栈 线性表 队列和广搜

吴益强

Python 版

日期：2022 年 4 月 2 日

1 栈

1.1 栈的定义

类似于子弹匣，后压进去的子弹，先射出去。支持四种操作：

```
top()       返回栈顶元素
push(x)     将 x 压入栈中
pop()       弹出并返回栈顶元素
isEmpty()   看栈是否为空
```

注意：要求上面操作复杂度都是 O(1)

1.2 栈的实现

用列表可以实现栈，四种操作的实现 (stack 为一个列表)：

```
top()       stack[1]
push(x)     stack.append
pop()       stack.pop
isEmpty()   len (stack) == 0
```

1.3 例题

一、字符串中的括号配对

字符串中可能有三种成对的括号，"[(","])"。判断字符串的括号是否都正确配对了不存在括号也算正确配对)。括号交叉算不正确配对，例如"1234[78" 。但是一对括号被包含在另一对括号里面，例如"12ab[k] 不影响正确性。

解题思路：从头到尾扫描字符串，碰到左括号就入栈。碰到右括号，就要求栈顶必须是一个和它配对的左括号，如果不是，则断定字符串不符合要求。如果是，则弹出栈顶。字符串扫描结束时，栈为空则为正确，不为空则为错误。

代码示例：

```
def match(s):            复杂度 O(n)
    stack = []
```

```
    pairs = {")":"(","]":"[","}":"{"}
    for x in s:
        if x in "{[(":
            stack.append(x)
        elif x in "}])":
            if len(stack) == 0 or stack[-1] != pairs[x]:
                return False
            stack.pop()
    return len(stack) == 0
print(match(input()))
```

二、后序表达式求值

运算符：+ - * /其中 * / 优先级高于 + -。原子：整数或者小数。

后序表达式递归定义：

1）一个原子是一个后序表达式

2) 两个后序表达式 a、b 加上一个运算符 c，是一个后序表达式。a b c 的计算方法和传统的中序表达式 (a) c (b) 一样。

以下都是后序表达式原子、运算符之间用空格分隔

```
3.4
5
5 3.4 +
等价于 5 + 3.4
5 3.4 +
6 / 等价于 (5+3.4)/6
5 3.4 +
6 * 3 + 等价于 (5+3.4)*6 + 3
```

解题思路：从左到右扫描一遍后序表达式，碰到原子就入栈，碰到运算符，就取出栈顶两个元素进行运算，并将结果压入栈中。扫描结束时，栈里应该只有一个元素，就是后序表达式的值。

代码示例：

```
def countSuffix(s):      计算后序表达式 s 的值，复杂度 O(
    s = s.split()
    stack = []
    for x in s:
        if x in "+-*/":
            a,b = stack.pop(),stack.pop()
            stack.append(eval(str(b) + x + str(a)))
        else:
            stack.append(x)
    return stack[0]
```

三、快速堆猪

小明有很多猪，他喜欢玩叠猪游戏，就是将猪一头头叠起来。猪叠上去后，还可以把顶上的猪拿下来。小明知道每头猪的重量，而且他还随时想知道叠在那里的猪最轻的是多少斤。

有三种输入

```
1)push n
n是整数(0<=0 <=20000)，表示叠上一头重量是n斤的新猪
2)pop
表示将猪堆顶的猪赶走。如果猪堆没猪，就啥也不干
3)min
表示问现在猪堆里最轻的猪多重。如果猪堆没猪，就啥也不干
```

样例输入

```
pop
min
push 5
push 2
push 3
min
push 4
min
```

样例输出

```
2
2
```

代码示例:

```
pig = []
min_pig = []
stack = []
minpig = 0
while True:
    try:
        x = input()
        if len(pig) == 0 :
            if x[:4] == 'push':
                n = int(x[4:])
                minpig = n
                stack.append(minpig)
                pig.append(n)
            else:
                continue
        elif len(pig) > 0:
            if x == 'pop':
                pig.pop()
                stack.pop()
                if len(pig) > 0:
                    minpig = stack[-1]
            elif x == 'min' :
                min_pig.append(minpig)
```

```
            elif x[:4] == 'push':
                n = int(x[4:])
                pig.append(n)
                if n < minpig:
                    minpig = n
                    stack.append(minpig)
                else:
                    stack.append(minpig)
    except EOFError:
        break
for j in min_pig:
    print(j)
```

