致 谢

2015 地空数算课程教材第二章翻译小组

组长

苏瑞冰

成员

蒋久阳、张虎来、胡哲、朱贺、宋欣源、葛天雨、钟涛、周杰、
黄佳旺、付帆飞、张沙洲

本文为第二章到第三章 3.3

2. 算法分析

1.1. 目标

- 了解为何算法分析非常重要；
- 能够采用“大 O”方法来描述算法执行时间；
- 了解在 Python 列表和字典类型中通用操作执行时间的“大 O”级别；
- 了解 Python 数据类型的具体实现对算法分析的影响；
- 了解如何对简单 Python 程序进行执行时间检测。

1.2. 什么是算法分析

计算机科学初学者经常拿自己的程序和其他人的进行比较。你也许也注意到很多的电脑程序看起来相似，特别是一些简单的程序。这时我们经常想到一个有趣的问题，当两个不同的程序可以解决相同的问题时，是不是其中一个优于另一个？

为了回答这个问题，我们需要记住的是，程序和它所代表的基本算法有着重要区别。在第 1 章中我们说到，算法是对问题解决的分步描述。这是一种能解决任何问题实例的方法，比如给定一个特定的输入，算法产生期望的结果。从另一方面看，程序则是采用某种编程语言实现的算法，同一个算法通过不同的程序员采用不同的编程语言，能产生很多程序。

进一步探究这种差异，我们可以思考 ActiveCode 1 中展现的函数。这个函数解决了一个大家熟知的问题，完成从 1 到 n 的累加。其中的算法使用了一个初始值为零的累加变量的概念。之后的解决方案是遍历 n 个整数，逐个累加到累计变量。

```
def sumOfN(n):
  theSum = 0
  for i in range(1,n+1):
     theSum = theSum + i

  return theSum

print(sumOfN(10))
```

代码 15 前 n 个正整数求和 (active1)

现在再看 ActiveCode 2 中的函数。可能第一眼看上去比较奇怪，但是进一步观察你会发现，这个函数所实现的功能与之前那个函数相同。这很明显是因为糟糕的编码。我们没有使用好的变量命名来增加可读性，而且中间使用了毫无作用的垃圾代码。

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

   return fred

print(foo(10))
```

代码 16 另一种前 n 个正整数求和 (active2)

回到前面我们提出的问题：是否一个程序会优于另外一个？答案取决于你自己的标准。如果你关心的是可读性，那么 sumOfN 函数当然比 foo 函数更好。实际上，你可能见过很多这样的例子在你的编程入门课程上，因为学习课程的其中一个目的，就是帮助你编写更具可读性的程序。然而，在这门课程中，我们主要感兴趣的是算法本身特性。（我们当然希望你可以继续努力写出更具可读性的代码。）

算法分析主要就是从计算资源消耗的角度来评判和比较算法。我们想要分析两种算法并且指出哪种更好，主要考虑的是哪一种可以更高效利用计算资源，或者更少占用资源。从这个角度，上述两段程序实际上是基本相同的，它们都采用了一样的算法来解决累计求和问题。

从这点上看，思考关于计算资源的真正意义是很重要的。我们可以通过两种方式看待它。一种是考虑算法解决问题过程中需要的存储空间或内存。问题解决方案所需的存储空间通常是由问题本身情况影响的，这是经常会有的现象，其中还包含算法本身的特定的空间需求，在这种情况下，我们会很小心地解释这种变化。

对于另一种空间需求，我们可以基于算法运行所需的时间来对它们进行分析和比较。这种方法有时会被称为“执行时间”或“运行时间”的算法。我们可以检测 sumOfN 函数运行时间的一种方式就是做基准分析。这意味着我们将跟踪程序运行出结果，从而得到所需实际时间。在 Python 中，我们可以用基准问题测试一个函数，通过对我们使用的系统标定起始时间和结束时间。在 time 模块中有一个叫做 time 的函数，可以在某些任意的起点获取以秒为单位的系统当前时间。通过在开始和结束时两次调用这个函数，然后计算两次时间之差，我们就可以得到精确到秒（大多数情况为分数）的运行时间。

列表 1

```
import time

def sumOfN2(n):
  start = time.time()

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

```
  end = time.time()

  return theSum,end-start
```

我在列表 1 中展示了在求和前后嵌入时间调用的原始 sumOfN 函数。函数返回一个元组，包括累计和及计算所需时间（以秒为单位）。如果我们连续运行这个函数 5 次，每次都完成 1 到 10,000 的累加，我们得到的结果如下：

```
>>>for i in range(5):
      print("Sum is %d required %10.7f seconds"%sumOfN(10000))
Sum is 50005000 required  0.0018950 seconds
Sum is 50005000 required  0.0018620 seconds
Sum is 50005000 required  0.0019171 seconds
Sum is 50005000 required  0.0019162 seconds
Sum is 50005000 required  0.0019360 seconds
```

我们发现这个时间是相当一致的，并且平均使用 0.0019 秒去执行这段代码。那么如果我们累加到 100,000 会怎样呢？

```
>>>for i in range(5):
      print("Sum is %d required %10.7f seconds"%sumOfN(100000))
Sum is 5000050000 required  0.0199420 seconds
Sum is 5000050000 required  0.0180972 seconds
Sum is 5000050000 required  0.0194821 seconds
Sum is 5000050000 required  0.0178988 seconds
Sum is 5000050000 required  0.0188949 seconds
>>>
```

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

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

在这种情况下，平均时间再次约为之前的 10 倍。

现在考虑 ActiveCode 3，它展现了一种解决求和问题的不同方法。这个函数，sumOfN3，利用了一个封闭方程∑ni=1i=(n)(n+1)2 去完成无迭代的累加到 n 的计算。

```
def sumOfN3(n):
  return (n*(n+1))/2

print(sumOfN3(10))
```

代码 17 无迭代求和 (active3)

如果我们对 sumOfN3 做同样的基准检测，给 n 赋五个不同的值（10,000, 100,000, 1,000,000, 10,000,000, 和 100,000,000），我们得到的结果如下：

```
Sum is 50005000 required 0.00000095 seconds
Sum is 5000050000 required 0.00000191 seconds
Sum is 500000500000 required 0.00000095 seconds
```

```
Sum is 50000005000000 required 0.00000095 seconds
Sum is 5000000050000000 required 0.00000119 seconds
```

对于输出我们需要关注两个关键点。第一，这种算法的运行时间比之前任何的例子都短很多。第二，不管 n 的取值多少运行时间都非常一致。看上去 sumOfN3 的运行时间几乎不受需要累计的数目影响。

但是这个基准测试真正告诉了我们什么？直观上，我们可以看到迭代算法似乎做了更多工作，因为一些程序步骤被重复执行。这可能就是它需要更长时间的原因。此外，迭代算法所需要的运行时间似乎会随着 n 值的增大而增大。然而，这里有一个问题。如果我们在不同的计算机上运行相同的函数，或者使用不同的编程语言，我们可能会得到不同的结果。如果计算机比较老旧，运行 sumOfN3 可能还会需要更长的时间。

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

1.1.1.“大 O”表示法

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

当我们试图用一个好的基本计量单位去计算比较上节中几个求和算法时，这个基本计量单位应被用作统计执行 sum（求和）的赋值语句的次数。 在函数 sumOfN 中，赋值语句的数量是 1（theSum=0）加上 n（我们执行 theSum=theSum+i 这条语句的次数）。我们可以把赋值语句的数量表示为函数 T，上例中 T(n)=1+n。其中变量 n 代表问题规模，我们可以看作：“T(n)是解决规模为 n 的问题所需的时间，即 n+1 步。”

在上节给出的求和函数中，用加法的总项数来表示问题规模是合适的。因此有求前 100000 个整数和比求前 1000 个整数和问题规模要大。由此可见，解决大规模问题所需的时间比小规模问题要长。我们的目标是要找出问题规模会怎么影响一个算法的执行时间。

然而计算机科学家把这个算法分析技术想的更深。其实精确的操作次数并不如 T(n)函数中起主导作用的那部分重要。换言之，当问题变大时，T(n)函数中的某些部分会盖过其他部分的贡献。这个主导部分将被用作最后的比较。数量级函数描述了 T(n)中随着 n 增加而增加速度最快的部分，称作“大 O”表示法，记作 O(f(n))，其中 f(n)表示 T(n)中的主导部分。数量级函数提供了计算过程中实用的真实步骤数的近似值。

上例 T(n)=n+1 中，当 n 越来越大时，常数 1 对最终结果的影响会越来越小。如果我们只需要 T(n)的近似值，我们不妨忽略 1，就把 O(n)当作算法执行时间。值得注意的是，常数 1 对于 T(n)是很重要的，然而当 n 增大时，忽略 1 的近似值也是很准确的。

再举一例，假设有这样一个算法，真实步骤数是 T(n)= 5+27n+1005。当 n 比较小时，如 n=1 或 n=2，常数 1005 起决定性作用。然而，当 n 越来越大时，项就越来越重要，其它两项对结果的影响则越来越小。同样，项中的系数 5，对于的增长速度来说也影响不大，所以可以在数量级中去掉 27n+1005，以及系数 5 的部分，确定为 O()。

有时算法的执行时间不仅仅依赖于问题规模，还取决于某些具体数据，尽管在上文求和那个例子中难以发现这点。对于这些算法，我们应把它的执行情况分为最好、最差和平均情况。一个特定的数据集会使

