2，线性表(2)

- 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

线性表的实现

- 顺序表结构（技术）是组织一组元素的最重要方式，它
  - 可以直接地实现线性表
  - 也是许多其他数据结构的实现基础
- 采用顺序表结构实现线性表，重要问题是加入/删除等操作的效率
  - 这类操作改变表中元素序列的结构，是典型的变动操作
  - 由于元素在存储区里连续排列
    - 加入/删除操作需要移动（可能很多的）元素，操作代价高
    - 表结构不够灵活，不容易调整和变化
- 如果一个表在使用中经常需要修改结构，用顺序表实现就不很方便，操作代价可能很高，根源在于元素存储的集中方式和连续性
- 如果程序里需要巨大的线性表，采用顺序表实现，就需要很大块的连续存储空间，这也可能造成存储管理方面的困难

链接表（链表）

- 线性表实现的基本需要：
  - 能够找到表中的首元素（无论直接或间接，通常很容易做到）
  - 从表里的任一个元素出发，可以找到它之后的下一个元素

  显然，把表元素保存在连续的存储区里，自然满足这两个需求，顺序关联是隐含的。但满足这两种需求，并不一定要连续存储元素
- 实现线性表的另一方式是基于链接结构，用链接显式地表示元素之间的顺序关联。基于链接技术实现的线性表称为链接表或链表
- 实现链接表的基本想法：
  - 把表元素分别存储在一批独立的存储块（称为结点）里
    - 保证从一个元素的结点可找到与其相关的下一个元素的结点
    - 在结点里用链接的方式显式记录元素（结点）之间的关联
  - 这样，只要知道表中第一个结点，就能顺序找到表里其他元素

表的实现(1)——链接表

- 链接表有多种不同的组织方式。下面先讨论最简单的单链表，其中每个结点里记录着下一元素的结点的标识。后面将介绍一些变形
- 建立链接结构需要功能强大的存储管理技术支持
  - 在 C 等基本编程语言里，这方面的技术细节很多，用起来也比较麻烦，编程中也容易出错
  - 在 Python 等编程语言里，对这类技术的支持非常全面。我们可以很方便的建立和使用复杂的链接结构。在基础编程中，实际上已经广泛用到这类结构（由语言提供），后面会看到更多例子
- 单链表（下面简称链表）结点的形式（链接域保存下一结点的标识）

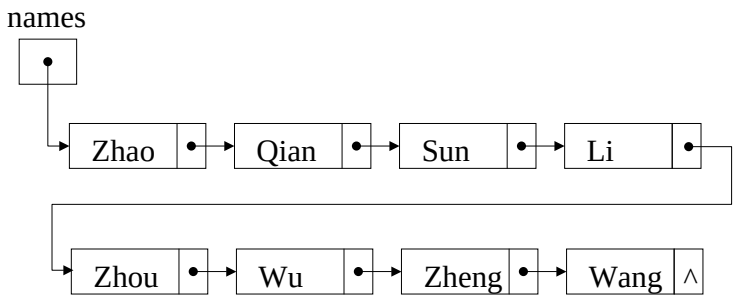
元素	链接域

- 在单链表里，与表里的 n 个元素对应的 n 个结点通过链接形成一条结点链。从表里的任一个结点都可以找到保存下一个元素的结点

单链表

- 要掌握一个单链表，就需要（也仅需要）掌握表的首结点，从它
  - 可以找到表的首元素（表里保存的数据）
  - 还可以找到表中下一结点的位置

  按同样方式继续下去，就可以找到表里的所有数据元素
- 表头变量：保存着链表第一个结点的标识（链接）的变量



单链表

- 总结一下：一个具体的表由一些具体结点构成
  - 每个结点（对象）有自己的标识（下面也常直接称其为链接）
  - 结点之间通过结点链接建立起顺序联系
  - 给表的最后一个结点（表尾结点）的链接域设置一个不会作为结点对象标识的值（Python 里自然应该用 None），称为空链接
- 通过判断是否空链接，可以知道是否已经到了表的结束
  - 在做检索表中元素的工作时，据此判断检索工作是否完成
  - 如果表头指针的值是空链接，说明“它所引用的表已经结束”。没有元素就已经结束，说明这个表是空表
- 在实现算法时，我们并不需要关心具体的表里各结点的具体链接的值是什么（它们总保存在表结构里），只需要关心链表的逻辑结构
  - 链表的操作也只需根据链表的逻辑结构考虑和实现

