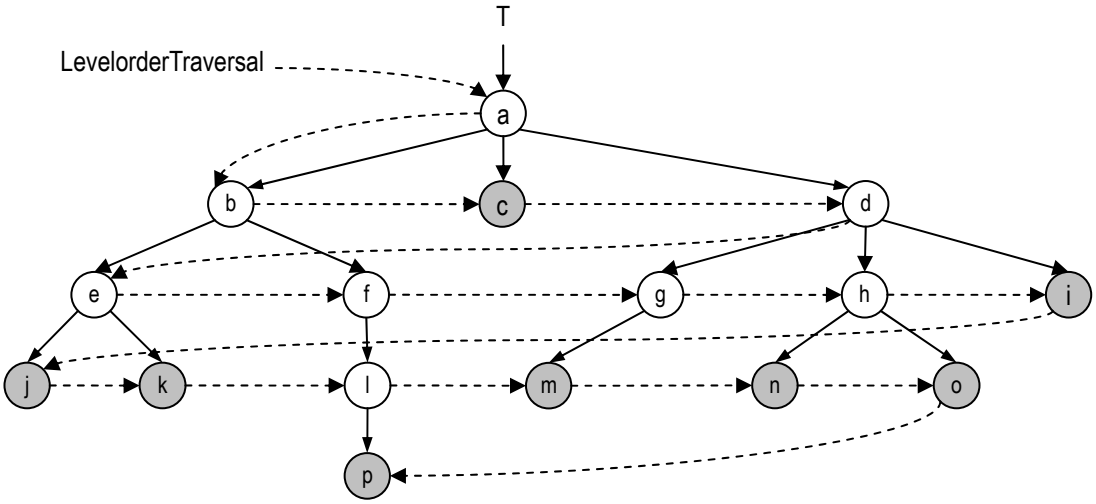
如 算法四.3 所示，层次遍历过程可以借助一个队列来实现：

```
算法：LevelorderTraversal(v)
输入：树节点v
输出：v所有后代的层次遍历序列
{
  if (null != v) {
    创建一个队列Q;
    Q.enqueue(v);//根节点入队
    while (!Q.isEmpty()) {//在队列重新变空之前
```

```
      u = Q.dequeue();//取出队列的首节点u
      访问并输出u;
      for (w = u.getFirstChild(); null != w; w = w.nextSibling())//依次将u的
        Q.enqueue(w);//每个孩子w加至队列中
    } //while
  } //if
}
```

算法四.3 层次遍历算法LevelorderTraversal()

如 图四.10 所示为层次遍历的一个实例：



图四.10 树的层次遍历序列：{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p}

4.3.6 树迭代器

树的遍历是许多复杂操作的基础，也构成了许多高级算法的基本框架，通常，这些算法都是通过在遍历的过程中对访问到的对象实施某种操作，以实现特定的功能。在 C 或 C++等程序语言中，可以借助函数指针将具体的操作蕴含于遍历算法之内，这样就可以编写一个统一的遍历算法，而在解决不同的应用问题时，我们只需集中精力有针对性地编写具体的操作。然而遗憾的是，出于对系统安全性的考虑，Java 并不支持函数指针这一机制。故此，我们只能另辟蹊径。

Java中解决这类问题的一种可行办法，就是采用第 §3.4 节所介绍的迭代器模式。具体地，我们可以基于第 3.4.2 节 代码三.11 所给出的Iterator接口，利用列表实现一个如 代码四.3 所示的IteratorTree类，其中的preorderIteratorTree()、postorderIteratorTree()和levelTraversalIterator()方法，分别按照前序遍历、后序遍历和层次遍历的次序，创建一个由树中所有节点组成的迭代器，以供后续操作使用。

```
/**
 * 基于列表实现的树迭代器
 */
```

```
package dsa;

public class IteratorTree implements Iterator {
  private List list;//列表
  private Position nextPosition;//当前（下一个）元素的位置

//默认构造方法
  public IteratorTree() { list = null; }

//前序遍历
  public void elementsPreorderIterator(TreeLinkedList T) {
    if (null == T)  return;//递归基

    list.insertLast(T);//首先输出当前节点

    TreeLinkedList  subtree = T.getFirstChild();//从当前节点的长子开始
    while (null != subtree) {//依次对当前节点的各个孩子
      this.elementsPreorderIterator(subtree);//做前序遍历
      subtree = subtree.getNextSibling();
    }
  }

//后序遍历
  public void elementsPostorderIterator(TreeLinkedList T) {
    if (null == T)  return;//递归基
    TreeLinkedList  subtree = T.getFirstChild();//从当前节点的长子开始
    while (null != subtree) {//依次对当前节点的各个孩子
      this.elementsPostorderIterator(subtree);//做后序遍历
      subtree = subtree.getNextSibling();
    }
    list.insertLast(T);//当所有后代都访问过后，最后才访问当前节点
  }

//层次遍历
  public void levelTraversalIterator(TreeLinkedList T) {
    if (null == T)  return;
    Queue_List  Q = new Queue_List();//空队
    Q.enqueue(T);//根节点入队
    while (!Q.isEmpty()) {//在队列重新变空之前
      TreeLinkedList  tree = (TreeLinkedList) (Q.dequeue());//取出队列首节点
      list.insertLast(tree);//将新出队的节点接入迭代器中
      TreeLinkedList  subtree = tree.getFirstChild();//从tree的第一个孩子起
      while (null != subtree) {//依次找出所有孩子，并
        Q.enqueue(subtree);//将其加至队列中
        subtree = subtree.getNextSibling();
      }
```

```
    }
  }

//检查迭代器中是否还有剩余的元素
  public boolean hasNext() { return (null != nextPosition); }

//返回迭代器中的下一元素
  public Object getNext() throws ExceptionNoSuchElement {
    if (!hasNext()) throw new ExceptionNoSuchElement("No next position");
    Position currentPosition = nextPosition;
    if (currentPosition == list.last())//若已到达尾元素，则
      nextPosition = null;//不再有下一元素
    else//否则
      nextPosition = list.getNext(currentPosition);//转向下一元素

    return currentPosition.getElem();

  }
}
```

代码四.3 基于列表实现的树迭代器

如此实现的前序、后序及层次遍历算法，为每个节点、每条边只需花费常数时间，因此，根据观察结论四.1，可以得出如下结论：

定理四.1 树的前序、后序及层次遍历，均可在 $\mathcal{O}(n)$时间内完成，其中 n 为树本身的规模。

§4.4 二叉树抽象数据类型及其实现

在算法领域，二叉树的重要地位是其它结构无法替代的。按照 定义四.15，所谓二叉树就是各节点不超过 2 度的有序树，因此每个节点的孩子（如果存在的话）可以左、右区分，分别称作左孩子和右孩子——之所以如此命名，是因为在画出二叉树时，通常都将左、右孩子分别画在其父节点的左下方、右下方。

在几乎所有的应用问题中，都可以看到二叉树的身影。而且令我们感到惊讶的是，尽管二叉树只是树的一种特例，但无论是从对应用问题的描述与刻画能力，还是从计算能力的角度来看，二叉树并不逊色于一般意义上的树。反过来，正是得益于其定义的简洁性以及结构的规范性，基于二叉树的算法往往可以更好地得到描述，其实现也更加简捷。

4.4.1 二叉树 ADT

二叉树抽象类型需要支持以下的基本方法：

表四.2 二叉树ADT支持的操作

操作方法	功能描述
getElement():	返回存放当前节点处的对象 输入：无 输出：对象
setElement(e):	将对象 e 存入当前节点，并返回其中此前所存的内容 输入：一个对象 输出：对象
getParent():	返回当前节点的父节点 输入：无 输出：树节点
getLChild():	返回当前节点的左孩子 输入：无 输出：二叉树节点
getRChild():	返回当前节点的右孩子 输入：无 输出：二叉树节点

与普通的树一样，不同的应用问题可能会在上述操作之外要求二叉树结构提供更多的方法。这些操作多属更新类操作，而且具体功能差异极大。在此，我们只能在二叉树 ADT 中定义相对通用的那些更新操作。在后续的讨论中，我们将结合各种具体的应用问题，相应地定义并实现一些其它的更新操作。

4.4.2 二叉树类的 Java 接口

根据上述规范，我们可以给出二叉树接口（代码四.4）以及相应的树节点接口（代码四.5）。

■ BinT ree 接口

```
/*
 * 二叉树接口
 */

package dsa;

public interface BinTree {
//返回树根
  public BinTreePosition getRoot();

//判断是否树空
  public boolean isEmpty();

//返回树的规模（即树根的后代数目）
  public int getSize();

//返回树（根）的高度
```

```
  public int getHeight();

//前序遍历
  public Iterator elementsPreorder();

//中序遍历
  public Iterator elementsInorder();

//后序遍历
  public Iterator elementsPostorder();

//层次遍历
  public Iterator elementsLevelorder();
}
```

代码四.4 二叉树ADT的Java接口

■ BinT reePosition 接口

与第 §3.2 节的做法一样，为了在遵循面向对象规范的同时保证效率，这里也将使用位置的概念来描述和实现二叉树节点。在 代码二.14 所定义Position接口的基础上，经过扩充可以定义出如 代码四.5 所示的BinTreePositition接口。

```
/*
 * 二叉树节点ADT接口
 */

package dsa;

public interface BinTreePosition extends Position {
//判断是否有父亲（为使代码描述简洁）
  public boolean hasParent();
//返回当前节点的父节点
  public BinTreePosition getParent();
//设置当前节点的父节点
  public void setParent(BinTreePosition p);

//判断是否为叶子
  public boolean isLeaf();

//判断是否为左孩子（为使代码描述简洁）
  public boolean isLChild();

//判断是否有左孩子（为使代码描述简洁）
  public boolean hasLChild();
//返回当前节点的左孩子
  public BinTreePosition getLChild();
```

```
//设置当前节点的左孩子（注意：this.lChild和c.parent都不一定为空）
  public void setLChild(BinTreePosition c);

//判断是否为右孩子（为使代码描述简洁）
  public boolean isRChild();
//判断是否有右孩子（为使代码描述简洁）
  public boolean hasRChild();
//返回当前节点的右孩子
  public BinTreePosition getRChild();
//设置当前节点的右孩子（注意：this.rChild和c.parent都不一定为空）
  public void setRChild(BinTreePosition c);

//返回当前节点后代元素的数目
  public int getSize();
//在孩子发生变化后，更新当前节点及其祖先的规模
  public void updateSize();
```

```
//返回当前节点的高度
  public int getHeight();
//在孩子发生变化后，更新当前节点及其祖先的高度
  public void updateHeight();

//返回当前节点的深度
  public int getDepth();
//在父亲发生变化后，更新当前节点及其后代的深度
  public void updateDepth();

//按照中序遍历的次序，找到当前节点的直接前驱
  public BinTreePosition getPrev();

//按照中序遍历的次序，找到当前节点的直接后继
  public BinTreePosition getSucc();

//断绝当前节点与其父亲的父子关系
//返回当前节点
  public BinTreePosition secede();

//将节点c作为当前节点的左孩子
  public BinTreePosition attachL(BinTreePosition c);

//将节点c作为当前节点的右孩子
  public BinTreePosition attachR(BinTreePosition c);

//前序遍历
  public Iterator elementsPreorder();
```

```
//中序遍历
  public Iterator elementsInorder();

//后序遍历
  public Iterator elementsPostorder();

//层次遍历
  public Iterator elementsLevelorder();
}
```

代码四.5 二叉树节点ADT的Java接口

除了继承Position接口的getElem()和setElem()方法外，这里还针对二叉树节点的操作要求，定义了一系列的方法。其中有些是辅助性的，旨在提高代码的简洁性和可读性。其中还定义了诸如secede()、attachL()和attachR()等方法，这是为后面（第七章）实现查找树的相关操作而埋下的伏笔。另外，这里还定义了对当前节点所有后代的若干遍历方法，与普通树的遍历方法（第 4.3.6 节）一样，它们也符合迭代器规范，可以生成相应的节点遍历序列。当然，鉴于二叉树的特点，这里增加了一种遍历算法——中序遍历。

4.4.3 二叉树类的实现

我们首先给出基于链表实现的二叉树节点类（代码四.6）以及相应的二叉树类（代码四.7），在稍后的第 §4.5 节中，我们再陆续讨论其中的相关算法。

■ 二叉树节点类的实现

```
/*
 * 基于链表节点实现二叉树节点
 */

package dsa;

public class BinTreeNode implements BinTreePosition {
  protected Object element;//该节点中存放的对象
  protected BinTreePosition parent;//父亲
  protected BinTreePosition lChild;//左孩子
  protected BinTreePosition rChild;//右孩子
  protected int size;//后代数目
  protected int height;//高度
  protected int depth;//深度

/*************************** 构造方法 ***************************/
  public BinTreeNode()
  { this(null, null, true, null, null); }
```

```
  public BinTreeNode(
    Object e,//节点内容
    BinTreePosition p,//父节点
    boolean asLChild,//是否作为父节点的左孩子
    BinTreePosition l,//左孩子
    BinTreePosition r)//右孩子
  {
    size = 1; height = depth = 0; parent = lChild = rChild = null;//初始化

    element = e;//存放的对象

  //建立与父亲的关系
    if (null != p)
      if (asLChild) p.attachL(this);
      else          p.attachR(this);

  //建立与孩子的关系
    if (null != l)  attachL(l);
    if (null != r)  attachR(r);

  }
```

```
/***************************** Position接口方法 *******************************/
//返回当前节点中存放的对象
  public Object getElem()
  { return element; }

//将对象obj存入当前节点，并返回此前的内容
  public Object setElem(Object obj)
  { Object bak = element; element = obj;  return bak; }

/***************************** BinTreePosition接口方法 ************************/
//判断是否有父亲（为使代码描述简洁）
  public boolean hasParent()
  { return null != parent; }
//返回当前节点的父节点
  public BinTreePosition getParent()
  { return parent; }
//设置当前节点的父节点
  public void setParent(BinTreePosition p)
  { parent = p; }

//判断是否为叶子
  public boolean isLeaf()
  { return !hasLChild() && !hasRChild(); }

//判断是否为左孩子（为使代码描述简洁）
```

```
//若当前节点有父亲，而且是左孩子，则返回true；否则，返回false
  public boolean isLChild()
  { return (hasParent() && this == getParent().getLChild()) ? true : false; }

//判断是否有左孩子（为使代码描述简洁）
  public boolean hasLChild()
  { return null != lChild; }
//返回当前节点的左孩子
  public BinTreePosition getLChild()
  { return lChild; }
//设置当前节点的左孩子（注意：this.lChild和c.parent都不一定为空）
  public void setLChild(BinTreePosition c)
  { lChild = c; }

//判断是否为右孩子（为使代码描述简洁）
//若当前节点有父亲，而且是右孩子，则返回true；否则，返回false
  public boolean isRChild()
  { return (hasParent() && this == getParent().getRChild()) ? true : false; }
//判断是否有右孩子（为使代码描述简洁）
  public boolean hasRChild()
  { return null != rChild; }
//返回当前节点的右孩子

  public BinTreePosition getRChild()

  { return rChild; }
//设置当前节点的右孩子（注意：this.rChild和c.parent都不一定为空）
  public void setRChild(BinTreePosition c)
  { rChild = c; }

//返回当前节点后代元素的数目
  public int getSize()
  { return size; }
//在孩子发生变化后，更新当前节点及其祖先的规模
  public void updateSize() {
    size = 1;//当前节点
    if (hasLChild())  size += getLChild().getSize();//左子树的规模
    if (hasRChild())  size += getRChild().getSize();//右子树的规模

    if (hasParent())  getParent().updateSize();//递归更新各个真祖先的规模记录
  }

//返回当前节点的高度
  public int getHeight()
  { return height; }
//在孩子发生变化后，更新当前节点及其祖先的高度
  public void updateHeight() {
    height = 0;//先假设没有左、右孩子
```

```
    if (hasLChild())  height = Math.max(height, 1+getLChild().getHeight());//左孩子
    if (hasRChild())  height = Math.max(height, 1+getRChild().getHeight());//右孩子

    if (hasParent())  getParent().updateHeight();//递归更新各个真祖先的高度记录
  }

//返回当前节点的深度
  public int getDepth()
  { return depth; }
//在父亲发生变化后，更新当前节点及其后代的深度
  public void updateDepth() {
    depth = hasParent() ? 1+getParent().getDepth() : 0;//当前节点

    if (hasLChild())  getLChild().updateDepth();//沿孩子引用逐层向下，
    if (hasRChild())  getRChild().updateDepth();//递归地更新所有后代的深度记录
  }

//按照中序遍历的次序，找到当前节点的直接前驱
  public BinTreePosition getPrev() {
  //若左子树非空，则其中的最大者即为当前节点的直接前驱
    if (hasLChild())  return findMaxDescendant(getLChild());
  //至此，当前节点没有左孩子
    if (isRChild()) return getParent();//若当前节点是右孩子，则父亲即为其直接前驱

  //至此，当前节点没有左孩子，而且是左孩子

    BinTreePosition v = this;//从当前节点出发
    while (v.isLChild())  v = v.getParent();//沿左孩子链一直上升
  //至此，v或者没有父亲，或者是父亲的右孩子
    return v.getParent();
  }

//按照中序遍历的次序，找到当前节点的直接后继
  public BinTreePosition getSucc() {
  //若右子树非空，则其中的最小者即为当前节点的直接后继
    if (hasRChild())  return findMinDescendant(getRChild());
  //至此，当前节点没有右孩子
    if (isLChild()) return getParent();//若当前节点是左孩子，则父亲即为其直接后继
  //至此，当前节点没有右孩子，而且是右孩子
    BinTreePosition v = this;//从当前节点出发
    while (v.isRChild())  v = v.getParent();//沿右孩子链一直上升
  //至此，v或者没有父亲，或者是父亲的左孩子
    return v.getParent();
  }

//断绝当前节点与其父亲的父子关系
```

```
//返回当前节点
  public BinTreePosition secede() {
    if (null != parent) {
      if (isLChild()) parent.setLChild(null);//切断父亲指向当前节点的引用
      else            parent.setRChild(null);

      parent.updateSize();//更新当前节点及其祖先的规模
      parent.updateHeight();//更新当前节点及其祖先的高度

      parent = null;//切断当前节点指向原父亲的引用
      updateDepth();//更新节点及其后代节点的深度
    }

    return this;//返回当前节点
  }

//将节点c作为当前节点的左孩子
  public BinTreePosition attachL(BinTreePosition c) {
    if (hasLChild())  getLChild().secede();//摘除当前节点原先的左孩子

    if (null != c) {
      c.secede();//c脱离原父亲
      lChild = c; c.setParent(this);//确立新的父子关系
      updateSize();//更新当前节点及其祖先的规模
      updateHeight();//更新当前节点及其祖先的高度

      c.updateDepth();//更新c及其后代节点的深度

    }

    return this;
  }

//将节点c作为当前节点的右孩子
  public BinTreePosition attachR(BinTreePosition c) {
    if (hasRChild())  getRChild().secede();//摘除当前节点原先的右孩子

    if (null != c) {
      c.secede();//c脱离原父亲
      rChild = c; c.setParent(this);//确立新的父子关系
      updateSize();//更新当前节点及其祖先的规模
      updateHeight();//更新当前节点及其祖先的高度
      c.updateDepth();//更新c及其后代节点的深度
    }

    return this;
  }
```

```
//前序遍历
  public Iterator elementsPreorder() {
    List list = new List_DLNode();
    preorder(list, this);
    return list.elements();
  }

//中序遍历
  public Iterator elementsInorder() {
    List list = new List_DLNode();
    inorder(list, this);
    return list.elements();
  }

//后序遍历
  public Iterator elementsPostorder() {
    List list = new List_DLNode();
    postorder(list, this);
    return list.elements();
  }

//层次遍历
  public Iterator elementsLevelorder() {
    List list = new List_DLNode();
    levelorder(list, this);
    return list.elements();
  }

/*************************** 辅助方法 ***************************/
```

```
//在v的后代中，找出最小者

  protected static BinTreePosition findMinDescendant(BinTreePosition v) {
    if (null != v)
      while (v.hasLChild()) v = v.getLChild();//从v出发，沿左孩子链一直下降
  //至此，v或者为空，或者没有左孩子
    return v;
  }

//在v的后代中，找出最大者
  protected static BinTreePosition findMaxDescendant(BinTreePosition v) {
    if (null != v)
      while (v.hasRChild()) v = v.getRChild();//从v出发，沿右孩子链一直下降
  //至此，v或者为空，或者没有右孩子
    return v;
  }

//前序遍历以v为根节的（子）树
  protected static void preorder(List list, BinTreePosition v) {
```

```
    if (null == v)  return;//递归基：空树
    list.insertLast(v);//访问v
    preorder(list, v.getLChild());//遍历左子树
    preorder(list, v.getRChild());//遍历右子树
  }

//中序遍历以v为根节的（子）树
  protected static void inorder(List list, BinTreePosition v) {
    if (null == v)  return;//递归基：空树
    inorder(list, v.getLChild());//遍历左子树
    list.insertLast(v);//访问v
    inorder(list, v.getRChild());//遍历右子树
  }

//后序遍历以v为根节的（子）树
  protected static void postorder(List list, BinTreePosition v) {
    if (null == v)  return;//递归基：空树
    postorder(list, v.getLChild());//遍历左子树
    postorder(list, v.getRChild());//遍历右子树
    list.insertLast(v);//访问v
  }

//层次遍历以v为根节的（子）树
  protected static void levelorder(List list, BinTreePosition v) {
    Queue_List  Q = new Queue_List();//空队
    Q.enqueue(v);//根节点入队
    while (!Q.isEmpty()) {
      BinTreePosition u = (BinTreePosition) Q.dequeue();//出队

      list.insertLast(u);//访问v

      if (u.hasLChild())  Q.enqueue(u.getLChild());
      if (u.hasRChild())  Q.enqueue(u.getRChild());
    }
  }
}
```

代码四.6 基于链表的二叉树节点实现

■ 二叉树类的实现

至此，我们可以给出基于链表实现的二叉树结构如下：

```
/*
 * 基于链表实现二叉树
 */
```

```
package dsa;

public class BinTree_LinkedList implements BinTree {
 protected BinTreePosition root;//根节点

/*************************** 构造函数 ***************************/
  public BinTree_LinkedList()
  { this(null); }

  public BinTree_LinkedList(BinTreePosition r)
  { root = r; }

/*************************** BinaryTree接口方法 ***************************/
//返回树根
  public BinTreePosition getRoot()
  { return root; }

//判断是否树空
  public boolean isEmpty()
  { return null == root; }

//返回树的规模（即树根的后代数目）
  public int getSize()
  { return isEmpty() ? 0 : root.getSize(); }

//返回树（根）的高度
  public int getHeight()
  {return isEmpty() ? -1 : root.getHeight(); }

//前序遍历
  public Iterator elementsPreorder()
  { return root.elementsPreorder(); }
```

```
//中序遍历
  public Iterator elementsInorder()
  { return root.elementsInorder(); }

//后序遍历
  public Iterator elementsPostorder()
  { return root.elementsPostorder(); }

//层次遍历
  public Iterator elementsLevelorder()
  { return root.elementsLevelorder(); }
}
```

代码四.7 基于链表的二叉树实现

§4.5 二叉树的基本算法

第 §4.4 节已经给出了二叉树相关算法的具体实现，下面我们就来对这些算法做一分析。请读者对照 代码四.6 和 代码四.7 中相应的Java方法，以理解具体的实现方式。

4.5.1 getSize()、getHeight()和 getDepth()

这三个方法的功能是，分别返回二叉子树的规模、树根节点的高度和深度。

这里为每个节点设置了三个变量size、height和depth，分别对应于这三个指标，这样，只需在$\mathcal{O}(1)$时间内返回相应的变量，即可实现相应的功能。请读者将这种实现与第 §4.3 节中相应算法的实现作一对比，体会两种方式的优、缺点。

4.5.2 updateSize()

若当前节点的孩子发生变化，比如原有的某个孩子被删除或者有新的孩子插入，就需要更新当前节点及其祖先的规模记录，以便后续的查询，updateSize()方法的功能正在于此。请注意，在这里，我们允许直接对任何节点执行这一操作。

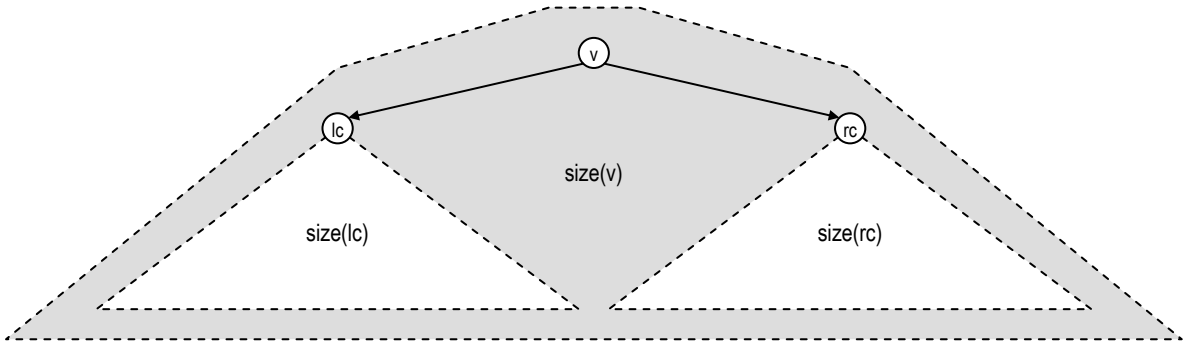


图 四.11 size(v) = 1 + size(lc) + size(rc)

观察结论四.13若节点 v 的左、右孩子分别为 lc 和 rc，则 size(v) = 1 + size(lc) + size(rc)。

因此，一旦其左、右子树的规模都已确定，我们就可以在$\mathcal{O}(1)$时间内得到以节点v为根的子树规模。当然，此后还需要从v出发沿parent引用逆行向上，依次更新各个祖先的规模记录。这可以描述为 算法四.4：

```
算法：updateSize(v)
输入：二叉树中任一节点v
输出：更新v的后代规模记录
{
  令size(v) = 1 + size(lc) + size(rc);//由观察结论四.13
  若v的父亲p存在，则调用updateSize(p)，递归地更新父亲的规模记录;//尾递归，可改写为迭代形式
```

```
}
```

算法四.4 upd ateSize()算法

若节点 v 的深度为 depth(v)，则总共需要修改 depth(v)+1 个节点的规模。为了更新一个节点的规模记录，只需执行两次 getSize()操作并做两次加法，故 updateSize()算法的总体运行时间为 $\mathcal{O}$(depth(v)+1)。

4.5.3 updateHeight()

同样地，在孩子发生变化后，也有必要更新当前节点的高度记录。

再次利用 推论四.5，只需读出左、右孩子的高度，取二者中的大者，再计入当前节点本身，就得到了当前节点v的新高度。当然，此后也需要从v出发沿parent引用逆行向上，依次更新各个祖先的高度记录。这一过程可以描述为 算法四.5：

```
算法：updateHeight(v)
输入：二叉树中任一节点v
输出：更新v的高度记录
{
  height(v) = 0;//先假设没有左、右孩子
  若v有左孩子lc，则令：height(v) = Max(height(v), 1 + height(lc));
  若v有右孩子lc，则令：height(v) = Max(height(v), 1 + height(rc));
  若v的父亲p存在，则调用updateHeight(p)，递归地更新父亲的高度记录;
}
```

算法四.5 upd ateHeight()算法

同样地，若节点 v 的深度为 depth(v)，则总共需要修改 depth(v)+1 个节点的高度记录。更新每一节点本身的高度记录，只需执行两次 getHeight()操作、两次加法以及两次取最大操作，不过常数时间，故 updateHeight()算法的总体运行时间也是 $\mathcal{O}$(depth(v)+1)。

这一算法还可以进一步优化。我们注意到，在逆行向上依次更新各祖先高度的过程中，一旦发现某个祖先的高度没有发生变化，算法即可提前终止。请读者按此思路自行改进。

4.5.4 updateDepth()

在父亲节点发生变化后，有必要更新当前节点的深度记录。

再次利用 推论四.6，只需读出新的父亲节点的深度，再加上一即得到当前节点新的深度。当然，此后还需要沿着lChild和rChild引用，逐层向下递归地更新每一后代的深度记录。

```
算法：updateDepth(v)
输入：二叉树中任一节点v
输出：更新v的深度记录
```

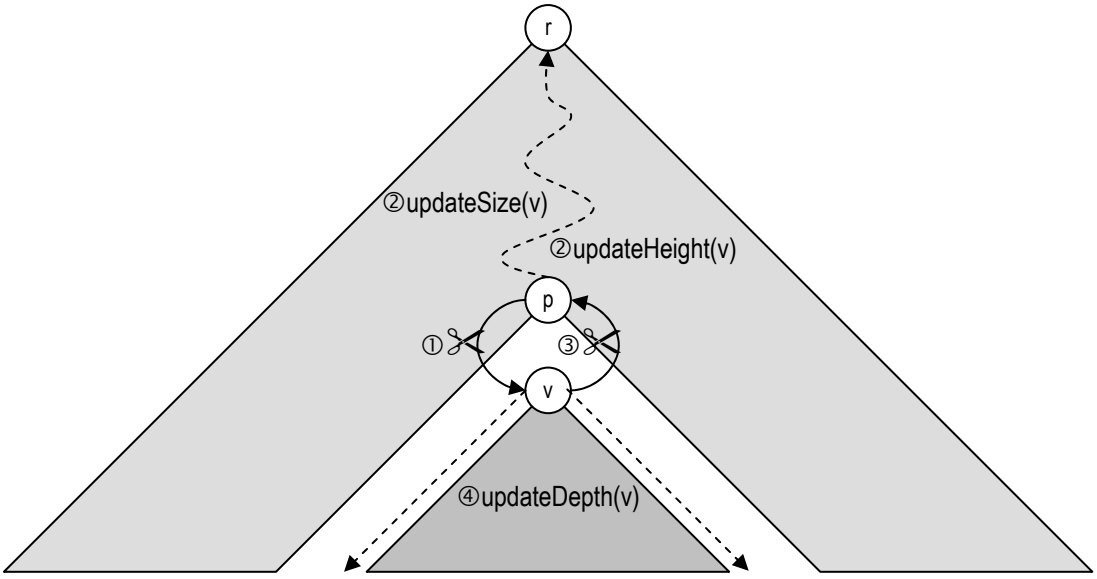
```
{
  若v的父亲节点p存在，则令depth(v) = depth(p)+1;
  否则，令depth(v) = 0;

  若v的左孩子lc存在，则调用updateDepth(lc);//沿孩子引用逐层向下，
  若v的右孩子rc存在，则调用updateDepth(rc);//递归地更新所有后代的深度记录
}
```

算法四.6 upd ateDepth()算法

若节点 v 的后代规模为 size(v)，则总共需要修改 size(v)个节点的深度记录。鉴于单个节点的深度记录可以在常数时间内得到更新，故 updateDepth()算法的总体运行时间为 $\mathcal{O}$(size(v))。与 updateHeight()算法类似，上述 updateDepth()算法也可以进行优化。

4.5.5 secede()

为了简化二叉树的动态操作的实现，这里专门设计了一个secede()方法。如 图四.12 所示，该方法的功能是，将以某一节点为根的子树从母树中分离出来。在后面 第七章介绍各种查找树时，这一方法与稍后介绍的attachL()和attachR()方法结合起来，将使得对有关算法的描述及其实现更加简捷明了，减少代码量，增加代码的可读性。



图四.12 s ecede(v)操作的4个步骤

secede()方法的具体操作过程可以描述为 算法四.7：

```
算法：secede(v)
输入：二叉树中任一节点v
输出：将以v为根的子树丛母树中分离出来
{
  若v有父亲{
```

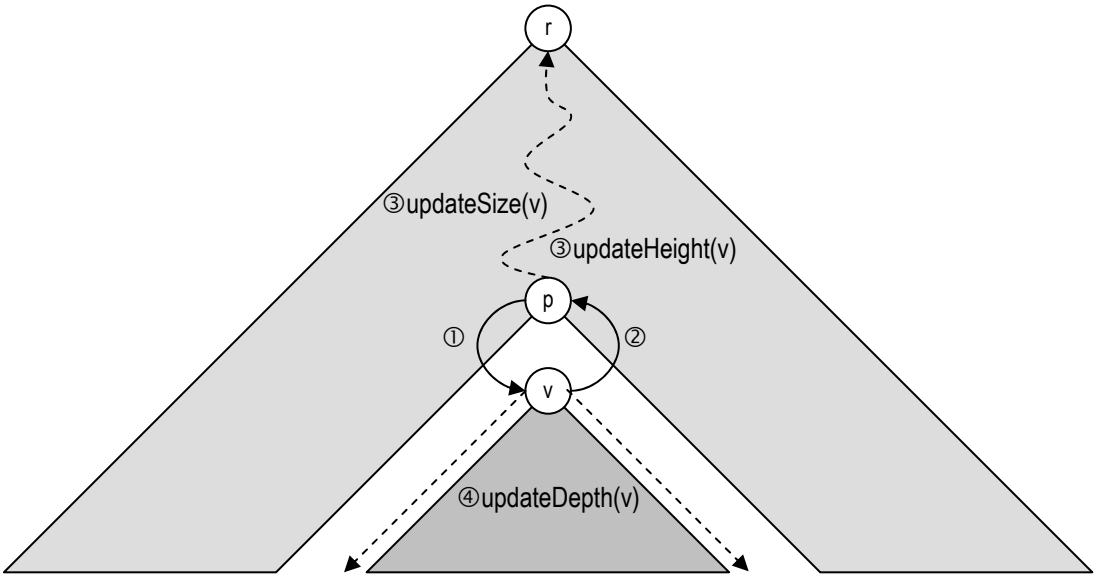
```
    ①  切断父亲指向v的引用;
    ②  调用updateSize(v)和updateHeight(v)，更新v及其祖先的规模记录和高度记录;
    ③  切断v指向父亲的引用;
    ④  调用updateDepth(v)，更新v及其后代的深度记录;
  }
}
```

算法四.7 s ecede()算法

由上可见，除了常数次对引用的设置外，这一算法无非是对节点 v 各执行了一次 updateSize(v)、updateHeight(v)和 updateDepth(v)操作，根据前面关于这三个算法的运行时间的分析结论，secede(v)算法的运行时间为 $\mathcal{O}$(depth(v)+ size(v)+1)，其中 depth(v)和 size(v)定义同上。

4.5.6 attachL()和 attachR()

如 图四.13 所示，这一对方法的功能是，将节点c作为左或右孩子与节点v联接起来。



图四.13 attach L()和attachR()操作的4个步骤

attachL(v, c)方法的操作过程如 算法四.8 所示，attachR(v, c)与之完全对称，请读者自己补充。

```
算法: attachL(p, c)
输入: 两个二叉树节点p与c
输出: 将c作为左孩子，与p联接起来
{
  ⓪  若p已经有左孩子lc，则首先调用secede(lc)将其摘除;
  ⓪  调用secede(c)，使c及其后代脱离原属母树;
  ①② 设置相应的引用，在p和c之间建立父子关系;
  ③  调用updateSize(p)和updateHeight(p)，更新节点p及其祖先的规模和高度;
  ④  调用updateDepth(c)，更新c及其后代节点的深度;
```

```
}
```

算法四.8 attach L()算法

4.5.7 二叉树的遍历

作为树的一种特例，二叉树自然继承了一般树结构的前序、后序以及层次等遍历方法。这三个遍历算法的实现与普通树大同小异，这里不再赘述。

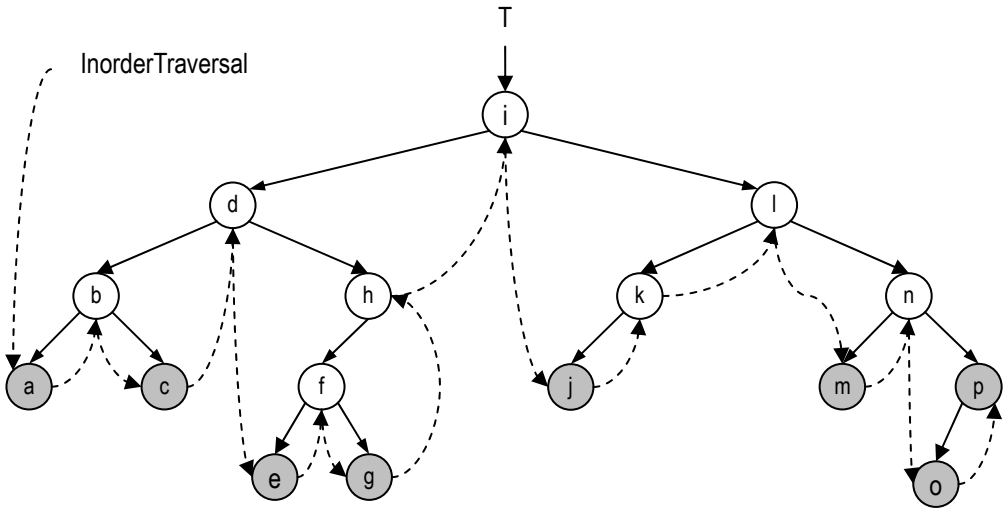
需要特别指出的是，对二叉树还可以定义一个新的遍历方法——中序遍历（Inorder traversal）。顾名思义，在访问每个节点之前，首先遍历其左子树；待该节点被访问过后，才遍历其右子树。类似地，由中序遍历确定的节点序列，称作中序遍历序列。第七章将要介绍的查找树，正是建立在中序遍历序列的基础之上，由此也可见这一遍历算法的重要性。

二叉树中序遍历的过程，可以描述为 算法四.9：

```
算法：InorderTraversal(v)

输入：二叉树节点v

输出：v所有后代的中序遍历序列
{
  if (null != v) {//设lc、rc分别为v的左、右孩子
    调用InorderTraversal(lc)对v的左子树做中序遍历;
    访问并输出v;
    调用InorderTraversal(rc)对v的右子树做中序遍历;
  }
}
```

算法四.9 中序遍历算法InorderTraversal()

图四.14 给出了二叉树中序遍历的一个实例。



图四.14 二叉树的中序遍历序列：{ a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p }

4.5.8 直接前驱、直接后继的定位算法

细心的读者可能会注意到，在如 图四.14 形式的二叉树表示中，只要规定“左（右）子树必须完全居于根节点的（左）右侧”，则所有节点在水平轴上投影的自左向右次序，恰好与中序遍历序列的次序吻合。从这个意义上说，中序遍历就是按照自左向右的次序访问各个节点。

按照这一思路，对于任一二叉树T，根据其中序遍历序列S(T)，我们都可以在其中所有节点之间定义出一个线性次序。因此，除首（末）节点外的每一节点都有唯一的直接前驱（后继）。实际上，在很多算法（比如第 7.1.5 节的二分查找树节点删除算法）中，我们的确需要找出某一指定节点的直接前驱。那么，除了做一次中序遍历外，有没有其它方法可以找出某一指定节点的直接前驱或直接后继？

答案是肯定的。在给出相应的算法之前，我们首先注意到以下事实：

观察结论四.14 二叉树中，除中序遍历序列中的首节点外，任一节点 v 的直接前驱 u 不外乎三种可能：

① v 没有左孩子，同时 v 是右孩子：此时，u 就是 v 的父亲节点；

② v 没有左孩子，同时 v 是左孩子：此时，从 v 出发沿 parent 引用逆行向上，直到第一个是右孩子的节点 w，则 u 就是 w 的父亲节点；

③ v 有左孩子：此时，从 v 的左孩子出发，沿 rChild 引用不断下行，最后一个（没有右孩子的）节点就是 u。

仍然以 图四.14 为例：c的直接前驱为b，即属于情况①；e的直接前驱为d，即属于情况②；h的直接前驱为g，即属于情况③。

实际上，观察结论四.14 已经直接给出了一个查找直接前驱的算法，其具体实现请参见 代码四.6 中的getPrev()及findMaxDescendant()方法。

直接后继的查找算法完全是对称的，请读者参照 代码四.6 自行分析、领会。

§4.6 完全二叉树的 Java 实现

第 §5.8 节将采用基于完全二叉树的堆结构实现优先队列ADT，故完全二叉树本身的实现将直接关系到堆操作的效率，因此我们在此首先解决这一问题。

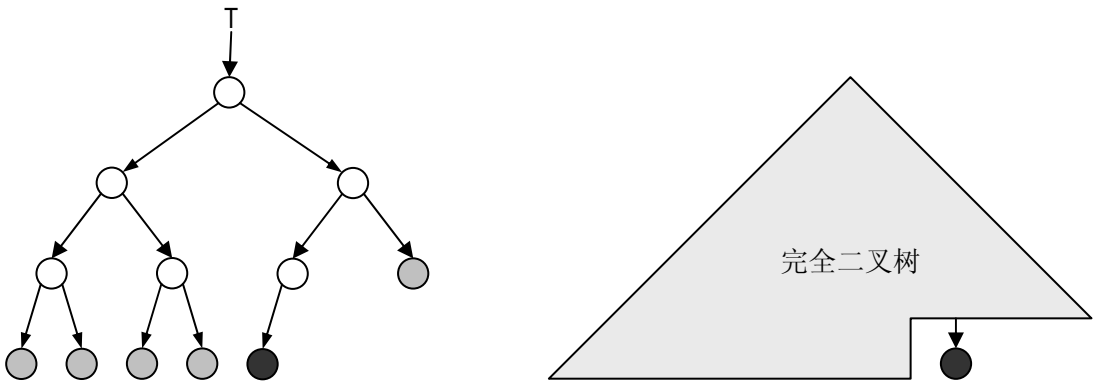
4.6.1 完全二叉树类的 Java 接口

完全二叉树的Java接口如 代码四.8 所示。

```
/*
 * 完全二叉树接口
 */

package dsa;

public interface ComplBinTree extends BinTree {
//生成并返回一个存放e的外部节点，该节点成为新的末节点
  public BinTreePosition addLast(Object e);

//删除末节点，并返回其中存放的内容
  public Object delLast();

//返回按照层次遍历编号为i的节点的位置，0 <= i < size()
  public BinTreePosition posOfNode(int i);
}
```

代码四.8 完全二叉树接口

可以看出，在BinTree接口（代码四.4）的基础上，这里增加了addLast()和delLast()两个操作。借助这两个操作，我们可以在完全二叉树中插入或删除末节点。如 图四.15 所示，这里所谓的“末节点”，是指完全二叉树的层次遍历序列中的末节点。



图四.15 完全二叉树的插入操作addLast()与删除操作delLast()

实际上，为了保证二叉树的完全性不致受到破坏，对完全二叉树的操作只能限于这两个方法。

4.6.2 基于向量的实现

■ 基于层次遍历序列的节点编号

不难注意到，只要给定规模 n，完全二叉树的结构就已完全确定。因此我们可以从 0 开始到 n-1，按照层次遍历的次序对各节点进行编号，并按照这一编号将各节点（的引用）组织为一个向量。由于各节点的编号是连续的，故总共只需 $\mathcal{O}(n)$的空间，就空间效率而言，这几乎没有任何浪费。然而反过来，根据各节点在向量中的秩，能否确定其父子关系呢？需要多少时间才能确定呢？

若将节点v的这种编号记作i(v)，则根节点的编号i(root) = 0，i(lchild(root)) = 1，i(rchild(root)) = 2，i(lchild(lchild(root)) = 3 ，…。一般地，定理四.2 指出了这种编号的一个简单而实用的性质（请读者自己证明）。

定理四.2 在完全二叉树中，

① 若节点 v 有左孩子，则 i(lchild(v)) = $2 \times i(v) + 1$；

② 若节点 v 有右孩子，则 i(rchild(v)) = $2 \times i(v) + 2$；

③ 若节点 v 有父节点，则 i(parent(v)) = $\lfloor (i(v) - 1)/2 \rfloor = \lceil (i(v)/2 \rceil - 1$。

根据这一性质，不难回答上面的问题。实际上，借助各节点在向量中的秩，的确可以找到其左、右孩子与父节点的秩；如果它们不存在，也可以判断出来。尤其重要的是，无论是那种情况，这些都可以在常数时间做到。

于是，根据第 3.1.3 节的 定理三.1，我们可以直接得出如下结论：

观察结论四.15若基于可扩充向量来实现完全二叉树，则就分摊复杂度而言，每次 addLast()和 delLast()操作都可以在 $\mathcal{O}(1)$时间内完成。

■ 完全二叉树节点类的 Java 实现

代码四.9 给出了完全二叉树节点类基于向量的具体实现。

```
/*
 * 基于秩实现的完全二叉树节点

 */


package dsa;

public class ComplBinTreeNode_Rank extends BinTreeNode implements BinTreePosition {
  private Vector  T;//所属的树
  private int rank;//在所属树中的秩
  private Object element;//存放的对象

//构造函数
  public ComplBinTreeNode_Rank (Vector t, Object obj) {
    element = obj;
    T = t;
```

```
    rank = T.getSize();
    T.insertAtRank(rank, this);
  }

//返回当前节点中存放的对象
  public Object getElem()
  { return element; }
//将对象obj存入当前节点，并返回此前的内容
  public Object setElem(Object obj)
  { Object bak = element;  element = obj;  return bak; }

//判断是否有父亲（为使代码描述简洁）
  public boolean hasParent()
  { return (0 != rank) ? true : false; }
//返回当前节点的父节点
  public BinTreePosition getParent()
  { return hasParent() ? (BinTreePosition) T.getAtRank((rank-1)/2) : null; }

//判断是否有左孩子（为使代码描述简洁）
  public boolean hasLChild()
  { return (1+rank*2 < T.getSize()) ? true : false; }
//返回当前节点的左孩子
  public BinTreePosition getLChild()
  { return hasLChild() ? (BinTreePosition) (T.getAtRank(1+rank*2)) : null; }

//判断是否有右孩子（为使代码描述简洁）
  public boolean hasRChild()
  { return (2+rank*2 < T.getSize()) ? true : false; }
//返回当前节点的右孩子
  public BinTreePosition getRChild()
  { return hasRChild() ? (BinTreePosition) (T.getAtRank(2+rank*2)) : null; }

//返回当前节点后代元素的数目
  public int getSize()  {
    int size = 1;
    if (hasLChild())  size += getLChild().getSize();
    if (hasRChild())  size += getRChild().getSize();

    return size;

  }

//返回当前节点的高度
  public int getHeight()  {
    int hL = hasLChild() ? getLChild().getHeight() : -1;
    int hR = hasRChild() ? getRChild().getHeight() : -1;
    return 1 + Math.max(hL, hR);
  }

//返回当前节点的深度
```

```
  public int getDepth() {
    return hasParent() ?  1+getParent().getDepth() : 0;
  }
}
```

代码四.9 基于秩实现的完全二叉树节点

■ 完全二叉树类的 Java 实现

代码四.10 给出了完全二叉树类基于向量的具体实现。

```
/*
 * 基于向量实现的完全二叉树
 */

package dsa;

public class ComplBinTree_Vector extends BinTree_LinkedList implements ComplBinTree {
  private Vector T;//向量

//构造方法：默认的空树
  public ComplBinTree_Vector()
  { T = new Vector_ExtArray();  root = null; }

//构造方法：按照给定的节点序列，批量式建立完全二叉树
  public ComplBinTree_Vector(Sequence s)
  { this(); if (null !=s) while (!s.isEmpty()) addLast(s.removeFirst()); }

/*---------- BinaryTree接口中各方法的实现 ----------*/
//返回树根（重写）
  public BinTreePosition getRoot()
  { return T.isEmpty() ? null : posOfNode(0); }

//判断是否树空（重写）
  public boolean isEmpty()
  { return T.isEmpty(); }

//返回树的规模（重写）

  public int getSize()

  { return T.getSize(); }

//返回树（根）的高度（重写）
  public int getHeight()
  {return isEmpty() ? -1 : getRoot().getHeight(); }

/*---------- ComplBinTree接口中各方法的实现 ----------*/
```

```
//生成并返回一个存放e的外部节点，该节点成为新的末节点
  public BinTreePosition addLast(Object e) {
    BinTreePosition node = new ComplBinTreeNode_Rank(T, e);
    root = (BinTreePosition) T.getAtRank(0);
    return node;
  }

//删除末节点，并返回其中存放的内容
  public Object delLast() {
    if (isEmpty())  return null;//若树（堆）已空，无法删除
    if (1 == getSize()) root = null;//若删除最后一个节点，则树空
    return T.removeAtRank(T.getSize()-1);
  }

//返回按照层次遍历编号为i的节点的位置，0 <= i < size()
  public BinTreePosition posOfNode(int i) {
    return (BinTreePosition)T.getAtRank(i);
  }
}
```

代码四.10 基于向量实现的完全二叉树

第五章

优先队列

存放数据只是数据结构的基本功能之一，数据结构另一方面的典型用途就是按照次序将数据组织起来。比如第 §2.2 节所介绍的队列结构，就可以用来描述和处理日常生活中的很多问题：在银行排队等候接受服务的客户，提交给网络打印机的打印任务，等等。在这类问题中，无论是客户还是打印任务，接受服务或处理的次序完全取决于其出现的时刻——首先到达的客户优先接受服务，首先提交的打印任务优先处理——即遵循所谓的“先入先出”原则。然而在很多实际应用问题中，这一简单的原则并不能使效率达到最高。试想若干病人正在某所医院唯一的门诊处等候接受治疗，他们已经按照“先入先出”的次序排成一个队列等待接受治疗，而此时忽然送来一位骨折的病人。要是固守“先到者优先接受服务”的原则，那么只有等到目前已经到达的所有病人都接受完治疗之后才能处理这位病人，显然，那样的话该病人将承受更长时间的巨大痛苦，而且更重要的是将会贻误治疗的最佳时机。因此，在医院中实际采取的是另一种做法——优先治疗这位骨折的病人。试想此时又送来一位心脏病突发患者，那么医生肯定会暂时把骨折病人放在一边，转而优先抢救心脏病人。

由此可以看出，在决定病人接受治疗的次序时，除了他们到达医院的先后次序，更主要的将取决于病情的严重程度。由这类问题可以抽象出本章将要讨论的优先队列（Priority queue）结构。这一结构在很多应用领域都可以派上大用场，比如各种事件队列的模拟、操作系统中多任务的调度及中断机制、采用词频调整策略的输入法等。另外，优先队列也是很多高级算法的基础，比如本书将要介绍的Huffman编码（第 5.10.4 节）、堆排序算法（第 §5.9 节）都要利用优先队列，而在采用空间扫描策略的算法中，优先队列是组织事件队列的最佳形式。

优先队列之所以具有广泛的用途，是得益于其高效率以及实现的简捷性。正如我们将要看到的，如果基于堆来实现优先队列，则其初始化构造可以在线性时间内完成，而每次插入或删除操作都可以在 $\mathcal{O}(\log n)$ 的时间内完成。

§5.1 优先级、关键码、全序关系与优先队列

正如医院根据病情严重程度确定病人接受治疗的次序一样，优先队列中各对象之间的次序是由它们共同的某个特征、属性或指标决定的，我们称之为“关键码”（Key）。关键码本身也是一个对象。

在不同的应用问题中，对象的关键码也不尽相同。有些情况下不同对象的关键码互异，有些情况下则允许多个对象的关键码雷同。有时对象的关键码一成不变，有时对象的关键码则可以修改。简单的关键码可以是一个数字、一个字符或一个字符串，复杂的关键码则可能是由多个数字和字符组合而成的复合对象。有的关键码可能就是对象内部的某一实例变量，而有的关键码则并非对象自身的天然属性，必须由具体的应用专门指定。

因此，一种很自然的构想就是将关键码定义为任意对象。然而，为此必须建立一种统一的、相容的形式，以支持不同对象之间的比较，否则无法确定优先级。

实际上，作为优先队列的一个基本要求，在关键码之间必须能够定义某种全序关系（Total order relation）。具体来说，任何两个关键码都必须能够比较大小。请注意，这里的“大小”是广义的。如果将这种全序关系用“≤”表示，则该关系还必须满足以下三条性质：

- 自反性：对于任一关键码 k，都有 $k \le k$
- 反对称性：若 $k_1 \le k_2$ 且 $k_2 \le k_1$，则 $k_1 = k_2$
- 传递性：若 $k_1 \le k_2$ 且 $k_2 \le k_3$，则 $k_1 \le k_3$

不难证明，具有以上性质的关系实际上就是所有对象之间的一个线性次序，因此必然是自洽的，也就是说，不会由此导出相互矛盾的结论。

所谓的优先队列也是对象的一种容器，只不过其中的每个对象都拥有一个关键码，在它们的关键码之间存在满足上述性质的某种全序关系“≤”。关键码可以是在对象插入优先队列时被人为赋予的，也可能就是对象本身具有的某一属性。

§5.2 条目与比较器

在给出优先队列的具体定义之前，还有两个问题有待明确：

- 在优先队列中，如何记录和维护各对象与其关键码之间的关联关系？
- 如何具体实现第 §5.1 节提出的全序关系，从而通过对象的比较能够找出其中的最小者？

为了解决这两个问题，下面我们需要分别构造出条目和比较器这两个类。

5.2.1 条目

引入条目这一概念，旨在解决上面的前一个问题。所谓一个条目（Entry），就是由一个对象及其关键码合成的一个对象，它反映和记录了二者之间的关联关系[1]。这样，通过将条目对象作为优先队列的元素，即可记录和维护原先对象与其关键码之间的关联关系。

代码五.1 给出了条目的接口描述。

```
/*
 * 条目接口
 */

package dsa;
```

[1] 这里，我们运用了面向对象程序设计的一种典型技巧——合成模式（Comosition p attern）——也就是说，将若干对象合成起来，构成一个更大的复合对象。比如，每个条目对象都是由一个数据对象及其关键码对象合成得到的。根据应用问题的不同需要，可以由更多个对象得到一个合成对象。通过一些技巧（比如采用列表结构），同类的合成对象甚至可以由任意个对象构成。

```
public interface Entry {

//取条目的关键码

  public Object getKey();
//修改条目的关键码，返回此前存放的关键码
  public Object setKey(Object k);

//取条目的数据对象
  public Object getValue();
//修改条目的数据对象，返回此前存放的数据对象
  public Object setValue(Object v);
}
```

代码五.1 条目接口

代码五.2 给出了默认条目类的实现。

```
/*
 * 默认条目
 */

package dsa;

public class EntryDefault implements Entry {
  protected Object key;
  protected Object value;

/*************************** 构造函数 ***************************/
  public EntryDefault(Object k, Object v)
  { key = k;  value = v; }

/*************************** Entry接口方法 ***************************/
//取条目的关键码
  public Object getKey()
  { return key; }
//修改条目的关键码，返回此前存放的关键码
  public Object setKey(Object k)
  { Object oldK = key;  key = k;  return oldK; }

//取条目的数据对象
  public Object getValue()
  { return value; }
//修改条目的数据对象，返回此前存放的数据对象
  public Object setValue(Object v)
  { Object oldV = value;  value = v;  return oldV; }
}
```

代码五.2 优先队列使用的默认条目

5.2.2 比较器

后一个问题似乎更难解决——毕竟，与 C++之类的语言不同，Java 并不支持对比较操作符（">"、"<"或"=="等）的重载。

一种直截了当的办法，就是针对不同的元素类型以及元素大小比较的不同方式，分别实现一种优先队列。这一策略的不足显而易见——通用性差，而且需要重复编写大量的代码。

另一种办法，就是实现基于某种 Comparable 接口实现一个关键码类，并将所有通常的比较方法封装起来，以支持关键码之间的比较。采取这一策略，我们只需编写一个队列类即可处理各种类型的关键码。然而按照这一策略，关键码的比较方式完全取决于关键码本身的类型，而在很多情况下，这并不能最终决定关键码的具体比较方式。以关键码"3"和"12"为例：若它们是整型对象，则按照整数大小的比较规则，可以判断前者小于后者；若它们是字符串对象，则按照词典序规则，判断结果正好相反。

既然如此，不如基于 Comparator 接口专门实现一个独立于关键码之外的比较器类，由它来确定具体的比较规则。在创建每个优先队列时，只要指定这样一个比较器对象，即可按照该比较器确定的规则，在此后进行关键码的比较。这一策略的另一优点在于，一旦不想继续使用原先的比较器对象，可以随时用另一个比较器对象将其替换掉，而不用重写优先队列本身。

5.2.3 Comparator 接口及其实现

按照上述思路，我们可以定义Comparator接口如 代码五.3 所示：

```
/*
 * 比较器接口
 */

package dsa;

public interface Comparator {
  public int compare(Object a, Object b);//若a>（=或<）b，返回正数、零或负数
}
```

代码五.3 Compar ator接口（也可以直接采用java.util.Comparator接口）

这一接口中指定义了一个方法 compare(a, b)，根据对象 a 和 b 的大小关系，它分别返回一个正数、零或负数。

比如，试考虑平面上坐标均为整数的点，即所谓的整格点（Grid points）。

```
/*
 * 平面整格点
 */
```

```
package dsa;

public class Point2D {
  private int x, y;



  public Point2D(int xx, int yy)
  { x = xx; y = yy; }

  public int getX()
  { return x; }

  public int getY()
  { return y; }
}
```

代码五.4 二维点对象

若需要在这样的点之间按照坐标的词典序比较大小，就可以基于 Comparator 接口实现如下类：

```
/*
 * 按照字典序确定平面点之间次序的比较器
 */

package dsa;

public class ComparatorLexicographic implements Comparator {
  public int compare(Object a, Object b) throws ClassCastException {
    int ax = ((Point2D) a).getX();
    int ay = ((Point2D) a).getY();
    int bx = ((Point2D) b).getX();
    int by = ((Point2D) b).getY();
    return (ax != bx) ? (ax - bx) : (ay - by);
  }
}
```

代码五.5 按照坐标词典序比较平面点大小的比较器

默认情况下，我们将使用如下比较器：

```
/*
 * Comparable对象的默认比较器
 */

package dsa;

public class ComparatorDefault implements Comparator {
  public ComparatorDefault() {}
  public int compare(Object a, Object b) throws ClassCastException {
```

```
    return ((Comparable) a).compareTo(b);
  }
}
```

代码五.6 优先队列的默认比较器

§5.3 优先队列 ADT 及 Java 接口

下面，我们给出优先队列的具体定义。

5.3.1 ADT 描述

优先队列 Q 必须支持以下两个基本操作：

表五.1 优先队列ADT支持的操作

操作方法	功能描述
getSize():	返回 Q 的规模 输入：无 输出：整数
isEmpty():	判断 Q 是否为空 输入：无 输出：布尔标志
getMin():	若 Q 非空，则返回其中的最小条目（并不删除） 否则，报错 输入：无 输出：条目
insert(k, obj):	将对象 obj 与关键码 k 合成一个条目，将其插入 Q 中，并返回该条目 输入：一个对象及一个关键码 输出：条目
delMin():	若 Q 非空，则从其中摘除关键码最小的条目，并返回该条目 否则，报错 输入：无 输出：条目

其中，getSize()和 isEmpty()是一般容器都须支持的方法，getMin()属于查询方法（它并不修改优先队列的内容），insert()和 delMin()则是优先队列特有的方法。需要强调的是，这里允许在同一优先队列中出现具有相同关键码的多个条目。

我们注意到，与 第二章介绍的栈与队列一样地，优先队列抽象类型的定义也非常简单。之所以如此，是因为对优先队列的操作过程完全由给定的关键码决定，而不必像 第三章所介绍的序列之类的结构，需要为每个操作指定秩或者位置。

5.3.2 Java 接口

按照上述优先队列ADT，可以写出如 代码五.7 所示的Java接口：

```
/*
 * 优先队列接口

 */


package dsa;

public interface PQueue {
//统计优先队列的规模
  public int getSize();

//判断优先队列是否为空
  public boolean isEmpty();

//若Q非空，则返回其中的最小条目（并不删除）;否则，报错
  public Entry getMin() throws ExceptionPQueueEmpty;

//将对象obj与关键码k合成一个条目，将其插入Q中，并返回该条目
  public Entry insert(Object key, Object obj) throws ExceptionKeyInvalid;

//若Q非空，则从其中摘除关键码最小的条目，并返回该条目；否则，报错
  public Entry delMin() throws ExceptionPQueueEmpty;
}
```

代码五.7 优先队列接口

其中ExceptionPQueueEmpty和ExceptionKeyInvalid意外错的定义，分别参见 代码五.8 和 代码五.9。

```
/*
 * 当试图对空的优先队列应用getMin()或delMin()方法时，本意外将被抛出
 */

package dsa;

public class ExceptionPQueueEmpty extends RuntimeException {
  public ExceptionPQueueEmpty(String err) {
    super(err);
  }
}
```

代码五.8 Ex ceptionPQueueEmpty意外错

```
/*
 * 当试图使用非法关键码时，本意外将被抛出
 */

package dsa;

public class ExceptionKeyInvalid extends RuntimeException {
  public ExceptionKeyInvalid(String err) {

    super(err);

  }
}
```

代码五.9 Ex ceptionKeyInvalid意外错

5.3.3 基于优先队列的排序器

利用优先队列的ADT，我们可以处理第 1.1.3 节定义的排序问题：给定由n个元素组成的序列S，如果其中各元素的关键码之间能够定义某种全序关系，试将它们按（非降）序排列。这一称作优先队列排序（Priority Queue Sorting）的算法分为两步：取一个空的优先队列Q，将S中的元素逐一摘出并插入Q中；然后，连续调用n次delMin()方法，从Q中取出n个元素并依次填回至S中。如 代码五.10 所示，基于如 代码一.1 所定义的排序器接口，可以实现基于优先队列的排序器如下：

```
/*
 * 基于优先队列的排序器
 */

package dsa;

public class Sorter_PQueue implements Sorter {
  private Comparator C;

  public Sorter_PQueue()
  { this(new ComparatorDefault()); }

  public Sorter_PQueue(Comparator comp)
  { C = comp; }

  public void sort(Sequence S) {
    Sequence T = new Sequence_DLNode();//为批处理建立优先队列而准备的序列
    while (!S.isEmpty()) {//构建序列T
      Object e = S.removeFirst();//逐一取出S中各元素
      T.insertLast(new EntryDefault(e, e));//用节点元素本身作为关键码
    }
//    PQueue pq = new PQueue_UnsortedList(C, T);
//    PQueue pq = new PQueue_SortedList(C, T);
    PQueue pq = new PQueue_Heap(C, T);
```

```
    while(!pq.isEmpty()) {//从优先队列中不断地
      S.insertLast(pq.delMin().getValue());//取出最小元素，插至序列末尾
      System.out.println("\t:\t" + S.last().getElem());
    }
  }
}
```

代码五.10 基于优先队列的排序算法

该算法的正确性由如下事实保证：序列 S 已经排序，当且仅当在 S 的任何一个后缀中，首元素都是最小的。需要特别指出的是，这里允许关键码出现重复。

该算法需要执行序列结构的n次removeAtRank()操作和insertAtRank()操作，还要执行优先队列结构的n次insert()操作和n次delMin()操作。然而，鉴于目前我们还不清楚优先队列的具体实现，无法确定insert()操作和delMin()操作所能达到的效率，所以还不能界定该算法的运行时间。实际上，与其说 代码五.10 给出了一个排序算法，还不如说这仅仅只是排序算法的一个框架，正如我们将在本章稍后要看到的那样，包括选择排序、插入排序和堆排序在内的多种排序算法，都可以纳入这一框架之中。

§5.4 用向量实现优先队列

在实现优先队列时，首先考虑到的就是利用第 §3.1 节介绍的向量结构。这一构想很容易兑现，比如，若用无序的向量来实现优先队列，则通过调用insertAtRank(getSize())将新条目接至向量末尾，即可实现insert(k, e)方法，而且这只需$\mathcal{O}$(1)时间。然而delMin()方法的实现则不那么幸运——我们只能通过getAtRank()方法逐一检查向量中的每个元素并通过比较找出最小者，这需要$\mathcal{O}$(n)的时间！

提高 delMin()效率的一种方法，就是保证向量元素按非升序排列——这样，只需直接调用 removeAtRank(getSize()-1)即可摘除最末尾（最小）的元素，从而在 $\mathcal{O}$(1)时间内完成 delMin()操作。然而，为了保证向量的有序性，每次插入新条目时都必须通过查找确定其位置。虽然查找可以在 $\mathcal{O}$(**log**n)的时间内完成，但为了给新条目腾出空位，后续条目需要顺次后移——这需要 $\mathcal{O}$(n)的时间。

由此看来，我们似乎无法兼顾 insert()和 delMin()操作的效率，除非不再利用简单的向量结构来实现优先队列。

当然，基于简单向量结构的策略在某些特殊场合也不失为一种选择。比如，在通常情况下，虽然 delMin()操作真正实施的次数不会超过 insert()操作的次数，但也大体相当。而有些应用问题中，前者的执行次数要远远少于后者。在这种情况下，用无序向量来实现优先队列的确可行。

§5.5 用列表实现优先队列

下面，我们将利用第 §3.2 节介绍的列表结构来实现优先队列类——也就是说，把条目作为列表的元素。与基于向量的实现类似地，基于列表的实现也有两种策略：使用无序列表或有序列表。

5.5.1 基于无序列表的实现及分析

我们首先给出如 代码五.11 所示的具体实现：

```
/*
 * 基于无序列表实现的优先队列
 */

package dsa;
```

```
public class PQueue_UnsortedList implements PQueue {
  private List L;
  private Comparator C;

//构造方法（使用默认比较器）
  public PQueue_UnsortedList()
  { this(new ComparatorDefault(), null); }

//构造方法（使用指定比较器）
  public PQueue_UnsortedList(Comparator c)
  { this(c, null); }

//构造方法（使用指定初始元素）
  public PQueue_UnsortedList(Sequence s)
  { this(new ComparatorDefault(), s); }

//构造方法（使用指定比较器和初始元素）
  public PQueue_UnsortedList(Comparator c, Sequence s) {
    L = new List_DLNode();
    C = c;
    if (null != s)
      while (!s.isEmpty()) {
        Entry e = (Entry)s.removeFirst();
        insert(e.getKey(), e.getValue());
      }
  }

//统计优先队列的规模
  public int getSize()
  { return L.getSize(); }

//判断优先队列是否为空
  public boolean isEmpty()
```

```
  { return L.isEmpty(); }

//若Q非空，则返回其中的最小条目（并不删除）;否则，报错
  public Entry getMin() throws ExceptionPQueueEmpty {
    if (L.isEmpty())
      throw new ExceptionPQueueEmpty("意外: 优先队列空");
    Position minPos = L.first();
    Position curPos = L.getNext(minPos);
    while (null != curPos)//依次检查所有位置，找出最小条目
      if (0 < C.compare(minPos.getElem(), curPos.getElem()))
        minPos = curPos;
    return (Entry) minPos.getElem();
  }

//将对象obj与关键码k合成一个条目，将其插入Q中，并返回该条目
  public Entry insert(Object key, Object obj) throws ExceptionKeyInvalid {

    Entry entry = new EntryDefault(key, obj);//创建一个新条目

    L.insertLast(entry);//接至列表末尾
    return(entry);
  }

//若Q非空，则从其中摘除关键码最小的条目，并返回该条目；否则，报错
  public Entry delMin() throws ExceptionPQueueEmpty {
    if (L.isEmpty())
      throw new ExceptionPQueueEmpty("意外：优先队列空");
    Position minPos = L.first();
    Iterator it = L.positions();
    while (it.hasNext()) {//依次检查所有位置，找出最小条目
      Position curPos = (Position) (it.getNext());
//      System.out.println("\t" + ((Entry)(curPos.getElem())).getKey());
      if  (0 <C.compare(
                ((Entry)(minPos.getElem())).getKey(),
                ((Entry)(curPos.getElem())).getKey())
          )
        minPos = curPos;
    }
    return (Entry) L.remove(minPos);
  }
}
```

代码五.11 基于无序列表实现的优先队列

可以看到，利用列表 ADT 中对应的方法，即可直接实现优先队列的 getSize()和 isEmpty()方法。

为了插入一个被赋予关键码key的对象obj，我们首先创建一个条目对象entry = (key, obj)，然后调用insertLast(entry)方法将其接至列表尾部。根据第 §3.2 节的结论，这只需$\mathcal{O}$(1)时间。

显然，如此构造出的列表将是无序的，故无论是 getMin()还是 delMin()方法，都需要对整个列表扫描一遍，才能从中找出最小的条目。假定对比较器 compare()方法的每次调用只需 $\mathcal{O}$(1)时间，则我们需要花费 $\mathcal{O}$(n)时间才能找到并取出优先队列中的最小条目——如此低的效率实在难以令人满意。而且实际上，只有在对每一条目都检查过至少一次之后，才能确定最小条目的位置，因此更准确地说，这一过程的时间复杂度为Θ(n)。

5.5.2 基于有序列表的实现及分析

与基于向量的实现一样，为了提高delMin()方法的效率，可以要求所有条目按非升次序排列。这样，只要直接调用removeLast()，即可在$\mathcal{O}$(1)时间内摘除最末尾（最小）的条目。具体实现如 代码五.12 所示。

```
/*
 * 基于有序（非升）列表实现的优先队列
 */
```

```
package dsa;

public class PQueue_SortedList implements PQueue {
  private List L;
  private Comparator C;

//构造方法（使用默认比较器）
  public PQueue_SortedList()
  { this(new ComparatorDefault(), null); }

//构造方法（使用指定比较器）
  public PQueue_SortedList(Comparator c)
  { this(c, null); }

//构造方法（使用指定初始元素）
  public PQueue_SortedList(Sequence s)
  { this(new ComparatorDefault(), s); }

//构造方法（使用指定比较器和初始元素）
  public PQueue_SortedList(Comparator c, Sequence s) {
    L = new List_DLNode();
    C = c;
    if (null != s)
      while (!s.isEmpty()) {
        Entry e = (Entry)s.removeFirst();
        insert(e.getKey(), e.getValue());
      }
  }
```

```
//统计优先队列的规模
  public int getSize()
  { return L.getSize(); }

//判断优先队列是否为空
  public boolean isEmpty()
  { return L.isEmpty(); }

//若Q非空，则返回其中的最小条目（并不删除）；否则，报错
  public Entry getMin() throws ExceptionPQueueEmpty {
    if (L.isEmpty())
      throw new ExceptionPQueueEmpty("意外：优先队列空");
    return (Entry) L.last();
  }

//将对象obj与关键码k合成一个条目，将其插入Q中，并返回该条目
  public Entry insert(Object key, Object obj) throws ExceptionKeyInvalid {
    Entry entry = new EntryDefault(key, obj);//创建一个新条目
    if (L.isEmpty()//若优先队列为空
     || (0 > C.compare(((Entry) (L.first().getElem())).getKey(), entry.getKey())))

        //或新条目是当前最大者

      L.insertFirst(entry);//则直接插入至表头
    else {//否则
      Position curPos = L.last();//从尾条目开始
      while (0 > C.compare(((Entry)(curPos.getElem())).getKey(), entry.getKey()))
          curPos = L.getPrev(curPos);//不断前移，直到第一个不小于entry的条目
      L.insertAfter(curPos, entry);//紧接该条目之后插入entry
    }
    return(entry);
  }

//若Q非空，则从其中摘除关键码最小的条目，并返回该条目；否则，报错
  public Entry delMin() throws ExceptionPQueueEmpty {
    if (L.isEmpty())
      throw new ExceptionPQueueEmpty("意外：优先队列空");
    return (Entry) L.remove(L.last());
  }
}
```

代码五.12 基于有序列表实现的优先队列

可以看到，为了维护列表的有序性，在插入条目时，必须顺序扫描现有的条目，通过比较确定正确的插入位置。在最好的情况下，只需一次比较；但在最坏情况下，却需要做 n 次比较。因此，

如此实现的 insert()方法的时间复杂度为 $\mathcal{O}(n)$。请读者自行验证：在等概率的情况下，平均复杂度也是 $\mathcal{O}(n)$。

§5.6 选择排序与插入排序

第 5.3.3 节曾介绍过基于优先队列ADT的排序器PQSorter（代码五.10），对应的算法分为两个阶段：首先将待排序的对象组织为一个优先队列，然后不断摘除当前的最小条目。由于当时我们尚不清楚优先队列的具体实现方式，所以未能准确地界定其时间复杂度。本节将针对第 §5.5 节中基于列表实现的两种优先队列，分别讨论排序器PQSorter的两种变型，并就其效率做一比较。这里依然假定，比较器的每一次compare()操作都可以在$\mathcal{O}(1)$时间内完成。

5.6.1 选择排序

如果采用无序列表来实现优先队列，则 PQSort 的前一步中的 n 次 insert()操作总共只需要 $\mathcal{O}(n)$ 时间。然而，随后一步的效率却太低了——为了执行每一次 delMin()操作，需要耗费的时间都将正比于优先队列当前的规模，因此第二阶段总共需要耗费 $\mathcal{O}(n^2)$的时间。

表五.2 基于无序列表的选择排序实例

操作方法	输出	基于无序列表的优先队列	排序结果
创建	N/A	{} N/A	
insert(3) N/A		{3} N/A	
insert(1) N/A		{1, 3}	N/A
insert(5) N/A		{5, 1, 3}	N/A
insert(2) N/A		{2, 5, 1, 3}	N/A
insert(4) N/A		{4, 2, 5, 1, 3}	N/A
delMin()	1	{4, 2, 5, 3}	{1}
delMin()	2	{4, 5, 3}	{1, 2}
delMin()	3	{4, 5}	{1, 2, 3}
delMin()	4	{5}	{1, 2, 3, 4}
delMin()	5	{}	{1, 2, 3, 4, 5}

5.6.2 插入排序

反之，如果采用有序列表来实现优先队列，则每次 delMin()操作都可以在常数时间内完成，故 PQSort 算法后一阶段总的时间复杂度可以降至 $\mathcal{O}(n)$。然而这并非没有代价。为了维护列表的有序性，前一阶段中的每一次 insert()操作，都需要对当前的列表进行扫描，以确定新条目的插入位置。在最坏的情况下，为此需要花费的时间将正比于优先队列当前的规模。因此，前一阶段的总体时间复杂度为 $\mathcal{O}(n^2)$。

表五.3 基于有序列表的插入排序实例

操作方法	输出	基于有序列表的优先队列	排序结果
创建	N/A	{} N/A	
insert(3) N/A		{3} N/A	
insert(1) N/A		{3, 1}	N/A
insert(5) N/A		{5, 3, 1}	N/A
insert(2) N/A		{5, 3, 2, 1}	N/A
insert(4) N/A		{5, 4, 3, 2, 1}	N/A
delMin()	1	{5, 4, 3, 2}	{1}
delMin()	2	{5, 4, 3}	{1, 2}
delMin()	3	{5, 4}	{1, 2, 3}
delMin()	4	{5}	{1, 2, 3, 4}
delMin()	5	{}	{1, 2, 3, 4, 5}

5.6.3 效率比较

表面看来，无论采用何种实现方式，PQSorter的效率都是一样的$\mathcal{O}(n^2)$，然而二者之间还是存在着细微而有趣的差别。实际上，正如第 5.5.1 节所指出的，若采用无序列表，delMin()方法的复杂度应为$\Theta(n)$。也就是说，无论如何，每次delMin()操作都需耗费线性的时间。因此，选择排序算法的效率也是$\Theta(n^2)$。

插入排序则不然。实际上，若采用有序列表，则 insert()操作只有在最坏的情况下才需要 $\mathcal{O}(n)$的时间；在最好的情况下，只需常数时间。比如，如果输入的元素本身就是按逆序排列的，则每次insert()操作都只需扫描列表的首元素即可确定新元素的插入位置（即插入后应该成为新的首元素）。也就是说，对于这类输入，每次 insert()操作都属于最好情况，于是前一阶段（进而整个算法）总共只需 $\mathcal{O}(n)$的时间，在这种情况下整个算法只需运行 $\mathcal{O}(n)$时间。

以上分析告诉我们，插入排序算法与选择排序算法的不同，在于前者的复杂度不仅取决于输入的规模，同时也与输入序列的次序有关。请读者自行证明：插入排序算法的时间复杂度为 $\mathcal{O}(n+I)$，其中 I 为输入序列中逆序对（即相对位置颠倒的一对元素）的数目。

不过，就大 O 记号所反映的复杂度上界而言，插入排序的效率依然不能令人满意——毕竟，在最坏情况下该算法仍需要运行 $\mathcal{O}(n^2)$的时间。为了进一步提高其效率，必须在优先队列的实现方面有所突破，而这正是下面将要讨论的主题。

§5.7 堆的定义及性质

实现优先队列最高效的方式，就是借助堆（Heap）结构。利用这一结构，优先队列的getMin()操作可以在$\mathcal{O}(1)$时间内完成，而insert()和delMin()操作均可以在$\mathcal{O}(\log n)$的时间内完成——相对于第

§5.5 节中的两种实现，效率几乎提高的一个线性因子。之所以能够达到如此高的效率，关键在于放弃了列表结构而转向树形结构。

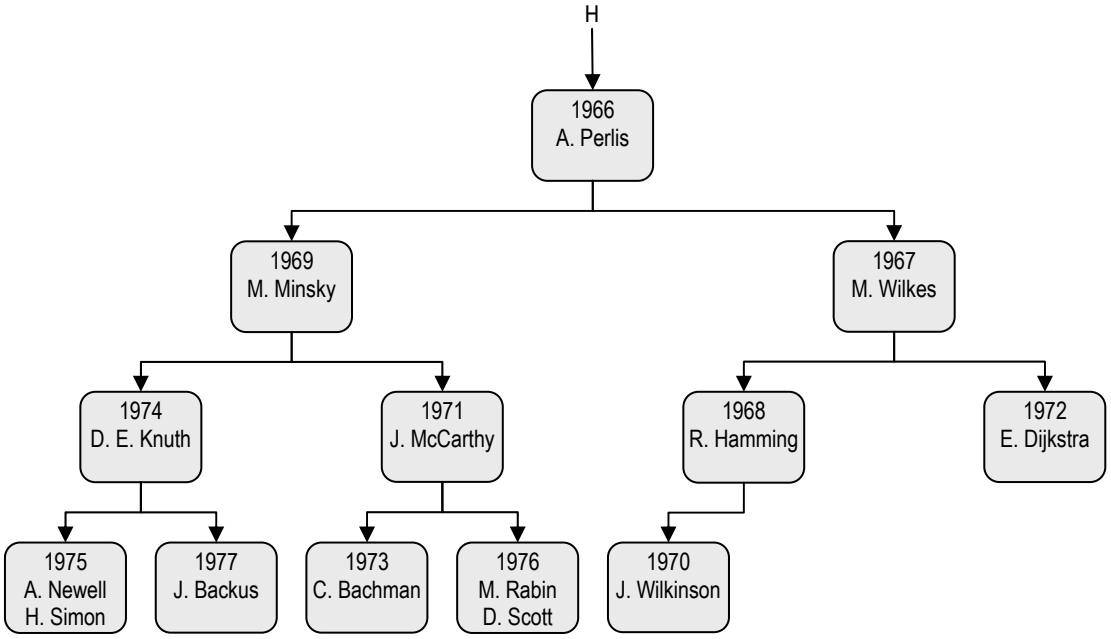
此前的两种实现之所以效率都不高，原因在于对优先队列定义的理解过于机械，任何时刻都分毫不差地保存了整个集合的全序关系，也就是说，所有元素始终都是按照全序排列的。只要稍微深入地对优先队列的定义做一推敲就不难发现，实际上，只要能够随时给出全局的最小条目即可，至于次小者、第三个最小者以及第 i 个最小者，我们并不关心。堆结构正是充分利用了优先队列的这一特点，在任何时候只保存了整个集合的一个偏序关系，从而这一结构在时间复杂度方面有了实质性的改进。

5.7.1 堆结构

由若干条目组成的一个堆[1]H，就是满足以下两条性质的一棵二叉树：

- 结构性：H 中各元素的联结关系应符合二叉树的结构要求（其根节点称作堆顶）；
- 堆序性：就其关键码而言，除堆顶外的任何条目都不小于其父亲。

在上下文不致发生歧义时，我们将直接用节点来指代其中存放的条目。



图五.1 以获奖年份为关键码，图灵奖前12届得主所构成的一个堆

下面这则定理，指出了堆结构最重要的一个特性：

定理五.1 H 中的最小条目必处于堆顶。

〖证明〗

[1] 请注意：这里的“堆”与操作系统或运行环境中的“内存堆”（Memory heap）没有任何直接联系。

根据堆序性，沿着任一节点到堆顶的通路，各条目的关键码单调非升。因此，最小条目不可能出现在堆顶之外的其它位置。 ❑

当然，也可以对称地定义堆序性：除堆顶外的任何条目都不大于其父亲。相应地，在如此定义的堆中，最大条目必处于堆顶。因此，按前一种形式定义的堆称作“小顶堆”，后者则称为“大顶堆”。

小顶堆和大顶堆是相对的，甚至可以相互转换。我们注意到，节点之间的大小关系完全取决于所使用的比较器。只要将比较器的 compare()方法对称地改写，就可以得到一个新的比较器，而按照新的比较器，无需对原有结构做任何更动，原先的小（大）顶堆即可转化为大（小）顶堆。

5.7.2 完全性

正如 第七章将要指出的，二叉搜索树的效率在很大程度上取决于树的高度（深度）。在此，为了降低堆的高度以提高操作的效率，我们对其增加一项要求：

✍ 完全性：堆必须是一棵完全二叉树（参见第 4.1.8 节）

在具有这一附加性质的堆中，除堆顶外还有一个特殊的节点——末节点（Last node），亦即在层次遍历（第 4.3.5 节）该堆时最后接受访问的那个节点。以如 图五.2 所示的堆为例，其末节点为“1970, J. Wilkinson”。

