

基本数据结构

2017 年 10 月 19 日

目录

1 目标	2
2 什么是线性数据结构？	2
3 栈	2
3.1 栈的举例	3
3.2 栈的抽象数据类型	4
3.3 用 Python 实现栈	5
3.4 栈的应用	6
3.4.1 简单括号匹配	6
3.4.2 平衡符号（通用）	8
3.4.3 十进制转换成二进制	9
3.5 中缀、前缀与后缀表达式	11
3.5.1 基本概念	11
3.5.2 中缀表达式转换前缀表达式和后缀表达式	12
3.5.3 中缀转后缀通用法	13
3.5.4 后缀表达式求值	15
4 队列	17
4.1 什么是队列	17
4.2 The Queue Abstract Data Type	18
4.3 Implementing a Queue in Python	18
4.4 队列任务：烫手山芋	19
4.5 队列应用：打印任务	20
4.6 主要模拟步骤	21
4.7 Python 实现	22
4.8 讨论	24

1 目标

- 理解栈、队列、双向队列和列表的抽象数据类型
- 使用 Python 列表实现栈、队列和双向队列的抽象数据类型
- 理解基本线性数据结构的性能
- 理解前缀、中缀和后缀表达式
- 使用栈计算后缀表达式。
- 使用栈将中缀表达式转换为后缀表达式。
- 使用队列进行基本时序仿真。
- 学会根据问题性质，选择使用栈、队列和双向队列等合适的数据结构。
- 能使用结点和引用将列表实现转换为链表实现。
- 能比较链表实现与 Python 的列表实现的性能。

2 什么是线性数据结构？

我们开始数据结构的学习，从四种简单而功能强大的结构开始：

1. 栈 (stack)
2. 队列 (queue)
3. 双端列表 (deque)
4. 顺序表 (sequence)

它们都是一些数据的集合，数据项之间的顺序由添加或删除的顺序决定。一旦一个数据项被添加，它就与之前和之后加入的元素保持一个固定的相对位置。诸如此类的数据结构被称为线性数据结构。

线性数据结构有两端，有时称为左和右，有时称为前与后，称为顶部和底部也无不可，叫什么名字并不重要。重要的是数据结构增加和删除数据的方式，特别是增删的位置。例如，一种结构可能只允许从一端添加数据，而另一种结构则两端都行。

这些变种的形式产生了计算机科学最有用的数据结构。他们出现在各种算法中，可用于解决很多重要的问题。

3 栈

栈是一种线性有序的数据元素集合，其中数据的增删操作都在同一端进行。进行操作的一端通常称为“顶部”，另一端称为“底部”。

- 栈的底部很重要：数据项越接近底部，它在栈里的时间就越长。
- **最近添加的项总是最先被移除**，这种排序原则称为“后进先出”（LIFO）。
- 栈按“时间长短”来排列数据项：新来的在栈顶，老家伙们在栈底。

例:

中国有句成语，叫后来居上。原典故是这样说的：陛下用群臣，如积薪耳，后来者居上。栈的方式，可不就是堆柴草吗？

3.1 栈的举例

栈的例子很常见。

例:

自助餐厅的盘，人们总是从上面拿盘子，拿走一个后面的人再拿下面的一个，（服务员端来一些新的，又堆在上面了）。

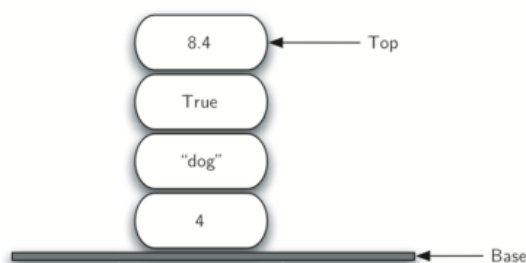
例:

又如一堆书，你只能看到最上面一本的封面，要看下面一本，就要把上面的先拿走。



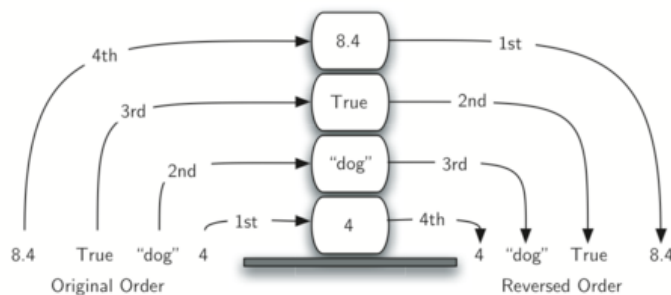
例:

下图展示了另一个栈，存储的是几个主要的 python 语言数据对象。



例:

与栈有关的思想来源于生活中的观察，假设你从一张干净的桌子开始，一本一本地放上书，这就是在建立栈。当你一本一本地拿走，想像一下，是不是先进后出？由于这种结构具有翻转顺序的作用，所以非常重要。下图展示了 Python 数据对象创建和删除的过程，注意观察他们的顺序。



例:

栈这种翻转性，在你用电脑上网时也用到。浏览器都有“返回”按钮，当你从一个链接到另一个链接，这时网址（URL）就被存进了栈。正在浏览的页就存在栈顶，点“返回”的时候，返回到刚刚浏览的页面。最早浏览的页面，要一直到最后才能看到。

3.2 栈的抽象数据类型

栈的抽象数据类型由以下结构和操作定义：

- 栈是结构化的、有序的数据集合，其增删操作都在“栈顶”进行，存储顺序是 LIFO。
- 栈的操作方法如下：
 1. `Stack()`：创建一个空栈，无参数，返回一个空栈。
 2. `push(item)`：向栈顶压入一个新数据项，需要一个数据项参数，无返回值。
 3. `pop()`：从栈中删除栈顶数据项，无参数，返回删除项，栈本身发生变化。
 4. `peek()`：返回栈顶数据项，但不删除。不需要参数，栈不变。
 5. `isEmpty()`：测试栈是否为空，无参数，返回布尔值。
 6. `size()`：返回栈中数据项的个数，无参数，返回值为整数。

