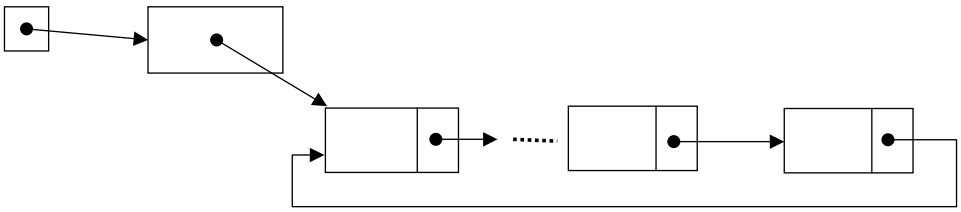
2。线性表(3)

- 计算机内存结构
- 线性表：概念
- **Python list**：线性表的一种实现
- 链接表
- 链接表的变形
- 应用实例

单链表的变形：循环单链表

- 单链表的另一变形是循环单链表，其中最后结点的 **next** 指向首结点



- 仔细分析，不难发现上面的设计在结构上有致命缺点
  - 首端加入删除都不能在 **O(1)** 时间完成，因为要修改尾结点的 **next**
  - 让表对象指向概念上的尾结点更有利（即存储一个尾结点链接），可支持 **O(1)** 的首端加入/删除，以及 **O(1)** 时间的尾端加入
- 操作实现的关键是加入/删除后表对象域的正确更新
  - 把问题想清楚，实现并不困难
  - 非变动操作（如 **find/printall**）也需修改，以单链表为基类没价值

循环单链表

- 定义循环单链表类（不作为 LList 派生类）

```
class LCList:
    def __init__(self):
        self.rear = None # 表对象只有一个 rear 域

    def isEmpty(self):
        return self.rear is None

    def prepend(self, elem):
        p = LNode(elem, None)
        if self.rear is None:
            p.next = p # 表中第一个结点建立初始的循环链接
            self.rear = p
        else:
            p.next = self.rear.next # 链在尾结点之后，就是新的首结点
            self.rear.next = p
```


循环单链表

- 元素加入和删除：

```
    def append(self, elem):
        self.prepend(elem) # 直接调用前段加入操作
        self.rear = self.rear.next # 修改 rear 使之指向新的尾结点

    def pop(self):
        if self.rear is None:
            raise ValueError
        p = self.rear.next
        if self.rear is p: # rear 等于其 next，说明表中只有一个结点
            self.rear = None # 弹出唯一结点后 rear 置空
        else:
            self.rear.next = p.next #一般情况，删去一个结点
        return p.elem
```

循环单链表

- 简单输出所有元素的方法，注意循环结束的条件

```
def printall(self):
    p = self.rear.next
    while True:
        print(p.elem)
        if p is self.rear:
            break
        p = p.next
```

- 下面是一段使用循环表的代码：

```
mlist = LCList()
for i in range(10):
    mlist.prepend(i)
for i in range(11, 20):
    mlist.append(i)
while not mlist.isEmpty():
    print(mlist.pop())
```

