



华章教育

面向CS2013计算机专业规划教材



数据结构与算法

Python语言描述

裘宗燕 著
北京大学

D *Data Structures
and Algorithms in Python*



机械工业出版社
China Machine Press



为采用的教师
提供教辅资源

下载网址: <http://www.hzbook.com>
教学支持: 010-68353079, 88378995

21世纪以来, Python已经发展成为世界上最受欢迎的编程语言之一, 使用非常广泛。由于其各方面的优点, Python正在被世界上越来越多的大学用作第一门程序设计课程的语言, 更多学校把它作为后续或选修课程的内容。国内也开始出现这种情况。作者从几年前开始基于Python语言讲授数据课程, 本书基于作者的教学经验和体会编写而成。

本书结合抽象数据类型结构的思想, 基于Python的面向对象机制, 阐述了各种基本数据结构的想法、性质、问题和实现, 讨论一些相关算法的设计、实现和特性。书中还结合研究了一些数据结构的应用案例。

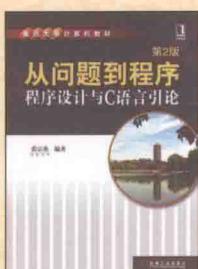
本书加强了一些目前程序设计实践领域特别关注的内容, 包括程序和数据结构设计中的安全性问题、正则表达式的概念和使用等。书中提供了大量编程练习题, 特别关注数据结构的设计和实现技术, 以及实际应用中的各方面问题。

本书要求学习者已有基本Python程序设计的知识和经验, 可以作为基于Python的计算机基础课程中的数据结构课程教材, 也可以作为学习了Python语言基本内容之后的一本面向对象等高级编程技术的进阶读物。

作者介绍



裴宗燕, 北京大学数学学院信息科学系教授。长期从事计算机软件与理论、程序设计语言和符号计算方面的研究与教学工作。已出版过多部著作和译著, 包括《程序设计语言基础》(译著, 北京大学出版社, 1990), 《Mathematics数学软件系统的应用与程序设计》(编著, 北京大学出版社, 1994), 《C++程序设计语言(特别版)》(译著, 机械工业出版社, 2002), 《C++语言的设计和演化》(译著, 机械工业出版社, 2002), 《程序设计语言——概念和结构》(合译, 机械工业出版社, 2002), 《从问题到程序——程序设计与C语言引论》(编著, 机械工业出版社, 2005年第1版, 2011年第2版)等。



从问题到程序——程序设计与C语言引论 第2版
书号: 978-7-111-33715-7
定价: 39.00元



上架指导: 计算机/数据结构

ISBN 978-7-111-52118-1



9 787111 521181 >

定价: 45.00元

投稿热线: (010) 88379604

客服热线: (010) 88378991 88361066

购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com

网上购书: www.china-pub.com

数字阅读: www.hzmedia.com.cn



数据结构与算法

Python语言描述

裴宗燕
北京大学 著

D *ata Structures
and Algorithms in Python*



图书在版编目 (CIP) 数据

数据结构与算法：Python 语言描述 / 裴宗燕著 . —北京：机械工业出版社，2015.12
(面向 CS2013 计算机专业规划教材)

ISBN 978-7-111-52118-1

I. 数… II. 裴… III. ① 数据结构—高等学校—教材 ② 算法分析—高等学校—教材 ③ 软件工具—程序设计—高等学校—教材 IV. ① TP311.12 ② TP311.56

中国版本图书馆 CIP 数据核字 (2015) 第 271070 号

Python 是目前国际上流行的用于教授第一门程序设计课程的语言，国内高校也开始使用。本书是结合国内数据结构课程现状，采用 Python 作为工作语言，全新编撰的一本数据结构教程。书中结合抽象数据类型结构的思想，基于 Python 的面向对象机制，阐述各种基本数据结构的性质、问题和实现，讨论一些相关算法的设计、实现和特性。书中还结合研究了一些数据结构的应用案例。

本书要求学习者已有基本 Python 程序设计的知识和经验，可以作为基于 Python 的计算机基础课程中的数据结构课程教材，也可以作为学习 Python 语言基本内容之后的一本面向对象等高级编程技术的进阶读物。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：余 洁

责任校对：董纪丽

印 刷：三河市宏图印务有限公司

版 次：2016 年 1 月第 1 版第 1 次印刷

开 本：185mm×260mm 1/16

印 张：22

书 号：ISBN 978-7-111-52118-1

定 价：45.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

前　　言

本书基于作者在北京大学用 Python 讲授相应课程的工作，用 Python 作为工作语言讨论数据结构和算法的基本问题，其撰写主要有下面几方面考虑：

- 作为以 Python 为第一门计算机课程之后相应的数据结构课程的教材。
- 结合数据结构和算法，讨论 Python 中重要数据类型的实现情况和性质，帮助读者理解 Python 语言程序，理解如何写出高效的 Python 程序。
- 展示 Python 的面向对象技术可以怎样运用。书中构造了一批相互关联的数据结构类，前面定义的类被反复应用在后续章节的数据结构和算法中。

基于这些情况，本书不但可以作为数据结构课程的教材，也可以作为学习 Python 语言编程技术的后续读物（假设读者已经有了 Python 编程的基本知识）。

由于 Python 语言的一些优点，近年来，国外已经有不少大学采用它作为第一门计算机科学技术课程的教学语言，包括许多一流大学。国内院校也可能参考这种趋势，出现这种变化。作者在北京大学数学学院开设了基于 Python 语言的程序设计和数据结构课程，通过亲身实践，发现用 Python 讲授这两门课程也是一种很好的安排。

用 Python 学习数据结构，最大的优点就是可以看到复杂的数据结构可以怎样一步步地从基本的语言机制构造起来。在一个章节里定义的数据结构，经常可以在后续章节的算法和数据结构中直接使用，如果不适用，常常可以通过简单的类派生来调整。还可以非常方便地用在各种习题和练习里，或用于解决实际问题。学生可以看到，书中的（或他们自己写的）代码不是玩具，而是切实有用的软件构件。在基于本书的课程中，很容易安排一些有一定规模的面向实际应用的开发课题，使学生得到更好的实际锻炼。

书中标 * 的节或小节作为选讲内容，或留给学生自己阅读。

本书的成型也得益于作者多年讲授基于 C 语言的数据结构课程的经验，张乃孝老师的《算法和数据结构》是作者一直使用的教材，本书在编写中也参考了该书的一些体例。此外，北京大学数学学院 2013 级的同学在学习中提出了许多很好的问题，参加课程辅导工作的刘海洋、胡婷婷、张可和陈晨也提供了很多帮助，在此表示特别的感谢。

裘宗燕
2015 年 8 月于北京

目 录

前言	
第 1 章 绪论	1
1.1 计算机问题求解	1
1.1.1 程序开发过程	1
1.1.2 一个简单例子	3
1.2 问题求解：交叉路口的红绿灯安排	4
1.2.1 问题分析和严格化	5
1.2.2 图的顶点分组和算法	6
1.2.3 算法的精化和 Python 描述	7
1.2.4 讨论	8
1.3 算法和算法分析	10
1.3.1 问题、问题实例和算法	10
1.3.2 算法的代价及其度量	14
1.3.3 算法分析	19
1.3.4 Python 程序的计算代价（复杂度）	21
1.4 数据结构	23
1.4.1 数据结构及其分类	24
1.4.2 计算机内存对象表示	26
1.4.3 Python 对象和数据结构	30
练习	32
第 2 章 抽象数据类型和 Python 类	34
2.1 抽象数据类型	34
2.1.1 数据类型和数据构造	34
2.1.2 抽象数据类型的概念	36
2.1.3 抽象数据类型的描述	37
2.2 Python 的类	39
2.2.1 有理数类	39
2.2.2 类定义进阶	40
2.2.3 本书采用的 ADT 描述形式	43
2.3 类的定义和使用	44
2.3.1 类的基本定义和使用	44
2.3.2 实例对象：初始化和使用	45
2.3.3 几点说明	47
2.3.4 继承	49
2.4 Python 异常	53
2.4.1 异常类和自定义异常	53
2.4.2 异常的传播和捕捉	54
2.4.3 内置的标准异常类	54
2.5 类定义实例：学校人事管理系统中的类	55
2.5.1 问题分析和设计	56
2.5.2 人事记录类的实现	57
2.5.3 讨论	62
本章总结	63
练习	64
第 3 章 线性表	66
3.1 线性表的概念和表抽象数据类型	66
3.1.1 表的概念和性质	66
3.1.2 表抽象数据类型	67
3.1.3 线性表的实现：基本考虑	69
3.2 顺序表的实现	69
3.2.1 基本实现方式	69
3.2.2 顺序表基本操作的实现	71
3.2.3 顺序表的结构	74
3.2.4 Python 的 list	76
3.2.5 顺序表的简单总结	78
3.3 链接表	79
3.3.1 线性表的基本需要和链接表	79
3.3.2 单链表	79
3.3.3 单链表类的实现	84
3.4 链表的变形和操作	88
3.4.1 单链表的简单变形	88
3.4.2 循环单链表	91
3.4.3 双链表	92
3.4.4 两个链表操作	95
3.4.5 不同链表的简单总结	98
3.5 表的应用	99
3.5.1 Josephus 问题和基于“数组”概念的解法	99
3.5.2 基于顺序表的解	100

3.5.3 基于循环单链表的解	101	5.4.1 队列抽象数据类型	155
本章总结.....	102	5.4.2 队列的链接表实现	155
练习.....	103	5.4.3 队列的顺序表实现	156
第 4 章 字符串	107	5.4.4 队列的 list 实现.....	158
4.1 字符集、字符串和字符串操作.....	107	5.4.5 队列的应用	160
4.1.1 字符串的相关概念	107	5.5 迷宫求解和状态空间搜索.....	162
4.1.2 字符串抽象数据类型	109	5.5.1 迷宫求解：分析和设计	162
4.2 字符串的实现.....	109	5.5.2 求解迷宫的算法	164
4.2.1 基本实现问题和技术	109	5.5.3 迷宫问题和搜索	167
4.2.2 实际语言里的字符串	110	5.6 几点补充.....	171
4.2.3 Python 的字符串	111	5.6.1 几种与栈或队列相关的结构	171
4.3 字符串匹配（子串查找）.....	112	5.6.2 几个问题的讨论	172
4.3.1 字符串匹配	112	本章总结.....	173
4.3.2 串匹配和朴素匹配算法	113	练习.....	173
4.3.3 无回溯串匹配算法（KMP 算法）.....	115	第 6 章 二叉树和树.....	176
4.4 字符串匹配问题.....	119	6.1 二叉树：概念和性质	176
4.4.1 串匹配 / 搜索的不同需要	120	6.1.1 概念和性质	177
4.4.2 一种简化的正则表达式	122	6.1.2 抽象数据类型	181
4.5 Python 正则表达式.....	123	6.1.3 遍历二叉树	181
4.5.1 概况	124	6.2 二叉树的 list 实现.....	183
4.5.2 基本情况	124	6.2.1 设计和实现	183
4.5.3 主要操作	125	6.2.2 二叉树的简单应用：表达式树	185
4.5.4 正则表达式的构造	126	6.3 优先队列.....	188
4.5.5 正则表达式的使用	132	6.3.1 概念	188
本章总结.....	132	6.3.2 基于线性表的实现	189
练习.....	133	6.3.3 树形结构和堆	191
第 5 章 栈和队列	135	6.3.4 优先队列的堆实现	192
5.1 概述.....	135	6.3.5 堆的应用：堆排序	195
5.1.1 栈、队列和数据使用顺序	135	6.4 应用：离散事件模拟.....	196
5.1.2 应用环境	136	6.4.1 通用的模拟框架	197
5.2 栈：概念和实现.....	136	6.4.2 海关检查站模拟系统	198
5.2.1 栈抽象数据类型	137	6.5 二叉树的类实现.....	202
5.2.2 栈的顺序表实现	137	6.5.1 二叉树结点类	203
5.2.3 栈的链接表实现	139	6.5.2 遍历算法	204
5.3 栈的应用.....	140	6.5.3 二叉树类	208
5.3.1 简单应用：括号匹配问题	140	6.6 哈夫曼树.....	209
5.3.2 表达式的表示、计算和变换	142	6.6.1 哈夫曼树和哈夫曼算法	209
5.3.3 栈与递归	149	6.6.2 哈夫曼算法的实现	210
5.4 队列.....	155	6.6.3 哈夫曼编码	211

6.7 树和树林.....	212	第 8 章 字典和集合.....	265
6.7.1 实例和表示	213	8.1 数据存储、检索和字典.....	265
6.7.2 定义和相关概念	213	8.1.1 数据存储和检索	265
6.7.3 抽象数据类型和操作	215	8.1.2 字典实现的问题	267
6.7.4 树的实现	216	8.2 字典线性表实现.....	269
6.7.5 树的 Python 实现	218	8.2.1 基本实现	269
本章总结.....	220	8.2.2 有序线性表和二分法检索	270
练习.....	220	8.2.3 字典线性表总结	272
第 7 章 图	224	8.3 散列和散列表.....	273
7.1 概念、性质和实现.....	224	8.3.1 散列的思想和应用	273
7.1.1 定义和图示	224	8.3.2 散列函数	275
7.1.2 图的一些概念和性质	225	8.3.3 冲突的内消解：开地址技术	277
7.1.3 图抽象数据类型	227	8.3.4 外消解技术	280
7.1.4 图的表示和实现	228	8.3.5 散列表的性质	280
7.2 图结构的 Python 实现	231	8.4 集合.....	282
7.2.1 邻接矩阵实现	231	8.4.1 集合的概念、运算和抽象数据 类型	282
7.2.2 压缩的邻接矩阵（邻接表）实现.....	233	8.4.2 集合的实现	283
7.2.3 小结	235	8.4.3 特殊实现技术：位向量实现	285
7.3 基本图算法.....	235	8.5 Python 的标准字典类 dict 和 set.....	286
7.3.1 图的遍历	236	8.6 二叉排序树和字典.....	287
7.3.2 生成树	238	8.6.1 二叉排序树	288
7.4 最小生成树.....	240	8.6.2 最佳二叉排序树	295
7.4.1 最小生成树问题	240	8.6.3 一般情况的最佳二叉排序树	297
7.4.2 Kruskal 算法	240	8.7 平衡二叉树.....	302
7.4.3 Prim 算法	243	8.7.1 定义和性质	302
*7.4.4 Prim 算法的改进.....	246	8.7.2 AVL 树类	303
7.4.5 最小生成树问题	247	8.7.3 插入操作	304
7.5 最短路径.....	248	8.7.4 相关问题	310
7.5.1 最短路径问题	248	8.8 动态多分支排序树.....	311
7.5.2 求解单源点最短路径的 Dijkstra 算法	248	8.8.1 多分支排序树	311
7.5.3 求解任意顶点间最短路径的 Floyd 算法	252	8.8.2 B 树	312
7.6 AOV/AOE 网及其算法.....	255	8.8.3 B+ 树	314
7.6.1 AOV 网、拓扑排序和拓扑序列	255	本章总结.....	315
7.6.2 拓扑排序算法	257	练习.....	316
7.6.3 AOE 网和关键路径	258	第 9 章 排序.....	319
7.6.4 关键路径算法	259	9.1 问题和性质.....	319
本章总结.....	261	9.1.1 问题定义	319
练习.....	262	9.1.2 排序算法	320

9.2 简单排序算法.....	323	9.4.2 归并算法的设计问题	333
9.2.1 插入排序	323	9.4.3 归并排序函数定义	333
9.2.2 选择排序	325	9.4.4 算法分析	335
9.2.3 交换排序	327	9.5 其他排序方法.....	335
9.3 快速排序.....	328	9.5.1 分配排序和基数排序	335
9.3.1 快速排序的表实现	329	9.5.2 一些与排序有关的问题	338
9.3.2 程序实现	330	9.5.3 Python 系统的 list 排序	339
9.3.3 复杂度	331	本章总结.....	340
9.3.4 另一种简单实现	332	练习.....	342
9.4 归并排序.....	332	参考文献.....	344
9.4.1 顺序表的归并排序	333		

第1章 绪论

作为基于 Python 语言的“数据结构与算法”教程，本章首先讨论一些与数据结构和算法有关的基础问题，还将特别关注 Python 语言的一些相关情况。

1.1 计算机问题求解

使用计算机是为了解决实际问题。计算机具有通用性，其本身的功能很简单，就是能执行程序，按程序的指示完成一系列操作，得到某些结果，或者产生某些效果。要想用计算机处理一个具体问题，就需要有一个解决该问题的程序。经过长期努力，人们已经为各种计算机开发了许多有用的程序。在面对一个需要解决的问题时，如果恰好有一个适用的程序，事情就很方便了：运行这个程序，让它去完成所需工作。

实际中的计算需求无穷无尽，不可能都有现成的程序。如果面对一个问题，但没有适用的程序，可能就需要编写一个。一般而言，人们需要的不是解决一个具体问题的程序，而是解决一类问题的程序。例如，一个文本编辑器不应该只能编辑出一个具体的文本文件，而应该能用于编辑各种文本文件；Python 解释器不是只能执行一个具体的 Python 程序，而是可以执行所有可能的 Python 程序。对于求平方根这样的简单问题，人们希望的也不是专用于求某个数（例如 2）的平方根的函数，而是能求任何数的平方根的函数。求平方根是一个问题，求 2 的平方根是求平方根问题的一个实例。人们开发（设计，编写）一个程序，通常是为了解决一个问题，该程序的每次执行能处理该问题的一个实例。

简言之，用计算机解决问题的过程分为两个阶段：程序开发者针对要解决的问题开发出相应的程序，使用者运行程序处理问题的具体实例，完成具体计算（实际上，是计算机按程序的指示完成计算。为简单起见，人们常说程序完成计算，这样说不会引起误解）。开发程序的工作只要做一次，完成的程序可以多次使用，每次处理一个问题实例。当然，对于复杂的程序，完成后通常还需要修改完善，消除错误，升级功能。但这些是后话，无论如何，用计算机解决问题的第一步是开发出能解决问题的程序。

1.1.1 程序开发过程

程序开发就是根据面对的问题，最终得到一个可以解决问题的程序的工作过程。真正的问题来自实际，是不清晰和不明确的，而程序是对计算机操作过程的精确描述，两者之间有着非常大的距离。因此，一般而言，程序开发工作需要经过一系列工作阶段才能完成。由于人的认识能力的限制，其中还可能出现反复。图 1.1 刻画了这一过程中的各个工作阶段，以及实际程序开发的工作流程。

分析阶段：程序开发的第一步是弄清问题。实际中提出（发现）的问题往往是模糊的，缺乏许多细节，是一种含糊的需求。因此，程序开发的第一步就是深入分析问题，弄清其方方面面的情况和细节，将问题严格化，最终得到一个比较详尽的尽可能严格表述的问题描述。在软件开发领域，这一工作阶段通常被称为需求分析。

设计阶段：问题的严格描述仍然是描述性的，而计算机求解是一个操作过程。“一个问

题是什么”与“怎样做才能解决它”并不是一件事，在真正编程之前，需要先有一个能解决这个问题的计算过程模型。这种模型包括两个方面，一方面需要表示计算中处理的数据，另一方面必须有求解问题的计算方法，即通常所说的算法。由于问题可能很复杂，其中牵涉的数据不仅可能很多，数据项之间还可能有错综复杂的关系。为了有效操作，就需要把这些数据组织好。数据结构课程的主要内容就是研究数据的组织技术。如何在良好组织的数据结构上完成计算是算法设计的问题，是本书讨论的另一个重点。

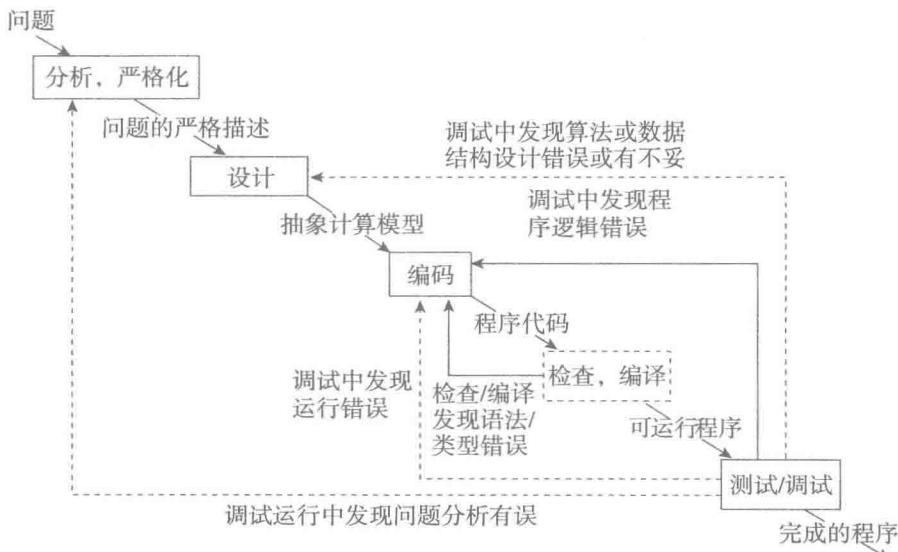


图 1.1 程序开发过程

编码阶段：有了解决问题的抽象计算模型，下一步工作就是用某种适当的编程语言实现这个模型，做出一个可能由计算机执行的实际计算模型，也就是一个程序。针对抽象计算模型的两个方面，编程中需要通过语言的各种数据机制实现抽象模型中设计的数据结构，用语言的命令和控制结构实现解决问题的算法。

检查测试阶段：复杂的程序通常不可能一蹴而就，写出的代码中可能有各种错误，最简单的是语法和类型错误。通过人或计算机（语言系统，编译器）的检查，可以发现这些简单错误。经过反复检查修改，最后得到了一个可以运行的程序。

测试 / 调试阶段：程序可以运行并不代表它就是所需的那个程序，还需要通过尝试性的运行确定其功能是否满足需要，这一工作阶段称为测试和调试。程序运行中可能出现动态运行错误，需要回到前面阶段去修改程序，消除这种错误。也可能发现得到的结果或效果不满足问题需要，这种错误称为逻辑错误。逻辑错误可能反映出编程中的失误，也可能是前面的算法设计有问题，甚至是开始的问题分析没做好。无论如何，发现错误之后，需要设法确定造成问题的原因，回到前面某个工作阶段去做适当的修正。然后根据情况在开发的后续步骤中做相应调整。这些工作需要反复进行，直至得到令人满意的程序。

图 1.1 和上面的说明阐释了从问题出发，最终得到可用程序的开发过程。在工作的第二和第三个阶段，算法和数据结构的设计和运用技术都扮演着重要角色：在第二个阶段需要设计抽象的数据结构和算法，第三个阶段考虑它们在具体编程语言中的实现。在设计阶段针对具体问题，建立一个可以用计算机实现的问题求解模型，而编码阶段（加上后续工作）真正实现这个

求解模型，完成一个可以在计算机上运行的程序。

相对而言，设计阶段的工作更困难一些。其工作基础是问题的说明性描述，有关信息并不能简单地映射到问题的操作性求解过程中，需要人的智力参与。为了完成这一工作，需要考虑被求解问题的性质和特点，参考人们用计算机解决问题的已有经验、已经开发的技术和方法。这方面的一些讨论将是本课程的重要内容。

编码阶段的工作相对容易一些。例如要用 Python 作为编程语言来解决问题，就需要把已经建立的抽象数据模型映射到 Python 语言可以表示的结构，把实际问题的抽象求解过程映射到一个用 Python 语言描述的计算过程。这两方面配合就得到了一个用 Python 语言写出的解决问题的程序。

下面将通过实例说明程序开发中的一些情况。

1.1.2 一个简单例子

虽然一个问题的说明性描述与其操作性描述表达的是同一个问题，但它们却非常不同。前者说明了需要解决的问题是什么，针对什么样的问题，期望什么样的解；而后者说明通过怎样的操作过程可以得到所要的解。对于一个给定的问题，用某种严格方式描述一个求解过程，且对该问题的每个实例，该过程都能给出解，这个描述就是解决该问题的一个算法。从算法到与之对应的程序，映射关系比较清晰。

现在用一个最简单的问题来说明。假设需要求出任一个非负实数的平方根。这句话是问题的一个非形式描述，工作的第一步就是需要把它严格化。

首先假设实数是一个已经清楚的概念，基于它考虑这个问题。在上面说明中，没有讲清楚的概念是平方根。根据数学中平方与平方根的定义，非负实数 x 的平方根就是满足等式 $y \times y = x$ 的非负实数 y 。这是一个严格的数学定义，说明了结果 y 应该满足的条件。但是，它并没有给出一种从任一 x 求出满足这个条件的 y 的方法。

从计算的角度看，上面定义还有一个重大缺陷：对于给定的数值，即使它只包含有穷位小数，其平方根通常也是一个无理数，不能写成数字的有穷表示形式。计算都需要在有穷步内完成，应该是一种有穷过程。因此一般而言，通过计算只能得到实数的平方根的近似值。在考虑求平方根的计算方法（算法）时，这个问题必须考虑，必须把近似误差作为参数事先给定。基于这一看法，上述问题可以修改为：对任意非负实数 x ，设法找到一个非负实数 y ，使得 $|y \times y - x| < e$ ，其中 e 是事先给定的允许误差。

这样就有了问题的一个严格描述。但这个描述是说明性的，说明了需要什么样的 y ，并没有告诉人们怎么得到这个结果。平方根是一个数学概念，要找到计算平方根的过程性描述（算法），也需要通过数学领域的知识。

人们已经提出了一些求平方根的方法。基本算术课程中介绍过如何求任一正实数的平方根，但在那个方法里需要做试除，不太适合机械进行（可以实现，但比较麻烦）。而求平方根的另一种算法称为牛顿迭代法，描述如下：

0. 对给定正实数 x 和允许误差 e ，令变量 y 取任意正实数值，如令 $y=x$ ；
1. 如果 $y \times y$ 与 x 足够接近，即 $|y \times y - x| < e$ ，计算结束并把 y 作为结果；
2. 取 $z=(y+x/y)/2$ ；
3. 将 z 作为 y 的新值，回到步骤 1。

首先，这是一个算法，因为它描述了一个计算过程。只要能做实数的算术运算、求绝对值和比较大小，就可以执行上面描述说明的计算过程。

但是，要确定这个算法能求出实数的平方根，还需要证明两个断言：①对任一正实数 x ，如果算法结束，它一定能给出 x 的平方根的近似值；②对任意给定的误差 e ，这个算法一定结束（实际上，这件事还与误差 e 和计算机实数表示精度有关）。

第一个断言很清楚，步骤 1 的条件 $|y \times y - x| < e$ 说明了这个断言成立。第二个断言则需要一个数学证明，证明计算过程中 y 值的序列一定收敛，其极限是 x 的平方根。这样，只要迭代的次数足够多， $|y \times y - x|$ 就能任意小，因此对任何允许误差 e ，这个循环都能结束。这个问题请读者自己考虑，这里不进一步讨论。

有了上面算法，写出相应 Python 程序已经不困难了。很容易定义一个完成平方根计算的 Python 函数，实现上述算法。下面是一个定义：

```
def sqrt(x):
    y = 1.0
    while abs(y * y - x) > 1e-6:
        y = (y + x/y)/2
    return y
```

其中变量 y 的初值为 1.0，允许误差为 10^{-6} 。通过用各种数值测试，可以看到这个函数确实能完成所需要的工作。

从这个简单实例可以看到从问题的描述出发最终得到一个可用程序的工作过程。由于求平方根的问题比较简单，特别是其中涉及的数据只是几个简单实数，数据组织的工作非常简单。下一节的实例将更好展现数据组织的有关情况。

还有一个情况值得注意。在上述例子中，最不清晰的一步就是从平方根的定义到求平方根的算法。算法设计是一种创造性工作，依赖于对问题的认识和相关领域的知识，没有放之四海而皆准的路径可循。计算机科学领域有一个研究方向是算法的设计与分析，计算机教育中有相应课程，其中讨论算法设计和研究的许多经验，总结算法设计中一些规律性的线索和思路。但经验也只是经验，在设计新算法时可以参考，但不能保证有效。算法分析则是分析算法的性质，将在 1.3 节进一步介绍。

1.2 问题求解：交叉路口的红绿灯安排

本节展示一个具体问题的计算机求解过程，以此说明在这种过程中可能出现的一些情况，需要考虑的一些问题，以及一些可能的处理方法。

交叉路口是现代城市路网中最重要的组成部分，繁忙的交叉路口需要用红绿灯指挥车辆通行与等待，正确的红绿灯调度安排对于保证交通安全、道路运行秩序和效率都非常重要。交叉路口的情况多种多样，常见形式有三岔路口、十字路口，也有较为少见的更复杂形式的路口。进一步说，有些道路是单行线，在中国的交叉路口还有人车分流和专门的人行 / 自行车红绿灯等许多情况，这些都进一步增加了路口交通控制的复杂性。要开发一个程序生成交通路口的红绿灯安排和切换，需要考虑许多复杂情况。

现在计划考虑的问题很简单，只考虑红绿灯（实际上是相应允许行驶路线）如何安排，可以保证相应的允许行驶路线互不交错，从而保证路口的交通安全。

为把问题看得更清楚，先考虑一个具体实例，希望从中发现解决问题的线索。作为实例的

交叉路口如图 1.2 所示[⊖]: 这是一个五条路的交叉口，其中两条路是单行线（图中用箭头标出），其余是正常的双向行驶道路。实际上，这个图形本身已经是原问题实例的一个抽象，与行驶方向无关的因素都已被抽象去除，例如道路的方位、不同道路交叉的角度、各条道路的实际宽度和通常的车流量等。

根据图形中表示的情况不难看到，从路口各驶入方向来的车辆，都可能向多个方向驶出，各种行驶方向的轨迹相互交错，形成很复杂的局面。按不同方向行驶的车辆可能相互冲突，存在着很实际的安全问题，因此，红绿灯的设计必须慎重！

现在的基本想法是对行驶方向分组，使得：

- 同属一组的各个方向行驶的车辆，均可以同时行驶，不出现相互交错的行驶路线，因此保证了安全和路线畅通。
- 所做的分组应该尽可能大些，以提高路口的效率（经济问题）。

第一条是基本的安全问题，丝毫不能妥协。而第二条只是经济问题，这个要求具有相对性。不难看出，允许同时行驶的方向越少，就越能保证安全。例如，每次只放行一个行驶方向，肯定能保证安全，但道路通行效率太低，因此完全不可取。另外还可以看到，这不是一个一目了然的问题，需要深入分析，才能找到较好的统一解决方法。

1.2.1 问题分析和严格化

图形的表达方式一目了然，特别有助于人们把握问题的全局。但是，如图 1.2 这样的图形并不适合表达问题细节和分析结果。如果把所有允许行驶方向画在图中，看到的将是来来去去、相互交错的许多有向线段，不容易看清其相互关系。

为了理清情况，应该先罗列出路口的所有可能行驶方向，例如从道路 A 驶向道路 B，从道路 A 驶向道路 C。为简便易读，用 AB、AC 表示这两个行驶方向，其他方向都采用类似的方式。不难列出路口总共有 13 个可行方向：AB, AC, AD, BA, BC, BD, DA, DB, DC, EA, EB, EC, ED。

采用这种抽象表示，问题的实质看得更清楚了。有了（给定了）一集不同的行驶方向，需要从中确定一些可以同时开绿灯的方向组。也就是说，需要为所有可能方向做出一种安全分组，保证位于每组里的行驶方向相互不冲突，可以同时放行。

这里的“冲突”又是一个需要进一步明确的概念。显然，两个行驶路线交叉是最明显的冲突情况[⊖]。如果对这样两个方向同时开绿灯，按绿灯行驶的车辆就有在路口撞车的危险，这是不能允许的。因此，这样的两个方向不能放入同一个分组。

为了弄清安全分组，需要设想一种表示冲突的方式，更清晰地描述冲突的情况。如果把所有行驶方向画在一张纸上，在相互冲突的方向之间画一条连线，就做出了一个冲突图。图 1.3 就是上面实例的冲突图，其中的 13 个小矩形表示所有的可能行驶方向，两个矩形之间有连线表示它们代表的行驶路线相互冲突。注意，在这个图形中，各个矩形的位置、大小等都

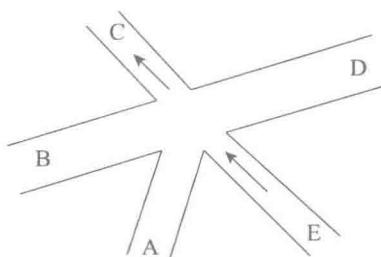


图 1.2 一个交叉路口实例的模型

[⊖] 本例参考张乃孝编著的《数据结构和算法》（由高等教育出版社出版）。

[⊖] 注意，“冲突”是对可能出现危险情况的认识，其定义也应该根据实际情况考虑。例如，在允许 BD 方向的情况下，是否同时允许 AD 和 ED，就可能有不同考虑。对冲突的不同定义将得到不同的解。

不代表任何信息，只有矩形中的符号和矩形之间的连线有意义。这样的图形构成了一种数据结构，也称为图，图中元素称为顶点，连线称为边或者弧。相互之间有边的顶点称为邻接顶点。第7章将会详细讨论这种结构。

有了冲突图，寻找安全分组的问题就可以换一种说法：为冲突图中的顶点确定一种分组，保证属于同一组的所有顶点互不邻接。显然，可能的分组方法很多。如前所述，将每个顶点作为一个组一定满足要求。但从原问题看，这里期待一种比较少的分组或者最少的分组。回到原问题，就是希望在一个周期中的红绿灯转换最少。

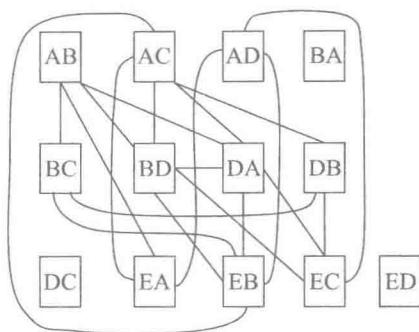


图 1.3 行驶线路冲突图

1.2.2 图的顶点分组和算法

经过一步步抽象和严格化，要解决的问题从交叉路口的红绿灯安排变成了一类抽象的图数据结构上的顶点分组。假设有这样一个图结构，现在需要考虑一个算法，其计算的结果给出一个满足需要的分组。对这个问题，安全性是第一位的基本要求，必须满足；而分组最少是进一步的附加要求，是一种追求。

以非相邻为条件的最佳顶点分组问题，实际上对应于有名的图最佳着色问题：把顶点看作区域，把相邻看作两个区域有边界，把不同分组看作给相邻顶点以不同颜色着色。著名的四色问题表明，对于以任一方式分割为区域的平面图，只需要用4种不同颜色着色，就能保证相邻区域都具有不同的颜色。但请注意，从交叉路口情况构造出的冲突图可能不是平面图[⊖]，因此完全可能需要更多的颜色。

人们对图着色的算法做过一些研究，目前的情况是：已经找到的最佳着色算法（得到最佳分组），基本上相当于枚举出所有可能的着色方案，从中选出使用颜色最少的方案。例如，一种直截了当的方法是首先基于图中顶点枚举出所有可能分组，从中筛选出满足基本要求的分组（各组中的顶点互不相邻），再从中选出分组数最小的分组。由于计算中需要考虑各种可能组合，这种组合数显然是顶点个数的指数函数，这个算法的代价太高了。能得到最佳分组的其他算法可能更复杂，但在效率上没有本质性提高。

下面考虑一种简单算法，它被称为是一种贪婪算法，或称贪心法。贪心法也是一种典型的算法设计思路（或称算法设计模式），其中的基本想法就是根据当时掌握的信息，尽可能地向得到解的方向前进，直到不能继续再换一个方向。这样做通常不能找到最优解，但能找到“可接受的”解，即给出一种较好的分组。

算法的梗概（伪代码）如下：

```

输入：图G          # 记录图中的顶点连接关系
集合verts保存G中所有顶点    # 建立初始状态
设置集合groups为空集      # 记录得到的分组，元素是顶点集合
while存在未着色顶点：        # 每次迭代用贪心法找一个新分组
    选一种新颜色
    在未着色顶点中给尽量多的无互连边的点着色（构建一个分组）
    记录新着色的顶点组

```

[⊖] 这里所说的平面图（图形），指在保持邻接关系不变的前提下调整顶点位置，可以将原图变换为另一个图，所得到的图中顶点间的边互不交叉。

```
# 算法结束时集合groups里记录着一种分组方式
# 算法细节还需要进一步考虑
```

现在考虑用这个算法处理图 1.3 中的冲突图，假定操作按照前面列出的 13 个方向的顺序进行处理。操作中的情况如下：

1. 选第 1 种颜色构造第 1 个分组：顺序选出相互不冲突的 AB、AC、AD，以及与任何方向都不冲突的 BA、DC 和 ED，做成一组；
2. 选出 BC、BD 和与它们不冲突的 EA 作为第二组；
3. 选出 DA 和 DB 作为第三组；
4. 剩下的 EA 和 EB 作为第四组。

由于上面算法中这几个分组内互不冲突，而且每个行驶方向属于一个分组，因此是满足问题要求的一个解。得到的分组如下：

```
{AB, AC, AD, BA, DC, ED}, {BC, BD, EA}, {DA, DB}, {EA, EB}
```

上面算法还有重要的细节缺失：一种新颜色的着色处理。现在考虑这个问题。

假设图 G 保存需着色图中顶点的邻接信息，集合 `verts` 是图中所有尚未着色的顶点的集合。显然，算法开始时 `verts` 应该是 G 中所有顶点的集合。用另一个变量 `new_group` 记录正在构造的用当前新颜色着色的顶点（一个集合），在上面算法的每次迭代中重新构造，每次开始做分组时将这个集合重新设置为空集。

在上面安排的基础上，找出 `verts` 中可用新颜色着色的顶点集的算法是：

```
new_group = 空集
for v in verts:
    if v与new_group中所有顶点之间都没有边：
        从verts中去掉v
        把v加入new_group

# 循环结束时new_group是可以用一种新颜色着色的顶点集合
# 用这段代替前面程序框架中主循环体里的一部分
```

这样就有了一个完整的算法。

检查上面算法，可以看到算法中假设了一些集合操作和一些图操作。集合操作包括判断一个集合是否为空集，构造一个空集，从一个集合里删除元素，向一个集合里加入元素，顺序获得集合里的各个元素（上面算法中 `for` 循环做这件事）。图操作包括获取图中所有顶点、判断两个顶点是否相邻。

上面讨论中实际上也介绍了两种最基本的算法设计方法：

- 枚举和选择（优选）：根据问题，设法枚举出所有可能的情况，首先筛选出问题的解（为此需要判断枚举生成的结果是否为问题的解），而后从得到的解中找出最优解（这一步需要能评价解的优劣）。
- 贪心法：根据当时已知的局部信息，完成尽可能多的工作。这样做通常可以得到正确的解，但可能并非最优。对于一个复杂的问题，全面考虑的工作代价可能太高，为得到实际可用的算法，常常需要在最优方面做出妥协。

1.2.3 算法的精化和 Python 描述

前面给出了一个解决图着色的抽象算法，它与实际程序还有很大距离。要进一步考虑如何实现，还有很多需要处理的细节，例如：

- 如何表示颜色？例如，可以考虑用顺序的整数。
- 如何记录得到的分组？可以考虑用集合表示分组，把构造好的分组（集合）加入 groups 作为元素（也就是说，groups 是集合的集合）。
- 如何表示图结构？

实际上，是否把“颜色”记入 groups 并不重要，可以记录或者不记录。下面的考虑是记录颜色，将“颜色”和新分组做成二元组。

现在考虑如何把上述算法映射到一个 Python 程序中，填充其中的许多细节。前面考虑的集合操作大都可以直接用 Python 的集合操作：

- 判断一个集合 vs 是空集，对应于直接判断 `not vs`。
- 设置一个集合为空，对应于赋值语句 `vs = set()`。
- 从集合中去掉一个元素的对应操作是 `vs.remove(v)`。
- 向集合里增加一个元素的对应操作是 `vs.add(v)`。

Python 的集合数据类型不支持元素遍历，但上述算法中需要遍历集合元素，还要在遍历中修改集合。处理这个问题的方法是在每次需要遍历时从当时的 `verts` 生成一个表，而后对表做遍历（并不直接对集合遍历）。

算法中需要的图操作依赖于图的表示，需要考虑如何在 Python 中实现图数据结构。图是一种复杂数据结构，应该支持一些操作。图的可能实现方法很多，第 7 章将详细讨论这方面问题。这里假定两个与图结构有关的操作（依赖于图的表示）：

- 函数 `vertices(G)` 得到 G 中所有顶点的集合。
- 谓词 `not_adjacent_with_set(v, group, G)` 检查顶点 v 与顶点集 group 中各顶点在图 G 中是否有边连接。

假设有了图结构及其操作，程序实现就不难了。下面是一个程序（算法）：

```
def coloring(G):                      # 做图G的着色
    color = 0
    groups = set()
    verts = vertices(G)                # 取得G的所有顶点，依赖于图的表示
    while verts:                      # 如果集合不空
        new_group = set()
        for v in list(verts):
            if not_adjacent_with_set(v, new_group, G):
                new_group.add(v)
                verts.remove(v)
        groups.add((color, new_group))
        color += 1
    return groups
```

这个算法实际上已经是一个 Python 函数，能对任何一个图算出顶点分组。其中欠缺的细节就是图的表示，以及函数里涉及的两个图操作。

1.2.4 讨论

完成了工作并不是结束。任何时候完成了一个算法或者程序，都应该回过头去仔细检查做出的结果，考虑其中有没有问题。做出了一个图着色程序，传递给它一个冲突图，它将返回一个分组的集合。现在应该回过头进一步认真考虑工作中遇到的各种情况和问题，包括一些前面没有关注的情况。

解唯一吗？

首先，可以看到，对于给定的交叉路口实例，可行的分组可能不唯一。除了前面几次提到的每个方向独立成组的平凡解之外，完全可能找到一些与前面给出的解的分组数相同的解。对前面问题实例，下面是另一种满足要求的分组：

$\{AB, EB, EC\}, \{AC, AD, BC\}, \{BA, BD, DB, ED\}, \{DA, DC, EA\}$

读者不难验证这一分组确实是该问题实例的解。

算法给出的解是确定的，依赖于算法中选择顶点的具体策略，以及对图中顶点的遍历顺序，即 `list(verts)` 给出的顶点序列中的顺序。

求解算法和原问题的关系

回顾前面从问题出发最终做出一个 Python 程序的工作过程：

1. 有关工作开始于交叉路口的抽象图示，首先枚举出所有允许通行方向；
2. 根据通行方向和有关不同方向冲突的定义，画出冲突图；
3. 把通行方向的分组问题归结为冲突图中不相邻顶点的划分问题，用求出不相邻顶点的分组作为交叉路口中可以同时通行的方向分组。

问题是，这样得到的结果满足原交叉路口问题实例的需要吗？

仔细分析可以看到，上面算法中采用的定义不统一：算法给出的结果是行驶方向的一种划分（各分组互不相交，每个顶点只属于一个分组）；而工作开始考虑安全分组时，只要求同属一组的顶点（行驶方向）是互不冲突的。也就是说，原问题允许一个行驶方向属于不同分组的情况（现实情况也是这样，可能某个行驶方向在多种情况下都是绿灯。典型的例子如无害的右转弯）。出现分组不相交的情况，原因是在构建新分组时一旦选择了某个顶点，就将其从未分组顶点集中删除，这样就产生了划分的效果。

要回到求解之前的考虑，并不一定推翻得到的算法，也可能在原算法上调整。例如，对于图 1.2 的交叉路口实例，前面算法给出：

$\{AB, AC, AD, BA, DC, ED\}, \{BC, BD, EA\}, \{DA, DB\}, \{EA, EB\}$

将分组尽可能扩充，加入与已有成员不冲突的方向，得到：

$\{AB, AC, AD, BA, DC, ED\}, \{BC, BD, EA, BA, DC, ED\},$

$\{DA, DB, BA, DC, ED, AD\}, \{EA, EB, BA, DC, ED, EA\}$

根据前面的定义，这样得到的各分组仍然是安全的，请读者检查。如何修改前面算法，使之能给出这样分组的问题留给读者考虑。

另一个问题前面已有所讨论，就是冲突概念的定义问题。前面采用行驶方向交叉作为不安全情况的定义，这只是一个合理选择。另外的情况是否看作冲突，可能存在不同考虑。如前面提到的直行与旁边道路的右转问题（例如，在允许 BD 方向的情况下，是否允许 AD 和 ED ）。对同一个路口，如果冲突的定义不同，做出的冲突图就会不同。实际定义可能需要反映交管部门对具体路口实际情况的考量，需要根据具体情况确定。

还有许多实际问题在前面算法中都没有考虑。例如，在用于控制交叉路口的红绿灯时，得到的行驶方向分组应该按怎样的顺序循环更替？这里能不能有一些调度的原则，能不能通过算法得出结果？另一些问题更实际，可能更难通过计算机处理。例如各个分组绿灯持续的时间，这里牵涉到公平性、实际需要等。可能还有其他问题。

从以上讨论中可以看到，前面工作中从问题出发逐步抽象，得到的算法处理的问题与原问题已经有了很大的距离。该算法的输入是经过人工分析和加工而得到的冲突图，做出冲突图需

要确定冲突的定义，是人为的一步。考虑实际需要的红绿灯调度，该算法产生的结果也缺乏许多信息，真正运用到实际还需要人工做许多工作。

对于上面这些情况，在用计算机解决实际问题时经常可能看到。

小结

本节的假设是希望做出一个程序，给它任意一个交叉路口的有关信息，它能对该路口的可能行驶方向做出一种安全分组。经过仔细分析问题，考虑了一些情况，前面讨论已经给出了一种解决问题的方案（算法），并进一步精化，给出了另一个采用 Python 语言形式描述的“函数定义”。但这还是一个可以运行的程序。

Python 是一种比较高级的语言，提供了许多高级的数据类型和相关操作。针对这个算法，Python 语言的集合和相关操作可以直接使用。但算法中的一些功能是 Python 语言没有的，主要是缺少图结构及若干相关操作。如果设法在 Python 语言里实现图结构及所需操作，这个“算法”就变成了一个可以运行的实际程序。

另一些编程语言没有高级的数据类型，只提供了很少的一组基本类型和几种数据组合机制。C 语言就是这样，只有几个类型和数组、结构、指针等低级组合机制。要在 C 语言里实现上述算法，就需要自己实现集合、图及相关操作。

在解决各种问题的程序里，经常要用到诸如集合、图等复杂的数据结构，用于表达计算中获得、构造和处理的复杂信息。理解这类高级结构的各方面性质，在编程语言里有效实现它们，是本书后面章节中讨论的主要问题。对于提供了一些高级数据结构的 Python 语言，本书还将介绍这些结构的性质和合理使用方法。

1.3 算法和算法分析

本节集中讨论算法的问题，特别是算法的性质及其分析技术。

1.3.1 问题、问题实例和算法

在考虑计算问题时，需要清晰地区分问题、问题实例和算法三个概念，并理解它们之间的关系，这就是本小节讨论的内容。

三个基本概念

考虑一个计算问题时，需要注意三个重要概念：

- **问题：**一个问题 W 是需要解决（需要用计算求解）的一个具体需求。例如判断任一个正整数 N 是否为素数，求任一个方形矩阵的行列式的值等。虽然可以严格定义“问题”的概念，但在这里还是想依靠读者的直观认识。总而言之，现实世界中存在许许多多需要用计算解决的问题，它们是研究和实现算法的出发点。
- **问题实例：**问题 W 的一个实例 w 是该问题的一个具体例子，通常可以通过一组具体的参数设定。例如判断 1013 是否为素数，或者求一个具体方形矩阵的行列式的值。显然，一个问题反映了其所有实例的共性。
- **算法：**解决问题 W 的一个算法 A ，是对一种计算过程的严格描述。对 W 的任何一个实例 w ，实施算法 A 描述的计算过程，就能得到 w 的解。例如，一个判断素数的算法应该能给出 1013 是否为素数的判断，也能判断其他正整数是否为素数；一个求矩阵的行列式值的算法必须能应用于任何矩阵，得到其行列式的值。

下面是一些计算问题及其实例：

- 求两个（二维）矩阵的乘积，求任意两个具体矩阵的乘积都是其实例。
- 将一个整系数多项式分解为一组不可约整系数多项式因子的乘积，计算具体多项式的因式是这个问题的实例。
- 将一个图像旋转 90 度，其实例是要求旋转的各种具体图像。
- 辨认出数字相机取景框里的人脸，每次拍照的取景框图像都是其实例。

要用计算机解决问题，需要开发能解决该问题的方法（和算法），然后实现相关的程序。解决一个问题的算法应能统一求解该问题的（所有）实例。对于一个可以用计算机解决的问题，完全可能有多种不同的算法。

算法的性质

一个算法是对一种计算过程的一个严格描述，人们通常认为算法具有如下性质：

- **有穷性**（算法描述的有穷性）：一个算法的描述应该由有限多条指令或语句构成。也就是说，算法必须能用有限长的描述说清楚。
- **能行性**：算法中指令（语句）的含义严格而且简单明确，所描述的操作（计算）过程可以完全机械地进行。
- **确定性**：作用于所求解问题的给定输入（以某种描述形式给出的要处理问题的具体实例），根据算法的描述将产生出唯一的确定的一个动作序列。使用算法就是把问题实例（的表示）送给它，通过确定性的操作序列，最终得到相应的解。
- **终止性**（行为的有穷性）：对问题的任何实例，算法产生的动作序列都是有穷的，它或者终止并给出该问题实例的解；或者终止并指出给定的输入无解。
- **输入 / 输出**：有明确定义的输入和输出。

满足确定性的算法也称为确定性算法。实际上，现在人们也关注更广泛的概念，例如考虑各种非确定性的算法，如并行算法、概率算法等。另外，人们也关注并不要求终止的计算描述，这种描述有时被称为过程（procedure）。

解决重要问题的算法不仅具有理论价值，也有实际价值。例如：

- Google、百度等网络搜索公司开发和使用的各种网络检索和排名算法，它们能有效地利用大量计算机设备，收集和整理存在于互联网上的大量杂乱无章的信息，满足网络用户的需要。其中非常关键的一件事是对与用户请求有关的信息做出很好的排序，把最可能满足用户需要的信息排在最前面，称为 ranking（排名）。
- 现代社会特别重视信息的传播和存储的安全性，在社会生活中使用越来越广泛的电子金融、电子商务，都对安全性提出了特别的要求。而计算机网络和网络服务的安全性依赖于所用的加密方法，高安全和高效率的加密解密算法极端重要。

算法的描述

前面说一个算法严格地描述了一种计算过程，但并没有说算法描述的形式和方式。实际上，算法与数学中的定义和定理类似，通常用于人与人之间的交流，但交流的内容是问题的解决过程，有关一个计算应该如何进行。主要目的是帮助人理解和思考相应的问题求解方法、技术和过程。由于主要为了人们阅读和使用，算法可以采用不同的描述形式，需要在易读、易理解和严格性之间取得某种平衡。算法的描述应基于清晰定义的概念，包含足够多的细节，使人可以按照算法一步步工作完成具体问题实例的求解。常见的描述形式如：

- 采用自然语言描述。用自然语言描述的计算过程可能比较容易阅读，但可能比较冗长

啰唆，也容易出现歧义，造成读者的误解。

- 采用在自然语言描述中结合一些数学记法或公式的描述形式。这是前一描述形式的变形，主要是为了简洁、严格（消除歧义），减少误解的可能性。
- 采用严格定义的形式化记法形式描述。例如：
 - 采用某种通用的计算模型描述方式，如采用著名的图灵机模型，用一个完成相应计算的图灵机描述算法。这种描述完全是严格的，没有歧义，但通常会非常繁琐，极难阅读，而且难以进一步使用。
 - 采用某种严格的专门为描述算法而定义的形式化描述语言。这样做可以避免歧义性，但目前还没有公认最为适用的语言。
- 用类似某种编程语言的形式描述算法过程，其中掺杂使用一些数学符号和记法，用于描述算法中一些细节和具体操作。前面实例中采用了类似 Python 的控制结构，结合自然语言和数学符号。利用编程语言的控制结构，可以简洁、清晰地反映算法中操作的控制、执行条件和执行顺序，常可以得到较为简洁的算法描述。这种方式的问题是可能涉及过多细节，不利于跨语言使用。这类问题应尽量避免。
- 采用某种伪代码形式，结合编程语言的常用结构、形式化的数学记法代表的严格描述和自然语言。这种方式与前一方式类似，但不过多拘泥于具体语言。

目前最常见的是后两种描述形式。在本书中需要描述算法时，通常采用 Python 语言程序的形式，结合自然语言和数学表达形式，主要用于局部的功能说明。

算法和程序

计算问题通常都很复杂，问题实例可能很大，解决它们需要执行数以千万计的具体操作。人工计算只能处理极简单问题的规模很小的实例，不能完成大规模计算。要解决有一定规模、有实际价值的问题，必须借助于能自动运行的计算机。今天能利用的就是常见的电子计算机。要指挥其工作，就需要做出计算机能执行的程序。

程序可以看作采用计算装置能够处理的语言描述的算法，由于它是算法的实际体现，又能在实际计算机上执行，因此被称为算法的实现。

程序可能用各种计算机语言描述。例如用直接对应于特定计算机硬件的机器语言或者汇编语言。也可以用通用的高级编程语言，如 C、Java 等。本书中将用 Python 语言描述程序，定义各种数据结构，描述各种算法。

程序和算法密切相关。在每一个程序背后都隐藏着一个或者一些算法。如果一个程序正确实现了一个能解决某个问题的算法，用这个程序处理该问题的实例就应该得到相应的解。此外，该程序运行时的各种动态性质，也应该反映它所实现的算法的性质，这样才是相应算法的合理实现。下面还会进一步讨论这个问题。

另一方面，由于程序是用计算机能处理的某种具体编程语言描述的，其中必然会包含一些与具体语言有关的细节结构和描述方式方面的特征。所用的语言不同，不仅可能影响算法描述的方便性，也可能影响到程序的运行效率。

由于这些情况，在抽象地考虑一个计算过程或考虑一个计算过程的抽象性质时，人们常用“算法”作为术语，用于指称相应计算过程的描述。而在考虑一个计算在某种语言里的具体实现和实现中的问题时，人们常用“程序”这一术语讨论相关问题。本书也采用这种通行的做法。此外，有时书中描述的是一个程序，但在讨论时却说“算法”。这时实际想说的就是该程序背后的与具体语言无关的计算过程。

算法设计与分析

算法是计算机科学技术中最重要的研究课题，其具体应用遍及计算机科学技术的所有研究和应用领域。由于算法的重要性，人们对算法设计和开发的经验做了许多总结，这一研究领域称为算法设计与分析。

一方面，所谓算法设计，就是从问题出发，通过分析和思考最终得到一个能解决问题的过程性描述的工作过程。算法设计显然是一种创造性工作，需要靠人的智慧和经验，不可能自动化。实际应用问题千变万化，解决它们的算法的表现形式也多种多样。但是许多算法的设计思想有相似之处，相关工作也有规律性可循。人们深入研究了前人在算法领域的工作经验和成果，总结出了一批算法设计的重要思路和模式。对算法设计方法做分类研究，可以给后人的学习提供一些指导。已有的设计经验和模式可能提供一些有用的线索和启发，供人们在设计解决新问题的新算法时参考。

算法设计中一些常见的通用想法可以称为算法设计模式。常见模式包括：

- 枚举法。根据具体问题枚举出各种可能，从中选出有用信息或者问题的解。这种方法利用计算机的速度优势，在解决简单问题时十分有效。
- 贪心法。如前所述，根据问题的信息尽可能做出部分的解，并基于部分解逐步扩充得到完整的解。在解决复杂问题时，这种做法未必能得到最好的解。
- 分治法。把复杂问题分解为相对简单的子问题，分别求解，最后通过组合起子问题的解的方式得到原问题的解。
- 回溯法（搜索法）。专指通过探索的方式求解。如果问题很复杂，没有清晰的求解路径，可能就需要分步骤进行，而每一步骤又可能有多种选择。在这种情况下，只能采用试探的方式，根据实际情况选择一个可能方向。当后面的求解步骤无法继续时，就需要退回到前面的步骤，另行选择求解路径，这种动作称为回溯。
- 动态规划法。在一些复杂情况下，问题求解很难直截了当地进行，因此需要在前面的步骤中积累信息，在后续步骤中根据已知信息，动态选择已知的最好求解路径（同时可能进一步积累信息）。这种算法模式被称为动态规划。
- 分支限界法。可以看作搜索方法的一种改良形式。如果在搜索过程中可以得到一些信息，确定某些可能的选择实际上并不真正有用，就可以及早将其删除，以缩小可能的求解空间，加速问题求解过程。

这里需要说明两点：首先，上述算法设计模式只是人们对经验的总结，只能借鉴，不应作为教条。其次，这些模式并不相互隔绝也不互相排斥。例如，一般而言复杂的问题都需要分解；而最简单的情况经常可能采用枚举和判断的方式处理。所以，在很多复杂算法中可以看到多种算法模式的身影，是多种模式的有机组合。还是应该强调，对于算法设计而言，并没有放之四海而皆准的设计理论或技术，只能借鉴和灵活运用前人经验。算法是智力活动的产物，一个好算法至少等价于一个好的定理及其证明，因为算法不仅说明了一件事可以用计算机处理，而且还可以实现为程序，真正利用计算机去解决问题。

在设计和使用算法的过程中，人们也深入地研究了算法的各种性质，包括所有算法共有的性质和一些具体算法的具体性质。在算法的共有性质中，最重要的就是算法的实施都需要耗费资源，即一个算法的实施必定蕴涵着一定量的时间和空间开销。算法分析的主要任务就是弄清算法的资源消耗。下一小节将专门讨论这个问题。算法分析的最重要作用是作为评价算法的一种标准。例如，解决同一个问题常常可以设计出多个不同算法，显然，工作中消耗时间和空间

更少的算法通常更为可取。

1.3.2 算法的代价及其度量

在研究现实世界中的计算问题时，必须考虑计算的代价。原因也很清楚：一方面，实际的计算机是一种物理设备，执行每个操作都耗费时间。虽然今天的计算机非常快，每秒可以完成数以十亿计的操作，但每次操作还是需要一点点时间，而大量的一点点时间累积起来就可能超过任意大的固定的时间限制。另一方面，计算中都需要保存被处理的数据，为了完成复杂计算，通常还需要保存许多中间结果。虽然今天最普通的计算机也有数以十亿计的存储单元，但其存储量毕竟不是无穷大，总有可能用完。

在考虑求解一个问题的具体算法时，需要考虑用它解决问题的代价，包括该算法：在求解过程中需要多少存储空间？完成问题实例的求解需要多长时间？使用者需要关心算法运行中存储占用的高位限，以及得到结果的时间。

度量的单位和方法

实际计算机的种类很多，其中各种操作的执行速度、存储单元具体大小都可能不同。对于算法的理论研究应该具有一定的抽象性，应该排除具体机器的非普遍性特征，抽象地反映算法的共性和本质，这样得到的抽象研究结果才有更广泛的指导意义。为了抽象地研究算法的性质，需要做一些简单假设：

- 所用的计算设备中为数据存储准备了一组基本单元，每个单元只能保存固定的一点有限数据。在考虑算法的时间性质时，将以此作为空间的单位。
- 机器能执行一些基本操作，计算中的一次操作消耗一个单位的时间。

在考虑具体的算法时，可以根据需要选择合理的存储单位，只要求其大小固定且有穷，以其计数值作为存储开销（空间开销）的基本度量。另一方面，可以根据情况选择一组合适的操作，假定执行它们需要一个单位时间，以其执行次数作为时间开销的基本度量。

作为评价算法的尺度，需要有一种合理的度量方法。现在首先讨论这个问题，讨论中主要考虑计算的时间开销，空间开销的情况类似，不再另行说明。

假设有了解决某具体计算问题 W 的一个具体算法 A，现在需要考虑 A 在计算中的时间开销。根据前面的讨论，算法 A 应该能处理问题 W 的任何实例。然而，同属一个问题的实例也有大有小，实例的规模不同，算法求解中需要付出的代价也可能不同。例如，采用同一个算法，判断 1013 是否为素数或判断 10013131301131 是否为素数，需要花费的时间通常是不同的。计算矩阵行列式值的问题更加明显，要计算 3×3 矩阵的行列式，只牵涉到 9 个元素的处理，而计算 3000×3000 矩阵的行列式，牵涉到的元素多了一百万倍，需要做更多运算，所需要的时间必然会多得多。

现在需要度量的是算法，同一个算法能应用于不同的实例，计算的实际代价通常与实例的规模（大小）有关。人们自然希望对一个算法的度量结果能适用于任何实例，因此这种度量必须反映实例的规模对计算开销的影响。为了处理这种情况，人们提出的方法是把一个算法的计算（时间和空间）开销定义为问题的实例规模的函数。也就是说，算法分析就是针对一个具体算法，设法确定一种函数关系，以问题实例的某种规模 n 为参量，反映出这个算法在处理规模 n 的问题实例时需要付出的时间（或者空间）代价。

把算法的代价看作规模的函数之后，很容易看到一种必然出现的情况：可能有一些算法，随着实例规模的增长，其时间（或空间）开销的增长非常快，而另一些算法的开销函数随着规

模增长而增长得比较慢。这种相异的函数关系实际上反映了算法的性质。从高等数学的角度看，这种函数关系描述一种增长趋势：在规模 n 趋向于无穷的过程中，有关函数的增长速度。下面将这两个函数关系称为算法的时间代价和空间代价。

与实例和规模有关的两个问题

这里还有两个问题需要说明。首先是问题实例的规模，这是分析一个算法时采用的基本尺度，算法的代价将基于它描述。尺度不同，有关算法代价的结论就会不同。实际情况会怎么样呢？先看看具体实例中的情况。

对判断素数的问题，有两个尺度可能作为实例的规模度量。一种可能是用被检查整数的数值，另一种可能是取被检查整数按某种进制（例如十进制或二进制）表示的数字串长度^Θ。这里出现了一个大问题：整数的数值与其进制表示的数字串长度之间有着指数关系，长 m 的数字串可以表示数值直至 $B^m - 1$ 的整数，其中 B 为进制的基数（例如 10 或者 2）。对矩阵行列式问题也有两种可能，可以取矩阵的维数作为尺度，或者取矩阵中的元素个数。两者之间也有一个平方的差异。虽然上述几种尺度选择好像都有道理，但是，选择不同的尺度描述算法的代价，就会得到很不一样的结果。

对算法的一般性研究不仅希望比较解决同一个问题的不同算法，还希望对解决不同问题的算法之间的关系有所认识。因此，人们希望有一种统一的尺度作为基础来讨论算法的代价。由于问题实例对应于算法的输入，送给素数判断算法的输入就是可能长也可能短的数字串，送给矩阵行列式算法的是表示矩阵的那一组元素。对实例的规模，最直接的反映就是输入数据的多少，这种看法覆盖了一切计算问题和算法。

按照上述观点，对于判断素数的问题，应该用整数表示的数字串长度作为问题规模的尺度；而对于矩阵的行列式求值，应该用矩阵元素的个数（或者元素个数乘以每个元素的表示长度，这里差一个常量因子）。对一些具体问题，有时人们也有习惯的做法。例如对于与矩阵有关的算法，人们讨论时经常用矩阵维数作为规模的尺度。按照这种做法给出的算法代价，与基于矩阵元素个数描述就差了一个平方。

还有一个问题也值得注意：使用同一个算法解决问题时，即使对于同样规模的实例，计算的代价也可能不同。以素数判断算法为例，同样由 100 个十进制数字表示的数，如果被判断的是偶数，算法启动后只需做一次检查，立刻就可以给出否定的结果；如果是奇数，就需要花更多时间。在这方面，处理不同问题的算法，其性质也可能不同。

考虑处理数值表（数值序列）的两个问题：如果希望求出表中数据的平均值，任何算法都需要先求出所有数据之和，对同样大小的实例，计算的代价都差不多。如果问题是找到数据中第一个小于 0 的项，情况就很不一样。采用顺序检查的算法处理同样包含 10000 个数据的表，有时很快就能得到结果，有时则要检查完所有的数据。

针对这种情况，在度量算法 A 的代价时（以时间为例），存在几种可能的考虑。在处理规模为 n 的问题实例时，考虑：

- 算法 A 完成工作最少需要多少时间？
- 算法 A 完成工作最多需要多少时间？
- 算法 A 完成工作平均需要多少时间？

^Θ 注意，只要基数 $B \geq 2$ ，不同基数下同一整数的编码长度只相差一个常数因子。例如整数 n 的二进制编码长度与十进制编码长度之比大约是 $\frac{\log_2 n}{\log_{10} n} = \log_2 10$ 。

第一种考虑的价值不大，因为它没提供什么有用信息。在实际中也是如此，对完成工作所需时间的最乐观的估计基本上没有价值。

第二种考虑提供了一种保证，说明在这段时间之内，本算法一定能完成工作。这种代价称为最坏情况的时间代价。

第三种考虑是希望对算法 A 做一个全面评价（处理规模为 n 的实例的平均时间代价），因此它完整全面地反映了这个算法的性质。但另一方面，这个时间代价并没有保证，不是每个计算都能在这一时间内完成。此外，“平均”还依赖于所选的实例，以及这些实例出现的概率。通常总需要做一些假定，例如假定规模同为 n 的各种实例是均匀分布的。然而在应用实践中，实际问题实例有自己的分布情况，未必与理论分布一致，而实际中的真实分布通常很难确定。这些都给平均代价概念的应用带来困难。

在有关算法的研究和分析中，人们主要关注算法的最坏情况代价，有时也关注算法的平均代价。本书后面的章节里也这样考虑。

常量因子和算法复杂度

对一个具体程序进行细致分析，有可能弄清在用它处理一个具体实例的过程中，各种操作执行的具体次数。通过对一些不同规模实例的分析，也有可能把得到的操作次数总结为问题实例规模的一个具体函数。当然，这样做牵涉到很多细节，再加上程序描述中可能出现分支的情况，给出一个统一的精确描述经常是非常困难的。

对于抽象的算法，通常无法做出精确度量。在这种情况下只能退而求其次，设法估计算法复杂性的量级。另一方面，分析算法性质是为了算法的设计和评价，最终是为了用程序实现算法。由于程序运行所用的计算机、描述程序的语言、实现方法等不同，同样算法在实际运行中所需的时间也可能有些变化。

这些情况说明，特别具体的细致分析虽然很好，但在实践中的价值有限。对于算法的时间和空间性质，最重要的是其量级和趋势，这些是代价的主要部分，而代价函数的常量因子可以忽略不计。例如，可以认为 $3n^2$ 和 $100n^2$ 属于同一个量级，如果两个算法处理同样规模实例的代价分别为这两个函数，就认为它们的代价“差不多”。基于这些考虑，人们提出描述算法性质的“大 O 记法”。

首先给出“大 O 记法”的严格定义：对于单调的整数函数 f ，如果存在一个整数函数 g 和实常数 $c > 0$ ，使得对于充分大的 n 总有 $f(n) \leq c \cdot g(n)$ ，就说函数 g 是 f 的一个渐近函数（忽略常量因子），记为 $f(n) = O(g(n))$ 。易见， $f(n) = O(g(n))$ 说明在趋向无穷的极限意义上，函数 f 的增长速度受到函数 g 的约束。

把上述描述方式应用于算法的代价问题。假设存在函数 g ，使得算法 A 处理规模为 n 的问题实例所用的时间 $T(n) = O(g(n))$ ，则称 $O(g(n))$ 为算法 A 的渐近时间复杂度，简称时间复杂度。算法的空间复杂度 $S(n)$ 的定义与此类似。

这里有几点说明。

首先，如果 $T(n) = O(g(n))$ ，那么对于任何增长速度比 $g(n)$ 更快的函数 $g'(n)$ ，显然也有 $T(n) = O(g'(n))$ 。这说明上述定义考虑的是算法复杂度的上限。虽然这种描述并不精确，但对于本书的讨论以及很多计算机实践已经足够了。

其次，虽然可以选择任意的合适的函数作为描述复杂性时使用的 $g(n)$ ，但是一组简单的单调函数已足以反映人们对基本算法复杂度的关注。在算法和数据结构领域，人们最常用的是下面这组渐近复杂度函数：

$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$

在后面讨论中，也经常把 $O(1)$ 称为常量复杂度，把 $O(\log n)$ 称为对数复杂度，把 $O(n)$ 称为线性复杂度，把 $O(n^2)$ 称为平方复杂度，把 $O(2^n)$ 称为指数复杂度，等等。注意，根据前面脚注，在考虑量级时对数的底并不是主要因素（可能差一个常量因子），因此可以忽略。另外，一个算法的时间复杂度为 $O(n)$ ，也常说这个算法是 $O(n)$ 时间的算法，或者说这个算法需要 $O(n)$ 时间等。

假设算法 A_1 具有平方时间复杂度，但不是线性复杂度，也就是说，其时间代价函数 $T_1(n)=O(n^2)$ ，但 $T_1(n)\neq O(n)$ ；而另一个算法 A_2 具有线性的时间复杂度。显然，只要问题的规模 n 足够大，算法 A_2 通常会比算法 A_1 快得多[⊖]。

图 1.4 描述了不同复杂度函数的增长速度。下表给出了对于几个具体 n 值 ($n=10, 20, 30, 40, 50$)，几个常用的复杂性描述函数的值，从中可以看到这些函数值的一些变化情况和趋势。

$\log n$	3.219	4.322	4.907	5.322	5.644
n	10	20	30	40	50
$n \log n$	32.19	86.44	147.21	212.88	282.2
n^2	100	400	900	1600	2500
n^3	1000	8000	27000	64000	125000
2^n	1024	1048576	$\approx 1.0 \times 10^9$	$\approx 1.0 \times 10^{12}$	$\approx 1.0 \times 10^{15}$

按照上面的理论，如果算法的改进只是加快了常量倍，也就是说减小了复杂性函数中的常量因子，算法的复杂度没有变化。但是，在许多实际情况中，这种改进也可能是很有意义的。例如，需要 3 天时间算出明天的天气预报，与只需要半天时间就能算出明天的天气预报，算法的实际价值是截然不同的。前者毫无价值而后者就能实用了。理论分析主要是给出了一种基本趋势，使人可以从较为宏观的角度认识所用的算法。

如果有 $T(n)=O(g(n))$ (或者 $S(n)=O(g(n))$)，函数 $g(n)$ 是算法的实时间开销的一个上界，并不表示实际开销真正具有与 $g(n)$ 同样的增长速度 (只说明其增长速度不超过 $g(n)$)。进一步说，也可以考虑复杂性的下界 (算法代价的增长速度不低于……)，或者上确界。一些算法分析书籍里引进了不同的记法表示这些概念。本书中将只所用“大 O 记法”，但在分析算法时，尽可能考虑“紧的”上界。对一些复杂的算法，找出其时间或空间开销的上确界很不容易。总而言之，“大 O 记法”给出的也是一种保证。

算法复杂度的意义

不难看出，算法的复杂度分级相当于 (高等数学里) 无穷大的阶，反映了在规模 n 趋于无穷大的过程中，算法代价增长的速度。算法的复杂度越高，其实施的代价随着规模增大而增长的速度就越快。问题是，这种情况很重要吗？

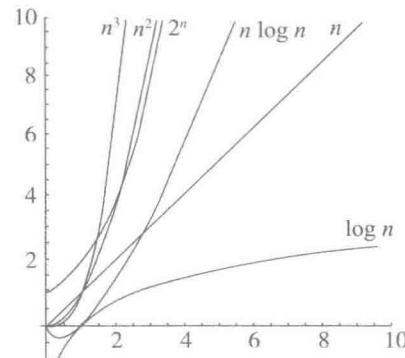


图 1.4 几个常见复杂度函数的增长情况

[⊖] 这里说“通常”，原因见前面有关算法处理问题实例具体情况的讨论。这句话的另一层面意思是，并不能保证对同样规模的每个实例， A_2 都比算法 A_1 更快。

看一个例子。假设解决某个具体问题的基本操作每秒钟可以完成 10^6 次，需要处理的问题实例的规模是 100。如果算法的时间复杂度是 $O(n)$ ，计算所需的时间可忽略不计（1/10000 秒的量级）；如果算法的时间复杂度是 $O(n^3)$ ，所需时间是 1 秒钟的量级；而如果算法的时间复杂度是 $O(2^n)$ ，求解问题所需时间将达到 10^{16} 年的量级（注意，迄今为止的宇宙寿命估计为 10^{10} 年的量级，可见上面的时间有多长）。如果计算机的速度提高 10000 倍，这个时间可以缩短到 10^{12} 年的量级，仍然是无法接受的。

可见，算法的复杂度反过来决定了算法的可用性。如果复杂度较低，算法就可能用于解决很大的实例，而复杂度很高的算法只能用于很小的实例，可用性很有限。算法的复杂度有重要的实际意义，决定了一些算法是否真正能用于实际（具体使用的机器是否提供了足够的存储，算法在实际要求的时间内能否完成）。例如：

- 做天气预报的程序，必须今天下午完成对明天的天气预报计算。如果不能按时计算出预报结果，这个算法就毫无价值。
- 数字相机的人脸识别程序，必须在几分之一秒内完成工作。过慢的算法会带来糟糕的用户体验，照相机的制造商不可能采用。对于这类问题，没有绝对的正误标准，如果找不到效率够高的好算法，可以考虑不那么准确但更快速的算法。

解决同一问题的不同算法

解决同一个问题可能存在不同算法。考虑一个简单问题：求斐波那契数列的第 n 项。根据斐波那契数列的数学定义，其第 n 项 F_n 定义如下：

$$\begin{aligned} F_0 &= F_1 = 1 \\ F_n &= F_{n-1} + F_{n-2}, \text{ 对于 } n > 1 \end{aligned}$$

根据这个定义可以直截了当地写出一个递归算法（用 Python 函数表示）：

```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

把参数 n 看作问题实例的规模，不难看出，计算 F_n 的时间代价（考虑求加法操作的次数）大致等于计算 F_{n-1} 和 F_{n-2} 的时间代价之和。这一情况说明，计算 F_n 的时间代价大致等比于斐波那契数 F_n 的值。根据已有的结论：

$$\lim_{n \rightarrow \infty} F_n = \left(\frac{\sqrt{5} + 1}{2} \right)^n$$

括号里的表达式约等于 1.618，所以计算 F_n 的时间代价按 n 值的指数增长。对于较大的 n ，这一计算就需要很长很长时间。

求斐波那契数还有另一个简单的递推算法：对 F_0 和 F_1 （如果 n 等于 0 或 1）直接给出结果 1；否则从 F_{k-1} 和 F_{k-2} 递推计算 F_k ，直至 k 等于 n 时就得到了 F_n 。

对应 Python 函数实现也很简单：

```
def fib(n):
    f1 = f2 = 1
    for k in range(1, n):
        f1, f2 = f2, f2 + f1
    return f2
```

用这个算法计算 F_n 的值，循环前的工作只做一次，循环需要做 $n-1$ 次。每次循环中只执行了几个简单动作，总的工作量（基本操作执行次数）与 n 值呈某种线性关系。

这个例子说明，解决同一问题的不同算法，其计算复杂度的差异可以很大，甚至具有截然不同的性质。通过分析算法（程序）复杂度，可以帮助使用者选择适用的算法；也可能发现已知算法的复杂性都太高，促使人们设法开发更好的算法。

当然，实际中遇到的情况未必像上面求斐波那契数列的两个算法这样黑白分明。在实际工作中也经常遇到一些情况，其中两个（或多个）可用算法各有所长，例如一个时间复杂度较低但空间复杂度较高，另一个的情况正好相反。在这种情况下，就需要做进一步的细致分析，权衡利弊，选择最适合实际情况的算法。

1.3.3 算法分析

算法分析的目的是推导出算法的复杂度，其中最主要技术是构造和求解递归方程，后面将简单介绍这方面的一些情况。由于本书中讨论的算法结构都比较简单，一般不需要高级的分析和推导技术。

基本循环程序

这里只考虑算法的时间复杂度，考虑最基本的循环程序，其中只有顺序组合、条件分支和循环结构。分析这种算法只需要几条基本计算规则：

0. 基本操作，认为其时间复杂度为 $O(1)$ 。如果是函数调用，应该将其时间复杂度代入，参与整体时间复杂度的计算。
1. 加法规则（顺序复合）。如果算法（或所考虑算法片段）是两个部分（或多个部分）的顺序复合，其复杂性是这两部分（或多部分）的复杂性之和。以两个部分为例：

$$T(n) = T_1(n) + T_2(n) = O(T_1(n)) + O(T_2(n)) = O(\max(T_1(n), T_2(n)))$$

其中 $T_1(n)$ 和 $T_2(n)$ 分别为顺序复合的两个部分的时间复杂度。由于忽略了常量因子，加法等价于求最大值，取 $T_1(n)$ 和 $T_2(n)$ 中复杂度较高的一个。

2. 乘法规则（循环结构）。如果算法（或所考虑的算法片段）是一个循环，循环体将执行 $T_1(n)$ 次，每次执行需要 $T_2(n)$ 时间，那么：

$$T(n) = T_1(n) \times T_2(n) = O(T_1(n)) \times O(T_2(n)) = O(T_1(n) \times T_2(n))$$

3. 取最大规则（分支结构）。如果算法（或所考虑算法片段）是条件分支，两个分支的时间复杂性分别为 $T_1(n)$ 和 $T_2(n)$ ，则有：

$$T(n) = O(\max(T_1(n), T_2(n)))$$

现在看一个简单实例。矩阵乘法由下面程序片段完成，它求出两个 $n \times n$ 矩阵 $m1$ 和 $m2$ 的乘积，存入另一个 $n \times n$ 矩阵 m 。假设矩阵在 Python 语言里实现为两层的表，两个参数矩阵已经有值，保存结果的 m 也已准备好，计算过程可描述为：

```
for i in range(n):
    for j in range(n):
        x = 0.0
        for k in range(n):
            x = x + m1[i][k] * m2[k][j]
        m[i][j] = x
```

这个程序片段是一个两重循环，循环体是一个顺序语句，其中还有一个内嵌的循环。根据上面的复杂性计算规则，可以做出下面推导：

$$\begin{aligned} T(n) &= O(n) \times O(n) \times (O(1) + O(n) \times O(1) + O(1)) \\ &= O(n) \times O(n) \times O(n) = O(n \times n \times n) = O(n^3) \end{aligned}$$

其中 $O(1) + O(n) \times O(1) + O(1)$ 部分表示循环体的复杂度，它包含两个基本语句和一个简单循环，化简后也得 $O(n)$ 。

再看另一个实例：求 n 阶方形矩阵的行列式的值。考虑两种不同算法。

高斯消元法：通过逐行消元，把原矩阵变换为一个上三角线矩阵，最后乘起所有对角线元素，就得到矩阵行列式的值。

算法如下：

```
# 设被求值矩阵为二维表A[0:n][0:n]
for i in range(n-1):
    用A[i][i]将A[i+1:n][i]的值都变为0
det = 0.0
for i in range(n):
    det += A[i][i]
```

最后求乘积的循环需要 $O(n)$ 时间。前一步的消元循环从 i 等于 0 做到 i 等于 $n-2$ ，对 i 迭代时需要做 $n-i-1$ 行的消元，在每行对 $n-i$ 个矩阵元素做乘法和减法运算，总的时间开销不超过 $O(n^2)$ ，因此算法的时间复杂度是 $O(n^3)$ 。

考虑另一个算法：直接基于矩阵行列式的定义。这个算法是递归的，为计算 n 阶方阵的行列式时需要算出 n 个 $n-1$ 阶的行列式，每计算一个 $n-1$ 阶行列式需要算出 $n-1$ 个 $n-2$ 阶的行列式……因此有下面推导：

$$\begin{aligned} T(n) &= n \times ((n-1)^2 + T(n-1)) > n \times T(n-1) \\ &> n \times (n-1) \times T(n-2) > O(n!) \end{aligned}$$

其中的平方项表示构造低阶矩阵的开销。复杂度 $O(n!)$ 比 $O(n^3)$ 增长得快得多，因此这个算法的复杂度极高，只能处理很小的矩阵，基本不实用。

递归算法的复杂度 *

这里简单介绍有关递归算法复杂度的理论结果，供读者参考。实际上，循环算法也可以看成简单的递归算法，但因为较简单，不用这里的结论大都可以处理。

递归算法通常具有如下的算法模式：

```
def recur(n):
    if n == 0:
        return g(...)
    somework
    for i in range(a):
        x = recur(n/b)
        somework
    somework
```

也就是说， n 值为 0 时直接得到结果，否则原问题将归结为 a 个规模为 n/b 的子问题，其中 a 和 b 是由具体问题决定的两个常量。另外，在本层递归中还需要做一些工作，上面描述里用 $somework$ 表示，其时间复杂度可能与 n 有关，设为 $O(n^k)$ 。这个 k 也应该是常量， $k=0$ 表示这部分工作与 n 无关。这样就得到了下面的递归方程：

$$T(n) = O(n^k) + a \times T(n/b)$$

有如下结论：

- 如果 $a > b^k$ ，那么 $T(n) = O(n^{\log_b a})$ 。

- 如果 $a=b^k$, 那么 $T(n)=O(n^k \log n)$ 。
- 如果 $a < b^k$, 那么 $T(n)=O(n^k)$ 。

这些结论可以涵盖很大一部分递归定义算法（程序）的情况。

注意, 因为这里要求 a 、 b 、 k 为常量, 由此有关结论不能处理前面行列式的递归计算, 在那里规模为 n 的问题归结为 n 个规模为 $n-1$ 的问题。

1.3.4 Python 程序的计算代价（复杂度）

本书将一直用 Python 语言编写程序。由于 Python 程序是算法的实现, 因此也可以而且经常需要考虑其复杂度问题。

时间开销

在考虑 Python 程序的时间开销时, 有一个问题特别需要注意: Python 程序中的很多基本操作不是常量时间的。

下面是一些情况:

- 基本算术运算是常量时间操作[⊖], 逻辑运算是常量时间运算。
- 组合对象的操作有些是常量时间的, 有些不是。例如:
 - 复制和切片操作通常需要线性时间 (与长度有关, 是 $O(n)$ 时间操作)。
 - `list` 和 `tuple` 的元素访问和元素赋值, 是常量时间的。
 - `dict` 操作的情况比较复杂, 后面第 8 章有详细讨论。
- 使用组合对象的程序, 需要特别考虑其中操作的复杂度。
- 字符串也应该看作组合对象, 其许多操作不是常量时间的。
- 创建对象也需要付出空间和时间, 空间和时间代价都与对象大小有关。对于组合对象, 这里可能有需要构造的一个个元素, 元素有大小问题, 整体看还有元素个数问题。通常应看作线性时间和线性空间操作 (以元素个数作为规模)。

在下面有关数据结构和算法的讨论中, 将会介绍和分析一些 Python 结构和操作的效率问题。这里首先列举一些具体情况, 更详细的情况见后面章节。

- 构造新结构, 如构造新的 `list`、`set` 等。构造新的空结构 (空表、空集合等) 是常量时间操作, 而构造一个包含 n 个元素的结构, 则至少需要 $O(n)$ 时间。统计说明, 分配长度为 n 个元素的存储块的时间代价是 $O(n)$ 。
- 一些 `list` 操作的效率: 表元素访问和元素修改是常量时间操作, 但一般的加入 / 删除元素操作 (即使只加入一个元素) 都是 $O(n)$ 时间操作。
- 字典 `dict` 操作的效率: 主要操作是加入新的关键码 - 值对和基于关键码查找关联值。它们的最坏情况复杂度是 $O(n)$, 但平均复杂度是 $O(1)$ 。这是非常有趣的现象, 也就是说, 一般而言字典操作的效率很高, 但偶然也会出现效率低的情况。

上面复杂度中的 n 都是有关结构中的元素个数。程序里经常用到这些操作, 它们的效率对程序效率有重大影响。另外, 有些操作的效率高。例如, 在表的最后加入和删除元素的操作效率高, 在其他地方加入和删除元素的效率低, 应该优先选用前者。

空间开销

在程序里使用任何类型的对象, 都需要付出空间的代价。建立一个表或者元组, 至少要占

[⊖] 其实这句话也不确切。浮点数算术运算应该认为是常量时间操作。Python 语言的 `int` 类型可以表示任意大小的整数值, 实际上是通过软件实现的。超过一定范围后, 整数越大, 一次算术运算耗费时间越长。

用元素个数那么多空间。如果一个表的元素个数与问题规模线性相关，建立它的空间付出至少为 $O(n)$ （如果元素也是新创建的，还需考虑元素本身的存储开销）。

相对而言，表和元组是比较简单的数据结构。集合和字典需要支持快速查询等操作，其结构更加复杂。包含 n 个元素的集合或字典，至少需要占用 $O(n)$ 的存储空间。

这里还有两个问题需要特别提出，请读者注意：

- 1) Python 的各种组合数据对象都没有预设的最大元素个数。在实际使用中，这些结构能根据元素个数的增长自动扩充存储空间。从空间占用的角度看，其实际开销在存续期间可能变大，但通常不会自动缩小（即使后来元素变得很少了）。举个例子，假设程序里建了一个表，而后不断加入元素导致表变得很大，而后又不断删除元素，后来表中元素变得很少，但占用的存储空间并不减少。
- 2) 还应该注意 Python 自动存储管理系统的影响。举个例子：如果在程序里建立了一个表，此后一直将其作为某个全局变量的值，这个对象就会始终存在并占用存储空间。如果将其作为某个函数里局部变量的值，或者虽然作为全局变量的值，但后来通过赋值将其抛弃，这个表对象就可以被回收。

总之，在估计 Python 程序的空间开销时，上面这些细节也需要考虑。

Python 语言提供了很多高级结构，使人很容易通过简短的程序完成很多工作，例如很容易构造出新的表或元组。这种方便性有时也会带来负面作用。懒散而且不警觉的编程者很容易写出一些貌似正确但实际上完全不能用的程序，其时间开销太大而且毫无必要，使程序只能处理很小的实例，基本上不能使用；或者空间开销太大，生成大量不必要的数据对象，直至用完了可用内存，导致程序崩溃。另一个显然的事实是，构造大量数据对象本身也需要时间，这两方面的问题往往是纠缠在一起的。

今后写程序要特别关注这些问题，应该关注程序的空间和时间开销。

Python 程序的时间复杂度实例

现在考虑一个很简单的例子。假设需要把得到的一系列数据存入一个表，其中得到一个数据是 $O(1)$ 常量时间操作。代码可以写为：

```
data=[]
while还有数据:
    x = 下一数据
    data.insert(0,x) #把新数据加在表的最前面
```

或者写为：

```
data=[]
while还有数据:
    x = 下一数据
    data.insert(len(data),x) # 新数据加在最后，或写data.append(x)
```

前一代码段把新元素加在已有元素之前，后一写法把新元素加在已有元素之后。两种写法都完成了所需工作，虽然产生的结果表里元素的顺序不同。现在的问题是，这两种写法的效率怎样？这个效率应该与表的结构和操作的实现方式有关。

显然这两个程序段的时间代价与循环的次数（表的最终长度）有关。但实际情况是：前一程序段需要 $O(n^2)$ 的时间才能完成工作，而后一个程序段只需要 $O(n)$ 时间。造成这种情况与 list 的实现方式有关，第 3 章将介绍有关情况。

作为另一个例子，考虑建立一个表，其中包含从 0 到 $10000 \times n - 1$ 的整数值，下面几个函

数都能完成这一工作：

```
def test1(n):
    lst=[]
    for i in range(n*10000):
        lst = lst + [i]
    return lst

def test2(n):
    lst=[]
    for i in range(n*10000):
        lst.append(i)
    return lst

def test3(n):
    return [i for i in range(n*10000)]

def test4(n):
    return list(range(n*10000))
```

还可能有许多其他做法也能完成这一工作。请读者自己做些试验，看看不同函数定义产生的计算代价随 n 的值而增长的趋势。

程序实现和效率陷阱

设计一个算法，通过对它的分析可能得到抽象算法的时间与空间复杂度。进而，采用某个编程语言（例如 Python）可以做出该算法的实现。这样就出现了一个问题：算法的实现（程序）的时间开销与原算法的时间复杂度之间有什么关系？

理想的情况应该是这样：作为算法的实现，相应程序的时间开销增长趋势应该达到原算法的时间复杂度。但是，如果实现做得不好，假设原算法的时间复杂度是 $O(n)$ ，其实现程序也可能比这差，例如达到 $O(n^2)$ 或者更差。这个讨论是想说明了一个问题，即在考虑程序的开发时，不但要选择好的算法，还要考虑如何做出算法的良好实现。

应该特别指出，用 Python 等高级语言编写程序存在一些“效率陷阱”，这种陷阱有可能使原本可以用计算机做的事情变得不可行（时间代价太大），或者至少也浪费了计算机和人的大量时间。在实际工作中，这种实现方式方面的（效率）缺陷很可能葬送掉一个软件，至少也会损害其可用性（降低其价值）。

上面的 Python 程序中就有这样的例子，例如函数 `test1`。最常见的错误做法就是毫无必要地构造一些可能很大的复杂结构。例如在递归定义的函数里的递归调用中构造复杂的结构（如 `list` 等），而后只使用其中的个别元素。从局部看，这样做使得常量时间的操作变成了线性时间操作。但从递归算法的全局看，这种做法经常会使多项式时间算法变成指数时间算法，也就是说，把原来有用的算法变成了基本无用的算法。

这种事例也说明了一个情况：Python 这样的编程系统提供了许多非常有用的功能，但是要想有效地使用它们，有必要了解数据结构的一些深入情况。本书的一个目标就是提供这方面的基本知识，帮助读者理解 Python 语言本身。

另外，有些读者也可能有非常兴趣知道 Python 这样强大的系统是如何构造起来的。本书也会提供一些这方面的基本信息。

1.4 数据结构

从程序输入和输出的角度看，用计算机解决问题，可以看作实现某种信息表示形式的转

换。如图 1.5 所示, 把以一种形式表示的信息(输入)送给程序, 通过在计算机上运行程序, 产生出以另一种形式表示的信息(输出)。如果:

- 具体的“信息表示 A”表达了需要求解的某个问题的实例。
- 得到的“信息表示 B”表达了与这个实例对应的求解结果。

那么就可以认为, 这个程序完成了该问题实例的求解工作。

为了能用计算机处理与问题有关的信息, 就必须采用某种方式表示它, 并将相应表示送入计算机。信息通过表示就变成了(计算机处理的)数据。与问题有关的信息可能很复杂, 不仅是可能数量庞大, 而且信息之间可能存在错综复杂的相互联系。为了能在计算中有效处理, 必须以适当的形式把蕴涵了这些信息的数据组织好。需要处理的信息的情况越复杂, 处理过程(计算过程)越复杂, 数据的良好组织就越重要。

1.4.1 数据结构及其分类

这里首先简单介绍几个概念, 而后介绍数据结构的一些情况。

信息、数据和数据结构

信息是目前在生活和工作中最经常听到的一个词汇, 人们都说今天已经进入信息时代。但是, 要想给信息这个概念一个容易理解的确切定义却不容易。直观地看, 任何存在着的事物, 其形态和运动都蕴涵着信息, 人的思想和行为也产生信息。人们希望用计算机处理的终极对象就是客观存在的各种信息, 因此说计算机是处理信息的工具。计算机本质上具有与人类发明的其他工具截然不同的性质。

显然, 要用计算机处理信息, 首先必须把信息转变成为计算机能处理的形式。这种转换可能由人完成, 也可能通过一些物理设备完成。例如, 数字照相机把一个客观场景转换为一幅数字化的图像; 数字麦克风把一段声音转换成某种编码的一串数字信号; 人通过敲击键盘把一段演讲记录下来变成一段标准编码的文本串; 通过光学扫描和文字识别, 也可以把纸张上的印刷文本转变为计算机可以处理的编码文本形式。

在计算机科学技术领域, 数据(data)就是指计算机(程序)能够处理的符号形式的总和, 或说是经过了编码的信息(信息的编码表示)。

在讨论计算机处理时, 经常要提到数据元素(data element), 用于指最基本的数据单位。在计算机硬件层面, 所有被存储和处理的数据最终都编码为二进制代码形式。一切数据最终都表现为二进制位的序列, 最基本的数据元素就是一个二进制位。但在计算机应用的各个层面上, 数据可能具有更加丰富多彩的表现形式, 这时说一个数据元素就是指在当前的上下文中作为整体保存和处理的一个数据单元。

实际中需要处理的通常都不是单个数据元素, 而是或多或少的一组数据元素。另一方面, 这些元素通常也不是独立的互不相关的个体, 它们相互之间经常有各种联系。一批数据元素和它们之间的联系一起, 反映了需要处理的问题的相关信息。

数据结构(data structure)研究数据之间的关联和组合的形式, 总结其中的规律性, 发掘特别值得注意的有用结构, 研究这些结构的性质, 进而研究如何在计算机里实现这些有用的数据结构, 以支持相应组合数据的高效使用, 支持处理它们的高效算法。在考虑数据结构时, 其数据元素作为原子性的单元, 可以任意简单或复杂, 没有任何限制。

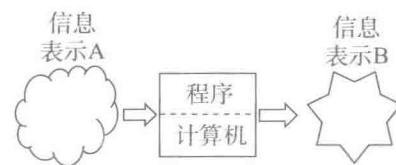


图 1.5 计算和数据表示的转换

抽象定义与重要类别

首先抽象地考虑数据结构的概念。从逻辑上看，一个数据结构包含一集数据元素，是一个有穷集，在这些元素之间有着某些特定的逻辑关系。按照这种观点抽象地看，一个具体的数据结构就是一个二元组

$$D = (E, R)$$

其中的 E 是数据结构 D 的元素集合，是某个数据集合 \mathcal{E} 的一个有穷子集，而 $R \in E \times E$ 是 D 的元素之间的某种关系。对于不同种类的数据结构，其元素之间的关系具有不同性质。

这个定义非常一般，对于关系 R 没有任何限制。人们对实际中常用的数据结构做了一些梳理，总结出一批特别有用的数据结构，主要有：

- 集合结构：其数据元素之间没有需要关注的明确关系，也就是说关系 R 是空集。这样的数据结构也就是其元素的集合，只是把一组数据元素包装为一个整体。这是最简单的一类数据结构。
- 序列结构：其数据元素之间有一种明确的先后关系（顺序关系）。存在一个排位在最前的元素，除了最后的元素外，每个元素都有一个唯一的后继元素，所有元素排列成一个线性序列。关系 R 就是这里的线性顺序关系。这种结构也称为线性结构，一个这样的复杂数据对象就是一个序列对象。序列结构还有一些变形，如环形结构和 ρ 形结构。其特点是每个元素只有（最多）一个后继，如图 1.6 所示。图中的小圆圈表示数据元素，箭头表示元素之间的关系，说明这种关系是有方向的。
- 层次结构：其数据元素分属于一些不同的层次，一个上层元素可以关联着一个或者多个下层元素，关系 R 形成一种明确的层次性，只从上层到下层（通常也允许跨层次）。层次关系又可以分为许多简单或复杂的子类别。
- 树形结构：层次结构中最简单的一种关系是树形关系，其特点是在一个树形结构中只有一个最上层数据元素，称为根，其余元素都是根的直接或间接关联的下层元素。进一步说，除根元素之外的每个元素，都有且仅有一个上层元素与之关联。树形数据结构简称为树，相应的复杂数据对象称为树形对象。第 6 章将详细讨论这类结构。树形结构的图示见 6.1 节和 6.7 节。
- 图结构：数据元素之间可以有任意复杂的相互联系。数学领域中的图概念是这类复杂结构的抽象，因此人们把这样的结构称为图结构，把这样的复杂对象称为图对象。第 7 章将详细讨论图结构。

实际上，可以认为图结构包含了前面几类结构，把那些结构看作图的受限形式。由于其中元素之间的关系受限，可能存在一些有意义的特殊性质。

算法和程序中的数据结构

抽象的数据结构可以从逻辑结构的角度分析和研究，但人们考虑更多的是用这些结构表现计算机需要处理的信息的特征、存储相关的信息及其内在联系。因此需要研究在面向实用时，各种数据结构表现出的性质和产生的问题。

用数据结构存储信息，不仅要考虑如何把抽象的数据结构映射到计算机或程序可以表达和

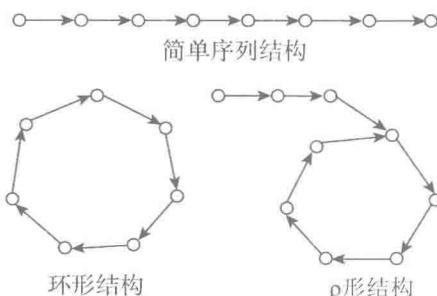


图 1.6 序列结构及其变形

操作的数据存储形式，还要考虑作用于具体数据结构的各种操作，如结构的建立、其中元素的访问、插入或删除元素等一般性操作。在实际应用中，经常还需要考虑一些面向具体应用问题的特殊操作。

数据结构上的操作需要通过算法实现。对复杂的数据结构，如树结构和图结构，存在许多非常有趣而且有用的算法。图算法已经发展成为算法领域的一个重要分支。在后面章节里讨论某种数据结构时，将介绍该结构上的一些重要算法。

在程序中使用一种数据结构，需要把这种结构映射到编程语言提供的基本数据机制，或者说是用编程语言的数据机制实现所需的数据结构。这一映射称为（抽象）数据结构的物理实现。编程语言系统（编译器或解释器）将把通过语言的数据机制定义的数据结构映射到计算机存储器中。前一层映射由设计和实现程序的人完成，良好的设计能为相应算法的实现提供更好的支持。后一层映射由编程语言的编译器或解释器完成，实际上是由设计和实现解释器 / 编译器的人们做出的选择。底层实现牵涉到有效利用计算机的存储器，上层实现牵涉到有效利用语言提供的各种数据机制。本章最后几小节将介绍这两方面的一些基本情况，第2章介绍抽象数据类型的思想和Python中与此相关的重要机制，后面各章在抽象地讨论某种数据结构后，都会讨论其Python实现问题。

结构性和功能性的数据结构

前面介绍了数据结构的概念，列举了一些计算中常用的数据结构，如线性结构、树结构和图结构。这些数据结构都对其数据元素之间的相互关系做出了一些规定，元素之间确实满足某种关系才能被称为线性结构或树结构等。也就是说，这些数据结构的最重要特征就是它们的结构，因此本书中将称它们为结构性的数据结构。

本书中还将讨论另一类数据结构，它们并没有对其元素的相互关系提出任何结构性的规定，而是要求实现某种计算中非常有用的功能。作为可以包含一批数据元素的结构，最基本的要求就是支持元素的存储和使用（使用也常称为元素访问）。这个基本要求实际上是功能性的要求，而非结构性的要求，因为它完全不涉及元素如何存储、元素之间如何关联。支持元素存储和访问的数据结构被称为容器。人们已经开发出许多不同的容器数据结构，它们各有一些功能方面的特点，是本书讨论的另一部分数据结构。

在下面讨论中，涉及的功能性数据结构包括栈、队列、优先队列、字典等。这些结构都支持以某一套方式存储和访问元素（包括删除元素），但不同结构提供了不同的元素访问特性，其具体操作表现出各自的特性，这些特性在抽象算法和实际程序里都非常重要。这些功能性数据结构的使用非常广泛。

由于只有功能要求，这类数据结构可以采用任何技术实现。实际中人们通常首先把这类结构映射到某种结构性的数据结构，而后采用相应的实现技术。有时也开发一些专门的实现技术。对这类数据结构的讨论主要在第5章和第8章。

本节剩下部分将集中关注数据结构的实现问题。下面将介绍计算机存储器的基本情况、一般编程语言中数据存储和组织的情况，进而简单综述Python语言的这方面功能。了解这些情况，有助于理解后面各章的内容。

1.4.2 计算机内存对象表示

为了理解数据结构的性质，理解数据结构实现和处理中的基本问题，需要对计算机内存结构和计算机存储管理方面的情况有所了解。本小节介绍这方面情况。

内存单元和地址

计算机（程序中）直接使用的数据保存在计算机的内存存储器（简称内存）。内存是CPU可以直接访问的数据存储设备。与之相对应的是外存储器，简称外存，如磁盘、光盘、磁带等。保存在外存里的数据必须先装入内存，而后CPU才能使用它们。

内存的基本结构是线性排列的一批存储单元。每个单元的大小相同，可以保存一个单位大小的数据。具体单元大小可能因计算机的不同而有所不同。在目前最常见的计算机中，一个单元可以保存一个字节（8位二进制代码）的数据。因此存放一个整数或者浮点数，需要连续的几个单元。例如标准的浮点数需要8个单元。

内存单元具有唯一编号，称为单元地址，或简称地址。单元地址从0开始连续排列，全部可用地址为从0开始的一个连续的正整数区间，如图1.7所示。

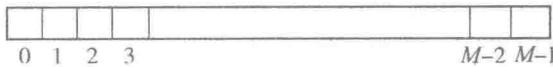


图1.7 计算机内存的基本结构

在程序执行中，对内存单元的访问（存取其中数据）都通过单元的地址进行，因此，要访问一个单元，必须先掌握其地址。在许多计算机中，一次内存访问可以存取若干单元的内容。例如目前常见的64位计算机，一次可以存取8个字节的数据，也就是说一次操作访问8个单元的内容。基于地址访问内存单元是一个O(1)操作，与单元的位置或整个内存的大小无关，这是分析与数据结构有关的算法时的一个基本假设[⊖]。在高级语言层面讨论和分析数据结构问题时，人们通常不关心具体的单元大小或地址范围，只假定所考虑数据保存在内存的某处，而且假定这种访问是常量时间的。

对象存储和管理

在程序运行中，可能需要构造、使用、处理各种各样的对象，它们都将在计算机的线性结构的内存里安排位置。为了表示程序中的一个对象，需要根据情况，在当时空闲的内存中确定一块或几块区域（内存区域指地址连续排列的一个或一些内存单元），把该对象的数据存入其中。在Python程序运行中，建立对象时需要安排存储，还有许多与对象存储和使用有关的管理工作。解释器的一个专门子系统（称为存储管理系统）负责这些工作。这一工作是自动进行的，编写程序的人不必关心。另外，当一个对象不再有用时，存储管理系统也会设法回收其占用的存储，以便在将来用于存储其他对象。

一个程序在运行中将不断建立一些对象并使用它们。建立的每个对象都有一个确定的唯一标识，用于识别和使用这个对象。在一个对象的存续期间，其标识保持不变，这也是一個基本原则。例如，Python标准函数id取得对象的标识，内置操作is和is not通过比较标识的方式判断是否为同一个对象。在具体系统里用什么作为对象标识，是系统设计者的考虑和选择。最简单的方式就是直接使用对象的存储位置（内存地址）。这显然是一种唯一标识，因为不会有两个不同对象存放在同一个存储位置。

对象的访问

在编程语言层面，知道了一个对象的标识就可以直接访问（使用）它。已知对象标识（无

⊖ 满足这样假设的存储器称为随机访问存储器（RAM，与只能较好支持顺序访问的磁盘和磁带等对应），其特点就是无论以怎样的随机顺序访问其中数据，每个访问所需的时间都是常量的。实际上，这一假设也是一种抽象。为了提高操作速度，新型计算机CPU都采用了多级缓存结构和复杂的数据调度算法。造成的结果是连续访问一批单元的速度远高于在较大内存区域里同样次数的随机访问。这一情况使数据结构的操作效率呈现出更复杂的现象。由于这些情况，基于随机访问假设的算法和程序分析与实际程序运行情况有一定偏差。但是，这种抽象仍有很好的指导意义。

论它是否直接为对象地址), 访问相应对象的操作可以直接映射到已知地址访问内存单元, 这种操作可以在常量时间完成(是O(1)时间操作)。

如果被访问的是一个组合对象, 其中包含了一组元素, 这些元素被安排在一块内存区域(一块连续的元素存储区)里, 而且每个元素的存储量相同。在这种情况下, 可以给每个元素一个顺序编号(通常称为下标, index)。对这种组合对象, 程序中经常需要访问它们的元素。如果知道了一个组合对象的元素存储区位置, 又知道要访问的元素的编号, 访问元素也是O(1)时间操作。这件事很容易证明: 假设所关心的元素存储区的起始位置是内存地址 p , 每个元素占用 a 个内存单元。再假设元素序列中首元素的下标为0。要访问下标为 k 的元素, 通过下式能立刻计算出该元素的位置 $\text{loc}(k)$:

$$\text{loc}(k)=p+k \times a$$

有了元素位置就能直接访问该元素了。上述公式是统一的, 只需做一次乘法和一次加法, 所用时间与元素的下标无关, 与组合对象中的元素个数也无关。

考虑另一种情况, 假设某类组合对象都包含同样的一组元素, 元素连续存放在一块存储区里, 但不同元素的大小可能不同。进而, 在同属这类的不同对象里, 排在同样位置的元素的大小相同。进一步说, 这类对象都采用同样存储方式。显然, 在上面条件下, 根据元素的排列顺序, 可以事先算出这类对象里各元素在对象存储区中的相对位置, 称为元素的存储偏移量。假设一个对象 o 的地址为 p , o 的元素 e_i 的偏移量为 r_i , 立即可以算出 e_i 的地址为 $p+r_i$, 有关情况参见图1.8(显然, e_0 的偏移量为0)。易见, 在这种情况下, 已知一个这种组合对象, 访问其中元素的操作也可以在O(1)时间完成。

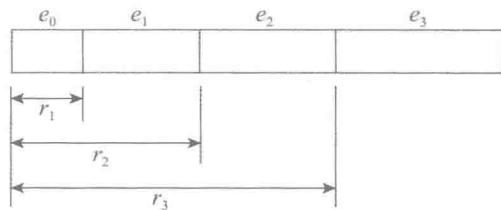


图1.8 对象存储、元素和偏移量

上面两种情况概括了数据结构中组合对象成分存储和访问操作的基本情况, 解释了对象访问和对象元素访问的时间复杂度问题。在下面有关数据结构的讨论中, 这两种操作很重要, 也是对与数据结构有关的算法做复杂度分析的基础。

对象关联的表示

基本类型数据(如字符、整数和浮点数)的内存表示方式由编程语言和计算机硬件确定, 一项数据占据确定数目的几个连续单元, 例如一个浮点数通常占连续的8个字节单元。

复杂数据对象(数据结构)包含了一批元素(数据成员), 这些元素本身也是数据对象, 可能是基本数据, 也可能是具有较简单内部结构的复杂对象, 元素间通常有某些联系。要表示这样的复杂结构, 就需要表示两方面的信息: 数据元素和元素之间的联系。具体元素的情况由对象的实际情况确定, 元素之间的联系以及如何表示这些联系则是一般性的问题, 是数据结构研究和实践中关注的主要问题。

在计算机内存里表示数据元素之间的联系, 只有两种基本技术:

- 1) 利用数据元素的存储位置隐式表示。由于内存是单元的线性序列, 知道了前一个元素的位置及其大小(存储占用量), 就能确定下一元素的位置。如果存储的是一系列大小相同的元素, 就可以利用前面公式直接算出序列中任何一个元素的位置。显然, 序列数据类型中元素的线性关系可以用这种方式表示。
- 2) 把数据元素之间联系也看作一种数据, 显式地保存在内存中。用这种方式可以表示数据元素之间任意复杂的关系, 因此这种技术的功能更强大。

第一种方式称为元素的顺序表示：在一个内存存储块里保存复杂结构中的多个数据元素，整个结构的整体也是一个数据对象，可以统一地安排和管理。

这种方式的最典型例子之一是简单字符串。假设每个字符用一个字节存储，为了创建一个字符串，首先为需要存储的字符确定一块足够大的内存块，然后把这个字符串的内容复制进去，得到的情况如图 1.9a 所示。图中箭头表示这块内存的起始位置（即是该字符串的标识），由它出发可以找到并使用该字符串的内容，可以把它赋给变量或传给函数。

但是，实际上上述做法通常还不够。内存单元里存储的都是二进制编码，对于字符串而言，存储的就是各字符的编码。由于字符串长短不一，仅从一串单元里保存的内容无法判断这一字符串到哪里结束。为了解决这个问题，通常需要约定一种存储安排。对于字符串，一种可能的方式是在其存储块的开始，用一个确定大小的部分记录该字符串的实际长度，随后部分保存字符串的实际内容。如图 1.9b 所示。

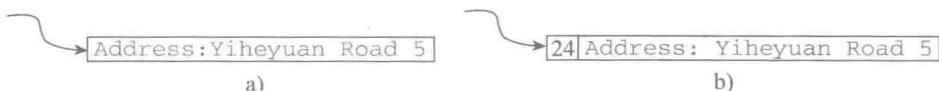


图 1.9 字符串对象的表示

这一做法结合了两个层次的表示：在上层看这是两项数据，一个整数表示字符串长度，紧接它的一块内存区域里保存字符串的内容；后一区域又是一系列单元，其中存储着字符串的各个字符。由于整数占用的内存单元数已知，根据前面讨论，知道了整个存储块的开始位置（对象标识），只需常量时间就能找到字符串里的任何一个字符。

这个例子说明，程序里使用的任何一种对象，即使简单如字符串，也需要设计一种合适的内存表示方式。要在程序中生成和处理任何对象，都需要设计好它们的存储方式。这种方式及其效果称为该对象的表示（representation）。图 1.9 及其讨论就说明了如何为字符串设计一种存储表示方式。如果需要自己从底层出发实现一类数据对象，第一件事就是为其确定一种表示方式，在 C 语言里写程序时经常需要做这种工作。在 Python 语言里，一些常用组合对象（包括字符串）已经有了现成的实现。由于其他人（Python 系统开发者）已经做了这部分工作，用 Python 编程序的人可以坐享其成。

字符串的结构比较简单，可以用一块连续存储区表示，类似情况如数学里的 n 维向量，可以表示为 n 个浮点数的序列。此外，数学中的矩阵是数据的二维阵列，元素类型相同，也可以考虑采用连续存储的表示方式。但是，并非所有对象都如此。

假设现在需要为记录书籍的信息设计一种对象。有关书籍的信息包含两个成分：作者和书名。表示它的对象也应该有两个成员，而且这两个成员都是字符串。对不同的书籍，这两个成员字符串的长度显然可能不同。虽然可以把两个字符串存放在一起，采用某种特殊分隔符做成一个字符串，但那种表示使用不方便，也影响操作效率。

一种合理的表示方式如图 1.10 所示，采用三个独立部分的组合表示一个完整的书籍对象。首先是一个二元结构，它表示书籍对象的整体，另外两个独立的字符串分别表示书籍对象的两个成分。在二元结构里记录两个成分字符串的引用信息（记录两个字符串对象的标识），这就是前面说的表示数据元素之间联系的第二种技术。这样，掌握了这个二元结构，通过其中记录的数据关联就可以找到有关

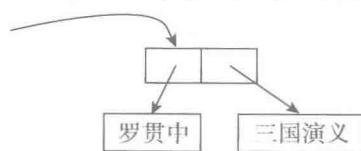


图 1.10 一种书籍对象的表示

书籍的所有信息了。

总结一下，这种表示技术是用一组独立的存储块（对象）表示一个复杂的数据对象，数据对象的成员用独立的成员对象表示，在一些块里记录与之相关的对象的关联信息（标识）。这种技术称为对象的链接表示技术，通过链接方式形成的复杂结构称为链接结构。链接技术经常被用来表示复杂的组合对象。图示中常用箭头表示对象间的关联，本书后面将一直这样做。关联是单向的，也称为链接或引用。

实际上，图 1.10 中出现的结构不仅仅是链接结构，作为书籍对象主体的二元结构本身又是一个连续表示的结构，这个结构包含了两个成分，其中记录了两个链接，它们的顺序位置确定了到哪里寻找作者信息、到哪里寻找书名。两个字符串对象都采用连续表示，整个对象又是由三个独立部分通过链接构成的。实际上，复杂数据对象的表示经常是连续结构和链接结构的组合，需要很好地结合这两种技术。

连续结构和链接结构是所有数据结构构造的基础。在后面章节里，读者将会看到如何基于这两种构造方式组合出一批最重要的数据结构。

1.4.3 Python 对象和数据结构

本书用 Python 作为编程语言讨论数据结构和算法的问题，这里简单概述 Python 语言中与数据表示有关的一些基本情况。在后面章节的讨论中，还将重点介绍 Python 的一些重要内置数据结构的实现和性质，以帮助读者更好地理解 Python 语言本身，并能在写程序时更好地使用它们，更好地思考（估计）其可能行为。

Python 变量和对象

高级语言里的变量（全局变量、函数的局部变量和参数）是内存及其地址的抽象。变量本身也需要在内存中安排位置，每个变量占用若干存储单元。语言系统需要有一套系统化的安排方式处理这个问题，下面讨论中不考虑。为了理解程序的行为，只需要假定在程序运行中总能找到根据作用域可见的变量，取得或修改它们的值。

在 Python 程序里，可以通过初始化（或提供实参）给变量（或函数参数）约束一个值，还可以通过赋值修改变量的值。这里的值就是对象，给变量约束一个对象，就是把该对象的标识（内存位置）存入该变量。图 1.11 形象地表示了几个变量及其约束值的情况，其中右边云状图形表示内存，几个对象有各自的存储位置，箭头表示变量与其值的约束关系。图 1.11 中只画了几个简单类型的对象，复杂组合对象的情况也类似。从变量出发访问其值是常量时间操作，这是在 Python 里分析程序的时间代价的基础。

Python 变量的值都是对象，可以是基本整数、浮点数等类型的对象，也可以是组合类型的对象，如 list 等。程序中建立和使用的各种复杂对象，包括 Python 函数等，都基于独立的存储块实现，通过链接相互关联。程序里的名字（变量、参数、函数名等）关联着作为其值的对象，这种关联可以用赋值操作改变。

Python 语言中变量的这种实现方式称为变量的引用语义，在变量里保存值（对象）的引用。采用这种方式，变量所需的存储空间大小一致，因为其中只需要保存一个引用。有些语言

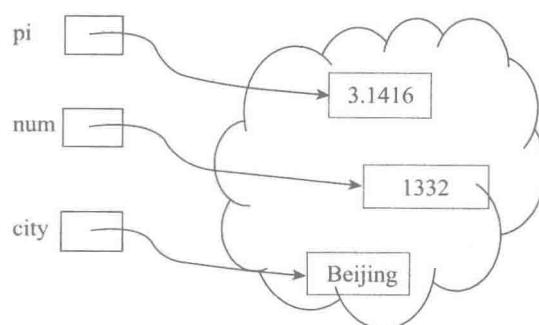


图 1.11 变量和对象约束

采用的不是这种方式，它们把变量的值直接保存在变量的存储区里，称为值语义。这样，一个整数类型的变量就需要保存一个整数所需的空间，一个浮点数变量就需要足够的空间存储一个浮点数。如果一个变量中需要保存很大的数据对象，它就需要占据更大的存储空间。例如 C 语言采用的就是变量的值语义。

Python 对象的表示

虽然在写 Python 程序时可以不关心各种对象的具体表示方式。但对这方面情况有所了解，可以帮助人们更清晰地理解程序的行为，特别是与执行效率有关的行为。基于前面的讨论，现在对 Python 的对象表示做一个概貌性的介绍。后面章节里讨论具体的数据结构时，还会介绍 Python 中一些结构的表示细节。

Python 语言的实现基于一套精心设计的链接结构。变量与其值对象的关联通过链接的方式实现，对象之间的联系同样也通过链接。一个复杂对象内部也可能包含几个子部分，相互之间通过链接建立联系。例如，如果一个 list 里包含了 10 个字符串，那么在实现中，在这个 list 对象里就会记录这 10 个字符串的链接关系（参看图 1.10）。

Python 里的组合对象可以具有任意大的规模（例如，list/tuple 都可以包含任意多个元素），每个对象需要的存储单元数可能不同（显然，不同的 list 有长有短），还可能有复杂的内部结构（也就是说，其元素可能又是复杂的数据对象）。在这类复杂对象的创建和使用中，存储安排和管理是比较麻烦的事情。好在 Python 程序内部有一个存储管理系统，负责管理可用内存，为各种对象安排存储，支持灵活有效的内存使用。程序中要求建立对象时，管理系统就会为其安排存储；某些对象不再有用时则回收其占用的存储。存储管理系统屏蔽了具体内容使用的细节，大大减少了编程人员的负担。

实际上，各种数据对象的具体表示方式，将对相关对象的各种操作的效率产生直接影响，间接影响着用 Python 开发的程序的效率。显然，Python 语言系统的实现者必定认真考虑了各种需要，选择了最合适的实现方式。但是由于实际中的需要千差万别，相互依赖，有些还相互冲突。在这种情况下，语言实现者只能权衡利弊，采用某些折中方案，这就导致某些使用方式更加高效，而另一些方式则较差。了解这方面的一些情况，可以帮助人们更有效地使用 Python 语言，写出更好的、执行效率更高的 Python 程序。

一般而言，在使用较低级的编程语言（例如 C 语言）工作时，人们可以更多地根据需要，自己设计各种数据结构的表示（实现）方法，设法得到较好的效果。在很多常规的数据结构课程和书籍中特别关注了这方面的问题。当然，这些设计和实现工作的细节繁多，编程相当复杂而琐碎，需要花费很多时间和精力。

在使用 Python 等高级编程系统进行程序开发工作，特别是做与复杂数据结构有关工作时，情况很不一样。由于 Python 语言提供了很多高级结构，编程中用起来很方便，大大减轻了开发的负担。但是，如不加注意地随意写程序，也很容易做出效率很差，甚至根本无法实际使用的程序。要避免这种情况，在用 Python 写程序时，就需要关注两方面的问题。一方面，人们也需要自己设计一些数据结构，这时需要考虑实现的效率问题，需要对构造的数据结构的基本技术有所了解。第 2 章将介绍所用的一种重要技术。另一方面，使用语言本身提供的各种高级结构，例如 Python 的 list 和 dict 等，也需要对这些结构的基本性质，以及实现它们的基本原理有准确理解，才能正确有效地使用它们。对于用 Python 开发较复杂的程序而言，上面所说的两方面理解都非常重要。随着本书中对各种数据结构问题的讨论，也会介绍一些与 Python 中具体数据类型有关的情况，帮助读者在这些方面建立起正确的认识。

Python 的几个标准数据类型

Python 语言提供了丰富的标准数据类型，每个类型都提供了大量相关操作。本书中实际使用的类型比较有限，使用的操作也不多。另外，为定义各种数据结构，本书中广泛使用了 Python 的面向对象机制，使用方式比较规范，第 2 章将详细介绍有关情况。下面简单概述一下本书中使用的 Python 组合类型。

`list`（表）是书中使用最多的组合数据类型。`list` 对象可以包含任意多个任意类型的元素，元素访问和修改都是常量时间操作。此外，`list` 对象是可变对象，在对象的存在期间可以任意地加入或删除元素。因此，程序中经常需要从空表开始，通过逐步加入元素的方法构造任意大的表。在后面一些数据结构的实现中，也使用这种技术。

`tuple`（元组）在保存元素和元素访问方面的性质与表类似，但其对象是不变对象，只能在创建时一下子构造出来，不能逐步构造。因此在本书中使用较少。

`dict`（字典）支持基于关键码的数据存储和检索，这里的关键码只能是不变对象（例如，可以是元组、字符串，但不能是表）。如果关键码是组合对象，其元素仍然必须是不变对象。在一个字典里可以容纳任意多的关键码 / 值关联，支持高效检索（平均时间为 $O(1)$ ）。后面讨论字典数据结构时，将说明怎样实现这种奇妙的对象。

其余组合数据类型在本书中几乎未使用，基本类型的情况很简单，不再赘述。

练习

一般练习

1. 设法证明求平方根的牛顿迭代法一定收敛。
2. 修改红绿灯安排实例中的算法，使之最后给出的各个分组最大，可以考虑从本章算法给出的分组出发，给每个分组加入不冲突的所有行驶方向。
3. 请完整写出采用高斯消元法求矩阵行列式的算法，并仔细分析其时间复杂度。
4. 解决同一问题有两个算法，一个算法的时间开销约为 $100n^3$ ，另一个约为 0.5×2^n 。问题规模为多大的情况下后一算法更快？
5. 假设现在需要对全世界的人口做一些统计工作，希望在 1 天之内完成工作。如果采用的算法具有线性复杂度，多快的计算机就足以满足工作需求？如果所用算法具有平方复杂度呢？用天河二号超级计算机大约能处理什么复杂度的算法？
6. 如果在宇宙大爆炸的那一刻启动了一台每秒十亿次的计算机执行斐波那契数列的递归算法，假设每条指令计算一次加法。到今天为止它可以算出第几个斐波那契数？这个数的数值应该是多少？
7. 假设你从 3 岁开始手工操作计算器每秒可以完成 3 次加法或乘法。执行求斐波那契数的递归算法，到今天为止你可能算出哪个斐波那契数？假设执行本章求矩阵乘积的算法，到今天为止能做出两个多大的矩阵的乘积？
8. 根据本章的讨论和自己的认识，设法对数据结构和算法的关系做一个综述和总结，并给出自己的认识，讨论其中的问题。

编程练习

1. 请回忆（查阅）基础数学中求平方根的方法。请定义一个 Python 函数，其中采用基础数学中

的方法求平方根。从各方面比较这个函数和基于牛顿迭代法的函数。

2. 基于 1.3.4 节中构造整数表的几个函数做一些试验，对一组参数值统计它们的执行时间。修改这些函数，重复试验，对 Python 中构造表的各种方法做一个总结。
3. 用 Python 字典表示冲突图结构，以关键码 (A, B) 关联的值为 True 表示冲突，没有值表示不冲突。用这种技术完成本章求无冲突分组的程序。
4. 基于前一题的类似设计，设法编写一个程序，使之能枚举出一个冲突图上的各种分组组合，从中找出最佳的无冲突分组。

第2章 抽象数据类型和 Python 类

在讨论具体的数据结构概念和技术之前，本章将首先介绍抽象数据类型的重要概念和 Python 面向对象的程序设计技术。后者可以看作一种实现抽象数据类型的技术，但还有所扩充，它也是本书中实现各种数据结构时使用的基本技术。

2.1 抽象数据类型

抽象数据类型（Abstract Data Type, ADT）是计算机领域中被广泛接受的一种思想和方法，也是一种用于设计和实现程序模块的有效技术。ADT 的基本思想是抽象，或者说是数据抽象（与函数定义实现的计算抽象或称过程抽象对应）。

按照抽象的思想，设计者在考虑一个程序部件时，应该首先给出一个清晰边界，通过一套接口描述说明这一程序部分的可用功能，但并不限制功能的实现方法。从使用者的角度看，一个程序部件实现了一种功能，如果适合实际需要，就可以通过其接口使用之，并不需要知道其实现的具体细节。Python 的函数就是一种功能部件，其头部定义了它的接口，描述了函数的名字及其对参数的要求。使用者只需要考虑函数的功能是否满足实际需要，还要保证调用式符合函数头部的要求，并不需要知道函数实现的任何具体细节。

在程序开发实践中人们逐渐认识到，仅有计算层面的抽象机制和抽象定义还不够，还需要考虑数据层面的抽象。能围绕一类数据建立程序组件，将该类数据的具体表示和相关操作的实现包装成一个整体，也是组织复杂程序的一种有效技术，可以用于开发出各种有用的程序模块。要把这种围绕着一类数据对象构造的模块做成数据抽象，同样需要区分模块的接口和实现。模块接口提供使用它提供的功能所需的所有信息，但不涉及具体实现细节。另一方面，模块实现者则要通过模块内部的一套数据定义和函数（过程）定义，实现模块接口的所有功能，从形式上和实际效果上满足模块接口的要求。

2.1.1 数据类型和数据构造

类型（数据类型）是程序设计领域最重要的基本概念之一。在程序里描述的、通过计算机去处理的数据，通常都分属不同的类型，例如整数或浮点数等。每个类型包含一集合法的数据对象，并规定了对这些对象的合法操作。各种编程语言都有类型的概念，每种语言都提供了一组内置数据类型，为每个内置类型提供了一批操作。

以 Python 为例，它提供的基本类型包括逻辑类型 `bool`、数值类型 `int` 和 `float` 等、字符串类型 `str`，还有一些组合数据类型。类型 `bool` 包括两个值（两个对象）`True` 和 `False`，可用操作包括 `and`、`or` 和 `not`；数值类型 `int` 包含很多值（整数对象），对它们可做加减乘除等运算；其他类型的情况相仿。开发程序时，应该根据需要选择合适的数据类型。

但是，无论编程语言提供了多少内置类型，在处理较为复杂的问题时，程序员或早或晚都会遇到一些情况，此时各种内置类型都不能满足或者不适合于自己的需要。在这种情况下，编程语言提供的组合类型有可能帮助解决一些问题。例如，Python 为数据的组合提供了 `list`、`tuple`、`set`、`dict` 等结构（它们也看作是类型），编程时可以利用它们把一组相关数据组织在一起，

构成一个数据对象，作为一个整体存储、传递和处理。

举个例子，假设程序里需要处理有理数。最简单朴素的想法是用两个整数表示一个有理数，分别表示其分子和分母，在此基础上实现所需要的运算和操作。在这种安排下，把有理数 $\frac{3}{5}$ 存入变量可能写成：

```
a1 = 3  
b1 = 5
```

利用 Python 函数可返回多对象元组和多项赋值的机制，加法函数可以如下定义：

```
def rational_plus(a1, b1, a2, b2):  
    num = a1*b2 + b1*a2  
    den = b1*b2  
    return num, den
```

下面是一个简单的函数使用实例：

```
a2, b2 = rational_plus(a1, b1, 7, 10)
```

不难想到，如果真这样写程序，很快就会遇到非常麻烦的管理问题：编程者需要时时记住哪两个变量记录的是一个有理数的分子和分母，操作时不能混淆不同的有理数；如果需要换一个有理数参与运算，也会遇到成对变量名的代换问题。程序比较复杂时，做这类事情很容易出错，而如果真发现程序有错误，确定错误的位置和更正也将极其费时费力。总而言之，用独立的分别存在的两个整数表示一个有理数，这种技术完全不可取。

一种简单改进是利用编程语言的数据组合机制，把相关的多项简单数据组合在一起。还是看有理数的例子，可以考虑用一个 Python 元组 (tuple) 表示一个有理数，约定用其中的第 0 项表示分子，用第 1 项表示分母。这样就可以写：

```
r1 = (3, 5)  
r2 = (7, 10)  
  
def rational_plus(r1, r2):  
    num = r1[0]*r2[1] + r2[0]*r1[1]  
    den = r1[1]*r2[1]  
    return num, den  
  
r3 = rational_plus(r1, r2)
```

现在情况显然好了很多，许多管理问题得到缓解。这就是数据构造和组织的作用。

但是，如果进一步考虑，就会发现这种做法仍然有多方面的缺陷，例如：

- 1) 在这里使用的不是特殊的“有理数”，而是普通的元组，因此不能将其与其他元组相互区分。例如，假设程序里还需要处理平面上的整数格点数据，格点也可能用整数的二元组表示，例如，(3, 5) 表示平面上 X 坐标为 3 而 Y 坐标为 5 的点。从概念上说，把一个有理数与一个格点相加完全是荒谬的事情。但 Python 编程语言，包括上面定义的 rational_plus 函数都不会认为这样做是一个错误。
- 2) 与有理数相关的操作并没有绑定于表示有理数的二元组。由于 Python 不需要说明函数参数的类型，这个问题表现得更加严重。
- 3) 在为有理数对象定义运算（函数）时，需要直接按位置取得其元素。对有理数这样结构很简单的对象，操作中只需区分位置 0 和 1，还算比较容易处理，给思维带来的负担也还可以忍受。如果需要处理的数据对象更复杂，例如其中包含了十几甚至几十个

不同成员。在为这种组合数据对象定义操作时，记住每个成员在对象里的位置并正确使用，也会变成非常麻烦的事情。修改数据表示就更让人头痛了。

以上几点和其他一些情况都说明，简单地使用语言提供的数据组合机制，对于处理复杂程序里的数据组织问题是不够的。上面前两点表明的主要问题是，按照这种方式构造和使用的“有理数”不是一个类型，因此不能得到 Python 语言里的类型功能（如类型检查）的支持。第 3 点说明，简单地用元组等表示内部结构复杂的数据，经常会导致程序不易阅读和理解，使程序难以编写正确，难以修改。抽象数据类型的思想和支持这种思想的编程语言机制能帮助解决这些问题，至少是使问题大大缓解。

2.1.2 抽象数据类型的概念

造成前一节中揭示出的编程缺陷，最重要的问题之一是数据的表示完全暴露，以及对象使用和操作实现对具体表示的依赖性。要克服这些缺点，就需要把对象的使用与其具体实现隔离开。理想情况是：在编程中使用一种对象时，只需考虑应该如何使用，不需要（最好是根本不能）去关注和触及对象的内部表示。这样的数据对象就是一种抽象数据单元，一组这样的对象构成一个抽象的数据类型，为程序里的使用提供了一套功能。

抽象数据类型的基本想法是把数据定义为抽象的对象集合，只为它们定义可用的合法操作，并不暴露其内部实现的具体细节，不论是其数据的表示细节还是操作的实现细节。当然，要使用一种对象，首先需要能构造这种对象，而后能操作它们。抽象数据类型提供的操作应该满足这些要求。一个数据类型的操作通常可以分为三类：

- 1) **构造操作**：这类操作基于一些已知信息，产生出这种类型的一个新对象。例如，基于一对整数产生出一个有理数对象；或者基于两个已有的有理数对象，产生出一个表示它们之和的有理数对象；等等。
- 2) **解析操作**：这种操作从一个对象取得有用的信息，其结果反映了被操作对象的某方面特性，但结果并不是本类型的对象。例如，可能需要有两个操作，分别从一个有理数获取其分子或者分母，操作的结果应该是整数（整数类型的对象）。
- 3) **变动操作**：这类操作修改被操作对象的内部状态。例如，对于一个银行账户对象，其类型就应该提供检查余额和修改余额的操作等。经过一次变动操作之后，对象还是原来的账户对象，仍然表示原来的银行客户的有关信息，但是对象内部记录的存款余额改变了，反映了实际客户账户的余额变动。

当然，一个抽象数据类型还应该有一个名字，用于代表这个类型。

其实，编程语言的一个内置类型就可以看作是一个抽象数据类型。Python 的字符串类型 str 是一个典型实例：字符串对象有一种内部表示形式（无须对外宣布），人们用 Python 编程序时并不依赖于实际表示（甚至不知道其具体表示方式）；str 提供了一组操作供编程使用，每个操作都有明确的抽象意义，不依赖于内部的具体实现技术。易见，Python 的整数类型 int 和实数类型 float 等的情况与 str 类似。当然，对于内置类型，语言有可能为它们提供一些额外的方便。如 Python 为字符串提供了文字量书写方式，可以看作简化的构造操作。从其他角度看，内置类型也就是一种抽象数据类型。

作为数据类型，特别是比较复杂的数据类型，有一个很重要的性质称为变动性，表示该类型的对象在创建之后是否允许变化。如果某个类型只提供上面的第 1 和第 2 类操作，那么该类型的对象在创建之后就不会变化，永远处于一个固定的状态。这样的类型称为不变数据类型，

这种类型的对象称为不变对象。对于这种类型，在程序里只能（基于其他信息或已有对象）构造新对象或者取得已有对象的特性，不能修改已建立的对象。如果一个类型提供了第 3 类操作，对该类型的对象执行这种操作后，虽然对象依旧，但其内部状态已经改变。这样的类型就称为可变数据类型，其对象称为可变对象。下面经常把不变数据类型和可变数据类型分别简称为什么类型和可变类型。

例如，Python 语言对 str 类型只提供了前两类操作，因此 str 是一个不变数据类型；对 list 类型提供了所有三类操作，它是一个可变数据类型。Python 语言里的 str、tuple 和 frozenset 是不变数据类型，而 list、set 和 dict 是可变数据类型。在编程中设计或定义抽象数据类型时，也要根据情况，决定是将其定义为不变类型还是可变类型。

前面的讨论实际上说明，程序员也需要掌握抽象数据类型的思想和技术。同时说明，编程语言应支持程序员定义自己的抽象数据类型。下面首先通过一些例子，考察在定义一个抽象数据类型时应该怎样思考问题，怎样描述抽象数据类型，描述中应该给出哪些信息。然后讨论 Python 语言怎样支持这一类定义。

2.1.3 抽象数据类型的描述

定义一个抽象数据类型，目的是要定义一类计算对象，它们具有某些特定的功能，可以在计算中使用。这类对象的功能体现为一组可以对它们使用的操作。当然，还需要为这一抽象数据类型确定一个类型名。

下面为抽象数据类型引进一种描述方式，其形式体现了抽象数据类型的主要特点。在后面介绍各种数据结构时，有关章节中也经常是先给出一个抽象数据类型的描述。写出这种描述的过程本身也很有意义，因为它能帮助开发者理清对希望定义的数据类型的想法，清晰地表述出各方面形式要求（如操作的名字、参数的个数和类型等）和功能要求（希望这个操作完成什么计算，或产生什么效果等）。

现在考虑一个简单的有理数抽象数据类型，有下面描述：

ADT Rational:	# 定义有理数的抽象数据类型
Rational(int num, int den)	# 构造有理数num/den
+(Rational r1, Rational r2)	# 求出表示r1+r2的有理数
-(Rational r1, Rational r2)	# 求出表示r1-r2的有理数
*(Rational r1, Rational r2)	# 求出表示r1*r2的有理数
/(Rational r1, Rational r2)	# 求出表示r1 / r2的有理数
num(Rational r1)	# 取得有理数r1的分子
den(Rational r1)	# 取得有理数r1的分母

这里用特殊名字 ADT 表示这是一个抽象数据类型类型的描述，随它之后给出被定义类型的名字。ADT 定义的主要部分描述了一组操作，每个操作的描述由两个部分组成：首先是用标识符或者特殊符号的形式给出的操作名和操作的参数表，随后用类似 Python 注释的形式给出操作的功能描述。另请注意，在描述操作的参数时，可以考虑在参数名前写一个类型名，表示这个参数应该具有的类型；也可以省略，通过文字叙述说明。

具体到上面的抽象数据类型，其名字是 Rational，其中共提供了 7 个操作。第一个操作以 Rational 作为名字，这种形式表示它是一个最基本的构造操作，从其他类型的参数出发构造本类型的对象。随后的几个算术运算也是构造操作，它们基于 Rational 类型的对象生成 Rational 类型的新对象。最后两个是解析操作，取得有理数对象的性质（成分）。

使用抽象数据类型的思想和技术，不但可以描述有理数一类数学类型，也可以描述实际应用中所需的各种类型。例如，下面描述了一个表示日期的抽象数据类型：

ADT Date:	# 定义日期对象的抽象数据类型
Date(int year, int month, int day)	# 构造表示year/month/day的对象
difference(Date d1, Date d2)	# 求出d1和d2的日期差
plus(Date d, int n)	# 计算出日期d之后n天的日期
num_date(int year, int n)	# 计算year年第n天的日期
adjust(Date d, int n)	# 将日期d调整n天 (n为带符号整数)

在这个描述里，同样用注释的形式给出了每个操作的解释。注意，上面这个类型里出现了一个第3类操作 `adjust`。举例说明其用途：假设在一个实际应用中建立了一个表示开会日期的对象，随后这个对象被系统中的许多地方（体现为具体的功能模块）共享，如会务、交通、餐饮、住宿等方面的管理子系统。后来出现了一些情况，导致会议的会期需要修改。这时存在两种修改方案：其一是用 `adjust` 操作去修改那个日期对象，由于对象共享，这样修改的效果将被各有关机构直接看到；第二个方案是另行构造一个表示新会期的对象，然后重新给各个部门发一轮通知，要求它们都用新日期对象替换原来的对象。显然这两种方案都能解决问题，但是基于它们的工作细节却大不相同。

上面看了两个抽象数据类型的例子，现在总结其中的一些情况：

- 一个 ADT 描述由一个头部和按一定格式给出的一组操作描述构成。
- ADT 的头部给出类型名，最前面是表示抽象数据类型的关键词 ADT。
- 操作的形式描述给出操作的名字、参数的类型和参数名。在 ADT 描述中，参数名主要用在解释这个操作的功能的地方（上面借用了 Python 的注释形式）。
- 各操作的实际功能用自然语言描述，这是一种非形式的说明，主要是为了帮助理解这些操作需要（能够）做什么，以便正确地实现和使用它们。

在抽象数据类型的描述中，其他方面都比较清晰和严格，用自然语言形式给出的功能描述则不然。自然语言有着天然的非精确性和歧义性，用它写的描述很难精确无误。这种描述的意义需要人去理解，误解是造成错误的最重要根源之一。

举例说，仔细考虑上面有关日期的 ADT，会发现一些说得不够清楚的地方。例如，“求出 d1 和 d2 的日期差”是什么意思？是否包含两端（或者一端）的日期？对整数 n “调整 n 天”的确切含义是什么？这些可能需要进一步解释。

实际上，这类问题确实比较难处理，因为上述说明是希望解释有关操作的“语义”，也就是它的意义、行为或者效果。对各种实际程序部件（或推而广之，各种程序和实际应用），精确而且正确地理解其中各种操作的功能，显然是最重要的一件事情。但是，语义的描述却很不简单。虽然在有关计算机科学技术的研究中，人们已经提出了一些描述语义的方法，但这些方法都比较复杂，确切的描述仍然不容易写好，也不容易理解。使用这些描述方式需要特别的学习和锻炼，绝大部分程序开发者没有这种经验，因此在实践中使用得不多。语义描述方面的进一步情况已经超出了本课程的范围，这里不再深入。所以，本书后面的实例还将继续使用自然语言的描述。当然，在写这种描述时，应该尽可能避免歧义性和误解。例如，可以在描述中结合使用数学符号和自然语言，对一些一般性情况和特殊情况，给出具体实例说明等。这种方式基本上能满足本书的需要。

ADT 是一种思想，也是一种组织程序的技术，主要包括：

- 1) 围绕着一类数据定义程序模块，如上面的 Rational 和 Date 都是这样。

- 2) 模块的接口和实现分离。上面只给出了模块的接口规范，包括模块名、模块提供的各个操作的名字和参数。每个操作还有非形式化的语义说明。
 - 3) 在需要实现时，从所用的编程语言里选择一套合适的机制，采用合理的技术，实现这种 ADT 的功能，包括具体的数据表示和操作。
- 如何在 Python 里实现抽象数据类型，是下一节的主题。

2.2 Python 的类

在讨论了抽象数据类型的基本思想和描述技术之后，现在考虑它们在 Python 语言里的实现。Python 语言里没有直接的 ADT 定义，实现 ADT 可以采用很多不同的技术。本节介绍最常用也是最自然的一种技术：利用 class 定义（类定义）实现抽象数据类型。本节假定学习者已有基本的 Python 编程经验，但不熟悉其中的 class 定义。熟悉这些内容的读者可以跳过本章下面的内容。本节从例子出发介绍 class 定义的使用、其结构和主要设施。下一节将进一步讨论 Python 中基于 class 的编程技术，称为面向对象技术。

2.2.1 有理数类

类（class）定义机制用于定义程序里需要的类型，定义好的一个类就像一个系统内部类型，可以产生该类型的对象（也称为该类的实例），实例对象具有这个类所描述的行为。实际上，Python 语言把内置类型都看作类。

在介绍 Python 类定义的详细情况之前，这里先给出一个类定义的实例。下面代码是一个简化的有理数类的一部分：

