数据结构

北京邮电大学 信息安全中心
武 斌、杨榆

上章内容

上一章（绪论）内容：

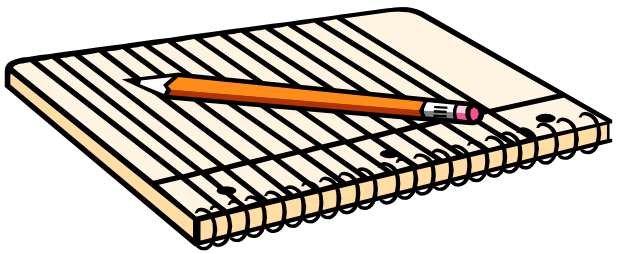
- 理解数据结构
- 知道数据结构的基本概念和术语
- 了解抽象数据类型的表示与实现
- 理解算法和算法分析
- 掌握计算语句频度和估算算法时间复杂度的方法

本次课程学习目标

学习完本次课程，您应该能够：

- 了解线性表的概念及其逻辑结构特性
- 理解顺序存储结构和链式存储结构的描述方法
- 掌握线性表的基本操作及算法实现
- 分析顺序存储结构的时间和空间复杂度

本章课程内容（第二章 线性表）



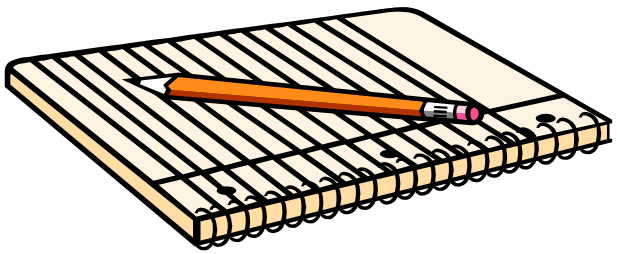
- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和实现
- 2.3 线性表的链式表示与实现
- 2.4 一元多项式的表示及相加

第二章 线性表

- 线性表是一种最常用且最简单的线性结构。
- 什么是线性结构？
  - ➔ 简言之，线性结构是一个数据元素的有序（次序）集合。
  - ➔ 它有四个基本特征：
    1. 集合中必存在唯一的一个“第一元素”；
    2. 集合中必存在唯一的一个“最后元素”；
    3. 除最后元素之外，其它数据元素均有唯一的“后继”；
    4. 除第一元素之外，其它数据元素均有唯一的“前驱”。

- 注意：这里的“有序”仅指在数据元素之间存在一个“领先”或“落后”的次序关系，而非指数据元素“值”的大小可比性。

线性表的类型定义



- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和表现
- 2.3 线性表的链式表示与实现
- 2.4 一元多项式的表示及相加

线性表的类型定义

- **线性表**(Linear List) ： 由n(n≧0)个数据元素(结点)a_1，a_2， …a_n 组成的有限序列。
  - 其中数据元素的个数n定义为**表的长度**。 （有n-1个有序对）
  - 当n=0时，称为**空表**，常常将非空的线性表(n>0)记作：

    (a_1，a_2，…a_n)
  - 这里的数据元素a_i (1≦i≦n)只是一个抽象的符号，其具体含义在不同的情况下可以不同。i为a_i 在线性表中的位序。
- **例1**、26个英文字母组成的字母表是一个线性表

  （A，B，C，…，Z）
- **例2**、同一花色的13张扑克牌可以构成一个线性表

  （2，3，4，…，J，Q，K，A）

线性表的类型定义

- 从以上例子可看出线性表的**逻辑特征**是：
  - 在非空的线性表，**有且仅有一个开始结点**a_1，它没有直接前趋，而仅有一个直接后继a_2；
  - **有且仅有一个终端结点**a_n，它没有直接后继，而仅有一个直接前趋a_{n-1}；
  - 其余的内部结点a_i ($2 \leq i \leq n-1$) 都有且仅有一个直接前趋 a_{i-1} 和一个直接后继 a_{i+1}。

线性表的类型定义

- 线性表是一种典型的线性结构。
  - 数据的运算定义在逻辑结构上，而运算的具体实现则在存储结构上进行。
  - 抽象数据类型线性表的定义如下：

ADT List {

数据对象：D＝{ a_i | a_i ∈ ElemSet， i=1,2,...,n, n≥0 }

数据关系：R＝{ <a_{i-1} ,a_i > | a_{i-1} ,a_i∈D, i=2,...,n }

基本操作：

InitList (&L)　　//{结构初始化}

操作结果：构造一个空的线性表 L 。

DestroyList (&L) //{销毁结构}

初始条件：线性表 L 已存在。

操作结果：销毁线性表 L 。

线性表的类型定义

ListEmpty(L)

