3 基本数据结构

3.1 目标

- 理解抽象数据结构：栈stack、队列queue、双端队列deque、列表list。
- 利用Python的列表实现抽象数据类型(ADTs) stack、queue、deque。
- 了解各种基本线性数据结构的实现的性能。
- 理解前缀，中缀和后缀表达式。
- 利用stack对后缀表达式进行求值。
- 利用stack将表达式从中缀转换为后缀。
- 利用queue进行基本的时间模拟。
- 能够识别问题的特性，并依次选用stack、queue或者deque中的合适的数据结构。
- 能够实现使用node和reference模式将抽象数据类型实现为链表。
- 能够对比链表与Python实现的列表的性能。

3.2 何为线性结构

在开始数据结构的学习前，首先考虑四个简单但很强大的概念：stack、queue、deque、list，它们都是基于添加或者删除方式来定序的数据容器。一旦添加某个项目，它的位置便定于前一个加入的元素和后来将要加入的元素之间。这种数据容器常常被称为线性数据结构。

线性结构可以认为是有两个端。有时被称为“左”和“右”或者“前”和“后”再或者“顶”和“底”。名称并不重要。线性结构与其它数据结构的区别在于元素添加或者删除的方式，尤其是添加和删除的位置。比如说，有些数据结构只允许新元素被添加在某一端；有的结构允许从两端添加。

这种区别形成了计算机科学中一些最有用的数据结构。它们会出现在很多算法里面并用来解决一系列重要问题。

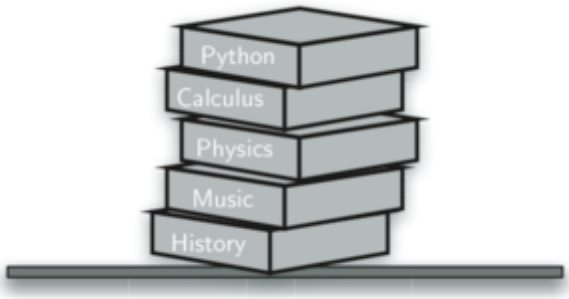
何为栈Stack

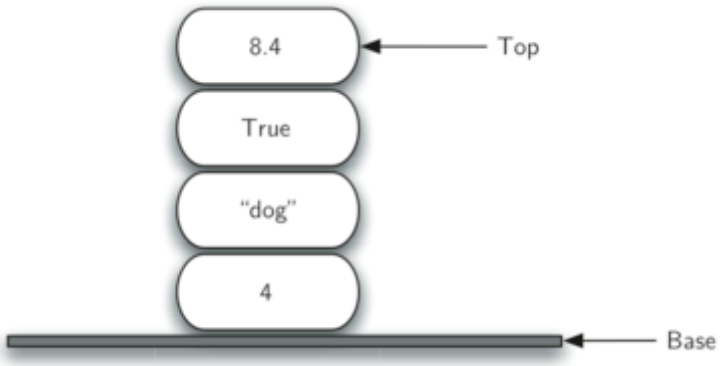
栈（有时称为叠加栈），是一种有序数据容器，其添加或者删除仅发生在同一端。这一端一般称之为“顶”，而另一端就被成为“底”。

栈的底有着重要意义，因为栈中越接近底的元素，说明其在战中储存得最久。最新添加的元素也是最先被移除的元素。这种定序法有时也被称为LIFO(last-in first-out，后进先出法），即根据在容器中停留的顺序来决定顺序。越新的元素越接近顶部，而越旧的项越接近底部。

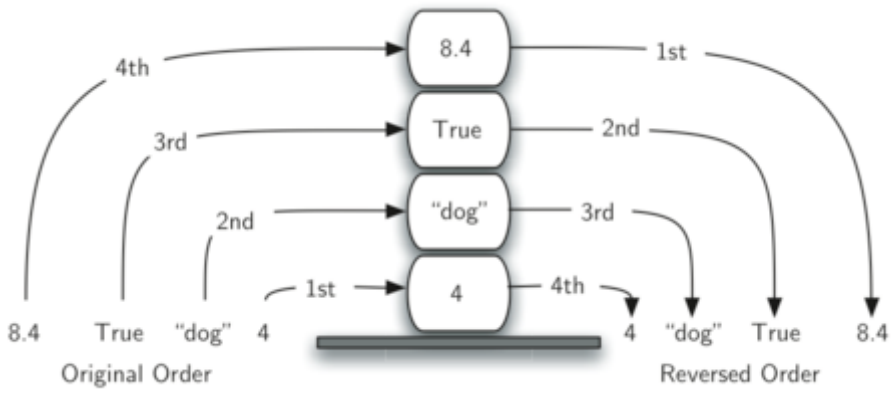
日常生活中有很多栈的例子。几乎每个咖啡馆都有一堆餐盘，取出了顶部的一个盘子后，餐盘堆中将暴露一个新的餐盘用于提供给接下来的顾客。想象书桌上的一堆书（图1），只有顶部的那本书的封面

是可见的。为了获取堆中的其它元素，必须将顶部的移除。图2演示了另一个栈，它包含一些Python的基本数据结构。





简单观察下栈中元素的添加和删除，便可发现栈最有用的一个思想之一。假设起始桌上有什么都没有。现在开始把书一本一本地放在顶部。这便是在构建一个栈。当移除项的时候，其顺序恰好与放置顺序相反。栈是非常重要的，因为它们可以用来反转元素的顺序。插入顺序和取出顺序相反。图3演示了Python数据对象栈的生成和元素的取出。注意对象的顺序。



../_images/simplereversal.png

考虑下这种逆转特性，也许在使用电脑时是经常出现的。比如说，网页浏览器都有返回按钮。当从一个网页导航至另一个网页时，这些网页都被存放在栈中（实际上是将URL放入栈中）。当前浏览的页面被放在栈顶，而最先浏览过的那个网页被放于底部。当点击返回按钮时，便会以相反的顺序浏览网页。

3.4 栈的抽象数据类型

栈的抽象数据类型通过下列结构和操作定义。如前所述，栈是结构化的，是一种有序元素的集合，它从被称为“顶”的一端进行元素的添加和删除。栈以后进先出法进行定序。栈的操作如下：

- Stack()创建一个新的空栈。它不需要参数，返回一个空栈。
- push(item)将新的元素加入栈的顶部。它需要元素作为参数，无返回结果。
- pop()将顶部的元素取出。它不需要参数，返回结果为被取出的元素。栈本身发生了变化。
- peek()返回栈顶的元素，但并不将其取出。不需要参数，并且不对栈进行修改。
- isEmpty()检测栈是否为空。它不需要参数，并且返回一个布尔值。
- size()返回栈中元素的数量。它不需要参数，并且返回一个整数。

比如说，如果**s**是已存在的空栈，表1展示了一些栈操作的结果。以列表的形式，将栈顶的元素展示在右侧。

Stack Operation	Stack Contents	Return Value
`s.isEmpty()`	`[]`	`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.isEmpty()`	`[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`

3.5 在Python中实现栈

既然已经明确将栈定义为一种抽象数据类型，那么现在就来使用Python实现栈。读者应该还记得的，给某个抽象数据类型以物理实现时，这种实现就是数据结构。

如第一章所述，在面向对象语言Python中，实现栈这种抽象数据类型的选择便是新建类。栈的操作以方法的形式实现。此外，为了实现作为元素容器的栈，充分利用Python本身提供的基本容器的强大和简单是非常有意义的。这里使用列表。

Python中的列表类提供了有序容器的机制和一系列方法。比如说，对于列表[2,5,3,6,7,4]，仅需要决定哪一端是作为栈的顶部。一旦确定，便可以使用列表的方法比如说append和pop来实现操作运算。

下面的栈的实现（可执行代码1）假设列表的一端会储存栈顶部的元素。当栈的规模变大时（即使用了push操作），新的元素会被添加在列表的顶部。pop操作也在同一侧进行。

可执行代码1 使用Python中的列表实现栈

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

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

     def push(self, item):
         self.items.append(item)

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

     def peek(self):
         return self.items[len(self.items)-1]

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

执行上述代码除了对类的定义外，什么也不会发生。必须要新建一个**Stack**对象并且使用它。可执行代码2按照表1中的操作演示了**Stack**类的使用。注意，**Stack**类的定义是从本书提供的pythonds模块导入的。

可执行代码2：stack_ex_1

```
from pythonds.basic.stack import Stack

s=Stack()

print(s.isEmpty())
s.push(4)
```

```
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

很重要的一点是，也可以选择列表的开头作为栈的顶部。这样一来，之前的pop和append方法便失效了，必须要索引到位置0（列表的第一个元素），然后显式地使用pop和insert。实现如下：

```
| 1 |class Stack:
| 2 |      def __init__(self):
| 3 |          self.items = []
| 4 |
| 5 |      def isEmpty(self):
| 6 |          return self.items == []
| 7 |
| 8 |      def push(self, item):
| 9 |          self.items.insert(0,item)
| 10 |
| 11 |       def pop(self):
| 12 |           return self.items.pop(0)
| 13 |
| 14 |       def peek(self):
| 15 |           return self.items[0]
| 16 |
| 17 |       def size(self):
| 18 |           return len(self.items)
| 19 |
| 20 | s = Stack()
| 21 | s.push('hello')
| 22 | s.push('true')
| 23 | print(s.pop())
```

修改抽象数据类型的物理实现同时保持其逻辑特征，这是一种实用抽象的典例。然而，虽然栈在另一种实现下也可以运行，但是考虑下这两种实现的性能，其实是天差地别的。还记得append和pop()操作都是O(1)。也就是说第一种实现无论栈中有多少元素，这两种操作的耗时都是一样的。而第二种实现由

于insert(0)和pop(0)的存在，所以其复杂度是O(n)。显然，虽然这两种实现在逻辑上是等效的，但是性能差距是非常大的。

3.6 配平括号表达式

现在使用栈来解决一些实际的计算机科学问题。毫无疑问读者曾写过类似的表达式：
(5+6)*(7+8)/(4+3)
其中括号用于调整操作的顺序。也许读者有使用Lisp等编程语言的经验，并且有类似下面的构建：

```
(defun square(n)
     (* n n))
```

这定义了名为**square**的函数，返回其参数**n**的平方。Lisp被诟病使用了太多太多的括号。

在这些例子中，括号必须以成对的方式实现。**成对括号**意味着，每个开括号必须有一个对应的闭括号与之成对，并且每组括号以正确的方式嵌套。考虑以下括号：

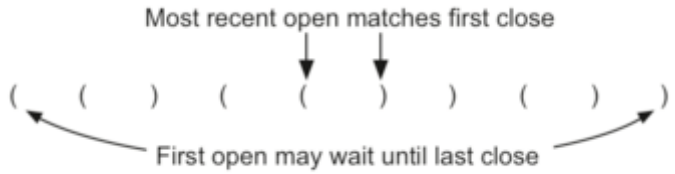
```
(()()()())

((((()))

((()((()))))
```

下面是一些并不成对的：

```
(((((((())

()))

