2，线性表(I)

- ❖ 计算机内存结构
- ❖ 数据结构的基本实现技术
- ❖ Python 对象和变量
- ❖ 线性表：概念
- ❖ Python list：线性表的一种实现
- ❖ 链接表
- ❖ 线性表的变形
- ❖ 应用

内存结构模型

- 要理解数据结构处理的问题，需要对计算机内存、存储管理方面重要基本问题有一些了解。现在首先介绍这方面的一些基本情况
- 计算机的基本内存结构：
  - ❑ 内存是线性排列的一批存储单元，单元有唯一编号，称为单元地址
  - ❑ 单元地址从 0 开始址连续排列，可用地址是一个连续整数区间
  - ❑ 对内存单元的访问（存取其中的数据）都通过单元地址进行。因此，要访问一个单元，必须先掌握其地址
  - ❑ <u>基于地址访问单元是 O(1) 操作，与单元位置或内存大小无关</u>

程序运行中构造、使用、处理的对象，都在这种线性结构里安排位置

内存和对象存储

- 程序运行中建立/存在的每个对象都要占用一块（或大或小的）内存
  - 建立的每个对象都有确定的唯一标识（例如内存位置），在其存续期间保持不变，这是一个基本原则
  - 知道一个对象的位置就能访问（使用）它，已知位置访问相应对象的操作可以在常量时间完成
- 如果一个组合对象包含一组元素，它们在一块元素存储区里连续存储，每个元素的存储量相同，基于存储区位置和编号访问元素是 **O(1)** 操作
  - 设对象的元素存储区的起始位置是 **p**，每个元素占用 **a** 个内存单元，再假设第一个元素编号为 **0**
  - 要访问编号为 **k** 的元素，其位置 **loc** 可以通过下式计算

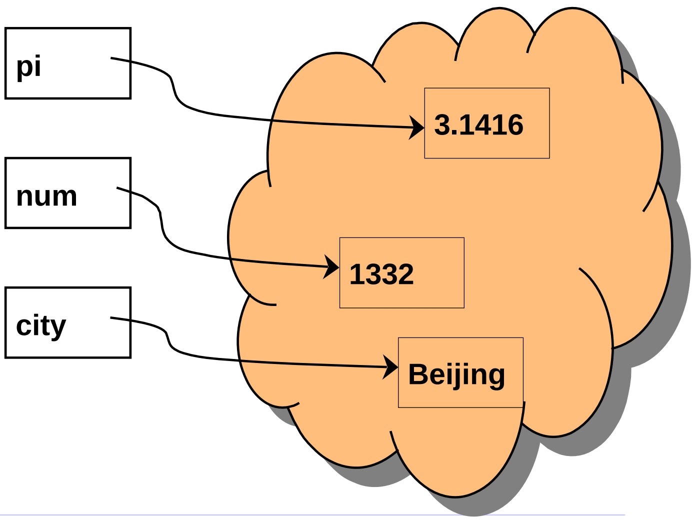
$$loc = p + k*a$$

  - 显然，计算元素位置（及访问元素）所用时间与元素编号无关，也与组合对象的元素个数无关（连续存储可以 **O(1)** 时间访问）

变量和对象

- 程序里的变量（全局的、局部的，以及函数参数）有系统化的存储安排方式，是另一套专门机制，下面讨论中不考虑。实际上：
  - 变量也在内存安排位置，每个变量占用若干存储单元
  - 程序运行中总能找到根据作用域可见的那些变量，取得或修改其值
- 在 **Python** 里，可以
  - 通过初始化（或提供实参）给变量（或函数参数）约束一个值（对象）
  - 用赋值修改变量的约束值
- 给变量约束一个值对象，就是把该对象的标识（内存位置）保存在变量里

  图示



变量和值

- Python 里变量的值都是对象，可以是：
  - 基本类型（如基本整数、浮点数等）的对象，大小固定且比较小
  - 复杂的对象，例如 list 等，可能比较大（包含一组成分对象），需要的存储单元可能不同（不同的 list 有长有短），可能有复杂的内部结构（例如，其元素又可能是复杂的数据对象），等等
- Python 程序运行时内部有一个专门的存储管理系统，负责管理程序可用的内存，支持灵活有效的内存使用
  - 程序中要求建立对象时，为这些对象安排存储
  - 当某些对象不再有用时回收它们占用的存储
  - 存储管理系统屏蔽了具体内存使用细节，减少编程人员的负担
- 在写 Python 程序时，通常不需要关心存储管理的具体细节
  - 但应注意，运行中存在的对象都需要存储，过多的对象有可能用完所有可用存储，这种情况下程序只能崩溃