初始条件：线性表L已存在。

操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE。

ListLength(L)

初始条件：线性表 L 已存在。

操作结果：返回 L 中元素个数。

PriorElem(L， cur_e，&pre_e)

初始条件：线性表 L 已存在。

操作结果：若cur_e是L中的数据元素，则用pre_e返回它的前驱，否则操作失败，pre_e 无定义。

NextElem(L, cur_e, &next_e)

初始条件：线性表 L 已存在。

操作结果：若cur_e是L中的数据元素，则用next_e返回它的后继，否则操作失败，next_e 无定义。

线性表的类型定义

GetElem(L, i, &e)

初始条件：线性表 L 已存在，1≤i≤LengthList(L)。

操作结果：用 e 返回 L 中第 i 个元素的值。

LocateElem(L, e, compare( ))

初始条件：线性表 L 已存在，compare( ) 是元素判定函数。

操作结果：返回 L 中第1个与 e 满足关系 compare( ) 的元素的位序。若这样的元素不存在，则返回值为0。

ListTraverse(L, visit( ))

初始条件：线性表 L 已存在，visit( ) 为元素的访问函数。

操作结果：依次对 L 的每个元素调用函数 visit( )。一旦 visit( ) 失败，则操作失败。

线性表的类型定义

- 容易看出以上7个操作的结果都没有改变线性表中的数据元素和数据元素之间的关系，因此它们都是“**引用型**”的操作，函数中的“L”是“传值参数”。

- 而以下4个操作的结果或修改表中的数据元素，或修改元素之间的关系，被称为“**加工型**”的操作，为了便于返回操作的结果，用“引用” 传递参数“ L ”，即在参数 L 之前加有符号“ & ”。

线性表的类型定义

//{加工型操作}

ClearList(&L)

初始条件：线性表 L 已存在。

操作结果：将 L 重置为空表。

PutElem(&L, i, &e)

初始条件：线性表L已存在，1≤i≤LengthList(L)。

操作结果：L 中第 i 个元素赋值同 e 的值。

ListInsert(&L, i, e)

初始条件：线性表 L 已存在，1≤i≤LengthList(L)+1。

操作结果：在 L 的第 i 个元素之前插入新的元素 e，L 的长度增1。

ListDelete(&L, i, &e)

初始条件：线性表 L 已存在且非空，1≤i≤LengthList(L)。

操作结果：删除 L 的第 i 个元素，并用 e 返回其值，L 的长度减1。

} ADT List

线性表类型的应用

- 上述各操作定义仅对抽象的线性表而言，还无法在程序设计中直接加以引用，进行一些更复杂的操作，比如，合并两个线性表等。
- 但可以利用这些操作进行算法的研究，并由此判断上述操作对线性表类型的定义是否“完整”，即：能否利用线性表的上述操作实现应用问题的算法设计。

线性表类型的应用

●例2-1： 已知集合 A 和 B，求两个集合的并集，使 A＝A∪B。

➔分析 从集合的观点看，此问题求解的方法很简单，只要对集合 B 中的所有元素一个一个地检查，看看在集合 A 中是否存在相同元素，若不存在，则将该元素插入到集合 A，否则舍弃之。

➔现假设以线性表 LA 和 LB 分别表示集合 A 和 B，即构造两个线性表 LA 和 LB，它们的数据元素分别为集合 A 和 B 中的成员。

➔由此，上述集合求并的问题便可演绎为对线性表作如下操作：扩大线性表 LA，将存在于线性表 LB 中而不存在于线性表 LA 中的数据元素插入到线性表 LA 中去。

线性表类型的应用

- 具体操作步骤为：
  - 从线性表 LB 中取出一个数据元素；
  - 依值在线性表 LA 中进行查询；
  - 若不存在，则将它插入到 LA 中。
- 容易看出，上述的每一步恰好对应线性表的一个基本操作：
  - GetElem (LB，i，e)；
  - LocateElem(LA， e， equal())；
  - ListInsert(&LA，n+1，e)
- 由此，得到求并集的算法如下页算法2.1所示。