(()()(()
```

识别成对或不成对括号，对于正确识别许多编程语言结构是很重要的。

问题便是，给出一种算法从左到右依次读取括号字符串，并确定它是否是平衡的。为了解决该问题需要先来仔细观察一下。从左至右处理符号时，最近的开括号必须与下一个闭括号搭配（如图4）并且，第一个开括号有可能必须要等到最后一个闭括号才能与其形成搭配。闭括号与开括号以它们出现的顺序的逆序进行匹配，它们由外向内匹配。这便给予提示，该问题可以用栈解决。



../_images/simpleparcheck.png

确定了栈是保存括号的合理数据结构后，该算法的语句便很直接了。如果是开括号，将其放在栈顶，表示它需要之后出现一个闭括号与之匹配。如果是闭括号，取出栈顶的首项。只要每一个闭括号都可以在栈顶取出开括号来与之对应，括号表达式便是平衡的。如果栈内没有开括号来与闭括号与之对应了，该表达式就是部平衡的。在最后，所有的括号都被处理完毕，栈必须是空的，否则也是部平衡的。Python实现代码如下可执行代码1所示：

```
from pythonds.basic.stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False

print(parChecker('((()))'))
print(parChecker('(()'))
```

该函数**parChecker**，以**Stack**类是可用的为基本条件，返回一个布尔值标识该括号字符串是否平衡。注意到布尔变量**balanced**被初始化为True，因为没有理由在开始给以这样的设定。

3.7 配平符号 （一般情况）

前文描述的配平括号问题是许多编程语言中一种更一般情况的特例之一。配平和嵌套不同的开/闭符号是经常出现的一个问题。比如说，在Python中，方括号[和]被用于列表；花括号{和}被用于字典；圆括号(和)用于元组和运算表达式。在保证每组开/闭符号都配平的前提下，可以将它们组合起来。如下的符号字符串：

```
{ { ( [ ] [ ] ) } ( ) }

[ [ { { ( ( ) ) } } ] ]

[ ] [ ] [ ] ( ) { }
```

必须保证不仅每个开符号有对应的闭符号，还必须保证不同类型的符号正确地搭配了。
比如说下面这些就是没有配平的：

```
( [ ) ]

( ( ( ) ] ) )

[ { ( ) ]
```

前一节给出的圆括号配对检测器可以很容易地拓展，来处理这些新的符号类型。回忆下，每一个开符号都被直接放入栈中等待闭符号与之配对。当闭符号出现时，唯一的区别是在现在这种情况下必须要保证栈顶的符号类型与之匹配。如果这两个类型并不匹配，那该表达式是没有配平的。并且，如果整个表达式都被处理完了，栈此时为空，则该表达式是配平的。

Python实现如可执行代码1所示。唯一的区别是，该函数调用了一个协助函数**matches**，用来完成符号配对。每一个从栈顶移除的符号必须与当前的闭符号进行配对。如果有不配对的情况出现，布尔变量便为设置为False。

可执行代码1：求解一般配平符号问题

```
from pythonds.basic.stack import Stack

def parChecker(symbolString):
```

```
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                       balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False

def matches(open,close):
    opens = "([{"
    closers = ")]}"
    return opens.index(open) == closers.index(close)


print(parChecker('{{([][])}()}'))
print(parChecker('[{()]'))
```

这两个例子演示了栈在处理计算机科学中的语言构建问题时，是非常重要的数据结构。几乎所有的表达式都有一些必须配平的嵌套符号。计算机科学中，栈还有其它非常重要的用途。接下来几节将继续说明。

3.8 十进制转二进制

在计算科学的学习中，读者应当已经或多或少接触到了二进制数字。二进制表示在计算机科学中是很重要的，因为所有计算机中存储的值都是以二进制数字字符串的形式存在的，即由1和0组成的字符串。如果没有在十进制和二进制进行转换，与计算机的交互是相当麻烦的。

整数是很常用的数据元素，计算机程序和计算中都会使用到。大家都是从数学课上接触到整数，当然也都是用十进制表示它们。实际上十进制数233_10和其对应二进制等效写法11101001_2分别被解释为：

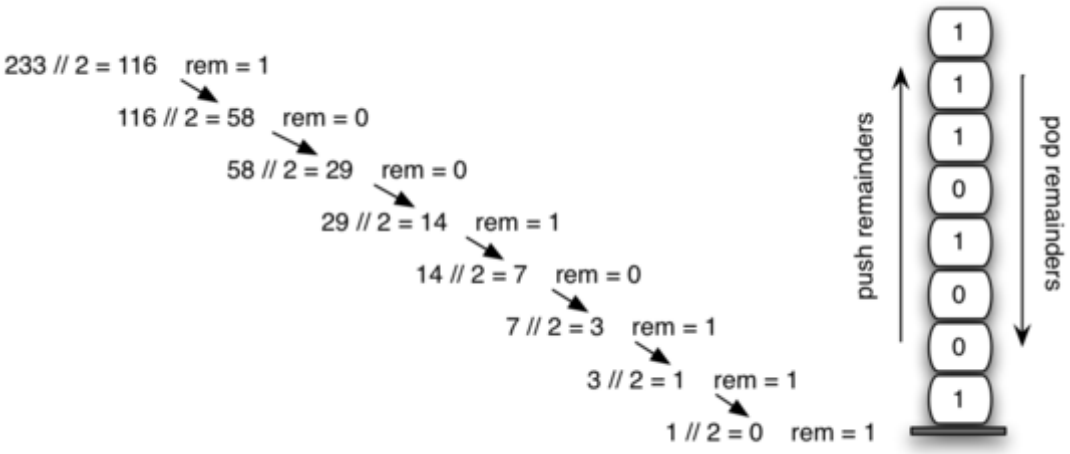
$$2 * 10^2 + 3 * 10^1 + 3 * 10^0$$

以及

$$1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

但是如何轻松地将整数转化为十进制数字呢？答案是利用一种被称为“以2相除”的办法，它使用栈来跟踪二进制结果的各个数字。

以2相除的算法首先假设整数是大于0的。接下来，使用一个简单的迭代将该整数除以2，并且跟踪余数。第一次被2相除，确定该值为整数或者奇数。偶数的话，则余数为0，那么在第一个位置，其数位为0。奇数余数为1，并且在第一个位置数位为1。可以将构将二进制数字视为数位序列。计算得到的第一个余数正是该序列的最后一个数位。如图5所示，可以发现逆序性使得栈似乎是解决该问题的合理数据结构。



../_images/dectobin.png

可执行代码1中的Python源码实现了算法以2相除。函数**divideBy2**接受1个十进制数字作为参数，重复地以2相除。代码中使用了内置模块的操作符，%，用来获得余数。在11-13行，当最终被除数变为0时，便构建了一个二进制字符串。11行构建了一个空字符串。栈中的二进制数位被依次取出，然后append到字符串的右端。然后返回该二进制字符串。

可执行代码1：十进制转二进制

```
from pythonds.basic.stack import Stack

def divideBy2(decNumber):
    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
```

```
        decNumber = decNumber // 2

    binString = ""
    while not remstack.isEmpty():
        binString = binString + str(remstack.pop())

    return binString

print(divideBy2(42))
```

二进制转换的算法可以轻松拓展使其可用于任何进制的转换。在计算机科学中，使用不同的编码是很常见的。最常见的是二进制，八进制以及十六进制。

整数233对应的八进制和十六进制分别为：351_8和$E9_16$，并且分别可以表示为：

$$3 * 8^2 + 5 * 8^1 + 1 * 8^0$$

和

$$14 * 16^1 + 9 * 16^0$$

函数devideBy2可以调整为不仅接受十进制，也接受一个转换的进制。那么“以2相除”可以直接用更一般的“以进制相除”。新函数**baseConverter**，如可执行代码2中所示，接受一个十进制数以及任何介于2-16的进制作为参数。余数依然是被放入栈中，直到被转换的值变为0。同样的从左到右字符串构建法稍微调整下便可使用了。二进制到十进制之间需要最大10个数位，即典型的0，1，2，3，4，5，6，7，8，9效果很不错。但是超过十的时候，问题就来了。这时便不可以直接使用余数了，因为它们本身就是用十进制二位数字标识的。作为取代，构建一组数位用于表示超过9的余数。

可执行代码2：十进制向任意进制转化

```
from pythonds.basic.stack import Stack

def baseConverter(decNumber,base):
    digits = "0123456789ABCDEF"

    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base
```

```
    newString = ""
    while not remstack.isEmpty():
        newString = newString + digits[remstack.pop()]