例:

设s是一个空栈，下表是一系列的操作，栈内数据和返回值。注意栈顶在右侧。

Stack Operation	Stack Contents	Return Value
<code>s.isEmpty()</code>	<code>[]</code>	<code>True</code>
<code>s.push(4)</code>	<code>[4]</code>	
<code>s.push('dog')</code>	<code>[4, 'dog']</code>	
<code>s.peek()</code>	<code>[4, 'dog']</code>	<code>'dog'</code>
<code>s.push(True)</code>	<code>[4, 'dog', True]</code>	
<code>s.size()</code>	<code>[4, 'dog', True]</code>	<code>3</code>
<code>s.isEmpty()</code>	<code>[4, 'dog', True]</code>	<code>False</code>
<code>s.push(8.4)</code>	<code>[4, 'dog', True, 8.4]</code>	
<code>s.pop()</code>	<code>[4, 'dog', True]</code>	<code>8.4</code>
<code>s.pop()</code>	<code>[4, 'dog']</code>	<code>True</code>
<code>s.size()</code>	<code>[4, 'dog']</code>	<code>2</code>

3.3 用 Python 实现栈

前面已经定义了栈的抽象数据类型，现在我们转向栈的实现。注意：当我们说抽象数据类型的物理实现时，指的是建立数据结构。

众所周知，Python 是面向对象的程序设计语言。象栈这样的抽象数据类型可通过类来实现，而栈的操作则可看做是类的方法。另外，栈作为数据项的集合，我们可以使用 Python 中强大而简单的 List 类来实现。

Python 中的 List 类已经建立了一个数据集合机制和相应的方法。假设存在一个列表 [2,5,3,6,7,4]，我们只需约定哪一端是栈顶哪一端是栈底，就可以使用 list 中的方法（如 `append` 和 `pop`）来实现栈的相关操作。

在以下的栈实现中，我们假定 List 的右侧是栈顶。

```
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)
```

这段代码仅仅是定义了 Stack 类，运行时什么反应也没有。以下代码将创建一个栈对象，并加入操作方法。

```
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())
```

注意，我们也可以选择列表的左侧作为栈顶。这样，前面的`pop`和`append`方法就不能用了，而必须指定索引 0（列表的第一项）以便对栈内数据进行操作。如：

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

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

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

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

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

s = Stack()
s.push('hello')
s.push('true')
print(s.pop())
```

对抽象数据类型实现方式的变更，仍能保持数据的逻辑特性不变，这就是“抽象”的一个实例。两种栈的方式都能工作，但性能表现却有很大的不同。`append()`和`pop()`都是 $O(1)$ ，这意味着，不管栈内有多少数据项，第一种实现的性能是常数级的，第二种实现的 `insert(0)`和 `pop(0)`却需要 $O(n)$ 。很明显，两种实现方式在逻辑上等同，但时间复杂度却不一样。

3.4 栈的应用

3.4.1 简单括号匹配

现在我们用栈来解决一个计算机科学上的实际问题。

- 你一定写过类似这样的算术算式： $(5 + 6) * (7 + 8) / (4 + 3)$ ，这里括号为了规范操作顺序。
- 你用过 LISP 语言的话，也许写过这样的语句： `(defun square(n)(* n n))`。这条语句定义了一个函数，用于返回参数 n 的平方值。Lisp 语言以用到大量的括号而闻名。

这两个例子中，括号必须是平衡的。平衡括号的意思是，每个左括号一定对应着一个右括号，并且括号能被正确嵌套。

平衡括号

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

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

不平衡括号

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

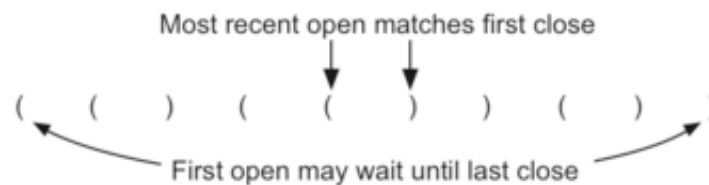
```
()))
```

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

正确地地区分平衡和不平衡括号，对很多编程语言都非常重要。

问题 1 设计一个算法，读入一串括号，并判断它们是否平衡。

仔细观察一下平衡式的结构特点可以发现：从左到右读入一串括号时，最早读到的一个右括号总是与他前面紧邻的左括号匹配；同样，最后一个右括号要与最先读到的左括号相匹配。即**右括号与左括号是反序的，并且它们从内到外一一匹配**。受此启发，我们会马上想到用栈来解决该问题。



一旦你明白栈适合保存括号，算法就简单了：

1. 创建一个空栈；
2. 从左到右读入括号串：
 - 若遇到左括号，把它压栈，说明后面需要一个右括号与之匹配。
 - 若遇到右括号，就弹出栈顶数据。
3. 只要栈内还有数据可以弹出与右括号匹配，这些括号就仍然是平衡的。任何时候，栈内没有左括号用来匹配了，这个字符串就没有平衡好。到字符串的最后，若平衡，栈应该是空的。

这个算法的 Python 代码如下：

```
1 from pythonds.basic.stack import Stack
2
3 def parChecker(symbolString):
4     s = Stack()
5     balanced = True
6     index = 0
7     while index < len(symbolString) and balanced:
8         symbol = symbolString[index]
9         if symbol == "(":
10             s.push(symbol)
11         else:
12             if s.isEmpty():
13                 balanced = False
```

```

14         else:
15             s.pop()
16
17         index = index + 1
18
19     if balanced and s.isEmpty():
20         return True
21     else:
22         return False
23
24 print(parChecker('(()))'))

```