单链表操作：基本操作

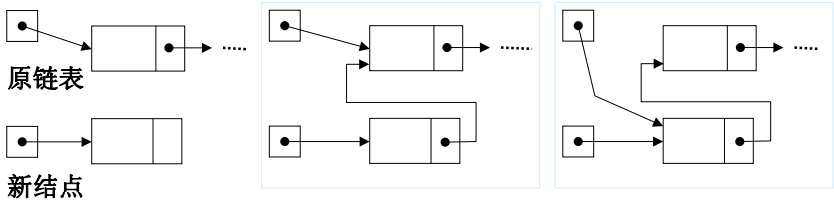
考虑链接表的几个基本操作

- 创建空链表：只需将表头变量设置为空链接
  - 在 Python 里将其设置为 None
- 删除链表：丢弃表的所有结点，与具体环境有关
  - 在一些语言（如 C 语言）里需要做许多事情，释放所用存储
  - 在 Python 里，只需简单将表指针设 None，就丢掉了整个链表的所有结点。Python 程序的存储管理系统会自动回收不用的存储
- 判断表是否为空：将表头变量的值与空链接比较
  - 在 Python 里检查其值是否为 None
- 判断表是否满：链接表不会满，除非存储空间用完

单链表操作：加入元素

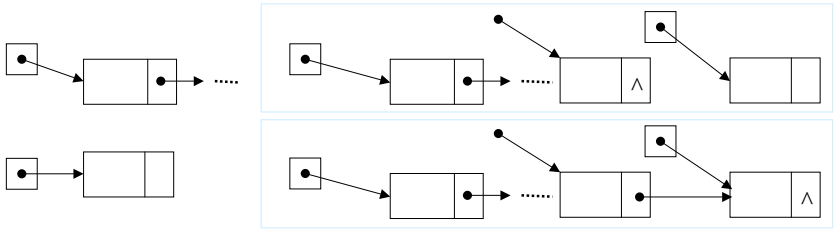
- 给单链表加入元素的一些基本情况
  - 位置可以为首端，尾端，定位。不同位置的操作复杂性可能不同
  - 加入元素不需要移动已有数据，只需为新元素安排一个新结点，然后把新结点连接在表里所需的位置

    通过修改链接，改变表的结构
- 首端加入：1) 创建一个新结点存入数据；2) 把原链表首结点的链接存入新结点的链接域；3) 修改表头变量使之引用新结点

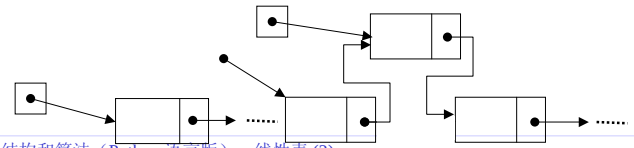


单链表操作：加入元素

- 尾端加入：1) 创建一个新结点存入数据；2) 表空时直接让表头变量引用这个新结点并结束，否则找到表尾结点；3) 令表尾结点的链接域引用这一新结点，并将新结点的链接域设置为空链接



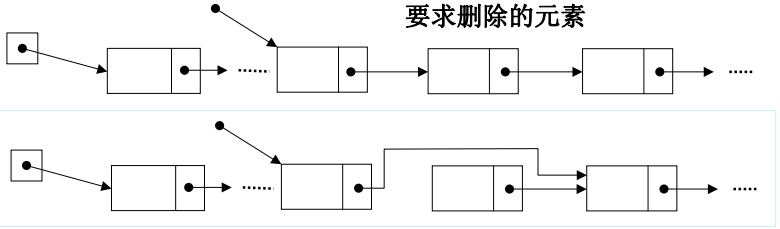
- 定位加入：1) 找到新结点加入位置的前一结点，不存在时结束；2) 创建新结点存入数据；3) 修改前一结点和新结点的链接域将结点连入



单链表操作：删除元素

删除元素，所用技术与加入元素类似

- 首端删除：直接修改表头指针，使之引用当时表头结点的下一个结点。Python 系统里会自动回收无用对象的存储块，下同
- 尾端删除：找到倒数第二个结点，将其链接域设置为空链接
- 定位删除：找到要删除元素所在结点的前一结点，修改它的链接域将要求删除的结点从表中去掉

