**算法引入（3.3）：**

####import time的方法记录时间

import time

star\_time = time.time()

for a in range(0,1001):

    for    b in range(0,1001):

        for c in range(0,1001):

            if a+b+c == 1000 and a\*\*2 + b\*\*2 == c\*\*2:

                print("a,b,c:%d,%d,%d:"%(a,b,c))

end\_time = time.time()

print("time:%d"%(end\_time-star\_time))

print("finished")

a,b,c:0,500,500:

a,b,c:200,375,425:

a,b,c:375,200,425:

a,b,c:500,0,500:

time:445

finished

[Finished in 445s]

T = 1000\*1000\*1000\*2 步骤

抽象为 T = n \* n \*n的时间复杂度

T(n)=n\*\*3

####改进后的

import time

star\_time = time.time()

for a in range(0,1001):

    for    b in range(0,1001):

        c = 1000 - a - b

        if  (a\*\*2 + b\*\*2 == c\*\*2):

            print("a,b,c:%d,%d,%d:"%(a,b,c))

end\_time = time.time()

print("time:%d"%(end\_time-star\_time))

print("finished")

a,b,c:0,500,500:

a,b,c:200,375,425:

a,b,c:375,200,425:

a,b,c:500,0,500:

time:4

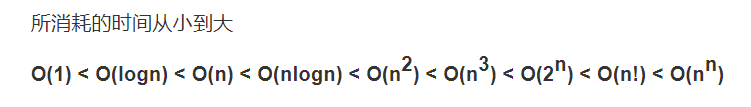
finished

[Finished in 4.9s]

T(n)=n\*\*2

**时间复杂度（3.3）:**

时间、步骤越多，则越复杂。n代表规模，抽象出来n的渐进函数



最坏时间复杂度：完成工作时，最大可能要跑的步骤，保证在这个范围内，处理此程度的问题，都能完成。

eg:

排序列表，【3,2,4,5,3,7,8】可能用到两重循环，时间复杂度为n\*n

排序列表，【3,2,4,5,3,7,8】可能用到一重循环，时间复杂度为n

因此最坏时间复杂度为O(n\*n)

时间复杂度的几条基本计算规则:

1. 基本操作，即只有常数项，认为其时间复杂度为O(1)

2. 顺序结构，时间复杂度按**加法**进行计算，步骤相加即可

3. 循环结构，时间复杂度按**乘法**进行计算，一个循环，就是一个步骤

4. 分支结构，时间复杂度取**最大值**，哪个分支大，那么就取时间复杂度最大的那个

5. 判断一个算法的效率时，往往只需要关注操作数量的**最高次项**，其它次要项和常数项可以忽略

6. 在没有特殊说明时，我们所分析的算法的时间复杂度都是指**最坏时间复杂度**

**7.函数另算时间复杂度**

**list操作的复杂度**

**尾部追加、弹出，比从开始或中间时间复杂度大。**

from timeit import Timer

def test1():

    l = []

    for i in range(1000):

        l = l + [i]

def test2():

    l = []

    for i in range(1000):

        l.append(i)

def test3():

    l = [i for i in range(1000)]

def test4():

    l = list(range(1000))

def test5():

    l = []

    for i in range(1000):

        l.extend([i])

def test6():

    l = []

    for i in range(1000):

        l.append(i)

def test7():

    l = []

    for i in range(1000):

        l.insert(0,i)

t1 = Timer("test1()","from \_\_main\_\_ import test1")

print("concat",t1.timeit(number=1000),"seconds")

t2 = Timer("test2()","from \_\_main\_\_ import test2")

print("append",t2.timeit(number=1000),"seconds")

t3 = Timer("test3()","from \_\_main\_\_ import test3")

print("comprehension",t3.timeit(number=1000),"seconds")

t4 = Timer("test4()","from \_\_main\_\_ import test4")

print("list range",t4.timeit(number=1000),"seconds")

t5 = Timer("test5()","from \_\_main\_\_ import test5")

print("extend",t5.timeit(number=1000),"seconds")

t6 = Timer("test6()","from \_\_main\_\_ import test6")

print("add\_tai",t6.timeit(number=1000),"seconds")

t7 = Timer("test7()","from \_\_main\_\_ import test7")

print("add\_zero",t7.timeit(number=1000),"seconds")

x = list(range(2000000))

pop\_zero = Timer("x.pop(0)","from \_\_main\_\_ import x")

print("pop\_zero",pop\_zero.timeit(number=1000),"seconds")

pop\_end = Timer("x.pop()","from \_\_main\_\_ import x")

print("pop\_end",pop\_end.timeit(number=1000),"seconds")