算法执行的极为不佳，这是最坏的情况。但同一个算法在另一个数据集又会执行的非常流畅。然而，大多数时候，算法执行情况在这两个极端之间。这样的平均状况体现了算法的主流性能。因此对算法的分析要看主流，而不被某几种特定的运行状况所迷惑。

随着算法学习的深入，如表 1 所示的一些常见的数量级函数会不断的出现。为了判断这些函数中哪一个在 T(n)中占主导地位，就要比较当 n 增大时谁更大。

表 1

函数 f(n)	名称
1	常数
logn	对数
n	线性
nlogn	对数线性
n2	平方
n3	立方
2n	指数

表格 11 常见的大 O 数量级函数

图 1 所示是表 1 中函数的图像。当 n 比较小时，函数之间在图像上的区分不是很大，难以确定谁占主导。当 n 增长到较大时，函数之间在图像上的区分很明显，容易看出其主要变化量级。

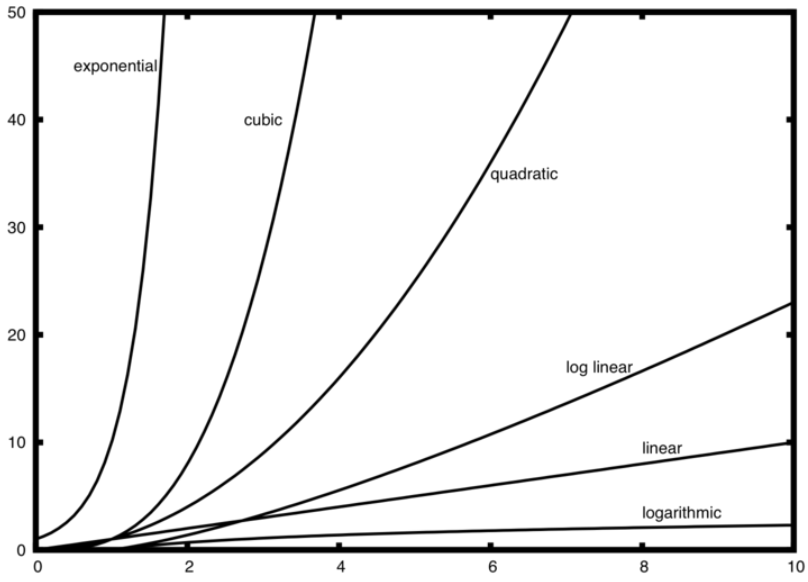

图 13 常见大 O 数量级函数的图像

最后举一例，如 listing2 所示的是一个 Python 代码片段。尽管这个程序并没有做什么事情，但对我们着手分析代码的执行情况还是很有用的。

Listing 2

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

不难看出，执行操作总数分为四项。第一项是常数 3，代表程序开始时的 3 个赋值语句。第二项是 $3n^2$, 因为根据嵌套迭代原理三个赋值语句分别执行了 n^2 次。第三项 2n 是有两个赋值语句重复执行了 n 次。最后，第四项是常数 1，代表最后一个赋值语句。这样我们得到 $T(n)=3+3n^2+2n+1=3n^2+2n+4$。再看指数项，会发现 n^2 项占主导地位，因此这段程序的数量级为 $O(n^2)$。注意当 n 增大时其他各项和主导项的系数都可以被忽略。

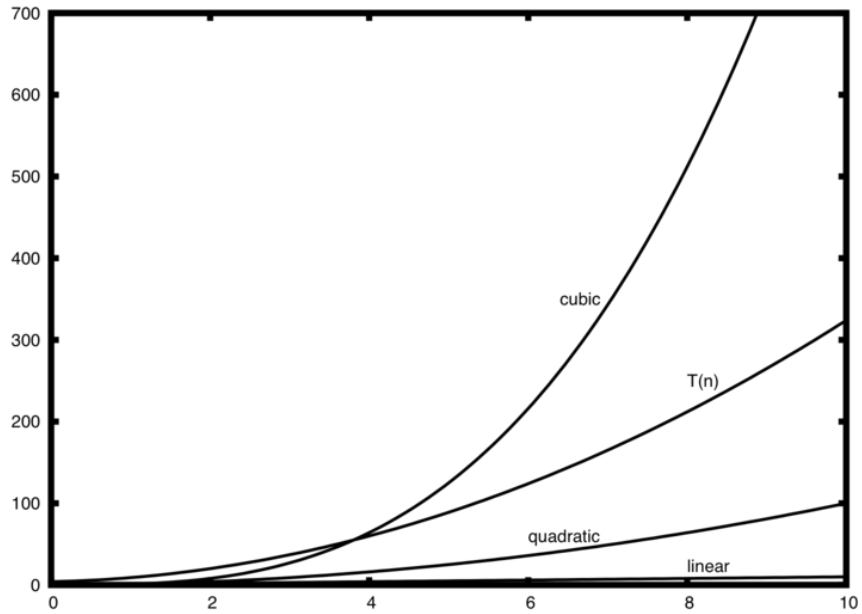

图 14 比较 T(n)与常见大 O 数量级函数

图 2 所示是 T(n)与一些常见大 O 数量级函数的比较。可以看出，一开始 T(n)比立方函数要大。但是当 n 变大时，立方函数很快超过了 T(n)。不难看出，当 n 继续增长时，T(n)朝平方函数趋近。

自我检测

写两个 Python 函数，功能为找到列表中最小数。要求函数 1 将每个数与其它所有数比较，运行时间数量级为 $O(n^2)$，函数 2 数量级为 O(n)。

1.1.2. 例子：“变位词”判断

一个展示不同数量级算法的好例子是典型的“变位词”判断问题。我们称两个字符串为变位词，如果第二个字符串是原字符串所有字符的重新排列。例如，'heart'和'earth'是变位词，'python'和'typhon'也是变位词。为了简单起见，假设问题中的两个字符串长度相同并完全由 26 个小写字母组成。我们的目标是写一个布尔函数，以两个词作为参数，返回真假，表示这两个词是否变位词

1.1.1.1. 解法 1：检查标记 Checking Off

第一个解决变位词问题的方法是检查第一个字符串中的每个字符是否在第二字符串中出现。如果每一个字符都能在两个字符串中相匹配标记，那么这两个字符串就是变位词。我们标记一个字符的方法是将其用 None 来替换。然而由于 Python 中字符串是不可变量，第一步应是将第二个字符串转换为一个列表。第一个字符串中的每个字符和第二个列表中字符进行检查，如果找到了就用 None 替换来标记。ActiveCode 1 展示了这个函数。

```
def anagramSolution1(s1,s2):
    alist = list(s2)

    pos1 = 0
    stillOK = True

    while pos1 < len(s1) and stillOK:
        pos2 = 0
        found = False
        while pos2 < len(alist) and not found:
            if s1[pos1] == alist[pos2]:
                found = True
            else:
                pos2 = pos2 + 1

        if found:
            alist[pos2] = None
        else:
            stillOK = False

        pos1 = pos1 + 1

    return stillOK

print(anagramSolution1('abcd','dcba'))
```

代码 18 Checking Off (active5)

为了分析这个算法，我们要注意到 s1 中 n 个字符的每一个都会引起一个最多迭代到 s2 列表中第 n 个字符的循环。列表中的 n 个位置各会被寻找到一次去匹配 s1 中的某一个字符。那么执行的总数就是从 1 到 n 的代数和。我们之前提到过它可以这样表示

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

当 n 变大时，n^2 相对于 n 将占主要部分，并且 1/2 可以忽略。因此，这个算法的数量级是 $O(n^2)$。

1.1.1.2. 解法 2：排序比较

另外一个方法则利用了尽管 s1 和 s2 并不相同，但若为变位词他们一定包含完全一样的字符的道理。因此，我们首先从 a 到 z 给每一个字符串进行排序，如果它们是变位词，那么我们将得到两个完全一样的字符串。ActiveCode 2 展示了这个方法。同样，我们可以先将每一个字符串转换成列表，然后利用 Python 中内建的列表排序方法进行排序。

```
def anagramSolution2(s1,s2):
    alist1 = list(s1)
    alist2 = list(s2)
```

```
    alist1.sort()
    alist2.sort()

    pos = 0
    matches = True

    while pos < len(s1) and matches:
        if alist1[pos]==alist2[pos]:
            pos = pos + 1
        else:
            matches = False

    return matches

print(anagramSolution2('abcde','edcba'))
```

代码 19 Sort and Compare (active6)

第一眼看你可能倾向于认为这个算法的复杂度是 O(n)，因为排序后只需要一个简单的循环去比较 n 个字符。然而那两个 Python 中内建的排序并不是没有任何消耗的。正如我们将要在后来的章节中看到的，排序的复杂度通常是 $O(n^2)$ 或者 $O(n\log_n)$，所以排序贡献了主要的循环操作。最终，这个算法和排序的复杂度数量级相同。

1.1.1.3. 解法 3：暴力

解决这个问题的的典型暴力方法是尝试所有的可能。为了解决变位词检测问题，我们可以简单的构造一个由 s1 中所有字符组成的所有可能的字符串的列表，并检查是否和 s2 相同。然而这个方法有一点困难之处。当我们构造由 s1 中字符组成的所有可能字符串时，第一个字符有 n 个可能，第二个字符有 n-1 种可能，第三个则是 n-2 种，以此类推。所有可能字符串的总数是 n∗(n−1)∗(n−2)∗...∗3∗2∗1，也就是 n!。尽管这些字符串中的一些可能是重复的，但程序不能提前预见到所以还是会产生 n!个字符串。