线性表类型的应用

- 算法2.1　Algo2-1.c

```
void union ( List &LA, List  LB )
{  // 将所有在线性表LB中但不在LA中的数据元素插入到 LA 中，
   // 算法执行之后，线性表 LB 不再存在。
    La_len = ListLength(LA); // 求得线性表LA的长度
    Lb_len = ListLength(LB); // 求得线性表LB的长度
    for(i=1;i<=Lb_len;i++)    // 依次处理LB中元素直至LB为空
    {
       GetElem(LB,i,e); //从LB中拿到第i个数据元素并赋给 e
       // 当LA中不存在和 e 值相同的数据元素时进行插入
       if (!LocateElem(LA,e,equal( ))
             ListInsert(LA,++La_len,e);
     } //for
} // union
```

- 算法2.1的时间复杂度为**O(ListLength(LA)*ListLength(LB))**
  - 假设GetElem、ListInsert操作的执行时间与表长无关
  - LocateElem的执行时间与表长成正比

线性表类型的应用

- **例2-2** 已知线性表LA和线性表LB中的数据元素按值非递减有序排列，现要求将LA和LB归并为一个新的线性表LC，且LC中的元素仍按值非递减有序排列。**（课堂）**

- **分析** LC中的数据元素是LA或LB中的数据元素，则先设LC为空表，然后将LA或LB中的元素逐个插入到LC中即可。为使LC中元素按值非递减有序排列，设两个指针i和j分别指向LA和LB中某个元素，若设i当前所指的元素为a，j当前所指的元素为b，则当前应插入到LC中的元素c为

$$c=\begin{cases} a, & \text{当} a \le b \text{时} \\ b, & \text{当} a > b \text{时} \end{cases}$$

- 上述归并算法如下页**算法2.2**所示。

线性表类型的应用

- 算法2.2 Algo2-2.c

```
void  MergeList ( List La,List Lb,List &Lc )
{
   InitList(Lc);
   i=j=1; k=0;
   La_len=ListLength(La);
   Lb_len=ListLength(Lb);
   while ((i<=La_len)&&(j<=Lb_len)) { // La和Lb均非空
      GetElem(La,i,ai);  GetElem (Lb , j, bj);
      if (ai<bj) { ListInsert(Lc,++k,ai); ++i; }
      else if (ai=bj) { ListInsert(Lc,++k,ai); ++i;  ++j; }
      else { ListInsert(Lc,++k,bj); ++j; }
   }
   while (i<= La_len) {
      GetElem ((La, i++, ai); ListInsert(Lc, ++k, ai);  }
   while (j<= Lb_len) {
      GetElem ((Lb, j++, bj); ListInsert(Lc, ++k, bi);  }
} // MergeList
```

A: ...3,5,15...　　C: ...3,5,8,14,15...
B: ....5,8,14...

- 算法2.2的时间复杂度为O(ListLength(LA)＋ListLength(LB))
  - 后面两个while只会执行一个

线性表类型的应用

- **例2-3** 已知一个“非纯集合”B，试构造一个集合 A，使 A 中只包含 B 中所有值各不相同的数据元素。

- **分析** 此问题即为从 B 中挑选出所有“彼此相异”的元素构成一个新的集合。如何区分元素的“相异”或“相同”，一个简单方法即为将每个从 B 中取出的元素和已经加入到集合 A 中的元素相比较。应对线性表作和例2-1相同的操作，具体的三步也都相同 。

  所不同之处仅仅在于两点：一是例2-1的算法中 LA 是已知的，而在此例算法中的 LA 是待新建的；二是例2-1在求得并集之后，原来的两个集合不再保留，而在此例中构建新的集合 A 的同时，原来的集合 B 不变。

- 具体算法如下页**算法**所示。

线性表类型的应用

●**算法2.3**

```
void purge(List &LA, List LB)
{   //构造线性表LA，使其只包含LB中所有值不相同的数据
    // 元素，算法不改变线性表LB
     InitList(LA);                          // 创建一个空的线性表 LA
     La_len = 0;
     Lb_len = ListLength(LB);               // 求线性表 LB 的长度
     for (i = 1；i <= Lb_len；i++)    // 依次处理 LB 中每个元素
     {
          GetElem(LB，i，e);          // 取 LB 中第 i 个数据元素赋给 e
          if (!LocateElem( LA，e，equal( ) )
               ListInsert( LA，++La_len， e );
                    // 当 LA 中不存在和 e 值相同的数据元素时进行插入
     } // for
} // purge
```

