SECOND EDITION

problem solving with algorithms and data structures using python

北京大学地空学院
数据结构与算法
SESSDSA 内部教材

Bradley N. Miller
David L. Ranum

FRANKLIN, BEEDLE & ASSOCIATES INCORPORATED
[INDEPENDENT PUBLISHERS SINCE 1985]

主讲教师：陈斌 gischen@pku.edu.cn

课程网站：http://gis4g.pku.edu.cn/course/pythonds

问题求解：算法与数据结构

（Python 版）

目录

* * 声明 * *

本教材中文版由

北京大学地空学院数据结构与算法课程

2015年及2016年上课同学

志愿翻译整理而成

英文原版教材地址：

http://interactivepython.org/runestone/static/pythonds/index.html

致谢

2015地空数算课程教材翻译编辑小组

组别	任务	小组成员
1	Ch1	柳晓萱（组长）、刘明辰、杨润 、杨礼萌、汪颖、党卓、汪诗舜、陈春含、李子涵、蓝坤
2	Ch2 ~ 3.3	苏瑞冰（组长）、蒋久阳、张虎来、胡哲、朱贺、宋欣源、葛天雨、钟涛、周杰、黄佳旺、付帆飞、张沙洲。
3	3.4 ~ 4.4	高鸿宇（组长）、汪诗舜、廖宸睿、秦树健、韦春婉、李然、刘志扬、韩甲源、阎述辰、李嘉琪、陈春含、刘杰。
4	4.5 ~ Ch5	庞磊（组长）、于润泽、蒋明轩、彭玉恒、武化雨、贾博、李昆鹏、陈哲萌、赵琰喆、温景充、余晓辉、龚世泽。
5	Ch6	张虎来（组长）、于润泽、余晓辉、宋欣源、汪建峰、高鸿宇、周安、黄佳旺、卢思奇、刘明辰、张沙洲。
6	Ch7	刘松吟（组长）、曹越、汪诗舜、石永祥、韩甲源、段鉴书、赵玖桐、陈哲萌、陈春含、张子玄、庞磊。

致谢

2016地空数算课程教材翻译编辑小组

组别	任务	小组成员
1	Ch1	张书莞（组长）、华思博、孟浩瀚、于曦彤、卢伟杰、耿晓状、李姿蓉、韦庆朗、江欣余、詹煌、刘宇飞、杨子浩
2	Ch2 ~ 3.3	孙新然（组长）、郑毅权、朱金顺、许午川、方先君、陈一潇、赵泽严、李宣孛、张家港、薛莅治、杨昕陶、王宁
3	3.4 ~ 4.4	秦珺峰（组长）、周哲、崔博、濮心翼、万紫荆、尹泽藩、赵旭炜、李京寰、黄杰、周强、黄鑫、张二禾
4	4.5 ~ Ch5	苏克凡（组长）、池昱霖、齐厚基、廖丝丝、凌坤、苏培臻、郭惠昀、杨帆、刘旭林、朱英杰、徐运铎、王琦
5	Ch6	卢国军（组长）、杨江南、赵宸兴宇、邬科元、邵锐成、陈喆、高红涛、夏运、毋轩琦、薛阶祺、李银、陈相
6	Ch7	刘晨（组长）、卢亚敏、武于靖、庞骁、张浩源、刘立超、胡圣懿、项楷、周思阳、孔淑媛、李逸飞、王召平

一. 引言

1.1. 目标

了解计算机科学、程序设计和问题解决的基本概念；

理解什么是“抽象”以及抽象在问题解决过程中的作用；

理解“抽象数据类型”的概念以及在实际操作中学会运用；

学习Python程序设计语言。

1.2. 开始学习

自从第一台需要人们用线缆和交换机向其传达指令的电子计算机问世以来，编程已发生了巨大改变。计算机科技的革新给计算机科学家提供了越来越多的工具和平台，方便他们在社会的各个领域一显身手。高速处理器、高速网络以及大容量存储器等在发展的同时也带来了难度螺旋式上升的种种问题，而计算机科学家们必须解决它们。发展固然很快，众多基本原则却能经久不变。计算机科学本质上是利用计算机来解决问题。

无疑你已经花费了大量时间学习解决问题所需要的基本能力，我们也相信你对自己理解问题并提出解决方案的能力已很有自信。你应当已经感受到了编写程序代码常常是具有相当难度的。然而，往往正是大型问题的繁杂，加上其解决方案与之相当的复杂性掩盖了解决问题的过程中涉及到的基本思路。

本章接下来将主要强调两个重要的方面。第一，回顾计算机科学以及算法与数据结构的基本框架，并特别强调我们学习这些内容的原因以及理解这些内容如何帮助我们更好地解决问题。第二，了解python语言。尽管无法提供详尽透彻的指导，本章将给出具体案例并对其余章节中将会涉及到的基本概念和思想做出解释。

1.3. 计算机科学是什么

通常来说计算机科学难以准确地被定义。这或许是因为人们对“计算机”这个词的滥用。你也许清楚，计算机科学并不仅仅是对计算机进行研究。虽然计算机是支撑这门学科的重要工具，但它也仅仅是工具而已。

计算机科学是对问题本身、问题的解决、以及问题求解过程中得出的解决方案的研究。面对一个特定问题，计算机科学家的目标是得出一个**算法（algorithm）**，写出一组解决该问题可能出现的任何情况的步步为营的指令。算法通过有限过程解决问题。算法是解决方案。

计算机科学可以被看作是对算法的研究。然而，我们必须小心翼翼地考虑到有一些问题是没有解决方案的。虽然证明这个说法超出了本课本的范围，有些问题无法解决这件事对研究计算机科学的人来说很重要。通过涵盖这两类问题，我们可以完整地定义计算机科学，计算机科学研究的是问题的解决方案以及没有解决方案的问题。

描述问题及其解决方案时，**“可计算”（computable）**这个词是很常见的。当存在解决某个问题的算法时，我们说该问题是可计算的。另一种对计算机科学的定义是:计算机科学研究的是问题是否可计算，算法是否存在。在大多数情况下，你将会注意到“计算机”这个词完全没有出现。我们认为问题的解决方案是独立于机器本身的。

计算机科学，就如它关注问题求解过程一样，也研究**抽象（abstraction）**。抽象使我们能以一种区分所谓的逻辑对象和物质对象的方式来看待问题及其解决方案。在常见的案例中这种基本想法对我们来说是很熟悉的。

考虑一下你今天去学校或工作时乘坐的交通工具。作为一个司机，一个汽车的使用者，为了使汽车达到预期目的，你和汽车之间有固定的交互方式。你坐进车里，插入钥匙，发动汽车，换挡，刹车，加速，行驶。用抽象的角度来看，我们可以说，你看到的是汽车逻辑性的一面。你使用的是汽车设计者提供的将你从一个位置转移到另一个位置的功能。这些功能有时也被称为**界面（interface）**。

另一方面，汽车修理工有非常不同的观点。他不仅知道如何驾驶汽车，而且知道运行我们认为理所当然的功能的所有必要的细节。他需要了解引擎如何工作，如何传递档位变化，如何控制温度等等。这被认为是物质的角度，发生在引擎盖下的细节。

我们使用计算机的时候也是一样的。大多数人用计算机写文档，收发邮件，上网，播放音乐，储存照片以及玩游戏，但他们对于这些程序具体是如何运行的一无所知。他们从逻辑上或是用户的角度看待计算机。计算机科学家、程序员、技术员以及系统管理员又持有一种对于计算机截然不同的看法。

他们必须知道操作系统具体是如何工作的，网络协议是如何配置的以及如何写各种代码来控制这些功能。他们必须能够控制那些用户认为是理所应当的底层的详细内容。

这两个例子相同的一点是：这些抽象化内容的用户，有时也叫做客户，不需要知道程序的详细内容，他们只需要知道界面是如何工作的就可以了。界面就是我们作为用户与在底层的复杂的算法实现过程交流的途径。再来看一个抽象化的例子——以Python 的`math` 模型为例。只要我们引用了这个模型，我们就可以进行下面的计算：

```
>>>import math
>>>math.sqrt(16)
4.0
>>>
```

这就是一个**过程抽象（procedural abstraction）**的例子，我们不一定要知道平方根是怎么运算的，我们只需要知道这个函数叫什么、如何使用它。如果引用的正确，我们就可以相信这个函数会提供给我们正确的答案。我们知道有人实现了这个问题的解决方式，我们只需要知道怎么用就好了。这有时被叫做过程的“黑箱”观点。我们只简单地描述接口信息(即函数名)，需要提供什么（即参数）以及返回值是什么。而函数实现的细节就被隐藏起来了（见图1.1）。

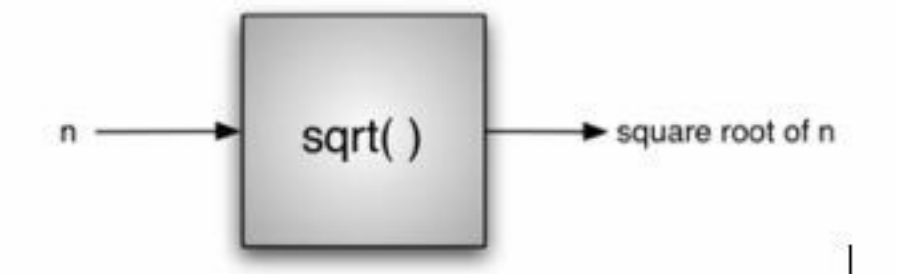


图 1.1 过程抽象

1.4. 什么是程序设计

程序设计（Programming）是将算法编码为计算机可执行的表示法或编程语言的过程。虽然如今存在多种编程语言以及多种计算机，但是最重要的第一步还是给出解决问题的方案。没有算法就没有程序。

计算机科学不是研究程序设计的，但程序设计是计算机科学家工作的重要部分。编程通常是我们为自己的解决方案创建一个表达的方式。因此，这种语言表达方式以及创建它的过程成为了这门学科的基本组成部分。

算法描述的是依据代表问题实体的数据和达到预期结果的一系列步骤的问题解决方案。程序设计语言必须提供一种计数性的方式来表达这个过程和这些数据。为此，程序设计语言提供了控制结构和数据类型。

控制结构使得算法的步骤能被简洁而明确地表达。算法至少要求是能够顺序处理、选择决策和迭代的结构。一种语言只要能够提供这几种基本的语句，就可以被用作算法的表达。

在计算机中，所有数据项都用一段二进制数字表示。为了使这些数字能代表数据，我们需要有**数据类型（data types）**。数据类型把这些二进制数据翻译成我们可以理解的、在解决问题中讲得通的内容。这些底层的、预置的数据类型（有时也叫做原始数据类型）是算法发展的基石。

例如很多编程语言都提供了“整型”这一数据类型。此时这一段储存在计算机内存中的二进制数字就被翻译成了整数，并被赋予了我们日常生活中所使用的典型的整数的意义（比如：23，654和-19）。另外，一种数据类型还提供了数据项可以参与的运算。以整型为例，常见的运算就有加、减、乘等。我们希望这个数字类型的数据能够参与这些算术运算中。

我们经常遇到的困难是实际上问题和他们的解决方案都很复杂。这些简单的，由程序设计语言提供的结构和数据类型尽管是可以表达复杂解决方案的，但却不利于我们思考解决方法的这个过程。我们需要一些办法来控制复杂程度并帮助我们创建解决方案。

1.5. 为何要学习数据结构和抽象数据类型

为了管理问题的复杂度和问题处理的进程,计算机科学家使用抽象概念使他们的关注点能够集中在总体框架上而不会在细节问题上纠缠。通过建立问题域的模型，我们可以更好更有效地处理问题。这些模型让我们能够针对问题本身，用一种更一致的方式，去描述我们的算法处理的数据。

早期，我们把对程序的抽象视为一种通过隐藏特定函数的细节让用户或用户可以在更高层次看问题的方法。**数据抽象（data abstraction）**的思想与之相似。**抽象数据类型（abstract data type）**（常简写为ADT），不涉及数据、操作如何被执行，只是关于如何查看数据和许可操作的逻辑性描述。这意味着我们关注的只是数据代表的含义而不是最终运行的过程。通过提出这种抽象概念，我们实现了对数据的**封装（encapsulation）**。这种理念就是通过对执行的数据的封装，使之从用户视野中消失。这就叫做**信息隐藏（information hiding）**。

图1.2 展示了抽象数据的概念及运行机制。用户通过界面实现交互，使用被抽象数据指定的操作。抽象数据类型是与用户交互的外壳。执行的内容被隐藏在更深的一层。用户不关注执行的细节

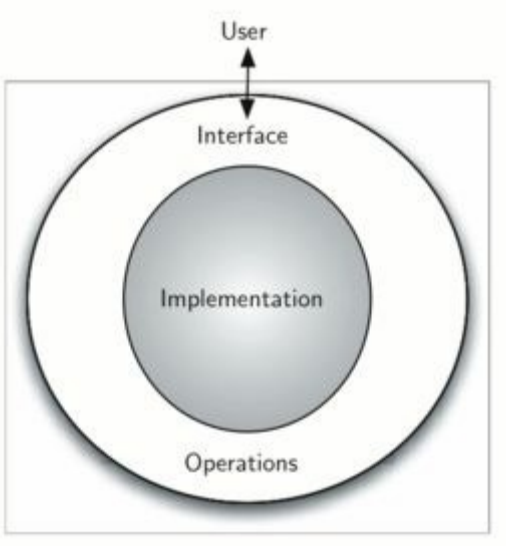


图 1.2 抽象数据类型

抽象数据类型的执行，常被称为**数据结构（data structure）**，它需要我们用一些程序设计和原始数据类型来提供一个关于数据的实际展示。如前文所述，这两种视图的分离可以在不描述模型建立的过程细节的情况下，为所求问题定义一个复杂的数据模型。这给数据提供了一个**独立于现实问题(implementation-independent）**的环境。因为执行一个抽象数据通常有不同的方法，因此执行的独立性便可允许程序员在不改变用户与数据交互方式的前提下，对执行细节进行调整。用户便可将注意力保持在解决问题的过程中。

1.6. 为何要学习算法

计算机科学家们从经验中学习。我们则从他人或自己解决问题的过程中学习。接触不同的解决问题的方法并观察不同的算法设计，帮助我们应对接下来的挑战。通过研究大量不同的算法，我们可以发展出模式识别机制，当以后相似问题出现时，便能更好地解决它。

算法常常各不相同。比如我们之前看到的“`sqrt`”的例子，完全可能有很多不同的方法去实现求平方根函数的计算：这个算法可能比另一个使用少得多的资源；某算法返回结果可能花费其他的算法10倍的时间。人们常想去比较这些解决方法。即使它们都能运行，某一种也可能优于另一种。我们可能会认为，某种更有效，或仅仅是运行更快，或占用更少内存。在研究算法的时候，应基于算法本身的特性比较两种解决方案，而并非基于程序或电脑执行算法的特点去比较。

在最坏的情况下，我们可能会遇到很难对付的问题，也就是说没有算法可以在短时间内解决问题。因此，将可解决问题、不可解决的问题、有解决方案却耗时耗资源的问题区分开来很重要。

识别和选择算法常会带来取舍问题。作为计算机科学家，除了解决问题的能力，我们还需要掌握解决方案评估的技能。最后，一个问题通常有很多解决方法。找到一个方案然后思考它是否是一个好方案将是我们周而复始的任务。

1.7. PYTHON 入门

本章我们将学习编程语言Python，并给出更多关于前文一些想法的详细实例。如果你新接触Python 或者觉得你需要进一步了解前文提出的话题，我们建议你去查阅“Python Language Reference”、“Python Tutorial”等资料。在这里，我们的目标是让你重新熟悉此语言并强化认识一些后续章节的核心概念。

Python 是一种现代化的、易学的、面向对象的编程语言。它拥有一系列强大的内置数据类型和易操作的控制命令。因为Python是一种解释型语言，通过浏览并描述交互式会话的方式，你可以很轻松地对它进行复查。你应该记得解释器显示你熟悉的“`>>>`”提示并评估你给出的Python结构。比如：

```
>>> print("Algorithms and Data Structures")
Algorithms and Data Structures
>>>
```

显示了提示，输出（`print`）函数，结果以及下一个提示。

1.7.1. 从数据开始

我们说过，Python 支持面向对象的编程范式。这意味着Python把数据当做问题解决过程的重点。在Python 里，和很多其他面向对象的编程语言一样，我们定义**“类”（class）**去描述数据的外观（状态）和功能（行为）。“类”类似于抽象数据类型，“类”的用户只能看到数据项的状态和行为。数据项在面向对象的范式里被称为**对象（objects）**。对象是类的一个实例。

1.7.1.1.预置核心数据类型

我们将通过讲述Python 中的基本数据类型开始我们的学习。Python 拥有两个主要的内嵌的有关数值的类，用以记录整数和浮点数。这两个Python中的类被称作int和float。而标准的算术运算符，如+，-，*，/和**（乘方），在使用时，可以通过使用括号来改变其运算顺序。其它的一些非常有用的运算符还有取模，用%实现，和地板除，用//实现。注意如果两个整数相除，其数学上的结果是浮点数，而在Python 中，整数类型的除法运算后只显示商的整数部分，截去了它的小数部分。

```
1 print(2+3*4)
2 print((2+3)*4)
3 print(2**10)
4 print(6/3)
5 print(7/3)
6 print(7//3)
7 print(7%3)
8 print(3/6)
9 print(3//6)
10 print(3%6)
11 print(2**100)
```

代码1.1 基本运算符(intro_1.1)

Python 中用以表达布尔数据类型的bool类，在表达真值的时候非常有用。对于布尔对象，状态变量的可能值为True 或者False，标准的布尔运算符为and , or, not。

```
>>> True
True
>>> False
False
>>> False or True
True
>>> not (False or True)
False
>>> True and True
True
```

布尔型的数据对象常常也被用作比较运算例如等于（==），大于（>）的结果表示。此外，关系运算符和逻辑运算符可以被组合起来解决更复杂的逻辑问题。表1.1中展示了关系算子与逻辑运算符，并在其后对其作用进行了举例说明。

运算符名称	运算符	解释
小于	<	小于运算符
大于	>	大于运算符
小于等于	<=	小于或等于
大于等于	>=	大于或等于
等于	==	相等
不等	!=	不相等
逻辑和	and	两者均为真方为真
逻辑或	or	两者有一为真即为真
逻辑非	not	真变假，假变真

表1.1　关系和逻辑运算符

```
print(5==10)

print(10 > 5)

print((5 >= 1) and (5 <= 10))
```

代码1.2 基本的关系和逻辑运算符(intro_1.2)

标识符就是程序语言中被用来表示名称的符号。在Python 中，标识符以字母或下划线（_）作为开始，大小写敏感，并且可以是任意长度。需要注意的是在命名的时候应该选择有意义的名称，这样才能使你的程序变得更加容易阅读和理解。

在Python 中，当一个名字第一次被用在了赋值语句的左边时，一个变量就随之产生了。赋值语句提供了联系标识符与值的途径。变量会持有一个对数据的引用，而并不是数据本身。参见下面的讲解：

```
>>> theSum = 0

>>> theSum

0

>>> theSum = theSum + 1

>>> theSum

1

>>> theSum = True

>>> theSum

True
```

赋值语句theSum = 0 创建了一个名为theSum 的变量，并且使该变量持有对数据对象0 的引用（见图1.3）。一般来说（如第二个例子），赋值语句的右半部分会经过求值运算，并把对最终结果的引用派给左边的标识符。在我们的例子中，变量及以theSum指向的数据的类型本来是整型，如果数据类型变了（见图1.4）比如上面例子里被赋了一个Bool 型的值True，那么变量类型也改变（即theSum也为不ool型）。这就说明，在Python 中，如果赋值的数据类型变了，那么变量的类型也会

跟着改变。赋值语句改变的是变量所执的引用，这是Python极其灵活的一个特征，所以一个变量可以指向许多种类型的数据。

图 1.3　变量持有对数据的引用

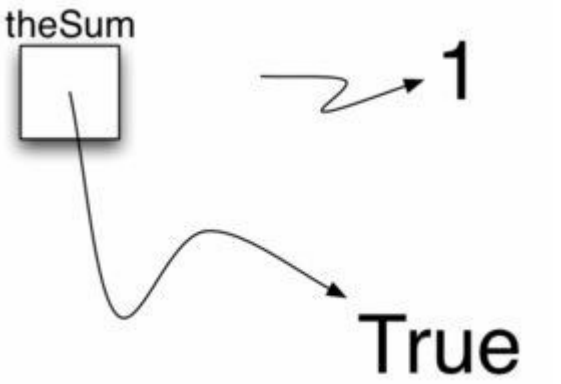


图 1.4 赋值改变了引用

1.7.1.2 预置集合数据类型

除了数值和布尔这两个类，Python还拥有一系列的强大的内嵌的数据容器。列表，字符串和元组都是有序容器，并且在结构上大致相似，但是分别拥有一些在应用中需要去好好理解的特别的属性。字典和集合则都是无序容器。

一个**列表**是包含零个或多个对Python 中数据的引用的有序容器。列表用方括号括起，里面的每个数据用逗号隔开。空的列表是`[]`。列表是异质的，这意味着列表中的数据不必要同一个类，并且列表可以像下面展示的那样被派给一个变量。第一个例子展示了列表中Python 数据的多种多样。

```
>>> [1,3,True,6.5]
[1, 3, True, 6.5]
>>> myList = [1,3,True,6.5]
>>> myList
[1, 3, True, 6.5]
```

注意到当Python中给一个列表赋值后，这个列表会被返回。那么为了记录下这个列表以备后用，我们应该将它的引用值传给一个变量。

因为列表是有序的，所以它支持一系列也可以被用在任何Python 中的序列的运算符。表1.2展示并给出了它们的作用：

名称	运算符	讲解
索引	`[ ]`	指向序列中的一个元素
连接	`+`	组合序列

重复	*	重复该序列
是否在其中	in	查询该元素是否在序列中
长度	len	获取序列长度
切片	[:]	切片操作

表1.2　对任何Python序列均有用的运算符

考虑到列表（序列）从0开始数起，切片操作myList[1:3]，会返回从下标为1到下标为3（但不包含3）的项目。

有时，你想要创建一个列表。这个任务可以用重复操作来快速完成。例如：

```
>>> myList = [0] * 6
>>> myList
[0, 0, 0, 0, 0, 0]
```

顺便再说一件有关重复操作非常重要的事，就是重复运算的结果是对序列中数据的引用的重复。这个可以在如下的展示中得到体现：

```
myList = [1,2,3,4]
A = [myList]*3
print(A)
myList[2]=45
print(A)
```

代码1.3 对引用的重复(intro_1.3)

变量A持有三份对原始名为mylist的序列的引用。注意在对mylist中的元素进行改变后，这种改变体现在了A中。

列表提供了一系列的用来构建数据结构的方法。表1.3提供了一个总结，关于它们的用法在之后的例子中呈现：

方法名	用法	解释
append	alist.append(item)	在列表末尾添加一个新项
insert	alist.insert(i,item)	在列表的某个位置插入一个项
pop	alist.pop()	移除并返回列表的最后一项
pop	alist.pop(i)	移除并返回列表的第i 项
sort	alist.sort()	对列表进行排序
reverse	alist.reverse()	反转列表
del	del alist[i]	删除在该位置上的元素
index	alist.index(item)	返回列表中第一个等于item 项的索引
count	alist.count(item)	返回列表中有多少项的值等于item
remove	alist.remove(item)	删除列表中第一个值等于item 的项

表1.3 Python列表中提供的方法

```
myList = [1024, 3, True, 6.5]
myList.append(False)
print(myList)
myList.insert(2,4.5)
print(myList)
print(myList.pop())
print(myList)
print(myList.pop(1))
print(myList)
myList.pop(2)
```

```
print(myList)
myList.sort()
print(myList)
myList.reverse()
print(myList)
print(myList.count(6.5))
print(myList.index(4.5))
myList.remove(6.5)
print(myList)
del myList[0]
print(myList)
```

代码1.4 列表处理方法示例(intro_1.4)

你可以看到一些方法，例如pop，返回了一个值并且同时修改了列表。其他的，像reverse，只是修改了列表没有返回值，pop默认处理最后一项，但也可以返回一个指定的元素并把它删去。使用这些类函数时，要注意下标也是从0开始的。你也会注意到熟悉的“.”符号，它用于让这个对象调用某个方法。myList.append(False)可以读作“使对象myList去执行append方法函数并给它一个值False”。就连最简单的例如整数的数据体以这个方式都可以调用方法函数。

```
>>> (54).__add__(21)
75
>>>
```

如上，我们让整数54执行名为“add”的方法函数（在Python中被称作__add__）然后将21作为值给add，结果就是它们的和75。当然我们一般写54+21。我们会在这节后面讲解更多关于方法的知识。

还有一个python中的与list相关的常用功能那就是range。range 产生了一个有顺序的对象，这个对象可以表示一个序列。在使用list里的功能时，可以将range对象里的值视为一个列表。下面是关于它的阐释：

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
range(5, 10)
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

对象 range 代表了一个整数的序列。默认会从 0 开始。如果你提供更多的参数，它可以从特定的点开始和结束，甚至可以跳过一些数。在第一个例子中，range(10)，这个序列从 0 开始到 10 却不包含 10。在第二个例子中，range(5, 10) 从 5 开始到 10 却不包含 10，range(5, 10, 2) 与上述例子类似，但是每两个数之间相差 2（不包含 10）。

字符串是可以储存零个或多个字母，数字，或其他符号的有序容器。我们称字母，数字和其他的符号为字符。字符串的值用引号(单引号和双引号都可以，但不能混用)与标识符区分。

```
>>> "David"
'David'
>>> myName = "David"
>>> myName[3]
'i'
>>> myName*2
'DavidDavid'
>>> len(myName)
5
>>>
```

因为字符串是序列，所以所有之前提及的对序列的操作都适用。另外，字符串也有它的一系列方法，一部分被展现在表 1.4 里，例如:

```
>>> myName
'David'
>>> myName.upper()
'DAVID'
>>> myName.center(10)
'   David    '
>>> myName.find('v')
2
>>> myName.split('v')
['Da', ‘id’]
```

在这些方法中，`split` 在处理数据方面非常有用。`split` 接受一个字符串然后返回一个用分隔字符作为切分点的字符串的列表。在上面的例子中，’ v’就是分隔符。如果没有给分隔符，`split` 会自动将 `tab` 符、换行符、空格符作为分隔符。

方法名	使用方法	解释
`center`	`astring.center(w)`	返回一个字符串，w 长度，原字符串居中
`count`	`astring.count(item)`	返回原字符串中出现item 的次数
`ljust`	`astring.ljust(w)`	返回一个字符串，w 长度，原字符串居左
`lower`	`astring.lower()`	返回一个字符串，全部小写

rjust	astring.rjust(w)	返回一个字符串，w 长度，原字符串居右
find	astring.find(item)	查询item，返回第一个匹配的索引位置
split	astring.split(schar)	以schar 为分隔符，将原字符串分割，返回一个列表

表 1.4　Python 中的字符串方法

字符串和列表之间最大的区别就是列表可以被修改而字符串不能。这被称作**可变性**。列表是 可变的，字符串是不可变的。比如，你可以通过赋值和索引改变一个列表中的元素，而对于 字符串这种操作是不允许的。

```
>>> myList
[1, 3, True, 6.5]
>>> myList[0]=2**10
>>> myList
[1024, 3, True, 6.5]
>>>
>>> myName
'David'
```

```
>>> myName[0]='X'
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in -toplevel-
    myName[0]='X'
TypeError: object doesn't support item assignment
>>>
```

元组 `tuple` 和列表 `list` 类似，也是异质数据序列容器，区别是，`tuple` 不可改变其中数据，就像字符串。元表不可被修改。元表表示为用圆括号括起的，用逗号隔开的一系列值。当然，作为一个序列，它也适用于上述的关于序列的运算符，例如 ：

```
>>> myTuple = (2,True,4.96)
>>> myTuple
(2, True, 4.96)
>>> len(myTuple)
3
>>> myTuple[0]
2
>>> myTuple * 3
(2, True, 4.96, 2, True, 4.96, 2, True, 4.96)
>>> myTuple[0:2]
(2, True)
>>>
```

但是，如果你试图改变元组里的一个元素，马上就会报错。注意有关 `error` 的消息中会包含问题的位置和原因。

```
>>> myTuple[1]=False
Traceback (most recent call last):
  File "<pyshell#137>", line 1, in -toplevel-
    myTuple[1]=False
TypeError: object doesn't support item assignment
>>>
```

集合是 0 个或多个数据的无序散列容器。集合不允许出现重复元素，表示为花括号括起的、用逗号隔开的一系列值。空的集合表示为 `set()`。集合是异质的，并且集合是可变的。例如：

```
>>> {3,6,"cat",4.5,False}
{False, 4.5, 3, 6, 'cat'}
>>> mySet = {3,6,"cat",4.5,False}
>>> mySet
{False, 4.5, 3, 6, 'cat'}
```

尽管集合是无序的，它仍然支持一些与之前所示的其他容器相类似的操作，表 1.5 回顾了这些操作，并且之后给出了如何使用这些操作的具体例子：

运算	运算	解释

属于关系	in	判断一个元素是否属于这个集合
元素数目	len	返回值是集合中元素的数目
|（并集）	集合 A | 集合B	返回一个新集合，这个集合是集合A,B 的并集
&（交集）	集合 A & 集合B	返回一个新集合，这个集合只有集合A,B 共有的元素，是集合A,B 的交集
-	集合 A – 集合B	返回一个新集合，这个集合是集合A 除去A 与B 共有的元素（A-（A∩B））
<=	集合 A <= 集合B	判断集合A 中的所有元素是否都在集合B 中，返回布尔值 True 或者False

表 1.5 Python 中集合的运算符

```
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>> len(mySet)
5
>>> False in mySet
True
>>> "dog" in mySet
False
>>>
```

集合提供了许多方法，用于实现一些数学操作，表 1.6 大致列举了一些这样的函数，之后也有一些使用实例。注意，union, intersection, issubset 和 difference 这些函数的功能也可用上述在表五中出现的运算符代替。

函数名	使用方法	解释
union	A.union(B)	返回一个新集合，这个集合含有A,B 中的所有元素，是集合A,B 的并集
intersection	A.intersection(B)	返回一个新集合，这个集合只有集合A,B 共有的元素，是集合A,B 的交集
difference	A.difference(B)	返回一个新集合，这个集合是集合 A 除去A 与B 共有的元素（A-（A∩B））
issubset	A.issubset(B)	判断集合A 中的所有元素是否都在集合B 中，返回布尔值 True 或者False
add	A.add(item)	把item 这个元素添加到集合A 中

remove	A.remove(item)	从集合A中除去item这个元素

表 1.6　Python 中集合操作的函数

```
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>> yourSet = {99,3,100}
>>> mySet.union(yourSet)
{False, 4.5, 3, 100, 6, 'cat', 99}
>>> mySet | yourSet
{False, 4.5, 3, 100, 6, 'cat', 99}
>>> mySet.intersection(yourSet)
{3}
>>> mySet & yourSet
{3}
>>> mySet.difference(yourSet)
{False, 4.5, 6, 'cat'}
```

```
>>> mySet - yourSet
{False, 4.5, 6, 'cat'}
>>> {3,100}.issubset(yourSet)
True
>>> {3,100}<=yourSet
True
>>> mySet.add("house")
>>> mySet
{False, 4.5, 3, 6, 'house', 'cat'}
>>> mySet.remove(4.5)
```

```
>>> mySet

{False, 3, 6, 'house', 'cat'}

>>> mySet.pop()

False

>>> mySet

{3, 6, 'house', 'cat'}

>>> mySet.clear()

>>> mySet

set()

>>>
```

介绍的最后一种 Python 的数据类型是**字典，**字典是由许多对相互之间有联系的元素组成的，每对都包含一个**键（key）**和一个**值（value）**。这种元素对被称为键值对，一般记作**键：值**（key：value）。字典的表示方法是，大括号内若干对键值对排列在一起，它们之间用逗号隔开。 举个例子:

```
>>> capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}

>>> capitals

{'Wisconsin': 'Madison', 'Iowa': 'DesMoines'}

>>>
```

我们可以通过元素的键来获取它的值，也可以添加另外的键值对来改动字典。“通过键来获取值”这一操作的语法和从序列中读取元素的操作差不多，只不过后者是通过序列下标来获取，而前者是通过键获取。添加新元素也是类似的，序列中添加一个新元素，要增加 一个下标，而字典则是增加一个键值对。

```
>>> capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}

 >>> print(capitals['Iowa'])
DesMoines
>>> capitals['Utah']='SaltLakeCity'

>>> print(capitals)
{'Wisconsin': 'Madison', 'Utah': 'SaltLakeCity', 'Iowa': 'DesMoines'}

>>> capitals['California']='Sacramento'
>>> print(len(capitals))
4
>>> for k in capitals:

print(capitals[k]," is the capital of ", k)

('Madison', ' is the capital of ', 'Wisconsin') ('SaltLakeCity', ' is the capital of ', 'Utah')
('DesMoines', ' is the capital of ', 'Iowa') ('Sacramento', ' is the capital of ', 'California')
```

代码 1.5 字典(intro_1.5)

切记，字典型对于**键**(key)的存储是没有特定的顺序的，如上例中，第一个添加的键值对（'Utah':'SaltLakeCity'）被放在了字典中的第一个位置,而第二个添加的键值对('California':'Sacramento')被放在了最后一个位置。关于键的摆放位置实际上是和“散列法”这一概念有关，这会在第四章中详细论述。在上例中我们也展示了 length 函数，它和之前几种数据类型一样，可以用来求字典中所含键值对的数目。

字典可以通过函数和运算符来操作。表 1.7 和表 1.8 分别给出了字典中所能使用的运算符，以及具有类似功能的函数，之后的一段代码给出了它们的实际操作。key，value 和 item 这三个函数分别能够给出字典中所有的键、值、键值对，然后你可以用 list 函数把它们得到的结果转 变为列表。get 函数有两种不同的用法。第一种，如果查找的 key 不在字典中，它就会返回 None；第二种，加入一个可选参数，如果查找的 key 不在字典中,就返回一个特定的值。

运算符	用法	解释
[]	mydict[‘key’]	返回key 这个键所对应的值，如果key 不存在，则会报错
In	key in mydict	如果key 这个键在字典中，那么就返回True，如果不在，就返回False
del	del mydict[‘key’]	在字典中移除key 这个键所对应的键值对

表 1.7 Python 中字典运算符

```
>>> phoneext={'david':1410,'brad':1137}
>>> phoneext
{'brad': 1137, 'david': 1410}
>>> phoneext.keys()
dict_keys(['brad', 'david'])
>>> list(phoneext.keys())
['brad', 'david']
>>> phoneext.values()
dict_values([1137, 1410])
>>> list(phoneext.values())
[1137, 1410]
>>> phoneext.items()
dict_items([('brad', 1137), ('david', 1410)])
>>> list(phoneext.items())
[('brad', 1137), ('david', 1410)]
>>> phoneext.get("kent")
>>> phoneext.get("kent","NO ENTRY")
'NO ENTRY'
>>>
```

函数名	使用方法	解释
keys	adict.keys()	以列表的形式返回adict 中的所有键(key)
values	adict.values()	以列表的形式返回adict 中的所有值(value)
items	adict.items()	以列表的形式返回adict 中的所有键值对列表的每个元素是包含键和值的元组
get	adict.get(key)	返回key 所对应的值，如果key 不存在，就返回None
get	adict.get(key,alt)	返回key 所对应的值，如果key 不存在，就返回alt

表 1.8　Python 中字典的函数

1.7.2 输入与输出

无论是获取数据，或者是返回某种结果，我们都经常需要与用户进行互动。如今的大多数程序都用一个对话框作为一种要求用户提供某种类型的输入的方式。虽然 Python 有对话框，但我们有更简单的函数可以解决这个问题。Python 就为我们提供了这样一个函数，它要求用户输入某些数据，同时以字符串的形式为用户提供输入提示(就是输入的时候提示个“please input your data”之类的)。这个函数就是 input。

Python 中的 ‘’ 函数后面的括号里只能接受单一的字符串，这个字符串被称为**提示符(prompt)**，因为这个字符串一般输入的是有用的文本信息，用于提示用户输入。举个例子,你可以这样:

```
aName = input('Please enter your name: ')
```

之后，无论用户在看到了提示之后输入了什么，它都会被存放在 aName 这个变量中。使用 input，我们就能写出提示用户输入数据的指令，然后把用户输入的数据存储起来，以便进行进一步的操作。例如，在下面的这两条语句中，第一条让用户输入他们的名字，第二条语 句对于他们输入的字符串进行了简单的处理，最后输出了处理的结果。

```
aName = input("Please enter your name ")
 print("Your name in all capitals is",aName.upper(),
     "and has length", len(aName))
```

代码 1.6　input 函数，返回一个字符串(intro_1.6)

input 函数在接收到用户的输入后，它的返回值是一个字符串。其表现了用户在看到输入提示后所输入的数据类型的准确特征，如果你想把这个字符串转化为另外一种数据类型，必须进行明确的类型转换。在下面的这些语句中,用户输入的字符串被转化为了浮点型，从而输入的数据可以进行进一步的运算处理。

```
sradius = input("Please enter the radius of the circle ")
radius = float(sradius)
diameter = 2 * radius
```

1.7.2.1 字符串格式化输出

在之前我们已经了解了 print 函数是一种非常简便的 Python 程序中输出数据的方式，它可以获取 0 个或多个参量并将其输出，默认用空格作为分隔符。事实上，我们可以通过设定 sep 参数来修改分隔符。此外,每一行输出都默认以新行字符(>>>)结束，使用 end 参数可以改变这一设定。以下的代码展示了如何变更。

```
>>> print("Hello")
Hello
>>> print("Hello","World")
Hello World
>>> print("Hello","World", sep="***")
Hello***World
>>> print("Hello","World", end="***")
Hello World***
>>>
```

在程序中，固定格式的输出往往是很有用的，Python 提供了一种替代性的输出,称为**格式化字符串**。格式化字符串提供了一个模板，其中的单词或空格会保持原样，而占位符则留给变量来填充。 例如这个语句：

```
print(aName, "is", age, "years old.")
```

包含单词“is”和“years old”,但是其中的 name 和 age 会随着变量值的改变而改变。利用格式化字符串,我们就能把这行语句写成:

```
print("%s is %d years old." % (aName, age))
```

这个简单的例子展示了一种新的串表达式。% 是字符串运算符，被称为**格式操作符**。在上面这个表达式中，% 的左边是模版或者格式字符串，右边则是一个容器，其中包含了替换格式字符串的变量值。右边容器中的变量的个数必须和左边字符串中的占位符的数目一致，在用变量值替换占位符的过程中，从左到右依次读取变量，按顺序替代占位符的位置。

现在我们更仔细观察这种格式化表达的两侧。格式字符串可以拥有一个或多个需要用变量 来替代的占位符。%后面的字母是转换字符，它显示了何种类型的变量将会填充到字符串的这个位置。在上面的那个例子中，%s 指定了填入的变量是字符串，而 %d 指定了是整数。其它可能的转换字符包括 i, u, f, e, g, c 或%。表 1.9 概述了各种不同类型的转换字符。

字符	输出格式
d, i	整型
u	无符号整型
f	浮点型，如m.ddddd
e	浮点型，如m.dddddde+/-xx
E	浮点型，如m.dddddE+/-xx
g	指数比-4 小或比5 大时使用%e，否则使用%f
c	单字符

s	字符串或者是能通过str()转为字符串的数据
%	输入一个%

表 1.9　格式化字符串转换字符

除了转换字符之外,你也可以在 % 和转换字符之间插入格式修饰符。格式修饰符可以用一段给定长度的空格使变量实现左对齐或者右对齐。格式修饰符也可在小数点后添加数字，指定字段宽度。表 1.10 给出了这些格式修饰符的使用。

修饰符类型	例子	描述
number	%20d	变量值占据20 个字符宽度
-	%-20d	变量值占据20 个字符宽度，左对齐
+	%+20d	变量值占据20 个字符宽度，右对齐
0	%020d	变量值占据20 个字符宽度，前置“0”
.	%20.2f	变量值占据20 个字符宽度，且保留两位小数
(name)	%(name)d	从字典中取key 为name 的值放在此处

表 1.10　格式修饰符

格式操作符(%)的右侧是一个由变量组成的容器，其中的变量值将会被插入左侧的格式字符串，这个容器既可以是元组，也可以是字典。如果这个容器是元组，那么变量值的插入是按顺序的，也就是说，元组中的第一个元素对应于格式化字符串的第一个占位符,然后从左往右依次对应。如果这个容器是字典,那么变量值是按照他们的键(key)来插入的，在这种情况下，所有的转换字符必须用表格中的(name)修饰符来指定键(key)的名称,从而插入其值。

```
>>> price = 24
>>> item = "banana"
>>> print("The %s costs %d cents"%(item,price))
The banana costs 24 cents
>>> print("The %+10s costs %5.2f cents"%(item,price))
The       banana costs 24.00 cents
>>> print("The %+10s costs %10.2f cents"%(item,price))
The       banana costs          24.00 cents
>>> itemdict = {"item":"banana","cost":24}
>>> print("The %(item)s costs %(cost)7.1f cents"%itemdict)
The banana costs      24.0 cents
>>>
```

除了能使用转换字符以及格式修饰符的格式化字符串外，Python 中的字符串也包括了一个名为 format 的函数，它可以和一个新的类 Formatter 结合，以实现更复杂的字符串格式化。关于 Python 的这一特色,可以查阅 Python Library 的参考手册。

1.7.3 控制结构

就像我们前面所说的,算法中有两个重要的控制结构：迭代和选择。Python 支持它们的不同形式，操作者可以选择对于所给问题最有用的表述。

对于迭代结构，Python 提供了一个 while 语句和 for 语句。while 语句可以在循环条件为真的情况下一直重复循环体。例如:

```
>>> counter = 1
>>> while counter <= 5:
...         print("Hello, world")
...         counter = counter + 1
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
```

重复输出“Hello world”5 次。循环条件在每次循环开始时均会判断，如果判断结果为真，循环主体将执行。由于强制缩进的格式，我们可以很容易看清楚 Python 中的 while 循环结构。

while 循环语句是一个非常常见的迭代结构，我们将会在很多不同的算法中使用到。在很多情况下,一个复合条件可以控制迭代结构 。例如 ：

```
while counter <= 10 and not done:
...
```

这段代码中，语句的主体只有在所有的条件都满足的情况下才会执行。变量 counter 的值必须小于或等于 10 并且变量 done(布尔型)必须是 False(not False 就是 True)，因此真+真=真。

这种类型的结构在各种情况下非常有用。另外一个迭代结构 for 语句可以和许多 Python 容器共同使用。只要这个容器是一个序列容器，for 语句可以遍历该容器的所有元素。例如:

```
>>> for item in [1,3,6,2,5]:
...     print(item)
...
1
3
6
2
5
```

`item`依次分配列表中的每个值`[1,3,6,2,5]`。然后执行迭代的主体。这适用于任何容器(列表、元组和字符串)。

`for` 语句的作用是明确迭代值的范围，例如:

```
>>> for item in range(5):
...     print(item**2)
...
0
1
4
9
16
>>>
```

该语句将会执行输出功能 5 次。函数将会返回一系列对象范围代表序列`(0,1,2,3,4,)`，每个值将分配给变量，然后将其平方后输出。

迭代结构另外一个非常有用的功能是处理字符串的字符。下面的代码片段遍历字符串的每个字符，并且将字符都添加到`列表`。结果是含有所有字母的单词的一个`列表`。

```
wordlist = ['cat','dog','rabbit']
letterlist = [ ]
for aword in wordlist:
    for aletter in aword:
        letterlist.append(aletter)
print(letterlist)
```

代码 1.7 遍历列表中的元素（intro_1.7）

选择语句允许程序员提出问题，然后根据结果执行不同的操作。大多数编程语言提供了两种形式的有用的结构：`if else` 和 `if`。下面是一个简单的二元选择使用 `if-else` 语句的例子。

```
if n<0:
    print("Sorry, value is negative")
else:
    print(math.sqrt(n))
```

在这个例子中， `n` 用于检查是否小于零。如果是，输出一个句子，指出它是负的。如果不是，则执行 `else` 子句，计算平方根。

选择结构，与任何控制结构一样可以嵌套，根据得到的结果可以决定是否问下一个问题。例如，假设 `score` 是一个变量，记录对计算机科学进行测试的得分。

```
if score >= 90:
    print('A')
else:
    if score >=80:
        print('B')
    else:
        if score >= 70:
            print('C')
        else:
            if score >= 60:
                print('D')
            else:
                print('F')
```

这个片段将通过输出字母将分数进行分类。如果比分是大于或等于 90，输出 `A`。如果没有(`else`)，下一个判断将会进行。如果比分大于或等于 80，即它必须在 80 年和 89 之间，否则第一个判断有误。此时输出 `B`。我们可以看到 Python 缩进模式有助于理解 `if` 和 `else` 之间的 联系，无需任何额外的语法元素。

另一种实现嵌套的表述使用 `elif` 作为关键字。`else` 和 `if` 结合，消除了额外的嵌套。最后的 `else` 仍然是必要的。 因为有可能以上条件均不成立。

```
if score >= 90:
    print('A')
elif score >=80:
    print('B')
elif score >= 70:
    print('C')
elif score >= 60:
    print('D')
else:
    print('F')
```

Python 中也有一个单向选择结构，`if` 语句。`if`语句中，如果条件为真，则执行操作。如果条件为假，仅执行后面的语句。例如,下面的代码片段首先检查变量 n 的值是否为负。 如果是,则其由绝对值函数修改它的值。无论如何，下一句代码是计算平方根。

```
if n<0:
    n = abs(n)
print(math.sqrt(n))
```

牛刀小试

尝试下面的练习来测试你是否理解了以上内容。修改 Activecode 8 的代码，使最终列表只包含每个字母的单一副本。

```
# the answer is: ['c', 'a', 't', 'd', 'o', 'g', 'r', 'b', 'i']
```

在列表中，存在一种替代方法使用迭代和选择结构创建列表。这被称为**列表解析**。列表解析可以让你创建基于某些处理或选择条件的列表。例如，如果我们想创建的小于 10 的整数的完全平方的列表，我们可以使用`for`语句:

```
>>> sqlist=[]
>>> for x in range(1,11):
        sqlist.append(x*x)
>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

使用列表解析,我们可以一步完成:

```
>>> sqlist=[x*x for x in range(1,11)]
>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

在 `for` 结构中,变量 x 取从 1 到 10(不包括 10)的整数值。 计算 `x * x` 的值后，将结果添加到新 建的列表中。列表解析中也可以使用选择语句，以便只有某些项目进行运算并添加到新的列表中。例如：

```
>>> sqlist=[x*x for x in range(1,11) if x%2 != 0]
>>> sqlist
[1, 9, 25, 49, 81]
>>>
```

这个列表解析语句建立了一个仅含有 1 到 10 以内奇数的完全平方的列表，任何支持迭代运算的序列都可以在列表解析中被用来构造一个新的列表。

```
>>>[ch.upper() for ch in 'comprehension' if ch not in 'aeiou']
['C', 'M', 'P', 'R', 'H', 'N', 'S', 'N']
>>>
```

牛刀小试

通过重做 Activecode8 测试你对列表解析的理解。另外，试试删除重复的元素。

```
# the answer is: ['c', 'a', 't', 'd', 'o', 'g', 'r', 'a', 'b', 'b', 'i', 't']
```

1.7.4 异常处理

写程序时，通常会出现两种错误。第一，语法错误，简单的说就是程序员在语句结构或表达中犯了一个错误。例如，在 `for` 语句中没有写冒号是错误的：

```
>>> for i in range(10)
SyntaxError: invalid syntax (<pyshell#61>, line 1)
```

在这种情况下，Python 编译器发现它不能完成该指令的处理，因为不符合语法的规则。当你第一次学习一门语言时，语法错误往往更频繁。

另一种错误是逻辑错误，程序可以执行，但给出的结果是错误的。这可能是由于底层算法错误或者错误地表达了该算法。此时，逻辑错误导致运行错误，程序终止。这些类型的运行错 误的通常被称为**程序异常(exceptions)**。

大多数的时候,初学者简单地认为异常是导致执行结束的运行错误。但是，大多数编程语言提供了一种方法来处理这些错误，让程序员遇到这些运行错误时有一些干预措施。此外， 如果他们需要异常对程序进行检测的话，程序员可以创建异常。

当异常发生时，我们称为异常被“提出”。可以通过 `try` 语句“处理”已提出的异常。例如，考虑一段代码，要求用户输入整数，然后调用 Python 库函数中的平方根函数。如果用户输入一个值，该值大于或等于 0 时，打印其平方根。如果用户输入一个负值，平方根函数将报告一个 `ValueError` 异常。

```
>>> anumber = int(input("Please enter an integer "))
Please enter an integer -23
>>> print(math.sqrt(anumber))
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    print(math.sqrt(anumber))
ValueError: math domain error
>>>
```

我们可以通过从 `try` 模块中调用 print 函数处理这个异常。相应的，模块除了“捕获”了异常，还会打印一条消息返回给用户。例如:

```
>>> try:
        print(math.sqrt(anumber))
    except:
        print("Bad Value for square root")
        print("Using absolute value instead")
        print(math.sqrt(abs(anumber)))
Bad Value for square root
Using absolute value instead
4.79583152331
>>>
```

`try` 语句发现异常产生于调用平方根函数时，然而会将该信息返回给用户，并提醒用户使用数值的绝对值。以确保我们在对一个非负数进行开方运算。这意味着，程序不会终止,，而是会继续到下一个语句。

另外，程序员可以通过使用 raise 语句自己制造运行异常。例如，我们不调用开方的函数，而检查输入值，然后调用异常。下面的代码片段显示了提出一个 RuntimeError 异常的结 果。请注意，该计划将仍然会终止，但现在导致终止异常是由程序员创建的。

```
>>> if anumber < 0:
...         raise RuntimeError("You can't use a negative number")
... else:
...         print(math.sqrt(anumber))
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
RuntimeError: You can't use a negative number
>>>
```

除了上述所示的 RuntimeError 异常,还有其他他种类的异常。所有可用的异常类型以及如何提出异常可见 Python 的参考手册。

1.7.5 定义函数

刚才的例子从数学模块 Python 函数中调用了开方这一函数。在一般情况下，我们可以通过定义一个函数隐藏任何计算的详细步骤。一个函数定义需要一个名字，一组参数，和函数主体。它也可以返回一个确定值。例如，下面定义一个返回输入值平方的简单函数。

```
>>> def square(n):
... return n**2
...
>>> square(3)
9
>>> square(square(3))
81
>>>
```

此函数语法的定义包括名称函数名称 square 和形式参数组合列表。在以上函数中，n 是唯一的形式参数，这表明 square 函数只需要输入一个数据即可运行。函数主体计算 n 平方的结果，并返回这个值。我们可以通过在 Python 环境中调用 square 函数来计算它。给 square 函数传递一个实际参数值，在上述例子中，我们把 3 这个实际数值传给了函数。注意，调用 square 函数返回的值可以被其他操作进行使用。

我们可以通过使用“牛顿法”实现我们自己的平方根函数。用牛顿法逼近平方根计算，使结果收敛于正确的值。等式 newguess= 1/2 *（oldguess+ N / oldguess）取一个值 n，重复地执行以上等式，每

一次等式计算结果 newguess 变成下一次迭代的 oldguess 带入进行计算。这里使用的初值为 n/2。代码 1 中有这样一个函数，它接受一个值 n，并返回做 20 次重复计算之后的值。同样，牛顿法的具体步骤被隐藏在函数定义内，用户不必知道该函数为达到预计目的是如何实现这个功能的。代码 1 还说明＃字符是作为注释标记使用的。＃后的任何字符都被忽略。

```
def squareroot(n):

root = n/2 #initial guess will be 1/2 of n

for k in range(20):

root = (1/2)*(root + (n / root))

return root

>>>squareroot(9)

3.0

>>>squareroot(4563)

67.549981495186216

>>>

29
```

牛刀小试

这里有一个涵盖了我们所学所有知识的题目。你可能已经听说了无限猴子定理？该定理指出，猴子随机在打字机键盘键入一个字符，经过无限时间后，肯定会键入一系列给定的文字，比如莎士比亚全集。好吧，假设我们用一个 Python 函数替换猴子。你认为多久以后才能生成一句莎士比亚的名言呢？我们将这句话定为：“methinks it is a weasel”

你不会希望在浏览器中运行这个程序，所以使用 Python IDE 吧。我们将模拟这个问题的方法是编写一个函数，该函数生成一个 27 个字符长度的字符串，从 26 个字母和空格中随机选择一个字符。我们将编写另一个函数，来比较随机生成的字符串和目标字符串。

第三个函数将反复调用生成和比较函数，那么如果所有目标字母都在随机字符串中出现了，我们就完成了。如果字母没有全部出现，我们会生成一个全新的字符串.为了让它更易于跟随你的程序的过程，第三个函数应该返回出到目前为止产生的最好的字符串，并返回在产生这个字符串之前每 1000 次尝试中产生其它不合题意的字符串的次数。

挑战

看看你是否可以这样优化程序，保留正确的字母，只修改符合到目前为止与目标字符串最接近的字符串中的一个字符。如果新生成的字符是目标字符串中需要的，我们就用这个字符覆盖前一个字符串中不合题意的字符，这是一种类似“爬坡”的算法。

1.7.6. PYTHON 面向对象编程：定义类

我们之前声明了 Python 是一种面向对象的编程语言，到目前为止我们已经用了很多内置的类来说明数据和控制的构成。但是，面向对象的编程语言一个最重要的特征是允许编程者（解决问题的人）来创造一个可以用来解决问题的数据模型的新的类。

记住我们运用抽象的数据类型来提供一个关于某种数据项目是什么样子（它的阐述）和它能做什么（它的方法）逻辑描述。通过建立类来实现一种抽象数据类型，一个编程者可以从中获得抽象过程的好处同时也提供必须的细节来将这些抽象实际运用到一个程序中。无论何时我们想实施一种抽象数据类型，我们将依靠新的类来做它。

1.7.6.1.示例：FRACTION 类

用一个很常见的例子来展现实施一个使用者定义的类是建立一个 Fraction 类来实施抽象数据类型。我们已经知道 Python 提供很多数字的类供我们使用。有时候，然而，可能创建一个“类似”fraction 的数据类型会更有可观性。

一个 fraction 如 3 / 5 包括两部分。上面的数字，称为分子，可以是任何整数。下面的数字，称为分母，可以是任何大于 0 的整数（负的 fraction 有负的分子）。虽说我们可以创建一个浮点数近似任何 fraction，但在有的情况下我们还是想用分数表示一个确切的值。

对 Fration 运算符应使对 Fraction 数据对象的运算操作像任何其他的数值一样，我们需要能加，减，乘和除。我们也希望能够显示 fraction 使用标准的“分数”的形式，例如 3 / 5。此外，所有 fraction 的运算应该以最简的形式返回结果，这样不管怎样进行计算，我们最后总是以最常见的形式得出结果。

在 Python 中，我们通过提供一个名字和设立一个方法（在语法上类似于函数定义）来定义一个新的类。举这个例子，

```
class Fraction:

#the methods go here
```

为我们提供了定义方法的框架。对于所有的类而言第一步提供的都应是构造函数。构造函数中定义了数据对象的创建方式。创建一个 Fraction 的对象，我们需要提供两块数据，分子和分母。在 Python 中，构造函数的方法经常被称作__init__（两个下划线在 init 之前和之后），如代码 1.8 所示。

```
class Fraction:

def __init__(self,top,bottom):

self.num = top

self.den = bottom
```

代码 1.8(intro_1.8)

注意，正式的参数列表包含三项（`self, top, bottom`）。`self` 是一个特殊的参数，都可以用来作为参考返回对象本身。它必须是第一个正式参数；然而，调用时它将永远不需给出一个实际参数值。如前所述，Fraction 需要两块状态数据，分子和分母。在构造函数符号 `self.num` 定义 Fraction 对象有一个内部的数据对象被称为 `num` 作为它的状态。同样，`self.den` 被创造为分母。这两个形参的值最初被分配的状态，使新的 fraction 对象知道它的起始值。

为创造一个 Fraction 类的实例，我们必须借助构造器。这会在使用类的名称和传递实际数值的时候发生（注意我们不会直接借助`__init__`）。例如，

```
myfraction = Fraction(3,5)
```

这里创建一个对象，称为 myfraction 代表分数 3 / 5（五分之三）。图 1.5 显示了这个目标现在的实现。

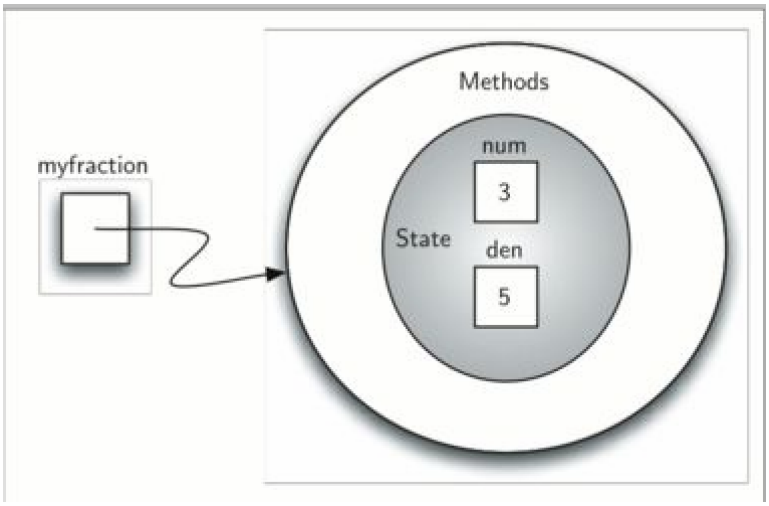


图 1.5 一个分数类的实例

我们要做的下一件事是实现抽象数据类型要求的行为。首先，考虑当我们尝试打印分数对象会发生什么。

```
>>> myf = Fraction(3,5)
>>> print(myf)
<__main__.Fraction instance at 0x409b1acc>
```

`fraction` 对象，`myf`，不知道如何回应这一打印要求。打印功能要求对象转换成字符串以至于该字符串可以输入输出。`myf` 唯一的选择是显示该变量中存储的实际引用（本身的地址）。这不是我们想要的。

我们有两种方法可以解决这个问题。一个是定义一个方法称为 show 将允许 Fraction 对象将自己作为一个字符串输出。我们可以实现这个方法，如代码 1.9 所示。如果我们之前创建的 Fraction 对象，我们可以让它 show（展示）它自己，换句话说，用适当的格式打印本身。不幸的是，这工作一般不可完成。为了使打印正常工作，我们需要告诉 Fraction 类如何转换成一个字符串。这就是打印功能完成任务所需要的。

```
def show(self):
print(self.num,"/",self.den)
>>> myf = Fraction(3,5)
>>> myf.show()
3 / 5
>>> print(myf)
<__main__.Fraction instance at 0x40bce9ac>
>>>
```

代码 1.9(intro_1.9)

在 Python 中，所有的类都有一个标准方法，但不一定都能正常工作。其中的一个__str__是一种方法来将对象转为字符串。此方法的默认实现是返回我们已经看到的地址的字符串。我们需要做的是将这个方法“更好”的实现。我们会说这个实现覆盖前一个，或者说它重新定义了方法的行为。

为做到这个，我们简单地定义一个方法，名称为`__str__`并给它一个新的实现如代码 4 所示。这个定义并不需要任何其他信息，除了特殊参数`self`。反过来，该方法会通过转变每一块的内部状态的数据为一个字符串，然后放置一个字符在字符串之间作为字符串关联的事物。结果字符串将在任何一个 Fraction 对象要求转换它自己为字符串时被返回。注意这个函数使用的各种方法。

```
def __str__(self):
return str(self.num)+"/"+str(self.den)
>>> myf = Fraction(3,5)
>>> print(myf)
3/5
>>> print("I ate", myf, "of the pizza")
I ate 3/5 of the pizza
>>> myf.__str__()
'3/5'
>>> str(myf)
'3/5'
>>>
```

代码 1.10(intro_1.10)

我们可以为我们的新`Fraction`类装载许多其他的方法。一些最重要的是基本的算术运算。我们希望能够创建两个分数对象然后他们加在一起使用标准的“`+`”符号。在这一点上，如果我们试图直接相加两个分数，则会得到如下：

```
>>> f1 = Fraction(1,4)
>>> f2 = Fraction(1,2)
>>> f1+f2
Traceback (most recent call last):
File "<pyshell#173>", line 1, in -toplevelf1+
f2
TypeError: unsupported operand type(s) for +:
'instance' and 'instance'
>>>
```

如果你仔细看看错误，你会看到错误是“+”号运算无法被当成运算符理解。

我们可以通过提供 Fraction 类重写方法解决这个问题。在 Python 中，这种方法被称为 `__add__`，它需要两个参数。第一，`self`，这个总是需要的，第二是在另一个被操作数的表达。例如，

```
f1.__add__(f2)
```

会要求 Fraction 对象 `f1` 来将 `f2` 加到到本身。这可以写为标准的符号格式，`f1 + f2`。

两部分必须有相同分母来相加。最简单的方法来确保他们具有相同的分母是简单地使用两分母的公倍数作为一个共同的分母，a / b + c / d =ad/ bd + cb / bd =（ad + cb）/ bd 的实现如代码 1.11 所示。此外，函数返回一个新的分子和分母的分数总和对象。我们可以通过写一个标准的分数算术表达式使用此方法，分配加法结果，然后打印我们需要的结果。

```
def __add__(self,otherfraction):
newnum = self.num*otherfraction.den + self.den*otherfraction.num
newden = self.den * otherfraction.den
return Fraction(newnum,newden)
>>> f1=Fraction(1,4)
>>> f2=Fraction(1,2)
>>> f3=f1+f2
>>> print(f3)
6/8
>>>
```

代码 1.11(intro_1.11)

方法“__add__”达到了我们的目的，但结果并不完美。6/8 的确是对的，但是它并不是最简分数。最好的答案应该是 3/4。为了保证运算结果为最简分数，我们需要一个辅助函数来识别并化简分数。这个函数需要能够找到分子分母的最大公因数，然后我们就可以将分子分母同时除以最大公因数，得到最简分数答案。

最著名的寻找最大公因数的方法是欧几里得算法，我们将在第 8 节对其做具体讨论。欧几里得算法规定：对于两个整数 m 和 n，如果 n 能整除 m，那么就将 n 除以 m 的结果作为新的 n，如果 n 不能再整除 m，那么最大公因数就是 n 或者 m 被 n 整除的余数。在这里简要地给出一个以迭代法实现的示例（请参见动态代码 1.12）。注意这个最大公因数的算法只适用于分母是正数的情况。因为我们可以把负分数的负号归结于分子，所以这种算法还是比较实用的。

```
def gcd(m,n):
while m%n != 0:
oldm = m
34
oldn = n
m = oldn
n = oldm%oldn
return n
print gcd(20,10)
```

函数来化简分数式。为了使所求分数为最简分数形式，令分子、分母同时除以它们的最大公约数。如对于 6/8 这个分数，最大公约数是 2。我们分数线上下同时除以 2 就得到了一个“新的分数”，3/4（请参见第 6 节）。

```
def __add__(self,otherfraction):
newnum = self.num*otherfraction.den + self.den*otherfraction.num
newden = self.den * otherfraction.den
common = gcd(newnum,newden)
return Fraction(newnum//common,newden//common)
>>> f1=Fraction(1,4)
>>> f2=Fraction(1,2)
>>> f3=f1+f2
>>> print(f3)
3/4
>>>
```

代码 1.12(intro_1.12)

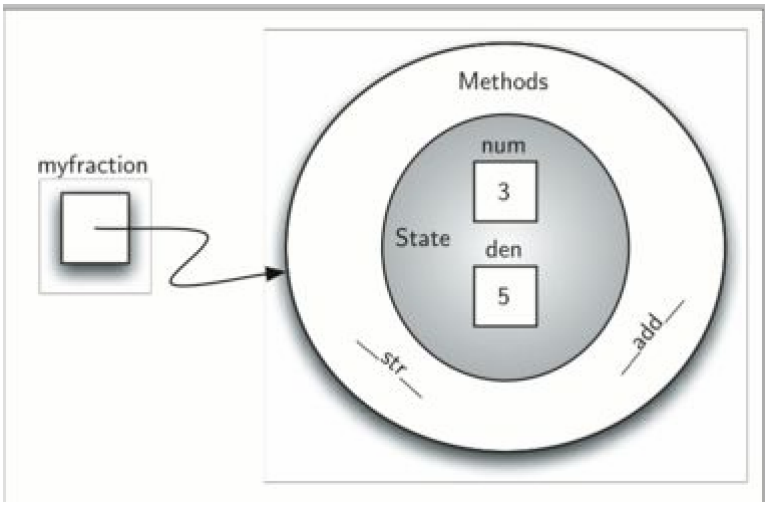


图 1.6 一个用 2 种方法实现分数类型的例子

对于分数对象，现在有 2 个非常有用的方法，如图 1.6。我们需要在我们的示例函数类中包含一个可以使两个分数相互比较的方法组。假设我们有两个分数对象，他们分别是 f1 和 f2，仅当 f1 与 f2 指向的是同一对象时（f1==f2）才为真命题，两个不同的对象即使含有相同的分子和分母，在这个实现中也不相等。这个现象被称为浅相等（见图 1.7）。

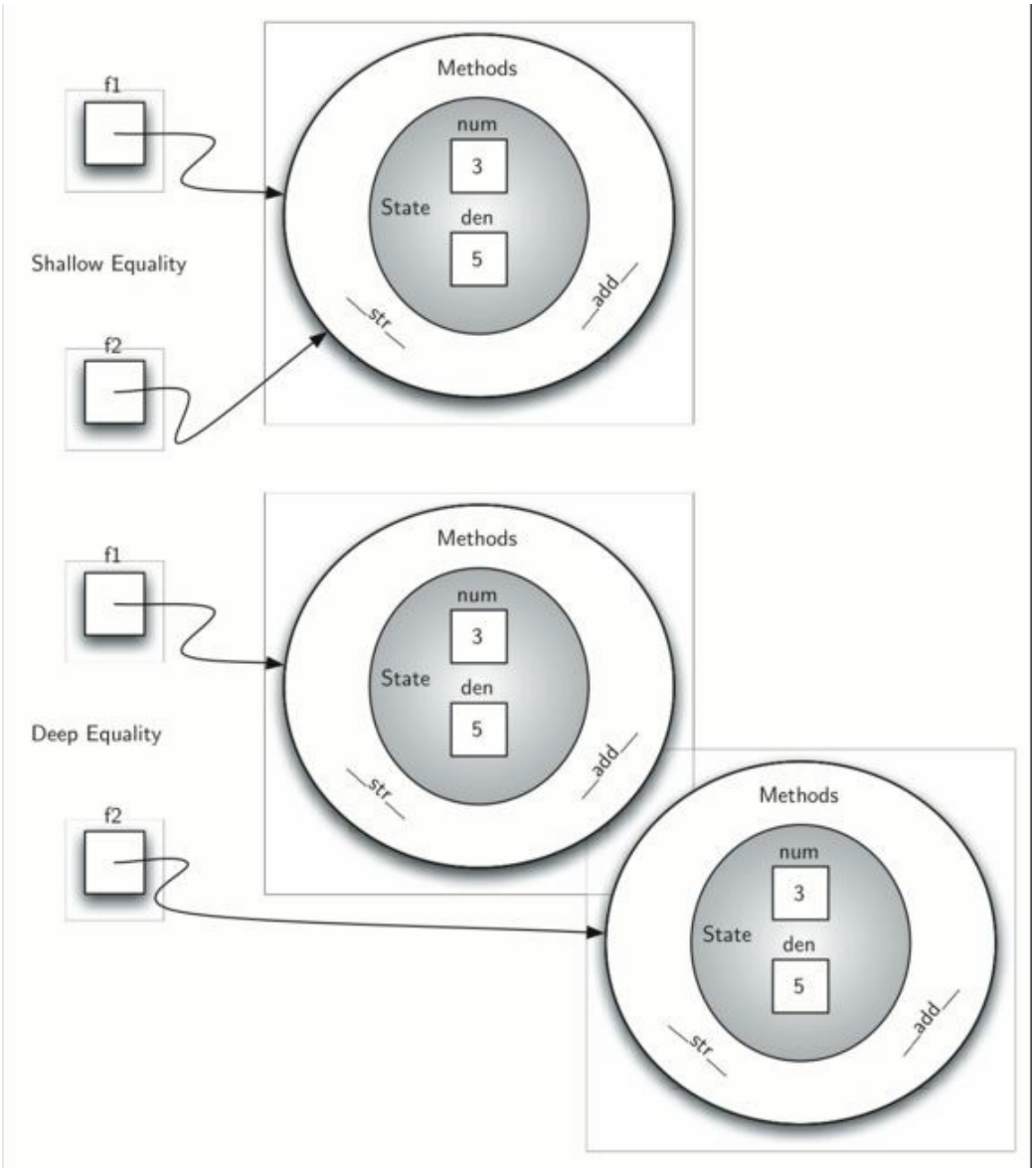


图 1.7 浅相等和深相等的比较

我们可以通过重建_eq_方法来建立深相等（见图 1.7）——即一种数值上相等，却并不一定是相同指向的相等方式。_eq_方法是另一种在很多类中均可用的标准方法。_eq_方法比较两个对象，若它们在数值上相等就返回真值，否则就返回假值。

在分数类中，我们通过再一次将两个分数放在同一条件下，再比较他们的分母（通分）的方法实现_eq_方法（见代码 1.13）。需要指出的是，还有其他的相关运算子是可以被重构的。比如，_le_方法可以判定“小于等于”。

```
def __eq__(self, other):

firstnum = self.num * other.den

secondnum = other.num * self.den

return firstnum == secondnum
```

代码 1.13(intro_1.13)

直到这里，完整的分数类已经在动态代码 2 中呈现了。剩下的算法和相关方法将作为训练留给读者。

牛刀小试

为加深理解在 Python 课程中运算符是怎样生效的，以及如何正确地写方法，请写一些包括*，/，-，以及比较运算符>和<的方法。

1.7.6.2 继承：逻辑门与门电路

最后一节我们将要介绍面对对象编程的另一个重要方面。继承是一个类联系另一个类的能力，就好像人与人之间能够相互联系,孩子们从父母继承特征。Python 子类能够继承父类的数据和行为特征，类似于孩子们继承父母的特征。这些类通常称为子类和父类。图 1.8 显示了内置 python 集合以及它们之间的相互关系我们称这样的一个继承层次结构的关系。例如，列表是有序集合的一个子集，这时，我们称列表为子类，有序集为父类（或者列表亚类、有序超类）。通常称为类的父子继承关系（the list IS-A sequential collection）。这意味着列表从有序集继承重要的特征序列，即基础数据的排序和操作（连接、重复、索引等）。

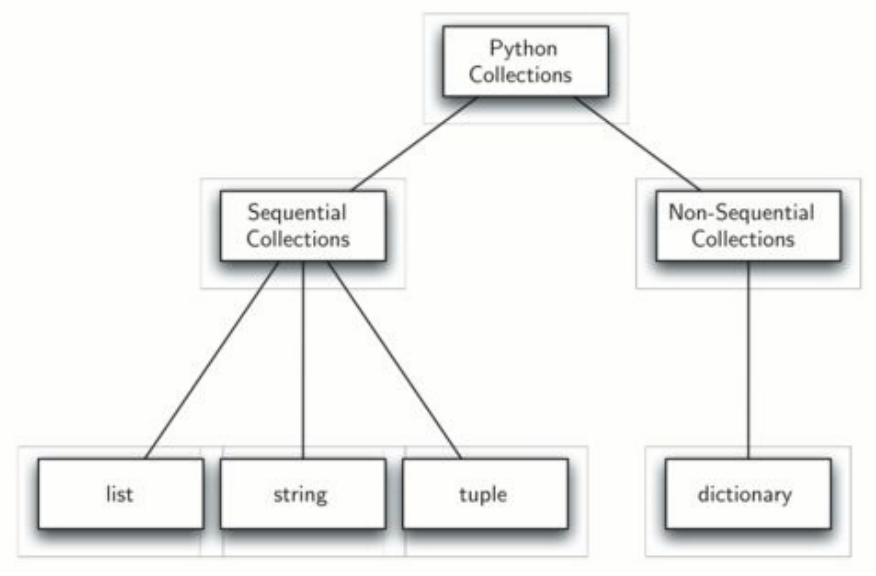


图 1.8 python 集合的继承层次结构

有序集合的类型包括列表(list)、元组(tuple)和字符串(string)。就像孩子们一方面继承父母的特征，另一方面也通过添加额外的特征来加以区分自己，这三者都继承了共同的数据组织和操作，但是，它们每一个都是独特的，区别在于数据是否均匀、是否不变。

通过这种分层方式来组织类，面对对象的编程语言允许在编写代码之前进行扩展，以满足新情景的需要。此外，用这种分层的方式来组织数据，我们可以更好地了解存在的关系以及更有效地构建抽象表示。

为了进一步探索这种想法，我们将构建一个模拟，用来模拟数字电路的程序。逻辑门(logic gate)是这个模拟的基本构建模块。这些电子开关代表布尔代数的输入和输出关系。一般来说，门只有一行简单的输出，而输出的值取决于给定的输入值。与门(AND gate)有两个输入端，每一个都可以是 0 或者 1（代表假或者真）。如果输入值都是 1，则输出是 1。然而，如果输入值有一个是 0 或者两个都是 0，那么输出是 0。或门(OR gate)同样也有两个输入端，如果输入有一个 1 或者两个都是 1，那么输出 1。只有两个输入都是 0，结果才是 0。非门(NOT gate)与其它两个门不同的是它只有一个输入端，输出值刚好与输入值相反。如果输入 0，

则结果是 1。类似地，输入 1，则结果是 0。图 1.9 显示了每一个门的典型代表。每一个门有一个表格来显示相应的门的输入——输出映射。

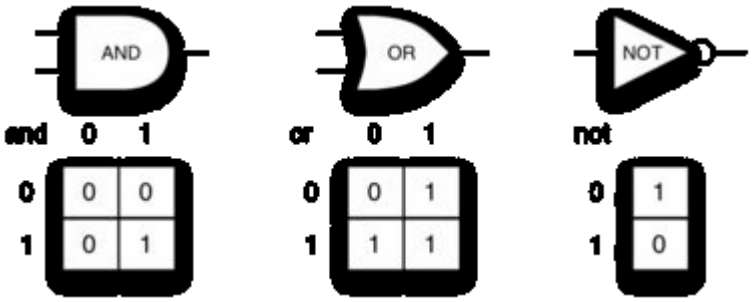


图 1.9　三种门的典型代表

通过结合这些不同模式的逻辑门，我们可以应用一组输入值来构建具有逻辑功能的电路。图 1.10 是一个由两个与门、一个或门和一个非门组成的电路。数值从两个与门输出后直接反馈给或门，或门的输出结果再进入非门。如果我们将一组输入值应用到四个输入行（每个与门有两个），数值经过处理后得到的结果就会在非门输出。示例见图 1.10。

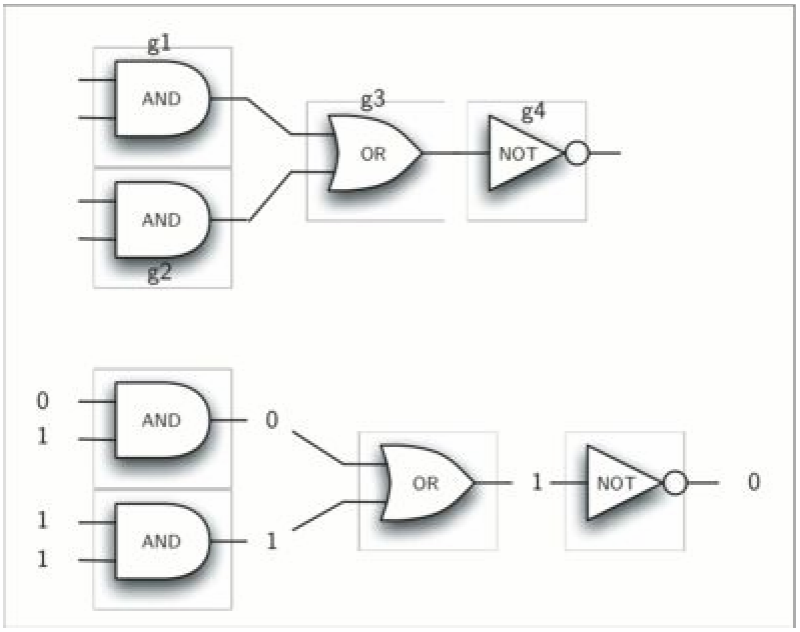


图 1.10 电路

为了实现一个电路，我们首先要构建一个逻辑门的表示。逻辑门很容易被组织成一个类继承层次结构（见图 1.11）。在层次结构的顶部，逻辑门类（LogicGate）代表逻辑门的最普遍特征,即有一个标签的门和一个输出端。子类的下一级别把逻辑门分为两个家族，有一个输入端的和有两个输入端的。再往下，每一个门都有特定的逻辑功能。

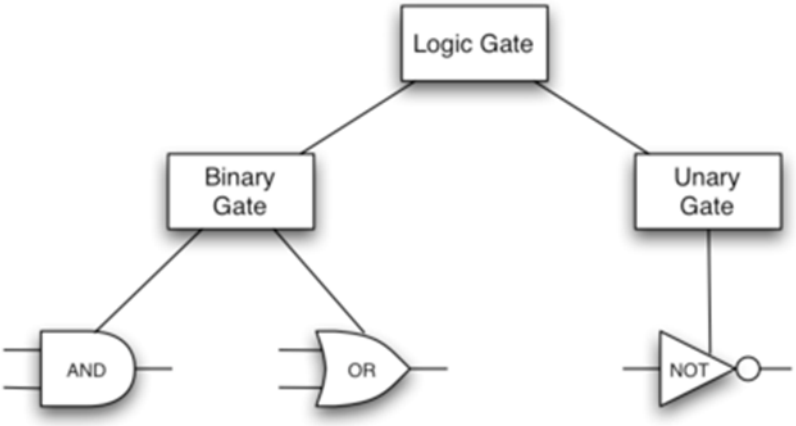


图 1.11 逻辑门的继承阶层架构

我们现在可以从最常见的 `LogicGate` 即逻辑门开始执行类了。如前所述，每一个门都有一个用于识别的标签和单独的输出行。此外，我们还需要方法使得一个门的用户能够向这个门索求标签。

每个逻辑门所需要的另外一种行为是获得它的输出值的能力。这要求门根据当前的输入而执行适当的逻辑。为了生成输出，门需要明确了解这一逻辑是什么。这意味着调用一种方法来执行逻辑运算。代码 1.14 向我们展示了一个完整的类。

```
class LogicGate:
    def __init__(self,n):
        self.label = n
        self.output = None
    def getLabel(self):
        return self.label
    def getOutput(self):
        self.output = self.performGateLogic()
        return self.output
```

代码 1.14 类(intro_1.14)

此时，我们并不会执行 `performGateLogic` 函数，因为我们不知道每个逻辑门将会怎样执行它的逻辑操作。这些细节将会包含在每一个被加入到阶层架构的门中，这是一个非常强大的面向对象编程思想。我们在写一个使用了尚不存在的代码的方法。参数 `self` 是实际门调用这个方法的一个引用。任何新加入阶层架构的逻辑门只需要执行 `performGateLogic` 函数，它便会在适当的时候被使用。一旦完成，门就可以提供它的输出值。这种可以扩展现存阶层架构并且为需要使用新类的阶层架构提供具体功能的能力对再利用现存代码来说是非常重要的。

我们基于输出行的数量对逻辑门进行分类。与门以及或门都有两条输出行，而非门则有一条输出行。`BinaryGate` 类是 `LogicGate` 的一个子类，而且会增加两条输出行。`UnaryGate` 也是 `LogicGate` 的子类，但是只有一个输出行。在计算机电路设计中，这些行有时也被称为“插脚”，所

以我们在执行过程中也会使用这一术语。

```
class BinaryGate(LogicGate):
    def __init__(self,n):
        LogicGate.__init__(self,n)
        self.pinA = None
        self.pinB = None
    def getPinA(self):
        return int(input("Enter Pin A input for gate "+ self.getLabel()+"-->"))
    def getPinB(self):
        return int(input("Enter Pin B input for gate "+ self.getLabel()+"-->"))
```

代码 1.15(intro_1.15)

```
    class UnaryGate(LogicGate):
        def __init__(self,n):
            LogicGate.__init__(self,n)
            self.pin = None
        def getPin(self):
            return int(input("Enter Pin input for gate "+ self.getLabel()+"-->"))
```

代码 1.16(intro_1.16)

代码 1.15 和代码 1.16 执行了这两个类。这些类的构造器都开始于一个对于使用父函数的 `__init__`方法的父类构造器的精确访问。当创建一个 `BinaryGate` 类的示例的时候，我们首先希望初始化所有从 `LogicGate` 继承的数据项。这种情况下，即是门的标签。之后构造器继续添加两条输入行（`pinA` 和 `pinB`）。这是一个我们在构建类的阶层架构时应当使用的非常普遍的模式。子类构造器需要访问父类构造器然后再转移到它们自己的有区分度的数据上。

