

2.3 线性表的链式存储结构

2.3.1 线性表的链式存储结构—链表

线性表：

$(a_0, a_1, \dots, a_i, a_{i+1}, \dots, a_{n-1})$

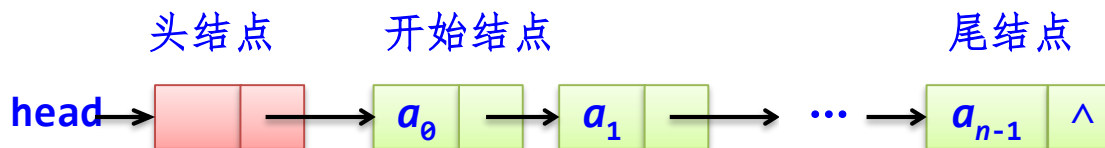


映射

结点

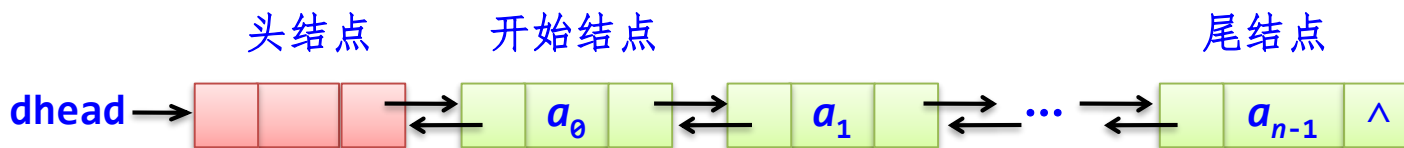
结点包含有元素值和前后继结点的地址信息

- 如果每个结点只设置一个指向其后继结点的指针成员，这样的链表称为线性单向链接表，简称**单链表**。



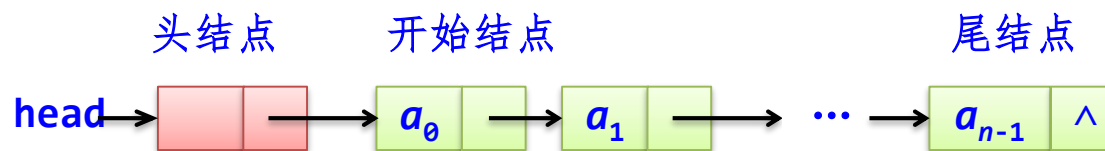
(a) 单链表

- 如果每个结点中设置两个指针成员，分别用以指向其前驱结点和后继结点，这样的链表称之为线性双向链接表，简称**双链表**。



(b) 双链表

2.3.2 单链表



每个结点为**LinkNode**类对象，包括存储元素的数据列表**data**和存储后继结点的指针属性**next**。



```
class LinkNode:                                #单链表结点类
    def __init__(self,data=None):                #构造方法
        self.data=data                          #data属性
        self.next=None                          #next属性
```

单链表类LinkedList

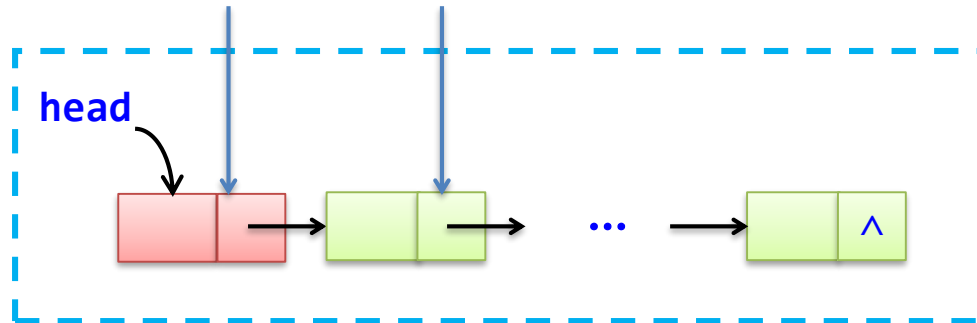
```
class LinkedList:                                #单链表类
    def __init__(self):                          #构造方法
        self.head=LinkNode()                   #头结点head
        self.head.next=None
#基本运算算法
```



结点引用方式:

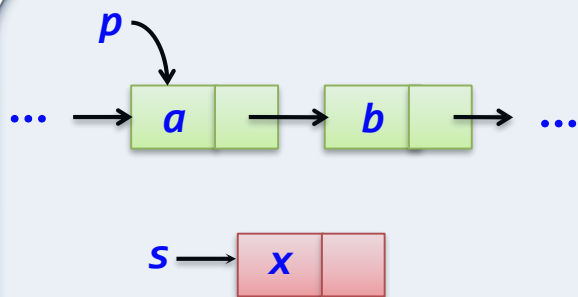
L.head L.head.next

单链表对象L:

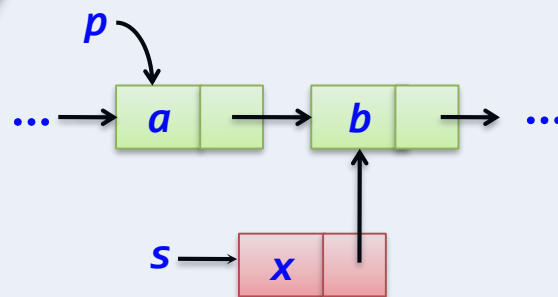


1. 插入和删除结点操作

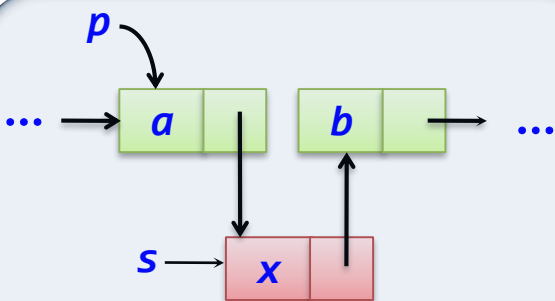
插入结点操作：将结点 s 插入到单链表中 p 结点的后面。



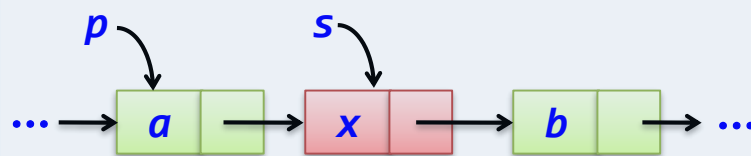
(a) 插入前



(b) $s.next = p.next$

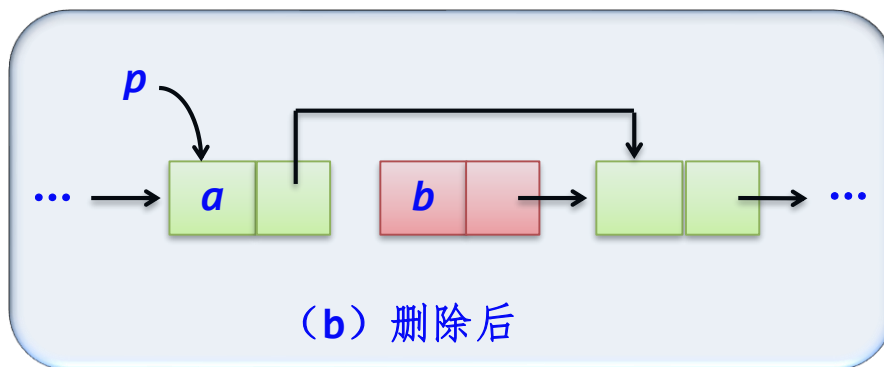
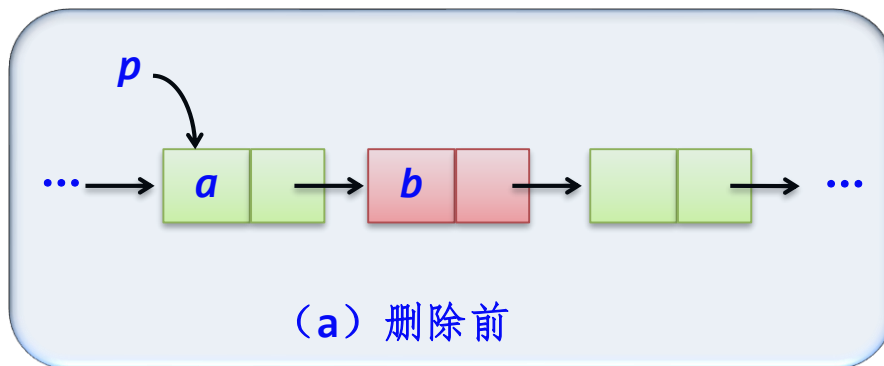


(c) $p.next = s$



(d) 插入后

删除结点操作：删除单链表中 p 结点的后继结点。

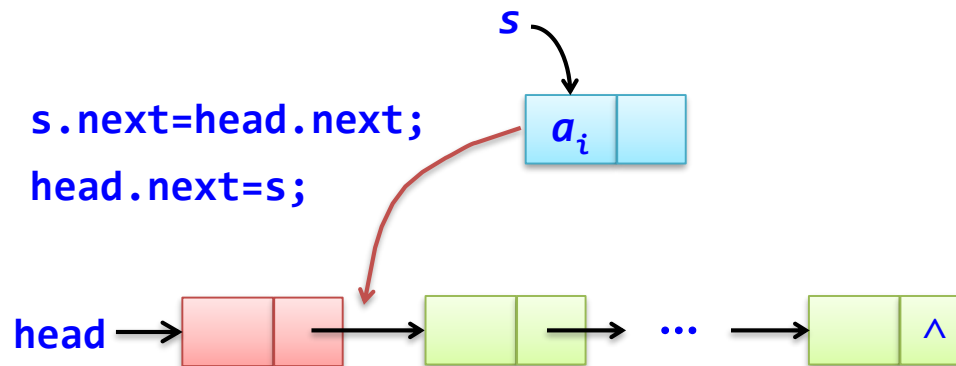


2. 整体建立单链表

- 通过一个含有 n 个元素的 a 数组来建立单链表。
- 建立单链表的常用方法有两种：头插法和尾插法。

头插法建表

- 从一个空表开始，依次读取数组 a 中的元素。
- 生成新结点 s ，将读取的数据存放到新结点的数据成员中。
- 将新结点 s 插入到当前链表的表头上。

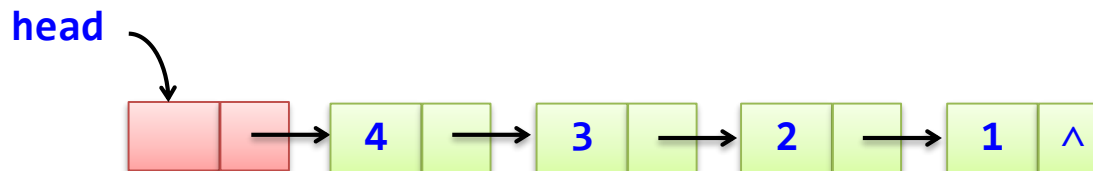


```
def CreateListF(self, a):  
    for i in range(0, len(a)):  
        s = LinkNode(a[i])  
        s.next = self.head.next  
        self.head.next = s
```

