致 谢

2015 地空数算课程教材 3.4～4.4 翻译小组

组长

高鸿宇

成员

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

本文为 3.4 到 4.4

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，叫做“队尾”（rear），已有元素的移出在队列的另一端，叫做“队首”（front）。当一个元素被加入队列之后，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，在队列里已停留时间最长的元素处于队首。这种顺序的规则有时候叫做“先进先出”（FIFO），有时候也叫做“先到先服务”。

队列最简单的例子就是我们每天都在排的队伍。我们排队等着看电影，在杂货店的结账队伍里等着付钱，我们在自主餐厅里也排队（这样我们就能从盘子组成的“栈”（数据结构，详见前一章）里边拿出我们的盘子）。一个遵守秩序的队伍，或者队列，对于进出都只有一条通道这一点具有非常严格的规定。不可以插进队列的中间，在到达队首之前也不能提前离开队列。图 1 是一个简单的由 Python 数据对象构成的队列：

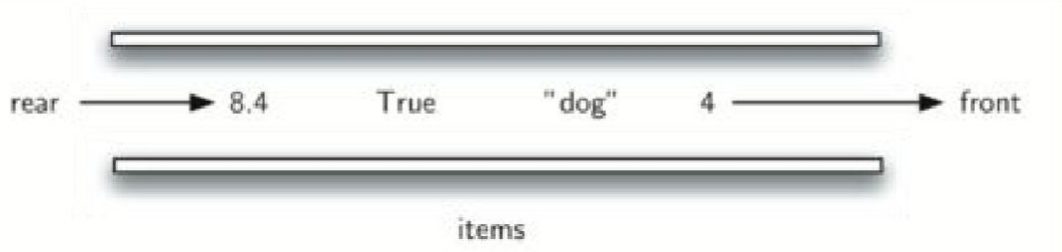

图 1 一个由 Python 数据对象构成的队列

计算机科学中有许多常见的关于队列的例子。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”一个“队伍”，其中有正在等待中的其他打印任务。最先进入队伍的任务就是接下来要完成的。如果你的任务是队伍中的最后一个，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，通常会通过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的显示更快。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，它们从一端进入队列，这一端叫做“队尾”，再从另一端被移出队列，这一端叫做“队首”。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，返回值为 Queue 对象；
·enqueue()将数据项添加到队尾，无返回值；
·dequeue()从队首移除数据项，返回值为队首数据项；
·isEmpty()测试是否为空队列，返回值为布尔值；
·size()返回队列中的数据项的个数。
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 1 列出了对队列进行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 是用 enqueue 方法最早被加进队列的元素，因此它被 dequeue 方法最早返回。

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 1　队列操作示例

3.4.3.在 Python 中实现 Queue

为了执行抽象数据类型队列，创建一个新类是再好不过的方式了。就像之前一样，我们还是利用强大而简单的列表来帮助建立队列的内部表示。
我们需要决定列表的哪一端做队尾，哪一端用来做队首。下面的一段执行代码假设队列的队尾在列表的 0 位置处。这使得我们能够利用 list 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)。

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

代码 1 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 1

对这个队列的进一步操作将会给出下面的结果：

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

自我检测

Q-9：假设你进行了下面的一系列队列操作：

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

队列里面剩下了什么元素？

a)‘hello’,‘dog’

b)‘dog’,3

c)‘hello’,3

d)‘hello’,‘dog’,3

3.4.4. 模拟算法：热土豆

队列运作方式的一个典型实例是模拟出一个需要运用到先进先出（FIFO）管理数据方式的真实情景。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 2）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。

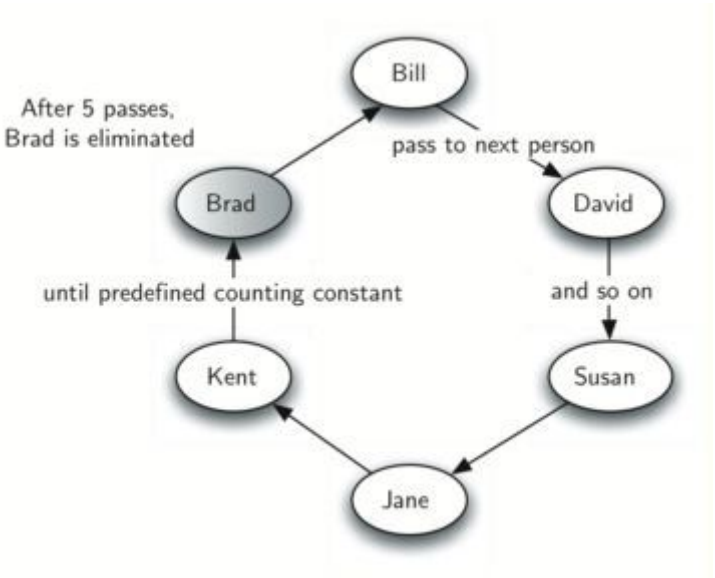

图 2 六人传土豆游戏

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。当最终时刻来临时，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。

我们可以大体上实现对于热土豆问题的模拟。我们的程序将要读入一个名字列表和用作计数的常数“num”。在经过重复多次基于num的计数后，程序将返回最终剩下的人的姓名。这时发生什么就是你所决定的了。

为了模拟这个环状结构，我们会用到队列（见图3）.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从队尾把她加入进队列。在所有站在她前面的人都轮过一遍后才会再次轮到她传土豆。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）。