    return newString

print(baseConverter(25,2))
print(baseConverter(25,16))
```

该问题的一种解决方案就是用字母来扩充数位组。

3.9 中缀、前缀和后缀表达式

在写出算术表达式如B*C*时，*表达式的形式提供了信息使得其可以被正确地解释。在此例中，变量B被变量C乘，因为它们之间有一个操作符*。这种写法被称为**中缀**，因为操作符位于两个操作对象之**中**。

考虑另一个中缀的例子，A+B*C*。*操作符+和*依然在操作对象之间，但有个问题，操作符对哪个操作对象起作用。+是在A和B上起作用，还是说*对B和C起作用。这个表达式看起来有点模糊不清。

实际上，读者都阅读并书写了大量这些表达式，并不会受到任何困扰。之所以这样是因为读者是了解操作符+和*的。每个操作符都有**优先**级。高优先级的操作符在低优先级的操作符之前先完成运算。唯一可以改变这种优先顺序的是括号。乘法的优先级是高于加和减的。对于相同等级的操作符，那么就使用从左至右的顺序或者说结合律。

对于读者来说这太明显不过了，但是对计算机来说，它必须精准地知道执行操作的顺序。

以下例子是等效的。分别以中缀、前缀、后缀表达式写出。

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +

Infix Expression	Prefix Expression	Postfix Expression
A + B + C + D	+ + + A B C D	A B + C + D +

3.9.1 中缀表达式转换为前后缀表达式

本书使用ad hoc方法实现中缀表达式与其等效前后缀表达式之间的转换。可以推测，存在算法实现将无论多复杂的表达式正确转化。

首先学习的一个技术是使用后文将深入讨论的全括号表达式记法。A+B*C*可以写作*(A+(B*C))来显式地指出乘法比加法优先级更高。然而进一步观察后，可以发现每对括号也标记了一对操作对象的开头和结尾，并且对应的操作符位于中间。

观察子表达式（B*C*）*的右侧括号。如果能够将乘法符号移动到那个位置，并且将左侧对应的括号去掉，获得BC*，实际上便完成了子表达式转化为后缀表达式。若再将加号操作符移到右侧括号的位置，并再次去掉对应的左侧括号，便将整个表达式都实现了向后缀表达式的转换。（如图6所示）

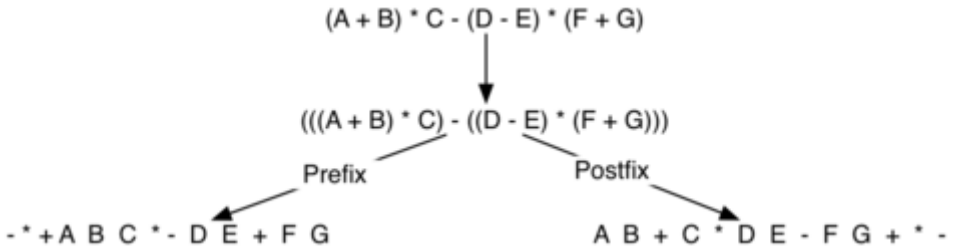
../_images/moveright.png

如果进行相同的操作，但是取而代之地，将操作符移往左边括号，那么便得到了前缀记法（如图7所示）。括号对的位置实际上给出了闭合操作符的最终位置。

../_images/moveleft.png

因此为了实现表达式的转换，无论多么复杂，无论是转为前缀或者后缀记法，都要按照操作的顺序来实现全括号化。操作符的左移或右移取决于目标是前缀或者后缀记法。

以下是一个更复杂的表达式：（A + B）*（C）-（D - E）*(F + G)。图8演示了其转为前缀或者后缀记法。



../_images/complexmove.png

3.9.2 一般中缀转后缀

为建立中缀表达式转为后缀表达式的算法，需要先仔细研究一下转化过程。

再次考虑表达式A+B*C*。*如前文所述，ABC+*是等效的后缀记法。读者应当已经注意到，操作对象A，B，C 的相对位置没有发生改变，仅有操作符发生了改变。再看看中缀表达式中的操作符。从左到右依次出现的第一个操作符是+。然而，在其后缀表达式中，+被放在了末尾，因为第二个操作符*的优先级比它高。在最终的后缀表达式中，原表达式中的操作符的顺序被反了过来。

在处理表达式的时候，操作符必须储存在某个地方因为它们对应的右侧操作对象还没有出现。并且，这些被储存起来的操作符也许需要逆序，因为它们的优先级不同。本例中的加法和乘法便是这样。因为加法操作符在乘法操作符之前出现了而且优先级更低，因此它要在乘法操作符之后再出现。因为这个顺序的逆，便可以使用栈来保存操作符，直到它们被使用。

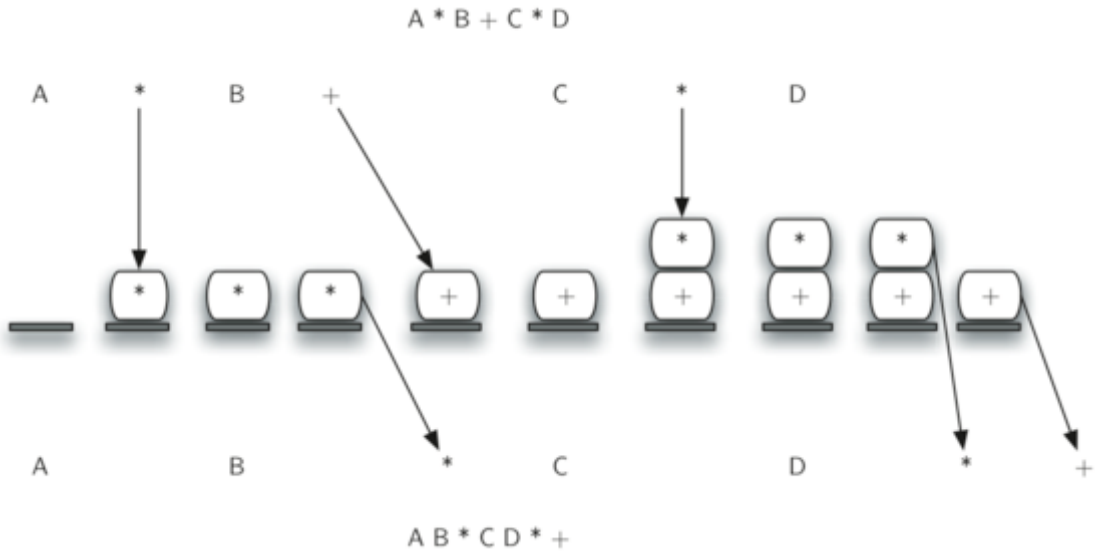
那（A + B）* C 呢？A B + C *是其等效前缀记法。同样地，从左至右处理中缀表达式，首先遇到的是+。在这种情况下，+ 已经放在了结果表达式中，因为有括号的存在，它的优先级高于乘法了。现在开始想想转换算法是如何运行的。当看到左侧的括号时，将其储存起来，因为即将有一个具有高优先级的操作符出现。该操作符必须等到对应的右侧括号出现时才能确定其位置（全括号化）。当右侧括号消失的时候，操作符便可以从栈里面取出。

当从左到右遍历该中缀表达式时，使用栈来储存操作符。这与第一个例子相反。栈的顶部始终是最新加入的操作符。当读取到一个新的操作符时，必须考虑其与栈中已有的操作符的优先级孰高孰低。

假设中缀表达式是由空格隔开的符号字符串。操作符为*, /, +, -, 并伴随有左括号和右括号。操作对象记号是单字母识别符 A, B, C, 等等。下面的步骤会按后缀记法生成符号的字符串。

1. 创建一个空栈称为opstack用来保存操作符。创建一个空列表用于保存输出。
2. 利用string的方法split将输入的中缀字符串转化为list
3. 从左到右遍历整个符号列表：
    - 如果符号是操作对象，将其append至输出列表的末端。
    - 如果符号是左括号，将其放入opstack。
    - 如果符号是右括号，对opstack进行pop操作，直到对应的左括号被去除。将每个操作符放入输出列表的末端。
    - 如果符号是*, /, +, - 中的一个操作符，将其放入opstack。然而，先要取出在opstack栈顶中比该操作符优先等级更高或者一致的所有操作符，然后将它们放入输出列表的末端。
4. 当输入表达式被处理完毕后，检查opstack，若栈中仍然有操作符，将他们依次取出并放入输出列表的末端。

图9 演示了该转换算法在表达式 A * B + C * D的应用。注意，第一个*操作符在+出现后便从栈顶移除了。然后，当第二个*出现时，+依然停留在栈中，因为乘法的优先级高于加法。在中缀表达式的末端，栈被pop两次，将两个操作符全部取出，并且把+作为后缀表达式的最后一个操作符。



../_images/intopost.png

为了利用Python实现该算法，利用了一个名为prec的字典来存储各操作符的优先级，将各操作符映射到一个整数值，该值表征了其优先级，此处就直接使用的3，2，1。左括号被设置为q全局最低值，这样可以保证任意操作符的优先级都高于他，然后被放置于栈顶。15行定义了操作对象应是大写字母或者数字。完整的转换函数如可执行代码1所示：

可执行代码1：中缀表达式转为后缀表达式（中-后）

```
from pythonds.basic.stack import Stack

def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in
"0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
```

```
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and (prec[opStack.peek()] >=
prec[token]):
                postfixList.append(opStack.pop())
            opStack.push(token)

    while not opStack.isEmpty():
        postfixList.append(opStack.pop())
    return " ".join(postfixList)

print(infixToPostfix("A * B + C * D"))
print(infixToPostfix("( A + B ) * C - ( D - E ) * ( F + G )"))
```

运行结果如下所示：

```
>>> infixtopostfix("( A + B ) * ( C + D )")
'A B + C D + *'
>>> infixtopostfix("( A + B ) * C")
'A B + C *'
>>> infixtopostfix("A + B * C")
'A B C * +'
>>>
```

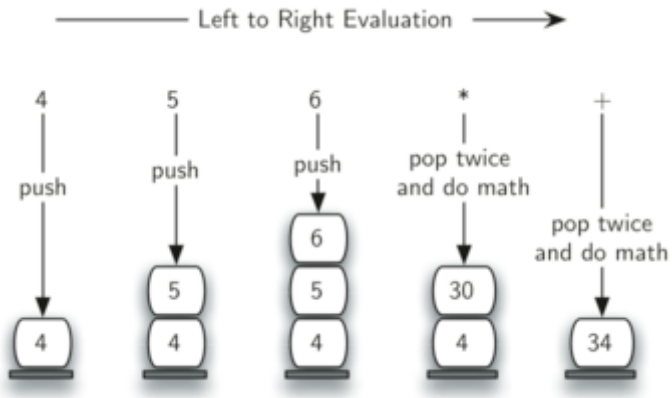
3.9.3 后缀表达式计算

作为栈的最后一个例子，来研究一下后缀表达式的计算。同样地，也考虑使用栈作为该问题的数据结构。然而，当遍历后缀表达式时，需要等待的是操作对象，而不是像转换过程中那样由操作符来等待。考虑该解决方案的另一种思路是，一旦在输入红发现操作符，编队最近的两个操作对象进行计算。

再仔细研究一下，比如说后缀表达式 4 5 6 * +。当从左到右进行扫描时，首先遇到的是操作对象4，5。此时，并不能确定这两个对象将会进行什么操作，直到扫描到了下一个符号。将它们全部放入栈中能够保证遇到操作符时，它们是可用的。

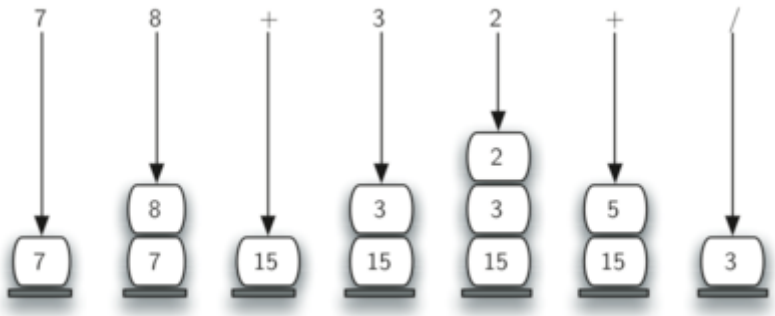
在本例中，下一个符号又是另一个操作对象。因此，跟之前一样，把它存入栈并且检查下一个符号。现在遇到了一个操作符，*。这意味着最近的两个操作对象要进行乘法运算。将栈pop两次便可以获得相应操作对象然后再进行乘法运算，得到的结果为30。

接下来将这个结果又放回栈中，这样一来它便可以作为下一个操作符的操作对象用来运算了。当处理到最后一个操作符时，栈中便只剩一个值了。将其pop，并且将它作为表达式的计算结果。图10演示了表达式的处理过程。



../_images/evalpostfix1.png

图11演示了一个复杂一些的例子，7 8 + 3 2 + /，有两点需要注意。首先，栈的规模随着子表达式的运算的进行而不断变化。其次，分号运算必须要仔细处理。后缀表达式中的操作对象的相对顺序并没有发生改变，后缀记法只会改变运算符的顺序。当除法的运算对象从栈中pop出时，它们需要作个反向处理。因为除法并不是可交换运算符，换句话说$15/5$与$5/15$是不一样的，必须保证运算对象顺序的正确性。



../_images/evalpostfix2.png

假定后缀表达式是以空格隔开的符号字符串，操作符有*、/、+、-，而且数字是仅有个位的整数。输出结果也将是个整数。

1. 创建一个名为operandStack的空栈。
2. 利用字符串方法split将字符串转为列表。
3. 从左至右遍历符号列表
    - 若符号是操作对象，将其从字符串转为整数，并将其放入operandStack栈顶。
    - 若符号是操作符*,/,+,-，则需要两个操作对象。将operandStack作两次pop，第一次pop作为该操作符的第二个操作对象，而第二个pop作为该操作符的第一个操作对象。执行数字运算。将结果放回operandStack。

4. 输入的表达式被全部处理完毕后，计算结果位于栈顶。将operandStack作pop，返回最终计算结果。

计算后缀表达式的完整函数如图2所示。为了实现数学运算，定义了一个domath函数，它接受两个操作对象和一个操作符，并进行正确的数学运算。

可执行代码2：后缀表达式的计算