#头插法：由数组a整体建立单链表
#循环建立数据结点s
#新建存放a[i]元素的结点s
#将s结点插入到开始结点之前,头结点之后



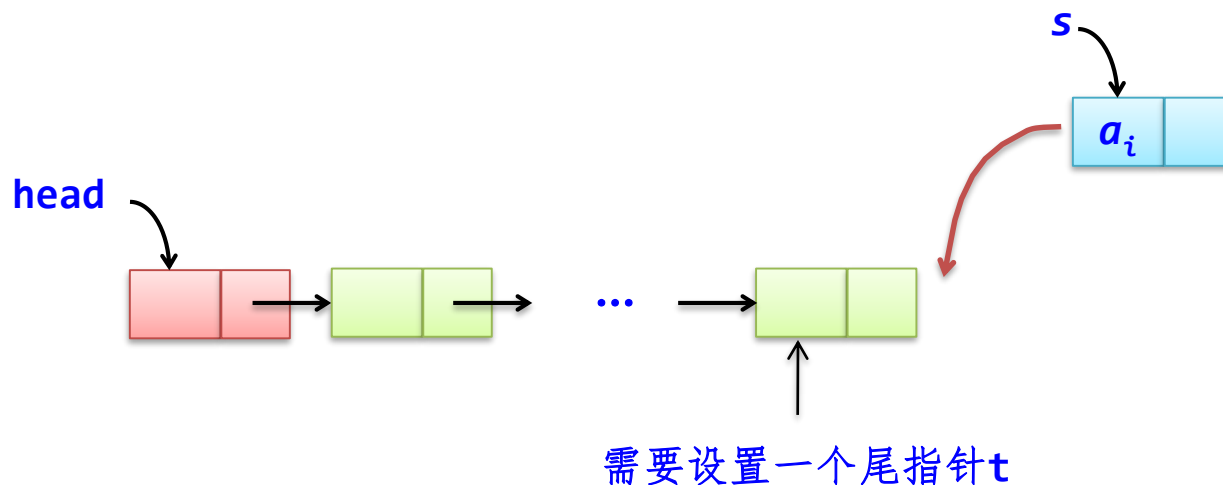
$a=[1,2,3,4]$, 调用CreateListF(a)



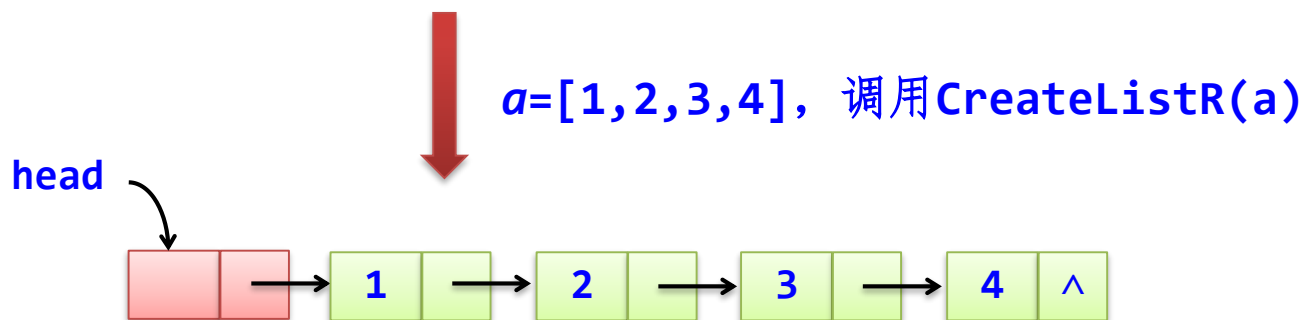
头插法建立的单链表中数据结点的次序与 a 数组中的次序正好相反

尾插法建表

- 从一个空表开始，依次读取数组 a 中的元素。
- 生成新结点 s ，将读取的数据存放到新结点的数据成员中。
- 将新结点 s 插入到当前链表的表尾上。



def CreateListR(self, a):	#尾插法：由数组a整体建立单链表
t=self.head	#t始终指向尾结点,开始时指向头结点
for i in range(0,len(a)):	#循环建立数据结点s
s=LinkNode(a[i])	#新建存放a[i]元素的结点s
t.next=s	#将s结点插入t结点之后
t=s	
t.next=None	#将尾结点的next成员置为空

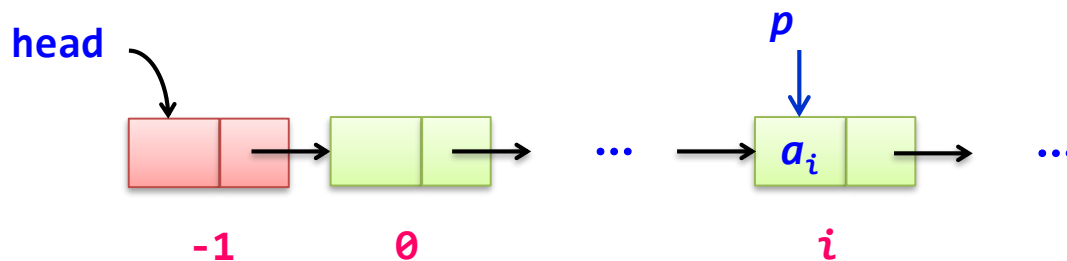


尾插法建立的单链表中数据结点的次序与 a 数组中的次序正好相同

3. 线性表基本运算在单链表中的实现

查找序号为 i ($0 \leq i \leq n-1$, n 为单链表中数据结点个数) 的结点

```
def geti(self, i):                #返回序号为i的结点
    p=self.head
    j=-1
    while (j<i and p is not None):
        j+=1
        p=p.next
    return p
```



(1) 将元素 e 添加到线性表末尾Add(e)

```
def Add(self, e):  
    p=self.head  
    s=LinkNode(e)  
    while p.next is not None:  
        p=p.next  
    p.next=s;
```

#在线性表的末尾添加一个元素 e
#新建结点 s
#查找尾结点 p
#在尾结点之后插入结点 s



(2) 求线性表的长度getsize()

```
def getsize(self):                                #返回长度
    p=self.head
    cnt=0
    while p.next is not None:                      #找到尾结点为止
        cnt+=1
        p=p.next
    return cnt
```

提示

若像顺序表中一样，在单链表中设置一个长度size，插入时size+=1，删除时size-=1。那么求长度的时间复杂度为 $O(1)$ 了。

(3) 求线性表中序号为*i*的元素GetElem(*i*)

<pre>def __getitem__(self,i):</pre>	#求序号为 <i>i</i> 的元素
<pre> assert i>=0</pre>	#检测参数 <i>i</i> 正确性的断言
<pre> p=self.geti(i)</pre>	
<pre> assert p is not None</pre>	# <i>p</i> 不为空的检测
<pre> return p.data</pre>	



调用方式: $x=L[i]$

(4) 设置线性表中序号为*i*的元素SetElem(*i*, *e*)

<code>def __setitem__(self, i, e):</code>	<code>#设置序号为<i>i</i>的元素</code>
<code> assert i>=0</code>	<code>#检测参数<i>i</i>正确性的断言</code>
<code> p=self.geti(i)</code>	
<code> assert p is not None</code>	<code>#<i>p</i>不为空的检测</code>
<code> p.data=e</code>	