实际上，上述定义可以推广至 m 叉树形结构（m>2），相应的堆结构称作 m 叉堆，而如上定义的堆也称作“二叉堆”，这是本书将着重讨论的。

§5.8 用堆实现优先队列

现在，我们可以来回答第 §5.6 节中尚未解决的问题：在利用优先队列进行排序时，如何兼顾insert()和delMin()两种操作，从而使得二者都能快速完成？

根据 引理四.1，满足完全性的堆必然不会太高。具体地，若insert()、delMin()操作的时间复杂度线性正比于堆的高度，则它们都可以在$\mathcal{O}(\log n)$的时间内完成。下面我们就来讨论如何达到这一效率。我们将会看到，利用堆结构，的确可以在$\mathcal{O}(\log n)$的时间内完成这两种操作。下面，我们首先给出堆结构的具体实现，然后对各主要算法进行讲解与分析。

5.8.1 基于堆的优先队列及其实现

利用第 §4.6 节实现的完全二叉树，可以进一步实现如 代码五.13 所示的优先队列。

```
/*
 * 利用堆实现优先队列
 */

package dsa;

public class PQueue_Heap implements PQueue {
  private ComplBinTree H;//完全二叉树形式的堆
```

```
  private Comparator comp;//比较器

//构造方法
  public PQueue_Heap()
  { this(new ComparatorDefault(), null); }

//构造方法: 默认的空优先队列
  public PQueue_Heap(Comparator c)
  { this(c, null); }

//构造方法: 根据某一序列直接批量式构造堆算法, S中元素都是形如(key, value)的条目
  public PQueue_Heap(Sequence S)
  { this(new ComparatorDefault(), S); }

//构造方法: 根据某一序列直接批量式构造堆算法, s中元素都是形如(key, value)的条目
  public PQueue_Heap(Comparator c, Sequence s) {
    comp = c;
    H = new ComplBinTree_Vector(s);
    if (!H.isEmpty()) {
      for (int i = H.getSize()/2-1; i >= 0; i--)//自底而上
        percolateDown(H.posOfNode(i));//逐节点进行下滤
    }
  }

/*-------- PQueue接口中定义的方法 --------*/
//统计优先队列的规模
  public int getSize()

  { return H.getSize(); }


//判断优先队列是否为空
  public boolean isEmpty()
  { return H.isEmpty(); }

//若Q非空, 则返回其中的最小条目（并不删除）;否则, 报错
  public Entry getMin() throws ExceptionPQueueEmpty {
    if (isEmpty())  throw new ExceptionPQueueEmpty("意外: 优先队列为空");
    return (Entry) H.getRoot().getElem();
  }

//将对象obj与关键码k合成一个条目, 将其插入Q中, 并返回该条目
  public Entry insert(Object key, Object obj) throws ExceptionKeyInvalid  {
    checkKey(key);
    Entry entry = new EntryDefault(key, obj);
    percolateUp(H.addLast(entry));
    return entry;
  }
```

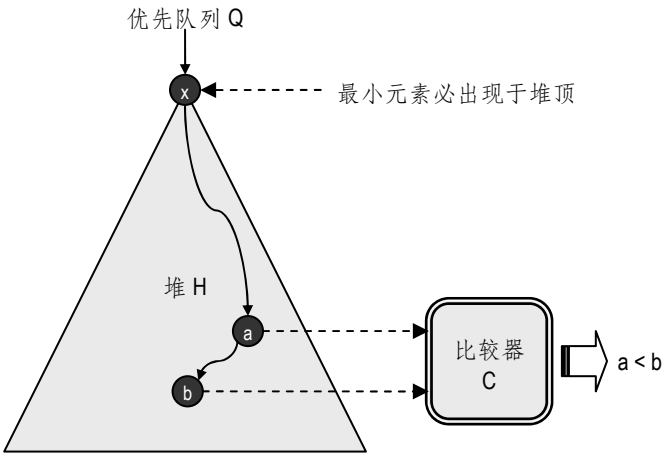
```
//若Q非空，则从其中摘除关键码最小的条目，并返回该条目；否则，报错
  public Entry delMin() throws ExceptionPQueueEmpty {
    if (isEmpty())  throw new ExceptionPQueueEmpty("意外：优先队列为空");
    Entry min = (Entry)H.getRoot().getElem();//保留堆顶
    if (1 == getSize())//若只剩下最后一个条目
      H.delLast();//直接摘除之
    else {//否则
      H.getRoot().setElem(((ComplBinTreeNode_Rank)H.delLast()).getElem());
        //取出最后一个条目，植入堆顶
      percolateDown(H.getRoot());
    }
    return min;//返回原堆顶
  }

/*-------- 辅助方法 --------*/
//检查关键码的可比较性
  protected void checkKey(Object key) throws ExceptionKeyInvalid {
    try {
      comp.compare(key, key);
    } catch (Exception e) {
      throw new ExceptionKeyInvalid("无法比较关键码");
    }
  }

//返回节点v（中所存条目）的关键码
  protected Object key(BinTreePosition v) {
    return ((Entry)(v.getElem())).getKey();
  }


/*-------- 算法方法 --------*/

//交换父子节点（中所存放的内容）
  protected void swapParentChild(BinTreePosition u, BinTreePosition v) {
    Object temp = u.getElem();
    u.setElem(v.getElem());
    v.setElem(temp);
  }

//上滤算法
  protected void percolateUp(BinTreePosition v) {
    BinTreePosition root = H.getRoot();//记录根节点
    while (v != H.getRoot()) {//不断地
      BinTreePosition p = v.getParent();//取当前节点的父亲
      if (0 >= comp.compare(key(p), key(v)))  break;//除非父亲比孩子小
      swapParentChild(p, v);//否则，交换父子次序
      v = p;//继续考察新的父节点（即原先的孩子）
```

```
    }
  }

//下滤算法
  protected void percolateDown(BinTreePosition v) {
    while (v.hasLChild()) {//直到v成为叶子
      BinTreePosition smallerChild = v.getLChild();//首先假设左孩子的（关键码）更小
      if (v.hasRChild() && 0 < comp.compare(key(v.getLChild()), key(v.getRChild())))
        smallerChild = v.getRChild();//若右孩子存在且更小，则将右孩子作为进一步比较的对象
      if (0 <= comp.compare(key(smallerChild), key(v))) break;//若两个孩子都不比v更小，
则下滤完成
      swapParentChild(v, smallerChild);//否则，将其与更小的孩子交换
      v = smallerChild;//并继续考察这个孩子
    }
  }
}
```

代码五.13 利用堆实现优先队列

利用堆结构表示的优先队列，包括以下要素（如 图五.2 所示）：

- 堆H。即一棵完全二叉树，其中各节点存有条目并根据其关键码满足堆序性。我们可以采用第 §4.6 节介绍的办法，通过向量实现H。为了叙述方便，以下用k(v)来表示节点v（所存放的）条目的关键字。
- 比较器 C。各关键码之间的全序关系，正是由 C 确定的。



图五.2 利用堆结构实现优先队列的原理

在查询优先队列规模时，实际上是查询堆H的规模，因为这棵完全二叉树是由向量实现的，故根据第 3.1.2 节的结论，isEmpty()和getSize()操作都可以在$\mathcal{O}(1)$时间内完成。另外，此时为了获取优先队列中的最小元素，只需取出堆顶，即向量中秩为 1 的元素（注意，这里的习惯与完全二叉树稍有不同，秩 0 单元空闲，从秩为 1 的单元开始存放元素），因此getMin()操作也可以在$\mathcal{O}(1)$时间内完成。

5.8.2 插入与上滤

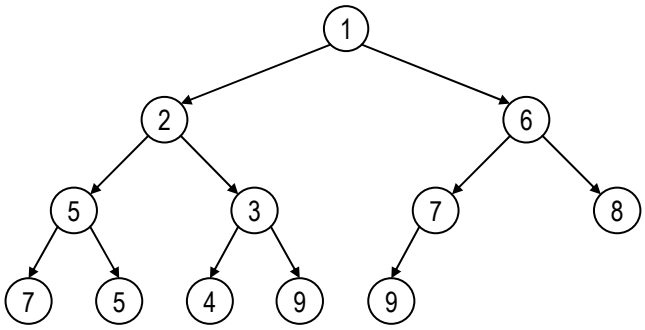
现在我们来讨论如何实现堆 H 的插入操作 insert(key, e)。

首先，我们可以通过完全二叉树的addLast()方法，直接将条目v = (key, e)作为最末尾的节点插入H中。以如 图五.3(a)所示的堆为例，加入末节点后，结果如 图五.3(b)所示。加入该条目后，H的规模扩大了，但依然是一棵完全二叉树——也就是说，堆的结构性并没有破坏。

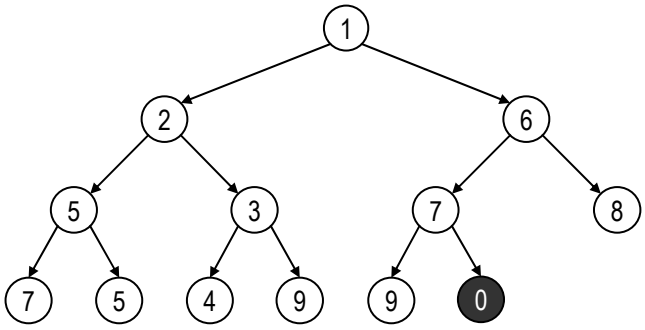
除非 H 原先是空的，否则新节点 v 必然有个父亲。此时，就 v 与其父亲（记作 u）关键码的大小关系而言，有两种可能。若 key(v) ≥ key(u)，则 v 的引入没有破坏 H 的堆序性，节点插入操作顺利完成。然而反之，若 key(v) < k ey(u)，则在此局部的堆序性不再满足。在这种情况下，如何恢复这一局部的堆序性呢？

最简单而直接的办法，就是交换v和u。如 图五.3(c)所示，这样交换之后，v和u之间的堆序性就得到了恢复。但是，随着v向上移动一层，它与新的父节点之间依然可能不满足堆序性。为了解决这一问题，我们可以再次采用上面的办法，将v与新的父节点交换，结果如 图五.3(d)所示。如有必要，不断重复这种交换操作。

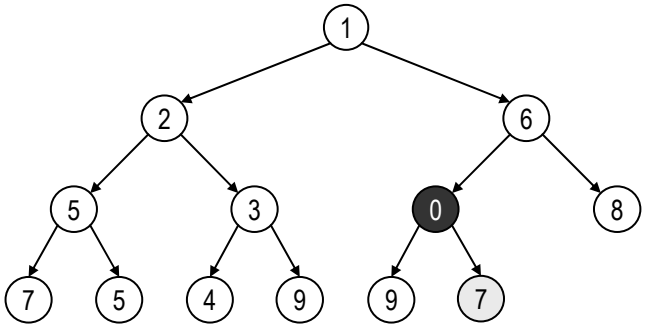
我们注意到，每经过一次交换，新插入的节点v就会向上移动一层。尽管这种情况有可能会持续发生，但v的高度不可能超过堆的高度h，因此在经过至多h交换后，全剧的堆序性最终必将恢复（图五.3(e)）。



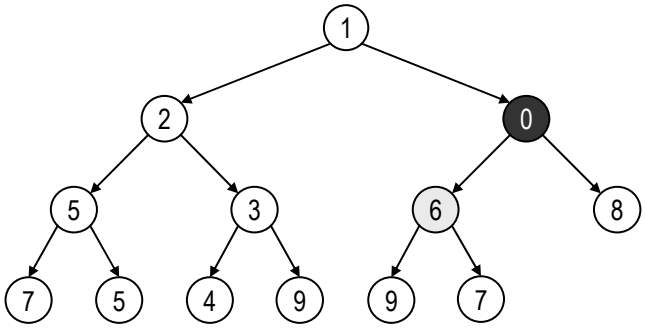
(a) 初始堆



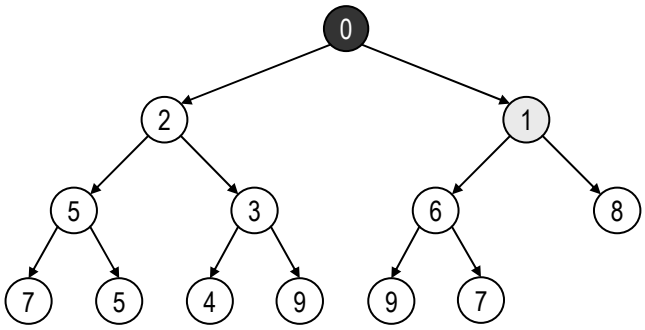
(b) 将新元素 0 作为完全二叉树的末元素接入



(c) 新元素 0 与其父节点 7 交换位置，上升一层



(d) 新元素 0 与其父节点 6 交换位置，上升一层



(e) 新元素 0 与其父节点 1 交换位置，最终完成插入

图五.3 堆节点插入操作的过程

新插入节点通过与父亲交换不断向上移动的这一过程，称作上滤（Percolating up）。若利用向量实现完全二叉树，每次这样的交换只需常数时间。故由 引理四.1 可知：

定理五.2 二叉堆的 insert()操作可以在 $\mathcal{O}(\log n)$的时间内完成，其中 n 为堆的规模。

5.8.3 删除与下滤

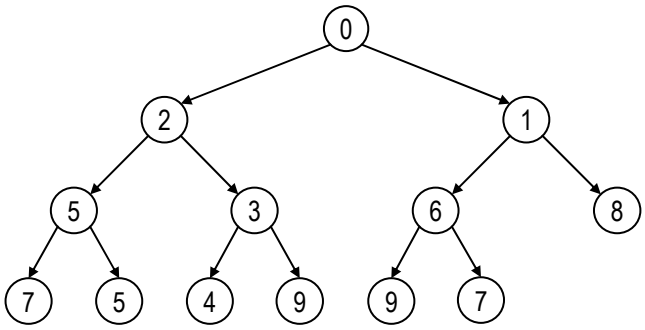
下面再来讨论 delMin()方法的实现。

如 图五.4(a)所示，根据堆的性质，最小元素必然出现于堆顶。然而如 图五.4(b)所示，在将堆顶节点摘除之后，堆的结构将不再完整，不再满足结构性。为了恢复其结构性，最简便的办法就是将最末尾的节点v移至堆顶位置，如 图五.4(c)所示。

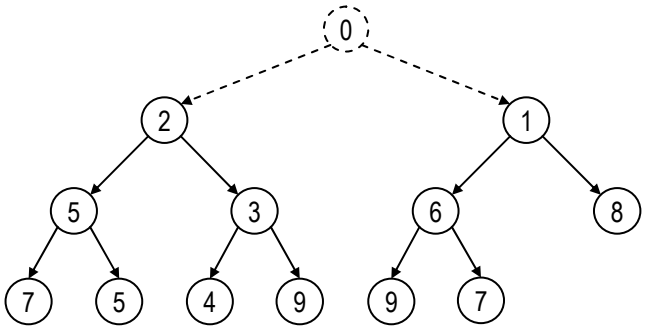
然而，除非此时堆中只剩单个节点，否则移至堆顶的 v 必然拥有后代。而且，一般而言 v 与它的某个（甚至两个）孩子之间将不满足堆序性——即 key(v)更大。这种情况下，又该如何恢复堆序性呢？

上面介绍的上滤操作，可以对称地运用于此。如果v与孩子之间违背堆序性，我们在v的两个孩子中挑选出更小者（记作u，实际上u也是v及其两个孩子中的最小者），并将v与u交换。如 图五.4(d)所示，经过这样的一次交换，v将下降一层；而更重要的是，v及其（原先的）两个孩子之间的堆序性随即得到恢复。

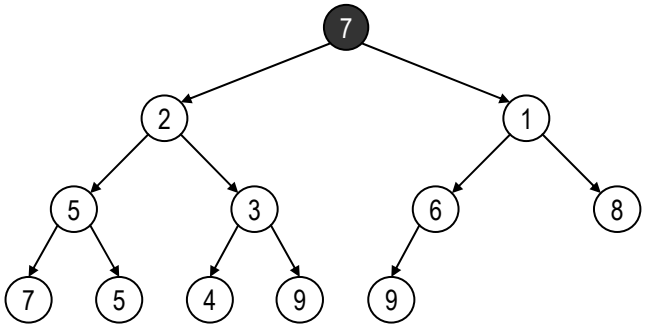
然而问题在于，若下降后的v还有新的孩子，则与此前一样，它们之间仍然可能会地违背堆序性。为此，我们可以反复运用上述交换的办法，直到如 图五.4(e)所示，整个H的堆序性得到恢复。



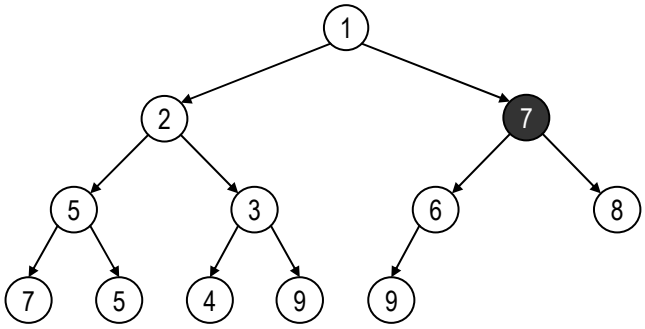
(a) 任一（小顶）堆中，最小元必处于堆顶



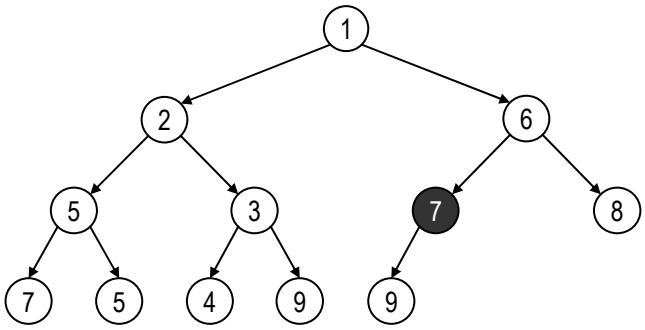
(b) 直接取出堆顶



(c) 将末元素 7 移至堆顶



(d) 元素 7 与其孩子 1 交换位置



(e) 元素 7 与其孩子 6 交换位置，最终完成删除操作

图五.4 堆节点删除操作的过程

在这一过程中，v 的高度逐层下降，因此我们称之为下滤（Percolating down）。同样地，由于 H 的高度为 $\mathcal{O}(\log n)$，故至多经过 $\mathcal{O}(\log n)$次交换后，下滤操作必然会终止。无论是在两个孩子中挑选出更小者，还是将其与 v 交换，都可以在常数时间内完成。综合考虑上述结论可知：

定理五.3 二叉堆的 delMin()操作可以在 $\mathcal{O}(\log n)$的时间内完成，其中 n 为堆的规模。

5.8.4 改变任意节点的关键码

有时，我们可能需要修改堆中某一条目的关键码，然而同样地，这很有可能也会破坏堆序性。如何才能通过调整重新得到一个合法的堆结构呢？这种调整能够在多少时间内完成呢？

请读者参照上面介绍的上滤和下滤算法，自行设计算法并具体实现。

5.8.5 建堆

在很多算法的初始化阶段，我们往往会遇到这样一个共同的问题：给定一组具有特定关键码的记录，将它们组织成一个二叉堆。我们称之为“建堆”（Heapification）问题。

■ 蛮力算法

乍看起来，这似乎并不成其为一个问题——只需从空堆开始，反复调用 insert()方法依次插入各个记录，即可完成这一任务。问题在于，采用这一蛮力策略建堆需要多少时间呢？

由 定理五.2，利用insert()方法插入第i个节点需要$\mathcal{O}(\log i)$的时间，因此依次插入所有n个节点总共需要的时间应该是：

$$\mathcal{O}(\log 1 + \log 2 + ... + \log n) = \mathcal{O}(\log n!) = ^{(1)} \mathcal{O}(n \log n)$$

实际上，如果不在乎花费这样多的时间，那么利用 第八章中的算法，完全可以对所有记录（按照关键码）做一次全排序。然而在这里，花费这么多时间所生成的堆却只能提供对应于一个偏序集的信息。从这个角度来看，我们完全有理由期望能够找到更好的建堆算法。

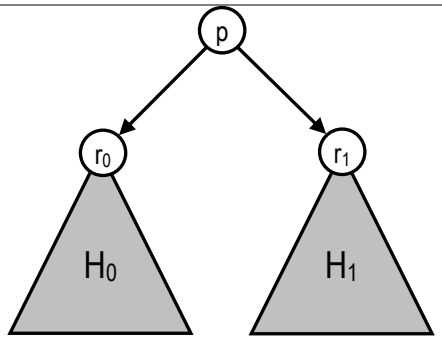
---

(1) 请读者验证 $\mathcal{O}(\log(n!)) = \mathcal{O}(\log(n^n))$

■ Robe rt Floyd 算法

如 图五.5 所示，不难证明以下事实的成立：

观察结论五.1 任意给定分别以 r_0、r_1 为根节点的堆 H_0、H_1 以及节点 p。为了得到对应于 $H_0 \cup H_1 \cup \{p\}$ 的一个堆，只需将 r_0 和 r_1 当作 p 的孩子，然后对 p 进行下滤。



图五.5 堆的合并

因此，我们可以将上述蛮力建堆策略的处理方向与次序颠倒过来。首先，将所有节点存储为一棵完全二叉树，从而满足结构性和完整性。为了恢复堆序性，可以自下而上，对各个内部节点实施下滤操作。根据 观察结论五.1，在所有内部节点都经过下滤之后，二叉树处处都将满足堆序性——亦即成为一个名副其实的堆。

这一策略可以描述为 算法五.1：

```
算法：Heapificate(N[], n)
输入：n个堆节点组成的序列N[]
输出：将输入的节点组建为一个堆
说明：Robert Floyd算法，O(n)时间
{
  创建容量为n的空堆H;
  将N[]中的n个节点直接复制到堆中; //结构性自然得到满足

  自底而上，依次下滤各内部节点; //根据观察结论五.1，此后堆序性也必然满足
  返回H;
}
```

算法五.1 通过自下而上的下滤创建堆

这一算法的具体实现，如 代码五.13 中的构造方法PQueue_Heap(Sequence S)所示。

■ 复杂度分析

那么 算法五.1 的效率如何？是否如我们所期待的，其运行时间低于$\mathcal{O}$(n**log**n)？

首先请注意到，在完全二叉树中对高度为h的任一节点进行下滤，至多需要进行h次节点交换。另一方面，若记堆的高度为d，则高度为h的节点不会超过 2^{d-h}个。因此，若设堆的规模为n，算法五.1消耗于下滤操作的时间为**S**(n)，则有：

$$S(n) \leq \sum_{h=0}^{d}(h \times 2^{d-h}) = 2^{d+1} - d - 2 < 2^{d+1}$$

另外，考虑到完全二叉树的上面 d 层（即深度为 0 到 d-1 的所有节点）构成一棵满树，故有：

$$n \geq 2^{d} - 1$$

于是，便有

$$S(n) < 2(n+1) = \mathcal{O}(n)$$

也就是说，在 算法五.1 中，所有下滤操作总共只需线性的时间。考虑到将n个节点组织为一棵完全二叉树也可以在线性时间内完成，故综合起来可以得出如下结论：

定理五.4 只需 $\mathcal{O}(n)$时间，即可将 n 个条目组织为一个二叉堆结构。

§5.9 堆排序

5.9.1 直接堆排序

正如我们在第 §5.8 节中已经看到的，通过堆结构可以高效率地实现优先队列ADT中的所有方法。现在，让我们针对这一实现形式，回过头来重新审视第 5.3.3 节给出的排序器Sorter_PQueue。

Sorter_PQueue采用的排序算法分为前、后两个阶段。前一阶段的任务，是将待排序的n个元素组织为一个优先队列Q；在后一阶段，不断将优先队列中最小的元素摘出，它们将依次构成一个有序序列。然而按照当时介绍的方法，无论是通过有序列表还是无序列表实现优先队列，都无法兼顾这两个阶段的效率。那么，要是按照第 §5.8 节的介绍，基于堆结构来实现优先队列，又将如何呢？

按照堆结构的思路，Sorter_PQueue方法的前一阶段实质上只不过是一次建堆操作。根据 定理五.4，只需线性时间即可完成这一任务。

在Sorter_PQueue方法的后一阶段，我们反复调用delMin()方法，不断取出Q中当前的最小元素。根据 定理五.3，对delMin()方法的每次调用只需$\mathcal{O}(\log n)$的时间，于是n次调用共需$\mathcal{O}(n\log n)$的时间。

综上所述，可以得出如下结论：

定理五.5 若元素之间的每次比较都可以在常数时间内完成，则基于堆结构实现的 Sorter_PQueue 排序器可以在 $\mathcal{O}(n\log n)$的时间内完成对 n 个元素的排序。

基于堆结构的这一排序算法，亦称作堆排序（Heapsort）。与第 §5.6 节中复杂度为$\mathcal{O}(n^2)$的两个算法相比，堆排序算法的改进十分明显。

第 8.3.2 节将会证明，基于比较操作的排序算法具有Ω(n**log**n)的时间复杂度下界（定理八.2），故就最坏情况的时间复杂度而言，堆排序算法已经达到了最优。由此也可以进一步推断，就insert()和delMin()操作的最坏情况时间复杂度而言，基于堆结构实现优先队列也已经达到了最优[1]。

5.9.2 就地堆排序

在很多场合，待排序的 n 个元素都以一个长度为 n 的数组 S[]的形式给出。这种情况下，我们可以进一步降低堆排序算法的复杂度。当然，我们只能降低时间复杂度的常系数。不过，在一般的应用问题中，这一改进的实际效果还是相当可观的。更重要的一点是，针对这类情况，我们可以大大降低该算法的空间复杂度。

按照第 5.9.1 节介绍的算法，必须将所有元素组织成为一个堆。为此，除输入数组之外，该算法还需要为堆结构消耗***O***(n)的辅助空间。实际上，完全可以将辅助空间的规模降至***O***(1)——也就是说，与输入数组的规模无关。显然，***O***(1)是辅助空间规模的极限，像这样除输入本身外只使用常数辅助空间的算法，称作就地算法（In-place algorithm）。下面，我们就来介绍一个这样的一个堆排序算法，该算法的具体方法及过程如下：

首先，我们将使用一个大小“颠倒”的比较器，其效果相当于将小顶堆转换为大顶堆。

我们将数组 S[]划分为左、右两部分：前缀子数组 S[0..k-1]存放待排序的元素，在算法执行的过程中，它们构成一个大顶堆；后缀子序列 S[k..n-1]（依序）存放当前已经排序的元素。请注意，这里从 0 号单元开始存放元素，故相应地，i 号元素的父节点应为$\lfloor\frac{i-1}{2}\rfloor$号元素，而左、右孩子应分别为 2i+1 和 2i+2 号元素。

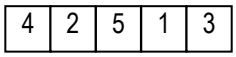
依然采用Sorter_PQueue算法的框架。建堆阶段过程如 图五.6(a)至(f)所示，首先将数组S[]看作一棵完全二叉树，然后采用 算法五.1，自下而上地对各内部节点实施下滤操作，最终得到一个大顶堆。这一过程与 算法五.1 本质上完全一致，只需***O***(n)时间；不同的是，这里只需要***O***(1)的附加空间，以支持下滤过程中的交换操作。

接下来，从规模为n的堆（存放于S[0..n-1]）开始，反复调用delMin()，不断取出当前堆顶元素。因为这里采用了一个大小颠倒的比较器，故实际上每次取出的S[0]都是当前堆中的最大元素。一般地，第k个取出的堆顶元素S[0]必然是所有n个元素中的第k大者，在最终按非降序排列的序列中，它应该存放于S[n-k-1]，k = 0..n-1 （请读者自己证明这一结论）。故此，在第k次迭代中，我们只需将当前的堆顶S[0]与S[n-k-1]交换（注意，这也只需借助一个辅助单元）；然后，通过对S[0]实施下滤操作，就可以在***O***(**log**(n-k-1)) = ***O***(**log**n)的时间内，将子数组S[0..n-k-2]重新调整为一个大顶堆。具体过程如 图五.6(g)至(o)所示。

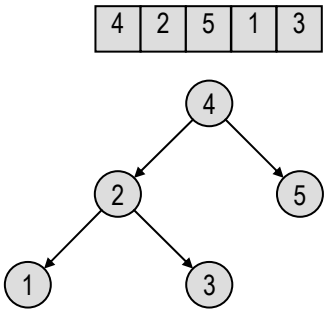
每经过一次这样的迭代，堆的规模都将减一，而后缀子序列则会增长一个单元。经过 n 次迭代之后，大顶堆将完全消失，而后缀子序列则将占据整个序列 S[0..n-1]，至此排序任务遂告完成。

---

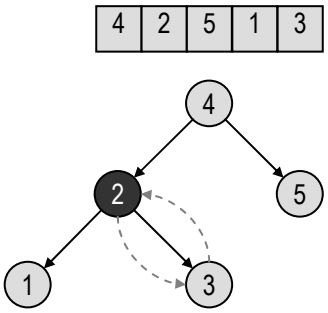
[1] 为严格证明这一推断，需要建立一个从“排序”问题到“堆插入、删除”问题的规约（Reduction）。



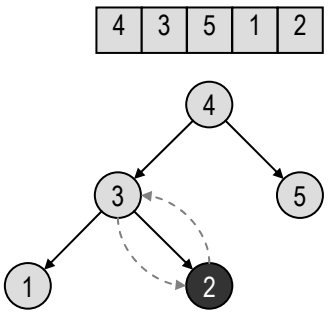
(a) 初始序列 S[0..4]



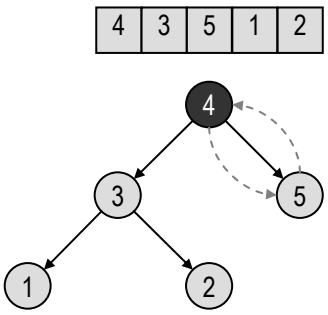
(b) 将 S[0..4]当作一棵完全二叉树



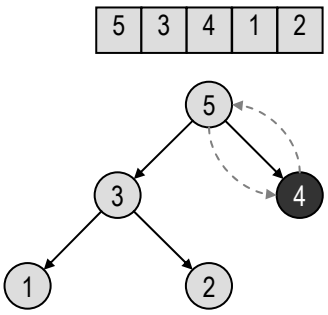
(c) 准备对内部节点 2 实施下滤



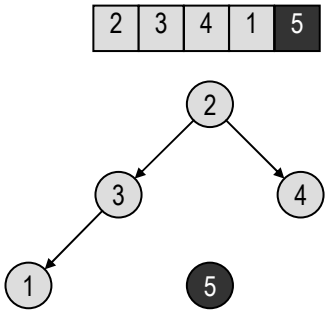
(d) 对内部节点 2 实施下滤之后



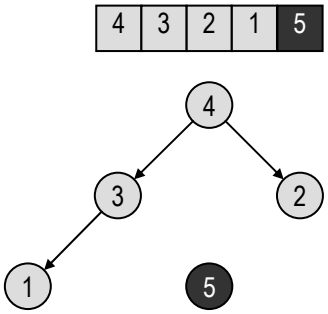
(e) 准备对内部节点 4 实施下滤



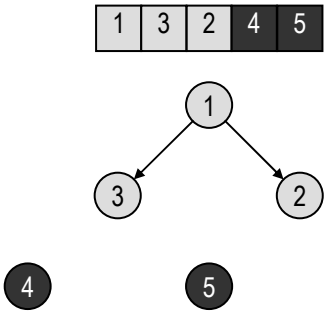
(f) 对内部节点 4 实施下滤之后，构成一个大顶堆



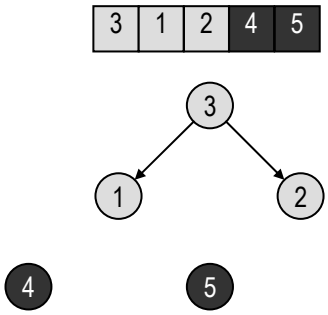
(g) 摘出堆顶元素 5，与末元素 2 交换位置



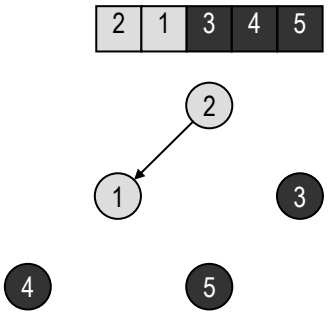
(h) 通过下滤，恢复堆序性



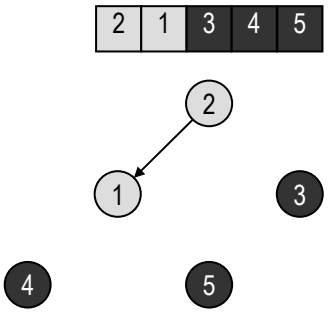
(i) 摘出堆顶元素 4，与末元素 1 交换位置



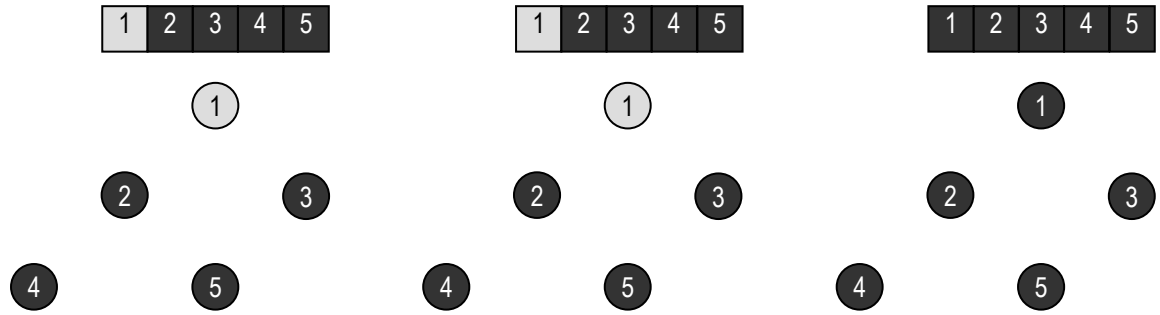
(j) 通过下滤，恢复堆序性



(k) 摘出堆顶元素 3，与末元素 2 交换位置



(l) 无需下滤，已经满足堆序性



(m) 摘出堆顶元素 2，与末元素 1 交换位置　(n) 单节点二叉树必是堆，无需下滤　(o) 排序完成

图五.6　就地堆排序：(a~f)构建大顶堆，(g~o)依次取出堆顶

请读者根据上面的介绍，自行实现就地堆排序算法。

§ 5.10 Huffman 树

本节将通过 Huffman 编码树的构造问题，介绍优先队列结构的具体应用。

5.10.1 二叉编码树

■ 二进制编码

通讯系统可以帮助人们将一段信息从发送端传送给接收端。最常见的信息形式是字符串，即由来自某一有限字符集 **Σ** 的若干个字符组成的一个序列 $M = (x_1, \ldots, x_n)$，$x_i \in \Sigma$，$1 \le i \le n$。在将 M 加载至信道上并发送之前，首先需要对 M 进行编码（Encoding）。通常采用的都是二进制编码——对于 **Σ** 中的每个字符 c，分别指定一个二进制串 e(c)，这可以描述为如下单射函数：

$$e() : \Sigma \mapsto \{0, 1\}^*$$

接收端在收到该该二进制串后，可以利用对应的逆函数进行解码（Decoding）：

$$e^{-1}() : \{0, 1\}^* \mapsto \Sigma$$

这样就可以得到发送的信息内容。

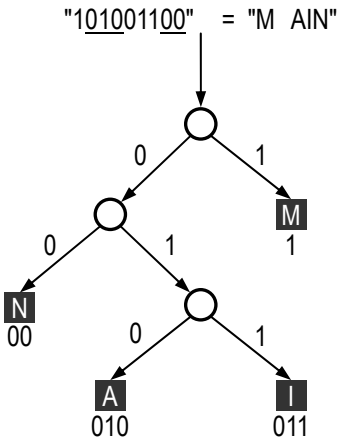
以英文字母集 **Σ** = {A, B, C, …, Z}为例，若需要传送字符串“MAIN”，一种可行的编码方式就是：

e('N') = "00"

e('A') = "010"

e('I') = "011"

e('M') = "1"



图五.7 英文字符串的二进制编码

如 图五.7 所示，我们可以将这一编码方式表示为一棵二叉树。其中，每一“父亲-左孩子”关系对应于一个二进制位'0'，每一“父亲-右孩子”关系对应于一个二进制位'1'。于是，若令每个字符分别对应于一匹叶子，则从根节点通往每匹叶子的路径，就对应于相应字符的二进制编码。因此，这样的一棵树也称作二叉编码树。

■ 二进制解码

反过来，根据这棵编码树也可以便捷地完成解码工作。还是如 图五.7 所示，从头开始扫描接收到的二进制信息流，从根节点开始，根据各比特位不断进入下一层节点；到达叶子节点后，输出其对应的字符，然后重新回到根节点，并继续扫描二进制流。实际上，这一过程可以在接收过程中实时进行，而不必等到所有的比特位都到达之后才开始解码。

依然以 图五.7 为例，假设接收到的二进制信息流为"101001100"，我们自左向右扫描其中的各比特位。在读出最前端的'1'后，我们相应地从根节点移动至叶子节点'M'，于是在输出这一字符之后，回到根节点并继续扫描。接下来，在依次读出'0'、'1'和'0'之后，到达叶子节点'A'，于是输出这一字符，并回到根节点继续扫描。随后，在依次读出'0'、'1'和'1'之后，到达叶子节点'I'，于是输出这一字符，并回到根节点继续扫描。最后，在依次读出'0'和'0'之后，到达叶子节点'N'，输出这一字符后扫描终止。这样，我们就得到了原始的英文字符串"MAIN"，解码工作完成。

5.10.2 最优编码树

在实际的通讯系统中，信道的使用效率是个很重要的问题，这在很大程度上取决于编码算法本身的效率。很自然地，我们当然希望能够用尽可能少的比特位来表示字符串。那么，如何做到这一点呢？在什么情况下能够做到这一点呢？

■ 平均编码长度

若将字符 c 在二叉编码树中对应的叶子的深度记为 depth(c)，则不难发现有：

观察结论五.2 每个字符 $c \in \Sigma$ 的编码长度为|e(c)| = depth(c)。

仍以 图五.7 为例，字符'M'、'N'、'A'和'I'的编码长度分别为 1、2、3 和 3，与它们的深度相一致。

定义五.1 对于任一字符集 Σ 的任一编码方式 e()，Σ 中各字符的编码长度总和 $\sum_{c\in\Sigma}|e(c)|$称作 e()（或其对应的二叉编码树）的编码总长度；单个字符的平均编码长度为 $\sum_{c\in\Sigma}|e(c)|/|\Sigma|$。

不难理解，平均编码长度是反映编码效率的一项重要指标，我们希望这一指标尽可能地小。

■ 最优编码树

定义五.2 对于任一字符集 Σ，若在所有的编码方式中，某一编码方式 e()使得平均编码长度最短，则称 e()为 Σ 的一种最优编码，与之对应的编码树称作 Σ 的一棵最优编码树。

我们注意到，对于同一字符集 Σ，所有深度不超过$|\Sigma|$的编码树只有有限棵，因此其中的总编码长度最小者必然存在——尽管不见得唯一。

■ 最优编码树的性质

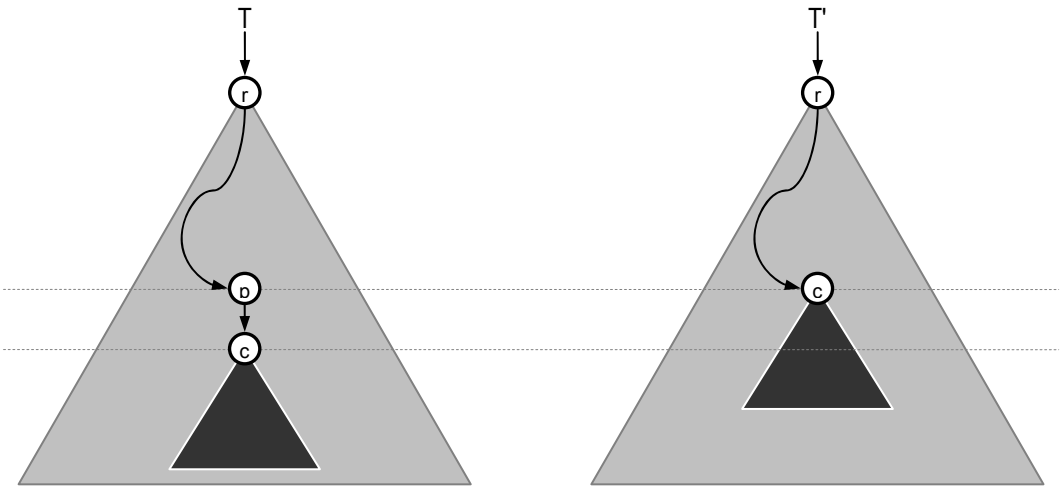
现在的问题是：对于任一给定的字符集 Σ，如何找到这样的一棵最优编码树呢？为此，我们首先需要更加深入地了解最优编码树的性质。

观察结论五.3 在最优二叉编码树中

① 每个内部节点的度数均为 2；

② 各叶子之间的深度差不超过 1。

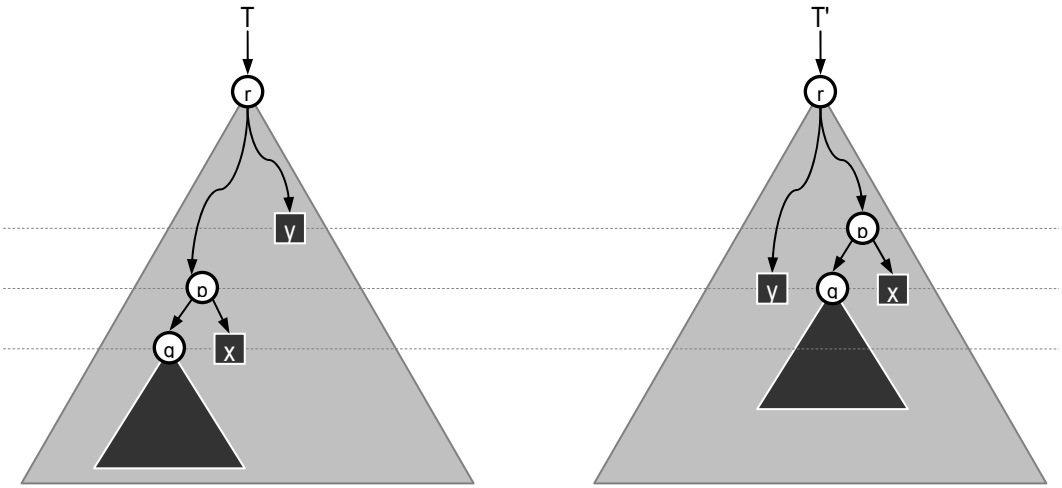
〖证明〗



图五.8 最优编码树的双子性

首先证明①。

如图五.8所示，假设在某棵最优二叉编码树T中存在一个度数为 1 的内部节点p，不妨设p唯一的孩子为节点c。于是，只要将节点p删除，并代之以子树c，则可以得到原字符集的另一棵编码树T'。不难看出，除了子树c中每匹叶子的编码长度均减少 1 之外，其余叶子的编码长度不变，因此与T相比，T'的平均编码长度必然更短——这与T的最优性矛盾。



图五.9 最优编码树的平衡性

再来证明②。

如图五.9所示，假设某棵最优二叉编码树T中存在深度相差超过1的两匹叶子x和y。不妨设x更深，并令p为x的父亲。于是根据①，作为内部节点的p必然还有另一个孩子q。请注意，子树q中至少含有一匹叶子。

现在，将叶子y与子树p交换，得到一棵新的树T'。易见，T'依然是原字符集的一棵二叉编码树。更重要的是，除了x、y以及子树q中的叶子外，其余叶子的深度均保持不变。经过这一交换操作之后，x的深度减少1，y的深度增加1，而q中各叶子的深度都将减少1。因此，与T相比，T'的平均编码长度必然更短——这与T的最优性矛盾。 ❑

推论五.1 基于由2|Σ|-1个节点构成的完全二叉树T，将Σ中的字符任意分配给T的|Σ|匹叶子，即可得到Σ的一棵最优编码树。

这一推论也直接给出了一个构造最优编码树的算法。

5.10.3 Huffman 编码与 Huffman 编码树

■ 字符出现概率

上一节所介绍的最优编码树，在实际应用中的利用价值并不大。不难看出，只有当Σ中各个字符在信息串中出现的概率相等时，其最优性才有意义，遗憾的是，这一条件很难满足。在实际应用中，Σ中各字符的出现频率不仅很少相等或相近，而且往往相差悬殊。以英文信息串为例，'e'、't'等字符的出现频率通常都是'z'、'j'等字符的数百倍。在这种情况下，就应该从另一角度衡量每个字符的编码长度。

表五.4 根据一篇典型的英文文章，对各字母出现频率的统计

字符	A	B	C	D	E	F	G	H	I	J	K	L	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
频率	623	99	239	290	906	224	136	394	600	5	56	306	586	622	148	10	465	491	732	214	76	164	16	139	13

■ 平均带权编码长度

若假设字符 c 出现的概率为 $p(c) \geq 0$，$\sum_{c\in\Sigma} p(c) = 1$，将其在二叉编码树中对应的叶子的深度记作 depth(c)，则

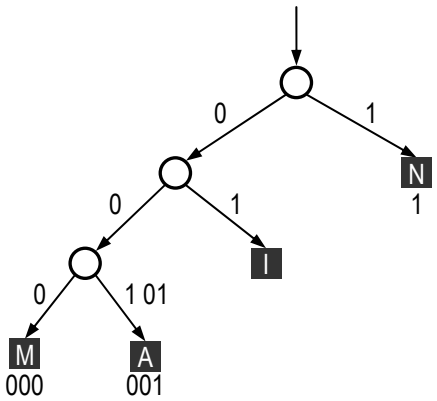
定义五.3 每个字符 $c\in\Sigma$ 的带权编码长度为 $|e(c)| = depth(c) \times p(c)$。

定义五.4 对于任一字符集 Σ 的任一编码方式 e()，Σ 中各字符的平均带权编码长度总和 $\sum_{c\in\Sigma}|e(c)|$ 称作 e()（或其对应的二叉编码树）的平均带权编码长度。

不难理解，在考虑字符出现频率时，如上定义的平均带权编码长度就是反映编码效率的一项重要指标，我们同样也希望这一指标尽可能地小。

"000001000001101" = "MAMANI", $\sum_{c\in\Sigma}|e(c)| = 15/6 = 2.5$

"000001000000010000 1001" = "MAMMAMIA", $\sum_{c\in\Sigma}|e(c)| = 23/8 = 2.875$



图五.10 考虑字符的出现频率时，可以用平均编码长度来评价编码的效率

以 图五.10 为例，若各字符出现的概率分别为 p('M') = 2/6、p('A') = 2/6、p('I') = 1/6 和 p('N') = 1/6（比如信息串"MAMANI"），则各字符的带权编码长度分别为：

$|e('M')| = 3\times(2/6) = 1$

$|e('A')| = 3\times(2/6) = 1$

$|e('I')| = 2\times(1/6) = 1/3$

$|e('N')| = 1\times(1/6) = 1/6$

相应地，这一编码方式对应的平均带权编码长度就是

$|e('M')| + |e('A')| + |e('I')| + |e('N')| = 2.5$

若各字符出现的概率分别为 p('M') = 4/8、p('A') = 3/8、p('I') = 1/8 和 p('N') = 0/8（比如信息串"MAMMAMIA"），则各字符的带权编码长度分别为：

$|e('M')| = 3\times(4/8) = 3/2$

$|e('A')| = 3\times(3/8) = 9/8$

|e('I')| = 2×(1/8) = 1/4

|e('N')| = 1×(0/8) = 0

相应地，这一编码方式对应的平均带权编码长度就是

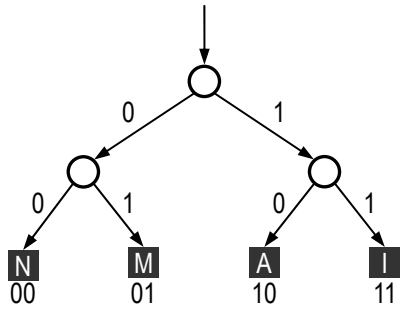
|e('M')| + |e('A')| + |e('I')| + |e('N')| = 2.875

■ 完全二叉编码树 ≠ 平均带权编码最短

平均带权编码长度能否更短呢？

"011001100011" = "MAMANI", $\sum_{c\in\Sigma}|e(c)|$ = 12/6 = 2

"0110010110011110" = "MAMMAMIA", $\sum_{c\in\Sigma}|e(c)|$ = 16/8 = 2

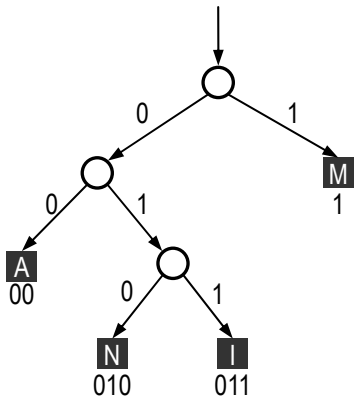


图五.11 考虑字符的出现频率时，完全二叉树未必是最优二叉编码树

我们首先想到的是前一节中提到的完全二叉编码树。如 图五.11 所示，由于此时每个字符的编码长度都是 2，故无论各字符的出现概率具体分布如何，其对应的平均带权编码长度都将为 2。

"100100010011" = "MAMANI", $\sum_{c\in\Sigma}|e(c)|$ = 12/6 = 2

"100110010110 0" = "MAMMAMIA", $\sum_{c\in\Sigma}|e(c)|$ = 13/8 = 1.625



图五.12 考虑字符的出现频率时，最优二叉编码树往往不是完全二叉树

然而与不考虑字符出现概率时不同的是，某些非完全的二叉编码树的平均带权编码长度，有可能更短。比如根据 图五.12 中的二叉编码树，对字符串"MAMANI"的平均带权编码长度为 2，与前者相当；但对字符串"MAMMAMIA"来说，平均带权编码长度却只有 1.625。

■ 最优带权编码树

定义五.5 对于任一字符集 Σ，在字符出现频率分布为 p()时，若某一编码方式 e()使得平均带权编码长度达到最短，则称 e()为（按照 p()分布的）Σ 的一种最优带权编码，其对应的编码树称作（按照 p()分布的）Σ 的一棵最优带权编码树。

当然，与不考虑字符出现概率时同理，最优带权编码树也必然存在，而且通常也不唯一。

■ 最优带权编码树的双子性

为了得出最优带权编码树的构造算法，我们还是从分析其性质入手。

观察结论五.4 在最优带权编码树中，内部节点的度数均为 2。

也就是说，最优带权编码树同样具有双子性。请读者自行证明这一结论。

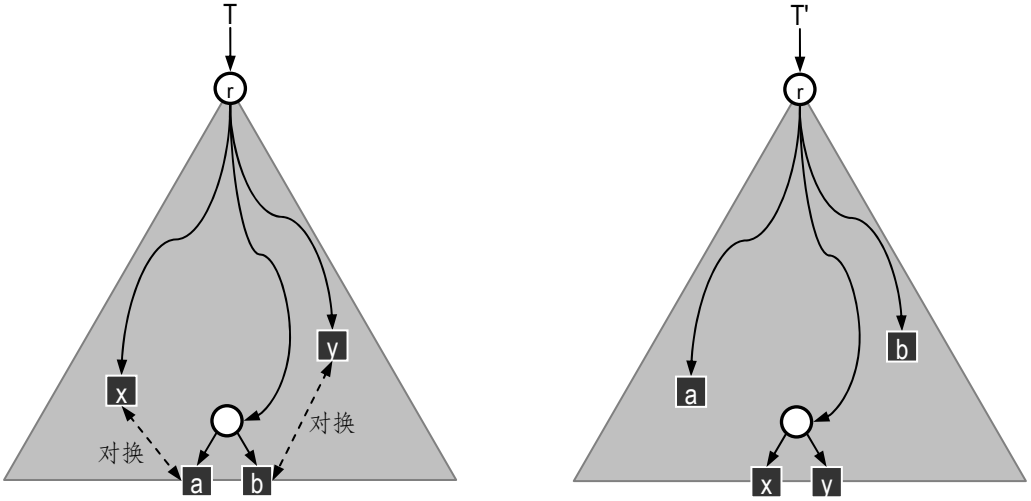
■ Huf fman 编码树的层次性

观察结论五.5 对于字符出现概率为 p()的任一字符集 Σ，若字符 x 和 y 在所有字符中的出现概率最低，则必然存在某棵最优带权编码树，使得 x 和 y 在其中同处于最底层，而且互为兄弟。

〖证明〗

任取一棵最优带权编码树 T。

根据观察结论五.4，在T的最低层节点中，必然可以找到一对兄弟a和b。现在，如图五.13 所示，我们交换节点a和x（如果它们不是同一节点的话），并且交换节点b和y（如果它们不是同一节点的话），从而得到同一字符集的另一棵编码树T'。



图五.13 最优编码树的层次性

显然，经过这样的交换，在 T'中 x 和 y 将成为最低层的一对兄弟节点。

另外，根据字符 x 和 y 的权重最小性，经过这样的交换，T'对应的平均带权编码长度绝不会增加。于是根据 T 的最优性，T'必然也是一棵最优编码树。 ❑

某些最优带权编码树所具有的这一特性，亦称作层次性。

定义五.6 满足层次性的最优带权编码树，称作 Huffman 编码树。

请注意，即使是对于字符出现概率确定的同一字符集，Huffman 编码树仍然有可能不唯一。

5.10.4 Huffman 编码树的构造算法

设在字符的某一出现概率分布 p()下，字符集 **Σ** 中出现概率最低的两个字符为 x 和 y。现考察另一字符集 **Σ'** = (**Σ**\{x, y}) ∪ {z}，并将新增字符 z 的出现概率设为 p(z) = p(x) + p(y)，其余字符的出现概率保持不变。若取 T'为 **Σ'**对应的一棵 Huffman 编码树，则根据 Huffman 编码树的层次性，不难得出如下推论：

推论五.2 将 T'中与字符 z 对应的叶子替换为一个内部节点，并在其下设置分别对应于 x 和 y 的两匹叶子，则所得到就是 Σ 的一棵 Huffman 编码树。

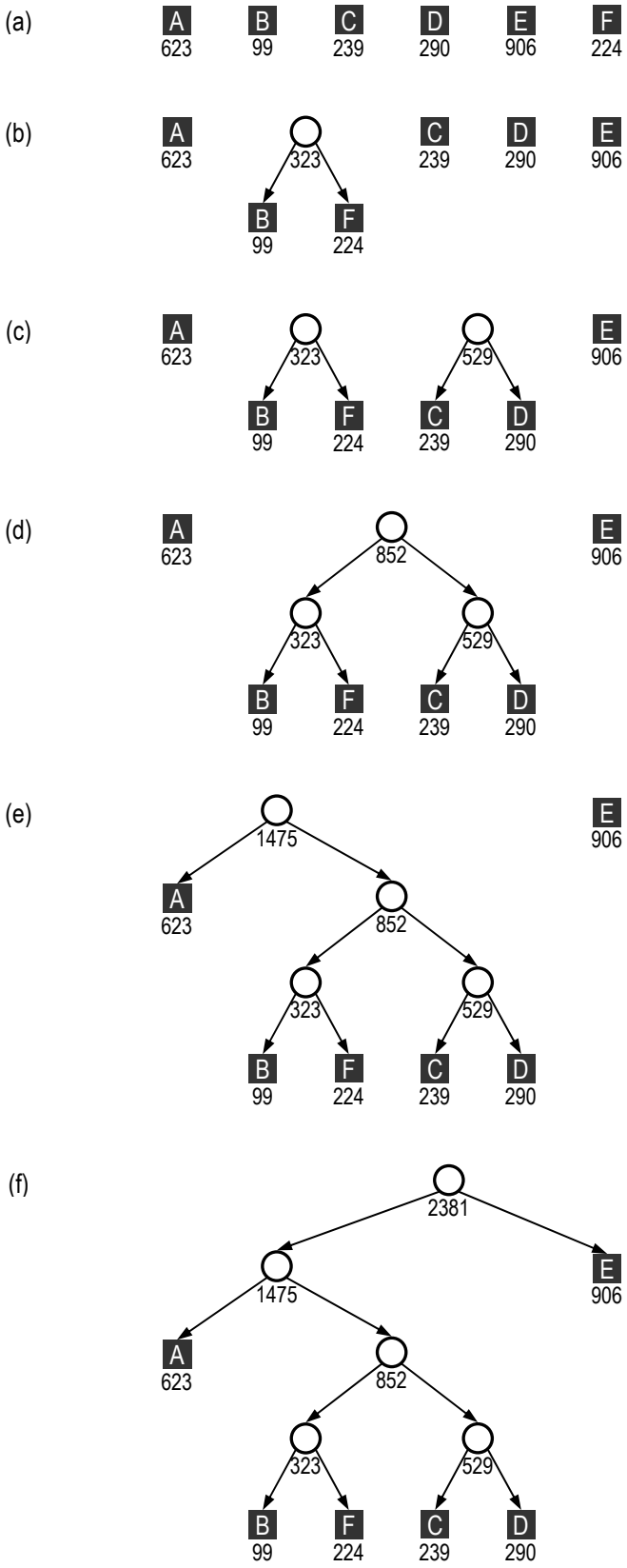
这一推论的证明，留给读者自己完成。

因此，对于字符出现概率满足一定分布的任一字符集 **Σ**，我们都可以按照如下算法来构造其对应的 Huffman 编码树：首先，对应于 **Σ** 中的每一字符，分别建立一棵由单个节点组成树，其权重就是该字符出现的频率，这|**Σ**|棵树组成一个森林 F。接下来，从 F 中选出权重最小的两棵树，创建一个新节点，并分别以这两棵树作为其左、右子树，从而将它们合成为一棵更高的树，其权重等于其左、右子树权重之和。这一选取、合并的过程将反复进行，直到最后 F 中只剩下一棵树——它就是我们所需要的 Huffman 编码树。

例如，针对如 表五.5 所示的字符集**Σ**，其Huffman编码树的构造过程如 图五.14 所示。

表五.5 由6个字符构成的字符集**Σ**，以及个字符的出现频率

字符	A B	C	D	E	F
出现频率	623 99 239		290	906	224



图五.14 Huffma n树构造算法实例

关于 Huffman 树的构造算法，还需要考虑一些退化情况。比如，有些字符的出现频率可能相等，或者虽然最初的字符权重互异，但经过若干次合并之后，森林 F 也可能会出现权重相等的子树。于是，在挑选权重最小的两棵子树时，将有可能出现歧义。幸运的是，这一问题并不难解决，我们把具体的方法设计与分析留给读者。

5.10.5 基于优先队列的 Huffman 树构造算法

一般地，设字符总数$|\Sigma| = n$。在上述算法中，初始化创建森林 F 只需$\mathcal{O}(n)$的时间。在接下来的每一次迭代中，如果采用普通的数据结构，为挑选出权重最小的两棵树需要花费$\mathcal{O}(n)$的时间，而将它们合二为一只需$\mathcal{O}(1)$时间。请注意，每经过一次迭代，森林 F 的规模就会减 1，因此经过 n-1 次迭代之后，F 的规模才能从 n 减至 1。总而言之，该算法需要运行$\mathcal{O}(n^2)$时间。

实际上，利用本章介绍的优先队列结构，可以更加高效地构造 Huffman 树。

具体方法是，始终将森林中的所有树（根）组织为一个优先队列，比如基于二叉堆实现的优先队列。这样，只要连续地调用 delMin()方法两次，就可以找出当前权重最小的两棵树。在将这两棵树合并为一棵新树之后，可以调用 insert()方法将其重新插入优先队列。这一过程将反复进行，每迭代一次，森林的规模就会减小 1。因此，经过 n-1 次迭代，森林中将只包含一棵树，即 Huffman 编码树。

就算法过程而言，上述方法没有多少新意。然而就算法的效率而言，则上述方法将有实质的改进。根据 定理五.3 和 定理五.2，两次delMin()操作加上一次insert()操作，可以在$\mathcal{O}(3\log n)$的时间内完成。记入树合并的时间，每次迭代只需$\mathcal{O}(1+3\log n)$时间。因此，整个构造过程总共所需的运行时间为：

$$(n-1) \times \mathcal{O}(1+3\log n) = \mathcal{O}(n\log n)$$

第六章

映射与词典

第五章介绍的优先队列是一种饶有趣味的数据结构，它不仅将一组元素组织起来，而且按照关键码的次序在各元素之间定义了一种优先级，从而能够支持快速的访问（getMin()）与更新（insert()和delMin()）操作。实际上，借助关键码直接查找数据元素并对其进行操作的这一形式，已经为越来越多的数据结构所采用，也成为现代数据结构的一个重要特征。本章将要讨论的映射（Map）及词典（Dictionary）结构，就是其中最典型的例子，它们对优先队列中利用关键码的思想做了进一步发挥和推广——不再只是可以读取或修改最小元素，而是能够对任意给定的关键码进行查找，并修改相应的元素。

与优先队列一样，映射和词典中存放的元素也是一组由关键码和数据合成的条目。二者之间的差别仅仅在于，映射要求不同条目的关键码互异，而词典则允许多个条目拥有相同的关键码。与优先队列不同的是，词典并不要求关键码之间能够定义某种全序关系。特别地，如果关键码之间的确具有某种全序关系，则对应的词典结构也称作有序词典（Ordered dictio nary）。正如我们将在第 §6.4 节看到的，可以利用向量来实现有序词典，并且可以为其ADT增添若干新的操作方法。

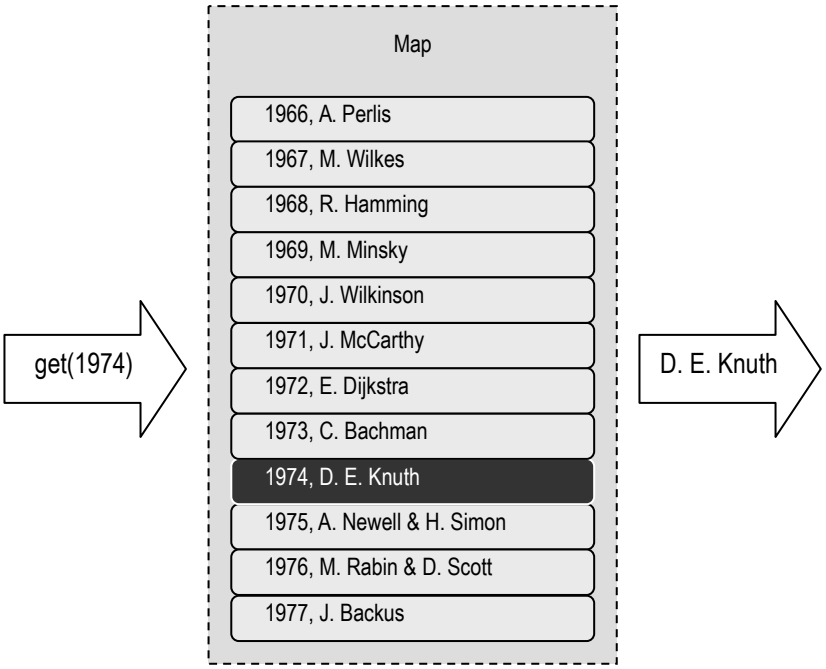
这里谈及的词典，与我们日常使用的词典有着密切联系：二者都可以看作是存放一组数据项（条目）的容器，而且都能够通过关键码查找到对应的数据项。不过，二者的区别也不小。作为数据结构的词典是动态的，我们可以在使用过程中不断加入、删除或修改条目；而通常的印刷式词典可以认为是静态的，只有在做修订后或重印时其内容才可能会有所变化。

本章还将介绍映射和词典的几种实现形式。比如，虽然利用无序列表即可简便地实现词典结构，但这一实现在实际应用中的效率很低。又如，利用散列表之类简单结构也可以实现映射结构，而且其性能在实际应用中足以令人满意。

实际上，利用平衡二分查找树也可以实现高效率的词典结构，这一问题将在 第七章中详细讨论。

§6.1 映射

映射（Map）也是一种存放一组条目的容器。与优先队列一样，映射中的条目也是形如(key, value)的组合对象，其中 key 为关键码对象，value 为具体的数据对象。需要特别指出的是，在映射中，各条目的关键码不允许重复冗余。比如，若准备将某个学校所有学生的记录组织为一个映射结构，则不能以年龄或班级作为关键码，因为不同记录的这些信息都有可能重复；反过来，通常学号都是学生的唯一标识，故可以将学号作为关键码。



图六.1 由前12届图灵奖得主构成的一个映射结构：其中以获奖年份作为唯一的关键码，并支持通过关键码的查询

既然映射中的元素由其关键码唯一标识，而且映射的作用就是通过关键码直接找到对应的元素，故不妨就把关键码理解为指向对应元素的“地址引用”。

6.1.1 映射的 ADT 描述

作为一个容器，映射 M 首先需要支持以下一般性的操作方法：

表六.1 映射ADT支持的操作

操作方法	功能描述
getSize()：	报告映射的规模，即其中元素的数目 输入：无 输出：非负整数
isEmpty()：	判断映射是否为空 输入：无 输出：布尔标志

就其抽象数据类型的特点而言，映射 M 还支持以下独有的操作方法：

表六.2 映射ADT特有的操作

操作方法	功能描述
get(key)：	若 M 中存在以 key 为关键码的条目，则返回该条目的数据对象 否则，返回 null 输入：一个关键码对象 输出：数据对象

操作方法	功能描述
put(key, value):	若映射中不存在以 key 为关键码的条目，则将条目(key, value)加入到 M 中并返回 null 否则，将已有条目的数据对象替换为 value，并返回原先的数据对象 输入：一个关键码对象和一个数据对象 输出：数据对象
remove(key):	若映射中存在以 key 为关键码的条目，则删除之并返回其数据对象 否则，返回 null 输入：一个关键码对象 输出：数据对象
entries():	返回映射中所有关键码对象的一个迭代器 输入：无 输出：条目对象的迭代器

以上面关于图灵奖得主的映射结构为例，从这样一个空的映射结构开始，依次执行若干操作，所得到的结果分别如下。

表六.3 映射结构操作实例

操作	映射结构	输出
isEmpty()	∅	true
put(1970, "J. Wilkinson")	{(1970, "J. Wilkinson")}	N/A
put(1968, "R. Hamming")	{(1970, "J. Wilkinson"), (1968, "R. Hamming")}	N/A
put(1977, "J. Backus")	{(1970, "J. Wilkinson"), (1968, "R. Hamming"), (1977, "J. Backus")}	N/A
get(1968)	{(1970, "J. Wilkinson"), (1968, "R. Hamming"), (1977, "J. Backus")}	"R. Hamming"
get(1977)	{(1970, "J. Wilkinson"), (1968, "R. Hamming"), (1977, "J. Backus")}	"J. Backus"
get(1974)	{(1970, "J. Wilkinson"), (1968, "R. Hamming"), (1977, "J. Backus")}	null
remove(1970)	{(1968, "R. Hamming"), (1977, "J. Backus")}	"J. Wilkinson"
isEmpty()	{(1968, "R. Hamming"), (1977, "J. Backus")}	false

需要指出的是，这里并未采用意外错的形式来处理退化情况——在执行 get()、put()或 remove()方法时，映射 M 中不存在具有指定关键码的条目——而是直接返回 null。之所以采取后一办法，是考虑到在实际应用中执行映射的这些方法时，出现退化情况的概率很高，若采用报意外错的方式，为了扔出与处理意外错需要耗费大量的时间，整个结构的效率将会大打折扣。

不过，直接返回 null 的做法也存在不足，其中最大的问题在于歧义性。比如，按照这一约定，我们将无法以 key = null 为条件进行查找。否则，一旦返回值为 null，我们将无所适从——在这种情况下，究竟是映射结构中不存在这样的条目，还是的确存在以 null 为关键码的条目？

6.1.2 映射的 Java 接口

```
/*
 * 映射结构接口
 */
```

```
package dsa;


public interface Map {
//查询映射结构当前的规模
  public int getSize();

//判断映射结构是否为空
  public boolean isEmpty();

//若映射中存在以key为关键码的条目，则返回该条目的数据对象；否则，返回null
  public Object get(Object key);

//若映射中不存在以key为关键码的条目，则插入条目(key, value)并返回null
//否则，将已有条目的数据对象替换为value，并返回原先的数据对象
  public Object put(Object key, Object value);

//若映射中存在以key为关键码的条目，则删除之并返回其数据对象；否则，返回null
  public Object remove(Object key);

//返回映射中所有条目的一个迭代器
  public Iterator entries();
}
```

代码六.1 映射结构的Java接口

6.1.3 判等器

由其 ADT 描述可知，映射结构必须能够比较任意一对关键码是否相等，每个映射结构在被创建的时候，都需要指定某一具体标准，以便进行关键码的比较。因此，为了实现映射结构，首先必须实现这样的一个判等器（Equality tester）：

```
EqualityTester  T;
```

该判等器提供一个判等方法：

$$\text{T.isEqualTo(key1, key2)} = \begin{cases} \text{true} & \text{若关键码key1与key2相等} \\ \text{false} & \text{否则} \end{cases}$$

当然，若关键码之间的确定义有某种全序关系，则可以将第 5.2.2 节所介绍的比较器Comparator直接当作判等器：对于任何一对关键码key1 和key2，总有

```
isEqualTo(key1, key2)  =  (0 == compare(key1, key2)) ? true : false;
```

然而就一般的映射结构而言，其中的关键码之间未必具有某种全序关系，在这种情况下，我们只能实现一个更加通用的判等器。实际上，Java中所有对象本身都已经提供了一个内建的判等方法equals()，尽管其通用性还达不到判等器的要求，但也不失为一种直接而简便的办法。本书将采用一种折衷的形式：定义一个标准的EqualityTester接口（代码六.2），再利用Java本身提供的equals()方实现一个通用的判等器EqualityTesterDefault（代码六.3）。

```
/*
 * 判等器接口
 */

package dsa;

public interface EqualityTester {
  public boolean isEqualTo(Object a, Object b);//若a与b相等，则返回true；否则，返回false
}
```

代码六.2 判等器EqualityTester接口

```
/*
 * 默认判等器
 */

package dsa;

public class EqualityTesterDefault implements EqualityTester {
  public EqualityTesterDefault() {}
  public boolean isEqualTo(Object a, Object b)
  { return (a.equals(b)); }//使用Java提供的判等器
}
```

代码六.3 默认判等器EqualityTesterDefault的实现

本书提倡使用通用的判等器。尽管 代码六.3 中的默认判等器也是通过标准的equals()方法实现的，但重要的是，利用这种模式，程序员完全可以编写出独立的通用判等器，而无需触及对象内部的结构。

6.1.4 java.util 包中的映射类

Java在java.util包中已定义了一个名为java.util.Map的映射接口，而且也约定禁止关键码的重复。与第 6.1.1 节定义的映射接口相比，java.util.Map接口并未直接提供迭代器方法，而是通过两个名为keys()和values()的方法，间接地提供关于关键码或数据对象的迭代器。

二者的另一差别在于，java.util.Map接口不使用外部判等器，而是直接利用equals()方法实现判等器。正如刚才已经指出的，这种办法的通用性不好：一旦默认的equals()方法对java.util中存放的对象不适用，我们只好去修改关键码类本身——这有悖于面向对象编程的封装原则。此外，这种办法的灵活性也欠佳：即使是同一类对象，在不同场合中判等的具体标准也可能不尽相同，更不是一

成不变的。以第 5.2.3 节中的Point2D对象为例：有的时候，x坐标相同的点被认为相等；有的时候，却是根据y坐标来判定一对点是否相等。

6.1.5 基于列表实现映射类

实现映射结构的最简单办法，就是直接将映射 M 中的条目组织成双向链表形式的一个列表 L。这样，getSize()和 isEmpty()方法可以直接套用 List 接口中对应的方法。而在 get(key)、put(key, value)和 remove(key)方法中为了确定操作条目的位置，可以将 S 中的元素逐一与给定的 key 做对比。

具体的实现如 代码六.4 所示：

```
/*
 * 基于列表实现映射结构
 */

package dsa;

public class Map_DLNode implements Map {
  private List L;//存放条目的列表
  private EqualityTester T;//判等器

//构造方法
  public Map_DLNode()
  { this(new EqualityTesterDefault()); }

//默认构造方法
  public Map_DLNode(EqualityTester t)
  { L = new List_DLNode();  T = t; }

/***************************** ADT方法 *****************************/
//查询映射结构当前的规模
  public int getSize()
  { return L.getSize(); }

//判断映射结构是否为空
  public boolean isEmpty()
  { return L.isEmpty(); }

//若M中存在以key为关键码的条目，则返回该条目的数据对象；否则，返回null
  public Object get(Object key) {
    Iterator P = L.positions();
    while (P.hasNext()) {
      Position pos = (Position)P.getNext();
      Entry entry = (EntryDefault) pos.getElem();
      if (T.isEqualTo(entry.getKey(), key)) return entry.getValue();
    }
    return null;
  }

//若M中不存在以key为关键码的条目，则将条目(key, value)加入到M中并返回null
```

```
//否则，将已有条目的数据对象替换为value，并返回原先的数据对象
  public Object put(Object key, Object value) {
    Iterator P = L.positions();

    while (P.hasNext()) {//逐一对比

      Position pos = (Position)P.getNext();//各个位置
      Entry entry = (EntryDefault) pos.getElem();//处的条目
      if (T.isEqualTo(entry.getKey(), key)) {//若发现key已出现在某个条目中，则
        Object oldValue = entry.getValue();//先保留该条目原先的数据对象
        L.replace(pos, new EntryDefault(key, value));//再替之以新数据对象
        return oldValue;//最后返回原先的数据对象。注意：返回null时的歧义
      }
    }//若此循环结束，说明key尚未在M中出现，因此
    L.insertFirst(new EntryDefault(key, value));//将新条目插至表首，并
    return null;//返回null标志
  }

//若M中存在以key为关键码的条目，则删除之并返回其数据对象；否则，返回null
  public Object remove(Object key)  {
    Iterator P = L.positions();
    while (P.hasNext()) {//逐一对比
      Position pos = (Position)P.getNext();//各个位置
      Entry entry = (EntryDefault) pos.getElem();//处的条目
      if (T.isEqualTo(entry.getKey(), key)) {//若发现key已出现在某个条目中，则
        Object oldValue = entry.getValue();//先保留该条目原先的数据对象
        L.remove(pos);//删除该条目
        return oldValue;//最后返回原先的数据对象。注意：返回null时的歧义
      }
    }//若此循环结束，说明key尚未在映射中出现，因此
    return null;//返回null标志
  }

//返回M中所有条目的一个迭代器
  public Iterator entries()
  { return new IteratorElement(L); }//直接利用List接口的方法生成元素迭代器
}
```

代码六.4 基于列表实现的映射结构

上述实现虽然简单，但效率不高。为了执行其中的 get(key)、put(key, value)或 remove(key)方法，都需要扫描整个列表，因此若列表规模（即映射规模）为 n，则这些方法的时间复杂度都是 $\mathcal{O}(n)$。当映射规模较大时，这一缺陷尤为明显，为此我们必须进行改进。

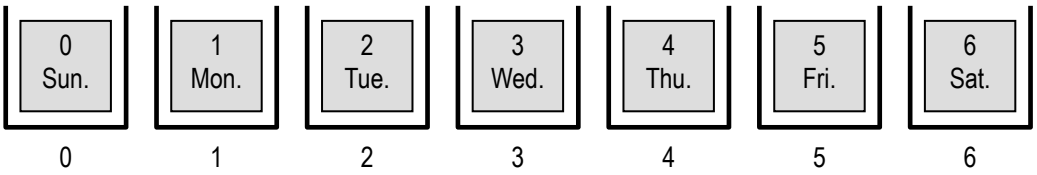
§6.2 散列表

如果将条目的关键码视作其在映射结构中的存放位置，则可以散列表（Hash t able）的形式来实现映射结构。与第 6.1.5 节基于列表的实现相比，基于散列表的实现效率有极大的提高。虽然就最坏情况而言，单次查找依然可能需要$O(n)$时间，但是就期望的平均性能而言，映射ADT中的所有操作都可以在$O(1)$时间内完成。

总体来说，散列表由两个要素构成：桶数组与散列函数。下面将分别进行讨论。

6.2.1 桶及桶数组

散列表使用的桶数组（Bucket array），其实就是一个容量为 N 的普通数组 A[0..N-1]，只不过在这里，我们将其中的每个单元都想象为一个“桶”（Bucket），每个桶单元里都可以存放一个条目。比如，倘若所有的关键码都是小于 N 的非负整数，我们就可以直接将 key 为关键码的那个条目（如果存在的话）存放在桶单元 A[key]内；为了节省空间，空闲的单元都被置为 null。既然映射结构要求不同条目的关键码互异，故每个桶单元中至多只需存放一个条目。如此一来，查找、插入和删除操作都可以在 $O(1)$时间内完成！



图六.2 将周日、周一至周六组织为一个容量为7的桶树组，即可在常数时间内完成每次查询

然而，为此欢呼雀跃还为时尚早，其实，上面提到的只是一种极为特殊的情况，通常时候往往不会这么碰巧。比如，与数组结构在其它场合暴露出来的缺陷一样，这里同样很难确定数组的最佳容量 N，故往往选用远大于映射实际规模 n 的某个 N。比如，若关键码都是 Int 类型的非负整数，则至少需要 2^{31} 个桶单元，这远远超出了一般映射结构本身的规模，故无疑是空间上的巨大浪费。另一个问题在于，这里假定关键码都是整数而且介于 0 与 N-1 之间，这一点在实际应用中也很难保证。为解决这些问题，我们需要某一函数，将任意关键码转换为介于 0 与 N-1 之间的整数——这个函数就是所谓的散列函数（Hash function）。

6.2.2 散列函数

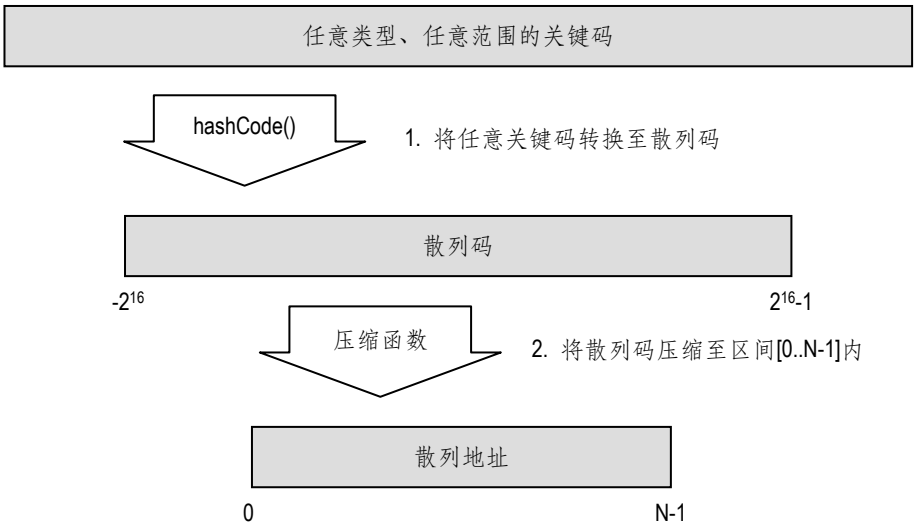
如上所言，为了将散列技术推广至一般类型的关键码，我们需要借助散列函数 h，将关键码 key 映射为一个整数 h(key) $\in$ [0. .N-1]，并将对应的条目存放至第 h(key)号桶内，其中 N 为桶数组的容量。如果将桶数组记作 A[]，这一技术就可以总结为“将条目 e = (key, value)存放至 A[h(key)]中”。反过来，为了查找关键码为 key 的条目，只需取出桶单元 A[h(key)]中存放的对象。因此，h(key)也被称作 e 的散列地址。

不过，若要兑现上述构思，还需要满足另一个条件——h()是一个单射，即不同的关键码key1 $\neq$ key2 必然对应于不同的散列地址h(key1) $\neq$ h(key2)。不幸的是，在绝大多数应用问题中，这一条件都很难满足。如果不同关键码的散列地址相同，我们就说该散列发生了冲突（Collision）。我们将在

第 6.2.6 节介绍解决冲突的办法，但是与其坐等冲突发生再来解决，还不如首先在设计和选择散列函数时多下功夫，以尽可能地降低发生冲突的概率。于是，这样一个问题就摆在了我们面前：选择什么样的散列函数才更好？

一个好的散列函数 h()必须兼顾以下两条基本要求：首先，h()应该尽可能接近单射；另外，对于任何关键码 key，h(key)的计算必须能够在 $\mathcal{O}(1)$时间内完成。

关于散列函数的计算，Java有其特有的习惯。Java将h(key)的计算划分为两步：如 图六.3 所示，首先将一般性的关键码key转换为一个称作“散列码”的整数，然后再通过所谓的“压缩函数”将该整数映射至区间[0..N-1]内。