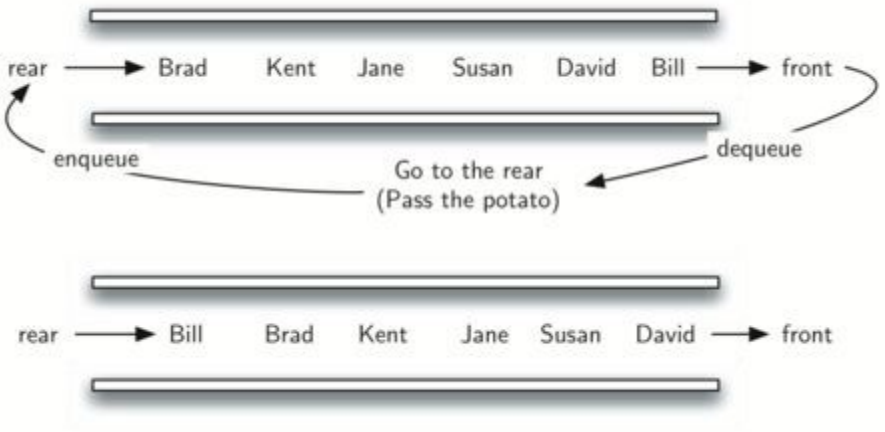

图3 用队列模拟热土豆问题

具体程序展示在代码1中。当热土豆函数以7为计数常数时被调用会返回Susan。

```
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1 热土豆问题的模拟实现

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。但这并不成为问题因为队列像是一个圆圈运作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。对于这个实现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。

3.4.5. 模拟算法：打印任务

一种更有趣的模拟算法可以用来分析本节前面提过的打印任务。学生们把待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。这种配制方式中蕴含着很多问题。最重要的问题应该是打印机是否可以处理确定数量的任务，如果不可以，学生们可能会等太长时间以至于错过下一堂课。

考虑计算机实验室中的以下情况：平均每天每小时有大约 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；如果转换成较高品质打印，则每分钟只能打印 5 页。较慢的打印速度可能会使学生平均等待时间过长。应该采取哪种打印模式？

我们可以建立模型来模拟这个实验。我们需要建立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没有完成的打印任务。我们关心的是学生平均的等待打印时间，这与一个任务在队列里的平均等待时间是等同的。

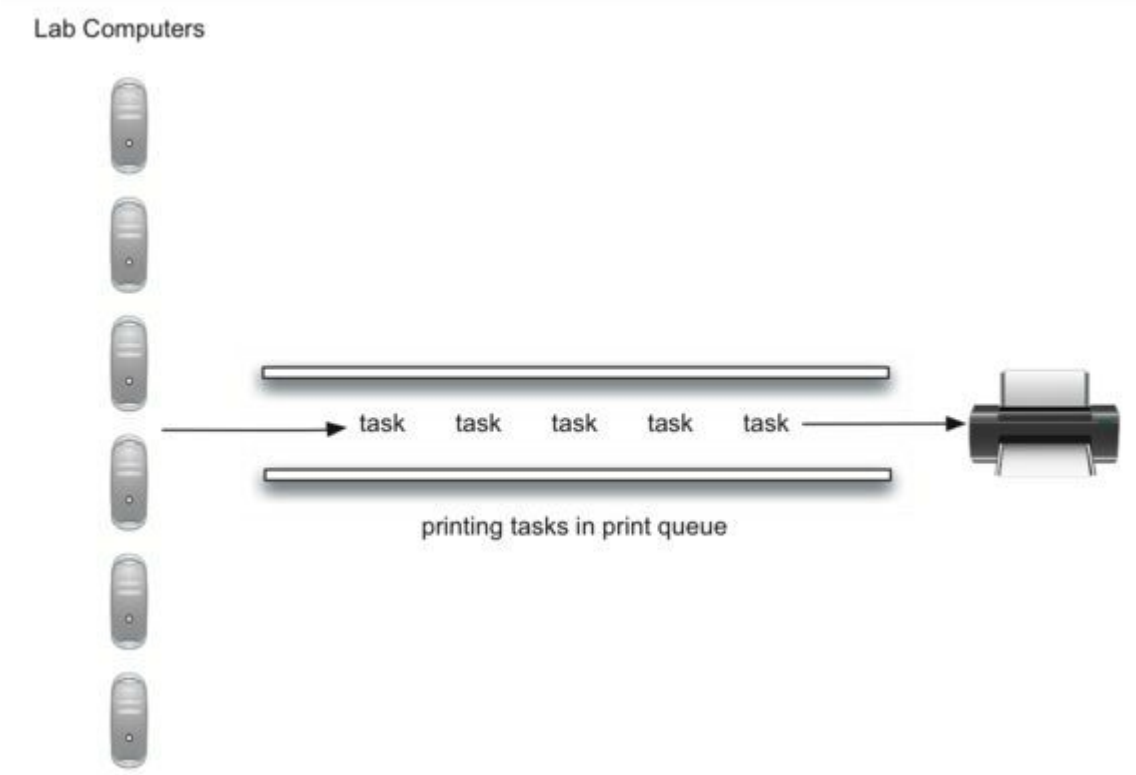

图 4 关于打印任务队列的计算机科学实验室

为了模拟出这个情景，我们需要使用一些概率。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 之间等概率取值，那么模拟的打印任务的长度就可以用 1-20 之间的随机数来生成，以保证 1-20 页长度的出现概率相等。

如果有 10 个学生在实验室里并且每人提交两个任务，那么平均每小时就有 20 个打印任务。那么对于每一秒来说，创建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。

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

每一秒钟我们可以用生成 1-180 范围内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，即认为一个任务被创建。注意有时候很多任务可能会接连生成，有时候可能需要等待很长时间才会生成一个任务，而实际情况就是这样。在知道了一些基本因素之后，我们模拟的目的是尽可能地接近真实情况。

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（打印过程中当前的每一秒 currentSecond）：

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

（2）如果打印机不在工作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从当前时间（currentSecond）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（剩余打印时间减一）。

（4）如果这个时候任务已经打印完了，也就是说剩余打印时间为 0 时，打印机就进入了不在工作的空闲状态。

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，我们需要为打印机、打印任务和打印队列这三个对应真实世界的对象自定义一些类。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。如果正在执行则表示打印机正忙（13—17行），需要等待的时间可以由正在运行的打印任务中的需打印张数求得。同时构造函数还能完成单位时间打印张数的初始化设置。方法tick用于减缩内设的任务完成所

需时间，并在一次任务结束后将打印机设置为闲置（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）用于代表一个单独的打印任务。当一个任务被创建时，通过调用特定方法产生1到20中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。

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

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中的时间。方法waitTime用于检索任务开始打印前的等待总时长。

```
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
```

代码3

主函数simulation（代码4）可以实现上述的算法。对象打印队列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方法得到一个1到180中间的随机数（32行）来模拟一个随机事件。模拟函数给我们提供了设置总时长和单位时间打印机打印张数。

```
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()

    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4

当我们运行模拟器时，我们不用担心每次的结果会不同，这是由于随机数的概率属性。我们更关心的是当我们改变给模拟器的参数时结果会有哪些变化趋势。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。不要忘记因为模拟器通过随机数进行测试，所以每次的结果会有所不同。

```
>>>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.
```

当经过了10次运行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的波动幅度，从最小平均17.27秒到最大平均239.61秒。你还可以看到在所有情况中只有两个是将任务全部完成的。

现在，我们将打印速度调整到每分钟10页，然后再运行十次。当有更快的打印速度时，我们期望在一小时的时限中能有更多的任务被全部完成。

```
>>>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.
```

你可以自己运行以上代码中的模拟器。

```
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
        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

我们想要探索的问题是现有的打印机是否可以在较高打印品质却较低打印速度的设置下完成打印任务。解决问题的方式是用一个模拟程序来模拟整个过程，将打印任务作为拥有可变长度和可变生成时间的随机事件。

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 小时结束时还有未打印的任务遗留在等待队列中。

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。学生们并不能因为打印而等待太长时间（6 分钟及以上），尤其是需要赶去上下一节课时。

这类对于模拟过程的分析有助于回答许多“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打印行为。例如：

- 如果注册的学生人数增加，平均每小时学生人数增加了 20 人会怎样？
- 如果周六时学生不需要着急上课会怎样？他们可以等待吗？
- 如果由于 Python 是如此简洁好用的一门语言，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决。然而需要指出的是，这些模拟只能基于建立模拟的假设来契合真实情景，如果要建立完全符合实际情况的良好模拟，需要真实的每小时打印机打印页数和每小时学生数量等数据。

自我检测

如何修改打印模拟程序反应学生数目增加时的情况？假设学生数目加倍，你需要作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长度缩短到原来的一半，修改代码来反映、模拟这种情境。最后，如何把学生数量参数化？比起修改代码我们更希望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，元素既可以在两端插入，也可以在两端删除。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 1 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，但对其中的数据项不满足严格的“后进先出”或“先进先出”顺序。 插入和删除操作的规律性需要由用户自己维持。

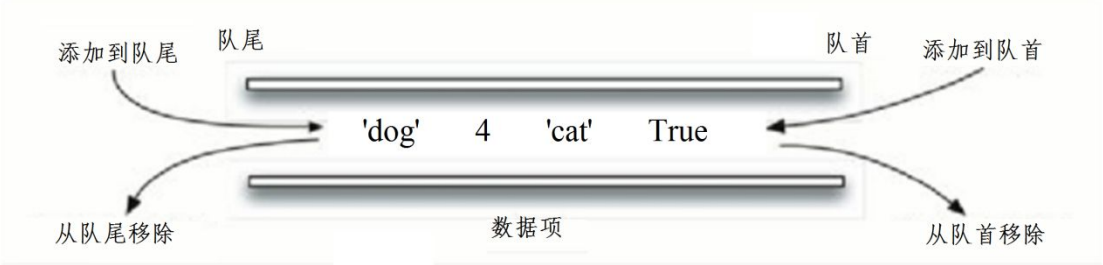

图 1　一个由 Python 数据对象构成的双端队列

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，元素可以从队首和队尾插入和删除。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、队尾的状态非常重要。

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 1 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做过的一样，我们将创造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 1

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)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。