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

1.1.1.4. 解法 4：计数比较

解决变位词问题的最后一个方法是利用任何两个变位词都有相同数量的 a,相同数量的 b，相同数量 c 等等。为了判断两个字符串是否为变位词，我们首先计算每一个字符出现的次数。由于总共有 26 个可能的字符，我们可以利用一个有 26 个计数器的列表，每一个计数器对应一个字符。每当我们看到一个字符，我们就在相对应的计数器上加一。最终，如果这两个计数器列表相同，则这两个字符串是变位词。ActiveCode 3 展示了这种方法。

```
def anagramSolution4(s1,s2):
    c1 = [0]*26
    c2 = [0]*26

    for i in range(len(s1)):
        pos = ord(s1[i])-ord('a')
        c1[pos] = c1[pos] + 1

    for i in range(len(s2)):
        pos = ord(s2[i])-ord('a')
        c2[pos] = c2[pos] + 1

    j = 0
    stillOK = True
    while j<26 and stillOK:
```

```
        if c1[j]==c2[j]:
            j = j + 1
        else:
            stillOK = False

    return stillOK

print(anagramSolution4('apple','pleap'))
```

代码 20 Count and Compare (active7)

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

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

这是一个常见的现象。很多情况下你都需要在时间和空间的权衡中做出选择。在这个例子中，额外的空间量并不足道。但是如果潜在的字母多达几百万个，这将会是一个问题。作为一个计算机学家，当要做出算法的选择时，需要你根据具体的问题来决定利用计算资源的最好方式。

自测

Q-1: 判断下列代码段的大 O 级别

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

a) O(n)
b) O(n^2)
c) O(log n)
d) O(n^3)

Q-2: 判断下列代码段的大 O 级别

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

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

a) O(n)
b) O(n^2)
c) O(log n)
d) O(n^3)

Q-3: 判断下列代码段的大 O 级别

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

a) O(n)
b) O(n^2)
c) O(log n)
d) O(n^3)

Python 数据结构的性能

既然大家对大 O 表示法以及各种功能的差异有了大体的了解，在这一节我们的目标是来讲授一些大 O 表示法在 Python 列表和字典是如何运转和执行的知识，然后我们将向你展示一些计时操作，以此来阐明在各种数据结构上使用某个操作的优缺点。这个对你去理解这些 Python 数据结构的性能是非常重要的，

因为随着我们在本书接下来的部分执行其它数据结构时，它们是我们将使用到的构件。在该节，我们不打算去解释这个操作为什么是这样的的问题，不过在接下来的章节我们将看到一些列表和字典的实现方式以及操作是如何依赖这些实现方式的。

2.3.1.列表 List

当要执行列表数据结构时，Python 的编写者有多种方式去实现这个目的。这些方式中的每一个都对列表运行操作的速度产生影响。为了做出正确的选择，他们着眼于人们最经常使用的列表数据结构的方式，然后他们去完善他们的列表使用手段，所以平常的操作是非常快的。当然，他们也会尽力去使那些不常用的操作速度快。但是当不得不去做一个折衷处理时，不常用的操作常常被牺牲用来支持那些更常用的操作。

两个普通操作—被编入和分派到一个索引位置，无论列表多么大，这些操作都花费相同的时间。当一个操作的速度像这样不依赖列表的大小，那么这个操作就是 O(1)。

另一个非常平常的程序任务是去扩充一个列表。这有两种方式去生成一个更长的列表。你可以用“append”操作或者串联运算符。这个“append”操作是 O(1)。然而，串联运算符是 O(k)，在这里 k 是指正在被连接的列表的大小。了解它对你是非常重要的，因为可以通过选择工作的正确的工具来使你的程序更加有效。

让我们来看一下四种不同的方法来生成从 0 到 n 的列表。首先，我们尝试一个回路然后通过串联生成一个列表，然后，我们用“append”的操作而不是串联操作。接下来，我们使用列表推导来生成一个列表。最后，可能是最明显的方法，通过列表结构体的访问使用“range”的功能。列表 3 展示了生成列表四种方式的代码。

列表 3

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

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

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

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

为了获得我们实行某个功能所花费的时间，我们将使用 python 的 timeit 的模块。这个 timeit 的模块被设计成通过在一个一致的工作环境运行程序以及使用尽可能与操作系统相似的计时机制来让 Python 开发者实现跨平台计时测量。

使用 timeit 时，你需要创建一个 Timer 的对象，这个对象的参数是两个 Python 语句，第一个参数是是你想要给其计时的 Python 语句，第二个参数是建立这个测试你将要跑的语句。这个 timeit 模块然后将算出执行该语句一定次数所花费的时间。默认的情况下，timeit 将会试着运行该程序一百万次。当程序被执行完时，它会以一个浮点数的形式，以秒为单位返回一个时间。然而，因为它执行程序执行了一百万次，当你去执行程序一次时，你可以看到一个以微秒为单位的数字。你也可以赋给 timeit 一个命名的数字参量，这个数字允许你指定所测试的程序要运行多少次。这接下来的会话展示运行我们每个测试程序 1000 次所花的时间。

```
t1 = Timer("test1()", "from __main__ import test1")
print("concat ",t1.timeit(number=1000), "milliseconds")
t2 = Timer("test2()", "from __main__ import test2")
print("append ",t2.timeit(number=1000), "milliseconds")
```

```
t3 = Timer("test3()", "from __main__ import test3")
print("comprehension ",t3.timeit(number=1000), "milliseconds")
t4 = Timer("test4()", "from __main__ import test4")
print("list range ",t4.timeit(number=1000), "milliseconds")

concat  6.54352807999 milliseconds
append  0.306292057037 milliseconds
comprehension  0.147661924362 milliseconds
list range  0.0655000209808 milliseconds
```

在上面这个实验里，我们要测试的语句是调用函数到 test1(),test()2,等等。这个设置语句或许对你来说非常陌生，因此让我们更详细地来认识它。你可能非常熟悉“from”、“import”语句，但是这个通常被用在一个 Python 程序文件的开头。在这种情况下，语句“from __main__ import test1”把来自“__main__”的命名空间的函数 test1 输入到 timeit 为计时操作建立起来的命名空间。Timeit 模块这样做是因为它要在某个环境下执行计时测试，这个环境要求能够对你可能已经生成的、并且偏离的、甚至可能以某种不可预见的方式与你的功能的执行产生冲突对变量进行整理。

在上面的实验中，我们可以清楚地看到“append”操作以 0.30 毫秒的速度远快于串联操作的 6.54 毫秒。在上述实验我们也展示了另外两个生成一个新列表的方法所需的时间：使用访问“range”和某个列表推导的列表结构体。值得一提的是，列表推导比“append”操作快了 2 倍多。

关于这个小实验一个最终的结论是：在上面你能看到的是所有的时间，包括一些访问测试功能的需要的时间，但是我们可以认为功能访问所用时间在所有四种情况下是完全相同的。因此，我们仍然可以获得此操作一个有意义的比较。它不是准确的去说串联操作花费了 6.54 毫秒，而应该是说串联测试操作花费了 6.54 毫秒。作为一个练习，你可以去测试访问一个空函数，然后从上面的时间中去掉访问空函数所花费的时间。

既然我们已经知道操作如何具体地测量，那么你可以去考虑表 2 去看看所有基础列表操作的大 O 效率。在仔细思考表 2 之后，你可以试着对“pop”的两个不同调用时间去发疑和探讨。当要删除的元素在列表的结尾时，复杂度是 O(1)，但是当去除的元素在列表中是第一个元素或者在列表中间其它地方时，复杂度便是 O(n)。这个的原因在于 Python 去执行列表的方式。当某项从列表的前头移走，这个列表剩余的每个元素就要移动到一个与开头更相近的位置（向左平移一个单位）。或许现在你认为这个是愚蠢的，但是如果你留心了表 2，你会发现这个执行可以让索引操作变成 O(1)，这是 Python 执行者认为的一个比较好的交易。

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

concatenate	O(k)
sort	O(n log n)
multiply	O(nk)

表格 12 Python 列表操作的大 O 效率

为了演示性能差异，我们用 timeit 模块来做另一个实验。通过从列表末尾删除一个项目，然后从列表开头一个项目，我们能够核实在一个已知大小的列表里“pop”操作的性能，同时我们也能测量“pop”对于不同大小列表的时间。我们所看到的是，纵使该列表大小在增长，项目从列表末尾弹出所花费的时间依然能够保持稳定。然而，从列表开头删除所需时间却随列表增大而增加。

列表 4 展示了一种方式去测量“pop”操作两种使用方式的差异。正像你从第一个例子所看到的那样，从末尾弹出所花费的时间是 0.0003 毫秒，然而从列表开头弹出却需 4.82 毫秒。对一个含有两百万元素的列表，两者相差 16000 倍。

在列表 4 这里有一些事情需要提一下：第一个是“from __main__ import x”语句，它不能定义一个函数，它能让我们在测试里使用列表中的对象 x。这个方法可以让我们去计单个删除语句的时间，然后获得单个操作最准确的时间。由于“timer”会重复 1000 次，因此必须指出列表通过环路大小每次增加 1 是非常重要的，否则列表的长度在不断减少。但是由于初始列表包含两百万个元素，所以我们只会减少列表总体大小的 0.05%，影响并会不会太大。