Python 还有一个函数名为 `super`，这是一个可以用来代替对父类精确命名的函数。这是一个更为普遍的机制，也被广泛使用，尤其是当一个类有不止一个父类时。但是我们不准备在这个介绍中讨论它。比如在我们上面的例子中，`LogicGate.__init__(self,n)` 可以被替换为 `super(UnaryGate,self).__init__(n)`。

`BinaryGate` 类唯一增加的一个性能是从两条输入行得到值的能力。由于这些值来自某些外部空间，我们便很简单地要求用户通过一个输入声明来提供它们。同样的执行过程也发生在 `UnaryGate` 类中，只不过在这一过程中只有一条输入行。

现在我们有了一个依赖于输入行的数量的针对逻辑门的普遍类，我们已经可以构建具有特殊行为的特殊逻辑门。比如，由于与门有两条输入行，`AndGate` 类可以作为 `BinaryGate` 类的一个子类。如前所述，构造器的第一行向上访问父类构造器（`BinaryGate`），相应地，它又向上访问它的父类构造器（`LogicGate`）。注意，因为 `AndGate` 类继承了两条输入行，一条输出行和一个标签，所以它并

未提供任何新的数据。

```
class    AndGate(BinaryGate):
        def __init__(self,n):
            BinaryGate.__init__(self,n)
        def performGateLogic(self):
            a = self.getPinA()
            b = self.getPinB()
            if a==1 and b==1:
                return 1
            else:
                return 0
```

代码 1.17(intro_1.17)

AndGate 需要添加的唯一的东西是一项特殊行为——执行之前描述过的布尔运算。这里我们可以提供 performGateLogic 方法的地方。对于一个与门，这种方法首先必须得到两个输入值，并仅当两个值都是 1 的时候才返回 1。代码 1.17 展示了这样一个完整的类。为了显示出 AndGate 类执行的效果，我们可以创建实例，观察实例计算的输出。下面的代码展示的是一个名为 g1 的 AndGate 对象，这里，g1 有一个内部标签“G1”。当我们调用 getOutput 方法时，对象首先会访问它的 performGateLogic 方法来相应地询问两条输入行。一旦值被提供，正确的输出就会显示出来。

```
>>> g1 = AndGate("G1")
>>> g1.getOutput(
Enter Pin A input for gate G1-->1
Enter Pin B input for gate G1-->0
0
```

代码 1.18(intro_1.18)

同样的过程也可以应用于或门和非门。OrGate 类也可以是 BinaryGate 的一个子类，NotGate 亦能扩充 UnaryGate 类。这两个类都需要提供它们自己的 performGateLogic 函数，因为这是它们的特殊行为。

下面，我们通过建立一个交互式门类的实例并输出结果，来演示单个门类的使用方法：

```
>>> g2 = OrGate("G2")
>>> g2.getOutput()
Enter Pin A input for gate G2-->1
Enter Pin B input for gate G2-->1
1
>>> g2.getOutput()
Enter Pin A input for gate G2-->0
Enter Pin B input for gate G2-->0
0
>>> g3 = NotGate("G3")
>>> g3.getOutput()
Enter Pin input for gate G3-->0
1
```

代码 1.19

既然我们已经掌握了基本的门类操作，接下来便可以创建组合逻辑电路了。组合逻辑电路需要把各个门组合连接起来，让一个门的计算结果流入另一个门成为输入值。为了实现这个操作，我们将执行一个新的类，叫做 Connector。

Connector 类并不属于门的阶层架构，而是在其两端分别含有门层（见图 1.10）。这种关系是面向对象的编程中十分重要的关系，叫做“HAS-A”关系，即非继承关系。在这里复习一下之前学过的“IS-A”关系，也即继承关系，是指子类与父类的相似性，如 UnaryGate 和 LogicGate。

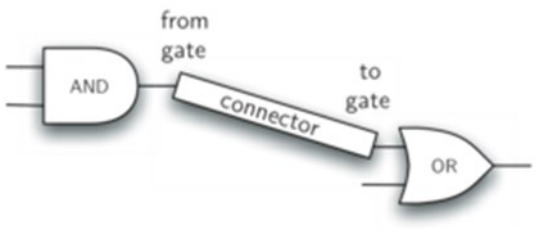


图 1.12 连接器连接着一个门的输出与另一个门的输入

现在有了 Connector 类之后，我们得知“组合非继承逻辑门”意味着连接器内部有门类的实例，但却并不属于门的阶层架构。在设计类的时候，分清楚“IS-A”关系（继承关系）与“HAS-A”（非继承关系）是十分重要的。

代码 1.19 展示了一个 Connector 类。连接器内有两个门类的实例，在每个实例中连接器对象作为 fromgate 或 togate，指示数据值从一个门的输出“流”入另一个门的输入。建立连接的时候，setNextPin 的命令是十分重要的（见后代码 1.20）。我们需要把这个方法添加到门类中，以便

每一个 `togate` 都能够选择合适的输入行作为连接。

```
class Connector:
    def __init__(self, fgate, tgate):
        self.fromgate = fgate
        self.togate = tgate
        tgate.setNextPin(self)
    def getFrom(self):
        return self.fromgate
    def getTo(self):
        return self.togate
```

代码 1.19(intro_1.19)

在 `BianryGate` 类中，有些门有两条输入行，而连接器只能连接一条。如果两条行都是可用的，我们默认选择 `pinA`。如果 `pinA` 已经被其他连接器连接了，我们便选择 `pinB`。想要实现一个门的连接，至少要有一条可用的输入行。

```
def setNextPin(self,source):
    if self.pinA == None:
        self.pinA = source
    else:
        if self.pinB == None:
            self.pinB = source
        else:
            raise RuntimeError("Error: NO EMPTY PINS")
```

代码 1.20(intro_1.20)

现在我们可以从两个地方得到输入数据：一种就是前述的从外部用户处获取，另一种是从上一个门的输出值中获取。这就要求 `getPinA` 和 `getPinB` 方法也进行适当的改进（见代码 1.21）：如果输入线没有任何可用连接（`None`），便与原来一样提示用户输入数据；如果有可用连接，连接便会运行，而从 `fromgate` 的输出值也会被重新检索。这便相应地使门自动执行逻辑过程。这种操作一直重复到所有的输入值都变为可用值并且最后的输出值变为问题中的门中所需要的输入值。总而言

之，组合电路逆向执行，找到必需的输入值来得到最后的输出结果。

```
def getPinA(self):
    if self.pinA == None:
        return input("Enter Pin A input for gate " + self.getName()+"-->")
    else:
        return self.pinA.getFrom().getOutput()
```

代码 1.21(intro_1.21)

下列代码片段实现了上段所述的组合逻辑电路：

```
>>> g1 = AndGate("G1")
>>> g2 = AndGate("G2")
>>> g3 = OrGate("G3")
>>> g4 = NotGate("G4")
>>> c1 = Connector(g1,g3)
>>> c2 = Connector(g2,g3)
>>> c3 = Connector(g3,g4)
```

代码 1.22(intro_1.22)

两个与门（`g1` 和 `g2`）的输出结果与或门（`g3`）相连接，而 `g3` 的输出结果与非门（`g4`）相连。最后，非门的输出值就是整个电路的输出值。例如：

```
>>> g4.getOutput()
Pin A input for gate G1-->0
Pin B input for gate G1-->1
Pin A input for gate G2-->1
Pin B input for gate G2-->1
0
```

代码 1.23(intro_1.23)

用 ActiveCode 4 自己尝试一下吧

```
class LogicGate:
    def __init__(self,n):
        self.name = n
        self.output = None
    def getName(self):
        return self.name
    def getOutput(self):
        self.output = self.performGateLogic()
        return self.output
    class BinaryGate(LogicGate):
    def __init__(self,n):
        LogicGate.__init__(self,n)
        self.pinA = None
        self.pinB = None
```

```
    def getPinA(self):
        if self.pinA == None:
            return int(input("Enter Pin A input for gate "+self.getName()+"-->"))
        else:
            return self.pinA.getFrom().getOutput()
```

```
    def getPinB(self):
        if self.pinB == None:
            return int(input("Enter Pin B input for gate "+self.getName()+"-->"))
        else:
            return self.pinB.getFrom().getOutput()
```

代码 1.24 完整的逻辑电路

牛刀小试

1. 创建一个有两个新逻辑门的类，一个叫做 NorGate ，另一个叫做 NandGate。NandGate 是一个 AndGate 输出前面连接一个“非”门 Not；NorGate 是一个 OrGate 输出前面连接一个“非”门 Not。

2. 创建一系列的门，证明下面的等式 `NOT (( A and B) or (C and D))` 与 `NOT( A and B ) and NOT (C and D)` 等价。确保你能用到自己在仿真时用到的新的门。

小结

- 计算机科学是研究问题解决的学科；
- 计算机科学以抽象为工具来表现过程与数据；
- 抽象数据类型允许编程者通过隐藏数据的细节来管理问题域的复杂度；
- Python 是一门强大的，而同时又易于使用的面向对象的程序设计语言；
- 列表，元组和字符串都是 Python 内置的有序集合；
- 字典和集合是数据的无序集合；
- 类允许编程者执行抽象数据类型；
- 编程者既可以重载既有方法，也可以编写新的方法；
- 类可以被组织为阶层架构；
- 一个类的构造器在继续自己的数据和行为前，总要调用它父类的构造器。

关键词

抽象数据类型	**抽象**	**算法**
类	**可计算的**	**数据抽象化**
数据结构	**数据类型**	**深相等**
字典	**封装**	**异常**
格式运算符	**格式化字符串**	**HAS-A 关系**
实现独立/实现无关	**信息隐藏**	**继承**
继承阶层架构	**接口**	**IS-A 关系**
列表	**列表解析**	**方法**
可变性	**对象**	**过程抽象化**
编程	**提示符**	**自身**
引用相等	**仿真**	**字符串**
子类	**超类**	**真值表**

问题讨论

1. 为大学校园里的人们构建一个类的阶层架构，包括教员，职员和学生。他们有什么共同点？是什么将他们彼此区分开来？

2. 为银行账户构建一个类的阶层架构。

3. 为不同类型的电脑构建一个阶层架构。

4. 用这一章中提到的类，构建一个交互式回路并对它进行测试。

编程练习

1. 执行一个简单方法 `getNum` 和 `getDen`，以返回分数的分子和分母。

2. 很多情况下我们希望分数一开始就能保持最简比。请修改 `Fraction` 类的构造器，让 `GCD` 可以对分数立即进行约分。注意这意味着`__add__`函数不再需要执行约分的功能。请进行必要修改。

3. 执行剩下的简单代数运算操作(`__sub__`, `__mul__`, 和 `__truediv__`)。

4. 执行剩下的关系型操作(`__gt__`, `__ge__`, `__lt__`, `__le__`, 和`__ne__`)。

5. 修改 `fraction` 类的构造器，使它能够检查以确定分数的分子和分母都是整数，如果有一个不是，则用 raise 语句抛出异常。

6. 在分数的定义中我们认为负分数有负分子和正分母。使用负分数可能导致一些关系型操作出错。通常来讲，这是个不必要的限制。请通过修改构造器的方式，使用户略过负数，让各种操作都能正确运行。

7. 搜索`__radd__`方法。它和`__add__`有什么不同？什么时候需要用到？执行`__radd__`。

8. 重复上题步骤，不过这次考虑`__iadd__`方法。

9. 搜索`__repr__`方法。它和`__str__`有什么不同？什么时候需要用到？执行`__repr__`。

10. 搜索其他逻辑门（如 `NAND`，`NOR` 和 `XOR`）。把它们加入组合逻辑电路中。你需要做多少额外编码？

11. 最简单的数字电路就是半加器，请搜索并运行这个电路。

12. 现在扩展上题数字电路并运行一个 8 bit 的全加器。

13. 本章展示的模拟电路是逆向执行的。换句话说，给定一个电路，为计算输出值，该电路通过输入值进行回溯，这期间也会用到一些别的输出结果。这个过程一直进行，直到找到需要用户从外界输入的值。请修改运行机制，以使程序正向运行，即电路一接受输入值就开始计算结果。

14. 设计一个类来表示一副纸牌。用两个类来运行你最喜欢的纸牌游戏。

15. 从报纸上找一个数独游戏，写一个程序解决它。

二． 算法分析

2.1.目标

- 了解为何算法分析的重要性
- 能够用大“O”表示法来描述算法执行时间
- 了解在 Python 列表和字典类型中通用操作用大“O”表示法表示的执行时间
- 了解 Python 数据类型的具体实现对算法分析的影响
- 了解如何对简单的 Python 程序进行执行时间检测

2.2.什么是算法分析

计算机初学者经常将自己的程序与他人的比较。你也可能注意到了电脑程序常常看起来很相似，尤其是那些简单的程序。一个有趣的问题出现了，当两个看起来不同的程序解决相同的问题时，一个程序会优于另一个吗？

为了回答这个问题，我们需要记住的是，程序和它所代表的基本算法有着重要差别。在第一章中我们说到，算法是问题解决的通用的分步的指令的聚合。这是一种能解决任何问题实例的方法，比如给定一个特定的输入，算法能产生期望的结果。从另一方面看，一个程序是用某种编程语言编码后的算法。同一算法通过不同的程序员采用不同的编程语言能产生很多程序。

为进一步探究这种差异，请阅读接下来展示的函数。这个函数解决了一个我们熟知的问题，计算前 n 个整数的和。其中的算法使用了一个初始值为 0 的累加变量的概念。解决方案是遍历这 n 个整数，逐个累加到累加变量。

```
def sum_of_n(n):
    the_sum =0
    for i in range(1,n+1):
        the_sum = the_sum + i
    return the_sum
print(sum_of_n(10))
```

代码 2.1 前 n 个正整数求和（active1）

现在看下面的 foo 函数。可能第一眼看上去比较奇怪，但是进一步观察你会发现，这个函数所实现的功能与之前代码 2.1 中的函数基本相同。看不太懂的原因是糟糕的编码。我们没有使用好的变量命名来增加可读性，并且在累加过程中使用了多余的赋值语句。

回到前面我们提出的问题：是否一个程序会优于另一个？答案取决于你自己的标准。如果你关心可读性，那么 sum_of_n 函数肯定比 foo 函数更好。实际上，在你的编程入门课程上你可能见过很多这样的例子，因为这些课程的目标之一就是帮助你编写更具可读性的程

代码 2.2 另一种前 n 个正整数求和（active2）

```
def foo(tom):
    fred=0
    for bill in range(1,tom+1):
        barney = bill
        fred = fred + barney
    return fred
print (foo(10))
```

序。然而，在这门课程中，我们主要感兴趣的是算法本身的特性。（我们当然希望你可以继续努力写出更具可读性的代码。）

```
import time
def sum_of_n_2(n):
    start = time.time()
    the_sum = 0
    for i in range(1,n+1):
        the_sum = the_sum + i
    end =time.time()
return the_sum, end - start = the_sum + i
```

算法分析主要就是从计算资源的消耗的角度来评判和比较算法。我们想要分析两种算法并且指出哪种更好，主要考虑的是哪一种可以更高效地利用计算资源。或者占用更少的资源。从这个角度，上述两个函数实际上是基本相同的，它们都采用了一样的算法来解决累加求和问题。

从这点上看，思考我们通过计算资源这个概念真正想表达的是什么是非常重要的。有两种方法来看待它。一种是考虑算法解决问题过程中需要的存储空间或内存。问题解决方案所需的存储空间通常是由问题本身情况影响的，然而，算法常常有本身特定的空间需求，在这种情况下，我们需要很小心地解释这种变化。

作为空间需求的一个可替代物，我们可以用算法执行所需时间来分析和比较算法。这种方法有时被称为算法的“执行时间”或“运行时间”。我们可以检测 sum_of_n 函数的运行时间的一种方法是做基准分析。这意味着我们将监测程序运行出结果所需要的时间。在 Python 中，我们可以考虑我们使用的系统通过标定开始时间和结束时间来作为标准衡量一个函数。在 time 模块有一个叫 time 的函数，它能返回在某些任意起点以秒为单位的系统当前时间。通过在开始和结束时两次调用这个函数，然后计算两次时间之差，我们就可以得到精确到秒（大多数情况为分数）的运行时间。

这个编码展示了在原始 sum_of_n 函数的求和前后插入时间调用的情况。这个函数返回一个元组，包括累加和及计算所需时间（以秒为单位）。如果我们连续运行这个函数 5 次，每次都完成 1 到 10,000 的累加，我们得到的结果如下：

```
>>>for i in range(5):
        Print(“Sum is %d required %10.7f seconds”% sum_of_n_2(10000))
Sum is 50005000 required 0.0018950 seconds
Sum is 50005000 required 0.0018620 seconds
Sum is 50005000 required 0.0019171 seconds
Sum is 50005000 required 0.0019162 seconds
Sum is 50005000 required 0.0019360 seconds
>>>
```

```
>>>for i in range(5):
        Print("Sum is %d required %10.7f seconds"% sum_of_n_2(1000000))
Sum is 500000500000 required 0.1948988 seconds
Sum is 500000500000 required 0.1850290 seconds
Sum is 500000500000 required 0.1809771 seconds
Sum is 500000500000 required 0.1729250 seconds
Sum is 500000500000 required 0.1646299 seconds
>>>
```

我们发现这个时间是相当一致的，执行这段代码平均需要 0.0019 秒。那么如果我们累加到 100,000 会怎么样呢？

又是这样，每次运行所需时间虽然更长，但非常一致，平均约为之前的 10 倍，进一步累加到 1,000,000 我们得到：

在这种情况下，平均时间再次约为之前的 10 倍。

现在考虑接下来的编码，它展示了一种解决求和问题的不同的方法。这个函数，sum_of_n_3,利用一个封闭方程$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$去完成无迭代的累加到 n 的计算。

```
def sum_of_n_3(n):
    return (n * (n +1)) / 2
```

代码 2.3 无迭代求和

如果我们对 sum_of_n_3 做同样的基准检测，n 赋 5 个不同的值（10000,100000,1000000,10000,000 和 100000000），我们得到的结果如下:

```
Sum is 50005000 required 0.00000095 seconds

Sum is 5000050000 required 0.00000191 seconds

Sum is 500000500000 required 0.00000095 seconds

Sum is 50000005000000 required 0.00000095 seconds

Sum is 5000000050000000 required 0.00000119 seconds
```

对于输出我们需要关注两个关键点。第一，这种算法的运行时间比之前任何例子都短很多。第二，不管 n 取值多少运行时间都非常一致。看上去 **sum_of_n_3** 的运行时间几乎不受需要累计的数目的影响。

但是这个基准测试真正告诉了我们什么？直观上，我们可以看到迭代算法似乎做了更多的工作，因为一些程序步骤被重复执行，这可能就是它需要更长时间的原因。此外，迭代算法所需要的运行时间似乎会随着 n 值的增大而增大。然而，还有一个问题。如果我们在不同的计算机上运行相同的函数，或者使用不同的编程语言，我们可能会得到不同的结果。如果计算机比较老旧，运行 sum_of_n_3 可能还需要更长的时间。

我们需要更好的方法来衡量算法运行的时间。基准测试技术可以计算世界运行时间，然而它并没有真的为我们提供一个有用的度量指标。因为这个指标依赖于特定的机器、程序、运行时段、编译器和编程语言。相反，我们需要一个不依赖程序或者使用的机器的指标。这种度量指标将有助于判断算法优劣，并且可以用来比较算法的具体实现。

2.2.1. 大“O”表示法

当我们试图用执行时间作为独立于具体程序或计算机的度量指标去描述一个算法的效率时，确定这个算法所需要的操作数或步骤数显得尤为重要。如果把每一小步看作一个基本计量单位，那么一个算法的执行时间就可以表达为它解决一个问题所需的步骤数。制定一个合适的基本计量单位是一个很复杂的问题，并且还要依赖于算法具体是怎样执行的。

当我们比较上述几个求和算法时，统计执行求和的赋值语句的次数可能是一个好的基本计量单位。在 **sum_of_n** 函数中，赋值语句的数量是 1（**the_sum=0**）加上 n（我们执行 **the_sum=the_sum+i** 的次数）。

我们可以用一个叫 T 的函数来表示赋值语句数量，如上例中 T(n) = 1 + n。变量 n 一般指“问题的规模”，可以理解为当要解决一个规模为 n，对应 n+1 步操作步数的问题，所需要的时间为 T(n)。

在前一节给出的求和函数中，用求和的项数来代表问题的规模是合理的。可以认为求前 100000 项整数的问题规模大于求前 1000 项整数。因此，求解大规模问题所需时间显然比求解小规模要多一些。那么我们的目标就是寻找程序的运行时间如何随着问题规模变化。

计算机科学家对这种算法分析技术进行了更为深远的思考。有限的操作次数对于 T(n)函数的影响，并不如某些占据主要地位的操作部分重要。换言之，当问题规模越来越大时，T(n)函数中的一部分几乎

掩盖了其他部分对于函数的影响。最终，这种起主导作用的部分用来对函数进行比较。**数量级函数**用来描述当规模 n 增加时，T(n)函数中增长最快的部分。这种数量级函数一般被称为**大“O”**表示法，记作 O(f(n))。它提供了计算过程中实际步数的近似值。函数 f(n)是原始函数 T(n)中主导部分的简化表示。

在上面的例子中，T(n) = 1 + n。当 n 增大时，常数 1 对于最后的结果来说越来越不重要。如果我们需要 T(n)的近似值，我们可以忽略常数 1，直接认为 T(n)的运行时间就是 O(n)。值得明白的是，1 对 T(n)固然重要，但是当 n 增加到很大时，忽略 1 得到的近似值同样精确。

另一个例子，假设某个算法程序实际步骤的精确值是 $T(n) = 5n^2 +27n+1005$。当 n 很小时，比如为 1 或 2 时，常数 1005 似乎对函数起到主要作用。但是当 n 越来越大时，n^2 项则起到最主要的作用。实际上，当 n 非常大时，其余两项对于最终的结果已经无足轻重了。与上个例子相似，当 n 越来越大时，我们就可以忽略其余项，只关注用 $5n^2$ 来代表 T(n)的近似值了。同样，系数 5 的作用也会越来越小，也可以忽略。我们就会说函数 T(n)的数量级 $f(n) = n^2$，即 $O(n^2)$。

尽管前面求和函数的例子没有体现，但我们还是注意到有时算法的运行时间还取决于具体数据而不仅仅是问题规模。对于这种算法，我们把它们的执行情况分为最好的情况，最坏的情况和平均情况。某个特定的数据集会使算法程序执行情况极差，这就是最坏的情况。然而另一个不同的数据集却能使这个算法程序执行情况极佳。不过，大多数情况下，算法程序的执行情况都介于这两种极端情况之间，也就是平均情况。因此要理解不同情况之间的差别，不被某些特殊情况所误导。

在你之后的算法学习中，你会不断见到表 2.1 所示的常见数量级函数。为了确定这些函数中哪些在 T(n)占主导作用，就要在 n 增大时对它们进行比较。

f(n)	函数名
1	常数函数
log n	对数函数
n	线性函数
n log n	线性对数函数
n^2	二次函数
n^3	三次函数
2^n	指数函数

表格 2.1 常见函数的大“O”表示法

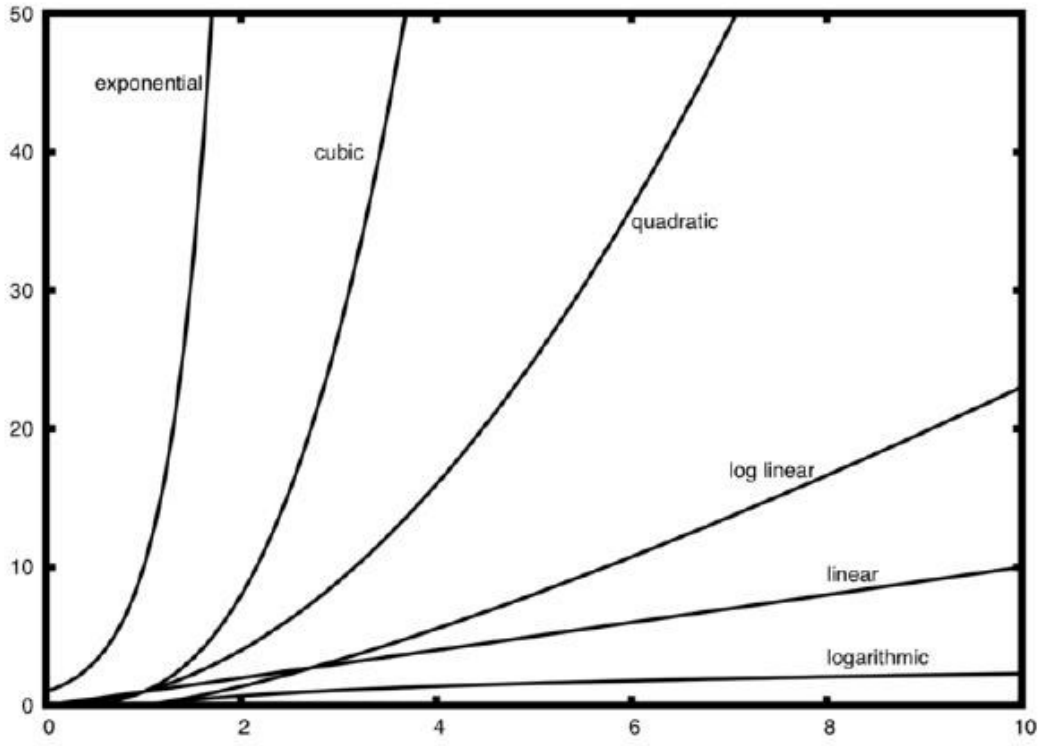


图 2.1 常见大“O”函数的图像

图 2.1 是表 2.1 中常见函数的图像。注意到当 n 很小时，函数之间不易区分，很难说谁占主导。然而，当 n 增大时，就能观察到明显的区别，很容易进行比较。

最后再举一个例子，假如我们有如下所示的一个 Python 代码片段，。尽管这个程序实现不了什么功能，但能指导我们如何对代码进行执行分析。

```
a = 5
b = 6
c = 10
for i in range(n):
        for j in range(n):
            x = i * i
            y = j * j
            z = i * j
for k in range(n):
    w = a * k + 45
    v = b * b
d = 33
```

代码 2.8

我们发现，任务操作总数分为四项。第一项是常数 3，代表程序开头的三个赋值语句。第二项是 $3n^2$，因为嵌套迭代（循环结构）中有三个赋值语句分别被重复执行了 n^2 次。第三项是 2n，表示两个赋值语句被重复执行了 n 次。最后一项是常数 1，代表最后的赋值语句。我们得到 $T(n) = 3+3n^2+2n+1 = 3n^2+2n+4$。看到指数项，我们自然地发现 n^2 项占主导，当 n 增大时，其他项和主导项的系数都可以忽略，所以这个代码片段的数量级就是 $O(n^2)$。

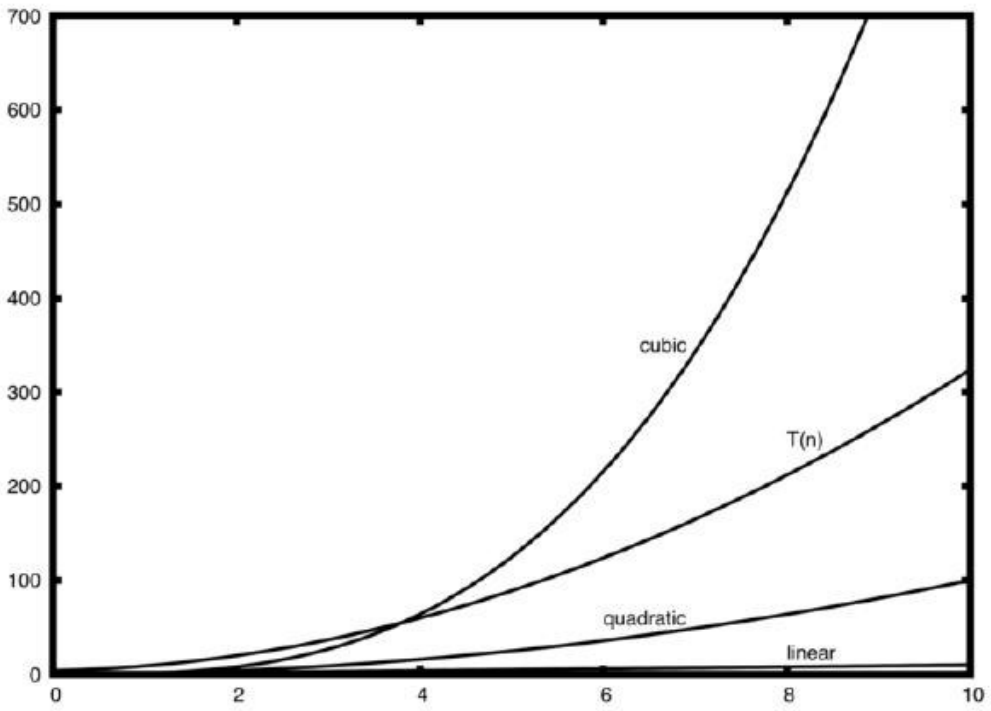


图 2.2　常见的大“O”函数与 T(n)函数对比

表 2.2 表示常见的大“O”函数与前面讨论到的 T(n)进行比较。起初 T(n)函数比三次函数大一些。但是，当 n 增大时，三次函数很快超过了 T(n)。不难发现，随着 n 增大，T(n)函数越来越接近二次函数。

小试牛刀

1.编写两个 Python 函数来寻找一个列表中的最小值。函数一将列表中的每个数都与其他数作比较，数量级是 $O(n^2)$.函数二的数量级是 O(n)。

2.2.2 变位词检测

经典的字符串变位词检测问题是比较不同数量级函数算法的一个典型例子。如果一个字符串是另一个字符串的重新排列组合，那么这两个字符串互为变位词。比如，”heart”与”earth”互为变位词，”python”与”typhon”也互为变位词。为了简化问题，我们设定问题中的字符串长度相同，都是由 26 个小写字母组成。我们需要编写一个接受两个字符串，返回真假，代表是否是一对变位词的布尔函数。

解法 1：检查标记

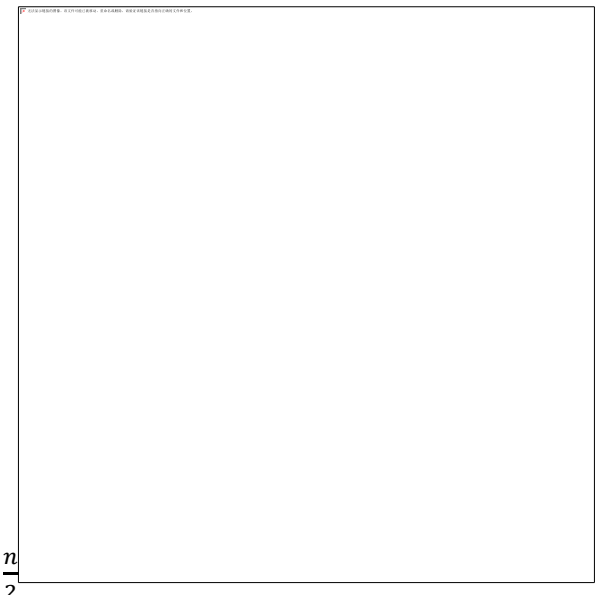
变位词问题的第一种解法是检查第一个字符串中的所有字符是不是都在第二个字符串中出现。如果能够把每一个字符都“检查标记”一遍，那么这两个字符串就互为变位词。检查标记一个字符要用特定值 None 来代替，作为标记。然而，由于字符串不可变，首先要把第二个字符串转化成一个

```
def anagram_solution1(s1,s2):
    a_list = list(s2)

    pos1 = 0
    still_ok = True

    while pos1 <len(s1) and still_ok:
        pos2 = 0
        found = False
        while pos2 <len(a_list) and not found:
            if s1[pos1] == a_list[pos2]:
                found = true
            else:
                pos2 = pos2 + 1
        If found:
            a_list[pos2] = None
        else:
            still_ok = false
        pos1 =pos1 + 1

    return still_ok
print(anagram_solution1('abcd','dcba')
```

列表。第一个字符串中的每一个字符都可以在列表的字符中去检查，如果找到，就用 None 代替以示标记。

代码 2.9

为了分析这个算法，我们要注意到 s1 中 n 个字符的每一个都会引起一个最多迭代到 s2 列表中第 n 个字符的循环。（考虑最坏的情况）列表中的 n 个位置各会被寻找一次去匹配 s1 中的某个字符，那么执行总数就是从 1 到 n 的代数和。我们之前提到过它可以这样表示：



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

当 n 变大时，n^2 相对于 n 将占主要部分，并且 1/2 可以忽略。因此，这个算法的复杂度是 $O(n^2)$。

```
Def anagram_solution2(s1,s2):
A_list1=list(s1)
A_list2=list(s2)

A_list1.sort()
A_list2.sort()

Pos=0
Matches=True

While pos < len(s1) and matches:
If    a_list1[pos] == a_list2[pos]:
Pos=pos + 1
Else:
Matches=False

Return matches
Print(anagram_solution2('abcde','edcba'))
```

代码 xxx 解法一

解法二：排序比较法

尽管 s1 和 s2 并不相同，但若为变位词它们一定包含完全一样的字符，利用这一特点，我们可以采用另一种方法。我们首先从 a 到 z 给每一个字符串按字母顺序进行排序，如果它们是变位词，那么我们将得到两个完全一样的字符串。此外，我们可以先将字符串转化为列表，再利用 Python 中内建的 sort 方法对列表进行排序。下面代码展示了这种方法。

第一眼看上去你可能会认为这个算法的复杂度是 O(n)，毕竟排序后只需要一个简单的循环去比较 n 个字符。然而对 Python 内建的 sort 方法的两次使用并非毫无消耗。事实上，正如我们在后面的章节中将要看到的，排序方法的复杂度往往都是 $O(n^2)$或者 $O(n\log n)$，所以排序贡献了这个函数主要的循环操作。最终，这个算法和排序的复杂度相同。

解法三：暴力匹配算法

解决这个问题的典型暴力方法是尝试所有的可能。为了解决变位词检测问题，我们可以简单地构造一个由 s1 中所有字符组成的所有可能的字符串的列表，并检查 s2 是否在列表中。然而这个方法有一个困难之处。当我们构造由 s1 中字符组成的所有可能字符串时，第一个字符有 n 个可能，第二个字符有 n-1 种可能，第三个则是 n-2 种，以此类推。所有可能字符串的总数是 n*(n-1)*(n-2)*...*3*2*1。也就是 n!。尽管这些字符串中的一些可能是重复的，但程序不能提前预见到，所以还是会产生 n!个字符串。

事实上当 n 变大时，n!增长的比 2^n还要快。如果 s1 有 20 个字符，将会有 20!=2,432,902,008,176,640,000 个可能的字符串。如果我们每秒进行一个尝试，这将会花费我们 77,146,816,596 年去尝试所有列表。这大概不是一个好方法。

解法四：计数比较法

解决变位词问题的最后一个方法利用了任何变位词都有相同数量的 a，相同数量的 b，相同数量的 c 等等。为判断两个字符串是否为变位词，我们首先计算每一个字符在字符串中出现的次数。由于共有 26 个可能的字符，我们可以利用有 26 个计数器的列表，每个计数器对应一个字符。每当我们看到一个字符，就在相对应的计数器上加一。最终，如果这两个计数器列表相同，则这两个字符串是变位词。下面展示了这种方法：

代码 xxx 计数比较法

```
Def anagram_solution4(s1,s2):
C1 = [0] * 26
C2 = [0] * 26

For i in range(len(s1)):
Pos = ord(s1[i]) - ord('a')
C1[pos] = c1[pos] + 1
        For i in range(len(s2)):
Pos = ord(s2[i]) - ord('a')
C2[pos] = c2[pos] + 1

J = 0
Still_ok = True
Wile j < 26 and still_ok:
If c1[j] == c2[j]:
J = j+1
Else:
Still_ok=False

Return still_ok

Print(anagram_solution4('apple','pleap'))
```

同样，这个方法有一些循环操作。然而不同于第一个方法，所有循环都不是嵌套的。前两个计数字符数的循环都是 n 重。而因为字符串中总共有 26 种可能的字符，第三个比较两个计数列表的循环总是执行 26 步。把它们全部加起来就得到 T(n)=2n+26，也就是 O(n)。这样，我们就找到了一个解决这个问题的线性复杂度的算法。

在结束这个问题之前，我们需要讨论一些关于空间需求的事情，尽管最后一个方法可以以线性的时间复杂度来运行，但是这是以使用了额外的空间来存储两个计数器列表为代价的。换句话说，这个算法牺牲了空间来换取时间。

这是一个常见的现象。很多情况下你需要在时间和空间的权衡中做出选择。在这个例子中，额外的空间消耗并不足道。但是如果可能的字母多达几百万种，这将是一个问题。作为一个计算机科学家，当要做出算法选择时，需要你根据具体问题来决定利用计算资源的最好方式。

牛刀小试

Q-1：判断下列代码段的大 O 级别

```
test = 0
for i in range(n):
   for j in range(n):
      test = test + i * j
```

- O(n)
- O(n^2)
- O(log n)
- O(n^3)

Q-2：判断下列代码段的大 O 级别

```
test = 0
for i in range(n):
   test = test + 1

for j in range(n):
   test = test - 1
```

- O(n)
- O(n^2)
- O(log n)
- O(n^3)

Q-3：判断下列代码段的大 O 级别

2.3 PYTHON 数据结构的性能

既然大家已经大体了解了大 O 表示法以及不同算法间复杂度的差异，在这一节我们的目标是来讲授一些关于 Python 中列表和字典操作的大 O 复杂度的知识。然后我们将展示一些计时操作，以此来阐明在各种数据结构上使用某个操作的优缺点。这对你去理解这些 Python 数据结构的性能是非常重要的，因为随着我们在本书接下来的部分学习其他数据结构，它们将是我们需要使用到的构件。在本节中，我们不打算去解释这些性能为什么是这样的问题，不过在接下来的章节中，我们将看到一些列表和字典的实现方式以及它们的操作是如何依赖于这些实现方式的。

2.3.1 列表 LIST

当要执行列表数据结构时，Python 的编写者有多种方式去实现这个目的。这些方式中的每一个都对列表运行的速度产生影响。为了做出正确的选择，他们着眼于人们最经常使用的列表数据结构的方式，然后完善他们的列表操作手段，所以常用的操作是非常快的。当然，他们也会尽力提高那些不常用操作的速度。但是当不得不去做一个权衡处理时，不常用操作常常被牺牲用来支持那些更常用操作。

索引和分派到一个索引位置是两个常见操作，它们无论列表多大，操作花费的时间都相同。当一个操作的速度像这样不依赖于列表的大小，那么这个操作就是 O(1)。

另一个非常常用的程序操作是去扩充一个列表。这有两种方式去生成一个更长的列表。你可以用“append”操作或者串联运算符。这个“append”操作是 O(1)。然而，串联运算符是 O(k)，这里 k 是指正在被连接的列表的大小。了解它对你非常重要，因为通过选择工作的正确工具可以使你的程序更加有效。

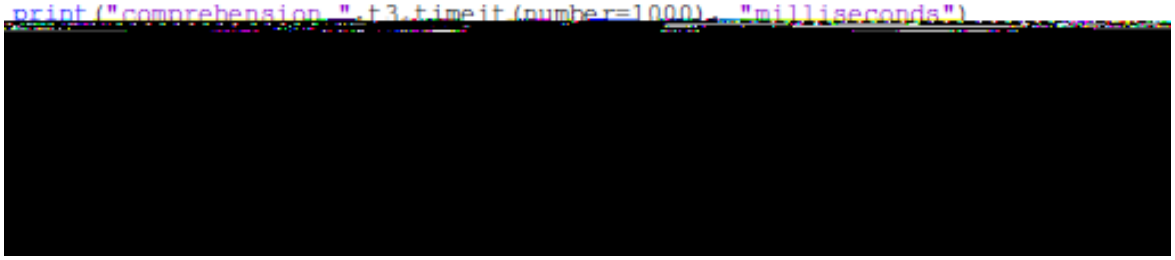
让我们来看一下四种不同的方法来生成从 0 到 n 的列表。首先，我们尝试用 for 循环体通过串联生成一个列表，然后我们可以用“append”代替串联操作。接下来，我们可以使用列表解析来生成一个列表。最后，也许是最明显的方法，通过列表结构体的访问来使用“range”的功能。下面展示了生成列表四种方法的代码。

```
def test1():
    l = []
    for i in range(1000):
        l = l + [i]

def test2():
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))
```

为了获取我们程序运行所需要的时间，我们需要引用 Python 中的 timeit 模块。这个 timeit 模块是被设计成在一个持续稳定的环境中，尽可能使用与计算机操作系统相似的计时机制，让 Python 的开发者实现跨平台运行时间的测量。

为使用 timeit 模块，你需要创建一个 Timer 对象，这个对象的参数是两个 Python 语句。第一个参数是你想进行计时的 Python 语句；第二个参数是建立这次测试你将要运行的语句。timeit 模块就将测量运行这个语句一定次数多花费的时间。如果不加要求，timeit 模块的默认运行次数是一百万次。运行结束后，它将以浮点数的形式返回运行的总时间（单位：秒）。但是，由于它默认运行语句一百万次，当你执行程序一次时，它返回的结果是以微秒为单位的。你也可以在 timeit 中附上一个名叫 number 的参数，这样你就可以指定程序被执行的次数。下图将展示对我们的每一个程序执行 1000 次，分别需要花费的时间。

```
# Import the timeit module
import timeit
# Import the Timer class defined in the module
from timeit import Timer
# If the above line is excluded, you need to replace Timer with
   timeit.Timer when defining a Timer object
t1 = Timer("test1()", "from __main__ import test1")
print("concat ",t1.timeit(number=1000), "milliseconds")
t2 = Timer("test2()", "from __main__ import test2")
print("append ",t2.timeit(number=1000), "milliseconds")
t3 = Timer("test3()", "from __main__ import test3")
print("comprehension ",t3.timeit(number=1000), "milliseconds")
```



在上面的实验中，我们进行测试的语句是调用函数到 test1（），test2（），等等。这个设置语句可能会让你感觉到十分奇怪，所以让我们更加细致的考虑它。你可能对 from，import 语句十分熟悉，但这个语句常常被运用在 Python 程序文本的开头。在这种情况下，from __main__ import test1 这个语句将来自 __main__命名空间的函数 test1 调用到为进行时间测量而建立的 timeit 命名空间中去。Timeit 模块这样做是以为它需要在某中特定的环境中运行，这种环境要求将你可能已经生成的，或已经偏离的，或可能以某种不可预测的方式对你程序的运行产生干扰的变量进行整理。

从上面的实验中，我们可以清楚的看到，运行时间为 0.30 毫秒的 append 操作明显快于运行时间为 6.54 毫秒的串联操作。在上面的实验中，我们还展示了创建一个列表（list）的两种额外的方式：调用 range 进行列表创建和列表推导。我们惊喜的发现列表推导的速度足足是在 for 循环中逐个 append 操作的两倍。

关于这个小实验的最后一个结论是，上面我们所看到的所有运行时间，都包括了进行访问测试所耗费的时间。但我们可以认定，在这四种情形中，访问操作所耗费的时间是完全相同的，所以我们对这四种操作运行时间的比较是有意义的。它不能准确的说串联操作耗时 6.54 毫秒，应该说串联操作测试函数耗时 6.54 毫秒。作为练习，你可以测试调用一个空函数所耗费的时间，并把这个时间从上面测试得出的结果中减去。

既然我们已经了解了如何对操作进行具体的测量，那么你可以看图表 2.2，了解所有列表基本操作的大 O 效率。在你对图表 2.2 进行仔细思考后，可能会对 pop 操作的两个不同的时间感到疑惑。当 pop 操作每次从列表的最后一位删除元素时复杂度为 O（1），而将列表的第一个元素或中间任意一个位置的元素删除时，复杂度则为 O（n）。这样迥然不同的结果是由 Python 对列表的执行方式造成的。在 Python 的执行过程中，当从列表的第一位删除一个元素，其后的每一位元素都将向前挪动一位。你可能觉得这种操作很愚蠢，但当你仔细看完图表 2.2 后会发现这种执行方式会让 index 索引操作的复杂度降为 O（1）。这种运行时间的权衡是 Python 执行者认为正确高效的。

Operation	Big-O Efficiency
indexx[]	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n+k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

Table 2.2: Big-O Efficiency of Python List Operators

为进一步论证这些操作表现的差异性，让我们用 timeit 模块进行另一个实验。我们的目标是能够核实以下两种 pop 操作的表现，第一种是在一个已知长度的列表中从列表的末端删除元素，而第二种则是从这个列表的开头删除元素。我们还想测量出对不同长度的列表进行 pop 操作的时间。我们想要看到的是，随着列表长度的增加，从列表末端删除元素的 pop 操作时间保持稳定，而从列表开头删除元素的 pop 操作则随着长度的增加而增加。

下面的代码展示了对这两种 pop 操作的区别进行测量的一个尝试。正如你从第一个例子中看见的那样，从列表末尾删除的 pop 操作耗时 0.0003 毫秒，然而从列表开头删除则耗时 4.82 毫秒。对于一个有 2 百万个元素的列表，两种操作耗费的时间相差 16000 倍。

```
pop_zero = Timer("x.pop(0)",
                 "from __main__ import x")
pop_end = Timer("x.pop()",
                "from __main__ import x")

x = list(range(2000000))
pop_zero.timeit(number=1000)
4.8213560581207275

x = list(range(2000000))
pop_end.timeit(number=1000)
0.00031614303588886719
```

对于这个代码，我们有很多地方值得去注意。第一点就是 from __main__ import x 语句。尽管我么并没有定义一个函数，但我们仍然想在测试中可以使用列表对象 x。这种方式使我们可以对单个 pop 操作进行计时，并且使我们得到对单个操作最准确的测量时间。因为计时操作执行 1000 次，所以必须指出的是在循环过程中每进行一次操作，列表都将缩短一位。但是由于起始列表有两百万个元素，进行 1000 次操作仅减少了 0.05%，这种改变是很微小的，几乎无影响。

尽管第一个测试证明了 pop（0）操作确实比 pop（）操作要慢，但并不能确切的说明 pop（0）复杂度为 O（n）而 pop（）的复杂度为 O（1）。为了证实这个结果，我们需要综合不同列表长度的测试结果。

```
pop_zero = Timer("x.pop(0)", "from __main__ import x")
pop_end = Timer("x.pop()", "from __main__ import x")
print("pop(0) pop()")
for i in range(1000000,100000001,1000000):
    x = list(range(i))
    pt = pop_end.timeit(number=1000)
    x = list(range(i))

pz = pop_zero.timeit(number=1000)
print("%15.5f, %15.5f" %(pz,pt))
```

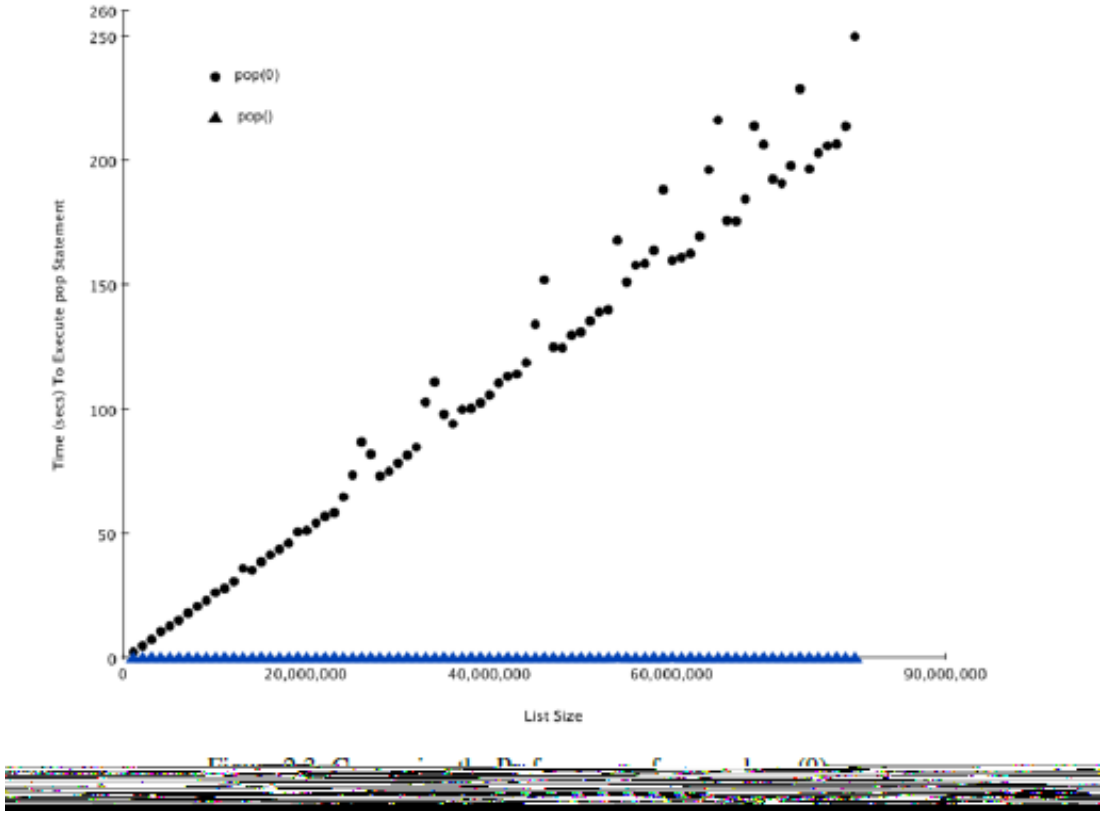


图 2.3 pop 与 pop(0) 操作性能的比较

图 2.3 展示了我们实验的结果。 可以看到，随着列表的逐渐变长，pop(0)所需要的时间也同样增加。但是 pop() 的时间却始终保持基本不变。不出所料，这正是 **O(n)**与 **O(1)** 算法的行为。

我们小实验的误差可能来源于我们计算机上同时运行的其他进程。当我们测量时间时，这些进程可能使我们的代码运行变慢。即使我们试图最小化计算机上其他事情的影响，用时上也总是存在着一定的不确定度。这就是为什么我们要将测试循环上千次——为了获得足够的信息以使结果具有足够的可信度。

2.3.2 字典

Python 中第二个主要的数据结构是字典。回想一下，字典与列表的不同之处在于你需要通过一个键（key）来访问条目，而不是通过一个坐标。在本书的后面我们会介绍多种字典的实现方式，不过现在我们要说的重点是，字典条目的访问和赋值都是 **O(1)**的时间复杂度。字典的另一个重要的操作是所谓的“包含”。检查一个键是否存在于字典中也只需 **O(1)**的时间。其他字典操作的时间效率都已经在 表 2.3 中列出。需要注意的是，这里所列出的都是平均时间复杂度。在一些罕见的情况下，包含、访问和赋值都可能退化为 **O(n)**，不过这将是我们之后研究字典的实现时才会讨论的内容。

操作	时间效率（大 O 表示法）
复制	O(n)
访问	O(1)

赋值	O(1)
删除	O(1)
包含（in）	O(1)
迭代	O(n)

表格 2.2 字典操作效率表（大 O 表示法）

我们的最后一个性能实验将会对比列表和字典的包含操作的效率。在这一过程中我们将会验证列表的包含操作是 **O(n)**，为字典的是 **O(1)**。这个实验很简单，我们将生成一个自然数（range）的列表，然后随机地选取一个数字，检查其是否在列表中。如果我们之前的效率表是正确的话，列表越大，所用的时间也就越长。

然后我们将在一个以数字为键的字典上重复这个实验。这次我们会发现检查一个数字是否在字典中要快得多，而且即使字典变大，检查所用的时间也保持不变。

下面的代码实现了这个比较。请注意我们实际上是在做完全相同的操作。区别仅仅在于在第 7 行，**x** 是一个列表，而在第 9 行的 **x** 是一个字典。

```
1 import timeit
2 import random
3
4 for i in range(10000,1000001,20000):
5 t = timeit.Timer("random.randrange(%d) in x"%i,
6 "from __main__ import random,x")
7 x = list(range(i))
8 lst_time = t.timeit(number=1000)
9 x = {j:None for j in range(i)}
10 d_time = t.timeit(number=1000)
11 print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))
```

代码 xxx 对比列表和字典的包含操作的效率

图 2.4 总结了我们的结果。可以看到，字典始终比列表快，对于最小的只包含 10, 000 个元素的情况，字典比列表快 89.4 倍，而在最大的 990, 000 个元素时，字典要快 11603 倍！另外可以注意到，随着列表的增大，其包含操作所需要的时间是线性增长的，这证实了我们之前关于其时间复杂度是 **O(n)**的论断。与此同时，字典的包含操作用时保持不变，即使字典的大小不断变大也是如此。实际上，10,000 个元素的字典的包含操作用了 0.004 毫秒，而 990,000 个元素的字典同样也用了 0.004 毫秒。

图 2.4 对比列表和字典的包含操作的效率

因为 Python 是一门仍在发展中的语言，其幕后总在进行着各种改进。关于 Python 的数据结构性能的最新信息可以在 Python 网站上找到。对于这部分，Pythonwiki 上有一页关于时间复杂度的很好的介绍，可以在 Time Complexity Wiki 上找到。

牛刀小试

1. 下面的列表操作中哪一个是 O(1)的？

    A. list.pop(0)

    B. list.pop()

    C. list.append()

    D. list[10]

    E.以上全是

2. 下面的字典操作中哪一个是 O(1)的？

    A. '*x*' in my_dict

    B. del my_dict['*x*']

    C. my_dict['*x*'] == 10

    D. my_dict['*x*'] = my_dict['*x*'] + 1

E.以上全是

2.4 小结

- 算法分析是一种与具体实现无关的衡量算法（好坏）的方法。
- 大 O 表示法可以按照随着问题规模的变化时算法的主要变化行为把算法分类。

2.5 关键字

平均情况	大“O”表示法	暴力匹配算法
检查标记	指数（的）	线性（的）
对数线性（的）	对数（的）	数量级
二次（的）	时间复杂度	最坏的情况

2.6 问题讨论

1. 给出下面代码段的时间复杂度（大 O 表示法）：

```
for i in range(n):
k = 2 + 2
```

2. 给出下面代码段的时间复杂度（大 O 表示法）：

```
i = n
while i > 0:
k = 2 + 2
i = i // 2
```

3. 给出下面代码段的时间复杂度（大 O 表示法）：

```
for i in range(n):
for j in range(n):
k = 2 + 2
```

4. 给出下面代码段的时间复杂度（大 O 表示法）：

```
for i in range(n):
for j in range(n):
for k in range(n):
k = 2 + 2
```

5. 给出下面代码段的时间复杂度（大 O 表示法）：

```
for i in range(n):
k = 2 + 2
for j in range(n):
k = 2 + 2
for k in range(n):
k = 2 + 2
```

2.7 编程练习

1. 设计一个实验来验证列表索引操作是 O(1)的
2. 设计一个实验来验证字典的读取和赋值操作都是 O(1)的
3. 设计一个实验来比较列表列表和字典的删除（del）运算符的性能
4. 给出一个寻找一个随机数列中第 k 小的数的 O（nlog(n)）的算法
5. 你能把前面一题中的算法改进为 O（n）吗？解释原因

三．基本数据结构类型

3.1 学习目标

- 了解抽象数据类型：栈 stack、队列 queue、双端队列 deque 和列表 list；
- 用 Python 列表数据结构，来实现 stack/queue/deque 抽象数据类型的构建；
- 了解各种基本线性数据结构的性能和使用方法；
- 了解前缀、中缀和后缀表达式；
- 采用栈 stack 对后缀表达式进行求值；
- 采用栈 stack 将中缀表达式转换为后缀表达式；
- 采用队列 queue 进行基本的时间模拟；
- 能够明确问题类型，选用 stack、queue 或者 deque 中合适的数据结构；
- 能够采用节点和引用模式来将抽象数据类型 list 实现为链表；，
- 能够比较链表和列表的算法性能。

3.2 什么是线性结构？

在我们开始数据结构的学习之前，先来看看四个简单但非常强大的概念：栈，队列，双端队列，和列表。这四种数据集合的项的由添加或删除的方式整合在一起。当添加一个项目时，它就被放在这样一个位置：在之前存在的项与后来要加入的项之间。像这样的数据集合常被称为**线性数据结构**。

你可以想象线性结构有两个端，有时候，我们称这两端为“左”和“右”，或者“前”和“后”，或者“顶”和“底”。其实名字不重要，区别线性结构和其他结构的依据是项进行添加和删除的方式，尤其是添加和删除发生的位置。例如，有的结构可能仅允许仅在一端加项；有的结构可能会允许从两端移除项。

这些变化引起了计算机科学中的一些最有用的数据结构的出现。他们出现在许多算法中被用用来解决各种重要问题。

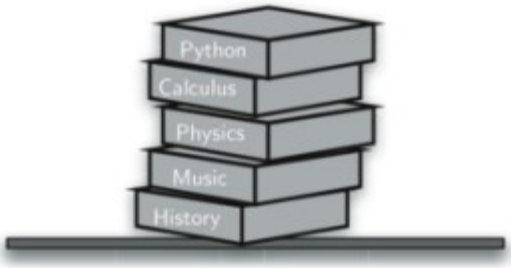
3.3 栈

3.3.1 什么是栈？

一个**栈**（有时称“叠加栈”）是一个项的有序集合。添加项和移除项都发生在同一“端”。这一端通常被称为“顶”。另一端的顶部被称为“底”。

栈的“底”是有标志性的，因为存储在栈中更靠近“底”的项就是栈中储存时间最长的项。最新添加的项在移除项时也会第一个被移除。这种排序原则有时也称为 **LIFO** 法，也就是**“后进先出”**。项的排序基于它在集合中存在的时间长度。越新的项越靠近“顶”，越老的项越靠近“底”。

在日常生活中有许多栈的实例。几乎所有的餐厅有一堆餐盘，你可以拿走最顶上的一个，排在你后面一个人将拿走原来你餐盘下的那个。又或者在桌上叠着一摞书（图 3.1）。只有最上面的一个本书的封面可以看见。如果要看到其他书，我们需要拿掉那些放在上面的书。图 3.2 是另一个栈结构。这包含着一些原始的 Python 数据对象。



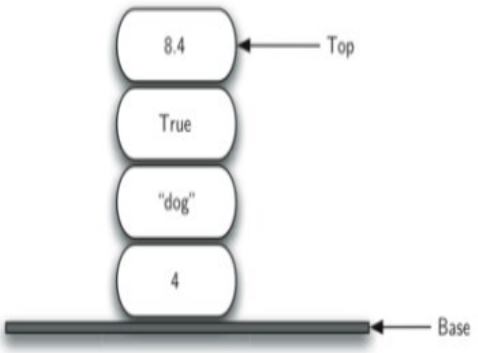


图 3.1：书的栈图 3.2： python 原始对象的栈

栈的最有用的一个思路，来自对项添加、然后移除的简单观察。假设你的面前有一张干净的桌面。现在把书一本一本地放在前一本项上。这时你就在构建一个栈。当你移除书籍时，它们被拆移除的顺序和它们被放置是的顺序完全相反。栈很重要，因为它们可以用于反转项的顺序。放入的顺序和拿出顺序相反。图 3.3 展示了 Python 数据对象的栈被创造、然后移除项时的情况。注意对象的顺序。

图 3.3：栈的逆转属性

考虑到这种逆转性，你可以想一想你使用电脑时栈发生的例子。例如，每个浏览器都有后退按钮。当你浏览网页时，那些网页就被放在栈中（实际上是将网址放在栈中）。你正在浏览的当前页被放在“顶”，你打开的第一个网页被放在“底”。如果你点击后退按钮，你就会开始以相反的顺序浏览网页。

栈操作	栈内容	返回值
s.is_empty()	[]	True

s.push(4)	[4]	
s.push('dog')	[4,'dog']	
s.peek()	[4,'dog']	'dog'
s.push(True)	[4,'dog',True]	
s.size()	[4,'dog',True]	3
s.is_empty()	[4,'dog',True]	False
s.push(8.4)	[4,'dog',True,8.4]	
s.pop()	[4,'dog',True]	8.4
s.pop()	[4,'dog']	True
s.size()	[4,'dog']	2

表 3.1：简单的栈操作

3.4 栈的抽象数据类型

栈的抽象数据类型是由以下结构和操作定义的。堆栈是结构化的，如上面所描述的，栈是一个有序的项的集，项添加和删除的一端称为“顶”。栈的命令是按后进先出进行的。栈的操作如下：

Stack()创建一个新的空栈。它不需要参数，并返回一个空栈。

Push(item)将新项添加到堆栈的顶部。它需要参数 item 并且没有返回值。

pop()从栈顶删除项目。它不需要参数,返回 item。栈被修改。

peek()返回栈顶的项，不删除它。它不需要参数。堆栈不被修改。

isEmpty()测试看栈是否为空。它不需要参数，返回一个布尔值。

size()返回栈的项目数。它不需要参数，返回一个整数。

例如，如果 S 是一个已存在的空栈，那么表 3.1 展示一些栈操作的结果。在栈中，栈顶的项目被列在最右边。

3.4. 队列

3.4.1. 什么是队列

队列（**Queue**）是一系列有顺序的元素的集合，新元素的加入在队列的一端，这一端叫做“队尾”（rear），已有元素的移除发生在队列的另一端，叫做“队首”（front）。当一个元素被加入到队列之后，它就从队尾开始向队首前进，直到它成为下一个即将被移出队列的元素。

最新被加入的元素必须处于队尾，在队列停留最长时间的元素处于队首。这种原则有时候叫做“**先进先出**”（**FIFO, first-in first-out**），有时候也叫做“先到先服务”。

有关队列最简单的例子就是我们每天都在排的队伍。我们排队看电影，在杂货店里排队等着付钱，在自助餐厅里我们也排队（这样我们就能从盘子组成的“**栈**（**Stack**）”里边拿出我们的盘子）。一个遵守秩序的队伍，或者队列，严格规定进出都只能有一条通道这。不可以插进队列的中间，也不能在到达队首之前提前离开队列。图 3.3 是一个简单的由 Python 数据对象构成的队列：

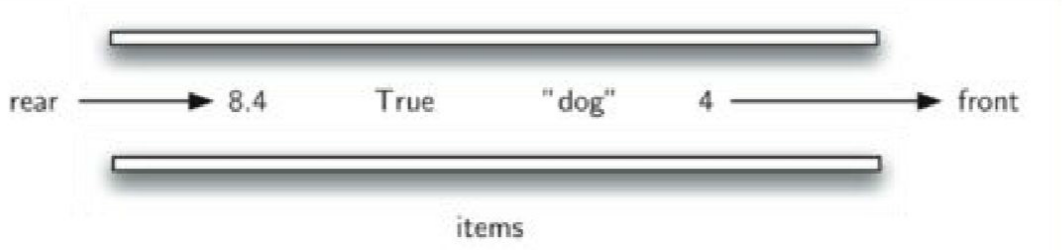


图 3.3 一个由 Python 数据对象构成的队列

计算机科学中也有许多常见的关于队列的例子。例如：假设我们的计算机实验室有 30 台电脑联网到一台打印机上。当学生们想要打印时，他们的打印任务就会“进入队伍”，其中有正在等待打印的其他任务。最先进入队伍的任务就是接下来要完成的。如果你的任务在队伍的最后，那么你就要等着在你前面的所有任务都完成。稍后我们还会进一步探讨这个有趣的例子的细节。

除了这种打印队列之外，操作系统还会使用一系列的队列对计算机中的进程进行控制，通常会使用一种能尽量快地执行程序、服务尽量多的用户的排队算法来决定接下来的操作。同样，在我们打字的时候，有时候键盘的敲击会比字母在屏幕上显示得更快。这是由于电脑那时候也在执行其它的任务。键盘的敲击信号被储存在一个类似于队列的缓冲区域，这样它们最终就能以正确的顺序显示在屏幕上。

3.4.2.抽象数据类型 QUEUE（队列）

抽象数据类型队列通过以下的一些结构和操作来定义。如前文所述，一个队列由一系列有序的元素构成，它们从叫做“队尾”的一端进入队列，再从叫做“队首”的另一端被移出队列。队列保持“先进先出”的特性。下面是队列的一些操作：

- ·`Queue()`创建一个空队列对象，无需参数，返回空的队列；
- `enqueue(item)`将数据项添加到队尾，无返回值；
- ·`dequeue()`从队首移除数据项，无需参数，返回值为队首数据项；
- ·`isEmpty()`测试是否为空队列，无需参数，返回值为布尔值；
- `size()`返回队列中的数据项的个数，无需参数。

举个例子，如果我们申明变量 q 是一个已经创建的空队列，表 3.2 列出了对队列进行一系列操作的结果。这个队列中包含的内容也被显示出来，可以看出队列的队首在右边。4 是最早被加进队列的元素，因此它被最早返回。

队列操作	队列内容	返回值
q=Queue()	[]	Queue 对象

q.isEmpty()	[]	True
q.enqueue(4)	[4]	
q.enqueue('dog')	['dog',4]	
q.enqueue(True)	[True,'dog',4]	
q.size()	[True,'dog',4]	3
q.isEmpty()	[True,'dog',4]	False
q.enqueue(8.4)	[8.4,True,'dog',4]	
q.dequeue()	[8.4,True,'dog']	4
q.dequeue()	[8.4,True]	'dog'
q.size()	[8.4,True]	2

表 3.2 队列操作示例

3.4.3.在 PYTHON 中实现 QUEUE

为了实现队列这个抽象数据类型，创建一个新的类是最好的方式。就像之前一样，我们还是利用简单而强大的**列表**（**List**）来帮助建立队列的新类。

我们需要决定列表的哪一端做队尾，哪一端用来做队首。下面的一段代码设定队列的队尾在列表的 0 位置。这使得我们能够利用列表的 insert 功能来向队列的队尾添加新的元素。而 pop 操作则可以用来移除队首的元素（也就是列表的最后一个元素）。这也意味着 enqueue 的复杂度是 O(n)，而 dequeue 的复杂度是 O(1)。

代码 3 展示的对 Queue 类型的操作就是表 1 中我们进行的一系列操作。

```
1   class Queue:
2       def __init__(self):
3           self.items = []
4
5       def isEmpty(self):
6           return self.items == []
7
8       def enqueue(self, item):
9           self.items.insert(0,item)
10
11      def dequeue(self):
12          return self.items.pop()
13
14      def size(self):
15          return len(self.items)
16
17  q=Queue()
18
19  q.enqueue(4)
20  q.enqueue('dog')
21  q.enqueue(True)
22  print(q.size())
```

代码 3

对这个队列的进一步操作将会出现下面的结果：

```
>>> q.size()
3
>>> q.isEmpty()
False
>>> q.enqueue(8.4)
>>> q.dequeue()
4
>>> q.dequeue()
'dog'
>>> q.size()
2
```

牛刀小试

1. 假设你进行了下面一系列的队列操作：

```
q = Queue()
q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
q.dequeue()
```

队列里面还剩下什么元素？

a)‘hello’,‘dog’

b)‘dog’,3

c)‘hello’,3

d)‘hello’,‘dog’,3

3.4.4. 模拟算法：热土豆

提出队列运作方式的一个典型实例也就是模拟出一个需要运用到先进先出（FIFO）数据管理方式的真实情景。首先，让我们来考虑一个叫做热土豆的儿童游戏。在这个游戏中（见图 3.4）小孩子们

围成一个圆圈并以最快的速度接连传递物品，并在游戏的一个特定时刻停止传递，这时手中拿着物品的小孩就离开圆圈，游戏进行至只剩下一个小孩。

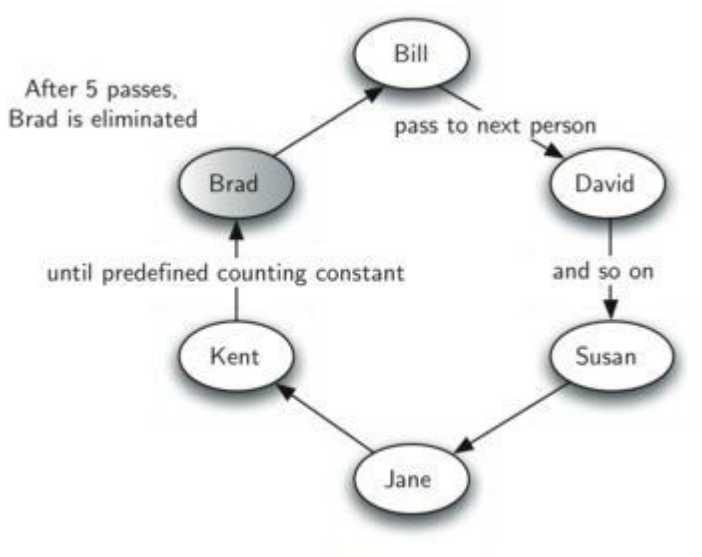


图 3.4 六人传土豆游戏

现在，我们也将热土豆问题称作 Josephus 问题。这个故事是关于公元 1 世纪著名历史学家 Flavius Josephus 的，传说在犹太民族反抗罗马统治的战争中，Josephus 和他的 39 个同胞在一个洞穴中与罗马人相对抗。当注定的失败即将来临之时，他们决定宁可死也不投降罗马。于是他们围成一个圆圈，其中一个人被指定为第一位然后他们按照顺时针进行计数，每数到第七个人就把他杀死。传说中 Josephus 除了熟知历史之外还是一个精通于数学的人。他迅速找出了那个能留到最后的位置。最后一刻，他没有选择自杀而是加入了罗马的阵营。这个故事还有许多不同的版本。有些是以三为单位进行计数，有些则是让最后一个留下的骑马逃走。但不管是哪个版本，其核心原理都是一致的。

我们大体上可以实现对于热土豆问题的**模拟**（**Simulation**）。我们的程序将要读入一个名字列表和用作计数的常数“num”。在经过多次基于 num 的计数后，程序将返回最终剩下的人的姓名。这时那个人会怎么样就由你所决定了。

为了模拟这个环状结构，我们会用到队列（见图 3.5）。假设拿着土豆的孩子位于队首，当开始传递土豆时，模拟器会将那个孩子从队首移出队列然后立即让他从队尾进入队列。所有在他前面的人都轮过一遍后才会再次轮到他传土豆。每经过“num”次出队入队的过程后，站在队首的孩子就会永久离开队列，游戏将在新的圆圈中继续进行。这个过程会一直持续到只有一个名字剩下（即队列规格为 1 时）。

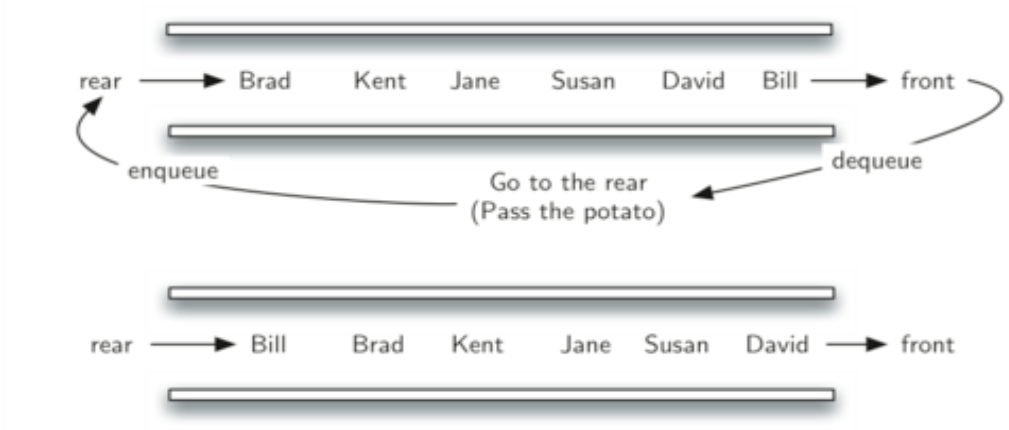


图 3.5 用队列模拟热土豆问题

```
from pythonds.basic.queue import Queue


def hotPotato(namelist, num):
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name)

    while simqueue.size() > 1:
        for i in range(num):
            simqueue.enqueue(simqueue.dequeue())

        simqueue.dequeue()

    return simqueue.dequeue()

print(hotPotato(["Bill","David","Susan","Jane","Kent","Brad"],7))
```

代码 3. 热土豆问题的模拟实现

具体程序展示在代码 3.中。当热土豆函数以 7 为计数常数时被调用会返回 Susan。

可以看到在这个例子中给出的传土豆数要大于列表中人名的个数。但这并不是问题，因为队列像是一个圆圈，当计数到达最后一人的时候就会回到开始，并继续计数直到到达给定值。同时，我们还注意到列表将按照顺序被读入队列，列表中的第一个名字会在队首出现。在这个例子中，Bill 是列表中的第一个元素，所以就被移入了队列的第一个。这个问题的各种拓展情况会在练习中给出，比如说随机数热土豆问题。

一种更为有趣的模拟算法可以用来分析本节前面提过的打印任务。回想一下学生们把打印任务传输给共享打印机，这些任务便被放入一个先进先出的顺序队列。这种结构中蕴含着很多问题。最重要的问题应该是打印机是否能处理确定数量的任务，如果不能，学生们可能会等太长时间以至于错过下一堂课。

考虑计算机科学实验室中的以下情况:平均每天任意一个小时有大约 10 个学生在实验室里,在这一小时中通常每人发起 2 次打印任务,每个打印任务的页数从 1 到 20 页不等。实验室中的打印机比较老旧，如果以草稿模式打印，每分钟可以打印 10 页；打印机可以转换成较高品质的打印模式，但每分钟只能打印 5 页。较慢的打印速度可能会使学生等待太长时间。应该采取哪种打印模式?

我们可以通过建立模型模拟这个实验来做决定。我们需要使用相应的量来代表学生、打印任务和打印机(图 4)。每当学生提交打印任务，即将其加入与打印机相连的等待队列。每当打印机完成一件打印任务，它会检查等待队列里是否还有剩余的打印任务。我们关心的是每个学生等待打印的平均时长，这等于每个任务在队列里的等待平均时间。

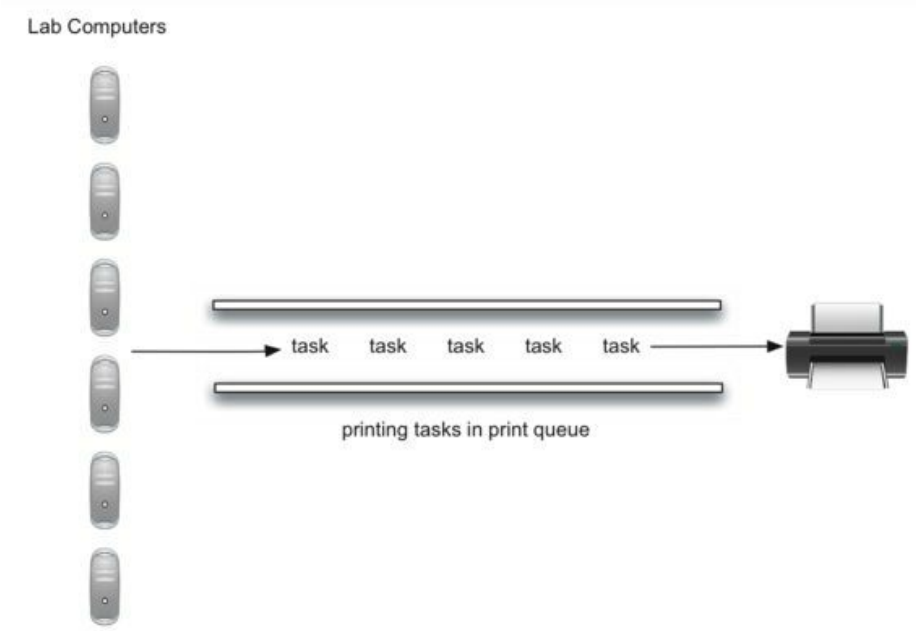


图 3.6 关于打印任务队列的计算机科学实验室

为了模拟出这个情景,我们需要使用“概率”的概念。例如，学生可能会打印 1-20 页长度的文档，如果每个任务的打印页数是在 1 到 20 之间等概率取，那么一次实际打印任务的页数就可以用 1 到 20 之间的随机数来模拟，那意味着 1 到 20 之间任意页数出现的概率相等。

如果实验室里有 10 个学生并且每人打印两次，那么平均每小时就有 20 个打印任务。如此的话，在任意一秒，产生一个打印任务的概率有多大呢?可以考虑任务数与时间的比值。每小时打印 20 个任务即平均每 180 秒生成一个打印任务。

每一秒钟我们可以生成 1-180 之间的随机数来模拟生成一个打印任务的可能性。如果生成的随机数是 180，即认为生成了一个打印任务。注意有时候可能会接连生成很多任务,有时候也可能需要等待很长时间才出现一个任务,而实际情况就是这样。在知道了一些基本参数之后,我们的目的是尽可能精确地模拟真实情况。

$$\frac{20\ tasks}{1\ hour} \times \frac{1\ hour}{60\ minutes} \times \frac{1\ minute}{60\ seconds} = \frac{1\ task}{180\ seconds}$$

3.4.6. 主要模拟步骤

以下为主要的模拟过程:

1、创建一个打印任务队列。每个任务在生成时被赋予一个“时间邮戳”。队列在开始时是空的。

2、对于每一秒(打印过程中的当前秒（currentSecond):

(1)是否有新的打印任务生成?如果有，把它加入打印队列，并把当前秒作为其“时间邮戳”。

(2)如果打印机空闲并且有任务正在等待队列中:

1 从打印队列中移除下一个打印任务并且将其提交给打印机;

2 从当前秒中减去“时间邮戳”值,计算得到该任务的等待时间;

3 将该任务的等待时间添加到一个列表中,以用于后续操作;

4 基于打印任务的页数,求出需要多长的打印时间。

(3)如果此时打印机在工作中，那对于打印机而言，就工作了一秒钟；对于打印任务而言,它离打印结束又近了一秒钟(剩余打印时间减 1)。

(4)如果此时打印任务已经完成,也即是剩余打印时间为 0 时，打印机就进入空闲状态。

3、在整个模拟算法完成后，依据生成的等待时间列表中的数据，计算平均打印时间。

3.4.7 PYTHON 实现

当我们设计这个模拟程序时,我们需要为描述前述的三个真实世界的对象：打印机(Printer)、打印任务(Task)和打印队列(PrintQueue)来自定义一些类。模拟打印机的类 Printer(代码 2)需要实时监测是否正在执行打印任务。如果是，则表示打印机正忙(13—17 行)，需要的等待时间可以由当前任务的打印张数求得。同时初始构造函数还要能完成单位时间打印张数的初始化设置。方法 tick 用于减去内设任务的完成所需时间，并在一次任务结束后将打印机设为闲置(11 行)。

```
class Printer:
    def __init__(self, ppm):
        self.pagerate = ppm
        self.currentTask = None
        self.timeRemaining = 0

    def tick(self):
        if self.currentTask != None:
            self.timeRemaining = self.timeRemaining - 1
            if self.timeRemaining <= 0:
                self.currentTask = None

    def busy(self):
        if self.currentTask != None:
            return True
        else:
            return False

    def startNext(self,newtask):
        self.currentTask = newtask
        self.timeRemaining = newtask.getPages() * 60/self.pagerate
```

代码2

类 Task(代码 3)代表一个单独的打印任务。当一个任务被创建时，随机数生成器产生一个 1 到 20 之间的一个随机数代表需打印的页数。我们选择调用 random 模块中的 randrange 方法。

每个打印任务还需要定义一个用于计算等待时间的对象 timestamp。timestamp 会记录下任务被创建和被放入打印队列时的时间。方法 waitTime 用于检索任务开始打印前在队列中的等待时长。

代码 3

```
import random


class Task:
    def __init__(self,time):
        self.timestamp = time
        self.pages = random.randrange(1,21)

    def getStamp(self):
        return self.timestamp

    def getPages(self):
        return self.pages

    def waitTime(self, currenttime):
        return currenttime - self.timestamp
```

主函数 simulation(代码 4)可以实现上述算法。对象 printQueue 是我们现有的抽象数据类型队列的一个实例。我们借助布尔数学体系的方法 newPrintTask 决定是否生成新的打印任务。同时我们再一次调用 random 模块中的 randrange 函数来得到一个 1 到 180 之间的随机整数(32 行)，据此模拟这个随机事件。模拟函数让我们可以为打印机设置总时长和单位时间打印张数。

```
>>> import random
>>> random.randrange(1,21)
18
>>> random.randrange(1,21)
8
```

```
from pythonds.basic.queue import Queue
import random
def simulation(numSeconds, pagesPerMinute):

    labprinter = Printer(pagesPerMinute)
    printQueue = Queue()
    waitingtimes = []

    for currentSecond in range(numSeconds):
      if newPrintTask():
         task = Task(currentSecond)
         printQueue.enqueue(task)
      if (not labprinter.busy()) and (not printQueue.isEmpty()):
        nexttask = printQueue.dequeue()
        waitingtimes.append(nexttask.waitTime(currentSecond))
        labprinter.startNext(nexttask)

      labprinter.tick()
```

代码4

当我们运行模拟器时，每次的结果将会是不同的，这是由于随机数具有概率属性。我们应该关心的是当我们调整模拟器的参数时，结果会有什么变化趋势。下面是一些结果。

首先我们让模拟器运行一个 60 分钟(3600 秒)每分钟打印五张的情况。而且，我们将要进行五次独立的试验。因为模拟器通过随机数进行测试，所以每次运行会返回不同的结果。

```
>>>for i in range(10):
        simulation(3600,5)


Average Wait 165.38 secs 2 tasks remaining.
Average Wait    95.07 secs 1 tasks remaining.
Average Wait    65.05 secs 2 tasks remaining.
Average Wait    99.74 secs 1 tasks remaining.
Average Wait    17.27 secs 0 tasks remaining.
Average Wait 239.61 secs 5 tasks remaining.
Average Wait    75.11 secs 1 tasks remaining.
Average Wait    48.33 secs 0 tasks remaining.
Average Wait    39.31 secs 3 tasks remaining.
Average Wait 376.05 secs 1 tasks remaining.
```

运行 10 次之后后我们可以看到平均等待时间是 122.155 秒。我们还可以观察到平均等待的时间有很大差异的，从最小平均 17.27 秒到最大平均 239.61 秒。还可以看到在所有情况中只有两次任务被全部完成了。

现在，我们将打印速度调整到每分钟 10 页，然后再运行十次。有了更快的打印速度，我们希望在一小时的时限中能有更多的任务被完成。

运行模拟器的结果如下：

```
>>>for i in range(10):
        simulation(3600,10)

Average Wait      1.29 secs 0 tasks remaining.
Average Wait      7.00 secs 0 tasks remaining.
Average Wait     28.96 secs 1 tasks remaining.
Average Wait     13.55 secs 0 tasks remaining.
Average Wait     12.67 secs 0 tasks remaining.
Average Wait      6.46 secs 0 tasks remaining.
Average Wait     22.33 secs 0 tasks remaining.
Average Wait     12.39 secs 0 tasks remaining.
Average Wait      7.27 secs 0 tasks remaining.
Average Wait     18.17 secs 0 tasks remaining.
```

```
from pythonds.basic.queue import Queue
import random
class Printer:
    def __init__(self, ppm):
        self.pagerate = ppm
        self.currentTask = None
        self.timeRemaining = 0
    def tick(self):
        if self.currentTask != None:
            self.timeRemaining = self.timeRemaining - 1
            if self.timeRemaining <= 0:
                self.currentTask = None
    def busy(self):
        if self.currentTask != None:
            return True
        else:
            return False
    def startNext(self,newtask):
        self.currentTask = newtask
        self.timeRemaining = newtask.getPages() * 60/self.pagerate
class Task:
    def __init__(self,time):
        self.timestamp = time
```

你可以自己运行以下代码中的模拟器。

```
            self.pages = random.randrange(1,21)
        def getStamp(self):
            return self.timestamp

        def getPages(self):
            return self.pages

        def waitTime(self, currenttime):
            return currenttime - self.timestamp



def simulation(numSeconds, pagesPerMinute):

        labprinter = Printer(pagesPerMinute)
        printQueue = Queue()
        waitingtimes = []

        for currentSecond in range(numSeconds):

           if newPrintTask():
              task = Task(currentSecond)
              printQueue.enqueue(task)

           if (not labprinter.busy()) and (not printQueue.isEmpty()):
              nexttask = printQueue.dequeue()
              waitingtimes.append( nexttask.waitTime(currentSecond))
              labprinter.startNext(nexttask)

            labprinter.tick()

        averageWait=sum(waitingtimes)/len(waitingtimes)
        print("Average Wait %6.2f secs %3d tasks remaining."%(averageWait,printQueue.size()))
```

```
def newPrintTask():
    num = random.randrange(1,181)
    if num == 180:
        return True
    else:
        return False

for i in range(10):
    simulation(3600,5)
```

代码 5 打印队列的模拟

3.4.8. 讨论

我们想要回答的问题是，现有的打印机如果被设置成打印品质较高而打印速度较低的模式，是否可以完成打印任务。我们采用的方法是用一个模拟程序来模拟打印任务，它是页数和任务生成时间可变的随机事件。

以上结果表明每分钟打印 5 页时,平均等待时间从最少 17 秒到最多 376 秒(约 6 分钟)不等。当打印速度加快时,最少平均等待时间缩短到 1 秒,最多仅 28 秒。此外，以 5 页每秒的速度打印时，10 次试验中有 8 次在 1 小时结束时仍有任务遗留在队列中（未完成）。

因此,我们可以得出结论:降低打印速度以获得更好的打印品质或许不是一个好方法。学生们不能因为打印而等待太长时间，尤其是在需要赶去上下一节课时，对他们来说 6 分钟的等待时间似乎太长了。

这种模拟分析可以让我们回答许多“如果……会怎样”的问题。我们所要做的只是改变模拟中的各种参数，便可以模拟出任何规模的有趣的行为。例如:

- 如果注册的学生人数增加，平均学生数增加了 20 人会怎样?
- 如果周六时学生不需要去上课会怎样?他们等待得起吗?
- 如果由于 Python 是一门强大的语言,，代码长度大大缩短，平均每个打印任务的页数减少,又会怎样?