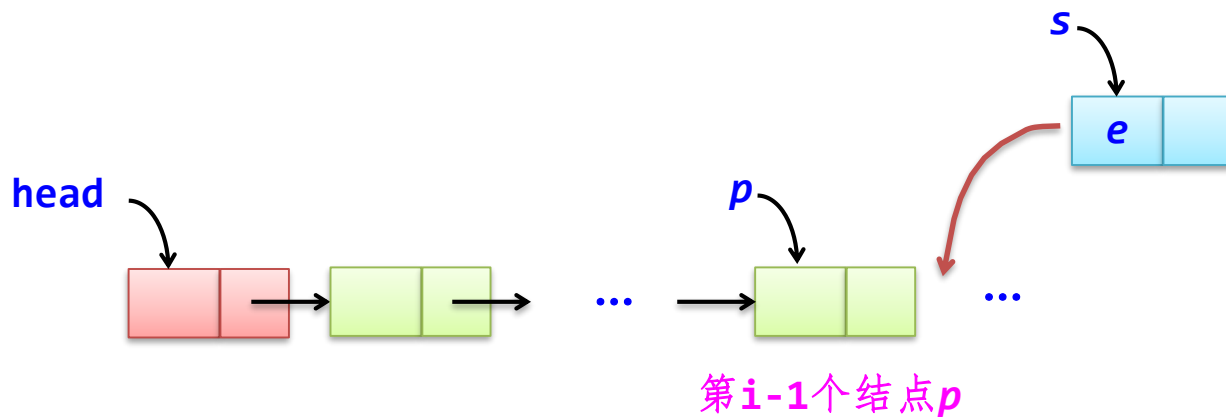


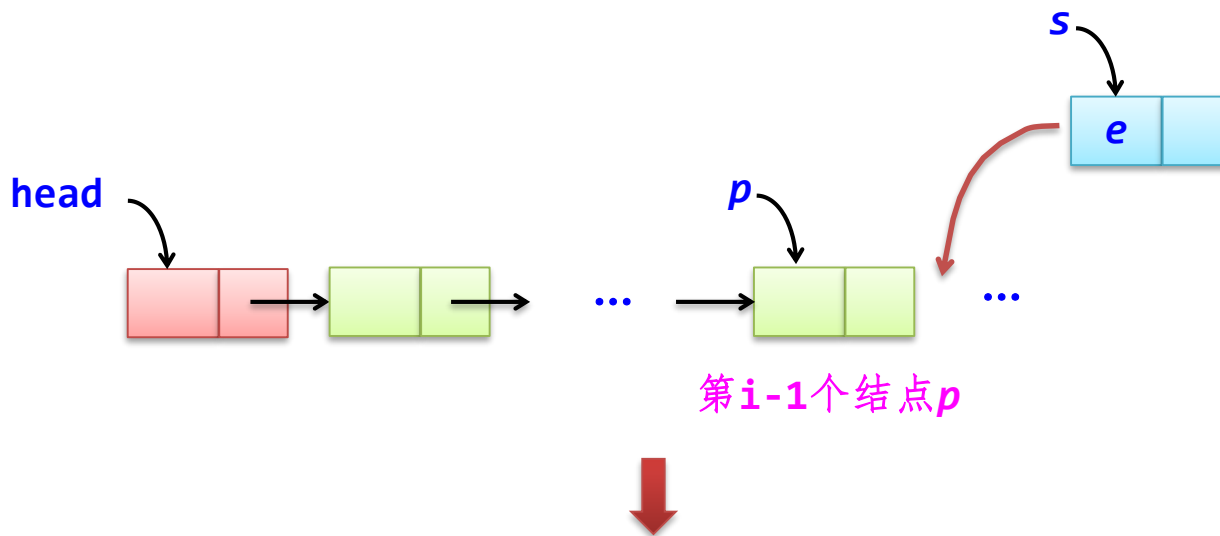
调用方式: `L[i]=e`

(5) 求线性表中第一个值为 e 的元素的逻辑序号GetNo(e)

```
def GetNo(self,e):                                #查找第一个为e的元素的序号
    j=0
    p=self.head.next
    while p is not None and p.data!=e:
        j+=1                                       #查找元素e
        p=p.next
    if p is None:
        return -1                                #未找到时返回-1
    else:
        return j                                  #找到后返回其序号
```

(6) 在线性表中插入 e 作为第 i 个元素 $\text{Insert}(i, e)$





```
def Insert(self, i, e):
```

```
    assert i >= 0
```

```
    s = LinkNode(e)
```

```
    p = self.geti(i-1)
```

```
    assert p is not None
```

```
    s.next = p.next
```

```
    p.next = s
```

#在线性表中序号 i 位置插入元素 e

#检测参数 i 正确性的断言

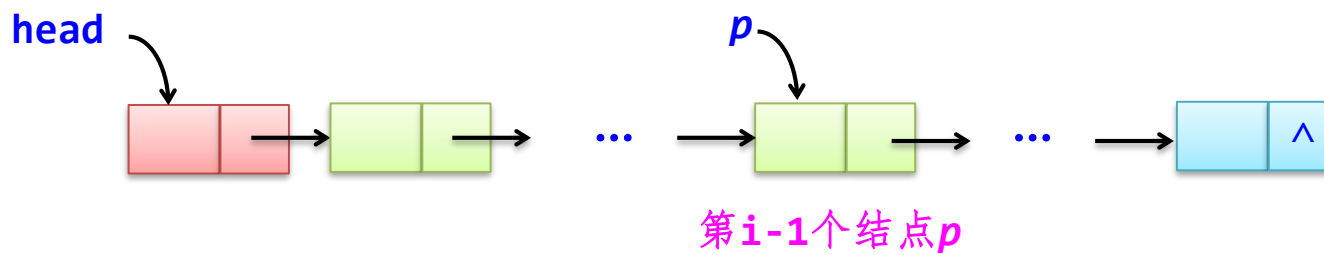
#建立新结点 s

#找到序号为 $i-1$ 的结点 p

p 不为空的检测

#在 p 结点后面插入 s 结点

(7) 在线性表中删除第 i 个数据元素Delete(i)



删除第*i*个结点算法



def Delete(self,i):	#在线性表中删除序号 <i>i</i> 位置的元素
assert i>=0	#检测参数 <i>i</i> 正确性的断言
p=self.geti(i-1)	#找到序号为 <i>i-1</i> 的结点 <i>p</i>
assert p!=None and p.next!=None	# <i>p</i> 和 <i>p.next</i> 不为空的检测
p.next=p.next.next	#删除 <i>p</i> 结点的后继结点

(8) 输出线性表所有元素display()

```
def display(self):                                #输出线性表
    p=self.head.next
    while p is not None:
        print(p.data,end=' ')
        p=p.next
    print()
```

2.3.3 单链表的应用算法设计示例

1. 基于单链表基本操作的算法设计

【例2.6】有一个长度大于2的整数单链表L，设计一个算法查找L中中间位置的元素。

例如， $L = (1, 2, 3)$ ，返回元素为2； $L = (1, 2, 3, 4)$ ，返回元素为2。

计数法：计算出L的长度 n ，假设首结点的编号为1，则满足题目要求的结点的编号为 $(n-1)/2+1$ （这里的除法为整除）。置 $j=1$ ，指针 p 指向首结点，让其后续移 $(n-1)/2$ 个结点即可。

```
def Middle1(L):                                #求解算法1
    j=1
    n=L.getsize();
    p=L.head.next;                             #p指向首结点
    while j<=(n-1)//2:                         #找中间位置的p结点
        j+=1
        p=p.next;
    return p.data
```

快慢指针法：设置快慢指针**fast**和**slow**，首先均指向首结点，当**fast**结点后面至少存在两个结点时，让慢指针**slow**每次后移动一个结点，让快指针**fast**每次后移动两个结点。否则**slow**指向的结点就是满足题目要求的结点。

```
def Middle2(L):                                #求解算法2
    slow=L.head.next
    fast=L.head.next                            #均指向首结点
    while fast!=None and fast.next!=None and fast.next.next!=None:
        slow=slow.next                        #慢指针每次后移1个结点
        fast=fast.next.next                  #快指针每次后移2个结点
    return slow.data
```

【例2.7】有一个整数单链表L，其中可能存在多个值相同的结点，设计一个算法查找L中最大值结点的个数。

解：先让 p 指向首结点，用 $maxe$ 记录首结点值，将其看成是最大值结点， cnt 置为1。按以下方式循环直到 p 指向尾结点为止：

(1) 若 $p.next.data > maxe$ ，将 $p.next$ 看成新的最大值结点，置 $maxe = p.next.data$ ， $cnt = 1$ 。

(2) 若 $p.next.data == maxe$ ， $maxe$ 仍为最大结点值，置 $cnt++$ 。

(3) p 后移一个结点。

最后返回 cnt 即为最大值结点个数。

```
def Maxcount(L):
```

```
    cnt=1
```

```
    p=L.head.next;
```

```
    maxe=p.data
```

```
    while p.next!=None:
```

```
        if p.next.data>maxe:
```

```
            maxe=p.next.data
```

```
            cnt=1
```

```
        elif p.next.data==maxe:
```

```
            cnt+=1
```

```
        p=p.next
```

```
    return cnt
```

#求解算法

#p指向首结点

#maxe置为首结点值

#循环到p结点为尾结点

#找到更大的结点

#p结点为当前最大值结点

【例2.8】有一个整数单链表L，其中可能存在多个值相同的结点，设计一个算法删除L中所有最大值的结点。

解：过程如下：

- 先遍历L的所有结点，求出最大结点值maxe。
- 再遍历一次删除所有值为maxe的结点，在删除中，通过（pre, p）一对指针指向相邻的两个结点，若p.data==maxe，再通过pre结点删除p结点。

```

def Delmaxnodes(L):
    p=L.head.next
    maxe=p.data
    while p.next!=None:
        if p.next.data>maxe:
            maxe=p.next.data
        p=p.next

    pre=L.head
    p=pre.next
    while p!=None:
        if p.data==maxe:
            pre.next=p.next
            p=pre.next
        else:
            pre=pre.next
            p=pre.next

```

```

#求解算法
#p指向首结点
#maxe置为首结点值
#查找最大结点值maxe

#pre指向头结点
#p指向pre的后继结点
#p遍历所有结点
#p结点为最大值结点
#删除p结点
#让p指向pre的后继结点

#pre后移一个结点
#让p指向pre的后继结点

