线性表的实现：顺序表、链表

关振扬

September 14, 2025

本章主要内容

- 线性表的逻辑结构
- 顺序表：存储与实现
- 向量表
- 链表：存储与实现
- 顺序表、向量表与链表的比较
- 其他一些相关的存储方法

线性结构（例子）

学　号	姓　名	性　别	出生日期	政治面貌
0001	陆　宇	男	1986/09/02	团员
0002	李　明	男	1985/12/25	党员
0003	汤晓影	女	1986/03/26	团员
⋮	⋮	⋮	⋮	⋮

(a)　学生学籍登记表

(b)　线性结构

线性表的定义

Definition

线性表，亦简称表，是 n 个句有相同类型的数据元素的有限序列。

线性表的长度定为该表中的元素个数。当长度为 0 时称为空表，写作 $L = (\quad)$；非空表则记为 $L = (a_0, a_1, \ldots, a_{n-1})$。

注意：这边我们采取与书本不同的序号；书的从 1 数起，我们从 0 开始。

线性表的特性

$$a_0 — a_1 — a_2 — a_3 — \ldots — a_{n-1}.$$

1. 有限性：表中的元素个数有穷的。
2. 相同性：数据元素的类型都是一样的。
3. 顺序性：表中有相邻关系，a_i 与 a_{i+1} 相邻，a_i 的后继是 a_{i+1}，a_{i+1} 的前驱是 a_i。a_0 无前驱，a_{n-1} 无后继，其他元素均有前驱与后继。

线性表的 ADT

ADT 类型名: `List`

数据的关系：

线性表的数据元素类型相同，相邻元素有前驱和后继关系。

操作：构造函数、析构函数、长度、是否空表、插入元素、删除元素、查找元素（所在位置的元素、元素的所在位置）

具体操作可按实际应用增减，但上面列举的部分比较常用。

ADT 续：构造函数

`InitList`（在 C++ 实现的时候不是这个名称，构造函数的名称就是类的名称）

- 前置条件：表不存在
- 输入：无
- 功能：表的初始化
- 输出：无
- 后置条件：建一个空表

ADT 续：析构函数

`DestroyList`（在 C++ 实现的时候不是这个名称，构造函数的名称与类的名称有关）

- 前置条件：表已存在
- 输入：无
- 功能：销毁表，释放动态申请的空间
- 输出：无
- 后置条件：已释放所有在这个表的操作申请过的空间

ADT 续：长度

`length`

- 前置条件：表已存在
- 输入：无
- 功能：回传表的长度
- 输出：表的长度
- 后置条件：表没变化

ADT 续：是否空表

`isEmpty`

- 前置条件：表已存在
- 输入：无
- 功能：回传表是否为空表（若是，回传 `true`；否则回传 `false`）
- 输出：如功能
- 后置条件：表没变化

ADT 续：插入元素

`insertEleAtPos`

- 前置条件：表已存在
- 输入：整数 i ，元素 x
- 功能：把元素 x 插入到表的第 i 个位置。如 $L = (a, b, c)$，$i = 1$，则插入结果为 $L = (a, x, b, c)$。
- 输出：若插入失败，抛出异常
- 后置条件：若插入成功，表中的第 i 个元素会是 x。

ADT 续：删除位置 i 的元素

`deleteEleAtPos`

- 前置条件：表已存在
- 输入：整数 i
- 功能：删除位置 i 的元素
- 输出：若删除成功，回传被删除的元素；否则抛出错误
- 后置条件：若删除成功，表中的元素减少一个。

ADT 续：查找所在位置的元素

`getEleAtPos`

- 前置条件：表已存在
- 输入：整数 i
- 功能：回传表的第 i 个元素
- 输出：若 i 为合理位置，则回传该元素；否则抛出错误
- 后置条件：表没变化

ADT 续：查找元素的所在位置

`locateEle`

- 前置条件：表已存在
- 输入：元素 x
- 功能：回传元素 x 在表中首次出现的位置；若不存在，回传 -1。
- 输出：如功能
- 后置条件：表没变化

顺序表的基本实现思路

- 顺序表 (Sequential List) 的存储结构是用一个数列 (array) 存储所有数据，另外用一个整数变量存储表的长度。
- 很明显这个是一个连续存储结构，因此我们不用特别写出前驱后继的关系。
- 问题：具体实现需要数列的长度，根据实现方法可能会有其他 ADT 没写上的限制。(假设这个长度为 1000，那么插入第 1001 个东西的时候就有问题了)
- 可以看为在一维数组增加一堆通用功能。

顺序表的实现：初始化、销毁、长度、是否空表

- 私有变量与上页，设置为一个固有长度的数列以及整数。（这个假设意味着这个表有容量限制）
- 初始化只需把目前长度设为 0。
- 由于没有申请任何空间，因此销毁时什么都不用处理
- 长度查询就是直接回传目前长度的变量。
- 是否空表就是查询目前长度是否为 0。
- 以上均为 $O(1)$ 的时间复杂度