concat 3.241317615 seconds

append 0.22771206399999988 seconds

comprehension 0.11583036200000008 seconds

list range 0.05100204099999983 seconds

extend 0.30615788199999994 seconds

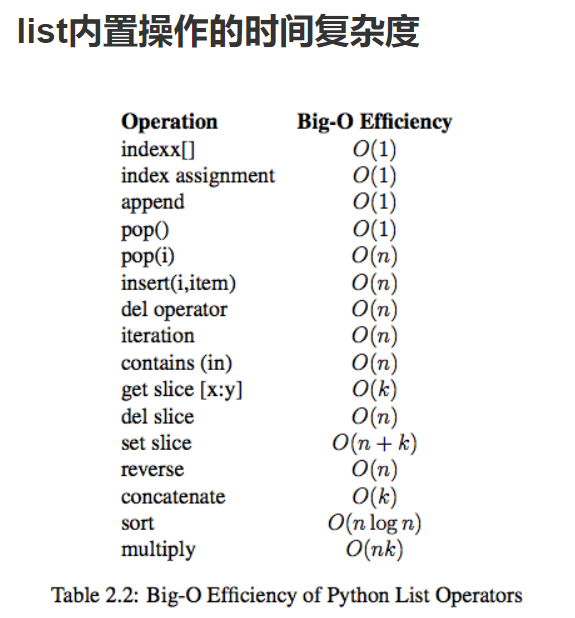
add\_tai 0.1959453 seconds

add\_zero 1.1672260740000002 seconds

pop\_zero 1.6525936790000006 seconds

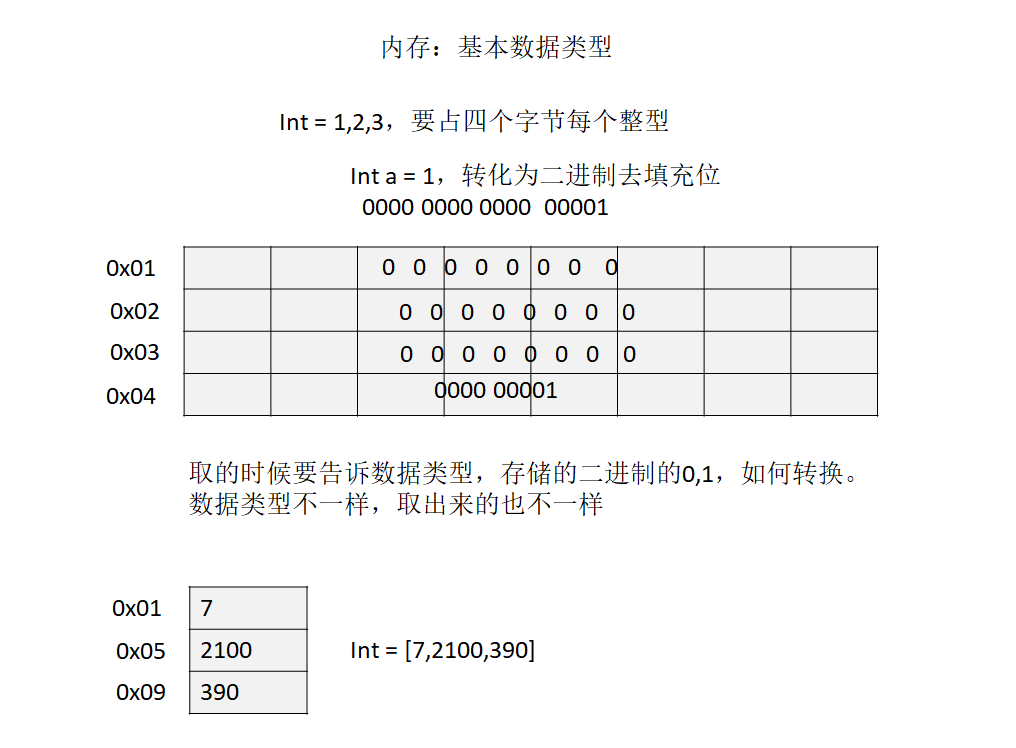
pop\_end 0.0002201599999995807 seconds

[Finished in 8.0s]

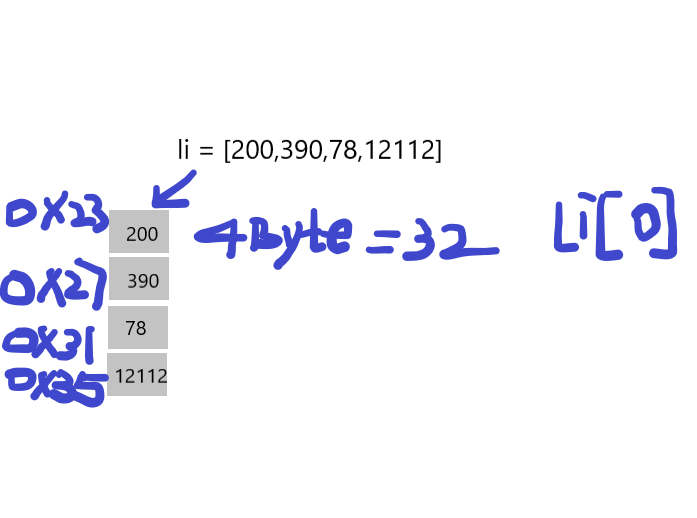


**顺序表（3.4）:**

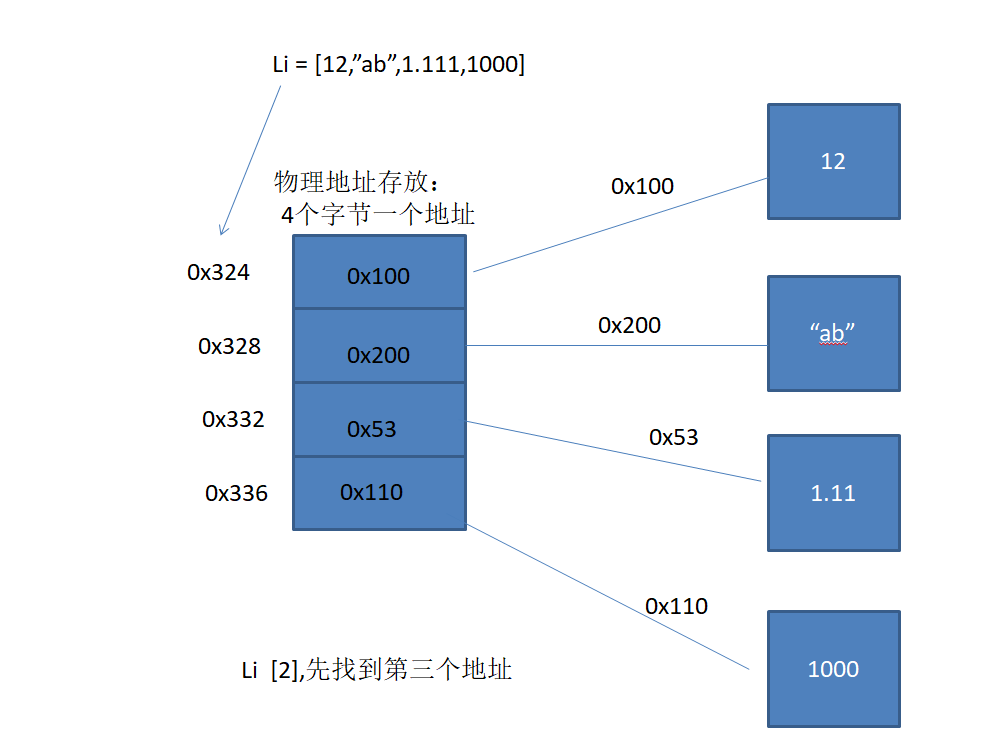
1. ***内存：***



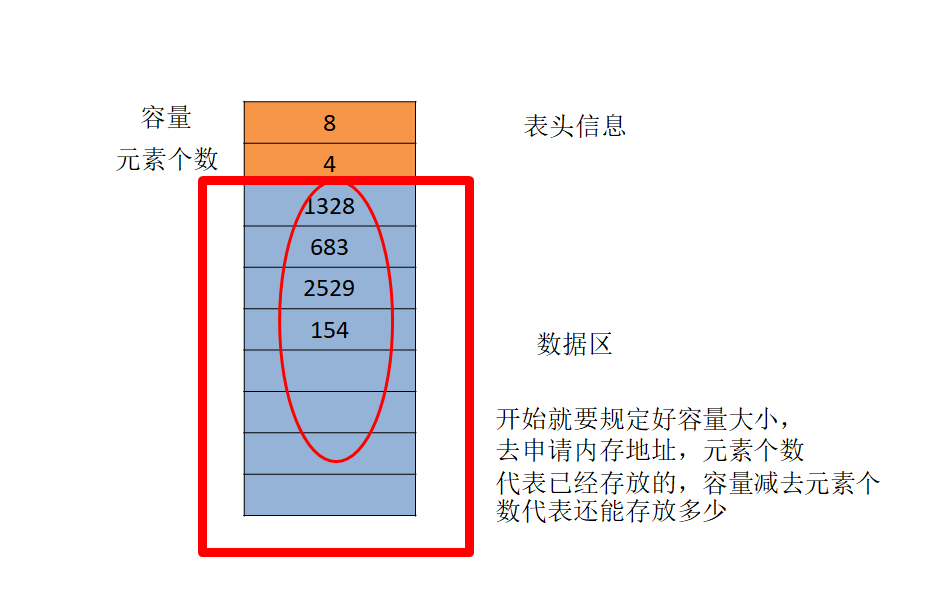
li = [200,390,78,12112],32位系统，一个int占4个字节。设起始位置为0x23，每加一个int，多占4个byte，索引的时候也按地址去找。如：li[3] = 0x23 + 3\*4Byte。按顺序去存储的数据结构叫顺序表。内存地址具有连续性。



