数据结构与算法（Python版）

什么是线性结构

陈斌 北京大学 gischen@pku.edu.cn

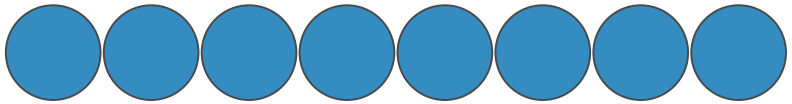
什么是线性结构Linear Structure

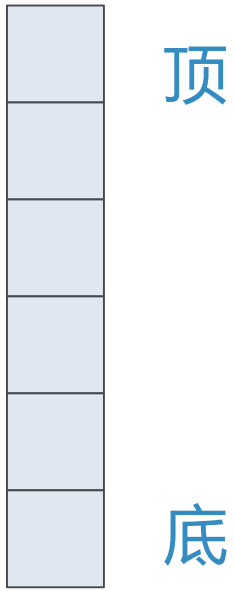
- 线性结构是一种有序数据项的集合，其中每个数据项都有**唯一**的前驱和后继
  - 除了第一个没有前驱，最后一个没有后继
  - 新的数据项加入到数据集中时，只会加入到原有某个数据项之前或之后
  - 具有这种性质的数据集，就称为线性结构

什么是线性结构Linear Structure

- 线性结构总有两端，在不同的情况下，两端的称呼也不同

  有时候称为“左”“右”端、“前”“后”端、“顶”“底”端

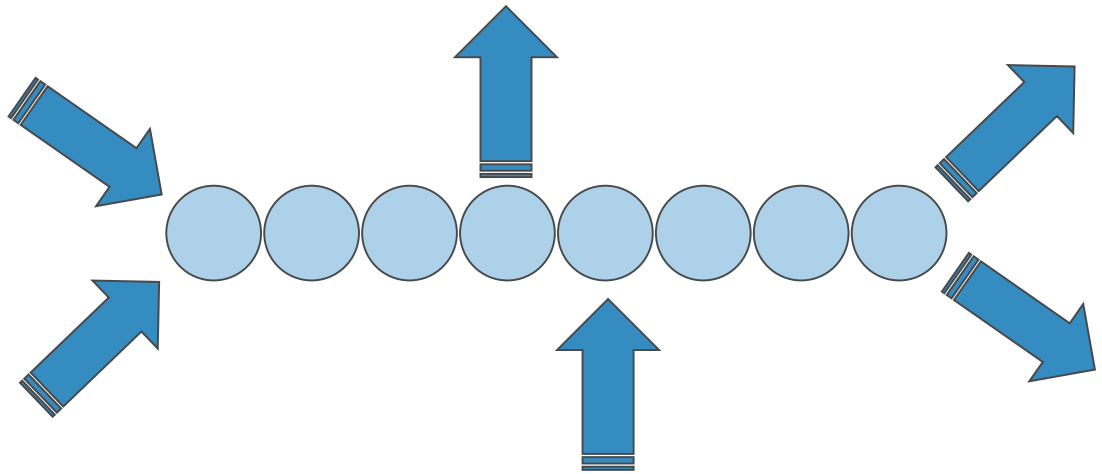




什么是线性结构Linear Structure

- **两端的称呼并不是关键，不同线性结构的关键区别在于数据项增减的方式**

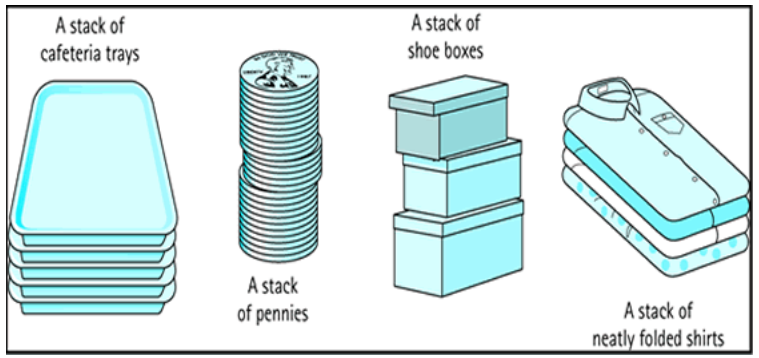
  有的结构只允许数据项从一端添加，而有的结构则允许数据项从两端移除



什么是线性结构Linear Structure

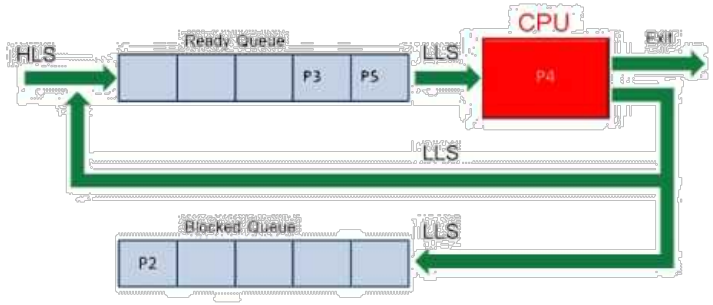
- 我们从4个最简单但功能强大的结构入手，开始研究数据结构
- 栈Stack，队列Queue，双端队列Deque和列表List

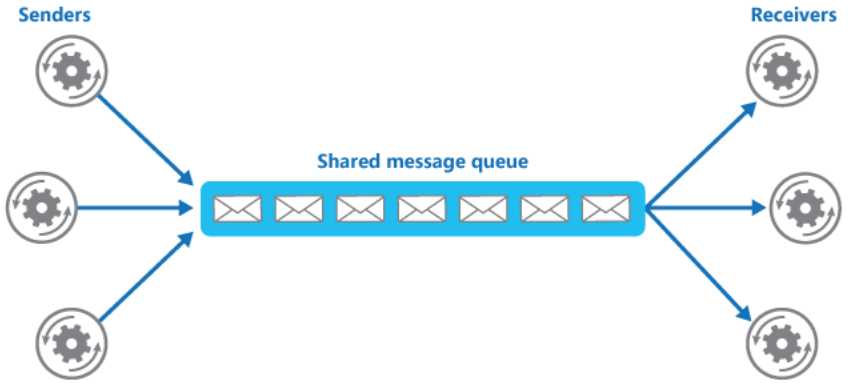
  这些数据集的共同点在于，数据项之间只存在先后的次序关系，都是线性结构

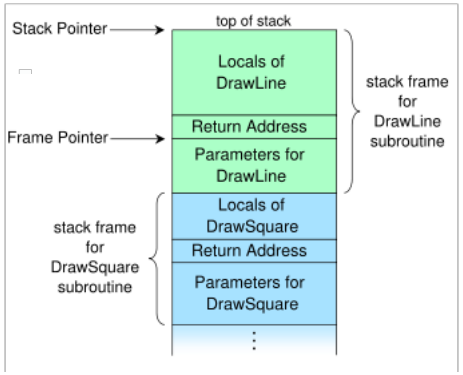


什么是线性结构Linear Structure

- 这些线性结构是应用最广泛的数据结构，
- 它们出现在各种算法中，用来解决大量重要问题







数据结构与算法（Python版）

栈抽象数据类型及Python实现

陈斌 北京大学 gischen@pku.edu.cn

栈Stack：什么是栈?

- 一种有次序的数据项集合，在栈中，数据项的加入和移除都仅发生在同一端
  - 这一端叫栈“顶top”，另一端叫栈“底base”
- 日常生活中有很多栈的应用
  - 盘子、托盘、书堆等等

栈Stack：什么是栈?

- 距离栈底越近的数据项，留在栈中的时间就越长

  而最新加入栈的数据项会被最先移除

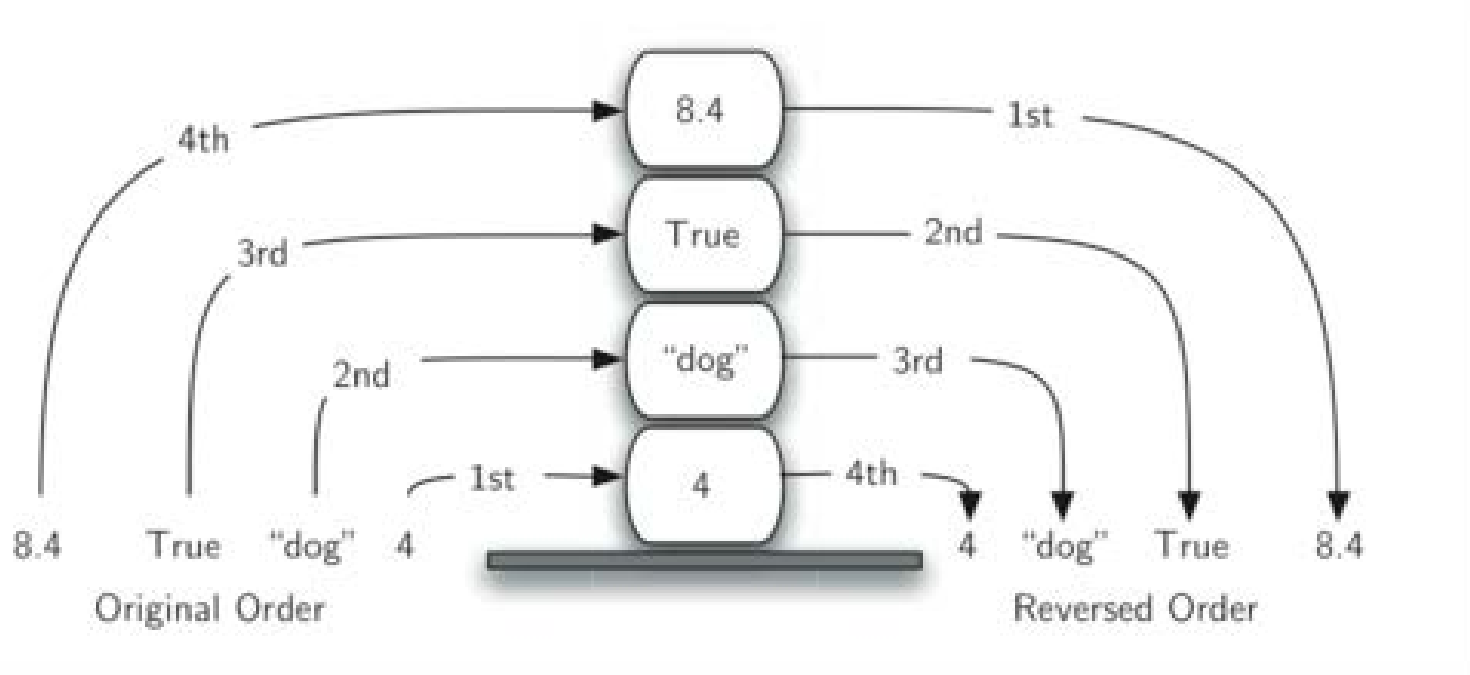
- 这种次序通常称为“后进先出LIFO”：Last in First out

  这是一种基于数据项保存时间的次序，时间越短的离栈顶越近，而时间越长的离栈底越近

栈的特性：反转次序

- 我们观察一个由混合的python原生数据对象形成的栈

  进栈和出栈的次序正好相反



栈的特性：反转次序

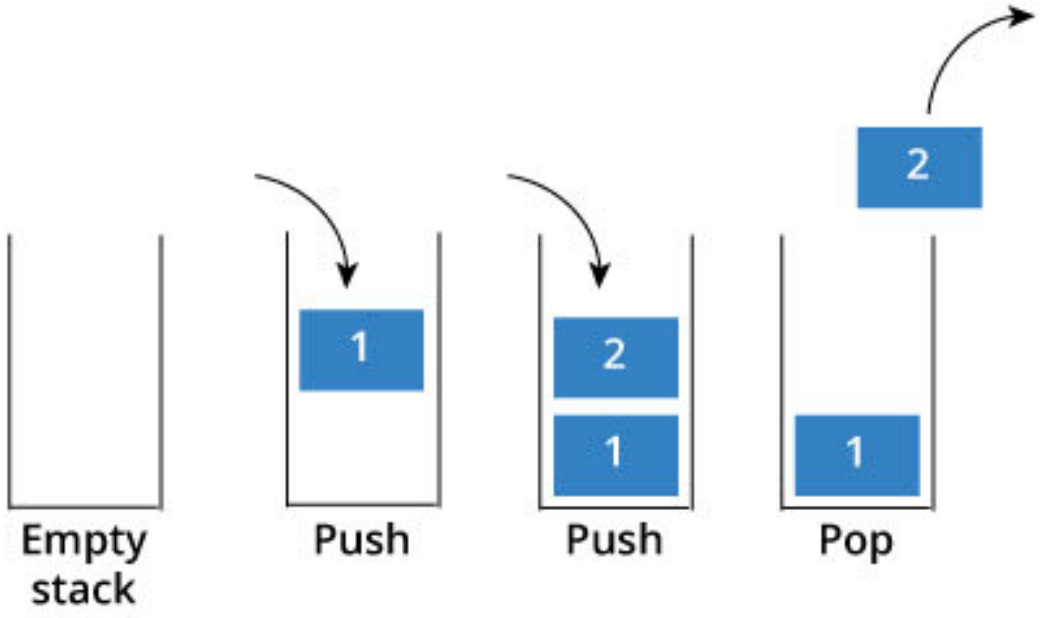
- 这种访问次序反转的特性，我们在某些计算机操作上碰到过

浏览器的“后退back”按钮，最先back的是最近访问的网页

Word的“Undo”按钮，最先撤销的是最近操作

抽象数据类型Stack

- 抽象数据类型“栈”是一个有次序的数据集，每个数据项仅从“栈顶”一端加入到数据集中、从数据集中移除，栈具有后进先出LIFO的特性



抽象数据类型Stack

- 抽象数据类型“栈”定义为如下的操作

  Stack()：创建一个空栈，不包含任何数据项

  push(item)：将item加入栈顶，无返回值

  pop()：将栈顶数据项移除，并返回，栈被修改

  peek()：“窥视”栈顶数据项，返回栈顶的数据项但不移除，栈不被修改

  isEmpty()：返回栈是否为空栈

  size()：返回栈中有多少个数据项

抽象数据类型Stack：操作样例

Stack Operation	Stack Contents	Return Value
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