```

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

```

```

True
False

```

3.4.2 平衡符号（通用）

对很多编程语言而言，上一节所讲的圆括号匹配只能算一个特例。不同种类的左符号和右符号的平衡与嵌套实在非常普遍。比如在 Python 中，`[]` 用于列表，`{ }` 用于字典，`()` 用于元组和算数表达式。多种符号的混合应用也需要保持相应的平衡关系。

平衡符号：既要保证左右平衡，也要求种类匹配。

```

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

```

不平衡符号

```

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

```

前面讲到的圆括号平衡算法很容易扩展到其他种类的符号中：**每个左符号被压栈，然后等匹配的右符号出现**。此时唯一的不同，就是左右匹配的同时，必须检查符号的种类也要匹配。如果发现不匹配，整个字符串就是不平衡的。最后，当整个字符串处理完毕且栈被清空时，字符串就是完全平衡的。

Python 语言的实现方法如下：与圆括号平衡相比，仅多调用一个辅助函数 `matches()`，以帮助检查符号的类型匹配。每个从栈顶弹出的元素必须检查是否与当前的右符号同一种类，如果不匹配，`balanced` 被赋值为 `False`。

```

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('[{()}]'))

```

```

True
False

```

3.4.3 十进制转换成二进制

计算机的内部数据以二进制形式存储，所有的数据都是由 0 和 1 组成的串。幸亏二进制和日常数据格式之间能够相互转换，不然计算机可一点都不好玩了。

计算机程序里，整数无处不在。数学上也有整数，当然是十进制的整数，或者说叫做以 10 为基数的整数。十进制 $(233)_{10}$ 以及对应的二进制表示 $(11101001)_2$ 分别解释为

$$2 \times 10^2 + 3 \times 10^1 + 3 \times 10^0$$

$$51 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

有一种很容易地把十进制转为二进制的方法，叫做“除二取余法”，它用栈来保存二进制的位。该算法可描述为：从一个大于 0 的整数开始，通过递归法连续除以 2，并保存除 2 得到的余数。第一次除以 2 可以判断这个数是偶数还是奇数。偶数除以 2 的余数是 0，这个二进制位就 0；奇数除以 2 的余数是 1，这个位就是 1。这样连续相除得到一串的 0 或 1，第 1 次得到的位实际是最后一位。

除二取余法的 Python 实现

```

1 from pythonds.basic.stack import Stack
2
3 def dec2bin(decNumber):

```

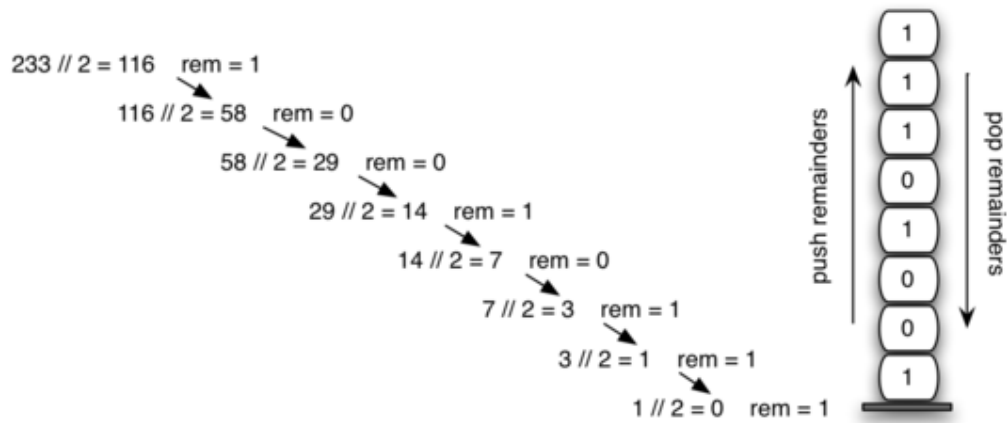


图 1: 除二取余法（又见反转，这表明可利用栈的特性来解决该问题）

```

4     remstack = Stack()
5
6     while decNumber > 0:
7         rem = decNumber % 2
8         remstack.push(rem)
9         decNumber = decNumber // 2
10
11     binString = ""
12     while not remstack.isEmpty():
13         binString = binString + str(remstack.pop())
14
15     return binString

```

```
print(divideBy2(42))
```

```
101010
```

以上算法可以很容易地扩展到任意进制的转换。计算机科学比较常用二进制、八进制和十六进制，如 $(233)_{10}$ 对应的八进制和十六进制分别为： $(351)_8$, $(E9)_{16}$ ，可表示为：

$$3 \times 8^2 + 5 \times 8^1 + 1 \times 8^0$$

$$14 \times 16^1 + 9 \times 16^0$$

将 `divideBy2()` 稍作修改便可用于转换其他进制，“除二取余法”变成“除基取余法”，新函数名为 `baseConverter`，它有两个参数：第一个是任意十进制整数，第二个是任意 2 到 16 之间的基数。余数仍被压栈，直到商为 0。从左到右的字符串生成过程也是一样的。

问题 2 当基数超过 10 以后，余数也可能超过 9，而余数是需要被压栈的，那它该如何表示？

解决办法是建立一个集合，包括一些字母符号。比如十六进制使用了 6 个字母，可建立一个字符串存储相应位置的字符，如 0 在 0 位上、1 在 1 位上、A 在 10 位上、B 在 11 位上，…。当一个余数出栈时，便可索引到该字符串上找到正确的字符并追加到答案的后面。比如 13 出栈，13 位上的 D 追加到结果中。