```
1  class Deque:
2      def __init__(self):
3          self.items = []
4
5      def isEmpty(self):
6          return self.items == []
7
8      def addFront(self, item):
9          self.items.append(item)
10
11     def addRear(self, item):
12         self.items.insert(0,item)
13
14     def removeFront(self):
15         return self.items.pop()
16
17     def removeRear(self):
18         return self.items.pop(0)
19
20     def size(self):
21         return len(self.items)
22
23 d=Deque()
24 print(d.isEmpty())
25 d.addRear(4)
26 d.addRear('dog')
27 d.addFront('cat')
28 d.addFront(True)
29 print(d.size())
30 print(d.isEmpty())
31 d.addRear(8.4)
32 print(d.removeRear())
33  print(d.removeFront())
```

代码 2 Deque 运行实例

3.5.4 “回文词”判定

一个用双端队列数据结构可以轻松解决的有趣问题是经典的“回文词”问题。回文词是指正读和反读都一样的词，如：radar、toot 和 madam。我们想要建立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 33）

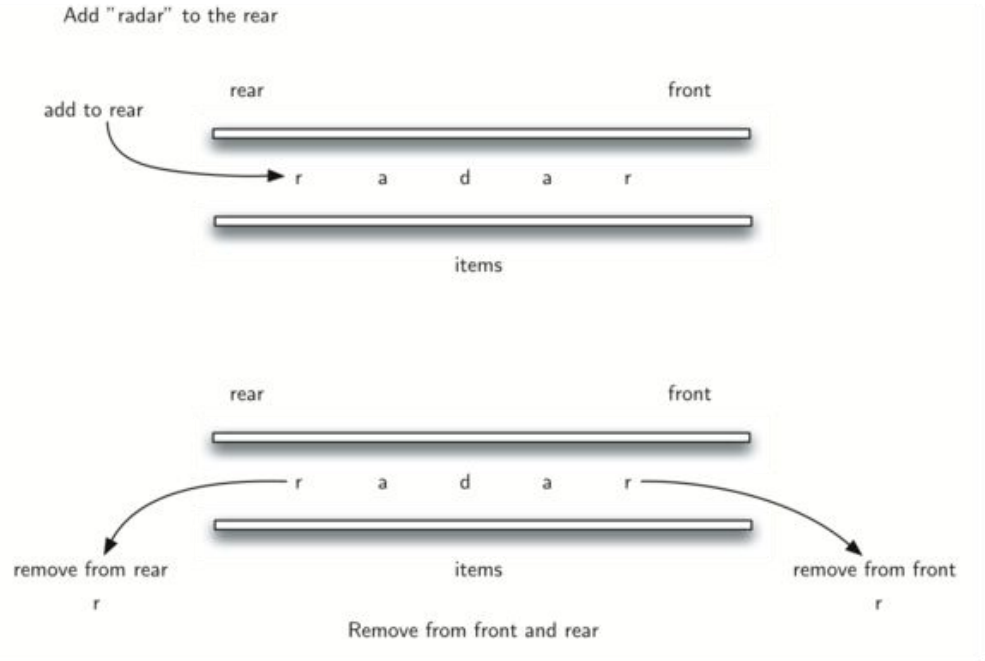

图 33 双端队列

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，我们已经使用了 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，无序列表的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 1 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 2），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 1 不受其物理位置约束的项目

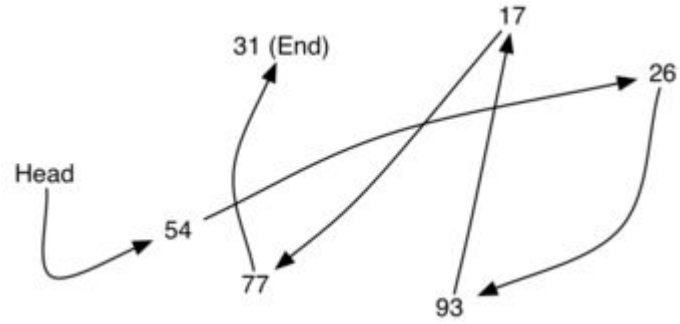

图 2 使用显式链接确定的相对位置

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）。需要指出，我们将通常以如图 4 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 1

我们以常见的方式创建了节点类。

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

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。

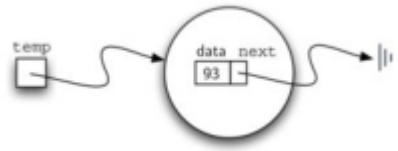

图 3 节点类对象包含本身的项目数据和对下一个节点的引用

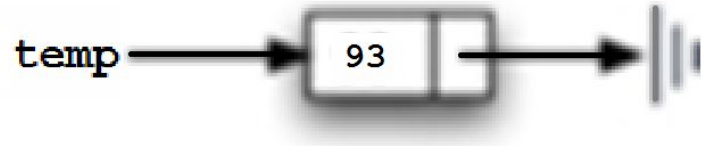

图 4 一个典型的节点类对象的实现

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。有了这个想法，UnorderedList 类必须保持一个对第一节点的引用。代码 2 显示了构造它的构造器。注意每个 Unordered List 对象将保持列表的头一个节点的引用。

```
class UnorderedList:

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

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

代码 2

在最初当我们建立一个列表时，其中没有任何元素。如图 5 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 6 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，该节点有一个链接到下一个节点（一项）的引用。需要注意的一点是，列表类本身不包含任何节点。相反，它只包含对链接结构的第一个节点的引用。

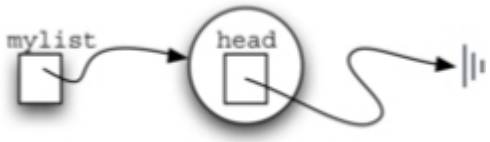

图 5：一个空列表

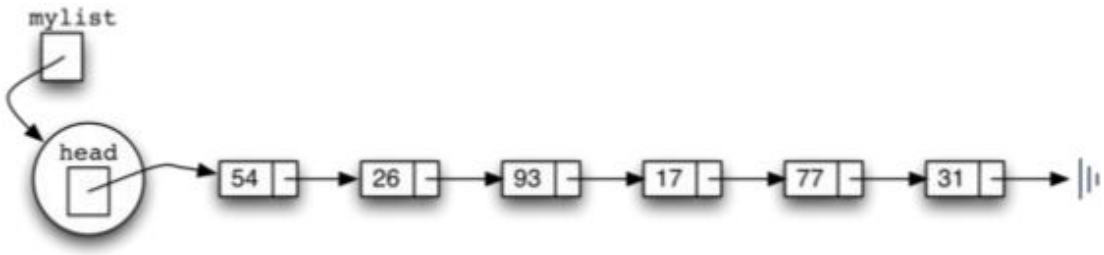

图 6：一个由整数组成的链接列表

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

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在链表的哪一部分创建新的元素。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。记住，把新的元素放置在最简单的位置是很有意义的。

记住，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 7 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。

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

代码 4

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 8 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。