用Python实现ADT Stack

- 在清楚地定义了抽象数据类型Stack之后，我们看看如何用Python来实现它
- Python的面向对象机制，可以用来实现用户自定义类型

  将ADT Stack实现为Python的一个Class

  将ADT Stack的操作实现为Class的方法

  由于Stack是一个数据集，所以可以采用Python的原生数据集来实现，我们选用最常用的数据集List来实现

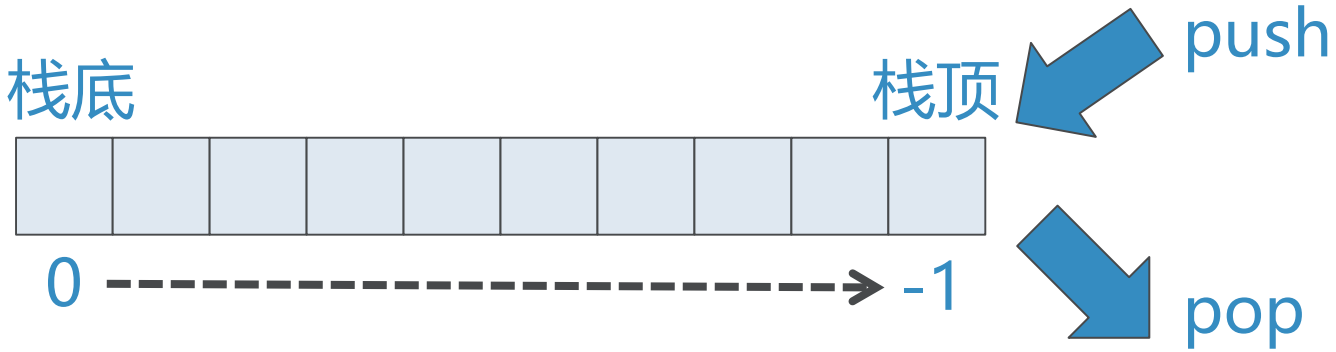
用Python实现ADT Stack

- **一个细节：Stack的两端对应list设置**

  可以将List的任意一端（index=0或者-1）设置为栈顶

  我们选用List的末端（index=-1）作为栈顶

  这样栈的操作就可以通过对list的append和pop来实现，很简单！



用Python实现ADT Stack

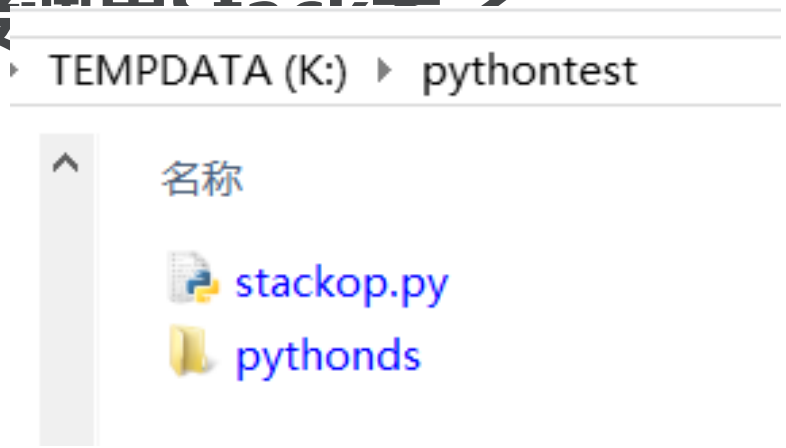

```
class Stack:
    def __init__(self):
        self.items = []

    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)
```

课程配套代码：pythonds模块

- 解包拷贝到练习目录下，与练习程序**平级**
- 调用方法

```
from pythonds.basic.stack import Stack
```

- 在与pythonds目录平级的stackop.py文件里面，按照上面的import导入后，就可以直接调用Stack类了



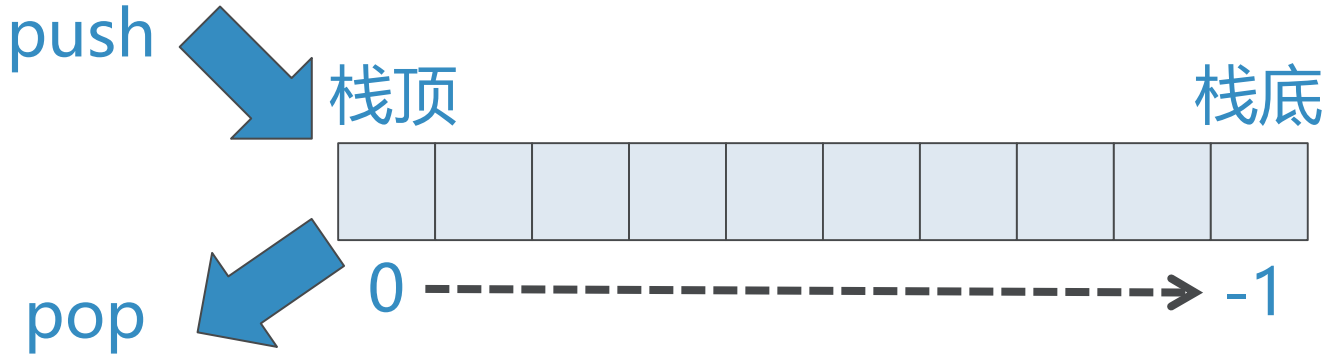
Stack测试代码


```
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())
```

```
>>> ======
>>>
True
dog
3
False
8.4
True
2
>>>
```

ADT Stack的另一个实现

- 如果我们把List的另一端（首端index=0）作为Stack的栈顶，同样也可以实现Stack



ADT Stack的另一个实现

- 不同的实现方案保持了ADT接口的稳定性

  但性能有所不同，栈顶首端的版本（左），其push/pop的复杂度为O(n)，而栈顶尾端的实现（右），其push/pop的复杂度为O(1)

```
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)
```

```
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)
```

数据结构与算法（Python版）

栈的应用：简单括号匹配

陈斌 北京大学 gischen@pku.edu.cn

栈的应用：简单括号匹配

- 我们都写过这样的表达式：

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

  这里的括号是用来指定表达式项的计算优先级

- 有些函数式语言，如Lisp，在函数定义的时候会用到**大量**的括号

  比如：(defun square(n)

  (* n n))

  这个语句定义了一个计算平方值的函数

栈的应用：简单括号匹配

- 当然，括号的使用必须遵循 “平衡”规则

  首先，每个开括号要恰好对应一个闭括号；

  其次，每对开闭括号要正确的嵌套

  正确的括号：(()()()())，((((()))))，
  (()((())()))

  错误的括号：((((((())，()))，(()()(()

- 对括号是否正确匹配的识别，是很多语言编译器的基础算法

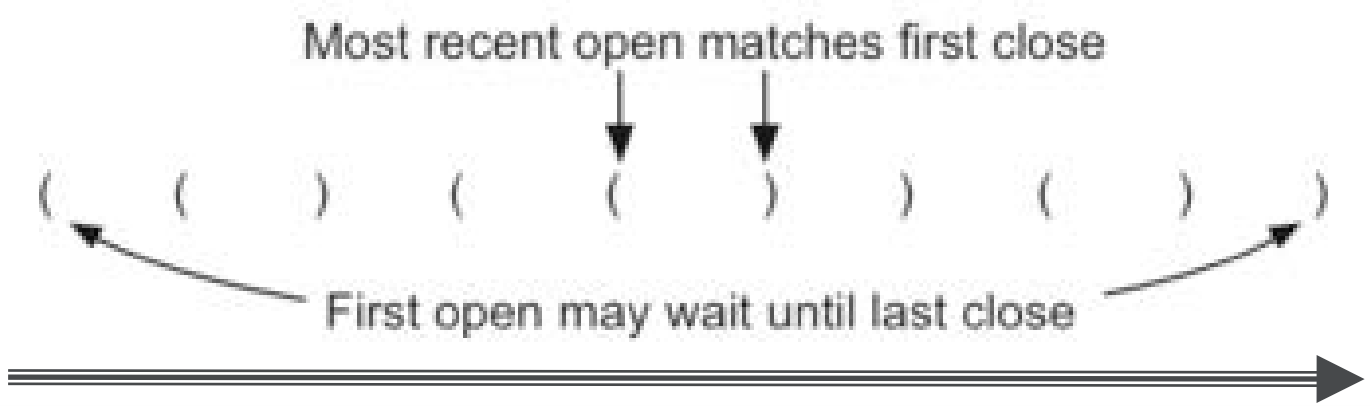
栈的应用：简单括号匹配

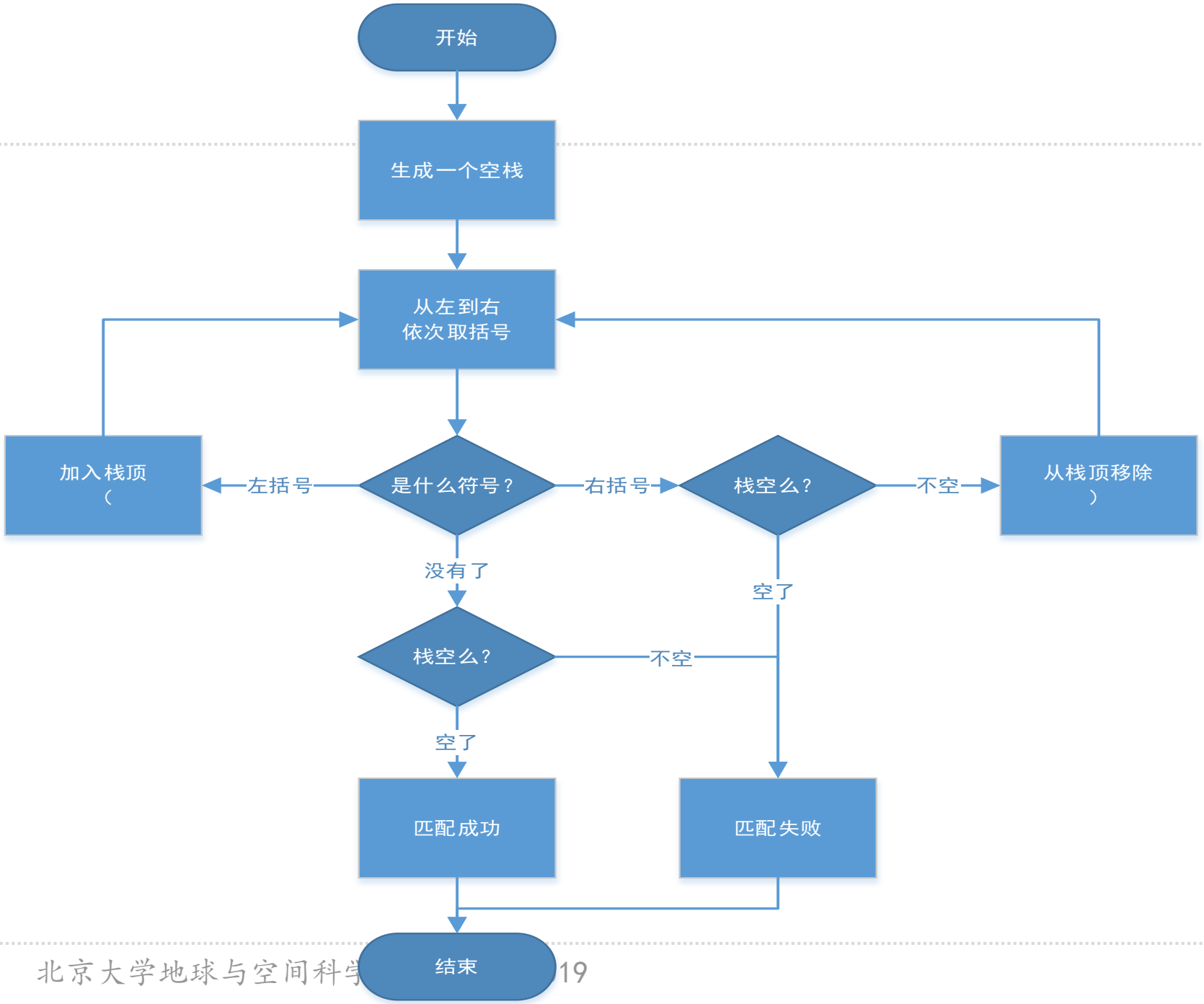
- 下面看看如何构造括号匹配识别算法

从左到右扫描括号串，最新打开的左括号，应该匹配最先遇到的右括号

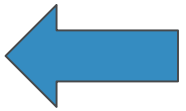
这样，第一个左括号（最早打开），就应该匹配最后一个右括号（最后遇到）

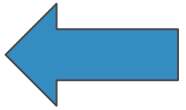
这种次序反转的识别，正好符合栈的特性！




开始
生成一个空栈
从左到右
依次取括号
是什么符号？
左括号
加入栈顶
（
右括号
栈空么？
不空
从栈顶移除
）
没有了
空了
栈空么？
不空
空了
匹配成功
匹配失败
结束

```
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('(()'))
```





更多种括号的匹配

- 在实际的应用里，我们会碰到更多种括号
  - 如python中列表所用的方括号“[]”
  - 字典所用的花括号“{}”
  - 元组和表达式所用的圆括号“()”
- 这些不同的括号有可能混合在一起使用，
- 因此就要注意各自的开闭匹配情况

更多种括号的匹配

- 下面这些是匹配的

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

- 下面这些是不匹配的

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

通用括号匹配算法：代码

- 需要修改的地方

碰到各种左括号仍然入栈

碰到各种右括号的时候需要判断栈顶的左括号是否跟之同一种类

```
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
```