```
class Rational0:  
    def __init__(self, num, den=1):  
        self.num = num  
        self.den = den  
  
    def plus(self, another):  
        den = self.den * another.den  
        num = (self.num * another.den +  
               self.den * another.num)  
        return Rational0(num, den)  
  
    def print(self):  
        print(str(self.num)+"/"+str(self.den))
```

下面对这段代码做些解释：

- class 是关键字，表示由这里开始一个类定义。class 之后是给定的类名和一个表示类头部结束的冒号。这部分（这一行）称为类定义的头部，随后是类定义的体部分，形式上就是一个语句组。定义一个类，通常是为了创建该类的实例，称为该类的实例对象，简称这个类的对象。例如，上面有理数类的名字是 Rational0，定义它就是为了在程序里创建和使用 Rational0（一种有理数）对象。
- 类的体部分通常主要是一批函数定义，所定义的函数称为这个类的方法。最常见的方法是操作本类的实例对象的方法，称为实例方法。这种方法总是从本类的对象出发去调用，其参数表里的第一个参数就表示实际使用时的调用对象，通常以 self 作为参数名。例如，Rational0 类里定义了三个实例方法。下面讨论中简单地说“方法”时，

指的总是实例方法，其他情况在后面介绍。

- 在一个类里，通常都会定义一个名为 `__init__` 的方法（其名字是在 `init` 的前后各加两个下划线符号），称为初始化方法，其工作是构造本类的新对象。创建实例对象采用函数调用的描述形式，以类名作为函数名，这时系统将建立一个该类的新对象，并自动对这个对象执行 `__init__` 方法。例如，下面语句

```
r1 = Rational0(3, 5)
```

就是要求创建一个值为 $3/5$ 的有理数对象，并把这个新对象赋给变量 `r1`。调用式应给出除 `self` 之外的其他实际参数。`Rational0` 类的 `__init__` 方法要求两个实参，上面语句中是 3 和 5。求值表达式时 Python 系统先建立一个新对象，然后把这个对象作为 `__init__` 方法的 `self` 参数去执行方法体。

在 `Rational0` 类的 `__init__` 方法里有两个语句，要求用实参的值给 `self.num` 和 `self.den` 赋值。在实例方法的体中，`self.fname` 形式的写法表示本类实例对象的属性，其中 `fname` 称为属性名。与 Python 变量的情况类似，程序里不需要说明对象有哪些属性，赋值时就会创建。上面初始化方法要求给 `Rational0` 对象的两个属性赋值，创建本类对象时就会为它建立相应的属性并赋以相应的值。

- 类里的其他实例方法也应该以 `self` 作为第一个参数。对它们的调用需要从本类的实例出发，用圆点形式描述。如果写

```
r2 = r1.plus(Rational0(7, 15))
```

赋值右边的表达式调用方法 `plus`，`r1` 的值被称为 `plus` 方法的调用对象，方法中 `self` 参数将约束于该对象。调用中的实参表达式 `Rational0(7, 15)` 创建另一个有理数对象，它将作为 `plus` 方法的第二个实参约束到形参 `another`。

- 上面类定义里的 `print` 方法只有一个 `self` 参数，其调用形式就应该是 `r1.print()`，不要求其他实参。该方法以字符串形式输出对象 `r1` 的内容。

在定义了类 `Rational0` 之后，如果送给 Python 系统下面语句：

```
r1 = Rational0(3, 5)
r2 = r1.plus(Rational0(7, 15))
r2.print()
```

解释器将输出 $80/75$ 。容易看到这个结果没有化简，虽然正确却不是最合适的形式。如果程序里用这样的有理数对象做复杂计算，计算结果的分子和分母都会变得越来越大。虽然 Python 支持任意大的整数，得到的结果应该是正确的，但存储大的整数需要大的空间，计算也更费时间。所以，实现有理数的计算时应该考虑化简。

下面将考虑一个更完整合理的有理数实现，顺便介绍类定义的更多情况。

2.2.2 类定义进阶

如前所述，类定义的一类重要作用是支持创建抽象的数据类型。在建立这种抽象时，人们不希望暴露其实现的内部细节。例如，对于有理数类，不希望暴露这种对象内部是用两个整数分别表示分子和分母。对更复杂的抽象，信息隐藏的意义可能更重要。由于隐藏抽象的内部信息在软件领域意义重大，有些编程语言为此提供了专门机制。Python 语言没有专门服务于这种需求的机制，只能依靠一些编程约定。

首先，人们约定，在一个类的定义里，由下划线 `_` 开头的属性名（和函数名）都当作内部

使用的名字，不应该在这个类之外使用。另外，Python 对类定义里以两个下划线开头（但不以两个下划线结尾）的名字做了特殊处理，使得在类定义之外不能直接用这个名字访问。这是另一种保护方式。下面定义更好的有理数类时将遵循这些约定。

上节最后说到有理数的化简问题。在建立有理数时，应该考虑约去其分子和分母的最大公约数，避免无意义的资源浪费。为了完成化简，需要定义一个求最大公约数的函数 gcd。这里出现了一个问题：应该在哪里定义这个函数。稍加分析就会发现，现在出现了两个新情况：首先，gcd 的参数应该是两个整数，它们不属于被定义的有理数类型。此外，gcd 的计算并不依赖任何有理数类的对象，因此其参数表中似乎不应该以表示有理数的 self 作为第一个参数。但另一方面，这个 gcd 是为有理数类的实现而需要使用的一种辅助功能，根据信息局部化的原则，局部使用的功能不应该定义为全局函数。综合这两点情况，gcd 应该是在有理数类里定义的一个非实例方法。

Python 把在类里定义的这种方法称为静态方法（与实例方法不同），描述时需要在函数定义的头部行之前加修饰符 @staticmethod。静态方法的参数表中不应该有 self 参数，在其他方面没有任何限制。对于静态方法，可以从其定义所在类的名字出发通过圆点形式调用，也可以从该类的对象出发通过圆点形式调用。本质上说，静态方法就是在类里面定义的普通函数，但也是该类的局部函数。

还有一个问题也需要考虑：前面简单有理数类的初始化方法没有检查参数，既没检查参数的类型是否合适（显然，两个实参都应该是整数），也没有检查分母是否为 0。此外，人们传送给初始化方法的实参可能有正有负，内部表示应该标准化，例如，保证所有有理数内部的分母为正，用分子的正负表示有理数的正负。这些检查和变换都应该在有理数类的初始化方法里完成，保证构造出的有理数都是合法合规的对象。

考虑了上面这些问题后，可以给出下面的有理数类定义（部分）：

```
class Rational:
    @staticmethod
    def _gcd(m, n):
        if n == 0:
            m, n = n, m
        while m != 0:
            m, n = n % m, m
        return n

    def __init__(self, num, den=1):
        if not isinstance(num, int) or not isinstance(den, int):
            raise TypeError
        if den == 0:
            raise ZeroDivisionError
        sign = 1
        if num < 0:
            num, sign = -num, -sign
        if den < 0:
            den, sign = -den, -sign
        g = Rational._gcd(num, den)
        # call function gcd defined in this class.
        self._num = sign * (num//g)
        self._den = den//g
```

在这个类里定义了一个局部使用的求最大公约数的静态方法 _gcd，在初始化方法里使用。初始化函数在开始处检查参数的类型和分母的值，不满足要求抛出适当的异常。随后的 if 语句提取有理数的符号，几个检查之后 sign 值为 1 表示是正数，-1 表示是负数。最后用化简后

的分子和分母设置有理数的数据属性。

下面考虑 Rational 类的其他方法。首先，在上面定义中把有理数对象的两个属性都当作内部属性[⊖]，不应该在类之外去引用它们。但实际计算中有时需要提取有理数的分子或分母。为满足这种需要，应该定义一对解析操作（也是实例方法）：

```
def num(self): return self._num
def den(self): return self._den
```

现在考虑有理数的运算。在前面的简单有理数类里定义了名字为 plus 的方法。对于有理数这种数学类型，人们可能更希望用运算符（+、-、*、/ 等）描述计算过程，写出形式更自然的计算表达式。Python 语言支持这种想法，它为所有算术运算符规定了特殊方法名[⊖]。Python 中所有特殊的名字都以两个下划线开始，并以两个下划线结束。例如，与 + 运算符对应的名字是 __add__，与 * 对应的名字是 __mul__。下面是实现有理数运算的几个方法定义，其他运算不难类似地实现：

```
def __add__(self, another):      # mimic + operator
    den = self._den * another.den()
    num = (self._num * another.den() +
           self._den * another.num())
    return Rational(num, den)

def __mul__(self, another):      # mimic * operator
    return Rational(self._num * another.num(),
                   self._den * another.den())

def __floordiv__(self, another): # mimic // operator
    if another.num() == 0:
        raise ZeroDivisionError
    return Rational(self._num * another.den(),
                   self._den * another.num())

# .....
# 其他运算符可以类似定义：
# -:__sub__, /:__truediv__, %:__mod__, etc.
```

这里有几个问题值得提出。首先，通过在每个方法最后用 Rational(..., ...) 构造新对象，所有构造出的对象都保证能化简为最简形式，不需要在每个建立新有理数的地方考虑化简问题。这种做法很值得提倡。

另外，上面定义除法时用的是整除运算符 “//”。在除法方法的开始检查除数并可能抛出异常，也是常规的做法。按照 Python 的惯例，普通除法 “/” 的结果应是浮点数，对应方法名是 __truediv__，如果需要可以另行定义，实现从有理数到浮点数的转换。

还请注意一个情况：算术运算都要求另一个参数也是有理数对象。如果希望检查这个条件，可以在方法定义的开始加一个条件语句，用内置谓词 isinstance(another, Rational) 检查。另外，由于 another 是另一个有理数对象，上面方法定义中没有直接去

[⊖] 不难想清楚，这两个属性确实应该作为内部属性。如果类的外部随便设置有理数的分子和分母，有可能造成一些不应该出现的情况。首先，外部设置的值可能不满足有理数的需要，例如不是整数，或者分母不是大于 0 的整数。还有，作为一种数学对象，有理数不应该是可变对象。显然下面的情况是不可接受的：计算中得到了一个有理数，在后来的使用中这个数却会不断变化。

[⊖] 实际上，Python 为语言中所有运算符定义了特殊方法名，详情请查看 Python 文档。

访问其成分，而是通过解析函数。

有理数对象经常需要比较相等和不等，有些类的对象需要比较大小。Python 为各种关系运算提供了特殊方法名。下面是有理数相等、小于运算的方法定义：

```
def __eq__(self, another):
    return self._num * another._den() == self._den * another._num()

def __lt__(self, other):
    return self._num * other._den() < self._den * other._num()

# 其他比较运算符可以类似定义：
# !=:__ne__, <=:__le__, >=__gt__, >=__ge__
```

不等、小于、大于等运算可以类似地实现。

为了便于输出等目的，人们经常在类里定义一个把该类的对象转换到字符串的方法。为了保证系统的 str 类型转换函数能正确使用，这个方法应该采用特殊名字 __str__，内置函数 str 将调用它。下面是有理数类的字符串转换方法：

```
def __str__(self):
    return str(self._num) + "/" + str(self._den)

def print(self):
    print(self._num, "/", self._den)
```

至此一个简单的有理数类就基本完成了，其中缺少的一些运算的定义情况类似，读者自己完成已经没有实质性困难。

在程序定义好一个类之后，就可以像使用 Python 系统里的类型一样使用它。首先是创建类的对象，形式是采用类名的函数调用式，前面已经说明：

```
five = Rational(5) # 初始化方法的默认参数保证用整数直接创建有理数
x = Rational(3, 5)
```

如果一个变量的值是这个类的对象，就可以用圆点记法调用该类的实例方法：

```
x.print()
```

由于有理数类定义了 str 转换函数，可以直接用标准函数 print 输出：

```
print("Two thirds are", Rational(2, 3))
```

对于有理数，可以使用类中定义的算术运算符和条件运算符：

```
y = five + x * Rational(5, 17)
if y < Rational(123, 11): ...
```

还可以获得对象的类型，或者检查对象和类的关系：

```
t = type(five)
if isinstance(five, Rational): ...
```

总而言之，从使用的各方面看，用类机制定义的类型与 Python 系统的内部类型没有什么差别，地位和用法相同。Python 标准库的一些类型就是这样定义的。

2.2.3 本书采用的 ADT 描述形式

本书后面章节将主要采用 Python 的面向对象技术和类结构定义各种数据类型，为了更好地与之对应，这里对 ADT 的描述形式做一点改动。后面使用的 ADT 描述将模仿 Python 类定

义的形式，也认为 ADT 描述的是一个类型，因此：

- ADT 的基本创建函数将以 `self` 为第一个参数，表示被创建的对象，其他参数表示为正确创建对象时需要提供的其他信息。
- 在 ADT 描述中的每个操作也都以 `self` 作为第一个参数，表示被操作对象。
- 定义二元运算时也采用同样的形式，其参数表将包括 `self` 和另一个同类型对象，操作返回的是运算生成的结果对象。
- 虽然 Python 函数定义的参数表里没有描述参数类型的机制，但为了提供更多信息，在下面写 ADT 定义时，有时还是采用写参数类型的形式，用于说明操作对具体参数的类型要求。在很多情况下，这样写可以省略一些文字说明。

按这种方式描述的有理数对象 ADT 如下：

ADT Rational:	# 定义有理数的抽象数据类型
Rational(self, int num, int den)	# 构造有理数num/den
+ (self, Rational r2)	# 求出本对象加r2的结果（有理数）
- (self, Rational r2)	# 求出本对象减r2的结果
* (self, Rational r2)	# 求出本对象乘以r2的结果
/ (self, Rational r2)	# 求出本对象除以r2的结果
num(self)	# 取得本对象的分子
den(self)	# 取得本对象的分母

2.3 类的定义和使用

前面给出了两个有理数类的定义，帮助读者得到一些有关 Python 类机制的直观认识。本节将介绍 Python 类定义的进一步情况。本书中对类的使用比较规范，涉及的与 Python 类定义相关的机制不多，只需要有最基本的了解就可以学习后面内容。另一方面，本书的主题是数据结构和算法，并不计划全面完整地介绍 Python 语言的面向对象机制和各种使用技术。本节主要想给读者提供一些可参考的基本材料，因此，下面有关 Python 语言的相关介绍将限制在必要的范围内，供读者参考，不深入讨论。有关 Python 面向对象技术的更多细节，请读者参考其他书籍和材料。

2.3.1 类的基本定义和使用

本小节介绍 Python 类定义和使用的基本情况。

类定义

类定义的基本语法是：

```
class <类名>:  
    <语句组>
```

一个类定义由关键词 `class` 开始，随后是用户给定的类名，一个冒号，以及一个语句组。这个语句组称为类（定义）的体部分。

与函数定义类似，类定义也是 Python 的一种语句。类定义的执行效果（语义）就是建立起这个定义描述的类。在 Python 里建立的类也是一种对象，表示一个数据类型。类对象的主要作用就是可以创建这个类的实例（称为该类的实例对象）。

一个类定义确定了一个名字空间，位于类体里面的定义都局部于这个类体，这些局部名字在该类之外不能直接看到，不会与外面的名字冲突。在执行一个类定义时，将以该定义为作用

域创建一个新的名字空间。类定义里的所有语句（包括方法定义）都在这个局部名字空间里产生效果。这样创建的类名字空间将一直存在，除非明确删除（用 `del`）。当一个类定义的执行完成时，Python 解释器创建相应的类对象，其中包装了该类里的所有定义，然后转回到原来的（也就是该类定义所在的）名字空间，在其中建立这个新的类对象与类名字的约束。在此之后通过类名字就能引用相应的类对象了。

在很多类定义的体里只有一组 `def` 语句（前面 `Rational` 类就是如此），为这个类定义一组局部函数。实际上，完全可以在那里写其他语句，其可能用途后面有简单讨论。类里定义的变量和函数等称为这个类的属性。这里的函数定义常采用一种特殊形式，下面将详细介绍具有这种特殊形式的函数与“方法”之间的关系。

类定义可以写在程序里的任何地方，这一点与函数定义的情况类似。例如，可以把类定义放在某个函数定义里，或者放在另一类定义里，效果是在那里建立一个局部的类定义。但在实践中，人们通常总是把类定义写在模块最外层，这样定义的（类）类型在整个模块里都可以用，而且允许其他模块里通过 `import` 语句导入和使用。

类对象及其使用

前面说了，执行一个类定义将创建起一个类对象，这种对象主要支持两种操作：属性访问和实例化（即创建这个类的实例对象）。

在 Python 语言里，所有属性引用（属性访问）都采用圆点记法。例如，基于模块名引用其中的函数（如 `math.sin(...)` 等），类也是如此。在程序里，可以从类名出发，通过属性引用的方式访问有定义的属性，取得它们的值。类里的数据属性（相当于在类里有定义的局部变量）可以保存各种值，类中函数定义生成其函数属性，这种函数属性的值就是一个函数对象，可以通过类名和属性名，采用圆点记法调用。此外，每个类对象都有一个默认存在的 `__doc__` 数据属性，其值是该类的文档串。

在定义好一个类后，可以通过实例化操作创建该类的实例对象。实例化采用函数调用的语法形式，最简单情况就像是调用一个无参函数，例如

```
x = className()
```

假设 `className` 是一个已有定义的类。上面语句将创建 `className` 类的一个新实例（实例对象），并把该对象赋给变量 `x`。如果 `className` 类里没有定义初始化函数，这一简单调用创建的是该类的一个空对象，其中没有数据属性。

2.3.2 实例对象：初始化和使用

虽然 Python 允许先创建空对象，而后再逐步加入所需的属性，但在实际编程中人们大都不这样做。这样做不太合适的主要原因有两个：一是因为一个个地给对象加入属性，工作很琐碎也很麻烦；更重要的是这样创建出属于同一个类的对象，需要自己维持某种规范性，否则就不能保证在程序运行中的安全使用。

实例对象的初始化

创建一个类的实例对象时，人们通常希望对它做一些属性设置，保证建立起的对象状态完好，具有所需要的性质，也就是说，希望在创建类的实例对象时自动完成适当的初始化。Python 类中具有特殊名字 `__init__` 的方法自动完成初始化工作：

- 如果在一个类里定义了 `__init__` 方法，在创建这个类的实例时，Python 解释器就会

自动调用这个方法。

- `__init__`方法的第一个参数（通常用`self`作为参数名）总表示当前正在创建的对象。方法体中可以通过属性赋值的方式（形式为`self.fname`的赋值，`fname`是自己选定的属性名）为该对象定义属性并设定初始值。
- `__init__`可以有更多形式参数。如果存在这种参数，在创建该类的实例对象时，就需要为（除第一个`self`之外的）形式参数提供实际参数值，用表达式写在类名后面的实参表里。在创建实例对象时，这些实际参数将被送给`__init__`，使它可以基于这些实参对实例对象做特定的初始化。

在前面定义有理数类时定义了初始化函数。因此，语句

```
twothirds = Rational(2, 3)
```

的执行中将完成一系列动作：①创建一个`Rational`类型的对象；②调用`Rational`类的`__init__`函数给这个对象的两个属性赋值；③返回新建的对象。最后，赋值语句把这个新对象赋给变量`twothirds`，作为其约束值。

类实例（对象）的数据属性

对于已有的类实例对象，可以通过属性引用的方式访问其数据属性。在Python语言里，类实例的数据属性不需要专门声明，只要给对象的属性赋值，就会自动建立这种属性（就像普通变量一样）。每个实例对象是一个局部名字空间，其中包含该对象的所有数据属性及其约束值，对象的全体数据属性的取值情况构成该对象的状态。如果建立的是空对象，它就有一个空的名字空间。如果在类里定义了初始化函数，创建的实例对象就会包含该函数设置的属性。例如，上面语句创建的`Rational`类的实例有一个局部名字空间，其中包含两个属性名`num`和`den`，它们被分别约束于整数值2和3。

由于上述情况，人们在定义类时，通常总是通过自动调用的`__init__`函数建立实例对象的初始状态，用类里定义的其他函数查看或修改实例对象的状态。实际上，Python允许在任意方法里给原本没有的（也就是说，初始化函数没建立的）属性赋值，这种赋值将扩大该对象的名字空间。但在实际中这种做法并不多见。

一个实例对象是一个独立的数据体，可以像其他对象一样赋给变量作为约束值，或者传进函数处理，或者作为函数的结果返回等。实例对象也可以作为其他实例对象的属性值（无论是同属一个类的实例对象，或不属于同一个类的实例对象），这种情况形成了更复杂的对象结构。在复杂的程序里，这种情况很常见。

方法的定义和使用

除数据属性外，类实例的另一类属性就是方法。

在一个类定义里按默认方式定义的函数，都可以作为这个类的实例对象的方法。但是，如果确实希望类里的一个函数能作为该类实例的方法使用，这个函数至少需要有一个表示其调用对象的形参，放在函数定义的参数表里的第一个位置。这个形参通常取名`self`（实际上可以用任何名字，用`self`作为参数名是Python社团的习惯做法）。除了`self`之外，还可以根据需要为函数引入更多形参。下面将称类里的这种函数为（实例）方法函数。除了是在类里定义而且至少有一个形参外，方法函数并没有别的特殊之处，从其他方面看它们就是一般的函数，Python关于函数的规定在这里都适用。

简单地说，如果类里定义了一个方法函数，这个类的实例对象就可以通过属性引用的方式

调用这个函数。在用 `x.method(...)` 的形式调用方法函数 `method` 时，对象 `x` 将被作为 `method` 的第一个实参，约束到方法函数的第一个形参 `self`，其他实参按 Python 有关函数调用的规定分别约束到 `method` 的其他形参，然后执行该函数的体代码。

说得更准确些。如果在程序里通过某个类 C 的实例对象 o，以属性引用的形式调用类 C 里定义的方法函数 m，Python 解释器就会创建一个方法对象，把实例对象 o 和方法函数 m 约束在这个方法对象里。在（后面）执行这个方法对象时，o 就会被作为函数 m 的第一个实参。在函数 m 的定义里通过形参 `self` 的属性访问，都实现为对调用对象 o 的属性访问（取值或赋值）。看一个具体例子（和写法）：

- 假设在类 C 里定义了方法函数 m，C.m 就是一个函数，其值是普通的函数对象，就像采用其他方式定义的函数一样，例如 `math.sin` 的值就是一个函数对象。
- 假设变量 p 的值是类 C 的一个实例，表达式 p.m 的值就是基于这个实例和函数 m 建立的一个方法对象。
- 使用方法对象的最常见方式是直接通过类实例做方法调用。例如，假设类 C 的方法函数 m 有三个形参，变量 p 的值是类 C 的实例，从 p 出发调用 m 就应写成 `p.m(a, b)` 的形式，这里假设 a 和 b 是适合作为另两个参数的表达式。
- 从上面的说明不难看到，方法调用 `p.m(...)` 实际上等价于函数调用 `C.m(p, ...)`。方法的其他参数可以通过调用表达式中的其他实参提供。
- 方法对象也是一种（类似函数的）对象，可以作为对象使用。例如，可以把方法对象赋给变量，或者作为实参传入函数，然后在函数的其他地方作为函数去调用。在上面假设的情况下，程序里完全可以写“`q=p.m`”，而后可以在其他地方写调用 `q(a, b)`，表示用 a 和 b 作为实参调用这个方法。

注意，方法对象和函数对象不同，它实际上包含了两个成分：一个是由类中的函数定义生成的函数对象，另一个是调用时约束的（属于相应类的）一个实例对象。在这个方法对象最终执行时，其中的实例对象将被作为函数的第一个实参。

2.3.3 几点说明

对于类定义、方法定义等机制，有下面几点说明：

- 在执行了一个类定义，从而创建了相应的类对象之后，还可以通过属性赋值的方式为这个类（对象）增加新属性。不仅可以为其增加数据属性，也可以增加函数属性。但是这时需要特别当心，如果新属性与已有函数属性同名，就会覆盖同名的属性，这种情况有时可能是编程错误。人们一般采用特殊的命名规则避免这种错误。同样情况也可能出现在 `__init__` 方法里。在初始化方法里赋值的属性与类定义中的方法同名是一种常见编程错误，应特别注意。举例说，假设 Rational 类定义了名字为 num 的解析操作，如果在 `__init__` 函数里给 `self.num` 赋值，就会覆盖同名的方法定义。前面类定义里的数据属性名是 `_num`，也避免了这种名字冲突。
- 如果需要在一个方法函数里调用同一个类里的其他方法函数，就需要明确地通过函数的第一个参数（`self`），以属性描述的方式写方法调用。例如，在方法函数 f 里调用另一个方法函数 g，应该写 `self.g(...)`。
- 从其他方面看，方法函数也就是定义在类里面的函数。其中也可以访问全局名字空间里的变量和函数，必要时也可以写 `global` 或 `nonlocal` 声明。

- Python 提供了一个内置函数 `isinstance`, 专门用于检查类和对象的关系。表达式 `isinstance(obj, cls)` 检查对象 `obj` 是否为类 `cls` 的实例, 当 `obj` 的类是 `cls` 时得到 `True`, 否则得到 `False`。实际上, `isinstance` 可以用于检测任何对象与任何类型的关系。例如检查一个变量或参数的值是否为 `int` 类型或 `float` 类型等。

静态方法和类方法

除了前面介绍的实例方法之外, 类里还可以定义另外两类函数:

- 第一类是前面介绍过的静态方法, 定义形式是在 `def` 行前加修饰符 `@staticmethod`。静态方法实际上就是普通函数, 只是由于某种原因需要定义在类里面。静态方法的参数可以根据需要定义, 不需要特殊的 `self` 参数。可以通过类名或者值为实例对象的变量, 以属性引用的方式调用静态方法。例如, 在前面 `Rational` 类里用 `Rational._gcd(...)` 的形式调用静态方法 `_gcd`, 也可以写 `self._gcd(...)`。注意, 静态方法没有 `self` 参数。这也意味着, 无论采用上面哪种调用形式, 参数表里都必须为每个形参提供实参, 这里没有自动使用的 `self` 参数。
- 类里定义的另一类方法称为类方法, 定义形式是在 `def` 行前加修饰符 `@classmethod`。这种方法必须有一个表示其调用类的参数, 习惯用 `cls` 作为参数名, 还可以有任意多个其他参数。类方法也是类对象的属性, 可以以属性访问的形式调用。在类方法执行时, 调用它的类将自动约束到方法的 `cls` 参数, 可以通过这个参数访问该类的其他属性。人们通常用类方法实现与本类的所有对象有关的操作。

这里举一个例子。假设所定义的类需要维护一个计数器, 记录程序运行中创建的该类的实例对象的个数。可以采用下面的定义:

```
class Countable:  
    counter = 0  
  
    def __init__(self):  
        Countable.counter += 1  
  
    @classmethod  
    def get_count(cls):  
        return Countable.counter  
  
  
x = Countable()  
y = Countable()  
z = Countable()  
  
print(Countable.get_count())
```

类定义的其他部分省略。

为了记录本类创建的对象个数, `Countable` 类里定义了一个数据属性 `counter`, 其初值设置为 0。每次创建这个类的对象时, 初始化方法 `__init__` 就会把这个对象计数器加一。类方法 `get_count` 访问了这个数据属性。上面程序片段在运行时将输出整数 3, 表示到执行 `print` 语句为止已经创建了 3 个 `Countable` 对象。

类定义的作用域规则

类定义作为 Python 语言里的一种重要定义结构, 也是一种作用域单位。在类里定义的名字(标识符)具有局部作用域, 只在这个类里可用。如果需要在类定义之外使用, 就采用基于

类名字的属性引用方式。例如，下面定义是合法的：

```
class C:  
    a = 0  
    b = a + 1  
  
x = C.b
```

然而，在前面例子里可以看到一个情况：`counter` 是类 `Countable` 的数据属性，但是在 `Countable` 类的两个方法里，都是通过类名和圆点形式，采用属性引用的形式访问 `counter`。实际上，在 Python 里必须这样做，在这方面，类作用域里的局部名字与函数作用域里局部名字有不同的规定。

对于函数定义，其中局部名字的作用域自动延伸到内部嵌套的作用域。正因为这样，如果在一个函数 `f` 里定义局部函数 `g`，在 `g` 的函数体里可以直接使用 `f` 里有定义的变量，或使用在 `f` 里定义其他局部函数，除非这个名字在 `g` 里另有定义。

对于类定义，情况则不是这样。在类 `C` 里定义的名字（`C` 的数据属性或函数属性名），其作用域并不自动延伸到 `C` 内部嵌套的作用域。因此，如果需要在类中的函数定义里引用这个类的属性，一定要采用基于类名的属性引用方式。

私有变量

在面向对象的程序设计领域，人们通常把类实例对象里的数据属性称作实例变量。因为它们就像是定义在实例对象的名字空间里的变量。

在一些面向对象语言里，允许把一些实例变量定义为私有变量，只允许在类定义的内部访问它们（也就是说，只允许在实例对象的方法函数里访问），不允许在类定义之外使用。实际上，在类之外根本就看不到这种变量，这是一种信息隐藏机制。Python 语言里没有为定义私有变量提供专门机制，没有办法说明某个属性只能在类的内部访问，只能通过编程约定和良好的编程习惯来保护实例对象里的数据属性。

在 Python 编程实践中，习惯约定是把以一个下划线开头的名字作为实例对象内部的东西，永远不从对象的外部去访问它们。无论这样的名字指称的是（类或类实例）的数据成员、方法，还是类里定义的其他函数。也就是说，在编程中永远把具有这种名字的属性看作类的实现细节。在前面的 `Rational` 类里，数据属性 `_num` 和 `_den`、函数属性 `_gcd` 都是这种情况，在这个类定义之外都不应该使用。

另外，如果一个属性以两个下划线开头（但不是以两个下划线结尾），在类之外采用属性访问方式直接写这个名字将无法找到它。Python 解释器会对类定义具有这种形式的名字做统一的改名。有关情况见 Python 语言文档。

此外，具有 `__add__` 形式（前后各有两个下划线）的名字有特殊的意义，除了前面介绍过的表示各种算术运算符、比较运算符的特殊名字和 `__init__`、`__str__` 之外，还有一大批特殊名字。有关细节请查看 Python 语言文档。

实际上，在 Python 编程中，上述约定不仅仅针对类及其实例对象，也适用于模块等一切具有内部结构的对象。

2.3.4 继承

基于类和对象的程序设计被称为面向对象的程序设计，在这里的基本工作包括三个方面：定义程序里需要的类（也是定义新类型）；创建这些类的（实例）对象；调用对象的方法完成计

算工作，包括完成对象之间的信息交换等。

在 Python 语言里做面向对象的程序设计，首先要根据程序的需求定义一组必要的类。前面几小节已经介绍了类定义的基本机制，本节将介绍另一种重要机制——继承。继承的作用主要有两个：一个是可以基于已有的类定义新类，通过继承的方式复用已有类的功能，重复利用已有的代码（已有的类定义），减少定义新类的工作量，简化新功能的开发，提高工作效率。另一个作用实际上更重要，就是建立一组类（类型）之间的继承关系，利用这种关系有可能更好地组织和构造复杂的程序。

继承、基类和派生类

在定义一个新的类时，可以列出一个或几个已有的类作为被继承的类，这样就建立了这个新定义类与指定的已有类之间的继承关系。通过继承定义出的新类称为所列已有类的派生类（或称子类），被继承的已有类则称为这个派生类的基类（或父类）。派生类将继承基类的所有功能，可以原封不动地使用基类中已定义的功能，也可以根据需要修改其中的一些功能（也就是说，重新定义其基类已有的某些函数属性）。另一方面，派生类可以根据需要扩充新功能（定义新的数据和 / 或函数属性）。

在概念上，人们把派生类（子类）看作其基类（父类）的特殊情况，它们的实例对象集合具有一种包含关系。假设类 C 是类 B 的派生类，C 类的对象也看作 C 的基类 B 的对象。人们经常希望在要求一个类 B 的实例对象的上下文中可以使用其派生类 C 的实例对象。这是面向对象编程中最重要的一条规则，称为替换原理。许多重要的面向对象编程技术都需要利用类之间的继承关系，也就是利用替换原理。

一个类可能是其他类的派生类，它又可能被用作基类去定义新的派生类。这样，在一个程序里，所有的类根据继承关系形成了一种层次结构（显然不能出现类之间的循环继承，这种情况是编程错误，Python 系统很容易检查这种错误）。Python 有一个最基本的内置类 `object`，其中定义了一些所有的类都需要的功能。如果一个类定义没说明基类，该类就自动以 `object` 作为基类。也就是说，任何用户定义类都是 `object` 的直接或间接派生类。另外，Python 系统定义了一批内置类，各种基本类型形成了一套层次结构。系统中的内置异常也形成了一套层次结构，有关情况在 2.4 节简单介绍。

基于已有类 `BaseClass` 定义派生类的语法形式是：

```
class <类名>(BaseClass, ...):  
    <语句组>
```

列在类名后面括号里的“参数”就是指定的基类，可以有一个或者多个，它们都必须在这个派生类定义所在的名字空间里有定义。Python 允许用更复杂的表达式描述所需要的基类，只要这个表达式的值确实是个类对象。例如，可以在用 `import` 语句导入另一个模块之后，利用在该模块里有定义的类作为基类，定义自己的派生类。

Python 内置函数 `issubclass` 检查两个类是否具有继承关系，包括直接的或间接的继承关系。如果 `cls2` 是 `cls1` 直接的或间接的基类，表达式 `issubclass(cls1, cls2)` 将返回 `True`，否则返回 `False`。实际上，Python 的一些基本类型之间也有子类（子类型）关系。详情请查看标准库手册中有关基本类型的介绍。

作为最简单的例子，下面定义了一个自己的字符串类：

```
class MyStr(str):  
    pass
```

这个类将继承内置类型 str 的所有功能，没做任何修改或扩充。但它是另一个新类，是 str 的一个派生类。有了这个定义，就可以写：

```
s = MyStr(1234)
issubclass(MyStr, str)
isinstance(s, MyStr)
isinstance(s, str)
```

第一个语句创建了一个 MyStr 类型的对象，后三个表达式的值都是 True，其中 issubclass 判断子类关系（派生关系），显然 MyStr 是 str 的派生类。最后一个表达式为真，是因为派生类的对象也是基类的对象。

派生类常需要重新定义 __init__ 函数，完成该类实例的初始化。常见情况是要求派生类的对象可以作为基类的对象，用在要求基类对象的环境中。在使用这种对象时，可能调用派生类自己定义的方法，也可能调用由基类继承的方法。因此，在这种派生类的实例对象里就应该包含基类实例的所有数据属性，在创建派生类的对象时，就需要对基类对象的所有数据属性进行初始化。完成这一工作的常见方式是直接调用基类的 __init__ 方法，利用它为正创建的实例里那些在基类实例中也有的数据属性设置初值。也就是说，派生类 __init__ 方法定义的常见形式是：

```
class DerivedClass(BaseClass):
    def __init__(self, ...):
        BaseClass.__init__(self, ...)
        .... # 初始化函数的其他操作
    .... # 派生类的其他语句（和函数定义）
```

这里继承 BaseClass 类定义派生的 DerivedClass 类。在调用基类的初始化方法时，必须明确写出基类的名字，不能从 self 出发去调用。在调用基类的 __init__ 时，必须把表示本对象的 self 作为调用的第一个实参，可能还需要传另一些（合适的）实参。这个调用完成派生类实例中属于基类的那部分属性的初始化工作。

在派生类里覆盖基类中已定义的函数时，也经常希望新函数是基类同名函数的某种扩充，也就是说，希望新函数包含被覆盖函数的已有功能。这种情况与 __init__ 的情况类似，处理方法也类似：在新函数定义里，可以用 BaseClass.methodName(...) 的形式调用基类方法。实际上，可以用这种形式调用基类的任何函数（无论该函数是不是被派生类覆盖，是不是正在定义的这个函数）。同样需要注意，在这种调用中，通常需要把表示本对象的 self 作为函数调用的第一个实参。

方法查找

如果从一个派生类的实例对象出发去调用方法，Python 解释器需要确定应该调用哪个函数（在哪个类里定义的函数）。查找过程从实例对象所属的类开始，如果在这里找到，就采用相应的函数定义；如果没找到就到这个类的基类里找。这个过程沿着继承关系继续进行，在某个类里找到所需要的函数后就使用它。如果查找过程进行到已经没有可用的基类，最终也没找到所需函数属性，那就是属性无定义，Python 解释器将报告 AttributeError 异常。Python 解释器处理派生类的定义时，将在构造出的类对象里记录其基类的信息，以支持使用这个类（及其对象）时的属性查找。

如前所述，定义派生类时可以覆盖基类里已有的函数定义（也就是说，重新定义一个同名函数）。按照上述查找过程，一旦某函数在派生类里重新定义，在其实例对象的方法调用解析

中，就不会再去使用基类里原来定义的方法了。

假设在某个实例对象调用的一个方法 `f` 里调用了另一个方法 `g`，而且后一方法也是基于这个实例对象调用的（通过 `self.g(...)`）。在这种情况下，查找方法 `g` 的过程就只与这个实例对象（的类型）有关，与前一方法 `f` 是在哪个类里定义的情况无关。

考虑一个实例。假定 `B` 是 `C` 的基类，两个类的定义分别是：

```
# code showing dynamic binding
class B:
    def f(self):
        self.g()
    def g(self):
        print('B.g called.')

class C(B):
    def g(self):
        print('C.g called.')
```

如果在创建 `B` 类的实例对象 `x` 之后调用 `x.f()`，显然将调用 `B` 类里定义的 `g` 并打印出“`B.g called.`”。但如果创建一个 `C` 类的实例对象 `y` 并调用 `y.f()` 呢？

由于 `C` 类里没有 `f` 的定义，`y.f()` 实际调用的是 `B` 类里定义的 `f`。由于在 `f` 的定义里出现了调用 `self.g`，现在出现了一个问题：如何确定应该调用的函数 `g`。从程序的正文看，正在执行的方法 `f` 的定义出现在类 `B` 里，在类 `B` 里，`self` 的类型应该是 `B`。如果根据这个类型去查找 `g`，就应该找到类 `B` 里定义的函数 `g`。采用这种根据静态程序正文去确定被调用方法的规则称为静态约束（另一常见说法是静态绑定）。但 Python 不这样做，它和多数常见的面向对象语言一样，基于方法调用时 `self` 所表示的那个实例对象的类型去确定应该调用哪个 `g`，这种方式称为动态约束。

这样，`y.f()` 的执行过程将是：由于 `y` 是值是 `C` 类的实例对象，首先基于它确定实际应该调用的方法函数 `f`。由于 `C` 类里没有 `f` 的定义，按规则应该到 `C` 类的基类中去查找 `f`。在 `C` 的基类 `B` 里找到了 `f` 的定义，因此应该执行它。下一个问题是在函数 `f` 的执行中遇到了调用 `self.g()`。由于当时 `self` 的值是一个 `C` 类的实例对象，确定 `g` 的工作再次从调用对象所属的 `C` 类开始进行。由于 `C` 类里存在函数 `g` 的定义，它就是应该调用的方法，执行这个方法函数将打印出“`C.g called.`”。

在程序设计领域，这种通过动态约束确定调用关系的函数称为虚函数。

标准函数 `super()`

Python 提供了一个内置函数 `super`，把它用在派生类的方法定义里，就是要求从这个类的直接基类开始做属性检索（而不是从这个类本身开始查找）。采用 `super` 函数而不直接写具体基类的名字，产生的查找过程更加灵活。如果直接写基类的名字，无论在什么情况下执行，总是调用该基类的方法，而如果写 `super()`，Python 解释器将根据当前类的情况去找到相应的基类，自动确定究竟应该使用哪个基类的属性。

函数 `super` 有几种使用方式，最简单的是不带参数的调用形式，例如

```
super().m(...)
```

如果在一个方法函数的定义里出现这个调用语句，执行到这个语句时，Python 解释器就会从这个对象所属类的基类开始，按照上面介绍的属性检索规则去查找函数 `m`。下面是一段用于说明相关问题的简单代码：

```
class C1:
```

```

def __init__(self, x, y):
    self.x = x
    self.y = y
def m1(self) :
    print(self.x, self.y)
.....
class C2(C1):
    def m1(self):
        super().m1()
        print("Some special service.")
.....

```

如果执行类 C2 里的 m1，Python 解释器将从 C2 的基类开始找 m1（也就是说，从 C1 开始查找）。由于 C1 里有 m1 的定义，最终调用的是 C1 里的函数 m1。显然，这种形式的 super 函数调用（并进而调用基类的某方法函数）只能出现在方法函数的定义里。在实际调用时，当前实例将被作为被调用函数的 self 实参。

函数 super 的第二种使用形式是 super(C, obj).m(...), 这种写法要求从指定的类 C 的基类开始查找函数属性 m，调用里出现的 obj 必须是类 C 的一个实例。Python 解释器找到函数 m 后将用 obj 作为该函数的 self 实参。这种写法可以出现在程序的任何地方，并不要求一定出现在类的方法函数里。

函数 super 其他调用形式的使用情况更特殊，这里就不详细介绍了。

2.4 Python 异常

现在简单介绍 Python 异常与类的关系，语言内建的异常类层次结构，以及 Python 语言如何利用面向对象的观点组织异常处理过程。编程中有时需要自己定义异常（类型），如果需要这样做，就应该选一个系统异常类，从它派生。

2.4.1 异常类和自定义异常

异常是 Python 语言中的一套特殊的控制机制，主要用于支持错误的检查和处理，也可以用于实现特殊的控制转移。如果程序执行中发生异常，无论是解释器发现的异常情况（例如除零或类型错误等），还是通过 raise 语句引发的异常，正常执行控制流立刻终止，解释器转入异常处理模式，查找能处理所发生异常的处理器。如果找不到相应的异常处理器，在交互解释环境下，系统将在环境中输出错误信息，结束当前执行并回到系统的交互状态，等待下一输入。在直接执行方式下，当前程序直接终止。

程序运行中发生的每个异常都有特定的名字，如 ValueError、TypeError、ZeroDivisionError 等，解释器根据发生的异常去查找处理器。Python 里处理异常的结构是 try 语句。每个 try 语句可以带有任意多个 except 子句，这种子句就是异常处理器，子句头部用一个表达式描述它捕捉和处理的异常。

实际上，Python 的异常都是类（class），运行中产生异常就是生成相应类的实例对象，异常处理机制完全基于面向对象的概念和性质。全体内部异常类构成了一个树形结构，所有异常类的基类是 BaseException，其最主要的子类是 Exception，内置异常类都是这个类的直接或间接派生类。如果用户需要定义异常，就应该从系统异常类中选择一个合适的异常，从它派生出自己的异常类。例如：

```

class RationalError(ValueError):
    pass

```

最简单的情况（很常见）只是希望定义一种特殊异常，并不需要这种异常有什么特殊功能（如上即是）。在这种情况下，选一个系统异常类派生自己的异常类，类体不需要定义任何属性。但为了语法完整，可以在这里写一个 `pass` 语句。

如果运行中发生异常，查找相应处理器的工作由解释器完成，这里的工作方式也基于对象和类的关系。在一个 `except` 子句头部可以列出一个或多个异常名（一般说，是表示异常类的表达式），列出多个异常名时需要采用括号括起的元组形式。列在 `except` 子句头部的异常名表示本异常处理器准备捕捉和处理的异常。

运行中发生的异常与处理器的匹配按面向对象的方式处理。假设运行中发生的异常是 `e`，如果一个异常处理器头部列有异常名 `E`，且 `isinstance(e, E)` 为真，那么这个处理器就能捕捉并处理异常 `e`。举例说，如果运行中引发了一个 `RationalError` 异常，某个处理器头部列出了 `RationalError`，或者 `ValueError`，或者 `Exception`，该处理器都能捕捉这个异常。当然，匹配 `ValueError` 的处理器还能捕捉和处理其他异常，匹配 `Exception` 的处理器能捕捉和处理各种主要异常。

2.4.2 异常的传播和捕捉

运行中的异常可能发生在模块层面的语句的执行中，更多情况是发生在某个函数的执行中。假设在函数 `f` 的执行中发生异常 `e`，当前执行立即中断，解释器转入异常处理模式，设法找到处理 `e` 的处理器。有关查找过程如下：

- 首先在发生异常的函数体里查找处理器：
 - 如果发生异常的语句位于一个 `try` 语句体里，首先顺序检查这个 `try` 语句后部的各 `except` 子句，检查是否存在能处理 `e` 的处理器。
 - 如果发生异常的 `try` 语句的所有异常处理器都不能处理 `e`，解释器转去查看包围着该 `try` 语句的外围 `try` 语句（如果存在），检查是否存在能与 `e` 匹配的异常处理器。这个查找过程将在 `e` 发生的函数 `f` 里逐层进行。
 - 如果 `e` 不能在函数 `f` 里处理，`f` 的执行异常终止，`e` 在函数 `f` 这次执行的调用点重新引发，导致又一轮处理器查找工作。查找规则与上面一样。
- 如果上面查找过程在某一步找到了与 `e` 匹配的处理器，解释器就转去执行该 `except` 子句的体（异常处理器代码）。执行完这段代码后，解释器回到正常执行模式，从该异常处理器所在的 `try` 语句之后继续执行。
- 上述查找过程可能导致函数一层层以异常方式退出，有可能一直退到当前模块的最上层也没有找到与之匹配的处理器：
 - 如果程序是在解释器的交互方式下执行，Python 解释器终止该模块执行并回到交互状态，输出错误信息后等待用户的下一个命令。
 - 如果程序是自主执行（或称按批处理方式执行），该程序立即终止。

如果异常发生在主模块的表层（不在任何函数里），处理过程同样如上所述。

在异常处理过程中还可能出现一些情况。例如，正在执行处理器代码时又发生了新异常；或者处理中遇到某些特殊情况，需要引发新的异常。Python 语言里还有关于这些情况的细节规定，这里就不继续讨论了。

2.4.3 内置的标准异常类

Python 语言定义了一套标准异常类，它们都是 `BaseException` 的派生类，其最重要子

类是 `Exception`, 标准异常类都是 `Exception` 的直接或间接派生类。

下面是一些常见异常, 后面有些例子从其中一些异常派生自己的异常类:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- ArithmeticError
    |    +-- FloatingPointError
    |    +-- OverflowError
    |    +-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- EOFError
+-- ImportError
+-- LookupError
|    +-- IndexError
|    +-- KeyError
+-- NameError
|    +-- UnboundLocalError
+-- OSerror
|    +-- BlockingIOError
|    +-- ChildProcessError
|    +-- ConnectionError
|        +-- BrokenPipeError
|        +-- ConnectionAbortedError
|        +-- ConnectionRefusedError
|        +-- ConnectionResetError
|    +-- FileExistsError
|    +-- FileNotFoundError
|    +-- InterruptedError
|    +-- IsADirectoryError
|    +-- NotADirectoryError
|    +-- PermissionError
|    +-- ProcessLookupError
|    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|    +-- NotImplementedError
+-- SyntaxError
|    +-- IndentationError
|        +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|        +-- UnicodeDecodeError
|        +-- UnicodeEncodeError
|        +-- UnicodeTranslateError
```

这个图并不完全, 还有一些被分类为“警告”(Warning)的异常等。

2.5 类定义实例：学校人事管理系统中的类

作为本章内容的总结, 现在考虑一个综合性的实例: 为一个学校的人员管理系统定义所需的表示人员信息的类, 它们都是数据抽象。

2.5.1 问题分析和设计

学校里有两大类人员，即学生和教职工，他们都是需要在系统里表示的对象。分析这两类人员需要记录的信息，可以看到这里有一些值得注意的情况：与两类人员有关的信息中存在一些公共部分，又有各自的特殊情况：

- 首先，作为人员信息，无论学生或教职工都有姓名、性别、年龄等公共信息。另外，为了便于学校管理，学生应该有一个学号，教职工也应该有一个职工号。
- 作为学生应该有学习记录，包括属于哪个院系、注册时间，特别是学习期间已经学过的各门课程及其成绩等。
- 教职工应该有入职时间、院系、职位和工资等信息。

由于两类人员的信息既有共性又有特殊性，特别适合采用面向对象的类继承机制处理。这里的考虑是首先定义一个公共的人员类，提供记录和查询人员基本信息的功能。然后从这个公共类分别派生出表示学生的类和表示教职工的类。

显然，表示人员或者其特殊情况（学生或教职工）都是数据表示问题，应该采用抽象数据类型的思想对其进行分析和设计。在开始具体的类定义（编程）之前，下面首先考虑如何设计出几个合适的抽象数据类型（ADT）。

基本人员 ADT 的设计

首先考虑一般人员 ADT 的定义。为建立这个 ADT 的具体对象，需要提供一组基本信息，包括有关人员的姓名、性别、出生年月日和一个人员编号（学号或职工号，这里要求提供，具体人员类 ADT 可以考虑具体的生成规则）。ADT 的解析操作包括提取人员的编号、姓名、性别、出生年月日和年龄。还应允许人员改名，为此定义一个变动操作。由于人员记录可能需要排序，为此要有一个对象之间的“小于”运算符。还需要为输出定义一些辅助操作。根据这些考虑，可以给出下面抽象数据类型定义：

ADT Person:	# 定义人员抽象数据类型
Person(self, strname, strsex, tuple birthday, str ident)	# 构造人员对象
id(self)	# 取得该人员记录中的人员编号
name(self)	# 取得该人员记录中的姓名
sex(self)	# 取得该人员记录中的性别
birthday(self)	# 取得该人员记录中的出生年月日
age(self)	# 取得该人员的年龄
set_name(self, str name)	# 修改人员姓名
<(self, Person another)	# 基于人员编号比较两个记录
details(self)	# 给出人员记录里保存的数据详情

为管理一个学校的人员，还需考虑人数统计。这件事可以通过类里的数据属性和类方法完成，在下面的实现中可以考虑这个问题。

学生 ADT 的设计

每个学生属于一个院系，入学时确定。另外，学生的学号编制应该按一套规则自动生成，不需要人为选择。新的解析操作包括查看学生所属院系和入学时间（年），查看学生成绩单。变动操作应包括设置选课记录和课程成绩（是变动操作）等。

根据上述分析，可以给出下面抽象数据类型定义。这里借用了 Python 面向对象机制中类继承的形式，Student(Person) 表示继承 Person 抽象数据类型中除构造函数之外的其他操作。可以不引进这种缩写，但需要把 Person ADT 的定义体重抄一遍。

```

ADT Student(Person):
    Student(self, strname, strsex, tuple birthday, str department) # 定义学生ADT
    department(self) # 构造学生对象
    en_year(self) # 取得学生所属院系
    scores(self) # 取得学生入学年度
    set_course(self, str course_name) # 取得学生的成绩单
    set_score(self, str course_name, int score) # 设置选课
    # 设置课程成绩

```

实现这个 ADT 时还需要实现一个生成学号的内部函数。具体技术后面考虑。

教职工 ADT 的设计

与学生的情况相对应，教职工 ADT 应有取得其院系、工资、入职时间等的解析函数，以及设置这些数据的变动操作：

```

ADT Staff(Person):
    Staff(self, strname, strsex, tuple birthday, tuple entry_date) # 定义教职工ADT
    department(self) # 构造教职工对象
    salary(self) # 取得教职工的所属院系
    entry_date(self) # 取得教职工的工资额
    position(self) # 取得教职工的人职时间
    set_salary(self, int amount) # 取得教职工的职位
    ..... # 设置工资额

```

完成了这些设计之后，下面可以进一步考虑其程序实现了。

2.5.2 人事记录类的实现

这里的考虑是定义几个类，实现前面设计的各个 ADT。在定义这些类之前先定义两个异常类，以便在定义人事类的操作中遇到异常情况时引发特殊的异常，使用这些类的程序部分可以正确地捕捉和处理。

人们在定义自己的特殊异常类时，多数时候都采用最简单的方式：只是简单选择一个合适的 Python 标准异常类作为基类，派生时不定义任何方法或数据属性。针对准备实现的类定义，这里派生两个专用的异常类：

```

class PersonTypeError(TypeError):
    pass

class PersonValueError(ValueError):
    pass

```

两个类的体都只有一个 `pass` 语句，只是为了填补语法上的缺位。在下面操作中遇到参数的类型或者值不满足需要时就引发这两个异常。

此外，由于人事类定义需要处理一些与时间有关的数据，直接采用 Python 标准库的有关功能类是最合适的。引进 `datetime` 标准库包：

```
import datetime
```

下面讨论几个人事管理类的实现。

公共人员类的实现

首先考虑基本人员类的定义，将这个类命名为 `Person`。

为了统计在程序运行中建立的人员对象的个数，需要为 `Person` 类引进一个数据属性

`_num`, 每当创建这个类的对象就将其值加一。Person类的`__init__`方法里完成这一工作。下面是Person类的开始部分：

```
class Person:
    _num = 0

    def __init__(self, name, sex, birthday, ident):
        if not (isinstance(name, str) and
                sex in ("女", "男")):
            raise PersonValueError(name, sex)
        try:
            birth = datetime.date(*birthday)      # 生成一个日期对象
        except:
            raise PersonValueError("Wrong date:", birthday)
        self._name = name
        self._sex = sex
        self._birthday = birth
        self._id = ident
        Person._num += 1                      # 实例计数
```

`__init__`方法的主要工作是检查参数合法性，设置对象的数据属性。这些检查非常重要，只有通过细致检查，才能保证建立起的人员对象都是合法对象，使用这些对象的程序可以依赖于它们的合法性。对人员的名字，这里只要求它是一个字符串。对于性别，要求实参是两个汉字字符串之一，用运算符`in`检查。

最麻烦的问题是出生日期的检查。朴素的考虑可能是要求实参为一个三元组，三个元素都是整数，分别表示年、月、日。但是不难想到，并非任意三个整数都构成合法的年月日数据，例如(2015, 23, 48)就不是。可以自己实现完整的检查，但是很麻烦。上面的函数定义利用了标准库包`datetime`里的`date`类，其构造函数要求三个参数，如果实参不是合法日期值就会引发异常。在调用`date`构造日期对象时使用了拆分实参的描述方式。在上面方法定义里，`try`语句的异常处理器没有给定异常名，这说明它将捕捉构造`date`对象时发生的所有异常，处理器的体说明在这种情况下引发`PersonValueError`。后面几个语句都很简单，最后一个语句完成生成实例的计数工作。

Person类的其他方法都非常简单：

```
def id(self): return self._id
def name(self): return self._name
def sex(self): return self._sex
def birthday(self): return self._birthday
def age(self): return (datetime.date.today().year -
                      self._birthday.year)

def set_name(self, name): # 修改名字
    if not isinstance(name, str):
        raise PersonValueError("set_name", name)
    self._name = name

def __lt__(self, another):
    if not isinstance(another, Person):
        raise PersonTypeError(another)
    return self._id < another._id
```

这里有几个小问题需要做一点解释：①标准库`date`类的`today`方法返回函数调用时刻的日期，这也是一个`date`对象。`date`对象的`year`属性记录其年份，上面定义的`age`方法利用

这些功能计算出两个年份之差，得到这个人的年龄。②实现小于运算的方法要求另一个参数也是 Person，然后根据两个人员记录的 _id 域的大小确定记录的大小关系。其余情况都非常简单，无须赘述。

在这个类里还需要定义一个类方法，以便取得类中的人员计数值。另外定义了两个与输出有关的方法，它们都很简单：

```
@classmethod
def num(cls): return Person._num

def __str__(self):
    return ".join((self._id, self._name,
                  self._sex, str(self._birthday)))"

def details(self):
    return ", ".join(("编号: " + self._id,
                      "姓名: " + self._name,
                      "性别: " + self._sex,
                      "出生日期: " + str(self._birthday)))
```

这里的办法是让 `__str__` 提供对象的基本信息，`details` 方法提供完整细节。请注意，字符串的 `join` 方法要求参数是可迭代对象，这里先做出一个元组。

至此 Person 类的基本定义就完成了。下面是使用这个类的几个语句：

```
p1 = Person("谢雨洁", "女", (1995, 7, 30), "1201510111")
p2 = Person("汪力强", "男", (1990, 2, 17), "1201380324")
p3 = Person("张子玉", "女", (1974, 10, 16), "0197401032")
p4 = Person("李国栋", "男", (1962, 5, 24), "0196212018")

plist2 = [p1, p2, p3, p4]
for p in plist2:
    print(p)

print("\nAfter sorting:")
plist2.sort()
for p in plist2:
    print(p.details())

print("People created:", Person.num(), "\n")
```

由于定义了 `__str__` 方法，因此可以直接用 `print` 输出 Person 对象。后几个语句还展示了可以对人员对象的表排序（表的 `sort` 方法里使用了“小于”运算符），以及通过 Person 类名调用类方法 `num` 的情况。

总而言之，这个类实现了前面 ADT 要求的功能。

学生类的实现

现在考虑学生类 `Student` 的实现。在这里需要关注几件事：① `Student` 对象也是 `Person` 对象，因此，在建立 `Student` 对象时，应该调用 `Person` 类的初始化函数，建立起表示 `Person` 对象的那些数据属性^Θ。②这里希望 `Student` 类实现一种学号生成方式。为了保证学号的唯一性，最简单的技术就是用一个计数变量，每次生成学号时将其加一。这个变量应该是 `Student` 类内部的数据，但又不属于任何 `Student` 实例对象，因此应该用类的数据属

^Θ 如前所述，在定义派生类的初始化函数时，通常都先调用基类的初始化函数。当然，是否需要调用应该根据实际需要灵活处理，但上述做法是惯例。

性表示。③学号生成函数只在 Student 类的内部使用，但并不依赖于 Student 的具体实例。根据这些情况，该函数似乎应该定义为静态方法。但是，这个函数并不是独立的，它依赖于 Student 类中的数据属性。根据前面的讨论，应该将其定义为类方法，在其中实现所需学号生成规则。

基于上面考虑，Student 类的初始化函数定义如下：

```
class Student(Person):
    _id_num = 0

    @classmethod
    def _id_gen(cls):      # 实现学号生成规则
        cls._id_num += 1
        year = datetime.date.today().year
        return "1{:04}{:05}".format(year, cls._id_num)

    def __init__(self, name, sex, birthday, department):
        Person.__init__(self, name, sex, birthday,
                        Student._id_gen())
        self._department = department
        self._enroll_date = datetime.date.today()
        self._courses = {} # 一个空字典
```

这里的学号用一个数字字符串表示，利用 str 的 format 方法构造学号：规定学生学号的首位为 1，以便与职工号区分；把学生的入学年份用 4 位十进制数字的形式编码在学号里；最后是 5 位的序号。学生对象里还要记录学生的院系和入学报道日期，最后用一个字典记录课程学习成绩，初始时设置为空字典。

其他方法都很容易考虑，下面只给出与选课和成绩有关的三个方法：

```
def set_course(self, course_name):
    self._courses[course_name] = None

def set_score(self, course_name, score):
    if course_name not in self._courses:
        raise PersonValueError("No this course selected:",
                               course_name)
    self._courses[course_name] = score

def scores(self): return [(cname, self._courses[cname])
                           for cname in self._courses]
```

这里假定了必须先选课，最后才能设定课程成绩。最后一个方法给出所有成绩的列表，其中用了一个表描述式，非常方便。

继续考虑可以发现一个问题：虽然 Person 类的 details 方法仍然可用，但 Student 对象包含的信息更多，原方法不能展示这些新属性。为了满足 Student 类的实际需要，必须修改 details 方法的行为，也就是说，需要定义一个同名的新方法，覆盖基类中已有定义的 details 方法。在定义这种新方法时，应该维持原方法的参数形式，并提供类似的行为，以保证派生类的对象能用在要求基类对象的环境中（“替换原理”）。此外，在新方法里，经常需要首先完成基类同名方法所做的工作。这件事可以通过在新方法里调用基类的同名方法实现。下面是新的 details 方法的定义：

```
def details(self):
    return ", ".join((Person.details(self),
```

```

    "入学日期: " + str(self._enroll_date),
    "院系: " + self._department,
    "课程记录: " + str(self.scores())))

```

当然，并不是每个派生类的覆盖方法都需要重复基类方法的工作，是否调用基类被覆盖的方法，应根据需要确定。如果需要，必须通过基类名去调用^Θ。

其余方法都非常简单，这里不再给出。

教职工类的实现

下面将教职工类命名为 Staff，也定义为 Person 类的子类，继承 Person 类的基本定义。教职工记录对象也应包含 Person 类对象的所有数据属性，以便可以对它们调用 Person 类里定义的方法。在这些数据属性的基础上，Staff 类还需要为其实例扩充一些数据属性，定义一组方法。

首先，Staff 类也需要为教职工对象实现一个职工号生成函数，同样定义为类方法，基于 Staff 类的数据属性完成工作。这里假定职工号的首字符为 0，其中编码了具体教职工的出生年份，在加上一个内定的序号。其余方法的定义都比较自然，下面是 Staff 类的重要部分，一些简单的方法没有给出，读者很容易自己补全：

```

class Staff(Person):
    _id_num = 0
    @classmethod
    def _id_gen(cls, birthday):      # 实现职工号生成规则
        cls._id_num += 1
        birth_year = datetime.date(*birthday).year
        return "0{:04}{:05}".format(birth_year, cls._id_num)

    def __init__(self, name, sex, birthday, entry_date=None):
        super().__init__(name, sex, birthday,
                         Staff._id_gen(birthday))
        if entry_date:
            try:
                self._entry_date = datetime.date(*entry_date)
            except:
                raise PersonValueError("Wrong date:",
                                       entry_date)
        else:
            self._entry_date = datetime.date.today()
        self._salary = 1720          # 默认设为最低工资，可修改
        self._department = "未定"   # 需要另行设定
        self._position = "未定"     # 需要另行设定

    def set_salary(self, amount):
        if not type(amount) is int:
            raise TypeError
        self._salary = amount

    def set_position(self, position):
        self._position = position
    def set_department(self, department):
        self._department = department

```

^Θ 请注意：派生类覆盖了基类的同名方法。如果从 self 出发调用 details，实际调用的将是本类的 details。要调用基类的同名方法，必须通过基类的名字或 super 函数。

```
def details(self):
    return ", ".join((super().details(),
                      "入职日期: " + str(self._entry_date),
                      "院系: " + self._department,
                      "职位: " + self._position,
                      "工资: " + str(self._salary)))
```

请注意：在这里定义初始化方法和 details 方法时，都用 super() 表示要求调用基类的方法，也是为了展示 super 函数的使用技术。在使用 super() 时不需要提供 self 参数（请与 Student 中的情况对比）。实际上，在做复杂的面向对象程序时，人们更提倡采用 super 函数描述这种调用，而不是直呼基类的名字。直接写基类名将造成类定义代码之间更密切的关联，对程序的修改不利。

下面是一些使用这个类的语句：

```
p1 = Staff("张子玉", "女", (1974, 10, 16))
p2 = Staff("李国栋", "男", (1962, 5, 24))

print(p1)
print(p2)

p1.set_department("数学")
p1.set_position("副教授")
p1.set_salary(8400)

print(p1.details())
print(p2.details())
```

2.5.3 讨论

本节通过几个大学人事信息类的定义，总结了利用 Python 面向对象的继承机制，以及在已有类的基础上定义派生类时可能遇到的各种问题，展示了 Python 面向对象编程的许多机制。在上面的类定义里，还介绍了类的数据属性和类方法的使用。

在面向对象编程领域，定义派生类主要有两种用途：

- 1) 定义基类对象中的一类特殊个体，它们具有与基类对象类似的行为，可以作为基类对象使用（替换原理），但通常还有一些自己的特殊功能。为满足这种需要，从基类派生将能直接共享基类定义的操作，通过调用基类的初始化方法，建立派生类对象中与基类对象相同的部分。派生类对象继承基类的方法属性，可以用重新定义的方式覆盖原有方法，也可以定义新方法。在上面实例中，Student 和 Staff 类都是公共人事类 Person 的派生类，其对象都是特殊的 Person 对象。
- 2) 只是为了重用基类已有的功能，而将一个类定义为派生类。实际中有时也有这种需要，主要是为了代码的重用，这也是面向对象中继承机制的一类用途。

在定义一个类时，有时需要保存一些与整个类有关但并不特定于具体实例对象的信息，或者需要一些与整个类有关的功能。这些就需要通过类的数据属性和类方法实现。类的数据属性通过类层面的赋值语句定义，类方法需要用特殊前缀 @classmethod 描述。在上面的实例里，多次使用了这方面的功能，如做对象计数，或为对象生成唯一编号等。这些在实际程序中都经常遇到。

基类和派生类是相对的。例如，为建立一个学校的人事系统，可能还需要从 Staff 出发派生出更具体的教职工类，如教师类、职员类等，或从学生类出发派生出具体的本科生类、硕

士生类、博士生类等。它们又需要有一些特殊的行为。本章后面的习题提出了一些这方面的问题，供读者参考。

本章总结

随着计算机科学技术和软件领域的发展，人们逐渐认识到，数据的抽象和计算过程的抽象同样重要。以建立数据抽象为目标的抽象数据类型的思想逐渐发展起来，对程序和软件系统的设计以及编程语言的发展都产生了广泛而深远的影响。新的编程语言都为建立数据抽象设置了专门的结构或机制。所有设计优良的复杂软件系统在其设计和实现的许多方面都会反映并实践着抽象数据类型的思想。掌握抽象数据类型的基本思想和实践技术，是从简单编程走向复杂的实际应用开发历程中的重要一步。

抽象数据类型的基本思想是抽象定义与数据表示和数据操作的实现分离。定义抽象数据类型，首先要描述好这一类型的对象与外界的接口，通过一组操作（函数）描述。这样的接口定义在程序中划出了一条明晰的分界：一边是抽象数据类型的实现，可以采用适合具体需要的任何技术；另一边是使用这个抽象数据类型的其他程序部分，它们只需要相对于给定的操作接口定义，完全不必考虑有关功能是如何实现的。这种分离能很好地支持程序的模块化组织，是分解和实现大型复杂系统的最重要基础技术。

Python 语言里专门用于支持数据抽象的机制是类（class）及其相关结构。解释器处理完一个类定义后生成一个类对象。类对象也是一种复合对象，具有类定义里描述的所有数据属性和函数属性，约束到给定的类名，就像函数定义将生成的函数对象约束于函数名一样。类对象的最重要功能就是可以通过调用的形式生成该类的实例对象。如果类中定义了名字为 `__init__` 的初始化函数，生成实例对象时就会自动调用它；如果没有定义这个函数，生成的将是一个空对象。人们通常用初始化函数为实例对象建立数据属性，设置实例对象的初始状态。这样生成的实例对象可以通过方法调用的形式，使用其所属类中定义的各个实例函数。在类里还可以定义静态方法和类方法。

面向对象编程的另一个重要机制是继承，用于支持基于已有的一或几个类定义新类。这样定义的新类称为派生类（或子类），被继承的类称为基类（或父类）。派生类可以利用基类的所有机制，可以重新定义基类中已有的方法，改变自己的实例对象的行为，或者通过定义新方法的方式扩充新实例对象的行为。在方法调用时，Python 采用动态约束规则，根据调用对象的类型确定应该调用的函数。面向对象的观点把派生类看作基类的特殊情况，派生类的对象也看作基类的对象。如果在定义新类时不指明基类，Python 就认为基类是 `object`。这样，一个程序里的类定义形成了一种层次结构，其中最高层的类是 `object`，其他类都是 `object` 的派生类。对复杂的程序，开发者可以通过恰当的类层次结构设计，把程序中的各种数据组织好，以利于程序的开发和维护。

综上所述，基于 Python 的类机制，不仅可以定义出一个具体的抽象数据类型，而且可以定义出一组相关的具有层次关系的抽象数据类型。在定义新类型时，可以通过继承的方式尽可能利用已有定义的功能，提高工作效率。良好设计的类层次结构还使开发者可以把所需操作定义在适当的抽象层次上，使操作尽可能地通用化。

Python 的异常处理机制是完全基于类和对象的概念构造起来的。系统定义了一组异常类，形成了一套标准的异常类层次结构。引发一个异常就是生成相应异常类的一个对象。Python 解释器的异常查找机制设法找到与异常匹配的处理器，匹配条件就是发生的异常对象属于处理器

描述的异常类。在这里应该注意，派生类的对象也是基类的对象。因此，捕捉基类异常的处理器也能捕捉属于派生类的异常。如果用户需要定义自己的异常，只需要选择一个系统异常类，基于它定义一个派生类。

练习

一般练习

1. 复习下面概念：抽象数据类型，接口，实现，过程抽象和数据抽象，类型，内置类型和用户定义类型，表示，不变类型和可变类型，不变对象和可变对象，类，类定义和类对象，类对象名字空间，类的属性（数据属性和函数属性），类的实例（实例对象，对象），方法，实例方法，self参数，方法和函数，函数`isinstance`，初始化方法，实例的属性和属性赋值，静态方法，类方法，实例变量和私有变量，Python的特殊方法名，继承，基类，派生类，方法覆盖，替换原理，类层次结构，类`object`，函数`issubclass`，类方法查找，静态约束和动态约束，函数`super`，Python标准异常，异常类层次结构，`Exception`异常，Python异常的传播和捕捉。
2. 请列举出数据类型的三类操作，说明它们的意义和作用。
3. 为什么需要初始化函数？其重要意义和作用是什么？
4. 设法说明在实际中某些类型应该定义为不变类型，另一些类型应该定义为可变类型。请各举出两个例子。
5. 请简要说明在定义一个数据类型时应该考虑哪些问题？
6. 请检查本章给出的`Date`抽象数据类型，讨论其中操作的语义说明里有哪些不精确之处，设法做些修改，消除描述中的歧义性。
7. 请解释并比较类定义中的三类方法：实例方法、静态方法和类方法。
8. 列出Python编程中有关类属性命名的约定。
9. 请通过实例比较类作用域与函数作用域的差异。
10. 试比较本章采用元组实现有理数和采用类实现有理数的技术，讨论这两种不同方式各自的优点和缺点。

编程练习

1. 定义一个表示时间的类`Time`，它提供下面操作：
 - a) `Time(hours, minutes, seconds)` 创建一个时间对象；
 - b) `t.hours()`、`t.minutes()`、`t.seconds()` 分别返回时间对象`t`的小时、分钟和秒值；
 - c) 为`Time`对象定义加法和减法操作（用运算符`+`和`-`）；
 - d) 定义时间对象的等于和小于关系运算（用运算符`==`和`<`）。

注意：`Time`类的对象可以采用不同的内部表示方式。例如，可以给每个对象定义三个数据属性`hours`、`minutes`和`seconds`，基于这种表示实现操作。也可以用一个属性`seconds`，构造对象时算出参数相对于基准时间0点0分0秒的秒值，同样可以实现所有操作。请从各方面权衡利弊，选择合适的设计。

上面情况表现出“抽象数据类型”的抽象性，其内部实现与使用良好隔离，换一种实现方式（或改变一些操作的实现技术）可以不影响使用它的代码。

2. 请定义一个类，实现本章描述的 Date 抽象数据类型。
3. 扩充本章给出的有理数类，加入一些功能：
 - a) 其他运算符的定义；
 - b) 各种比较和判断运算符的定义；
 - c) 转换到整数（取整）和浮点数的方法；
 - d) 给初始化函数加入从浮点数构造有理数的功能（Python 标准库浮点数类型的 as_integer_ratio() 函数可以用在这里）。

对应运算符的特殊函数名请查看语言手册 3.3.7 节（Emulating numeric types）。
4. 本章 2.2.2 节中有理数类的实现有一个缺点：每次调 `__init__` 都会对两个参数做一遍彻底检查。但是，在有理数运算函数中构造结果时，其中一些检查并不必要，浪费了时间。请查阅 Python 手册中与类有关的机制，特别是名字为 `__new__` 的特殊方法等，修改有关设计，使得到的实现能完成工作但又能避免不必要的检查。
5. 请基于 2.5 节的工作继续扩充，为该学校人事系统定义研究生类、教师类和职员类。

第3章 线性表

在程序里，经常需要将一组（通常是同为某个类型的）数据元素作为整体管理和使用，需要创建这种元素组，用变量记录它们，传进传出函数等。一组数据中包含的元素个数可能发生变化（可以加入或删除元素）。在有些情况下，可能需要把这样一组元素看成一个序列，用元素在序列里的位置和顺序，表示实际应用中的某种有意义的信息，或者表示数据之间的某种关系。线性表（简称表）就是这样一组元素（的序列）的抽象。一个线性表是某类元素的一个集合，还记录着元素之间的一种顺序关系。

线性表是最基本的数据结构之一，在实际程序中应用非常广泛，它还经常被用作更复杂的数据结构的实现基础。在实际应用中可能需要各种各样的线性表，如整数的表、字符串的表、某种复杂结构的表等。Python语言的内置类型 `list` 和 `tuple` 都以具体的方式支持程序里的这类需要，它们都可以看作线性表的实现。

本章将讨论线性表概念及其两种基本实现方法、若干变形和一些应用实例。

3.1 线性表的概念和表抽象数据类型

表可以看作一种抽象的（数学）概念，也可以作为一种抽象数据类型。本节首先介绍表的概念和它的一些抽象性质，而后定义一个表抽象数据类型。

3.1.1 表的概念和性质

在抽象地讨论线性表时，首先要考虑一个（有穷或无穷的）基本元素集合 E ，集合 E 中的元素可能都是某个类型的数据对象。

集合 E 上的一个线性表就是 E 中一组有穷个元素排成的序列 $L=(e_0, e_1, \dots, e_{n-1})$ ，其中 $e_i \in E$ 且 $n \geq 0$ 。在一个表里可以包含 0 个或多个元素，序列中的每个元素在表里有一个确定的位置，称为该元素的下标。在本书的讨论中，下标总是从 0 开始编号（一些书籍中选择从 1 开始）。不包含任何元素的表称为空表。

一个表中包含的元素的个数称为这个表的长度。显然，空表的长度为 0。

表元素之间存在一个基本关系，称为下一个关系。对于表 $L=(e_0, e_1, \dots, e_{n-1})$ ，其下一个关系是二元组的集合 $\{\langle e_0, e_1 \rangle, \langle e_1, e_2 \rangle, \dots, \langle e_{n-2}, e_{n-1} \rangle\}$ 。下一个关系是一种顺序关系，即线性关系，线性表是一种线性结构。

在一个非空的线性表里，存在着唯一的一个首元素和唯一的一个尾元素（或称末元素）。除首元素之外，表中的每个元素 e 都有且仅有一个前驱元素；除了尾元素之外的每个元素都有且仅有一个后继元素。

可以把线性表作为一种数学对象加以研究，为之建立抽象的数学模型，研究其性质。但就本书的目的而言，重要的是把线性表作为程序中使用的一种数据类型和构造数据结构的一种基本方式。下面讨论这方面的问题。

3.1.2 表抽象数据类型

从实际编程和应用的角度看，线性表是一种组织数据元素的数据结构，现在考虑如何将其定义为一种抽象数据类型。作为一种抽象描述，有两类人群需要从各自角度考虑这种类型的问题：线性表的实现者和它的使用者：

- 从实现者角度，对于一种数据结构有两个问题必须考虑：①如何把该结构内部的数据组织好（为它设计一种合适的表示）；②如何提供一套有用而且必要的操作，并有效实现这些操作。显然，这两个问题相互有关联。
- 从使用者角度看，线性表是一种有用的结构。需要考虑该结构提供了哪些操作，如何有效使用以解决自己的问题。实际使用会对表的实现者提出一些要求。

显然，不仅对于线性表存在这样的两个视角，并因为视角的不同带来了一系列相关问题，对于其他数据结构也都如此。在设计、定义和使用抽象数据类型时，都会遇到这两个不同的观察角度及其相互关系，它们既统一又有分工。这种情况与函数的定义与使用类似。因此，下面讨论中的一些考虑具有普遍意义。

在一个数据结构里，通常需要保存一些信息，需要把这些信息以某种形式存储起来。数据结构的具体表示完全是其内部的东西，在数据结构之外看不到也不应该关心。但具体的表示可能对这一数据结构上各种操作的实现和性质产生重要影响。对于比较复杂的数据结构，可能存在多种不同的表示方式（后面会看到更多具体例子），设计时需要考虑的因素很多，其中利弊得失的权衡可能很复杂，不容易做出有理有据的决策。数据结构课程提供的信息就是为了帮助人们理解这些问题，理解常见的各种重要利弊权衡。

线性表的操作

下面首先从使用者的角度，考虑一个线性表数据结构应该提供哪些操作。

- 首先，作为抽象类型的线性表是一组数据对象的集合，应该提供创建线性表对象的操作。一种简单操作是创建空表对象，这时不需要提供其他信息。如果需要创建包含一些元素的表，就要考虑如何为创建操作提供初始元素序列的问题。
- 程序中可能需要检查一个表，获取它的各方面信息。例如，可能需要判断一个表是否为空，考察其中的元素个数（求得表的长度），检查一个表里是否存在某个特定数据对象等。为此需要定义一些获取表中信息的解析操作。
- 需要动态改变表的内容，包括加入新元素或删除已有元素。加入新元素有一些不同方式，如简单加入只要求新元素加入表中，定位加入要求把新元素存放在表中的确定位置。删除元素可以是定位删除，删掉某位置的元素，或是按内容删去特定元素。后一种删除还有删除一个元素或删除等于某指定元素的所有元素的问题。例如，可能要求删除一个整数表里的一个0元素，或者所有等于0的元素。此外，还可以是给定一个条件，要求删除所有满足（或不满足）条件的元素。例如删除一个整数表里所有负数。
- 一些涉及一个或两个表的操作。例如表的组合操作，希望得到一个表，其中包含了两个表中所有元素的表；或者从已有的表得到一个新表，其中每个元素都是原表中的元素按某种规则（操作）修改后的结果；等等。
- 涉及对表中每一个元素进行的操作。注意，这是一个操作类，给定任何一个对单个表元素的操作，就有一个与之对应的对表中所有元素进行的操作。这类操作在进行中需要逐个访问表中元素，对每个元素做同样的事情。这是一种操作模式，称为对表元素的遍历。下面也将考虑表的遍历问题。

最后这两类操作都可以实现为变动操作，让它们实际修改被操作的表，在该表上直接实现操作的效果；也可以实现为非变动操作，让操作总是建立一个新表，与原表相比是多了（加入了）或少了（删除了）一个或一些元素，或者每个元素都按特定方式修改。

上面讨论至少可以给读者一个印象：如何设计一类表对象，并没有一个固有正确的操作集合，而是有许多可能，需要根据实际需要考虑问题并做出选择。

上面列出的这些操作中，有些是所有数据结构都需要的“标准操作”。例如结构的创建操作、对组合结构的判空操作。对元素个数可能变化的对象，需要有检查元素个数的操作。不同编程语言或实现方式也可能影响需要实现的操作集合。在一些语言环境里（例如C程序语言），一个数据结构不再使用时，需要显式地执行操作去销毁它，释放该结构占用的存储资源。由于Python系统能自动处理这方面事务，在这里就不必考虑销毁操作的问题。对于数据结构的一些实现方式，其中组合结构的容量有限，已经装满元素的结构将不能再加入新元素；如果采用这样的实现，就必须定义判断结构满的操作。

另外，一种具体的数据结构可能提供一些特殊的专门操作。例如，集合数据结构需要支持各种常用集合运算（求并集、交集、补集等）；图数据结构要提供判断结点是否相邻（两点间是否有边）的操作。

表抽象数据类型

现在考虑一个简单的表抽象数据类型，其中定义了一组最基本的操作。由于是计划用Python的类来实现这种类型，定义中的有些设计反映了这方面考虑。但从整体上看，这个抽象数据类型的描述是一般性的，并不依赖于Python语言。

ADT List:	#一个表抽象数据类型
List(self)	#表构造操作，创建一个新表
is_empty(self)	#判断self是否为一个空表
len(self)	#获得self的长度
prepend(self, elem)	#将元素elem加入表中作为第一个元素
append(self, elem)	#将元素elem加入表中作为最后一个元素
insert(self, elem, i)	#将elem加入表中作为第i个元素，其他元素的顺序不变
del_first(self)	#删除表中的首元素
del_last(self)	#删除表中的尾元素
del(self, i)	#删除表中第i个元素
search(self, elem)	#查找元素elem在表中出现的位置，不出现时返回-1
forall(self, op)	#对表中的每个元素执行操作op

在上面定义中，借用了Python语言及其类定义的一些表达方式。各种操作中的self参数表示被操作的List对象，其他参数用于为操作提供信息。其中的elem参数均表示参与操作的表元素，i表示表中元素的位置（下标），最后一个操作的op参数表示对表元素的某个具体操作，应该在实际调用遍历操作时提供。

根据前面的讨论，显然还可以为这个类型增加更多操作，但上面这些基本操作已能反映表操作的各方面特征。下面的讨论将基于这组操作进行。

显而易见，上面定义的抽象数据类型描述的是一类变动数据对象，其中的prepend、append等操作都是修改被操作的表对象本身，其操作效果体现在这个表的修改上。完全可以定义一个非变动的表抽象数据类型，为此只需把上面ADT里修改表的操作都变成构造新表的操作，在新表里体现各种操作的效果。这一工作留给读者自己完成。

3.1.3 线性表的实现：基本考虑

下面研究表数据结构的实现问题，这里主要需要考虑两方面情况：

- 1) 计算机内存的特点，以及保存元素和元素顺序信息的需要。
- 2) 各种重要操作的效率。如果程序里需要一个表，其创建操作只执行一次，但在其存在期间，可能反复地多次以各种方式使用，其中使用最频繁的操作通常包括表的性质判断函数 `is_empty` 等，还有（定位）访问、加入和删除元素，元素遍历等。在考虑表的实现结构时，需要特别考虑这些操作的实现效率。

另外，元素遍历就是依次访问表里的所有（或一批）元素，操作效率与访问元素的个数有关。由于是遍历所有元素，自然希望完成整个的复杂度不超过 $O(n)$ 。下面可以看到，加入 / 删除 / 访问元素的操作效率与表的实现结构有关。

基于各方面考虑，人们提出了两种基本的实现模型：

- 1) 将表中元素顺序地存放在一大块连续的存储区里，这样实现的表也称为顺序表（或连续表）。在这种实现中，元素间的顺序关系由它们的存储顺序自然表示。
- 2) 将表元素存放在通过链接构造起来的一系列存储块里，这样实现的表称为链接表，简称链表。

参考这两种基本实现模型，还可以开发出一些不同的具体实现技术。不同实现方式在一些方面各有长短，实际中可以根据具体情况和需要选择合适的技术，有关问题下面讨论。3.2 节首先讨论顺序表的实现，3.3 节讨论链表。

3.2 顺序表的实现

顺序表的基本实现方式很简单：表中元素顺序存放在一片足够大的连续存储区里，首元素（第一个元素）存入存储区的开始位置，其余元素依次顺序存放。元素之间的逻辑顺序关系通过元素在存储区里的物理位置表示（隐式表示元素间的关系）。

3.2.1 基本实现方式

最常见情况是一个表里保存的元素类型相同，因此存储每个表元素所需的存储量相同，可以在表里等距安排同样大小的存储位置。这种安排可以直接映射到计算机内存和单元，表中任何元素位置的计算非常简单，存取操作可以在 $O(1)$ 时间内完成。

设有一个顺序表对象，其元素存储在一片元素存储区，该存储区的起始位置（内存地址）已知为 l_0 。假定表元素编号从 0 开始（符合 Python 和许多编程语言的约定），元素 e_0 自然应存储在内存位置 $\text{Loc}(e_0)=l_0$ 。再假定表中一个元素所需的存储单元数为 $c=\text{size}(e)$ ，在这种情况下，就有下面简单的元素 e_i 的地址计算公式：

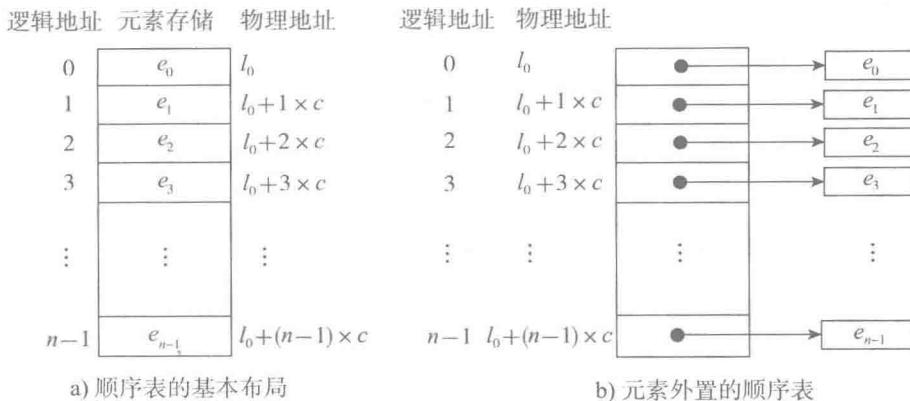
$$\text{Loc}(e_i) = \text{Loc}(e_0) + c \times i$$

表元素的大小 $\text{size}(e)$ 通常可以静态确定（例如，元素是整数或实数，或包含一组大小确定的元素的复杂结构），在这种情况下，计算机硬件将能支持高效的表元素访问。另一方面，如果表中元素的大小有可能不同，也不难处理，只要略微改变顺序表的存储结构，就仍然可以保证 $O(1)$ 时间的元素访问操作（下面很快就会介绍）。

顺序表元素存储区的基本表示方式（布局）如图 3.1a 所示，元素的下标是其逻辑地址，可以用上面公式计算出其物理地址（实际地址）。根据前面有关计算机内存结构的讨论，元素访

问是具有 $O(1)$ 复杂度的操作。

如果表元素的大小不统一，按照上面方案将元素顺序存入元素存储区，将无法通过统一公式计算元素位置。这时可以采用另一种布局方案，如图 3.1b 所示，将实际数据元素另行存储，在顺序表里各单元位置保存对相应元素的引用信息（链接）。由于每个链接所需的存储量相同，通过上述统一公式，可以计算出元素链接的存储位置，而后顺链接做一次间接访问，就能得到实际元素的数据了。注意，在图 3.1b 里的 c 不是数据元素的大小，而是存储一个链接所需的存储量，这个量通常很小。



a) 顺序表的基本布局

b) 元素外置的顺序表

图 3.1 顺序表的两种布局方案

在图 3.1b 所示的存储安排中，顺序表里保存的不是实际数据，而是找到实际数据的线索，这样的顺序表也被称为对实际数据的索引，这是最简单的索引结构，在后面章节里还会讨论其他更复杂的索引结构。

在确定了顺序表的基本表示方式之后，还需要进一步考虑线性表所需的各种操作的特点和需求，以及由它们带来的结构性问题。

线性表的一个重要性质是可以加入 / 删除元素，这也就是说，在一个表的存续期间，其长度（其中元素的个数）可能变化。这就带来了一个问题：在建立一个表时，应该安排多大的一块存储区？表元素存储块需要安排在计算机内存里，一旦分配就占据了内存里的一块区域，有了固定的大小（并因此确定了容量，即确定了元素个数的上限）。而且，该块的前后都可能被其他有用对象占据，存储块的大小不能随便变化，特别是无法扩充。

在建立一个顺序表时，一种可能是按建立时确定的元素个数分配存储。这种做法适合创建不变的顺序表，例如 Python 的 tuple 对象。如果考虑的是变动的表，就必须区分表中（当前的）元素个数和元素存储区的容量。在建立这种表时，一个合理的方法是分配一块足以容纳当前需要记录的元素的存储块，还应该保留一些空位，以满足增加元素的需要。

这样，在一个顺序表的元素存储区里，一般情况是保存着一些元素，还存在一些可以存放元素的空位。在这种情况下，需要约定元素的存放方式，通常把已有元素连续存放在存储区前面一段，空位都在后面。为了保证正确操作，就需要记录元素存储区的大小和当前的元素个数。这样，一个顺序表对象的完整信息如图 3.2 所示，这是一个具有可容纳 8 个元素的存储区，且当前存放了 4 个元素的顺序表。

容量	8
元素个数	4
0	1328
1	693
2	2529
3	154
4	
5	
6	
7	

图 3.2 一个顺序表

易见：元素存储区的大小决定了表的容量，这是分配存储块时确定的，对于这个元素存储区，容量值保持不变。而元素个数的记录要与表中实际元素个数保持一致，在表元素变化时（加入或删除元素时），都需要更新这一记录。

3.2.2 顺序表基本操作的实现

有了表容量和元素个数信息，表操作的实现方式已经很清楚了。下面假设用 \max 表示表的容量，即可能容纳的最大元素个数， num 表示当前元素个数。

创建和访问操作

创建空表： 创建空表时，需要分配一块元素存储，记录表的容量并将元素计数值设置为 0。图 3.3 表示一个容量为 8 的空表。注意，创建新表的存储区后，应立即将两个表信息记录域（ \max 和 num ）设置好，保证这个表处于合法状态。

表的各种访问操作都很容易实现。

简单判断操作： 判断表空或表满的操作很容易实现，即表空当且仅当 $\text{num}=0$ ，表满当且仅当 $\text{num}=\max$ 。这两个简单操作的复杂度都是 $O(1)$ 。

访问给定下标 i 的元素： 访问表中第 i 个元素时，需要检查 i 的值是否在表的合法元素范围内，即 $0 \leq i \leq \text{num}-1$ 。超出范围的访问是非法访问。位置合法时需要算出元素的实际位置，由给定位置得到元素的值。显然，这个操作不依赖于表中元素个数，因此也是 $O(1)$ 操作。

遍历操作： 要顺序访问表中元素，只需在遍历过程中用一个整数变量记录遍历达到的位置。每次访问表元素时，通过存储区开始位置和上述变量的值在 $O(1)$ 时间内可算出相应元素的位置，完成元素访问（取元素值或修改元素值）。找下一元素的操作就是加一，找前一元素的操作就是减一，遍历中要保证下标加一减一后的访问不超出合法范围。

查找给定元素 d 的（第一次出现的）位置： 这种操作称为检索或查找（searching）。在没有其他信息的情况下，只能通过用 d 与表中元素逐个比较的方式实现检索，称为线性检索。这里需要用一个基于下标的循环，每步用 d 与当前下标的表元素比较。下标变量控制循环的范围，从 0 开始至等于表中的元素个数时结束。在找到元素时返回元素下标，找不到时可以返回一个特殊值（例如 -1，因为它不会是元素的下标，很容易判断）。

查找给定元素 d 在位置 k 之后的第一次出现的位置： 与上面操作的实现方式类似，只是需要从 $k+1$ 位置的元素开始比较，而不是从位置 0 开始。

另一种需求与最后这两个操作类似：给定一个条件，要求找到第一个满足该条件的元素，或者某位置之后满足条件的下一个元素。条件可以是定义查找操作时确定的，更一般情况是允许在调用查找操作时提供一个描述条件的谓词函数。这两个操作的实现方式与上面两个操作类似，只是在其中不比较元素，而是检查条件是否成立。

最后几个操作都需要检查表元素的内容，属于基于内容的检索。数据存储和检索是一切计算和信息处理的基础，后面有关字典和检索的一章将深入研究这个问题。

总结一下：不修改表结构的操作只有两种模式，或者是直接访问，或者是基于一个整型变量，按下标循环并检查和处理。前一类操作都具有 $O(1)$ 的时间复杂度，后一类操作与访问的元素个数相关，复杂度为 $O(n)$ ，这里的 n 是表中元素个数。

\max	8
num	0
0	
1	
2	
3	
4	
5	
6	
7	

图 3.3 一个空表

变动操作：加入元素

下面考虑表的变动操作，即各种加入和删除元素的操作。其中在表的尾端加入和删除操作的实现很简单，在其他位置加入和删除的操作麻烦一些。

加入元素的各种情况参看图 3.4，假定是在前面图 3.2 所示的连续表状态下，以几种不同的方式加入一个新元素。

尾端加入新数据项：这种情况要求把新数据项存入当时表中的第一个空位，即下标 num 的位置。如果这时 $num = max$ ，即元素个数等于容量，表满了，操作就会失败（后面将介绍处理这种情况的其他技术）。如果表不满就直接存入元素，并更新表的元素计数值。显然，这是一个 $O(1)$ 操作。如果希望把元素简单地加入表中，没有其他要求，就应该采用尾端加入的方式，因为这样做操作最简单，效率最高。

图 3.4a 显示的是在图 3.2 的表尾端加入新元素 111 后的状态。

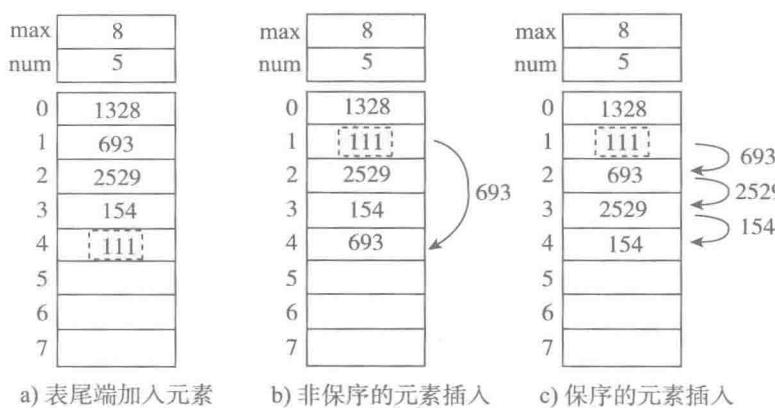


图 3.4 顺序表加入元素的几种处理方式

新数据存入元素存储区的第 i 个单元：这是一般情况，尾端加入是这里的特殊情况，即其中的 i 恰好等于 num ；首端加入也是它的特殊情况。在一般情况下，需要首先检查下标 i 是否为插入元素的合法位置，从下标 0 到 num （包括 num ，如果表不满）都是合法位置。确定合法后就可以考虑插入了。通常情况下位置 i 已经有数据（除非 i 等于 num ），要把新数据存入这里，又不能简单抛弃原有的数据，就必须把该项数据移走。移动数据的方式需要根据操作的要求确定。此外，操作前也要检查表是否已满，操作结束后的状态还应该保持有数据的单元在存储区前段连续，以及 num 的正确更新。

如果操作不要求维持原有元素的相对位置（不要求保序），可以采用简单处理方式：把原来位于 i 的元素移到位置 num ，放到其他已有元素之后，腾出位置 i 放入新元素，最后把元素计数值 num 加一。这一操作仍能在 $O(1)$ 时间完成。图 3.4b 显示了在原表中位置 1 插入数据 111 后的状态，原来在位置 1 的 693 被移到位置 4。

如果要求保持原有元素的顺序（保序），就不能像上面那样简单地腾出空位，必须把插入位置 i 之后的元素逐一下移，最后把数据项存入位置 i 。这样操作的开销与移动元素的个数成正比，一般而言受限于表中元素个数，最坏和平均情况都是 $O(n)$ 。图 3.4c 描绘的是完成在位置 1 插入数据 111 并要求保序操作后的状态。

变动操作：删除元素

现在考虑元素删除操作。下面示例以图 3.4c 中的表为操作前的状态。

尾端删除数据：删除表尾元素的操作非常简单，只需将元素计数值 num 减一。这样，原来的表尾元素不再位于合法的表下标范围，相当于删除了。对于图 3.4c 的表，删除表尾元素后的状态如图 3.5a 所示，从逻辑上说，最后的 154 已经看不到了。

删除位置 i 的数据：这是一般的定位删除，尾端删除是其特殊情况，其中 i 恰好等于 $\text{num} - 1$ 。首端删除也是它的特殊情况。对一般情况，首先要检查下标 i 是否为当前表中合法的元素位置，合法位置是 $0 \leq i \leq \text{num} - 1$ 。下标非法时可以采用引发异常的策略，也可以忽略它，什么也不做。确认下标合法后实际删除元素。

与加入元素的情况类似，这里也分为两种情况：如果没有保序要求，就可以简单处理，直接把当时位于 $\text{num} - 1$ 的数据项拷贝到位置 i ，覆盖原来的元素。如果有保序要求，就需要将位置 i 之后元素的数据逐项上移。删除后修改表的元素计数值。

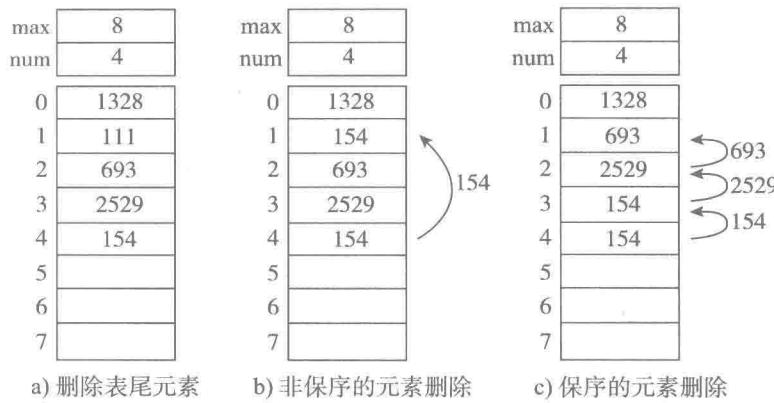


图 3.5 顺序表元素删除的几种处理方式

图 3.5b 给出了在非保序方式下删除下标 1 元素的结果。同样，由于元素计数值减一，最后的 154 也不再会被访问了。图 3.5c 给出了删除同一个元素，但保持其他元素原有顺序的操作结果，这使顺序表回到了图 3.2 的状态。

显然，尾端删除和非保序定位删除操作的时间复杂度是 $O(1)$ ，而保序定位删除的复杂度是 $O(n)$ ，因为其中可能移动一系列元素。

基于条件的删除：这种删除操作不是给定被删元素的位置，而是给出需要删除的数据项 d 本身，或者是给出一个条件要求删除满足这个条件的（一个、几个或者所有）元素。显然，这种操作与前面讨论过的检索有关，需要找到元素后删除它或它们。

这种操作也需要通过循环实现，循环中逐个检查元素，查到要找的元素后删除。显然，一般而言，这类操作都是线性时间操作，复杂度为 $O(n)$ 。

顺序表及其操作的性质

顺序表的一些操作比较简单，复杂度为常量的 $O(1)$ 。下面想仔细地重新考虑定位加入和删除操作的复杂度，首端和尾端加入 / 删除是它们的特例；这里只考虑保序的操作，前面说过这一对操作的复杂度是 $O(n)$ ，现在详细讨论这个问题。

假设给定的顺序表里有 n 个元素，在其中下标 i 的位置加入一项新数据，需要移动 $n-i$ 个元素；而删除下标为 i 的数据项需要移动 $n-i-1$ 个元素。再假设在位置 i 加入和删除元素的概率分别是 p_i 和 p'_i ，不难看到：执行加入操作中平均的元素移动次数是 $\sum_{i=0}^n (n-i) \cdot p_i$ ，删除操作

的平均元素移动次数是 $\sum_{i=0}^{n-1} (n-i-1) \cdot p'_i$ 。考虑平均复杂度时涉及各种情况的分布。作为最简单的情况，假设在所有位置操作出现的概率相同，可以算出这两个操作的平均时间复杂度是 $O(n)$ 。最坏情况的时间复杂度显然也是 $O(n)$ 。

如前所述，各种访问操作，如果其执行中不需要扫描表内容的全部或者一部分，其时间复杂度都是 $O(1)$ ，需要扫描表内容操作时间复杂度都是 $O(n)$ 。后一种情况的例子如根据元素的值进行检索，遍历表中元素求出它们的和（如果元素是数值）等。

表的顺序实现（顺序表）的总结：

- 优点： $O(1)$ 时间的（随机、直接的）按位置访问元素；元素在表里存储紧凑，除表元素存储区之外只需要 $O(1)$ 空间存放少量辅助信息。
- 缺点：需要连续的存储区存放表中的元素，如果表很大，就需要很大片的连续内存空间。一旦确定了存储块的大小，可容纳单元个数并不随着插入 / 删除操作的进行而变化。如果很大的存储区里只保存了少量数据项，就会有大量空闲单元，造成表内的存储浪费。另外，在执行加入或删除操作时，通常需要移动许多元素，效率低。最后，建立表时需要考虑元素存储区大小，而实际需求通常很难事先估计。

在下一小节里将会看到，最后这个“固定容量”的缺点实际上是可以解决的。

3.2.3 顺序表的结构

从前面的讨论中已经看到，一个顺序表的完整信息包括两部分，一部分是表中的元素集合，另一部分是为实现正确操作而需记录的信息，即那些有关表的整体情况的信息。根据前面的考虑，后一部分信息中包括元素存储区的容量和当前表中的元素个数两项。一个具体数据结构应该具有一个对象的整体形态。对于顺序表，就是要把上述两块信息关联起来，形成一个完整的对象。怎样把这两部分信息组织为一个顺序表的实际表示呢？这是实现顺序表时需要解决的第一个问题。

两种基本实现方式

由于表的全局信息只需要常量存储，对于不同的表，这部分的信息规模相同。根据计算机内存的特点，至少有两种可行表示方式，如图 3.6 所示。

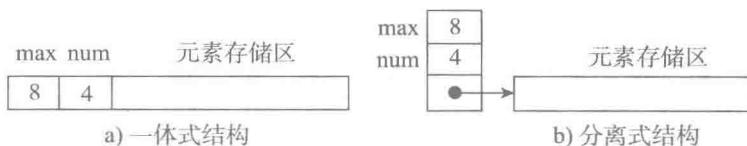


图 3.6 顺序表的两种实现方式

在图 3.6a 所示的实现方式中，存储表信息的单元与元素存储区以连续的方式安排在一块存储区里，几部分数据的整体形成一个完整的表对象。这种一体式实现比较紧凑，有关信息集中在一起，整体性强，易于管理。另外，由于连续存放，从表对象 L 出发，根据下标访问元素，仍然可以用与前面类似的公式计算位置，只需加一个常量：

$$\text{Loc}(e_i) = \text{Loc}(L) + C + i \times \text{size}(e)$$

其中 C 是数据成分 max 和 num 的存储量。

这种方式也有些缺点。例如，这里的元素存储区是表对象的一部分，因此不同的表对象大

小不一。另一个问题是创建后元素存储区就固定了，参看下面的讨论。

图 3.6b 给出了另一种实现方式，其中表对象里只保存与整个表有关的信息（容量和元素个数），实际元素存放在另一个独立的元素存储区对象里，通过链接与基本表对象关联。这样表对象的大小统一，但不同表对象可以关联不同大小的元素存储区。这种实现的缺点是一个表通过多个（两个）独立的对象实现，创建和管理工作复杂一些。

采用后一种实现方式，基于下标的元素访问仍然可以在常量时间完成。有关操作要分为两步：首先从表对象找到元素存储区，而后用前面公式计算存储位置。显然，这一操作代价比一体式实现中的代价略高，但仍然只需常量时间。

替换元素存储区

分离式实现的最大优点是带来了一种新的可能：可以在标识不变的情况下，为表对象换一块元素存储区。也就是说，表还是原来的表，其内容可以不变，但是容量改变了。在实际使用中，如果不断向一个顺序表里加入元素，最终一定会填满其元素存储区。如果该表采用一体式实现方式，此时再向表中加入元素的操作就会失败。由于表对象安排在内存，它的两边可能有其他对象，一般而言，不可能直接扩大其存储。要想继续程序的工作，就只能另外创建一个容量更大的表对象，把元素搬过去。但这是一个新对象，要修改当时使用原对象的所有位置，使之改用新对象，这个任务通常很难完成。

如果采用分离式技术实现，问题就很容易解决了。这时可以在不改变对象的情况下换一块更大的元素存储区，使加入元素操作可以正常完成。操作过程如下：

- 1) 另外申请一块更大的元素存储区。
- 2) 把表中已有的元素复制到新存储区。
- 3) 用新的元素存储区替换原来的元素存储区（改变表对象的元素区链接）。
- 4) 实际加入新元素。

经过这几步操作，还是原来那个表对象，但其元素存储区可以容纳更多元素，而所有使用这个表的地方都不必修改。这样就做出了一种可扩充容量的表，只要程序的运行环境（计算机系统）还有空闲存储，这种表就不会因为满了而导致操作无法进行。人们也把采用这种技术实现的顺序表称为动态顺序表，因为其容量可以在使用中动态变化。

后端插入和存储区扩充

一个较大的顺序表，通常都是从空表或者很小的表出发，通过不断增加元素而构造起来的。如果在创建表的时候不能确定这个表的最终大小，又需要保证操作的正常进行，采用动态顺序表技术就是最合理的选择。这样的表能随着元素的加入动态变化，自动满足实际应用的需要。现在考虑这种表的增长过程的时间复杂度问题。

假设随着操作的进行，动态顺序表的大小从 0 逐渐扩大到 n 。如果采用前端插入或一般定位插入的方式加入数据项，每次操作的时间开销都与表长度有关，总时间开销应该与 n 的平方成正比，也就是说，整个增长过程的时间复杂度为 $O(n^2)$ 。

现在考虑后端插入。由于不需要移动元素，一次操作本身的复杂度是 $O(1)$ 。但在这里还有一个情况需要考虑：在连续加入了一些数据项后，表的当前元素存储区终将被填满，这时就需要换一块存储区。更换存储区时需要复制表中的元素，整个复制需要 $O(m)$ 时间（其中 m 是当时的元素个数）。这样就出现了一个问题，即应该怎样选择新存储区的大小呢？这件事牵涉到空闲存储单元的量和替换存储的频度问题。

考虑一种简单策略：每次替换存储时增加 10 个元素存储位置。这种策略可称为线性增长，10 是增长的参数。假设表长（表中元素个数）从 0 不断增长到 1000，每加入 10 个元素就要换一次存储，复制当时的所有元素。总的元素复制次数是：

$$10 + 20 + 30 + \dots + 990 = 10 \times \sum_{i=1}^{99} i = 49500$$

对一般的 n ，很容易算出总的元素复制次数大约是 $1/20 \times n^2$ 。也就是说，虽然每次做尾端插入的代价是 $O(1)$ ，加上元素复制之后，一般而言，执行一次插入操作的平均代价还是达到了 $O(n)$ 。虽然这里出现的是一个分数因子，但结果还是不理想。

容易想到，这里总操作开销比较高，是因为在不断插入元素的过程中频繁替换元素存储区，而且每次替换复制的元素越来越多。修改替换时增加的空位数，例如把 10 改成 100，只能减小 n^2 的常量系数，不能带来本质的改进。应该考虑一种策略，其中随着元素数量的增加，替换存储区的频率不断降低。人们提出的一种策略是每次容量加倍。

假设表元素个数从 0 增加到 1024，存储区大小的序列是 1, 2, 4, 8, …, 1024，表增长过程中复制元素的次数是：

$$1 + 2 + 4 + \dots + 512 = \sum_{i=0}^9 2^i = 1023$$

其中 $9 = \log_2 1024 - 1$ 。对一般的 n ，在整个表的增长过程中，元素复制次数也是 $O(n)$ 。也就是说，采用存储增长的加倍策略和尾端插入操作，将一个表从 0 增长到 n ，插入操作的平均时间复杂度是 $O(1)$ 。不同替换策略带来了大不相同的操作复杂度。

请注意，后一个策略在操作复杂度上有明显优势，但也付出了代价。考虑在上面两种不同策略下，连续加入元素的增长过程：对于前一策略，无论 n 取什么值，元素存储区的最大空闲单元数是 9；而对于后一种策略，空闲单元可以达到差不多 $n/2$ 。也就是说，为获得时间上的优势，在这里需要付出空间代价。

还有一个问题值得注意：动态顺序表后端插入的代价不统一，大多数可以在 $O(1)$ 时间完成，但也会因为替换存储区而出现高代价操作。当然，高代价操作的出现很偶然，并将随着表的增大而变得越来越稀疏。另一方面，不间断地插入元素是这里的最坏情况，一般情况下插入和删除操作交替出现，替换存储区的情况会更稀少。但无论如何，高代价操作是可能出现的，应该注意这一情况，特别是在开发时间要求特别高的应用时。

缓解上述问题的一种可能性是增加一个设定容量的操作。这样，如果程序员知道下面一段操作的时间要求必须得到保证，就可以在操作前根据情况把表设定（修改）到一个足够大的容量，保证在关键计算片段中不会出现存储区替换。

3.2.4 Python 的 list

前几小节介绍了顺序表的各方面情况和实现技术。由于本书使用 Python，这里不准备给出在 Python 里定义顺序表的实际代码，因为 Python 的 list 和 tuple 就采用了顺序表的实现技术，具有前面讨论的顺序表的所有性质。本小节介绍这方面情况，以帮助学习者进一步理解 Python 的这两种类型，知道如何正确合理地使用它们。

tuple 是不变的表，因此不支持改变其内部状态的任何操作。在其他方面，它与 list 的性质类似。因此，下面将集中关注 list 的情况。

list 的基本实现技术

Python 标准类型 list 就是一种元素个数可变的线性表，可以加入和删除元素，在各种操作中维持已有元素的顺序。其重要的实现约束还有：

- 基于下标（位置）的高效元素访问和更新，时间复杂度应该是 $O(1)$ 。
- 允许任意加入元素（不会出现由于表满而无法加入新元素的情况），而且在不断加入元素的过程中，表对象的标识（函数 `id` 得到的值）不变。

这些基本约束决定了 list 的可能解决方案：

- 由于要求 $O(1)$ 时间的元素访问，并能维持元素的顺序，这种表只能采用连续表技术，表中元素保存在一块连续存储区里。
- 要求能容纳任意多的元素，就必须能更换元素存储区。要想在更换存储区时 list 对象的标识 (`id`) 不变，只能采用分离式实现技术。

在 Python 的官方实现中，list 就是一种采用分离式技术实现的动态顺序表，其性质都源于这种实现方式^Θ。Python 的 list 采用了前面介绍的元素存储区调整策略，如果需要反复加入元素，用 `lst.insert(len(lst), x)` 比在一般位置插入的效率高。Python 提供了另一等价写法 `lst.append(x)`，如果合适就应优先选用。

在 Python 的官方系统中，list 实现采用了如下的实际策略：在建立空表（或者很小的表）时，系统分配一块能容纳 8 个元素的存储区；在执行插入操作（`insert` 或 `append` 等）时，如果元素区满就换一块 4 倍大的存储区。但如果当时的表已经很大，系统将改变策略，换存储区时容量加倍。这里的“很大”是一个实现确定的参数，目前的值是 50000。引入后一个策略是为了避免出现过多空闲的存储位置。如前所述，通过这套技术实现的 list，尾端加入元素操作的平均时间复杂度是 $O(1)$ 。

一些主要操作的性质

list 结构的其他特点也由其顺序表实现方式决定。例如，由于其中一般位置的插入和删除操作都是保序的，要移动一些表元素，因此需要 $O(n)$ 时间。在其他位置加入元素时也要检查存储区是否已满，如果满了就需要换一块存储，把原有元素复制过去。这些操作可以优化，例如在复制元素的过程中完成新元素加入。

再看其他操作的情况。对那些所有序列（无论是否变动序列）都有的共性操作，复杂度由操作中需要考察的元素个数确定。

- `len()` 是 $O(1)$ 操作，因为表中必须记录元素个数，自然可以简单地取用。
- 元素访问和赋值，尾端加入和尾端删除（包括尾端切片删除）都是 $O(1)$ 操作。
- 一般位置的元素加入、切片替换、切片删除、表拼接（`extend`）等都是 $O(n)$ 操作。`pop` 操作默认为删除表尾元素并将其返回，时间复杂度为 $O(1)$ ，一般情况的 `pop` 操作（指定非尾端位置）为 $O(n)$ 时间复杂度。

Python 的一个问题是没有提供检查一个 list 对象的当前存储块容量的操作，也没有设置容量的操作，一切与容量有关的处理都由 Python 解释器自动完成。这样做的优点是降低编程负担，避免了人为操作可能引进的错误。但这种设计也限制了表的使用方式，前面提出的策略在这里也难以使用了。

^Θ 请注意：Python 还有非官方实现，其中 list 完全可以采用其他技术。如果使用这类系统，可能需要了解它的有关情况。但按照 Python 语言的设计，动态顺序表是 list 的最合理实现方式。

几个操作

下面考察 list 的几个特殊操作。

`lst.clear()` 清除表 `lst` 的所有元素，这应该是一个 $O(1)$ 操作，但具体实现的情况未见说明。有两种明显的可行做法：

- 简单地将表 `lst` 的元素计数值（表的长度记录）设置为 0。这样元素存储区里的所有元素都看不到了，自然应该看作空表。
- 另行分配一个空表用的存储区，原存储区直接丢弃。Python 解释器的存储管理器将自动回收这个存储块（可能在将来某个时候）。

第一种实现最简单，操作效率高，但不能真正释放元素存储区占用的存储。如果在执行 `clear` 之前这个表很长，执行操作后表里已经没有元素了，但仍占用原有的大块存储。第二种实现是再次从空表开始，如果这个表再增长到很大，过程中又要一次次更换存储区。

语句 `lst.reverse()` 修改表 `lst` 自身，将其元素顺序倒置，原来的首元素变成尾元素，其他元素的情况类似。很容易做出下面的实现（假设它定义在 `list` 类里，并假设元素存储区的域名为 `elements`）。显然，这个操作的复杂度是 $O(n)$ 。

```
def reverse(self):
    elems = self.elements
    i, j = 0, len(elems)-1
    while i < j:
        elems[i], elems[j] = elems[j], elems[i]
        i, j = i+1, j-1
```

标准类型 `list` 仅有的特殊操作（一个对象方法）是 `sort`，它对表中元素排序。这是一个变动操作，操作中移动表存储区里的元素。有关算法问题在第 9 章讨论。最好的排序算法的平均和最坏情况时间复杂度都是 $O(n \log n)$ 。Python 的排序算法就是这样。

3.2.5 顺序表的简单总结

顺序结构（技术）是组织一组元素的最重要方式。在顺序表结构中，直接采用顺序结构实现线性表，这种结构（技术）也是许多其他数据结构的实现基础。在后面章节里将会反复看到这方面的实例。

采用顺序表结构实现线性表：

- 最重要特点（优势）是 $O(1)$ 时间的定位元素访问。很多简单操作的效率也比较高。
- 这里最重要的麻烦是加入 / 删除等操作的效率问题。这类操作改变表中元素序列的结构，是典型的变动操作。由于元素在顺序表的存储区里连续排列，加入 / 删除操作有可能要移动很多元素，操作代价高。
- 只有特殊的尾端插入 / 删除操作具有 $O(1)$ 时间复杂度。但插入操作复杂度还受到元素存储区固定大小的限制。通过适当的（加倍）存储区扩充策略，一系列尾端插入可以达到 $O(1)$ 的平均复杂度。

顺序表的优点和缺点都在于其元素存储的集中方式和连续性。从缺点看，这样的表结构不够灵活，不容易调整和变化。如果在一个表的使用中需要经常修改结构，用顺序表去实现就不太方便，反复操作的代价可能很高。

还有一个问题也值得提出：如果程序里需要巨大的线性表，采用顺序表实现就需要巨大块的连续存储空间，这也可能造成存储管理方面的困难。

下一节中讨论的链接表在这两个方面都有优势。

3.3 链接表

本节考虑线性表的另一种实现技术。

3.3.1 线性表的基本需要和链接表

回忆一下线性表的定义，它就是一些元素的序列，维持着元素之间的一种线性关系。实现线性表的基本需要是：

- 能够找到表中的首元素（无论直接或者间接，这件事通常很容易做到）。
- 从表里的任一元素出发，可以找到它之后的下一个元素。

把表元素保存在连续的存储区里（顺序表），自然能满足这两个需求，其中元素间的顺序关联是隐含的。但是，要满足这两种需求，并不一定需要连续存储元素。显然，对象之间的链接也可以看作一种顺序关联，基于它也可以实现线性表。

实现线性表的另一种常用方式就是基于链接结构，用链接关系显式表示元素之间的顺序关联。基于链接技术实现的线性表称为链接表或者链表。

采用链接方式实现线性表的基本想法如下：

- 把表中的元素分别存储在一批独立的存储块（称为表的结点）里。
- 保证从组成表结构中的任一个结点可找到与其相关的下一个结点。
- 在前一结点里用链接的方式显式地记录与下一结点之间的关联。

这样，只要能找到组成一个表结构的第一个结点，就能顺序找到属于这个表的其他结点。从这些结点里可以看到这个表中的所有元素。

下面将只考虑在每个结点里只存储一个表元素的情况，在一个结点里存储多个表元素的实现方式请读者自己考虑。本章最后有些讨论。

链接技术是一类非常灵活的数据组织技术，实现链表有多种不同的方式。下面首先讨论最简单的单链表，其中在每个表结点里记录着存储下一个表元素的结点的标识（引用 / 链接）。后面将介绍另外一些结构的链表，它们各有所长，支持不同的需要。在下面的讨论中，将把“存储着下一个表元素的结点”简称为“下一结点”。

实际上，要建立（复杂的）链接结构需要功能强大的存储管理技术支持。在 C 语言等编程语言里，开发这方面的程序时涉及的技术细节很多，编程中也容易出错，完成的链表使用起来也比较麻烦。在 Python 等新型编程语言里，对于与链接结构有关的高级技术的支持非常全面，在这里可以很方便地建立和使用各种复杂的链接结构。实际上，利用 Python 语言编写简单的程序时已经广泛用到了各种链接结构（它们由 Python 语言默认提供和处理），后面会看到这方面的一些例子。

3.3.2 单链表

单向链接表（下面将简称为单链表或者链表）的结点是一个二元组，形式如图 3.7a 所示，其表元素域 elem 保存着作为表元素的数据项（或者数据项的关联信息），链接域 next 里保存同一个表里的下一个结点的标识。

在最常见形式的单链表里，与表里的 n 个元素对应的 n 个结点通过链接形成一条结点链，如图 3.7b 所示。从引用表中首结点的变量（图 3.7b 中变量 p）可以找到这个表的首结点，从表

中任一结点可以找到保存着该表下一个元素的结点（表中下一结点），这样，从 p 出发就能找到这个表里的任一个结点。

要想掌握一个单链表，就需要（也只需要）掌握这个表的首结点，从它出发可以找到这个表里的第一个元素（即在这个表结点里保存的数据，保存在它的 elem 域中），还可以找到这个表里的下一结点（有关信息保存在这个结点的 next 域中）。按照同样的方式继续下去，就可以找到表里的所有数据元素。

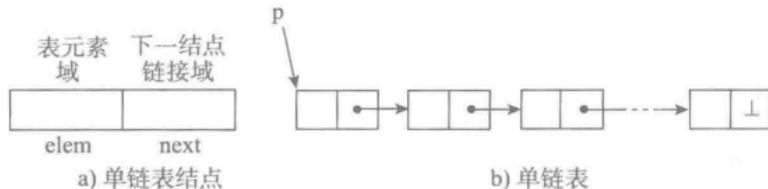


图 3.7 单链表的结点和单链表

也就是说，为了掌握一个表，只需要用一个变量保存着这个表的首结点的引用（标识或称为链接）。今后将把这样的变量称为表头变量或表头指针。

总结一下：

- 一个单链表由一些具体的表结点构成。
- 每个结点是一个对象，有自己的标识，下面也常称其为该结点的链接。
- 结点之间通过结点链接建立起单向的顺序联系。

为了表示一个链表的结束，只需给表的最后结点（表尾结点）的链接域设置一个不会作为结点对象标识的值（称为空链接），在 Python 里自然可以用系统常量 None 表示这种情况，在图 3.7 里用接地符号“上”表示链表结束，下面将一直这样表示。

通过判断一个（域或变量）值是否为空链接，可知是否已到链表的结束。在顺序扫描表结点时，应该用这种方法确定操作是否完成。如果一个表头指针的值是空链接，就说明“它所引用的链表已经结束”，这是没有元素就已结束，说明该表是空表。

在实现链表上的算法时，并不需要关心在某个具体的表里各结点的具体链接值是什么（虽然保存在表结构里的值都是具体的），只需要关心链表的逻辑结构。对链表的操作也只需要根据链表的逻辑结构考虑和实现。

为方便下面的讨论，现在定义一个简单的表结点类：

```
class LNode:
    def __init__(self, elem, next_=None):
        self.elem = elem
        self.next = next_
```

这个类里只有一个初始化方法，它给对象的两个域赋值。方法的第二个参数用名字 next_，是为了避免与 Python 标准函数 next 重名。这也是 Python 程序中命名的一个惯例。第二个参数还提供了默认值，只是为了使用方便。

基本链表操作

现在考虑链表的基本操作及其实现方式。

创建空链表：只需要把相应的表头变量设置为空链接，在 Python 语言中将其设置为 None，在其他语言里也有惯用值，例如有的语言里用 0 作为这个特殊值。

删除链表：应丢弃这个链表里的所有结点。这个操作的实现与具体的语言环境有关。在一些语言（如 C 语言）里，需要通过明确的操作释放一个个结点所用的存储。在 Python 语言中这个操作很简单，只需简单地将表指针赋值为 None，就抛弃了链表原有的所有结点。Python 解释器的存储管理系统会自动回收不用的存储。

判断表是否为空：将表头变量的值与空链接比较。在 Python 语言中，就是检查相应变量的值是否为 None。

判断表是否满：一般而言链表不会满，除非程序用完了所有可用的存储空间。

加入元素

现在考虑给单链表加入元素的操作，同样有插入位置问题，可以做首端插入、尾端插入或者定位插入。不同位置的操作复杂度可能不同。

首先应该注意，在链表里加入新元素时，并不需要移动已有的数据，只需为新元素安排一个新结点，然后根据操作要求，把新结点连在表中的正确位置。也就是说，插入新元素的操作是通过修改链接，接入新结点，从而改变表结构的方式实现的。

表首端插入：首端插入元素要求把新数据元素插入表中，作为表的第一个元素，这是最简单的情况。这一操作需要通过三步完成：

- 1) 创建一个新结点并存入数据（图 3.8a 表示要向表头变量 head 的链表加入新首元素 13，为它创建了新结点，变量 q 指着该结点。这是实际插入前的状态）。

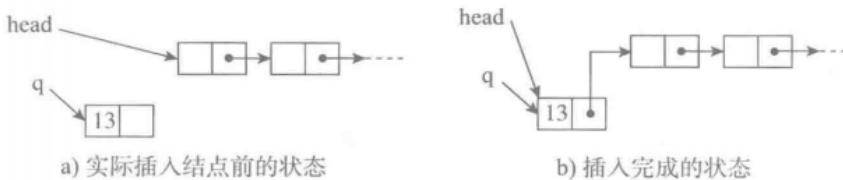


图 3.8 单链表，新元素插人在表头

- 2) 把原链表首结点的链接存入新结点的链接域 next，这一操作将原表的一串结点链接在刚创建的新结点之后。
- 3) 修改表头变量，使之指向新结点，这个操作使新结点实际成为表头变量所指的结点，即表的首结点（图 3.8b 表示设置链接的这两步操作完成后的状态，新结点已成为 head 所指链表的首结点，13 成为这个表的首元素。注意，示意图中链接指针的长度和形状都不表示任何意义，只有图中的链接指向关系有意义）。

注意，即使在插入前 head 指向的是空表，上面三步也能正确完成工作。这个插入只是一次安排新存储和几次赋值，操作具有常量时间复杂度。

示例代码段：

```
q = LNode(13)
q.next = head.next
head = q
```

一般情况的元素插入：要想在单链表里的某位置插入一个新结点，必须先找到该位置之前的那个结点，因为新结点需要插入在它的后面，需要修改它的 next 域。如何找到这个结点的问题将在后面讨论，先看已经找到了这个结点之后怎样插入元素。

设变量 pre 已指向要插入元素位置的前一结点，操作也分为三步：

- 1) 创建一个新结点并存入数据 (图 3.9a 是实际插入前的状态)。
- 2) 把 pre 所指结点 next 域的值存入新结点的链接域 next, 这个操作将原表在 pre 所指结点之后的一段链接到新结点之后。
- 3) 修改 pre 的 next 域, 使之指向新结点, 这个操作把新结点链入被操作的表, 整个操作完成后的状态如图 3.9b 所示。



图 3.9 单链表, 新元素插入在 pre 之后

注意, 即使在插入前 pre 所指结点是表中最后一个结点, 上述操作也能将新结点正确接入, 作为新的表尾结点。

示例代码段:

```
q = LNode(13)
q.next = pre.next
pre.next = q
```

删除元素

删除链表中元素, 也可通过调整表结构删除表中结点的方式完成。这里也区分两种情况: 删除表头结点的操作可以直接完成, 删除其他结点也需要掌握其前一结点。

删除表首元素: 删除表中第一个元素对应于删除表的第一个结点, 为此只需修改表头指针, 令其指向表中第二个结点。丢弃不用的结点将被 Python 解释器自动回收。

图 3.10a 展示了这个操作。示例代码只有一个语句:

```
head = head.next
```

一般情况的元素删除: 一般情况删除须先找到要删元素所在结点的前一结点, 设用变量 pre 指向, 然后修改 pre 的 next 域, 使之指向被删结点的下一结点。

图 3.10b 展示了这个操作。实际删除代码也只有一个语句:

```
pre.next = pre.next.next
```

显然, 这两个操作都要求被删结点存在。

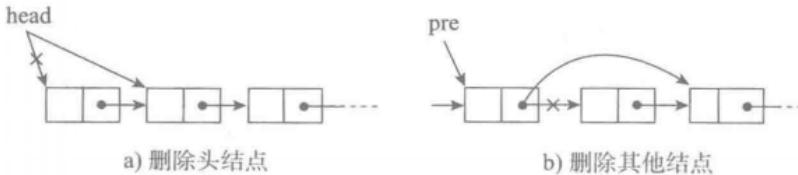


图 3.10 单链表结点删除

如果在其他编程语言里删除结点, 还可能要自己释放存储。Python 的自动存储管理机制能自动处理这方面的问题, 使编程工作更简单, 也保证了安全性。

扫描、定位和遍历

在一般情况下插入和删除元素，都要求找到被删结点的前一结点。另外，程序里也可能需要定位链表中的元素、修改元素、逐个处理其中元素等。这些操作都需要检查链表的内容，实际上是检查表中的一些（或全部）结点。

由于单链表只有一个方向的链接，开始情况下只有表头变量在掌握中，所以对表内容的一切检查都只能从表头变量开始，沿着表中链接逐步进行。这种操作过程称为链表的扫描，这种过程的基本操作模式是：

```
p = head
while p is not None and 还需继续的其他条件:
    对p所指结点里的数据做所需操作
    p = p.next
```

循环的继续（或结束）条件、循环中的操作由具体问题决定。循环中使用的辅助变量 `p` 称为扫描指针。注意，每个扫描循环必须用一个扫描指针作为控制变量，每步迭代前必须检查其值是否为 `None`，保证随后操作的合法性。这与连续表的越界检查类似。

上面表扫描模式是最一般的链表操作模式，下面介绍几个常用操作的实现。

按下标定位：按 Python 惯例，链表首结点的元素应看作下标 0，其他元素依次排列。确定第 i 个元素所在结点的操作称为按下标定位，可以参考表扫描模式写出：

```
p = head
while p is not None and i > 0:
    i -= 1
    p = p.next
```

假设循环前变量 `i` 已有所需的值，循环结束时可能出现两种情况：或者扫描完表中所有结点还没有找到第 i 个结点，或者 `p` 所指结点就是所需。通过检查 `p` 值是否为 `None` 可以区分这两种情况。显然，如果现在需要删除第 k 个结点，可以先将 `i` 设置为 $k-1$ ，循环后检查 `i` 是 0 且 `p.next` 不是 `None` 就可以执行删除了。

按元素定位：假设需要在链表里找到满足谓词 `pred` 的元素。同样可以参考上面的表扫描模式，写出的检索循环如下：

```
p = head
while p is not None and not pred(p.elem):
    p = p.next
```

循环结束时或者 `p` 是 `None`；或者 `pred(p.elem)` 是 `True`，找到了所需元素。

完整的扫描称为遍历，这样做通常是需要对表中每个元素做一些事情，例如：

```
p = head
while p is not None:
    print(p.elem)
    p = p.next
```

这个循环依次输出表中各元素。只以条件 `p is not None` 控制循环，就能完成一遍完整的遍历。同样模式可用于做其他工作。

链表操作的复杂度

总结一下链表操作的时间复杂度。

- 创建空表： $O(1)$ 。
- 删除表：在 Python 里是 $O(1)$ 。当然，Python 解释器做存储管理也需要时间。

- 判断空表: $O(1)$ 。
- 加入元素 (都需要加一个 T (分配) 的时间):
 - 首端加入元素: $O(1)$ 。
 - 尾端加入元素: $O(n)$, 因为需要找到表的最后结点。
 - 定位加入元素: $O(n)$, 平均情况和最坏情况。
- 删除元素:
 - 首端删除元素: $O(1)$ 。
 - 尾端删除元素: $O(n)$ 。
 - 定位删除元素: $O(n)$, 平均情况和最坏情况。
 - 其他删除: 通常需要扫描整个表或其一部分, $O(n)$ 。

扫描、定位或遍历操作都需要检查一批表结点, 其复杂度受到表结点数的约束, 都是 $O(n)$ 操作。其他在工作中有此类行为的操作也至少具有 $O(n)$ 时间复杂度。