```

2. 基于整体建立单链表的算法设计

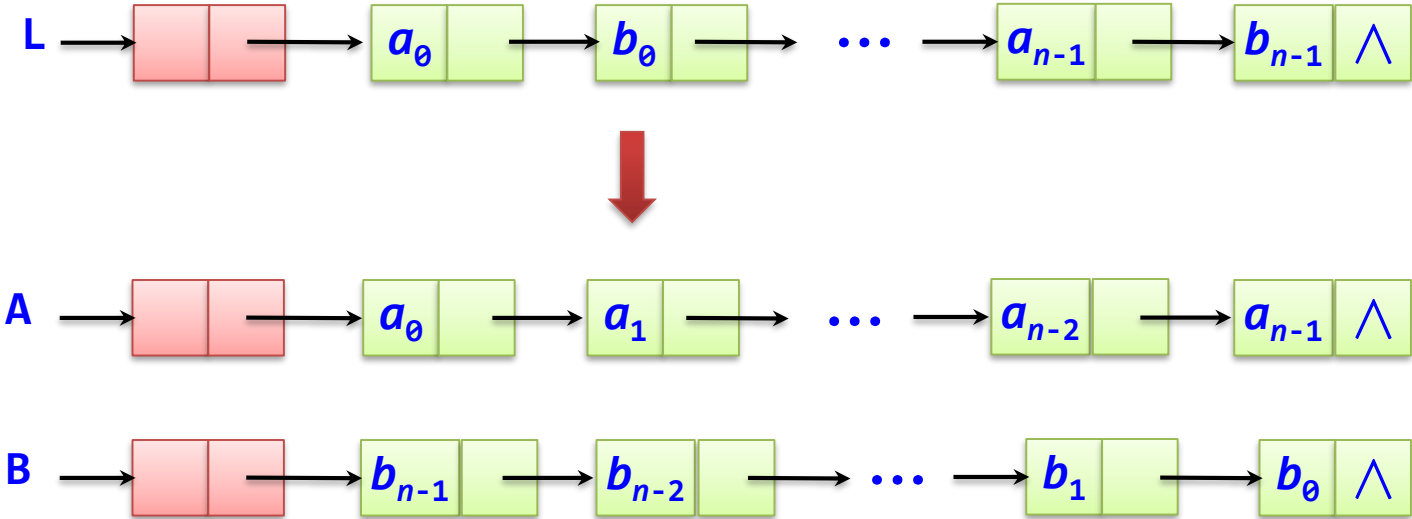
【例2.9】有一个整数单链表L，设计一个算法逆置L中的所有结点。例如L=(1, 2, 3, 4, 5)，逆置后L=(5, 4, 3, 2, 1)。

```
def Reverse(L):  
    p=L.head.next  
    L.head.next=None  
  
    while p!=None:  
        q=p.next  
        p.next=L.head.next  
        L.head.next=p  
        p=q
```

#求解算法
#p指向首结点
#将L置为一个空表
#q临时保存p结点的后继结点
#将p结点插入到表头

【例2.10】 有一个含 $2n$ 个整数的单链表 $L = (a_0, b_0, a_1, b_1, \dots, a_{n-1}, b_{n-1})$ ，设计一个算法将其拆分成两个带头结点的单链表A和B。

其中 $A = (a_0, a_1, \dots, a_{n-1})$ ， $B = (b_{n-1}, b_{n-2}, \dots, b_0)$ 。

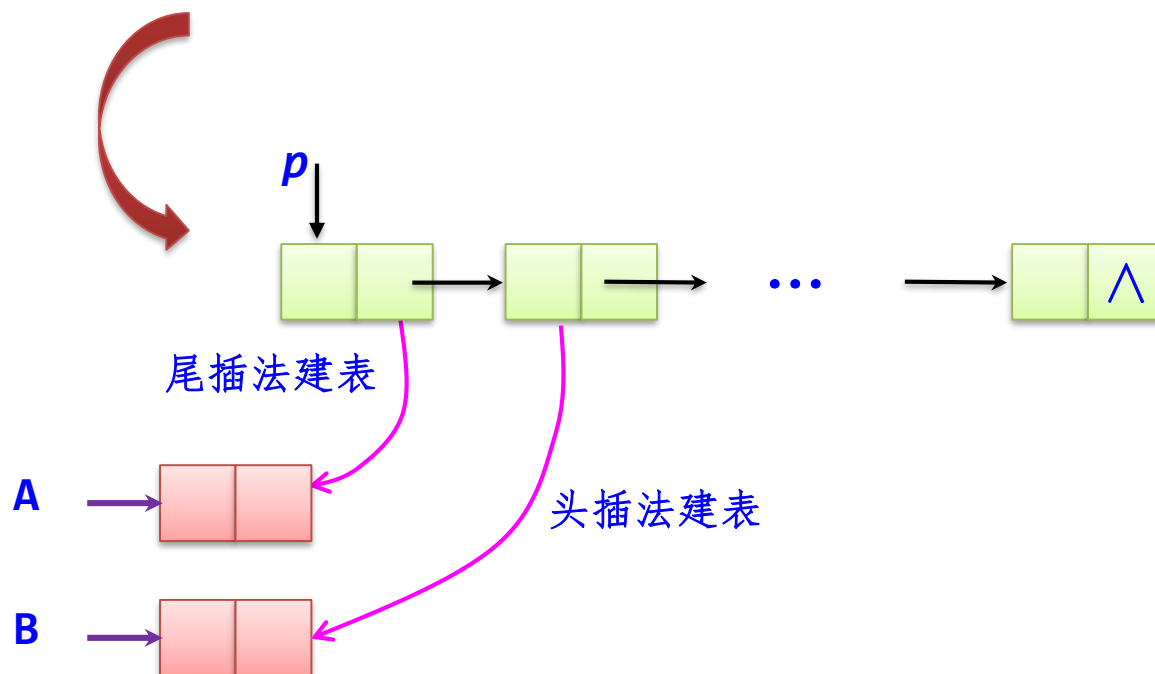


思路

$$L = (a_0, b_0, a_1, b_1, \dots, a_{n-1}, b_{n-1})$$



$$A = (a_0, a_1, \dots, a_{n-1}), \quad B = (b_{n-1}, b_{n-2}, \dots, b_0)$$



```
def Split(L,A,B):
```

```
    p=L.head.next
```

```
    t=A.head
```

```
    while p!=None:
```

```
        t.next=p
```

```
        t=p
```

```
        p=p.next
```

```
    if p!=None:
```

```
        q=p.next;
```

```
        p.next=B.head.next
```

```
        B.head.next=p
```

```
        p=q
```

```
    t.next=None
```

#求解算法

#p指向L的首结点

#t始终指向A的尾结点

#遍历L的所有数据结点

#尾插法建立A

#p后移一个结点

#临时保存p结点的后继结点

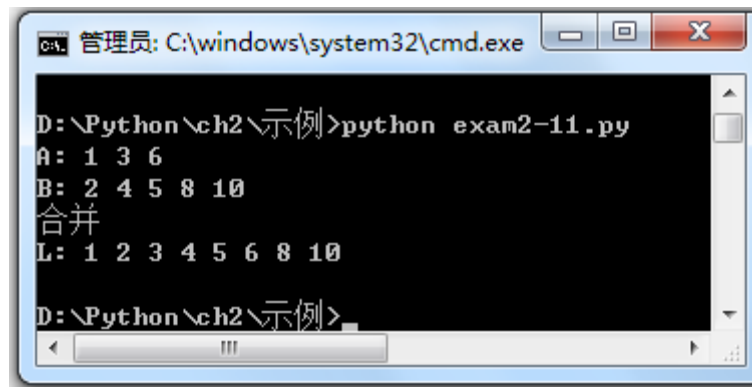
#头插法建立B

#p指向q结点

#尾结点next置空

3. 有序单链表的算法设计

【例2.11】 有两个递增有序整数单链表A和B，设计一个算法采用二路归并方法将A和B的所有数据结点合并到递增有序单链表C中。
要求算法的空间复杂度为 $O(1)$ 。



```
管理员: C:\windows\system32\cmd.exe
D:\Python\ch2\示例>python exam2-11.py
A: 1 3 6
B: 2 4 5 8 10
合并
L: 1 2 3 4 5 6 8 10
D:\Python\ch2\示例>
```

算法的空间复杂度为 $O(1)$?

```
def Merge2(A,B):
```

```
    p=A.head.next
```

```
    q=B.head.next
```

```
    C=LinkList();
```

```
    t=C.head
```

```
    while p!=None and q!=None:
```

```
        if p.data<q.data:
```

```
            t.next=p
```

```
            t=p
```

```
            p=p.next
```

```
        else:
```

```
            t.next=q
```

```
            t=q
```

```
            q=q.next
```

```
    t.next=None
```

```
    if p!=None: t.next=p;
```

```
    if q!=None: t.next=q;
```

```
    return C
```

```
#求解算法
```

```
#p指向A的首结点
```

```
#q指向B的首结点
```

```
#新建单链表C
```

```
#t为C的尾结点
```

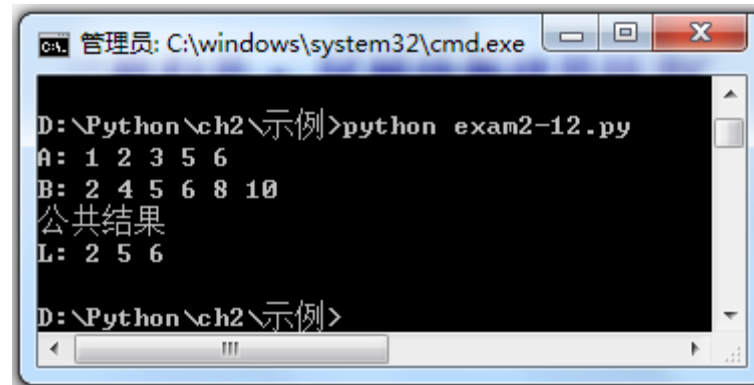
```
#两个单链表都没有遍历完
```

```
#将较小结点p链接到C的末尾
```

```
#将较小结点q链接到C的末尾
```

```
#尾结点next置空
```

【例2.12】有两个递增有序整数单链表A和B，假设每个单链表中没有值相同的结点，但两个单链表中存在相同值的结点，设计一个尽可能高效的算法建立一个新的递增有序整数单链表C，其中包含A和B相同值的结点，要求算法执行后不改变单链表A和B。



```
管理员: C:\windows\system32\cmd.exe
D:\Python\ch2\示例>python exam2-12.py
A: 1 2 3 5 6
B: 2 4 5 6 8 10
公共结果
L: 2 5 6
D:\Python\ch2\示例>
```

A=(1,3,5,7,8)

B=(1,2,5,8,10,11)



二路归并



C=(1,5,8)