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

通用括号匹配算法

- HTML/XML文档也有类似于括号的开闭标记，这种层次结构化文档的校验、操作也可以通过栈来实现

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Hello World</title>
6  </head>
7  <body>
8  <h1>Hello!</h1>
9  </body>
10 </html>
```

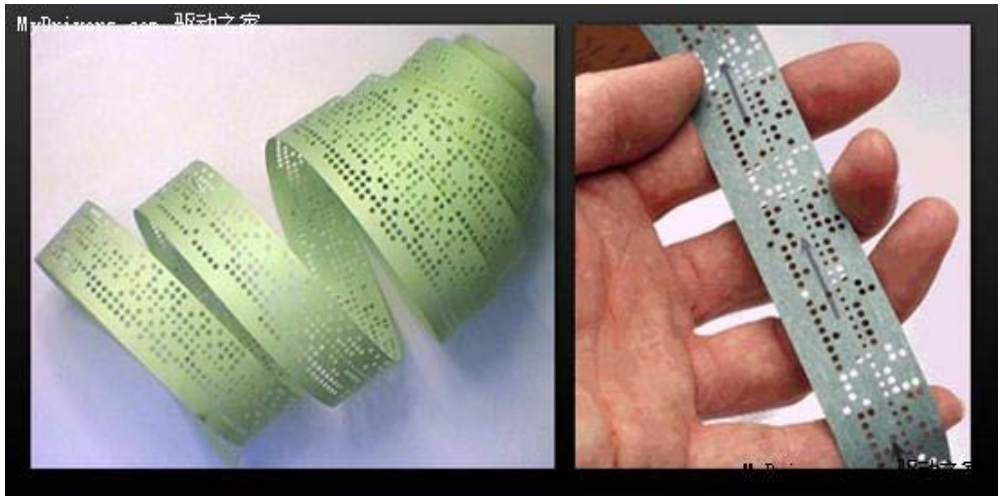
数据结构与算法（Python版）

栈的应用：十进制转换为二进制

陈斌 北京大学 gischen@pku.edu.cn

十进制转换为二进制

- 二进制是计算机原理中最基本的概念，作为组成计算机最基本部件的逻辑门电路，其输入和输出均仅为两种状态：0和1
- 但十进制是人类传统文化中最基本的数值概念，如果没有进制之间的转换，人们跟计算机的交互会相当的困难



十进制转换为二进制

- 所谓的“进制”，就是用多少个字符来表示整数

  十进制是0～9这十个数字字符，二进制是0、1两个字符

- 我们经常需要将整数在二进制和十进制之间转换

  如：$(233)_{10}$的对应二进制数为$(11101001)_2$，

  具体是这样：

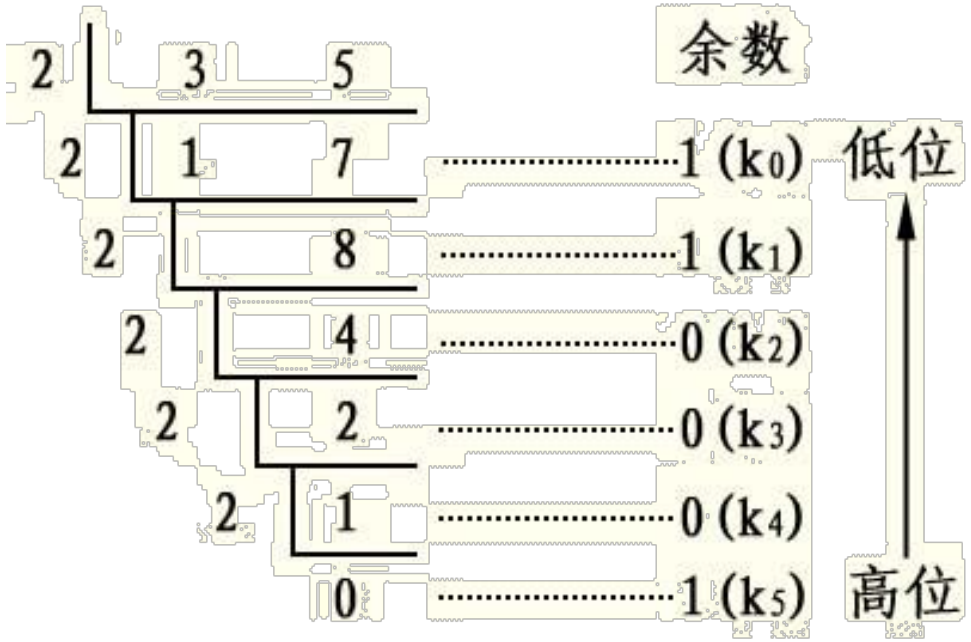
  $(233)_{10}=2\times10^2+3\times10^1+3\times10^0$

  $(11101001)_2=1\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$

十进制转换为二进制

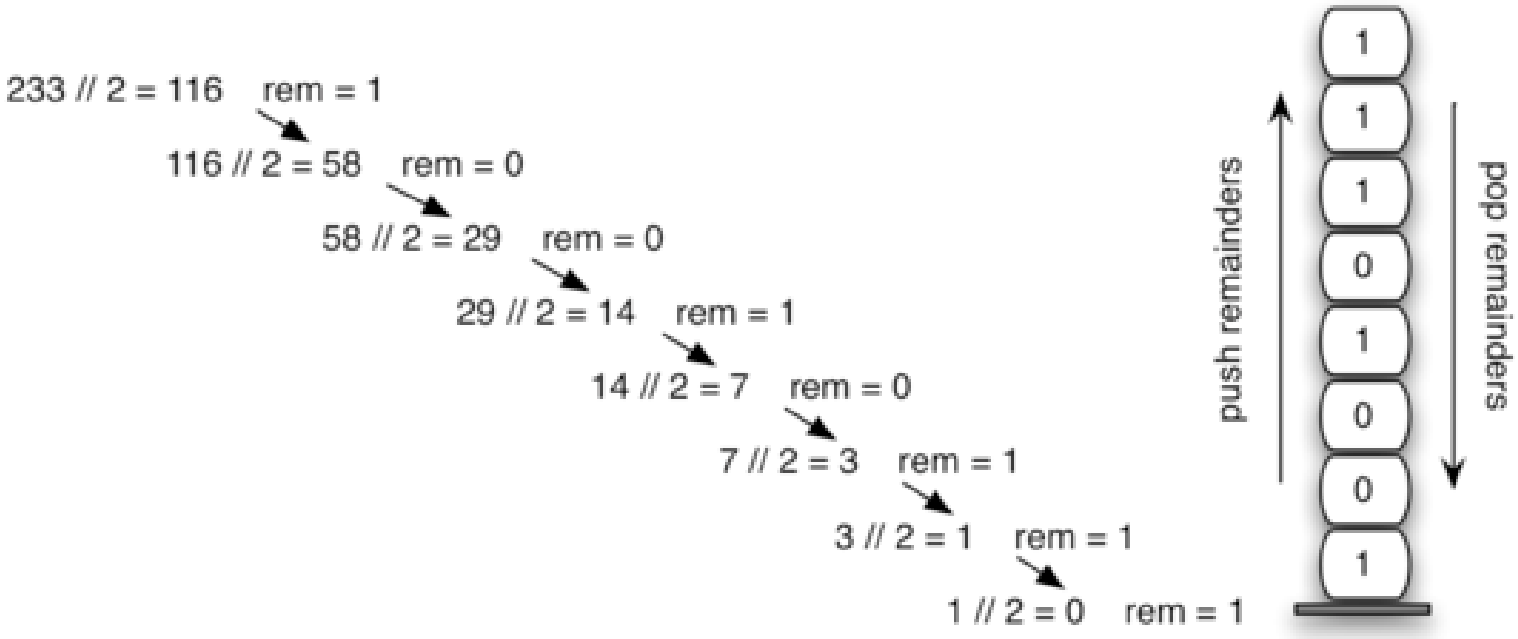
- 十进制转换为二进制，采用的是“**除以2求余数**”的算法

  将整数不断除以2，每次得到的余数就是由低到高的二进制位



十进制转换为二进制

- “除以2”的过程，得到的余数是从低到高的次序，而输出则是从高到低，所以需要一个栈来反转次序



十进制转换为二进制：代码


```
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))
```

求余数

整数除

扩展到更多进制转换

- 十进制转换为二进制的算法，很容易可以扩展为转换到任意N进制

  只需要将“除以2求余数”算法改为“除以N求余数”算法就可以

- 计算机中另外两种常用的进制是八进制和十六进制

  $(233)_{10}$等于$(351)_{8}$和$(E9)_{16}$

  $(351)_{8}=3\times8^{2}+5\times8^{1}+1\times8^{0}$

  $(E9)_{16}=14\times16^{1}+9\times16^{0}$

扩展到更多进制转换

- 主要的问题是如何表示八进制及十六进制

  二进制有两个不同数字0、1

  十进制有十个不同数字0、1、2、3、4、5、6、7、8、9

  八进制可用八个不同数字0、1、2、3、4、5、6、7

  十六进制的十六个不同数字则是0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F

十进制转换为十六以下任意进制：代码


```
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))
```

数据结构与算法（Python版）

表达式转换（上）

陈斌 北京大学 gischen@pku.edu.cn

中缀表达式

- 我们通常看到的表达式象这样：B*C，很容易知道这是B乘以C
- 这种操作符（operator）介于操作数（operand）中间的表示法，称为“**中缀**”表示法
- 但有时候中缀表示法会引起混淆，如“**A+B*C**”

  是A+B然后再乘以C

  还是B*C然后再去加A?

中缀表达式中的优先级

- 人们引入了操作符“**优先级**”的概念来消除混淆
  - 规定高优先级的操作符先计算
  - 相同优先级的操作符从左到右依次计算
  - 这样A+B*C就没有疑义是A加上B与C的乘积
- 同时引入了**括号**来表示**强制**优先级，括号的优先级最高，而且在嵌套的括号中，内层的优先级更高
  - 这样(A+B)*C就是A与B的和再乘以C

全括号中缀表达式

- 虽然人们已经习惯了这种表示法，但计算机处理最好是能明确规定所有的计算顺序，这样无需处理复杂的优先规则
- 引入**全括号表达式**：在所有的表达式项两边都加上括号

  A+B*C+D，应表示为((A+(B*C))+D)

- 可否将表达式中操作符的位置稍**移动**一下？

前缀和后缀表达式

- 例如中缀表达式A+B

  将操作符移到前面，变为“+AB”

  或者将操作符移到最后，变为“AB+”

- 我们就得到了表达式的另外两种表示法：

  “**前缀**”和“**后缀**”表示法

  以操作符相对于操作数的位置来定义

前缀和后缀表达式

- 这样A+B*C将变为前缀的“+A*BC”，后缀的“ABC*+”

  为了帮助理解，子表达式加了下划线

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 * +

前缀、中缀和后缀表达式

- 再来看中缀表达式“(A+B)*C”，按照转换的规则，前缀表达式是“*+ABC”，而后缀表达式是“AB+C*”
- 神奇的事情发生了，在中缀表达式里必须的括号，在前缀和后缀表达式中消失了?
- 在前缀和后缀表达式中，**操作符的次序**完全决定了运算的次序，不再有混淆

  所以在很多情况下，表达式的计算机表示都避免用复杂的中缀形式

前缀、中缀和后缀表达式

❖ 下面看更多的例子

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 +

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的形式

  一定得有个算法来转换任意复杂的表达式

- 为了分解算法的复杂度，我们从“全括号”中缀表达式入手

  我们看A+B*C，如果写成全括号形式：

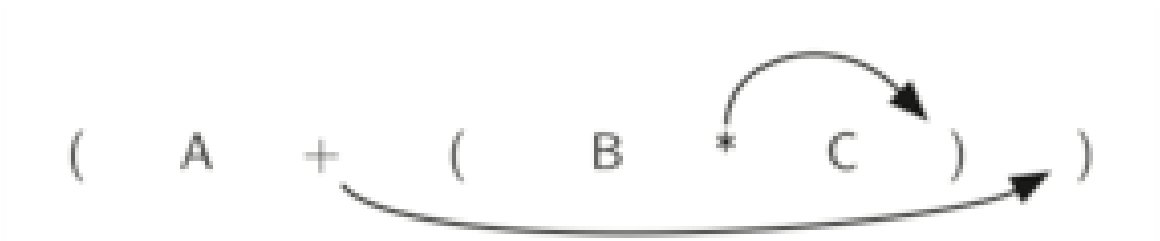
  (A+(B*C))，显式表达了计算次序

  我们注意到每一对括号，都包含了一组完整的操作符和操作数

中缀表达式转换为前缀和后缀形式

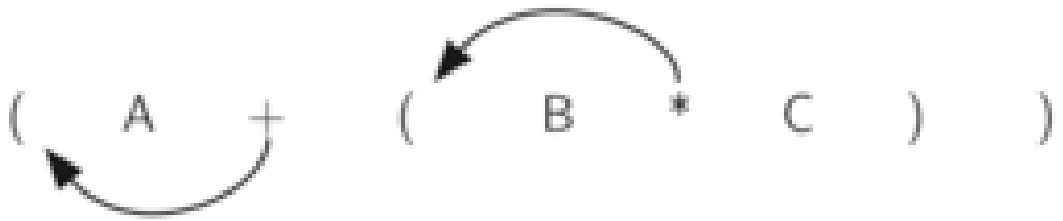
- 看子表达式(B*C)的**右括号**，如果把操作符*移到右括号的位置，替代它，再删去左括号，得到BC*，这个正好把子表达式转换为**后缀**形式

  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



中缀表达式转换为前缀和后缀形式

- 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

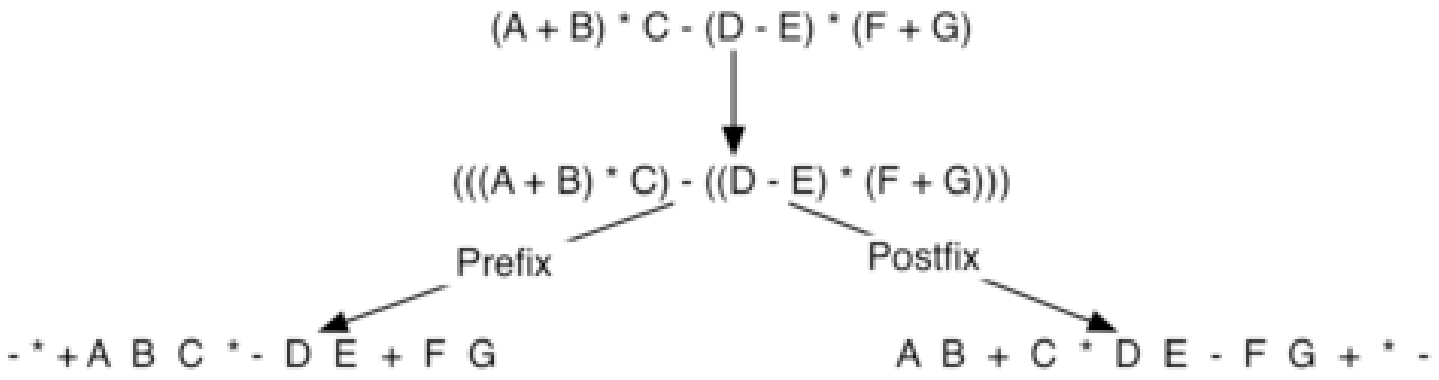


中缀表达式转换为前缀和后缀形式

- 所以说，无论表达式多复杂，需要转换成前缀或者后缀，只需要两个步骤

  将中缀表达式转换为全括号形式

  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



数据结构与算法（Python版）

表达式转换（下）

陈斌 北京大学 gischen@pku.edu.cn

通用的中缀转后缀算法

- 我们来讨论下通用的中缀转后缀算法
- 首先我们来看中缀表达式A+B*C，其对应的后缀表达式是ABC*+

  操作数ABC的顺序没有改变。

  操作符的出现顺序，在后缀表达式中反转了

  由于*的优先级比+高，所以后缀表达式中操作符的出现顺序与运算次序一致。

通用的中缀转后缀算法

- 在中缀表达式转换为后缀形式的处理过程中，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，需要把操作符先保存起来

- 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。

  在A+B*C中，+虽然先出现，但优先级比后面这个*要低，所以它要等*处理完后，才能再处理。

- 这种**反转**特性，使得我们考虑用**栈**来保存暂时**未处理**的**操作符**

通用的中缀转后缀算法

- 再看看(A+B)*C，对应的后缀形式是AB+C*

  这里+的输出比*要早，主要是因为括号使得+的优先级提升，高于括号之外的*

- 回顾上节的“**全括号**”表达式，后缀表达式中操作符应该出现在左括号对应的右括号位置

  所以遇到左括号，要标记下，其后出现的操作符优先级提升了，一旦扫描到对应的右括号，就可以马上输出这个操作符

通用的中缀转后缀算法

- 总结下，在从左到右扫描逐个字符扫描中缀表达式的过程中，采用一个**栈**来暂存未处理的操作符
- 这样，**栈顶**的操作符就是**最近**暂存进去的，当遇到一个新的操作符，就需要跟栈顶的操作符比较下优先级，再行处理。

通用的中缀转后缀算法：流程

- 后面的算法描述中，约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*/+-()
  而操作数单词则是单字母标识符A、B、C等。
- 首先，创建空栈opstack用于**暂存操作符**，空表postfixList用于保存后缀表达式
- 将中缀表达式转换为单词（token）列表