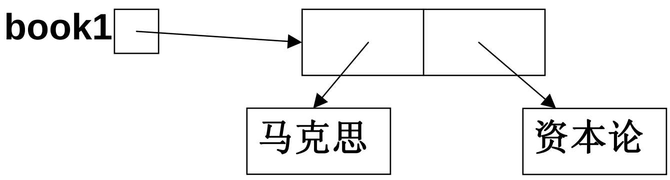
对象创建和变量约束

- 假设要给变量 s 赋值一个新字符串，系统需要：
  - 找一块足够大的内存块，把字符串的内容复制进去
  - 把内存块的地址信息存入变量 s

- 这样做通常不够
  - 内存单元里存储的都是二进制编码，仅从单元里存储的内容无法判断这一字符串到哪里结束
  - 需要有一种安排（约定）。由于字符串可以有任意的长度，一种可能安排是在相应存储块的开始记录字符串长度，如：

“表示”及其设计

- 程序中生成和处理的对象都要以某种方式保存，因此要设计好它们的存储方式。这种方式及其效果称为该对象的“表示”（**representation**）。前例就是为字符串设计了一种存储表示
- 字符串的结构最简单，可以用一块连续存储区表示，类似的情况如数学里的 **n** 维向量。但并非所有对象都如此
- 假设要表示一类对象，它有两个成员对象，都是字符串，一个表示作者（作者名），一个表示图书标题（书名）
  - 书名和作者都可以用字符串表示，但两个字符串长度不定
  - 作为整体的对象怎样表示，才能支持灵活方便的处理？
- 一种做法是用三块内存：



- 变量保存二元结构的地址
- 二元结构里存字符串地址
- 这种结构称为链接结构，用于表示复杂结构和联系

Python 的对象表示

- **Python** 系统的实现基于一套精心设计的链接结构
  - 各种复杂对象，甚至 **Python** 程序等，都基于独立的存储块实现，通过链接相互关联。有关情况将随着课程进展逐渐看清楚
  - 各种数据对象的表示方式，对相关结构上各种操作的效率有着简单性的影响，也间接影响着用 **Python** 做的程序
  - 理解这些结构，可以帮助我们更有效地使用 **Python**
- 一般而言
  - 基于较低级的编程语言（例如 **C**）工作时，可以根据需要设计数据结构的表示（实现）方法。常规数据结构课程关注这方面问题
  - 基于 **Python** 等高级编程系统工作时，特别是做复杂工作时，也常需要自己设计一些数据结构，还需要对语言提供的各种结构的基本原理有很好理解，才能更有效地使用它们
- 下面将开始本课程的主干部分。首先是一种最常用的结构：线性表

线性表

- 程序里经常需要将一组（某类型的）元素作为整体管理和使用
  - 该组数据里元素个数可能变化（可以加入或删除元素）
  - 有可能需要把这样一组元素看成一个序列，元素在序列里的位置和顺序可能表示实际应用中某种有意义的信息或关系
  - 这样一组元素（的序列）的抽象就是线性表（简称表）。线性表是一种元素集合，其中还记录着元素间的一种顺序关系
- 线性表是最基本的一种数据结构
  - 在程序里应用很广泛
  - 还常作为更复杂的数据结构的实现基础
  - 例如整数的表，字符串的表，某种复杂结构的表等
  - **Python** 的 **list** 和 **tuple** 支持这类需要，可看作是线性表的实现
- 本章讨论线性表概念，两种基本实现方式，若干变形和应用实例