列表 4

```
popzero = timeit.Timer("x.pop(0)", "from __main__ import x")

popend = timeit.Timer("x.pop()", "from __main__ import x")

x = list(range(2000000))
popzero.timeit(number=1000)
4.8213560581207275
x = list(range(2000000))
popend.timeit(number=1000)
0.0003161430358886719
```

虽然我们的第一个测试已经展示了“pop(0)”事实上是比“pop()”要慢的，但是它没有证实“pop(0)”是 O(n)和“pop()”是 O(1)。为了证实这个结论，我们需要去看下访问一个列表的操作的性能。列表 5 实施了这次测试。

列表 5

```
popzero = Timer("x.pop(0)",
"from __main__ import x")
popend = Timer("x.pop()",
               "from __main__ import x")
print("pop(0)   pop()")
for i in range(1000000,100000001,1000000):
    x = list(range(i))
    pt = popend.timeit(number=1000)
    x = list(range(i))
    pz = popzero.timeit(number=1000)
print("%15.5f, %15.5f" %(pz,pt))
```

图 3 展示了实验的结果。你可以看到，随着列表的变得越来越大，“pop(0)”操作花费的时间也随之增加，然而“pop()”操作所需时间却依旧保持平稳。对 O(n)和 O(1)算法来说，事实上这就是我们所希望看到的。

在我们的小实验里，一些误差来源，包括当其他程序正在计算机上运行，都可能会减慢我们的代码。因此，尽管我们尽可能地去减少计算机做的其它的事情，最终仍然会存在一些变差。这就是为什么要获得足够多的信息以确保实验结果的可信度，而且要循环运行测试 1000 次。

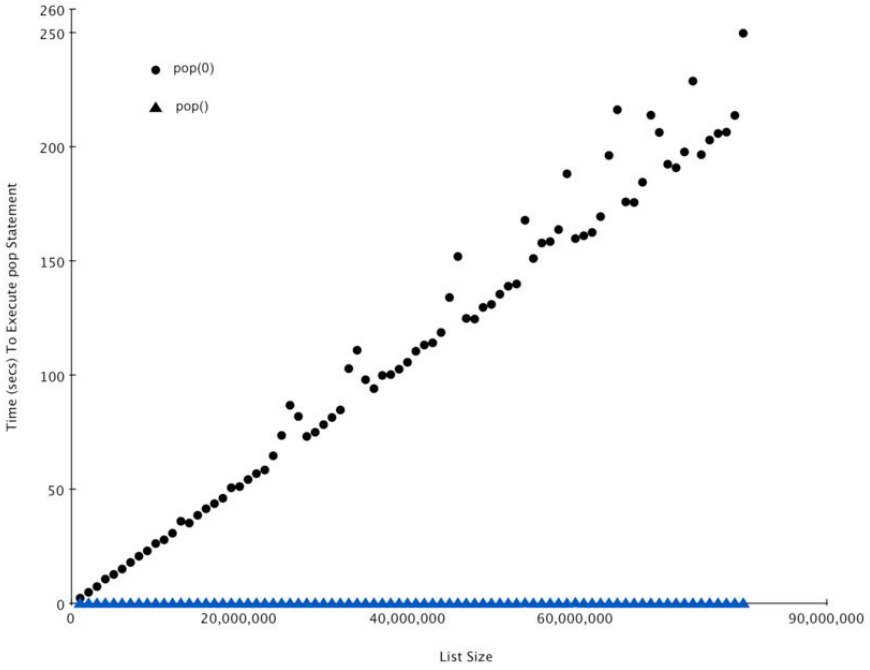

图 15 比较 pop 和 pop(0)的差异

2.3．2.字典 Dictionary

Python 中第二重要的数据结构就是字典。就如同大家回忆起（之前的内容），字典区别于列表之处在于你在字典中找到数据项通过一个 key（关键码）而不是一个位置。本书后面你将会了解到许多执行字典的方法，现在(关于字典)你需要注意到的最重要的事就是对字典中项的赋值和获取大 O 数量级都是 O（1）。另一个重要操作就是包含操作（判断一个 key（关键码）是否在字典中）的大 O 数量级也是 O（1）。所有字典操作的效率都概括在表 3 中。而对字典性能的一个重要方面的注解就是我们所提供表中的效率是基本性能。在某些罕见的情况下，包含，取值，赋值操作可能性能劣化到 O（n），但是在之后的章节，当我们谈到执行字典的不同方法时，我们会习惯的。

表 3: 字典操作的大 O 性能

复制	O(n)
取值	O(1)
赋值	O(1)
删除值	O(1)
包含	O(1)
迭代	O(n)

对于最后一个性能试验，我们将会对比字典和列表中的包含操作的性能。在其过程中，我们会确信列表包含操作的大 O 数量级是 O（n），而字典的是 O（1），这个实验中我们用来做比较的两者很简单，我们会做一个含有一个范围数字的列表，然后随机选取数字并检查是否在列表里，如果性能表正确并且数字在列表中，列表越大，检索花费时间越长。

字典中我们重复上述实验，以数字做 key（关键码），在此实验中，我们看到，确定数字是否在字典中不仅更快，而且检查耗时是不变的，即使字典变大。Listing6 执行了比较，注意我们确实做了一样的操作：“数字在容器中”。不同之处在于第七行的 X 是列表，第九行的 X 是字典。

Listing6

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

表格 4 概括了运行 Listing6 的结果，可以看到字典（的速度）一致地更快。对于包含 10000 个元素的最小列表，字典比它快 89.4 倍，而对于包含 990000 个元素的最大列表，字典速度更是快了 11603 倍！同样能看到，对于列表包含操作所耗时间随着列表增大而线性增加。这证实了列表的包含操作数量级是 O（n）这个结论。并且，字典中的包含操作时间是一定的，即时字典大小增长了。实际上，对于包含 10000 个数据项的字典，包含操作耗时 0.004 毫秒，而对于包含 990000 个数据项的字典，耗时同样 0.004 毫秒。

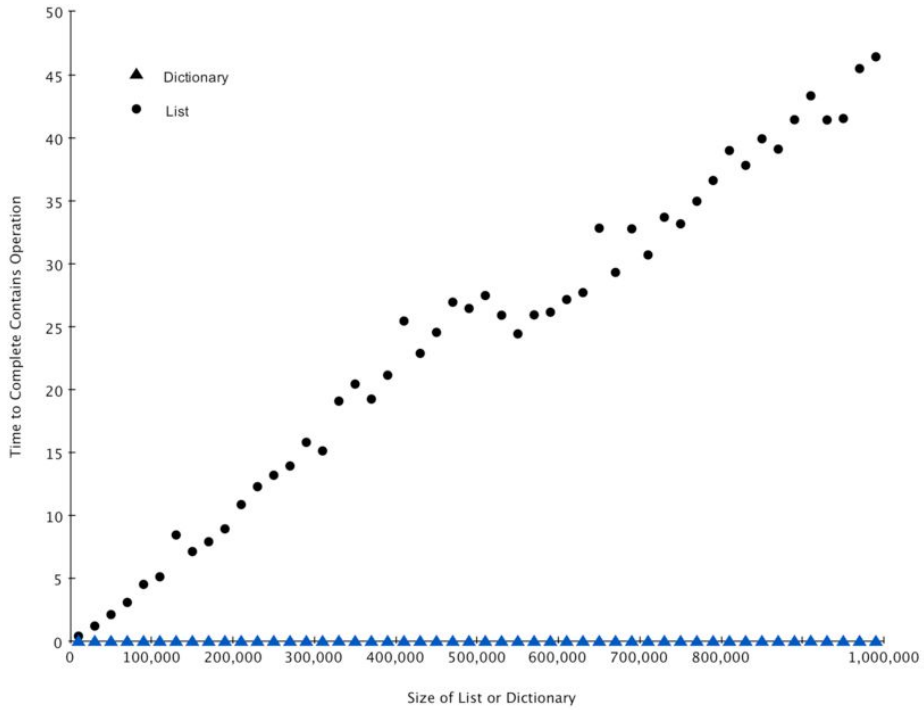

图 16　列表与字典 in 操作的对比

因为 python 是一门正在发展的语言，情景之后还有改变进行。关于 python 数据结构的最新信息可以在 python 网址上找到。自写 python 的维基百科起，编者有一段美妙的时光，复杂的书页则在 the Time Complexity Wiki 上。

自测

Q-4：以下哪个 List 操作不是 O（1）？

```
a) list.pop(0)
    b) list.pop()
    c) list.append()
d) list[10]
e) all of the above are O(1)
```

Q-5：以下哪个字典操作是 O（1）？

```
a) 'x' in mydict
    b) del mydict['x']
    c) mydict['x'] == 10
    d) mydict['x'] = mydict['x'] + 1
e) all of the above are O(1)
```