```
A+B*C =split=> ['A', '+', 'B', '*', 'C']
```

通用的中缀转后缀算法：流程

❖ 从左到右扫描中缀表达式单词列表

如果单词是操作数，则直接添加到后缀表达式列表的末尾

如果单词是左括号“(”，则压入opstack栈顶

如果单词是右括号“)”，则反复弹出opstack栈顶操作符，加入到输出列表末尾，直到碰到左括号

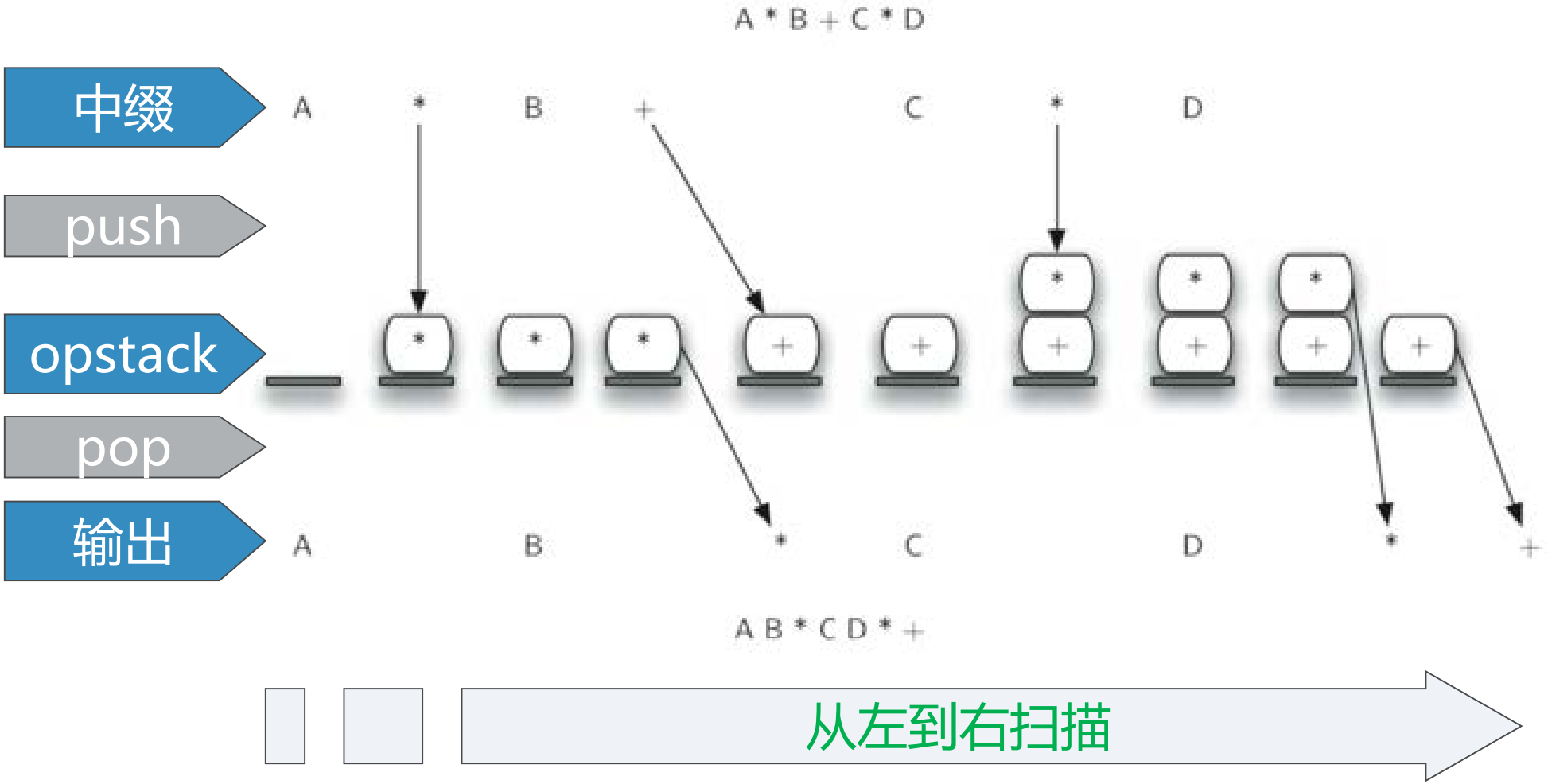
如果单词是操作符“*/+-”，则压入opstack栈顶

- 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

通用的中缀转后缀算法：流程

- 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**
- 把输出列表再用join方法合并成后缀表达式字符串，算法结束。

通用的中缀转后缀算法：实例



代码


```
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)
```

记录操作符优先级

解析表达式到单词列表

操作数

(

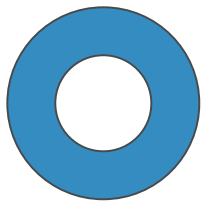
)

操作符

操作符

合成后缀表达式字符串

通用的中缀转后缀算法



数据结构与算法（Python版）

后缀表达式求值

陈斌 北京大学 gischen@pku.edu.cn

后缀表达式求值

- 作为栈结构的结束，我们来讨论“**后缀表达式求值**”问题
- 跟中缀转换为后缀问题不同，
- 在对后缀表达式从左到右扫描的过程中，
- 由于操作符在操作数的**后面**，
- 所以要**暂存操作数**，在碰到操作符的时候，再将暂存的两个操作数进行实际的计算

  仍然是栈的特性：操作符只作用于离它最近的两个操作数

后缀表达式求值

- 如“4 5 6 * +”，我们先扫描到4、5两个操作数
- 但还不知道对这两个操作数能做什么计算，需要继续扫描后面的符号才能知道
- 继续扫描，又碰到操作数6
- 还是不能知道如何计算，继续暂存入栈
- 直到“*”，现在知道是栈顶两个操作数5、6做乘法

后缀表达式求值

- 我们弹出两个操作数，计算得到结果30

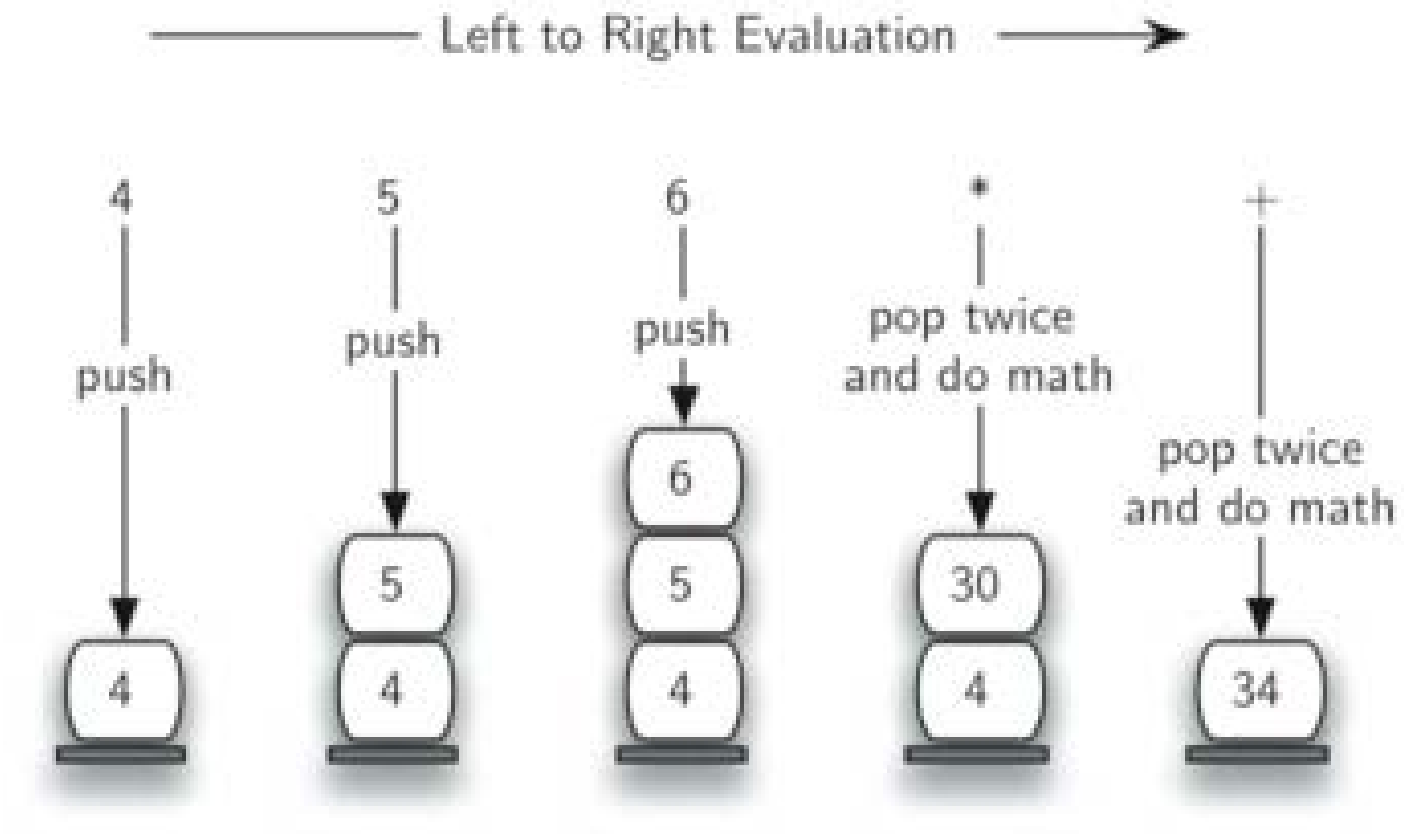
  需要注意：

  先弹出的是右操作数

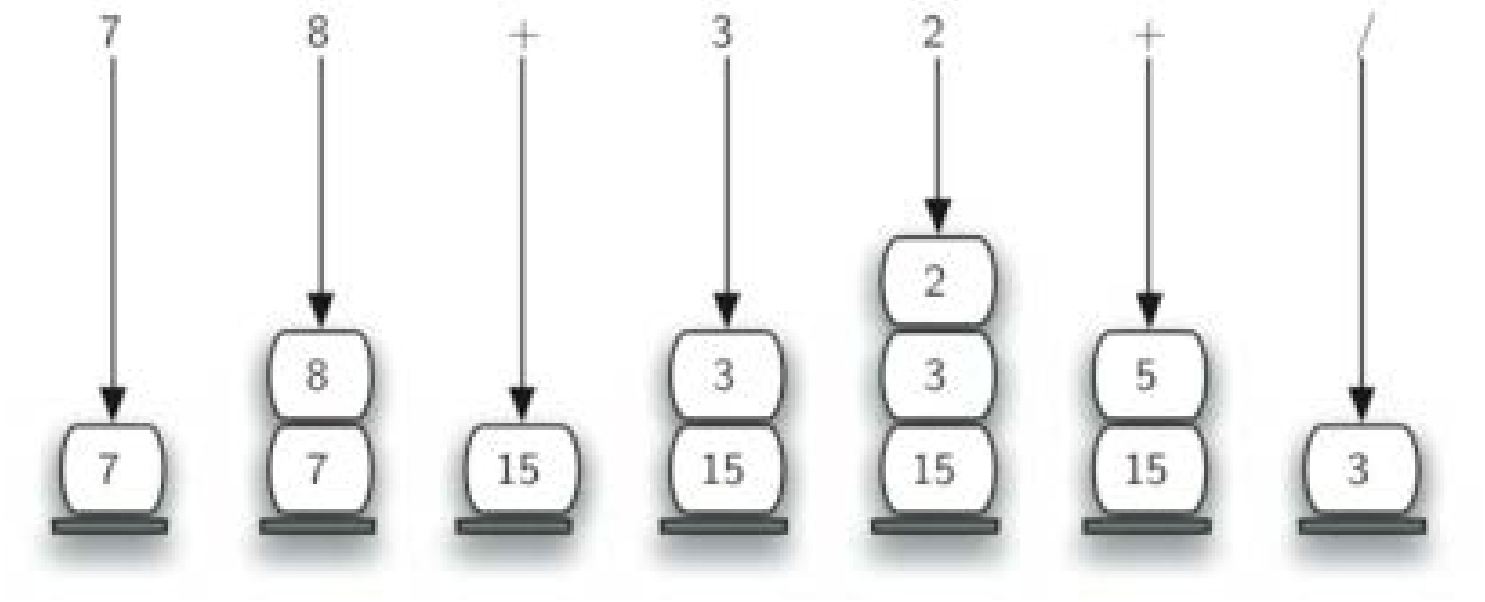
  后弹出的是左操作数，这个对于-/很重要！

- 为了继续后续的计算，需要把这个中间结果30压入栈顶
- 继续扫描后面的符号
- 当所有操作符都处理完毕，栈中只留下1个操作数，就是表达式的值

后缀表达式求值：实例


Left to Right Evaluation
4
push
4
5
push
5
4
6
push
6
5
4
*
pop twice
and do math
30
4
+
pop twice
and do math
34

后缀表达式求值：实例



后缀表达式求值：流程

- 创建空栈operandStack用于**暂存操作数**
- 将后缀表达式用split方法解析为单词（token）的列表
- 从左到右扫描单词列表

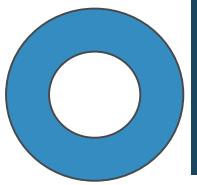
  如果单词是一个操作数，将单词转换为整数int，压入operandStack栈顶

  如果单词是一个操作符（*/+-），就开始求值，从栈顶弹出2个操作数，先弹出的是右操作数，后弹出的是左操作数，计算后将值重新压入栈顶
- 单词列表扫描结束后，表达式的值就在栈顶
- 弹出栈顶的值，返回。

```
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
```

操作数

操作符



数据结构与算法（Python版）

队列抽象数据类型及Python实现

陈斌 北京大学 gischen@pku.edu.cn

队列Queue：什么是队列?

- 队列是一种有次序的数据集合，其特征是

  新数据项的添加总发生在一端（通常称为“尾rear”端）

  而现存数据项的移除总发生在另一端（通常称为“首front”端）

- 当数据项加入队列，首先出现在队尾，随着队首数据项的移除，它逐渐接近队首。

队列Queue：什么是队列?

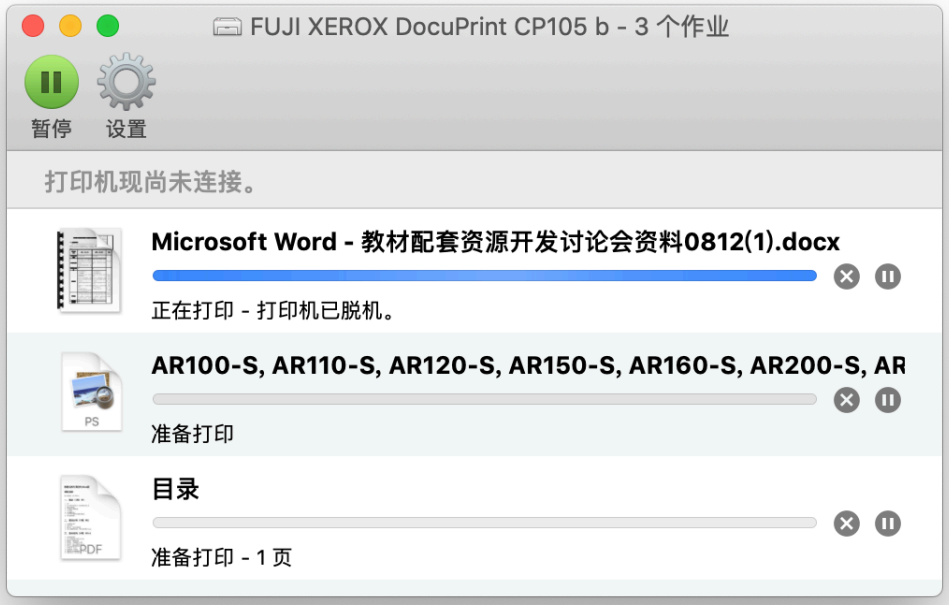
- 新加入的数据项必须在数据集末尾等待，而等待时间最长的数据项则是队首
- 这种次序安排的原则称为（FIFO:First-in first-out）先进先出
  或“先到先服务first-come first-served”
- 队列的例子出现在我们日常生活的方方面面：排队
- 队列仅有一个入口和一个出口
  不允许数据项直接插入队中，也不允许从中间移除数据项

计算机科学中队列的例子：打印队列

- 一台打印机面向多个用户/程序提供服务

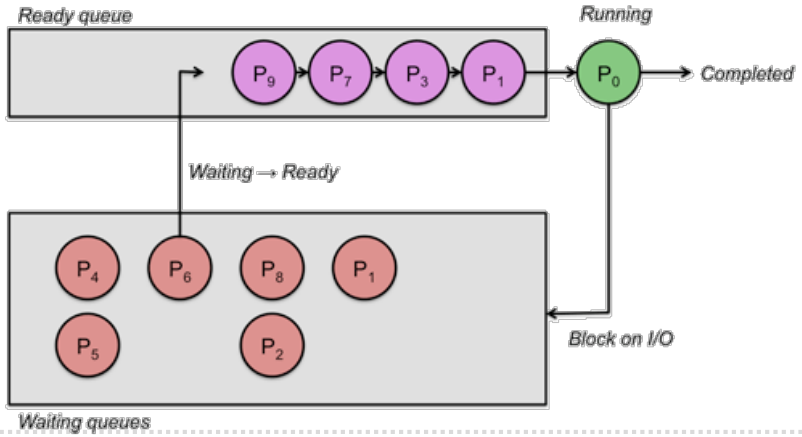
  打印速度比打印请求提交的速度要慢得多

  有任务正在打印时，后来的打印请求就要排成队列，以FIFO的形式等待被处理。



计算机科学中队列的例子：进程调度

- 操作系统核心采用多个队列来对系统中同时运行的进程进行调度
  - 进程数远多于CPU核心数
  - 有些进程还要等待不同类型I/O事件
- 调度原则综合了“先到先服务”及“资源充分利用”两个出发点

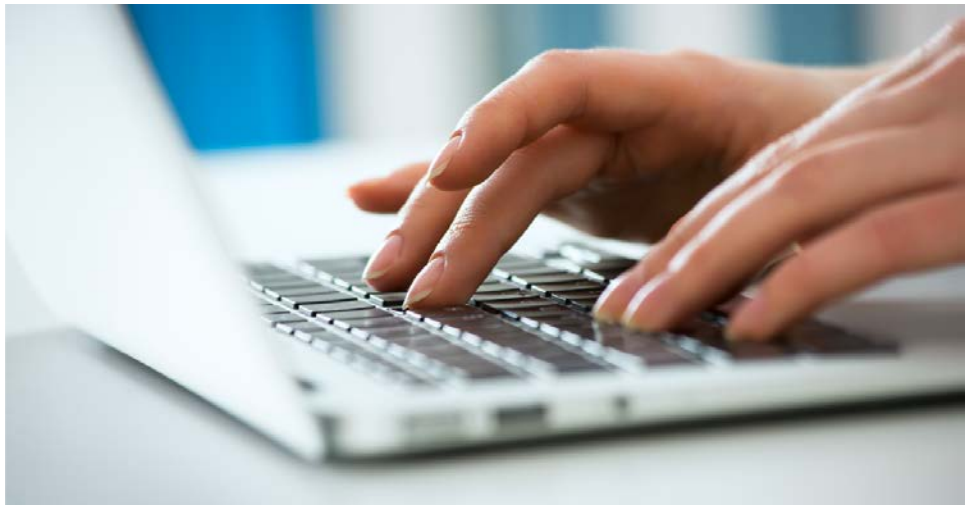


计算机科学中队列的例子：键盘缓冲

- 键盘敲击并不马上显示在屏幕上

  需要有个队列性质的缓冲区，将尚未显示的敲击字符暂存其中，

  队列的先进先出性质则保证了字符的输入和显示次序一致性。



抽象数据类型Queue

- 抽象数据类型Queue是一个有次序的数据集合
  - 数据项仅添加到“尾rear”端
  - 而且仅从“首front”端移除
  - Queue具有FIFO的操作次序

抽象数据类型Queue

- **抽象数据类型Queue由如下操作定义：**

  Queue()：创建一个空队列对象，返回值为Queue对象；

  enqueue(item)：将数据项item添加到队尾，无返回值；

  dequeue()：从队首移除数据项，返回值为队首数据项，队列被修改；

  isEmpty()：测试是否空队列，返回值为布尔值

  size()：返回队列中数据项的个数。

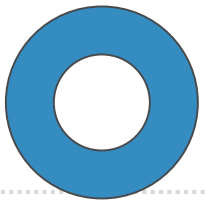
抽象数据类型Queue

队列操作	队列内容	返回值
q=Queue()	[]	Queue object
q.isEmpty()	[]	True
q.enqueue(4)	[4]	
q.enqueue('dog')	['dog',4]	
q.enqueue(True)	[True,'dog',4]	
q.size()	[True,'dog',4]	3
q.isEmpty()	[True,'dog',4]	False
q.enqueue(8.4)	[8.4,True,'dog',4]	
q.dequeue()	[8.4,True,'dog']	4
q.dequeue()	[8.4,True]	'dog'
q.size()	[8.4,True]	2

Python实现ADT Queue

- 采用List来容纳Queue的数据项
  - 将List首端作为队列尾端
  - List的末端作为队列首端
  - enqueue()复杂度为O(n)
  - dequeue()复杂度为O(1)
- 首尾倒过来的实现，复杂度也倒过来