二路归并 + 尾插法新建单链表C

```
def Commnodes(A,B):
```

```
    p=A.head.next
```

```
    q=B.head.next
```

```
    C=LinkList();
```

```
    t=C.head
```

```
    while p!=None and q!=None:
```

```
        if p.data<q.data:
```

```
            p=p.next
```

```
        elif q.data<p.data:
```

```
            q=q.next
```

```
        else:
```

```
            s=LinkNode(p.data)
```

```
            t.next=s
```

```
            t=s
```

```
            p=p.next
```

```
            q=q.next
```

```
    t.next=None
```

```
    return C
```

```
#求解算法
```

```
#p指向A的首结点
```

```
#q指向B的首结点
```

```
#新建单链表C
```

```
#t为C的尾结点
```

```
#两个单链表都没有遍历完
```

```
#跳过较小的p结点
```

```
#跳过较小的q结点
```

```
#p结点和q结点值相同
```

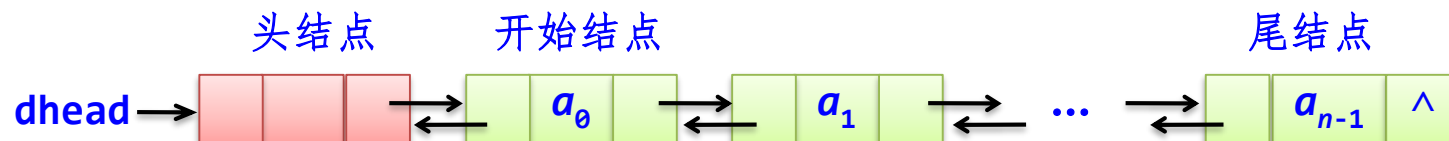
```
#新建s结点
```

```
#将s结点链接到C的末尾
```

```
#尾结点next置空
```

- 本算法的时间复杂度为 $O(n+m)$ 。
- 空间复杂度为 $O(\text{MIN}(n, m))$ 。
- 其中 m 、 n 分别为A、B单链表中的数据结点个数，MIN为取最小值函数，因为单链表C中最多只有 $\text{MIN}(n, m)$ 个结点。

2.3.4 双链表



每个结点为**DLinkNode**类对象，包括存储元素的列表**data**、存储前驱结点指针属性**prior**和后继结点的指针属性**next**。

DLinkNode类



```
class DLinkNode:                                #双链表结点类
    def __init__(self,data=None):                #构造方法
        self.data=data                          #data属性
        self.next=None                          #next属性
        self.prior=None                         #prior属性
```

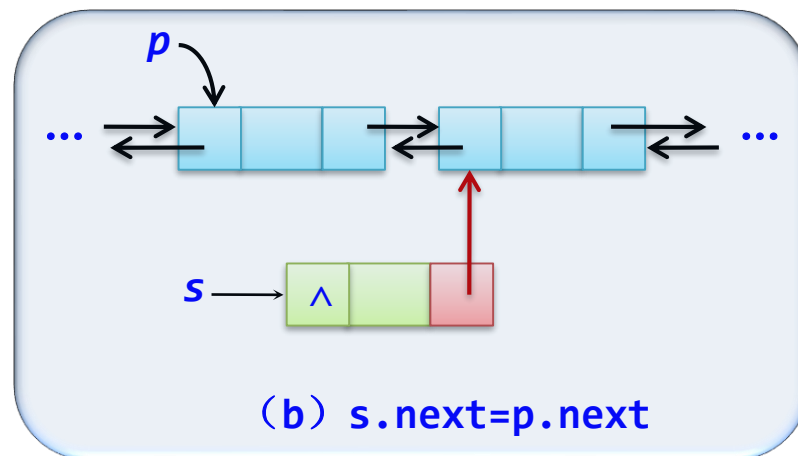
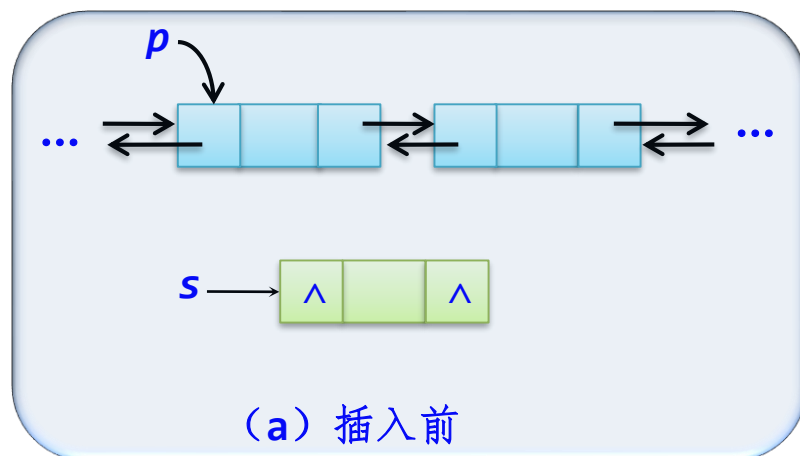
双链表类DLinkedList

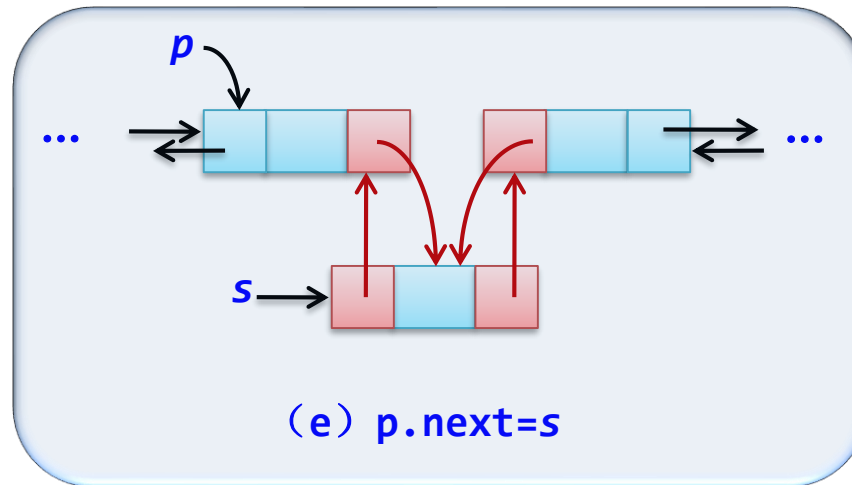
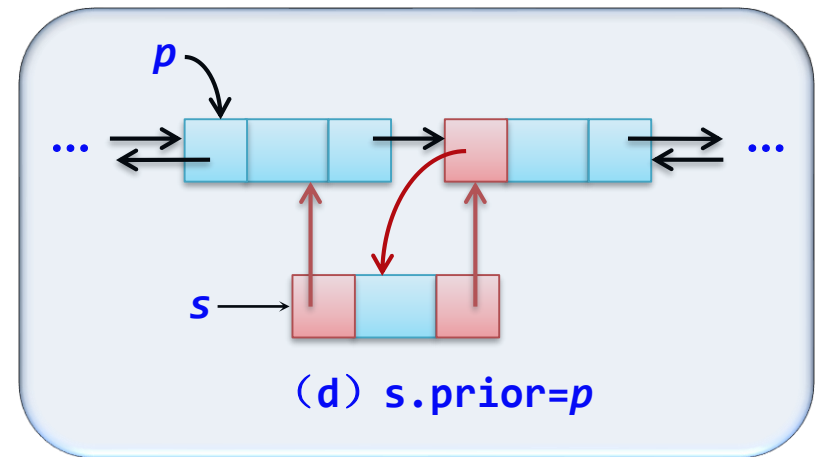
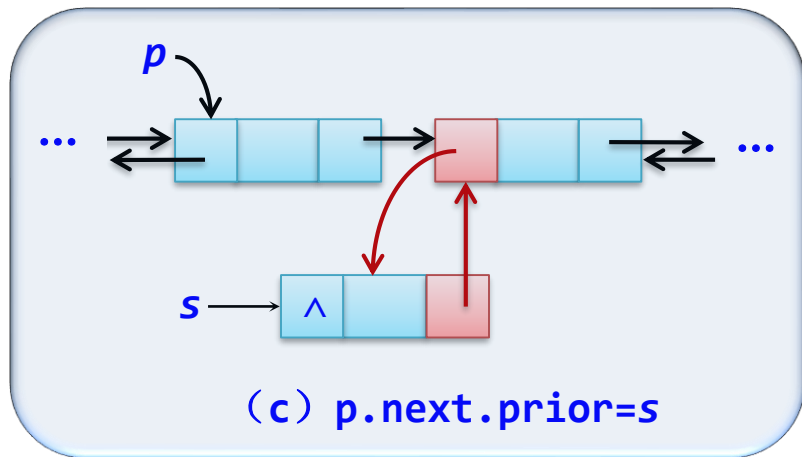
```
class DLinkedList:                                #双链表类
    def __init__(self):                          #构造方法
        self.dhead=DLinkNode()                 #头结点dhead
        self.dhead.next=None
        self.dhead.prior=None
#基本运算算法
```