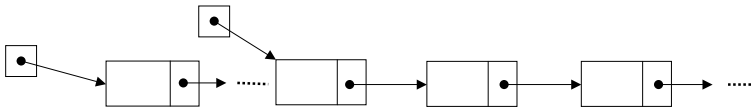


- 后两种删除操作的工作方式类似，算法细节可能略有不同

单链表操作：扫描和遍历

- 许多操作中需要扫描表里一个个结点，可能检查其中的元素，如
  - 这种操作的过程称为遍历，顺序检查一个数据结构的所有元素
  - 求表元素的个数
  - 在表中查找特定位置的元素，或查找满足某些条件的元素

  进行这类操作，都需要用一个（或几个）扫描变量



- 有些表操作比较复杂，例如表元素排序

  排序问题存在许多有趣的算法，以后有专门的讨论
- 现在来考虑表操作的时间复杂性

单链表操作：复杂性

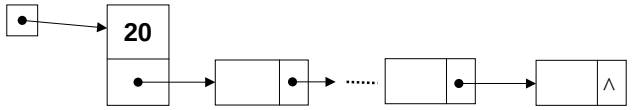
- 基本操作：
  - 创建空表：O(1)
  - 删除表：在 Python 里是 O(1)。当然存储管理也需要时间
  - 判断空表：O(1)
- 加入元素（都需要加一个 T(分配) 的时间）：
  - 首端加入元素：O(1)
  - 尾端加入元素：O(n)，因为需要找到表的最后结点
  - 定位加入元素：O(n)，平均情况和最坏情况
- 删除元素：
  - 首端删除元素：O(1)；尾端删除：O(n)
  - 定位删除元素：O(n)，平均情况和最坏情况
  - 其他删除通常需要扫描整个表或其一部分，O(n) 操作

单链表操作：复杂性

- 其他操作，如果需要扫描整个表或其一部分，都是 O(n) 操作。如
  - 求表的长度（表中元素个数）
  - 定位表中的元素；等等
- 一类典型的表操作是扫描整个表，对表中每个元素做同样的工作（即遍历操作）。例如，输出所有的元素值，将它们累积到一个变量里等。这种工作可以通过一个循环完成

  遍历操作的复杂性应该是 O(n) * T(元素操作)

- 有可能改造表的表示方式，提高一些操作的效率。例如，如果工作中经常需要求表长度，可以考虑采用下面结构（加一个表头对象）：



这样做，在加入/删除元素时需要维护个数记录，有得有失

单链表的 Python 实现

- 实现链接结构，需要定义相应的类，首先是表示结点的类
- 下面是一个简单的结点类：

```
class LNode : # 只定义初始化操作
    def __init__(self, elm, nxt):
        self.elem = elm
        self.next = nxt
```

- 简单的使用代码（Python 允许直接访问对象的普通数据域）：

```
llist1 = LNode(1, None); pnode = llist1
for i in range(2, 11):
    pnode.next = LNode(i, None)
    pnode = pnode.next
pnode = llist1
while pnode is not None:
    print(pnode.elem)
    pnode = pnode.next
```


单链表实现：几个基本操作

- 我们希望基于结点 LNode 定义一种链接表类型，为此定义一个表类

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

    def isEmpty(self):
        return self.head == None

    def prepend(self, elem):
        self.head = LNode(elem, self.head)
```

- LList 对象只有一个 head 域，指向表中的首结点。几个操作（方法）：
  - 初始建立的表里没有结点（空表）
  - 根据 head 的值判断是否空表
  - prepend 在表首端加入一个包含新元素的（新）结点

单链表实现：尾端加入

- append 在表最后加入一个包含新元素的结点

```
def append(self, elem):
    if self.head == None:
        self.head = LNode(elem, None)
        return
    p = self.head
    while p.next != None:
        p = p.next
    p.next = LNode(elem, None)
```

注意，这里需要区分两种情况：

- 如果加入新元素时原表为空，就用 head 记录新加的结点
- 如果表不空，需要先通过循环找到当时表里的最后一个结点，然后用这个结点的 next 域记录新结点的链接
- 复杂性（最坏情况）显然为 O(n)

单链表实现：首/尾端弹出

- 首/尾端弹出元素的方法（删除操作与此类似）