图六.3 Ja va计算散列函数的过程

6.2.3 散列码

如上所言，Java 可以帮助我们将任意类型的关键码 key 转换为一个整数，称作 key 的散列码（Hash code）。请注意，散列码距离我们最终所需的散列地址还有很大距离——它不见得落在区间[0..N-1]内，甚至不见得是正的整数。不过这并不要紧，在这一阶段我们最关心的是：各关键码的散列码之间，应尽可能地减少冲突。显然，要是在这一阶段就发生冲突，后面的冲突就无法避免。此外，从判等器的判等效果来看，散列码必须与关键码对象相互一致：被判等器 EqualityTester 判定为相等的两个关键码，对应的散列码也应该相等。

■ Jav a 中的散列码

Java 的通用类 Object 提供了一个默认的散列码转换方法 hashCode()，利用它可以将任意对象实例映射为“代表”该对象的某个整数。具体来说，hashCode()方法的返回值是一个 32 位 int 型整数。通常，这一方法会被 Java 中的每个对象继承。实际上，这一默认 hashCode()方法所返回的不过就是对象在内存中的存储地址。遗憾的是，这一看似再自然不过的方法，实际上存在着严重的缺陷，因此我们在使用时需格外小心。

比如，这种散列码的转换办法对字符串型关键码就极不适用。若两个字符串对象完全相等，本应该将它们转换为同一散列码，但由于它们的内存地址不同，由 hashCode()得到的散列码将绝对不

会一样。实际上，在实现 String 类时，Java 已经将 Object 类的 hashCode()方法改写为一种更加适宜于字符串关键码的方法。如果你需要在映射结构中使用特定类的关键码，那么最好也按照前面所讲的原则，重写出更为适宜的专用 hashCode()方法。

下面，我们就针对常见的若干数据类型，介绍对应的散列码转换办法。

■ 强制转换为整数

对于那些可以表示为不超过 32bit 形式的数据类型（比如 Java 的基本类型 byte、short、int 和 char），我们可以直接用它们的这种表示来作为散列码。为此，我们只需通过类型强制转换，将它们转化为 32bit 的整数。这一方法也同样适用于 float 类型的变量 x，比如我们可以利用调用 Float.floatToBits(x)，将返回的整数作为散列码。

■ 对成员对象求和

对于 long 和 double 之类长度超过 32bit 的基本类型，若套用上面强制转换的办法，则必然会丢失某些位的信息，从而导致大量的冲突。可行的办法是，将高 32 位和低 32 位分别看作两个 32 位整数，将二者之和作为散列码。相应的实现如下：

```
static int hashCode(long i)
{ return (int) ((i>>32) + (int) i); }
```

这一方法可以推广至由任意多个整数构成的合成对象，我们可以将其成员对象各自对应的整数累加起来，截取低 32 位作为散列码。

■ 多项式散列码

字符串虽然也可以看作是由多个字符组合而成的对象，但这种对象与通常的组合对象不同——其中各字符之间具有特定的次序。同一组字符，可以组成意义完全不同的字符串，比如"shop"和"hops"、"step"和"pets"等。如果简单地将各字符对应的整数之和作为散列码，由同一组字符构成的所有字符串都会相互冲突。n 个互异字符能够组成 n!个不同的字符串，若 n=10，则这 10 个互异字符所能构成的 3,628,800 个字符串都会发生冲突。

为此，我们可以取一个非零的常数 $a \neq 1$。对于字符串"$x_0x_1\ldots x_{n-1}$"，可以取其散列码为：

$$x_0a^{n-1} + x_1a^{n-2} + \ldots + x_{n-2}a^1 + x_{n-1}$$

从数学的角度来看，这就相当于用该字符串中的字符依次作为一个多项式中各项的系数，因此称之为“多项式散列码”（Polynomial hash code）。

常数 a 的选取很有讲究。由于这种多项式的计算不可避免地会出现数值溢出，所以 a 本身的低位 bit 不能为零，否则也会造成大量的冲突。针对不同类型的字符串，应该通过实验确定 a 最佳的取值。比如，实验表明，对于英语单词之类的字符串，a = 33、37、39 或 41 都是不错的选择。

■ 循环移位散列码

多项式散列码的一个明显不足在于计算复杂，对该方法的一种改进，就是利用循环移位来模拟乘法运算的效果。具体的实现如下所示：

```
static int hashCode(String s) {
  int h = 0;
  for (int i=0; i<s.length(); i++) {
    h = (h<<5) | (h>>27);
    h += (int) s.charAt(i);
  }
  return h;
}
```

这里采用的是循环左移 5 位，对于英语单词来说，这也是通过实验统计得出的最佳值。

6.2.4 压缩函数

至此，我们已经掌握了将不同对象转化为散列码的办法，但这还不够。如果直接将散列码作为桶数组的单元地址，则桶的容量将达到 2^{32} = 4 G，对于大多数应用问题来说，即使能够提供如此大的空间，其利用率也往往极低。因此，有必要将散列码进一步压缩至我们希望的[0..N-1]区间内。

■ 模余法

最简单的压缩办法，就是取 N 为素数，并将散列码 i 映射为

$$|i| \mod N$$

之所以将 N 选取为素数，是为了最大程度地将散列码均匀地映射至[0..N-1]区间内。比如，对于散列码集合{200, 205, 210, 215, …, 690, 695, 700}，若选取 N = 100，则其中的每个散列码都会与另外的至少四个关键码相冲突；而若改用 N = 101，则不会有任何冲突。

若所有关键码都是在[0..N-1]内随机均匀分布的，则其中每一对关键码发生冲突的概率都是 1/N。因此，越是能够使得这个概率接近于 1/N，我们选用的散列函数就越好。选择素数 N 是一个简单易行的策略，但亦非尽善尽美。比如，若关键码的分布具有 pN + q 的模式，则仍然会发生不少冲突。

■ MAD 法

为了解决上述问题，可以采用一种将乘法（Mutiply）、加法（Add）和除法（Divide）结合起来的方法，该方法也因此得名。具体来说，对于散列码 i，MAD 法会将 i 映射为：

$$|a \times i + b| \mod N$$

其中 N 仍为素数，a>0，b>0，$a \bmod N \neq 0$，它们都是在确定压缩函数时随机选取的常数。

6.2.5 冲突的普遍性——生日悖论

散列表的基本思想，是采用一个桶数组 A[]，然后借助一个散列函数 h()，根据条目(key, value)的关键码 key 直接得到其对应的桶单元编号 A[h(key)]，从而快速地完成访问与修改。然而遗憾的是，很难保证不同关键码所对应的桶编号不致冲突。实际上，不发生任何冲突的概率是非常小的。

我们可以考虑这样一个实际问题：某课堂上的所有学生中，是否由某两位在同一天过生日（称作生日巧合）？请注意，这里只考虑生日的月份与日子，而不管具体年份。不过在这里，我们更加关心的问题是：发生生日巧合的概率是多少？

不妨将一年当作一个容量为 365 的桶数组，以生日作为关键码将所有学生组织为一个散列表。于是，上述关于生日巧合的问题就可以严格地转化并表述为：对于这样的散列表，关键码之间至少发生一次冲突的可能性有多大呢？我们将 n 个学生在场时对应的这一概率记作 $P_{365}(n)$。

若学生总数 $n > 365$，则根据鸽巢原理，冲突将注定出现，即 $P_{365}(n) = 100\%$。

对于 $n \le 365$，运用排列组合的知识不难证明：

$$P_{365}(n) = 1 - \frac{365!}{(365-n)! \times 365^n}$$

这一概率与我们的知觉相差很大。实际上，哪怕只有 23 个学生在场，你也值得打赌认定存在生日的巧合——因为 $P_{365}(23) = 50.7\%$。

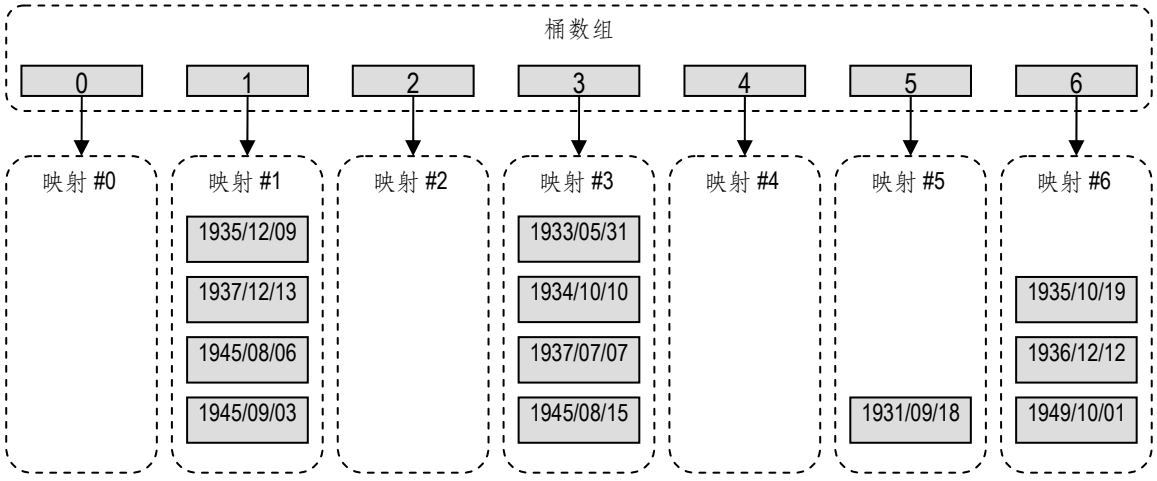
随着学生的增多，这一概率会急剧上升。如果每堂课都有 100 个学生参加，都持续 15 分钟，那么即使你在自己的有生之年废寝忘食、夜以继日地上课，也未必能够“碰巧”遇到不发生生日巧合情况的一堂课。

6.2.6 解决冲突

由上面的分析可知，冲突是普遍存在的，我们可以设法降低冲突发生的可能性，但最终都无法彻底回避冲突。那么，一旦发生冲突，有什么有效的方法可以解决冲突呢？

- 分离链（Separate chaining）

解决冲突最直截了当的一种办法，就是将所有相互冲突的条目组成一个（小规模的）映射结构，存放在它们共同对应的桶单元中。也就是说，桶单元 A[i] 对应于映射结构 M_i，其中存放所有满足 h(key) = i 的条目(key, value)。



图六.4 利用分离链解决散列冲突

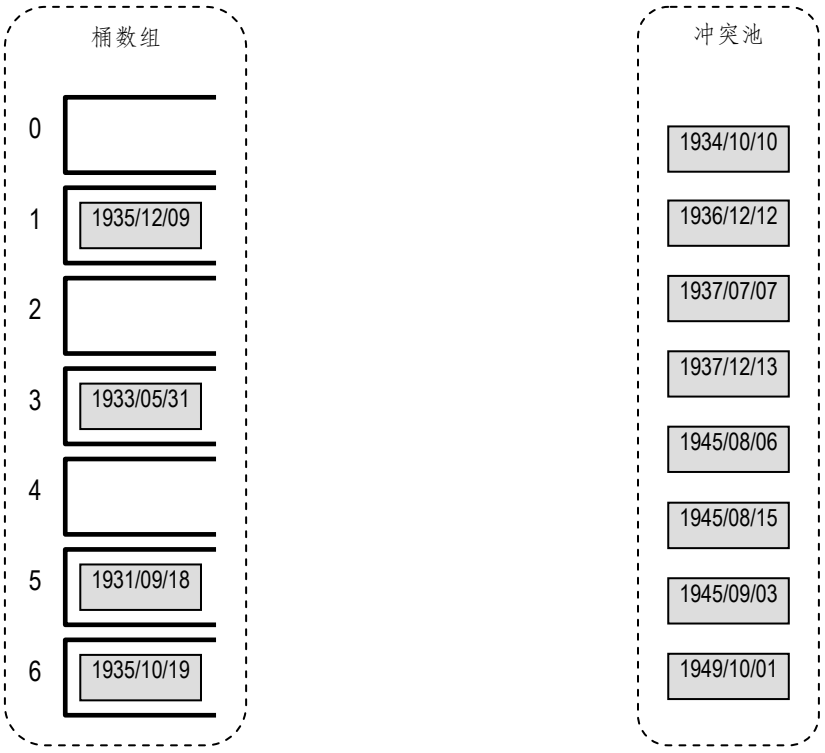
如 图六.4 所示，按照这一思路，针对关键码key的任何操作，都将转化为对一个与之相对应的映射结构$M_{h(key)}$的操作。比如put(key, value)操作，将首先在$M_{h(key)}$中查找关键码等于key的条目。若存在这样的条目，则将其中的数据对象替换为value；否则，生成一个新条目(key, value)，并将其插入$M_{h(key)}$中。get(key)操作和remove(key)操作的过程也与此类似，都要首先在$M_{h(key)}$中查找关键码key。

实际上，既然好的散列函数能够将所有关键码尽可能均匀地分布到桶数组的各个单元，所以通常$M_{h(key)}$的规模的确都不大，其中相当多的桶中只含有单个条目，有些甚至是空的。因此，完全可以采用第 6.1.5 节介绍的办法，直接通过链表结构来实现每个M_i——这种解决冲突的策略也由此得名。

假设我们正在将 n 个条目组织为散列式映射结构，如果用的桶数组容量为 N，我们希望每个桶单元的容量都接近 n/N——这个值称为该散列表的装填因子（Load factor），记作λ = n/N。只要选用的散列函数足够好，基于散列表实现的映射结构的各种基本操作都能够在 $O(\lceil n/N \rceil)$的时间内完成。通常的装填因子λ都小于 1（即选取 N>n），故这些操作的时间复杂度都将是 $O(1)$。

- 冲突池（Collision pool）

解决冲突的另一种办法，就是在散列表A[]之外另设一个映射结构P，一旦（在插入条目时）发生冲突，就将冲突的条目存入P中（如 图六.5 所示）。从效果来看，这相当于将所有冲突的条目存入一个缓冲池，该方法也因此得名。



图六.5 利用冲突池解决散列冲突

为此，get(key)、put(key, value)和 remove(key)操作中的查找算法都需做相应的调整：如果桶单元 A[h(key)]为空且不带记号，则报告“查找失败”；否则，若该桶中存放的不是所需的条目，则转到 P 中继续查找。

remove(key)算法也需做相应的调整：如果找不到待删除的条目，则操作可以立即完成；若在冲突池 P 中找到该条目，则可利用映射结构 ADT 提供的操作来实施删除；若待删除条目出现在散列表的桶 A[h(key)]中，则不仅需要删除该条目，还要继续给该桶做个标记——否则，在冲突池中与 key 冲突的那些关键码都将会“丢失”掉。

冲突池法的构思简单，易于实现，在冲突不甚频繁的场合，仍不失为一种较好的选择。

其实，由于冲突池本身也是一个映射结构，故这种结构也可以理解为散列表的递归形式。

■ 开放定址（Open addressing）

分离链策略可以非常便捷地实现映射结构的各种操作算法，但也不是尽善尽美。比如，就数据结构本身而论，这一策略需要借助列表作为附加结构，将相互冲突的条目分组存放。这不仅会增加代码出错的可能，而且也需要占用更多的空间。在一些十分讲求空间效率的应用（比如，手持移动设备的操作系统或应用程序）中，这些矛盾将尤为突出。在这类情况下，我们只能在不借助附加结构的条件下解决散列的冲突。开放定址就是这样一种策略，实际上，由这一策略可以导出一系列的变型，比如线性探测法、平方探测法以及双散列法等等。由于不能使用附加空间，所以开放定址策略要求装填因子$\lambda \le 1$，通常都要保证$\lambda \le 0.5$。

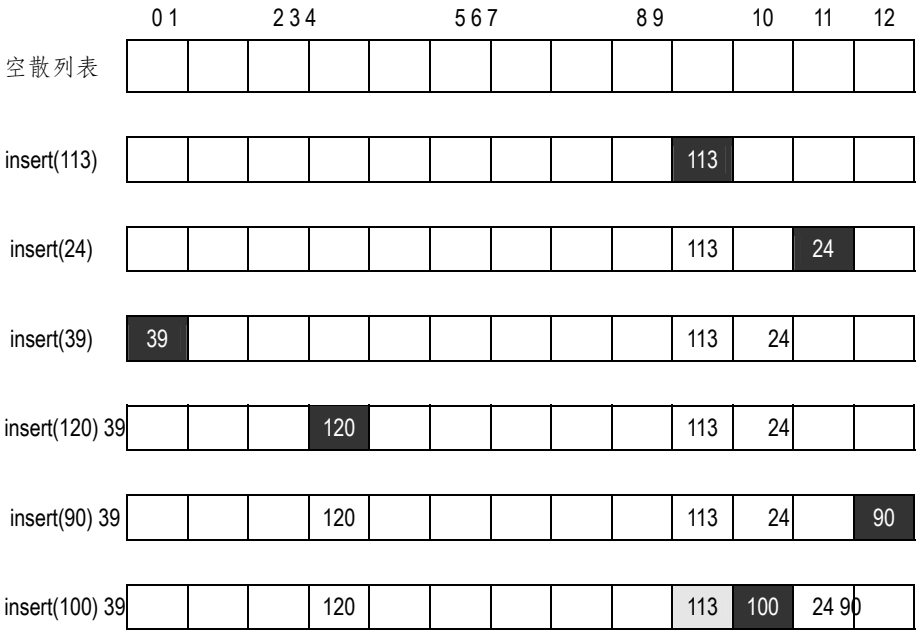
■ 线性探测（Linear probing）

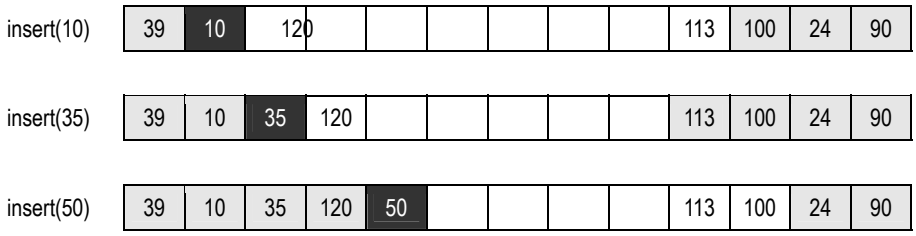
采用开放定址策略，最简单的一种形式就是线性探测法。也就是说，在执行 put(key, value)操作时，倘若发现桶单元 A[h(key)]已经被占用，则转而尝试桶单元 A[h(key)+1]。要是 A[h(key)+1]也被占用了，就继续尝试 A[h(key)+2]。如果还有冲突，则继续尝试 A[h(key)+3]。如此不断地进行尝试，直到发现一个可以利用的桶单元。当然，为了保证桶地址的合法性，第 i 次尝试的桶单元应该为 A[(h(key)+i) mod N]，i = 1, 2, 3, …。

按照这种办法，被尝试的桶单元地址构成一个线性等差数列，故由此得名。

相应地，get(key)和 remove(key)操作中的查找算法也需要有所调整。此时，若首次对 A[h(key)]的查找失败，并不意味着关键码 key 未在映射中出现。实际上，我们还需要依次扫描 A[h(key)]后续的各个桶单元 A[h(key)+1]、A[h(key)+2]、A[h(key)+3]、…，直到发现关键码为 key 的条目（查找成功），或者到达一个空桶（查找失败）。同样地，为了保证桶地址的合法性，这里也需要对桶地址关于 N 取模。需要指出的是，由于保证了$\lambda = n/N < 1$，所以上述查找不致出现无限循环的情况，在经过最多 n 次迭代之后，必然会得到确定的结论。

关于如何建立基于线性探测策略的散列表，请参见 图六.6 给出的实例。





图六.6 建立一个基于线性探测法的散列表：桶数组容量为13，采用模余压缩函数

定义六.1 若条目 e = (key, value)在长度为 N 的散列表 A 中被存放于 A[j]，i = h(key)，则 A[i]、A[(i+1) mod N]、A[(i+2) mod N]、…、A[j]称作 e 的**查找前驱桶单元**。

为了顺利地实现线性探测的构思，需要具备一定的条件，其中最重要的一条如下：

观察结论六.1 （**查找前驱桶非空条件**）设 E 为存放于长度为 N 的散列表 A 中的一组条目，则可以根据相互冲突的关系将 E 中的所有条目划分为若干等价类：

$$C_i = \{e = (key, value) \in E \mid h(key) = i\}, i = 0, \ldots, N-1$$

若 H 是基于线性探测法实现的，则对于任一条目 $e \in C_i$，e 的**查找前驱桶单元**均非空。

只有如此，才能保证线性探测式查找的顺利进行，因此也称之为线性探测法的“桶地址连续条件”。这一条件不仅需要在各种操作进行之前得到满足，而且在每一操作完成之后也应继续保持，以便进行后续的操作。不难看出，对于 get(key)和 put(key, value)操作而言，这一点都不成问题。然而，remove(key)操作却未必。

试考察 e = (key, value)所属的等价类 C_i，i = h(key)，不妨设$|C_i| \geq 2$。在按照上述查找算法确定了 e 所在的桶单元之后，倘若简单地将其移除，则可能会使（C_i 内甚至 C_i 外的）某一条目 f 的某个查找前驱桶单元为空，从而使“**查找前驱桶非空条件**”将不再保持。于是，在后续针对 f 的查找中，即使 f 存在，也将因无法抵达 f 而报告“查找失败”。

为了解决这一问题，在每次将目标条目 e 移除之后，还需要将以 e 为查找前驱单元的所有条目依次前移一个单元，填补删除 e 后留出的空桶，以保持查找前驱桶非空条件。不过，这一解决办法无疑会增加 remove(key)操作的时间复杂度。

在强调删除效率的场合，可以采用另一种变通的解决办法，以避免大量条目的移动。每次移除条目 e 之后，可以为 e 原先占用的空桶做个特殊的记号。如此一来，查找算法只需稍做修改：每次遇到标有这种记号的桶单元，都直接转向后继。另外，put(key, value)操作中的查找算法也需要有所改动：在查找的过程中，需要记录下最靠前的带有记号的桶单元——若最终查找失败，则将新条目存放到其中。

由上可见，尽管线性探测法可以节约空间，但各操作的相应实现却要复杂得多。然而，这一办法并不为人们青睐的原因，还远不止此。线性探测法的最大缺陷在于，基于这一策略的散列表中往往会存在大量的条目堆积（Clustering）。实际上，因为不能使用任何附加的空间，所以线性探测法每解决一次冲突都必然占用一个空桶，于是未来发生冲突的可能性也随之增加。比如，我们拟将一组条目{3, 16, 17, 18, 19, …, 25}（这里不妨忽略各条目的数据对象，只考虑其关键码）组织为一个

散列表。假设散列表长 N = 13，采用模余法确定地址。若按照关键码递增的次序逐一插入，则除 3 以外的所有条目都对应于一次冲突，总共出现 10 次。反过来，若按照关键码递减的次序逐一插入，则只有在插入 3 时才出现一次冲突。前一次序之所以出现大量的冲突，正是由于条目的堆积而导致的。

实验统计表明，在一般情况下条目堆积现象也很普遍，当装填因子超过 0.5 时，这一问题更为突出。平方探测法就是克服这一缺点的一种有效办法。

- 平方探测（Quadratic probing）

平方探测法是对线性探测法的改进。具体来说，就是在发生冲突时，依次对桶单元

$$A[(h(key) + j^2) \bmod N],\ j = 0, 1, 2, \ldots$$

进行探测，直到发现一个可用的空闲桶。

这一策略可以很好地解决条目堆积的问题。这里充分利用了二次函数的特点，随着冲突次数的增加，其探测的步长将以线性的速度增长，而不是像线性探测法那样始终采用固定步长（=1）。因此，一旦发生冲突，这一办法可以使待插入条目快速地“跳”离条目聚集的区段。

不过，这一方法也存在一些不足。首先，与线性探测法一样，基于平方探测法的 remove(key) 操作实现非常复杂。

其次，尽管这一策略可以有效回避条目堆积的现象，但还是会出现所谓的二阶聚集（Secondary clustering）现象——条目虽然不会连续地聚集成片，却会在多个（间断的）位置多次“反弹”。

最后，如果散列表容量 N 不是素数，则有可能会出现循环反弹，以至于无法插入的情况。即使 N 选为素数，也必须保证装填因子$\lambda < 0.5$，否则即便存在空桶，仍有可能无法找不到插入位置。

- 双散列（Double hashing）

双散列也是克服条目堆积现象的一种有效办法。为此我们需要选取一个二级散列函数 g()，一旦在插入 e = (key, value)时发现 A[h(key)]已被占用，则不断尝试 A[h(key) + j×g(key)]，j = 1, 2, …。

显然，对任何关键码 key，函数值 g(key)都不能为零——否则就会在“原地踏步”。通常，若散列表长度为素数 N，则取另一素数 q < N，并取

$$g(key) = q - (key \bmod q)$$

为了尽可能地使关键码均匀分布，可以通过实验统计确定最佳的 q 值。

- 综合比较

相比而言，分离链策略的算法简单，但需要耗费更多空间。开放定址策略正好相反，可以尽可能地节省空间，但算法需做较复杂的调整。理论分析和实验统计都表明，就通常的桶数组容量 N 与装填因子λ而言，分离链策略的时间效率要远远高于其它的方法。因此，除非在存储空间非常紧张的场合，我们都建议采用分离链策略解决冲突。

6.2.7 基于散列表实现映射类

如 代码六.5 所示，我们给出基于散列表实现的映射结构：

```
/*
 * 基于散列表实现的映射结构
 * 采用分离链策略解决冲突
 */

package dsa;

public class Map_HashTable implements Map {
  private Map[] A;//桶数组，每个桶本身也是一个（基于列表实现的）映射结构
  private int N;//散列表长
  private final double maxLemda = 0.75;//装填因子上限
  private int size;//映射结构的规模
  private EqualityTester T;//判等器

//默认构造方法
  public Map_HashTable()
  { this(0, new EqualityTesterDefault()); }

//构造方法
  public Map_HashTable(int n, EqualityTester t) {
    T = t;
    N = p(n);//桶数组容量取为不小于n的最小素数
    A = new Map[N];
    for (int i=0; i<N; i++) A[i] = new Map_DLNode(T);
    size = 0;
  }

/***************************** 辅助方法 *****************************/
//散列定址函数（采用模余法）
  private int h(Object key)
  { return key.hashCode() % N; }

//判断n是否为素数
  private static boolean prime(int n) {
    for (int i=3; i<1+Math.sqrt(n); i++)
      if (n/i*i == n) return false;
    return true;
  }

//取不小于n的最小素数
  private static int p(int n) {
    if (3>n)  n = 3;
    n = n | 1;//奇数化
    while (!prime(n)) n += 2;
    return n;
  }

/***************************** ADT方法 *****************************/
```

```
//查询映射结构当前的规模
  public int getSize()
  { return size; }

//判断映射结构是否为空
  public boolean isEmpty()
  { return 0==size; }

//若M中存在以key为关键码的条目，则返回该条目的数据对象；否则，返回null
  public Object get(Object key)
  { return A[h(key)].get(key); }

//若M中不存在以key为关键码的条目，则将条目(key, value)加入到M中并返回null
//否则，将已有条目的数据对象替换为value，并返回原先的数据对象
  public Object put(Object key, Object value) {
    Object oldValue = A[h(key)].put(key, value);
    if (null==oldValue) {//若插入的条目未出现于原散列表中，则
      size++;//更新规模记录
      if (size > N * maxLemda)  rehash();//若装填因子过大，则重散列
    }
    return oldValue;
  }

//若M中存在以key为关键码的条目，则删除之并返回其数据对象；否则，返回null
  public Object remove(Object key) {
    Object oldValue = A[h(key)].remove(key);
    if (null!=oldValue) size--;
    return oldValue;
  }

//返回M中所有条目的一个迭代器
//将各桶对应的映射结构的迭代器串接起来，构成整体的迭代器
  public Iterator entries() {
    List L = new List_DLNode();
    for (int i=0; i<N; i++) {
      Iterator it = A[i].entries();
      while (it.hasNext())  L.insertLast(it.getNext());
    }
    return new IteratorElement(L);
  }

//重散列
  private void rehash() {
    Iterator it = this.entries();
    N = p(N<<1);
    A = new Map[N];//桶数组容量至少加倍
    for (int i=0; i<N; i++) A[i] = new Map_DLNode(T);//为每个桶分配一个子映射
```

```
    while (it.hasNext()) {//将其对应的映射结构中的
      Entry e = (Entry)it.getNext();//各条目逐一取出，将其
      Object k = e.getKey();//关键码和

      Object v = e.getValue();//数据对象

      A[h(k)].put(k, v);//整合为新的条目，插入对应的子映射中
    }
  }
}
```

代码六.5 基于散列表实现的映射结构

上述实现利用了第 6.1.5 节给出的Map_DLNode类。正如前面所构思的，这里的每个桶单元都对应于一个Map_DLNode类的子映射结构，分别对应于某一组相互冲突的条目。这样，在根据给定关键码确定了其对应的桶单元及子映射结构之后，对Map_HashTable映射结构的各操作方法都可以借助Map_DLNode对象对应的方法加以实现。

为了生成迭代器，只需将所有 Map_DLNode 子映射结构各自的迭代器串接起来。因此，只要每个桶（子映射结构）对应的迭代器可以在线性时间内生成，则整个映射结构的迭代器也可以在线性时间内生成。

6.2.8 装填因子与重散列

■ 装填因子

对于散列表的性能而言，装填因子λ = n/N 是最重要的影响因素。如果λ > 1，则冲突在所难免。实际上，关于散列表平均复杂度的分析结果指出，采取分离链策略时应该保持λ < 0.9，采取开放定址策略时则应该保持λ < 0.5——这些都得到了实验统计的证明，建议读者自己动手对这些结论做一验证。

若采用分离链策略，则在发生冲突的桶中，对条目的查找将退化为对链表的查找。因此，随着λ不断接近于 1，发生冲突的概率也将不断接近于 100%，从而导致更多的时间消耗于对链表的查找，使得各种操作的效率下降。在最坏的情况下，几乎所有的条目都聚集在同一个桶中。此时，对映射结构的操作将退化为对单一链表的操作，因此每次操作都需要 $\mathcal{O}(n)$时间。当然，只要选用适宜的散列函数，这种情况发生的概率几乎为零。

当采用开放定址策略时，随着λ超过 0.5 并不断提高，条目在桶数组中聚集的程度将急速加剧，于是，需要经过越来越多次的探测才能完成一次查找。

■ 重散列

综上所述，无论采用分离链策略还是开放定址策略，都必须将装填因子限制在一定范围以下。这对保证查找的效率来说十分重要，对开放定址策略来说这一点尤为关键。一旦装填因子过大，则

须采取措施将其降低。为此，通常都采用重散列的方法——将所有条目全部取出，将桶数组的规模加倍，然后将各条目重新插入其中（参见 代码六.5 中的rehash()方法）。

例如，Java 内建的 java.util.HashMap 类实现了映射结构的 ADT，在创建该类的对象时，程序员可以指定装填因子的上限（默认设置为 0.75）。一旦装填因子超出这一范围，java.util.HashMap 会自动进行重散列（Rehashing）。

如果不计其中函数p()和prime()消耗的时间，每次调用rehash()方法对容量为N的散列表做重散列都需要$\mathcal{O}(N)$的时间。因此，与第 3.1.3 节中对基于可扩充数组实现的向量的分析同理，每次重散列都应该至少将桶数组的容量加倍，唯此方能将重散列操作的分摊复杂度控制在$\mathcal{O}(1)$量级。

然而，如果要严格地保证桶数组容量始终为素数，则需要更多的时间。以 代码六.5 为例，我们需要从 2N+1 开始不断尝试，直到第一个至少是原容量N两倍的素数。这里采用的方法并未经过优化，为了判断一个奇数n是否为素数，需要做$\mathcal{O}(\sqrt{n})$乘法和除法。对于通常的应用问题来说，n不会太大，可以采用素数查找表之类的办法来加速这一计算。

实验表明，经过重散列之后，原先聚集的条目一般都会比较均匀地分散开。尽管依然有可能需要做多次重散列，散列表仍不失为实现映射结构的一种行之有效的办法。

§6.3 无序词典

与前面介绍的映射结构一样，词典结构也是用来存放条目对象的一种容器，而且对其中条目的类型没有限制，只要是形如(key, value)的组合对象即可。不过，词典与映射之间有一个非常重要的差别——词典不再要求其中各条目的关键码互异。这一点与我们日常使用的纸介质词典类似，不少单次都具有多种解释，每一种解释分别对应于一个词条。因此，我们往往将词典中的条目直接称作“词条”。

总体而言，词典可以分为两大类：无序词典和有序词典。顾名思义，前一类词典中存放的条目无所谓次序，我们只能（利用某一判等器）比较一对条目（的关键码）是否相等；而在后一类词典所存放的条目之间，则（根据某一比较器）定义了某种全序关系，因此也相应地能够支持 first()、last()、prev()和 succ()之类的方法。

本节将讨论无序词典的描述、实现和分析，第 §6.4 节将讨论有序词典。

6.3.1 无序词典的 ADT 描述

作为一个容器，词典 D 首先必须支持以下一般性的操作方法：

表六.4 词典ADT支持的操作

操作方法	功能描述
getSize():	报告词典的规模，即其中元素的数目 输入：无 输出：非负整数

操作方法	功能描述
isEmpty():	判断词典是否为空 输入：无 输出：布尔标志

就其抽象数据类型的特点而言，映射 M 还需要支持以下独有的操作方法：

表六.5 词典ADT特有的操作

操作方法	功能描述
find(key):	若词典中存在以 key 为关键码的条目，则返回其中的一个条目 否则，返回 null 输入：一个关键码对象 输出：条目对象
findAll(key):	若词典中存在以 key 为关键码的条目，则返回这些条目组成的迭代器 否则，返回 null 输入：一个关键码对象 输出：条目对象的迭代器
insert(key, value):	插入条目(key, value)，并返回该条目 输入：一个关键码对象及一个数据对象 输出：条目对象
remove(key):	若词典中存在以 key 为关键码的条目，则将摘除其中的一个并返回 否则，返回 null 输入：一个关键码对象 输出：条目对象
entries():	返回词典中所有关键码对象的一个迭代器 输入：无 输出：条目对象的迭代器

请注意，这里只提供了 findAll()方法，而没有提供 removeAll()方法。其原因在于，只需反复调用 remove(key)方法，即可完成“将关键码为 key 的所有条目删除”的任务。

需要特别指出的是，Java本身提供的抽象类java.util.Dictionary和java.util.Map并不属于我们在这里定义的词典，它们更接近于第 §6.1 节中介绍的映射ADT，因为这两个抽象类都禁止不同条目拥有同一关键码。实际上，与 第五章中的优先队列结构一样，Java本身并未提供任何与这里的词典ADT相对应的抽象类。

6.3.2 无序词典的 Java 接口

根据对无序词典的上述ADT描述，可以得到如 代码六.6 所示的无序词典结构的Java接口：

```
/*
 * （无序）词典结构接口
 */
```

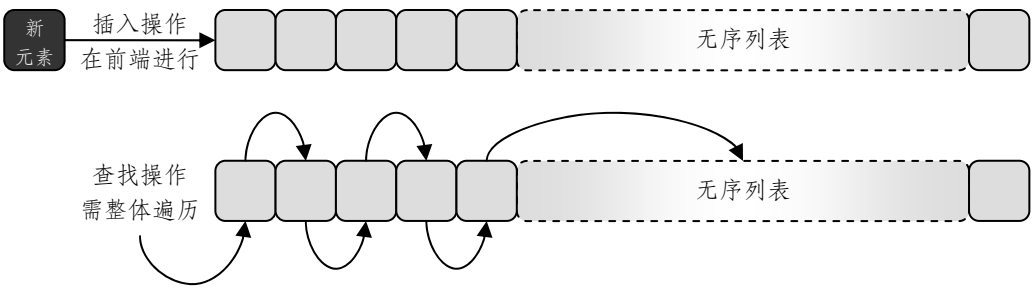
```
package dsa;

public interface Dictionary {
//查询词典结构当前的规模
  public int getSize();

//判断词典结构是否为空
  public boolean isEmpty();

//若词典中存在以key为关键码的条目，则返回其中的一个条目；否则，返回null

  public Entry find(Object key);


//返回由关键码为key的条目组成的迭代器
  public Iterator findAll(Object key);

//插入条目(key, value)，并返回该条目
  public Entry insert(Object key, Object value);

//若词典中存在以key为关键码的条目，则将摘除其中的一个并返回；否则，返回null
  public Entry remove(Object key);

//返回词典中所有条目的一个迭代器
  public Iterator entries();
}
```

代码六.6 无序词典结构的Java接口

与映射结构的情况一样，这里也有一个关于退化情况的处理问题：如果因未能找到具有某一指定关键码的条目而导致查找失败，应该如何表示？这里依然沿用第 6.1.1 节的办法：一旦查找失败，就直接返回null。

6.3.3 列表式无序词典及其实现

实现上述无序词典ADT的一种简单办法，就是如 图六.7 所示，利用无序列表来存放所有的条目。于是，为了插入新条目，我们只需将其接至表的前端（或后端）；为了删除条目，只需沿着列表逐一检查各条目，直到发现一个与指定关键码吻合的条目；其它方法也可以类似地实现。



图六.7 利用无序列表实现词典结构的原理

■ 基于无序列表实现的无序词典

利用无序列表实现的无序词典结构，如 代码六.7 所示。

```
/*
 * 基于列表实现（无序）词典结构
 */

package dsa;


public class Dictionary_DLNode implements Dictionary {


  private List L;//存放条目的列表
  private EqualityTester T;//判等器

//构造方法
  public Dictionary_DLNode()
  { this(new EqualityTesterDefault()); }

//默认构造方法
  public Dictionary_DLNode(EqualityTester t)
  { L = new List_DLNode();  T = t; }

/***************************** ADT方法 *****************************/
//查询词典结构当前的规模
  public int getSize()
  { return L.getSize(); }

//判断词典结构是否为空
  public boolean isEmpty()
  { return L.isEmpty(); }

//若词典中存在以key为关键码的条目，则返回其中的一个条目；否则，返回null
  public Entry find(Object key) {
    Iterator P = L.positions();
    while (P.hasNext()) {
      Position pos = (Position)P.getNext();
      Entry entry = (EntryDefault) pos.getElem();
      if (T.isEqualTo(entry.getKey(), key)) return entry;
    }
    return null;
  }

//返回由关键码为key的条目组成的迭代器
  public Iterator findAll(Object key) {
    List  list = new List_DLNode();
    Iterator P = L.positions();
    while (P.hasNext()) {
      Position pos = (Position)P.getNext();
      Entry entry = (EntryDefault) pos.getElem();
```

```
      if (T.isEqualTo(entry.getKey(), key))
        list.insertLast(entry);
    }
    return new IteratorElement(list);
  }

//插入条目(key, value)，并返回该条目
  public Entry insert(Object key, Object value) {
    Entry entry = new EntryDefault(key, value);//创建新条目
    L.insertFirst(entry);//将新条目插至表首，并
    return entry;//返回null标志

  }


//若词典中存在以key为关键码的条目，则将摘除其中的一个并返回；否则，返回null
  public Entry remove(Object key) {
    Iterator P = L.positions();
    while (P.hasNext()) {//逐一对比
      Position pos = (Position)P.getNext();//各个位置
      Entry entry = (EntryDefault) pos.getElem();//处的条目
      if (T.isEqualTo(entry.getKey(), key)) {//若发现key已出现在某个条目中，则
        Entry oldEntry = entry;//先保留该条目
        L.remove(pos);//删除该条目
        return oldEntry;//最后返回原先的条目
      }
    }//若此循环结束，说明key尚未在词典中出现，因此
    return null;//返回null标志
  }

//返回词典中所有条目的一个迭代器
  public Iterator entries()
  { return new IteratorElement(L); }//直接利用List接口的方法生成元素迭代器
}
```

代码六.7 基于无序列表实现的（无序）词典结构

■ 性能分析

借助无序列表 L 实现词典，每个词条只占用常数的空间，故总体只需 $O(n)$的空间。

在此，我们总是通过调用 L.insertFirst()方法将新条目插入至表首，因此每次插入操作只需 $O(1)$时间。

不幸的是，除了（通过迭代器）逐一检查每个条目之外，我们没有更好的办法对无序列表进行查找。也就是说，虽然在最好情况时 find()操作只需要常数时间，但在最坏情况下，每次 find()操作

都需要 $\mathcal{O}(n)$的时间。同理，每次 remove()操作也需要 $\mathcal{O}(n)$的时间。更糟的是，为了完成 findAll()操作，我们总是需要检查每一条目，因此该操作需要$\Theta(n)$的时间。

总体看来，如此实现的词典结构能够支持快速有效的插入操作，但查找与删除操作的效率却极低。但这并不意味着这一实现方法毫无价值。实际上，这一形式的词典结构在很多实际问题中都得到了广泛的应用。这方面的例子，包括 Internet 流量日志记录（Log file）、财务数据库系统的审计跟踪（Audit trail）等。这些应用问题具有一个共同的特点：相对而言，插入操作极其频繁，但查找、删除操作却极其罕见。比如热门网站的访问日志，几乎时时刻刻都需要插入大量的新条目，而只有在系统出现故障等情况下才需要对日志记录进行分析——果真如此，完全可以在需要时将日志专门整理为某种更加高效的结构，以便进行查找、对比、筛选和排序等处理。

当然，很多应用问题也对查找和删除操作的效率提出了很高的要求，显然，在这类场合就必须通过其它的形式来实现词典结构，比如下面将要介绍的基于散列表的实现。

6.3.4 散列表式无序词典及其实现

散列的思想不仅可以用以实现映射ADT，也可以用来实现无序词典ADT。具体来说，我们还是使用一个桶数组A[]，其中的每个桶分别对应于一组关键码相等（或者说相互冲突）的条目；我们采用分离链策略，将每一组这样的条目再组织为一个子词典，存放在对应的桶中。实际上，只要装填因子足够小，并选取适当的散列函数以保证各条目的均匀分布，则每个这类子词典的规模都不会很大，因此无需采用复杂的结构加以实现——比如，这里就直接利用了第 6.3.3 节中 代码六.7 基于无序列表实现的Dictionary_DLNode结构。

■ 基于散列表实现的无序词典

基于散列表实现的无序词典结构，如 代码六.8 所示：

```
/*
 * 基于散列表实现的（无序）词典结构
 * 采用分离链策略解决冲突
 */

package dsa;

public class Dictionary_HashTable implements Dictionary {
  private Dictionary[] A;//桶数组，每个桶本身也是一个（基于列表实现的）词典结构
  private int N;//散列表长
  private final double maxLemda = 0.75;//装填因子上限
  private int size;//词典结构的规模
  private EqualityTester T;//判等器

//默认构造方法
  public Dictionary_HashTable()
  { this(0, new EqualityTesterDefault()); }

//构造方法
  public Dictionary_HashTable(int n, EqualityTester t) {
```

```
    T = t;
    N = p(n);//桶数组容量取为不小于n的最小素数
    A = new Dictionary[N];
    for (int i=0; i<N; i++) A[i] = new Dictionary_DLNode(T);
    size = 0;
  }

/****************************** 辅助方法 ******************************/
//散列定址函数（采用模余法）
  private int h(Object key)
  { return key.hashCode() % N; }

//判断n是否为素数
  private static boolean prime(int n) {
    for (int i=3; i<1+Math.sqrt(n); i++)

      if (n/i*i == n) return false;

    return true;
  }

//取不小于n的最小素数
  private static int p(int n) {
    if (3>n)  n = 3;
    n = n | 1;//奇数化
    while (!prime(n)) n += 2;
    return n;
  }

/****************************** ADT方法 ******************************/
//查询词典结构当前的规模
  public int getSize()
  { return size; }

//判断词典结构是否为空
  public boolean isEmpty()
  { return 0==size; }

//若词典中存在以key为关键码的条目，则返回其中的一个条目；否则，返回null
  public Entry find(Object key)
  { return A[h(key)].find(key); }

//返回由关键码为key的条目组成的迭代器
  public Iterator findAll(Object key)
  { return A[h(key)].findAll(key); }

//插入条目(key, value)，并返回该条目
  public Entry insert(Object key, Object value) {
    Entry entry = A[h(key)].insert(key, value);//将新条目插至桶A[h(key)]对应的子词典
```

```
    size ++;//更新规模记录
    if (size > N * maxLemda)  rehash();//若装填因子过大，则重散列
    return entry;//返回null标志
  }

//若词典中存在以key为关键码的条目，则将其摘除并返回；否则，返回null
  public Entry remove(Object key) {
    Entry oldEntry = A[h(key)].remove(key);
    if (null!=oldEntry) size--;
    return oldEntry;
  }

//返回词典中所有条目的一个迭代器
  public Iterator entries() {
    List L = new List_DLNode();
    for (int i=0; i<N; i++) {
      Iterator it = A[i].entries();

      while (it.hasNext())  L.insertLast(it.getNext());

    }
    return new IteratorElement(L);
  }

//重散列
  private void rehash() {
    Iterator it = this.entries();
    N = p(N<<1);
    A = new Dictionary[N];//桶数组容量至少加倍
    for (int i=0; i<N; i++) A[i] = new Dictionary_DLNode(T);//为每个桶分配一个子词典
    while (it.hasNext()) {//将其对应的词典结构中的
      Entry e = (Entry)it.getNext();//各条目逐一取出，将其
      Object k = e.getKey();//关键码和
      Object v = e.getValue();//数据对象
      A[h(k)].insert(k, v);//整合为新的条目，插入对应的子词典中
    }
  }
}
```

代码六.8 基于散列表实现的（无序）词典结构

■ 性能分析

只要始终保证装填因子足够小，同时选用适当的散列函数，则每个桶对应的子词典都不至于太大。在满足这些条件的情况下，find()、insert()和 remove()操作的平均时间复杂度为 $\mathcal{O}(1)$，而 findAll()操作的复杂度为 $\mathcal{O}(1+m)$，其中 m 为查找命中的条目的实际数目。

需要特别注意的是，这里 findAll()操作复杂度的给出形式，是我们首次看到的。与通常度量复杂度的方式不同，该操作的执行时间不仅取决于输入的规模，而且反过来还取决于输出的规模。因此，这类算法也称作是“输出敏感的”（Output-sensitive）。

§6.4 有序词典

前面曾提到，基于无序列表实现的词典结构非常适用于解决网络访问日志之类的应用问题，这类问题的共同特点是：插入操作频繁，查找、删除操作却极少进行。另外一些问题则正好相反，它们要求频繁地进行查询，但插入、删除操作相对更少，这方面的例子包括在线电话簿、订票系统等。

就查找操作而言，散列表的平均性能不错。然而在最坏情况下（比如散列函数选取不当，或者运气不佳），可能几乎所有条目都会相互冲突，于是对散列表的查找会退化为对单一列表的查找。实际上，只要有Ω(n)个条目相互冲突，散列表的查找就需要Ω(n)的时间。

在对系统响应速度要求极高的场合（比如各种实时处理或服务系统），散列表的上述不足往往是致命的。比如在航天飞机的控制系统中，如果查找算法的效率时高时低，那么一旦出现查找的最坏情况，后果将不堪设想。在这类环境中，不仅要求查找算法的平均性能较好，而且算法在最坏情况下的效率也不能很低。为更好地满足上述应用的需要，我们只能放弃无序列表和散列表，转而求助于新的数据结构。

6.4.1 全序关系与有序查找表

图六.8 借助有序查找表实现词典结构

（假设关键码都是整数，这里只给出了各条目的关键码，而忽略了其中具体的数据）

只要在关键码之间定义有某一全序关系，我们就可以将词典中的条目组织为一个有序向量S，如 图六.8 所示。这里之所以选择向量而不是列表，是因为正如我们马上就要看到的，前者能够支持对词典的快速查找。如此实现的词典，也称作有序查找表（Ordered search table）。

如果采用第 3.1.3 节所介绍的可扩充数组，如此实现的词典结构只占用$\mathcal{O}$(n)的空间，这与第 6.3.3 节所介绍无序列表式词典相当。只要知道目标条目的秩，就可以在$\mathcal{O}$(1)时间内找到并访问该条目，这是列表式词典难以做到的。另外，我们马上就会看到，只要已知条目的关键码，则可以在$\mathcal{O}$(**log**n)的时间内找到该条目——相对于列表式词典$\mathcal{O}$(n)的查找效率，这是很大的改进。然而，有序词典的更新却需要更多的时间。为了保持有序查找表的完整性与一致性，每删除一个条目后，我们都需要将其后续的条目前移；在插入一个条目之前，我们都需要将其后续的条目后移。就最坏情况的复杂度而言，此类更新操作均需要$\mathcal{O}$(n)的时间。

6.4.2 二分查找

尽管有序查找表的更新效率不高，但由此却可以将查找效率提高至 $\mathcal{O}$(**log**n)。

我们采用通常的习惯，通过条目的秩来表示它们之间的全序关系，即：

key(a) ≤ key(b) 当且仅当 r ank(a) ≤ r ank(b)

形象地说，也就是“小（大）的条目排列在前（后）”。

在针对关键码 key 进行查找的过程中，若目前尚不能断定条目 e 是否被命中，就称之为“候选条目”（Candidate entry）。查找刚开始时，有序查找表中的所有条目 S[0..n-1]都是候选条目。

由于所有条目都是按序排列的，故可以采用“逐步缩小范围”的思想来实现查找。一般地，若候选条目的秩介于 lo 到 hi 之间，则取居中的秩 mi = ⌊(lo+hi)/2⌋，然后将条目 S[mi]的关键码与目标关键码 key 做比较。比较的结果不外乎三种可能：

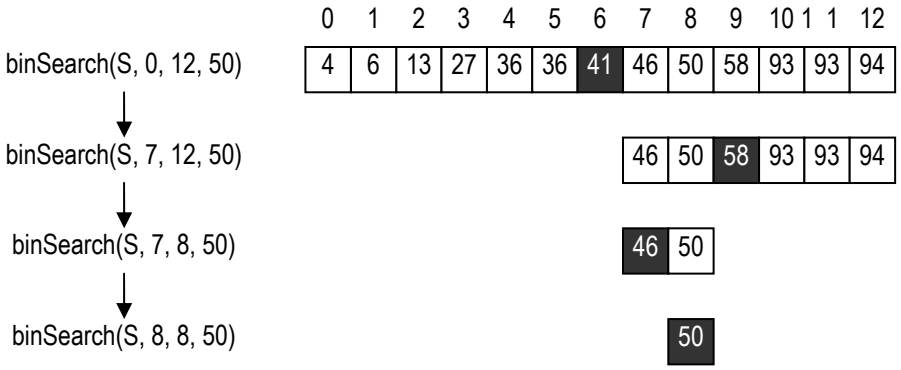
1. S[mi].getKey() = key。此时该条目命中。倘若只需找到一个这样的条目，查找算法即告完成。
2. S[mi].getKey() > key。这说明 mi 到 hi 之间的条目均可排除，查找范围可以缩小至 S[lo..mi-1]。
3. S[mi].getKey() < key。这说明 lo 到 mi 之间的条目均可排除，查找范围可以缩小至 S[mi+1..hi]。

上述过程可以描述为 算法六.1：

```
算法：binSearch(S, lo, hi, key)
输入：有序查找表S，非负整数lo和hi，关键码key

输出：在S中的区间S[lo..hi]中，查找关键码为key的条目

说明：首次调用形式为binSearch(S, 0, n-1, key)，其中n = |S|
{
  若lo > hi，则返回“未找到”; //递归基：查找范围缩小至空
  mi = (lo+hi)/2; //区间的中间位置
  若key > S[mi].key，则返回binSearch(S, mi+1, hi, key); //在右半区间中继续查找
  若key < S[mi].key，则返回binSearch(S, lo, mi-1, key); //在左半区间中继续查找
  否则，返回mi; //命中，查找成功
}
```

算法六.1 有序查找表的二分查找算法

图六.9 给出了二分查找的一个实例：



图六.9 有序查找表的二分查找过程

这一算法不断二分式地排除候选条目，故称作“二分查找”（Binary search），或“折半查找”。其正确性是由以下事实保证的：

观察结论六.2 在对有序查找表的二分查找过程中，每次需要深入左（右）半区间继续查找时，当前接受比较的条目不会命中，而且被忽略的右（左）半区间也必然不含目标关键码。

6.4.3 有序词典的 ADT 描述

从 ADT 的角度，有序词典可以看作是无序词典的扩充，也就是说，只需在无序词典 ADT 的基础上再增加以下操作：

表六.6 有序词典ADT支持的操作

操作方法	功能描述
first():	若词典非空，则返回其中关键码最小的条目 否则，返回 null 输入：无 输出：条目对象
last():	若词典非空，则返回其中关键码最大的条目 否则，返回 null 输入：无 输出：条目对象
successors(key):	返回由关键码不小于 key 的条目依非降序组成的迭代器 输入：一个关键码 输出：条目迭代器
predecessors(key):	返回由关键码不大于 key 的条目依非升序组成的迭代器 输入：一个关键码 输出：条目迭代器

需要指出的是，与无序词典的情况类似，尽管 Java 也提供了一个有序映射结构的接口 java.util.SortedMap，但该类对象中同样禁止多个条目拥有相同的关键码。

6.4.4 有序词典的 Java 接口

基于无序词典的接口，在扩充了以上方法之后，即可得到如 代码六.9 所示的有序词典接口：

```
/*
 * 有序词典接口
 */

package dsa;

public interface SortedDictionary extends Dictionary {
//若词典非空，则返回其中关键码最小的条目；否则，返回null
  public Entry first();

//若词典非空，则返回其中关键码最大的条目；否则，返回null
```

```
  public Entry last();

//返回由关键码不小于key的条目依非降序组成的迭代器
  public Iterator successors(Object key);

//返回由关键码不大于key的条目依非升序组成的迭代器
  public Iterator predecessors(Object key);
}
```

代码六.9 有序词典的Java接口

6.4.5 基于有序查找表实现有序词典

这里利用可扩充数组实现有序查找表，并在此基础上利用二分查找算法，实现如 代码六.10 所示的有序词典结构：

```
/*
 * 基于有序查找表实现的有序词典
 */

package dsa;

public class SortedDictionary_ExtArray implements SortedDictionary {
  Vector  S;//有序查找表
  Comparator C;//比较器

//默认构造方法
  public SortedDictionary_ExtArray()
  { this(new ComparatorDefault()); }

//构造方法
  public SortedDictionary_ExtArray(Comparator comp)
  { S = new Vector_ExtArray();  C = comp; }

/*************************** 辅助方法 ***************************/
//二分查找
//返回值可能是命中元素的秩，也可能是key可以插入的秩
//具体如何，需要进一步检查
//不变性：若将key按照返回的秩插入有序向量，向量依然有序
  private static int binSearch(Vector s, Comparator c, Object key, int lo, int hi) {
    if (lo > hi) return lo;//递归基，查找失败

    int mi = (lo + hi)>>1;//取中值
    Entry e = (Entry)s.getAtRank(mi);//居中的条目
    int flag = c.compare(key, e.getKey());//比较关键码
```

```
    if (flag < 0)       return binSearch(s, c, key, lo, mi-1);//转向左半区间
    else if (flag > 0)  return binSearch(s, c, key, mi+1, hi);//转向右半区间
    else                return mi;//命中
  }

/*************************** 无序词典ADT方法 ***************************/
//查询词典结构当前的规模
  public int getSize()
  { return S.getSize(); }

//判断词典结构是否为空
  public boolean isEmpty()
  { return S.isEmpty(); }

//若词典中存在以key为关键码的条目，则返回其中的一个条目；否则，返回null
  public Entry find(Object key) {
    int k = binSearch(S, C, key, 0, S.getSize()-1);//查找关键码为key的条目

    if (0 > k || k >= S.getSize() || (0 != C.compare(key,
((Entry)S.getAtRank(k)).getKey())))

      return null;//若这样的条目不存在，则返回失败标志
    return (Entry) S.getAtRank(k);
  }

//返回由关键码为key的条目组成的迭代器
  public Iterator findAll(Object key) {
    List L = new List_DLNode();//创建一个链表L

    int k = binSearch(S, C, key, 0, S.getSize()-1);//查找关键码为key的条目
    if (0 > k || k >= S.getSize() || (0 != C.compare(key,
((Entry)S.getAtRank(k)).getKey())))
      return new IteratorElement(L);//若这样的条目不存在，则返回空迭代器

    L.insertFirst(S.getAtRank(k));//将e插入L中

    int lo = k;//从S[k-1]开始
    while (0 <= --lo) {//不断向前搜索
      if (0 != C.compare(key, ((Entry)S.getAtRank(lo)).getKey())) break;//直到第一个不
命中的条目
      L.insertFirst(S.getAtRank(lo));//将命中的条目插入L中
    }

    int hi = k;//从S[k+1]开始
    while (++hi < S.getSize()) {//不断向后搜索
      if (0 != C.compare(key, ((Entry)S.getAtRank(hi)).getKey())) break;//直到第一个不
命中的条目
```

```
      L.insertLast(S.getAtRank(hi));//将命中的条目插入L中
    }

    return new IteratorElement(L);//由L创建迭代器，返回之
  }

//插入条目(key, value)，并返回该条目
  public Entry insert(Object key, Object value) {
    Entry e = new EntryDefault(key, value);//创建新条目

  //若词典为空，则直接插入新元素
    if (S.isEmpty())  return (Entry) S.insertAtRank(0, e);

  //通过二分查找，确定可插入位置
  //请读者自己检查：即便key在S中为最小或最大，都可以正常插入
    return (Entry) S.insertAtRank(binSearch(S, C, key, 0, S.getSize()-1), e);
  }

//若词典中存在以key为关键码的条目，则将摘除其中的一个并返回；否则，返回null
  public Entry remove(Object key) {

    int k = binSearch(S, C, key, 0, S.getSize()-1);//查找关键码为key的条目
    if (0 > k || k >= S.getSize() || (0 != C.compare(key,
((Entry)S.getAtRank(k)).getKey())))
      return null;//若这样的条目不存在，则返回失败标志
    return (Entry) S.removeAtRank(k);
  }

//返回词典中所有条目的一个迭代器
  public Iterator entries() {
    List L = new List_DLNode();
    for (int i=0; i<S.getSize(); i++)
      L.insertLast(S.getAtRank(i));
    return new IteratorElement(L);//直接利用List接口的方法生成元素迭代器
  }

/***************************** 有序词典ADT方法 *****************************/
//若词典非空，则返回其中关键码最小的条目；否则，返回null
  public Entry first()
  { return (S.isEmpty()) ? null : (Entry) S.getAtRank(0); }

//若词典非空，则返回其中关键码最大的条目；否则，返回null
  public Entry last()
  { return (S.isEmpty()) ? null : (Entry) S.getAtRank(S.getSize()-1); }

//返回由关键码不小于key的条目依非降序组成的迭代器
  public Iterator successors(Object key) {
```

```
    List L = new List_DLNode();//创建一个链表L

    int k = binSearch(S, C, key, 0, S.getSize()-1);//查找关键码为key的条目
    if (0 > k || k >= S.getSize() || (0 != C.compare(key,
((Entry)S.getAtRank(k)).getKey())))
      return new IteratorElement(L);//若这样的条目不存在，则返回空迭代器

    while (0 <= --k)//从S[k-1]开始向前搜索，直至符合要求的、秩最小的元素
      if (0 != C.compare(key, ((Entry)S.getAtRank(k)).getKey()))  break;
    while (S.getSize() > ++k)//将后继的所有元素依次
      L.insertLast(S.getAtRank(k));//插入L中

    return new IteratorElement(L);//由L创建迭代器，返回之
  }

//返回由关键码不大于key的条目依非升序组成的迭代器
  public Iterator predecessors(Object key) {
    List L = new List_DLNode();//创建一个链表L

    int k = binSearch(S, C, key, 0, S.getSize()-1);//查找关键码为key的条目

    if (0 > k || k >= S.getSize() || (0 != C.compare(key,
((Entry)S.getAtRank(k)).getKey())))

      return new IteratorElement(L);//若这样的条目不存在，则返回空迭代器

    while (S.getSize() > ++k)//从S[k-1]开始向后搜索，直至符合要求的、秩最大的元素
      if (0 != C.compare(key, ((Entry)S.getAtRank(k)).getKey()))  break;
    while (0 <= --k)//将前驱的所有元素依次
      L.insertLast(S.getAtRank(k));//插入L中

    return new IteratorElement(L);//由L创建迭代器，返回之
  }
}
```

代码六.10 基于有序查找表实现的有序词典

第七章

查找树

在 第六章中，我们介绍了词典ADT的几种实现。以其中最常用的有序词典ADT为例，若采用无序列表来实现，则尽管insert()操作可以在常数时间内完成，但find()操作在最坏情况下却需要线性时间，故remove()操作也需要$\mathcal{O}(n)$时间；而findAll()操作则需要$\Theta(n)$时间。若采用散列表来实现，则尽管其平均性能不错，但在最坏情况下find()操作仍然需要$\mathcal{O}(n)$时间。第 6.4.5 节还给出了基于有序表的实现，利用二分查找策略，可以在$\mathcal{O}(\log n)$时间内完成一次find()操作；然而，由于有序表本身必须借助（定长或可扩充）数组实现，所以insert()和remove()操作依然需要$\mathcal{O}(n)$的时间。

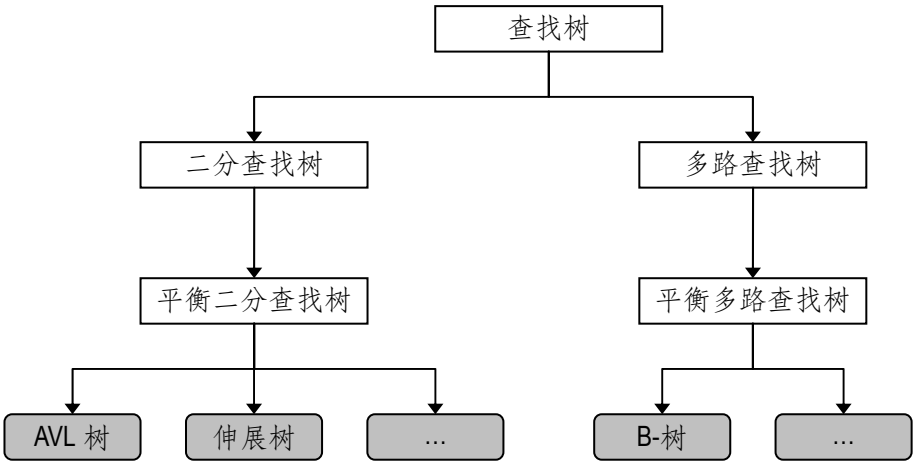
那么，究竟有没有可能以更高的效率实现有序词典 ADT 呢？如果可能，应该采用什么样的数据结构呢？这些正是本章将要回答的问题。

在第 §7.1 节中，我们将首先对二分查找的概念进行推广，并相应地定义二分查找树结构，进而基于这一结构实现词典ADT——这样，在同样只使用$\mathcal{O}(n)$空间的前提下，有序词典ADT所定义的各种操作都可以很方便地得到实现。不过，就最坏情况的时间复杂度而言，这一实现与此前的实现并没有本质的提高。尽管如此，这部分内容依然十分重要，因为这种基于树形结构的思想正是本章后续内容的基础和出发点。

接下来的第 §7.2 节将继续贯彻这一构思，提出平衡二分查找树的概念，引入AVL树这一典型的平衡二分查找树，并基于AVL树实现有序词典ADT。凭借精巧的平衡调整算法，即使是在最坏情况下，对AVL树结构的每一次访问或更新操作都可以在$\mathcal{O}(\log n)$时间内完成。至此，前面所提出的问题得到了圆满的解答。

针对实际应用问题的不同特点，平衡二分查找树有众多的变型。比如在第 §7.3 节中，我们将针对实际应用中普遍存在的数据访问的局部性特点，提出“最常用者优先”的原则，并基于这一策略引入并实现伸展树结构。由于其实现的简洁性，伸展树结构的应用范围极广。尽管就最坏情况的复杂度而言，这一结构的性能并不好，然而正如我们将要看到的，就分摊复杂度的意义而言，伸展树仍然可以在$\mathcal{O}(\log n)$的时间内完成每一次访问或更新操作，这一点非常重要。

第 §7.4 节将对平衡二分查找树进行推广，从而引入另一种重要变型——平衡多路查找树。该节还将详细介绍平衡多路查找树的一种典型结构——B-树。采用这类结构的目的，旨在弥合内、外存在访问速度上的巨大差异。通过分析我们将会看到，B-树的确可以更高效地解决涉及磁盘之类外存储器操作的实际问题。



图七.1 查找树的分类

围绕查找树这一主题，本章将要介绍若干种相关的数据结构，它们之间的关系可以表示为 图七.1。

本章依然沿用此前采用的惯例：用返回值 null 表示查找失败。

§7.1 二分查找树

基于列表和向量结构，第六章分别介绍了词典ADT的两种典型实现方式。遗憾的是，就时间复杂度而言，这两种方式都不能兼顾所有操作的效率，因此无法令人满意。本节将引入二分查找树结构，旨在将以上两种方式的优点结合起来，最终高效地实现有序词典ADT中的所有操作。

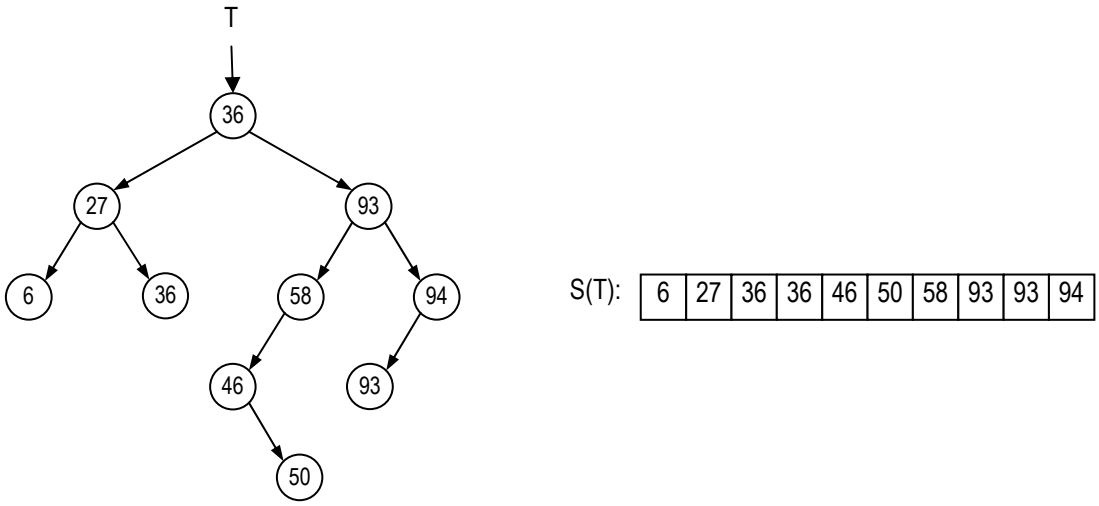
7.1.1 定义

定义七.1 所谓的一棵二分查找树（Binary search tree）T，要么是一棵空树，要么是以 r = (key, value) 为根节点的二叉树，而且其左、右子树都是二分查找树，同时

❶ 在 r 的左子树中，所有节点（如果存在的话）的关键码均不大于 key；

❷ 在 r 的右子树中，所有节点（如果存在的话）的关键码均不小于 key。

这里再次强调，为了与有序词典结构的定义一致，这里并不要求二分查找树中各节点的关键码互异。



图七.2 二分查找树T的中序遍历序列S(T)必然按非降序排列

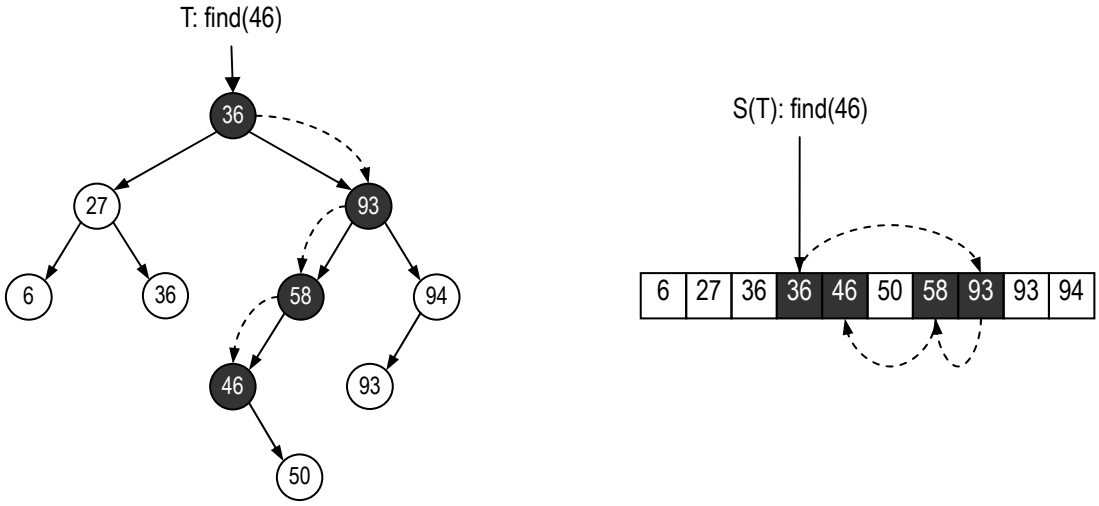
如 图七.2 所示，通过父节点、孩子节点之间的链接关系，二分查找树根据关键码的大小，将所有条目组织为一个树状层次结构。实际上，有序词典中条目关键码之间的全序关系，已经完全“蕴含”于这样的一个结构之中——只要对这棵树做一次中序遍历（第 4.5.7 节），即可得到所有条目的一个有序排列。

以下事实请读者自行证明：

引理七.1 二叉树 T 为二分查找树，当且仅当其中序遍历序列是单调非降的。

7.1.2 查找算法

二分查找树查找算法的构思是：从根节点开始，以递归的形式不断缩小查找范围，直到发现目标条目（查找成功）或查找范围缩小至空树（查找失败）。



图七.3 二分查找树的查找过程

图七.3 给出了针对关键码 46[1]的一次具体查找的过程。

在递归的每一层，一旦发现当前节点为 null，则说明查找范围已经为空，查找失败；否则，我们将目标关键码与当前节点的关键码做对比。与有序查找表的二分查找过程类似，这里的比较结果也不外乎三种可能：

1. 目标关键码更大（比如在节点 36 处）。此时，可以深入右子树中做递归查找；
2. 目标关键码更小（比如在节点 93 和 58 处）。此时，可以深入左子树中做递归查找；
3. 相等（比如在节点 46 处）。此时算法以“查找成功”结束。

基于上述策略的查找过程可以描述为 算法七.1，具体的实现请参照 代码七.2 中的binSearch()方法。

```
算法：binSearch(v, key)
输入：二叉树中的节点v，一个关键码key
输出：在以v为根节点的（子）树中，找出关键码为key的节点；若不存在这样的节点，则返回最后被访问的节点
要求：首次调用时，v为树根节点
{
  置当前节点u = v;
  不断地迭代 {
    将当前节点u与目标关键码key做比较;
    若目标关键码更小，则
      若u有左孩子，则令u = u.lChild;
      否则（查找失败），直接返回u;
    否则，若若目标关键码更大，则
      若u有右孩子，则令u = u.rChild;
      否则（查找失败），直接返回u;
    否则（查找命中），直接返回u;
  }
}
```

算法七.1 二分查找树的查找算法

■ 正确性

实际上，根据 引理七.1，只要将二分查找树T中的节点与其中序遍历序列S(T)中的元素一一对应起来便可看出：实质上，对T的二分查找几乎等同于第 6.4.2 节所介绍的针对有序查找表的二分查找，如 图七.3 所示。二者的差别仅仅在于：这里在确定每次递归的方向时，并不是严格地在查找范围内取秩居中的元素作为比较对象，而是以当前节点的左或右孩子（如果存在的话）作为比较对象。

与第 6.4.2 节对有序查找表二分查找算法的分析类似，二分查找树查找算法的正确性也可以由以下事实保证：

[1] 在上下文不致发生歧义的前提下，为使叙述简明，我们将直接用其关键码来指称查找树中对应的节点。

观察结论七.1 在 binSearch ()算法中每次深入左（右）子树时，被忽略的右（左）子树必然不含目标节点。

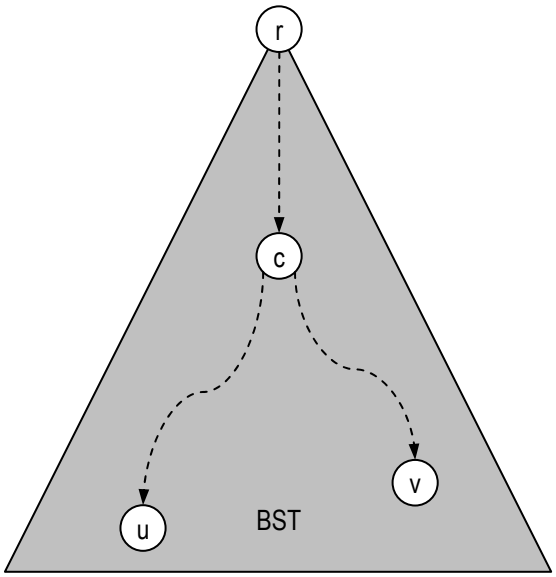
■ 确定性

我们曾经强调过，这里定义的二分查找树允许多个节点拥有相等的关键码。既然如此，binSearch()算法将会返会其中的哪一个呢？

引理七.2 在任一二分查找树 T 中，若至少存在一个关键码为 key 的节点，则这些节点中深度最小者必然唯一，而 binSearch()算法找出的正是这一节点。

〖证明〗

首先证明：在关键码同为 key 的节点（若存在）中，深度最小者必唯一。



图七.4 对于二分查找树中关键码相同的任意一对节点u和v，其最低共同祖先c = lca(u, v)的关键码必与它们相同

否则，任取深度最小的两个节点u和v，令c为它们的最低共同祖先（定义四.12）。显然，$u \neq c \neq v$，而且u和v不可能处于c的同一侧子树。不失一般性，设u、v分别属于c的左、右子树。于是便有：

$$key = key(u) \leq key(c) \leq key(v) = key$$

即

$$key(c) = key$$

这与“u、v 是关键码为 key 的深度最小节点”矛盾。

接下来，我们注意到以下事实：在对二分查找树的查找过程中，接受比较的各个节点的深度比然不断递增。因此，在关键码为 key 的所有节点中，唯一的那个深度最小的节点必然首先接收比较，并使得算法以成功告终。 ❑

■ 效率

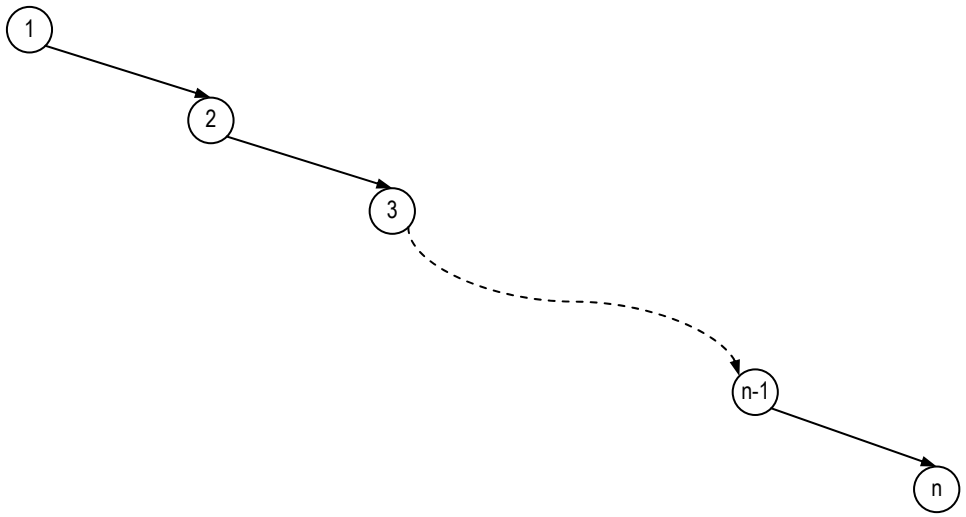
不难看出，在二分查找树的每一层次上，上述算法至多访问一个节点，而且每次只需常数时间，故其总体运行时间取决于被访问节点的最大深度。在最好情况下，目标条目恰好出现在树根（或附近）处，此时只需 $O(1)$时间。然而，要是查找失败，则算法必然终止于某一叶子节点处，此时一般需要更多的时间。由此我们可以得出如下结论：

引理七.3 在二分查找树中查找一个节点需要 $O(h)$时间，其中 h 为目标节点的深度（查找成功时），或者是二分查找树的高度（查找失败时）。

那么，h 等于多少？最小是多少？最大又是多少？

推论四.3 早已指出：由n个节点构成的二叉树，树高至少是$\Omega(\log n)$。反过来，由完全二叉树（第 4.1.8 节和第 §4.6 节）的例子可以看出，这一下界是紧的——也就是说，n个节点的确可以构成高度为$O(\log n)$的二分查找树。在这种情况下，每次查找操作所需的时间都不会超过$O(\log n)$。

在第 §7.2 节中，我们将给出所谓“平衡二分查找树”的严格定义。这类二叉树的一个重要性质，就是树高不超过$O(\log n)$。因此，对这类查找树的每一查找操作都可以在$O(\log n)$的时间内完成。



图七.5 二分查找树可能退化为有序列表

然而，一般的二分查找树对其高度没有任何限制，在最坏情况下，由n个节点构成的二分查找树的高度可以达到$\Theta(n)$。如 图七.5 所示，当所有节点都没有左孩子时，就是这样的一种极端情况。此时，查找操作在最坏情况下可能需要线性的时间——这并不奇怪，因为实际上，这样的一棵“二分”查找树已经退化为了一个不折不扣的有序查找表。

7.1.3 完全查找算法

■ findAllNodes() 算法

我们希望通过findAll(key)方法，得到由关键码为key的所有节点组成的一个迭代器。为此，我们可以利用如 算法七.2 所示的算法findAllNodes(v, key, list)，在以v为根节点的（子）树中，找出关键

码为key的所有节点，将它们加入到列表list中。这样，只要调用findAllNodes(root, key, list)，即可将整棵树中关键码为key的所有节点组织为列表list。最后，利用列表自身提供的elements()方法，即可得到对应的迭代器。

```
算法：findAllNodes(v, key, list)
输入：关键码key
输出：由关键码为key的所有节点组成的一个迭代器
{
  if (v为空) return;//递归基：空树情况
  BSTNode e = binSearch(v, key);//在以v为根节点的子树中，查找Sv(key)的最低共同祖先
  if (null != e) {//若找到了一个这样的节点，则
    findAllNodes(e.getLChild, key, list);//递归查找其左子树
    list.insertLast(e);//将节点e插入列表
    findAllNodes(e.getRChild, key, list);//递归查找其右子树
  }
}
```

算法七.2 完全查找算法

该算法的具体实现请参见 代码七.2 中的findAll()方法。

■ 正确性

为了说明 findAllNodes()算法的正确性，我们来考察二分查找树 T 中关键码为 key 的所有节点，由这些节点组成的集合记作 S^T(key)。

根据 引理七.2，可以得出如下推论：

推论七.1 S^T(key)中深度最小的节点必唯一，它就是 S^T(key)中所有节点的最低共同祖先，而且在 findAllNodes()算法中调用 binSearch()算法之后，即可找到该节点。