1. 插入和删除结点操作

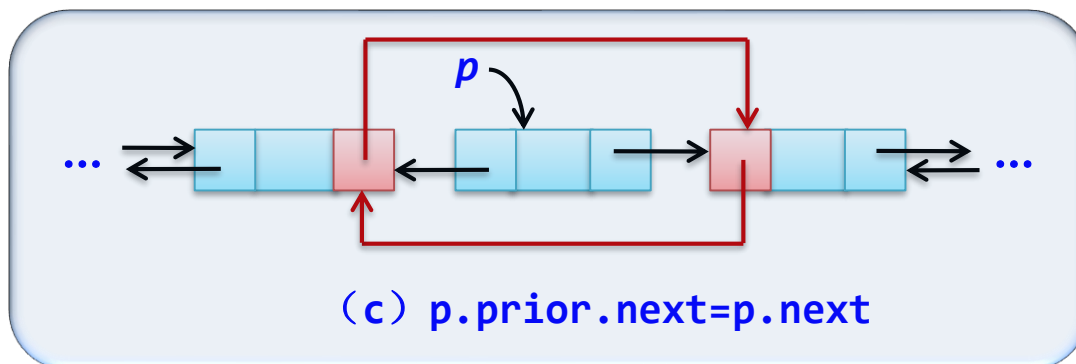
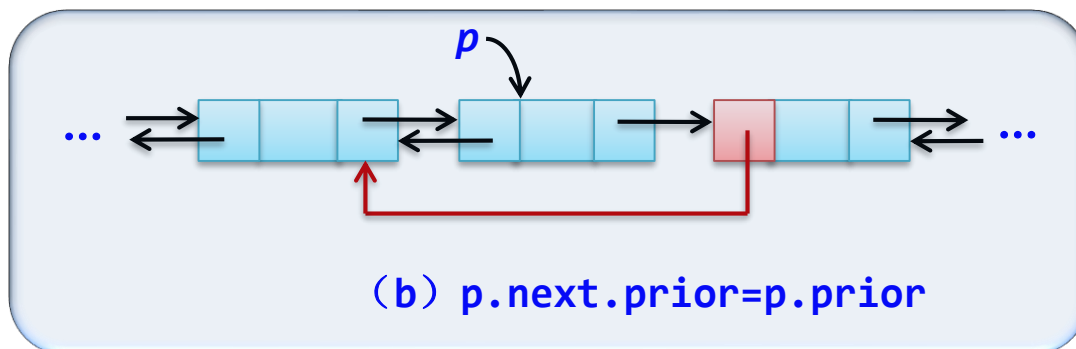
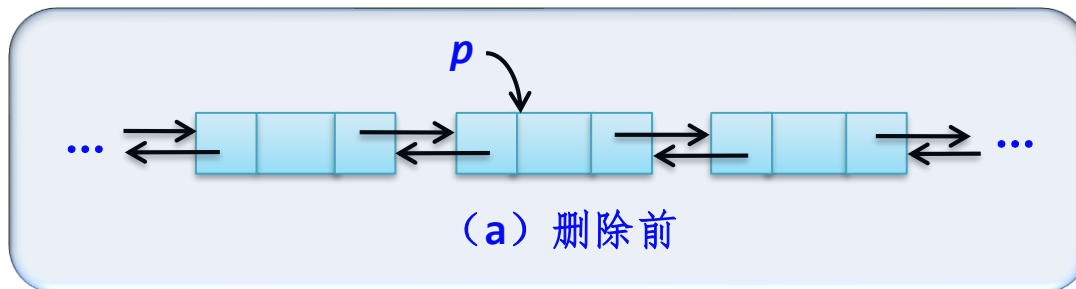
插入结点操作：将结点 s 插入到双链表中 p 结点的后面。





尽可能让间接结点指针修改靠前执行!

删除结点操作：删除双链表中的 p 结点。

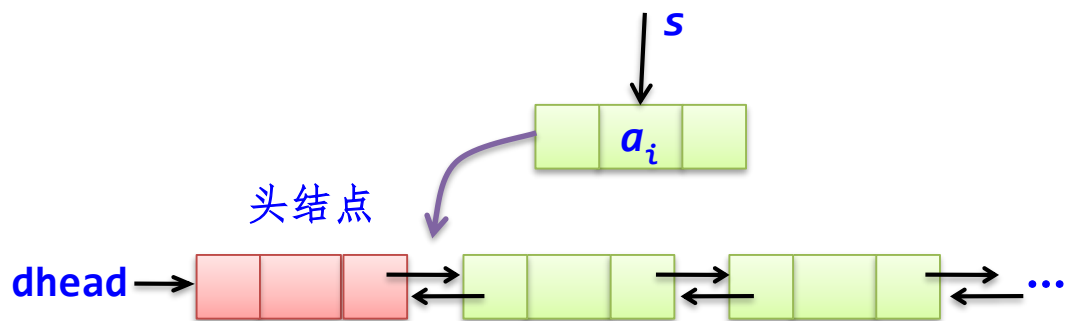


2. 整体建立双链表

- 通过一个含有 n 个元素的 a 数组来建立双链表。
- 建立双链表的常用方法有两种：头插法和尾插法。

头插法建表

<code>def CreateListF(self, a):</code>	<code>#头插法：由数组a整体建立双链表</code>
<code> for i in range(0,len(a)):</code>	<code>#循环建立数据结点s</code>
<code> s=DLinkNode(a[i])</code>	<code>#新建存放a[i]元素的结点s，将其插入到表头</code>
<code> s.next=self.dhead.next</code>	<code>#修改s结点的next成员</code>
<code> if self.dhead.next!=None:</code>	<code>#修改头结点的非空后继结点的prior</code>
<code> self.dhead.next.prior=s</code>	
<code> self.dhead.next=s</code>	<code>#修改头结点的next</code>
<code> s.prior=self.dhead</code>	<code>#修改s结点的prior</code>



尾插法建表

```
def CreateListR(self, a):
```

```
    t=self.dhead
```

```
    for i in range(0,len(a)):
```

```
        s=DLinkNode(a[i])
```

```
        t.next=s
```

```
        s.prior=t
```

```
        t=s
```

```
    t.next=None
```

#尾插法：由数组a整体建立双链表

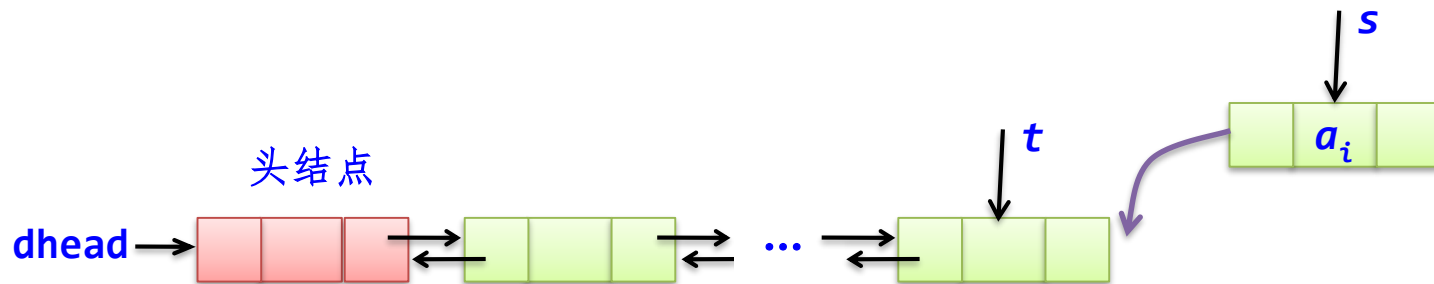
#t始终指向尾结点,开始时指向头结点

#循环建立数据结点s

#新建存放a[i]元素的结点s

#将s结点插入t结点之后

#将尾结点的next成员置为None



3. 线性表基本运算在双链表中的实现

- 许多运算算法（如求长度、取元素值和查找元素等）与单链表中相应算法是相同的。
- 涉及结点插入和删除操作的算法需要改为按双链表的方式进行结点插入和删除。

在双链表dhead中序号为*i*的位置上插入值为*e*的结点的算法

```
def Insert(self, i, e):  
    assert i >= 0  
    s = DLinkNode(e)  
    p = self.geti(i-1)  
    assert p is not None  
    s.next = p.next  
    if p.next != None:  
        p.next.prior = s  
    p.next = s  
    s.prior = p
```

#在线性表中序号*i*位置插入元素*e*
#检测参数*i*正确性的断言
#建立新结点*s*
#找到序号为*i-1*的结点*p*
#*p*不为空的检测
#修改*s*结点的next属性
#修改*p*结点的非空后继结点的prior属性

#修改*p*结点的next属性
#修改*s*结点的prior属性

说明

也可以在双链表中找到序号为*i*的结点*p*（找后继结点），再在*p*结点之前插入*s*结点（后继仅仅之前插入新结点）。

在双链表dhead中删除序号为*i*的结点的算法

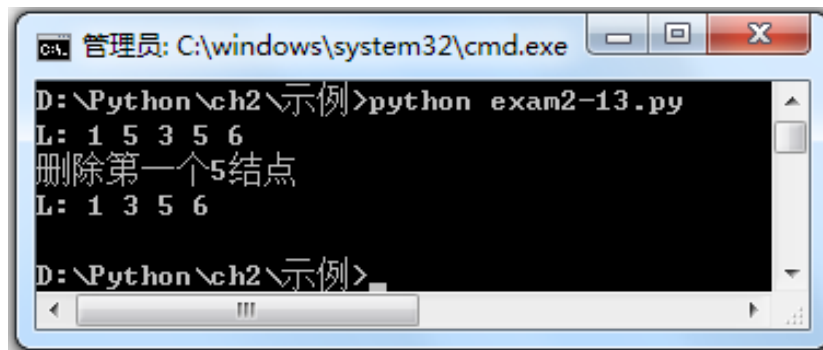
<pre>def Delete(self,i): assert i>=0 p=self.geti(i) assert p is not None p.prior.next=p.next if p.next!=None: p.next.prior=p.prior</pre>	<pre>#在线性表中删除序号i位置的元素 #检测参数i正确性的断言 #找到序号为i的结点p #p不为空的检测 #修改p结点的前驱结点的next #修改p结点非空后继结点的prior</pre>
---	---

说明

也可以找到序号为*i*-1的结点*p*（找前驱结点），再删除其后继结点。

2.3.5 双链表的应用算法设计示例

【例2.13】设计一个算法，删除整数双链表L中第一个值为x的结点，若不存在值为x的结点，则不做任何改变。

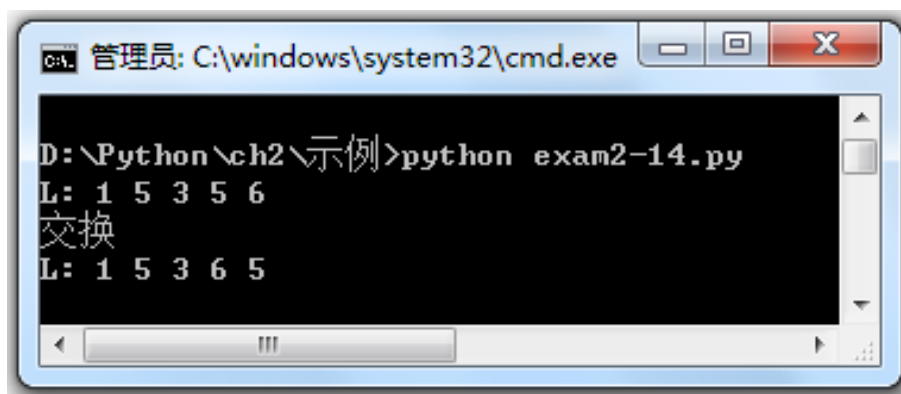


```
管理员: C:\windows\system32\cmd.exe
D:\Python\ch2\示例>python exam2-13.py
L: 1 5 3 5 6
删除第一个5结点
L: 1 3 5 6
D:\Python\ch2\示例>
```