思考：若B为有序表的情况？

线性表类型的应用

- **算法2.3（B为非递减或非递增的有序表）**

```
void purge(List &LA, List LB)
{   //构造线性表LA，使其只包含LB中所有值不相同的数据
    // 元素，算法不改变线性表LB
     InitList(LA);                          // 创建一个空的线性表 LA
     La_len = 0;
     Lb_len = ListLength(LB);               // 求线性表 LB 的长度
     GetElem(Lb, 1, e1);e1--;               // 初始化最近一次插入LA中元素
     for (i = 1；i <= Lb_len；i++)          // 依次处理 LB 中每个元素
     {
          GetElem(LB，i，e2);               // 取 LB 中第 i 个数据元素赋给 e2
          if (! equal( e1,e2) ) {
               ListInsert( LA，++La_len， e2)；e1=e2}
                    // 当e2与上一次插入LA的元素e1不同时进行插入
    } // for
} // purge
```

线性表类型的应用

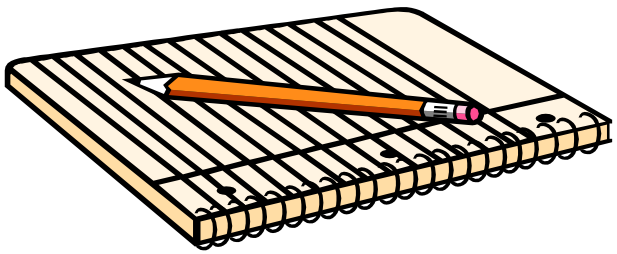
这个算法思想还有一个前提是，已知集合符合集合论中的约定"集合中的元素都是彼此相异的"。

- **例2-4** 判别两个集合是否相等。
  - 两个集合相等的**充分必要条件**是它们具有相同的元素。当以线性表表示集合时，两个线性表的长度应该相等，且表中所有数据元素都能一一对应，但相同的数据元素在各自的线性表中的“位序”不一定相同。
  - 由此，“判别两个线性表中的数据元素是否完全相同”的算法的基本思想为：
    - 首先判别两者的表长是否相等；
    - 在表长相等的前提下，如果对于一个表中的所有元素，都能在另一个表中找到和它相等的元素的话，便可得到“两个线性表表示的集合相等”的结论；
    - 反之，只要有一个元素在另一个表中不能找到相等元素时，便可得出"不等"的结论。

线性表类型的应用

```
bool  isEqual(List LA, List LB)
 {  La_len = Listlength(LA);
    Lb_len = Listlength(LB);
    if ( La_len != Lb_len ) return FALSE; // 两表的长度不等
   else {     i = 1; found = TRUE;
      while ( i<= La_len && found )
      {  GetElem( LA, i, e );                  // 取得 LA 中一个元素
       if ( LocateElem(LB, e, equal( ) )
              i++;                              // 依次处理下一个
        else found = FALSE;                     // LB中没有和该元素相同
                                                   的元素
      } // while
      return found;
   } // else
} // isEqual
```

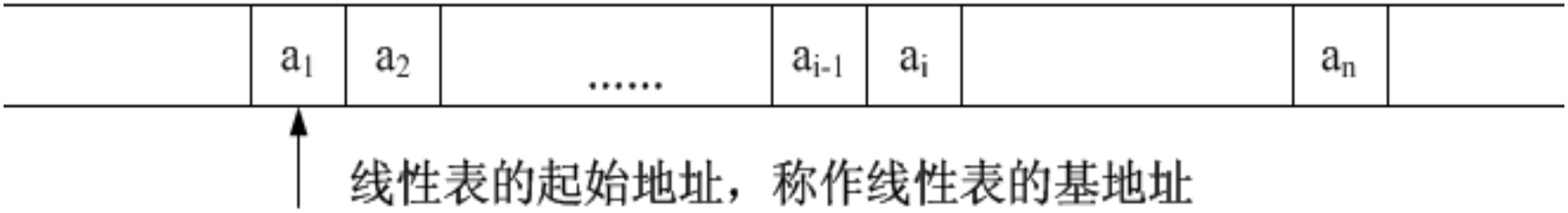
线性表的顺序表示和表现



- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和表现
- 2.3 线性表的链式表示与实现
- 2.4 一元多项式的表示及相加