```
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)
```



数据结构与算法（Python版）

队列的应用：热土豆

陈斌 北京大学 gischen@pku.edu.cn

热土豆问题（约瑟夫问题）

- “击鼓传花”的土豆版本
- 传烫手的热土豆，鼓声停的时候，手里有土豆的小孩就要出列。

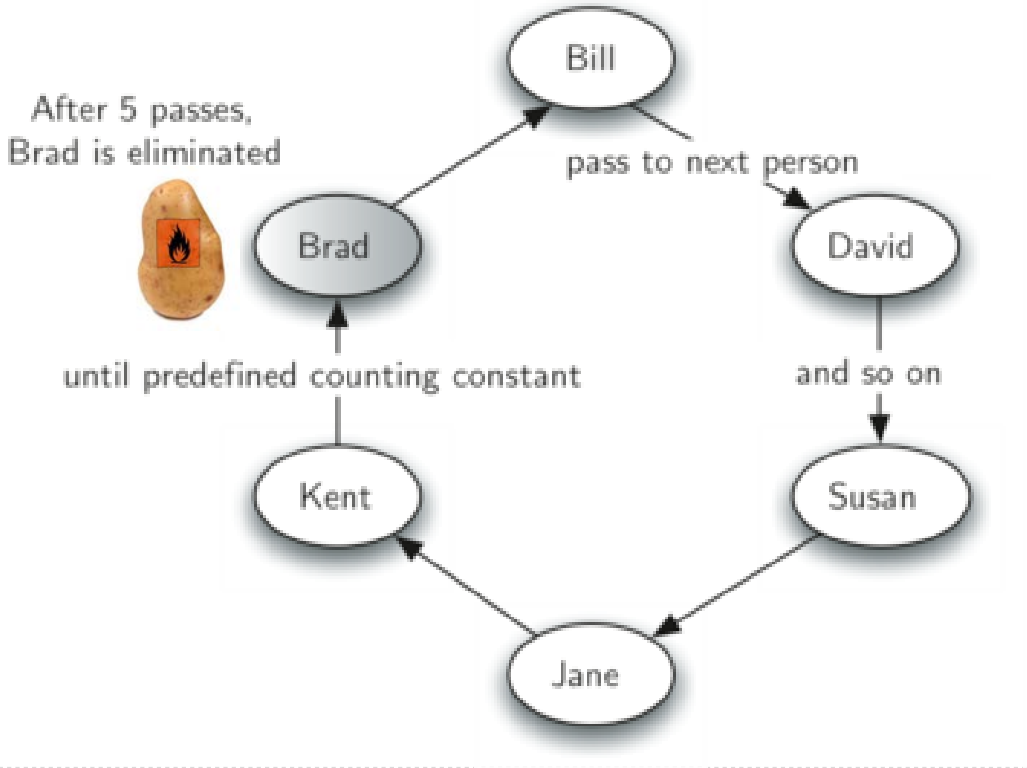
热土豆问题（约瑟夫问题）

- 如果去掉鼓，改为传过固定人数，就成了“现代版”的约瑟夫问题

传说犹太人反叛罗马人，落到困境，约瑟夫和39人决定殉难，坐成一圈儿，报数1～7，报到7的人由旁边杀死，结果约瑟夫给自己安排了个位置，最后活了下来……

热土豆问题：算法

- 用队列来实现热土豆问题的算法，参加游戏的人名列表，以及传土豆次数num，算法返回最后剩下的人名



热土豆问题：算法

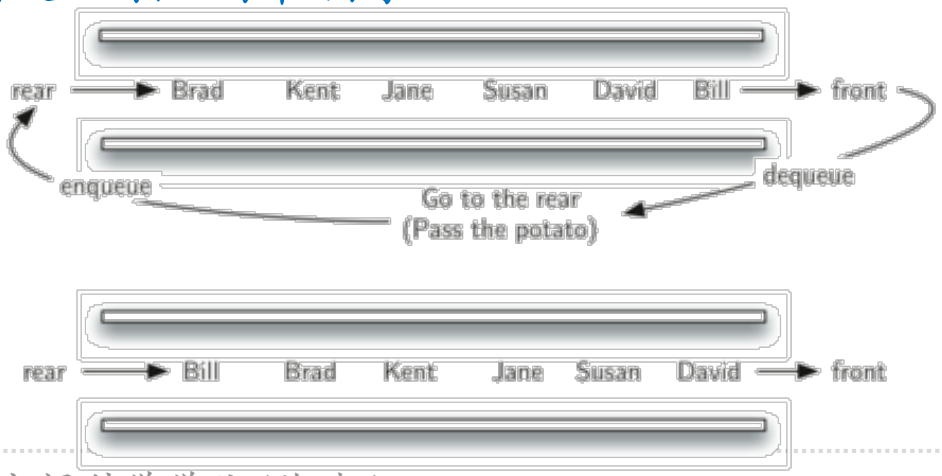
- 模拟程序采用队列来存放所有参加游戏的人名，按照传递土豆方向从队首排到队尾

  游戏时，队首始终是持有土豆的人

- 模拟游戏开始，只需要将队首的人出队，随即再到队尾入队，算是土豆的一次传递

  传递了num次后，将队首的人移除，不再入队

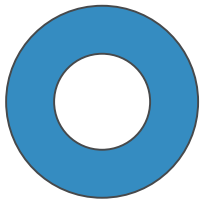
  如此反复，直到队列中剩余1人



热土豆问题：代码


```
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))
```

一次传递



数据结构与算法（Python版）

队列的应用：打印任务（上）

陈斌 北京大学 gischen@pku.edu.cn

模拟算法：打印任务

- 多人共享一台打印机，采取“先到先服务”的队列策略来执行打印任务
- 在这种设定下，一个首要的问题就是：

  这种打印作业系统的容量有多大？

  在能够接受的等待时间内，系统能容纳多少用户

  以多高频率提交多少打印任务？

模拟算法：打印任务

- **一个具体的实例配置如下：**

  一个实验室，在任意的一个小时内，大约有10名学生在场，

  这一小时中，每人会发起2次左右的打印，每次1～20页

- **打印机的性能是：**

  以草稿模式打印的话，每分钟10页，

  以正常模式打印的话，打印质量好，但速度下降为每分钟5页。

模拟算法：打印任务

- 问题是：怎么设定打印机的模式，让大家都**不会等太久**的前提下**尽量提高打印质量**？
- 这是一个典型的决策支持问题，但无法通过规则直接计算
- 我们要用一段程序来**模拟**这种打印任务场景，然后对程序运行结果进行**分析**，以支持对打印机模式设定的**决策**。

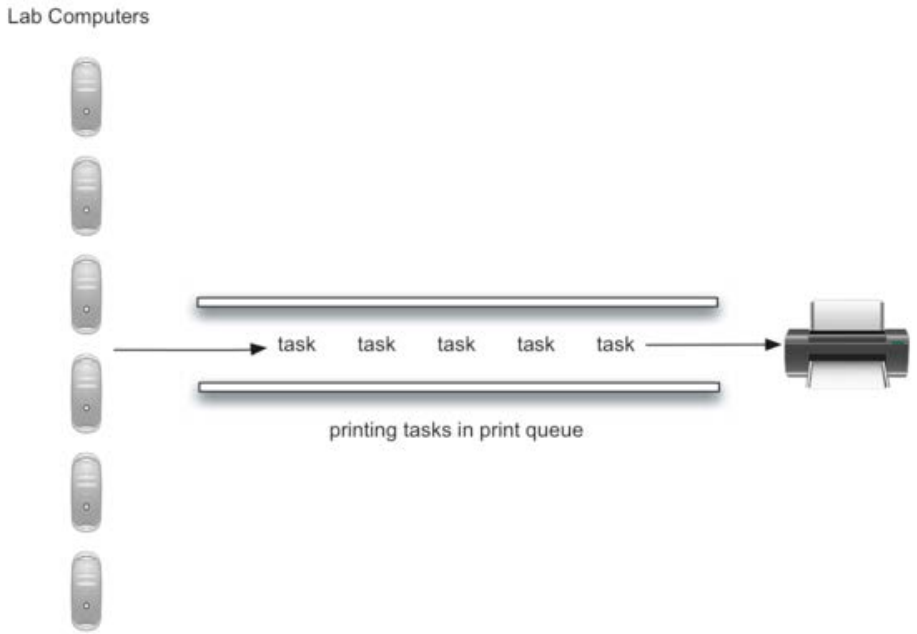
如何对问题建模?

- 首先对问题进行抽象，确定相关的对象和过程

  抛弃那些对问题实质没有关系的学生性别、年龄、打印机型号、打印内容、纸张大小等等众多细节

如何对问题建模?

- **对象：打印任务、打印队列、打印机**
  - 打印任务的属性：提交时间、打印页数
  - 打印队列的属性：具有FIFO性质的打印任务队列
  - 打印机的属性：打印速度、是否忙



如何对问题进行建模?

- **过程：生成和提交打印任务**

  确定**生成概率**：实例为每小时会有10个学生提交的20个作业，这样，概率是每180秒会有1个作业生成并提交，概率为**每秒1/180**。

  确定**打印页数**：实例是1～20页，那么就是1～20页之间概率相同。

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

如何对问题进行建模?

- **过程：实施打印**

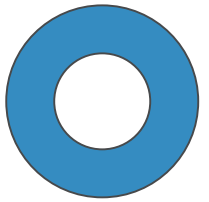
  当前的打印作业：正在打印的作业

  打印结束倒计时：新作业开始打印时开始倒计时，回0表示打印完毕，可以处理下一个作业

- **模拟时间：**

  统一的时间框架：以最小单位（秒）均匀流逝的时间，设定结束时间

  同步所有过程：在一个时间单位里，对生成打印任务和实施打印两个过程各处理一次



数据结构与算法（Python版）

队列的应用：打印任务（下）

陈斌 北京大学 gischen@pku.edu.cn

打印任务问题：模拟流程

- 创建打印队列对象
- 时间按照秒的单位流逝

  按照概率生成打印作业，加入打印队列

  如果打印机空闲，且队列不空，则取出队首作业打印，记录此作业等待时间

  如果打印机忙，则按照打印速度进行1秒打印

  如果当前作业打印完成，则打印机进入空闲
- 时间用尽，开始统计平均等待时间

打印任务问题：模拟流程

- **作业的等待时间**
  - 生成作业时，记录生成的时间戳
  - 开始打印时，当前时间减去生成时间即可
- **作业的打印时间**
  - 生成作业时，记录作业的页数
  - 开始打印时，页数除以打印速度即可

```
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
```

打印速度

打印任务

任务倒计时

打印1秒

打印忙?

打印新作业

打印任务问题：Python代码2


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

生成时间戳

打印页数

等待时间

1/180概率生成作业

```
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()))
```

模拟

时间流逝

打印任务问题：运行和分析1

- 按5PPM、1小时的设定，模拟运行10次

  总平均等待时间93.1秒，最长的平均等待164秒，最短的平均等待26秒

  有3次模拟，还有作业没开始打印

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


Average Wait  67.00 secs   0 tasks remaining.
Average Wait  26.00 secs   0 tasks remaining.
Average Wait  46.00 secs   2 tasks remaining.
Average Wait 115.00 secs   0 tasks remaining.
Average Wait  53.00 secs   0 tasks remaining.
Average Wait 121.00 secs   0 tasks remaining.
Average Wait 164.00 secs   1 tasks remaining.
Average Wait 136.00 secs   0 tasks remaining.
Average Wait 122.00 secs   2 tasks remaining.
Average Wait  81.00 secs   0 tasks remaining.
```