删除整数双链表L中第一个值为x的结点

<code>def Delx(L,x):</code>	<code>#求解算法</code>
<code> p=L.dhead.next</code>	<code>#p指向首结点</code>
<code> while p!=None and p.data!=x:</code>	<code>#查找第一个值为x的结点p</code>
<code> p=p.next</code>	
<code> if p!=None:</code>	<code>#找到值为x的结点p</code>
<code> if p.next!=None:</code>	
<code> p.next.prior=p.prior</code>	<code>#删除p结点</code>
<code> p.prior.next=p.next</code>	

【例2.14】设计一个算法，将整数双链表L中最后一个值为x的结点与其前驱结点交换。若不存在值为x的结点或者该结点是首结点，则不做任何改变。



```
管理员: C:\windows\system32\cmd.exe
D:\Python\ch2\示例>python exam2-14.py
L: 1 5 3 5 6
交换
L: 1 5 3 6 5
```

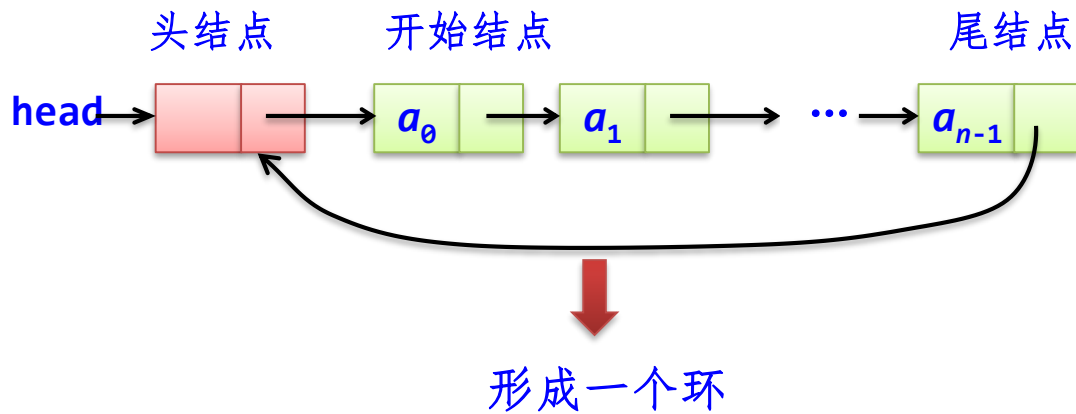
将整数双链表L中最后一个值为x的结点与其前驱结点交换

```
def Swap(L,x):  
    p=L.dhead.next  
    q=None;  
    while p!=None:  
        if p.data==x:q=p  
        p=p.next  
    if q==None or L.dhead.next==q:  
        return  
    else:  
        pre=q.prior  
        pre.next=q.next  
        if q.next!=None:  
            q.next.prior=pre  
        pre.prior.next=q  
        q.prior=pre.prior  
        pre.prior=q;  
        q.next=pre
```

#求解算法
#p指向首结点
#查找最后一个值为x的结点q
#不存在x结点或者该结点是首结点
#直接返回
#找到值为x的结点q
#删除q结点
#将q结点插入到pre结点之前

2.3.6 循环链表

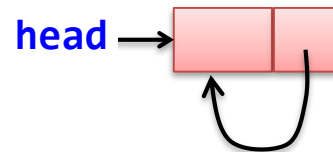
1. 循环单链表



循环单链表类CLinkList

```
class CLinkList:                                #循环单链表类
    def __init__(self):                          #构造方法
        self.head=LinkNode()                   #头结点head
        self.head.next=self.head               #构成循环单链表
#线性表的基本运算算法
```

结点类型与
非循环单链
表中的结点
类型相同



循环单链表的插入和删除结点操作与非循环单链表的相同，所以两者的许多基本运算算法是相似的，主要区别如下：

- 初始只有头结点`head`，在循环单链表的构造方法中需要通过`head.next=head`语句置为空表。
- 循环单链表中涉及查找操作时需要修改表尾判断的条件，例如，用`p`遍历时，尾结点满足的条件是`p.next==head`而不是`p.next==None`。

【例2.15】 有一个整数循环单链表L，设计一个算法求值为x的结点个数。

<code>def Count(L, x):</code>	<code>#求解算法</code>
<code> cnt=0;</code>	<code>#cnt置为0</code>
<code> p=L.head.next</code>	<code>#首先p指向首结点</code>
<code> while p!=L.head:</code>	<code>#遍历循环单链表</code>
<code> if p.data==x:</code>	
<code> cnt+=1</code>	<code>#找到一个值为x的结点cnt增1</code>
<code> p=p.next</code>	<code>#p后移一个结点</code>
<code> return cnt</code>	

【例2.16】编写一个程序求解约瑟夫（Joseph）问题。有 n 个小孩围成一圈，给他们从1开始依次编号，从编号为1的小孩开始报数，数到第 m 个小孩出列，然后从出列的下一个小孩重新开始报数，数到第 m 个小孩又出列，…，如此反复直到所有的小孩全部出列为止，求整个出列序列。

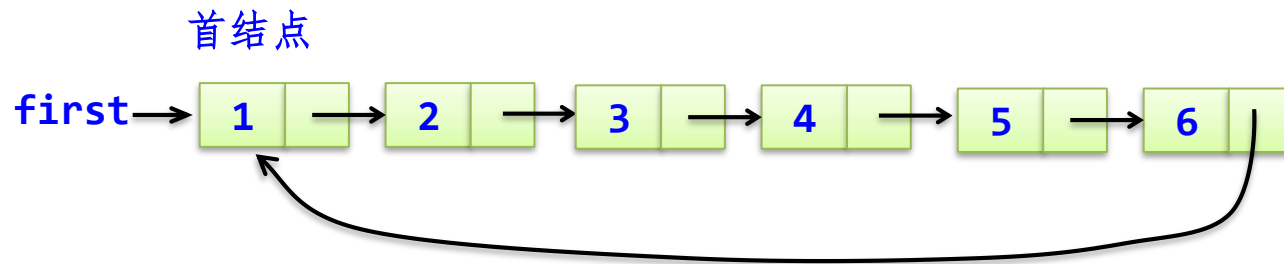
如当 $n=6$ ， $m=5$ 时的出列序列是5，4，6，2，3，1。

解：（1）设计存储结构

本题采用不带头结点的循环单链表存放小孩圈，其结点类如下：

```
class Child:                                #结点类型
    def __init__(self,no1):                 #构造方法
        self.no=no1                        #编号no属性
        self.next=None                     #next属性
```

例如， $n=6$ 时的初始循环单链表如下图所示，**first**指向开始报数的小孩结点，初始时指向首结点。



(2) 设计基本运算算法

设计一个求解约瑟夫问题的Joseph类，其中包含：

- n 、 m 整型成员和首结点指针**first**成员。
- 构造方法用于建立有 n 个结点的不带头结点的循环单链表**first**。
- **Jsequence**方法用于产生约瑟夫序列的字符串。

```
class Joseph:
```

```
    def __init__(self, n1, m1):
```

```
        self.n=n1
```

```
        self.m=m1
```

```
        self.first=Child(1);
```

```
        t=self.first
```

```
        for i in range(2,self.n+1):
```

```
            p=Child(i)
```

```
            t.next=p
```

```
            t=p
```

```
        t.next=self.first
```

#求解约瑟夫问题类

#构造方法

#循环单链表首结点

#建立一个编号为*i*的新结点*p*

#将*p*结点链到末尾

#构成一个首结点为*first*的循环单链表

def Jsequence(self):	#求约瑟夫序列
for i in range(1,self.n+1):	#共出列n个小孩
p=self.first	#每次都从 first 开始
j=1	
while j<self.m-1:	#从 first 结点开始报数，报到第 m-1 个结点
j+=1	#报数递增
p=p.next	#移到下一个结点
q=p.next	#q指向第 m 个结点
print(q.no,end=' ')	#该结点的小孩出列
p.next=q.next	#删除q结点
self.first=p.next	#从下一个结点重新开始
print()	

(3) 设计主程序

设计如下主程序求解一个约瑟夫序列。

```
n=6  
m=3  
L=Joseph(n,m)  
print("n=%d, m=%d的约瑟夫序列:" %(n,m))  
L.Jsequence()
```