这些问题全都可以通过修改以上的模拟程序来解答。然而重要的是知道这些模拟的好坏取决于用于建立模拟的假设,良好的模拟需要真实的每小时任务数量和每小时学生数量等数据。

牛刀小试

1. 如何修改打印模拟程序以反映学生数目增加时的情况?假设学生数目加倍,你需要做一些合理的、有关如何整合这些模拟情况的假设。如何实现呢?修改代码!同样地，假设平均每个打印任务的页数减半，也可以修改代码来反映这种变化。最后，如何把学生数量参数化?比起修改代码我们可以将学生数目变成模拟程序的一个参数。

3.5.双端队列

3.5.1. 什么是双端队列

双端队列（deque 或 double-ended queue）与队列类似，也是一系列元素的有序组合。其两端称为队首（front）和队尾（rear），元素在到达两端之前始终位于双端队列中。与队列不同的是，双端队列对元素添加和删除的限制不那么严格，元素可以从两端插入，也可以从两端删除。可以说，双端队列这种混合的线性数据结构拥有栈和队列各自拥有的所有功能。图 3.7 是一个由 Python 数据对象构成的双端队列。 应当指出，双端队列虽然具备栈和队列的许多特征，但其中的数据项不满足严格的“后进先出”或“先进先出”顺序，这使得插入和删除操作的规律性需要由用户自己维持。

图 3.7 一个由 Python 数据对象构成的双端队列

3.5.2. 抽象数据类型

Deque 抽象数据类型双端队列(Deque)由以下一些结构和操作来定义。如前文所述，双端队列由一系列有序的元素组织而成，元素可以从队首或队尾插入、删除。下面是双端队列的一些操作。

- `Deque()` 创建一个空双端队列，无参数，返回值为 Deque 对象。
- `addFront(item)` 在队首插入一个元素，参数为待插入元素，无返回值。
- `addRear(item)` 在队尾插入一个元素，参数为待插入元素，无返回值
- `removeFront()` 在队首移除一个元素，无参数，返回值为该元素。双端队列会被改变。
- `removeRear()` 在队尾移除一个元素，无参数，返回值为该元素。双端队列会被改变。
- `isEmpty()` 判断双端队列是否为空，无参数，返回布尔值。
- `size()` 返回双端队列中数据项的个数，无参数，返回值为整型数值。

下面举例说明。假定 d 是创建的一个空双端队列，下表给出了对其进行一系列操作后的结果。注意靠近队首的元素在右侧。由于双端队列两端均可插入、删除，在添加和移除数据项时有时容易造成混乱，所以时刻关注队首、队尾的状态是非常重要的。

双端队列操作	双端队列内容	返回值
d=Deque()	[]	Deque 对象
d.isEmpty()	[]	True
d.addRear(4)	[4]	
d.addRear(‘dog’)	[‘dog’,4]	
d.addFront(‘cat’)	[‘dog’,4, ‘cat’]	

d.addFront(True)	[‘dog’,4, ‘cat’,True]	
d.size()	[‘dog’,4, ‘cat’,True]	4
d.isEmpty()	[‘dog’,4, ‘cat’,True]	False
d.addRear(8.4)	[8.4,‘dog’,4, ‘cat’,True]	
d.removeRear()	[‘dog’,4, ‘cat’,True]	8.4
d.removeFront()	[‘dog’,4, ‘cat’]	True

表 3.3 双端队列操作

3.5.3 在 PYTHON 中实现双端队列 DEQUE

就像我们在之前的章节中所做的一样，我们将建立一个新的类来实现抽象数据类型 Deque。再一次地，我们可以通过 Python 中列表提供的一套非常好的方法来实现 Deque 的细节。我们实现的 Deque（代码 1)假设尾队尾在列表的 0 位置。

```
class Deque:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def addFront(self, item):
        self.items.append(item)
    def addRear(self, item):
        self.items.insert(0,item)
    def removeFront(self):
        return self.items.pop()
    def removeRear(self):
        return self.items.pop(0)
    def size(self):
        return len(self.items)
```

代码 3.

在 removeFront 中我们用 pop 方法删除列表中的最后一个元素。而在 removeRear 中我们用 pop(0)来删除列表中的第一个元素。同样地，我们需要在 addRear 中用 insert 方法（第 12 行）因为 append 方法只能在列表的尾端添加新元素。

代码 2 所示的操作就是表 1 中对 Deque 进行的一系列操作。

```
class Deque:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def addFront(self, item):
        self.items.append(item)
    def addRear(self, item):
        self.items.insert(0,item)
    def removeFront(self):
        return self.items.pop()
    def removeRear(self):
        return self.items.pop(0)
    def size(self):
        return len(self.items)
d=Deque()
print(d.isEmpty())
d.addRear(4)
d.addRear('dog')
d.addFront('cat')
d.addFront(True)
print(d.size())
print(d.isEmpty())
d.addRear(8.4)
print(d.removeRear())
print(d.removeFront())
```

代码 3. Deque 运行实例

你可以看出这里与描述队列和栈的代码有许多相似的地方。你也应该可以看到在执行过程中从头头添加或删除项目是O(1)，而从尾部添加或删除是O(n)。

3.5.4 “回文词”判定

一个能用双端队列数据结构轻松解决的问题是经典的“回文词”问题。**回文词**指的是正读和反读都一样的词，如：radar、toot 和 madam。我们想要编写一个算法来检查放入的字符串是否为回文词。

这个问题的解决方案是用一个双端队列来存储这个字符串。我们遍历这个字符串并把它的每个字母添加到双端队列的尾端。现在这个双端队列看起来非常像一个普通队列，但我们可以利用双端队列两端的对称性。双端队列的首端用来存储第一个字符，尾端用来存储最后一个字符。（见图3.8）

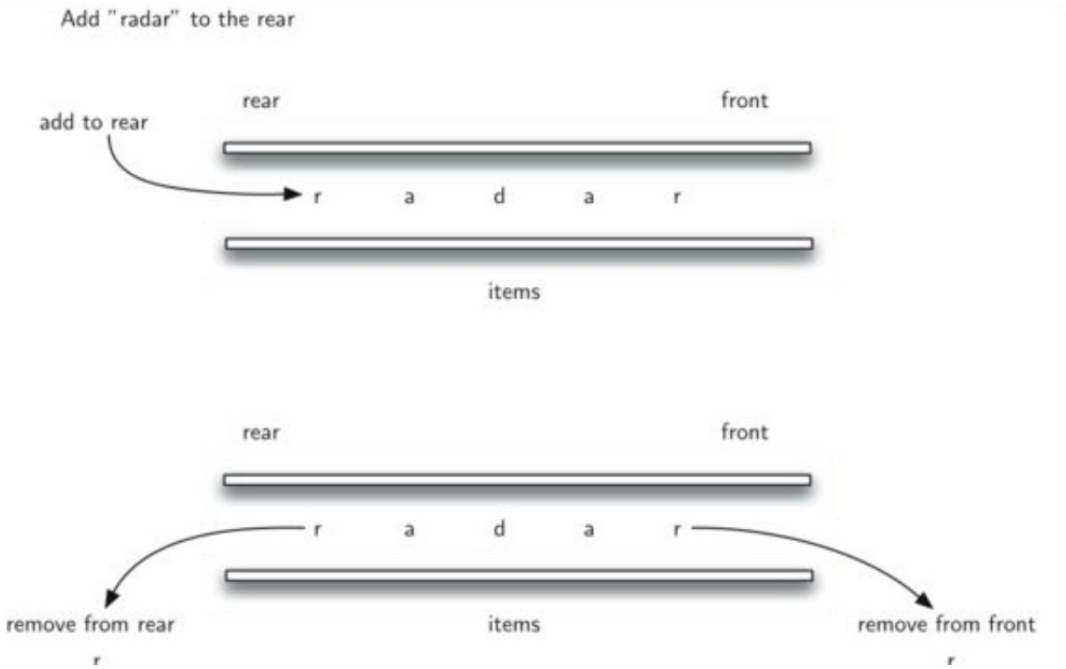


图 3.8 双端队列

因为我们能够同时取出两端的字符，所以我们可以比较它们是否相同，如果相同就继续比较剩下的双端队列的首尾字符。如果我们持续比较首尾字符并发现它们相同，最后字符串要么被比较完，要么只剩下一个字符，这取决于字符串的原始长度是奇数还是偶数。不管哪种情况，这个字符串都是一个回文词。回文词判断函数的实现在函数 palchecker（代码 3）中。

```
from pythonds.basic.deque import Deque
def palchecker(aString):
    chardeque = Deque()
    for ch in aString:
        chardeque.addRear(ch)
    stillEqual = True
    while chardeque.size() > 1 and stillEqual:
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False
    return stillEqual
print(palchecker("lsdkjfskf"))
```

```
print(palchecker("radar"))
```

代码 3 通过双端队列实现的回文词判断函数(palchecker)

3.6 列表 LIST

在整个基本数据结构的讨论中，我们使用了 Python 列表来实现抽象数据类型。列表是一个功能强大而简单的收集容器，并为程序员提供了各种各样的操作。然而， 并非所有的编程语言都有内置的 list 列表类型。在这些情况下，程序员必须自己来实现列表。

列表是一些元素的集合，每个元素拥有一个与其它元素不同的相对位置。更具体地说，我们把这种类型的列表称为一个无序列表。我们可以认为列表有第一项、第二项、第三项……也可以索引到列表的开始（第一项）或列表的最后（最后一项）。为简单起见，我们假设列表不能包含重复项。

例如，由整数 54，26，93，17，77，31 组成的集合可能是一个简单的无序列表的考试成绩。注意，我们已经把它们写入了用逗号分隔的值的列表（一种表示结构的常用方法）。当然，Python 会把此列表显示为**[54,26,93,17,77,31]**。

3.6.1. 抽象数据类型无序列表 UNORDEREDLIST

如上面所描述的，无序列表结构是一个由各个元素组成的集合，在其中的每个元素拥有一个不同于其它元素的相对位置。一些可用的无序列表的方法如下。

* list() 创建一个新的空列表。它不需要参数，而返回一个空列表。

*add(item) 将新项添加到列表，没有返回值。假设元素不在列表中。

*remove(item) 从列表中删除元素。需要一个参数，并会修改列表。此处假设元素在列表中。

*search(item) 搜索列表中的元素。需要一个参数，并返回一个布尔值。

*isEmpty() 判断列表是否为空。不需要参数，并返回一个布尔值。

*size() 返回列表的元素数。不需要参数，并返回一个整数。

*append(item) 在列表末端添加一个新的元素。它需要一个参数，没有返回值。假设该项目不在列表中。

*index(item) 返回元素在列表中的位置。它需要一个参数，并返回位置索引值。此处假设该元素原本在列表中。

*insert(pos,item) 在指定的位置添加一个新元素。它需要两个参数，没有返回值。假设该元素在列表中并不存在，并且列表有足够的长度满足参数提供的索引需要。

*pop() 从列表末端移除一个元素并返回它。它不需要参数，返回一个元素。假设列表至少有一个元素。

*pop(pos) 从指定的位置移除列表元素并返回它。它需要一个位置参数，并返回一个元素。假设该元素在列表中。

3.6.2.采用链表实现无序列表

为了实现无序列表，我们将构建一个**链表**。回想一下，我们需要确保元素的相对位置正确。然而，我们无需使用连续的内存来定位链表中的元素。例如，考虑图 3.9 中显示的项的集合。看起来这些值已被随机放置。如果我们可以在每个项目保持一些明确的信息，即下一个项目的位置（见图 3.10），那么每个项目的相对位置就可以通过以下简单的链接从一个项目到下一个来确定。

图 3.9 不受其物理位置约束的项目

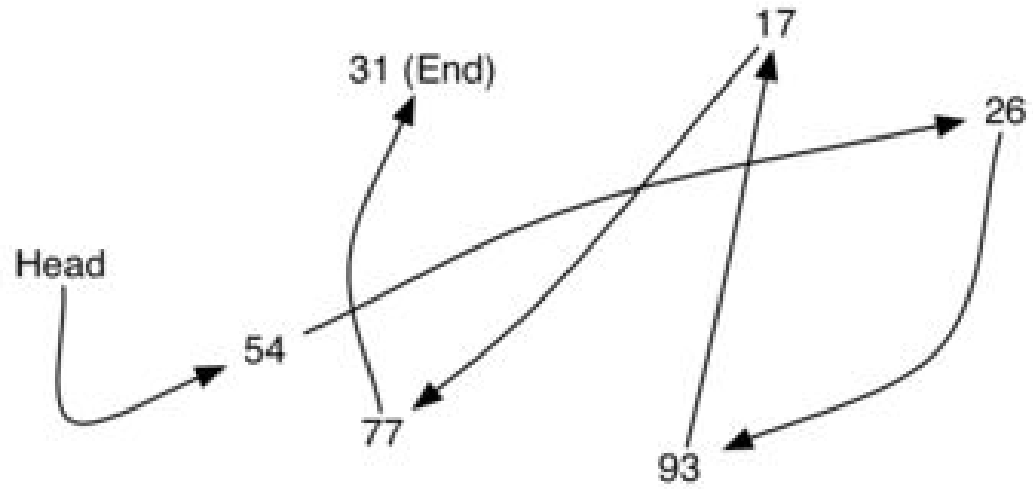


图 3.10 使用显示链接确定的相对位置

需要注意的是，该列表的第一项的位置必须被明确指出。一旦我们知道第一项是什么，第一项就可以告诉我们第二项是什么，以此类推。从外部指向的第一项通常被称为链表的**头**。同样地，链表的最后一项需要告诉我们有没有下一个项目。

3.6.2.1. 类：节点 NODE

用链表实现的基本模块是**节点**。每个节点对象必须持有至少两条信息。首先，节点必须包含列表元素本身。我们将这称为该节点的“**数据区**”（data field）。此外，每个节点必须保持到下一个节点的引用。代码 1 显示了 Python 的实现方法。如构造一个节点”93”（见图 3.11）。需要指出，我们将通常以如图 3.12 所示的方式代表一个节点对象。节点类还包括访问和修改的常用方法：返回节点数据和引用到下一项。

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None
    def getData(self):
        return self.data
    def getNext(self):
        return self.next
    def setData(self,newdata):
        self.data = newdata
    def setNext(self,newnext):
```

```
        self.next = newnext
```

代码 3.

我们以常见的方式创建了节点类。

```
>>> temp = Node(93)
>>> temp.getData()
93
```

代码 3.

Python 的特殊值 None 将在节点类和之后的链表类中发挥重要的作用。引用 None 意味着没有下一个节点。在构造器中，一个节点的对下一节点引用的初始赋值是 None。因为这有时被称为把节点“接地”（grounding），我们在图中将使用标准化的“接地”标志来表示一个值为 None 的引用。用 None 来作为你在初始化时对下一个节点的引用是一个极妙的主意。

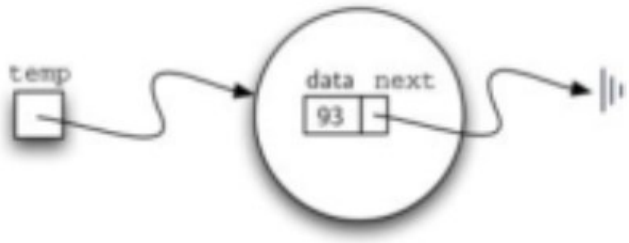


图 3.11 节点类对象包含本身的项目数据和对下一个节点的引用

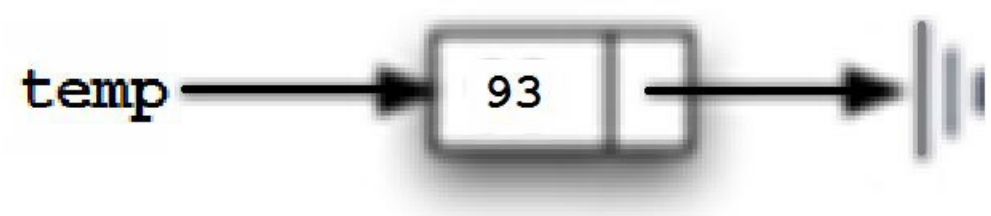


图 3.12 一个典型的节点类对象的实现

3.6.2.2. 类：无序列表 UNORDERED LIST

正如我们之前提到的，无序列表将由一个节点集合组成，每一个节点采用显式引用链接到下一个节点。只要我们知道第一个节点的位置（包含第一项），在这之后的每个元素都可以通过以下链接找到下一个节点。为实现这个想法，UnorderedList 类必须保持一个对第一节点的引用。代码 2 显示了这种构造结构。注意每个 UnorderedList 对象将保持列表的头一个节点的引用。

```
class UnorderedList:
    def __init__(self):
        self.head = None
```

代码 3.

```
>>> mylist = UnorderedList()
```

在最初当我们建立一个列表时，其中没有任何元素。如图 3.13 所示，这些赋值语句建立了一个连接好的链表。正如我们在定义节点类时讨论到的，特殊引用——None 会再被用来说明列表的头不指向任何东西。最终，如图 3.14 所示，之前给出的示例列表将由一个链表来表示。列表的头指向包含列表的第一项的第一节点。以此类推，该节点有一个指针指向到下一个节点（的内容）。需要注意的一点是，列表类本身不包含任何节点。取而代之的是，它包含对链式存储结构的第一个节点的引用。

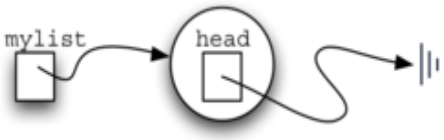


图 3.13 一个空列表

图 3.14 一个由整数组成的链表

is_empty（判断链表是否为空）方法仅仅只是检查了列表的头是否指向 None。其结果以布尔值表示，如果链表中没有节点，self.head==None 的结果只能是 True。因为新链表为空，所以构造器和是否为空的检验必须与另一个保持一致。这个显示了使用指向 None 来指示链表结构的结束。在 Python 中，None 可以与任何指向相提并论。如果两个指向同一个物体，那么他们是平等的。我们在后续的方法中经常用到这一点。

```
def isEmpty(self):
    return self.head == None
```

代码 3.

那么，我们如何把元素放入链表？我们需要实现添加（add）的方法。在此之前，我们需要解决的重要问题是我们应该将新的元素放在链表的哪一部分。因为这个列表是无序的，新元素在列表中相对于其他元素的具体位置是不重要的。新的元素可以放置在任何地方。鉴于此，把新的元素放置在最简单的位置最好。

考虑到链表结构只为我们提供了一个入口，即该列表的头，所有其他节点只能通过访问第一个节点，然后通过下一个引用链接到那里。这意味着最容易增加新节点的地方是在头部，或者说在列表的开始。换句话说，我们将把新项目作为列表的第一个项目，而现有的项目将需要连接到这个新第一个项目，这样它们就会跟随过来。

在图 6 中展示的列表是通过多次添加数字这个方法来建构的。

```
>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)
```

请注意，由于 31 是添加到列表中的第一项，那么它最终将在链表的最后一个节点，因为之后的每一项被添加在它前面。同时，由于 54 是最后加入的元素，它将成为链表中的第一个节点的数据值。

add 方法如代码 4 所示。列表中的每个元素必定属于一个节点。第二行创建了一个新的节点并将插入的元素作为节点的数据。现在我们必须通过链接这个新的节点与原有的结构来完成插入元素的工作。这需要图 3.15 所示的两个步骤。第一个步骤（代码 4 第 3 行）是把新插入节点的引用设为原来列表的头节点。由于列表中的其他部分已经和这个新节点正确地连接了，我们可以把列表头部 head 指向这个新的节点。代码第 4 行就是这一步骤，它确定了列表的头部。

上述两个步骤的顺序是十分重要的。如果将第 3 行和第 4 行的顺序颠倒过来会如何呢？如果我们先更改这个列表的头部，那么结果将如图 3.16 所示。由于头部是外部对链表内部节点的唯一引用，所有原有的节点将会丢失并且无法被再度访问。

```
def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp
```

代码 3.

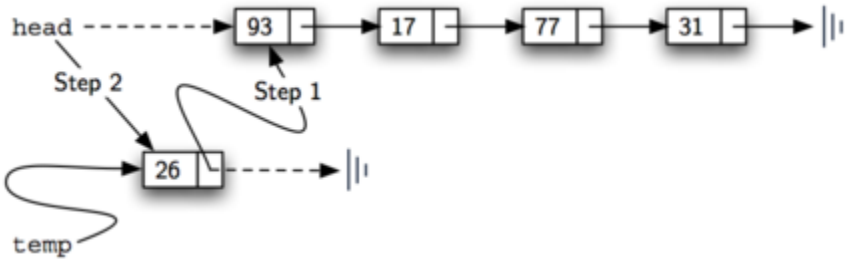


图 3.15：向一个两步的进程中添加一个新节点

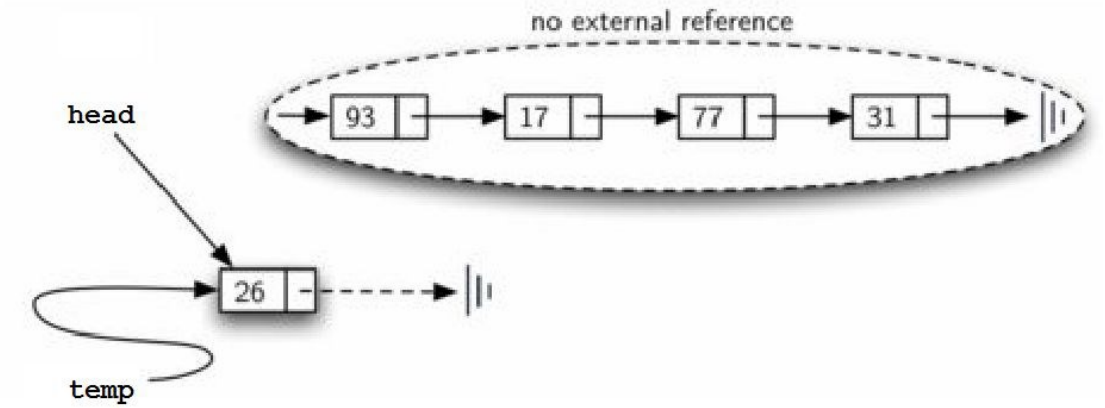


图 3.16：改变两个步骤顺序的后果

接下来我们所要实现的方法——求元素个数（size）、查找（search）和移除（remove），全部是基于一个叫做**链表的遍历**（traversal）的操作的。遍历指的是有序地访问每一个节点的过程。为了做到这一点，我们可以使用一个外部引用，它最开始指向列表的第一个节点。每当我们访问一个节点时，我们通过“侧向移动”（traversing）到下一个引用的方式，将外部引用移动到下一个节点。

为了实现“size 求元素个数”的方法，我们需要遍历链表，并且记录出现过的节点个数。代码 5 是计算列表中节点个数的 Python 代码。我们把这个外部引用称为“当前”（current），在第二行中它被初始化，指向列表的头部。最初我们并没有发现任何节点，所以计数的初值被设定为 0。第四到第六行实际上实现了这次遍历。只要这个外部引用没有遇到列表的尾端（None），我们就将 current 移动到下一个节点，正如第 6 行所示。和前文相同，把引用和 None 进行比较的操作非常有用。每当 current 移动到了一个新的节点，我们就把计数器加 1(count)。最终，我们在循环结束后返回了计数值。图 3.17 展示了这样一个进程。

```
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.getNext()
    return count
```

代码 3.

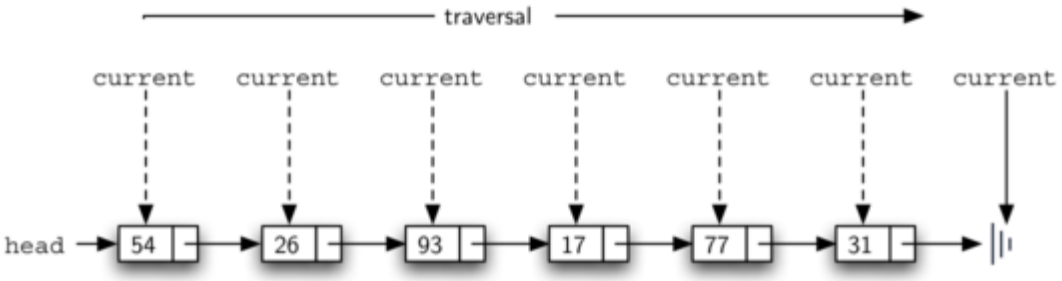


图 3.17 将列表从头遍历到尾

在一个无序表中查询一个数值这一方法的实现同样需要用到遍历。每当我们访问一个链表中的节点时，我们会判断储存在此的数据是否就是我们所要找的元素。然而在这个例子中，我们并没有必要遍历整个列表。事实上，如果我们的工作进行到了列表的底端，这意味着我们所要寻找的元素在列表中并不存在。同样，如果我们找到了那个元素，那么就没有必要继续寻找了。

代码 6 实现了查询(search)这一方法。同 size 方法一样，遍历在列表的头部被初始化（第 2 行）。我们同样使用一个叫做 found 的布尔变量来表示我们是否找到了我们所要找寻的元素。考虑到我们在遍历开始时并没有找到那个元素，found 被设为假(False)（第 3 行）。第 4 行中的循环同时考虑了上述的两种情况。只要还有余下的未访问节点并且我们还没有找到元素，我们便继续检查下一个节点。第 5 行中的条件语句判断所寻的数据项是否在节点 current 之中。如果是，那么 found 被设为真(True)。

```
def search(self,item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found = True
        else:
            current = current.getNext()
    return found
```

代码 3.

以调用查询(search)过程来查找“17”为例。

```
>>> mylist.search(17)
True
```

由于 17 在列表中，遍历进程只需要进行到那个含有 17 的节点。此时变量 found 将会被设为真(True)，并在条件失效时返回 True，如上图所示。这个过程如图 3.18。

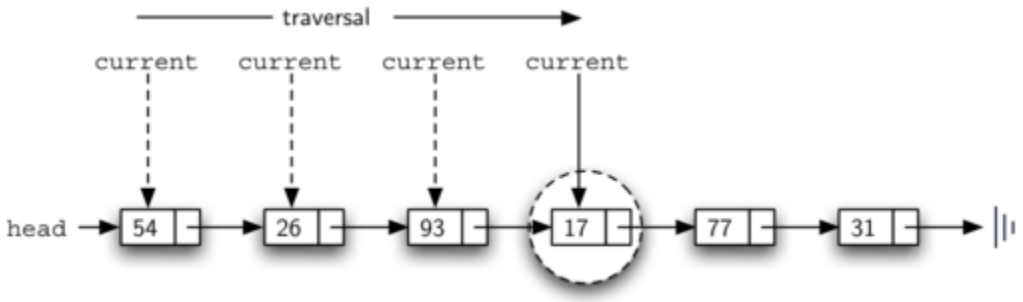


图3.18：成功找到数值“17”

“移除(remove)”这个方法需要两个步骤。首先我们需要遍历这个列表，来找寻我们想要移除的元素。只要找到了这个元素（假设它存在），就必须移除它。第一步同查询(search)十分接近。我们使用一个外部引用，让它开始时指向链表的头部，顺着链表遍历，直到找到要移除的元素为止。由于我们假设待移除的元素一定存在，那么循环将会在遍历到列表底部前终止。所以，我们这时只需要再使用一个布尔变量 found。

当 found 为真(True)时，current 将会是对包含了要移除元素的一个引用。但我们要如何移除它？一个可行的方案是，用一些标记来替代要移除的元素，从而表明原先的元素已经不在列表中了。这个方案的问题是，节点的数目将与数据项的数目不相同。通过移除整个节点来移除元素会是更可行的方式。

为了移除含有待移除元素的节点，我们需要修改前一个节点的链接方式，使它引用 current 紧跟着的节点。不幸的是，在链表中没有办法从后往前移动。因为 current 引用的节点处于一个我们需要作修改的节点的后方，当我们移除 current 所指节点后，已经无法对前一个节点进行必要的修改操作。

解决这个难题的方法是，在遍历链表时使用两个外部引用。current 不变，仍然标记当前遍历到的位置。新加入的引用——我们叫“前一个”（previous）——在遍历过程中总是落后于 current 一个节点。这样，当 current 停在待删除节点时，previous 即停在链表中需要修改的节点处。

代码 7 展示了完整的移除(remove)过程。第二第三行给两个外部引用赋了初始值。注意到 current 如同其他的“遍历”实例一样，从列表的头部开始。然而，我们设定 previous 总是访问 current 的前一个节点。因此，previous 的初值设为 None，因为头部之前没有节点（如图 3.19）。布尔变量 found 将会再次被用于控制这次循环。

在第六到第七行我们区分了储存在节点中的单元是否是我们想要移除的元素。如果是，found 将会变成真(True)。如果我们没有找到元素，previous 和 current 必须同时向后移动一个节点。同样，这两个操作的顺序至关重要，首先 previous 要向后移动一个节点，到 current 的位置，然后 current 才能移动。这一过程常常被称为“一寸寸蠕动”（inch-worming），因为 previous 先要跟上 current，current 才能向前移动。图 3.20 展示了 previous 和 current 在顺着列表而下寻找含有 17 的节点时的移动方式。

```
def remove(self,item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
```

```
                found = True
            else:
                previous = current
                current = current.getNext()

        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
```

代码 3.

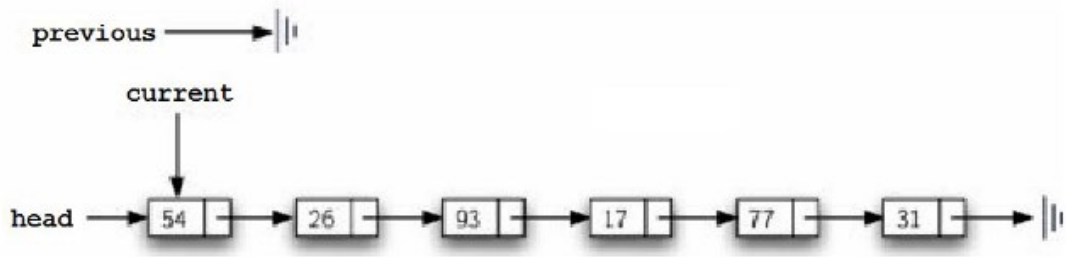


图3.19：两次引用previous和current的初值

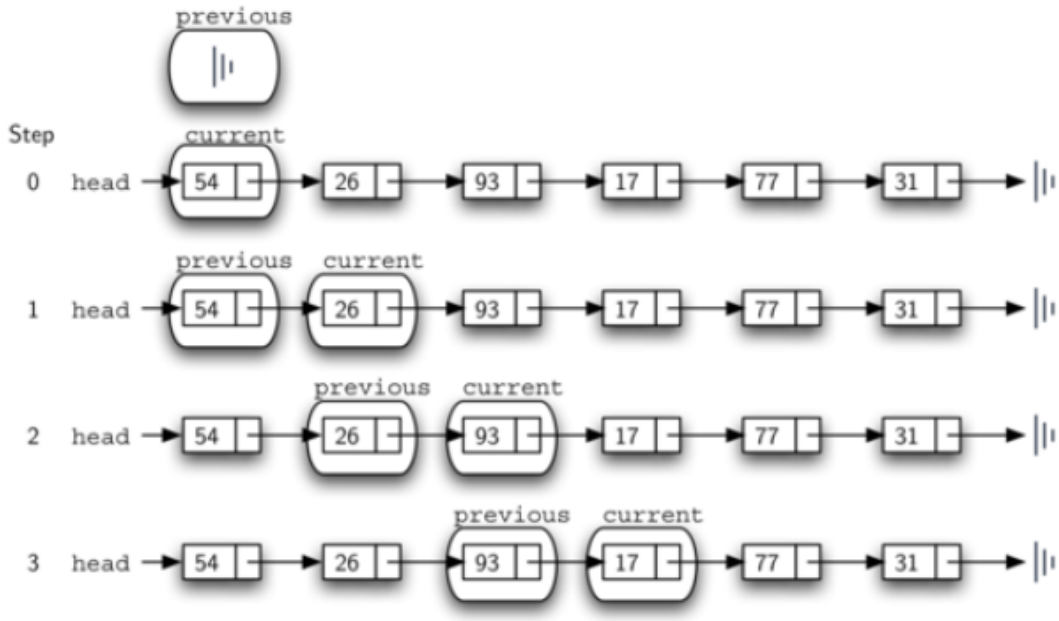


图3.20：previous和current沿列表移动

移除(remove)工作中的查询步骤一旦完成，我们需要从链表中移除那个节点。图 3.21 显示了一个必须进行改动的连接。然而这里又有一点需要特别说明。如果要移除的那个元素恰好是列表中的第一个，那么 current 会引用链表第一个节点。这也就意味着 previous 的引用会是 None。我们之前提

到，previous 要指向那个引用要发生变化的节点。在这种情况下，需要改动的不是 previous，而是链表的头节点（如图 3.22）。

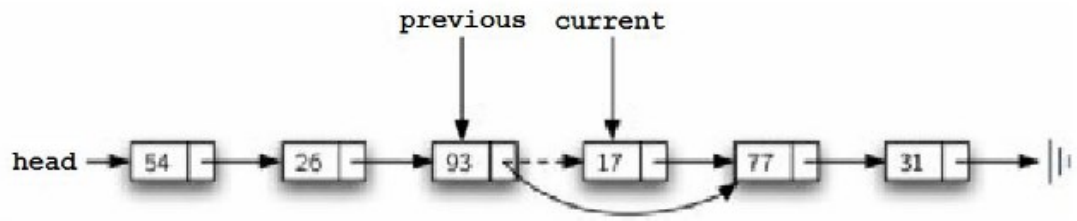


图3.21：移除列表中间的一个节点

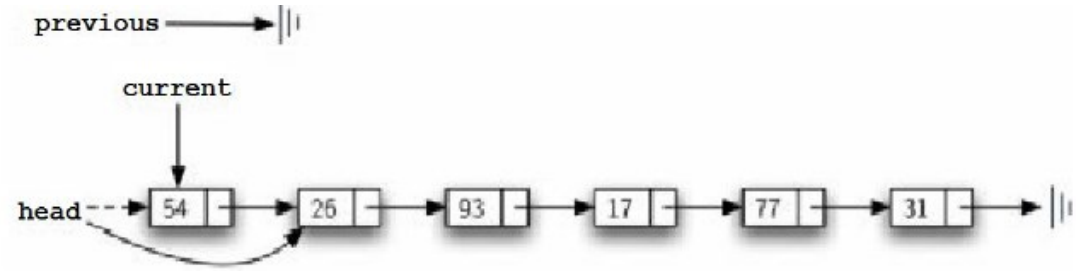


图3.22：移除列表第一个节点

代码 7 的第 12 行让我们可以检查我们所要处理的情况是否是上述的特殊情况。如果 previous 没有移动，那么当布尔变量 found 已经为真时，previous 仍然是 None。这种情况下（第 13 行），链表头部 head 要发生变化，引用紧跟 current 的那个节点，实际效果就等于从列表中移除第一个节点。而当 previous 不是 None 时，要移除的节点一定在链表中表头后方的某处。这时 previous 将会让我们找到所要移除的节点的前一个节点。第 15 行调用了 previous 的 setNext 方法来完成这次移除。注意到在两种情况中，需要改动的节点或表头最终都指向了 current 的后一个节点。读到这里，人们常常产生的疑问是，上述的两种情况是否同样适用于被移除的节点是最后一个的情形。这个问题读者可以自己思考。

你可以用代码 1 中的无序列表来进行测试。

剩下的方法——追加（append），插入（insert），索引（index），弹出（pop）,都将作为你的练习。记住任何一个操作中都要同时考虑对象在列表头部和其他位置这两种情况。同样，插入、索引、弹出需要我们给列表中的位置命名。我们假定列表中位置的名称是从 0 开始的整数。

牛刀小试

1.实现无序列表的 append 方法。并给出你的程序的时间复杂度。

2.在上一个问题中，你很有可能给出一个时间复杂度为 O（n）的算法。但如果你给“无序表”类添加一个变量，你可以写出时间复杂度为 O（1）的算法。将你的算法的时间复杂度简化为 O（1）。注意！这时你需要考虑非常多的特殊情况，同时要修改 add 方法。

3.6.3 抽象数据类型：有序列表

现在，我们考虑一类被称为有序列表的列表。例如，如果前文所示的整数列表是一个有序列表(升序)，那么它可以写成 17，26，31，54，77 和 93。因为 17 是最小的，所以它是列表中的第一个元素。同样，因为 93 是最大的元素，它在列表中的位置是最后的。

有序列表的结构是一个数据的集合体，在集合体中，每个元素相对其他元素有一个基于元素的某些基本性质的位置。假设我们已经在列表元素中定义了一个有意义的比较大小的操作，则排序通常是升序或降序。有序列表的许多方法和无序表是一样的。

OrderedList()：创建一个新的空有序列表。它返回一个空有序列表并且不需要传递任何参数。

add(item):在保持原有顺序的情况下向列表中添加一个新的元素，新的元素作为参数传递进函数而函数无返回值。假设列表中原先并不存在这个元素。

remove(item):从列表中删除某个元素。欲删除的元素作为参数，并且会修改原列表。假设原列表中存在欲删除的元素。

search(item)：在列表中搜索某个元素，被搜索元素作为参数，返回一个布尔值。

isEmpty()：测试列表是否为空，不需要输入参数并且其返回一个布尔值。

size()：返回列表中元素的数量。不需要参数，返回一个整数。

index(item)：返回元素在列表中的位置。需要被搜索的元素作为参数输入，返回此元素的索引值。假设这个元素在列表中。

pop()：删除并返回列表中的最后一项。不需要参数，返回删除的元素。假设列表中至少有一个元素。

pop(pos)：删除并返回索引 pos 指定项。需要被删除元素的索引值作为参数，并且返回这个元素。假设该元素在列表中。

3.6.4. 实现有序列表

为了实现有序列表，我们必须记住，元素的相对位置取决于某种已经定义了的属性。前文给出的整数有序列表(54，17，26，31，77，93)可以表示为一个如图 3.23 所示的链结构。再次看到，节点和链表结构是示意元素相对位置的理想工具。

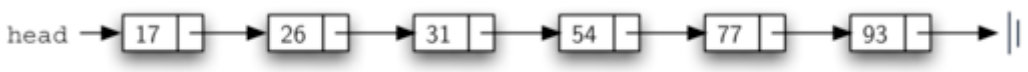


图 3.23 有序的链表

为了实现 OrderedList 类,我们将使用在无序列表中使用过的方法。再一次，一个指向 None 的头指针将表示一个空的列表。（见代码 8）

```
class OrderedList:
    def __init__(self):
    self.head = None
```

代码 8

当我们考虑有序列表的方法时,我们应该注意到,isEmpty 和 size 方法的实现和无序列表相同，因为它们只处理列表中节点的数量而不考虑节点实际的值。同样，remove 方法也不需要改动,因为我们仍然需要找到某个节点然后删除它。但剩下的两个方法：search 和 add 需要一些修改。

为了在一个无序列表中搜索某个节点，我们需要遍历节点直到找到它或者遍历完整个列表（None）。事实证明，当要搜索的元素在列表中时，可以在有序列表中使用与无序列表中相同的方法，但当我们要找的元素不在列表中时，我们可以利用有序列表的顺序来尽快结束搜索。

例如,图 3.24 显示了在有序链表中搜索值 45 的过程：我们从列表的头开始遍历，首先我们和 17 进行比较，因为 17 不是我们要找的项，我们移动到下一个节点。此时，26 依然不是我们要找的，所以我们接着移动到 31，再到 54。现在，有序列表将和无序列表有所不同。由于 54 不是我们要找的项，在之前我们会接着移向下一个节点，但现在，由于列表是有序的，我们不必这么做，因为这是没有必要的。一旦当前节点的值大于我们要找的值，我们就可以停止搜索并返回 False，因为这个值不可能在这个链表中了。

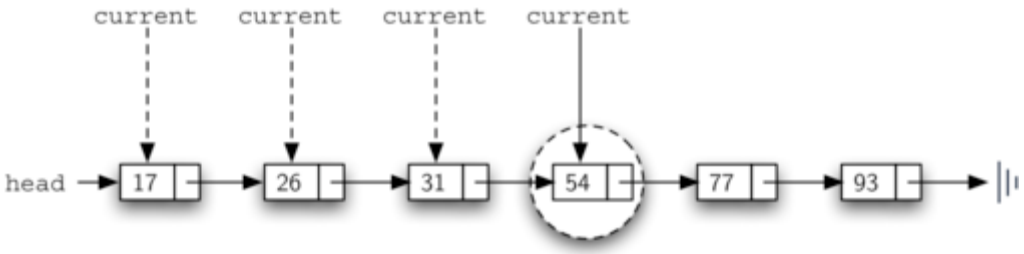


图 3.24

代码 9 显示了完整的搜索方法。通过添加另一个名为 stop，初始化为 False 的布尔变量（第 4 行），我们可以很容易的应用于新的情况。当 stop 的值为 False（不停止）时，我们继续在链表中搜索（第 5 行）；如果发现任何节点包含的数据大于我们正在需找的值，我们将 stop 设 True（9-10 行）。剩余的操作和无序表一完全相同。

代码 9

```
def search(self, item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.get_data() == item:
            found = True
        else:
            if current.get_data() > item:
                stop = True
            else:
                current = current.get_next()
    return found
```

改动最大的方法是 add。回想一下在无序列表中的 add 方法，只需要在原列表头加一个新的节点。这是最简单的方法。不幸的是，这在有序列表中是行不通的。在有序列表中，我们必须将新的节点添加到原列表某个特定的位置。

假设我们的有序列表中已有：17，26，54，77 和 93，我们想添加值 31，则 add 方法必须在 26 和 54 之间添加这个新的节点。图 17 显示了具体做法。正如前面解释的，我们需要遍历链表来寻找添加新节点的位置。遍历时，当我们遍历完了整个列或者当前节点的值大于我们要添加的值时，我们就找到了添加新节点的位置。在我们的例子中，找到了值 54 就停止遍历。

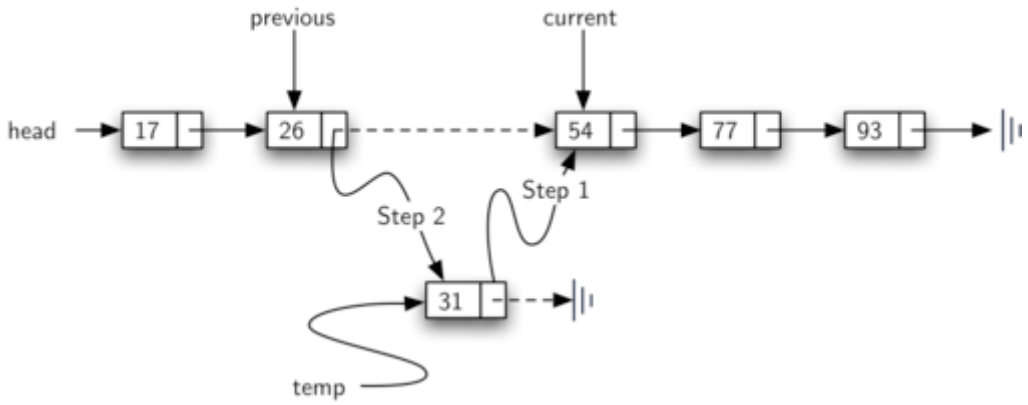


图 3.25 向有序链表添加元素

```
def add(self, item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.get_data() > item:
            stop = True
        else:
            previous = current
            current = current.get_next()
    temp = Node(item)
```

```
    if previous == None:
        temp.set_next(self.head)
        self.head = temp
    else:
        temp.set_next(current)
        previous.set_next(temp)
```

代码 10

正如我们在使用无序列表时，需要有一个额外的指针，即 previous，因为当前节点不提供对需要修改的节点（前一个节点）的访问方法。代码 10 显示了完整的 add 方法。行 2 -3 设置指针 current 和 previous。当 current 依次遍历整个列表时，行 9-10 每次都让 previous 指针指向 current 当前的节点。行 5 的条件只允许 current 移向值比要添加的值大的节点，或者遍历完整个列表再添加。在这两种情况中，我们找到了添加新节点的位置，循环结束。该方法其余的步骤完成了最后的两步过程，如图 3.25 所示。当一个新的节点被创建后，唯一剩下的问题是这个节点是应该添加在链表开头还是添加在链表中间。再一次，判断 previous 是否为 None（行 13）可以提供答案。

OrderedList 类和迄今为止已经讨论了的方法可以在 ActiveCode 1 中找到。其余方法留作练习。当使用有序列表的时候，你应该仔细考虑无序的实现能否使用的问题。

3.6.5. 链表实现算法分析

链表的分析：

当分析链表方法的复杂度时，我们应该考虑它们是否需要遍历链表。考虑一个有 n 个节点的链表，isEmpty 方法复杂度是 O（1），因为它只需要检查链表的头指针是否为 None。对于方法 size，则总需要 n 个步骤，因为除了遍历整个链表以外，没有办法知道链表的节点数。因此，size 方法的复杂度是 O（n）。无序列表的 add 方法的复杂度是 O（1），因为我们永远只需要在链表的头部简单地添加一个新的节点。但是，search、remove 和在有序列表中的 add 方法，需要遍历。尽管在平均情况下，它们可能只需要遍历一半的节点，但这些方法的复杂度都是 O(n)，因为在最糟糕的情况下需要遍历整个链表。

你可能还注意到，这些方法的实现性能与 Python 的内置列表 list 不同，这表明 Python 中的 list 不是这么实现的。实际上，Python 中的列表的实现是基于数组的，我们将在另一章详细讨论。

3.7.小结

线性数据结构以有序的方式维持它们的数据。

栈（Stack）是具有后进先出（LIFO）特性的有序的简单数据结构。

栈（Stack）的基本操作是 push，pop 和 isEmpty。

队列（Queue）是具有先进先出（FIFO）特性的有序的简单数据结构。

队列（Queue）的基本操作是 enqueue,dequeue 和 isEmpty。

前缀表达式，中缀表达式和后缀表达式都是书写表达式的方式。

栈（Stacks）对于设计算法并求值以及转化表达式非常有效。

栈（Stacks）具有反转的特性。

队列（Queue）可以帮助构建时序仿真。

模拟实验使用随机数生成器来创建一个真实的环境从而使我们回答“假设”类型的问题。

双端队列（Deque）是允许像栈和队列一样的混合行为的数据结构。

双端队列（Deque）的基本操作是 addFront, addRear, removeFront, removeRear 和 isEmpty。

列表（List）是具有相对位置的元素的集合。

链表的实现保持逻辑顺序而不要求数据项依次存放在连续的存储空间。

对链表表头的修改是一种特殊情况。

3.8.关键词（按：依英文原词的词典顺序排列）

匹配括号	**数据区**	**双端队列**
先进先出（FIFO）	**全括号**	**表头**
中缀	**后进先出（LIFO）**	**线性数据结构**
链表	**链表遍历**	**列表**
节点	**回文序列**	**后缀**
优先级	**前缀**	**队列**
模拟实验	**栈**	

表 3.4

3.9.问题讨论

1．将下列值通过“除以二”转化为二进制。写出余数的栈。

a) 17

b) 45

c) 96

2．将下列中缀表达式转化为前缀表达式（使用全括号的方法）：

a) (A+B)*(C+D)*(E+F)

b) A+((B+C)*(D+E))

c) A*B*C*D+E+F

3．将上述的中缀表达式转化为后缀表达式（使用全括号的方法）。

4．采用直接的转化算法将上述的中缀表达式转化为后缀表达式。写出转化时栈的实时变化。

5．计算下列后缀表达式的值。写出当每个操作数和操作符被处理时栈的实时变化。

a) 2 3 * 4 +

b) 1 2 + 3 + 4 + 5 +

c) 1 2 3 4 5 * + * +

6．队列（Queue）的一种替换实现是使用一个列表使队列的尾在列表的末端。这样的替换操作会对其大 O 数量级产生什么样的影响？

7．在链表中，使用 add 方法时执行顺序相反的结果是什么？参考结果是什么？可能会什么样的问题？

8．解释当数据项在最后一个节点时链表的 remove 方法如何实现。

9．解释当数据项是链表中唯一一个节点时链表的 remove 功能如何实现。

3.10.编程练习

1．修改中缀表达式转为后缀表达式的算法使之能处理错误输入。

2．修改后缀表达式求值的算法使之能处理错误输入。

3．实现一个结合了中缀到后缀的转化法和后缀的求值算法的直接求中缀表达式值的方法。你的求值法应该从左至右处理中缀表达式中的符号，并且使用两个栈来完成求值，一个存储操作数，一个存储操作符。

4．将上一题的中缀求值法转化为一个计算器。

5．实现一个 Queue，使用一个 list 使 Queue 的尾部在 list 的末端。

6．设计并实现一个实验，对以上两种 Queue 进行基准比较。你从这个实验中学到了什么？

7．实现一个队列并使它的 enqueue 和 dequeue 方法平均时间复杂度都是 O(1)。也就是说，在大多数情况下 enqueue 和 dequeue 都是 O（1），除了在一种特殊情况下 dequeue 可能为 O（n）。

8. 考虑一个现实生活中的情况。制定一个问题，然后设计一个可以帮助解决问题的模拟实验。可能的情况包括：

a）洗车店一字排开的汽车

b）在杂货店结账的顾客

C）在跑道起飞、降落的飞机

d）一个银行柜员

一定要解释清楚做的任何假设，并且提供该方案必须包含的和概率有关的数据。

9．修改热土豆模拟实验，采用一个随机选择的数值，使每轮实验不能通过前一

次实验来预测。

10．实现基数排序。十进制的基数排序是一个使用了“箱的集合”（包括一个主箱和 10 个数字箱）的机械分选技术。每个箱像队列（Queue）一样，根据数据项的到达顺序排好并保持它们的值。算法开始时，将每一个待排序数值放入主箱中。然后对每一个数值进行逐位的分析。每个从主箱最前端取出的数值，将根据其相应位上的数字放在对应的数字箱中。比如，考虑个位数字，534 被放置在数字箱 4，667 被放置在数字箱 7。一旦所有的数值都被放置在相应的数字箱中，所有数值都按照从箱 0 到箱 9 的顺序，依次被取出，重新排入主箱中。该过程继续考虑十位数字，百位数字，等等。当最后一位被处理完后，主箱中就包含了排好序的数值。

11．括号匹配问题的另一个例子是超文本标记语言（HTML）。在 HTML 中，标记以开始（opening tag，<tag>）和结束（closing tag，</tag>）的形式存在，它们必须成对出现来正确地描述 web 文档。这个非常简单的 HTML 文档：

只是为了表明语言中标记的匹配和嵌套结构。写一个程序，它可以检查 HTML 文档中是否有匹配的开始和结束标记。

```
<html>
        <head>
                <title>
                        Example
                </title>
        </head>
        <body>
                <h1>Hello, world</h1>
        </body>
</html>
```

. ☒

12. 扩展 Listing 2.15 的程序来处理带空格的回文序列。比如，I PREFER PI 是一个回文序列，因为如果忽略空格，它向前和向后读是一样的。

13. 为了实现 length 方法，我们在链表中计算节点的数目。一种代替的方法是链表中的节点的数目作为附加的数据片段储存在链表表头中。修改无序列表类，包含这个信息并且重新编写 length 方法。

14. 实现 remove 方法，使得当列表中没有相应数据项的时候它能正常运行。

15. 修改列表使它允许重复。有哪些方法将受到这种变化的影响？

16. 实现无序列表类的__str__方法。对于列表而言，什么是一个好的字符串形式的表现？

17. 实现__str__方法，使列表以 Python 的形式表现出来（用方括号）。

18. 实现无序列表中的其他操作(append, index, pop, insert)。

19. 实现一个无序列表的切片方法。它包含 start 和 stop 两个参数，返回一个从 start 位置开始向后到 stop 位置但不包含 stop 位置的列表的副本。

20. 实现定义在 OrderedList 里的其他功能。

21. 考虑无序和有序列表之间的关系。有没有可能将有序列表作为无序列表的一个继承，从而更有效的实现有序表？实现这个继承体系。

22. 实现一个使用链表的栈。

23. 实现一个使用链表的队列。

24. 实现一个使用链表的双端队列。

25. 设计并实现一个实验，使它可以比较 Python 内置的 list 和用链表实现的 list 的性能。

26. 设计并实现一个实验，使它可以比较用 Python 内置的 list 实现的栈、队列和用链表实现的栈、队列的性能。

27. 上述链表的实现被称为单向链表（singly linked list），因为每个节点在序列中有单一的对下一个节点的引用。另一种代替的实现方式被称为双向链表（doubly linked list）。在此实现中，每个节点都有对下一个节点的引用（通常称为“后继” next）和对前一个节点的引用（通常称为“前驱” back）。链表表头同样有两个引用，一个指向链表的第一个节点，一个指向最后一个。用 Python 实现这段代码。

28. 实现一个可以有平均值的队列。

4.递归 RECURSION

4.1 目标

了解某些用其它方法难解的问题或许有简单的递归解法

学会如何用递归的方式写程序

理解和运用递归的三大法则

了解递归是迭代（iteration）的一种形式

实现问题的递归描述

了解递归在计算机系统中如何实现

4.2 什么是递归

递归是一种解决问题的方法，它把一个问题分解为越来越小的子问题，直到问题的规模小到可以被很简单直接解决。通常为了达到分解问题的效果，递归过程中要引入一个调用自身的函数。乍一看，递归算法并没有什么特别的地方，但是，利用递归我们能够写出极为简明的解决问题的方法，而且如果不用递归，这些问题将具有很大的编程难度。

4.2.1 计算数字列表的和

我们先从一个简单的问题开始我们的探究，这个问题不需要递归也可以解决。假如你想对一个数字列表进行求和（例如[1,3,5,7,9]），代码 4.1 所示的是一个通过迭代函数（for 循环）求和的程序。这个函数用一个变化着的“累加器”变量（theSum）对列表里面所有的数进行累加求和，也就是从 0 开始，依次加上列表中的每个数。

代码 4.1 迭代求和

```
def list_sum(num_list):
    the_sum = 0
    for i in num_list:
        the_sum = the_sum + i
return the_sum
print(list_sum([1,3,5,7,9]))
```

现在，假设我们不能使用 while 循环或者 for 循环，那么你会如何对数字列表中的数进行求和呢？如果你是个数学家，那么你首先想到的也许是：按照定义，加法是一个有两个参数——两个数字——的函数。为了将数字列表的问题重新定义为对两个参数求和的问题，我们可以利用全括号的表达式来重新表示列表，就像下面这种形式：

((((1+3)+5)+7)+9)

我们也可以从另一个方向来加入括号：

(1+(3+(5+(7+9))))

我们注意到最内层的括号中是（7+9），这是不需要任何循环或者特殊的结构就能解决的。事实上，我们可以用以下一系列简化后的式子来计算最终的加和。

那么我们怎样将这个思想转化为 Python 代码呢？首先让我们从 Python 列表的角度来重新叙述这个问题。由于数字列表的和是列表中的第一个元素（numList[0]）和剩下所有的元素（numList(1:)）之和的和，求和问题可以归纳成以下的式子：

listSum(numList)=first(numList)+listSum(rest(numList))

在这个等式中，first 代表列表中的第一个元素，而 rest 代表的是列表中除了第一个元素以外的其他所有元素。此式很容易在 Python 中用代码 4.2 表示出来。

```
def list_sum(num_list):
    if len(num_list) == 1:
        return num_list[0]
    else:
        return num_list[0] + list_sum(num_list[1:])
print(list_sum([1,3,5,7,9]))
```

代码 4.2 递归求和

在这个代码中，有一些关键点需要注意。首先，在第二行中，我们检查这个列表中是否只有一个元素。这个检查非常关键，因为它是函数能结束运行的必要条件。对一个长度为 1 的列表来说，它的和显然只是这个列表中的数，其次，在第五行中，函数调用了自身！这就是我们把后一个计算程序称为“递归”的原因。递归函数就是一种调用自身的函数。

图 4.1 展示了对列表[1,3,5,7,9]求和的一系列递归调用和返回的过程。你可以将这一系列的递归调用看做一次次简化问题的过程。每次进行递归调用过程就是在解决一个规模更小的问题，当问题的规模达到最小时结束递归。

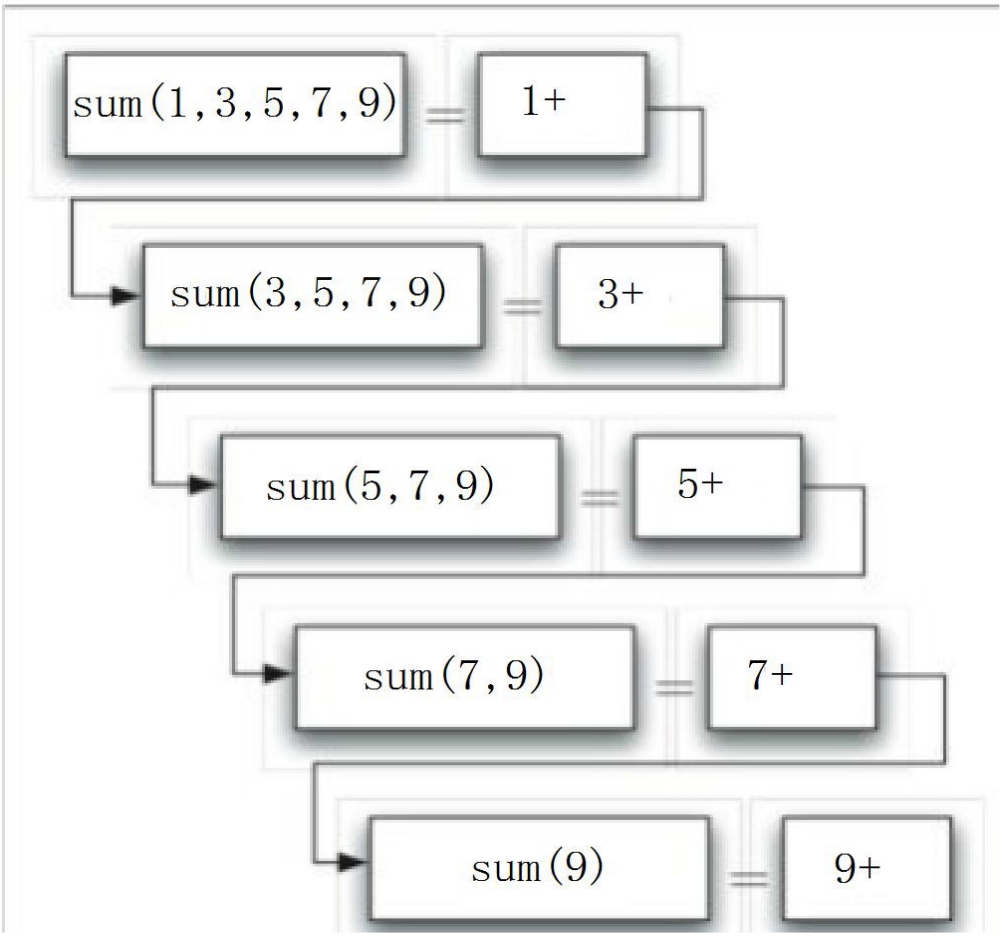


图 4.1：数字列表求和中的递归调用

当问题的规模缩小到我们可以轻易解决时，我们开始把每个小规模问题的答案连接起来直到解决最初提出的问题。图 4.2 展示了列表求和在一系列递归调用之后的回溯过程，当子列表的和返回到最顶端的求和问题时，我们就得到了整个问题的答案。

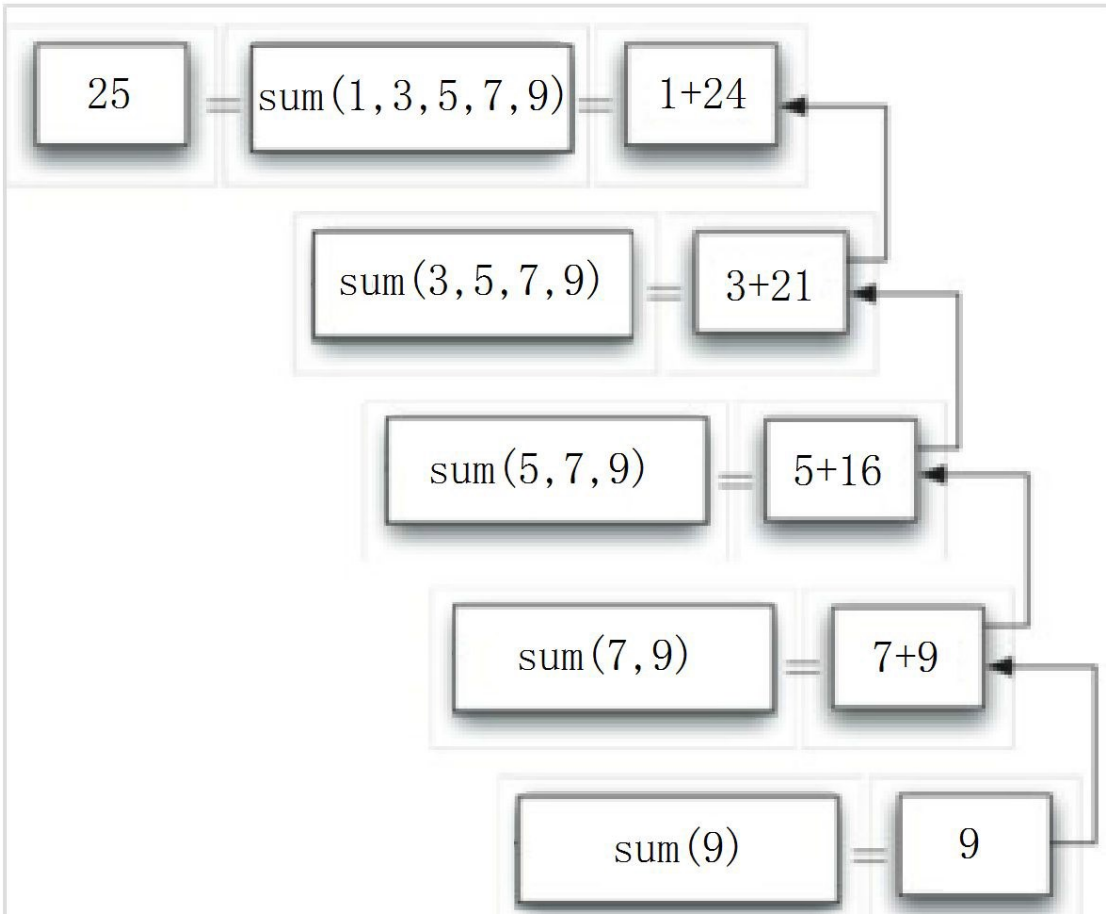


图 4.2：数字列表求和递归函数的返回过程

4.2.2 递归三大定律

就像阿西莫夫（I.Asimov）的“机器人三定律”一样，递归算法也要遵循三条重要的定律：

- 递归算法必须有个基本结束条件
- 递归算法必须改变自己的状态并向基本结束条件演进
- 递归算法必须递归地调用自身

让我们来更加细致地了解每一条定律，并看看它们是怎样在“数字列表求和”这个算法中体现的。首先，基本结束条件是一种可以让算法的递归操作结束的情况。基本的结束条件通常是一个规模小到可以直接解决的问题。在数字列表求和算法中，基本结束条件是一个长度为 1 的列表。

为了遵循第二定律，我们必须改变算法的状态，使之向基本结束条件演进。状态的改变意味着算法中使用的一些数据被改变了。通常情况下，这些代表着我们问题的数据以某种方式变小了。在数字列表求和算法中，基本的数据结构是一个列表，所以我们应该专注于在列表上下功夫来改变算法的状态。由于基本结束条件是长度为 1 的列表，那么向基本结束条件演进的最自然的过程就是缩短列表的长度。我们在代码 2 的第五行中可以看到具体的做法：将列表的长度减少 1，再调用求和算法来计算这个更短的列表。

最后一条定律就是算法必须调用自身，这正是”递归 “的定义。递归算法对于许多初学者来说是个难理解的概念。作为一个初学者，你已经看到了递归函数的优越性了吧，因为你可以把一个大问题不断分解为更小的问题，解决这些小问题只要分别写一些简单的函数就行了。当我们谈论递归的时候好像我们把自身置入了循环当中了。我们有一个问题需要用一个函数来解决，但是这个函数需

要调用自身来解决问题!但在逻辑上这是完全不循环的，递归的逻辑就是利用优美的程序把问题分解为更小更简单的子问题来解决的。

在这个章节的剩余部分我们可以看到更多的关于递归算法的例子，在每个例子中我们将关注如何根据递归三大定律来设计解决问题的方法。

小试牛刀：

Q-15 用 listsum 计算数列求和[2,4,6,8,10]要进行多少次递归调用？

A)6 B)5 C)4 D)3

Q-16 假设你要写一个关于计算某个数阶乘的递归算法。fact(n) return n*n-1*n-2…

规定 0！=1，什么将会是最合适的基本结束条件？

a)n==0

b)n==1

c)n>=0

d)n<=1

4.2.3.将整数转化成任意进制表示的字符串形式

假设你想要将一个整数转化为二进制到十六进制之间任意进制的字符串形式。例如，把整数 10 转换为十进制表示的字符串“10”，或二进制表示的字符串“1010”。尽管有很多算法可以解决这个问题，包括在”栈”的章节中讨论的算法，但用递归的思想解决该问题还是非常简洁优雅的。

下面我们将以十进制的 769 为例看一个具体的问题。假设我们有一个字符序列来对应前 10 个数字，形如 convString=“0123456789”。通过在字符串中检索，很容易将小于 10 的整数转换成对应的字符串。例如，如果数字是 9，那么字符串是 convString[9]或"9"。如果我们可以将数字 769 拆成三个单独的数字 7、6 和 9，然后很容易地将它转换成字符串。小于 10 的整数看起来是个不错的基本结束条件。

确定进位数之后整个算法将包含 3 个部分：

1）将原始整数分解为一连串的单个数字。

2）通过在字符序列中检索将单个数字转换成字符串。

3）将这些单个数字的字符串连接起来，形成最终的结果。

下一步就是要解决如何改变状态，使其向基本结束条件演进。既然我们正在讨论关于整数的问题，那么就让我们考虑一下什么数学运算会把一个数字分解。最有可能的运算是除法和减法。虽然减法可以实现，但我们并不清楚应该减去多少。整数的除法取余的运算给了我们一个明确的的方向。让我们来看看如果把试图转换的数字除以进位数会发生什么。

用整数的除法将 769 除以 10，我们得到 76 余 9。这给了我们两个好的结果。

首先，余数是一个小于进位数的数字，通过检索，它可以直接转换成一个字符。

第二，我们得到了一个小于初始数字的新数字"76"，它能让我们向着“找到小于进位数的单个数字”的基本结束条件演进。现在我们的任务是再将 76 转换为它的字符串形式。我们将再一次使用除法取余得运算，这样分别得到 7 和 6.

最后，我们将问题简化为转换数字 7 为字符串形式，因为它满足基本结束条件“n<base，base=10”，我们可以很容易转化。我们刚才的一系列操作如图 4.3 所示。注意，我们想要记住的数字在图示右侧的余数方框中。

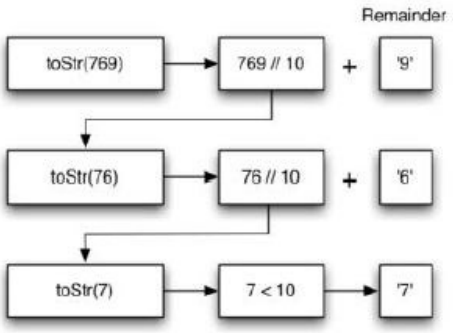


图 4.3：进制基数为 10 转换整数到字符串

代码 4.3 显示了进位数在 2 到 16 之间时实现上述算法的 Python 代码。

```
def to_str(n, base):
    convert_string = "0123456789ABCDEF"
    if n < base:
        return convert_string[n]
    else:
        return to_str(n / base, base) + convert_string[n % base]

print(to_str(1453, 16))
```

代码 4.3 递归转换整数到字符串

请注意，在第 3 行我们检查进位数 base，当 n 小于 base 时我们才会进行转换。当我们检测到基本结束条件时，我们会停止递归过程并且仅仅从 convertString 序列中返回字符串。在第 6 行里，我们在满足递归第二、三条定律的基础上，通过运用除法，做到了递归算法调用自身和减小问题规模这两点。

让我们再次回顾这个算法,这次我们将整数 10 转换为 2 进制表示的字符串。(“1010”)。

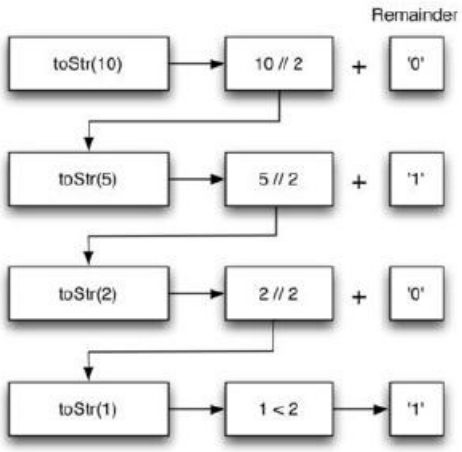


图 4.4：转换整数 10 为字符串形式的 2 进制表示

图 4.4 显示了我们得到了期待的结果，但是似乎数字的次序错了。该算法运行正确，因为我们在第 6 行先使用递归调用，然后添加了字符串表示的余数。如果我们反向返回 convertString 的查找结果并返回 toStr 调用，生成的字符串将会反向！但通过递归调用返回结果后再进行连接操作，我们得到了正确的结果。这应该使你想起我们在前一章中对栈的讨论。

小试牛刀：

写一个函数，接受一个字符串作为参数，并返回一个反向的新字符串。

写一个函数，接受一个字符串作为参数，如果字符串是一个回文，则返回 True，否则返回 False。如果一个字符串向前和向后的拼写均相同，那么它是一个回文。例如:radar 是一个回文。一些优秀的回文也可以是短语,但是你需要在验证前删除空格和标点符号。例如:madam i’m adam 是一个回文。其他有趣的回文包括:

- kayak
- aibohphobia
- Live not on evil
- Reviled did I live, said I, as evil I did deliver
- Go hang a salami; I’m a lasagna hog.
- Able was I ere I saw Elba
- Kanakanak （阿拉斯加州的一个城镇）
- Wassamassaw（南达科他州的一个城镇）

4.3 栈帧：实现递归

```
import Stack # As previously defined
r_stack = Stack()
def to_str(n, base):
        convert_string = "0123456789ABCDEF"
```

代码 4.4

```
    while n > 0:
        if n < base:
            r_stack.push(convert_string[n])
        else:
            r_stack.push(convert_string[n % base])
        n = n // base
    res = ""
    while not r_stack.is_empty():
        res = res + str(r_stack.pop())
    return res
print(to_str(1453, 16))
```

我们设想一下，如果不是按上文“toStr()”那样通过递归调用将所得的单个字符连在一起输出得到结果，而是通过修改算法，以类似于递归执行时的先后顺序，把这些字符压入一个栈中来得出结果，会是如何呢？修改后的代码如下所示：

在上一节的递归算法中，当我们每次递归调用 toStr() 时，就相当于这里把一个字符压入了栈中，回到上一个例子中我们能发现当我们第四次调用 toStr()这个函数后，这里的栈就会变成如图 4.5 所示。现在我们只需要把栈中的元素推出栈，然后再把他们连接在一起输出就得到了最后的结果：“1010”

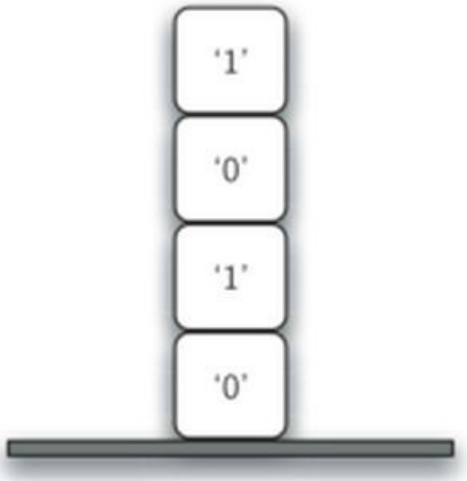


图 4.5 在进制转换时栈中所存字符

前面的例子使我们能够了解 Python 是如何执行递归函数的调用的。在 Python 中，当一个函数被调用时，系统会分配一个栈帧去处理函数中的那些局部变量。当执行完函数，并得到了一个返回值时，这个值会被留在栈顶等待被调用的函数来处理。图 4.6 所表示的是图 4.4 中第四行得到返回值后栈的情形：

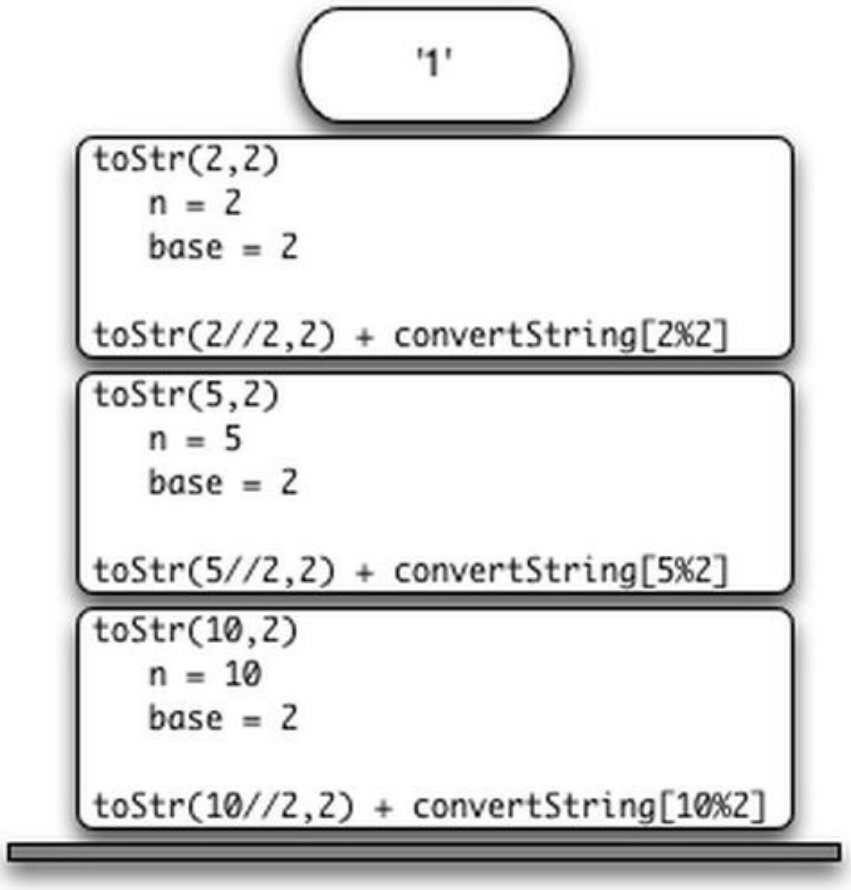


图 4.6 执行 toStr(10,2)的调用栈

要注意的是，当调用 toStr(2//2,2) 时，会在栈顶留下一个返回值“1”，这个返回值随后会被用于 toStr(2//2,2)+convertString[2%2] 中替代 toStr(2//2,2)，使语句变为”1”+convertString[2%2]，这就会导致执行后，字符串“10”会被留在栈顶。在这一点上，Python 的调用栈和我们修改后的程序（Listing4）中所使用的栈不同，在这个例子中，你可以认为是栈顶的返回值替代了 Listing4 中累加器的作用。栈帧也给函数所能使用的变量规定了作用域，即使我们一遍又一遍的调用相同的函数，每次调用都会给函数中的局部变量产生一个新的作用域。（作用域：变量名在程序中的可用范围）

4.4. 图示递归

在前一节中，我们学习了一些运用递归就很容易解决的问题，然而，有时候建立一个递归的思维模型，或者是将递归函数的执行过程实现可视化是很困难的，这就导致了人们很难掌握递归。在这一节中，我们将看几个用递归画出有趣的图案的例子。当你看到这些图案的成形过程，你就会对递归的过程有更深的领悟，从而帮助你更好的理解递归。

代码4.5:用海龟画一个递归螺线