概念和术语

- 抽象讨论线性表时，考虑一个基本元素集合 $E = \{e_0, \ldots, e_{N-1}\}$，其中的元素可能是某个类型的成员
- 表是元素的有穷序列，有 **0** 个或多个元素 $(e_0, e_1, \ldots, e_{n-1})$，$n \geq 0$
  - 元素的位置称为其下标，下标从 **0** 开始编号（也可选择从 **1** 开始）
  - 表中元素的个数称为表的长度，长度为 **0** 的表是空表
  - 元素间基本关系是下一个关系：$<e_0, e_1>, <e_1, e_2>, \ldots, <e_{n-2}, e_{n-1}>$，这是一种顺序关系（线性关系）
- 线性表是一种线性结构。在一个非空线性表里：
  - 存在唯一的 “首元素”，唯一的“尾元素”（末元素）
  - 除首元素外，表中每个元素都有且只有一个前驱元素
  - 除尾元素外，表中每个元素都有且只有一个后继元素
- 可以把线性表作为数学对象建立抽象模型，最后有几张幻灯片供参考

线性表

- 从实际角度看，线性表是一种组织数据元素的结构。作为一种抽象的数据结构，需要从两个角度考虑
  - 从实现者角度需要考虑两个问题：**1**，如何把该结构内部的数据组织好（为它设计一种合适的表示）；**2**，如何提供一套有用而且必要的操作，并有效实现这些操作。显然两者相关
  - 从使用者角度，需考虑该结构提供了哪些操作，如何有效使用以解决自己的问题。实际使用会对表的实现者提出一些要求
- 两种角度既有统一又有分工。情况与函数的定义与使用类似
  - 数据结构的表示完全是内部的东西，外面看不到。但它会对这一数据结构上各种操作的实现和性质产生重要影响
  - 对复杂的数据结构，由于存在多种可能表示（后面会看到），设计时需要考虑的因素很多，利弊得失的权衡可能困难而复杂
- 下面首先从使用者的角度，考虑表数据结构应提供哪些必要操作

数据结构的操作

- 作为一种包含元素（可以没有，也可以有许多）的数据结构，通常都需要提供一些“标准”操作，例如：
  - 创建和销毁这种数据结构（的实例）
  - 判断一个数据结构是否空（没有元素）。如果数据结构的容量有限制，还需判断它是否满（不能再加入新元素）
  - 向结构中加入元素或从中删除元素
  - 访问结构里的元素
- 不同编程语言也可能影响需要实现的操作集合

  例如，**Python** 能自动回收不用的对象，因此不需要销毁结构的操作
- 除上述共性操作外，具体数据结构还需要提供一些特殊操作。如：
  - 集合数据结构需要支持各种集合运算（求并集，交集等）
  - 图数据结构要提供判断结点是否相邻（两点间是否有边）的操作

数据结构的操作

- 从作用看，数据结构的操作可以分为三类：
  - 构造操作，它们构造出该数据结构的一个新实例
  - 访问操作，它们从已有数据结构中提取某些信息，但不创建新结构，也不修改被操作的结构
  - 变动操作，它们修改已有的数据结构
- 从支持操作类型的角度看，数据结构可以分为两类：
  - 不变数据结构，只支持前两类操作，不支持变动操作。创建之后结构和存储的元素信息都不改变，所有得到该类结构的操作都是创建新的结构实例。例子如 **Python** 的 **tuple** 和 **frozenset**
  - 变动数据结构，支持变动操作。在创建之后的存续期间，其结构和所保存的信息都可能变化。例子如 **Python** 的 **list, dict, set**
- 实现数据结构时，可以根据需要考虑是实现为不变数据结构，还是实现为可变数据结构，所实现的结构提供哪些操作等

线性表的操作

- 假设为表数据结构取一个类型名 `List`。为简洁严格地表示操作的对象和结果，下面介绍一种数学表达形式，形式如下面描述：

  ```
  opname : T1 * T2 -> ResT
  ```

  其中 `opname` 是操作名，`T1 * T2` 表示有两个参数，类型分别为 `T1` 和 `T2`，`->` 之后的 `ResT` 表示操作的结果类型

  用 `()` 表示没有参数或者操作不返回任何结果
- 考虑一些有用的操作，应根据需要选取能说明操作意义的名字
- 表数据结构基本的创建和销毁，判断空和满的操作

  ```
  newList : () -> List          # 创建一个空表
  delList : List -> ()          # 销毁表
  emptyList : List -> bool      # 表空?
  fullList : List -> bool       # 表满?
  ```