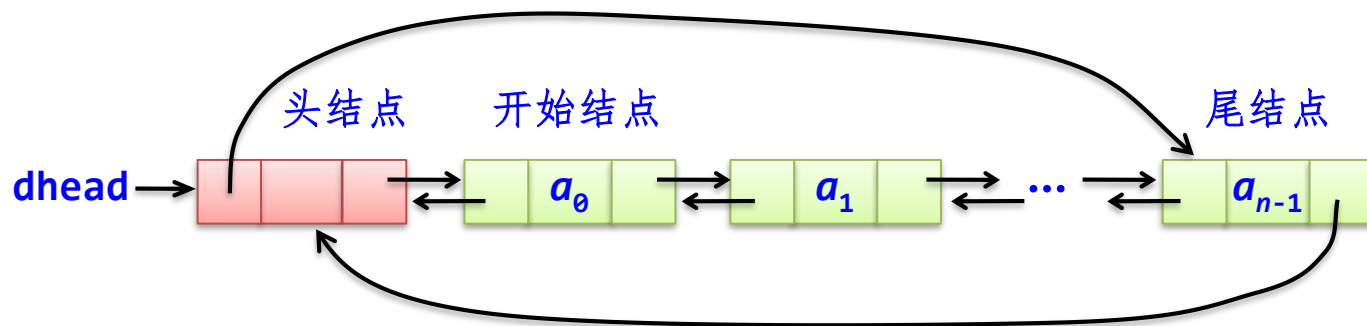
(4) 执行结果

本程序的执行结果如下：

n=6, m=3的约瑟夫序列：

3 6 4 2 5 1

2. 循环双链表



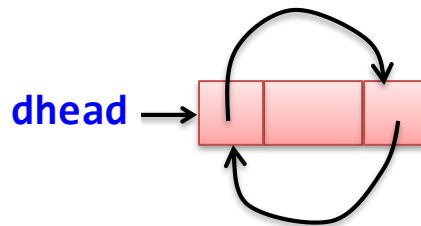
- 形成两个环
- 可以快速找到尾结点

循环双链表类DCLinkList

```
class CDLinkList:                                #循环双链表类
    def __init__(self):                          #构造方法
        self.dhead=DLinkNode()                 #头结点dhead
        self.dhead.next=self.dhead
        self.dhead.prior=self.dhead
```

#线性表的基本运算算法

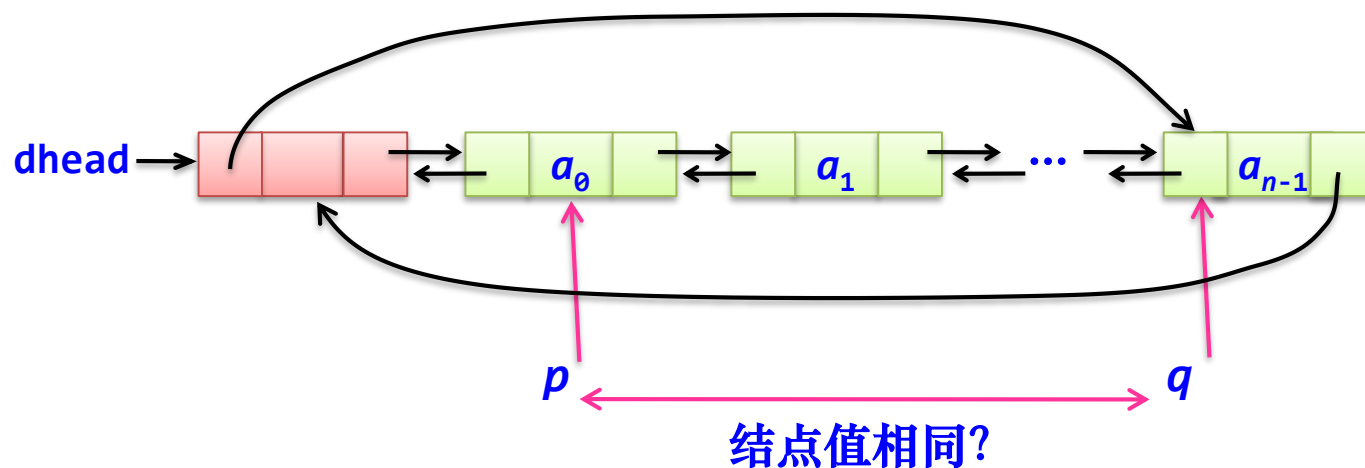
结点类型与
非循环双链
表中的结点
类型相同



循环双链表的插入和删除结点操作与非循环双链表的相同，所以两者的许多基本运算算法是相似的，主要区别如下：

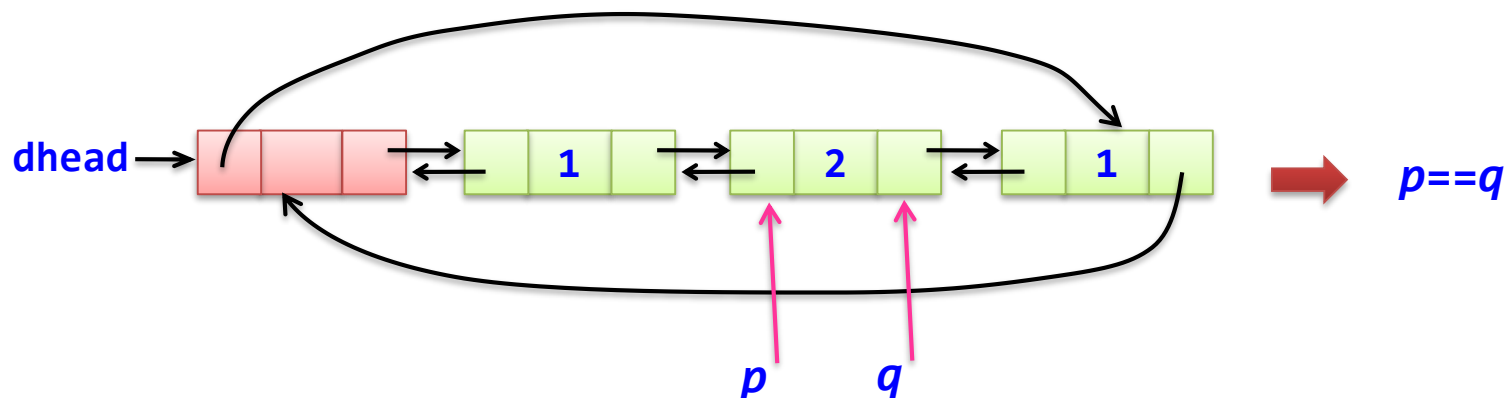
- 初始只有头结点 `dhead`，在循环双链表的构造方法中需要通过 `dhead.prior=dhead` 和 `dhead.next=dhead` 两个语句置为空表。
- 循环双链表中涉及查找操作时需要修改表尾判断的条件，例如，用 `p` 遍历时，尾结点满足的条件是 `p.next==dhead` 而不是 `p.next==None`。

【例2.18】有一个带头结点的循环双链表L，其结点data成员值为整数，设计一个算法，判断其所有元素是否对称。如果从前向后读和从后向前读得到的数据序列相同，表示是对称的；否则不是对称的。

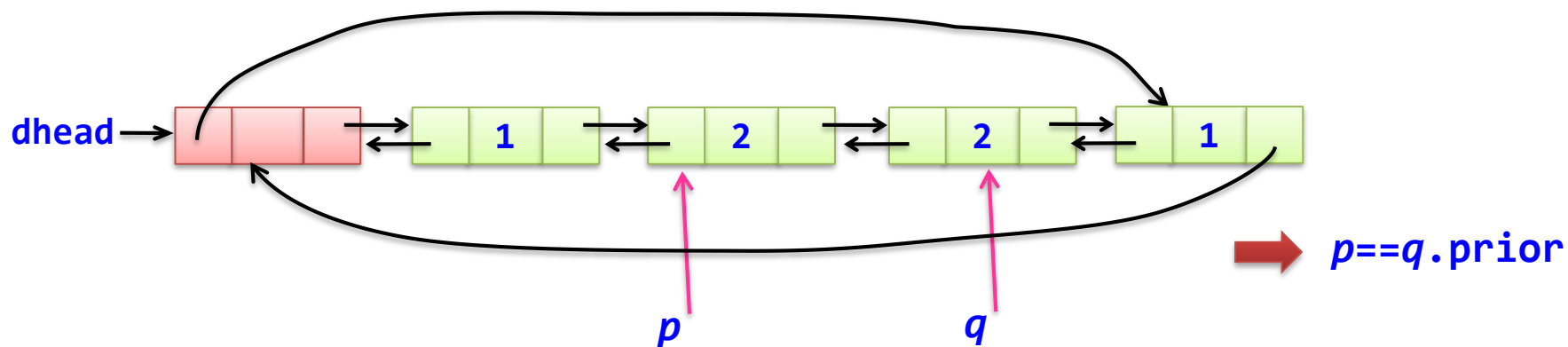


循环结束条件?

(1) 结点个数为奇数



(2) 结点个数为偶数



def Symm(L):	#求解算法
flag=True;	#flag表示L是否对称, 初始时为真
p=L.dhead.next;	#p指向首结点
q=L.dhead.prior;	#q指向尾结点
while flag:	
if p.data!=q.data:	#对应结点值不相同, 置flag为假
flag=False	
else:	
if p==q or p==q.prior: break	
q=q.prior	#q前移一个结点
p=p.next	#p后移一个结点
return flag	

2.4 顺序表和链表的比较

1. 基于空间的考虑

$$\text{存储密度} = \frac{\text{结点中数据本身占用的存储量}}{\text{整个结点占用的存储量}}$$

一般地，存储密度越大，存储空间的利用率就越高。

- 顺序表的存储密度为1，而链表的存储密度小于1。仅从存储密度看，顺序表的存储空间利用率高。
- 顺序表需要预先分配初始空间，所有数据占用一整片地址连续的内存空间，如果分配的空间过小，易出现上溢出，需要扩展空间导致大量元素移动而降低效率；如果分配的空间过大，会导致空间空闲而浪费。而链表的存储空间是动态分配的，只要内存有空闲，就不会出现上溢出。
- **结论：**当线性表的长度变化不大，易于事先确定的情况下，为了节省存储空间，宜采用顺序表作为存储结构。当线性表的长度变化较大，难以估计其存储大小时，为了节省存储空间，宜采用链表作为存储结构。

2. 基于时间的考虑

- 顺序表具有随机存取特性，给定序号查找对应的元素值的时间为 $O(1)$ ，而链表不具有随机存取特性，只能顺序访问，给定序号查找对应的元素值的时间为 $O(n)$ 。
- 在顺序表中插入和删除操作时，通常需要平均移动半个表的元素。而在链表中插入和删除操作仅仅需要修改相关结点的指针成员，不必移动结点。
- **结论：**若线性表的运算主要是查找，很少做插入和删除操作，宜采用顺序表作为存储结构。若频繁地做插入和删除操作，宜采用链表作为存储结构。