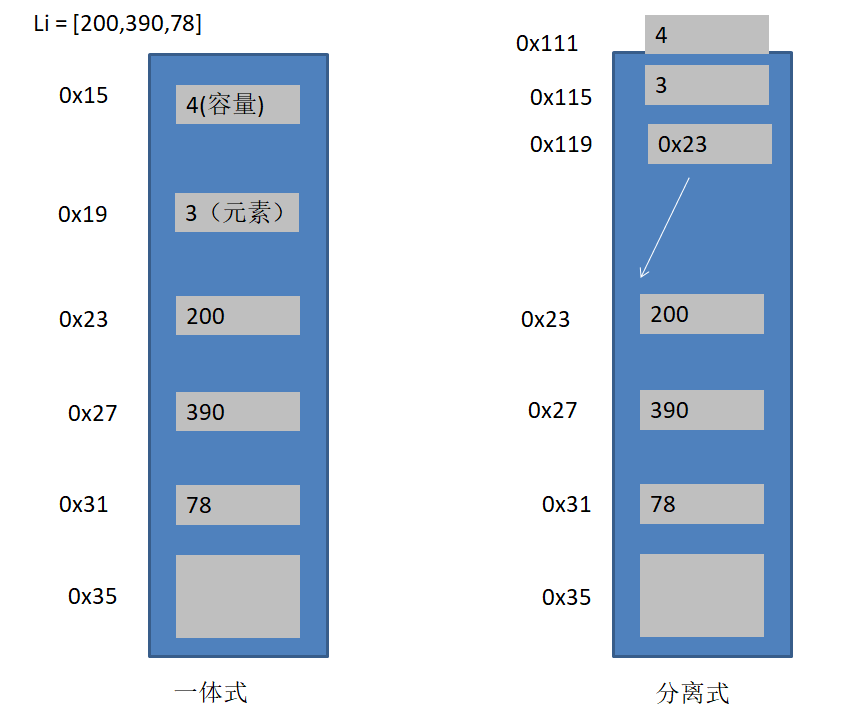
   li = [12,"ab"，1.111，1000],每个数据的大小有不相同的情况。32位的用四个字节（一个字节八位）代表一个地址。每个地址大小一样，都为四个字节。此时采用，申请地址去存数据。地址也可以存储起来，再次申请地址存储地址信息，地址信息为顺序表。通过地址进行指向，找到了地址就找到了数据。