求表的长度

在使用链表时, 经常需要求表的长度, 为此可以定义一个函数:

```
def length(head):
    p, n = head, 0
    while p is not None:
        n += 1
        p = p.next
    return n
```

这个函数采用表扫描模式, 遍历表中所有结点完成计数。

显然, 这个求表长度的算法所用时间与表结点个数成正比, 具有 $O(n)$ 时间复杂度。

实现方式的变化

以求表的长度为例, 如果程序经常需要调用上面函数, $O(n)$ 复杂度就可能成为效率问题。如果表很长, 执行该函数就可能造成可察觉的停顿。解决这个问题的一种方法是改造单链表的实现结构, 增加一个表长度记录。显然, 这个记录不属于任何表元素, 是有关表的整体的信息。表示这件事的恰当方法是定义一种链表对象, 把表的长度和表中的结点链表都作为这个表对象的数据成分, 如图 3.11 所示。

图中变量 p 指向表对象, 这个对象的一个数据域记录表中元素个数 (图中的 20 表示这个表当时有 20 个结点), 另一个域引用着该表的结点链。采用了这种表示方式, 求表长度的操作就可以简单返回元



图 3.11 增加了表对象的单链表

素计数值的值。但另一方面, 这种表的每个变动操作都需要维护计数值。从整体看有得有失。这种调整消除了一个线性时间操作, 可能在一些应用中很有意义。

3.3.3 单链表类的实现

前一节讨论了链表的各种操作, 以及重要的链表扫描模式。本节基于上面讨论, 研究如何所用 Python 语言实现一个链表类。

3.3.2 节定义了链表的结点类 LNode, 下面是一段简单使用代码:

```
llist1 = LNode(1)
p = llist1
```

```

for i in range(2, 11):
    p.next = LNode(i)
    p = p.next

p = llist1
while p is not None:
    print(p.elem)
    p = p.next

```

第一个循环逐步建立起一个链表，结点元素取整数 1 到 10 的值，第二个循环输出表中各结点的元素值。后一循环是前面讨论的扫描循环的实例，前一个 for 循环用迭代器控制，把一个新结点链接在已有的表结点链的最后。

根据 Python 语言的规定，这里的 p is not None 可以简单地只写一个 p。但为清晰起见，本书中没做这种简化。

自定义异常

为能合理地处理一些链表操作中遇到的错误状态（例如，方法执行时遇到了无法操作的错误参数），首先为链表类定义一个新的异常类：

```

class LinkedListUnderflow(ValueError):
    pass

```

这里把 `LinkedListUnderflow` 定义为标准异常类 `ValueError` 的子类，准备在空表访问元素等场合抛出这个异常。在这些情况下抛出 `ValueError` 也没问题，但定义了自己的异常类，就可以写专门的异常处理器，在一些情况下可能有用。

Python 语言提供了一套预定义的标准异常，每个异常都是一个类，按继承关系形成了一个层次结构，具体情况可以参看 2.4 节。按 Python 语言的规定，程序里所有自定义的异常类必须是标准异常类的派生类，更确切地说，它们都应该继承异常类 `Exception` 或者它的直接或间接派生类。如果在程序里需要定义异常，就应该从标准异常类中选一个最接近所需的异常，在定义自己的异常时继承这个异常类。

LList 类的定义，初始化函数和简单操作

现在基于结点类 `LNode` 定义一个单链表对象的类，在这种表对象里只有一个引用链接结点的 `_head` 域，初始化为 `None` 表示建立的是一个空表。注意，这个链表基本上是图 3.11 的设计，但表对象里没有元素计数，增加元素计数域的变形留作练习。判断表空的操作检查 `_head`；在表头插入数据的操作是 `prepend`，它把包含新元素的结点链接在最前面；操作 `pop` 删除表头结点并返回这个结点里的数据：

```

class LList:
    def __init__(self):
        self._head = None

    def is_empty(self):
        return self._head is None

    def prepend(self, elem):
        self._head = LNode(elem, self._head)

    def pop(self):
        if self._head is None: # 无结点，引发异常
            raise LinkedListUnderflow("in pop")
        e = self._head.elem

```

```

    self._head = self._head.next
    return e

```

这里把 `LList` 对象的 `_head` 域作为对象的内部表示，不希望外部使用。再强调一次，Python 程序中属于这种情况的域按照习惯用单个下划线开头的名字命名。上面定义里的几个操作都很简单，只有 `pop` 操作需要检查对象的状态，表中无元素时引发异常。

后端操作

在链表的最后插入元素，必须先找到链表的最后一个结点。其实现首先是一个扫描循环，找到相应结点后把包含新元素的结点插入在其后。下面是定义：

```

def append(self, elem):
    if self._head is None:
        self._head = LNode(elem)
        return
    p = self._head
    while p.next is not None:
        p = p.next
    p.next = LNode(elem)

```

这里需要区分两种情况：如果原表为空，引用新结点的就应该是表对象的 `_head` 域，否则就是已有的最后结点的 `next` 域。两种情况下需要修改的数据域不一样。许多链表变动操作都会遇到这个问题，只有表首端插入 / 删除可以统一处理。

现在考虑删除表中最后元素的操作，也就是要删除最后的结点。前面说过，要从单链表中删除一个结点，就必须找到它的前一结点。在尾端删除操作里，扫描循环应该找到表中倒数第二个结点，也就是找到 `p.next.next` 为 `None` 的 `p`。下面定义的 `pop_last` 函数不仅删去表中最后元素，还把它返回（与 `pop` 统一）。

在开始一般性扫描之前，需要处理两个特殊情况：如果表空没有可返回的元素时应该引发异常。表中只有一个元素的情况需要特殊处理，因为这时应该修改表头指针。一般情况是先通过循环找到位置，取出最后结点的数据后将其删除：

```

def pop_last(self):
    if self._head is None: # 空表
        raise LinkedListUnderflow("in pop_last")
    p = self._head
    if p.next is None: # 表中只有一个元素
        e = p.elem
        self._head = None
        return e
    while p.next.next is not None: # 直到p.next是最后结点
        p = p.next
    e = p.next.elem
    p.next = None
    return e

```

其他操作

`LList` 类的下一个方法是找到满足给定条件的表元素。这个方法有一个参数，调用时通过参数提供一个判断谓词，该方法返回第一个满足谓词的表元素。显然，这个操作也需要采用前面的基本扫描模式。定义如下：

```

def find(self, pred):
    p = self._head
    while p is not None:

```

```

if pred(p.elem):
    return p.elem
p = p.next

```

最后一个方法非常简单，但实际中可能很有用。在开发一个表类的过程中，人们会经常想看看被操作的表的当时情况：

```

def printall(self):
    p = self._head
    while p is not None:
        print(p.elem, end='')
        if p.next is not None:
            print(', ', end='')
        p = p.next
    print('')

```

最后的 print 只是为了输出一个换行符号，其参数是空串。

下面是一段简单的使用链表的代码：

```

mlist1 = LList()
for i in range(10):
    mlist1.prepend(i)
for i in range(11, 20):
    mlist1.append(i)
mlist1.printall()

```

这里先建立一个空表，而后通过循环在表首端加入 9 个元素（9 个整数），又通过循环在表尾端加入 9 个元素（整数），最后顺序输出表里的所有元素。

表的遍历

人们把线性表一类的对象称为汇集（collection）对象，它们本身是对象，其中又包含着一组元素对象。显然，Python 内置的 list、tuple 等都是汇集对象类。对于汇集对象，最典型的使用方法之一就是逐个使用其中的元素，这种操作称为遍历。上面 printall 是一个完成特定工作的遍历操作，它逐个输出表中元素。实际使用链接表时，可能需要对其元素做各种操作，因此，链表类应该以某种方式支持这一类使用。

传统的遍历方式是为汇集对象类定义一个遍历函数，它以一个操作为参数，将其作用到汇集对象的每个元素上。例如，可以考虑为 LList 定义下面方法：

```

def for_each(self, proc):
    p = self._head
    while p is not None:
        proc(p.elem)
        p = p.next

```

proc 的实参应该是可以作用于表元素的操作函数，它将被作用于每个表元素。假如 list1 是以字符串为元素的表，下面语句将一行一个地输出这些字符串：

```
list1.foreach(print)
```

在经典编程语言的程序里，常常可以看到上面这种传统技术，其优点是比较规范，主要缺点是使用不够灵活，不容易与其他编程机制配合使用。为了解决使用中的不便，人们经常用 lambda 表达式定制出在这里使用的操作参数。

遍历一组数据是程序中最重要的一类工作方式，新型编程语言都为这类操作提供了专门的支持工具，特别是希望能把对用户定义类型的处理纳入统一的编程形式中。Python 语言为内部汇集类型提供的遍历机制是迭代器，标准使用方式是放在 for 语句头部，在循环体中逐个处

理汇集对象的元素，这样就可以很方便地实施各种操作。LList 是汇集类型，因此也应该为它提供类似的操作方式，使之能用在标准的操作框架里。

在 Python 语言里，要想定义迭代器，最简单的方式是定义生成器函数。在类里也可以定义具有这种性质的方法。下面是为 LList 类定义对象的一个迭代器：

```
def elements(self):
    p = self._head
    while p is not None:
        yield p.elem
        p = p.next
```

有了这个方法，在代码里就可以写：

```
for x in list1.elements()
    print(x)
```

以这种方式处理链接表，在形式上与处理 list 或 tuple 差不多。

筛选生成器

如果深入思考，就会看出前面的 find 方法有点尴尬，经常不适用，因为用它只能取得满足 pred 的第一个元素。但在被操作的表里可能有多个这样的元素。

在经典的编程语言里，这个问题没有很自然的解决方案，但在 Python 语言（和一些新型语言）中，这件事很容易处理：只需简单改造 find 方法，定义一个功能与之类似的迭代器。把这种迭代器放在 for 语句头部，就能在循环体里逐个处理满足条件的表元素了。由于方法的意义改变了，原方法名 find 已经不合适，下面将新方法命名为 filter，表示要基于给定的谓词筛选出表中满足 pred 的元素。方法定义的改动非常简单：

```
def filter(self, pred):
    p = self._head
    while p is not None:
        if pred(p.elem):
            yield p.elem
        p = p.next
```

这个方法不再是一个普通的方法了。它也是一个生成器方法，使用方式与前面的 elements 类似，只是需要多提供一个谓词参数。

本节定义了一个简单的链表类，它提供了一些有用的操作。从这些操作的定义中可以看到链表操作函数定义的许多特点。可以根据需要为 LList 增加其他操作，如定位插入和删除等，这些都留作读者的练习。作为提示，请读者查看 Python 基本类型 list 的操作，看看 LList 里缺了什么，考虑如何为 LList 类实现它们。

3.4 链表的变形和操作

链表并非只有前面讨论的一种。实际上，人们提出了许多形式不同的链表设计，它们各有优点和适用环境。下面首先介绍单链表的简单变形，而后再介绍双链表。

3.4.1 单链表的简单变形

即使同为单链表（即每个结点只有一个指针域），也存在多种不同的设计，而且完全可以根据需要和认识修改已有的设计。现在从前面定义的简单单链表的一个缺点出发，讨论一种修改后的设计。

前面单链表实现有一个缺点：尾端加入元素操作的效率低，因为这时只能从表头开始查找，直至找到表的最后一个结点，而后才能链接新结点。

在实际中，需要频繁地从表的两端加入元素的情况也很多见。现在的问题是，能不能改进表的设计，提高后端插入操作的效率？

图 3.12 给出了一种可行设计，其中的表对象增加一个表尾结点引用域。有了这个域，只需常量时间就能找到尾结点，在表尾加入新结点的操作就可能做到 $O(1)$ 。

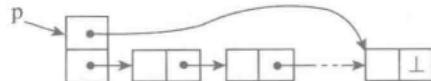


图 3.12 带有尾结点引用的单链表

应该注意到：链表的这一新设计与前面单链表的结构近似，这种结构变化应该不影响非变动操作的实现，只影响到表的变动操作。在这种情况下，有可能重用前面定义（或者前面定义的一部分）吗？

通过继承和扩充定义新链表类

实际中经常遇到这样的问题：需要的程序部件和某个已有部件很像，但也有些不同。在这种情况下，一个简单想法是把原来的代码复制一份，在其基础上修改。但是，一旦复制了代码，引进了重复片段，很多麻烦就不可避免地出现了。维护两份类似代码很麻烦，不但两者都可能需要修改，还可能需要确保维护修改的一致性。

第 2 章讲过，面向对象编程技术为解决这方面的问题提供了支持，允许基于已有类（基类）定义新类（派生类）。这种派生类将继承其基类的所有功能（数据域和方法），可以定义新的数据域，定义新的方法，还可以重新定义基类里已定义的方法（覆盖已有方法）。下面通过把链表的新变形作为派生类的例子，展示如何做这件事情。

链表类 `LList` 提供了（具有图 3.12 所示新结构的）新链表类对象的许多功能，应该尽可能设法利用。用面向对象编程的说法，即应该考虑把新链表类定义为 `LList` 的派生类。这样，这个新类就能继承 `LList` 的所有非变动操作。实际上，作为派生类会继承基类的所有操作，但原有的变动操作不符合需要，必须重新定义。

从数据域看，新类的对象需要增加一个尾结点引用域，在这个类的初始化函数里应正确设置这个域，变动操作都可能修改这个域，需要重新定义。

第 2 章最后介绍了如何在定义新类的时候指定其基类。将现在要定义的类命名为 `LList1`，这个类的定义的头部就应该是：

```
class LList1(LList):
    .... # 方法定义和其他
```

实际上，任何用户定义类都是某个类的派生类。如果定义时不注明基类，按 Python 的规定，这个类自动以公共类 `object` 作为基类。也就是说，前面定义的类 `LNode` 和 `LList` 都以 `object` 作为基类，是 `object` 的派生类。前面定义的异常类也继承了已有的异常类，是特定标准异常类的派生类。

初始化和变动操作

下面考虑类的方法定义，首先是初始化方法。注意，`LList1` 类的对象也是 `LList` 类的对象，是想作为链表使用的，因此这种对象里应该有 `LList` 对象的所有数据域（虽然这里只有一个 `_head`，但其他基类可能有很多），在 `LList1` 的初始化函数中，应该首先初始化 `LList` 对象的那些数据域。做这件事，最合理而且方便的方式就是对 `self` 对象调用 `LList` 类的初始化函数。现在还要初始化一个尾结点引用域。由于是作为内部数据域，用 `_rear` 作为域名，将它也初始化为 `None`：

```
def __init__(self):
    LList.__init__(self)
    self._rear = None
```

考虑前端插入操作。加入包含新数据的结点，操作方式与 LList 里一样，但现在还要考虑尾结点引用域的设置：如果原来是空表，新加人的第一个结点也是最后一个结点。这说明需要重新定义 prepend 覆盖原来的操作。下面是一种定义方式：

```
def prepend(self, elem):
    self._head = LNode(elem, self._head)
    if self._rear is None: # 是空表
        self._rear = self._head
```

这里用检查 _rear 是否为 None 的方式判断空表，实际上要求在表空的时候，不仅 _head 是 None，_rear 也必须是 None。相应的删除操作也需要保证这一点。

回忆一下，从 LList 继承的判断表空操作只检查 _head。同一个类里的其他操作应该与它一致。下面定义虽然比上面的长一点，但更合适：

```
def prepend(self, elem):
    if self._head is None:
        self._head = LNode(elem, self._head)
        self._rear = self._head
    else:
        self._head = LNode(elem, self._head)
```

在一个类里，不同的方法在处理同一事项上应保持一致，下面有专门讨论。

增加了尾结点引用后，可以直接找到尾结点，在其后增加结点的操作也能更快完成。这个操作的性能是本次设计修改的主要收获。注意，在链表操作定义中，通常都需要区分被修改的是头变量（域）的情况还是一般情况。函数定义：

```
def append(self, elem):
    if self._head is None: # 是空表
        self._head = LNode(elem, self._head)
        self._rear = self._head
    else:
        self._rear.next = LNode(elem)
        self._rear = self._rear.next
```

现在应该考虑弹出元素的操作 pop，有趣的是它不需要重新定义。之所以能这样，与统一用 _head 为 None 判断空表有关。请读者自己分析。

最后是弹出末元素的操作。若采用新的表结构，这个函数没有变简单，反而稍微麻烦了一点。现在删除了尾结点之后还需更新 _rear。由于确定了统一用 _head 的值判断表空，删除最后一个结点使表变空时，不需要给 _rear 赋值 None：

```
def pop_last(self):
    if self._head is None: # 是空表
        raise LinkedListUnderflow("in pop_last")
    p = self._head
    if p.next is None: # 表中只有一个元素
        e = p.elem
        self._head = None
        return e
    while p.next.next is not None: # 直到p.next是最后结点
        p = p.next
    e = p.next.elem
    p.next = None
    self._rear = p
```

```
    return e
```

下面是一段使用这个类的代码：

```
mlist1 = LList1()
mlist1.prepend(99)
for i in range(11, 20):
    mlist1.append(randint(1,20))

for x in mlist1.filter(lambda y: y % 2 == 0):
    print(x)
```

新类的基本使用形式与 `LList` 相同，变化的只是后端插入操作的效率。最后一个语句输出表 `mlist1` 里的所有偶数，其中用一个 `lambda` 表达式描述筛选条件。

类设计的内在一致性

现在结合上面的例子，简单讨论类设计中的一个重要原则。

一个类定义是一个整体，它描述了一种程序对象。类定义比较复杂，其中可以有许多成分，特别是可能定义了许多方法。每个方法定义是类定义中的一个独立片段，编程语言（Python）对不同方法之间的关系并没有任何约束，也不对这样一组方法定义做任何相互关系方面的检查。但是，作为同一个类定义的成分，这些方法需要相互协调，保持一致，才能保证所定义的程序对象有意义。这件事需要编程序的人考虑和保证。

以类 `LList1` 为例，前面讨论了两种可能设计：一种设计要求在表空的时候 `_head` 和 `_rear` 的值都是 `None`，另一种设计只要求这时 `_head` 为 `None`。基于两种设计都能正确实现这个类，但是一旦选定了一种设计，类中的所有方法都必须遵照这种设计来定义，包括所有的插入和删除操作。

一般情况，在设计一个类时总需要考虑一套统一的规则。类的初始化方法建立起的对象应满足这些规则，操作也不能破坏规则，这样定义的类才是有效的。

当然，不同规则可能影响类定义的细节和复杂程度。在上面例子里采用了后一种设计，也是因为它更简单，需要维护的关系少一点。还考虑到基类的已有设计，尽可能与其保持一致，以便更多地利用已有的功能。

3.4.2 循环单链表

单链表的另一常见变形是循环单链表（简称循环链表），其中最后一个结点的 `next` 域不用 `None`，而是指向表的第一个结点，如图 3.13a 所示。但如果仔细考虑，就会发现在链表对象里记录表尾结点更合适（如图 3.13b），这样可以同时支持 $O(1)$ 时间的表头 / 表尾插入和 $O(1)$ 时间的表头删除。当然，由于循环链表里的结点连成一个圈，哪个结点算是表头或表尾，主要是概念问题，从表的内部形态上无法区分。

现在考虑实现一个循环单链表类，采用图 3.13b 的表示。下面只讨论几个典型操作，它们反映了循环链表各方面的特点，更多操作留给读者作为练习。

循环单链表操作与普通单链表的差异就在于扫描循环的结束控制。易见，一些不变操作的实现也要修改，如 `printall`。



图 3.13 循环单链表

这种表对象只需一个数据域 `_rear`，它在逻辑上始终引用着表的尾结点。前端加入结点，就是在尾结点和首结点之间加入新的首结点，尾结点引用不变。通过尾结点引用很容易实现这个操作。另一方面，尾端加入结点也是在原尾结点之后（与首结点之间）插入新结点，只是插入后要把它作为新的尾结点，因此需要更新尾结点引用。这两个操作都要考虑空表插入的特殊情况。对于输出表元素的操作，关键在于循环结束的控制。下面实现中比较扫描指针与表头结点的标识，到达了表头结点就结束。前端弹出操作也很容易实现，后端弹出操作（留作练习）需要通过一个扫描循环确定位置。

循环单链表类

下面循环单链表类定义只实现了几个典型方法，供参考：

```
class LCLList: # 循环单链表类
    def __init__(self):
        self._rear = None

    def is_empty(self):
        return self._rear is None

    def prepend(self, elem): # 前端插入
        p = LNode(elem)
        if self._rear is None:
            p.next = p # 建立一个结点的环
            self._rear = p
        else:
            p.next = self._rear.next
            self._rear.next = p

    def append(self, elem): # 尾端插入
        self.prepend(elem)
        self._rear = self._rear.next

    def pop(self): # 前端弹出
        if self._rear is None:
            raise LinkedListUnderflow("in pop of CLLList")
        p = self._rear.next
        if self._rear is p:
            self._rear = None
        else:
            self._rear.next = p.next
        return p.elem

    def printall(self): 输出表元素
        if self.is_empty():
            return
        p = self._rear.next
        while True:
            print(p.elem)
            if p is self._rear:
                break
            p = p.next
```

前面简单链表的演示代码也可以用在这里，只需要修改类名。

3.4.3 双链表

单链表只有一个方向的链接，只能做一个方向的扫描和逐步操作。即使增加了尾结点引

用，也只能支持 $O(1)$ 时间的首端元素加入 / 删除和尾端加入。如果希望两端插入和删除操作都能高效完成，就必须修改结点（从而也是链表）的基本设计，加入另一方向的链接。这样就得到了双向链接表，简称双链表。有了结点之间的双向链接，不仅能支持两端的高效操作，一般结点操作也会更加方便。当然，这样做也需要付出代价：每个结点都需要增加一个链接域，增加的空间开销与结点数成正比，是 $O(n)$ 。如果每个表结点里的数据规模比较大，新增加的开销可能就显得不太重要了。

为了支持首尾两端的高效操作，双链表应该采用图 3.14 所示的结构，包含一个尾结点引用域。易见，从双链表中任一结点出发，可以直接找到其前后的相邻结点（都是 $O(1)$ 操作）。而对单链表而言，只能方便地找到下一个结点，要找前一结点，就必须从表头开始逐一检查（通过一次扫描）。

可以直接找到当前结点的前后结点，使得双链表的许多操作都很容易地进行。下面假定结点的下一结点引用域是 `next`，前一结点引用域是 `prev`。

结点操作

先考虑结点删除。实际上，只要掌握着双链表里一个结点，就可以把它从表中取下，并把其余结点正确链接好。图 3.15 说明了这个操作。示例代码是：

```
p.prev.next = p.next
p.next.prev = p.prev
```

这两个语句使 `p` 所指结点从表中退出，其余结点保持顺序和链接。如果要考虑前后可能无结点的情况，只需增加适当的条件判断。

在任一结点的前后加入结点的操作也很容易局部完成，只需掌握确定加入位置的这个结点。易见，加入一个结点需要做四次引用赋值，请读者自己考虑。

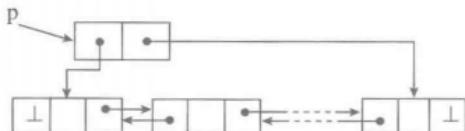


图 3.14 带有尾结点引用的双链表

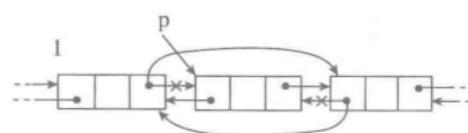


图 3.15 双链表的结点删除

双链表类

现在考虑定义一个双链表类。

首先，双链表的结点与单链表不同，因为结点里多了一个反向引用域。可以考虑独立定义，或者在 `LNode` 类的基础上派生。这里用派生方式定义：

```
class DLNode(LNode): # 双链表结点类
    def __init__(self, elem, prev=None, next_=None):
        LNode.__init__(self, elem, next_)
        self.prev = prev
```

使用的方式与链表结点类似。

下面定义了一个双链表类，从带首尾结点引用的单链表类 `LList1` 派生，采用图 3.14 所示的结构。空表判断和 `find`、`filter`、`printall` 方法都可以继承，它们执行中只使用 `next` 方向的引用，用在双链表上也完全正确。

类中的几个变动操作都需要重新定义，因为它们需要设置前一结点引用 `prev`。可以看到，这里的首端和尾端的插入 / 删除方法中都不需要循环，因此都能在常量时间内完成。如

果仔细检查下面的两对方法，可以发现它们几乎是对称的，其中的 `_head` 与 `_rear` 对应，`next` 与 `prev` 对应。

```
class DLLList(LList1):          # 双链表类
    def __init__(self):
        LList1.__init__(self)

    def prepend(self, elem):
        p = DLNode(elem, None, self._head)
        if self._head is None:      # 空表
            self._rear = p
        else:                      # 非空表，设置prev引用
            p.next.prev = p
        self._head = p

    def append(self, elem):
        p = DLNode(elem, self._rear, None)
        if self._head is None:      # 空表插入
            self._head = p
        else:                      # 非空表，设置next引用
            p.prev.next = p
        self._rear = p

    def pop(self):
        if self._head is None:
            raise LinkedListUnderflow("in pop of DLLList")
        e = self._head.elem
        self._head = self._head.next
        if self._head is not None: # _head空时不需要做任何事
            self._head.prev = None
        return e

    def pop_last(self):
        if self._head is None:
            raise LinkedListUnderflow("in pop_last of DLLList")
        e = self._rear.elem
        self._rear = self._rear.prev
        if self._rear is None:
            self._head = None      # 设置 _head保证is_empty正确工作
        else:
            self._rear.next = None
        return e
```

前面各种链接表的演示代码在这里都能工作，只是有些操作的性能改善了，从代码执行的输出结果上看不到什么不同。

循环双链表

双链表也可以定义为循环链表，也就是说，让表尾结点的 `next` 域指向表的首结点，而让表首结点的 `prev` 域指向尾结点。如图 3.16 所示。易见，在这种表里，各结点的 `next` 引用形成了向下一结点方向的引用环，而各结点的 `prev` 引用形成向前一结点方向的引用环。两个环相向而行。实际上，图中尾结点指针并不必要。

有意思的是，由于在这种表里存在双向链接，无论是掌握着表的首结点还是尾结点，都能高效实现首尾两端的元素加入 / 删操作 ($O(1)$ 复杂度)。

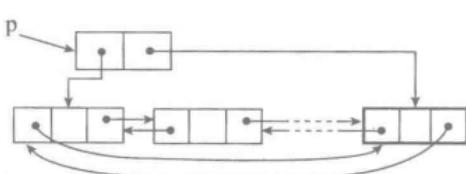


图 3.16 循环双链表

这种链表的实现留作练习。

3.4.4 两个链表操作

前面几小节讨论了链表的几种重要变形，并且展示了如何在各种不同的结点链接结构上实现几个最基本的操作。现在讨论两个更有趣一点的链表操作，从它们的实现中可以看到链表操作的更多特点。

链表反转

首先考虑表元素的反转，也就是 Python 中 list 类 reverse 操作的工作。在 3.2.4 节最后给出了一个顺序表上的示例性实现。反转顺序表中元素的算法用两个下标，通过逐对交换元素位置并把下标向中间移动的方式工作，直到两个下标碰头时操作完成。

同样的操作模式可以用在双链表上，因为双链表中结点有 next 和 prev 两个引用，同时支持两个方向的扫描操作，这个问题留作个人练习。在单链表上也可以实现这种操作方式，实现元素反转。但是单链表不支持从后向前找结点，要找前一结点，只能从头开始做，这就使算法需要 $O(n^2)$ 时间。请读者想想还有别的办法吗？

请注意，对顺序表而言，改变其中元素的顺序的方法只有一种，就是在表中搬动元素。而对于链表，实际上存在着两种方法：可以在结点之间搬动元素，也可以修改结点的链接关系，通过改变结点的链接顺序来改变表元素的顺序。

根据前面讨论，通过搬动元素的方式实现单链表中的元素反转很不方便，而且效率很低。下面考虑基于修改链接的方法，看看有什么可能。

对于单链表，有一个情况很重要：在首端插入 / 删除元素或结点是最方便的操作，只需要 $O(1)$ 时间。实现单链表操作时，最好能在首端进行。

如果不间断向一个表的首端插入结点，最早放进去的结点将在表的最后（即尾结点），而从表的首端取下结点，最后取下的是尾结点。也就是说，从一个表的首端不断取下结点，将其加入另一个表的首端，就形成了一个反转过程。取下和加入操作都是 $O(1)$ 的，总时间开销是 $O(n)$ ，所以这个过程就是一个高效的反转算法。

下面的函数作为 LList 类的一个方法，最后把反转后的结点链赋给表对象的 _head 域。链表类 LList1 继承 LList 时，必须重新定义这个方法，因为它需要反转操作在完成基本工作后正确设置 _rear：

```
def rev(self):
    p = None
    while self._head is not None:
        q = self._head
        self._head = q.next # 摘下原来的首结点
        q._next = p
        p = q           # 将刚摘下的结点加入 p 引用的结点序列
    self._head = p       # 反转后的结点序列已经做好，重置表头链接
```

在实际生活中也经常能见到类似的过程，如通过调度把一列火车车厢的顺序颠倒过来。另外，如果桌上有一摞书，一本本拿下来放到另一处，叠成另一摞，也是这个操作的实例。

链表排序

现在考虑一个更复杂的操作：对链表中的元素排序。把一组物品按某种顺序排列是在真实世界中经常需要做的工作，数据处理中也经常需要将数据排序。

排序是一种数据序列操作，希望对序列中数据项的位置做一些调整，使它们按某种特定的顺序排列。由于这是数据处理中经常需要做的事情，人们对排序问题做了很多研究，开发出许多重要算法。本书第9章将专门研究这个问题。本节只是想通过排序问题，讨论表操作的一些技术。下面讨论中假定数据可以用“`>`”和“`<=`”等关系运算符比较，希望完成的是将序列（表）里的数据按“`<=`”关系从小到大排序。

Python的list类型有一个sort方法，可以完成list的元素排序。如果lst的值是一个list类型的对象，`lst.sort()`将把lst中元素从小到大进行排序。另外，也可以用标准函数sorted对各种序列进行排序，`sorted(lst)`生成一个新的表（list类型的对象），其中元素是lst的元素排序的结果。

链表里也存储着数据的序列，因此也经常有对其中元素排序的需求。下面讨论单链表的排序问题，以及相关的算法和实现。

这里只准备考虑一种简单的排序算法，称为插入排序。其基本想法是：

- 1) 在操作过程中维护一个排好序的序列片段。初始时该段只包含一个元素，可以是任何一个元素，因为一个元素的序列总应该认为是排序的。
- 2) 每次从尚未处理的元素中取出一个元素，将其插入已排序片段中的正确位置，保持插入后的序列片段仍然是正确排序的。
- 3) 当所有元素都加入了排序的片段时，排序工作完成。

先看一个顺序表（list）排序函数，帮助理解插入排序的操作过程。首先要考虑已排序段的安排，放在哪里。由于未排序部分越来越小，每做一次上面的操作2减少一个元素；已排序部分越来越大，每次增加一个元素。因此，可以让这两个部分共用原来的表，例如在表前部积累排序片段，不需要额外的存储。

排序过程中，被排序表的状态如图3.17a所示：下标i之前的段已经从小到大排序，从i开始的段尚未处理，下一步考虑位置i的元素d的插入问题，并正确完成这一工作。这样一次处理一个元素后i值加一，直至i的值超出表的右端时排序完成。

一个元素的处理也需要通过一个循环。在这个循环中，需要维持已排序元素的相对顺序，并最终确定d的正确插入位置。循环开始时取出d，使位置i变为空位。循环中的状态如图3.17b所示：新下标变量j记录空位，并逐步左移。每次迭代将j-1位置的元素与d比较，如果d较小，就把位于j-1的元素右移到j位置，并将j值减一（表示空位左移了）。这样在j到i之间就会积累一段大于d的元素。反复做到位置j之前元素不大于d时，将d放入空位。显然，直至i的子序列仍保持有序。将i加一后又回到了图3.17a的状态。



图3.17 顺序表插入排序过程的示意图

这个算法中需要嵌套的两重循环：

```
def list_sort(lst):
    for i in range(1, len(lst)): # 开始时片段[0:1]已排序
        x = lst[i]
        j = i
        while j > 0 and lst[j-1] > x:
```

```

lst[j] = lst[j-1] # 反序逐个后移元素至确定插入位置
j -= 1
lst[j] = x

```

现在考虑单链表的排序算法。注意：由于这里只有 next 链接，扫描指针只能向下一个方向移动，不能从后向前查找结点（或找元素）。另外，如前所述，这里也存在两种可能完成排序的做法：移动表中元素，或者调整结点之间的链接关系。下面将考虑两个算法，它们分别采用这两种技术完成排序，用的都是插入排序方法。

首先考虑基于移动元素的单链表排序算法。采用插入排序方法，在这里也是每次拿一个未排序元素，在已排序序列中找到正确位置后插入。但是，由于处理的是单链表，为了有效操作，算法中只能按下一个的方向检查和处理表元素。

下面算法工作中的基本状态如图 3.18 所示。其中扫描指针 crt 指向当前考虑的结点（假设这里的表元素为 x），在一个大循环中每次处理一个表元素并前进一步。对一个元素的处理分两步完成：第一步从头开始扫过小于或等于 x 的表元素，直至确定了图 3.18 中已排序段里标出虚线的位置，找到了第一个大于 x 的表元素；第二步是做一系列“倒换”，把 x 放入正确位置，并将其他表元素后移。下面函数实现了这个过程：

```

def sort1(self):
    if self._head is None:
        return
    crt = self._head.next                                # 从首结点之后开始处理
    while crt is not None:
        x = crt.elem
        p = self._head
        while p is not crt and p.elem <= x: # 跳过小元素
            p = p.next
        while p is not crt:                  # 倒换大元素，完成元素插入的工作
            y = p.elem
            p.elem = x
            x = y
            p = p.next
        crt.elem = x                                # 回填最后一个元素
        crt = crt.next

```

函数里比较技术性的一段是倒换大元素的循环，每次迭代取出一个结点里的数据，然后把手头数据 x 存入，前进一步。请仔细考察其中的操作和顺序。

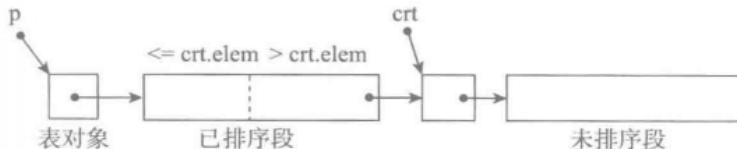


图 3.18 单链表插入排序（移动元素方法）的中间状态

现在考虑通过调整链接的方式实现插入排序。这种方法的操作过程比较好理解，就是一个一个取下链表结点，将其插入一段元素递增的结点链中的正确位置。

这个过程由下面的方法实现。如果被处理的表为空或只包含一个元素，它自然是排序的，工作完成。表更长时就需要处理。函数里用 rem 记录除第一个元素之外的结点段，然后通过循环把这些结点逐一插入 _head 关联的排序段。

函数的内层循环在排序段查找 rem 结点的插入位置。这里用了两个扫描指针 p 和 q，它

们亦步亦趋地前进，直到 p 所指结点的元素更大或已到排序段尾，这时结点 rem 应该插入 q 和 p 之间。随后的条件语句分别处理表头插入和一般情况插入，最后连接好排序段并将 rem 推进一步。大循环结束时全部结点都插入排序段，工作完成。

```
def sort(self):
    p = self._head
    if p is None or p.next is None:
        return

    rem = p.next
    p.next = None
    while rem is not None:
        p = self._head
        q = None
        while p is not None and p.elem <= rem.elem:
            q = p
            p = p.next
        if q is None:
            self._head = rem
        else:
            q.next = rem
        q = rem
        rem = rem.next
        q.next = p
```

在这个定义里，需要特别注意最后三个赋值语句。赋值的次序不能出错，否则将导致被处理的表段丢失，就不可能完成排序工作了。

两个排序函数都假定定义在 `LList` 类里，用在其他地方时需适当修改。

3.4.5 不同链表的简单总结

3.3 节和本节介绍了多种不同的链表结构，现在对它们做一个简单的总结，其中讨论时间复杂度时用的 n 均指表的长度。

- 基本单链表包含一系列结点，通过一个方向的链接构造起来。它支持高效的（ $O(1)$ 的）前端（首端）插入和删除操作，定位操作或尾端操作都需要 $O(n)$ 时间。
- 增加了尾结点引用域的单链表可以很好地支持首端 / 尾端插入和首端弹出元素，它们都是 $O(1)$ 时间复杂度的操作，但不能支持高效的尾端删除。
- 循环单链表也能支持高效的表首端 / 尾端插入和首端弹出元素。在这种表上扫描，需要特别注意结束判断问题。
- 双链表中每个结点都有两个方向的链接，因此可以高效地找到前后结点。如果有尾结点引用，两端插入和删除操作都能在 $O(1)$ 时间完成。循环双链表的性质类似。
- 对于单链表，遍历和数据检索操作都只能从表头开始，需要 $O(n)$ 时间。对于双链表，这些操作可以从表头或表尾开始，复杂度不变。与它们对应的两种循环链表，遍历和检索可以从表中任何一个地方开始，但要注意结束条件。

链接表有一些重要的优点，分析如下：

- 表结构是通过一些链接起来的结点形成的，结点（及其中表元素）之间的顺序由链接关系决定，链接可以修改，因此表的结构很容易调整和修改。
- 不需要修改结点里的数据元素或移动它们，只通过修改结点之间的链接，就能灵活地修改表的结构和数据排列方式。例如，加入 / 删除一个或多个元素，翻转整个表，重排

表中元素顺序，将一个表根据需要划分为两个表或多个表，等等。

- 整个表由一些小的存储块构成，比较容易安排和管理。用 Python 编写程序时，这些问题由解释器负责，程序员不必处理，但了解情况也很重要。

链接表也有一些明显的缺点，主要是一些操作的代价比较大：

- 定位访问（基于位置找到元素）需要线性时间，这是与顺序表相比的最大劣势。
- 简单单链表上的尾端操作需要线性时间。增加一个尾指针，可以将尾端插入变成常量时间操作，但仍不能有效实现尾端删除。双链表通过在每个结点里增加第二个链接，可以实现两端的高效插入和删除。
- 要找当前元素的前一元素，必须从头开始扫描表结点。这种操作应尽量避免。双链表可以解决这个问题，但每个结点要付出更多存储代价。
- 为存储一个表元素，需要多用一个链接域，这是实现链接表的存储代价。双链表可以提高链表操作的灵活性，但需要增加两个链接域。

3.5 表的应用

本节通过一个简单的例子展示表结构的使用。这里给出了同一个问题的几种不同实现，其中使用了不同的表结构。

3.5.1 Josephus 问题和基于“数组”概念的解法

Josephus 问题是数据结构教材中一个常见的实例：假设有 n 个人围坐一圈，现在要求从第 k 个人开始报数，报到第 m 个数的人退出。然后从下一个人开始继续报数并按同样规则退出，直至所有人退出。要求按顺序输出各出列人的编号。

本节考虑第一种解决方法：基于 Python 的 list 和固定大小的“数组”概念，也就是说，在这里把 list 看作元素个数固定的对象，只修改元素的值，不改变表的结构（不用加入或删除元素的操作）。这相当于摆了一圈 n 把椅子，人可以走但椅子在那里且位置不变。基于这种设计可以有多种实现方法。下面的方法是给每个人赋予一个编号，没有人的情况用 0 表示，各 list 的元素记录这些编号。算法梗概：

- 初始：
 - 建立一个包含 n 个人（的编号）的表。
 - 找到第 k 个人，从那里开始。
- 处理过程中采用把相应表元素修改为 0 的方式表示已出列，反复做：
 - 数 m 个（尚在坐的）人，遇到表的末端就转回下标 0 继续。
 - 把表示第 m 个人的表元素修改为 0。
- n 个人出列即结束。

下面算法里用 i 表示数组下标，其初值取 $k-1$ ，内层循环中每次加一并通过取模 n 保持 i 的取值范围正确。大循环一次迭代出列一人，共计执行 n 次迭代。循环体里的 count 计数直至 m （通过内层循环），计数中跳过空椅子。其他部分都很容易理解。

```
def josephus_A(n, k, m):
    people = list(range(1, n+1))

    i = k-1
    for num in range(n):
```

```

count = 0
while count < m:
    if people[i] > 0:
        count += 1
    if count == m:
        print(people[i], end="")
        people[i] = 0
    i = (i+1) % n
if num < n-1:
    print(", ", end="")
else:
    print("")
return

```

这里函数名中的 A 表示数组实现。函数的使用实例如下：

```
josephus_A(10, 2, 7)
```

定义这个函数的主要麻烦是表下标的（循环）计数 i 和表中有效元素的计数 $count$ 相互脱节。计数变量 $count$ 达到 m 时输出表元素下标 i ，并将该元素置 0，表示这个人已经出列。输出了最后一个元素后让 `print` 输出一个换行符。

这个算法的复杂度不容易分析。内层循环的总迭代次数是 i 加 1 的次数，应该与 n 和 m 有关，因为 m 影响到找到下一元素的操作步数。考虑两个特殊情况：

- 当 $m=1$ 时，每次内循环只执行一次迭代，总的时间开销是 $O(n)$ 。
- 当 $m=n$ 时，先考虑计算到最后表中只剩一个元素的情况。不难看到，内层循环需要遍历整个表 n 遍，每一遍只能把 $count$ 的值加 1，因此，为了删除这个元素，花费的时间就是 $O(n^2)$ 。整个计算中 i 加 1 的次数大约是

$$n \times \left(\frac{n}{n} + \frac{n}{n-1} + \dots + \frac{n}{3} + \frac{n}{2} + \frac{n}{1} \right) \approx n^2 \times \log n$$

可见这个算法分析中的麻烦，这里不进一步讨论了。

3.5.2 基于顺序表的解

现在考虑另一个算法：把保存人员编号的 `list` 按表的方式处理，一旦确定了应该退出的人，就将表示其编号的表元素从表中删除。

这样，随着计算的进行，所用的表将变得越来越短。下面用 `num` 表示表的长度，每退出一人，表的长度 `num` 减一，至表长度为 0 时计算工作结束。采用这种想法和设计，表中的元素全 是有效元素（不再出现表示没人的 0），元素计数与下标计数得到统一，所以下标更新可以用 $i=(i+m-1)\%num$ 统一描述。

基于这些想法写出的程序非常简单：

```

def josephus_L(n, k, m):
    people = list(range(1, n+1))

    num, i = n, k-1
    for num in range(n, 0, -1):
        i = (i + m-1) % num
        print(people.pop(i),
              end=" " if num > 1 else "\n"))
    return

```

函数体是一个简单的循环计数，很容易理解。其中用 `list` 的 `pop` 操作删除元素。最后

的输出语句值得说一下，其中用了一个条件表达式确定一次输出的结束字符串：如果整个计算还没完成（num 的值大于 1 时），就在输出了出列人的编号后加上一个逗号；如果所有人的编号都已输出，就输出一个换行符。

函数调用的形式与前一函数相同，如：josephus_L(10, 2, 7)。

这个算法的复杂度比较容易考虑。虽然函数循环只执行 n 次，但由于输出语句调用了 list 的 pop 操作，它需要线性时间，所以算法复杂度是 $O(n^2)$ 。由于这里直接加 m 计算位置，所以算法的复杂度与 m 无关。

3.5.3 基于循环单链表的解

现在考虑基于循环单链表实现一个算法。

从形式上看，循环单链表可以很直观地表示围坐一圈的人，顺序数人头可以自然地反映为在循环表中沿着 next 链扫描，一个人退出可以用删除相应结点的操作模拟。在这样做之后，又可以沿着原方向继续数人头了。

根据上面分析，这个算法应该分为两个阶段：

- 1) 建立包含指定个数（和内容）的结点的循环单链表，这件事可以通过从空表出发，在尾部逐个加入元素的方式完成。
- 2) 循环计数，找到并删除应该退出的结点。

具体实现可以有多种方式，例如：为原循环单链表增加一个循环数数的函数，然后写一段程序建立所需的循环单链表，并完成操作。下面的实现采用了另一种方式，即基于循环单链表类派生出一个专门的类，用其初始化方法完成全部工作。

派生类 Josephus 的实现中没有增加“当前人指针”一类设置，采用了另一种考虑，把计数过程看作人圈的转动（结点环的转动）。这个类里定义了新方法 turn，它将循环表对象的 rear 指针沿 next 方向移 m 步（相当于结点环旋转）。

这个类的初始化函数首先调用基类 LCList 的初始化函数建立一个空表，然后通过一个循环建立包含 n 个结点和相应数据的初始循环表。最后的循环反复调用 turn 方法，找到并逐个弹出结点，输出结点里保存的编号。

```
class Josephus(LCList):
    def turn(self, m):
        for i in range(m):
            self._rear = self._rear.next

    def __init__(self, n, k, m):
        LCList.__init__(self)
        for i in range(n):
            self.append(i+1)
        self.turn(k-1)
        while not self.is_empty():
            self.turn(m-1)
            print(self.pop(),
                  end="\n" if self.is_empty() else ", ")
```

虽然这里用了一个类作为基础，其使用的方式却与调用一个函数类似，建立这个类的对象就是完成一次计算，如 Josephus(10, 2, 7)。所创建对象本身并不重要。

这个算法的时间复杂度比较容易考虑：建立初始表的复杂度是 $O(n)$ ，后面循环的算法复杂度为 $O(m \times n)$ ，因为总共做了 $m \times n$ 次一步旋转，每次是 $O(1)$ 。

本章总结

本章讨论了线性表的概念、其基本运算和抽象数据类型，以及两种实现技术（顺序表和链接表实现）。线性表是一种比较简单的数据结构，是 n 个数据元素的有限序列，各元素在表中有特定的排列位置和前后顺序关系。

顺序表

在线性表的顺序存储实现（顺序表）中，元素存储在一块大存储区里，它们之间的逻辑顺序关系通过实际存储位置直接反映。顺序表里只需要存放数据元素本身的信息，因此存储密度大，空间利用率高。另外，元素的存储位置可以基于下标通过简单的算式算出，因此可以在 $O(1)$ 时间内随机存取。这些是顺序存储结构的优点。

另一方面，由于采用连续方式存储元素，顺序表技术的灵活性不足。为了有效支持元素的插入和删除，顺序表中需要预留一些空闲的空间。如果顺序表的长度变化较大，通常需要按最大需要安排存储空间。这些情况带来的空置存储位置也形成额外的存储开销。此外，在顺序表里插入和删除都可能需要移动许多元素，操作的代价通常都比较高。这些是线性表顺序存储结构的缺点。特殊情况是表的尾端插入和删除为 $O(1)$ 操作。

由于采用连续方式存储元素，需要首先安排确定大小的存储空间。这样，如果程序运行中不断加入元素，最终会填满整个元素空间，这时再插入元素的操作就无法完成了。动态顺序表技术可以解决这个问题，避免由于表满而导致程序被迫终止。采用这种技术出现了一个新问题：原本高效的后端插入操作，也可能由于扩存而比较耗时。这种偶发的情况可能影响程序的行为，在开发对时间效率要求高的程序时，需要特别注意。另一情况是扩存策略，需要在平均操作效率和闲置存储量之间权衡。

链接表

在链接实现（链接表）里，表元素保存在一批较小的存储块里，通过显式的链接将这些块连成一串，形成一种链式结构。结点的链接结构直接反映元素的顺序关系。

在链接表中，表元素之间的顺序由它们所在的结点之间的链接显式表示，因此表结点可以任意安排位置，灵活地调整结构，通过调整链接完成表元素的插入、删除和各种重整顺序的操作。这些情况说明，链接表实现的灵活性较强，操作的实现方式更加灵活多样。对于一些应用，这种灵活性是非常重要的。

也应该看到另一些情况。为了实现链接表，每个结点都需要增加一个链接域，付出额外的存储空间代价。但是表中并不需要预留空闲位置，有多少元素就建立多少结点。对于一个链表，能直接掌握的只是其中一两个结点（首结点，以及可能的尾结点），要访问其他结点，只能顺着链接一步步去查找。因此链表中的元素不能随机访问，按位置访问的代价很高。由于这种情况，使用链表的最合理方式是前端操作和按元素顺序访问。增加了尾结点引用后，可以支持受限的后端操作，包括表元素访问和新元素插入，但不包括删除。引入相反方向的链接（双链表的前向链接），可以增加一种数据访问顺序，也使表中间结点的操作更加方便，但并没有改变链接表的本质。

链表的另一个特点是存在很多变形。为突破简单单链表带来的操作限制，可以引入尾结点指针，建立循环链接，或者为每个结点增加一个反向链接（形成双链表）。每种变形都能使一些操作的实现更直接，提高操作的效率；但要付出存储的代价，也给一些操作的实现带来新问题，如尾指针的维护、前向链接的维护等。

讨论

实际上，顺序表与链接表并不是决然相对的两种技术。本章讨论的链接表采用一个结点里存放一个表元素的方式，是最常见的方式，也是一种极端的安排。实际上，完全可以在一个链接结点里存储多个表元素，形成连续存储和链接存储的一种混合形式。从这个角度看，顺序表也是链接表的一种特殊形式，这种链接表只有一个结点，所有表元素都保存在这个结点里。回忆一下图 3.6 的分离式实现，情况可以看得很清楚。

顺序表的另一个优点是访问的局部性。表元素顺序映射到内存中连续的单元里，下一个元素的实际存储位置与当前元素很近。由于目前各种新型计算机体系结构的特点，顺序访问内存中相近位置的效率比较高，而真正的随机内存访问效率较低。对于链接表，逻辑上的下一个结点可能被安排在内存中的任何位置，逻辑上的顺序访问实际上可能是对计算机内存中许多不同地方的随机访问，效率比较低。

另一方面，顺序表需要较大的连续空间，这一情况有时也会带来问题。如果需要建立保存很多元素的巨大的表，采用顺序表可能给 Python 解释器的存储管理带来麻烦。链接表的结点是小块存储，无论表有多么大，存储管理问题都容易处理。

如前所述，顺序结构和链接结构是程序里构造复杂数据结构的两种基本技术，顺序表和链接表是这两种技术的典型代表。对这两种表的操作技术也是最典型的复杂数据结构操作技术，以及最重要的设计和编程技术。通过对这两种表结构的操作练习，能够大大提升初学者的编程水平。本章学习的一个重点是链接表操作，这是高级编程技术的一个难点，也是（几乎）一切高级编程技术的基础。

此外，本章内容还展示了一些重要的面向对象编程技术，讨论了如何通过继承已有的类构造新的具有类似功能（或结构）的类，以满足实际需要。

练习

一般练习

1. 复习下面概念：线性表（表），基本元素集合，元素集合和序列，下标，空表，（表的）长度，顺序关系（线性关系），首元素，尾元素，前驱和后继，数据抽象的实现者和使用者，顺序表（连续表）和链接表（链表），顺序表的元素布局，索引和索引结构，容量，（元素）遍历，查找（检索），定位，加入和删除元素，尾端加入（插入）和删除，保序插入和删除，表的一体式实现和分离式实现，动态顺序表，元素存储区的增长策略（线性增长，加倍增长），元素反转和排序，链接结构，单链表（单向链接表），链接，表头变量（表头指针），空链接，链表处理的扫描模式，汇集对象，尾结点引用，循环单链表，双向链接表（双链表），循环双链表，链表反转，链表排序，Josephus 问题，随机存取，顺序存取，访问的局部性，类定义的内在一致性。
2. 下面哪些事物的相关信息适合用线性表存储和管理，为什么？
 - a) 在银行排队等候服务的顾客；
 - b) 书架上的一排书籍；
 - c) 计算机桌面上的各种图标及其相关信息；
 - d) 计算机的文件和目录（文件夹）系统；
 - e) 个人的电话簿；

- f) 工厂流水线上的一系列工位；
 - g) 个人银行账户中的多笔定期存款；
 - h) 一辆汽车的所有部件和零件。
3. 假设需要频繁地在线性表的一端插入 / 删除元素。如果用顺序表实现，应该用哪一端作为操作端？如果用链接表实现呢？为什么？
 4. 假设线性表的一个应用场景是基于位置访问表中元素和在表的最后插入 / 删除元素，采用哪种结构实现线性表最为合理，为什么？
 5. 在某种链接表使用场景中，最常用操作是在表首元素之前插入或删除元素，以及在表尾元素之后插入元素。此时采用哪种实现技术最为合适？为什么？
 6. 请列举出顺序表的主要缺点，如果改用链接表能否避免这些缺点？请交换两者角色并重新考虑类似的问题。
 7. 请仔细总结顺序表和链接表的特点，并设法提出一些操作场景，在其中采用一种结构比较合适而采用另一种则非常不合适。
 8. 设法总结出一些设计和选择的原则，说明在什么情况下应该优先使用顺序表，在什么情况下应该优先使用链接表。
 9. 请考虑两个排序序列（例如元素按“ $<$ ”关系从小到大排序）的合并操作，称为归并，并设计一种算法实现这种序列的归并。分析你设计的算法，如果其复杂性不是 $O(\max(m, n))$ ，请修改算法使之达到 $O(\max(m, n))$ （其中 m 和 n 是两个序列的长度）。
 10. 请从操作实现的方便性和效率的角度比较带尾结点指针的单链表和循环单链表，以及它们相对于简单单链表的优点和缺点等，并总结在什么情况下应该使用这两种结构，或应优先使用其中某一种。
 11. 请全面比较循环单链表和双链表的各方面特点。

编程练习

1. 检查本章开始定义的线性表抽象数据类型和 3.3 节定义的链表类 `LList`，给 `LList` 加入在抽象数据类型中有定义，但 `LList` 类没定义的所有操作。
2. 请为 `LList` 类增加定位（给定顺序位置的）插入和删除操作。
3. 给 `LList` 增加一个元素计数值域 `num`，并修改类中操作，维护这个计数值。另外定义一个求表中元素个数的 `len` 函数。Python 的内置标准函数 `len` 可以自动调用用户定义类里的相关函数 `__len__`，也可以用它作为方法名。请比较这种实现和原来没有元素计数值域的实现，说明两者各自的优缺点。
4. 请基于元素相等操作 “`==`” 定义一个单链表的相等比较函数。另请基于字典序的概念，为链接表定义大于、小于、大于等于和小于等于判断。
5. 请为链接表定义一个方法，它基于一个顺序表参数构造一个链接表；另请定义一个函数，它从一个链接表构造出一个顺序表。
6. 请为单链表类增加一个反向遍历方法 `rev_visit(self, op)`，它能按从后向前的顺序把操作 `op` 逐个作用于表中元素。你定义的方法在整个遍历中访问结点的总次数与表长度 n 是什么关系？如果不是线性关系，请设法修改实现，使之达到线性关系。这里要求遍历方法的空间代价是 $O(1)$ 。（提示：你可以考虑为了遍历而修改表的结构，只要能在遍历的最后将表结构复原。）

7. 请为单链表类定义下面几个元素删除方法，并保持其他元素的相对顺序：
- del_minimal() 删除当时链表中的最小元素；
 - del_if(pred) 删除当前链表里所有满足谓词函数 pred 的元素；
 - del_duplicate() 删除表中所有重复出现的元素。也就是说，表中任何元素的第一次出现保留不动，后续与之相等的元素都删除。
- 要求这些操作的复杂度均为 $O(n)$ ，其中 n 为表长。
8. 请为单链表类定义一个变动方法 interleaving(self, another)，它把另一个单链表 another 的元素交错地加入本单链表。也就是说，结果单链表中的元素是其原有元素与单链表 another 中元素的一一交错的序列。如果某个表更长，其剩余元素应位于修改后的单链表的最后。
9. 考虑实现单链表插入排序的另一个想法：插入排序也就是把要排序的元素一个个按序插入到一个元素已经排好序的链表里，从空链表开始。请根据这个想法实现另一个完成单链表排序的插入排序函数。
10. 定义一个单链表剖分函数 partition(lst, pred)，其参数为单链表 lst 和谓词函数 pred，函数 partition 返回一对单链表（一个序对），其中第一个单链表包含着原链表 lst 里所有值满足 pred 的结点，第二个链表里是所有其他结点。注意，两个表里的结点还应保持原表里结点的相对顺序。也就是说，如果在某结果表里结点 a 的后继结点是 b，在原表 lst 里 a 一定位于 b 之前。
11. 扩充本章给出的循环单链表类 CLLIST，实现 LList1 中有定义的所有方法。
12. 请为循环单链表类扩充一个方法 interleaving(self, another)，要求见上面针对简单单链表的有关习题。
13. 请为循环单链表类定义前面各习题中针对简单单链表类提出的方法。
14. 请基于 Python 的 list 实现一个元素排序的顺序表类，其中元素按“ $<$ ”关系从小到大排序存放。考虑需要定义的方法并给出定义，包括一个方法 merge(self, another)，其参数 another 是另一个排序顺序表。该方法将 another 的元素加入本顺序表，并保证结果表中的数据仍是正确排序的。
15. 请从简单单链表类派生一个排序单链表类，表中元素按“ $<$ ”关系从小到大排序存放。首先考虑需要覆盖的方法并给出定义。为该类增加方法 merge(self, another)，参数 another 也是排序单链表。该方法将链表 another 的元素加入本链表，并保证结果链表中的数据仍是正确排序的。
16. 请为双链表类定义 reverse 方法和 sort 方法，要求通过搬移结点中数据的方式实现这两个操作。
17. 请为双链表类定义 reverse1 方法和 sort1 方法，要求操作中不移动结点中的数据，只修改结点之间的链接。
18. 实现双链表排序的一种可能做法是直接利用单链表的排序函数，只将结点按 next 方向正确排序链接，最后重新建立 prev 链接关系。请基于这个想法为双链表类实现一个排序方法，其中直接调用单链表类的插入排序方法。
19. 请实现一个循环双链表类。
20. 请考虑一种在一个结点里存储 16 个元素的单向链接表，定义一个类实现这种链表，为这个类定义各种重要的线性表操作。请从各方面比较这种实现与每个结点存储一个元素的简单

实现。

21. 利用(顺序或链接)表和第2章的人事记录类, 实现一个简单的学校人事管理系统。首先分析问题, 描述一个人事管理ADT, 而后实现这个系统。由于不需生成多个实例, 可以用类的数据属性保存人事信息(的表), 用一组类方法实现必要操作。这是一种在Python语言中建立单例(singleton)数据抽象的技术。
22. 利用链接表实现一种大整数类BigInt。用一个链表表示一个大整数, 表中每个结点保存一位十进制整数, 这样, 任意长的链表就可以保存任意长的整数了。请实现这种大整数的各种重要运算。
23. 链接表里的结点都是独立存在的对象, 有可能脱离原来所在的表, 或者从一个表转移到另一个表。从表对象出发通过遍历可以访问表中的每个结点(及其数据), 而从一个表结点出发则无法确定它属于哪个表(或者不属于任何一个表)。请分析这个问题, 考虑在什么场景下确定结点的归属问题有重要意义。考虑下面的技术: 为每个结点增加一个“表指针”指向其所属的表。定义一个类实现这种表。

第4章 字符串

文字处理是最重要的一类计算机应用。另一方面，各种计算机应用系统都需要与人交互，因此或多或少都需要处理文字信息。最基本的文本处理是文本处理，处理对象是结构简单的语言文本，即是某种（或某些）自然语言中的基本文字符号构成的序列。在计算机领域中，这样的基本文字符号称为字符，符号的序列称为字符串。

4.1 字符集、字符串和字符串操作

讨论字符串及其数据结构实现，以及字符串处理，首先需要有一个确定的字符集。这里把字符作为一个抽象概念，字符集就是有穷的一组字符构成的集合。在实际工作中，人们经常考虑的是计算机领域广泛使用的某种标准字符集，如 ASCII 字符集或者 Unicode 字符集。实际上，完全可以用任意一个数据元素集合作为字符集。

基于字符串处理的需要，要求字符集上有一种确定的序关系，称为字符序，也就是说字符串里的字符上定义的一种顺序。因此，对这里的任意两个字符，它们或者相等（是同一字符），或者某个字符排在另一字符之前。下面用数学的小于符号表示排在前面，这样，对字符集里任意两个字符， $<$ 、 $=$ 、 $>$ 三种关系之一成立。

字符串（简称为串）可以看作一类特殊的线性表，表中元素取自选定的字符集。从这个角度看，字符串数据结构似乎就是一类特殊的线性表，而线性表的相关问题都已经在第3章研究过了，没有重复的必要。但实际上不尽然，因为字符串操作有其特点，而且其中许多操作并不是常见的线性表操作。对于线性表，人们经常考虑的是元素与表的关系、元素的插入和删除。而在考虑字符串时，人们关注的操作不同于一般线性表，经常需要把字符串作为一个整体使用和处理，考虑许多以整个串为对象的操作。

4.1.1 字符串的相关概念

现在首先介绍与字符串有关的一些重要概念：

- **字符串的长度：**一个字符串中字符的个数称为该串的长度，长度为 0 的串称为空（字符）串。显然，在任意字符集里只有唯一的一个空串。下面用 $|s|$ 表示字符串 s 的长度。举例说， $abcd$ 的长度为 4， $aaabcd$ 的长度为 6。
- **字符在字符串里的位置：**与一般线性表类似，字符串里的字符顺序排列，一个串里的每个字符有其确定的位置（下标）。本书将始终用 0 开始的自然数表示字符在字符串里的位置，字符串里首字符的下标是 0。举例说，在字符串 $abbreviation$ 里，第一个 a 的下标为 0，第二个 a 的下标为 7，字母 v 的下标是 5， n 的下标为 11。
- **字符串相等：**字符串的相等基于字符集里的字符定义。说两个字符串 s_1 和 s_2 相等，如果它们的长度相等，而且位于两个串中对应位置的各对字符两两相同。两个字符相同即为同一字符，这一问题显然可以判定。下面用“ $=$ ”表示字符串相等。
- **字典序：**字典序是字符串上的一种序关系，基于字符序定义。对于字符串

$$s_1 = a_0 \ a_1 \cdots a_{n-1} \quad s_2 = b_0 \ b_1 \cdots b_{m-1}$$

如果存在一个 $k \geq 0$ 使得对任何 $i < k$ 都有 $a_i = b_i$, 但 $a_k < b_k$; 或者 $n < m$ 且对所有的 $0 \leq i < n$ 都有 $a_i = b_i$, 则 $s_1 < s_2$ 。

也就是说, 从左向右查看两个串中下标相同的各对字符, 遇到的第一对不同字符的字符序决定了这两个字符串的顺序; 另外, 如果两个串中相同下标的各对字符都相同, 但其中一个串较短, 那么就认为它较小, 排在前面。举例说, 考虑英文字母的集合, 字符序采用字母表中的顺序 $abcd\cdots$ 。这样就有字符串 abc 小于 add , 也小于 $abcd$, 因为虽然 abc 和 $abcd$ 的前 3 个字符相同, 但后者较长。

- **字符串拼接:** 拼接是最重要的一种字符串操作。直观说, 两个串的拼接将得到另一个串, 其中首先顺序出现第一个串里的所有字符, 之后是第二个串里的所有字符。在不同书籍里, 字符串拼接的记法常常不同, Python 语言里用加号 (+) 表示字符串拼接, 本书也将采用这种记法。对上面的两个字符串 s_1 和 s_2 ,

$$s_1 + s_2 = a_0 a_1 \cdots a_{n-1} b_0 b_1 b_2 \cdots b_{m-1}$$

- **子串关系:** 称字符串 s_1 是另一字符串 s_2 的子串, 如果存在串 s 和 s' 使得

$$s_2 = s + s_1 + s'$$

直观说, 如果串 s_1 与串 s_2 中的一个连续片段相同, 就说 s_1 是 s_2 的子串。显然, 一个串可以是或者不是另一个串的子串。如果 s_1 是 s_2 的子串, 也说 s_1 在 s_2 中出现, 并称 s_2 中与 s_1 相同的那个字符段的第一个字符的位置为 s_1 在 s_2 中出现的位置。注意, 在 s_2 中完全可能存在多个与 s_1 相同的段, 这时就说 s_1 在 s_2 中多次出现。进而, 如果 s_1 在 s_2 中多次出现, 不同的出现也可能不是独立存在的, 可能相互重叠。例如, $babb$ 在 $babbabbbbabb$ 里有三次出现, 前两个出现重叠, 第一个出现在位置 0, 第二个在位置 3, 第三个在位置 8。根据子串的定义, 很显然, 空串是任何一个字符串的子串; 另一方面, 任何字符串也是该串自身的子串。

- **前缀和后缀** 是两种特殊子串: 如果存在 s' 使 $s_2 = s_1 + s'$, 则称 s_1 为 s_2 的一个前缀; 如果存在 s 使 $s_2 = s + s_1$, 则称 s_1 为 s_2 的一个后缀。直观地说, 一个串的前缀就是该串开头的任意一段字符构成的子串, 后缀就是该串最后的任意一段字符构成的子串。显然, 空串和 s 既是 s 的前缀, 也是 s 的后缀。
- 其他有用的串运算如: 串 s 的 n 次幂是连续的 n 个 s 拼接而成的串 (在一些书籍里用 s^n 表示, 在 Python 语言里用 $s*n$ 表示)。串替换指将一个串里的一些 (互不重叠的) 子串代换为另一些串得到的结果。由于可能重叠, 必须规定代换的顺序, 如从左到右。

还有许多有用的串运算, 有关情况可以参考 Python 的 str 类型, 或其他语言的字符串类型。字符串处理的经典语言是 SNOBOL。目前流行的各种脚本语言, 包括 Python、Perl 等, 都提供了非常丰富的字符串功能。

与字符串有关的一些研究

字符串集合和拼接操作构成了一种代数结构, 空串是拼接操作的“单位元”(幺元)。请注意, 拼接操作有结合律但是没有交换律, 对任意 s_1 、 s_2 、 s_3 ,

$$(s_1 + s_2) + s_3 = s_1 + (s_2 + s_3)$$

但一般而言 $s_1 + s_2 \neq s_2 + s_1$ 。

从数学上看, 字符串集合加上拼接操作, 构成一个半群。这是一种典型的非交换半群。由于存在单位元, 因此是一种幺半群。

关于串的理论有许多研究工作。20世纪40年代研究者基于串和串替换操作, 提出了一种

称为 post 系统的计算模型。这是一种与图灵机等价的计算模型。

另外，(串)重写系统 (rewriting system) 也是计算机理论的一个研究领域，研究基于字符串替换的计算，一直非常活跃，有许多重要结果和应用成果。

4.1.2 字符串抽象数据类型

现在考虑字符串抽象数据类型的定义。在这里首先有一种选择：是将字符串定义为一种不变数据类型呢，还是定义为一种可变数据类型？一些语言里定义的是不变字符串类型，例如 Python，也有些语言和程序库提供了可变的字符串类型。

下面是一个简单的字符串 ADT，其中定义了一些字符串操作：

```
ADT String:  
    String(self,sseq)      #基于字符序列sseq建立一个字符串  
    is_empty(self)         #判断本字符串是否空串  
    len(self)              #取得字符串的长度  
    char(self,index)       #取得字符串中位置index的字符  
    substr(self,a,b)       #取得字符串中[a:b]的子串，左闭右开区间  
    match(self,string)     #查找串string在本字符串中第一个出现的位置  
    concat(self,string)    #做出本字符串与另一字符串string的拼接串  
    subst(self,str1,str2)  #做出将本字符串里的子串str1都替换为str2的结果串
```

最后两个操作可以实现为变动操作，实际修改本字符串；也可以实现为非变动操作，操作中生成满足要求的另一个字符串。

这个抽象数据类型里的大部分操作都很简单，只有 match 和 subst 操作比较复杂。不难看出，subst 操作的基础也是 match，因为需要找到 str1 在串里一个个出现的位置。子串检索操作（也称为子串匹配）是字符串的核心操作，本章后面部分将详细研究这个问题。

4.2 字符串的实现

4.2.1 基本实现问题和技术

字符串是字符的线性序列，可以采用线性表的各种实现技术实现，用顺序表或链接表的形式表示。例如，采用一体式顺序表形式，可用于表示创建时确定大小的字符串；采用分离式顺序表形式，适合用于表示创建后需要动态变化大小的字符串（参看图 3.6）。显然，如果需要实现可变字符串对象，那么就必须采用后一技术。

另外，还可以根据串本身的特点和串操作的特点考虑其他表示方式。当然，无论如何表示，实现的基础只能是基于顺序存储或 / 和链接结构。这里的关键问题是，所用表示方式应能较好地支持字符串的管理和相关操作的实现。

考虑字符串的表示时，有两个重要的方面必须确定，它们是：

- 字符串内容的存储。这里有两个极端：①把一个字符串的全部内容存储在一块连续存储区里；②把串中每个字符单独存入一个独立存储块，并将这些块链接起来。连续存储的主要问题是需要大块存储，极长的字符串可能带来问题；而一个字符一块存储，需要附加一个链接域，额外存储开销比较大。实际中完全可以采用某种中间方式，把一个串的字符序列分段保存在一组存储块里，并链接起这些存储块。

- 串结束的表示。不同字符串的长度可能不同，如果采用连续表示方式，由于存储在计算机里的都是二进制编码，从存储内容无法判断哪里是串的结束。所以，必须采用某种方式表示字符串的结束。这里也有两种基本方式：①用一个专门的数据域记录字符串长度，就像前面连续表中的 num 域；②用一个特殊编码表示串结束，为此需要保证该编码不代表任何字符。C 语言的字符串采用了第 2 种方式。

由于字符串是字符的线性表，基本实现问题在前面都已讨论过，不再赘述。

现在考虑字符串的操作。不难看到，许多串操作可以看作线性表常规操作的具体实例，包括判空串、求串长度、字符检索、字符插入和删除等。串拼接也就是基于已有的表建立新表，或者直接修改已有的表（作为变动操作的拼接）。下面考虑一个复杂一点的操作——串替换，通过它的实现进一步了解串操作中可能遇到的一些情况。

如前所述，串替换操作要求把一个串（可称为主串） s 中某个子串 t 的所有出现都替换为另一个串 t' 。这个操作牵涉到三个串，即被处理的主串 s ，作为被替换对象需要从 s 中替换掉的子串 t ，以及用于代换 t 的 t' 。不难看到替换中的一些问题：

- 被替换串 t 可能在 s 中出现多次，需要通过一系列具体的子串代换完成整个替换。
- t 在 s 里的多次出现有可能重叠（回忆前面的例子），只能规定一种代换顺序（例如从左到右），被一个具体代换破坏的子串不应再代入新串。
- 在完成了一次子串代换后，应该从代入的新串之后继续工作。即使代入新串后形成的部分中存在与 t 匹配的片段，也不应在本次串替换中考虑。
- 无法预知 s 里有 t 的几个独立出现，因此，即使事先知道 s 、 t 、 t' 的长度，也无法根据它们算出替换后结果串的长度。这件事给串替换的实现带来一些困难。

实际上，做这种子串代换只能逐步完成，由于事先无法确定结果串的长度，因此构造结果串的过程中可能需要扩充已有的存储。这方面的问题前面已经讨论过。另一方面，易见串替换的关键是找到子串匹配，这是后面讨论的一个重要问题。

4.2.2 实际语言里的字符串

由于字符串处理的重要性，许多编程语言提供了标准的字符串功能，例如：

- C 语言的标准库有一组字符串函数（其中功能由头文件 string.h 描述），一些 C 语言系统提供扩展的字符串库。
- C++ 语言标准库里提供了另一个字符串库〈string〉。
- Java 语言的标准库也包含一个字符串库。
- 许多脚本语言（包括 Python）都提供了功能丰富的字符串库。

许多实际的字符串库采用动态顺序表结构作为字符串的表示方式，这样既能支持任意长的字符串，又能比较有效地实现各种重要的字符串操作。不同语言里的字符串功能采用了不同的设计，最重要的差异包括以下几个方面：

- 字符串是实现为一种不变数据类型，还是实现为可变数据类型。Python 的字符串是不变数据类型，也有一些语言的字符串是可变类型。
- 是否另有一个独立的字符类型。Python 没有字符类型，这里的单个字符就是长度为 1 的字符串。这种看法的优点是统一性，减少概念，但也有缺点。有些语言有独立的字符类型，例如 C 语言和由 C 发展出的 C++、Java 等。
- 基本字符集的选择。早期语言都以简单的字符编码集合作为字符集，其中只有 128 或

者 256 个字符，例如 ASCII 字符集或扩充的 ASCII 字符串。新近的发展主要是由于国际化的潮流，计算机系统需要处理更多的自然语言信息，包括汉语等大型字符集。Python 和另一些新语言（如 Java）都明确选择 Unicode 作为其编码字符集，就是为了跟上时代的潮流，满足计算机软件国际化的需要。

实际上，要支持不同的字符串操作和不同的应用环境，也可能需要不同的实现方式。例如，有些应用系统需要记录和处理极长的字符串，如支持操作 MB（大约为 10^6 ）或更长的大字符串，对于这种需要，采用连续存储就可能带来较难处理的管理问题。

再看一个具体例子。最常见的一类应用程序是文本编辑器（包括程序编辑器），其中被编辑文本也是字符串。编辑器需要支持一大批操作，它们来自实际编辑工作的各种具体需要。例如，在一个位置反复插入字符或加入字符序列，标记 / 复制 / 粘贴操作，查找和反复替换，各种格式化（对于文本编辑器就是确定换行位置、适当加入空格等），等等。为了有效实现这些操作，需要仔细设计字符串的表示方式。

4.2.3 Python 的字符串

本节从数据结构和算法的角度，简单综述 Python 语言的字符串。

Python 标准类型 str 可以看作抽象的字符串概念的一个实现。str 是一个不变类型，其对象创建后内容和长度都不变化，但不同 str 对象的长度可能不同，因此需要在对象里记录字符串长度。在 Python 官方实现里，str 对象采用了第 3 章讨论提到的一体式顺序表形式，如图 4.1 所示。实际上，在一个 str 对象的头部，除了记录字符串长度外，还记录了一些解释器用于管理对象的信息，它们是为系统内部操作服务的。



图 4.1 Python 字符串对象的表示

str 的操作

str 对象的操作分为两类：

- 获取 str 对象的信息，如得到串长，检查串的内容是否全为数字等。
- 基于已有 str 对象构造新的 str 对象，包括切片、构造小写 / 大写复制、各种格式化等。切分操作 split 等是构造包含多个字符串的表。更复杂的操作如 replace，它就是前面字符串 ADT 里的 subst。

有一些操作属于子串匹配，如 count 检查子串出现次数，endwith 检查后缀，find/index 找子串的位置等。这类操作非常重要。

str 构造操作的实现

检查字符串内容的操作可以分为两类：

- $O(1)$ 时间的简单操作，包括求串长度的 len 和定位访问字符（注意，由于 Python 没有字符类型，定位访问是构造只包含一个字符的字符串）等。
- 其他操作都需要扫描整个串的内容，包括 Python 不变序列的一些共有操作（包括 in、not in、min/max），各种字符串类型判断（如是否全为数字等）。这些操作都需通过一个循环逐个检查串中字符才能完成工作，因此都是 $O(n)$ 时间操作。

有些操作需要构造新字符串，情况比较复杂。这类操作实现的基本模式包括两部分工作：一是需要为欲构造的新字符串安排存储，再就是根据被操作串（和可能的参数串）以及操作确定的特定方式，在新存储块里做出所需的新串。

以切片操作 `s[a, b, k]` 为例，其算法应该是：

- 1) 根据 a 、 b 、 k 的值和 s 的长度计算出新字符串的长度，分配存储。
- 2) 用一个形式为 “`for i in range(a, b, k): ...`” 的语句逐个把 $s[i]$ 复制到新串里顺序的各个位置。
- 3) 返回这个新建的字符串。

更复杂的操作如 `replace`，如果不弄清原串里被替换字符串的出现次数，就不能事先分配好合适的存储。这一操作有两种可行实现方法：一种是首先对整个串做一次匹配，确定该代换串出现的次数（和位置），基于这一数据可以算出代换结果串的大小，下面的事情就简单了。另一种方法是在构造新串的过程中动态调整大小，采用上一章讨论的动态顺序表技术。这两种操作都作为课后练习。

许多 Python 操作的基础是子串查找和匹配，这是下一节的主题。

4.3 字符串匹配（子串查找）

从前面讨论中可以看到，对于字符串对象，最重要的操作之一是子串匹配。这个操作不仅本身很重要，还是许多其他字符串操作的基础。由于其重要性，互联网上有专门的 Wiki 网页[⊖]讨论相关的问题。子串匹配问题也被称为字符串匹配（string matching）或者字符串查找（string searching）。有些教科书里称其为模式匹配（pattern matching），但实际上，模式匹配是一个内涵更广的概念，详见 4.4 节的讨论。

本节将集中关注字符串匹配问题，还要介绍两个典型的匹配算法。

4.3.1 字符串匹配

假设有两个串（其中的 t_i 、 p_j 是字符）：

$$t = t_0 \ t_1 \ t_2 \cdots t_{n-1}$$

$$p = p_0 \ p_1 \ p_2 \cdots p_{m-1}$$

字符串匹配就是在 t 中查找与 p 相同的子串的操作（或过程）。下面将称 t 为目标串，称 p 为模式串。通常有 $m \ll n$ ，也就是说，模式串的长度远小于目标串的长度。

实际上，这一定义还可以推广，后面有进一步讨论。

在实际中 n 可能非常大， m 也可能有一定的规模，程序里可能需要做许多模式串和 / 或许多目标串之间的匹配，所以匹配算法的效率非常重要。

应用

许多计算机应用中最基本的操作就是字符串匹配。例如：

- 在使用编辑器或字处理系统工作时，人们经常需要在文本中查找单词或句子（或者中文字符及其词语），在程序里找拼写错误的标识符等。
- 对于 Email 服务器和客户端程序的垃圾邮件过滤器，通过检查邮件标题、发件人或内容中是否包含特定字符序列，评价其是否属于垃圾邮件。
- Google 等网络搜索系统的技术基础就是在互联网的网页中查找与各种检索需求（通常都是比较短的字符串）匹配的网页。
- 各种防病毒软件主要就是在各种文件里检索表征病毒的片段，也是串匹配。

近年分子生物学领域的发展更是把串匹配技术的应用推向高潮。据说今天全世界计算能

[⊖] 见 [Wiki:`http://en.wikipedia.org/wiki/String_searching_algorithm`](http://en.wikipedia.org/wiki/String_searching_algorithm)。

力中相当大的一部分（有说超过一半）是在做 DNA（脱氧核糖核酸）匹配，即是一种串匹配。DNA 是细胞核里的一类长分子，在遗传中起着核心作用。DNA 由四种碱基构成，分别是腺嘌呤（adenine）、胞嘧啶（cytosine）、鸟嘌呤（guanine）、胸腺嘧啶（thymine）。它们的不同组合构成了各种氨基酸、蛋白质和其他更高级的生命结构。

抽象地，DNA 片段可以看作 a、c、g、t 四个符号（分别表示上述四个碱基）构成的符号串，如 acgatactagacagt。考察在某种蛋白质分子中是否出现了某 DNA 片段，可以看成是拿这个 DNA 片段在蛋白质分子中做匹配。另外，DNA 分子可以被切断和拼接，这些动作由各种特殊的酶完成，而酶也是采用特定模式来确定剪切或拼接的位置，就像是在做串匹配。另外，所谓的 DNA 计算就是利用酶完成的。

实际的串匹配问题

实际应用中模式匹配的规模（ n 和 m ）可能非常大，而且有严苛的时间要求。具体应用的场景也有许多变化。例如：

- 需要检索的文本可能很大，经常需要用一个模式串在其中反复检索。
- 网络搜索需要处理数以亿万计的网页，对付来源于世界各地的发生频率极高的千奇百怪的检索需求。
- 防病毒软件要在合理时间内检查数以十万计的文件（在目前的普通微机上，文件的数据量也以 GB 计），而且需要同时处理一大批病毒特征串。
- 运行在服务器上的邮件过滤程序，需要在很短的时间内扫描数以万计的邮件和附件，用已知的或特定的一组模式串在其中匹配，模式串的集合还经常变化。
- 为了生物 / 疾病 / 药物的研究和新作物 / 生品种培养等生物学工程应用，需要用大量 DNA 模式与大量 DNA 样本（都是 DNA 序列）匹配。

由于在计算机科学、生物信息学等许多领域的重要应用，串模式匹配问题已成为一个极其重要的计算问题。高效的串匹配算法也变得越来越重要。目前国际上有几个集中关注字符串匹配问题的国际学术会议，也曾经有过相关专门的国际竞赛。总而言之，字符串匹配是在理论和实际中都非常重要的计算问题。

4.3.2 串匹配和朴素匹配算法

由于字符串匹配的重要性，人们对这个问题做了许多研究，开发出一批很有意义（也非常有趣）的算法。在各种讨论算法的教科书和专著中，都包括对串匹配算法的专门讨论，在网络上也有许多材料，如网络百科全书 Wiki 中的专门介绍。

粗看起来，字符串匹配是一个非常简单的问题。因为字符串是最简单数据（字符）的简单线性序列，其结构也最简单（简单的顺序结构）。因此人们很容易想到最简单而且直接的算法。但实际情况是，这种直接而简单的算法多半不是高效算法，因为它们并没有很好地利用问题的内在性质。通过研究，对这个貌似简单的字符串匹配问题，人们已开发出许多想法和做法上“大相径庭”的算法。下面将介绍两个算法。

还有一个情况也值得注意：前面提到（实例中也说明了）实际应用中的不同需要。例如，是用一个模式在很长的目标串里反复匹配（确定出现位置），还是用一组（可能更多）模式在一个或一组目标串里确定是否有匹配，等等。不同算法在处理不同的实际情况时，也可能有不同的表现。一些算法更适应某些特定的使用情况。这方面的细节和分析超出了本课程的范围，不再仔细讨论，但这个问题请读者注意。

串匹配算法

做字符串匹配的基础是逐个比较字符。从串匹配的角度看，两个字符的比较只需要得到相同或者不同的结论，是一个逻辑判断。

如果从目标串的某个位置 i 开始，模式串里的每个字符都与目标串里的对应字符相同，就是找到了一个匹配。如果在比较中遇到了一对不同的字符，那就是不匹配，说明模式串不能与目标串中从位置 i 开始的子串匹配。串匹配算法设计的关键有两点：①怎样选择开始比较的字符对；②发现了不匹配后，下一步怎么做。对这两点的不同处理策略，就形成了不同的串匹配算法。从朴素匹配算法和 KMP 匹配算法这两个匹配算法中可以看到一些情况，更多实例可以参考其他专业书籍或网络上的材料。

朴素的串匹配算法

最简单的朴素匹配算法采用最直观可行的策略：

- ①从左到右逐个字符匹配；②发现不匹配时，转去考虑目标串里的下一个位置是否与模式串匹配。

图 4.2 给出了一个匹配示例。上面的长串是目标串，下面是模式串。在初始状态 (0) 两个串的起始字符对齐。顺序比较，立即发现第一对字符不同。将模式串右移一位得到状态 (1)。顺序比较第一对字符相同，但第二对字符不同。将模式串再右移一位。这样继续到状态 (3)，模式串的 5 个字符都与目标串对应字符相同，找到了一个匹配。继续做下去还可能找到更多匹配。

下面是朴素串匹配算法的一个实现：

```

def naive_matching(t, p):
    m, n = len(p), len(t)
    i, j = 0, 0
    while i < m and j < n:
        if p[i] == t[j]:
            i, j = i + 1, j + 1
        else:
            i, j = 0, j - i + 1
    if i == m:
        return j - i
    return -1

```

i==m说明找到匹配
字符相同！考虑下一对字符
字符不同！考虑t中下一位置
找到匹配，返回其开始下标
无匹配，返回特殊值

算法的实现很容易理解，不需要更多解释。

上述串匹配算法非常简单，容易理解，但其效率很低。造成其低效的主要因素是执行中可能出现回溯：匹配中遇一对字符不同时，模式串 p 将右移一个字符位置，随后的匹配回到模式串的开始（重置 $j = 0$ ），也回到目标串中前面的下一个位置，从那里再次由 p_0 开始比较字符。

由于这种操作策略，算法的效率很低。最坏情况是每一趟比较都在模式串的最后遇到了字符不匹配的情况，在这种匹配中总共需要做 $n-m+1$ 趟比较，总的比较次数为 $m \times (n-m+1)$ ，所以，这个算法时间复杂性为 $O(m \times n)$ 。

下面是出现最坏情况的一个实例：

模式串：00000001

朴素匹配算法的效率低，根源在于把每次字符比较看作完全独立的操作，完全没有利用字典序。

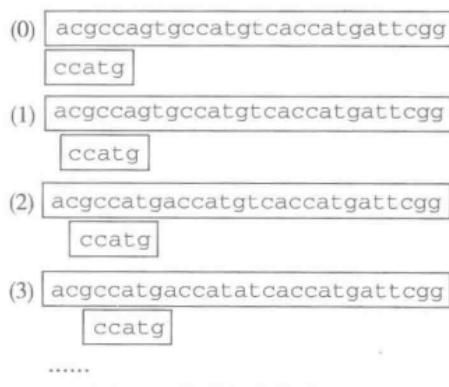


图 4.2 朴素的字符串匹配

字符串本身的特点（每个字符串都是特殊的，只有有穷多个不同字符，等等），也没有尽可能地利用前面已经做过字符比较中得到的信息。

从数学上看，这样做相当于认为目标串和模式串里的字符都是完全随机的量，而且有无穷多种可能取值，因此任意两次字符比较相互无关也不可借鉴。实际情况并不是这样，字符串中的字符取值来自一个有穷集合，而且每个串都具有确定的有穷长度。特别是模式串，通常不太长，而且在匹配中被反复使用。

各种改进的算法都以这样或那样的方式利用了字符串的这些特点。下面介绍一个高效的字符串匹配算法，其基础也在于这些方面。

4.3.3 无回溯串匹配算法（KMP 算法）

KMP 算法是一个高效的串匹配算法。由 D. E. Knuth 和 V. R. Pratt 提出，J. H. Morris 也几乎同时独立发现了这个算法，因此它被称为 KMP 算法。这是本书中的第一个非平凡算法，基于对问题的深入分析和理解。这个算法比较复杂，不太容易理解。但与朴素的串匹配算法相比，KMP 算法的效率有本质的提高。

基本考虑

为了理解 KMP 算法的想法，首先需要了解朴素匹配算法的缺陷。现在仔细考察一下朴素匹配算法的执行过程及其问题。

设目标串是 ababcabcacbab，模式串是 abcac。图 4.3 左边一列给出了朴素匹配算法执行中的一系列情况。状态 (0) 的匹配进行到模式串中字符 c 时失败，此前有两次成功匹配，从中可知目标串前两个字符与模式串前两个字符相同。由于模式串的前两个字符不同，与 b 匹配的目标串字符不可能与 a 匹配，所以状态 (1) 的匹配一定失败。朴素匹配算法未利用这种信息，做了无用功。再看状态 (2)，这里前 4 个字符都匹配，最后匹配 c 时失败。由于模式串中第一个 a 与其后的两个字符 (bc) 不同，用 a 去匹配目标串里的 b、c 也一定失败。跳过这两个位置不会丢掉匹配点。另一方面，模式串中下标为 3 的字符也是 a，它在状态 (2) 匹配成功，首字符 a 不必重做这一匹配。朴素的匹配算法中没有考虑这些问题，总是一步步移位并从头比较。

从实例的分析可知，由于模式串在匹配之前已知，而且通常在匹配中反复使用，如果先对模式串做一些分析，记录得到的有用信息（如其中哪些位置的字符相同或不同），就有可能避免一些不必要的匹配，提高匹配效率。这种做法是实际匹配前的静态预处理，只需要做一次。记录下来的信息可以在匹配中反复使用。

KMP 算法的精髓就是开发了一套分析和记录模式串信息的机制（和算法），而后借助得到的信息加速匹配。对上面实例，用 KMP 算法的匹配过程如图 4.3 右列所示。

在状态 (0) 匹配到第一个 c 失败时，由于已知前两个字符不同，KMP 算法直接把模式串移两个位置，模式串开头的 a 移到 c 匹配失败的位置，达到状态 (1)。这次匹配直到模式串最后的 c 处失败，由于已知模式串 c 之前是 a，首字符也是 a，而且两个字符之间的字符与它们

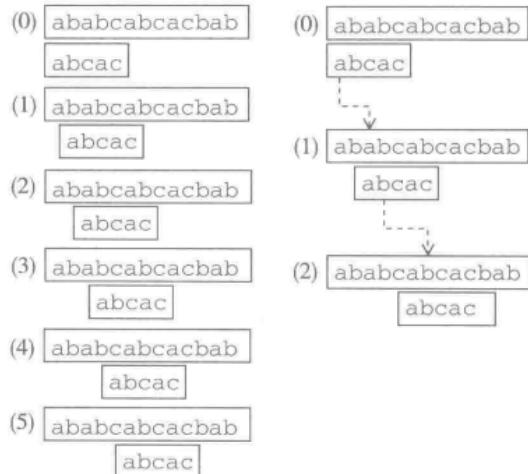


图 4.3 朴素匹配和 KMP 匹配过程

不同，不可能有匹配。KMP 算法直接把模式串的 b 移到刚才匹配 c 失败的位置（前面字符 a 肯定匹配，不必再试），达到状态(2)。接下去从模式串的 b 继续匹配，找到了一个成功匹配。在这个过程中未出现重新检查目标串前面字符的情况（无回溯）。

问题分析

KMP 算法的基本想法是匹配中不回溯。如果匹配中用模式串里的 p_i 匹配某个 t_j 时失败了（遇到了 $p_i \neq t_j$ 的情况），就找到某个特定的 k_i ($0 \leq k_i < i$)，下一步用模式串中字符 p_{k_i} 与目标串里的 t_j 比较。也就是说，在匹配失败时把模式串前移若干位置，用模式串里匹配失败字符之前的某个字符与目标串中匹配失败的字符比较。

要实现这种策略，关键在于确定匹配失败时模式串如何前移。也就是说，对模式串里的每个字符的 p_i ，必须能找到相应的位置 k_i 。这里出现了一个问题：对于匹配失败的 p_i ，无论目标串里的 t_j 怎样，与之对应的下标 k_i 都一样吗？如果是，就可能预先把这些对应关系计算出来，在匹配中直接使用；如果不是这样，上面设想的策略就没有价值了。

KMP 算法设计中的关键认识是：在 p_i 匹配失败时，所有的 p_k ($0 \leq k < i$) 都已经匹配成功（否则就不会考虑 p_i 的匹配问题）。这也就是说，在目标串中 t_j 之前的所有字符也就是模式串 p 的前 i 个字符 $p_0 p_1 p_2 \dots p_{i-1}$ 。这一情况说明，原本似乎应该根据目标串 t 中 t_j 之前已匹配的一段来决定模式串的前移位置，实际上只需根据模式串 p 本身的情况就可以决定了。这说明，完全可以在实际地与任何目标串匹配之前，通过对模式串本身的分析，解决好匹配失败时应该怎样前移的问题。

从上面分析中得到的结论是：对 p 中的每个 i ，都有与之对应的下标 k_i ，与被匹配的目标串无关。有可能通过对模式串 p 的预分析，得到每个 i 对应的 k_i （为每个 p 找到与之对应的 p_{k_i} ）。假设模式串 p 的长度是 m ，现在需要对每个 i ($0 \leq i < m$) 计算出对应的 k_i 并将其保存起来，以便在匹配中使用。为此可以考虑用一个长为 m 的表 pnext ，用表元素 $\text{pnext}[i]$ 记录与 i 对应的 k_i 值。

这里还有一种特殊情况：在一些 p_i 的匹配失败时，有可能发现在用 p_i 匹配之前做过的所有模式串字符与目标串字符的比较都没有实际利用价值。在这种情况下，下一步就应该从头开始，用 p_0 匹配与 t_{j+1} 比较。如果遇到这种特殊情况时，就在 $\text{pnext}[i]$ 里存入 -1 。显然，对任何一个模式，都有 $\text{pnext}[0] = -1$ 。

KMP 算法

现在假设已经根据模式串做出了 pnext 表，考虑 KMP 算法的实现。

核心的匹配循环很容易写出，如下：

```
while j < n and i < m:
    if i == -1:
        j, i = j+1, i+1
    elif t[j] == p[i]:
        j, i = j+1, i+1
    else:
        i = pnext[i]
```

显然前两个 if 分支可以合并，循环简化为：

```
while j < n and i < m:
    if i == -1 or t[j] == p[i]:
        j, i = j+1, i+1
    else:
        i = pnext[i]
```

下面是基于上述循环的匹配函数定义：

```
def matching_KMP(t, p, pnext):
    """ KMP串匹配，主函数."""
    j, i = 0, 0
    n, m = len(t), len(p)
    while j < n and i < m:
        if i == -1 or t[j] == p[i]:
            j, i = j+1, i+1
        else:
            i = pnext[i]
    if i == m:
        return j-i
    return -1
```

现在考虑这个算法的复杂性，关键是其中循环的执行次数。注意，在整个循环中 j 的值是递增的，但其加一的总次数不会多于 $\text{len}(t)$ 。而且， j 递增时 i 的值也递增。而在 if 的另一分支，语句 $i=pnext[i]$ 总使 i 值减小。但 if 的条件又保证变量 i 的值不小于 -1（表示素值不会小于 -1，一旦等于 -1，下次迭代就会走另一个分支），因此 $i=pnext[i]$ 的执行次数不会多于 i 值递增的次数。由这些情况可知，循环次数不会多于 $O(n)$ ，因此这个算法复杂性也是 $O(n)$ ， n 为目标串的长度。

构造 pnext 表：分析

现在考虑 $pnext$ 表的构造，以图 4.4 中的情况为例（即图 4.3 右列状态(1)）。这时位置 i 的字符是最后一个 c ，对应的位置 k_i 的字符应该是 b 。可见：

- 模式串移动之后，作为下一个用于匹配的字符的新位置，其前缀子串应该与匹配失败的字符之前同样长度的子串相同。
- 如果匹配在模式串的位置 i 失败时，而位置 i 的前缀子串中满足上述条件的位置不止一处，那么只能做最短的移动，将模式串移到最近的那个满足上述条件的位置，以保证不遗漏可能的匹配。

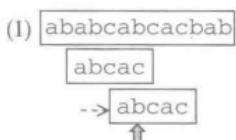


图 4.4 构造 $pnext$ 表

现在考虑 $pnext$ 表的构造问题，首先已知 k_i 的值只依赖于模式串本身的前 i 个字符。在下面的分析和讨论中参考了图 4.5。

首先，如图 4.5(1) 所示，目标串中位置 j 之前的 i 个字符也就是模式串的前 i 个字符，也就是说，目标串中的子串 $t_{j-i} \dots t_{j-1}$ 就是 $p_0 \dots p_{i-1}$ 。

现在需要找到一个位置 k ，下次匹配用 p_k 与前面匹配失败的 t_j 比较，也就是把模式串移到 p_k 与 t_j 对准的位置，如图 4.5(2) 所示。如果移动得正确，模式串里的子串 $p_0 \dots p_{k-1}$ 就应该与子串 $p_{i-k} \dots p_{i-1}$ 匹配。而这两个子串分别为串 $p_0 \dots p_{i-1}$ 的长度为 k 的前缀和后缀。这样，确定 k 的问题

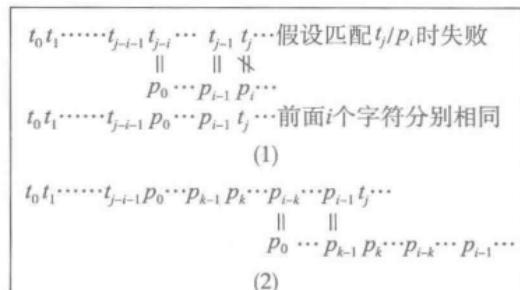


图 4.5 表 $pnext$ 的分析与构造

就变成了确定 $p_0 \dots p_{i-1}$ 的相等前缀和后缀的长度。显然， k 值越小表示移动越远。前面说移动距离应尽可能短。与之对应，应该找的 k 是 $p_0 \dots p_{i-1}$ 的最长的相等前缀和后缀（不包括 $p_0 \dots p_{i-1}$ 本身，但可以是空串）的长度，这样才能保证不会跳过可能的匹配。

从图 4.5 中可以看到, 如果 $p_0 \cdots p_{i-1}$ 的最长相等前后缀的长度为 k ($0 \leq k < i-1$), 在 $p_i \neq t_j$ 时, 模式串就应该右移 $i-k$ 位, 这也就是说, 应该把 $\text{pnext}[i]$ 设置为 k 。

现在只需对模式串里的每个位置 i , 求出模式串的子串 $p_0 \cdots p_{i-1}$ 的最长相等前后缀的长度。KMP 算法的设计者们提出了一种巧妙的递推算法。

递推计算最长相等前后缀的长度

有关递推计算的情况请参考图 4.6。假设现在要对子串 $p_0 \cdots p_{i-1} p_i$ (也就是对位置 i) 递推计算最长相等前后缀的长度, 这时对 $\text{pnext}[i-1]$ 已经计算出结果为 $k-1$ 。比较 p_i 与 p_k , 有两种情况:

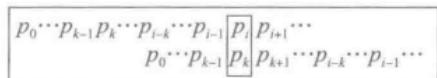


图 4.6 最长相等前后缀长度的递推计算

- 1) 如果 $p_i = p_k$, 那么对于 i 的最长相等前后缀的长度, 比对 $i-1$ 的最长相等前后缀的长度多 1, 由此应将 $\text{pnext}[i]$ 设置为 k , 然后考虑下一个字符。

- 2) 否则就应该把 $p_0 \cdots p_{k-1}$ 的最长相等前缀移过来继续检查。

注意, 第 2 种情况并没有设置, 只是继续检查。不难确定, $p_0 \cdots p_{k-1}$ 的最长相等前缀也是 $p_0 \cdots p_{i-1}$ 的相等前缀。移过来后继续检查是正确的。

已知 $\text{pnext}[0]=-1$ 和直至 $\text{pnext}[i-1]$ 的已有值求 $\text{pnext}[i]$ 的算法:

- 1) 假设 $\text{pnext}[i-1]=k-1$ 。如果 $p_i=p_k$, 那么 $p_0 \cdots p_i$ 的最长相等前后缀的长度就是 k , 将其记入 $\text{pnext}[i]$, 将 i 值加一后继续递推 (循环)。
- 2) 如果 $p_i \neq p_k$, 就将 k 设置为 $\text{pnext}[k]$ 的值 (将 k 设置为 $\text{pnext}[k]$, 也就是转去考虑前一个更短的保证匹配的前缀, 可以基于它继续检查)。
- 3) 如果 k 的值等于 -1 (这个值一定是由于第 2 步而来自 pnext), 那么 $p_0 \cdots p_i$ 的最长相等前后缀的长度就是 0, 设置 $\text{pnext}[i]=0$, 将 i 值加 1 后继续递推。

pnext 表构造算法

这里是基于上面分析定义的算法:

```
def gen_pnext(p):
    """生成针对p中各位置i的下一检查位置表, 用于KMP算法"""
    i, k, m = 0, -1, len(p)
    pnext = [-1] * m      # 初始数组元素全为 -1
    while i < m-1:        # 生成下一个pnext元素值
        if k == -1 or p[i] == p[k]:
            i, k = i+1, k+1
            pnext[i] = k # 设置pnext元素
        else:
            k = pnext[k] # 退到更短相同前缀
    return pnext
```

函数开始时建立一个元素值全为 -1 的表, 循环中为下标 0 之后的各元素赋值。具体处理完全按上面分析的情况, 例如: `gen_pnext("abbcabcaabbcaa")` 将给出表 `[-1, 0, 0, 0, 0, 1, 2, 0, 1, 1, 2, 3, 4, 5]`, 具体情况请读者自己分析。

易见, 这个算法的形式与前面 KMP 串匹配的主函数极为类似, 算法分析的情况可以照搬, 结论是本算法的时间复杂性为 $O(m)$, 其中 m 是模式串长度。

pnext 生成算法的改进

在 pnext 生成算法里, 设置 $\text{pnext}[i]$ 时还可以做些优化。有关情况参看图 4.5 的(1)和(2), 由于匹配失败时有 $p_i \neq t_j$ 。设 $\text{pnext}[i]=k$, 如果发现 $p_i=p_k$, 那么也就一定有 $p_k \neq t_j$ 。所

以，在这种情况下，实际上模式串应该右移到 $p_{\text{next}}[k]$ (而不是仅右移到 $p_{\text{next}}[i]$)，下一步应该用 $p_{\text{next}}[k]$ 与 t 比较。这一修改可以把模式串右移更远，有可能提高效率。修改后的函数定义如下：

```
def gen_pnext(p):
    """生成针对p中各位置i的下一检查位置表，用于KMP算法，有稍许修改的优化版本。
    """
    i, k, m = 0, -1, len(p)
    pnext = [-1] * m
    while i < m-1: # 生成下一个pnext元素
        if k == -1 or p[i] == p[k]:
            i, k = i+1, k+1
            if p[i] == p[k] :
                pnext[i] = pnext[k]
            else:
                pnext[i] = k
        else:
            k = pnext[k]
    return pnext
```

还用上面的例子，`gen_pnext("abbcabcaabbcaa")` 给出的表是 $[-1, 0, 0, 0, -1, 0, 2, -1, 1, 0, 0, 0, -1, 5]$ 。不难证明，改进算法产生的 p_{next} 表里的各元素不会大于原算法产生的表中对应的元素。

KMP 算法的时间复杂性及其他

显然，一次 KMP 算法的完整执行包括构造 p_{next} 表和实际匹配，设模式串和目标串长度分别为 m 和 n ，KMP 算法的时间复杂性是 $O(m+n)$ 。由于在多数通常情况 $m \ll n$ ，因此可以认为这个算法复杂性为 $O(n)$ ，显然优于朴素匹配算法的 $O(m \times n)$ 。

注意，许多场景中需要用一个模式串反复在一个或多个目标串里匹配。例如，在文本里查找 / 替换某个字符串就是典型例子。在这种情况下，构造模式串的 p_{next} 表的工作只需做一次，后面匹配中只须简单地反复使用。这是最适合 KMP 算法的场景。

如果要处理的是这种情况，可以考虑定义一个模式类型，将 p_{next} 表作为模式对象的一个成分，在以模式串作为参数构造模式对象时做好对象的 p_{next} 表。实际匹配函数使用这种模式对象去匹配目标串。这一工作留作练习。

KMP 算法的一个重要优点是执行中不回溯。这种性质在一些实际应用中特别有价值，因为它支持一边读入一边匹配，不回头重读就不需要保存被匹配的串。在处理从外部（外存或网络等）获取的大量信息时，这种算法非常合适。

前面说过，人们还提出了其他的模式匹配算法。另一个经典算法由 R. S. Boyer 和 J. S. Moore 提出，其中采用自右向左的匹配方式，其中也用了一个失败匹配移动表。如果字符集较大而且匹配很罕见（许多实际情况就是如此，如在文章里找单词，在邮件里找垃圾过滤关键字），其速度可能远远高于 KMP 算法。有兴趣的读者可以自己找相关材料看一看。

4.4 字符串匹配问题

前面几节讨论的字符串匹配是最简单的情况，匹配基于最简单的字符比较，其中的模式串就是普通字符串，所做匹配是在目标串里查找等于模式串的子串。也就是说，比较的一方是表示模式的字符串，另一方是目标字符串的所有可能子串。实际中还有很多与此类似，但又有或

多或少差异的需要，现在讨论一些情况。

4.4.1 串匹配 / 搜索的不同需要

通过简单的串匹配，在长字符串（文本）里找到与之相同的子串，或者从许多字符串里选出一些串，都是字符串匹配 / 搜索。基本串匹配有很广泛的应用，前面举过一些例子，如正文编辑器中最常用的查找和替换操作，网络搜索引擎在网页中检查检索串等。

对简单串匹配的实际使用，也存在许多不同的场景，例如用一个模式串在目标串里反复检索找出一些或者所有出现，在一个目标串里检查是否出现了一组模式串中的任何一个，或者在一批目标串里检查一个或一组模式串是否出现，等等。

但另一方面，在实际中人们需要查找的可能不是某个特定的字符串，而是具有某种形式的字符串。例如：

- 找出一个文本中所有双引号括起的词语。
- 找出图书馆中书名包括“数据结构”和“算法”两个词的所有书籍。
- 找出一个 Python 程序里所有三个参数的函数定义的函数名。
- 找出一个网页文件里所有形为 href="..." 的段（这是 HTML 网页里的链接）。
- 找出一个 DNA 片段中所有以某碱基段开始以另一碱基段结束的片段。
- 找出计算机可执行文件中的某种片段模式（例如检查病毒），以一种形式的片段开始到另一片段结束，其中出现了某些片段。

在这种匹配需求中，需要考虑的不是一个模式串，而是一组模式串，可能有穷，也可能无穷。显然，简单罗列（枚举）模式串的方式不能适应这里的需要，因为需要考虑的模式串可能很多，甚至无穷多。要想处理这类推广了的字符串匹配问题，就需要考虑字符串集合的描述，以及对于是否属于一个字符串集合的检查。

注意，上述检查需求有共性：它们都只牵涉到字符串的表面形式，并不牵涉其意义。例如，找出一批数字串（看作整数）中值大于 1573 的串，就不是字符串匹配问题，因为牵涉到字符串形式之外的“意义”问题。在这里讨论的匹配中，不需要计算一个一般的谓词函数，只需独立地检查一个个字符等于什么或者不等于什么。

模式，字符串和串匹配（串检索）

为了下面的讨论，需要重新定义“模式”的概念。这里考量的模式具有一种特定的表达形式，一个模式描述了（能匹配）一些字符串（一个字符串集合）。易见，前面几节讨论的模式串是这个模式概念的特例：模式的形式就是普通的字符串；每个模式描述的字符串集合都是单元素集合，其中只包含这个字符串本身；匹配的条件就是字符串相等。对于一般的模式，都需要回答这三个问题：模式的形式是什么，描述的字符串集合如何确定，怎样做匹配。显然，判断是否匹配（或者找出匹配）是一个可能的计算问题。

在考虑字符串集合的描述和匹配时，需要考虑两个问题：

- 为了描述（希望关注的）字符串集合，需要一种严格描述方式。这种方式应能描述很多有用的字符串集合。一种“系统化的”描述方式，就是一种描述串模式的语言（简单字符串匹配的“模式语言”就是字符串本身）。
- 如何（或者是否有可能）高效实现所希望的检查（匹配）。

如果模式描述语言的功能更强，就有可能描述更多更复杂的模式（对应的，描述更多字符串集合），但相应匹配算法的复杂性也可能变高。

人们在这方面做了许多研究，有许多重要的理论结果。当模式语言变得比较复杂以后，或许只能做出具有指数复杂性的匹配算法，这种情况使模式语言丧失了实用意义；如果模式语言进一步复杂，其模式匹配问题甚至可能变为不可计算问题。也就是说，根本不可能写出完成匹配的算法，这样的描述语言就完全没有价值了。由此，有意义（有价值）的模式描述语言是描述能力和处理效率之间的合理平衡。

通配符和简单模式语言

如果读者对 DOS 操作系统或者 Windows 命令窗口（cmd）有所了解，就可能知道其中用于描述文件名的“通配符”。在普通字符串里加入几个特殊的通配符，就构成了一种简单的模式描述语言。这里简单介绍一下。

DOS 或 Windows cmd 的一些命令中需要写被操作的文件名。例如命令 dir 要求列出文件及其相关信息，可以用一种文件名描述形式要求列出一些文件的信息。在有关文件名（路径）的描述中，可以使用两个通配符“*”和“？”，符号“？”可以与任意一个实际字符匹配，符号“*”可以与任意一串字符匹配。这样就可以描述一组文件。

例如，*.py 描述的是所有最后三个字符是 .py 的文件名，因此它将与所有以 py 为扩展名的文件名匹配。f*.* 描述的字符串集合里包含所有以字母 f 开始，其后有任意多个字符和一个圆点符号，再有任意多个字符的字符串。a?b 描述所有 3 个字符的串，其首字符为 a，尾字符为 b，中间的字符可以任意。

在普通字符串的基础上增加通配符，就形成了一种功能更强一点的模式描述语言。这里的一个模式描述了一字符串集，能与该集合中任何一个串匹配。增加了可以与任意长的字符串匹配的“*”符号，就能描述无穷的字符串集合，例如 a* 描述了所有以 a 开头的字符串，这是一个无穷集合。上面有关文件名的两个描述也是这种情况。

但是，仅仅加入了通配符的模式语言还不够灵活，描述能力不强，能描述的字符串集合很受限。因此，需要考虑描述能力的进一步扩充。

正则表达式

一种非常有意义和实用价值的模式语言称为正则表达式（Regular Expression，也被简写为 regex、regexp，或者 RE/re）。这种描述形式由逻辑学家 S. Kleene 提出，现在已经在计算机科学技术领域得到了广泛应用。4.5 节将介绍 Python 标准库支持的正则表达式功能，下面先从理论上介绍正则表达式的一些基本情况。

正则表达式是一种描述字符串集合的语言，其基本成分是字符集里的普通字符、几种特殊组合结构，以及表示组合的括号。一个正则表达式描述了字符集上的一个特定字符串集合。理论的正则表达式如下定义，其中的 α 和 β 都是正则表达式：

- 正则表达式里的普通字符只与该字符本身匹配。
- 顺序组合 $\alpha\beta$ ：如果 α 能匹配字符串 s ， β 能匹配字符串 t ，那么这两个正则表达式的连写 $\alpha\beta$ 就能匹配字符串拼接 $s+t$ 。
- 选择组合 $\alpha|\beta$ ：如果 α 能匹配字符串 s ， β 能匹配字符串 t ，它们的选择组合 $\alpha|\beta$ 既能匹配 s ，也能匹配 t 。
- 星号 α^* ：如果 α 能匹配字符串 s ，星号表达式 α^* 能匹配空串、 s 、 $s+s$ 、 $s+s+s$ 等。也就是说，它能与 0 段或任意多段 s 的拼接串匹配。

上面说明中借用了 Python 的字符串拼接写法。理论著作里大多直接用 st 表示 s 和 t 的拼接。正则表达式中还可以用括号表示表达式的组合，见下面例子。

这里是几个例子：

- 正则表达式 abc 只与字符串 abc 匹配。
- $a(b^*)(c^*)$ 与所有包含一个 a 之后任意多个 b，再之后任意多个 c 的串匹配。
- $a((b|c)^*)$ 与所有在一个 a 后出现任意多个 b 和 c 的串匹配。

这里说的“任意多个”也可以是 0 个。

注意，从理论上说通配符并不必要。因为字符集有穷，所以前面介绍的通配符“?”可以通过选择组合符“|”描述；通配符“*”可以通过“|”和星号描述。

正则表达式在信息处理中非常有用，人们做出了许多实现。实际使用的正则表达式有不同的设计，它们都是理论的正则表达式的子集或者变形，基于对易用性和实现效率等方面的考虑，有些还有所扩充。各种脚本语言（包括 Python）都提供了正则表达式功能，常规语言也正在或计划把正则表达式纳入其标准库，如 C、C++、Java 等语言都有正则表达式包。经过人们在 Perl 语言的设计中对正则表达式概念的精炼，现在已经基本形成了一套比较标准的形式。可以找到许多介绍正则表达式的书籍或文章，一些人把正则表达式说成是程序员必备的重要武器，网上也有很多对正则表达式的讨论。

4.4.2 一种简化的正则表达式

正则表达式是一种比较复杂的模式描述语言，完成其与字符串匹配的高效算法是一件比较复杂的工作。利用正则表达式与有穷自动机的等价性，人们已经开发出几种经典的实现技术。由于比较复杂，在这里不进一步讨论。

但作为一个例子，也帮助读者理解正则表达式匹配中的一些情况，下面介绍一个简化版本的正则表达式，从中可以看到匹配算法实现中的一些情况[⊖]。

模式语言

这里考虑一种简化的正则表达式，其中包含下面的描述元素：

- 任一字符仅与其本身匹配。
- 圆点符号“.”可以匹配任意字符。
- 符号“^”只匹配目标串的开头，不匹配任何具体字符。
- 符号“\$”只匹配目标串的结束，不匹配任何具体字符。
- 符号“*”表示其前面那个字符可匹配 0 个或任意多个相同字符。

其中“^”和“\$”符号至多在模式的开始和结束出现一次。

上述规则定义了一种简单的模式描述语言。这里也假定了一个默认的字符集，第一条所用的字符取自该集合。与一般的正则表达式相比，在这里没有括号，也没有选择描述符，因此描述能力大大降低了。下面是几个模式示例：

```
p1 = "a*b.*"
p2 = "^ab*c.$"
p3 = "aa*bc.*bca"
```

匹配算法

这里给出一个朴素的匹配算法 `match`，其中定义了两个局部函数，分别处理从模式中某

⊖ 本例源自《代码之美》(Beautiful Code)一书的第 1 章，由 B. Kernighan 撰写，原程序用 C 语言描述。这里用 Python 重新编写，实现方式做了全面修改。

个位置开始的匹配和模式中的星号：

```
def match(re, text):
    def match_here(re, i, text, j):
        """检查从text[j]开始的正文是否与re[i]开始的模式匹配"""
        while True:
            if i == rlen:
                return True
            if re[i] == '$':
                return i+1 == rlen and j == tlen
            if i+1 < rlen and re[i+1] == '*':
                return match_star(re[i], re, i+2, text, j)
            if j == tlen or (re[i] != '.' and re[i] != text[j]):
                return False
            i, j = i+1, j+1

    def match_star(c, re, i, text, j):
        """在text里跳过0个或多个c后检查匹配"""
        for n in range(j, tlen):
            if match_here(re, i, text, n):
                return True
            if text[n] != c and c != '.':
                break
        return False

    rlen, tlen = len(re), len(text)
    if re[0] == '^':
        if match_here(re, 1, text, 0):
            return 1
    for n in range(tlen):
        if match_here(re, 0, text, n):
            return n
    return -1
```

主函数里首先考虑模式开头是否为“^”符号，如果是，就只考虑与目标串前缀的匹配。否则就对 text 中的各个位置调用 match_here，考虑是否匹配。如果找到了匹配，函数返回匹配的开始位置，否则返回 -1 表示无匹配。

函数 match_here 检查从模式串 re 中位置 i 开始的部分是否与正文串 text 中位置 j 开始的部分匹配。函数里用了一个循环，循环体里是一系列的 if 语句。i == rlen 表示模式串已经处理完了，这说明找到了匹配。第二个 if 处理模式最后的“\$”符号，如果模式串和正文串都结束就是匹配成功。第三个 if 处理“*”符号，调用专门的函数 match_star。最后一个 if 检查几种匹配失败的情况，条件为真时返回匹配失败。否则就将变量 i 和 j 加一后继续。

函数 match_star 处理星号的情况，其第一个参数是允许任意次重复的那个字符。这个函数的实现很简单：从正文中剩下的每个位置开始检查能否与星号之后的模式匹配；否则检查正文串中下一字符是否与可重复字符匹配，如果匹配就跳过它并继续。

可以对这里的模式语言做些扩充，简单的扩充留作练习。

4.5 Python 正则表达式

大多数主要编程语言都在语言本身或者通过标准库提供了正则表达式功能。另一方面，正则表达式（或类似描述方式）在实际程序和系统中使用广泛。为帮助读者了解这方面情况，作为本章最后一节，下面简单介绍 Python 语言的正则表达式包。如前所述，由于在 Perl 语言里的精炼，各种实际语言或程序库中对正则表达式的描述已经趋向统一。了解 Python 正则表达式

式的一些情况后，转去使用其他正则表达式功能也会很容易。

4.5.1 概况

Python 语言的正则表达式功能由标准包 `re` 提供。利用正则表达式可以较容易地实现许多复杂字符串操作。要想正确地使用 `re` 包，需要：

- 理解正则表达式的描述规则（语法）和效用（语义）。
- 了解正则表达式的一些典型使用方法和情景。

Python 正则表达式采用字符串字面量的形式描述（即引号括起的字符序列）。从 Python 语言的角度看它们就是普通的字符串，但在用于 `re` 包提供的一些特殊操作时，一个具有正则表达式形式的字符串代表一个字符串模式，描述了一个特定的字符串集合。这类操作就是 `re` 包提供的正则表达式匹配操作。

`re` 包的正则表达式描述形式实际上构成一个特殊的小语言：

- 语法：`re` 包规定的一套特殊描述规则，符合这组规则的字符串就是正则表达式，可以用在 `re` 包提供的各种匹配操作中。
- 语义：一个正则表达式所描述的字符串集，这也是下面介绍的重点。

在 Python 官方文档中，有关 `re` 包的章节给出了正则表达式描述的各方面细节。

4.5.2 基本情况

本小节先介绍一些基本情况。

原始字符串

在深入讨论 Python 正则表达式之前，首先介绍原始字符串（raw string，也是文字量）的概念。原始字符串是在 Python 里书写字符串文字量的一种形式，这种文字量的值和普通文字量一样，就是 `str` 类型的字符串对象。

原始字符串的形式是在普通字符串文字量前加 `r` 或 `R` 前缀，例如：

```
R"abcdefg"      r"C:\courses\python\progs"
```

原始字符串只有一点特殊，就是其中的反斜线字符“\”不作为转义符，在相应字符串对象（`str` 对象）里原样保留。但位于单 / 双引号前的反斜线符号仍作为转义符。

Python 语言引入原始字符串只是为了使一些字符串文字量的写法略微简单。例如，描述 Windows 文件路径的原始字符串 `r"C:\courses\python\progs"`，如果用普通文字量的形式，应该写成 `"C:\\courses\\python\\progs"`，其中的反斜线符号需要双写，有点麻烦。如果用普通文字量形式写模式串，希望去匹配正文里的“\”字符，情况更麻烦。要匹配一个“\”字符需要写四个反斜线“\\\\\\”。有关详情参看 Python 官方文档的 HOWTO 部分。在后面介绍中将会看到原始字符串的几种常用情况。

元字符（特殊字符）

正则表达式包 `re` 规定了一组特殊字符，称为元字符。在匹配字符串时，它们起着特殊的作用。这种字符一共有 14 个：

```
. ^ $ * + ? \ | { } [ ] ( )
```

首先应注意，在（普通）字符串里，或者说在字符串的普通使用中，这些字符都是普通字符（“\”除外），只有在把字符串作为正则表达式，用于 `re` 包提供的一些特殊操作时，这些字符

才有特殊的意义。下面将把作为正则表达式使用的串称为模式串，将非特殊字符称为常规字符，它们是描述正则表达式的基础。在作为正则表达式使用时，模式串里的常规字符只与该字符自身匹配。所以，如果一个模式串里只包含常规字符，这个串就只能与自己匹配。也就是说，常规字符串是最基本的正则表达式。

4.5.3 主要操作

在详细介绍正则表达式中元字符的意义和使用技术之前，现在先介绍 re 包提供的几个操作。可以通过这些操作使用正则表达式（还有其他操作，后面介绍）。在下面的函数说明中，参数表里的 pattern 表示模式串（描述正则表达式的字符串），string 表示被处理的字符串，repl 表示替换串，即操作中使用的另一个字符串。

- 生成正则表达式对象：re.compile(pattern, flag=0)

本操作从 pattern 生成与之对应的正则表达式对象，这种对象可以用于下面介绍的几个操作。操作实例如：

```
r1 = re.compile("abc")
```

这个语句生成与 "abc" 对应的正则表达式对象，并将其赋给变量 r1。

说明：re 包的许多操作都有一个 flag 选项，并专门为操作的这一选项参数提供了一组特殊标记。这方面的功能是不重要的细节，下面讨论中将不涉及。

实际上，下面几个操作都能自动地从 pattern 串生成正则表达式对象。但如果一个模式串需要反复使用，先用 compile 生成正则表达式对象并记入变量，可以避免在实际使用中重复生成。下面几个函数都有一个 pattern 参数，既可以用模式串作为实参，也可以用正则表达式对象作为实参。

- 检索：re.search(pattern, string, flag=0)

在 string 里检索与 pattern 匹配的子串。如找到就返回一个 match 类型的对象；否则返回 None。match 对象里记录成功匹配的相关信息，可以根据需要检查和使用。也可以把 match 对象简单作为真值用于逻辑判断。有关细节将在下面介绍。

- 匹配：re.match(pattern, string, flag=0)

检查 string 是否存在一个与 pattern 匹配的前缀。匹配成功时返回相应的 match 对象，否则返回 None。例如：

```
re.search(r1, "aaabcbcbaabc") #将匹配成功  
re.match(r1, "aaabcbcbaabc") #返回None
```

- 分割：re.split(pattern, string, maxsplit=0, flags=0)

以 pattern 作为分割串将 string 分段。参数 maxsplit 指明最大分割数，用 0 表示要求处理完整个 string。函数返回分割得到的字符串的表。例如：

```
re.split(' ', "abc abb are not the same")  
#得到： ['abc', 'abb', 'are', 'not', 'the', 'same']  
re.split(" ", "1 2 3 4") # 分割出了几个空串  
#得到： ['1', '2', '', '3', '', '', '4']
```

- 找出所有匹配串：re.findall(pattern, string, flags=0)

返回一个表，表中元素是按顺序给出的 string 里与 pattern 匹配的各个子串（从左到右，非重叠的）。如果模式里只有常规字符，做这种匹配的价值不大，因为结果表里的所有字符串相

同。但用一般的正则表达式模式，情况就可能不同。

4.5.4 正则表达式的构造

还有一些操作将在后面介绍。这一小节里将逐步介绍正则表达式的一些具体情况。阅读中需要注意两点：

- 一种表达式（或者元符号）的描述形式（构造形式）。
- 这种表达式能匹配怎样的字符串（集合）。

正则表达式的最基本构造方式是顺序组合 $\alpha\beta$ ：如果 α 能匹配字符串 s , β 能匹配字符串 t , 那么这两个正则表达式的连写 $\alpha\beta$ 就能匹配字符串拼接 $s+t$ (s 、 t 为字符串, 按 Python 表达式的写法, $s+t$ 是两个字符串的拼接)。

在模式串里, 可以用圆括号表示结组, 确定模式描述符的作用范围。下面会看到, 在有些情况下必须使用括号。圆括号还有其他作用, 后面介绍。

此外, 在正则表达式里同样可以写普通 Python 字符串中使用的转义字符描述形式, 例如, `\n` 表示换行、`\t` 表示制表符等。这些表示的都是匹配中的常规字符。正则表达式里的空格也作为常规字符, 因此只能与自己匹配。

字符组

有一些表达方式描述了一组字符, 或者说, 某些正则表达式可以与一组字符中的任何一个字符匹配。这类描述形式称为字符组描述。

字符组描述符 [...] : 与方括号中列出的字符序列中的任一字符匹配。字符组里字符的排列顺序并不重要。例如, `[abc]` 可以与字符 `a` 或 `b` 或 `c` 匹配。

在字符组描述的括号里, 除了可以列出一组字符外, 还有一些其他形式:

- 区间形式是(顺序列出字符形式的)字符组描述的缩写形式。例如, 字符组 `[0-9]` 匹配所有十进制数字字符。可以用任何一对数字或(大小写)字母写出区间形式的描述, 这种字符组与介于两字符之间的所有字符匹配, 包括这两个字符。区间形式还可以连写, 或者与字符列表混写, 如 `[34ad-fs-z]`。显然, `[0-9a-zA-Z]` 能匹配所有的数字和字母(英文字母)。
- 特殊形式 `[^...]` 表示对 `^` 之后列出的字符组求补, 也就是说, 这种字符组表达式与所有未列在括号里的字符匹配。例如, 字符组 `[^0-9]` 匹配非十进制数字的所有字符, `[^\t\v\n\f\r]` 匹配所有非空白字符(除去空格/制表符/换行符)。

例如, `a[1-9][0-9]` 包含字符组的顺序, 它能匹配 `a10`, `a11`, ..., `a99`。

如果需要在字符组里包括字符“`^`”, 就不能把它直接放在第一个位置。可以放在后面, 或者写“`\^`”。如果需要在字符组里包括“`-`”或“`]`”, 也必须写“`\-`”或“`\]`”。

圆点字符(.) : 圆点是“通配符”, 它能匹配任何字符。

例如, 模式串 `a..b` 匹配所有以 `a` 开头 `b` 结束的四字符串。

为了方便, `re` 采用转义串的形式定义了一些常用字符组, 包括:

- `\d`: 与十进制数字匹配, 等价于 `[0-9]`。
- `\D`: 与非十进制数字的所有字符匹配, 等价于 `[^0-9]`。
- `\s`: 与所有空白字符匹配, 等价于 `[\t\v\n\f\r]`。
- `\S`: 与所有非空白字符匹配, 等价于 `[^\t\v\n\f\r]`。
- `\w`: 与所有字母数字字符匹配, 等价于 `[0-9a-zA-Z]`。

- \w: 与所有非字母数字字符匹配, 等价于 [^0-9a-zA-Z]。

还有一些类似描述, 提供这些特殊字符只是为了使用方便。根据上面的说明可知, 这些特殊字符组都可以直接描述。

例如, p\w\w\w 与以 p 开头随后为任意三个字母数字的串匹配。

重复

实际中经常需要用到一个模式 (或模式中一个部分) 的重复匹配, 可能是任意多次的重复匹配或规定次数的重复匹配。

重复描述符: re 正则表达式的基本重复运算符是 “*”, 模式 α^* 要求匹配模式 α 能匹配的字符串的 0 次或任意多次重复。例如, `re.split('[,]*', s)` 将按 (任意多个) 空格和逗号切分字符串 s。例如:

```
re.split('[ , ]*', '1 2, 3 4, , 5')      得到 ['1', '2', '3', '4', '5']
re.split('a*', 'abbaaabbbabbababbabb')
得到 ['', 'bb', 'bbbbb', 'bb', 'b', 'bb', 'bb']
```

在考量重复匹配时, 有一个问题值得注意。先用一个例子说明, 计算 `re.match('ab*', 'abbbbbbc')` 时, 模式 'ab*' 可以与字符串里的 a 匹配, 也可以与 ab 匹配, 等等。那么它究竟匹配哪个串呢? 一般正则表达式匹配都提供了两种可能:

- 贪婪匹配: 模式与字符串里有可能匹配的最长子串匹配。对上面例子采用贪婪匹配规则, 模式 ab^* 就应该匹配 `abbbbb`。`re` 规定 “*” 运算符做贪婪匹配。
- 非贪婪匹配 (吝啬匹配): 模式与有可能匹配的最短子串匹配。对所有重复模式描述符, Python 都提供了相应的吝啬匹配描述符。

还有一个与 “*” 略微不同的重复运算符 “+”, 表示其作用模式的 1 次或多次重复。与 “*” 描述符不同, “+” 描述符要求至少一次匹配。

例: 描述正整数的一个简单模式是 '\d+', 它等价于 '\d\d*'。

可选描述符: “?” 是可选 (片段) 描述符表示, “ $\alpha?$ ” 可以与空串或与 α 匹配的字符串匹配, 也就是说, 它要求与 α 匹配的字符串的 0 或 1 次重复匹配。

例: 描述整数 (表示整数的字符串) 的一种简单模式是 '-?\d+'。当然, 这种模式也可以与数字 0 的串匹配。

重复次数描述符: 确定次数的重复用 $\{n\}$ 描述, $\alpha\{n\}$ 与 α 匹配的串的 n 次重复匹配。

例: 描述北京固话号码的模式串可以写 '(010-)?[2-9][0-9]{7}'。它说明, 这种串总以 010- 开始, 随后是数字 2 到 9, 再后是 7 个十进制数字。

请注意: 在许多实际情况中, 人们写出的正则表达式描述的是实际字符串集合的超集。上面就是一例。显然, 并不是每个号码都对应一个电话。但这种描述通常可以满足需要。例如, 上面模式可用于在文本中检索北京市的固话号码。

在这个实例中出现了圆括号, 用于描述 “?” 的作用范围, 这对圆括号不能少。显然, *、?、{3} 都有作用范围问题 (优先级), `re` 包规定它们都作用于最小表达式。例如, '010-?' 表示字符串中的 '-' 可选, 而 '(010-)?' 表示整个段可选。

重复次数的范围描述符: 重复次数的范围用 $\{m,n\}$ 描述。例如, $\alpha\{m,n\}$ 与 α 匹配的串的 m 到 n 次重复匹配, 包括 m 次和 n 次。

例如, `a{3,7}` 与 3 到 7 个 a 构成的串匹配。`go{2,5}gle` 与 `google`、`gooogle`、`goooogle`、`oooooogle` 匹配。

重复范围描述符中的 m 和 n 是可以省略的， $\alpha\{n\}$ 等价于 $\alpha\{0, n\}$ ，而 $\alpha\{m,\}$ 等价于 $\alpha\{m, \text{infinity}\}$ 。易见，前面几种重复描述符都可以用这种形式表示：

- $\alpha\{n\}$ 等价于 $\alpha\{n, n\}$ ， $\alpha?$ 等价于 $\alpha\{0, 1\}$
- α^* 等价于 $\alpha\{0, \text{infinity}\}$ ， α^+ 等价于 $\alpha\{1, \text{infinity}\}$

$*$ 、 $+$ 、 $?$ 、 $\{m, n\}$ 都采取贪婪匹配规则，与被匹配串中最长的合适子串匹配（因为它们可能出现在更大的模式里，还需要照顾上下文的需要）。

非贪婪匹配描述符：与各种贪婪重复运算符对应，这里还有一组非贪婪匹配运算符。它们的形式分别是 $*?$ 、 $+?$ 、 $??$ 、 $\{m, n\}?$ （在上述各运算符后增加一个问号），其语义与上面几个运算符分别对应，但采用非贪婪匹配（最短匹配）策略。

选择

选择描述符：描述符“|”描述与两种或多种情况之一的匹配。如果 α 或者 β 与一个串匹配，那么 $\alpha|\beta$ 也与这个串匹配。

例如 $a|b|c$ 可以匹配 a 或者 b 或者 c 。 $[abc]$ 可以看作其简写，书写更简洁方便，还允许简写如 $[a-z]$ ，但只能用于单字符选择。用选择描述符可以写 $(ab)|cd$ ， $(ab^*)|c^*$ 。这些都不能用字符组的方式描述。

例：`'0|[1-9]\d*'` 可以匹配 Python 语言里的十进制整数（注意，Python 负号看作运算符，非 0 整数不能以 0 开头）。如果是针对独立的字段，可以用这个模式。但它还会与 0123 开头的 0 匹配，与 abc123、a123b 里的 123 匹配。它们在程序里不看作整数，都需排除。由此可见，实际匹配中还有上下文要求，后面考虑。

例如，匹配国内固定电话号码的模式：`'0\d{2}-\d{8}|0\d{3}-\d{7,8}'`。这样写考虑了区号有两位或三位，三位区号时局地号码为 7 位或 8 位。这个正则表达式描述的也是实际集合的超集。例如，实际中两位区号只有 010/020/021/022/023，这一段可以精化为 `0(10|20|21|22|23)-\d{8}`，另一选择段可以精化为 `0[3-9]\d{2}-\d{7,8}`。精化后的模式串匹配的字符串集合更小（模式更精确）。

“|”描述符的结合力最弱，比顺序组合的结合力还弱。因此在上面描述的许多地方没有用括号。

首尾描述符

几个描述符用于匹配一行或一个串的头尾。

行首描述符：以“`^`”符号开头的模式，只能与一行的前缀子串匹配。

例如，`re.search('^for', 'books for children')` 将得到 `None`。但是，`re.search('^for', 'books\nfor children')` 将匹配成功。

行尾描述符：以“`$`”符号结尾的模式只与一行的后缀子串匹配。

例如，`re.search('fish$', 'cats like to eat fishes')` 将得到 `None`，但 `re.search('like$', 'cats like\nnto eat fishes')` 将匹配成功。

注意，一行的前缀和后缀也包含整个被匹配串的前缀和后缀。如果串里有换行符，那么还包括换行符前的子串的后缀和换行符后的子串的前缀。见上例。

串首描述符：`\A` 开头的模式只与整个被匹配串的前缀匹配。

串尾描述符：`\Z` 结束的模式只与整个被匹配串的后缀匹配。

至此所有 14 个元字符已经全部介绍完毕。总结一下：其中有三对括号，分别用于描述优先级结合、字符组和重复次数；圆点是通配符；星号和加号表示重复；`^` 和 `$` 表示行首和行尾，

反斜线符号还是转义符。

这里还应该再特别提出转义字符 \，以它作为引导符定义了一批转义元字符，如 \d、\D 等。它还被用于在模式串里写各种非打印字符（如 \t、\n……）及其自身 (\\\) 等，用在字符串组描述 “[]” 里写 “\^”、“_” 和 “\]”。

单词边界

现在再介绍两个转义元字符，用于描述特殊子串的边界。

\b 描述单词边界，它在实际单词边界位置匹配空串（不匹配实际字符）。单词是字母数字的任意连续序列，其边界就是非字母数字的字符或者无字符（串的开头 / 结束）。

这里有一个比较糟糕的问题：在 Python 的字符串字面量里 \b 表示退格符，而在 re 的正则表达式里 \b 表示单词边界。处理这个问题有两种办法：

1) 将模式串里的 \ 双写，这就表示把 \ 本身送给 re.compile 等函数。例如 '\b123\b'

将不匹配 abc123a 里的 123，但匹配 (123, 123) 里的 123。

2) 用 Python 原始字符串，其中的 \ 不转义。上面模式可写为 r'\b123\b'。

实例：匹配 Python 程序里整数的模式可以写为 '\b(0|[1-9]\d*)\b'，或者用原始字符串形式简单写为 r'\b(0|[1-9]\d*)\b'。例如，

```
re_int = r'\b(0|[1-9]\d*)\b'
```

实例：如果希望匹配可能带正负号的一般整数，可以考虑用模式 '[+-]? \b(0|[1-9]\d*)\b'。但是这个模式还能匹配 x+5 里的 +5，却不能不匹配 3+-5 里的 -5（注意这里在负号和 5 之间有空格）。还是不够理想。这些情况说明，数值前面的字符串不是一个简单的字符串匹配问题。

易见，在书写和使用正则表达式时，需要考虑被匹配对象的情况。

例：考虑写一个函数，求出在一个 Python 程序里出现的所有整数之和（注意，Python 把一元的正负号看作运算符，不是整数的一部分）。下面是一个定义：

```
def sumInt(fname):
    re_int = r'\b(0|[1-9]\d*)\b'
    inf = open(fname)
    if inf == None:
        return 0
    int_list = map(int, re.findall(re_int, inf.read()))
    # 也可以修改，改用分行读入的方式
    s = 0
    for n in int_list:
        s += n
    return s
```