```
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))
```

```
11001
19
```

3.5 中缀、前缀与后缀表达式

3.5.1 基本概念

当写下算术表达式 $B * C$ 时，你马上就知道这是在计算“变量 B 乘以变量 C ”，因为 $*$ 出现在两个变量的中间。这种表达式称为**中缀表达式**，因为操作符在变量的“中间”。

而对中缀表达式 $A + B * C$ ，操作符 $+$ 和 $*$ 仍在操作数中间，但问题来了：操作符是操作哪个数？是 $+$ 操作 A 和 B 还是 $*$ 操作 B 和 C 呢？这似乎有点模糊。

事实上这种表达式我们经常见，也经常写，但从来没有怀疑过。原因是我们知道操作符的优先级。优先级高的操作符优先计算，除非用括号改变顺序。优先级顺序是乘除加减，如果两个操作符在同一级别，那就从左到右依次进行。

现在用优先级顺序来解释 $A + B * C$ 。 B 和 C 先相乘，然后 A 与乘积相加。 $(A + B) * C$ 将强制 A 和 B 先相相加，再相乘。但 $A + B + C$ 就是从左到右相加。

是的，这些对你来说太显而易见了。但是请记住，计算机需要精确地知道操作符的行为和顺序。有一种书写表达式的方法叫做“完全括号”，这种表达式把每一个操作符都加了括号，表达完全精确，也不必记忆优先级规则。

- $A + B * C + D$ 写成 $((A + (B * C)) + D)$ ，表明先算乘法，再算左边的加法；
- $A + B + C + D$ 写成 $((((A + B) + C) + D))$ ，表明加法操作从左向右结合。

$A + B$ 是操作符放在中间，如果把操作符放在操作数前面呢？变成 $+ A B$ 。放在后面呢？ $A B +$ 。是不是看起来很奇怪。这两种变型形成新的格式，叫做前缀与后缀。前缀就是操作符放在他们的操作数前面，后缀就是放在后面。看看下表会更清楚：

中缀表达式	前缀表达式	后缀表达式
$A + B$	$+ A B$	$A B +$
$A + B * C$	$+ A * B C$	$A B C * +$

例：中缀表达式转换为前缀、后缀表达式

对于中缀表达式 $A + B * C$,

- 前缀表达式为 $+ A * B C$

操作数顺序不变， $*$ 紧接在 B 和 C 之前，表示 $*$ 优先于 $+$ 。然后 $+$ 出现在 A 和乘法的结果之前。

- 后缀表达式为 $A B C * +$

操作数顺序不变，因为 $*$ 紧接在 B 和 C 之后出现，表示 $*$ 具有高优先级， $+$ 优先级低。

虽然操作符在它们各自的操作数前后移动，但是操作数的顺序相对于彼此保持完全相同。

例：中缀表达式转换为前缀、后缀表达式

考虑中缀表达式 $(A + B) * C$ ，括号在乘法之前强制执行加法。

- 写成前缀表达式时， $+$ 简单的移动到 $A B$ 之前，得 $+ A B$ 。这个操作的结果成为乘法的第一个操作数。 $*$ 移动到整个表达式的前面，得出 $* + A B C$ 。
- 写成后缀表达式时， $+$ 简单的移动到 $A B$ 之后，得 $A B +$ 。这个操作的结果成为乘法的第一个操作数。 $*$ 移动到整个表达式的后面，得出 $A B + C *$

问题 3 把这三种表达方式放在下表中对比一下，见证奇迹的时刻到了，括号去哪儿了？为什么前缀和后缀不需要括号？

中缀表达式	前缀表达式	后缀表达式
$(A + B) * C$	$* + A B C$	$A B + C *$

在前缀和后缀中，操作符和他们的操作数之间关系清晰，他们的位置就说明了计算顺序，不需要象中缀那样，额外用括号来帮助分辨。因此，在很多情况下，中缀是最不好用的表达式。下表提供了更多的对比例子，请仔细对比它们是怎样安排位置来保证计算正确的。

中缀表达式	前缀表达式	后缀表达式
$A + B * C + D$	$+ + A * B C D$	$A B C * + D +$
$(A + B) * (C + D)$	$* + A B + C D$	$A B + C D + *$
$A + B + C + D$	$+ + + A B C D$	$A B + C + D +$