有了这些基本操作，再加上对循环表结构的理解，其他操作（如判空，各种遍历操作）不难完成

留作课下的个人自由练习

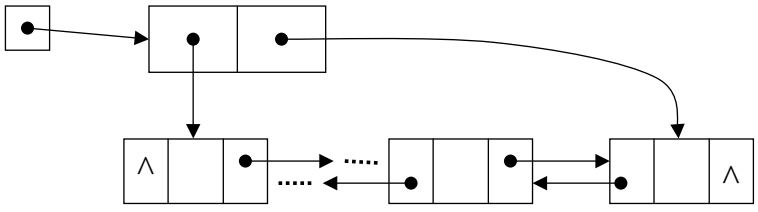
双向链接表（双链表）

- 单链表只能支持首端元素加入/删除和尾端加入的 **O(1)** 实现

  如果希望高效实现两端的加入/删除，就必须修改结点的设计

  结点间双向链接不仅支持两端的高效操作，一般结点操作也更方便

- 支持首尾端高效操作的双向链接表（双链表）结构：



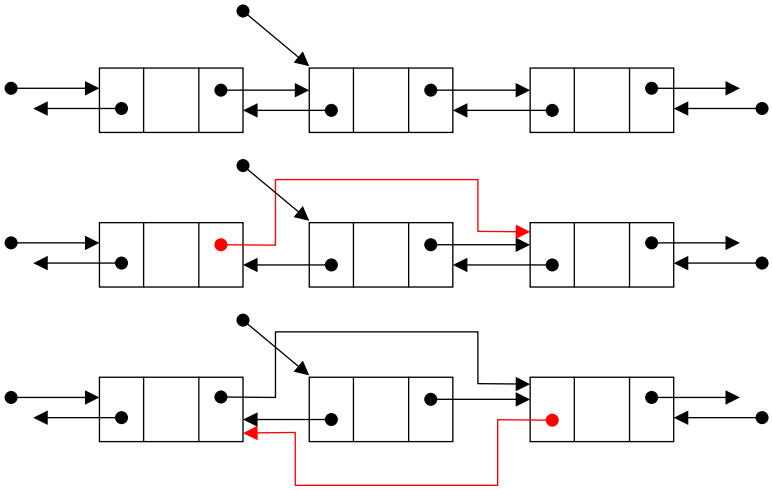
从任一结点出发，可以直接找到前后的相邻结点。而对单链表，找后面一个结点很方便，而找前一结点就必须从表头开始逐一检查

- 每个结点需要增加了一个前向引用域，指向前一结点（存储开销）
- 与前面一样，增加了尾结点引用，才能实现高效的尾端操作

双链表

- 在双链表里可以直接找到前后的结点，因此做许多操作比较方便

  例如，只要掌握着双链表里一个结点，就可以把它从表中取下，同时将表里的其余结点正确链接好



- 在表中任一结点的前面（或后面）加入一个结点的操作也很容易在局部完成。易见，加入一个结点需要做四次指针赋值（自己考虑）

双链表的 Python 实现

- 现在考虑双链表的一个简单实现。首先要定义一个新结点类：

```
class LDNode(LNode): # 作为 LNode 的派生类
    def __init__(self, prev, elem, nxt):
        LNode.__init__(self, elem, nxt)
        self.prev = prev
```

  在单链表结点的基础上扩充了一个前一结点指针

- 基于带尾结点引用的单链表派生一个双链表类：

```
class LDList(LList1):
    def __init__(self):
        LList1.__init__(self)
```

  基类的非变动操作都可继承，无需重新定义（它们不用 **prev** 链接）

- 下面只考虑前后端的元素加入和删除操作

  将看到，两对操作完全是对称的，都能在 **O(1)** 时间完成

双链表：加入元素

- 加入元素的一对操作：

```
def prepend(self, elem):
    p = LDNode(None, elem, self.head)
    self.head = p
    if self.rear is None: # insert in empty list
        self.rear = p
    else: # otherwise, create the prev reference
        p.next.prev = p

def append(self, elem): # 与 prepend 对称
    p = LDNode(self.rear, elem, None)
    self.rear = p
    if self.head is None: # insert in empty list
        self.head = p
    else: # otherwise, create the next reference
        p.prev.next = p
```