线性表的操作

- 表元素加入操作，可考虑：

```
prepend : List * Data -> List   # 首端加入
append : List * Data -> List    # 尾端加入
insert : List * int * Data -> List # 定位加入
  # 要求将元素加入表中特定位置，原处该位及其后的元素后移
```

- 表元素删除操作，可以考虑：

```
delFront : List -> List          # 删除首元素
delEnd : List -> List            # 删除末元素
delete : List * int -> List     # 定位删除
delElem : List * Data -> List    # 删除一个 Data
delAllElem : List * Data -> List # 删除所有 Data
```

- 定位元素访问，取得位于指定位置（下标）的元素：

```
getElem : List * int -> Data
```


线性表的操作

- 其他操作，如

```
length : List -> integer         # 表元素的个数
locate : List * Data -> integer
  # 查找元素在表中第一个出现的位置，没有时返回特殊值（如-1）
sortList : List -> List
  # 将表中元素按上升序重新排列（sorting，排序）
```

- 具体表结构可能只实现一部分操作，也可能根据需要考虑其他操作
- 牵涉到表内容（或结构）变化的操作有两种可能考虑
  - 总构造一个新表作为操作的结果
  - 按操作的需要直接修改作为参数的表

  前一方式语义更清晰，后一方式可能效率更高（少创建新结构）

  对不变的表（如 Python 的 tuple）只能按前一方式定义操作

表数据结构的实现模型

- 实现表数据结构，主要考虑两方面的情况
  - 计算机内存的特点，以及保存元素和元素顺序信息的需要
  - 重要操作的效率。其中使用最频繁的操作通常是：（定位）元素访问，元素加入，元素删除，元素遍历
- 元素遍历就是依次访问表里的所有（或一批）元素
  - 操作效率与访问元素的个数有关
  - 遍历所有元素的操作，希望其复杂性不超过 **O(n)**
- 加入**/**删除**/**访问元素的操作效率与表的实现结构有关
- 基于各方面考虑，人们提出了两种基本实现模型
  - 将表元素顺序存放在的一大块连续的存储区里，这样实现的表也称为顺序表（或连续表），元素顺序有自然的表示
  - 将表元素存放在通过链接构造起来的一系列存储块里（链接表）

顺序表模型

- 顺序表的基本实现方式
  - 元素顺序存放在一片足够大的连续存储区里。表中首元素存入存储区开始的位置，其余元素依次顺序存放
  - 通过元素在存储区里的“物理位置”表示元素之间的逻辑顺序关系（隐式表示元素间的关系）
- 一般情况是表元素所需存储量相同，因此顺序表中任一元素的位置都可简单计算出来，存取操作可以在 **O(1)** 时间内完成
- 元素 e_i 的地址计算公式（元素编号从 **0** 开始）：

$$\mathbf{Location}(e_i) = \mathbf{Location}(e_0) + c * i$$

其中 **c = size_of(**元素**)**，是一个元素的存储量（元素大小）

元素大小通常可以静态确定（如元素是整数，实数，或包含若干大小确定的元素的复杂结构）

顺序表的元素存储

- 如果表中元素的大小有可能不同，只要略微改变顺序表的存储结构。仍能保证 **O(1)** 时间的元素访问操作（下一页介绍）
- 顺序表的基本表示方式：

逻辑地址	数据元素
0	e_0
1	e_1
…	…
i	e_i
…	…
n-1	e_{n-1}

存储地址	数据元素
$Location(e_0)$	e_0
$Location(e_0)+c$	e_1
…	…
$Location(e_0)+i*c$	e_i
…	…
$Location(e_0)+(n-1)*c$	e_{n-1}

顺序表的元素存储

- 如果表中要保存的元素的情况复杂，大小不一，或者还有复杂的内部结构，可以采用链接方式，在表中保存元素链接（链接的大小相同）

存储地址	数据元素	
$Location(a_0)$	• →	e_0
$Location(a_0)+c$	• →	e_1
…		…
$Location(a_0)+i*c$	• →	e_i
…	…	…
$Location(a_0)+(n-1)*c$	• →	e_{n-1}