因此，从这一最低共同祖先节点出发继续深入查找时，被忽略的节点必然不可能是候选节点。也就是说，S^T(key)中的每一节点都不会遗漏掉。

7.1.4 插入算法

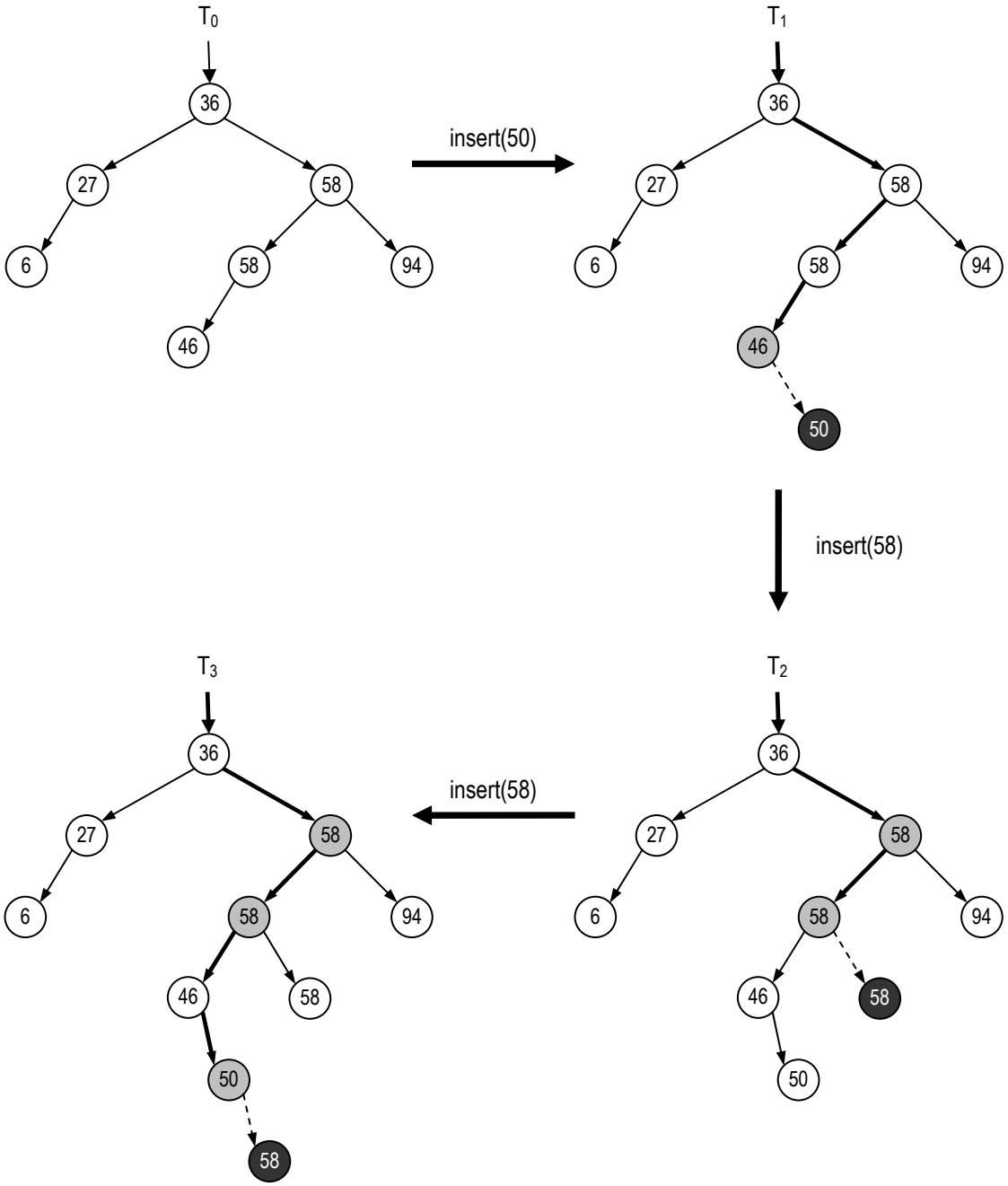
■ 算法

为了在二分查找树中插入一个节点，我们需要根据其关键码key，利用查找算法binSearch()确定插入的位置及方向，然后才将新节点作为叶子插入。具体过程可以描述为 算法七.3：

```
算法：insert(v, key, value)
输入：以v为根节点的子树，关键码key以及数据value
输出：将条目(key, value)插入二分查找树中，并返回该条目
{
```

```
  若当前的树为空，则生成并返回一棵包含单节点(key, value)的二分查找树;


  p = 树根v;
  while (true) {
    调用binSearch(p, key)，在以p为根节点的子树中查找关键码为key的最高节点
    若key ≠ p.key，则根据二者的大小关系将新节点作为p的右或左孩子插入，并返回新节点
    若key = p.key，则有两种可能：
    ① 若p没有左或右孩子，则将新节点作为p的左或右孩子插入，并返回新节点;
    ② 若p已有两个孩子，则取p = p.lChild;
  }
}
```

算法七.3　二分查找树的节点插入算法



图七.6 在二分查找树T_0中依次插入节点50、58和58：灰色节点为每次调用binSearch()算法时最后访问到的节点，粗边表示查找过程所经过的路径；黑色节点为新插入的节点，虚边表示新生成的树边

图七.6 给出了节点插入过程的一个实例。首先插入的是关键码 50：查找binSearch(36, 50)终止于节点 46，随后新节点 50 作为它的右孩子插入，结果如T_1所示。接下来插入的是关键码 58：查找binSearch(36, 58)终止于第 1 层的节点 58；由于该节点左、右孩子皆存在，故从其左孩子出发继续查找binSearch(58, 58)，并终止于第 2 层的节点 58；该节点只有左孩子，故新节点可以作为其右孩子插入，结果如T_2所示。最后，再次插入一个关键码 58：查找binSearch(36, 58) 先后终止于 46、58 和 50，由于 58 > 50，故新节点 58 作为 50 的右孩子插入，结果如T_3所示。

该算法的具体实现请参见 代码七.2 中的insert()方法。

■ 效率

由上可见，为了插入节点v，binSearch()查找算法需要访问的节点，都分布在由根节点通往v的路径上，而且这些节点恰好分别被访问一次。于是根据 推论四.1，可得如下结论：

引理七.4 在二分查找树中插入一个节点需要 O(h)时间，其中 h 为被插入节点的深度。

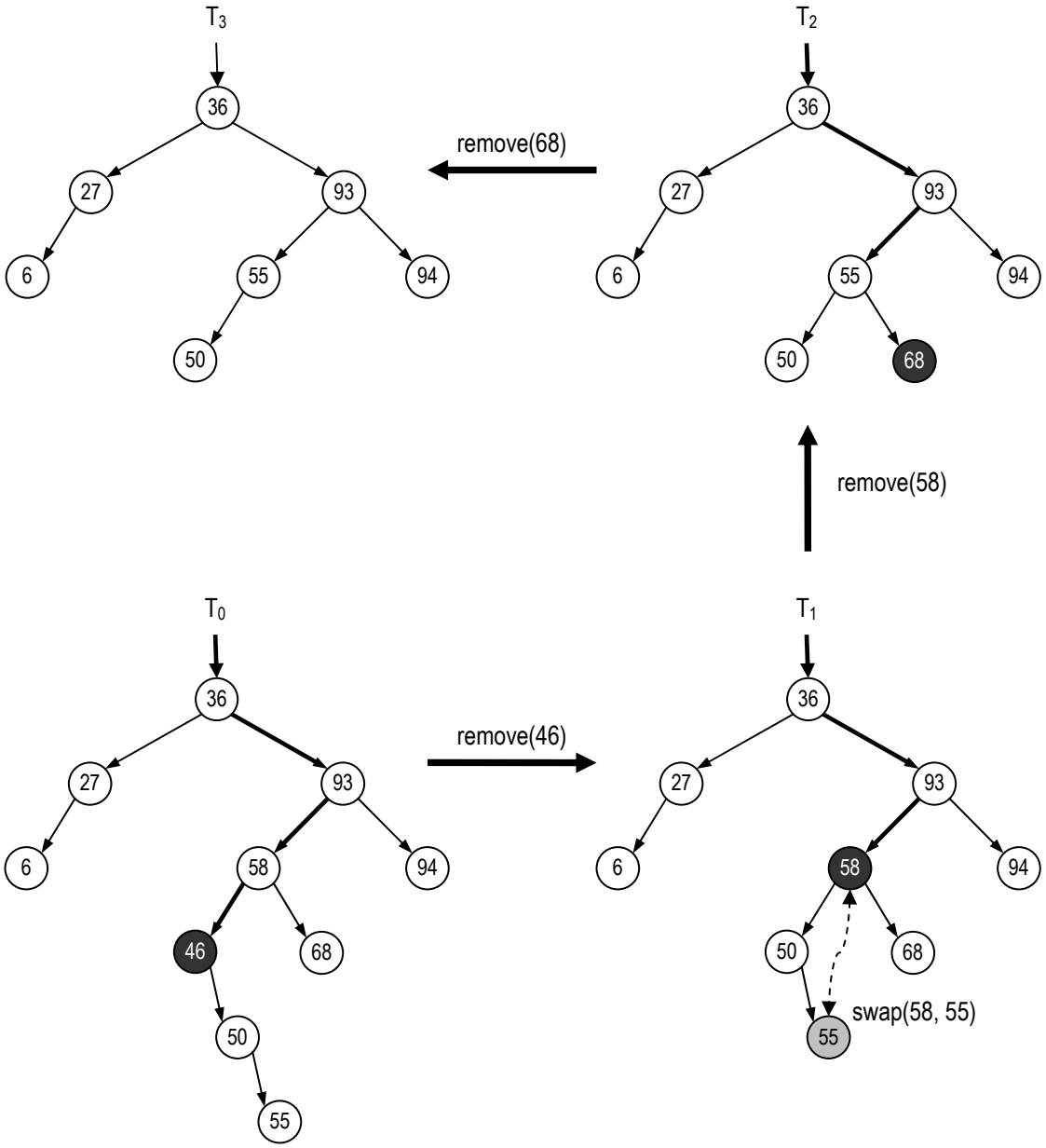
7.1.5 删除算法

■ 算法

为了从二分查找树中删除关键码为key的节点，我们首先也需要通过算法binSearch()判断树中是否的确存在这样的节点；如果存在，我们要确定其位置，然后才能将其摘除。这一过程可以描述为 算法七.4：

```
算法：remove(r, key)
输入：子树r，关键码key
输出：若在以r为根节点的子树中存在关键码为key的节点，则删除它，并返回其中存放的条目
{
  调用binSearch(r, key)，在子树r中查找关键码为key的最高节点
  若key ≠ v.key，则说明目标节不存在，故返回null;

  若v没有左孩子，则
    摘除v，代之以v的右孩子;
  否则 {
    在v的左子树中找出其直接前驱w; //请注意，w必然没有右孩子
    将v与w交换位置;
    摘除v，代之以v的左孩子;
  }
}
```

算法七.4 二分查找树的节点删除算法



图七.7 在二分查找树T_0中依次删除节点46、58和68：黑色节点为待删除的节点；灰色节点表示待删除节点的直接前驱，虚边表示节点位置的交换

图七.7 给出了节点删除过程的一个实例。首先，在T_0中删除节点 46：经查找确定其位置后，发现节点 46 没有左孩子，于是直接摘除该节点并代之以其右孩子（节点 50）后，结果如T_1所示。接下来，在T_1中删除节点 58：经查找确定其位置后，发现节点 58 的左、右孩子都存在，于是利用第 4.5.8 节的相关算法，在左子树 50 中找出节点 58 的直接前驱 55，将节点 58 与 55 交换位置后，再将 58 摘除，得到T_2。最后，在T_2中删除节点 68：节点 68 是叶子，属于无左孩子情况的一个特例——请读者自己分析其删除的过程。

节点删除算法的具体实现，参见 代码七.2 中的remove()方法。

■ 效率

与对节点插入算法的分析同理，可以得出以下结论：

引理七.5 在二分查找树中删除一个节点需要 $O(h)$时间，其中 h 为被删除节点的深度。

需要特别强调的是，若在摘除之前曾经将节点与其直接前驱交换过位置，则这里 h 应该是指直接前驱原先的位置。

7.1.6 二分查找树节点类的实现

按照 BinTreePosition 和 Entry 接口规范，在 BinTreeNode 类的基础上，可以实现如下的 BSTreeNode 类：

```
/*
 * 基于链表实现的BST节点类
 */

package dsa;

public class BSTreeNode extends BinTreeNode implements BinTreePosition, Entry {
/*************************** 构造方法 ***************************/
  public BSTreeNode()
  { super(); }

  public BSTreeNode(
    Object e,//节点内容
    BinTreePosition p,//父节点
    boolean asLChild,//是否作为父节点的左孩子
    BinTreePosition l,//左孩子
    BinTreePosition r)//右孩子
  { super(e, p, asLChild, l, r); }

/*************************** 实现Entry接口的方法 ***************************/
//返回当前节点的关键码
  public Object getKey()
  { return ((Entry)getElem()).getKey(); }

//修改条目的关键码，返回此前存放的关键码
  public Object setKey(Object k)
  { return ((Entry)getElem()).setKey(k); }

//取条目的数据对象
  public Object getValue()
  { return ((Entry)getElem()).getValue(); }

//修改条目的数据对象，返回此前存放的数据对象
  public Object setValue(Object v)
  { return ((Entry)getElem()).setValue(v); }
}
```

代码七.1 二分查找树节点类的实现

7.1.7 二分查找树类的实现

按照 Dictionary 接口规范，在 BinTree_LinkedList 类的基础上，可以实现如下的 BSTree 类：

```
/*
 * 基于链表式BST实现的词典结构
 * 基于BinTree进行扩充
 */

package dsa;

public class BSTree extends BinTree_LinkedList implements Dictionary {
/*************************** 实例变量 ***************************/
  protected Comparator C;//比较器
  protected BinTreePosition lastV;//最后操作的节点，以便AVL树、伸展树重平衡

/*************************** 构造方法 ***************************/
  public BSTree()
  { this(null, new ComparatorDefault()); }

  public BSTree(BinTreePosition r)
  { this(r, new ComparatorDefault()); }

  public BSTree(BinTreePosition r, Comparator c)
  { root = r; C = c; }

/*************************** 词典方法 ***************************/
//若词典中存在以key为关键码的条目，则返回其中的一个条目；否则，返回null
  public Entry find(Object key) {
    if (isEmpty())  return null;
    BSTreeNode u = binSearch((BSTreeNode)root, key, C);
    return (0 == C.compare(key, u.getKey())) ? (Entry)u.getElem() : null;
  }

//返回由关键码为key的条目组成的迭代器
  public Iterator findAll(Object key) {
    List s = new List_DLNode();
    finAllNodes((BSTreeNode)root, key, s, C);
    return s.elements();
  }

//插入条目(key, value)，并返回该条目
//lastV指示被插入的节点
  public Entry insert(Object key, Object value) {
    Entry e = new EntryDefault(key, value);//创建新的元素

    if (isEmpty()) {//插入根节点的情况
      lastV = root = new BSTreeNode(e, null, true, null, null);//插入新节点
    } else {//插入一般节点的情况
```

```
      BSTreeNode p = (BSTreeNode)root;//从根节点开始，查找可插入位置

      boolean asLeftChild;//表示新节点是否作为p的左孩子插入

      while(true) {//不断地
        p = binSearch(p, key, C);//查找关键码为key的节点，直至
        if (C.compare(key, p.getKey()) < 0)//查找失败于无左孩子节点，或
          { asLeftChild = true; break; }
        else if (C.compare(key, p.getKey()) > 0)//查找失败无右孩子节点，或
          { asLeftChild = false; break; }
        else if (!p.hasLChild())//查找成功，且可作为左孩子插入，或
          { asLeftChild = true; break; }
        else if (!p.hasRChild())//查找成功，且可作为右孩子插入，或
          { asLeftChild = false; break; }
        else//否则
          p = (BSTreeNode)p.getLChild();//在左子树中继续查找（当然，在右子树中查找亦可）
      }//至此，新节点可以作为p的孩子插入，插入的方向由childType确定
      lastV = new BSTreeNode(e, p, asLeftChild, null, null);//插入新节点
    }//else

    return e;
  }

//若词典中存在以key为关键码的条目，则摘除这样的一个节点，并返回其中存放的条目；否则，返回null
//lastV指示被删除节点的父亲
  public Entry remove(Object key) {
    if (isEmpty())  return null;//空树

    BinTreePosition v = binSearch((BSTreeNode)root, key, C);//查找
    if (0 != C.compare(key, ((BSTreeNode)v).getKey()))  return null;//若查找失败，则返
回null

  //至此查找必成功，v为待删除节点
    if (v.hasLChild()) {//若v的左子树非空，则
      BinTreePosition w = v.getPrev();//在v的左子树中找出其直接前驱w
      w.setElem(v.setElem(w.getElem()));//交换v和u的数据对象
      v = w;//这样，相当于删除w
    }
    //至此，v至多只有一个孩子
    //下面，删除v，代之以其孩子
    lastV = v.getParent();//取待删除节点v的父亲
    BinTreePosition u = v.hasLChild() ? v.getLChild() : v.getRChild();//取v的孩子u
    if (null == lastV)//若v恰为树根
      { if (null != u) u.secede();  root = u; }//将u作为树根
    else {//否则
```

```
      if (v.isLChild())//若v是p的左孩子，则
        { v.secede(); lastV.attachL(u); }//摘出v，将u作为p的左孩子

      else//否则

        { v.secede(); lastV.attachR(u); }//摘出v，将u作为p的右孩子
    }
    return (Entry) v.getElem();//返回被删除节点中存放的的元素
  }

//返回词典中所有条目的一个迭代器
  public Iterator entries() {
    List list = new List_DLNode();
    concatenate(list, (BSTreeNode)root);
    return list.elements();
  }

/**************************** 辅助方法 ****************************/
//在以v为根的子树中查找关键码为key的节点（假设该子树不为空）
//  若找到，则返回该节点
//  否则，返回被访问的最后一个节点
//为了确定是否成功，上层方法需要再检查一次返回节点的关键码
  protected static BSTreeNode binSearch(BSTreeNode v, Object key, Comparator c) {
    BSTreeNode  u = v;//当前节点
    while (true) {//不断地
      int comp = c.compare(key, u.getKey());//将当前节点与目标关键码做比较
      if (comp < 0)//若目标关键码更小，则
        if (u.hasLChild())//若u有左孩子
          u = (BSTreeNode)u.getLChild();//递归查找左子树，或
        else
          return u;//终止于无左孩子节点
      else if (comp > 0)//若目标关键码更大，则
        if (u.hasRChild())//u有右孩子
          u = (BSTreeNode)u.getRChild();//递归查找右子树，或
        else
          return u;//终止于无右孩子节点
      else
        return u;//查找命中
    }
  }

//在以v为根节点的（子）树中，递归地找出关键码为key的所有节点
//这些节点被组织为一个列表（借此可以生成一个迭代器）
  protected static void finAllNodes(BSTreeNode v, Object k, List s, Comparator c) {
    if (null == v)  return;//递归基：空树
    int comp = c.compare(k, v.getKey());
```

```
    if (0 >= comp)  finAllNodes((BSTreeNode)v.getLChild(), k, s, c);//查找左子树
    if (0 == comp)  s.insertLast(v);//命中
    if (0 <= comp)  finAllNodes((BSTreeNode)v.getRChild(), k, s, c);//查找右子树
  }


//将v的所有后代节点（中存放的条目）组织为一个列表（借此可以生成一个迭代器）
  protected static void concatenate(List list, BSTreeNode v) {
    if (null == v)  return;
    concatenate(list, (BSTreeNode) v.getLChild());
    list.insertLast(v.getElem());
    concatenate(list, (BSTreeNode) v.getRChild());
  }
}
```

代码七.2 二分查找树结构的实现

7.1.8 二分查找树的平均性能

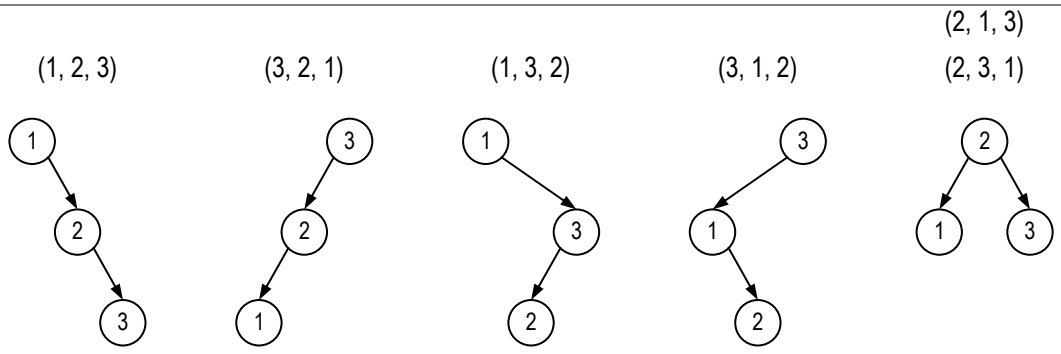
根据 引理七.3、引理七.4 和 引理七.5，对二分查找树的查找、插入和删除操作，在最坏情况下都需要$\mathcal{O}(n)$时间才能完成，就这一点而言，似乎与 第六章实现的词典结构没有区别。那么，就平均复杂度而言，二分查找树的性能又将如何呢？是否还是这个水平？

实际上，这里的“平均”一词的定义并不严格。下面，我们将针对两种随机统计的口径，给出二分查找树的平均性能。

- 随机生成

不妨考虑关键码互异的 n 个条目$\{e_1, e_2, \ldots, e_n\}$。对于这些条目的任一全排列$\sigma = (e_{i_1}, e_{i_2}, \ldots, e_{i_n})$，若从空树开始，依次调用 insert()算法将它们插入树中，都可以得到这 n 个条目的一棵二分查找树$T(\sigma)$。

定义七.2 与随机排列σ相对应的二分查找树 T(σ)，称作由σ生成的二分查找树。



图七.8 三个条目{1, 2, 3}的6个全排列各自生成的二分查找树

对于任意n个互异关键码，总共有n!个全排列。如果假定这n!个排列作为输入的概率均等，则只要将它们各自生成的二分查找树（如 图七.8 所示）的平均查找长度进行平均，所得到的总体平均查找长度将能反映二分查找树的平均查找性能。而下面这则定理明确地界定了这一复杂度。

定理七.1 由 n 个互异条目随机生成的 BST，平均查找长度为 $\mathcal{O}(\log n)$。

- 随机组成

细心的读者也许已经发现，同一组条目的不同排列所生成二分查找树有可能雷同。以 图七.8 为例，排列(2, 1, 3)与(2, 3, 1)生成的实际上是同一棵二分查找树，在做平均时，这棵树被统计了两次。从这个角度讲，对所有n!个随机排列进行平均，并不能反映二分查找树的平均查找性能。

根据以上分析，n 个条目所能构成的二分查找树的数目，将远远小于 n!。因此，我们可以假定树中的 n 个节点是给定的，然后在中序遍历次序保持一致的前提下，统计它们能够构成的二分查找树的数目。

观察结论七.2 在保持中序遍历次序的前提下，由 n 个互异节点构成的每棵二叉树，都是一棵二分查找树（称作由这些节点组成的一棵二分查找树）。

那么，这些拓扑结构互异的二分查找树共有多少棵呢？

定理七.2 由 n 个互异节点组成的二分查找树，总共有$\frac{(2n)!}{n!(n+1)!}$棵。

如 图七.8 所示，由三个节点所能组成的二分查找树，总共有$\frac{6!}{3!4!}$ = 5 棵。

如果假设这$\frac{(2n)!}{n!(n+1)!}$棵树出现的概率均等，则通过对它们各自的平均查找长度进行平均，也可以得到一个总体的平均指标：

定理七.3 由 n 个条目随机组成的 BST，平均查找长度为 $\mathcal{O}(\sqrt{n})$。

§7.2 AVL 树

7.2.1 平衡二分查找树

第 §7.1 节介绍的二分查找树，可以很好地实现有序词典结构，但在最坏情况下，二分查找树将退化为链表，此时的查找效率会降至$\mathcal{O}(n)$，其中n为树的规模。

根据 引理七.3，就最坏情况的复杂度而言，查找效率决定于二分查找树的高度，因此在节点数目固定的前提下，二分查找树的高度越低越好——从树的形态来看，也就是使树尽可能平衡。然而从另一方面看，推论四.3 告诉我们，不能指望二分查找树的性能优于$\Omega(\log n)$。

如果将高度不超过 $\mathcal{O}(n\log n)$的二分查找树称作“平衡的”，那么，能否始终保持二分查找树的平衡性呢？下面我们就来看看，如何通过等价变换，使失衡的二分查找树重新平衡。

7.2.2 等价二分查找树

定义七.3 中序遍历序列相同的任意两棵二叉树，称作相互“等价的”。

比如在 图七.9 中，T与S就是一对等价二叉树。

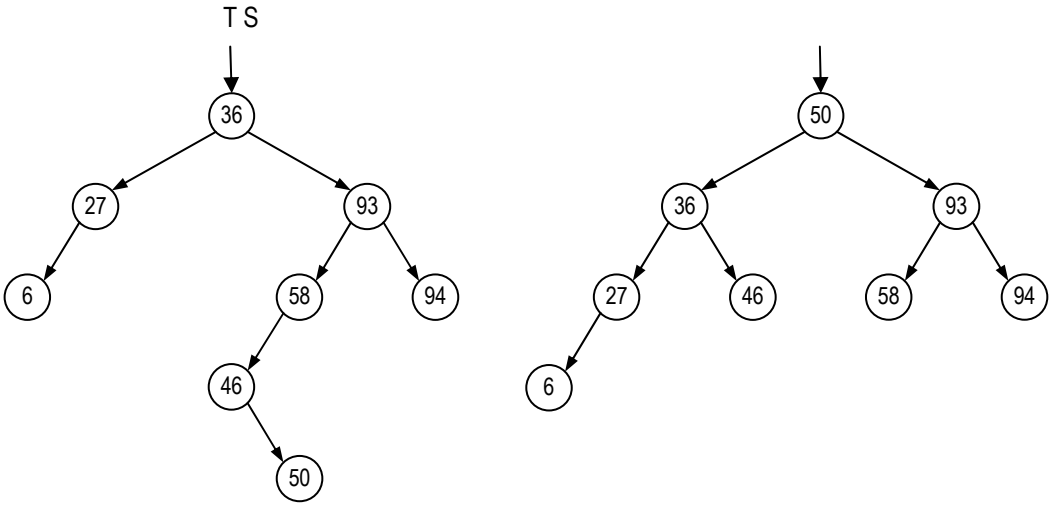
引理七.6 由 n 个节点组成的任何一棵二分查找树 T，都与某一棵高度不超过⌊$\log_2 n$⌋的二分查找树 S 等价。

〖证明〗

对 T 做中序遍历，设对应的中序遍历序列为$\sigma(T) = \{t_1, \ldots, t_n\}$。

考虑由 n 个节点组成的完全二叉树 S 及其中序遍历序列$\sigma(S) = \{s_1, \ldots, s_n\}$。

只要按照$s_i.key = t_i.key$的原则将T中各节点分别复制给S中对应的节点，则S必与T等价。此时，根据 引理四.1，S的高度为⌊log n⌋。 ❑



图七.9 由8个节点组成的二分查找树T，以及与之等价、高度为3的完全二叉树S：二者的中序遍历序列都是{6, 27, 36, 46, 50, 58, 93, 94}

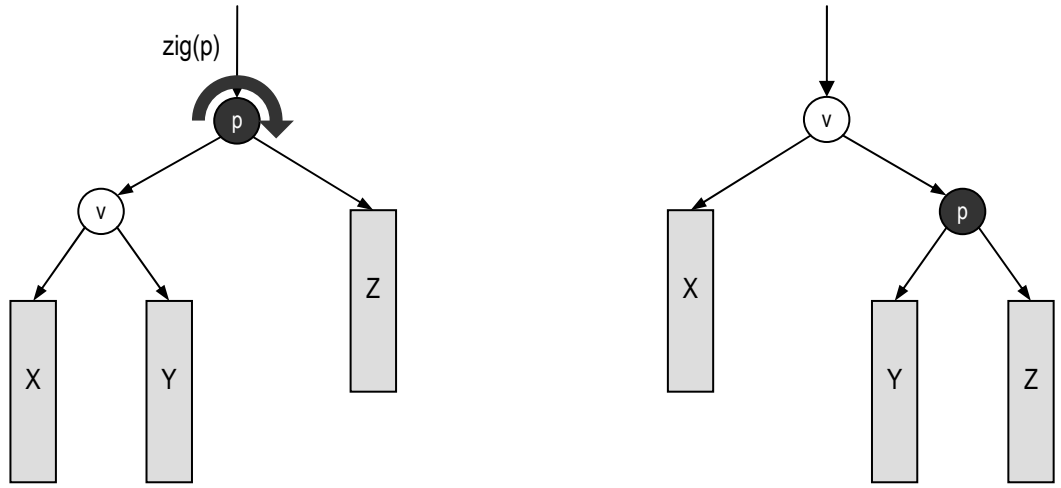
7.2.3 等价变换

引理七.6 告诉我们: 每一棵二分查找树都与某一棵平衡二分查找树相互等价，比如，必然与由原先n个节点组成的一棵完全二叉树等价。另外，这一引理的证明还直接给出了一种转换方法。然而不幸的是，这一转换方法的效率太低——仅考虑其中的中序遍历过程，就需要线性时间。能否更高效地实观这一平衡化过程呢？

一种快速的重平衡策略，就是只进行若干局部的节点旋转操作。

■ zig 旋转

如 图七.10(a)所示，假定节点v是节点p的左孩子，X和Y分别是v的左、右子树，Z为p的右子树。所谓围绕节点p的zig旋转操作（记作zig(p)），就是重新调整这些节点的位置，将p作为v的右孩子，将X作为v的左子树，将Y和Z分别作为p的左、右子树（图七.10(b)）。

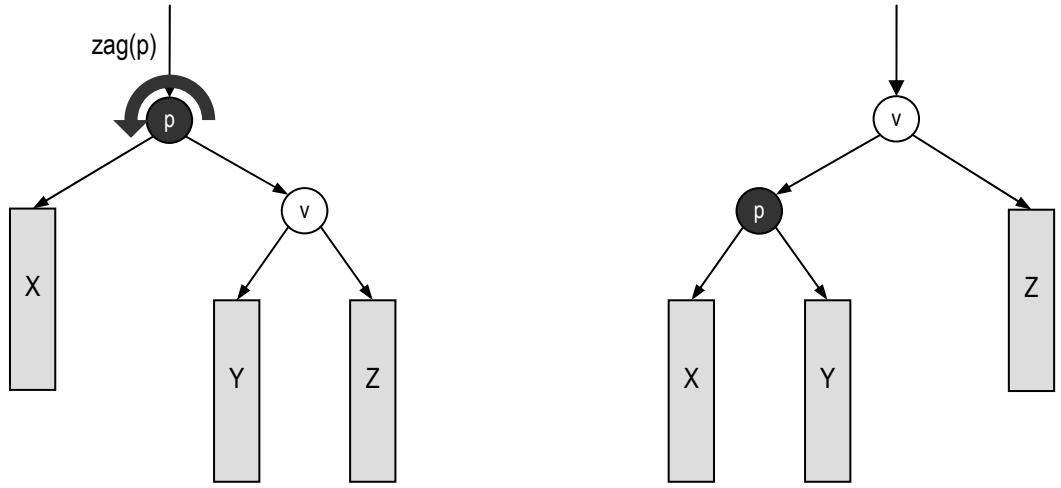


(a) 旋转前 (b) 旋转后

图七.10 zig(p)：顺时针旋转操作

■ zag 旋转

对称地，如 图七.11(a)所示，假定节点v是节点p的右孩子，Z和Y分别是v的右、左子树，X为p的左子树。所谓围绕节点p的zag旋转操作（记作zag(p)），就是将p作为v的左孩子，将Z作为v的右子树，将Y和X分别作为p的右、左子树。



(a) 旋转前 (b) 旋转后

图七.11 zag(p)：逆时针旋转操作

■ 效率

由上可见，无论是 zig(p)和 zag(p)，都只涉及常数次基本操作，于是便有如下结论：

引理七.7 zig 和 zag 旋转操作都可以在常数时间内完成。

■ 效果

对 zig 和 zag 两种旋转操作的作用稍作观察，不难得到如下结论：

观察结论七.3 设 p 为二分查找树 T 中的任一节点，节点 v 为 p 的左（右）孩子，且经 zig(p)（zag(p)）旋转操作之后得到树 T'。则
① v 在 T'中的深度较之它在 T 中的深度减少一；
② 树 T'依然是一棵二分查找树；
③ T'与树 T 等价。

7.2.4 AVL 树

■ 平衡因子

在二分查找树中，任一节点 v 的平衡因子都定义为“其右、右子树的高度差”，记作

balFac(v) = height(lc(v)) - height(rc(v))

这里需要再次强调，空树的高度定义为-1。

■ AVL 树

根据平衡因子，我们可以定义一种特殊的二分查找树：

定义七.4 在二分查找树T中，若所有节点的平衡因子的绝对值均不超过 1，则称T为一棵AVL树[1]。

AVL 树的上述特性在任何局部都满足，故：

观察结论七.4 A VL 树的任一子树也必是 AVL 树。

■ 平衡性

不难看出，在完全二叉树中，节点的平衡因子非 0 即 1，故完全二叉树必是AVL树，但反之不然。完全二叉树的平衡性可以由 引理四.1 保证；那么，AVL树的平衡性又如何呢？我们马上就会看到：就大$\mathcal{O}$记号的意义而言，AVL树也是平衡的。

引理七.8 高度为 h 的 AVL 树，至少包含 Fib(h+3) - 1 个节点。

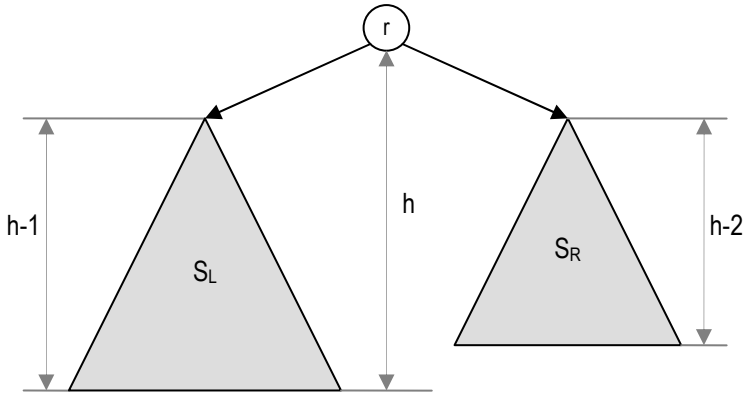
〖证明〗

用数学归纳法证明。考察任一 AVL 树 T。

首先，当 h = 0 时，T 中至少包含 Fib(3) - 1 = 2 – 1 = 1 个节点，命题成立；当 h = 1 时，T 中至少包含 Fib(4) - 1 = 3 – 1 = 2 个节点，命题也成立。

[1] 由 G. M. Adelson-Velsky 和 E. M. Landis 于 1962 年发明，并以他们名字的首字母命名。

假设对于高度不超过 h 的任何 AVL 树，该命题均成立。现考察高度为 h 的所有 AVL 树，并取 S 为其中节点最少的任何一棵（请注意，这种节点最少的树可能不止一棵）。



图七.12 在高度固定为h的前提下，节点最少的AVL树

如 图七.12 所示，设S的根节点为r，r的左、右子树分别为S_L和S_R。记S_L（S_R）的高度为h_L（h_R），其中包含的节点数为$|S_L|$（$|S_R|$）。于是就有：

$$|S| = 1 + |S_L| + |S_R|$$

根据 观察结论七.4，S_L和S_R也都是AVL树，而且高度不超过h-1。进一步地，在考虑到AVL树有关平衡因子的要求的同时，既然S中的节点数最少，故S_L和S_R的高度只能是一个为h-1，另一个为h-2（不失一般性，设$h_L = h-1$，$h_R = h-2$）。而且，在所有高度为h_L（h_R）的AVL树中，S_L（S_R）中包含的节点也应该最少。因此，根据归纳假设，可得如下关系：

$$|S| = 1 + (Fib(h+2) - 1) + (Fib(h+1) - 1)$$

根据 Fibonacci 数列的定义，可得：

$$|S| = Fib(h+2) + Fib(h+1) - 1 = Fib(h+3) - 1$$

证毕。 ❑

根据 引理七.8，可以立即得到如下结论：

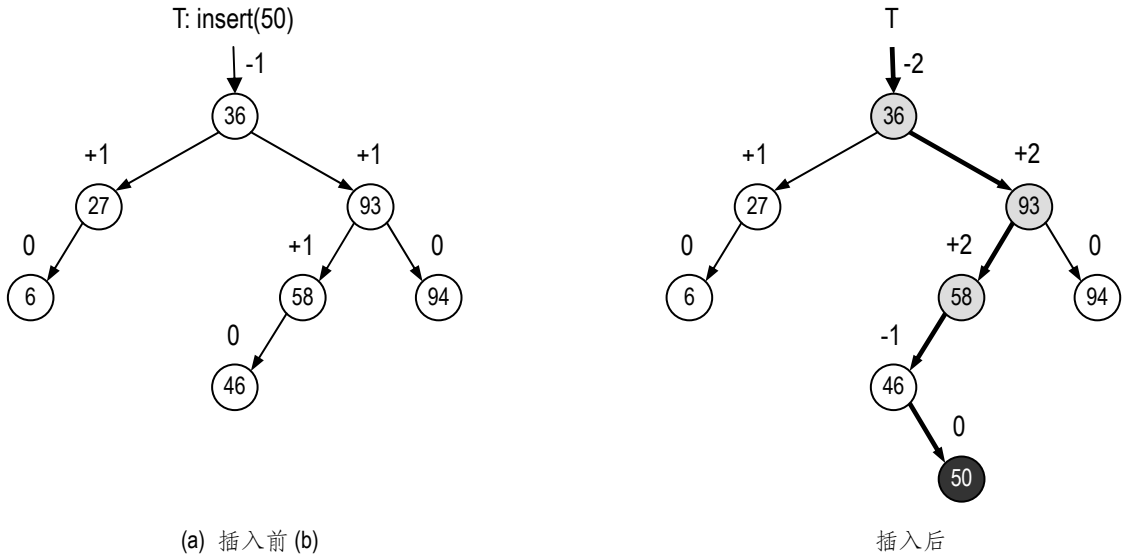
定理七.4 由 n 个节点构成的 AVL 树的高度为 $O(\log n)$。

也就是说，就大 O 记号的意义而言，AVL 树的确是平衡的。

7.2.5 插入节点后的重平衡

与一般的二分查找树一样，AVL 树也不是静态的，也应支持插入、删除等动态修改操作。然而在经过这类操作之后，某些节点的高度可能发生变化，以致于不再满足 AVL 树的条件。在这种情况下，如何使之重新恢复平衡呢？

■ 失衡



(a) 插入前 (b) 插入后

图七.13 插入节点后，AVL树可能失衡

以插入操作为例，我们按照普通二分查找树的插入算法，在如 图七.13(a)所示的AVL树T中插入节点 50。于是，如 图七.13(b)所示，节点 58、93 和 36 都将失去平衡。

■ 失衡节点集

一般地，若在插入新节点 x 之后 AVL 树 T 失去平衡，则可以将失衡的节点组成集合 $U_T(x)$。关于 $U_T(x)$，请读者自行验证以下事实：

观察结论七.5 $U_T(x)$中的每个节点都是 x 的祖先，且高度不低于 x 的祖父。

以上事实的一个推论就是，$U_T(x)$中各节点的深度互异，该集合的规模不超过 depth(x)-1。若取 g(x)为 $U_T(x)$中最深的节点，则 g(x)必然存在且唯一。

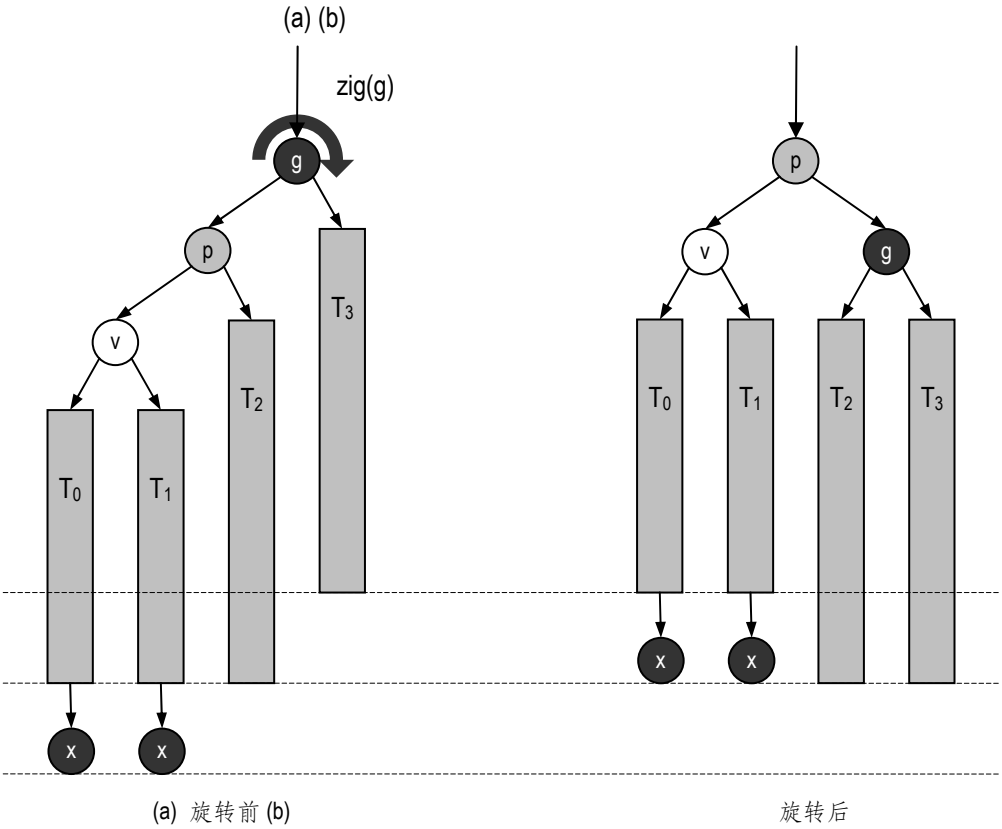
■ 重平衡

为了修正失衡的现象，我们可以从x出发逆行向上，依次检查x各层祖先的平衡因子，直到发现g(x)。在x与g(x)之间的通路上，设p为g(x)的孩子，v为p的孩子。根据 观察结论七.5，p必是x的真祖先，v必是x的祖先。

以下，我们将根据祖孙三代节点 g(x)、p 和 v 的位置关系，通过对 g(x)和 p 的旋转使得这一局部恢复平衡。随后我们将看到，一旦这一局部恢复平衡，整棵树 T 的平衡性也将得到恢复。

■ 单旋

如 图七.14 (a) 所示，设节点v是p的左孩子，且p是g的左孩子。这种情况下，必定是由于在v的后代中插入了新节点x而使得g(x)不再平衡。针对这种情况，可以节点g(x)为轴做顺时针旋转zig(g(x))，从而得到如 图七.14 (b)所示的等价二分查找树。我们将这一过程称作“单旋调整”。



图七.14 通过单旋操作使AVL树重新平衡

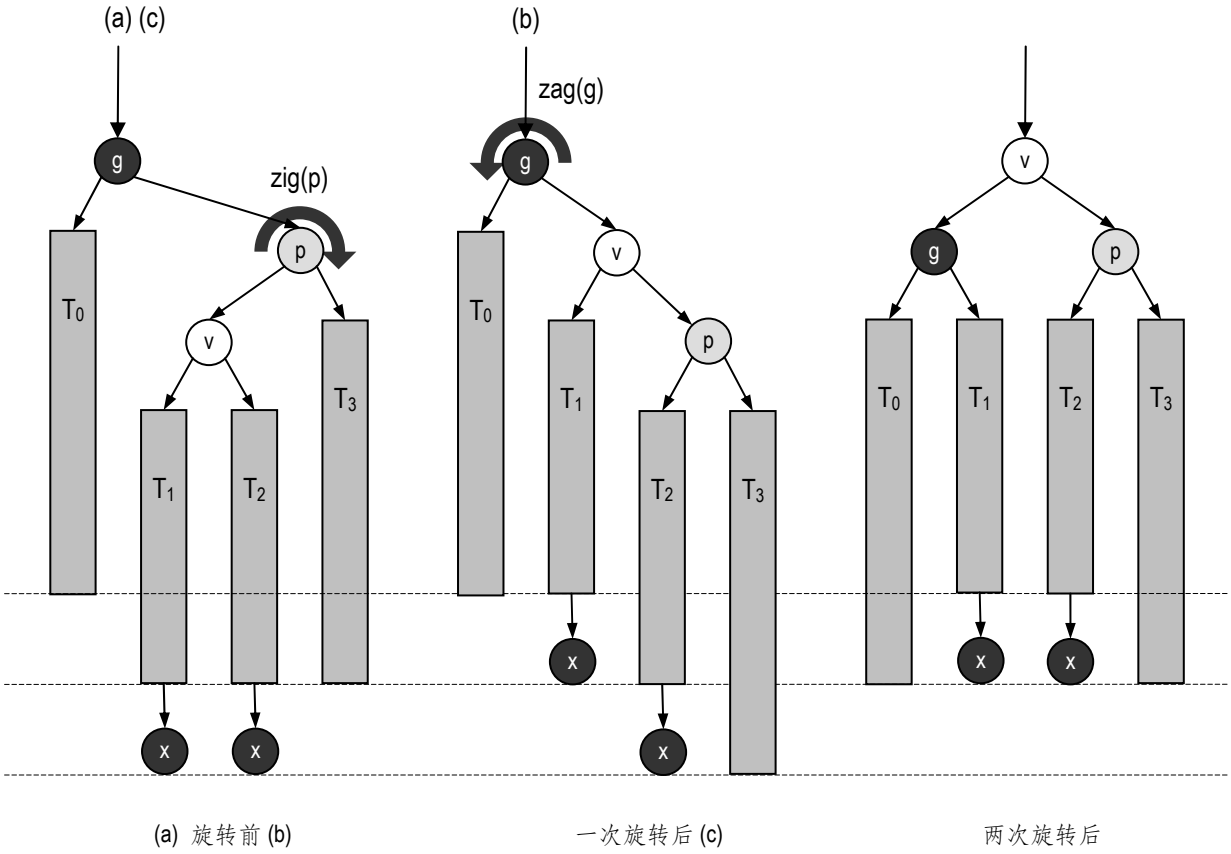
不难看出，当这一局部经过单旋调整后，失衡的节点 g(x)将重新平衡。更重要的是，经这番调整之后，这一子树也将恢复至 x 被插入之前的高度，因此 x 所有其它祖先的平衡因子也都会得到恢复——换而言之，在这种情况下，只需一次旋转操作，即可使整棵 AVL 树恢复平衡。

既然原树是平衡的，故为了查找节点g(x)，我们只需花费$\mathcal{O}$(**log**n)的时间；另外，根据 引理七.7，旋转调整过程也只需$\mathcal{O}$(**log**n)的时间。因此，上述过程总共只需$\mathcal{O}$(**log**n)时间。

当然，还有一种对称的情况：节点 v 是 p 的右孩子，且 p 是 g(x)的右孩子。参照上述调整过程，相信读者能够自行给出重新平衡的算法，并对其效果及效率作一分析。

■ 双旋

如 图七.15(a)所示，设节点v是p的左孩子，而p是g(x)的右孩子。这种情况下，也必定是由于在v的后代中插入了新节点x而使得g(x)不再平衡。针对这种情况，可首先以节点p为轴做顺时针旋转zig(p)，从而得到如 图七.15(b)所示的等价二分查找树。然后，以节点g(x)为轴做逆时针旋转zag(g(x))，从而得到如 图七.15(c)所示的等价二分查找树。我们将这一过程称作“双旋调整”。



图七.15 通过连续的两次旋转操作使AVL树重新平衡

不难看出，这一局部经过双旋调整后，失衡的节点 g(x)将重新平衡。更重要的是，经这番调整之后，这一子树也将恢复至 x 被插入之前的高度，因此 x 所有其它祖先的平衡因子也都会得到恢复——换而言之，在这种情况下，只需两次旋转操作，即可使整棵 AVL 树恢复平衡。

同样地，g(x)可以在$\mathcal{O}$(**log**n)的时间内找到；另外，根据 引理七.7，两次单旋分别只需$\mathcal{O}$(**log**n)时间。因此，整个双旋调整过程只需$\mathcal{O}$(**log**n)时间。

当然，这里也有一种对称的情况：节点 v 是 p 的右孩子，而 p 是 g(x)的左孩子。也请读者自行给出并分析相应的调整算法。

■ 效果

根据上述两类情况的处理方法，可以总结出以下结论：

观察结论七.6 在 AVL 树中插入节点 x 后，若 g(x)是失衡的最低节点，则经过上述单旋或双旋调整之后，不仅能使局部重新平衡同时高度也复原，而且整棵树也将重获平衡。

因此，可进一步得到以下结论：

引理七.9 在 AVL 树中插入一个节点后，至多只需经过两次旋转即可使之恢复平衡。

■ 效率

AVL树也是二分查找树，故根据 引理七.4，为了将节点x插入其中，在进行重平衡调整之前需要的时间不超过树的高度，根据 定理七.4 即$\mathcal{O}(\log n)$。为了找到最低的失衡节点g(x)，最坏情况下需要检查所有共$\mathcal{O}(\log n)$个祖先。而根据 引理七.7，每次旋转都可在常数时间内完成，故插入节点后AVL树的重平衡过程只需常数时间。

综合以上三部分时间，可以得出如下结论：

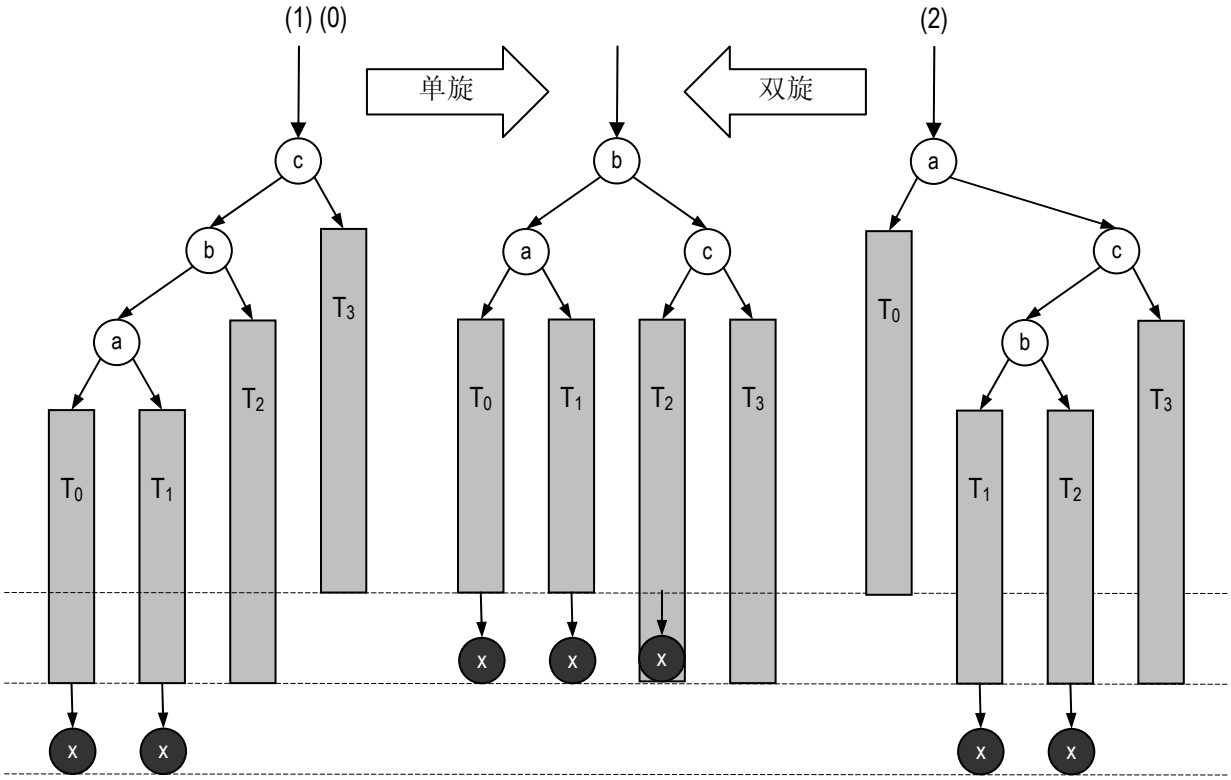
定理七.5 AVL 树的节点插入操作可以在$\mathcal{O}(\log n)$时间内完成。

■ 统一算法

上述重平衡的方法，需要根据失衡节点及其孩子、孙子的位置关系，分别做单旋或双旋调整，若直接按这一思路实现，代码量大且流程繁杂，容易出错。为此，本节将引入一种统一的算法，得益于其简明性，这种实现不仅直接、方便，而且代码精练，错误隐患少。

以前面的方法相同，统一算法首先也要从刚插入的节点 x 出发逆行而上，直到发现最低的失衡节点 g(x)。沿着 g(x)到 x 的通路，依然设 p 为 g(x)的孩子，v 为 p 的孩子。在这一局部，以 p 的兄弟为根有一棵子树，以 v 的兄弟为根也有一棵子树，加上 v 本身的两棵子树，共有四棵子树（注意，其中某些子树可能为空）。易见，在这一局部，以 g(x)为根的子树就是由这三个节点和四棵子树组成的。

接下来，我们对这四棵子树和三个节点重新命名。具体地，我们将根据中序遍历的次序，将四棵子树依次记作T_0、T_1、T_2和T_3；同时，从小到大将g(x)、p和v重新命名为a、b和c。这一重命名的结果如 图七.16 所示。



图七.16 节点插入后的统一重新平衡算法

比如，针对 图七.14(a)和 图七.15(a)的两类情况，这三个节点与四棵子树的重新命名结果分别如 图七.16(1)和(2)所示。

一旦完成这一重命名，我们就可以将这三个节点与四棵子树重新组合起来。如 图七.16(0)所示，以节点b作为新的子树根，将a和c分别作为b的左、右孩子，将T_0 和T_1 分别作为a的左、右子树，将T_2 和T_3 分别作为c的左、右子树。

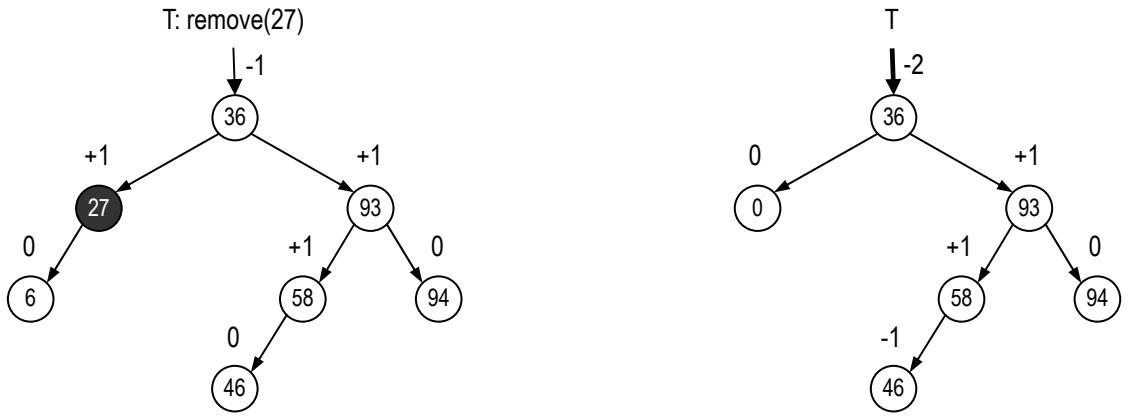
只需将这一结果分别与 图七.14(b)和 图七.15(c)做一对照即可看出，统一算法的效果与此前的单旋、双旋算法完全一致。

另外，请读者自行验证：节点插入统一算法的复杂度依然是 $\mathcal{O}(\log n)$。

7.2.6 节点删除后的重平衡

■ 失衡

为了删除AVL树T中的节点x，我们首先采用第 7.1.5 节中的 算法七.4，将T当作一棵普通的二分查找树，从中删除节点x。



图七.17 删除节点后，AVL树可能失衡

与插入操作一样，在AVL树中删除节点后同样会引发失衡（如 图七.17 所示）。此时，我们也可以通过旋转使之恢复平衡。

■ 失衡节点集

若在删除节点x[㈠]之后AVL树T失去平衡，我们也可以将失衡的节点组成集合$V_T(x)$。$V_T(x)$具有与前面定义的$U_T(x)$类似的性质：

观察结论七.7 $V_T(x)$中的每个节点都是 x 的祖先。

由此同样可知，$V_T(x)$中各节点的深度互异，而且，若取 g(x)为 $V_T(x)$中最深的节点，则 g(x)必然唯一存在。

■ 重平衡

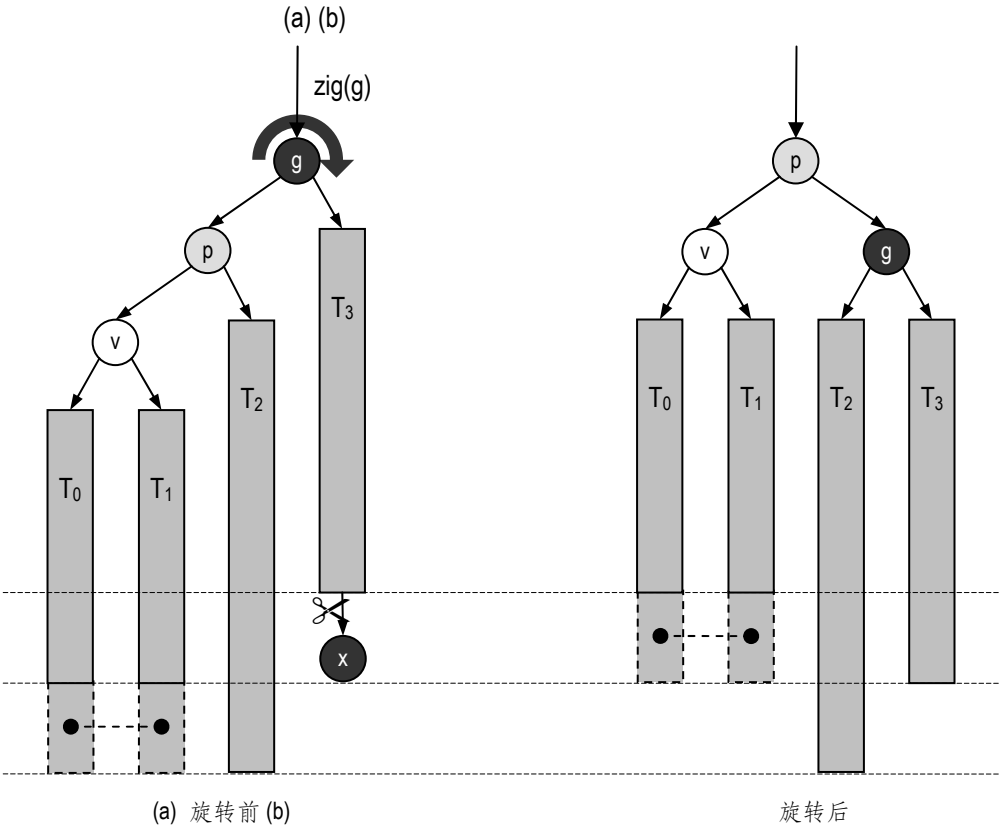
为了修正失衡的现象，我们首先从 x 出发逆行向上，依次检查 x 各层祖先的平衡因子，直到发现 g(x)。此时的 g(x)既然失衡，其左、右孩子的高度应至少相差 2，我们将其中的高者记作 p。既然 p 的高度至少为 1，故必有孩子。我们按照以下规则在 p 的孩子中选出节点 v：若 p 的两个孩子不一样高，则取 v 为其中的高者；否则，取 v 与 p 同向（亦即，v 与 p 同为左孩子，或者同为右孩子）。

根据祖孙三代节点 g(x)、p 和 v 的位置关系，通过对 g(x)和 p 的旋转同样可以使得这一局部恢复平衡。不过，与插入操作不同的是，删除操作后局部平衡的恢复并不意味着整棵树 T 的平衡也得到恢复。在讨论整体平衡的方法之前，以下我们首先介绍局部平衡的算法。可以分为三类情况。

■ 单旋(1)

如 图七.18(a)所示，设节点v是p的左孩子，且p是g的左孩子；子树T_0和T_1的高度相差不超过 1，而且节点v与子树T_2高度相等。这种情况下，必定是由于在T_3中删除了节点x而使得g(x)不再平衡。

[㈠]请注意，若在删除之前 x 需要与其直接前驱 w 交换，则以下叙述中的 x 实际上指的是 w。

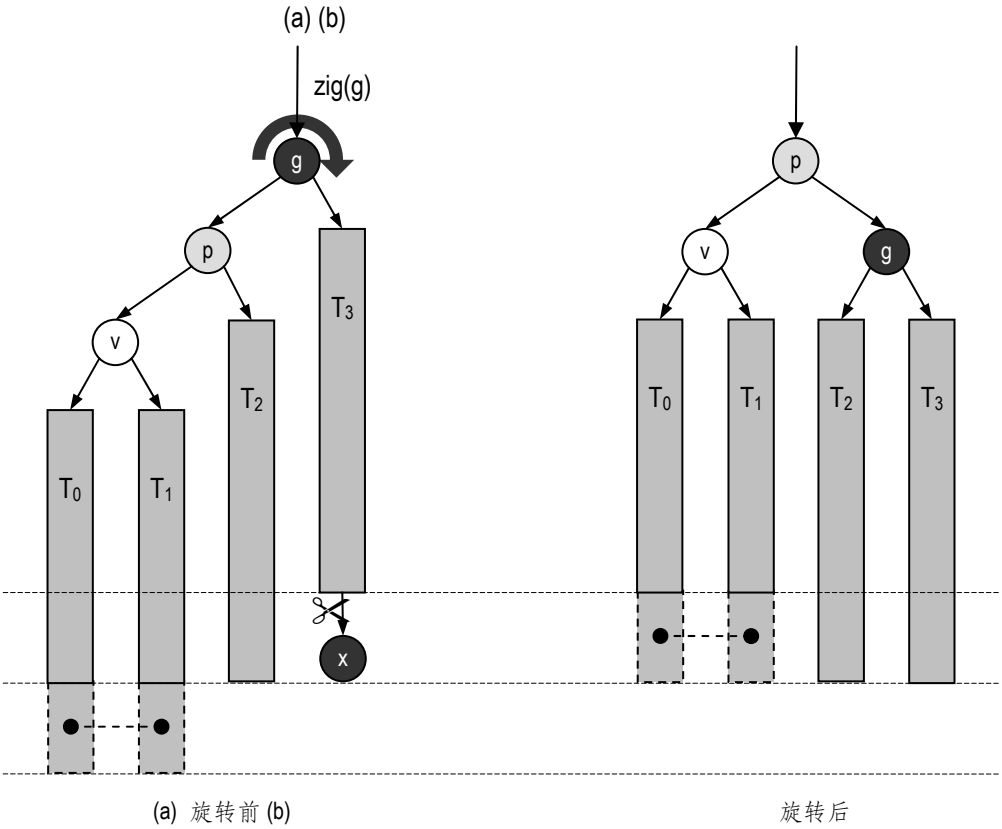


图七.18 节点删除后后重平衡：单旋(1)

针对这种情况，可以如 图七.18(b)所示，通过单旋调整zig(g(x))在这一局部恢复平衡。

- 单旋(2)

如 图七.19(a)所示，设节点v是p的左孩子，且p是g的左孩子；子树T_0和T_1的高度相差不超过 1，而且节点v比子树T_2高一层。这种情况下，也必定是由于在T_3中删除了节点x而使得g(x)不再平衡。

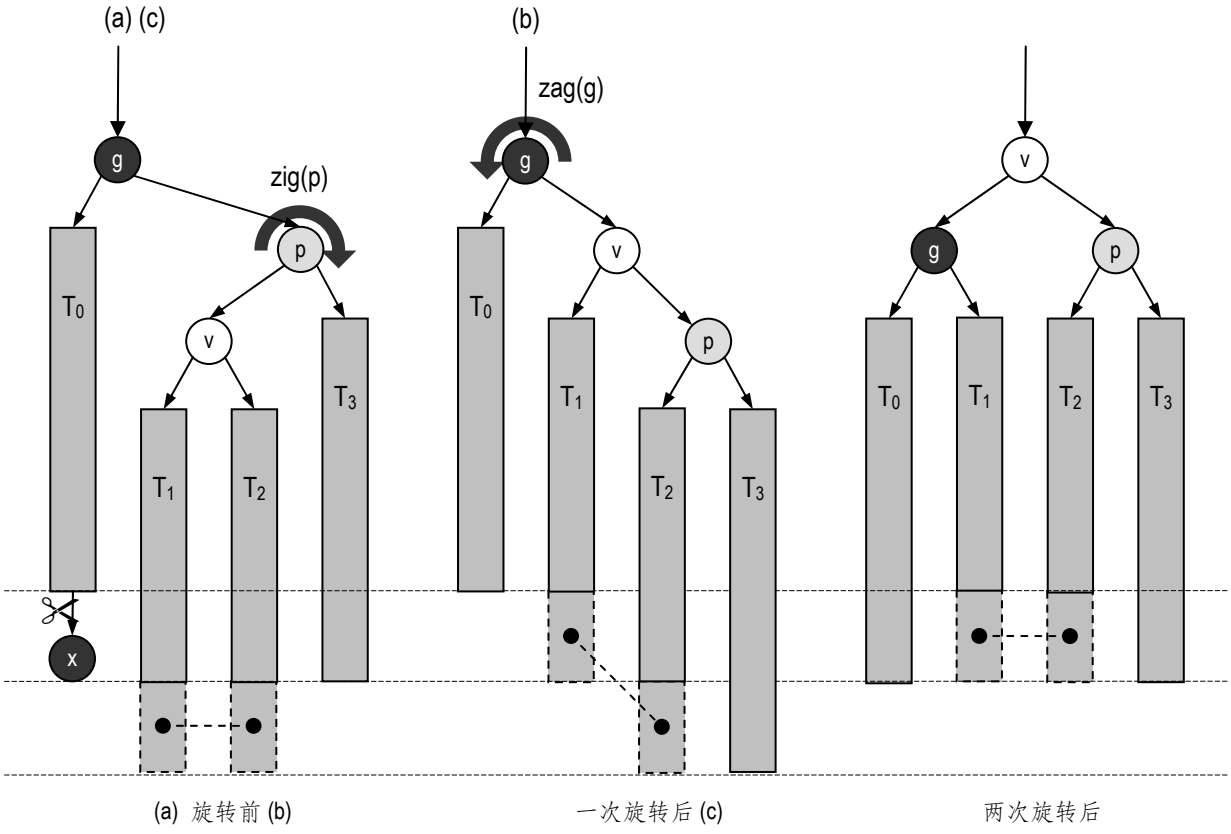


图七.19 节点删除后后重平衡：单旋(2)

针对这种情况，也可以如 图七.19 所示，通过单旋调整zig(g(x))在这一局部恢复平衡。

■ 双旋

如 图七.20 (a)所示，设节点v是p的左孩子，而p是g的右孩子；子树T_1和T_2的高度相差不超过 1，而且节点v比子树T_3高一层。这种情况下，必定是由于在T_0中删除了节点x而使得g(x)不再平衡。



图七.20 节点删除后后重平衡：双旋

针对这种情况，可以如 图七.20 所示，通过双旋调整在这一局部恢复平衡。

与以上例子对称的各种情况，请读者自行给出相应的调整算法并做分析。

■ 失衡的传播

与节点插入极其不相同的是，在删除节点之后，经过单旋或双旋调整尽管可以使局部恢复平衡，但全局依然可能存在失衡的现象。请注意，在 图七.19 和 图七.20 的情况下经单旋调整后虽然局部平衡得到了恢复，但这一局部子树的高度却同时降低了一层，于是，某些祖先节点将有可能因此失去平衡。这种由于对失衡后代的重平衡而造成更高层祖先失衡的现象，称作“失衡传播”。

幸运的是，就以下事实的意义而言，这种传播是单调的：

观察结论七.8 在删除 AVL 节点后，经过上述单旋或双旋调整，最深失衡节点的深度必然减小。

也就是说，即使会出现新的失衡节点，也只能出现在更高层处。因此，我们只需继续逆行向上，找到新的最深失衡节点（如果存在的话），然后再次运用以上方法使之恢复平衡。这一过程可以不断重复，直到不再有失衡节点。

■ 效率

根据 定理七.4，上述调整过程至多需要重复$\mathcal{O}(\log n)$次，由此可以得出如下结论：

引理七.10 在 AVL 树中删除一个节点后，至多只需经过$\mathcal{O}(\log n)$次旋转操作即可使之恢复平衡。

根据 引理七.7，每次旋转都可在常数时间内完成，故在AVL树的重平衡过程中，所有旋转操作总共需要的时间不超过$O(\log n)$。另外，为了查找各个最深失衡节点，我们只需要对x与根节点之间通路上的节点各访问一次，由 定理七.4，这部分时间也不会超过$O(\log n)$。最后，还需计入此前作为常规二分查找树删除节点所需的时间，根据 引理七.5，这部份时间不超过$O(\log n)$。因此综合起来，可以得到如下结论：

定理七.6 AVL 树的节点删除操作可以在$O(\log n)$时间内完成。

7.2.7 AVL 树的 Java 实现

将以上算法引入第 7.1.7 节给出的BSTree类，可以得到AVL树的Java实现如 代码七.3 所示。

```
/*
 * AVL树
 * 基于BSTree的扩充
 */

package dsa;

public class AVLTree extends BSTree implements Dictionary {
/*****************************  构造方法  *****************************/
  public AVLTree() { super(); }
  public AVLTree(BinTreePosition r) { super(r); }
  public AVLTree(BinTreePosition r, Comparator c) { super(r, c); }

/*****************************  词典方法（覆盖父类BSTree）  *****************************/
//插入条目(key, value)，并返回该条目
  public Entry insert(Object key, Object value) {
    Entry e = super.insert(key, value);//调用父类方法完成插入
    root = rebalance(lastV.getParent(), root);//从插入节点的父亲开始重新平衡化
    return e;
  }

//若词典中存在以key为关键码的条目，则将摘除其中的一个并返回；否则，返回null
  public Entry remove(Object key) {
    Entry e = super.remove(key);//调用父类方法完成删除
    if (null != e)  root = rebalance(lastV, root);//从删除节点的父亲开始重新平衡化
    return e;
  }

/*****************************  辅助方法  *****************************/
//从节点z开始，自上而下重新平衡化
//返回后，root仍为平衡后的（整棵）树的根节点
  protected static BinTreePosition rebalance(BinTreePosition z, BinTreePosition r) {
    if (null == z)  return r;
    while (true) {//从z开始，向上逐一检查z的祖先
```

```
      if (!isBalanced(z)) rotate(z);//若z节点失去平衡，则通过旋转使之重新平衡

      if (!z.hasParent()) return z;

      z = z.getParent();//继续检查其父亲
    }//while
  }

//判断节点v是否平衡
  protected static boolean isBalanced(BinTreePosition v) {
    if (null == v)  return true;

    int lH = (v.hasLChild()) ? (v.getLChild().getHeight()) : -1;
    int rH = (v.hasRChild()) ? (v.getRChild().getHeight()) : -1;
    int deltaH = lH - rH;

    return (-1 <= deltaH) && (deltaH <= 1);
  }

//通过旋转，使节点z的平衡因子的绝对值不超过1（支持AVL树）
//返回新的子树根
  public static BinTreePosition rotate(BinTreePosition z) {
    BinTreePosition y = tallerChild(z);//取y为z更高的孩子
    BinTreePosition x = tallerChild(y);//取x为y更高的孩子
    boolean cType = z.isLChild();//记录：z是否左孩子
    BinTreePosition p = z.getParent();//p为z的父亲
    BinTreePosition a, b, c;//自左向右，三个节点
    BinTreePosition t0, t1, t2, t3;//自左向右，四棵子树
  /******** 以下分四种情况 ********/
    if (y.isLChild()) {//若y是左孩子，则
      c = z;  t3 = z.getRChild();
      if (x.isLChild()) {//若x是左孩子
        b = y;  t2 = y.getRChild();
        a = x;  t1 = x.getRChild(); t0 = (BSTreeNode)x.getLChild();
      } else {//若x是右孩子
        a = y;  t0 = y.getLChild();
        b = x;  t1 = x.getLChild(); t2 = (BSTreeNode)x.getRChild();
      }
    } else {//若y是右孩子，则
      a = z;  t0 = z.getLChild();
      if (x.isRChild()) {//若x是右孩子
        b = y;  t1 = y.getLChild();
        c = x;  t2 = x.getLChild(); t3 = (BSTreeNode)x.getRChild();
      } else {//若x是左孩子
        c = y;  t3 = y.getRChild();
        b = x;  t1 = x.getLChild(); t2 = (BSTreeNode)x.getRChild();
      }
    }
```

```
    //摘下三个节点
    z.secede();

    y.secede();

    x.secede();

    //摘下四棵子树
    if (null != t0) t0.secede();
    if (null != t1) t1.secede();
    if (null != t2) t2.secede();
    if (null != t3) t3.secede();

    //重新链接
    a.attachL(t0);  a.attachR(t1);  b.attachL(a);
    c.attachL(t2);  c.attachR(t3);  b.attachR(c);

    //子树重新接入原树
    if (null != p)
      if (cType)  p.attachL(b);
      else        p.attachR(b);

    return b;//返回新的子树根
  }//rotate

//返回节点p的孩子中的更高者
  protected static BinTreePosition tallerChild(BinTreePosition v) {
    int lH = v.hasLChild() ? v.getLChild().getHeight() : -1;
    int rH = v.hasRChild() ? v.getRChild().getHeight() : -1;

    if (lH > rH)  return v.getLChild();
    if (lH < rH)  return v.getRChild();

    if (v.isLChild()) return v.getLChild();
    else              return v.getRChild();
  }

//返回节点p的孩子中的更矮者
  protected static BinTreePosition shorterChild(BinTreePosition v) {
    int lH = v.hasLChild() ? v.getLChild().getHeight() : -1;
    int rH = v.hasRChild() ? v.getRChild().getHeight() : -1;

    if (lH > rH)  return v.getRChild();
    if (lH < rH)  return v.getLChild();

    if (v.isLChild()) return v.getRChild();
    else              return v.getLChild();
  }
}
```

代码七.3 AVL 树类的Java实现

请注意，这里的重平衡过程采用了统一算法。

§7.3 伸展树

第 §7.2 节介绍的AVL树，是平衡二分查找树的一种完美实现方式，但实际上，平衡二分查找树的实现方式还远不止于此，伸展树[1]就是另外一种形式。相对于AVL树，伸展树更为简单。首先，与AVL树不同，伸展树无需对节点实施显式的平衡化操作，而是代之以一种直观而简便的操作——将最近被访问的节点推至树根——这一操作称作伸展（Splaying）。因此，在伸展树中，各节点不再需要记录高度、深度或平衡因子之类的信息，故节点结构本身也相对简单。

就时间复杂度而言，伸展树并不保证每次访问都能在 $O(\log n)$时间内完成，事实上，在最坏情况下，对伸展树的单次访问可能需要线性量级的时间。尽管如此，伸展树还是有其存在意义及价值，这是因为，在对伸展树任意的足够多次连续访问中，分摊下来每次访问只需 $O(\log n)$时间。

7.3.1 数据局部性

信息处理的一种典型模式，就是将一组数据项视作一个数据集合，把它们组织为某种适宜的数据结构，进而反复对其（中的元素）进行操作，平衡二分查找树就是这样的一个典型实例。仅从算法的角度来看，各次操作都是相互独立的，各数据元素被访问的机会及次序也是随机的，因此通常对数据结构效率的分析，主要是分析单次操作所需的时间。然而在实际应用中，这一假定并不完全成立。恰恰相反，对于任何一个实际的数据结构，不仅各次操作之间具有极强的相关性，而且各数据元素被访问的机会也极不平衡，访问次序更不是随机的。存在于对同一数据结构的连续多次访问之间的这种相关性，称作“数据局部性”。

数据的局部性有两方面的含义：

① 刚被访问过的元索，极有可能在不久的未来再次被访问到；

② 下一将被访问的元索，极有可能就处于不久之前刚被访问过的元索附近（如果所有元素之间存在某种线性次序的话）。

就平衡二分查找树而言，数据的局部性具体表现为：

① 刚被访问过的节点，极有可能在不久的未来再次被访问到；

② 下一将被访问的节点，极有可能就处于不久之前刚被访问的节点附近。

数据相关性在实际应用中普遍存在，比如，存储器的缓存技术之所以被广泛而有效地采用，就是因为实际应用中对存储器的访问也有具有极高的数据相关性。

对词典结构的访问，通常也具有很强的数据局部性。那么，针对这一特性，我们应该如何加以利用呢？下面，就让我们结合伸展树的实例予以说明。

---

[1] 1985 年由 Sleator 和 Tarjan 发明： D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Trees. JACM, 32:652-686, 1985.