2.4. 小结

- 算法分析是对一个算法进行与具体实现无关的性能分析；
- 大 O 表示法可以依据一个算法对不同规模问题的主干处理过程，对算法的性能进行分级。

2.5. 关键词

基本情况	大 O 表示法	暴力
核对	指数的	线性
对数线性	对数	数量级
二次的	时间复杂度	最差情况

2.6. 问题讨论

1.判断下列代码片段的大 O 级别：

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

2. 判断下列代码片段的大 O 级别：

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

3. 判断下列代码片段的大 O 级别：

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

4. 判断下列代码片段的大 O 级别：

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

5. 判断下列代码片段的大 O 级别：

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

6. 判断下列代码片段的大 O 级别：

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

2.7. 编程练习

1. 做计时实验，验证 List 的按索引取值确实是 O(1)
2. 做计时实验，验证 Dict 的 get item 和 set item 都是 O(1)
3. 做计时实验，比较 List 和 Dict 的 del 操作符性能 del lst[i]/del dic[key]
4. 给定一个随机顺序的数列表，写一个复杂度为 O(nlogn)的求第 k 小的数的算法
5. 请改进上述的算法，使之复杂度降低为 O(n)

3. Stack 栈

基本数据结构

3.1 目标

- 了解抽象数据类型：栈 stack、队列 queue、双端队列 deque 和列表 list；
- 能够采用 Python 列表数据类型，来实现 stack/queue/deque 等抽象数据类型；
- 了解基本线性数据结构各种具体实现算法的性能；
- 了解前缀、中缀和后缀表达式；
- 采用 stack 对后缀表达式进行求值；
- 采用 stack 将中缀表达式转换为后缀表达式；
- 采用 queue 进行基本的点名报数模拟；
- 能够识别问题属性，选用 stack、queue 或者 deque 中更为合适的数据结构；
- 能够通过节点和节点引用的模式，采用链表来实现抽象数据类型 list；
- 能够比较链表实现与 Python 的 list 实现之间的算法性能。

3.2 线性结构

我们将开始我们的数据结构的研究，需要考虑四个简单但非常强大的概念: 栈，队列，双端队列，和列表。它们是数据集合的示例，它们的项序取决于它们如何添加或删除。一旦添加一个项目，它停留在的那个位置，它相对于其他元素之前和之后，这些常被称为线性数据结构。

线性结构可以被认为是有两个端。有时，这些目标被称为“左”和“右”或在某些情况下，“前”和“后”。你也可以称他们为“顶”和“底”。给“端”名字不重要。区别一个线性结构的是一项进行添加和删除的方式，特别是在这些添加和删除发生的位置。例如，一个结构可能允许新的项目应仅在一端加。一些结构可能会让项目要从两端切除。这些变化引起了计算机科学中的一些最有用的数据结构的出现。他们出现在许多算法，可以用来解决各种重要问题。

一个堆栈（有时称为“下推栈”）是一个有序的集合项目添加新项目和现有项目的去除总是在同一

地方。这一端通常被称为“顶”。另一端的顶部被称为“基地”。
堆栈的基础是重要的因为存储在靠近基地代表那些已经在堆栈中储存时间最长的堆栈物品。最近添加的项目的一个位置是第一次被删除。这种排序原则有时也被称为 LIFO 法，即后进先出。它提供了一个排序基于集合中的时间长度。新项目的顶部附近，而老项目基地附近。

栈的例子很多发生在日常生活中。几乎所有的餐厅有一堆托盘或板可以在顶部，露出一种新的托盘或板的下一个客户。想象一下，在桌上的一摞书（图 1）。只有一本书的封面是可见的就是最上面的一个。为了访问堆栈中的其他书，我们需要删除那些排在上面的书。图 2 显示了另一个栈。这其中包含着一些原始的 Python 数据对象。

图 17 书的栈

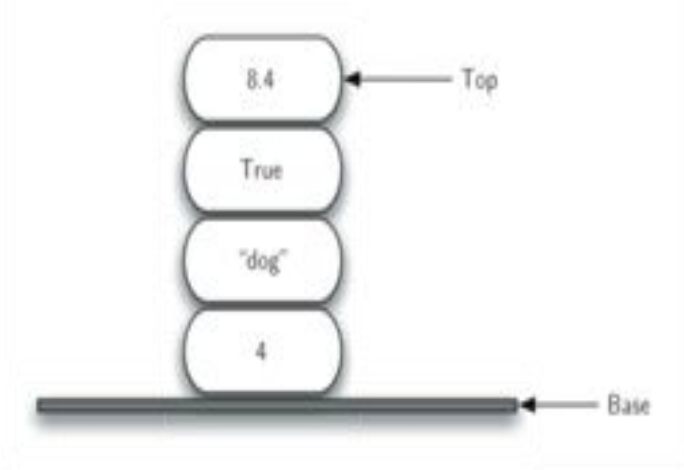

图 18 原始 python 的栈

一个相关的堆栈的最有用的想法来自项目的简单的观察，当它们被它们添加然后删除的时候。假设你开始了一个干净的桌面。现在把书一本一次放在彼此的顶部。你就是在构建一个栈。考虑会发生什么，当你开始删除书籍。他们正在拆除的顺序是完全相反的顺序，栈是非常重要的，因为它们可以用于反转项目的顺序。插入的顺序和拆卸顺序相反。图 3 显示了 Python 数据对象的堆栈，这是创造和再当物品被删除。注意对象的顺序。

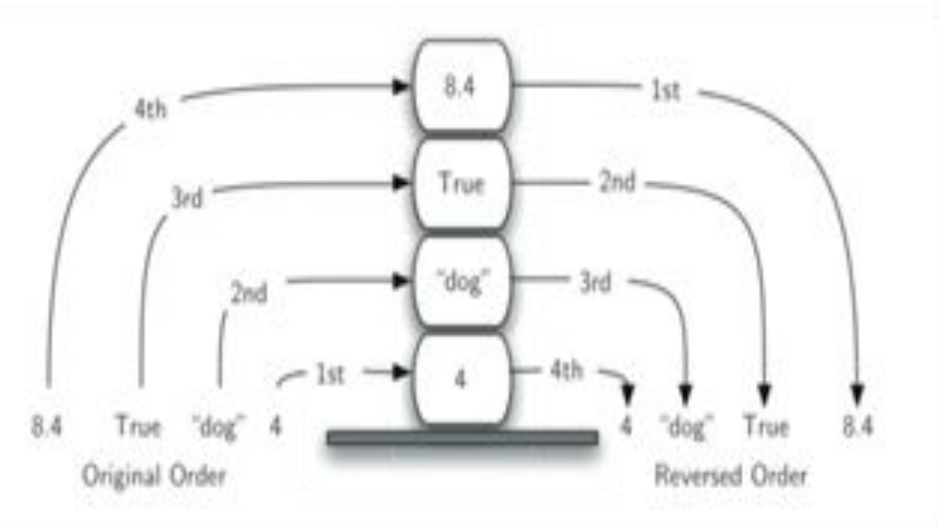

图 19 栈内物品的反转性

考虑到这种逆转性，你也许可以认为栈产生于你使用你的电脑。例如，每个浏览器都有后退按钮。当你浏览网页时，这些网页都放在一堆（实际上是将堆栈顶上的网址）。你正在浏览的当前页上的第一页是在基地。如果你点击后退按钮，你开始以相反的顺序通过网页。

3.3（1.2）抽象数据类型栈

栈的抽象数据类型是由以下结构和操作定义。堆栈是结构化的，如上面所描述的，作为一个有序的收藏的物品，物品的添加和删除的一端称为“顶”。栈的命令是按后进先出进行的。Stack()创建一个新的空栈。它不需要参数，并返回一个空栈。

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

pop()从栈顶删除项目。它不需要参数,返回 item。栈被修改。

peek()从栈返回顶部项目而不删除它。它不需要参数。堆栈是不被修改的。

isEmpty()测试看栈是否为空。它不需要参数，返回一个布尔值。

size()返回堆栈上的项目数。它不需要参数，并返回一个整数。

例如，如果 S 是一个堆栈，已被创建并开始是空的，那么表 1 显示一个顺序栈的操作结果。在栈顶内容，项目被列在右边。

栈的操作	栈内的内容	返回值
s= Stack()	[]	Stack object
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

表格 13 示例栈的操作

3.3.3 用 Python 实现堆栈。

现在我们有了明确的栈的抽象数据类型，我们将我们的注意力转向使用 Python 来实现堆栈。回想一下，当我们给出了一个抽象数据类型的物理实现，我们说的这种实现指的是实现一种数据结构。

正如我们在第 1 节中描述的，在 Python 中，正如在任何面向对象的编程语言，对于一个抽象数据类型的实现选择如栈是一种新的“类”的创造。堆栈操作是实现方法。进一步，实现一个栈，这是一个元素的集合，它是利用由 Python 提供的原始集合的简单和强大。我们将使用一个列表。

记得，Python 中的列表类提供了一个有序的收集机制和一套方法。例如，如果我们有列表[2,5,3,6,7,4]，我们只需要决定这个列表的哪一端将考虑为堆栈的顶部，那一端作为底部，一旦作出决定，该操作可以采用列表的方法如追加和删除。

下面的栈的实现（ActiveCode 1）假设列表的末尾将栈顶元素。随着堆栈的增长（如 push 操作发生），新的项目将在列表的末尾添加。弹出操作会操作同端。

```
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)
```

代码 21 用 python 列表实现栈 (stack_1ac)

记住，什么事情也没有发生，当我们点击运行按钮以外的其他类的定义。我们必须创建一个堆栈的对象，然后使用它。activecode 2 显示动作栈类我们执行的操作序列从表 1。注意，堆栈类的定义是从 pythonds 模块引入。

Note
The pythonds module contains implementations of all data structures discussed in this book. It is structured according to the sections: basic, trees, and graphs. The module can be downloaded from pythonworks.org.

```
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())
```