***2.顺序表存储结构：***

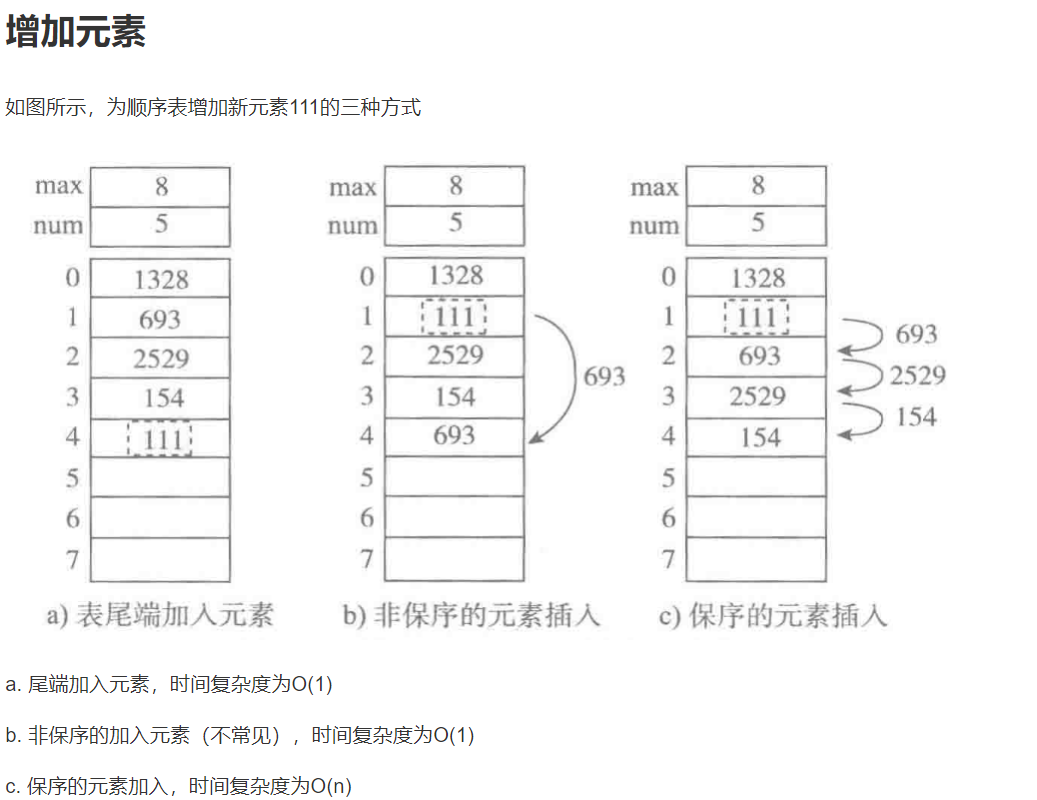


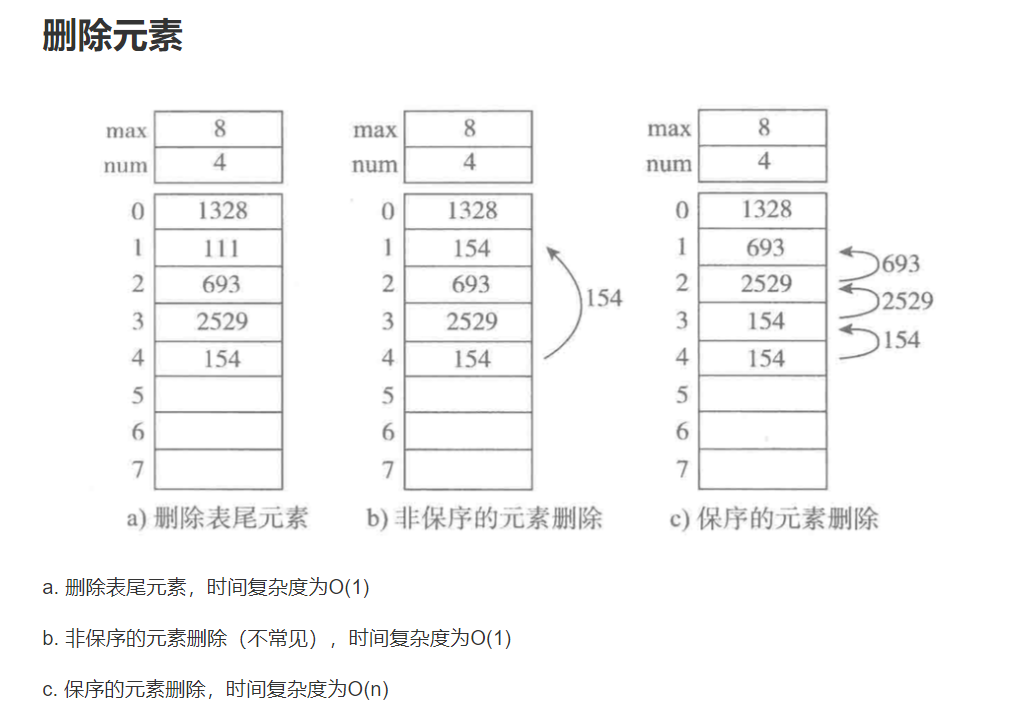
***3.顺序表的两种基本实现方式：***



在一体式结构中，如果要存储超过容量的，要申请新的内存地址，原内存空间释放。所有都都要换做新地址。但是分离式的，只需要原表头的地址指向新的数据存储区域，只需更新数据区域。一体式表头的起始地址（变量名所指向的地址）要指向新的表头地址，分离式的不需要。

***4.顺序表的操作：***

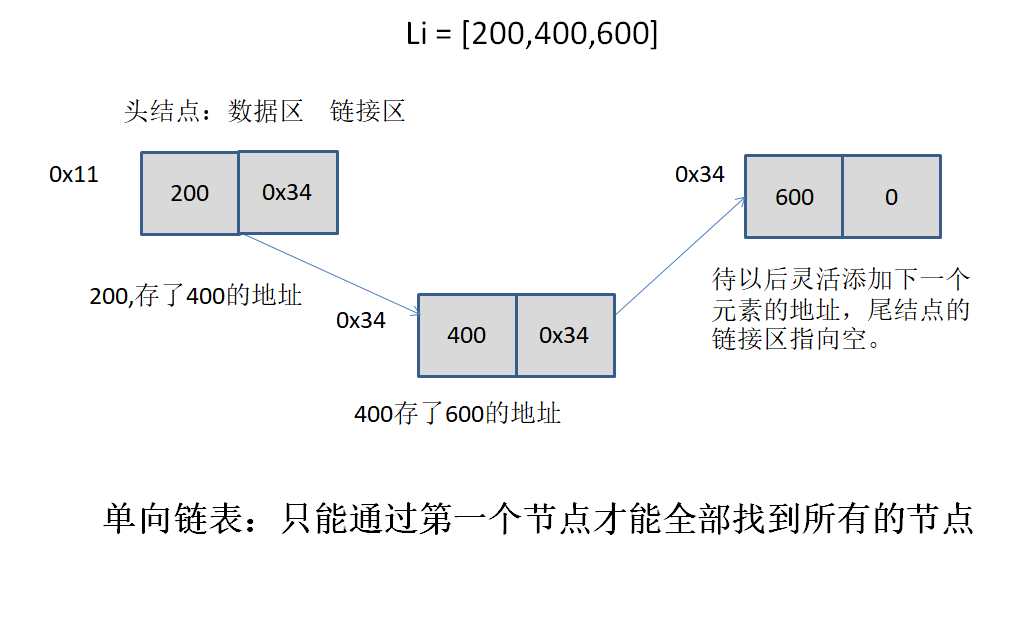




Python中的list和tuple都是顺序表结构。

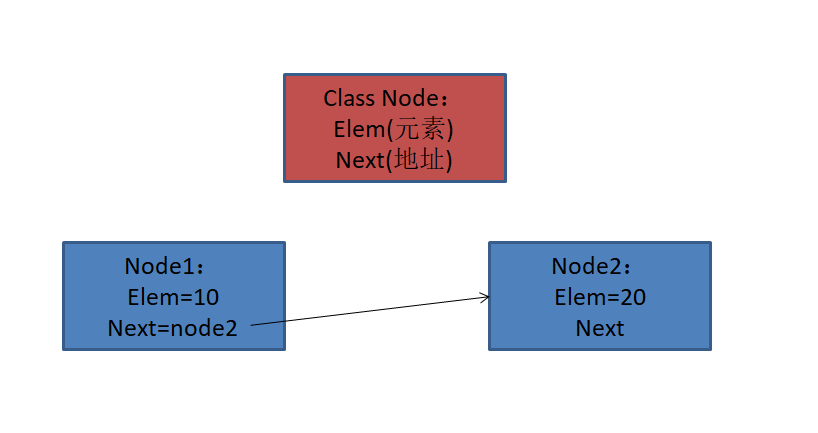
**链表（3.5）:**

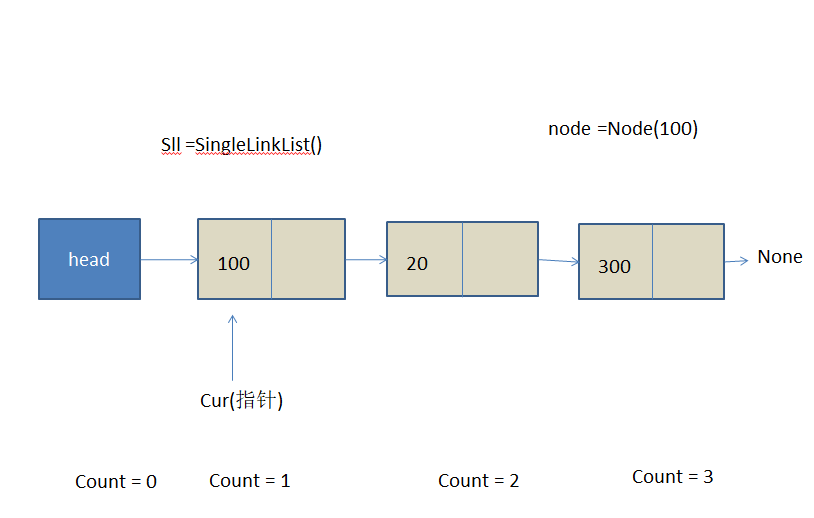
  单向链表也叫单链表，是链表中最简单的一种形式，它的每个节点包含两个域，一个信息域（元素域）和一个链接域。这个链接指向链表中的下一个节点，而最后一个节点的链接域则指向一个空值。是一维空间的一根线串起来了。链表和顺序表都是线性表。



**单链表类的实现（3.6）:**

  代表引用的地址、连接。变量名a可以指向整型数据，函数，类等，赋值的过程相当于改变地址的过程。next指向下一个节点。私有变量





**单链表的操作（3.9）:**

   单向链表也叫单链表，是链表中最简单的一种形式，它的每个节点包含两个域，一个信息域（元素域）和一个链接域。这个链接指向链表中的下一个节点，而最后一个节点的链接域则指向一个空值。

# coding:utf-8

class Node(object):