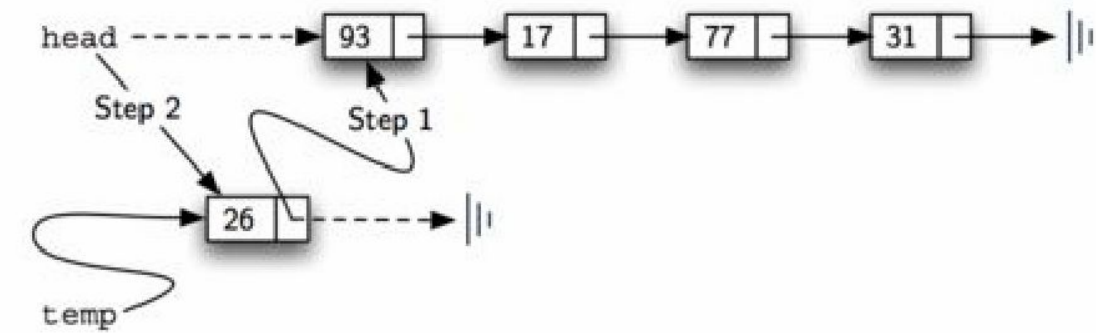

图 7：向一个两步的进程中添加一个新节点

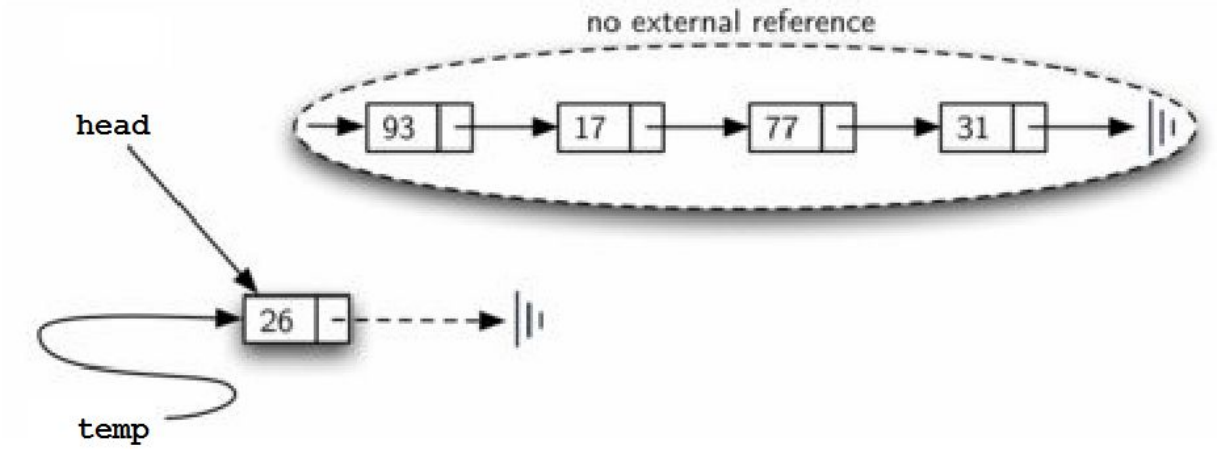

图 8：改变两个步骤顺序的后果

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“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。最终，我们在迭代结束后返回了计数值。图 9 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 5