c 是链接的大小。元素另外表示，为另外的简单元素或复杂结构

顺序表

- 基本表示方式确定后，还要进一步考虑表结构和操作的特点
  - 表的一个重要性质是可以加入/删除元素
  - 也就是说，在一个表存续期间，其长度可能变化
- 问题：建立表时采用多大一块存储区？
  - 存储块一旦分配，就有了固定大小（确定的元素容量）
  - 按建立时确定的元素个数分配存储，适合创建不变表（如 **tuple**）。要考虑变动的表，就应该区分元素个数和存储区容量
- 合理的办法：分配足以容纳所需元素的存储块，可以有一些空位
  - 表里的一般情况是存在着一些元素和一些可以存放元素的空位
  - 应约定元素的存放方式，通常把元素连续放在存储区的前面一段
  - 为保证正确操作，需要记录块大小和现有元素个数的信息

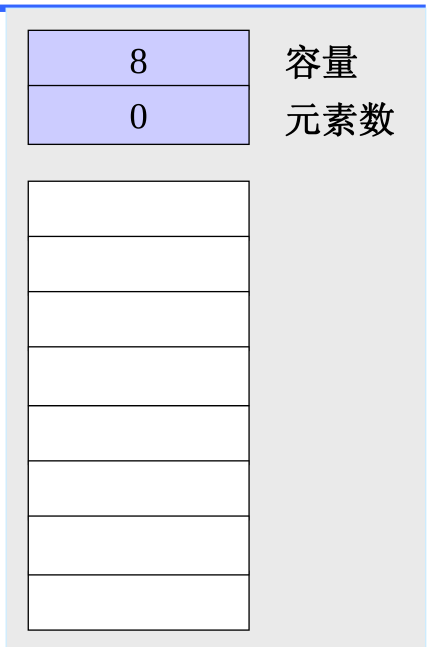
顺序表：实现（布局）和操作

- 易见：
  - 元素存储区的大小决定了表的容量
  - 个数记录要与实际元素个数保持一致。表元素变化时需要维护这一记录
  - 元素个数等于容量表示这个表已满，再加入元素就会失败（或考虑其他技术）
- 访问第 **i** 个元素时计算位置直接找到。复杂性 **O(1)**（显然，只能在元素范围内访问）
- 空/满判断很容易实现：
  - 表空 **iff** 元素计数值等于 **0**
  - 表满 **iff** 元素计数值等于容量
  - 显然都是 **O(1)** 操作

顺序表的操作

- 创建空表是分配一块存储，记录容量并设置元素计数值为 **0**，右图是个容量为 **8** 的空表

  建立新表后应立即设置两个记录域（例如 **max** 和 **n**），保证表处于合法状态
- 只要掌握着元素存储区开始位置（首元素的位置），各种访问操作都很容易实现
- 遍历操作：
  - 只需在遍历过程中用一个整数变量记录遍历达到的位置
  - 通过存储区开始位置和上述变量的值，**O(1)** 时间可算出相应元素的位置
  - 找下一元素的更新操作就是加一，找前一元素的操作就是减一



顺序表的操作

- 访问给定下标 **i** 的元素
  - 需判断 **i** 值是否在表当时的合法元素范围内（**0 ≤ i ≤ n-1**）
  - 不在范围内是非法访问，合法时从给定位置取得元素的值
- 查找给定元素 **d** 的（第一次出现的）位置
  - 通过循环，将 **d** 与表里的元素逐个比较
  - 通过下标变量控制循环，从 **0** 开始至大于表中元素个数时结束
  - 找到元素时返回元素下标，找不到时返回一个特殊值（例如 **–1**）
- 查找给定元素 **d** 在位置 **k** 之后的第一次出现的位置
  - 与上面操作的实现方式类似
  - 只是从 **k+1** 位置的元素开始比较（而不是从位置 **0**）
- 不修改表结构的操作都是这两种模式（直访，或按下标循环并检查）

顺序表的操作（尾端操作）

- 考虑加入和删除元素的操作。尾端加入和删除操作的实现很简单，在其他位置加入删除的操作麻烦些
- 尾端加入元素（**O(1)** 操作）
  - 检查表是否满，表满时操作失败
  - 把新数据存入元素存储区的第 **n** 个单元
  - 将元素计数变量 **n** 加一
- 尾端删除元素（**O(1)** 操作）
  - 简单地把元素计数变量 **n** 减一
- 首端加入和定位加入都比较麻烦，因为
  - 要保证元素在存储区前段连续存储
  - 可能需要维持原有元素的顺序