转义元字符 \B 是 \b 的补，它也是匹配空串，但要求在相应位置是字母或者数字。

实例：re.findall('py.\B', 'python, py2, py342, py1py2py4') 将得到 ['pyt', 'py3', 'py1', 'py2']。

匹配对象（match 对象）

许多匹配函数在匹配成功时返回一个 match 对象，其中记录了所完成的匹配中得到的信息，可以根据需要取用。现在介绍这方面的情况。

首先，调用匹配操作的结果可以用于逻辑判断，匹配成功时得到的 match 对象总表示逻辑真，不成功得到的 None 自然表示假。例如可以写：

```
match1 = re.search(pt, text)
if match1:
    ... match1 ... text ... # 使用match对象的处理操作
```

此外，这种 match 对象提供了一组方法，用于检查和使用与匹配有关的信息。下面介绍一些基本用法，更多信息（包括可选参数）见 re 包文档。在下面的介绍中，mat 总表示通过匹配得到的一个 match 对象。

- 取得被匹配的子串：mat.group()

通过匹配成功得到了 mat，所用的正则表达式自然与目标串里的一个子串成功匹配。通过操作 mat.group() 就能得到这个子串。

- 在目标串里的匹配位置：mat.start()

取得 mat 代表的成功匹配在目标串里的实际匹配位置，这是目标串的一个字符位置，即是被匹配子串开始字符的下标。

- 目标串里被匹配子串的结束位置：mat.end()

结束位置采用 Python 的常规表示方式。假设 text 是得到 mat 的函数调用中作为匹配目标的那个字符串。下面关系成立：

```
mat.group() == text[mat.start():mat.end()]
```

也就是说，被匹配串等于由 start() 和 end() 确定的切片。

- 目标串里被匹配的区间：mat.span()

得到匹配的开始和结束位置形成的二元组，也就是说

```
mat.span() == mat.start(), mat.end()
```

- 其他：mat.re 和 mat.string（这两个表达式是数据域访问，不是函数）

分别取得这个 match 对象所做匹配的正则表达式对象和目标串。应用实例见后。

模式里的组（group）

在正则表达式描述中的另一个重要概念是组（group）。用圆括号括起的模式段 (...) 也是一个模式，在考虑匹配时，它与被括起的子模式匹配的串匹配。还需说明一点，在此同时圆括号还确定了一个被匹配的“组”（字符段）。

在一次成功匹配中，模式串里的各个组也都成功匹配，与它们匹配的那一组字符串将从 1 开始编号，而后可以通过方法调用 mat.group(n) 获取，这里的 mat 是通过匹配操作得到的 match 对象。作为特殊情况，组 0 就是与整个模式匹配的字符串，也可以简单地用 mat.group() 获取（可认为 0 是参数的默认值）。

模式里各对圆括号确定的组按开括号的顺序编号，例如，执行语句

```
mat = re.search('.((.)e)f', 'abcdef')
```

后，表达式 mat.group() 的值是 'cdef'，mat.group(1) 的值是 'de'，而 mat.group(2) 的值是 'd'。

此外，mat.groups() 将得到一个序对，其中包含从编号 1 开始的各个组匹配的串。例如，对上面匹配，mat.groups() 将得到 ('de', 'd')。

组还有一个重要用途，就是在匹配中应用前面的成功匹配，建立前后的部分匹配之间的约束关系。由成功的组匹配确定的字符串，可以在模式串的后面部分用 \n 的形式“引用”，表示要求在该位置匹配同一子串。这里的 n 是一个表示组序号的整数。

例如: `r'(\.{2}) \1'` 可以匹配字符串 'ok ok' 或者 'no no'，但是不能匹配 'no oh'。在模式中的组成功匹配 'no' 之后，模式串要求匹配一个空格，而后匹配与前面的组匹配的同样子串 'no'，但这时遇到的是 'oh'，自然无法匹配。

注意，模式串里的组编号引用应该写 `\1`、`\2` 等。但是，在普通字符串里，`\1` 表示二进制编码为 1 的那个（特殊）字符（通常这个字符被写成 `0x01`，但写 `\1` 也对）。而现在需要在模式串里出现 `\1`、`\2` 等。为解决这个问题，上面用原始字符串形式简化写法。同样模式串的另一种写法是 `'(\.{2}) \\1'`。

Python 的 `re` 包还提供了另一些扩充表示，还有一些细节，这里不再讨论。读者可以参看 Python 文档的有关部分。

其他匹配操作

`re` 包里还定义了另外一些操作，包括：

- `re.fullmatch(pattern, string, flags=0)`。如果整个 `string` 与 `pattern` 匹配成功，就返回记录匹配成功信息的 `match` 对象，不成功时返回 `None`。
- `re.finditer(pattern, string, flags=0)`。这个方法的功能与 `findall` 类似，但它不是返回一个表，而是返回一个迭代器。可以像其他迭代器一样使用。例如用在 `for` 语句头部，顺序取得各非重叠匹配得到的 `match` 对象。
- `re.sub(pattern, repl, string, count=0, flags=0)`。本方法生成替换结果的串，其中 `string` 里与 `pattern` 匹配的各非重叠子串顺序用另一参数 `repl` 代换。如果 `repl` 是字符串就直接代换。也允许 `repl` 是一个以 `match` 对象为参数的函数，这种情况用该函数对匹配得到的 `match` 对象调用的返回值代换被匹配的子串。

例：如果希望把字符串 `text`（例如一个 Python 程序）里所有的 `\t` 都代换为 4 个空格，可以简单地写成 `re.sub('\\t', ' ', text)`。

程序包 `re` 还定义了另外的操作，有关情况见 Python 文档中有关 `re` 的介绍。

正则表达式对象

前面说过，`re.compile(pattern)` 生成一个正则表达式对象，这种对象可以在匹配中反复使用。实际上，正则表达式对象本身支持另一组方法，与直接用 `re.method` 形式调用前面介绍的匹配函数相比，正则表达式对象的方法功能更强，使用也更灵活。

下面介绍中的 `regex` 代表一个正则表达式对象，方括号括起的部分可选：

- 检索：`regex.search(string[, pos[, endpos]])` 检索给定的目标串 `string`。可以指定检索的开始和结束位置。按 Python 惯例，两个位置确定了一个左闭右开的区间。默认情况是从头到尾检索 `string`；如果只给 `pos` 就做到串结束。
- 匹配：`regex.match(string[, pos[, endpos]])` 检查给定的串 `string` 是否有与 `regex` 匹配的前缀。可用 `pos` 指定开始匹配前缀的位置，用 `endpos` 给定被匹配段的终点。
- 完全匹配：`regex.fullmatch(string[, pos[, endpos]])` 检查 `string` 里由指定范围构成的子串是否与 `regex` 匹配，默认范围是整个串。

下面两个方法与 `re` 的同名操作功能类似，但可以指明匹配区间：

- `regex.findall(string[, pos[, endpos]])`
- `regex.finditer(string[, pos[, endpos]])`

下面两个操作与 `re` 的同名操作功能相同：

- 切分: `regex.split(string, maxsplit=0)`
- 替换: `regex.sub(repl, string, count=0)`

另外, 表达式 `regex.pattern` 取得生成 `regex` 的那个模式串。

4.5.5 正则表达式的使用

本小节介绍正则表达式的一些简单使用方式, 供读者参考。

在一些情况中, 目标串里可能存在一些 (可能很多) 与所用正则表达式匹配的子串, 需要逐个处理。这种情况下, 采用匹配迭代器的方式最方便。编程模式是:

```
rel = re.compile(pattern)          # 这里写实际的模式串
for mat in rel.finditer(text) :    # text是被匹配的目标串
    ... mat.group() ...           # 取得被匹配的子串, 做所需操作
    ... text[mat.start()] ... text[mat.end()] ...
```

注意: 操作 `mat.group()`、`mat.start()` 和 `mat.end()` 都只能访问被匹配串的内容, 所做操作不能 (也不会) 修改目标串。如果需要基于正则表达式做字符串的匹配和替换 (生成替换后的串), 首先应该考虑能不能用正则表达式的 `sub` 方法。如果能直接写出准备代入的新串, 就用这个新串作为 `sub` 方法中对应 `repl` 的实参。如果需要代入的新串与被匹配的子串有关, 可以按某种规则从被匹配的串构造出来, 就应该定义一个函数来生成新串, 以这个函数作为 `sub` 方法的 `repl` 参数。

处理更复杂的匹配情况时, 可能需要逐一确定匹配成功的位置, 然后完成所需操作。每次匹配可能使用不同的模式。这种循环自然应该用 `while` 描述: 用一个记录位置的变量 `pos` 存储匹配的起始位置, 在每次循环迭代中正确更新 `pos` 的值。

本章总结

本章介绍了字符串数据结构及其操作。虽然字符串可以看作字符的线性表, 但是字符串操作有其特殊之处。典型的字符串都是以字符串 (不是字符串的基本字符元素) 作为操作对象, 有些操作的结果也是字符串, 重要操作包括构造、拼接、子串替换等。

字符串匹配是许多串操作的基础, 由此受到广泛重视, 人们提出了很多字符串匹配算法。朴素的匹配算法很容易理解, 也容易实现, 但效率低。KMP 匹配算法首先分析模式串, 记录一套移位信息, 而后利用记录的信息实现无回溯匹配。其主要思想就是尽可能利用好已经做过的字符比较的结果, 从而达到 $O(n)$ 的高效率 (n 为目标串长度)。人们提出的其他字符串匹配算法也包含了有趣的思想, 值得关注。

字符串匹配的推广是基于正则表达式的串匹配。这种匹配有坚实的理论基础, 在实践中应用广泛。本章介绍了正则表达式的概念和匹配问题, 用一个简化的正则表达式模型帮助读者理解匹配中的一些问题。最后介绍了 Python 语言的正则表达式功能。其他语言或库的正则表达式定义都与 Python 类似, 可供参考。

模式匹配问题还有许多可能的扩展。例如:

- 近似匹配: 一些串中数据是通过测量得到的, 原本就不准确。另一些情况下原本就不需要准确匹配, 可以有少量失误情况。具体的近似判断可以根据应用的需要定义, 例如, 定义两个串的接近程度、定义一种“距离”等。
- 还有其他更复杂情况中的模式匹配问题。例如, 字符串是一种一维描述, 可以考量二维或者高维描述中的模式匹配, 等等。

练习

一般练习

1. 复习下面概念：字符，字符集，字符串（串），ASCII，Unicode，字符序，字符串长度，空串，字符的位置（下标），字符串相等，字典序，拼接，子串，子串的出现位置，前缀和后缀，串的幂，串替换，子串检索（子串匹配），Python 的 str 类型，字符串匹配，模式匹配，目标串，模式串，朴素匹配算法，无回溯串匹配算法（KMP 算法），模式，模式语言，描述能力与匹配算法的复杂性，通配符，正则表达式，正则表达式匹配，Python 标准库 re 包，Python 原始字符串，元字符，常规字符，顺序组合（拼接），字符组，重复模式，选择模式，组概念。
 2. 字符串 abcdab 有多少个不同子串？请列出它的所有前缀和后缀（子串）。
 3. 找出模式串 acba 在目标串 abccacbacbacabcabbacbbbabcbacba 中的所有出现。
 4. 请用 Python 正则表达式描述下面模式：
 - a) 你所在学校的学号；
 - b) 你所在学校的职工号；
 - c) 你所在城市的带区位号的固话号码；
 - d) 身份证号码；
 - e) 互联网的 IP 地址；
 - f) 图书馆中计算机书籍的馆藏目录图书编号的集合。
- 注意：一般而言正则表达式描述的字符串集合是实际集合的超集，对于复杂的问题，通常无法给出相应字符串集合的精确描述，只需描述一个适当的超集。
5. 请利用 Python 的正则表达式功能实现下面操作：
 - a) 选出一个浮点数数据文件中所有采用科学记数法表示的数据；
 - b) 找出网页（html 文件）里的所有链接；
 - c) 找出一个 Python 文件中定义的所有全局函数名字。

编程练习

1. 针对 Python 的 str 对象，自己实现一个 replace 操作函数。
2. 定义生成器函数 tokens(string, seps)，其中 string 参数是被处理的字符串，seps 是描述分隔字符的字符串，都是 str 类型的对象。该生成器逐个给出 string 里一个个不包含 seps 中分隔字符的最大子串。
3. 请基于链接表的概念定义一个字符串类，每个链接结点保存一个字符。实现其构造函数（以 Python 的 str 对象为参数）。请定义下面方法：求串长度，完成字符串替换，采用朴素方式和 KMP 算法实现的子串匹配。
4. 为上述链接表字符串类增加下面方法：
 - a) find_in(self, another)，确定本字符串中第一个属于字符串 another 的字符所在结点的位置，返回表示这个位置的整数；
 - b) find_not_in(self, another)，与上面函数类似，但它要查找的是不属于 another 的字符；

- c) `remove(self, another)`, 从 `self` 里删除串 `another` 里的字符。
5. 实际中经常需要在一个长字符串里查找与某几个字符串之一匹配的子串。请考虑这一问题并设计一个合理的算法，实现这个算法并分析其复杂性。
6. 请参考 4.3.3 节最后的建议，定义一个模式类，其构造函数基于一个字符串参数生成对象内部的 `pnext` 数组。修改 KMP 算法，使之使用这种模式对象。
7. 考虑一种字符串匹配方法：如果当前字符匹配成功则继续考虑下一字符，如果失败就将模式串右移一个位置继续匹配。经过连续的一些次成功匹配到达了模式串右端后，重新从模式串左端开始补足必要的字符匹配，直至确定一次完整的模式串匹配。在这种匹配中任何时候失败，总按上面的方式，继续用匹配失败的那个模式串字符与目标串的下一个字符比较。请开发这种想法，实现一个字符串匹配函数。
8. R. S. Boyer 和 J. S. Moore 提出了另一种串匹配算法，采用自右向左比较模式串字符的匹配方式，其中也用了一个失败匹配移动表。请基于这一想法深入分析，考虑如何实现一种字符串匹配算法（请自己查找相关材料）。
9. 在互联网 Wiki 中有关字符串匹配的网页里描述了一些字符串匹配算法，请从中选出一种或几种其他算法，定义 Python 函数实现该算法。
10. 对 4.4.2 节的简化正则表达式做一点扩充：增加元字符（+）表示前一字符的 1 次或任意多次重复，“?” 表示前一字符可选出现。请实现相应的匹配函数。

第5章 栈和队列

在常用的数据结构中，有一批结构被称为容器。一个容器结构里总包含一组其他类型的数据对象，称为其元素，支持对这些元素的存储、管理和使用。一类容器具有相同性质，支持同一组操作，可以被定义为一个抽象数据类型。作为容器数据结构，它们都保证存入的元素被保存在容器里，尚未明确删除的元素总可以访问，而取出并删除的元素就不再存在于容器中了。第3章介绍的线性表就是一类容器，该类数据对象除了可以保存元素，支持元素访问和删除外，还记录了元素之间的一种顺序关系。

本章主要介绍另外两类最常用的容器，分别称为栈（stack）和队列（queue），它们都是使用最广泛的基本数据结构。本章还将介绍它们的一些重要应用。

5.1 概述

栈和队列都是保存数据元素的容器，这就意味着可以把元素存入其中，或者从中取出元素使用。这两种结构支持的元素访问操作包括查看，即只是取得元素的有关信息；还包括元素弹出，即在取得元素的同时将其从容器中删除。

栈和队列主要用于在计算过程中保存临时数据，这些数据是计算中发现或者产生的，在后面的计算中可能需要使用它们。在计算中这种情况很常见：工作中产生的中间数据暂时不用或者用不完，就有必要把当时不能立刻用掉的数据存起来。如果可能生成的数据项数在编程时就可以确定，问题比较简单，可以设置几个变量作为临时存储。但如果需要存储的数据项数不能事先确定，就必须采用更复杂的机制存储和管理，这样的存储机制称为缓冲存储或者缓存。栈和队列就是使用最多的缓冲存储结构。

5.1.1 栈、队列和数据使用顺序

栈和队列也是最简单的缓存结构，它们只支持数据项的存储和访问，不支持数据项之间的任何关系。因此，这两种结构的操作集合都很小，很简单，其中最重要的就是存入元素和取出元素两个操作。作为数据结构，它们还需要提供几个任何数据结构都需要的操作，如结构的创建、检查空（可能还有检查满）状态等。

在计算中，中间数据对象的生成有早有晚，存在时间上的先后顺序。在后面使用这些元素时，也可能需要考虑它们的生成时间顺序。最典型的两种顺序是：

- 根据数据生成的顺序，较后生成并保存的数据需要先行使用和处理。例如做数学题，遇到推导进行不下去时，人们通常是退回一步去考虑其他可能性。另一个典型的例子是一摞碗，可以把新的碗摞上去，而取一个时，总是取得最后放上去的碗。
- 按先后顺序处理，先生成的数据先处理。现实生活中这样的例子很多，所有的排队服务都属于这种模式。例如到银行办业务需要排队，先到者先得到服务。具体等待方式并不重要，常见的如直接排入一个等待队列，或是拿（顺序）号后等候叫号，每次得到服务的总是此前尚未得到服务的最早来的顾客。

在这两种典型情况下，被访问数据（使用并可能删除）都按默认的特定方式确定，并不需

要以任何方式指定具体元素。这是最常见的两种缓存和使用方式，栈和队列就是支持按这两种顺序使用元素的缓存数据结构。

由于上述使用特征，栈和队列的实现结构只需要保证元素存入和取出的顺序，并不需要记录或保证新存入的元素与容器中已有元素之间的任何关系。鉴于这两种结构在保证元素的存取时间顺序关系方面的特点，人们也说：

- 栈是保证元素后进先出（后存入者先使用，Last In First Out, LIFO）关系的结构，简称为 LIFO 结构。
- 队列是保证元素先进先出（先存入者先使用，First In First Out, FIFO）关系的结构，简称为 FIFO 结构。

对于一个栈或队列，在任何时候，下一次访问或 / 和删除的元素都默认地唯一确定了。只有新的存入或删除（弹出）操作才会改变下一次默认访问的元素。从元素操作的层面看，栈和队列的性质完全是抽象和逻辑的，对于如何实现这种关系（如何落实存取之间的时间顺序关系）没有任何约束，任何能满足要求的技术均可使用。

另一方面，从实现的角度，应该考虑最简单而且自然的技术。由于计算机存储器的特点，要实现栈或队列，最自然的技术就是用元素存储的顺序表示它们的时间顺序，这也就是说，应该用线性表作为栈和队列的实现结构。

举例说：如果将元素进入一个存储结构（栈或队列）实现为线性表的后端插入，那么，当时还在结构里的存储时间最久的元素，总是表中最前面的那个元素。如果作为队列，下次访问或删除的就应该是这个元素。而结构里最新的元素总是表中最后的那个元素，如果作为栈，下次访问或删除的就应该是这个元素。

5.1.2 应用环境

栈和队列是计算中使用最广泛的缓存结构，其使用环境可以总结如下：

- 计算过程分为一些顺序进行的步骤（任何复杂一点的计算都是这样）。
- 计算中执行的某些步骤会不断产生一些后面可能需要的中间数据。
- 产生的数据中有些不能立即使用，但又需要在将来使用。
- 需要保存的数据的项数不能事先（在编程序的时候）确定。

在这种情况下，通常就需要用一个栈或者队列作为缓存结构。

不难看到，上面的问题在实际中普遍存在，因此计算机软件里通常都有许多栈或队列。栈和队列也是许多重要算法的基础，它们的性质和操作效率对许多重要算法的效率有决定性影响。后面章节里可以看到许多这方面实例。

由于栈和队列在计算机应用中的重要性，Python 的基本功能中已经包含了对栈的支持，可直接用 `list` 实现栈的功能。此外，Python 标准库还提供了一种支持队列用途的结构 `deque`，有关情况将在后面简单介绍。

本章将分别讨论这两种结构的情况，包括它们的性质和模型、实现和问题，以及一些应用实例。还将讨论一些一般性问题。

5.2 栈：概念和实现

栈（stack，在一些书籍里称为堆栈）是一种容器，可存入数据元素、访问元素、删除元素等。存入栈中的元素之间相互没有任何具体关系，只有到来的时间先后顺序。在这里没有元素

的位置、元素的前后顺序等概念。

栈的基本性质保证，在任何时刻可以访问、删除的元素都是在此之前最后存入的那个元素。因此，栈确定了一种默认元素访问顺序，访问时无需其他信息。

5.2.1 栈抽象数据类型

前面的讨论已经说明，栈的基本操作是一个封闭的集合（与线性表的情况不同）。现在给出一个栈抽象数据类型的描述，其中定义的操作包括：栈的创建（创建一个空栈）、判断栈是否为空、将元素压入栈中（也称进栈或入栈）、从栈中弹出元素并将其返回（也称为退栈或者出栈），以及检查栈元素（访问最后入栈元素）。

最后两个操作都遵循栈的 LIFO 原则，被访问或 / 和删除的元素按 LIFO 规则唯一确定。很显然，如果在做这两个操作的时候该栈为空，操作无定义。在具体实现时需要考虑和适当处理这个问题。下面是栈的抽象数据类型描述：

ADT Stack:	
Stack(self)	# 创建空栈
is_empty(self)	# 判断栈是否为空，空时返回True否则返回False
push(self, elem)	# 将元素elem加入栈，也常称为压入或推入
pop(self)	# 删除栈里最后压入的元素并将其返回，常称为弹出
top(self)	# 取得栈里最后压入的元素，不删除

虽然栈的基本操作是很明确的集合，但其具体设计还是可以有些变化。例如，如果实现结构容纳元素的能力是固定的，就需要增加一个判栈满的函数。也有些栈结构设计中 pop 操作只删除元素但不返回。这些只是小变化，不影响其基本结构。

栈的线性表实现

如前所述，线性表可以看作栈的一种最自然的实现方式，栈可以实现为（可以看作）在一端进行插入和删除的线性表。因此，许多教科书直接把栈称为后进先出表（LIFO 表），或者下推表。在表实现中，执行插入和删除操作的一端被称为栈顶，另一端称为栈底。访问和弹出的都应该是栈顶元素，有关的情况如图 5.1 所示。

用线性表的技术实现栈时，操作只在表的一端进行，不涉及另一端，更不涉及表的中间部分。由于这种情况，自然应该选择实现最方便而且保证两个主要操作效率最高的那一端作为栈顶。根据前面有关线性表的讨论，易见下面情况：

- 对于顺序表，后端插入和删除是 O(1) 操作，应该用这一端作为栈顶（采用顺序表实现，图 5.1 里的表顺序应该反过来，用表尾作栈顶）。
 - 对于链接表，前端插入和删除都是 O(1) 操作，应该用这端作为栈顶。
- 在实际中，栈都采用这两种技术实现。

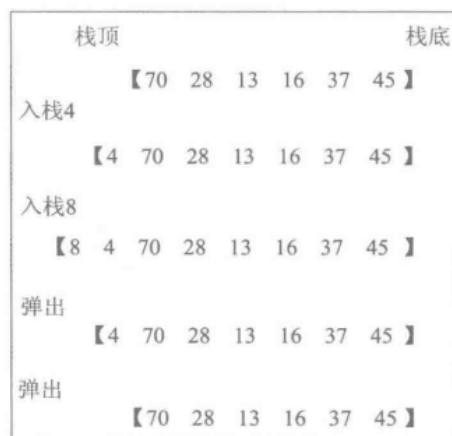


图 5.1 线性表作为栈

5.2.2 栈的顺序表实现

实现栈结构之前，先考虑为操作失败的处理定义一个异常类。Python 程序里的自定义异常

都应定义为 `Exception` 的派生类。编程时通过继承已有的异常类定义自己的异常类。由于操作时栈不满足需要（如空栈弹出）可以看作参数值错误，采用下面定义：

```
class StackUnderflow(ValueError): # 栈下溢（空栈访问）
    pass
```

这里把异常 `StackUnderflow` 定义为 `ValueError` 的子类，只简单定义了一个类名，类体部分只有一个 `pass` 语句，未定义任何新属性，因为不准备提供 `ValueError` 之外的新功能，只是想与其他 `ValueError` 异常有所区分，程序出错时能产生不同的错误信息。必要时可以定义专门的异常处理操作。自定义异常和 Python 内置异常类似，同样通过 `except` 捕捉和处理，但（显然）只能通过 `raise` 语句引发。

采用顺序表技术实现栈，首先会遇到实现顺序表时提出的各种问题，如：是采用简单顺序表实现，还是采用动态顺序表（分离式实现的顺序表）？如果采用简单的顺序表技术，就可能出现栈满的情况，继续压入元素就会溢出，应该检查和处理。而如果采用动态顺序表，栈的存储区满时可以置换一个更大的存储区。这种情况下又会出现存储区的置换策略问题，以及分期付款式的 $O(1)$ 时间复杂度问题。

Python 的 `list` 及其操作实际上提供了与栈的使用方式有关的功能，可以直接作为栈来使用。相关情况可以如下考虑（假定 `lst` 的值是一个表）：

- 建立空栈对应于创建一个空表 `[]`，判空栈对应于检查是否为空表。
- 由于 `list` 采用动态顺序表技术（分离式实现），作为栈的表不会满。
- 压入元素操作应在表的尾端进行，对应于 `lst.append(x)`。
- 访问栈顶元素应该用 `lst[-1]`。
- 弹出操作也应该在表尾端进行，无参的 `lst.pop()` 默认弹出表尾元素。

由于 `list` 类型采用的是动态顺序表技术（第3章已经讨论），压栈操作具有分期付款式的 $O(1)$ 时间复杂度，其他操作都是 $O(1)$ 时间操作。

把 `list` 当作栈使用，完全可以满足应用的需要。但是，这样建立的对象还是 `list`，提供了 `list` 类型的所有操作，特别是提供了一大批栈结构原本不应该支持的操作，威胁栈的使用安全性（例如，栈要求未经弹出的元素应该存在，但表允许任意删除）。另外，这样的“栈”不是一个独立类型，因此没有独立类型的所有重要性质。

为了概念更清晰，实现安全，操作名也更易理解，可以考虑基于顺序表定义一个栈类，使之成为一个独立的类型，成为前面抽象数据类型的一个合格实现。把 Python 的 `list` 隐藏在这个类的内部，作为其实现基础（对应于前面说的用表实现栈）。

下面是一个栈类的定义，其中用一个 `list` 类型的数据属性 `_elems` 作为栈元素存储区，把 `_elems` 的首端作为栈底，尾端作为栈顶：

```
class SStack():
    # 基于顺序表技术实现的栈类
    def __init__(self):
        # 用list对象 _elems 存储栈中元素
        self._elems = [] # 所有栈操作都映射到list操作

    def is_empty(self):
        return self._elems == []

    def top(self):
        if self._elems == []:
            raise StackUnderflow("in SStack.top()")
        return self._elems[-1]
```

```
def push(self, elem):
    self._elems.append(elem)

def pop(self):
    if self._elems == []:
        raise StackUnderflow("in SStack.pop()")
    return self._elems.pop()
```

所有操作的实现方式都直接映射到对应的表操作。如前所述，初始创建对象时建立一个空栈；`top` 和 `pop` 操作都需要先检查栈的情况，在栈空时引发异常。属性 `_elems` 是只在内部使用的属性，这里采用下划线开头的名字。

另请注意这里的两个 `raise` 语句，它们都包含了一个字符串实参，执行中实际产生的异常对象将携带这个信息。在异常的引发处，可以用这种实参向最终捕捉到这个异常的处理器传递一些信息。在这里给出的简单字符串将在 Python 解释器报错时显示，标明异常发生的位置，用于帮助检查程序错误。在更复杂的情况里，可以通过异常的这种参数传递有用的状态信息，供异常处理器使用。

有了上面定义，下面是一段使用这个类的代码：

```
st1 = SStack()
st1.push(3)
st1.push(5)
while not st1.is_empty():
    print(st1.pop())
```

5.2.3 栈的链接表实现

本小节考虑如何基于链接表技术实现一种栈类。链接栈的基础结构就是链接表，可以直接借用第 3 章定义的 `LNode` 类作为链中的结点。

下面的讨论既是为了完整，也有实际意义。由于可以自动扩大存储区，前面基于顺序表的栈类不会出现栈满的情况，因此应该能满足绝大部分的实际需要。在这种情况下，为什么还需要考虑基于链接的栈实现？对这个问题的回答主要考虑顺序表的两个特点：扩大存储需要做一次高代价操作；另外，顺序表需要完整的大块存储区。采用链接技术，在这两个问题上都有优势。链接实现的缺点是更多依赖于解释器的存储管理，每个结点的链接开销，以及链接结点在实际计算机内存中任意散布可能带来的操作开销。

前面说过，由于所有栈操作都在线性表的一端进行，采用链接表技术，自然应该用表头一端作为栈顶，表尾作为栈底，使操作实现方便，效率也高。按照这种安排，很容易定义出一个链接栈类。这里是一个定义：

```
class LStack():    # 基于链接表技术实现的栈类，用LNode作为结点
    def __init__(self):
        self._top = None

    def is_empty(self):
        return self._top is None

    def top(self):
        if self._top is None:
            raise StackUnderflow("in LStack.top()")
        return self._top.elem
```

```

def push(self, elem):
    self._top = LNode(elem, self._top)

def pop(self):
    if self._top is None:
        raise StackUnderflow("in LStack.pop()")
    p = self._top
    self._top = p.next
    return p.elem

```

这个类的使用方式与 SStack 完全一样。从使用的角度看，这两个类除了类名不同外，完全可以相互替代。这也是抽象数据类型的功劳。

5.3 栈的应用

栈是最简单的数据结构，其实现直截了当，不需要过多讨论。栈的应用非常广泛，本节介绍若干简单的应用，供读者参考，也希望能举一反三。

栈是算法和程序里最常用的辅助结构，基本用途基于两个方面：

- 如前所述，使用栈可以很方便地保存和取用信息，因此它常被作为算法或程序里的辅助存储结构，临时保存信息，供后面操作中使用。
- 栈具有后进先出的性质，利用这种性质可以保证特定的存取顺序。对于许多实际应用，这种后进先出性质非常重要。

实际上，许多实际应用利用了这两方面的特性。

作为最简单的应用实例，栈可以用于颠倒一组元素的顺序。只需把所有元素按原来的顺序全部入栈，再顺序取出，就能得到反序后的序列。假设 list1 存储需要倒序的元素系列，执行下面代码段：

```

st1 = SStack()
for x in list1:
    st1.push(x)
list2 = []
while not st1.is_empty():
    list2.append(st1.pop())

```

现在 list2 里存储的是 list1 中序列的反序序列，整个操作的代价为 $O(n)$ 。

如果允许入栈和出栈操作任意交错，通过不同的入栈出栈操作序列，可以得到不同的元素序列。但请注意，这种做法并不能得到原序列的任意排列，结果序列有一定的规律性。请读者自己分析和研究。

下面介绍栈的若干典型应用，有些是非常具体的特殊应用，也有些是一大类应用的具体实例，更具典型性。

5.3.1 简单应用：括号匹配问题

在许多正文中都有括号，特别是在表示程序、数学表达式的正文片段里，括号有正确配对问题。作为例子，下面考虑 Python 程序里的括号，在这里可以看到：

- 存在多种不同的括号，下面只考虑其中三种：圆括号、方括号和花括号。
- 每种括号都包括一个开括号和一个闭括号，相互对应。括号括起的片段可能嵌套，各种括号应该正确地嵌套并分别配对。

不难总结出检查括号配对的原则：在扫描正文过程中，遇到的闭括号应该与此前最近遇到且尚未获得匹配的开括号配对。如果最近的未匹配开括号与当前闭括号不配对，或者找不到这样的开括号，就是匹配失败，说明这段正文里的括号不配对。

由于存在多种不同的括号对，每种括号都可能出现任意多次，而且还可能嵌套。为了检查是否匹配，扫描中必须保存遇到的开括号。由于写程序时无法预知要处理的正文里会有多少括号需要保存，因此不能用固定数目的变量保存，必须用缓存结构。

由于括号的出现可能嵌套，需要逐对匹配：当前闭括号应该与前面最近的尚未配对的开括号匹配，下一个闭括号应与前面次近的括号匹配。这说明，需要存储的开括号的使用原则是后存入者先使用，符合 LIFO 原则。

进而，如果一个开括号已配对，就应该删除这个括号，为随后的匹配做好准备。显然，在扫描中，后遇到并保存的开括号将先配对并被删除，这就是按出现的顺序后进先出。这些情况说明，用栈保存遇到的开括号可以正确支持匹配工作。

通过上面分析，处理这个问题的脉络已经很清楚了：

- 顺序扫描被检查正文（一个字符串）里的一个个字符。
- 检查中跳过无关字符（所有非括号字符都与当前处理无关）。
- 遇到开括号时将其压入栈。
- 遇到闭括号时弹出当时的栈顶元素与之匹配。
- 如果匹配成功则继续，发现不匹配时检查以失败结束。

现在考虑定义函数完成这个检查。在这里需要区分哪些字符是括号、哪些是开括号，还需要知道括号之间的配对关系。先定义几个变量记录检查中有用的数据：

```
parens = "()[]{}"
open_parens = "([{"
opposite = {"}": "(", "]": "[", ")": "("} # 表示配对关系的字典
```

将变量 parens 初始化为包含所有括号字符的字符串，用于检查扫描中遇到的字符是否为括号；open_parens 记录所有开括号字符；opposite 是个字典，从任一闭括号找到与之对应的开括号，用于确定匹配关系。

有了上面这些数据准备，检查函数的定义已经很简单了：

```
def check_parens(text):
    """括号配对检查函数，text是被检查的正文串"""
    parens = "()[]{}"
    open_parens = "([{"
    opposite = {"}": "(", "]": "[", ")": "("} # 表示配对关系的字典

    def parentheses(text):
        """括号生成器，每次调用返回text里的下一括号及其位置"""
        i, text_len = 0, len(text)
        while True:
            while i < text_len and text[i] not in parens:
                i += 1
            if i >= text_len:
                return
            yield text[i], i
            i += 1

    st = SStack() # 保存括号的栈
```

```

for pr, i in parentheses(text):
    if pr in open_parens:           # 对text里各括号和位置迭代
        st.push(pr)                # 开括号，压进栈并继续
    elif st.pop() != opposite[pr]:   # 不匹配就是失败，退出
        print("Unmatching is found at", i, "for", pr)
        return False
    # else:这是一次括号配对成功，什么也不做，继续

print("All parentheses are correctly matched.")
return True

```

函数的主体部分很简单，先建立一个栈 `st`，然后逐个处理遇到的括号（由独立的括号生成器函数 `parentheses` 获得）：开括号直接进栈，闭括号与栈顶括号匹配。如果匹配就继续，发现不匹配时输出一条信息后结束。注意循环体里的 `if` 语句，它没有最后的 `else` 部分，因为在相应情况下什么也不需要做。函数里在这个位置写了一个注释，明确说明这里的问题已经考虑了，也是为了避免误解。对于此类情况，人们有时也写一个以 `pass` 作为体的 `else` 分支，明示在这里不需要做任何动作。

这里的重要工作片段由局部函数 `parentheses` 完成，这是一个扫描正文的生成器（generator）函数，可称为括号生成器，它逐个返回找到的一个个括号及其在串中的位置。取得位置信息是为了生成更好的出错信息。

回忆一下，生成器是 Python 中一种用函数形式定义的迭代器，其中的 `yield` 语句产生结果，可以用在需要迭代器的地方。每次执行到 `yield` 产生出下一个值，函数结束（执行 `return` 语句或执行完整个函数体）导致迭代结束。

局部函数 `parentheses` 在执行中顺序扫描参数 `text` 里的字符，直至检查完所有字符时结束。如果在扫描中遇到括号，就生成出这个字符和它在 `text` 里的位置，交给主程序处理。与函数的返回值一样，Python 生成器也可以一次生成一组值，主函数代码里的 `for` 循环里用两个变量接收生成器产生的值。

函数 `check_parens` 的使用非常简单，读者很容易确认它确实能完成工作。

定义完这样一个函数之后，现在做一点总结：

- 1) 函数里利用了栈的后进先出性质，这是完成工作的关键。
- 2) 函数定义中首先准备好了所需的数据。虽然在整个程序里这几个变量都只在一个地方用，但把它们独立出来，使程序更清晰易读，也易于修改，值得提倡。
- 3) 本函数的定义中充分利用了 Python 丰富的数据结构和操作，如 `in` 和 `not in` 等关系运算符和字典结构等。
- 4) 把检查正文并找出括号的工作定义为一个独立的生成器函数，使整个函数的代码有了很好的功能划分，意义清晰，也更便于修改维护。

当然，上面工作中定义的只是一个能初步解决问题的函数，要把它应用到实际环境里，还需要做些工作。例如，如果真要检查实际 Python 程序里的括号配对问题，在正文扫描中就需要跳过注释和字符串等。由于上面函数设计有了很好的功能划分，要处理 Python 程序中的括号，只需要修改其中的括号生成器函数，主函数中完成括号匹配部分完全不需要修改。这一工作留给读者自己练习。

5.3.2 表达式的表示、计算和变换

算术表达式是人一生中最早接触的形式化描述，人们从小学时代起就开始写各种数学表达

式，先是写算术表达式，后来写代数表达式等。

表达式和计算的描述

数学表达式中最重要的构造符号是一组二元运算符。在最常见表达形式中，二元运算符写在其两个运算对象中间，这种写法称为中缀表示形式，按照中缀表示写出的表达式称为中缀表达式。中缀表示是一般人最早开始使用，也是最习惯的表达式形式。

但是实际上，作为表达式的表示形式，中缀表示很难统一地贯彻。首先是一元和多元运算符都难以中缀形式表示。在代数表达式里，函数可以有任意多个运算对象（函数的参数），通常把函数符号（函数名）写在运算对象前面，这种写法称为前缀表示。为了界定函数的参数范围，通常用括号将它们括起。一元运算符可能写在运算对象前面（如正负号）或后面（如阶乘运算符）。可见，表达式的习惯表示是前缀表示和中缀表示等的结合。

假设每个运算符的元数（运算对象个数）确定且唯一[⊖]。在描述表达式时，最重要问题是准确描述计算的顺序。中缀表达式的主要缺点就在这里：它不足以表示所有可能的运算顺序，需要通过辅助符号、约定和 / 或辅助描述机制。

首先，必须为中缀表达式引进括号的概念，规定括号里的运算先做（这是一种显式描述计算顺序的机制），以便于人们直接说明一些计算的特定顺序。写出所需的所有括号，有时会非常麻烦。为了缓解这种麻烦，人们又引进了优先级的概念，给各种运算符规定了不同的优先级（或优先级关系，如先乘除后加减），规定优先级高的运算符结合性强，多个运算符相继出现时，优先级高的运算先做。有了这些还不够，还要规定具有相同优先级的运算符相继出现时的计算顺序（运算符的结合性）。

实际上，（数学）表达式并不一定采用这种习惯写法，例如，可以采用纯粹的前缀表示，这样写出的表达式称为前缀表达式。还有一种写法称为后缀表示，其中所有运算符（包括函数）都写在它们的运算对象之后，这样写出的表达式称为后缀表达式。实际上，后缀表达式特别适合计算机处理。前缀表达式也被称为波兰表达式，由波兰数学家 J. Lukasiewicz 于 1929 年提出；后缀表达式也称为逆波兰表达式。

现在考虑计算顺序的描述情况。有趣的是，在前述基本假设（所有运算符的元数均已知且唯一）下，前缀表达形式和后缀表达形式都不需要引进括号，也不需要任何有关优先级或结合性的规定，已自然地描述任意复杂的表达式的计算顺序：对于前缀表示，每个运算符的运算对象，就是它后面出现的几个完整表达式，表达式个数由运算符的元数确定。对于后缀表示，情况类似但位置相反。

由此可见，中缀表达式的表达能力最弱，而给中缀表达式增加了括号后，几种表达方式具有同等表达能力。优先级和结合性的规定只是为了使用方便。

首先对几种不同的表达式形式做一下对比。按照三种表达形式的规定，人们习惯的“3+2”，在前缀形式下应该写成“+3 2”，在后缀形式下应该写成“3 2 +”。下面是同一个算术表达式的三种等价表示形式：

中缀形式：(3 - 5) * (6 + 17 * 4) / 3

前缀形式：/ * - 3 5 + 6 * 17 4 3

后缀形式：3 5 - 6 17 4 * + * 3 /

这三个表达式描述的是同一个计算过程。

[⊖] 实际上这个假设也有点问题。例如+和-通常既表示二元运算符，又表示一元运算符（正号和负号）。但根据上下文可以判断出每个具体出现对应的元数，因此可以区分这两种情况。

下面将首先考虑后缀表达式的求值问题，其中假定要处理的是简单算术表达式，运算对象都是浮点数形式表示的数，只有四个运算符“+”、“-”、“*”、“/”（加减乘除），它们都是二元运算符。而后讨论不同表达式形式之间的转换（这里只考虑中缀形式和后缀形式），研究相应的转换算法。最后讨论中缀表达式的求值问题。

后缀表达式的计算

为便于分析有关计算过程，先假设有一个函数 `nextItem()`，调用它就能得到输入里的下一个项，或是一个运算对象，或是一个运算符。根据后缀表达式的计算规则，计算过程应该是顺序检查表达式里的一个个项，分两种情况：

- 遇到运算对象时，应该记录它以备后面使用。
- 遇到运算符（函数名也一样，下面不考虑）时，应该根据其元数（下面假定都是二元运算符），取得前面最近遇到的几个运算对象或已完成运算的结果（二元运算符都取2个），应用这个运算符（或函数）计算，并保存其结果。

显然，现在又遇到了需要记录信息以备将来使用的问题。那么，应该怎么记录这些信息呢？由于表达式可以任意复杂，不能事先确定需要记录的信息的项数，必须用一个缓存结构。采用什么结构则要根据计算的性质决定。首先分析情况：

- 需要记录的是已经掌握的数据，无论这些数据是直接由外部得到，还是前面计算出来，都需要缓存。因为当时这些中间结果还不能立刻使用。
- 每次处理运算符，应使用的是此前最后记录的几个结果（具体项数根据运算符的元数确定，现在都是2）。

显然，这里的情况是典型的后保存先使用，应该用栈作为缓存结构。

上面的分析其实已经说明了有关算法的基本结构，实际编程就是用Python语言写出这个算法。先考虑算法的框架，不难给出下面描述：

假定`st`是一个栈，算法的核心是下面循环

```
while 还有输入:
    x = nextItem()
    if is_operand(x):      # 是运算对象，转换为浮点数并入栈
        st.push(float(x))
    else:                  # 否则，是(二元)运算符
        a = st.pop()       # 第二个运算对象
        b = st.pop()       # 第一个运算对象
        ... ...             # 根据运算符x对a和b计算
        ... ...             # 计算结果压入栈
```

这里用到几个辅助函数，它们的实现依赖于具体情况，包括：①被求值的表达式从哪里获得；②表达式里的元素如何表示（例如，可以考虑用字符串表示）。

下面假定后缀表达式的求值由一个函数完成，被求值表达式由参数得到，对应实参应是一个字符串，其内容是要求值的后缀表达式，项之间有空格分隔。

为了程序清晰，这里先定义一个包装过程，把字符串分割成项的表：

```
# 定义一个函数把表示表达式的字符串转化为项的表
def suffix_exp_evaluator(line):
    return suf_exp_evaluator(line.split())
```

注意上面的算法框架，如果处理运算符时栈中元素不足两个，操作就应失败。还有，处理完整个表达式时，栈里应该只剩下计算结果。这两个操作都需要检查栈的深度，但前面的栈类

不支持这种操作。为满足程序需要，这里用继承的方式定义一个扩充的栈类，增加一个检查栈深度的方法（不改变栈元素，不会破坏栈的安全性）：

```
class ESStack(SStack):
    def depth(self):
        return len(self._elems)

def suf_exp_evaluator(exp):
    operators = "+-*/"
    st = ESStack() # 扩充功能的栈，可用depth()检查栈元素个数

    for x in exp:
        if x not in operators:
            st.push(float(x)) # 不能转换将自动引发异常
            continue

        if st.depth() < 2: # x必为运算符，栈元素不够时引发异常
            raise SyntaxError("Short of operand(s).")
        a = st.pop() # 取得第二个运算对象
        b = st.pop() # 取得第一个运算对象

        if x == "+":
            c = b + a
        elif x == "-":
            c = b - a
        elif x == "*":
            c = b * a
        elif x == "/": # 这里可能引发异常ZeroDivisionError
            c = b / a
        else:
            break
        # else分支不可能出现，写在这里是为了清晰

        st.push(c)

    if st.depth() == 1:
        return st.pop()
    raise SyntaxError("Extra operand(s).")
```

这里改用 for 循环，对 exp 中每个项迭代一次。非运算符的处理是进栈。一旦确定当前项是运算符，先检查栈中运算对象是否够用，不够时抛出异常。处理完整个表达式时，栈里应该只剩一项结果，否则就是表达式结构有误，也应引发异常。

为方便用户使用，定义一个交互式的驱动函数（主函数）：

```
def suffix_exp_calculator():
    while True:
        try:
            line = input("Suffix Expression: ")
            if line == "end": return
            res = suf_exp_evaluator(line)
            print(res)
        except Exception as ex:
            print("Error:", type(ex), ex.args)
```

这里有几点值得注意：

- 首先，这里假定后缀表达式里不同的项和运算符之间都有空格。在这种情况下，简单地用 `str.split()` 就能得到一个项表。如果没有这种要求，就需要采用更复杂的技术去设法解析函数的输入。这方面细节不是这里讨论的主题，从略。有兴趣的读者可以自己考虑。但无论如何，都需要先严格定义合法表达式的形式。
- 函数里用一个 `try` 语句捕捉用户使用中出现的异常，保证这个计算器不会因为用户输入错误而结束。由于是交互式程序，用户输入出错的可能性时时存在，例如表达式里出现除 0 错，或者表达式形式有误。作为交互式程序，应保证用户在使用中出错时程序不崩溃，而且能合理地继续下去，必要时应该给出一些信息。
- 语句 `except Exception as ex` 有两重意思：`Exception` 是所有系统和用户定义异常的基类，捕捉 `Exception` 就是捕捉所有异常，这保证无论出什么问题，这个计算器都能继续工作。`as ex` 部分要求把捕捉的异常约束于变量 `ex`，使后面的处理器代码能通过 `ex` 使用异常里包含的信息。最后的 `print` 语句先输出串“`Error:`”，然后输出所捕捉异常的类型，以及引发异常时给的实际参数。例如，前面计算器里有“`raise SyntaxError("Extra operand(s).")`”，如果捕捉到这个语句引发的异常，输出信息就会包括“`Extra operand(s).`”。

中缀表达式到后缀表达式的转换

前面说过，在几种表达式形式中，中缀表达式的情况最复杂，其中出现了括号等新情况，因此求值也不太容易处理。前一小节定义了一个后缀表达式计算器，如果能把中缀表达式转换为后缀表达式，就可以借用前面定义的后缀表达式求值器了。

现在考虑如何完成这一转换。为分析这里的情况，重看前面的例子

中缀：`(3 - 5) * (6 + 17 * 4) / 3`

后缀：`3 5 - 6 17 4 * + * 3 /`

对照两个表达式，可以看到转换中需要考虑的一些基本情况：①在扫描中缀表达式的过程中，如果遇到运算对象，就应该直接将其送出，作为后缀表达式的一个项（在后缀表达式里，运算对象出现在对应的运算符之前）；②把运算符放入后缀表达式也就是要求运算，需要仔细控制送出的时机。在考虑运算符在后缀表达式里的出现位置时，必须处理好中缀表达式的优先级和结合性等问题。有下面这些情况：

- 中缀表达式中运算符的处理比较麻烦：扫描中遇到一个运算符时不能将其输出，只有看到下一运算符的优先级不高于本运算符时，才应该（能够）去做本运算符要求的计算，或说这时才应该把本运算符送入后缀表达式。说得更准确些，遇到运算符 o 时，应该把它与前面还没处理的运算符（假设是 o' ）比较。如果 o 的优先级不高于 o' ，就做 o' 要求的计算（也就是输出 o' ）。而后记录 o （无论做没做 o' 的计算）。
- 可以看到，在这个过程里，总要拿当前运算符与前面最近的且尚未送走的运算符比较，而且可能删除前面的运算符，这也是后保存的信息先使用（与前面保存的最后一个运算符比较，而且可能用掉它）。另一方面，处理中需要记录的运算符个数也无法事前确定，因此应该用一个栈保存尚未处理的运算符。
- 另一方面，转换中还需要处理括号问题。表达式里的括号是配对的，左括号标明了一个应该优先计算的子表达式的起点，需要记录。右括号说明应该先计算的子表达式到此为止。因此，在遇到右括号时，需要逐个送出（弹出）栈里的运算符（做它们要求的计算，排在后面的必定具有更高优先级），直至遇到左括号时也将其弹出。

- 此外，在扫描完整个中缀表达式时，栈里可能还剩下一些运算符。它们的计算都应该进行，也就是说，应该把它们一一弹出送到后缀表达式。

上面分析已经囊括了在扫描处理中缀表达式的过程中可能遇到的所有情况，其中都要求后遇到的（运算符）先处理，而且只需要考虑运算符和括号，用一个记录运算符的栈就足够了。在操作中，还需要特别注意检查栈空的情况。

下面考虑转换算法，为使算法清晰，同样先准备一些数据：

```
priority = {"+":1, "-":3, "*":5, "/":5}
infix_operators = "+-* / ()" # 把 '(', ')' 也看作运算符，但特殊处理
```

字典 `priority` 为每个运算符关联一个优先级。这里给开括号 “(” 关联了一个很低的优先级，保证其他运算符都不会将其弹出，只有对应的 “)” 才能弹出它。另外，在考虑表达式中项的分类时，把括号也看作运算符。

基于这些分析，可以给出下面的转换函数定义。这里用一个表记录得到的后缀表达式，最后作为函数的结果返回：

```
def trans_infix_suffix(line):
    st = SStack()
    exp = []

    for x in tokens(line): # tokens是一个待定义的生成器
        if x not in infix_operators: # 运算对象直接送出
            exp.append(x)
        elif st.is_empty() or x == '(': # 左括号进栈
            st.push(x)
        elif x == ')': # 处理右括号的分支
            while not st.is_empty() and st.top() != '(':
                exp.append(st.pop())
            if st.is_empty(): # 没找到左括号，就是不配对
                raise SyntaxError("Missing '(.')")
            st.pop() # 弹出左括号，右括号也不进栈
        else: # 处理算术运算符，运算符都看作左结合
            while (not st.is_empty() and
                   priority[st.top()] >= priority[x]):
                exp.append(st.pop())
            st.push(x) # 算术运算符进栈

    while not st.is_empty(): # 送出栈里剩下的运算符
        if st.top() == '(': # 如果还有左括号，就是不配对
            raise SyntaxError("Extra '(.')")
        exp.append(st.pop())

    return exp
```

函数里的变量 `exp` 用于记录转换得到的后缀表达式，采用项表的形式。函数的实现与前面各种情况的分析对应：遇运算对象直接输出；左括号总进栈；右括号特殊处理；遇到运算符时与栈顶元素比较并根据情况处理，最后将运算符进栈。

为测试方便，定义一个专门用于测试的辅助函数：

```
def test_trans_infix_suffix(s):
    print(s)
    print(trans_infix_suffix(s))
    print("Value:", suf_exp_evaluator(trans_infix_suffix(s)))
```

生成器函数 `tokens` 逐一产生输入表达式里的各个项：

```

def tokens(line):
    """ 生成器函数，逐一生成 line 中的一个个项。项是浮点数或运算符。
    本函数不能处理一元运算符，也不能处理带符号的浮点数。 """
    i, llen = 0, len(line)
    while i < llen:
        while line[i].isspace():
            i += 1
        if i >= llen:
            break
        if line[i] in infix_operators:           # 运算符的情况
            yield line[i]
            i += 1
            continue

        j = i + 1                               # 下面处理运算对象

        while (j < llen and not line[j].isspace() and
               line[j] not in infix_operators):
            if ((line[j] == 'e' or line[j] == 'E')      # 处理负指数
                and j+1 < llen and line[j+1] == '-'):
                j += 1
            j += 1
        yield line[i:j]                         # 生成运算对象子串
        i = j

```

这个函数的设计比较简单，其中对合法的表达式有一些合理假设：输入是一个字符串 line，它表示一个中缀表达式；表达式的各项之间可以有任意多个空格；加、减、乘、除符号作为运算符，它们与前后项之间可以有或没有空格；非空格且非运算符的一段就是一个运算对象，其中如果出现 E 或 e，它后面可以有一个负号表示负的指数部分。函数里并不仔细检查运算对象是否符合 Python 对浮点数格式的要求。

显然，这个计算器的处理功能比较有限，它不能处理负号（和负数），也不能处理一元运算符。为更好地处理算术表达式，首先需要定义“项”的形式和表达式的结构，然后根据它们实现将字符串分解为项的函数（生成器函数）。这一过程称为词法分析和语法分析，有关思想和技术是“编译原理”课程的内容，这里不讨论。也可以只考虑算术表达式，设计一套较简单的处理方式。有关分析和处理留给读者考虑。

中缀表达式的求值

由前面讨论可知，中缀表达式的求值比后缀表达式复杂，但结合前两小节的两组函数可以完成这一求值。请读者定义一个函数，调用前面定义的函数完成这一计算。

现在考虑另一种方式，不经过后缀表达式过渡，实现一个直接求值中缀表达式的函数。在这样做时，需要统一考虑下面几个问题：

- 运算符的优先级。
- 括号的作用。
- 根据扫描表达式过程中的情况，确定完成各个运算的时机。
- 在需要做某个运算时，能找到正确的运算对象。

前一小节研究从算术表达式中缀形式到后缀形式的转换算法，已经解决了上面的前三个问题。转换处理中需要用一个运算符栈，在扫描表达式的过程中比较运算符的优先级，并根据情况压入/弹出它们。在操作中也完成了对运算符优先级和括号的处理。实际上，在该过程生成后缀表达式中运算符的时刻，也就是需要执行运算的时刻（按后缀表达式的性质，计算中遇到

运算符就做相应的计算)。现在考虑上面的最后一个问題。

由于现在要直接计算中缀表达式，扫描中遇到运算对象时不能输出，而应该临时保存起来，在需要用的时候能比较容易地找到它们。根据后缀表达式的计算规则，每当需要执行一个运算时，对应的运算对象应该是前面最近遇到的运算对象，或是最近计算得到的结果。在后缀表达式计算器里用一个栈保存这些数据，现在可以模拟它，另外引进一个数据栈，保存运算对象和计算得到的中间结果。

有了上面提出的这两个栈，就可以把两个过程编织在一起，组合成一个统一的过程。总结这个计算过程，其中存在下面几种情况和相应操作：

- 1) 扫描中遇到运算对象将其压入数据栈(后缀表达式计算器的动作)。
- 2) 遇到运算符和括号时，按照前面转换算法中的方式，分几种情况处理：
 - ①左括号，压入运算符栈。
 - ②右括号，弹出运算符完成计算，直至弹出对应左括号。
 - ③其他运算符，按照前面做法基于优先级处理。
- 3) 确定了应该应用一个运算符时：
 - ①其运算对象(数据)就是数据栈顶的两个项。
 - ②完成计算得到的结果还需要压进数据栈。
- 4) 整个表达式处理完后，逐项弹出运算符栈剩下的运算符并完成计算。

易见，这里的第1和第3两项与后缀表达式计算中的处理方式一致，第2和第4两项与表达式转换算法中的处理方式一致。

函数实现的其他细节可以参考前面两小节里的算法，如其中优先级的处理细节。辅助函数tokens可以直接使用。写出函数定义的工作作为本章的练习。

5.3.3 栈与递归

如果在一个定义(如Python函数)中引用了被定义的对象(被定义的函数)本身，这种定义被称为递归定义。类似的，如果在一种数据结构里的某个或某几个部分具有与整体同样的结构，人们也说这是一种递归结构。

例如，在Python语言里定义一个函数时，允许在函数的定义体里出现对这个函数自身的调用(称为递归调用)。这说明，在该函数需要完成的工作中，有一部分需要通过对自己的调用完成。这是递归定义的典型例子。另一方面，由结点通过链接构成的单链表，是递归结构的一个实例：这样的表或者为空；而在其非空时，去掉第一个结点之后剩下的结点链仍然具有同样的结构。

在递归定义或结构中，递归的部分必须比原来的整体简单，这样才有可能达到某种终结点(称为递归定义的出口)，显然，这种终结点不能是递归的。在递归结构中，必须存在非递归的基本结构构成的部分。如果不是这样就会出现无限递归，不能成为良好的定义。例如，结点链的空链接(前面用None表示)就是递归的终点。

作为另一个例子，一种简单表达式的递归定义如下：

常量、变量是表达式；

如果e1,e2是表达式，op是运算符，那么e1 op e2、op e1、(e1)也是表达式。

这里的常量和变量是基本表达式，它们不是递归的。其他表达式(组合表达式)是基于更简单的表达式和运算符构造起来的，这里有二元运算符、一元运算符和括号。

阶乘函数的递归计算

例如阶乘函数 $n!$ ，其数学定义就是递归的：

$$\text{fact}(n) = \begin{cases} 1, & n = 0 \\ n \times \text{fact}(n-1), & n > 0 \end{cases}$$

相应的 Python 函数定义（一种正确定义方式）如下：

```
def fact(n):TT
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

递归算法（和递归定义的函数）很有用，特别适用于所要解决的问题、要计算的函数或者要处理的数据具有递归性质的情况。但这里有一个问题：在递归函数的执行中将递归地调用自己，而且还可能继续这样递归调用。在计算机上如何实现这种过程？

考虑上面递归定义的函数 `fact`，看看它所引起的计算中的情况，以 `fact(6)` 的计算为例，不难看到下面一些显然的情况：

- 为了得到 `fact(6)` 的结果，必须先算出 `fact(5)`。
- 在计算 `fact(6)` 时这个函数的参数 `n` 取值 6，而在递归调用计算 `fact(5)` 时，函数的参数 `n` 取值 5，如此递归下去的情况类似。
- 递归调用计算出 `fact(5)` 的值之后还需要乘以 6（调用 `fact(5)` 之前参数 `n` 的值），以得到 `fact(6)` 的计算结果。这说明在递归调用 `fact(5)` 时，参数 `n` 的值是 6 的情况需要记录（保存）。同样，在计算 `fact(5)` 中调用 `fact(4)` 时，也需要记录这个调用之前参数 `n` 的值是 5 的事实。向下继续递归的情况类似。
- 显然，需要这样记录的数据的量与递归的层数成正比（线性关系），一般而言没有数量上的限制，因此不能用几个整型变量保存。
- 在这样一系列调用中保存的数据，如 6、5 等，较后保存的将被先使用，因为函数返回的顺序与调用顺序相反，后进入的层次先返回。

如前所述，这种后进先出的使用方式和数据项数的无明确限制，就说明需要（也应该）用一个栈来支持递归函数的实际运行，这个栈称为程序运行栈。

现在看一看阶乘函数导致的计算。假定需要计算 `fact(3)`，其执行中将调用 `fact(2)`，进而调用 `fact(1)` 及 `fact(0)`，图 5.2 展示了这一串调用之间的控制流和数据流关系。调用 `fact(3)` 以实际参数 3 启动函数 `fact` 的执行，执行中调用 `fact(2)`，进而调用 `fact(1)`。调用到 `fact(0)` 返回，得到的值 1 作为 `fact(0)` 的结果参与 `fact(1)` 里的计算，算出的结果进一步返回。整个过程一直进行到初始调用 `fact(3)` 返回结果 6 时结束。

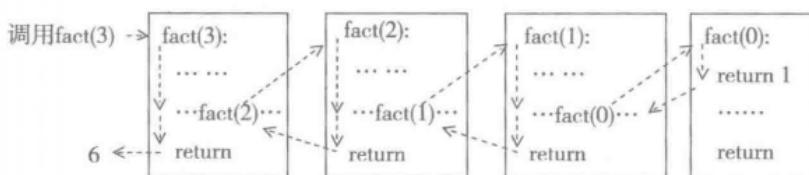


图 5.2 计算 `fact(3)` 过程中的控制和数据流程

图 5.3 描绘了在这个计算过程中程序运行栈的变化情况。各小图中标着 `n` 一列的格子表示

保存在栈里的函数参数，标着 fact 的格子表示调用 fact 的返回值，标着 res 的表示本次函数调用的结果，显然 res 应等于 $n * \text{fact}$ 。图中箭头表示状态变化的过程。

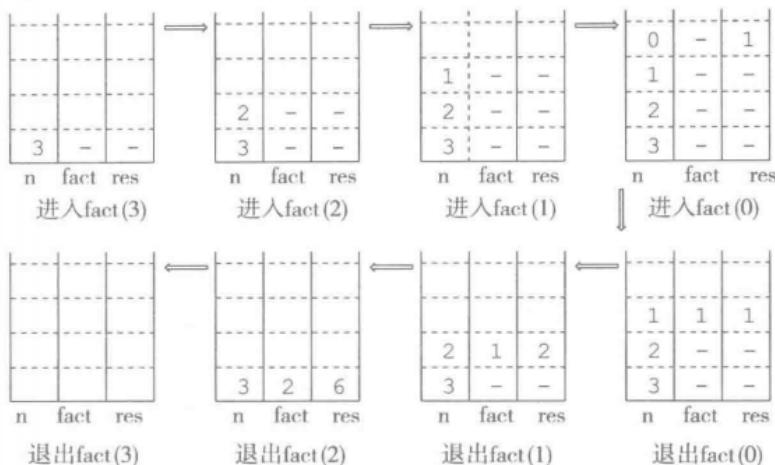


图 5.3 计算 $\text{fact}(3)$ 过程中运行栈的变化情况示意

第一个小图表示函数调用 $\text{fact}(3)$ 开始执行的状态，这时参数 3 压入栈；随后执行调用 $\text{fact}(2)$ ，参数 2 压入栈，等等。直到执行 $\text{fact}(0)$ 时直接算出 res 等于 1 后各调用逐一返回：调用 $\text{fact}(0)$ 返回结果 1，使尚未结束的 $\text{fact}(1)$ 调用算出 res 值为 1，等等。这样一层层计算并退出，直到 $\text{fact}(3)$ 算出结果 6 并最终退出为止。

栈与递归 / 函数调用

上面讨论说明，要支持递归定义函数的实现，需要一个栈（运行栈）保存递归函数执行时每层调用的局部信息，留待函数调用返回后继续使用。

一般而言，对递归定义的函数（设其名字是 rec），其执行中都有局部的状态，包括函数的形式参数和局部变量及其保存的数据。rec 在执行中递归调用自己之前的局部状态需要在相应调用之后使用，因此这些信息需要保存。

编程语言实现的做法就是用一个运行栈，对 rec 的每个调用都在这个栈上为之开辟一块区域，称为一个函数帧（简称帧），其中保存这个调用的相关信息。rec 函数的执行总以栈顶的帧作为当前帧，所有局部变量都在这里有所体现。在进入 rec 的下一个递归调用时，就为它建立一个新帧，在这个帧里实现新调用的执行。当函数从下一层递归调用中返回时，rec 的上一层执行取得下层函数调用得到的结果，执行系统弹出已经结束的调用对应的帧，然后回到调用前的那一层执行时的状态。

所有递归程序的执行都是用这种模式实现的。实际上，一般的函数调用和退出的方式也与此类似。例如在函数 f 里调用函数 g，在函数 g 里调用函数 h，h 里调用函数 r，执行时的控制和数据流程如图 5.4 所示，其形式与图 5.2 完全一样。

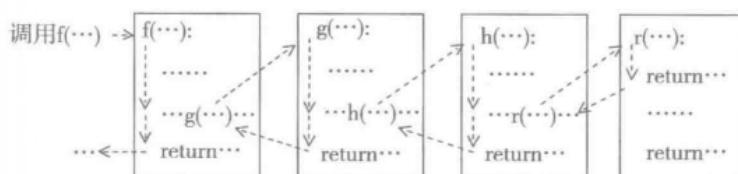


图 5.4 几个函数嵌套调用过程中的控制和数据流程

在目前各种编程语言的实现里，都按照这种模式实现了一套支持函数调用和返回的机制，其中最重要的数据结构就是一个运行栈，栈里保存所有已经开始（被调用）执行但还没有结束的函数的局部信息，包括局部变量的值约束等。

当然，一般函数调用的情况与同一个函数的一系列递归调用有些不同：每个函数的局部状态通常是不同的，它们的参数个数可能不同，局部变量的个数也可能不同。因此在一个函数被调用时，需要根据具体情况为其准备一个大小适当的函数帧。

栈与函数调用 *

在程序执行中，函数的嵌套调用是按“后调用先返回”的规则进行，这种规则符合栈的使用模式，因此栈可以很自然地支持函数调用的实现。显然，在基于栈的函数调用执行中，需要做一些规范的动作，这些动作在实际的程序代码中没有写出，但是在程序的执行中，需要由语言系统在内部完成。

函数调用时的内部动作分为两部分：在进入新的函数调用之前需要保存一些信息，退出一次函数调用时需要恢复调用前的状态。这两部分动作分别称为函数调用的前序动作和后序动作。下面简单介绍它们，以帮助读者了解高级语言执行的内幕。对一个具体的语言实现，例如一个 Python 解释器，完成这些工作的细节和顺序有可能不同，但需要做的基本事项都差不多。

函数调用的前序动作按顺序包括：

- 为被调用函数的局部变量和形式参数分配存储区（称为函数帧 / 活动记录 / 数据区）。
- 将所有实参和函数的返回地址存入函数帧（实参形参的结合 / 传值）。
- 将控制转到被调用函数入口。

函数调用的后序动作（函数返回时完成）是：

- 将被调用函数的计算结果存入指定位置。
- 释放被调用函数的存储区（帧）。
- 按以前保存的返回地址将控制转回调用函数。

每次函数调用的前后都需要执行这些动作。递归定义的函数在每次进行递归函数调用时，也都自动执行这些动作。由此可以看出，函数调用是有代价的：在得到程序代码的模块化和语义清晰性等优势的同时，可能付出执行时间的代价。新型处理器结构都针对函数调用的实现做了专门的优化设计，大大减小了函数调用带来的程序性能下降。

递归与非递归

对递归定义的函数，每个实际调用时执行的都是该函数体的那段代码，只是需要在一个内部的运行栈里保存各次调用的局部信息。这种情况说明，完全有可能修改函数定义，把一个递归定义的函数改造为（变换为）一个非递归的函数，在函数里自己完成上面这些工作，用一个栈保存计算中的临时信息，完成同样的计算工作。

现在以阶乘函数的递归定义为例，看看如何写出一种与之对应的非递归形式，用自己定义的栈模拟系统的运行栈。改造得到的一个函数定义如下：

```
def norec_fact (n): # 自己管理栈，模拟函数调用过程
    res = 1
    st = SStack();
    while n > 0:
        st.push(n)
        n -= 1
    while not st.is_empty():
```

```
res *= st.pop()
return res
```

这里未严格按递归函数执行中的情况翻译，只在栈里保存了必要的信息。例如这里的计算结果和 res 都没有进栈。但这个函数定义反映了原来那个递归函数执行中的一些基本情况。例如，函数不断递归调用的一段是不断向一个栈里压入元素，而在一个个函数返回的一段时间里，则不断取出栈中元素，加入计算。

递归函数和非递归函数

在前面的小节里给出了求阶乘的递归函数定义，上面又给出了使用一个局部栈保存计算的中间信息的非递归函数定义。实际上，可以证明：任何一个递归定义的函数（程序），都可以通过引入一个栈保存中间结果的方式，翻译为一个非递归的过程。与此对应，任何一个包含循环的程序都可翻译为一个不包含循环的递归函数。

这两个翻译过程都是可计算的，也就是说，可以写出完成这两种翻译的程序，把任何递归定义的函数翻译为完成同样工作的非递归定义的函数，或者把任何包含循环的程序翻译为不包含循环的递归程序（递归函数定义）。

在做阶乘计算的递归函数里只有一个递归调用，这种形式的递归函数定义很容易翻译成非递归的函数定义。如果函数里出现了多个递归调用，算法的翻译将复杂许多，但还是可以完成的。在一些数据结构书籍里介绍了有关的翻译算法，但有关算法的主要价值在理论方面，实用性不强，生成的定义通常很复杂，这里不准备介绍了。

如果遇到一个递归算法，希望做出它的一个非递归实现，更合适的方法是分析算法的具体情况，弄清计算的细节，然后根据得到的认识自己设计出相应的非递归函数。很多简单情况下可能并不需要用栈，但一般情况下需要一个栈。优化的设计只在栈里保存必要的信息，就像前面对 fact 所做的那样。

后面讨论二叉树时，还会进一步考虑递归与非递归算法之间的关系。

进一步说，在目前的新型计算机上，函数调用的效率损失多半都可以接受，通常直接采用递归定义的函数就可以满足需要，不一定需要考虑非递归的函数定义。只有一些极为特殊的情况，由于效率要求特别高等原因，可能需要做这种工作。

栈的应用：简单背包问题

现在介绍一个典型问题：背包问题。其求解算法用递归的方式描述很简单，但通过自己管理一个栈存储中间数据的方式定义非递归算法，则比较复杂。

问题的描述：一个背包里可放入重量为 $weight$ 的物品，现有 n 件物品的集合 S ，其中物品的重量分别为 w_0, w_1, \dots, w_{n-1} 。问题是能否从中选出若干件物品，其重量之和正好等于 $weight$ 。如果存在就说这一背包问题有解，否则就是无解。

这个问题可以有许多变形，例如可以要求在存在解时给出一个解；也可以改变问题的条件，例如物品不是 n 件而是 n 种，每种都有任意多件可用等。这个问题来自实际，是许多实际问题的抽象。现实生活中的许多货物运输安排、装车、装箱、一些材料的剪裁等，都具有这一问题反映出的一些性质。

现在考虑问题的求解。假设 $weight \geq 0, n \geq 0$ 。用记法 $knap(weight, n)$ 表示 n 件物品相对于总重量 $weight$ 的背包问题，在考虑它是否有解时，通过考虑一件物品的选或者不选，可以把原问题划分为两种情况：

- 如果不选最后一件物品（其重量是 w_{n-1} ），那么 $knap(weight, n-1)$ 的解也就是 $knap(weight, n)$

的解，如果找到前者的解也就找到了后者的解。

- 如果选择最后一件物品，那么如果 $\text{knap}(\text{weight} - w_{n-1}, n-1)$ 有解，其解加上最后一件物品就是 $\text{knap}(\text{weight}, n)$ 的解，即前者有解后者也有解。

在考虑本问题时，应该注意其特殊条件：集合中的每种物品有且仅有一件，用了或者不考虑了，也就没有了。因此有上面两种归结的情况。

根据上面分析可知，这个问题具有递归的性质：对 n 件物品的背包问题，可以归结为两个 $n-1$ 件物品的背包问题。一个针对同样重量但物品种类减一；另一个则是减少了重量，物品种类也减一。任一子问题有解，原问题就有解。

在上述两种情况下，原问题都被归结到更简单的问题，这样下去，最后可以归结到几种最简单的情况：

- 重量 weight 已经等于 0，这说明问题有解。
- 重量 weight 已经小于 0，由于不断归结中所需的重量递减，有可能出现这种情况，这说明按照已做的安排不能得到一个解。
- 重量大于 0 但已经没有物品可用，说明按照这种安排无解。

求解背包问题的递归算法

基于上述分析，不难写出一个递归定义的函数求解这一问题。

这里的问题只要求给出一个肯定或否定的回答，下面定义一个返回逻辑值的函数，返回 `True` 表示存在解，返回 `False` 表示无解。

上面三种简单情况可以直接得到有或无解的结论；前两种一般情况都把原问题归结到规模较小的问题。这几种情况分别表示计算的基础情况和递归情况，可以直接翻译为递归定义的函数。这样写出的递归函数定义如下：

```
def knap_rec(weight, wlist, n):
    if weight == 0:
        return True
    if weight < 0 or (weight > 0 and n < 1):
        return False
    if knap_rec(weight - wlist[n-1], wlist, n-1):
        print("Item " + str(n) + ":", wlist[n-1])
        return True
    if knap_rec(weight, wlist, n-1):
        return True
    else:
        return False
```

函数的三个参数分别是总重量 weight ，记录各物品重量的表 $wlist$ 和物品数目 n 。前两个 `if` 语句处理三种简单情况，直接得到 `True` 或 `False` 的结果。后两种情况通过递归调用得到结果。

在 $\text{knap_rec}(\text{weight} - \text{wlist}[n-1], \text{wlist}, n-1)$ 为真时，本层调用实际选定了一件物品。为提供更多信息，这个函数还产生一项输出，其中列出所选的各物品的顺序号和重量。这样，递归最深的位置最先输出，列出物品的顺序与其在 $wlist$ 里的顺序一致。另一可能做法即让函数调用返回所选物品的表，有关修改非常简单。

在背包问题的递归算法里出现了两个递归调用：

```
def knap_rec( weight, wlist, n ):
    if ... :
        if knap_rec(weight - wlist[n-1], wlist, n-1):
            print("Item " + str(n) + ":", wlist[n-1])
            return True
```

```

if knap_rec(weight, wlist, n-1):
    return True
else: ...

```

按规范方式翻译，得到的非递归函数定义比较长，这里不再讨论。有兴趣的读者可以参考前面的讨论，自己按部就班地完成翻译工作。

如前所述，在将具体递归算法转换为非递归算法时，通常总可以做些优化，主要是只在栈里保存必要的信息。这个问题留作练习，请读者自己考虑如何写出一个（简单些的）非递归算法。在第6章讨论二叉树算法时，还会讨论递归与非递归算法的问题。那里的算法中也涉及两个递归调用，其中的一些想法可以参考。

5.4 队列

队列(queue)，或称为队，也是一种容器，可存入元素、访问元素、删除元素。

5.4.1 队列抽象数据类型

队列中也没有位置的概念，只支持默认方式的元素存入和取出。其特点就是保证在任何时刻可访问和 / 或删除的元素，都是在此之前最早存入队列而至今未删除的那个元素。因此队列是先进先出（First In First Out，简写为 FIFO）结构。前面说过，可以借用数据的线性存储顺序表示数据存储时间的先后顺序，用线性表作为队列的实现结构，利用元素的存储顺序表示其访问顺序。由此，一些教科书里也把队列称为先进先出表。

队列的基本操作也是一个封闭集合，通常包括：创建新队列对象（如创建空队列）；判断队列是否为空（还可能需要判断满）；将一个元素放入队列（一般被称为入队，enqueue）；从队列中删除一个元素并返回它（称为出队，dequeue）；查看队列里当前（最老的）元素（但并不删除）等。图 5.5 描述了用线性表实现队列的方式。

下面是一个简单的队列抽象数据类型。易见，队列操作与栈操作一一对应，只是通常采用另一套习惯的操作名（enqueue/dequeue/peek）。

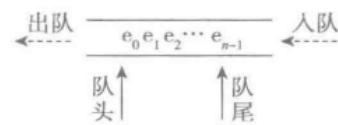


图 5.5 队列

ADT Queue:	
Queue(self)	#创建空队列
is_empty(self)	#判断队列是否为空，空时返回True否则返回False
enqueue(self, elem)	#将元素elem加入队列，常称为入队
dequeue(self)	#删除队列里最早进入的元素并将其返回，常称为出队
peek(self)	#查看队列里最早进入的元素，不删除

下面考虑队列的实现技术，其基础是线性表技术，但由于队列操作的特点和对高效操作的需求，这里还存在一些特殊的情况和问题。

5.4.2 队列的链接表实现

采用线性表技术实现队列，就是利用元素位置的顺序表示入队时间的先后关系。队列操作要求先进先出，从线性顺序看，这就要求在表的两端进行操作，不像栈那样在表的一端操作，实现起来也稍微麻烦一些。

首先考虑采用链接表技术的队列实现，这里没有实质性技术困难。由于需要在链接表的两

端操作，从一端插入元素，从另一端删除。最简单的单链表只支持首端高效操作，在另一端操作需要 $O(n)$ 时间，不适合作为队列的实现基础。前面讨论过带表尾端指针的单链表，它支持 $O(1)$ 时间的尾端插入操作。再加上表首端的高效访问和删除，基于单链表实现队列的技术已经很清晰了。如图 5.6 所示。

有了尾端指针，尾端加入元素是 $O(1)$ 时间操作，可以用作队列的入队操作 enqueue。首端访问和删除都是 $O(1)$ 时间操作，分别看作队列的 peek 和 dequeue。可见，单链表的技术和实现可以直接用于队列，只需要修改几个重要操作的名字，把 append 改为 enqueue，pop 改为 dequeue，将 top 改为 peek 等。它们都是 $O(1)$ 时间操作，已经是很满意的实现，不需要进一步讨论了。

5.4.3 队列的顺序表实现

现在考虑如何基于顺序表技术实现队列数据结构。首先分析其中的问题。

基于顺序表实现队列的困难

首先假设用顺序表的尾端插入实现 enqueue 操作，根据队列的性质，出队操作应该在表的首端进行。为了维护顺序表的完整性（表元素在表前端连续存放），出队操作取出当时的首元素后，就需要把表中其余元素全部前移，这样将得到一个 $O(n)$ 时间的操作。Python 里 list 对象的 pop(0) 操作就是这样，前面研究线性表实现时也讨论过这个问题。反过来实现的情况与此类似：从尾端弹出元素是 $O(1)$ 时间操作，但从首端插入就是 $O(n)$ 时间操作。两种设计都无法排除 $O(n)$ 操作，因此都不理想。

另一种可能是在队首元素出队后表中的元素不前移，但记住新队头位置。这一设计也有问题，图 5.7 展示了这种队列运行中的情况：假设初始情况如图 5.7a 所示，经过一系列入队和出队操作（如图 5.7b ~ e 所示），队头 / 队尾位置变量的值将随着操作移动。从操作效率看，每个操作都能在 $O(1)$ 时间完成。但另一方面，表中元素序列却好像随着操作向表尾方向“移动”，表前端留下越来越多的空位。



图 5.7 基于顺序表实现队列时操作中的情况

表元素存储区大小是固定的，经过反复的入队和出队操作，一定会在某次入队时出现队尾溢出表尾（表满）的情况。而在出现这种溢出时，表前部通常会有些空位，因此这是一种“假性溢出”，并不是真的用完了整个元素区。假如元素存储区能自动增长（例如基于 Python 的 list 实现），随着操作进行，表前端就会留下越来越大的空区，而且这片空区永远也不会用到，完全浪费了。显然不应该允许程序运行中出现这种情况。

上面讨论说明，如果基于顺序表实现队列，几种简单设计都不能令人满意。因此需要另辟

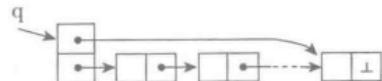


图 5.6 基于单链表的队列实现

蹊径。实际上，从图 5.7 可以看到：在反复入队和出队操作中，队尾最终会到达表存储区末端。这时表末端已经没有空闲位置供入队元素了。但与此同时，存储区首端却可能有些空位，是有可能利用的。这样就得到了一种顺理成章的设计：如果入队时队尾已经达到存储区末尾，应该考虑转到存储区开始的位置去入队新元素。

循环顺序表

前面分析提出了一种合理的基于顺序表实现队列的设计，把（一定大小的）顺序表（存储区）看作一种环形结构，认为其最后存储位置之后是最前的位置，形成一个环形，图 5.8 给出了一个实例。队列元素保存在这种结构中的一段连续位置里，有关管理和操作都能比较自然地实现。图 5.8 是一个包含 8 个单元的顺序表：

- 在队列使用中，顺序表的开始位置并不改变，图 5.8 中是一个包括 8 个位置的表，例如变量 `q.elems` 始终指向表元素区开始。
- 队头变量 `q.head` 记录当前队列里第一个元素的位置（图中是位置 4）；队尾变量 `q.rear` 记录当前队列里最后元素之后的第一个空位（图中是位置 1）。
- 队列元素保存在顺序表的一段连续单元里，用 Python 的写法是 `[q.head:q.rear]`，左闭右开区间里。

图 5.8 中的队列里有 5 个元素，存储在从位置 4 到位置 0 的一段。两个变量的值之差（取模存储区长度）就是队列里的元素个数。

实际上，上面几条也是这种队列的操作必须维护的性质。初始时队列为空，应该让 `q.head` 和 `q.rear` 取相同值，表示顺序表里一个空段。具体取值无关紧要，例如，它们都取值 0 显然满足要求。不变操作都不改变有关变量的值，不会破坏上述性质。变动操作可能修改有关变量的值，因此要特别注意维护上面的性质。

出队和入队操作分别需要更新变量 `q.head` 和 `q.rear`，正确更新操作如下：

```
q.head = (q.head+1) % q.len
q.rear = (q.rear + 1) % q.len
```

这里的 `q.len` 表示顺序表的长度。

现在考虑队列状态判断问题。按上面设计，`q.head == q.rear` 表示队空。从图 5.8 的队列状态出发做几次入队操作，可能到达图 5.9 所示的状态，这时队列里已有 7 个元素。如果再加入一个元素顺序表就满了，但又会出现 `q.head == q.rear` 的情况，这个状态无法与队列空的判断相互区分。

人们提出的一种解决办法是把图 5.9 的状态“看作”队列满，即把队满条件定义为 $(q.rear+1) \% q.len == q.head$ ，其中 `%` 表示取模运算。采用这种方法，将在表里留下一个不用的空位。实际上，基于循环顺序表，存在多种队列实现方式。下面将要介绍的队列类采用了另一种技术。

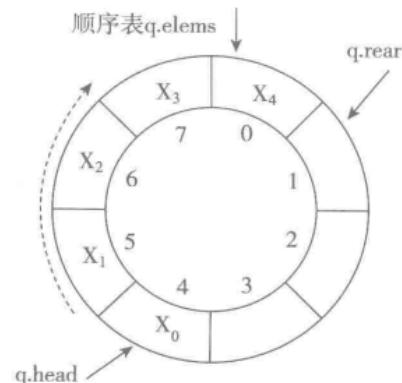


图 5.8 环形顺序表和队列

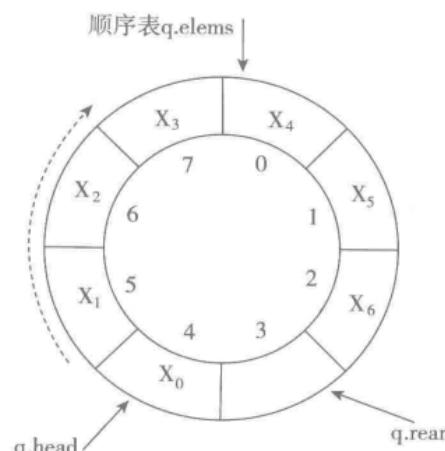


图 5.9 环形顺序表队列的满判断

队列元素可能充满顺序表的存储器，如果不希望入队操作失败，就应该采用分离式的顺序表，必要时替换更大的存储区。有关问题前面已讨论，不再赘述。

5.4.4 队列的 list 实现

现在考虑一个具体实现示例：基于 Python 的 list 实现顺序表示的队列。

如前所述，最直截了当的实现方法将得到一个 $O(1)$ 时间复杂度的 enqueue 操作和 $O(n)$ 时间的 dequeue 操作。简单实现可以基于固定大小的 list，队列满时抛出异常，有关工作留给读者。要实现更好的队列数据结构，应该采用循环队列的想法。

现在考虑定义一个可以自动扩充存储的队列类。请注意，这里很难直接利用 list 的自动存储扩充机制，有两方面原因：首先是队列元素的存储方式与 list 元素的默认存储方式不一致。list 的元素总在其存储区的最前面一段；而队列的元素可能是表里的任意一段，有时还分为头尾两段。如果 list 自动扩充，其中的队列元素就有可能失控。另一方面，list 没提供检查元素存储区容量的机制，队列操作中无法判断系统何时扩容。由于没有很好的办法处理这些困难，下面考虑自己管理存储（list）。

基本设计

首先，队列可能由于空而无法 dequeue 等，为此定义一个异常：

```
class QueueUnderflow(ValueError):
    pass
```

这里继承系统提供的 ValueError 异常，异常类的体同样为空。

将定义的顺序表队列类命名为 SQueue，首先考虑其基本设计：

- 在 SQueue 对象里用一个 list 类型的成分 _elems 存放队列元素。
- 与图 5.8 中的设计不同，这里考虑用两个属性 _head 和 _num 分别记录队列首元素所在位置的下标和表中元素个数。
- 用 Python 的 list 作为队列存储区。需要检查当前的表是否已满，必要时换一个存储表，因此要记录当前表的长度，下面用属性 _len。

数据不变式

要完成的队列是一个比较复杂的结构，其状态由一批变量（或对象属性）及其取值表示，牵涉到队列对象的四个属性 _elems、_head、_num 和 _len。在一个结构完好的队列对象里，这些属性的取值必须满足一定的关系。

另一方面，队列对象有一组操作，其中有些操作（变动操作）的执行可能改变一些对象属性的取值。为此需要弄清怎样的修改是正确的。如果某个（某些）操作的实现有错误，或者几个操作的实现相互不协调，执行这些操作就可能破坏队列对象的状态。由此可见，对于队列对象的成分修改应该有一些原则，所有操作的实现都必须遵循这些原则。实现一种数据结构里的操作时，最基本的问题就是这些操作需要维护对象属性之间的正确关系，这样一套关系被称为这种数据结构（现在考虑的是队列）的数据不变式。队列的数据不变式描述了“什么是一个完好的队列对象”。所有队列操作必须维护队列的数据不变式，才能保证队列对象在不断操作的过程中始终保持完好状态。

数据不变式说明对象的不同属性（成分）的性质，描述它们应满足的逻辑约束关系。如果一个对象的成分取值满足数据不变式，就说明这是一个状态完好的对象。数据不变式也对相关操作提出了一些约束，作为正确实现的基本保证，包括两方面：

- 所有构造对象的操作，都必须把对象成分设置为满足数据不变式的状态。也就是说，对象的初始状态应该满足数据不变式。
- 每个对象操作都应保证不破坏数据不变式。也就是说，如果对一个状态完好的对象应用一个操作，该操作完成时，还必须保证对象处于完好的状态。

显然，有了上面两条，这类对象在使用中就能始终处于良好状态。

下面队列实现中考虑的数据不变式是（用非形式的描述方式）：

- `_elems` 属性引用着队列的元素存储区，它是一个 `list` 对象，`_len` 属性记录存储区的有效容量（由于 Python 语言的特点，无法获知该 `list` 对象的实际大小）。
- `_head` 是队列中首元素（当时在队列里的最早存入的那个元素）的下标，`_num` 始终记录着队列中元素的个数。
- 当时队列里的元素总保存在 `_elems` 里从 `_head` 开始的连续位置中，新入队的元素存入由 `_head + _num` 算出的位置，但如果需要把元素存入下标 `_len` 的位置时，改为在下标 0 位置存入该元素。
- 在 `_num == _len` 的情况下出现入队列操作，就扩大存储区。

前面讨论过队列的其他设计，同样可以写出有关的数据不变式。有关情况请读者自己考虑，例如，请详尽准确地写出 5.4.3 节讨论的队列实现中的不变关系。

下面考虑用一个类来实现这种队列。在这个实现里：

- `__init__` 操作建立空队列，它设置对象的属性，保证上述不变式成立。
- 两个修改对象的变动操作（入队和出队）都维持不变式的成立。后面将仔细检查有关情况，这样一组操作形成了一套相互协调的实现。

显然，不变操作不需要检查，因为它们不改变对象状态，自然维持不变式。

队列类的实现

根据前面的设计，队列的首元素在 `self._elems[self._head]`，`peek` 和 `dequeue` 操作应该取这里的元素；下一个空位在 `self._elems[(self._head + self._num) % self._len]`，入队的新元素应该存入这里。

另外，队列空就是 `self._num == 0`，当前队列满是 `self._num == self._len`，这个条件成立时应该换一块存储区。

下面是队列类的定义：

```
class SQueue():
    def __init__(self, init_len=8):
        self._len = init_len          # 存储区长度
        self._elems = [0]*init_len     # 元素存储
        self._head = 0                 # 表头元素下标
        self._num = 0                  # 元素个数

    def is_empty(self):
        return self._num == 0

    def peek(self):
        if self._num == 0:
            raise QueueUnderflow
        return self._elems[self._head]

    def dequeue(self):
        if self._num == 0:
```

```

        raise QueueUnderflow
    e = self._elems[self._head]
    self._head = (self._head+1) % self._len
    self._num -= 1
    return e

def enqueue(self, e):
    if self._num == self._len:
        self.__extend()
    self._elems[(self._head+self._num) % self._len] = e
    self._num += 1

def __extend(self):
    old_len = self._len
    self._len *= 2
    new_elems = [0]*self._len
    for i in range(old_len):
        new_elems[i] = self._elems[(self._head+i)%old_len]
    self._elems, self._head = new_elems, 0

```

本类的 `__init__` 方法设置队列的初始状态，这个状态显然满足上面讨论的不变式。`__init__` 的参数 `init_len` 使之可以指定表的初始长度，默认长度为 8。判断空队列的操作即简单检查属性 `_num` 的值，`peek` 操作取得队列首元素，它和 `dequeue` 操作都需要先判断队列是否为空。

`dequeue` 操作取得队列首元素后修改 `_head` 属性的值，其中的取模操作保证其值正确更新。出队一个元素也使队中元素个数减一。显然，这里对两个属性的修改都没有破坏队列对象的数据不变式。

最复杂的操作是 `enqueue`，因为它在操作中可能扩大元素存储区。先考虑队列不满的情况，这时最后两个语句分别把新元素存入正确位置并更新元素计数值，它们维持了对象的不变式。如果队列满，`enqueue` 就会调用 `__extend` 方法，该方法将存储区长度加倍，把原有元素搬迁到新表里（最前面的位置），最后设置队列对象的 `_elems` 和 `_head` 属性，转到新存储区，并保证所有属性的取值相互一致。注意，这时新元素尚未入队，但 `__extend` 返回后，`enqueue` 的最后语句将正常完成这一工作。

5.4.5 队列的应用

队列在各种计算机程序和软件里使用非常广泛，下面列举一些例子，帮助读者了解有关情况。在后面章节里还会看到一些具体实例。

文件打印

一台计算机可能连着一台打印机，有些打印机连在局部网络上，用户可以通过网络把文件送去打印。

由于打印机速度有限，用户的使用需求也会有各种变化，有可能出现这样的情况：当打印机正在工作，打印一个文件的过程中，又有用户通过某些方式送来一个或几个需要打印的文件。对于多台机器共享的网络打印机，经常会出现接收到一些打印任务，但却由于正在工作不能立即处理它们的情况。在这些情况中，待打印文件都需要缓存。

打印机的管理程序（可能出现在不同层次）管理着一个缓存打印任务的队列。接到新任务时，如果打印机正在忙，管理程序就将该任务放入队列。一旦打印机完成了当前工作，管理程序就查看队列，取出其中最早的任务送给打印机。

万维网服务器

考虑连接在因特网上的一个万维网（Web）服务器系统，其功能就是接收来自因特网的请求，设法找到或做出所需要的页面，发送给提出请求的网络客户。

在服务器的运行中，会不断接收来自网络的请求。请求可能来自网络上不同地域的用户，在不同时刻这种请求的出现频度会有非常大的波动。最典型的例子如淘宝网一类的电商，或者12306铁路订票网络，它们的高峰处理任务可能比平时高出若干数量级。如果瞬时到达的请求很多，服务器系统不可能立即处理这些请求，就会把来不及处理的请求放入一个待处理请求队列。一旦服务器系统里的某个处理器（或处理线程）完成了手头的任务，它就到队列里取走一个未处理请求，通常也采取先来先服务的原则。

Windows 系统和消息队列

微软的 Windows 系统是围绕着“消息”概念和机制构造、活动的（因此被称为消息驱动的系统），所有应用程序都是围绕消息构造起来的。

在 Windows 系统运行中，各种活动（窗口界面操作、各种输入输出、程序活动）都可能 / 可以产生各种消息，要求某些系统程序或用户程序对它们做出响应。

Windows 系统维护着一些消息队列，保存接收到的各种消息。系统的消息分发机制检查消息队列里的消息，根据情况把它们分发给相应（系统或应用）程序。这些程序在处理消息的过程中又可能生成新的消息。

在 Windows 系统里工作的每个程序都有一个隐含的消息队列。程序的最高层基本结构就是一个消息处理循环，循环的每次迭代首先检查自己的消息队列，如果没有消息就进入等待状态，有消息就取出来处理。

与此类似，在一般的计算机系统里，不同处理器或处理进程（线程）之间的通信也可能需要消息队列作为缓冲（这种方式称为异步通信）。通信服务软件（或硬件）负责消息的分发和传递，把发给一个进程的消息放入该进程的消息队列。进程在需要消息时查看自己的消息队列，根据情况取出处理或者等待（如果还没有收到必要的消息）。

离散事件系统模拟

队列的另一类常见应用称为离散事件系统模拟。

离散事件系统是真实世界中许多实际系统的抽象。人们做这种模拟，是希望通过计算机程序的运行来模拟真实系统的活动情况，帮助理解真实系统实际运行中的行为，或者为计划中的实际系统的设计和实现方式做一些准备。

可以用离散事件模拟方式考察的真实系统有些共性：这类（真实世界的）系统的行为表现为一些活动，称为事件。这些事件都需要处理，而对已有事件的处理又可能产生新的事件。由于事件的产生和处理之间存在着速度上的差异和波动，因此模拟中经常会存在很多等待处理的事件。最简单最常见的处理方式是采用“先发生先处理”的工作原则，实现这种模拟就需要用队列记录正在等待的序列。

离散事件系统的实例很多，例如：

- 银行等待服务的顾客、服务席位和服务时间。
- 高速公路收费站通道和服务安排。
- 大楼电梯系统设计和安排。
- 计算机网络中的各种服务系统，等等。

在实现这种模拟系统时，一般而言，需要做下面工作：

- 通过调查或设计，选择一批模拟参数（例如，顾客抵达的平均频度）。
- 根据实际情况引入一些随机因素，以反映真实世界中各种非确定性情况（例如，确定顾客到达约为每2分钟一人，正负1分钟，有随机性）。
- 用一个或一批队列保存各种待处理活动（例如正等待的顾客）。

在生成的活动中保存一些反映真实世界情况的信息（例如，顾客到达的时间、接受服务的开始时间、离开的时间等），以便所实现的模拟系统最后能完成一些统计工作，得到有参考价值的模拟结果（如平均等待时间等），用以指导系统设计。

在离散系统模拟中，还经常需要考虑另外一些因素，如服务发生或者完成的时间、任务的紧迫性等。这种情况就不是简单的先来先服务，需要其他控制手段。在下一章讨论优先队列之后，将会考虑一个离散事件模拟系统的具体实例。

5.5 迷宫求解和状态空间搜索

本节讨论栈和队列的一大类应用，从一个具体实例（迷宫求解）开始，后面将讨论一类非常广泛的问题的处理方法。

5.5.1 迷宫求解：分析和设计

解迷宫是一类常见智力游戏，也是许多实际问题的反映和抽象。例如，在公路网或铁路网上查找可行的或最优的路线，电子地图中的路径检索，计算机网络传输的路由检索等，情况都与此类似。下面的讨论从具体到一般，从一类具体迷宫问题的求解开始。

迷宫问题

对于迷宫问题，一般是给定了一个迷宫图，包括图中的一个入口点和一个出口点，要求在图中找到一条从入口到出口的路径。各种具体的迷宫可能具有不同的表面结构，给出的迷宫图各种各样，但问题实质都差不多，是一种找路径的问题。

不难看出，搜索从入口到出口的路径，这种问题具有递归性质：

- 从迷宫的入口开始检查，这是初始的当前位置。
- 如果当前位置就是出口，已经找到出口，问题已解决。
- 如果从当前位置已无路可走，当前正在进行的探查失败，需要按一定方式另行继续搜索，这是迷宫搜索的技术或策略问题，下面讨论。
- 从可行方向中取一个向前进一步，从那里继续探索通往出口的路径。

具体迷宫是抽象问题的实例。

现在考虑一种简单形式的迷宫。图5.10左图给出了这种迷宫的一个例子，其形式是一组位置构成的矩形阵列，空白格子表示可通行，带阴影的格子表示围墙或障碍。这种迷宫是平面的，形式比较规范，每个空位置的上/下/左/右四个方向有可能也是空位置，每次允许在某个方向上移动一步。下面考虑这种迷宫的计算机求解问题。

这种迷宫可以直接映射到二维的0/1矩阵，因此很容易在计算机里表示。图5.10右边给出了左边迷宫对应的0/1矩阵，迷宫里的空位置用整数0表示，障碍和边界用1表示。下面考虑如何通过程序求解这样的迷宫。

迷宫问题分析

首先分析要处理的问题。可见：

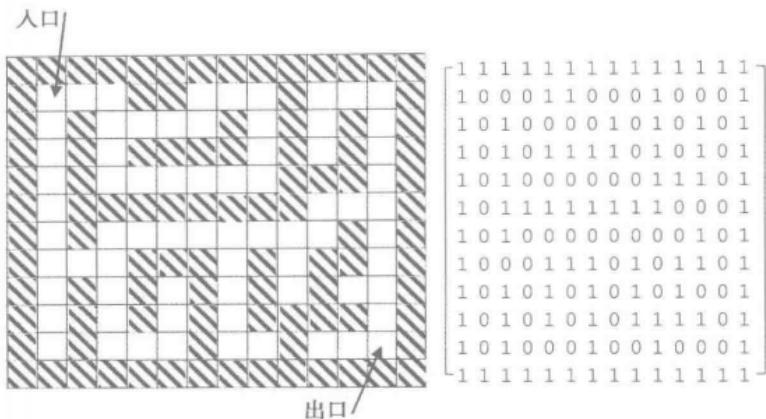


图 5.10 一个简单的迷宫

- 在一个迷宫里有一集位置，其中有些可通行的空位置相互直接连通（表现为阵列中邻接的空格，或者矩阵里邻接的 0 元素），一步可达。
- 一个空位置有几个相邻位置（其中可能有空位置，形成路径分支），也就是说，向前探查可能发现多个前进方向，可能遇到多个路径分支的情况。
- 这里的目标是找到从入口到出口的一条路径（而不是找到所有的能行路径）。
- 为找到所需路径，可能需要逐一探查前进的不同可能性（分支路径）。

由于不存在其他指导信息，这里的工作方式只能是试探并设法排除无效探索方向。这一工作中显然需要缓存一些信息：如果当前位置有多个可能的继续探查方向，由于下一步只能探查一种可能，因此必须记录不能立即考虑的其他可能方向，供后面参考和使用。不记录而丢掉了重要信息，就可能出现实际有解但却不能找到的情况。

存在不同的搜索方式，可以比较冒进，也可以稳扎稳打。如果搜索到当前位置还没找到出口，可以继续向前走，直到无路可走时才考虑后退，换一条没走过的路继续。也可以在每一步都从最早记录的有选择位置向前进，找到下一个可达位置并记录它。

显然，这里需要保存已经发现但尚未探索的分支方向信息。由于无法确定需要保存的数据项数，只能用某种缓存结构，首先应该考虑使用栈或队列。搜索过程只要求找到目标，对如何找到没有任何约束，这就意味着两种结构都有可能使用。当然，采用不同缓存结构，会对所实现的搜索过程有重要影响：

- 按栈的方式保存和使用信息，实现的探索过程是每步选择一种可能方向一直向前，直到无法前进才退回到此前最后选择点，换路径继续该过程。
- 按队列方式保存和使用信息，就是总从最早遇到的搜索点不断拓展。

这方面的细节后面有详细讨论。下面先考虑基于栈的算法。

用栈保存探索中的信息，实际上是尽可能利用已经走过的路，只有不得已时才后退到最近分支位置并转向。下面考虑采用这种方式的迷宫求解算法，算法里用一个栈保存可能存在分支的位置信息。用队列记录信息的方式留待后面研究。

问题表示和辅助结构

要解决上述迷宫问题，首先要设计一种问题表示形式。用计算机解决问题，第一步就是选择合适的数据表示。如前所述，迷宫本身可以用一个元素值为 0/1 的矩阵表示，在 Python 里可以用以 list 为元素的 list（两层的 list，基础元素是整数，相当于整数的矩阵）。迷宫入口和出

口可以各用一对下标表示（确定一对位置）。

有一个情况需要注意：搜索过程有可能在某些局部路径中兜圈子，这样，即使实际存在到出口的路径，程序也可能无休止地运行但却永远找不到解。为防止出现这种情况，程序运行中必须采用某种方法记录已探查过的位置，在继续搜索的过程中，需要不断检查有关记录，保证不重复地探查同一个位置。

记录已探查位置的基本方式有两种：①采用某种专门结构记录这种信息；②把已经检查过的信息直接标记在“地图”（迷宫图）上。专门记录的信息不容易检查；直接记录在迷宫图上的信息使用更方便，下面将采用这种方式。

假设确定了用一个矩阵表示迷宫，算法初始时通路上的位置用0表示，非通路位置用1表示。在搜索过程中需要记录已经检查过的位置，方式是把对应矩阵元素标记为2。计算中发现元素值为2就不再去探索。这种记录方式很简单，也容易判断：元素值非0就是不能通行的（或者不是通路，或者是已经探查过）。

还需要找到一种确定当前位置可行方向的技术，希望容易检查和使用。注意，在到达某个位置后，搜索过程需要选一个方向前进。如果再次退回这里，就要改走另一方向，直至所有方向都已探查为止，如果还没找到出口就应该退一步。

这里需要记录方向信息。显然，每个格子有四个相邻位置，为正确方便地处理这几个可能方向，需要确定一种系统检查方法。下面把一个位置的四邻看作其“东南西北”四个方位，按这个顺序逐一考虑各方向的位置。对于单元 (i, j) ，其四个相邻位置的数组元素下标如图5.11所示，其中东边(E)位置的下标是 $(i, j+1)$ ，南边位置的下标是 $(i+1, j)$ ，等等。

	N $(i-1, j)$	
W $(i, j-1)$	(i, j)	E $(i, j+1)$
S $(i+1, j)$		

图5.11 四邻的下标计算

为能方便地计算相邻位置，这里定义一个二元组（二元序对）的表，其元素是从位置 (i, j) 得到其四邻位置应该加的数对：

```
dirs = [(0,1), (1,0), (0,-1), (-1,0)]
```

对于任何一个位置 (i, j) ，给它加上 $\text{dirs}[0]$ 、 $\text{dirs}[1]$ 、 $\text{dirs}[2]$ 、 $\text{dirs}[3]$ ，就分别得到了该位置东西南北的四个相邻位置。有了这个表，通过一个简单的循环就可以完成检查当前位置的四邻的工作了。

为了使算法的描述更加方便，先定义两个简单的辅助函数。这里假设两个函数的参数 pos 都是二元序对，形式为 (i, j) ：

```
def mark(maze, pos):      # 给迷宫maze的位置pos标2表示“到过了”
    maze[pos[0]][pos[1]] = 2
def passable(maze, pos):  # 检查迷宫maze的位置pos是否可行
    return maze[pos[0]][pos[1]] == 0
```

5.5.2 求解迷宫的算法

有了上面的准备，现在研究求解迷宫的算法。下面将会给出几个不同的算法，还要讨论它们的特点和相互关系。

迷宫的递归求解

首先考虑采用递归方式定义迷宫求解算法。由于问题的递归性质，采用递归方式描述的算法通常相对简单，而且不需要辅助数据结构（不需要栈）。如前所述，编程语言系统（如

Python 解释器) 为支持递归程序, 在内部也用了一个运行栈。递归搜索算法实际上就是借用系统的运行栈保存中间信息。

首先把前面提出的有关迷宫求解问题的递归观点改写如下:

- 每个时刻总有一个当前位置, 开始时这个位置是迷宫入口。
- 如果当前位置就是出口, 问题已解决。
- 否则, 如果从当前位置已无路可走, 当前的探查失败, 回退一步。
- 取一个可行相邻位置用同样方式探查, 如果从那里可以找到通往出口的路径, 那么从当前位置到出口的路径也就找到了。

这些描述可以直接翻译成一个递归的迷宫求解算法。在整个计算开始时, 把迷宫的入口(序对)作为检查的当前位置, 算法过程就是:

- mark 当前位置。
- 检查当前位置是否为出口, 如果是则成功结束。
- 逐个检查当前位置的四邻是否可以通达出口(递归调用自身)。
- 如果对四邻的探索都失败, 报告失败。

递归实现的核心函数如下:

```
def find_path(maze, pos, end):
    mark(maze, pos)
    if pos == end:
        print(pos, end=" ")
        return True
    for i in range(4):
        nextp = pos[0]+dirs[i][0], pos[1]+dirs[i][1]
        # 考虑下一个可能方向
        if passable(maze, nextp):
            if find_path(maze, nextp, end):
                print(pos, end=" ")
                return True
    return False
```

函数的参数 pos 表示搜索的当前位置, 应该以迷宫的入口和出口作为实参调用这个函数。函数的基本部分是一个循环, 在这个循环中局部变量 nextp 依次取四邻位置并检查。函数中用到前面定义的数据 dirs 和两个辅助函数, 它将按照从出口到入口的顺序(反序)输出一系列下标序对, 表示路径上经历的位置。

栈和回溯法

下面考虑不用递归技术求解迷宫问题的方法, 其中一种方法称为回溯法, 这种算法在工作中执行两种基本动作: 前进和后退。

前进:

条件: 当前位置存在尚未探查的四邻位置。

操作: 选定下一个位置并向前探查。如果还存在其他可能未探查的分支, 就记录相关信息, 以便将来使用。

如果找到出口, 则成功结束。

后退(回溯):

条件: 遇到死路, 不存在尚未探查的四邻位置。

操作: 退回最近记录的那个分支点, 检查那里是否还存在未探查分支。如果有, 就取一个未探查邻位置作为当前位置并前进, 没有就将其删除并继续回溯。

已穷尽所有可能：不能找到出口时以失败结束。

易见，这里分支位置的记录和使用 / 删除具有后进先出性质，应该用栈保存位置信息。遇到分支点将相关信息入栈，删除分支点时将其信息弹出。

实际上，回溯法是一种很重要的算法设计模式，通常总是用一个栈作为辅助结构，保存工作中发现的回溯点，以便后面考虑其他可能性时使用。可以应用回溯法求解的具体问题之间的差别可能很大，它们的共性都是从一个出发点开始，设法找到目标（因此也称为搜索）；都需要使用一个栈，搜索过程的行为分为向前搜索和向后回溯。回溯法的一种典型实现方法如下（在栈里保存与搜索有关的信息）：

1) 首先把出发点放入栈中。

2) 在栈不空的条件下，反复做下面几个操作（栈空时以失败结束）：

- ① 弹出一项以前保存的信息（作为当前点）。
- ② 检查从这里出发前进的可能性（找下一个探查点）。
- ③ 如果可以向前（存在下一可行位置）：
 - 把从当前点出发的其他可能存入栈。
 - 把下一个探查点也入栈。

注意：①由于已将下一探查点入栈，下次迭代自然会将其取出使用。②如果在当前点已经不存在前进可能，算法将直接转到下一次迭代，弹出更早保存的点（就是进一步回溯）；而找到（并压入）下一探查点就是前进。

迷宫的回溯法求解

回到迷宫问题。这里需要从入口开始搜索，遇到出口时成功结束。遇分支结点时按上面说的方式记录信息，继续探查并可能回溯。

这里还有一个问题：搜索中把哪些位置入栈？存在着两种合理选择：①在从入口到当前位置的探索中，把途径的所有位置都入栈；②途经每个位置时，先检查那里的情况，只在栈里保存还存在其他未探查方向的位置。后一方式有可能节省栈空间。

仔细分析，又可以看到两个情况：

- 如果前面把一个存在未探查方向的位置压入栈中，后来回溯到这里时，该位置也可能不再存在未探查方向了（原有未探查方向在此期间已经检查过）。
- 为了在算法的最后输出找到的路径，也需要知道路径上所有的位置。

鉴于这些情况，下面算法记录探索中经过的所有位置，主要是为输出找到的路径。

迷宫问题算法框架（用一个栈记录搜索中需保存的信息）如下：

```
入口start相关信息（位置和尚未探索方向）入栈；
while 栈不空：
    弹出栈顶元素作为当前位置继续搜索
    while 当前位置存在未探查方向：
        求出下一探查位置nextp
        if nextp 是出口：
            输出路径并结束
        if nextp 尚未探查：
            将当前位置和nextp顺序入栈并退出内层循环
```

算法实现

根据迷宫的表示以及回溯后继续搜索的需要，现在存入栈的是序对 (pos, nxt)，其中分支点位置 pos 用行 / 列坐标的序对表示，nxt 是整数，表示回溯到该位置的下一探索方向，

4个方向分别编码为0、1、2、3，表示表dirs的下标。

```
def maze_solver(maze, start, end):
    if start == end:
        print(start)
        return
    st = SStack()
    mark(maze, start)
    st.push((start, 0)) # 入口和方向0的序对入栈
    while not st.is_empty():
        pos, nxt = st.pop() # 走不通时回退
        for i in range(nxt, 4): # 取栈顶及其探查方向
            for i in range(nxt, 4): # 依次检查未探查方向
                nextp = (pos[0] + dirs[i][0], # 算出下一位置
                          pos[1] + dirs[i][1])
                if nextp == end: # 到达出口，打印路径
                    print_path(end, pos, st)
                    return
                if passable(maze, nextp): # 遇到未探查的新位置
                    st.push((pos, i+1)) # 原位置和下一方向入栈
                    mark(maze, nextp) # 新位置入栈
                    st.push((nextp, 0)) # 退出内层循环，下次迭代将以新栈顶为当前位置继续
                    break # 找不到路径
    print("No path found.")
```

有关这个算法，还有几点说明。首先，算法中发现一个新位置后，总是先标记它，然后将其压入栈。这个做法保证了栈里保存的都是做过标记的位置，也保证了任何位置不会被压入两次。其次，在这个栈里，一个栈元素（位置）的下一个元素总是到达它的路径上的前一个位置。这是搜索中的一个不变性质，保证了栈里的元素构成一条路径。最后，在找到了一条到达出口的路径后程序结束，这时栈里保存的路径不包括end（它也是当时nextp的值）和当前位置pos。函数print_path应该把这两个位置和栈中的位置一起输出。这个函数的具体定义没有任何新意，这里不讨论了。

5.5.3 迷宫问题和搜索

前面说过，迷宫问题是一大类问题的代表，这类问题称为状态空间搜索问题，其基本特征可以描述如下：

- 存在一集可能状态（位置、情况等。这一状态集合可能很大）。例：迷宫问题中的所有可能位置。
- 有一个初始状态 s_0 ，一个或多个结束状态，或者有判断成功结束的方法。例：迷宫问题中的人口是初始状态，出口表示结束状态。
- 对每个状态 s ，有 $\text{neighbor}(s)$ 表示与 s 相邻的一组状态（一步可达的状态）。例：迷宫中每个位置的四个相邻位置。
- 有一个判断函数 $\text{valid}(s)$ 判断 s 是否为可行状态。前面迷宫算法里的函数 passable 实现这种判断。
- 问题：找出从 s_0 出发到达某个（或全部）结束状态的路径；或者从 s_0 出发，设法找到一个或者全部解（即，一个或者全部结束状态）。

这类问题被称为状态空间搜索或者路径搜索。

根据前面讨论，这种问题可以用递归方法求解，通过函数递归调用的方式向前探查，通过函数返回的方式回溯。也可以用一个栈保存中间信息，采用非递归方式通过回溯法求解。可以（需要）通过状态空间搜索解决的问题很多，经典的简单实例如：

- 八皇后问题（在国际象棋棋盘上为八个皇后安排位置，使之不能相互攻击）。
- 骑士周游问题（在国际象棋棋盘上为骑士找到一条路径，使之可以经过棋盘的每个格子恰好一次。骑士走法与中国象棋的马一样，走“日”字）。

这些都可以作为状态空间搜索问题的练习。

许多实际应用问题需要通过空间搜索的方式解决，例如许多调度、规划、优化问题（如背包问题），数学定理证明（基于一些事实和推理规则）等等。

状态空间搜索：栈和队列

对于一个希望用计算的方法处理的问题，根据人们对问题的认识深度，存在两种不同的思考和处理方法：

- 如果对问题的研究和理解比较深入，已经有了全局性和系统性认识，就可能做出一个专门算法，直接去解决问题。这样的算法直接而高效，专门针对所处理问题。研究解决问题的高效算法是计算机领域始终不渝的追求。
- 但是对于许多问题，人们的认识还不够深入全面，做不出解决问题的算法。这样的问题还能用计算机处理吗？人们一直在思考。实际上，如果已经有一些对问题空间的局部性认识，就有可能把该问题的求解转化为一个状态空间的搜索问题。因此可以说，搜索法是一种通用问题求解方法（general problem solving）。

显然，针对一个问题的算法能一网打尽该问题的所有实例，而搜索方法则是一种试探。由于面对的问题存在许多未知，搜索可能成功也可能失败，或者无穷无尽地运行而得不到结果。

在搜索的进展过程中，面临的情况一般是：已经探查了从初始状态可达的一些中间状态；一些已经探查的中间状态存在着尚未探查的相邻状态。显然，对任何从初始状态可达的中间状态，与它相邻的可达状态也是从初始状态可达的。从一个中间状态继续向前探查，可能得到新的可达状态。如果某个新确定的可达状态是结束状态，那就找到了初始状态到结束状态的路径；否则，就应该把新的可达状态加入已探查的中间状态集。也就是说，搜索过程中需要记录那些存在未探查邻居的中间状态，以备后面使用。

为记录存在尚未完全探索后继状态的中间状态，需要用缓存结构。原则上说栈和队列结构都可以使用，但结构的选择将对搜索进展方式产生重大影响。现在分析这个问题。

在前面的迷宫算法里用的是栈，栈的特点是后进先出。“后进”的状态就是在搜索过程中较晚遇到的状态，即是与开始状态距离较远的状态。“后进先出”意味着总是从最后遇到的状态出发考虑继续向前探索，实际表现是尽可能向前检查，尽可能向远处探索，只有在后来的状态已经无法继续前进时，才考虑退回到前面最近保存的状态，换一种可能性继续搜索。后退并考虑其他可能性的动作就是回溯。

如果用队列作为缓存结构，队列的先进先出特性就会反映到搜索过程中。“先进”队列的状态是在搜索过程中较早遇到的状态，也就是与开始状态距离较近的状态。“先进先出”要求先考虑距离近的状态，先考虑从这些状态向外扩展，这样产生的实际上是一种从各种可能性“齐头并进”式的搜索。在这种搜索过程中没有回溯，只是一种逐步扩张的过程。作为示例，下面考虑基于队列的迷宫求解算法。

基于队列的迷宫求解算法

基于队列的迷宫求解算法重用前面所有基本设计，包括问题的表示方式（矩阵表示和矩阵元素的取值方式）、枚举相邻位置的方法（方向表 dirs 和对方向的循环处理），以及两个基本操作函数（标记函数 mark 和位置检查函数 passable）。

新算法的基本框架：

```
将start标记为已达
start入队
while 队列里还有未充分探查的位置：
    取出一个位置pos
    检查pos的相邻位置
        遇到end成功结束
        尚未探查的都mark并入队
队列空，搜索失败
```

基于这一算法框架，很容易定义相应的函数，其基本结构与前面用栈的算法类似，只是改用队列作为缓存结构（这里没有递归）：