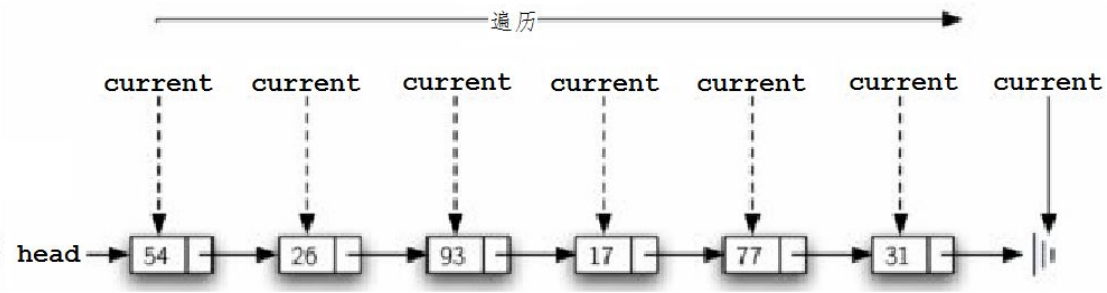

图 9：将列表从头遍历到尾。

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（第 3 行）。第 4 行中的迭代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。

```
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 6

以调用查询过程来查找“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，并在条件失效时返回 True，如上图所示。这个过程如图 10。

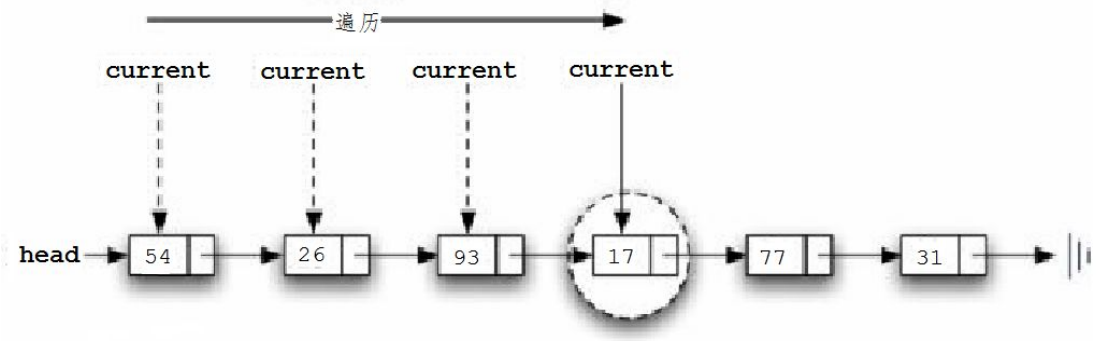

图 10：成功找到数值“17”

“移除”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，那么迭代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 11）。布尔变量 found 将会再次被用于控制这次迭代。

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。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 12 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 7

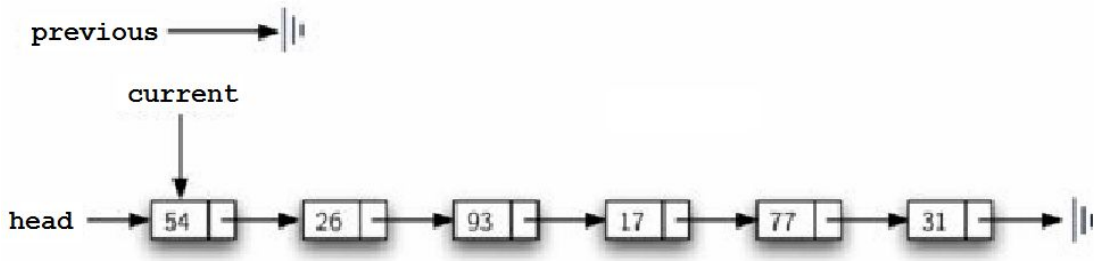

图 11：两次引用 previous 和 current 的初值

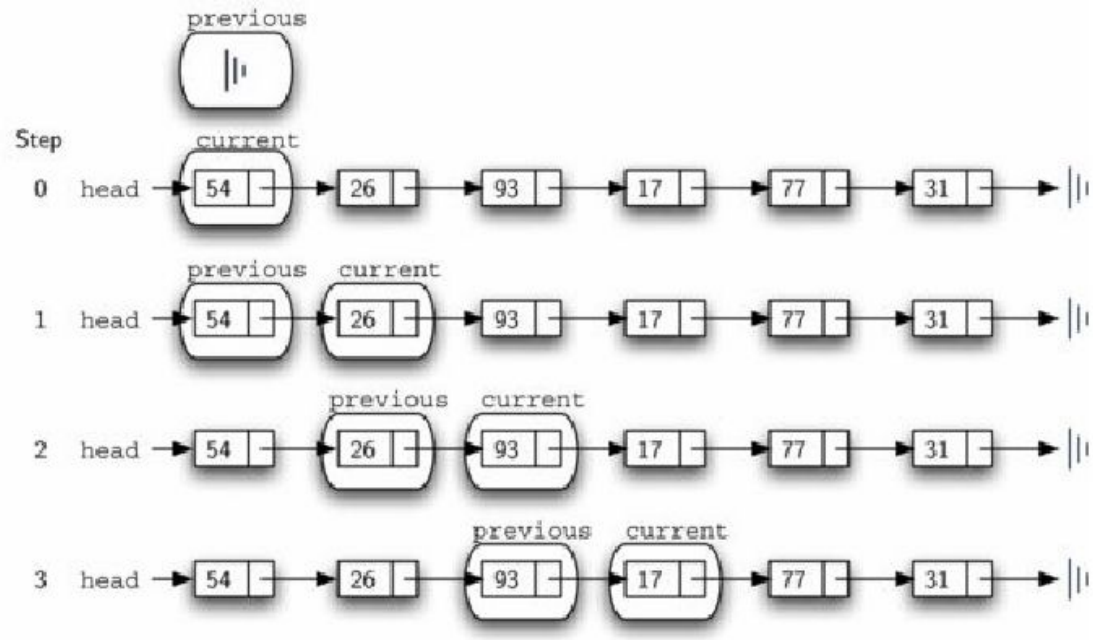

图 12：previous 和 current 沿列表移动

移除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 13 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 14）。

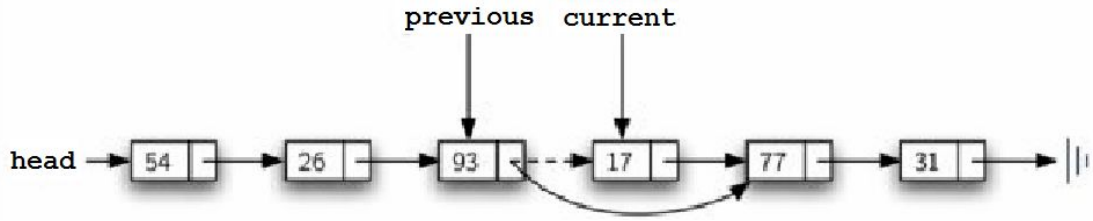

图 13：移除列表中间的一个节点

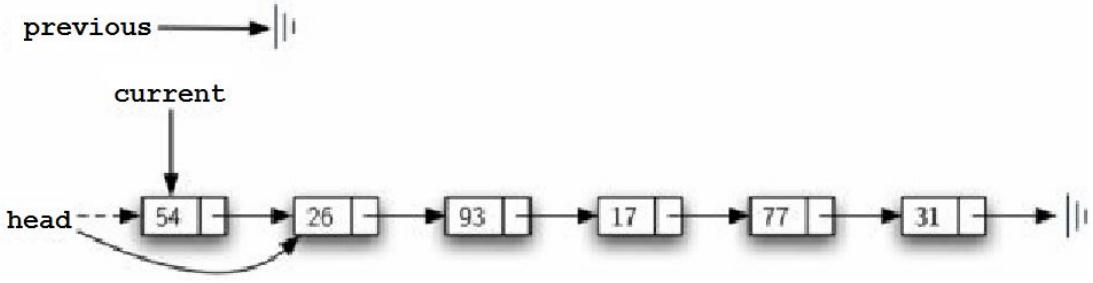

图 14：移除列表第一个节点

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，引用紧跟着 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。这个问题将由你来自行思考。

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。

自我检测

第一部分：实现无序列表的 append 方法。并给出你的程序的时间复杂度。

第二部分：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 OrderedList

现在，我们考虑一个称为有序列表类型的列表。例如，如果前面所示的整数列表是有序列表(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，它是列表中的最后一个元素。有序列表的结构是：基于元素间的某种相对关系来进行排序，再来确定元素列表中的相对位置。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但不返回其他参数。它不需要传递任何参数，返回一个空列表。
- add(item):在保证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，而函数不带返回值。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()：返回列表中的元素的数量。不需要参数，返回一个整数。
- index(item)：返回元素在列表中的位置。需要要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做参数，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。上面给出的整数有序列表(54，17，26，31，77，93)可以表示为一个如图 15 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。

Figure 15: An Ordered Linked List

图 15 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相同的方法。再一次，一个指向 None 的头指针将指向一个空的列表。（见代码 8）

Listing 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实节点的实际的值。同样，remove 方法也不需要改动,因为我们仍然只是需要找到某一个节点然后删除它。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，我们需要遍历所有的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，我们可以利用有序这一特征来尽快结束搜索。

例如,图 16 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，在以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有序列表中了。

../_images/orderedsearch.png

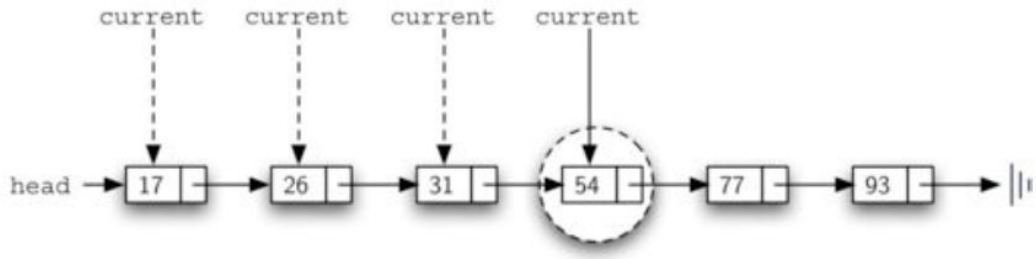

Figure 16: Searching an Ordered Linked List

图 16

代码 9 显示了完整的搜索方法。在新的情况里，很容易通过添加另一个名为 stop，初始化为 False 的布尔变量来控制（第 4 行）。当 stop 的值为 False（不停止）时，我们继续在列表中搜索（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 设为 True（9-10 行）。剩余的行为和无序表一样。

Listing 9

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

    return found
```

代码 9

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 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表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 我们就停止。

../_images/linkedlistinsert.png

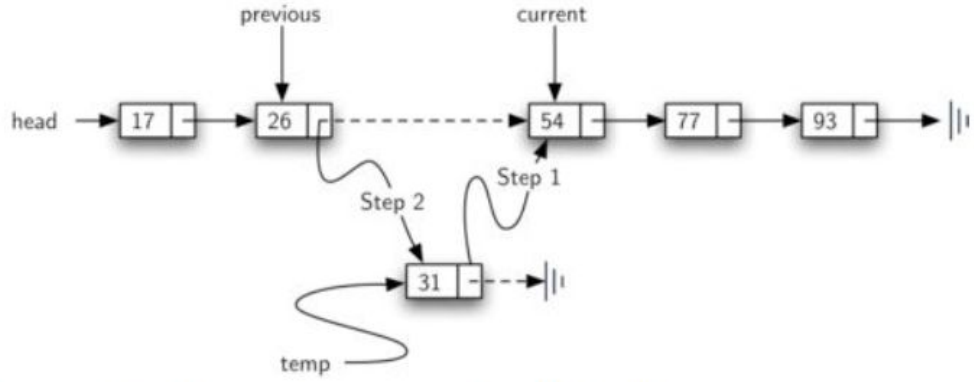

Figure 17: Adding an Item to an Ordered Linked List

图 17 向有序链表添加元素

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 再次每次都让 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，当我们找到了添加新节点的位置时，循环结束。

该方法的其余的步骤如图 17 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列表开头还是添加在列表的中间。再一次，previous==None（行 13）可以提供答案。

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

    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

代码 10

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。当使用有序表时，你应该仔细考虑顺序的问题。

3.6.5. 链表实现算法分析

链表的分析：

当考虑链表的方法的复杂度时，我们应该考虑是否需要遍历链表。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 是 O（n）。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的添加一个新的节点。但是，search、remove 和在有序列表中的 add 方法，需要进行遍历。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。

您可能还注意到,这些实现的性能不如 Python 的内置列表 list，这表明，Python 中的 list 不是这么来实现的。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）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）是具有相对位置的数据项的集合。
- 链表的实现保持逻辑顺序，不要求数据项依次存放在连续的存储空间。
- 对链表的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	