8
4

255
108
44
1024

顺序表的操作（加入元素）

- 首端/定位加入元素时需要移动已有元素，腾出要求存入元素的位置

  假设要把 **168** 加在前面表里的位置 **1**

- 不要求保持原有元素顺序，把指定位置元素移到最后，存入新元素（**O(1)**）

255	255
108	168
44	44
1024	1024
	108

- 要求保持原有元素顺序，必须逆序地逐个下移后面元素，直至腾出指定位置后将元素放入（**O(n)**）

255	255	255
108		168
44	108	108
1024	44	44
	1024	1024

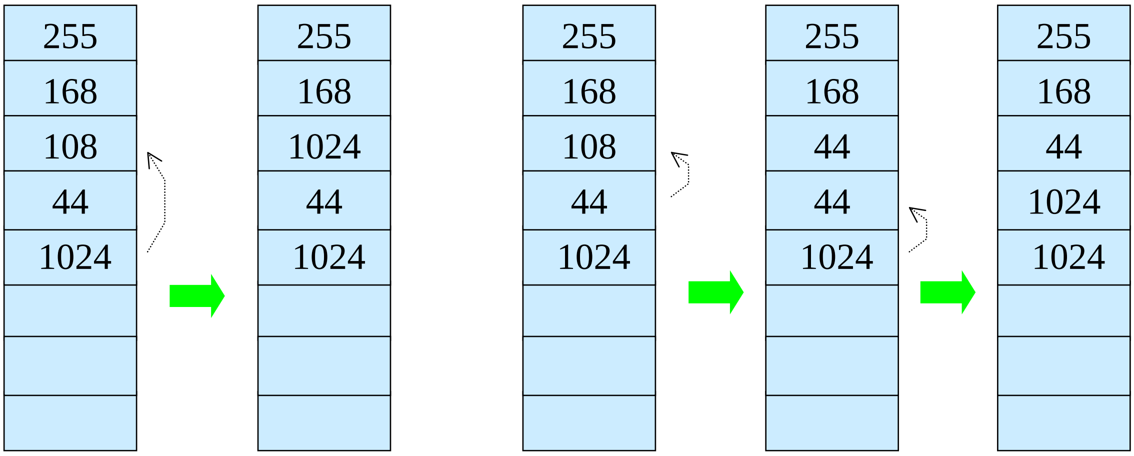
- 加入新元素后需要更新元素计数值

顺序表的操作（删除）

- 首端及定位元素删除都需要移动已有元素，保证所有元素连续存储

  假设要删除位置 **2** 的元素

- 如不需要保持原有元素顺序。可以直接用最后一个元素覆盖指定位置的元素
- 如果需要保持原有元素顺序位置，就必须顺序地逐个将指定位置之后的所有元素上移



- 删除元素后更新元素计数值，最后的元素就“看不到了”

顺序表的操作复杂性

- 一些操作的复杂性是常量 **O(1)**。现在特别考虑定位加入和删除操作的复杂性（首端加入删除是它们的特例）
- 在 **n** 个元素的顺序表里下标 **i** 处加入元素，需要移动 **n - i** 个元素；删除下标为 **i** 的元素需要移动 **n - i - 1**个元素

  设在位置 **i** 加入和删除元素的概率分别是 p_i 和 p_i'

  加入操作平均移动次数 $$\sum_{i=0}^{n}(n-i)p_i$$

  删除操作平均移动次数 $$\sum_{i=0}^{n-1}(n-i-1)p_i'$$

- 考虑平均复杂性时要考虑实例分布，依赖于实际情况

  如果各种情况平均分布，要维持顺序，平均时间复杂性是 **O(n)**

  最坏情况是首端加入/删除，要求维持顺序时复杂性也是 **O(n)**

顺序表的操作复杂性

- 访问操作
  - 不需要扫描表的操作，复杂性都是 **O(1)**
  - 扫描表内容操作复杂性 **O(n)**。如根据元素值检索，累计元素值等
- 顺序实现（顺序表）总结
  - 优点：
    - **O(1)** 时间（随机，直接）按位置存取元素
    - 元素存储紧凑，除表元素存储外只需 **O(1)** 空间存放辅助信息
  - 缺点：
    - 需要连续的大块存储区存放表中的元素，可能有大量空闲单元
    - 加入删除操作时通常要移动许多元素
    - 需要考虑元素存储区的大小（有时事先很难估计）