```
def maze_solver_queue(maze, start, end):
    if start == end:                                # 特殊情况
        print("Path finds.")
        return
    qu = SQueue()
    mark(maze, start)
    qu.enqueue(start)                             # start位置入队
    while not qu.is_empty():                      # 还有候选位置
        pos = qu.dequeue()                         # 取出下一个位置
        for i in range(4):                         # 检查每个方向
            nextp = (pos[0] + dirs[i][0],
                       pos[1] + dirs[i][1]) # 列举各位置
            if passable(maze, nextp):               # 找到新的探索方向
                if nextp == end:                    # 是出口
                    print("Path find.")          # 成功！
                    return
                mark(maze, nextp)
                qu.enqueue(nextp)                 # 新位置入队
    print("No path.")                            # 没有路径，失败！
```

虽然上述算法能完成迷宫搜索，但还有一个问题没处理：找到的路径在哪里？

在前面基于栈的迷宫求解算法里，栈中保存着一些位置，在每个位置之下就是到达它的路径上的前一个位置。这样，搜索中找到出口时，当时的栈正好保存着从入口到出口的一条路径，只需追踪栈中元素就可以得到找到的路径。

而对基于队列的算法，队列里保存的位置及其顺序与路径无关。例如，如果一个位置有几个“下一探查位置”，它们就会顺序进入队列。这样，在找到了出口时，根据队列里当时的信息，没办法追溯出一条成功路径。这一情况说明，要想在队列算法结束时得到从迷宫入口到出口的路径，必须在搜索中另行记录有关信息。

注意：假设搜索中从当前位置 a 找到了下一位置 b，如果从 b 能到达出口，那么 a 就是最终的成功路径上位置 b 之前一个位置，而这个关系就是最终路径里的一段。为了在搜索到达出口时能获得相关路径，遇到一个新位置时就需要记住其前驱位置。只要算法过程中做好这种记录，到达出口后反向追溯就可以得到所需路径了。

在这里需要记录的是一种关系，从一个对象找到与之相关的另一对象。在 Python 里记录这种关系，最方便的方法是使用字典结构 dict。读者不难自己做出相应的修改。

基于栈和队列的搜索过程

上面讨论说明，如果需要做搜索，栈或队列都可以用作其中的缓存结构。但是，在什么情况下应该优先考虑某一种方式呢？为了回答这个问题，需要深入分析这两种结构蕴涵的搜索方

式，理解它们的性质。下面的讨论还是以迷宫搜索为例。

图 5.12 上部给出了基于栈搜索迷宫的一个场景，其中标交叉符号的是已经检查过的，并已确定不可能在通往出口路径上的位置。圆圈表示当时栈里记录的位置。注意，这里是按东南西北的顺序检查四邻。

从这个图中可以看到一些情况。首先，基于栈的搜索可能进入一个局部区域，只有穷尽了那里的状态并发现无法到达目标后才能从那里退出来。这种搜索比较“冒进”，可能在一条毫无价值的路上走很远，褒义说是勇往直前，贬义就是“不撞南墙不回头”。如果顺利，可能只探查了不多的位置就找到了解，但也可能陷入很深的无解区域。另外，如前所述，任何时候栈里保存的位置总是从入口开始的一条路径。

图 5.12 下半部分给出的是基于队列搜索迷宫过程中的一一个场景。图中标交叉符号的位置已检查过，标圆圈的是当时队列里记录的位置。有一个情况非常值得注意：已经检查的位置连成一片，其中没有遗漏，而位于队列里的位置构成这片区域的前沿，它们构成已检查区域与未探索区域之间的分界。

基于队列搜索是一种稳扎稳打、步步为营的搜索，只有在检查完所有与入口同样距离的位置之后才更多前进一步。

基于搜索过程的特点，人们把基于栈的搜索称为深度优先搜索（depth-first search），把基于队列的搜索称为宽度优先搜索（width-first search）。

* 深度和宽度优先搜索的性质

深度优先和宽度优先是两种最基本的搜索方法。本小节对它们做一些分析和对比。实际需要搜索时，应根据问题的性质和搜索方法的特点选择合适的方法。

假设搜索问题有解，能否保证找到解：应该注意到，深度优先方法总是沿着遇到的搜索路径一路前行，在遍历完沿一条路径走下去可达的搜索区域后，才会退出这条路径。如果一个不包含解的子区域很大，算法进入这样的子区域就会浪费很多时间。如果存在包含无穷多状态的无解子区域，即使其他地方有解，深度优先搜索算法也可能永远找不到解。宽度优先搜索则不是这样。由于它是从近到远逐步“扫荡”，只要存在到达解的有穷长路径，这种搜索过程一定能找到解，而且最先找到的必定是最短的路径（最近的解）。

从上面讨论还可以看到一个情况：使用深度优先搜索，在分支结点对不同分支的选择非常重要。前面迷宫算法没考虑这方面情况，一方面问题比较简单，另外也没有其他可以帮助做选择的信息。处理实际问题时，应该关注分支选择问题。

如果找到解，如何得到相应路径：在基于栈的搜索中，很容易在栈里保存一条路径上历经的状态序列。如果采用基于队列的搜索，就必须用其他方法记录与路有关的信息。前面讨论中提出的技术是在到达每个状态时记录其前一状态，最后追溯这种“前一状态”链，枚举出路径

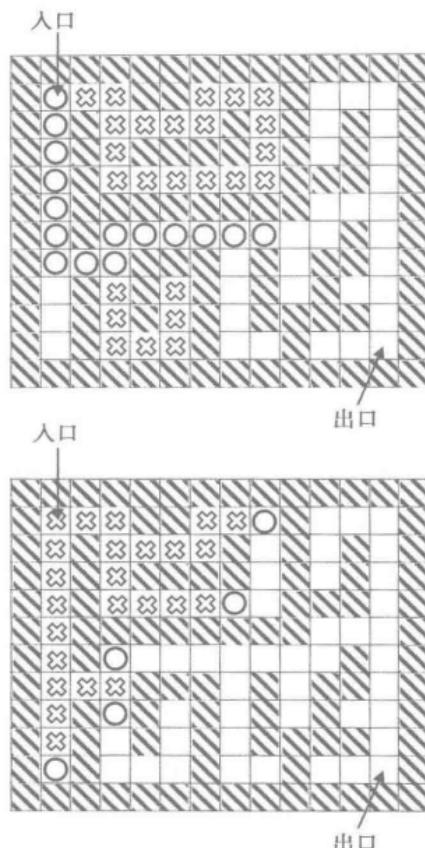


图 5.12 基于栈或队列的迷宫路径搜索

上的状态（相当于对每个状态形成一个栈）。对于迷宫这类状态不多的情况，这种技术的代价可以接受。但如果状态空间巨大，前一状态记录也可能成为很大的负担，特别是其中很多记录可能完全没有用，记录的信息未必与解路径有关。

搜索所有可能的解和最优解：前面迷宫实例只要求得到一个解（一条路径），实际中也可能要求得到所有的解，或者最优解（例如最近的解）。深度优先搜索在找到了一个解之后继续回溯，有可能找到下一个解，遍历完整个状态空间就能找到所有的解。另外，只有找出所有的解，才能从中选出最优解。对宽度优先搜索，找到一个解之后继续也有可能找到其他解，但是到各个解的路径将越来越长，得到的第一个解就是最优解。

如果状态空间不是有穷的，就不可能枚举出所有的解；如果状态空间非常大，枚举出所有的解也不现实。对这些情况，不应该用深度优先搜索去找最优解。

搜索的时间开销：应该注意到，无论是使用栈还是队列，探查一个状态的开销都可以看作 $O(1)$ 时间操作。因此，求解搜索问题的时间代价受限于状态空间的规模。实际求解的代价正比于找到解之前访问的状态个数。有关情况上面已经讨论了。

搜索的空间开销：在搜索状态空间的过程中需要保存一些中间状态，搜索算法的空间开销就是在整个搜索过程中保存在栈或队列里的最大元素项数。两种搜索方法在这方面的表现也很不一样，实际情况也是由状态空间的情况决定的。对于深度优先搜索，所需要的栈空间由找到一个解（或者所有解）之前遇到过的最长那一条搜索路径确定。最长路径越长，计算中需要的存储量就越大。而对宽度优先搜索，所需的队列空间由搜索过程中可能路径分支最多的那一层确定。如果搜索中出现分支非常多的情况，算法需要的存储量就可能非常大。此外，采用宽度优先搜索，为了得到路径，还需要另外的存储，其存储量与搜索区域的大小线性相关。图 5.12 中宽度优先搜索实例的宽度非常小，这是一个具体情况，并不具有代表性。实际中经常遇到极宽的搜索空间，存储中间状态和路径都可能代价很大。

总结：如果一个问题可以用空间搜索的方式解决，采用栈或者队列实现深度优先或宽度优先的搜索，都有可能解决问题。具体采用什么技术要看实际问题的各方面性质。特别是搜索空间的深度和宽度情况，它们决定搜索的空间开销；还需考虑是要找到一个解、全部解，还是最优解等情况。上面对两种搜索算法的性质分析可供参考。

5.6 几点补充

最后介绍一些与栈和队列有关的情况，作为前面基本内容的补充。

5.6.1 几种与栈或队列相关的结构

人们也常把栈和队列看作访问受限的线性表。栈是只能在一端插入和删除元素的表，队列是在一端插入而在另一端删除元素的表。两者都是访问方式最少的容器。实际中也可能需要放松一些限制，例如只能在一端插入元素，但允许从两端检查和删除；或保持只允许一端访问，但可以在两端插入元素。这些构成了表或队列的扩充结构。

双端队列

在实际中上述结构都不太重要，但人们经常把这几种需求合在一起，提供一种称为双端队列的结构，并为其打造了一个特殊的名字 deque（发音为“迪克”），它允许在两端插入和删除元素，因此功能覆盖上面的所有结构。作为效率要求很高的缓存结构，这里要求两端的四种变动操作都应该具是 $O(1)$ 时间复杂度。

根据至今为止的讨论，可以看到双端队列的两种实现方式。首先，双链表结构可以实现两端的常量时间插入和删除操作，因此可用于实现双端队列。本章在介绍队列的连续表实现时，开发了一种“循环表”结构，它也可以支持双端操作，能得到平均的 $O(1)$ 时间，只是在扩充存储区时需要线性时间。有关实现留作练习。

Python 的 deque 类

Python 标准库的 `collections` 包里定义了一种 `deque` 类型，提供了一种 Python 版的双端队列，支持元素的两端插入和删除。`deque` 对象的两组插入和删除都是常量时间操作，因此满足上面的要求。`deque` 类采用一种双链表技术实现，但其中每个链接结点里顺序存储一组元素。这样，靠近两端的下标访问也可以在常量时间完成。

除了最基本的两端插入和删除操作 `append`、`appendleft`、`pop`、`popleft` 之外，`deque` 还支持另外一些操作，包括在两端扩充一段元素等。有关情况请查看 Python 手册。

由于双端队列的功能涵盖队列的功能，因此在开发实际应用时，也可以直接用标准库的 `deque` 作为队列的实现，选择一对操作作为入队 / 出队操作。

5.6.2 几个问题的讨论

栈和队列是常用结构，其操作效率极为重要。读者可能提出这样的问题：既然链接实现能提供统一的 $O(1)$ 时间操作，为什么还要考虑顺序实现？前者可以保证 $O(1)$ 操作时间，而后的插入操作平均为 $O(1)$ 时间，但最坏情况需要线性时间。

这个问题的回答不很简单，牵涉到计算机硬件的一个重要特性。在实际计算机里，能与新型 CPU 的速度匹配的存储器的成本非常高。为提高性价比，目前各种计算机都采用分级缓存结构，填补 CPU 与主内存（相对很慢）之间的速度差。如果程序要用某内存单元 d 的数据，缓存管理硬件就会把包含单元 d 的一片连续内存单元复制到高速缓存。如果随后 CPU 的一批操作中使用的数据都在这片单元范围内，它就直接在高速缓存里使用，不需要再实际访问内存，大大提高了操作效率。与内存相比，高速缓存的存储量小得多，能放在那里的数据有限，硬件负责自动管理在内存和高速缓存之间的数据交换。

这种机制造成了一种现象：如果连续进行的一批内存访问是局部的，操作速度就会快得多。因此人们在考虑程序的效率时，一个重要线索就是尽可能使对计算机内存的使用局部化。在 Python 语言的层面上，顺序表是局部化的典型代表，在可能的情况下应尽量使用，特别是在效率要求较高的计算中。另一方面，链接结构的一个特点是，其中结点可能在内存中任意分配。即使程序顺着链接逐个访问结点，在内存层面的表现通常也是在许多位置随机的单元之间跳来跳去。因此，链接表在带来灵活性的同时，效率上可能有明显的付出。这些情况就是需要考虑顺序表实现的最重要原因。

另外，在一些语言里，建立顺序表结构还可以避免复杂的存储管理。Python 的 `list` 不是那样，因此可以灵活增长。Python 提供了另一些可能效率更高的数据类型，例如内置的 `bytes` 和 `bytearray` 类型，库中的 `array` 类型等。它们可以看作受限的 `list`，只能存储特定类型的对象，如数值或字符，但存储效率和操作效率较高。其中一个因素是 `list` 采用元素外置的存储方式，而上面几种类型都采用元素内置的方式（参看图 3.1）。注意，对于 `list`，从表结构到元素也是链接，可能带来效率损失。

本章总结

栈和队列是使用最广泛的两种辅助性数据结构，主要用于数据缓存，在算法和实际系统中经常可以见到。栈和队列都是最简单的容器，可以保存一些数据元素供后面的计算中使用。在对这两种结构存入或取出数据时，不能指定数据的位置或内容，它们只支持默认方式的操作。栈和队列的差异在于存入和取出数据元素的顺序。栈保证任何时候可以访问和删除的元素都是此前最后存入的那个元素，以后进先出（LIFO）作为基本性质。队列保证任何时候可以访问和删除的元素都是此前存入但尚未删除的元素中最早进入队列的元素，以先进先出（FIFO）作为基本性质。

栈和队列需要保证的性质与时间有关。时间是线性的，因此最自然的实现方式就是用数据结构的存储顺序表示时间先后。也就是说，用线性表中元素的顺序表示栈和队列中元素到达的顺序。一般的栈和队列都基于线性表的技术实现。正因为这样，栈也被一些人称为后进先出表，队列被称为先进先出表。

线性表的两种实现技术——顺序表和链接表，都可以用于实现栈和队列。栈的实现比较简单。后进先出意味着各种重要操作都在表的一端进行，自然应该选择插入和删除操作效率最高的一端。对顺序表，应该在尾端操作；对链接表，应该在首端操作。

队列要求先进先出，因此需要在表的一端插入，另一端删除。对于链接表，加了尾结点指针后，尾端插入具有 $O(1)$ 复杂度。将其作为元素入队端，首端作为出队端，就能得到高效的实现。用顺序表实现队列，还需采用循环顺序表的技术。

栈和队列有很多应用，包括支持程序设计语言中的函数调用和递归函数实现、状态空间搜索等非常重要的应用，以及大量具体应用。发现程序里需要栈或队列，首先是看到程序运行中会不断产生一些数据，需要保存起来以备后面使用。这种情况说明需要引进一种缓存结构。具体应该用栈还是队列，要根据实际应用对元素使用顺序的要求。本章许多例子的讨论中都非常详细地分析了面临的情况，可供参考。

在讨论基于循环顺序表的队列实现的一节里还介绍了数据不变式的概念，以及如何借助于数据不变式，保证实现一个数据结构的一批操作相互协调。有关的概念和讨论非常重要，应认真领会，在实现各种数据结构时应灵活应用。

练习

一般练习

- 复习下面概念：容器，元素，容器数据结构，栈（堆栈），队列（队），缓冲存储（缓存），后进先出（LIFO，后进先出表），先进先出（FIFO），实现结构，入栈（压入）操作，出栈（弹出）操作，栈顶，栈底，括号匹配问题，表达式的中缀表示（前缀表示，后缀表示），波兰表达式，逆波兰表达式，表达式求值，表达式形式转换，运算符栈，数据栈，函数的递归定义，递归结构，递归调用，运行栈，函数帧（栈帧），入队，出队，循环顺序表，数据不变式，消息，消息驱动的系统，消息队列，离散事件系统和模拟，迷宫问题，当前位置，探查，回溯法，搜索，状态空间搜索，路径搜索，通用问题求解方法，深度优先搜索，宽度优先搜索，最优解，双端队列（deque）。
- 考虑符号 a、b、c、d 按顺序进栈，允许在进栈过程中任意插入、弹出操作。请列出这样做可

能产生的所有出栈元素序列。

3. 如果栈的压入序列为 1、2、3、4、5，则下面哪个（哪些）不可能是栈的弹出序列：
 - a) 2, 3, 4, 1, 5
 - b) 5, 4, 1, 3, 2
 - c) 2, 3, 1, 4, 5
 - d) 1, 5, 4, 3, 2
4. 设顺序将整数 1, 2, 3, …, n 压入栈中，并全部弹出形成的输出序列是 a_1, a_2, \dots, a_n 。如果 a_1 是整数 k，那么 a_2 可能是什么？如果 a_i 是整数 k，那么 a_{i+1} 可能是什么？
5. 在前一题目的假设下，产生的输出序列中不可能出现 $i < j < k$ ，使得 $a_j < a_k < a_i$ 。请设法证明这个结论。
6. 把下面中缀表达式翻译为与之等价的前缀形式：

$$(a + b) * (c - d)$$

$$2 * (a + b) / c / d$$
7. 把下面后缀表达式翻译为与之等价的中缀形式：

$$a\ b\ c\ / f\ d\ -\ *\ 3\ r\ / + -$$

$$2\ a\ +\ b\ / c\ d\ -\ e\ f\ -\ *\ /$$
8. 写出下面数学表达式的前缀形式和后缀形式：

$$1 + (2 - 3 \times 5) / 4$$

$$(7 + 4) \times (2 - 6 / 3) + 4$$
9. 假设火车调度场有一个 Y 形调度线，其一条支线上有一列客车车厢，其中任意交错出现硬座车厢和卧铺车厢，现需要将它们重新排列为硬座在前卧铺在后的一列客车，从另一支线推出。这里只有一个车头在 Y 形铁路顶端，请为其设计一个调度算法。
10. 将 1、2、3、4 作为双端队列输入数据（可在任一端加入），存在多少种可能的输出序列？如果输入数据是 1、2、3、4、5，情况怎么样？

编程练习

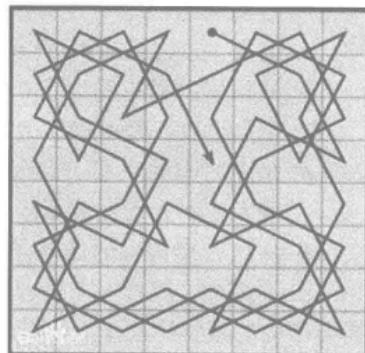
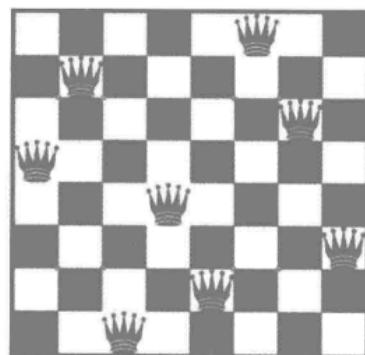
1. 改造 5.3 节的括号匹配检查函数，让它从指定文件读入数据，在发现不匹配时，不仅输出不匹配的括号，还输出该括号在原文件里的行号和字符位置。
2. 请修改前一题完成的函数，使之可以检查实际 Python 程序里的三种括号匹配。
3. 考察 Python 语言标准，改进中缀表达式到后缀表达式里的 tokens 生成器，使之能更好地处理所有符合 Python 语法的算术表达式。
4. 请定义一个函数，它组合使用本章讨论的中缀表达式到后缀表达式的转换函数和后缀表达式的求值函数，实现对中缀表达式的求值功能。
5. 请定义一个独立的中缀表达式求值函数，不经过后缀表达式转换。注意，一种技术如本章 5.3.2 节中讨论的，使用两个栈，一个存储运算对象和计算的中间结果，另一个存储尚未处理的运算符。也可以考虑只用一个栈的方法。
6. 请定义一个函数，它的输入是一个后缀表达式，输出是一个加了所有括号的中缀表达式。这里不考虑括号的必要性，加进所有括号可保证计算顺序正确。请分析算法的时间和空间复杂度。
7. 设法实现一个不用递归的背包求解函数。实现该算法的关键在于需要考虑两种递归情况时，

必须把一种可能保存起来，然后按另一可能继续走下去。如果按一种可能做下去能得到结论，保留的信息都可以丢掉；否则就需要找到最近的另一种可能安排，检查能否从这里找到解。这里还有一个难点是如何输出合格的物品配置。

8. 有一个非常著名的函数称为 Ackerman 函数，其特点是函数值增长得特别快。该函数的定义如下：

$$\text{Ack}(m, n) = \begin{cases} m - 1 & m = 0 \\ \text{Ack}(m-1, 1) & m \neq 0 \wedge n = 0 \\ \text{Ack}(m-1, \text{Ack}(m, n-1)) & m \neq 0 \wedge n \neq 0 \end{cases}$$

- a) 请定义一个递归函数实现 Ackerman 函数；
 - b) 请定义一个非递归函数实现 Ackerman 函数；
 - c) 请画出用非递归函数计算 $\text{Ack}(2, 1)$ 的过程中栈的变化情况。
9. 请详尽描述 5.4.3 节讨论的队列实现的数据不变式，基于这组不变式实现另一个队列类，并逐一验证所实现的操作确实满足有关不变式的要求。
10. 请基于连续表技术实现一种双端队列结构，保证其两端插入和删除均为常量时间操作。
11. 双端队列的一种显然实现方法是采用双链表技术，请做出这种实现。
12. 八皇后问题（参考右图上）：国际象棋的棋盘为 8 行 8 列 64 个格子，棋子置于格中。皇后是国际象棋里威力最大的棋子，可以在直、横和两个斜线方向上攻击其他棋子。八皇后问题要求在棋盘上为八个皇后安排位置，使之不能相互攻击。例如，右图下给出的是问题的一个解。请为求解这个问题设计一套适当的数据表示，并实现一个求解该问题的非递归程序。进而：
- a) 修改上面定义的函数，使之可以产生出所有满足条件的皇后布局。
 - b) 修改上面工作使之能处理一般的 n 皇后问题。
13. 骑士周游问题：国际象棋中的骑士，行棋方式与中国象棋中的马类似，走“日”字。现在的问题是在国际象棋棋盘上为骑士找到一条路径，使之可以经过棋盘的每个格子恰好一次。右图是这个问题的一个解。请定义一个非递归函数求解这个问题，函数的参数是骑士的初始位置，程序求出路径，返回途经位置的表。
14. 请设法统计在骑士周游过程中程序回溯的步数及其随着已走路径长度的变化而变化的趋势：
- a) 你可能发现，随着剩余结点越来越少，回溯的情况会越来越多，连续回溯的步数也会越来越长。请分析原因，设法提出缓解方法。
 - b) 请修改所用搜索方法，加入适当的分支选择策略，提高求解程序的效率。



第6章 二叉树和树

下面几章将研究一些复杂的数据结构，它们也是一些基本元素的汇集，但元素之间不是简单的线性关系，可能存在更复杂的联系。这种情况下，在一个包含 n 个元素的数据结构里，元素之间的最远距离就不是 n ，可能小得多。

元素之间的复杂联系可用于表示数据之间更复杂的关系，实际中也确实有这种需要。更复杂的联系给数据的组织和使用带来更多可能性，也带来更多问题。复杂的数据结构可能存在更多不同的组织方式，通常也存在更多不同的合理实现方法。另一方面，处理结构中元素的方法（算法）也可能变得比较复杂，常常需要借助于一些辅助数据结构，如栈和队列。数据之间的复杂结构可能影响处理算法的设计和复杂性。例如，可能使一些问题存在高效的解决算法，也可能使有些操作更难完成，算法效率更低。

本章研究复杂结构中最简单的一类结构，称为树形结构。这是一类非常重要的结构，在实际中使用广泛，它们不但本身很有用，还反映了许多计算过程的抽象结构。这些都将随着讨论而逐渐清晰起来。本章将要讨论的树和二叉树，都属于树形结构。

树形结构也是由结点（结构中的逻辑单元，可用于保存数据）和结点之间的连接关系（一种后继关系）构成，但其结构与线性结构（表）不同，最重要的特征包括：

- 一个结构如果不空，其中就存在着唯一的起始结点，称为树根（root）。
- 按结构的连接关系，树根外的其余结点都有且只有一个前驱（这点与线性结构一样）。但另一方面，一个结点可以有 0 个或者多个后继（因此与线性结构不同）。另外，在非空的树结构中一定有些结点并不连接到其他结点。这种结点与表的尾结点性质类似，但在一个树结构里可以存在多个这种结点。
- 结构里的所有结点都在树根结点通过后继关系可达的结点集合里。换句话说，从树根结点出发，经过若干次后继关系可以到达结构中的任一个结点。
- 结点之间的联系不会形成循环关系，这也说明，结点之间的联系形成了一种序，但一般而言不像线性表那样形成一个全序。
- 从这种结构里的任意两个不同结点出发，通过后继关系可达的两个结点集合，或者互不相交，或者一个集合是另一个集合的子集。

树形结构的重要特征使之与其他更复杂的结构不同。此外，在树形结构里的结点形成了一种层次结构，可以用于表示各种常见的层次性关系。

下面首先讨论树形结构中相对简单，也是使用最广泛的二叉树结构。

6.1 二叉树：概念和性质

二叉树是一种最简单的树形结构，其特点是树中每个结点至多关联到两个后继结点，也就是说，一个结点的关联结点数可以为 0、1 或 2。另一个特点是一个结点关联的后继结点明确地分左右，或为其左关联结点，或为其右关联结点。

6.1.1 概念和性质

本节介绍二叉树的定义和一些相关概念，给出二叉树的一些重要性质。

定义和图示

定义（二叉树） 二叉树是结点的有穷集合。这个集合或者是空集，或者其中有一个称为根结点的特殊结点，其余结点分属两棵不相交的二叉树，这两棵二叉树分别是原二叉树（或说是原二叉树的根结点）的左子树和右子树。

显然，上面二叉树的定义是一个递归定义，所定义的二叉树是一种递归结构。一棵二叉树可能有两棵子树，其子树也是二叉树，结构与整棵树相同。对于非空二叉树，其结点集合非空，至少包含一个根结点。但是其子树可以为空，两棵子树可以都为空。如果根结点的子树都空，就是一棵只包含根结点的二叉树。另请注意，二叉树的两棵子树有明确的左右之分，讨论子树时必须明确说明是左子树还是右子树。

二叉树有一种很直观的图形表示，能帮助理解其抽象定义。图 6.1 给出了几棵二叉树的图示，其中的小圆圈代表二叉树的结点，根结点画在最上面，其两棵子树画在下面的左右两边，用两条连线连接根结点和这两棵子树。

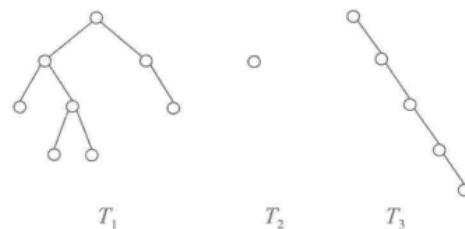


图 6.1 三棵二叉树

【例 6.1】 图 6.1 中给出了几棵二叉树。其中左边一棵比较符合直观，根结点有两棵子树，具有一层层的结构；中间是一棵只包含根结点的二叉树；右边也是一棵二叉树，但其每个结点（除最下一个外）都只有一棵右子树。

几个基本概念

现在介绍一些与二叉树有关的概念。

不包含任何结点的二叉树称为空树；只包含一个结点的二叉树是一棵单点树；一般而言，一棵二叉树可以包含任意（但有穷多）个结点。

一棵二叉树的根结点称为该树的子树根结点的父结点；与之对应，子树的根结点称为二叉树根结点的子结点。注意，父结点和子结点的概念是相对的。

可认为从父结点到子结点有一条连线，称为从父结点到子结点的边。注意，这种边有方向，形成一种单方向的父结点 / 子结点关系（父子关系）。基于父子关系可以定义其传递关系，称为祖先 / 子孙关系，它决定了一个结点的祖先结点，或子孙结点。另外，父结点相同的两个结点互为兄弟结点。易见，一棵二叉树（或其中子树）的根结点 r 是这棵树（这棵子树）中所有其他结点的祖先结点，而这些结点都是 r 的子孙结点。

在二叉树里有些结点的两棵子树都空，没有子结点。这种结点称为树叶（结点）。树中其余结点称为分支结点。注意：分支结点可以只有一个分支（一个子结点）。对于二叉树，只有一个分支时必须说明它是其左分支还是右分支。

一个结点的子结点个数称为该结点的度数。显然，二叉树中树叶结点的度数为 0，分支结点的度数可以是 1 或者 2。

根据定义，一棵二叉树只有五种可能的形态，如图 6.2 所示（从左到右）：可以是空二叉树；或者只有

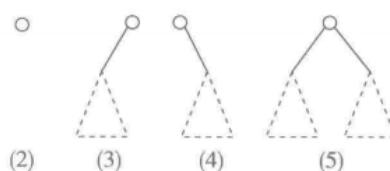


图 6.2 二叉树的 5 种可能形态

根结点，是单点树；或者只有根结点和左子树；或者只有根结点和右子树；或者两棵子树俱全。

路径、结点的层和树的高度

根据祖先结点和子孙结点的定义，从一个祖先结点到其任何子孙结点都存在一系列边，形成从前者到后者的联系。这样一系列首尾相连的边称为树中的一条路径，路径中边的条数称为该路径的长度。易见，从一个结点到其子结点有一条长度为1的路径。为统一起见，也认为从每个结点到其自身有一条长度为0的路径。

显然，从一棵二叉树的根结点到该树中的任一结点都有路径，而且路径唯一。对二叉树的任意子树也有同样结论。

二叉树是一种层次结构。将其树根看作最高层元素，如果有子结点，其子结点看作下一层元素。规定二叉树根的层数为0。对位于 k 层的结点，其子结点是 $k+1$ 层的元素，如此下去，二叉树的所有结点可以按这种关系分为一层层的元素。易见，从树根到树中任一结点的路径长度就是该结点所在的层数，也称为该结点的层数。

一棵二叉树的高度（也称为深度）是树中结点的最大层数（也就是这棵树里的最长路径的长度）。树的高度是二叉树的整体性质。只有根结点的树高度为0。人们一般不讨论空树的高度（在数据结构领域没有定义）。

实际上，上面这些概念的适用范围并不限于二叉树，对后面讨论的一般树结构也适用。

二叉树的性质

二叉树有很多非常有用的性质，现在讨论其中一些性质。作为数据结构，二叉树最重要的性质就是树的高度和树中可以容纳的最大结点个数之间的关系。树的高度类似于表长，是从根结点（首结点）到其他结点的最大距离。在长为 n 的表里只能容纳 n 个结点，而在高为 h 的二叉树中则可能容纳大约 2^h 个结点，这是表与树的最大不同点。

性质 6.1 在非空二叉树第 i 层中至多有 2^i 个结点($i \geq 0$)。

证明：对二叉树的层数做归纳。对 $i=0$ ，第0层至多有一个根结点， $2^0=1$ 。假设第 i 层至多有 2^i 个结点，由于每个结点至多有两个子结点，因此第 $i+1$ 层至多有 $2 \times 2^i = 2^{i+1}$ 个结点。根据数学归纳法，结论成立。

性质 6.2 高度为 h 的二叉树至多有 $2^{h+1}-1$ 个结点($h \geq 0$)。

证明：与性质 6.1 的证明类似。

性质 6.3 对于任何非空二叉树 T ，如果其叶结点的个数 n_0 ，度数为2的结点个数为 n_2 ，那么 $n_0=n_2+1$ 。

证明：可以根据二叉树的5种不同形态，通过结构归纳法证明。由于条件中说明了树非空，这里只需要考虑4种形态的情况。设 T 是二叉树，用 $L(T)$ 表示 T 中叶结点个数， $B(T)$ 表示 T 中度数为2的结点的个数。证明如下。

基础：单点二叉树只有一个结点，它就是叶结点，无度数为2的结点，结论成立。

归纳1：如果树 T 包含根结点 r 且只有左子树 T_1 。根据归纳假设， $L(T_1)=B(T_1)+1$ 。易见，整个二叉树 T 的叶节点和度数为2的结点个数都与 T_1 中一样，即 $L(T)=L(T_1)$ ， $B(T)=B(T_1)$ ，所以 $L(T)=B(T)+1$ 。 T 只有右子树的情况类似，证明从略。

归纳2：如果原树 T 包含根结点 r 和非空左右子树 T_1 和 T_2 ，由归纳假设， $L(T_1)=B(T_1)+1$ ， $L(T_2)=B(T_2)+1$ 。由于 $L(T)=L(T_1)+L(T_2)$ ， $B(T)=B(T_1)+B(T_2)+1$ （加1是因为根结点 r 的度数为2），结论成立。证明完毕。

满二叉树、扩充二叉树

下面介绍两类特殊的二叉树。

满二叉树：如果二叉树中所有分支结点的度数都是 2，则称它为一棵满二叉树。满二叉树是一般二叉树的一个子集。

[例 6.2] 图 6.3 给出了两个满二叉树的实例。右边满二叉树属于一类特例，其中每一层的结点都满，在同样高度的二叉树中结点个数达到了上限。

性质 6.4 满二叉树里的叶结点比分支结点多一个。

由于满二叉树里的分支结点都是度数为 2 的结点，这是性质 6.3 的推论。

扩充二叉树：对二叉树 T ，加入足够多的新叶结点，使 T 的原有结点都变成度数为 2 的分支结点，得到的二叉树称为 T 的扩充二叉树。扩充二叉树中新增的结点称为其外部结点，原树 T 的结点称为其内部结点。空树的扩充二叉树规定为空树。

[例 6.3] 图 6.4 给出的是图 6.1 里的二叉树 T_1 及其扩充二叉树。

从形态看，任何二叉树的扩充二叉树都是满二叉树。根据上面性质 6.4，很显然，扩充二叉树的外部结点的个数比内部结点的个数多 1。

性质 6.5 (扩充二叉树的内部和外部路径长度)

扩充二叉树的外部路径长度 E 是从树根到树中各外部结点的路径长度之和，内部路径长度 I 是从树根到树中各内部结点的路径长度之和。如果该树有 n 个内部结点，那么 $E=I+2\times n$ 。

证明：设原二叉树为 T ，其扩充二叉树为 T' ， T' 的外部路径长度为 E ，内部路径长度为 I 。下面根据 T 的结构做归纳：

基础： T 是空树时 T' 也是空树，结论显然成立。设 T 只有根结点，其扩充二叉树有两个外部结点，内部路径长度 $I=0$ ，外部路径长度 $E=2$ 。这时树中正好有 1 个内部结点，显然 $E=I+2\times n$ 。

归纳 1：假设 T 只有根结点 r 和左子树 T_1 ，但其右子树空。显然 T_1 结点数为 $n-1$ 。设 T_1 的扩充二叉树 T'_1 有 $n-1$ 个内部结点和 n 个外部结点。设 T'_1 的外部路径长度是 E_1 ，内部路径长度是 I_1 ，由归纳假设 $E_1=I_1+2\times(n-1)$ 。考虑 T 的扩充二叉树 T' 。首先，从 T 的根 r 到 T'_1 的任一内部结点的路径长度都比在 T'_1 中增加了 1，增加了 r 到 r 的内部路径长度为 0，所以 $I=I_1+(n-1)$ 。另一方面， r 到 T'_1 的 n 个外部结点的路径有同样情况，还需加上从 r 到其新增右子结点的外部路径长度为 1。所以，

$$\begin{aligned} E &= E_1 + n + 1 = I_1 + 2 \times (n-1) + n + 1 && [\text{根据归纳假设}] \\ &= (I_1 + (n-1)) + 2 \times (n-1) + 2 = I + 2 \times n \end{aligned}$$

归纳 2：假设 T 有根 r ，其左右子树 T_1 和 T_2 都不空。采用上面同样记法并记 T_1 的结点数（也是 T'_1 的内部结点数）为 m ，那么 T_2 的结点数为 $n-m-1$ 。由归纳假设

$$E_1 = I_1 + 2 \times m \quad E_2 = I_2 + 2 \times (n-m-1)$$

设 T 的扩充二叉树是 T' 。根据与上面情况类似的理由，在 T' 里， r 到 T'_1 的内部结点的路径长度之和是 I_1+m ， r 到 T'_1 的外部结点的路径长度之和是 E_1+m+1 ； r 到 T'_2 的内部结点的路径

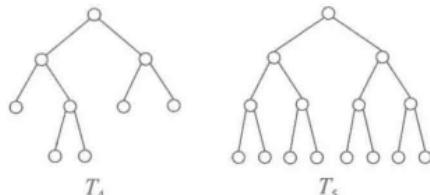


图 6.3 两棵满二叉树

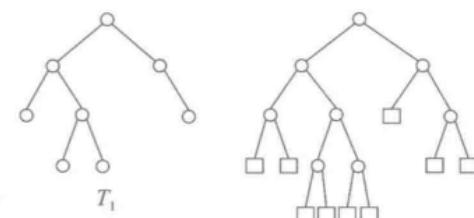


图 6.4 扩充二叉树

长度之和是 $I_2 + (n - m - 1)$, 到其外部结点的路径长度之和是 $E_2 + (n - m)$, 还有 r 到 r 的长度为 0 的内部路径。综上:

$$\begin{aligned} E &= (E_1 + m + 1) + (E_2 + (n - m)) \\ &= (I_1 + 2 \times m + m + 1) + (I_2 + 2 \times (n - m - 1) + (n - m)) \quad [\text{归纳假设}] \\ &= (I_1 + m) + (I_2 + (n - m - 1)) + (2 \times m + 2 \times (n - m - 1) + 2) \\ &= I + 2 \times n \end{aligned}$$

证明完毕。

完全二叉树

对于一棵高度为 h 的二叉树, 如果其第 0 层至第 $h-1$ 层的结点都满 (也就是说, 对所有 $0 \leq i \leq h-1$, 第 i 层有 2^i 个结点)。如果最下一层的结点不满, 则所有结点在最左边连续排列, 空位都在右边。这样的二叉树就是一棵完全二叉树。

【例 6.4】 前面图 6.3 里的二叉树 T_5 就是一棵完全二叉树。当然, T_5 属于完全二叉树中特殊的一类, 其他完全二叉树最下一层的结点不满, 图 6.5 给出了两棵完全二叉树。易见, 在完全二叉树里, 除最下两层外, 其余层都是度数为 2 的分支结点; 而且除了最下最右的分支结点度数可能为 1 外, 其余分支结点的度数均为 2。

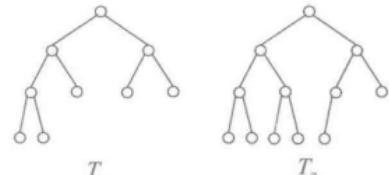


图 6.5 两棵完全二叉树

性质 6.6 n 个结点的完全二叉树高度 $h = \lfloor \log_2 n \rfloor$, 即为不大于 $\log_2 n$ 的最大整数。

证明: 设完全二叉树 T 包含 n 个结点, 高度是 h 。由于 T 在 n 个结点的二叉树里最低, 根据性质 6.2, $2^h - 1 < n \leq 2^{h+1} - 1$, 即 $2^h \leq n < 2^{h+1}$ 。取对数得到 $h \leq \log_2 n < h + 1$ 。可见 h 为不大于 $\log_2 n$ 的最大整数, 得证。

性质 6.7 (完全二叉树) 如果 n 个结点的完全二叉树的结点按层次并按从左到右的顺序从 0 开始编号, 对任一结点 i ($0 \leq i \leq n-1$) 都有:

- 序号为 0 的结点是根。
- 对于 $i > 0$, 其父结点的编号是 $(i-1)/2$ 。
- 若 $2 \times i + 1 < n$, 其左子结点序号为 $2 \times i + 1$, 否则它无左子结点。
- 若 $2 \times i + 2 < n$, 其右子结点序号为 $2 \times i + 2$, 否则它无右子结点。

证明: 根据下标编号、结点所在层等, 对下标做归纳证明。从略。

性质 6.7 是完全二叉树最重要的性质, 使其可以方便地存入一个表或数组, 直接根据元素下标就能找到一个结点的子结点或父结点 (也就是说, 完全确定二叉树的结构), 无须以其他方式记录树结构信息。图 6.6 说明了如何把图 6.5 中的二叉树

T_6 存入一个表, 检查元素下标, 很容易验证性质 6.7 成立。

注意, 根据性质 6.1, 二叉树第 i 层有 2^i 个结点, 根据性质 6.2, 前 $i-1$ 层结点如果全满, 共计有 $2^i - 1$ 个。由于根的下标是 0, 第 i 层元素从下标 $2^i - 1$ 的位置开始存放, 连续 2^i 个元素属于这一层。

这说明从完全二叉树到线性结构有自然的双向映射, 可以方便地从相应线性结构恢复完全二叉树。显然, 一般二叉树没有这种性质。下面很快就会看到这一性质的价值和用途。

一般而言, 对于 n 个结点的二叉树有如下情况 (直观的

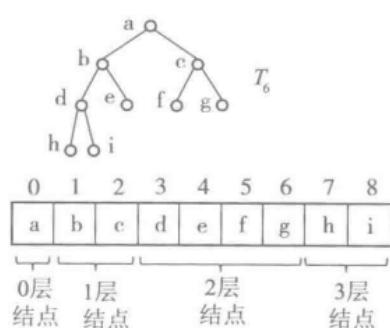


图 6.6 完全二叉树与连续表

看法):

- 如果它足够“丰满整齐”(树中很少度数为1的分支结点,且最长路径的长度差不多),树中最长路径的长度将为 $O(\log n)$ 。例如,完全二叉树都是这样。
- 如果它比较“畸形”,最长路径的长度可能到达 $O(n)$ 。典型情况如图6.1中的 T_3 ,或者图6.7里的几个例子。其中都有些特别长的路径。

也就是说,一般而言, n 个结点的二叉树中最长路径为 $O(n)$ 。可以证明,对于所有的 n 个结点的二叉树,其最长路径的平均值为 $O(\log n)$ 。

下面把二叉树作为一种存储数据元素的汇集数据结构,研究其实现问题,包括二叉树的表示、构造和基本操作等。

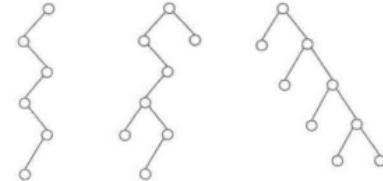


图6.7 三棵比较“畸形”的二叉树

6.1.2 抽象数据类型

首先,结点是二叉树的基础,通常主要用结点保存与应用有关的信息。此外,作为二叉树的表示,还需要记录二叉树的结构信息,至少需要保证能检查结点的父子关系,例如,能从一个结点找到其左/右子结点。

下面是一个基本的二叉树抽象数据类型的定义:

ADT BinTree:	#一个二叉树抽象数据类型
BinTree(self,data,left,right)	#构造操作,创建一个新二叉树
is_empty(self)	#判断self是否为一个空二叉树
num_nodes(self)	#求二叉树的结点个数
data(self)	#获取二叉树根存储的数据
left(self)	#获得二叉树的左子树
right(self)	#获得二叉树的右子树
set_left(self,btree)	#用btree取代原来的左子树
set_right(self,btree)	#用btree取代原来的右子树
traversal(self)	#遍历二叉树中各结点数据的迭代器
forall(self,op)	#对二叉树中的每个结点的数据执行操作op

二叉树的基本操作应该包括创建二叉树。构造一棵二叉树需要基于两棵已有的二叉树和希望保存在树根结点的一项数据。空二叉树的表示是个问题。由于空二叉树没有信息,在实现中可以用某个特殊值表示,例如在Python里用None表示。实际实现也可以另外引进一个专门表示二叉树的结构,把树结点置于其管辖之下。

除了构造函数外,二叉树的其他操作还包括判断空树is_empty;三个访问操作分别访问存储在根结点的数据和左右子树。另外还可以考虑修改左右子树的操作。还需要对二叉树结点(数据)的遍历操作,下面专门讨论这个操作。

6.1.3 遍历二叉树

每棵二叉树有唯一的根结点,可以将其看作这棵二叉树的唯一标识,是基于树结构的处理过程的入口(参看下面的例子)。从根结点出发应该能找到树中所有信息,其基础是从父结点找到两个子结点。因此,实际中常用二叉树的根结点代表这棵二叉树,其左右子树由它们的根结点代表。这种看法在实际表示和算法设计方面都很重要。

二叉树中每个结点可能保存一些数据,因此也是一种汇集型的数据结构。如前所述,对任何汇集结构,都有逐一处理其中所存数据元素的问题,即遍历(周游, traversal)其中元素。遍

历一棵二叉树，就是按某种系统化的方式访问二叉树里的每个结点一次。这一过程可以基于二叉树的基本操作实现，遍历中可能操作结点里的数据。

实际上，很多复杂的二叉树操作需要基于遍历实现。例如找一个结点的父结点，在二叉树里做这件事就像在单链表里找前一结点。

二叉树的结构比较复杂，因此系统化遍历有多种可能的方式，下面将讨论几种不同算法。遍历二叉树就像前面讨论过的状态搜索，以根为起始点，存在两种基本方式：

- 深度优先遍历，顺着一条路径尽可能向前探索，必要时回溯。对于二叉树，最基本的回溯情况是检查完一个叶结点。由于无路可走，只能回头。
- 宽度优先遍历，在所有路径上齐头并进。

深度优先遍历

按深度优先方式遍历一棵二叉树，需要做三件事：遍历左子树、遍历右子树和访问根结点（可能需要操作其中的数据）。下面用 L、R、D 表示这三项工作，参看图 6.8。

选择这三项工作的不同执行顺序，就可以得到三种常见遍历顺序（这里假定了总是先处理左子树，否则就是 6 种）：

- 先根序遍历（按照 DLR 顺序）。
- 中根序遍历（按 LDR），也称对称序。
- 后根序遍历（按 LRD）。

由于二叉树的子树也是二叉树，将一种具体的遍历顺序（方式）继续运用到子树的遍历中，就形成了一种遍历二叉树的统一方法。

在遍历过程中遇到子树为空的情况，就立即结束处理并转去继续做下一步工作。例如，在先根序遍历中遇到左子树为空，就转去遍历相应的右子树。

【例 6.5】 按不同深度优先方式遍历图 6.9 给出的二叉树。

按先根序遍历（先访问根结点，而后以同样方式顺序遍历左子树和右子树），得到的结点访问序列：

A B D H E I C F J K G

按后根序遍历（先以同样方式遍历左右子树，最后访问根结点），得到的序列：

H D I E B J K F G C A

按对称序（中根序，先以同样方式遍历左子树，而后访问根结点，最后再以同样方式遍历右子树）遍历的访问序列：

D H B E I A J F K C G

关于二叉树的遍历有如下概念：

- 按先根序遍历二叉树得到的结点序列称为先根序列。
- 按后根序遍历二叉树得到的结点序列称为后根序列。
- 按对称序遍历二叉树得到的结点序列称为对称序列（中根序列）。

如果二叉树中每个结点有唯一标识，就可以用结点标识描述这些序列（如上例所示）。显然，给定的二叉树唯一确定了它的先根序列、后根序列和对称序列。但给定了一棵二叉树的任一种遍历序列，都无法唯一确定相应的二叉树。

命题 如果知道了一棵二叉树的对称序列，又知道另一遍历序列（无论是先根还是后根序

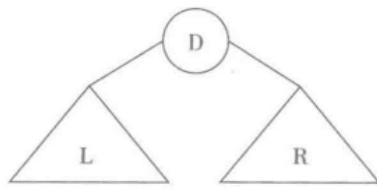


图 6.8 二叉树的三个部分与遍历

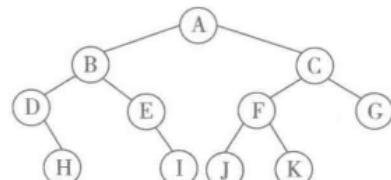


图 6.9 二叉树遍历的实例

列), 就可以唯一确定这个二叉树(请自己考虑和证明)。

宽度优先遍历

现在考虑按宽度优先顺序遍历二叉树。如前所述, 宽度优先是按路径长度由近到远地访问结点。对二叉树做这种遍历, 也就是按二叉树的层次逐层访问树中各结点。与状态空间搜索的情况一样, 这种遍历不能写成一个递归过程。

在宽度优先遍历中只规定了逐层访问, 并没有规定同一层结点的访问顺序。但从算法的角度看, 必须规定一个顺序, 常见的是在每一层里都从左到右逐个访问。实现这一算法需要用一个队列作为缓存。

宽度优先遍历又称为按层次顺序遍历, 这样遍历产生的结点序列称为二叉树的层次序列。对前例中图 6.8 的二叉树, 其层次序列是:

A B C D E F G H I J K

遍历与搜索

请注意, 一棵二叉树可以看作一个状态空间(已在第 5 章讨论): 根结点对应状态空间的初始状态, 父子结点链接对应状态的邻接关系。按这种看法, 一次二叉树遍历也就是一次覆盖整个状态空间的搜索, 前面所有有关状态空间搜索的方法和实现技术都可以原样移植到二叉树遍历问题中。例如, 递归的搜索方法、基于栈的非递归搜索(也即上面讨论的深度优先遍历)。基于队列的宽度优先搜索对应于这里的层次序遍历。

二叉树的特点是一个结点最多有两个子结点。另一方面, 在二叉树遍历中, 从一条路走下去, 绝不会与另一条路相交(树形结构的两个不同分支没有公共结点), 不必考虑循环访问的问题。这些特殊情况都有助于简化算法的实现。

遍历是一种系统化的结点枚举过程, 实际中未必需要检查全部结点, 有时需要在找到所需信息后结束。在二叉树上也可能需要做这种搜索。

最后, 在状态空间的搜索过程中记录从一个状态到另一状态的联系, 将其看作结点间链接, 就会发现这种搜索过程实际上构造出了一棵树, 称为搜索树。当然, 一般而言, 这样形成的结构不是二叉树而是一般的树。但无论如何, 这种情况都进一步说明了树的遍历与状态空间搜索之间的紧密联系。

6.2 二叉树的 list 实现

本节考虑在 Python 里实现二叉树的一种简单技术和它的一个应用。这种技术不仅可以用于实现二叉树, 也可用于实现后面讨论的一般树。

简单看, 二叉树结点也就是一个三元组, 元素是左右子树和本结点数据。Python 的 list 或 tuple 都可以用于组合这样的三个元素, 两者的差异仅在于变动性。如果要实现一种非变动二叉树, 可以用 tuple 作为组合机制, 要实现可以修改结构的二叉树就应该用 list。下面讨论用 list 构造二叉树, 其中所有的基本考虑都适用于 tuple。

6.2.1 设计和实现

二叉树是递归结构, Python 的 list 也是递归结构。基于 list 类型很容易实现二叉树, 例如, 可以采用下面的设计:

- 空树用 None 表示。
- 非空二叉树用包含三个元素的表 $[d, l, r]$ 表示, 其中:

- d 表示存在根结点的元素。

- l 和 r 是两棵子树，采用与整个二叉树同样结构的 list 表示。

显然，这样做把二叉树映射到一种分层的 list 结构，每棵二叉树都有与之对应的（递归结构的）list。例如下面是一棵二叉树的 list 表示：

```
['A', ['B', None, None],
 ['C', ['D', ['F', None, None],
         ['G', None, None]],
  ['E', ['H', None, None],
       ['I', None, None]]]
```

实际上，这种表示也就是一种著名的编程语言 Lisp 里采用的嵌套括号表示形式。在上面描述中，将处于同一层的子树相互对齐，只是为了便于阅读。图 6.10 中画的是与上面 list 表示对应的二叉树的图示，其中把存储在每个结点的字符串标注在结点里面。

相关的二叉树的实现和操作都很简单。下面是实现基本操作的一组函数定义。这些函数的实现只是示意，说明这种二叉树实现可以如何做，并没有写得很完善。例如其中没有特别考虑参数的合法性问题。

```
def BinTree(data, left=None, right=None):
    return [data, left, right]

def is_empty_BinTree(btree):
    return btree is None

def root(btree):
    return btree[0]

def left(btree):
    return btree[1]

def right(btree):
    return btree[2]

def set_root(btree, data):
    btree[0] = data

def set_left(btree, left):
    btree[1] = left

def set_right(btree, right):
    btree[2] = right
```

构造函数 BinTree 为后两个参数提供了默认值，主要是为了使用方便，表示不给左右子树时这两个子树为空。

基于上述构造函数的嵌套调用，可以做出任意复杂的二叉树，例如：

```
t1 = BinTree(2, BinTree(4, BinTree(8)))
```

这相当于写： $t1 = [2, [4, [8, None, None], None], None]$ 。

可以修改二叉树中的任何部分，例如：

```
set_left(left(t1), BinTree(5))
```

其中把 $t1$ 的左子树的左子树换成了 BinTree(5)，使 $t1$ 的值变成：

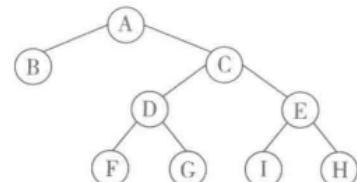


图 6.10 二叉树实例

```
[2, [4, [5, None, None], None], [8, None, None]]
```

这是一棵高度为 2 的二叉树。list 内部的嵌套层数等于树的高度。

基于三元 list (或者三元 tuple) 实现的二叉树，也有后面对二叉树的讨论中提出的各种计算问题。另一方面，list 是 Python 的一种标准类型，能够表示二叉树的是其中的一类特殊形式。这方面的问题前面已经讨论过。如果需要，完全可以以上面这种表示技术为基础，实现一种专门的二叉树类，这一工作留给读者完成，后面几节有关二叉树实现的讨论都可作为参考。

6.2.2 二叉树的简单应用：表达式树

本小节探讨二叉树的一个应用：表达式树。这一应用主要利用二叉树的结构。下面首先分析表达式的结构，它与二叉树的关系，以及基于二叉树的实现技术。最后的表达式实现采用前面简单的二叉树实现技术。

二元表达式和二叉树

数学表达式（算术表达式）具有分层次的递归结构，一个运算符作用于相应运算对象，其运算对象又可以是任意复杂的表达式。二叉树的递归结构正好用来表示这种表达式：二叉树中结点与子树的关系可用于表示运算符对运算对象的作用关系。

下面考虑只包含二元运算符的表达式，称为二元表达式。实际上，这里提出的技术其应用范围远不止二元表达式，完全可用于表示一般的（数学）表达式、逻辑表达式等。算术表达式是二元数学表达式的特殊情况，其中的基本表达式都是数。

二元表达式可以很自然地映射到二叉树（运算符都是二元的）：

- 以基本运算对象（数和变量）作为叶结点中的数据。
- 以运算符作为分支结点的数据：
 - 其两棵子树是它的运算对象。
 - 子树可以是基本运算对象，也可以是任意复杂的二元表达式。

实际上，一元运算符、一元和二元函数也可以纳入上述表示方式，多元函数的问题在后面有简单讨论。各种数学软件都采用了与此类似的表示方式。

【例 6.6】（二元表达式的遍历序列）易见，一个结构正确的二元表达式对应于一棵满二叉树，例如图 6.11 中的二元表达式。现在考虑对这一表达式树的先根、后根和中根序遍历得到的符号序列。

先根序遍历得到“ $\times - a b + / c d e$ ”，正是该表达式的前缀表示。

后根序遍历得到的序列是“ $a b - c d / e + \times$ ”，正是该表达式的后缀表示形式（即其波兰表示法的形式）。

对称序遍历得到“ $a - b \times c / d + e$ ”，基本上是相应数学表达式的中缀表示，只是缺少表示计算顺序的括号（因此未能表达正确的计算顺序）。

构造表达式

由于建立起来的数学表达式绝不会变化，数学运算和操作都是基于已有表达式构造新表达式。因此，一种合理方式是把它实现为一种“不变的”二叉树，下面用 Python 的 tuple 作为实现基础，用三元的 tuple 实现二叉树结点。

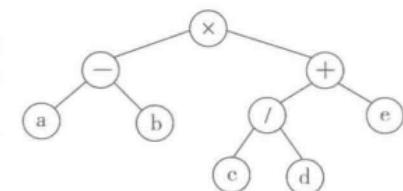


图 6.11 二叉树遍历的实例

为使有关的表示更简洁清晰，对上面提出的二叉树的表示做一点修改。在二元表达式对应的 tuple 里，基本运算对象（数或变量）将直接放在空树的位置，作为基本对象。例如，按前面的设计，表达式“ $3 * (2 + 5)$ ”直接映射到二叉树是

```
('*', (3, None, None),
 ('+', (2, None, None), (5, None, None)))
```

这样表示总会出现大量没有实际意义的 None。为了避免这种情况，下面将其简化表示为（带括号的前缀表达式，括号表示运算符的作用范围）：

```
('*', 3, ('+', 2, 5))
```

采用这种修改的表达方式，表达式由两种结构组成：

- 如果是序对（tuple），就是运算符作用于运算对象的复合表达式。
- 否则就是基本表达式，也就是数或者变量。

上述两条可用于解析表达式的结构，实现对表达式的处理。

下面定义几个表达式构造函数：

```
def make_sum(a, b):
    return ('+', a, b)

def make_prod(a, b):
    return ('*', a, b)

def make_diff(a, b):
    return ('-', a, b)

def make_div(a, b):
    return ('/', a, b)

# 其他构造函数与此类似，略
```

下面语句构造出一个简单的算术表达式：

```
e1 = make_prod(3, make_sum(2, 5))
```

显然，采用这种结构可以构造出具有任意复杂结构的表达式。

用字符串表示变量，就能构造出各种代数表达式，例如

```
make_sum(make_prod('a', 2), make_prod('b', 7))
```

在定义表达式处理函数时，经常需要区分基本表达式（直接处理）和复合表达式（递归处理）。为分辨这两种情况，定义一个判别是否为基本表达式的函数：

```
def is_basic_exp(a):
    return not isinstance(a, tuple)
```

为简单起见，这里认为只有 int、float、complex 三个具体数值类型的对象，判断是否数值的函数可以用下面定义：

```
def is_number(x):
    return (isinstance(x, int) or isinstance(x, float)
           isinstance(x, complex))
```

表达式求值

前面定义的二元表达式是简单数学表达式的一种 Python 实现。在这些定义的基础上，可

以根据需要实现各种表达式操作。例如，可以定义 Python 函数求两个二元表达式的和、乘积等。作为例子，这里考虑一个求表达式值的函数。

根据数学，有下面表达式求值规则：

- 对表达式里的数和变量，其值就是它们自身。
- 其他表达式根据运算符的情况处理，可以定义专门处理函数。
- 如一个运算符的两个运算对象都是数，就可以求出一个数值。

还有些情况，如加数是 0，乘数是 0 或 1，可对有关的部分表达式做求值和化简。总而言之，所谓求值，就是对表达式做一些简单化简，把能计算的都算出来。

下面是求值函数的基本部分：

```
def eval_exp(e):
    if is_basic_exp(e):
        return e
    op, a, b = e[0], eval_exp(e[1]), eval_exp(e[2]) #**
    if op == '+':
        return eval_sum(a, b)
    elif op == '-':
        return eval_diff(a, b)
    elif op == '**':
        return eval_prod(a, b)
    elif op == '/':
        return eval_div(a, b)
    else:
        raise ValueError("Unknown operator:", op)
```

这里的基本方法就是基于实际运算符，把不同计算分发给具体函数处理。其中特别值得注意的是标着“**”的语句，其中通过对本函数的两个递归调用处理子表达式，完成可能的计算。后面可以看到，对二叉树的所有递归处理都将采用这种模式。

在最后一行引发异常时，给异常 ValueError 提供了两个实参。实际上，在引发异常时可以送给它任意多个实参。如果没有其他处理，在 Python 解释器显示异常信息时，将输出这些实际参数的值（例如，对上面情况，它输出字符串 Unknown operator 和 op 的实际值）。如果希望捕捉并处理这种异常，可以通过 Python 提供的特殊机制取得这些实参的值。有关情况参看 Python 的语言和标准库手册。

下面是对和式和除式求值的函数，其他函数类似（从略）：

```
def eval_sum(a, b):
    if is_number(a) and is_number(b):
        return a + b
    if is_number(a) and a == 0:
        return b
    if is_number(b) and b == 0:
        return a
    return make_sum(a, b)

def eval_div(a, b):
    if is_number(a) and is_number(b):
        return a / b
    if is_number(a) and a == 0:
        return 0
    if is_number(b) and b == 1:
        return a
    if is_number(b) and b == 0:
```

```

raise ZeroDivisionError
return make_div(a, b)

```

读者不难完成这个求值器（见本章后面的练习）。

扩充

上面简单表达式有很大的扩充空间。首先是可以定义更多的表达式操作，例如：

- 定义以比较好读的形式输出这种表达式（请读者自己考虑，后面有讨论）。
- 二元表达式的各种运算（其中化简是很困难的工作，除了简单化简）。
- 高级的数学操作。例如，针对某个变量求导代数表达式的函数（得到另一代数表达式，比较容易），求表达式的不定积分（不容易做好）。

显然，上面的表达式定义形式也可以扩充。由于 tuple 可以有任意多个成员，因此可用于表示一元或多元函数的作用，例如，允许写

```
['+', 2, 4, ['*', 15, ['sin', 2.3], 'x']]
```

请读者自己考虑如何扩充，如何修改前面的定义。

沿着这个方向继续工作，可以在 Python 语言里实现一个完成复杂表达式计算的系统，其功能类似于使用广泛的数学软件 Maple 或 Mathematica。

6.3 优先队列

本节将讨论另一种重要缓存结构：优先队列。从原理上说，这种结构与二叉树没有直接关系。但是基于对一类二叉树的认识，可以做出优先队列的一种高效实现。因此，本节的内容也可以看作二叉树的应用。

6.3.1 概念

首先介绍优先队列的概念。

作为缓存结构，优先队列与栈和队列类似，可以将数据元素保存其中，可以访问和弹出。优先队列的特点是存入其中的每项数据都另外附有一个数值，表示这个项的优先程度，称为其优先级。优先队列应该保证，在任何时候访问或弹出的，总是当时在这个结构里保存的所有元素中优先级最高的。如果该元素不弹出，再次访问还将得到它。在一些情景中，可能出现不同元素具有相同优先级的情况。如果不止一个元素的优先级最高，优先队列将保证访问或弹出的是它们中的一个，具体是哪个元素由内部实现确定。

抽象地看，需要缓存的是一个有序集 $S = (D, \leq)$ 的元素，这里的“ \leq ”是集合 D 上的一个全序（非严格的），表示元素的优先关系。优先队列要求保证“最优先元素先出”（或先用）。具体集合和序由实际需要确定，同一集合上也可能有不同的序，在实现优先队列时，必须确定使用的具体序关系。

允许 D 中不同元素具有相同优先级的情况，也就是说，可能有不同的 $a, b \in D$ ，对它们有 $a \leq b$ 且 $b \leq a$ 。这时就说 a 和 b 优先级相同。在这种应用情景中，如果要求保证优先级相同的元素先进先出（希望优先队列同时具有队列的 FIFO 性质），那就只能做出效率较低的实现。如果只要求保证访问（或弹出）的总是当时存在的最优先元素中的一个，不要求一定是其中最早进入优先队列的元素，那么就存在效率更高的实现。

优先关系可以代表数据的某种性质，例如用于描述实际中：

- 各项工作的计划开始时间（实际和模拟中都可能使用）。

- 一个大项目中各种工作任务的急迫程度（或截止期）。
- 银行客户的诚信度评估（用于决定优先贷款）。

显然，这类需求在实际中无穷无尽。

优先队列的操作也很简单，应包括：

- 创建，判断空（还可以有清空内容、确定当前元素个数等）。
- 插入元素，访问和删除优先队列里（当时最优先）的元素。

下面考虑优先队列的实现问题。

6.3.2 基于线性表的实现

首先考虑一种简单方法：基于连续表技术实现优先队列。

有关实现方法的考虑

由于连续表可以存储数据元素，显然有可能作为优先队列的实现基础。数据项在连续表里的存储顺序可用于表示数据之间的某种顺序关系。对于优先队列，这个顺序可用于表示优先级关系，例如，让数据的存储位置按优先顺序排列。

但是，从使用的角度看，用户只关心优先队列的使用特性。按优先级顺序存储数据项是一种可能，但并不必需。不难想到实际上存在着两种可能的实现方案：

- 1) 在存入数据时，保证表中元素始终按优先顺序排列（作为一种数据不变式），任何时候都可以直接取到当时在表里最优先的元素。采用有组织的元素存放方式，存入元素的操作比较麻烦，效率可能较低，但访问和弹出时比较方便。
- 2) 存入数据时采用最简单的方式（例如，对顺序表存入表尾端，对链接表存入表头），需要取用时，通过检索找到最优先的元素。采用这种无组织的方式存放元素，把选择最优元素的工作推迟到访问 / 取出时，存入操作的效率高但取用麻烦。如果需要多次访问同一个元素但不弹出，就应该采用其他技术，避免重复检索。当然，也可以在一次检索后记录最优先元素，再次使用时就可以直接使用了。这种技术还需要与元素弹出相互配合，有关实现中的问题，请读者考虑。

经过比较和分析（请读者自己分析其合理性），下面准备采用第一种技术：在加入新数据时，设法确定正确的插入位置，保证表元素始终按优先顺序排列。另一方面，由于下面考虑采用连续表实现，为保证访问和弹出最优先数据项的操作能在 $O(1)$ 时间内完成，最优先的项应该出现在表的尾端。

基于 `list` 实现优先队列

现在考虑基于 Python 的实现问题。采用连续表技术，应该用一个 `list` 对象存储数据。`list` 对象能根据存储元素的实际需要自动扩大存储区，但也应该注意：任何时候都只能在合法范围内使用下标表达式，超范围赋值或取值是运行错误，会引发 `IndexError` 异常。因此，在需要插入新元素时，只能先设法确定正确插入位置，而后调用 `list` 的 `insert`（或其他操作）做定位插入。

在下面实现中，假定需要存储的数据元素用“`<=`”比较优先级，值较小的元素优先级更高。这一点不难根据需要修改。

首先定义一个异常类（类体为 `pass`），在优先队列类里使用：

```
class PrioQueueError(ValueError):
    pass
```

将优先队列定义为一个类：

```
class PrioQue:
    def __init__(self, elist=[]):
        self._elems = list(elist)
        self._elems.sort(reverse=True)
```

引进参数使人可以提供一组初始元素。注意这里的参数默认值，以可变对象作为默认值是一种危险动作，应特别注意。在这里用 `list` 转换，有两个作用：首先是对实参表（包括默认值空表）做一个拷贝，避免共享。此外，这样也使构造函数的实参可以是任何可迭代对象，例如迭代器或者元组等。最后调用 `list` 的 `sort` 方法，其中的 `reverse` 参数要求做从大到小的排序。也就是说，这里以较小作为较优先。改用较大没有任何困难。

插入元素是这里最复杂的操作，需要找到正确的插入位置：

```
def enqueue(self, e):
    i = len(self._elems) - 1
    while i >= 0:
        if self._elems[i] <= e:
            i -= 1
        else:
            break
    self._elems.insert(i+1, e)
```

`while` 循环从最后的元素开始检查，结束时 `i` 或为 `-1`，或是第一个大于 `e` 的元素的下标，最后的定位插入总是正确的。`while` 的条件还保证了优先度相同元素的正确排列顺序，使同优先级的元素能先进先出。

另外几个方法都比较简单：

```
def is_empty(self):
    return not self._elems

def peek(self):
    if self.is_empty():
        raise PrioQueueError("in top")
    return self._elems[-1]

def dequeue(self):
    if self.is_empty():
        raise PrioQueueError("in pop")
    return self._elems.pop()
```

对连续表实现的分析

现在分析这一实现中各操作的效率，各操作的效率（复杂度）都很清晰：插入元素是 $O(n)$ 操作，其他都是 $O(1)$ 操作。注意，即使插入时表的存储区满，需要换一块存储，操作效率的复杂度量级也没有变，仍然是 $O(n)$ 。

前面分析中还提出了另一种实现技术方案。不难看到，如果采用该方案，插入元素的操作具有 $O(1)$ 的平均复杂度（替换表存储时需要 $O(n)$ 时间）。另一方面，检查和弹出队列中的最优先元素都是 $O(n)$ 操作。

这里的讨论只研究了采用连续表技术的优先队列实现，实际上，也可以用链接表技术实现优先队列，几个主要操作的复杂度情况与连续表类似。总而言之，采用线性表技术实现优先队列，无论采用怎样的具体实现技术，在插入元素与取出元素的操作中总有一种是具有线性复杂度的操作，这一情况不能令人满意。

6.3.3 树形结构和堆

下面考虑改善优先队列的操作性能的可能性。

线性和树形结构

首先分析效率低的原因。采用前面讨论的实现，按序插入操作低效，其根源就是需要沿着表顺序检索插入位置。表长度是 n ，检索（和最终插入）必然需要 $O(n)$ 时间。这说明，只要元素按优先级顺序线性排列，就无法避免线性复杂性问题：对于顺序表，需要移动 $O(n)$ 个元素；对链接表，需要顺着链接爬行 $O(n)$ 步。这些情况意味着，如果不改变数据的线性顺序存储方式，就不可能突破 $O(n)$ 的复杂度限制。要做出操作效率更高的优先队列，必须考虑其他数据结构组织方式。

一般而言，确定最优先元素并不需要与所有其他元素比较。以体育比赛中的淘汰赛为例，假设有 n 名选手参加，首先需要进行 $n-1$ 场比赛确定冠军，每个选手只需进行约 $\log_2 n$ 场比赛。决出冠军后，要确定真正的第二名，只需亚军与所有输给冠军的人比赛，只需要沿着冠军胜利的路线比赛，不超过 $\log_2 n$ 次。

上面情况说明，利用树形结构的祖先 / 子孙序，有可能得到更好的操作效率。但要想在优先队列的实现中利用树形结构的优势，还需要解决一个问题：如何在反复插入和删除元素的过程中，持续保持树形结构的特点和操作效率。

堆及其性质

采用树形结构实现优先队列的一种有效技术称为堆。从结构上看，堆就是结点里存储数据的完全二叉树，但堆中数据的存储要满足一种特殊的堆序：任一个结点里所存的数据（按所考虑的序）先于或等于其子结点（如果存在）里的数据。

根据堆的定义，不难看到：

- 在一个堆中从树根到任何一个叶结点的路径上，各结点里所存的数据按规定的优先关系（非严格）递减。
- 堆中最优先的元素必定位于二叉树的根结点里（堆顶）， $O(1)$ 时间就能得到。
- 位于树中不同路径上的元素，这里不关心其顺序关系。

如果所要求的序是小元素优先，构造出来的堆就称为小顶堆（小元素在上），堆中每个结点的数据均小于或等于其子结点的数据。如果要求大元素优先，做出的堆称为大顶堆，每个结点里的数据都大于或等于其子结点的数据，堆顶是最大元素。

图 6.12 形象地描绘了一个堆的形状（也就是完全二叉树的形状），以及堆中一条路径的情况。除最下一层右边可能有所欠缺，堆里各层结点全满。图中从根到叶的路径上越小的圆圈越接近根，表示一种序关系。

前面讲过：一棵完全二叉树可以自然而且信息完全地存入一个连续的线性结构（例如连续表，参看图 6.6），因此，一个堆也可以自然地存入一个连续表，通过下标就能方便地找到树中任一结点的父结点 / 子结点。

在下面讨论中说到堆时，经常是指存储在连续表里的一棵完全二叉树，其元素的存储情况形成了一个堆。由于这种结构与堆一一对应，可以不区分这两个概念。

堆和完全二叉树还有下面几个重要的性质（参考图 6.13）：

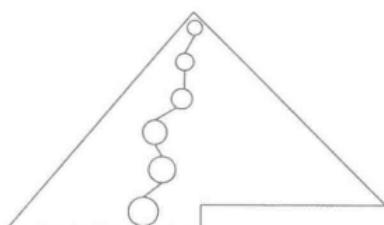


图 6.12 堆和堆中的路径

Q1：在一个堆的最后加上一个元素（在相应连续表的最后加一个元素），整个结构还是可以看作一棵完全二叉树，但它未必是堆（最后元素未必满足堆序）。

Q2：一个堆去掉堆顶（表中位置0的元素），其余元素形成两个“子堆”，完全二叉树的子结点/父结点下标计算规则仍然适用，堆序在路径上仍然成立。

Q3：给由Q2得到的表（两个子堆）加入一个根元素（存入位置0），得到的结点序列又可看作完全二叉树，但它未必是堆（根结点未必满足堆序）。

Q4：去掉一个堆中最后的元素（最下层的最右结点，也就是对应的连续表里的最后一个元素），剩下的元素仍构成一个堆。

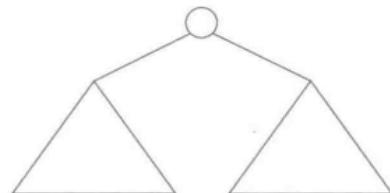


图 6.13 完全二叉树的根和子树

堆与优先队列

现在考虑如何基于堆的概念和结构实现一种优先队列结构。

首先，用堆作为优先队列，可以直接得到堆中的最优先元素， $O(1)$ 时间。但要实现优先队列，还需要解决两个问题：

- 如何实现插入元素的操作：向堆（优先队列）中加入了一个新元素，必须能通过操作，得到一个包含了所有原有元素和刚加入的新元素的堆。
- 如何实现弹出元素的操作：从堆中弹出最小元素后，必须能把剩余元素重新做成堆。

下面将看到，这两个操作均可以在 $O(\log n)$ 时间内完成。其他操作都很简单，都是 $O(1)$ 时间操作，后面的代码可以参考，不再专门讨论。

6.3.4 优先队列的堆实现

解决堆插入和删除的关键操作称为筛选，又分为向上筛选和向下筛选。

插入元素和向上筛选

首先考虑向堆中加入元素的操作。根据性质Q1，在一个堆的最后加入一个元素，得到的结果还可以看作完全二叉树，但未必是堆。为把这样的完全二叉树恢复为堆，只需做一次向上筛选，如图6.14所示。

向上筛选的方法：不断用新加入的元素（设是 e ）与其父结点的数据比较，如果 e 较小就交换两个元素的位置。通过这样的比较和交换，元素 e 不断上移。这一操作一直做到 e 的父结点的数据小于等于 e 时，或者 e 已经到达根结点时停止。这时经过 e 的所有路径上的元素满足所需顺序，其余路径仍保持有序，因此这棵完全二叉树满足堆序，整个结构已恢复为一个堆。这样就得到了在基于堆的优先队列的插入操作：

- 把新加入元素放在（连续表里）已有元素之后，执行一次向上筛选操作。
- 向上筛选操作中比较和交换的次数不会超过二叉树中最长路径的长度。根据完全二叉树性质，加入元素操作可以在 $O(\log n)$ 时间完成。

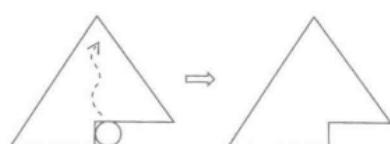


图 6.14 插入元素和向上筛选

弹出元素和向下筛选

由于堆顶元素就是最优先元素，应该弹出的元素就是它。但弹出堆顶元素后，剩下的元素已经不再是堆。但根据性质Q2，剩余元素可以看作两个“子堆”。根据Q3只需填补一个堆

顶元素就可以将它们做成一棵完全二叉树。根据 Q4，从原堆的最后取下一个元素，其余元素仍然是堆。把这个元素放在堆顶就得到了一棵完全二叉树，它比弹出操作前的堆少一个元素，而且除了堆顶元素可能不满足堆序外，其余元素都满足堆序。此时的状态如图 6.15 左图所示，下面需要设法把结构重新恢复为一个堆。

在这种情况下恢复堆的操作称为向下筛选：设两个子堆 A 和 B 加上根元素 e 构成一棵完全二叉树，现在需要把它们做成一个堆，操作步骤是：

- 用 e 与 A、B 两个“子堆”的顶元素（根）比较，最小者作为整个堆的顶。
- 若 e 不是最小，最小的必为 A 或 B 的根。设 A 的根最小，将其移到堆顶，相当于删去了 A 的顶元素。
- 下面考虑把 e 放入去掉堆顶的 A，这是规模更小同一问题。
- B 的根最小的情况可以同样处理。
- 如果某次比较中 e 最小，以它为顶的局部树已经成为堆，整个结构也成为堆。
- 或者 e 已经落到底，这时它自身就是一个堆，整个结构也成为堆。

到达最后两种情况，重新构造堆的工作就完成了。

总结一下优先队列弹出操作的实现，分为三个大步骤：

- 弹出当时的堆顶。
- 从堆最后取一个元素作为完全二叉树的根。
- 执行一次向下筛选。

前两步都是 $O(1)$ 时间操作。最后一步需要从完全二叉树的根开始做，一步操作需要做两次比较，操作次数不长于树中路径的长度。根据完全二叉树的性质，从这种优先队列里弹出元素的操作具有 $O(\log n)$ 时间复杂度。

基于堆的优先队列类

现在基于前面的认识定义一个优先队列类。在这个类的对象里，还是用一个 list 存储元素，要考虑 list 的使用方式。不难看出，应该在表尾端加入元素，以首端作为堆顶，与前面用排序表实现的情况相反[⊖]。

下面是该类的构造函数和两个简单方法：

```
class PrioQueue:
    """ Implementing priority queues using heaps """
    def __init__(self, elist=[]):
        self._elems = list(elist)
        if elist:
            self.buildheap()

    def is_empty(self):
        return not self._elems
    def peek(self):
        if self.is_empty():
            raise PrioQueueError("in peek")
        return self._elems[0]
```

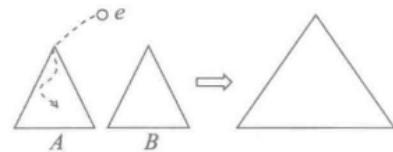


图 6.15 删除元素和向下筛选

[⊖] 请分析交换表两端的安排会遇到怎样的情况。

在这里只有构造函数还需要做一点解释。这里为构造函数提供了一个表参数，使人可以为优先队列提供一批初始元素。与前面的排序表实现一样，list(elist) 从 elist 出发做出一个表拷贝有几个意义：做拷贝使内部的表脱离原来的表，排除共享；对默认情况也建立一个新的空表，避免了以可变对象作为默认值的 Python 编程陷阱；允许以任何可迭代对象作为参数，扩大了这个构造函数的适用范围。显然参数表中的元素不一定满足堆序，buildheap 方法将其转变为堆，具体实现后面讨论。

现在考虑入队操作。对于前面已经详细讨论的完成操作的技术，这里给出的就是有关技术的一个实现，主要工作由 siftup 完成：

```
def enqueue(self, e):
    self._elems.append(None)  # add a dummy element
    self.siftup(e, len(self._elems)-1)

def siftup(self, e, last):
    elems, i, j = self._elems, last, (last-1)//2
    while i > 0 and e < elems[j]:
        elems[i] = elems[j]
        i, j = j, (j-1)//2
    elems[i] = e
```

注意：在 siftup 的实现里，并没有先存入元素后再考虑交换，而是“拿着它”去查找正确插入的位置。循环条件保证跳过的元素都是优先度较低的元素，在检查过程中将它们逐个下移。循环结束时确定的 i 就是应该存入元素的位置。

弹出元素的操作稍微复杂一点。取出堆顶元素 e0 后弹出最后元素，然后做一次向下筛选恢复堆中的元素顺序，最后返回 e0。

```
def dequeue(self):
    if self.is_empty():
        raise PrioQueueError("in dequeue")
    elems = self._elems
    e0 = elems[0]
    e = elems.pop()
    if len(elems) > 0:
        self.siftdown(e, 0, len(elems))
    return e0

def siftdown(self, e, begin, end):
    elems, i, j = self._elems, begin, begin*2+1
    while j < end:  # invariant: j == 2*i+1
        if j+1 < end and elems[j+1] < elems[j]:
            j += 1  # elems[j] 不大于其兄弟结点的数据
        if e < elems[j]:  # e 在三者中最小，已找到了位置
            break
        elems[i] = elems[j]  # elems[j] 在三者中最小，上移
        i, j = j, 2*j+1
    elems[i] = e
```

与 siftup 类似，在 siftdown 的实现里也是拿着新元素找位置，没有采用先存入元素再逐步交换的方法，可以稍微节省一点开销。

最后考虑堆的初始构建，这里要基于一个已有的 list 建立初始堆。下面的做法基于如下事实：一个元素的序列已经是堆；如果元素位置合适，在表里已有的两个“子堆”上加一个元素，通过一次向下筛选，就可以把这部分元素调整为一个更大的子堆。

把初始的表看作一棵完全二叉树，从下标 end//2 的位置开始，后面表元素都是二叉树

的叶结点，也就是说，它们中的每一个已是一个堆。从这里开始向前做，也就是从完全二叉树的最下最右分支结点开始，向左一个个建堆，然后再到上一层建堆，直至整个表建成一个堆。

如图 6.16 所示，对于这里的每一步，要做的工作就是在两个已有的堆上加一个根元素，将它们形成的完全二叉树调整为一个堆。下面函数里的循环描述了这部分工作：

```
def buildheap(self):
    end = len(self._elems)
    for i in range(end//2, -1, -1):
        self.siftdown(self._elems[i], i, end)
```

构建操作的复杂性

前面已经分析了入队和弹出操作的复杂性，其他操作都是 $O(1)$ 操作。现在考虑上面堆构建函数的复杂性。仅有的问题是其中的循环：

```
for i in range(end//2, -1, -1):
    siftdown(self._elems[i], i, end)
```

设被处理的完全二叉树中有 n 个元素，高度为 h 。在这棵树里：

- 高度为 2 的子树共计大约 $n/4$ ($n/2^2$) 棵。把每一棵这样的子树调整为一个堆，根元素移动的距离不超过 1。
- 高度为 3 的子树共计大约 $n/8$ ($n/2^3$) 棵，把每一棵这样的子树调整为一个堆，根元素移动的距离不超过 2。
-

对这些操作的移动次数求和：

$$\begin{aligned} C_1(n) &\leq \sum_{i=0}^{h-1} (h-i)2^{i+1} = \sum_{j=1}^h j \cdot 2^{h-j+1} \quad (j = h-i) \\ &= \sum_{j=1}^h 2 \cdot j \cdot 2^{-j} \cdot 2^h \leq 2n \sum_{j=1}^h j/2^j \quad \left(\sum_{j=1}^h j/2^j \leq 2 \right) \\ &\leq 4n = O(n) \end{aligned}$$

由此可得到结论：堆构建操作的复杂性是 $O(n)$ 。

总结一下：基于堆的概念实现优先队列，创建操作的时间复杂度是 $O(n)$ ，这件事只需做一次。插入和弹出操作的复杂度是 $O(\log n)$ ，效率比较高。插入操作的第一步是在表的最后加入一个元素，可能导致 `list` 对象替换元素存储区，因此可能出现 $O(n)$ 的最坏情况。但这也保证了堆不会因为满而导致操作失败。另外，所有操作中都只用了一个简单变量，没用其他结构，所以它们的空间复杂度都是 $O(1)$ 。

6.3.5 堆的应用：堆排序

如果在一个连续表里存储的数据是一个小顶堆（元素之间的关系满足堆序），按优先队列的操作方式反复弹出堆顶元素，能够得到一个递增序列。不难想到，这就形成了一种可行方法，可以用于完成对连续表中元素的排序工作。

基于这种技术完成排序工作，还需要解决两个问题：

- 连续表里的初始元素序列通常不满足堆序。这个问题前面已经讨论了，就是上一小节

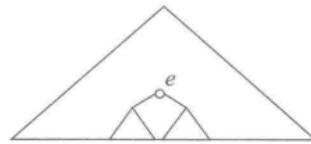


图 6.16 构建初始堆过程中的一步

里优先队列里的初始建堆工作。

- 选出的元素存放在哪里？能不能不用其他空间？

实际上，第二个问题也很好解决：随着元素弹出，堆中元素也越来越少。每弹出一个元素，表的后部就会空出一个位置，正好用于存放弹出的元素。但如果直接这样存放，采用小顶堆工作时，最后排出的序列是从左到右越来越小。如果希望元素从小到大排序，可改程序采用大顶堆工作，或者排序函数退出前反转表中元素（需要 $O(n)$ 时间）。这些修改都留给读者完成。下面只是基于前面讨论给出一个函数定义：

```
def heap_sort(elems):
    def siftdown(elems, e, begin, end):
        i, j = begin, begin*2+1
        while j < end:           # invariant: j == 2*i+1
            if j+1 < end and elems[j+1] < elems[j]:
                j += 1             # elems[j] 小于等于其兄弟结点的数据
            if e < elems[j]:       # e在三者中最小
                break
            elems[i] = elems[j]   # elems[j] 最小, 上移
            i, j = j, 2*j+1
            elems[i] = e

        end = len(elems)
        for i in range(end//2, -1, -1):
            siftdown(elems, elems[i], i, end)
        for i in range((end-1), 0, -1):
            e = elems[i]
            elems[i] = elems[0]
            siftdown(elems, e, 0, i)
```

在这个函数里重新定义了 `siftdown` 操作，作为主函数的内部函数。主函数体主要是两个循环：第一个循环建堆，从位置 i 开始，以 end 为建堆范围的边界；第二个循环逐个取出最小元素，将其积累在表的最后，放一个退一步。

函数的复杂度很容易分析：初始建堆为 $O(n)$ 时间，第二个循环做 n 次，每次取出一个元素后放好新顶元素并做一次向下筛选，新堆顶下行的距离不超过 $\log n$ 。显然第二个循环的总开销为 $O(n \log n)$ 。函数里只用了几个局部变量，空间复杂度是 $O(1)$ 。

有关排序问题，第 9 章有专门讨论。那里要介绍一些与排序有关的概念，还要把这个堆排序算法和其他算法放在一起分析和比较。

6.4 应用：离散事件模拟

现在考虑第 5 章提过的一类重要计算机应用：离散事件系统模拟。如前所述，被模拟系统的行为可以抽象为一些离散事件的发生，所发生事件可以引发新的事件等，人们希望通过计算机模拟理解系统行为，评价或设计真实世界中实际的或所需的系统。

适用于这种模拟的系统，其行为特征是：

- 系统运行中可能不断发生一些事件（带有一定的随机性）。
- 一个事件在某个时刻发生，其发生有可能导致其他事件在未来发生。

这类系统就可以用下面讨论的方式模拟。

作为例子，现在考虑一个负责检查过境车辆的海关检查站：

- 车辆按一定时间间隔到达，间隔有一定随机性，设其范围为 $[a, b]$ 分钟。
- 由于车辆的不同情况，每辆车的检查时间为 $[c, d]$ 分钟。

- 海关可以开 k 个通道。
- 希望理解开通不同数量的通道数对车辆通行的影响。

这样的系统就比较适合采用离散事件模拟的方式做试验。例如，在考虑建设这个海关时，通过模拟海关的运行情况，分析需要建几条检查通道等。目标是提供较好的服务，保证合理的车辆等待时间和合理的建设成本（不出现很多空闲通道）。

模拟中的事件经常需要排队，队列结构适合用于记录事件。很多情况下还牵涉到时间或其他排序因素，优先队列结构可能提供所需要的功能。从下面实例可以看到这两类结构的应用，其中还介绍了面向对象设计的一些想法。

6.4.1 通用的模拟框架

做这种模拟，其中的基本想法就是按事件发生的时间顺序处理。在模拟系统里用一个优先队列保存已知在将来某些特定时刻发生的事件，系统的运行就是不断从优先队列里取出等待事件，一个个处理，直至整个模拟结束。

事件的具体处理（运行）由具体的模拟问题确定。在一些事件的处理中可能引发另一个或一些新的（将在以后的某个时刻发生的）事件，这些事件应该放入优先队列，在它们应该发生的时刻运行（被系统处理）。在模拟过程进行中，系统中始终维护着一个当前时间，也就是当时正在发生那个事件的时间。

这里的计划是首先开发一个通用的模拟框架类，通过它实现上述基本过程，而后基于这个基础类，开发一个具体的模拟实例系统。

这里是根据上面的分析实现的一个通用模拟器类：

```
from random import randint
from prioqueue import PrioQueue
from queue_list import SQueue

class Simulation:
    def __init__(self, duration):
        self._eventq = PrioQueue()
        self._time = 0
        self._duration = duration

    def run(self):
        while not self._eventq.is_empty():          # 模拟到事件队列空
            event = self._eventq.dequeue()
            self._time = event.time()               # 事件的时间就是当前时间
            if self._time > self._duration:         # 时间用完就结束
                break
            event.run()                            # 模拟这个事件，其运行可能生成新事件

    def add_event(self, event):
        self._eventq.enqueue(event)

    def cur_time(self):
        return self._time
```

这个模拟系统用到了几个模块，包括前面定义的模块和标准库的 `random` 包。

`Simulation` 类的对象（模拟器对象）实现模拟过程，其中用了一个优先队列记录模拟中缓存的事件，称为事件队列。属性 `_time` 记录当前时间，`_duration` 记录模拟的总时长。`Simulation` 类定义了两个主要方法和两个简单辅助方法。其中最主要的方法是 `run`，它实

现一次完整模拟：只要事件队列不空，就从中取下（按时间排列的）第一个事件，用该事件的时间设置系统的当前时间，然后令该事件运行。

具体事件的行为由其 run 方法确定，在该方法的运行中可能生成新的事件。这里采用了一种典型的模拟技术，让实际被模拟系统的各种事件自己去推动整个模拟的进行。在这种模拟系统里，每个事件是一个对象，有其特定的具体行为，由相应的事件类定义。这种模拟技术称为面向对象的离散事件模拟。

对于任何一个具体的模拟，只需要根据情况实现一组特殊的事件（类）。为了规范所有事件类的形式，这里先定义一个公共事件基类，在其中实现几个所有事件都需要的公共操作。最后还定义了一个什么都不做的 run 方法：

```
class Event:  
    def __init__(self, event_time, host):  
        self._ctime = event_time  
        self._host = host  
  
    def __lt__(self, other_event):  
        return self._ctime < other_event._ctime  
  
    def __le__(self, other_event):  
        return self._ctime <= other_event._ctime  
  
    def host(self):  
        return self._host  
  
    def time(self):  
        return self._ctime  
  
    def run(self): # 具体事件类必须定义这个方法  
        pass
```

请注意构造函数的 host 参数，它表示有关事件的发生所在的模拟系统（称为宿主系统）。在事件执行时，可能需要访问其宿主系统。

上面两个类构成了一个采用面向对象技术的离散事件模拟框架。如果要基于它实现一个具体的模拟系统，只需把被模拟系统里的具体事件类定义为上面 Event 的派生类，在不同派生类里根据实际情况定义 run 方法。此外，还需要根据具体模拟的需要，定义一个具体的模拟系统类。

下面实例说明了怎样完成这一工作。

6.4.2 海关检查站模拟系统

现在用本节开始处提到的海关检查站作为具体例子，展示如何开发一个具体的模拟系统。对于这个系统，假设有如下的基本考虑：

- 海关的职责是检查过往车辆，这里只模拟一个通行方向的检查。
- 假定车辆按一定速率到达，有一定随机性，每 a 到 b 分钟有一辆车到达。
- 海关有 k 条检查通道，检查一辆车耗时 c 到 d 分钟。
- 到达的车辆在一条专用线路上排队等待，一旦有一个检查通道空闲，正在排队的第一辆车就进入该通道检查。如果车辆到达时有空闲通道而且当时没有等待车辆，它就立即进入通道开始检查。
- 希望得到的数据包括车辆的平均等待时间和通过检查站的平均时间。

综上所述，模拟的参数包括两个时间区间、通道数，还有总模拟时间。

模拟类

现在考虑如何针对这些需求，设计一个实际的模拟类。把这个类命名为 Customs，它的一个对象完成一次模拟工作。

首先分析完成这种模拟需要记录哪些信息。为了运行 Customs 的模拟过程，需要一个实际驱动模拟的 Simulation 对象。由于到达车辆可能排队，这个类的对象里需要有一个队列对象，记录正在排队的车辆。属性 waitline 的值是这个等待队列。几个检查通道是海关的内部资源，用表 gates 表示各通道的状态，用值 0 和 1 表示通道空闲或者正被占用。另外几个属性记录一些统计数据。

上述分析的结果都反映在 Customs 类的初始化函数里：

```
class Customs:  
    def __init__(self, gate_num, duration,  
                 arrive_interval, check_interval):  
        self.simulation = Simulation(duration)  
        self.waitline = SQueue()  
        self.duration = duration  
        self.gates = [0]*gate_num  
        self.total_wait_time = 0  
        self.total_used_time = 0  
        self.car_num = 0  
        self.arrive_interval = arrive_interval  
        self.check_interval = check_interval
```

有一批方法实现一些简单操作或服务，如累积时间和车辆计数，几个简单方法直接把工作传给 simulation、waitline 等。另外两个方法找空闲通道、释放完成工作的通道：

```
def wait_time_acc(self, n):  
    self.total_wait_time += n  
  
def total_time_acc(self, n):  
    self.total_used_time += n  
  
def car_count_1(self):  
    self.car_num += 1  
  
def add_event(self, event):  
    self.simulation.add_event(event)  
  
def cur_time(self):  
    return self.simulation.cur_time()  
  
def enqueue(self, car):  
    self.waitline.enqueue(car)  
  
def has_queued_car(self):  
    return not self.waitline.is_empty()  
  
def next_car(self):  
    return self.waitline.dequeue()  
  
def find_gate(self):  
    for i in range(len(self.gates)):  
        if self.gates[i] == 0:
```

```

        self.gates[i] = 1
        return i
    return None

def free_gate(self, i):
    if self.gates[i] == 1:
        self.gates[i] = 0
    else:
        raise ValueError("Clear gate error.")

```

最后两个方法分别实施模拟和输出统计数据：

```

def simulate(self):
    Arrive(0, self) # initially generate one car
    self.simulation.run()
    self.statistics()

def statistics(self):
    print("Simulate " + str(self.duration)
          + " minutes, for "
          + str(len(self.gates)) + " gates")
    print(self.car_num, "cars pass the customs")
    print("Average waiting time:",
          self.total_wait_time/self.car_num)
    print("Average passing time:",
          self.total_used_time/self.car_num)
    i = 0
    while not self.waitline.is_empty():
        self.waitline.dequeue()
        i += 1
    print(i, "cars are in waiting line.")

```

实现模拟的方法 `simulate` 把第一个到达事件加入模拟器，而后调用模拟器开始模拟，完成后通过统计函数输出数据。统计函数的功能就是输出一些数据。

排队队列里的对象表示通行的车辆。类 `Car` 实现这种对象，对象中记录了一辆车的到达时间，定义很简单：

```

class Car:
    def __init__(self, arrive_time):
        self.time = arrive_time

    def arrive_time(self):
        return self.time

```

为了能看到模拟过程中发生的情况，定义一个输出日志条目的函数。这种函数在程序开发后的调试运行阶段特别有用：

```

def event_log(time, name):
    print("Event: " + name + ", happens at " + str(time))
    pass

```

事件类

最后定义模拟中可能发生的事件。所谓事件，也就是这个系统运行中出现的一些关键情况。通过分析，可以看到海关检查站系统模拟需要下面几种事件：

- 汽车到达事件。一辆车到达时需要生成一个 `Car` 对象，在其中记录到达时间。这一事件的发生还意味着若干时间之后还将到达下一辆车等。为此定义一个类 `Arrive`，规范此类事件的行为。

- 汽车开始检查事件，因为这种事件太简单，下面未定义专门的类。
- 汽车检查完毕的离开事件，定义类 Leave。

上述事件类都是 Event 的派生类，放入事件队列的就是它们的对象。事件对象的构造函数完成必要的设置，包括将事件自身加入队列等。它们的 run 函数描述相应事件发生时应该出现的各种情况（动作、状态变化等）。

首先是 Arrive 类，其中只定义了一个构造函数和一个 run：

```
class Arrive(Event):
    def __init__(self, arrive_time, customs):
        Event.__init__(self, arrive_time, customs)
        customs.add_event(self)

    def run(self):
        time, customs = self.time(), self.host()
        event_log(time, "car arrive")
        # 生成下一个Arrive事件
        Arrive(time + randint(*customs.arrive_interval),
               customs)
        # 下面是本到达车辆事件的行为
        car = Car(time)
        if customs.has_queued_car():      # 有车辆在等，进入等待队列
            customs.enqueue(car)
            return
        i = customs.find_gate()          # 检查空闲通道
        if i is not None:               # 有通道，进入检查
            event_log(time, "car check")
            Leave(time + randint(*customs.check_interval),
                   i, car, customs)
        else:
            customs.enqueue(car)
```

构造函数首先记录事件发生的时间等信息，然后把事件压入事件队列，等待处理。run 方法描述了这个事件真正发生时的活动：首先创建下一个车辆到达事件，然后实现该车到达时的应有行为。到达事件应该创建一个 Car 对象表示到达的车辆，如果当时有排队的车辆，该车也加入排队；无车辆等待先查看是否有空闲通道，如果能开始检查就创建一个离开事件，否则也进入队列等待（虽无排队车辆，但各通道都在忙）。

离开事件类的定义如下，这种对象有两个属性，分别记录该离开事件的相关车辆和检查通道，以便在实际发生事件时使用：

```
class Leave(Event):
    def __init__(self, leave_time, gate_num, car, customs):
        Event.__init__(self, leave_time, customs)
        self.car = car
        self.gate_num = gate_num
        customs.add_event(self)

    def run(self):
        time, customs = self.time(), self.host()
        event_log(time, "car leave")
        customs.free_gate(self.gate_num)
        customs.car_count_1()
        customs.total_time_acc(time - self.car.arrive_time())
        if customs.has_queued_car():
            car = customs.next_car()
            i = customs.find_gate()
```

```

event_log(time, "car check")
customs.wait_time_acc(time - car.arrive_time())
Leave(time + randint(*customs.check_interval),
      self.gate_num, car, customs)

```

离开事件的关键是最后的条件语句。在一辆车离开检查站时，需要检查有无排队等待的车辆。如果有就开始检查其中第一辆车，还要生成相应的离开事件。

实际模拟

完成了上面所有工作，定义好相应的类和函数之后，就可以实际执行模拟了。下面几行代码先定义了模拟所需的一套参数，然后基于这些数据创建了一个模拟系统对象，最后一个语句启动模拟：

```

car_arrive_interval = (1, 2)
car_check_time = (3, 5)
cus = Customs(3, 480, car_arrive_interval, car_check_time)
cus.simulate()

```

下面是模拟系统的一段典型输出。由于这个系统运行中有随机因素，因此每次运行的实际情况可能不同。下面的输出说明，本次运行模拟了一个包含 3 个通道的海关检查站，共模拟 480 分钟（8 小时）。最后是车辆的平均排队时间和平均通过时间。由于有日志函数，系统还逐一输出了很多事件（删去大部分）：

```

Event: car arrive, happens at 0
Event: car check, happens at 0
Event: car arrive, happens at 1
...
...
Event: car leave, happens at 480
Event: car check, happens at 480
Simulate 480 minutes, for 3 gates
315 cars pass the customs
Average waiting time: 0.3904761904761905
Average passing time: 4.390476190476191
0 cars are in waiting line.

```

如果消去对日志函数的调用，模拟将只输出统计结果。

至此，这个具体的模拟系统就全部完成了。它也说明了要基于前面的模拟框架实现具体模拟系统时需要做哪些工作，可供参考。总结一下这个工作：

- Simulation 类和 Event 类实现了一个支持离散事件模拟的通用框架。
- 实际事件类的 run 方法通过生成新事件完成模拟过程的实际控制。
- Customs 类实现了海关检查站模拟系统的基础支撑功能和主控函数。
- 其中用一个队列作为缓冲，保存已经到来但还不能立即检查的车辆。

本章最后的练习提出了一些进一步的工作。

6.5 二叉树的类实现

前面介绍了二叉树概念，介绍了基于 Python 的 list 或 tuple 的实现，还介绍了二叉树的一些应用。这些应用主要是利用了二叉树的结构，特别是利用了二叉树的结点个数与高度之间的对数关系，以提高一些操作的效率。

现在考虑如何在 Python 里定义一种二叉树数据结构类型。实际上，可以基于前面讨论过

的 list 或 tuple 的实现技术，以其作为二叉树类型的内部表示方式。这种可能的做法留给读者研究，并请将其与下面讨论的技术做些对比。

下面的实现方法也称为链接实现，与连续表的链接实现技术类似：用一个数据单元表示一个二叉树结点，通过子结点链接（指针）建立结点之间的联系。采用这种表示方法，只要掌握了一棵二叉树的根结点，也就掌握了整个二叉树。

6.5.1 二叉树结点类

由于二叉树由一组结点组成，现在先定义一个表示二叉树结点的类。结点通过链接引用其子结点，没有子结点时用 None 值，也就是说，空二叉树直接用 None 表示。

下面是基于这些考虑定义的二叉树结点类：

```
class BinTNode:  
    def __init__(self, dat, left = None, right = None):  
        self.data = dat  
        self.left = left  
        self.right = right
```

这个结点类的构造函数接收三个参数，分别为其结点数据和左右子结点。两个参数都用默认值时构造出的是叶结点。

下面语句构造了一棵包含三个结点的二叉树，变量 t 引着树根结点：

```
t = BinTNode(1, BinTNode(2), BinTNode(3))
```

基于 BinTNode 类的对象构造的二叉树具有递归的结构，很容易采用递归的方式处理。下面两个函数定义展示了处理这种二叉树的典型技术：

```
# 统计树中结点个数:  
def count_BinTNodes(t):  
    if t is None:  
        return 0  
    else:  
        return 1 + count_BinTNodes(t.left) \  
               + count_BinTNode(t.right)  
  
# 假设结点中保存数值，求这种二叉树里的所有数值和：  
def sum_BinTNodes(t):  
    if t is None:  
        return 0  
    else:  
        return t.dat + sum_BinTNodes(t.left) \  
               + sum_BinTNodes(t.right)
```

注意，其中一行最后的反斜线符号表示续行。

可以看到，递归定义的二叉树操作具有统一的模式，包括两个部分：

- 描述对空树的处理，应直接给出结果。
- 描述非空树情况的处理：
 - 如何处理根结点（处理根结点数据时应直接给出结果）。
 - 通过递归调用分别处理左、右子树。
 - 基于上述三个部分处理的结果得到对整个树的处理结果。

本章后面练习中提出了一些问题，都有可能采用这一模式描述。

6.5.2 遍历算法

6.1.3节抽象地讨论了遍历二叉树的深度和宽度优先算法，本节考虑如何针对链接的二叉树结点构成的二叉树，定义实现遍历二叉树的函数。

递归定义的遍历函数

要实现按深度优先方式遍历二叉树，采用递归方式定义函数非常简单。采用非递归方式定义这类函数也有意义，放在后面讨论。

下面是按先根序遍历二叉树的递归函数：

```
def preorder(t, proc): # proc是具体的结点数据操作
    if t is None:
        return
    proc(t.data)
    preorder(t.left)
    preorder(t.right)
```

按中根序和后根序遍历二叉树的函数与此类似，只是其中几个操作的排列顺序不同。本书所附代码文件里有相关定义，读者也很容易自己写出来。

这里假定 `t` 的实际参数是 `BinTNode` 类型的对象。如果认为需要，可以在函数最前面加断言语句 `assert(isinstance(t, BinTNode))` 检查参数类型，也可以用条件语句检查，并在类型不正确时抛出异常。

为能看到具体二叉树的情况，现在定义一个以易读形式输出二叉树的函数。这里采用带括号的前缀形式输出，为显示空子树输出一个符号“^”：

```
def print_BinTNodes(t):
    if t is None:
        print("^", end="")
        return
    print("(" + str(t.data), end="")
    print_BinTNodes(t.left)
    print_BinTNodes(t.right)
    print(")", end="")
```

易见，这个函数也是递归定义的先根序遍历。下面是使用示例：

```
t = BinTNode(1, BinTNode(2,BinTNode(5)), BinTNode(3))
print_BinTNodes(t)
输出: (1(2^(5^)))(3^))
```

可以看出，如果不在遇到空树输出一个记号，只有一棵子树时就无法区分左右了。

宽度优先遍历

要实现采用宽度优先方式的二叉树遍历函数，同样需要用一个队列。下面定义里使用了前面定义的 `SQueue` 类：

```
from SQueue import *
def levelorder(t, proc):
    qu = SQueue()
    qu.enqueue(t)
    while not qu.is_empty():
        n = qu.dequeue()
        if t is None: # 弹出的树为空则直接跳过
            continue
        proc(n.data)
        if n.left is not None:
            qu.enqueue(n.left)
        if n.right is not None:
            qu.enqueue(n.right)
```

```

qu.enqueue(t.left)
qu.enqueue(t.right)
proc(t.data)

```

在处理一个结点时，函数先将其左右子结点顺序加入队列。这样实现的就是对每层结点从左到右的遍历。上面写法也可能把一些空树加入队列，浪费存储空间。可以考虑在操作前检查子结点的存在情况，有关修改很简单。

非递归的先根序遍历函数

下面讨论非递归定义的深度优先遍历算法，出于一些有意义的考虑：

- 从中可以进一步理解递归与非递归的关系。
- 进一步看清二叉树遍历的具体过程和一些性质。
- 有关算法本身也有用，还可以看到分析问题和设计算法的一些情况。

这里先考虑定义一个实现了先根序遍历的非递归函数。在三种深度优先遍历中，这种遍历方式的非递归描述最简单。

根据已有的认识，在这个函数里需要一个栈，保存树尚未访问过部分的信息。显然，即使确定了采用先根序遍历的方式，还是可能有多种不同的实现方法。下面考虑一种方法，其中的基本想法很简单：

- 由于采取先根序，遇到结点就应该访问，下一步应该沿着树的左枝下行。
- 但结点的右分支（右子树）还没有访问，因此需要记录，将右子结点入栈。
- 遇到空树时回溯，取出栈中保存的一个右分支，像一棵二叉树一样遍历它。

每次遇到空树，总是遍历一棵子树的工作完成。如果这是一棵左子树，对应右子树应该是当时的栈顶元素。如果这是一棵右子树，说明以它为右子树的更大子树已完成遍历，下一步应处理更上一层的右子树。算法还有一些细节，主要是循环的控制。

循环中需要维持一种不变关系：假设变量 *t* 一直取值为当前待遍历子树的根，栈中保存着前面遇到但尚未遍历的那些右子树。这样，只要当前树非空（这棵树需要遍历）或者栈不空（还存在未遍历的部分），就应该继续循环，这就是循环继续的条件。

循环体中应该先处理当前结点的数据，而后沿着树的左分支下行，一路上把经过结点的右分支压入栈，与此也需要用一个循环。内部循环直至遇到空树时回溯，从栈里弹出一个元素（最近的一棵右子树），要做的工作也是遍历一棵二叉树。

把这些问题都看清楚以后，定义出的函数非常简单：

```

def preorder_nonrec(t, proc):
    s = SStack()
    while t is not None or not s.is_empty():
        while t is not None:      # 沿左分支下行
            proc(t.data)          # 先根序先处理根数据
            s.push(t.right)        # 右分支入栈
            t = t.left
        t = s.pop()               # 遇到空树，回溯

```

如果变量 *tree* 的值是一棵二叉树，其结点中保存的是可打印数据，下面语句将逐项输出该树里的数据内容，用空格分隔：

```
preorder_nonrec(tree, lambda x:print(x, end=" "))
```

这里用了一个 *lambda* 表达式，其中定制了输出形式，输出一项后不换行。

非递归的中根序遍历算法与先根序算法类似，在本书所附的代码文件里有，请读者自己分

析。建议读者先设法写出一个定义，再与文件里的定义比较。

非递归算法的一个价值是把算法过程完整暴露出来，便于进行细致的分析。现在考虑非递归的先根序遍历算法的复杂性。

时间复杂性：上面函数在整个执行中将访问每个结点一次，一部分子树（所有右子树）被压入和弹出栈各一次（栈操作是 $O(1)$ 时间的），`proc(t.data)` 操作的复杂性与树的大小无关，所以整个遍历过程需要花费 $O(n)$ 时间。

空间复杂性：这里的关键因素是遍历中栈可能达到的最大深度（栈中元素的最大个数），而栈的最大深度由被遍历的二叉树的高度决定^Θ。由于二叉树高度可能达到 $O(n)$ ，所以，在最坏情况下，算法的空间复杂性是 $O(n)$ 。另外，前面说过， n 个结点的二叉树的平均高度是 $O(\log n)$ ，所以，非递归先根序遍历的平均空间复杂性是 $O(\log n)$ 。

在一些情况下，修改实现方法也可能降低空间开销。对上面函数，修改其定义，只把非空的右子树进栈，在许多情况下能减少一些空间开销。有关修改请读者完成。

通过生成器函数遍历

用 Python 写程序，在考虑遍历数据汇集结构的时候，总应该想到迭代器。前面讨论链表数据结构时，介绍过遍历其中数据元素的 `elements` 和 `filter`。简单修改前面的非递归先根序遍历函数，就能得到一个二叉树迭代器：

```
def preorder_elements(t):
    s = SStack()
    while t is not None or not s.is_empty():
        while t is not None:
            s.push(t.right)
            yield t.data
            t = t.left
        t = s.pop()
```

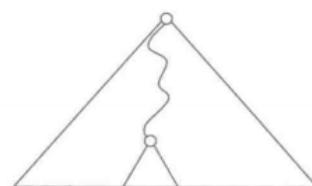
其他非递归定义的遍历算法，包括中根序和下面讨论的后根序算法，以及层次遍历算法，都可以同样直截了当地修改成迭代器。但递归算法不行，请读者自己分析其中的原因。在研究链接表迭代器时，已经看到基于迭代器处理汇集中元素的优势。由此可见，非递归遍历算法另一重要用途就是作为实现迭代器的基础。

非递归的后根序遍历算法

在二叉树的几种非递归遍历中，后根序算法最难写。在这种遍历中，每个根结点都要经过三次：第一次遇到它时立即转去处理其左子树，从左子树回到这里就应转到右子树，从右子树回来才应该处理根结点数据，而后返回上一层。

每个问题都可能写出多个不同的具体算法。下面介绍的一种后根序遍历算法，是能找到的各种算法中最简短的一个。为解释这个算法，请参考图 6.17。图中曲折线表示运行中某个时刻（实际上是每个时刻）位于栈里的结点序列，以被遍历的二叉树的根结点（曲折线上端的小圆圈）作为栈底元素，直至栈顶，这些结点构成树中的一条路径。这也意味着栈里每个结点的父结点就是位于它下面的那个结点。

另外，变量 `t` 的值是当前结点（可能空）。在实现遍历 图 6.17 非递归后根序遍历中的情况



^Θ 实际情况还有些复杂，本节最后讨论。

的循环中维持一种不变关系：

- 栈中结点序列的左边是二叉树已遍历过的部分，右边是尚未遍历的部分。
- 如果 t 不空，其父结点就是栈顶结点。
- t 空时栈顶就是应访问的结点。

根据被访问结点是其父结点的左子结点或右子结点，就可以决定下一步怎么做：如果是左子结点就转到右兄弟；如果是右子结点，就应该处理根结点并强制退栈。

函数定义中的一个重要技术是一个下行循环（函数体中的内层循环），目标是找到下一个应访问结点。循环中用了一个条件表达式，要求在有左子树时持续向左下行，没有左子树时向右一步。该循环结束说明栈顶结点没有左右子树，应该访问。如果外层循环一次迭代不进入内层循环体，就说明栈顶结点的左右子树都已遍历，应该访问栈顶结点。

函数定义很简短，但算法中的想法有些复杂：

```
def postorder_nonrec(t, proc):
    s = SStack()
    while t is not None or not s.is_empty():
        while t is not None:      # 下行循环，直到栈顶的两子树空
            s.push(t)
            t = t.left if t.left is not None else t.right
        t = s.pop()                # 注意这个条件表达式的意义：能左就左，否则向右一步
        proc(t.data)               # 栈顶是应访问结点
        if not s.is_empty() and s.top().left == t:
            t = s.top().right     # 栈不空且当前结点是栈顶的左子结点
        else:
            t = None              # 没有右子树或右子树遍历完毕，强迫退栈
```

注意：①内层循环找当前子树的最下最左结点，将其入栈后终止；②如果被访问结点是其父的左子结点，直接转到其右兄弟结点继续；③如被处理结点是其父的右子结点，设 t 为 `None` 将迫使外层循环的下次迭代弹出并访问更上一层的结点。

非递归的遍历

现在分析非递归遍历算法的一些情况。首先，从时间复杂性看，几种不同遍历算法中都是访问每个结点一次，压栈（退栈）的次数不超过结点个数，因此，时间复杂性都是结点个数的线性函数，即 $O(n)$ 。

但对于空间复杂性，不同非递归遍历算法的情况则可能不同。考虑图 6.18 给出的几棵二叉树，它们的高度都达到 $O(n)$ 量级。

首先，如果用递归算法遍历这些二叉树，在遍历中，函数递归的深度都会达到 $O(n)$ ，因此需要用到 $O(n)$ 的辅助空间。

非递归的后根序算法在遍历中一定会把树中最长的一整条路径全部存入栈。因此，如果被遍历二叉树的深度为 n ，无论其具体结构如何，栈的深度都会达到 n 。

考虑先根序的非递归遍历算法，前面说过，其中可以只入栈非空的右子树。如果真那样做了，在遍历任何单枝树（如图中的 K_1 或 K_2 ）时，栈的深度都不会超过 1。最坏情况是图中的树 K_3 的情况，栈的深度可能达到大约 $n/2$ ，由此也是 $O(n)$ 。

中根序遍历的情况与先根序类似。请读者设法写出合适的程序，使栈空间使用达到最少。

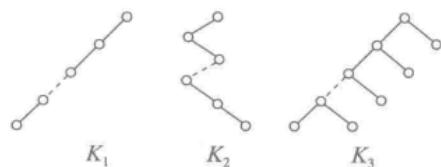


图 6.18 几棵深度达到 $O(n)$ 的二叉树

6.5.3 二叉树类

直接基于结点构造的二叉树具有递归的结构，可以很方便地递归处理。但这样的“二叉树结构”也有不统一之处：其中用 None 表示空树，但 None 的类型并不是 BinTNode。此外，基于二叉树结点构造的结构，就像前面基于链表结点构造的结点链，并不是一种良好封装的抽象数据类型。解决这些问题的方法就是定义一个二叉树类，以前面定义的 BinTNode 结点链接而成的树形结构作为其内部表示。

下面是二叉树类定义的基本部分：

```
class BinTree:
    def __init__(self):
        self._root = None

    def is_empty(self):
        return self._root is None

    def root(self):
        return self._root

    def leftchild(self):
        return self._root.left

    def rightchild(self):
        return self._root.right

    def set_root(self, rootnode):
        self._root = rootnode

    def set_left(self, leftchild):
        self._root.left = leftchild

    def set_right(self, rightchild):
        self._root.right = rightchild
```

还可以考虑定义一个或多个元素迭代器，例如：

```
def preorder_elements(self):
    t, s = self._root, SStack()
    while t is not None or not s.is_empty():
        while t is not None:
            s.push(t.right)
            yield t.data
            t = t.left
        t = s.pop()
```

其他操作可以根据需要定义，也可以考虑以这个二叉树为基类定义派生类，后面会看到这方面的例子。在上面定义中，除了遍历操作外，其他操作都具有 O(1) 的时间复杂度。在这种表示中，求父结点的操作比较难实现。它相当于单链表结构里的求前一结点操作，只能通过一次从根开始的遍历才能实现，最坏时间代价是 O(n)。

如果工作经常需要找父结点，可以考虑给每个结点增加一个父结点链接域，在设置子结点链接关系的同时设置好父结点链接。图 6.19 给出了采用这种表示方法的二叉树示意图。请读者定义这样的结点类和二叉树类。可以自己定义，也可以继承 BinTNode 和 BinTree。

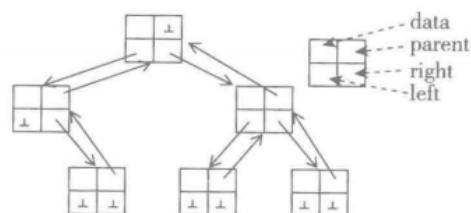


图 6.19 带父结点链接的二叉树

6.6 哈夫曼树

哈夫曼树 (Huffman tree) 是一种重要的二叉树，在信息领域有重要的理论和实际价值。这里的讨论将其看作二叉树的一种重要应用。

6.6.1 哈夫曼树和哈夫曼算法

前面介绍过扩充二叉树的外部路径长度，即根到其外部结点的路径长度之和：

$$E = \sum_{i=1}^m l_i$$

其中 m 是扩充二叉树中外部结点的个数， l_i 是从根到外部结点 i 的路径长度。

带权扩充二叉树的外部路径及其长度

考虑上面概念的一种扩充。给扩充二叉树的每个外部结点标一个数值，称为该结点的权，表示与该叶有关的某种性质。进而定义带权扩充二叉树的外部路径长度为

$$WPL = \sum_{i=1}^m w_i l_i$$

其中 w_i 是外部结点 i 的权。

【例 6.7】 考虑图 6.20 中的带权扩充二叉树，外部路径长度很容易计算：

左边树： $(2+3+6+9) \times 2 = 38$

右边树： $9+6 \times 2+(2+3) \times 3 = 36$

可见，最规整的树未必路径最短。

哈夫曼树

定义 设有实数集 $W=\{w_0, w_1, \dots, w_{m-1}\}$ ， T 是一棵扩充二叉树，其 m 个外部结点分别以 w_i ($i=1, 2, \dots, n-1$) 为权，而且 T 的带权外部路径长度 WPL 在所有这样的扩充二叉树中达到最小，则称 T 为数据集 W 的最优二叉树或者哈夫曼树。

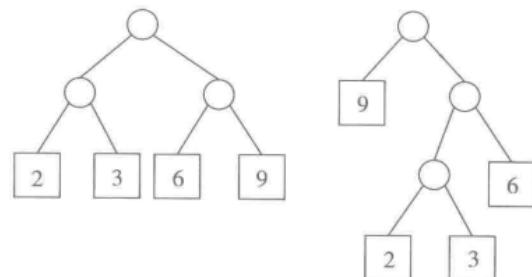


图 6.20 两棵带权扩充二叉树

显然，以同一集实数为外部结点权的扩充二叉树可能有许多，它们的 WPL 值可能不同。图 6.20 的两棵树即是一例。注意，这里将 W 看作集合而不是序列，在构造相应的扩充二叉树时，集合中的实数值可以按任意方式选取。

构造哈夫曼树的算法

哈夫曼 (D. A. Huffman) 提出了一个算法，它能从任意的实数集合构造出与之对应的哈夫曼树。这个构造算法描述如下：

- 算法的输入为实数集 $W=\{w_0, w_1, \dots, w_{m-1}\}$ 。
- 在构造中维护一个包含 k 棵二叉树的集合 F ，开始时 $k=m$ 且 $F=\{T_0, T_1, \dots, T_{m-1}\}$ ，其中每个 T_i 是一棵只包含权为 w_i 的根结点的单点二叉树。
- 算法过程中重复执行下面两个步骤，直到集合 F 中剩下一棵树为止：
 - 构造一棵新二叉树，其左右子树是从集合 F 中选取的两棵权最小的二叉树，其根结点的权值设置为这两棵子树的根结点的权值之和。
 - 将所选的两棵二叉树从 F 中删除，把新构造的二叉树加入 F 。

易见，步骤2每做一次， F 里的二叉树就减少了一棵，这就保证了本算法必定结束。

另一方面，要证明这一算法做出的是哈夫曼树（最优二叉树）则不太容易（请读者自己考虑）。只能用结构归纳法，关键是怎么论述清楚归纳证明的一步。

还请注意：给定集合 W 上的哈夫曼树不唯一。如果 T 是集合 W 上的哈夫曼树，交换其中任意一个或多个结点的左右子树，得到的仍是 W 上的哈夫曼树。

【例6.8】 考虑实数集 $W=\{2, 3, 7, 10, 4, 2, 5\}$ ，图6.21展示的构造过程从这个集合出发，做出了一棵哈夫曼树。

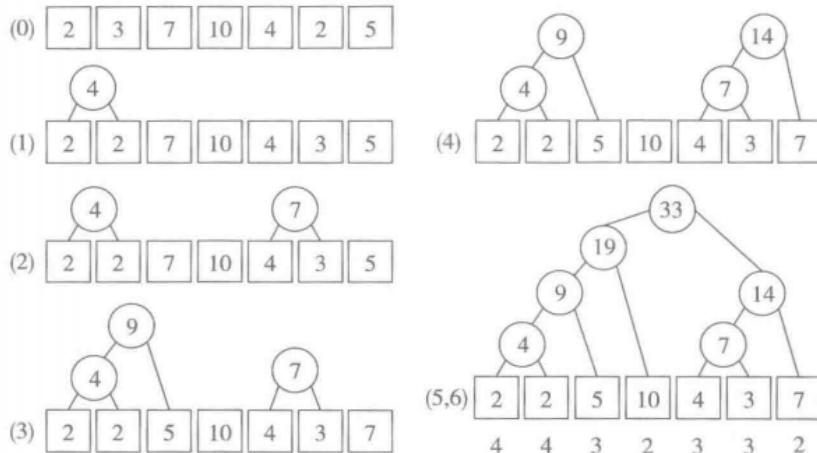


图6.21 构造哈夫曼树

构造中，有时会遇到存在多种选择的情况。例如在图6.21里的第(2)步，存在两个权值为4的树，可以选择其中任意一个与权值为3的树合并。不同选择会导致不同的哈夫曼树，但其外部路径的长度一定相等。

6.6.2 哈夫曼算法的实现

现在考虑哈夫曼树构造算法的实现。由于这里是用Python编程，显然应该很好利用这个语言的各方面特点。

构造算法

显然，构造算法执行中需要维护一组二叉树，而且要知道每棵树（其树根结点的）的权值。可以考虑用二叉树的结点类构造哈夫曼树，在树根结点记录树的权值。

在算法的执行中，需要不断选出权值最小的两棵二叉树，并基于它们构造出一棵新二叉树。很容易想到，最佳选择就是用一个优先队列存放这组二叉树，按二叉树根结点的权值排列优先顺序，从小到大。

算法开始时建立起一组单结点的二叉树，以权值作为优先码存入优先队列，要求先取出队列里的最小元素。然后反复做下面两件事，直至优先队列里只有一个元素：

- 1) 从优先队列里弹出两个权最小的元素（两棵二叉树）。
- 2) 基于所取的二叉树构造一棵新的二叉树，其权值取两棵子树的权值之和，并将新构造的二叉树压入优先队列。

这里还有两个必须解决的小问题：需要为二叉树定义一个序，权值小的二叉树在前；需要检查优先队列中的元素（二叉树）个数，以便在只剩一棵时结束。这些都可以通过扩充前面已

定义的类型实现。这里基于已有的类派生出两个类：

```
class HTNode(BinTNode):
    def __lt__(self, othernode):
        return self.data < othernode.data

class HuffmanPrioQ(PrioQueue):
    def number(self):
        return len(self._elems)
```

第一个类定义以二叉树结点类作为基类，定义了一个专门为构造哈夫曼树用的结点类，其特点就是增加了一个“小于”比较操作。随后定义了一个专门为哈夫曼算法服务的优先队列类，其中增加了一个检查队列中元素个数的方法。

做好了上面的准备，哈夫曼树生成算法的实现直截了当，基本上就是前面算法的文字描述的简单翻版，其中操作过程直接反映了所需的计算工作：

```
def HuffmanTree(weights):
    trees = HuffmanPrioQ()
    for w in weights:
        trees.enqueue(HTNode(w))
    while trees.number() > 1:
        t1 = trees.dequeue()
        t2 = trees.dequeue()
        x = t1.data + t2.data
        trees.enqueue(HTNode(x, t1, t2))
    return trees.dequeue()
```

这里唯一需要解释的是参数。可以用任何可迭代对象作为这个函数的参数。函数里的第一个循环获取可迭代对象的一个个值，并将其加入优先队列。

算法分析

哈夫曼树构造算法的时间复杂性主要是两个循环。

第一个循环建立起 m 棵二叉树，并把它们加入优先队列。按上面写法，这部分计算的时间复杂度是 $O(m \log m)$ ，因为加入一个元素时需要做一次 $O(\log m)$ 复杂度的筛选。读者应该记得，前面讨论过基于堆的优先队列和堆排序的初始建堆，那里使用的方法只需要 $O(m)$ 时间。请读者自己修改程序，改用前面的方法。

第二个循环需要做 $m-1$ 次，每次减少一棵树。构造新树的时间复杂性与 m 无关，是 $O(1)$ 复杂度的操作。每次迭代把一棵新二叉树加入优先队列，需要 $O(\log m)$ 时间。整个循环是 $O(m \log m)$ 。可见，这个算法的时间复杂度是 $O(m \log m)$ 。

算法执行中构造出一棵包含 $2m-1$ 个结点的树，所以其空间复杂度是 $O(m)$ 。

6.6.3 哈夫曼编码

哈夫曼树有许多应用，针对不同应用，需要给树中的权值赋予不同的意义。现在讨论它的一个重要应用：哈夫曼编码。这是当年哈夫曼研究并提出哈夫曼树的出发点，在信息理论里有重要的理论意义，也有实际价值。

定义

最优编码问题：给定基本数据集合：

$$C = \{c_0, c_1, \dots, c_{m-1}\}, \quad W = \{w_0, w_1, \dots, w_{m-1}\}$$

其中集合 C 是需要编码的字符集合， W 为 C 中各个字符在实际信息传输（或者信息存储）中出

现的频率。现在要求为 C 设计一套二进制编码，使得：

- 1) 用这种编码存储 / 传输时的平均开销最小。
- 2) 对任一对不同字符 c_i 和 c_j ，字符 c_i 的编码不是 c_j 编码的前缀。

第二个条件可以使解码更方便，容易判断是否已经得到一个字符的编码。因为任一字符的编码都不是另一字符的编码的前缀，只要已经得到的编码段对应于一个字符的编码，就可以确定原文里必然是这个字符^Θ。

哈夫曼编码的生成

哈夫曼提出了一种解决这个问题的方法，即所谓哈夫曼编码。构造的方法就是首先构造出一棵哈夫曼树，基于它做出哈夫曼编码。有关过程是：

- 以 $W = \{w_0, w_1, \dots, w_{m-1}\}$ 作为 m 个外部结点的权，以 $C = \{c_0, c_1, \dots, c_{m-1}\}$ 中字符作为外部结点标注，基于 W 和相应结点集构造出一棵哈夫曼树。
- 在得到的哈夫曼树中，在从树中各个分支结点到其左子结点的边上标注二进制数字 0；在所有到右子结点的边上标注数字 1。
- 以从根结点到一个叶结点（外部结点）的路径上的二进制数字序列，作为这个叶结点的标记字符的编码，这样得到的就是哈夫曼编码。

可以证明：对于任意的数据集合对 (W, C) ，按上述方式得到的哈夫曼编码是字符集 C 的最优（最短）编码。哈夫曼编码在编码理论里有重要意义，是给定字符集（在确定的概率分布情况下）的最优编码。

显然，不难利用前面给出的代码，实现一个程序生成字符串的哈夫曼编码。相应的开发工作留作读者练习。

【例 6.9】 假设现在有“字符：权值”组 $\{a:2, b:3, c:7, d:4, e:10, f:2, h:5\}$ ，要求通过哈夫曼的算法做出相应的哈夫曼编码。

首先需要做出与数据中的权值对应的哈夫曼树，也就是图 6.21 最后的结果。按哈夫曼编码的要求在树中各边标注数字 0 和 1，得到如图 6.22 所示情况，其中外部结点标注对应的字符。参看这棵树中的路径，就得到了下面编码：

a:	0000
b:	101
c:	11
d:	100
e:	01
f:	0001
h:	001

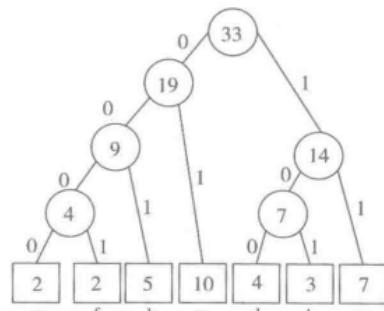


图 6.22 哈夫曼编码实例

假设收到了报文：001010010000100000010101100，根据图 6.22 中的哈夫曼树，很容易可以得到对应的解码正文：hehadabed。

6.7 树和树林

现在考虑一般的树和树的集合，称为树林。作为概念，树代表很广泛的一类树形结构，在许多方面有与二叉树类似的概念，但二叉树并不是树的特例。

作为树形结构，树具有前面提出的树形结构的所有共性性质。

Θ 注意：这里考虑的是编码效率的优化问题，不同字符的编码长度可以不同。目前常用的 ASCII 码等采用等长编码（所有字符用同样长的二进制序列编码），没考虑这里讨论的优化。

6.7.1 实例和表示

现实世界中的许多事物可以抽象出一种树形的结构，例如：

- 家族关系。家谱的简单情况是树形，长辈有子女，世代分层。
- 机关、组织、公司的组织结构关系。
- 复杂机械设备的零部件组成；等等。

这类结构具有明显的层次性，其中的高层元素可能与低层元素有关，同层元素之间相互无关，也没有从低层到高层的关联。此外，与不同元素相关的元素集合互不相交。这些都符合本章开始讨论的树形结构的基本性质。将这些结构里的基本元素抽象为结点，将结构中上层元素与下层元素的联系抽象为结点之间的联系，就得到了树。

例如，一个大家庭四代同堂，曾祖一辈有三个孩子，这些第2代分别有两个、一个和三个孩子，第3代的一些孩子已经有了第4代，共计14人。给这个家庭的每个人一个字母标记，用集合形式表示如下：

- 所有人的集合是： $N = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N\}$ 。
- 反映其家庭结构的父子关系是： $R = \{\langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle, \langle B, E \rangle, \langle B, F \rangle, \langle C, G \rangle, \langle D, H \rangle, \langle D, I \rangle, \langle G, J \rangle, \langle G, K \rangle, \langle H, L \rangle, \langle I, M \rangle, \langle I, N \rangle\}$ 。

这样的（结点）集合 N 和（结点之间的）关系 R 反映了上述家庭的组成情况，形成该家庭组成的一种抽象描述。而这一描述就其构成而言，形成了一棵树。

虽然这种集合形式的描述是严格而且明确的，但却不太直观。为了帮助理解和学习，人们提出了树的多种直观的表达形式，主要是图示形式。这些形式都比较容易理解。当然，这类图示只能用于表示较小的树。

最典型的图示形式与前面二叉树类似，用圆圈表示树中结点，结点间联系用圆圈之间的连线表示。上层结点画在上面，因此结点之间的上下关系表示了联系方向。例如，图 6.23 画出的就是上面的家庭关系树。

人们也考虑用其他方法描绘图的结构。例如文氏图（Venn Diagram，也称韦恩图）由 19 世纪英国的哲学家和数学家 John Venn 发明，用于描述集合关系。把树中向下关联看作集合包含，就可以用文氏图描述树了。但该方式实际使用不多。

另一种较常见的是嵌套括号表示法，书写比较方便，计算机也容易处理。这种形式在前面讨论二叉树时已经多次出现，例如在二叉树的 list 表示和递归定义的二叉树输出函数中。用嵌套括号表示，每个括号里的第一项是本层（结点的）数据，随后内嵌的括号及内容表示树中的子部分，其结构与外层一样。这种表示由著名的编程语言 Lisp 引入。

6.7.2 定义和相关概念

本小节给出树的定义，介绍一些与之相关的概念，讨论其性质。其中许多概念和性质与二叉树类似。树（tree）是具有递归性质的结构，其定义也是递归的。

定义 一棵树是 n ($n \geq 0$) 个结点的有限集 T ，当 T 非空时满足：

- T 中有且仅有一个特殊结点 r 称为树 T 的根。
- 除根结点外的其余结点分为 m ($m \geq 0$) 个互不相交的非空有限子集 T_0, T_1, \dots, T_{m-1} ，每个集合 T_i 为一棵非空树，称为 r 的子树（subtree）。

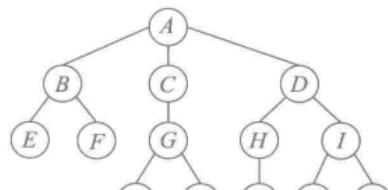


图 6.23 树的实例

这里的结点是不加定义的概念。定义中要求子树都非空，这样就能保证子树的个数（和树的结构）有确切的定义。

结点个数为 0 的树称为空树。一棵树可以只有根但没有子树 ($m=0$)，这就是单结点的树，只包含一个根结点。易见，树与二叉树类似，也是一种层次性结构。应该把子树的根看作树根的下一层元素，一棵树里的结点可以这样分为一层层结点。

一棵树（的树根）可能有多棵子树，因此就有子树的排列顺序是否有意义的问题。在有序树中，每个结点的子树都有明确规定了的顺序，因此可以说第一棵子树，下一棵子树等；而在考虑无序树时，认为一个结点的不同子树没有顺序关系^Θ。有序树的图示中子树从左到右排列，因此，按照有序树的观点，图 6.24 中的两个图表示的是两棵不同的树。而按照无序树的观点，它们表示了同一棵树。

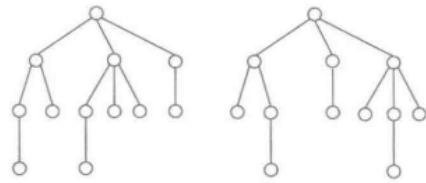


图 6.24 两棵树：相同还是不同

由于计算机表示中存在自然的顺序，所以数据结构里主要考虑有序树。

相关概念

一些概念的定义与二叉树类似，如：父结点和子结点，边（有方向），兄弟结点，树叶和分支结点，祖先或子孙关系，祖先结点和子孙结点，路径和路径长度，结点的层数，树的高度（或深度）。在有序树里可以考虑子结点的顺序，因此可以说最左结点等。在这里，一个结点的度数也定义为其子结点的个数。与二叉树不同，树中结点的度数可以任意，树的度数定义为该树中度数最大的结点的度数。

最后请注意，二叉树中的子结点有明确的左右之分，而且其结点的最大度数是 2。在度数为 2 的有序树中结点也有序且最大度数为 2。但这两者却是不同的概念。不同之处只有一点：假设树中的某个分支结点只有一个子结点，在二叉树中必须说明它是左分支还是右分支，而在度数为 2 的有序树里就没有这个概念了。

树林

定义 0 棵或多棵树（显然互不相交）的集合称为一个树林。

树和树林可以相互递归定义：一棵空树就是一个空树林。如果树 T 不空，它可分解为二元组 $T=(r, F)$ ，其中 r 是树根， F 是 r 的所有子树构成的树林。设树 T 有 m ($m \geq 0$) 棵非空子树，则 $F=(T_0, T_1, \dots, T_{m-1})$ ，其中的 T_i 为 r 的第 i 棵子树（这是有序树的情况；对于无序树， F 没有顺序）。这一讨论说明，树和树林可以相互递归定义：非空树由树根及其子树树林构成，而树林则由一组树组成。

对于树林，同样需要区分有序树林和无序树林的概念。对于有序树林，可以说它的第一棵树，下一棵树等等。

树、树林与二叉树的关系

实际上，存在一种一一对应关系，可以把任何一个（有序）树林映射到一棵二叉树，而其逆映射把这棵二叉树映射回原来的树林。这个映射定义如下：

- 顺序连接同一结点的各子结点（它们在原树林里互为兄弟结点），作为这些结点的右分支的边（也就是说，将树 / 树林中下一兄弟作为二叉树里的右分支）。

^Θ 一些教科书中说无序树的子树是一个集合，这种说法有点问题。树中的一个结点可以有两棵或多棵完全一样的子树，因此，一个结点的所有子树并不是集合。

- 保留每个结点到其第一个子结点的连接作为该结点的左分支，并删去这个结点到它的其他子结点的连接（即，原树林里第一个子结点作为二叉树里的左分支）。

这里把树林里各棵树的根也看作兄弟结点，需要从第一棵树的根开始建立连接。

【例 6.10】 看图 6.25 中左边的树林，其中包含三棵树。要得到对应的二叉树，先联系起三棵树的根，并在各树中连接兄弟结点，如右图中增加的粗黑实线。再删除从各结点到其除第一个子结点之外其他子结点的连线（如图中虚线所示的连线），就得到了对应的二叉树。要想让图示更规范，只需稍微调整子结点的位置。

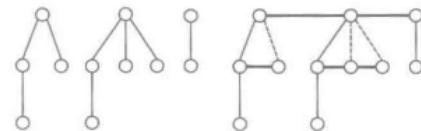


图 6.25 树林到二叉树的映射

二叉树到树林的转换也不难完成：

- 对每个结点，在它与其左子结点的向右路径上的每个结点间加一条边。
- 删除原二叉树中每个结点向右路径上所有的边。

图 6.26 给出了一个简单的例子。按上面步骤，图中首先加了两条边，用粗实线表示，然后删除了四条边（图中虚线），最后得到的是一个包含了三棵树的树林。

树可以看作由一棵树组成的树林，因此树与二叉树的一个子集之间存在一一对应关系。具体情况很简单，请读者考虑。

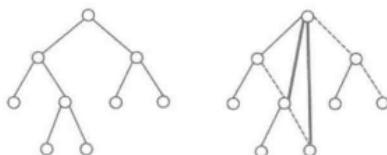


图 6.26 二叉树到树林的转换

树的性质

作为一种抽象结构，树也有许多重要性质。其中一些与二叉树的性质类似。

性质 6.8 在度数为 k 的树中，第 i 层至多有 k^i 个结点。

性质 6.9 度数为 k 、高为 h 的树中至多有 $\frac{(k^{h+1}-1)}{(k-1)}$ 个结点。

k 度完全树是如下的 k 度树，其中除最下一层分支结点中最右的那个结点的度数可能小于 k 外，其余分支结点的度数均为 k 。有下面性质：

性质 6.10 n 个结点的 k 度完全树，高度 $h = \lfloor \log_k n \rfloor$ ，即不大于 $\log_k n$ 的最大整数。

易见， n 个结点的 k 度完全树，其高度比 n 个结点的完全二叉树小一个常量因子 $\log_2 k$ 。在实际中这个常量因子也可能有价值，可以利用。

性质 6.11 n 个结点的树里有 $n-1$ 条边。

6.7.3 抽象数据类型和操作

前面两小节讨论的是作为抽象概念的树，下面考虑树作为数据结构的问题。

抽象数据类型

首先定义一个树抽象数据类型，描述其相关的操作：

ADT Tree:	# 一个树抽象数据类型
Tree(self, data, forest)	# 构造操作，基于树根数据和一组子树
is_empty(self)	# 判断是否为一棵空树
num_nodes(self)	# 求树中结点个数
data(self)	# 取得树根存储的数据
first_child(self, node)	# 取得树中结点node的第一棵子树
children(self, node)	# 取得树中结点node的各子树的迭代器

```

set_first(self, tree)      # 用tree取代原来的第一棵子树
insert_child(self, i, tree) # 将tree设置为第i棵子树，其他子树顺序后移
traversal(self)            # 遍历树中各结点之数据的迭代器
forall(self, op)           # 对树中的每个结点的数据执行操作op

```

与二叉树的情况类似，上面的大部分操作都比较清晰，但遍历操作的情况不同。由于树具有复杂的结构，同样存在多种系统地遍历树中结点的方式。

树的遍历

在考虑树遍历问题时，因为还要遍历子树，必须关注操作的顺序。在这里合适的讨论对象是有序树，需要考虑其第一棵子树、下一棵子树，或者子树的序列等。

与二叉树一样，遍历就是访问树中所有结点且每个结点恰好访问一次的过程。遍历算法需要采用某种系统化的方式。考虑树遍历时，同样应该区分深度优先和宽度优先两类基本方式。进而，深度优先方式中还有多种不同的遍历顺序。

按深度优先或宽度优先的遍历方式访问结点，其操作过程就像是按照深度优先搜索或宽度优先搜索访问结点。实际上，在一般状态空间搜索问题里，搜索过程中经历的状态（看作结点）和状态之间的转移关系（看作边）形成了一棵“树”，称为搜索树。搜索过程就是按某种顺序“遍历”这棵树（虽然这棵树没有显式表示）。

首先考虑宽度优先遍历，或称按层次遍历，算法中需要一个队列：

开始：将树根加入队列。

过程：重复下面动作直至队列空：

- 弹出队列里的首结点并访问；
- 将该结点的子结点顺序加入队列。

按层次方式遍历图 6.27 中的树，得到的序列在图中给出。

与二叉树类似，树的深度优先遍历也存在多种方式，其差别也在于遍历中访问根结点信息的时刻。对于先根序和后根序，是在访问所有子树之前或者之后访问根；中根序在这里的意义不明确，例如，有些人将此定义为在访问第一棵子树之后访问根结点。

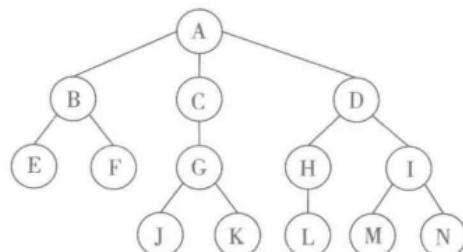
显然，先根序或后根序遍历算法都可以用递归或非递归的方式描述。在非递归描述的算法中，需要用一个栈保存已经发现但还来不及访问的分支。算法实现的主要问题是正确处理回溯，为此要记录所考察结点的已经访问分支或者下一个应该访问分支。这一问题在前面栈的应用（迷宫搜索）一节已有实例，请读者参考。不再进一步讨论。

图 6.27 也给出了采用先根序和后根序访问图中的树得到的序列。

6.7.4 树的实现

现在考虑树的实现技术。与二叉树类似，在这里也需要存储树的结点和结点之间的联系。但树的结构不如二叉树规整，实现时需要考虑这一情况。

此外，由于树结构比较复杂，因此带来了更多可能性。不同表示方法有各自的优点和缺点，应该根据实际情况和需要选择。常用的树表示方法有：



按层次：A, B, C, D, E, F, G, H, I, J, K, L, M, N
先根序：A, B, E, F, C, G, J, K, D, H, L, I, M, N
后根序：E, F, B, J, K, G, C, L, H, M, N, I, D, A

图 6.27 树的遍历

- 子结点引用表示法，父结点引用表示法。
- 子结点表表示法。
- 长子 - 兄弟表示法等。

子结点引用表示

树的最基本表示方法是子指针表示法，其基本设计与二叉树的链接表示法类似：用一个数据单元表示结点，通过结点间的链接表示树结构。但这里有一个麻烦：树结点的度数不定，而且结点的度数差可能很大。这一事实给表示树结点带来了很大困难。如果没有编程语言的支持，实现一般的树结构将牵涉到比较复杂的编程技术。在这种情况下，一种简单考虑是只支持度数不超过固定 m 的树，也就是说，树中分支结点至多允许 m 棵子树。图 6.28 描绘了这种树结点的布局。这里假设树结点有 k 个分支，用一个数据域记录这一信息。

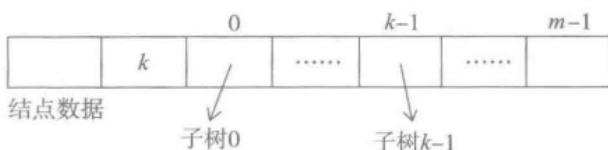


图 6.28 树结点的布局

采用这种结点布局，最大的缺点是会出现大量空闲的结点引用域。

性质 6.12 在 m 度结点表示的 n 个结点的树中，恰有 $n \times (m-1) + 1$ 个空树引用域。

子指针表示法的优点是直接反映了树的结构，操作方便灵活，能很好支持树结构变动。如果辅之以高级实现技术，可以表示任意复杂的树。在 Python 语言里实现树结构，可以采用前面基于 list 实现二叉树的技术，每个结点用一个 list 对象实现。有关工作留给读者作为练习。

父结点引用表示

为克服子结点引用法空间闲置大的缺点，人们考虑了父结点引用表示法。在任何树形结构（包括二叉树）里，除了树根之外的每个结点都有（且仅有）一个父结点。因此，如果在子结点里记录父结点关系，在每个结点里就只需要一个引用域。但要注意，这时所有的引用路径都将从叶结点开始，而子结点相互独立。这说明，仅靠父结点引用关系无法掌握整棵树。为解决这个问题，人们提出一种方法：用一个顺序表表示树，每个表元素对应于一个结点，包含两个部分：结点数据和父结点引用。

图 6.29 中给出了一个例子，其中的父结点引用采用顺序表下标的方式记录。在这里，通过连续表的整体结构可以掌握整棵树。为表示更多信息，树中结点通常在表里按某种遍历顺序排列。图 6.29 中的结点按层次序排列，做层次遍历只需顺序访问。

父结点引用技术的优点是存储开销小，除结点信息外，每个结点只需要一个父结点引用域，对 n 个结点的树，需要 $O(n)$ 附加空间，与树的度数无关。

由于结构中只记录了父结点关系，要想从父结点找到子结点，就必须通过查找过程，复杂度是 $O(n)$ 。由于采用连续表表示，插入和删除结点要解决一些管理问题，也有一些技术和设计问题。有关细节请读者考虑，可作为设计和编程练习。

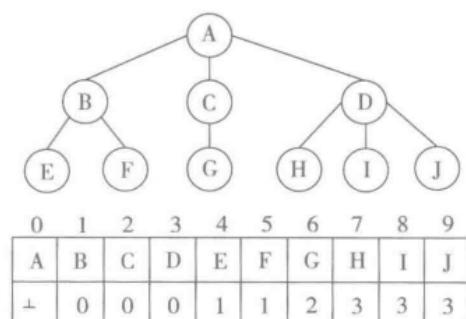


图 6.29 树的父结点引用表示

子结点表表示

另一种常见实现技术称为子结点表表示。其中用一个连续表存储树中各结点的信息，每个结点关联一个子结点表，记录树的结构。

图 6.30 给出的是图 6.29 中树的子结点表表示的示意图，其中子结点表用链接结构实现。实际上，也可以采用连续表结构。

在这种表示中有两种单元：一种单元表示结点，是结点数据和子结点表头指针的二元组；另一种是子结点表的结点单元。

与父结点引用表示法类似，通常树结点表中的结点也按某种遍历序排列。由于还有子结点表，通过它们可以直接找到相关子结点，各种操作实现都比较方便。

采用子结点表表示，每个结点需要一个子结点表引用域，每条边需要一个表结点。由于 n 个结点的树有 $n-1$ 条边，这两种开销都是 $O(n)$ ，需要增加这些存储，与树中结点的度数无关。

基于子结点表技术实现表类的工作也留作设计和编程练习。

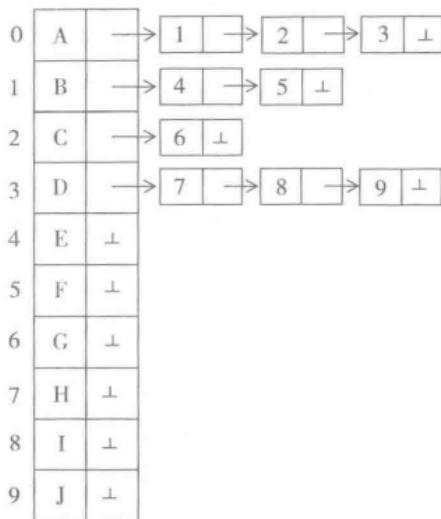


图 6.30 树的子结点表表示

长子 - 兄弟表示法

另一技术是长子 - 兄弟表示法，其实就是树的二叉树表示。采用这种表示，每个树结点对应于二叉树的一个结点。二叉树中结点 d 的左子结点是原树中 d 的第一个子结点，而二叉树中 d 的右子结点是原树中 d 的下一个兄弟结点。

这种表示的细节不再进一步讨论。下面考虑在 Python 中实现树的一种技术。

6.7.5 树的 Python 实现

由于 Python 提供了许多高级数据功能，在 Python 里实现树存在很多不同技术，有些技术非常方便。前一小节讨论了树的几种抽象实现技术，每种技术都可以在 Python 里落实，实现为一个树类。这里不准备讨论它们，而是考虑一种简单技术。

树的 Python 的 list 实现

树结点包含两部分信息：结点本身的数据以及一组子树。一种显然的 Python 实现方式是用二元组表示树，例如包含两个成员的元组或者包含两个成员的表。其中第一个成员表示结点，第二个成员是表示子树序列（这部分也可以用元组或者表实现）。如果要表示结点数据和 / 或子树有可能变动的树，在这里就应该用表 (list)。

下面考虑另一种方式，把结点数据和子树序列包装在一个表里。这一设计实际上是前面二叉树的 list 实现的直接扩充。首先定义一个构造这种树的函数：

```
class SubtreeIndexError(ValueError):
    pass

def Tree(data, *subtrees):
    return [data] + list(subtrees)
```

注意，这里的 `subtrees` 是一个序列参数，在函数调用中，它将约束到除去第一个实参之外

的其他实参的序列。序列也可迭代，因此可以作为 extend 的参数。

在下面实现中假设空树直接用 None 表示，还定义了另外几个操作函数：

```
def is_empty_Tree(tree):
    return tree is None

def root(tree):
    return tree[0]

def subtree(tree, i):
    if i < 1 or i > len(tree):
        raise SubtreeIndexError
    return tree[i + 1]

def set_root(tree, data):
    tree[0] = data

def set_subtree(tree, i, subtree):
    if i < 1 or i > len(tree):
        raise SubtreeIndexError
    tree[i+1] = subtree
```

下面是若干使用示例：

```
tree1 = Tree('+', 1, 2, 3)
tree2 = Tree('*', tree1, 6, 8)
set_subtree(tree1, 2, Tree('+', 3, 5))
```

这里还是采用了前面 6.2 节表达式树的想法，叶结点直接用数据项表示。上面操作构造了两个表达式，其中把加法和乘法看作多元运算符。由于这种树直接用 Python 的 list 实现，可以用标准函数 print 查看树的情况。

其他操作留给读者完成。

定义树类

前面讨论了一种技术，把树结构直接映射到嵌套的表。由于 Python 的表可以自动扩展，这种树中的结点没有度数限制，数据可以修改，子树也可以增删修改。

但如前所述，这样构造出的树不是一个特殊的树类型的对象（只是特殊形式的嵌套表），因此不能得到 Python 语言类型系统的支持。根据前面的经验，这件事不难解决，可以采用上述设计作为内部表示，定义出一个树类，由这个类提供各种操作树的方法。有了前面实现二叉树类型的经验，这个工作不难完成。

在同样的外部接口之下，一个类的内部表示可以采用不同的设计。前面讨论中提出，一个树结点的相关信息包括两部分，一项结点数据和一组子树。可以按这种看法设计树对象中的基本结点构件，其中采用一个数据域和一个子结点表域。

```
class TreeNode:
    def __init__(self, data, subs=[]):
        self._data = data
        self._subtrees = list(subs)

    def __str__(self):
        return "[TreeNode {} {}]".format(self._data,
                                         self._subtrees)
```

这里的 `__str__` 方法将自动被 `str(...)` 标准函数调用，主要用于显示和输出。

相应的树类不难定义，下面只给出了它的初始化函数：