顺序表

- 顺序表是线性表的顺序存储表示的简称，它指的是，"用一组地址连续的存储单元依次存放线性表中的数据元素"，即以"存储位置相邻"表示"位序相继的两个数据元素之间的前驱和后继的关系(有序对< ，>)"，并以表中第一个元素的存储位置作为线性表的起始地址，称作线性表的基地址。如下图所示。



顺序表

- 不失一般性，假设每个数据元素占据的存储量是一个常量 C，则后继元素的存储地址和其前驱元素相隔一个常量，即：$LOC(a_i) = LOC(a_{i-1}) + C$

  ↑一个数据元素所占存储量

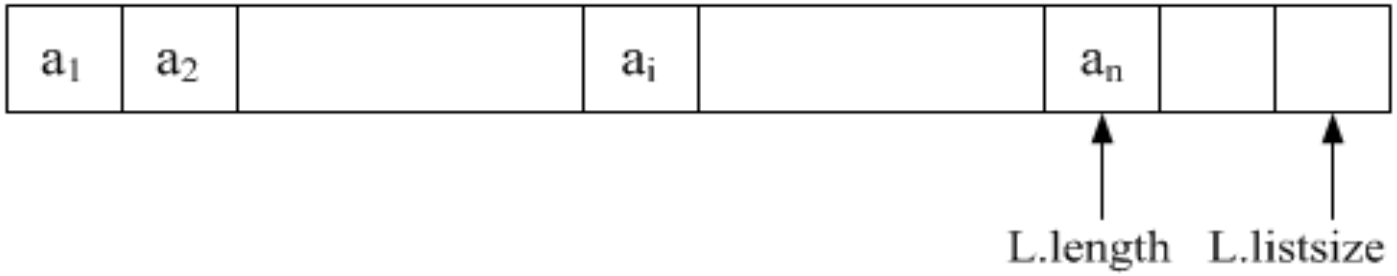
- 由此，所有数据元素的存储位置均可由第一个数据元素的存储位置得到

  $LOC(a_i) = LOC(a_1) + (i-1) \times C$

  ↑基地址

顺序表

- 由于线性表的长度是**可变**的，因此对顺序表的定义除了需要一个存储元素的**一维数组空间**以外，还需要**两个数据成员**：其中一个指示顺序表中**已有的元素个数**，另一个指示该顺序表允许存放的数据**元素个数的最大值**，如下图所示。ElemType 为元素类型。



顺序表

●用C/C++语言描述的顺序表类型如下所示：

```
// 存储结构
  const int LIST_INIT_SIZE=100;   // 线性表存储空间初始分配量
  const int LISTINCREMENT=10;   // 线性表存储空间分配增量
    typedef struct {
        ElemType *elem;          // 存储空间基址
        int length;              // 当前长度
        int listsize;            // 允许的最大存储容量
                                 // (以sizeof(ElemType)为单位)
    } SqList;                    //  顺序表
```

顺序表中基本操作的实现

- 从顺序表的存储结构定义容易看出，由于顺序表的“长度”是个“显值”，且由于第i个元素恰好存储在数组的第 i 个分量(数组下标为 i-1)中，因此其“求长”、“判空”以及“存取第 i 个数据元素”等操作都很容易实现。下面重点讨论顺序表类型定义中五个操作的实现。
  - 一、初始化操作
  - 二、元素定位操作（课堂）
  - 三、插入元素操作（课堂）
  - 四、删除元素操作（课堂）
  - 五、销毁结构操作

顺序表中基本操作的实现

- 一、初始化操作
- 算法2.4

```
Status InitList(SqList &L)
{       // 构造一个空的线性表 L
        L.elem =  (Elemtype *)malloc(LIST_INIT_SIZE *
    sizeof(Elemtype));
        if (!L.elem)    exit((OVERFLOW);          // 存储分配失败
         L.length = 0;                    // 顺序表的初始长度为0
         L.listsize = LIST_INIT_SIZE;   //初始存储容量
         return OK;
} // InitList
```

- 此算法的时间复杂度为O(1)