3.5.2 中缀表达式转换前缀表达式和后缀表达式

至此，我们都是手动地把中缀转换为前缀和后缀。也许应该有一种算法，能够转换任意复杂的表达式。

回顾一下前面提到的“完全括号”，如 $A + B * C$ 写成 $(A + (B * C))$ 以保证乘法的高优先级。仔细观察可以发现，每一对括号内都是一个计算过程，包括一对操作数和一个操作符的完整计算。

看子表达式 $(B * C)$ ，如果把乘号移至右括号的位置，再去掉相应的左括号，就变成了 $B C *$ ，亦即 $(B * C)$ 的后缀式。更进一步，把加号移至其相应的右括号位置并取而代之，再去掉相应的左括号，整个后缀表达式就出来了，如图2：



图 2: 中缀转后缀

如果换个方向，操作符左移取代左括号并去掉右括号，就得到前缀表达式。看来括号的位置，是找到操作符位置的线索，见图3

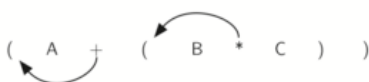


图 3: 中缀转前缀

总之，要转换表达式，无论多么复杂，无论前缀还是后缀，可以先完全括号化，然后将操作符前移或后移取代括号。

这是一个更复杂的转换例子： $(A + B) * C - (D - E) * (F + G)$ ，图4展示了转换过程。

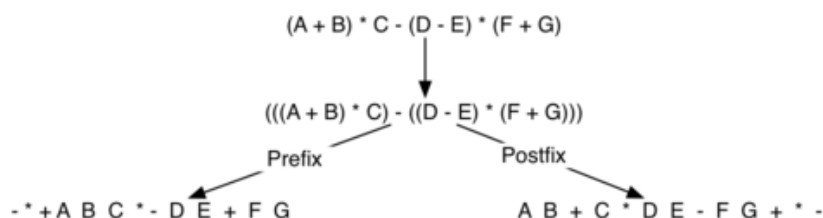


图 4: 中缀转前缀与后缀

3.5.3 中缀转后缀通用法

现在我们要开发一个算法，把任何中缀表达式转换为后缀表达式。为此，我们先来仔细看看其转换过程。

仍然考虑中缀表达式 $A + B * C$ ，其等价的后缀表达式是 $A B C * +$ ，注意操作数 A 、 B 和 C 保留了原来的相对位置，只有操作符改变了位置。在中缀表达式中，从左到右第一个出现的是 $+$ ，而在后缀式中， $+$ 最后出现，因为 $*$ 的优先级高于 $+$ 。也就是说，操作符在中缀表达式中的顺序和后缀表达式中相反。

在处理表达式的时候，操作符应该先保存在某处，因为操作符读进来的时候，它右边的操作数还没到。另外因为优先级的关系，保存的顺序要反转。就象上面说的乘号和加号一样，加号先出现，但因为乘法优先，加法先来也得靠后站。因为顺序反转的关系，考虑使用栈来保存操作符。

象 $(A + B) * C$ 怎么办呢？其后缀表达式是 $A B + C *$ 。从左到右的顺序，先读到了 $+$ ，但是当读到 $*$ 的时候， $+$ 已经找好位置，因为括号的优先级高于 $*$ 。上一段的规则遇到了新问题。这里就要考虑有括号的时候怎么办。当读到左括号的时候，我们把左括号作为操作符保存起来，标志着一个高优先级的操作就要到了，直到匹配的右括号出现，左括号才能出栈。

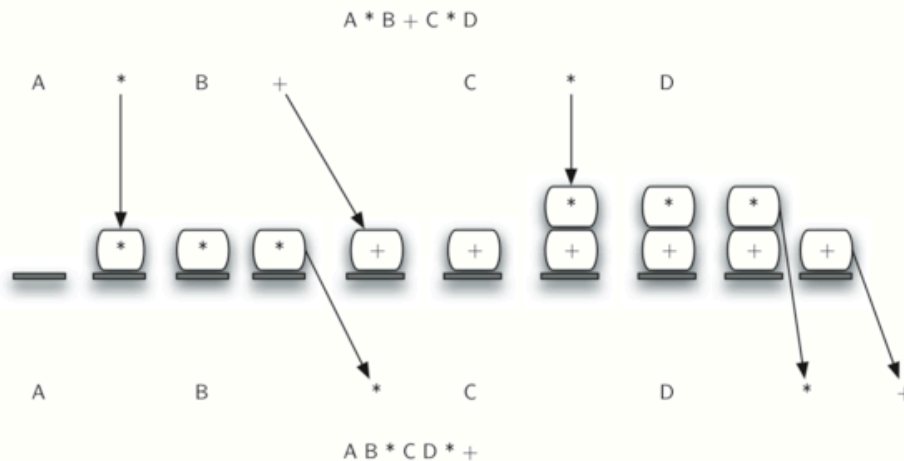
扫描中缀式的时候，要用一个栈来保存操作符，因栈的特性提供了反转功能。栈顶项总是我们最近一次压栈的操作符。每当读到一个新操作符，总要与栈顶的符号比较一下优先级。

设中缀表达式是由操作符和操作数组成的字符串，其中操作符包括 $*$ 、 $/$ 、 $+$ 、 $-$ ，还有 $($ 、 $)$ ，而操作数则包括字母或数字。按以下操作，可将中缀表达式转换成后缀表达式。

1. 创建一个空栈 (opstack) 以保存运算符，创建一个空列表 (postfixList) 以保存输出项。
2. 利用字符串的 split 方法将中缀表达式转换为列表。
3. 从左到右扫描列表，对于每个元素：
 - 如果是操作数，追加到输出列表。
 - 如果是 $($ ，压栈到 opstack。
 - 如果是 $)$ ，循环出栈，直到左括号出栈。此前出栈的元素追加到输出列表。
 - 如果是运算符 ($*$ 、 $/$ 、 $+$ 或 $-$)，先把栈内优先级大于当前操作符的项目全部出栈并追加到输出列表，然后把当前操作符压栈。
4. 当输入列表检索完成时，检查栈，把剩下的元素全部出栈并加到输出列表尾部。

例：

将 ' $A * B + C * D$ ' 转换为后缀表达式



以下是该算法的 Python 实现，其中使用了一个字典 (prec) 保存操作符的优先级，每个操作符映射一个整数以便作优先级的比较。注意左括号被定义为最低的优先级 1，这样每个操作符与之比较时，都会高于它。