2 线性表

2.1 顺序表

- 即 Python 的列表，以及其它语言中的数组
- 元素在内存中连续存放;
- 根据下标访问元素时间 O(1).
- 在头部或中间插入删除元素时间 O(n)
- 在尾部添加、删除元素时间 O(1) 通过预先多分配已有元素固定倍数的空间来实现
- 几乎不需要花费额外存储空间

2.2 链表

- 元素在内存中并非连续存放
- 访问第 i 个元素，复杂度为 O(n)
- 已经找到插入或删除位置的情况下，插入和删除元素的复杂度 O(1)
- 有多种形式：单链表、循环单链表、双向链表、循环双向链表

1. 单链表

```
链表结构形式:
class LinkList:
    def __init__(self,head = None,size = 0):
        self.head,self.size = head,size
head:表头元素指针
size:链表元素个数
```

```
一个节点的表示形式:
class Node:
    def __init__(self,data,None):
        self.data, self.next = data, next
data:数据
```

```
next:指向下一个节点的指针，即下一个节点。链表最后一个结点该值为None
```

```
操作复杂度:
 表头插入删除: (1)
 表尾添加、删除: O(n) 要先从头开始找到表尾
 在指定位置p进行插入删除:  O(1)
```

```
单链表指定位置插入元素:
nd = Node(data )      新建节点包含数据 data
nd.next = p.next
p.next = nd
还要修改链表的size 。
空链表插入第一个元素的情况单独处理。
```

```
单链表指定位置删除元素:
删除p结点后面的结点:
p.next = p.next.next
被删除的结点会被Python解释器自动回收
```

2. 循环单链表

```
链表结构形式:
class LinkList:
    def __init__(self,tail = None,size = 0):
        self.tail,self.size = tail,size
tail:表尾元素指针,tail.next算表头
size:链表元素个数
```

```
操作复杂度:
 表头插入、删除 O(1)
 表尾添加:  O(1)
 表尾删除 : O(n) 要从表头开始找到表尾前面那个结点
```

```
class Node:
  def __init__(self, data, next=None):
    self.data, self.next = data, next
class LinkList:  #循环链表
  def __init__(self):
    self.tail = None
    self.size = 0
  def isEmpty(self):
    return self.size == 0
  def pushFront(self,data):
    nd = Node(data)
    if self.tail == None:
      self.tail = nd
      nd.next = self.tail
```

```
    else:
      nd.next = self.tail.next
      self.tail.next = nd
    self.size += 1
  def pushBack(self,data):
    self.pushFront(data)
    self.tail = self.tail.next
  def popFront(self):
    if self.size == 0:
      return None
    else:
      nd = self.tail.next
      self.size -= 1
      if self.size == 0:
        self.tail = None
      else:
        self.tail.next = nd.next
    return nd.data
  def printList(self):
    if self.size > 0:
      ptr = self.tail.next
      while True:
        print(ptr.data,end = " ")
        if ptr == self.tail:
          break
        ptr = ptr.next
      print("")

  def remove(self,data):
    if self.tail:
      flag = False
      for _ in range(self.size):
        val = self.popFront()
        if flag == False and val == data:
          flag = True
        else:
          self.pushBack(val)
      return flag
```

3. 双向链表

```
链表结构形式:
class LinkList:
    def __init__(self,head = None,tail = None,size = 0):
        self.head self.tail, self.size = head,tail,size
head:表头元素指针
size:链表元素个数
tail:表尾元素指针
```

```
表结点结构形式:
class Node:
    def __init__(self , data, prev = None, next= None):
        self.data, self.prev, self.next = data,prev,next
data:数据
prev:指向上一个节点的指针, 即上一个节点。链表头一个结点该值为 None
next:指向下一个节点的指针, 即下一个节点。链表最后一个结点该值为 None
```

```
操作复杂度:
两端增删元素: O(1)
```

4. collections.deque 结合链表和顺序表的特点。是一张双向链表，每个结点是一个 64 个元素的顺序表。

3 队列和广搜

3.1 队列的概念

即排队的队列。只能一头进 (push)，另一头出 (pop). 先进先出. 要求进出的复杂度都是 O(1). 如果用列表的 append 进，pop(0) 出，则出的复杂度为 O(n).