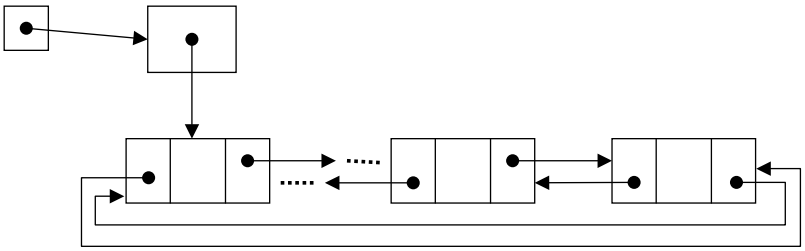

双链表：弹出（删除）元素

```
def pop(self):
    if self.head is None:
        raise ValueError
    e = self.head.elem
    self.head = self.head.next # 删除当前首结点
    if self.head is None:
        self.rear = None # 如果删除后表空，把 rear 也置空
    else:
        self.head.prev = None # 首结点的 prev 链接置空
    return e

def poplast(self): # 与 pop 对称
    if self.head is None:
        raise ValueError
    e = self.rear.elem
    self.rear = self.rear.prev
    if self.rear is None:
        self.head = None # 删除后表空，把 head 也置空
    else:
        self.rear.next = None
    return e
```

循环双链表

- 也可以根据需要，对双链表做一些变形，例如循环双链表：



由于这里有双向链接，掌握着表的首结点（或尾结点），同样能高效实现首尾两端的元素加入/删除操作（**O(1)** 复杂性）

- 下面介绍单链表的两个操作，从中可以看到链表操作的一些特点
  - 表元素反转，要求修改被操作的表，其中元素按原顺序反转
  - 表元素排序，要求修改被操作的表，使表中元素按 **<** 排好顺序

表元素反转

- 反转一个表顺序的元素，算法通常用两个下标，逐对交换元素位置
  - 在双链表上很容易采用该方法实现表元素反转，留作个人练习
  - 在单链表上也可以实现，但单链表不支持从后向前找结点，只能每次从头开始找，算法需要 **$O(n^2)$** 时间。自己想想有别的办法吗？
- 注意：对顺序表，修改表元素顺序的方法只有一种：在表中搬动元素；而对链表却有两种方法：在结点之间搬动元素，或修改链接关系
  - 在单链表里通过搬元素方式实现元素反转，不方便，且效率低
  - 下面考虑基于修改链接的方法，看看有什么可能性
- 首先：表前端加入和删除元素或结点是最方便的操作，**O(1)**
  - 如果不断在前端加入结点，最早放进去的结点在最后（是尾结点）
  - 从表的前端取下结点，最后取下的是尾结点
  - 结合这两个过程，很容易做出一个高效的反转算法

表元素反转

- 方法定义：

```
def rev(self):
    p = None
    while self.head is not None:
        q = self.head
        self.head = q.next # 摘先原来的首结点
        q.next = p
        p = q              # 将刚摘下的结点加入 p 引用的结点序列
    self.head = p      # 反转后的结点序列已经做好，重置表头链接
```

- 通过调度把一列火车车厢的顺序颠倒过来，就是这样的过程

表元素排序

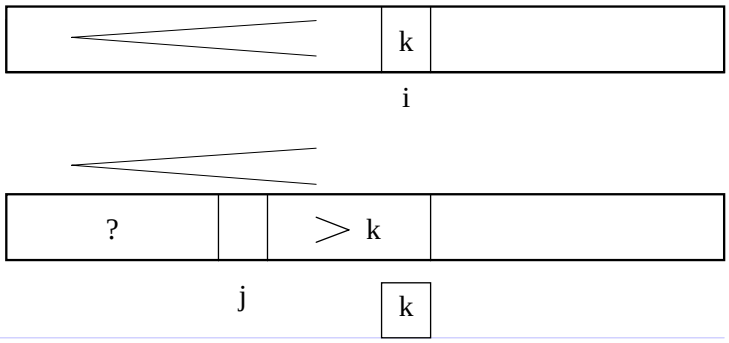
- 把一组物品按顺序排列是在真实世界里经常需要做的工作
  - 数据处理中也经常需要将数据排序
  - 如 lst 的值是 list 对象，lst.sort() 完成 lst 对象中元素的排序工作；sorted(lst) 生成一个新表，其元素是 lst 元素的排序结果
- 用链接表存储数据，也常需要考虑其中元素的排序问题
  - 排序是重要的数据处理问题，人们提出了许多算法（后面考虑）
  - 下面考虑一种简单的排序算法：插入排序
- 插入排序的基本想法：
  - 操作中维护一个排好序的片段，初始时该段只包含一个元素
  - 每次从未排序部分取出一个元素，加入已排序片段中正确位置，保持该片段的正确排序状态；所有元素都加入排序片段时工作完成
  - 通常在表的前部积累排好序的片段