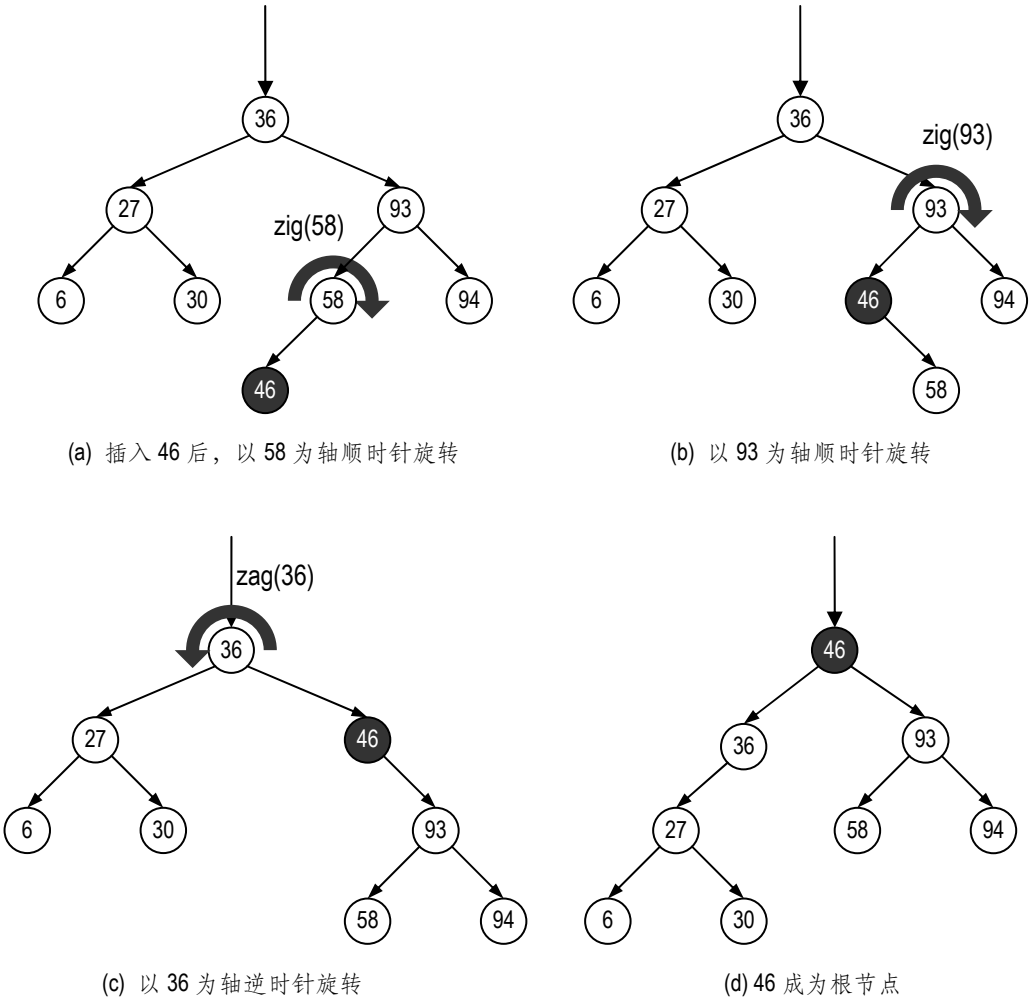
7.3.2 逐层伸展

■ 简易伸展树

引入伸展树的最初动机，在于利用平衡二分查找树结构数据局部性的一种极端情形：刚被访问过的节点，极有可能就是下一将被访问的节点。

在这一条件下，一种简捷而有效的策略就是：每次访问过某个节点之后，都通过某种方式将其移到树根处。这样，在绝大部分时候，被访问的目标节点都恰好位于树根处，因此只需常数时间即可完成一次访问——显然，这一结果要比 AVL 树更好。那么，这一策略应该如何具体实现呢？

我们首先想到的还是第 7.2.3 节介绍的旋转操作。根据 观察结论七.3，zig和zag旋转操作都是二分查找树的等价变换。因此，每次访问过某一节点v之后，我们只要反复地对v的父节点实施相应的旋转操作，就可以不断降低v的深度，最终使它成为树根——这正是我们所需要的。



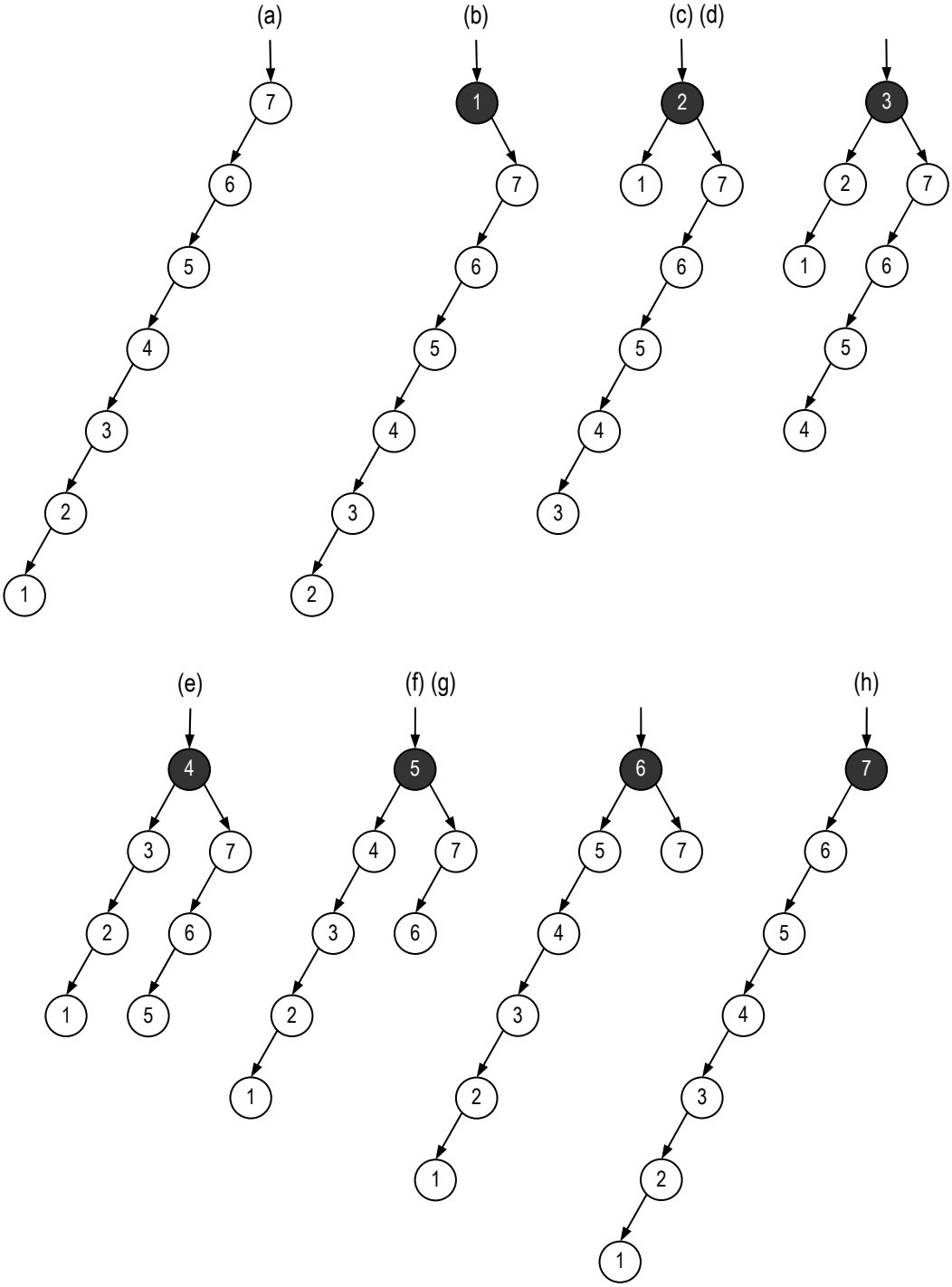
图七.21 在访问过节点46后，自下而上依次对其三个真祖先进行旋转（zig(58)、zig(93)和zag(36)），使之上升至树根

如 图七.21 所示的就是这一过程的一个实例。

至此我们已经得到了伸展树的雏形（也称作简易伸展树），实际上，真正的伸展树所采用的策略与上述策略相差无几。然而，基于这一策略的简易伸展树的确存在致命的缺陷——对于某些访问序列，每次访问的分摊时间复杂度将高达Ω(n)。

■ 最坏情况

让我们来考虑这样的一个实例。假设从空树开始，我们依次插入关键码分别为 1、2、…、n的n个节点，按照上述策略，将得到如 图七.22(a)所示的一棵简易伸展树T。



图七.22 简易伸展树的最坏情况

接下来，我们再依次访问这n个节点。首先，访问过节点 1 之后，它将被伸展到树根（如 图七.22(b)所示）；访问过节点 2 之后，它也将被伸展到树根（如 图七.22(c)所示）；…等到节点n被访问过并被伸展到树根之后，树T的形状如 图七.22(h)所示。

这样的 n 次连续访问，总共需要多少时间呢？只需统计这一过程中所进行旋转操作的总次数。不难看出，对节点 1 的访问需要做 n-1 次旋转；而对于 $2 \le k \le n$，对节点 k 的访问则需要做 n-k+1 次旋转。因此，旋转操作的总数为

$$(n-1) + \{ (n-1) + (n-2) + \ldots + 1 \} \quad = \quad n^2 + n - 2 \quad = \quad \Omega(n^2)$$

因此分摊下来，每次访问需要Ω(n)的时间。较之 AVL 树，这一效率要低将近一个线性因子。

另外更糟糕的是，在经过这样的连续 n 次访问之后，树 T 的形状将会复原。因此，只要按照这一次序对 T 反复访问，则无论总的访问次数 m 有多大，分摊下来每次访问都将需要Ω(n)时间——这可不是我们所希望的。

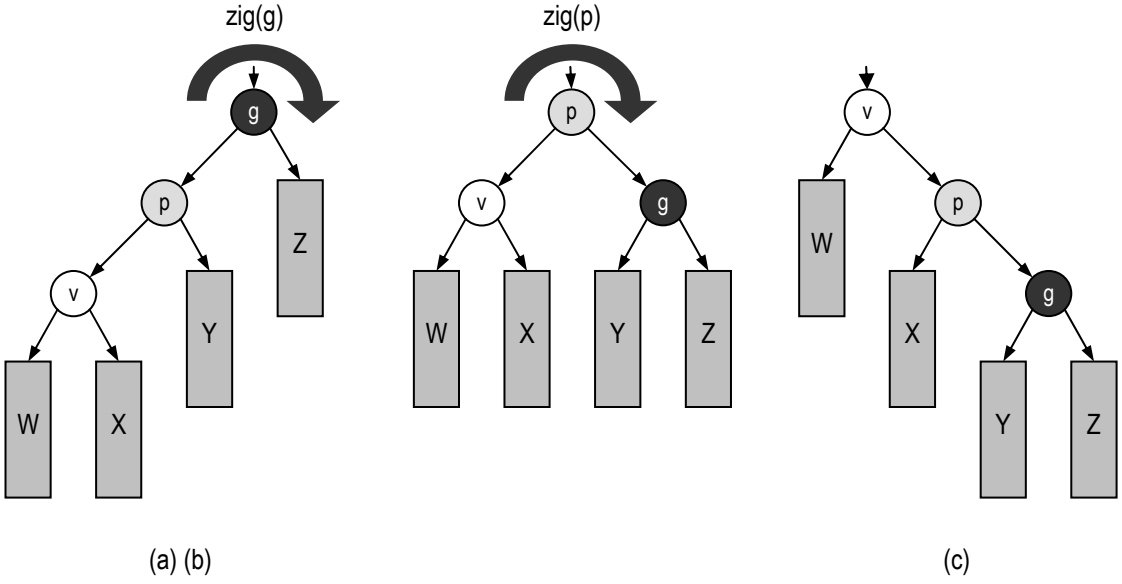
那么，这类最坏的访问序列能否回避呢？具体地，应该如何回避呢？

7.3.3 双层伸展

解决上述问题的一种简便而有效的方法，就是对伸展的策略稍作改进：不再是逐层伸展，而是两层两层地进行伸展。也就是说，每次都从当前节点 v 出发上溯两层，根据其父亲 p 以及祖父 g 的相对位置进行相应的旋转。以下，我们根据祖孙三代节点的位置关系，分两类情况介绍处理的方法。

■ zig-zig / zag-zag 操作

如 图七.23(a)所示，假设v是p的左孩子，而p也是g的左孩子；W和X分别是v的左、右孩子，Y和Z分别是p和g的右孩子。



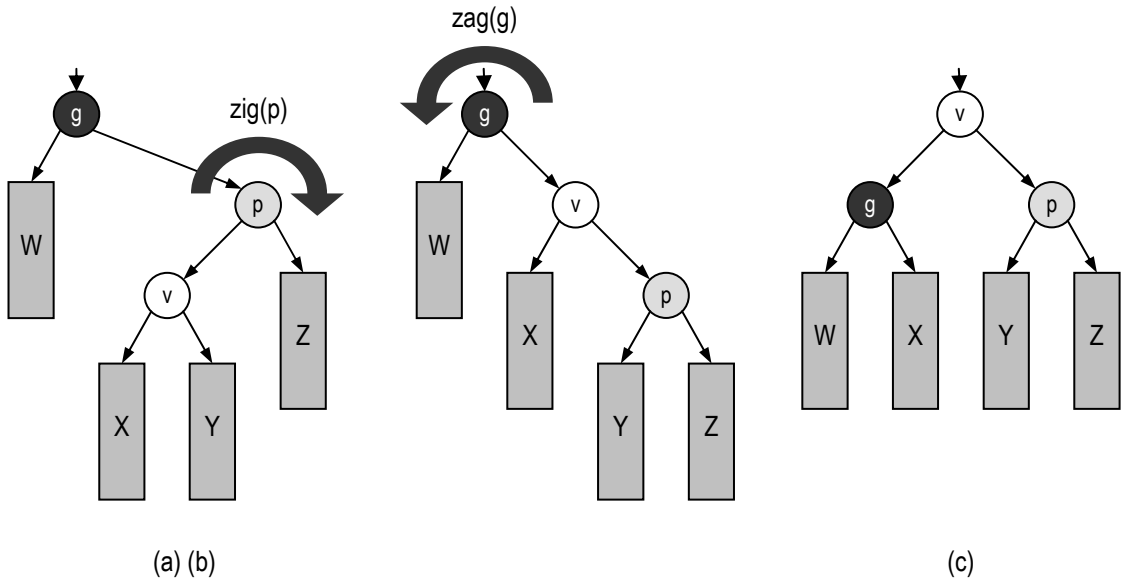
图七.23 通过zig-zig操作，将节点v上推两层

针对这种情况，我们首先以节点g为轴做zig(g)旋转，结果如 图七.23(b)所示。然后，继续以p为轴做zig(p)旋转，得到如 图七.23(c)所示的结果。这样的连续两次zig旋转，称作zig-zig操作。

还有一种对称的情形：v 是 p 的右孩子，而 p 也是 g 的右孩子。不难看出，这一情况可以通过连续的两次 zag 旋转进行调整——这一过程也因此称作 zag-zag 操作。请读者自行绘出这一操作的过程。

■ zig-zag / zag-zig 操作

如 图七.24 所示，假设v是p的左孩子，而p却是g的右孩子；W是g的左孩子，X和Y分别是v的左、右孩子，Z是p的右孩子。



图七.24 通过zig-zag操作，将节点v上推两层

针对这种情况，我们首先以节点p为轴做zig(p)旋转，结果如 图七.24(b)所示。然后，继续以g为轴做zag(g)旋转，得到如 图七.24(c)所示的结果。这样的zig旋转以及随后的zag旋转，称作zig-zag操作。

还有一种对称的情形：v 是 p 的右孩子，而 p 却是 g 的左孩子。不难看出，这一情况可以通过zag(p)旋转和一次 zig(g)旋转进行调整——这一调整过程称作 zag-zig 操作。请读者自行绘出这一操作的过程。

■ 效果、效率及特殊情况

关于上述操作，不难得到如下结论：

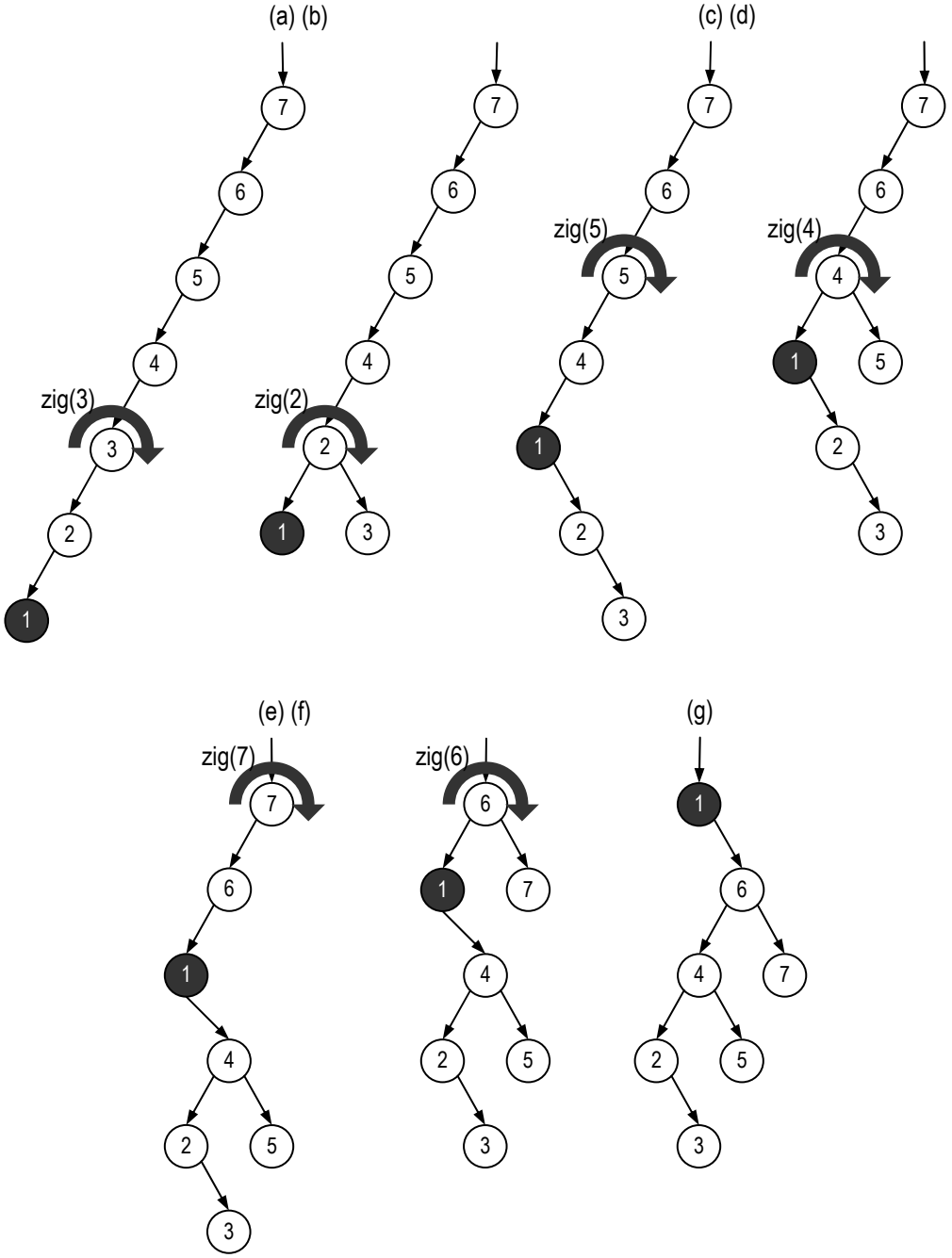
观察结论七.9 上述两类操作，均可在常数时间内完成；每经过一次这样的调整，节点 v 都会上升两层。

请注意，倘若 v 最初的深度为奇数，则经过若干次上述调整操作之后，v 的祖父节点将不再存在。此时，我们只需根据 v 与其父亲节点 p 的相对位置，对 p 实施一次 zig(p)或者 zap(p)，即可最终将 v 推至根节点。

7.3.4 分摊复杂度

重新审视 图七.22 中的最坏实例后我们可以发现，这一访问序列之所以会导致Ω(n)的分摊运行时间，是因为在该简易伸展树中不仅在某个时刻可能存在很深的节点，而且在这种节点“不幸”被访问之后，整棵树的高度是以算术级数递减的，因此实际上任何时刻都存在深度至少是n/2 的节点。那么，按照上述双层伸展的策略将每一刚被访问过的节点推至树根位置，是否可以避免如 图七.22 所示的最坏情况呢？

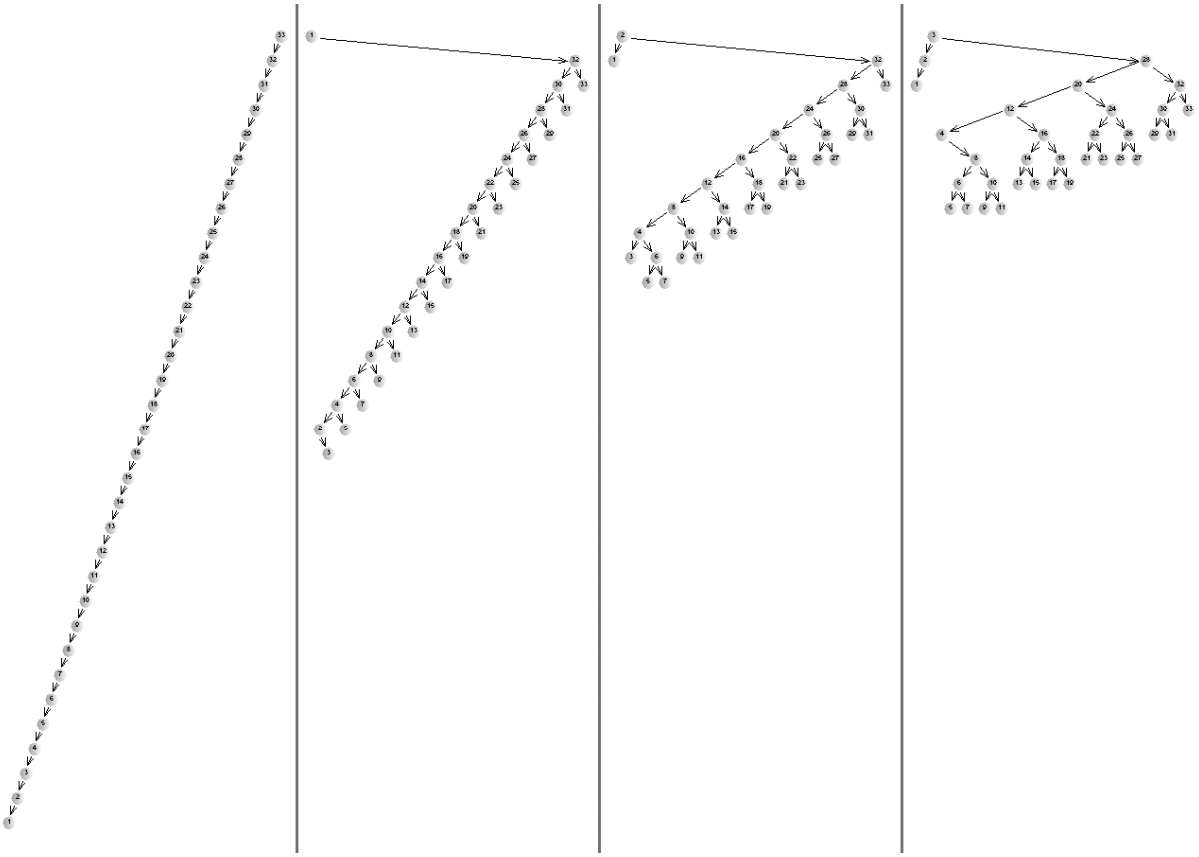
图七.25 针对上述问题给出了正面的回答。



图七.25 即使节点v的深度为Ω(n)，采用双层伸展策略进行调整后，不仅可以将v推至树根位置，而且树高也将降低一半

在如 图七.25(a)所示的伸展树中，一旦节点 1 被访问过，则经过一系列的双层伸展操作（图七.25(b~g)）后，不仅可以将该节点推至树根，而且可以使树的高度减半。

图七.26 则给出了一个节点更多、更具一般性的例子。



图七.26 一旦伸展树中某一较深的节点被访问到，则树高就能迅速地减半，直至接近最优的$\mathcal{O}(\log n)$

由此可见，只要采用上述改进的伸展策略，尽管在某一特定时刻的确可能存在很深的节点，但只要这类“坏的”节点一旦被访问到，经双层伸展之后树的高度就会以几何级数缩减。即使每次都“恶意”地去访问伸展树中最低的叶子，整个树的高度也会很快地降低至$\mathcal{O}(\log n)$左右。正是由于这一特性，伸展树虽然不能杜绝最坏情况的发生，却能有效地控制最坏情况发生的频率。

事实上，采用双层伸展策略的伸展树不仅能够避免如 图七.22 所示的最坏情况，而且正如以下定理将要指出的，就分摊时间复杂度的意义而言，对于伸展树，没有任何“坏的”访问序列。

定理七.7 ① 对伸展树的单次访问，最好情况下需要$\mathcal{O}(1)$时间，在最坏情况下需要$\Omega(n)$时间；
② 在对伸展树任意多次连续的访问过程中，每次访问的分摊时间复杂度为$\mathcal{O}(\log n)$。

〖证明〗

定理的前一部分不难证明。在最好情况下，被访问的节点恰处于树根处（或者与树根非常接近），于是只需做常数次比较。最坏情况如 图七.22(a)所示，由于树高为Ω(n)，故为了访问最低的节点，至少需要进行线性次比较。

定理后一部分的证明过程[1]复杂且繁琐，在此省略。 ❑

7.3.5 伸展树的 Java 实现

至此，我们可以给出伸展树的具体实现如下：

```
/*
 * 伸展树
 * 基于BSTree的扩充
 */

package dsa;

public class SplayTree extends BSTree implements Dictionary {
/*************************** 构造方法 ***************************/
  public SplayTree() { super(); }
  public SplayTree(BinTreePosition r) { super(r); }
  public SplayTree(BinTreePosition r, Comparator c) { super(r, c); }

/*************************** 词典方法（覆盖父类BSTree） ***************************/
//若词典中存在以key为关键码的条目，则返回其中的一个条目；否则，返回null
  public Entry find(Object key) {
    if (isEmpty())  return null;
    BSTreeNode u = binSearch((BSTreeNode)root, key, C);
    root = moveToRoot(u);
    return (0 == C.compare(key, u.getKey())) ? (Entry)u.getElem() : null;
  }

//插入条目(key, value)，并返回该条目
  public Entry insert(Object key, Object value) {
    Entry e = super.insert(key, value);//调用父类方法完成插入
    root = moveToRoot(lastV);//重新平衡化
    return e;
  }

//若词典中存在以key为关键码的条目，则将摘除其中的一个并返回；否则，返回null
  public Entry remove(Object key) {
    Entry e = super.remove(key);//调用父类方法完成删除
    if (null != e && null != lastV) root = moveToRoot(lastV);//重新平衡化
    return e;
```

[1] 对此有兴趣的读者，可以参阅：R. E. Tarjan. Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods, 6(2):306-318, 1985.

```
  }


/*************************** 辅助方法 ***************************/
//从节点z开始，自上而下重新平衡化
  protected static BinTreePosition moveToRoot(BinTreePosition z) {
    while (z.hasParent())
      if (!z.getParent().hasParent())
        if (z.isLChild()) z = zig(z);
        else              z = zag(z);
      else
        if (z.isLChild())
          if (z.getParent().isLChild()) z = zigzig(z);
          else                          z = zigzag(z);
        else
          if (z.getParent().isLChild()) z = zagzig(z);
          else                          z = zagzag(z);
    return z;
  }

//v为左孩子
//顺时针旋转v，使之上升一层（伸展树）
//返回新的子树根
  protected static BinTreePosition zig(BinTreePosition v) {
    if (null != v && v.isLChild()) {//v必须有父亲，而且必须是左孩子
      BinTreePosition p = v.getParent();//父亲
      BinTreePosition g = p.getParent();//祖父
      boolean asLChild = p.isLChild();//父亲是否祖父的左孩子
      v.secede();//摘出v（于是p的左孩子为空）
      BinTreePosition c = v.getRChild();//将v的右孩子
      if (null != c)  p.attachL(c.secede());//作为p的左孩子
      p.secede();//摘出父亲
      v.attachR(p);//将p作为v的右孩子
      if (null != g)//若祖父存在，则将v作为其孩子
        if (asLChild) g.attachL(v);
        else          g.attachR(v);
    }
    return v;
  }

//v为右孩子
//逆时针旋转v，使之上升一层（伸展树）
//返回新的子树根
  protected static BinTreePosition zag(BinTreePosition v) {
    if (null != v && v.isRChild()) {//v必须有父亲，而且必须是右孩子
      BinTreePosition p = v.getParent();//父亲
```

```
      BinTreePosition g = p.getParent();//祖父
      boolean asLChild = p.isLChild();//父亲是否祖父的左孩子
      v.secede();//摘出v（于是p的左孩子为空）

      BinTreePosition c = v.getLChild();//将v的左孩子

      if (null != c)  p.attachR(c.secede());//作为p的右孩子
      p.secede();//摘出父亲
      v.attachL(p);//将p作为v的左孩子
      if (null != g)//若祖父存在，则将v作为其孩子
        if (asLChild) g.attachL(v);
        else          g.attachR(v);
    }
    return v;
  }

//v为左孩子，父亲为左孩子
//顺时针旋转v，使之上升两层（伸展树）
//返回新的子树根
  protected static BinTreePosition zigzig(BinTreePosition v) {
    if (null != v && v.isLChild() && v.hasParent() && v.getParent().isLChild()) {
      BinTreePosition p = v.getParent();//父亲
      BinTreePosition g = p.getParent();//祖父
      BinTreePosition s = g.getParent();//曾祖父
      boolean asLChild = g.isLChild();//祖父是否曾祖父的左孩子
      g.secede();
      p.secede();
      v.secede();
      BinTreePosition c;//临时变量，辅助孩子的移动
      c = p.getRChild(); if (null != c) g.attachL(c.secede());//p的右孩子作为g的左孩子
      c = v.getRChild(); if (null != c) p.attachL(c.secede());//v的右孩子作为p的左孩子
      p.attachR(g);//g作为p的右孩子
      v.attachR(p);//p作为v的右孩子
      if (null != s)//若曾祖父存在，则将v作为其孩子
        if (asLChild) s.attachL(v);
        else          s.attachR(v);
    }
    return v;
  }

//v为右孩子，父亲为右孩子
//顺时针旋转v，使之上升两层（伸展树）
//返回新的子树根
  protected static BinTreePosition zagzag(BinTreePosition v) {
    if (null != v && v.isRChild() && v.hasParent() && v.getParent().isRChild()) {
      BinTreePosition p = v.getParent();//父亲
```

```
      BinTreePosition g = p.getParent();//祖父
      BinTreePosition s = g.getParent();//曾祖父
      boolean asLChild = g.isLChild();//祖父是否曾祖父的左孩子
      g.secede();
      p.secede();

      v.secede();

      BinTreePosition c;//临时变量，辅助孩子的移动
      c = p.getLChild(); if (null != c) g.attachR(c.secede());//p的左孩子作为g的右孩子
      c = v.getLChild(); if (null != c) p.attachR(c.secede());//v的左孩子作为p的右孩子
      p.attachL(g);//g作为p的左孩子
      v.attachL(p);//p作为v的左孩子
      if (null != s)//若曾祖父存在，则将v作为其孩子
        if (asLChild) s.attachL(v);
        else          s.attachR(v);
    }
    return v;
  }

//v为左孩子，父亲为右孩子
//顺时针旋转v，使之上升两层（伸展树）
//返回新的子树根
  protected static BinTreePosition zigzag(BinTreePosition v) {
    if (null != v && v.isLChild() && v.hasParent() && v.getParent().isRChild()) {
      BinTreePosition p = v.getParent();//父亲
      BinTreePosition g = p.getParent();//祖父
      BinTreePosition s = g.getParent();//曾祖父
      boolean asLChild = g.isLChild();//祖父是否曾祖父的左孩子
      g.secede();
      p.secede();
      v.secede();
      BinTreePosition c;//临时变量，辅助孩子的移动
      c = v.getLChild(); if (null != c) g.attachR(c.secede());//v的左孩子作为g的右孩子
      c = v.getRChild(); if (null != c) p.attachL(c.secede());//v的右孩子作为p的左孩子
      v.attachL(g);//g作为v的左孩子
      v.attachR(p);//p作为v的右孩子
      if (null != s)//若曾祖父存在，则将v作为其孩子
        if (asLChild) s.attachL(v);
        else          s.attachR(v);
    }
    return v;
  }

//v为右孩子，父亲为左孩子
//顺时针旋转v，使之上升两层（伸展树）
```

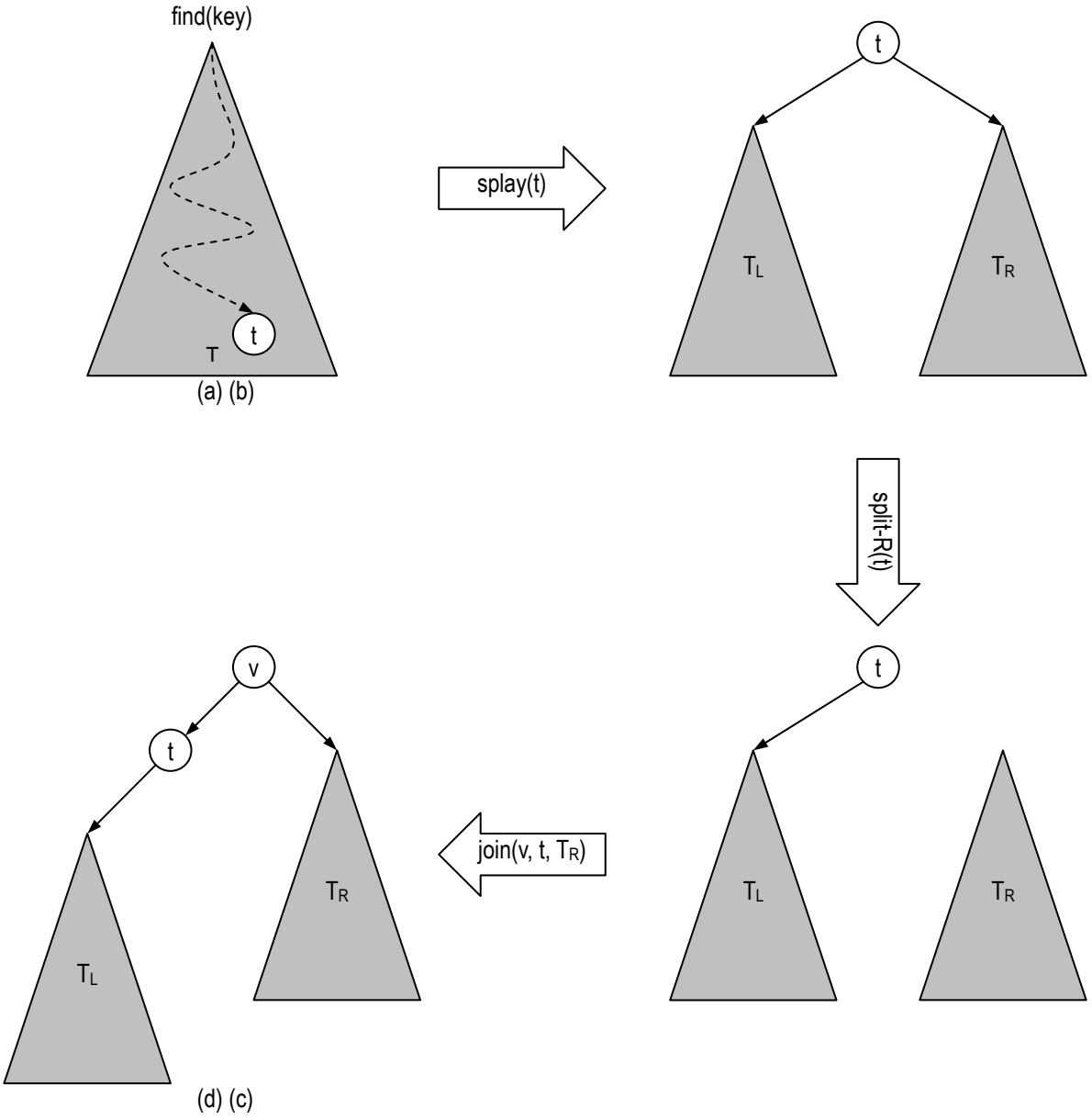
```
//返回新的子树根
  protected static BinTreePosition zagzig(BinTreePosition v) {
    if (null != v && v.isRChild() && v.hasParent() && v.getParent().isLChild()) {
      BinTreePosition p = v.getParent();//父亲
      BinTreePosition g = p.getParent();//祖父
      BinTreePosition s = g.getParent();//曾祖父
      boolean asLChild = g.isLChild();//祖父是否曾祖父的左孩子

      g.secede();

      p.secede();
      v.secede();
      BinTreePosition c;//临时变量，辅助孩子的移动
      c = v.getRChild(); if (null != c) g.attachL(c.secede());//v的右孩子作为g的左孩子
      c = v.getLChild(); if (null != c) p.attachR(c.secede());//v的左孩子作为p的右孩子
      v.attachR(g);//g作为v的右孩子
      v.attachL(p);//p作为v的左孩子
      if (null != s)//若曾祖父存在，则将v作为其孩子
        if (asLChild) s.attachL(v);
        else          s.attachR(v);
    }
    return v;
  }
}
```

代码七.4 伸展树类的实现

7.3.6 插入

为了在伸展树中插入一个新节点，我们可以如 代码七.4 中的insert()方法所示，首先调用常规二分查找树的节点插入算法（算法七.3），然后通过双层伸展，将对新插入的节点推至树根。

不过，我们还可以其它方式来实现伸展树的节点插入，下面介绍的就是一种变型算法。



图七.27 伸展树的节点插入算法

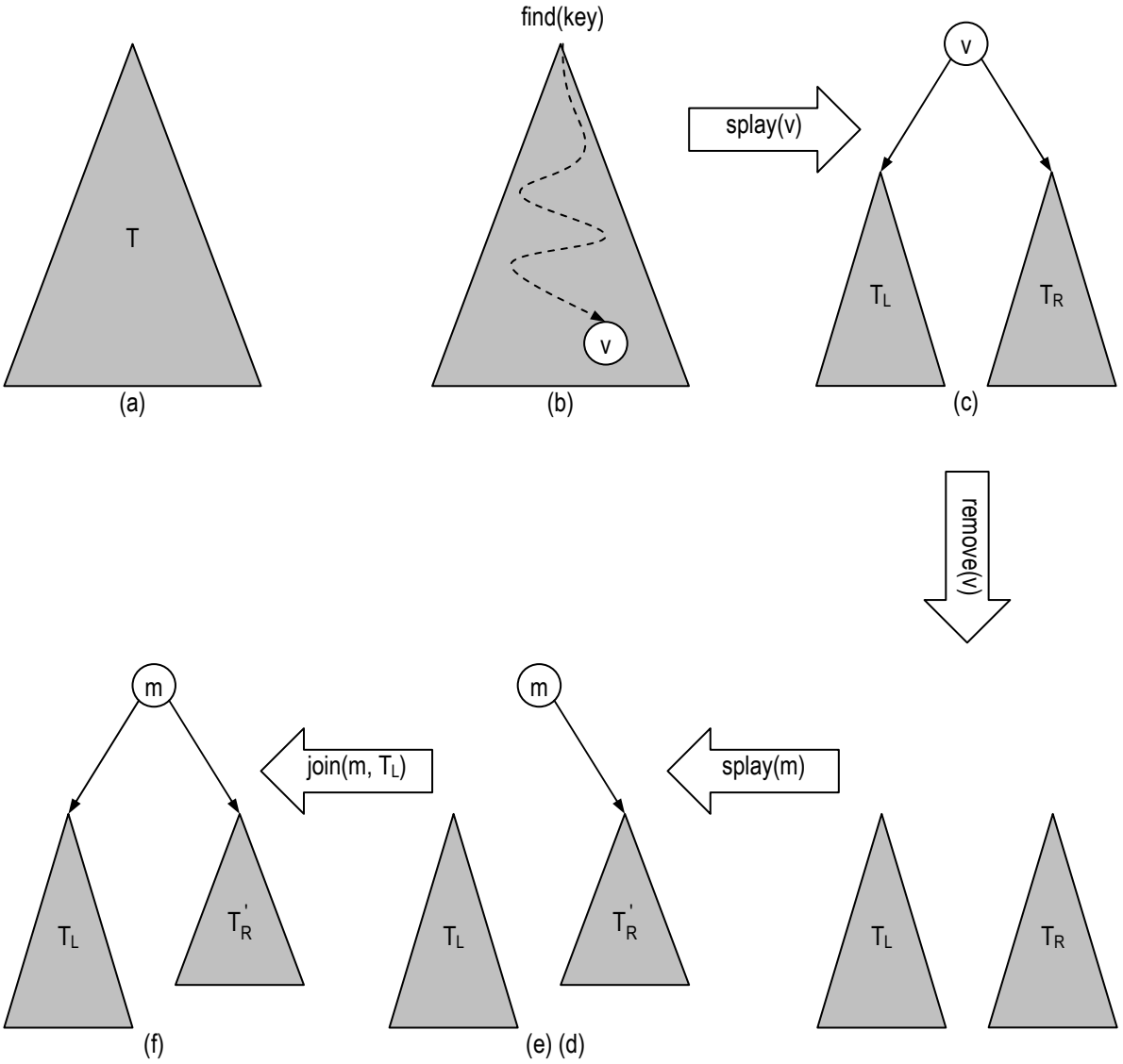
为了将节点v插入至伸展树T中，如 图七.27(a)所示，我们首先针对v的关键码key进行查找，并将最后访问的节点记作t。然后，我们通过双层伸展操作，将t推至树根（如 图七.27(b)，设t的左、右子树分别为T_L和T_R）。接下来，将根据v与t的（关键码）大小关系，以t为界将T分裂为两棵子树。不失一般性地，假设v大于t。此时如 图七.27(c)，我们可以将t指向其右孩子的指针删除；接下来，将节点v作为树根，以t作为其左孩子，以T_R作为其右子树（图七.27(d)）。

针对 v 小于 t 的情况，请读者自行补充相应的算法过程。

7.3.7 删除

为了从伸展树中删除一个节点，我们可以如 代码七.4 中的remove()方法所示，首先调用常规二分查找树的节点删除算法（算法七.4），然后通过双层伸展，将查找过程中最后访问到的节点推至树根。

不过，我们还可以其它方式来实现伸展树的节点删除，下面介绍的就是一种变型算法。



图七.28 伸展树的节点删除算法

为了从伸展树T中删除关键码为key的某一节点，如 图七.28(b)所示，我们也需首先针对该关键码进行查找，不妨设命中的节点为v。同样地，我们通过双层伸展操作将v推至树根。如 图七.28(c)，记v的左、右子树分别为T_L和T_R。接下来，将v摘除（如 图七.28(d)所示，设T_R中的最小节点为m）。然后，再次通过双层伸展操作，将m推至树根（图七.28(e)）。请注意，此时m必然没有左孩子。因此，可以将T_L作为m的左子树，即构成一棵完整的伸展树（图七.28(f)）。

当然，也可以取 m 为 T_L 中的最大节点，相应的算法请读者自行补充。

§7.4 B-树

7.4.1 分级存储

作为人类认识自然及生产实践的重要工具，自其诞生以来短短的近 60 年间，现代电子计算机的发展速度超过了历史上的任何一种其它工具。就计算能力而言，最初的ENIAC[㈠]每秒只能够执行 5000 次加法运算，而今天的“蓝色基因”每秒已经能够执行 7×10^{13} 次浮点运算[㈡]。就存贮能力而言，情况似乎也是如此：ENIAC只有一万八千个电子管，而今天即使是售价不过数百元的普通硬盘，存贮容量也能达到 100GB，内存的常规容量也已达到GB量级。

然而，从实际应用的需求来看，问题规模的扩张速度要远远高于计算机存贮能力的增长速度。以数据库为例，在上世纪八十年代初，典型数据库的规模为 10~100MB；廿年后的今天，典型数据库的规模已经需要以 TB 为单位来计量。相对于如此的发展速度，计算机存贮能力的增长速度远远滞后，而且随着时间推移，这一矛盾日益尖锐。

另一方面，几乎所有类型的存贮器都具有一个共同的特性: 容量越大，速度越慢；反之，容量越小，速度越快。因此，一味提高存贮器容量的做法也是不切实际的。

为了解决上述矛盾，一种有效的方法就是分级存贮。以最简单的二级分级存贮策略为例，这样的典型实例之一，就是由内存与外存（磁盘）组成的二级存贮系统。这一策略的构思是: 将整个数据库存放于外存中，同时在内存中存放最常用数据的复本。这样，借助于有效算法，可以将内存高速度的优点与外存大容量的优点结合起来，而此时的内存则扮演了高速缓存的角色。

实际上，在分级存贮系统中，各级存贮器的速度不仅有所差异，而且通常极其悬殊。还是以内存与磁盘为例，前者的访问速度一般在 10~100ns 左右，而后者的访问速度约在 10ms 左右，二者之差高达 5 到 6 个数量级。如果把对内存的访问比作随手拿起书桌上的钢笔，那么对外存的访问就相当于乘火车从北京到广州某办公室的书桌上拿起钢笔，然后再乘火车返回。因此，为了省去对外存的一次访问，我们宁愿多访问内存一百次、一千次甚至一万次。也正因为这一原因，在衡量此类算法时，我们会更多地考虑其涉及的外存访问次数，而忽略内存操作的次数。

当问题规模很大，以至于内存无法容纳时，即使是前面介绍的平衡二分查找树，在效率方面仍大打折扣。而下面将要介绍的 B-树，却是能够高效解决这类问题的一种数据结构。例如，若将存放于外存的 1G 个记录组织为 AVL 树，则每次访问需要做约 30 次外存访问；而如果采用 256 阶 B-树，则每次访问对应的外存访问次数将减少至 4~5 次。

㈠ 第一台电子计算机，194 6 年 2 月 15 日诞生于美国宾夕法尼亚大学工学院。

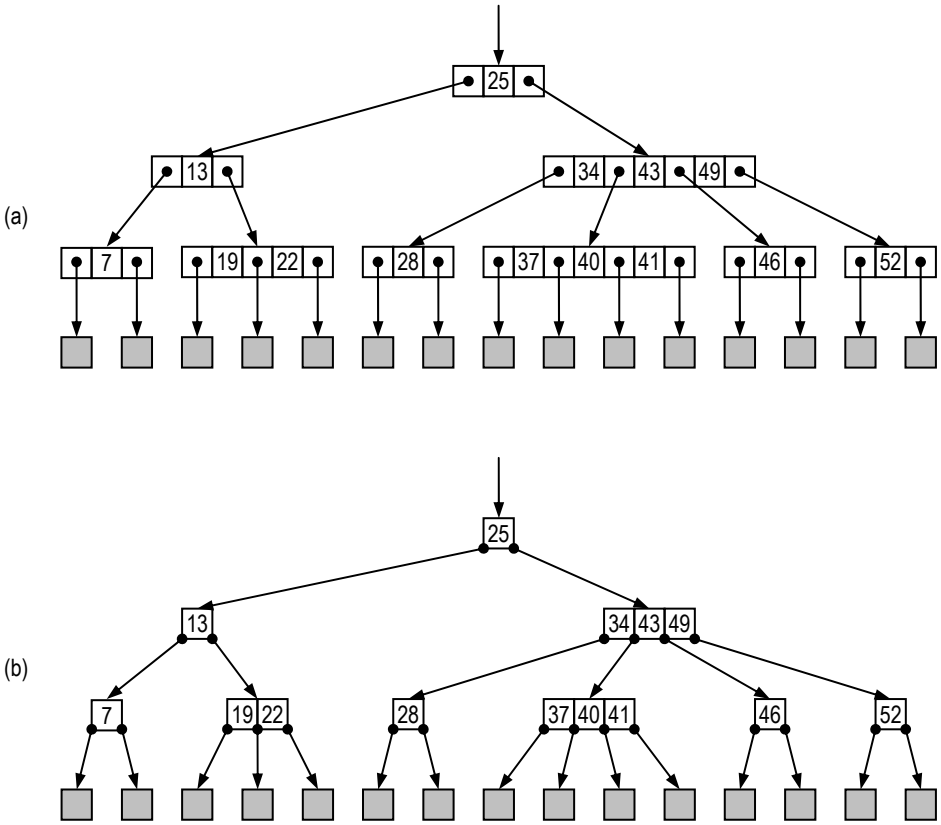
㈡ 200 4 年 11 月，一台于美国劳伦斯·利沃莫尔国家实验室（Lawrence Livermore National Laboratory, LLNL）在建的蓝色基因/L，在只完成整机四分之一安装的时候，就以这样的运算速度名列当年世界超级计算机 500 强榜首。

7.4.2 B-树的定义

如 图七.29 所示，所谓m阶B-树 [1]，即满足以下条件的m路平衡查找树：其中的每一内部节点，都存有n个关键码$\{K_1 < K_2 < ... < K_n\}$和n+1 个引用$\{A_0, A_1, A_2, ..., A_n\}$，$n+1 \le m$。对于每一非根内部节点，都有$n+1 \ge \lceil m/2 \rceil$；对于根节点，除非它同时也是叶子，否则必有$n+1 \ge 2$。

每个引用 A_i 分别指向一棵子树 T_i，而且若 $i \ge 1$，则 T_i 中的每一关键码 key 都满足 $key > K_i$；若 $i \le n-1$，则 T_i 中的每一关键码 key 都满足 $key < K_{i+1}$。与一般的查找树不同，为了简化叙述，这里我们假定 B-树中的所有关键码互异。

另外，所有叶子节点的深度相等，即它们都处于同一层。



图七.29 (a) 由9个内部节点、15个叶子节点和14个关键码组成的一棵4阶B-树，及其(b)紧凑表示

在 图七.29 中，(a)就是一棵 4 阶B-树。通常，我们将更多地以如(b)所示的形式绘出B-树的结构，相比之下，这一形式更为简洁紧凑。

7.4.3 关键码的查找

在一棵B-树T中针对关键码key的查找过程与二分查找树基本类似：从根节点开始，通过关键码的比较，不断深入下一层，直到某一关键码命中（查找成功）或者到达某一叶子节点（查找失败）。但B-树的查找过程与二分查找树也略有区别：由于每个节点内通常都存放了多个关键码，所以可能

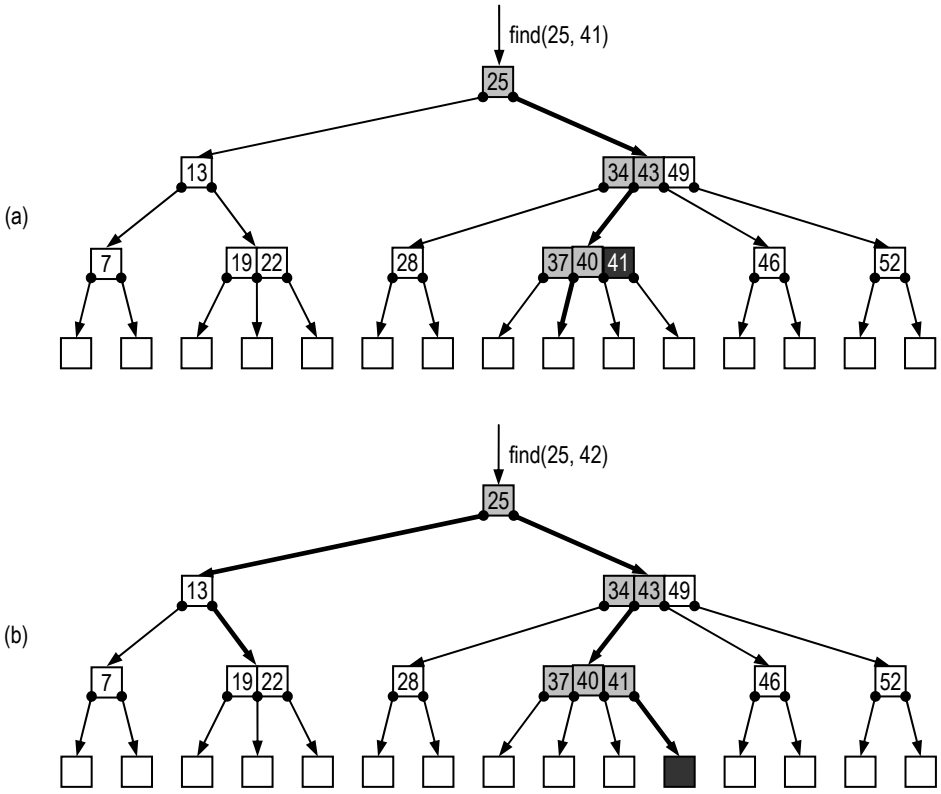
[1] 197 0 年由 R. Bayer 和 E. McCreight 合作发明。

需要多次比较，才能确定应该向下一层的哪一个节点继续深入查找。这一过程具体地可以描述为 算法七.5：

```
算法：find(v, key)
输入：B-树节点v，关键码key
输出：若关键码key在以v为根节点的B-树内存在，则返回该关键码的位置；否则，返回null
{
  若v是叶子，则返回null; //查找失败

  在v中顺序查找关键码key;

  若找到关键码key，则
    返回其位置; //查找成功
  否则 //设key介于v中两个相邻关键码之间，即Ki < key < Ki+1
    调用find(&Ai, key); //沿着引用Ai，深入到下一层继续查找
}
```

算法七.5 B- 树的查找算法

图七.30(a)和(b)分别给出查找成功和失败的两个例子。



图七.30 B- 树的查找实例：(a) 查找成功；(b)查找失败。
其中粗边表示查找过程中访问过的边，灰色方块表示查找过程中比较过的关键码，
黑色方块表示查找终止的位置

7.4.4 性能分析

正如此前所指出的，B-树最适宜于存放大规模的数据，由于这类数据的规模之大，只能将其中很小的一部分存放于内存中。实际的做法是，将每个这样的数据集组织成一棵B-树，并存放于外存；B-树的根节点常驻内存。一旦需要查找，则按照 算法七.5，首先将根节点当作当前节点，并在其中查找。即使当前节点中不存在目标关键码，也可以确定其中的一个引用，通过这个引用可以找到外存中的某一下层节点。随后，该节点将会被读入内存，作为新的当前节点，并再做一次上述查找……。这种过程不断重复，直到在当前节点中找到目标关键码，或者当前节点变成叶子。

由此可见，在 B-树中查找节点所需的时间，不外乎消耗于两类操作：将某一节点从外存中读入内存，以及对内存中的节点进行查找。正如此前所交待的，鉴于内存、外存在访问速度上的巨大差异，前一部分时间必然是主要的，后一部分时间则可以忽略（正因为此，节点内部的查找可以采用简单的算法，比如顺序查找）。故此，B-树查找的效率将主要取决于外存访问的次数。那么，对 B-树的每次查找，需要进行多少次外存的读取呢？

首先，与二分查找算法类似地，在对 B-树的每一次查找过程中，在每一层上至多只有一个节点被访问到。由此可知，每次查找所需进行的外存访问，至多不超过 $\mathcal{O}(h)$次，其中 h 为树高。那么，树高 h 是多少呢？最大多少？最小又是多少？

下面这则引理回答了上述问题：

引理七.11若存有 N 个关键字的 m 阶 B-树的树高为 h，则 $\log_m(N+1) \le h \le \log_{\lceil m/2 \rceil}((N+1)/2) + 1$。

〖证明〗

①首先证明 $h \le \log_{\lceil m/2 \rceil}((N+1)/2) + 1$。

在关键码总数一定的前提下，为使 B-树最高，每个内部节点都应该包含尽可能少的关键码。按照 B-树的定义，各层节点数至少是：

$n_0 = 1$

$n_1 = 2$

$n_2 = 2 \times \lceil m/2 \rceil$

$n_3 = 2 \times \lceil m/2 \rceil^2$

...

$n_{h-1} = 2 \times \lceil m/2 \rceil^{h-2}$

$n_h = 2 \times \lceil m/2 \rceil^{h-1}$

现考察叶子所在层，叶子的数目应该恰好比关键码总数多 1，故有

$$N+1 = \#\text{叶子节点} \ge 2 \times (\lceil m/2 \rceil)^{h-1}, \quad h \ge 1$$

即

$$h \le \log_{\lceil m/2 \rceil}((N+1)/2) + 1 = \mathcal{O}(\log_{\lceil m/2 \rceil}(N/2))$$

②再证明 $h \geq \log_m(N+1)$。

在关键码总数一定的前提下，为使 B-树最矮，每个内部节点都应该包含尽可能多的关键码。按照 B-树的定义，各层节点数至多是：

$n_0 = 1$

$n_1 = m$

$n_2 = m^2$

...

$n_{h-1} = m_{h-1}$

$n_h = m_h$

考察叶子所在层，与①同理有

$$N+1 = \#\text{ 叶子节点} \leq m^h$$

即

$$h \geq \log_m(N+1) = \Omega(\log N)$$

证毕。 ❑

由此立即可以得出结论：

引理七.12 存有 N 个关键字的 m 阶 B-树的高度 $h = \Theta(\log_m N)$。

因此，每次查找至多需要访问 $\mathcal{O}(\log_m n)$个节点，即需要做 $\mathcal{O}(\log_m n)$次外存读取。因此，若假设每次外存操作都只需常数时间，则有如下结论：

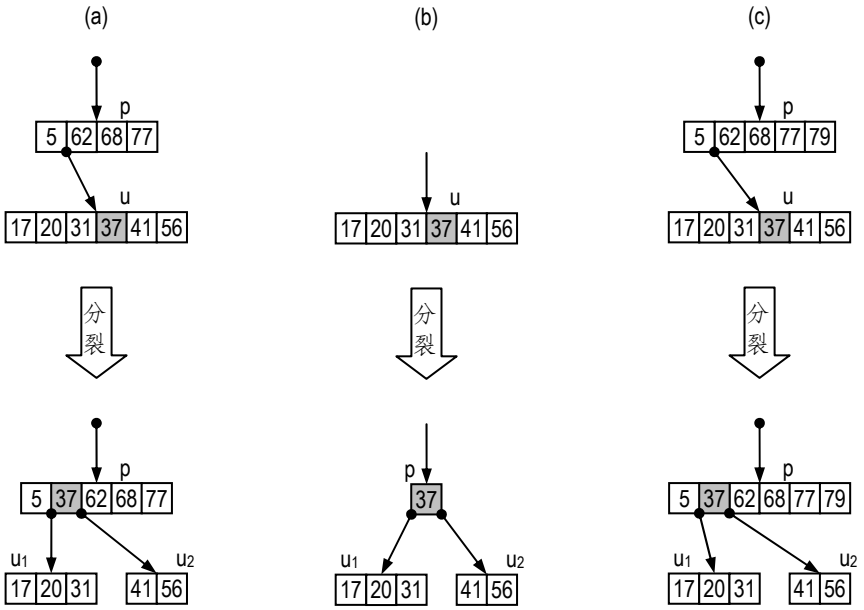
定理七.8 对存有 N 个关键字的 m 阶 B-树的每次查找操作，都可以在 $\mathcal{O}(\log_m N)$的时间内完成。

7.4.5 上溢节点的处理

在以下第 7.4.6 节将要介绍的关键码插入算法中，需要处理一种非法的情况：由于新关键码的引入，内部节点u中关键码的数目可能多达m个——称作节点的上溢（Overflow）。我们首先介绍这种情况的处理方法。

当上溢情况发生之后，设 u 中的关键码依次为$\{k_1, \ldots, k_m\}$。我们以关键码 $k_{\lfloor m/2 \rfloor+1}$ 为界，将该节点分裂为两个节点：

$$\begin{cases} u_1 = \{k_1, \ldots, k_{\lfloor m/2 \rfloor}\} \\ u_2 = \{k_{\lfloor m/2 \rfloor+2}, \ldots, k_m\} \end{cases}$$



图七.31 B- 树中溢出节点的分裂

接下来如 图七.31(a)所示，若u的父节点p存在，则将关键码$k_{\lfloor m/2 \rfloor+1}$插入p中；在节点p中，将关键码$k_{\lfloor m/2 \rfloor+1}$两侧的子树引用分别指向$u_1$和$u_2$。当然，u的父节点不见得存在。果真如此，如 图七.31(b)所示，则可以创建一个只含单个关键码$k_{\lfloor m/2 \rfloor+1}$的节点p。

观察结论七.10 在处理好每一上溢节点之后，其父节点中的关键码必然会增加一个。

特别指出的是，如 图七.31(c)所示，由于新关键码的引入，又有可能进而导致父节点p的上溢。下一小节将介绍这种情况的处理方法。

7.4.6 关键码的插入

为了在B-树v中插入一个新的关键码key，首先需要调用 算法七.5，通过find(v, key)进行查找。倘若查找成功，则考虑到“B-树中各关键码互异”的条件，将不再插入重复的关键码，插入操作即告完成。否则，查找过程必然终止于某一叶子节点。在这一查找过程中，将最后一个被访问到的内部节点记作u。

我们直接将关键码key插入到u中。若此后节点u中关键码的数目依然合法（不超过m-1 个），则插入操作也可立即结束（如 图七.32(b)所示）。否则，按照第 7.4.5 节介绍的方法，对节点u进行分裂。

在接纳了来自u的一个关键码之后，若u的父节点p依然合法（图七.32(d)），则插入操作也可告完成。否则，我们可以继续对p进行分裂，并将其中居中的关键码上交给p的父节点（图七.32(g)）。

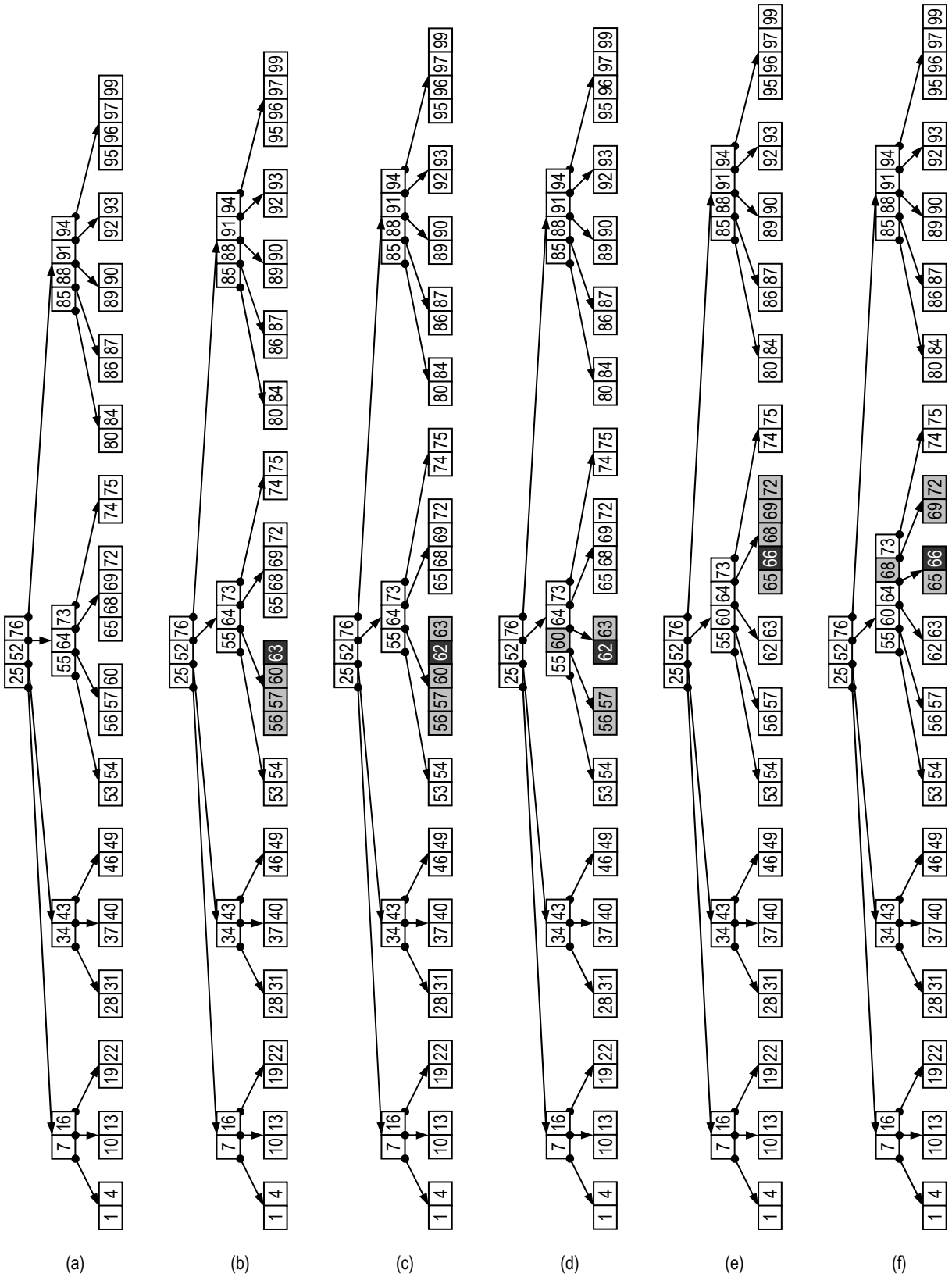
这一过程可能需要反复迭代，直到当前节点不再出现上溢的情况，或者已经到达树根（（图七.32(k)））。整个算法的过程可以描述为 算法七.6：

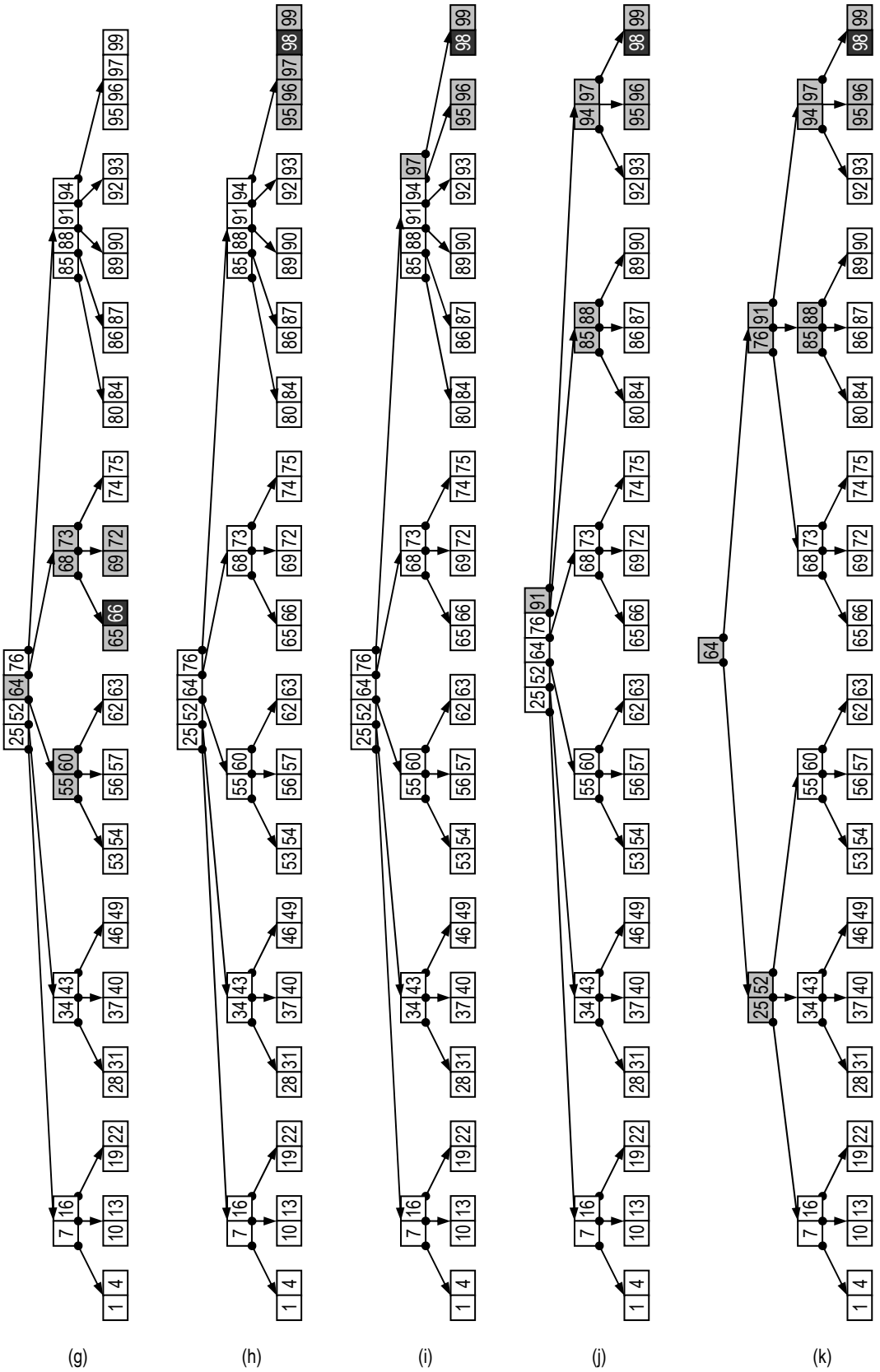
```
算法：insertKey(u, key)
输入：B-树节点u，关键码key
输出：将关键码key插入至节点u中。若因此致使u溢出，则将u一分为二，并返回居中的关键码
```

```
{
  将关键码key插入至节点u中;

  if (u中的关键码数目 ≥ m) { //因为新关键码的引入致使节点u溢出
    以居中的关键码为界，将节点u分裂为三部分:
      u   =  u1              ∪ {k⌊m/2⌋+1}   ∪ u2
          =  {k1, …, k⌊m/2⌋} ∪ {k⌊m/2⌋+1}   ∪ {k⌊m/2⌋+2, …, km};
    令p为u的父节点;
    若p不存在，则创建一个空节点p;
    insertKey(p, k⌊m/2⌋+1); //递归地将关键码k⌊m/2⌋+1插入至p中;
    在节点p中，将关键码k⌊m/2⌋+1两侧的子树引用分别指向u1和u2;
}

算法: insert(v, key)
输入: B-树节点v，关键码key
输出: 将关键码key插入至以v为根节点的B-树中
{
  if (find(v, key)) return; //若关键码key在B树中已经存在，则不必插入

  令u为查找过程中访问过的最后一个节点;
  insertKey(u, key); //调用insertKey()，将关键码key插入节点u中
}
```

算法七.6 B- 树的关键码插入算法

图七.32 给出了连续四次插入操作的一个实例：


(a)
(b)
(c)
(d)
(e)
(f)



图七.32 对5阶B-树(a)的插入操作：(b)插入63，无需分裂；(c~d)插入62，分裂一次；(e~g)插入66，分裂两次；(h~k)插入98，一直分裂到根

不难注意到，若不计递归，算法 insertKey()本身只需常数时间。同时，insertKey()算法在 B-树的每一深度上至多递归调用一次。由此可知如下结论：

定理七.9 对存有 N 个关键字的 m 阶 B-树的每次插入操作，都可以在 $\mathcal{O}(\log_m N)$的时间内完成。

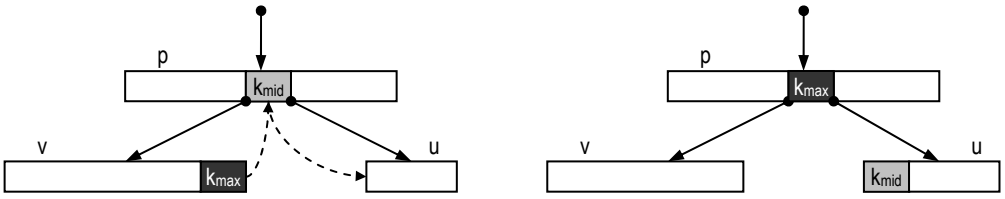
7.4.7 下溢节点的处理

在以下第 7.4.8 节将要介绍的关键码删除算法中，需要处理一种非法的情况：由于其中某一关键码的删除，内部节点u中关键码的数目减少到⌈m/2⌉-2 个——这种情况称作节点的下溢（Underflow）。我们首先介绍这种情况的处理方法。

当节点 u 发生下溢时，根据 u 的左、右兄弟所含关键码的数目，可分为如下情况相应处理：

- 左兄弟 v 至少包含⌈m/2⌉个关键码

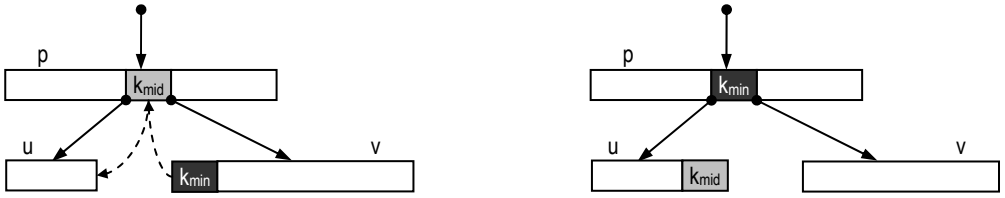
此时，如 图七.33 所示，可以从v中取出最大的关键码k_{max}，将父节点p中介于v和u之间的关键码k_{mid}替换为k_{max}，然后将key_{mid}插至u的最左侧。



图七.33 下溢的节点向父亲“借”一个关键码，父亲再向左兄弟“借”一个关键码

- 右兄弟 v 至少包含⌈m/2⌉个关键码

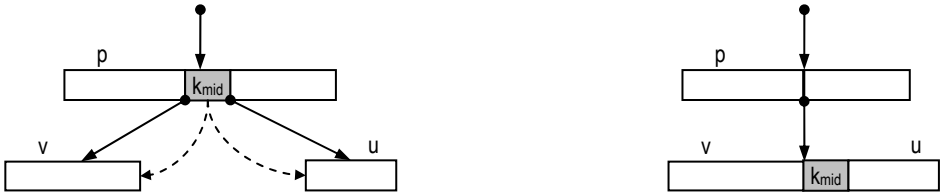
此时，如 图七.34 所示，可以从v中取出最小的关键码k_{min}，将父节点p中介于u和v之间的关键码k_{mid}替换为k_{min}，然后将key_{mid}插至u的最右侧。



图七.34 下溢的节点向父亲“借”一个关键码，父亲再向右兄弟“借”一个关键码

- 没有一个兄弟至少包含⌈m/2⌉个关键码

当然，即便如此，节点u仍至少有一个兄弟。如 图七.35 所示，不妨设它有左兄弟v，于是v中恰好包含⌈m/2⌉-1 个关键码。此时，可以取出父节点p中介于u和v之间的关键码k_{mid}，然后通过该关键码将节点v和u合并起来。



图七.35 下溢的节点向父亲“借”一个关键码，然后与左兄弟“粘接”成一个节点

综合以上三种情况可以看出：

观察结论七.11 在处理好一个下溢节点之后，其父节点中的关键码不会增加，但有可能会减少一个。

那么，要是由于关键码的减少进而导致父节点的下溢，又该如何处理呢？下一节将回答这一问题。

7.4.8 关键码的删除

为了从B-树v中删除一个关键码key，也首先需要调用 算法七.5，通过find(v, key)进行查找。倘若查找失败，则说明关键码key不存在，删除操作即告完成。否则，将关键码key所在的内部节点记作u。与 算法七.4 类似地，若u不是最底层的内部节点，则在key对应的左（右）子树中找到其直接前驱（后继），并与之交换位置。

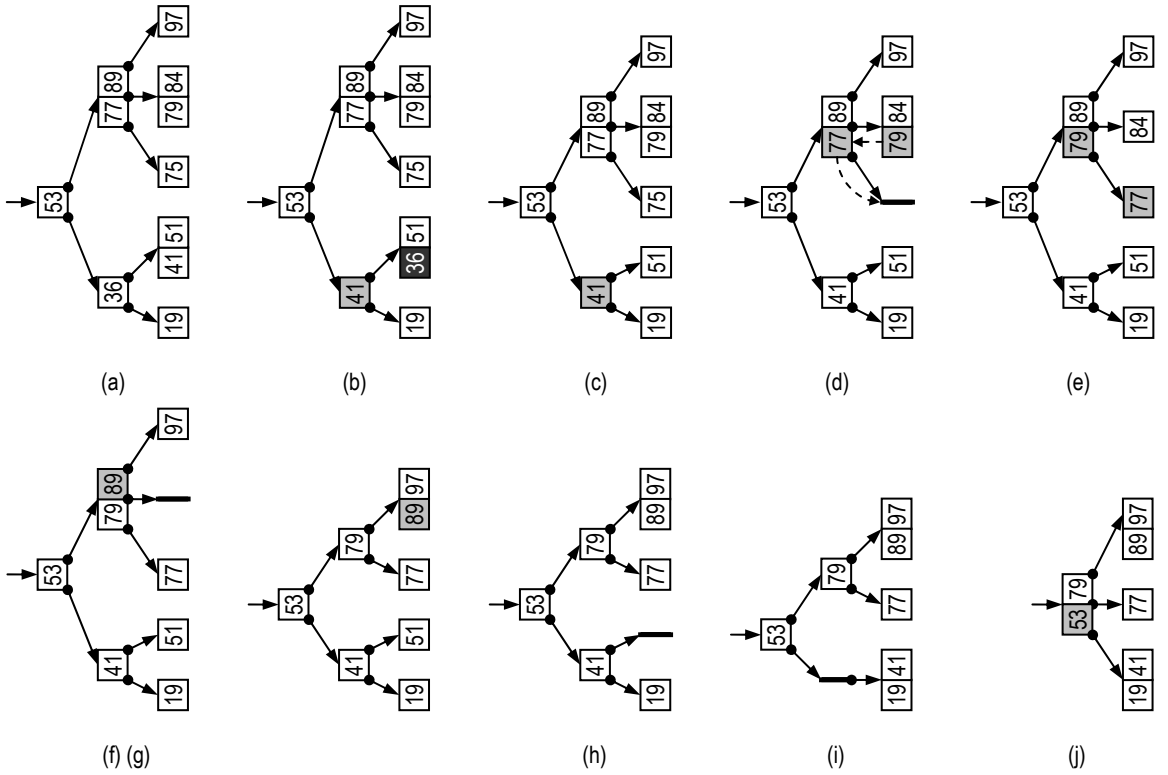
通过对换，总能保证节点u处于内部节点的最底层。现在，我们直接将key从u中删去。若此后节点u中关键码的数目依然合法（不少于$\lceil m/2 \rceil-1$ 个），则删除操作也可立即结束。否则，可以按照第 7.4.7 节介绍的方法来修复这个下溢的节点。

在向孩子“借出”一个关键码后，倘若父节点中的关键码也减少到$\lceil m/2 \rceil-2$ 个，我们就再次采用以上方法，对下溢的父节点进行修复。

当然，这种修复可能需要反复进行，直到当前节点不再下溢，或者已越过树根节点。

```
算法：remove(v, key)
输入：B-树节点v，关键码key
输出：若关键码key在以v为根节点的B-树内存在，则将其删除，并将这棵树重新调整为合法的B-树
{
  if (null == find(v, key)) return; //若关键码key在B树中不存在，则无法删除

  令u为key所在的内部节点;
  若u不是处于最底层，则将key与它的直接前驱（后继）关键码交换，以保证u处于最底层;

  从节点u中删除关键码key;
  若因此导致u下溢，则自下而上，不断修复各个下溢的节点，直到当前节点不再下溢或者已越过树根;
}
```

算法七.7 B- 树的关键码删除算法



图七.36 对3阶B-树(a)的删除操作

图七.36 给出了对一棵 3 阶B-树（图(a)）连续执行四次删除操作的过程。首先是删除 36：将该关键码与它的直接后继 41 交换后（图(b)），直接删除 36（图(c)）。接下来删除 75：直接删除 75 后（图(d)），从父节点中借来关键码 77，父节点再向右兄弟借来关键码 79（图(e)）。然后再删除 84：直接删除关键码 84（图(f)），然后从父节点借来关键码 89，再与右兄弟合并（图(g)）。最后删除 51：直接删除关键码 51（图(h)），然后从父节点借来关键码 41，再与左兄弟合并（图(i)），父节点进而再向祖父节点借来 53，再与父节点的右兄弟合并（图(j)）。

由上可见，为了删除一个关键码，无论是查找目标关键码还是修复下溢的节点，所需的时间都线性正比于树高，故有如下结论：

定理七.10 对存有 N 个关键字的 m 阶 B-树的每次删除操作，都可以在 $O(\log_m N)$的时间内完成。

第八章

排序

“天之道，损有余而补不足”，自然万物发展的规律，都是倾向于消除差异。无独有偶，热力学第二定律也指出：任一封闭的系统，都会朝着熵增加的方向发展。从信息论的角度来看，也就是倾向于更加无序。然而，“人之道，则不然，损不足以奉有余”，人总是偏好有序。就数据处理而言，有序性的确十分有用。以第 6.4.2 介绍的二分查找算法为例，只有对有序向量，才能保证每次都能在$\mathcal{O}(n\log n)$的时间完成查找。

所谓排序，就是按照某种次序，重新排列某一序列中的所有元素。为此，任意一对元素之间都应该能够比较大小，即在所有元素之间可以定义一个全序关系。排序不仅可以作为查找等操作的预处理计算，而且也是实际应用中需要反复进行的一项基本操作，因此本章将集中讨论排序的算法，并介绍排序算法的具体应用。

排序算法种类繁多。根据其处理数据的规模与存储特点，可分为内部排序和外部排序算法：前者处理的数据规模不大，内存足以容纳；后者处理的数据规模很大，必须将数据存放于外部存储器中，在处理过程的任何时刻，内存中只能容纳其中的一小部分数据。根据输入不同的形式，排序算法则可以划分为脱机算法与在线算法：在前一种情况中，待排序的数据是以批处理的形式给出的；而在网络计算之类的环境中，待排序的数据则是实时生成的，在排序算法开始运行时，数据并未完全就绪，而是随着排序算法本身的进行而逐步给出的。针对不同的体系结构，也需要采用不同的排序算法，由此又可以划分为串行和并行两大类排序算法。另外，根据排序算法是否采用随机策略，还有确定式和随机式之分。本章讨论的范围，主要集中于确定式、串行的、脱机的内部排序算法。

实际上，此前的章节已经介绍过若干排序算法：起泡排序（第 1.3.4 节）、选择排序（第 5.6.1 节）、插入排序（第 5.6.2 节）和堆排序（第 §5.9 节）。因此，本章将结合分支策略，集中讨论另外两种主流的排序算法——归并排序和快速排序——并给出基于比较式排序算法的复杂度下界。

§8.1 归并排序

8.1.1 分治策略

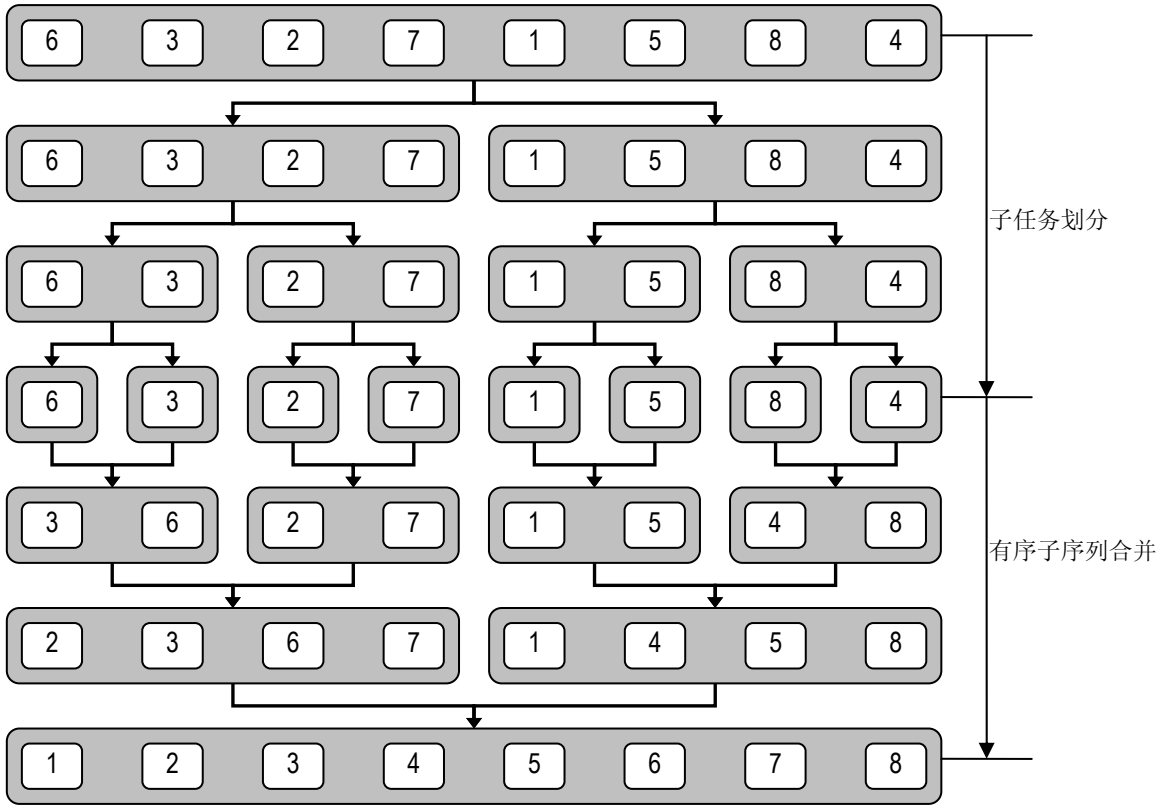
为了解决一个规模较大的问题，我们可以将其分解为两个子问题，并借助递归分别得到它们的解，然后将子问题的解合并成原问题的解——这就是分治（Divide-and-conquer）策略。由著名计算机科学家冯·诺依曼于 1945 年发明的归并排序算法，就是分治策略的一个典型应用。这一算法具体过程的描述为 算法八.1：

```
算法：mergeSort(S, n)
输入：n个元素组成的无序序列S
输出：经排序，S成为有序序列
{
  if (1 < n)  //不超过一个元素组成的序列S必然有序，故可直接返回S；否则
```

```
    S1 = S[0..⌈n/2⌉-1];      S2 = S[⌈n/2⌉..n-1];  //将S尽可能均匀地划分为前后两个子序列
    mergeSort(S1, ⌈n/2⌉);    mergeSort(S2, ⌊n/2⌋);  //通过递归，分别对两个子序列进行排序
    S = merge(S1, S2);  //将两个有序子序列归并得到有序序列S
  }
  return S;
}
```

算法八.1 归并排序算法

图八.1 给出了归并算法的一个实例。



图八.1 对一个由8个元素组成的序列进行归并排序

8.1.2 时间复杂度

为了保证分治策略的效率，首先必须保证子问题的划分及其解的合并都能快速完成，通常，都要求这两部分计算可以在线性时间内完成。另外，划分出来的子问题应该是相互独立的，也就是说，每个子问题的解不受其它子问题的影响。最后，子问题的规模不能相差悬殊，最好能够相等或者接近。

事实上，归并排序算法完全满足上述要求。首先，以居中的位置为界，只需$\mathcal{O}(n)$时间即可将待排序的序列均匀地划分为左、右两个子序列，而且各子序列的排序不受另一子序列的影响。更重要的是，正如我们很快就将在第 8.1.3 节看到的，根据这两个子序列各自的排序结果，可以在线性时间内获得整个序列的排序结果。