顺序表的实现：插入元素

目标：把元素 x 插入到位置 i。效果如图示：

$$\text{插入前：}(a_0, a_1, \ldots, a_{i-1}, a_i, \ldots, a_{n-1})$$
$$\text{插入后：}(a_0, a_1, \ldots, a_{i-1}, x, a_i, \ldots, a_{n-1})$$

这里我们注意到 a_{i-1} 与 a_i 有前驱后继的关系变化。

这边具体的实现方法就是要先把 $a_i, \ldots, a_{n-1}$ 往后移一格，然后放入元素 x。（当然也要把长度加 1。）这个运行速度跟位置 i 和目前表长有关。

有些 i 是非法的，比如 i 大于长度不行；另外当表满时也不行。

顺序表的实现：删除位置 i 的元素

目标：把第 i 个位置的元素。效果如图示：

$$\text{插入前：}(a_0, a_1, \ldots, a_{i-1}, a_i, a_{i+1} \ldots, a_{n-1})$$
$$\text{插入后：}(a_0, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1})$$

这里我们注意到 a_{i-1} 与 a_{i+1} 有前驱后继的关系变化。

这边具体的实现方法就是要先把 $a_{i+1}, \ldots, a_{n-1}$ 往前移一格。（当然也要把长度减 1。）这个运行速度跟位置 i 和目前表长有关。

有些 i 是非法的，比如 i 大于长度不行；另外，我们无法避免碎片（fragment）的问题：这个往前移的过程，a_{n-1} 的值仍然存在于原来的位置上。

顺序表的实现：查找

- 查找第 i 个位置的元素：当这个位置合理，则回传数据；否则抛错。
- 查找数据的所在位置：这个只能以一个位置找，直到第一次找到这个数据后就马上回传位置。
- 上述的时间复杂度怎样了？

顺序表的优缺点

- 优点：
  1. 不用特别处理元素的逻辑关系
  2. 随机存取：可以快速存取表中任意位置的元素（通常指的是 $O(1)$ 的效率）
- 缺点：
  1. 插入、删除需要移动大量元素
  2. 容量限制
  3. 碎片

向量表 (Vector List)

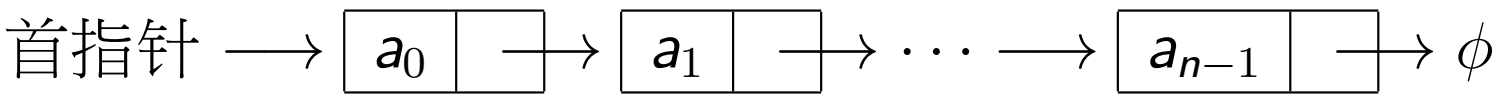
- 基本操作与顺序表一样，保留了除了容量问题的优缺点。
- 数列是动态申请的，初始化时首次申请数列的空间；因此销毁时需要释放空间。
- 需要一个新变量来存储目前表的容量。
- 主要分别为插入的处理：当表满时，申请一个新的数列空间（一般为原容量的双倍），复制所有数据到新数列，然后把旧的数列空间释放。

单链表 (Singly Linked List)

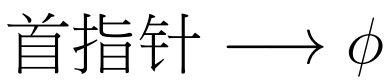
- 由于数据的物理存储不一定按顺序，所以我们要有办法表示数据的关系（前驱、后继）。
- 于是我们对每个数据同时配上若干个指针来达到目的。
- 单链表代表我们就是每个数据配上一个指针（称为结点）：

单链表中的空表与否、遍历

这是一个非空表：



以下是一个空表：



上述的首指针即是指向首个数据的指针。

遍历方法就是不断的读出与结点配上的指针往前走，直到配上的指针为空。

单链表的具体实现：插入

假设我们决定把数据插入到第 0 个位置，以下我们分非空表与空表对这进行分析：

空表：做一个新的结点，其指针指向空白，放入数据，然后首指针指向这个新结点。

非空表：做一个新的结点，其指针指向原本首指针指向的结点，放入数据，然后首指针指向这个新结点。

如果是插入到第 i 个位置（$i \neq 0$，并为合理位置），则过程如下：先查到第 $i-1$ 个数据相关的结点 N，然后做一个新的结点，其指针指向 N 指向的结点，放入数据，然后 N 指向这个新结点。

单链表的实现：小技巧

由前面的描述，我们可以看到不同的情况进行的插入都有不同的处理。

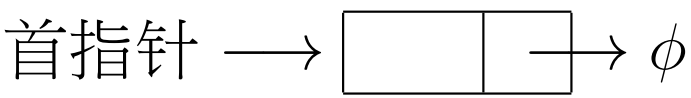
其实这样子太麻烦了。所以如果可以将全部情况统合其他会比较好。不同处理的原因某部分是因为插入点的前驱是否存在。