    """节点"""

    def \_\_init\_\_(self, elem):

        # \_elem 存放数据元素

        self.elem = elem

        # \_next是下一个节点的标识

        self.next = None

class SingleLinkList(object):

    """单链表"""

    def \_\_init\_\_(self, node=None):

        # 私有属性，记录头结点

        self.\_head = node

    def is\_empty(self):

        """链表是否为空"""

        return self.\_head == None

    def length(self):

        """链表长度"""

        # cur游标，用来移动遍历节点

        cur = self.\_head

        # count记录数量

        count = 0

        while cur != None:

            count += 1

            cur = cur.next

        return count

    def travel(self):

        """遍历整个链表"""

        cur = self.\_head

        while cur != None:

            print(cur.elem)

            cur = cur.next

    def add(self, item):

        """链表头部添加元素，头插法"""

        node = Node(item)

        node.next = self.\_head  # 新节点指向原有的首节点

        self.\_head = node  # 原有的首节点指向新节点

    def append(self, item):

        """链表的尾部添加元素,尾插法"""

        node = Node(item)

        cur = self.\_head

        if self.is\_empty():  # 空列表的添加

            self.\_head = node

        else:

            while cur.next != None:

                cur = cur.next

            cur.next = node

    def insert(self, pos, item):

        """制定位置添加元素

        :param pos 从0开始，inser(2,100),下标为2的地方添加100"""

        if pos <= 0:  # 认为在链表最头部插入元素

            self.add(item)

        elif pos > (self.length() - 1):  # 认为尾插

            self.append(item)

        else:

            pre = self.\_head  # pre指针指向第一个元素

            count = 0

            while count < (pos - 1):  # 循环结束后，指针在pos前一个位置停下来了

                count += 1

                pre = pre.next

            node = Node(item)

            node.next = pre.next  # 新元素的next指向pre的下一个元素的地址

            pre.next = node  # pre的next区域指向新元素

    def remove(self, item):

        """删除节点,不是删除下标指向的位置"""

        cur = self.\_head

        pre = None

        while cur != None:

            if cur.elem == item:

                # 先判断此节点是否是头结点

                # 对于头结点来说，

                if cur == self.\_head:

                    self.\_head = cur.next

                else:

                    pre.next = cur.next

                break

            else:

                pre = cur

                cur = cur.next

    def search(self, item):

        """查找节点是否存在"""

        cur = self.\_\_head

        while cur != None:

            if cur.item == item:

                return True

            else:

                cur = cur.next

        return False

if \_\_name\_\_ == "\_\_main\_\_":

    ll = SingleLinkList()

    print(ll.is\_empty())

    print(ll.length())

    ll.append(1)

    print(ll.is\_empty())

    print(ll.length())

    ll.append(2)

    ll.append(3)

    ll.append(4)

    ll.append(5)

    ll.append(6)

    print(ll.is\_empty())

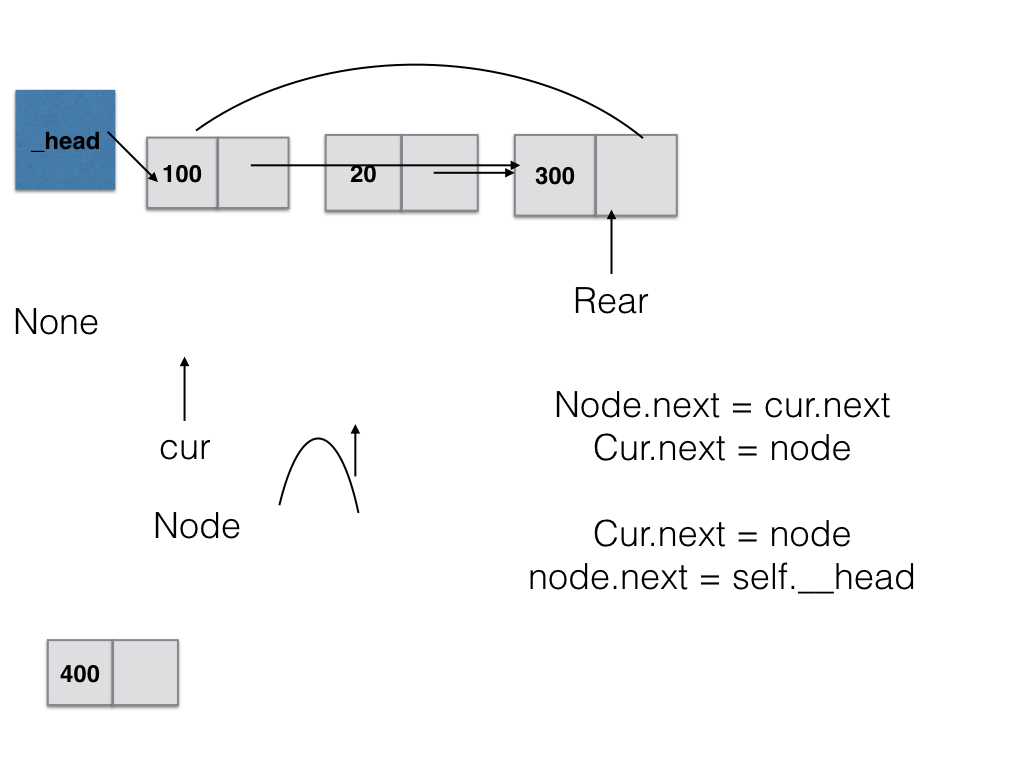
    print(ll.length())

链表的插入删除很快，主要慢在要从头到尾去找元素。顺序表的查找很快，但删除、插入很慢，因为要整体进行位移操作。时间复杂度如下：



**单向循环链表的操作（3.9-3.12）:**

  单链表的一个变形是单向循环链表，链表中最后一个节点的next域不再为None，而是指向链表的头节点。



# coding:utf-8

####单向循环列表

class Node(object):

    """节点"""

    def \_\_init\_\_(self, elem):

        # \_elem 存放数据元素

        self.elem = elem

        # \_next是下一个节点的标识

        self.next = None

class SingleCycleLinkList(object):

    """单项循环链表"""

    def \_\_init\_\_(self, node=None):

        # 私有属性，记录头结点

        self.\_head = node

        if node:

            node.next = node#指向下一个

    def is\_empty(self):

        """链表是否为空"""

        return self.\_head == None

    def length(self):

        """链表长度"""

        # cur游标，用来移动遍历节点

        if self.is\_empty():

            return 0

        cur = self.\_head

        # count记录数量

        count = 1

        while cur.next != self.\_head:

            count += 1

            cur = cur.next

        return count

    def travel(self):

        """遍历整个链表"""

        #空链表不返回值

        if self.is\_empty():

            return

        cur = self.\_head

        while   cur.next != self.\_head:

            print(cur.elem,end=' ')

            cur = cur.next

        #退出循环，cur指向尾结点，但尾结点的元素未打印，手动打印

        print(cur.elem)

    def add(self, item):