表元素排序

- 先看一个对 list 的元素排序的函数，帮助理解操作的过程

```
def listSort(lst) :
    for i in range(1, len(lst)): # seg [0:0] is sorted
        x = lst[i]
        j = i
        while j > 0 and lst[j-1] > x:
            lst[j] = lst[j-1]  # 反序逐个后移元素，直至确定插入位置
            j -= 1
        lst[j] = x
```

- 算法的两个参考图示：

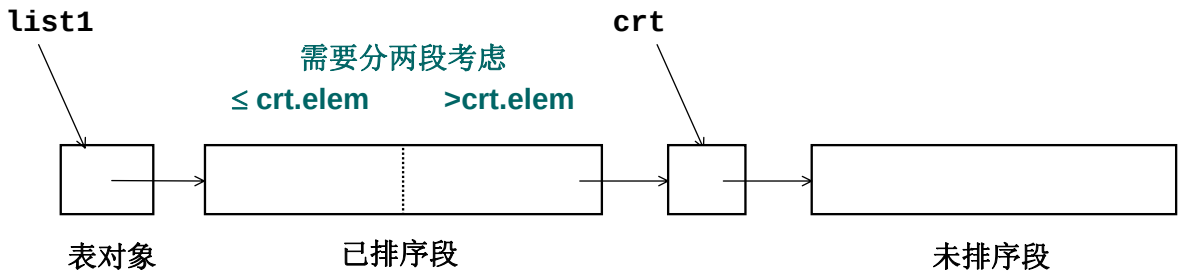


单链表排序

- 下面考虑单链表元素的排序
  - 注意：只有 next 链接，不能从后向前找结点（找元素）
  - 存在完成排序的两种可能做法：移动元素，或调整链接
  - 下面分别考虑采用这两种技术实现的算法
- 先考虑基于移动元素的单链表排序