```
from pythonds.basic.stack import Stack

def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()

def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2

print(postfixEval('7 8 + 3 2 + /'))
```

值得注意的是，后缀表达式转换和后缀表达式计算程序中，都假定了输入的表达式没有错误。以这些程序为起点，读者可以轻松地引入探索错误检测和报告。在本章末，这将作为一个练习。

3.10 何为队列Queue

队列queue是一种有序数据容器，其增加操作位于其“队尾”，而其删除操作位于“队首”。当某一个元素加入该队列时，它便不断向队首移动，直到成为即将被移除的那一项。

最新加入队列的元素必须位于容器的末端。在容器中储层得最久的元素位于队首。这种定序方法有时被称为**FIFO**，即**先进先出法**。

队列最简单的例子是便是生活中经常见到的排队。人们排队看电影，排队进入杂货店，在自助餐厅排队（这样便可以对餐盘pop，即依次获取）。秩序良好的队列，必须严格地只有一个入口和一个出口，不可插队，也不可在到达队首前离开。图1是一个简单的Python数据对象的队列。

../_images/basicqueue.png

计算机科学有一些非常常见的队列。例如，假设实验室有30台电脑和1台打印机。当学生想要进行打印时，它们的打印任务会进入队列中，队列中也有其它等待中的打印任务。位于队首的任务是即将被处理的。位于队列最后的任务，必须要等待其前面的所有任务都完成才能被处理。稍后将进一步分析。

除了打印队列外，操作系统使用大量队列来控制计算机内的进程。这种进程的调度通常是基于某种算法，这种算法能够使得程序可以尽快地执行并且尽量供更多用户服务。同样地，在打字的时候，有时敲击完成后屏幕上的显示有所延迟，这是因为那一瞬间计算机也在处理其它任务。按键信号被放置于类似于杜列的缓冲区，这样它们最终会在屏幕上以正确的顺序显示。

3.11 抽象数据类型队列queue

抽象数据类型队列通过以下结构和操作来定义。如前文所述，队列是一种有序容器，其增加操作位于队尾，其删除操作位于队首，符合先进先出。队列的操作如下。

- Queue()创建一个空队列。无需参数并返回空队列。
- enqueue(item)将元素加入队列尾部。需要元素作为参数，无返回。
- dequeue()从队首去掉元素。无需参数并返回被去掉的元素。队列本身发生了修改。
- isEmpty()判断队列是否微孔。无需参数并返回布尔值。
- size()返回队列中元素个数。无需参数并返回整数。

作为例子，假设q是一个空队列，则一系列队列操作的结果如表1所示。队首位于右侧。

Queue Operation	Queue Contents	Return Value
`q.isEmpty()`	`[]`	`True`

Queue Operation	Queue Contents	Return Value
`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`

3.12 在Python中实现队列Queue

同样地，实现抽象数据类型queue的合理方法是创建一个类。跟之前一样，本文使用简单而强大的列表来作为队列的内部实现。

首先需要确定列表的哪一端作为队尾或者队首。代码1中的实现以列表的索引0为队尾。这样便可以使用列表的insert函数来向队尾添加新元素，pop操作用来移除队首的项。这样一来，enqueue的复杂度是O(n)，而dequeue的复杂度则是O(1)。

代码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)
```

代码实例1 对表1的操作进行了演示。

```
| 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()) |
```

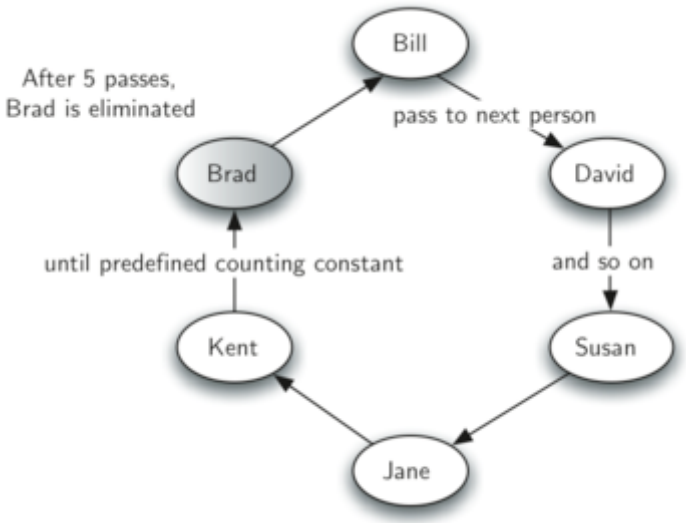
队列的其它一些操作结果如下：

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

```
>>> q.size()
2
```

3.13 模拟：热土豆

演示队列的运行的一个实际应用便是模拟使用FIFO管理方式的真实情景。首先，考虑一个叫热土豆的儿童游戏。在这个游戏中（如图2所示），孩子们围成一个圆圈并以最快的速度向下一位传递物品。在游戏中的某个时刻，停止传递，手中有物品（土豆）的小孩就离开圆圈，然后继续游戏直到仅剩一个小孩子。

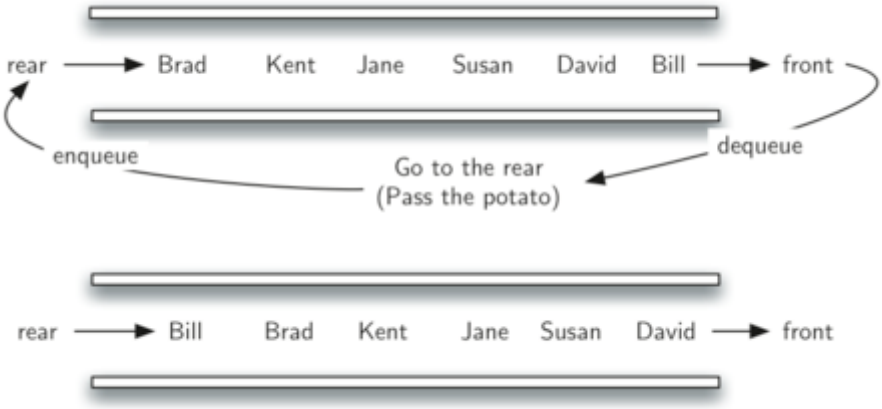


../_images/hotpotato.png

该游戏实际上是等价于一个著名的现代问题，Josephus问题。传说，公元1世纪的历史学家Flavius Josephus在犹太民族反抗罗马统治的起义中，与他的39个同胞在一个洞穴中与罗马人对抗。虽然失败已经注定，但他们决意宁死也不做罗马的奴隶。他们排成一个圈，其中某一个人被指定为第一个，然后按顺时针方向计数，每数到第7个人便将其杀死。Josephus居然还是一个数学家，他立刻算出并站到了那个能留到最后的位置。到了最后，他并没有自杀而是选择了投降罗马人。这个故事有其它版本。有些是以3为计数单位，有些是说最后一个人是可以骑马逃走的。总之，其核心原理是一样的。

本文将实现热土豆的一种大体上的模拟。程序将输入一个名字列表和一个常数，称为"num"，用来作为计数的基本单位。它将返回经过多次基于num的计数后剩下的最后一个人。对这个人的处理就随便了。

本文使用了一个队列。假定当前握着土豆的孩子位于队首。一旦开始传递土豆，模拟器将这个孩子移除队首并放入队尾，现在必须等到他前面的那些孩子都轮过一次后他才能再次到达队首。每经历num次dequeue+enqueue操作后，位于队首的孩子就被永久移出队列，并开始下一次循环，直到队列中仅剩一个。



../_images/namequeue.png

程序如可执行代码1所示。使用7的hotPatato函数返回结果是Susan。

** 可执行代码1：热土豆模拟 **

```
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))
```

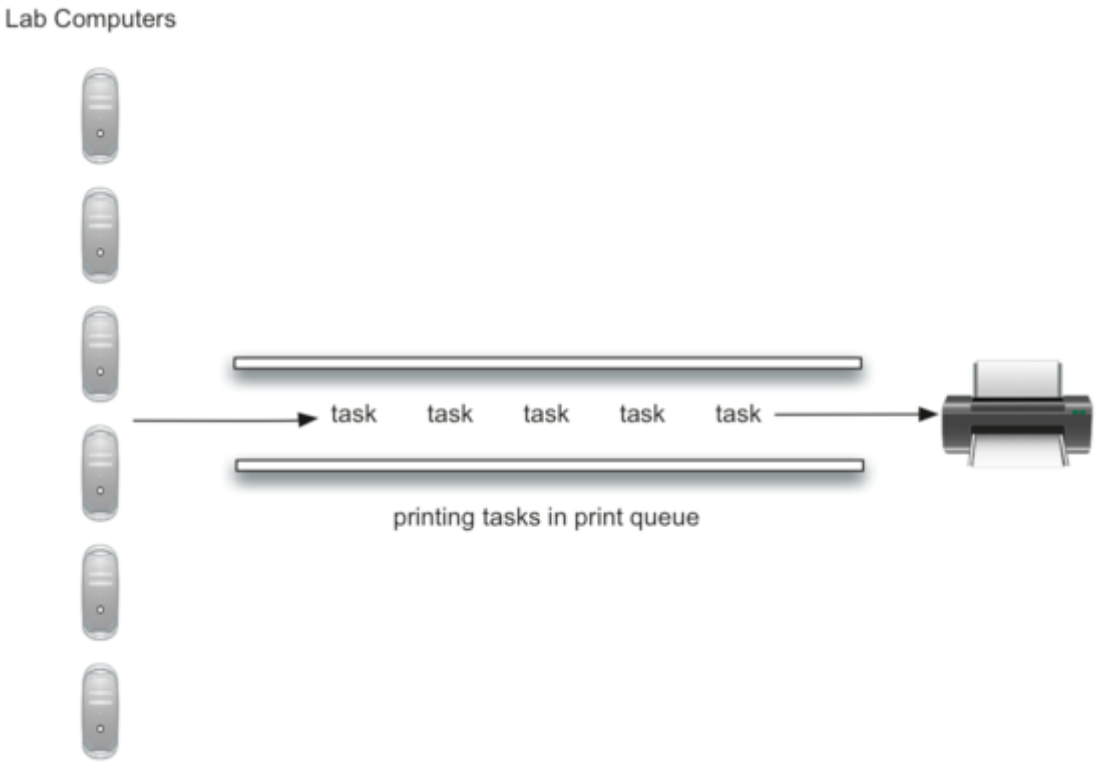
注意到在本例中，计数基本单位比名字列表中的元素个数还要大。这也并不是各问题，因为queue是以圆圈的方式循环运行。同时，列表被载入了队列中，这样以来列表中的第一个名字便成为了队列的队首。

3.14 模拟：打印任务

一种更有趣的模拟可以用来研究前文提到的打印队列问题。回想一下，学生将打印任务发送至共享打印机，任务以先进先出的方式排列于队列中。这种配置存在很多问题。最重要的问题也许是，打印机到底能不能处理某一个数量的任务。如果不能的话，学生不得不等待太长时间以至于错过下堂课。

考虑计算机科学实验室的以下情况。平均每天在任意1个小时之内都有大约10各学生在工作，在此期间，通常每个人会发起两次打印任务，每个打印任务在1到20页之间不等。实验室的打印机比较老旧，以草稿模式每分钟仅10页，以高质量模式每分钟仅5页。打印速度越慢，学生等得越久。那么应该采取哪种打印速度?