打印任务问题：运行和分析2

❖提升打印速度到10PPM、1小时的设定

总平均等待时间12秒，最长的平均等待35秒，最短的平均等待0秒，就是一提交就打印了

而且，所有作业都打印了

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


Average Wait  35.00 secs   0 tasks remaining.
Average Wait   8.00 secs   0 tasks remaining.
Average Wait  29.00 secs   0 tasks remaining.
Average Wait   0.00 secs   0 tasks remaining.
Average Wait  13.00 secs   0 tasks remaining.
Average Wait   5.00 secs   0 tasks remaining.
Average Wait   0.00 secs   0 tasks remaining.
Average Wait   8.00 secs   0 tasks remaining.
Average Wait  17.00 secs   0 tasks remaining.
Average Wait   5.00 secs   0 tasks remaining.
```

打印任务问题：讨论

- **为了对打印模式设置进行决策，我们用模拟程序来评估任务等待时间**
  - 通过两种情况模拟仿真结果的分析，我们认识到
  - 如果有那么多学生要拿着打印好的程序源代码赶去上课的话
  - 那么，必须得牺牲打印质量，提高打印速度。
- **模拟系统对现实的仿真**
  - 在不耗费现实资源的情况下—有时候真实的实验是无法进行的
  - 可以以不同的设定，反复多次模拟
  - 来帮助我们进行决策。

打印任务问题：讨论

- 打印任务模拟程序还可以加进不同设定，来进行更丰富的模拟

  学生数量加倍了会怎么样？

  如果在周末，学生不需要赶去上课，能接受更长等待时间，会怎么样？

  如果改用Python编程，源代码大大减少，打印的页数减少了，会怎么样？

打印任务问题：讨论

- 更真实的模拟，来源于对问题的更精细建模，以及以真实数据进行设定和运行
- 也可以扩展到其它类似决策支持问题

  如：饭馆的餐桌设置，使得顾客排队时间变短

数据结构与算法（Python版）

双端队列抽象数据类型及Python实现

陈斌 北京大学 gischen@pku.edu.cn

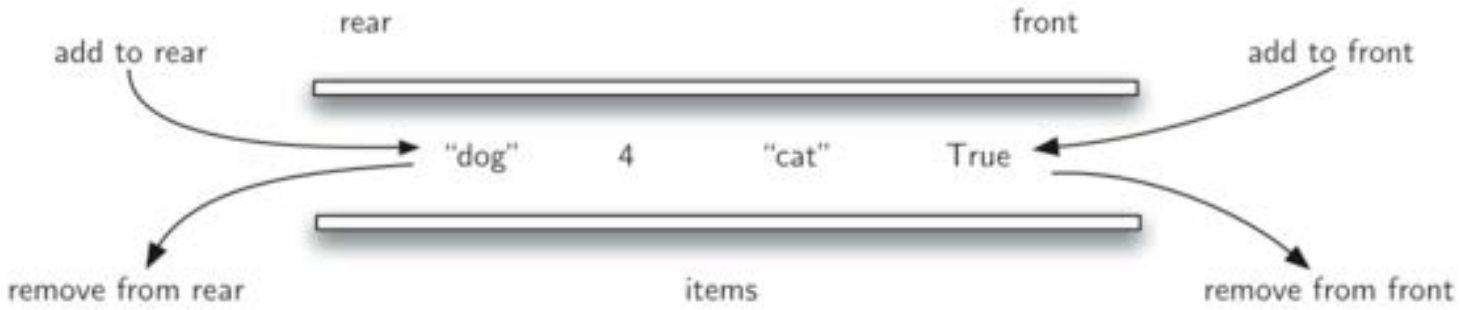
双端队列Deque：什么是Deque?

- **双端队列Deque是一种有次序的数据集，**

  跟队列相似，其两端可以称作“首”“尾”端，

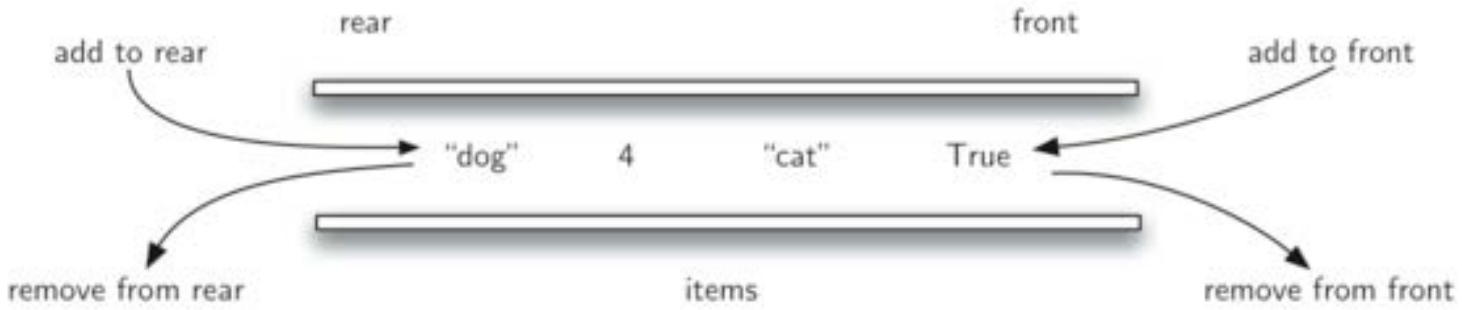
  但deque中数据项既可以从队首加入，也可以从队尾加入；数据项也可以从两端移除。

  某种意义上说，双端队列集成了栈和队列的能力



双端队列Deque：什么是Deque?

- 但双端队列并不具有内在的LIFO或者FIFO特性
  - 如果用双端队列来模拟栈或队列
  - 需要由使用者自行维护操作的一致性



抽象数据类型Deque

- deque定义的操作如下：

  Deque()：创建一个空双端队列

  addFront(item)：将item加入队首

  addRear(item)：将item加入队尾

  removeFront()：从队首移除数据项，返回值为移除的数据项

  removeRear()：从队尾移除数据项，返回值为移除的数据项

  isEmpty()：返回deque是否为空

  size()：返回deque中包含数据项的个数

抽象数据类型Deque

双端队列操作	双端队列内容	返回值
d=Deque()	[]	Deque object
d.isEmpty()	[]	True
d.addRear(4)	[4]	
d.addRear('dog')	['dog',4,]	
d.addFront('cat')	['dog',4,'cat']	
d.addFront(True)	['dog',4,'cat',True]	
d.size()	['dog',4,'cat',True]	4
d.isEmpty()	['dog',4,'cat',True]	False
d.addRear(8.4)	[8.4,'dog',4,'cat',True]	
d.removeRear()	['dog',4,'cat',True]	8.4
d.removeFront()	['dog',4,'cat']	True

Python实现ADT Deque

- 采用List实现
  - List下标0作为deque的尾端
  - List下标-1作为deque的首端
- 操作复杂度
  - addFront/removeFront O(1)
  - addRear/removeRear O(n)

```
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)
```

“回文词”判定

- “回文词”指正读和反读都一样的词

  如radar、madam、toot

  中文“上海自来水来自海上”

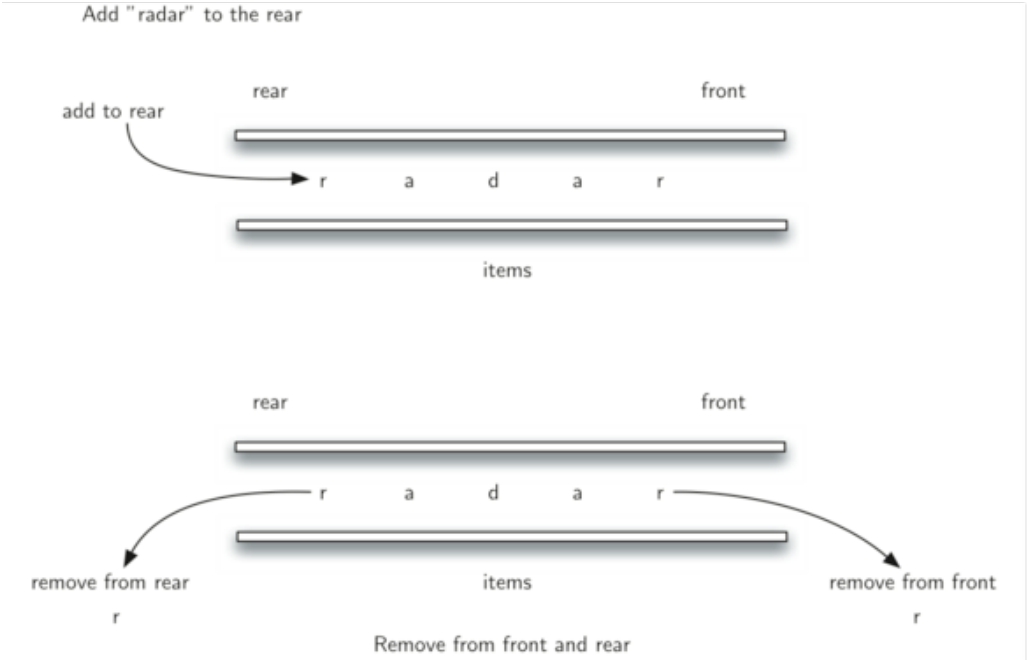
  “山东落花生花落东山”

“回文词”判定

- 用双端队列很容易解决“回文词”问题

  先将需要判定的词从队尾加入deque

  再从两端同时移除字符判定是否相同，直到deque中剩下0个或1个字符



“回文词”判定：代码


```
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"))
```

数据结构与算法（Python版）

无序表抽象数据类型及Python实现

陈斌 北京大学 gischen@pku.edu.cn

列表List：什么是列表?

- 在前面基本数据结构的讨论中，我们采用Python List来实现了多种线性数据结构
- 列表List是一种简单强大的数据集结构，提供了丰富的操作接口

  但并不是所有的编程语言都提供了List数据类型，有时候需要程序员自己实现。

列表List：什么是列表?

- 一种数据项按照**相对位置**存放的数据集

  特别的，被称为“无序表unordered list”

  其中数据项只按照存放位置来索引，如第1个、第2个……、最后一个等。

  （为了简单起见，假设表中不存在重复数据项）

- 如一个考试分数的集合“54, 26, 93, 17, 77和31”

- 如果用无序表来表示，就是[54, 26, 93, 17, 77, 31]

抽象数据类型：无序表List

- 无序表List的操作如下：

  List()：创建一个空列表

  add(item)：添加一个数据项到列表中，假设item原先不存在于列表中

  remove(item)：从列表中移除item，列表被修改，item原先应存在于表中

  search(item)：在列表中查找item，返回布尔类型值

  isEmpty()：返回列表是否为空

  size()：返回列表包含了多少数据项

抽象数据类型：无序表List

- 无序表List的操作如下：

  append(item)：添加一个数据项到表末尾，假设item原先不存在于列表中

  index(item)：返回数据项在表中的位置

  insert(pos, item)：将数据项插入到位置pos，假设item原先不存在与列表中，同时原列表具有足够多个数据项，能让item占据位置pos

  pop()：从列表末尾移除数据项，假设原列表至少有1个数据项

  pop(pos)：移除位置为pos的数据项，假设原列表存在位置pos

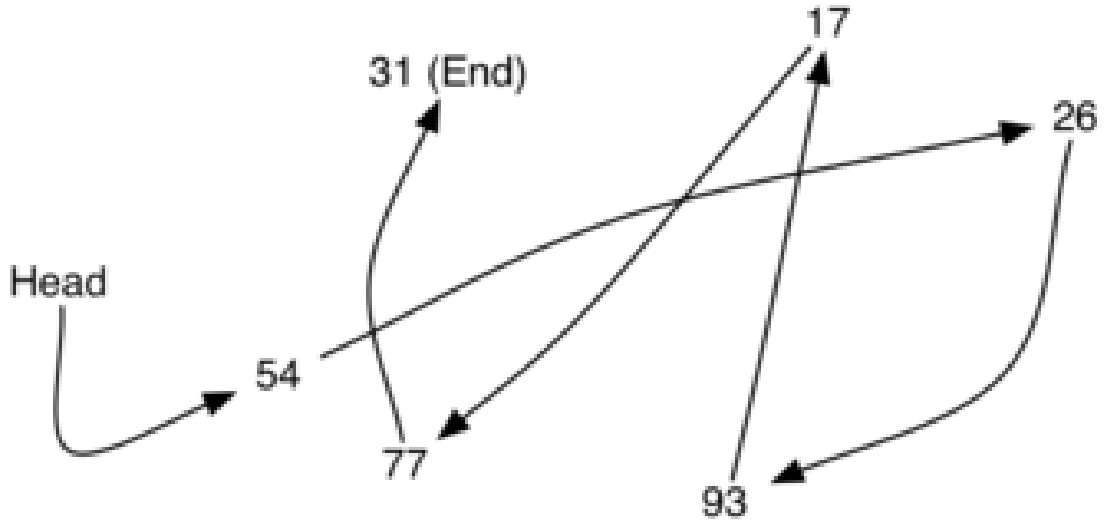
采用链表实现无序表

- 为了实现无序表数据结构，可以采用链接表的方案。
- 虽然列表数据结构要求保持数据项的前后相对位置，但这种前后位置的保持，并**不要求**数据项**依次存放**在连续的存储空间

采用链表实现无序表

- 如下图，数据项存放位置并没有规则，但如果在数据项之间建立**链接指向**，就可以保持其**前后相对**位置

  第一个和最后一个数据项需要显式标记出来，一个是队首，一个是队尾，后面再无数据了。

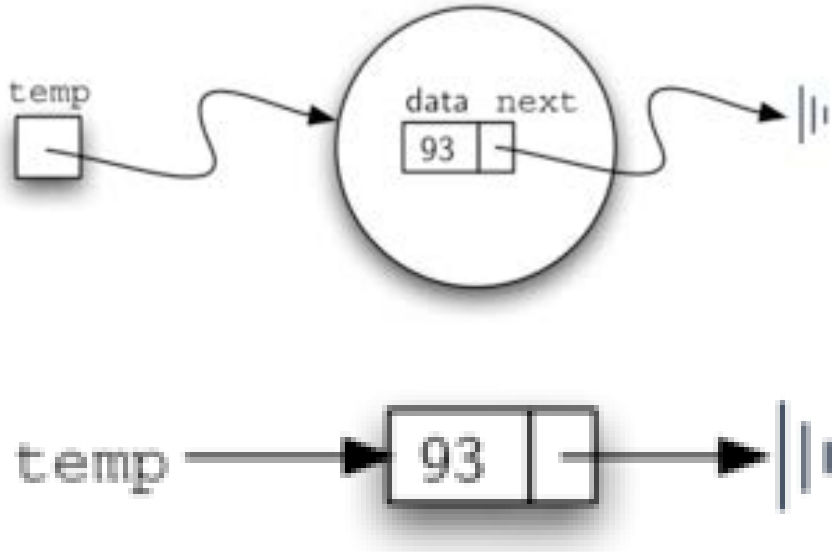


链表实现：节点Node

- 链表实现的最基本元素是节点Node

  每个节点至少要包含2个信息：数据项本身，以及指向下一个节点的引用信息

  注意next为None的意义是没有下一个节点了，这个很重要



链表实现：节点Node

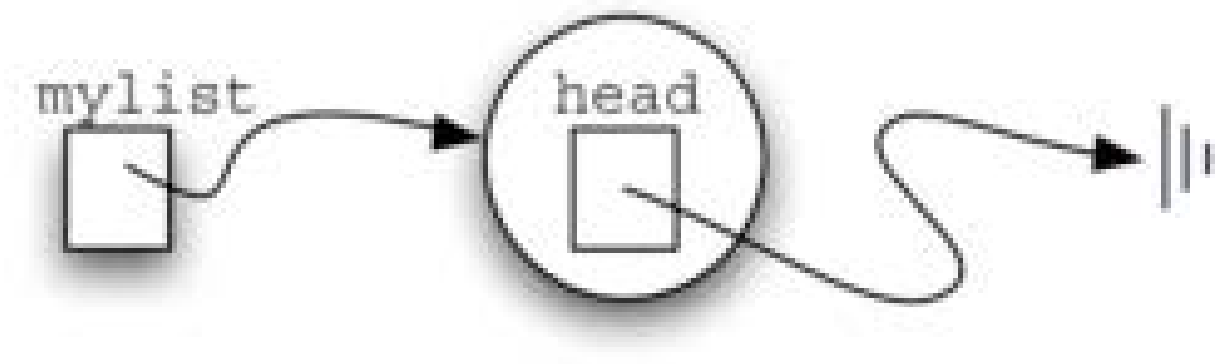