        """链表头部添加元素，头插法"""

        node = Node(item)

        if self.is\_empty():

            self.\_head = node

            node.next = node

        else:

            cur = self.\_head

            #找到尾结点

            while cur.next != self.\_head:

                cur = cur.next

        # 退出循环，cur指向尾结点，但尾结点的元素未打印，手动打印

            node.next = self.\_head

            self.\_head = node

            cur.next = self.\_head

    def append(self, item):

        """链表的尾部添加元素,尾插法"""

        node = Node(item)

        if self.is\_empty():  # 空列表的添加

            self.\_head = node

            node.next = node

        else:

            cur = self.\_head

            # 找到尾结点

            while cur.next != self.\_head:

                cur = cur.next

            node.next = self.\_head #最后添加的节点指向第一个

            cur.next = node#之前的最后一个指向新的最后一个

    def insert(self, pos, item):

        """制定位置添加元素

        :param pos 从0开始，inser(2,100),下标为2的地方添加100"""

        if pos <= 0:  # 认为在链表最头部插入元素

            self.add(item)

        elif pos > (self.length() - 1):  # 认为尾插

            self.append(item)

        else:

            pre = self.\_head  # pre指针指向第一个元素

            count = 0

            while count < (pos - 1):  # 循环结束后，指针在pos前一个位置停下来了

                count += 1

                pre = pre.next

            node = Node(item)

            node.next = pre.next  # 新元素的next指向pre的下一个元素的地址

            pre.next = node  # pre的next区域指向新元素

    def remove(self, item):

        if self.is\_empty():  # 空列表直接返回

            return

        """删除节点,不是删除下标指向的位置"""

        cur = self.\_head

        pre = None

        while cur.next != self.\_head:

            if cur.elem == item:

                # 头结点的删除，先判断此节点是否是头结点

                # 对于头结点来说，删除第一个节点的方法为头结点直接指向第二个节点，要找到尾部，使尾部也指向新的头结点

                if cur == self.\_head:

                # 头结点的情况

                #找尾结点,还要创建游标

                    rear = self.\_head

                    while rear.next != self.\_head:

                        rear = rear.next

                    #退出循环后rear指向尾结点,让头结点指向第二个节点，让rear的next区域指向头结点。

                    self\_head = cur.next

                    rear.next = self.\_head

                    #self.\_head = cur.next

                #中间节点的删除

                else:

                    pre.next = cur.next

                return

            #如果没有找到，游标继续游动

            else:

                pre = cur

                cur = cur.next

        #尾结点的删除，跳出循环，cur指向尾结点

        if cur.item == item:

            if cur == self.\_head:

                self.\_head = None

            else:

                pre.next = cur.next

    def search(self, item):

        """查找节点是否存在"""

        if self.is\_empty():  # 空列表

            return False

        else:

            cur = self.\_\_head

            while cur.next != self.\_head:

                if cur.item == item:

                    return True

                else:

                    cur = cur.next

            # 退出循环，cur指向尾结点，但尾结点的元素未打印，手动打印

            if cur.elem == item:

                return  True

            return False

if \_\_name\_\_ == "\_\_main\_\_":

    lcl = SingleCycleLinkList()

    print(lcl.is\_empty())

    print(lcl.length())

    ll.append(1)

    print(lcl.is\_empty())

    print(lcl.length())

    lcl.append(2)

    lcl.append(3)

    lcl.append(4)

    lcl.append(5)

    lcl.append(6)

    lcl.travel()

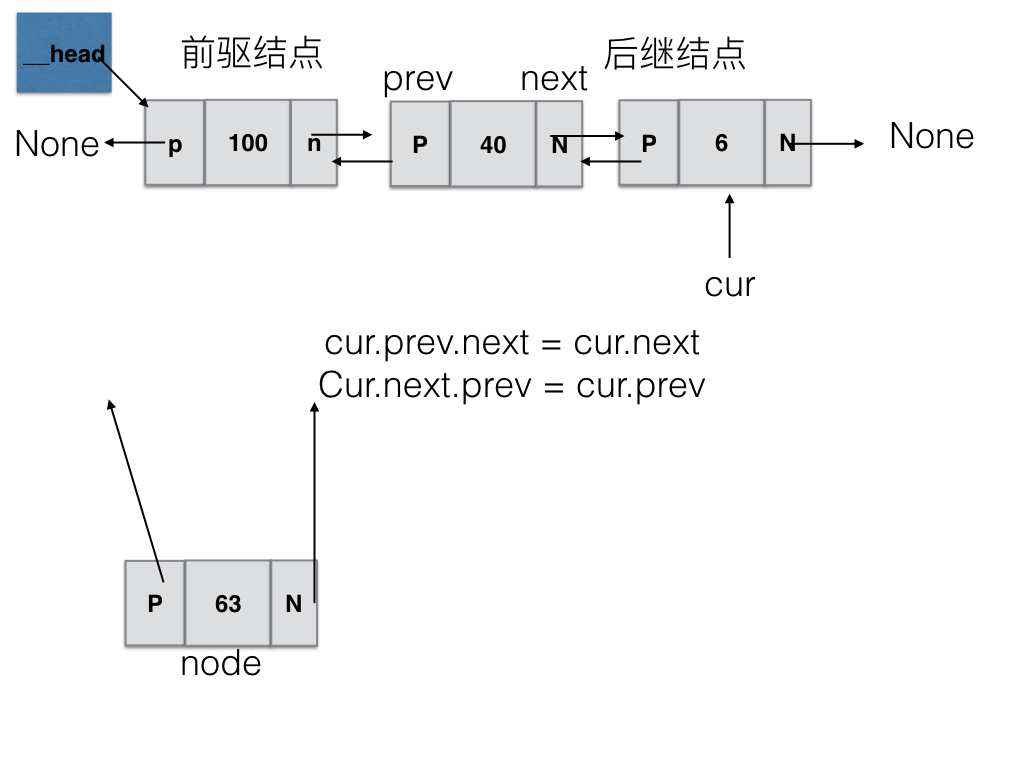
    lcl.remove(4)

    print(lcl.is\_empty())

    lcl.travel()

    print(lcl.length())

**双向链表的操作（3.12）:**



#双向链表，也可以继承SingleLinkList，只需改写add，append，remove，insert

class Node(object):

    """节点"""

    def \_\_init\_\_(self, elem):

        # \_elem 存放数据元素

        self.elem = elem

        # \_next是下一个节点的标识

        self.next = None

        #\_prev是前一个节点

        self.prev = None

class DoubleLinkList(object):

    """双链表"""

    def \_\_init\_\_(self, node=None):

        # 私有属性，记录头结点

        self.\_head = node

    def is\_empty(self):

        """链表是否为空,与双链表相同"""

        return self.\_head == None

    def length(self):

        """链表长度,与双链表相同"""

        # cur游标，用来移动遍历节点

        cur = self.\_head

        # count记录数量

        count = 0

        while cur != None:

            count += 1

            cur = cur.next

        return count

    def travel(self):

        """遍历整个链表,与双链表相同"""

        cur = self.\_head

        while cur != None:

            print(cur.elem,end=" ")

            cur = cur.next

        print()

    def add(self, item):