```
1 from pythonds.basic.stack import Stack
2
3 def infix2Postfix(infixexpr):
4     prec = {}
```

```

5     prec["*"] = 3
6     prec["/"] = 3
7     prec["+"] = 2
8     prec["-"] = 2
9     prec["("] = 1
10    opStack = Stack()
11    postfixList = []
12    tokenList = infixexpr.split()
13
14    for token in tokenList:
15        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
16            postfixList.append(token)
17        elif token == '(':
18            opStack.push(token)
19        elif token == ')':
20            topToken = opStack.pop()
21            while topToken != '(':
22                postfixList.append(topToken)
23                topToken = opStack.pop()
24        else:
25            while (not opStack.isEmpty()) and \
26                (prec[opStack.peek()] >= prec[token]):
27                postfixList.append(opStack.pop())
28            opStack.push(token)
29
30    while not opStack.isEmpty():
31        postfixList.append(opStack.pop())
32    return " ".join(postfixList)

```

```

print(infix2Postfix("A * B + C * D"))
print(infix2Postfix("( A + B ) * C - ( D - E ) * ( F + G )"))
print(infix2Postfix("A + B * C"))

```

```

A B * C D * +
A B + C * D E - F G + * -
A B C * +

```

3.5.4 后缀表达式求值

这一节将设计一个算法，以计算一个后缀表达式的值。无论何时看到输入一个操作符，最近的两个操作数就是操作对象。为了说清楚一点，我们来看两个例子：

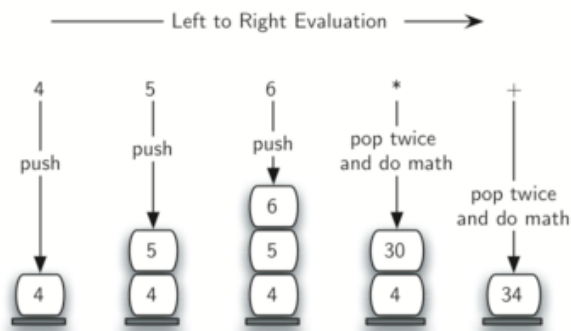
例：

考虑后缀表达式 `4 5 6 * +`。从左到右扫描时，首先得到 4 和 5，不过此时，并不知道该如何处理这两个数，直到看到后面的操作符。所以要把这两个数先压栈，得到操作符以后再出栈。

该例中，下一个符号仍然是操作数，所以照旧压栈，并检查下一个。现在看到操作符 `*`，这意味着

最近两个操作数要用来做乘法。出栈两次，得到两个操作数并相乘（在本例中是结果是 30）。

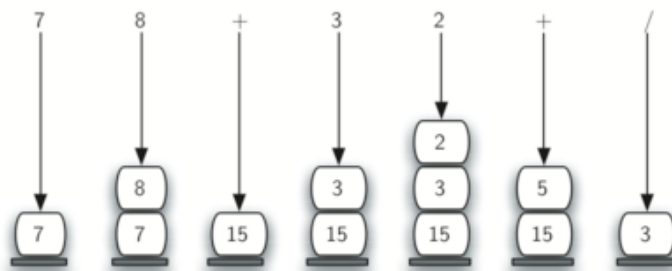
这个计算结果要压回到栈内，并作为下一个操作符的对象。当最后一个操作符工作结束，栈内应该只有一个数值，出栈并作为计算结果返回。图?? 显示了该求值过程中栈内容的变化。



例:

再来看一个稍微复杂的表达式: $7\ 8\ +\ 3\ 2\ +\ /\$ 。这里有两点要注意:

- 栈的大小，随着子表达式的计算过程而膨胀，收缩，再膨胀。
- 除法操作符要小心处理，因为后缀表达式的操作数顺序不变，但当两个操作数出栈时，顺序反了。因为除法不支持交换律，所以 $15/5$ 与 $5/15$ 不同，必须保证顺序没有交错。



算法假定后缀表达式是一系列被空格分隔的字符，操作符是 $*/+-$ ，操作数假定是一位整数。最终结果也是整数。

1. 建立一个空栈 operandStack
2. 字符串使用 split 转为列表
3. 从左到右检索列表，对于每个元素，
 - 如果是操作数，字符转为整数，压栈
 - 如果是操作符，出栈两次。第一次出栈的是第二个操作数，第二次出栈的是第一个操作数。计算结果，并压回栈。
4. 检索结束，出栈结果就是返回值。

完整的函数代码如下，其中的 doMath 是算法辅助函数，定义为两个操作数和一个操作符的计算。


```

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 + /'))

```

4 队列

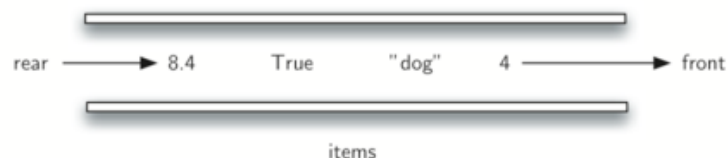
4.1 什么是队列

队列是有序数据集合，队列的特点，是在头部删除数据项，称为前端，在尾部增加数据项，称为后端。数据项总是在开始的时候排在队伍的后端，慢慢向前走，直到排到最前面，轮到它的时候离开队列。

刚进来的排在后端，待在队伍里时间最长的在前端，这种排列规则叫做 FIFO，意思是“先进先出”，或者叫做“先来先服务”。

例：

最简单的例子就是平时我们的排队，象排队买票看电影，在超市排队付款，在自助餐厅排队取盘子（嗯，盘子可是后进先出的，那是栈规则）。队列严格执行一字排开的规则，一个方向进，同一方向出，不许插队，不许离队。下图是个 Python 数据对象的队列。



例:

计算机科学里也有队列的例子，象我们实验室有 30 台电脑只有 1 台打印机，学生们要打印的时候，所有的打印任务排队等候，排在第一的马上就能打印，排在最后的就要等所有其他人都打完了才开始。随后我们会探讨这个很有意思的例子。

例:

除打印队列外，操作系统使用了不同的队列控制系统进程。象调度系统就是使用了队列算法以保证尽可能快地执行程序，并响应尽可能多的用户。比如有时候打字时发现敲了键盘，屏幕却延迟响应，这是因为系统系统正做其他事情，所以把键盘事件放在缓冲队列里，所以稍有延迟，不过最终还是会显示出来。

4.2 The Queue Abstract Data Type

队列的抽象数据类型由下面的操作定义。队列是结构化的、有序的数据集，前端删除数据，后端加入数据，保持 FIFO 属性：

1. Queue(): 定义一个空队列，无参数，返回值是空队列。
2. enqueue(item): 在队列尾部加入一个数据项，参数是数据项，无返回值。
3. dequeue(): 删除队列头部的数据项，不需要参数，返回值是被删除的数据，队列本身有变化。
4. isEmpty(): 检测队列是否为空。无参数，返回布尔值。
5. size(): 返回队列数据项的数量。无参数，返回一个整数。

举例说明，q 是一个刚创建的空队列，表 1 分别显示了操作、表内数据和返回值。4 是第一个加入队列的，所以也是第一个出队的。

4.3 Implementing a Queue in Python

有了队列的抽象数据类型，我们可以创建一个类来实现队列。和以前一样，我们采用 Python 列表来创建队列类。

队列也是有序的，所以需要决定队列的哪一头作为队列的前端和尾端。在下面的实现代码中，我们约定列表的 0 位置是队列的尾部，这样的好处是，可以直接使用列表的 insert 方法在队尾加入数据，使用 pop 方法在队列的前端（这时是列表的最后一个数据）删除数据。从性能上分析，这意味着 enqueue 是 $O(n)$ ，而出队是 $O(1)$ 。