```
class Node:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None

    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
```

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

链表实现：无序表UnorderedList

- 可以采用**链接节点**的方式构建数据集来实现无序表
- 链表的**第一个**和**最后一个**节点最重要

  如果想访问到链表中的所有节点，就必须从第一个节点开始沿着链接遍历下去



链表实现：无序表UnorderedList

- 所以无序表必须要有对**第一个节点**的引用信息

  设立一个属性head，保存对第一个节点的引用

  空表的head为None

```
class UnorderedList:

    def __init__(self):
        self.head = None

>>> mylist= UnorderedList()
>>> print mylist.head
None
```

链表实现：无序表UnorderedList

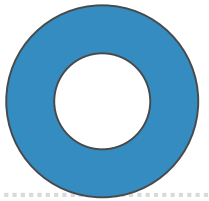
- 随着数据项的加入，无序表的head始终指向链条中的第一个节点

  注意！无序表mylist对象本身并不包含数据项（数据项在节点中）

  其中包含的head只是对首个节点Node的引用

  判断空表的isEmpty()很容易实现

  - `return self.head == None`



数据结构与算法（Python版）

无序表的链表实现

陈斌 北京大学 gischen@pku.edu.cn

链表实现：无序表UnorderedList

- 接下来，考虑如何实现向无序表中添加数据项，实现**add方法**。
- 由于无序表并没有限定数据项之间的顺序
- 新数据项可以加入到原表的**任何**位置
- 按照实现的性能考虑，应添加到**最容易加入**的位置上。

链表实现：无序表UnorderedList

- 由链表结构我们知道
- 要访问到整条链上的所有数据项
- 都必须从表头head开始沿着next链接逐个向后查找
- 所以添加新数据项最快捷的位置是**表头**，整个链表的首位置。

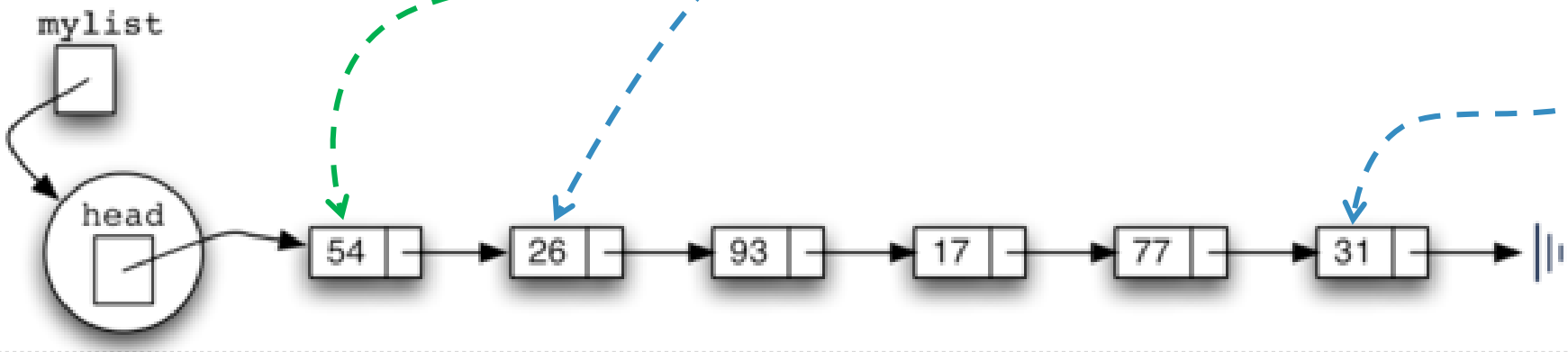
链表实现：无序表UnorderedList

- add方法
- 按照右图的代码调用，形成的链表如下图

31是最先被加入的数据项，所以成为链表中最后一个项

而54是最后被加入的，是链表第一个数据项

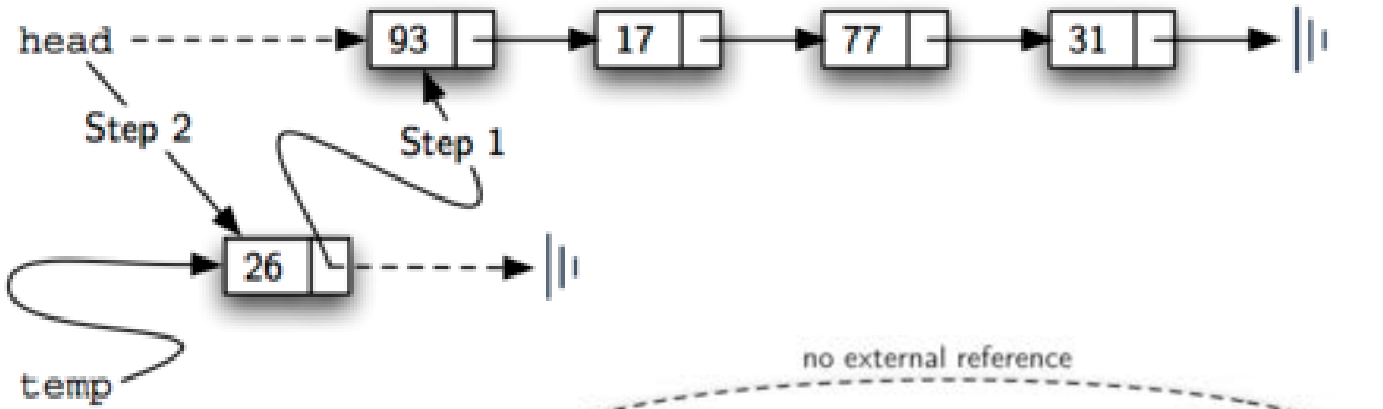
```
mylist.add(31)
mylist.add(77)
mylist.add(17)
mylist.add(93)
mylist.add(26)
mylist.add(54)
```

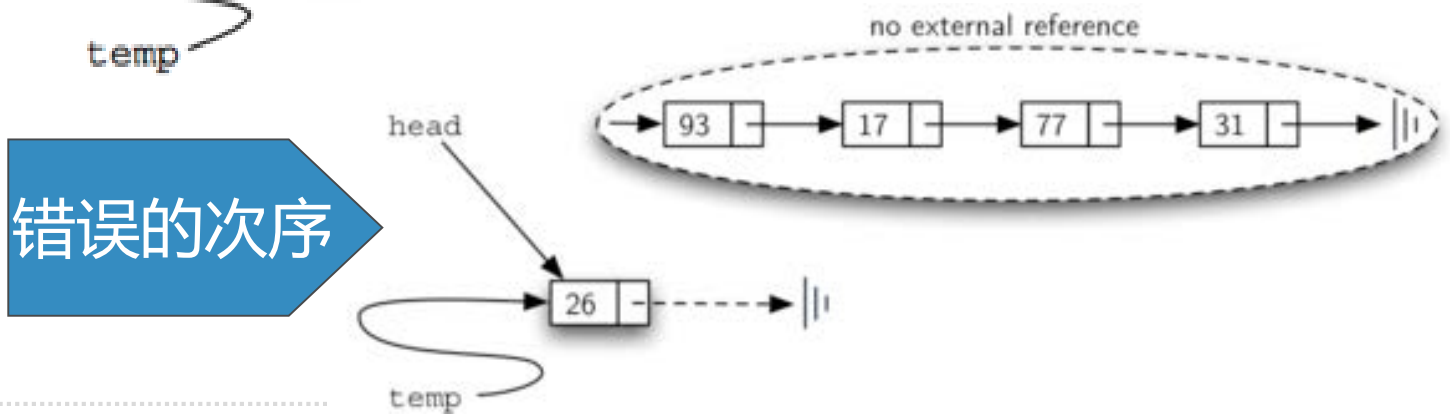


链表实现：add方法实现

❖ 链接次序很重要！

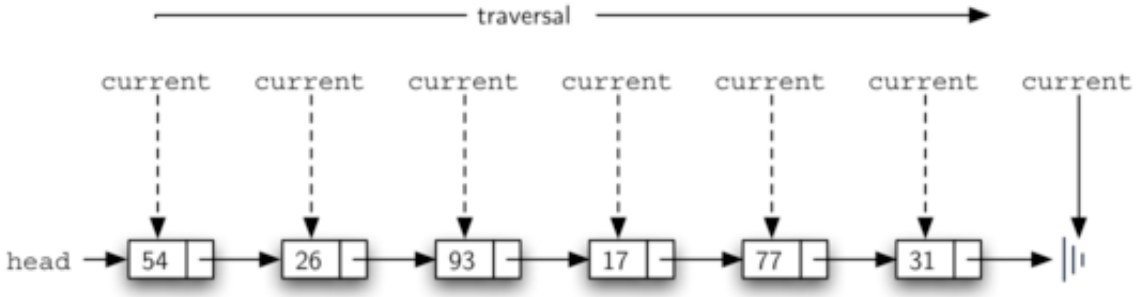
```
def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp
```





链表实现：size

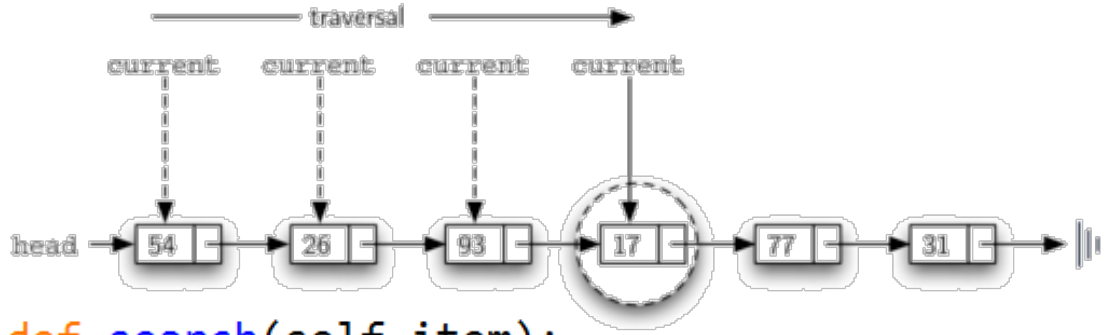
- size：从链条头head开始遍历到表尾同时用变量累加经过的节点个数。




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

    return count
```

链表实现：search

- 从链表头head开始遍历到表尾，同时判断当前节点的数据项是否目标




```
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
```

链表实现：remove(item)方法

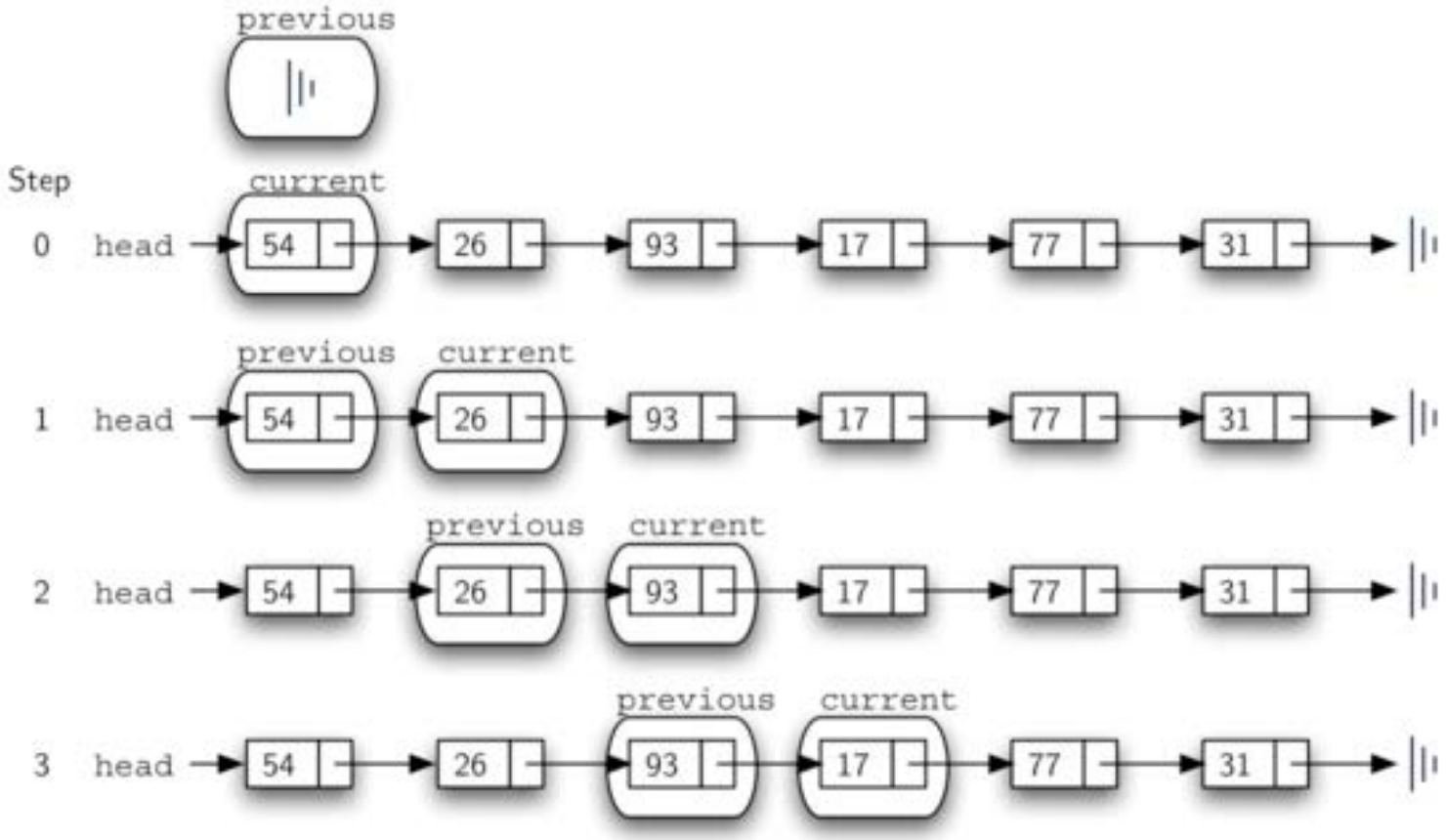
- 首先要找到item，这个过程跟search一样，但在删除节点时，需要**特别的**技巧