```
import turtle


myTurtle = turtle.Turtle()
myWin = turtle.Screen()


def drawSpiral(myTurtle, lineLen):
    if lineLen > 0:
        myTurtle.forward(lineLen)
  myTurtle.right(90)
  drawSpiral(myTurtle,lineLen-5)


drawSpiral(myTurtle,100)
myWin.exitonclick()
```

我们用来作图的工具是 Python 的海龟作图模块，其名称为 turtle。海龟作图模块是所有版本的 Python 的标配，并且操作简单，具体表现也很简单。你可以创建一只海龟，然后它就可以执行前进、后退、向左转、向右转等一系列操作。你也可以控制海龟的尾巴抬起和放下。当海龟的尾巴放下来的时候，随着海龟的移动，它的尾巴就会在屏幕上留下痕迹，也就是在画图了。为了提升图案的艺术价值，你可以改变海龟尾巴（也就是画笔）的宽度，也可以改变制图时画笔的颜色。

接下来我们就通过一个简单的例子，来展示一些海龟作图的基本方法。我们将会用海龟通过递归的方法来画一个螺旋线，代码 4.5 便是实现这个的代码。在我们引入了 turtle 模块后，我们创建一只海龟。当一只海龟被创建后，我们也就同时创建了一个能让它在里面作图的窗口。接下来我们就来定 drawSpiral() 这个函数。这个简单函数的基本结束条件是：当我们想画的线的长度（也就是函数中的 lineLen 变量）减小到 0 或者小于 0。当线的长度大于 0 时，我们便命令海龟前进 lineLen 个单位，然后向右转 90 度，然后在这里递归，再度执行 drawSpiral()函数，但是传入一个小于 lineLen 的值作为要画的线的长度。在代码的最后，我们调用了 myWin.exitonclick() 函数，这是一个便利的小技巧，它将乌龟至于待机状态，等到你单击窗口时，页面才会清空，然后程序退出。

这基本就是海龟作图模块中你需要知道的所有基本内容，有了这些，你就能画出一些漂亮的图案了。下一个程序中我们将来画分形树。分形（fractal）来源于数学中的一个分支，并且和递归颇有相似。分形的定义是，无论你如何放大图形，你所看到的每一个区域中的碎片形状都和原来的形状相同。在自然界中，分形的例子有大陆的海岸线、雪花、山脉，即便是树和灌木也可以算是分形的一种。正是自然界中这些广泛存在的分形现象，使得程序员们能够为动画电影制作出非常逼真的场景。我们的下一个例子就是画一颗分形树。

为了搞明白如何画一棵分形树，想想如何用分形的思维方式来描述一棵树是很有用的。一定要记住我们上面所说的分形指的是那些无论如何放大都有自相似性的东西，如果用它来描述一棵树或者灌木，我们就可以说，树的每一个小分枝都有与整棵树相同的形状和特征。利用这种想法，我们就可以将树定义为一个向左右分叉的躯干，如果再引入递归的概念，反复的将其应用于整棵树，就可以理解为一颗大的树就是由这些不断左右分叉的小的树组成的。

现在就让我们来把上面的这个思路翻译为 Python 代码，如下所示。如果你仔细来看，第五行和第七行实现的是递归调用，第五行是在实现海龟右转 20 度后进行的递归调用，这就是上一段中所说到的向右的分叉；然后在第七行中，海龟又进行了第二次递归调用，这一次是在向左转 40 度后执行的。海龟必须向左转 40 度的原因是它必须抵消先右转的 20 度，然后再向左转额外的 20 度才能画左边的树。此外，我们需要注意，每次我们对 tree 进行递归调用的时候，branchLen 会被减去一些

```
def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-10,t)
        t.right(20)
        t.backward(branchLen)
```

量，这是为了确保递归出的分叉会越来越小。还需要注意开头第二行的 if 语句，它是对整个递归的基本结束条件的检验，当树枝短于一定长度，递归就会被终止。

代码 4.6 是这个分形树例子的完整代码，在你运行这段代码以前，请你设想一下这棵树是如何成形的，仔细研究一下那些递归调用，想想这棵树的最终会是怎样一个形状；它会是左右对称，两边同时画？还是会先画右边然后再画左边？

请留心观察，树上每一个分叉点是如何对应一次递归调用的，以及这棵树是如何一路朝右画直到它最短的分支的？你可以在图 4.7 中看到这些。现在请你观察，直到画完整个右半部分之前，这个程序在画完一支后是如何回归树干的？你可以在图 4.8 中看到这棵树的整个右半部分。然后，程序便会继续运行来画树的左半部分，然而它并不是在每次分叉的时候直接画最左边的那一支，而是像之前那样，在到达最短的那根树枝之前，先画出左半分支的整个右半部分，然后再返回画左半部分。

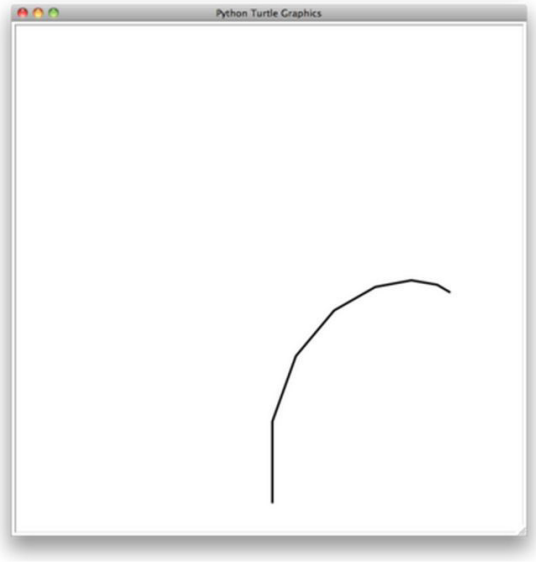


图 4.7 分形树的起始

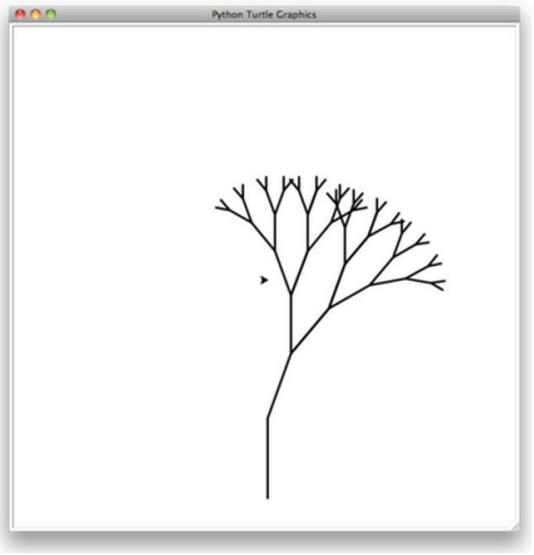


图 4.8 分形树的第二部分——右半部分

```
import turtle


def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-15,t)
        t.right(20)
        t.backward(branchLen)


def main():
    t = turtle.Turtle()
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75,t)
    myWin.exitonclick()


main()
```

代码 4.6：用递归法画一棵树

这个简单的画树程序对你来说只是一个起点，或许你也已经发现这棵树看起来并不真实，因为自然界并不总像这个电脑程序所描绘的那样对称。这章结束的练习题会给你提供一些想法，教会你去探索一些有趣的选项来使你的树看上去更真实。

牛刀小试

用下面一个或者所有的选项来修改你的递归树程序：

1. 修改树枝的粗细，实现随着树枝的长度缩短，线条也相应变细

2. 修改树枝的颜色，实现当树枝非常短的时候，使它拥有像树叶的颜色

3. 修改海龟转向时候的角度，使得每根树枝的倾斜角度可以在一定范围内随机变化，例如选取在 15 到 45 之间变化，多试几次，做成你觉得最好看的样子

4. 修改递归调用时树枝的长短，每次可以减去一个随机数，而不是一个固定值

4.4.1. 谢尔宾斯基三角形

另外一种展现自相似性的分形图形就是谢尔宾斯基三角形，如图三所示，它展现的是一个三向递归的算法。徒手画一个谢尔宾斯基三角形的步骤非常简单：从单个的大三角形开始，取它的各边中点作三条中位线，这样就把它分成了四个新的三角形；剔除掉这四个新三角形中最中间的那个，对其余三个角上的三角形重复以上的操作。每当你画出这一系列的三角形之后，你就可以不停地将这些步骤应用于那三个角上的三角形。如果你的铅笔足够细，你就能无限的重复这些步骤。现在，我想你在继续读下去之前很可能要自己尝试画谢尔宾斯基三角形了，那就用上面的方法画一个吧！

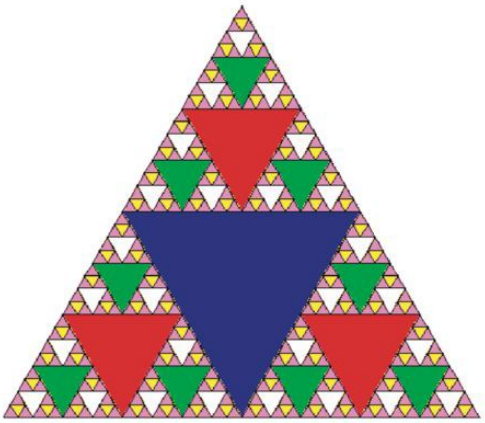


图4.9 谢尔宾斯基三角形

既然我们可以无限地运行这个算法，那么它的基本结束条件又是什么呢？我们可以看出，它的基本结束条件可以被任意设置为我们想要它划分三角形的次数，我们想要它执行几次，他就可以执行几次。有时候，这个划分的次数被称为相似形维数，每当我们执行一次递归调用的时候，相似性维数就减 1，直到它减到 0 为止，这时候便停止递归调用。代码 4.7 便是画出谢尔宾斯基三角形的代码。

```
def sierpinski(points,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow', \
                'violet','orange']
    drawTriangle(points,colormap[degree],myTurtle)
    if degree > 0:
        sierpinski([points[0], \
                    getMid(points[0], points[1]), \
                    getMid(points[0], points[2])], \
                   degree-1, myTurtle)
        sierpinski([points[1], \
                    getMid(points[0], points[1]), \
                    getMid(points[1], points[2])], \
                   degree-1, myTurtle)
        sierpinski([points[2], \
                    getMid(points[2], points[1]), \
                    getMid(points[0], points[2])], \
                   degree-1, myTurtle)
def main():
    myTurtle = turtle.Turtle()
    myWin = turtle.Screen()
    myPoints = [[-100,-50],[0,100],[100,-50]]
    sierpinski(myPoints,3,myTurtle)
    myWin.exitonclick()
main()
```

代码4.7：画一个谢尔宾斯基三角形

这个程序遵循了前面所说的画法，Sierpinski 函数做的第一件事就是画最外面的大三角形，然后就有三个递归调用，每一次调用都是为了划分三个连中点所得的小三角形中的一个。这里，我们再一次使用 Python 的海龟作图模块。你可以在 Python 提示符之后输入 help(“turtle”)来查看这个模块中所有可以被使用的功能函数。

仔细观察代码然后思考画三角形的顺序是怎么样的？确切的顺序应当是取

决于最开始是如何设定的，现在我们假设这个顺序是先左下，再上，再右下，又因为 Sierpinski 函数调用了它自己，所以它直到它所需要划分的最小的三角形，都是从左下角的三角形开始分割，分割完成后，去掉中间的三角形后用颜色填充剩下的三角形，然后返回上方角落的三角形，待其分割到最小的时候，去掉中间的，然后给剩下的三角形填充颜色，最后，转到右下角的三角形，同样划分到最小，然后再填充颜色。

有时候，根据函数调用的图示去理解递归算法是很有帮助的。图 4 告诉了我们 Sierpinski 函数的递归调用总是先处理左下角的三角形，图中轮廓线为黑色的是正在被执行的函数操作，被灰色填充的是等待被执行的操作，越靠近图 4.10 的底部，分割出的三角形就越小。Sierpinski 一次就执行一个相似性维数，一旦图中左下角的操作被完成，它就会继续去处理中间的，然后是右边的，以此类推……

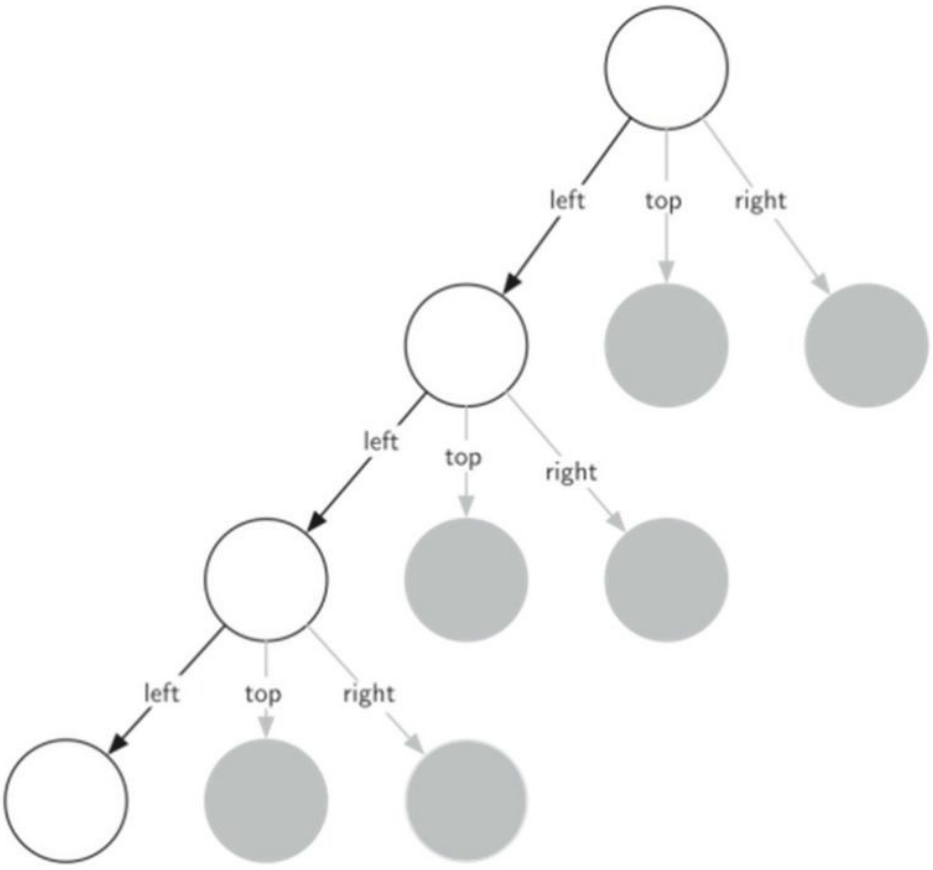


图4.10　Sierpinski三角形的建立过程

Sierpinski 函数很大程度上依赖于 getMid 函数，getMid 提取两个参数作为两个端点，然后返回这两个端点组成的线段的中点。此外，代码 4.7 中还有一个函数用于给三角形填充颜色，它使用了 Python 海龟作图中的 begin_fill 和 end_fill 操作。

4.5.复杂递归问题

在之前的章节中我们探究了一些相对容易和灵活有趣的问题来帮助大家理解递归。在这个部分我们将考虑一些用迭代法解决很困难但用递归却很简单的问题。最后我们会以一个看似能用简洁的递归解决但实际却不能的问题来作为结尾。

4.5.1.河内塔问题

河内塔问题是法国数学家爱德华·卢卡斯于1883年发现的。他受到一个关于印度教寺庙的传说的启发，故事中这一问题交由年轻僧侣们解决。最开始，僧侣们得到三根杆子，64个金圆盘堆叠在其中一根上，每个圆盘比其下的小一点。僧侣们的任务是将64个圆盘从一根杆上转移到另一根杆上，但有两项重要的限制,一是他们一次只能移动一个圆盘,一是不能将大圆盘放在小圆盘之上。僧侣们日以继夜地工作，每秒移动一个圆盘。 传说中，当工作完成之时寺庙就会崩塌，世界则将不复存在。

传说很有趣，但也不用为世界末日将要到来而担心。毫无差错地完成这项工作需要$2^{64}-1=18,446,774,073,709,551,615$次移动。如果每秒移动一次则需要584,942,417,335年!显然实际上会更长。

图4.11展示了圆盘从第一根杆移到第三根杆过程中的一个状态。我们注意到,根据规则,圆盘在每根杆上都是从大到小堆放的以保持小盘永远在大盘上面。如果你之前没有尝试过解决这个问题，现在可以试一下。不需要想象圆盘和杆，一沓书或者纸就可以。

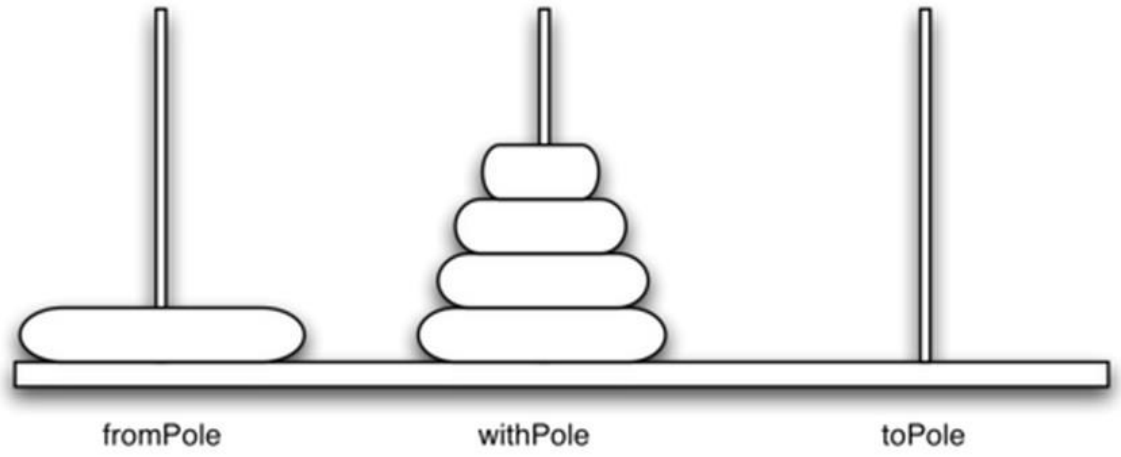


图4.11:河内塔问题的一个例子

我们如何用递归解决这个问题呢？我们又如何完全解决这一类问题？基本情况又是什么？让我们从递归调用的最底端入手。假设你有一个五个圆盘组成的塔，最开始在一号杆上。如果你已经知道如何将有四个圆盘的小塔从一号杆移到二号杆,就可以很容易地将第五个圆盘移动到三号杆,然后将四个圆盘的塔从二号杆移动到三号杆。但是如果你不知道如何移动有四个圆盘的塔呢？这时又假设你知道如何将有三个圆盘的塔移到三号杆；然后你就可以将第四个圆盘移动到二号杆，然后再将位于三号杆的有三个圆盘的塔移到其上。但是如果你不知道如何移动有三个圆盘的小塔又怎么办呢？那考虑先将有两个圆盘的小塔移动到二号杆,再将第三个圆盘移动到三号杆,最后将两个圆盘的小塔移动到三号杆会如何呢？但如果你连这个也不会，该如何处理？显然你知道将单个圆盘移到三号杆十分简单甚至可以说无需思考，似乎这就是这一问题最基础的部分。

下面是关于将塔经由中间杆，从起始杆移到目标杆的抽象概述：

1、把圆盘数减一层数的小塔经过目标杆移动到中间杆☒

2、把剩下的圆盘移动到目标杆☒

3、把圆盘数减一层数的小塔从中间杆，经过起始杆移动到目标杆

只要我们一直遵循大的圆盘保持在底层的规则，我们就能用以上的三步来递归，很容易地处理任意多圆盘的问题。上述概要唯一缺少的是对基本情况的识别。最简单的河内塔问题是只有一个圆盘时的情况，在这种情况下，我们只需要将单个圆盘移动到它的目标杆。所以只有一个圆盘的情况是我们的基本情况。此外，上述步骤可以通过减少1、3步中小塔的高度使问题向基本情况靠近。

可以发现代码描述与英文描述非常接近。算法简化的关键在于两个不同的递归调用的使用，分别在第三行和第五行。第三行中，我们把起始杆上的除了最下面的圆盘全部移动到中间杆，下一行将原来在最底层的最大圆盘移动到目标杆，之后在第五行，我们将位于中间杆的圆盘移动到最大圆盘的上面。当小塔高度为0即为最简情况，在这种情况下无需继续，函数可以直接返回。要记住这种

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
```

对基本情况的处理的重要之处在于moveTower的返回是moveDisk被调用的先决条件。

下面展示的moveDisk函数非常简单,它所做的事情就是输出一个圆盘从一根杆移动到另一根杆的过程。如果你输入并运行 moveTower程序，将发现它能很高校地解决河内塔问题。

以下程序为三个圆盘的情况的一个完整解决方案。

```
def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)
```

现在在读过moveTower和moveDisk的代码后，你可能会好奇为什么我们没有一个用以精确地追踪哪一个圆盘在哪一个杆上的数据框。这里有一个提示: 如果你要精确地记录圆盘的移动,你可以用

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)
moveTower(3,"A","B","C")
```

三个栈分别对应三根杆。Python提供了栈,我们只需要调用就行了。

4.6.探索迷宫

在本节中,我们将着眼一个关于拓宽机器人活动范围的问题：如何走出迷宫？如果你有一个打扫房间的Roomba吸尘机器人，你可能会想根据你在本章所学知识对它重新编程了。我们要做的是帮助海龟在虚拟的迷宫寻找出路。迷宫问题可以追溯到希腊神话中忒修斯进入迷宫猎杀人身牛头怪的故事。忒修斯用了一个线球以使自己在杀掉怪物后能够找到出口。在我们的问题中，我们假设我们的海龟落入迷宫的中央，需要找到出路。请看图4.12思考如何走出迷宫。

为了简化问题,我们假定这个迷宫被分成正方形。每块小方格要么开放要么被不可通过的墙壁占据。海龟只能通过迷宫中开放的部分,如果海龟遇到墙就需要尝试不同方向。海龟需要一个系统的步骤以找到正确的方式走出迷宫,下面是具体的步骤:

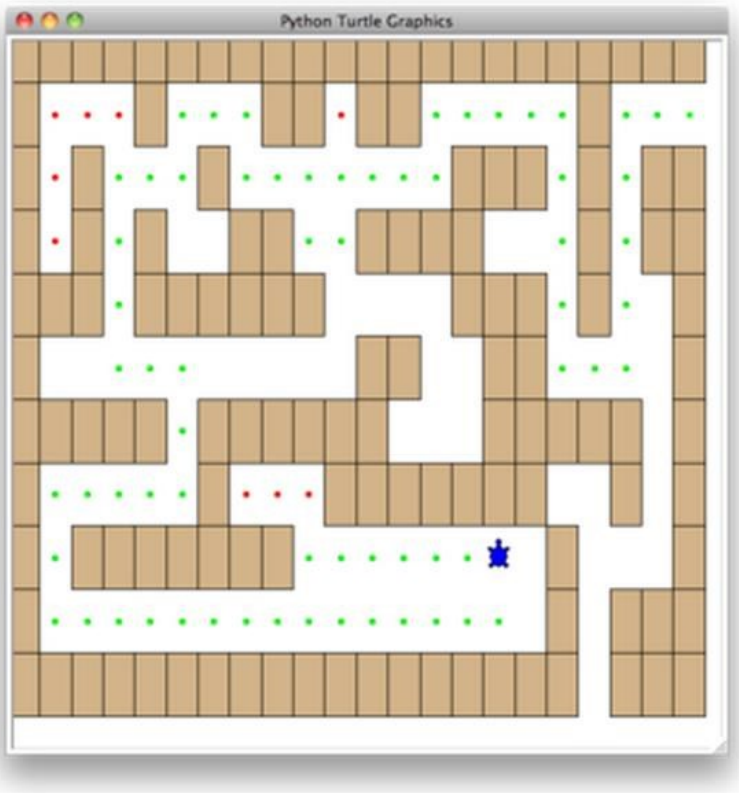


图4.12已解决的迷宫问题

- 在初始位置尝试向北走一步，以此为开始递归程序。 ☒
- 北面走不通的情况下则向南尝试，其后开始递归。☒
- 南面走不通的情况下则向西尝试，其后开始递归。☒
- 如果北面、南面、西面都走不通，则从东面开始并递归程序。 ☒
- 如果四个方向都走不出，则被困在迷宫中，以失败告终。

尽管看起来简单但仍有几处细节需要说明。假设我们第一步迈向北，按照程序下一步也将是向北。但是如果北边是墙壁, 我们将按程序的下一步尝试向南。不幸的是走向南后，将回到起点。按照递归将陷入无尽的循环。因此, 我们需要一个记住走过的路径的方法。想象我们有一包面包屑可以沿路撒下，如果我们准备向某个方向前进时发现路上已经有了面包屑，我们应该立刻退回并尝试下一个方向。在算法的代码中我们将看到回退一步与返回递归调用同样方便。

对于所有的递归算法我们都要找到基本情况。在前面几段描述中，已经设想了几种情况。在此算法中，有四种基本情况需要考虑: ☒

1、海龟碰到“墙壁”，方格被占用无法通行。

2、海龟发现表示此方格已访问过，为避免陷入循环不在此位置继续寻找。

3、海龟碰到位于边缘的通道，即找到迷宫出口。

4、海龟在四个方向上探索都失败。

为了让程序运行，我们需要用一种方法来表示迷宫。为了使之更为生动，我们将用海龟模块来绘制并探索迷宫，这样我们可以看到算法的动态效果。迷宫对象将提供以下几种方法供我们在写算法时使用：

- __init__ 用以读取迷宫数据，初始化迷宫内部，并找到海龟初始位置。
- draw_maze 用以在屏幕上绘制迷宫。
- update_position 用以更新迷宫内的状态及在窗口中改变海龟位置。
- is_exit 用以判断当前位置是否为出口。

迷宫类包含索引操作符［］以使算法能够方便地存取任一方格的状态。

让我们来看一下被称为searchFrom的搜索代码。代码如下述。注意此函数包括三个参数：一个

```
def searchFrom(maze, startRow, startColumn):
      maze.updatePosition(startRow, startColumn)
    #   Check for base cases:
    #   1. We have run into an obstacle, return false
    if maze[startRow][startColumn] == OBSTACLE :
            return False
     #   2. We have found a square that has already been explored
      if maze[startRow][startColumn] == TRIED:
return False
# 3. Success, an outside edge not occupied by an obstacle
if maze.isExit(startRow,startColumn):
      maze.updatePosition(startRow, startColumn, PART_OF_PATH)
      return True
maze.updatePosition(startRow, startColumn, TRIED)
# Otherwise, use logical short circuiting to try each
# direction in turn (if needed)
found = searchFrom(maze, startRow-1, startColumn) or \
            searchFrom(maze, startRow+1, startColumn) or \
            searchFrom(maze, startRow, startColumn-1) or \
            searchFrom(maze, startRow, startColumn+1)
if found:
      maze.updatePosition(startRow, startColumn, PART_OF_PATH)
else:
      maze.updatePosition(startRow, startColumn, DEAD_END)
return found
```

迷宫对象、起始行、起始列。由于递归中每一次调用在逻辑上都要重新开始搜索，这一点十分重要。

在算法中可以看到代码所做的第一件事是调用updatePosition（第二行）。这一步是为了将算法可视化，如此，海龟在迷宫中的移动就可以被观察到。接下来算法查验了四种基本情况的前三种：海龟是否碰壁 (第五行)？海龟是否回到标记过的地方(第八行)？海龟是否找到了出口(第十一行)？如果这些情况无一符合，递归搜索继续。

你会发现,在递归过程中有四次递归调用searchFrom。很难估算这些递归将被调用多少次，因为它们彼此联系。如果searchFrom第一次调用返回true，则后三个不需要调用。可以将此理解为向（当前行数－1，当前列数）前进一步（即向北）是走出迷宫的一个步骤。如果向北走并非可行之路则将尝试向南的递归调用。如果南边失败，则尝试向西,最后是向东。如果全部四个递归调用都返回false，则说明是死胡同。这时应重新编译程序，对不同的调用顺序进行尝试。

迷宫类的代码如下所示。__init__方法接受一个文件名称作为其唯一参数。此文件是使用“+”字符作为墙壁围出空心正方形空间,并用字母“S”来表示起始位置的迷宫文本文件。下面即为一个迷宫的例子。

```
++++++++++++++++++++++
+   +   ++ ++        +
      +     ++++++++++
+ +    ++  ++++ +++ ++
+ +   + + ++    +++  +
+          ++  ++  + +
+++++ + +      ++  + +
+++++ +++  + +  ++   +
+          + + S+ +  +
+++++ +  + + +     + +
++++++++++++++++++++++
```

图4.13 一个迷宫的例子

迷宫的内部则数据项为字符列表的列表表示。所述maze_list实例变量的每一行也是列表。该二级列表中每一个位置由上述一种字符填充。数据文件的内部表示如下:

```
[ ['+','+','+','+',...,'+','+','+','+','+','+','+'],
  ['+',' ',' ',' ',...,' ',' ',' ','+',' ',' ',' '],
  ['+',' ','+',' ',...,'+','+',' ','+',' ','+','+'],
  ['+',' ','+',' ',...,' ',' ',' ','+',' ','+','+'],
  ['+','+','+',' ',...,'+','+',' ','+',' ',' ','+'],
  ['+',' ',' ',' ',...,'+','+',' ',' ',' ',' ','+'],
 ['+','+','+','+',...,'+','+','+','+','+',' ','+'],
  ['+',' ',' ',' ',...,'+','+',' ',' ','+',' ','+'],
  ['+',' ','+','+',...,' ',' ','+',' ',' ',' ','+'],
  ['+',' ',' ',' ',...,' ',' ','+',' ','+','+','+'],
 ['+','+','+','+',...,'+','+','+',' ','+','+','+']]
```

update_position方法，如下所示，采用内部显示观察海龟是否已经碰壁。它还用“.”或“ - ” 更新内部表示来指示乌龟以访问过的位置与死胡同。此外，该方法使用两个辅助方法move_turtle和drop_bread_crumb以更新屏幕上的图。

最后,is_exit方法用当前位置验证出口。当海龟到达0行、0列、最右或最末行时满足出口条件。

```
        rowsInMaze = rowsInMaze + 1
        self.mazelist.append(rowList)
clas    columnsInMaze = len(rowList)
        self.rowsInMaze = rowsInMaze
        self.columnsInMaze = columnsInMaze
        self.xTranslate = -columnsInMaze/2
        self.yTranslate = rowsInMaze/2
        self.t = Turtle(shape='turtle')
        setup(width=600,height=600)
        setworldcoordinates(-(columnsInMaze-1)/2-.5,
                            -(rowsInMaze-1)/2-.5,
                            (columnsInMaze-1)/2+.5,
col =                       (rowsInMaze-1)/2+.5)
for c


        def drawMaze(self):
            for y in range(self.rowsInMaze):

    col = col + 1
```

```
            for x in range(self.columnsInMaze):
                if self.mazelist[y][x] == OBSTACLE:
                    self.drawCenteredBox(x+self.xTranslate,-y+self.yTranslate,'tan')
self.t.color('black','blue')
def drawCenteredBox(self,x,y,color):
    tracer(0)
    self.t.up()
    self.t.goto(x-.5,y-.5)
    self.t.color('black',color)
    self.t.setheading(90)
    self.t.down()
    self.t.begin_fill()
    for i in range(4):
        self.t.forward(1)
        self.t.right(90)
    self.t.end_fill()
    update()
```

```
        tracer(1)
    def moveTurtle(self,x,y):
        self.t.up()
        self.t.setheading(self.t.towards(x+self.xTranslate,-y+self.yTranslate))
        self.t.goto(x+self.xTranslate,-y+self.yTranslate)
    def dropBreadcrumb(self,color):
        self.t.dot(color)
    def updatePosition(self,row,col,val=None):
        if val:
            self.mazelist[row][col] = val
        self.moveTurtle(col,row)
        if val == PART_OF_PATH:
            color = 'green'
        elif val == OBSTACLE:
            color = 'red'
        elif val == TRIED:
            color = 'black'
        elif val == DEAD_END:
```

```
            color = 'red'
        else:
            color = None
        if color:
            self.dropBreadcrumb(color)


def isExit(self,row,col):
        return (row == 0 or
                    row == self.rowsInMaze-1 or
                    col == 0 or
                    col == self.columnsInMaze-1 )
def __getitem__(self,idx):
        return self.mazelist[idx]
```

完整的程序展示在后。程序使用以下所示存有迷宫的数据文件maze2.txt。

```
++++++++++++++++++++++
+   +   ++ ++         +
      +     ++++++++++
+ +    ++  ++++ +++ ++
+ +   + + ++    +++  +
+          ++  ++  + +
+++++ + +      ++  + +
+++++ +++  + +  ++   +
+          + + S+ +  +
+++++ +  + + +     + +
++++++++++++++++++++++
```

应注意本例中出口与海龟位置距离很近故十分简单。

```
# Completed maze program
# Takes maze2.txt as input
import turtle
PART_OF_PATH = 'O'
TRIED = '.'
OBSTACLE = '+'
DEAD_END = '-'

class Maze:
    def __init__(self, maze_file_name):
        rows_in_maze = 0
        columns_in_maze = 0
        self.maze_list = []
        maze_file = open(maze_file_name,'r')
        rows_in_maze = 0
        for line in maze_file:
            row_list = []
            col = 0
```

```
            for ch in line[: -1]:
                (columns_in_maze - 1) / 2 + .5,
                (rows_in_maze - 1) / 2 + .5)
    def draw_maze(self):
        self.t.speed(10)
        for y in range(self.rows_in_maze):
            for x in range(self.columns_in_maze):
                if self.maze_list[y][x] == OBSTACLE:
                    self.draw_centered_box(x + self.x_translate,
                        - y + self.y_translate, 'orange')
        self.t.color('black')
        self.t.fillcolor('blue')
    def draw_centered_box(self, x, y, color):
        self.t.up()
        self.t.goto(x - .5, y - .5)
        self.t.color(color)
        self.t.fillcolor(color)
        self.t.setheading(90)
        self.t.down()
        self.t.begin_fill()
```

```
            elif val == TRIED:
                color = 'black'
            elif val == DEAD_END:
                color = 'red'
            else:
                color = None
            if color:
                self.drop_bread_crumb(color)
    def is_exit(self, row, col):
        return (row == 0 or
                row == self.rows_in_maze - 1 or
                col == 0 or
                col == self.columns_in_maze - 1)
    def __getitem__(self,idx):
        return self.maze_list[idx]
def search_from(maze, start_row, start_column):
    # try each of four directions from this point until we find a
way out.⊠# base Case return values:⊠# 1. We have run into an obstacle, return false
```

```
maze.update_position(start_row, start_column) if maze[start_row][start_column] == OBSTACLE :
return False☒# 2. We have found a square that has already been explored
    if maze[start_row][start_column] == TRIED      or\maze[start_row][start_column] ==
DEAD_END:
        return False
    # 3. We have found an outside edge not occupied by an obstacle
    if maze.is_exit(start_row,start_column):
        maze.update_position(start_row, start_column, PART_OF_PATH)
        return True
    maze.update_position(start_row, start_column, TRIED)
    # Otherwise, use logical short circuiting to try each direction
    # in turn (if needed)
    found = search_from(maze, start_row-1, start_column) or \
            search_from(maze, start_row+1, start_column) or \
            search_from(maze, start_row, start_column-1) or \
            search_from(maze, start_row, start_column+1)
    if found:
        maze.update_position(start_row, start_column, PART_OF_PATH)
```

```
    else:
        maze.update_position(start_row, start_column, DEAD_END)
return found
my_maze = Maze('maze2.txt')
my_maze.draw_maze()
my_maze.update_position(my_maze.start_row, my_maze.start_col)
search_from(my_maze, my_maze.start_row, my_maze.start_col)
```

自我测试

1. 修改迷宫搜索程序,使得调用searchFrom按照不同的顺序进行。观察程序运行。你能否解释为什么程序的行为不同？能否预测一下顺序改变后海龟的路线？

4.7 动态规划

在计算机科学中，许多程序是为使一些问题得到最优解而写；例如，找到两点间的最短路径，找到最匹配一组点的线，或找到满足某些条件的最小对象集。计算机学家有许多策略来解决这些问题。这本书的目的之一就是为你揭示几个不同的解题策略。**动态规划**是这类求最优解问题的解决策略之一。

优化问题的一个典型例子就是用最少的硬币来找零。假设你是一家自动售货机制造商的程序员。你的公司正设法在每一笔交易找零时都能提供最少数目的硬币以便工作能更加简单。假设一个顾客投了1美元来购买37美分的物品。你用来找零的硬币的最小数量是多少？答案是六枚硬币：两个25 美分，一个10美分，三个1美分。我们是怎么得到六个硬币这个答案的呢？首先我们要使用面值中最大的硬币（25美分），并且尽可能多的使用它，接着我们再使用下一个可使用的最大面值的硬币，也是尽可能多的使用。这种方法被称为**贪心算法**，因为我们试图尽可能快的解决一个问题。

当我们使用美国硬币时，贪心算法工作的很好，但假设你的公司决定在Lower Elbonia （注释： 漫画中杜撰的原东欧共产主义国家的南部）也部署自动售货机，那个地方除了有 1，5，10和25美分的硬币外，还有21美分的硬币。在这种情况下，贪心算法就不能找到63美分找零问题的最优解了。多了21面值的美分，贪心算法的答案仍是六个硬币，然而问题的最优解是三个21美分的硬币。

让我们来看看一个肯定能让我们找到问题的最优解的算法。既然这一章是关于递归的，你可能已经猜到我们将使用递归的方法解决问题。首先我们要弄清楚基本结束条件。如果我们要找的零钱的价值和某一种硬币的价值一样，那么答案很简单，只要一个硬币。

如果价值不匹配，我们就有几种选择。我们需要的是一个 1 美分加上给原始价值减去 1 美分找零所需硬币数量的最小值，或者一个 5 美分加上给原始价值减去 5 美分找零所需硬币数量的最小值，或者一个 10 美分加上给原始价值减去 10 美分找零所需硬币数量的最小值，等等。所以，给原始总数找零的硬币数量可以根据下面的方法计算：

$$numCoins = min\begin{cases} 1 + numCoins(originalamount - 1) \\ 1 + numCoins(originalamount - 5) \\ 1 + numCoins(originalamount - 10) \\ 1 + numCoins(originalamount - 25) \end{cases}$$

```
def recMC(coinValueList,change):
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(coinValueList,change-i)
            if numCoins < minCoins:
                minCoins = numCoins
    return minCoins
print(recMC([1,5,10,25],63)
```

列表7中展示了我们刚才所描述的算法。在第3行，我们检查基本结束条件；也就是说，需要兑换的找零数等于我们硬币的某个面值。如果我们没有等于找零数目的硬币面额，那么我们就对每个小于我们找零总数的不同的硬币值调用递归。第6行展示了我们应该怎样通过使用一个硬币面值的列表，以帮助我们筛选出比当前找零价值小的硬币的列表。通过选定硬币的值，递归调用减小了我们需要找零的零钱总数。第7行展示了递归调用。注意，在同一行，我们要给硬币总数加1，这是因为我们使用了一枚硬币。只需加1就相当于：满足基本结束条件时，我们就做一次递归调用。

以上算法的问题就是它太低效了。事实上，它需要67716925次递归调用才能得出有4种硬币时找零63美分问题的最优解！为了理解我们的算法中的致命缺陷，观察图 4.3，它列出了我们为了找到兑换26美分的最优解时所需的377次函数调用中的一小部分。

图中每一个节点对应一次recMC函数调用。节点上的数字显示了我们要计算的需要找零的硬币总量。箭头上的数字则显示了我们使用的硬币的面额。顺着图形，我们可以看到图中任何点的硬币的组合。主要问题就是我们做了大量的重复计算。例如，该图显示，这种算法会重复计算为15美分找零的最优解至少三次。每一次这种计算都要调用52次函数。显然我们浪费了大量的时间和精力来重复计算旧的结果。

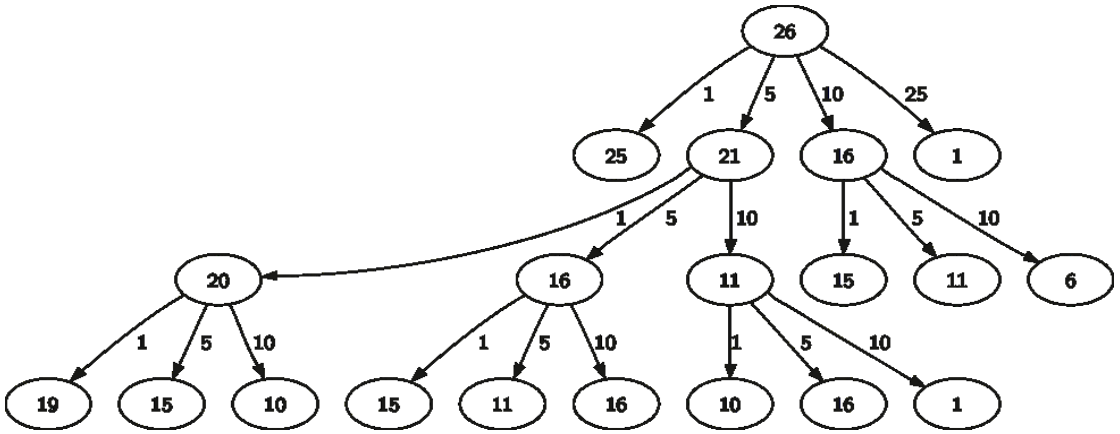


图4.3 Listing7的函数调用树

减少我们的工作量的关键在于记住一些出现过的结果，这样就能避免重复计算我们已经知道的结果。一个简单的解决方案就是我们将所找到的给硬币找零的最小数目存储在一个表中。然后在我们计算一个新的最小值之前，可以先查表看这个结果是否已知。如果表中已经有了这个结果，我们就可以从表中引用这个值而不是重复计算。有效编码1展示了包含了查表法的改善算法。

```
def recDC(coinValueList,change,knownResults):
    minCoins = change
    if change in coinValueList:
        knownResults[change] = 1
        return 1
    elif knownResults[change] > 0:
        return knownResults[change]
    else:
         for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recDC(coinValueList, change-i,
                                 knownResults)
            if numCoins < minCoins:
                minCoins = numCoins
                knownResults[change] = minCoins
    return minCoins
print(recDC([1,5,10,25],63,[0]*64))
```

有效编码1：查表法递归硬币计数(lst_change2)

注意，在第六行我们添加了一个测试来检查表中是否包含了为某个特定数目找零的硬币的最小值。如果没有，我们就调用递归来计算这个最小值并把它存储在表中。使用这个改进后的算法减少了我们得到了使用4种硬币找零63美分的问题答案，只需要221次递归函数调用！

虽然有效编码1的算法是正确的，但看起来感觉好像被黑客攻击过一样。此外，如果我们观察knowResults列表我们会发现，表中还有不少空洞。事实上，我们目前所采用的方法还不是动态规划，我们只是使用了一种叫做“函数值缓存”，或者一般称为“缓存”的方法改善了程序的性能。

真正的动态规划会采用更系统化的方法来解决问题。动态规划的解决方法是从为1分钱找零的最优解开始，逐步递加上去，直到我们需要的找零钱数。这就保证了在算法的每一步过程中，我们已经知道了兑换任何更小数值的零钱时所需的硬币数量的最小值。

让我们来看看我们如何在11美分找零时用最小数目的硬币使用量来填表。图4.4列出了过程。我们先从1美分开始。唯一可行的解决方案就是一个硬币（1美分）。下一行显示了为1美分和2美分找零的最小硬币数量。同样，唯一的答案是两个1美分硬币。在第5行，事情开始变得有趣了。现在我们考虑两种情况，五个1美分硬币或一个5美分硬币。我们如何决定那个才是做好的选择呢？通过查表我们可以看到，为4美分找零的数量是四个，再加一个1美分变成了5美分，相当于有五个硬币。或者我们可以考虑0个硬币加上一个5美分硬币等于5美分，只有一个硬币。由于1比5小，我们把1存储在列表中。很快到了表的结尾，该考虑11美分了。图5显示了我们必须考虑的三种情况：

1．一个1美分加上为 11-1=10美分找零的最小值（1）

2．一个5美分加上为 11-5=6美分找零的最小值（2）

3．一个10美分加上为 11-10=1美分找零的最小值（1）

情况1和3都给出了为11美分找零的最小值是2个硬币的答案。

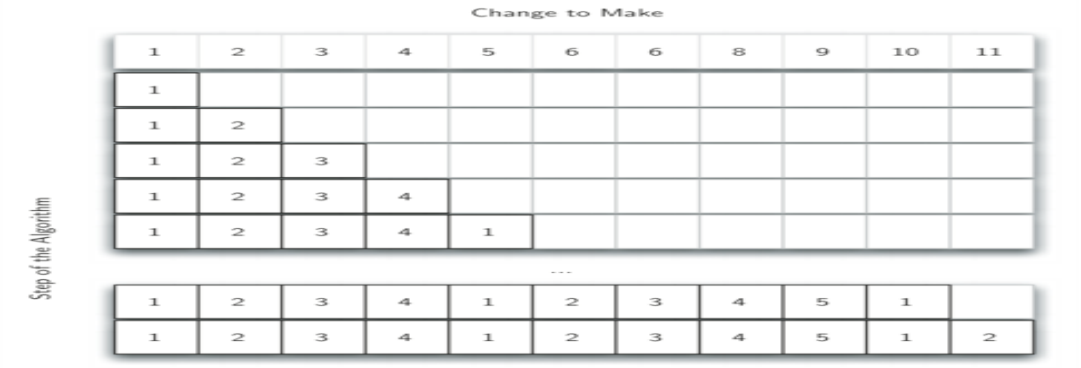


图4.4 需要找零的硬币数量的最小值

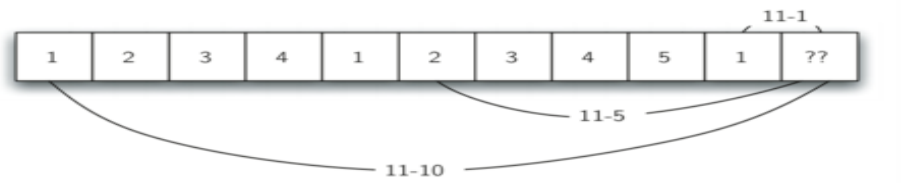


图4.5 考虑为11美分找零的最小值的3种方案

列表8是一个为解决找零问题所设计的动态规划算法。dpMakeChnge有三个参数：一个有效硬币面值的列表、我们想要兑换硬币的数值、一个包含所有部分找零最优解的列表。当函数运行完，minCoins会包含从0到所需兑换数值的每一个数值对应的最优解。

```
def dpMakeChange(coinValueList,change,minCoins):
    for cents in range(change+1):
        coinCount = cents
        for j in [ c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
                coinCount = minCoins[cents-j] + 1
        minCoins[cents] = coinCount
    return minCoins[change]
```

注意，dpMakeChange不是一个递归函数，即使我们开始使用了递归解决这个问题。必须要认识到的是，你可以写一个递归算法来解决问题，但这并不意味着它就是最好和最有效的解决方案。这个函数的大部分内容是做循环，循环从第4行开始。在这个循环中，我们要考虑使用所有可能的硬币面值来为cents中所指定的数值兑换硬币。像我们在上面举的给11分钱兑换硬币的例子，我们把部分找零的最优解记录下来并保存在minCoins列表中。

尽管我们的找零算法在找出所需硬币数量的最小值上做得很好，但是它并不能真的帮助我们兑换硬币，因为我们没有跟踪记录我们使用的硬币。我们可以很容易地扩展dpMakeChange来跟踪记录我们所使用的硬币，只要简单的记录我们为minCoins的每一项添加的最后一个硬币就可以了。如果我们知道了最后一个添加的硬币，就可以简单的减去这个硬币的币值来找到最优解列表中之前的一项进行找零。我们可以一直倒退访问列表直到回到列表的最开始。

有效编码2显示了改善了性能的dpMakeChnge算法，它能够跟踪硬币使用的路径，同时有一个printCoins的功能，通过重访列表，打印出每个使用过的硬币的值。这表明了这个算法可以解决我们在Lower Elbonia的朋友的问题。main的前两行设置了要转换的总量，并且创建了可使用的硬币面值的列表。接下来的两行创建了我们存储结果的列表。coinUsed 是一个我们用来找零的硬币的列表，coinCount是为表中相应的位置的量找零所需的硬币数量的最小值。

注意我们打印出的硬币值直接来自coinUsed阵列。第一次调用函数时，我们从阵列的63位置开始并打印出21。然后我们得到63-21=42并观察列表的第42级。我们又一次发现存储了21。最后，列表的21级也包含了21，最后我们得到了三个21分硬币。

```
def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
    for cents in range(change+1):
        coinCount = cents
        newCoin = 1
        for j in [c for c in coinValueList if c <= cents]:
                if minCoins[cents-j] + 1 < coinCount:
                    coinCount = minCoins[cents-j]+1
                    newCoin = j
        minCoins[cents] = coinCount
        coinsUsed[cents] = newCoin
    return minCoins[change]
```

```
def printCoins(coinUsed,change):
    coin = change
    while coin > 0:
        thisCoin = coinsUsed[coin]
        print(thisCoin)
        coin = coin - thisCoin
def main():
    amnt = 63
    clist = [1,5,10,21,25]
    coinsUsed = [0]*(amnt+1)
    coinCount = [0]*(amnt+1)
    print("Making change for",amnt,"requires")
    print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
    print("They are:")
    printCoins(coinsUsed,amnt)
    print("The used list is as follows:")
    print(coinsUsed)
main()
```

有效编码2：圆满的解决了找零问题(lst_dpremember)

4.8 小结

在本章我们看了几种递归算法的例子。这些算法向你展示了几种不同的可以用递归解决的问题，表明递归是一种解决问题的有效技术。本章的要点有以下内容：

- 所有递归算法必须具备基本结束条件。
- 递归算法必须要减小规模，改变状态，向基本结束条件演进。
- 递归算法必须调用自身（递归地）。
- 某些情况下，递归可以代替迭代循环。
- 递归算法通常能够跟所要解决的问题的表述很自然地契合。
- 递归不总是最合适的算法，有时候递归算法可能会引发巨量的重复计算。

4.9 关键词

基本结束	**条件解码（解密）**
动态规划递归	**递归调用**
栈帧	

4.10 问题讨论

1. 为汉诺塔问题设计一个调用栈。假设开始时有三个盘片。

2. 运用上文描述的递归规则，用纸和笔画一个谢尔宾斯基三角形。

3. 运用找零的动态规划算法，找出为33美分找零所需硬币的最少数量。除了通常的硬币面值以外，假设你还有面值为8美分的硬币。

4.11 词汇表

基本结束条件

在一个递归函数中，不会引起进一步的递归调用的条件语句的一个分支。

数据结构

一种数据的组织方案，可以使数据使用更加方便。

异常

程序运行时出现的错误。

异常处理

通过将一部分代码放入一个try-except结构来防止异常终止程序。

不可变数据类型

一种不可以改变的数据类型。元素的操作或不可变类型的切片操作会导致运行错误。

无限递归

一个函数不停地递归调用自己，无法达到基本结束条件。最终，一个无限递归导致运行出现错误。

可变数据类型

一种可以修改的数据类型。所有的可变类型都是复合类型。列表和字典（见下一章）是可变数据类型；字符串和元组不是。

raise

使用 raise 语句引发一个异常。

递归

调用已经执行的函数的过程。

递归调用

调用已经执行的函数的语句。递归也可以不是直接的——函数f可以调用g，g调用h，而h又调用回f。

递归定义

一个定义自己本身的定义。为了发挥作用，递归定义必须包含不是递归的基本结束条件，使它不同于循环定义。递归的定义通常可以提供一种优雅的方式来表达复杂的数据结构。

元组

包含一个由任何类型的元素组成的序列的数据类型，像一个列表，但是不可变。元组可以用于任何一个必须是不可变类型的情况，比如字典中的key（见下一章）。

元组赋值

使用一个赋值语句就可以给元组中的所有元素赋值。赋值是平行发生的，而不是顺序发生的，这对于两个元组之间数值的交换十分有用。

编程练习

1. 写一个递归函数来计算一个数的阶乘。

2. 写一个递归函数来反转一个列表。

3. 修改用递归来画树的程序，使用以下所提供的想法中的一种或几种:

● 树枝的粗细可以变化，随着树枝长度函数branchLen变化使树枝缩短，线条也相应变细。

● 树枝的颜色可以变化，当树枝长度函数branchLen变化从而使树枝非常短的时候，使树枝的颜色看起来像树叶的颜色。

● 让树枝倾斜角度在一定范围内随机变化，如15~45度之间，运行一下看看怎样比较好看。

● 树枝的长度函数branchLen可以变化，使之不用一直保持同样的数值，而是在一定范围内随机变化。

如果你实现了以上所有功能，你将会获得一棵非常逼真的树。

4. 找出或发明一种算法绘制分形山。提示：解决此问题的一种方法是再次利用三角形。

5. 写一个递归函数来计算斐波那契数列。递归函数的性能和利用迭代计算的方法来比较，效果如何？

6. 采取一种方法解决汉诺塔问题，利用三个栈来跟踪盘子的轨迹。

7. 使用海龟绘图模块，编写递归程序画出希尔伯特曲线。

8. 使用海龟绘图模块，编写递归程序画出科赫雪花。

9. 写一个程序来解决以下问题：你有两个水壶，一个4加仑的水壶和一个3加仑的水壶，两个水壶上都没有标记，有一个泵可以用来往水壶中加水。如何做能在4加仑的水壶中恰好得到2加仑的水？

10. 总结上题，使得你的解决方案的参数包括每个水壶的大小，和最后大壶中所剩的水量。

11. 写一个程序，解决以下问题：三个传教士和三个食人族来到河边，河边只有一艘船，每次可以载两个人，每个人都必须要过河来继续旅程。然而，如果河岸上的食人族人数超过传教士人数，传教士将被吃掉。找出每次的过河方案使每个人都安全地到达河的另一边。

12. 用海龟绘图模块，将汉诺塔问题的解决过程做成动画。提示：你可以改变海龟的形状，可以将海龟的形状改为方块的盘子。

13. Pascal三角形是一个由数字组成的三角形，这些数字以如下方式在每一行交错排列：

$$a_{nr} = \frac{n!}{r!\,(n-r)!}$$

这个方程是一个二项式系数的方程。可以通过添加数字来建立Pascal三角形，每行的数字都是由上一行的对角线数字相加而得。Pascal三角形举例如下：

1 1

1 2 1

1 3 3 1

1 4 6 4 1

写一个程序输出Pascal三角形。程序里应接受一个参数来定义三角形的行数。

14. 假设你是一个计算机科学家或是一个艺术小偷，闯入了一个艺术画廊。你身上只有一个背包可以用来偷出宝物，这个背包只能装W英镑的艺术品，但你知道每一件艺术品的价值和它的重量。运用动态规划写一个函数，来帮助你获得最多价值的宝物。你可以利用下面的例子来编写程序：假设你的背包可以容纳的总重量为20，你有如下5件宝物：

item	weight	value
1	2	3
2	3	4
3	4	8
4	5	8
5	9	10

15. 这个问题叫做单词最小编辑距离问题，在很多领域的研究中起到了很大作用。假设你想把单词“algorithm”变为“alligator”。对于每一个字母，你有三种变换方式：从源单词复制一个字母到目标单词，计5分；从源单词删除一个字母，计20分；在目标单词插入一个字母，计20分。最后将一个单词转换为另一个的分数可以被拼写检查系统使用，用来给彼此相似的单词提供建议。使用动态规划技术，写一种算法得到任何两个单词之间的最小编辑距离。

5. 排序与搜索

5.1.目标

- 了解和实现顺序搜索和二分法搜索。
- 了解和实现选择排序、冒泡排序、归并排序、快速排序、插入排序和希尔排序。
- 了解用散列法实现搜索的技术。
- 了解抽象数据类型：映射Map。
- 采用散列实现抽象数据类型Map。

5.2.搜索

现在我们将会把我们的注意力放在搜索和排序上，它们是计算中所出现的最常见的问题。在本节中,我们将学习搜索。排序的问题我们将在之后的章节学习。搜索的算法过程就是在一些项的集合中找到一个特定的项。搜索过程通常会根据特定项是否存在来给出回答True或者False。有时它可能会返回特定项所出现的位置。我们在此的目的，就只是关心集合成分的问题。

在Python中,有一个非常简单的方法来判别特定项是否在列表中。我们使用in这个运算符。

```
>>> 15 in [3,5,2,4,1]
False
>>> 3 in [3,5,2,4,1]
True
>>>
```

尽管我们很容易写出这个代码，但为了回答这个问题必须执行一个潜在的程序。事实证明,有很多不同的方法来寻找一个特定项。我们感兴趣的是这些算法的工作原理以及它们互相比较时的优劣之分。

5.2.1.顺序搜索

当数据项被存储在集合中时，如存储在一个列表中，我们说，它们有一个线性或顺序的关系。 每一个数据项存储在一个与其他数据项相对的位置。在Python列表，这些相对位置所对应的是单个项的索引值。由于这些索引值是有一定次序的，可以依次访问它们。这一过程产生了第一个搜索方法，**顺序搜索**。

图1显示了顺序搜索的工作原理。从列表中的第一项开始，我们按照初始顺序从一项移到下一项，直到我们发现正在寻找的数据项或者遍历所有数据项。如果我们遍历了所有数据项，我们就会发现我们正在寻找的数据项是不存在的。

图5.1 整数列表的顺序搜索

这个Python算法的实现如CodeLens 1所示。函数的参数为一个列表和一个我们正在寻找的数据项，并且返回一个布尔值，判断它是否存在。布尔变量found初始化为False，如果我们发现我们所要找的数据项在列表中，就将布尔变量赋为True。

```
def sequentialSearch(alist, item):
    pos = 0
    found = False
    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos + 1
    return found
testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequentialSearch(testlist, 3))
print(sequentialSearch(testlist, 13))
```

CodeLens：一个无序表的顺序排序 （搜索1)

5.2.1.1.排序算法的分析

分析搜索算法，我们需要确定一个基本单元的计算。回想一下，为了解决这一问题，我们必须重复执行一些简单的步骤。对于搜索，计算比对执行次数是有必要的。每次对比都有可能发现我们正在寻找的数据项，也有可能没发现。此外，我们再作一个猜想。数据列表无论如何不能保证是有序的。数据项是被随机放置到列表中。换句话说，我们寻找的目标项可能在任意位置，对列表的每一个位置，我们找到它的概率是相等的。

如果目标项不在列表内，唯一的办法就是比对现有的每一个数据项。如果这里有n个数据项，那么顺序搜索就要求n次比对去发现目标项不在列表里面。在列表包含目标项的情况下，算法分析就不是这么的直接。这儿通常有三种不同的情况发生。最好的情况就是我们要找的数据项就在我们第一次比对的位置(列表的开头)。我们仅仅需要一次比对。最坏的情况是直到最后一次比对（第n次比对），我们才能发现目标项。

平均情况怎么样呢？在平均情况下，我们大约会在列表中央发现目标项，就是说，我们需要比对n/2的数据项。然而，随着n较大，不管系数是什么，在我们的近似中都变得无关紧要了，所以顺序搜索的复杂度是O(n)。表1总结了以上所说的这些可能的结果。

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

表5.1无序表顺序搜索的比对

我们早先假定我们集合里面的数据项都是被打乱放入的，这样，数据之间便没有相对关系。如果在某些情况下，数据项是有序的，顺序搜索又会发生什么？我们能让我们的搜索技术更高效吗？

假定列表是按照一个递增的顺序构建的(从小到大)。如果我们要找的数据项在列表里面，它在n个位置中的任意位置的机会如以前一样是相等的。我们依然将经过相同的比较次序去发现目

标项。然而，如果数据项不在列表里面，将会有一个明显的优势。图2就展示了搜索50这个数据项的算法流程。注意，在到54之前一直是执行比对操作的。然而，在这一点上，我们知道了一些另外的东西。因为这个列表是被排好序的，所以在54之前找不到，那么比54大的数据项肯定也不能满足条件。在这种情况下，算法并没有要求继续去遍历所有数据项来表明目标项没有被找到。它可以立刻停止。CodeLens 2展示了经过此次变化后的顺序搜索。

图5.2 一个整数有序表的顺序搜索

```
def orderedSequentialSearch(alist, item):
    pos = 0
    found = False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
        else:
            if alist[pos] > item:
                stop = True
            else:
                pos = pos + 1
    return found
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(orderedSequentialSearch(testlist, 3))
print(orderedSequentialSearch(testlist, 13))
```

CodeLens：一个有序列表的顺序搜索（搜索2）

表2总结了以上所说的这些可能的结果。注意，当我们可以发现项不在列表时，最好结果只需要看一个数据项。平均一下，通过遍历n/2的数据项，我们将知道目标项是否在列表中。

item is present	1	n	$\frac{n}{2}$
item is not present	1	n	$\frac{n}{2}$

表5.2 有序列表顺序搜索的比较

自我测试

1：假定你在做列表[15, 18, 2, 19, 18, 0, 8, 14, 19, 14]的一个顺序搜索，为了找到18，你需要做多少次比对？

a) 5　　b) 10　　c) 4　　d) 2

2：假定你在做有序列表[3, 5, 6, 8, 11, 12, 14, 15, 17, 18]的一个顺序搜索，为了找到13，你需要做多少次比对？

a) 10　　b) 5　　c) 7　　d) 6

5.2.2.二分法搜索

如果所采用的比较方法更聪明一些，我们可以更好地利用有序表的优势。在顺序搜索中，当我们和第一项相比较时，如果第一个数据项不是我们要找的项，最多还有n-1项待比对。**二分搜索**将从中间项开始检测，而不是按顺序搜索列表。如果查找项与我们刚搜索到的项匹配，则搜索结束。如果不匹配，我们可以利用列表的有序性来排除掉一半的剩余项。如果查找项比中间项大，我们可以把列表中较小的那一半全部和中间项可以从接下来的考察中排除了。因为如果查找项在列表中，那它一定在较大的那一半。

接下来，我们可以在较大的一半中重复这个过程。从中间项开始，拿它和查找项作比较。再来一次，我们要么找到了查找项，要么从中间分割列表，并因此排除掉另一大部分我们的搜索区域。图3展示了这种算法如何快速找到值为54的项。完整的过程在CodeLens 3中展示了出来。

图5.3 二分搜索一个整数有序表

```
def binarySearch(alist, item):
    first = 0
    last = len(alist) – 1
    found = False
    while first <= last and not found:
        midpoint = ( first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint -1
            else:
                first = midpoint +1
    return found
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(orderedSequentialSearch(testlist, 3))
print(orderedSequentialSearch(testlist, 13))
```

CodeLens：二分搜索一个有序表（搜索 3）

在我们继续分析之前，我们应该了解到这个算法是分而治之策略的一个很好的例子。分而治之意味着我们把一个问题分成更小的规模，用一些方法解决这些更小规模问题，然后重组整个问题来得到结果。当我们对一个列表执行二分搜索时，我们首先选择中间项。如果搜索项比中间项

小，我们可以在原来列表的左半部分执行二分搜索，同样地，如果搜索项更大，我们可以在右半部分执行二分搜索。无论怎样，这是一个对较小的列表实现二分搜索的递归调用。CodeLen 4 展示了这种递归。

```
def binarySearch(alist, item):
    if  len(alist) == 0 :
        return  False
    else:
        midpoint = len(alist)//2
        if  alist[midpoint] == item:
            return True
        else:
            if  item < alist[midpoint]:
                return binarySearch (alist[:midpoint],item)
            else:
                return binarySearch (alist[midpoint+1:],item)
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

CodeLens 4：二分搜索——递归版本（搜索4）

5.2.2.1. 二分搜索分析

为了分析二分搜索算法，我们需要回顾每一个从考察中对比、排除的约一半的待搜索项。这种算法遍历整个列表所需的比对次数最多是多少呢？如果我们开始有n项，大约有 n/2 项将在第一次比对之后被遗弃。在第二次比对之后，将会有约n/4，然后是n/8，n/16，等等。我们能分割列表多少次呢？我们在表3中看到了答案。

项数	分割次数
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
……	……
I	$\frac{n}{2^i}$

表 3：二分搜索的表格式分析

当我们分割的次数足够多时，我们结束于一个只有一项的列表，无论它是不是我们要找的那一项，总之，我们完成了遍历。比对的次数需要通过解方程**n/2^i=1**得到。解得**i=log（n）**。最大比对次数是关于列表中项数的对数。因此，二分搜索的复杂度是**O(log(n))**。

我们还需要处理一个在分析中所附加的消耗。在上面的递归处理演示中，递归调用 binarySearch(alist[:midpoint],item) 使用了切片操作符来创建左半部分列表，然后传递到下一个调用（右半部分也一样）。我们上面所做的分析假定切片操作消耗的是常数时间。然而，我们知道Python的切片操作实际上是O(k)。这意味着二分搜索使用切片将不会运行严格的对数时间。幸运的是这可以通过列表的开始和结束索引值补救。这个索引可以像我们在表3中精心设计的那样。我们将这个实现的操作留作练习。

即使二分搜索通常比顺序搜索要好，值得注意的是，对于较小的n值，排序所附加的消耗可能是不值得的。事实上，我们应当一直考虑进行额外的排序工作来得到搜索优势是否是有效开销。如果我们可以排序一次然后搜索许多次，排序开销并不那么显著。然而，对于大列表，哪怕是一次排序的消耗也可能是巨大的，从一开始简单执行顺序搜索也许是最好的选择。

自我检测

Q-40：假如你有以下已排好序的列表[3,5,6,8,11,12,14,15,17,18]，用二分搜索算法进行搜索。为了找到项8，下列哪组数反映了正确的对比顺序？

a) 11,5,6,8

b) 12,6,11,8

c) 3,5,6,8

d) 18,12,6,8

Q-41：假如你有以下已排好序的列表[3,5,6,8,11,12,14,15,17,18]，用二分搜索算法进行搜索。为了找到项16，下列哪组数反映了正确的对比顺序？

a) 11,14,17

b) 18,17,15

c) 14,17,15

d) 12,17,15

5.2.3. 散列

在之前的章节中，我们利用储存在一个集合中的数据之间的相对位置关系，实现了对搜索算法效率的提高。例如，已知一个列表是一个有序表，那么我们就可以利用二分法实现数据搜索，将算法的时间复杂度控制在对数级别上。在这一节中，我们将尝试进一步建立一种新的数据结构，基于它的搜索算法的时间复杂度为**O(1)**。这个概念被称为**散列**。

为了实现这一数据结构，当我们尝试去寻找某一数据时，我们需要知道更多关于这一数据项可能在哪些位置出现的信息。如果所有的数据项都在恰当的位置上，那么我们就可以利用对应关系到那个位置上看看该数据项是否存在。然而，情况通常都不会这么简单。

散列表是一种数据的集合，其中的每个数据都通过某种特定的方式进行存储以方面日后的查找。散列表的每一个位置叫做**槽**，能够存放一个数据项，并以从0开始递增的整数命名。例如，第一个槽记为0，下一个记为1，再下一个记为2，并以此类推。在初始条件下，散列表中是没有任何数据的，即每个槽都是空的。我们可以利用列表实现一个散列表，它的每一个元素都被初始化为None。图4展示了一个长度**m=11**的散列表，换言之，这个散列表中有m个槽，它们被依次命名为0到10。

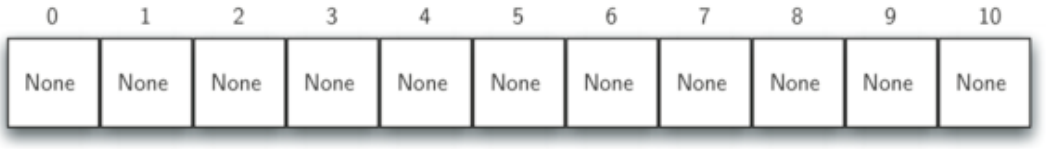


图4： 有11个空槽的散列表

某个数据项与在散列表中存储它的槽之间的映射叫做**散列函数**。散列函数可以将任意一个数据项存储到集合中并返回一个介于槽命名区间内的，即0与m-1之间的整数。假设我们有一列整数54、26、93、17、77、31。我们的第一个散列函数，有时被称为“求余”，简单地将要存储的数据项与散列表的大小相除，返回余数作为这个数据项的散列值**(h(item)=item%11)**。表4给出了上面例子中所有数据项的散列值。值得注意的是，为了保证求得的散列值落在散列表的大小内，这个求余得到散列值的方法将广泛地运用于所有种类的散列函数中。

数据项	散列值
54	10
26	4
93	5
17	6
77	0
31	9

表格 4 ：利用求余方法的简单散列函数

一旦求出了散列值，我们就可以将每一个数据项插入到散列表中制定的位置中，如图 5 所示。另外我们还可以注意到 11 个槽中的 6 个被占据。一般地，我们把槽被占据的比例叫做**负载因子**。在这个例子中，负载因子$\boldsymbol{\lambda = \frac{\text{数据项个数}}{\text{散列表的大小}}}$。在这个例子中，$\boldsymbol{\lambda = \frac{6}{11}}$

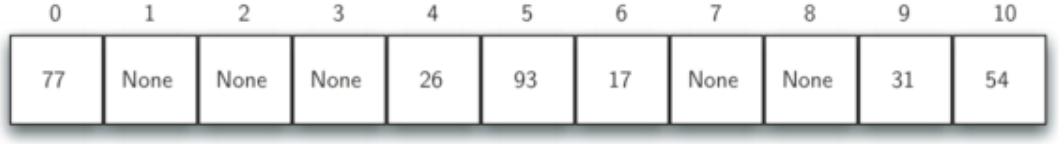


图5：填入了6个数据项的散列表

现在当我们想要查找一个数据项时，我们只需要使用散列函数去计算得到这个数据项对应槽的名字并在这个槽中检查该数据项是否存在即可。这个搜索过程的时间复杂度为**O（1）**，因为通常计算出散列值和得到散列表在该位置的索引需要一定的时间。如果所有数据项都在散列函数所规定的位置上，我们就已经得到了一个时间复杂度为常数量级的搜索算法。

但是你可能已经看出了这个方法的问题所在，它仅能在每一个数据项在散列表中占有不同的槽的情况下才能正常运作。例如，如果我们在散列表中插入一个新的数据项44，它的散列值为0（**44 % 11 = 0**）。但是由于77也拥有相同的散列值0，问题也就随之产生。根据这种求余的散列函数，两个甚至多个数据就需要存储在同一个槽中。这种情况被称为**冲突**。显然，冲突导致散列方法出现了问题。我们将在后面的章节讨论这些问题。

5.2.3.1. 散列函数

对于一组给定的数据项，如果一个散列函数可以将每一个数据项都映射到不同的槽中，那么这样的散列函数叫做**完美散列函数**。如果已知数据项是固定不变的，我们就可能构造一个完美散列函数（你可以参考练习来获得更多有关完美散列函数的知识）。但是，如果给定的是任意的一组数据项，那么就没有一种系统化的方法来构造一个完美的散列函数。幸运的是，我们并不需要一个绝对的完美散列函数，不完美的散列函数一样可以在实用中有优秀的表现。

一种实现完美散列函数的方法就是扩大散列表的尺寸，直到所有可能的数据项变化范围都被散列表所包含。这一方法保证了每一个数据项都有一个不同的槽。尽管这个方法对于少量数据是可行的，但是当它面对大量可能的数据项时显得捉襟见肘。例如，如果存储的数据项是9位的社会安全码，这个方法就需要差不多10亿个槽，如果我们只想保存班里25个同学的数据，我们就会浪费大量的存储空间。

我们的目标是创建一个能够将冲突的数量降到最小，计算方便，并且最终将数据项分配到散列表中的散列函数。这里有几种普通的方法去扩展求余方法，我们将在这里考察其中几个。

折叠法创建散列函数的基本步骤是：首先将数据分成相同长度的片段（最后一个片段长度可能不等）。接着将这些片段相加，再求余得到其散列值。例如，如果我们有一串电话号码436-555-4601，我们可以两个一组将这个号码分成5段（43,64,55,46,01）。然后相加得到。如果我们假设散列表有11个槽，我们就需要将和进行求余。因为，所以电话号码436-555-4601就存放在槽1中。有的折叠法还包含一个每跳过一个数就将下一个数反转再相加的过程。在上面的例子中，我们得到，求余得到$\mathbf{219\ \%\ 11 = 10}$。

另一个创建散列函数的数值方法叫做**平方取中法**。我们首先将数据取平方，然后取平方数的某一部分。例如数据项是44，我们首先计算。接着取出中间的两位数93，然后再进行求余运算，最终得到表5展示了这些数据项在求余法和平方取中法中得到的结果。你应该确认你明白它们是如何计算得到的。

数据项	求余法结果	平方取中法结果
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

表格5：求余法和平方取中法的比较

我们也可以为非数字的数据项，例如字符串创建散列表，’cat’ 可以看做一个连续的ASCII数值。

```
>>> ord('c')
99
>>> ord('a')
97
>>> ord('t')
116
```

然后我们可以将三个数值加起来，接着用求余法得到一个散列值（如图6）。表1展示了一个散列函数，用来产生一个字符串和一个表的大小，并返回一个0到表大小-1的散列值。

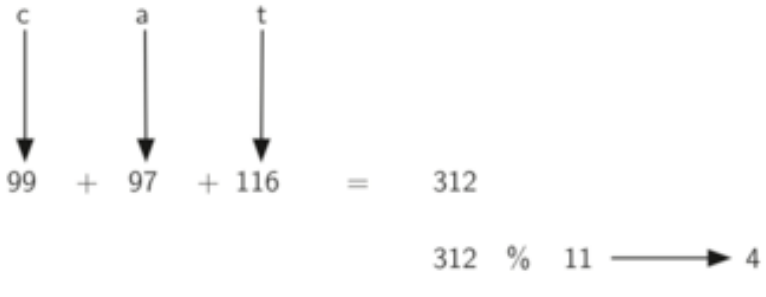


图6：用ASCII数值散列一个字符串

```
def hash(astring, tablesize):
    sum = 0
    for pos in range(len(astring)):
        sum = sum + ord(astring[pos])
    return sum%tablesize
```

表1

有趣的是，当我们用散列函数纪录散列值的时候，颠倒的字母构成的单词会得到相同的散列值。为了纠正这一情况，我们可以将字母的位置作为权重因子。图7展示了一种可能利用权重因子的方式。我们把调整这一散列函数的过程留作练习。

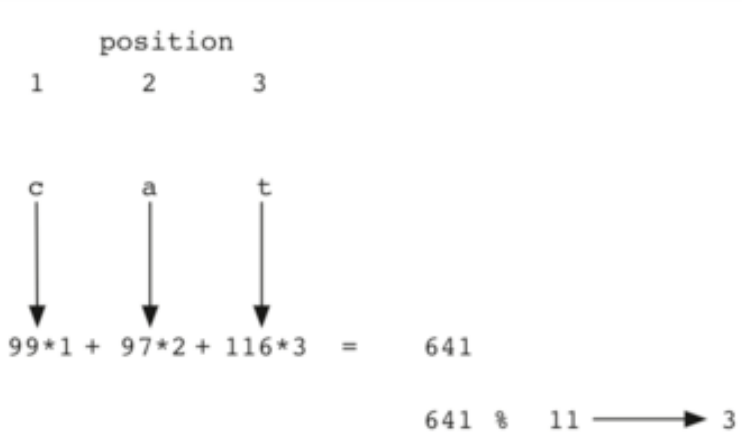


图7：用权重因子对一个字符串求散列值

你们还可以想出其他的一些方法去计算数据的散列值。但是最重要的事情就是散列函数必须足够高效以防止它成为占据存储空间和搜索进程的主要部分。如果散列函数过于复杂，导致花费大量的时间去计算槽的名称，可能还不如进行简单的顺序搜索或者二分法搜索，这就失去了散列的意义。

5.2.3.2.冲突解决方法

我们现在回到冲突问题。当两个数据散列到相同的槽，我们必须用一种系统化的方法将第二个数据放到散列表里。这个过程叫做**冲突解决**。正如我们前面提到的，如果散列函数是完美的，冲突就不会出现。但是，既然这一般来说是不可能的，冲突解决就成为实现散列很重要的一部分。

一种解决冲突的方法就是搜索散列表并寻找另一个空的槽来存放这个有冲突的数据。一种简单的方法就是从发生冲突的位置开始顺序向下开始寻找，直到我们遇到第一个空的槽。注意到我们可能需要回到第一个槽（循环）来实现覆盖整个散列表。这种冲突解决方法叫做**开放地址**，它试图在散列表中去寻找下一个空的槽。通过系统地向后搜索每一个槽，我们将这种实现开放地址的技术叫做**线性探测**。

图8展示了通过简单的求余散列函数对一个扩展的数据集合进行处理后的结果（ 54 、26 、93 、17 、77 、31 、44 、55 、20 ）。上面的表格4展示了原始数据的散列值。图5展示了原始散列表中存储的数据项。当我们尝试把44放到槽0的时候，冲突出现了。通过线性探测，我们一个接一个槽地检查，直到我们找到一个空位置。在这里，我们找到了槽1。

同样的，55本应该放在槽0，但是这里必须被放到槽2，因为它是接下来的第一个空槽。最后一个数据项是20，本应该放到槽9。但是由于槽9是满的，于是我们开始线性探测，我们先后查看了槽10,0,1,和2，最终我们找到了空槽3。

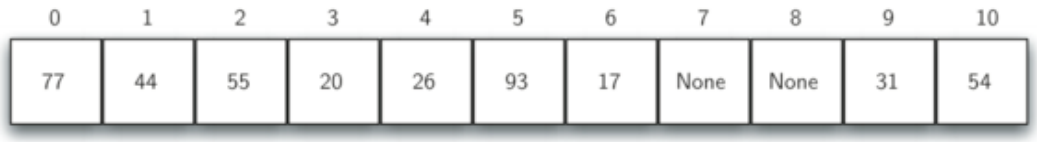


图8：利用线性探测解决冲突

一旦我们利用开放地址和线性探测建立一个散列表，最重要的问题就是我们要利用相同的方法去搜索数据。假设我们寻找数据93，先计算其散列值得到5，接着查看槽5，发现是93，所以我们返回真。那如果我们寻找20呢？现在散列值是9，但是槽9现在存放的是31。这时我们不能简单地返回假，因为我们知道这可能存在冲突。现在我们需要进行线性搜索，从槽10开始，直到我们找到数据20或者找到一个空槽。

线性探测法的一个缺点是产生**集中**的趋势：数据会在表中聚集。这意味着如果对于同一散列值产生了许多冲突时，周边的一系列槽都将会被线性探测填充。这将会对正在被填入的新数据产生影响，就像我们之前看到的，将数字20加入散列表中一样。当一簇值加入到散列表中槽0时必须跳过原槽直到找到新的空槽。这种集中正如图9所示。

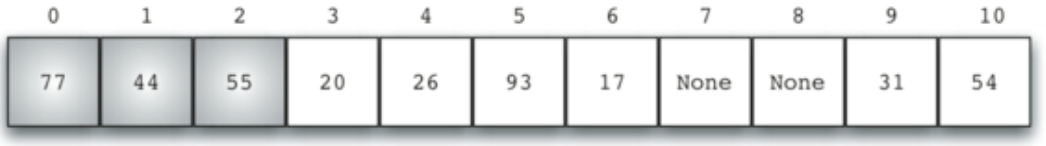


图9：一簇对应槽0的数据

一种解决集中问题的方法是扩展线性探测技术，我们不再按顺序一个一个地寻找新槽，而是跳过一些槽，这样能更加平均地分配出现冲突的数据，进而潜在地减少集中问题出现的次数。图10展示了同样的数据项如何通过“+3”线性探测的方法解决冲突的。“+3”表示一旦一个冲突出现，我们将每次跳过两个槽来寻找下一个新的空槽。

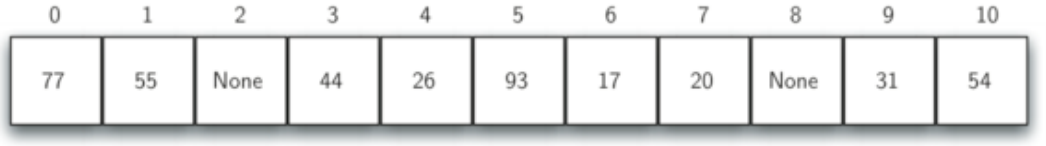


图10：用“+3”法解决冲突

这种冲突时为数据寻找新槽的进程被称作**再散列**。再散列函数的形式是**newhashvalue=rehash(oldhashvalue)**，对于简单的线性探测，它的具体形式是**rehash(pos)=(pos+1)%sizeoftable**。对于“+3”形式的再散列，它的具体形式是**rehash(pos)=(pos+3)%sizeoftable**。归纳起来的形式是**rehash(pos)=(pos+skip)%sizeoftable**。需要注意的一点是，选择跳过的槽的个数必须保证所有槽最终都能被遍历。否则，有些槽将会被闲置。为了保证这一点，我们通常建议将槽的数目设置成质数，这也是我们在例子里设置11个槽的原因。