```
class Tree:  
    def __init__(self):  
        self._root = None  
    # .....  
    # .....
```

完成这个类的工作也留给感兴趣的读者。

本章总结

树形结构是计算中使用广泛的一种较为复杂的数据结构。一方面，它具有层次性，同时又可以表达数据之间的一对多关系，因此可以用于描述更多实际数据的有关信息。著名的 XML 数据即是一种基于树结构表示的标准数据形式，正在被广泛用于各种领域的数据表示、存储和交换。另一些与 XML 类似的数据表示形式也在发展。此外，树形结构也被广泛用于各种重要软件的内部，例如操作系统管理下的文件系统，高级语言处理系统中的各种基本数据表示。实际上，高级语言程序（包括 Python 程序）本身就是树形结构，是一层层语言结构的嵌套。高级语言里的表达式、控制结构、作用域等也都是嵌套的树形结构。这些情况都说明，树形结构具有非常重要的理论价值和实用价值。

树形结构是典型的递归结构，采用递归的方式可以比较简洁自然地描述树形结构操作和处理算法。另一方面，如果需要，也可以借助于栈或队列辅助结构，写出处理树形结构的非递归方式算法。本章中，特别是在讨论树遍历时，给出了一些处理树形结构的算法实例。

在处理包含许多数据元素的结构时，都需要考证逐个访问其中元素的过程，这就是遍历。对于树这样的比较复杂的结构，存在着多种不同的遍历方法。系统化的遍历方法首先分为深度优先和宽度优先两类。根据遍历访问根结点信息的时机，深度优先遍历又有多种不同方式，主要是先根序、中根序和后根序三种方式。其中先根序处理最为简单，而后根序相对复杂，特别是实现后根序处理的非递归算法。但实际上有许多处理需要按后根序的方式进行，例如在第 5 章和本章里都有讨论的表达式求值。

实际上，树遍历也就是一种状态空间搜索。树的深度优先处理过程对应于状态空间的深度优先搜索，树的宽度优先处理过程对应于状态空间的宽度优先搜索。对任何状态空间搜索，探索路径的整体也形成了一种树形，称为搜索树。

树形结构也有许多不同的子类，其中二叉树和一般的树是最重要的树形结构。这两种结构之间有一种等价关系，有序树林（一组有序树的有序汇集）与二叉树之间可以相互转换。在实际中，人们经常采用二叉树作为基础实现树形结构。

二叉树和树都有许多重要应用。本章中介绍了表达式树、堆和优先队列、哈夫曼树和哈夫曼算法等。还有许多实际应用，如用于数学表达式、离散事件模拟等。

二叉树和树都有多种不同的实现方式，使用最多的是结点链接表示。一般树的结构不规则，不同结点的度数可能相差很大。为了在空间占用量和操作方便性之间取得某种平衡，人们为树的实现提出了多种不同的方法，也经常利用树与二叉树之间的等价关系表示树。

练习

一般练习

1. 复习下面概念：树形结构，树根，前驱，后继，二叉树，左子树和右子树，空树，单点树，

父结点 / 子结点, 左 / 右子结点, 父子关系, 祖先 / 子孙关系, 祖先 / 子孙结点, 树叶, 分支结点, 结点的度数, 路径, 路径长度, 结点的层数, 树的高度 (深度), 满二叉树, 扩充二叉树, 内部结点和外部结点, 内部路径长度和外部路径长度, 完全二叉树, 二叉树的遍历, 深度优先遍历, 先根序 (先序) 遍历, 中根序 (对称序 / 中序) 遍历, 后根序 (后序) 遍历, 先根序列, 对称 (中根) 序列, 后根序列, 宽度优先遍历 (按层次顺序遍历), 层次序列, 表达式树, 二元表达式, 算术表达式, 表达式求值, 优先队列, 优先关系, 优先级, 堆, 堆序, 小顶堆, 大顶堆, 筛选 (向下筛选, 向上筛选), 堆排序, 离散事件模拟, 模拟框架, 事件队列, 宿主系统, 二叉树的链接实现, 二叉树结点类, 非递归的二叉树遍历算法, 非递归后序遍历, 下行循环, 二叉树类, 带权扩充二叉树, 哈夫曼树 (最优二叉树), 哈夫曼算法, 哈夫曼编码, 最优编码, 树和树林, 树根, 有序树和无序树, 有序树林和无序树林, 搜索树, 子结点引用表示法, 父结点引用表示法, 长子 - 兄弟表示法。

2. 三个结点 a、b、c 可以构造出多少棵不同的二叉树? 请画出它们。
3. 四个结点可以构造出多少棵不同的 (一般的) 树? 请画出它们。
4. 若一棵二叉树有 10 个度为 2 的结点, 6 个度为 1 的结点, 它有几个叶结点?
5. 已知一棵二叉树有 36 个叶结点, 这棵树的全部结点至少有多少个?
6. 将二叉树的概念推广到三叉树, 一棵 244 个结点的三叉树至少有多高?
7. 请根据二叉树的几种结构形态, 严格证明扩充二叉树的外部路径长度和内部路径长度之间的关系式: $E=I+2\times n$ 。
8. 二叉树采用嵌套的 list 表示时, 空表表示空树。请总结出空表的个数与树中分支结点和叶结点个数的关系公式, 并严格证明这一关系公式。
9. 对图 6.31 中的二叉树, 请:
 - a) 做出其先根序、中根序和后根序序列;
 - b) 将其转换为树 (或树林), 而后做出得到的树 (或树林) 的先根序、中根序和后根序序列;
 - c) 比较和讨论上面两组遍历得到的结果。
10. 对上题图示的二叉树, 做出其扩充二叉树, 并求出这棵扩充二叉树的内部路径长度和外部路径长度。
11. 已知一个算术表达式的中缀形式为 “ $A+B*C-D/E$ ”, 后缀形式为 “ $ABC*+DE/-$ ”, 请给出其前缀形式。
12. 如果知道了一棵二叉树的先根序列和后根序列, 能够确定原来的二叉树吗? 如果能请证明之, 不能请举出反例。
13. 确定所有满足下面条件的二叉树:
 - a) 其先根序列与中根序列相同;
 - b) 其后根序列与中根序列相同;
 - c) 其后根序列的反转与中根序列相同;
 - d) 其先根序列与后根序列相同。
14. 满足什么条件的非空二叉树的先根序列正好等于其后根序列的反转?
15. 请证明下面结论: 在一棵树的先根序列、中根序列和后根序列中, 所有树叶结点的相对位置都一样。

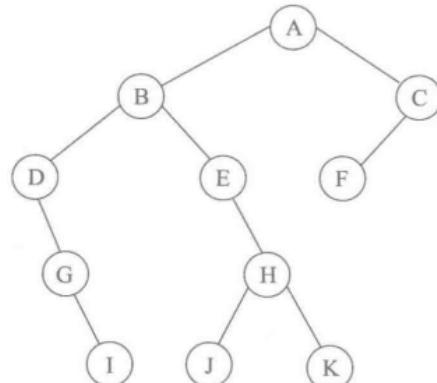


图 6.31 题 9 和题 10 中使用的二叉树

16. 什么是前缀编码？说明如何利用二叉树设计二进制的前缀编码？其实前缀编码并不限于二进制编码，请考虑如何推广这个概念。
17. 下面几个符号编码集合中，哪些是或不是前缀编码：
- a) {0, 10, 110, 11101, 10111}
 - b) {11, 10, 001, 101, 0001}
 - c) {00, 010, 0110, 1010, 1001}
 - d) {b, c, aa, ac, aba, abb, abc}
18. 假设通信中使用的字符 a、b、c、d、e、f、g、h、i、j、k 在电文中出现的频率分别是 3、8、5、17、10、6、19、15、23、16、9，请做出相应的哈夫曼编码。
19. 假设用于通信的电文都由 8 个字母组成，这些字母在电文中出现的频率分别为 7、19、4、6、32、3、21、12。请做出 8 个字母的哈夫曼编码。另外，采用 0 ~ 7 的二进制表示形式是另一种编码方案。请比较这两种编码方案的优缺点。
20. 请证明哈夫曼算法的正确性，即通过该算法构造出的二叉树一定是哈夫曼树。
21. 在一棵树中只有度数为 $k \geq 1$ 的结点和度数为 0 的叶结点（也就是说，它是一棵满的 k 叉树）。已知树里有 m 个度数为 k 的结点，它有多少个叶结点？
22. 设树林 F 包含三棵树，其结点个数分别为 N_1 、 N_2 和 N_3 。 T 是与 F 对应的二叉树， T 的根结点的右子树有多少个结点？
23. 假设树 T 包含 n_1 个度数为 1 的结点， n_2 个度数为 2 的结点，…， n_m 个度数为 m 的结点，该树有多少个叶结点？
24. 请考虑如何写出一个算法判断一棵二叉树是否为满二叉树，另外设计一个算法判断一棵二叉树是否为完全二叉树。
25. 假设现在需要交换一棵二叉树中各结点的左右子树，请设计完成这一操作的递归算法和非递归算法。

编程练习

1. 请基于 6.2 节的讨论，基于 Python 的 list 实现一种二叉树类。
2. 请完成 6.2.2 节的二元表达式求值器。
3. 请在表达式的 list 表示基础上完成下面工作：
 - a) 定义函数 variables(exp)，它求出表达式 exp 里出现的所有变量的集合；
 - b) 扩充本章函数 eval_exp，增加一个参数 values，对应实参应是一个字典，以可能在表达式里出现变量为关键码，关联到变量的值。这个 eval_exp 求值中遇到变量时检查该字典，用字典提供对应值取代这个变量；
 - c) 设 exp 是可以包含变量的代数表达式（只有加和乘运算），请定义函数 derive(exp, var)，它给出 exp 对变量 var 的导函数表达式。
4. 基于上述工作实现一个交互式二元表达式计算器，其中定义一种（程序）变量机制，可用于记录前面输入的表达式或已做计算的值。
5. 参考 6.2.2 节最后的讨论，基于上一题的工作扩充，实现一个具有更广泛可用性的表达式计算器。
6. 请定义一个二元表达式类，在其中实现一些重要的表达式计算。
7. 6.3.2 节讨论了基于线性表的优先队列实现。其中在提出方案 2 的最后，概述了一种临时记录检索到的最优先元素的技术。请进一步开发这种技术，实现一个采用这种技术的优先队列类型。请仔细分析该节中提出的几种技术的优缺点。

8. 请修改海关检查站模拟系统，改用每个检查通道一个等待队列的管理策略。对这种新策略做一些模拟，并将模拟结果与共用等待队列的策略比较。
9. 假定一个银行网点有 4 个服务柜台，每个柜台前可能有一个等待队列。顾客到达时如果有空闲柜台就直接去办理业务，没有空柜台就选择当时最短的队列排队等待。此外，如果某柜台业务员空闲且其等待队列已空，该业务员将从当时有人的某个队列叫一个顾客过来。请做一个系统模拟该网点的运转：先选择一组可以根据情况设定的模拟参数，而后开发这个系统。最后选择几组不同的参数做一些模拟。
10. 6.5.2 节定义了一个输出二叉树的函数。请写一个读入这种输出形式，构造出相应的二叉树的函数。假设结点里保存的数据是整数。
11. 修改 6.5.2 节给出的非递归定义的先根序遍历函数，只在右子树非空时才将其进栈。考虑这时的循环条件和函数内的语句，尽可能做些优化。
12. 请设法定义非递归的先根序和中根序遍历函数，使栈空间的使用达到最少。请论证所定义函数确实具有这种性质。
13. 请为 6.5.2 节的二叉树类增加下面方法，其中相等判断、拷贝、结点计数等请考虑递归和非递归两种实现，还请考虑能否借助类中已有的遍历操作实现：
 - a) 一个层次序的遍历方法；
 - b) 方法 `__eq__(self, another)` 判断另一棵树 `another` 是否与 `self` 相等。两棵树相等当且仅当其结构相同且每个结点的数据相等；
 - c) 方法 `clone(self)` 生成 `self` 的一个拷贝；
 - d) 方法 `count_nodes(self)` 返回树中叶结点个数和非叶结点个数的序对。
14. 设二叉树不同结点的标识唯一。假定有了一棵二叉树的先序序列和中序序列，分别用 Python 的表表示，请定义一个函数生成该二叉树的嵌套表形式。
15. 请将 Python 的 `list` 或 `tuple` 作为内部表示，定义一个类实现二叉树数据类型。并请对比这种实现与本章中定义结点类和二叉树类的实现。
16. 请考虑 6.5.3 节最后提出的带父结点链接的二叉树实现方法，定义这种二叉树类，并认真研究这种类上的算法。请尽可能地利用新增加的指针提高操作效率，并设法减少算法的辅助空间开销。
17. 定义一个函数，对任何给定的字符集及其出现概率，生成对应的哈夫曼编码。
18. 请基于 Python 的 `list` 对象实现一个基于父结点引用的树类，参考树抽象数据类型类型，实现相关操作。
19. 请基于子结点表技术的基本思想，设计和实现一个树类，定义必要的操作。
20. 请完成 6.7.5 节提出的借助于 Python 语言内置类型 `list` 的树实现，定义必要的操作。也可以自己做出另一种设计。
21. 6.7.5 节的最后提出了两种树类的实现技术。请参考那里的讨论，实现一个或者两个完整的树类，实现必要的操作。
22. 假设需要保存的元素分为 5 个不同的优先级，但希望相同优先级的元素能按照 FIFO 顺序被取出使用。请设计并实现一种能满足这一需要的数据结构。

第7章 图

图是一种抽象的数学结构，研究抽象对象之间的一类二元关系及其拓扑性质。数学领域里有一个称为“图论”的研究分支，专门研究这种拓扑结构。

在计算机的数据结构领域和课程里，图被看作一类复杂数据结构，可用于表示具有各种复杂联系的数据集合。图结构在实际中应用非常广泛。

由于图的结构比较复杂，其中有许多重要的计算问题，人们针对这些问题提出了许多重要而有趣的算法。图算法是算法研究的一个重要分支，有许多论文和专著。也由于图结构的复杂性，因此人们开发了许多实现方式，不同方式互有长短。本章将介绍图的基本知识、一些基本实现方法、若干基本计算问题和几个重要算法。

7.1 概念、性质和实现

7.1.1 定义和图示

一个图是一个二元组 $G=(V, E)$ ，其中：

- V 是非空有穷的顶点集合（也可以有空图的概念，但实际意义不大）。
- E 是顶点偶对（称为边）的集合， $E \subseteq V \times V$ 。
- V 中的顶点也称为图 G 的顶点， E 中的边也称为图 G 的边。

顶点是图中的基本个体，可以表示任何讨论中需要关心的实体。

图分为有向图和无向图两类。有向图中的边有方向，是顶点的有序对；无向图中的边没有方向，是顶点的无序对。

在下面讨论中，有向图里的有向边将用尖括号形式表示，例如 $\langle v_i, v_j \rangle$ 表示从顶点 v_i 到顶点 v_j 的有向边，其中 v_i 称为这条边的始点， v_j 是该边的终点。在有向图里， $\langle v_i, v_j \rangle$ 和 $\langle v_j, v_i \rangle$ 表示图中两条不同的有向边。无向图中的无向边用圆括号形式表示，在这里 (v_i, v_j) 和 (v_j, v_i) 表示的是同一条无向边。

如果在图 G 里有边 $\langle v_i, v_j \rangle \in E$ （对于无向图，有 $(v_i, v_j) \in E$ ），则称顶点 v_j 为 v_i 的邻接顶点或邻接点（无向图里的邻接关系是双向的），也称这条边为与顶点 v_i 相关联的边，或者 v_i 的邻接边。边集合 E 表示的是顶点之间的邻接关系。

在下面讨论中，对于所关注的图有两个限制：①这里不考虑顶点到自身的边，也就是说，若 $\langle v_i, v_j \rangle$ 或 (v_i, v_j) 是图 G 的边，则要求 $v_i \neq v_j$ ；②同一对顶点之间没有重复出现的边，若 $\langle v_i, v_j \rangle$ 或 (v_i, v_j) 是图 G 的边，那么它就是这两个顶点之间唯一的边。去掉这些限制将得到另一类稍微不同的数学对象，也可以研究它们。

【例 7.1】 令 $G_1=(V_1, E_1)$ ，其中 $V_1=\{a, b, c\}$ 是 3 个顶点的集合， $E_1=\{\langle a, b \rangle, \langle b, c \rangle, \langle c, b \rangle\}$ 。再令 $G_2=(V_2, E_2)$ ，其中 $V_2=\{1, 2, 3, 4\}$ 是 4 个顶点的集合， $E_2=\{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)\}$ 。 G_1 是一个以 a, b, c 为顶点的有向图，而 G_2 是一个以 1、2、3、4 为顶点的无向图。

图的图示

图的上述形式化定义很清晰，按这种定义描述的图易于用机械方式处理（例如送给计算机

处理), 但人看起来不那么直观。为了方便理解, 人们提出了图的一种图示形式。在学习图的基本理论和算法处理时, 这种图示很有帮助。每个图与其图示之间一一对应, 在下面讨论中把它们等同对待, 画出一个图示时说它是一个图。

在图的图示中, 顶点用小圆圈表示, 顶点的标记写在顶点旁边, 边用顶点之间的连线表示。有向图里的边用带单向箭头的连线表示, 无向图里的边用简单连线表示。

【例 7.2】 图 7.1 中画出了例 7.1 里定义的两个图, 左边是有向图 G_1 , 右边是无向图 G_2 。注意, 在这种图示里, 顶点的位置、顶点之间的距离、不同的边之间有没有交叉或者如何交叉等都无关紧要, 关键是存在哪些顶点、顶点间有怎样的邻接关系。这里只关心拓扑结构。

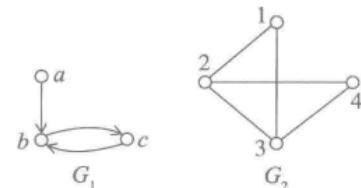


图 7.1 两个图的实例

7.1.2 图的一些概念和性质

下面介绍图的一些基本概念和性质。

完全图: 任意两个顶点之间都有边的图(有向图或无向图)称为完全图。显然:

- n 个顶点的无向完全图有 $n \times (n-1)/2$ 条边。
- n 个顶点的有向完全图有 $n \times (n-1)$ 条边。

应该注意一个重要事实: $|E| \leq |V|^2$, 即 $|E|=O(|V|^2)$ 。也就是说, 对于不同的图, 其中边的条数可能达到顶点个数的平方, 但也可能很少。这个事实在考虑与图有关的算法的时间和空间复杂性时经常用到。

【例 7.3】 图 7.2 给出了两个完全图, G_3 是有向完全图, G_4 是无向完全图。

度(顶点的度): 一个顶点的度就是与它邻接的边的条数。对于有向图, 顶点的度还分为入度和出度, 分别表示以该顶点为始点或者终点的边的条数。

性质 7.1 (顶点数、边数和顶点度数的关系) 无论对于有向图还是无向图, 顶点数 n 、边数 e 和顶点度数满足下面关系:

$$e = \frac{1}{2} \sum_i D(v_i)$$

其中 $D(v_i)$ 表示顶点 v_i 的度数, 这里要求对所有顶点的度数求和。

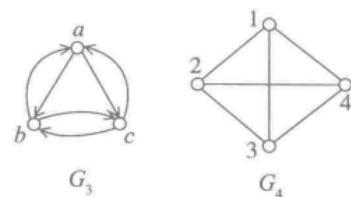


图 7.2 两个完全图

路径和相关性质

路径是图的一个重要概念。对于图 $G=(V, E)$, 如果存在顶点序列 $v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_m}$, 使得 $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_{m-1}}, v_{i_m}) \in E$ (它们都是图 G 的边, 对于有向图是 $\langle v_{i_1}, v_{i_2} \rangle, \dots, \langle v_{i_{m-1}}, v_{i_m} \rangle \in E$), 则说从顶点 v_{i_0} 到 v_{i_m} 存在路径, 并称 $\langle v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_m} \rangle$ 是从顶点 v_{i_0} 到 v_{i_m} 的一条路径。

下面是一些与路径相关的概念:

- 路径的长度就是该路径上边的条数。
- 回路(环)指起点和终点相同的路径。
- 如果一个环路除起点和终点外的其他顶点均不相同, 则称为简单回路。
- 简单路径是内部不包含回路的路径。也就是说, 该路径上的顶点除起点和终点可能相同外, 其他顶点均不相同。因此, 简单回路也是简单路径。

注意，从一个顶点 v_i 到另一顶点 v_j 可能存在路径，也可能不存在路径。如果存在路径，则可能有一条或者多条路径，不同路径的长度也可能不同。特别的，如果从 v_i 到 v_j 存在非简单的路径（即包含环路的路径），那么从 v_i 到 v_j 就有无穷多条不同的路径，其中一些路径之间的不同就在于在环路上绕圈的次数不同。

有根图：如果在有向图 G 里存在一个顶点 v ，从顶点 v 到图 G 中其他每个顶点均有路径，则称 G 为有根图，称顶点 v 为图 G 的一个根。

例如，前面给出的有向图 G_1 和 G_3 均为有根图。其中图 G_1 的根是顶点 a ，图 G_3 的三个顶点都是根。这也说明，有根图中的根可能不唯一。

连通图

连通：如果在无向图 G 中存在从 v_i 到 v_j 的路径（它显然也是从 v_j 到 v_i 的路径），则说从 v_i 到 v_j 连通（显然，由于是无向图，从 v_j 到 v_i 也连通，因此也说 v_i 与 v_j 之间连通），或者说从 v_i 可达 v_j 。为简化讨论，下面将始终默认任一顶点 v_i 与其自身连通，可达其自身。对有向图，连通性也可以类似地定义，但这里的连通性可以不是双向的。

连通无向图：如果无向图 G 中任意两个顶点 v_i 与 v_j 之间都连通，则称 G 为连通无向图。

强连通有向图：如果对有向图 G 中任意两个顶点 v_i 和 v_j ，从 v_i 到 v_j 连通并且从 v_j 到 v_i 也连通（请注意，有向图里的路径有方向，这里要求两顶点间两个方向的路径都存在），则称 G 为强连通有向图。

显然，完全无向图都是连通图，完全有向图都是强连通有向图。但反过来不对，存在非完全的连通无向图和强连通有向图。

【例 7.4】 图 7.3 里的 G_5 是一个强连通有向图， G_6 是一个连通无向图，它们都不是完全图。

性质 7.2（最小连通无向图的边数）包含 n 个顶点的最小连通无向图 G 恰有 $n-1$ 条边。

最小连通图，即其为连通图，但去掉其中任一条边将不再是连通图。通过对 G 的顶点个数做数学归纳法，很容易证明这个性质。

性质 7.3（最小有根图的边数）包含 n 个顶点的最小有根图（即去掉任一条边将不再是有根图）恰好包含 $n-1$ 条边。可以类似证明。

从某种角度看，第 6 章里讨论的树可以看作图的一个子类。满足性质 7.2 的图 G 称为无向树。无向树中的任何一个顶点都可以看作树的根，从它出发可以到达任何其他顶点。满足性质 7.3 的有向图称为有向树，这个有根图的根可以看作树根，树中存在从树根到其他每一个顶点的路径。

子图、连通子图

对于图 $G=(V, E)$ 和图 $G'=(V', E')$ ，如果 $V' \subseteq V$ 而且 $E' \subseteq E$ ，就称 G' 是 G 的一个子图。特别的，根据定义， G 也是其自身的子图。

子图就是原图的一部分，而且需要满足图的基本定义，主要是其中任一条边的两个顶点必须都在这个子图中。

【例 7.5】 图 7.4 给出了有向图 G_1 的几个子图，其中的竖虚线只是为了分隔不同的子图，没有其他意义。当然， G_1 也是自身的子图。另外，图 G_1 也是完全图 G_3 的子图。图 7.3 里的

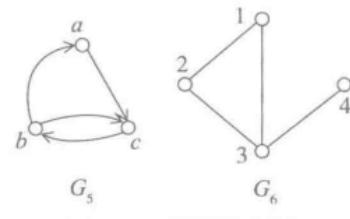


图 7.3 连通图的例子

G_5 和 G_6 分别是前面 G_3 和 G_4 的子图。

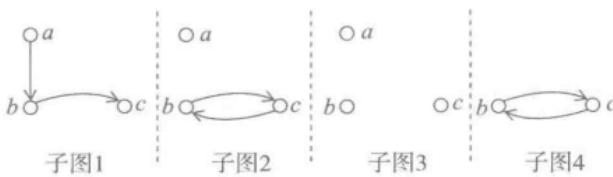


图 7.4 图 G_1 的三个子图

一个图可能不是连通图（或强连通图），但它的一些子图则可能是连通的（或强连通的），这种子图称为原图的连通子图（对于有向图，是强连通子图）。

图 G 的一个极大连通子图（连通分量） G' 是图 G 的一个连通子图，而且 G 中不存在真包含 G' 的连通子图。也就是说，从连通性的角度看，图 G' 的顶点和边集合都已经不能扩充，是极大的（如果增加顶点就会不连通，也没有其他边可加）。如果图 G 本身连通，它将只有一个连通分量，就是 G 本身。如果 G 不连通，则其连通分量多于一个。

不难看到，无向图 G 的所有连通分量形成了图 G 的一个划分，每个连通分量包含图 G 的一集顶点和它们之间的所有边，不同连通分量的顶点集合互不相交，而这些顶点集和边集的并就是原来的 G 。

与上面定义类似，有向图 G 的一个极大强连通子图称为它的一个强连通分量。但请注意，图 G 的强连通分量只形成其顶点的一个划分，所有强连通分量的未必等于图 G ，可能少一些连接不同连通分量的有向边。这一点与无向图不同。

为帮助理解有向图的情况，考虑图 7.4 里的几个有向图（图 G_1 的子图）。子图 4 是个强连通有向图，只有一个连通分量。子图 2 不是强连通图，但请注意，由于顶点总与其自身连通，因此顶点 a 自身也是该图的一个子图，是它的一个强连通分量。因此子图 2 包含两个强连通分量。类似的，子图 3 包含 3 个强连通分量。最后看子图 1，其中任意两个顶点都不相互连通，因此它有 3 个强连通分量，但图里的两条边不属于任何一个分量。从子图 1 还看到了一个情况：有向图的强连通分量之间可能有单向连通。

带权图和网络

如果图 G 中的每条边都被赋予一个权值，则称 G 为一个带权图（可以是带权有向图或者带权无向图）。边的权值可用于表示实际应用中与顶点之间的关联有关的某些信息。带权的连通无向图也被称为网络。

【例 7.6】 图 7.5 给出了两个带权图，图中标在边旁的数表示该边的权。 G_7 是一个带权有向图， G_8 是一个带权无向图（网络）。

带权图和网络都是实际应用非常广泛的图结构，后面将讨论它们的一些重要性质、计算问题和相关算法，从中也可以看到它们的一些可能应用。

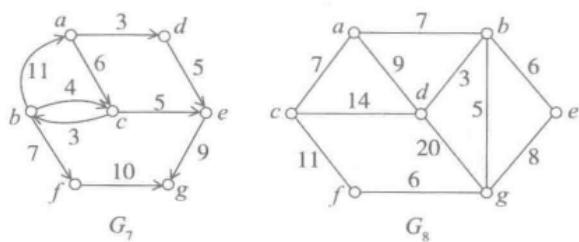


图 7.5 带权图实例

7.1.3 图抽象数据类型

上面一直把图作为一个抽象的数学结构，介绍图和与图有关的一些数学定义，讨论并证明了图的一些性质。就本书的目标而言，主要还是把图看作一种数据结构，希望研究它的实现、

操作和图上的一些计算问题。

首先考虑定义一个图抽象数据类型。作为一种复杂的数据结构，可以给图定义许多有用的操作。下面定义中给出了一些操作。

ADT Graph:	
Graph(self)	# 一个图抽象数据类型
is_empty(self)	# 图构造操作，创建一个新图
vertex_num(self)	# 判断是否为一个空图
edge_num(self)	# 获得这个图中的顶点个数
vertices(self)	# 获得这个图中边的条数
edges(self)	# 获得这个图中的顶点集合
add_vertex(self, vertex)	# 获得这个图中的边集合
add_edge(self, v1, v2)	# 将顶点vertex加入这个图
get_edge(self, v1, v2)	# 将从v1到v2的边加入这个图
out_edges(self, v)	# 获得v1到v2边有关的信息，没有时返回特殊值
degree(self, v)	# 取得从v出发的所有边
	# 检查v的度

创建图可以是创建一个空的图，或是基于有关图中顶点的一批数据，创建一个只包含这些顶点但是没有边的图，或者基于顶点和边的数据创建一个新图。可以根据具体情况考虑，为图的构造函数引进其他必要的参数。

上面的抽象数据类型只是作为例子，在实际中要根据应用的需求考虑操作集合。定义操作时还需要考虑是无向图还是有向图。例如，对有向图，顶点的度应该分为出度和入度，邻接边也需要区分是出边还是入边。实际中一般只使用出度和出边。

除了上面的操作外，还需考虑遍历图中所有顶点或者所有边的操作。这里的遍历与树的遍历有许多差异，最重要有两点：

- 图中可能有回路，到同一个顶点可能有多条路径（树结构里没有这些情况）。因此，在遍历中需要避免再次进入已经遍历过的部分。
- 图有可能不连通，或者这个图不是有根图，即使是有根图，算法也可能没有从图中的根开始遍历。因此，要完成对一个图里所有顶点（或者边）的遍历，遍历完图中可达的那个部分（在无向图里是一个连通分量，在有向图里是一个或几个强连通分量），并不一定是整个图遍历的结束，还需要考虑从初始顶点不可达的部分。

下面将会看到，图遍历问题实际对应于一般的状态空间搜索问题。

7.1.4 图的表示和实现

图的结构比较复杂，在其表示中需要表示顶点及顶点间的边。一个图可能有任意多个顶点（但有穷），图中任意两个顶点之间都可能存在边。顶点就是一个集合，每个顶点可能保存一些由实际应用确定的信息，在这里不是重点。在抽象的讨论中，主要关注的是边与顶点的邻接关系和顶点间的邻接关系，这些是图表示的关键。

在一个图里可能包含很复杂的结构关系，其表示和实现方法自然可以有许多变化。此外，图结构需要支持比较大的一集操作，采用不同实现方法，各种操作的效率可能差别很大。实际中需要实现一些图上的算法，各种常用操作在不同算法里的重要性也可能不同。由于这些情况，人们为图的实现开发了许多技术，也不难设计出别的实现技术。下面将介绍几种最基本的技术，实际需要时，应根据具体应用的情况做出合理选择。

邻接矩阵

图的最基本表示方法是邻接矩阵表示法，邻接矩阵是表示图中顶点间邻接关系的方阵。对于 n 个顶点的图 $G=(V, E)$ ，其邻接矩阵是一个 $n \times n$ 方阵，图中每个顶点（按顺序）对应于矩阵里的一行和一列，矩阵元素表示图中的邻接关系。

最简单的邻接矩阵是以 0/1 为元素的方阵。对于图 G ，其邻接矩阵是

$$A_{ij} = \begin{cases} 1, & \text{如果顶点 } v_i \text{ 到 } v_j \text{ 有边} \\ 0, & \text{如果顶点 } v_i \text{ 到 } v_j \text{ 无边} \end{cases}$$

对于带权图，其邻接矩阵元素的定义是

$$A_{ij} = \begin{cases} w(i, j), & \text{如果顶点 } v_i \text{ 到 } v_j \text{ 有边，且该边的权是 } w(i, j) \\ 0 \text{ 或 } \infty, & \text{如果顶点 } v_i \text{ 到 } v_j \text{ 无边} \end{cases}$$

对于无边的情况，是以 0 还是 ∞ 为值，应根据实际需要确定 (∞ 用某个特殊值表示)。有向图或无向图都可以这样表示，无向图中的一条边对应于两个矩阵元素。

邻接矩阵表示（而且只表示）了图中的顶点数和顶点间的联系（边），每个顶点对应矩阵的一个行列下标，通过一对下标可以确定图中一条边的有无（或者取得该边的权）。如果顶点本身还有其他信息，就需要另行表示，例如可以考虑另建一个顶点表，用与矩阵同样的下标引用相应的表元素。

【例 7.7】 图 7.3 的两个图的邻接矩阵分别为（其中令 $a/b/c$ 分别对应下标 1/2/3）：

$$A_{G_3} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$A_{G_4} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

易见，无向图的邻接矩阵都是对称矩阵，因为其邻接关系是对称的。有向图就不一定是这样。另外，由于没考虑顶点到自身的边，上面矩阵的对角线元素都是 0。如果希望用矩阵表示图中的连通关系，就应该令矩阵的对角线元素为 1，因为默认各顶点到自身连通。此外，邻接矩阵中的标号 i 的一行和一列都对应于顶点 i ，行对应于它的出边，列对应于它的人边，行 / 列中非零元的个数对应于顶点 i 的出度 / 入度。

【例 7.8】 图 7.5 中带权图 G_7 的邻接矩阵为（其中令 $a/b/c/\cdots$ 对应于下标 1/2/3/ \cdots ）：

$$A_{G_7} = \begin{pmatrix} 0 & \infty & 6 & 3 & \infty & \infty & \infty \\ 11 & 0 & 4 & \infty & \infty & 7 & \infty \\ \infty & 3 & 0 & \infty & 5 & \infty & \infty \\ \infty & \infty & \infty & 0 & 5 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & 9 \\ \infty & \infty & \infty & \infty & \infty & 0 & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

用邻接矩阵表示带权图时，出现了两个新问题：首先是无边情况的表示，可以根据情况选择 0

或 ∞ ；还有就是对角线元素的表示问题，也可以根据情况选择 0 或者 ∞ ，其选择可以与无边情况相同或者不同。如果图中的权表示从顶点到顶点的一种代价，无边时的代价无穷大，而顶点到自身的代价是 0。上面邻接矩阵反映了这些情况。

为符合 Python 语言的习惯，在下面讨论中，矩阵的行列下标从 0 开始编号。

邻接矩阵表示法的缺点

从上面例子可以看到，图的邻接矩阵经常是比较稀疏的，其中有信息的元素比例不大，大量元素是表示无边的值。对实际应用中很大的图，这种情况更加明显。举个例子，假设现在需要表示中国的铁路线路图。目前中国铁路车站大约有 6000 个，对应的图中需要有 6000 个顶点，用边表示车站之间有铁路相连。显然，绝大部分车站对应的顶点度数为 2，因为它们只与另外两个车站有联系。这种情况说明，这个图的边数大约是顶点数的 2 倍。用邻接矩阵表示，矩阵里大约有 6000^2 个元素，而其中只有大约 6000×2 个元素有实际信息，99.96% 的元素只是为了说明相应的边不存在。

在实际应用中，很多图都是这种情况，其中的边数与顶点数呈线性关系，而不是平方关系。采用邻接矩阵表示这种图，空间浪费将非常大。进一步考虑，这种情况对程序（算法）的实际时间开销也有影响。如果在计算过程中需要遍历图中的所有边，采用邻接矩阵表示，检查所有矩阵元素，至少要花费等比于顶点数平方的时间。

为了降低图表示的空间代价（在一些情况下也提高计算的效率），人们提出了许多其他技术，它们都可看作邻接矩阵的压缩版，如：

- 邻接表表示法；
- 邻接多重表表示法；
- 图的十字链表表示；等等。

下面主要介绍邻接表表示法。

在 Python 语言里实现图，也有许多具体方法。例如，可以用 Python 语言的内置数据类型直接实现邻接矩阵或者邻接表，还可以考虑其他方法。具体情况后面讨论。

图的邻接表表示

所谓邻接表，就是为图中每个顶点关联一个边表，其中记录这个顶点的所有邻接边。这样，一个顶点的表，其中每个顶点又关联一个边表，就构成了图的一种表示。具体实现可以采用链接表或者连续表，这些问题前面已经讨论。采用某种具体形式的顶点表和边表，就形成了一种特殊的邻接表表示。

顶点是图中最基本的成分，通常有标识，也可以顺序编号，以便可以通过编号随机访问。而边是图中的附属部件，经常需要访问一个顶点的各条边。另一方面，图中的顶点通常不变化，而边的增减情况较多。由于这些情况，实际中人们经常用一个顺序表表示图中顶点，每个顶点关联一个表示其邻接边表的链表，这就形成图 7.6 所示的结构。这里给出的是图 G_7 的出边表表示。在对应顶点 i 的边表里，每个表结点对应一条边，结点里记录该边终点的下标。一个顶点关联的边表长度就是它的出度。

在图 7.6 里顶点的信息只有其标记，可以根据实际需要增加更多数据域，存储所需的任何信息。 G_7 是一个带权图，图 7.6 只描述了它的连接关系，如果要表示边的权值，就需要在每个链表结点里增加相应的域。

图 7.7 给出了用邻接表表示无向图 G_8 的情况。这里的表示方法与有向图相同，只是同一条边应该出现在两个邻接点的边表里，在整个图表示中，所有链接表的结点数之和等于图

中边数的两倍。同样，在这个示意图里也没有表现边的权值。

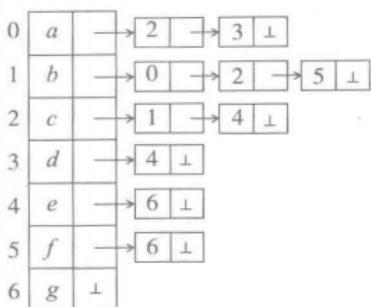


图 7.6 有向图 G_7 的邻接表表示

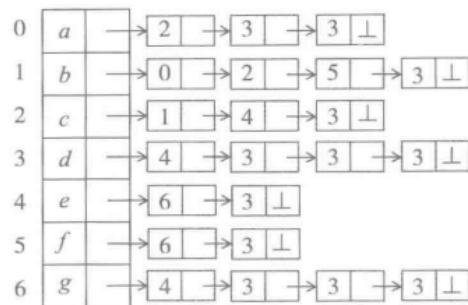


图 7.7 无向图 G_8 的邻接表表示

7.2 图结构的 Python 实现

本节探讨在 Python 语言里实现图结构的技术。

在 Python 里实现图数据结构，可以有许多方法，下面是几个例子：

- 用以 `list` 为元素的 `list` (两层的表) 或者 `tuple` 的 `tuple` (两层的 `tuple` 结构) 作为邻接矩阵的直接实现来表示图结构。如前面所言，这种表示方法结构简单，使用也很方便，容易判断顶点的邻接关系。但存储代价较大，不适合很大的图。
- 定义一种字典 (`dict` 对象)，以顶点的下标序对 (i, j) 作为关键码，实现从顶点对到邻接关系的映射 (例如，值为 1 表示有边，`None` 表示无边)。这种实现的检索效率很高 (平均为 $O(1)$ 时间)，采用适当的技术，可以做到图的空间开销与图中边数成正比[⊖]，适合表示稀疏矩阵 (邻接矩阵经常是这样)。但 Python 的字典本身比较复杂，而且基于顺序存储技术实现，是否适合大型图需要试验。
- 用 Python 内置的 `bytearray` 字节向量类型或者标准库的 `array` 类型。`bytearray` 是内置类型，与 `str` 类似，但为可变类型。`bytearray` 对象的元素是二进制字节，可用于表示边的存在与否，存储效率较高。`array` 是标准库里定义的数值汇集类型，其对象的元素可以是整数或浮点数等基本类型的值，可用于表示带权图。有关细节参见 Python 手册，这里不准备进一步讨论。如果读者计划用这些类型作为图的实现基础，下面讨论中的许多想法也可以参考。
- 自定义类型实现邻接表表示，其中具体数据的组织可以考虑上述各种技术。

下面将讨论两种自定义的实现方法——定义两个表示图的类。

在下面讨论中，假定有一个全局变量 `inf` 表示无穷大。在一些情况下，当带权图中 v 到 v' 无边，需要以 `inf` 的值作为边的权值。`inf` 的值应大于程序里可以计算出的任何值。在 Python 里可以定义这种值，只需要写：

```
inf = float("inf") # inf 的值大于任何 float 类型的值
```

⊖ 关键是不在字典里实际表示无边。如果对所有无边的 (i, j) 实际设置 `None` 值，空间开销至少是 $O(n^2)$ 。利用字典的 `get` 方法可以避免这种开销。设 `g` 是表示图的字典。如果用 `g[i, j]` 的形式访问字典项，关键码无定义时将会报 `KeyError` 异常。但用 `g.get(i, j)` 的形式访问，关键码无定义时返回 `None`。利用后一情况，就可以只在 `g` 中设置有边的情况。这种技术也可以用于表示带权图，为此只需利用 `get` 的第二个参数 (默认值为 `None`) 提供无边情况的默认值。参看 Python 手册。

7.2.1 邻接矩阵实现

首先考虑基于邻接矩阵技术定义一个实现图的类，其中矩阵元素可以是 1 或者权值，表示有边，或者用一个特殊值表示“无关联”。

为满足不同的应用需要，这里为用户提供一种选择，在构造图对象时可以通过参数为无关联的情况提供一个特殊值。构造函数的参数 unconn 服务于这个目的，其默认值为 0。

图构造函数的主要参数是 mat，表示初始的邻接矩阵。这里要求它是一个二维的表参数，提供图的基本构架，主要是确定图的顶点个数。构造函数将基于给定的矩阵参数建立一个图，做出参数矩阵的一个拷贝。在构造这个拷贝之前，函数先确认参数合法，检查给定矩阵是否为方阵。二维表的拷贝通过一个生成式完成，其中的切片操作 mat[i][:] 构造一行的拷贝。下面是有关定义，包括几个基本操作：

```
class Graph:
    """基本图类，采用邻接矩阵表示"""

    def __init__(self, mat, unconn=0):
        vnum = len(mat)
        for x in mat:
            if len(x) != vnum:      # 检查是否为方阵
                raise ValueError("Argument for 'Graph' must be square")
        self._mat = [mat[i][:] for i in range(vnum)] # 做拷贝
        self._unconn = unconn
        self._vnum = vnum

    def vertex_num(self):
        return self._vnum

    def _invalid(self, v):
        return 0 > v or v >= self._vnum

    def add_vertex(self):
        raise GraphError(
            "Adj-Matrix does not support 'add_vertex'.")

    def add_edge(self, vi, vj, val=1):
        if self._invalid(vi) or self._invalid(vj):
            raise GraphError(str(vi) + ' or ' + str(vj) +
                             " is not a valid vertex.")
        self._mat[vi][vj] = val

    def get_edge(self, vi, vj):
        if self._invalid(vi) or self._invalid(vj):
            raise GraphError(str(vi) + ' or ' + str(vj) +
                             " is not a valid vertex.")
        return self._mat[vi][vj]
```

几个牵涉顶点的操作都首先检查顶点下标的合法性。

这个简单的图类并未计划支持增加顶点，因此把 add_vertex 操作定义为直接引发异常。给邻接矩阵表示增加一个顶点，不仅需要给矩阵增加一行，还要为每行增加一个元素，比较麻烦。当然，如果真需要也可以做到，留给读者实现。

许多图算法需要逐个处理一个顶点的各条出边。下面定义一个方法返回相应出边的表，它调用一个内部的静态函数完成工作。采用下面实现方式，每次对具体顶点的调用将构造一个新表，如果用完就丢掉，下次还需要重新做。为避免这种代价，可以设法记录已经构造的表，有关设计和实现请读者考虑。

```
def out_edges(self, vi):
```

```

if self._invalid(vi):
    raise GraphError(str(vi) + " is not a valid vertex.")
return self._out_edges(self._mat[vi], self._unconn)

@staticmethod
def _out_edges(row, unconn):
    edges = []
    for i in range(len(row)):
        if row[i] != unconn:
            edges.append((i, row[i]))
    return edges

```

这里用一个静态方法构造出一个结点表，顶点 v 的出边用 (v', w) 的形式表示，其中 v' 是该边的终点， w 是边的信息（对于带权图，就是边的权）。如果 g 是一个图对象， v 是其合法顶点下标，用下面形式的循环可以方便地处理 v 的各条出边：

```

for vv, w in g.out_edges(v):
    ... vv ... v ...

```

循环中， vv 和 w 将逐个取得 v 的各条出边的终点和权值。

最后为 `Graph` 类定义采用特殊名 `__str__` 的方法函数，为这个类的对象提供一种转换为字符串的形式（可以用于输出）。Python 规定，对一种类型的对象作用内置函数 `str` 时，去调用该类的 `__str__` 方法。下面是方法定义：

```

def __str__(self):
    return "[\n" + ",\n".join(map(str, self._mat)) + "\n]" +
        "\nUnconnected: " + str(self._unconn)

```

函数生成的字符串包括两个部分，分为几行：首先是表示图结构的两层表，生成为嵌套表的形式，每个元素表生成一行；随后是“Unconnected: nnn”，说明无关联情况的表示方式。请注意，在构造表的基础部分时使用了 `str` 的 `join` 方法，分隔符是换行和逗号。这种技术特别适合构造由一种分隔符连接的多段字符串。

上面定义的图类已经可以满足本章讨论的需要，给这个图类扩充其他操作的工作留作课后练习，前面的图抽象数据类型和下面讨论可供参考。

7.2.2 压缩的邻接矩阵（邻接表）实现

邻接矩阵的缺点是空间占用与顶点数的平方成正比，可能带来很大的浪费。以前面提到的中国铁路路线图为例，采用上一小节的类，99.9% 以上的元素是 `inf`（或者 0）。另外，邻接矩阵不容易增加顶点，不太适合以逐步扩充的方式构造图对象。

现在考虑一种“压缩的”表示形式，也就是邻接表技术，把这个类命名为 `GraphAL`（表示邻接表图类）。在 `GraphAL` 类的对象里，每个顶点 v 的所有邻接边用一个（不一定等长的）`list` 对象表示（也可以用链接表，请读者考虑），元素形式为 (v', w) ，其中 v' 是边的终点， w 是边的信息（权）。一个 `GraphAL` 对象的主要部分就是这种 `list` 的 `list`，每个元素对应一个顶点。请注意这个设计与前面 `Graph` 类的相似性。

首先可以看到，采用这种设计很容易给已有的图添加顶点。为此只需在外层表中增加一个表示新顶点的项，对应的边表设为空表，通过随后的操作加入新边。也就是说，这种表示能更好地支持以逐步扩充的方式构造大型图对象。

`GraphAL` 类的定义继承 `Graph` 类，提供同样接口，内部使用同一套数据域。由于内部表示采用完全不同的形式，类中的主要方法都需要重新定义，少数方法可以继承。实际上，完全

可以不采用继承 Graph 的方式定义这个类。但那样做需要拷贝几个重用的方法，采用继承还是有益的。新类定义的开始部分如下：

```
class GraphAL(Graph):
    def __init__(self, mat=[], unconn=0):
        vnum = len(mat)
        for x in mat:
            if len(x) != vnum:      #检查是否为方阵
                raise ValueError("Argument for 'GraphAL' .")
        self._mat = [Graph._out_edges(mat[i], unconn)
                    for i in range(vnum)]
        self._vnum = vnum
        self._unconn = unconn
```

这里为初始化方法的 mat 参数提供空表作为默认值，以支持从空图出发逐步构造所需图对象^Θ。这也是本类的一种典型使用方式，实际中也有需要。最后调用 Graph 类的内部函数 _out_edges 构造出所需的边表。

下面是 GraphAL 类里其他方法的定义：

```
def add_vertex(self):           #增加新顶点时安排一个新编号
    self._mat.append([])
    self._vnum += 1
    return self._vnum - 1

def add_edge(self, vi, vj, val=1):
    if self._vnum == 0:
        raise GraphError("Cannot add edge to empty graph.")
    if self._invalid(vi) or self._invalid(vj):
        raise GraphError(str(vi) + ' or ' + str(vj) +
                          " is not valid vertex.")
    row = self._mat[vi]
    i = 0
    while i < len(row):
        if row[i][0] == vj:      #修改mat[vi][vj]的值
            self._mat[vi][i] = (vj, val)
            return
        if row[i][0] > vj:       #原来没有到vj的边，退出循环后加入边
            break
        i += 1
    self._mat[vi].insert(i, (vj, val))

def get_edge(self, vi, vj):
    if self._invalid(vi) or self._invalid(vj):
        raise GraphError(str(vi) + ' or ' + str(vj) +
                          " is not a valid vertex.")
    for i, val in self._mat[vi]:
        if i == vj:
            return val
    return self._unconn

def out_edges(self, vi):
    if self._invalid(vi):
        raise GraphError(str(vi) +
                          " is not a valid vertex.")
    return self._mat[vi]
```

^Θ 这里又出现了用可变对象作为参数默认值的情况，要保证总是建立新拷贝。

加入新顶点的方法很简单，加入新边的方法最复杂。采用这一复杂设计的原因是看到了一种现象：从邻接矩阵构造出的边表中隐藏着一种顺序，即在边表里的出边按各顶点在邻接矩阵里的下标顺序排列。加入新边的操作应该保持这种顺序，保证这种图对象的操作有确定的顺序。如果简单地把新边加在边表最后，图操作的顺序就没有保证了^Θ。显然，采用这种设计，也使插入操作的效率等比于边表的长度。但考虑到图的构造（和扩充）只做一次，在这里多用一点时间，通常不会成为问题。

边表的设计本身也是可以考虑的问题。采用顺序表（如这里用 `list`）和顺序检索方式的插入/访问，如果顶点出度很小，操作效率不是问题。如果顶点的出度有可能很大，也可以考虑采用二分检索技术，或者为每个顶点关联一个表示边表的字典（Python 的 `dict`），记录从邻接顶点到边值的关联，以支持快速插入和访问。

7.2.3 小结

上面定义了两个表示图的类，其内部实现不同但接口相同。这样，从这两个类的对象出发，能调用的方法完全一样，支持同样的使用方式。

但也应该看到，由于两个类采用的数据表示不同，各种操作的实现方法不同，操作的时间和空间效率也不同。例如，从一个顶点出发访问其所有出边，对于 `Graph` 对象，操作效率正比于图中结点个数，对 `GraphAL` 对象则正比于顶点的出边数。另外，要确定两个顶点之间是否有边，对于 `Graph` 对象是常量时间操作，而对 `GraphAL` 对象就需要扫描一个顶点的出边表。下面会看到这些对算法效率的影响。

由于上面两个类的对象支持同一套方法，如果基于这套方法定义函数，实现重要的图算法，有关定义将能适用于这两种不同的图实现。

最后，上面两个类主要表示了边的信息，其中的顶点只是一个编号。如果在实际图中的顶点还有更多的关联信息，例如名字和其他相关数据，就需要扩充上面的类。例如，可以考虑在图对象里增加一个顶点表或顶点字典。

7.3 基本图算法

很多实际问题可以抽象为图和图上的计算问题，例如：

- 互联网和移动电话网的路由（几乎每个人每天都在用）。
- 集成电路（和印制电路板）的设计和布线。
- 运输和物流中的各种规划安排问题。
- 工程项目的计划安排。
- 许多社会问题计算，如金融监管（例如关联交易检查）。

一旦从应用中抽象出一个图，应用问题可能就变成图上的一个算法问题。

本章后面部分将讨论一些图上的问题和求解算法，这些问题都有很清晰的应用背景，算法里也蕴涵着巧妙的想法和理论根据，其正确性需要（也可以）严格证明。图算法的复杂度非常重要，因为需要用它们处理的问题规模可能很大，低效的算法根本无法处理有一定规模的问题实例。此外，由于图结构的复杂情况，这些算法中常常需要用到前面讨论的一些数据结构，特别是栈、队列和优先队列。

^Θ 是否在操作中维持某种顺序是一项设计选择，涉及操作的效率和算法意义的确定性。

7.3.1 图的遍历

前面已经讨论过遍历问题。对于图的遍历，就是按某种方式系统地访问图中每个顶点而且仅访问一次的过程，也称为图的周游。实际上，前面讨论过的状态空间中的状态和相邻关系就形成了一个图，图的遍历对应于一次穷尽的状态空间搜索。

在图的实现中，顶点可能具有可利用的结构。例如，如果采用邻接矩阵或者邻接表表示图，就可以通过下标遍历所有顶点，操作的实现非常方便。但是，有些情况要求基于图中的顶点邻接关系进行遍历。例如，要找出从一个顶点可达的所有顶点，或者找到满足特定条件的可达顶点，就必须基于图的结构进行遍历了。

一般的图未必是连通图。基于图结构的遍历只能从一个（或几个）顶点出发，根据图的结构只能访问顶点所在的连通分量（对于有向图，是该顶点的可达子图）里的全部顶点。由于被处理的图可能不连通，要完成整个图的遍历，在完成了一个可达部分的遍历后，还需要考虑对图中尚未遍历的其他部分的处理。

与一般状态空间搜索的方式对应，完成图的遍历，其基本方法同样是深度优先遍历和宽度优先遍历两种。但要注意，图与树不同，图中到达同一个顶点的路径可能不止一条，还可能存在回路。因此，在遍历中必须避免多次处理图中同一个部分的潜在问题。第 5 章研究迷宫搜索时曾经讨论过这个问题，下面也采用标记顶点的技术。

下面讨论深度和宽度优先遍历的性质及其程序实现。

深度优先遍历和宽度优先遍历

采用深度优先遍历方式处理一个图，也就是按照深度优先搜索（Depth-First Search）的方式实施整个遍历过程。假定从指定顶点 v 出发，深度优先遍历的做法是：

- 首先访问顶点 v ，并将其标记为已访问。
- 检查 v 的邻接顶点，从中选一个尚未访问的顶点，从它出发继续进行深度优先搜索（这是递归）。不存在这种邻接顶点时回溯（邻接顶点可能排了一种顺序）。
- 反复上述操作直到从 v 出发可达的所有顶点都已访问（递归）。
- 如果图中还存在未访问的顶点，则选出一个未访问顶点，由它出发重复前述过程，直到图中所有顶点都已访问为止。

通过深度优先遍历顺序得到的顶点序列称为该图的深度优先搜索序列（Depth-First Search 序列），简称为 DFS 序列。显然，对图中任一顶点的邻接点采用不同的访问顺序，就可能得到不同的 DFS 序列（因此，DFS 序列不唯一），如果规定了图中各顶点的邻接点顺序，也就确定了 DFS 序列。

图的宽度优先遍历通过宽度优先搜索（Breadth-First Search）的方式实施遍历。假设从指定顶点 v_i 出发，宽度优先遍历的过程如下：

- 先访问顶点 v_i 并将其标记为已访问。
- 依次访问 v_i 的所有相邻顶点 $v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_{m-1}}$ （可能规定某种顺序），再依次访问与 $v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_{m-1}}$ 邻接的所有尚未访问过的顶点……如此进行下去直到所有可达顶点都已访问。
- 如果图中还存在未访问顶点，则选择一个未访问顶点，由它出发按同样方式基于宽度优先搜索工作，直到所有顶点都已访问为止。

通过宽度优先遍历得到的顶点序列，称为这个图中顶点的宽度优先搜索序列（Breadth-First Search 序列），或者称为 BFS 序列。与 DFS 的情况类似，如果规定了图中各顶点的邻接点顺序，BFS 序列就确定了。

【例 7.9】 看两个简单例子，首先考虑图 7.5 中 G_7 的 DFS 序列。假设从顶点 a 出发开始遍历，一个顶点的不同邻接顶点按照 a, b, c, \dots 的顺序处理。得到的序列是：

a, c, b, f, g, e, d

考虑图 7.5 中 G_8 的 BFS 序列。采用上面同样假定，得到的序列是：

a, b, c, d, e, g, f

深度优先遍历的非递归算法

现在考虑图的遍历算法。显然，这种算法可以采用递归或者非递归的技术实现。下面只考虑非递归的深度优先遍历算法，其基本过程的框架与前面迷宫求解类似，同样需要用一个栈作为辅助数据结构。其他遍历算法留作练习。

假设图对象是前面定义的 Graph 或 GraphAL 类的实例，现在要求遍历从给定顶点 v_0 出发可达的顶点集。下面考虑基于 Graph 或 GraphAL 对象支持的基本操作描述算法，因此对这两个图类的对象都适用。

前面提到的一个问题必须考虑，就是防止多次遍历同一顶点。在下面的算法里，采用了一个内部的表对象记录访问历史，对应每个顶点有一个表元素。当一个顶点被访问时，将该顶点下标对应的表元素设置为 1。初始时这个表的所有元素取 0 值。下面是遍历函数的定义，它返回得到的 DFS 序列：

```
def DFS_graph(graph, v0):
    vnum = graph.vertex_num()
    visited = [0]*vnum
    visited[v0] = 1
    DFS_seq = [v0]
    st = SStack()
    st.push((0, graph.out_edges(v0)))
    while not st.is_empty():
        i, edges = st.pop()
        if i < len(edges):
            v, e = edges[i]
            st.push((i+1, edges))
            if not visited[v]:
                DFS_seq.append(v)
                visited[v] = 1
                st.push((0, graph.out_edges(v)))
                # 下面访问的边组
    return DFS_seq
```

算法中入栈的元素形式为 (i, edges) ，其中 edges 是某个顶点的边表， i 是边表的下标，表示当这个序对弹出时应该考虑的下一条边的下标。

现在考虑上面算法的复杂度。在本章的讨论中，算法复杂度（时间或空间）一般都基于图中顶点数 $|V|$ 和边数 $|E|$ 度量，这样做显然是合理的。

时间复杂度：函数开始时构造 visited 表和 DFS_seq 表，时间复杂度是 $O(|V|)$ 。算法进行中入栈和出栈操作的次数对应于图中边数，总开销是 $O(|E|)$ 时间。遇到一个未访问顶点时需要将其出边表入栈。对于 Graph 对象，构造所有出边表的总耗时为 $O(|V|^2)$ ，而对于 GraphAL 图实现，取出边总耗时是 $O(|E|)$ 。

由于显然有 $|E| \leq |V|^2$ ，综合起来，对于 Graph 对象（图的邻接矩阵实现），整个遍历的时间复杂度是 $O(|V|^2)$ ；而对于 GraphAL 对象（图的邻接表实现），操作的时间复杂度是 $O(\max(|V|, |E|))$ 。对于比较稀疏的图，后一情况的代价较低。

空间复杂度：`visited` 和 `DFS_seq` 都需要 $O(|V|)$ 空间，栈的深度也不会超过顶点个数，所以算法的空间复杂度是 $O(|V|)$ 。

从上面讨论可以看到数据结构的不同实现方法对算法复杂度的影响。

这里只给出了一个遍历算法。采用类似的基本设计，也可以给出图的宽度优先遍历，或者基于递归描述的深度优先遍历。另外，与前面章节里讨论的线性表或树的情况类似，对于图，也可以不生成遍历序列，而是给遍历函数引进一个操作函数参数，或者定义一个遍历图结构的生成器函数。有关工作都留作练习。

7.3.2 生成树

现在考虑图上的生成树概念及其相关算法。

如果图 G 是连通无向图或者强连通有向图（实际上只要求是有根有向图），从无向图 G 中任一顶点 v_0 出发，或者从有根有向图的根 v_0 出发，到图中其他各个顶点都存在路径。本小节（如果不专门提出）讨论中考虑的都是这两种图。

性质 7.4（图中的路径与路径上的边数） 如果图 G 有 n 个顶点，必然可找到 G 中的一个包含 $n-1$ 条边的边集合，这个集合里包含了从 v_0 到其他所有顶点的路径（很容易通过数学归纳法证明）。

图 G 中满足上述性质的 $n-1$ 条边（加上 G 所有顶点）形成了图 G 的一个子图 T 。由于 T 包含 n 个顶点且只有 $n-1$ 条边，因此它不可能包含任何回路，形成了一棵树（有向或无向的树，以 v_0 为根结点）。

对无向图 G ，满足上述性质的子图 T 是 G 的一个最小连通子图（去掉其中任意一条边后将不再连通）。由于图 T 是树形结构，因此称 T 为 G 的一棵生成树。

对于强连通有向图（或有根有向图） G ，满足上述性质的子图 T 是 G 的一个最小的有根子图（以 v_0 为根）。 T 也称为 G 的一棵生成树。

首先请注意，如果一个图有生成树，其生成树也可能不唯一。

性质 7.5（生成树的边数） n 个顶点的连通图 G 的生成树包含恰好 $n-1$ 条边。无向图 G 的生成树就是 G 的一个最小连通子图，是一个无环图。有向图的生成树中所有的边都位于从根到其他顶点的（有方向的）路径上。

一般而言，一个无向图 G 可能非连通。但由于任何无向图都可以划分为一组连通分量，因此每个无向图都存在生成树林。

性质 7.6（图的生成树林的边数） 包含 n 个顶点、 m 个连通分量的无向图 G 的生成树林恰好包含 $n-m$ 条边。

通过对 G 的连通分量数做归纳，不难证明这个性质。

遍历和生成树

从连通无向图或强连通有向图中任一顶点出发遍历，或从有根有向图的根顶点出发遍历，都可以访问到所有顶点。在遍历中经过的边加上原图的所有顶点，就构成该图的一棵生成树。通过遍历构造生成树的过程可以按深度优先方式或宽度优先方式进行，在遍历中记录访问的所有顶点和经过的边，就得到原图的深度优先生生成树（简称 DFS 生成树），或者宽度优先生生成树（BFS 生成树）。

【例 7.10】 图 7.8 展示的是图 7.5 中两个图的生成树。左边是对图 G_7 从顶点 a 出发遍历得到的 DFS 生成树，右边是对 G_8 从顶点 b 出发遍历得到的 BFS 生成树。

构造 DFS 生成树

现在考虑构造 DFS 生成树的算法。构造 BFS 生成树的情况与此类似，留作练习。

前面讨论过如何实现 DFS 遍历，剩下的问题是如何构造并给出所需的 DFS 生成树。生成树的顶点就是原图的顶点，现在的关键是设法表示生成树上的边。

注意：生成树上的边形成了从初始顶点到其他顶点的一簇路径。在这簇路径里，一个顶点可能有多个“下一顶点”，但每个顶点至多有一个“前一顶点”。记录路径的一种方式是记录所有的“前一顶点”关系。有了这些信息，遍历完所有顶点之后，根据前一顶点关系就能追溯出所有的路径了。

在考虑路径上的前一顶点关系时，对应每个顶点只需要记录一项信息。设图中有 vnum 个顶点，可以用一个包含 vnum 个元素的表 span_forest 记录得到的路径信息，令表项 span_forest[vj] 的形式是序对 (vj, e)，其中 vj 是从 v0 到 vi 的路径上 vi 的前一顶点，e 是从 vj 到 vi 的邻接边的信息。如果只考虑简单的图，有关边存在的“1”实际上可以不计。但这里希望统一处理各种有意义的图，包括不丢失带权图中边上的有效信息，因此采用如上所述的统一设计。

有了这些考虑，可以实现递归或非递归的 DFS 生成树构造算法，或者 BFS 生成树构造算法。下面只考虑一个 DFS 递归算法，其余算法留作练习。

构造 DFS 生成树：递归算法

下面定义的函数 DFS_span_forest 将生成参数 graph 的 DFS 生成树林，其中主要工作由内部的递归定义函数 dfs 完成。

主函数里的局部变量 span_forest 是一个包含 vnum 个元素的表，初始时元素值都是 None，表示到顶点的路径尚未找到。实际上，这个 None 也表示该顶点尚未访问，所以就不再需要前面遍历算法里使用的 visited 表了。

主函数里的循环是考虑到图 graph 可能不连通。循环中找到下一个未遍历顶点，从它出发做出一棵生成树。if 条件成立就是找到了尚未访问的顶点，给它的标记是到自己的边长为 0（这个特殊标记也表示该顶点是树林中一棵生成树的根），然后调用函数 dfs 做出以这个顶点为根的生成树。

函数 dfs 的定义非常简单。这里只有一点值得提出：函数执行中需要修改非局部的变量 span_forest，因此需要将其声明为 nonlocal。

```
def DFS_span_forest(graph):
    vnum = graph.vertex_num()
    span_forest = [None]*vnum

    def dfs(graph, v):          # 递归遍历函数，在递归中记录经由边
        nonlocal span_forest # span_forest是nonlocal变量
        for u, w in graph.out_edges(v):
            if span_forest[u] is None:
                span_forest[u] = (v, w)
                dfs(graph, u)
    for v in range(vnum):
```

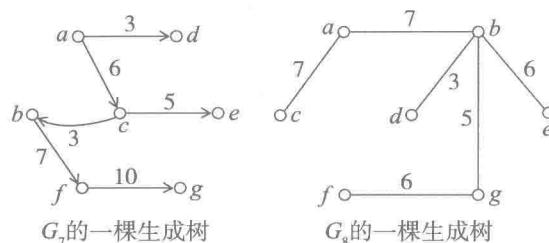


图 7.8 带权图实例

```

if span_forest[v] is None:
    span_forest[v] = (v, 0)
    dfs(graph, v)
return span_forest

```

由于 `span_forest` 的值是一个表对象，通过参数传递也能正常工作。

现在考虑算法的时间复杂度，情况与遍历算法类似：

- 主函数的循环对每个顶点检查一次，其自身的执行时间为 $O(|V|)$ 。
- 在子函数 `dfs` 的各次调用的所有递归访问中，表 `span_forest` 的每个顶点将设置一次，不会重复设置。而这种设置的次数与 `dfs` 递归调用的次数一样（因为这两个语句顺序执行）。但另一方面，`dfs` 里的循环可能访问到图中每条边，但至多一次。
- 对于 `Graph` 类型的对象，取所有出边需要 $O(|V|^2)$ 时间。
- 综合起来可知，对于图的邻接矩阵实现，本算法的时间复杂度是 $O(|V|^2)$ ；对于图的邻接表实现，算法的时间复杂度是 $O(\max(|V|, |E|))$ 。

空间复杂度：`span_forest` 需要 $O(|V|)$ 空间，`dfs` 函数递归的深度也不会超过 $O(|V|)$ 。所以这个算法的空间复杂度是 $O(|V|)$ 。

7.4 最小生成树

下面几节主要讨论带权图上的计算问题和相关算法。本节讨论带权连通无向图（网络）上的最小生成树问题以及求最小生成树的两个算法。

7.4.1 最小生成树问题

假定 G 是一个网络，其中的边带有给定的权值，自然也可以做出它的生成树。现将 G 的一棵生成树中各条边的权值之和称为该生成树的权。

网络 G 可能存在多棵不同的生成树，不同生成树的权值也可能不同，其中权值最小的生成树称为 G 的最小生成树（Minimum Spanning Tree, MST）。显然，任何一个网络都必然有最小生成树，但其最小生成树也可能不唯一。

最小生成树有许多实际应用。例如，将网络顶点看作城市，边看作连接城市的通信网，边的权看作连接城市的通信线路的成本，根据最小生成树建立的通信网就是这些城市之间成本最低的通信网。

可类似地考虑各种网络问题，如成本最低的城市间（或者村庄间）的公路网、输电网或者有线电视网；城市的输水管网、暖气管线、配送中心与线路网络规划；集成电路或印制电路板的地线、供电线路；等等。

在表示这类应用的带权图中，各结点到其自身的权值应取为 0，到其他结点有边时有具体权值，无边时权值取无穷大 `inf`。下面算法中均采用这一假定。

7.4.2 Kruskal 算法

Kruskal 算法是一种构造最小生成树的简单算法，其中的思想比较简单。

基本方法

设 $G=(V, E)$ 是一个网络，其中 $|V|=n$ 。Kruskal 算法构造最小生成树的过程是：

- 初始时取包含 G 中所有 n 个顶点但没有任何边的孤立点子图 $T=(V, \{\})$ ， T 里的每个顶点自成一个连通分量。下面将通过不断扩充 T 的方式构造 G 的最小生成树。

- 将边集 E 中的边按权值递增的顺序排序，在构造中的每一步顺序地检查这个边序列，找到下一条（最短的）两端点位于 T 的两个不同连通分量的边 e ，把 e 加入 T 。这导致两个连通分量由于边 e 的连接而变成了一个连通分量。
- 每次操作使 T 减少一个连通分量。不断重复这个动作加入新边，直到 T 中所有顶点都包含在一个连通分量里为止，这个连通分量就是 G 的一棵最小生成树。

如果这样做不能得到一个包含 G 的所有顶点的连通分量，则原图不连通，没有最小生成树。算法做出的是 G 的最小连通树林。这一算法的正确性不难用归纳法证明。算法得到的显然是原图的生成树，关键是证明这棵生成树最小。

为了证明做出的生成树最小，只需归纳地证明加入第 i 条边后，得到的包含 $n-i$ 棵树的树林权值最小。初始没有边，自然成立。加入了 $n-1$ 条边后得到的生成树也最小。归纳步骤的严格论述请读者自己考虑。

【例 7.11】 考虑用 Kruskal 算法构造图 7.9 中给出的 G_9 的最小生成树。构造过程见图 7.9 所示。构造过程的初始状态中 T 见图 (1)。第一步首先选择图中的最短边 (b, d) ，将其加入 T 得到 (2) 的状态。这时再找最短边，有两条长为 5 的边都可以减少一个连通分量。任选其中的 (a, d) 加入 T ，就得到了状态 (3)。这时尚未用到的另一最短边不能减少连通分量数，将其抛弃（这些情况也说明了最小生成树不唯一）。选择下一条最短边 (c, f) 加入 T 得到状态 (4)。随后再顺序选取两条长为 7 的边和一条长为 8 的边，最后得到状态 (5)。由于这时只剩下一个连通分量，最小生成树的构造工作完成。

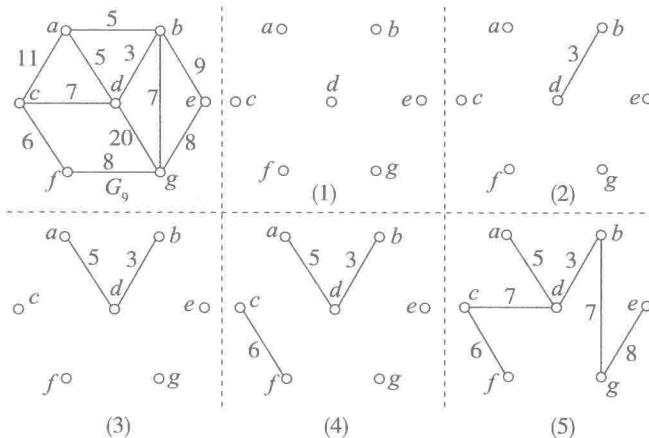


图 7.9 构造最小生成树的 Kruskal 算法的操作过程示例

算法实现中的问题

考虑下面 Kruskal 算法的抽象描述：

```

T = (V, {})
while T 中所含边数小于 n-1:
    从 E 中选取当前最小边 (u, v), 将它从 E 中删除
    if (u, v) 两端点属于 T 的不同连通分量:
        将边 (u, v) 加入 T
    
```

算法中有两个问题需要考虑。其一是当前最短边的选取，有多种可能做法。例如每次扫描剩下的边，选出最短边；或者先将所有的边排序后顺序选取；还可以用一个优先队列。几种做法的性质不同，请读者分析。另一个问题更重要，那就是如何判断两个顶点在当时的 T 里属于

不同连通分量。一种显然的方法是在 T 中检查这条边的两个端点，看它们之间是否已有路径。但是这种做法太麻烦也太费时间。

人们提出的一种有效方法是为每个连通分量确定一个代表元，如果两个顶点的代表元相同，它们就相互连通，属于同一连通分量。这样，如果确定顶点的代表元是常量时间操作，判断两顶点是否连通也就只需要常量时间了。

在算法执行过程中，需要记录和维护（必要时更新）每个顶点的代表元。最麻烦的问题是合并连通分量时的代表元维护。加入一条边减少了连通分量，这时需要选一个顶点，让被合并的两个连通分量里的顶点都以它为代表元。完成这件事的简单方法是从原来的两个代表元中任选一个，而后更新另一连通分量中顶点的代表元。

在下面实现这个算法时，将用一个下标为顶点编号的表 reps 记录各顶点的代表元关系，这样，从顶点出发找到代表元只需要 $O(1)$ 时间。

在算法初始时，每个顶点都应该以其自身作为代表元 ($\text{reps}[i] = i$)，因为它所在的连通分量里只有自己。在算法过程中需要不断更新 reps ，保证 $\text{reps}[v]$ 总是顶点 v 的代表元的下标。为了记录逐步构造出的最小生成树，这里另用了一个表 mst （最小生成树的首字母缩写）累积最小生成树的边。

算法的 Python 实现

下面给出 Kruskal 算法的一个实现。除表 reps 和 mst 外，这个算法还使用了另一个表 edges ，在其中存储图 graph 所有的边，并调用 Python 表的 sort 操作将这些边按权值从小到大排好序。随后的操作就是逐个选择最短的有效边。这是前面提出的第二种做法。采用其他做法的实现请读者自己完成。

边表的形式是 (w, v_i, v_j) ，其中 w 是这条边的权值， v_i 和 v_j 是其两个端点。在主循环里顺序检查 edges 里的边（按权值从小到大），如果一条边的两端点代表元不同，就将其加入 mst ，并更新一个连通分量的代表元。这样反复做到 mst 里积累了 $n-1$ 条边（成功得到最小生成树），或者所有边都已检查完毕（没有最小生成树），循环结束。这时在 mst 里保存的是 graph 的最小生成树，或最小生成树林。

```
def Kruskal(graph):
    vnum = graph.vertex_num()
    reps = [i for i in range(vnum)]
    mst, edges = [], []
    for vi in range(vnum):           # 所有边加入表edges
        for v, w in graph.out_edges(vi):
            edges.append((w, vi, v))
    edges.sort()                     # 边按权值排序, O(n log n)时间
    for w, vi, vj in edges:
        if reps[vi] != reps[vj]:      # 两端点属于不同连通分量
            mst.append((vi, vj), w)   # 记录这条边
            if len(mst) == vnum-1:     # |V|-1条边, 构造完成
                break
            rep, orep = reps[vi], reps[vj]
            for i in range(vnum):       # 合并连通分量, 统一代表元
                if reps[i] == orep:
                    reps[i] = rep
    return mst
```

Kruskal 算法的复杂度

现在考虑上面 Kruskal 算法实现的复杂度。假设被处理的图是 $G=(V, E)$ ，复杂度基于其顶

点集合和边集合的大小 $|V|$ 和 $|E|$ 描述。

时间复杂度分析：

- 建立边表 `edges` 用 $O(|E|)$ 时间，排序时间为 $O(|E| \log |E|)$ （这一复杂度由 Python 中排序函数的时间性质确定）。
- 主循环里的操作分为两个分支，一个进入条件体，另一个不进入：
 - 整个循环体执行不超过 $O(|E|)$ 次，时间是 $O(|E|)$ 。
 - 进入条件体记录边的次数最多 $|V|-1$ 次，每记录一条边后修改代表元的小循环需要 $O(|V|)$ 时间，这部分的复杂度是 $O(|V|^2)$ 。
 - 主循环的时间复杂度是 $O(\max(|E|, |V|^2)) = O(|V|^2)$ 。
- 总的时间复杂度是 $O(\max(|E| \log |E|, |V|^2))$ 。

空间复杂度：算法中用了一个保存边的表 `edges`，另外两个表 `reps` 和 `mst`（大小均由图中顶点数确定）。因此算法的空间复杂度是 $O(\max(|E|, |V|))$ 。如果处理的是连通图，总有 $O(|E|) \geq O(|V|)$ ，所以空间复杂度是 $O(|E|)$ 。

注意：如果被操作的图采用邻接矩阵表示，建立边表需要 $O(|V|^2)$ 时间，主循环之前部分的复杂度就是 $O(\max(|E| \log |E|, |V|^2))$ ，但整个算法的复杂度不变。

一些数据结构教科书上不建立排序的边表，而是直接检查原图的拷贝，一次次找最小元。这样实现 Kruskal 算法，时间复杂度将达到 $O(|V|^3)$ ，空间复杂度是 $O(|V|^2)$ 。更好的实现方法采用其他技术和辅助结构，可以使算法时间复杂度降到 $O(|E| \log |V|)$ ，其中最重要的是采用一种巧妙的技术完成集合的合并（代表元维护）。

这些讨论也说明，从本质上说，Kruskal 算法是一种抽象想法，可称为一个抽象算法（或称为算法模式）。实现该算法时，可以采用不同的具体辅助数据结构和不同实现方法。不同实现可能具有不同的时间和空间复杂度。

7.4.3 Prim 算法

下面考虑解决同一问题的另一种算法，称为 Prim 算法，其设计出发点与 Kruskal 算法完全不同，基于最小生成树的一种重要性质。这个算法的基本做法是从一个顶点出发，逐步扩充包含该顶点的部分生成树 T 。开始时， T 只包含初始顶点而且没有边，算法最终做出了一棵最小生成树。

MST 性质及其证明

最小生成树有一个重要的 MST 性质，叙述如下：

设 $G=(V, E)$ 是一个网络， U 是 V 的任一真子集，设 $e=(u, v) \in E$ 且 $u \in U$, $v \in V-U$ （也就是说， e 的一个端点在 U 里，另一个不在），而且 e 在 G 中所有一个端点在 U 而另一端点在 $V-U$ 的边中权值最小，那么 G 必有一棵包括边 e 的最小生成树。

证明：取 G 的一棵最小生成树 T （根据定义，网络的最小生成树必定存在），如果 e 属于 T 则性质得证。否则将 e 加入 T 得到 G 的另一子图 T' 。由于 T 连通且是生成树，所以 T' 中必定存在环。由 T 中无环可知这个环一定包含边 e 。由于 e 的一端在 U 而另一端在 $V-U$ ，所以在环中至少还存在另一条边，其一端在 U 而另一端在 $V-U$ ，设这条边为 e' 。从 T' 中去掉 e' 得到 G 的另一个子图 T'' ，易见下面两个性质成立：① T'' 连通；② T'' 包含 $n-1$ 条边，因此 T'' 是 G 的另一棵生成树。根据 e 的定义， e 的权值不大于 e' ，因此新生成树 T'' 的权不大于 T ，即为所需。

Prim 算法的基本想法

Prim 算法是 MST 性质的直接应用，其基本思想是：从一个顶点出发，利用 MST 性质选择最短连接边，扩充已连接的顶点集并加入所选的边，直至结点集合里包含了图中的所有顶点；或最终确定这个图不是连通图。

算法细节：

- 从图 G 的顶点集 V 中任取一顶点（例如顶点 v_0 ）放入集合 U 中，这时 $U=\{v_0\}$ ，令边集合 $E_T=\{\}$ ，显然 $T=(U, E_T)$ 是一棵树（只包含一个顶点且没有边）。
- 检查所有一个端点在集合 U 里而另一个端点在集合 $V-U$ 的边，找出其中权最小的边 $e=(u, v)$ （假设 $u \in U, v \in V-U$ ），将顶点 v 加入顶点集合 U ，并将 e 加入边集合 E_T 。易见，扩充之后的 $T=(U, E_T)$ 仍是一棵树。
- 重复上面步骤直到 $U=V$ （所构造的树已经包含了所有顶点）。这时集合 E_T 里有 $n-1$ 条边，子图 $T=(U, E_T)$ 就是 G 的一棵最小生成树。

算法正确性：这一算法最后得到的必然是 G 的最小生成树。请读者自己考虑如何证明这个算法的正确性。

【例 7.12】 图 7.10 展示了 Prim 算法从顶点 a 出发构造 G_9 的最小生成树的过程：开始的状态如(1)所示，其中已经属于最小生成树 T 的顶点集合 U 只包含 a 。这里从 U 到不属于 U 的顶点的边有三条，其中两条最短边的长度都是 5。第一步选择边 (a, b) 并将顶点 b 加入 U ，达到状态(2)。现在从 U 到非 U 顶点的边有 4 条，从中选择最短的边 (b, d) 并将顶点 d 加入集合 U ，得到状态(3)。现在从 U 到非 U 顶点的边有 5 条，其中两条权为 7 的边最短。选择边 (c, d) 并将顶点 c 加入 U 得到状态(4)。继续下去在加入边 (c, f) 、 (b, g) 后发现新边 (e, g) 更短，将其加入就得到了如(5)所示的生成树，即为所求。

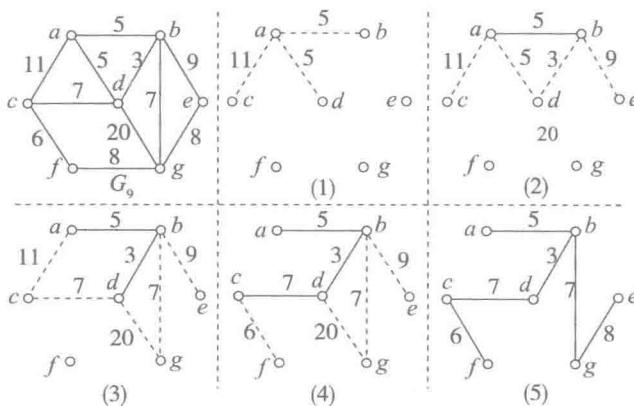


图 7.10 构造最小生成树的 Prim 算法的操作过程示例

Prim 算法的实现

最小生成树用类似 DFS 生成树构造算法的数据表示，用包含 $vnum$ 个元素（ $vnum$ 为图中顶点数）的表 mst 记录所构造的最小生成树的边，其元素的形式是 $((i, j), w)$ 。例如，如果元素 $mst[1]$ 的值是 $((1, 2), 10)$ ，表示顶点 1 已经属于 U ，且顶点 1 到 2 的边在 mst 中，其权值为 10。 $mst[i]$ 的值是 None 表示顶点 i 还不属于 U 。

工作中用一个优先队列 $cands$ 记录候选的最短边，其元素的形式为元组 (w, i, j) ，表示从顶点 i 到 j 的候选边的权值为 w 。优先队列以 w 值作为优先级，值较小的排在前面， w 值相同

者不需要控制前后，可任意选择。

算法过程：

- 初始把 $(0, 0, 0)$ 放入优先队列，表示从顶点 0 到其自身的长为 0 的边。
- 循环在第一次迭代中先把顶点 0 记入最小生成树顶点集 U ，方法是设置元素 $mst[0] = ((0, 0), 0)$ 。然后把顶点 0 到其余顶点的边按权值存入优先队列 $cands$ ，表示待考察的候选边集合。
- 随后的循环执行中反复选择 $cands$ 里记录的最短边 (u, v) 。如果确定了它是连接 U 中顶点与 $V - U$ 顶点的边（也就是说，发现 v 是属于 $V - U$ 的顶点），就把这条边及其权记入 $mst[v]$ ，并把 v 的出边存入 $cands$ ；否则就直接丢掉。

结束时 mst 中是最小生成树的 $vnum$ 条边（包括边 $(0, 0)$ 以方便实现）。

下面算法里用了一个优先队列 $cands$ 记录候选边， mst 的作用与前面算法相同。开始将边 $(0, 0, 0)$ 压入队列，表示从顶点 0 到自身的长度为 0 的边，第一个元素是权值。然后执行算法的主循环，直到 mst 记录了 n 个顶点（成功构造出最小生成树）或优先队列空（说明图 $graph$ 不连通，没有最小生成树）时结束。

```
def Prim(graph):
    vnum = graph.vertex_num()
    mst = [None]*vnum
    cands = PrioQueue([(0, 0, 0)])           # 记录候选边 (w, vi, vj)
    count = 0
    while count < vnum and not cands.is_empty():
        w, u, v = cands.dequeue()            # 取当时的最短边
        if mst[v]:
            continue                      # 邻接顶点v已在mst，继续
        mst[v] = ((u, v), w)                # 记录新的MST边和顶点
        count += 1
        for vi, w in graph.out_edges(v):   # 考虑v的邻接顶点vi
            if not mst[vi]:                # 如果vi不在mst则这条边是候选边
                cands.enqueue((w, v, vi))
    return mst
```

每次迭代从 $cands$ 里弹出一条边，优先队列保证这是当时队列里的最短边。变量 v 是边的另一端点，程序保证在压入这条边时 v 不在已知的部分生成树里，但到了出队列时， v 有可能已经被加入生成树。因此，取到 v 之后还需要检查 $mst[v]$ 是否为 $None$ 值，非 $None$ 说明 v 已经在集合 U 里，直接转到循环的下一次迭代；否则将其加入 mst ，计数，然后考虑从它出发有没有边可以到达不属于集合 U 的顶点。

Prim 算法的复杂度

现在考虑上面 Prim 实现的工作效率。

时间复杂度：初始化部分是 $O(|V|)$ ，算法时间主要用于选择最小生成树的边。主循环体的执行次数与考察和加入优先队列的元素个数有关，也与顶点的个数 $|V|$ 有关，不超过 $|V|$ 次。考虑到每条边至多进出队列一次，以及优先队列入队出队的 $O(\log n)$ 复杂度，这方面的时间开销是 $O(|E| \log |E|)$ ；另一方面，构造 $O(|V|)$ 条边的时间开销不低于 $O(|V|)$ 。一般而言前者大于后者（对于连通图， $O(|E|) \geq O(|V|)$ ），所以主循环的时间复杂度为 $O(|E| \log |E|)$ ，整个算法的时间复杂度也是 $O(|E| \log |E|)$ 。

空间复杂度：算法中用了一个表 mst 和一个优先队列 $cands$ ，大小分别是 $O(|V|)$ 和 $O(|E|)$ 。对连通图有 $|E| \geq |V|$ ，由此空间复杂度是 $O(|E|)$ 。

Prim 算法也是一种抽象算法，可以有许多不同的实现。不使用优先队列也可能做出时间复杂度为 $O(|V|^2)$ 的算法^Θ。请注意，由于 $|E| < |V|^2$ ，所以 $\log |E| < 2 \times \log |V|$ ，由此可得 $O(\log |E|) < O(2 \times \log |V|)$ ，所以 $O(\log |E|) = O(\log |V|)$ 。因此，也可以说上面给出的 Prim 算法的时间复杂度是 $O(|E| \log |V|)$ 。对于边比较稀疏的图，也就是说，如果 $|E| = O(|V|)$ ，这里的算法效率更高。

注意：如果图用邻接矩阵表示，提取一个顶点的邻接边就是 $O(n)$ 操作，整个算法的时间开销将为 $O(\max(|V|^2, |E| \log |E|))$ 。从这里又看到，当数据结构采用不同的实现方法时，同样算法的效率也不一样。

*7.4.4 Prim 算法的改进

前面 Prim 算法实现的一个缺点是可能把没价值的边存入队列 `cands`，使计算的空间复杂度达到 $O(|E|)$ （通常超过结点个数 $|V|$ ）。对不同的计算实例，实际加入队列的边的情况可能差异很大。

不难看到，为了实现 Prim 算法，实际上只需要记录连接顶点集合 U 和集合 $V-U$ 的边。这就是说，为完成这一计算，只需保存不超过 $|V|$ 条就够了。因此，有可能把算法的空间开销降到 $|V|$ 的水平。下面讨论在算法的实现中降低空间需求可能性。当然，在降低空间需求时，也不希望其时间复杂度变差。为讨论方便，下面把连接集合 U 中顶点和集合 $V-U$ 中顶点的边称为连接边，关键在于高效使用和操作这个边集合。

为了保证算法的效率，所用数据结构必须有效支持下面操作：

- 获得连接边中的最短边。显然可以采用堆结构，复杂度为 $O(\log |V|)$ 。
- 向 U 中加入新顶点，可能发现到 $V-U$ 中顶点的更短连接边。为此需要从结点出发找到与之对应的堆元素，修改其权值并恢复堆结构。

通过简单地扫描堆中元素，可以实现第二个操作，找到相应元素并修改其权值，然后再做一次筛选恢复堆结构。由于堆中元素只有堆序，找到如上操作的所需元素需要线性时间，操作的复杂度将是 $O(|V|)$ 。这样实现将会提高整个算法的复杂度。解决这个问题的关键是设法高效地修改堆元素的权值并恢复堆结构。如果不影响算法的时间复杂度，这个操作就必须在 $O(\log |V|)$ 时间内完成。这些分析说明，为了改进 Prim 算法的实现，需要一种类似优先队列的结构，它能支持（其中的元素个数为 n ）：

- $O(n)$ 时间建堆， $O(\log n)$ 时间取最小元操作 `getmin`（堆支持这些）。
- 从某种 `index`（在这里考虑的是顶点标号）出发的 $O(1)$ 的 `weight(index)` 操作，得到与 `index` 对应的权值；以及从 `index` 和新的权值出发， $O(\log n)$ 时间的减权值操作 `dec_weight(index, w)`。

称这种结构为可减权堆，设其定义类名为 `DecPrioHeap`，支持：

- 初始建堆 `DecPrioHeap(list)`，其中 `list` 中的项形式为 `(w, index, other)`。建立起来的这个类的对象 `decheap` 是以 `list` 的项为元素、以 `w` 为权的小顶堆。这个操作就是一般的建堆操作，按常规方式实现，但需要考虑下面操作的需要。
- `decheap.getmin()` 弹出堆中最小元并恢复堆， $O(\log n)$ 复杂度。
- `decheap.weight(ind)` 得到 `ind` 为 `index` 的元素的权值， $O(1)$ 时间复杂度。

^Θ 可参考张乃孝老师的《算法与数据结构》（高等教育出版社出版）。

- `decheap.dec_weight(ind, w, other)` 在 w 小于 `decheap` 里具有 ind 的元素的权值时，将该元素改为 $(w, ind, other)$ 并恢复堆的结构，要求复杂度为 $O(\log n)$ 。
- `decheap.is_empty()` 在 `decheap` 为空时返回 `True`, $O(1)$ 操作。

有了这个类之后，改造 Prim 算法的工作就可以完成了。这种结构是可以实现的，但有点复杂，留作练习（有一点难度）。

下面是基于 `DecPrioHeap` 改造后的算法：

```
def Prim(graph):
    vnum = graph.vertex_num()
    wv_seq = [[graph.get_edge(0, v), v, 0]
               for v in range(vnum)]
    connects = DecPrioHeap(wv_seq) # 连接的顶点堆, |V|个元素
    mst = [None]*vnum
    while not connects.is_empty():
        w, mv, u = connects.getmin() # 取得最近顶点和连接边
        if w == infinity: # 最近顶点不连通, 该图没有生成树
            break
        mst[mv] = ((u, mv), w) # 这就是新的 MST 边
        for v, w in graph.out_edges(mv): # 检查mv的邻接边
            if not mst[v] and w < connects.weight(v):
                connects.dec_weight(v, w, mv) # 更短就修改
    return mst
```

算法的空间复杂度降为 $O(|V|)$ ，时间复杂度是 $O(\max(|V| \log |V|, |E| \log |V|))$ 。

7.4.5 最小生成树问题

网络的最小生成树是一个计算问题，是许多实际问题的抽象，有非常重要的应用背景。人们对它做了很多进一步研究，取得了许多成果。

对于同一个问题，基于不同的想法有可能设计出许多不同的算法。例如，Prim 算法和 Kruskal 算法分别基于网络的 MST 性质和简单连通分量的最低代价互联，它们都能构造出任意一个图的最小生成树。进一步说，两者都是抽象的算法，基于它们都可能设计出多种不同的具体算法。其中可能选用不同的数据结构，具体过程也可能有异。不同的实现（实际算法）又可能具有不同的复杂度。

进一步说，由于算法具有一定的抽象性，尤其是针对抽象问题的算法，例如解决图上计算问题的算法。许多算法本身并不规定基础数据结构的表示方式，也不规定具体的实施方法。基于同一个算法，基于不同的辅助数据结构和操作，有可能做出许多不同的实现。由于这种情况，即使一个算法（本质上）很好，但如果实现不好，也不能充分发挥算法的优势。另一方面，算法的不同实现可能需要不同的支持结构，实现本身的复杂性也可能差异巨大。进一步说，基于某个算法的一套具体设计，可以（用某个编程语言）写出具体的程序代码。如果编程不当，也可能达不到算法可能达到的最高效率（复杂度），由此还需要理解所用的语言机制和数据结构。

回到最小生成树问题。由于这是一个非常重要的问题，有关研究一直在继续。近年的发展包括提出了一个证明了复杂度为 $O(|E|)$ 的随机算法（概率意义的复杂度），一个证明了复杂度为 $O(|E| \alpha(|E|, |V|))$ 的已知最优算法，其中的 α 是 Ackermann(n, n) 的逆函数[⊖]（2002 年）。还有一些并行算法方面的工作。

[⊖] Ackermann 函数是非常著名的函数，函数值增长极快。其逆函数几乎为常量函数，增长极慢。

有关最小生成树 (MST) 研究还没结束。研究者已经确定该问题的时间复杂度下界为 $O(|E|)$, 但还没找到这样的算法, 既没有证明这是下确界 (有可能达到), 也没找到更高的下界。有兴趣的读者可以查看网上其他相关材料, 如 MIT 出版社的《Introduction to Algorithms》(高等教育出版社影印)。

7.5 最短路径

本节讨论带权有向图或带权无向图 (网络) 上的路径问题。在这类图中, 从一些顶点到其他顶点可能有路径, 而且可能有多条不同路径, 因此就出现了怎样找路径, 怎样找到最好路径的问题。后者就是本节讨论的最短路径问题。在本节中提到的图, 总是指带权有向图或带权无向图。

7.5.1 最短路径问题

带权有向图或带权无向图 (网络) 中的每条边都附有一个权值, 通常用于表示实际应用中顶点之间联系的某种度量, 表示其关联的紧密程度, 例如长度、成本、代价等。这种权值一般具有可加性, 可以看作一种抽象的“距离”。

定义 (路径长度和顶点距离) 在网络或带权有向图里:

- 从顶点 v 到 v' 的一条路径上各条边的长度之和称为该路径的长度。
- 从 v 到 v' 的所有路径中长度最短的路径就是 v 到 v' 的最短路径, 最短路径的长度称为从 v 到 v' 的距离, 记为 $\text{dis}(v, v')$ 。

最短路径在实际应用中特别有意义, 许多调度问题与此有关, 例如:

- 运输 (最短里程, 最短运费, 最低成本, 最少时间等)。
- 加工或者工作的流程; 等等。

从顶点 v 出发通过适当的遍历过程, 可以确定能到达 v' 的所有路径, 自然可以从中找出最短路径。这种方法显然可行, 例如, 可以采用前面讨论过的深度优先或宽度优先搜索方法。但这不是很有效的方法, 其中还要处理一些麻烦 (例如, 如果存在环, 可达路径就有无穷多条)。请读者考虑如何写出这样的算法, 并分析算法的复杂度。

由于最短路径问题的重要性, 人们已经开发了一些更为有效的算法。

进一步考虑, 最短路径问题还可以分为单源点最短路径, 即从一个顶点出发到图中其余各顶点的最短路径问题; 以及所有顶点之间的最短路径问题。下面两小节将分别介绍针对这两个问题的两个有效算法。前面提出的求一个顶点到另一顶点的最短路径, 也是一个很清晰的问题, 但它没有特殊的有效算法。但是, 该问题显然可以采用单源点问题算法的同样模式求解, 一旦找到目标顶点, 就可以提前结束计算了。

7.5.2 求解单源点最短路径的 Dijkstra 算法

Dijkstra 算法是一个非常著名的算法, 由计算机科学家 E. W. Dijkstra [⊖] 提出。该算法能求出一个给定顶点到图中所有其他顶点的最短路径。这个算法也顺便解决了对给定起始顶点 v 和目标顶点 v' 求最短路径的问题。

[⊖] E. W. Dijkstra (迪杰斯特拉, 1930—2002), 著名计算机科学家, 在程序设计、编程语言、并发程序等领域做出了卓越贡献, 1972 年获图灵奖。

基本想法

Dijkstra 算法的限制是要求图中所有边的权不小于 0。显然大部分实际问题都满足这个要求。后来也有人提出了在允许负数权边的图中求单源点最短路径的算法，有关情况可以参看 Wiki 的有关页面或其他相关文献。Dijkstra 算法的基本想法和工作过程与 Prim 算法有些类似，利用了另一个与 MST 性质类似的性质。

假设现在要找图 G 中从顶点 v_0 到其他顶点的最短路径。在 Dijkstra 算法的执行过程中，也把图中的顶点分为两个集合：当时已知最短路径的顶点集合 U ，以及尚不知道最短路径的顶点集合 $V - U$ 。在算法的执行过程中逐步扩充已知最短路径的顶点集合，每步从顶点集合 $V - U$ 中找出一个顶点（它是当时已经能确定最短路径的顶点）加入 U 。反复执行这样的步骤，直至找到从顶点 v_0 到其他所有顶点的最短路径。该算法能同时给出这些最短路径及其长度（距离）。

剩下的问题就是如何找到适当的顶点扩充集合 U ，也就是说，如何在集合 $V - U$ 里找到下一个能确定最短路径的顶点？这一操作依赖于下面的一个性质。

如果上述想法能工作，在算法运行中的每一步，总是有些顶点在 U 中，另一些不在 U 中。为统一考虑所有顶点，针对程序执行中的每个时刻，为图中所有顶点定义一种与初始点 v_0 相关的统一度量，称为已知最短路径长度（或已知距离） $\text{cdis}(v_0, v)$ ：

$$\text{cdis}(v_0, v) = \begin{cases} \text{dis}(v_0, v), & \text{如果 } v \in U \\ \min \{ \text{dis}(v_0, u) + w(u, v) \mid u \in U \wedge w(u, v) \neq \infty \}, & \text{如果存在这样的 } u \\ \infty, & \text{其他} \end{cases}$$

其中 $w(u, v)$ 表示从 u 到 v 的边上的权。请特别注意中间一条，其意思是：如果已知从 v_0 到 u 的距离（由于 $u \in U$ ，其最短路径已知），而且存在从 u 到 v 的边，那么从 v_0 到 v 的当前已知距离，就是在所有经由满足上述条件的 u 的间接路径中最短的那一条路径的长度。显然，随着已知距离的顶点不断增加（随着 U 不断增长），有可能发现经由另一顶点的其他间接路径，因此可能使这种“当前已知距离”变小。特别的，有些顶点原来不知道间接路径，后来发现了到它的间接路径。

性质 7.7（已知的和实际的最短路径）如果 v' 在当前所有不属于 U 的顶点中 cdis 值最小，那么 $\text{dis}(v_0, v') = \text{cdis}(v_0, v')$ 。也就是说，从 v_0 到 v' 的当前已知距离就是其实际距离，因此到它的最短路径已知，现在就可以把 v' 加入顶点集合 U 。

这一性质不难证明（请读者自己想想）。根据上述性质，在构造最短路径的每一步，只需从所有当前还不属于 U 的顶点中选择 cdis 值最小的顶点加入 U 。

Dijkstra 算法梗概

假设现在要找图 G 中从顶点 v_0 到其余顶点的最短路径。根据上面的讨论，可以给出 Dijkstra 算法的梗概如下：

初始：

- 在集合 U 中放入顶点 v_0 ， v_0 到 v_0 的距离为 0。
- 对 $V - U$ 里的每个顶点 v ，如果 $(v_0, v) \in E$ （即存在直接的边），则到 v 的已知最短路径长度设为 $w(v_0, v)$ ，否则令 v 的已知最短路径长度为 ∞ 。这里的 $w(v_0, v)$ 是从 v_0 到 v 的边的权值。

反复做：

- 从 $V - U$ 中选出当时已知最短路径长度最小的顶点 v_{\min} 加入 U ，因为这时到 v_{\min} 的已知

最短路径长度 $\text{cdis}(v_0, v_{\min})$ 就是 v_0 到 v_{\min} 的距离。

- 由于 v_{\min} 的加入, $V-U$ 中某些顶点的已知最短路径可能改变。如果从 v_0 经过 v_{\min} 到 v' 的路径比原来已知的最短路径更短, 就说明发现了到 v' 的新的已知最短路径(及其长度), 该路径经过 v_{\min} 到 v' 。在这种情况下更新到 v' 的已知最短路径及距离的记录, 保证下面能正确地继续从 $V-U$ 中选择顶点。

反复选择顶点并更新到所有非 U 顶点的最短路径信息, 直到从 v_0 可达的所有顶点都在集合 U 中为止。如果这时还存在未加入 U 的顶点, 就说明被处理的图不连通(对于有向图, 是存在从 v_0 出发不可达的顶点)。

性质 7.8 (最短路径中前段也是最短路径) 如果 v' 是从初始点 v_0 到某顶点 v 的最短路径 p 上 v 的前一个顶点, 那么从路径 p 去掉最后顶点 v 得到的路径 p' 也是 v_0 到 v' 的最短路径。也就是说, 一条最短路径的前面任何一段都是 v_0 到这段路径的终点的最短路径。

这个性质很容易通过反证法证明: 如果路径 p' 不是 v_0 到 v' 的最短路径, 那么很显然, p 也不是 v_0 到 v 的最短路径。下面的算法里利用了这个性质。

【例 7.13】 图 7.11 里给出了 Dijkstra 算法的一个应用实例, 这里求解的是一个带权有向图中从顶点 a 出发到各顶点的最短路径。

图中的灰色线表示原图的边。图中各状态中的实线表示属于已经找到的某条最短路径中的边, 短划线表示当时 U 和 $V-U$ 之间的边界边。对于已经确定了有穷的已知最短距离的顶点, 在图中相应的顶点旁用圆括号括起的形式标出已知最短距离。为简便起见, 无穷大的距离都不标。如果已经确定了最短路径, 顶点边的距离值用方括号形式表示。到顶点 a 本身的距离 0 都没有标出。

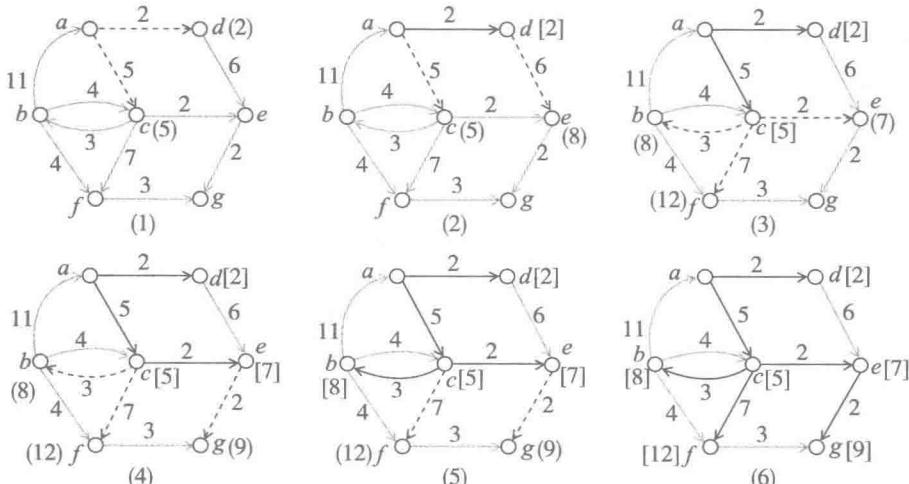


图 7.11 Dijkstra 算法实例

在初始状态(1), 只有 a 在集合 U , 这时有两条边界边分别到顶点 c 和 d 。选择距离 a 最近的 d 加入 U 并标记相应的边, 标出新发现的到顶点 e 的边界边, 得到状态(2)。这时的最近的非 U 顶点是 c , 将其加入 U 后发现了 3 条新的边界边。找到的到顶点 e 的新路径比原来的已知最短路径更短, 记录这时的边界边和顶点的已知距离, 得到状态(3)。将这时最近的非 U 顶点 e 加入 U , 记录到 e 的最短路径。由于新发现了从 e 一步可以到达 g , 记录相应的边界边, 得到状态(4)。这时虽然边界边中最短的是到 g 的边, 但顶点 b 距离 a 更近, 因此应该把

b 加入 U 。将 b 加入 U 后发现了一条到 f 的新路径，但其长度并不短于此时已知到 f 的最短路径（事实上，两条路径一样长），因此不需要更新路径，得到状态(5)。再经过两步，最后的状态(6)给出了所有路径。

Dijkstra 算法的 Python 实现

现在考虑如何定义一个函数，实现 Dijkstra 最短路径算法。函数的参数是被操作的图 graph 和图中的一个顶点 v0。设变量 vnum 记录图 graph 的顶点数。

算法中的一个问题是如何记录从 v0 到各顶点的最短路径。根据性质 7.8，要记录到顶点 v 的最短路径，只需记录 v0 到 v 最短路径上 v 的前一顶点。如果对每个顶点都有了这样的记录，就可以追溯这些记录，找出从 v0 到顶点 v 的最短路径。由此可见，记录所有最短路径只需要用一个 vnum - 1 个元素的边集合。

下面算法里用一个 vnum 元的表 paths 记录路径，其元素 paths[v] 的形式为 (v', p)，说明从 v0 到顶点 v 的最短路径上的前一顶点是 v'，该最短路径的长度是 p。此外，函数里还用 paths[v] 取值 None 表示 v 还不在 U 里。

求解最短路径的候选边集以路径长度作为排序码记录在优先队列 cands 里，队列元素形式为 (p, v, v')，表示从 v0 经 v 到 v' 的已知最短路径的长度为 p。根据 p 的值在 cands 里排序，保证总选出最近的未知距离顶点。

每次选出的具有最小 p 值的边。如果其终点 v' 在 $V - U$ ，就将其加入 paths，并将经由 v' 可达的其他顶点及其路径长度记入 cands。显然，如果 cands 已有到某顶点的路径，但后来发现的新路径更短，优先队列保证最短的路径被先行取出，满足算法的需要。

```
def dijkstra_shortest_paths(graph, v0):
    vnum = graph.vertex_num()
    assert 0 <= v0 < vnum
    paths = [None]*vnum
    count = 0
    cands = PrioQueue([(0, v0, v0)])           #初始队列
    while count < vnum and not cands.is_empty():
        plen, u, vmin = cands.dequeue()          #取路径最短顶点
        if paths[vmin]:                         #如果其最短路径已知则继续
            continue
        paths[vmin] = (u, plen)                  #记录新确定的最短路径
        for v, w in graph.out_edges(vmin):        #考察经由新U顶点的路径
            if not paths[v]:                      #是到尚未知最短路径的顶点的路径，记录它
                cands.enqueue((plen + w, vmin, v))
        count += 1
    return paths
```

这个算法的基本结构和 Prim 算法相同，只是记入优先队列的“权值”不同，而且这些权值是在准备入队的时刻计算出来的。

Dijkstra 算法的复杂度

现在考虑 Dijkstra 算法的复杂度。

时间复杂度：

- 算法中初始化部分的时间复杂度不超过 $O(|V|)$ 。
- 算法主体部分与 Prim 算法类似，复杂度是 $O(|E| \log |E|)$ 。

算法的空间复杂度是 $O(\max(|E|, |V|)) = O(|E|)$ ，主要是表 paths 需要保存 $|V|$ 个顶点的信息，而在优先队列 cands 可能保存 $O(|E|)$ 条边的相关信息。

不难看到在这个算法里，一旦找到了到某个顶点的更短路径，可以设法直接更新与此顶点相关的信息，这一点与 Prim 算法里的情况类似。采用前面讨论过的“可减权堆”，可以做出一个只需要 $O(|V|)$ 空间的算法，有关算法请读者自己考虑。

7.5.3 求解任意顶点间最短路径的 Floyd 算法

本小节研究图中各对顶点之间的最短路径问题。一个显然解法是多次执行 Dijkstra 算法，依次把图中的各顶点作为初始顶点。有关实现留作练习。

本小节将给出另一个想法完全不同的算法，它能一次计算出所有最短路径。这个算法称为 Floyd 算法（或 Floyd-Warshall 算法），由 R. Floyd[⊖]提出，其中采用与 S. Warshall 的图中可达性算法相同的技术，可以直接计算出各对顶点间的所有最短路径及其长度。

基本想法

Floyd 算法基于图的邻接矩阵表示，所做的基本工作就是求出可达（有边相邻）关系的传递闭包，但同时还要记录求出的所有路径及其长度。设 n 个顶点的图 $G=(V, E)$ 的邻接矩阵是 A ，其中对角线元素的值都是 0，表示从各顶点到自身的距离为 0，其余元素是权值，无边的情况用 ∞ 表示。算法的基本想法是：

- 如果有边 $(v, v') \in E$ ，那么它自然是顶点 v 到 v' 的路径，其长度可以由边的权值直接得到，即是 $A[v][v']$ 。无边时可以看作存在长度为 ∞ 的直接路径。
- 但是，从 v 到 v' 的直接路径未必是从 v 到 v' 的最短路径，有可能存在从 v 到 v' 更短路径，途中经过其他顶点。
- 算法的考虑是采用一种系统化的方法，检查和比较从 v 到 v' 的可能经过任何顶点的所有路径，从中找出最短路径。

问题就在于对于所有的顶点对，如何同时有效地完成这种检查（和计算）。

Floyd 等人开发出一种机械的方法，其算法的具体过程如下，这里假定图 G 的顶点顺序排列为 $v_0, v_1, \dots, v_i, \dots, v_{n-1}$ 。

开始：对每对 v 和 v' ，从 v 到 v' 的途中不经过任何顶点的路径长度已知。如果存在从 v 到 v' 的边，这个长度就是该边的权，无边时认为存在长度为 ∞ 的路径。

$k=0$ ：对每对 v 和 v' ，除前一步已知的路径外，从 v 到 v' 的途经顶点的下标不大于 k （此时就是不大于 0，实际上是只能经过顶点 v_0 ）的路径可分为两段（如果没有路径，就认为存在长度为 ∞ 的路径。下同）：

$$\langle v, v_0 \rangle, \langle v_0, v' \rangle$$

这一路径的长度是两段路径的长度之和。比较这一“新路径”和前一步已知的路径（是已知路径中最短的），可以确定从 v 到 v' 的途经顶点的下标不大于 0 的最短路径。

$k=1$ ：对每对 v 和 v' ，除至此已知的路径外（路径中途经顶点的下标小于等于 0），从 v 到 v' 的途经顶点下标不大于 k （现在是不大于 1）的路径可以分为两段：

$$\langle v, \dots, v_1 \rangle, \langle v_1, \dots, v' \rangle$$

在这两段路径内部经过的顶点的下标都不大于 0，路径及其长度都已在前一步确定。这种新路径的长度是两段路径的长度之和。用这样确定的新路径与从 v 到 v' 的已知最短路径（其中途经的顶点的下标小于等于 0）比较，就可确定从 v 到 v' 的途经顶点的下标小于等于 1 的最短路径。

[⊖] R. Floyd (1936.6.8—2001.9.25)，著名计算机科学家，在算法、软件、程序语言和程序验证领域做出了重要贡献，1978 年获得图灵奖。

$k=2$: 类似, 从略。

.....

考虑一般的 k :

对于每对顶点 v 和 v' , 前面步骤已经考察了所有的从 v 到 v' 的途经顶点的下标小于等于 $k-1$ 的路径, 并已获知这些路径中的最短路径及其长度。

在这一步, 对每对顶点 v 和 v' , 考虑从 v 到 v' 的途经顶点的下标小于等于 k 的所有路径。其中已经考察过的路径的途经顶点的下标都小于等于 $k-1$, 显然, 尚未考察过的路径都可以分为两段:

$$\langle v, \dots, v_k \rangle, \langle v_k, \dots, v' \rangle$$

在这两段路径中途经顶点的下标都小于等于 $k-1$, 两段的长度在做这一步之前都已经知道, 这样的新路径的长度就是两段路径的长度之和。用该路径与已知的从 v 到 v' 的最短路径 (其中途经的顶点下标都小于等于 $k-1$) 比较, 就可确定从 v 到 v' 的途经顶点的下标小于等于 k 的最短路径。

.....

如此继续, 直到做完 $k=n-1$ 的情况, 也就是说, 考察完从 v 到 v' 的途经顶点的下标不大于 $n-1$ 的所有路径之后, 对每对 v 和 v' , 已经确定了从 v 到 v' 的所有可能路径中的最短路径。算法结束。

虽然上面假定结点的下标为 0 到 $n-1$, 对下标为 1 到 n 也可以类似定义。

Floyd 算法的实现

要实现 Floyd 算法过程, 需要用递推的方式生成一系列 $n \times n$ 方阵 A_k ($0 \leq k \leq n$), 其中 $A_k[i][j]$ 表示从 v_i 到 v_j 的途经顶点可为 v_0, v_1, \dots, v_{k-1} 的最短路径的长度。

矩阵 A_0 就是图的邻接矩阵 A , $A_0[i][j]$ 是图中 v_i 到 v_j 的边的权, 也就是从 v_i 到 v_j 的不经过任何顶点的最短路径长度。最后, $A_n[i][j]$ 是从 v_i 到 v_j 的最短路径的长度。

矩阵序列 A_0, A_1, \dots, A_n 可以递推计算 ($0 \leq i, j \leq n-1$):

- $A_0[i][j] = A[i][j]$, 直接由邻接矩阵得到。
- 对一般的 k , $A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k] + A_k[k][j]\}$, 其中 $0 \leq k \leq n-1$ 。在这一步新考虑了所有途经顶点 v_k 的路径, 因此, $A_{k+1}[i][j]$ 就是从 v_i 到 v_j 的途经顶点的下标不大于 k 的最短路径的长度。
- 做到最后, $A_n[i][j]$ 为从 v_i 到 v_j 的最短路径的长度。

在这一递推过程中生成一系列矩阵, 但是不是每步都需要做一个新矩阵? 假设现在已经算出了矩阵 A_k , 现考虑 A_{k+1} 的计算。上面给出的公式是:

$$A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k] + A_k[k][j]\}$$

也就是说, 新矩阵中的 $A_{k+1}[i][j]$ 或者就是 $A_k[i][j]$ (如果它比较小), 或者是通过对矩阵中的第 k 列和第 k 行中的元素 (下面 $2n$ 个元素) 求和计算出来的:

$$\begin{aligned} & A_k[0][k], A_k[1][k], \dots, A_k[n-1][k] \\ & A_k[k][0], A_k[k][1], \dots, A_k[k][n-1] \end{aligned}$$

注意: 如果在计算 A_{k+1} 的过程中得到的矩阵里第 k 行或第 k 列元素与 A_k 中对应元素不同, 就不能直接在原矩阵里修改, 因为那样做后再取元素 $A[i][k]$ 或者 $A[k][j]$, 得到的就不是原来 A_k 的元素, 而是修改过的值。出现这种情况, 就必须另用一个新矩阵实现 A_{k+1} 。也就是说, 如果计算下一矩阵的过程中有可能修改后面还要用的元素, 就需要另用一个新矩阵; 否则就不需

要另建新矩阵，可以直接在原来的矩阵里修改。

实际上，计算 A_{k+1} 的过程中不会修改矩阵第 k 行或第 k 列的元素，因为：

$$A_{k+1}[i][k] = \min\{A_k[i][k], A_k[i][k] + A_k[k][k]\}$$

$$A_{k+1}[k][j] = \min\{A_k[k][j], A_k[k][k] + A_k[k][j]\}$$

而对任何的 k , A_k 的对角线元素 $A_k[m][m]$ 总是 0 (对所有的 m)，也就是说总有：

$$A_{k+1}[i][k] = A_k[i][k] \quad (\text{第 } k \text{ 行不变})$$

$$A_{k+1}[k][j] = A_k[k][j] \quad (\text{第 } k \text{ 列不变})$$

因此，在整个计算中可以用一个二维表 A 实现所有的 A_k ，递推计算新矩阵可以通过直接修改 A 中元素的方式实现。在计算 $A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k] + A_k[k][j]\}$ 时，如果需要修改矩阵元素，就直接用赋值 $A[i][j] = A[i][k] + A[k][j]$ 实现。

算法还需要给出所有最短路径。这里的做法是另计算一系列 n 阶方阵 N_k (下面代码里用 $nvertex$)，其中 $N_k[i][j]$ 的值是从 v_i 到 v_j 的中间可经过顶点 v_0, v_1, \dots, v_{k-1} 的最短路径上，顶点 v_i 的后继顶点 v_l 的下标 (与前面 A_k 对应)。由于从 v_l 到 v_j 的最短路径也有记录，可以根据它查找下一个后继顶点，直至得到整个路径。

- 初始时，如果 $A_0[i][j] = \infty$ (没有边)，则令 $N_0[i][j] = -1$ ，否则就令 $N_0[i][j] = j$ ，表示在 v_i 到 v_j 的路径上 v_i 的后继顶点是 v_j 。
- 在由 v_k 计算 A_{k+1} 时，如果 $A_{k+1}[i][j]$ 被设为 $A_k[i][k] + A_k[k][j]$ ，就设置 $N_{k+1}[i][j]$ 等于 $N_k[i][k]$ ，表示在 v_i 到 v_j 的路径上 v_i 的后继顶点，就是已知的从 v_i 到 v_k 的路径上 v_i 的后继顶点。这一轮计算完成后，每个 $N_{k+1}[i][j]$ 都是在从 v_i 到 v_i 的可以途经顶点 $v_0, v_1, \dots, v_{k-1}, v_k$ 的路径上 v_i 的后继顶点。
- 整个计算完成时， $N_n[i][j]$ 就是从 v_i 到 v_j 的最短路径上 v_i 的后继顶点。追溯这个矩阵，可得到任何一对结点之间的最短路径。

实现 Floyd 算法的 Python 函数

函数 `all_shortest_paths` 里用了两个 $n \times n$ 矩阵作为工作区，其中 `a` 记录已知最短路径长度，`nvertex` 记录已知最短路径上的下一顶点。

函数定义如下：

```
def all_shortest_paths(graph):
    vnum = graph.vertex_num()
    a = [[graph.get_edge(i, j) for j in range(vnum)]
         for i in range(vnum)] # create a copy
    nvertex = [[-1 if a[i][j] == inf else j
                for j in range(vnum)]
               for i in range(vnum)]

    for k in range(vnum):
        for i in range(vnum):
            for j in range(vnum):
                if a[i][j] > a[i][k] + a[k][j]:
                    a[i][j] = a[i][k] + a[k][j]
                    nvertex[i][j] = nvertex[i][k]
    return (a, nvertex)
```

虽然前面的讨论很复杂很长，但写出的程序却极其简单。前面的讨论已经论证了这一算法确实能正确完成工作。

Floyd 算法的复杂度

与程序的结构对应，Floyd 算法的复杂度分析也非常简单。

时间复杂度：算法的初始化部分生成两个各包含 $|V|^2$ 个元素的矩阵，时间复杂度为 $O(|V|^2)$ 。迭代构造长度矩阵 a 和路径矩阵 n_{vertex} 的部分是三重循环，时间复杂度为 $O(|V|^3)$ ，这也是 Floyd 算法的时间复杂度。

空间复杂度：使用了两个矩阵完成计算并存放结果，另外用了几个辅助变量。显然，算法需要 $O(|V|^2)$ 的空间。

最短路径问题

本节讨论了最短路径问题和两个算法，这些算法里都蕴涵着有趣的想法。

Dijkstra 算法基于类似于最小生成树的思想，所做的也是一种“宽度优先搜索”，其中利用了一种类似 MST 的性质，按路径的长度逐步扩张。这个算法在探索中及时更新已知的最短路径，每一步找到一个可确定最短路径的顶点，同时也找到了到达该顶点的最短路径。一步完成后更新路径信息，保证记录的都是至今已知的最短路径。这是典型的动态规划方法（在计算过程中保留一些信息，支持过程中的动态决策）。

Floyd 算法基于完全不同的考虑，其目标是直接求出所有顶点之间的最短路径及其长度。这个算法的基本做法也是为了问题的最终解决逐步积累信息，根据已有的信息，不断更新包含着解的部分信息的记录，最终得到问题的解。这一算法也被认为是一个典型的动态规划算法。在算法执行的过程中，逐步求出越来越接近原问题的子结构（子问题）的最优解，最后得到原问题的最优解。

两个算法都值得认真学习理解。当然，其中 Floyd 算法不适合人工操作，因为其中的操作非常机械而繁琐，比较缺乏直观。

7.6 AOV/AOE 网及其算法

本节讨论两种有着广泛应用背景的网络（实际上是有向图），介绍两个重要的计算问题和解决它们的两个算法。

7.6.1 AOV 网、拓扑排序和拓扑序列

本小节考虑有向图的一类应用：用图中的顶点表示某个有一定规模的“工程”里的不同活动，用图中的边表示各项活动之间的先后顺序关系。这样的有向图称为顶点活动网（activity on vertex network），或称 AOV 网。

工程和工作安排

在一项工程中，经常存在一些具有一定独立性的工作任务（活动）。由于各项工作的特点，不同工作之间存在着一些制约关系，完成了一些工作之后才能开始另一些工作。这里需要解决的一个问题就是做出满足制约关系的工作安排。把 AOV 网应用到这里，可以用图中的边表示工作之间的制约关系，通过对 AOV 网络的处理做出工程计划。

对于一些实际问题，AOV 网的顶点或边还可能带有权值。可以考虑如“最优”安排问题，求出所有可能计划中的最佳计划，网络中的流问题等。

【例 7.14】 一种常见 AOV 网实例是大学课程的先修关系。课程知识有前后联系，一门课可能以其他课程的知识为基础，学生想选修某门课程时，要看是否已修过所有先修课程。

下面表格里列出了计算机专业若干课程及其先修课程：

课程编号	课程名称	先修课程
C1	高等数学	
C2	程序设计基础	
C3	数据结构	C1, C2
C4	离散数学	C1
C5	普通物理	C1
C6	编译原理	C2, C3, C4
C7	计算机原理	C3, C4, C5
C8	操作系统	C3, C4, C6
C9	数据库原理	C3, C7, C8
C10	计算机网络	C4, C7, C8

图 7.12 给出的是与上面课程表里的课程及其先修关系对应的 AOV 网络，其中顶点代表课程，边表示先修关系。

拓扑排序和拓扑序列

可以把 AOV 网络里的有向边看作一种“顺序”关系。拓扑排序问题就是问，在一个 AOV 网里的活动能否排成一种全序。

定义 对于给定的 AOV 网 N ，如果 N 中的所有顶点能排成一个线性序列

$$S = v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_{n-1}}$$

满足：如果 N 中存在从顶点 v_i 到顶点 v_j 的路径，那么 S 里 v_i 就排在 v_j 之前，则 S 称为 N 的一个拓扑序列，而构造拓扑序列的操作称为拓扑排序。

一个 AOV 网未必有拓扑序列。不难证明，一个 AOV 网存在拓扑序列，当且仅当它不包含回路（存在回路，意味着某些活动的开始要以其自己的完成作为先决条件，这种现象称为活动之间的死锁）。例如，图 7.13 里的两个 AOV 网都没有拓扑序列。

下面列出拓扑排序的一些性质。

性质 7.9 如果一个 AOV 网有拓扑序列，其拓扑序列未必唯一。

例如，下面是图 7.12 中的 AOV 网的两个拓扑序列：

C1, C2, C3, C4, C5, C6, C7, C8, C9, C10

C2, C1, C4, C3, C6, C8, C5, C7, C10, C9

性质 7.10 将 AOV 网 N 的任一个拓扑序列反向得到的序列，都是 N 的逆网（即是把 N 的每条边反转得到的那个 AOV 网）的一个拓扑序列。

假设用 AOV 网表示一个工程：网中顶点表示工程中的活动（工序、任务等），顶点之间的有向边代表活动之间的制约关系（前一活动完成后，后一活动才能进行）。如果在实际条件下工程中的动作只能串行进行，则那么该 AOV 网的一个拓扑序列，就是整个工程得以顺利完成的一种可行方案。

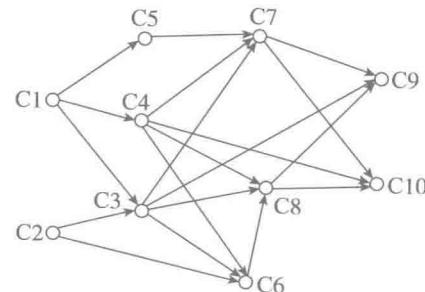


图 7.12 AOV 网实例：课程的关系

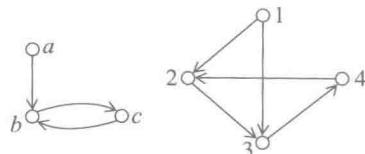


图 7.13 两个不存在拓扑序列的 AOV 网

7.6.2 拓扑排序算法

任何无回路 AOV 网 N 都可以做出拓扑序列，方法很简单：

- 从 N 中选出一个入度为 0 的顶点作为序列的下一顶点。
- 从 N 网中删除所选顶点及其所有的出边。
- 反复执行上面两步操作，直到已经选出了图中的所有顶点，或者再也找不到入度非 0 的顶点时算法结束。

如果剩下入度非 0 的顶点，就说明 N 中有回路，不存在拓扑序列。

【例 7.15】根据上述算法，很容易做出图 7.12 中课程先修关系 AOV 网的拓扑序列。下面是另外两个不同的拓扑序列：

C1, C5, C4, C2, C3, C6, C8, C7, C10, C9

C1, C4, C5, C2, C3, C7, C6, C8, C9, C10

实际上，该 AOV 网的拓扑序列还有很多，请读者自己考虑。

拓扑排序很有用，但它有一个隐含假定：AOV 网中的活动只能一个个地顺序进行。实际中的问题可能更加复杂，是另一个类似问题。以课程安排为例，学生完全可以同时学习多门课程，只要它们的先修课程都已经学过了。如果考虑这种情况，就可以问一些其他问题，例如，至少需要几个学期可以完成上述课程？

实现技术和函数定义

现在考虑拓扑排序的程序实现。直接基于图（的拷贝）和顶点删除应该能完成工作，但需要拷贝整个图，空间代价比较高；又需要在图中不断找入度为 0 的顶点，时间代价也会很高。下面考虑另一种方法，设法有效记录所需信息，降低操作代价。

顶点之间的制约关系决定了顶点的入度。入度是一个整数，用一个整数表就能记录所有顶点的入度。下面算法里用一个表 `indegree`，以顶点为下标。初始时，将表中各元素设置为对应的图中顶点的入度。在随后的计算中，一旦选中一个顶点，就可以根据其出边的情况将其邻接点的入度分别减一。

这里又出现了另一问题：工作中需要反复找出入度为 0 的顶点。通过扫描 `indegree` 的方法，需要花费线性时间，效率低。实际上，只有入度减一操作有可能把顶点的入度变成 0，如果这时记下这种顶点，后面需要顶点时就可以直接取用了。

为了处理这类情况，人们提出了一种技巧：在 `indegree` 里维持一个“0 度表”，表中记录当时已知的所有入度为 0 但还没有处理的顶点。具体做法是：用一个变量 `zerov` 记录“第一个”入度为 0 的顶点的下标；用表元素 `indegree[zerov]` 记录下一个人度为 0 的顶点的下标；如此类推。如果最后一个人度为 0 的顶点的下标是 v ，就在 `indegree[v]` 中存入 -1，表示“0 度表”到此结束。

这个“0 度表”就像在 `indegree` 里保存了一个顶点栈：变量 `zerov` 记录栈顶的位置（下标），-1 表示栈结束。如果发现新的 0 度顶点，例如 v ，就把当时的 `zerov` 值存入 `indegree[v]`，然后把 v 存入 `zerov`。这相当于将 v 入栈。如果要选一个 0 度元素，就用 `zerov` 的值，并把 `zerov` 修改为 `indegree[zerov]` 的值（对应于栈元素的弹出）。

`topological_sort` 的基本工作过程是：

- 确定所有顶点的入度存入 `indegree`，用 0 度表记录其中入度为 0 的顶点。
- 反复选择入度为 0 的顶点并维护 0 度表。
- 最后返回拓扑序列，失败（无拓扑序列）时返回 `False`。

函数定义如下：

```
def toposort(graph):
    vnum = graph.vertex_num()
    indegree, toposeq = [0]*vnum, []
    zerov = -1
    for vi in range(vnum):
        for v, w in graph.out_edges(vi):
            indegree[v] += 1
    # 建立初始的入度表
    for vi in range(vnum):
        if indegree[vi] == 0:
            indegree[vi] = zerov
            zerov = vi
    # 建立初始的0度表
    for n in range(vnum):
        if zerov == -1:
            return False
        vi = zerov
        # 不存在拓扑序列
        zerov = indegree[zerov]
        toposeq.append(vi)
        # 把一个vi加入拓扑序列
        for v, w in graph.out_edges(vi):
            indegree[v] -= 1
            # 检查vi的出边
            if indegree[v] == 0:
                indegree[v] = zerov
                zerov = v
    return toposeq
```

算法分析

时间复杂度：

- 设置 `indegree` 初值用了一个两重循环，时间复杂度为 $O(\max(|E|, |V|))$ ，检查入度为零的顶点并建立 0 度表需要 $O(|V|)$ 时间。
- 工作的主要部分是一个两重循环，时间复杂度也是 $O(\max(|E|, |V|))$ 。
- 整个算法的时间复杂度为 $O(|E| + |V|)$ ，对于连通图就是 $O(|E|)$ 。

空间复杂度：`indegree` 和 `toposeq` 都是 $|V|$ 规模的数组，显然，这个算法的空间复杂度是 $O(|V|)$ 。

如果图用邻接矩阵表示，矩阵里可能出现许多表示无邻接边的元素。如果矩阵比较稀疏，就会浪费很多空间，处理这种无用的边也要多花费很多时间：为正确设置顶点入度，需要检查每条边一次，时间是 $O(|V|^2)$ ，主循环的时间复杂度也是 $O(|V|^2)$ 。因此，如果处理采用邻接矩阵表示的图，这个算法时间复杂度就是 $O(|V|^2)$ 。

7.6.3 AOE 网和关键路径

AOE 网（Activity On Edge network）是另一类常用的带权有向图。这是一类非常重要的 PERT（Program Evaluation and Review Technique，规划评估和评审技术）模型，最早是在美国军方支持下开发出来的，用于大型工程的计划和管理。其雏形曾在 20 世纪 40 年代用于美国原子弹开发的曼哈顿计划，有广泛的实际工程应用。

抽象地看，AOE 网是一种无环的带权有向图，其中：

- 顶点表示事件，有向边表示活动，边上的权值通常表示活动的持续时间。
- 图中一个顶点表示的事件，也就是它的入边所表示的活动都已完成，它的出边所表示的活动可以开始的那个状态，把这一情况看作事件。

实际工程或复杂事务里的一批相关活动(工作项目、任务等),可以用一个AOE网描述(抽象),然后就可以基于这个网考虑活动的安排了。

【例7.16】图7.14给出的AOE网中包括15项活动、9个事件。图中标的是 $a_i:n$ 表示该边代表的活动名为 a_i ,权值为 n 。图中事件 v_0 表示整个工程可以开始的状态;事件 v_4 表示活动 a_5 、 a_8 已经完成,活动 a_{10} 、 a_{11} 可以开始的状态,事件 v_8 表示整个工程结束。

图中显示,活动 a_0 需要7个单位时间完成,活动 a_1 需要13个单位时间完成,等等。整个工程开始,活动 a_0 、 a_1 、 a_2 就可以同时开始了,而活动 a_3 、 a_4 需要等到事件 v_1 发生后才能开始, a_5 、 a_6 、 a_7 要等到事件 v_2 发生之后才能进行,如此等等。当活动 a_{12} 、 a_{13} 、 a_{14} 完成时,整个工程就完成了。

AOE网中描述的活动可以并行地进行,只要一项活动(一条边)的前提事件均已发生(也就是说,以该边的始点为终点的所有活动都已经完成),这项活动就可以开始。所以,完成整个工程的最短时间,就是从开始顶点到完成顶点的最长路径的长度(即,路径上各条边的权值之和)。这种最长路径称为AOE网的关键路径,AOE网上最重要的一项计算工作是找出其中的关键路径。

7.6.4 关键路径算法

现在考虑如何开发出一种方法,能确定AOE网 $G=(V, E)$ 里的关键路径。下面假定顶点 v_0 是 G 中的开始事件, v_{n-1} 是结束事件, $w(\langle v_i, v_j \rangle)$ 为边 $\langle v_i, v_j \rangle$ 的权。

首先定义几组变量,用它们记录关键路径计算中确定的信息:

- 1) 事件 v_j 的最早可能发生时间 $ee[j]$ 。显然,这一时间要根据在它之前的事件(顶点)和相关活动(边及其权值)确定,不可能更早发生。 $ee[j]$ 可以递推计算:

$$ee[0]=0 \text{ (初始事件总在时刻 } 0 \text{ 发生)}$$

$$ee[j]=\max \{ee[i]+w(\langle v_i, v_j \rangle) \mid \langle v_i, v_j \rangle \in E\}, \quad 1 \leq j \leq n-1$$

对每个 j ,只需要考虑以 v_j 为终点的入边集。

- 2) 事件 v_i 的最迟允许发生时间 $le[i]$ 。事件的发生有可能推迟更晚,但有些事件过迟发生就会延误整个工程的进度。可以根据已知的 ee 值反向地递推计算:

$$le[n-1]=ee[n-1] \text{ (最后一个事件绝不能再延迟)}$$

$$le[i]=\min \{le[j]+w(\langle v_i, v_j \rangle) \mid \langle v_i, v_j \rangle \in E\}, \quad 0 \leq i \leq n-2$$

与上面类似,对每个 i ,只需要考虑以 v_i 为始点的出边集。

- 3) 在这个网络中活动 $a_k=\langle v_i, v_j \rangle$ 的最早可能开始时间 $e[k]=ee[i]$,以及它的最迟允许开始时间 $l[k]=le[j]-w(\langle v_i, v_j \rangle)$ (只要该活动的实际开始不晚于这个时间,就不会拖延整个工程的工期)。

活动集合 $A=\{a_k \mid e[k]=l[k]\}$ 中的所有活动称为这个AOE网里的关键活动,因为它们中的任何一个推迟开始,都会延误整个工程的工期。 $E-A$ 中的活动为非关键活动。对于非关键活动, $l[k]=e[k]$ 不等于0,这个差表示开始活动 a_k 的时间余量,这是在不延误整体工期的前提下,活动 a_k 可以推迟开始的时限。所有完全由关键活动构成的从初始点到终点的路径就是图 G 中的关键路径。关键路径可能不止一条,可以同时得到。

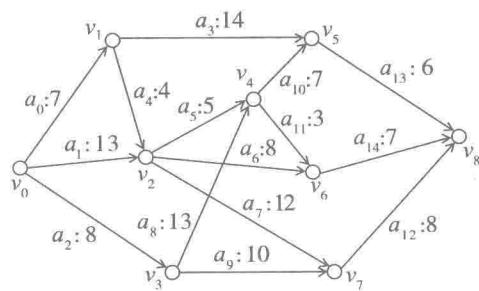


图7.14 AOE网实例

【例 7.17】 图 7.15 给出了求图 7.14 中 AOE 网关键路径的过程示意。图中每个顶点所标的 $[ee, le]$ 中的两项记录有关事件的最早可能发生时间和最迟允许发生时间。图 7.15 左图给出了通过一步步向前推算得到 ee 值的情况，右图给出了通过一步步回向推算得到 le 值的情况。基于这两组值就能确定这个 AOE 网里的关键活动，在右图里用粗线箭头标出。这里有两条关键路径。

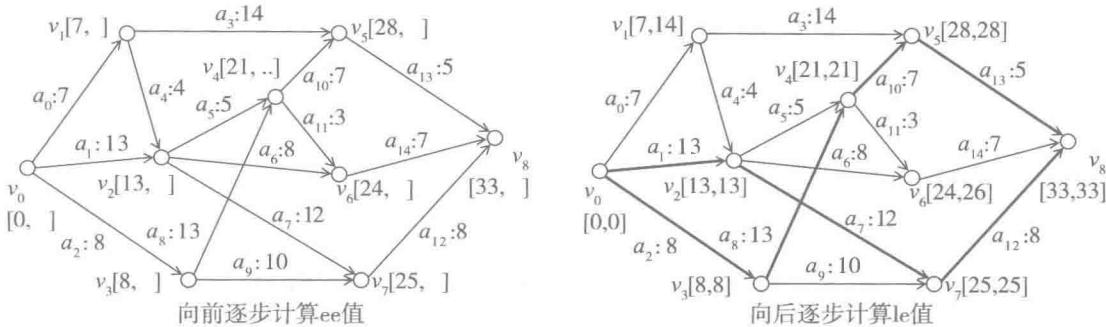


图 7.15 AOE 网实例

关键路径：算法

前面讨论中提出的关键路径算法，可以直截了当地翻译为一个 Python 实现，其中一步步计算出有关信息。在下面的实现中用两个 list 表示 e_e 和 l_e ，它们都以 AOE 网中的顶点为下标，其中每项记录对应于一个事件的时间。

应该注意，为正确生成 ee 和 le 的数据，计算需要按一定的顺序进行。对于 ee ，只有算出了前面制约事件的 ee 值之后，才能进一步计算被制约事件的 ee 值。对 le 的计算顺序正好相反。不难看到，对 ee 的可行计算顺序就是 AOE 网上的拓扑顺序，可以按顶点的任何拓扑序列计算，而对 le 的正确计算顺序就是逆拓扑顺序，可以按拓扑序列反向计算。因此，在正式开始计算之前，需要先得到该 AOE 网的一个拓扑序列。

算法实现分为下面几步：

- 1) 生成 AOE 网的一个拓扑序列。
 - 2) 生成 ee 表的值，应该按拓扑序列的顺序计算。
 - 3) 生成 le 数组的值，应该按拓扑序列的逆序计算。
 - 4) 最后，数据组 e 和 l 可以一起计算（很简单）。由于只希望得到关键路径，下面并不显示表示这两组数据，而是直接求出关键活动。

在下面算法里，步骤 1 直接调用前面的拓扑排序过程，步骤 2 和 3 定义为独立的内部过程，最后一步直接计算并收集确定的关键活动，函数返回得到的所有关键活动可能表示多条关键路径：