3.2 队列的实现

队列实现方法一:

用足够大的列表实现，维护一个队头指针和队尾指针，初始：head = tail = 0

```
1、head指向队头元素, tail指向队尾元素的后面
2、push(x) 的实现:
    queue[tail] = x
    tail+=1
3、pop() 的实现:
    head+= 1
4、判断队列是否为空:
    head == tail
```

队列实现方法二:

如果不想浪费空间开足够大的列表，而是想根据实际情况分配空间，则可以用列表头尾循环法实现队.

```
1) 预先开设一个 capacity 个空元素的列表 queue head = tail = 0
2) 列表没有装满的情况下:
  1、push(x) 的实现:
    queue[tail] = x
    tail = (tail+1) % capacity
  2、pop() 的实现:
```

```
    head = (head+1) % capacity
  3、判断队列是否为空:
    head == tail
  (capacity可以是 4,8,16.....)
3)若一个 push 操作后导致列表满:
  1、建一个大小是原列表 k 倍大的新列表 (k>1 可以取 1.5,2.....)
  2、将原列表内容全部拷贝到新列表, 作为新队列
  3、重新设置新列表的 head 和 tail
  4、原列表空间自动 被 Python 解释器回收
```

注意：导致队列满的 push 的时间复杂度是 O(n)。平均 push 操作是 O(1) Python 列表.append() 做到 O(1) 的实现也是这种原理，且 k 取 1.125 ，空间换时间. 若每次增加空间只增加固定数量，比如 20 个单元，则 push 平均复杂度还是 O(n)

Python 中的队列 collections 库中的 deque 是双向队列，可以像普通列表一样访问，且在两端进出，复杂度都是 O(1)

```
import collections
dq = collections.deque
dq.append('a') #右边入队
dq.appendleft(2) #左边入队
dq.extend([100,200]) #右边加入 100,200
dq.extendleft(['c','d']) #左边依次加入 'c','d'
print(dq.pop ()) #>>200 右边出队
print(dq.popleft ()) #>>d 左边出队
print(dq.count ('a')) #>>1
dq.remove('c')
print(dq) #>>deque ([2,'a',100])
dq.reverse()
print(dq) #>>deque ([100, 'a',2])
print(dq[0],dq [-1],dq [1]) #>>100 2 a
print(len(dq)) #>>3
```

3.3 广度优先搜索

广度优先搜索算法如下：（用 QUEUE）

(1) 把初始节点 S0 放入 Open 表中；

(2) 如果 Open 表为空，则问题无解，失败退出；

(3) 把 Open 表的第一个节点取出放入 Closed 表，并记该节点为 n；

(4) 考察节点 n 是否为目标节点。若是，则得到问题的解，成功退出；

(5) 若节点 n 不可扩展，则转第 (2) 步；

(6) 扩展节点 n ，将其不在 Closed 表和 Open 表中的子节点 (判重) 放入 Open 表的尾部，并为每一个子节点设置指向父节点的指针 (或记录节点的层次)，然后转第 (2) 步。

3.4 广度优先搜索例题

1、假设农夫起始位于点 3，牛位于 5，N=3,K=5，最右边是 6。如何搜索到一条走到 5 的路径?

2、一个矩阵，它表示一个迷宫，其中的 1 表示墙壁，0 表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。

基础广搜。先将起始位置入队列每次从队列拿出一个元素，扩展其相邻的 4 个元素入队列要用二维标志列表判重)，直到队头元素为终点为止。队列里的元素记录了指向父节点（上一步）的指针。

队列元素：(r,c,father)

r,c: 节点的坐标

father : 父节点在队列中的下标从 a 走到 b, 则 a 是 b 的父节点。

判重的二维列表：flags[i][j] 表示 (i,j) 那个位置是否走过，即是否入过队列迷宫.