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．将下列的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，执行以相反的顺序使用 add 功能的结果是什么？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 功能如何实现。
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的中缀求值法。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 进行基准程序测试并进行比较。你从这个实验中学到了什么？
7．实现一个队列使它的 enqueue 和 dequeue 功能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
一定要讲清楚你做的任何假设，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一个“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，标记以开始标记（opening tag，<tag>）和结束标记（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：

```
<html>
   <head>
      <title>
         Example
      </title>
   </head>

   <body>
      <h1>Hello, world</h1>
   </body>
</html>
```

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。

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的头中。修改无序列表类，包含这个信息并且重写编写 length 方法。

14. 实现 remove 方法，使得当列表中没有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 slice 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）。链表的头同样有两个引用，一个指向链表的第一个节点，一个指向最后一个。用 Pythons 实现这段代码。

28. 设计并实现一个可以有平均值的队列。

4.递归 Recursion

4.1.目标

- 了解某些难解的问题具有简单的递归解决法；
- 学会如何用递归方式写程序；
- 了解和应用递归的三法则；
- 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是如何实现的。

4.2 什么是递归

递归是一种解决问题的方法，它把问题不断地分解为越来越小的子问题，直到问题的规模小到可以用非常简单直接的方式来解决为止。为了达到分解问题的效果，递归过程中一般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，而且如果不使用递归，这些问题一般都具有很大的编程难度。

4.2.1 计算数列表的和

我们先来考察一个比较简单的问题，这个问题不用递归也可以直接解决：对一个数字列表（如[1,3,5,7,9]）进行求和。一个通过迭代功能（for 循环）求和的程序如代码 1 所示。该功能利用一个变化着的“累加器”变量（theSum）对列表中的所有数进行累加式的求和，也就是从 0 开始，依次加上列表中的每一个数。

```
def listsum(numList):
    theSum = 0
    for i in numList:
        theSum = theSum + i
    return theSum
print(listsum([1,3,5,7,9]))
```

代码 1 迭代求和

现在，假设我们不能使用 while 循环和 for 循环，那么如何对列表中的所有数进行求和？如果你是一个数学家，那么你首先想到的也许是：按照定义，加法是一个只有两个参数——两个数字——的函数。为了将对数字列表求和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表达式来重新表示列表，表达为如下形式：

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

当然我们也可以从另外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（7+9），这是不需要循环或其他特殊的结构就可以进行计算的。因此，实际上我们可以通过以下一系列简化后的式子来计算最后的加和。

```
total= (1+(3+(5+(7+9))))
total= (1+(3+(5+16)))
total= (1+(3+21))
total= (1+24)
total= 25
```

那我们要怎样将这个思想转化成一段 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 2 表示出来。

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

代码 2 递归求和

在这个代码中，有一下些关键点需要注意。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 的列表求和是简单的，因为结果就是这个元素。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 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递归结束。

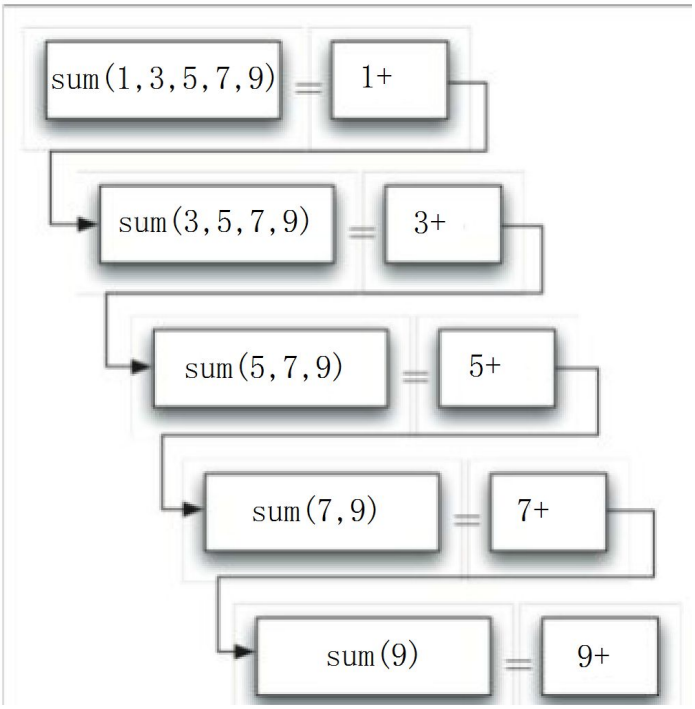

图 1：数字列表求和中的递归调用

当问题规模达到最小时，我们开始将这些小规模问题的答案连接起来直到解决最开始提出的问题。图 2 展示了列表求和在一系列递归调用之后的回溯过程，当子列表求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。

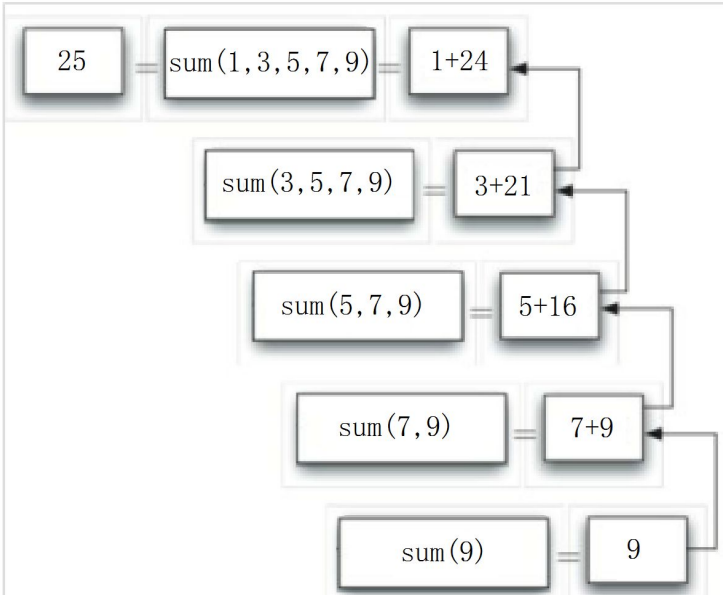

图 2 数字列表求和递归函数的返回过程

4.2.2 递归的三定律

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：

- 递归算法必须有一个基本结束条件；
- 递归算法必须能改变状态向基本结束条件演进；
- 递归算法必须递归地调用自身。

我们来更仔细地研究每一条定律，来看看它们是怎样在“数字列表求和”的算法中体现的。首先，基本结束条件是一种可以让程序结束递归的情况。基本的结束条件通常是一个规模小到足以直接解决的问题。在数列求和算法中，基本结束条件是长度为 1 的列表。

为了遵循第二条定律，我们必须改变程序的状态，为算法向基本结束条件演进做准备。程序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了我们的问题的数据以某种方式变小了。在数列求和算法中，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化的最自然的过程就是缩短列表长度。我们在代码 2 的第五行中看到的具体做法就是将列表的长度减少 1，再调用这个更短的列表的列表求和算法。

最后一条定律就是算法程序必须调用自身。这正是“递归”的定义。递归算法对于初学者来说是一个难理解的概念。作为一个初学者，你已经看到了递归功能的优越性，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。当我们谈到递归的时候就好像把我们自己置身与循环之中。我们拥有一个解决问题的方法，但是这个方法需要调用自身来解决问题！但这在逻辑上是完全不循环的，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。

在这个章节的剩余部分我们可以看到更多关于递归算法的例子。在每一个事例中我们将专注于如何根据递归三定律来设计解决问题的算法。

自我检测

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. 将整数转换为字符串形式的任意进制表示

假设你想要将一个整数转换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”。尽管有很多算法来解决这个问题，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。

下面让我们以进制基数 10、数字 769 为例看一个具体问题。假设我们有一个字符序列对应前 10 个数字，形如 convString=“0123456789”。通过在字符序列中检索很容易将比 10 小的整数转换成对应的等值字符。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，那么再将它转换为字符串是十分容易的。小于 10 的整数看起来是一个不错的基本结束条件。