```
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)

```

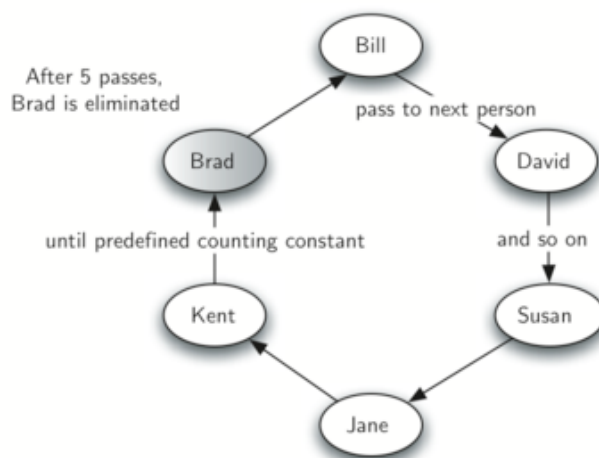
```

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

```

4.4 队列任务：烫手山芋

为了展示队列的应用，我们模拟一种真实的先进先出的情形。作为开始，我们观察一种儿童游戏，叫烫手的山芋（hotpotato），在这个游戏中，孩子们排成一圈，把手里的东西一个传一个。在某种情形下，停止传递，手上拿着烫手的山芋的人就要被请出来，其他的人继续玩，直到只剩一个人。

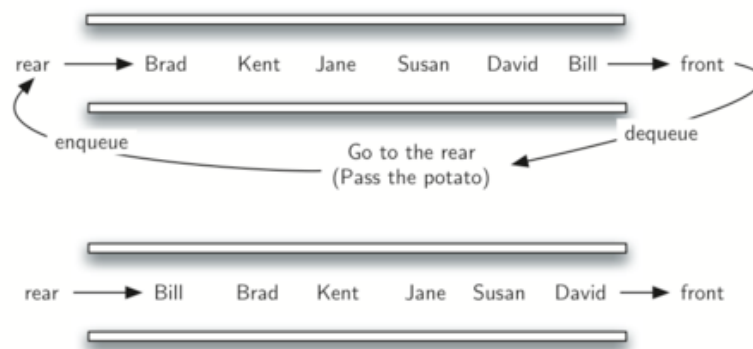


从现代意义上说，这个游戏等价于著名的约瑟夫问题。据说，一世纪左右，历史学家弗拉维·约瑟夫

与犹太人一起反抗罗马。一次，约瑟夫和他的 39 个同志一起在山洞里抵抗，不过眼看就要失败了，他们决定宁死也不做罗马的奴隶。他们围坐成一圈，一个人一个编号，按顺时针方向，每第七个人就要被杀死。据说约瑟夫是个数学家，他马上就on知道按这规则，应该坐在哪个位置会留到最后。看来约瑟夫最后没有自杀，相反却投降了。这个故事有很多版本，有的版本说是每 3 个人杀一个，有的说最后一个人可以骑马逃脱，但不管怎样，思想是相同的。

我们引入一个烫手的山芋的模拟过程，参数是一个名字列表和一个常数 num。num 用来计数，最后函数返回经多轮计数后，剩下的最后一个人的名字。后来发生什么，就看你的了。

为了模拟这个圆圈，我们使用队列（图 3）。假定开始拿着山芋的孩子站在队伍的前端，一经传出山芋后，模拟程序只需要简单地把这个孩子移出队列，然后再将他加入尾部，然后他在尾部再逐步前移，直到再次轮到他。经过 num 次出队入队之后，前端的孩子最终被完全清出队列，然后剩余的人继续游戏，直到最后一个。



```
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()