另一种线性探测方法叫做**二次探测法**。我们不是每次在冲突中选择跳过固定个数的槽，而是使用一个再散列函数使每次跳过槽的数量会依次增加1,3,5,7,9，以此类推。这意味着如果原槽为第h个，那么再散列时访问的槽为第**h+1**，**h+4**，**h+9**，**h+16**个，以此类推。换言之，二次探测法使用一个连续的完全平方数数列作为它的跳跃值。图11显示了我们的例子在运用二次探测法时的填充结果。

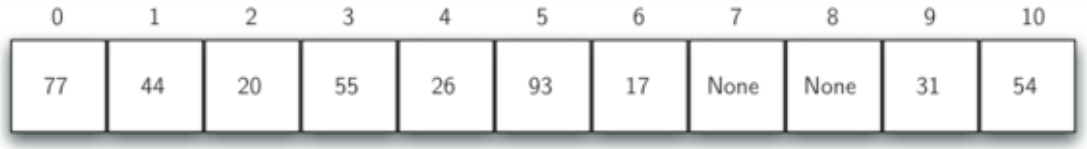


图11：利用二次探测法解决冲突

另一个解决冲突的替代方法是允许每一个槽都能填充一串而不是一个数据（称作链）。**链**能允许多个数据填在散列表中的同一个位置上。当冲突发生时，数据还是填在本应该位于的槽中。随着一个槽中填入的数据的增多，搜索的难度也就随之增加。图12显示了数据在用数据链方法填入散列表的结果。

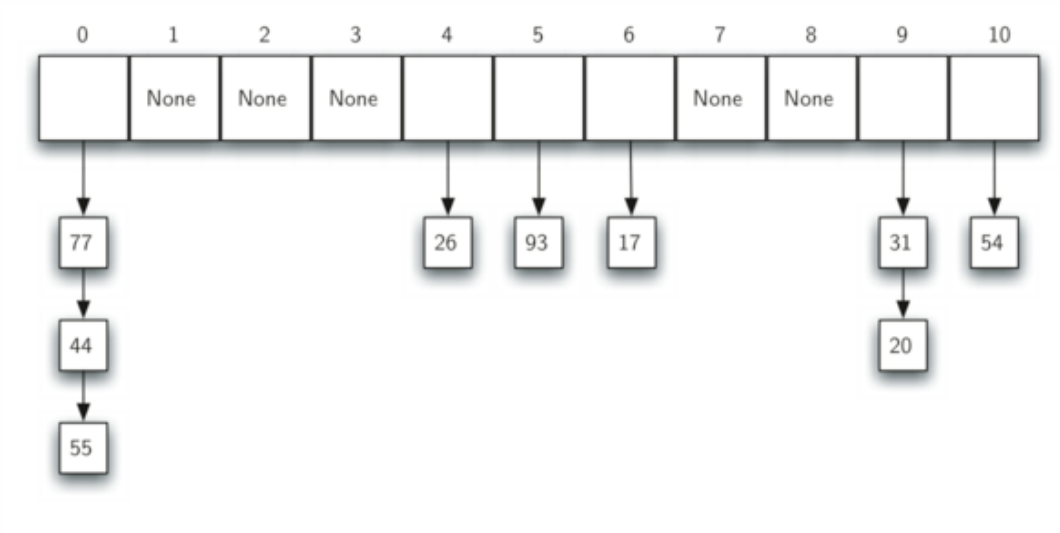


图12：用数据链方法解决冲突

自我检测

Q-17 ：对于一个有13个槽的散列表，27和130将会被填入第几个槽中？

a）1, 10

b）13, 0

c）1, 0

d）2, 3

Q-18 ：假设你将下列11个数据根据线性探测方法填入散列表中，哪一个选项最好地表达了填入之后散列表的状况？113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99

a) 100, __, 113, 114, 105, 116, 117, 97, 108, 99

b) 99, 100, __, 113, 114, 116, 117, 105, 97, 108

c) 100, 113, 117, 97, 14, 108, 116, 105, 99, __, __

d) 117, 114, 108, 116, 105, 99, __, 97, 100, 113

5.2.3.3.实现映射的抽象数据类型

字典是Python中最有用的数据类型之一。字典是一个可以储存密钥-数据对的关联数据类型。密钥是用来查找和它相关联的数据值的。我们通常把这个想法称作**映射**。

映射的抽象数据类型定义如下：它以一个密钥与一个数据值之间关联的无序集合作为结构。映射中的密钥都是独特的，以保证和数据值之间的一一对应关系。映射有以下的相关操作：

- `Map()` 产生一个新的空映射，返回一个空映射的集合。
- `Put(key,val)` 往映射中添加一个新的密钥-数据值对。如果密钥已经存在，那么将旧的数据值置换为新的。
- `get(key)` 给定一个 key 值，返回关联的数据，若不存在，返回`None` 。
- `del` 从映射中删除一个密钥-数据值对，声明形式为 `del map[key]`
- `len()` 返回映射中的存储密钥-数据值对的个数

- in 当表述是key in map，返回 True否则返回 False

字典的一个巨大的好处在于给定密钥时，我们能够迅速的找到与其关联的数据值。为了使得这种快速搜索得以实现，我们需要一种高效的搜索方法。我们可以使用一个顺序列表或者使用二分法搜索，但是更好地方法是使用散列表的方式。正如之前所描述的，在一个散列表中搜索数据的时间复杂度在**O（1）**级别。

在表2中我们运用两个列表来创造一个散列表类（HashTable class）来实现映射的数据结构类型。其中一个称为slots（槽），用来存储密钥，另一个平行列表称作data，用来存储数据值。当我们查找一个密钥时，对应的data列表中的位置保存着对应的数据值。依照之前的想法，我们把密钥表当作一个散列表来处理。注意到散列表的初始大小为11。尽管散列表的槽数的选择是任意的，但是将其定为一个质数还是很有意义的，这样能够使解决冲突问题的算法尽可能发挥最大作用。

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
```

表2

散列函数运用的是简单的求余方法。这里的冲突解决技术是运用“+1”的线性探测。其中put函数（见表3）假设最终一定能找到一个能让新的密钥填入的槽，除非它已经在self.slots中存在。基于这样的假设，它能够计算出最初的散列值，如果发现对应的槽不为空时，调用再散列（rehash）函数直到找到空槽位置。如果一个非空的槽已经含有该密钥，那么就将其数据值替换为当前数据值。

```
def put(self,key,data):
  hashvalue = self.hashfunction(key,len(self.slots))
if self.slots[hashvalue] == None:
    self.slots[hashvalue] = key
    self.data[hashvalue] = data
  else:
    if self.slots[hashvalue] == key:
      self.data[hashvalue] = data  #replace
    else:
      nextslot = self.rehash(hashvalue,len(self.slots))
      while self.slots[nextslot] != None and \
                        self.slots[nextslot] != key:
        nextslot = self.rehash(nextslot,len(self.slots))
      if self.slots[nextslot] == None:
        self.slots[nextslot]=key
        self.data[nextslot]=data
      else:
        self.data[nextslot] = data #replace
def hashfunction(self,key,size):
     return key%size
def rehash(self,oldhash,size):
    return (oldhash+1)%size
```

表

相似的，get函数（见表4）也是首先计算最初的散列值。如果结果不在对应的槽中，再散列（reshash）函数就会被用来确定一下个可能存储该密钥的位置。注意到第15确保了我们没有再

次回到原槽，保证了搜索操作不会陷入死循环。如果这种情况发生了，那么我们已经遍历所有可能的槽，这个密钥一定是不存在的。

散列表类（HashTable class）的最后一些操作提供了额外的字典类功能。我们重载了运算符__getitem__和__setitem__允许使用“[]”对字典进行访问。这表示一旦一个散列表被建立，我们所熟悉的索引操作符都将是可用的。我们把剩下的方法留作练习。

```
def get (self,key):
    startslot = self.hashfunction(key,len(self.slots))

    data = None
    stop = False
    found = False
    position = startslot
    while self.slots[position] != None and not found and not stop:
        if self.slots[position] == key:
            found = True
            data = self.data[position]
        else:
            position = self.rehash(position,len(self.slots))
            if position = startslot:
                stop = True
    return data

def __getitem__(self,key):
    return self.get(key)
```

代码 4.4

下面的部分展示了散列表类（HashTable class）的操作。首先我们建立一个散列表并保存一些包含整数密钥和字符串数据值的数据。

```
>>> H = Hashtable()
>>>H[54] = “cat”
>>>H[26] = “dog”
```

接下来我们将访问并修改一些数据。注意密钥20对应的数据值被替换了。

```
>>>H[20]
'chicken'
>>>H[17]
'tiger'
>>>H[20] = 'duck'
>>>H[20]
'duck'
>>>H.data
[77,44,55,20,26,93,17,None,None,31,54]
>>>H.data
['bird','goat','pig','duck','dog','lion','tiger',None,None,'cow','cat']
>> print(H[99])
```

据而必须的比对次数。

在分析散列表的使用情况时最重要的因素是负载因子。从概念上来看，如果较小，那么发生冲突的概率就较低，这意味着数据项有更大的可能填充在它们本该处于的位置上。如果较大，这意味着整个散列表接近于被填满，紧接着会造成越来越多的冲突。冲突的解决也会变得越来越困难，需要越来越多的比对操作来找到一个空槽。如果通过数据链方法解决冲突，逐渐增多的冲突预示着在每条链上会存储越来越多的数据项。

和先前一样，我们将会有一个成功的和一个不成功的搜索结果。对于一个利用开放地址与线性探测的成功搜索，平均比对操作次数大约是 $\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$，而一个相同方法的不成功的例子平均比对操作次数可能会达到$\frac{1}{2}\left(1+\left(\frac{1}{1-\lambda}\right)^2\right)$。如果我们使用数据链方法，成功例子的平均比对次数是$1+\frac{\lambda}{2}$，而在最失败的情况下，将会进行λ次比对。

5.3.排序

5.3.1.冒泡排序

冒泡排序要对一个列表多次重复遍历。它要比较相邻的两项，并且交换顺序排错的项。每对列表实行一次遍历，就有一个最大项排在了正确的位置。大体上讲，列表的每一个数据项都会在其相应的位置“冒泡”。

它们的顺序是否正确。如果列表有n项，第一次遍历就要比较n-1对数据。需要注意，一旦列表中最大（按照规定的原则定义大小）的数据是所比较的数据对中的一个，它就会沿着列表一直后移，直到这次遍历结束。

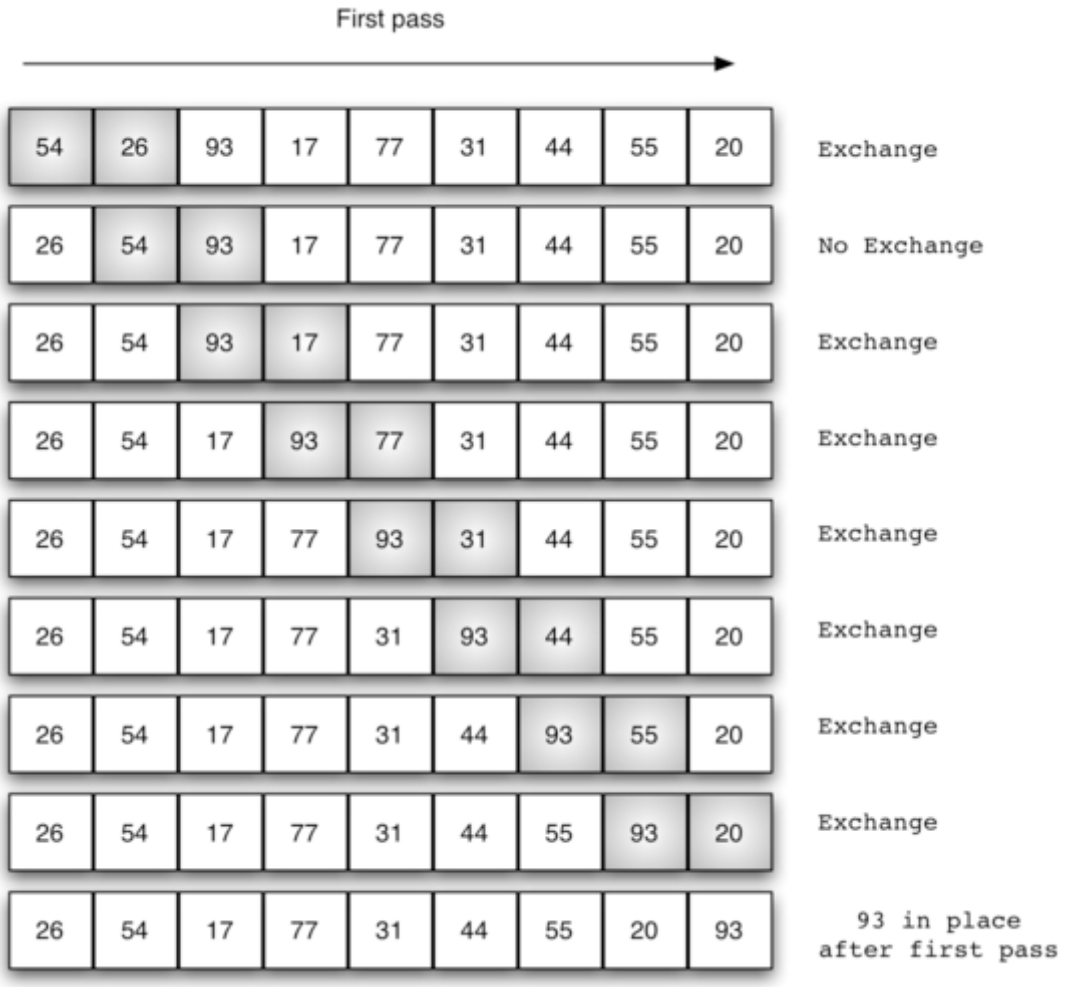


图5.1　冒泡排序：第一次遍历

第二次遍历开始时，最大的数据项已经归位。现在还剩n-1个待排数据项，即有n-2个要比较的数据对。由于每一次遍历都会使下一个最大项归位，所需要遍历的总次数就是n-1。完成n-1次遍历之后，最小的数据项一定已经归位，此时不需要再执行其他步骤。代码1是完成冒泡排序的函数。它以待排列表为参数，通过交换必要的数据项实现对列表的修改，完成排序。

交换数据项的操作，有时也叫“交换（swap）”，Python中的交换操作与其他许多编程语言不同。通常来说，交换列表中的两项需要一个暂存位置（一个附加的储存空间）。以上这段代码可以交换列表中的第i项和第j项。如果没有暂存位置的话，其中一个值就会被覆盖。

```
temp = alist[i]
alist[i] = alist[j]
alist[j] = temp
```

但是在Python里，可以“同时赋值”。通过“a,b=b,a”的语句就可以让两个赋值语句同时进行（如图2）。用同时赋值的方法可以用一条语句实现交换操作。

代码1中5-7行交换**第i项**和**第i+1项**用了前述的三行代码，其实我们也可以用同时赋值来交换数据。

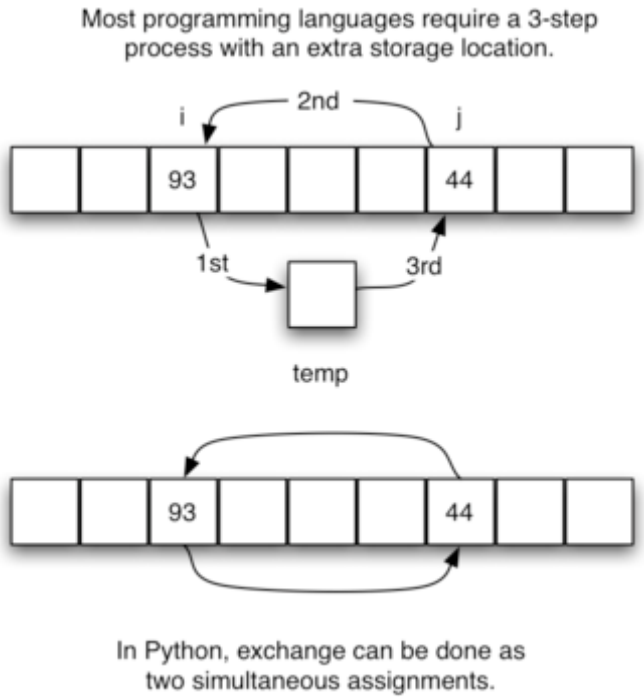


图5.2　Python中　　两个值的交换

下面的代码展示了实现冒泡排序的完整函数。

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for I in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp
alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

代码5.1　冒泡排序(lst_bubble)

下面我们来分析冒泡排序。需要注意到，不管初始列表中的数据如何排列，排一个长度为n的列表都要进行n-1次遍历。表1显示了每次遍历需要比较的次数。总的比较次数是从1到n-1的所有正整数的和，即$1/2(n^2-n)$。比较复杂度为$O(n^2)$。在最好的情况下，如果列表已经排好序，就不需进行交换；但是在最坏的情况下，每一次比较都要进行一次交换，平均下来，我们交换操作次数是比较次数的一半。

遍历次数	比较次数
1	n-1
2	n-2
3	n-3
…	…
n-1	1

表格5.1　冒泡排序每次遍历的比较次数

因为冒泡排序必须要在最终位置找到之前不断交换数据项，所以它经常被认为是最低效的排序方法。这些“浪费式”的交换操作消耗了许多时间。但是，由于冒泡排序要遍历整个未排好的部分，它可以做一些大多数排序方法做不到的事。尤其是如果在整个排序过程中没有交换，我们就可断定列表已经排好。因此可改良冒泡排序，使其在已知列表排好的情况下提前结束。这就是说，如果一个列表只需要几次遍历就可排好，冒泡排序就占有优势：它可以在发现列表已排好时立刻结束。代码2就是改良版冒泡排序。它通常被称作**“短路冒泡排序”**。

```
def shortBubbleSort(alist):
        exchanges = True
        passnum = len(alist)-1
        while passnum > 0 and exchanges:
          exchanges = False
          for i in range(passnum):
                if alist[i]>alist[i+1]:
                      exchange = True
                      temp = alist[i]
                      alist = alist[i+1]
                      alist[i+1] = temp
                passnum = passnum-1
```

1．给定排序列表[19，1，9，7，3，10，13，15，8，12]，如下哪个列表是冒泡排序第三次遍历之后的列表？

a）[1，9，19，7，3，10，13，15，8，12]

b）[1，3，7，9，10，8，12，13，15，19]

c）[1，7，3，9，10，13，8，12，15，19]

d）[1，9，19，7，3，10，13，15，8，12]

5.3.2.选择排序

选择排序提高了冒泡排序的性能，它每遍历一次列表只交换一次数据，即进行一次遍历时找到最大的项，完成遍历后，再把它换到正确的位置。和冒泡排序一样，第一次遍历后，最大的数据项就已归位，第二次遍历使次大项归位。这个过程持续进行，一共需要n-1次遍历来排好n个数据，因为最后一个数据必须在第n-1次遍历之后才能归位。

图3展示了选择排序的整个过程。每一次遍历，最大的数据项被选中，随后排到正确位置。第一次遍历排好了93，第二次排好了77，第三次排好了55，以此类推。代码1展示了实现选择排序的函数。

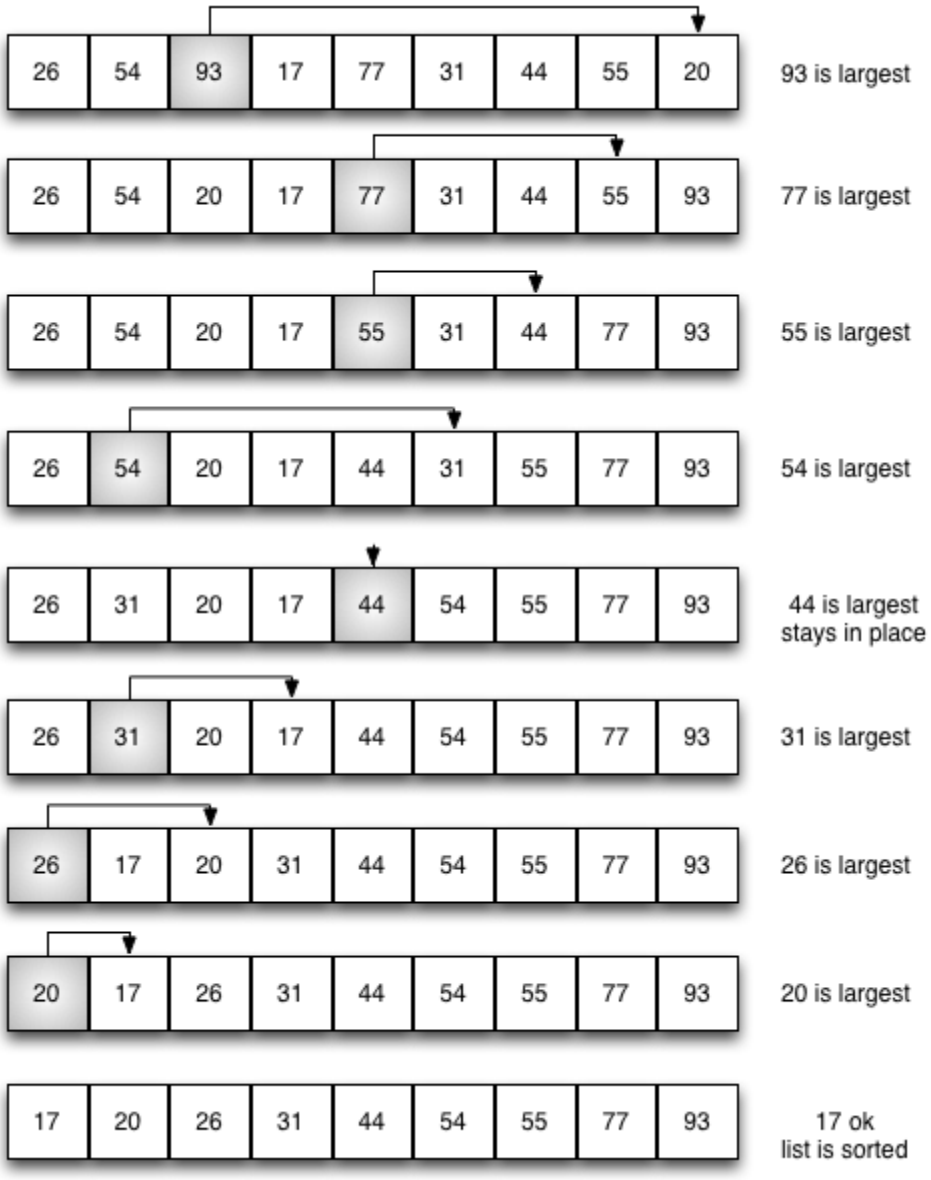


图5.3 选择排序

```
def selectionSort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location
temp = alist[fillslot]
alist[fillslot] = alist[positionOfMax]
alist[positionOfMax] = temp

alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

自我测试

1．给定排序列表[11，7，12，14，19，1，6，18，8，20]，如下哪个列表是选择排序第三次遍历之后的列表？

a）[7，11，12，1，6，14，8，18，19，20]

b）[7，11，12，14，19，1，6，18，8，20]

c）[11，7，12，14，1，6，8，18，19，20]

d）[11，7，12，14，8，1，6，18，19，20]

5.3.3.插入排序

插入排序的算法复杂度仍然是$O(n^2)$，但其工作原理稍有不同。它总是保持一个位置靠前的已排好的子表，然后每一个新的数据项被“插入”到前边的子表里，排好的子表增加一项。图4展示了插入排序的过程。阴影区域的数据代表了程序每运行一步后排好的子表。

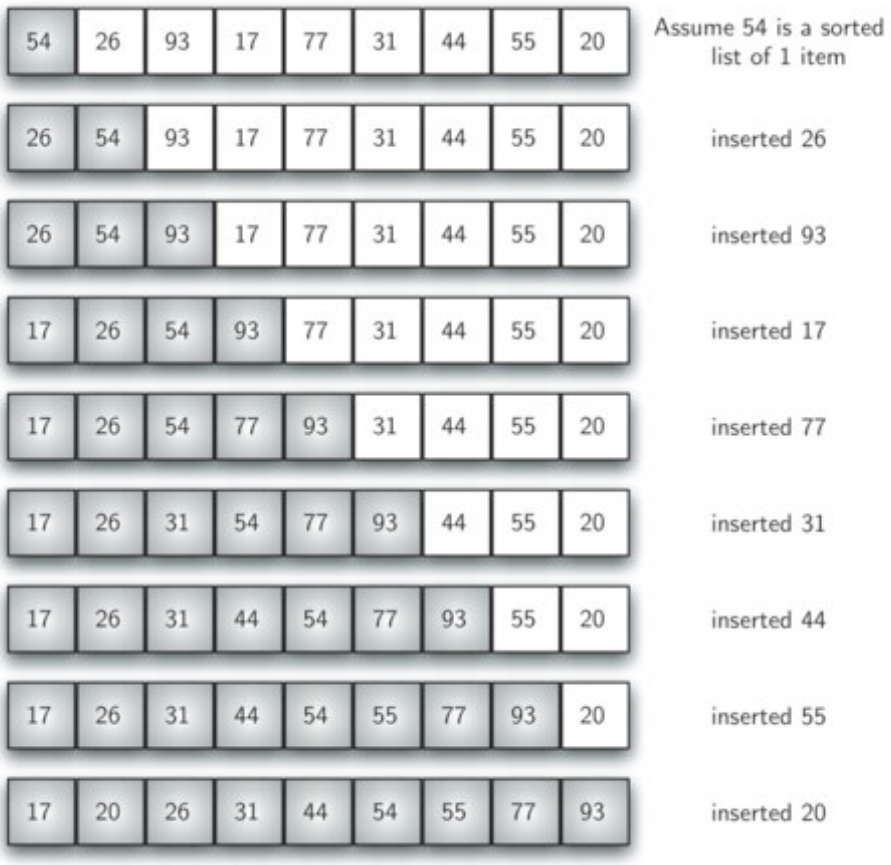


图5.4　插入排序（insertionSort）

我们认为只含有一个数据项的列表是已经排好的。每排后面一个数据（从1开始到n-1），这个的数据会和已排好子表中的数据比较。比较时，我们把之前已经排好的列表中比这个数据大的移到它的右边。当子表数据小于当前数据，或者当前数据已经和子表的所有数据比较了时，就可以在此处插入当前数据项。

图5.4详细地展示了第五步排序的过程。程序运行到当前位置，已排好的子表中包含了“17，26，54，77，93”五个数据。我们想让31插入该子表中。第一次，31和93比较，93要移到31右边。同理，77和54也要移位。遇到26时，移动步骤停止，31被插入到此处。此时我们就有了一个含6个数据项的已排好的子表。

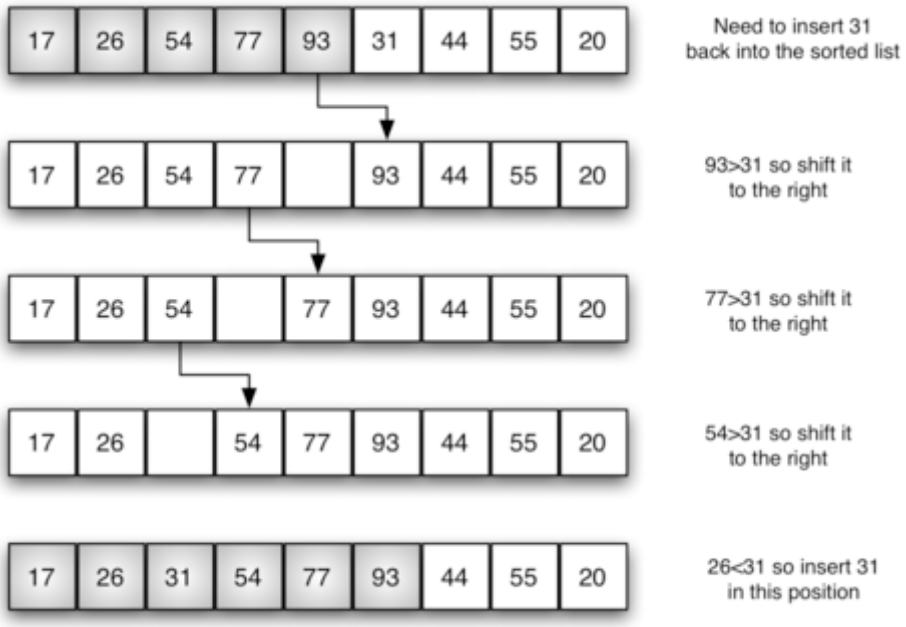


图5.5　插入排序：第五步排序

在插入排序（代码5.4）的实现中，排n个数据仍然需要进行n-1次遍历（n-1步）。由于每个数据需要被插入到之前排好的子表中，故迭代操作要从只有一个位置的情况开始，在余下n-1个位置重复进行。程序第8行是转移操作，即把某个数据放到后一个位置，空出当前位置以待数据插入。需要注意的是，这并不是前两个排序方法里的完整的“交换”步骤。

插入排序需要进行的最多的比较次数仍是从1到n-1的所有的整数之和，即复杂度为$O(n^2)$。但是，最好的情况下，每排一个数据只需要一次比较，即列表已经排好的情况。

关于“转移”与“交换”操作的考虑也很重要。通常情况下，“转移”的步骤约为“交换”步骤的1/3，因为它只有一次赋值操作。在基准测试中，插入排序将展示非常好的性能。

```
def insertionSort(alist):
    for index in range(1,len(alist)):

        currentvalue = alist[index]
        position = index

        while position>0 and alist[position-1]>currentvalue:
            alist[position]=alist[position-1]
            position = position-1

        alist[position]=currentvalue
alist = [54,26,93,17,77,31,44,55,20]
```

1. 给定排序列表[15, 5, 4, 18, 12, 19, 14, 10, 8, 20] ，如下哪个列表是插入排序第三次遍历之后的列表？

   a) [4, 5, 12, 15, 14, 10, 8, 18, 19, 20]

   b) [15, 5, 4, 10, 12, 8, 14, 18, 19, 20]

   c) [4, 5, 15, 18, 12, 19, 14, 10, 8, 20]

   d) [15, 5, 4, 18, 12, 19, 14, 8, 10, 20]

5.3.4. 希尔排序

希尔排序有时又叫做“缩小间隔排序”，它以插入排序为基础，将原来要排序的列表划分为一些子列表，再对每一个子列表执行插入排序，从而实现对插入排序性能的改进。划分子列的特定方法是希尔排序的关键。我们并不是将原始列表分成含有连续元素的子列，而是确定一个划分列表的增量“i”，这个i更准确地说，是划分的间隔。然后把每间隔为i的所有元素选出来组成子列表。

如图6，这里有一个含九个元素的列表。如果我们以3为间隔来划分，就会分为三个子列表，每一个可以执行插入排序。这三次插入排序完成之后，我们得到了如图7所示的列表。虽然这个列表还没有完全排好序，但有趣的是，经过我们对子列的排序之后，列表中的每个元素更加靠近它最终应该处在的位置。

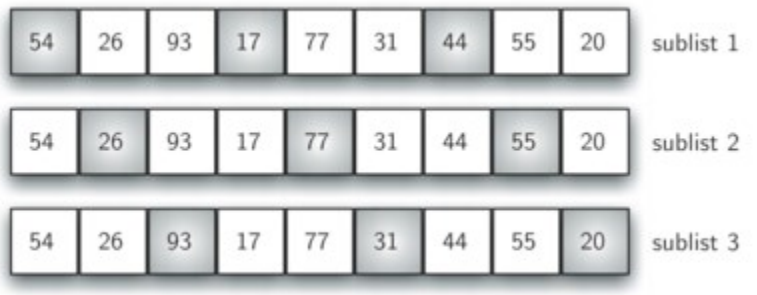


图 5.6 以3为间隔的希尔排序

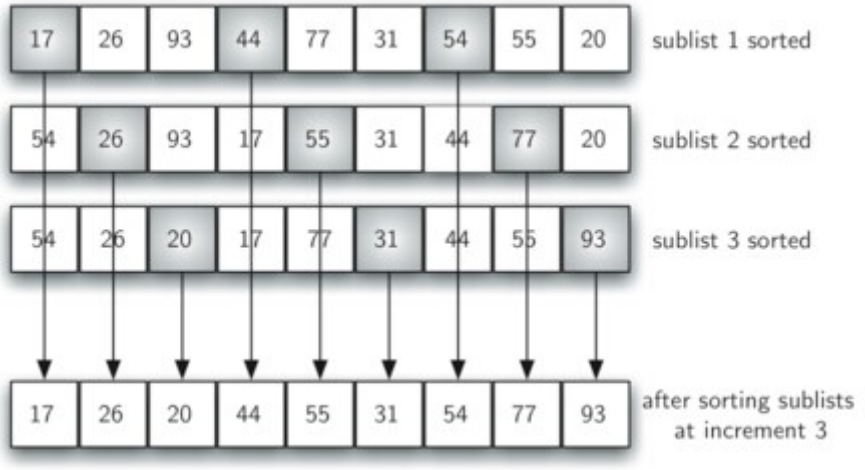


图 5.7 对每个子列排序后的情况

图8为我们展示了最终以1为间隔进行插入排序，即标准的插入排序的过程。注意到，通过对前面子列进行排序之后，我们减少了将原始列表排序时需要比对和移动的次数。在这个例子中，我们仅需要再进行四次移动就可以完成排序。

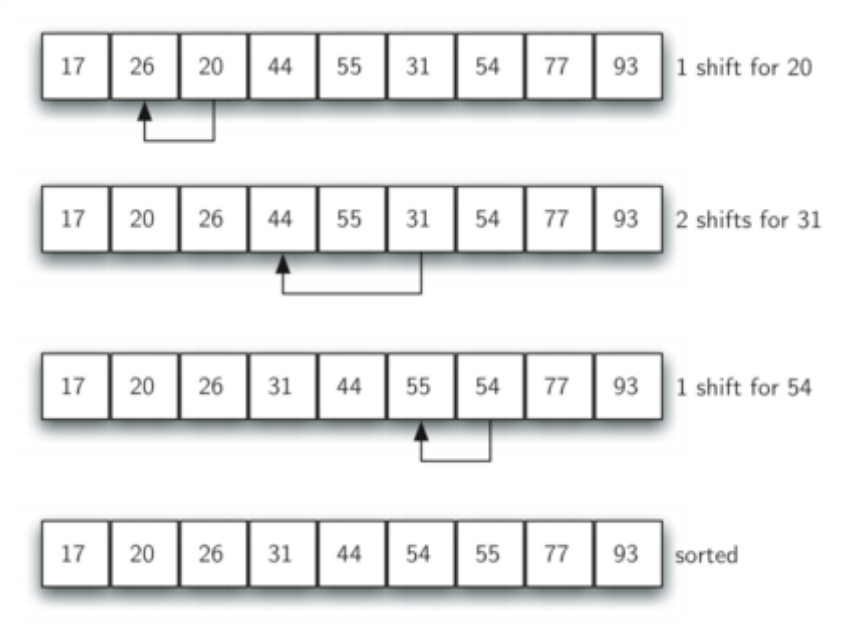


图 5.8 最后以1为间隔的插入排序

之前我们提到过，特定选取划分间隔的方法是希尔排序的独特之处。代码1中的函数使用了一组不同的间隔。在这个例子中，我们从含2个元素的子列开始排序；下一步排含4个元素的子列。最终，整个数列用基本的插入排序排好。图9显示了我们的例子中用这个间隔的第一组子列表（可能有四种情况）。

下面对希尔排序函数的调用会显示出以不同间隔部分排序后的列表，最后一次排序间隔为1。

代码 5.1　希尔排序

```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:

    for startposition in range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

    print("After increments of size",sublistcount,
                                    "The list is",alist)

    sublistcount = sublistcount // 2

def gapInsertionSort(alist,start,gap):
for i in range(start+gap,len(alist),gap):

        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
        alist[position]=alist[position-gap]
        position = position-gap

        alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
shellSort(alist)
print(alist)
```

乍一看，你可能会觉得希尔排序不会比插入排序要好，因为它最后一步执行了一次完整的插入排序。但事实上，最后的一次排序并不需要很多次的比对和移动，因为正如上面所讨论的，这个列表已经在之前的对子列的插入排序中实现了部分排序。换句话说，每个步骤使得这个列表与原来相比更趋于有序状态。这使得最后的排序非常高效。

对希尔排序的综合算法分析已经远超出本书的讨论范围，基于对该算法的描述，我们可以说它的时间复杂度大致介于O(n)和$O(n^2)$之间。如果使用某些间隔时，它的时间复杂度为$O(n^2)$。通过改变间隔的大小，比如以2k-1（1,3,5,7,15,31等等）为间隔，希尔排序的时间复杂度可以达到$O(n^{3/2})$。

自我检测

1. 给定列表：[5, 16, 20, 12, 3, 8, 9, 17, 19, 7]，下面那个表示了以3为间隔完成所有交换后的列表情况？

   a) [5, 3, 8, 7, 16, 19, 9, 17, 20, 12]

   b) [3, 7, 5, 8, 9, 12, 19, 16, 20, 17]

   c) [3, 5, 7, 8, 9, 12, 16, 17, 19, 20]

   d) [5, 16, 20, 3, 8, 12, 9, 17, 20, 7]

5.3.5.归并排序

我们现在把注意力转移到用分而治之的策略来改进排序算法的表现。我们要学的第一种算法就是归并排序。归并排序是一种递归算法，它持续地将一个列表分成两半。如果列表是空的或者只有一个元素，那么根据定义，它就被排序好了（最基本的情况）。如果列表里的元素超过一个，我们就把列表拆分，然后分别对两个部分调用递归排序。一旦这两个部分被排序好了，那么这种被叫做归并的最基本的操作，就被执行了。归并是这样一个过程：把两个排序好了的列表结合在一起组合成一个单一的，有序的新列表。图10就展示了我们熟悉的作为例子的列表如何被归并排序算法拆分，而图11则展示了这些简单的列表被排序好并被重新归并回去的过程。

在源代码1中展示的归并排序函数开始于询问基本条件问题。如果列表的长度小于或等于一，那么我们已经有了一个排好序的列表并不需要做任何更多的过程了。如果不是，即列表长度大于一，那么我们用Python的切片操作将左右两部分拆开。注意到这一点是非常重要的：列表可能不拥有偶数个项目。但是这无关紧要，因为长度的最大差别也不会超过一。

一旦归并排序函数被调用处理左右两半边（8-9行），它假定它们已经被排序完了。而剩余部分的函数（11-31行）是用来把两个小的有序列表合成一个大的有序列表的。注意归并操作是重复地把两个小的有序列表中的最小元素一一放回原列表的。

归并排序函数被增加了一个打印语句（第二行）用来显示每次被调用之前被排序的列表的内容。同时也有一条打印语句（32行）用来显示归并的过程。记录显示了函数在我们的示例列表里的执行结果。显示了有44,55和20的列表并没有被平均地拆分，44被分给了第一个列表而55和20被分给了第二个。我们可以很容易地看到拆分过程是如何最终产生一个能迅速被另一些有序列表合并的列表的。

为了分析归并算法，我们需要考虑它实施的两个不同步骤。第一步，列表被拆分，我们已经（在二分查找中）计算过，我们能通过logn的数量级的计算将长度为n的列表拆分。而第二个过程是合并。每个列表中的元素最终将被处理并被放置在排序好的列表中，所以合并操作对一个长度为n的列表需要n的数量级的操作。因此分析结果就是，拆分需要logn数量级的操作而每次拆分需要n数量级的操作因此最终操作的复杂度为nlogn。归并排序是一种O(nlogn) 的算法。

回忆起，切片操作对于一个长度为K的切片需要K的复杂度，为了保证归并排序的复杂度为O(nlogn)，我们需要移开切片操作，这是可能的，只要我们简单地在递归的时候传入列表的开始和结束的参数就可以了。我们将这个步骤作为一个课后练习。

很重要的是要意识到在拆分列表时归并排序函数需要额外的空间来存放被拆分出来的两个部分。如果列表很大的话，这额外空间将是一个很重要的因素，可能使得这种排序被运用在大数据集合时出现错误。

自我检测

1. 给定如下数字列表[21, 1, 26, 45, 29, 28, 2, 9,16, 49, 39, 27, 43, 34, 46, 40]
   哪个答案展示了归并排序被三次调用后的结果？

   a) [16, 49, 39, 27, 43, 34, 46, 40]

   b) [21,1]

   c) [21, 1, 26, 45]

   d) [21]

2. 给定如下数字列表[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]
   哪个答案展示了最先被合并的两个列表？

   a) [21, 1] 和[26, 45]

   b) [[1, 2, 9, 21, 26, 28, 29, 45] 和[16, 27, 34, 39, 40, 43, 46, 49]

   c) [21] 和[1]

   d) [9] 和[16]

5.3.6.快速排序

快速排序用了和归并排序一样分而治之的方法来获得同样的优势，但同时不需要使用额外的存储空间。经过权衡之后，我们发现列表不分离成两半是可能的，当这发生的时候，我们可以看到，操作减少了。

快速排序首先选择一个中值。虽然有很多不同的方法来选择这个数值，我们将会简单地选择列表里的第一项。中值的作用在于协助拆分这个列表。中值在最后排序好的列表里的实际位置，我们通常称之为分割点的，是用来把列表变成两个部分来随后分别调用快速排序函数的。

图12展示了54将被用来作为我们的第一个中值。由于我们已经看过这儿例子很多遍了，我们知道54最终将占据31的位置。随后，分区过程将要开始。它将找到分割点的位置并且同时将移动项目到列表中的合适位置（比中值小的数放在左边，大的放在右边）

分区过程由设置两个位置标记开始——让我们叫它们左标记和右标记——在列表的第一项和最后一项。分区过程的目标是把相对于中值在错误的一边的数据放到正确的一边，同时找到分割点。图13展示了这个过程，同时我们把54放到了正确的位置。

我们是这样开始的：我们不断把左标记向右移动直到它指向了一个比中值更大的数字。我们然后把右标记向左移动直到我们找到一个比中值更小的数字。在这个时候我们就找到了两个相对于最终的分割点在错误的位置的元素。在我们的例子中，就是93和20。现在我们可以交换这两个元素，然后重复这个步骤了。

在右标记变得比左标记小的时候，我们停止，此时右标记在的位置就是分割点在的位置。而中值就可以和分割点的内容互换位置而被放置在正确的位置上了。

另外，每个分割点左边的元素都比中值小，每个右边的都比它大了。这个列表就可以在分割点被分成两半然后快速排序可以在这两个部分被分别调用了。

在源代码1中所示的快速排序函数中，调用了递归函数，quickSortHelper。
quickSortHelper从与归并排序相同的最基本的情况开始。如果列表长度小于或者等于1，那么它

已经排过序。如果列表的长度更大，那么列表可以被分割并进行递归排序。分裂函数的实现过程在前面已经描述。

来分析快速排序函数，注意一个长度为n的列表，如果每次分裂都发生在列表的中央，那么将会重复logn次分裂。为了找到分割点，n个项目中的每一个都需要和中值进行对比。那么，综合起来是nlogn。此外，快速排序不需要像在归并排序时所需的额外内存。

不幸的是，在最坏的情况下，分割点可能不在中间，可以是非常偏左或偏右，留下一个很不均匀的分裂。在最坏的情况下，给一个长度为n的列表排序，分成了给一个长度为0的列表排序和给一个长度为n-1的列表排序。然后，给一个长度为n-1的列表排序，分成了给一个长度为0的列表排序和给一个长度为n-2的列表排序，以此类推。最终，用以上所述的递归过程，成了时间复杂度为0（n^2）的排序。

我们之前提到过，有多种不同的方法用来选择中值。特别的是，我们可以通过使用一种名为“三点取样”的方法，来尝试着降低不均匀分割的可能性。为了选择中值，我们要考虑列表中第一个、中间一个以及最后一个三个元素。在我们的例子中，它们分别是54,77和20。现在选取中间值（在我们的例子中是54），并且使用它作为中值（当然，这和我们最初使用的中值相同）。这种方法在于当列表的第一项不趋于处于列表中间位置时，三个值的中间值将会是一个更好的中值选择。当最初的列表已经经过一定程度的排序时，这种方法就显得尤其有用。我们将这种中值的选择方法留下作为一个练习。

自我检测

1. 给定列表[14, 17, 13, 15, 19, 10, 3, 16, 9,12]。依据给出的快速排序算法，下面哪个选项表示的是上述列表在快速排序时，在第2次分裂后的列表内容：

   a) [9, 3, 10, 13, 12]

   b) [9, 3, 10, 13, 12, 14]

   c) [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]

   d) [9, 3, 10, 13,12, 14, 19, 16, 15, 17]

2. 给定列表[1, 20, 11, 5, 2, 9, 16, 14, 13,19]。使用“三点取样”，下面哪个选项表示的是上述列表在快速排序时，得到的第1个“中值”：

   a) 1　b) 9　c) 16　d) 19

3. 下面哪些算法，即使在最坏情况下，复杂度还保证是0(nlog n)

   a) 谢尔排序

   b) 快速排序

   c) 归并排序

   d) 插入排序

5.4.小结

- 在无序表或者有序表上的顺序搜索，其时间复杂度为0(n)；
- 在有序表上进行二分查找，在最差情况下，复杂度为0(log n)；

- 散列表可以实现常数级时间的搜索；
- 冒泡排序、选择排序和插入排序是$O(n^2)$的算法；
- 谢尔排序在插入排序的基础上进行了改进，采用对递增子表排序的方法，其时间复杂度可以在O(n)和$O(n^2)$之间；
- 归并排序的时间复杂度是O(nlog n)，但归并的过程需要额外存储空间；
- 快速排序的时间复杂度是O(nlog n),但如果分割点偏离列表中心的话，最坏情况下会退化到O(n^2)。快速排序不需要额外的存储空间。

5.5 关键词

二分搜索	**冒泡排序**	**数据项链**
聚集	**冲突**	**冲突解决**
折叠法	**间隔**	**散列函数**
散列表	**散列过程**	**插入排序**
线性探测	**负载系数**	**映射**
三点取样	**归并**	**归并排序**
平方取中	**开放定址**	**分裂**
完美散列函数	**中值**	**二次探查**
快速排序	**再散列**	**选择排序**
顺序查找	**谢尔排序**	**改进冒泡**
槽	**分裂点**	

5.6.问题讨论

1. 使用在本章中给出的散列表的性能公式，计算平均情况下必要的比较次数，当散列表中已占槽数与总槽数之比为：

- 10%
- 25%
- 50%
- 75%
- 90%
- 99%

在什么情况下，你认为散列表太小了？请解释。

2. 使用将在字符串中所在的位置作为权重因子的方法，修改散列函数。

3. 我们使用一个将在字符串中所在的位置作为权重因子的散列函数。设计一个可选择的加权方案。这些函数的偏向有何不同？

4. 研究完美散列函数。使用一个姓名的列表（同学，家人……），用完美散列函数生成对应的散列值。

5. 产生一个随机的整数列表，并表示该列表是怎样被下列的算法排序的:

- 冒泡排序
- 选择排序
- 插入排序
- 谢尔排序(增量自取)
- 归并排序
- 快速排序(中值自取)

6. 考虑下面的整数列表：[1,2,3,4,5,6,7,8,9,10]，表示该列表是怎样被下列的算法排序的:

- 冒泡排序
- 选择排序
- 插入排序
- 谢尔排序（增量自取）
- 归并排序
- 快速排序(中值自取)

7. 考虑下面的整数列表：[10,9,8,7,6,5,4,3,2,1]，表示该列表是怎样被下列的算法排序的:

- 冒泡排序
- 选择排序
- 插入排序
- 谢尔排序(增量自取)
- 归并排序
- 快速排序(中值自取)

8. 考虑下面的字母列表：['P','Y','T','H','O','N']，表示该列表是怎样被下列的算法排序的:

- 冒泡排序
- 选择排序
- 插入排序
- 谢尔排序（增量自取）
- 归并排序
- 快速排序(中值自取)

9. 为快速排序中选择中值设计一些可选择的策略。例如，选择中间项。实现新的快排算法并且使用一些随机的数据集进行测试。在什么条件下，你的新算法表现的比本章所给的算法更优或更劣？

5.7.编程练习

1. 设置一个随机实验，测试在整数列表中顺序搜索和二分搜索的区别。使用教材中给出的二分搜索（递归和迭代）。生成一个随机的、有序的整数列表。对每个函数做一个标准检查程序分析。你的结果是什么？你能解释一下吗？

2. 用递归算法实现二分搜索，避免用切片操作。记得，对子列表传递的除了列表外，还要有开始以及结束位置在列表中的索引值。通过一个随机产生的有序整数列表来与课件中采用切片操作的二分搜索比较性能。
3. 为散列表实现的ADT Map实现len方法（__len__）。
4. 为散列表实现的ADT Map实现in方法（__contains__）。
5. 你如何从一个使用数据项链的方法来解决冲突的散列表中删除项目？如果使用的是开放定址的方法呢？必须处理的特殊情况是什么？实现实现Hashtable类的del方法。
6. 在散列表实现的Map中，散列表的大小被定为101。如果散列表被填满，它需要增长。重新实现put方法，使得当负载因子达到某一预定值时（你可以决定这一值基于你对负载与性能的评估），散列表会自动调整自身的大小。
7. 实现二次探查作为再散列手段。
8. 使用一个随机数生成器，产生一个500个整数的列表。使用本章中的一些排序算法进行基准分析。比较它们在执行速度上的快慢。
9. 使用同时赋值的方法实现冒泡排序。
10. 冒泡排序可以被修改为向两个方向冒泡。第一次操作向上冒泡，第二次操作向下冒泡。这种交替模式一直持续到不需要更多的操作。实现这种改变并描述在什么情况下这种改变是合适的。
11. 使用同时赋值的方法实现选择排序。
12. 使用不同的增量设置在同一个列表上，进行一个谢尔排序的基准分析。
13. 不使用切片操作实现归并排序函数。
14. 提高快速排序的一种方法是在较短长度（称之为“分裂限制”）的列表中使用插入排序。为什么这是有意义的？重新实现快速排序并用它来为随机整数列表排序。使用不同大小的分裂限制进行分析。
15. 实现“三点取样法”，用来在快速排序中确定中值的改进，进行一个实验来比较两种方法。

6.树和树算法

6.1.目标

- 理解树的数据结构及其使用方法；
- 树用于实现 ADT Map；
- 用列表来实现树；
- 用类和引用来实现树；
- 以递归数据结构实现树；
- 用堆来实现优先队列；

6.2.树的例子

我们已经学过了像栈和队列这样的线性数据结构，对递归也有了一定的了解，现在让我们来关注另一种常见的数据结构——**树**（Tree）。树在计算机科学的各个领域中被广泛应用，包括操作系统，图形学，数据库系统和计算机网络。树结构和自然界的树有许多相似的地方，也有根、枝和叶，它们的不同之处在于计算机中的树结构根在顶部而叶子则在底部。

在我们开始学习树之前，让我们先来看看几个常见的例子。第一个例子生物学中的分类树。图 6.1 是一个动物分类的例子。从这个简单的例子中，我们可以看出树结构的几个性质。这个例子说明的第一个性质是树是分层的，这里分层的意思是树的顶层部分更加宽泛一般而底层部分更加精细具体。在这个例子中，最上层是“界”，它下面的一层（上层的子层）是“门”，然后是“纲”等等。但是，无论我们细分到多少层，这里面包含的生命体仍是动物。

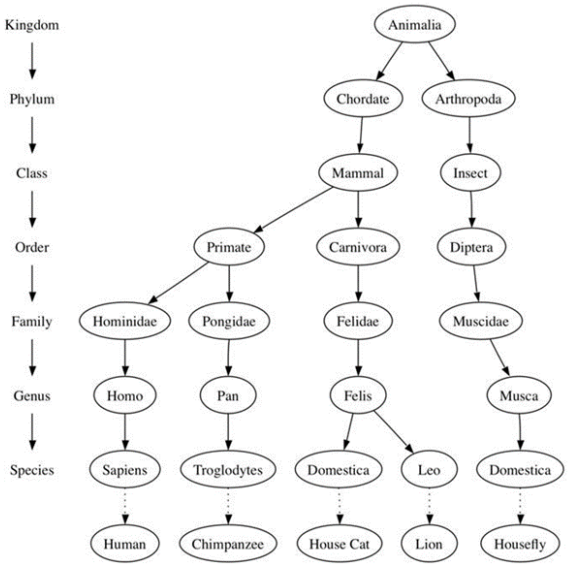


图 6.1 一些动物的分类树

注意到你可以从树的顶层开始然后沿着圆圈和箭头构成的一条路径到达树的底层。在树的 每一层我们都可以问自己一个问题，然后沿着相符的那条路径继续走下去。比如我们可以问 “这个动物是脊椎动物还是无脊椎动物？”，如果回答是“脊椎动物”我们就沿着脊椎动物这条路继续走下去，然后接着问“这个脊椎动物是哺乳动物吗？”，如果回答“不是哺乳动物”我们就卡在这里了（不过仅限于这个简单的例子会有这种情况）。当我们到达哺乳动物这一层的时候我们问自己“这个哺乳动物是灵长类动物还是食肉动物？”。我们可以沿着路径一直走下去，直到树的最底层，也就是我们平常的命名了。

树的第二个性质是一个节点（node）的所有子节点（children）和另一个节点的子节点是完全独立的。比如“猫属”有两个子节点“家生”和“野生”，“蝇属”中也有一个“家生”，但它和“猫属”中的“家生”是完全不同而且相互独立的。这意味着我们可以在不影响“猫属”的子节点的情况下更改“蝇属”的子节点。

树的第三个性质就是它的每个叶节点（leaf）都是不同的。对每一种动物，我们都可以从根节点（root）开始沿着一条特定的路径找到它对应的叶节点，并把它和其他动物区分开，例如对于家猫，我们可以沿着 动物界 → 脊索动物门 → 哺乳动物纲 → 食肉动物目 → 猫科 → 猫属 → 家猫找到它。

另一个树结构的例子就是你可能每天都会用到的文件系统。在一个文件系统中，磁盘的分支或者文件夹构建成了一棵树。图 6.2 展示了一小部分 Unix 文件系统的分层情况。

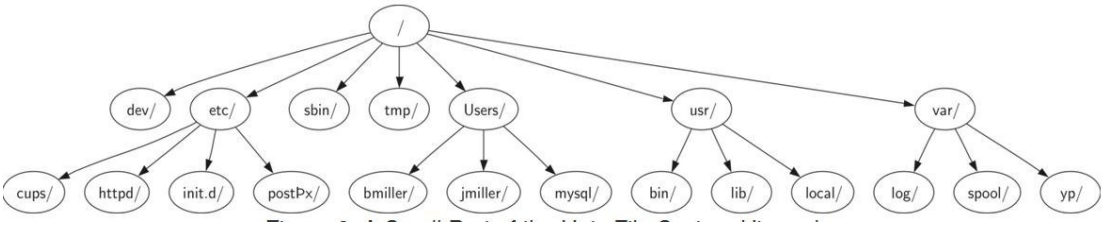


图 6. 2 一小部分 Unix 文件系统的分层情况

这个文件系统树和自然界的树也很相像。你可以从根节点出发沿着一条路径到任意分支。这条路径会把这个子分支（包括它里面的所有文件）和其他分支区别开。树结构的另一个重要性质，就是你可以将树下层的所有部分（叫做**子树 subtree**）移动到树的另一位置而不影响更下层的情况，这是由树的分层性决定的。例如，我们可以将所有标注/etc 的子树从根节点下移动到 usr/下面。这样做会将 httpd 的路径从/etc/httpd 改变成/usr/etc/httpd，但是对 httpd 的内容及其子节点的内容不会有影响。

最后一个树的例子是一个网页。下面是一个利用超文本标记语言（HTML）编写的简单网页。图 6.3 是与构成网页的超文本标记语言中的标签相对应的树。

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
<li>List item one</li>
<li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```

代码 6. 1

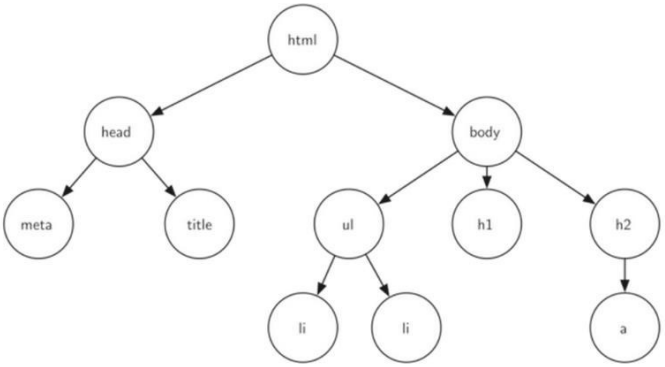


图 6.3 与网页的构成元素相对应的树

上面的 HTML 源代码和它对应的树说明了另一种分层方式。我们发现树的每一层都对应 超文本标记符的一层嵌套。代码的第一个标记符是<html>同时最后一个是</html>。这一页中 所有其他的标记符也都是成对的。试一下你就会发现这种嵌套的特点在树的每一层都是成立 的。

6.3.术语表和定义

既然我们已经看过几个树的例子了，我们将正式定义树以及构成它的要素。

节点（Node）

节点（Node）是树的基本构成部分。它可以有其他专属的名称，我们称之为“键（key）”。 一个节点可能有更多的信息，我们称之为“负载（payload）”。尽管负载信息和树的许多算法并不直接相关，但是它对于树的应用至关重要。

边（Edge）

边（Edge）也是另一个树的基本构成部分。边连接两个节点，并表示它们之间存在联系。每个节点（除了根节点）都有且只有一条与其他节点相连的入边（指向该节点的边），每个节点可能有许多条出边（从该节点指向其他节点的边）。

根节点（Root）

根节点是树中唯一一个没有入边的节点。在图 6.2 中，“/”是树的根节点。

路径（Path）

路径是由边连接起来的节点的有序排列。例如：动物界 → 脊索动物门 → 哺乳动物纲 → 食肉动物目 → 猫科 → 猫属 → 家猫 就是一条路径。

子节点集（Children）

当一个节点的入边来自另一个节点时，我们称前者是后者的子节点，同一个节点的所有子节点构成子节点集。在图 6.2 中，节点 log/, spool/, yp/是节点 var/的子节点。

父节点（Parent）

一个节点是它的出边所连接的所有节点的父节点。在图 6.2 中，节点 var/是节点 log/, spool/, yp/的父节点。

兄弟节点（Sibling）

同一个节点的所有子节点互为兄弟节点，在文件系统树中节点 etc/和节点 usr/是兄弟节点。

子树（Subtree）

子树是一个父节点的某个子节点的所有边和后代节点所构成的集合。

叶节点（Leaf Node）

没有子节点的节点成为称为叶节点。例如图 6.1 中的“人”和“黑猩猩”就是叶节点。

层数（Level）

一个节点的层数是指从根节点到该节点的路径中的边的数目。例如，图 6.1 中“猫属”的层数为 5。定义根节点的层数为 0。

高度（Height）

树的高度等于所有节点的层数的最大值。图 6.2 中树的高度为 2。

我们已经定义好所需的基本术语了，现在可以正式定义树了。我们将用两种方式定义，一种需要用到节点和边，而另一种更为有用的定义方式是利用递归定义。

定义一：树是节点和连接节点的边的集合，它有以下特征：

- 有一个节点是根节点；
- 除了根节点外的每一个节点 n，都通过一条边与另一个节点 p 相连，p 是 n 的父节点；
- 可以沿着唯一的路径从根节点到达每个节点；
- 如果这个树的每个节点都至多有两个子节点，我们称它为二叉树；

图 6.4 展示了一个符合定义一的树，每条边的箭头指出了连接的方向；

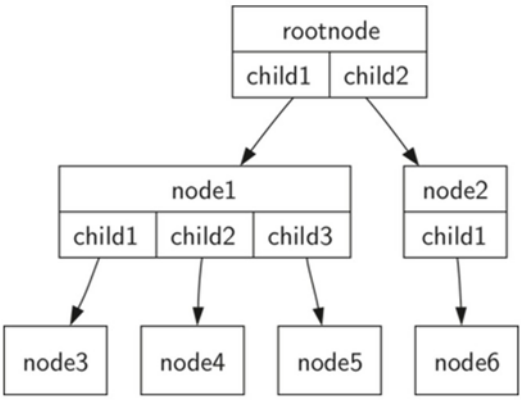


图 6. 4 由节点和边构成的树

定义二：每个树或者为空或者包含一个根节点和零个或多个子树，其中每个子树也符合这样的定义。每个子树的根节点和其父树的根节点之间通过边相连。图 6.5 展示了这种递归定义的树。通过树的递归定义，我们知道图 6.5 中的树至少有 4 个节点，因为每个三角形所代表的子树必须有根节点。它也可能有更多的节点，但我们并不知道，除非更深入地了解这棵树。

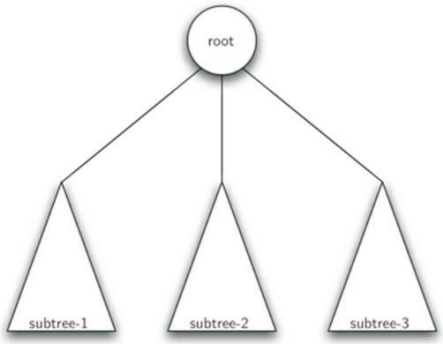


图 6.5 树的递归定义

6.4.通过嵌套列表实现树

在用嵌套列表实现树时，我们将用 Python 的列表数据结构来编写上面定义的功能。虽然把 界面写成列表的一系列方法与我们已实现的其他抽象数据类型有些不同，但这样做的是有趣 的，因为它为我们提供一个简单、可以直接检查的递归数据结构。在列表实现树时，我们将存储根节点的值作为列表的第一个元素。列表的第二个元素是一个表示其左子树的列表，第三个元素是表示其右子树的另一个列表。为了说明这个存储技术，让我们来看一个例子。图 6.6 示出一个简单的树以及相应的列表实现。

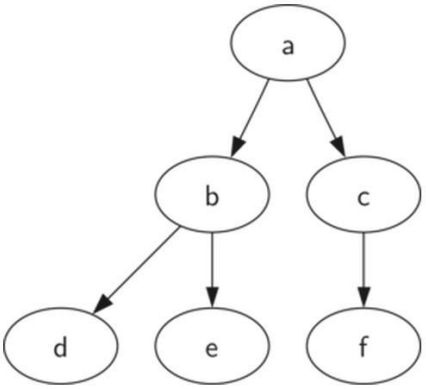


图 6.6 一个简单的树

```
myTree = ['a', root
          ['b',    #left subtree
          ['d', [], []],
          ['e', [], []] ],
          ['c',    #right subtree
          ['f', [], []],
```

请注意，我们可以使用标准索引访问列表的子树。树的根节点 myTree[0]，根节点的左子树是 myTree[1]，右子树是 myTree[2]。 代码 6.3 说明了如何用列表创建简单树。一旦树被构建，我们可以访问根和左、右子树。嵌套列表法的一个非常好的特性是子树的结构与树相同；这个结构本身是递归的！子树具有一个根值和两个表示叶节点的空列表。嵌套列表法的另一个优点是它容易扩展到多叉树。在树不仅仅是一个二叉树的情况下，另一个子树只是另一个列表。

我们将提供一些功能，这些功能使我们很容易像树一样使用列表。请注意，我们不会去定义一个二叉树类。我们将编写的功能将只是帮助我们操作标准列表使之类似于树类型。

BinaryTree 功能只是构建包含一个根节点和两个表示子节点的空列表的列表。给左子树添加到树的根，我们需要插入一个新的列表到根列表的第二个位置。我们必须注意，如果列表中已经有元素在第二个位置，我们需要跟踪它，并将新节点插入树中作为其左子节点。代码 6.5 展示了插入左子节点 Python 代码。

```
myTree = ['a', ['b', ['d',[],[]], ['e',[],[]] ], ['c', ['f',[],[]], []] ]
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
```

代码 6. 5

请注意，插入一个左子节点前，我们首先应获取对应于当前左子节点的列表（可能是空的）。然后，我们添加新的左子节点，将原来的左子结点作为新节点的左子节点。这使我们能够将新节点插入到树中的任何位置。insertRight 的代码类似于 insertLeft，并已展示在代码 6.6 中。

```
def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
```

为了完善树的功能（参见代码 6.7），让我们编写几个用于获取和设置树的根值的功能，以及获得左子树或右子树的功能。

```
def getRootVal(root):
    return root[0]
def setRootVal(root,newVal):
    root[0] = newVal
def getLeftChild(root):
    return root[1]
def getRightChild(root):
    return root[2]
```

代码 6.8 练习了我们刚才编写树的功能。你应该自己尝试一下，其中一个练习要求你画出代码 6.8 中生成的树的结构。

代码 6. 7

```
def BinaryTree(r):
    return [r, [], []]
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch, [], []])
    return root
```

```
def getRootVal(root):
    return root[0]
def setRootVal(root,newVal):
    root[0] = newVal
def getLeftChild(root):
    return root[1]
def getRightChild(root):
    return root[2]
r = BinaryTree(3)
insertLeft(r,4)
insertLeft(r,5)
insertRight(r,6)
insertRight(r,7)
l = getLeftChild(r)
print(l)
setRootVal(l,9)
print(r)
insertLeft(l,11)
print(r)
print(getRightChild(getRightChild(r)))
```

代码 6.8 用于实现树的基本功能的 Python 模块 (bin_tree)

自我测试

1. 执行如下代码后，树的内容为

```
x = BinaryTree('a')
insertLeft(x,'b')
insertRight(x,'c')
insertRight(getRightChild(x),'d')
insertLeft(getRightChild(getRightChild(x)),'e')
```

代码 6. 9

a) ['a', ['b', [], []], ['c', [], ['d', [], []]]]

b) ['a', ['c', [], ['d', ['e', [], []], []]], ['b', [], []]]

c) ['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []]]]

d) ['a', ['b', [], ['d', ['e', [], []], []]], ['c', [], []]]

2. 写一个 buildTree 函数，通过调用上述嵌套列表操作函数，生成如图 6.10 所示的树

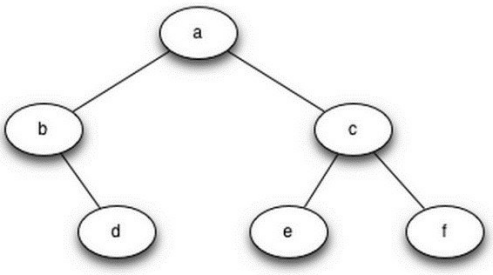


图 6. 7

6.5. 节点和引用

我们第二种实现树的方式使用节点和引用。在这种情况下，我们将定义具有根值，以及左子树和右子树属性的类。由于这种实现方式与面向对象的编程方式联系更紧密，我们将继续使用这种实现方式完成本章的其余部分。

使用节点和引用，我们认为该树的结构可能会类似于图 6.8 所示。

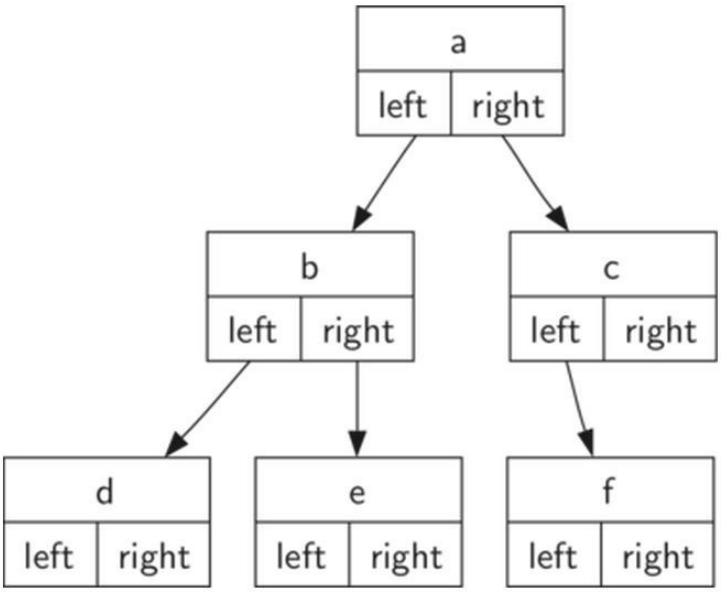


图 6. 8 一个使用节点和引用实现的简单的树

我们将开始使用如代码 6.10 所示的所使用节点和引用方法的简单类定义。重要的是要记住这种方式通过是 left 和 right 属性引用其他 BinaryTree 类实现的。例如，当我们插入一个新的左子节点到树中时，我们创建了另一个 BinaryTree 的实例，并修改了根节点的 self.leftChild 使之指向新的树。

注意到代码 6.10 中，构造函数需要得到一些类型的对象存储在根中。就像你可以在列表中储存

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

代码 6. 10

任何一种你喜欢的类型，树的根对象可以指向任何一种类型。在前面的例子中，我们存储节点的名称作为根值。使用节点和引用来实现树，在图 6.8 中，我们创建了二叉树类的六个实例。

接下来让我们看一下除根节点外需要构建的功能。为了添加左子节点，我们将创建一个新的二叉树对象，并设置根的 left 属性指向这个新对象。insertLeft 的代码如代码 6.11 所示。

```
def insertLeft(self,newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t
```

代码 6. 11

我们必须考虑两种情况进行插入。第一种情况的特征是，没有现有左子节点。当没有左子节点时，简单地将新节点添加到树中即可。第二种情况的特征是，当前存在左子节点。在第二种情况下，我们插入一个节点并将已存在的子节点降级。第二种情况是由代码 6.11 第 4 行的 else 语句所处理的。

insertRight 的代码必须考虑一个对称组的情况。或者没有现有的右子节点，或者我们必须将新节点插入根和现有的右子节点之间。插入的代码如代码 6.12 所示。

```
def insertRight(self,newNode):
    if self.rightChild == None:
        self.rightChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.rightChild = self.rightChild
        self.rightChild = t
```

代码 6. 12

为了完善树简单二叉树的定义，我们将编写几个用于访问左右子节点和根值得方法（参见代码 6.13）

```
def getRightChild(self):
    return self.rightChild
def getLeftChild(self):
    return self.leftChild
def setRootVal(self,obj):
    self.key = obj
def getRootVal(self):
    return self.key
```

代码 6.13

既然我们已经有了所有创建和操作二叉树的方法，让我们再进一步检查它的结构。让我们生成一个简单的树，以节点 a 为根节点，并添加节点 B 和 C 作为子节点。代码 6.14 创建了树， 并存储一些值在 key，left，right 中。注意，根节点的左右子节点本身就是 BinaryTree 类的不同实例。 正如我们在树的原始递归定义中所说的，这使我们能够把一个二叉树的任何子节点当成二叉树本身。

```
    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t
    def getRightChild(self):
        return self.rightChild
    def getLeftChild(self):
        return self.leftChild
    def setRootVal(self,obj):
        self.key = obj
    def getRootVal(self):
        return self.key
r = BinaryTree('a')
print(r.getRootVal())
print(r.getLeftChild())
r.insertLeft('b')
print(r.getLeftChild())
print(r.getLeftChild().getRootVal())
r.insertRight('c')
print(r.getRightChild())
print(r.getRightChild().getRootVal())
r.getRightChild().setRootVal('hello')
print(r.getRightChild().getRootVal())
```

自我测试

代码 6. 13 练习使用节点和引用的实现方式（bintree）

3. 写一个 buildTree 函数，使用节点和引用方法，生成如图 6.9 所示的二叉树

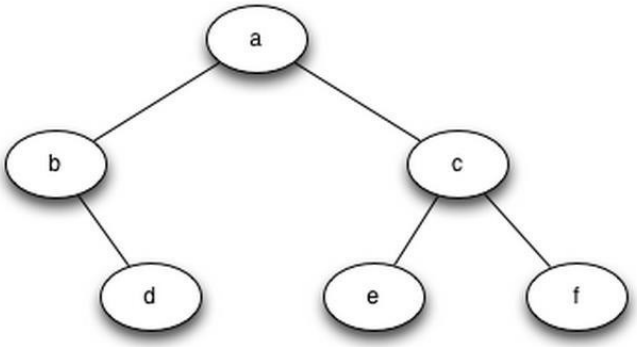


图 6. 9

6.6 解析树

在实现了树(Tree)数据结构之后，现在我们来看一个例子，它将告诉你怎么样利用树(Tree)去解决一些实际的问题。在这个章节，我们将关注一些解析树。解析树可以用来呈现例如句子或者数学达式等真实世界中的结构。

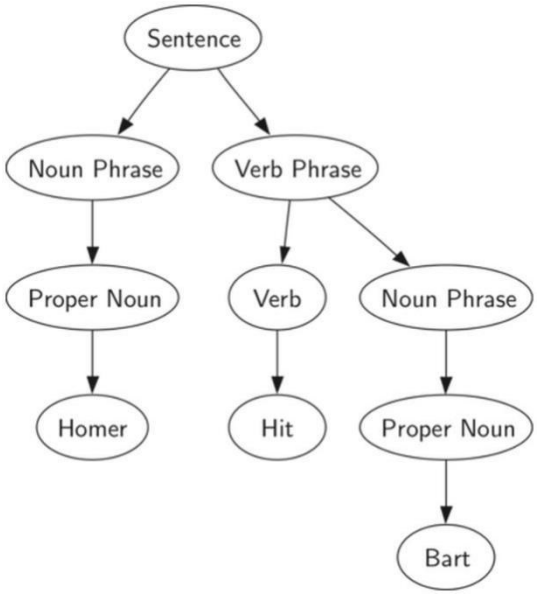


图 6.10 一个简单句子的解析树

图 6.10 显示了一个简单句子的层次结构。将一个句子表征为一个树(Tree)的结构,能使我们通过利用子树来处理句子中的每个独立成分。

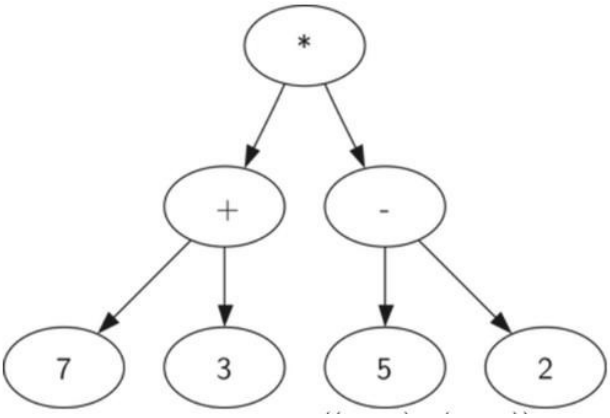


图 6.11 ((7+3)*(5-2)的解析树

如图 6.11 所示，我们能将(7+3)*(5-2)这样一个数学表达式表示成一个解析树。我们在见了这个全括号表达式之后,会怎样理解这个表达式呢？我们知道乘法比加法和减法有着更高的优先级。而因为表达式中的括号，我们知道在做乘法运算之前,需要先计算括号内的加法和减法表达式。树的层次结构就能帮助我们理解整个表达式的运算顺序。在计算最上层的乘法运算前，我们先要计算子树中的加法和减法。左子树的加法运算结果为 10,右子树的减法运算结果为 3。利用树的层次结构，一旦我们计算出了子节点中表达式的结果，我们就能够将整个子树用一个节点来替换。运用这个替换步骤，我们将会得到一个简化的树，如图 6.12 所示。

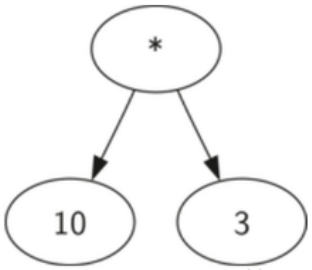


图 6.12 化简后的((7+3)*(5−2))的解析树

在本节的剩余部分，我们将更加详细地讨论解析树。尤其是：

怎样根据一个全括号数学表达式来建立其对应的解析树

怎样计算存在解析树中的数学表达式的值

怎样根据一个解析树恢复出原先的数学表达式

建立解析树的第一步需要将表达式字符串(string)分解成单个字符列表。一共有四种类型的字符：左括号，右括号，操作符和操作数。每当读到一个左括号时,就代表着有一个新的表达式，我们就需要创建一个与之相对应的新的树。相反,每当读到一个右括号时，就代表这一个表达式的结束。另外，操作数将成为叶节点(leaf node)和他们所属的操作符的子节点(children)。最后，每个操作符都应该有一个左子节点和一个右子节点。通过上面的分析我们定义以下四条规则：

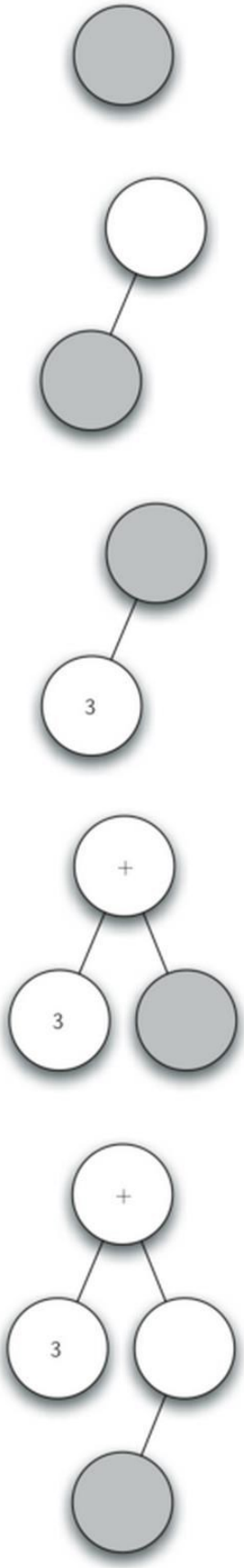
如果当前读入的字符是 `'('` ,添加一个新的节点(node)作为当前节点的左子节点，当前节点下降。

如果当前读入的字符在列表 `['+','-','/','*']` 中，将当前节点的根值设置为当前读入的字符。添加一个新的节点(node)作为当前节点的右子节点，当前节点下降。

如果当前读入的字符是一个数字，将当前节点的根值设置为该数字，当前节点变为它的父节点(parent)。

如果当前读入的字符是 `')'` ，当前节点变为其父节点(parent)。

在我们编写 Python 代码之前,让我们一起先来看一个具体的例子。我们将使用(3+(4*5))这个表达式。我们将表达式分列为如下字符的列表：`['(', '3', '+', '(', '4', '*', '5' ,')',')']` 。一开始，我们将从一个仅包括一个空的根节点的解析树开始。图 6.13 为我们展示了该解析树的内容和结构随着每个新的字符被读入是怎样变化的。


3
+
3
+
3

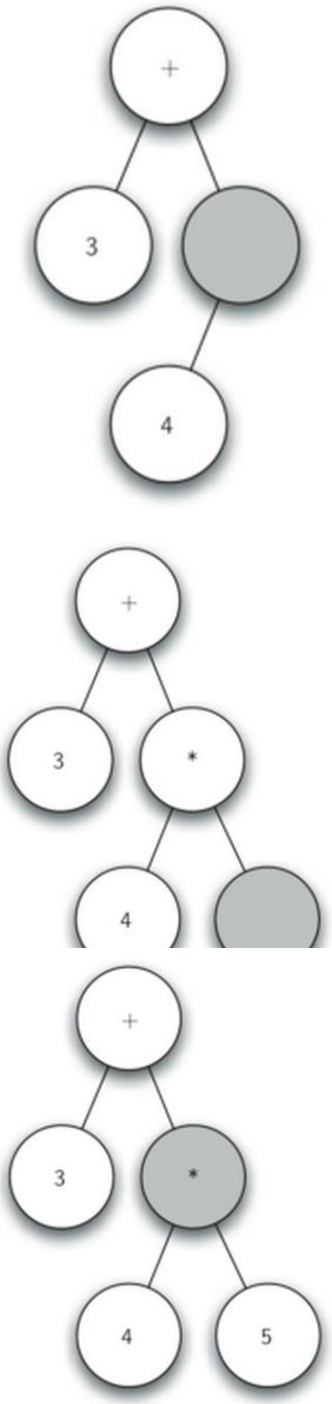


图 6.13 解析树结构的演变图

根据图 6.13 我们可以得知解析树是如何一步步建立起来的：

a.创建一个空的树。

b.读入'('为第一个字符，根据规则 1，创建一个新的节点作为当前节点的左子结点,并将当前 节点变为这个新的子节点。

c.读入'3'为下一个字符。根据规则 3，将当前节点的根值赋值为 3，然后返回当前节点的父 节点。

d.读入'+'为下一个字符。根据规则 2,将当前节点的根值赋值为 +，然后添加一个新的节点 作为其右子节点，并且将当前节点变为这个新的子节点。

e.读入')'为下一个字符。根据规则 1，创建一个新的节点作为当前节点的左子结点，并将当 前节点变为这个新的子节点。

f.读入'4'为下一个字符。根据规则 3，将当前节点的根值赋值为 4，然后返回到当前节点的 父节点。

g.读入'*'为下一个字符。根据规则 2，将当前节点的根值赋值为 *，然后添加一个新的节点 作为其右子节点,并且将当前节点变为这个新的子节点。

h.读入'5'为下一个字符。根据规则 3，将当前节点的根值赋值为 5，然后返回到当前节点的 父节点。

i.读入')'作为下一个字符。根据规则 4，我们将当前节点变为当前节点'*'的父节点。

j.读入')'作为下一个字符。根据规则 4，我们将当前节点变为当前节点'+'的父节点,然而当前 节点没有父节点，所以我们完成了解析树的构建。

从上面的例子可以看出,在构建解析树的过程中我们需要保持对当前节点和其父节点的追踪。而树的连接方式就提供给我们了获得一个节点的子节点的方法—— `getLeftChild` 和 `getRightChild`,但是我们怎么样来追踪一个节点的父节点呢？一个简单的方法就是利用堆栈在遍历整个树的过程中保持对父节点的跟踪。每当我们要下降到当前节点的子节点时，我们先将当前节点压入栈中。而当我们想要返回当前节点的父节点时，我们就能从堆栈中弹出该父节点。

通过上述的规则，使用堆栈(Stack)和二叉树(BinaryTree)操作,我们现在就能编写构建解析树的Python 函数了。解析树生成函数的代码如下所示。

```
from pythonds.basic.stack import Stack
from pythonds.trees.binaryTree import BinaryTree
def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
    for i in fplist:
        if i == '(':
            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()
        elif i not in ['+', '-', '*', '/', ')']:
            currentTree.setRootVal(int(i))
            parent = pStack.pop()
            currentTree = parent
        elif i in ['+', '-', '*', '/']:
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
```

```
        else:
            raise ValueError
    return eTree
pt = buildParseTree("( ( 10 + 5 ) * 3 )")
pt.postorder()  #defined and explained in the next section
```

代码 6.15 解析树的生成函数

四条解析树建立的规则被编写为了四个 if 从句，分别在第 11,15,19,24 行。在这几处你能看到通过调用一些 BinaryTree 和 Stack 的方法而实现这些规则的代码。在这个函数中唯一的差错检查是在 else 语句中，如果我们不能辨认出从列表中读入的字符,那么我们就会报一个 ValueError 的错误。

现在我们已经建立了一个解析树,我们能用它来干什么呢？首先,我们将写一个函数来计算解析树的值，并返回该计算结果。为了实现这个函数,我们将利用树的分层结构。重新看一下图 6.11 回想一下我们能够将原始的树替换为化简后的树(图 6.12)。这提示了我们可以写一个能够通过计算每个子树的值从而算出整个解析树值的递归算法。

就像我们以前实现递归算法那样，我们将从识别基本问题开始来设计递归计算表达式值的函数。这个递归算法的最小基本问题是检查一个操作符是否为叶节点。在解析树中，叶节点总是操作数。因为数字变量如整数和浮点数不需要更复杂的分析，evaluate 函数只需要简单地返回叶节点中储存的数字就可以。使函数朝向基本情形前进的递归过程就是调用 evaluate 函数获取当前节点的左子树、右子树的值。递归调用使我们有效地朝叶节点移动。

为了将两个递归调用的结果整合在一起，我们只需简单地使储存在父节点中的操作符应用到两个子节点返回的运算结果上。在图 6.12 中，我们能看到两个子节点的值分别为 10 和 3。对它们使用乘法运算就能得到最终结果 30。

递归函数 evaluate 的代码如 Listing1 所示。首先，我们会得到当前节点的左子节点、右子节点的参数值。如果左右子节点的值都是 None,那么我们就能知道这个当前节点是一个叶节点。这个检查在第 7 行。如果当前节点不是一个叶节点,那么就会查看当前节点中的操作符,并将它运用到通过递归求值得到的左右子节点结果的计算上来。

为了实现这个算法，我们使用了键值分别为’ +’ ,’ - ‘,’ *’ 和’ /’ 的字典(dictionary)。存在字典里的值是 Python 的操作符模块(operator module)中的函数。这个操作符模块为我们提供了很多常用操作符的函数。当我们在字典中查找一个操作符时，相应的函数功能会被取回。因为这个取回的变量是一个函数,我们按通常函数调用的方式来调用它们，如函数名(变量 1, 变量 2)。所以查找操作符 ‘+’ (2,2)就等价于 operator.add(2,2)。

```
def evaluate(parseTree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.truediv}

    leftC = parseTree.getLeftChild()
    rightC = parseTree.getRightChild()

    if leftC and rightC:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(leftC),evaluate(rightC))
    else:
        return parseTree.getRootVal()
```

代码 6.16 解析树的生成函数

最后，我们将通过图 6.13 创建的解析树对 evaluate 函数进行具体分析。当我们第一次调用 evaluate 函数时，我们传递了整个解析树的根节点作为 parseTree 参数。然后我们获得了其左右子树的参数值，并得知它们的存在。递归调用发生在第 9 行。我们从查看根节点中的操作符开始——

'+'。' +' 操作符对应了有两个参数变量的函数 operator.add。通常对一 Python 函数调用而言。Python 第一件做的事情就是评估传递给函数的参数。这里，这两个参数都是 evaluate 函数的递归调用值。通过从左到右的求值过程,第一个递归调用从左边开始。在第一个递归调用中，evaluate 函数被赋予了左子树的值。我们发现这个节点没有左、右子节点,所以我们在一个叶节点上。当我们在叶节点上时，我们返回这个叶节点存储的数值作为求值的结果即可。因此我们返回整数 3。

现在，为了完成顶层调用 operator.add 函数的计算，我们已经有了其中一个参数了。但是我们还没有全部完成。继续从左到右计算参数,现在递归调用用来计算根节点的右子节点。我们发现这个节点既有左子节点又有右子节点，所以我们查找这个节点中存储的操作符为 '*' ，然后调用明白这个操作符函数并将它的左右子节点作为函数的两个参数。此时,递归调用都将去往叶节点，各自的值分别为整数 4 和 5。得到这两个参数值后，我们返回 operator.mul(4,5)的值。此时，我们已经计算好了顶层操作符 '+' 的两个操作数了，我们所需要做的只是完成调用函数 operator.add(3,20)即可。这个结果就是整个表达式树(3+(4*5))的值，为 23。

6.7 树的遍历

之前我们已经了解了树数据结构的基本功能了，现在我们来看一看树的一些其他使用模式。这些使用模式按照访问节点的方式不同可以分为 3 种。这是三种访问树中的所有节点的常用模式,它们之间的不同在于访问每个节点的次序不同。我们称这种对所有节点的访问为遍历(traversal)。这三种遍历分别叫做**前序遍历(preorder)**，**中序遍历(inorder)**和**后序遍历(postorder)**。我们先来给出它们的具体定义，然后举例看看它们的应用。

前序遍历(preorder)：在前序遍历中，我们先访问根节点，然后递归地前序遍历访问左子树，再递归地前序遍历访问右子树。

中序遍历(inorder)：在中序遍历中，我们递归地中序遍历访问左子树，然后访问根节点，最后再递归地中序遍历访问右子树。

后序遍历(postorder)：在后序遍历中，我们先递归地后序遍历访问左子树和右子树，最后访问根节点。

现在我们用几个例子来说明这三种不同的遍历方式。首先我们先看看前序遍历。我们通过用树结构呈现一本书的例子来看看前序遍历的方式。书是树的根节点，每一章是根节点的子节点，每一节是章的子节点，每一小节是每一节的子节点,以此类推。图 6.14 是一本只取了两章的书。虽然遍历的算法适用于含有任意多子节点的树结构，但暂时我们只关注二叉树。

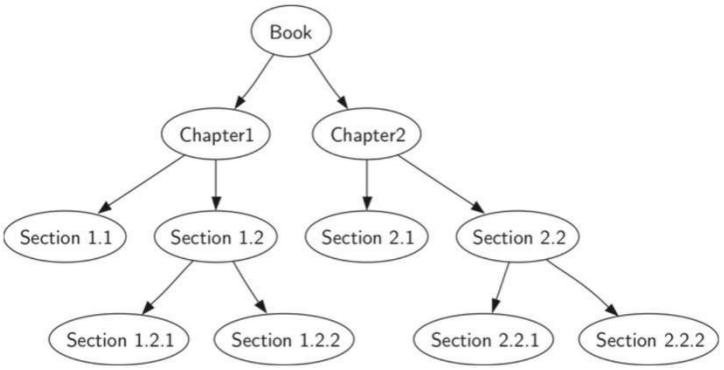


图 6.14 用树结构来表示一本书

设想你要从头到尾阅读这本书。前序遍历恰好符合这种顺序。从根节点(书)开始我们按照前序遍历的顺序来阅读。我们递归地调用 preorder 到其左子节点,在这里是第一章，我们继续递归地调用 preorder 到左子节点第一节 1.1。第一节 1.1 没有子节点,我们不再递归下去。当我们阅读完 1.1 节后我们回到第一章，这时我们还需要递归地访问第一章的右子树 1.2 节。由于我们先访问左子树，

所以我们先看 1.2.1 节，再看 1.2.2 节。当 1.2 节读完后,我们又回到第一章。之后我们再返回根节点(书)然后按照上述步骤访问第二章。

由于用递归来编写，前序遍历的代码异常的简洁优美。代码 6.17 给出了一个二叉树前序遍历的 Python 代码。

你可能想知道，什么是写类似前序遍历算法的最佳方式？应该是简单地使用作为数据结构的树，还是应该作为树数据结构的一种方法？表代码 6.17 展示了一个需要二叉树作为参数的作为外部函数的前序遍历的算法版本。由于我们的基本情况是简单地检查是否存在树，这个外部函数就显得非常简介优美。如果树参数为 None，则函数就会返回而不采取任何行动。

```
def preorder(tree):
    if tree:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
```

代码 6.17 二叉树的前序遍历函数

我们也可以让 preorder 作为二叉树类中的内置方法，这部分代码如代码 6.18 所示。注意这一代码从外部移到内部所产生的变化。概括的说,我们我们只是将 “tree” 换成了 “self” 。但是我们也要修改代码基本情况。内置的方法在递归调用 preorder 之前必须检查左右子节点是否存在。

```
def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()
```

代码 6.18 二叉树类中的前序遍历方法

内置和外置这两种方法哪种更好一些呢？一般来说外置函数更好。原因在于你很少需要单纯遍历整个树。多数情况下你只是想利用基本的遍历方法来实现其他的事情。事实上我们马上就会看到后序遍历的算法和我们之前写的解析树求值的代码关系紧密。因此我们接下来将都按照外部函数的形式书写遍历的代码。.