代码 22 (stack_ex_1)

需要注意的是，我们可以选择使用列表项部的开始而不是结束时实现堆栈，这一点很重要重要。在这种情况下，以前的删除和添加方法将不再成立，我们将在索引位置 0（列表中的第一项）准确地使用 Pop 和 Insert。实施是 codelens 1 所示。

```
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())
```

代码 23 可替代的栈的实现(stack_cl_1)

这种能力在保持逻辑特征中改变一个抽象数据类型的物理实现是一个在工作中抽象的例子。然而，尽管该栈的工作方式，如果我们考虑这两种实现的性能，肯定是有差别的。记得 push 和 pop()操作是 O（1）。这意味着，不管有多少项目在堆栈，第一个实现将在固定的时间执行 push 和 pop。第二实施的性能，insert（0）和 pop（0）操作都需要 O（n）,n 为一个栈的大小，显然，即使实现在逻辑上是等价的，他们在进行基准测试时会有非常不同的时间代价。

自我检测：

Q10：给定下面的栈的操作序列，在堆栈时序列完成时栈顶的项目是什么？

```
m = Stack()
m.push('x')
```

```
m.push('y')
m.pop()
m.push('z')
m.peek()
```

a) 'x'
b) 'y'
c) 'z'
d) The stack is empty

Q11：给定下面的栈的操作序列，在堆栈序列完成时栈顶项目是什么？

```
m = Stack()
m.push('x')
m.push('y')
m.push('z')
while not m.isEmpty():
  m.pop()
  m.pop()
```

a) 'x'
b) the stack is empty
c) an error will occur
d) 'z'

Q12 写一个函数 revstring（mystr），使用栈反转的字符串。

3.3.4 简单括号匹配。

我们现在要注意用栈来解决真正的计算机科学问题。毫无疑问你写的算术表达式，如:

```
(5+6)*(7+8)/(4+3)
```

在圆括号用于命令操作的性能。你也可能在一种语言如 Lisp 的结构中有一些编程经验

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

这定义了一个函数，名叫“乘方”，将返回其参数 n 的平方，Lisp 是使用因为大量的括号而臭名昭著。

在这些例子中，括号必须出现在一个平衡的方式。平衡括号意味着每一个开放的象征有一个相应的关闭符号和一对圆括号对正确的嵌套。考虑下面的括号字符串正确的平衡方式：

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

((((()))

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

比较以下，这是不平衡的：

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

()))

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

区分括号是正确的平衡和不平衡的，是认识许多编程语言结构的一个重要组成部分和基本能力。

接下来的挑战是写一个算法，将从左至右读取括号字符串，并决定它们是否平衡。为了解决这个问题我们需要做一个重要的观察。当你的符号从左到右排列时，最近的开括号必须匹配下一个关闭的符号（见图 4）。另外，第一开口符号的处理可能要等到匹配的最后一个符号。结束标志按与开始标志相反的顺序打开符号匹配；他们的匹配自内而外进行。这是一个线索，堆栈可以用来解决这个问题。

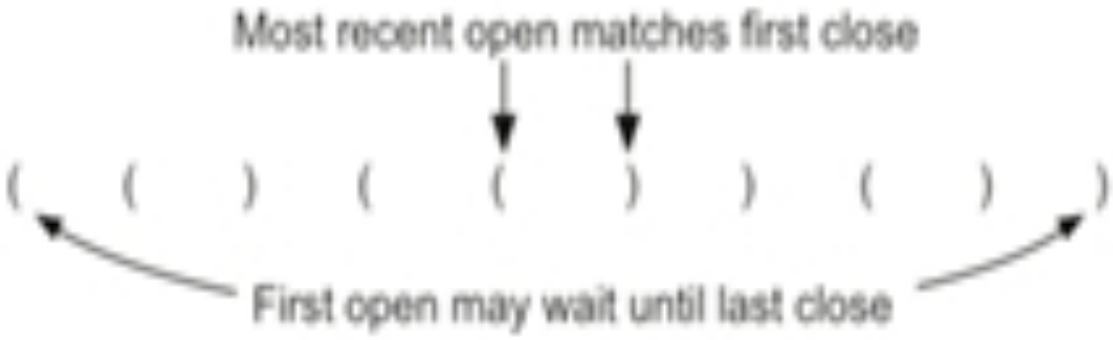

图 20 合并括号

一旦你认为一个栈是匹配括号算法的适当的数据结构，算法的语句是简单的。从一个空栈，操作从左到右的括号字符串。如果一个符号是一个开括号，把它在栈中为一个信号，相应的关闭标志就需要出现。如果，在另一方面，符号是括号，弹出栈。只要是有可能弹出栈匹配每一个闭括号符号，括号就会保持平衡。一旦在某些时候时候在堆栈如果没有开放的字符匹配关闭符号，字符串就是不平衡。在字符串的结尾，当所有的符号进行处理，堆栈应该是空的。Python 代码来实现这个算法是 ActiveCode 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('(()'))
```

代码 24 解决平衡括号问题(parcheck1)

这个函数， parchecker，假定一个栈类可以操作括号字符串并返回一个布尔值来表示结果是否平衡。注意布尔变量，它被初始化为真，如果括号为真就表示平衡。如果当前的符号是“(”，然后把它推到堆栈上（9–10 行）。注意在 15 行，pop 是从堆栈中简单地删除一个符号。返回值没有使用因为我们知道在它之前必须看到一个开括号。最后（19–22 行），只要表达式是平衡而且堆栈已被完全清除，输出的字符串就代表一个正确的平衡括号的序列。

3.3.5 匹配符号（通用情况）

上面显示的平衡括号的问题是一个更一般的情况在许多编程语言中出现的具体案例。平衡和嵌套不同类型的打开和关闭括号的一般问题经常出现。例如，Python 中的方括号，[and]，用于表；大括号，{and}，用于字典；和括号，（and），用于元组和算术表达式。我们可以混合的符号，只要他们每一个都保持开闭括号的匹配关系的。符号字符串例如：

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

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

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

它们是适当的平衡，不仅每一个开括号都有相应的关闭括号符号，而括号的类型匹配得很好。
比较以下不平衡的字符串：

```
([)]

(()))

[{()]
```

简单的括号检查从以前的部分可以很容易地扩展到处理这些新类型的括号。记得，每一个开放的括号，都是在堆栈上等待与之匹配的结束括号在序列中出现。唯一不同的是，当关闭符号出现时，我们必须检查以确保它的类型正确地匹配在栈顶的闭括号。如果两个括号不匹配，字符串是不平衡的。再次，如果整个字符串都被处理，而且并没有东西留在堆栈上，字符串是正确的平衡。

Python 程序实现这是在 ActiveCode 1 所示。唯一的变化出现在 16 行调用 helper 函数，在那里匹配，协助符号匹配。每个从堆栈中删除的符号都必须检查它是否符合当前的关闭标志。如果发生不匹配，布尔变量设置为 false。

```
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('[{()]'))
```

代码 25 解决一般性的字符括号匹配(parcheck2)

这两个例子表明，在计算机科学中堆栈是语言处理中非常重要的数据结构。几乎任何你能想到的符号，有某种类型的嵌套符号，用来在一个平衡的秩序匹配。在计算机科学中还有其他许多重要的地方必须使用堆栈。我们将在下一节继续探讨。

3.3.6 十进制数转换为二进制

在你的计算机科学的研究中，也许你已经以一个或另一种方式接触到的一个二进制数的概念。二进制表示是重要的，因为在计算机科学中所有的值存储在计算机的形式为一串二进制数字，一串 0 和 1。如果没有来回转换常见的表示和二进制数之间的能力，我们就会和电脑相处得很尴尬。

整数的值是常见的数据项。它们总是在用计算机程序和计算中。我们在数学课已经了解他们，而且使用十进制数字，或以 10 进制表示它们。十进制数 23310 和其相应的等值二进制数 111010012 被解释为

$2\times10^2+3\times10^1+3\times10^0$

和

$1\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$

但是我们怎么能很容易地转换成二进制数整数的值呢？答案是用一种称为“除以 2”的方法使用堆栈跟踪的数字的二进制结果。

“除以 2”算法将假设我们开始与一个大于 0 的整数。一个简单的迭代，然后不断地将十进制数除以 2 并记录余数。第一部给出的信息是判断返回值是偶数还是奇数。一个偶数会有余数 0。它会在那些地方有数字 0。一个奇数将有一个余数 1，于是在那些二进制位有数字 1。我们认为把我们的二进制数建造成一个数字序列；我们计算的第一个余数事实上是我们二进制数的最后一个。如图 5 所示，我们再次看到了反转特性，这是可以用栈作为合适数据类型解决问题的一个信号。

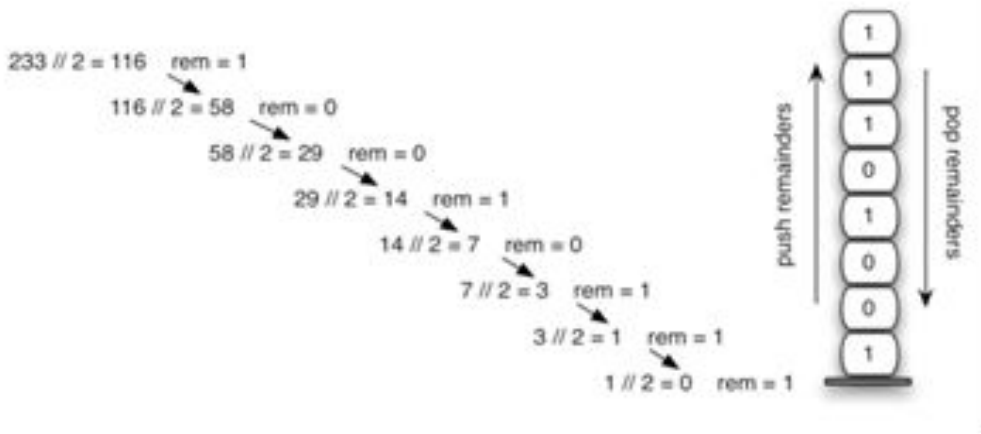

图 21 十进制到二进制转化

ActiveCode 1 用 Python 代码实现了“除以 2”算法。 该函数 divideBy2 是接受一个十进制数作为参数，并把它反复除以 2。第 7 行采用内置模块操作方法，“%”，即取余，第 8 行然后将其压入栈中。在除过程中发现余数是 0 后，就 11-13 行构造一个二进制字符串。11 行创建了一个空字符串。二进制数从堆栈中一个一个弹出，附加到字符串的右端，然后返回二进制字符串。