确定进制基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，让我们考虑一下什么数学运算可能会把一个数字分解。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。整数除法取余的运算给我们一个明确的的方向。让我们看看如果我们把试图转换的数字除以进制基数会发生什么。

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

首先，余数是一个小于我们的进制基数的数字，通过查找它可以立即转换成一个字符。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“找到小于进制基数的单个数字”的基本结束条件演进。现在我们的工作是再将 76 转换为它的字符串形式。我们将再一次使用整数的除法加上取余得运算分别得到 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 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。

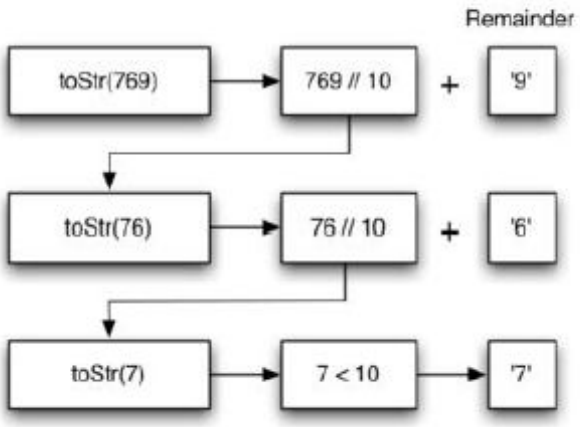

图 3：进制基数为 10 转换整数到字符串

代码 1 显示了进制基数在 2 到 16 之间时实现上述算法的 Python 代码。

```
def toStr(n,base):
   convertString = "0123456789ABCDEF"
   if n < base:
      return convertString[n]
   else:
      return toStr(n//base,base) + convertString[n%base]
print(toStr(1453,16))
```

代码 1 递归转换整数到字符串

请注意，在第 3 行我们检查进制基数 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的规模这两点。

让我们再次追溯算法,这次我们将转换整数 10 为字符串形式的 2 进制表示(“1010”)。

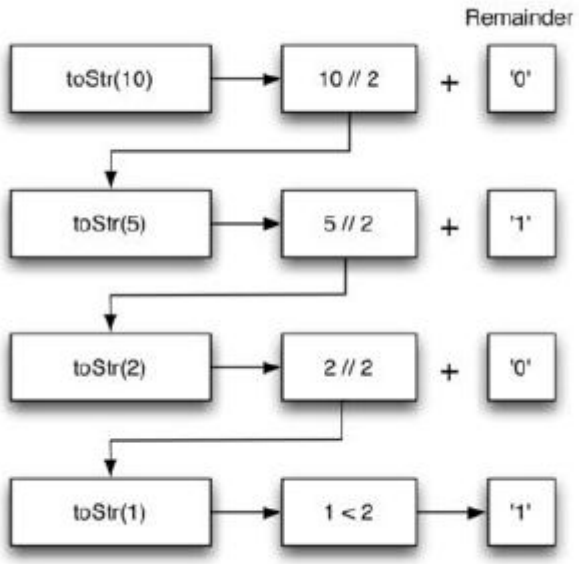

图 4：转换整数 10 为字符串形式的 2 进制表示

图 4 显示我们得到了期待的结果，但是看起来数字的次序错了。该算法运行正确，因为我们在第 6 行先使用递归调用，然后添加了余数的字符串表示。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。

自我检测

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

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，类似于递归被执行的先后顺序，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：

```
from pythonds.basic.stack import Stack
rStack = Stack()
def toStr(n,base):
    convertString = "0123456789ABCDEF"
    while n > 0:
        if n < base:
            rStack.push(convertString[n])
        else:
            rStack.push(convertString[n % base])
        n = n // base
    res = ""
    while not rStack.isEmpty():
        res = res + str(rStack.pop())
    return res

print(toStr(1453,16))
```

代码 把一个整数转入一个栈再变成字符串

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，这里的栈就如图 4-5 所示。现在我们只需要把栈中的元素推出栈，

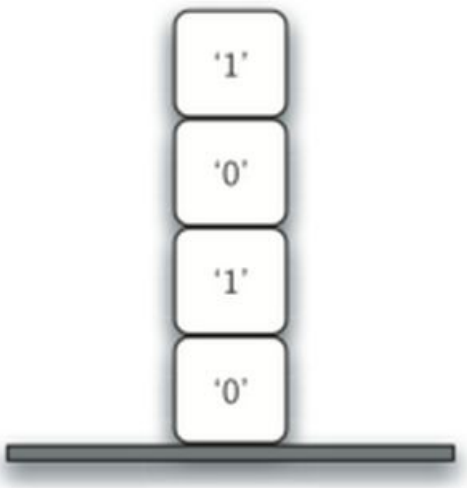

Figure 4-5 在进制转换时栈中所存字符

然后再把他们连在一起输出就得到了最后的结果：“1010”

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：

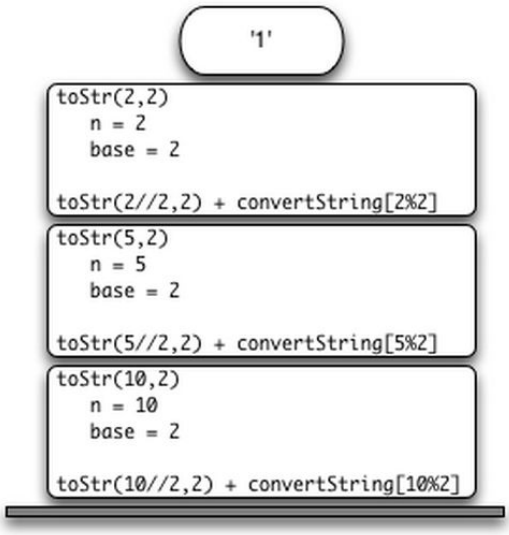

Figure 4-6 执行 toStr(10,2)的调用栈

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很容易就解决的问题，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，我们会谈一些用递归画画的有趣的例子，在你看着这些图案成形的过程中，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。

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向上或向下，向下，也可以把尾巴抬起来和放下去。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（画笔）的宽度，也可以改变制图时所使用的颜色。

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，以下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，我们也就同样创建了一个能让它在里面作图的窗口。接下来我们就来定义 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（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。

```
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()
```

代码 用海龟画一个递归螺线

这就是海龟作图模块中你所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。下一个程序中我们来画分形树。分形（fractal）来源于数学中的一支，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。正是自然界中广泛存在的这些分形的现象，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颗分形树，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整颗树的形状和特征。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，第五行是在命令海龟右转 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，而海龟必须向左转 40 度，理由是它必须抵消先前向右转的 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，每次我们对树进行递归调用的时候，branchLen 会被减去一些量，这是为了确保递归树的分叉会越来越小。还需要注意开头第二行的 if 语句，它是整个递归的基本结束条件，当树枝越来越短时，递归就会被终止。

```
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)
```

