

消耗时间大小

O(1) < O(logn) < O(n) < O(nlogn) < O(n2) < O(n3) < O(2n) < O(n!) < O(nn)

**timeit模块**

timeit模块可以用来测试一小段Python代码的执行速度。

**class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>)**

Timer是测量小段代码执行速度的类。

stmt参数是要测试的代码语句（statment）；

setup参数是运行代码时需要的设置；

timer参数是一个定时器函数，与平台有关。

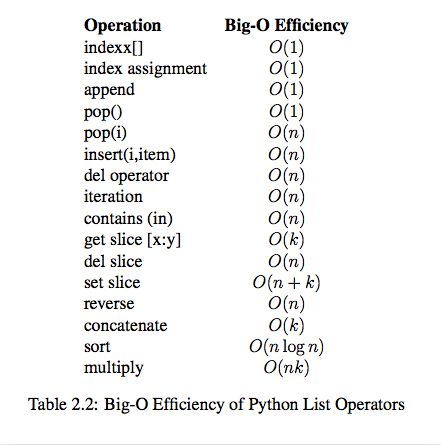
**timeit.Timer.timeit(number=1000000)**

Timer类中测试语句执行速度的对象方法。number参数是测试代码时的测试次数，默认为1000000次。方法返回执行代码的平均耗时，一个float类型的秒数。

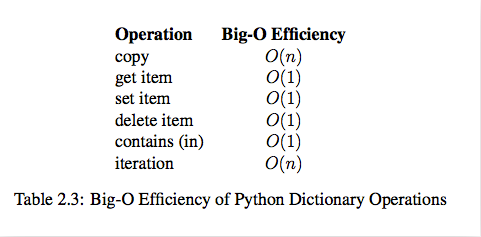
eg:time1 = timeit.Timer(“func”,”from \_\_main\_\_ import func”)

print(time1.timeit(1000))

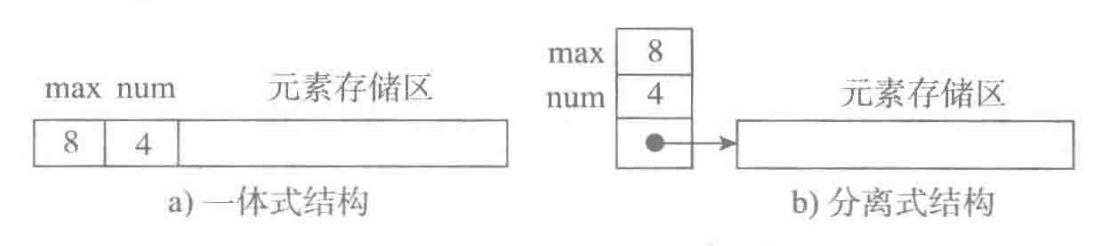
**list内置操作的时间复杂度**



**dict内置操作的时间复杂度**



**顺序表的两种基本实现方式**



**扩充的两种策略**

1.每次扩充增加固定数目的存储位置，如每次扩充增加10个元素位置，这种策略可称为线性增长。

特点：节省空间，但是扩充操作频繁，操作次数多。

2.每次扩充容量加倍，如每次扩充增加一倍存储空间。

特点：减少了扩充操作的执行次数，但可能会浪费空间资源。以空间换时间，推荐的方式。

**链表与顺序表的对比**



**节点实现**

class SingleNode(object):

"""单链表的结点"""

def \_\_init\_\_(self,item):

# item存放数据元素

self.item = item

# next是下一个节点的标识

self.next = None

**单链表的操作**

* is\_empty() 链表是否为空
* length() 链表长度
* travel() 遍历整个链表
* add(item) 链表头部添加元素
* append(item) 链表尾部添加元素
* insert(pos, item) 指定位置添加元素
* remove(item) 删除节点
* search(item) 查找节点是否存在

**单链表的实现**

class SingleLinkList(object):

"""单链表"""

def \_\_init\_\_(self):

self.\_\_head = None

def is\_empty(self):

"""判断链表是否为空"""

return self.\_\_head == None

def length(self):

"""链表长度"""

# cur初始时指向头节点

cur = self.\_\_head

count = 0

# 尾节点指向None，当未到达尾部时

while cur != None:

count += 1

# 将cur后移一个节点

cur = cur.next

return count

def travel(self):

"""遍历链表"""

cur = self.\_\_head

while cur != None:

print cur.item,

cur = cur.next

print ""

**头部添加元素**

def add(self, item):

"""头部添加元素"""

# 先创建一个保存item值的节点

node = SingleNode(item)

# 将新节点的链接域next指向头节点，即\_head指向的位置

node.next = self.\_\_head

# 将链表的头\_head指向新节点

self.\_\_head = node

**尾部添加元素**

def append(self, item):

"""尾部添加元素"""

node = SingleNode(item)

# 先判断链表是否为空，若是空链表，则将\_head指向新节点

if self.is\_empty():

self.\_\_head = node

# 若不为空，则找到尾部，将尾节点的next指向新节点

else:

cur = self.\_\_head

while cur.next != None:

cur = cur.next

cur.next = node

**指定位置添加元素**

def insert(self, pos, item):

"""指定位置添加元素"""

# 若指定位置pos为第一个元素之前，则执行头部插入

if pos <= 0:

self.add(item)

# 若指定位置超过链表尾部，则执行尾部插入

elif pos > (self.length()-1):

self.append(item)

# 找到指定位置

else:

node = SingleNode(item)

count = 0

# pre用来指向指定位置pos的前一个位置pos-1，初始从头节点开始移动到指定位置

pre = self.\_\_head

while count < (pos-1):

count += 1

pre = pre.next

# 先将新节点node的next指向插入位置的节点

node.next = pre.next

# 将插入位置的前一个节点的next指向新节点

pre.next = node

**删除节点**

def remove(self,item):

"""删除节点"""

cur = self.\_\_head

pre = None

while cur != None:

# 找到了指定元素

if cur.item == item:

# 如果第一个就是删除的节点

if not pre:

# 将头指针指向头节点的后一个节点

self.\_\_head = cur.next

else:

# 将删除位置前一个节点的next指向删除位置的后一个节点

pre.next = cur.next

break

else:

# 继续按链表后移节点

pre = cur

cur = cur.next

**查找节点是否存在**

def search(self,item):

"""链表查找节点是否存在，并返回True或者False"""

cur = self.\_\_head

while cur != None:

if cur.item == item:

return True

cur = cur.next

return False

排序

冒泡

*def* mao(*list1*):  
 n = len(*list1*)  
 *for* i *in* range(n-1):  
 *for* j *in* range(n-i-1):  
 *if list1*[j]>*list1*[j+1]:  
 *list1*[j],*list1*[j+1] = *list1*[j+1],*list1*[j]

**二叉树的性质(特性)**

**性质1:** 在二叉树的第i层上至多有2^(i-1)个结点（i>0）  
**性质2:** 深度为k的二叉树至多有2^k - 1个结点（k>0）  
**性质3:** 对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1;  
**性质4:**具有n个结点的完全二叉树的深度必为 log2(n+1)  
**性质5:**对完全二叉树，若从上至下、从左至右编号，则编号为i 的结点，其左孩子编号必为2i，其右孩子编号必为2i＋1；其双亲的编号必为i/2（i＝1 时为根,除外）

哈希表(散列表)