```

def event_latest_time(vnum, graph, toposeq, eelast):
    le = [eelast]*vnum
    for k in range(vnum-2, -1, -1):  # 逆拓扑顺序
        i = toposeq[k]
        for j, w in graph.out_edges(i):
            if le[j] - w < le[i]:  # 事件i应更早开始?
                le[i] = le[j] - w
    return le

def crt_paths(vnum, graph, ee, le):
    crt_actions = []
    for i in range(vnum):
        for j, w in graph.out_edges(i):
            if ee[i] == le[j] - w:  # 关键活动
                crt_actions.append((i, j, ee[i]))
    return crt_actions

toposeq = toposort(graph)
if not toposeq:  # 不存在拓扑序列，失败结束
    return False
vnum = graph.vertex_num()
ee = events_earliest_time(vnum, graph, toposeq)
le = event_latest_time(vnum, graph, toposeq, ee[vnum-1])
return crt_paths(vnum, graph, ee, le)

```

下面简单讨论一些实现细节。

函数 `events_earliest_time` 建立 `ee` 表时将其元素初始化为 0。随后的循环按拓扑序列逐个处理顶点，一旦发现结束时间更晚的路径，就更新相应的 `ee` 值（最早可能时间）。函数 `event_latest_time` 首先把 `le` 的元素都赋为工程结束顶点的时间，然后按拓扑排序的逆序向前逐个更新最迟允许时间（表 `le` 的元素）。函数 `crt_paths` 搜集起所有关键事件，最后返回它们的表。表中元素的形式是序对 (i, j, t) ，表示从顶点 i 到顶点 j 的事件应该在 t 时刻开始进行。

算法复杂度

在图的邻接表表示上执行本算法，拓扑排序的时间复杂度为 $O(|V|+|E|)$ ；求事件的最早可能时间和允许最迟时间、活动的最早开始时间和最晚开始时间，需要检查图中每个顶点和每个顶点边表中所有边，各检查一次，时间复杂度均为 $O(|V|+|E|)$ 。因此，求关键路径算法的时间复杂度为 $O(|V|+|E|)$ 。如果图采用邻接矩阵表示，算法就需要 $O(|V|^2)$ 时间。

空间复杂度：算法中需要保存拓扑序列和事件时间的表，空间复杂度为 $O(|V|)$ 。

总结一下本节的两个算法：

- 拓扑序列是有向无环图的一个重要概念，拓扑排序算法的思想很简单。但是，求拓扑序列是许多有向图算法的基础，这个概念和算法都很重要。
- 关键路径是带权有向无环图的一种重要概念，广泛用于工程规划领域。算法需要按拓扑顺序遍历结点，计算顶点和边的最早和最迟时间。

本章总结

图是一类比较复杂的非线性数据结构。本章开始介绍了图的基本概念和一些性质，而后讨论了图的一些实现技术，以及实现中的一些问题。

图的两种最典型实现技术是邻接矩阵表示法和邻接表表示法。邻接表可以看作邻接矩阵的

一种压缩形式，其优点是可能节约存储，特别是对极为稀疏的图（例如，图中边的条数 $|E|$ 与顶点的个数 $|V|$ 成正比）。在实际应用中需要处理很多大型的图，这些图通常都很稀疏。本章定义了两个图类，分别采用邻接矩阵和邻接表作为内部表示。两个类提供同样的接口，这就使后面几节中讨论的算法都能适用于这两个类的对象。

从本章后一部分对各种算法的分析中可以看到，基础数据结构的实现方式有可能影响算法的效率。许多算法需要遍历图中所有的边，其时间复杂度以图的边数作为一个基本度量。从理论上说，边数 $|E|$ 可能达到 $|V|^2$ 的数量级，但在实际应用中的图通常都很稀疏。因此，在处理实际问题时，采用邻接表技术（与邻接矩阵相比），许多算法的实际表现可能好得多。

为了处理和利用图这种复杂结构，人们研究了图上的许多计算问题，开发出许多有趣的算法。本章中介绍了一些基本计算问题和重要算法，主要有：

- 图的宽度优先和深度优先遍历算法。
- 生成树和带权图的最小生成树问题和算法。
- 带权图上的单源点和任意顶点之间的最短路径算法。
- 活动网络上的拓扑排序及关键路径算法。

本章的重点是掌握图的概念、性质和存储表示，掌握这里讨论的重要计算问题、重要算法的基本思想和工作过程，以及实现中的一些有趣技术。

练习

一般练习

1. 复习下面概念：图，二元关系，拓扑结构，图论，图算法，顶点和边，有向图和无向图，有向边及其始点和终点，无向边，邻接（顶）点，邻接边，邻接关系，完全图，顶点的度，入度和出度，路径，路径的长度，回路（环），简单回路，简单路径，有根图，连通，连通无向图（连通图），强连通有向图，最小连通图，最小有根图，无向树，有向树，子图，（无向图的）连通子图，（有向图的）强连通子图，极大连通子图（连通分量），极大强连通子图（强连通分量），带权图，网络，邻接矩阵，顶点表，邻接表表示法，图的遍历（周游），可达顶点，深度优先遍历和宽度优先遍历，深度优先搜索（DFS）序列，宽度优先搜索（BFS）序列，生成树，DFS 生成树，BFS 生成树，（网络的）最小生成树，Kruskal 算法，连通分量的代表元，Prim 算法，MST 性质，最短路径问题，带权图上的路径长度，Dijkstra 算法，Floyd 算法，AOV 网，拓扑序列和拓扑排序，制约关系，AOE 网（一类带权有向图），关键路径。
2. 设有向图 $G=(V, E)$ ，其中 $V=\{a, b, c, d, e, f, g\}$ ， $E=\{\langle a, f \rangle, \langle a, c \rangle, \langle c, d \rangle, \langle b, e \rangle, \langle f, b \rangle, \langle b, g \rangle, \langle e, b \rangle, \langle d, f \rangle, \langle d, e \rangle, \langle c, f \rangle, \langle f, g \rangle\}$ 。
 - a) 画出这个有向图。
 - b) 给出其邻接矩阵表示。
 - c) 给出其链接表表示。
 - d) 判断这个图是否强连通。如果强连通，请给出一个经过图中所有顶点的环路；如果不是强连通的，请给出其中的各个强连通分量。
3. 如果有向图用链接矩阵表示，如何回答下面问题？
 - a) 图中共有多少条边？
 - b) 从一个顶点到另一顶点是否有边？

- c) 一个顶点的出度?
d) 一个顶点的入度?
4. 假设一个图包含 n 个顶点, 如果是无向图最多有几条边? 有向图呢?
5. 一个包含 n 个顶点的连通(无向)图最少有几条边? 一个包含 n 个顶点的强连通(有向)图最少包含几条边?
6. 设 n 个顶点的无向图中的边恰好形成一个环路, 该图有多少棵不同的生成树?
7. 图 7.16 的有向图包含几个强连通分量?
8. 请求出图 7.16 有向图里从结点 v_0 到 v_9 的所有简单路径。
9. 在 Kruskal 算法中需要不断选择最短边。7.4.2 节开始提出了几种可能做法, 请分析它们各自的优缺点, 计算其复杂度。
10. 图 7.17 给出了一个带权无向图, 请针对它完成下面工作:
- 做出其邻接矩阵;
 - 做出其邻接表表示;
 - 将其看作简单的无向图, 用深度优先搜索方法求出其 DFS 序列和 DFS 生成树;
 - 用宽度优先搜索方法求出其 BFS 序列和 BFS 生成树;
 - 用 Kruskal 算法求出其最小生成树(不一定唯一);
 - 用 Prim 算法求出其最小生成树。

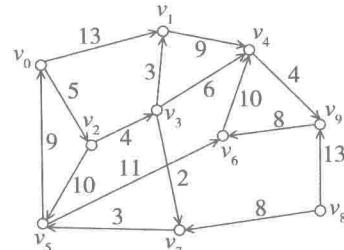


图 7.16

11. 图 7.16 给出的是一个带权有向图。请用 Dijkstra 算法求出图中从 v_0 出发到其他顶点的最短路径。
12. 请求出图 7.18 中从顶点 v_0 出发的两个不同的拓扑排序序列。

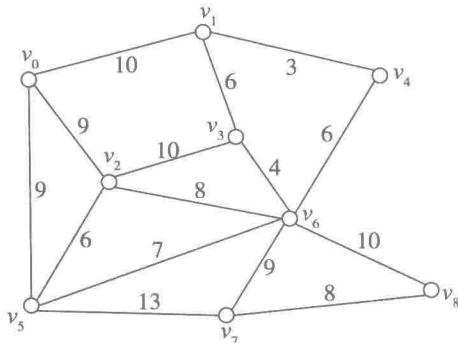


图 7.17

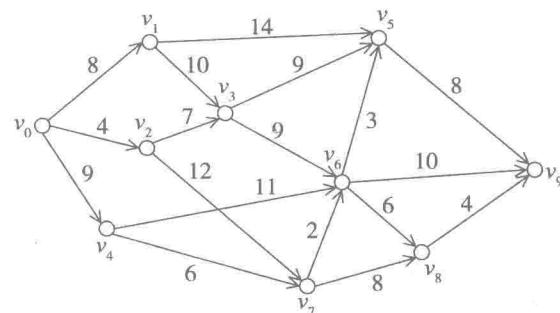


图 7.18

13. 图 7.18 给出了一个带权有向图, 请求出图中从 v_0 到 v_9 的所有关键活动和关键路径。
14. 给定(有向或无向)图 G 和图中两个顶点 u 和 v , 要求确定是否存在从 u 到 v 的路径, 请设计解决这个问题的算法。
15. 请设计一个算法, 检查给定的有向图 G 中是否存在回路, 并在 G 存在回路的情况下给出一条回路, 要求算法的复杂性为 $O(n^2)$ 。
16. 请设计一个算法, 求出不带权的无向连通图中距离顶点 v_0 的最短路径长度(即路径上的边数)为 L 的所有顶点, 要求尽可能高效。
17. 无环的连通无向图 $T=(V, E)$ 的直径是图中所有顶点对之间最短路径长度的最大值, 即, T

的直径 $\text{dia}(T) = \max \{ \text{dis}(u, v) \mid u, v \in V \}$ ，其中 $\text{dis}(u, v)$ 表示顶点 u 与顶点 v 之间最短路径的长度（即最短路径中所包含的边数）。请设计一个算法求无环的连通无向图的直径，并分析算法的时间复杂度。

编程练习

1. 请按 7.2 节开始处的建议，基于 Python 字典定义一种图类型，实现图抽象数据类型中给出的所有基本操作。请分析各种操作的时间和空间复杂度，并比较这种实现技术与 7.2 节给出的图的邻接矩阵实现和邻接表实现。在分析和比较中，请特别关注本章考虑的各种算法的实现效率。
2. 在 7.2.1 节里提出了记录已构造的出边表的想法。请仔细研究这个问题，提出一套可行的实现方法，并完成实现这种功能。
3. 为 7.2.1 节定义的图类增加其他有用操作，例如前面习题提到的确定顶点间路径的操作，检查图中是否存在回路，如果有就给出回路的操作等。
4. 修改 7.2.2 节的基于邻接表技术的图类，用自定义的链表实现邻接表。
5. 7.2 节最后讨论了图中顶点可能关联一些信息，建议给图对象扩充一个顶点表（或顶点字典）。请修改某个图类的定义，实现这种想法。在做这项工作时，请给图对象增加一些使用顶点的有用操作。
6. 请参考 7.3.1 节给出的深度优先图遍历算法，设计并实现其他（递归的或非递归的）深度优先和宽度优先遍历算法。
7. 请实现遍历图结点和图中边的迭代器。有关的迭代器应返回什么值？
8. 请实现构造 BFS 生成树的算法，以及构造 DFS 生成树的非递归算法。
9. 7.4.2 节提出了几种选择最小元素的方法，给出的算法中采用了其中一种。请修改算法的实现，采用另一种不同的选取方法。
10. 实现 7.4.4 节所需的可减权堆 DecPrioHeap，支持所需操作，并论证所做实现具有文中提出的操作复杂性。
11. 请基于可减权堆重新实现 Dijkstra 算法，保证算法在不付出更大时间代价的情况下，空间复杂度变成 $O(|V|)$ 。
12. 在 toposort 函数里另用一个表 toposeq 记录找到的拓扑序列。实际上这个表不必要，完全可以在 indegree 表里记录拓扑序列。请考虑这种可能性，考虑是否需要约定从结果表中得到拓扑序列的方法。考虑如何修改原函数实现这种可能性。需要付出多少时间代价？算法的时间复杂度会改变吗？
13. 二部图（bipartite graph） $G=(V, E)$ 是一类无向图，其顶点集 V 可以划分为两个不相交子集 V_1 和 $V_2=V-V_1$ ，使 V_1 中的任意两顶点在图 G 中不相邻，且 V_2 中任意两顶点在 G 中不相邻。换句话说，图中任一条边的两端均分属这两个顶点集。
 - a) 请举出一个结点个数为 7 的二部图和一个结点数为 7 的非二部图。
 - b) 请用定义函数 bigraph 判断无向图 G 是否是二部图，并分析程序的时间复杂度。如有必要可使用栈或队列数据结构。

第8章 字典和集合

本章的内容与前几章有些不同，这里不准备介绍一种新的数据结构，而是要讨论计算中最重要的一类问题的许多不同解决方式，以及与之相关的问题和性质。

8.1 数据存储、检索和字典

计算机的基本功能是存储和处理数据，首先是存储数据，处理也需要先找到被处理的数据，或称为访问数据。所以，数据的存储和访问是计算机最基本的功能。

8.1.1 数据存储和检索

第1章介绍了计算机存储的基本概念和一些情况。数据访问的基本方式是基于存储位置，如果知道所需数据保存在哪里，只需要常量时间就可以得到它。使用变量，基于下标提取表中元素等，都是常量时间访问数据的具体例子。但是，在许多情况下，计算过程（程序）并不确知数据的存储位置，但却需要使用某些数据。真实世界里也经常遇到这种情况，查字典就是一个典型实例：人们知道字典里有某个字的解释，但并不确知这个字位于字典里的哪一页。在这种情况下，第一个任务就是找出这个字所在的页码。对于计算，就是要找到数据的存储位置。这个操作称为检索。

数据的存储和检索是计算中最重要、最基本的一类工作，也是各种计算机应用和信息处理的基础。数据需要存储和使用，因此经常需要检索。本章研究的问题就是数据的存储和检索（或称查询）技术，下面是几个实际例子：

- 各种计算机化的电子字典，其基本功能就是基于算法的各种数据检索。
- 图书馆编目目录和检索系统，支持读者以各种方式检索书籍资料的有关信息。
- 规模巨大的有机物数据库，需要基于有机物结构或者光谱等参数进行检索。
- 完成多元多项式乘法的算法，在求出了多项式乘积的一个因子后需要合并同类项，为此就需要在已经得到的项表里检索。
- 万维网搜索（实际上是检索的英文原词 searching 的另一种翻译）系统，如 Google 或百度等。这里有两个层面的问题：网络搜索系统通过广泛检查网络上的各种信息建立起一个巨大的内部数据库，这是网络搜索公司内部的搜索和信息积累。遇到用户请求时，网络服务系统做一次数据库检索，将检索结果送给用户。

概述

数据检索牵涉到两个方面，一方面是已存储的数据集合，另一方面是用户检索时提供的信息。具体检索可以是确定特定数据是否存在于数据集中，相当于集合成员判断；也可以是希望找到与所提供信息相关的数据，类似于在字典里查词语的解释。在后一种方式中，检索时提供的信息被看作检索码或关键码（key）。这种关键码也常作为数据的一部分，存储在数据集里，这就是基于关键码的数据存储和检索，是本章讨论的主题。

作为检索基础的关键码，通常是数据项的某种（可能具有唯一性的）特征，可以是数据内容的一个组成部分，也可以是专门为数据检索建立的标签。后一种情况实际上也很常见。以学

校的学生记录为例，原本只需记录与学生有关的信息。而为每个学生指定一个学号，只是为了检索方便。这一类关键码通常是数据的唯一标识。

下面的讨论将不考虑关键码究竟是什么，那是实际应用考虑的问题。这里的基本假设是抽象的：需要存储的数据元素由两个部分组成，其中一部分是与检索有关的关键码，另一部分是与之关联的数据。下面讨论将始终在这一假设下进行。

字典就是支持基于关键码的数据存储与检索的数据结构。在一些专业书籍或编程语言里，字典也被称为查找表、映射或者关联表等。字典实现数据的存储和检索，而需要存储和检索的信息和环境有许多具体情况，因此可能要求不同的实现技术。

字典的实现中可以用到前面讨论过的许多想法和结构，包括各种线性结构、树形结构，或者它们的各种组合，还会涉及在这些结构上操作的许多算法。下面将讨论基于顺序表、二叉树和其他树形结构等的字典实现。进一步说，基于一种基础结构，也可能有不同的数据组织方式。下面将会看到顺序表、二叉树等的不同使用方式。

数据存储和访问是计算中最基本的操作，对系统的整体效率影响很大。另一方面，复杂系统里经常需要存储大量数据。由于这些原因，人们特别关注字典的实现效率，包括存储空间的利用率和操作效率。

在字典的使用中，最重要也是使用最频繁的操作就是检索（search，也称查找）。在考虑字典的实现时，检索效率是最重要的考虑因素。当然，由于数据规模不同，检索效率的重要性也可能不同。在开发数据密集型的应用时，人们会更关注效率问题。

基于关键码的检索是最简单、最基本的检索问题，其中的基本操作就是比较两个关键码是否相同。更一般的问题是需要根据某些线索，从巨大的数据集中找出相关数据，其中可能需要做更复杂的匹配，或者带有“模糊”性质的匹配，或者基于复杂数据内容的匹配。今天人们每天都在使用的网络检索系统，也可以看作字典概念的发展。

字典操作和效率

实际中使用的字典可以分为两类：

- 静态字典：在建立之后，这种字典内容和结构都不再变化，主要操作只有检索。对这种字典，需要考虑创建的代价，但最重要的是检索效率。无论如何，创建工作只需要做一次，而检索是在字典的整个生命周期中反复进行的操作。
- 动态字典：在初始创建后，这种字典的内容（和结构）将一直处于动态变动之中。对这种字典，除了检索之外，最重要的基本操作还包括数据项的插入和删除等。在考虑这种字典的实现时，不仅需要考虑检索的效率，还必须考虑插入和删除操作的效率，需要在更多相互影响的因素中做权衡。

对于动态字典，还有一个问题也必须重视：这里的插入和删除操作可能导致字典结构的变化。如果需要支持长期使用，在设计时就需要考虑字典的实现方式能否在长期的动态变化中保持良好结构，能否始终保证良好的检索（和其他操作）效率？也就是说，良好设计的字典应该保证其性能不会随着操作的进行而逐渐恶化。

在字典中检索，最终将得到一个结果，或是检索成功而且得到了所需的数据；或是确认了要找的数据不存在，此时可能返回某种特殊标志。有关检索效率的评价标准，通常考虑的是在一次完整检索过程中比较关键码的平均次数，通常称为平均检索长度（Average Search Length，ASL），其定义是（其中 n 为字典中的数据项数）：

$$\text{ASL} = \sum_{i=0}^{n-1} p_i \cdot c_i$$

其中 c_i 和 p_i 分别为第 i 项数据元素的检索长度和检索概率。如果字典中各元素的检索概率相等，也就是说，如果 $p_i = 1/n$ ，那么 $\text{ASL} = \frac{1}{n} \sum c_i$ 。上述定义中只考虑了被检索关键码在字典中存在的情况（正确的检索算法保证存在的关键码一定能找到），在很多情况下，还需要考虑字典中不存在被检索关键码的情况。

字典和索引

实际上，字典是两种功能的统一：

- 作为一种数据存储结构，支持在字典里存储一批数据项。
- 提供支持数据检索的功能，设法维护从关键码找到相关数据的联系信息。

后一功能也称为索引，其存在的目的就是为检索服务。

做基于关键码的检索，就是要实现从关键码到数据存储位置的映射，而这种映射也就是索引。有时人们也专门研究索引结构的问题。但是，索引结构本身并不存储数据，只提供（基于关键码的）检索功能，因此它只能作为字典的附属结构，不可能独立存在。一个具体的索引结构可能提供与它所关联的基本字典不同的检索方式，如另外提供了一套关键码等。具体到一个字典，它可以没有附属的索引，只有自身提供的检索方式；也可以附有一个或者多个索引，支持多种不同方式的检索。在真实世界里也有这样的情况。例如，考虑现实中的《新华字典》，它的基本部分存储着大量词条，基于拼音排序，可以直接检索。除此之外还提供了若干种专门索引，如部首检字表、难字检字表等。

本章后面部分讨论的各种技术和结构，只要与检索有关，都既可以用于实现字典，也可以用于实现索引。在用于实现字典时，关键码关联于实际数据，数据保存在本字典的内部。如果是用于实现索引，它就只提供了从关键码到相应的数据项存储位置的映射，而实际数据存储在与这个索引相关的字典里（的具体存储位置）。

8.1.2 字典实现的问题

下面将研究一些实现字典的技术。在具体研究前先讨论几个一般性问题。

字典抽象数据类型

在开始讨论字典的实现及其问题之前，先定义一个抽象数据类型如下：

ADT Dict:	# 字典抽象数据类型
Dict(self)	# 字典构造函数，创建一个新字典
is_empty(self)	# 判断self是否为一个空字典
num(self)	# 获知字典中的元素个数
search(self, key)	# 检索字典里key的关联数据
insert(self, key, value)	# 将关联(key, value)加入字典
delete(self, key)	# 删除字典中关键码为key的元素
values(self)	# 支持以迭代方式取得字典里保存的各项关联中的value
entries(self)	# 支持以迭代方式获得字典的关联中的逐对key和value二元组

字典的基本操作就是检索，对于动态字典，还需支持插入和删除元素。此外，在每个时刻，字典里保存着一些元素（数据项），也可能为空。上面抽象数据类型提供了相应的访问操作。操作 values 以迭代器方式返回字典里的逐项值，最后的 entries 将返回一个迭代器，以便

程序中逐对使用字典里的关联。

注意，各种字典都不应该允许修改字典关联中的关键码，因为关键码用于确定其所在项在字典里存储的位置，以支持高效检索。如果允许修改关键码，就可能破坏字典数据结构的完整性，导致后续检索操作失败。

字典元素：关联

如前所述，在支持基于关键码的存储与检索的字典里，一个数据项可以划分为两个部分：其一是与检索有关的关键码部分，另外就是与检索无关的其他数据部分。显然，数据项的插入 / 删除也与关键码关系密切，因为插入操作的结果将会在检索中使用，而删除具体数据项的操作中需要有一种指定具体项的方法，通常也采用指定关键码的方式。由于这些原因，下面将始终把字典里的数据项简单地分为两部分，一部分是与检索有关的关键码，另一部分称为值，与检索和其他操作（例如插入 / 删除）的实现方式无关，但在实际应用中却可能非常重要。这样，一个数据项就是一种二元组，下面称之为关联。

为了下面讨论的方便，现在首先定义一个关联对象的类 `Assoc`，假定本章下面讨论的字典都以 `Assoc` 对象为元素：

```
class Assoc:
    def __init__(self, key, value):
        self.key = key
        self.value = value
    def __lt__(self, other):      # 有时（有些操作）可能需要考虑序
        return self.key < other.key
    def __le__(self, other):
        return self.key < other.key or self.key == other.key
    def __str__(self):           # 定义字符串表示形式便于输出和交互
        return "Assoc({0},{1})".format(self.key, self.value)
```

如果变量 `x` 的值是一个关联对象，表达式 `x.key` 将取得其关键码，而 `x.value` 取得其中的关联值。

除构造函数外，`Assoc` 类里还定义了两个顺序比较操作。前面说过，Python 解释器遇到比较运算符“`<`”，就会去找类里定义的 `__lt__` 方法（表示 `less than`）；类似的，“`<=`”运算符关联于类中定义的 `__le__` 方法（`less than or equal to`）。

在这里定义“`<`”和“`<=`”，是因为有时可能需要比较两个数据项，例如在使用 `sorted` 等标准函数时，或者在字典的实现中需要比较数据项的大小等。如果需要，也可以定义其他关系运算符（如大于、大于等于等）。

字典的实现

从最基本的存储需求看，字典就是以关联为元素的汇集，前面讨论过的各种数据汇集结构都可用作字典的实现基础。例如线性表，是元素的线性顺序汇集。如果以关联作为元素，就可以看作字典了。下面将首先考虑这种实现。

字典实现还有一些细节。例如，字典插入有一个特殊情况：要求插入的关键码已经在字典里存在了。在实际应用中遇到了这种情况，应该根据实际需要去处理。常见处理方式包括修改已有关键码的关联项的值部分，或者插入一个新数据项（如果所用的字典允许出现关键码重复的多个数据项），或者报错等。与之对应，执行删除操作时有可能找不到相应的项，也可能实际存在多个关键码相同的项，同样应该根据实际需要决定如何处理这些情况。以存在多个匹配项的情况为例，在这里需要决定是删除一个具有这种关键码的项，还是删除所有的匹配项。

下面的讨论中采用了一种简单的方式：如果插入时遇到关键码相同的项，就简单修改其关联值；在删除时没找到要删的元素就什么也不做。

在考虑字典的实现时，最重要的问题是字典操作的实现效率。由于字典可能规模较大，需要频繁执行查询等操作，操作的效率非常重要。本章下面各节将讨论一系列字典实现技术。首先考虑线性表，主要是连续表；而后是另一种很特别的技术：散列表，它也基于连续表存储数据项，但采用一套特殊的索引方式。在此之后是一些基于树结构的字典实现技术，及其一些与之相关的问题。

8.2 字典线性表实现

线性表里可以存储信息，因此可以作为字典的实现基础。本节讨论这个问题。

8.2.1 基本实现

将关键码和值的关联作为元素（数据项）顺序存入线性表，形成关联的序列，可以作为字典的一种实现技术。检索就是在线性表里查找具有特定关键码的数据项，数据项的插入和删除等都是普通的线性表操作。

在 Python 语言里，这种顺序字典可以用 `list` 实现。其中的关联可以用前面定义的 `Assoc` 类对象，也可以用二元的 `tuple` 或 `list` 实现。

由于没有其他信息，检索时只能用要找的关键码在表中（顺序）查找。遇到关键码 `key` 相同的字典项就是检索成功，返回与关键码关联的 `value`；检查完表中所有的项但未遇到要找的关键码，就是检索失败。

由于没有任何限制，插入新关联可以简单地用 `append` 实现；删除可以在定位后用 `list` 的 `pop` 操作实现（还可以考虑基于要删除项的内容的删除等）。

可以考虑以 `list` 作为内部表示，自己定义一个字典类，该类的对象就是字典，字典操作实现为类里的对象方法。定义的框架如下（具体定义留作练习）：

```
class DictList:
    def __init__(self):
        self._elems = []

    def is_empty(self):
        return not self._elems

    .....
    .....
# end of the class
```

如果插入操作的实现把元素直接放在已有元素之后（放在最后，不检查重复），这就是一个 $O(1)$ 时间复杂度的操作。删除元素时需要首先检索，确定了元素的位置后删除（是简单的表中删除元素），时间复杂度为 $O(n)$ 。这里的 n 是表的长度。

最重要的操作是检索（实际上，删除操作也需要先做检索），现在分析其复杂性，考虑检索中的比较次数：

$$\text{ASL} = 1 \times p_0 + 2 \times p_1 + \dots + n \times p_{n-1} = \frac{1}{n}(1 + 2 + \dots + n)$$

上式的第二步假定每个关键码的检索概率相同，都为 $\frac{1}{n}$ 。最后

$$\text{ASL} = \frac{n+1}{2} = O(n)$$

在上述分析中，只考虑了对字典中已有关键码的检索情况。如果被检索关键码不属于字典的关键码集，经过 n 次比较后检索将以失败结束，复杂度也是 $O(n)$ 。综合两种情况，整体的平均值仍为 $O(n)$ 。

基于线性表的字典实现，优点和缺点都非常明显：

- 数据结构和算法都很简单，检索、删除等操作中只需要比较关键码相同（或不同），适用于任意关键码类型（例如，并不要求关键码集合存在某种顺序关系）。
- 平均检索效率低（线性时间），表长度 n 较大时，检索很耗时。
- 删除操作的效率也比较低，因此不太适合频繁变动的字典。

另外，在字典的动态变化中，这种字典的各种操作的效率不变。但这并不是什么优点，因为它们都已经是效率最低的操作了。

8.2.2 有序线性表和二分法检索

要想提高操作效率，就需要把字典里的数据项组织好，使之具有可利用的结构，从而能更好地支持检索。具有合理内部结构的字典上有可能实现高效检索。

如果关键码取自一个有序集合（存在某种内在的序，例如整数的小于等于关系，字符串的字典序等），就可以按关键码大小的顺序排列字典里的项（从小到大或从大到小）。在数据项按序排列时，可以采用二分法实现快速检索。

二分法检索是一种重要的检索技术，其基本思想是按比例逐步缩小需要考虑的数据范围，从而快速逼近需要找的数据。采用二分法检索关键码排序的顺序表字典，基本操作过程如下（设字典里的数据项按关键码的升序排列）：

- 1) 初始时，考虑的元素区间是整个字典（一个顺序表）。
- 2) 取所考虑的元素范围里位置居中的那个数据项，比较该项的关键码与检索关键码，如果它们相等则检索结束。
- 3) 如果检索关键码较大，则把检索范围修改为中间项之后的半区间；如果检索关键码较小，就把检索范围修改为中间项之前的半区间。
- 4) 如果在关注范围里仍有数据就回到步骤 2 继续；否则检索失败结束。

在元素有序的表上做二分法检索的函数可定义如下：

```
def bisearch(lst, key):
    low, high = 0, len(lst)-1
    while low <= high:          # 范围内还有元素
        mid = low + (high - low)//2
        if key == lst[mid].key:
            return lst[mid].value
        if key < lst[mid].key:
            high = mid - 1       # 在低半区继续
        else:
            low = mid + 1        # 在高半区继续
```

可以继承前面基于表的字典类，定义一个新的字典类：

```
class DictOrdList(DictList):
    .....
    def search(self, key):
    .....
    def insert(self, key, data):
```

```

def delete(self, key):
    .....
    .....
    # end of class

```

这里的 search、insert 和 delete 方法都必须重新定义。在插入或删除表中元素时，insert 和 delete 都必须保持字典中的元素有序，search 则需要利用元素有序的性质，采用二分法实现。具体实现留给读者完成。

二分法检索实例

【例 8.1】 考虑二分法检索的情况，具体实例是一个包含 11 个整数的表：

位置	0	1	2	3	4	5	6	7	8	9	10
关键码	5	13	19	21	37	56	64	75	80	88	92

采用二分法检索，找到表中居于位置 5 的元素（整数 56）只需要做 1 次比较，找到位于位置 2 和 8 的数据需要做 2 次比较，找到位置 0、3、6、9 的元素需要做 3 次比较，找到另外 4 个位置的元素需要做 4 次比较。

图 8.1 中的二叉树形象地表示了对这个表中所有元素的检索过程，这样的树称为二分法检索过程的判定树。树中结点所标的数是数据项的关键码。在检索过程中，用检索关键码与这个关键码比较，如果检索关键码较小，就转到相应左子结点继续，较大时转到相应右子结点。检索过程沿着从根结点到某个目标结点的路径前进，在每个结点处（对应于表中一个位置）做了一次比较。在树根找到结果，只需要做 1 次比较。一般而言，通过检索找到某个结点的比较次数等于该结点的层数加 1。

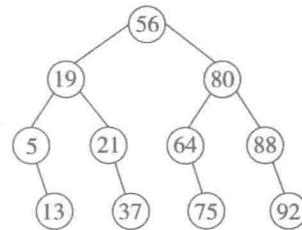


图 8.1 二分法检索的判定树实例

算法分析

在向表中插入数据时，必须维持关键码有序。为此需要找到新数据项的正确插入位置，通过逐项后移数据的方式腾出空位后插入新数据项。如果采用顺序检索的方式，找位置就需要线性时间。无论怎样具体实现这一过程，插入数据项的 insert 都会变成 $O(n)$ 操作。在删除操作中也需要移动表中的数据项，保证剩下数据的正确顺序，所以删除也是 $O(n)$ 时间操作。由于表中元素有序，实现插入操作时也可以考虑用二分法检索位置；在删除操作中也可以先用二分法检索元素。但由于随后的实际元素插入或删除都需要为保序而移动表中元素，所以线性时间是无法避免的。

通过观察可知，在元素排序的顺序表里，在一次成功检索的过程中，所做比较次数不超过树的高度加一。因此， n 个结点的二分判定树的高度不超过 $\lfloor \log_2 n \rfloor$ 。也就是说，采用二分法检索，成功时的比较次数不超过 $\lfloor \log_2 n \rfloor + 1$ 。

现在考虑所有可能的检索，包括所有的成功检索和失败检索。针对例 8.1 中的排序表字典图，图 8.2 给出了对应的完整的二叉判定树，其中描述了这个实例的所有检索成功和不成功情况的判定路径。不难看出，这棵树正好是图 8.1 中二叉判定树的扩充二叉树（参考 6.1 节）。图中的小矩形框表示各种检索不成功的情况，它们是扩充二叉树的外部结点。例如，标着 (13, 19)

的矩形框表示被检索的关键码值位于区间 (13, 19) 里, 不包含两端的值。以这个区间的值作为关键码检索, 检索失败就到达了这个方框。根据这个图可以看到, 在检索不成功时, 最大比较次数也是 $\lfloor \log_2 n \rfloor + 1$ (也是 $O(\log_2 n)$)。

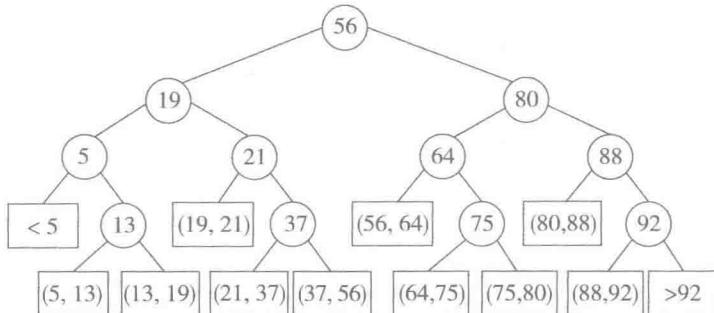


图 8.2 二分法检索的完整判定树实例, 包括所有成功和失败的检索

通过数学推导也可以得到同样结论。假设表中有 $n=2^h-1$ 个元素, 相应二叉判定树的扩充二叉树的高度为 h 。在检索中每做一次比较, 需要考虑的范围缩小一半, 通过 k 次比较可能检查到的元素数如下表所示:

比较次数	可能涉及的元素个数
1	$1=2^1-1$
2	$3=2^2-1$
3	$7=2^3-1$
.....
k	2^k-1

通过 h 次检索就能检查到表中的任一元素。对一般的 n , 如果 $2^k-1 < n \leq 2^{k+1}-1$, 检查到任何元素的最大检索长度为 $k+1$ 。所以, 一般而言, 对包含 n 个元素的字典做二分法检索, 通过下面计算可以得到其最大检索长度:

$$2^k < n + 1 \leq 2^{k+1}, \text{ 也就是说, } 2^k \leq n < 2^{k+1}$$

$$\text{由此可得 } k \leq \log_2 n < k + 1$$

所以, 比较次数不超过 $\lfloor \log_2 n \rfloor + 1$ 。

总结一下, 采用有序的顺序表和二分法检索 (结合其他情况):

- 主要优点是检索速度快, $O(\log n)$ 。
- 插入和删除时需要维护数据项的顺序, 因此都是 $O(n)$ 操作 (虽然检索插入 / 删除位置可以用二分法, 为 $O(\log n)$ 时间, 但完成操作需要移动元素, 为 $O(n)$ 时间)。
- 二分法技术只能用于关键码可以排序、数据项按关键码排序的字典, 而且只适用于顺序存储结构, 需要连续的存储块, 不适合实现很大的动态字典。

8.2.3 字典线性表总结

虽然也可以考虑基于单链表或双链表技术实现字典, 但通过分析可知:

- 如果字典里的数据项任意排列, 插入时可以简单地在表头插入, 是 $O(1)$ 操作; 检索和删除都需要顺序扫描整个表, 是 $O(n)$ 操作。

- 如果表中的数据项按关键码升序或者降序排列，插入需要检索正确位置，为 $O(n)$ 操作；检索和删除同样需要顺序扫描检查，平均检查半个表，为 $O(n)$ 操作。

由上述分析可知：采用链接表实现字典没有任何明显的优势，而且无法利用关键码排序的价值。由此，实际中人们很少采用链接表技术实现字典。另一方面，采用链接表，具体的实现技术很简单，不需要更进一步讨论。

问题和思考

采用线性表技术实现字典，只能适合一部分简单需求，如字典的规模比较小，而且不常出现动态操作的情况。但在实际中，各种应用经常需要存储和检索很大的数据集合，字典中的数据内容不断动态变化的情况也很常见。采用简单的顺序表或链接表实现字典，顺序检索的效率太低，在效率上常常不能满足实际应用的需要。采用排序的顺序表和二分检索能大大提高检索速度，但仍有两大问题：

- 不能很好地支持数据的变化（数据插入和删除的效率低）。
- 必须采用连续方式存储整个数据集合。如果数据集很大（实际中经常需要存储很大的数据集，可能包含成百兆或几千兆 (10^9) 的数据项），连续存储方式就很难接受了。

如果要支持存储很大的、经常变动的数据集合，而且希望高效检索，必须考虑其他组织方式。人们为此开发了另外一些结构，主要分为两类：

- 基于散列 (hash) 思想的散列表（也常被音译为哈希表）。
- 基于各种树形结构的数据存储和检索技术（利用树结构的特性，即在不大的深度范围内可以容纳巨大数量的结点）。

下面几节讨论有关情况。

8.3 散列和散列表

在讨论各种树形结构的应用可能性之前，首先讨论在计算机科学技术领域使用非常广泛的散列技术及其在字典方面的应用，即所谓的散列表 (hash table)。

8.3.1 散列的思想和应用

从字典的需要出发，研究基于关键码的检索，可以提出下面的问题：

什么情况下基于关键码能最快找到所需的数据？

根据计算机中数据存储和使用的方式，可以得到下面回答：

如果数据项连续存储，而关键码就是存储数据的地址（或下标）！

显然，在这种情况下，只需 $O(1)$ 时间就能得到所需要的数据了。

但是，一般而言，字典所用的关键码可能不是整数，由此不能作为下标。另一方面，即使关键码是整数，也可能因为取值范围太大而不适合作为下标。例如，由于身份证号码的唯一性，最适合作为国家的身份信息系统的关键码。但中国的身份证号码是一个 18 位整数，取值范围达到 10^{18} 。如果用它作为数据存储的下标（地址），相关的（顺序）表就需要有 10^{18} 个元素位置，而实际中国人口大约为 10^9 量级。也就是说，这个表的实际填充率约为 $1/10^9$ ，这种存储方式显然是非常不合适的。

但参考上面的观点，从这里继续思考下去，很容易理解散列表的基本思想：如果一种关键码不能或者不适合作为数据存储的下标，可以考虑通过一个变换（一个计算）把它们映射到一种下标。这样做，就把基于关键码的检索转变为基于整数下标的直接元素访问，有可能得到一

一种高效的字典实现技术。

以散列表的思想实现字典，具体方法是：

①选定一个整数的下标范围（通常以0或1开始），建立一个包括相应元素位置范围的顺序表。

②选定一个从实际关键码集合到上述下标范围的适当映射 h ：

- 在需要存入关键码为 key 的数据时，将其存入表中第 $h(key)$ 个位置。
- 遇到以 key 为关键码检索数据时，直接去找表中第 $h(key)$ 个位置的元素。

这个 h 称为散列函数，也常被称为哈希（hash）函数或杂凑函数，它就是从可能的关键码集合到一个整数区间（下标区间）的映射。

散列思想在信息领域的应用

上面从信息存储和检索的需要介绍了散列表的概念。实际上，散列的思想是在信息和计算领域中逐渐发展起来的一种极其重要的思想，其应用远远超出了数据存储和检索的范围。所谓散列，一般而言，就是以某种精心设计的方式，从一段可能很长的数据生成一段很短（经常为固定长度）的信息串，例如简单的整数或字符串，最终都是二进制串，而后利用这种二进制串做某种有用的事情。散列技术在计算机和信息技术领域非常有价值，应用极广，可能用在各种数据处理、存储、检索工作中。例如：

- 文件的完整性检查：定义一个散列函数，从一个软件的所有文件内容算出一个数或者一个字符串，需要时检查实际文件的散列值是否与其匹配。在实际安装软件时，经常看到提示说“正在检查文件完整性”，所做的就是基于散列技术检查软件的安装文件，看看它的散列值是否与系统提供的散列值相同。这显然是一种概率性的检查，但出错的概率极低。无论是传输或拷贝中无意造成的文件破坏，或者有意修改安装文件（如黑客或计算机病毒），要得到同样散列值的概率微乎其微，可以完全忽略。
- 互联网技术中到处都使用和依靠散列函数，用于网页传输中的各种安全性和正确性检查，包括各种网络认证和检查、各种网络协议。
- 计算机安全领域中大量使用散列技术，例如在各种安全协议里。

将散列技术应用于数据的存储和检索，就得到了散列表。基本做法如前所述。要想有效地实现一个散列表，不但需要选择合适的关键码映射（散列函数），还要考虑由于采用映射方式确定存储和检索位置而带来的各种问题。

散列技术：设计和性质

如果某个字典的可能关键码集合非常小，例如关键码只可能是0~19这组整数，直接用它们作为数据存储下标就是最好的选择。再如ASCII字符集的编码只有0~127，相应字形库就应该用这组整数作为存储的下标。

一般情况都不是这样。可能的关键码集合通常都非常大。例如，假设关键码是10个字母以内的英文字字符串，不难算出这个关键码集合的规模达到约 1.4×10^{14} ，不可能直接作为（或对应到）字典的下标。针对一种关键码实现字典时，所用下标集合通常都远远小于关键码集合的规模。也就是说，在实现散列表时，通常都有：

$$|\text{KEY}| \gg |\text{INDEX}|$$

其中 $|\text{KEY}|$ 和 $|\text{INDEX}|$ 分别表示关键码集合和下标集合的规模。

这说明，在通常情况下，散列函数 h 是从一个大集合到一个小集合的映射。在这种情况下，它显然不可能是单射，必然会出现多个不同的关键码被 h 对应到同一个下标位置的情况，

也就是说，必然会出现 $key_1 \neq key_2$ 但 $h(key_1) = h(key_2)$ 的情况。如果实际出现了这种情况，人们就说在这里出现了冲突（或者碰撞），此时也称 key_1 和 key_2 为 h 下的同义词。根据上面分析，冲突是散列表使用中必然会出现的情况，因此，在考虑散列表的实现时，必须考虑如何解决冲突的问题。

对于一个规模固定的散列表，一般而言，表中的元素越多，出现冲突的可能性也就越大。这里的一个重要概念是负载因子。这是一种很有用的度量值，是考察散列表在运行中的性质的一个重要参数，其定义是：

$$\text{负载因子 } \alpha = \frac{\text{散列表中当时的实际数据项数}}{\text{散列表的基本存储区能容纳的元素个数}}$$

如果数据项直接保存在基本存储区里，那么将总有 $\alpha \leq 1$ 。无论如何，负载因子的大小与冲突出现的可能性密切相关：负载因子越大，出现冲突的可能性也越大。显然，如果扩大散列表的存储空间（增加可能的存储位置），就可以降低其负载因子，减小出现冲突的概率。但负载因子越小，散列表里空闲空间的比例也就越大。因此这里也有得失权衡的问题。

另一方面，无论负载因子的情况怎样，冲突总是可能出现的。特别是在实现供其他人使用的一般散列表时，开发者无法了解实际使用中的情况，在设计时就必须考虑冲突的处理问题。由于散列表的重要性和广泛应用，人们深入研究散列表的设计，提出了一些冲突处理技术。基于这些处理方式，形成了多种不同的散列表实现结构。

总结一下，如果要基于散列技术实现字典，必须解决两个大问题：散列函数的设计，冲突消解机制。下面首先考虑散列函数的设计问题。

8.3.2 散列函数

在设计字典时，首先应该根据实际需要确定数据项集合，选定相应的关键码集合 KEY。为了实现散列表字典，还要确定一个存储位置区间 INDEX，例如，选择从 0 开始的一段下标。KEY 和 INDEX 是两个有限集，分别为将要定义的散列函数的参数域（定义域）和值域，是定义散列函数的基础。从本质上说，任何从 KEY 到 INDEX 的全函数 f 都能满足散列函数的基本要求，因为对任何 $key \in KEY$ ，都有 $f(key) \in INDEX$ ，函数值落在合法下标范围内。但另一方面，散列函数的选择可能影响出现冲突的概率。

有些函数显然不好。例如，把所有关键码都映射到下标值为 0 的函数，虽然能满足定义域和值域的基本要求，但它将使冲突的出现最大化。由此可见，选择（设计）散列函数，最重要的考虑应该是尽可能减少出现冲突的可能性。

在设计散列函数时，有价值的考虑包括：

- 函数应能把关键码映射到值域 INDEX 中尽可能大的部分。显然，如果扩大函数值的范围（在 INDEX 内），出现冲突的可能性就会下降。如果某个下标不是散列函数的可能值，这个位置可能就无法用到。应该尽量避免这种情况。
- 不同关键码的散列值在 INDEX 里均匀分布，有可能减少冲突。当然，实际情况还与真实数据里不同关键码值出现的分布有关。如果不知道关键码的实际分布（例如，开发一个字典库模块时的情况就是这样），就只能考虑均匀分布。
- 函数的计算比较简单。这一要求很显然：使用散列表的本意是希望提高效率，而计算散列函数的开销是各种操作的开销中的一部分。

在这些基本考虑下，人们提出了一些设计散列函数的方法。

用于整数关键码的若干散列方法

有些方法适用于已知实际关键码集合的情况，有的依赖于对实际数据的分析。如果已知需要存储的关键码集合及其分布，有可能设计出最合适的散列函数，能有效缩短整数的取值范围，甚至有可能保证使用中绝不会出现冲突。

下面简单介绍的几种方法都只适用于整数关键码，但也可以结合后面的方法使用，作为实际散列函数计算过程的一部分。

数字分析法：对于给定的关键码集合，分析所有关键码中各位数字的出现频率，从中选出分布情况较好的若干数字作为散列函数的值。

例如，假设要处理的是下面第一列的关键码集合：

key	$h_1(key)$	$h_2(key)$
000125 <u>6</u> 72	62	6
000125 <u>8</u> 73	83	8
000125 <u>7</u> 76	76	7
000125 <u>4</u> 72	42	4
000125 <u>3</u> 79	39	3
000125 <u>6</u> 72	62	6

散列函数 h_1 选择关键码的百位和个位数字拼接，把 9 位十进制的关键码数映射到 2 位十进制数的下标值，只需用一个 100 个元素的表存储字典数据，而且使用中不会出现冲突。另一可能的散列函数 h_2 选取关键码的百位数字，把关键码映射到 0 到 9 的范围内，这样就进一步节约了存储，但其中出现了冲突，需要解决。

显然，只有在关键码集合已知的情况下，才能有效地使用这种方法。然而，最常见的情况是需要存储和使用的数据不能在设计字典之前确定，具体使用的关键码和分布情况事先未知，这种分析方法就不能用了。

折叠法：将较长的关键码切分为几段，通过某种运算将它们合并。例如用加法并舍弃进位的运算，或者用二进制串运算。下面看一个例子。

假设关键码均为 10 位整数，这里考虑将其分为 3 位一段，把得到的 3 位整数相加并去掉进位，以得到的结果作为散列函数的值。例如，对于关键码 1456268793，切分三个三位数和一个一位数，计算： $1+456+268+793=1518$ ，去掉进位后取散列值为 518。这样就把 10 位整数关键码映射到 $[0, 999]$ 区间了。

中平方法：先求出关键码的平方，然后取出中间的几位作为散列值。

例如，现在考虑取关键码平方值的百位到万位。对关键码 1456268793，其平方是 2120718797465676849，从中取得散列值为 768。

从上述实例可以看出，对于整数关键码，散列函数的设计有两方面追求：其一是把较长的关键码映射到较小的区间，另一个就是尽可能消除关键码与映射值之间明显的规律性。通俗地说，散列函数的映射关系越乱越好，越不清晰越好。

常用散列函数

一些方法只适应于某些特殊情况，但也有些方法居于通用性。通用的散列方法只能基于对关键码集合的均匀分布假设，下面介绍两种常用的散列函数：

- 除余法，适用于整数关键码。
- 基数转换法，适用于整数或字符串关键码。

实际中最常用的就是这两种关键码。

除余法：关键码 key 是整数，用 key 除以某个不大于散列表长度 m 的整数 p ，以得到的余数（或者余数加 l ，由下标开始值确定）作为散列地址。

为了存储管理方便，人们经常将 m 取为 2 的某个幂值，此时 p 可以取小于 m 的最大素数（如果连续表的下标从 1 开始，可以用 $key \bmod p+1$ ）。例如，当 m 取 128、256、512、1024 时， p 可以分别取 127、251、503、1023。除余法在实际中使用最为广泛，还常被用于将其他散列函数的结果归入所需区间。

前面说，设计散列函数的一个基本想法是使得到的结果尽可能没有明显规律。在采用除余法时，如果用偶数作为除数，就会出现偶数关键码映射到偶数散列值，奇数关键码映射到奇数散列值的情况。这种情况应该避免。

除余法的一个缺点是相近的关键码（如值相差 1）将映射到相近的值。如果关键码的数字位数较多，可以考虑用较大的除数求余数，然后去掉最低位（或去掉最低的一个或几个二进制位），以排除最低位的规律性。还可以考虑其他方法排除规律性。

基数转换法：先考虑整数关键码。取一个正整数 r ，把关键码看作基数为 r 的数（ r 进制的数），将其转换为十进制或二进制数。通常 r 取素数以减少规律性。

例如，取 $r=13$ 。对于关键码 335647，有下面计算：

$$\begin{aligned}(335647)_{13} &= 3 \times 13^5 + 3 \times 13^4 + 5 \times 13^3 + 6 \times 13^2 + 4 \times 13^1 + 7 \\ &= (6758172)_{10}\end{aligned}$$

这时关键码的取值范围可能不合适，可以考虑用除余法，或者折叠法，或者删除几位数字等方法，将其归入所需下标范围。

实际中也经常遇到字符串作为关键码的情况。最常见的散列方法是把一个字符看作一个整数（直接用字符的编码值），把一个字符串看作以某个整数为基数的“整数”。在这样做时，人们建议以素数 29 或 31 为基数，通过基数转换法把字符串转换为整数，再用整数的散列方法（例如除余法），把结果归入散列表的下标范围。

下面是用 Python 写出的一个字符串散列函数：

```
def str_hash(s):
    h1 = 0
    for c in s:
        h1 = h1 * 29 + ord(c)
    return h1
```

对于其他非整数的关键码，常见做法是先设计一种方法把它转换到整数，而后再用整数散列的方法。在各种散列方法最后，可以用除余法把关键码归入所需范围。

8.3.3 冲突的内消解：开地址技术

前面已经说明，采用散列技术实现字典，冲突是必然出现的事件，因为散列函数是从大集合到小集合的全函数，必然会出现两个不同元素的函数值相同的情况。因此，在设计散列表时，必须确定一种冲突消解方案。

人们提出了一些冲突消解方法，从实现方式上可以分为两类：

- 内消解方法（在基本的存储区内部解决冲突问题）。
- 外消解方法（在基本的存储区之外解决冲突）。

在散列表的使用中需要插入数据项时，用散列函数根据关键码算出了存储位置，但却发现

那里已经有关键码不同的项，这时就知道出现了冲突，必须处理。

对于冲突处理技术，有两方面的基本要求：

- 1) 保证当前这次存入数据项的工作能正常完成。
- 2) 保证字典的基本存储性质：在任何时候，从任何以前存入字典而后没有删除的关键码出发，都能找到对应的数据项。

下面介绍几种常用的冲突消解方法。

开地址法和探查序列

内消解的基本方法称为开地址法，其基本想法是：在准备插入数据并发现冲突时，设法在基本存储区（顺序表）里为需要插入的数据项另行安排一个位置。为此需要设计一种系统的且易于计算的位置安排方式，称为探查方式。

抽象的方法是为散列表定义一种易于计算的探查位置序列。首先定义：

$$D = d_0, d_1, d_2 \dots,$$

这里的 D 是一个整数的递增序列， $d_0=0$ 。而后定义探查序列为

$$H_i = (h(key) + d_i) \bmod p$$

这里的 p 为一个不超过表长度的数。在实际插入数据项时，如果 $h(key)$ 位置空闲就直接存入（这相当于使用 d_0 ）；否则就逐个试探一个个位置 H_i ，直至找到第一个空位时把数据项存入。具体的增量序列有许多可能的设计，例如：

- 1) 取 $D=0, 1, 2, 3, 4, \dots$ ，简单的整数序列，这种方法称为线性探查。
- 2) 设计另一个散列函数 h_2 ，令 $d_i=i \times h_2(key)$ ，称为双散列探查。

开地址法示例

假设关键码为整数，存储数据的表长度为 13，下标范围是 $0 \sim 12$ 。采用简单的散列函数，取 $h(key)=key \bmod 13$ 。这样就确定了散列表的基本设计。

假设有关键码集合（相关数据不需要考虑）：

$$\text{KEY} = \{18, 73, 10, 5, 68, 99, 22, 32, 46, 58, 25\}$$

通过计算可以得到：

$h(18)=5$	$h(73)=8$	$h(10)=10$
$h(5)=5$	$h(68)=3$	$h(99)=8$
$h(22)=9$	$h(32)=6$	$h(46)=7$
$h(58)=6$	$h(25)=12$	

首先考虑线性探查法，图 8.3 给出了插入这些数据的过程中的一些情况：插入前 3 个数据项的过程中没出现冲突，得到状态 (1)。随后要插入关键码 5，算出的散列值是 5，但位置 5 已经存在其他数据，出现冲突。这时就按线性探查规则，将关键码存入位置 6。随后关键码 68 存入位置 3，关键码 99 因冲突存入位置 9，得到状态 (2)。下一个关键码 22 应该存入位置 9，前面也没出现过映射到这里的关键码，但是由于冲突消解，原来应该映射到位置 8 的关键码 99 被存在了这里。按线性探查规则检查随后的位置 10 也已被占，只能将 22 存入位置 11。下面的情况类似，下一关键码应该存入位置 6，但只能存入位置 7；关键码 46 映射到位置 7，但经过一系列试探后存入位置 12。最后的关键码 58 和 25 情况类似，分别存入位置 0 和 1。存入所有关键码后的状态如图中 (3) 所示。

从上面的过程中可以清晰地看到，随着表中数据的增加，产生冲突的可能性也不断增长。而且可以看到数据在表中逐渐堆积成段，使线性探查序列变得越来越长。新关键码不仅与前面

的同义词冲突，而且还可能与由于冲突而迁移来的关键码冲突，情况变得越来越糟糕，字典操作的效率大大下降。

考虑双散列探查，取 $h_2(key) = key \bmod 5 + 1$ 。图 8.4 给出了采用这一套函数存入上面关键码过程中的一些状态。

前 3 项插入中没有冲突，还是达到(1)所示的状态。随后要插入关键码 5，但其位置已有数据。按双散列探查的规则计算 $h_2(5)=1$ ，正好是顺序检查。这导致关键码 5 存入位置 6。下一个 68 存入位置 3；随后的 99 要求存入位置 8 但发现冲突。求出 $h_2(99)=5$ ，探查一步发现位置 0 空闲，将 99 存入这里。随后的关键码 22 被直接存入其散列位置 9，达到状态(2)。下一关键码 32 散列到 6 但发现冲突。求出 $h_2(32)=3$ ，继续探查两步后将 32 存入位置 12。随后的 46 没有冲突被存入位置 7。关键码 58 本应存入位置 6 但出现冲突，求出 $h_2(58)=4$ ，探查一步存入位置 11。最后的 25 被存入位置 1。情况如(3)所示。

0	1	2	3	4	5	6	7	8	9	10	11	12
				18		73	10					
0	1	2	3	4	5	6	7	8	9	10	11	12
			68		18	5		73	99	10		

(1)

0	1	2	3	4	5	6	7	8	9	10	11	12
				18			73		10			
0	1	2	3	4	5	6	7	8	9	10	11	12
	99		68		18	5		73	22	10		

(2)

0	1	2	3	4	5	6	7	8	9	10	11	12
58	25		68		18	5	32	73	99	10	22	46

(3)

图 8.3 散列表的线性探查实例

图 8.4 散列表的双散列探查实例

在双散列探查的过程中，检查的位置以不同方式跳跃进行。这种情况有可能减少关键码堆积的发生。当然，随着表中元素增加，冲突越来越严重的情况是不会改变的。

检索和删除

现在考虑开地址方法下的检索与删除操作。显然，这两个操作共同的第一步是找到关键码的位置，或者确定其不存在，也就是检索。

检索操作：在开地址法的散列表上做检索，工作过程与插入操作的第一步类似。对于给定的 key ：

- 1) 调用散列函数，求出 key 对应的散列地址。
- 2) 检查相应存储位置，如果该位置没有数据项，就说明这个散列表里不存在相应的关键码，检索操作以失败结束。
- 3) 否则（所检查的位置有数据），比较 key 与保存在所确定位置的关键码，如果两者匹配则检索以成功结束。
- 4) 否则，根据散列表的探查序列找到下一个地址，并回到步骤 2。

可见，为判定找不到元素，还需要为单元的无值状态确定一种表示方式。

删除操作的第一步也是基于关键码找到要删除的元素，与检索操作的过程完全一样。但开地址法给删除操作带来了一个麻烦：被删除的数据有可能处于其他元素的探索路径上。如果简单地将其删掉，就可能切断其他元素的探索路径，导致那些元素“失联”，此后它们虽然还在字典里，但却不能被找到。这是不能允许的。

解决这个问题的办法是：不在被删除的元素位置放入空位标志，而是存入另一个特殊标记。在执行检索操作时，将这种标记看作有元素并继续向下探查；而执行插入操作时，则把这种标记看作空位，把新元素存入这里。

8.3.4 外消解技术

在散列表存储区内部解决冲突，可用的手段有限。现在考虑外消解技术。

溢出区方法

人们提出的一种技术是另外设置一个溢出存储区。当插入关键码的散列位置没有数据时就直接插入，发生冲突时将相应数据和关键码一起存入溢出区。数据在溢出区里顺序排列。对应的检索和删除操作也是先找到散列位置，如果那里有数据但关键码不匹配，就转到溢出区顺序检索，直至找到要找的关键码，或确定相应数据不存在。

如果冲突项很少，溢出区里的实际数据非常少，这种方式的效果还不错。当随着溢出区中数据的增长，字典的性能将趋向线性。

桶散列

另一可能做法是数据项不存放在散列表的基本存储区里，而是另外存放。在散列表里保存对数据项的引用。基于这种想法可以开发出很多不同的设计。这种设计被称为桶散列，下面讨论其中最简单的设计，称为拉链法。

在桶散列技术里，散列表的每个元素只是一个引用域，引用着一个保存实际数据的存储桶，桶散列的名称由此而来。具体字典可以采用不同的存储桶结构，在拉链法中一个存储桶就是一个链接的结点表。字典中的数据项并不保存在散列表的主存储区，而是存入相应结点表中的结点里，具有同样散列值的数据项（互为同义词）都保存在这个散列值对应的链表里。图 8.5 描绘了一个采用拉链法的散列字典。采用这种技术，所有数据项都可以统一处理（无论其是否为冲突项），而且允许任意的负载因子。

假设现在要存入的关键码集合 $KEY = \{18, 73, 10, 5, 68, 93, 24, 32, 46, 58, 25\}$ ，再假定散列表的存储区有 8 个存储位置，用取模 8 的余数作为散列函数。将这些关键码顺序加入这个字典，得到的状态如图 8.5 所示。

在这种结构上的各种操作都可以采用统一的模式，通过顺序表的直接位置映射和链接表的顺序操作完成：插入操作需要先找到关键码的散列位置，然后执行插入。最简单实现是把新数据项插在链接表的前端，如果不允许出现重复关键码，就必须检查整个链表。检索关键码时，先通过散列函数找到相应的结点表，而后在其中顺序检查。删除操作的过程与检索类似，找到后删除链接表元素。

在实际应用中，桶散列技术可以有许多变化。拉链法是最简单的桶结构，可以换为其他结构。这种“同义词表”也可以采用顺序表或散列表，还可以采用其他结构。桶散列结构可用于实现大型字典，用于组织大量的数据，包括外存文件等。

8.3.5 散列表的性质

现在讨论散列表的一些重要的一般性质。很容易看到，这里的操作在性质上存在着许多随机因素，与表的状态有关，细致的数学分析只能是概率的。下面只是讨论一些现象，并不准备做严格的数学分析。

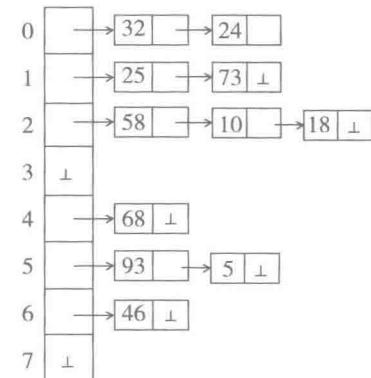


图 8.5 桶散列结构的字典

扩大存储区，用空间交换时间

首先可以看到，无论采用哪种消解技术，随着元素的增加，散列表的负载因子增大，出现冲突的可能性也会同样增大。在开地址法中，表中数据的增加最终将导致存储区溢出；采用溢出区方法，产生的情况是溢出区越来越大，检索效率越来越趋于线性；在拉链法中，负载因子增大就表现为链的平均长度增加。

显然，可以考虑在负载因子达到一定程度时，扩大散列表的基本存储表。在前面讨论顺序表时，已经仔细研究过扩大存储的策略和性质。现在的新问题是不能简单地把元素拷贝到新存储区。存储区扩大后，需要相应调整散列函数，以便尽可能利用增加的存储单元（除余法可以用在这里）。然后，需要把字典里已有数据项重新散列到新存储区。可见，扩大存储要付出重新分配存储区和再散列装入数据项的代价。

有趣的是，对于散列表，只需简单地扩大存储，就能从概率上提高字典的操作效率。这是明显的用空间交换时间（这是计算机科学技术领域的一条基本原理）。

负载因子和操作效率

根据上面的讨论可知，散列表的负载因子对效率有决定性的影响。人们对这个问题做了一些研究，总结出一些情况和经验。采用内部消解技术时，负载因子 $\alpha \leq 0.7 \sim 0.75$ 时，散列表的平均检索长度接近于常数。

如果采用桶散列技术，负载因子 α 就是桶的平均大小（采用拉链法时是链的平均长度）。这种技术可以容忍任意大的负载因子，但随着负载因子变大，检索时间也趋于线性（显然，平均桶长 = 数据项数 / 桶数），只是除以一个常量因子。

应该看到，散列表字典的许多性质都是概率性的，而不是确定性。人们说散列表是一种高效的字典实现技术，有些基本假设：

- 实际存入字典的数据（的关键码）的散列函数值分布均匀。
- 字典散列表的负载因子不太高（试验证明应在 0.7 以下）。

在这些假设下，散列表字典检索、插入、删除操作的时间开销可以看作常量。

虽然散列表字典从概率上看极为高效，但事情也有另一面，有一些必然的情况：

- 其常量时间的操作代价是平均代价，不是每次操作的实际代价。由于不同元素的探查序列长度不一，可能出现有些操作代价较高的情况。
- 一般而言，关键码冲突是必然会发生的情况，并因此导致不同操作的代价之间的差异可能很大，而且不能事先确知。
- 不断插入元素将导致负载因子不断增大。为了保证操作效率就需要扩大存储，从而导致一次很高代价的插入操作。而且，无法预计这种高代价操作在什么时候出现。
- 字典内部数据项的排列顺序无法预知，也没有任何保证。

还有一些不好的情况也有可能出现，例如：

- 有可能出现实际数据（的关键码）的散列值相对集中的情况，导致一些操作的性能非常差。极端情况是大量常用数据项集中在一个或几个散列值，导致非常长的探查序列，从而使散列表字典非常低效。
- 基于内消解机制的字典，在长期使用中性能通常会变差。因为在长期使用可能产生很长的已删除元素序列，影响很多操作的效率。

可能技术和实用情况

前面介绍的是基本散列表操作。实际实现还可以考虑许多问题，例如：

- 给用户提供检查负载因子和主动扩大散列表存储区的操作。这样，用户就可以在一段效率要求高的计算之前，根据需要首先设定足够大的存储。
- 对于开地址散列表，记录或检查被删除项的量或比例，在一定情况下自动整理。最简单的方法是另外分配一块存储区，把散列表里的有效数据项重新散列到新区。这种重新散列可以消去开地址散列表里所有已删除项的空位。

散列表字典在计算机软件中使用非常广泛，人们已经积累了许多设计和实现的经验。许多编程语言或标准库提供了基于散列表的字典结构，经常被称为 map，或者 table，或者 dictionary。有些软件以散列表作为基本实现技术，如著名的数学软件 Maple 等。

Python 的标准类型 dict 就是基于散列表实现的，有关情况将在下面介绍。Python 系统实现中的很多地方也使用了散列表，后面将简单介绍。

8.4 集合

集合是数学中最重要的一个基本概念，主要关注个体与个体的汇集之间的关系。其中的个体称为元素，是不加定义的概念，个体的汇集称为集合。

在计算机科学技术领域，具体的数据项可以看作个体，数据项的汇集就是集合，因此可以借用数学的概念和相关的定义，处理数据与数据汇集之间的关系。这就是讨论作为数据结构的集合概念和结构时希望考虑的问题。

8.4.1 集合的概念、运算和抽象数据类型

在讨论集合时，首先有一批被关注的个体，它们有清晰的定义而且互不相同。具体的个体是什么依赖于实际问题，在讨论集合的概念及其相关论题时并不关心。

概念和集合描述

一个集合 S 就是一些个体的汇集。如果集合 S 里有个体 e ，就说 e 是 S 的一个元素，或者说 e 属于集合 S ，用 $e \in S$ 表示这个事实。个体 e 不属于集合 S 用 $e \notin S$ 表示。包含（当前考虑的）所有个体的集合称为全集。

描述集合的一种方法是明确列出其中的所有元素，这种写法称为集合的外延表示，具体写法是用一对大括号，在其中列出集合的所有元素。例如 $\{1, 2, 3\}$ 表示了一个包含三个整数的集合。显然，这种外延表示只能描述有穷集合。

集合的另一种描述方法是给出集合中的元素应该满足的性质，这种表示称为集合的描述式，或者集合的内涵表示。这种写法需要用到某种描述性质的方法，严格的方法是用逻辑公式。具体写法是 $\{e \mid p\}$ ，其中 e 可以是一个表达式，描述集合中的元素， p 描述一些变量的性质，这些变量用在表达式 e 里“生成”集合的元素。例如：

$\{x \mid x \text{ 是自然数且没有除 } 1 \text{ 和其自身之外的因子}\}$

$\{x+2y \mid x, y \in \mathbb{N} \wedge x \bmod y = 3\}$

$\{s+t \mid s \text{ 是全为字符 } a \text{ 的串, } t \text{ 是全为字符 } b \text{ 的串}\}$

在第一和第三个描述式里，条件（性质）用自然语言和符号的组合形式描述，在第二个集合描述中用逻辑公式描述。在下面讨论中，有些地方也借助于读者的直观理解，并不特别强调用严格完整的逻辑公式。第一个集合是素数集合；第二个集合中的符号 \mathbb{N} 表示自然数集，所定义集合的元素通过所有满足条件的自然数算出；第三个集合里包含了所有前面一段全是 a 后随一段全是 b 的字符串。

一个集合中元素的个数称为该集合的基数，或说是该集合的大小。无穷集合也有不同的基数，是集合论中的一个基本结论。一个集合里可以不包含任何元素，这种集合称为空集，用 \emptyset 或 $\{\}$ 表示。

两个集合 S 和 T 相等，当且仅当它们包含同样的元素。

集合之间另一重要关系是子集关系。如果集合 S 中所有的元素都是集合 T 的元素，则说 S 是 T 的子集，记为 $S \subseteq T$ 。显然，一个集合也是其自身的子集，空集是任何集合的子集。此外，如果两个集合相等，则它们互为子集。如果 S 是 T 的子集，但两个集合不相等（也就是说， T 包含了不属于 S 的元素），则说 S 是 T 的真子集，用 $S \subset T$ 表示。

当然，一个集合可以是或者不是另一集合的子集， S 不是 T 的子集用 $S \not\subseteq T$ 表示，类似的记法还有 $S \not\subset T$ 。

集合运算

集合运算是从已有集合出发构造新集合。几个重要集合运算如下：

求并集运算从两个集合 S 和 T 出发求出它们的并集，记为 $S \cup T$ ，定义是：

$$S \cup T = \{e \mid e \in S \vee e \in T\}$$

这里的 \vee 表示逻辑公式的或运算，也就是说， $S \cup T$ 的元素或是 S 的元素，或者是 T 的元素，而且恰好是那些元素。

求交集运算从两个集合 S 和 T 出发求出它们的交集，记为 $S \cap T$ ，定义是：

$$S \cap T = \{e \mid e \in S \wedge e \in T\}$$

这里的 \wedge 表示逻辑公式的与运算，也就是说， $S \cap T$ 包含且仅包含了那些既属于 S 也属于 T 的元素。

求差集运算从两个集合 S 和 T 出发求出它们的差集，记为 $S - T$ ，定义是：

$$S - T = \{e \mid e \in S \wedge e \notin T\}$$

也就是说， $S - T$ 包含且仅包含那些属于 S 但不属于 T 的元素。

抽象数据类型

在讨论集合的实现问题之前，先定义一个抽象数据类型如下：

ADT Set:	# 集合抽象数据类型
Set(self)	# 集合构造函数，创建新的空集
is_empty(self)	# 检查 self 是否为一个空集
member(self, elem)	# 检查 elem 是否为本集合中的元素
insert(self, elem)	# 将元素 elem 加入集合，为变动操作
delete(self, elem)	# 从集合中删除元素 elem，为变动操作
intersection(self, oset)	# 求出本集合和另一集合 oset 的交集
union(self, oset)	# 求出本集合和另一集合 oset 的并集
different(self, oset)	# 求出本集合减去另一集合 oset 的差集
subset(self, oset)	# 判断本集合是否为 oset 的子集
.....	

作为数据结构的集合，应该支持建立新集合、修改已有集合，以及基于已有集合建立新集合的操作，还有检查集合与元素关系的 member 检查、判断子集关系的 subset 检查。这里的集合运算（求交集、并集和差集）都是非变动操作，它们生成一个新集合作为运算的结果。也可以定义相应的变动操作。

8.4.2 集合的实现

现在考虑集合的实现。集合显然是一种汇集数据结构，可以采用前面讨论过的各种实现元

素汇集的技术。

在这里最基本的成员关系判断需要在集合里做检索，集合的元素在这里起的作用类似于关键码在字典中的作用，只是没有关联数据。这也就是说，任何字典实现技术都可以用于实现集合（包括后面将要讨论的技术），为此，只需把集合元素直接存储在保存字典项（关联）的位置。集合的最基本操作是判断元素与集合的关系，对应于字典查询；集合数据结构所需要的创建/空集检查/加入/删除等操作，字典里都有与之对应的操作。

要实现集合数据结构，需要考虑的新问题只剩下常用集合运算的实现。求并集和交集、求差集（也被称为相对于某个集合的补集）都是从两个已有集合得到另一个集合。当然，在做这些事情时，特别需要考虑操作的实现效率。

简单线性表实现

显然，线性表技术可以作为实现集合的基础，现在研究这个问题。

考虑用顺序表作为集合的实现结构，立刻就遇到了一个问题。由于从集合之外看不到集合内部，只能通过 member 判断元素关系。实现中是否保持集合中元素的唯一性，只是一种设计选择。从使用的角度看，集合的实现只需要保证下面性质：将一个元素加入集合后，判断它存在于集合里的结果必须是真；而没加入的元素，或者已删除的元素，判断它存在于集合里的结果是假。只要满足这些约束，内部实现可以有些变化。

易见，下面两套插入和删除方式都可以正确实现集合的功能：

- 1) 插入元素时，检查它是否已在集合里，保证集合（线性表）中元素的唯一性；删除元素时找到第一个相同元素并将其删除。
- 2) 插入元素时简单将其加入集合（线性表）里；删除时检查整个表，删除指定元素的所有拷贝（由于插入操作的实现方式，同一元素可能有多个出现）。

不难看到，这两套实现方案的操作代价有些不同。下面只考虑保持元素唯一性的实现方式，另一种方式请读者自己考虑。

有了保持元素唯一性的要求，建立和维护一个集合，就是建立和维护一个无重复元素的顺序表。判断元素关系就是在表中检索元素的存在性，是 $O(n)$ 时间操作，其中集合元素个数为 n 。下面考虑集合运算的实现，这是新问题。

先看求两个集合的交集，这时需要找到所有同属两个集合的元素。为找到这些元素，就要逐个考察一个集合的元素，如果它也属于另一集合就将其加入结果集合。设被操作的两个集合的元素分别为 m 和 n 个。检查第一个集合的各元素，需要检查元素 m 次。判断每个元素是否也属于另一集合，是 $O(n)$ 操作。所以，求交集操作的复杂度是 $O(m \times n)$ 。不难看到，求并集和求差集的情况与此类似，几个操作的代价都比较高。

由于这些操作都用顺序扫描的方式工作，具体实现是采用顺序表或链接表都可以，操作效率方面没有本质的不同。总而言之，用简单线性表实现集合，技术比较简单，主要缺点是元素判断和几个集合运算的操作效率都比较低。

排序顺序表实现

前面已知，如果元素上存在一种序关系，例如整数的小于等于、字符串的字典序等，就可以在顺序表里采用排序的方式存储元素。这样做，检索（元素判断）可以在 $O(\log n)$ 时间完成，与简单顺序表相比有质的改进。要保证元素唯一性就必须扫描表中元素，设表中元素从小到大排序，插入元素时可以用下面方式：

假设要把e插入S

用二分检索在S里查找e
如果找到就结束
否则就确定了插入位置，后移后面的元素并实际插入e

显然，这个操作的复杂度与简单顺序表中的相应操作一样，实际使用中有可能更高效（如果 S 里有 e 就不必插入元素）。删除操作也用同样方式实现。

更重要的是，采用排序方式存储表中元素，能够大大提高各种集合运算的效率。作为例子，这里只考虑求交集的算法，其他操作可以类似地实现。

假设需要求交集的集合 S 和 T 由两个 Python 的表 s 和 t 表示，结果集合用表 r 表示。下面是求交集的算法（注意，这里假设 s 和 t 的元素都从小到大排列）：

```
r = []
i = 0      # i和j是s和t中下一次检查的元素的下标
j = 0
while i < len(s) and j < len(t):
    if s[i] < t[j] :
        i += 1
    elif t[j] < s[i] :
        j += 1
    else:  # s[i] = t[j]
        r.append(s[i])
        i += 1
        j += 1
# 现在r就是得到的交集
```

显然，如果 s 的当前元素小于 t 的当前元素，s 的当前元素肯定不应该在两集合的交集里，应该推进一步去考虑 s 的下一元素；t 中当前元素较小的情况是同样道理。只有两个集合的当前元素相等时，该元素属于交集。

现在考虑上述算法的复杂度。有一个情况很显然：主循环的每次迭代总能处理掉两个集合里的至少一个元素，所以，这个操作的时间复杂度是 $O(m+n)$ ，与简单顺序表的 $O(m \times n)$ 相比是质的提高。不难看出，类似技术也可以用于实现并集 / 差集操作，将这两个操作的时间复杂度也变为 $O(m+n)$ 。

散列表实现

完全可以考虑基于散列表实现集合，这样：

- 一个集合就是一个散列表。
- 插入 / 删除元素对应于散列表中插入 / 删除关键码。
- 集合元素判断对应于关键码检索。
- 各种集合运算都基于上面几个散列表操作，采用建立新散列表的方式实现，没有实质性的困难。

这样，各种集合操作的效率就完全由散列表的性质确定，具有概率性。在最佳情况下，各种操作都很高效：插入 / 删除和元素判断操作的代价几乎是常量；求交集 / 并集 / 差集具有近乎 $O(m+n)$ 复杂度，其中 m 和 n 为参加运算的两个集合的大小。注意，为实现这些操作，需要有一种遍历散列表的方法，不难实现。

如果已经有了散列表结构，基于它实现一种集合数据类型不是很困难的工作。可作为简单的编程练习，这里不进一步讨论。

8.4.3 特殊实现技术：位向量实现

在结束这一节之前，最后考虑一种特殊的集合实现技术。

不难看到，一个元素是否属于一个集合，是一种二值判断。基于这一认识，人们提出了一种专门的集合实现技术：集合的位向量表示。如果在程序里需要使用的一批集合对象有一个（不太大的）公共超集 U ，也就是说，需要实现和使用的集合都是某个集合 U 的子集，就可以考虑用位向量技术实现这些集合。具体实现方法是：

- 假定 U 包含 n 个元素，给每个元素确定一个编号作为该元素的下标。
- 对任何一个要考虑的集合 S （注意，根据上面的假定， $S \subseteq U$ ），用一个 n 位的二进制序列（位向量） v_S 表示它，方法是：对任何元素 $e \in U$ ，如果 $e \in S$ ，就令 v_S 里对应于 e （的下标）的那个二进制位取值 1，否则就令该位取值 0。

显然，要检查一个元素 e 是否属于集合 S ，只需检查 v_S 里对应于 e 的那个位是否为 1；将元素 e 加入集合 S 实现为将 v_S 里对应于 e 的位设置为 1；从 S 里删除 e 实现为将 v_S 里对应于 e 的位设置为 0。另外， v_S 里值为 1 的二进制位的个数就是 S 的元素个数；全 0 的 n 位二进制序列对应于空集，全 1 的二进制序列对应于 U 。

例如，假设 U_1 是集合 $\{a, b, c, d, e, f, g, h, i, j\}$ ，包含 10 个元素，按字母序将这些字母分别对应到下标 0, 1, 2, …, 9, U_1 的任何子集都能用一个 10 位的位向量表示：{} 用 0000000000 表示， U_1 用 1111111111 表示， $S_1 = \{a, b, d\}$ 对应于 1101000000， $S_2 = \{a, e, i\}$ 对应于 1000100010。

位向量集合的集合操作都可以通过逐位操作实现：

- v_S 和 v_T 的第 i 位都是 1 时， $v_{S \cap T}$ 的第 i 位取值 1，否则取值 0。
- v_S 和 v_T 的第 i 位都是 0 时， $v_{S \cup T}$ 的第 i 位取值 0，否则取值 1。
- v_S 的第 i 位是 1 而 v_T 第 i 位是 0 时， v_{S-T} 第 i 位取值 1，否则取值 0。

例如，设 U_1 和下标安排皆同上例，下面是几个集合运算的例子：

子集	位向量表示
$S = \{a, b, d\}$	1101000000
$T = \{a, e, i\}$	1000100010
$S \cap T = \{a\}$	1000000000
$S \cup T = \{a, b, d, e, i\}$	1101100010
$S - T = \{b, d\}$	0101000000

应注意：采用位向量表示，操作的代价都需要基于集合 U 的大小度量，而不能基于被操作集合的实际元素个数度量。因为，无论被操作集合的元素多么少，即使是空集，它的表示也与其他集合一样大，扫描其元素也需扫描整个位向量。

集合的位向量表示比较紧凑，空间利用率较高，在一些情况下比较适用：全集 U 的规模适当，不是太大，而且需要处理的是 U 的一批子集。

位向量集合常被用在操作效率要求比较高，或者存储资源比较受限的环境中。通常采用较低级的语言实现，例如用 C 语言。

如前所述，所有适合用于实现字典的技术，都可以用于实现集合。后面讨论的字典实现技术都可以作为集合的实现基础，有关情况请读者考虑。

8.5 Python 的标准字典类 dict 和 set

由于字典和集合都是程序中经常需要使用的数据结构，Python 语言的内置类型包括一个字典类型（dict）和两个集合类型（set 和 frozenset）。

在 Python 语言的官方实现里，字典和两个集合类型都是基于散列表技术实现的数据结构，

采用内消解技术解决冲突。下面介绍一些细节，以 dict 为例。

- dict 类型的对象采用散列表技术实现，其元素是 key-value（关键码 – 值）对，关键码可以是任何不变对象，值可以是任何对象。
- 在创建空字典或者很小的字典时，初始分配的存储区可容纳 8 个元素。
- dict 对象在负载因子超过 2/3 时自动更换更大的存储区，并把已经保存的内容重新散列到新存储区里。如果当前字典对象不太大，就按当时字典中实际元素的 4 倍分配新存储区。当字典里的元素超过 50000 时，改为按实际元素个数的 2 倍分配新存储区。

上面说的是字典 dict，其实 Python 中集合的情况与此类似，许多实现代码完全一样。当然，frozenset 是不变对象，一旦建立后不会动态变化。

字典等类型除了作为提供给编程序的人使用的数据结构外，在解释器系统的实现中也很有用。例如，在官方 Python 系统，不少内部机制的实现也基于字典结构，如全局 / 模块 / 类的名字空间等。在程序运行中，需要找到各种名字的关联信息。在一个名字空间里，特别是全局的或大型模块里，可能定义了很多名字，用字典实现效率较高。

Python 规定 dict 的关键码，以及 set 和 frozenset 的元素都只能是不变对象，是为保证散列表的完整性（为保证数据项检索和删除的正确实现）。这里的原因也很清楚：如果允许关键码为可变对象，插入元素时根据关键码计算的位置，在关键码变动后就不符合散列检索的要求了，随后就无法正确找到它们。如果确实需要修改数据的关键码，只能是先删除数据项，而后用新关键码重新插入。

在 Python 标准函数中有一个 hash 函数，其功能就是按特定的方式计算参数的散列值。对一个对象调用 hash 函数，它或者返回一个整数，或者抛出一个参数异常，表示本函数对该对象无定义。hash 函数无定义的对象如 list 对象，或包含可变成分的序对对象，调用时将抛出异常 “`TypeError: unhashable type`”。在 dict、set 等类型的实现中，都用这个 hash 函数计算关键码（或元素）的散列值。

对 Python 的各种内置的不变类型，hash 函数都有定义，包括内置的不变组合类型，如 str、tuple、frozenset 等。这个函数的定义还保证，当 `a == b` 成立时两个对象的 hash 值也相同。

Python 在这里采用了与其他机制一致的设计：程序调用标准函数 hash 时，解释器将到参数所属的类里去找名为 `__hash__` 的方法。在 Python 的内置类型中，标准函数 hash 有定义的类型都有自己的 `__hash__` 方法，没有 `__hash__` 方法即是 hash 函数无定义。

如果希望自定义类的对象也能作为 dict 或 set 等的关键码，就应该为这个类定义一个 `__hash__` 方法。该类的对象是否可变是用户自己的事情。如果对象可变，加入字典或集合后修改这种对象的值，带来的后果只能自己负责。

8.6 二叉排序树和字典

从实现字典功能的角度看，前面研究的两种结构各有优点和缺点：

- 基于线性表实现字典，结构简单，易于实现，但是
 - 基于简单的线性表，检索效率很低，插入 / 删除的效率也很低。
 - 基于元素排序的顺序表，检索效率大大提高，但插入 / 删除的效率仍然很低。这种实现只适用于关键码上存在某种序的情况。

用顺序表作为实现字典的基础技术，总存在一些低效操作，因此只适用于小型字典的实现，不适合用于实现大型字典。

- 散列字典的操作效率高（基于概率考虑），对关键码类型无特殊要求，应用广泛。但是，基于散列的字典没有确定性的效率保证，不适合用于对效率有严格要求的环境。另外，散列表中不存在遍历元素的明确顺序。

还有一点也值得提出。这两种结构都把字典元素存储在一个连续存储块里，管理比较方便 \ominus 。但这样做，如果字典很大，就需要很大块的连续存储，动态修改不太方便（例如顺序表插入和删除，可能移动很多元素），也难以用于实现非常巨大的字典。而目前大型的计算机应用系统越来越多，对巨型字典的需求与日俱增。

为了更好地支持存储内容的动态变化，应该考虑采用链接结构。此外，基于链接结构，也很容易用大量的小存储块构造出大规模的容器，包括巨型的字典。

考虑上面分析中看到的各种情况，很容易想到树形结构，因为：

- 它们可以用链接方式实现，因此比较容易处理元素的动态插入 / 删除问题。
- 在树形结构里，从根到任意节点的平均路径的长度可以小到结点个数的对数，因此有可能实现高效操作（只要能维持良好的树形结构）。

人们研究可能用于实现字典的各种树形结构，除了有关正确高效地实现字典功能的基本考虑外，还关注了另外一些重要问题，如：

- 支持高效的结构调整，保证长期工作的系统仍能维持良好的性能。
- 支持实现大型字典的典型需要，例如支持数据库系统等的实现。
- 尽可能很好地利用计算机系统的存储结构，例如，很好地利用内存和外存（磁盘、磁带等），以及硬件缓存等多层次存储结构。
- 用于为大型数据集合建立索引，提高各种复杂（复合）查询的效率。

下面主要讨论最简单的树形结构——二叉树，研究基于二叉树的字典实现技术。可以看到，在用于实现字典时，二叉树也有多种不同的使用方式。最后一节将简单介绍其他树形结构在实现字典方面的应用。

在开始讨论之前，首先要请读者特别注意二叉树（以及其他树形结构）的最重要特点（包括正反两个方面）：

- 如果树结构“良好”，其中最长路径的长度与树中结点个数呈对数关系。
- 如果树结构“畸形”，其中最长路径的长度可能与结点个数呈线性关系。

8.6.1 二叉排序树

本节首先介绍二叉排序树（Binary Sort Tree）的概念。简言之，二叉排序树是一种存储数据的二叉树，把字典的数据存储和查询功能融合在二叉树的结构里。采用二叉树作为字典存储结构，有可能得到较高的检索效率；采用链接式的实现方式，数据项的插入、删除操作都比较灵活方便。这里的基本想法是：

- 在二叉树的结点里存储字典中的信息。
- 为二叉树安排好一种字典数据项的存储方式，使字典查询等操作可以利用二叉树的平

\ominus 除了桶散列技术，它是连续存储和链接存储的组合。但其中还是要有一个连续的元素存储区，随着字典内元素的增长，这个存储区也可能变得很大。

均高度（通常）远小于树中结点个数的性质，使检索能沿着树中路径（父子关系）进行，从而获得较高的检索效率。

二叉排序树可以用于实现关键码有序（存在明确定义的序关系）的字典，树中数据的存储和使用都利用了数据（或关键码）的序。

定义和性质

定义 二叉排序树是一种在结点里存储数据的二叉树。一棵二叉排序树或者为空，或者具有下面的性质：

- 其根结点保存着一个数据项（及其关键码）。
- 如果其左子树不空，那么其左子树的所有结点保存的（关键码）值均小于（如果不要求严格小于，也可以是“不大于”）根结点保存的（关键码）值。
- 如果其右子树不空，那么其右子树的所有结点保存的（关键码）值均大于它的根结点保存的（关键码）值。
- 非空的左子树或右子树也是二叉排序树。

显然，二叉排序树也是一种递归结构，其左右子树具有与整棵树同样的结构。不仅要求它们是二叉树，还要确定结点中存放数据的方式。

根据二叉排序树中数据的存储方式，不难看出，如果对一棵二叉排序树做中序遍历，得到的将是一个按关键码值上升排序的序列。如果树中存在重复关键码，虽然关键码相同的数据项的前后顺序不能确定，但这些项必定位于中序遍历序列中的相邻位置。

显然，前面介绍过的二分法检索的判定树都是二叉排序树。

例如，考虑关键码的序列 $KEY=[36, 65, 18, 7, 60, 89, 43, 57, 96, 52, 74]$ 。图 8.6 给出了两棵二叉树，其结点里都存储着这组数据。仔细检查不难确认，这两棵二叉树都是二叉排序树。由此实例可见，同一集数据对应的二叉排序树不唯一。

下面的讨论将集中关注二叉排序树本身的结构，忽略关键码以外的其他数据项部分，因为它们对这种数据结构的性质和操作都没有影响。

性质 一棵结点中存储着关键码（数据）的二叉树是二叉排序树，当且仅当通过中序遍历这棵二叉树得到的关键码序列是一个递增序列。

实际上，二叉排序树既可以作为字典，将数据直接保存在树结点里，也可作为索引结构，实际数据另外存储，在二叉树的结点里与关键码关联的是数据的存储位置信息。因此，下面的讨论将不区分二叉排序树是作为字典的索引还是作为字典本身，因为从检索的角度看，两者没有什么差别。讨论其他结构时也这样考虑。

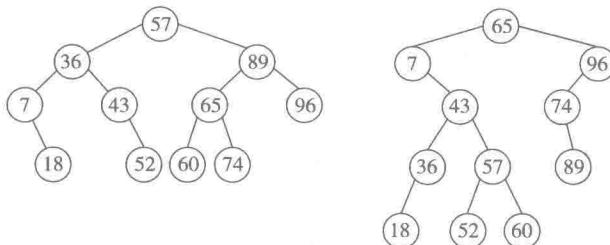


图 8.6 二叉树排序树实例

二叉排序树上的检索

现在考虑二叉排序树的实现。显然，任何可以用于实现二叉树的技术都能用于实现二叉排

序树。但是，为了支持树结构的动态变化，最常见的是采用链接结构，可以考虑基于前面定义的 BinTNode 类实现二叉排序树。

在研究二叉排序树类的完整实现之前，首先考虑二叉排序树上的检索算法。由于在这种树（及其子树）的根中保存的数据总把树中的数据划分为较小和较大的两组，用检索关键码与之比较，就可以知道下一步应该到哪棵子树去检索（这一过程是递归的）。下面的函数用一个循环实现这个过程：

```
def bt_search(btreetree, key):
    bt = btreetree
    while bt is not None:
        entry = bt.data
        if key < entry.key:
            bt = bt.left
        elif key > entry.key:
            bt = bt.right
        else:
            return entry.value
    return None
```

检索过程很清晰，就是根据被检索关键码与当前结点关键码的比较情况，决定是向左走还是向右走。遇到要检索关键码时成功结束，函数返回关键码的关联值；在无路可走时失败结束，函数返回 None。

二叉排序树（字典）类

现在考虑基于二叉排序树的思想定义一个字典类，并分析和实现类中的各主要算法。这个类的接口应该参考前面给出的字典抽象数据类型，其基础元素是前面定义的二叉树结点类和类，检索方法是上面检索函数的简单修改：

```
class DictBinTree:
    def __init__(self):
        self._root = None

    def is_empty(self):
        return self._root is None

    def search(self, key):
        bt = self._root
        while bt is not None:
            entry = bt.data
            if key < entry.key:
                bt = bt.left
            elif key > entry.key:
                bt = bt.right
            else:
                return entry.value
        return None
```

实现二叉排序树字典最关键的操作是数据项的插入和删除，这两个操作通常都要修改树的结构。例如，在做插入操作时，不仅需要保证二叉树的结构完整，将新的数据项加入树中（需要插入新结点），还要保持树中结点数据的正确顺序。

首先考虑字典项的插入。易见，对插入操作的基本要求是能够把新数据项加入字典（二叉排序树）中，并维持二叉树的完整性，包括关键码的顺序要求。为此，就需要找到加入新结点的正确位置，并将新结点正确连接到树中。

字典插入操作总会遇到一个问题：如果遇到与检索关键码相同的数据项怎么办？这种情况应该根据实际需要处理，有多种可能做法。例如将这种情况看作出错，或者什么也不做（不插入新项），或者允许关键码重复的项而直接插入新结点^Θ。下面简单处理这个问题，总用新值替换关键码的已有关联值，这就保证字典不会出现关键码重复的项。

查找位置就是用关键码检索，基于检索插入数据的基本算法如下：

- 如果二叉树空，就直接建立一个包括新关键码和关联值的树根结点。
- 否则搜索新结点的插入位置，沿子结点关系向下：
 - 遇到应该走向左子树而左子树为空，或者应该走向右子树而右子树为空时，就是找到了新字典项的插入位置，构造新结点并完成实际插入。
 - 遇到结点里的关键码等于被检索关键码，直接替换关联值并结束。

函数的实现如下：

```
def insert(self, key, value):
    bt = self._root
    if bt is None:
        self._root = BinTNode(Assoc(key, value))
        return
    while True:
        entry = bt.data
        if key < entry.key:
            if bt.left is None:
                bt.left = BinTNode(Assoc(key, value))
                return
            bt = bt.left
        elif key > entry.key:
            if bt.right is None:
                bt.right = BinTNode(Assoc(key, value))
                return
            bt = bt.right
        else:
            bt.data.value = value
            return
```

应该给字典定义一个迭代器方法，生成其中所有值的序列，以便字典的使用者通过 `for` 循环或其他方式使用字典里的数据。下面的生成器方法完成提供这一功能：

```
def values(self):
    t, s = self._root, SStack()
    while t is not None or not s.is_empty():
        while t is not None:
            s.push(t)
            t = t.left
        t = s.pop()
        yield t.data.value
        t = t.right
```

这个函数做的就是中序遍历。对字典而言，按其他遍历序的价值不大。

有时用户可能希望得到字典里各关键码 - 值对，最简单的方式是修改上面函数里的 `yield` 语句，让它直接返回 `t.data`，这是一个 `Assoc` 类对象。但如果深入思考，就会发现这种做法极不安全。实际上，作为 `t.data` 值的对象不仅存储着一对特殊的关键码 - 值关联，

^Θ 请读者考虑如何实现，及其对检索、删除等操作的影响，作为练习。

这个值还有其他意义：其中的关键码是整个二叉排序树的数据完整性的一部分。如果用户得到了这个关联对象，有意或无意地修改了其中 key 成分的值，整个字典可能就被破坏了，不再是一个二叉排序树了。而用户未必清楚这样做的后果，无论有关用户手册中发出多么严重的警告，或早或晚一定会有人在这里出错。

看到这种危险，下面方法里采用了另一种合理的设计：在找到了所需的关联后，取出其中的关键码和值，重新做成一个序对返回。

```
def entries(self):
    t, s = self._root, SStack()
    while t is not None or not s.is_empty():
        while t is not None:
            s.push(t)
            t = t.left
        t = s.pop()
        yield t.data.key, t.data.value
        t = t.right
```

这样，只要关键码本身不是可变对象，用户就不能破坏字典的完整性。这一讨论也再次说明了 Python 对其内置字典 dict 的关键码的基本要求的意义。

下面考虑二叉排序树结构中最复杂的操作：删除具有给定关键码的元素。要求还是两条，既应该修改被操作的二叉树，保证需要删除的数据项被实际删除，又保证其他数据项仍然可以正常查询和使用。简而言之，删除所要求的元素后，剩下的结构还必须是二叉排序树。当然，如果要求删除的关键码不存在，二叉树应该不变。

完成这种删除有许多可能做法。例如，遍历这棵树，用树中的项另做一棵二叉排序树，构造中丢掉要删除的项。如前所述，剩下的项可以做出多棵不同的树。这个方法的缺点是代价高，需要耗费与树中结点成比例的时间。

要降低操作代价，就应尽量利用原有结构，只做局部的修改和调整。下面介绍一种方法供参考，并请读者考虑其他删除方法。这里的基本想法是先找到需要删除的树结点，将其去除。如果删除破坏了二叉排序树的结构，就在被删结点周围做尽可能小的局部调整。请注意二叉排序树的性质：对其做中序遍历，得到的关键码序列应该是递增的。在分析和检查下面方法的正确性时，可以利用这个性质。

假设已经确定应该删除结点 q ，它是其父结点 p 的左子结点（为 p 的右子结点的情况类似），这时有两种情况：

- 1) q 是叶结点，这时只需将其父结点 p 到 q 的引用置为 None，删除就完成了。显然，树中剩下的结点仍然构成一棵二叉排序树。
- 2) q 不是叶结点，那么就不能简单删除，需要把 q 的子树连接到删除 q 之后的树中，而且要保证关键码的顺序。这时又可以分为两种情况（参看图 8.7）：
 - ① 如果 q 没有左子结点，情况如图 8.7 中 (1) 上图所示。这时只需把 q 的右子树直接改作其父结点 p 的左子树，得到 (1) 下图的状态。在删除之前，中序遍历序列中先是树中 L 部分的中序遍历序列，而后是 q ，再后是 q 的右子树 N 的中序序列，再是 p 和 R 部分的中序序列。删除之后，除了结点 q 已经不在，其余部分的顺序不变。如果原来序列中的关键码递增，删除后也一样。
 - ② q 有左子树，情况如图 8.7 中 (2) 上图所示（虽然 q 可能没有右子树，但可以统一处理）。这时先找到 q 的左子树的最右结点，设为 r ，显然它没有右子树。用 q 的左子

结点代替 q 作为 p 的左子结点，并将 q 的右子树作为 r 的右子树，就得到(2)下图的状态。比较删除前和删除后的中序序列，可知这种做法正确。

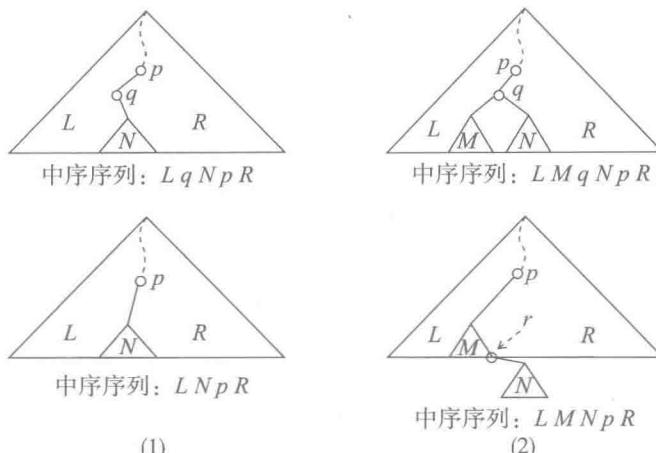


图 8.7 二叉树排序树的数据删除

由于在所有情况下，删除了结点后的树的中序遍历序列中，结点的关键码仍然是递增排列的，所以这样的删除操作是正确的。

有读者可能对图 8.7 中(2)下图的状态不太满意，觉得这样操作可能导致二叉树的高度增加。应该说，上述操作方式确实有这种可能性，当然具体操作是否导致树增高与原树的结构有关。前面说过，完成删除操作的方法不止一种。首先查找到要删除的结点，而后删除并恢复树结构的方法也不止一种。本章练习要求读者设计另一种方法。

下面是实现删除的方法，完全按上面讨论的方式工作，其中统一处理了被删结点 q 是其父结点 p 的左子结点或右子结点的情况。另外，在实际删除时还要检查被删结点的情况，如果它是根结点就需要特殊处理：

```

def delete(self, key):
    p, q = None, self._root      # 维持p为q的父结点
    while q is not None and q.data.key != key:
        p = q
        if key < q.data.key:
            q = q.left
        else:
            q = q.right
        if q is None:
            return          # 树中没有关键码key

    # 到这里q引用要删除结点，p是其父结点或None(这时q是根结点)
    if q.left is None:          # 如果q没有左子结点
        if p is None:           # q是根结点，修改__root
            self._root = q.right
        elif q is p.left:       # 根据q和p的关系修改p的子树引用
            p.left = q.right
        else:
            p.right = q.right
    return

    r = q.left                  # 找q左子树的最右结点
    while r.right is not None:

```

```

r = r.right
r.right = q.right
if p is None:           # q是根结点，修改 _root
    self._root = q.left
elif p.left is q:
    p.left = q.left
else:
    p.right = q.left

```

为了检查二叉树的情况，定义一个输出树中信息的方法。这种方法在程序开发中很有用，在这个类的实际使用中可能很少用到：

```

def print(self):
    for k, v in self.entries():
        print(k, v)

# END class

```

最后定义一个函数，它不是二叉排序树类的一部分，是一个独立函数，基于一系列数据项（关键码和值的二元组）建立起一棵二叉排序树：

```

def build_dictBinTree(entries):
    dic = DictBinTree()
    for k, v in entries:
        dic.insert(k, v)
    return dic

```

性质分析

现在考虑上述二叉排序树实现的性质，包括其中各种操作的效率。

许多操作的情况在第 6 章讨论二叉树时已经研究过，包括遍历操作，不再赘述。这里主要考虑新操作，也就是检索、插入和删除，它们都是基于关键码的操作。

易见，检索过程是所有三个操作中共有的部分，插入和删除都需要先做检索。在插入操作中，检索完成后在实际插入结点的动作是局部的，常量时间就能完成。在删除操作中，找到要删除的结点之后，还可能再做一次检索。在上面算法里是找到被删结点的左子树中的最右结点，而后的操作只需要常量时间。显然，两段检索都在树中的一条路径上，先到达被删除结点，再继续到达其左子树的最右结点。因此，在删除操作中需要检查的结点总数不超过树中最长路径的长度。

综观这三个操作中的情况，决定操作开销的那部分动作就是沿着二叉排序树中的一条路径下行。因此，这些操作的效率依赖于被操作树的结构，每做一次比较下行一层，最大开销受囿于树中最长路径的长度，也就是二叉树的高度。

根据二叉树的性质可知，如果被操作树的结构良好，其高度与树中结点个数 n 成对数关系，检索的时间开销就是 $O(\log n)$ 。如果树结构畸形（例如第 6 章图 6.7 里的几棵树），检索关键码的时间开销也有可能达到 $O(n)$ ，这是最坏情况的时间复杂度。插入和删除操作的情况也一样。实际上，不难做出一棵情况最坏的树。例如，如果作为函数 build_dictBT 的实际参数的序列中的项按关键码递增或递减顺序排列，得到的就会是一个高度等于结点个数的二叉排序树。请读者自己验证这一事实。

人们对各种二叉树的情况做过研究，结论是 n 个结点的所有可能二叉树的平均高度是 $O(\log n)$ 。基于这个事实，可以认为在二叉排序树中做检索，平均复杂度是 $O(\log n)$ 。另一种考虑基于 n 个不同关键码的所有排列（共计 $n!$ 个）。假设把这 $n!$ 个序列送给函数 build_

`dictBT`, 可以得到 $n!$ 棵二叉排序树^Θ。算出用这 n 个关键码在每棵二叉排序树上的平均检索长度, 求出所有长度的平均值, 得到的结果也是 $O(\log n)$ 。在有关算法分析的教材和著作中可以找到有关分析的细节, 例如《Introduction to Algorithms》(高等教育出版社影印本, 265 页)。其他许多关于算法的书籍上也有。

另一方面, 二叉排序树中检索操作的空间复杂度是 $O(1)$, 操作中不需要复杂的辅助结构。插入和删除操作的情况也一样。

8.6.2 最佳二叉排序树

一般的二叉排序树有可能出现检索路径特别长的情况, 由此可以提出一个问题: 对于一组给定关键码, 最好的二叉排序树是什么样的? 是否存在一个构造算法? 要回答这个问题, 首先要定义一种评价标准。显然, 这个标准应该基于检索效率。

平均检索长度

在一棵二叉排序树中实际检索时, 可能使用树中存在的关键码, 但也可能使用树中并不存在的关键码。在考虑二叉排序树的检索效率时, 应该综合考虑这些情况, 基于所有可能情况及其出现的频繁程度, 做出某种平均检索长度, 用于评价这棵二叉排序树的优劣。基于这样的考虑, 人们给出了下面的定义。

首先, 用树中存在的关键码检索, 能找到存储着相应数据项的结点, 这种检索称为成功检索。如果用树中不存在的关键码检索, 最终将到达一个结点, 在这里基于检索关键码确定了一个前进方向 (向左或右), 但该方向不存在结点, 检索失败。成功的检索总在树中某个结点结束, 而失败检索在被检索的树上没有表示。将第 6 章介绍的扩充二叉树用在这里, 其外部结点正好表示各种失败检索的情况。图 8.8 是一个例子, 左边是一棵二叉排序树, 右边是其扩充二叉树。矩形结点表示树的外部结点, 其中用 (m, n) 的形式描述了一个开区间。易见, 用这个开区间里的值作为关键码检索, 最终将到达这个外部结点。

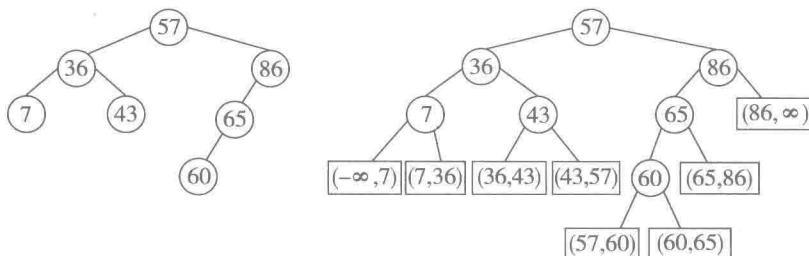


图 8.8 一棵二叉树排序树及其扩充二叉树

按照中序遍历这种扩充二叉树, 在得到的结点序列里内部和外部结点交叉排列。如果从 0 开始分别标记内部结点和外部结点, 第 i 个内部结点位于第 i 个外部结点和第 $i+1$ 个外部结点之间。称这样的结点标记序列为扩充二叉排序树的对称序列。前面说过, 对一个排序的关键码序列, 存在多棵不同的二叉排序树。不难确认, 这些二叉排序树的中缀遍历序列都是一样的, 它们的扩充二叉排序树的对称序列也完全一样。

在这种扩充二叉排序树里, 关键码的平均检索长度由下面公式给出

^Θ 注意, 不同的排列有可能得到相同的二叉排序树, 请自己检查。但这并不影响下面的讨论。

$$E(n) = \frac{1}{w} \left[\sum_{i=0}^{n-1} p_i (l_i + 1) + \sum_{i=0}^n q_i l'_i \right]$$

$$w = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^n q_i$$

其中：

- l_i 是内部结点 i 的层数， l'_i 是外部结点 i 的层数。
- p_i 是检索内部结点 i 的关键码的频度，确定一个内部结点，需要做的比较次数是结点所在的层数加一。
- q_i 是被检索关键码属于外部结点 i 代表的关键码集合的频度，检索到达一个外部结点的比较次数恰好等于该结点所在的层数。
- 这样， p_i/w 就是检索内部结点 i 的关键码的概率； q_i/w 是被检索的关键码属于外部结点 i 的关键码集合的概率。将 p_i, q_i 看作相应结点的权。

最佳二叉排序树就是使检索的平均比较次数达到最少的二叉排序树，也就是说，它应该使 $E(n)$ 的值达到最小。下面的问题是，如果给定了一组关键码，以及一组分布 p_i 和 q_i ，怎样才能构造出相应的最佳二叉排序树？

简单情况：检索概率相同

首先考虑最简单的情况，即有：

$$\frac{p_0}{w} = \frac{p_1}{w} = \dots = \frac{p_{n-1}}{w} = \frac{q_0}{w} = \frac{q_1}{w} = \frac{q_2}{w} = \dots = \frac{q_n}{w} = \frac{1}{2n+1}$$

在这种情况下，

$$E(n) = \frac{1}{2n+1} \left[\sum_{i=0}^{n-1} p_i (l_i + 1) + \sum_{i=0}^n q_i l'_i \right]$$

$$= (\text{IPL} + n + \text{EPL}) / (2n+1) = (2 \cdot \text{IPL} + 3n) / (2n+1)$$

其中

$$\text{IPL} = \sum_{k=1}^n \lfloor \log_2 k \rfloor = (n+1) \lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2$$

在 IPL 最小时这棵树达到最佳，也就是说最低的树最好。

现在考虑相应的构造算法，其基本想法就是左右子树的结点数均分。由于二叉树是递归的，构造子树可以采用与整个树同样的方法，用递归方式描述最为简单。

假设表 a 中保存的是按照关键码排序的一组字典项，算法如下：

- 0: 令 $\text{low} = 0$, $\text{high} = \text{len}(a) - 1$
- 1: 令 $m = (\text{high} + \text{low})/2$
- 2: 把 $a[m]$ 存入正在构造的二叉排序树的根结点 t , 递归地：
 - 将基于元素片段 $a[\text{low}: m-1]$ 构造的二叉排序树作为 t 的左子树。
 - 将基于元素片段 $a[m+1: \text{high}]$ 构造的二叉排序树作为 t 的右子树。
 - 片段为空时直接返回 None , 表示空树。

显然，以关键码个数作为 n ，基于已经按关键码排序的序列构造最佳二叉排序树，上述构造算法的复杂度为 $O(n)$ 。另外，从排序算法的有关研究可知，对于 n 个关键码序列，最高效

的排序算法的复杂度为 $O(n \log n)$ 。由此，如果从任意一组关键码出发，整个构造过程的复杂度将是 $O(n \log n)$ 。

下面定义一个最佳二叉排序树类，其初始化方法从给定的参数构造出一棵最佳二叉排序树。让这个类继承 `DictBinTree`，以便共享其中一些操作。

构造函数的参数可以是任何序列，但要求其中的元素可以比较大小（也就是说，对于其中元素，小于运算符“`<`”有定义），这样才能使用 Python 内置的 `sort` 函数排序，得到一个排序的表。在这个类里只定义了一个静态方法，采用递归方式定义，实现上面的算法。在这个算法的实现中，只有一点值得提出：在递归调用中需要用到原表中的片段。在下面函数里没有真做切片，而是直接传递整个表和一对表示范围的下标。这样做可以避免实际做出很多表片段的拷贝，是一种更为合适的方法^Θ。

```
class DictOptBinTree(DictBinTree):
    def __init__(self, seq):
        DictBinTree.__init__(self)
        data = sorted(seq)
        self._root = DictOptBinTree.buildOBT(data, 0,
                                              len(data)-1)

    @staticmethod
    def buildOBT(data, start, end):
        if start > end:
            return None
        mid = (end + start)//2
        left = DictOptBinTree.buildOBT(data, start, mid-1)
        right = DictOptBinTree.buildOBT(data, mid+1, end)
        return BinTNode(Assoc(*data[mid]), left, right)
```

其他操作可以继承自 `DictBinTree`。但请注意，那里的插入和删除操作只能保证二叉排序树的结构完整，保证字典的基本功能。但它们不能保证二叉排序树的最佳性质，反复做插入和删除的动态操作，通常会导致字典的性能逐渐变差。

8.6.3 一般情况的最佳二叉排序树

一般而言，对于树中的不同关键码以及非树中关键码的各个区间（外部结点），访问概率未必相同。如何处理一般的情况并构造出最佳二叉树？下面讨论这个问题。虽然从实用角度看这种构造的意义不大，因为实际应用中各种可能关键码的概率很难获得。但下面算法设计中的思想却很值得学习，这种想法被称为动态规划。

问题和性质

假设给定了递增排序的关键码序列 $key_0, key_1, \dots, key_{n-1}$ ，需要构造的二叉排序树的内部结点分别与这些关键码对应，设它们依次为 v_0, v_1, \dots, v_{n-1} ，相应的外部结点依次为 $e_0, e_1, \dots, e_{n-1}, e_n$ 。假设检索达到 v_0, v_1, \dots, v_{n-1} 而成功结束的频度分别为 p_0, p_1, \dots, p_{n-1} ，检索到达外部结点 $e_0, e_1, \dots, e_{n-1}, e_n$ 而失败结束的频度分别为 $q_0, q_1, \dots, q_{n-1}, q_n$ 。现在要求构造一棵二叉排序树使平均检索长度达到最小，也就是使下面代价函数达到最小：

$$E(0, n) = \sum_{i=0}^{n-1} p_i(l_i + 1) + \sum_{i=0}^n q_i l'_i$$

^Θ 请读者考虑，如果这个函数的每个递归都实际地做出一个切片，会带来怎样的空间开销。而按照现在的做法，算法的空间复杂度怎样？

$E(0, n)$ 是包含 n 个内部结点和 $n+1$ 个外部结点的树的带权路径长度。

最佳二叉排序树有一个重要性质：它的任何子树也是最佳的二叉排序树。这一点很容易用反证法证明。这个性质说明，一棵最佳二叉排序树是两棵较小的最佳二叉排序树的组合。如果存在多种可能的组合，都能得到包含同样内部结点（和外部结点）的二叉排序树，就应该选择其中的最佳组合。下面的算法中使用了这个想法。

首先应看到，由于存在检索频度的差异，层数最小的树未必最佳。图 8.9 是一个说明这种情况的例子，其中在内部和外部结点下面标出了关键码的频度。显然，左边二叉排序树最低，可以计算出其搜索路径长度之和为 $3 \times 1 + (2+7+2+1+4+9) \times 2 = 53$ 。对于右边的树，相应的值是 $(7+9) \times 1 + (3+4) \times 2 + (2+2+1) \times 3 = 45$ 。注意，内部结点的搜索中比较次数为其所在层数加一，外部结点的搜索中比较次数等于其所在层数。

构造方法

现在考虑如何解决问题，也就是说，如何基于给定的一组数据构造出相应的最佳二叉排序树。为了讨论方便，首先引进一些记法。下面讨论中将用 $T(i, j)$ 表示包含内部结点 v_i, \dots, v_{j-1} 和相应外部结点 e_i, \dots, e_{j-1}, e_j 的最佳二叉排序树。显见，这是扩充二叉排序树的对称序列中的一段结点，表示包含这些内部和外部结点的最佳二叉排序树。另外，再借用前面的记法 $E(0, n)$ ，用 $E(i, j)$ 表示这棵最佳二叉排序树的权值。

举例说， $T(0, 1)$ 就表示包含内部结点 v_0 和外部结点 e_0, e_1 的最佳二叉排序树；而 $T(2, 5)$ 表示包含内部结点 v_2, v_3, v_4 和外部结点 e_2, e_3, e_4, e_5 的最佳二叉排序树。

当问题比较复杂时，可能无法直接得到所需要的结果。面对这种情况下，人们提出的一种技术是逐步推进，在每步计算中根据已知的信息做一些选择，得到一些局部结果，这样积累了信息，也为下一步计算做好准备。这种工作方式被称为动态规划。第 7 章的 Dijkstra 算法就是一个典型：假设希望找到图中从顶点 v_i 到顶点 v_j 的最短路径，即使有直接的邻接边，它也未必是最短。Dijkstra 算法的工作方式是维护从顶点 v_i 到其余顶点的“已知最短路径”，并在一步计算中不断更新，直到确定了到顶点 v_j 的最短路径为止。

下面介绍的算法采用了类似的思想，从最小的最佳二叉树做起，逐步做出所需的最佳二叉树。讨论中将结合一个具体例子说明构造过程。这里以图 8.9 中的数据为例，其中关键码集合是 $\{A, B, C\}$ ，内部结点的权值分别是 $\langle 2, 3, 7 \rangle$ ，外部结点的权值分别为 $\langle 2, 1, 4, 9 \rangle$ 。显然，需要构造的最佳二叉排序树包含 3 个内部结点和 4 个外部结点。

步骤 1：首先构造出所有只包含一个内部结点的最佳二叉排序树。对每个 $i \in [0, n]$ ，内部结点只有 v_i 的二叉排序树仅有棵，它自然是最佳二叉排序树。对每个 i 做出 $T(i, i+1)$ ，计算出相应的 $E(i, i+1) = q_i + p_i + q_{i+1}$ ，就得到了 n 棵只包含一个内部结点的最佳二叉排序树。

对上面实例，3 棵只包含一个内部结点的最佳二叉排序树见图 8.10。

步骤 2：对每个 $i \in [0, n-2]$ ，构造只包含内部结点 v_i, v_{i+1} 的最佳二叉排序树 $T(i, i+2)$ 。以构造 $T(0, 2)$ 为例，这时有两种可能：①以 v_0 为新树的根，以 e_0 为左子树，以 $T(1, 2)$ 为右子树；②以 v_1 为新树的根，以 $T(0, 1)$ 为左子树，以 e_2 为右子树。为了得到最佳二叉

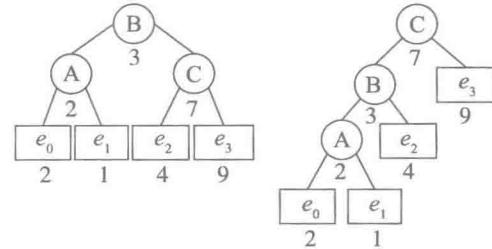


图 8.9 同样数据集的两棵二叉排序树

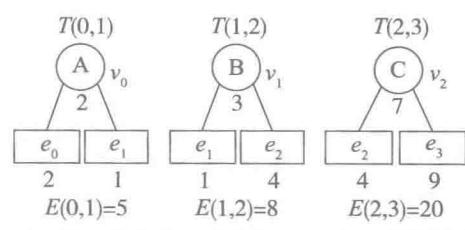


图 8.10 只包含一个内部结点的二叉排序树

排序树，需要分别算出这两棵树的带权路径长度，选出较小的一棵作为 $T(0, 2)$ 。所有 $T(i, i+2)$ 都按同样方式构造。由于前一步已构造出所有只包含一个内部结点的最佳二叉排序树，这一步不难完成。

对上面实例，首先考虑 $T(0, 2)$ ，它的两个选择见图 8.11 中 (1) 和 (2)，由于 (2) 的带权路径长度更小，为 17，以它作为 $T(0, 2)$ 。按同样方法可以做出 $T(1, 3)$ ，见图 8.11 中 (3)。

步骤 3：对每个 $i \in [0, n-3]$ ，构造出内部结点只有 v_i, v_{i+1}, v_{i+2} 的最佳二叉排序树 $T(i, i+3)$ 。以构造 $T(0, 3)$ 为例，存在 3 种可能性：①以 v_0 为新树的根，以 e_0 为左子树，以 $T(1, 3)$ 为右子树；②以 v_1 为新树的根，以 $T(0, 1)$ 为左子树，以 $T(2, 3)$ 为右子树；③以 v_2 为新树的根，以 $T(0, 3)$ 为左子树，以 e_3 为右子树。分别计算出这 3 棵树的带权路径长度，从中选出值较小的一棵作为 $T(0, 3)$ 。其他 $T(i, i+3)$ 都可以类似构造。由于前面步骤已构造出所有只包含 1 个和 2 个内部结点的最佳二叉排序树，这一步不难完成。

对于上面实例，经过这一步得到的 $T(0, 3)$ 就是最终结果。三种可能的构造情况如图 8.12 所示，比较它们的带权路径长度，选出 (3) 即为所需的最佳二叉排序树。

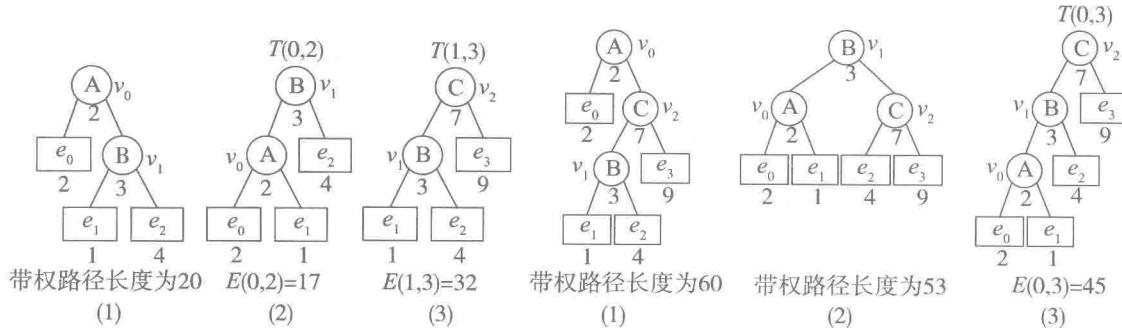


图 8.11 构造包含两个内部结点的最佳二叉排序树

图 8.12 构造包含 3 个内部结点的最佳二叉排序树

一般步骤 $m > 1$ ：根据上面讨论，可以总结出这种构造过程中一般步骤的情况。工作进展到第 m 步时 ($m \leq n$)，对每个 i ($0 < i < m$)，前面步骤已经构造出 $n-i+1$ 棵包含 i 个内部结点的最佳二叉排序树：

$$T(0, i), T(1, 1+i), \dots, T(n-i, n)$$

在第 m 步要基于它们构造出 $n-m+1$ 棵包含 m 个内部结点的最佳二叉排序树：

$$T(0, m), T(1, 1+m), \dots, T(n-m, n)$$

在构造每棵树时，都要考虑所有可能的组合构造方式，从中选出最佳的一棵树，即带权路径长度最短的那一棵二叉排序树。

下面算法的基本想法就是逐步构造出包含越来越多结点的最佳二叉排序子树，最终构造出包含所有结点的最佳二叉排序树。总结一下：

- 步骤 1：构造出所有只包含 1 个内部结点的最佳二叉排序树 $T(0, 1), T(1, 2), \dots, T(n-1, n)$ 。
- 步骤 2：基于第一步得到的结果，构造出所有只包含 2 个内部结点的最佳二叉排序树 $T(0, 2), T(1, 3), \dots, T(n-2, n)$ 。
-
- 步骤 n ：基于前 $n-1$ 步得到的结果构造出 $T(0, n)$ ，工作完成。

计算带权路径长度

在实施上面计算方法的每一步，还需要算出所有候选二叉排序树的带权路径长度。通过遍

历所有路径可以算出这个值，但显然太麻烦，计算代价太高。

实际上，在构造一棵二叉排序树时，其带权路径长度可以基于其两棵子树的带权路径长度计算出来，现在分析其中的情况。

考虑关键码为 $\text{key}_i, \text{key}_{i+1}, \dots, \text{key}_{j-1}$ 的任一段内部结点 ($0 \leq i \leq j \leq n$)，与之对应的内 / 外部结点权值的交错序列为 $q_i, p_i, q_{i+1}, \dots, p_{j-1}, q_j$ 。可以计算出这段权值交错序列的分段权值之和 $W(i, j) = p_i + p_{i+1} + \dots + p_{j-1} + q_i + q_{i+1} + \dots + q_{j-1} + q_j$ 。显然这种和值与树结构无关，可以对每对 i 和 j 事先算出留用。

在构造树 $T(i, j)$ 时，对于所有 $i < k < j - 1$ ， $T(i, k)$ 和 $T(k+1, j)$ 都已经构造好，而且它们的代价应该已知，设其分别为 $C(i, k)$ 和 $C(k+1, j)$ 。

对每个 k 构造以关键码为 key_k 的内部结点为根， $T(i, k)$ 和 $T(k+1, j)$ 为左右子树的候选二叉树。不难确认其权值为 $C_k(i, j) = W(i, j) + C(i, k) + C(k+1, j)$ ，这是因为树中所有结点都增加了一层。显然，只包含一个内部结点的最佳二叉树的权值可以直接算出，更大的最佳二叉树可以用上述方法递推算出，这样就得到了所有候选树的代价。在计算的每一步，从确定范围的所有可能构造的树中找出使 $C_k(i, j)$ 值达到最小的 k ，与之对应二叉排序树就是最佳的树 $T(i, j)$ ，其根结点是 v_k 。

上面讨论说明，新构造的最佳二叉树的代价可以直接从构造中使用的已知最佳二叉排序树的代价算出，无须重新基于新树的结构去计算：

$$C(i, j) = W(i, j) + \min\{C(i, k) + C(k+1, j) \mid i < k < j - 1\}$$

算法设计和实现

解决了所有技术问题之后，现在考虑算法的具体实现。

对于这个树构造而言，具体的关键码是什么并不重要，只有它们的排列位置和出现频度在计算中有意义。因此，算法的输入是两组描述内部 / 外部结点出现频度的参数： n 个元素的参数 w_p 给出内部结点的访问频度， $n+1$ 元的 w_q 给出外部结点的访问频度。算法的结果应当清晰地描述所需要的最佳二叉排序树。

为实现上面讨论的最佳二叉排序树构造，算法进行中需要维护一些信息。包括事先算出所有的内外结点交错的频度序列的分段权和 $W(i, j)$ （对所有的 $0 \leq i \leq j \leq n$ ，后续计算中需要使用）。算法进行中逐步构造出越来越大的最佳二叉排序子树 $T(i, j)$ ，以及计算出的代价 $C(i, j)$ ，这些都需要记录以支持后续步骤的进行。

这三组数据都通过两个指标 i 和 j 访问，一种比较方便的方式是用三个二维矩阵记录，在 Python 里用二维表实现^Θ，下面采用这种技术。在三个矩阵中：

- $r[i][j]$ 记录构造出的最佳子树 $T(i, j)$ 的根结点下标。
- $c[i][j]$ 记录子树 $T(i, j)$ 的代价。
- $w[i][j]$ 表示树中相应的内外交错结点段的权值之和。

根据下标 i 和 j 的情况， r 、 c 和 w 都应该是 $(n+1) \times (n+1)$ 的二维矩阵，算法中实际上只使用它们的上三角部分。有关情况参看图 8.13。

易见， w 主对角线元素 $w[i][i]$ 可以直接由参数得到，它们就是参数表 q 的元素。 w 上三角部分的其他元素可以通过递推方式一层层推算出来。

矩阵 r 和 c 的主对角线并不使用，算法从上副对角线开始，向右上方一层层计算出这两个

^Θ 另一种明显选择是用三个字典，以 (i, j) 序对作为关键码。有关修改留给读者完成。

矩阵上三角部分的元素。最终计算出 $r[0][n]$ 是作为结果的最佳二叉排序树的根, $c[0][n]$ 是这个树的权, 即其带权路径长度。计算过程如图 8.13 所示。

算法结束后, 根据 $r[0][n]$ 的值可以逐步提取出得到的二叉排序树。例如, 假定内部结点为 8 个, 编号 $0 \sim 7$, 计算结束后 $r[0][8]$ 的值是 4。那么就可以知道, 二叉排序树的树根是结点 4, 其左子树的根结点的编号保存在 $r[0][4]$, 其右子树的根结点的编号保存在 $r[5][8]$ 。按同样方式追溯下去, 就可以一步步得到构造出的树的完整结构。根据矩阵构造出二叉树的过程留作练习。

做好了上面的准备, 实际构造算法并不长:

```
def build_opt_btree(wp, wq):
    """ Assume wp is a list of n values representing
    weights of internal nodes, wq is a list of n+1 values
    representing weights of n+1 external nodes. This
    function builds the optimal binary searching tree from wp and wq.
    """
    num = len(wp)+1
    if len(wq) != num:
        raise ValueError(
            "Arguments of build_opt_btree are wrong.")
    w = [[0]*num for j in range(num)]
    c = [[0]*num for j in range(num)]
    r = [[0]*num for j in range(num)]
    for i in range(num):          # 计算所有的 w[i][j]
        w[i][i] = wq[i]
        for j in range(i+1, num):
            w[i][j] = w[i][j-1] + wp[j-1] + wq[j]
    for i in range(0, num-1):    # 直接设置只包含一个内部结点的树
        c[i][i+1] = w[i][i+1]
        r[i][i+1] = i

    for m in range(2, num):
        # 计算包含m个内部结点的最佳树 (n-m+1棵)
        for i in range(0, num-m):
            k0, j = i, i+m
            wmin = inf
            for k in range(i, j):
                # 在[i,j]里找使c[i][k]+c[k+1][j]最小的k
                if c[i][k] + c[k+1][j] < wmin:
                    wmin = c[i][k] + c[k+1][j]
                    k0 = k
            c[i][j] = w[i][j] + wmin
            r[i][j] = k0

    return c, r
```

算法返回两个矩阵的序对作为结果。

算法分析

这个算法比较容易分析, 其时间复杂度为 $O(n^3)$, 对于较大的 n 将非常耗时; 算法的空间复杂度为 $O(n^2)$ 。

计算具有任意权值的最佳二叉排序树有一定意义, 但更多是理论价值。它说明了即使问题如此复杂, 还是存在一般性的构造方法。这一算法在实际中使用很少。原因之一是创建这种树的代价比较高, 其二是实际中很难得到有价值的访问分布情况。

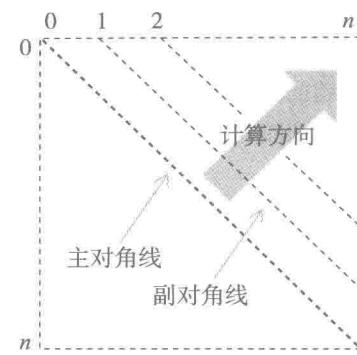


图 8.13 矩阵 $r/c/w$ 的情况

另一方面，这个算法的结构本身则非常典型。如前所述，这里采用的算法设计模式被称为动态规划。动态规划算法的特点是：在整个计算过程中，逐步建立并维持好一大批子问题的解（在这里维护的是较小的最佳二叉排序树），在每一步基于这些小的子问题的解构造出更大的子问题的解，最终构造出整个问题的解。动态规划法在算法设计中使用广泛，特别是在设计复杂的算法时非常有用。

8.7 平衡二叉树

虽然最佳二叉排序树很好，可以保证最佳的检索效率，但其缺点也很明显。构造这种结构需要掌握所有元素（关键码，包括非字典元素的关键码）的分布情况。其构造成本高只是一个缺点，最重要的是，这种树只适合作为静态字典的表示，不能很好地支持由于动态操作而产生的动态变化。“峣峣者易折，皎皎者易污”，最佳的性质最容易破坏而难以保持，在插入和删除中很难维护树结构的最佳状况。

由于“最佳”是一种全局性质，无法从局部上把握和检查，一旦破坏，也难以通过局部调整恢复。虽然有可能设计出能保持二叉排序树最佳性质的插入和删除算法，但由于一次插入和删除可能造成大范围的影响，恢复最佳结构的代价一定很高，使这种操作失去实用价值。如果对最佳二叉排序树做了一系列插入和删除，其中不进行维护，树结构就可能不断恶化，导致字典性能下降，最坏情况的检索性能可能趋向于 $O(n)$ 。

但在另一方面，实际应用非常需要既能维持高效检索，又能支持动态操作的排序树结构。在这种情况下，人们只能改变思考方向，摒弃对最佳的追求，设法在操作中维持某种“比较好”的结构。基于这种想法，人们已经开发出多种不同的排序树结构，其共性都是设法提供某些接近最佳的性质（树高与结点数成对数关系），换取动态操作中较容易进行局部维护。本节介绍的平衡二叉树即是其中一种。

使用二叉树结构支持字典操作，必须避免树结构变成某种退化形式，出现超长的检索路径。最基本的追求是有保证的 $O(\log n)$ 检索效率，保证每个检索操作都能在 $O(\log n)$ 时间内完成。这就要求二叉树结构的高度始终维持为 $O(\log n)$ ，也只要求这一点，可以允许其实际高度不超过最佳情况的某个常数倍。还要保证插入和删除动态操作的性能，使它们的实际操作能在一条路径上完成，也具有 $O(\log n)$ 复杂度。

下面主要介绍平衡二叉排序树，又称 AVL 树，由苏联的 Georgy Adelson-Velsky 和 E. M. Landis 发明，并以他们的名字命名。与之类似的结构还有红黑树、B 树等。下一节还将简单介绍 B 树一类结构的基本情况。

8.7.1 定义和性质

平衡二叉排序树的基本考虑是：如果树中每个结点的左右子树的高度差不多（“平衡”），整个树的结构也会比较好，不会出现特别长的路径。

定义 平衡二叉排序树是一类特殊的二叉排序树，它或为空树，或者其左右子树都是平衡二叉排序树（是递归结构），而且其左右子树的高度之差的绝对值不超过 1。

易见，这里的平衡是一种局部性质，有可能通过局部的信息描述。整棵树的平衡由各结点的平衡情况刻画，结点平衡可以用一个简单的平衡因子（Balance Factor, BF）描述。这里将 BF 定义为该结点的左子树的高度减去右子树的高度之差，其可能取值只有 -1、0 和 1 三种情况。应该特别注意，在平衡二叉树的结点里并不记录其左右子树的高度，只记录 BF 值。

显然，完全二叉树和等概率情况的最佳二叉排序树都是平衡二叉树。而平衡二叉树要求的条件更弱一些。图 8.14 给出了两棵平衡的二叉树(1)和(2)，以及一棵不平衡的二叉树(3)，图中结点旁所标的数值为结点的 BF 值。叶结点的平衡因子都是 0，所以省略。显然，只要有一个结点的 BF 值超标，这棵树就不是平衡二叉树了。

从图 8.14 可以看出，平衡二叉树可以比相同高度的完全图稀疏很多，但到底可能稀疏多少？换个角度，同样是 n 个结点，最差（层数最多）的平衡二叉树可能有多高？能保证 $O(\log n)$ 的高度吗？要回答这些问题，可以实际计算一下。现在考虑各种高度的“临界”平衡二叉树，也就是说，在相同高度中结点数最少的平衡二叉树。图 8.15 里给出了高度最低的几棵临界平衡二叉树。

不难看出，这些树结构是递归的。 $i > 1$ 时树 B_i 总是以 B_{i-1} 和 B_{i-2} 作为子树，再加一个根结点构成。考虑 B_i 的结点个数 $\#(B_i)$ ，有如下递推公式：

$$\begin{aligned}\#(B_0) &= 1 \\ \#(B_1) &= 2 \\ \#(B_i) &= \#(B_{i-1}) + \#(B_{i-2}) + 1\end{aligned}$$

这个公式很像斐波那契数列的递推公式：

$$\begin{aligned}F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2}\end{aligned}$$

很容易用数学归纳法证明 $\#B_i = F_{i+3} - 1$ 。由于：

$$F_i \approx \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^i \approx 0.447 \times 1.618^i$$

基于这些结果可知 n 个结点的平衡二叉树是 $O(\log n)$ 。实际上树高

$$h < \frac{3}{2} \log_2(n+1)$$

也就是说，与最佳二叉排序树相比，最长路径的长度仅差一个常量因子。

8.7.2 AVL 树类

平衡二叉排序树（AVL 树）也是二叉排序树，因此，在这种树上的检索就是在普通二叉排序树上检索，时间代价的最坏情况受限于树中最长路径的长度。上面事实说明，如果能维持平衡二叉树的结构，检索操作就能在 $O(\log n)$ 时间内完成。

剩下的问题就是如何让树结构在动态变化中维持平衡。在这里做插入和删除，不但要保持树的结构和树中结点关键码的正确排序，还需要维持树的平衡。插入和删除一定会改变树的结构，完成基本操作后树的平衡有可能被破坏。如果出现这种情况，就需要调整树结构，设法恢复平衡。要保证操作的时间代价为 $O(\log n)$ ，调整就必须是局部的。注意， $O(\log n)$ 是树中路径的长度特性。如果在完成实际结点的插入或删除后，为维持平衡只需要在一条路径上调整，

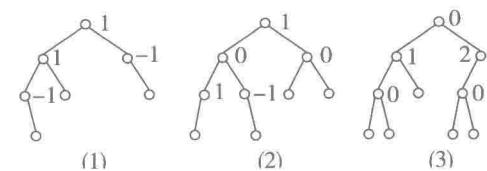


图 8.14 平衡和不平衡的二叉树

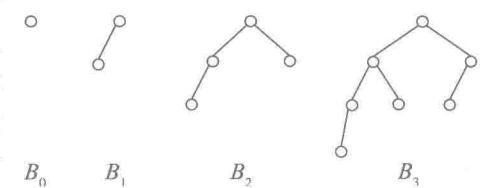


图 8.15 临界的平衡二叉树

就可能满足 $O(\log n)$ 的复杂度要求。

AVL 树的动态操作（插入 / 删除）将采用如下方式实现：首先根据关键码确定位置，实际插入或删除结点。如果这时出现树失衡的情况，就设法进行局部调整，恢复树的平衡。下面可以看到，插入和删除操作后的调整都可以在树中的一条路径上一遍完成，因此插入和删除操作的时间代价为 $O(\log n)$ 。

基本定义

为了实现 AVL 树，二叉树的每个结点里需要增加一个平衡因子记录。下面考虑创建一个 AVL 树类，主要是研究插入操作的实现，并简单讨论删除操作的问题。

首先把 AVL 树结点类定义为二叉树结点类的子类，增加一个 `bf` 域。叶结点的 `bf` 值是 0，类的初始化方法设置这个域：