后序遍历的代码如代码 6.19 所示,它除了将 print 语句移到末尾之外和前序遍历的代码几乎一样。

```
def postorder(tree):
    if tree != None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print(tree.getRootVal())
```

代码 6.19 二叉树的后序遍历函数

我们已经见过了后序遍历的一种一般应用，也就是解析树求值。我们再来看代码 6.15 ，先求左子树的值，再求右子树的值，然后再利用操作符的函数调用将它们在根节点处结合。假设我们的二叉树只储存表达式树的数据。我们可以改写求值函数并尽量模仿后序遍历的代码代码 6.18 ），代码 6.20 所示。

```
def postordereval(tree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
        if res1 and res2:
            return opers[tree.getRootVal()](res1,res2)
        else:
            return tree.getRootVal()
```

代码 6.20 采用后序遍历法重写表达式求值代码

我们发现代码 6.20 的形式和代码 6.19 是一样的，区别在于在代码 6.19 中我们输出键而在代码 6.20 中 我们返回键。这使我们可以通过第 6 行和第 7 行将递归得到的值储存起来。之后我们利用这些保存起来的值和第 9 行的运算符一起运算。

本节的最后一种遍历方式为中序遍历。在中序遍历中，我们先访问左子树，之后是根节点，最后访问右子树。代码 6.21 给出了中序遍历的代码。我们发现这三种遍历的函数代码只是改变了 print 语句相对于递归函数调用的位置。

```
def inorder(tree):
  if tree != None:
      inorder(tree.getLeftChild())
      print(tree.getRootVal())
      inorder(tree.getRightChild())
```

代码 6.21 二叉树的中序遍历函数

如果我们对解析树进行一个简单的中序遍历，我们将得到没有圆括号的原始表达式。我们可以修改基础的中序遍历的算法使我们复原全括号表达式。只要做如下修改：在递归访问左子树之前输出左括号，然后在访问右子树之后输出右括号。修改的代码见代码 6.22 。

```
def printexp(tree):
    sVal = ""
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild())+')'
    return sVal
```

代码 6.22 采用中序遍历递归算法来生成全括号中缀表达式

我们发现 printexp 函数对每个数字也加了括号，这些括号显然没必要加。在本章最后的练习中会有练习要求你修改 printexp 函数来去除这些括号。

6.8 二叉堆 BINARY HEAP 实现的优先队列

在前面的章节里我们学习了“先进先出”(FIFO)的数据结构：队列(Queue)。队列一种重要的变体叫做**“优先队列”(Priority Queue)**。优先队列的出队(Dequeue)操作和队列一样，都是从队首出队。但在优先队列内部,数据项的次序是由它们的“优先级”来确定的:有最高优先级的数据项排在队首,而优先级最低的数据项则排在队尾。这样，优先队列的入队(Enqueue)操作就可能需要将数据项挤到队列前方。我们将会发现,对于下一章要学的一些图算法，优先队列是很有用的一种数据结构。

你可能很自然地会想到用排序函数和队列的一些简单方法来实现优先队列。但是，在列表里插入一个数据项的复杂度是 O(n)，列表排序的复杂度是 O(nlogn)。我们可以用别的方法来降低复杂度。一个实现优先队列的经典的方法便是采用**二叉堆(Binary Heap)**数据结构。二叉堆能将优先队列的入队和出队复杂度都保持在 O(logn).。

二叉堆的有趣之处在于，其逻辑结构用图形表示很像二叉树，但我们却可以仅仅用一个列表来实现它。二叉堆通常有两种:最小成员 key 排在队首的称为**“最小堆(min heap)”**，反之，最大 key 排在队首的是**“最大堆(max heap)”**。在这一节里我们会使用最小堆。我们把最大堆的使用留作练习题。

6.8.1 二叉堆操作

数据结构二叉堆的基本操作定义如下：

- BinaryHeap():创建一个新的空二叉堆对象
- insert(k):加入一个新数据项到堆中
- findMin():返回堆中的最小项,最小项仍保留在堆中
- delMin():返回堆中的最小项,同时从堆中删除
- isEmpty():返回堆是否为空
- size():返回堆中数据项的个数
- buildHeap(list):从一个 key 列表创建新堆

如代码 6.23 所示是一些二叉堆方法使用的示例。可以看到无论我们以哪种顺序把数据项添加到堆里，每次都会移除最小的数据项。我们接下来就要实现这种想法。

```
from pythonds.trees.binheap import BinHeap
bh = BinHeap()
bh.insert(5)
bh.insert(7)
bh.insert(3)
bh.insert(11)
print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
```

代码 6.23 二叉堆方法使用的示例

6.8.2 二叉堆实现

6.8.2.1 二叉堆的结构性质

为了使堆操作高效运行，我们将利用二叉树的操作复杂度为对数级这一性质来实现堆操作。同时，为了使堆操作的复杂度始终保持在对数水平上，就必须始终保持二叉树的“平衡”。平衡的二叉树树根左右子树有着相同数量的节点。在我们的堆实现中，我们采用“**完全二叉树**”的结构来近似实现“平衡”。完全二叉树，指每个内部节点都有两个子节点，最多可有一个节点例外。图 6.15 所示是一个完全二叉树。

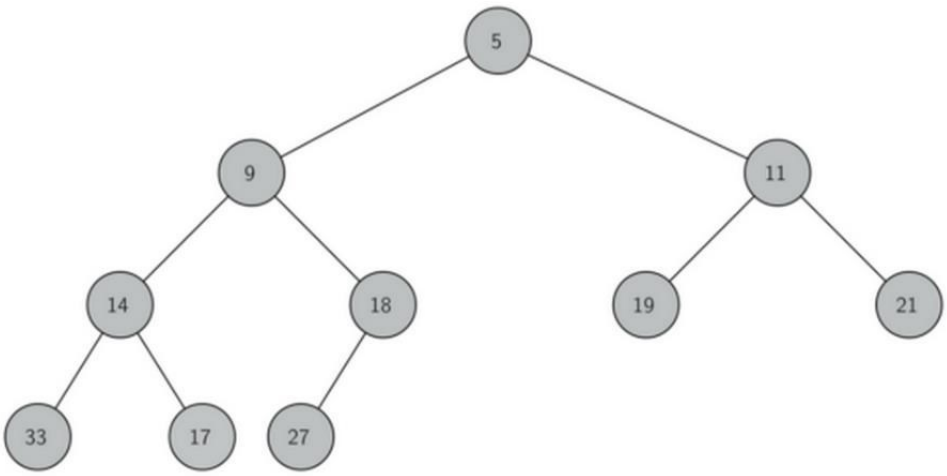


图 6.15：完全二叉树

完全树的另一个有趣的特性是我们能用单个列表来实现完全树而不需要使用节点，引用或嵌套列表。因为对于完全树，如果节点在列表中的位置为 p，那么其左子节点的位置为 2p，类似的，其右子节点的位置为 2p+1。当我们要找任意节点的父节点时，可以直接利用 python 的整数除法。若节

点在列表中的位置为 n，那么父节点的位置是 n//2。图 6.16 所示是一个完全树和树的列表表示，注意位置在 2p 和 2p+1 的节点与其父节点和子节点的关系。完全树的列表表示和整个数据结构的性质，让我们能够使用简单的数学方法高效地遍历一个完全树。这也带来了二叉堆的高效实现。

6.8.2.2 堆的次序性

我们用来在堆里储存数据项的方法依赖于维持堆次序。所谓**堆次序**，是指堆中任意一个节点 x，其父节点 p 中的 key 均小于或等于 x 中的 key。图 6.16 也给出了一个具有堆次序性的完全树。

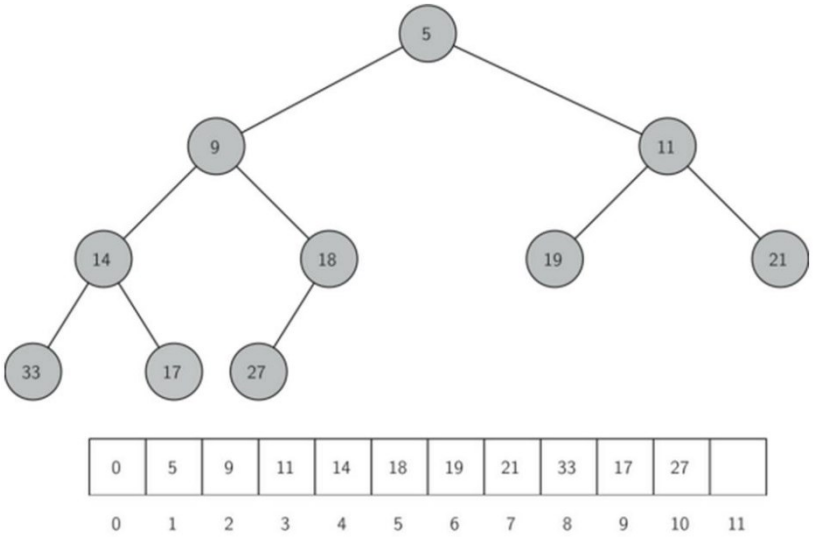


图 6.16：完全树和其列表表示

6.8.2.3 二叉堆操作的实现

接下来我们开始利用构造函数来实现二叉堆。因为可以采用一个列表来保存堆数据，构造函数仅仅需要初始化一个列表和一个 currentSize 来跟踪记录堆当前的大小。代码 6.24 所示是是构造二叉堆的 python 代码。其中表首下标为 0 的项并没有用到，但为了后面代码可以用到简单的整数乘除法，仍保留它。

代码 6.24

```
class BinHeap:
    def __init__(self):
```

接下来我们要实现的是 insert（key）方法。往列表中添加数据项最简单也最高效的方式是直接将数据项添加到列表末尾。这样做虽然保持了完全树的性质，但无法保持堆的次序性。不过，我们可以通过比较新加入的数据项和父节点的方法来恢复堆的次序性。如果新加入的数据项比父节点要小，可以把它与父节点互换位置。图 6.17 所示是一系列交换操作来使新加入的数据项“上浮”到正确位置。

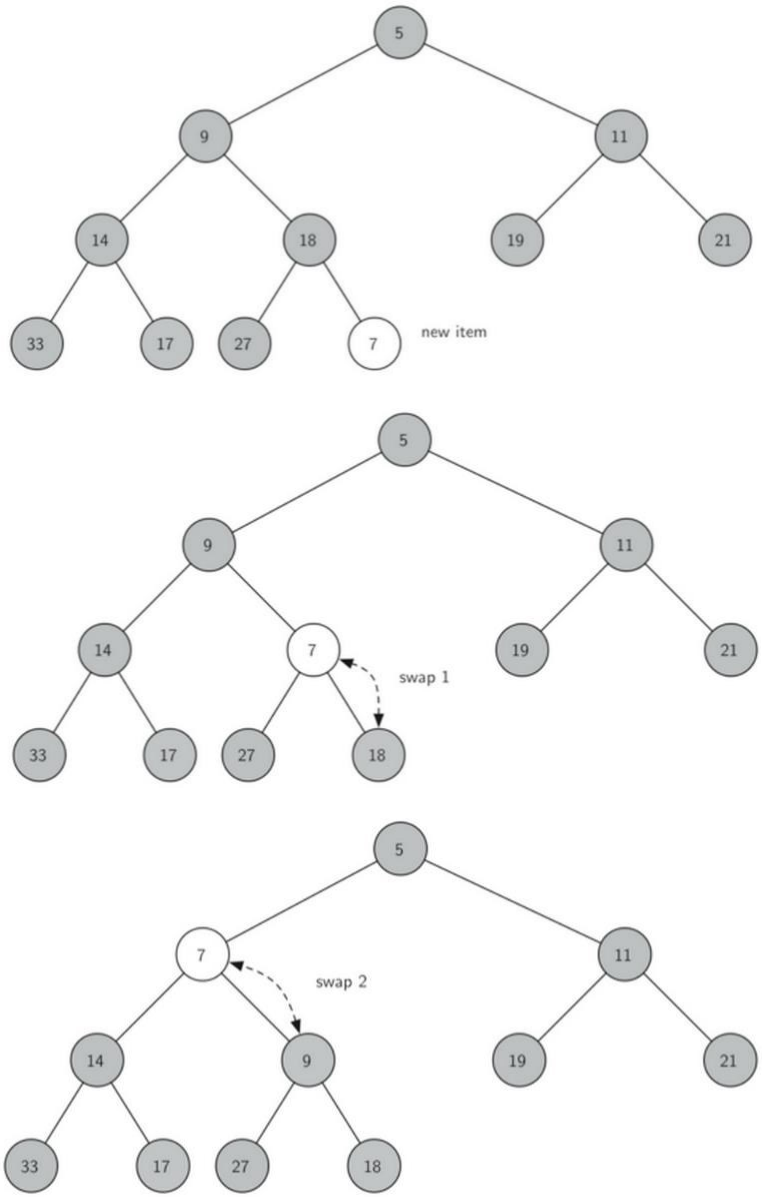


图 6.17：新节点“上浮”到其正确位置

当我们让一个数据项“上浮”时，我们保证了新节点与父节点以及其他兄弟节点之间的堆次序。当然，如果新节点非常小，我们可能仍需要把它向上交换到另一个层级。事实上，我们需要不断交换，直到到达树的顶端。代码 6.25 所示是“上浮”方法，它把一个新节点“上浮”到其正确位置来保持堆次序。这里很好地体现了我们在 heapList 中没有用到的元素 0 的重要性。只需要做简单的整数除法:将当前节点的下标除以 2，我们就能计算出任何节点的父节点。

如图 6.17，我们已经可以写出 insert（key）方法的代码。insert（key）方法很大一部分是由 percUp 函数完成的（见代码 6.26）。当树添加新节点时，调用 percUp 就可以将新节点放到正确的位置上。

```
def percUp(self,i):
    while i // 2 > 0:
      if self.heapList[i] < self.heapList[i // 2]:
          tmp = self.heapList[i // 2]
          self.heapList[i // 2] = self.heapList[i]
          self.heapList[i] = tmp
      i = i // 2
```

代码 6.25

```
def insert(self,k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.percUp(self.currentSize)
```

代码 6.26

既然已经定义了 insert（key）方法，我们来看看 delMin（）方法。堆次序要求根节点是树中最小的数据项，因此很容易找到最小项。比较困难的是移走根节点的数据项后如何恢复堆结构和堆次序。我们可以分两步走。首先，用最后一个节点来代替根节点。移走最后一个节点维持了堆结构的性质。如此简单的替换，还是可能破坏堆次序。这就要用到第二步:将新节点“下沉”来恢复堆次序。图 6.18 所示是一系列交换操作来使新节点“下沉”到正确位置。

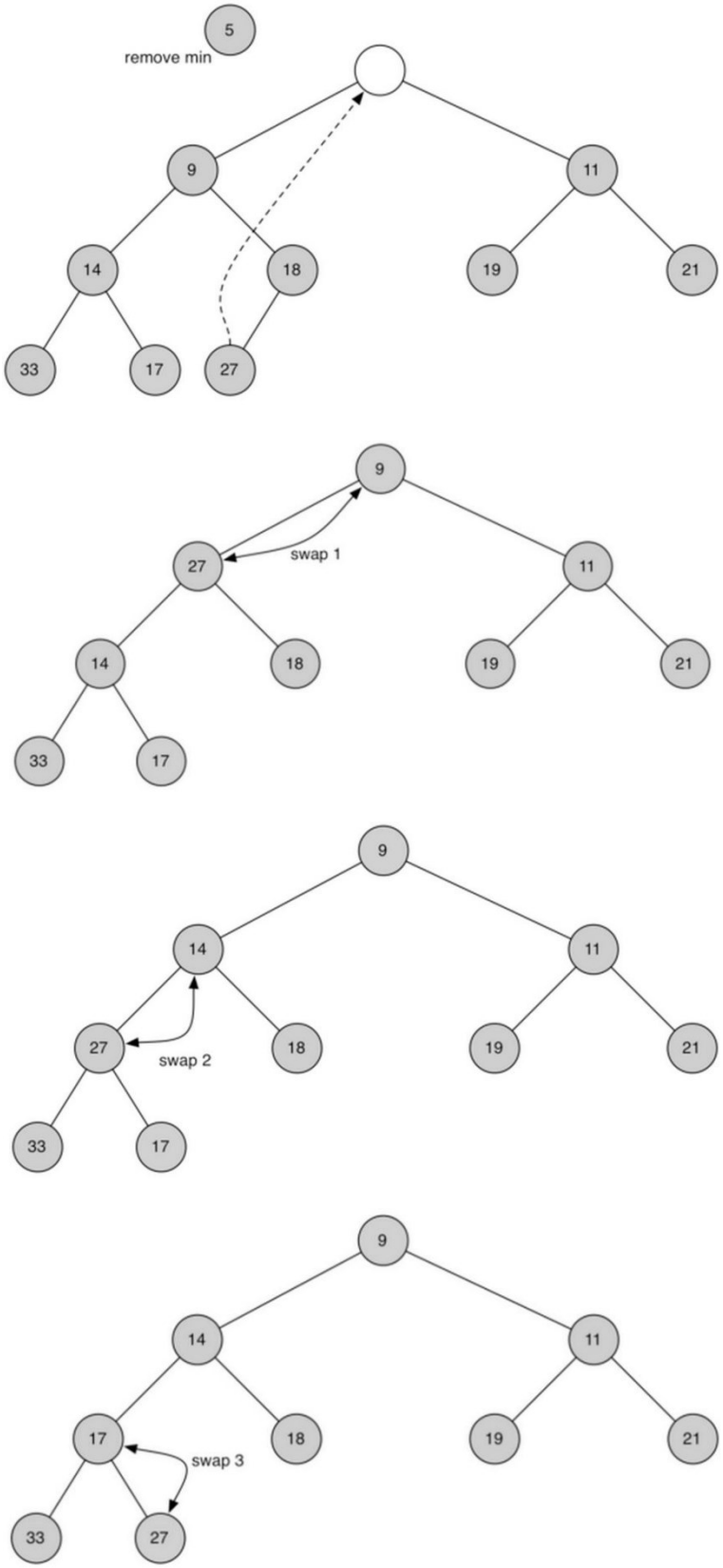

5
remove min
9
11
14
18
19
21
33
17
27
9
swap 1
27
11
14
18
19
21
33
17
9
14
11
27
swap 2
18
19
21
33
17
9
14
11
17
swap 3
18
19
21
33
27

图 6.18：节点“下沉”

为了维持堆的次序性，我们只需要交换根节点和比根节点小的最小子节点。首次交换后，我们可能需要不断循环交换过程，直到新的根节点比两个子节点都小。代码 6.27 所示是下降新节点所需的 percDown 和 minChild 方法的代码。

代码 6.27

```
def percDown(self,i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] > self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
        i = mc

def minChild(self,i):
    if i * 2 + 1 > self.currentSize:
        return i * 2
    else:
        if self.heapList[i*2] < self.heapList[i*2+1]:
            return i * 2
        else:
            return i * 2 + 1
```

```
def delMin(self):
    retval = self.heapList[1]
    self.heapList.pop()
    self.percDown(1)
```

```
    self.heapList[1] = self.heapList[self.currentSize]
    self.currentSize = self.currentSize -1
```

代码 6.28

代码 6.28 所示是 delMin（）操作的代码。可以看到困难的部分由一个辅助函数处理，即 percDown。

有关二叉堆的最后一部分便是找到方法从无序表生成一个“堆”。我们最自然的想法是：用 insert（key）方法，将无序表中的数据项逐个插入到堆中。对于一个排好序的列表，我们可以用二分搜索找到合适的位置来插入下一个 key，操作复杂度是 O(logn).然而插入一个数据项到列表中间需要将列表其他数据项移动为新节点腾出位置，操作复杂度是 O(n).因此用 insert（key）方法的总代价是 O(nlogn)。其实，我们能直接将整个列表生成堆，将总代价控制在 O（n）。代码 6.29 所示是生成堆的操作代码。

代码 6.29

```
def buildHeap(self,alist):
    i = len(alist) // 2
    self.currentSize = len(alist)
    self.heapList = [0] + alist[:]
    while (i > 0):
        self.percDown(i)
        i = i -1
```

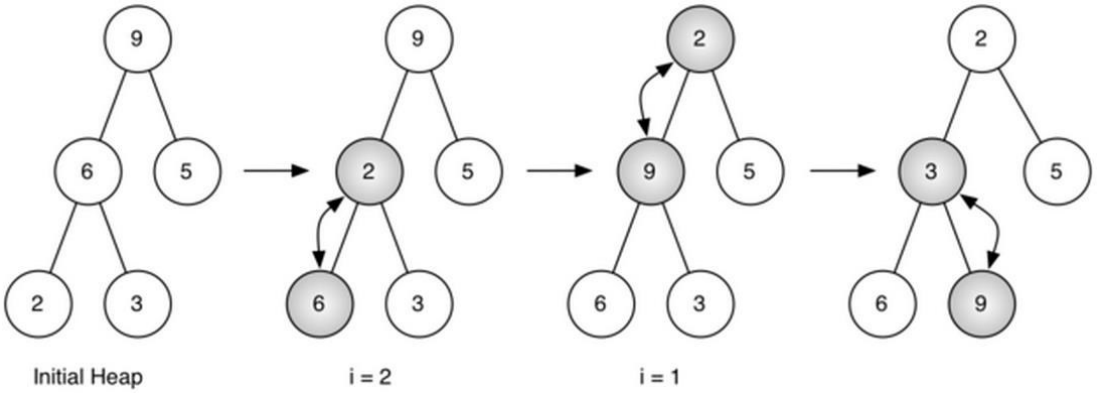


图 6.19：从列表[9,6,5,2,3]生成一个二叉堆

图 6.19 所示是 buildHeap（list）方法将初始树[9, 6, 5, 2, 3]中的节点移动到正确的位置时所做的交换操作。尽管我们从树中间开始，然后回溯到根节点，但 percDown 方法保证了最大子节点总是“下沉”。因为堆是完全树，任何经过中间点的节点都是叶节点，因此没有子节点。注意，当 i=1 时，我们从根节点开始下沉，这就需要大量的交换操作。可以看到，图 6.19 最右边的两颗树，首先 9 从根节点的位置上移走，移到下一层级之后，percDown 进一步检查它此时的子节点，保证它下降到不能下降为止，即下降到正确的位置。这就导致了第二次交换：9 和 3 的交换。由于 9 已经移到了树的最底层，便无法进一步交换了。比较一下图 6.19 所示的一系列交换的列表表示和树表示是很有用的。

```
i = 2    [0, 9, 5, 6, 2, 3]
i = 1    [0, 9, 2, 6, 5, 3]
i = 0    [0, 2, 3, 6, 5, 9]
```

动态代码 6.30 所示是完全二叉堆的实现。

```
    def insert(self,k):
```

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
    def percUp(self,i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i // 2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2
```

```
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i*2] < self.heapList[i*2+1]:
                return i * 2
            else:
                return i * 2 + 1
    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval
```

```
    def buildHeap(self,alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1
bh = BinHeap()
bh.buildHeap([9,5,6,2,3])

print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
```

动态代码 6.30：完全二叉堆示例（completeheap）

这里需声明：能在 O(n)的代价下生成二叉堆看起来可能有点神秘，并且其证明超出了本书的范围。但是，要理解用 O(n)的代价能生成堆的关键是记住 logn 因子来源于树的高度。而对于 buildHeap 方法中许多操作来说，树的高度比 logn 要小。

既然我们已经能在时间复杂度为 O(n)情况下从列表生成堆，那么在本章的最后我们可以做一个练习：用堆来构建一个排序算法来对列表进行复杂度为 O(nlogn)的排序。

6.9 二叉搜索树

我们已经看到在一个集合中得到键值对的两种不同的方法。回忆一下这些集合是如何实现 ADT MAP 的。这两种我们讨论的 ADT MAP 的实现方式是列表的二分查找和散列表。在这个部分，我们将要学习**二叉搜索树**作为从键指向值的另一种方式，在这种情形中我们对数据在树中的实际位置不感兴趣，但是我们对用二叉树结构来提供更有效率的搜索感兴趣。

6.9.1 搜索树操作

在我们看这种实现方式之前，让我们回顾一下 ADT MAP 提供的接口。我们会发现，这种接口和 Python 的字典非常相似。

- Map() 建立一个新的空 map。
- put(key,val) 在 map 中增加了一个新的键值对。如果这个键已经在这个 map 中了，那么就用新的数据来代替旧的数据。
- get(key) 提供一个键，返回 map 中保存的数据，否则返回 None。.
- del 用 del map[key]这种形式从 map 中删除键值对。
- len() 返回 map 中保存的键值对的数目。
- in 对给定的键是否存在 map 中进行判断，如果所给的键在 map 中，返回 True

6.9.2 搜索树实现

一个二叉搜索树，如果左子树中键值 Key 都小于父节点，而右子树中键值 Key 都大于父节点，我们将这种树称为 **BST 搜索树**。如上节所述，当我们实现 Map 方法时，BST 方法将引导我们实现这一点。图 6.20 显示了二叉搜索树的这一特性，显示的键没有任何关联的值。注意：这种属性适用于每个父节点和子节点。所有在左子树的键值是小于根的键值的，所有的键值在右子树均大于根。

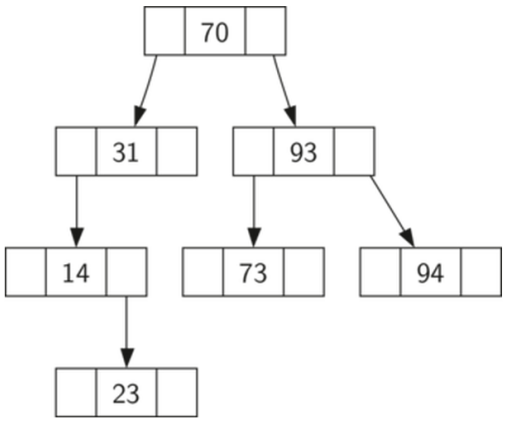


图 6.20 一个由 Python 数据对象构成的双端队列

既然你已经知道什么是二叉搜索树了，让我们来看看二叉搜索树如何实现。在我们按顺序插入70，31，93，94，14，23，73的键后，图1所示搜索树代表了存在的节点。因为70是被插入到树的第一个键，它变成了根。接下来，31小于70，因此是70的左子节点。接下来，93大于70，因此是70 的右子节点。我们现在有两个层次的子树填充，所以接下来的重点，将会是31或者93的左右子树。由于94大于70和93，所以成为93的右子树。同样，14小于70和31，因此成为31的左子节点。23也小于31，因此必须在31的左子树中。然而，它大于14，因此成为14的右子节点。

实现二叉搜索树，我们将使用节点和引用方法，这类似于一个我们用来实现链表和表达式树的过程。但是，因为我们必须能够创建和使用一个空二叉搜索树，所以我们的实现将使用两个类： 第一个类我们称为BinarySearchTree，第二个类我们称之为TreeNode。BinarySearchTree类有一个引用指向TreeNode即二叉搜索树的根。在大多数情况下，外部方法中要定义一个函数， 以便在类外看看子树是否是空的，如果在子树上有节点，要求有基本的方法，在BinarySearchTree类中，把根定义为一个参数。在树是空或者我们想删除根节点键值的情况下， 我们就必须采取特别行动。BinarySearchTree类的建立以及一些其他方面的方法代码如代码6.31所示：

```
class BinarySearchTree：
    def __init__(self)：
        self.root = None
        self.size = 0
    def length(self)：
        return self.size
    def __len__(self)：
        return self.size
    def __iter__(self)：
        return self.root.__iter__()
```

代码 6. 31

TreeNode类提供了许多辅助函数，使BinarySearchTree类的方法更容易做到。TreeNode类的构造，以及这些辅助函数的代码如代码6.32所示。正如你在表中看到的，上述的这些辅助函数，可以根据节点的位置辨别该节点属于何类子节点（左或右），以及该节点的子节点属于何类子节点（左或右）。TreeNode类也将明确地为每个节点追踪对父节点的引用。当我们讨论del操作符的实现时，你将看到为什么这是重要的。

TreeNode类实现的另一个有趣方面是，我们使用了Python中的可选参数，如代码6.32所示。可选参数让我们很容易在不同的情况下创建一个TreeNode，有时我们想要创建一个新的TreeNode带有一个父节点和一个子节点。就现存的父节点和子节点，我们可以把它们作为参数。其余时候我们只会创建一个带键-值对的TreeNode，而不传入任何参数。在这种情况下，我们使用可选参数的缺省值。

```
    def replaceNodeData(self,key,value,lc,rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
            if self.hasRightChild():
self.rightChild.parent = self


        def isLeftChild(self):
            return self.parent and self.parent.leftChild == self
        def isRightChild(self):
            return self.parent and self.parent.rightChild == self
        def isRoot(self):
            return not self.parent
        def isLeaf(self):
            return not (self.rightChild or self.leftChild)
        def hasAnyChildren(self):
            return self.rightChild or self.leftChild
        def hasBothChildren(self):
            return self.rightChild and self.leftChild
```

代码 6. 32

既然我们有了BinarySearchTree类shell和TreeNode，现在让我们写put方法以创建二叉搜索树。put方法是一个BinarySearchTree类的方法。它将检查树是否已经有根节点。如果没有，put将创建一个新的TreeNode并把它作为树的根节点，作为子树的根。如果一个根节点已经到位，我们就调用它自己，进行递归，用辅助功能_put按下列算法来搜索树：

- 从树的根开始搜索，比较新的键值，如果新的键值是小于当前节点。搜索左子　树，如果新的键值是大于当前节点，搜索右子树。 ☒
- 当无左（或右）子树的搜索，我们发现的位置就是应该在子树中安装新节点的位置。 ☒
- 向树添加一个节点，在上一步发现插入对象的位置创建一个新的TreeNode。 ☒

代码6.33显示，在树中插入新节点的Python代码。_put函数按照上述的步骤递归编写。注意，当一个新的子节点插入时，CurrentNode作为父节点传给新的树。对插入重复键值处理不恰当是我们插入实现的一个问题。因为树实现过程中，对于重复的键值将创建一个与原来的节点具有相同键值的右子树作为具有原始键值的节点的子节点。这样做的结果是带有新键值的节点在搜索过程中将永远不会被找到。更好处理重复键值插入问题的方法是用新key指向的value替换掉旧有的value。我们把修复这个bug作为练习留给大家。

随着put方法的定义，我们可以很容易地重载[ ]作为操作符，通过添加 __setitem__的方法来调用put方法。这使我们能够编写比如myZTiptree['plymouth’] = 55446一样的python语句，这看上去就像Python字典。

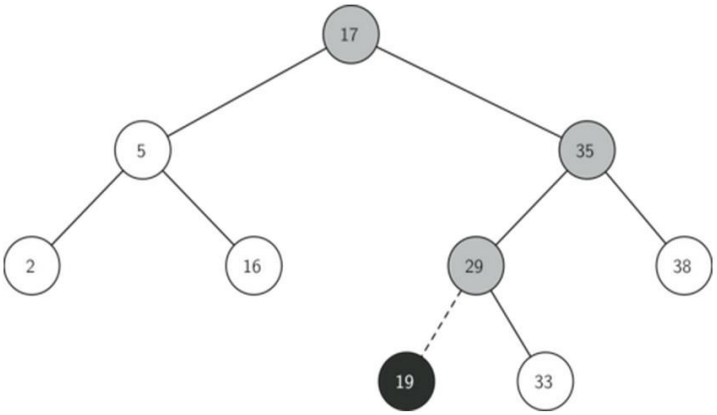
```
def put(self,key,val)：
    if self.root：
        self._put(key,val,self.root)
    else：
        self.root = TreeNode(key,val)
        self.size = self.size + 1
    def _put(self,key,val,currentNode)：
        if key < currentNode.key：
            if currentNode.hasLeftChild()：
                self._put(key,val,currentNode.leftChild)
            else：
                currentNode.leftChild = TreeNode(key,val,parent=currentNode)
    else：
        if currentNode.hasRightChild()：
```

代码 6. 33

```
def __setitem__(self,k,v)：
self.put(k,v)
```

代码 6. 34

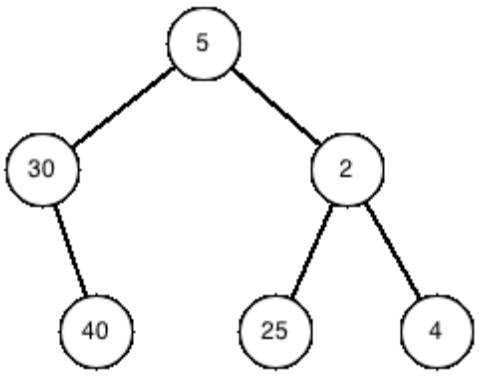
图6.21显示了用于插入新节点到一个二叉搜索树的过程。如下的灰色节点显示了这个插入过程中所遍历到的树的节点。



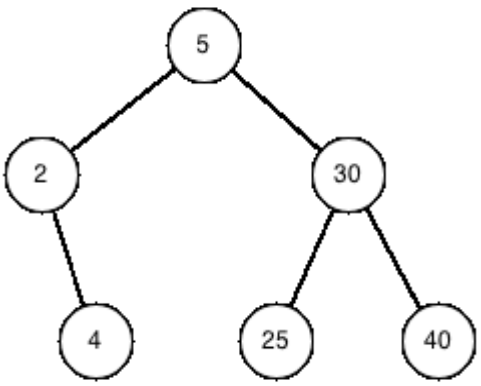
☒图 6.21 插入一个节点键值 Key= 19

自我检测

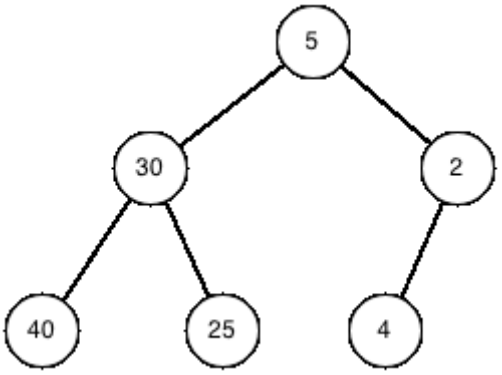
4. 如下哪个树正确地显示了按顺序插入键值5，30，2，40，25，4后的二叉搜索树？



a）



b）



c）

一旦树被构建，接下来的任务就是给定键实现对值的检索。get方法比put更容易，因为它递归地搜索子树，直到发现无匹配的叶节点或找到一个匹配的键。当一个匹配的键被发现后，存储在节点的值被返回。

```
def get(self,key)：
    if self.root：
        res = self._get(key,self.root)
        if res：
            return res.payload
        else：
            return None
    else：
        return None

def _get(self,key,currentNode)：
    if not currentNode：
        return None
    elif currentNode.key == key：
        return currentNode
    elif key < currentNode.key：
        return self._get(key,currentNode.leftChild)
    else：
        return self._get(key,currentNode.rightChild)

def __getitem__(self,key)：
    return self.get(key)
```

代码 6.35

代码6.35显示了代码get，_get和__getitem__。_get方法的搜索代码，使用_put方法中选择左或右子节点的逻辑。请注意，使用_get方法返回一个TreeNode给get，这使得_get成为一个灵

活的辅助方法，对于其他的BinarySearchTree方法可能需要利用TreeNode中除了负载以为的其他数据。

通过实施__getitem__方法，我们可以写若干Python语句，使它们看起来就像我们访问字典一样，而事实上我们只是在用一个二叉搜索树，例如Z = myziptree ['fargo']。如你可以看到，所有的__getitem__方法都是调用get。

调用get函数时，我们可以为BinarySearchTree写__contains__方法从而能够使用操作符in。__contains__会简单的调用get，当get返回值的时候就返回True，如果get返回None就返回False。__contains__方法代码如表 6 所示：

代码 6.36

```
def __contains__(self,key):
    if self._get(key,self.root):

if 'Northfield' in myZipTree:
    print("oom ya ya")
```

回顾__contains__重载了操作符in，这允许了我们写这样的语句：

代码 6.37

最后，我们来关注在二叉搜索树中最具挑战性的方法，删除一个（参见列表7）。首要任务是找到要删除的搜索树的节点。如果树有一个以上的节点，我们使用_get方法搜索找到需要删除的TreeNode;如果树只有一个节点，这意味着我们要移除树的根，但是我们仍然必须检查以确保根的键是否匹配要删除的键。在以上两种情况下，如果未发现键值，del操作符就会报错。

```
def delete(self,key):
    if self.size > 1:
        nodeToRemove = self._get(key,self.root)
```

```
            if nodeToRemove：
                self.remove(nodeToRemove)
                self.size = self.size - 1
            else：
                raise KeyError('Error, key not in tree')
        elif self.size == 1 and self.root.key == key：
            self.root = None
            self.size = self.size - 1
        else：
            raise KeyError('Error, key not in tree')

def __delitem__(self,key)：
    self.delete(key)
```

代码 6.38

一旦我们发现含有要删除的键的节点，有三种情况，我们必须考虑：

1. 要删除的节点没有子树（见图三）.
2. 要删除的节点没有子树（见图四）.
3. 要删除的节点没有子树（见图五）.

第一种情况是简单的（参见代码6.39）。如果当前节点没有子树，所有我们需要做的是删除该节点并把指向该节点的引用移动给其父节点。本例的代码显示在这里：

```
if currentNode.isLeaf()：
    if currentNode == currentNode.parent.leftChild：
        currentNode.parent.leftChild = None
    else：
        currentNode.parent.rightChild = None
```

代码 6.39

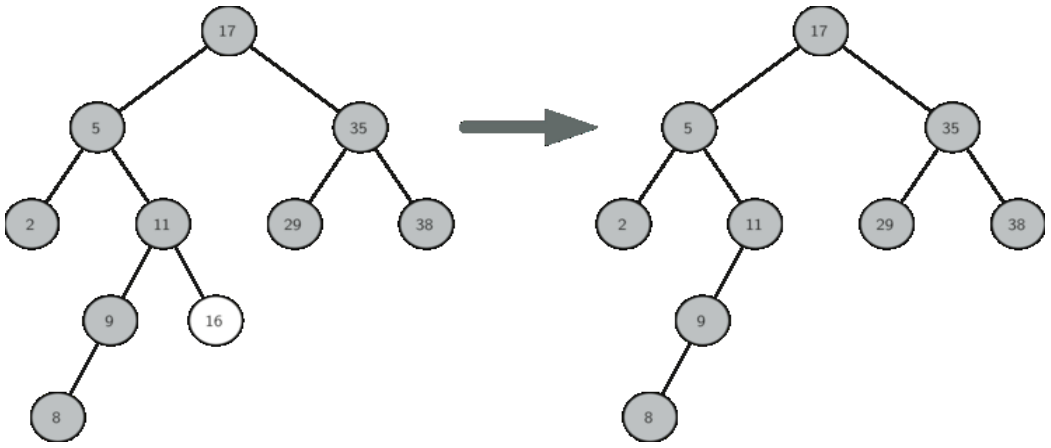


图 6.22 删除节点16，而这个节点没有子树的情况

第二种情况只是稍微复杂（参见代码6.40）。如果一个节点只有一个子节点，那我们可以提升子树以代替其父树的位置。在这种情况下，代码如下表所示。你看这段代码，就会看到有六种情况考虑。由于有一个左或右子树的情况是对称的，我们将只讨论在当前节点有左子树的情况下，然后做对称对称。决策过程如下：

1.如果当前节点是左子节点，那我们只需要更新当前节点的左子节点指向当前节点的父节点引用，然后将父节点对左子节点的引用更新到当前节点的左子节点。

2.如果当前节点是一个右节点，那我们只需要更新当前节点的左子节点指向当前节点的父节点引用，然后将父节点对右子节点的引用更新到当前节点的左子节点。

3.如果当前节点没有父树节点，它必须是根节点。在这种情况下，我们只需更换键，有效载荷，左子节点和右子节点，方法是调用根节点的replaceNodeData方法。

```
else：  # this node has one child
    if currentNode.hasLeftChild()：
        if currentNode.isLeftChild()：
            currentNode.leftChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.leftChild
            elif currentNode.isRightChild()：
                currentNode.leftChild.parent = currentNode.parent
```

```
                currentNode.parent.rightChild = currentNode.leftChild
        else：
            currentNode.replaceNodeData(currentNode.leftChild.key,
                                        currentNode.leftChild.payload,
                                        currentNode.leftChild.leftChild,
                                        currentNode.leftChild.rightChild)
    else：
        if currentNode.isLeftChild()：
            currentNode.rightChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.rightChild
        elif currentNode.isRightChild()：
            currentNode.rightChild.parent = currentNode.parent
            currentNode.parent.rightChild = currentNode.rightChild
        else：
            currentNode.replaceNodeData(currentNode.rightChild.key,
                                        currentNode.rightChild.payload,
                                        currentNode.rightChild.leftChild,
                                    currentNode.rightChild.rightChild)
```

代码 6.40

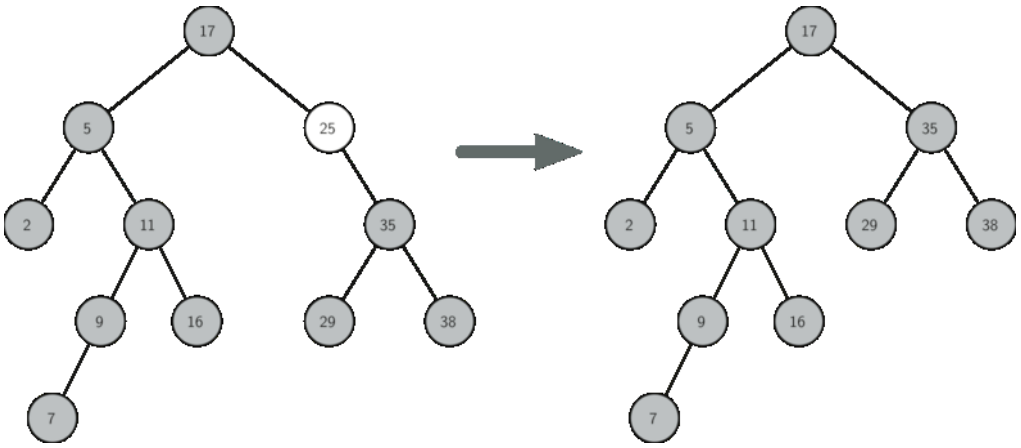


图 6.23 删除节点25，它是只有一个子节点的节点

第三种情况是最难处理的情况（参见代码6.41）。如果一个节点有两个子节点，我们不可能简单地其中一个作为提升至父节点的位置，这就需要寻找一个节点，用来代替一个计划删除的节点，我们需要的这个节点，需要保存现有的左、右子树以及二叉搜索树关系。符合该要求的节点有树中第二大的键。我们称这个节点为继任者，然后我们将一路寻找继任者，继任者保证没有一个以上的子

节点，所以我们知道如何用已经知道的两种情况实现它。一旦继任者被移动，我们把它放在将被删除的子节点处。

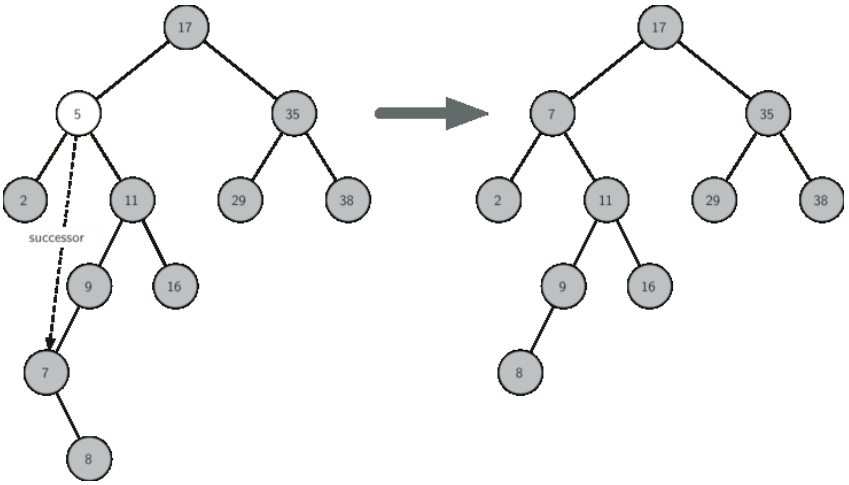


图 6.24 删除节点5，有两个子树的节点

第三种情况代码如下表所示。 注意，我们是用辅助方法 findSuccessor和findMin来找到继任者的。而移动继任者，我们利用方法spliceOut。我们用spliceOut的原因是它能帮助我们直接找到需要分割的节点，做出正确的变化。我们也可以递归调用删除，但那样我们就会浪费时间反复寻找关键节点。

```
elif currentNode.hasBothChildren():      #interior
        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload
```

代码 6.41

找到继任者的代码所示（参见代码6.42），你可以看到一个TreeNode类的方法。这个代码利用二叉搜索子树的中序遍历，按从最小到最大打印出子树中的节点。寻找继任者有三种情况需要考虑：

1. 如果节点有右子节点，那么继任者是在右子树中最小的键。
2. 如果节点没有右子节点，是其父节点的左子节点，那么父节点是继任者。
3. 如果节点是其父节点的右子节点，而本身无右子节点，那么该节点的继任者是其父节点的继任者，不包括这个节点。

当从二叉搜索树中删除节点时，第一种情况是唯一对我们重要的。但findSuccessor方法将在本章最后练习中探索其他用途。

☒findMin 方法是找到一个子树中的最小键。要相信，最小值在任何二叉搜索子树中都是子树的左子树。因此 findMin 方法只简单地追踪左子树，直到找到没有左子树的叶节点。

我们还需要看看二叉搜索树的最后一个接口方法。假设我们想要按顺序简单地遍历树上所有的键值。这是我们用字典所做能的事，为什么不在树也实现呢?我们已经知道如何使用中序遍历二叉树，然而，写一个迭代器需要更多一点的工作，因为每次调用迭代器时，一个迭代器只返回一个节点。

Python提供了一个非常强大的函数yield，使用时创建一个迭代器。yield和return相似，它也返回一个值给调用者。yield也有额外的步骤来记住函数运行目前的状态，以便下次调用函数时，继续从当前状态执行。创建可迭代对象的函数被成为生成器。

二叉树的中序迭代器代码如代码6.43所示。仔细看看这个代码：乍一看，你可能会认为代码是非递归的。但是请记住，__iter__重写for x in的操作符来实现迭代，所以它确实是递归!因为它是对TreeNode进行递归，_iter__在TreeNode类中定义。

```
def findSuccessor(self):
    succ = None
    if self.hasRightChild():
        succ = self.rightChild.findMin()
    else:
        if self.parent:
            if self.isLeftChild():
                succ = self.parent
```

```
                else：
                    self.parent.rightChild = None
                    succ = self.parent.findSuccessor()
                    self.parent.rightChild = self
        return succ

def findMin(self)：
        current = self
        while current.hasLeftChild()：
            current = current.leftChild
        return current

def spliceOut(self)：
        if self.isLeaf()：
            if self.isLeftChild()：
                self.parent.leftChild = None
else：
        self.parent.rightChild = None
        elif self.hasAnyChildren()：
        if self.hasLeftChild()：
            if self.isLeftChild()：
                self.parent.leftChild = self.leftChild
            else：
                self.parent.rightChild = self.leftChild
                self.leftChild.parent = self.parent
        else：
            if self.isLeftChild()：
                self.parent.leftChild = self.rightChild
```

```
            else：
                self.parent.rightChild = self.rightChild
                self.rightChild.parent = self.parent
def __iter__(self)：
    if self：
        if self.hasLeftChild()：
            for elem in self.leftChiLd：
                yield elem
        yield self.key
        if self.hasRightChild()：
            for elem in self.rightChild：
                yield elem
```

代码 6.42

此刻，你可能想下载BinarySearchTree和TreeNode类的完整代码：

```
class TreeNode：
    def __init__(self,key,val,left=None,right=None,parent=None)：
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self)：
        return self.leftChild

    def hasRightChild(self)：
        return self.rightChild
```

```
def hasRightChild(self):
    return self.rightChild

def isLeftChild(self):
    return self.parent and self.parent.leftChild == self

def isRightChild(self):
    return self.parent and self.parent.rightChild == self

def isRoot(self):
    return not self.parent

def isLeaf(self):
    return not (self.rightChild or self.leftChild)

def hasAnyChildren(self):
    return self.rightChild or self.leftChild

def hasBothChildren(self):
    return self.rightChild and self.leftChild

def replaceNodeData(self,key,value,lc,rc):
    self.key = key
    self.payload = value
    self.leftChild = lc
    self.rightChild = rc
    if self.hasLeftChild():
        self.leftChild.parent = self
    if self.hasRightChild():
            self.rightChild.parent = self
```

```
class BinarySearchTree：
    def __init__(self)：
        self.root = None
        self.size = 0

    def length(self)：
        return self.size

    def __len__(self)：
        return self.size

    def put(self,key,val)：
        if self.root：
            self._put(key,val,self.root)
        else：
            self.root = TreeNode(key,val)
            self.size = self.size + 1
    def _put(self,key,val,currentNode)：
        if key < currentNode.key：
            if currentNode.hasLeftChild()：
                self._put(key,val,currentNode.leftChild)
            else：
                currentNode.leftChild = TreeNode(key,val,parent=currentNode)
        else：
            if currentNode.hasRightChild()：
                self._put(key,val,currentNode.rightChild)
            else：
                currentNode.rightChild = TreeNode(key,val,parent=currentNode)
```

```
def __setitem__(self,k,v):
    self.put(k,v)

def get(self,key):
    if self.root:
        res = self._get(key,self.root)
        if res:
            return res.payload
        else:
            return None
    else:
        return None

def _get(self,key,currentNode):
    if not currentNode:
        return None
    elif currentNode.key == key:
        return currentNode
    elif key < currentNode.key:
        return self._get(key,currentNode.leftChild)
    else:
        return self._get(key,currentNode.rightChild)
def __getitem__(self,key):
    return self.get(key)

def __contains__(self,key):
    if self._get(key,self.root):
        return True
    else:
        return False
```

```
def delete(self,key):
    if self.size > 1:
        nodeToRemove = self._get(key,self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size = self.size - 1
        else:
            raise KeyError('Error, key not in tree')
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size = self.size - 1
    else:
        raise KeyError('Error, key not in tree')
def __delitem__(self,key):
    self.delete(key)

def spliceOut(self):
    if self.isLeaf():
        if self.isLeftChild():
            self.parent.leftChild = None
        else:
            self.parent.rightChild = None
    elif self.hasAnyChildren():
        if self.hasLeftChild():
            if self.isLeftChild():
                self.parent.leftChild = self.leftChild
            else:
                self.parent.rightChild = self.leftChild
                self.leftChild.parent = self.parent
```

```
                else：
                    self.parent.rightChild = self.leftChild
                    self.leftChild.parent = self.parent
            else：
                if self.isLeftChild()：
                    self.parent.leftChild = self.rightChild
                else：
                    self.parent.rightChild = self.rightChild
                    self.rightChild.parent = self.parent

    def findSuccessor(self)：
        succ = None
        if self.hasRightChild()：
            succ = self.rightChild.findMin()
        else：
            if self.parent：
                if self.isLeftChild()：
                    succ = self.parent
                else：
                    self.parent.rightChild = None
                    succ = self.parent.findSuccessor()
                    self.parent.rightChild = self
        return succ

    def findMin(self)：
        current = self
        while current.hasLeftChild()：
            current = current.leftChild
        return current
```

```
def remove(self,currentNode)：
    if currentNode.isLeaf()：      #leaf
        if currentNode == currentNode.parent.leftChild：
            currentNode.parent.leftChild = None
        else：
            currentNode.parent.rightChild = None
    elif currentNode.hasBothChildren()：      #interior
        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload
    else：      # this node has one child
        if currentNode.hasLeftChild()：
            if currentNode.isLeftChild()：
                currentNode.leftChild.parent = currentNode.parent
                currentNode.parent.leftChild = currentNode.leftChild
            elif currentNode.isRightChild()：
                currentNode.leftChild.parent = currentNode.parent
                currentNode.parent.rightChild = currentNode.leftChild
            else：
                currentNode.replaceNodeData(currentNode.leftChild.key,
                                            currentNode.leftChild.payload,
                                            currentNode.leftChild.leftChild,
                                            currentNode.leftChild.rightChild)
    else：
            if currentNode.isLeftChild()：
                currentNode.rightChild.parent = currentNode.parent
                currentNode.parent.leftChild = currentNode.rightChild
            elif currentNode.isRightChild()：
                currentNode.rightChild.parent = currentNode.parent
```

```
                    currentNode.parent.rightChild = currentNode.rightChild
                else：
                    currentNode.replaceNodeData(currentNode.rightChild.key,
                                                currentNode.rightChild.payload,
                                                currentNode.rightChild.leftChild,
                                                currentNode.rightChild.rightChild)


mytree = BinarySearchTree()
mytree[3]="red"
mytree[4]="blue"
mytree[6]="yellow"
mytree[2]="at"

print(mytree[6])
print(mytree[2])
```

6.9.3 搜索树分析

通过当前完成的二叉树的实现， 我们将对我们已经实现的方法进行一个快速的分析。 让我们先看一下 put 这个方法。限制它的执行效果的是二叉树的高度。回想一下语法阶段，树的高度指的是根和底部的叶节点之间的边的数目。 高度作为一种限制因素是因为当我们在树中寻找一个插入节点的合适位置时，我们在每一级至多进行一次比较。

二叉树的高度可能是什么样的呢？对这个问题的答案取决于键是怎么被加到树中的。 如果键是以随机的顺序加到树中的，那么当节点的数目为 **n** 时，树的高度将会大概在 $\log_2 n$。

这是因为键随机地散布的话，大约有一半的节点会比根节点大，另一半会比它小。记住在二叉树中，根上有一个节点，下一级有两个，再下一级有四个。当级数为 **d** 时，这一级的节点数目为 **2d**。 当 **h** 代表树的高度的时候， 一个完美平衡二叉树中节点的总数目是 $2^{h+1}-1$。

一个完美平衡二叉树中左子树和右子树的节点数目是一样多的。当树中有 **n** 个节点时，put 执行的最差结果的算法复杂度是 $O(\log_2 n)$。要注意到这与之前段落里

说的关系是逆运算的关系。所以 $\log_2 n$ 给了我们树的高度，这代表了把一个新节点插入一个合适位置所需要做的 put 的最多次数。

不幸的是，通过插入排序好了的键，建造一个高度为 **n** 的搜索树是可能的。图 6.25 就展示了这种树的一个例子，在这个情形下，这个 put 方法的执行复杂度为 **O(n)**。

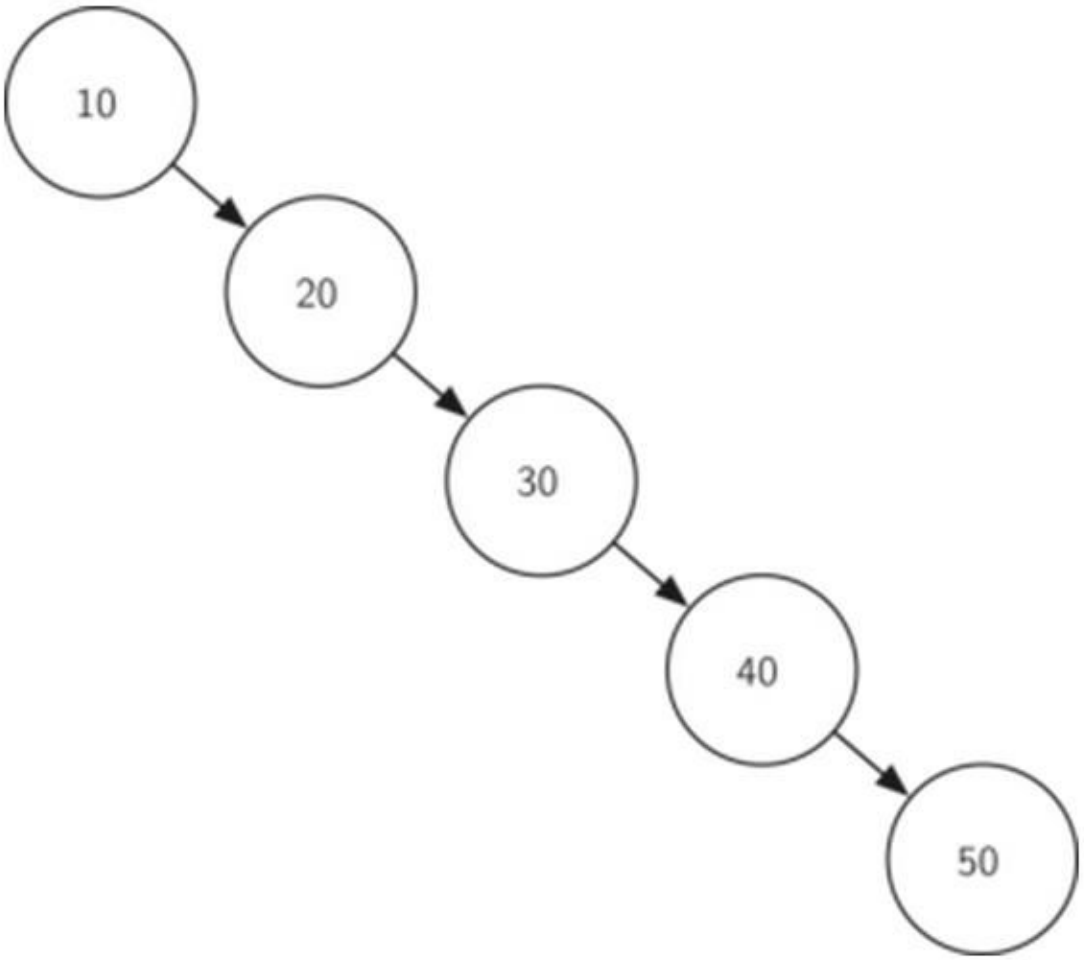


图 6.25 一个倾斜的二叉搜索树会有一个很差的执行效果

现在你已经理解了 put 方法的执行会受到树的高度的限制，你可能猜测其他的方法，get,in 和 del 也是受限制的，因为通过 get 搜索树来发现键，在最坏的情形下，要一直搜索树到底而未找到键。乍一看 del 也许看上去更复杂，因为它也许需要在删除操作完成之前一直查找下一个继承节点。但是记得，寻找继承节点的最坏情形也只是树的高度，这意味着你只需要简单地把工作加倍，因为加倍是乘以一个常数因子，所以它没有改变最坏的情形（时间复杂度）。

6.10 平衡二叉搜索树

在上一节中我们学习了建立一个二叉搜索树。我们知道，当树变得不平衡时 get 和 put 操作会使二叉搜索树的性能降低到 **O(n)** 。在这一节中我们将看到一种特殊的二叉搜索树，它可以自动进行调整，以确保树时时保持平衡。这种树被称为 **AVL** 树，由发明他的人：G.M.Adelson-Velskii 和 E.M.Landis 而命名。

AVL 树实现图（Map）的抽象数据类型，就像一个普通的二叉搜索树，唯一不同的是这棵树的工作方式。为实现 AVL 树我们需要在树中的每个节点加入一个**平衡因子**（balance factor）以跟踪其变化情况。我们通过比较每个节点的左右子树的高度完成比较。更正式的定义是，一个节点的平衡因子定义为左子树的高度和右子树的高度之差。

$$balanceFactor = height(leftSubTree) - height(rightSubTree)$$

利用以上的平衡因子的定义，如果平衡因子小于零，我们称子树“左重” （left-heavy） 。 如果平衡因子小于零，那么子树是“右重” （right-heavy） 。 如果平衡因子是零，树是完美的平衡。为实现 AVL 树的目的，并获得具有平衡的树，我们将定义如果平衡因子是-1，0 或 1,那么这个树是平衡的。一旦树中的节点的平衡因子超出了这个范围， 我们需要将树恢复平衡。图 6.26 是一个不平衡的“右重”树的例子，其中每个节点都标注了平衡因子。

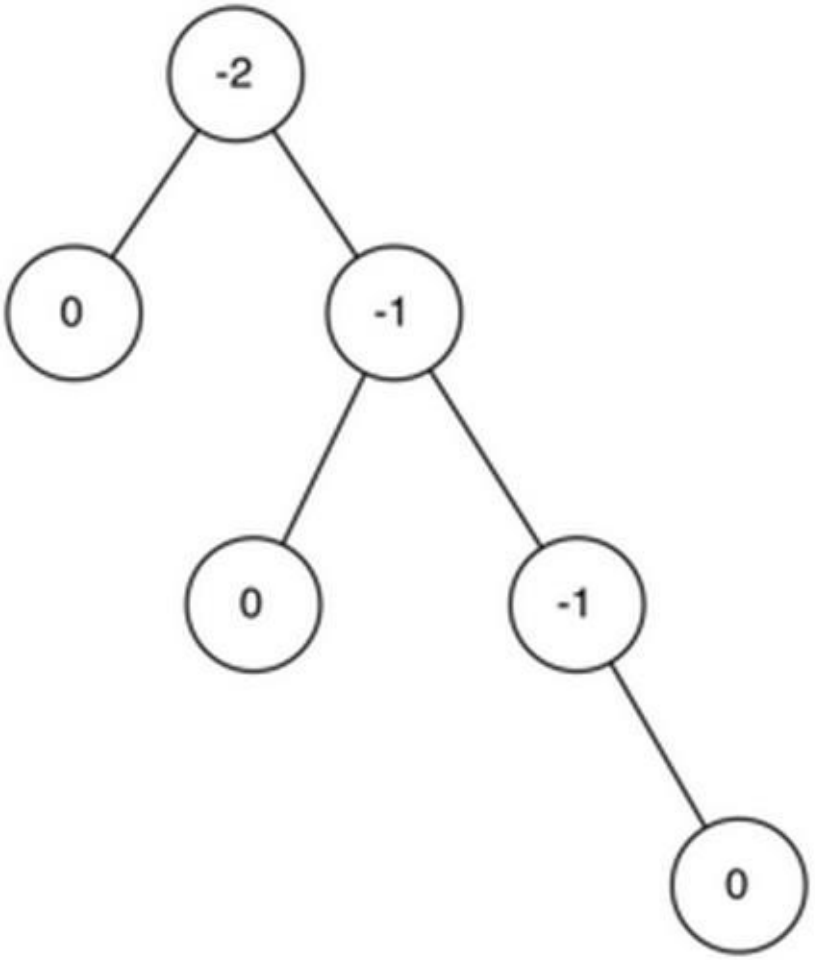


图 6.26：一个平衡因子不平衡的右重树

6.10.1 AVL 树性能

在我们继续进行之前让我们看看引入这个新的平衡因素的结果。我们的需要是， 确保树上总是有一个平衡因子- 1，0，或 1，可以使对键的操作得到更好的大 O 性能。首先，我们来思考有这个平衡条件后，最坏情况下的树发生了什么变化。有两个可能的考虑，左重树和右重树。如果我们考虑树的高度为 0，1，2 和 3，图 6.27 举出了在新规则下可能的出现的最不平衡的左重树的例子。

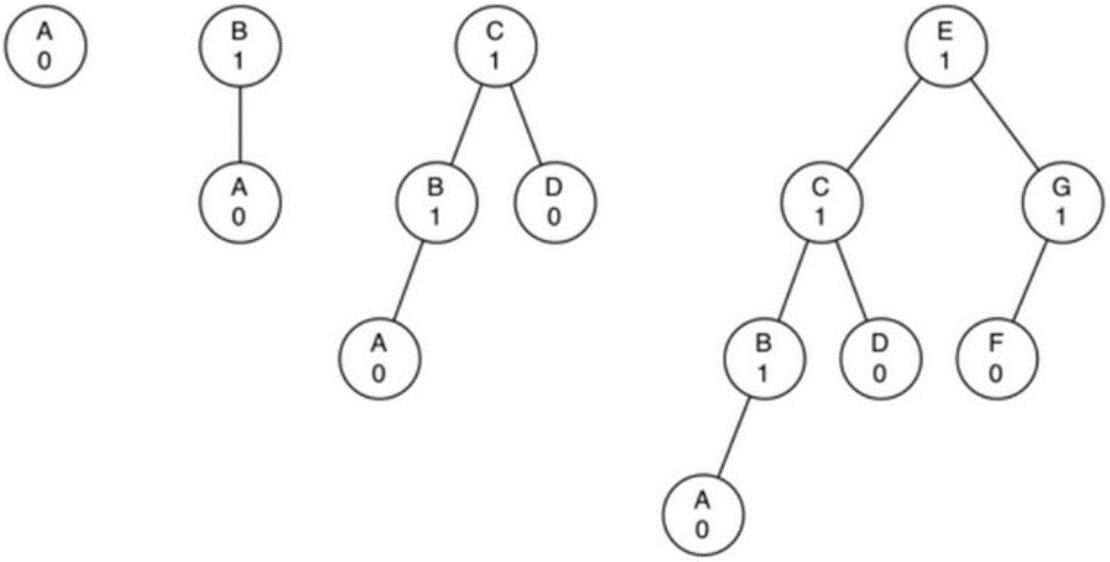


图 6.27：最坏的情况下，左重的 AVL 树

看一下树上的节点的总数，我们就会发现一棵高度为 0 的树有 1 个节点， 一个高度为 1 的树有 **1 + 1 = 2** 个节点，一个高度为 2 的树有 **1 + 1 + 2 = 4**，一棵高度为 3 的树有 **1 + 2 + 4 = 7** 个节点。更普遍的模式，我们看到高度为 h 的树的节点数 **h（N_h）**是：

$$N_h = 1 + N_{h-1} + N_{h-2}$$

可能你很熟悉这个公式， 因为它和斐波那契序列非常相似。 我们可以利用这个公式通过树中的节点的数目推导出一个 AVL 树的高度。在我们的印象中，斐波那契数列和斐波那契数定义为：

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \text{ for all } i \geq 2$$

数学中一个重要的结果是，随着斐波那契序列的数字越来越大 F_i/F_{i-1} 越来越接近于黄金比例**Φ（**$\Phi = \frac{1+\sqrt{5}}{2}$**）** 。如果你想看到上式的推导过程你可以查阅数学教科书。我们将简单地使用这个方程近似 **Fi : Fi =** $\Phi i/\sqrt{5}$ 。如果利用这种近似我们可以将 N_h 的方程改写为：

$$N_h = F_{h+2} - 1, h \geq 1$$

通过用黄金比例近似代替斐波那契数列的项我们可以得到：

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

如果我们整理这些方程的项，并两边以 2 为底取对数，然后求解 h，则可以导出：

$$\log N_h + 1 = (H+2)\log\Phi - \frac{1}{2}\log 5$$

$$h = \frac{\log N_h + 1 - 2\log\Phi + \frac{1}{2}\log 5}{\log\Phi}$$

$$h = 1.44 \log N_h$$

这个推导告诉我们，在任何时候我们的 AVL 树的高度等于树中节点数以 2 为底的对数的常数（等于 1.44） 倍。 这对搜索我们的 AVL 树来说是好消息因为它限制搜索复杂度到 **O(logN)** 。

6.10.2 AVL 树实现

既然，我们已经证明，保持一个 AVL 树的平衡将是一个很大的性能提升，让我们看看我们如何增加向树中插入一个新的键值的算法。因为所有的新键是作为叶节点插入树的， 我们知道一个新叶的平衡因子为零， 所以我们对刚刚插入的节点没有新的要求。 但是一旦有新叶插入我们必须更新其父节点的平衡因子。新的叶节点如何影响父节点的平衡因子取决于该叶节点是左节点还是右节点。如果新节点是右节点，父节点的平衡因子将减少一。如果新节点是左节点，父节点的平衡因子将增加一。这种关系可以递归地应用于新的节点的前两个节点，并且有可能影响每一个前面的节点一直到树的根。 由于这是一个递归过程， 我们可以考察两个更新平衡因子的基本条件：

1. 递归调用已达到树的根。.

2. 父节点的平衡因子已调整为零。显然，一旦子树平衡因子为零，那么父节点的平衡因子就不会改变了。

我们将 AVL 树作为 BinarySearchTree 的子类实现。首先，我们将覆盖 _put 方法写一个新的 updateBalance 辅助方法。这些方法如代码 6.44 所示。除了第 7 行和第 13 行对 updateBalance 的调用，你会注意到 _put 的定义和简单的二叉搜索树是完全相同的。

