数据结构与算法

基本数据结构

张晓平

武汉大学数学与统计学院

2017 年 9 月 25 日

目录

目标

目标

- 理解抽象数据类型的栈，队列，deque 和列表。
- 能使用 Python 列表实现 ADT 堆栈，队列和 deque。
- 了解基本线性数据结构实现的性能。
- 了解前缀、中缀和后缀表达式格式。
- 使用栈来实现后缀表达式。
- 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。
- 学会在问题中合理的使用栈、队列和 deques 等数据结构。
- 能使用节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。

线性数据结构

线性数据结构

1. 栈（stack）
2. 队列（sequence）
3. 双端列表（deque）
4. 列表（list）

它们都是一类数据的集合，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

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。

重要的是将数据结构区增加和删除数据的方式，特别是增删的位置。例如，一种结构可能允许只从一端添加数据，而另一种则两端都行。

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，并可以用于解决很多重要的问题。

栈

栈

栈是一个线性有序的数据元素集合，其中数据的增加删除操作总发生在同一端。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，因为元素越接近底部，在栈中的时间越长。
- 最近添加的项是最先会被移除的。这种排序原则称为“后进先出，last in first out(LIFO)”。
- 栈是按时间长短来排列元素的。新来的在栈顶，老家伙们在栈底。

栈

栈的举例

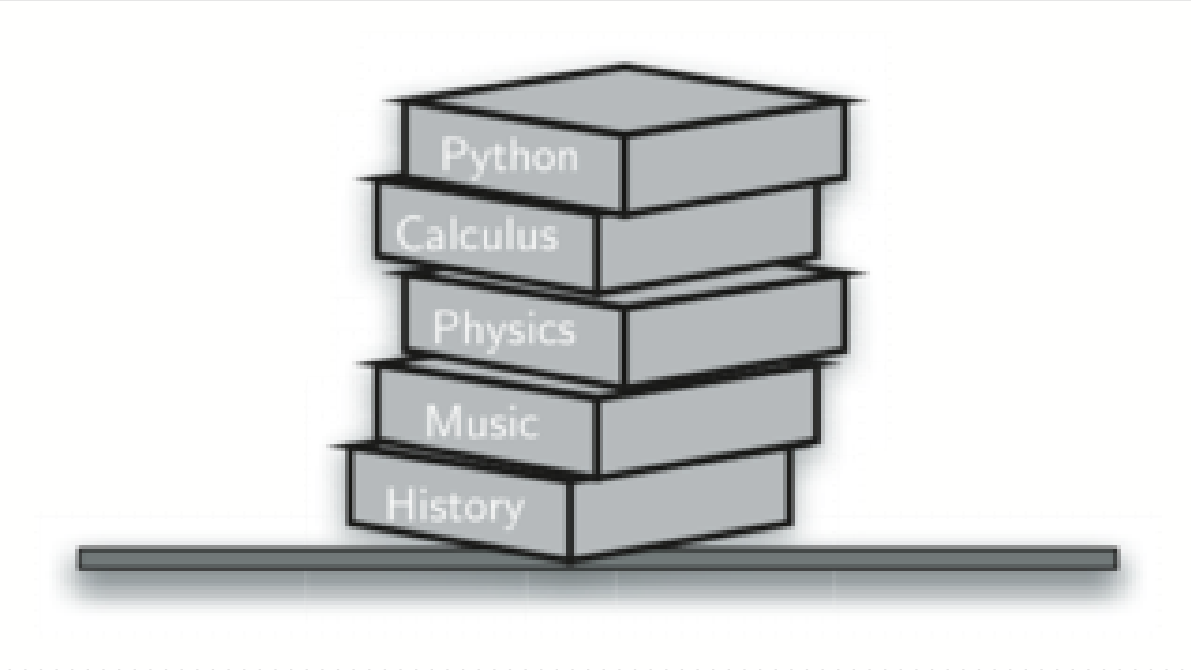
例:

自助餐厅的盘，人们总是从上面拿走盘子，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)。

栈的举例

例:

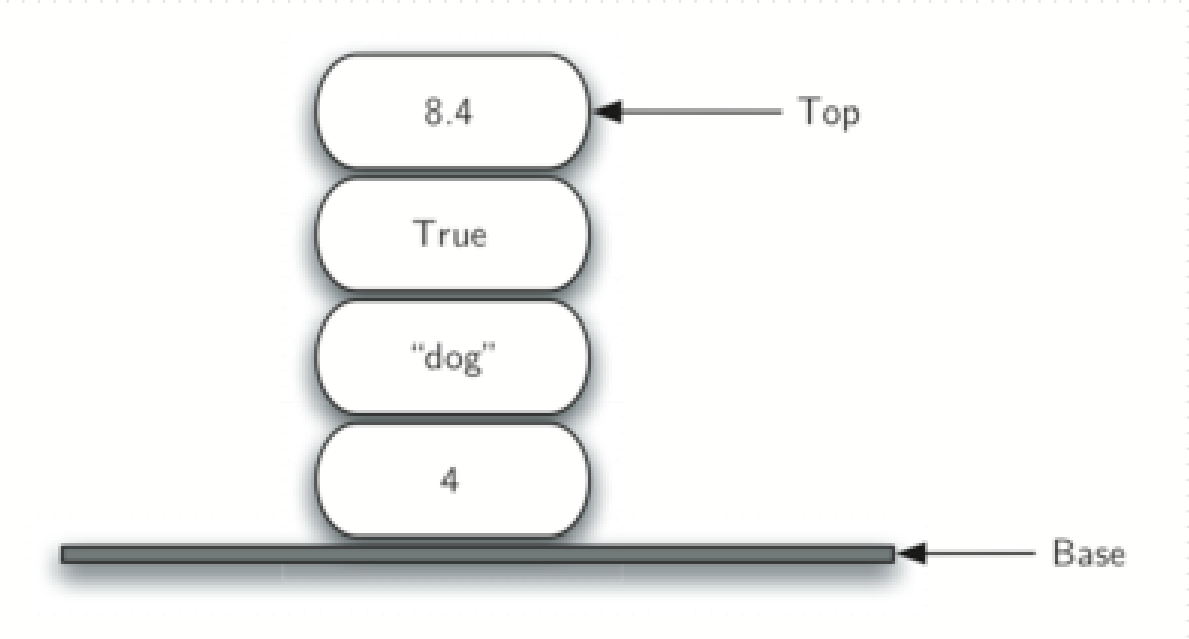
又如一堆书，你只能看见最上面一本的封面，要看下面一本，就得把上面的先拿走。



栈的举例

例:

下图展示了另一个栈，存储的是几个主要的 Python 数据对象。



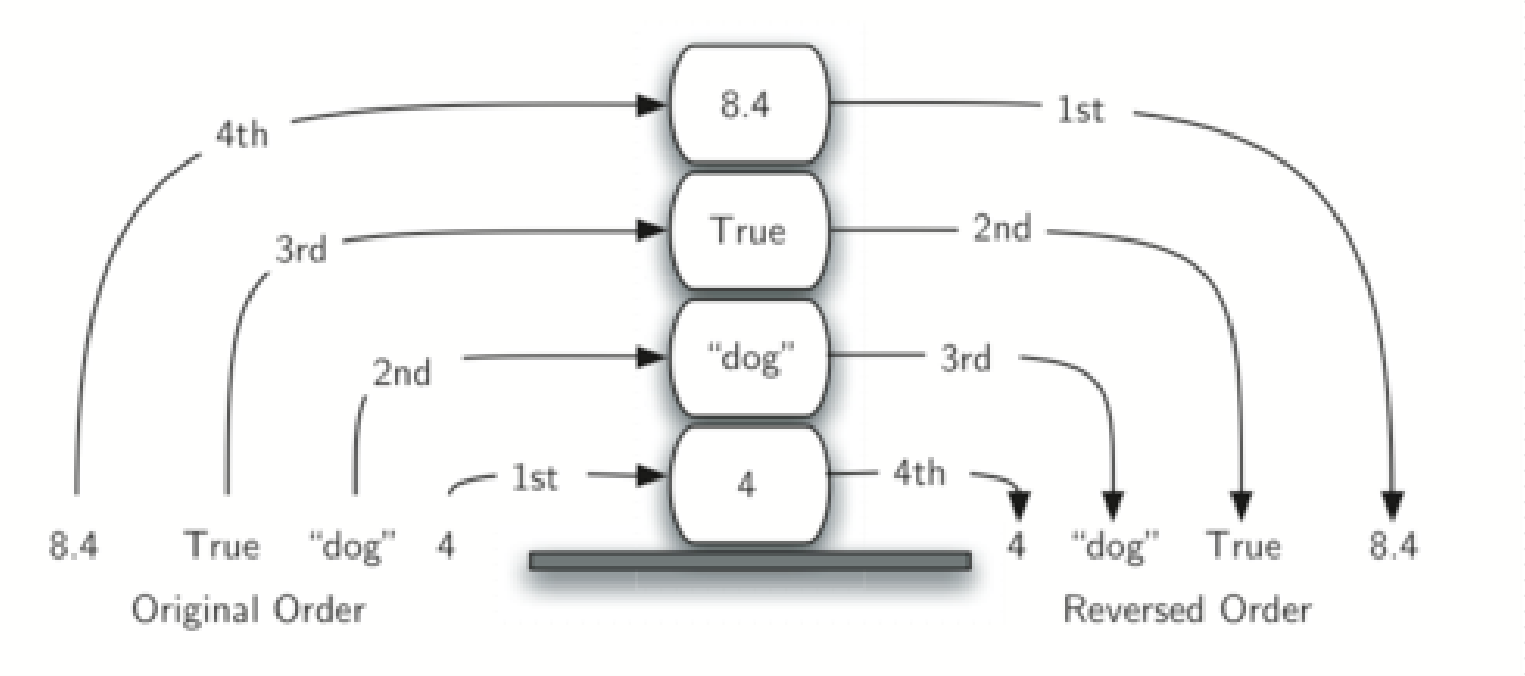
栈的举例

和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。

栈的举例

例:

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。



栈的举例

例:

栈这种翻转性，在你用电脑上网的时候也用到了。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。

栈

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。如上所述，栈是结构化的、有序的数据集合，它的增删操作都在叫做“栈顶”的一端进行，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。

栈的抽象数据类型

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

栈

用 Python 实现栈

现在我们已经将栈清楚地定义了抽象数据类型，我们将把注意力转向使用 Python 实现栈。回想一下，当我们给抽象数据类型一个物理实现时，我们将实现称为数据结构。正如我们在第 1 章中所描述的，在 Python 中，与任何面向对象编程语言一样，抽象数据类型（如栈）的选择的实现是创建一个新类。栈操作实现为类的方法。此外，为了实现作为元素集合的栈，使用由 Python 提供的原语集合的能力是有意义的。我们将使用列表作为底层实现。

用 Python 实现栈

回想一下，Python 中的列表类提供了有序集合机制和一组方法。例如，如果我们有列表 $[2, 5, 3, 6, 7, 4]$，我们只需要确定列表的哪一端将被认为是栈的顶部。一旦确定，可以使用诸如 append 和 pop 的列表方法来实现操作。以下栈实现假定列表的结尾将保存栈的顶部元素。随着栈增长（push 操作），新项将被添加到列表的末尾。pop 也操作列表末尾的元素。

用 Python 实现栈

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

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

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

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

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

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

注意这里只定义了类的实现，我们需要创建一个栈，然后使用它。

用 Python 实现栈

以下代码展示了我们通过实例化 Stack 类执行上述表格中的操作。注意，Stack 类的定义是从 pythonds 模块导入的。

```
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())
```