        """链表头部添加元素，头插法"""

        node = Node(item)

        node.next = self.\_head  # 新节点指向原有的首节点

        self.\_head = node  # 原有的头节点指向新节点

        node.next.prev = node#原有首节点还要反过来指向node

    def append(self, item):

        """链表的尾部添加元素,尾插法"""

        node = Node(item)

        cur = self.\_head

        if self.is\_empty():  # 空列表的添加

            self.\_head = node

        else:

            while cur.next != None:

                cur = cur.next

            cur.next = node

            node.prev = cur

    def insert(self, pos, item):

        """制定位置添加元素

        :param pos 从0开始，inser(2,100),下标为2的地方添加100"""

        if pos <= 0:  # 认为在链表最头部插入元素

            self.add(item)

        elif pos > (self.length() - 1):  # 认为尾插

            self.append(item)

        else:

            cur = self.\_head  #cur指针指向第一个元素

            count = 0

            while count < pos:  # 循环结束后，指针在pos位置停下来了

                count += 1

                cur = cur.next

            node = Node(item)

            node.next = cur  # 新元素的next指向cur元素的地址

            node.prev = cur.prev  # 新元素的前驱为cur元素的前驱

            cur.prev.next = node #cur前面一个元素指向新元素

            cur.prev = node #cur元素的前驱指向新元素

    def remove(self, item):

        """删除节点,不是删除下标指向的位置"""

        cur = self.\_head

        while cur != None:

            if cur.elem == item:

                # 先判断此节点是否是头结点

                # 对于头结点来说，

                if cur == self.\_head:

                    self.\_head = cur.next

                    if cur.next:

                        #判断链表是否只有一个节点，只有一个节点的话，cur.next.pre可能会报错

                        cur.next.prev = None#置空前驱节点

                else:#中间节点

                    cur.prev.next = cur.next#删除cur

                    if cur.next:#考虑尾结点

                        cur.next.prev = cur.prev

                break

            else:

                cur = cur.next

    def search(self, item):

        """查找节点是否存在，与单链表一样"""

        cur = self.\_\_head

        while cur != None:

            if cur.item == item:

                return True

            else:

                cur = cur.next

        return False

if \_\_name\_\_ == "\_\_main\_\_":

    ll = SingleLinkList()

    print(ll.is\_empty())

    print(ll.length())

    ll.append(1)

    print(ll.is\_empty())

    print(ll.length())

    ll.append(2)

    ll.append(3)

    ll.append(4)

    ll.append(5)

    ll.append(6)

    print(ll.is\_empty())

    print(ll.length())

if \_\_name\_\_ == "\_\_main\_\_":

    Dll = DoubleLinkList()

    print(Dll.is\_empty())

    print(Dll.length())

    Dll.append(1)

    print(Dll.is\_empty())

    print(Dll.length())

    Dll.append(2)

    Dll.append(3)

    Dll.append(4)

    Dll.append(5)

    Dll.append(6)

    Dll.travel()

    Dll.add(100)

    Dll.travel()

    Dll.insert(3,6)

    Dll.travel()

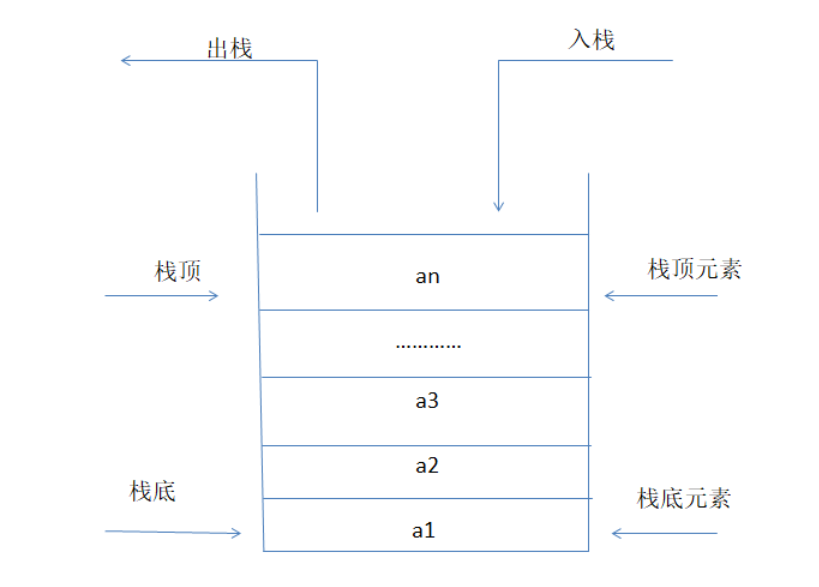
    Dll.remove(5)

    Dll.travel()

    print(Dll.length())

**栈与队列（3.12）:**

  栈，有些地方称为堆栈，是一种容器，可以存入数据元素，访问元素，删除元素，它的特点在于只允许在容器的一端（栈顶，top）进行加入数据（push），和输出数据（pop），没有位置概念，保证任何时候可以访问、删除元素都是此前最后存入的那个元素，确认了一种默认的访问顺序。后进先出。



####栈的实现

#push()压栈,pop()弹栈,peek()返回栈顶元素，不出栈，is\_empty判断栈是否为空,size()栈里面还有多少元素

class Stack(object):

    def \_\_init\_\_(self):

        self.\_list = []#用顺序表实现栈

    def push(self,item):

        #尾部添加元素

        self.\_list.append(item)#尾部作为栈顶，添加元素,时间复杂度比较低相较于头插

    def pop(self):

        #尾部弹出元素

        return self.\_list.pop()

    def peek(self):

        #返回尾部元素

        if self.\_list:

            return self.\_list[-1]

        else:

            return None

    def is\_empty(self):

        #判断是否为空

        return self.\_list == []

        #return not self.\_list

    def size(self):

        #返回栈的长度

         return len(self.\_list)

    if \_\_name\_\_ == "\_\_main\_\_":

         s = Stack()

         s.push(1)

         s.push(2)

         s.push(3)

         s.push(4)

         print(s.pop())

         print(s.pop())

         print(s.pop())

         print(s.peek())

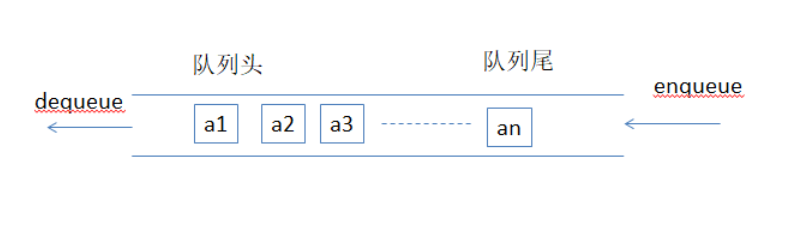
         s.size()

         s.is\_empty()

         print(s)