```
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
            self._put(key,val,currentNode.leftChild)
        else:
            currentNode.leftChild =TreeNode(key,val,parent=currentNode)
            self.updateBalance(currentNode.leftChild)
    else:
        if currentNode.hasRightChild():
            self._put(key,val,currentNode.rightChild)
        else:
            currentNode.rightChild =TreeNode(key,val,parent=currentNode)
            self.updateBalance(currentNode.rightChild)
def updateBalance(self,node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)
        return
```

代码 6.44

_put 的大部分工作正是由这个新的 updateBalance 方法完成的。这实现了我们刚才描述的递归过程。 这个再平衡方法首先检查当前节点是否十分不平衡， 以至于需要重新平衡 （16 行） 。

如果当前节点需要再平衡， 那么只需要对当前节点进行再平衡， 而不需要进一步更新父节点。如果当前节点不需要再平衡， 那么父节点的平衡因子需要调整。 如果父节点的平衡因子非零，那么算法通过父节点递归调用 updateBalance 方法继续往上传递到树的根。

当对一棵树进行再平衡是必要的，我们该怎么做呢？有效的再平衡是使 AVL 树很好地工作而不牺牲性能的关键。 为了让一个 AVL 树恢复平衡， 我们会在树上执行一个或多个 **“旋转”（rotation)。**

为了了解什么是旋转，让我们看一个很简单的例子。思考一下图 6.28 的左半部分树。这棵树是不平衡的，平衡因子为- 2。为了让这棵树平衡，我们将绕根节点 A 的子树节点进行左旋转。

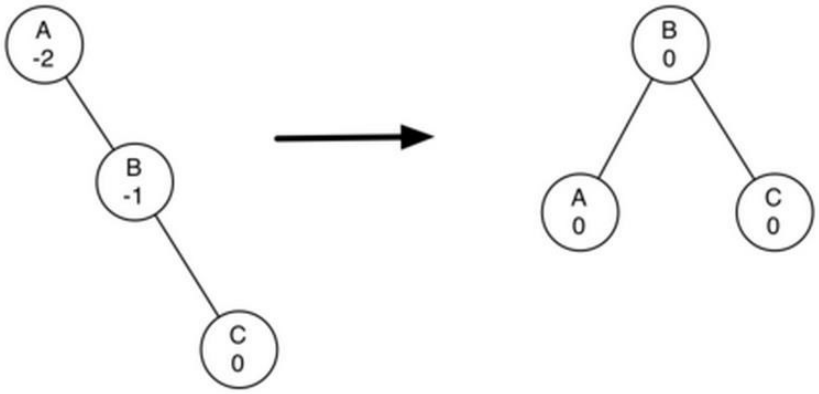


图 6.28：使用左旋转变换不平衡树

执行一个左旋转我们需要做到以下几点：

1. 使右节点（B）成为子树的根。

2. 移动旧根（A）到新根的左节点。

3. 如果新根 （B） 原来有左节点， 那么让原来 B 的左节点成为新根左节点 （A） 的右节点。注：由于新根（B）是 A 的右节点，在这种情况下移动后的 A 的右节点一定是空的。这使得我们不用多想就可以直接给移动后的 A 添加右节点。

虽然这个程序概念上相当简单， 但是代码的细节有点棘手， 因为为了维持二叉搜索树的所有性质，必须以绝对正确的顺序把节点移来移去。此外，我们需要确保正确地更新了所有的 parent 指针。

让我们通过观察一个稍微复杂的树来说明右旋转。图 6.29 的左边展现了一棵 “左重” 的树，根的平衡因子为 2。执行一个正确的右旋转，我们需要做以下几点：

1. 使左节点（C）成为子树的根。

2. 移动旧根（E）到新根的右节点。

3. 如果新根（C）原来有右节点（D） ，那么让 D 成为新根右节点（E）的左节点。注：由于新根（C）是 E 的左节点，在这种情况下移动后的 E 的左节点一定是空的。这使得我们不用多想就可以直接给移动后的 E 添加左节点。

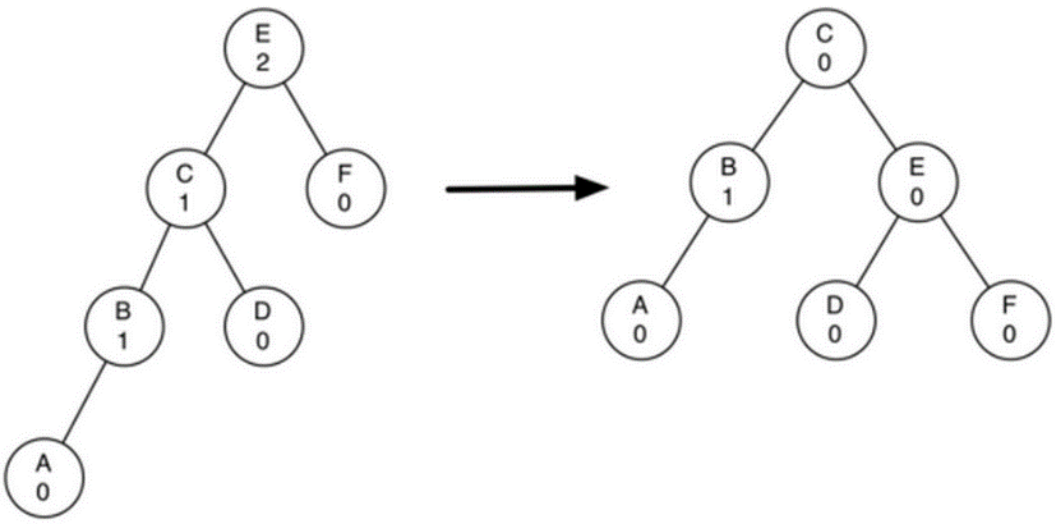


图 6.29：使用正确的旋转变换不平衡树

既然现在你已经看过了旋转的过程，了解了旋转的方法，让我们看看代码。代码 6.45 同时显示了右旋转和左旋转代码。在第 2 行，我们创建一个临时变量来跟踪新的子树的根。正如我们之前所说的新的根是旧根的右节点。现在，右节点已经被存储在这个临时变量。我们将旧根的右节点替换为新根的左节点。

下一步是调整两节点的父指针。如果新根 newroot 原来有左节点，左节点的新父节点变成老根。新根的父节点将成为旧根的父节点。如果旧根是整个树的根，那么我们必须让整棵树的根指向这个新的根。否则，如果旧的根是左节点，那么我们改变左节点的父节点到一个新的根；否则，我们改变右节点的父节点到一个新的根（行 10-13） 。最后我们设置的旧根的父节点成为新的根。这里有很多复杂的中间过程，所以我们建议你一边浏览这个函数的代码，一边看图 6.28。rotateRight 方法和 rotateLeft 是对称的，所以我们会让您自学 rotateRight 代码。

代码 6.45

```
def rotateLeft(self,rotRoot):
    newRoot = rotRoot.rightChild
    rotRoot.rightChild = newRoot.leftChild
    if newRoot.leftChild != None:
        newRoot.leftChild.parent = rotRoot
    newRoot.parent = rotRoot.parent
    if rotRoot.isRoot():
        self.root = newRoot
    else:
        if rotRoot.isLeftChild():
            rotRoot.parent.leftChild = newRoot
        else:
            rotRoot.parent.rightChild = newRoot
    newRoot.leftChild = rotRoot
    rotRoot.parent = newRoot
    rotRoot.balanceFactor = rotRoot.balanceFactor + 1 -
min(newRoot.balanceFactor, 0)
    newRoot.balanceFactor = newRoot.balanceFactor + 1 +
max(rotRoot.balanceFactor, 0)
```

最后，行 16-17 需要一些解释。在这两行我们更新旧根和新根的平衡因子。因为所有其他的动作是移动整个子树， 被移动的子树内的节点的平衡因子不受旋转的影响。 但我们如何在没有完全重新计算新的子树的高度的情况下更新平衡因子？下面的推导将让你明白， 这些代码都是正确的。

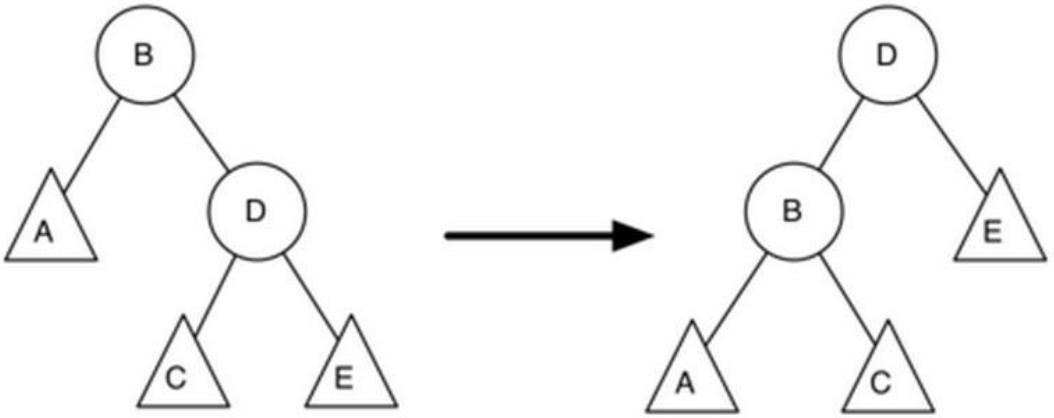


图 6.30：左旋转

图 6.30 显示了一个左旋转。B 和 D 是中心节点，A，C，E 是其子树。让 h_x 表示以 X 为根节点的子树的高度。通过定义我们知道：

$$newBal(B) = h_A - h_C$$
$$oldBal(B) = h_A - h_D$$

但我们知道，D 的旧的高度也可以通过 **1+max(h_C,h_E)**给定，也就是说，D 的高度为两子树高度中较大者加 1。记住，h_C 和 h_E 没有改变。所以，把上式代入第二个方程，可以得到：

$$oldBal(B) = h_A - (1 + max(h_C, h_E))$$

然后两方程作差。下面是作差的步骤，使用了一些代数方法简化方程 **newBal(B)**。

$$newBal(B) - oldBal(B) = h_A - h_C - (h_A - (1 + max(h_C, h_E)))$$
$$newBal(B) - oldBal(B) = h_A - h_C - h_A + (1 + max(h_C, h_E))$$
$$newBal(B) - oldBal(B) = h_A - h_A + 1 + max(h_C, h_E) - h_C$$
$$newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C$$

接下来我们移动 **oldBal(B)** 到方程的右端并利用 **max(a,b)-c=max(a-c,b-c)**

$$newBal(B) = oldBal(B) + 1 + max(h_C - h_C, h_E - h_C)$$

但是，**h_E-h_C**和**− oldBal(D)** 相同。所以我们可以用另一个相同的说法：**max(− a,− b)=− min(a,b)**，所以我们可以通过以下步骤完成对 **newBal(B)** 的推导：

$$newBal(B) = oldBal(B) + 1 + max(0, -oldBal(D))$$
$$newBal(B) = oldBal(B) + 1 - min(0, oldBal(D))$$

我们很容易知道，现在方程所有的项都是已知数。如果我们记得 B 是 rotRoot， D 是 newRoot，我们可以看出这完全符合 16 行的语句，或：

```
rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(0,newRoot.balanceFactor)
```

一个类似的推导给出更新节点 D 的方程，以及右旋转后的平衡因子。我们把这些作为你的小测验。

现在你可能会认为我们已经做完了。 我们知道如何做左右旋转，以及在什么时候做一个向左或向右旋转，但看看图 6.31。由于节点 A 的平衡因子是- 2 我们应该做一个左旋转。但是，当我们在左旋转时会发生什么？

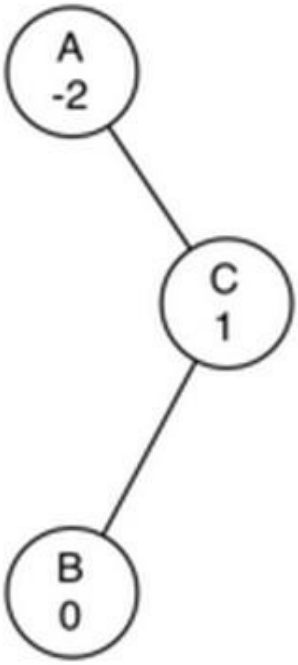


图 6.31：一个更难平衡的不平衡的树

图 6.32 告诉了我们，左旋转后，我们仍然不平衡。如果我们要做一个右旋转来试图再平衡，我们回到原来的状态。

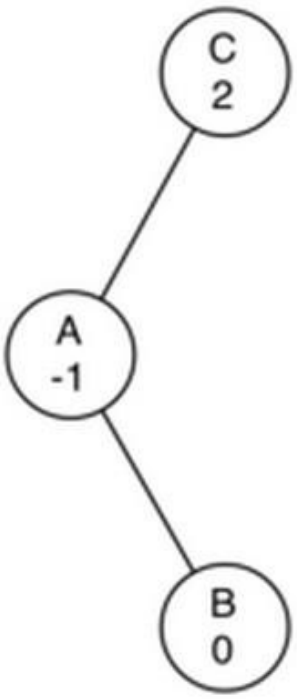


图 6.32：左旋转后树在其他方向不平衡

要纠正这个问题，我们必须使用以下规则：

1. 如果子树需要左旋转使之平衡， 首先检查右节点的平衡因子。 如果右节点左重则右节点做右旋转，然后原节点左旋转。

2. 如果子树需要右旋转使之平衡， 首先检查左节点的平衡因子。 如果左节点右重然后左节点左旋转，然后原节点右旋转。

图 6.33 显示了这些规则如何解决我们在图 6.31 和图 6.32 中遇到的困境。 首先， 以 C 为中心右旋转，使树变成一个较好的形状；然后，以 A 为中心左旋转，整个子树恢复平衡。

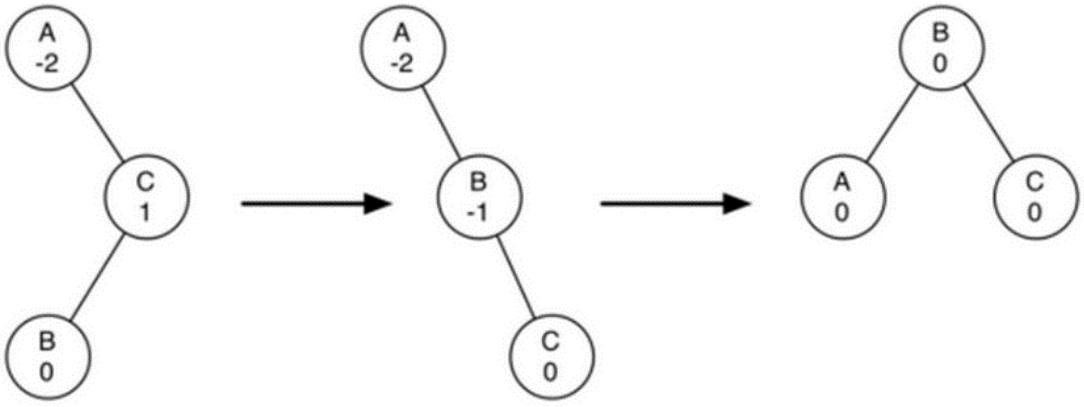


图 6.33：右旋转后左旋转

实现这些规则的代码可以从我们“再平衡”（rebalance）的方法中找到，如代码 6.46 所示。上面的第一条规则从第二行 if 语句中开始实现。第二条规则是由第 8 行 elif 语句开始实现的。

```
def rebalance(self,node):
    if node.balanceFactor < 0:
        if node.rightChild.balanceFactor > 0:
            self.rotateRight(node.rightChild)
            self.rotateLeft(node)
        else:
            self.rotateLeft(node)
```

代码 6.46

本章的“问题讨论”提供了一个需要先左旋转后右旋转的再平衡树的例子。此外，“问题讨论”为你提供机会来平衡一些树，比在图 6.32 中的树更复杂一点。

通过保持树始终平衡，我们可以确保 get 方法运行的时间复杂度为 **$O(\log_2(n))$** 。但问题是 put 方法的时间复杂度是多少？我们把 put 方法分解为 put 的每一步操作。由于每一个新节点都是作为叶节点插入的，每一轮更新所有父节点的平衡因子最多只需要 **$\log_2 n$** 次操作，每一层一次。如果子树是不平衡的最多需要两个旋转把子树恢复平衡。但是，每个旋转的操作是 **O(1)**的复杂度，即使我们 put 操作仍然是 **$O(\log_2(n))$**的复杂度。

现在，我们已经实现了一个能使用的 AVL 树，除非你需要删除一个节点。我们把删除节点和随后的更新和再平衡作为你的练习。

6.11 ADT MAP 实现小结

通过过去两章，我们已经看过了好几种可以用于实现 Map 抽象数据类型的数据结构。列表的二分搜索，哈希表，二叉搜索树和平衡二叉树。作为这一章的结束，让我们总结这些数据结构在 Map 的对键值 key 的操作中的性能（见表 6.1）。

操作	已经排好序的列表	哈希表	二叉搜索树	AVL 树
put	O(n)	O(1)	O(n)	$O(\log_2 n)$
get	$O(\log_2 n)$	O(1)	O(n)	$O(\log_2 n)$
in	$O(\log_2 n)$	O(1)	O(n)	$O(\log_2 n)$
del	O(n)	O(1)	O(n)	$O(\log_2 n)$

表 6.1：比较不同 Map 实现的性能表现

6.12 小结

这一章我们讨论了树的数据结构。 树数据结构使我们能实现很多有意思的算法。 在这章我们已经用树数据结构做了下面这些事情：

- 用二叉树对表达式进行语法分析和求值
- 用二叉树实现 ADT map
- 用平衡二叉树（AVL 树）实现 ADT map
- 用二叉树实现最小堆
- 用最小堆实现优先队列

6.13 关键词

AVL 树	**二叉堆**	**二叉搜索树**
二叉树	**子节点**	**完全二叉树**
边	**堆顺序属性**	**高度**
中序	**叶节点**	**层数**
ADT map	**最大最小堆**	**节点**
父节点	**路径**	**后序**
前序	**优先队列**	**根**
旋转	**兄弟节点**	**后继**
子树	**树**	

6.14 问题讨论

1. 根据下列对树函数的调用，画出相应的树结构：

```
>>> setRootVal(r,9)
>>> r = BinaryTree(3)
>>> insertLeft(r,4)
[3, [4, [], []], []]
>>> insertLeft(r,5)
[3, [5, [4, [], []], []], []]
>>> insertRight(r,6)
[3, [5, [4, [], []], []], [6, [], []]]
>>> insertRight(r,7)
[3, [5, [4, [], []], []], [7, [], [6, [], []]]]
```

2. 创建(4*8)/6−3 的表达式树，然后画出该算法的流程。

3. 考虑以下整数列表：[1,2,3,4,5,6,7,8,9,10]。通过插入方法生成该列表的树，请画出该二叉树。

4. 考虑以下整数列表：[10,9,8,7,6,5,4,3,2,1]。通过插入方法生成该列表的树，请画出该二叉树。

5. 生成一个随机整数列表。通过一次插入列表中一个整数的方法生成二叉堆树，请画出该二叉堆树。

6. 根据前面问题的列表，通过将列表作为 buildHeap 方法的参数生成二叉堆树，请画出该二叉堆树。

7. 按照顺序插入下列 key 值：68,88,61,89,94,50,4,76,66 和 82，画出生成的二叉搜索树的结果。

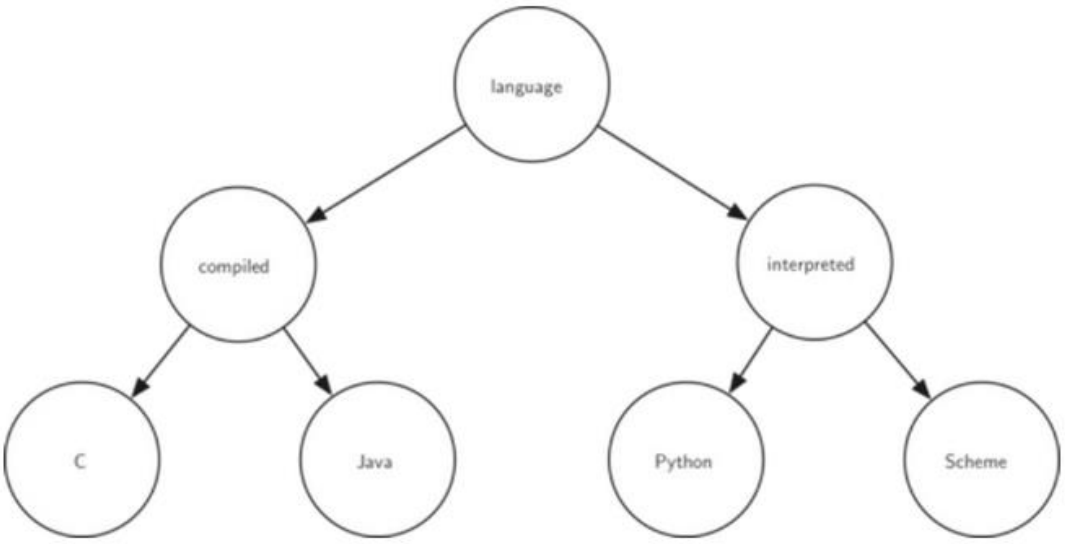
8. 生成一个随机整数的列表。通过插入该列表中的整数生成二叉搜索树，画出生成的二叉搜索树的结果。

9. 考虑下列整数列表：[1,2,3,4,5,6,7,8,9,10]，通过一次插入一个整数的方法生成二叉堆，画出生成的二叉堆的结果。

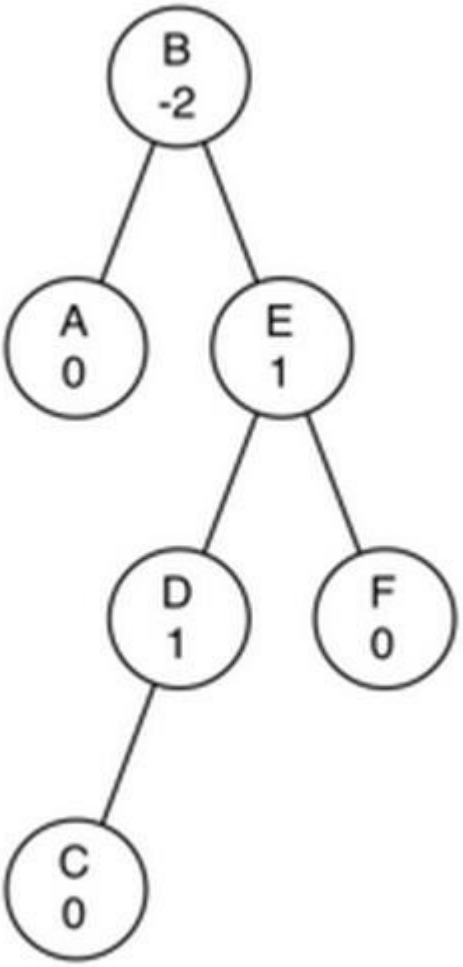
10. 考虑下列整数列表： [10,9,8,7,6,5,4,3,2,1]， 通过一次插入一个整数的方法生成二叉堆，画出生成的二叉堆的结果。

11. 考虑我们使用的两种不同的用于实现二叉树的遍历的方式。 当我们将它实现为一个方法（method） ，为什么我们要在调用前序遍历 preorder 之前检查，然而当我们用函数（function）来实现的时候，可以在函数内部检测？

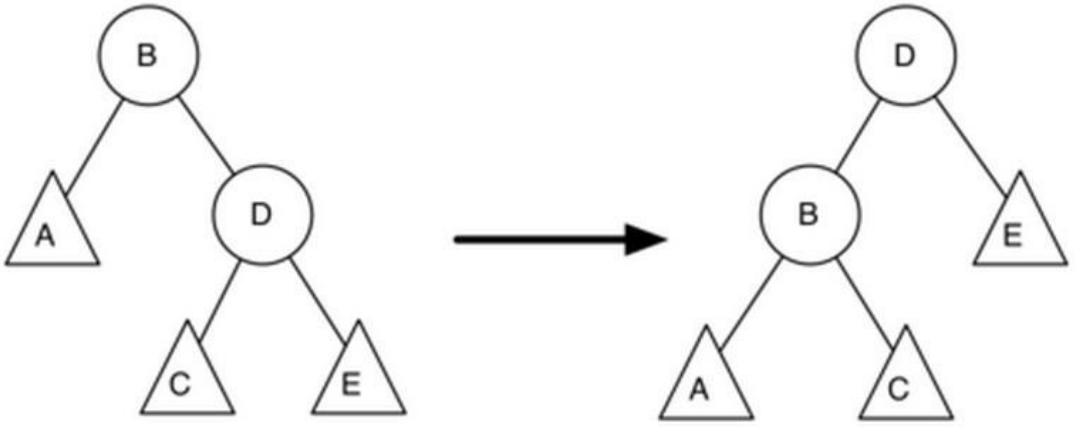
12. 请说明在构建下列二叉树的过程中必要的函数调用：



13. 给定下面的树，请说明经过怎样合适的旋转后可以让它变得平衡。



14. 将下面的情形作为起始点，推导给出更新的 D 节点平衡因子的方程。



6.15 小试牛刀

1. 扩展 buildParseTree 函数，使它能够解决没有空格分隔的数学表达式。

2. 修改 buildParseTree 函数和 evaluate 函数，用它们解决布尔表达式（与 and，或 or 和非 not）。记住非是一个一元操作符，这会让你的编程稍稍复杂一点。

3. 使用 findSuccessor 方法，写一个二叉搜索树的非递归的前序遍历。

4. 修改二叉搜索树的代码使其成为链式的。写一个链式的二叉搜索树的非递归的前序遍历方法。一个链式的二叉搜索树中，每个节点只有对其后继的引用（即没有 parent 这个参量—译者注） 。

5. 修改二叉搜索树代码，使其能够正确解决重复键值问题。也就是说，如果一个键值

（key）已经存在于该树当中，新的负载（value）需要取代旧的而不是添加另外一个相同的键值。

6. 实现一个大小有限的二叉堆。也就是说，这个堆仅仅保存 n 个优先级最高的数据项。如果堆的大小即将超过 n，最不重要的节点将被丢弃。

7. 整理 printexp 函数，使其不包括多余的括号。

8. 使用 buildHeap 方法，写一个排序函数，它能在 **O(nlogn)**时间内将一个列表排序。

9. 写一个函数，它可以为数学表达式构建解析树，并能够计算某些特殊形式的数学表达式的导函数。

10. 实现二叉堆，作为一个最大堆。

11. 使用 BinaryHeap 类，实现一个新的类 PriorityQueue。你的 PriorityQueue 类应该包括构造函数，加上入队 enqueue、出队 dequeue 方法。

7. 图和图算法

7.1. 目标

- 理解图的概念及使用
- 通过多种方法实现图抽象数据类型
- 了解图如何用于解决不同领域的问题

在这一章我们学习图。图是比我们上一章学习的树更普遍的结构，事实上你可以认为树是一种特殊的图。图可以被用来表述世界上很多有趣的事情，包括道路系统，城市间航线，因特网的联接，甚至是你完成计算机科学学位所必修的课程顺序。我们在这一章将看到一旦我们很好地表述了某个问题，我们可以使用标准的图算法来解决一些看起来非常困难的问题。

对人类来说，看懂路线图并了解不同地点之间的关系是相当容易的事情，而计算机并没有这样的能力。但是，我们也可以将路线图看作一个图。当我们这样做的时候，可以让计算机为我们做一些有趣的事情。如果你用过地图网站，你就会知道计算机可以找到一个地方到另一个地方最短、最快捷或最简单的路径。

作为学习计算机科学的学生，你可能想知道修完学位所需课程的学习顺。图就是一个表达课程的先修依赖关系的好方法。图 7.1 展现了在路德学院完成计算机学位必须完成的课程及其顺序。

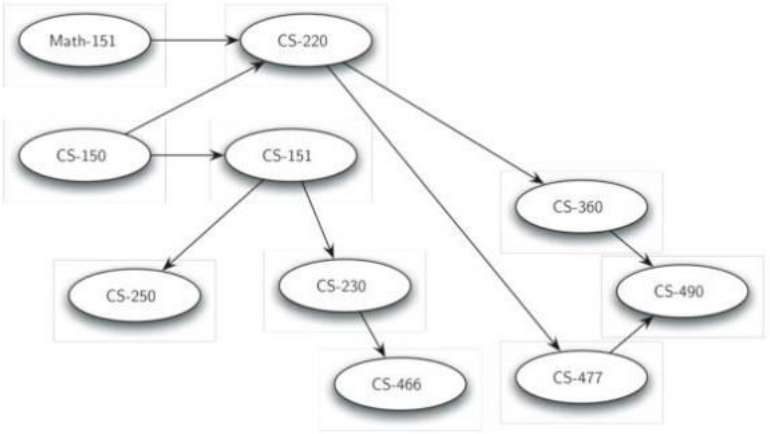


图 7.1 计算机科学学位先修依赖关系

7.2. 词汇表及定义

看完了图的一些例子，我们可以更正式地定义一个图和它的构成要素。通过我们对树的讨论，我们已经知道了一些术语：

顶点 Vertex

顶点（也称“节点 node”）是图的基础部分。它具有名称标识“key”。顶点也可以有附加的信息项“playload”。

边 Edge

边（也称“弧 arc”）是图的另一个基础组成部分。如果一条边连接两个顶点，则表示两者具有联系。边可以是单向的，也可以是双向的。如果一个图中的边都是单向的，我们就说这个图是“有向图 directed graph/digraph”。上图所显示的先修依赖图很明显是一个有向图，因为你必须先修某些课程才能学习其他课程。

权重 Weight

为了表达从一个顶点到另一个顶点的“代价”，可以给边赋权。例如，一个连接两个城市的道路图中，两个城市之间的距离就可以作为边的权重。

有了这些定义，我们可以正式定义一个图。图可以用 G=(V,E)来表述。对于图 G，V 是顶点的集合，E 是边的集合。每个边是一个元组(v, w)，w, v∈V。我们可以在边元组中加入第三个要素来表述权重。它的一个子图 s 就是边 e 和顶点 v 的集合，e ⊂E 并且 v ⊂V。

图 7.2 展现了另一个简单有向赋权图。这个图可以表示成 6 个顶点：

V={V0, V1, V2, V3, V4, V5}

及 9 条边的集合：

E={(v0,v1,5),(v1,v2,4),(v2,v3,9),(v3,v4,7),(v4,v0,1),(v0,v5,2),(v5,v4,8),(v3,v5,3),(v5,v2,1)}

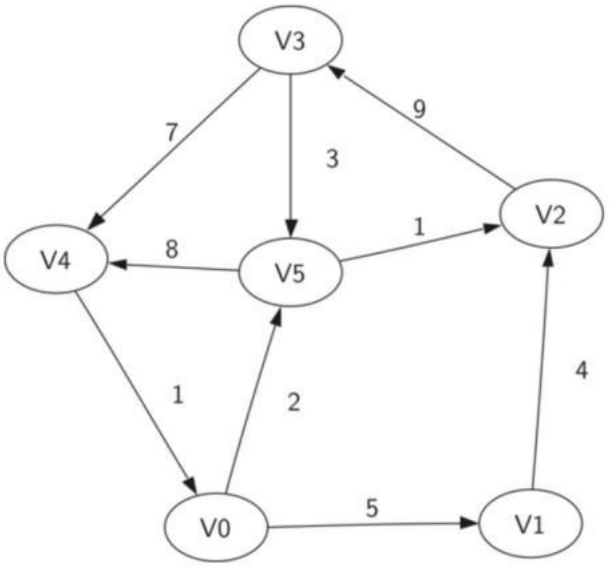


图 7.2 有向图的简单例子

图 7.2 中的图展现了其他两个重要的图的术语：

路径 Path

图中的路径，是由边依次连接起来的顶点序列。我们将路径定义为 P=(w1, w2,..., wn)，其中对于所有 1<=i<=n-1, (wi, wi+1) ∈ E。无权路径的长度为边的数量，等于 n-1。带权路径的长度为所有边权重之和。如图 2 中从 V3 到 V1 的一条路径是顶点序列(v3,v4,v0,v1)，其边为{(v3,v4,7),(v4,v0,1),(v0,v1,5)}。

圈 Cycle

有向图里的圈是首尾顶点相同的路径。例如，图 2 里路径(V5,V2,V3,V5)就是一个圈。没有圈的图称为“无圈图 acyclic graph”，没有圈的有向图称为“有向无圈图 directed acyclic graph 或 DAG”。接下来我们会发现，当问题可以被表述成 DAG 时，我们可以解决一些很重要的问题。

7.3. 图抽象数据类型

图抽象数据类型（ADT）有如下定义：

Graph()：创建一个空的图

addVertex(vert)：将一个顶点 Vertex 对象加入图中

addEdge(fromVert, toVert)：添加一条有向边

addEdge(fromVert, toVert, weight)：添加一条带权的有向边

getVertex(vertKey)：查找图中名称为 vertKey 的顶点

getVertices()：返回图中所有顶点列表

in：按照 vert in graph 的语句形式，返回顶点是否存在图中。如果存在则返回 True，否则返回 False

有几种方法可以在 Python 实现图抽象数据结构（ADT），需要在不同的应用中加以权衡。图的实现有两个著名的方法，邻接矩阵 adjacency matrix 和邻接表 adjacency list。我们将说明这两种不同的选择，并作为 Python 的类来实现邻接矩阵。

7.4. 邻接矩阵

图最容易的实现方法之一就是采用二维矩阵。在矩阵实现方法中，每行和每列都代表图中的顶点。如果顶点 v 到顶点 w 之间有边相连，则将值储存在矩阵的 v 行，w 列。当两个顶点通过边来连接，我们就说它们就是邻接的。图 7.3 展现了图 7.2 中图的邻接矩阵，每一格的值代表了从顶点 v 到顶点 w 边的权重。

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

图 7.3 图的邻接矩阵表示

邻接矩阵的优点是简单，对于简单的图来说很容易看出节点之间的联系状态。然而，我们也注意到大部分的矩阵分量是空的，这种情况我们称矩阵是“稀疏”的。矩阵并不是一个储存稀疏数据的有效途径。事实上，在 Python 里你必须不厌其烦地制造图 3 这样的矩阵结构。

当边的数量庞大时，邻接矩阵是图的一个良好的实现方法。但是我们所说的庞大到底是多大呢？需要多少边来填充矩阵？因为图中每个顶点对应着一行一列，填满矩阵需要的边的数量是$|V|^2$。一个充满的矩阵是每个顶点都与其他任何顶点相连。实际的问题很少可以到达这样的连接量。本章我们要考虑的问题是稀疏图。

7.5. 邻接表

一个实现稀疏图的更高效的方案是使用邻接表 adjacency list。在这个实现方法中，我们维护一个包含所有顶点的主列表（master list），主列表中的每个顶点，再关联一个与自身有边连接的所有顶点的列表。在实现顶点类的方法里，我们使用字典而不是列表，此时字典中的键（key）对应顶点标识，而值（value）则可以保存顶点连接边的权重。图 7. 4 展现了图 7.2 中图的邻接表。

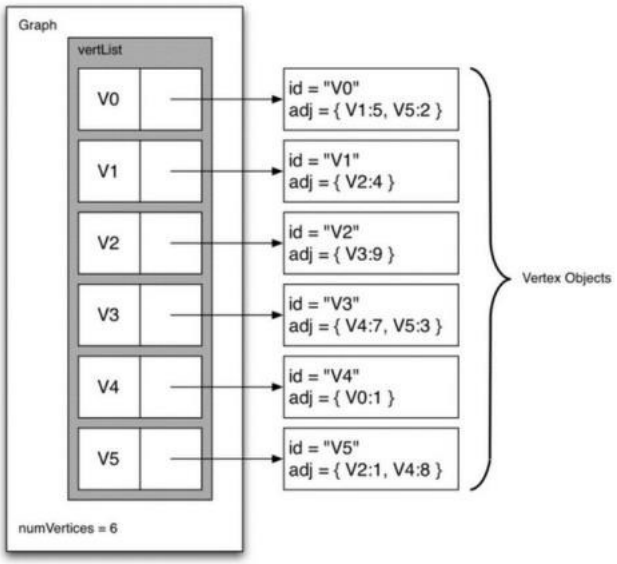


图 7.4 图的邻接列表表示

邻接表实现方法的优点是允许我们高效地表示一个稀疏图。邻接链表还使我们很容易找到某个顶点与其他顶点的所有连接。

7.6.实现

在 Python 中使用字典将使得邻接表的实现变得很容易。在我们实现图表抽象数据类型时，我们可以创建两个类，Graph 和 Vertex（详见表一和表二）。Graph 保存了包含所有顶点的主表，Vertex 则描绘了图表中顶点的信息。每一个 Vertex 使用一个字典来记录顶点与顶点间的连接关系和每条连接边的权重，这个字典被称作 connectionTo（self. connectionTo）。下面的列表给出了 Vertex 类的代码。构造函数（__init___）简单地初始化了（一般为字符串的）id 和 connectionTo 字典。addNeighbor 方法被用来添加从一个顶点到另一个顶点的连接。getConnections 方法用以返回以 connectionTo 字典中的实例变量所表示的邻接表中的所有顶点。getWeight 方法可以通过一个参数返回顶点与顶点之间的边的权重。

Listing1：

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}
    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight
    def __str__(self):
    return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connec
tedTo])
    def getConnections(self):
        return self.connectedTo.keys()
    def getId(self):
        return self.id
    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

代码 7.1

下表实现的 Graph 类，包含了一个将顶点名称映射到顶点对象的字典。在图 7.4 中这个字典对象被阴影灰色框所代表，Graph 也提供了向图中添加顶点和将一个顶点与另一个连接起来的方法。getVertices 方法可以返回图中所有顶点的名称。另外我们可以通过实现__iter__方法简化对特定图中所有顶点对象的遍历。这两种方法允许你通过顶点名称或顶点对象本身去遍历图中的顶点。

Listing2:

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0
    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex
    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None
    def __contains__(self,n):
        return n in self.vertList
    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
            self.vertList[f].addNeighbor(self.vertList[t], cost)
    def getVertices(self):
        return self.vertList.keys()
    def __iter__(self):
        return iter(self.vertList.values())
```

代码 7.2

运用刚刚定义的 Graph 和 Vertex 类，我们通过下面的 Phython 代码可以实现图 7.2 所表示的图。首先，我们创造六个顶点并将其从 0 编码到 5，然后我们显示 Vertex 字典。注意到从 0 到 5 的每一个 key 我们都已经创建了一个 Vertex 实例。接下来，我们添加边来将顶点之间连接起来。最后，用一个嵌套循环来核实图表的每条边都被正确地储存了起来。你在这个过程的最后还应该检查“边”列表的输出是否与图 7.2 一致。

Listing3:

```
>>> g = Graph()
>>> for i in range(6):
...     g.addVertex(i)
>>> g.vertList
```

```
{0: <adjGraph.Vertex instance at 0x41e18>,
1: <adjGraph.Vertex instance at 0x7f2b0>,
2: <adjGraph.Vertex instance at 0x7f288>,
3: <adjGraph.Vertex instance at 0x7f350>,
4: <adjGraph.Vertex instance at 0x7f328>,
5: <adjGraph.Vertex instance at 0x7f300>}
>>> g.addEdge(0,1,5)
>>> g.addEdge(0,5,2)
>>> g.addEdge(1,2,4)
```

FOOL

POOL

POLL

POLE

PALE

SALE

SAGE

```
>>> g.addEdge(2,3,9)
>>> g.addEdge(3,4,7)
>>> g.addEdge(3,5,3)
>>> g.addEdge(4,0,1)
>>> g.addEdge(5,4,8)
>>> g.addEdge(5,2,1)
>>> for v in g:
...     for w in v.getConnections():
...         print("( %s , %s )" % (v.getId(), w.getId()))
... ( 0 , 5 ) ( 0 , 1 ) ( 1 , 2 ) ( 2 , 3 ) ( 3 , 4 ) ( 3 , 5 ) ( 4 , 0 ) ( 5 ,
 4 ) ( 5 , 2 )
```

代码 7.3

7.7. WORD LADDER 词梯问题

我们考虑下面这个被称作“词梯”的问题以开始我们对图算法的学习。比如，将单词“ FOOL”转变成单词“ SAGE”。在词梯问题中，你必须以一次只改变一个字母的方式来逐步转变单词。每一步你都必须将一个单词转变成另一个单词，并且不允许转变成一个不存在的单词。词梯问题是 1878 年由《爱丽丝漫游奇境》的作者 Lewis Carroll 发明出来。下面的单词序列展示出了针对上述例子的一种可能解。

词梯问题还有很多的变形。比如你可能被要求以给定的步数完成单词转换，或者你必须使用一个特定的单词。在本节内容中，我们感兴趣的是如何算出从开始单词到目标单词所需要的最小转换次数。

既然这一章讨论的是图，我们自然能用一个图算法来解决这个问题。我们的纲要如下：

- 以图的形式描绘出单词之间的关系；
- 利用被称作广度优先搜索（BFS）的图算法找到一条从开始单词到目标单词的最短路径。

7.7.1. 建立 WORD LADDER 图

我们第一个问题是去解决如何将大量单词组成的集合转变成图。我们想要的是在两个仅差一个字母的单词之间连一条边。如果我们可以创建一个这样的图表，那么任一从一个单词到另一个单词的路径都是某个词梯问题的一个解。图 7.5 显示了一个由一些单词构成的小图，可以解决从“ FOOL”转变成“SAGE”的词梯问题。值得注意的是，该图是无向图，并且边是没有权的。

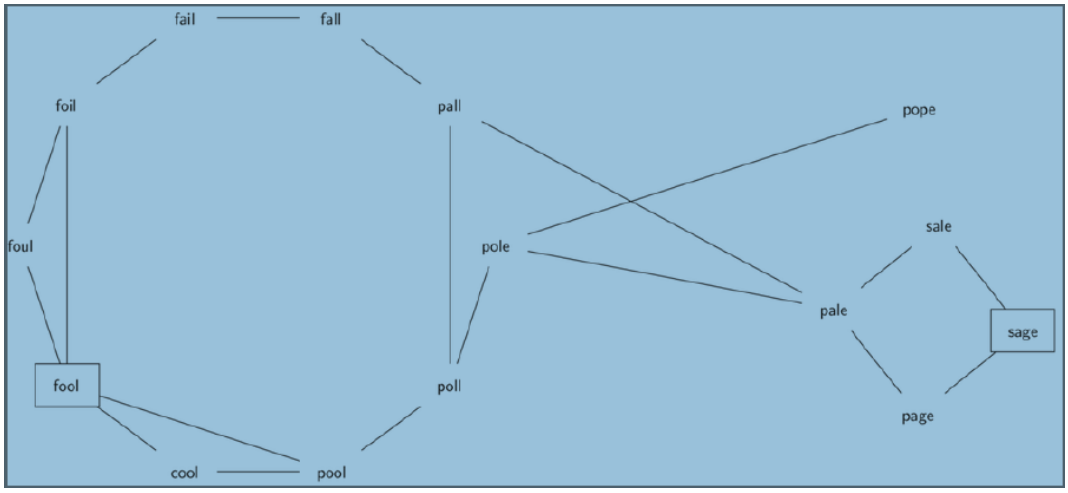


图 7.5 一个小词梯图

我们可以用几种不同的方法来创建我们用来解决词梯问题的图。我们先假设我们有一个单词列表,其中的单词都是一样长。作为我们的第一步，我们可以为列表里的每个单词在图中创建一个顶点。而为将这些单词连接起来，我们可以将列表中的每个词与所有其他单词比较。当我们比较的时候，我们想要看到有多少字母是不同的。如果这两个词只有一个字母不同，我们就可以在图中创建一条连接它们的边。对于少量的单词，这种方法工作得很好；但是假定我们有一个 5110 个单词的列表。大致说来，将列表中的每个词都与所有其他单词比较是一个 $O(n^2)$算法。 对 5110 个单词，n^2比 2600 万还大。

我们可以通过使用下面的方法来优化。假设我们有非常多的桶,每个桶外都贴有一个四个字母的单词标签，并且标签上有且仅有一个字母被‘ _’(通配符)所代替。例如，考虑图 7.6,我们就可能会将一个桶贴上“ pop_”， 当我们处理我们的列表中的每个词，都将其与每个桶比较，使用“_”作为一个通配符,那么“pope” 和“ pops” 都与“ pop_”匹配。每次我们找到一个匹配的桶,我们把单词放在桶里。一旦我们把所有单词都放在适当的桶里，我们便知道，同一个桶里的所有单词都是相互连接的。

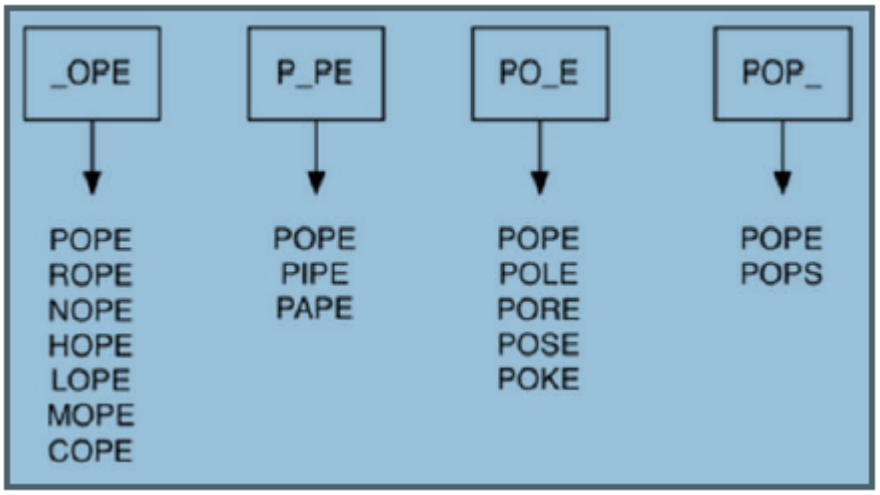


图 7.6 单词桶：只有一个字母不同的单词的集合

在 Python 中，我们可以通过使用一个字典来实现我们刚刚描述的方案。我们刚刚提到的桶上的标签在我们的字典中作为 key，键值 value 是标签对应的单词列表。一旦我们创建了字典我们就可以生成图。我们以在图中为每一个单词创建一个顶点开始，然后在同一个键值列表的任意两个单词对应的顶点之间连一条边。代码 7.4 显示了构建图所需的 Python 代码。

```
from pythonds.graphs import Graph

def buildGraph(wordFile):
    d = {}
    g = Graph()
    wfile = open(wordFile,'r')
    # create buckets of words that differ by one letter
    for line in wfile:
        word = line[:-1]
        for i in range(len(word)):
            bucket = word[:i] + '_' + word[i+1:]
            if bucket in d:
                d[bucket].append(word)
            else:
                d[bucket] = [word]
    # add vertices and edges for words in the same bucket
    for bucket in d.keys():
        for word1 in d[bucket]:
            for word2 in d[bucket]:
                if word1 != word2:
                    g.addEdge(word1,word2)
```

代码 7.4

既然这是我们第一个现实生活中的图的问题,你可能想知道这个图的稀疏程度如何?我们为这个问题准备的 4 个字母的单词列表足足有 5110 个单词那么长。如果我们要使用一个邻接矩阵存储,矩阵会共有 5110*5110=26112100 个格子。 而由 buildGraph 函数构建出来的图只有 53286 条边,所以矩阵将只有 0.20%的格子被填充了!可见,这个图的确是非常稀疏的。

7.7.2. 实现广度优先搜索(BFS)

在建立了词梯问题的图之后,我们就能把注意力转向寻找最短单词变化序列的算法,我们将要用到的这一图算法被称作广度优先搜索(BFS)算法。 BFS 是实现图搜索的最简单的算法之一,同时,它也是我们之后所要学习的一系列其它重要图算法的原型。

已知一个图 G 和它的一个起始顶点 s,广度优先搜索(BFS)通过搜索图中的边来找到图 G 中所有和 s 有路径相连的顶点。其显著的特点是在搜索达到距离 k+1 的顶点之前,BFS 会找到全部距离为 k 的顶点。有一个很好的方法去想象广度优先搜索(BFS)的运行原理,那就是建造一棵以 s 为根的树的过程,一次建造树的一层,同时,广度优先搜索(BFS)在增加层次前,会保证将始顶点所有的子顶点都添加在了树中。

为了进一步追踪这一过程,这里的 BFS 算法在搜索过程中,会给每一个顶点染色为白色、灰色或黑色。每一个顶点在被构建时都被初始化为白色,在这之后,白色代表的是尚未被发现的顶点。当一个顶点被第一次发现后,它被染成灰色,当广度优先搜索(BFS)完全探索完一个顶点后,它被染成黑色。这意味着一旦一个节点染成了黑色,它就没有邻近的白色节点;而另一方面,如果一个顶点被标识为了灰色,这就意味着其附近可能还存在着未探索的顶点等待被探索。

下面的代码 7.5 中的广度优先搜索算法使用的是前文出现过的邻接表来实现的图。此外,它还使用了一个队列来决定下一步应该探索哪一个顶点,我们之后就会知道这一操作有多么重要。

另外,词梯问题的广度优先搜索(BFS) 算法使用的是扩展版的 Vertex 类,这个新的类中包括三个新的实例变量:距离,父顶点和颜色,并且这三种实例变量都有相应的取值以及设置的方法。扩展部分的代码在 pythonds 的代码包中都有,但我们在这里不作说明,因为它们的实现方法我们之前都已经学过了。

广度优先搜索(BFS) 从起始顶点 s 开始,此时 s 的颜色被设置为灰色,代表它现在已经被发现了,另外两个参数——距离和父顶点,对于起始节点 s 初始设置为了 0 和 None。随后,起始节点会被加入到一个队列中,下一步便是系统地探索队首顶点。这个过程通过迭代(遍历)队首顶点的邻接列表来完成,每检查邻接表中的一个顶点,便会维护这个顶点的颜色参量,如果颜色是白色的,就说明这个节点尚未被探索,也就会按下述四步操作:

1、 新的未探索的顶点 nbr,标记为灰色;

2、 nbr 的父顶点被设置为当前节点 currentVert;

3、 nbr 的距离被设置为当前节点的距离加一;

4、 nbr 被加入队尾,这一操作使得直到 nbr 在当前顶点的邻接列表中的所有顶点被搜索完后,才能够进行下一层次的探索操作。

```
from pythonds.graphs import Graph, Vertex
from pythonds.basic import Queue
def bfs(g,start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue = Queue()
    vertQueue.enqueue(start)
    while (vertQueue.size() > 0):
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance() + 1)
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)
```

代码 7.5

作为一个例子，我们现在来看看 bfs 函数是如何构建与图 7.5 所示的图相对应的广度优先树的。首先我们从起始顶点 fool 开始，将所有与其相连的顶点都添加到树中，这些邻接的顶点包括 pool、foil、foul 和 cool，这些顶点作为新的顶点被添加到队尾。图 7.7 所示是这一操作结束后，树和队列的状态。

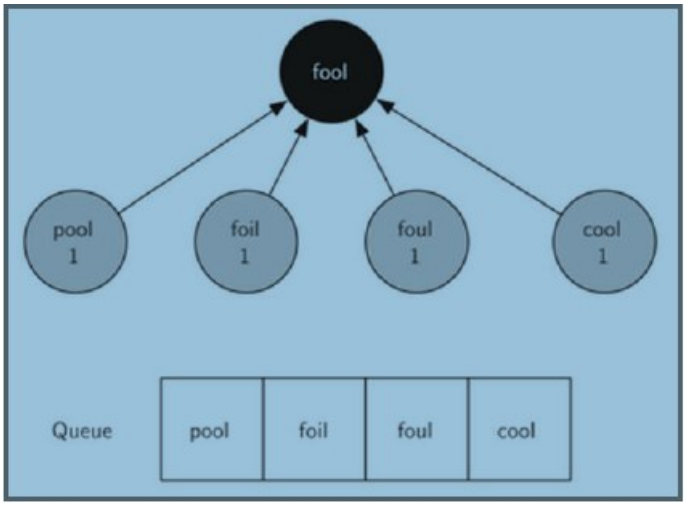


图 7.7 BFS 的第一步

bfs 函数的下一步是将位于队首的节点 pool 从队列中移除，并且对其邻接节点重复第一步中的操作。然而，当 bfs 函数检查节点 cool 时，会发现它的颜色已经染为了灰色，这代表它已经被发现过了，并且表明从起始节点到 cool 之间有一条更短的路径，cool 已经存在于队列中留待下一

步探索。探索顶点 pool 后，唯一新加入队列的顶点是 poll。这一步之后，树和队列的状态如图 7.8 所示。

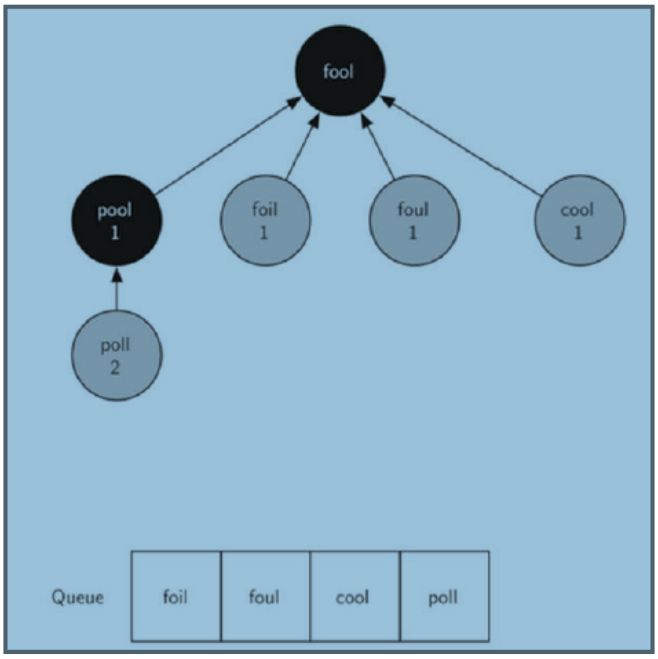


图 7.8　BFS 的第二步

队列中的下一个节点是 foil，探索 foil 后唯一能够加入树中的节点是 fail；队列中接下来的两个顶点 cool 和 poll 在被探索之后，都没有新的顶点加入树和队列。图 7.9 展示的是将第二层级的顶点都探索过后，树和队列的状态。

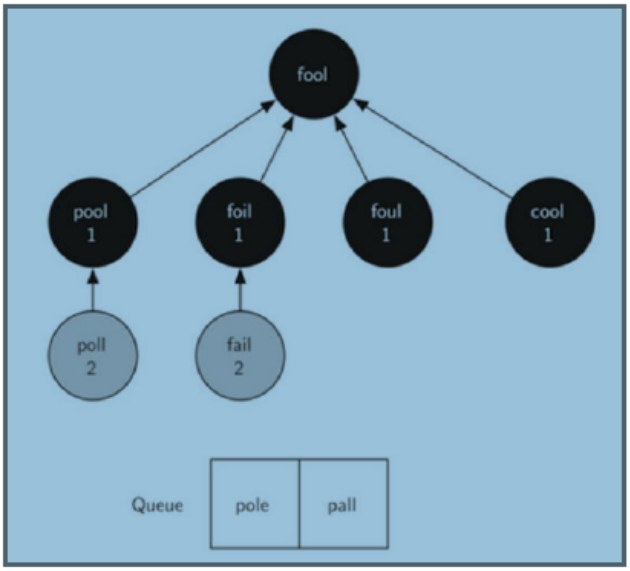


图 7.9 完成探索一层之后的树

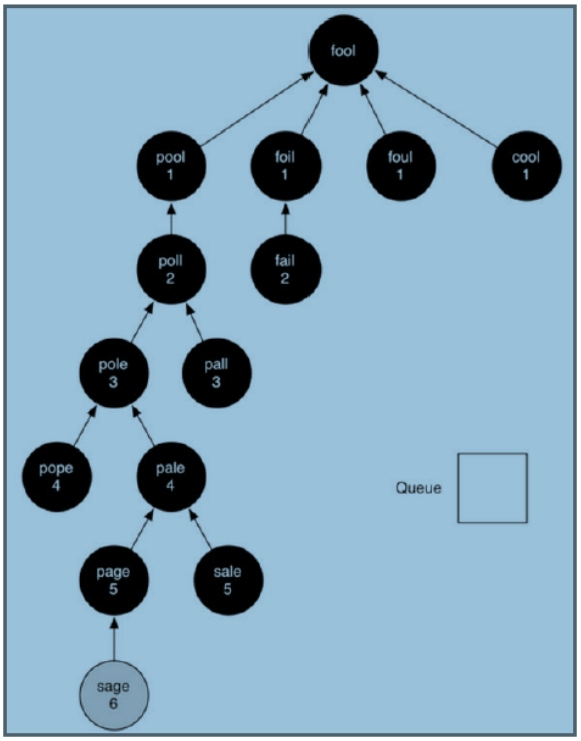


图 7.10 最终的广度优先搜索(BFS)树

读者最好是自己动手去模拟这个算法，以便于对这个过程有更好的理解，图 7.10 是图 7.7 中所有顶点都扩展搜索后所得到的最终的 BFS 树。但是 BFS 算法最神奇的地方在于我们不仅解决了我们最开始提出的那个 fool 如何变到 sage 的问题，更是在这过程中还顺便解决了许多别的问题。我们可以从 BFS 树的任意一个节点出发，沿着指向父节点的箭头返回到根节点，从而得到从这个单词变化到 fool 的最短词梯。 下面的函数代码 7.6 展示了如何去通过父节点链接来打印出整个词梯。

```
def traverse(y):
    x = y
    while (x.getPred()):
        print(x.getId())
        x = x.getPred()
    print(x.getId())
traverse(g.getVertex('sage'))
```

代码 7.6

7.7.3. 广度优先搜索（BFS）的分析

在我们继续深入研究其它图算法之前，先来分析一下 BFS 算法的时间复杂度。首先我们观察到，当 while 循环被执行时，图的顶点集合|V|中的每个顶点最多被访问一次（也可以从每一个顶点在被发现和加入队列之前都是白色的得出相同的结论），因而 while 循环拥有 O（V）的复杂度。另外，对于嵌套在 while 语句中的 for 语句，其对于图的边集|E|中的每一条边至多会被执行一次。这是因为每个顶点只会最多被出队一次，而对于从顶点 u 到顶点 v 的边，只有当顶点 u 出队

的时候我们才会检查，所以每条边最多被检查一次，因此这个循环的复杂度是 O（E）。这两个循环的时间复杂度加起来，也就是 O（V+E）。

当然，执行 BFS 只是整个过程中的一部分，找到连接从起始顶点到目标顶点的路径是另一部分。最坏的情况的从起始顶点到目标顶点就是一条长链，在这种情况下遍历所有的顶点的复杂度就是 O（V）。正常情况下的复杂度是小于|V|的基数的，但我们仍然把 O（V）作为其时间复杂度。

最后，至少对于这个问题而言，我们需要时间来建立最开始的图，现在我们把对 buildGraph 函数的复杂度分析作为课后练习交由读者完成。

7.8. 骑士周游问题

用来阐明我们的第二个图算法的又一经典问题是“骑士周游”。骑士周游问题是在国际象棋棋盘上仅用“骑士”这个棋子进行操作。问题的目的是找到一条可以让骑士访问所有格子，并且每个格子只能走一次的走棋序列。一个这样的走棋序列称为一次“周游”。多年以来，骑士周游问题已经吸引了无数的数学家、棋手、计算机科学家。在 8×8 的国际象棋棋盘上，目前知道的合格的“周游”数量上界有 1.035*10^35 这么多。然而，走棋过程中无路可走的情况就更多了。显然这是一个要么需要真正的智慧要么占用无数计算资源的问题，或者两者都需要。

尽管现在研究人员已经研究了很多不同的算法来解决骑士周游问题，图搜索依旧是最便于理解和编写的算法之一。下面，我们采用两步走的方案来解决这个问题：

- 将棋盘上合法的走棋次序表示为一个图
- 采用图搜索算法搜寻一个长度为（行×列 - 1）的路径，此路径上恰包含每个顶点一次

7.8.1. 建立骑士周游图

为了将骑士周游问题表示为图，我们将用到如下两个想法：将每个棋盘格用图的顶点表示；骑士合法移动的可以用图的一条边表示。 图 7.10 展示了骑士合理的走法以及其在图中所对应的边。

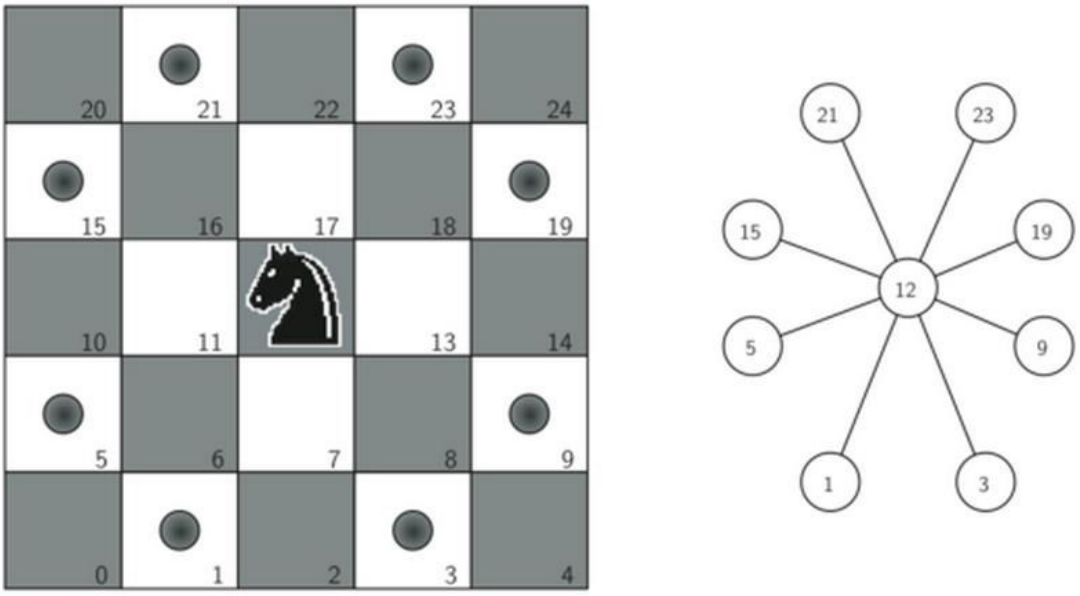


图 7.10 在 12 这个格子上的骑士的能走的格子以及与之对应的图

我们可以利用代码 7.6 中的 Python 函数去建立一个 n*n 棋盘对应的完整图。 knightGraph 函数遍历整张棋盘。在每个格子上 knightGraph 函数调用 genLegalMoves 辅助函数来创建一个当前这个格子上骑士所有合法移动的列表。所有的合法移动都被转换为图上的边。 另一个辅助函数 posToNodeId 根据棋盘上位置的行列信息转换成一个线性顶点数，类似于如图 7.10 所示的线性坐标。

```
from pythonds.graphs import Graph

def knightGraph(bdSize):
    ktGraph = Graph()
    for row in range(bdSize):
        for col in range(bdSize):
            nodeId = posToNodeId(row,col,bdSize)
            newPositions = genLegalMoves(row,col,bdSize)
            for e in newPositions:
                nid = posToNodeId(e[0],e[1],bdSize)
                ktGraph.addEdge(nodeId,nid)
    return ktGraph

def posToNodeId(row, column, board_size):
    return (row * board_size) + column
```

代码 7.6　建立一个 n*n 棋盘对应的完整图

genLegalMoves 函数（代码 7.7）获取骑士当前的位置并生成所有八个走棋步骤。legalCoord 辅助函数确保任何个步骤不会超出棋盘（会导致 IndexError ）。

```
def genLegalMoves(x,y,bdSize):
    newMoves = []
    moveOffsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),( 1,-2),( 1,2),( 2,-1),( 2,1)]
    for i in moveOffsets:
        newX = x + i[0]
        newY = y + i[1]
        if legalCoord(newX,bdSize) and \ legalCoord(newY,bdSize):
            newMoves.append((newX,newY))
    return newMoves

def legalCoord(x,bdSize):
    if x >= 0 and x < bdSize:
        return True
    else:
        return False
```

代码

7.7 `genLegalMoves` 函数

图 7.11 显示了 8*8 棋盘上所有可能的移动方式，图中只有 336 条边。注意到靠近边的格子比中心的格子连接的边更少。进一步地，我们可以看到这个图很稀疏。如果这张图每两个顶点都连有边，则有 4096 条边，因为只有 336 条边，这个邻接矩阵只有 8.2%被填充。

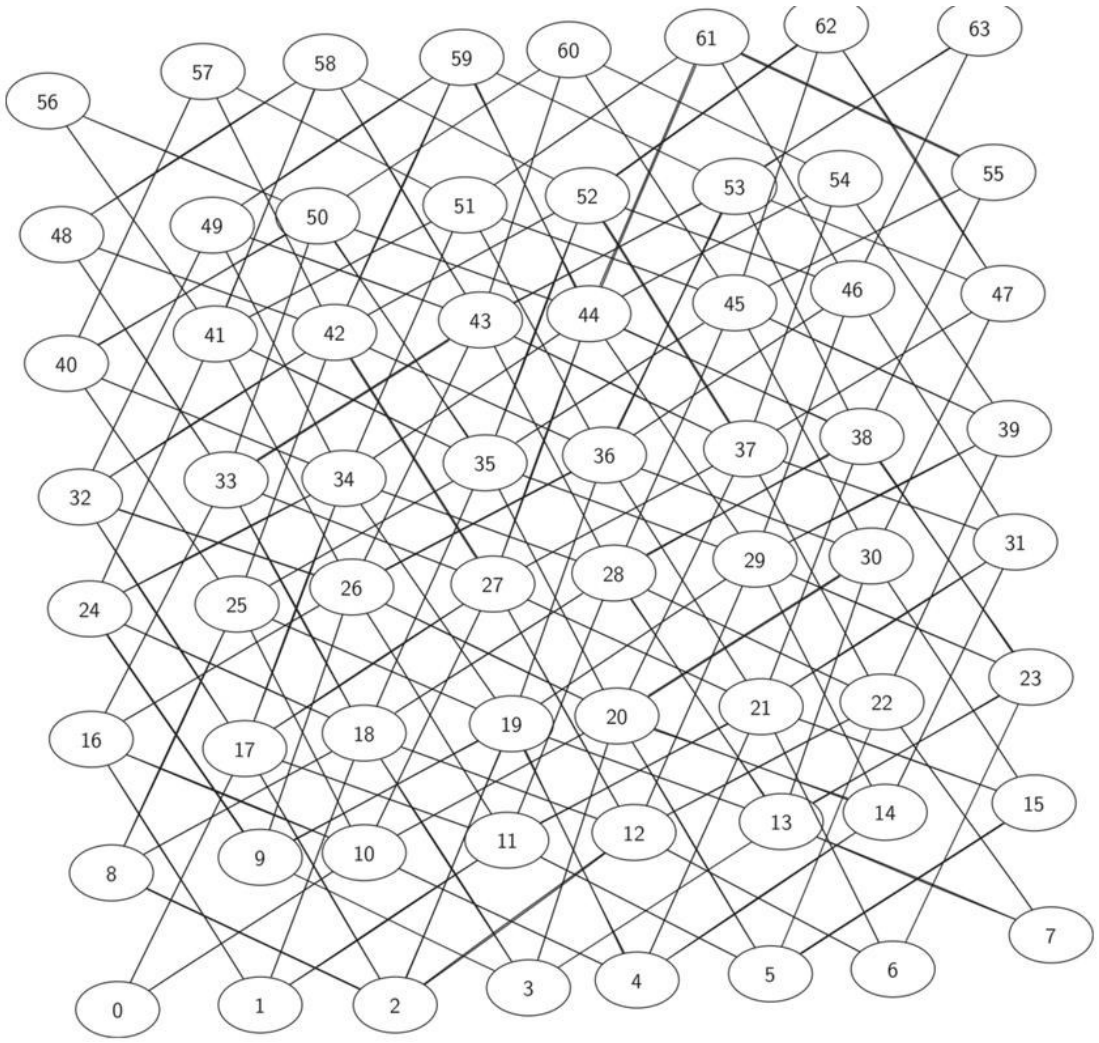


图 7.11　8×8 棋盘上骑士的所有合法走棋步骤

7.8.2. 实现骑士周游

我们用于解决骑士周游问题的搜索算法是 **Depth First Search (DFS)**——深度优先搜索算法。前面讨论的 BFS（广度优先搜索算法）是一次建立搜索树的一层，而 DFS 算法则是尽可能深的搜索树的一枝。这一节我们将介绍两个深度优先搜索(DFS) 的实现算法。第一个 DFS 算法被我们直接用于解决骑士周游问题，其每个顶点最多访问一次。另一个 DFS 算法更为一般，但是在树建立时允许重复访问节点。第二种算法可作为其它图算法的基础。

图的深度优先探索很适合用于发现一条包含 63 条边的路径。我们会看到当 DFS 走到一条死路（没有任何可能的合法移动方式）时，它会沿着树返回直到该节点有路可走。

函数 `knightTour` 需要四个传递参量： `n` ，当前树的深度； `path` ，这个节点前所有已访问的点的列表； `u` ，我们能够探索的点； `limit` ，搜索总深度限制。该函数递归使用：当 `knightTour` 被调用时，首先检查基础状态：如果 path 包含有 64 个节点，函数 `knightTour` 返回 `True` 表示已经找到一条可周游的路径；如果 path 还不够长，我们选择一个新节点并以此为参数调用自身。

DFS 算法还需要使用“颜色”来追踪图中哪些节点已经被访问过了。未访问的节点染为白色，访问过的染为灰色。如果当前节点的全部邻居节点都被访问且没有达到访问全部 64 个节点，我们就到了一条死路，这时我们需要回溯。回溯机制在 `knightTour` 返回 `False` 时启动（即递归的终止条件）。在 BFS 里面我们用 queue（队列）来跟踪要访问的节点，而在 DFS 里由于我们使用了递

归，也即默认使用了 Stack（栈）来实现我们的回溯机制。当我们从 knightTour 函数返回 False 时（ line11），我们依然在 while 循环里面并在 nbrList 中寻找下一个要搜索的节点。

```
from pythonds.graphs import Graph, Vertex
def knightTour(n,path,u,limit):
    u.setColor('gray')
    path.append(u)
    if n < limit:
        nbrList = list(u.getConnections())
        i = 0
        done = False
        while i < len(nbrList) and not done:
            if nbrList[i].getColor() == 'white':
                done = knightTour(n+1, path, nbrList[i], limit)
            i = i + 1
        if not done: # prepare to backtrack
            path.pop()
            u.setColor('white')
    else:
        done = True
    return done
```

代码 7.8

我们来运行一个骑士周游的问题的简单实例。你可以参考下图来模拟搜索的步骤。在这个例子中我们假设 getConnections （ line6）返回的节点列表按字母序。我们从调用 knightTour (0,path,A,6)开始。

knightTour 从图 7.12 的 A 节点开始。与 A 相邻的节点是 B 和 D， B 在字母序上在 D 前，dfs 算法选择 B 来展开下一级（图 7.13）。对 B 的探索通过 knightTour 的递归调用来实现。与 B 相邻的是 C 和 D，所以 knightTour 先选择探索 C 节点。然而，就像你在图 7.14 看到的那样，节点 C 没有相邻的白色节点，是一条死胡同。这时，我们将 C 的颜色改回为白色（同时返回）。这时 knightTour 返回 False 。然后从递归调用回溯到仍有可搜索的子节点的节点，在这里是 B。探索列表中的下一个节点是节点 D，所以 knightTour 函数开始节点 D 进行递归调用。从节点 D 开始， knightTour 可以连续进行递归调用，直到我们再次到节点 C（图 7.17，图 7.18，图 7.19）。然而，这次我们到节点 C 时 n<limit 的判断结果是假，所以我们知道我们已经遍历了这张图里的所有节点。这时我们可以返回 True 来表明我们已经成功地找到了这张图的一个周游方式。当我们返回这个列表，path 记录的是 [A,B,D,E,F,C] ，这正是我们需要的遍历这张图的顺序。（实际上就是找 Hamilton 圈）

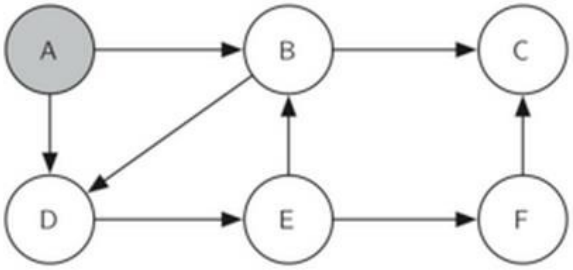


图 7.12 从 A 开始

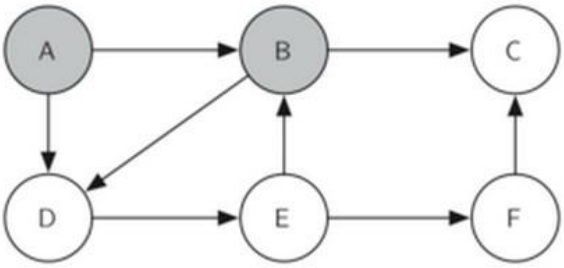


图 7.13 探索 B

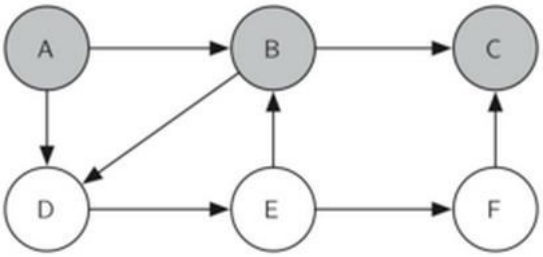


图 7.14 节点 C 是一条死路

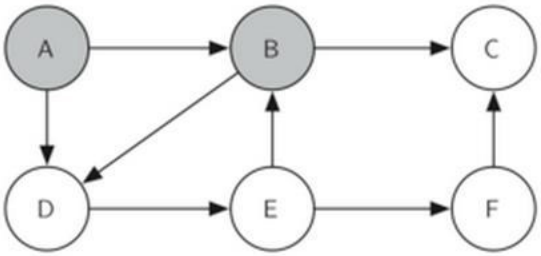


图 7.15 回溯到 B

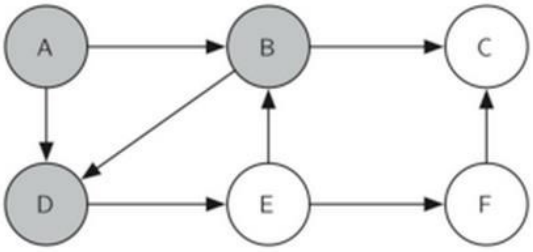


图 7.16 探索 D

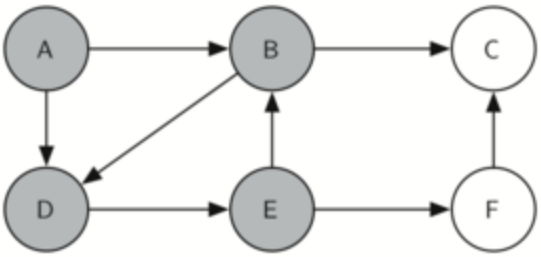


图 7.17 探索 E

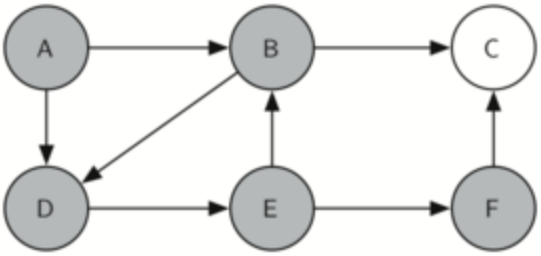


图 7.18 探索 F

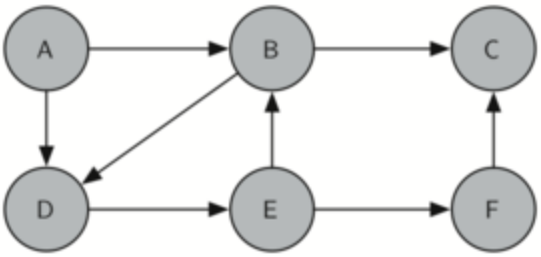


图 7.19 完成

图 7.20 展示了一个 8*8 棋盘上的一个完整周游图。其实还有很多种成功的可能，有些是对称的。而通过一些对函数的修饰，你也能得到一个开始和中止于同一格子的周游路线。

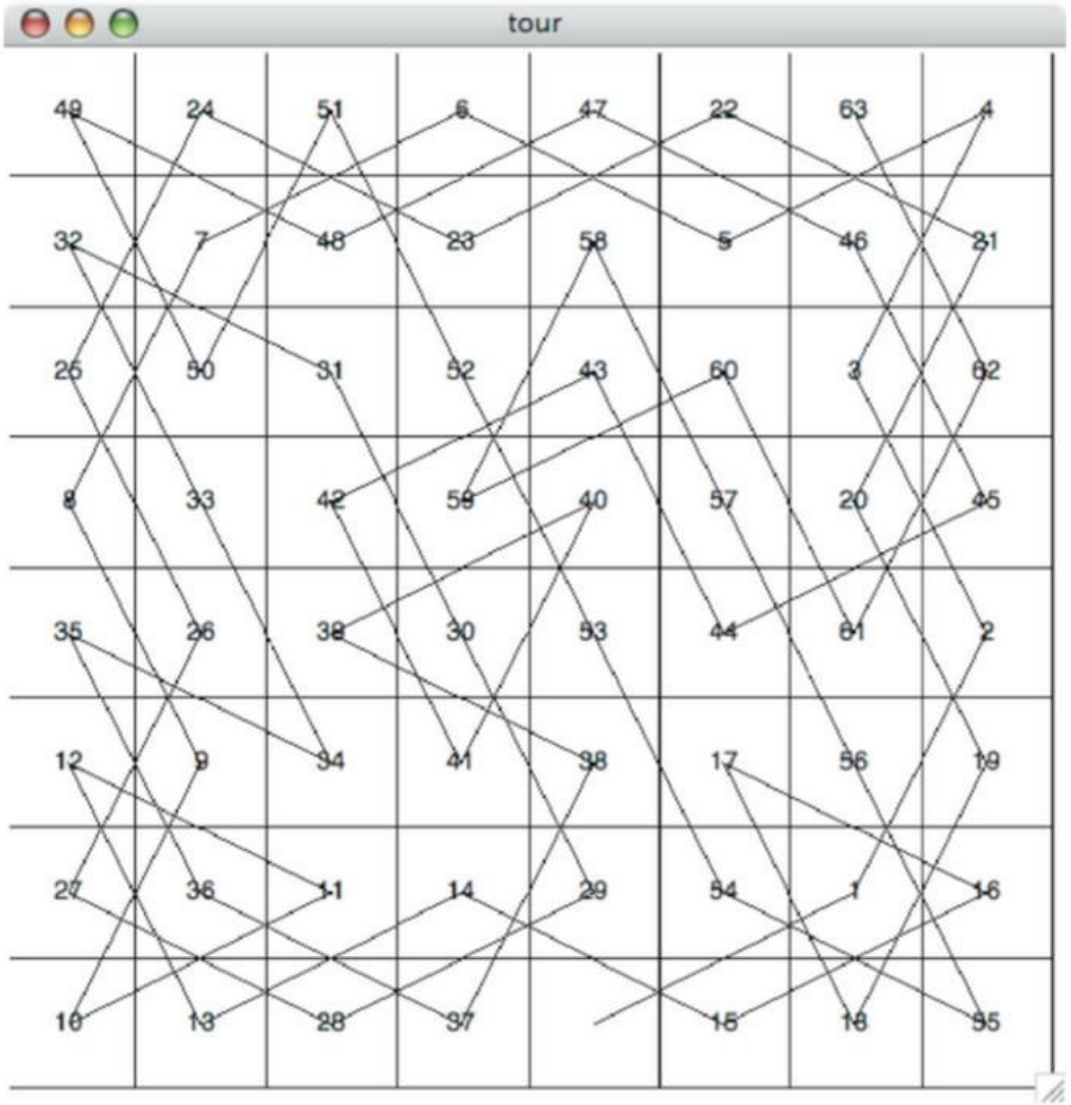


图 7.20 一个棋盘上完整的周游

7.8.3. 骑士周游分析

在我们实现通用的深度优先搜索前，还有一个关于骑士周游问题的有趣话题需要考虑，这就是算法的性能。特别地，骑士周游问题高度依赖于你选择顶点搜索先后次序的方法。例如，在 5×5 的棋盘上，一个相当快的电脑可以在 1.5 秒内找到一个周游路径。但如果你在 8×8 的棋盘上找会怎么样呢？在这种情况下，根据你电脑运行速度的不同，你可能需要等半小时才能得到结果。这种情况出现的原因就是我们当前实现的骑士周游算法是一个时间复杂度为指数 $O(k^N)$的算法，其中 N 是棋盘格的数目,k 是一个小的常数。图 7.21 可以帮助我们形象化地为什么是这样。树的根节点代表搜索的起始点，从这一点开始，算法生成并检查了骑士每一个可能的移动位置。就像我们之前注意到的一样，骑士可移动位置的多少取决于骑士在棋盘中的位置。在角上骑士只有两个合法的移动位置，在与角相邻的方格中有三个合法的移动位置，在棋盘的中间则有八个。图 7.22 展示了棋盘上每个方格中可能移动位置的个数。在树的第二层对于我们正在探索的位置又有 2 到 8 个可能的移动位置。可能的需要检查的位置数与搜索树的节点数一样多。

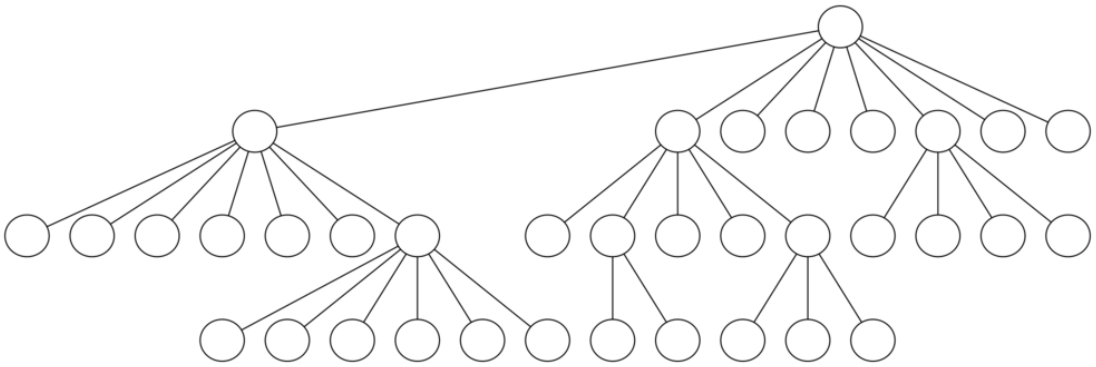


图 7.21 一个骑士周游问题的搜索树

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

图 7.22 棋盘上每个方格的可能移动数目

我们已经知道 N 层高的二叉树的节点总数是 $2^{(N+1)}$ -1，对于一个最多有八个子节点而不是两个的树来说这个数目会更大。因为每一个节点的分枝因子是可变的，我们可以通过计点的平均分枝因子来估计节点总数。需要注意的是这个算法是指数增长的： $k^{(N+1)}$-1， k 是棋盘上的平均分枝因子。让我们看看它增长得多块！对于一个 5×5 的棋盘，这个树将会有 25 层深，或者将第一层深度设为 0 使 N=24。平均分枝因子 k=3.8，所以这棵搜索树的节点总数是 3.8^{25}-1，或者说 3.12×10^{14}。对于 6×6 的棋盘， k=4.4，总节点数为 1.5×10^{23}，而对于一个常规的 8×8 棋盘， k=5.25，总节点数为 1.3×10^{46}。当然，由于这个问题有很多种解，我们不用探索每一个节点，但是即使我们只探索需要探索的那一部分节点，对于这个问题仅仅是常数乘子的缩减，并没有改变这个问题指数增长的现状。我们将把这个留作练习，看看你是否将 k 表达成棋盘大小的函数。