```
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))
```

代码 26 从十进制转化位二进制(divby2)

二进制转换算法可以很容易地扩展到任何进制执行转换。在计算机科学中使用许多不同的编码是很常见的。其中最常用的是二进制，八进制（进制为 8），和十六进制（进制 16）。

十进制数 233 对应的八进制和十六进制的等价值为 3518 和 e916 被解释为

$3\times8^2+5\times8^1+1\times8^0$

和

$14\times16^1+9\times16^0$

”divideby2“功能可以修改，用来接受不仅十进制值也可以是一个其他目标进制。“除以 2”想法被简单地替换为一个更一般的“除以进制”的算法。一个新的功能函数叫做 baseConverter，如 ActiveCode 2 所示，以十进制数 2 和 16 之间的任何进制为参数。其余数仍推到栈中直到被转换的值变为 0。可以用一个轻微的变化来构造相同的从左到右的字符串。2 进制到 10 进制需要通过最多 10 个数字，所以典型的数字 0，1，2，3，4，5，6，7，8，和 9 就能成功。问题是，当我们超 10 进制。我们不能再简单地用余数，因为他们的余数表示为一个两位的十进制数。相反，我们需要创建一组数字可以用来表示那些超过 9 的余数。

```
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))
```

解决这个问题的办法是增加了一些包括字母符号。例如，十进六制使用六个英文字母和 10 个数字来表示 16 个进制位。为了实现这一点，创建一个数字串（第 4 行 Listing6 中）,存储在相应位置的数字。0 是在位置 0，1 是在位置 1，A 是在位置 10，B 是在位置 11，等等。当一个余数从堆栈中弹出，它可以被用来索数字串，正确产生的数字可以添加到返回值中。例如，如果余数 13 是从堆栈中弹出，该字符“D”被添加到结果字符串中。

自我检测

Q-6：一个八进制数 25 的值是什么

Q-7：一个十六进制数 256 的值是什么

Q-8：在 26 进制表示 26 的值是什么

3.3.7. 中缀、前缀和后缀表达式

当你写下算术表达式，例如 B*C，这种形式的表达提供了信息使得你能准确地解译它。 在 B*C 中，变量 B 和变量 C 是相乘的，这是由于表达式中的*出现在 B 和 C 之间。因为运算符出现在运算对象之间，我们将这种表示方法称为中缀表示法。

考虑另一个中缀的例子，A+B*C。运算符+和*仍然在运算对象之间，但是问题出现了：运算符作用的对象是谁？A 是与 B 进行加法运算还是与 B，C 的乘积？中缀表示法有点儿模棱两可。

事实上，你已经阅读和书写中缀表达式很长时间，它们不会给你带来任何问题。原因是，你知道运算符+和*的其他一些事情：每个运算符都有一个优先级。高优先级的运算符先于低优先级的运算符进行运算，乘、除法比加、减法先运算，相同优先级的运算符按照从左到右的顺序进行运算。唯一能改变这种运算顺序的是括号。

我们使用运算符的优先级来解译这个麻烦的表达式 A+B*C：B 和 C 先相乘，其结果再与 A 相加。对于（A+B）*C，由于存在括号，所以加法先于乘法运算。而在 A+B+C 中，按照从左到右的顺序，左侧的加法先于右侧的加法运算。

尽管这些过程对你来说很明显，但对于电脑（需要确切知道每个运算符的运算顺序）而言是很困难的。一种不会产生运算顺序混乱的表示方法是，引入所谓的全括号表示法，即对每一个运算符使用一对括号，以表明运算顺序，这种表示法很明确，没有必要去记住一系列复杂的运算顺序的规则。

表达式 A+B*C+D 可以写成((A + (B * C)) + D)，从中可以看出乘法最先运算，接着是左侧的加法，最后是右侧的加法。根据相同优先级从左到右的规则，A + B + C + D 可以写成(((A + B) + C) + D)。

还有两种初看起来不那么明显但很重要的表达方法。考虑中缀表达式 A + B，如果我们把运算对象之间的运算符+移到前端，结果是+ A B；同理，我们把运算对象之间的运算符+移到末端，结果是 A B +，这些（样）看起来有点奇怪。

运算符位置的改变产生了两种表达法——前缀表示法和后缀表示法。前缀表示法要求所有的运算符位于运算对象的前端。相反的，后缀表示法则要求所有的运算符位于运算对象的后端。下面）一些例子将有助于理解这两种表达法（表 14）。

A + B * C 用前缀表示法表示为+ A * B C，乘法紧紧位于运算对象 B 和 C 之前，表示*优先于+，而加法位于 A 和乘法运算结果之前。

A + B * C 用后缀表示法表示为 A B C * +，乘法紧紧位于运算对象 B 和 C 之后，表示*优先于+，接着才是加法。尽管运算符被移动到相应的运算对象之前或之后，但运算对象之间的相对位置却保持一致——总是 A 到 B 再到 C。

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 * +

表 14 中缀、前缀和后缀表达式的例子

现在，考虑中缀表达式(A + B) * C。在这个例子中，中缀表达式中要求括号的存在——以使得加法可以优先于乘法。然而，前缀表达式只需将+移至 A 和 B 之前得到+ A B，然后将*移到所有对象之前得到* + A

B C，同样使得加法先于乘法运算。同理，后缀表达式表示为 A B + C *，同样的，加法先于乘法运算。

再次考虑这三种表示法（表 15）。一些重要的变化出现了：括号哪里去了？为何前缀、后缀表达式中没了括号？答案是，在前缀、后缀表示法中，运算符的位置能够明确地表示其运算的对象。只有中缀表示法需要额外的符号，如括号。运算符在前缀、后缀表达式中的位置完全决定了运算的顺序，无需依靠其他。因此，中缀表示法是使用中最不理想的一种表示法。

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

表 15 带括号的表达式

表 16 显示的是一些额外的中缀表达式的例子，以及与其等价的前缀、后缀表达式。确保你理解了他们是如何在运算顺序上等价的。

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 * +
A + B + C + D	+ + + A B C D	A B + C + D +

表 16 中缀、前缀和后缀表达式的额外的例子

3.3.7.1 中缀表达式转换为前缀和后缀形式

目前为止我们仅手工转换了几个中缀表达式到前缀和后缀表达式，那你可能会想,是否有一个算法来转换任意复杂的表达式呢？

最容易想到的方法便是利用我们之前提到过的全括号表达式的概念。例如表达式 A + B * C 可以写作(A + (B * C))，这样就显式表达了计算次序。进一步观察可以发现，每一对括号都标定了一个完整操作的起止位置，这个完整操作包含一对运算对象和它们中间的运算符。

观察子表达式(B*C)的右括号，如果把操作符“*”移到右括号的位置，替代它，再删去左括号，得到 BC*，正好把子表达式转换为后缀形式！进一步再把更多的操作符移动到相应的右括号处替代之，再删去左括号，那么整个表达式就完成了到后缀表达式的转换（见图 22）。

图 22：将运算符移至右侧得到后缀表达式

同样的，如果我们把操作符移动到左括号的位置替代之，然后删掉所有的右括号，也就得到了前缀表达式（见图 23）。事实上，每对圆括号的位置都确定了最终得到的前缀或后缀表达式中运算符的位置。

图 23：将运算符移至左侧得到前缀表达式

所以，无论一个表达式有多复杂，要转换成前缀或后缀表达式时都只需要两个步骤：
首先，将中缀表达式转换为全括号形式。然后，将所有的操作符移动到子表达式所在的左括号（前缀）或者右括号（后缀）处，替代之，再删除所有的括号。

看一个更加复杂的例子：(A + B) * C - (D - E) * (F + G)。图 24 展示了将其转化为后缀和前缀表达式的过程。

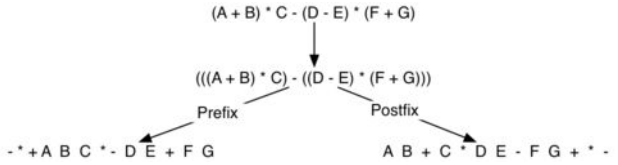

图 24：把一个复杂的表达式转化为前缀或后缀表达式

3.3.7.2 通用的中缀转后缀算法

现在我们需要一个通用的能把任意中缀表达式转化为后缀表达式的算法。要实现这个算法我们需要更加细致地观察转化的过程。

仍然考虑表达式 A + B * C，其对应的后缀表达式为 A B C * + 。注意到操作数 A B C 的顺序并没有改变，只是操作符的位置变了，“+”由原来的从左到右第一个操作符变为了最后一个，这是由于“*”的优先级比“+”高。结果是操作符的出现顺序在后缀表达式中反转了。

在中缀表达式转换为后缀形式的处理过程中，由于操作符比操作数要晚输出，所以在扫描到对应的第二个操作数之前，需要先把操作符保存起来。而这些暂存的操作符，由于优先级的规则，还有可能要反转次序输出。正如在例子 A+B*C 中，“+”虽然先出现，但其优先级比后面的“*”要低，所以要等“*”处理完后才能处理。这种反转特性，使得我们考虑用栈（stack）来保存暂时未处理的操作符。

再来看 (A+B)*C，其对应的后缀表达式是 AB+C*。这一次，“+”的输出要比“*”早，这是因为圆括号使得“+”的优先级提升，高于括号之外的“*”。这反映了转换算法的工作过程：从左到右看，如果遇到左括号就标记下来，其后出现的操作符优先级提升了，一旦再遇到对应的右括号，就可以马上输出这个操作符。

在从左到右逐个扫描中缀表达式中的字符过程中，我们用一个栈来暂存未处理的操作符，这样可以提供必要的顺序反转。栈顶的操作符总是最新暂存进去的，当读取到一个新的操作符时，就需要跟栈顶的操作符比较下优先级再进行行处理。

假定中缀表达式是由空格隔开的一系列单词（token）构成，操作符单词包括 *、/、+、- 和 ( )，而操作数单词则是单字母标识符 A、B、C 等。以下步骤可以产生一个字符串形式的后缀表达式：

1. 创建空栈 opstack 用于暂存操作符，空表用于保存后缀表达式。
2. 用字符串的 split 方法将中缀表达式转换为单词（token）的列表。
3. 从左到右扫描中缀表达式单词列表。
   a) 如果单词是一个操作数，则直接添加到后缀表达式列表的末尾。
   b) 如果单词是一个左括号“(”，则压入 opstack 栈顶。
   c) 如果单词是一个右括号“)”，则反复弹出 opstack 栈顶的操作符，加入到输出列表末尾，直到碰到左括号。
   d) 如果单词是一个操作符 *、/、+ 或 - ，则压入 opstack 栈顶。但在压入之前，要比较其与栈顶操作符的优先级，如果栈顶的高于或等于它，就要反复弹出栈顶操作符，加入到输出列表

末尾，直到栈顶的操作符优先级低于它。

4. 中缀表达式单词列表扫描结束后，把 opstack 栈中的所有剩余操作符依次弹出，添加到输出列表末尾。

图 25 演示了转换算法对 A * B + C * D 的作用过程。注意，由于乘法的优先级高于加法，当算法扫描到“+”时，第一个“*”首先被弹出，而当算法扫描到第二个“*”时，“+”却不被弹出，仍留在栈中。在中缀表达式被扫描完之后，栈将剩余的所有操作符弹出（pop），“+”便被放在后缀表达式的尾部。

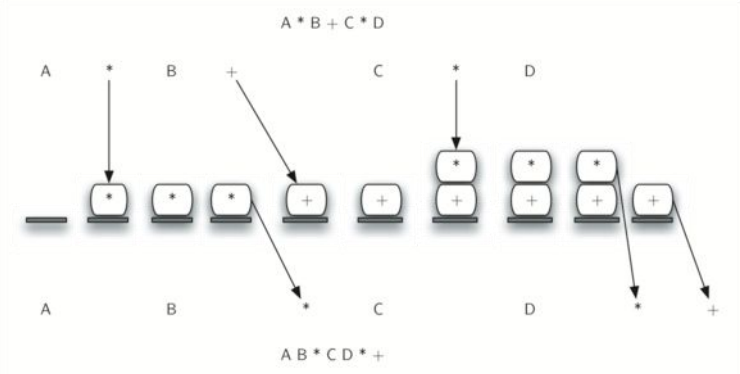

图 25：将 A * B + C * D 转化为后缀表达式

为了将算法编写为 python 代码，我们用一个字典 prec 来保存各个运算符的优先级， prec 把运算符映射为一个整数以利于优先级的比较。左括号的优先级应设为最低，这样所有操作符的优先级都高于它，从而被压入到它的上部。第 15 行把操作数定义为任意的大写字母或数字。完整的算法实现见代码 28：