```
class AVLNode(BinTNode):
    def __init__(self, data):
        BinTNode.__init__(self, data)
        self.bf = 0
```

AVL 树是一种二叉排序树，将这个类定义为 `DictBinTree` 的子类，所有不改变结构的方法都可以继承，但实现插入和删除操作的方法需要重新定义，因为现在需要考虑平衡的维护问题，这是本类的特殊要求。

与二叉排序树字典类一样，AVL 树类 `DictAVL` 的初始化方法建立一棵空树（看作空的 AVL 树），空树没有结点，自然是平衡的。如果需要，也可以给初始化方法增加一个初始化序列参数，要求基于该序列建立 AVL 树。当然，如果那样做，就需要在初始化方法里调用本类的 `insert` 方法（后面定义）把数据逐项插入树中。也可以设法直接构造。

```
class DictAVL(DictBinTree):
    def __init__(self):
        DictBinTree.__init__(self)
```

这个类的大部分方法都已经有了（通过继承），需要实现的只剩下插入和删除方法，因为需要维持平衡。下面集中讨论这个问题。

8.7.3 插入操作

AVL 树上的插入操作，前一部分等同于普通二叉排序树的插入，即首先通过检索在树中找到插入位置，而后加入新结点。这时出现了新问题：树原来是平衡的，但在插入结点后可能使树中的某个局部失衡，应考虑必要的调整。

插入后的失衡和调整

首先应该看到，如果在检索插入位置的过程中，所有途经结点的 `BF` 值均为 0，那么实际插入结点不会导致这些结点失衡，只是它们的 `BF` 值应修改为 1 或 -1。其余结点的 `BF` 值不变，整棵树也不会失衡。实际插入后更新途经结点的 `BF` 值，操作就完成了。

如果不是上面情况，那么一定存在一棵包含实际插入点的最小非平衡子树，即那棵包含新结点插入位置的、其根结点的 `BF` 非 0 的最小子树^Θ。如果插入新结点后这棵子树仍保持平衡，而且其高度不变，那么整棵二叉排序树也将保持平衡（由于该子树的高度不变，在它外面的树结点的 `BF` 值都保持不变）。进一步说，如果插入新结点后的结构调整和 `BF` 值修改都能在该子树内部的一条路径上完成，插入操作的复杂度将不超过 $O(\log n)$ 。下面将要说明，情况确实如此。

^Θ 注意这里用“非平衡”指 `BF` 非 0，与下面“失衡”不同。失衡指 `BF` 超出合法值的情况。

假设插入结点所在的最小非平衡子树的根结点为 a , 如图 8.16 中(1)所示, 其左子树较高(右子树较高的情况类似)。如果插入点在 a 的右子树(较低的子树), 插入结点后只需调整结点 a 之下直至插入点的路径上所有结点的 BF 值(根据 a 的选择, 这些结点的 BF 值原来都为 0), 并将 a 的 BF 值修改为 0。由于以 a 为根的子树高度不变, 插入和调整对其他部分没有影响, 整个树保持平衡, 插入操作圆满完成。

如果新结点应插人在 a 的左子树(较高子树), 就会破坏 a 的平衡, 如图 8.16 中(2)所示。在这种情况下必须设法恢复结点 a 的平衡。请注意, 新结点插人在 a 的较高子树, 也说明另一子树较低。从较高子树中调整结点到另一子树, 降低其高度, 就能恢复 a 点的平衡。这样调整后维持子树的高度不变, 整个插入操作对该子树之外部分的平衡没有任何影响。只要恢复操作的复杂性不超过 $O(\log n)$, 就保证了插入操作的效率。

下面考虑具体的恢复操作, 分为四种情况处理:

- LL 型调整 (a 的左子树较高, 新结点插人在 a 的左子树的左子树)。
- LR 型调整 (a 的左子树较高, 新结点插人在 a 的左子树的右子树)。
- RR 型调整 (a 的右子树较高, 新结点插人在 a 的右子树的右子树)。
- RL 型调整 (a 的右子树较高, 新结点插人在 a 的右子树的左子树)。

易见, 后两种情况与前两种情况分别对应(RR 对应 LL, RL 对应 LR), 分别插人在 a 的子树的外侧和内侧。从后面程序代码也可以看到, 两组操作完全对称。在下面讨论中只需关心以 a 为根的这棵子树, 整棵树的其余部分没有任何变化。

LL (RR) 失衡和调整: 情况如图 8.17 所示。插入操作前, a 的以 b 为根的左子树较高。由于 a 是最小非平衡子树的根, b 的平衡因子一定是 0, 其两棵子树等高。注意, 图中子树 $A/B/C$ 的高度相同, 这棵子树的对称序列是 $A\ b\ B\ a\ C$, 其中大写字母表示相应子树的对称性序列。

新结点插入图中子树 A , 导致结点 a

失衡, 如图 8.17 中(2)所示。在这种情况下做一个顺时针旋转, 调整结点 b 和 a 的位置关系: 将结点 b 作为调整后的子树的根结点, a 作为 b 的右子结点, b 原来的右子树 B 作为 a 的左子树。不难看到, 现在 a 的两棵子树同高, b 的两棵子树也同高, 且与插入前以 a 为根的子树同样高。调整完成。

易见, 调整后所得子树的对称序列是 $A'\ b\ B\ a\ C$, 其中 A' 是 A 中适当位置加入的新结点关键码。与插入前的对称序列比较, 可知调整没有破坏结点的正确顺序。

将 LL 调整实现为 AVL 树类里的一个静态方法:

```
@staticmethod
def LL(a, b):
    a.left = b.right
    b.right = a
    a.bf = b.bf = 0
```

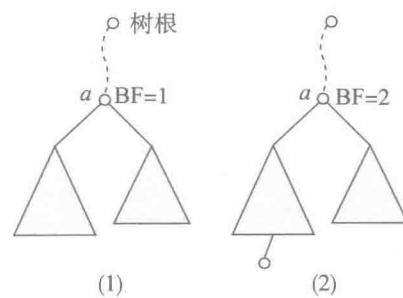


图 8.16 AVL 树插入和最小非平衡子树

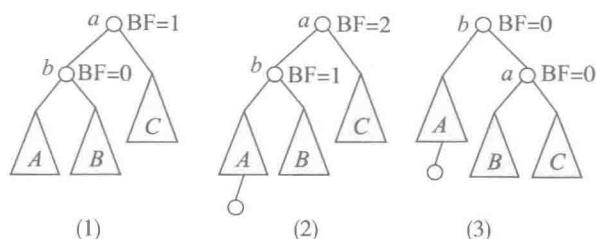


图 8.17 LL 失衡和调整恢复

```
return b
```

这里假定函数的两个参数分别是最小非平衡子树的根 a 和其左子树的根 b ，这个方法返回调整后得到的新子树的根。

与之对应的是 RR 失衡和调整，处理方式与 LL 调整完全对称。两个参数分别是最小非平衡子树的根 a 和其右子树的根 b ，方法也返回新子树的根：

```
@staticmethod
def RR(a, b):
    a.right = b.left
    b.left = a
    a.bf = b.bf = 0
    return b
```

LR (RL) 失衡和调整：情况如图 8.18 所示。插入操作前， a 的以 b 为根的左子树较高。同样，由于 a 是最小非平衡子树的根， b 的平衡因子是 0。由于新结点要插入在左子树的内侧，考虑 b 的右子结点 c 。以 c 为根的子树与 A 和 D 一样高，新结点将插入这棵子树中。还请注意，由于 a 是最小非平衡子树的根，结点 c 的平衡因子也是 0。另外，以 a 为根的子树的对称序列是 $A b B c C a D$ ，其中大写字母表示相应子树的对称性序列。

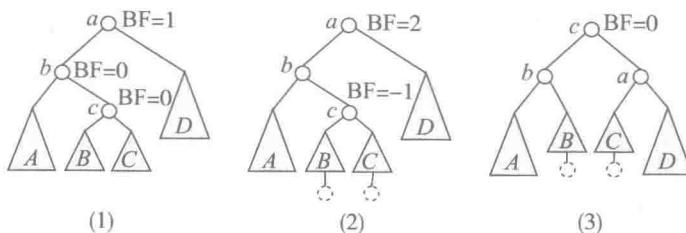


图 8.18 LR 失衡和调整恢复

新结点可能插入 c 的左子树或者右子树，插入后 a 点失衡，情况如图 8.18 中 (2) 所示。接下来的调整是把 c 提升为这棵子树的根，以 b 和 a 分别作为 c 的左右子结点。根据新结点究竟插入在原来 c 的哪棵子树， b 和 a 的平衡因子可能为 0 或者 $1/-1$ ，无论如何都不失衡，子树的新根结点 c 的平衡因子为 0。调整后子树的高度与插入前一样，对称序列为 $A b B' c C' a D$ ，无论新结点是插入 B 或者 C ，结果序列都正确排序。

下面是实现 LR 调整的静态方法。与前面 LL 调整一样，参数 a 和 b 分别为最小非平衡子树的根及其左子结点，函数返回调整后的子树的根结点。

```
@staticmethod
def LR(a, b):
    c = b.right
    a.left, b.right = c.right, c.left
    c.left, c.right = b, a
    if c.bf == 0:      # c 本身就是插入结点
        a.bf = b.bf = 0
    elif c.bf == 1:    # 新结点在c的左子树
        a.bf = -1
        b.bf = 0
    else:              # 新结点在c的右子树
        a.bf = 0
        b.bf = 1
    c.bf = 0
    return c
```

这里有一个特殊情况，就是插入前 b 为叶结点，c 就是新插入的结点。这种情况也是将 c 作为结果子树的根，b 和 a 作为其左子结点和右子结点。

RL 失衡和调整的情况与 LR 失衡是对称的，下面是实现该调整的静态方法：

```
@staticmethod
def RL(a, b):
    c = b.left
    a.right, b.left = c.left, c.right
    c.left, c.right = a, b
    if c.bf == 0:      # c 本身就是插入结点
        a.bf = 0
        b.bf = 0
    elif c.bf == 1:    # 新结点在 c 的左子树
        a.bf = 0
        b.bf = -1
    else:              # 新结点在 c 的右子树
        a.bf = 1
        b.bf = 0
    c.bf = 0
    return c
```

插入操作示例

现在用一个例子展示 AVL 树插入操作中的情况和各种调整。图 8.19 给出了对一棵 AVL 树执行一系列插入的操作过程。图中标在结点圆圈内的是关键码，标在结点旁边的是平衡因子值，叶结点的 0 平衡因子一律省略。

设该树的初始状态如图 8.19 中(0)所示，只包含两个结点。在这种状态下插入“23”，确定的最小非平衡子树根结点在(0)中用灰色标出（下同）。插入后该结点失衡状态如(1)所示。新结点插入在左子树的左边，应该做 LL 调整。插入操作完成时得到状态(2)。下面插入关键码“11”（状态(3)），没有结点失衡。再次插入“18”的新结点链接在最小非平衡结点的左子树的右边，结果是失衡状态(4)。随后插入“69”和“81”造成 RR 失衡状态(7)。做 RR 调整后恢复平衡。最后顺序插入“63”和“60”出现了失衡状态(9)。做一次 RL 调整使树恢复平衡。

注意，在图 8.19 中，出现失衡情况时只修改了最小非平衡子树根以下的平衡因子，未修改其上层结点的平衡因子（图中状态(4)、(7)、(9)都有这种情况），因为随后的调整恢复了子树的高度，其他平衡因子不会改变。

总结和分析

总结一下前面的讨论：插入新结点后出现失衡的情况只有上面几种，经过局部的旋转调整都可以恢复平衡。具体做法是先插入新结点，发现失衡后只需要在最小不平衡子树的根结点附近做局部调整，就可以把该子树根的平衡因子变成 0，而且保证这棵子树的高度与插入前相同。另外，所做的调整不会改变树中数据的排序序列。

由于插入结点并完成必要调整后，这棵子树与插入前在这个位置的子树高度相同，其结构变化（加入新结点以及其后的可能调整）对子树之外的其他部分毫无影响。因此，整棵二叉树中其他结点的平衡因子都不需要修改。随着这棵子树恢复平衡，整棵二叉排序树也恢复了平衡。

插入算法的操作分成几个阶段：首先找到插入位置并实际插入新结点，然后可能需要修改一些结点的平衡因子，发现失衡时做些局部调整。这里所有的操作都是在树中的一条路径上进行的，所以 AVL 树插入操作的代价为 $O(\log n)$ 。

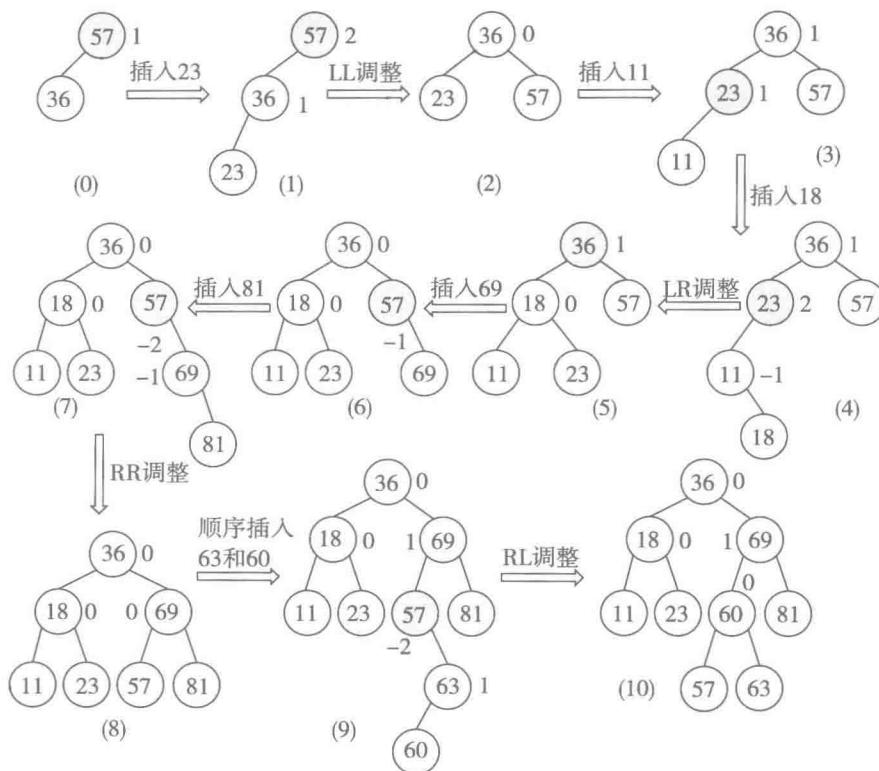


图 8.19 AVL 树上的一系列插入和调整

插入操作的实现

现在考虑插入操作的具体实现，其操作过程是：

1) 查找新结点的插入位置，并在查找过程中记录遇到的最小不平衡子树的根：

- 用一个变量 a 记录距插入位置最近的平衡因子非 0 的结点，由于可能需要修改这棵子树，在此过程中用另一变量 pa 记录 a 的父结点。
- 如果不存在这种结点，需要考虑的 a 就是树根。
- 如果在新结点插入后出现失衡， a 就是失衡位置。
- 实际插入新结点。

2) 修改从 a 的子结点到新结点的路径上各结点的平衡因子：

- 由于 a 的定义，这段结点原来都有 $BF = 0$ 。
- 插入后用一个扫描变量 p 从 a 的子结点开始遍历，如果新结点插入在 p 的左子树，就把 p 的平衡因子改为 1，否则将其改为 -1。

3) 检查以 a 为根的子树是否失衡，失衡时做调整：

- 如果 $a.BF == 0$ ，插入后不会失衡，简单修改平衡因子并结束。
- 如果 $a.BF == 1$ 而且新结点插入其左子树，就出现了失衡：
- 新结点在 a 的左子结点的左子树时做 LL 调整。
- 新结点在 a 的左子结点的右子树时用 LR 调整。
- 如果 $a.BF == -1$ 而且新结点在其右子树则出现失衡：
- 新结点在 a 的右子结点的右子树时用 RR 调整。

- 新结点在 a 的右子结点的左子树时用 RL 调整。

4) 连接好调整后的子树，它可能应该作为整棵树的根，或作为 a 原来的父结点的相应方向的子结点（左子结点或右子结点）。

下面函数实现了这一过程：

```

def insert(self, key, value):
    a = p = self._root
    if a is None:
        self._root = AVLNode(Assoc(key, value))
        return
    pa = q = None
    while p is not None:
        if key == p.data.key:
            p.data.value = value
            return
        if p.bf != 0:
            pa, a = q, p
            q = p
            if key < p.data.key:
                p = p.left
            else:
                p = p.right
        # q是插入点的父结点, pa, a记录最小非平衡子树
        node = AVLNode(Assoc(key, value))
        if key < q.data.key:
            q.left = node
            # 作为左子结点
        else:
            q.right = node
            # 或右子结点
        # 新结点已插入, a 是最小不平衡子树
        if key < a.data.key:
            p = b = a.left
            d = 1
        else:
            p = b = a.right
            d = -1
        # 修改b到新结点路径上各结点的BF值, b为a的子结点
        while p != node:
            if key < p.data.key:
                p.bf = 1
                p = p.left
            else:
                p.bf = -1
                p = p.right
            # p的右子树增高
        if a.bf == 0:
            a.bf = d
            return
        if a.bf == -d:
            a.bf = 0
            return
        # 新结点在较高子树, 失衡, 必须调整
        if d == 1:
            if b.bf == 1:
                b = DictAVL.LL(a, b)
            else:
                b = DictAVL.LR(a, b)
        else:
            if b.bf == -1:
                b = DictAVL.RR(a, b)
            else:
                b = DictAVL.RL(a, b)
        # 新结点在a的左子树
        # LL调整
        # LR调整
        # 新结点在 a 的右子树
        # RR调整
        # RL调整
    
```

```

if pa is None:                                # 原a为树根, 修改 _root
    self._root = b
else:
    if pa.left == a:
        pa.left = b
    else:
        pa.right = b

```

8.7.4 相关问题

最后讨论一些相关问题。

插入操作的复杂度

可以确定插入操作的时间复杂度：插入结点的最大时间开销受限于树的深度，由此是 $O(\log n)$ 。在检索插入位置的同时也找到了最小不平衡子树。在最小不平衡子树里修改平衡因子的时间也不超过树的高度，也是 $O(\log n)$ 。四种调整都是在最小非平衡子树的根附近修改几个结点关系，是 $O(1)$ 操作。因此插入算法的时间复杂度是 $O(\log n)$ 。

删除操作

AVL 树中关键码（数据）删除算法的基本实现方式与插入操作类似，也是先确定结点并删除，而后调整结构恢复。一种可能算法分几步完成：

- 检索需要删除的结点。
- 把删除任意结点的问题变成删除某棵子树的最右结点的问题，为此只需找到被删结点左子树的最右结点并交换两个结点的位置（或交换关键码和数据）。
- 实际删除结点（就是二叉排序树的删除）。
- 如果出现失衡就调整树结构，恢复平衡。

删除时的调整会遇到新的困难。插入中的调整只涉及一个结点周围的结构结点，而在删除时，有可能需要调整一系列结点的结构关系。但无论如何，这些调整都是在树中的一条路径上进行的，因此操作复杂度与路径长度呈线性关系，复杂度也是 $O(\log n)$ 。

综合 AVL 树各种操作的情况，可知这种结构能较好支持动态字典。

几种二叉排序树的对比

现在对本章讨论的几种基于二叉排序树结构做一点总结：

- 简单二叉排序树能支持字典操作，其检索操作的平均效率高，插入和删除操作的实现比较简单，平均操作效率也是 $O(\log n)$ 。但这种结构有致命缺点，就是操作的高效率并没有保证。这种树的结构是在插入和删除的动态操作中自然形成的，没有任何控制，有可能出现结构退化的情况，导致各种操作都非常低效。
- 最佳二叉排序树构造比较费时，能保证最高的检索效率。但如果需要做数据的插入或删除操作，操作后维持其最佳性的代价太大，因此不适用于动态字典。
- 平衡二叉排序树（AVL 树）的检索效率与最佳二叉排序树处于同样数量级，其主要优点是插入和删除后的维护（失衡后的恢复）可以在局部完成，因此插入和删除操作复杂度都不超过 $O(\log n)$ 。用于实现动态字典，在长期运行和反复动态修改中，可以始终保证 $O(\log n)$ 的操作复杂性。这种结构的缺点就是操作的实现比较复杂。

由于这些情况，在内存里使用的动态字典也经常采用平衡二叉排序树或其他性质类似的结构实现作为存储结构。这样的字典操作效率有确定性的保证。与之对应，散列表的效率只有概

率性的保证。当然，要采用二叉排序树一类结构，还要求数据（或关键码）上有一种合乎需要的序关系。

由于 AVL 树的插入和删除操作比较复杂，人们一直希望找到性质与之类似但操作更为简单局部的其他树形结构，这方面的研究有一些成果，如红黑树等，它们也能支持动态字典的实现。有关情况请读者参考其他著作，这里不进一步讨论。

8.8 动态多分支排序树

二叉排序树类的结构以二叉树为基础，主要技术是控制不同子树的高度差，保证全树高度与所存数据项数之间的对数关系，从而保证了检索效率。只要动态操作能限制在树中一条路径上进行，就能高效完成结构的变化。

其他树形结构的性质与二叉树类似，只要树的结构良好，最长路径的长度与树中结点个数同样具有对数关系，就可能作为字典的实现基础。

8.8.1 多分支排序树

现在考虑用一般树结构实现字典。为了保证高效率的检索，必须采用某种排序树结构，使检索能沿着树中路径进行。这里还有一个问题：在一般树结构中结点的度数没有限制，不利于计算机实现。为方便实现，最好采用统一规模的结点，而一旦确定了结点规模，也就决定了树结点的最大分支数。具体采用怎样的结点度数上限，可以根据理论分析和实际需要确定。下面讨论将提供一些线索，首先假定这个上限大于 2。

在多分支排序树中，一个结点里可能存储多个关键码，需要维持这些关键码和相应子树关键码的排序关系。图 8.20 描绘了一个包含四棵子树的根结点，为保证检索效率，结点里的关键码必须能为检索导航。图中结点包含 3 个关键码，它们将可能关键码空间划分为 4 段，各子树里的关键码应分别在这 4 段中取值。这样，比较检索关键码与结点里的关键码，就可以决定进入哪棵子树继续检索。

从这个示意图中还可以看到另一些情况：这里一个结点保存着多个关键码，要为进入子树导航，关键码应该排序存放。另外，从全局的角度看，树中检索同样沿着从根开始的一条路径进行，但其中在每个结点还需要做一次顺序表检索，找出正确的下行分支。如果一个结点里的关键码很多，可以考虑采用二分法检索。最后，关键码的插入和删除也有许多麻烦。由于这里的情况比较复杂，设计具体多分支树时，必然存在很多可能的变化。

控制多分支排序树结构的一种重要技术是控制结点的分支数，限制它们只能在一定的范围内变动。如前所述，多分支排序树通常采用统一大小的结点，例如确定结点最大分支数为 $m > 2$ 。为保证结点中存储空间的利用率，还需要规定结点的最小分支数 $m' \geq 2$ 。允许具体结点的分支数（子树个数）在这两个值的范围内变化。

采用多分支排序树实现字典，同样需要控制最长路径的长度。二叉树中分支结点的度数只能是 1 或 2。现在的树结点允许更多分支，灵活性更强，存在更多控制树中路径长度的方法。然而，从图 6.7 中几棵二叉树的情况可知，仅仅控制结点的分支数不足以保证理想的树结构，还需要其他控制技术。

与二叉排序树相比，调整多分支排序树结构的手段更多。例如：

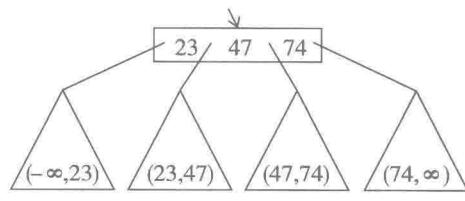


图 8.20 一个 4 分支结点的关键码情况

- 插入新数据项时未必需要建立新结点。如果检索最后找到的叶结点不满（其关键码少于上限 m ），可以直接把新数据项存入这里。
- 如果查找到结点 p 需要存入数据项，但是 p 已满，也可能不加入新结点，而是考察能否把 p 中的一些数据项调整到内容较少的兄弟结点。如果可行，还需调整 p 与其兄弟结点的导航关键码（在 p 的父结点里），操作完成后这棵树的结构没变。
- 如果 p 和其兄弟结点的关键码都已满，无法调整。采用下面策略有可能避免树的高度增加（最长路径不变长）：考虑 p 的父结点，如果其中的导航关键码不满，可以为 p 增加一个兄弟结点，把应该存入 p 中的一部分数据放入新结点，并给 p 的父结点增加一个导航关键码。这样做虽然树结构变了，但没有增高。

上面说的是插入操作。删除字典中数据项的情况与之对应，从结点删除关键码，可能导致剩下的关键码太少，这时可以考虑在兄弟结点之间调整关键码，或者考虑合并结点。

基于这些分析和想法，人们开发了一些多分树结构，它们也被广泛用于实现字典，特别是存储于外存结构的大型字典。这些结构的共性特征是在动态操作中，始终保持从根到所有叶节点的路径长度完全相同。在这种情况下，只要保证每个分支结点至少有两个分支，在 n 个结点的树里，路径长度就不会超过 $O(\log n)$ ，因此可以保证检索的效率。

实际的多分支排序树都是基于这些基本考虑，设计了一套保证结构良好的技术。由于树中最长路径的长度相同，树中数据根据关键码顺序存储，检索效率高。下面先考虑一种称为 B 树的结构。

8.8.2 B 树

B 树是一种动态的多分支排序树，采用前面讨论的技术，通过维护结点分支数保证树具有良好结构。B 树常被用于实现大型数据库（也可以看作字典）的索引。数据库记录（数据项）保存在一批外存区块里。外存（例如磁盘等）的特点是一个存储块的存储量比较大，可以保存很多数据项。另外，内外存数据交换通常以外存数据块为单位进行。由于这些情况，B 树的结点也用外存存储块大小的块表示，其中记录的是关键码到数据（记录）存储位置的索引，一个结点（存储块）里可能保存很多索引。

例子和定义

图 8.21 描绘了一个字典的逻辑结构，其中用一棵 4 阶 B 树（阶表示树中结点的最大分支数，见下面定义）作为索引，树中每个分支结点至多有 4 个分支。B 树中的叶结点（图中用小圆圈表示）实际上并不存在（其中并不保存信息，因此不需要显式表示）。图中最下面顺序链接的矩形表示保存实际数据的存储块。可以看到树中结点的度数差异。

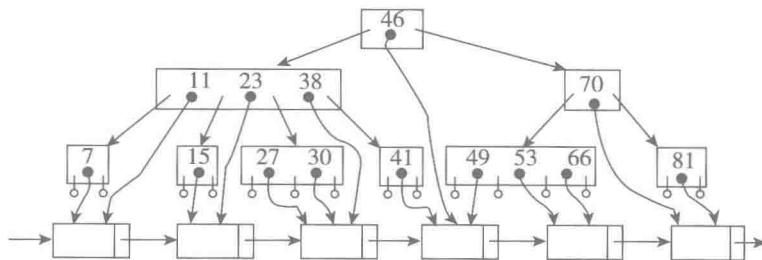


图 8.21 一棵 4 阶 B 树及相关的数据存储块

定义（B 树） 一棵 m 阶 B 树或者为空，或者具有下面特征：

- 树中分支结点至多有 $m-1$ 个排序存放的关键码。根结点至少有一个关键码，其他结点至少有 $\lfloor (m-1)/2 \rfloor$ 个关键码。所有叶结点位于同一层，仅用于表示检索失败，实际上不需要表示（例如，可以用空引用表示）。
- 如果一个分支结点有 j 个关键码，它就有 $j+1$ 棵子树，这一结点中保存的是一个序列 $\langle p_0, k_0, p_1, k_1, \dots, p_{j-1}, k_{j-1}, p_j \rangle$ ，其中 k_i 为关键码， p_i 为子结点引用，而且 k_i 大于 p_i 所引子树里的所有关键码，小于 p_i 所引子树里的所有关键码。

操作

现在考虑 B 树的几个主要操作：检索、插入和删除。

检索。基于关键码从根出发，在遇到的分支结点里检索关键码，或者找到而成功；或者确定了一棵可能存在被检索关键码的子树并转入该子树继续检索，这样下去直至成功；或者检索到达叶结点时确定检索失败。

插入新数据。基于关键码找到应该插入数据的位置 p （检索达到叶结点，实际插入的位置 p 是该叶结点的索引所在的结点）。

- 如果结点 p 的数据项少于 $m-1$ 项，就直接在这里按序插入。
- 否则分裂结点 p ，把 p 中关键码和新关键码中较大的一半放入新建的另一个结点，居中的关键码插入 p 的父结点里相应位置。
- 结点分裂时将关键码插入父结点，也是关键码插入操作。如果该父结点当时已经有 $m-1$ 个关键码，就需要分裂它并导致将某个关键码插入其更上一层结点。这种分裂可能传播到根结点，根结点的分裂导致这棵树升高一层。

删除数据。基于关键码找到删除项所在的结点 p ，具体删除分几种情况。

- 如果 p 是最下层结点：
 - 如果结点其中关键码多于 $(m-1)/2$ 个，直接删除并结束。
 - 如果结点 p 中关键码个数不足，首先考虑能否由其兄弟结点调整一些关键码过来。如果能行，可以考虑平均分配两个结点的关键码，并正确设置位于 p 的父结点里的分隔关键码。
 - 如果 p 及其兄弟结点的关键码过少无法调整，就将 p 与其一个兄弟结点合并，这时位于其父结点的分隔关键码也要拿到合并后的结点里。合并动作也导致 p 的父结点里减少一个关键码（相当于删除），这也可能导致父结点与其兄弟结点之间的关键码 / 数据调整，或者结点合并。
 - 这种结点合并可能一层层传播，可能一直传播到根结点。如果当时根仅有两个子结点而且它们需要合并，就会使整棵树降低一层。
- 如果需要删除的关键码（和数据）在上层的分支结点，就先找到其左子树的最右关键码（该关键码一定在最下层结点，设其为 r ），把这个关键码（和数据）复制到被删除关键码的位置。随后的工作就像是从 r 原来的位置删除一样，同样有上面说的各种情况：直接删除、兄弟间调整或结点合并。

图 8.22 描绘了在一棵 4 阶 B 树中做一系列插入和删除操作，导致的树结构变化过程。从中可以看到结点的分裂、合并、合并的传播，以及树高度下降的诸多情况。

总结 B 树的设计原则：

- 1) 保持树形结构和结点中的关键码有序，用分支结点的关键码作为相应子树关键码的区分关键码，保证检索操作能正确进行。

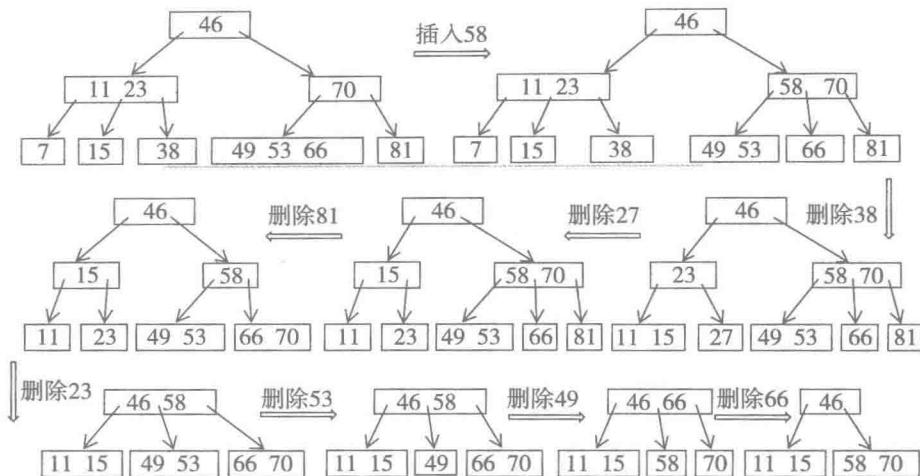


图 8.22 4 阶 B 树的一系列插入和删除

2) 保证树中从根到所有叶结点的路径等长，并保证分支结点中关键码的个数在确定的最大范围内变化，并因此保证树结构良好。

在上述原则的基础上，具体实现也允许一些变化。例如，插入关键码遇到结点满，可以考虑兄弟结点之间的关键码调整等。有关情况请读者考虑。

上面讨论已清晰地给出了 B 树结构的设计及其问题，具体实现留给读者考虑。

B 树的应用

B 树的特点是结点内检索和结点间检索的结合，采用结点分裂和合并的技术控制树高，保证到所有叶结点的路径长度相同。由于它可以支持任意大小（任意度数）的结点，B 树的组织方式很适合作为外存字典的基本结构。

外存储器（如磁盘等）通常被硬件和操作系统划分为一大批较大存储块，例如每个块的大小为 2048 或 4096 字节。外存设备的特点是查找一个存储块的速度较慢，而连续读取一个块的速度较快。根据这种情况，在实现外存字典时，人们通常用一个存储块表示 B 树的一个结点，程序采用一次一块（即一次一个结点）的方式与外存交换信息。由于一个块里可能保存几百个关键码，包含百万个结点的 B 树也不过很少几层。一次检索（或插入、删除）只需访问几个存储块，因此能达到较高的效率。常用的数据库系统，通常都是基于 B 树或其他类似结构实现的，例如下面要简单介绍的 B+ 树。

另一方面，内存字典则主要采用散列表技术或二叉排序树结构实现，B 树一类的结构很少被用在这里。

8.8.3 B+ 树

B+ 树是另一种与 B 树类似的结构，但其概念和实现稍微简单一些，在实际中使用得比 B 树更多。首先看其定义。

定义 一棵 m 阶的 B+ 树或者为空，或是满足下面条件的树：

- 树中每个分支结点至多有 m 棵子树，除根结点外的分支结点至少有 $\lfloor m/2 \rfloor$ 棵子树。如果根结点不是叶结点，至少有两棵子树。
- 关键码在结点里排序存放。分支结点里的每一个关键码关联着一棵子树，这个关键码

等于其所关联子树的根结点里的最大关键码。叶结点里的每个关键码关联着一个数据项的存储位置，数据项另行存储。

注意 B+ 树与 B 树的不同。在这里，分支结点的关键码不是子树的区分关键码，而可以看作子树的索引关键码。另外，分支结点的关键码并不关联数据项，只有叶结点的关键码关联数据项。因此，一个叶结点可以看作一个基本索引块，其中每个关键码对应一项数据的索引，而分支结点可以看作索引的索引。整个 B+ 树形成一个分层索引结构。图 8.23 给出了一棵 4 阶 B+ 树。

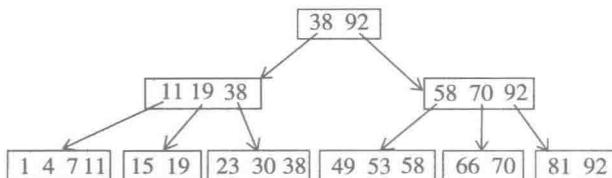


图 8.23 一棵 4 阶 B+ 树

B+ 树的操作与 B 树类似。检索操作通过检查分支结点的关键码确定进入哪棵子树，直至到达叶结点时找到了数据项，或者确定被检索的关键码不存在。插入和删除中同样采用结点的分裂 / 合并，以及兄弟结点之间的关键码移动方式处理各种情况，维护树结构良好。

B+ 树的操作比 B 树简单，因此使用更广泛。由于 B+ 树的许多情况与 B 树类似，操作的原则也类似，这里不继续讨论其细节了。

B 树和 B+ 树都通过允许结点分支在一定范围内变化的方式维护树结构，能支持动态操作。它们通常都被用于实现外存字典。

本章总结

字典是计算机软件系统中最常用的一类结构，实现数据的存储和基于关键码的检索。许多数据结构技术都可以用于实现字典，常用的技术是线性表、散列技术和树形结构。采用不同技术实现的字典具有不同的性质，适合不同的字典规模和使用方式，具有不同的操作复杂性。这方面的情况在实践中非常重要。

线性表结构最简单，很容易实现，缺点是无法避免高代价的操作。如果表中元素随意存放，检索关键码的代价与表中元素个数成正比，常常不能满足实际需要。一个简单改进是按关键码的顺序存储元素，这样就能使用二分法检索，大大提高检索的效率。但这时的插入和删除操作都具有线性复杂度。

散列表技术在实际中被广泛使用，Python 的内置 dict 和 set 等类型就是基于散列表技术实现的。散列表的基础是一个线性表（或数组），通过一个映射（散列函数）从关键码得到一个整数，作为元素在表中的存储位置。在情况良好时，散列表可以得到近乎常量的操作复杂度，包括检索、插入和删除。但采用散列技术必然会出现冲突，需要考虑冲突解决方法。内消解技术包括线性探查和双散列等，外消解技术如溢出区和拉链法（桶散列），这些都使散列表的实现变得更复杂，执行中也要耗费时间。此外，维持元素存储区的低存储密度可以降低出现冲突的概率，这是明显的用空间交换时间。散列表的主要缺点有两方面，其一是其操作效率等性质都是概率的，无法绝对避免较为低效或非常低效的具体操作。另一问题是遍历字典中数据元素时，遍历的顺序由实现和实际情况决定。

人们基于树形结构开发出很多不同的字典实现技术。基于二叉树的字典结构都源于二叉排

序树，在结点里存储数据，并保证单调的对称遍历序列。在这种树上检索操作的平均复杂度为 $O(\log n)$ 。但是，基本二叉排序树不能保证树结构良好，插入和删除总会改变树的结构，有可能导致树的结构出现退化情况，最坏情况下检索操作就可能需要 $O(n)$ 时间。基于一组给定的分布情况，存在构造出结构最好的二叉树（称为最佳二叉排序树）的算法。但这种最佳结构只能静态构造，难以在动态操作中维护。

人们开发了一些能支持动态操作，而且比较容易维护良好结构的二叉树结构，主要有 AVL 树（平衡二叉排序树）和红黑树等。它们的基本想法都是放弃最佳要求，定义一套可以局部检查的性质，保证满足该性质的二叉树都是结构良好的（树高度为 $O(\log n)$ ）。进一步说，在动态操作中只需局部调整就能维持有关的性质。这样就能保证检索 / 插入 / 删除都具有 $O(\log n)$ 最坏情况复杂度，从而适合用于实现内存字典。这些结构的缺点是实现比较复杂，而且要求关键码集合存在某种可用的序。

基于一般的树结构，人们开发了一些多分支排序树结构。一棵这种树中的结点度数不能大于某个给定常整数 m ，但也允许变化，通过限制结点的最小度数（大于等于 2）和其他技术控制路径长度。常用的 B 树和 B+ 树都属于多分支排序树，它们的一个结点允许很多分支（通常远大于 2），树中所有叶结点位于同一层，从根到它们的路径等长，从而保证了检索操作的效率。在插入和删除操作中保持树结构良好的技术包括调整兄弟结点之间的关键码，以及结点分裂 / 合并等。这些树结构适用于很大的结点，可以将结点映射到外存的存储单位，能很好发挥外存储器的优势，规避其劣势。由于这些情况，B 树和 B+ 树被广泛用于实现外存字典，特别是用在各种数据库管理系统里。

练习

一般练习

- 复习下面概念：数据存储和访问，检索（查找），检索码（关键码），关联数据，字典（查找表，映射，关联表），静态字典，动态字典，平均检索长度，索引，关联，有序集合，二分法检索，判定树，散列表（哈希表），散列，散列函数（哈希函数，杂凑函数），冲突（碰撞），同义词，负载因子，数字分析法，折叠法，中平方法，除余法，基数转换法，内消解技术，外消解技术，开地址法，探查序列，线性探查，双散列探查，溢出区方法，桶散列，拉链法，集合和元素，属于关系，全集，外延表示，内涵表示（集合描述式），基数，空集，并集，交集，补集，排序线性表，位向量，二叉排序树，最佳二叉排序树，动态规划，带权路径长度，平衡二叉排序树（AVL 树），平衡因子，失衡和调整，多分支排序树，B 树，B+ 树。
- 采用简单线性表实现集合时，可以考虑保持或者不保持元素的唯一性。请分析采用这两种不同技术时各主要操作的效率，以及存储占用方面的性质。
- 假设公共超集 $U = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n\}$ ，请用位向量表示：
 - $s_1 = \{a, d, f, g, h, i, m\}$ ， $s_2 = \{b, c, g, h, i, k, m, n\}$ ， $s_3 = \{a, b, d, f, g, h, n\}$
 - $U - s_1$ ， $s_1 \cup s_2$ ， $s_1 \cap s_2$ ， $s_1 \cup s_2 - s_3$ ， $s_1 \cup s_2 \cup s_3$
- 设散列表字典中包含 17 个存储位置，用 $k \% 17$ 作为散列函数。请首先给定一组 12 个关键码的序列，采用下面冲突消解技术，画出将全部关键码存入后的状态：
 - 采用线性探查法消解冲突；

- b) 采用另一函数 $k \% 5 + 2$ 的双散列技术消解冲突。
- c) 将整数 1、2、3 按不同顺序插入一棵空的二叉排序树，总共可以产生多少棵不同的二叉排序树？请画出这些二叉排序树。
6. 图 8.24 中是一棵二叉排序树，假定存入树中的关键码是 1 到 9，请确定每个关键码的存储位置。
7. 将序列 46、78、35、99、70、48、121、10、66、54、26 依次插入一棵开始为空的二叉排序树。请画出全部插入完成时二叉树的情况，并求出在关键码检索概率相等的情况下，所有成功检索情况的平均长度。
8. 在前一题构造出的二叉排序树中删除关键码 78、66、46、70、121，请画出每一步删除后二叉排序树的情况。
9. 哪些关键码插入序列可以生成图 8.25 中的二叉排序树？请列出所有满足条件的序列。
10. 给定关键码集合 {Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec}，采用字典序。请完成下面工作：
- 将上述关键码顺序插入空二叉排序树，画出结果，求出其平均检索长度；
 - 将上述关键码顺序插入空 AVL 树，画出结果，求出其平均检索长度。
11. 设有关键码 a、b、c、d、e，且有 $p_0=p_1=2$, $p_2=p_4=3$, $p_3=1$, $q_0=q_3=1$, $q_1=q_2=q_4=3$, $q_5=7$ 。请做出相应的最佳二叉排序树。
12. 设有关键码 1, 2, …, 20, 请画出一棵 4 阶 B 树，其中包含这些关键码。再请画出一棵 4 阶 B+ 树，其中包含这些关键码。
13. 请设计一个算法，在给定的二叉排序树里找到两个不同关键码所在结点的最小公共祖先，也就是距离这两个结点最近的公共祖先结点。

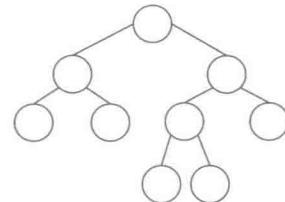


图 8.24

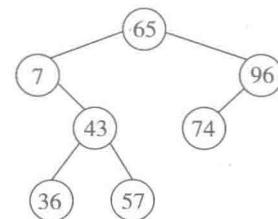


图 8.25

编程练习

- 请基于简单顺序表定义一个字典类，实现各种操作。
- 请基于第 1 题定义的类，派生出一个采用二分法检索的字典类。首先弄清需要重新定义哪些操作，而后在派生类里定义它们。
- 请基于简单线性表定义一个集合类，实现各种操作。
- 请基于排序线性表的概念定义一个集合类。可以考虑从第 3 题定义的类派生，或者完全重新定义，实现重要的集合运算。
- 请基于散列表结构实现一个集合类，定义重要的集合操作。
- 请实现一个二叉排序树字典，它允许关键码重复的项。如果插入时遇到同样的关键码，这种字典另行记录这个关键码和它关联的新数据。请考虑这种情况对整个字典设计的影响，以及在这种情况下的检索和删除操作。首先做出一个合理设计，并给出充分理由说明设计的合理性，而后实现这个字典类。
- 修改 8.6.1 节定义的二叉排序树字典类，使其初始化方法可以接受一个二叉树结点（及其关联子树）作为参数。为保证字典类正确工作，初始化方法需要检查所给参数及其子树是否构成合法的二叉排序树结构。请完成这个初始化函数。

扩充上述的初始化方法，使之也可以接受一个表（其元素是关键码 / 数据的关联）作为参数。初始化方法将把表中的项逐个插入排序树中。

8. 请给 8.6.1 节的二叉排序树字典类增加一个操作 `merge(self, another)`，它把另一棵二叉排序树 `another` 的结点并入本排序树。采用递归方式比较容易定义好这一操作。希望操作的时间复杂性是 $O(n+m)$ ，其中 n 和 m 是两棵树中的结点个数。注意，在归并后应该把 `another` 树里的根指针置空。
9. 8.6.1 节提出了一种二叉排序树的删除算法。请考虑与之不同的删除算法，要求能正确删除数据项，而且保证在任何二叉排序树上执行删除操作，树高度都不会增加。请实现所提出的算法，并从效率、删除效果等方面比较两个算法。
10. 8.6.3 节提出可以用三个字典取代构造最佳二叉排序树时使用的三个矩阵。请基于字典重新实现最佳二叉排序树的构造算法。
11. 8.6.3 节的最佳二叉树构造算法 `build_opt_btree` 完成后返回一对矩阵（两个二维的表）。请另外定义一个函数，其参数是内部结点和外部结点的权值表。它先调用上述函数构造出描述最佳二叉排序树的两个矩阵，然后根据矩阵中的信息，使用 `BinTNode` 结点对象构造出相应的二叉排序树。
12. 请收集你所在班级（或年级）的同学姓名，以此作为关键码设计一个散列字典（用一个固定大小的 `list` 作为基本存储）。由于姓名用的是字符串，可以应用 Python 标准函数 `hash` 得到一个散列值，再通过取模将函数值归结到字典下标范围内。冲突消解采用双散列的方式，同样对姓名的 `hash` 值做散列。把字典的大小作为参数，以便统计负载因子从 0.5 变到 1.0 的不同情况下，在所有姓名插入字典过程中出现冲突的次数。

第9章 排序

与前面几章的内容不同，本章不讨论任何数据结构，而是集中关注一个重要的基本计算问题和一组解决这一问题的重要而有趣的算法。

9.1 问题和性质

排序（sorting）就是整理数据的序列，使其中元素按照特定顺序排列的操作。在排序的过程中，序列里的数据元素保持不变，但其排列顺序可能改变。

在现实生活中，排序也是一种非常有意义的操作，人们在日常生活中经常做各种事物的排序工作，在整理各种材料或者物品时，所做的事情经常就是把它们按照某种需要的性质排序。例如整理书籍、各种文件和表格、货架上的商品等。

排序也是计算中最重要、最经常需要做的一项工作，在许多算法和实际系统里都需要做各种各样的排序。有人做过这方面的研究，得到的结果说明，在计算机数据处理中很大比例的工作是做数据的某种排序。

排序可以使数据的存储方式更具结构性，有利于对它们处理。对于一些信息处理工作，排好序的数据更容易使用。排序也是许多算法的重要组成部分，在许多算法里需要对计算中使用的数据进行排序。前面已经看到一些例子，例如：

- 排好序的序列可以采用二分法查找（效率高）。
- Kruskal 算法中需要对图中的边按权值排序。
- 最佳二叉树生成算法需要所用的数据序列按关键码排序。

9.1.1 问题定义

一次具体排序，总是针对某数据集合 S 的元素组成的序列进行的，基于集合 S 上的某种具体的序关系。一类常见的序关系是该集合上的全序。

集合上的序

集合 S 上的一个全序关系，记为 \leq ，是集合 S 上的一种自反、传递和反对称的关系。对 S 中的任一对元素 e 和 e' ，必有 $e \leq e'$ 或者 $e' \leq e$ 成立。例如，整数集合上的小于等于关系、字符串的字典序等，都是典型的全序关系。

不同排序工作也有许多可能的变化情况。例如，如果排序所基于的序是一种简单全序，也就是说，两个元素按序相等当且仅当它们是同一个元素（反对称关系），那么一组具体的数据元素就有唯一确定的顺序。实际情况也常常不是这样，很多情况下需要考虑的序把被排序数据归结为一组（有序的）等价类，同属一类的数据元素都被认为是“相等”的。这样的实际例子很多，例如要求按年龄将一个学校的学生排序，就会有很多人同一年出生，甚至同年同月同日出生；按所发薪金给一个公司的雇员排序，也会有一些人工资相同。

定义

可以给排序问题一个抽象的定义：假设考虑数据集合 S ，集合 S 的元素上有一个序关系 \leq ，一个排序算法 sort 就是从 S 的元素序列到 S 的元素序列的映射。对 S 的任意元素序列 s ， $s' =$

$\text{sort}(s)$ 是 s 的一个排列 (序列中的元素不变, 但其排列顺序可能调整), 使得对 s' 中任意一对相邻元素 e 和 e' , 都有 $e \leq e'$ 。

如果用 $\text{loc}_s(e)$ 表示 e 在序列 s 里的位置, 易见, 在排序后的序列 s' 里, 元素是按照序 \leq 上升存放的, 也就是说, $\text{loc}_{s'}(e) \leq \text{loc}_s(e)$ 当且仅当 $e \leq e'$ 。这里的前一个 “ \leq ” 表示整数 (位置) 的小于关系, 后一个 “ \leq ” 表示这一排序所用的数据集上的序。

显然, 对于同一集实际数据, 完全可能存在很多种不同的但都有意义的序, 而且在计算中的不同时刻, 可能需要对同一个数据序列做不同的排序。例如, 有时可能需要按“小于等于”关系对整数序列排序, 有时可能需要按“大于等于”关系对整数序列排序。作为实际的例子, 有时需要对学校的学生记录按平均成绩排序, 有时需要按某个科目的成绩排序等。因此, 在对某数据集的序列做排序时, 需要明确说明采用哪种序。另一方面, 对一些典型数据集, 也有一些常用的典型序, 如整数的“小于等于”, 字符串的字典序等。

9.1.2 排序算法

虽然可以抽象地讨论排序问题, 但本书和相关课程更关心可以用于计算机的排序方法, 即排序算法。由于排序工作在计算中的重要性, 人们对它进行了不少研究, 提出了许多(不同的)排序算法, 这些算法的基本想法差别很大, 但都能完成排序工作。

人们考虑的一些算法非常朴素而直观, 描述也很简单, 易于理解, 但它们的效率都比较低。另一些算法中更深刻地反映了排序问题的某些本质, 因此效率比较高, 但通常也更复杂一些。例如, 在前面有关二叉树的一章里, 在讨论优先队列和堆的时候介绍的堆排序算法就是一个典型。显然这样的算法不是很容易想出来的。从下面讨论中还会看到更多情况, 看到如何完成排序工作的更多想法。

由于排序是计算中最重要的工作之一, 排序算法的研究一直受到计算机科学技术工作者的重视, 不断有新的研究结果出现, 包括经典算法的调整和实现方法。此外, 程序运行环境的进步也不断带来一些新情况和新问题, 促使人们重新考虑和调整已有的算法。例如, 人们一直在研究能更好利用现代新型硬件结构上(例如多层次存储器上)的优化排序算法, 适用于多核系统、多处理器系统、分布式系统中的高效排序算法等。

基于比较的排序

下面主要考虑基于数据元素中的关键码及其比较操作的排序, 这是排序算法处理的一类最常见情况。还有一些其他排序的考虑, 后面简单介绍。

这里假设需要排序的是某种数据记录的序列 $R_0, R_1, R_2, \dots, R_{n-1}$, 每个记录里有一个或几个支持排序的关键码, 这些关键码相对简单, 存在易于判断的序关系。例如, 关键码就是整数或字符串, 序关系是整数的“小于等于”, 或者字符串的字典序。除此之外, 在数据记录里还可以有任意多的其他成分, 但它们与排序无关, 因此在下面的讨论中省略。为方便讨论, 假定需要考虑的总是 R_i 中的关键码 K_i 。

基于关键码排序, 也就是根据记录中所考虑的那个关键码的序关系整理记录序列, 使之成为按关键码排序的序列。下面将在一次排序中考虑关键码称为排序码。另一方面, 即使是要求针对一个关键码(排序码)对数据记录的序列排序, 可能是要求按排序码的递增顺序, 或者按其递减顺序。为简化讨论, 下面假定总是按递增顺序, 这样也不失一般性。

在一个排序工作的执行过程中, 如果待排序的记录全部保存在内存, 这种工作就称为内排序; 针对外存(磁盘、磁带等)数据的排序工作称为外排序。很多实际算法比较适合做内排序;

也有些排序算法特别考虑到外存的特点，因此特别适合外排序工作，这类算法也被称为外排序算法。本章主要讨论一些内排序算法，其中的归并排序算法是大多数外排序算法的基础。后面将简单介绍外排序算法的一些情况。

前面说，排序工作要求数据集合中存在一种可用的序。由前面讨论可知，如果数据本身没有自然的序，也可以给它造出一种序。最典型的方法就是设计一种 hash 函数，把数据集的元素映射到某个有序集，如整数集合的子集。

基本操作、性质和评价

在考虑算法时，最基本的问题是其时间和空间复杂度。现在的研究对象是基于关键码比较的排序算法，为了在某种合理的抽象层次上考虑它们的时间复杂度和空间复杂度，需要确定关注的基本操作，以其作为时间单位，时间复杂性反映排序过程中这个（或这些）操作的执行次数。还需确定某种抽象的空间单位。

现在要做的是数据记录排序，而且基于关键码比较，比较之后有可能要调整数据记录的位置（顺序）。根据这些情况可以确定两种最重要的基本操作：

- 比较关键码的操作，通过这种操作确定数据的顺序关系。
- 移动数据记录的操作，用于调整记录的位置和 / 或顺序。

在下面讨论各种算法时，总是以被排序序列的长度（即序列中元素的个数）作为问题规模参数 n ，讨论在完成整个排序的过程中执行上述两种操作的次数（的量级）。以此作为评价算法效率的度量（时间复杂度）。

理论研究已经得到了一个明确的结论：基于关键码比较的排序问题，时间复杂度是 $O(n \log n)$ 。也就是说，实现这一过程的任何算法都不可能优于 $O(n \log n)$ 。人们已经开发出来的一些算法达到了这样的效率，因此已经是最优的算法。例如前面讨论过的堆排序算法。后面还会看到几个具有类似性质的算法。

排序是针对已有序列做的一项操作，以此数据序列本身占用的空间与排序无关，是做排序之前就存在并且在使用着的空间。在分析排序算法的空间复杂度时，应该考虑的是为了执行这个算法所需要的空间，因为这部分空间需求由具体算法决定，反映了排序算法的特征。应该看到，这种算法的目的是对已有的序列排序，算法完成后被排序的序列依然存在。因此，算法执行中使用的空间是临时性的辅助空间，用过之后就可以释放了。

在考虑内存排序算法的空间需求时，人们特别关注其空间复杂度能否是常量的。常量的空间开销意味着排序工作可以在原序列（表）里完成，只需要用几个简单变量作为操作中的临时存储。具有这种性质的算法也被称为原地排序算法。

此外，算法本身的复杂程度也是一个需要考虑的因素。复杂的算法不容易实现，编程开发的代价高，而且其代码也要占据更多空间。当然，另一方面，算法的实现只需要做一次，因此这是一个次要因素。

上面这些实际上是对任何算法都需要考虑的问题。除了这些具有普遍意义的性质外，排序算法也有一些与问题相关的特有性质，主要是下面两个。

稳定性：这是排序算法的一种重要性质，稳定的算法在实际中可能更有用。在做一种排序时，待排序序列里可能出现两个不同记录 R_i 和 R_j （设有 $0 \leq i < j \leq n-1$ ）的排序码相同的情况，即有 $K_i = K_j$ 。一般而言，序列中可能有一些排序码相等的记录。显然，在排序之后，所有关键码相等的记录应该是结果序列中连续的一段，但这样的一段记录如何排列呢？首先，由于排序码相同，它们怎样排列都不影响排序算法的正确性，换句话说，排序问题本身并不关心关

键码相同的记录如何排列。但是，实际问题有可能提出更多要求。

举个例子：假定现在要按照成绩的绩点对一个系里的同学排序。在排序之前，全体学生记录是按年级分段存放的。很显然，在排序后的序列里，绩点相同的学生记录将被集中到一起。现在的问题是，绩点同为 4.0 的同学在结果序列里还会按年级分段？还是各年级的同学记录混乱地穿插交错？

如果某个排序算法能保证：对于待排序的序列里任一对排序码相同的记录 R_i 和 R_j ，在排序之后的序列里 R_i 与 R_j 的前后顺序不变，就称这种排序算法是稳定的。也就是说，稳定的算法能够维持序列中所有排序码相同记录的相对位置。如果一个排序算法不能保证上述条件，它就是不稳定的。

稳定的算法可能更适合一些实际情况的需要，上面就是一个实际例子。如果用一个稳定的排序算法处理上述问题，在得到的排序序列里，绩点相同的学生记录子序列也是按年级分段的。一般的情况是，排序之前原序列里的顺序可能隐含一些有用的信息，表示一些与实际问题有关的性质。稳定的排序算法将维持这些信息和性质。

适应性：这是排序算法的另一很有价值的性质。排序操作可能被用于处理不同长短的序列，复杂度的描述方式考虑了这一问题。但是，即使是同样长的被排序序列，情况也很不一样。例如，有时被排序的序列可能很接近排好序的形式，或者原本就是已经排好序的序列，在这些情况下，一个排序算法能否更快完成工作？排序算法的适应性考虑这个问题。如果一个排序算法对接近有序的序列工作得更快，就称这种算法具有适应性。具有适应性的算法也有实际价值，因为实际中常常需要处理接近排序的序列。

排序算法的分类

为了理解各种排序算法在想法和做法等方面的异同，人们经常把典型的排序算法分为一些类别，用类别的名字凸显该类算法的特点。实际上，排序算法可以从不同角度、按不同方式分类。下面是一种常见的分类方法，基于排序的基本操作方式或特点：

- 插入排序
- 选择排序
- 交换排序
- 分配排序
- 归并排序
- 外部排序

下面的讨论并不按照这种分类，但将介绍其中主要分类的一些想法。其中一些简单排序算法只做简单介绍，也有些排序算法在前面已经介绍过，如堆排序算法和简单插入排序算法。人们已经提出的排序算法很多，这里介绍的只是一些最经典的算法。

记录结构

在下面各节中讨论排序算法时，使用的示例数据结构就是一个表（序列的一种连续表示形式），假定表中元素是下面定义的 record 类的对象：

```
class record:  
    def __init__(self, key, datum):  
        self.key = key  
        self.datum = datum
```

排序中只关心 record 对象里的 key 成分，但为了完成排序，经常需要把整个对象搬来搬去。

Python 程序采用引用语义，所谓“搬动”对象不过是复制其引用，这种操作可以映射到计算机硬件，总能在极短的常量时间内完成。

另一方面，下面总假定在 key 成分上所需的关系运算符 ($>$ 、 $<$ 、 \geq 、 \leq 等) 已有定义，并要求以“ \leq ” 运算符的判断确定数据记录的顺序。为简单起见，直接称所做的排序为从小到大排序，或按关键码递增的方式排序。

9.2 简单排序算法

本节介绍几种简单的排序算法，它们的共同特点是简单且最坏情况的复杂度高。

9.2.1 插入排序

插入排序，顾名思义其基本操作方式是插入，不断把一个个元素插入一个序列中，最终得到排序序列。为此只需维持好所构造序列的排序性质，最终就能自然地得到结果。作为插入操作的起点，需要有一个初始的已排序序列。显然无元素的序列能满足要求，只有一个元素的序列也能满足要求，无论其中元素是什么。

算法的考虑和实现

有了上述基本想法，下面的问题就是如何在具体的序列表示下有效实现这一过程。由于现在考虑的是连续表排序，又希望尽可能少用辅助空间，最合适的方法是把正在构造的排序序列嵌入原来的表中。如果能做到这件事，排序中就只需要用几个简单变量，算法就可以做到只需要 $O(1)$ 的空间。

插入过程中需要一个个地处理未排序元素，最简单的方法是按下标处理。处理了一个元素就能留下一个空位。如果该空位与已排序序列相连，就可以直接用作该序列的延伸位置。把这些考虑综合起来，就得到了图 9.1 所示的状态安排。其中在连续表的前段积累已排序序列，通过这个序列的不断生长，最终完成整个序列的排序工作。如图中所示，连续表的右边一段是尚未处理的元素，每次考虑这段的最左元素，即图中用 i 标识的元素。

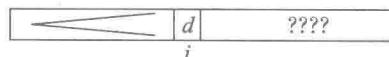


图 9.1 插入排序过程中的状态

在 3.4.4 节介绍单链表排序算法时，已经讨论了如何处理当前记录的问题，这里不再重复。简言之，就是找到第 i 个元素在前段的插入位置（并维持其余元素的序对顺序）。把上述考虑综合起来，就得到了 record 序列的插入排序算法：

```
def insert_sort(lst) :
    for i in range(1, len(lst)): # 开始时片段[0:1]已排序
        x = lst[i]
        j = i
        while j > 0 and lst[j-1].key > x.key:
            lst[j] = lst[j-1] # 反序逐个后移元素，确定插入位置
            j -= 1
        lst[j] = x
```

本算法是 3.4.4 节给出的排序算法的简单修改，其中比较记录的排序码。显然，同样算法可以有许多不同写法。这只不过是一种具体写法，可供参考。

同样采用插入排序方法，完全可以采用不同的数据安排方式。例如，可以在连续表的高端积累排序元素，写出另一个算法。该算法同样采用插入排序的思想，只是落实的方式不同。请读者自己作为练习。

算法分析

上面的算法很简单，其复杂度分析也比较简单。

算法的空间复杂度很清楚：计算中只用了两个简单变量，用于辅助定位和完成序列元素的位置转移。因此算法的空间复杂度是 $O(1)$ ，与序列大小无关。

考虑算法的时间：外层循环总是要进行 $n-1$ 次，内层循环的次数与实际比较的情况有关。变量 j 的初始值越来越大，取值从 1 逐渐增大到 $n-1$ 。如果第二个条件总是失败，也就是说，每次处理的元素比前面所有的元素都小，这个元素就会移到最前面，内层循环体的执行次数等于 j 的值，即为

$$1+2+\cdots+(n-1)=n\times(n-1)/2$$

另一个极端是第二个条件总是第一次就成立，也就是说，被处理元素的关键码总大于已排序的所有元素（因为大于其中最大的元素）。这种情况下内层循环不执行。不难看到，第一种情况对应于原序列逆序排列，第二种情况对应于原序列已排序。

基于上面分析，可以得到算法的关键码比较次数和元素移动次数：

- 关键码比较的次数由内层循环的执行次数决定。最少为 $n-1$ 次，对应于内层循环体一次也不执行；最多为 $n\times(n-1)/2$ 次，对应于总循环到 j 等于 0 的情况。
- 记录移动次数包括内层循环外面的两次和循环中做的一些次，最少为 $2\times(n-1)$ 次，最多为 $2\times(n-1)+n\times(n-1)/2$ 次。

也就是说，最坏情况下算法的复杂度是 $O(n^2)$ ，但如果原序列有序，只需 $O(n)$ 时间。由上面分析可知，这个算法具有适应性。

现在考虑算法的平均时间。显然，一个元素可能插入已排序序列里的任何位置。假设插入各位置的概率相等，内层循环每次迭代的次数就是 $j/2$ ，求和后得到 $n\times(n-1)/4$ ，结合上面讨论，复杂度仍然是 $O(n^2)$ 。

这个简单插入排序算法是稳定的：因为在内层循环中检索插入位置的过程中，一旦发现前面的元素与当前元素关键码相等，就不再移动元素了。这种做法保证了关键码相同的元素不会交换位置，因此算法稳定。

不难看出，稳定性是一个具体算法的性质，而不是排序方法的性质。例如，只需把上述算法内层循环的第二个条件改为 `lst[j-1].key > x.key`，虽然算法的其他性质没变，还是一个简单插入排序算法，但已经不稳定了。

插入排序算法的变形

在插入排序中需要检索元素的插入位置，而且是在排序的（部分）序列里检索。这提示了另一可能方案：采用二分法检索插入位置。

但稍微分析不难看出，这种做法不可能从根本上改变算法的性质：虽然每次检索位置的代价降低了，但找到位置后还需要顺序移动元素，腾出空位将元素插入。后一操作仍然可能需要线性时间。

在基于二分法检索插入位置时，要保证得到的是已排序段中关键码相同元素后面的位置，才能得到稳定的算法。有关算法的开发留给读者完成。

插入排序的思想很容易用于链接表，3.4.4 节给出了一个单链表的插入排序算法。该章练习里有些与此相关的练习。这些问题不再讨论。

相关问题

- 插入排序是最重要的简单排序算法，原因有两点：第一是其实现简单，第二是其自然的稳

定性和适应性。因此，这种算法经常被用于一些高级排序算法，作为其中的组成部分。例如，有一种称为 shell 排序的算法，采用一种变动间隔的排序技术，其中用简单插入排序作为基础算法，可以得到更高的操作效率。另外有些效率很高的算法在排序中采用了切分待排序序列的方式工作（例如后面可以看到的快速排序和归并排序），当切分得到的序列很短时，就转到简单的插入排序算法。Python 语言内置排序算法也是这样，见后。

9.2.2 选择排序

在插入排序中每次操作处理哪个记录并不重要，关键在于把被处理记录插入已排序序列中的正确位置，因此可以采用最方便的方式取记录，即按顺序提取。选择排序的想法与之对应，这里的重要决策（和基本操作）是选择合适记录，只要严格按递增方式选出记录（每次选最小元素），简单地顺序排放就能完成排序工作。

选择排序的基本思想也很简单：

- 维护需要考虑的所有记录中最小的 i 个记录的已排序序列。
- 每次从剩余未排序的记录中选取关键码最小的记录，将其放在已排序序列记录的后面，作为序列的第 $i+1$ 个记录，使已排序序列增长。
- 以空序列作为排序工作的开始，做到尚未排序的序列里只剩一个元素时（它必然为最大），只需直接将其放在已排序的记录之后，整个排序就完成了。

显然，如果需要做从大到小的排序，只需要每次选取最大元素。

在这种基本想法之下，还需要解决两个问题：第一是如何选择元素；第二是做出适当安排，尽可能利用现有序列的存储空间，避免另行安排存储。

简单选择排序

最简单的选择方法是顺序扫描序列中的元素，记住遇到的最小元素。一次扫描完毕就找到了一个最小元素。反复扫描就能完成排序工作。另一方面，选出了一个元素，原来的序列中就出现了一个空位，可以把这些空位集中起来存放排好序的序列。

综合这些考虑，在被排序表的前段积累已排序序列，就可以得到一个简单的选择排序算法。有关排序过程中的基本状态如图 9.2 所示。在排序过程中
的任何时刻，表的前段积累了一批递增的已经排好序的记录，而且它们都不大于任何一个未排序记录。下一步从未排序段中选出最小的记录，将其存放在已排序记录段的后面。这样在只剩一个记录时，其关键码一定最大，工作即可结束。

图 9.2 选择排序中的状态

这里出现了一个问题：如何腾出紧随已排序段的那个位置。直接选择排序算法中简单交换这里的元素与未排序段选出的最小记录。这一做法既把新选出的元素存放好，又填补起选走一个元素留下的空位，看起来两全其美。

直接选择排序算法

现在考虑上述算法的实现。显然，操作中需要反复选择 $n-1$ 次，可以用一个确定次数的循环实现。每次选择需要扫描所有未排序元素，以确定其中关键码最小者，这件事也是一个确定次数的重复操作。易见，这种排序可以用两重 for 循环实现。

算法很简单：

```
def select_sort(lst):
```

```

for i in range(len(lst)-1):           # 只需循环len(lst)-1次
    k = i
    for j in range(i, len(lst)):       # k是已知最小元素的位置
        if lst[j].key < lst[k].key:
            k = j
    if i != k:                      # lst[k] 是确定的最小元素, 检查是否需要交换
        lst[i], lst[k] = lst[k], lst[i]

```

易见，这种直接选择排序的比较次数与被排序表的初始状态无关，两个 `for` 循环总是按固定方式重复执行那么多次。比较的次数总是

$$1+2+\cdots+(n-1)=n\times(n-1)/2$$

记录移动的次数则依赖于具体情况。如果每次确定的 `k` 值都等于 `i`，就达到了最少的移动次数 0，这对应于被排序表中的记录已递增排序的情况。最差情况是每次都交换，上面算法里采用了一个并行赋值，可以认为移动次数是 $2\times(n-1)$ 。综合考虑，这种直接选择排序算法的平均和最坏情况时间复杂度都是 $O(n^2)$ 。从上面分析还可以看出，这个算法没有适应性，在任何情况下时间开销都是 $O(n^2)$ 。

另一方面，算法中只用到几个变量，空间复杂度是 $O(1)$ 。

现在考虑算法的稳定性。可以注意到，从左到右扫描找到的最小记录，一定是顺序上的第一个最小记录。即使序列中还有与之关键码等值的记录，在最后的排序序列里，它们也应该排在这次选择的记录之后。所以，检索最小记录并将其放到排序段之后，不会破坏稳定性。但另一方面，把原来在排序段之后的那个记录（设为 `x`）交换到最小记录的位置，却可能造成关键码相同元素交换次序，因为 `x` 的这种移动可能越过一批记录，无法保证其中不出现与 `x` 的关键码相同的记录。这说明上述算法不稳定。

然而，看到上面的问题也就找到了一种修改上述算法，使之变为稳定的方法。只需在找到了最小记录之后，依次逐个后移位置 `k` 之前的那些尚未排序的元素，腾出所需的空位后把最小记录存入。修改后的算法就是稳定的。

试验说明，直接选择排序算法的实际平均排序效率低于插入排序算法。由于在方方面面都不如插入排序，直接选择排序很少被实际使用。

提高选择的效率

选择排序比较低效，原因就在于其中的顺序比较：每次选择一个元素，都是从头开始做一遍完全的比较，在整个排序过程中做了很多重复比较工作。

要想提高选择排序的效率，就需要改变选择方式。重要的是设法记录在做选择的过程中，已做过的关键码比较获得的信息。在这种基本想法下，人们研究了各种树形选择技术，设法通过沿着树中路径比较的方式选出最小元素，也就是说，利用树高度与元素个数之间的对数关系。这种想法的最重要结果就是在 6.3.5 节介绍的堆排序算法。

堆排序是一种高效的选择排序算法，基于堆的概念。该算法高效的主要原因是在堆里积累了前面所做的比较中得到的信息。由于堆的结构特征，这种信息可以很自然地重复利用。另一方面，由于堆和顺序表的关系，一个堆和一个排序序列可以很方便地嵌入同一个顺序表，不需要任何辅助结构。堆排序算法的有关细节见 6.3.5 节，不再重复。现在仅从排序的角度考察一下这个算法的特点。

根据前面的分析可知，采用堆排序技术，初始建堆需要 $O(n)$ 时间，随后每选择一个元素不超过 $O(\log n)$ 时间，所以堆排序算法的时间复杂性是 $O(n \log n)$ 。另一方面，堆排序也是一种原位排序算法，工作中只需要用几个辅助变量，空间复杂性是 $O(1)$ 。在这两方面，堆排序

都已经达到了最佳的情况。

堆排序的最大问题是不稳定。在初始建堆和一系列选择元素后的筛选中，元素都沿着堆中（二叉树中）的路径移动。这种移动路径与连续表里自然的顺序相互交叉，在这种移动中，很可能出现相同关键码记录的顺序被交换的情况。因此，堆排序的特点决定了它的不稳定性，而且很难做出稳定的堆排序算法。另外堆排序也没有适应性。在选取了最小元素后，总取出最后一个元素做筛选，经常导致较长的筛选路径。

最后，在常规的自然的堆排序算法实现中，每次选择关键码最小的记录（采用小顶堆），得到的将是从大到小的排序序列；每次选择关键码最大的记录（采用大顶堆），才能得到关键码从小到大的排序序列。

9.2.3 交换排序

另一种排序方法基于完全不同的看法：一个序列中的记录没排好序，那么其中一定有逆序存在。如果交换所发现的逆序记录对，得到的序列将更接近排序序列；通过不断减少序列中的逆序，最终可以得到排序序列。采用不同的确定逆序方法和交换方法，可以得到不同交换排序方法。

起泡排序

起泡排序是一种典型的（也是最简单的）通过交换元素消除逆序实现排序的方法。其中的基本操作是比较相邻记录，发现相邻的逆序对时就交换它们。通过反复比较和交换，最终完成整个序列的排序工作。显然：如果序列中每对相邻记录的顺序正确（前一记录不大于后一记录），整个序列就是一个排序序列。

图 9.3 展示了对一个简单序列的起泡排序过程。这里从左到右顺序比较一对对相邻记录，发现逆序后马上交换，然后再做下一次比较。可以看到一些情况：

- 每一遍检查可以把一个最大元素交换到位，一些较大元素右移一段，可能移动很远。
- 从左到右比较，导致小元素一次只左移一位。个别距离目标位置很远的小元素，如本例中的 10，可能延误整个排序进程。

比较和交换导致较大记录右移，就像水中气泡浮起，是这种排序方法名称的由来。

一次完整扫描（比较和交换）能保证把一个最大的元素移到未排序部分的最后。通过一遍遍扫描，表的最后将积累起越来越多排好顺序的大元素。每遍扫描，这段元素增加一个，经过 $n-1$ 遍扫描，一定能完成排序。此外，做一遍，扫描的范围可以缩短一项。把这些考虑综合起来就得到了下面的算法：

```
def bubble_sort(lst):
    for i in range(len(lst)):
        for j in range(1, len(lst)-i):
            if lst[j-1].key > lst[j].key:
                lst[j-1], lst[j] = lst[j], lst[j-1]
```

算法的改进

虽然有时起泡排序确实需要做满 $n-1$ 遍，如图 9.3 中的情况，但那是特例，只有被排序

初始状态:	30	13	25	16	47	26	19	<u>10</u>	
第1遍:	13	25	16	30	26	19	<u>10</u>	47	
第2遍:	13	16	25	26	19	<u>10</u>	30	47	
第3遍:	13	16	25	19	<u>10</u>	26	30	47	
第4遍:	13	16	19	<u>10</u>	25	26	30	47	
第5遍:	13	16	<u>10</u>	19	25	26	30	47	
第6遍:	13	<u>10</u>	16	19	25	26	30	47	
第7遍:	<u>10</u>	13	16	19	25	26	30	47	

图 9.3 起泡排序实例

表的最小元素恰好在最后时才会出现这种情况。在其他情况下，扫描就不需要做那么多次，如果发现排序已经完成就可以及早结束。易见，如果在一次扫描中没遇到逆序，就说明排序工作已经完成，可以提前结束了。

按照这种想法改进的算法如下：

```
def bubble_sort(lst):
    for i in range(len(lst)):
        found = False
        for j in range(1, len(lst)-i):
            if lst[j-1].key > lst[j].key:
                lst[j-1], lst[j] = lst[j], lst[j-1]
                found = True
        if not found:
            break
```

在外层循环里增加了一个辅助变量 `found`，在内层循环开始之前将 `found` 赋为 `False`。如果扫描检查中遇到了逆序，就给 `found` 赋 `True` 值。在内层循环结束后检查 `found`，其值为 `False` 表示未发现逆序，立刻结束循环。

这样做可能提高效率，而且使算法具有了适应性。不难看到，如果被排序序列有序，经过第 1 遍扫描，函数就结束了。其间只做了 $n-1$ 次比较。

起泡排序的性质很清楚：最坏情况时间复杂度为 $O(n^2)$ ，平均时间复杂度也为 $O(n^2)$ 。改进的方法在最好情况下的时间开销为 $O(n)$ 。起泡排序算法也是一种原位排序算法，其中使用的辅助空间是 $O(1)$ 。

起泡排序算法的稳定性依赖于其中相等的元素不交换，很自然。

情况分析

试验说明，起泡排序的效率比较低，实际效果劣于复杂度相同的简单插入排序算法。究其原因，可能有两个方面。其一是反复交换中做的赋值操作比较多，累积起来代价也比较大。另一方面，一些距离最终位置很远的记录可能拖累整个算法。在简单起泡排序中，导致这种结果主要是那些距离远的小记录。

要缓解第二个问题，应该想办法让元素大跨步地向其最终位置移动。下一节介绍的快速排序工作方式就有这种效果。

另一种简单方法是交错起泡，以便把小元素快速移向左方。具体做法是一遍从左向右扫描，下一遍从右向左，交替进行。重看图 9.3 的例子，采用交错技术的情况如图 9.4 所示，4 遍就完成排序，其中两遍向右，两遍向左。相应函数的定义留作练习。

初始状态:	30 13 25 16 47 26 19 10
第1遍:	13 25 16 30 26 19 10 47
第2遍:	10 13 25 26 30 26 19 47
第3遍:	10 13 16 25 26 19 30 47
第4遍:	10 13 16 19 25 26 30 47

图 9.4 交错起泡排序实例

9.3 快速排序

快速排序是一种著名的排序算法，1960 年前后由英国计算机科学家 C. A. R. Hoare 提出，作为最早的采用递归方式描述的一种优美算法（当时新开发的 Algol 60 编程语言引进了递归描述方式），展示了递归描述形式的威力。在各种基于关键码比较的内排序算法中，快速排序是实践中平均速度最快的算法之一。快速排序算法还被人们评为“20 世纪最具影响力的十个算法”之一。

快速排序实现中也采用了发现逆序和交换记录位置的方法，但算法中最基本的思想是划

分，即按某种标准把考虑的记录划分为“小记录”和“大记录”，并通过递归不断划分，最终得到一个排序的序列。其基本过程是：

- 选择一种标准，把被排序序列中的记录按这种标准分为大小两组。显然，从整体的角度，这两组记录的顺序已定，较小一组的记录应该排在前面。
- 采用同样方式，递归地分别划分得到的这两组记录，并继续递归地划分下去。
- 划分总是得到越来越小的分组（可能越来越多），如此工作下去直到每个记录组中最多包含一个记录时，整个序列的排序完成。

快速排序算法有许多不同的实现方法，下面主要介绍其中一种针对顺序表的实现。实际上，快速排序的思想同样可以用于链接表，这一工作留作练习。

9.3.1 快速排序的表实现

在实现排序工作时，人们希望尽可能在表的内部完成排序，尽可能少使用辅助空间。对快速排序，一个重要设计目标是希望在原表的内部实现划分，也就是说，通过在表内移动记录将它们分为大小两段。根据后面工作需要，将小记录移到表的左部，大记录移到右部。这样，整个递归完成时表中记录就自然有序了。

下一个问题是需要确定一种划分规则。现在考虑最简单的划分方式：取序列中第一个记录，以其关键码为标准划分其他记录，把关键码小的记录移到表的左边，关键码大的记录移到右边。划分完成后表中间将留下一个空位，这就是作为比较标准的记录的正确位置。把这个记录存入，其位置就固定了，在随后的操作中不需要改变。

下一步是用同样方式分别处理两段记录，并继续递归处理。当一个记录分段只有一个元素或没有元素时，其排序已完成。完成所有分段的排序，也就完成了整个表的排序。

实际上，完全可以采用其他方法选择划分标准和移动记录，不同具体做法形成了顺序表上快速排序的不同实现，后面有简单讨论。在不同教科书中可能看到不同的快速排序算法实现，但其基本思想相同，都基于关键码划分表中的记录，得到两个分段后再递归处理分段。当然，完全可以用循环的方式写出快速排序程序。

(一次) 划分的实现

现在考虑一次划分的实现。假设现在考虑一段记录，取出其中第一个记录作为标准，设其为 R 。由于对大小记录的安排，划分中的一般状态如图 9.5a 所示。已知的小记录积累在左边，大记录积累在右边，中间是尚未检查的记录。

为了完成划分，还需要利用好表中空位。取出记录 R 使表左边出现了一个空位（图 9.5b）。这时从右端开始检查，就可以利用这个空位，把发现的第一个小记录移到左边。这一迁移操作也导致右边留下一个空位（图 9.5c），可供存放在左边发现的一个大记录。下面算法中采用这种交替的工作方式。

算法中利用两个下标变量 i 和 j ，其初值分别是序列中第一个和最后一个记录的位置。在划分过程中，它们的值交替地作为空位和下一被检查记录的下标。然后取出第一个记录 R ，设其排序码为 K ，作为划分标准。

- 交替进行下面两套操作：
 - 状态如图 9.5b 所示。从右向左逐个检查 j 一边的记录，检查中 j 值不断减一，直至找到第

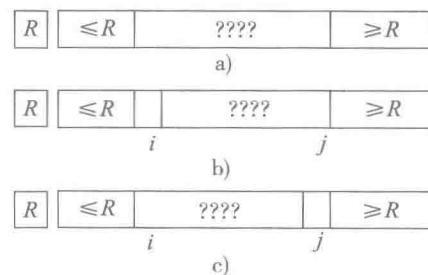


图 9.5 快速排序一次划分中的状态

一个关键字小于 K 的记录，将其存入 i 所指的空位。注意，移动记录后位置 j 变成空位， i 值加一后指向下一需要检查的记录。

○ 状态如图 9.5c 所示。从左向右逐个检查 i 一边的记录，检查中 i 值不断加一，直至找到第一个关键字大于 K 的记录并将其存入 j 所指空位。移动记录又得到图 9.5b 所示的状态，转做上面操作。

- 重复交替进行上述两套操作，直到 i 不再小于 j 为止。由于第一种操作中 j 值不断减小，第二种操作中 i 值不断增大，划分一定能完成。
- 划分结束时 i 与 j 的值相等，指向当时表中的空位。将记录 R 存入空位（其位置已正确确定），一次划分完成。

一次划分完成后对两边子序列按同样方式递归处理。由于要做两个递归，快速排序算法的执行形成了一种二叉树形式的递归调用。

9.3.2 程序实现

为方便使用，下面定义一个非递归的主函数，其中调用一个递归定义的函数（可以作为局部定义），给定划分范围初值：

```
def quick_sort(lst):
    qsort_rec(lst, 0, len(lst)-1)
```

递归过程的框架：

```
def qsort_rec(lst, l, r):
    if l >= r: return # 分段无记录或只有一个记录
    i = l;
    j = r
    pivot = lst[i] # lst[i] 为初始空位
    while i < j: # 找pivot的最终位置
        ... ...
        ... ...
    lst[i] = pivot # 将 pivot 存入为其确定的最终位置
    qsort_rec(lst, l, i-1) # 递归处理左半区间
    qsort_rec(lst, i+1, r) # 递归处理右半区间
```

局部变量 $pivot$ 保存作为调整记录位置的检查标准记录（枢轴元素），要求大记录向右移动而小记录向左移动。最后将这个记录存入确定的位置，对两边递归。

递归的快速排序函数定义如下：

```
def qsort_rec(lst, l, r):
    if l >= r: return # 分段无记录或只有一个记录
    i = l
    j = r
    pivot = lst[i] # lst[i] 是初始空位
    while i < j: # 找pivot的最终位置
        while i < j and lst[j].key >= pivot.key:
            j -= 1 # 用j向左扫描找小于pivot的记录
        if i < j:
            lst[i] = lst[j]
            i += 1 # 小记录移到左边
        while i < j and lst[i].key <= pivot.key:
            i += 1 # 用i向右扫描找大于pivot的记录
        if i < j:
```

```
lst[j] = lst[i]          # 大记录移到右边
j -= 1
lst[i] = pivot          # 将pivot存入其最终位置
qsort_rec(lst, l, i-1)  # 递归处理左半区间
qsort_rec(lst, i+1, r)  # 递归处理右半区间
```

其过程完全遵循上面的框架，只是填入了划分过程的细节，包括各种比较条件、变量 i 和 j 的修改操作、元素移动操作等。

9.3.3 复杂度

首先考虑算法的时间复杂度，以被排序序列中记录的个数为 n 。首先，很容易确认，在整个快速排序算法的工作过程中，记录移动的次数不大于比较次数，因此只需考虑比较的次数。而比较的次数与划分的情况有关：

- 如果每次划分都把所处理区域划为长度基本相等的两段，很显然，只需大约 $\log n$ 层划分，就能使最下层的每个分段长度不超过 1。而在划分一层记录的过程中，关键码比较次数不超过序列长度（每层都有些枢轴元素不需要比较，越往下这种元素越多），综合起来可知，排序中关键码的比较次数不超过 $O(n \log n)$ 。
- 但是，如果每层划分得到的两段中总有一段为空，另一段的记录个数只比本层划分前少一个。在这种情况下，要使所有分段的长度不大于 1，就要做 $n-1$ 层划分。完成各层划分的比较次数从 $n-1$ 逐层减少到 1。显然，这种情况下总的比较次数是 $O(n^2)$ 。举例说，待排序序列已经是升序或降序时，都会出现这种情况。

由上面分析可知，快速排序的最坏情况时间复杂度是 $O(n^2)$ 。

抽象地看，快速排序产生的划分结构，可以看作以枢轴记录为根，以两个划分分段进一步划分的结果作为左右子树的二叉树。根据二叉树理论，所有 n 个结点的二叉树的平均高度是 $O(\log n)$ 。因此，快速排序的平均时间复杂度是 $O(n \log n)$ 。

排序低效的原因是分段不均衡，根源是划分标准没有取好。为减少最坏情况出现，可以考虑修改划分的判据。例如，有人提出“三者取中”规则：在每趟划分前比较分段中第一个、最后一个和位置居中的三个记录的关键码，取其中值居中的记录与首记录交换位置，而后基于这个记录的关键码做划分。这种做法可以减少最坏情况出现的概率，但无法消除最坏情况。请读者设法举出对于这种改进的最坏情况实例。

现在考虑快速排序算法的空间复杂度。从表面看，在函数定义里只用到 i 和 j 等几个变量。但应注意，函数 `qsort_rec` 是递归定义的，每次（每层）递归调用都需要再次创建这些变量。另外，为支持递归执行，解释器也需要使用运行栈的空间，这些都是为实现排序而使用的辅助空间。虽然不知道解释器为每层递归花费的具体空间量，但应该认为它是常量，与递归深度和函数调用次数无关。因此，分析前面快速排序算法的空间复杂度，最重要的问题就是递归的深度，最坏情况是 $O(n)$ 。

这个空间复杂度与具体实现有关，可以改进。采用递归定义，执行方式由解释器确定，不容易控制。可以换一种方法，在程序里引入一个栈以保存未处理分段的信息，用非递归方式实现快速排序算法，可以更灵活地选择分段处理的顺序。如果每次划分后，把较长分段的信息入栈，先处理较短的分段，记录分段信息的栈的深度就不会超过 $O(\log n)$ ，所以快速排序的辅助空间可以做到为 $O(\log n)$ 。

最后，常见的（包括这里的）快速排序算法是不稳定的，但也有人研究并提出了一些稳定

的快速排序算法。从前面示例可以看出，快速排序算法不会因为原序列接近有序而更高效（适得其反），因此它也不具有适应性。

9.3.4 另一种简单实现

快速排序算法的实现方法也可以有很多变化。本小节给出另一种非常简洁的实现，其工作方式与前面的算法差别很大，但也同样是一种快速排序算法。

这个算法在运行中一次划分的中间状态如图 9.6 所示。其中的 R 是作为划分标准的记录，以其关键码 K 作为分界线，将表中记录（一般而言，是表中一个分段中的记录）划分为两组。工作过程中本分段的记录（除 R 外）顺序分为三组：小记录，大记录，未检查记录。这里用两个下标变量， i 的值总是最后一个小记录的下标，而 j 的值是第一个未处理记录的下标。每次迭代比较 K 与记录 j 的关键码，有两种情况：

- 记录 j 较大：这时简单地将 j 加一，又恢复到图 9.6 中状态。
- 记录 j 较小：这时需要把这个记录调到左边，做法是将 i 加一，而后交换 i 和 j 位置的记录，并将 j 值加一，又恢复到图 9.6 中状态。

划分完成后还需把 R 移到正确位置。这件事也很简单，只需交换它与位置 i 的记录。这时处于 R 之前的记录均小于 R ，其后的记录都大于等于 R 。划分完成，下面按同样方式处理小记录分段和大记录分段（递归），直至排序完成。

下面是函数的实现。与上面讨论中略微不同的是，这里用了一个 for 循环，因为要检查的范围已知，做的就是一个个检查记录：

```
def quick_sort1(lst):
    def qsort(lst, begin, end):
        if begin >= end:
            return
        pivot = lst[begin].key
        i = begin
        for j in range(begin + 1, end + 1):
            if lst[j].key < pivot:           # 发现一个小元素
                i += 1
                lst[i], lst[j] = lst[j], lst[i] # 小元素换位
        lst[begin], lst[i] = lst[i], lst[begin] # 枢轴元就位
        qsort(lst, begin, i - 1)
        qsort(lst, i + 1, end)

    qsort(lst, 0, len(lst) - 1)
```

本算法的性质与前面快速排序算法相同，不再赘述。

9.4 归并排序

归并是一种典型的序列操作，其工作是把两个或更多有序序列合并为一个有序序列。基于归并的思想也可以实现排序，称为归并排序。基本方法如下：

- 初始时，把待排序序列中的 n 个记录看成 n 个有序子序列（因为一个记录的序列总是排好序的），每个子序列的长度均为 1。
- 把当时序列组里的有序子序列两两归并，完成一遍后序列组里的排序序列个数减半，每个子序列的长度加倍。

R	$<R$	$\geq R$????
i		j	

图 9.6 快速排序的另一种实现

- 对加长的有序子序列重复上面的操作，最终得到一个长度为 n 的有序序列。

这种归并方法称为简单的二路归并排序，其中每次操作都是把两个有序序列合并为一个有序序列。也可考虑三路归并或更多路的归并。

易见，归并操作是一种顺序性操作，其中按自然存在的顺序使用已排序序列，也按顺序产生归并后的结果序列。前面说过，外存数据比较适合顺序处理，但不适合随机访问。此外，归并操作过程中对数据的访问具有局部性，适合外存数据交换的特点，特别适合处理一组记录形成的数据块。由于这些情况，归并操作适合用于处理存储在外存的大量数据。实际上，许多实际的外存数据排序算法都是基于归并操作设计和实现的。

下面讨论（内存中）顺序表的归并排序，从中可以看到和理解归并操作的情况，以及其中可能遇到的各种问题。

9.4.1 顺序表的归并排序

下面讨论顺序表上的二路归并排序算法。先看一个例子：假设给定的初始记录序列为 25, 67, 54, 33, 20, 78, 65, 49, 17, 56, 44。图 9.7 给出对这些数据进行归并，最终得到有序序列的过程，其中用下划线标出一个个有序序列。在初始状态中，每个记录自成一个有序序列。第 1 遍将这些序列归并为一组长度为 2 的有序序列，注意其中最后的 44 没有归并对象，原样留到下一步。第 2 步归并出 3 个长度为 4 的有序子序列，最后一个序列中记录数不足 4 个。再经过两遍归并，最后得到了所需的排序序列。

初始状态:	25	67	54	33	20	78	65	49	17	56	44
第1遍:	25	67	33	54	20	78	49	65	17	56	44
第2遍:	25	33	54	67	20	49	65	78	17	44	56
第3遍:	20	25	33	49	54	65	67	78	17	44	56
第4遍:	17	20	25	33	44	49	54	56	65	67	78

图 9.7 归并排序实例

9.4.2 归并算法的设计问题

弄清了基本算法过程，下面的问题是如何实施这种计算过程。在这里需要一遍遍地归并，出现的问题就是把作为归并结果的序列放在哪里。

前面讨论各种排序算法时，一直考虑“原地排序”的问题，即在原表里完成排序工作。对于归并，两段被归并子序列和归并后的序列长度相同，原则上说有可能实现“原地归并”。但是实施这种操作不容易。人们已经提出了一些技术，但都比较复杂。有兴趣的读者请自己查资料，或者自己考虑可行的方法。

在下面的算法里采用了一种简便的处理方式，为实现归并另外开辟了一片同样大小的存储区，也就是建立了另一个同样大小的表，把一遍归并的结果放在那里。在一遍归并做完后，在保存归并结果的新表里存储着加倍长度的有序序列，而且原来的表已经闲置了。在这种情况下，可以考虑调换两个表的角色，在原表中累积下一遍归并的结果。这样来来回回做几遍，就能完成整个排序工作了。

显然，采用这种方式，实际上至少需要 $O(n)$ 的辅助空间。这是付出空间代价，获得算法实现的简便性。在实际中人们也经常这样做。

9.4.3 归并排序函数定义

参考图 9.7，可以看到归并排序的工作分为几个层次：最上层控制一遍遍归并，完成整个表的排序工作；在一遍处理中需要分别完成一对对递增序列的归并；在归并每一对序列中又需要一个个地处理序列元素。由于这种情况，为了描述得清晰易读，下面的表归并排序算法也分

三层实现(其中用一个同样大小的辅助表):

- 1) 最下层: 实现表中相邻的一对有序序列的归并工作, 将归并的结果存入另一个顺序表里的相同位置。
- 2) 中间层: 基于操作 1(一对序列的归并操作), 实现对整个表里顺序各对有序序列的归并, 完成一遍归并, 各对序列的归并结果顺序存入另一顺序表里的同位置分段。
- 3) 最高层: 在两个顺序表之间往复执行操作 2, 完成一遍归并后交换两个表的地位, 然后再重复操作 2 的工作, 直至整个表里只有一个有序序列时排序完成。

由于在一般情况下, 被排序表的长度不是 2 的幂, 因此需要特别考虑表最后的不规则情况。下面算法中的几个地方都考虑了这方面问题。

先考虑最下面一层函数 `merge`, 它完成表中连续排放的两个有序序列的归并工作。`merge` 需要把一个表的元素按序排列在另一个表里, 因此应有两个表参数: 被归并的有序段位于表 `lfrom` 中, 归并结果需要存入表 `lto` 里对应位置分段中。显然必须有参数说明两个被归并分段的下标范围。由于这两个分段相邻, 以及整个表最后的分段可能不规范(长度不足), 下面定义中用三个下标参数说明被处理分段的范围, 需要归并的两个有序分段分别是 `lfrom[low:mid]` 和 `lfrom[mid:high]`, 显然, 归并结果应存入 `lto[low:high]`。这里按 Python 的惯例, 采用左闭右开的区间表示。

函数里的第一个循环处理两个分段都有未归并元素的情况, 其每次迭代取出两个分段中当时的最小记录(它一定是这两分段之一的最小记录), 把它移到 `lto` 表里的下一个位置。在循环体里还要正确更新几个下标变量。当某个分段不再有更多记录时本循环结束。后随的两个循环把另一分段中剩下的元素逐一复制到表 `lto` 中。请注意, 这两个循环中只有一个会真正执行。下面写法比较简单, 也完全正确。

```
def merge(lfrom, lto, low, mid, high):
    i, j, k = low, m, low
    while i < mid and j < high: # 反复复制两分段首记录中较小的
        if lfrom[i].key <= lfrom[j].key:
            lto[k] = lfrom[i]
            i += 1
        else:
            lto[k] = lfrom[j]
            j += 1
        k += 1
    while i < mid: # 复制第一段剩余记录
        lto[k] = lfrom[i]
        i += 1
        k += 1
    while j < high: # 复制第二段剩余记录
        lto[k] = lfrom[j]
        j += 1
        k += 1
```

函数 `merge_pass` 实现一对对分段的一遍归并, 它需要知道表长度和分段长度, 参数 `llen` 和 `slen` 分别表示这两个长度。第一个循环处理一对对长度都为 `slen` 的分段, 随后的 `if` 语句处理表最后剩下的两个或一个分段:

```
def merge_pass(lfrom, lto, llen, slen):
    i = 0
    while i + 2 * slen < llen: # 归并长slen的两段
        merge(lfrom, lto, i, i + slen, i + 2 * slen)
```

```
i += 2 * slen
if i + slen < llen:           # 剩下两段, 后段长度小于slen
    merge(lfrom, lto, i, i + slen, llen)
else:                         # 只剩下一段, 复制到表lto
    for j in range(i, llen):
        lto[j] = lfrom[j]
```

最后是主函数 `merge_sort`。它采用前面讨论的方法, 先安排另一个同样长度的表, 而后在两个表之间往复地做一遍遍归并, 直至完成工作。

```
def merge_sort(lst):
    slen, llen = 1, len(lst)
    templst = [None]*llen
    while slen < llen:
        merge_pass(lst, templst, llen, slen)
        slen *= 2
        merge_pass(templst, lst, llen, slen)  # 结果存回原位
        slen *= 2
```

整个序列的实际排序完成时, 得到的结果有可能正好放在 `templst` 里, 这时再执行一次 `merge_pass` (循环体里的第二个 `merge_pass` 调用) 就能把结果复制回原来的表 `lst`。无论怎样, 整个排序工作都能顺利完成。

9.4.4 算法分析

首先考虑算法的时间复杂度。易见, 做完第 k 遍归并后, 有序子序列的长度将为 2^k 。因此, 完成整个排序需要做的归并遍数不会多于 $\log_2 n + 1$ 。进而, 在每遍归并中做的比较次数为 $O(n)$, 所以, 总的比较次数和移动次数都为 $O(n \log n)$ 。

空间复杂度: 在上面给出的算法里用了一个辅助表, 它与原记录序列的长度相同。因此这个算法的空间复杂度是 $O(n)$ 。这是归并排序算法的主要弱点。人们已经在减少归并排序算法的辅助空间方面做了很多研究, 取得了一些成果。

上面给出的归并排序算法是稳定的。实际上, 做出稳定的归并排序并不难, 只需特别关注归并中遇到排序码相同的记录时的选择顺序。只要在关键码相同时采用左序列元素先行的原则, 就能保证算法的稳定性。

另一方面, 这个归并排序算法没有适应性, 无论对什么样的序列它都做 $\log_2 n$ 遍归并。修改归并排序算法, 使之具有适应性的工作留给读者自己考虑。下面介绍的 Python 系统的排序算法, 也就是顺着这个思路研究下去取得的成果。

9.5 其他排序方法

本章最后介绍与排序有关的另一些重要情况, 包括 Python 内置的排序算法等。

9.5.1 分配排序和基数排序

有关排序问题的研究主要集中在基于 (关键码) 比较的排序算法, 前几节介绍了一些算法以及它们背后的重要思想。本节简单介绍有关排序的另一种想法, 它并不基于关键码比较, 而是基于一种固定位置的分配和收集。

分配与排序

如果关键码只有很少几个不同的值, 存在一种简单而直观的排序方法:

- 为每个关键码值设定一个桶（即是能容纳任意多个记录的容器，例如用一个连续表或链接表，参考前面桶散列表的局面描述，见图 8.5）。
- 排序时简单地根据关键码把记录放入相应桶中。
- 存入所有记录后，顺序收集各个桶里的记录，就得到了排序的序列。

例如，如果关键码取值为整数 $0 \sim 9$ ，只需 10 个桶就能完成排序工作。做一遍分配，所有关键码相同的记录都在一个桶里，顺序收集各桶中的记录并将其顺序排列，只要存入元素和收集的方式维持原序列顺序，就可以完成稳定的排序。

这种排序算法也有实用价值。例如，假设需要按成绩绩点对全校同学的学习记录排序。由于绩点只有 2 位数字，总共几十个可能值。只需要对应每个绩点值设置一个桶，通过一遍记录分配和一遍收集，就能得到稳定排序的结果。显然，按照合理的算法实现，这一过程可以在 $O(n)$ 时间内完成，复杂度低于采用关键码比较排序的最优算法，在实际使用中很可能更加快捷。具体算法的实现留作练习。

如果关键码的可能取值集合很大，例如取值是范围为 $0 \sim 2^{32}-1$ 的整数。采用分配排序，首先需要建立大量的桶（为每个整数建一个桶），但在实际排序中绝大部分的桶可能是空的，显然这不是适合分配排序的场景。

多轮分配和排序

如果只允许为数不多的一小批桶，分配排序方法的应用范围就太窄了。人们提出了一种扩充其能力的想法，其中采用元素适合分配排序的元组作为关键码，通过多轮分配和收集，完成以这种元组为关键码的记录的排序工作。

抽象地看，这样的元组关键码相当于某种“字符串”，关键码元组的每个元素是一个“字符”，关键码相当于字符串，其自然的序就是这种串的字典序。

作为一个具体例子，考虑以 (a,a,a) 形式的三元组作为关键码，其中 a 为数字，取值来自集合 $\{0, 1, 2, 3\}$ 。考虑下面的待排序关键码序列：

$(1,3,2) (3,1,3) (2,2,0) (0,1,1) (2,0,2) (3,0,1) (0,3,2) (1,0,3) (0,2,0) (2,1,1)$
 $(3,1,0) (2,0,1) (1,1,0) (0,1,3) (2,3,1) (0,2,2) (0,3,0) (3,3,1) (2,1,3) (3,1,1)$

下面考虑如何对这样的关键码排序。

现在的关键码是等长的元组，要求按字典序排序，应该利用元组的结构，顺序处理其中的元素。由于关键码中每个元素的可能取值很少，可以用分配排序完成针对一个元组元素的排序。显然，顺序处理元组中元素存在两种方法，下面分析其中情况：

- 从最高位（最左位）开始以此考虑关键码元素，这称为高位优先方法（Most Significant Digit first, MSD）。按这种方法处理一遍，得到原序列的一种分割，如图 9.8 所示。要完成排序，下一步应该考虑各子序列的排序（基于关键码的下一元素），然后考虑子序列的子序列排序。直到所有最小的子序列完成排序，

整个序列的排序研究完成了。这一过程显然可以实现，但其中也有些问题比较麻烦，主要是需要考虑越来越多的子序列，需要记录相关信息。

- 从最低位开始，称为（Least Significant Digit first, LSD）。首先看按这种方法对上面数据分配一遍的情况，如图 9.9 所示。这时将记录顺序收集，各关键码的最后一位分段递

0:	$(0,1,1) (0,3,2) (0,2,0) (0,1,3) (0,2,2) (0,3,0)$
1:	$(1,3,2) (1,0,3) (1,1,0)$
2:	$(2,2,0) (2,0,2) (2,1,1) (2,0,1) (2,3,1) (2,1,3)$
3:	$(3,1,3) (3,0,1) (3,1,0) (3,3,1) (3,1,1)$

图 9.8 按最高位做一次分配

增。如果再按关键码的中间元素分配和收集，只要注意维持稳定性，在得到的每个分段中（中间位相同），仍然可以保证最后一位递增。图 9.10 给出了这样分配得到的状况。将图中序列顺序收集，得到的序列是按后两位递增排序的。易见，再针对最高位做一遍分配和收集，就能得到按整个三位排序的关键码序列。

0:	(2,2,0)	(0,2,0)	(1,1,0)	(0,3,0)			
1:	(0,1,1)	(3,0,1)	(2,1,1)	(2,0,1)	(2,3,1)	(3,3,1)	(3,1,1)
2:	(2,0,2)	(0,3,2)	(0,2,2)				
3:	(3,1,3)	(1,0,3)	(0,1,3)	(2,1,3)			

图 9.9 按最低位做一次分配

0:	(3,0,1)	(2,0,1)	(2,0,2)	(1,0,3)			
1:	(1,1,0)	(0,1,1)	(2,1,1)	(3,1,1)	(3,1,3)	(0,1,3)	(2,1,3)
2:	(2,2,0)	(0,2,0)	(0,2,2)				
3:	(0,3,0)	(2,3,1)	(3,3,1)	(0,3,2)			

图 9.10 按中间位做一次分配

由上面分析和实例可见，采用最低位优先方式，处理过程的实现更加规范而简单。下面考虑这种方法的实现。

如果每位关键码都是数字，上述关键码元组就像是按某种进制（以某个数为基数）表示的一个整数，排序过程中就是从低到高逐位进行分配和收集。这样的处理过程就像是按基数逐位处理，因此这种多轮分配排序也被称为基数排序。

算法设计与实现

现在考虑基数排序在 Python 中的一种简单实现。假设：

- 需要排序的仍然是 Record 类型的顺序表，即 Python 的 list。
- 记录中的关键码是十进制数字的元组，包含 r 个元素。
- 排序算法的参数是表 lst 和关键码元组长度 r 。

为实现基数排序过程，算法中需要一组记录桶，而且需要稳定的分配和收集。考虑采用一种简单实现方法，用一组 10 个 list 作为记录桶。显然，如果把这种桶放入一个顺序表，就可以用关键码的位作为桶的下标，直接索引相应的桶。

下面是基于这一设计描述的算法：

```
def radix_sort(lst, d):
    rlists = [[] for i in range(10)]
    llen = len(lst)
    for m in range(-1, -d - 1, -1):
        for j in range(llen):
            rlists[lst[j].key[m]].append(lst[j])
        j = 0
        for i in range(10):
            tmp = rlists[i]
            for k in range(len(tmp)):
                lst[j] = tmp[k]
                j += 1
            rlists[i].clear()
```

这里给出一些解释：

- 1) rlists 是包含 10 个记录桶的顺序表，初始状态中所有的桶为空。
- 2) 作为函数主体的大循环从关键码的最低位 ($x.key[-1]$) 开始工作，逐位向上，直至完成排序（做到 $-d$ 为止）。
- 3) 第一个内层循环完成一轮分配，循环体只有一个语句，每次执行把一个记录附加在由关键码位确定的桶（顺序表）的最后。
- 4) 第二个内层循环完成收集工作，顺序处理各个桶里的记录，将它们转存入原来的顺序表。

表。这里用一个临时变量 `tmp` 引用被处理的桶，既为了描述方便，也能避免反复索引一个表元素的开销。

由于 Python 中的变量保存的是对象引用，在这个算法里并没有做实际的元素复制操作，所做的都是记录引用赋值。

算法分析及其他

在下面分析中， n 表示被排序表的长度， r 表示基数， d 表示关键码元组长度。

上述算法里用了一个表的表作为临时存储，其空间占用情况依赖于 Python 中 `list` 的 `clear` 操作的实现方法。如果 `clear` 简单地为表换一块空表存储区，上面算法的空间占用将不超过 $O(n)$ 。如果 `clear` 直接把表的元素长度记录置 0，上面算法的空间复杂度将可能达到 $O(m \times n)$ 。其中最坏情况是所有记录的关键码元组相同，并因此在不同分配轮中分配到不同的桶。考虑前面例子，如果所有记录的关键码都是 $(2, 1, 0)$ ，在三次分配中，全部记录依次被分配到三个表里，使这三个表的长度都增长到可以容纳 n 个元素。

考虑算法的空间复杂度。显然，外层循环需要执行 d 次，给复杂度引进了一个因子 d 。第一个内层循环执行 n 次迭代。第二个内层循环中的小循环做实际收集，其循环体的总执行次数不超过 $O(n)$ 。但无论如何，第二个内层循环也要执行 r 次迭代。综合这些情况，算法的时间复杂度可以描述为 $O(d \times (n + \max(n, r))) = O(d \times (n + r))$ 。

基数排序算法的另一种常见技术是用链接表实现记录桶，如果那样做就可以摆脱 Python 中 `list` 操作实际实现方式的影响（空间复杂度不再依赖于 Python 表操作的实现）。此外，用链表作为记录的存储桶，收集时可以简单地取下链表结点链，将它们顺序连接起来，因此可以节省时间。有关实现请读者自己考虑。

9.5.2 一些与排序有关的问题

现在讨论一些与排序有关的问题，包括一些实际问题。

混成方法

各种排序方法也可以从另一个角度分为两大类，一类是比较简单的排序方法，以简单插入排序、起泡排序等为代表；另一类是比较复杂的排序方法，如快速排序、归并排序、堆排序等。两类方法各有特点，例如：

- 简单排序方法实现简单，但理论的时间复杂度高。从实际角度看，如果只需要对很短的序列排序，采用简单排序方法就足够快了。
- 复杂排序方法的实现比较复杂，但理论的时间复杂度低。如果需要排序的序列很长，一定要使用某种高效的排序算法，以保证程序的效率。

然而，实际排序程序也完全可以采用多种算法的组合，下面将之称为混成方法。如果仔细观察，不难发现在一些复杂排序算法里，最后也需要处理较短序列的排序问题。快速排序和归并排序是这方面的典型：

- 在快速排序算法中，序列被划分为越来越短的片段。如果序列已经很短，例如短于十几个元素，快速排序还要做几次递归调用（或进栈、退栈）。这些辅助操作也耗时，体现在复杂度描述中忽略了的常量因子，实际上也需要付出一定时间。对很短的序列，采用简单插入排序，实际效果很可能优于采用快速排序。
- 归并排序的工作顺序正好与快速排序相反，是从较短的有序序列归并出越来越长的序

列。这里就有作为归并基础的问题。从 8 个各自包含一个元素的序列出发，通过几遍归并得到包含 8 个元素的有序序列，其中也需要做许多函数调用（按前面实现）。与对这 8 个元素做简单插入排序相比，采用归并排序的实际时间消耗可能更多。

由于这些情况，实际程序里的排序功能，特别是各种程序库里的排序函数，通常都不是纯粹地采用一种算法，而是使用两种或两种以上方法的组合。最常见的是归并排序和插入排序的组合，以及快速排序和插入排序的组合等。

采用多种排序算法，就出现了算法之间的“切换”问题，需要决定在什么样的条件下从一种方法切换到另一种方法。简单情况如上面的讨论，以序列的长度作为依据，其中最佳切换点的选择通常需要做一些试验。

这方面的问题不进一步讨论了。下一小节将简单介绍 Python 官方实现解释器中采用的排序方法，它就是一种混成排序方法。

稳定性问题

前面讨论了稳定性对于排序的重要意义。应该看到，需要考虑稳定性的根源是序列中可能出现关键码相等的记录，如果不存在关键码相等的情况，所用的排序算法是否稳定就不是需要考虑的问题了。

基于前面的讨论和上面看法，还可以发现，实际上，任何一种排序算法都可以改造为一种稳定的排序算法，方法是：

- 修改序列中各个记录的关键码，用序对 (key_i, i) 作为记录 R_i 的关键码，其中 key_i 是 R_i 原来的关键码， i 是 R_i 在被排序序列中的位置（下标）。
- 按照字典序，采用新关键码对序列排序。也就是说，在比较记录的关键码时，首先按原关键码的比较方式考量两个记录的 key 部分，如果能确定顺序则已有结论，在 key 部分相同时比较 i 部分，确定从小到大的顺序。

采用这种方法，原来关键码不同的记录依然会按照原顺序排序。原来关键码相同的记录仍然被集中在一起，但将按照它们在原序列中的顺序排序。显然得到了稳定性。从另一个角度看，这一排序中使用的新关键码具有唯一性。

上述改造方法具有普适性，适合任何算法，但也付出了代价：需要扫描被排序序列，修改记录的关键码，排序中比较关键码的开销也有增加。更重要的是需要为每个记录增加一个序列下标项，相关的空间开销是 $O(n)$ 。

9.5.3 Python 系统的 list 排序

Python 系统有一个内置的排序函数 `sort`，可以对任何可迭代对象排序，得到一个排序的表。另外，表 `list` 类的对象也有一个 `sort` 方法。其实两者共享同一个排序算法，这是一种混成式排序算法，称为 Timsort，可译为蒂姆排序。

基本情况

蒂姆排序是一种基于归并技术的稳定排序算法，其中结合使用了归并排序和插入排序技术，最坏时间复杂度是 $O(n \log n)$ 。该算法具有适应性，在被排序的数组元素接近排好序的情况下，它的时间复杂度可能远小于 $O(n \log n)$ ，有可能达到线性时间。蒂姆排序算法在最坏情况下需要 $n/2$ 工作空间，因此其空间复杂度是 $O(n)$ 。但另一方面，如果情况比较有利，它只需要很少临时存储空间。

蒂姆排序算法比较适合许多实际应用中常见的情况，特别是被排序的数据序列分段有序或者基本有序，但其中也有些非有序元素的情况。这一算法是在 Python 语言和系统的开发过程中，由 Tim Peters 在 2002 年设计和开发的。人们通过许多试验（使用随机数据或实际数据），结论是蒂姆排序在平均性能上超过快速排序，是目前实际表现最好的排序算法。虽然它未能从理论上克服归并排序 $O(n)$ 空间开销的弱点，但实际上经常不需要很大工作空间。由于这些情况，蒂姆排序已被另外一些重要软件采纳。例如 Java SE7（第 7 标准版本）已经改用蒂姆排序（虽然它与原算法不完全兼容，也就是说，不保证排序结果相同）；Android 平台和 GNU 的开源数值语言 Octave 等也采用蒂姆排序。

算法概览

这里简单介绍蒂姆排序的基本思想。其主要优势是克服了归并排序没有适应性的缺陷，又保持了其稳定性的特征，并尽可能利用实际数据的情况。

蒂姆排序的基本工作过程和方式如下：

- 考察待排序序列中非严格单调上升（后一记录大于等于前一记录）或严格单调下降（后一记录严格小于前一记录）的片段，反转其中的严格下降片段。
- 采用插入排序，对连续出现的几个特别短的上升序列排序，使整个序列变成一系列（非严格）单调上升的记录片段，每个片段都长于某个特定值。
- 通过归并产生更长的排序片段，控制这一归并过程，保证片段的长度尽可能均匀。归并中采用一些策略，尽可能地减少临时空间的使用。

通过反复归并，最终得到排序序列。

这里不再进一步讨论蒂姆排序的技术细节。关心这方面情况的读者可以从网络上找到很多信息，也可以查看各种公开的蒂姆排序实现代码。

在该算法的发展过程中还出现了一个有趣的插曲。有研究者希望用形式化方法严格证明该算法，但发现无法完成这个证明。通过仔细研究，发现 Tim Peters 的蒂姆排序算法存在错误（2015 年 2 月报告），这一算法的所有实现都存在同样错误。经过十几年的开发，以及许多人的仔细检查、大量测试和长期使用，都没发现这个错误。这个错误只会出现在序列长度超过 2^{49} 的情况下，而目前实际计算机还不能处理如此大的序列（天河 2 号的存储器大约为 2^{50} 字节）。这一事件一方面说明了保证软件正确可靠何等困难，同时也说明了严格的数学证明在提高软件可靠性方面的潜力。该错误已经被更正。

本章总结

这里先给出几种排序算法的理论比较，再讨论一些问题。

几种排序算法的比较

下表总结了本章中讨论的主要排序算法的时间和空间复杂度，以及稳定性和适应性方面的情况。研究者已经证明，基于关键码比较的排序时间复杂性下界为 $O(n \log n)$ 。因此，最理想的排序算法是 $O(n \log n)$ 时间、 $O(1)$ 空间、稳定，最好还具有适应性。现在还不知道是否存在这种算法，但至今没有找到！

从下表中还可以看出，一些算法在一些指标上达到最优情况。另外，虽然一些算法的复杂度相同，在实践中的表现也可能有差异。

排序算法	最坏情况 时间复杂度	平均情况 时间复杂度	最好情况 时间复杂度	空间复杂度	稳定性	适应性
简单插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	是	是
二分插入排序	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(1)$	是	是
表插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	是	是
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否	否
堆选择排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	否	否
起泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	是	是
快速排序	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	否	否
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	是	否
蒂姆排序	$O(n \log n)$	$O(n \log n)$	$O(n)$	$O(n)$	是	是

其中二分插入排序的最好情况是做了 n 次检索但不需要移动元素。

一些讨论

从平均时间上看：在实践中快速排序法的速度非常快，但其最坏情况具有平方复杂度，不如归并和堆排序。序列长度 n 较大时归并排序通常比堆排序更快，但其实现需要很大的辅助空间。但发展到今天，蒂姆排序在实际应用中处于最优地位，统计数据说明它优于其他排序，而且还具有稳定性。其缺点是 $O(n)$ 的辅助空间。

在各种简单朴素的排序算法中，直接插入排序最简单。如果序列中的记录基本有序而且序列长度 n 比较小，就应优先选用直接插入排序算法。这种算法还经常与其他排序方法结合使用，例如，实用的快速排序程序在划分得到很短的分段时通常转用插入排序，归并排序也常用插入算法建立初始的有序序列，蒂姆排序也用插入排序做前期处理。

如果待排序序列接近有序，直接插入排序算法能很快完成排序工作，也就是说具有适应性。起泡排序也比较快，但可能受到小记录与最终位置距离的困扰。高效排序算法中只有蒂姆排序具有适应性，其实现中也利用了简单插入排序。

简单排序算法多是稳定的，但大部分时间性能好的排序都不稳定，如快速排序、堆排序等。一般来说，如果排序过程中只做相邻记录的关键字码比较和局部调整，通常能得到稳定的排序方法。在高效的排序算法中，只有归并排序能很自然地得到稳定性。蒂姆排序的最主要操作也是归并，它结合了其他技术，得到了很好的效果。应注意，稳定性是具体算法实现的性质，采用同一种排序方法，有可能做出稳定的和不稳定的实现。前面讨论中曾以插入排序和选择排序为例，说明了这方面的情况。但有些算法的实现可以很自然地做到稳定（如插入排序、归并排序），另一些则需要附加的时间或空间开销（如选择排序等）。

在实际应用中，数据记录通常有一个主关键码，例如各种唯一标示码，如学号、身份证编号、用户账号、商品号等。这种关键码一般都具有唯一性。这样，如果需要做的是按主关键字排序，所用排序方法是否稳定就无关紧要了。但在实际中，也经常需要把记录中的其他成分作为排序码使用，例如按学生的姓名、籍贯、年龄、成绩等排序。在做这种排序时，就应该根据问题所需慎重选择排序方法，经常需要用稳定算法。如果用了不稳定的排序算法，可能还需要对具有相同关键码的记录段再次排序。

今天，各种主要语言大都（有可能是通过语言的标准库）提供了排序函数或其他排序功能，

所用的算法一般都具有 $O(n \log n)$ 的平均时间复杂性。但可能有些排序功能具有 $O(n^2)$ 最坏时间复杂度（例如其实现采用快速排序算法），一些排序函数采用的排序算法不稳定（如采用快速排序）。在使用语言或库的排序功能时，应该关注其各方面的特点，确认其功能是否满足实际应用需要。

练习

一般练习

1. 复习下面概念：排序问题，全序关系，反对称关系，等价类，排序方法（排序算法），堆排序算法，基于关键码比较的排序，排序码，内排序，外排序，外排序算法，归并排序算法，排序中的基本操作（关键码比较和数据记录移动），最优排序算法，原地排序算法，稳定性，适应性，插入排序，选择排序，交换排序，分配排序，直接插入排序算法，二分法插入排序算法，shell 排序算法，直接选择排序算法，树形选择，交换和排序，逆序，起泡排序算法，交错起泡排序，快速排序，划分，归并排序算法，归并操作，二路归并，分配排序，多轮分配和收集，高位优先方法，低位优先方法，基数排序，混成排序方法，蒂姆排序算法，主关键码，其他关键码。
2. 设有 $N > 50000$ 个元素的序列（表），元素可以比较大小，但排列没有任何顺序保证。现希望选出其中最大的 100 个元素，或希望找到其中从大到小排在第 100 位的元素，怎样能最快找到这个元素？操作的复杂性如何？
3. 现在有 n 个整数关键码，需要把其中的负值调整到非负值之前。请设计一个算法完成这一工作，要求：只使用 $O(1)$ 空间，时间复杂度是 $O(n)$ 。请分析你的算法在执行中总共需要做多少次关键码比较。
4. 假设现在要处理的数据项的关键码是整数，请考虑一种算法，它能最快地将关键码小于 -10 的项移到左边，关键码大于 10 的项移到右边，中间是关键码介于 -10 和 10 之间的数据项。你的算法应该在线性时间完成工作，而且要尽量快速。
5. 假设有 n 个不同的关键码，希望找到其中最大的关键码和最小的关键码。最直截了当的方法需要做 $2n - 3$ 次关键码比较。请设计一种方法，不需要做 $2n - 3$ 次比较就能找出所需的值。你的算法需要做多少次比较？
6. 数据序列（数据集）的中位值是统计中的重要概念。请设计一个算法，它能在线性时间内找到一组整数的中位值。

编程练习

1. 请用一组随机生成的数据试验本章讨论的几个排序算法，关键码使用某个范围内的整数表示。分析得到的试验数据，并对其做一些总结。
2. 请定义一个插入排序算法，让它在原序列的高端积累已排序的元素。
3. 请采用二分法在插入排序中找到插入位置，而后再实际插入元素。请分析所做的算法，特别关注其稳定性。如果它不稳定，请设法修改使之稳定。
4. 请定义一个选择排序函数，它每次选择剩余记录中最大的记录，完成从小到大递增顺序的排序工作。
5. 请实现一个稳定的选择排序算法。

6. 9.2.3 节最后提出了一种交错起泡的排序技术。请定义一个采用这种技术的排序函数，并用随机生成的表做一些试验，比较它和简单起泡排序的性能。
7. 请为第 3 章的单链表类定义一个采用选择排序思想的排序方法。
8. 请为第 3 章的单链表类定义一个采用快速排序思想的排序方法。
9. 请实现采用“三者取中”策略的快速排序函数，用随机生成的表作为实例，比较采用这种策略的函数和本章中的快速排序函数。
10. 请提出另一种改进划分标准的方法，基于该方法实现一个快速排序函数，并将实现的函数与正文中的快速排序函数比较。
11. 请实现一个非递归的快速排序算法，算法在选择处理分段时采用本章中提出的相关方法，保证其空间需求达到最少。
12. 考虑 9.5.1 节提出的基于绩点对学生记录排序的工作。请根据具体情况做出一种设计和实现，保证排序工作可以在 $O(n)$ 时间完成 (n 是学生记录数)。
13. 9.5.1 节最后说基数排序算法的另一种常见技术是用链接表实现桶，那样做可以摆脱 Python 中 list 操作的潜在影响（空间复杂度不再依赖于 Python 表操作的实现技术）。用链表作为记录存储桶，收集时可以简单地取下链表结点链，顺序连接起来，因此可以节省时间。请采用这种技术实现一个基数排序函数。
14. 请考虑 9.5.2 节提出的组合方法，选择一种组合方式实现相应的排序函数，并通过一些试验确定合适的方法转换时机。
15. 请基于蒂姆排序算法实现一个排序函数。

封面

书名

版权

前言

目录

第1章 绪论

1.1 计算机问题求解

1.1.1 程序开发过程

1.1.2 一个简单例子

1.2 问题求解：交叉路口的红绿灯安排

1.2.1 问题分析和严格化

1.2.2 图的顶点分组和算法

1.2.3 算法的精化和Python描述

1.2.4 讨论

1.3 算法和算法分析

1.3.1 问题、问题实例和算法

1.3.2 算法的代价及其度量

1.3.3 算法分析

1.3.4 Python程序的计算代价（复杂度）

1.4 数据结构

1.4.1 数据结构及其分类

1.4.2 计算机内存对象表示

1.4.3 Python对象和数据结构

练习

第2章 抽象数据类型和Python类

2.1 抽象数据类型

2.1.1 数据类型和数据构造

2.1.2 抽象数据类型的概念

2.1.3 抽象数据类型的描述

2.2 Python的类

2.2.1 有理数类

2.2.2 类定义进阶

2.2.3 本书采用的ADT描述形式

2.3 类的定义和使用

2.3.1 类的基本定义和使用

2.3.2 实例对象：初始化和使用

2.3.3 几点说明

2.3.4 继承

2.4 Python异常

2.4.1 异常类和自定义异常

- 2.4.2 异常的传播和捕捉
- 2.4.3 内置的标准异常类
- 2.5 类定义实例：学校人事管理系统中的类
 - 2.5.1 问题分析和设计
 - 2.5.2 人事记录类的实现
 - 2.5.3 讨论
- 本章总结
- 练习

第3章 线性表

- 3.1 线性表的概念和表抽象数据类型
 - 3.1.1 表的概念和性质
 - 3.1.2 表抽象数据类型
 - 3.1.3 线性表的实现：基本考虑
- 3.2 顺序表的实现
 - 3.2.1 基本实现方式
 - 3.2.2 顺序表基本操作的实现
 - 3.2.3 顺序表的结构
 - 3.2.4 Python的list
 - 3.2.5 顺序表的简单总结
- 3.3 链接表
 - 3.3.1 线性表的基本需要和链接表
 - 3.3.2 单链表
 - 3.3.3 单链表类的实现
- 3.4 链表的变形和操作
 - 3.4.1 单链表的简单变形
 - 3.4.2 循环单链表
 - 3.4.3 双链表
 - 3.4.4 两个链表操作
 - 3.4.5 不同链表的简单总结
- 3.5 表的应用
 - 3.5.1 Josephus问题和基于“数组”概念的解法
 - 3.5.2 基于顺序表的解
 - 3.5.3 基于循环单链表的解

- 本章总结

- 练习

第4章 字符串

- 4.1 字符集、字符串和字符串操作
 - 4.1.1 字符串的相关概念
 - 4.1.2 字符串抽象数据类型
- 4.2 字符串的实现

- 4.2.1 基本实现问题和技术
- 4.2.2 实际语言里的字符串
- 4.2.3 Python的字符串
- 4.3 字符串匹配（子串查找）
 - 4.3.1 字符串匹配
 - 4.3.2 串匹配和朴素匹配算法
 - 4.3.3 无回溯串匹配算法（KMP算法）
- 4.4 字符串匹配问题
 - 4.4.1 串匹配 / 搜索的不同需要
 - 4.4.2 一种简化的正则表达式
- 4.5 Python正则表达式
 - 4.5.1 概况
 - 4.5.2 基本情况
 - 4.5.3 主要操作
 - 4.5.4 正则表达式的构造
 - 4.5.5 正则表达式的使用

本章总结

练习

第5章 栈和队列

- 5.1 概述
 - 5.1.1 栈、队列和数据使用顺序
 - 5.1.2 应用环境
- 5.2 栈：概念和实现
 - 5.2.1 栈抽象数据类型
 - 5.2.2 栈的顺序表实现
 - 5.2.3 栈的链接表实现
- 5.3 栈的应用
 - 5.3.1 简单应用：括号匹配问题
 - 5.3.2 表达式的表示、计算和变换
 - 5.3.3 栈与递归
- 5.4 队列
 - 5.4.1 队列抽象数据类型
 - 5.4.2 队列的链接表实现
 - 5.4.3 队列的顺序表实现
 - 5.4.4 队列的list实现
 - 5.4.5 队列的应用
- 5.5 迷宫求解和状态空间搜索
 - 5.5.1 迷宫求解：分析和设计
 - 5.5.2 求解迷宫的算法
 - 5.5.3 迷宫问题和搜索

5.6 几点补充

5.6.1 几种与栈或队列相关的结构

5.6.2 几个问题的讨论

本章总结

练习

第6章 二叉树和树

6.1 二叉树：概念和性质

6.1.1 概念和性质

6.1.2 抽象数据类型

6.1.3 遍历二叉树

6.2 二叉树的list实现

6.2.1 设计和实现

6.2.2 二叉树的简单应用：表达式树

6.3 优先队列

6.3.1 概念

6.3.2 基于线性表的实现

6.3.3 树形结构和堆

6.3.4 优先队列的堆实现

6.3.5 堆的应用：堆排序

6.4 应用：离散事件模拟

6.4.1 通用的模拟框架

6.4.2 海关检查站模拟系统

6.5 二叉树的类实现

6.5.1 二叉树结点类

6.5.2 遍历算法

6.5.3 二叉树类

6.6 哈夫曼树

6.6.1 哈夫曼树和哈夫曼算法

6.6.2 哈夫曼算法的实现

6.6.3 哈夫曼编码

6.7 树和树林

6.7.1 实例和表示

6.7.2 定义和相关概念

6.7.3 抽象数据类型和操作

6.7.4 树的实现

6.7.5 树的Python实现

本章总结

练习

第7章 图

7.1 概念、性质和实现

- 7.1.1 定义和图示
- 7.1.2 图的一些概念和性质
- 7.1.3 图抽象数据类型
- 7.1.4 图的表示和实现
- 7.2 图结构的Python实现
 - 7.2.1 邻接矩阵实现
 - 7.2.2 压缩的邻接矩阵（邻接表）实现
 - 7.2.3 小结
- 7.3 基本图算法
 - 7.3.1 图的遍历
 - 7.3.2 生成树
- 7.4 最小生成树
 - 7.4.1 最小生成树问题
 - 7.4.2 Kruskal算法
 - 7.4.3 Prim算法
 - 7.4.4 Prim算法的改进
 - 7.4.5 最小生成树问题
- 7.5 最短路径
 - 7.5.1 最短路径问题
 - 7.5.2 求解单源点最短路径的Dijkstra算法
 - 7.5.3 求解任意顶点间最短路径的Floyd算法
- 7.6 AOV/AOE网及其算法
 - 7.6.1 AOV网、拓扑排序和拓扑序列
 - 7.6.2 拓扑排序算法
 - 7.6.3 AOE网和关键路径
 - 7.6.4 关键路径算法
- 本章总结
- 练习

第8章 字典和集合

- 8.1 数据存储、检索和字典
 - 8.1.1 数据存储和检索
 - 8.1.2 字典实现的问题
- 8.2 字典线性表实现
 - 8.2.1 基本实现
 - 8.2.2 有序线性表和二分法检索
 - 8.2.3 字典线性表总结
- 8.3 散列和散列表
 - 8.3.1 散列的思想和应用
 - 8.3.2 散列函数
 - 8.3.3 冲突的内消解：开地址技术

- 8.3.4 外消解技术
- 8.3.5 散列表的性质
- 8.4 集合
 - 8.4.1 集合的概念、运算和抽象数据类型
 - 8.4.2 集合的实现
 - 8.4.3 特殊实现技术：位向量实现
- 8.5 Python的标准字典类dict和set
- 8.6 二叉排序树和字典
 - 8.6.1 二叉排序树
 - 8.6.2 最佳二叉排序树
 - 8.6.3 一般情况的最佳二叉排序树
- 8.7 平衡二叉树
 - 8.7.1 定义和性质
 - 8.7.2 AVL树类
 - 8.7.3 插入操作
 - 8.7.4 相关问题
- 8.8 动态多分支排序树
 - 8.8.1 多分支排序树
 - 8.8.2 B树
 - 8.8.3 B + 树

本章总结

练习

第9章 排序

- 9.1 问题和性质
 - 9.1.1 问题定义
 - 9.1.2 排序算法
- 9.2 简单排序算法
 - 9.2.1 插入排序
 - 9.2.2 选择排序
 - 9.2.3 交换排序
- 9.3 快速排序
 - 9.3.1 快速排序的表实现
 - 9.3.2 程序实现
 - 9.3.3 复杂度
 - 9.3.4 另一种简单实现
- 9.4 归并排序
 - 9.4.1 顺序表的归并排序
 - 9.4.2 归并算法的设计问题
 - 9.4.3 归并排序函数定义
 - 9.4.4 算法分析

9.5 其他排序方法

9.5.1 分配排序和基数排序

9.5.2 一些与排序有关的问题

9.5.3 Python系统的list排序

本章总结

练习

参考文献