可以通过构造模拟器来对该实验室的情况进行建模，然后给出答案。对于学生、打印任务、打印机要给出其表征量，如图4。学生在提交打印任务时，将其放入等待列表中，即一个对应打印机的任务队列。当打印机完成一项任务时，它会检查队列中是否还有任务等待处理。我们关心的是平均每个学生等待打印的时间长度，这等价于求取队列中任务的平均等待时间。



../_images/simulationsetup.png

模拟这种情况需要利用概率来建模。比如说，学生可能会打印1到20页的文档。如果每个任务的打印页数是平均分布于1-20的话，那么每一次打印的实际页数就可以用闭区间[0:20]的随机数来表示。这意味着1到20页的出现概率是一样的。

如果实验室中有10个学生，并且每个人打印2次，那么平均每个小时会有20个打印任务。那么在给定的1秒内，产生一个打印任务的概率为多少？这可以通过任务数与时间的比值解决。每小时20个打印任务意味着平均每180秒1个打印任务：

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

对于每秒来说，可以通过生成位于闭区间[1:180]的随机数来模拟产生打印任务的概率。如果数字为180，便认定产生了一个打印任务。注意，有时会接连生成多个任务，有时也会长时间没有任务出现，这是模拟本身的特点。在知道了一些大概的参数后，尽可能地模拟真实情况。

3.14.1 主要模拟步骤

以下为主要模拟过程：

1. 创建一个打印任务队列。每个任务一旦生成便被打上了一个时间戳。队列在初始时为空。
2. 对每秒内（**currentSecond**）：
    - 是否有新的打印任务？若有，则将其加入队列中，并以currentSecond作为时间戳。
    - 若打印机空闲且队列中有任务：
        - 从打印队列中去掉下一个任务，并将其提交给打印机。
        - 将时间戳与currentSecond求差以计算该任务等待的时间。
        - 将等待时间加入列表中用于后续处理。
        - 根据打印任务的页数，计算出打印耗时。
    - 如果有必要，在当前的currentSecond，打印机将工作1秒。这样完成当前任务所需时间将会减少1秒。
    - 如果任务已完成，换句话说任务必要时间为0了，打印机便进入了闲置状态。
3. 在完成模拟后，根据等待时间列表来计算平均等待时间。

3.14.2 Python实现

设计这个模型，我们需要为上文提到的三个真实对象创建类：**Printer**, **Task**, **PrintQueue**。

打印机类（代码2）需要追踪当前是否有任务。如果有，那么就是忙碌状态（13-17行），并且完成任务所需时间可以根据该任务的页数来计算。构造器也允许设置 pages-per-minute（每分钟打印页数）。tick方法将计时器的时间减1，如果任务完成则将打印机设置为空闲状态。

```
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
```

Task类（代码3）代表单个打印任务。当打印任务创建时，随机数生成器会给出[1:20]内的页数。这里使用的是random模块中的randrange函数。

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

每个任务都必须要保存一个时间戳来计算等待时长。时间戳代表了该任务生成和放入队列中的时间。waitTime方法用来获取在打印开始前的等待时间。

** 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
```

simulation（代码4）实现了以上算法。printQueue对象是现有的ADT队列的一个实例。newPrintTask是一个返回布尔值的函数，用来描述是否生成了新任务。这里再次使用了函数randrange来返回[1:180]的整数（行32）。据此来模拟该随机时间。simulation函数可以为打印机设置总时间和每分钟打印页数。

** 代码4 **

```
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)
```

运行该模拟时，每次的结果都是不一样的，这是由于随机的概率性质导致的。重要的是当模拟的参数发生改变时，结果的变化趋势如何。一下是一些结果。

```
>>>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.
```

```
>>>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.
```

可以通过可执行代码1进行测试。

可执行代码1：打印机队列模拟

```
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)
```

3.14.3 讨论

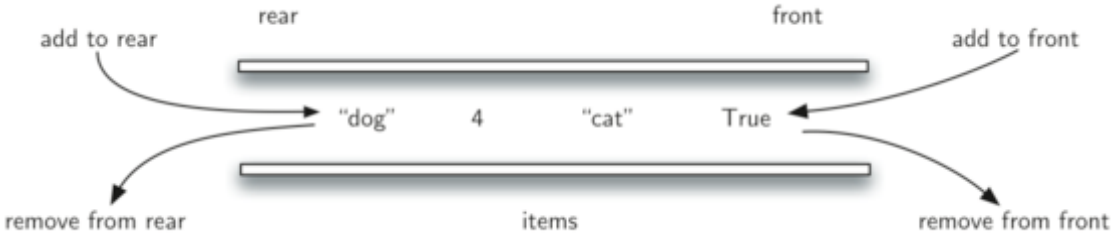
这里尝试回答打印机是否能在高质量的低速打印状态下满足打印任务负载。使用了模拟程序来将打印任务建模为具有变长度和生成时间的随机事件。

结果表面

3.15 双端队列deque

双端队列与队列相似，也是一种有序数据容器。它有两端，队首和队尾，与队列不同的是，元素在两端都可以增删。某种意义上，这种混合线性结构实现了栈和队列的所有功能。图1是由Python数据对象构成的deque。

应当指出，虽然deque具有很多栈和队列的特征，但它并不严格地要求先进先出或者后进先出，因此增删操作的连续性必须由用户自己维护。



../_images/basicdeque.png

3.16 抽象数据类型双端队列Deque

抽象数据类型双端队列deque由以下结构和操作定义。如前文所述，双端队列也是一系列元素的有序容器，两端分别称为队首和队尾，元素可以从两端进行增删操作。deque的操作如下。

- Deque() 创建一个空deque。无需参数并返回空deque。
- addFront(item) 向队首加入元素。接受该元素作为参数，返回空。
- addRear(item) 向队尾加入元素。接受该元素作为参数，返回空。
- removeFront() 从该deque移除位于队首的元素。无需参数，并返回该元素。
- removeRear() 从该deque移除位于队尾的元素。无需参数，并返回该元素。
- isEmpty() 检测该deque是否为空。无需参数，并返回布尔值。
- size() 返回deque中的元素个数。无需参数，并返回整数。

下面举例说明，假设d是一个创建好的空deque，表1演示了对其进行一系列操作的结果。注意，deque中队首的元素位于列表右侧。由于双端都可以进行增删操作有时容易引起混淆，因此一定要跟踪队首和队尾的状态。

Deque Operation	Deque Contents	Return Value
`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`

Deque Operation	Deque Contents	Return Value
`d.removeFront()`	`['dog',4,'cat']`	`True`

3.17 在Python中实现deque

跟前几节一样，本文将建立一个新的类用于实现抽象数据类型。同样，Python的列表提供了一些非常好的方法可以用来实现deque的细节。本文的实现设定deque的队尾位于列表的索引0处。

代码1

```
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)\
```

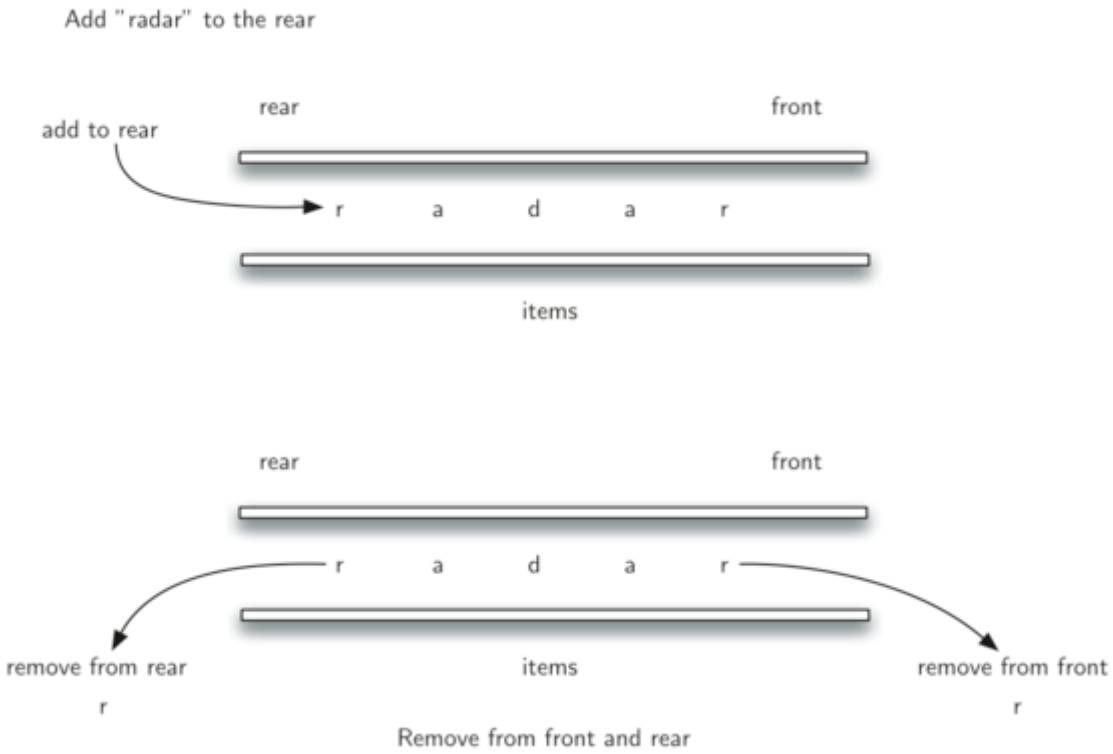
利用pop来实现removeFront。然后，只能用pop(0)方法来实现removeRear。类似地，addFront和addRear分别使用append和insert(0,item)实现。

读者可以发现，这与stack栈和queue队列的Python代码有很多相似点，并且该实现中，从队首进行增删操作的复杂度都是O(1)，而从队尾进行增删操作的复杂度都是O(n)。这对再强调一次，各种实现中都必须确定队首和队尾的位置。

3.18 回文词检测器

使用deque数据结构可以轻松解决经典的回文词问题。回文即正序和逆序都是一样的单词，比如说radar，toot，和madam。本节建立一个算法，检查输入的字符串是否为回文词。

该问题的解法使用一个deque来存放字符串的字母。从左至右处理字符串，并且将每个字母加至deque队尾。就现在而言，这个deque看起来很像一个queue。然而，它可以充分使用deque的双端特性。deque队首会保存字符串的第一个字母，而队尾则是其最后一个字母（如图2所示）



../_images/palindromesetup.png

可以直接将两个元素移除，然后将这两进行比较确定是否匹配。如果一直对队首和队尾的元素进行配对，最终deque中元素数量为0或者1，取决于字符串长度为奇或偶。不管是哪种情况，该字符串一定都是回文词。完整代码如可执行代码1所示。

