2015年6月24日 星期三

第二章 算法分析

一、何为算法分析

- 1、程序与算法的区别
- 算法是对问题解决的分步描述；而程序是用某种具体编程语言实现的算法
- 2、如何评估算法的好坏
- 多种指标（可读性、风格等），我们关注计算资源消耗［能否高效利用］
- 3、计算资源指标
- （1）算法解决问题时所需的空间和内存［难以区分数据占用、算法占用->不好用］
- （2）算法执行时间［用time模块测试］
- 只用时间作标准的坏处：同一种算法，语言不同或机器不同，时间差别很大

```
import time
def f(n):
    start = time.time()
    ....
    end = time.time()
    return end-start
```

- 4、大O表示法
- （1）基本操作数量函数T(n)：
- 一个算法所实施的操作数量和步骤数可以独立于具体程序和机器的度量指标
- 赋值语句同时包含了（表达式）计算和（变量）存储两个基本资源，很合适
- ［掌握看代码计算T(n)的方法］
- （2）数量级函数f(n)：
- 描述了T(n)中随n增加而增加最快的部分，是T(n)的主导部分
- （3）大O表示法：O(f(n))
- 分最好、最坏、平均情况，平均情况反映主流性能
- （4）常见数量级函数

1	logn	n	nlogn	n^2	n^3	2^n
常数	对数	线性	对数线性	平方	立方	指数

- 二、python数据结构的性能

- 1、对比list和dictionary的功能

operation	List	Dict [罕见情况下性能会劣化]
index[] = get item	O(1)	O(1)
index assignment = set item	O(1)	O(1)
append	O(1)	
pop()	O(1)	
pop(i)	O(n)	
insert(i,item)	O(n)	
del[]	O(n)	O(1)
contains —— in	O(n)	O(1)
get slice[x:y]	O(k)	
set slice	O(n+k)	
del slice	O(n)	
iteration???	O(n)	O(n)
reverse	O(n)	
concatenate —— +	O(k)	
sort	O(nlogn)	
multiply —— *	O(nk)	
copy????		O(1)

第三章 基本数据结构

一、线性数据结构(Linear Structure)

1、定义：数据项之间只存在先后的次序关系，新的数据项加入数据集中时，只会加入到原有的某个数据项之前或之后，具有这种性质的数据集即为线性数据结构。

2、其他：总有两端（称呼方式不同：左右、前后、顶底）；然而称呼并不是重点，不同数据结构的区别在于数据项增减的方式

二、栈(Stack)

1、定义：一种有次序的数据项集合，每个数据项仅从“栈顶”一端加入数据集中、从数据集中移除，具有后进先出LIFO的特性。（LIFO的定义:先入后出、等待时间长短...）

