

第三章随堂作业及上机作业

- 提交作业时请在压缩包内加上学号姓名

随堂作业3-1

› 右上的代码执行后，栈顶元素是什么？

A) 'x' B) 'y' C) 'z' D) 栈已经空了

```
m = Stack()
m.push('x')
m.push('y')
m.pop()
m.push('z')
m.peek()
```

› 右下的代码执行后，栈顶元素是什么？

A) 'x' B) 栈已经空了 C) 会出错！ D) 'z'

› 请写一个函数，用Stack的反转特性，将输入参数的字符串，反转输出

```
def revString(mystr):
    stack = Stack()
    for char in mystr:
        stack.push(char)
    result = ''
    while not stack.isEmpty():
        result += stack.pop()
    return result
```

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

随堂作业3-2

- 手工转换中缀表达式到后缀形式： $10 + 3 * 5 / (16 - 4)$
- 计算后缀表达式值： $17 \ 10 + 3 * 9 /$
- 扩展前述的 `infixToPostfix` 函数代码，使之能处理指数操作符“ $\wedge$ ”，写出扩展的部分代码

指数操作符用法如： $5 * 3 \wedge (4 - 2)$

```
(10 + ((3 * 5) / (16 - 4)))  
=> (10 ((3 * 5) (16 - 4) /) +)  
=> (10 ((3 5 *) (16 4 -) /) +)  
=> 10 3 5 * 16 4 - / +
```

```
17 10 + 3 * 9 /  
=> (((17 10 +) 3 *) 9 /)  
=> 9
```

```
prec = {}  
prec['^'] = 4  
prec['*'] = 3  
prec['/'] = 3  
prec['+'] = 2  
prec['-'] = 2  
prec['('] = 1
```

随堂作业3-3

- › 实现UnorderedList的append方法，并指出此方法的时间复杂度是什么？
- › 如果在UnorderedList类中添加一个变量，就可以将append方法的复杂度降低到 $O(1)$ ，但随着这个变量的引入，就还需要相应地修改add方法，请实现新的append/add方法。
- › 请解释下remove方法，当需要移除的节点在链表最后一个的情况，以及链表中仅有的一个节点移除的情况。

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

```
def append(self, item):  
    if self.head == None:  
        self.add(item)  
    else:  
        currentNode = self.head  
        nextNode = currentNode.getNext()  
        while nextNode != None:  
            currentNode = nextNode  
            nextNode = nextNode.getNext()  
        currentNode.setNext(Node(item))
```

```
def add(self, item):  
    temp = Node(item)  
    temp.setNext(self.head)  
    self.head = temp  
    if self.rear == None:  
        self.rear = temp  
def append(self, item):  
    if self.head == None:  
        self.add(item)  
    else:  
        self.rear.setNext(Node(item))
```

- 通过把“中缀转后缀”和“后缀求值”两个算法功能集成在一起(非简单的顺序调用),实现对中缀表达式直接求值,新算法还是从左到右扫描中缀表达式,但同时使用两个栈,一个暂存操作符,一个暂存操作数,来进行求值

```
def infixToPostfix(infixexpr):
    prec = {'*': 3, '/': 3, '+': 2, '-': 2, '(': 1}
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()
    for token in tokenList:
        if token.isdigit():
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            while opStack.peek() != '(':
                postfixList.append(opStack.pop())
            opStack.pop() # pop op '('
        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)
```

```
def doMath(op, a, b):  
    if op == '+':  
        return a + b  
    elif op == '-':  
        return a - b  
    elif op == '*':  
        return a * b  
    elif op == '/':  
        return a / b  
    else:  
        raise ValueError('Invalid op')
```

```
def postfixEval(postfixExpr):  
    operandStack = Stack()  
    tokenList = postfixExpr.split()  
    for token in tokenList:  
        if token.isdigit():  
            operandStack.push(int(token))  
        else:  
            b = operandStack.pop()  
            a = operandStack.pop()  
            result = doMath(token, a, b)  
            operandStack.push(result)  
    return operandStack.pop()
```

```
def infixEval(infixexpr):
    prec = {'*': 3, '/': 3, '+': 2, '-': 2, '(': 1}
    opStack = Stack()
    operandStack = Stack()

    def applyOp(op):
        b = operandStack.pop()
        a = operandStack.pop()
        result = doMath(op, a, b)
        operandStack.push(result)

    tokenList = infixexpr.split()
    for token in tokenList:
        if token.isdigit():
            operandStack.push(int(token))
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            while opStack.peek() != '(':
                applyOp(opStack.pop())
            opStack.pop() # pop op '('
        else:
            while (not opStack.isEmpty()) and \
                (prec[opStack.peek()] >= prec[token]):
                applyOp(opStack.pop())
            opStack.push(token)
    while not opStack.isEmpty():
        applyOp(opStack.pop())
    return operandStack.pop()
```

- 实现一个基数排序算法,用于10进制的正整数排序。思路是保持10个队列(队列0、队列1.....队列9、队列main),开始,所有的数都在main队列,没有排序。
- 第一趟将所有的数根据其10进制个位(0~9),放入相应的队列0~9,全放好后,按照FIFO的顺序,将每个队列的数合并排到main队列
- 第二趟再从main队列队首取数,根据其十位的数值,放入相应队列0~9,全放好后,仍然按照FIFO的顺序,将每个队列的数合并排到main队列
- 第三趟放百位,再合并;第四趟放千位,再合并
- 直到最多的位数放完,合并完,这样main队列里就是排好序的数列了

```
def radixSort(data, radix = 10):  
    slots = [Queue() for _ in range(radix)]  
    base = radix  
    while True:  
        for x in data:  
            slots[x % (radix * base) / base].enqueue(x)  
        if slots[0].size() == len(data):  
            break  
        data = []  
        for queue in slots:  
            while not queue.isEmpty():  
                data.append(queue.dequeue())  
        base *= radix  
    return data
```

- 扩展括号匹配算法,用来检查HTML文档的标记是否匹配。
- HTML标记应该成对、嵌套出现
- 开标记是<tag>这种形式,闭标记是</tag>这种形式
- 给一个HTML文件,算法检查是否有标记不匹配的情况

```
def htmlCheck(html):  
    s = Stack()  
    balanced = True  
    index = 0  
  
    while index < len(html) and balanced:  
        char = html[index]  
        if char == '<':  
            index += 1  
            tagName = ''  
            c = html[index]  
            index += 1  
            while c != '>':  
                tagName += c  
                c = html[index]  
                index += 1  
            if len(tagName) > 0 and tagName[0] == '/':  
                balanced = (not s.isEmpty()) and (s.pop() == tagName[1:])  
            else:  
                s.push(tagName)  
            index += 1  
    return balanced and s.isEmpty()
```

- 将热土豆问题的模拟程序,修改为模拟“击鼓传花”,即每次传递数不是常量值,而是一个随机数。

```
import random
def randomizedHotPotato(namelist):
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name)

    while simqueue.size() > 1:
        num = random.randint(0, simqueue.size())
        for i in range(num):
            simqueue.enqueue(simqueue.dequeue())
        simqueue.dequeue()
    return simqueue.dequeue()
```