```
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"))
```

3.19 列表

在对基本数据的讨论中，本书使用了Python的列表来实现一些抽象数据类型。列表是一种强大而简单的容器机制，为程序员提供了很多操作。然而，不是所有的编程语言都有列表容器。在这种情况下，程序员只能自己来实现列表了。

列表是一种元素的容器，其中的每一个元素都根据其它元素有一个相对的位置。更具体地说，这种列表被称为无序列表。列表可能有第一项，第二项，第三项，并且也可以索引到首项和末项。为简单起见，假设列表不含重复元素。

比如说，整数54，26，93，17，77，31的容器可能是考试分数的简单无序列表。注意，这里是用逗号分隔进行的书写，这是列表结构常用的表达方式。当然，Python会将该列表显示为[54,26,93,17,77,31]。

3.20 抽象数据类型：无序列表

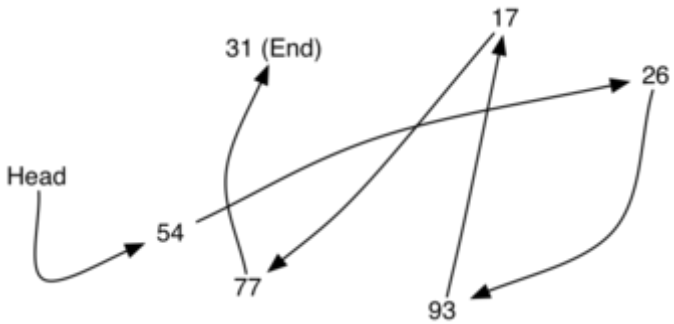
无序列表，如前文所述，是一种数据容器，其中的元素拥有与其它元素不同的相对位置。一些可能的无序列表的操作如下所示。

- List()　创建一个空列表。无需参数，返回该空列表。
- add(item) 将新元素添加至列表。它需要该元素作为参数，无返回结果。这假设了该元素不在列表中。
- remove(item) 将元素从列表中移除。它需要该元素作为参数，并改变了列表。这假设了该元素在列表中。
- search(item) 在列表中对元素进行搜索。它需要该元素作为参数，并且返回布尔值。
- isEmpty() 判断该列表是否为空。无需参数并返回布尔值。
- size() 返回列表中元素的个数。无需参数并返回整数。
- append(item) 将新元素加入列表的末端并保证它是容器中的最后一项。它需要该元素作为参数，并且返回为空。这假设了假设该元素不在列表中。
- index(item) 给出列表中该元素的位置。它以该元素作为参数，返回索引值。这假设了假设该元素在列表中。
- insert(pos, item) 将新元素添加至pos位置。它需要该项作为参数，返回空。这假设了列表中不存在item且在pos位置已经另有元素。

- pop() 去除并返回列表最后一个元素。无需参数并返回该元素。这假设了该列表至少有1个元素。
- pop(pos) 去除并返回在位置pos的项。需要该位置pos作为参数并返回该元素。这假设了在位置pos有元素存在。

3.21 实现无序列表：链表

为了实现无需容器，可以构建著名的**链表**。回想一下，元素相对位置的正确性是必须要确保的。然而，并不要求要将其放置于连续的内存空间中。比如说，考虑图1所示的元素。似乎这些值都是被随机放置的，如果在每一个元素中额外保存一些明确信息，即下一个元素的位置，那么每一个元素的相对位置都可以简单地通过前后两个元素的链进行表达了。



../_images/idea2.png

特别值得注意的是，列表首项的位置必须被明确指定。一旦确定了第一个元素的位置，便可以知道第二元素的位置，并依此类推。外部引用通常是指向列表的头部。类似地，最后一项必须确定在其之后便没有元素了。

3.21.1 类：节点Node

用以实现列表的基本模块是**node**（节点）。每个node对象必须至少保存两个信息。首先，节点必须包括列表元素本身，可以将其称为“数据区”。此外，每个节点必须保存对下一个节点的引用地址。代码1给出了Python代码实现。为了构造node，必须给出node的初始数据值。执行下面的赋值语句会产生一个包含值93的node。请记住，本书一般以图4的方式来标识node对象。Node类也应该包括一些常用的方法，比如说获取和调整数据及下一个节点的引用地址。

代码1

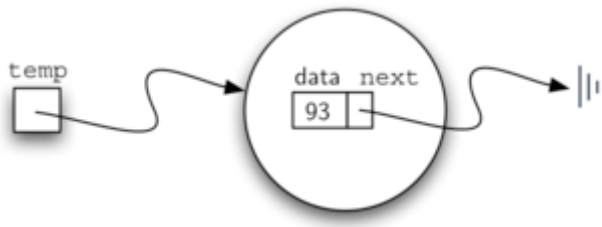
```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    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
```

按一般的方式创建Node对象即可：

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

Python的特殊值**None**在Node类和之后的链表中都起到了重要作用。对None的引用表明没有下一个节点了。注意到在构造器中，node对象的next被初始化为None。有时这被成为接地节点，因此用接地符号来表示指向None的节点。用None来显示地初始化节点对下一个的引用地址是很不错的操作。



../_images/node.png

3.21.2 类：无序列表 Unordered List

如前文所述，无序列表将由node对象组成，其中每一个元素都用显示的引用地址链接到下一个元素。只要确定第一个节点的位置，之后的每一个元素都可以沿着链依次找到。有了这个想法，那么很明显UnorderedList类必须要保存第一个节点的引用。其构造器如代码2所示。注意，每一个列表类都必须保存一个指向列表头的引用。

** 代码2 **

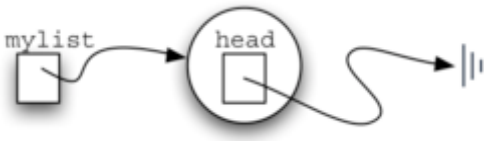
```
class UnorderedList:
```

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

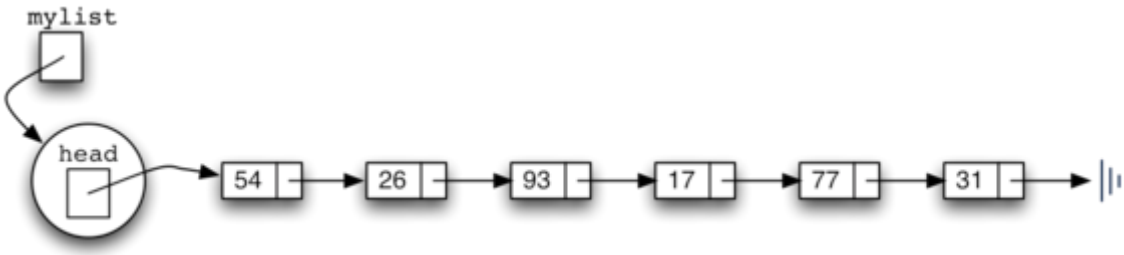
在初始化列表时，其中是没有元素的。赋值语句

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

创建了一个如图5所示的链表。与之前Node类相同，特殊引用None表明列表的头部并不指向任何东西。最终，如图6所示，之前给出的示例列表将以链表的形式给出。列表的头指向第一个节点，其中含有列表的第一个元素。以此类推，该节点包含一个指向下一节点的引用。注意，列表类本身不包含任何节点类，它仅保存了指向链式结构的第一个节点的引用。



../_images/initlinkedlist.png



../_images/linkedlist.png

isEmpty方法，如列表3所示，只需要检查列表的头是否指向None，布尔表达式**self.head==None**只有当链表中没有节点时才会为ture。因为新列表是空的，构造器和对是否为空的检查必须与另一个一致。这便是使用None来标识链式结构末端的好处。在Python中，None可以和任何引用进行比较。两个引用只有在它们指向同一对象时才是相等的。在后续的方法中经常使用到这一点。

** 代码3 **

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

那么，如何从列表中获取(get)元素？还得要实现add方法。然而，在这之前，还需要解决将新元素放于链表的何处这一问题。因为列表是无序的，新元素相对于列表中已有元素的位置其实是不重要的。新元素可以放置于与任何位置。明白这一点就知道，将新元素放于最简便的位置是最好的。

回想一下，列表仅提供一个入口，即列表的头。其它所有的节点只能通过访问第一个节点并沿着链依次获取。这意味着，新元素最简单的放置处恰是头部，或者说列表的开始。换句话说，将元素作为列表的第一项，并将现有的元素链接到这个新元素，便保持了链式结构。

图6所示的链表通过多次使用add方法进行了构建：

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

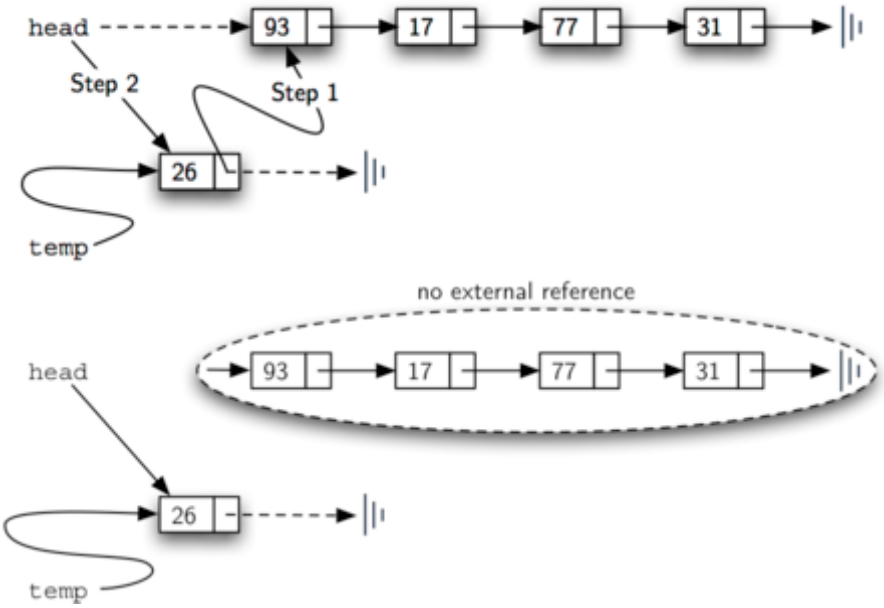
因为31是首先被加入列表的元素，它最终会成为链表的最后一个节点，因为其它的每一项都被放置于它的前面。并且，因为54是最后加入的一项，它会成为链表第一个节点的数据值。

代码4给出了add方法。列表中的每一个元素必属于某一个节点。行2创建了一个新节点并且将元素作为其数据。接下来必须将新街点链接到已有的结构体中。这需要如图7所示的两步。第一步（行3）将next，即新节点的引用指向原列表的第一个节点。这样一来，列表的其它元素都与该节点正确地链接了，现在便可以将列表的头部改为新加入的那个节点了。第4行的赋值语句设定了列表的头部。

上述两步的顺序是很重要的。如果行3和行4的顺序反了会怎样？如果先设定列表的头部，其结果如图8所示。由于头部是外部对列表节点的唯一引用源，这样做的话，其它的节点便会丢失并且无法再次访问了。

代码4

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

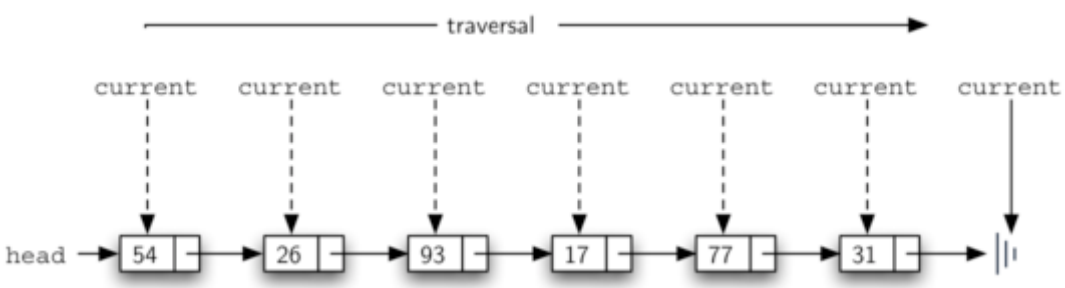


接下来实现size，search和remove方法，并且都基于所谓的**列表遍历**技术。遍历是指系统性地访问每个节点。为此，先利用一个外部地址来指向列表的第一个元素。当访问至某一节点时，利用**next**将这个引用地址指向为下一个节点。

为实现size方法，需要遍历整个列表并且跟踪出现的节点数。代码5演示了对列表中元素进行计数的方法。外部引用地址被命名为current并且在行2中初始化为链表的头部。在程序起始时，由于并没有遍历到任何节点，因此计数为0。行4-6实现了遍历。只要**current**引用地址变为列表的尽头（None），便通过行6的赋值语句将current指向下一个节点。再次强调，将引用与None进行对比是很重要的操作。每当current移至下一个节点时，便将count加1.最后，迭代完成后返回count。图9给出了全过程的示意图。

代码5

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

    return count
```