  注意，为有效实现，算法中只能按从头到尾的方向检查和处理

  做法仍然是每次拿出一个元素，在已排序序列中找到正确位置插入



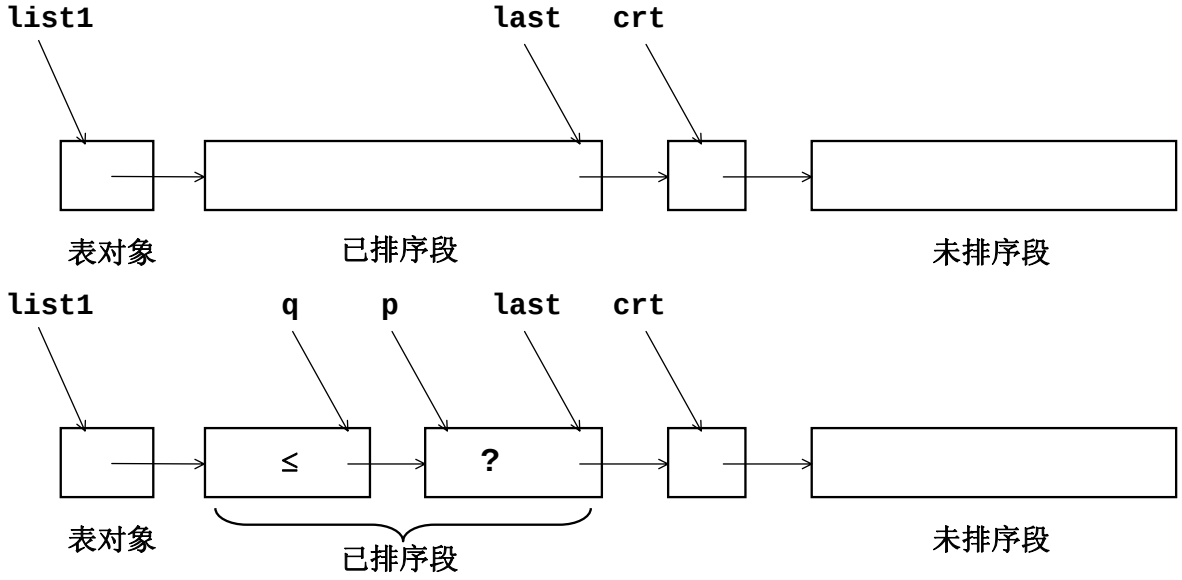
表元素排序

- 算法的实现：

```
def sortm(self):
  if self.head is None:
     return
  crt = self.head.next # 从首结点之后开始处理
  while crt is not None:
     x = crt.elem;
     p = self.head
     while p is not crt and p.elem <= x: # 跳过小元素
        p = p.next
     while p is not crt: # 倒换大元素，完成元素插入的工作
        y = p.elem; p.elem = x; x = y
        p = p.next
     crt.elem = x # 最后的回填
     crt = crt.next
```


表元素排序

- 现在考虑另一种方式：通过调整链接关系完成排序工作
- 基本的处理模式与前一算法类似，但这里不在结点之间移动表元素，而是把被处理结点取下来接到正确位置，逐步完成排序工作



表元素排序

- 根据前面讨论定义的函数：

```
def sort(self):
    if self.head is None:
        return
    last = self.head; crt = last.next # 初始，排序段只有一个结点
    while crt is not None: # 循环，一次处理一个结点
        p = self.head; q = None # 设置扫描指针初值
        while p is not crt and p.elem <= crt.elem:
            q = p; p = p.next # 顺序更新两个扫描指针
        if p is crt: # p 是 crt 时不用修改链接，设置 last 到下一结点 crt
            last = crt
        else:
            last.next = crt.next # 取下当前结点
            crt.next = p # 接好后链接
            if q is None:
                self.head = crt # 作为新的首结点
            else:
                q.next = crt # 或者接在表中间
        crt = last.next # 无论什么情况，crt 总是 last 的下一结点
```


实例：Josephus问题

- 问题：设有 n 个人围坐一圈，现在从第 k 个人开始报数，报到第 m 的人退出。然后继续报数，直至所有人退出。输出出列人顺序编号
- 这个问题上学期大家做过

  下面考察基于几种想法和数据结构的解决方法
- 方法1：基于 Python list 和固定大小的“数组”
  - 初始：建立一个包含 n 个人（的编号）的 list
  - 找到第 k 个人，从那里开始

    处理过程中，通过把相应表元素修改为 0 表示人已不在位
  - 反复做：
    - 数 m 个（尚在坐的）人
    - 把表示第 m 个人的元素修改为 0

    注意：数到 list 最后元素之后转到下标为 0 的元素继续

Josephus问题：数组算法

```
def JosephusA(n, k, m):
    people = list(range(1, n+1))

    i = k-1
    for num in range(n):
        count = 0
        while count < m:
            if people[i] > 0:
                count += 1
            if count == m:
                print(people[i], end = "")
                people[i] = 0
            i = (i+1) % n
            print(", " if num<n-1 else "\n",
                  end="" )
```

使用实例：JosephusA(10, 2, 7)

- 这里的主要麻烦是表下标的计数和元素计数脱节
- 总共 n 个元素，外层循环每次输出一个元素，执行 n 次迭代
- i 表示下标，内层循环迭代一次 i 加一，循环计数；count 统计人数，元素非 0 时加一
- 每次 count 达到 m 就输出当前元素，并将这个元素置 0，表示人已经不在位置
- num 为 n-1 时已经输出了最后一个元素，结束前用 print 输出一个换行符
- 算法复杂性 $O(n^2)$