顺序表的实现

- 顺序表的模型包括两部分内容
  - 一个元素存储区，存放表中的实际元素
  - 若干单元，存放一个表的全局信息（容量，表元素个数）
- 一个数据结构应该具有一个对象的整体形态。对顺序表，就是要把这两块信息组织关联起来。表的全局信息只需常量存储，存在两种组织方式
- 一块存储连续存放这两部分信息
- 用两块存储区，通过链接联系

称为“一体式实现”

称为“分离式实现”

顺序表的一般性讨论至此结束。下面考察一个实例：**Python** 的 **list**

Python 的 list

- list 是一种线性结构，可看作线性表的一种实现。重要特点
  - 基于下标（位置）的元素访问和更新操作，复杂性为 O(1)
  - 允许任意加入元素（不会出现“表满”而无法加入新元素的情况），而且在不断加入元素的过程中，表对象标识（ id(...) 的值）不变
- list 实现的基本约束和解决方案
  - 要求 O(1) 的元素访问并维持元素的顺序，只能采用连续表技术，元素保存在一块连续存储区
  - 要能容纳任意多元素，必须在元素个数将要超出存储区容量时换一块更大存储区。要想在替换存储时 id 不变，只能采用分离式实现
- 采用上述实现方法，自然的后果：
  - 一般的元素加入/删除都需要 O(n) 时间，需要移动许多元素
  - 如果要求加入元素时存储区已满，就需要换一块存储，把原有元素拷贝过去（优化：可以在拷贝过程中完成元素加入）

list 的逐步建立

- 较大的 list，通常是通过不断加入元素逐步建立起来的
  - 加入一个元素，一般情况需要 O(n) 时间
  - 建立起 n 个元素的表，就需要 $O(n^2)$ 时间
- 最好的情况：尾端加入，不需要移动元素
- 注意，不断在尾端加入元素的过程中可能出现两种情况
  - 如果存储区不满，O(1) 时间可以完成操作
  - 如果存储区满，就需要 O(n) 时间（换存储区，拷贝元素）
  - 最坏情况复杂性一定是 O(n)，但能否得到较好的平均时间？
- 情况：如果高开销操作很少出现，平均操作代价可能比较低。考虑
  - 每次替换存储区增加 10 个空位，10 次加入才有一次高代价
  - 但注意，(1/10) O(n) = O(n)，平均复杂性的性质没变

list 的逐步建立

- list 里元素越多（表越长），换一次存储区的代价也越高
  - 要想平均结果较好，随着表长度增加，换存储区的频度应降低
  - 一种可能做法：每次换存储区时，容量加倍
- 计算：假设表的初始容量为 1，不断增长到长度为 $2^{20} \approx 1{,}000{,}000$
  - 加入操作共做了大约 10^6 次
  - 替换存储时的元素拷贝：

    $1 + 2 + 4 + 8 + \ldots + 2^{19} \approx 2^{20}$
  - 总开销大约为 $2 * 2^{20}$，是 O(n) 的量级，平均 O(1)
- 一次高开销操作后，保证有很多次低开销操作，称为“分期付款式”的常量复杂性（平摊式的复杂性）
- Python 的 list 采用这种设计，因此 lst.insert(len(lst), x) 比一般位置加入的效率高，等价写法 lst.append(x)。如合适，应优先使用

list 的操作

- Python list 的实际实现策略
  - 建立空 list 时分配可以容纳 8 个元素的存储区
  - 元素区满时加入：换一块 4 倍大的存储区；但在表已经比较大时就会改变策略，换存储区时规模加倍
  - 效果：通过尾端加入元素，操作的平均复杂性是 O(1)
- 其他操作的性质由连续表的实现方式确定
  - 所有序列的共性操作，复杂性由操作中需要考察的元素个数确定，其中 len(.) 是 O(1) 操作
  - 元素访问和赋值，尾端加入和尾端（切片）删除是 O(1) 操作
  - 一般元素加入，切片替换，切片删除，表拼接（extend）等都是 O(n) 操作。pop 操作默认情况是尾端删除返回，为 O(1)，一般情况（指定非尾端位置）为 O(n)
- Python 没提供检查一个 list 的当前存储块容量的操作