../_images/traversal.png

在链表中查找某一个值也需要使用遍历技术。当访问链表中的每一个节点时，都检查一次是否是所查找的元素。在本例中，也许并不用遍历至链表末端。实际上，如果真的到了链表的末端，那说明所查找的元素必然不在此链表中。如果找到了，没必要继续遍历。

代码6给出了search的实现。跟size方法一样，遍历是从链表的头部开始的。另外也使用了一个命名为found的布尔值来标识是否找到了所查找的元素。因为在遍历开始的瞬间并没有找到目标元素，因此found被设置为False（行3）。行4中的遍历考虑了之前提到的两种情况。只要还有节点可以访问并且没有找到目标元素，便继续检查下一个元素。行5的判断语句检查当前节点是否出现了目标元素。如果找到了，则将found设置为True。

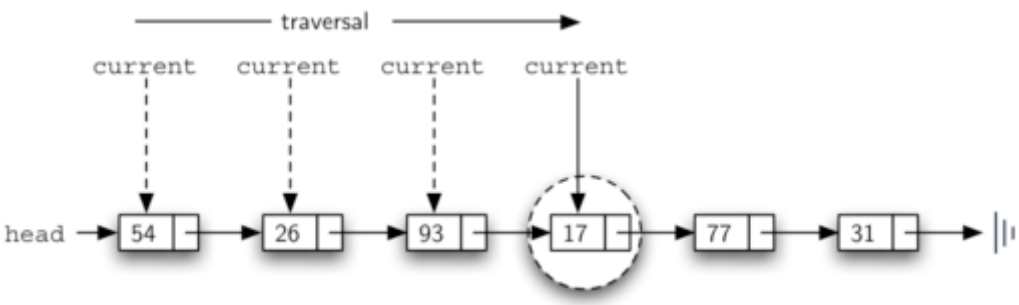
代码6

```
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
```

这里以17为例演示了search方法。

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

因为17确实是在链表中，遍历只需要到达包含17的那个节点即可。一旦抵达，便将found设置为True，这样while的条件语句便为False了，于是返回上例演示的结果。全过程如图10所示。



../_images/search.png

从逻辑上来说，remove方法需要两步。首先需要遍历链表来找到想要移除的目标元素。一旦找到该元素（假定列表中含有该元素），便将其移除。第一步非常类似于search。开始时外部引用地址指向链表的头部，遍历该链表直到发现目标元素。既然假设列表中含有目标元素，可以确定迭代会在抵达链表尾部的None之前停止。那么，在这种去哪个看下可以直接使用布尔值found。

当found变为True时，current应当指向包含待移除元素的节点。但是如何移除它？一种可能的办法是将值替换为某个标记来表示该元素已经不在链表中了。问题是这样一来，节点数便与元素数不一致了。将整个节点移除的话，其可行性高得多。

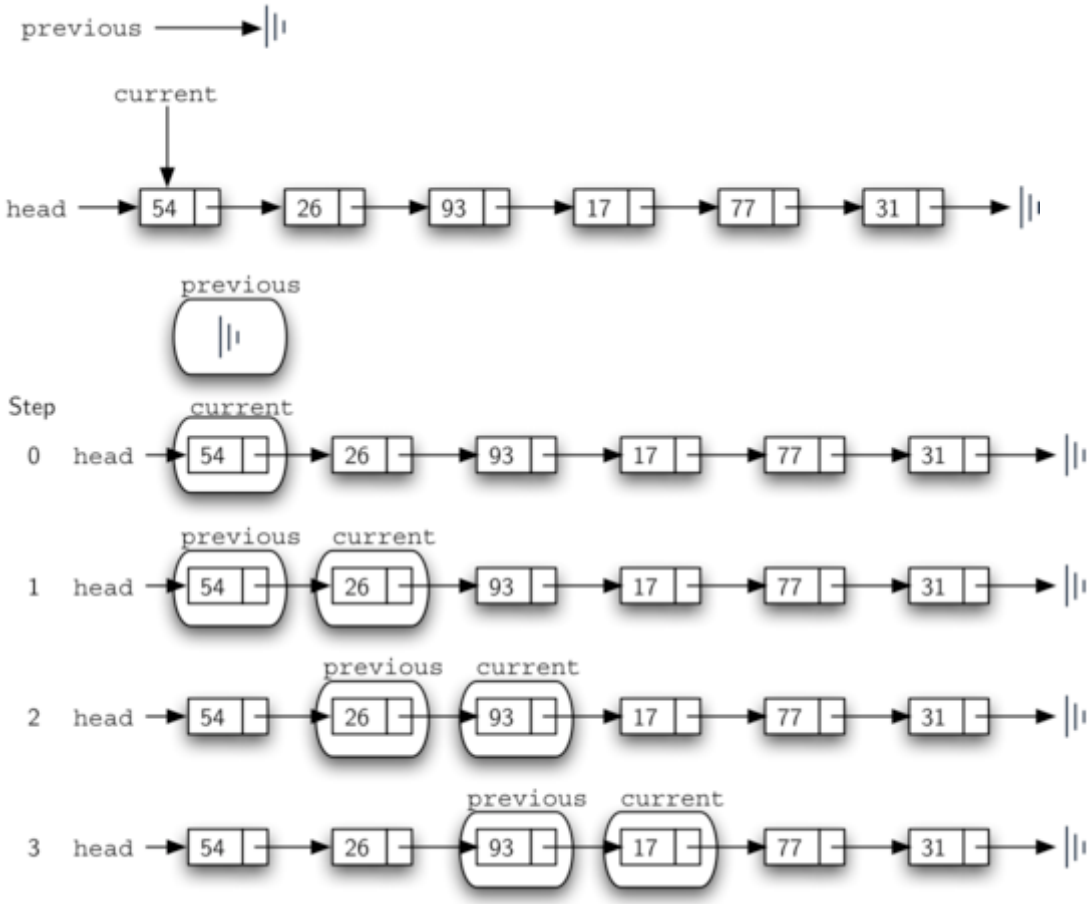
为了将包含该元素的节点删除，必须要将前一个节点的链接指向current之后的那个节点。不幸的是，在链表中是没有办法从后向前移动的。因为current指向的是需要修改的那个节点之后，所以修改操作已经不可行了。

解决这一困境的方法是使用两个外部引用。current仍照旧，标记当前遍历的位置。新的外部引用，命名为previous，在遍历时落后current一个节点。因此，previous的初始值为0，因为在头部之前是没有节点的。found仍然用来控制迭代。

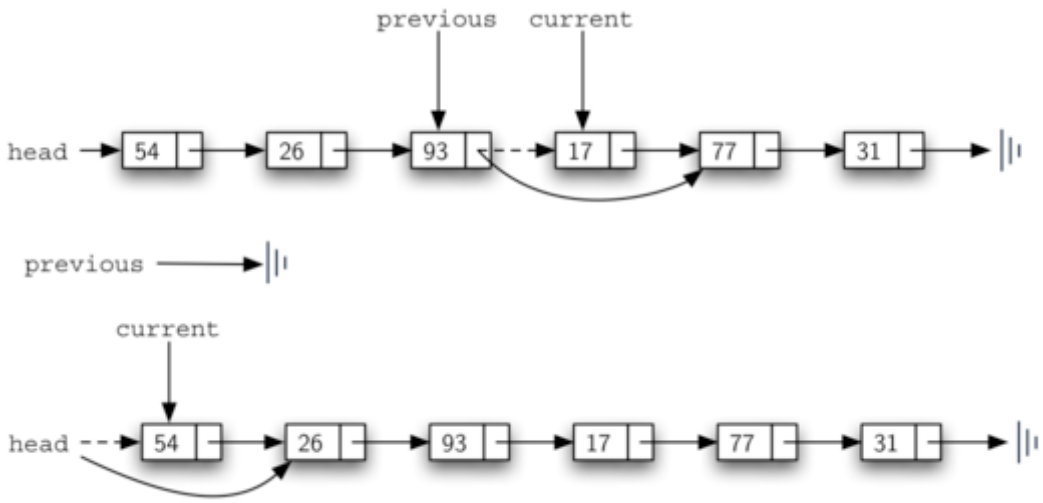
6-7行中检查当前节点储存的元素是否为目标元素。如果是，则将found赋为True。如果没有找到该元素，previous和current必须同时向后移动换一个节点。这一过程常被称为“蠕动”，因为在current继续向后移动之前，previous必须要跟上current。图12演示了在链表中查找元素为17的节点时，previous和current的运行方式。