```
def pop(self): # 首端弹出
    if self.head == None:
        raise ValueError
    e = self.head.elem
    self.head = self.head.next
    return e

def poplast(self): # 尾端弹出，显然复杂性为 O(n)
    if self.head == None: # empty list
        raise ValueError
    p = self.head
    if p.next == None: # list with only one element
        e = p.elem; self.head = None
        return e
    while p.next.next != None: # till p.next be the last node
        p = p.next
    e = p.next.elem; p.next = None
    return e
```


单链表实现：其他操作

- 下面两个操作实现有用的功能：

```
def find(self, pred): # 在表里找到第一个满足 pred 的元素返回
    p = self.head
    while p != None:
        if pred(p.elem):
            return p.elem
        p = p.next
    return None

def printall(self): # 输出表中所有元素
    p = self.head
    while p != None:
        print(p.elem)
        p = p.next
```

- 还可根据需要增加其他有用的操作，如定位删除等，留作自由练习

  请查看基本类型 list 的操作，想想如何为这个链接表类实现它们

单链表

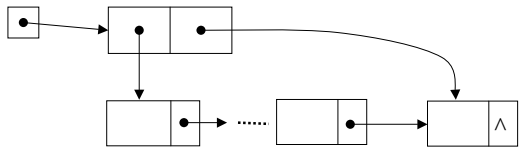
- 下面是一段简单的使用链接表的代码

```
mlist1 = LList()

for i in range(10):
    mlist1.prepend(i)

for i in range(11, 20):
    mlist1.append(i)

mlist1.printall()
```

  - 建立一个空表
  - 通过循环在表首端加入 10 个元素（整数）
  - 通过循环在表尾端加入 9 个元素（整数）
  - 顺序输出表里的所有元素

单链表的变形

- 单链表并非只有一种设计，可以根据需要和认识修改设计，例如
  - 前面实现的一个缺点是尾端加入操作的效率低
  - 实际中可能经常需要频繁地在表的两端加入元素
- 一种可能是采用下面的结构，给表对象增加一个对表尾结点的引用：



  这样，在尾端加入元素，也能做到 **O(1)**

- 注意：新设计的链表与前面单链表结构近似，结构变化应该只影响到表的变动操作，非变动操作不需要修改。有可能重用前面定义吗？

单链表的变形：带尾结点引用

- 面向对象技术支持基于已有的类（基类）定义新类（派生类）
  - 派生类继承其基类的所有功能（数据域和方法）
  - 派生类可以定义新的数据域，定义新的方法
  - 派生类可以重新定义基类里已定义的方法（覆盖已有方法）
- 回到链表，我们可以基于 LList 定义（具有前述结构的）新链表类
  - 让它继承 LList 的所有非变动操作
  - 增加一个尾结点引用域，重新定义表的变动操作
- 通过继承方式定义新类

```
class LList1(LList) :
    # 方法定义和其他
```

- Python 规定，定义时不注明基类，自动以公共类 object 作为基类

  前面的 LNode 和 LList 都以 object 作为基类

带尾结点引用的单链表

- LList1 定义为 LList 的派生类，覆盖 LList 的一些方法

```
class LList1(LList):
    def __init__(self):
        LList.__init__(self) # 调用 LList 的初始化方法
        self.rear = None

    def prepend(self, elem):
        self.head = LNode(elem, self.head)
        if self.rear == None: # the empty list
            self.rear = self.head # rear points also to the new node

    def append(self, elem):
        if self.head == None:
            self.prepend(elem) # call prepend, do the same
        else:
            self.rear.next = LNode(elem, None)
            self.rear = self.rear.next
```


带尾结点引用的单链表

- 首端和尾端删除方法也需要覆盖

```
def pop(self):
    if self.head is None:
        raise ValueError
    e = self.head.elem
    if self.rear is self.head: # list with one node
        self.rear = None
    self.head = self.head.next
    return e

def poplast(self):
    return None # to be implemented or simply use this
```

- 带尾结点记录的单链表可以很好支持前/尾端加入和首端弹出元素
  - 对尾端弹出（删除）操作，新结构也没有优势
  - 上面没实现（简单覆盖屏蔽了原方法的执行），可以自己实现