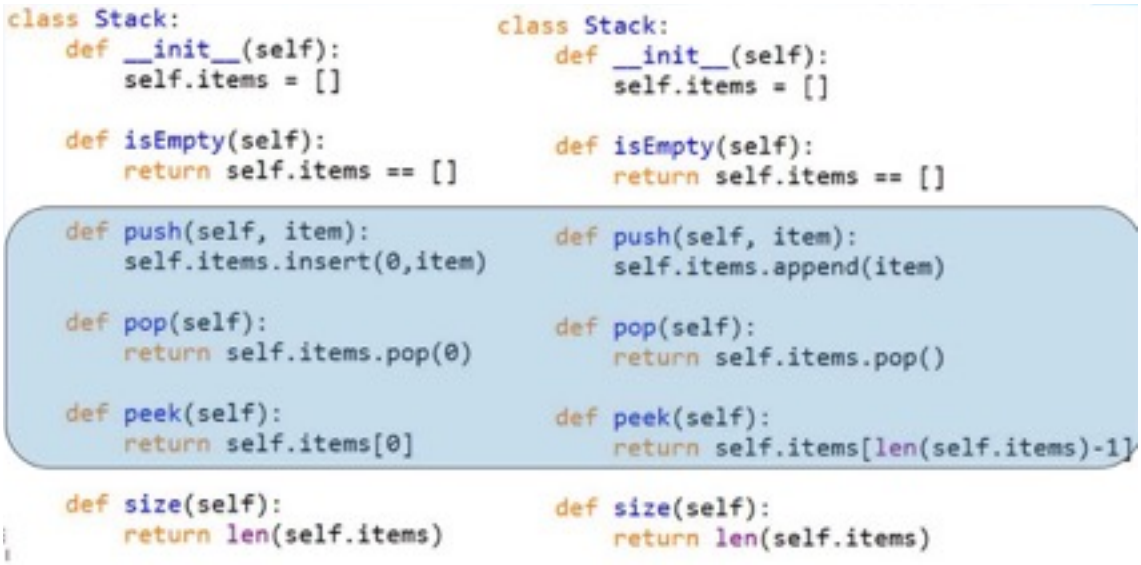
2、特性：反转次序， LIFO

3、Stack的操作

Stack类函数接口	备注	栈顶首端版 性能	栈顶尾端版 性能
Stack()	创建空栈		
push(item)	压入栈顶，无返回值	O(n)	O(1)
pop()	移除，返回栈顶数据项	O(n)	O(1)
peek()	返回栈顶数据项		
isEmpty()	返回是否为空栈		
size()	返回栈内数据项数目		

4、ADT Stack 的两种实现方式

左为栈顶首端，右为栈顶尾端



```
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)
```

三、队列（Queue）

1、 定义：是一个有次序的数据集合，数据项仅添加到尾“rear”端，而且仅从首端“front”移除，具有FIFO 的操作次序。（FIFO的定义：扯一扯）

2、特性：FIFO

3、Queue的操作

Queue类函数接口	备注	队首首端版 性能	队首尾端版 性能
Queue()	创建空队列		
enqueue(item)	添加到队尾，无返回值	O(n)	O(1)

Queue类函数接口	备注	队首首端版 性能	队首尾端版 性能
dequeue()	从队首移除，返回该项	O(1)	O(n)
isEmpty()	返回是否为空队列		
size()	返回队列内数据项数目		

4、Queue的实现方式（队首首端版）

```
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)
```

四、双端队列(Deque)

1、定义：一种有次序的数据项集合，数据项可以从两端加入或移除。（某种意义上集成了栈和队列的能力）

2、特性：不具有FILO、LILO等（因此需要用户自己维护特性）

3、Deque的操作

Deque类函数接口	备注	队首尾端版 性能
Deque()	创建空双端队列	
addFront(item)	添加到队首，无返回值	O(1)
addRear(item)	添加到队尾，无返回值	O(1)
removeFront()	从队首移除，返回该项	O(n)
removeRear()	从队尾移除，返回该项	O(n)
isEmpty()	返回是否为空双端队列	

Deque类函数接口	备注	队首尾端版 性能
size()	返回数据项数目	

4、ADT Deque的实现方式（队首尾端版）

```
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)
```

五、无序表(List 或 Unordered List)

1、定义：无序表是一个数据集，其中每个数据项都相对于其他数据项有一个位置。其中数据项只按照存放的相对位置来索引。

2、List的操作

［注：对于复杂度，python内置的list是基于数组的，并进行了优化，所以会比链表版性能高］

List类函数接口	备注	链表实现版 性能
list()	创建无序表	
add(item)	添加到列表中	O(1)
search(item)	返回布尔值	O(n)
remove(item)		O(n)
isEmpty()	返回是否为空列表	O(1)
size()	返回数据项数目	O(n)
append(item)		

List类函数接口	备注	链表实现版 性能
index(item)	返回该数据项的位置	
insert(pos,item)	插入后数据项位于pos位置	
pop()		
pop(pos)		

3、ADT List的实现方式(用链表来实现)

```
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
```

```
class UnorderedList:

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

```
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
```

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

    return count
```

```
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())
```

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

五、有序表(Ordered List)