栈

简单括号匹配

我们现在把注意力转向使用栈解决真正的计算机问题。

- 你会这么写算术表达式

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

其中括号用于命令操作的执行。

- 你可能也有一些语言的经验，如 Lisp 的构造

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

这段代码定义了一个名为 square 的函数，它将返回参数的 n 的平方。Lisp 使用大量的圆括号是臭名昭著的。

简单括号匹配

在这两个例子中，括号必须以匹配的方式出现。括号匹配意味着每个开始符号具有相应的结束符号，并且括号能被正确嵌套。考虑下面正确匹配的括号字符串：

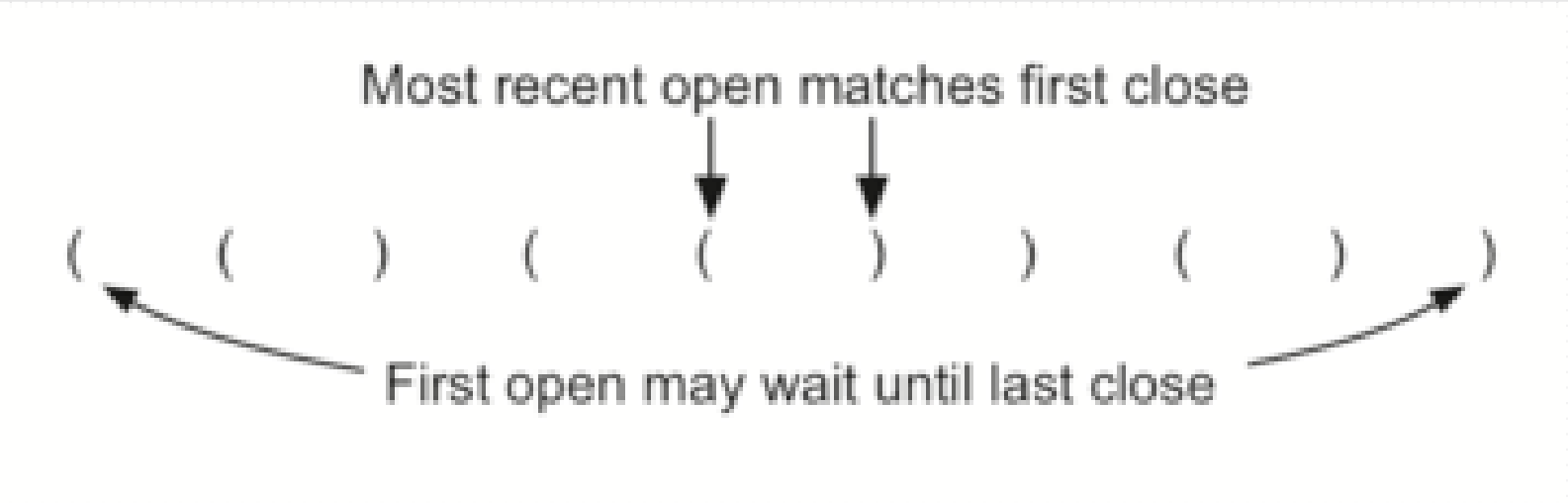
```
( ( ) ( ) ( ) ( ) )
( ( ( ( ) ) ) )
( ( ) ( ( ( ) ) ( ) ) )
```

对比那些不匹配的括号：

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

简单括号匹配

区分括号是否匹配的能力是识别很多编程语言结构的重要部分。具有挑战的是如何编写一个算法，能够从左到右读取一串符号，并决定符号是否平衡。为了解决这个问题，我们需要做一个重要的观察：从左到右处理符号时，最近开始符号必须与下一个关闭符号相匹配此外，处理的第一个开始符号必须等待直到其匹配最后一个符号。结束符号以相反的顺序匹配开始符号。他们从内到外匹配。这是一个可以用栈解决问题的线索。



简单括号匹配

一旦你认为栈是保存括号的恰当的数据结构，算法是很直接的。从空栈开始，从左到右处理括号字符串。如果一个符号是一个开始符号，将其作为一个信号，对应的结束符号稍后会出现。另一方面，如果符号是结束符号，弹出栈，只要弹出栈的开始符号可以匹配每个结束符号，则括号保持匹配状态。如果任何时候栈上没有出现符合开始符号的结束符号，则字符串不匹配。最后，当所有符号都被处理后，栈应该是空的。

简单括号匹配

```
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
```

简单括号匹配

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

```
True
False
```