代码7

```
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())
```



一旦remove的搜索操作完成，便需要将节点从链表中移除。图13给出了调整链接的示意图。这里有一点需要特别说明。如果目标元素恰好是链表中的第一个元素，current便会指向链表中的第一个元素，previous为None。前文说过，previous应是指向next将会发生变化的节点。然而在现在这种情况下，需要修改的不是previous而是链表的头部引用。



行12便可以处理上文提到的特殊情况。如果previous没有发生移动，当found变为True时，它仍然为None。此时（行13），链表的头部被改为current之后的那个节点，实际上便把原来的第一个节点从链表中删除了。如果previous不为None，那么目标节点应位于表头之后的某处。此时，previous便给出了要修改next的那个节点。行15在previous使用了setNext方法来完成移除。注意在两种情况下，改动的引用最终都变成了current.getNext()的结果。读到这里，读者往往会产生一个疑问，上述的两种情况是否适用于目标节点位于链表尾的情况？这个问题留给读者自己思考。

读者可以用可执行代码1测试unorderedList类。

```
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


class UnorderedList:

    def __init__(self):
        self.head = None

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

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

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

        return count

    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

    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())

mylist = UnorderedList()

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

print(mylist.size())
print(mylist.search(93))
print(mylist.search(100))

mylist.add(100)
print(mylist.search(100))
print(mylist.size())

mylist.remove(54)
print(mylist.size())
```

```
mylist.remove(93)
print(mylist.size())
mylist.remove(31)
print(mylist.size())
print(mylist.search(93))
```

剩下的方法比如append, insert, index和pop就作为练习。务必注意，一定要仔细考虑这些操作是发生在链表的头部还是其它地方。此外，insert，index和pop要求给链表的位置命名。本书要求列表中位置的名称是从0开始的整数。

3.22 抽象数据类型：有序列表

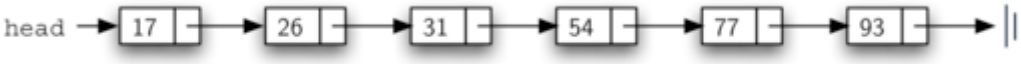
接下来研究有序列表。比如说，如果前文的整数列表是一个有序列表（升序），那么可以写作17，26，31，54，77。因为17是最小的元素，因此它位于列表的第一个位置。同样地，因为93是最大的，所以它位于列表的最后一个位置。

有序列表是一种有序容器，其中的元素根据其某基本性质来决定相对于其它元素的位置。假设这种对比是有准确定义且有意义的，此时所谓的“有序”典型地就是指升序或者降序。有序列表的许多操作同无序列表一致。

- OrderedList() 创建一个空有序列表。无需参数返回空列表。
- add(item) 保证原有顺序的前提下向列表添加新的元素。以该元素作为参数，无返回。这里假设列表中原先并没有该元素。
- remove(item) 从列表中移除元素。以该元素作为参数，会修改原列表。这里假设该元素存在于原列表中。
- search(item) 在列表中搜索目标元素。需要该元素作为参数并返回布尔值。
- isEmpty() 检测列表是否为空。无需参数并返回布尔值。
- size() 返回列表中元素的数量。需要该元素作为参数并返回索引值。这里假设该元素存在于列表中。
- index(item) 返回列表中该元素索在的位置。需要该元素作为参数并返回索引值。假设该元素存在于列表中。
- pop(item) 去除并返回列表中的最后一项。无需参数并返回该元素。假设该列表至少存在一个元素。
- pop(pos) 去除并返回列表中pos位置的元素。需要pos作为位置参数并返回被去除的元素。这里假定该列表pos处有元素。

3.23 实现有序列表

为了实现有序列表，一定要意识到元素间的相对位置是由某些基本特征决定的。之前给出的整数列表(17,26,31,54,77)可以用图15所示的链式结构标识。同样地，节点和链结构对于标识不同元素间的相对位置是很有用的。



../_images/orderlinkedlist.png

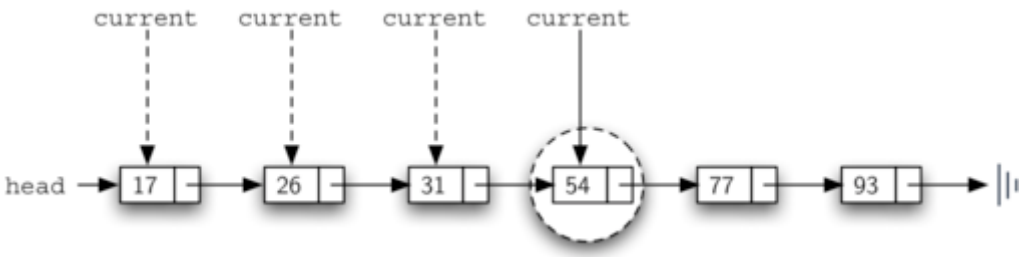
这里使用与UnorderedList类相似的方法来实现OrderedList类，同样，空列表通过指向None的head来表示，如代码8所示：

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

考虑OrderedList类的操作，注意到isEmpty和size方法可以用跟UnorderedList一样的实现，因为他们都仅处理节点的个数而与元素的值无关。类似的，remove方法也可用。然而，search和add方法需要一些调整。

无序列表的搜索需要遍历整个节点直到找到目标元素或者到达列表末（None）。事实上，在列表中确实有目标元素的情况下，相同的方法也可用于OrderedList。然而，如果目标元素不在列表中，可以利用有序性来尽早结束遍历。

比如说，图16中的例子中，当前正在查找值45。从列表的头部开始遍历，首先与17进行对比。因为17并不是目标因素，所以移向下一个节点，26，同样也不是，继续下移....在54时，按照之前的规则应该继续下移。然而，因为这个列表是有序的，这种操作是没有必要的了。一旦节点中的元素值超过了目标元素，搜索便可以停止并返回False，因为目标元素不可能存在于该链式列表的之后的位置了。



../_images/orderedsearch.png

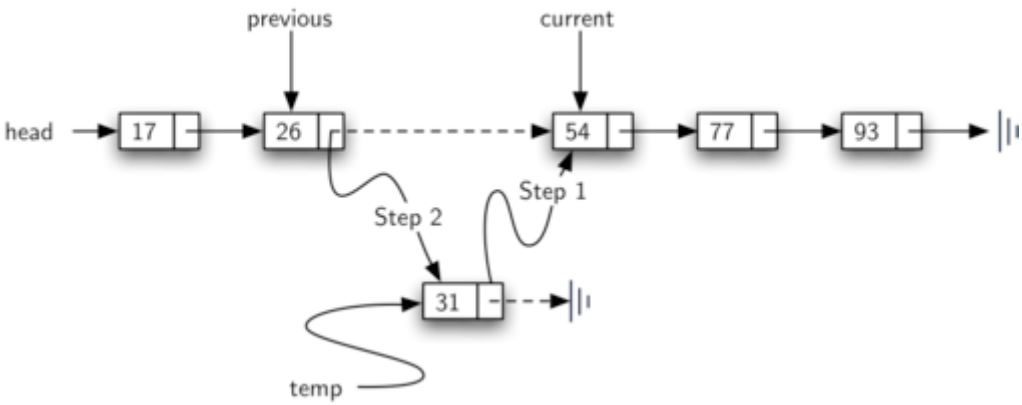
代码9给出了完整的search方法。很容易把刚刚说明的条件整合进来，只需要增加另一个布尔变量，stop，并且将其初始化为False（行4）。当stop为False（not stop），便可以继续在列表中向前搜索（行5）。如果发现某节点的元素比目标元素更大，便可以将stop设为True（行9-10），剩下的行与无需列表的search方法一致。

```
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
```

add的调整是最大的。回忆下，对于无需列表来说，add方法只需要将新节点放在列表的头部，这是最可行的方法。不幸的是，这对于有序列表并不适用，必须将新元素添加到原列表中的某个特定位置。

假设一个有序列表17，26，54，77，93，现在将31添加进去。add方法必须确定新元素是结余26和54之间的。图17演示了所需步骤。如之前所属，必须要遍历链表才能找到添加新节点的位置。可以明确的是，当所有节点都被遍历完或者当前节点的值比待添加元素更大时，便确定了新节点的位置。在给出的例子中，在54处停了下来。



../_images/linkedlistinsert.png

可见，对于有序列表来说，有必要给出额外的一个引用，同样地命名为previous，因为current并不能给出需要调整的那个节点的引用。代码10给出了add方法的完整代码。行2-3设置了两个外部引用而行9-10同样使得previpus在迭代时总落后current一个节点。判定条件（行5）要求必须存在下一个节点且当前节点的元素值不大于被添加元素时，迭代才继续。在两种情形下，迭代都将终止，此时便确定了新节点的位置。

该方法的剩余部分实现了图17的最后两步。一旦新节点，剩下的问题就是新节点将添加至链表头部还是中间的某个位置。同样， previous == None（行13）可以予以确定。

代码10

```
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)
```

orderedList类和上文讨论的方法都可在可执行代码1中找到。剩下的方法作为联系。使用有序列表时应当仔细考虑用于无需列表的实现能否直接适用。

可执行代码1

```
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
```

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

    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

    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)

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

    def size(self):
        current = self.head
```

```
        count = 0
        while current != None:
            count = count + 1
            current = current.getNext()

        return count


mylist = OrderedList()
mylist.add(31)
mylist.add(77)
mylist.add(17)
mylist.add(93)
mylist.add(26)
mylist.add(54)

print(mylist.size())
print(mylist.search(93))
print(mylist.search(100))
```

3.23.1 链表的分析

分析列表操作的复杂度需要确定其是否需要遍历。考虑有n个节点的链表。isEmpty()的复杂度为O(1)，因为它只需要检查头部引用是否为None。而size()总是需要n个步骤，因为不经过遍历的话便无法确定其到底有多少个节点。因此，length()的复杂度是O(n)。将新元素添加至无序列表的复杂度是O(1)，因为总是将新节点放于链表的头部。然而，search和remove方法以及有序列表的add方法，都需要遍历操作。尽管平均地来说仅会遍历一半的节点，但它们的复杂度仍为O(n)，因为在最差的情况下必须要遍历链表的每一个节点。

也许读者已经注意到了这些实现的性能跟Python的列表不一样。这实际上表明，Python列表并不是用链表实现的。Python列表实际上是基于数组思想的。在第八章会详细研究。

3.24 总结

- 线性数据结构以有序方式存储数据。
- 栈是采用LIFO（后进先出）的简单数据结构。
- 栈的基本操作有push，pop以及isEmpty。
- 队列是使用先进先出规则的一种简单数据结构。
- 前缀、中缀、后缀表达式都可以用来书写表达式。
- 栈可以用于设计表达式计算和转化的算法。
- 栈具有逆序的特点。

- 队列可用于构建计时模拟器。
- 使用随机数生成器可对实际生活进行模拟，并且给出一些“假设”问题的答案。
- 双端队列是一种结合了栈和队列各自特点的数据结构。
- 双端队列的基本操作是addFront，addRear，removeFront，removeRear，以及isEmpty。
- 列表是具有相对位置的元素的容器。
- 链表的实现在保持逻辑顺序的同时，对物理存储空间的顺序并无要求。
- 对链表头部的修改是一种需要特殊考虑的情况。