list 的几个特殊操作

- lst.clear() 应该是 O(1) 操作，具体实现情况未见说明。可能做法：
  - 简单将元素计数值设置为 0
  - 换一块空表默认大小的存储区
- lst.reverse() 修改 lst，将其元素倒置。很容易想到下面实现（放在 list 类里，假设元素存储区的域名为 elements），复杂性 O(n)

```
def reverse(self) :
    el = self.elements
    i = 0
    j = len(el)-1
    while i < j :
        el[i], el[j] = el[j], el[i]
        i, j = i+1, j-1
```

- list 的仅有特殊操作（方法）是 sort，完成被操作表的元素排序。有关算法后面讨论。最好的排序算法复杂性是 O(n log n)

所附几页的说明：

- 从概念上说，抽象数据类型是一类数学对象（代数对象），我们可以为一种具体的抽象数据类型建立形式化的数学理论
- 表的数学理论基于（不说明的）元素集合和公共集合（如整数），包括
  - 定义集合（表的集合基于抽象的向量，或称序列）
  - 相关运算
    - 需命名
    - 给定操作的签名（参数个数和类型，操作结果类型）
    - 定义操作的效果
  - 代数定律（公理）
    - 描述操作之间的关系
    - 可以不定义操作的结果，只给出代数定律
- 后面几张幻灯片给出一种 list 的理论

抽象模型（表的代数理论，供参考）

元素集合：　$\mathbb{E}$　　　$e \in \mathbb{E}$

线性表集合：　$\mathit{list}\ \mathbb{E}$　　　$\alpha, \beta \in \mathit{list}\ \mathbb{E}$

$$\alpha = \langle e_0, e_1, \cdots, e_n \rangle$$

基本操作

组合：　$e \oplus \langle e_0, \cdots, e_n \rangle \stackrel{\text{def}}{=} \langle e, e_0, \cdots, e_n \rangle$

取头部：　$hd\ \langle e_0, e_1, \cdots, e_n \rangle \stackrel{\text{def}}{=} e_0$

取尾部：　$tl\ \langle e_0, e_1, \cdots, e_n \rangle \stackrel{\text{def}}{=} \langle e_1, \cdots, e_n \rangle$

判空表：　$empty\ \alpha \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \text{if } \alpha = \langle\rangle \\ \mathsf{false} & \text{otherwise} \end{cases}$

操作的基调（Signature，原型，类型）

组合：　$\oplus : \mathbb{E} \times \mathit{list}\ \mathbb{E} \to \mathit{list}\ \mathbb{E}$

取头部：　$hd : \mathit{list}\ \mathbb{E} \to \mathbb{E}$

取尾部：　$tl : \mathit{list}\ \mathbb{E} \to \mathit{list}\ \mathbb{E}$

判空表：　$empty : \mathit{list}\ \mathbb{E} \to \mathbb{B}$　　$\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$

定义操作，例如表长度操作

基调：　$\# : \mathit{list}\ \mathbb{E} \to \mathbb{N}$

定义：　$\#\,\alpha \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \alpha = \langle\rangle \\ 1 + \#\, tl\ \alpha & \text{if } \alpha \neq \langle\rangle \end{cases}$

其他操作可类似定义

代数定律

设：　$e \in \mathbb{E}, \quad \alpha, \beta \in list\ \mathbb{E}, \quad \beta \neq \langle\rangle$

定律：

$$hd(e \oplus \alpha) = e$$

$$tl(e \oplus \alpha) = \alpha$$

$$(hd\ \beta) \oplus (tl\ \beta) = \beta \qquad \text{if } \beta \neq \langle\rangle$$

$$empty\ \langle\rangle = \textsf{true}$$

$$empty(e \oplus \alpha) = \textsf{false}$$

$$\#\ \langle\rangle = 0$$

$$\#(e \oplus \alpha) = 1 + \#\ \alpha$$

可定义更多操作/研究操作间的关系，做一套表的代数理论。

这种理论可以用于研究程序的构造和性质（这里不再深入）