因此，若将归并排序算法处理长度为 n 的序列所需的时间记作 **T**(n)，则有如下递推关系成立：

$$\mathbf{T}(n) = 2 \times \mathbf{T}(n/2) + \mathcal{O}(n)$$

另外，当子序列长度缩短到 1 时，递归即可终止，并将该序列作为解直接返回。故有

$$\mathbf{T}(1) = \mathcal{O}(1)$$

由这两个条件可以导出：

$$\mathbf{T}(n) = \mathcal{O}(n\log n)$$

定理八.1 归并排序算法可以在 $\mathcal{O}(n\log n)$的时间内对长度为 n 的序列完成排序。

请读者运用第 1.5.1 节介绍的递归跟踪的方法，自行给出 定理八.1 的另一种证明。

8.1.3 归并算法

所谓归并操作，就是将两个有序子序列合并为一个整体有序的序列。从上面的分析可以看出，在线性时间内完成归并，是归并排序算法能够达到 $\mathcal{O}(n\log n)$复杂度的关键（由该算法的命名也可以看出这一点）。那么，如何才能做到这一点呢？

下面，我们针对序列的两种典型形式——向量与列表——分别给出具体的算法。

■ 有序向量的归并

首先考虑两个有序子向量的归并，具体算法如 算法八.2 所示。

```
算法：mergeVector(S1, S2)
输入：两个非降的有序子向量S1和S2，长度分别为n和m
输出：将S1和S2归并为非降向量S
{
0.  初始化一个长度为n+m的向量S;
1.  r1 = n-1;   r2 = m-1;  //两个子向量末元素的秩
2.  while ((0 ≤ r1) or (0 ≤ r2)) {
    //在两个子向量变为空之前，不断地摘出两个末元素中的大者e
3.    if (0 > r1)                                    e = S2.removeAtRank(r2--);
4.    else if (0 > r2)                               e = S1.removeAtRank(r1--);
5.    else if (S1.getAtRank(r1) < S2.getAtRank(r2))  e = S2.removeAtRank(r2--);
6.    else                                           e = S1.removeAtRank(r1--);
    //并用该元素更新S中相应的元素
7.    S.replaceAtRank(2+r1+r2, e);
    }
8.  返回S;
}
```

算法八.2 有序向量的归并

引理八.1 算法 mergeVector()可在 $O(n+m)$时间内完成对长度分别为 n 和 m 的两个有序向量的归并。

〖证明〗

即证明 算法八.2 的运行时间为$O(n+m)$。

首先，第 0 句创建一个长度为 n+m 的向量，显然只需 $O(n+m)$时间。

第 1 句只需 $O(1)$时间。

第 2 到 7 句是一个循环。每经一次迭代，S_1 和 S_2 的总长度都会缩短一个单位，因此共需迭代 n+m 次。

每次迭代中，只需进行常数次比较操作、getAtRank()操作、replaceAtRank()操作和removeAtRank()操作。根据第 3.1.2 节对向量操作的复杂度分析(表三.4),每一getAtRank()和replaceAtRank()操作只需$O(1)$时间。虽然向量元素的删除操作removeAtRank()在最坏情况下需要$O(n)$时间，但是由于这里删除的总是向量的最后一个元素，无需对后续元素做前移处理，故也可以$O(1)$时间内完成。综上所述，每次迭代都可以在常数时间内完成，整个循环只需$O(n+m)$时间。

本引理得证。 ❑

■ 有序列表的归并

接下来考虑两个有序子列表的归并，具体算法如 算法八.3 所示。

```
算法: mergeList(S1, S2)
输入: 两个非降的有序列表S1和S2, 长度分别为n和m
输出: 将S1和S2归并为非降列表S
{
0.初始化一个空的列表S;
1.while (!S1.isEmpty() or !S2.isEmpty()) {
  //在两个子列表变为空之前, 不断地摘出两个首元素中的小者e
2.  if (S1.isEmpty())                                        e = S2.remove(S2.first());
3.  else if (S2.isEmpty())                                     e =
S1.remove(S1.first());
4.  else if (S1.first().element() > S2.first().element()) e = S2.remove(S2.first());
5.  else                                                     e = S1.remove(S1.first());
  //将该元素插至S的尾部
6.  S.insertLast(e);
  }
7.返回S;
}
```

算法八.3 有序列表的归并

引理八.2 算法 mergeLsit()可在 $O(n+m)$时间内完成对长度分别为 n 和 m 的两个有序列表的归并。

〖证明〗

与 引理八.1 的证明类似，关键在于说明在第 1 到 6 句循环中的每次迭代只需常数时间。

我们注意到，这样的每次迭代只需进行常数次比较、isEmpty()、first()、element()、remove()和insertLast()操作，而根据第 3.2.4 节的结论，只要基于双向链表实现列表，这些操作每次只需$O(1)$时间。

由此本引理得证。 ❑

8.1.4 Mergesort 的 Java 实现

基于如 代码一.1 所示的排序器接口，我们给出归并排序算法的具体实现如 代码八.1 所示：

```
/*
 * 归并排序算法
 * 针对向量和列表两种情况，merge()方法的具体实现不同
 * 这里只针对基于列表的序列，给出算法实现
 */

package dsa;

public class Sorter_Mergesort implements Sorter {
  protected Comparator C;

  public Sorter_Mergesort()
  { this(new ComparatorDefault()); }

  public Sorter_Mergesort(Comparator comp)
  { C = comp; }

  public void sort(Sequence S) {//Mergesort
    int n = S.getSize();
    if (1>=n) return;//递归基
    Sequence S1 = new Sequence_DLNode();
    Sequence S2 = new Sequence_DLNode();
    while (!S.isEmpty()) {//将S均匀地分成两个子序列S1和S2
      S1.insertLast(S.remove(S.first()));
      if (!S.isEmpty()) S2.insertLast(S.remove(S.first()));
    }
    sort(S1); sort(S2);
    merge(S, S1, S2);
  }

  public void merge(Sequence S, Sequence S1, Sequence S2) {//有序列表的归并算法
    while (!S1.isEmpty() || !S2.isEmpty()) {
      Object e;
      //在两个子列表变为空之前，不断地摘出两个首元素中的小者e
      if (S1.isEmpty())
```

```
        e = S2.remove(S2.first());

      else if (S2.isEmpty())

        e = S1.remove(S1.first());
      else if (0 < C.compare(S1.first().getElem(), S2.first().getElem()))
        e = S2.remove(S2.first());
      else
        e = S1.remove(S1.first());
    //将该元素插至S的尾部
    S.insertLast(e);
    }//while
  }
}
```

代码八.1 归并排序的实现

请注意，这里只给出了针对列表结构的实现，至于向量结构所对应算法的实现，请读者参照以上实现自行补充。

§8.2 快速排序

8.2.1 分治策略

快速排序是分治策略的又一典型应用，该算法是由C. A. R. Hoare[㈠]于 1960 年发明的。

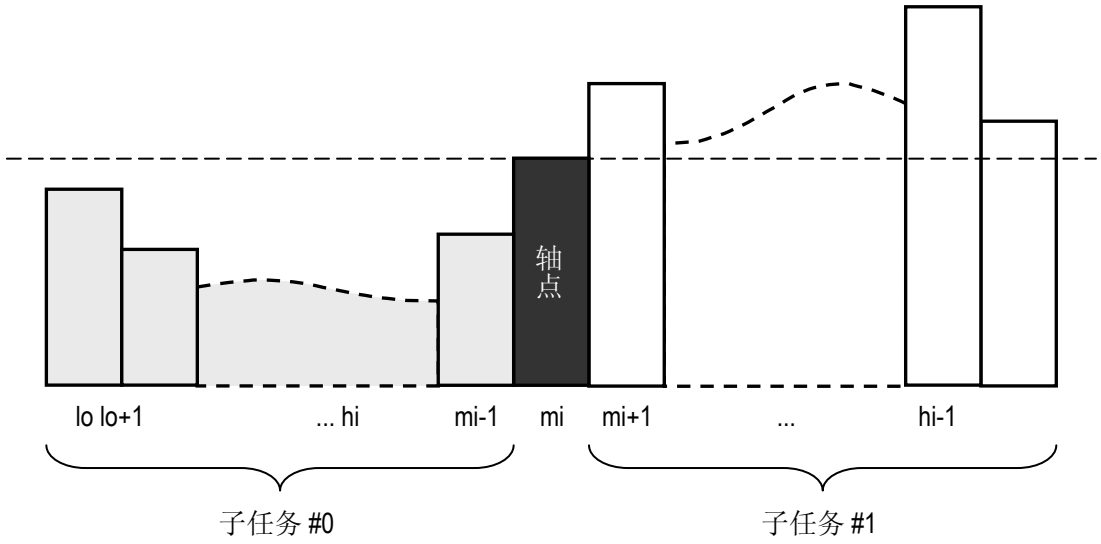
归并排序算法的主要计算量集中于有序子序列的归并，而快速排序算法正好相反，它可以在$\mathcal{O}(1)$时间内由子问题的解直接得到原问题的解，但为了将原问题划分为两个子问题，快速排序算法却需要$\mathcal{O}(n)$时间。特别值得指出的是，在进行字任务划分时，虽然快速排序算法能够确保子任务的相互独立性，但并不能保证子任务的规模大体相当，甚至有可能极不平衡，因此并不能保证$\mathcal{O}(n\log n)$的最坏情况复杂度。尽管如此，由于该算法易于实现，而且其平均时间复杂度足够低（正如该算法的名字所暗示的），它仍然受到人们的青睐，在实际应用中往往成为首选的排序算法。

8.2.2 轴点

在每个长度不小于 3 的序列S[lo..hi]中，对于任何lo < mi < hi，以每一个元素p = S[mi]为界，都可以将该序列分割为前、后两个子序列S_1 = S[lo..mi-1]和S_2 = S[mi+1..hi]，如 图八.2 所示。

定义八.1 若 S_1 中元素均不大于 p，S_2 中元素均不小于 p，则元素 p 称作序列 S 的一个轴点（pivot）。

㈠ Charles Antony Richard Hoare 爵士，英国计算机科学家，生于 1934 年 1 月 11 日，1980 年图灵奖得主。



图八.2 序列的轴点（这里用高度来表示各元素的大小）

不难看出，若果真能找到这样一个轴点，则它可以将原序列“天然地”划分为前、后两个相互独立的子序列，而且一旦子序列完成排序，即可立即（在 $\mathcal{O}(1)$时间内）得到整个序列的排序结果。快速排序算法正是递归地利用轴点的这一性质，以实现分治策略。

8.2.3 划分算法

■ 不含轴点的序列

然而事情并非如此简单，因为并非任一序列都含有轴点。为看出这一点，可以注意到这样一个事实：

观察结论八.1 若某一元素若是轴点，则经过排序之后，它的位置不应发生变化。

因此，只要在初始序列中所有元素都是错位的，则每个元素都不可能是轴点。比如，在 图八.3 中，我们只需将有序序列的各元素循环右移一个单元，即可得到一个不含轴点的序列。

59	3	7	11	15	19	23	27	31	35	39	43	47	51	55

图八.3 一个不含轴点的序列

■ 构造轴点

尽管如此，通过调整序列中元素的位置，依然可以“人为地”构造出一个轴点。具体过程如 算法八.4 所示。

```
算法：createPivot(S, lo, hi)
输入：序列S[lo..hi]
输出：调整S中各元素的位置，在S中构造出一个轴点，返回该轴点的秩
{
  while (lo < hi) {
```

```
    while ((lo < hi) && (S[lo] ≤ S[hi]) hi--;   swap(S[lo], S[hi]);

    while ((lo < hi) && (S[lo] ≤ S[hi]) lo++;   swap(S[lo], S[hi]);

  }
  return lo;
}
```

算法八.4 轴点构造算法

■ 正确性及效率

引理八.3 按照 算法八.4，可以在任一序列S[lo..hi]中构造出一个轴点，为此只需花费$\mathcal{O}$(hi-lo+1)时间。

〖证明〗

我们首先证明，算法八.4 的确可以构造出一个轴点。

为此，我们只需注意到该算法中的主循环满足以下不变性：

❶ 每次迭代后，lo 不可能减小，hi 也不可能增大，而且二者不可能同时保持不变；

❷ S[lo]左侧的任一元素都不大于 S[lo]，而 S[hi]右侧的任一元素都不小于 S[hi]。

另外易见，在循环结束、即将返回 lo 时，lo 和 hi 必相等。于是由不变性❷可知：S[lo]左侧的任一元素都不大于 S[lo]，而 S[lo]右侧的任一元素都不小于 S[hi]——也就是说，在算法结束时，S[lo]正是序列的一个轴点。

接下来，我们证明 算法八.4 只需运行$\mathcal{O}$(hi-lo+1)时间。

该算法的运行时间消耗在两个方面：移动 hi 和 lo，以及 swap()操作。上面已经指出，在算法结束时有 lo = hi 。因此，根据不变性❶，算法的主循环至多迭代 $\mathcal{O}$(hi-lo+1)次，也就是说，swap()操作总共只需 $\mathcal{O}$(hi-lo+1)时间。

另外，由于 hi 和 lo 始终都是单向（相向）移动，直至最后相遇，因此总共需要移动 hi-lo 次。 ❑

8.2.4 Quicksort 的 Java 实现

根据以上思路，可以实现如 代码八.2 所示的快速排序算法。

```
/*
 * 快速排序算法
 * 针对向量和列表两种情况，createPivot()方法的具体实现不同
 * 这里只针对基于向量的序列，给出算法实现
 */

package dsa;
```

```
public class Sorter_Quicksort implements Sorter {
  private Comparator C;

  public Sorter_Quicksort()
  { this(new ComparatorDefault()); }

  public Sorter_Quicksort(Comparator comp)
  { C = comp; }

  public void sort(Sequence s)//入口方法
  { qsort(s, 0, s.getSize()-1); }

  public void qsort(Sequence S, int lo, int hi) {//Quicksort
    if (lo >= hi) return;
    int mi = createPivot(S, lo, hi);
    qsort(S, lo, mi-1);
    qsort(S, mi+1, hi);
  }

  public int createPivot(Sequence S, int lo, int hi) {//确定轴点
    while (lo < hi) {
      while ((lo < hi) && (C.compare(S.getAtRank(lo), S.getAtRank(hi)) <= 0)) hi--;
      swap(S, lo, hi);
      while ((lo < hi) && (C.compare(S.getAtRank(lo), S.getAtRank(hi)) <= 0)) lo++;
      swap(S, lo, hi);
    }
    return lo;
  }

  private void swap(Sequence S, int i, int j) {//交换S[i]和S[j]
    Object temp = S.getAtRank(i);
    S.replaceAtRank(i, S.getAtRank(j));
    S.replaceAtRank(j, temp);
  }
}
```

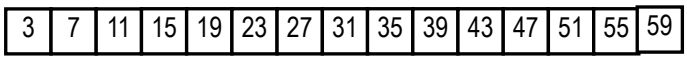
代码八.2 快速排序的实现

请注意，这里的 createPivot()方法只适用于基于向量的序列，在处理基于列表的序列时，这一实现的效率将很低。但实际上，无论是哪种类型的序列，都可以在线性时间内完成 createPivot()操作。请读者针对基于列表的序列，自行实现相应的 createPivot()方法。

8.2.5 时间复杂度

■ 最坏情况

引理八.3 告诉我们，采用 算法八.4，可以在线性时间内将原排序问题分解为两个相互独立的子问题，而且由各自排序后的子序列，可以在$\mathcal{O}(1)$时间内得到最终的有序序列。正如此前所指出的，这些都是分治策略达到高效率的必要条件。



图八.4 单调序列——轴点构造算法的最坏输入

不幸的是，另一个重要的必要条件——子任务规模接近——却无法保证。事实上，在此子任务的规模可能相差很悬殊。比如，读者可以自行验证，对于 图八.4 中的单调序列，若按照 算法八.4 的策略，总是将最左侧的元素选作轴点，则其后果是，总有一个子序列为空，而另一个与原序列几乎等长。

在这种情况下，如果将快速排序算法处理长度为 n 的序列所需的时间记作 $\mathbf{T}(n)$，那么就有如下递推关系成立：

$$T(n) = T(0) + T(n-1) + \mathcal{O}(n)$$

另外，当子序列长度缩短到 1 时，递归即可终止，并将该序列作为解直接返回。因此有

$$T(1) = \mathcal{O}(1)$$

由这两个条件可以导出：

$$T(n) = \mathcal{O}(n^2)$$

（请读者运用第 1.5.2 节介绍的递归跟踪法导出这一复杂度。）

因此，就最坏情况的运行时间而言，快速排序算法似乎并不名副其实。

■ 中值

实际上，对于如 图八.4 所示的单调序列，算法八.4 每次都是挑选出最左侧的元素，然后通过交换，将该元素移动到合适的位置，使之最终成为一个轴点。问题在于，倘若该元素就是当前的最小（或接近最小）的元素，则 算法八.4 必然会划分出两个规模相差悬殊的子任务。反之，只有当最初挑选的元素比较接近中间值（median）时，划分出的子任务规模才会接近。问题在于，如何很快找到这样一个元素呢？

当然，首先可以借助有关选取（Selection）的算法，在线性的时间内，从序列中找到一个接近中间值的元素。遗憾的是，限于篇幅，本书只能省去这部分内容。

还有一种简便易行的随机方法：每次从序列中随机地取出三个元素，挑选其中大小居中的元素，然后通过交换操作，使之成为轴点。这一策略不仅易于实现，而且效果很好。统计与分析都表明，采用这一策略后，最坏情况出现的概率将大大降低。

■ 平均运行时间

尽管从理论上分析，快速排序算法在最坏情况下需要运行平方量级的时间，但无论是实验统计还是更为细致的分析都指出，该算法的平均效率依然可以达到 $O(n\log n)$，这多少有些出人意料。而且，较之其它排序算法，快速排序算法时间复杂度中的常系数更小。加上其思路的直观以及易于实现的特性，自诞生起，快速排序算法一直受到人们的青睐。

§8.3 复杂度下界

在时间复杂度的各种指标中，“最坏情况复杂度”往往是最重要的，在一些特殊的场合中，甚至成为唯一的指标。比如对于计算机辅助的核电站、神经外科手术等系统而言，系统的平均响应时间、分摊响应时间都不具有任何意义。在这类应用中，人们更加关注的是系统在最坏情况下的响应速度——即使出现最坏情况，这类系统也必须能够迅速进行处理，否则后果不堪设想。

以排序算法为例，我们已经介绍过起泡排序（第 1.1.3 节）、选择排序（第 5.6.1 节）、插入排序（第 5.6.2 节）、堆排序（第 §5.9 节）、归并排序（第 §8.1 节）和快速排序（第 §8.2 节），就最坏情况的性能而言，这些算法都至少需要运行$\Omega(n\log n)$的时间。能否更快呢？

根据第 1.4.3 节的分析，基于任何一种特定的计算模型，任何一个问题的任一算法（如果存在的话）的时间复杂度都不可能低于某个最低值，称作该问题的复杂度“下界”。以“排序”这一问题为例，若限于采用比较树模型，其复杂度下界为$\Omega(n\log n)$。这就意味着，就最坏情况复杂度的意义而言，基于这一计算模型的任一$O(n\log n)$算法都是最优的。

下面我们就来证明这一结论。

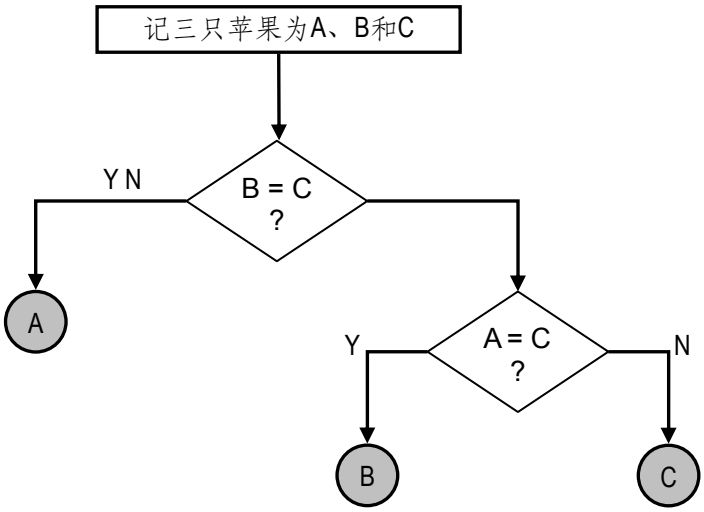
8.3.1 比较树与基于比较的算法

让我们考虑这样一个问题：三只苹果外观一样，其中的两只重量相同，另一只不同，如何找出重量不同的那只苹果？

如果可以使用一台不带刻度、只能判别是否重量相等的天平，我们可以采用以下算法：

```
算法：FindApple(A, B, C)
输入：三只苹果A、B和C，其中两只重量相同，另一只不同
输出：找出其中重量不同的那只苹果
{
  称量B和C;
  若B和C重量相等，则返回A;
  称量A和C;
  若A和C重量相等，则返回B;
  否则，返回C;
}
```

算法八.5 从三个苹果中选出重量不同者的算法



图八.5 从三只苹果中挑出重量不同者

这一算法的流程可以直观地表示为 图八.5 的形式。更准确地讲，图八.5 给出了 算法八.5 所有可能的执行流程。我们不难看出，这一表示形式具有以下特点：

① 该算法执行情况的所有可能，可以表示为一棵二叉树；

② 该树中每一内部节点分别对应于一次称量比较，每一内部节点的左、右孩子，分别对应于算法针对两种称量结果（是否等重）的执行方向；

③ 叶子节点对应于算法的终止以及相应的输出，算法的每一运行过程分别对应从根节点到某匹叶子节点的一条通路。

这样的一棵树，称作比较树（Comparison tree）。如果某一算法中的所有分支流程都是像 算法八.5 那样，完全取决于两个变量（常量）的比较结果，就可以用一棵比较树来表示。当然，通常的比较稍有不同：不仅可以判断两个量是否相等，还可以判断它们之间的大小关系。在这种情况下，对应的比较树就是一棵三叉树，即每个内部节点可以有三个孩子（分别对应小于、大于和相等三种情况）。

反过来，可以描述为比较树的算法，也被称作基于比较的算法（Comparison-based algorithms）。在本书所介绍的排序算法，都属于基于比较的算法。

8.3.2 下界

设 A 为任一基于比较的算法，**T**(A)为与之对应的比较树。

根据比较树的上述性质，不难得出如下结论：

观察结论八.2 在最坏情况下，算法 A 的运行时间不会低于Ω(h(T(A)))。其中 h(T(A))为 T(A)的高度。

特别地，针对基于比较的排序算法，有以下结论：

定理八.2 任一基于比较的排序算法，在最坏情况下至少需要运行Ω(n*log*n)时间，其中 n 为待排序元素的数目。

〖证明〗

假设任一基于比较的算法 A 可以正确地进行排序，其对应的比较树为 **T**(A)。

我们首先证明：若以这 n 个元素的任意两个排列分别作为输入运行算法 A，则两次运行不可能终止于 **T**(A)中的同一匹叶子。

对于任意两个排列λ_1和λ_2，我们总可以找到两个互异的元素 x_1 和 x_2，使得在λ_1 中 x_1 出现在 x_2 之前，而在λ_2 中 x_1 出现在 x_2 之后。倘若算法 A 在处理λ_1 和λ_2 时都终止于同一匹叶子，则说明该算法对这两种排列未加区分，故必然对其中某一排列的排序结果是错误的。这与算法的正确性相悖。

因此，**T**(A)中叶子节点的数目绝不可能少于n个元素的排列总数——n!。而具有Ω(n!)匹叶子的三叉树，其高度至少是$\Omega(\log_3(n!)) = \Omega(n\log n)$。故由 观察结论八.2，算法A的运行时间不会低于Ω(n**log**n)。 ❑

既然本书所介绍的排序算法都属于基于比较的算法，故根据 定理八.2，它们的最坏情况复杂度不可能低于Ω(n**log**n)，从这个意义上讲，堆排序（第 §5.9 节）、归并排序（第 §8.1 节）已经是最优的了。

需要强调的是，定理八.2 所给出的Ω(n**log**n)下界是针对比较树模型而言的。事实上，还存在很多不属于此类的排序算法（比如桶排序算法和基数排序算法），它们不能用这一模型进行描述，这些算法的最坏情况时间复杂度有可能低于这一下界。

第九章

计算机已经成为我们日常学习、办公不可或缺的工具，反过来，为了提高学习和工作的效率，对计算机信息处理能力的要求也越来越高，其中文本处理的能力就是极其重要的一个方面。今天，越来越多的文档、资料开始以数字化的形式出现，互联网所提供的信息总量也在以惊人的速度膨胀，因此在文本处理方面，目前的一个突出问题就是如何高效地处理海量的文本数据，而串（String）则是最常见的一种文本形式。

相对于此前介绍的数据结构，串中各元素的内容相对简单，分别称作一个字符。所有字符排成一个线性结构，它们都来自于某个集合，称作字符表。字符表的规模往往不大，以最常见的 ASCII 字符表为例，只有 128 个元素。然而，就其规模而言，串结构一般都远远超过其它结构，故我们也把串的规模称作其“长度”。因此，串中字符的重复率一般非常高。串作为一种数据结构的存在意义与价值，正体现在这一结构的这些特点。

鉴于串结构的特点，本章虽然也会提及该结构的 ADT 描述，但不再对其实现进行讨论，而是直接利用 Java 本身提供的 String 类。本章的重点，将放在串匹配算法上面。

§9.1 串及其 ADT

■ 串

我们今天所面对的文本数据，大多是以串的形式出现。串是由有限个字符组成的一种线性结构，其中每个字符都来自某个字符表（Alphabet）Σ，比如 ASCII 字符集或 Unicode 字符集。

■ ADT

串 ADT 的主要操作可以归纳为如下：

表九.1 串ADT支持的操作

操作方法	功能描述
length():	查询串的长度 输入：无 输出：非负整数
charAt(i):	返回第 i 个字符 输入：一个非负整数 输出：一个字符
substr(i, k):	返回从第 i 个字符起、长度为 k 的子串 输入：两个非负整数 输出：一个字符串
prefix(k):	返回长度为 k 的前缀 输入：一个非负整数 输出：一个字符串

操作方法	功能描述
suffix(k):	返回长度为 k 的后缀 输入：一个非负整数 输出：一个字符串
equals(T):	判断 T 是否与当前字符串相等 输入：一个字符串 输出：一个布尔标志
concat(T):	将 T 串接在当前字符串之后 输入：一个字符串 输出：一个字符串
indexOf(P):	若 P 是当前字符串的一个子串，则返回该子串的起始位置；否则返回-1 输入：一个字符串 输出：一个非负整数

比如，从串 S = "data structures"开始，依次执行如下操作，相应的结果如所示：

表九.2 串操作实例

操作	输出	字符串 S
length() 15		"data structures"
charAt(5) '	s'	"data structures"
prefix(4) "data"		"data structures"
suffix(10) "s	tructures"	"data structures"
concat("and algorithms")	"data structures and algorithms"	"data structures and algorithms"
equals("data structures")	false	"data structures and algorithms"
equals("data structures and algorithms")	true	"data structures and algorithms"
indexOf("string")	-1	"data structures and algorithms"
indexOf("algorithm")	20	"data structures and algorithms"

■ 特点

串具有两个突出的特点：结构简单，规模庞大。所谓结构简单，是指字符表规模不大，在某些应用问题中，字符表的规模甚至可能极小。以生物信息序列为例，组成蛋白质（文本）的氨基酸（字符）只有约 20 种，而组成DNA序列（文本）的碱基（字符）则只有 4 种。既然文本串也是线性结构，当然可以直接利用 第三章所介绍的向量或列表等序列结构来实现，因此就数据结构本身而言，串没有太多问题需要讨论。这里我们将直接采用Java本身提供的String类。

然而，这类文本的规模往往很大，其中每个字符都大量重复地出现。在处理这样的数据时，时间效率就成为了一个特出的问题，需要仔细研究。本章将集中讨论 indexOf()方法的高效算法。

§9.2 串模式匹配

9.2.1 概念与记号

在展开后续的讨论之前，有必要对与串相关的若干概念与记号做一介绍。一般地，

定义九.1 由 n 个字符构成的串记作 S = "$a_0\ a_1\ \ldots\ a_{n-1}$"，其中 $a_i \in \Sigma$。这里的 Σ 是所有可用字符的集合，称作字母表（Alphabet）。

比如，对于通常的文本处理，Σ 就是 ASCII 字符集或者 Unicode 字符集。又如，在生物信息学中，Σ 则可能是所有的碱基（DNA 序列），或者所有的氨基酸（蛋白质序列）。再如，若串是以二进制文件形式给出的，则字母表 Σ={0, 1}。

定义九.2 n 称为 S 的长度，记作|S| = n。

我们只考虑长度有限的串，即 $n < \infty$。特别地，

定义九.3 长度为零的串称为空串（Null string）。

在对串进行处理时，经常需要取出其中连续的某一片段。

定义九.4 所谓 S 的子串（Sub-string），就是起始于任一位置 i 的连续 k 个字符，记作 substr(S, i, k) = "$a_i\ a_{i+1}\ \ldots\ a_{i+k-1}$"，$0 \le i < n$，$0 \le k$。

有两种特殊的子串。

定义九.5 起始于位置 0、长度为 k 的子串称为前缀（Prefix），记作 prefix(S, k) = substr(S, 0, k)。

定义九.6 终止于位置 n-1、长度为 k 的子串称为后缀（suffix），记作 suffix(S, k) = substr(S, n-k, k)。

根据上述定义，可以得出以下结论：

观察结论九.1 空串是任何串的子串，也是任何串的前缀、后缀。

观察结论九.2 任何串都是自己的子串，也是自己的前缀、后缀。

因此，空串与串本身都需要特殊指称：

定义九.7 空串以及串本身亦称作平凡子串（前缀、后缀）。

定义九.8 空串以及非平凡子串（前缀、后缀）称作真子串（前缀、后缀）。

最后，我们给出两个串相等的条件：

定义九.9 串 S = "$a_0\ a_1\ a_2\ \ldots\ a_{n-1}$"和 T = "$b_0\ b_1\ b_2\ \ldots\ b_{m-1}$"称作相等，当且仅当二者长度相等，且对应的字符分别相同，即 n = m 且对任何 $0 \le i < n$ 都有 $a_i = b_i$。

9.2.2 问题

■ 应用与定义

在串文本的众多应用问题中，会反复涉及到一项非常基本的判断性操作：给定串 T（称作主串）和串 P（称作模式串），T 中是否存在的某个子串与 P 相同？如果存在，找到该子串在 T 中的起始位置。

比如，Unix Shell中的命令grep和DOS中的命令find，功能与此类似[1]。又如在生物信息的处理过程中，也经常需要在蛋白质序列中寻找特定的氨基酸模式，或者在DNA序列中寻找特定的碱基模式。再如，邮件过滤器会扫描电子邮件的地址、标题及正文，并根据事先定义的特征串判定是否为垃圾邮件。还有，反病毒系统也会对下载的或将要执行的程序进行扫描，根据事先提炼出的特征串判定是否含有病毒。

这类操作，都属于串模式匹配（String pattern matching）的范畴，简称串匹配。比如，若主串和模式串分别是

T = "Now is the time for all good people to come"

P = "people"

则匹配的位置应该是 29。

记主串 T 的长度为|T| = n，记模式串的长度|P| = m。通常，m 和 n 都是很大的整数，而且相对来说 n 更大，即满足 $2 \ll m \ll n$。比如，n = 100,000，m = 100。

■ 分类

实际上，根据具体应用的不同，串匹配问题有多种形式。有些场合属于串匹配检测（Pattern detection）问题：我们只关心是否存在匹配，而不关心具体的匹配位置。有些场合则属于定位（Pattern location）问题：若经判断的确存在匹配，则还需要确定具体的匹配位置。有些场合属于计数（Pattern counting）问题：倘若有多处匹配，统计出这些匹配子串的总数。有些场合则属于枚举（Pattern enumeration）问题：在有多处匹配时，报告出所有匹配的具体位置。

比如，以上邮件过滤器的例子就属于检测型问题：一旦特征匹配，即可判定为垃圾邮件，从而直接删除，或者将其隔离以待用户确认，此时我们并不关心特征串的具体位置。然而，反病毒系统的任务则属于枚举型问题：不仅必须在二进制代码中找出所有的病毒特征串，还需要报告它们的具体位置，以便修复。

---

[1] 这两个命令的格式分别是：

```
% grep <pattern> <file>
c:\> find "pattern" <file>
```

可见，二者都是通过文件形式来指定待查找的主串。

9.2.3 算法效率的测试与评价

串匹配是一个经典的问题，有名字的算法不下三十余种。由于串结构自身的特点，在设计和分析此类算法时也需做特殊的考虑，其中首先需要回答的一个问题就是：如何评估某一串匹配算法的性能？

多数读者首先会想到，能否采用通常评估算法性能的策略，假设主串 T 和模式串 P 都是随机生成的，然后分析它们的各种组合所需的时间呢？很遗憾，答案是否定的。

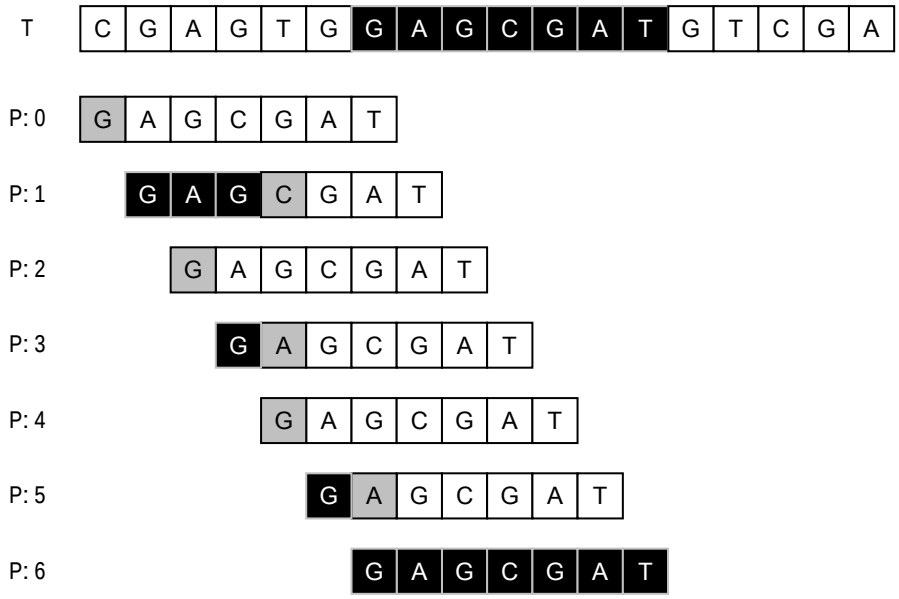
以基于字母表 $\Sigma=\{0, 1\}$ 的二进制串为例。在长度为 n 的主串中，长度为 m 的子串的数目为 n-m+1，当 $m \ll n$ 时，这个数目将接近于 n；另一方面，长度为 m 的二进制串数目为 2^m。因此，如果模式串也是随机选取的，则匹配成功的概率为 $n/2^m$。以 n = 100,000、m = 100 为例，这一概率只有 $100{,}000/2^{100} < 10^{-25}$。对于更长的模式串、更大的字母表，这一概率还将更低。因此，这一策略无法对算法的性能作出客观的评价。

可行的一种策略是，随机选取主串 T，但仅仅测试那些匹配成功的情况。为此，可以从 T 中随机取出长度为 m 的子串作为 P。这也是本书将采用的评价标准。

§9.3 蛮力算法

9.3.1 算法描述

蛮力串匹配算法是最直接、直观的方法。我们想象着将主串和模式串分别写在两条印有等间距方格的纸带上，主串对应的纸带固定，模式串的首字符与主串的首字符对齐，沿水平方向放好。如图九.1 所示，主串的前m个字符将与模式串的m个字符两两对齐。接下来，自左向右检查对齐的每一对字符：如果匹配，则转向下一对字符；如果失配，则说明在这个位置主串与模式串无法匹配，于是将模式串对应的纸带右移一个字符，然后从首字符开始重新对比。若经过检查，当前的m个字符对都是匹配的，则匹配成功，并返回匹配子串的位置。



图九.1 串模式匹配：蛮力算法

模式串中，黑色方格为经检查与主串匹配的字符，灰色方格为失配的字符，白色方格为无需检查的字符

9.3.2 算法实现

代码九.1 给出了蛮力算法的具体实现。

```
/*
 * 串模式匹配：蛮力算法
 * 若返回位置i > length(T) - length(P)，则说明失配
 * 否则，i为匹配位置
 */
import dsa.*;
import java.io.*;

public class PM_BruteForce {
////////////////////////////////////////////////////////////////////////
// T: 0   1   .   .   .   i   i+1 .   .   .   i+j .   .   n-1
//    --------------------|-------------------|------------
// P:                     0   1   .   .   .   j   .   .
//                        |-------------------|
////////////////////////////////////////////////////////////////////////
  public static int PM(String T, String P) {
    int   i;//模式串相对于主串的起始位置
    int   j;//模式串当前字符的地址

    for (i=0; i <= T.length()-P.length(); i++) {//主串从第i个字符起，与
      for (j=0; j<P.length(); j++) {//模式串的当前字符逐次比较
        if (T.charAt(i+j) != P.charAt(j)) break;//若失配，模式串右移一个字符
      }
```

```
      if (j >= P.length())  break;//找到匹配子串

    }


    return(i);
  }
}
```

代码九.1 蛮力串匹配算法的实现

9.3.3 算法分析

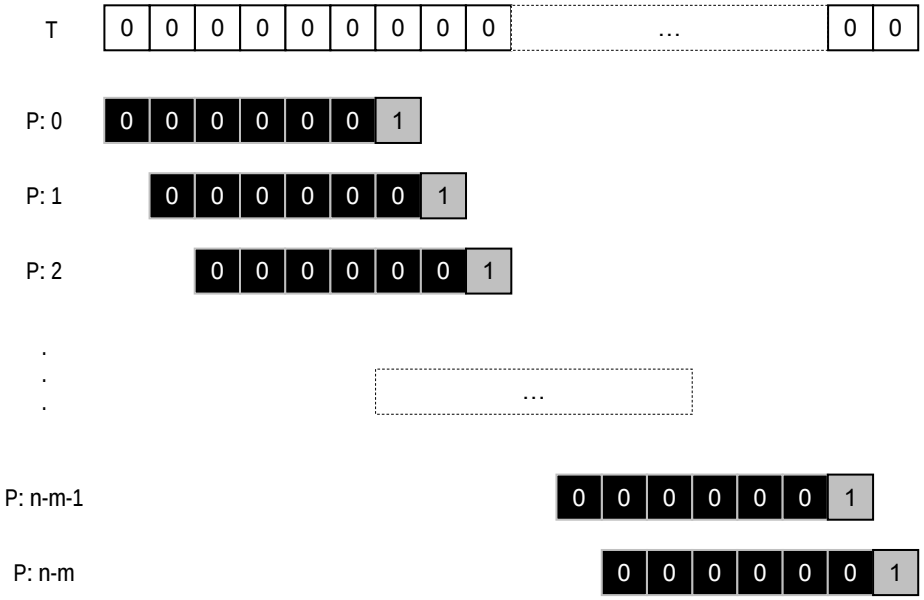
■ 正确性

我们首先来证明蛮力算法的正确性。首先，只有在某一轮的 m 次比较全部成功后，蛮力算法才会报告“发现匹配”，故该算法不会误报。另一方面，在发现匹配子串之前，蛮力算法会逐一尝试每一个可能匹配的位置，因此只要主串中存在与模式串匹配的子串，蛮力算法必然迟早会将其中最靠前的匹配子串报告出来，也就是说，该算法也不会漏报。

■ 运行时间

接下来分析该算法的时间复杂度。

我们注意到，从理论上讲，该算法至多需要迭代 n-m+1 轮，而且每一轮至多需要进行 m 次比较，也就是说，至多只需进行(n-m+1)×m 次比较。那么，这种最坏情况的确会发生吗？



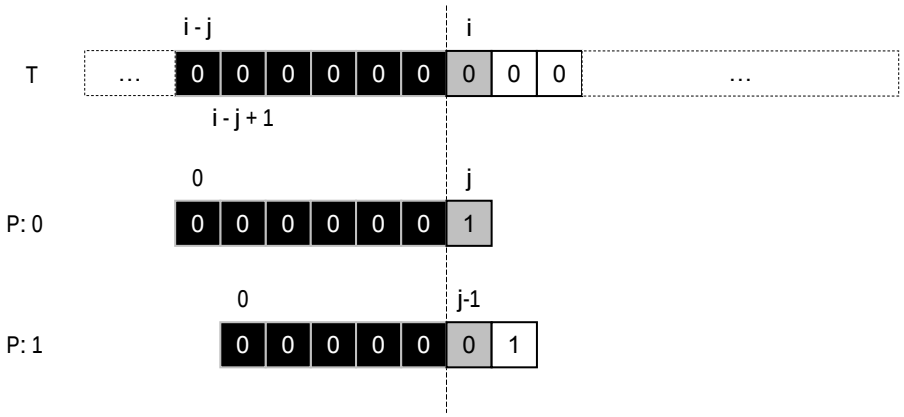
图九.2 蛮力算法的最坏情况

如 图九.2 所示的实例给出了肯定的回答。如果主串由n个'0'组成，而模式串除末字符为'1'外其它字符也都是'0'，那么的确需要进行n-m+1 轮迭代，而且每一轮都需要比较m对字符（m-1 次成功，1 次失败）。考虑到m≪n，故蛮力串匹配算法的时间复杂度为$O(n \times m)$。

§ 9.4 Knuth-Morris-Pratt 算法

9.4.1 蛮力算法的改进

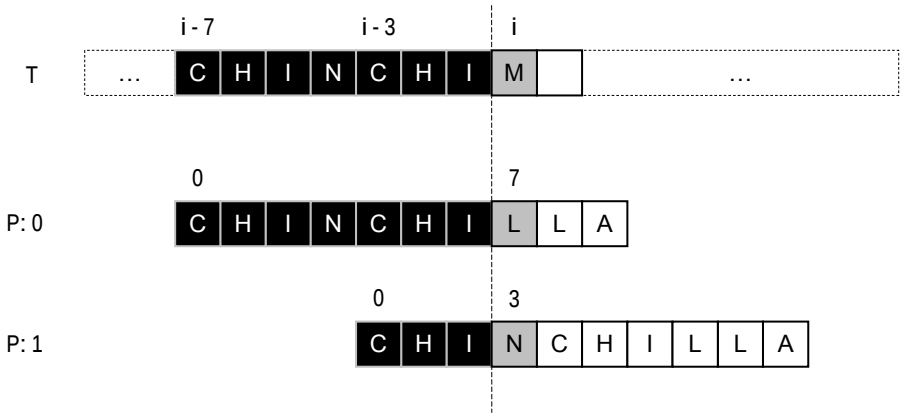
上一节的分析表明，在最坏情况下蛮力算法的运行时间为主串、模式串长度的乘积，因此只适用于小规模的串匹配应用。对最坏情况稍加观察即可发现，之所以需要大量的时间，是因为存在大量的局部匹配（每一轮的 m 次比较中，只有最后一次是失败的），而且一旦失配，主串、模式串的字符指针都要回退，并从头开始下一轮尝试。实际上，绝大部分的这类字符比较操作都是不必要的，因为关于主串中此前曾经比较成功过的字符，我们已经掌握了它们的所有信息。只要充分利用这些信息，就可以大大提高匹配算法的效率。



图九.3 利用以往的成功比较所提供的信息，可以避免主串字符指针的回退

如 图九.3 所示，用T[i]和P[j]分别表示当前正在接受比较的一对字符。当本轮比较进行到最后一对字符并发现失配后，蛮力算法将会让这两个字符指针回退（即令i = i-j+1 和j = 0），然后从这一位置继续比较。事实上，指针i完全不必回退——因为通过前一轮比较我们已经清楚地知道，主串的子串T[i-j..i-1]完全是由字符'0'组成的。因此，在回退之后紧接下来的一轮迭代中，前j-1 次比较将注定会成功。既然如此，完全可以让指针i保持不动并且令j = j-1，然后继续比较。请注意，如此将可以省去j-1 次比较！

上述“i 保持不动并且令 j = j-1”的含义，可以理解为“将 P 相对于 T 向右移动一个单元，然后从刚才失配的位置继续比较”。实际上，利用以往的成功比较所提供的信息，不仅可以避免主串字符指针的回退，而且有可能使模式串尽可能大跨度地右移。

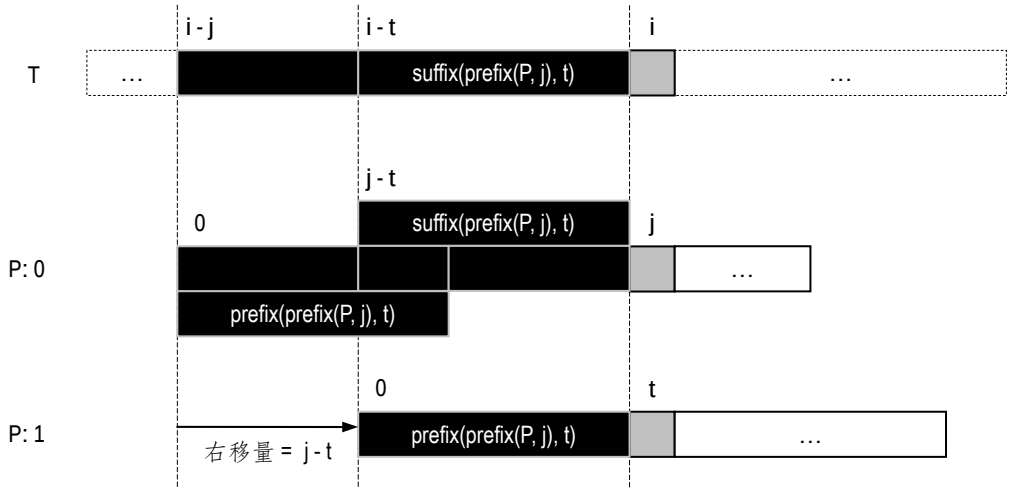


图九.4 利用以往的成功比较所提供的信息，有可能使模式串大跨度地右移

再来考察如 图九.4 所示的例子。当本轮比较中发现T[i] ≠ P[7]失配后，应该将模式串P右移多少个单元呢？有必要逐个单元地右移吗？

稍加观察即可发现，在这一情况下，移动一个、两个或三个单元都是徒劳的。事实上，根据以往的比较结果，必然有 T[i-7..i] = P[0..7] = "CHINCHI"。如果在此局部能够实现匹配，则至少在 T[i] 左侧的那些字符应该是匹配的——比如，当 P[0]与 T[i-3]对齐时，就属于这样的情况。如果再注意到 i-3 是能够如此匹配的最靠左位置，就可以放心地将模式串右移 7-3 = 4 个单元，使 i 保持不变，令 j = 3，然后继续比较。

9.4.2 next[]表的定义及含义



图九.5 利用以往的成功比较所提供的信息，在安全的前提下尽可能大跨度地右移模式串

一般地，如 图九.5 所示，假设前一轮迭代终止于T[i] ≠ P[j]。根据上面的分析，指针i可以不用回退，并且接下来的一轮迭代将从T[i]与P[t]的比较开始。现在的问题是，t应该等于多少呢？

如 图九.5 所示，前一轮迭代中经过比较匹配的范围为：

P[0..j-1] = T[i-j ..i-1]

即

prefix(P, j) = substr(T , i-j, j)

如果模式串 P 经右移之后能够与 T 的某一（包含 T[i]在内的）子串完全匹配，一个必要条件就是：

prefix(prefix(P, j), t) = T[i-t..i-1] = suf fix(prefix(P, j), t)

也就是说，在串 P 中，长度为 t 的真前缀与长度为 t 的真后缀完全匹配。更准确地，值得试探的 t 必然来自集合：

$N(P, j) = \{ t \mid prefix(prefix(P, j), t) = suffix(prefix(P, j), t), 0 \le t < j \}$

请注意，集合 N(P, j)只取决于模式串 P 本身以及前一轮迭代失配的位置 j，而与主串 T 无关。

从 图九.5 也可以看出，如果下一轮迭代从T[i]与P[t]的比较开始，其效果相当于将模式串P右移了j-t个单元。因此，为了不至于遗漏可能的匹配，需要在集合N(P, j) 中挑选最大的t——也就是说，当有多个值得试探的右移方案时，我们总是保守地选择其中移动距离最短者。因此，只要令

next[j] = max(N(P, j))

则一旦发现 P[j]与 T[i]失配，就可以转而用 P[next[j]]与 T[i]继续比较。

9.4.3 KMP 算法描述

上述处理方法可以描述为 算法九.1，即著名的KMP算法[1]。该算法的具体实现参见 代码九.2 中的PM()方法。

```
算法：串匹配算法KMP(P, T)
输入：模式串P和主串T
输出：若P与T的某一子串匹配，则输出第一个匹配位置i；否则返回值i > |T|-|P|
{
  构造next[]表；
  j = i = 0;  //将P和T的首字符对齐
  while (P仍然在T的范围内) {  //设P和T的字符指针分别为j和i
      if (0>j || P[j]==T[i]) {  //若匹配，或P已移出最左侧
        i++;  j++;  //则转到下一对字符
      } else  //否则
        j = next[j];  //P右移，T不用回退
  }
  return(i-j);  //返回最终P相对T的位置
}
```

算法九.1 KMP 算法

[1] 根据三位发明者名字 Knuth、Morris 和 Pratt 的首字母命名。

9.4.4 next[]表的特殊情况

既然空串是任何非空串的真子串（前缀、后缀），故只要 j > 0，就必有 0 ∈ N(P, j)。在这种情况下，N(P, j)必然非空，故“在其中取最大值”这一操作才有意义。然而反过来，倘若 j = 0，则前缀 prefix(P, j)本身就是空串，它没有任何真子串，于是集合 N(P, j) = ∅。在这种情况下，又应该如何定义 next[0]呢？

按照串匹配算法的构思，倘若某一轮迭代在比较第一对字符时就失配，我们就应该将模式串直接右移一个字符，然后从模式串的首字符起继续下一轮比较。这一处理方法可以等效为令 next[0] = -1。

我们也可以通过一个假象的处理方法来理解关于next[0]的定义：如 表九.3 所示，在P[0]的左侧“附加”一个通配符P[-1]，这个字符与任何字符都是匹配的。

表九.3 next[] 表实例：假想地附加一个通配符P[-1]

-1	0		1	2	3	4	5	6	7	8	9
*	C		H	I	N	C	H	I	L	L	A
-1		0	0	0	0	1	2	3	0	0	

9.4.5 next[]表的构造

需要特别指出的是，如上定义的 next[]表，只取决于模式串 P 以及失配的位置 j，而主串无关。因此，next[]表的构造可以作为预处理，只需进行一次即可。

幸运的是，next[]表的构造算法与 KMP 算法本身几乎一样。如果注意到集合 N(P, j)中的每一元素 t 都对应于串 prefix(P, j)自己与自己的一个匹配，就不会对此感到奇怪了。

next[]表构造算法的具体实现，参见 代码九.2 中的BuildNext()方法。

9.4.6 next[]表的改进

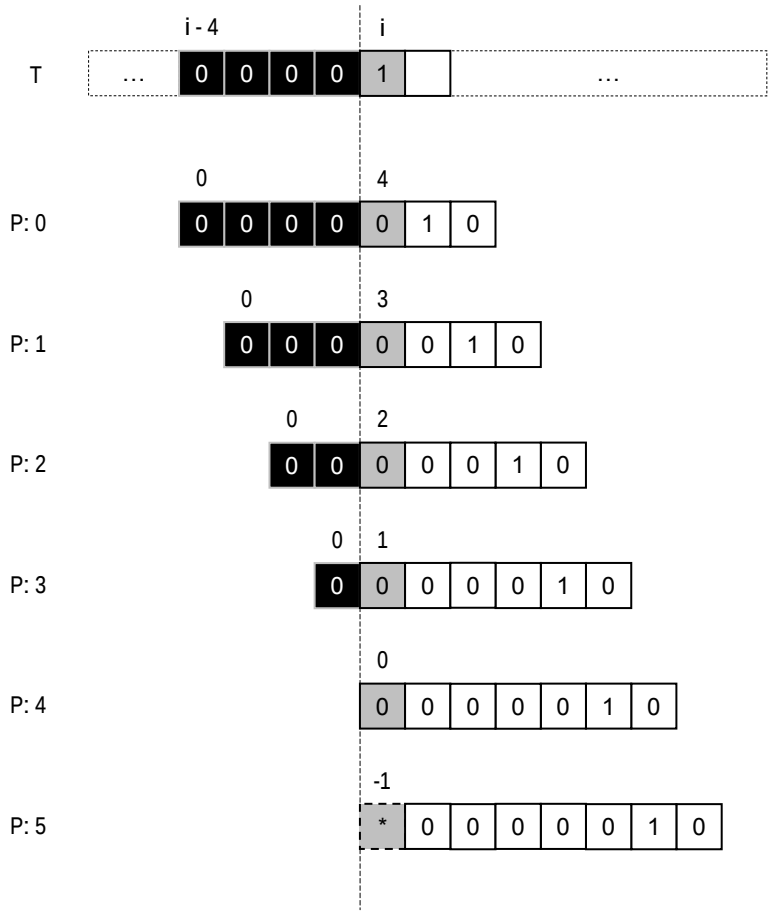
利用以上的 next[]表，KMP 算法可以避免大量不必要的比较操作。然而在某些情况下，依然存在进一步提高效率的可能。

试考察模式串P = "0000010"，按照第 9.4.2 节的定义，其对应的next[]如 表九.4 所示。

表九.4 next[] 表仍有待优化的实例

-1	0		1	2	3	4	5	6
*	0		0	0	0	0	1	0
-1		0	1	2	3	4	0	

如 图九.6 所示，假设前一轮迭代终止于T[i] = '1' ≠ '0' = P[4]。



图九.6 根据前一节所定义的next[]表，仍有可能进行多次不必要的比较操作

于是，接下来KMP算法将按照这一next[]表，依次用P[3]、P[2]、P[1]和P[0]与T[i]做比较。从 图九.6 可以看出，这四次比较都是失败的。

实际上，如果说 P[4]与 T[i]的比较还算必要的话，那么后续的这四次比较都是多余的，因为它们注定会失败。为了看出这一点，我们只需注意到 P[4] = P[3] = P[2] = P[1] = P[0] = '0'。既然经过比较已经发现 T[i] ≠ P[4]，为什么还要徒劳地用与 P[4]相同的字符去“重蹈覆辙”呢？

从算法策略的层次来看，第 9.4.2 节所定义的next[]表可以帮助我们充分利用以往成功的比较所提供的信息，这正是KMP算法得以加速的根本原因。然而实际上，此前已经进行过的比较还不止这些，确切地说，还有那些失败的比较——它们同样可以向我们提供信息，但却被忽略了。依然以 图九.6 为例，以往失败的比较已经提供了一条重要的信息——T[i] ≠ P[4]——而我们却未能加以利用。之所以会进行后续四次不必要的比较，原因正在于此。

为了在next[]表中引入这类信息，需要将第 9.4.2 节所定义的集合N(P, j)修改为：

$$N(P, j) = \{ t \mid prefix(prefix(P, j), t) = suffix(prefix(P, j), t) \text{ 且 } P[j] \neq P[t], 0 \leq t < j \}$$

相应地，算法九.1 也需稍作修改。如 代码九.2 中的BuildNextImproved()方法所示，在P[0..j-1]中发现长度为t的匹配前缀和后缀之后，还需判断P[j]是否等于P[t]。只有当P[j] ≠ P[t]时，才能采用t作为next[j]；否则，需要转而用next[t]作为next[j]。

表九.5 改进后的next[]表

-1	0		1	2	3	4	5	6
*	0		0	0	0	0	1	0
-1			-1	-1	-1	-1	4	-1

表九.4 中的next[]，经改进之后如 表九.5 所示。读者可以参照 图九.6，对next[]表两个版本的效率做一对比。

9.4.7 KMP 算法的 Java 实现

```
/*
 * 串模式匹配：KMP算法
 * 若返回位置i > length(T) - length(P)，则说明失配
 * 否则，i为匹配位置
 */
import dsa.*;
import java.io.*;

public class PM_KMP {
////////////////////////////////////////////////////////////////////////
// T: 0   1   .   .   .   i   i+1 .   .   .   i+k .   .   n-1
//    --------------------|-------------------|------------
// P:                     j   j+1 .   .   .   j+k .   .
//                        |-------------------|
////////////////////////////////////////////////////////////////////////
  public static int PM(String T, String P) {//KMP算法
    int[]   next = BuildNextImproved(P);//构造next[]表

    int   i = 0;//主串指针
    int   j = 0;//模式串指针

    while(j<P.length() && i<T.length()) {//自左向右逐个比较字符
        ShowProgress(T, P, i-j, j);   ShowNextTable(next, i-j, P.length());
  System.out.println();
      if (0>j || T.charAt(i) == P.charAt(j)) {//若匹配，或P已移出最左侧（提问：这两个条件能否交换次序？）
        i++;  j++;//则转到下一对字符
      } else//否则
        j = next[j];//模式串右移（注意：主串不用回退）
    }//while

    return(i-j);
  }

  protected static int[] BuildNext(String P) {//建立模式串P的next[]表
    int[]   next = new int[P.length()];//next[]表
```

```
  int   j = 0;//“主”串指针

  int   t = next[0] = -1;//“模式”串指针

  while (j < P.length()-1)
    if (0 > t || P.charAt(j) == P.charAt(t)) {//匹配
      j++;  t++;
      next[j] = t;//此句可以改进...
    } else//失配
      t = next[t];

    for (j=0; j<P.length(); j++)  System.out.print("\t"+P.charAt(j));
System.out.print("\n");
    ShowNextTable(next, 0, P.length());

  return(next);
}

protected static int[] BuildNextImproved(String P) {//建立模式串P的next[]表（改进版本）
  int[]   next = new int[P.length()];//next[]表

  int   j = 0;//“主”串指针
  int   t = next[0] = -1;//“模式”串指针

  while (j < P.length()-1)
    if (0 > t || P.charAt(j) == P.charAt(t)) {//匹配
      j++;  t++;
      next[j] = (P.charAt(j) != P.charAt(t)) ? t : next[t];//注意此句与未改进之前的区别
    } else//失配
      t = next[t];

    for (j=0; j<P.length(); j++)  System.out.print("\t"+P.charAt(j));
System.out.print("\n");
    ShowNextTable(next, 0, P.length());

  return(next);
}


protected static void ShowNextTable(//显示next[]表，供演示分析
  int[] N,
  int   offset,
  int   length)
{
  int   i;
  for (i=0; i<offset; i++)  System.out.print("\t");
```

```
    for (i=0; i<length; i++)  System.out.print("\t"+N[i]);
  System.out.print("\n\n");
  }

  protected static void ShowProgress(//动态显示匹配进展

    String  T,//主串

    String  P,//模式串
    int     i,//模式串相对于主串的起始位置
    int     j)//模式串的当前字符
  {
    int     t;

    System.out.println("-----------------------------------------");
    for (t=0; t<T.length(); t++)  System.out.print("\t"+T.charAt(t));
  System.out.print("\n");

    if (0 <= i+j) {
      for (t = 0; t < i+j; t++) System.out.print("\t");
      System.out.print("\t|");
    }
    System.out.println();

    for (t = 0; t < i; t++) System.out.print("\t");
    for (t=0; t<P.length(); t++)  System.out.print("\t"+P.charAt(t));
  System.out.print("\n");
    System.out.println();
  }
}
```

代码九.2 KMP 串匹配算法的实现

9.4.8 性能分析

以上可以看出，相对于蛮力算法，KMP算法（即 代码九.2 中的PM()方法）可以避免很多不必要的比较操作。然而就时间复杂度的意义而言，这是实质性的改进吗？

乍看起来，最坏情况时的运行时间似乎并未得到改进。从算法流程的角度看，该算法也要进行O(n)轮迭代，而且每一轮迭代依然有可能需要进行O(m)次字符比较，如此看来，在最坏情况下 KMP 算法仍有可能需要执行 O(nm)次比较操作。然而正如我们马上就要看到的，更为精确的分析表明，即使在最坏情况下，KMP 算法也只需运行线性的时间。

请注意 PM()方法中的两个字符指针 i 和 j。我们令 k = 2i-j，然后来考察在 PM()方法的执行过程中 k 的变化。

观察结论九.3 while 循环每迭代一轮，k 都会严格地递增。

〖证明〗

只要分别检验 while 循环内部的 if-else 分支，即可证明上述结论：如果执行 if 分支，则 i 和 j 同时加一，于是 k = 2i-j 也将加一；若执行 else 分支，则 i 保持不变，同时由于 next[j] < j 总是成立，故 j 至少会减小一，于是 k = 2i-j 也将至少增加一。 ❑

算法一开始，i = j = 0，故 k = 2×0 - 0 = 0。算法结束时，i ≤ n 且 j ≥ 0，故有 k ≤ 2n。既然每次迭代后 k 都是严格递增，故 while 循环至多需要执行 2n-1 轮。不难看出，while()循环的内部只需要 $\mathcal{O}(1)$时间，故如果不计构造 next[]表所需的时间，KMP 算法只需运行线性时间。

实际上，构造 next[]表的算法（无论是 BuildNext()还是 BuildNextImproved()）流程与 PM()的主体部分完全一样，仿照上述分析方法也可以证明：next[]表可以在 $\mathcal{O}(m)$的时间内构造出来。

综上所述，可以得出以下结论：

定理九.1 KMP 算法的运行时间为 $\mathcal{O}(n+m)$，其中 n 和 m 分别文本串和模式串的长度。

§9.5 BM 算法

KMP 算法的思想可以总结为：不断将模式串与文本串比较，一旦局部失配，则利用此前比较所给出的信息，尽可能长距离地移动模式串。请注意，在比较模式串与文本串时，这里的扫描方向是“自左向右”。实际上，很多模式匹配算法采用了其它的扫描方向，比如“自右向左”或者“从中间向两边”等。

BM算法[1]采用的就是“自右向左”的扫描次序。该算法的构思是：不断自右向左地比较模式串P与主串T，一旦发现失配，则利用此前的扫描所提供的信息，将P右移一定距离，然后重新自右向左扫描比较。该算法有两种启发式策略——借助坏字符（Bad Character）和好后缀（Good Suffix）确定移动的距离——也可将二者结合起来，同时采用。

[1] 得名于其发明者。R. S. Boyer & J. S. Moore. A fast string searching algorithm, Communications of the ACM. 1977, 20:762-772.

9.5.1 坏字符策略

■ 坏字符

图九.7 坏字符策略：通过右移模式串P，使T[i+j]得到匹配

如 图九.7 所示，在自右向左比较模式串P[0..m-1]与主串的子串T[i..i+m-1]的过程中，假设在P[j]处首次发现失配：T[i+j] = 'b' ≠ 'a' = P[j]。此时，我们应该用P中的哪个字符对准T[i+j]并重新自右向左比较呢？

我们注意到，若 P 能够与 T 的某一（包括 T[i+j]在内的）子串匹配，则必然也应在 T[i+j] = 'b' 处匹配；反之，若与 T[i+j]对准的字符不是'b'，则不可能匹配。因此，只需将 P 中的每一字符'b'对准 T[i+j]，然后重新自右向左比较。

为了避免 P 的左移，我们可以选用 P 中最靠右的字符'b'（如果存在的话），将其与 T[i+j]对齐，然后重新做一遍自右向左的扫描比较。具体来说，若 P 中最靠右的字符'b'为 P[k] = 'b'，则 P 的右移量为 j - k。

由上可见，右移距离只取决于当前失配位置 j 以及 k 值。那么，k 值又是如何确定的呢？是否需要每次重新计算呢？

■ BC[] 表及其构造算法

我们注意到，对于任一给定的模式串P，k值只取决于字符T[i+j] = 'b'。因此，我们可以将其视作从字符表到整数的一个函数BC()[1]：

$$BC(c) = \begin{cases} k & (\text{若pattern}[k] = c, \text{且对所有的} i > k \text{都有pattern}[i] \neq c) \\ -1 & (\text{若pattern}[] \text{中不含字符} c) \end{cases}$$

需要强调的是，对于给定的字符表，函数BC[]的取值只取决于具体的模式串P，而与主串T无关。因此与KMP算法中的next[]表一样，也可以通过预处理计算出BC[]表，并将其组织为一张查询表（Look-up table），供匹配算法使用。BC[]表的构造过程可以描述为 算法九.2：

```
算法：BC[]表构造算法BuildBC(P)
输入：模式串P
```

[1] BC 是 Bad Character 的缩写。——作者

```
输出：P对应的BC[]表
{
  对于字母表Σ中的每一字符c，令BC[c] = -1;  //首先假设该字符没有在P中出现
  自左向右扫描P中的各个字符
    如果当前字符为P[j] = c，则令BC[c] = j;  //更新各字符的BC[]值
  return(BC[]);
}
```

算法九.2 BC[] 表构造算法

算法九.2 的具体实现，参见 代码九.3 中的BuildBC()方法。

■ 正确性

请注意，这里是按照从左到右（即下标递增）的次序对模式串进行扫描，故只要某个字符 c 在 P 中出现过，则在算法结束时 BC[c]中记录的就是最靠右的 c 的位置。由此可得如下结论：

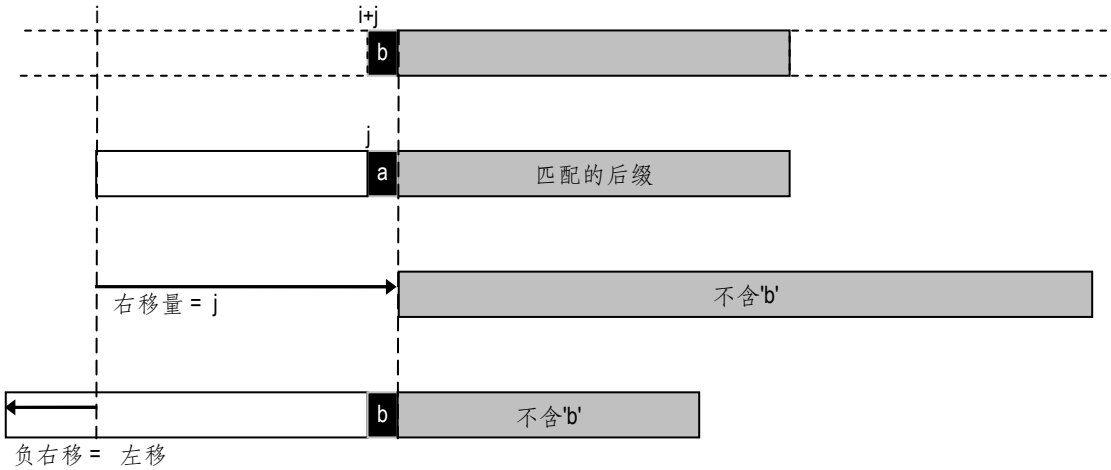
观察结论九.4 算法 BuildBC(P)能够正确地构造出模式串 P 对应的 BC[]表。

■ 预处理时间

算法九.2 所需时间可以划分为两部分，分别消耗于其中的两个循环。前一个循环是对字符表Σ中的每个字符分别做初始化，故需要$O(|\Sigma|)$时间。后一循环对模式串进行扫描，由于每个字符的处理只需$O(1)$时间，故共需$O(m)$时间。由此可得如下结论：

定理九.2 BC[] 表可以在 $O(|\Sigma|+m)$时间内构造出来，其中|Σ|为字符表的规模，m 为模式串的长度。

■ 特殊情况



图九.8 坏字符策略：T[i+j]在P中未出现，或者右移量为负数

还有些特殊情况需要处理。如 图九.8 所示，有时P串中根本就不含字符'b'，此时可以直接将该串整体移过，用P[0]对准T[i+j+1]，然后自右向左继续比较。

另外，即使 P 串中含有字符'b'，却也有可能出现的位置太靠右，使得 k = BC['b'] ≥ j。在这一情况下，j-k 将不再是正数，若以此距离进行右移，实际效果将是左移——显然，这是不必要的。因此，为处理这一情况，只需简单地将 P 串右移一个字符，然后重新自右向左扫描比较。

9.5.2 好后缀策略

■ 好后缀

BM 算法的思想，是尽可能地利用此前已进行过的比较所提供的信息，以加速模式串的移动。上述坏字符策略，就很好地体现了这一构思：既然已经发现 P[j]与 T[i+j]不匹配，就应该从 P 中找出一个与 T[i+j]匹配的字符，将二者对齐之后，重新自右向左开始比较。

然而，仔细分析后我们可以发现，坏字符策略只利用了此前（最后一次）失败的比较所提供的信息。实际上，在失败之前往往还会有一系列成功的比较，它们也能提供大量的信息，对此我们能否加以利用呢？

在 图九.9 中，假设前一轮自右向左的比较终止于T[i+j] ≠ P[j]。如果分别记

$$\begin{cases} W = \text{substr}(T, i+j+1, m-j-1) \\ U = \text{suffix}(P, j+1, m-j-1) \end{cases}$$

于是只要 j ≤ m-2，则后缀 U 必非空，而且满足：

$$U = W$$

正如我们马上就要看到的，利用后缀 U 所提供的信息可以加速模式串的右移，因此我们将这样的后缀称作“好后缀”。

■ 好后缀策略

如 图九.9 所示，假设经过右移使T[i+j]与P[k]相互对齐之后，模式串P能够与主串T的某一（包含T[i+m-1]在内的）子串匹配，即

$$P = \text{substr}(T, i+j-k, m)$$

既然如此，如果记

$$U' = \text{substr}(P, k+1, m-j-1)$$

则必然有：

$$U' = W = U$$

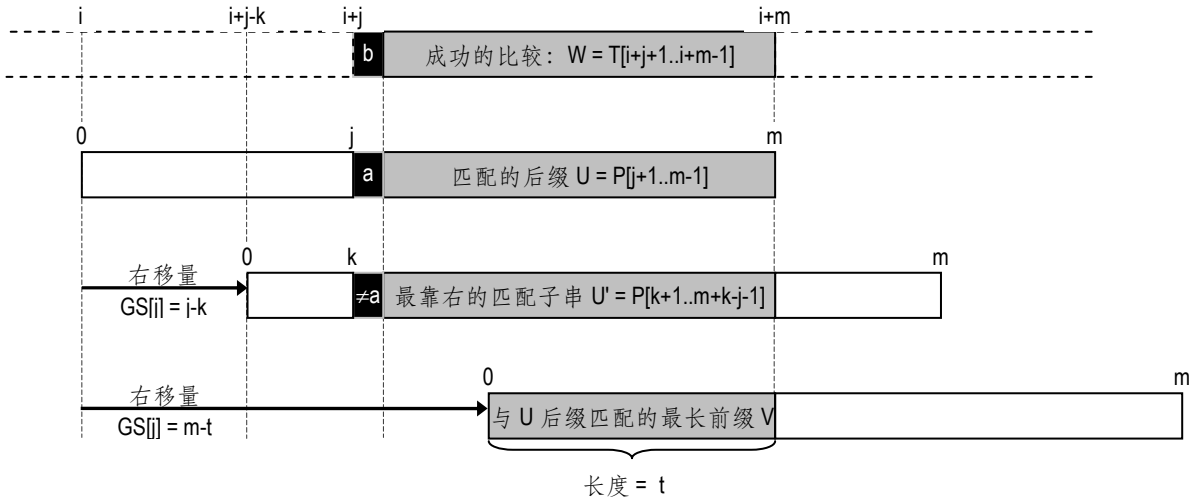
也就是说，在模式串 P 中，存在某一长度为 m-j-1 的子串与 P 自己的后缀匹配——只有在满足这一条件的位置，才有可能使 P 得到匹配。当然，这只是模式串 P 与主串 T 匹配的一个必要条件，在将这两段匹配的子串对齐之后，还需要重新自右向左进行比较。

此外，还要加上另一个必要条件：P[k] ≠ P[j]。否则，与第 9.4.6 节关于KMP算法中next[]表的改进同理，这一对齐位置注定不会出现P的整体匹配。

若模式串 P 中存在多个满足上述必要条件的子串，我们不妨取其中最靠右者，以避免此后 P 串的左移。

如果满足上述必要条件的子串 U'起始于 P[k+1]，则对应的右移量就是 j-k。请注意，这个右移量只取决于 P 串以及失配的位置 j，因此与 KMP 算法中的 next[]表一样，也可以通过预处理在事先计算出来，并将结果保存为一张查找表 GS[0..m-1]。

■ 特殊情况



图九.9 好后缀策略

有一种特殊情况需要处理：如 图九.9 所示，有的时候，模式串P中没有任何子串与好后缀U完全匹配。此时，我们可以从模式串P的所有前缀中，找出一个尽可能长的、与U的（真）后缀匹配的前缀V。如果V的长度为t，则取GS[j] = m-t。

查找表GS[]的具体构造算法，参见 代码九.3 中的BuildGS()方法。这里，ComputeSuffixSize()方法的作用，是计算P的各前缀与P的各后缀的最大匹配长度，即对于P的每一前缀X = prefix(P, j+1)，计算出

$$SS[j] = \max\{s \mid suffix(X, s) = suffix(P, s)\}$$

9.5.3 BM 算法

综上所述，BM算法可以描述为 算法九.3：

```
算法：串匹配算法BoyerMoore(P, T)
输入：模式串P和主串T
输出：若P与T的某一子串匹配，则输出第一个匹配位置i；否则返回值i > |T|-|P|
{
  构造BC[]表;
  构造GS[]表;

  i = 0;  //将P和T的首字符对齐
  while (P仍然在T的范围内) {  //设P和T的字符指针分别为j和i+j
    j = |P|-1;  //从最右侧的字符开始
    while (P[j] == T[i+j])  //自右向左逐一比较各对字符，直至发现失配或者
      if (0 > --j)  break;  //所有字符对经过比较都获成功
```

```
    if (所有比较都成功)
      return(i);  //返回匹配位置
    else
      i += MAX(GS[j], j - BC[T[i+j]];  //在位移量BC和GS中选择大者，右移模式串
  }

  //执行至此，说明P已经移出T的范围，即T中不存在与P匹配的子串
  return(i);  //注意，此时的返回值i满足i > |T|-|P|，可依此判断是否存在匹配
}
```

算法九.3 BM 算法

请注意，这里同时采用了坏字符、好后缀两种启发策略，并在它们给出的右移量中取大者。该算法的具体实现，参见 代码九.3 中的PM()方法。

9.5.4 BM 算法的 Java 实现

```
/*
 * 串模式匹配：Boyer-Moore算法

 * 若返回位置i > length(T) - length(P)，则说明失配

 * 否则，i为匹配位置
 */
import dsa.*;
import java.io.*;

public class PM_BM {
  final static int CARD_CHAR_SET = 256;//字符集规模
////////////////////////////////////////////////////////////////////////
// T: 0   1   .   .   .   i   i+1 .   .   .   i+j .   .   n-1
//    --------------------|-------------------|------------
// P:                     0   1   .   .   .   j   .   .
//                        |-------------------|
////////////////////////////////////////////////////////////////////////
  public static int PM(String T, String P) {
  //预处理
    int[]   BC = BuildBC(P);
    int[]   GS = BuildGS(P);

  //查找匹配
    int i = 0;//模式串相对于主串的起始位置（初始时与主串左对齐）
    while (T.length()-P.length() >= i) {//在到达最右端前，不断右移模式串
      int j = P.length() - 1;//从模式串最末尾的字符开始
      while (P.charAt(j) == T.charAt(i+j))//自右向左比较
        if (0 > --j)  break;
```

```
            ShowProgress(T, P, i, j); System.out.print("\n");
      if (0 > j)//若极大匹配后缀 == 整个模式串（说明已经完全匹配）
        break;//返回匹配位置
      else//否则
        i += MAX(GS[j], j - BC[T.charAt(i+j)]);//在位移量BC和GS之间选择大者，相应地移动模式串
    }
    return(i);
  }

  /*
   *构造Bad Charactor Shift表BC[]
   *
   *  0                       BC['X']                                  m-1
   *  |                       |                                        |
   *  ........................X****************************************
   *                          |<------------ 不含字符'X' ------------>|
   *
   * 复杂度 = O(m + CARD_CHAR_SET)
   ************************************************************************/
  protected static int[] BuildBC(String P) {
  //初始化
    int[] BC = new int[CARD_CHAR_SET];//BC[]表

    int j;

    for (j = 0; j < CARD_CHAR_SET; j++)
      BC[j] = -1;//首先假设该字符没有在P中出现

  //自左向右迭代：更新各字符的BC[]值
    for (j = 0; j < P.length(); j++)
      BC[P.charAt(j)] = j;//P[j]曾出现在位置j——鉴于这里的扫描次序是从左到右（即下标递增），故只要某个字符ch在P中出现过，BC[ch]就会记录下其中的最靠右的出现位置
          System.out.println("-- BC[] Table ---------------");
          for (j = 0; j < CARD_CHAR_SET; j++)
            if (0 <= BC[j]) System.out.print("\t"+(char)j);
          System.out.println();
          for (j = 0; j < CARD_CHAR_SET; j++)
            if (0 <= BC[j]) System.out.print("\t"+BC[j]);
          System.out.println("\n");
    return(BC);
  }

  /*
   * 计算P的各前缀与P的各后缀的最大匹配长度
   * 对于P的每一前缀P[0..j]，SS[j] = max{s | P[j-s+1..j] = P[m-s..m-1]}
   *
   *  0                       m-SS[j]                 m-1
   *  |                       |                       |
```

```
 *  ......................*************************
 *                       |                       |
 *                       <-------- SS[j] -------->
 *                       |                       |
 *             ..........*************************..................
 *             |         |                       |                  |
 *             0         j-SS[j]+1               j                  m-1
 *
 * 复杂度 = O(m)
 ***************************************************************************/
protected static int[] ComputeSuffixSize(String P) {
  int   m = P.length();
  int[] SS = new int[m];//Suffix Size Table
  int   s, t;//子串P[s+1, ..., t]与后缀P[m+s-t, ..., m-1]匹配
  int   j;//当前字符的位置

//  对最后一个字符而言，与之匹配的最长后缀就是整个P串，故...
  SS[m-1] = m;

//  从倒数第二个字符起，自右向左扫描P，依次计算出SS[]其余各项
  s = m-1;  t = m-2;
  for (j = m-2; j >= 0; j--) {
    if ((j > s) && (j-s > SS[(m-1-t)+j]))
      SS[j] = SS[(m-1-t)+j];
    else {
      t = j;//与后缀匹配之子串的终点，就是当前字符

      s = MIN(s, j);//与后缀匹配之子串的起点

      while ((0 <= s) && (P.charAt(s) == P.charAt((m-1-t)+s)))
        s--;//似乎是二重循环，难道复杂度是平方量级?
      SS[j] = t-s;//与后缀匹配之最长子串的长度
    }
  }
      System.out.println("-- SS[] Table -------");
      for (j=0; j<m; j++) System.out.print("\t"+P.charAt(j));
System.out.println();
      for (j=0; j<m; j++) System.out.print("\t"+SS[j]);
System.out.println("\n");
  return(SS);
}

/*
 * 构造Good Suffix Shift表GS[]
 * 复杂度 = O(m)
 ***************************************************************************/
protected static int[] BuildGS(String P) {
  int   m = P.length();
  int[] SS = ComputeSuffixSize(P);//计算各字符对应的最长匹配后缀长度
```

```
  int[] GS = new int[m];//Good Suffix Index
  int   j;
  for (j = 0; j < m; j++) GS[j] = m;

/*
 *              i < m-j-1（失配位置）
 *              |
 *  0           |         m-j-1                m-1
 *  |           |         |                    |
 *  ............A#########**********************
 *              |         |                    |
 *              |         <---- Suffix Size ----><------ GS Shift ------>
 *              |         <---- SS[j] = j+1 ----><-------- m-j-1 ------->
 *              |         |                    |                       |
 *                        **********************.........................
 *                        |                    |                       |
 *                        0                    j                      m-1
 *                        <--<--<--<--<--<--< 自右向扫描 <--<--<--<--<--<
 *
 ************************************************************************/
  int i = 0;
  for (j = m-1; j >= -1; j--)//提问：反过来（自左向右）扫描可以吗？为什么？
    if (-1 == j || j+1 == SS[j])//若定义SS[-1] = 0，则可统一为：if (j+1 == SS[j])
      for (; i < m-j-1; i++)//似乎是二重循环，难道复杂度是平方量级？
        if (GS[i] == m)
          GS[i] = m-j-1;

/*

 *               m-SS[j]-1（失配位置）

 *               |
 *  0            |m-SS[j]        m-1
 *  |            ||            |
 *  ..................A**********************
 *               ||            |
 *               |<--- Suffix Size ----><-- GS Shift ->
 *               |<------- SS[j] ------><--- m-j-1 --->
 *               ||            |         |
 *            .....B**********************.............
 *            |   |            |         |
 *            0   j-SS[j]+1      j         m-1
 *            >-->-->-->-->--> 自左向右扫描 --->-->-->-->
 *
 ************************************************************************/
  for (j = 0; j < m-1; j++)//提问：反过来（自右向左）扫描可以吗？为什么？
    GS[m-SS[j]-1] = m-j-1;

        System.out.println("-- GS[] Table ---------------");
        for (j=0; j<m; j++) System.out.print("\t"+P.charAt(j));
System.out.println();
```

```
          for (j=0; j<m; j++) System.out.print("\t"+GS[j]);
  System.out.println("\n");
    return(GS);
  }

  protected static void ShowProgress(//动态显示匹配进展
    String  T,//主串
    String  P,//模式串
    int     i,//模式串相对于主串的起始位置
    int     j)//模式串的当前字符
  {
    int     t;

    System.out.println("-----------------------------------------");
    for (t=0; t<T.length(); t++)  System.out.print("\t"+T.charAt(t));
  System.out.print("\n");

    if (0 <= i+j) {
      for (t = 0; t < i+j; t++) System.out.print("\t");
      System.out.print("\t|");
    }
    System.out.println();

    for (t = 0; t < i; t++) System.out.print("\t");
    for (t=0; t<P.length(); t++)  System.out.print("\t"+P.charAt(t));
  System.out.print("\n");
    System.out.println();
  }

  protected static int MAX(int a, int b)

  { return (a>b) ? a : b; }


  protected static int MIN(int a, int b)
  { return (a<b) ? a : b; }
}
```

代码九.3 BM　串匹配算法的实现

9.5.5 性能

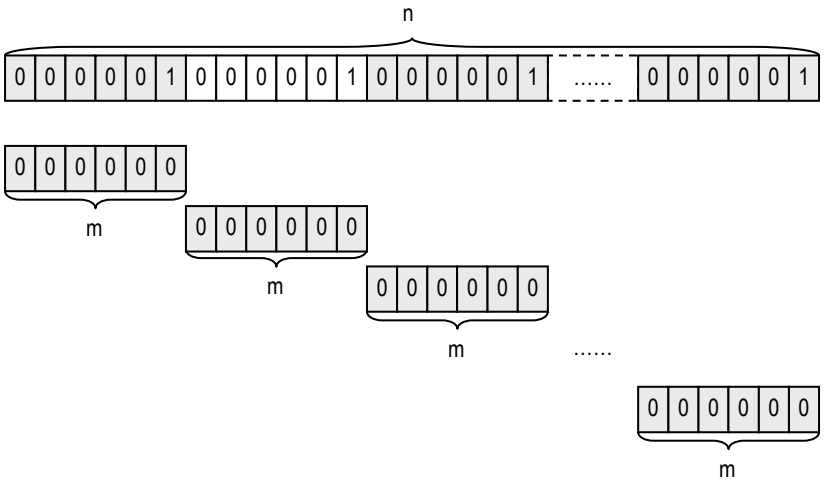
- 运行时间

  若文本串 P 和模式串 T 的长度分别为 n 和 m，BM 算法需要进行多少次字符的比较操作呢？

Knuth等人[1]在 1977 年证明，最坏情况下不超过 7n次。1980 年，Guibas和Odlyzko[2]将这个上界改进至 4n。1990 年，Cole[3]进一步改进为 3n，而且证明这个上界是紧的。鉴于这些证明都过于复杂，在此省略。

定理九.3 BM 算法的运行时间为 $O(n+m)$，其中 n 和 m 分别文本串和模式串的长度。

■ 最好情况



图九.10 BM 算法的最好情况

在通常情况下，该算法的实际运行时间甚至往往会低于线性复杂度。而在最好的情况下，每经过常数次比较，BM算法就可以将模式串右移m个字符（即整体右移），因此只需经过$O(n/m)$次比较，算法即可终止。图九.10 中给出的就是这样的一个例子。

[1] D. E. Knuth, J. Morris, V. Pratt. Fast Pattern Matching in Strings. SIAM Journal on Computing, 6(1):323-350, 1977.