Josephus问题：连续表算法

- 现在考虑另一算法：将保存人员编号的 list 按照表的方式处理
  - 算出应该退出的人之后，将其从表里删除
  - 计算过程中表越来越短，用 num 表示表的长度，每退出一人，表长度 num 减一
  - 至表长度为 0 时工作结束
- 由于表里全是有效元素
  - 元素计数与下标计数统一了
  - 下标更新可以用 i = (i + m - 1) % num 统一描述
- 基于这些想法写出的程序很简单

Josephus问题：连续表算法

```
def JosephusL(n, k, m):
    people = list(range(1, n+1))

    i = k-1
    for num in range(n, 0, -1):
        i = (i + m-1) % num
        print(people.pop(i), end = "")
        print(", " if num > 1 else "\n", end="")
```

- 简单的循环计数，很容易理解
- 调用实例： **JosephusL(10, 2, 7)**
- 貌似线性时间算法，但不是
  - **pop** 操作需要线性时间 **O(n)**
  - 算法复杂性是 $O(n^2)$

Josephus问题：循环链表算法

- 考虑基于循环单链表实现一个算法
  - 从形式看，循环单链表很好地表现了围坐一圈的人
  - 顺序地数人头，可以自然地反映为在循环表中沿 **next** 链扫描
  - 一个人退出，可以用删除相应结点的操作模拟。这样做之后，就可以沿着原方向继续数人头
- 算法应分为两个阶段：
  - 建立包含指定个数（和内容）的结点的循环单链表，可以通过从空表出发逐个在尾部加入元素的方式完成
  - 循环计数，并删去需要退出的结点
- 具体实现可以有多种方式，例如：为原循环单链表增加一个循环数数的函数，然后写一段程序建立所需的循环单链表，并完成操作

  下面实现基于循环单链表类，派生出一个专门的类，用其初始化方法完成全部工作

Josephus问题：循环链表算法

- 这里采用另一种实现方法
  - 从 LCList 派生出一个类
  - 用这个类的初始化函数完成所有工作
- 派生类 Josephus 增加了新方法 turn，它将循环表对象的 rear 指针沿 next 移 m 步
- 初始化函数调用基类 LCList 的初始化函数建立一个空表

  第一个循环建立包含 n 个结点的循环表

  最后的循环逐个弹出结点并输出结点里保存的编号

```
class Josephus(LCList):
    def turn(self, m):
        for i in range(m):
            self.rear = self.rear.next
    def __init__(self, n, k, m):
        LCList.__init__(self)
        for i in range(n):
            self.append(i+1)
        self.turn(k-1)
        while not self.isEmpty():
            self.turn(m-1)
            print(self.pop(), end="")
            print(", " if num > 1 else "\n",
                  end="")
```

使用实例： JosephusL(10, 2, 7)

算法复杂性 O(m*n)

链接表：优点和缺点

- 优点：
  - 表结构是通过一些链接起来的结点形成的，结构很容易调整修改
  - 不修改结点里的数据元素，只通过修改链接，就能灵活地修改表的结构和内容。如加入/删除一个或多个元素，翻转整个表，重排元素的顺序，将一个表分解为两个或多个等
  - 整个表由一些小存储块构成，较容易安排和管理（不是我们的事）
- 缺点，一些操作的开销大：
  - 基于位置找到表中元素需要线性时间
  - 尾端操作需要线性时间（增加尾指针可以将尾端加入元素变为常量操作，但仍不能有效实现尾端删除）
  - 找当前元素的前一元素需要从头扫描表元素（双链表可以解决这个问题，但每个结点要付出更多存储代价），应尽量避免
- 每个元素增加了一个链接域（存储代价），双链表增加两个链接域