下一段代码是这个分形树例子的完整代码，在你运行这段代码以前，请你设想一下这可树是如何成形的；仔细研究那些递归调用，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？

```
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 用递归法画一棵树

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 1 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 2 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，而是和之前一样，在到达最短的那根树枝之前，先画左半分支的整个右半部分，然后再返回话左半部分。

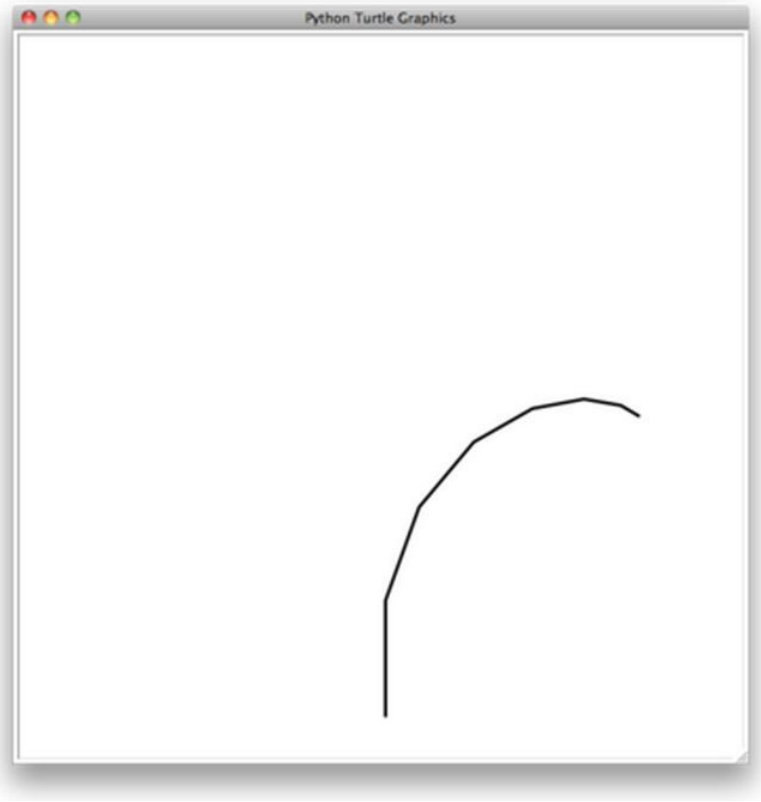

图 1 分形树的起始

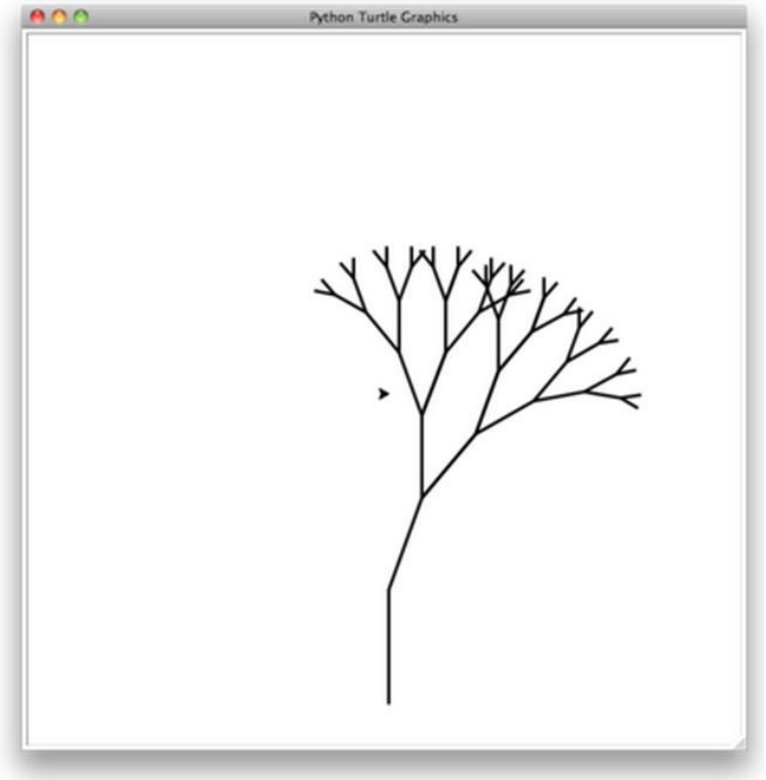

图 2 分形树的第一部分——右半部分

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写的那样对称。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。

自我检测

用下面一条或者所有的选项来修改你的递归树程序：

- 修改树枝的粗细，实现随着树枝缩短，也相应变细
- 树枝的颜色可以变化，当树枝非常短的时候，使之看起来像树叶的颜色
- 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
- 修改递归调用时树枝的长短，每次可以减去一个随机数，而不是一个固定的值

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，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：从一个单个的大三角形开始，取它的各边中点作三条中位线，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想要试一试自己来画谢尔宾斯基三角形了吧，那

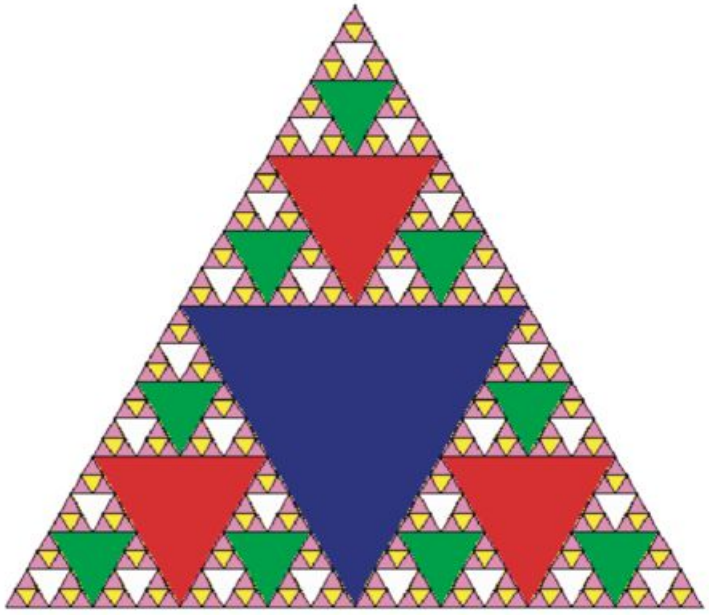

图 3 谢尔宾斯基三角形

就用上面的方式画一个吧！

既然我们可以无限的运行这个算法，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，相似性维数就减一，直到让他到 0 为止，这时候便停止递归调用。画谢尔宾斯基三角形的代码如下：

```
import turtle

def drawTriangle(points,color,myTurtle):
    myTurtle.fillcolor(color)
    myTurtle.up()
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.down()
    myTurtle.begin_fill()
    myTurtle.goto(points[1][0],points[1][1])
    myTurtle.goto(points[2][0],points[2][1])
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.end_fill()
```

```
def getMid(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

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()
```

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。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，然后返回先分割上方角落的三角形，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，用图解的方式去理解递归算法的函数调用是很有帮助的。图 4 就告诉了我们，sierpinski 函数的递归调用总是先做左下角的三角形，图中被轮廓为黑

色线条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，朝向图 4 的底部走得越远，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，他就会继续执行上边的三角形，然后是右边的，以此类推……

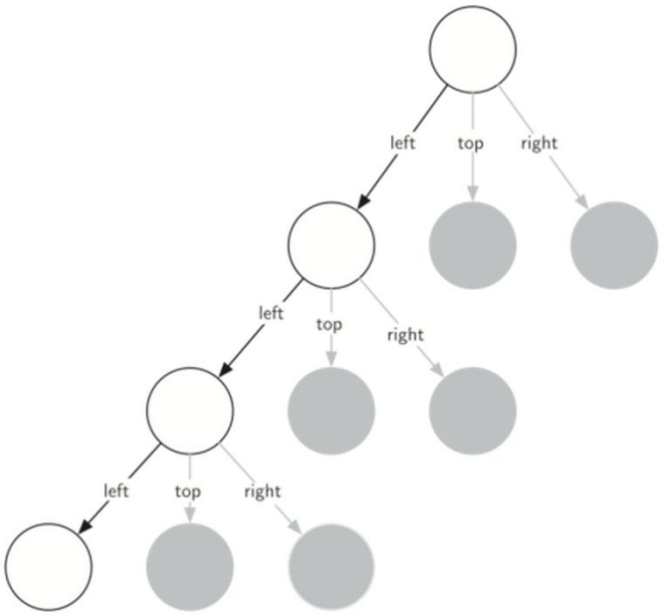

图 4 Sierpinski 三角形的建立过程

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。