  队列，是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。

队列是一种先进先出的线性表，允许插入的一端叫队尾，允许删除的一端为队头，队列不允许在中间部位进行操作，假设对列是q = (a1,a2,...,an),那么a1便是队头元素，而an是队尾元素，这样我们删除时，总是从a1开始，插入时，总是在对列最后。



class Queue(object):

    """对列"""

    def \_\_init\_\_(self):

        self.\_list = []

    def enqueue(self,item):

        """往队列尾部添加一个item元素"""

        self.\_list.append(item)

        #self.\_list.insert(0,item)

    def dequeue(self):

        """从对列头部删除一个元素"""

        return self.\_list.pop(0)

        #return self.\_list.pop()根据操作情况选择哪种方式

    def is\_empty(self):

        """判断一个对列是否为空"""

        return not self.\_list

    def size(self):

        """返回对列大小"""

        return len(self.\_list)

if \_\_name\_\_ == "\_\_main\_\_":

    q = Queue()

    q.enqueue(1)

    q.enqueue(2)

    q.enqueue(3)

    q.enqueue(4)

    print(q.dequeue())

    print(q.dequeue())

    print(q.dequeue())

    print(q.dequeue())

class Deque(object):

    """双端对列,头部存，头部取，尾部存，尾部取"""

    def \_\_init\_\_(self):

        self.\_list = []

    def add\_front(self, item):

        """头部存"""

        self.\_list.insert(0,item)

    def add\_rear(self, item):

        """尾部存"""

        self.\_list.append(item)

    def pop\_front(self):

        """列头部删除"""

        return self.\_list.pop(0)

    def pop\_rear(self):

        """尾部删除"""

        return self.\_list.pop()

    def is\_empty(self):

        """判断一个对列是否为空"""

        return not self.\_list

    def size(self):

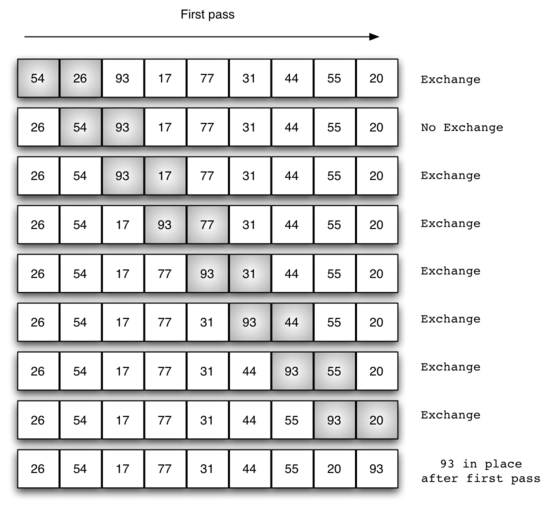
        """返回对列大小"""

        return len(self.\_list)

**冒泡排序（3.13）：**

冒泡排序算法的运作如下（时间复杂度O(N^2)）：

* 比较相邻的元素。如果第一个比第二个大（升序），就交换他们两个。
* 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
* 针对所有的元素重复以上的步骤，除了最后一个。
* 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。



#冒泡排序

def bubble\_sort(alist):

    lenght = len(alist)

    for j in range(1,lenght):#代表班长从头走到尾到底走了多少次，要走n-1次，每走一次找到一个最大值

        count = 0#记录进行了几次交换

        for i in range(lenght-j):#班长从头走到尾，每次排序都会把最大的往后排，所以后面的几个都排好了，就不用了管了，在下标为（0，lenght-2）（0，lenght-3）……遍历

            if alist[i]>alist[i+1]:

                alist[i],alist[i+1] = alist[i+1],alist[i]

                count += 1#如果第一次遍历就没有交换，那么传入的数组本来就是有序的，就不在执行外层循环，减少了时间复杂度

        if count == 0:

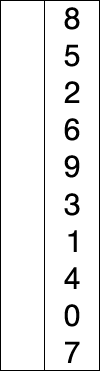
            return tuple(alist)

    return tuple(alist)

**选择排序（3.13）：**

  选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。不稳定的排序方法。

红色表示当前最小值，黄色表示已排序序列，蓝色表示当前位置。

****

#选择排序，自己写的

def select\_sort(alist):

    lenght = len(alist)

    for i in range(0,lenght-1):#从位置0开始排序，遍历过的是排过序的，未遍历过的需要再从里面找最小的

        for j in range(i+1,lenght):#第i个元素依次和后面的进行比较，因为i前面的都是排过序的。

            if alist[j] < alist[i]:

                alist[i],alist[j] = alist[j],alist[i]#alist[i]始终记录的是未排序里面的最小的，交换的次数较多

return tuple(alist)

#标准的

def selection\_sort(alist):

    n = len(alist)

    # 需要进行n-1次选择操作

    for i in range(n-1):

        # 记录最小位置

        min\_index = i

        # 从i+1位置到末尾选择出最小数据

        for j in range(i+1, n):

            if alist[j] < alist[min\_index]:

                min\_index = j

        # 如果选择出的数据不在正确位置，进行交换

        if min\_index != i:

            alist[i], alist[min\_index] = alist[min\_index], alist[i]

alist = [54,226,93,17,77,31,44,55,20]

selection\_sort(alist)

print(alist)

**插入排序（3.13）：**

  插入排序（英语：Insertion Sort）是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中**从后向前**扫描，找到相应位置并插入。插入排序在实现上，在从后向前扫描过程中，需要反复把**已排序元素逐步向后挪位**，即如果前面一个元素比待插入的元素大，就要交换二者的位置。

  与选择排序的区别，选择排序每次找未排序序列的最小值放到已排序序列的末尾。不用考虑插入顺序问题。而插入排序减少了去未排序序列找最小值的过程，而是直接拿未排序序列的第一个插入到已排序的序列中，这样就新增了一个如何找到其在已排序序列中位置的问题。算法复杂度一个发生在后，一个发生在前。

**最优时间复杂度为o(n)，对于排好序的while循环直接break,最坏时间复杂度o(n^2)。稳定的，就是说相等元素不改变其原先的前后位置。**

#插入排序

def insert\_sort(alist):

    #选择第几个元素开始往前放

    n = len(alist)

    for j in range(1,n):

        #j=[1,2,3,4....,n-1]

        #i 代表的是内层循环的开始

        i = j

        #执行的是从无序序列中取出第一个元素，即i位置的元素，去找到其在有序序列中的位置

        while i>0:#从后面往前面放的过程

            if alist[i]<alist[i-1]:

                #交换已排序的序列

                alist[i],alist[i-1] = alist[i-1],alist[i]

                i -= 1

            else:

                break

    return alist

alist = [1,4,5,9,-8,0]

print(insert\_sort(alist))

def insert\_sort(alist):

    # 从第二个位置，即下标为1的元素开始向前插入

    for i in range(1, len(alist)):

        # 从第i个元素开始向前比较，如果小于前一个元素，交换位置

        for j in range(i, 0, -1):

            if alist[j] < alist[j-1]:

                alist[j], alist[j-1] = alist[j-1], alist[j]

alist = [54,26,93,17,77,31,44,55,20]

insert\_sort(alist)

print(alist)