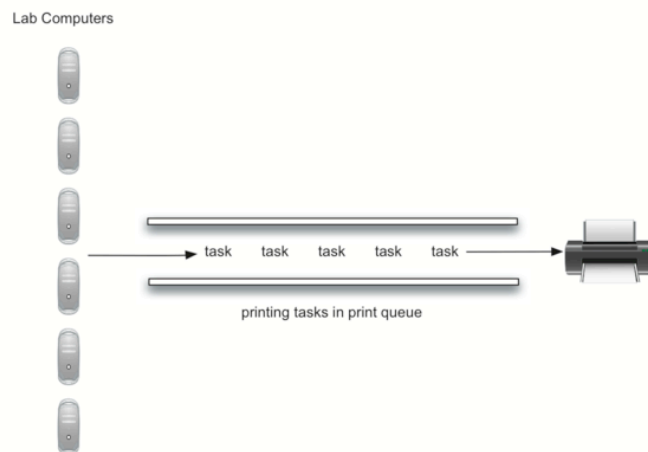
print(hotPotato(["Bill", "David", "Susan", "Jane", "Kent", "Brad"], 7))
```

4.5 队列应用: 打印任务

在本章开始我们就谈到过模拟打印任务队列的行为。回想学生时代，向共享打印机发送一个打印任务，并被放在一个先到先服务的队伍中等候处理。这种方法有很多的问题，最重要的是，打印机是否能够完成这么多数量的任务，如果打不完，学生们会等待太长时间，导致错过下次课。

考虑一下这种情形：计算机科学实验室里，平均每小时有 10 个学生在完成作业，这些学生在这段时间里一般打印两次，每次任务可能是 1-20 页不等。实验室的打印机有点老，按草稿质量能够打印 10 页/分钟，如果切换到高质量，则只能打印 5 页/分钟，越慢等的时候越长，那么应该设置成多少？

我们可以通过建立一个实验室模型帮助决策。需要建立学生、打印任务和打印机的模型。因为学生发出打印任务时，需要把任务加入打印任务队列。当打印机完成一个任务后，它要到队列中查看是否还有任务要处理。我们感兴趣的是，学生们的平均等待时间，也等同于队列中任务的平均等待时间。



这个模型需要一点概率知识。例如，学生打印长度 1-20 页，如果每个长度可能性相等，那么实际长度可以用一个 1-20 之间的随机数来模拟。这表示，1-20 之间的长度机会均等。

如果 10 个学生每人打印两次，那么平均每小时有 20 个打印任务。在每一秒钟产生一个打印任务的可能性多大？这就要考虑任务与时间的比率。20 个任务每小时，意味着平均每 180 秒产生一个任务。

$$20\text{任务/小时} \times 1\text{小时}/60\text{分钟} \times 1\text{分钟}/60\text{秒} = 1\text{任务}/180\text{秒}$$

我们可以通过产生一个 1-180 之间的随机数，来模拟每秒钟产生一个新任务的概率。如果数字是 180，那么任务已经产生了。要注意很多任务可能排成队，也有可能等好久也没有一个任务，这就是模拟的特性，我们的模拟总是想尽可能地接近真实情况。

4.6 主要模拟步骤

主要分这几个步骤。

1. 建立打印任务队列，每个任务要打上到达的时间戳，开始时是空的队列。

2. 对每一秒 (currentSecond)

(a) 有没有新任务到达？如果有，如果有，加入队列，加上 (currentSecond) 时间戳。

(b) 如果打印机空闲且有任务在等待：

i. 从队列中除去下一个任务，并分配给打印机。

ii. 当前时间减去加入队列的时间戳，就是本任务的等待时间

iii. 把等待时间加入一个列表，后续处理。

iv. 根据本任务的纸张数，计算打印所需时间

(c) 如果需要，打印机工作 1 秒钟，也要从任务所需时间中减去 1 秒钟

(d) 如果打印完成，也就是上一步中把时间减到 0 了，打印机进入空闲状态

3. 模拟完成，根据任务时间列表，计算平均等待时间

4.7 Python 实现

为了实现任务模拟，我们设计三个类对应现实世界的三种对象：Printer, Task, 和 PrintQueue

Printer 类用来跟踪当前是否在工作，如果是，它的状态是 busy(13-17)，并且 busy 时间用纸张数计算出来。构造函数可以设置每分钟打印页数。Tick 方法递减内部时间，并将打印机设为 idle.

```
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 类代表一个打印任务。产生任务时，构造函数提供一个 1-20 之间的随机数，代表任务的纸张数。要用到 random 模块的 randrange 方法.

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

每个任务也要保存一个时间戳用来计算等待时间。这个时间戳代表产生任务并加入队列的时间，waitTime 方法用来计算任务在开始打印之前等待的时间。

```
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
```

模拟程序 (Listing 4) 用来实现上面所述的算法。printQueue 对象是队列的抽象数据类型的实例，一个布尔值辅助函数，newPrintTask 用来决定是否新任务产生，我们再次使用了 random 模块 randrange 方法返回一个 1-180 之间的随机数。打印任务每 180 秒到来一次，我们从一个范围内的随机整数中产生 180 的方法模拟随机事件 (32 行)。这个模拟功能允许我们设置总时间和打印机的打印速度。

```
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 10 次以后，我们能看得出平均等待时间是 122.155 秒，也能看到平均数据变化的范围很大，从最小的 17.27 秒到最大的 239.61 秒，而且只有 2 次的打印任务是完成的。

现在我们调整打印速度到 10ppm，再运行 10 次，通过更快的打印机任务，我们希望在一个小时的时间里，有更多的打印任务得以完成。

```
>>>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.
```

4.8 讨论

这次模拟的目的，是要问题这样一个问题，即通过降低打印速度提高打印^④，这台打印机是否能够承担这个打印负荷。这个过程就是编写一个模拟程序，为打印任务建模，并以随机的发生时间和随机的工作长度进行验证。

以上的输出表明，如果在 5ppm 的速度，平均等待时间是从 17 秒到 376 秒 (约 6 分钟)，如果用速度加快，最小时间变为 1 秒到 28 秒。另外在 5ppm 的速度下，8 成的打印任务在 1 个小时内没有打完。

这样，我们会给出建议说，提高质量降低速度，不是一个好办法。学生们不能等那么长时间来等待，特别是下一节还有课，6 分钟的等待太长了。

这种模拟分析能够回答很多问题，特别是这类“如果怎样，那么怎样”类的问题，因为我们只要修改模拟程序的参数，就能模拟很多有趣的行为。比如

- 如果人数增多，现在有 20 个学生，是什么情形？
- 如果是周六，学生们不必去上课，他们会等待吗？
- 因为 python 语言太强大了，程序都变短，所以学生们的打印任务都变小了，会怎么样？

这些问题都可能修改模拟参数来得到答案，当然，必须明白，模拟程序只是使用可能真实假定数据。现实中的打印任务长度和每小时内要打印的学生数，也需要构建一个模拟程序。