1、定义：有序表是一个数据集，其中每个数据项都依照某可比性质（如整数的大小）来决定在列表中的位置，越“小”的项越靠近列表的头。

2、Ordered List的操作

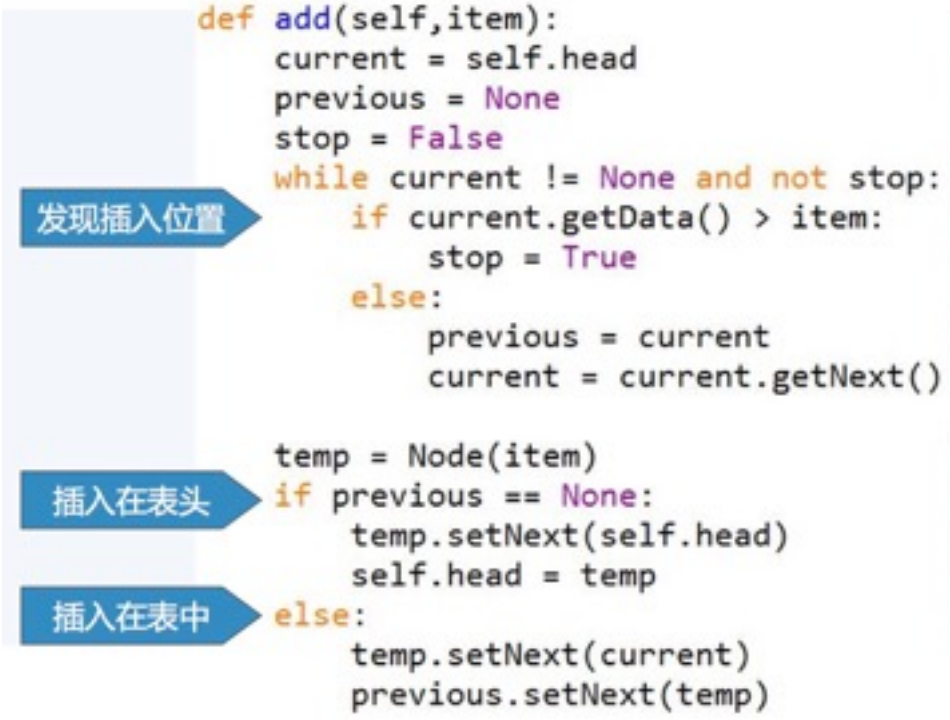
OrderedList类函数接口	备注	链表实现版 性能
OrderedList()	创建无序表	
add(item)	添加到列表中	O(n)
search(item)	返回布尔值	O(n)
remove(item)		O(n)
isEmpty()	返回是否为空列表	O(1)
size()	返回数据项数目	O(n)
index(item)	返回该数据项的位置	
pop()		
pop(pos)		

3、ADT OrderedList的实现方式(用链表来实现)

大部分方法跟list差不多，但add 和search需要重新做

```
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
```




```
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)
```

第四章 递归

一、递归(Recursion)

1、定义：递归是一种解决问题的方法，其精髓是将问题分解为规模更小的相同问题，持续分解，直到问题规模小到可以用非常简单直接的方式来解决。

2、特征：递归对问题分解方式奇特，最基本的算法特点就是，在算法流程中调用自身。

二、递归三定律

1、递归算法必须有一个基本结束条件（最小规模问题的直接解决）

2、递归算法必须能改变状态向基本结束条件演进（减小问题规模）

3、递归算法必须调用自身（解决减小了规模的相同问题）

三、递归的内部实现

1、当一个函数被调用的时候，系统会把调用时的现场（包括所有对局部变量，以及返回地址）压入到调用栈Call Stack；

2、每次压入栈到现场数据称为栈帧；

3、当函数执行完成，返回时，要从调用栈的栈顶取得返回地址，把返回值放到栈顶，恢复现场，弹出栈帧，按地址返回。

四、缓存

使用一个表将计算过程中已计算过的最优值保存下来，以后每次计算前先查找表里是否已有该值，若没有再进行计算并补充表。［计算中间结果的表有不少空洞］

五、动态规划

从最简单的情况开始到达所需问题规模的循环，每一步都依靠以前的最优解来得到本步骤的最优解，直到得到答案。

第五章 排序与搜索

一、顺序搜索

1、定义：如果数据项保存在如列表这样的集合中，我们称这些数据项有线性关系或顺序关系，通过下标，我们可以顺序访问和查找数据项，这种技术就称为“顺序搜索”。从列表中第一个数据项开始逐个比对，若到最后还没找到，则搜索失败。

2、无序表的顺序搜索：基本计算步骤是数据的比对

性能如下

case	best case	worse case	average case
item in present	1	n	n/2
item not in present	n	n	n

3、有序表的顺序搜索：基本计算步骤是数据的比对；在所搜数据不在表中的情况下能节省一些比对次数，但总复杂度还是O(n)

case	best case	worse case	average case
item in present	1	n	n/2
item not in present	1	n	n/2

二、二分搜索