我们如果可以让所有情况都有前驱、并去掉需要对首指针改变的话，那么就全部降到最后一种情况了。

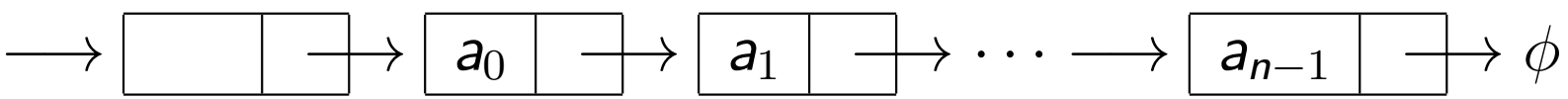
单链表的实现：小技巧（续）

我们利用的方法是一个空的首结点，然后首指针就指向这个空结点。以下为新的空表，以及一般有数据的表：

空表：



有数据的表：



以下我们会以这个表示为基础说明各个操作的实现。

单链表的实现：构造与销毁

- 构造：我们要做一个新的空结点，其指针指向空白(NULL)，首指针指向这个结点。长度设为 0。
- 销毁：我们依次遍历整个表的点，并每个释放其空间：

  当首指针指向的结点非空 (!=NULL) 时，不断执行以下：

  1. 暂存首指针指向的结点至 K。
  2. 首指针指向 K 指向的结点。
  3. 释放结点 K 的空间。

单链表的实现：长度，是否空表

- 是否空表：如有长度的私有变量，则直接检查是否为 0；否则可以检查首指针指向的结点是否在指向 NULL。
- 长度：若有长度的私有变量，则直接回传；否则可遍历计数，过程大概如下：

  设置变量 $count = 0$，以及指针 N 为首指针指向的结点。当 N 指向的结点非空 (!=NULL) 时，不断执行以下：

  1. $count++$。
  2. N 指向其指向结点的下一个结点。

单链表的实现：插入

- 把数据 x 插入到第 i 个位置。
- 遍历到第 $i-1$ 个数据的结点，记为 N。
- 做一个新的结点，把数据放入，把新结点的指针指向 N 指向的结点。
- 把 N 的指针指向那个新结点。

单链表的实现：删除

- 删除第 i 个位置的数据。
- 遍历到第 $i-1$ 个数据的结点，记为 N。
- 用一个结点指针 K 指向 N 指向的结点。
- 把 N 的指针指向 K 指向的下一个结点。
- 暂存 K 的数据，释放 K 的结点空间。
- 回传暂存的数据。

单链表的实现：查找

- 每个结点遍历，直到空白或找到数据为止。
- 时间复杂度？

题外话：算法设计

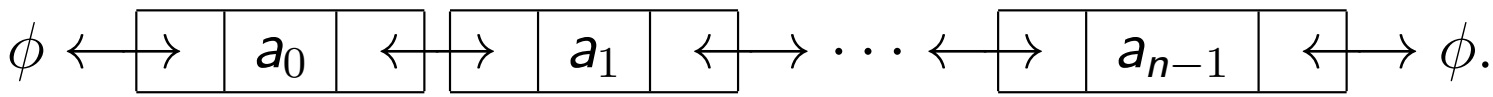
- 设计算法时，先确定要求、输入、输出。
- 写一、两个例子
- 可从正逆方向考虑如何解决问题
- 写出大概思路、分析是否有边界例子（或特殊情况）
- 验证

两者的比较

- 存储结构：连续与链接
- 时间性能：主要分别在查找；在给定指针的情况下插入、删除也有差别。
- 空间性能：结点层面的差别、结构层面的差别
- 前者较适合查找频繁的情况，如果删除、插入频繁则使用后者。

如何提高查找链表结点前驱的方法？

- 比如给定一个结点，在单链表中找出其前驱的方法只有遍历直到找到这个结点为止……
- 但如果我们把结点的结构改变的话，是有可能提高这个相率的。
- 例如：双链表，每个节点配两个指针，分别指向前驱与后继：



- 当然相关插入删除等操作要重新考虑一下。
- 此外还可以把尾的结点的后继指向首数据的结点，并把首数据的前驱指向尾结点，做成双循环链表。

其他一些线性结构存储

- 静态链表：使用数列存取一堆结点，结点之间的指针用数组位置取代。
- 间接寻址：这是对于顺序表的一个改版，生成的数列不直接存数据，而是存一个指向数据的指针。这个改版的好处是插入、删除移动的只是这堆指针，而不是整个数据。(对于每个数据元素比较庞大时，这个确保了移动量很小)。

C++ 模板（例子）

```
template <class T>
T Max(T x, T y){
      return x>=y?x:y;
}
int i=Max(1,2);
double x=Max(1.2,2.5);
```