[2] L. J. Guibas and A. M. Odlyzko. A New Proof of the Linearity of the Boyer-Moore String Search Algorithm. SIAM Journal on Computing, 9(4): 672-682, 1980.

[3] COLE, R., 1994, Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm, SIAM Journal on Computing 23(5):1075-1091.（该项工作最早于 1990 年以技术报告形式给出）

第十章

弥诺陶洛斯（Minotaur）是希腊神话中半人半牛的怪物，它藏身于一个精心设计的迷宫之中。这个迷宫的结构极其复杂，一般人一旦进入其中，都休想走出来。不过，在公主阿里阿德涅（Ariadne）的帮助下，古希腊英雄特修斯（Theseus）还是想出了一个走出迷宫的方法。特修斯带上一团线去闯迷宫，在入口处，他将绳线的一头绑在门上。然后，他不断查找迷宫的各个角落，而绳线的另一头则始终抓在他的手里，跟随他穿梭于蜿蜒曲折的迷宫之中。借助如此简单的工具，特修斯不仅终于找到了怪物并将其杀死，而且还带着公主轻松地走出了迷宫。实际上，这个方法就是本章将要介绍的图算法之一。

特修斯之所以能够成功，关键在于他借助绳线来掌握迷宫内各个通道之间的联接关系，而在很多的问题中，掌握类似的信息都是至关重要的。通常，这类信息所反映的都是一组对象之间的二元关系，比如城市交通图中联接于不同点之间的街道，抑或 Internet 中联接于两个 IP 之间的路由等。在某种程度上，我们前面所讨论过的树结构也可以携带和表示这种二元关系，只不过树结构中的这类关系仅限于父、子节点之间。然而在一般情况下，我们所需要的二元关系往往是定义在任意一对对象之间。实际上，这样的二元关系恰恰正是图论（Graph theory ）所讨论的问题。因此，本章将介绍基于图的数据结构，并讨论相应的算法，以有效地针对这种关系进行计算。

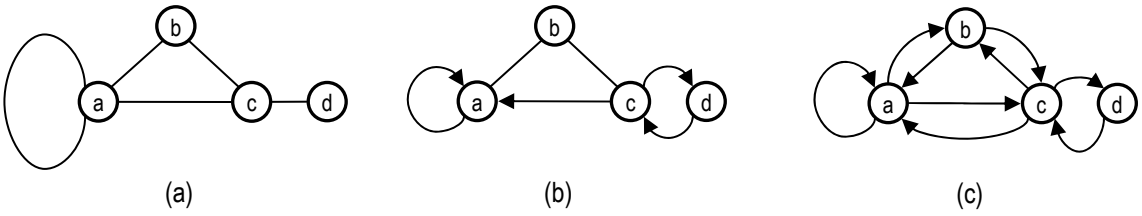
§10.1 概述

在众多的实际应用中，图结构都是描述与解决应用问题的一个基本而强有力的工具，比如交通图、航线图、电路图、冲突图或可见性图等。这里所谓的图（Graph）可以表示为 G = (V, E)，其中集合 V 中的对象称作顶点（Vertex），而集合 E 中的每一元素都对应于 V 中某一对顶点——说明这两个顶点之间存在某种关系——称作边（Edge）。因此，本章所要讨论的“图”，并非通常所指的图形、报表或者数学上的函数图像之类。当然，这里还约定 V 和 E 都是有限集，通常记 n = |V|、m = |E|。

关于图的术语，在不同文献中不尽相同，比如有的将顶点称作节点（Node），或将边称作弧（Arc）。本书将统一为顶点和边。显示图结构的一种简单而有效的方法，就是将顶点表示为小圆圈或小方块，用联接于其间的线段或曲线来表示对应的边。

10.1.1 无向图、混合图及有向图

图中的边可以是没有方向的，也可以是有方向的。如果边 e = (u, v)所对应的顶点 u 和 v 不是对等的，或者存在某种次序，就称 e 为有向边（Directed edge）。如果 u 和 v 的次序无所谓，e 就是一条无向边（Undirected edge）。注意：无向边(u, v)也可以记作(v, u)，而有向边(u, v)和(v, u)则是不同的两条边。



图十.1 (a) 无向图、(b)混合图和(c)有向图

若E中所有的边都没有方向，则称G为无向图（Undirected graph）；若E中同时包含无向边和有向边，则称G为混合图（Mixed grap h）；若E中只含有向边，则称G为有向图（Directed graph，简称Digraph）。相对而言，有向图的通用性更强，因为无向图和混合图都可以转化为有向图——如 图十.1 所示，只需将其中的每一条无向边(u, v)替换为一对有向边(u, v)和(v, u)。因此，本章将主要针对有向图，介绍图结构及其算法的具体实现。

为了加深对图结构的认识，试考虑如下几个图的实例：

1. 将电影界的所有演员构成顶点集 V，其中两位演员 u 和 v 如果曾经共同出演过至少一部影片，就在他们之间定义一条边 e = (u, v) ∈ E。按照演艺圈的惯例，合作者之间相互对等，故这样的每一条边都是无向边，对应的图 G 则为无向图。
2. 对于任一 Java 程序，若将其中的类构成顶点集 V，且如果类 u 是类 v 的子类，则定义一条从 v 指向 u 的有向边。当然，如此定义出来的图 G 属于有向图。

10.1.2 度

若边e = (u, v)，则顶点u和v也称作e的端点（End vertices或Endpoints）。如果e是从u指向v的有向边，则u称作起点（Origin）或尾端点（Tail），v称作终点（Destination）或头端点（Head）。我们也称u和v是相邻的（Adjacent），称e与v、u是相关联的（Incident）。顶点v的关联边的总数，称为v的度数（Degree），记作deg(v)。以 图十.1(a)为例，有deg(a) = deg(c) = 3

在有向图中，以u为起点的有向边称作u的出边（Outgoing edge），以v为终点的边则称作v的入边（Incoming edge ）。v的出边总数称作v的出度（Out-degree），记作outdeg(v)；入边总数称作入度（In-degree），记作indeg(v)。以 图十.1(c)为例，有outdeg(a) = indeg(a) = outdeg(c) = indeg(c) = 3。

10.1.3 简单图

图中所含的边并不见得能构成一个集合（Set），准确地说它们构成了一个复集（Multiset）——其中允许出现重复的元素。比如，若在某对顶点之间有多条无向边，或者两条有向边的起点和终点完全一样，就属于这种情况。这类重复的边也称作平行边（Parallel e dges）或多重边（Multiple edges）。例如，要是用顶点表示城市，用边表示城市之间的飞机航线，则有可能在某一对城市之间存在多条航线；如果用顶点表示演员，则某两位演员合演过的影片很有可能不止一部。

无论是无向图还是有向图，还有另一种特殊情况：与某条边关联的是同一个顶点（如 图十.1 中的顶点a）——这样的边称作自环（Self-loop）。在某些特定的应用问题中，这类边的确是有意义的——比如在城市交通图中，沿着某条街道，有可能会不经过任何交叉路口而直接返回原处。

不过，这些特殊情况通常并不多见。不含上述特殊边的图，称作简单图（Simple graph）。对简单图而言，其中的边必然构成一个集合，而且每条边只能联接于不同的顶点之间。在本书中，除非特别说明，我们讨论的图一般都限于简单图。处理简单图的数据结构与算法，一般都可以推广至非简单的图——尽管具体过程多少有些繁琐。

10.1.4 图的复杂度

于其它算法一样，本章也将对有关图算法的时间、空间复杂度进行分析和比较，为此我们往往是以图中的顶点数与边数的总和作为输入的规模，这个总和也就是图结构本身的复杂度。

观察结论十.1 对于任何无向图 G = (V, E)，都有 $\sum_{v\in V}\mathbf{deg}(v) = 2|E|$。

〖证明〗

只需注意到，在累加各点度数的过程中，每条边都恰好被统计两次。 ❑

观察结论十.2 对于任何有向图 G = (V, E)，都有 $\sum_{v\in V}\mathbf{indeg}(v) = \sum_{v\in V}\mathbf{outdeg}(v) = |E|$。

〖证明〗

这是因为，在累加各点入（出）度数的过程中，每条边都恰好被统计一次。 ❑

那么，在由 n 个顶点组成的图中，至多有多少条边呢？

观察结论十.3 设简单图 G 包含 n 个顶点和 m 条边。

① 若 G 是无向图，则有 $m \le n(n-1)/2$；

② 若 G 是有向图，则有 $m \le n(n-1)$。

〖证明〗

对于简单的无向图G，由于其中各边对应的顶点对互不相同，而且没有自环，所以每个顶点的度数不会超过n-1，于是根据 观察结论十.1，有 $\sum_{v\in G}\mathbf{deg}(v) = 2m \le n(n-1)$。

对于简单的有向图G，同样由于G的简单性，每个顶点的出度不会超过n-1，于是根据观察结论十.2，有 $\sum_{v\in G}\mathbf{outdeg}(v) = m \le n(n-1)$。同理可得 $\sum_{v\in G}\mathbf{indeg}(v) \le n(n-1)$。 ❑

由 观察结论十.3 可知：

推论十.1 在由 n 个顶点构成的简单图中，边的数目为 $O(n^2)$。

10.1.5 子图、生成子图与限制子图

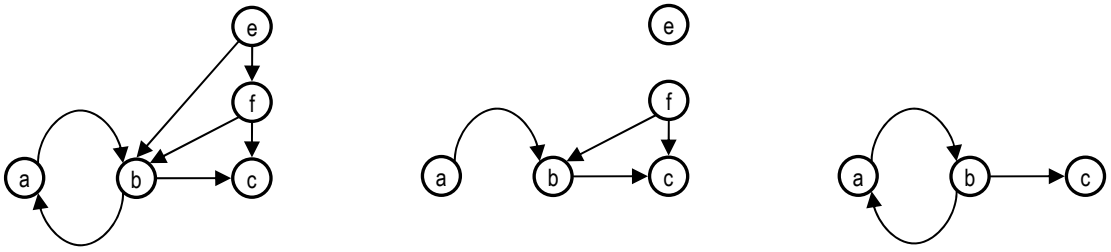
设 G = (V, E)和 G' = (V', E')。

定义十.1 如果 V' ⊆ V 且 E' ⊆ E，则称 G'是 G 的一个子图（Subgraph）。

以 图十.1(c)为例，若V' = {a, b, c}且E' = {(a, b), (a, c)}，则G' = (V', E')就构成了原图G的一个子图。

定义十.2 如果 V' = V 且 E' ⊆ E，则称 G'是 G 的一个生成子图（Spanning subgraph）。

还是以 图十.1(c)为例，若V' = {a, b, c, d}且E' ⊆ E，则G' = (V', E')就是原图G的一个生成子图。



(a) 有向连通图 G　　(b) G 的生成子图　　(c) G 限制于{a, b, c}的子图

图十.2　有向连通图的生成子图及其限制子图

定义十.3 若 U ⊆ V，则在删除 V\U 中的顶点及其关联边之后所得到的 G 的子图，称为 G 限制在 U 上的子图，记做 $G|_U = (U, E|_U)$。

10.1.6 通路、环路及可达分量

定义十.4 所谓图中的一条通路或路径（Path），就是由（不一定互异的）m+1 个顶点与 m 条边交替构成的一个序列$\rho = \{v_0, e_1, v_1, e_2, v_2, \ldots, e_m, v_m\}$，$m \geq 0$，而且 $e_i = (v_{i-1}, v_i)$，$1 \leq i \leq m$。

m 称作该通路的长度，记作 m = |ρ|。

定义十.5 长度 $m \geq 1$ 的路径，若第一个顶点与最后一个顶点相同，则称之为环路（Cycle）。

如果组成通路ρ的所有顶点各不相同，则称之为简单通路（Simple path）；如果在组成环路的所有顶点中，除$v_0 = v_m$外均各不相同，则称之为简单环路（Simple cycle）。如果组成通路ρ的所有边都是有向边，而且每一e_i都是从v_{i-1}指向v_i，$1 \leq i \leq m$，则称ρ为有向通路（Directed path）；类似地，也可以定义有向环路（Directed cycle）。例如在 图十.1(c)中，{a, (a, b), b, (b, c), c, (c, d), d}是一条简单的有向通路，而{a, (a, b), b, (b, c), c, (c, a), a}则是一条简单的有向环路。

在描述简单图中的通路或环路时，我们只需依次给出组成该通路或环路的各个顶点，而不必再给出具体的边。

关于简单路径，不难验证如下事实：

观察结论十.4 在图 G = (V, E)中，若记 n = |V|，则

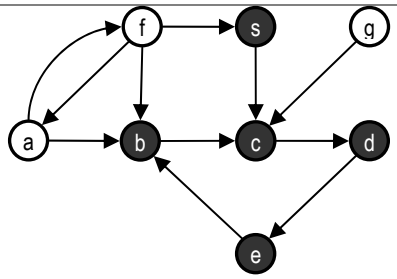
① 简单路径长度不超过 n-1；

② 长度为 k 的简单路径的总数不超过 n!/(n-k-1)!，$1 \le k \le n-1$；

③ G 中简单路径的数目是有限的。

在有向图 G 中，若从顶点 s 到 v 有一条通路，则称 v 是“从 s 可达的”。

定义十.6 对于指定的顶点 s，从 s 可达的所有顶点所组成的集合，称作 s 在 G 中对应的可达分量，记作 $V_r(G, s)$。



图十.3 图G中顶点s对应的可达分量（黑色顶点）

关于可达性，不难验证如下事实：

观察结论十.5 若 $v \in V_r(G, s)$，则有一条简单路径从顶点 s 通往 v。

10.1.7 连通性、等价类与连通分量

考察图 G = (V, E)。在顶点 u, v ∈ V 之间，如果既存在一条从 u 到 v 的通路ρ(u, v)，也存在一条从 v 到 u 的通路ρ(v, u)，则称 u 和 v 是连通的，记作 u ~ v。

实际上，若 G 是无向图，则“ρ(u, v)存在”当且仅当“ρ(v, u)存在”。而对于有向图 G 来说，“ρ(u, v)和ρ(v, u)同时存在”即意味着“存在一条同时经过 u 和 v 的环路”。故此，有向图中相互连通的顶点也被称为是互相“强连通的”（Strongly connected）。

定义十.7 有向图中，由一组相互强连通的顶点构成的极大集合，称作一个强连通分量。

关于连通关系，有以下结论：

观察结论十.6 若 u~v，则

① 必然存在一条经过 u 和 v 的环路，而且

② 对于该环路上的任一顶点 w，都有 w ~ u 和 w ~ v。

观察结论十.7 连通关系“~”满足如下性质：

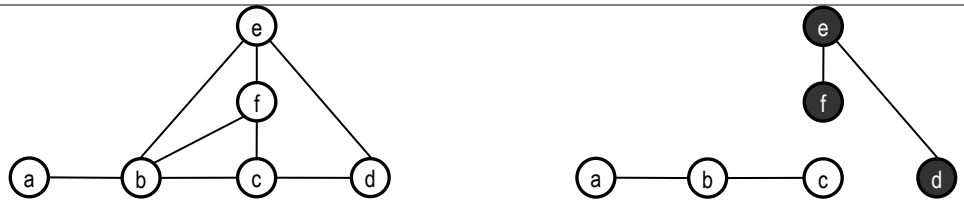
① 反身性：对于任何顶点 v，都有 v ~ v 成立；

② 对称性：u ~ v 仅当 v ~ u；

③ 传递性：对于任何顶点 u、v 和 w，只要 u ~ v 且 v ~ w，则必有 u ~ w。

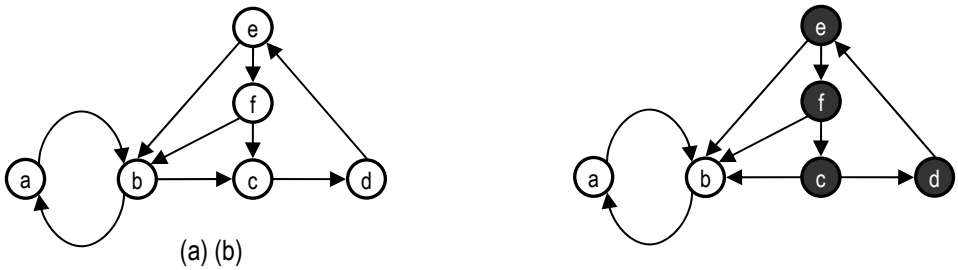
由此可知：

推论十.2 连通关系“~”是顶点集上的一个等价关系



(a) (b)

图十.4. (a) 无向连通图与(b)由两个连通分量组成的无向图



(a) (b)

图十.5. (a) 有向（强）连通图与(b)由两个（强）连通分量组成的有向图

根据等价关系“~”，可以将顶点集V划分为若干等价类，每一等价类都称作图G的一个连通分量（Connected compone nt）。由单个连通分量组成的图称为连通图（Connected grap h），有向连通图也可以称作“强连通图”（Strongly-connected graph）[十]。

10.1.8 森林、树以及无向图的生成树

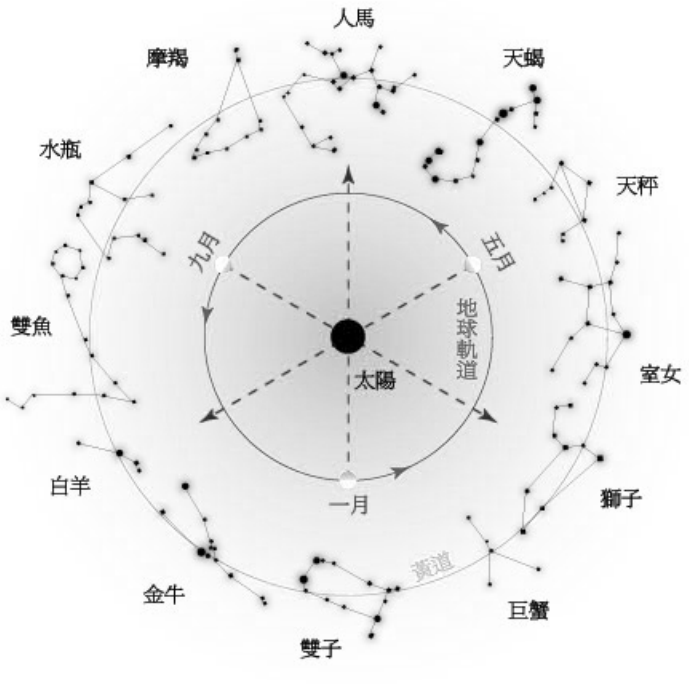
考察无向图 G = (V, E)。

定义十.8 若 G 中不含任何环路，则称之为森林（Forest）。

定义十.9 连通的森林称作树（Tree）。

不难看出，森林中的每一连通分量都是一棵树，反之亦然。

[十] 也可以定义所谓的“弱连通图”，即在其中任何一对顶点 u 和 v 之间，至少有一条从 u 到 v 的通路或者有一条从 v 到 u 的通路。



图十.6 若将星空图视作一个森林，则其中的每个星座都构成一棵树

关于树、森林和连通图，有如下一些性质：

观察结论十.8 设 G 为由 n 个顶点与 m 条边组成一幅无向图。

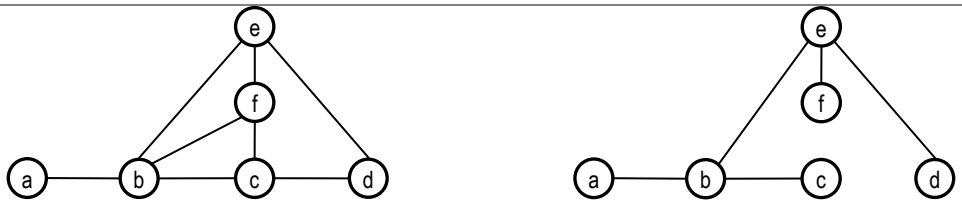
① 若 G 是连通的，则 $m \geq n-1$；

② 若 G 是一棵树，则 $m = n-1$；

③ 若 G 是森林，则 $m \leq n-1$。

其证明作为习题留给读者自行完成。

定义十.10 若 G 的某一生成子图 G'为一棵树，则称 G'为 G 的一棵生成树（Spanning tree）。



(a) (b)

图十.7. 无向连通图(a)及其生成树(b)

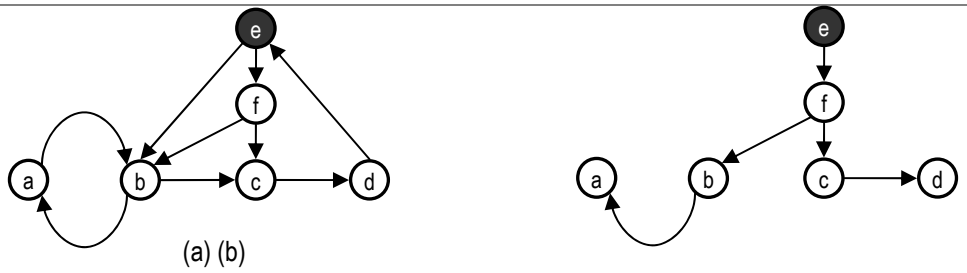
10.1.9 有向图的生成树

在有向图 G = (V, E)中，若存在某个顶点 s 满足 $V_r(G, s) = V$（即从 s 可到达所有顶点）且 s 到任一顶点的通路唯一，同时 G 中不含回路，则称 G 为一棵以 s 为根的有向树。

若 G 的某一生成子图 G'为一棵（以某一顶点 s ∈ V 为根的）有向树，则也称 G'为 G 的生成树。

考察有向连通图 G = (V, E)的一个无环生成子图 T = (V, E')，E' ⊆ E。

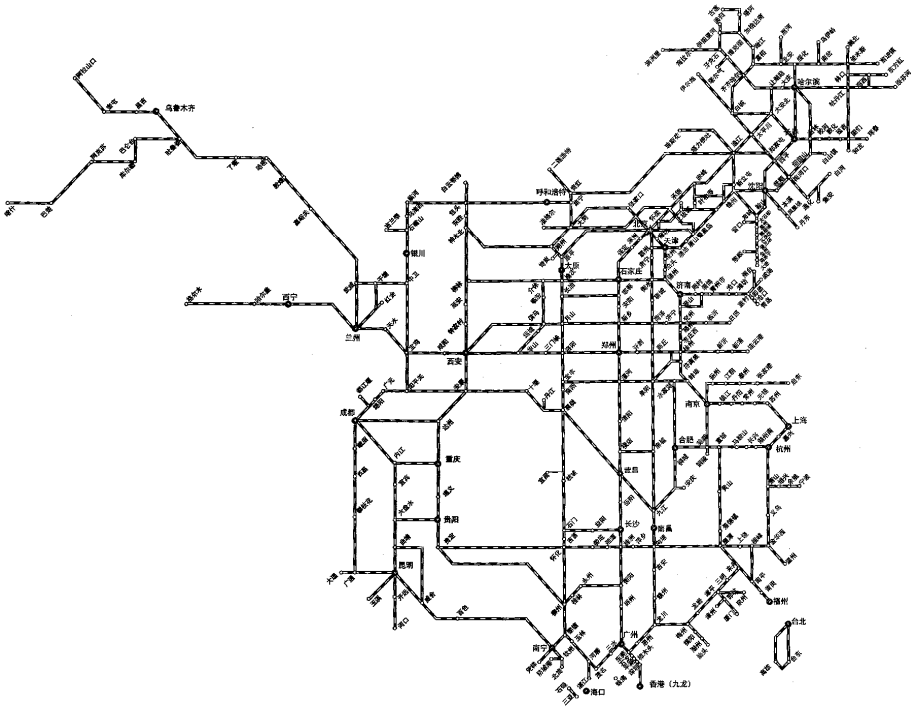
定义十.11若存在顶点 r ∈ V，使得对于任何顶点 v ∈ V，都有一条从 r 通往 v 的有向通路，则称 T 为 G 的一棵（以 r 为根的）生成树。



图十.8 有向连通图(a)及其已顶点e为根的生成树(b)

10.1.10 带权网络

有些时候，图不仅需要表示元素之间是否存在某种关系，而且还需要表示这一关系的某一细节。以铁路运输为例，可以用顶点表示城市，用顶点之间的边表示城市之间是否有铁路联接。然而，为了更为细致地描述铁路运输网，还需要记录每段铁路的长度、运输成本等信息。



图十.9 中国铁路客运网

为适应这类应用的需求，我们需要为每条边设置相应的数据域，以记录对应的信息。具体地，我们定义一个从边集 E 到实数的一个映射：

$$w : E \mapsto R$$

对于任一边 e ∈ E，weight(e)称作 e 的权重。

引入权重函数之后的图 G(V, E, weight())，称作带权图（Weighted graph）或带权网络（Weighted network），有时也简称为网络。

定义十.12 若σ = (u_0, u_1, u_2, …, u_k)是带权网络 G 中的一条路径，则 $|\sigma| = \sum_{i=1}^{k} w(u_{i-1}, u_i)$称作带权路径σ的权重或长度。

§10.2 抽象数据类型

正如前面所言，出于通用性的考虑，本章仅以有向图为例介绍图结构及其算法。

10.2.1 图

作为一种抽象数据类型，有向图必须支持以下操作：

表十.1 有向图ADT支持的操作

操作方法	功能描述
vNumber():	返回顶点集 V 的规模
eNumber():	返回边集 E 的规模
vertices():	返回所有顶点的迭代器
vPositions():	返回所有顶点位置的迭代器
edges():	返回所有边的迭代器
ePositions():	返回所有边位置的迭代器
areAdjacent(u, v):	判断顶点 u 和 v 是否相邻
remove(v):	将顶点 v 从顶点集中删除，并返回之
remove(e):	将边 e 从边集中删除，并返回之
insert(v):	在顶点集 V 中插入新顶点 v，并返回其位置
insert(e):	在边集 E 中插入新边 e，并返回其位置

支持以上操作的有向图，可以描述为如 代码十.1 所示的Java接口。

```
/*
 * （有向）图结构接口
 *无向图也可以看成是有向图，为此，只需将每条无向边替换为对称的一对有向边
 */

package dsa;

public interface Graph {
```

```
//取图中顶点、边的数目
  public int vNumber();

  public int eNumber();

//取图中所有顶点、顶点位置的迭代器
  public Iterator vertices();
  public Iterator vPositions();

//返回图中所有边、边位置的迭代器
  public Iterator edges();
  public Iterator ePositions();

//检测是否有某条边从顶点u指向v
  public boolean areAdjacent(Vertex u, Vertex v);
//取从顶点u指向v的边，若不存在，则返回null
  public Edge edgeFromTo(Vertex u, Vertex v);

//将顶点v从顶点集中删除，并返回之
  public Vertex remove(Vertex v);
//将边e从边集中删除，并返回之
  public Edge remove(Edge e);
//在顶点集V中插入新顶点v，并返回其位置
  public Position insert(Vertex v);
//在边集E中插入新边e，并返回其位置
  public Position insert(Edge e);
}
```

代码十.1 （有向）图结构接口

10.2.2 顶点

作为一种抽象数据类型，有向图的顶点必须支持以下操作：

表十.2 有向图顶点ADT支持的操作

操作方法	功能描述
getInfo()	返回顶点中记录的信息
setInfo(x)	设置顶点中记录的信息
outDeg()	返回顶点的出度
inDeg()	返回顶点的入度
outEdges()	返回所有出边构成的迭代器
inEdges()	返回所有入边构成的迭代器
outEdgePositions()	返回所有出边的位置构成的迭代器

操作方法	功能描述
inEdgePositions()	返回所有入边的位置构成的迭代器
getVPosInV()	返回当前顶点在所属的图的顶点集 V 中的位置
getStatus()	返回顶点的当前状态
setStatus(s)	将当前顶点的状态更新为 s，并返回原先的状态
getDStamp()	返回顶点的起始时间标签（针对 DFS）
setDStamp(s)	设置顶点的起始时间标签（针对 DFS）
getFStamp()	返回顶点的结束时间标签（针对 DFS）
setFStamp(s)	设置顶点的结束时间标签（针对 DFS）
getDistance()	返回顶点至起点的最短距离（针对 BFS）
setDistance(s)	设置顶点至起点的最短距离（针对 BFS）

支持以上操作的有向图顶点，可以描述为如 代码十.2 所示的Java接口。

```
/*
 * （有向）图的顶点结构接口
 *无向图也可以看成是有向图，为此，只需将每条无向边替换为对称的一对有向边
 */

package dsa;

public interface Vertex {
//常量
  final static int UNDISCOVERED = 0;//尚未被发现的顶点
  final static int DISCOVERED = 1;//已被发现的顶点
  final static int VISITED = 2;//已访问过的顶点

//返回当前顶点的信息
  public Object getInfo();
//将当前顶点的信息更新为x，并返回原先的信息
  public Object setInfo(Object x);

//返回当前顶点的出、入度
  public int outDeg();
  public int inDeg();

//返回当前顶点所有关联边、关联边位置的迭代器
  public Iterator inEdges();
  public Iterator inEdgePositions();
  public Iterator outEdges();
  public Iterator outEdgePositions();

//取当前顶点在所属的图的顶点集V中的位置
  public Position getVPosInV();
```

```
//读取、设置顶点的状态（DFS + BFS）
  public int getStatus();

  public int setStatus(int s);


//读取、设置顶点的时间标签（DFS）
  public int getDStamp();
  public int setDStamp(int s);
  public int getFStamp();
  public int setFStamp(int s);

//读取、设置顶点至起点的最短距离（BFS或BestFS）
  public int getDistance();
  public int setDistance(int s);

//读取、设置顶点在的DFS、BFS、BestFS或MST树中的父亲
  public Vertex getBFSParent();
  public Vertex setBFSParent(Vertex s);
}
```

代码十.2 （有向）图的顶点结构接口

10.2.3 边

作为一种抽象数据类型，有向图的边必须支持以下操作：

表十.3 有向图边ADT支持的操作

操作方法	功能描述
getInfo():	返回边中记录的信息
setInfo(x):	将当前边的信息更新为 x，并返回原先的信息
getEPosInE():	返回边在所属的图的边集 E 中的位置
getVPosInV(i):	返回 v[i]在顶点集 V 中的位置（i=0 或 1，分别对应于起点、终点）
getEPosInI(i):	返回边在其两个端点的关联边集 I(v[i])中的位置
getType():	返回边的类别（针对遍历）
setType(t):	设置边的类别（针对遍历）：将当前边的类别更新为 t，并返回原先的类别

支持以上操作的有向图的边，可以描述为如 代码十.3 所示的Java接口。

```
/*
 * （有向）图的边结构接口
 *无向图也可以看成是有向图，为此，只需将每条无向边替换为对称的一对有向边
```

```
 */

package dsa;

public interface Edge {
//常量

  final static int UNKNOWN = 0;//未知边


  final static int TREE = 1;//树边
  final static int CROSS = 2;//横跨边
  final static int FORWARD = 3;//前向跨边
  final static int BACKWARD = 4;//后向跨边

//返回当前边的信息（对于带权图，也就是各边的权重）
  public Object getInfo();
//将当前边的信息更新为x，并返回原先的信息
  public Object setInfo(Object x);

//取当前边在所属的图的边集E中的位置
  public Position getEPosInE();
//取v[i]在顶点集V中的位置（i=0或1，分别对应于起点、终点）
  public Position getVPosInV(int i);
//当前边在其两个端点的关联边集I(v[i])中的位置
  public Position getEPosInI(int i);

//读取、设置边的类别（针对遍历）
  public int getType();
  public int setType(int t);
}
```

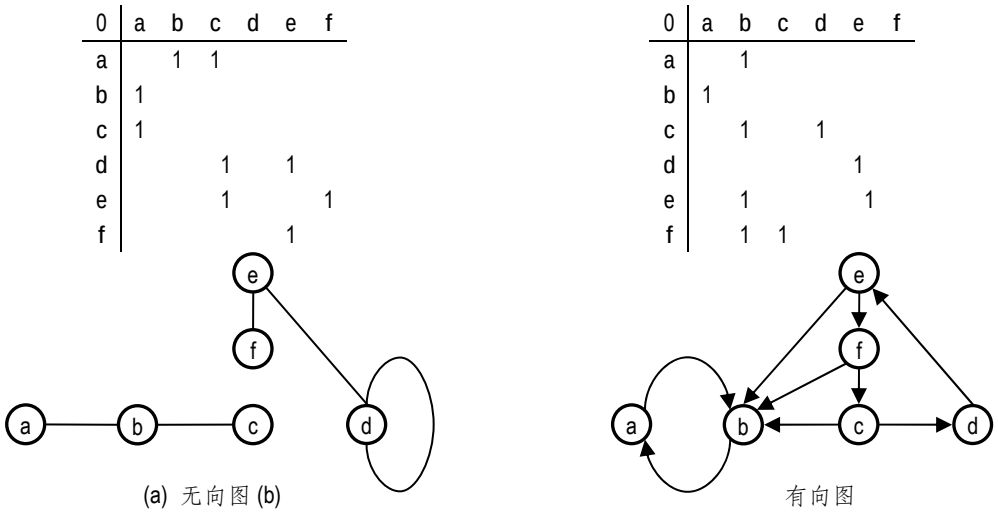
代码十.3 （有向）图的边结构接口

§10.3 邻接矩阵

10.3.1 表示方法

如上定义的图 ADT 有多种实现方式，其中最直接的就是基于邻接矩阵（Adjacency matrix）的实现。具体地，若图 G 中包含 n 个顶点，我们就使用一个 n×n 的方阵 A，并使每一顶点都分别对应于某一行（列）。既然图所描述的是这些顶点各自对应的元素之间的二元关系，故可以很自然地将任意一对元素 u 和 v 之间可能存在二元关系与矩阵 A 中对应的单元 A[u, v]对应起来：1 或 true 表示存

在关系，0 或 false 表示不存在关系。这一矩阵中的各个单元分别描述了一对元素之间可能存在的邻接关系，故此得名。



图十.10 借助邻接矩阵表示图

图十.10 给出了邻接矩阵的两个实例，其中(a)是无向图，(b)是有向图。为了查看方便，这里采用了一个约定：凡是空白的单元，其值默认为 0。

实际上，这一表示形式也可以推广至带权图，具体方法是，将每条边的权重记录在该边对应得矩阵单元中。

鉴于这种实现非常简单，在此不再详细讨论其实现细节，下面直接对其性能做一分析。

10.3.2 时间性能

借助邻接矩阵，可以简便而高效地实现图ADT、顶点ADT和边ADT中的各个方法。各方法的时间复杂度可以分别归纳为 表十.4、表十.5 和 表十.6，其中N×N为邻接矩阵的容量，N为事先确定一个足够大的整数。

表十.4 基于邻接矩阵实现的图ADT操作的时间复杂度

操作	时间复杂度	实现方法
vNumber()	$O(1)$	引入对应的变量
eNumber()	$O(1)$	引入对应的变量
vertices()	$O(N)$	从 N 行中检出 n 个有效单元
vPositions()	$O(N)$	从 N 行中检出 n 个有效单元
edges()	$O(N^2)$	从 N^2 个单元中检出有效单元
ePositions()	$O(N^2)$	从 N^2 个单元中检出有效单元
areAdjacent(u, v)	$O(1)$	直接检查单元 A[u, v]
remove(v)	$O(N)$	清除 v 对应的行、列
remove(e)	$O(1)$	若 e = (u, v)，则直接清除单元 A[u, v]
insert(v)	$O(N)$	从 N 行（列）中找出空闲的一行（一列）

操作	时间复杂度	实现方法
insert(e)	$O(1)$	若 e = (u, v)，则直接设置单元 A[u, v]

表十.5 基于邻接矩阵实现的顶点ADT操作的时间复杂度

操作	时间复杂度	实现方法
getInfo()	$O(1)$	引入对应的变量
setInfo(x)	$O(1)$	
outDeg()	$O(N)$	需遍历顶点对应的行或列[1]
inDeg()	$O(N)$	
outEdges()	$O(N)$	需遍历顶点对应的行
outEdgePositions()	$O(N)$	
inEdges()	$O(N)$	需遍历顶点对应的列
inEdgePositions()	$O(N)$	
getVPosInV()	$O(1)$	直接返回顶点对应行、列号
getStatus()	$O(1)$	引入对应的变量
setStatus(s)	$O(1)$	
getDStamp()	$O(1)$	
setDStamp(s)	$O(1)$	
getFStamp()	$O(1)$	
setFStamp(s)	$O(1)$	
getDistance()	$O(1)$	
setDistance(s)	$O(1)$	

表十.6 基于邻接矩阵实现的边ADT操作的时间复杂度

操作	时间复杂度	实现方法
getInfo()	$O(1)$	引入对应的变量
setInfo(x)	$O(1)$	
getEPosInE()	$O(1)$	直接返回矩阵中对应单元的位置
getVPosInV(i)	$O(1)$	直接返回顶点对应行、列号
getEPosInI(i)	$O(1)$	直接返回对应单元的行、列号
getType()	$O(1)$	引入对应的变量
setType(t)	$O(1)$	

10.3.3 空间性能

以上基于邻接矩阵的实现方法直观易懂、思路简明，而且能够高效地实现图的大多数 ADT 操作。

[1] 通过引入变量，可改进至 $O(1)$，但在插入、删除边时需更新头、尾顶点的度数记录。

如果仅仅需要表示顶点之间是否存在关系，则只需为矩阵 A 的每个单元分配一个比特位即可。如果是带权网络，则根据具体的权值函数 weight()，需要为每个单元分配一个整数或实数的空间。无论何种情况，每一单元均只占用常数量的空间。因此，如果图中含有 n = |V|个顶点，则对应的邻接矩阵需要占用 $\mathcal{O}(n^2)$的空间。这一空间效率太低，无法令人满意。具体来说，邻接矩阵的不足主要体现在以下两个方面。

首先，尽管n个顶点构成的图中最多可能包含$m = n^2$条边，但在大多数情况下，边的数目都远远达不到这个量级。以最常见的平面图[㈠]为例，可以证明必有$m = \mathcal{O}(n)$。因此，除非是处理稠密图，否则邻接矩阵中的大多数单元都将是重复闲置的。

另一方面，矩阵结构是静态的，通常都是事先估计一个较大的整数 N，然后创建一个 N×N 的矩阵。然而，图的规模往往都是动态变化的，因此如果 N 不是足够大，极有可能出现因空间“不足”而无法加入新顶点的情况——而实际上，此时系统并非没有更多空间可以提供。为了降低这种情况发生的概率，必须选用足够大的 N，而如此一来，单元闲置的程度也将加剧。

当然，这种实现方式并非一无是处。在处理小规模或稠密的图时，邻接矩阵简便易行的特点就可以充分发挥出来，成为解决这类问题的首选方案。

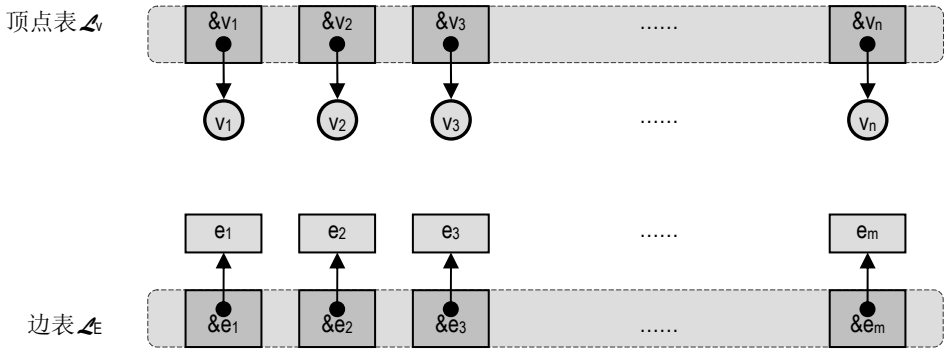
§10.4 邻接表

由上面的分析可知，邻接矩阵的空间效率之所以低，是因为其中大量的单元所对应的边有可能并未在图中出现，这也是静态向量结构普遍的不足。既然如此，为什么不像 第三章那样，将向量改进为列表呢？实际上，按照这一思路的确可以导出图结构的另一种表示与实现形式，即本节将要介绍的邻接表（Adjacency list）。图的这一实现方式不仅更为高效，而且就渐进复杂度的意义而言，邻接表结构所需占用的空间量将线性正比于图结构本身的复杂度——根据此前的定义，即图中实际包含的顶点数与边数之和。

10.4.1 顶点表和边表

按照定义，任一有限集 V 以及定义于其上的某一二元关系 E 都构成了一幅图 G = (V, E)，我们将 V 和 E 分别称作顶点集、边集。反过来，任何一幅图 G = (V, E)都是由对应的 V 和 E 确定的。因此，为了实现图结构，首先要着手解决顶点集和边集的表示和实现。

[㈠] Plan ar graphs，存在至少一种平面嵌入方案的图。直观地理解，就是在各边互不相交（于内部）的前提下，可以画在平面上的图。



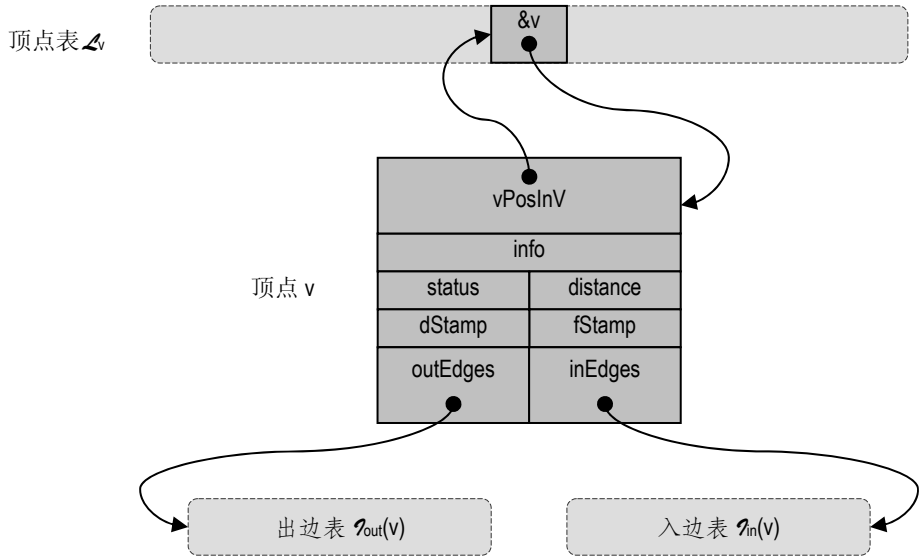
图十.11 用列表实现图的顶点集与边集

如 图十.11 所示，我们分别将图结构中的顶点和边组织为两个列表$\mathcal{L}_V$和$\mathcal{L}_E$。更准确地说，列表$\mathcal{L}_V$（$\mathcal{L}_E$）中的每个元素只是一个引用，分别指向集合V（E）中的某一个顶点（某一条边）。

当然，以上只是初步的构思，为了实现完整的图结构，我们还要详细地给出边和顶点的实现。

10.4.2 顶点与邻接边表

如 图十.12 所示，首先要为每个顶点v设置若干变量。其中，info是一个引用，它指向的对象将提供与顶点相关的具体信息。status、distance、dStamp和fStamp等变量是为支持遍历操作而设置的，稍后在第 §10.5 节将看到它们的具体作用。



图十.12 基于列表实现的顶点结构

引用 vPosInV 指向当前节点 v 在顶点表$\mathcal{L}_V$中的位置。

与每个顶点 v 相关联的边，也分别组织为列表形式。鉴于这里只考虑有向图，故实际的做法是将 v 的出边和入边分别组织为两个列表，并借助引用 outEdges 和 inEdges 分别指向它们。出（入）边表中的每一元素，实际上就是 v 的某一出（入）边在全局边表中的位置。

将出、入邻接边组织为列表形式是邻接边表法的主要特色，这种方法也因此得名。具体的实现参见 代码十.4。

```
/*
 * 基于邻接边表实现图的顶点结构
 */
package dsa;

public class Vertex_List implements Vertex {
//变量
  protected Object info;//当前顶点中存放的数据元素
  protected Position vPosInV;//当前顶点在所属的图的顶点表V中的位置
  protected List outEdges;//关联边表：存放以当前顶点为尾的所有边（的位置）
  protected List inEdges;//关联边表：存放以当前顶点为头的所有边（的位置）
  protected int status;//（在遍历图等操作过程中）顶点的状态
  protected int dStamp;//时间标签：DFS过程中该顶点被发现时的时刻
  protected int fStamp;//时间标签：DFS过程中该顶点被访问结束时的时刻
  protected int distance;//到指定起点的距离：BFS、Dijkstra等算法所确定该顶点到起点的距离
  protected Vertex bfsParent;//在最短距离树（BFS或BestFS）中的父亲

//构造方法：在图G中引入一个属性为x的新顶点
  public Vertex_List(Graph G, Object x) {
    info = x;//数据元素
    vPosInV = G.insert(this);//当前顶点在所属的图的顶点表V中的位置
    outEdges = new List_DLNode();//出边表
    inEdges = new List_DLNode();//入边表
    status = UNDISCOVERED;
    dStamp = fStamp = Integer.MAX_VALUE;
    distance = Integer.MAX_VALUE;
    bfsParent = null;
  }

//返回当前顶点的信息
  public Object getInfo() { return info; }
//将当前顶点的信息更新为x，并返回原先的信息
  public Object setInfo(Object x) { Object e = info; info = x; return e; }

//返回当前顶点的出、入度
  public int outDeg() { return outEdges.getSize(); }
  public int inDeg() { return inEdges.getSize(); }

//返回当前顶点所有关联边、关联边位置的迭代器
  public Iterator inEdges() { return inEdges.elements(); }
  public Iterator inEdgePositions() { return inEdges.positions(); }
  public Iterator outEdges() { return outEdges.elements(); }
  public Iterator outEdgePositions() { return outEdges.positions(); }
```

```
//取当前顶点在所属的图的顶点集V中的位置
  public Position getVPosInV() { return vPosInV; }


//读取、设置顶点的状态（DFS + BFS）

  public int getStatus() { return status; }
  public int setStatus(int s) { int ss = status; status = s; return ss; }

//读取、设置顶点的时间标签（DFS）
  public int getDStamp() { return dStamp; }
  public int setDStamp(int s) { int ss = dStamp; dStamp = s; return ss; }
  public int getFStamp() { return fStamp; }
  public int setFStamp(int s) { int ss = fStamp; fStamp = s; return ss; }

//读取、设置顶点至起点的最短距离（BFS）
  public int getDistance() { return distance; }
  public int setDistance(int s) { int ss = distance; distance = s; return ss; }

//读取、设置顶点在的DFS、BFS、BestFS或MST树中的父亲
  public Vertex getBFSParent() { return bfsParent; }
  public Vertex setBFSParent(Vertex s) { Vertex ss = bfsParent; bfsParent = s; return
ss; }
}
```

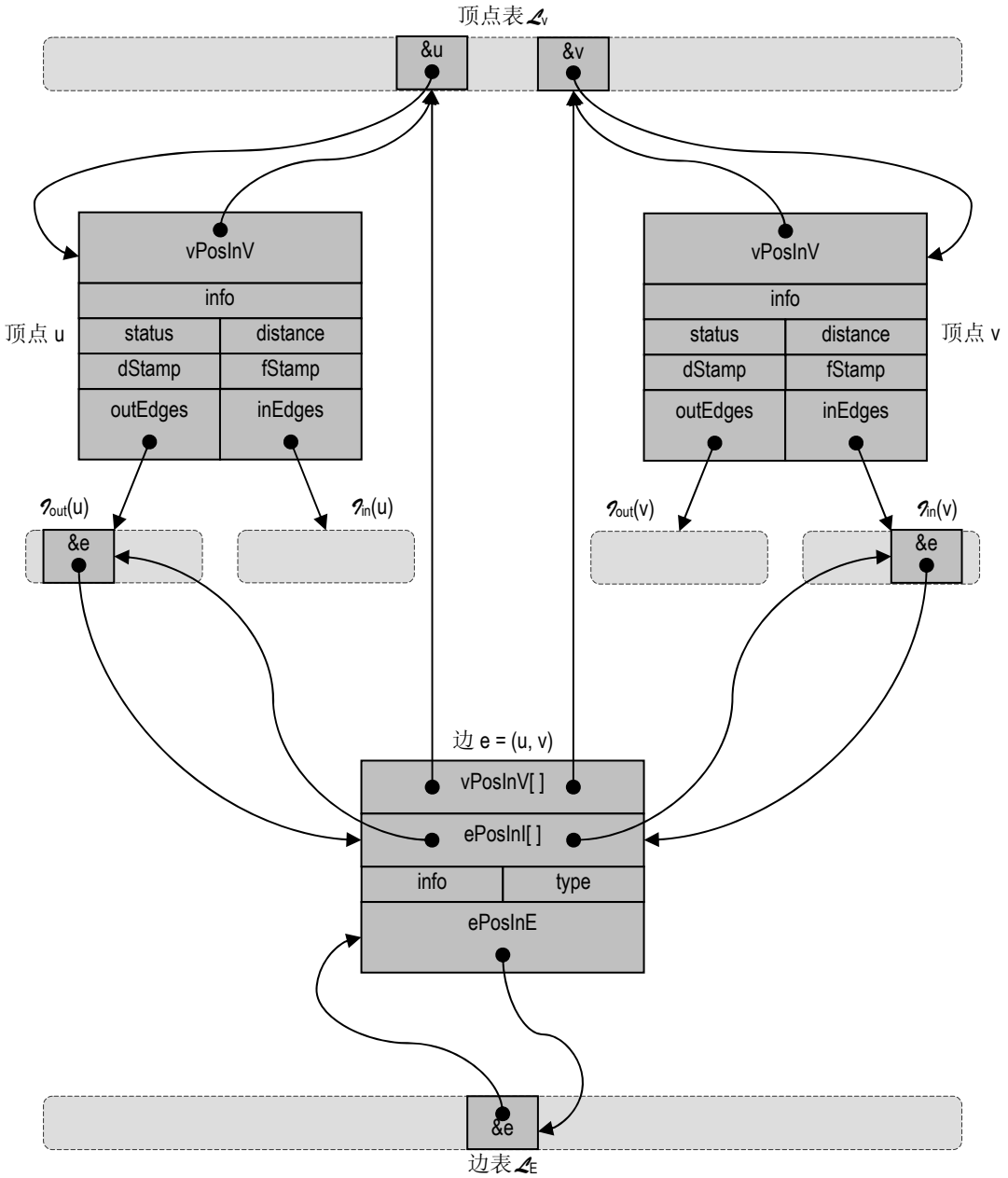
代码十.4 图的顶点类型实现

为了得到顶点的出、入度，这里直接调用了对应邻接边表的getSize()方法。根据第 3.2.4 节关于双向链表各种操作的时间复杂度的结论，这里的outDeg()和inDeg()方法都只需$O(1)$时间。

由于这里采用了列表结构，故出、入边及其位置的迭代器都可以直接调用列表结构相应的迭代器构造方法（参见第 §3.4 节）。

10.4.3 边

如 图十.13 所示，为描述边e的信息，首先要为其设置一个引用info，根据不同的应用问题，info所指向的对象将提供有关边e的不同类型的信息。在经过某些遍历（例如DFS）之后，图中的边将被划分为不同类型，这将由变量type来标识。



图十.13 基于列表实现的边结构

考察有向边 e = (u, v)。在 e 对应的边对象中，引用 ePosInE 指向当前边 e 在全局边表$\mathcal{A}_E$中的位置；引用vPosInV[0]和vPosInV[1]分别给出了起点u和终点v在顶点表$\mathcal{A}_V$中的位置；引用ePosInI[0]给出了边 e 在其起点 u 的出边邻接表 $\mathcal{T}_{out}(u)$中的位置，而引用 ePosInI[1]则给出了边 e 在其终点 v 的入边邻接表 $\mathcal{T}_{in}(v)$中的位置。

具体的实现参见 代码十.5。

```
/*
 * 基于邻接边表实现图的边结构
 */

package dsa;
```

```
public class Edge_List implements Edge {

//变量
  protected Object info;//当前边中存放的数据元素
  protected Position ePosInE;//当前边在所属的图的边表中的位置
  protected Position vPosInV[];//当前边两个端点在顶点表中的位置
  protected Position ePosInI[];//当前边在其两个端点的关联边表中的位置
      //约定：第0（1）个顶点分别为尾（头）顶点
      //禁止头、尾顶点相同的边
  protected int type;//（经过遍历之后）边被归入的类别

//构造方法：在图G中，生成一条从顶点u到v的新边（假定该边尚不存在）
  public Edge_List(Graph G, Vertex_List u, Vertex_List v, Object x) {
    info = x;//数据元素
    ePosInE = G.insert(this);//当前边在所属的图的边表中的位置
    vPosInV = new DLNode[2];//当前边两个端点在顶点表中的位置
    vPosInV[0] = u.getVPosInV(); vPosInV[1] = v.getVPosInV();
    ePosInI = new DLNode[2];//当前边在其两个端点的关联边表中的位置
    ePosInI[0] = u.outEdges.insertLast(this);//当前边加入u的邻接（出）边表
    ePosInI[1] = v.inEdges.insertLast(this);//当前边加入v的邻接（入）边表
    type = UNKNOWN;
  }

//返回当前边的信息
  public Object getInfo() { return info; }
//将当前边的信息更新为x，并返回原先的信息
  public Object setInfo(Object x) { Object e = info; info = x; return e; }

//取当前边在所属的图的边集E中的位置
  public Position getEPosInE() { return ePosInE; }
//取v[i]在顶点集V中的位置（i=0或1，分别对应于起点、终点）
  public Position getVPosInV(int i) { return vPosInV[i]; }
//当前边在其两个端点的关联边集I(v[i])中的位置
  public Position getEPosInI(int i) { return ePosInI[i]; }

//读取、设置边的类别（针对遍历）
  public int getType() { return type; }
  public int setType(int t) { int tt = type; type = t; return tt; }

}
```

代码十.5 图的边类型实现

其中采用的边构造算法，可以描述如下：

```
算法：Edge_List(G, u, v, x)
输入：图G以及其中的两个顶点u和v，对象x

输出：在图G中，生成一条从顶点u到v的新边，并以x作为该边的信息对象

{
  令新边的信息对象info = x;
  将新边插入至G的边表E中，并将返回的位置保存至ePosInE;
  将顶点u在顶点表V中的位置保存至vPosInV[0];
  将顶点v在顶点表V中的位置保存至vPosInV[1];
  将新边插入至u的出边邻接表;
  将新边插入至v的入边邻接表;
}
```

算法十.1 基于邻接表的边构造算法

10.4.4 基于邻接表实现图结构

综合以上基于邻接边表实现的顶点与边结构，可以进而实现图结构如 代码十.6 所示。

```
/*
 * 基于邻接边表实现图结构
 */

package dsa;

public class Graph_List implements Graph {
//变量
  protected List E;//容器：存放图中所有边
  protected List V;//容器：存放图中所有顶点

//构造方法
  public Graph_List() { E = new List_DLNode(); V = new List_DLNode(); }

//取图的边表、顶点表
  protected List getE() { return E; }
  protected List getV() { return V; }

//取图中顶点、边的数目
  public int vNumber() { return V.getSize(); }
  public int eNumber() { return E.getSize(); }

//取图中所有顶点、顶点位置的迭代器
  public Iterator vertices() { return V.elements(); }
  public Iterator vPositions() { return V.positions(); }
```

```
//返回图中所有边、边位置的迭代器
  public Iterator edges() { return E.elements(); }
  public Iterator ePositions() { return E.positions(); }


//检测是否有某条边从顶点u指向v

  public boolean areAdjacent(Vertex u, Vertex v)
  { return (null != edgeFromTo(u, v)); }
//取从顶点u指向v的边，若不存在，则返回null
  public Edge edgeFromTo(Vertex u, Vertex v) {
    for (Iterator it = u.outEdges(); it.hasNext();) {//逐一检查
      Edge e = (Edge)it.getNext();//以u为尾的每一条边e
      if (v == e.getVPosInV(1).getElem())//若e是(u, v)，则
        return e;//返回该边
    }
    return null;//若不存在这样的(u, v)，则返回null
  }

//将顶点v从顶点集中删除，并返回之
  public Vertex remove(Vertex v) {
    while (0 < v.outDeg())//将以v为尾的所有边
      remove((Edge)(((Vertex_List)v).outEdges.first()).getElem());//逐一删除
    while (0 < v.inDeg())//将以v为头的所有边
      remove((Edge)(((Vertex_List)v).inEdges.first()).getElem());//逐一删除
    return (Vertex)V.remove(v.getVPosInV());//在顶点表中删除v
  }
//将边e从边集中删除，并返回之
  public Edge remove(Edge e) {
    ((Vertex_List)e.getVPosInV(0).getElem()).outEdges.remove(e.getEPosInI(0));//从起
点的出边表中删除e
    ((Vertex_List)e.getVPosInV(1).getElem()).inEdges.remove(e.getEPosInI(1));//从终
点的入边表中删除e
    return (Edge)E.remove(e.getEPosInE());//从边表中删除e
  }

//在顶点集V中插入新顶点v，并返回其位置
  public Position insert(Vertex v) { return V.insertLast(v); }
//在边集E中插入新边e，并返回其位置
  public Position insert(Edge e) { return E.insertLast(e); }
}
```

代码十.6 基于邻接表的图结构

这里主要涉及三个算法，具体分析如下。

■ 判断任意一对顶点是否相邻

```
算法：areAdjacent(u, v)
输入：一对顶点u和v
输出：判断是否有某条边从顶点u指向v

{
  取顶点u的出边迭代器it;
  通过it逐一检查u的每一条出边e;
  一旦e的终点为v，则报告true;
  若e的所有出边都已检查过，则返回false;
}
```

算法十.2 相邻顶点判别算法

areAdjacent()算法需要检查 u 的所有出边，因为对每条边的检查都可以在常数时间内完成，故该算法需要运行 $\mathcal{O}$(1+outDeg(u))时间。

这一算法还有一个对称的版本：逐一检查 v 的所有入边，看其中是否有一条边的起点为 u。与上述分析同理，这一版本的时间复杂度为 $\mathcal{O}$(1+inDeg(v))。

将这两个版本融合起来，可以得到更为优化的一个算法：首先花费常数的时间在 outDeg(u)和 inDeg(v)中确定小者，然后在规模更小的邻接边表中进行查找。因为 outDeg(u)和 inDeg(v)可以在 $\mathcal{O}$(1)时间内得到，故这一混合式算法只需运行 $\mathcal{O}$(1+min(outDeg(u), inDeg(v))时间。

■ 删除边

```
算法：RemoveEdge(e)
输入：边e = (u, v)
输出：将边e从边集E中删除
{
  从起点u的出边邻接表中删除e;
  从终点v的入边邻接表中删除e;
  从边表E中删除e;
}
```

算法十.3 边删除算法

e在出、入边邻接表和全局边表中的位置，都可以在$\mathcal{O}$(1)时间内通过e中相应的变量直接获得，因此根据第 3.2.4 节关于双向链表各种操作的时间复杂度的结论，这里的三次删除操作都可以在常数时间内完成，故RemoveEdge()算法只需运行$\mathcal{O}$(1)时间。

■ 删除顶点

```
算法：removeVertex(v)
输入：顶点v
输出：将顶点v从顶点集V中删除
{
  扫描v的出边邻接表，（调用removeEdge()算法）将所有边逐一删除；
  扫描v的入边邻接表，（调用removeEdge()算法）将所有边逐一删除；

  在顶点表V中删除v；

}
```

算法十.4 顶点删除算法

首先，将 v 从顶点表 V 中删除只需常数时间。因此，既然 removeEdge()算法的每次调用只需常数时间，故 removeVertex(v)算法的运行时间将取决于其出、入边邻接表的总规模，具体而言，就是 $\mathcal{O}$(1+outDeg(v)+inDeg(v)) = $\mathcal{O}$(1+max(outDeg(v), inDeg(v)))。

§10.5 图遍历及其算法模板

图算法是一个庞大的家族，其中最为基本的一类操作就是本节将要着重讨论的图遍历算法，这也是几乎所有图算法的基础。与第 4.3.4 节介绍的树遍历算法一样，图遍历算法的目的也是要访问所有的顶点，而且仅访问一次。与树遍历不同的是，图遍历还要求访问所有的边，而且仅访问一次。实际上，树遍历也有这个要求，只不过鉴于树不含环路的特点，“对每个顶点访问且仅访问一次”已经蕴涵了“对每条边访问且仅访问一次”。诸如深度优先搜索等图遍历算法，不要求访问到所有的边，而且还要求对各边进行分类，以供后续的算法利用。

与树遍历算法一样，作为一种基本的操作，图遍历算法必须能够高效地实现。幸运的是，正如我们马上就会看到的，深度优先和广度优先这两类最重要的图遍历算法都可以在线性时间内完成，准确地说，如果顶点数和边数分别为 n 和 m，则这两类算法都只需 $\mathcal{O}$(n+m)时间。显然，这已经是最好的结果了。

尽管图遍历算法的具体形式不尽相同，但就其本质而言，它们还是具有很大的共性。因此，这里将其中共同的部分抽取出来，整理成如 代码十.7 所示的模板形式。

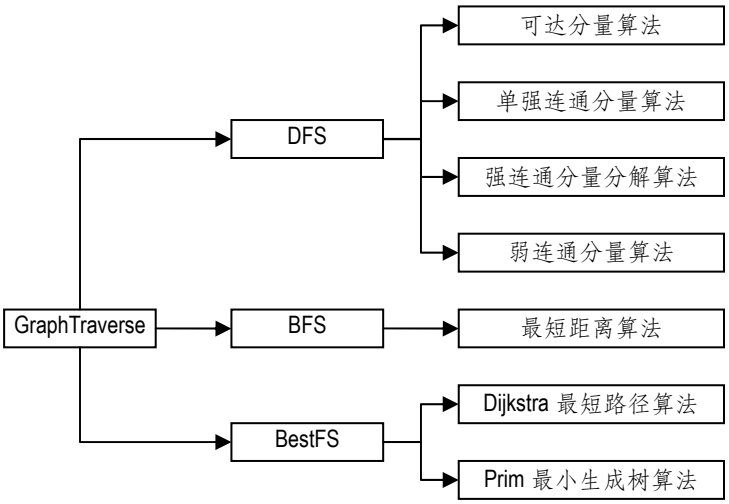
```
/*
 * （有向）图的遍历算法模板
 */

package dsa;

public abstract class GraphTraverse {
//常量
```

```
  final static int UNDISCOVERED = 0;//尚未被发现的顶点
  final static int DISCOVERED = 1;//已被发现的顶点
  final static int VISITED = 2;//已访问过的顶点
  final static int UNKNOWN = 0;//未知边
  final static int TREE = 1;//树边
  final static int CROSS = 2;//横跨边
  final static int FORWARD = 3;//前向跨边
  final static int BACKWARD = 4;//后向跨边


//变量

  protected Graph G;//图

//构造方法
  public GraphTraverse(Graph g) { G = g; }

//将G中各顶点的标志、各边的分类复位（s为遍历起点）
  protected void reset(Vertex s) {
    for (Iterator it = G.vertices(); it.hasNext();) {//所有
      Vertex v = (Vertex)it.getNext();//顶点的
      v.setStatus(UNDISCOVERED);//状态置为UNDISCOVERED
      v.setDistance(Integer.MAX_VALUE);//最短距离初始化为无穷大
    }
    for (Iterator it = G.edges(); it.hasNext();)//所有边
      ((Edge)it.getNext()).setType(UNKNOWN);//置为UNKNOWN
  }

//遍历过程中对顶点v的具体访问操作的模板：取决于、服务于具体的算法algorithm()
  protected abstract Object visit(Vertex v, Object info);

//基于遍历操作实现的其它算法的模板：s为起始顶点，info向算法传递参数及保存算法的返回信息
  public abstract Object algorithm(Vertex s, Object info);

//遍历算法模板
  protected abstract Object traverse(Vertex v, Object info);//从顶点v出发做遍历
}
```

代码十.7 图遍历算法的模板类

除了图对象 G 之外，这里还定义了三个抽象方法。其中 traverse()方法是具体的遍历算法，访问到每个顶点时的具体执行的操作则由 visit()方法决定。利用不同的 traverse()方法，可以实现其它特定功能的算法，这些具体的算法由 algorithm()方法描述。

在后续各节中，我们将介绍最主要的三种遍历算法：深度优先遍历、广度优先遍历以及最佳优先遍历。这些遍历算法是众多图算法的基础，如 图十.14 所示：基于深度优先遍历，可以实现可达分量算法（第 10.6.5 节）、单强连通分量算法（第 10.6.6 节）、强连通分量分解算法（第 10.6.7 节）以及弱连通分量算法（第 10.6.8 节）；基于广度优先遍历，可以实现最短距离算法（第 10.7.5 节）；基于最佳优先遍历，可以实现Dijkstra最短路径算法（第 10.8.5 节）和Prim-Jarnik最小生成树算法（第 10.8.7 节）。



图十.14 基于图遍历模板的各种遍历算法及其应用实例

§10.6 深度优先遍历

就其重要性而言，深度优先搜索（Depth-First Search - DFS）无疑是最为重要的图遍历算法，基于 DFS 的框架，可以导出和建立不计其数的图算法。以本章开头英雄特修斯从迷宫中救出公主的故事为例，为了在迷宫中寻找从起点（迷宫入口）至终点（公主所在位置）的通路，我们可以将迷宫中不同位置之间的联接关系表示为一幅图，于是该问题就转化为判断（对应于起点和终点的）两个顶点之间的可达性问题，利用 DFS 算法可以漂亮地加以解决，而且一旦找到通路，不仅可以顺利抵达终点与公主会合，还能安全地沿这条通路返回。这方面其它的例子还包括：确定某一顶点所属的连通分量，构造图的生成树或生成森林，以及找出图的关节点（Articulation Point）等等。

10.6.1 深度优先遍历算法

深度优先遍历的策略非常直接而简单，可以归纳为一个原则：凡是可以并值得前进的位置，就向前进。这里的“可以”是指在当前位置与下一位置之间存在一条联边（如果是有向图，则是有向边），而“值得”是指下一位置尚未到达过。

如果从顶点 v 开始深度优先遍历，我们首先访问 v。接下来，从 v 的出边中找到一条尚未访问的边 e ＝ (v, u)，并递归地从 u 处继续遍历。直到当前顶点 v 的所有出边都已访问过（或者 v 根本就没有出边），才不再继续递归，并从 v 处回溯。这一过程不断继续，直到重新回溯到最初的起始顶点。

深度优先遍历算法为每个顶点 v 设置了一个标记 status[v]，以表明其所处的状态。在遍历过程中，顶点的状态分为以下三种：

- UNDISCOV ERED：在开始深度优先遍历之前，每个顶点都被初始化为这一状态，这表明顶点尚未被遍历算法发现；
- DISCOVER ：一旦遍历算法抵达某一顶点，该顶点旋即切换至这一状态，这表明顶点已经被发现，但尚未访问完毕。
- VISIT ED：任一被发现的顶点迟早都会被访问完成，并从该位置回退至此前的位置，此时该顶点将切换至该状态，表明对其访问已经结束。

此外，经过深度优先遍历，每个顶点 v 还都会被标注上两个时间标签：dStamp[v]和 fStamp[v]。这里，可以假想着有一只系统时钟 clock，在遍历开始时它指向刻度 0。在遍历过程中，每向前一步或向后回退一步，时钟都向前走一个时间单位。具体来说，dStamp[v]就是顶点 v 被发现的时刻，而 fStamp[v]则是算法从 v 处回溯的时刻。

深度优先遍历算法可以描述为 算法十.5：

```
算法：DFS(G, s)
输入：有向图G及其中的顶点s
输出：从s出发，对G做深度优先遍历，并对访问到的边进行分类，对访问到的顶点做时间标签
假设：首次调用之前，所有顶点的状态都已置为UNDISCOVERED，所有边的分类置为UNKNOWN，clock置为0
{
  dStamp[s] = clock++;
  将s标记为DISCOVERED;
  调用visit(s)对s进行访问;
  逐一检查s的每一条出边e = (s, u)，根据顶点u的不同状态分别处理 {
    ① 若u尚未被发现（即status[u]＝UNDISCOVERED），则
        将e标记为树边（TREE）;
        调用DFS(G, u)递归搜索;
    ② 若u已被发现但对其的访问尚未完成（即status[u]＝DISCOVERED），则
        将e标记为后向跨边（BACKWARD）;
    ③ 若对u的访问已经完成（即status[u]＝VISITED），则比较s和u的时间标签
        若u比s更早被发现（即dStamp(u)<dStamp(s)），则
          将e标记为横跨边（CROSS）;
        否则
          将e标记为前向跨边（FORWARD）;
  }//至此，s的所有出边都已访问，故在回溯之前令
  fStamp[s] = clock++;
  将s标记为VISITED;
}
```

算法十.5 深度优先算法DFS()

10.6.2 边分类

从 算法十.5 可以看出，深度优先遍历算法会将所有被访问到的边划分为四种类型，它们各自的含义以及判别规则可以归纳如下：

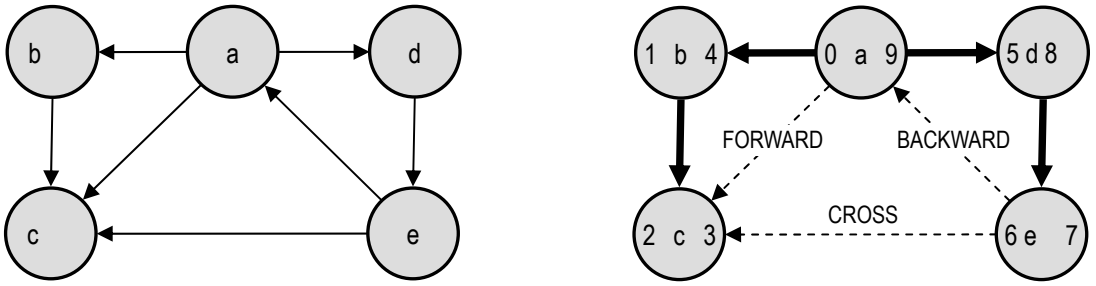
❶ TREE：树边。如果从当前顶点 v 前进到一个处于 UNDISCOVERED 状态的顶点 u，则(v, u)就是一条树边。

❷ BACKWARD：后向跨边。如果发现当前顶点 v 的某一出边(v, u)指向一个处于 DISCOVERED 状态的顶点 u，则(v, u)就是一条后向跨边。

❸ FORWARD：前向跨边。如果发现当前顶点 v 的某一出边(v, u)指向一个处于 VISITED 状态的顶点 u，而且 u 比 v 更晚被发现，则(v, u)就是一条前向跨边。

❹ CROSS：横跨边。如果发现当前顶点 v 的某一出边(v, u)指向一个处于 VISITED 状态的顶点 u，而且 u 比 v 更早被发现，则(v, u)就是一条横跨边。

图十.15 给出了一个深度优先遍历的实例，该有向图包含 5 个顶点和 7 条边，遍历从顶点a开始。这里约定，若当前顶点有多条出边，则依相邻顶点的编号为序逐一检查。



(a) 有向图 (b) 深度优先遍历结果

图十.15 有向图及其经过深度优先遍历之后的结果

顶点 a 首先在时刻 0 被 DFS()算法发现，并被标记为 DISCOVERED。在接下来的时刻 1 和时刻 2，顶点 b 和顶点 c 将相继被发现，根据规则❶，边(a, b)和(b, c)将被标记为树边（TREE）。

顶点 c 没有任何出边，故不再递归，于是对它的访问在时刻 3 宣告完成，算法 将 c 被标记为 VISITED 后回溯到顶点 b。此时，顶点 b 不再有尚未访问的出边，于是 b 在时刻 4 转入 VISITED 状态，算法回溯到顶点 a。随后，算法发现从顶点 a 有一条尚未访问的边指向已被标记为 VISITED 的顶点 c，而且 a 比 c 更早被发现（0 < 2），根据规则❸，(a, c)应属于前向跨边。

接下来，算法将分别在时刻 5 和 6 陆续发现处于 UNDISCOVERED 状态的顶点 d 和 e，并将它们标记为 DISCOVERED。随后，算法发现从顶点 e 有一条边指向处于 DISCOVERED 状态的顶点 a，根据规则❷，(e, a)应属于后向跨边。算法进而发现从顶点 e 有另一条边指向处于 VISITED 状态的顶点 c，而且 e 比 c 更晚被发现（6 > 2），根据规则❹，(e, c)应属于横跨边。

10.6.3 可达分量与 DFS 树

从顶点 s 出发对图 G = (V , E) 做深度优先遍历之后，我们把所有被访问到的顶点所组成的集合记作 $V_d(G, s)$，把所有被标记为 TREE 的边所构成的集合记做 $E_d(G, s)$，记子图 $T_d(G, s) = (V_d(G, s), E_d(G, s))$。

$E_d(G, s)$中的边为什么被称作“树边”？它们与“树”有何关系？它们与什么树有关系？

首先，我们注意到以下事实：

观察结论十.9 G 中存在一条由 s 到 u 的通路（即 $u \in V_r(G, s)$），当且仅当

① 从 s 出发的深度优先遍历会抵达 u；

② 遍历结束后，u 被标记为 VISITED；并且

③ 在 $T_d(G, s)$中存在一条从 s 到 u 的通路。

通过对通路长度做数学归纳，不难证明上述结论，我们将此留给读者自行完成。

根据 观察结论十.9，可以得到如下结论：

定理十.1 $V_r(G, s) = V_d(G, s)$

也就是说，从 s 出发的深度优先遍历所访问到顶点，恰好就是从 s 可达的顶点。

观察结论十.10 在 $T_d(G, s)$中，从 s 到任一顶点有且仅有一条通路。

〖证明〗

“有”显然，只需证明“仅有”。

反证，取两条通路$\rho_1 = \{s, e_1, v_1, e_2, v_2, \ldots, e_i, v_i, e_{i+1}, w\}$和$\rho_2 = \{s, f_1, u_1, f_2, u_2, \ldots, f_j, u_j, f_{j+1}, w\}$，$v_i \neq u_j$。

根据 DFS()算法，$e_{i+1} \in E_d(s)$当且仅当 w 作为 v_i 的一个处于 UNDISCOVERED 状态的邻居被发现，并随即切换至 DISCOVERED 状态；同理，$f_{j+1} \in E_d(s)$当且仅当 w 作为 u_j 的一个处于 UNDISCOVERED 状态的邻居被发现，并随即切换至 DISCOVERED 状态。然而，这两种情况只可能发生其一。 ❑

观察结论十.11 $T_d(G, s)$中不含环路。

〖证明〗

反证，任取 T 中的环路$\rho = \{v_m, e_1, v_1, e_2, v_2, \ldots, e_m, v_m\}$。

既然所有顶点的 dStamp 标签互异，故不失一般性，假设 v_m 是最后一个被发现的——在这 m 个顶点中，v_m 的 dStamp 最大，即 $dStamp(v_i) < dStamp(v_m)$，$1 \le i < m$。

考察有向边 $e_1 = (v_m, v_1) \in E_d(v)$。这条边之所以能够被标记为 TREE，只有一种可能：当 DFS()算法抵达 v_m 之后发现这条边，而且当时 v_1 的状态是 UNDISCOVERED。这就意味着，v_1 在 v_m 之后才被发现，即 $dStamp(v_1) > dStamp(v_m)$，矛盾。 ❑

现在，由 观察结论十.10 和 观察结论十.11，再根据第 10.1.9 节的定义可知：

定理十.2 $T_d(G, s))$是 $G|_{V_d(G, s)}$的一棵以 s 为根的生成树。

特别地，

推论十.3 若所有顶点都从 s 可达，即 $V_d(G, s) = V$，则 $T_d(G, s))$是 G 的一棵以 s 为根的生成树。

至此，我们已经能够直接回答本节开头所提出的那些问题——$E_d(G, s)$中的边被称作“树边”是名副其实的，因为它们构成了一棵以遍历起始顶点为根的树 $T_d(G, s))$。因为树 $T_d(G, s))$是通过对图 G 做深度优先遍历而导出的，故我们称之为 DFS 树。

10.6.4 深度优先遍历算法模板

同样地，出于通用性方面的考虑，我们还是通过抽象类的方式实现DFS算法，具体如 代码十.8 所示。

```
/*
 * （有向）图的深度优先遍历算法模板
 */

package dsa;

public abstract class DFS extends GraphTraverse {
//变量
  protected static int clock = 0;//遍历过程中使用的计时钟

//构造方法
  public DFS(Graph g) { super(g); }

//深度优先遍历算法
  protected Object traverse(Vertex v, Object info) {//从顶点v出发，做深度优先查找
    if (UNDISCOVERED != v.getStatus())  return null;//跳过已访问过的顶点（针对非连通图）
    v.setDStamp(clock++);   v.setStatus(DISCOVERED);  visit(v, info);//访问当前顶点
    for (Iterator it = v.outEdges(); it.hasNext();) {//检查与顶点v
      Edge e = (Edge)it.getNext();//通过边e = (v, u)
      Vertex u = (Vertex)e.getVPosInV(1).getElem();//相联的每一顶点u
      switch (u.getStatus()) {//根据u当前的不同状态，分别做相应处理
        case  UNDISCOVERED ://若u尚未被发现，则
          e.setType(TREE);//e被归类为“树边”
          traverse(u, info);//从u出发，继续做深度优先查找
          break;
        case  DISCOVERED ://若u已经被发现，但对其访问尚未结束，则
          e.setType(BACKWARD);//将e归类为“后向跨边”
```

```
          break;
        default ://VISITED，即对u的访问已经结束
          if (u.getDStamp() < v.getDStamp())//若相对于v，u被发现得更早，则
            e.setType(CROSS);//将e归类为“横跨边”
          else//否则
            e.setType(FORWARD);//将e归类为“前向跨边”
          break;
      }
    }//至此，v的所有邻居都已访问结束，故
    v.setFStamp(clock++); v.setStatus(VISITED);//将v标记为VISITED
    return null;//然后回溯
  }
}
```