  current指向的是当前匹配数据项的节点

  而删除需要把前一个节点的next指向current的下一个节点

  所以我们在search current的同时，还要维护前一个(previous)节点的引用

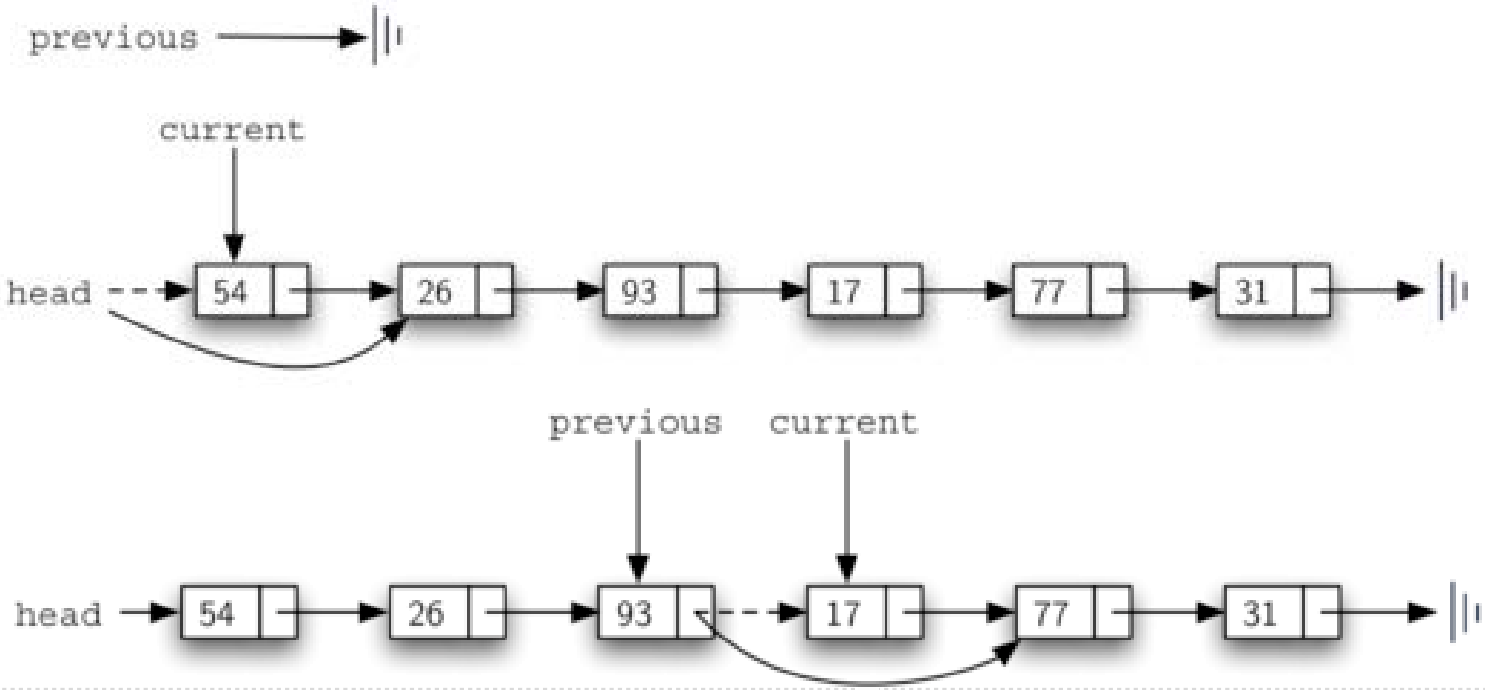
链表实现：remove(item)方法



链表实现：remove(item)方法

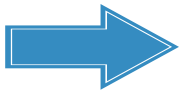
- 找到item之后，current指向item节点，previous指向前一个节点，开始执行删除，需要区分两种情况：

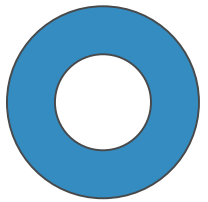
  current是首个节点；或者是位于链条中间的节



链表实现：remove(item)代码


```
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())
```





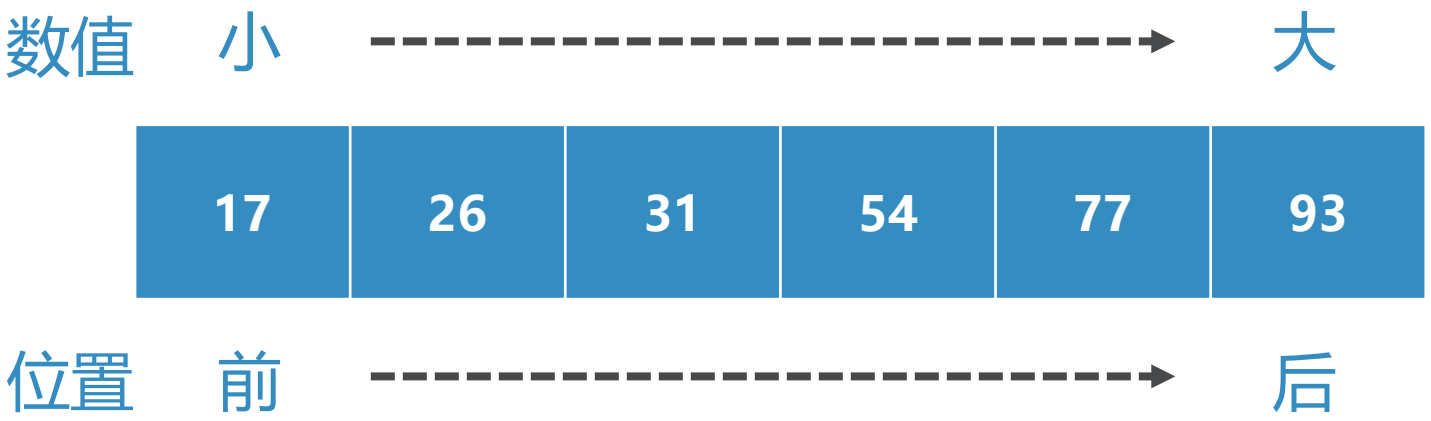
数据结构与算法（Python版）

有序表抽象数据类型及Python实现

陈斌 北京大学 gischen@pku.edu.cn

抽象数据类型：有序表OrderedList

- 有序表是一种数据项依照其某**可比性质**（如整数大小、字母表先后）来决定在列表中的**位置**
- 越“小”的数据项越靠近列表的头，越靠“前”



抽象数据类型：有序表OrderedList

- OrderedList所定义的操作如下：

  OrderedList(): 创建一个空的有序表

  add(item): 在表中添加一个数据项，并保持整体顺序，此项原不存在

  remove(item): 从有序表中移除一个数据项，此项应存在，有序表被修改

  search(item): 在有序表中查找数据项，返回是否存在

  isEmpty(): 是否空表

  size(): 返回表中数据项的个数

  index(item): 返回数据项在表中的位置，此项应存在

  pop(): 移除并返回有序表中最后一项，表中应至少存在一项

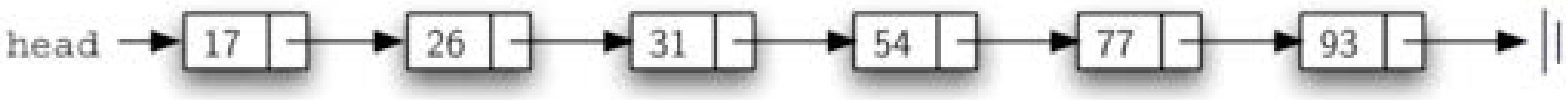
  pop(pos): 移除并返回有序表中指定位置的数据项，此位置应存在

有序表OrderedList实现

- 在实现有序表的时候，需要记住的是，数据项的相对位置，取决于它们之间的“大小”比较

  由于Python的扩展性，下面对数据项的讨论并不仅适用于整数，可适用于所有定义了__gt__方法（即'>'操作符）的数据类型

- 以整数数据项为例，(17, 26, 31, 54, 77, 93)的链表形式如图



有序表OrderedList实现

- 同样采用链表方法实现
- Node定义相同
- OrderedList也设置一个head来保存链表表头的引用

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

有序表OrderedList实现

- 对于isEmpty/size/remove这些方法，与节点的次序无关，所以其实现跟UnorderedList是一样的。
- search/add方法则需要有修改

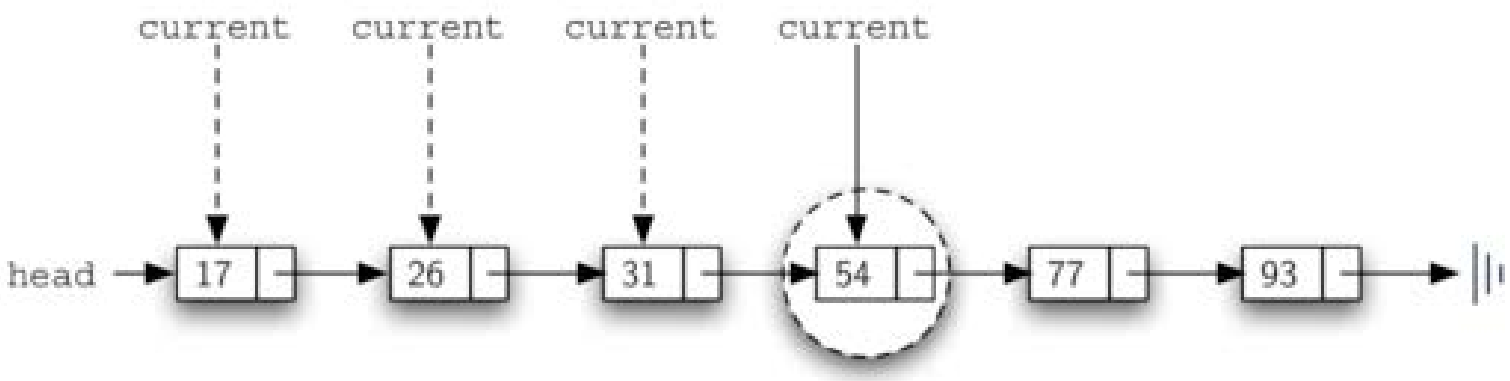
有序表实现：search方法

- 在**无序表**的search中，如果需要查找的数据项不存在，则会搜遍整个链表，直到表尾
- 对于**有序表**来说，可以利用链表节点有序排列的特性，来为search节省**不存在数据项**的查找时间

  一旦当前节点的数据项大于所要查找的数据项，则说明链表后面已经不可能再有要查找的数据项，可以直接返回False

有序表实现：search方法

- 如我们要在下图查找数据项45



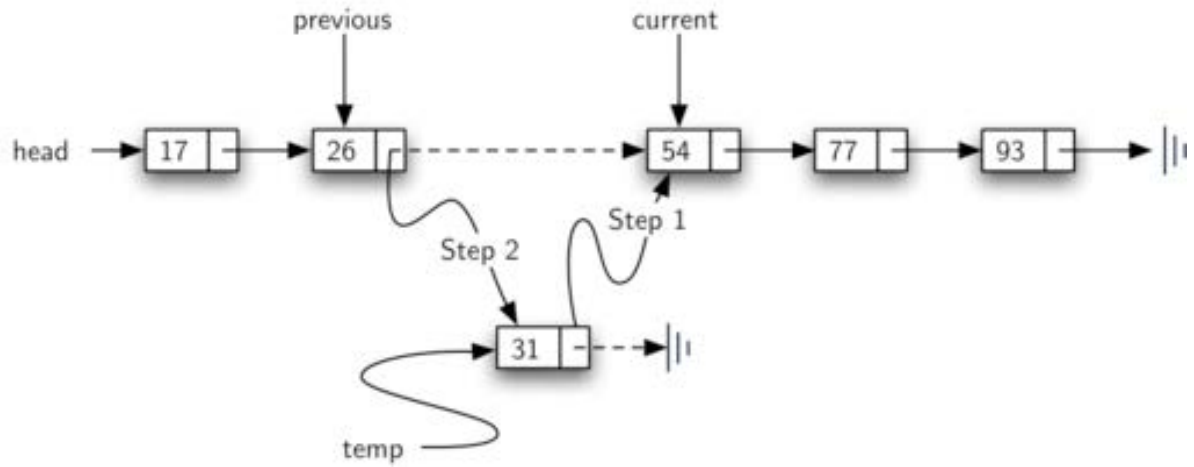
有序表实现：search方法


```
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
```

有序表实现：add方法

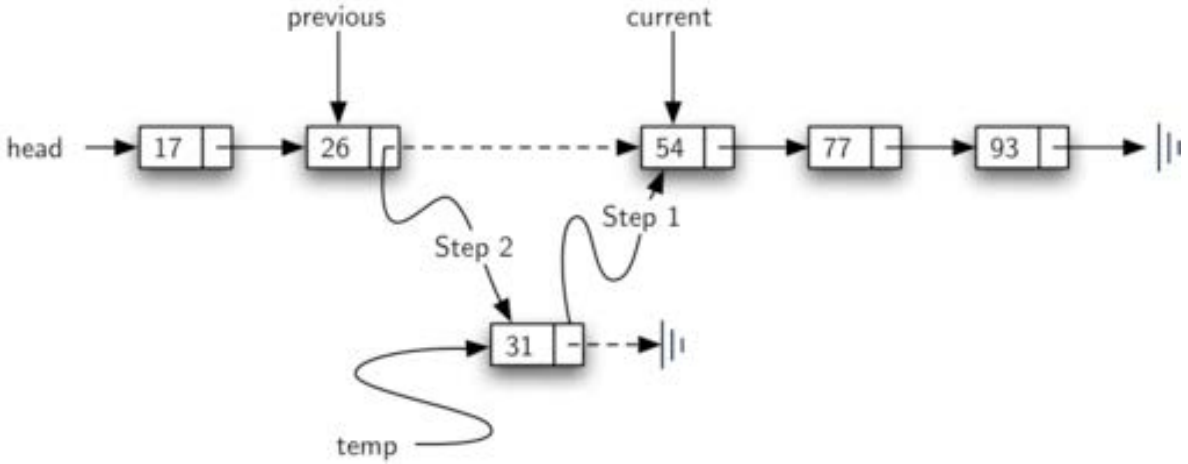
- 相比无序表，改变最大的方法是add，因为add方法必须**保证**加入的数据项添加在合适的位置，以**维护**整个链表的有序性

  比如在(17，26，54，77，93)的有序表中，加入数据项31，我们需要沿着链表，找到第一个比31大的数据项54，将31插入到54的前面



有序表实现：add方法

- 由于涉及到的插入位置是当前节点**之前**，而链表无法得到“前驱”节点的引用
- 所以要跟remove方法类似，引入一个previous的引用，跟随当前节点current
- 一旦找到首个比31大的数据项，previous就派上用场了



有序表OrderedList实现：add方法

发现插入位置

插入在表头

插入在表中

```
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)
```

链表实现的算法分析

- 对于链表复杂度的分析，主要是看相应的方法是否涉及到链表的**遍历**
- 对于一个包含节点数为n的链表

  isEmpty是O(1)，因为仅需要检查head是否为None

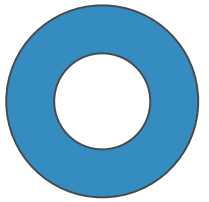
  size是O(n)，因为除了遍历到表尾，没有其它办法得知节点的数量

  search/remove以及有序表的add方法，则是O(n)，因为涉及到链表的遍历，按照概率其平均操作的次数是n/2

  无序表的add方法是O(1)，因为仅需要插入到表头

链表实现的算法分析

- 链表实现的List，跟Python内置的列表数据类型，在有些相同方法的实现上的时间复杂度不同
- 主要是因为Python内置的列表数据类型是基于**顺序存储**来实现的，并进行了优化。



数据结构与算法（Python版）

线性结构小结

陈斌 北京大学 gischen@pku.edu.cn

线性结构小结

- 线性数据结构Linear DS将数据项以某种线性的次序组织起来
- 栈Stack维持了数据项后进先出LIFO的次序

  stack的基本操作包括push，pop，isEmpty
- 队列Queue维持了数据项先进先出FIFO的次序

  queue的基本操作包括enqueue，dequeue，isEmpty

线性结构小结

- 书写表达式的方法有前缀prefix、中缀infix和后缀postfix三种

  由于栈结构具有次序反转的特性，所以栈结构适合用于开发表达式求值和转换的算法

- “模拟系统”可以通过一个对现实世界问题进行抽象建模，并加入随机数动态运行，为复杂问题的决策提供各种情况的参考

  队列queue可以用来进行模拟系统的开发

线性结构小结

- 双端队列Deque可以同时具备栈和队列的功能

  deque的主要操作包括addFront，addRear，removeFront，removeRear，isEmpty

- 列表List是数据项能够维持相对位置的数据集
- 链表的实现，可以保持列表维持相对位置的特点，而不需要连续的存储空间
- 链表实现时，其各种方法，对链表头部head需要特别的处理