幸运的是，有一种方法可以将 8×8 的棋盘的骑士周游问题提速到不到一秒内跑完。在下面的一段代码中我们展示了可以提速骑士周游问题的算法。这个叫 `orderByAvail` 的函数（见代码 4）将会被用在我们之前展示的代码中的 `u.getConnections` 的调用中。 `orderByAvail` 函数中至关重要的语句在第十行。这一行保证了我们下一步选择有最少可能移动位置的顶点。你可能认为这是起反作用的：为什么不选择那些有最多可能移动位置的节点呢？你可以很轻松的在排序后插入 `resList.reverse()` 这一行并且运行你的程序来尝试一下这种方法。

选择有最多可能移动位置的节点作为下一个顶点的问题在于骑士将倾向于在周游的早期访问棋盘中间的方格。当这样的事情发生的时候，骑士将容易被困在棋盘的一边而不能到达棋盘另一边未

被访问的方格。另一方面，首先去访问最少可能的格子会迫使骑士早早的进入边角的格子。 进而保证骑士早早访问那些不容易到达的角落，并且在需要的时候通过中间的方格跳跃着穿过棋盘。利用这种先验的知识来改进算法性能的做法，称作为“启发式规则”。人类每天应用启发式规则来做决定，启发式搜索经常被用在人工智能领域。这个启发式规则算法以 H. C. Warnsdorff 的名字来命名，被叫做 Warnsdorff 算法，他在 1823 年发表了自己的想法。

```
def orderByAvail(n):
    resList = []
    for v in n.getConnections():
        if v.getColor() == 'white':
            c = 0
            for w in v.getConnections():
                if w.getColor() == 'white':
                    c = c + 1
            resList.append((c,v))
    resList.sort(key=lambda x: x[0])
    return [y[1] for y in resList]
```

代码 7.9

7.8.4. 通用深度优先搜索

骑士周游问题是深度优先搜索中的一个特殊案例，它是以创建深度最深并无分支的优先搜索树为目标。事实上，更一般的深度优先搜索更容易实现。它的目标是尽可能深地搜索，连接图中尽可能多的顶点以及仅在必要时建立分支。

在深度优先搜索中建立不止一个树也是有可能的。当深度优先搜索算法建立一组树时，我们称之为**深度优先森林**。与广度优先搜索相同，深度优先搜索也使用前驱链接来创建树。此外， 深度优先搜索会使用两个附加的 `Vertex` 类的实例变量。这两个新的实例变量是“发现时间”和“ 结束时间”。“发现时间”记录某个顶点第一次出现前算法的操作步数。“完成时间”则是某个顶点被标记为黑色之前算法的操作步数。观察算法后我们会发现顶点的“发现时间” 和“完成时间”提供了一些我们在后续算法中可以使用的有趣性质。

我们的深度优先搜索的代码如代码 7.10 所示。由于 `dfs` 函数和它的辅助函数 `dfsvisit` 两个函数使用一个变量以在 `dfsvisit` 的调用过程中保持对时间的记录，我们选择将这段代码作为一个继承于 `Graph` 类的方法类。这个操作通过添加一个时间实例变量和 `dfs` 与 `dfsvisit` 两个方法拓展了图类。观察 11 行你会发现 `dfs` 算法在图中每个白色节点迭代调用 `dfsvisit` 来遍历图中的所有顶点。我们遍历所有顶点而不是简单地从某个选定起始顶点开始搜索是为了保证图中所有顶点都被考虑到并且没有顶点在深度优先森林中被遗漏。 `for aVertex in self` 这条语句或许看起来不同寻常，但是要记住在这例子中 `self` 是 `DFSGraph` 类的一个实例并且遍历一个图类的实例中所有的顶点是很自然的。

```
from pythonds.graphs import Graph
class DFSGraph(Graph):
    def __init__(self):
        super().__init__()
        self.time = 0
    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)
    def dfsvisit(self,startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
```

代码 7.10

虽然我们对 bfs 的实现只对那些拥有一条可以指回起始顶点的路径的节点感兴趣，但是创建一个广度优先森林来表示图中所有顶点对之间的最短路径是有可能的。我们将此留作练习。在我们接下来的两个算法中我们会看见为何保持对深度优先森林的追踪是重要的。

dfsvisit 方法以一个叫做 startVertex 的单一顶点开始并尽可能深地探索所有相邻白色顶点。如果你仔细观察 dfsvisit 的代码并将其与广度优先搜索进行比较，你应当注意到 dfsvisit 算法除了在内部循环的最后一行外与 bfs 几乎相同。 dfsvisit 递归调用自身以继续对更深层次的探索，而 bfs 通过将顶点添加到一个队列中以便后续探索。需要注意的是，在 bfs 使用队列的地方， dfsvisit 使用的是栈。你虽然在代码中看不见栈的形式，但是它暗含在 dfsvisit 的递归调用中。

以下插图说明了深度优先搜索算法对一个较小的图的操作。这些图中，虚线表示边已经检查，但在边的另一端的顶点已经被添加到深度优先树中。在代码中这个测试是通过检查另一端的顶点的颜色是否是非白色的来实现的。

搜索从图中的 A 顶点开始（图 7.23）。由于所有的顶点在开始时都是白色的所以算法首先访问 A 顶点。访问该节点的第一步操作是将其颜色设置为灰色以表明这个顶点已被探索过并且将“ 发现时间” 设置为 1。由于 A 顶点拥有两个相邻顶点（B 和 D）并且这两个顶点都需要被访问，所以我们任意地决定，比如我们按照字母表顺序依次访问相邻顶点。

接下来访问 B 顶点（图 7.24），然后它的颜色被设置为灰色并且它的“ 发现时间” 被设置为 2。由于 B 顶点同样有两个相邻顶点（C 和 D），所以我们按照字母表顺序接着访问 C 顶点。

在访问 C 顶点（图 7.25）的过程中我们到达了树的一枝的末端。在将 C 顶点涂为灰色并将“ 发现时间” 设置为 3 后，算法认为 C 顶点没有相邻顶点。这意味着我们完成了对 C 顶点搜索并且我们可以将其涂为黑色并设置“结束时间”为 4。你能在图 7.26 中看见这一阶段的情形。

由于 C 顶点在一枝的末端，所以我们返回 B 顶点并继续探索 B 顶点的相邻顶点。由于 B 顶点的唯一相邻顶点是 D，所以我们现在访问 D 顶点（图 7.27）并从此继续我们的搜索。D 顶点迅速将我们带向 E 顶点。 E 顶点有 B 和 F 两个相邻顶点。通常我们会按照字母表顺序探索这些相邻顶点，但由于 B 顶点已经被标记为灰色，所以算法识别出它不应该访问会导致算法陷入死循环的 B 顶点。所以继续探索列表中的下一个顶点 F。（图 7.28）

F 顶点只有一个相邻顶点 C，但由于 C 顶点已经被标记为黑色，不能继续探索，并且算法也到达了树的另一枝的末端。从此开始，你会看见算法一路运算返回初始顶点、设置“结束时间”并设置顶点颜色为黑色（图 7.29-图 7.34）。

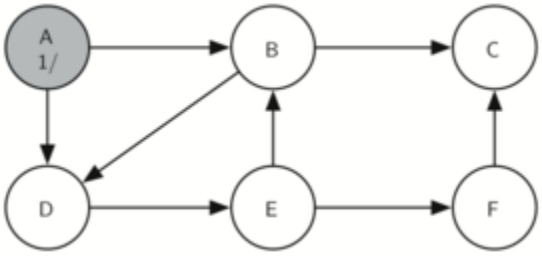


图 7.23 建立深度优先搜索树 10

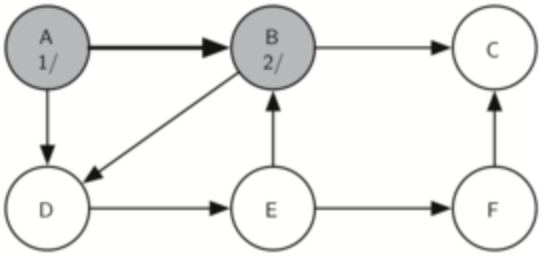


图 7.24 建立深度优先搜索树 11

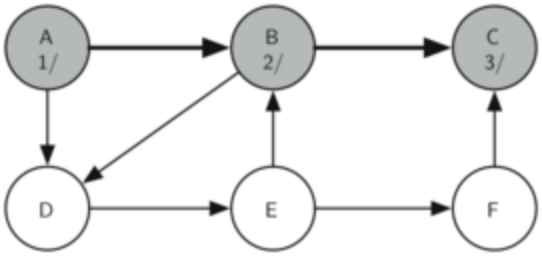


图 7.25 建立深度优先搜索树 12

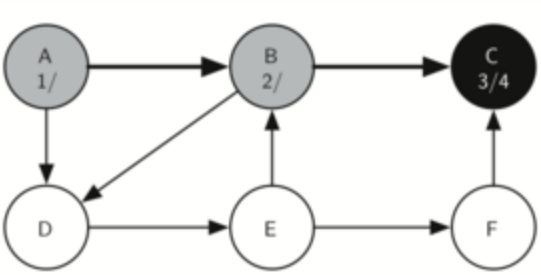


图 7.26 建立深度优先搜索树 13

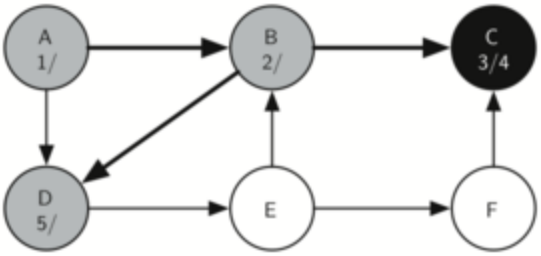


图 7.27 建立深度优先搜索树 14

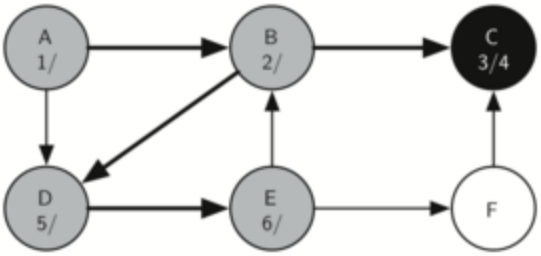


图 7.28 建立深度优先搜索树 15

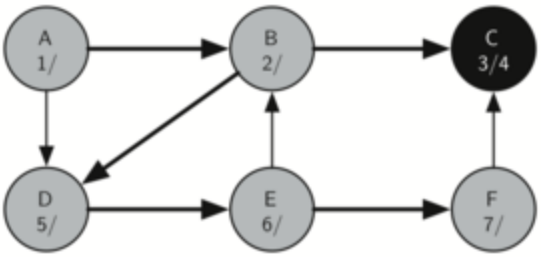


图 7.29 建立深度优先搜索树 16

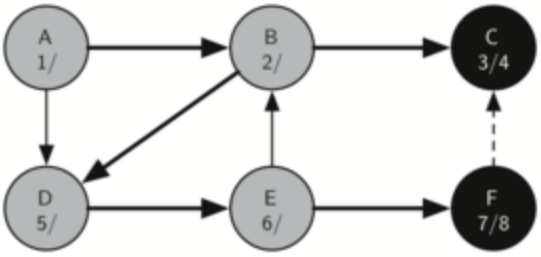


图 7.30 建立深度优先搜索树 17

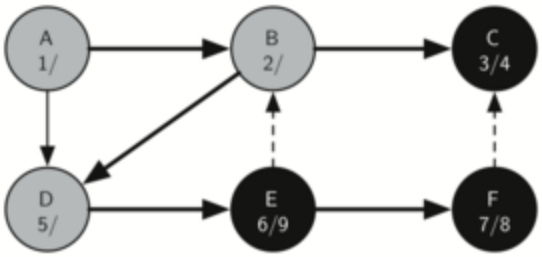


图 7.31 建立深度优先搜索树 18

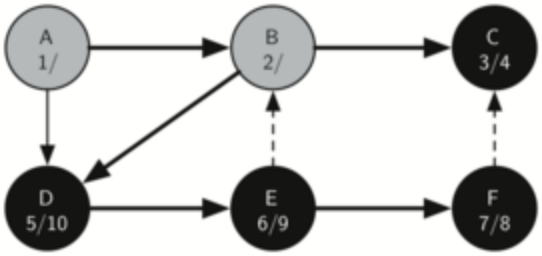


图 7.32 建立深度优先搜索树 19

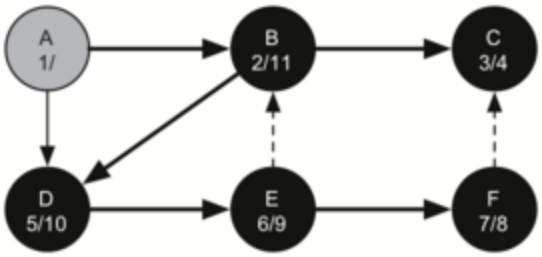


图 7.33 建立深度优先搜索树 20

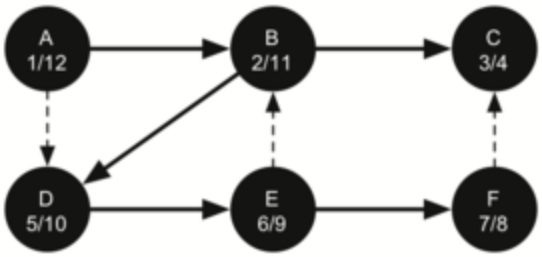


图 7.34 建立深度优先搜索树 21

“开始时间”和“ 结束时间” 展示了每个顶点的被称为**括号性质**的性质。这个性质意味着深度优先树中一个特定顶点的所有的子顶点拥有与它们的父顶点相比更晚的“发现时间”和比更早的“ 结束时间”。图 7.35 展示了深度优先搜索算法建立的树。

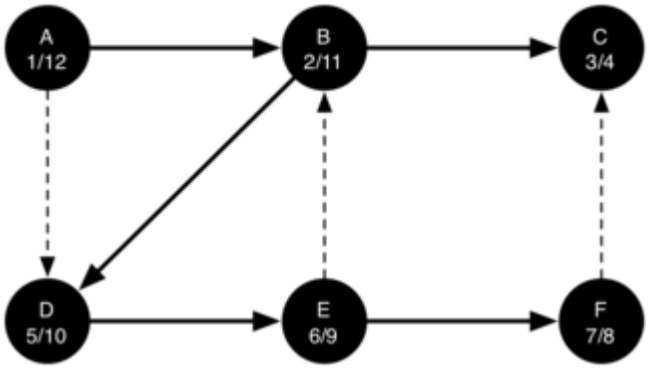


图 7.35 最终深度优先搜索树

7.9 拓扑排序

在计算机学家看来，几乎所有的问题都可以转化为一个图问题，为了验证这个说法，让我们考虑一个复杂的问题——做一个热香饼。食谱十分简单：鸡蛋 1 个，面粉 1 杯，油 1 汤匙，牛奶 3/4 杯。制作热香饼需要先预热平底锅，将所有原料混合均匀后，再将原料舀入加热好的锅中。待锅中的混合物开始冒泡时，将面饼翻面，煎至底部变为金黄色。在吃松饼之前你也可以加热一些果酱作为佐料。图 7.36 以图表的形式说明了这一过程。

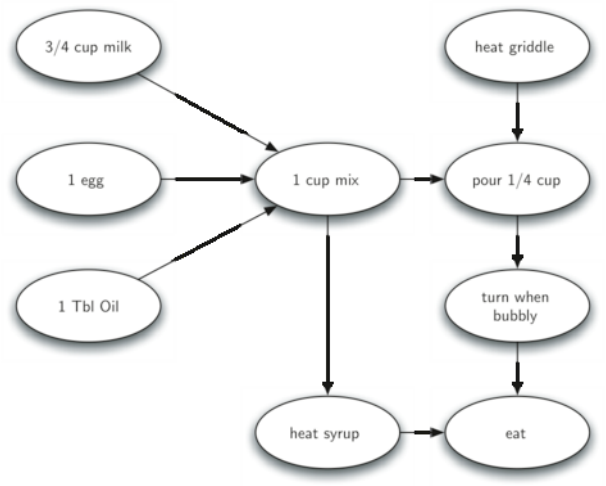


图 7.36： 制作热香饼的步骤

制作热香饼的困难之处在于第一步做什么。如图 7.36 所示，你可以从加热平底锅开始，也可以从向面粉中加入原料开始。为了帮助我们在制作热香饼时的每一步都能有一个清晰的顺序，我们可以求借助于一个叫**拓扑排序**的图算法。

拓扑排序可以将一个有向无圈图(DAG)转换为一个只含它所有顶点的线性排列，例如如果一个图 G 包含一个边界(v,w)，然后我们就可以按顺序将顶点 v 放在顶点 w 之前。DAG 在许多应用中都可以起到指示事件优先级的作用，制作热香饼只是其中一个例子。像这样的例子还有：设计软件工程日程，制作可以优化数据库查询的图表，解决矩阵相乘等问题。

拓扑排序是深度优先搜索(DFS)的一个简单却有效的应用。拓扑排序的算法遵从以下法则：

1.为所求图问题调用 DFS 函数。我们调用 DFS 的主要原因是为了计算每一个事件顶点的完成时间。

2.按完成时间降序将事件顶点存储在一个列表中。

3.将列表作为拓扑排序的结果返回。

图 7.37 展示了我们把 DFS 应用于图 7.36 中热香饼制作问题后所构造的深度优先森林。

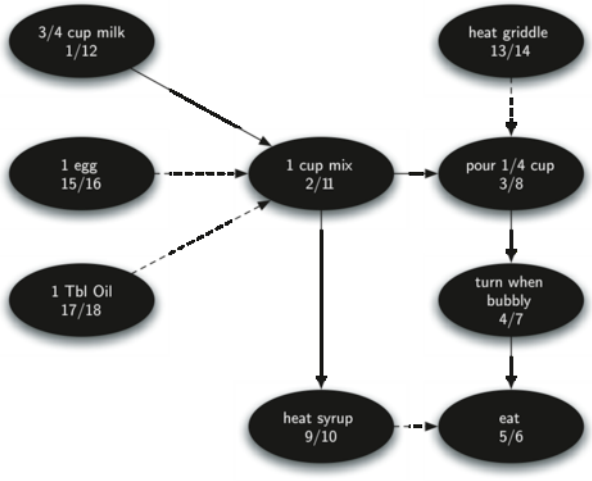


图 7.37：对做热香饼的图进行深度优先搜索的结果

最后，图 7.38 展示了为我们的图应用拓扑排序算法后的结果。现在，所有模棱两可的东西都被移除，我们可以确切地知道制作热香饼时每一步的顺序了。

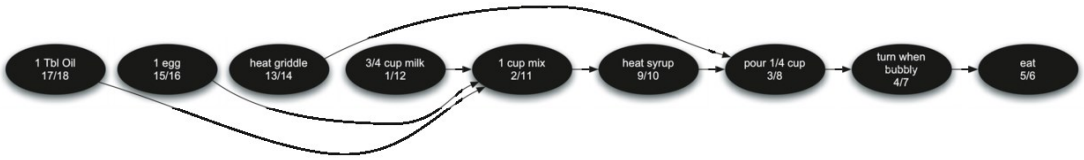


图 7.38：用有向无圈图(DAG)表示拓扑排序的结果

7.10 强连通分支

本章剩余部分我们将关注一些极其巨大的图，用来学习一些其他算法，这些图产生于互联网上主机之间的连接和网页之间的链接。我们将从这些网页开始。

像 Google 和 Bing 这样的搜索引擎都是从网页构成的巨大有向图中搜索想要得到的事实。为了将互联网转换为一张图，我们将每一张网页看作一个顶点，网页上的超链接看作联接顶点的边。图 7.39 显示了图的一个非常小的部分，这部分开始于（美国）路德学院的计算科学主页，通过网页之间的链接产生。当然，这张图可以相当巨大，所以我们不得不限制每张网站到与代码段所在主页的最大链接数不能超过 10 个。

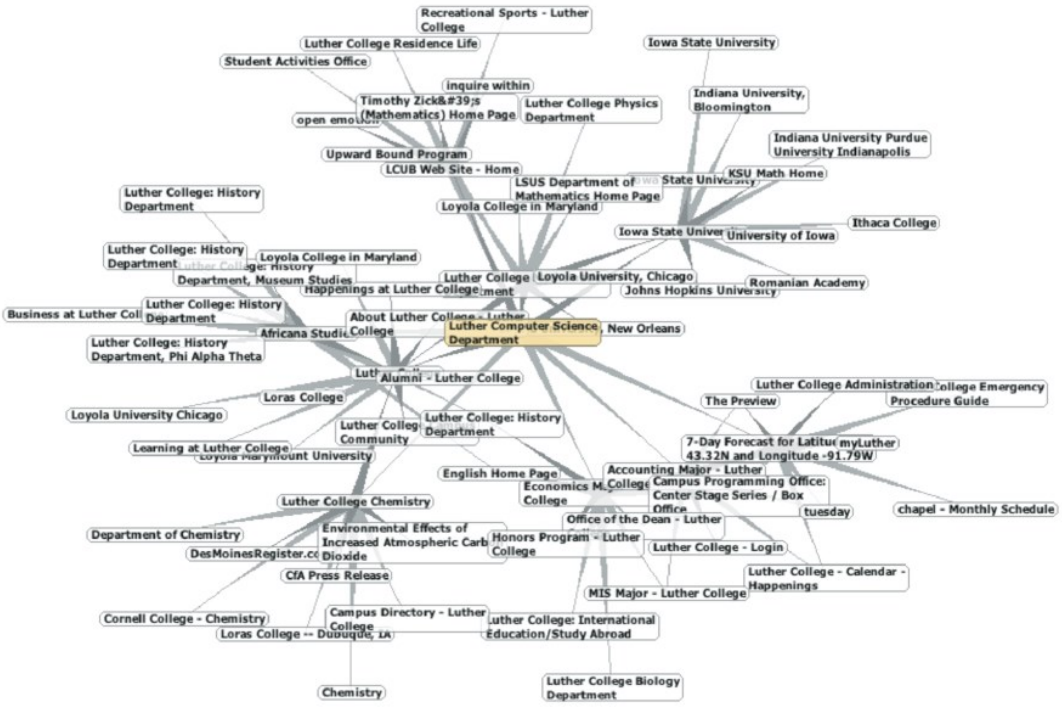


图 7.39：由源自路德学院计算科学主页的链接形成的图

如果你仔细研究了图 7.39，你也许会观察到一些有意思的现象。首先你也许会注意图中许多其他的网站也是路德学院的网站。其次，你也许会注意有一些链接指向了爱荷华的其他一些大学。第三，你可能发现有一些链接指向一些其他的人文艺术学院。

你可能会从中得出结论，这些将同一水平的网站聚合在一起的网页有一些潜在的结构。有一种图算法能够帮助我们发现图中高度相互关联的顶点，它被称为强连通分支算法**(SCC)**。我们正式在一张图 G 中定义一个**强连通分支** C，作为最大的顶点子集 $c \subset V$，因此对于其中每一对顶点 $v,w \in C$，我们均能使 w 和 v 相互连通。图 7.40 显示了一个仅有三个强连通分支的简易图，其中的强连通分支由不同的阴影区分开。

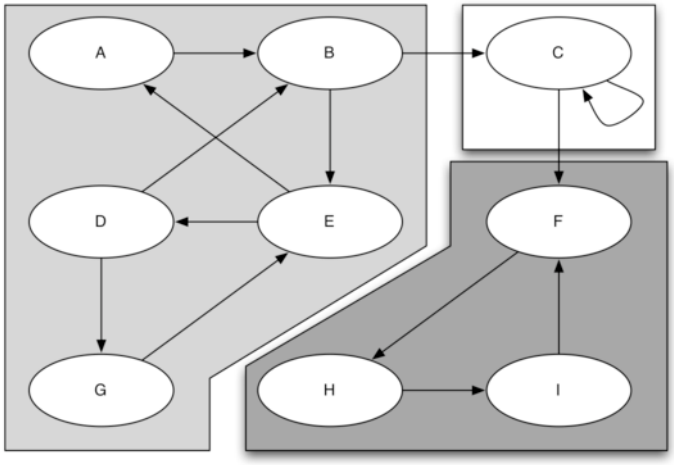


图 7.40：一张有三个强连通分支的有向图

一旦强连通分支被区分后，我们就可以将一个强连通分支中的所有顶点合并为单个更大的顶点从而简化图的形式。图 7.40 的简化形式见图 7.41。

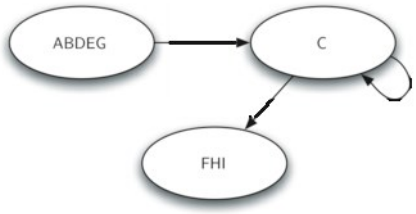


图 7.41：简化图

我们将再一次看到我们可以利用深度优先搜索开发强有力的算法。但在我们处理 SCC 算法之前我们必须考虑另一个定义。对于一个图 G，它的换置图被定义为所有的边都被逆转的一个新图 G^T。也就是说，如果原图 G 有一个从节点 A 至节点 B 的有向边，那么换置图 G^T 中会含一个从节点 B 至节点 A 的边。图 7.42 和 7.43 分别展示了一张简单图和它的换置图。

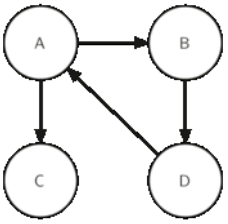


图 7.42：图 G

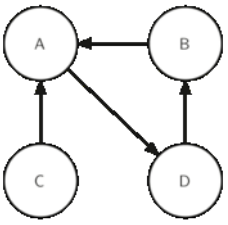


图 7.43：换置图 G^T

再看一遍上图。我们可以发现无论图 7.42 还是图 7.43 都有两个强连通分支。

我们现在可以描述该算法来计算整个图中的强连通分支。

1.调用深度优先搜索算法(DFS)计算图 G 中每个顶点的完成时间。

2.计算置换图 G^T

3.调用 DFS 计算图 G^T，但是当 DFS 主循环探索每个顶点时，减少完成时间。

4.步骤 3 生成的森林中每一棵树都是一个强连通分支，输出每一棵树每一个顶点的地址从而识别其分支。

让我们追溯图 7.40 中所描述的每步的操作。图 7.44 显示了对原图调用 DFS 算法时的起始和结束时间，图 7.45 显示了在转置图中运行 DFS 时起始和结束时间。

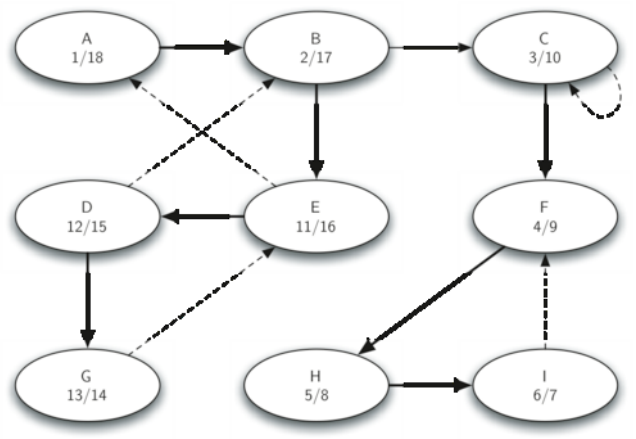


图 7.44：原图 G 的结束时间

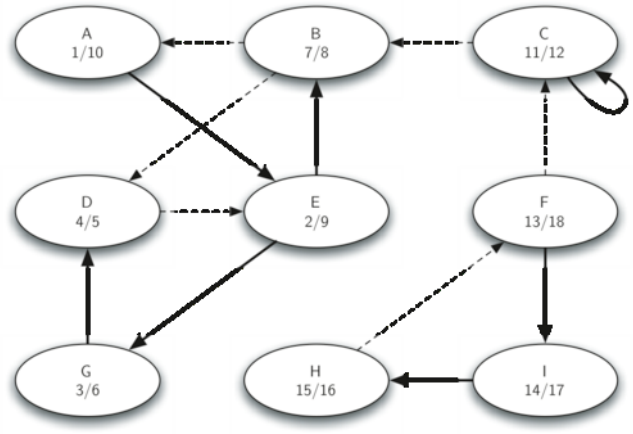


图 7.45：转置图 GT 的结束时间

最终，图 7.46 显示了强连通分支算法的步骤 3 中形成的三棵树组成的树林。你会注意到我们并没有提供给你 SCC 算法的 python 代码，因为我们将其留为测试题。

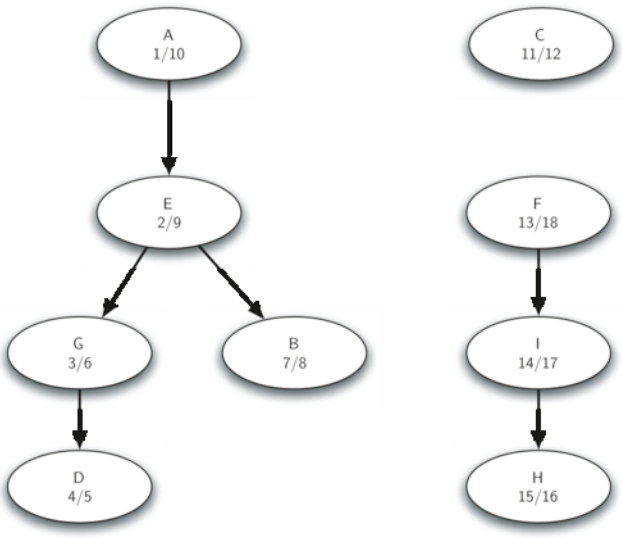


图 7.46：强连通分支

7.11 最短路径问题

当你浏览网页，发送邮件，或者在校园里另一个地方登录某实验室主机时，电脑后台会运行很多指令将你所使用的电脑的信息传输给另一台电脑。深入研究信息是如何通过网络在电脑之间传递

是计算机网络课程的首要课题。然而，我们讨论计算机网络的工作方式只是为了理解另一种十分重要的图运算法。

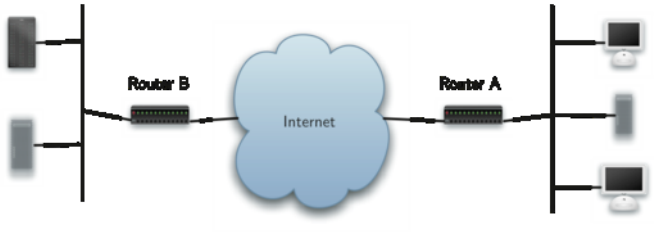


图 7.47：互联网连接概览

图 7.47 展示的是高度抽象概括后的互联网交流方式。当你使用自己的浏览器向服务器发出一个网页浏览请求时，这个请求必须遍历你当地局域网并且通过路由器发送到互联网上，在搜索过互联网后最终到达服务器所在局域网的路由器。然后你所要浏览的网页经由相同的路由器组传送回你的浏览器。图 1 中的云形标记里的“Internet”就是上述路径中额外需要的路由器，它们共同的任务是把你的信息从一个地方传输到另一个地方。如果你的电脑支持路由跟踪指令，你可以看到许多路由器在为你服务。下面的文本是路由跟踪指令的输出结果，它显示路德学院的网页服务器和明尼苏达大学的邮件服务器是通过 13 个路由器连接。

1 192.203.196.1

2 hilda.luther.edu (216.159.75.1)

3 ICN-Luther-Ether.icn.state.ia.us (207.165.237.137)

4 ICN-ISP-1.icn.state.ia.us (209.56.255.1)

5 p3-0.hsa1.chi1.bbnplanet.net (4.24.202.13)

6 ae-1-54.bbr2.Chicago1.Level3.net (4.68.101.97)

7 so-3-0-0.mpls2.Minneapolis1.Level3.net (64.159.4.214)

8 ge-3-0.hsa2.Minneapolis1.Level3.net (4.68.112.18)

9 p1-0.minnesota.bbnplanet.net (4.24.226.74)

10 TelecomB-BR-01-V4002.ggnet.umn.edu (192.42.152.37)

11 TelecomB-BN-01-Vlan-3000.ggnet.umn.edu (128.101.58.1)

12 TelecomB-CN-01-Vlan-710.ggnet.umn.edu (128.101.80.158)

13 baldrick.cs.umn.edu (128.101.80.129)(N!) 88.631 ms (N!)

Routers from One Host to the Next over the Internet

Routers from One Host to the Next over the Internet

互联网上每一个路由器都连接着一个或多个其他的路由器。所以如果你在同一天中的不同时间段执行路由跟踪指令，很有可能发现你发送的信息在不同的时间经过了不同的路由器。这是因为任何两个路由器之间的连接都不是没有代价的，而是受线路拥挤情况、时间以及许多其他因素的影响。这样，当你知道我们可以用一张含有权边的图来代表路由器之间的网络时，就不会感到惊讶了。

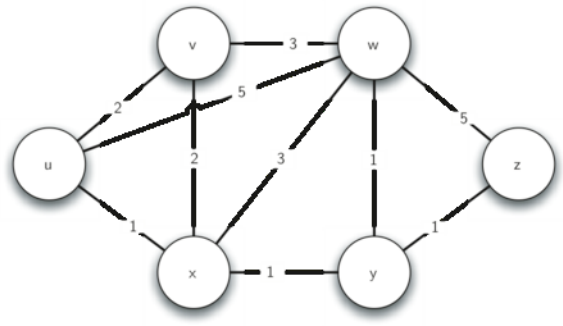


图 7.48：互联网路由器之间的连接和权重

图 7.48 展示了一个用有权重图来代表互联网路由器之间相互连接情况的小例子。我们想要解决的问题是找到一条总权重最小的路线，并且使之实现传递任何信息的功能。这个问题听起来很熟悉，因为它与我们用广度优先算法解决过的问题很相似，区别在于现在我们要考虑的是整条路线的总权值，而不是路线中的节点数。需要特别注意的是，如果所有边的权值都相等，那么这两个问题是等价的。

7.11.1. DIJKSTRA 算法

我们即将使用的选择加权最短路径的算法被称为Dijkstra算法。“Dijkstra”算法是一 种迭代算法，用来提供从一个确定的开始节点到所有图中其他节点的最短路径，这与广度优先搜索的结果很类似。

为了追踪从开始节点到每一个结束节点的总权值，我们将在顶点Vertex类中使用dist 作为实例变量。成员dist包含从起点到目的节点的最小权值路线的当前总权值。对于图中的每一个节点，算法均会重复进行一次，但各顶点重复的先后顺序是由一个优先队列控制的，队列中决定顺序的参量是dist值。当节点被创建时dist被设置成一个很大的数，虽然理论上说应该将其设为正无穷但实际操作中我们只需要将它设置成比实际问题中可能出现的任何距离都要大的值即可。

Dijkstra 算法的代码如表代码 7.11 所示。当算法结束时各个距离都被正确地设置成表中每节点之间的前驱连接。

```
from pythonds.graphs import PriorityQueue, Graph, Vertex

def dijkstra(aGraph,start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in aGraph])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() \
                    + currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
            nextVert.setDistance( newDist )
            nextVert.setPred(currentVert)
            pq.decreaseKey(nextVert,newDist)
```

代码 7.11

Dijkstra 算法使用了优先队列，回忆一下，在学习树的时候我们曾经基于“堆”结构实现了优先队列。简单的优先队列实现与 Dijkstra 算法中的优先队列实现有一些不同。首先，优先队列 PriorityQueue 类储存了键值对元组，这对于 Dijkstra 算法来说非常重要，因为优先队列中的 key 值必须与图中节点的 key 值匹配，同时，键值对中的值用来决定优先级，也决定了 key 在优先队列中的位置。在这种实现优先队列的方法中，我们把节点间的距 离作为优先级，因为我们都知道，在探索下一个节点时，我们总是想要探索有最小距离的节 点。第二点不同之处是添加了 decreaseKey 方法。正如你所看到的，当队列中已经存在了到 某个节点的距离，而这个距离减小的时候，这个节点将被移到队首。

让我们沿着下面几张图片的顺序理解 Dijkstra 算法的一个应用实例。开始节点是 u， 与其相邻的三个节点是 v，w 和 x。因为开始时到 v，w，x 的距离都被初始化成 sys.maxint， 从起始节点到这三个节点的代价就是它们的直接代价，所以我们更新这三个节点的代价值。 同时，设置每个节

点的前驱节点为 u，并且把距离作为 key 值将每个节点都放进优先队列中。算法当前状态见图 7.49。

下一次 while 循环中我们考察与 x 相邻的节点。之所以先考察 x 是因为 x 的总代价最小，因此被弹到优先队列的顶端。对于 x 节点，我们考察与它相邻的节点 u，v，w，和 y。对于 每一个相邻的节点，我们要检查经由 x 节点的当前节点的距离是否小于已知的距离。明显可 以看出 y 满足条件，因为它的距离是 sys.maxint。节点 u，v 不满足条件，因为它们各自的 固有距离是 0 和 2。然而，我们现在可以知道，经由 x 到 w 的距离小于直接从 u 到 w 的距离。 因此需要更新 w 的距离值，并将 w 的前驱节点从 u 改为 x。算法当前状态见图 7.49。下一步是检查与 v 相邻的顶点（见图 7.51）。这一步对于图本身无影响，所以我们转而检查节点 y。此时发现，对于 w 和 z 来说经由 y 代价更小，所以如上段所述调整其距离值和前驱节点（见图 7.52）。 最后检查节点 w 和节点 z（见图 7.52 和图 7.54），然而没有发现需要改变的地方，因此优先队列变为空的，Dijkstra 算法结束。

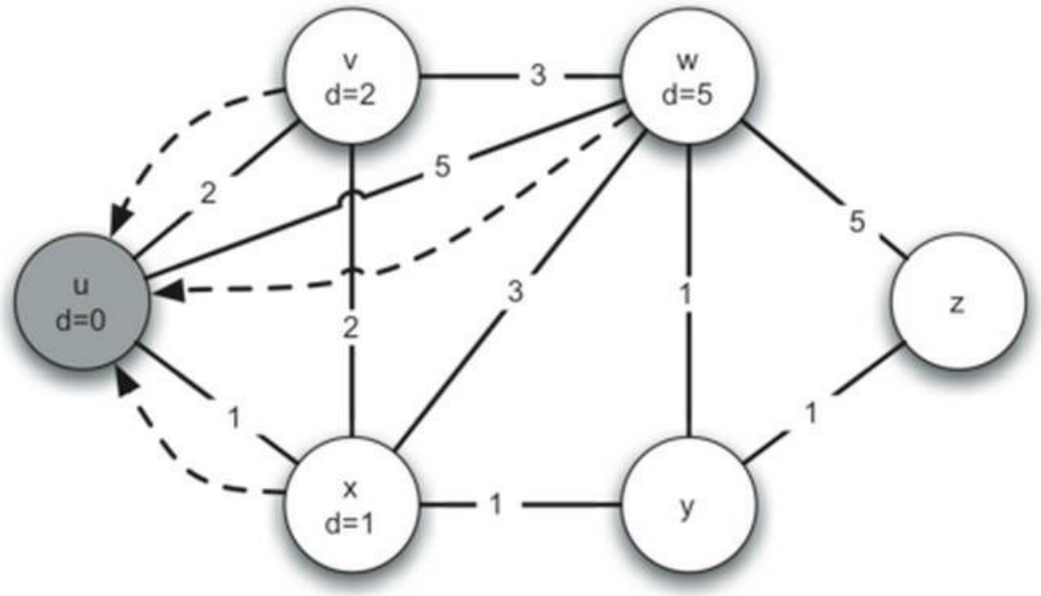


图 7.49：追踪 Dijkstra 算法

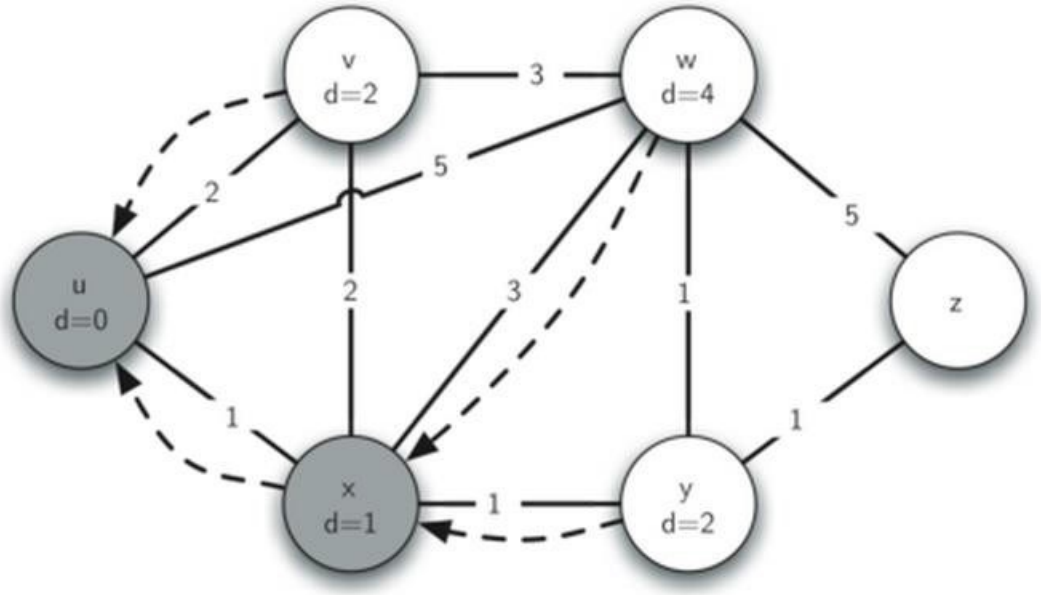


图 7.50：追踪Dijkstra算法

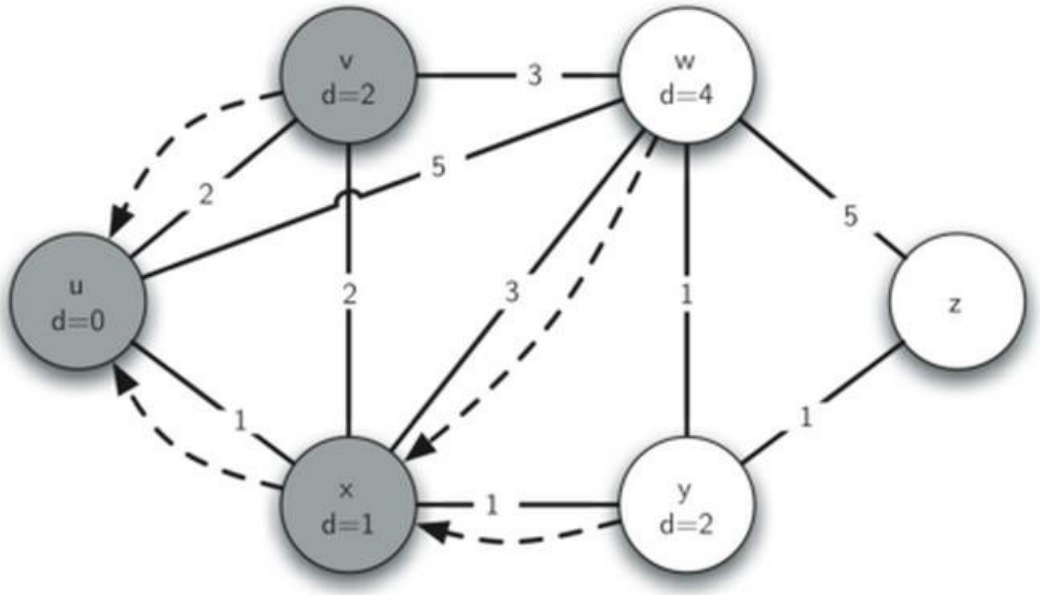


图 7.51：追踪Dijkstra算法

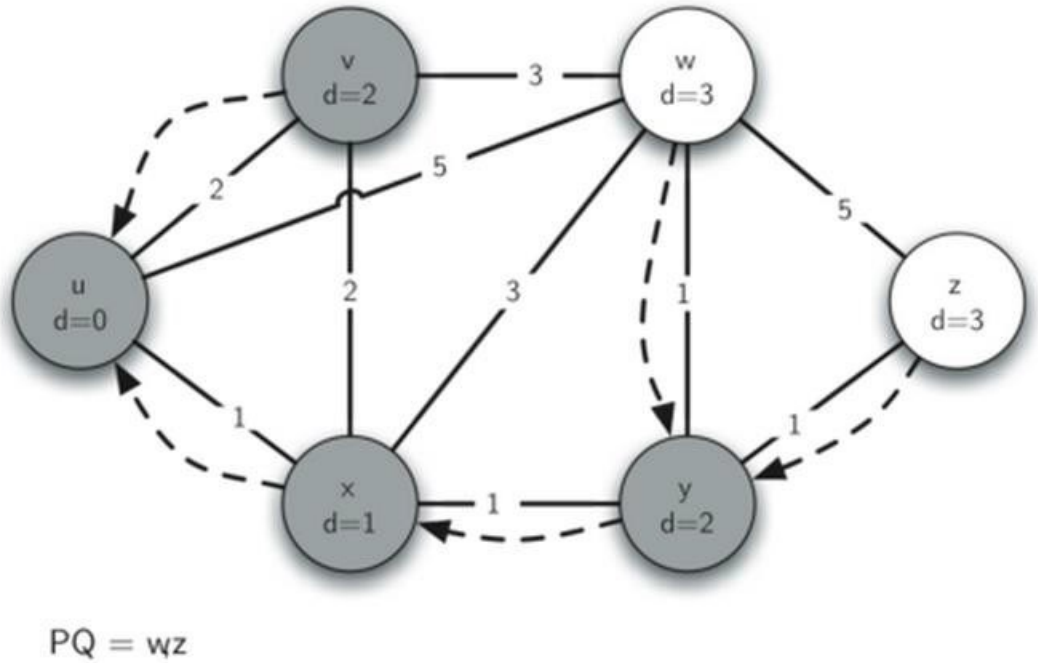


图 7.52：追踪Dijkstra算法

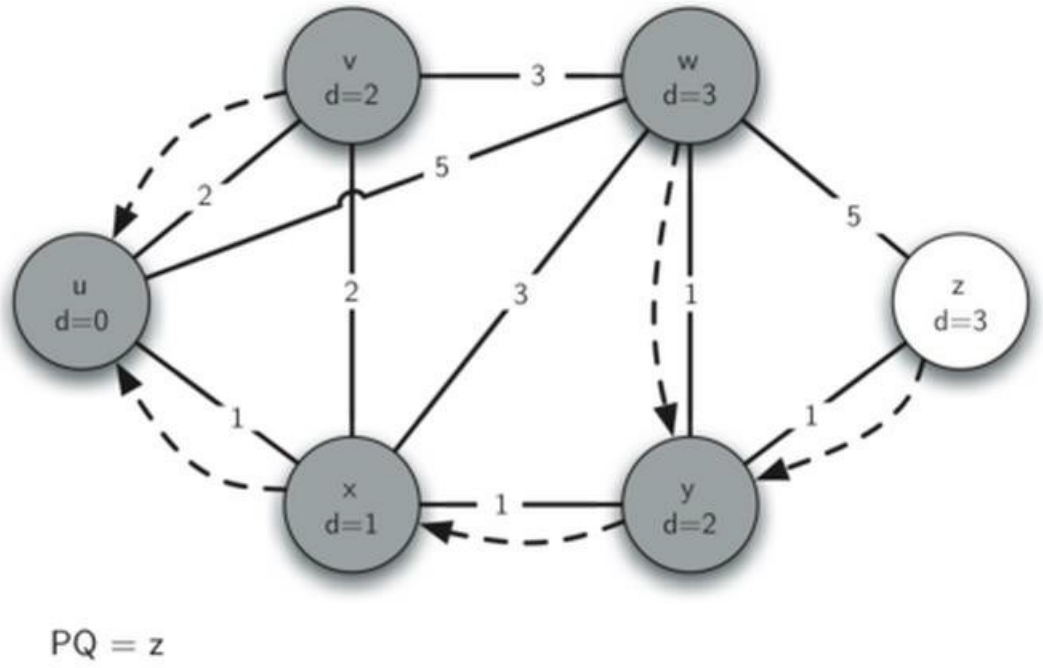


图7.53：追踪Dijkstra算法

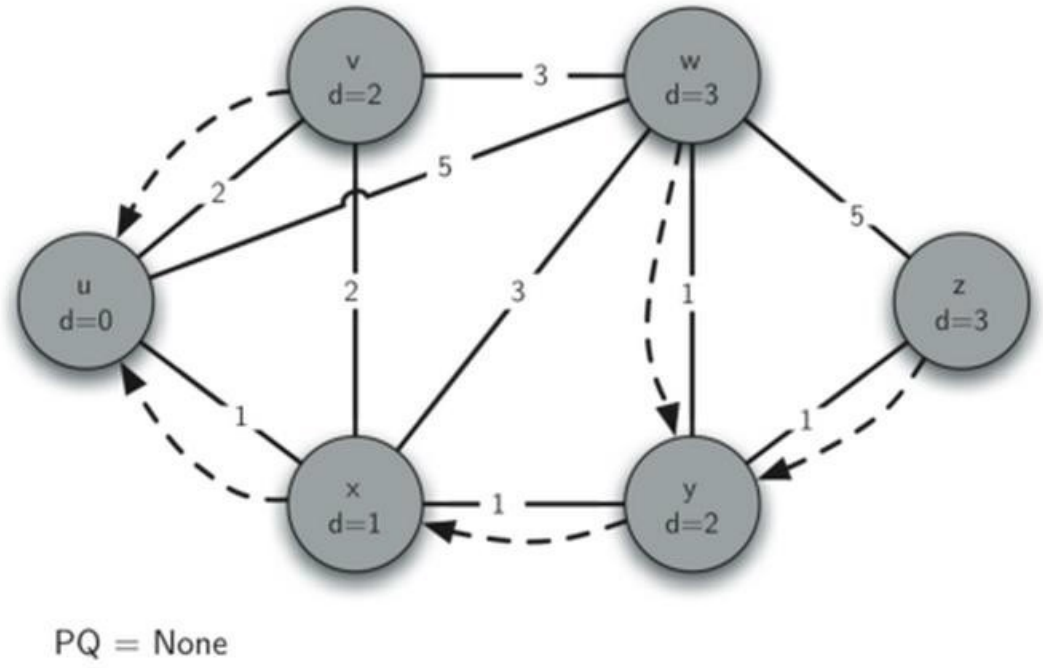


图 7.54：追踪Dijkstra算法

十分重要的一点是，Dijkstra 算法只适用于所有权值都为正数的情况。如果在某条边引入了负的权值，那么整个算法将陷入无限循环。

我们需要注意的是，为了在互联网上传递信息，其他的算法也被用来寻找最短路径。在 互联网问题上使用Dijkstra算法的一个问题是你必须有完整的图结构否则算法无法运行。这暗示着每个路由器需要拥有互联网上所有路由器的完整地图实际操作中这是不可能做到的，因此其他的算法允许路由器在传递信息的过程中发现新的图结构。这种算法被称为“距离向量”路由算法。

7.11.2. DIJKSTRA 算法分析

最后，进行Dijkstra算法的时间复杂度分析。首先，由于我们最初把图中的每一个节点都加入了优先队列，所以建立优先队列的时间复杂度为 O(V)。建立完队列之后，while 循环对于每个节点都会执行一次。因为所有顶点在一开始就都被加入，并且仅当循环执行完 后才会被移出。循环体内部 delMin 操作的时间复杂度为O（Vlog（V）For循环对于图中的每条边都会执行一次，在循环体内部 decreaseKey 操作的时间复杂度为 O（Elog（V） 所以结合起来，总的时间复杂度为O（V+E）log（V）

7.12.PRIM 最小生成树算法

为了说明我们的最后一个图算法，让我们来考虑一个在线游戏设计者和无线电网络开发者面对的问题。这个问题是，他们想要高效地传递信息给那些可能正在收听的人。这在游戏中非常重要，它使每一名玩家都可以了解到其他玩家的实时位置这在无线电网联络中也很重要，它使每一名接入频道的听众可以得到他们需要的全部数据去重现他们正在收听的歌曲图 7.55 向我们阐明了这个广播问题。

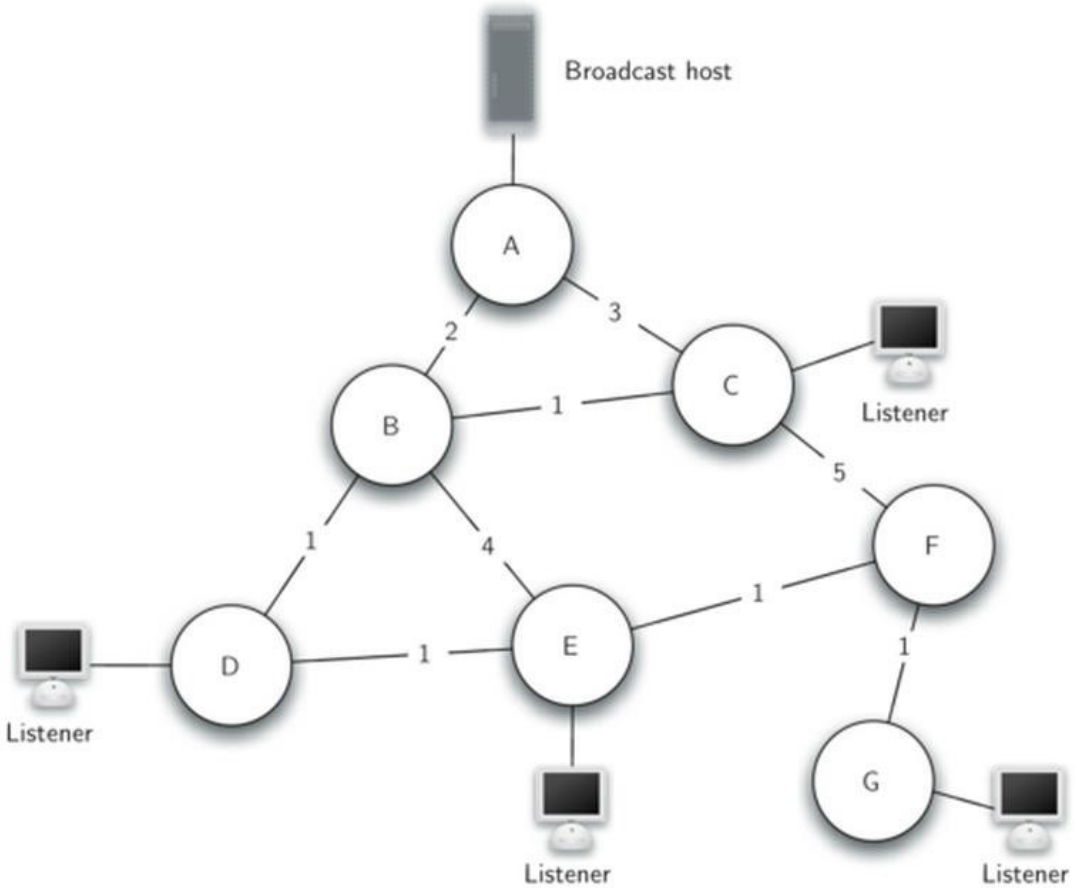


图 7.55：广播问题

关于这一问题有一些无脑的强制解决办法所以让我们先来看一看这些解决办法来帮助我们更好的理解广播问题。这也会在我们解决这一问题时让你更加欣赏我们给出的解决方法。在开始的时候，广播主持有一些所有听众都需要接收的信息。最简单的解决方式是广播主持有一个所有听众的名单然后向每一名听众逐一发送信息在图 7.55 中我们展现了一个小的 网络包括一个广播和一些听众。用第一种方式，每一段信息需要发送四次。假设使用的是最短路径，让我们来看看每一个路由器需要处理多少次相同的信息。

每一段从广播发出的信息都经过了路由器A，因此A看到了每一段信息的四次拷贝。路 由器C只看到了面向它的听众的信息的一次拷贝然而路由器B和D会看到每一条信息的三次拷贝因为路由器B

和 D在通往听众1,2,3的最短路径上当你考虑到播音主持每秒要发送上百条信息给一个收音机，这就会成为一个巨大的额外负担。

一个无脑的强制解决办法是播音主持每份信息只发送一份然后让路由器来把它们分类。在这种情形下，最简单的解决方式是一种叫做无控制流动的方法。这种方式根据安排工作。每段信息的开始时间(ttl)值设置为大于或者等于播音主持到距他最远的收听者之间距离的一些数。每一个路由器复制一份信息然后把信息传递给所有与它相邻的路由器。当信息传递 时 ttl 降低。每一个路由器持续地向与它相邻的所有路由器传递信息拷贝，直到 ttl 值变为 0。显然无控制流动的方法会比我们的第一种方法产生更多的不必要的信息。

这种解决方式属于最小重量生成树的一种。形式上我们给图G=(V,E)定义了如下的最小生成树T。T 是一个连接 V 中所有顶点的 E 的非循环子集。T 中边界重量的和是最小化的。

图 7.56 展示了一个简化版的广播图并且着重突出了在图中形成最小生成树的边界。现在 为了解决我们的广播问题，广播主持向网络中简要发送了一份播音信息的拷贝。每一个路由 器向作为生成树一部分的任意相邻路由器发送信息，包括刚刚向它发送信息的相邻路由器。 在此问题中，A 向 B 发送信息，B 向 D 和 C 发送信息，D 向 E 发送信息，E 向 F 发送信息，F 向 G 发送信息。没有路由器会重复接收同一份信息，并且所有对其感兴趣的听众都能够看到这一信息。一个连接V中所有顶点的E的非循环子集。T中边界重量的和是最小化的。

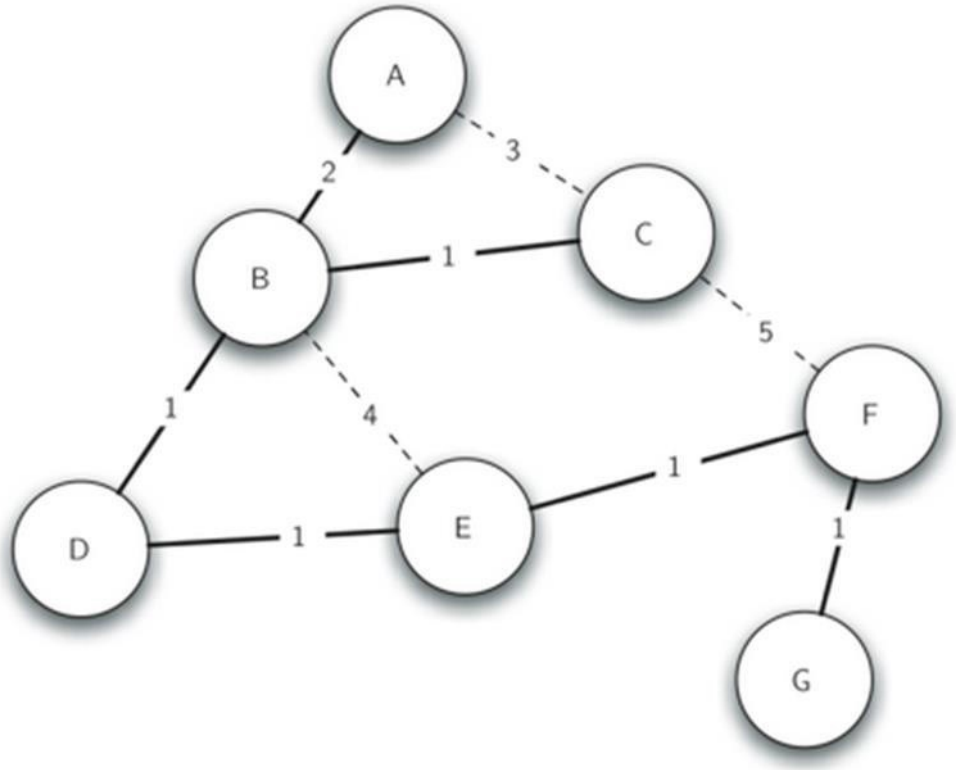


图7.56：广播图的最小生成树

我们将要用来解决这一问题的算法叫做 Prim 算法。Prim 算法隶属于“贪心算法”一类，因为每一步我们都会选择最简单的下一步。在这种情形下最简单的下一步将会沿着最低重量 的边缘。

我们的最后一步是开发 Prim 算法。

最基本的想法是构建一个如下的生成树：

```
While T is not yet a spanning tree
    Find an edge that is safe to add to the tree
    Add the new edge to T
```

步骤中的窍门是指引我们去找到一条安全的边界”。我们像定义任意边界那样定义安全边界，即连接在生成树中的一个顶点和不在生成树中的一个顶点。这确保了这个树总会保 持一个树的形状而不会产生循环。

执行 Prim 算法的 Python 代码在 代码 12 中展示。Prim 算法与 Diikstra 算法相似，因为他们都用优先队列去选择下一个添加到图中的顶点。

```
from pythonds.graphs import PriorityQueue, Graph, Vertex
def prim(G,start):
pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert) \
                    + currentVert.getDistance()
          if nextVert in pq and newCost<nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(nextVert,newCost)
```

代码 12

接下来的这些图（图7.57 到图 7.63) 展示了算法在样品树上面的运行情况。我们将顶点A作为起始点。到其他任意点的距离被初始化为无穷。看着 A 的相邻点我们可以更新顶点 B 和 C 的两个距离，因为A到B和C的距离小于无穷。这时B和C已到了优先队列的前端。更新之前 B和C的连接通过将它们指向A。需要着重强调的是我们还没有将B和C添加到生成树中。一个点在没有被移出优先队列之前是不会被添加到生成树中的。

既然 B 有着最小距离，接下来我们来讨论 B 点。检查B 的相邻顶点我们发现 D 和 E 可以被更新。D和E都能够得到新的距离值并且他们的之前连接可以被更新。移至优先队列 的下一个点我们找到了C。优先队列中唯一与C相邻的点是F，然后我们可以更新到F的距 离并且调整F在优先队列中的位置。

现在我们寻找点D的临近点。我们发现我们可以更新E并且将到E的距离从6减到4。当我们这么做时我们改变了从E指回D的连接，然后准备把它植入生成树的不同位置。剩下 的就是像你期待的那样，算法开始，将每一个新的点加入树中。

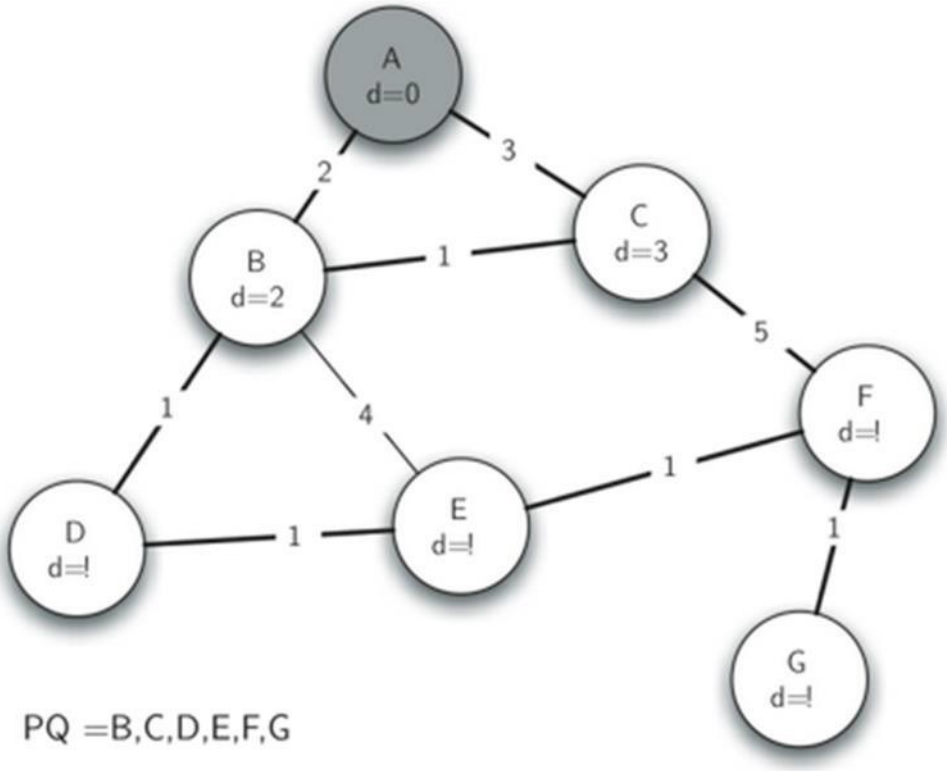


图 7.57：描绘Prim算法

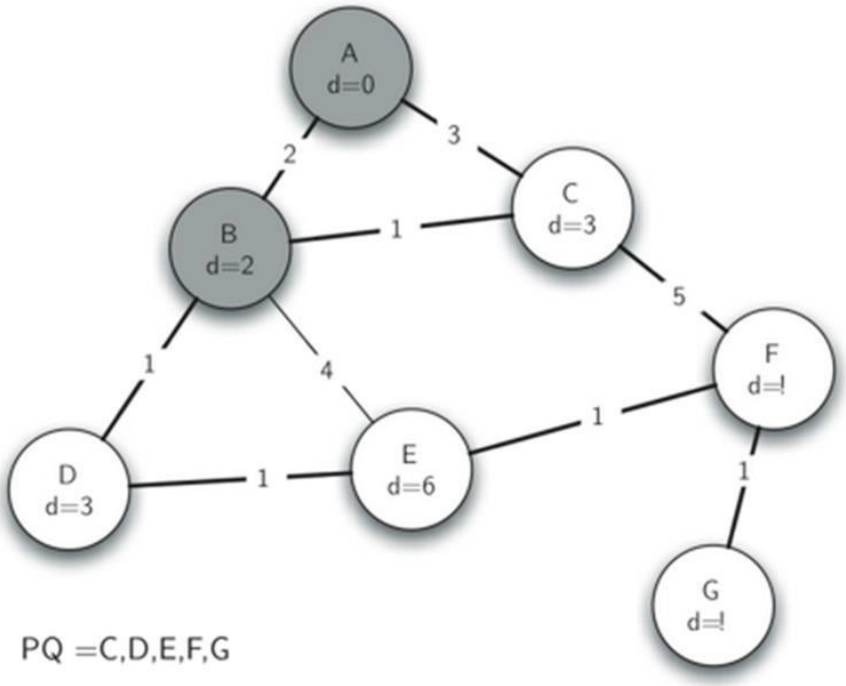


图 7.58：描绘 Prim 算法

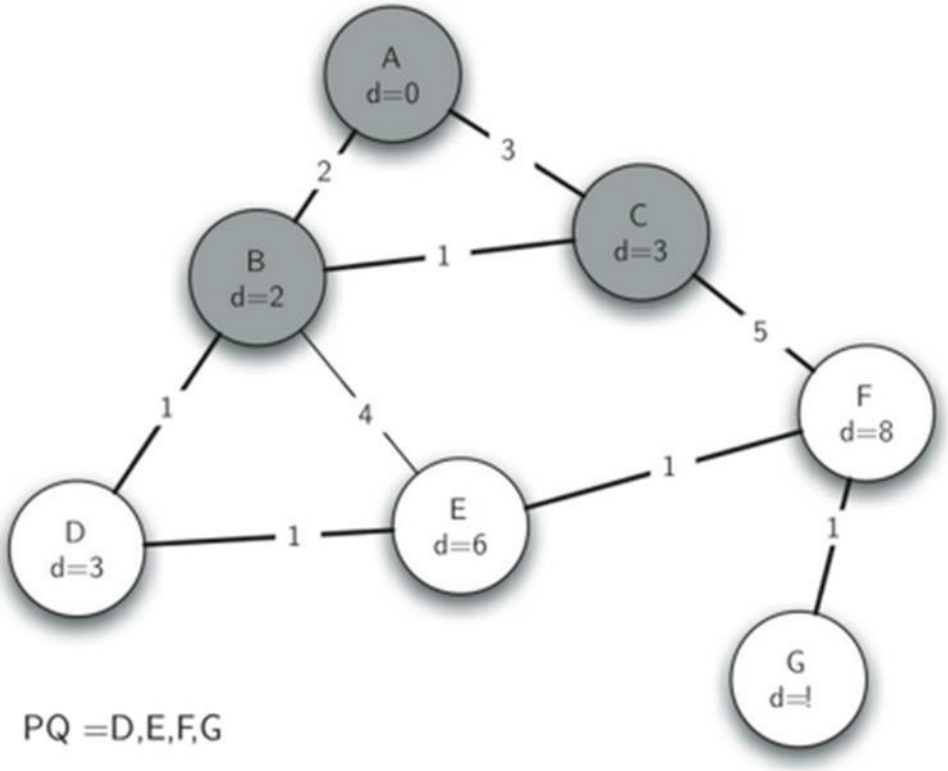


图 7.59：描绘Prim算法

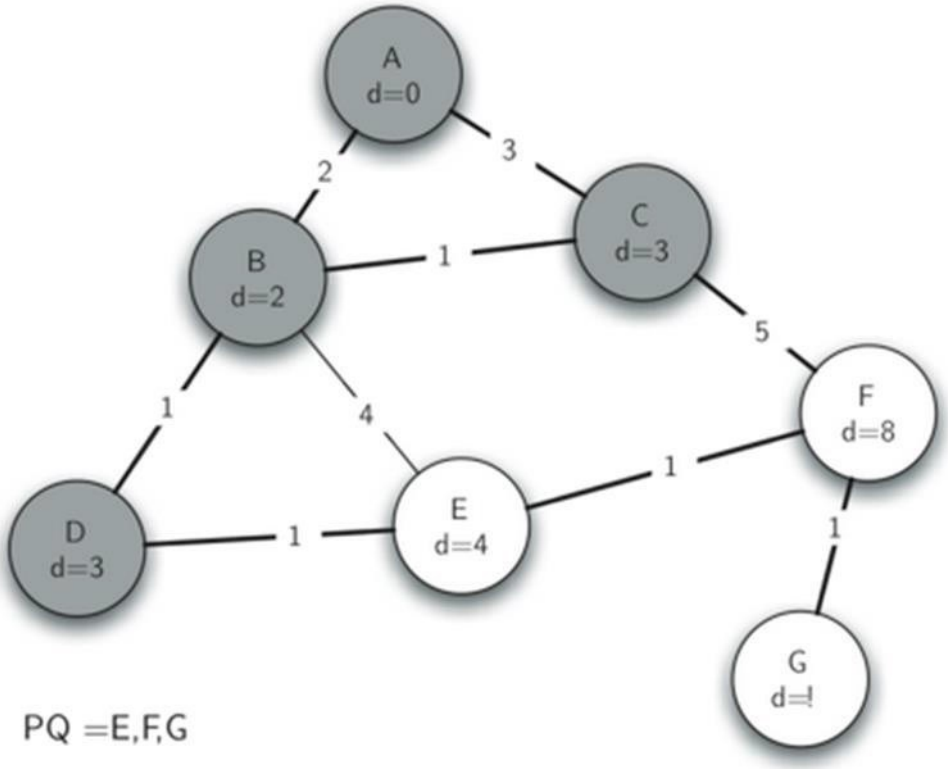


图 7.60：描绘Prim算法

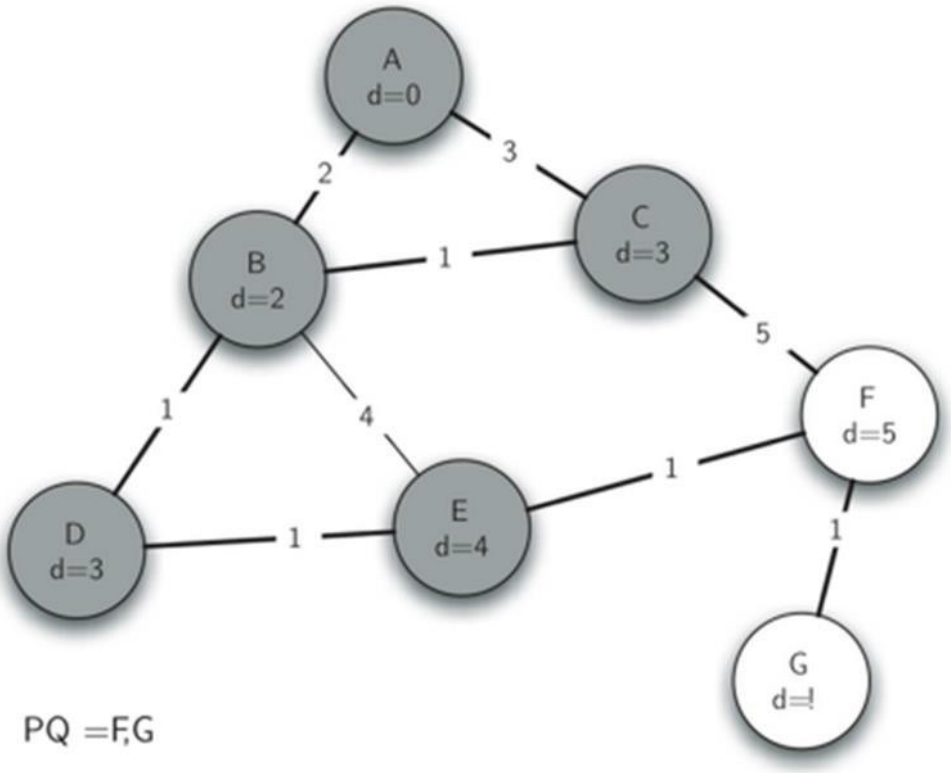


图 7.61：描绘Prim算法

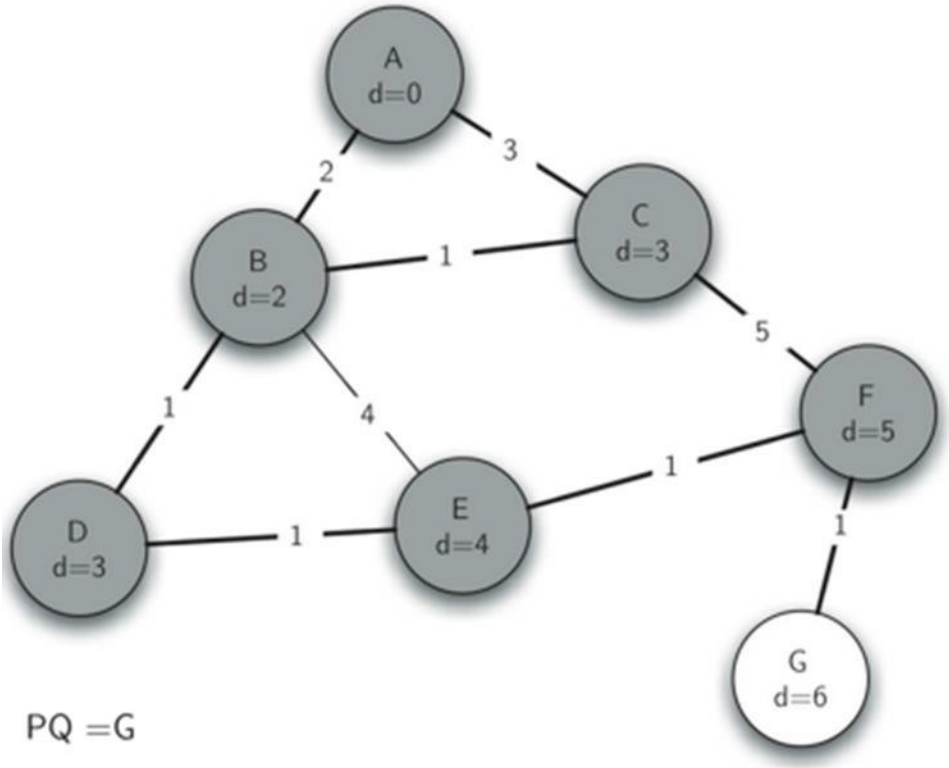


图 7.62：描绘Prim算法

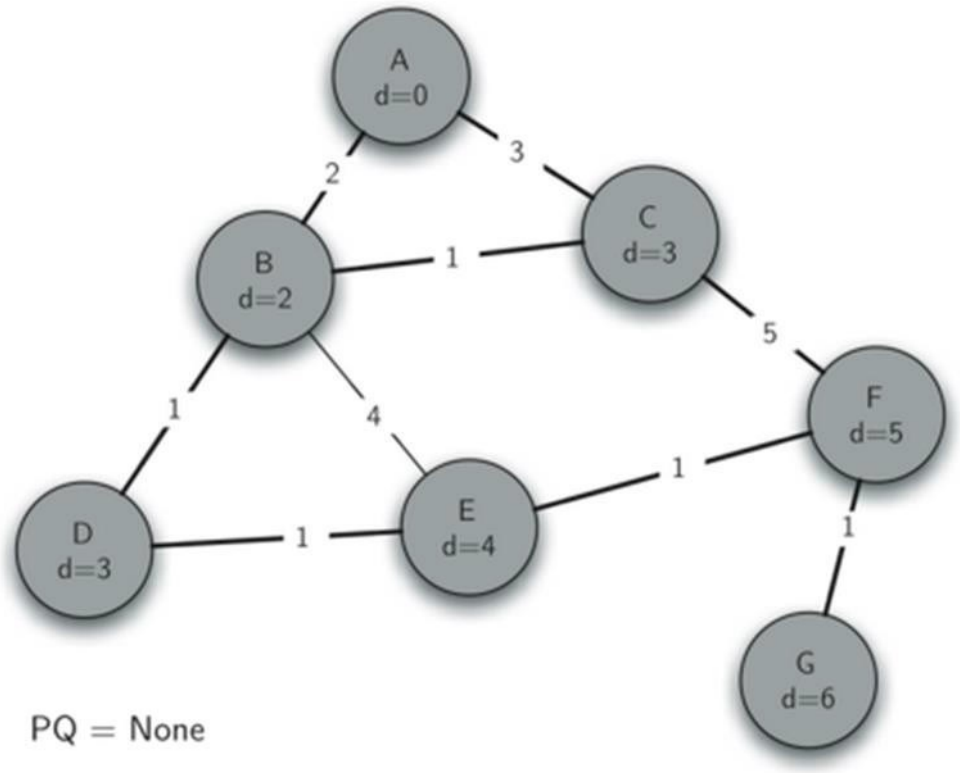


图 7.63：描绘Prim算法

7.13. 小结

本章我们学习了图抽象数据类型，以及若干实现方法。假如我们可以把一个原始问题转化成可以被表示成图的事物，我们就可以用图来解决许多问题。特别地，我们发现图在以下领域可以解决很多问题：

- 广度优先搜索算法BFS，解决无权图的最短路径问题
- 带权图的最短路径算法Dijkstra算法
- 图的深度优先搜索算法
- 用于简化图的强连通分支算法
- 用于排序任务的拓扑排序算法
- 用于广播消息的最小生成树算法

7.14 . 关键词

无圈图	邻接表	邻接矩阵

邻近的	广度优先搜索 (BFS)	圈
有圈图	有向无圈图	深度优先森林
深度优先搜索 (DFS)	有向图	有向无圈图 (DAG)
有向图	边的权重	边
括号性质	路径	最短路径
生成树	强连通分支 (SCC)	拓扑排序 & 不受限洪水
顶点	权重 t	

7.15 问题讨论

1、画出与下图所示邻接矩阵对应的图。

2、画出与下图所示的表示边的表格中对应的图。

3、去掉权重，在上个问题的图中实现广度优先搜索。

4、buildGraph 函数的大 O 时间复杂度是多少。

5、导出拓扑排序算法的大 O 时间复杂度。

6、导出强连通分支算法的大 O 时间复杂度。

7、分步展示 Dijkstra 算法在上面所示的图中的应用。

8、用 Prim 算法，找到上述图中的具有最小权重的生成树。

9、画出关系图来解释你发一封邮件所需的步骤。在你的图中执行一次拓扑排序。

10、导出表达骑士周游问题执行时间的指数的底的表达式。

11、解释为什么一般的深度优先搜索算法不适合解决骑士周游问题。

12、Prim 最小生成树算法的大 O 时间复杂度是多少。

7.16 编程练习

1. 修改深度优先搜索函数产生一个拓扑排序。

2. 修改深度优先搜索生成强连通分支。

3. 写出图的 transpose 方法。

4. 使用广度优先搜索，写一个算法来确定从每个顶点到其他所有顶点的最短路径（最短路径问题）。

5. 用广度优先搜索修改递归章节的迷宫程序，找到走出迷宫的最短路径。

6. 写一个程序来解决以下问题：你有两个罐子，一个是 4 加仑、另一个是 3 加仑。每个罐子上都没有标记。有一个泵，可以用来为罐子满上水。你如何使用 4 加仑的水罐得到两加仑的水？

7. 将上述问题一般化，你解决方案的参数包括每一个罐子的大小和最后留在较大罐子里的水量。

8. 写一个程序来解决以下问题：三个传教士和三个食人族到河边，发现一艘船一次只能载两人。每个人都要过河来继续旅程。然而，如果任意一边的岸上的食人族数量超过传教士，传教士将被吃掉。找到使每个人都安全地到达河的另一边的所有路径。