代码十.8 深度优先遍历算法模板

这里只实现了 traverse()方法，具体的 visit()及 algorithm()算法依然保持灵活性，可以根据具体的问题进行扩充。

根据 观察结论十.9，DFS(G, v)算法只能访问到从v可达的（即$V_r(v)$中的）那些顶点，而其它不可达的顶点呢？

为了使所有顶点都能被访问到，我们只需对 DFS()算法稍做扩充。请注意 traverse()方法的第一句，实际上，在引入这一句之后，我们只需对所有顶点逐一调用 traverse()方法，即可遍历到所有顶点，而且不会重复访问顶点。

如此实现的深度优先遍历算法（traverse()方法）的时间复杂度，是否如本章开头所言是线性的呢？为了统计，我们只需将消耗于每个顶点的时间累计起来。

观察结论十.12 对于 G 中从 s 可达的每一顶点 $v \in V_r(G, s)$

① traverse(v)算法递归调用一次且仅一次；

② 若不计 visit()调用，每次调用只需 $\mathcal{O}(1+outDeg(v))$时间（下层递归所需时间由对应顶点分摊）。

〖证明〗

由 观察结论十.9 可直接得到①，故只需证明②。

我们注意到，消耗于顶点 v 的时间不外乎以下两部分：

- 检查、更新 v 的状态，设置 v 的时间标签
- 对 v 的每条出边 e = (v, u)，根据顶点 u 的不同状态，分别处理

前一类操作共需常数时间，后一类操作对每条边也可以在常数时间内完成。

证毕。 ❑

于是，若不计对顶点具体的访问操作（visit()）所需的时间，则有如下结论：

定理十.3 从顶点 s 开始对图 G 的深度优先遍历，可以在 $\mathcal{O}(|V_r(G, s)| + |E|_{V_r(G, s)}|)$时间内完成。

〖证明〗

根据 观察结论十.12，DFS()算法总体的运行时间为

$$\sum_{v \in V_r(G, s)} \mathcal{O}(1+outDeg(v)) = \mathcal{O}\left(\sum_{v \in V_r(G, s)} 1 + \sum_{v \in V_r(G, s)} outDeg(v) \right)$$

再根据 观察结论十.2，上式等于

$$\mathcal{O}(n + m) = \mathcal{O}(|V_r(G, s)| + |E|_{V_r(G, s)}|)$$

证毕。 ❑

代码十.8 中的traverse()算法也可以改写为迭代版本，这作为作业留给读者完成。

10.6.5 可达分量算法

在处理图结构时，我们经常需要执行的一项操作就是：给定有向图G中的一个顶点s，计算出s所对应的可达分量$V_r(G, s)$（定义十.6）。

■ 算法

根据 观察结论十.9，$u \in V_r(G, s)$ 当且仅当从s出发的深度优先搜索会发现u并最终将其标记为VISITED。因此，基于在第 10.6.4 节建立的深度优先遍历算法模板（代码十.8），通过实现如 代码十.9 所示的visit()及algorithm()算法，就可以解决可达分量的计算问题。

```
/*
 * （有向）图基于DFS模板的可达分量算法
 */

package dsa;

public class DFSReachableComponent extends DFS {
//构造方法
  public DFSReachableComponent(Graph g) { super(g); }

//DFS过程中对顶点v的具体访问操作（info实际上是一个栈，记录所有可达的顶点）
  protected Object visit(Vertex v, Object info)
  { ((Stack)info).push(v); return null; }

//基于DFS的可达性算法：s为起始顶点（info实际上是一个栈，记录所有可达的顶点）
  public Object algorithm(Vertex s, Object info) {
    reset(s);
    Stack S = new Stack_Array();//保存从起始顶点可达的顶点
    traverse(s, info);//DFS
    return null;
  }
```

```
}
```

代码十.9 基于深度优先遍历的可达性判别算法

实际上，这里的 algorithm()算法并不复杂：直接调用 traverse()算法，从指定的顶点 s 出发做深度优先遍历；相应地，在对每个被发现的顶点进行访问时，visit()方法所做的只不过是将新发现的顶点压入栈 info 中。于是，待到遍历结束时，栈 info 中就保存了从 s 出发可达的所有顶点。如果需要枚举出这些顶点，只需遍历栈 info，并输出其中的各个顶点。

■ 效率

根据第 第二章有关栈结构实现效率的分析结论，这里的visit()方法只需常数时间，故根据 定理十.3 可知：

> **定理十.4 通过深度优先遍历，图 G = (V, E)中任一顶点 v 所对应的可达分量都可以在 $\mathcal{O}(|V_r(G, v)| + |E|_{V_r(G, v)}|)$时间内计算出来。**

■ 连通分量

这一算法同样适用于无向图。在这种情况下，算法所解决的实际上就是所谓的连通分量问题——确定某一顶点 s 所属的连通分量。这一算法的具体实现留给读者。

10.6.6 单强连通分量算法

本节讨论的问题是：对于图G中任一顶点s，计算出s所在的强连通分量C(G, s)（定义十.7）。

■ 镜像

对于任何边 e = (v, u)，我们称 R(e) = (u, v) 为 e 的镜像边，也就是说，R(e)的起点（终点）就是 e 的终点（起点）。对于任一有向图 G = (V, E)，我们称 R(E) = { R(e) | e ∈ E}为 E 的镜像边集，也就是说，集合 R(E)是由 E 中各边的镜像边组成的，反之亦然。此时，我们也称 R(G) = (V , R(E)) 为 G 的镜像图。

> **引理十.1** $C(G, s) = V_r(G, s) \cap V_r(R(G), s)$

该引理的证明留给读者自行完成。

这一引理指出，s 所在的连通分量，就是 s 在图 G 中的可达分量与 s 在图 R(G)中的可达分量的公共部分。

■ 算法

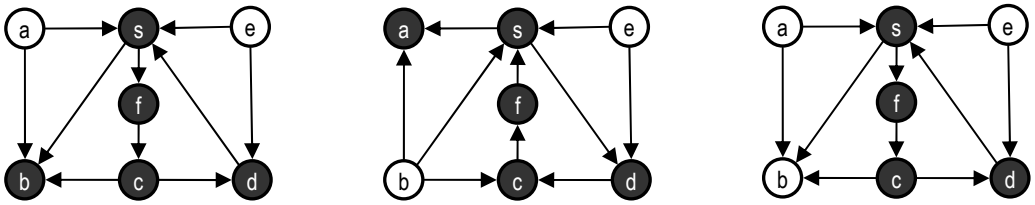
由此可以立即导出一个单强连通分量算法：

```
算法：DFSConnectedComponent(G, s)
输入：有向图G以及其中的一个顶点s
输出：s在G中所属的强连通分量C(G, s)
```

```
{
  调用DFSReachability(G, s)算法，计算出s在图G中的可达分量Vr(G, s);
  由G构造其镜像图R(G);
  调用DFSReachability(R(G), s)算法，计算出s在图R(G)中的可达分量Vr(R(G), s);
  找出Vr(G, s)和Vr(R(G), s)的公共部分，即得到C(G, s);
}
```

算法十.6 基于深度优先遍历的强连通分量算法

这一算法的具体实现，我们也留给读者。



图十.16 用黑色顶点表示的(a) Vr(G, s)、(b) Vr(R(G), s)以及(c) C(G, s)

■ 效率

这个算法的性能如何？

R(G)可以在$\mathcal{O}$(|V| + |E|)的时间内构造出来；根据 定理十.4，两次DFSReachability()调用分别需要$\mathcal{O}$(| V_r(G, v) | + | $E|_{V_r(G, v)}$ |)和$\mathcal{O}$(| V_r(R(G), v) | + | $E|_{V_r(R(G), v)}$ |)时间；为了找出V_r(G, s)和V_r(R(G), s)的公共部分，需要$\mathcal{O}$(| V_r(G, v) | + | $E|_{V_r(G, v)}$ | ＋ | V_r(R(G), v) | + | $E|_{V_r(R(G), v)}$ |)时间。由此可以得出如下结论：

定理十.5 通过深度优先遍历，图 G 中任一顶点 s 所在的强连通分量，可以在 $\mathcal{O}$(n + m)时间内计算出来。

实际上，若只要求计算出单独一个强连通分量，则利用第 §10.7 节的广度优先遍历算法，同样可以实现 算法十.6。

10.6.7 强连通分量分解算法

本节讨论的问题是：对于任一图 G，计算出组成 G 的所有强连通分量。比如，在某些场合，我们需要反复检测一对顶点是不是相互强连通的。此时，就可以将图分解为多个强连通分量。借助这一预处理结果，此后对于任何一对顶点，只要检查它们是否属于同一强连通分量，即可判断它们之间的强连通性。

利用第 10.6.6 节的 算法十.6，可以马上得到一个这样的算法：对图G中的每一个顶点v调用DFSConnectedComponent(G, v)算法，得到v所在的强连通分量。

不过，这一算法的效率却无法令人满意。由于 DFSConnectedComponent()算法每次需要运行$\mathcal{O}$(n + m)时间，故 n 次调用总共需要 $\mathcal{O}$(n(n + m))的时间。能否更快呢？

实际上，利用深度优先遍历，完全可以在$O(n+m)$的时间内对有向图进行强连通域分解。有兴趣的读者可以自行设计并实现这一算法。

10.6.8 浓缩图与弱连通性

■ 浓缩图

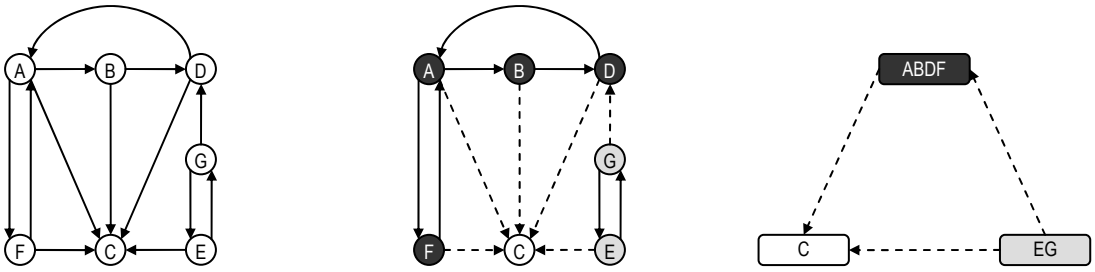
考察有向图 G = (V, E)。

如果 G 可以分解为 p 个强连通分量$\{S_1, S_2, \ldots, S_p\}$，则可以构造 $G\downarrow = (V\downarrow, E\downarrow)$如下：

$$V\downarrow = \{s_1, s_2, \ldots, s_p\}$$

$$E\downarrow = \{(s_i, s_j) \mid i\neq j \text{ 且有某条边从 } S_i \text{（中的某个顶点）指向 } S_j \text{（中的某个顶点）}\}$$

也就是说，$V\downarrow$中的p个顶点分别对应于p个强连通分量，$E\downarrow$中的边分别对应于强连通分量之间的联接关系。如 图十.17 所示，我们将如此构造出来的$G\downarrow$称作G的浓缩图（Condesation graph）。



(a) 有向图 G　　(b) G 的三个强连通分量 (c)　　G 的浓缩图

图十.17　有向图的强连通分量以及对应的浓缩图

根据 观察结论十.6，可以直接得出如下结论：

推论十.4 浓缩图不含环路。

■ 弱连通性

定义十.13　对于任意一对顶点 u 和 v，如果存在从 u 到 v 的通路或者从 v 到 u 的通路，则称它们“弱连通的”（Weakly connected）。

定义十.14　如果图 G 中每一对顶点之间都是弱连通的，则称 G 为弱连通图（Weakly-connected graph）。

那么，如何来判断一幅图是不是弱连通的呢？

引理十.2 图 G 是弱连通的，当且仅当 $G\downarrow$ 是一条通路。

该引理的证明留给读者。

■ 算法

根据 引理十.2，可以马上得到如下算法：

```
算法：WeakConnectivity(G)
输入：有向图G
输出：判断G是否弱连通
{
  调用强连通分量分解算法;
  由分解结果构造浓缩图G↓;
  检查G↓是否为一条通路;
}
```

算法十.7 有向图弱连通性判定算法

§10.7 广度优先遍历

与深度优先遍历算法一样，广度优先搜索（Breadth-First Search - BFS）也是一种基本而重要的一种遍历算法，基于这一遍历操作，也可以导出一系列的图算法，比如最短距离算法。

如果将图的深度优先遍历比作树的后序遍历（第 4.3.4 节），那么图的广度优先遍历就对应于树的层次遍历（第 4.3.5 节）。

10.7.1 广度优先遍历算法

广度优先搜索的过程并不复杂，在搜索过程中，我们始终维护一个活跃节点集 A，其策略可以归纳为一个原则：只有在集合 A 中的节点都已访问过后，才用其中各节点的后继替代它们，构成新的集合 A。当然，最初的集合 A 只有一个节点，即指定的遍历起点；当集合 A 变成空，而且没有新的顶点可以补充时，搜索即告完成。

与深度优先遍历算法相同，广度优先遍历算法也为每个顶点 v 设置了一个标记 status[v]，以指示其所处的状态，而且在遍历过程中顶点的状态也分为 DISCOVERED、DISCOVERED 和 VISITED 三种。

广度优先遍历算法可以描述为 算法十.8：

```
算法：BFS(G, s)
输入：有向图G及其中的顶点s
输出：从s出发，对G做广度优先遍历，并对访问到的边进行分类
假设：首次调用之前，所有顶点的状态都已置为UNDISCOVERED，所有边的分类置为UNKNOWN
{
  创建一个空队列Q;
  将s标记为DISCOVERED，并将s加入队列Q;
  调用visit(s, null)对v进行访问;
  在Q变空之前，不断迭代 {
    从Q中取出队首顶点v;
```

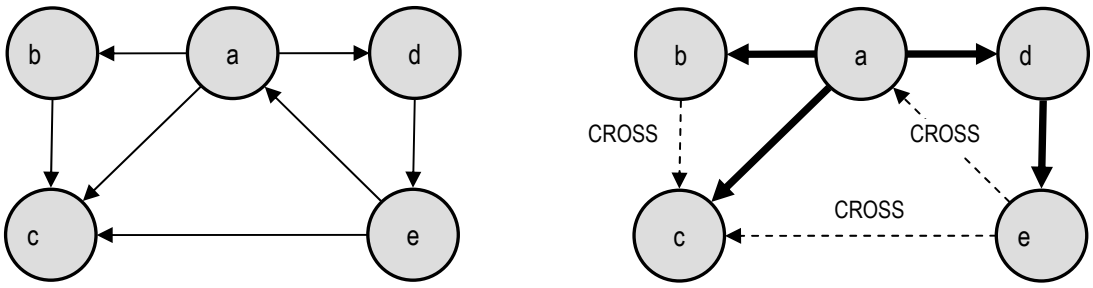
```
    逐一检查v的每一后继顶点u {
      若u尚未被发现（即status[u]＝UNDISCOVERED），则
        将有向边(v, u)标记为树边（TREE）；
        将u标记为DISCOVERED;
        令u加入队列Q;
        调用visit(u, v)对u进行访问;
      否则
        将有向边(v, u)标记为跨边（CROSS）；
    }//至此，v的所有出边都已访问，故
    将v标记为VISITED;
  }
}
```

算法十.8 广度优先算法BFS()

10.7.2 边分类

从算法十.8可以看出，通过广度优先遍历算法，可以将所有被访问到的边划分为两种类型，它们各自的含义以及判别规则也与深度优先遍历算法相仿：若当前顶点v前进到一个处于UNDISCOVERED状态的顶点u，则(v, u)就是一条树边（TREE）；否则，无论u是处于DISCOVERED或VISITED状态，(v, u)都是一条跨边（CROSS）。

图十.18给出了一个广度优先遍历的实例，该有向图包含5个顶点和7条边，遍历从顶点a开始。这里约定，若当前顶点有多条出边，则依相邻顶点的编号为序逐一检查。



(a) 有向图 G (b) 对 G 的广度优先遍历结果

图十.18 有向图及其经过广度优先遍历之后的结果

经过初始化，顶点a被标记为DISCOVERED，并加入队列Q。随后进入迭代循环。

首先，顶点a作为队首顶点出队。它的后继顶点b、c和d将依次接受检查。由于这三个顶点的状态都是UNDISCOVERED，故都再被标记为DISCOVERED之后，相继加入队列Q中；它们对应的边(a, b)、(a, c)和(a, d)均被标记为树边（TREE）；顶点a转入VISITED状态。

在随后的一轮迭代中，顶点b出队。它的后继顶点只有c，而c已经处于DISCOVERED状态，故边(b, c)被标记为跨边（CROSS），顶点b转入VISITED状态。

接下来，顶点 c 出队。由于它没有任何后继顶点，故直接转入 VISITED 状态。

然后，是顶点 d 出队。它的后继顶点 e 将从 UNDISCOVERED 状态切换至 DISCOVERED 状态并加入到队列 Q 中，边(d, e)被标记为树边（TREE），d 则转入 VISITED 状态。

最后，顶点 e 出队。它的后继顶点 a 和 c 此时已经处于 VISITED 状态，故边(e, a)和(e, c)被标记为树边（TREE），而 e 则转入 VISITED 状态。至此，队列 Q 已空，遍历结束。

10.7.3 可达分量与 BFS 树

从顶点 s 出发对图 G = (V , E) 做广度优先遍历之后，我们把所有被访问到的顶点所组成的集合记作 $V_b(G, s)$，把所有被标记为 TREE 的边所构成的集合记做 $E_b(G, s)$，记子图 $T_b(G, s) = (V_b(G, s), E_b(G, s))$。

与 DFS 算法类似地，我们也可以得到一下结论：

定理十.6 $V_r(G, s) = V_b(G, s)$

也就是说，从 s 出发的广度优先遍历所访问到顶点，恰好也是从 s 可达的顶点。

定理十.7 $T_b(G, s))$是 $G|_{V_b(G, s)}$的一棵以 s 为根的生成树。

推论十.5 若所有顶点都从 s 可达，即 $V_b(G, s) = V$，则 $T_b(G, s))$是 G 的一棵以 s 为根的生成树。

具体证明过程，情读者参照第 10.6.3 节自行补充。

由此可见，$E_b(G, s)$中的边之所以被称作“树边”，也是因为它们构成了一棵以遍历起始顶点为根的树 $T_b(G, s))$。同样地，因为树 $T_b(G, s))$是通过对图 G 做广度优先遍历而导出的，故我们称之为 BFS 树。

10.7.4 广度优先遍历算法模板

与DFS()算法一样，出于通用性方面的考虑，我们仍然采用抽象类实现BFS算法，具体如 代码十.10 所示。

```
/*
 * （有向）图的广度优先遍历算法模板
 */

package dsa;

public abstract class BFS extends GraphTraverse {
//构造方法
  public BFS(Graph g) { super(g); }

//广度优先遍历算法
  protected Object traverse(Vertex s, Object info) {//从顶点s出发，做广度优先查找
    if (UNDISCOVERED != s.getStatus())  return null;//跳过已访问过的顶点（针对非连通图）
    Queue Q = new Queue_List();//借用一个队列
```

```
    s.setStatus(DISCOVERED);//发现s后
    Q.enqueue(s);//随即令其入队
    visit(s, null);//并访问之
    while (!Q.isEmpty()) {//在队列变空之前
      Vertex v = (Vertex)Q.dequeue();//取出队首顶点v
      for (Iterator it = v.outEdges(); it.hasNext();) {//检查与顶点v
        Edge e = (Edge)it.getNext();//通过边e = (v, u)
        Vertex u = (Vertex)e.getVPosInV(1).getElem();//相联的每一顶点u
        if (UNDISCOVERED == u.getStatus()) {//若u尚未被发现，则
          e.setType(TREE);//将e归类为树边
          u.setStatus(DISCOVERED);//发现u后
          Q.enqueue(u);//随即令其入队
          visit(u, v);//并访问之
        } else {//若u已被发现，甚至已访问完毕（有向图），则
          e.setType(CROSS);//将e归类为跨边
        }

      }//至此，v的所有邻居都已访问结束，故

      v.setStatus(VISITED);//将v标记为VISITED
    }//while
    return null;
  }
}
```

代码十.10 广度优先遍历算法模板

与DFS()算法的实现一样，这里只实现了BFS()算法的traverse()方法，具体的visit()及algorithm()算法则可以根据具体的问题进行扩充。

另外，为了能够访问到所有的顶点而又不至于重复访问，我们需要在 traverse()方法的开头加上一句，以忽略已访问过的顶点。

参考第 10.6.4 节对深度优先遍历算法时间复杂度的分析方法，我们也可以得出类似的结论。

定理十.8 从顶点 s 开始对图 G 的广度优先遍历，可以在 $\mathcal{O}(|V_r(G, s)| + |E|_{V_r(G, s)}|)$时间内完成。

10.7.5 最短距离算法

■ 最短距离

最短距离问题是一般意义上最短路径问题的一个特例：假设有向图 G 中各边的长度相等（不妨设为 1），s 为其中指定的一个顶点，找出从 s 通往其余每个顶点的最短通路（如果不存在，则理解为无穷大）。

■ 广度优先搜索

实际上，利用广度优先搜索，可以很好地解决这一问题。

观察结论十.13 在广度遍历过程中的任一时刻

① 任一顶点处于 UNDISCOVERED 状态，当且仅当它尚未加入队列 Q；

② 任一顶点处于 DISCOVERED 状态，当且仅当它正在队列 Q 中；

③ 任一顶点处于 VISITED 状态，当且仅当它曾经加入过队列 Q，但现在已经出队。

〖证明〗

根据 代码十.10，我们不难注意到：一旦v被标记为DISCOVERED，便随即加入队列Q；一旦v出队，则随即被标记为VISITED。 ❑

观察结论十.14 各顶点将按照从 s 到它们的距离被发现，并按照这一次序被访问（调用 visit()）。

〖证明〗

同样地，从 代码十.10 可以看出：在任何时刻，队列Q中顶点都是按照各自到s的最短距离顺序排列的（队首顶点最近）。

因此根据队列先进先出的性质，本命题立即得证。 ❑

■ 算法

根据 观察结论十.13 和 观察结论十.14，可以设计出最短距离算法如 代码十.11 所示：

```
/*
 * （有向）图基于BFS的最短距离算法
 */

package dsa;

public class BFSDistance extends BFS {
//构造方法
  public BFSDistance(Graph g) { super(g); }

//顶点访问操作：在本算法中，info是顶点v的前驱
  protected Object visit(Vertex v, Object info) {
    if (null == info)//v为BFS的起始顶点
      v.setDistance(0);
    else
      v.setDistance(((Vertex)info).getDistance()+1);//设置v到s的距离 = 前驱的距离+1
    return null;
  }

//基于BFS实现的最短距离算法：s为起始顶点，info向算法传递参数
  public Object algorithm(Vertex s, Object info) {
    reset(s);
    traverse(s, info);//BFS：到起点的最短距离记录在各顶点的distance域中
```

```
    return null;
  }
}
```

代码十.11 基于广度优先遍历的最短距离算法

这一算法是对BFS()算法模板（代码十.10）的扩充，实质的工作不多，无非是实现了具体的algorithm()和visit()方法而已。

在 algorithm()方法中，首先将所有顶点的最短距离初始化为无穷大（系统支持的最大整数Integer.MAX_VALUE），然后调用 traverse(s)，从顶点 s 出发做广度优先遍历。请注意，这里的 visit()方法通过模板中定义的 info 对象传递参数，即被访问顶点的前驱。如果顶点 v 没有前驱，则说明 v 就是起点 s，故可以将其最短距离设置为 0；否则，v 的最短距离将在其前驱的最短距离的基础上增加一个单位。

■ 效率

不难看出，这里所实现visit()方法的每次调用只需常数时间，故根据 定理十.8 有：

定理十.9 通过广度优先遍历，图 G = (V, E)中任一顶点 v 到其它顶点的最短距离可以在 $O(n + m)$时间内计算出来，其中 n = |V|，m = |E|。

§10.8 最佳优先遍历

最佳优先遍历可以看作是一般遍历算法的推广，其遍历策略也非常直观而且简单。

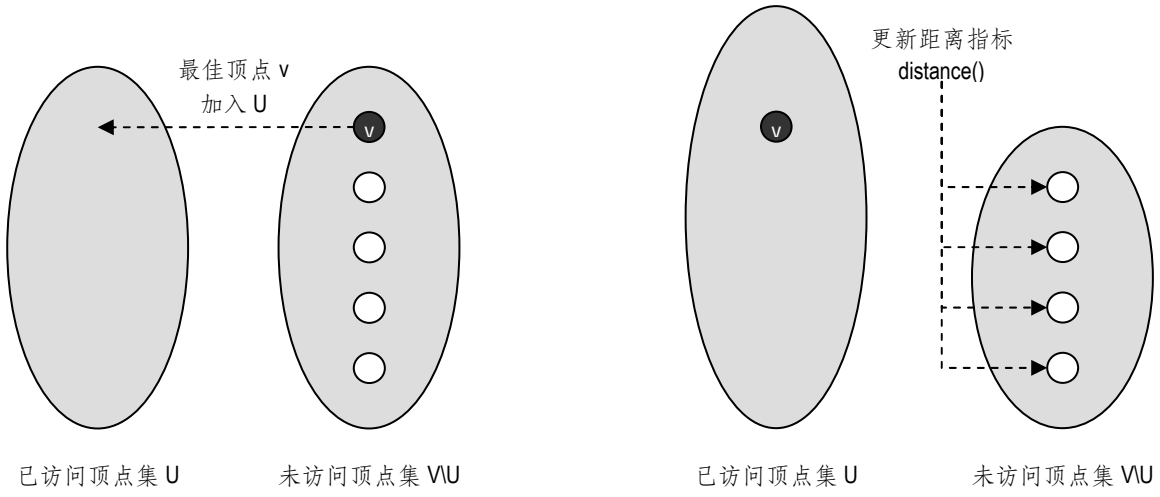
10.8.1 最佳优先遍历算法

■ 构思与框架

一般地，我们给图中每个顶点 v 设置一个特定的距离指标 distance(v)，这一指标反映了顶点 v 到已访问顶点集 U 的某种“最短距离”。请注意，这里的“距离”概念是广义的，“最短距离”指标的具体定义将取决于不同的应用问题。通常，这一指标越小，表示某种意义上的更优；而任一时刻具有最短距离的顶点则被称作最佳顶点，这一遍历算法也由此得名。

但就总体思路而言，解决这些问题的算法具有惊人的相似性，使得它们都可以统一在最佳优先遍历算法的一般性框架之下。

在进行最佳优先遍历之前，各顶点的距离指标都被指定为某一初始值。在遍历过程的每一次迭代中，我们首先从尚未访问到的顶点中挑选出距离指标最小者，并将其加入到已访问顶点集中；然后，逐一检查尚未访问到的各个顶点，并更新其距离指标。这一过程不断重复，直到所有顶点都已被访问到。



图十.19 最佳优先遍历

■ 算法描述

最佳优先遍历的具体过程可以描述为 算法十.9：

```
算法：BestFS(G, s)
输入：有向图G及其中的顶点s
输出：从s出发，对G做最佳优先遍历，并对访问到的边进行分类
假设：调用之前，所有顶点的状态都已置为UNDISCOVERED，所有边的分类置为UNKNOWN
{
  若s已被访问（status[s] != UNDISCOVERED），则直接返回;
  令Distance(s) = 0; //最近 = 最佳

  不断地 {
    在所有尚未访问过的顶点中，找出距离已访问点集最近的顶点v;
    调用visit(v)，对v进行访问，并将其加入已访问点集（将其状态设置为status[v] = VISITED）;
    调用updateDistanceAfter(v)，更新各顶点到已访问点集的最短距离;
  } 直到所有顶点都已被访问
}
```

算法十.9 最佳优先算法BestFS()

10.8.2 最佳优先遍历算法模板

这里，如 代码十.12 所示，我们还是以抽象类的形式实现最佳优先遍历算法BestFS：

```
/*
 * （有向）带权图最佳优先遍历
 */

package dsa;
```

```
public abstract class BestFS extends GraphTraverse {
//构造方法
  public BestFS(Graph g) { super(g); }

//更新尚未访问的顶点到已访问点集的最短距离（取决于具体的算法）
  protected abstract void updateDistanceAfter(Vertex v);

//最佳优先遍历算法
  protected Object traverse(Vertex s, Object info) {//从顶点s出发，做最佳优先遍历
    if (UNDISCOVERED != s.getStatus())  return null;//跳过已访问过的顶点（针对非连通图）
    s.setDistance(0);//设置s到已访问点集的距离

    Vertex v;//最佳顶点
    while (null != (v = bestVertex())) {//只要还有最佳顶点
      visit(v, null);//在发现并访问v之后
      updateDistanceAfter(v);//更新尚未访问的顶点到已访问集的最短距离
    }//while
    return null;

  }


//顶点访问操作：在本算法中，info无用
  protected Object visit(Vertex v, Object info) {
    v.setStatus(VISITED);//设置“已访问”标记
    return null;
  }

//基于BestFS实现的最短距离算法：s为起始顶点，info向算法传递参数
  public Object algorithm(Vertex s, Object info) {
    reset(s);//初始化，标记复位
    traverse(s, info);//BestFS：到起点的最短距离记录在各顶点的distance域中
    return null;
  }

//从尚未访问的顶点中选出最佳者
//对于Dijkstra算法而言，就是与已访问集连通、距离最近的顶点（及距离不是无穷的最近顶点）
//若没有这样的顶点，则返回null
  protected Vertex bestVertex() {
    int bestValue = Integer.MAX_VALUE;//最佳指标（越小越好）
    Vertex bestVertex = null;//最佳顶点
    for (Iterator it = G.vertices(); it.hasNext();) {//逐个检查
      Vertex u = (Vertex)it.getNext();//各个
      if (UNDISCOVERED == u.getStatus())//尚未访问的顶点u
        if (bestValue > u.getDistance())//若u到已访问点集距离更近，则
        { bestValue = u.getDistance();  bestVertex = u; }//更新最佳记录
```

```
    }
    if ((null != bestVertex) && (null != bestVertex.getBFSParent()))//最佳顶点与其父亲
之间的联边
      G.edgeFromTo(bestVertex.getBFSParent(), bestVertex).setType(TREE);//被标记为
TREE

    return bestVertex;
  }
}
```

代码十.12 最佳优先遍历算法模板

可以看到，其中只有一个抽象方法—— updateDistanceAfter(v)。正如我们马上就要看到的，根据这一抽象方法的不同实现，最佳优先遍历算法可以用来求解不同的问题，比如最短路径问题、最小生成树问题等等。

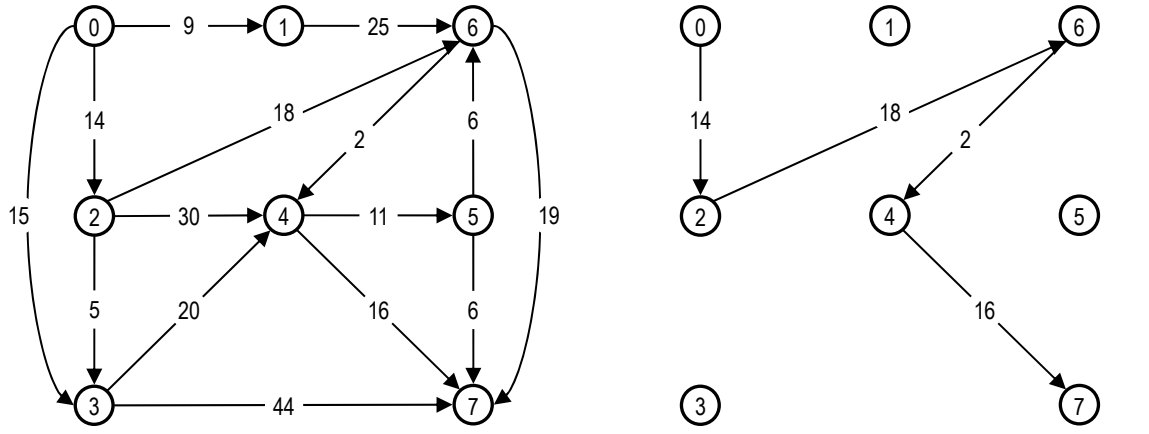
10.8.3 最短路径

在很多应用领域中，有向带权图都被用来描述某个网络，比如通讯网络、交通网络等。这种情况下，各边的权重就对应于两点之间信道的成本或交通费用。于是，一类典型的问题就是，在任意指定的两点之间如果存在通路，那么最小的联络成本或费用是多少？从这个意义上讲，最优的联络方案又是什么？

这类问题都属于本节将要讨论的最短路径问题，我们将从其形式化定义入手，通过对最优解所具有性质的刻画，找出普遍的规律，最终得到求解最短路径的Dijkstra算法，并基于第 10.8.2 节的最佳优先遍历模板（代码十.12）实现这一算法。

■ 问题及定义

给定有向带权图 G = (V, E)以及其中的一个顶点 s∈V，我们关心的问题是：对于 V 中的任一顶点 v，若存在从 s 到 v 的通路，那么其中最短的通路有多长？该通路由哪些边构成？



(a) 有向带权图 (b) 顶点 0、7 之间最短路径长度为 14+18+2+16 = 50

图十.20 有向带权图及其中顶点之间的最短路径

也就是说，我们需要找出从顶点s出发通往其它各个顶点的最短路径，故顶点s也称作起点或源点（Source）。图十.20(a)给出了这样的一个具体实例，其中的顶点 0 被指定为起点。图十.20(b)则给出了从顶点 0 通往顶点 7 的最短路径及其长度。

若存在从 s 通往节点 v 的最短路径，则其长度也称作从“s 到 v 的最短距离”，记作$\delta(s, v)$。

■ 存在性

为了保证“最短路径”的存在性，通常我们都假定所有边的权重为正数（不存在的边，可以理解为“权重等于无穷大”）。在这一条件下，我们可以注意到以下事实：

观察结论十.15只要顶点 v 是从起点 s 可达的，则从 s 通往节点 v 的最短路径必然存在，且$\delta(s, v)$必然唯一存在。

〖证明〗

考察从 s 通往 v 的所有简单路径，将它们组成集合$\mathcal{P}(s, v) = \{\sigma_1, \sigma_2, \ldots\}$，并将它们的长度组成集合$\mathcal{L}(s, v) = \{|\sigma| \mid \sigma \in \mathcal{P}(s, v)\}$。

既然所有边的权重均为正数，故只要最短路径$\pi(s, v)$存在，则它也必然会出现在$\mathcal{P}(s, v)$中。

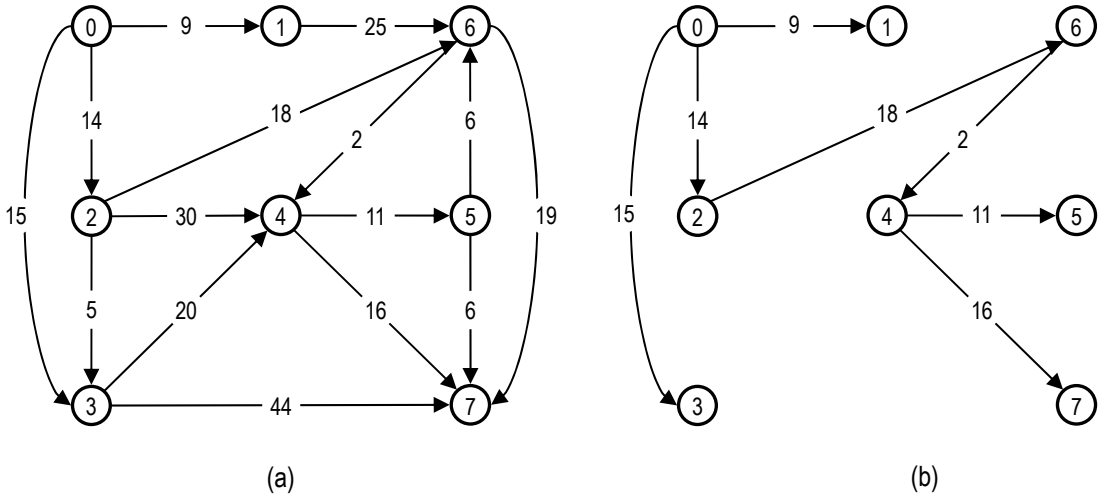
一方面，根据 观察结论十.4，$|\mathcal{P}(s, v)| < \infty$，$|\mathcal{L}(s, v)| < \infty$。另一方面，既然从s出发可以到达v，故根据 观察结论十.5，必有$\mathcal{P}(s, v) \neq \varnothing$，$\mathcal{L}(s, v) \neq \varnothing$。

因此，作为整数集的一个非空有限子集，$\mathcal{L}(s, v)$ 中必然存在一个唯一的最小元素$\delta(s, v)$，与之对应的任何一条路径，都是从 s 通往节点 v 的最短路径。 ❑

需要强调的是，观察结论十.15 虽然确保了最短路径的存在性，但这并不意味着最短路径必然是唯一的。实际上，在任何一对顶点之间，最短路径都有可能不是是唯一的，请读者自行构造这样实例。

■ 最短路径生成树

定义十.15 相对于同一起点 s，$V_r(G, s)$中所有顶点所对应的最短路径合起来将构成 $V_r(G, s)$的一棵生成树，称作最短路径生成树。



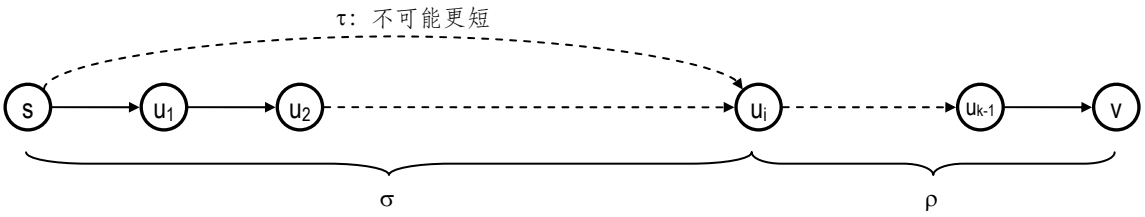
图十.21 有向带权图(a)中的所有最短路径，构成了一棵生成树(b)

需要指出的是，两点之间的最短路径可能不是唯一的（请读者自行给出实例）。倘若从 s 通往每一节点的最短路径都是唯一的，则最短路径生成树也必然唯一，此时我们将其记作 SPT(G, s)。

■ 单调性

观察结论十.16 若从顶点 s 到 v 的一条最短路径为 $\pi = (u_0 = s, u_1, u_2, \ldots, u_k = v)$，则对于任何 $0 \le i \le k$，$(s, u_1, u_2, \ldots, u_i)$也是从顶点 s 到 u_i 的一条最短路径。

〖证明〗



图十.22 最短路径的任一前缀也是最短路径

如 图十.22 所示，记$\sigma = (s, u_1, u_2, \ldots, u_i)$，$\rho = (u_i, u_{i+1}, \ldots, v)$。反证。σ不是从顶点s到$u_i$的最短路径。

既然u_i是从s可达的，故根据 观察结论十.15，从s到u_i的最短路径必然存在，不妨任取其中一条为τ，$|\tau| < |\sigma|$。

请注意，$\tau + \rho$也是从顶点 s 通往 v 的一条路径，而且其权重

$$|\tau| + |\rho| < |\sigma| + |\rho| = |\pi|$$

这就是说，π不可能是从顶点 s 到 v 的最短路径。矛盾。 ❑

考虑到“所有边的权重为正数”的假设，由上可得如下推论：

推论十.6 沿着任一最短路径π = (s, u_1, u_2, …, u_k)，从 s 到各顶点 u_i 的最短距离必然是严格递增的，i = 1, …, n。

10.8.4 最短路径序列

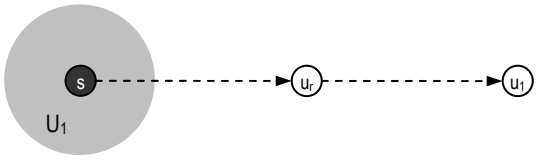
在给出最短路径的算法之前，有必要对最短路径的性质做进一步了解。

若有向带权图 G 中的所有顶点都是从顶点 s 可达的，则可以根据 s 到每个顶点的最短路径长度，将它们排成一个非降序列：{u_0, u_1, …, u_{n-1}}。显然，u_0 就是 s 本身——它到自己的最短距离为 0。但是，其它的顶点都有什么特点？如何找到它们呢？

■ u_1

观察结论十.17从 s 到 u_1 的最短路径由一条边(s, u_1)组成，δ(s, u_1) = |(s, u_1)|；而且，在从 s 发出的所有边中，边(s, u_1)最短。

〖证明〗



图十.23 从s通往u_1的最短路径

首先，在从s通往u_1的最短路径上，不可能还有其它的节点。否则，如 图十.23 所示，任取这样的一个节点u_r，r ≥ 2。根据 推论十.6，δ(s, u_r) < δ(s, u_1)，与u_1和u_r的定义矛盾。

其次，倘若从 s 发出的另一条边(s, u_r)比(s, u_1)更短，则同样也与 u_1 的定义矛盾。❑

以 图十.21 为例，若起点s为节点 0，u_1 就是节点 1，而(0, 1)是从节点 0 发出的三条边(0, 1)、(0, 2)和(0, 3)中的最短者。

基于 观察结论十.17，可以立即得到一个确定u_1的算法：

```
算法：DetermineU1
{
  逐一检查从s发出的每条边;
  找出其中的最短者（若有多条，任取其一），其终点即是u1;
}
```

算法十.10 确定u_1

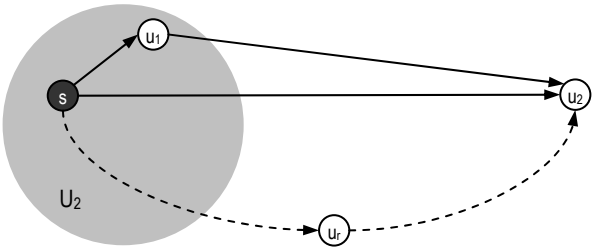
■ u_2

观察结论十.18从 s 到 u_2 的最短路径，只有两种可能：

① 由一条边(s, u_2)组成，$\delta(s, u_2) = |(s, u_2)|$；或者

② 由(s, u_1)和(u_1, u_2)组成，$\delta(s, u_2) = \delta(s, u_1) + |(u_1, u_2)|$。

〖证明〗



图十.24 从s通往u_2的最短路径

反证。如 图十.24 所示，假设在从s通往u_2的最短路径上，还有第三个节点u_r，$r \geq 3$。根据 推论十.6，$\delta(s, u_r) < \delta(s, u_2)$，与$u_2$和$u_r$的定义矛盾。 ❑

根据 观察结论十.18，可以立即得到一个确定u_2的算法：

```
算法: DetermineU2
{
  令集合U = {s, u1};
  对于每个节点v ∈ V\U
    令δ(s, v) = min{δ(s, u) + |(u, v)| | u ∈ U};

  找出δ(s, t) = min{δ(s, v) | v ∈ V\U};
  t就是u2; //若有多个，任取其一
}
```

算法十.11 确定u_2

■ u_k

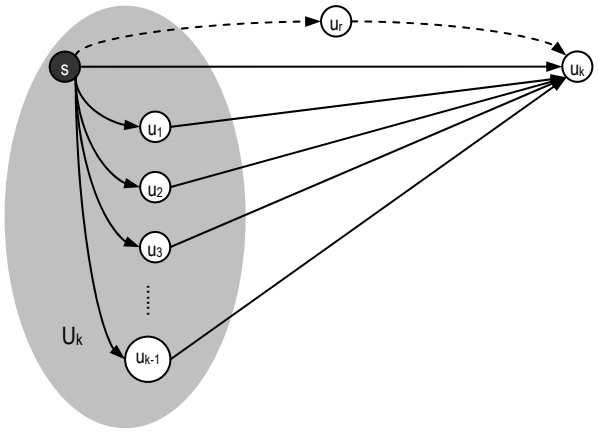
以上关于u_1和u_2的结论，可以推广至u_k，$1 \leq k < n$：

引理十.3

① 从 s 到u_k的最短路径，由从 s 到某一u_i的最短路径和(u_i, u_k)组成，$0 \leq i < k$；

② $\delta(s, u_k) = \delta(s, u_i) + |(u_i, u_k)|$。

〖证明〗



图十.25 从s通往u_k的最短路径

反证。如 图十.24 所示，假设在从s通往u_k的最短路径上，有某个节点u_r，$r \geq k+1$。根据 推论十.6，$\delta(s, u_r) < \delta(s, u_k)$，与$u_k$和$u_r$的定义矛盾。 ❑

以 图十.21 为例，若起点s为节点 0，则u_1、u_2和u3 分别为节点 1、2 和 3，它们对应的最短距离分别为$\delta(0, 1) = 9$、$\delta(0, 2) = 14$ 和$\delta(0, 3) = 15$。在所有形如

$$\delta(0, u_i) + |(u_i, v)|, \quad 0 \leq i \leq 3, v \in \{4, 5, 6, 7\}$$

的路径长度中，节点 6 对应的路径长度最短（14 + 18 = 32），故下一最近节点 $u_4 = 6$。

根据 引理十.3，可以立即得到一个确定u_k的算法：

```
算法：DetermineUk
{
  令集合Uk = {s = u0, u1, u2, …, uk-1};
  对于每个节点v ∈ V\Uk
    令δ(s, v) = min{δ(s, u) + |(u, v)| | u ∈ Uk};

  找出δ(s, t) = min{δ(s, v) | v ∈ V\Uk};
  t就是uk; //若有多个，任取其一
}
```

算法十.12 确定u_k

10.8.5 Dijkstra 算法

■ 最佳选取策略

将第 10.8.2 节介绍的最佳优先遍历策略与第 10.8.3 节介绍的确定u_k的算法结合起来，就得到了一个解决最短路径问题的算法，即著名的Dijkstra[㈠]算法。

这一算法为每个节点设置了一个域，用以记录从起点到它们的最短距离，从最佳优先遍历模板的角度看，也就是每个节点的距离指标。算法以迭代方式反复执行，依次找出u_0、u_1、u_2、…、u_{n-1}。具体地，在第k轮迭代中，若将此前已经找出的前k个最近节点组成集合$U_k = \{u_0, u_1, \ldots, u_{k-1}\}$，则只要按照 算法十.12 检查所有形如

$$\delta(s, u) + |(u, v)|, \quad u \in U_k, v \in V \backslash U_k$$

的距离，根据 引理十.3，其中的最小者就是第k个最近的节点u_k。不难看出，这一选取策略与最佳优先遍历算法中每次挑选下一访问节点的策略（代码十.12 中的bestVertex()方法）完全一致，因此可以直接套用最佳优先遍历的模板。

■ 距离指标的更新算法

与一般的最佳优先遍历一样，一旦确定了 u_k，还要进一步令 $U_{k+1} = U_k \cup \{u_k\}$，并更新 $V \backslash U_{k+1}$ 中各个节点的距离指标，以为下一轮迭代做好准备。

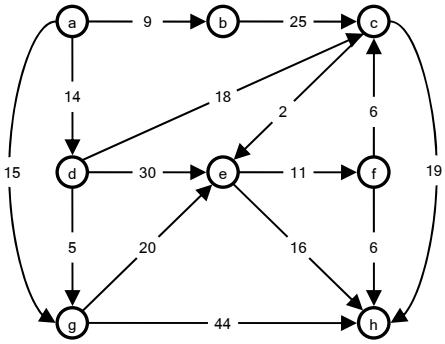
为此，对于 $V \backslash U_{k+1}$ 中的每个节点 v，我们只需将$\delta(s, u_k) + |(u_k, v)|$与它此前的距离指标$\delta(s, v)$作比较，并用二者中的小者作为节点 v 新的距离指标。

初始情况下，可以令 $U_0 = \{s = u_0\}$。

■ 最短路径生成树

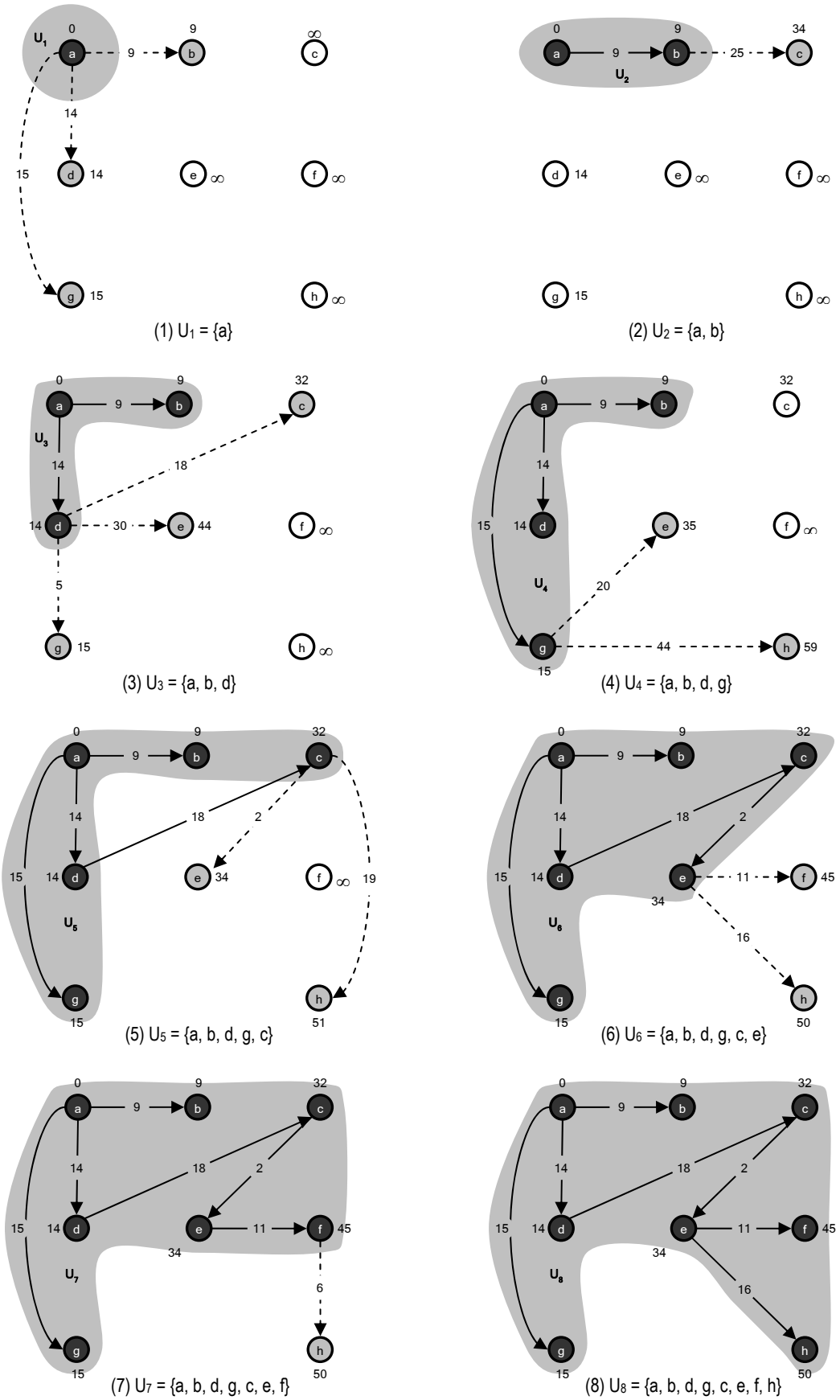
若下一最近节点 u_k 对应的最短距离为$\delta(s, u_i) + |(u_i, u_k)|$，$0 \le i < k$，则边$(u_i, u_k)$就是最短路径生成树中的一条边。为此，我们也可以按照最佳优先遍历模板中的方法，将 u_i 置为 u_k 的父节点。

■ 实例



(0) 有向图 G

㈠ 193 0/05/11 – 2002/08/06，杰出的计算机科学家，1972 年图灵奖得主。



图十.26 Di jkstra算法实例

黑色节点组成集合U_k,U_k中每次引入新节点后需要更新距离指标的节点标记为灰色

图十.26 给出了Dijkstra算法一次完整的执行过程。有向带权图G如 图十.26(0)所示，其中的节点a作为起点，所有节点的距离指标都被初始化为无穷大。图十.26(1)中，距离指标最优（最短）的节点u_0 = a 首先被选出，随后其相邻节点b、d和g的距离指标也相应地得到更新。图十.26(2)中，距离指标最优的节点u_1 = b 被选出，其相邻节点c的距离指标得到更新。图十.26(3)中，距离指标最优的节点u_2 = d 被选出，其相邻节点c、e和g的距离指标得到更新。图十.26(4)中，距离指标最优的节点u_3 = g 被选出，其相邻节点e和h的距离指标得到更新。图十.26(5)中，距离指标最优的节点u_4 = c 被选出，其相邻节点e和h的距离指标得到更新。图十.26(6)中，距离指标最优的u_5 = e被选出，其相邻节点f和h的距离指标得到更新。图十.26(7)中，距离指标最优的节点u_6 = f被选出，其相邻节点h的距离指标得到更新。最后，在 图十.26(8)中，距离指标最优的节点u_7 = h被选出。至此，我们得到了每个节点的最短距离、它们对应的最短路径以及整体的最短路径生成树。

■ Jav a 实现

基于最佳优先遍历模板，可以很简捷地实现上述Dijkstra算法。如 代码十.13 所示，这里只需在抽象类BestFS的基础上，实现其中的updateDistanceAfter()方法，该方法的作用是，在每次将新节点引入集合U_k之后，相应地更新$V\backslash U_{k+1}$ 中各节点的距离指标。

```
/*
 * （有向）带权图的单源点最短路径算法
 */

package dsa;

public class BestFSDijkstra extends BestFS {
//构造方法
  public BestFSDijkstra(Graph g) { super(g); }

//更新尚未访问的顶点到源点的最短距离
  protected void updateDistanceAfter(Vertex v) {
    for (Iterator it = v.outEdges(); it.hasNext();) {//检查与顶点v
      Edge e = (Edge)it.getNext();//通过边e = (v, w)
      Vertex w = (Vertex)e.getVPosInV(1).getElem();//相联的每一顶点w
      int weight = ((Integer)e.getInfo()).intValue();//根据边(v, w)的权重
      if (w.getDistance() > v.getDistance() + weight) {//取原距离与新距离中的小者
        w.setDistance(v.getDistance() + weight);
        w.setBFSParent(v);
      }
    }
  }
}
```

代码十.13 基于最佳优先遍历的Dijkstra算法

10.8.6 最小生成树

无论是从算法理论还是实际应用的角度看，最小生成树问题的重要性都很突出。就前一方面而言，最小生成树是众多算法的基础，同时也可以用来近似求解很多 NP 难题（比如著名的 TSP 问题）。就后一方面而言，从网络结构设计、集成电路布线到刺绣路径规划等众多的应用领域，最小生成树算法都是不可或缺的。

实际应用中的这类典型问题可以归纳为：对于给定的一组对象，需要在最低的成本下，将它们联接成为一个具有连通性的系统。如果利用图来描述这一问题，也就是要构建一个连通图。

显然，为了保证成本最低，首先这幅连通图包含的边就应该尽可能少——也就是说，它应该是一棵树。正如我们马上就要看到的，如果用其中各边权重的总和当作成本，那么成本最小的树必然存在（尽管不见得唯一），我们称之为输入节点的一棵“最小生成树”。

我们将通过无向图来定义和描述这一问题，通过对其性质的刻画与分析得出一般性的规律，并进而引出构造最小生成树的Prim-Jarnik算法。最后，通过将无向图转化为有向图，我们同样可以基于第 10.8.2 节的最佳优先遍历模板（代码十.12）实现这一算法。

■ 问题及定义

给定连通的无向带权图 G = (V, E)，其中各边权重均为正数。

设T = (V, E')为G的一棵生成树（定义十.10）。

定义十.16 E' 中各边权重的总和 $\sum_{e \in E'} w(e)$，称作生成树 T 的代价或成本。

我们来考察G的所有生成树。根据 观察结论十.8 的第②部分，G的每一棵生成树都恰好由|V|-1条边组成，因此G的生成树总数必然有限，不会超过$\binom{|E|}{|V|-1}$棵。另外，既然各边权重均为正数，故每棵生成树的成本也必为正数。由此可知，其中成本最低的生成树必然存在。

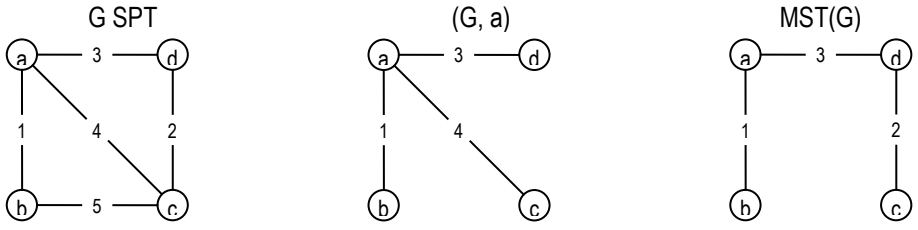
定义十.17 无向带权图 G 的所有生成树中，成本最低者称作 G 的最小生成树。

需要指出的是，尽管最小生成树必然存在，但不见得唯一（请读者自行给出这样的例子）。正因为此，我们才称之为“minimal spanning tree”，而不是“minimum spanning treee”。当图 G 的最小生成树唯一时，我们将它记作 MST(G)。

■ MST ≠ SPT

在第 10.8.3 节，我们曾经引入过最短路径生成树的概念（定义十.15）。那么，最短路径生成树与此处引入的最小生成树是否同一概念呢？

如 图十.27 所示的实例给出了否定的回答。



(a) 图 (b) 最短路径生成树 (c) 最小生成树

图十.27 即使仅就拓扑结构而言，同一幅图的最短路径生成树与最小生成树也不一定相同

■ 最短桥

考察节点集的任一子集 $U \subseteq V$。

定义十.18 E 中形如(a, b), $a \in U$, $b \in V \backslash U$ 的每一条边，都称作横跨于 U 和 V\U 之间的一座桥。

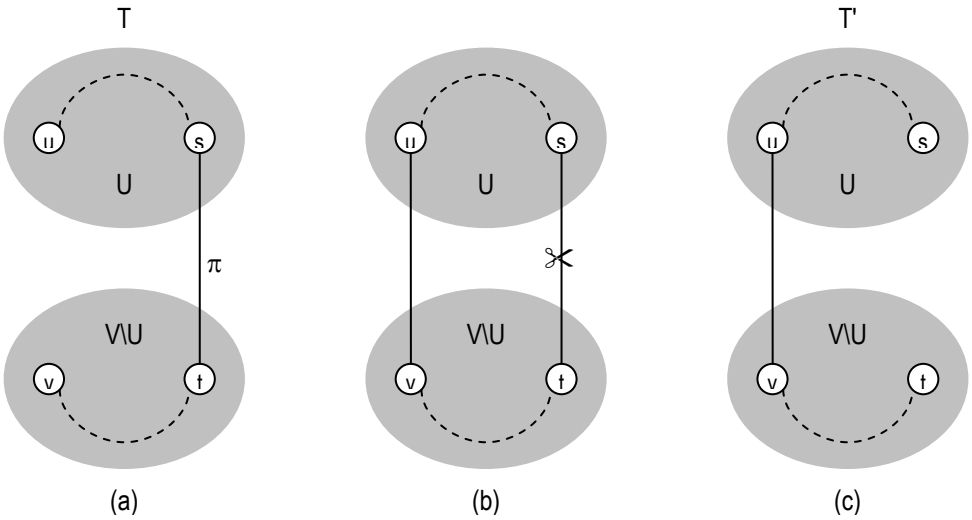
根据树的连通性，最小生成树 MST(G)中必然含有横跨于 U 和 V\U 之间的（至少）一座桥。

反过来，不难理解，横跨于 U 和 V\U 之间的桥通常不止一条。那么，其中的哪些桥将被 MST(G)采用呢？以下引理回答了这个问题：

引理十.4 横跨于 U 和 V\U 之间的所有桥中，权重最小的桥必然会被至少一棵最小生成树采用。

〖证明〗

反证。假设(u, v)为横跨于 U 和 V\U 之间的最短桥，$u \in U$，$v \in V \backslash U$，而且(u, v)没有被任何最小生成树采用。



(a) (b) (c)

图十.28 最短桥必被某一棵最小生成树采用

如图十.28(a)所示，考察图G的任意一棵最小生成树T。

根据树的连通性（观察结论四.2），在u和v之间必然存在（唯一的）一条通路π。不难看出，路径π上必然存在（至少）一座横跨于U和V\U之间的桥$(s, t) \neq (u, v)$，$s \in U$，$t \in V \backslash U$。

现在，如 图十.28(b)所示，将桥(u, v) 加入T中，于是T中将出现唯一的一条环路——即由π和(u, v) 组成的环路。只要将桥(s, t) 从T中删去，就可以消除这一环路，同时也不至于破坏T的连通性——也就是说，T' = T∪{u, v})\{(s, t)}依然是G的一棵生成树(如 图十.28(c) 所示)。

对比 T 和 T'可以看出，二者的成本之差等于 weight(s, t) – weight(u, v)。根据桥(u, v) 的最短性，T'的成本不会高于 T，因此 T'也是图 G 的一棵最小生成树——这与假设矛盾。

证毕。 ❑

10.8.7 Prim-Jarnik 算法

■ 最佳选取策略

根据 引理十.4，可以立即得到构造最小生成树的Prim-Jarnik算法[十]。

这一算法维护节点集V的一个子集U，以及限制子图$G|_U$（定义十.3）的一棵最小生成树T。

整个算法由|V|轮迭代组成。初始时，从 V 中任意挑选一个节点构成集合 U，T 为空树。每一次迭代中，在横跨于 U 和 V\U 之间的所有桥中，挑选出最短者(u, v)，u∈U，v∈V\U，然后将 v 加入 U 中，将(u, v)加入 T 中。

这一过程可以描述为 算法十.13：

```
算法：Prim-Jarnik(G)
输入：连通的无向带权图G = (V, E)
输出：图G的最小生成树
{
  任取节点x∈V，令U = {x};
  令T = ∅;
  while (U ≠ V) {
    在横跨于U和V\U之间的所有桥中，找出最短者(u, v)，u∈U，v∈V\U; //若有多条，任取其一
    令U = U ∪ {v};
    令T = T ∪ {(u, v)};
  }
}
```

算法十.13 Prim-J arnik算法

■ 正确性

根据 引理十.4，由数学归纳法可以得出如下推论：

推论十.7 Prim-Jarnik 算法中，每一轮迭代之后，T 都是限制子图 $G|_U$ 的一棵最小生成树。

[十] 这一算法由 Jarnik (1930)和 Prim (1956)独立发明，由此得名。

当该算法结束时，集合 U = V，限制子图 $G|_U$ 就是 G 本身，于是 T 就是图 G 的一棵最小生成树。

定理十.10 经过|V|轮迭代，Prim-Jarnik 算法可以构造出连通无向带权图 G = (V, E)的一棵最小生成树。

■ 距离指标的更新

可以再次利用最佳优先遍历模板（第 10.8.2 节）来实现Prim-Jarnik算法。为此，我们只需为V\U中的每个节点v设置一个距离指标distance(v)，该指标等于节点v到集合U的最短距离，即

$$distance(v) = \min \{ weight(u, v) \mid u \in U \}$$

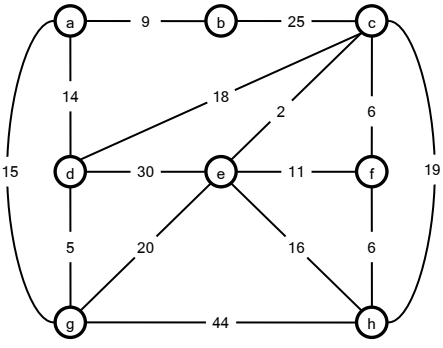
在经过一次迭代、挑选出节点 v 并将其加入子集 U 之后，我们需要立即更新 V\U 中各节点的这一距离指标。直接根据 distance()的定义，需要对每个节点 $w \in V\backslash U$，扫描 U 中的所有节点，以得到新的 distance(w)。按照这一方法，需要经过 $O(|V|)$的时间才能更新单个节点的距离指标，经过 $O(|V|^2)$的时间才能更新所有节点的距离指标，于是|V|轮迭代总共需要 $O(|V|^3)$的时间。显然，这一效率无法令人满意。

幸运的是，完全可以在更短的时间内完成距离指标的更新。实际上，由于每经过一次迭代，U 中仅仅会引入一个新的节点 v，而且原有的节点不变，故为了更新某一节点 $w \in V\backslash U$ 的距离指标，只需将其原先的距离指标与 weight(v, w)做一比较，并保留其中的更小者——采用这一方法，只需常数时间。

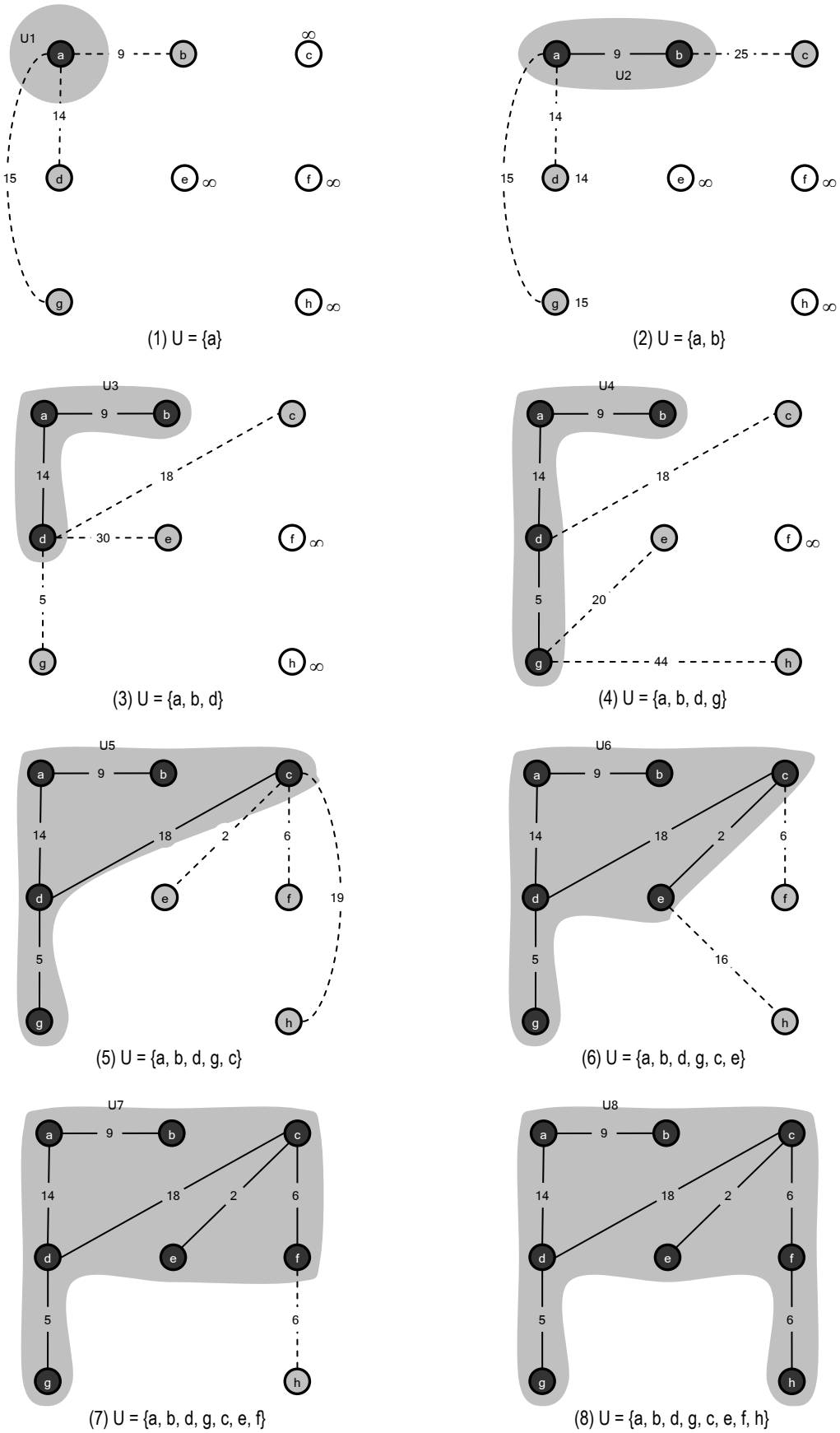
综合以上结论，可以得到如下定理：

定理十.11采用 Prim-Jarnik 算法，可以在 $O(|V|^2)$的时间内构造出连通无向带权图 G = (V, E)的一棵最小生成树。

■ 实例



(0)



(1) U = {a}　(2) U = {a, b}

(3) U = {a, b, d}　(4) U = {a, b, d, g}

(5) U = {a, b, d, g, c}　(6) U = {a, b, d, g, c, e}

(7) U = {a, b, d, g, c, e, f}　(8) U = {a, b, d, g, c, e, f, h}

图十.29 Prim-J arnik算法实例

黑色节点来自集合U，U中每次引入新节点后需要更新距离指标的节点标记为灰色

图十.29 给出了Prim-Jarnik算法一次完整的执行过程。无向带权图G如 图十.29(0)所示，首先集合U初始化为空集，所有节点的距离指标都被初始化为无穷大。图十.29(1)中，节点a首先被选出并加入集合U，随后其相邻节点b、d和g的距离指标也相应地得到更新。图十.29(2)中，距离指标最优（小）的节点b被选出，其相邻节点c的距离指标得到更新。图十.29(3)中，距离指标最优的节点d被选出，其相邻节点c、e和g的距离指标得到更新。图十.29(4)中，距离指标最优的节点g被选出，其相邻节点e和h的距离指标得到更新。图十.29(5)中，距离指标最优的节点c被选出，其相邻节点e、f和h的距离指标得到更新。图十.29(6)中，距离指标最优的节点e被选出，其相邻节点f和h的距离指标得到更新。图十.29(7)中，距离指标最优的节点f被选出，其相邻节点h的距离指标得到更新。最后，在 图十.29(8)中，距离指标最优的节点h被选出。至此，我们就得到了图G的一棵最小生成树。

请将 图十.29 与 图十.26 进行比照，以体会这两种算法的不同，以及二者之间的共性。

■ Jav a 实现

基于最佳优先遍历模板（代码十.12），可以很简捷地实现上述Prim-Jarnik算法。

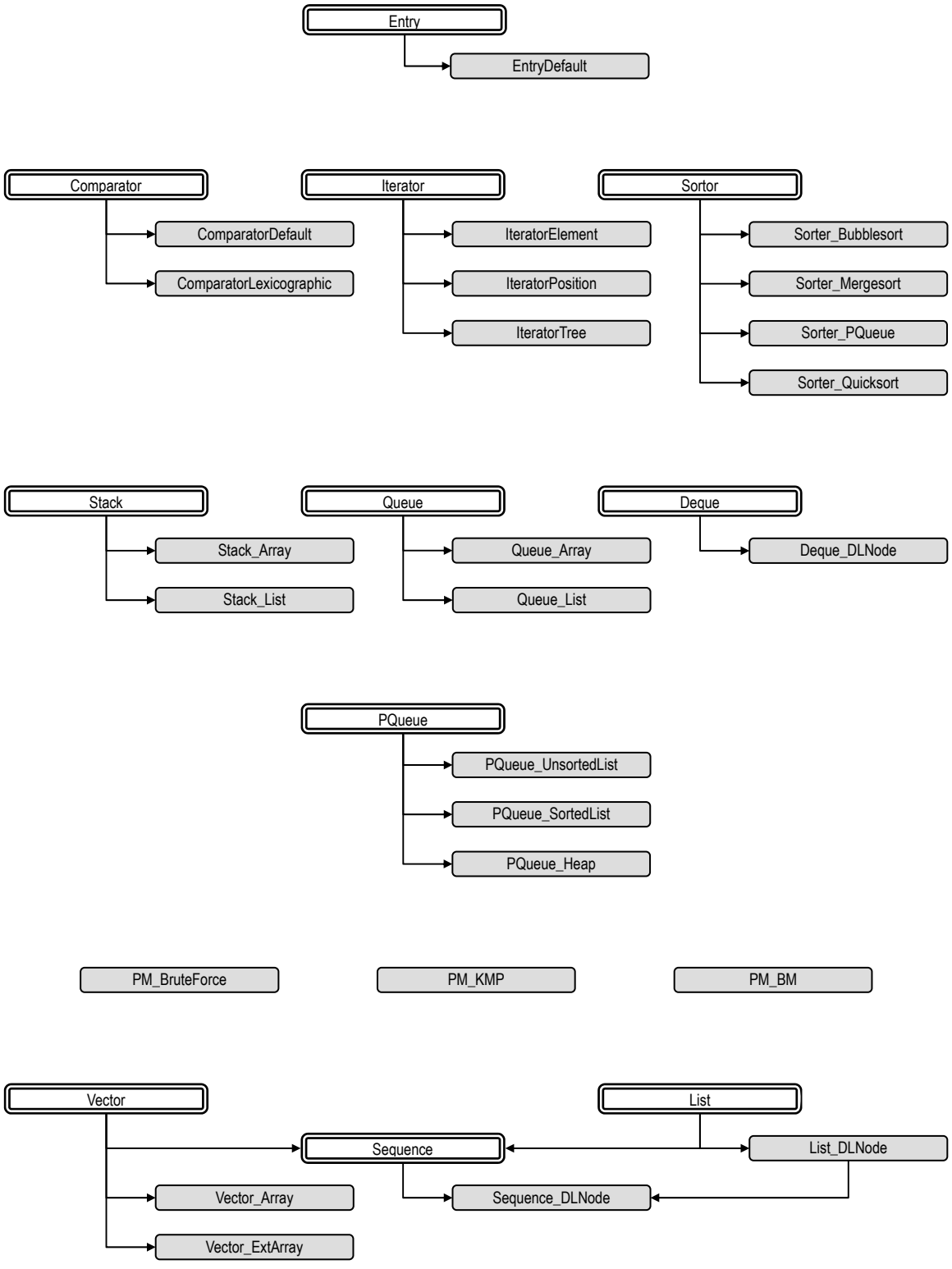
如 代码十.14 所示，这里只需在抽象类BestFS的基础上，实现具体的updateDistanceAfter()方法，该方法的作用，是在每次将新节点引入集合U之后，相应地更新V\U中各节点的距离指标。

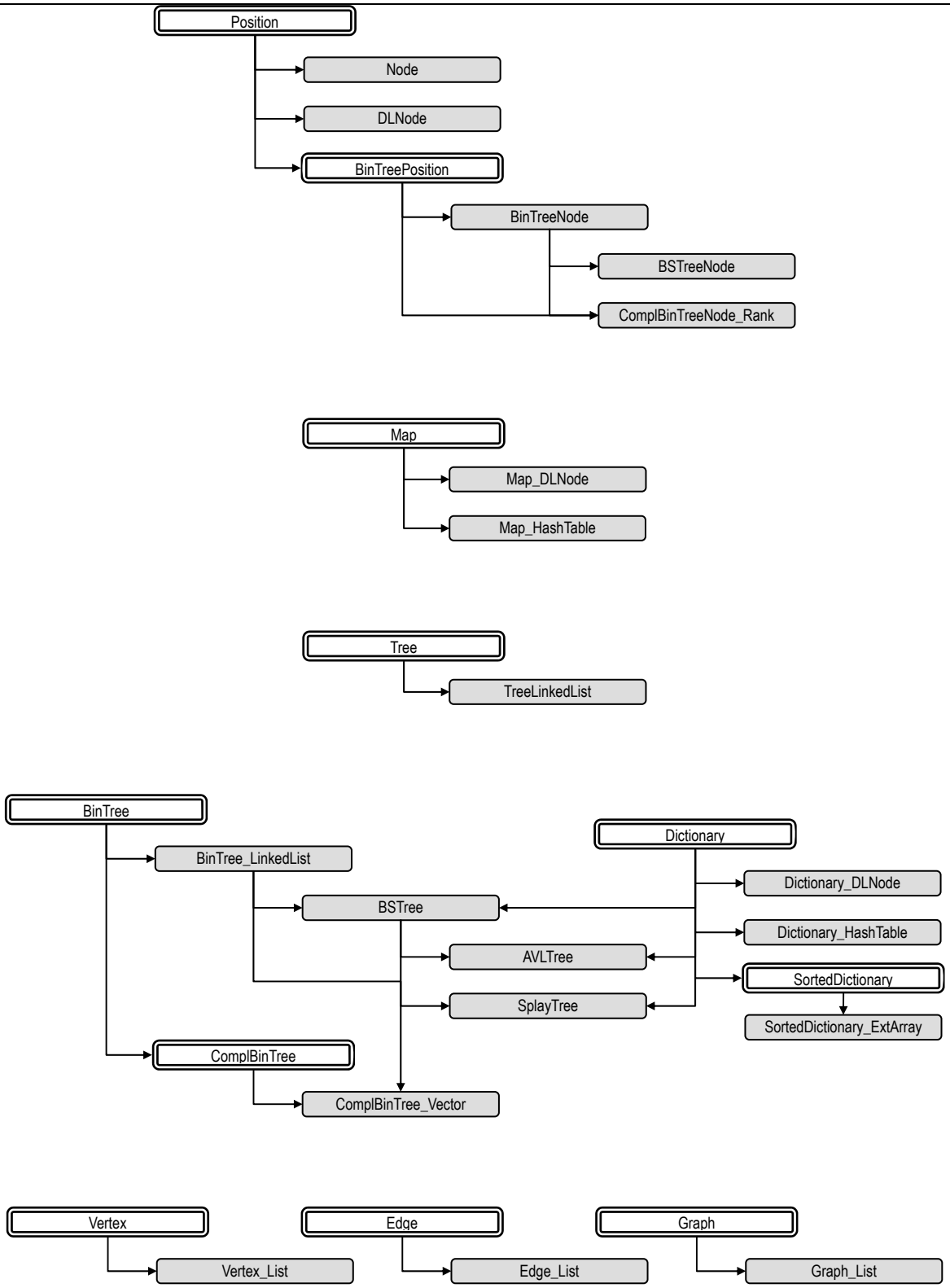
```
/*
 * （有向）带权图最小生成树的Prim-Jarnik算法
 */

package dsa;

public class BestFSPrim extends BestFS {
//构造方法
  public BestFSPrim(Graph g) { super(g); }

//更新尚未访问的顶点到已访问集的最短距离
  protected void updateDistanceAfter(Vertex v) {
    for (Iterator it = v.outEdges(); it.hasNext();) {//检查与顶点v
      Edge e = (Edge)it.getNext();//通过边e = (v, w)
      Vertex w = (Vertex)e.getVPosInV(1).getElem();//相联的每一顶点w
      int weight = ((Integer)e.getInfo()).intValue();//根据边(v, w)的权重
      if (w.getDistance() > weight) {//取原距离与新距离中的小者
        w.setDistance(weight);
        w.setBFSParent(v);
      }
    }
  }
}
```

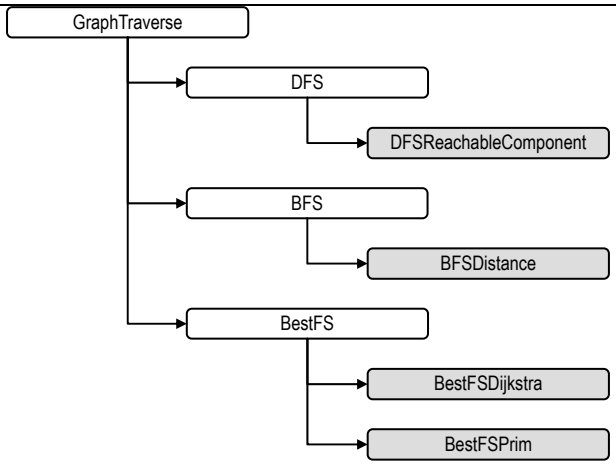
代码十.14 基于最佳优先遍历的Prim-Jarnik算法

附录

DSA 类关系图


Entry
EntryDefault
Comparator
ComparatorDefault
ComparatorLexicographic
Iterator
IteratorElement
IteratorPosition
IteratorTree
Sortor
Sorter_Bubblesort
Sorter_Mergesort
Sorter_PQueue
Sorter_Quicksort
Stack
Stack_Array
Stack_List
Queue
Queue_Array
Queue_List
Deque
Deque_DLNode
PQueue
PQueue_UnsortedList
PQueue_SortedList
PQueue_Heap
PM_BruteForce
PM_KMP
PM_BM
Vector
Vector_Array
Vector_ExtArray
Sequence
Sequence_DLNode
List
List_DLNode


Position
Node
DLNode
BinTreePosition
BinTreeNode
BSTreeNode
ComplBinTreeNode_Rank
Map
Map_DLNode
Map_HashTable
Tree
TreeLinkedList
BinTree
BinTree_LinkedList
BSTree
AVLTree
SplayTree
ComplBinTree
ComplBinTree_Vector
Dictionary
Dictionary_DLNode
Dictionary_HashTable
SortedDictionary
SortedDictionary_ExtArray
Vertex
Vertex_List
Edge
Edge_List
Graph
Graph_List


GraphTraverse
DFS
DFSReachableComponent
BFS
BFSDistance
BestFS
BestFSDijkstra
BestFSPrim

插图索引

表格索引

算法索引

代码索引

定义索引

观察结论索引

推论索引

引理索引

定理索引

关键词索引

K

P

Y

L

H

X

H

Y

SH

W

H

W

C

X

K

T

C

L

ZH

X

D

Q

D

Y

ZH

X

ZH

D

P

SH

J

C H

S

Z

B

M

L