```
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 )"))
```
代码 28：中缀表达式转化为后缀表达式（inToPost）

在 Python 解释器中的部分执行结果如下：

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

3.3.7.3 后缀表达式求值

作为最后一个栈的例子，我们将会考虑如何求后缀表达式的值。在此例中，栈依然是所选的数据结构。然而，当你扫描后缀表达式时，需要等待是运算数而不是如上述转化法中的运算符。一个新的解决办法是，一旦有一个操作符输入，两个最近的操作数就被用来计算。

我们来关注更细节的问题，考虑后缀表达式 4 5 6 * +。当你从左到右扫描此式，你会首先遇到操作数 4 和 5。这里，直到你看到下一个运算符之前，你并不清楚需要对这些数做什么。将每个数放入栈中，以确保在下一个运算符出现时这些数是可用的。

在本例中，下一个符号是另一个操作数。所以和之前一样，将它压入栈中并检查下一个符号。现在我们看到了一个操作符 *，这意味着两个最近的操作数需要被用在一个乘法运算中。通过两次弹出栈顶端数据的办法，我们可以获得适当的操作数并进行乘法运算（本例结果为 30）。

我们现在通过将运算结果放回栈顶的方法来控制它，这样它可以在后一个在表达式中的操作符出现时被当作操作数用来计算。当最后一个运算符运行完后，栈顶将仅会剩余一个值。弹出并返回这个值，作为表达式的结果。图 26 展示了在例子中的表达式在整个运算过程中栈中内容的变化。

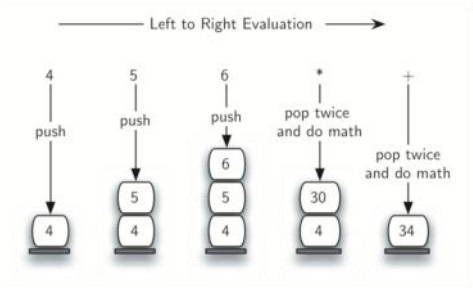

图 26 计算过程中栈中的内容

图 27 展示了一个稍微复杂的例子，7 8 + 3 2 + /。在这个例子里，我们需要注意两点。首先，栈的规模在子表达式运算过程中增长，收缩，之后再增长。其次，除法运算需要小心地处理。回想一下，后缀表达式中的操作数是按着它们的原始顺序出现的，因为后缀表达式仅改变了操作符的位置。当除法的操作数从栈顶弹出，它们是颠倒的。由于除法不是一个可交换算符，也就是说 15/5 和 5/15 是不同的，我们必须清楚操作数的顺序不可交换。

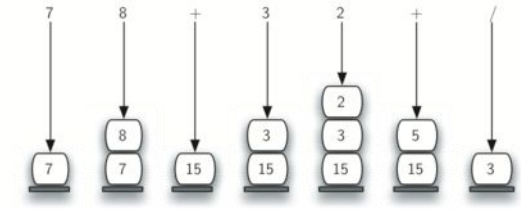

图 27 一个更复杂的计算过程

假设后缀表达式是一个由空格分界的字符串。操作符为*、/、+和-，并且操作数被假定为单位数。将会输出一个整数结果。

1. 创建一个名为 operandStack 的空栈。
2. 用字符串解析(split)方法将字符串转化为列表。
3. 从左至右扫描被提取的单词列表。
   a)如果被提取的是一个操作数，将它从字符串转化为一个整数并把此值压入 operandStack 栈中。
   b)如果被提取的是一个操作符*、/、+或-，则需要两个操作数与之匹配。弹出并清除(pop) operandStack 栈顶的两个元素。第一个弹出值是第二个操作数，而第二个弹出值是第一个操作数。进行计算操作。将结果压回 operandStack 栈中。
4.当输入的表达式被处理完，结果位于栈顶。弹出并返回 operandStack 栈顶的这个值。

完整的后缀表达式求值函数在代码 2 中列出。为了帮助计算，我们定义了一个辅助函数 doMath 来接受两个操作数和一个运算符并进行合理的计算操作。

```
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 + /'))
```

代码 29 后缀表达式求值 (postfixeval)

这里需要强调的是，后缀表达式和前缀表达式的求值程序是仅在我们输入了没有错误的表达式的前提下可正确运行。以这些程序为起点，你可以很容易地观察到检错和反馈功能是如何被容纳进来的。我们将这个程序作为本章末尾的一个练习。

自测

Q-12. 在不利用已有的动态代码 infixToPostfixd 的前提下，将以下表达式转化为后缀表达式：
10 + 3 * 5 / (16 - 4)
Q-13. 17 10 + 3 * 9 / =
Q-14. 建立一个 infixToPostfix 函数，使其可以转化下列表达式 5 * 3 ^ (4 - 2) ，请将答案粘贴在下边。