顺序表中基本操作的实现

- 二、元素定位操作（课堂）

在顺序表中“查询”是否存在一个和给定值满足判定条件的元素的最简单的办法是，依次取出结构中的每个元素和给定值进行比较。

- 演示2-2-1.swf

顺序表中基本操作的实现

- 算法2.5　**int** LocateElem(SqList L, ElemType e，**Status** (*compare)(ElemType， ElemType))

```
{   // 在顺序表L中查找第1个值与 e 满足判定条件compare( )的元素，
    // 若找到，则返回其在 L 中的位序，否则返回0。
    i = 1;                          // i 的初值为第1元素的位序
    p = L.elem;                     // p 的初值为第1元素的存储位置
    while (i <= L.length && !(*compare)(*p++，e)) // *p++等同*(p++)
        ++i;                        // 依次进行判定
    if (i <= L.length) return i;  // 找到满足判定条件的数据元素为第 i 个元素
        else return 0;            // 该线性表中不存在满足判定的数据元素
} // LocateElem
```

- 此算法的时间复杂度为O(ListLength(L))
  - 算法中的基本操作是“判定”，它出现在 while 循环中，而函数 compare() 的时间复杂度显然是个常量。因此执行判定的次数取决于元素在线性表中的“位序”，至多和表长相同。

顺序表中基本操作的实现

- 算法2.5 //其他实现方法，时间复杂度相同

```
int LocateElm (SqList L, ElemType e, Status (* compare)(ElemType, ElemType){
  for(i = 0; i < L.length; i++){
    if compare(*(L.elem + i),e) break; // *(L.elem+i)等同L.elem[i]
  }
  if i == L.length return 0;
  else return i+1;
}
```

顺序表中基本操作的实现

- 三、插入元素操作（课堂）
  - 首先分析，“插入元素”使线性表的逻辑结构发生什么变化？
  - 假设在线性表的第i个元素之前插入一个元素e，使得线性表 $(a_1, \cdots, a_{i-1}, a_i, \cdots, a_n)$ 改变为 $(a_1, \cdots, a_{i-1}, e, a_i, \cdots, a_n)$ 即：
    - (1) 改变了表中元素之间的关系，使 $\langle a_{i-1}, a_i \rangle$ 改变为 $\langle a_{i-1}, e \rangle$ 和 $\langle e, a_i \rangle$
    - (2) 表长增1
  - 由于顺序表是以“存储位置相邻” 表示“元素之间的前驱和后继关系”，则必须“移动元素”来体现“元素之间发生的变化”。
- 演示2-2-2.swf

顺序表中基本操作的实现

●三、插入元素操作（课堂）

➔ 元素$a_i, \cdots, a_n$每个均后移一个位置

1. 应该是$a_{k+1}=a_k$，k的取值范围n降序到i
2. i=1时，循环n次；
3. i=n时，循环1次；
4. i=n+1时，循环0次；

➔ 新元素e放入位序i处，表长增1

顺序表中基本操作的实现

- 算法2.6

```
Status ListInsert(SqList &L，int pos， ElemType e)
{  // 新元素位序合法值1≤pos≤Listlength(L)+1，在第i个位置前插新元素e
   if (pos < 1 || pos > L.length+1)  return ERROR;// 位置不合法
   if(L.length  >= L.listsize){// 顺序表已满，不能容纳新元素，需扩展有序表容量
      newbase = (ElemType *)realloc(L.elem,(L.listsize + LISTINCREMENT)
      * sizeof(ElemType)); // 每次扩容增加LISTINCREMENT个元素空间
      if (!newbase)      exit(OVERFLOW); // 分配失败
      L.elem = newbase;   L.listsize += LISTINCREMENT;}
   for (j=L.length-1；  j>=pos-1；--j) // 位序[1,n]对应数组下标为[0,n-1]
      L.elem[j+1] = L.elem[j]；// 插入位置及之后的元素右移
   L.elem[pos-1] = e；                  // 插入 e
   ++L.length；                         // 表长增1
   return OK；
} // ListInsert
```

- **此算法的时间复杂度为 O(ListLength(L))**
- 演示2-2-3.swf

算法中的基本操作是“移动元素”。 pos=1时，插入在第一个元素之前。此为最坏情况，for 循环执行次数为 L.length。

顺序表中基本操作的实现

- 四、**删除元素操作（课堂）**
  - 同样首先分析，“删除元素”使线性表的**逻辑结构**发生什么变化？
  - 假设删除线性表中第i个元素，使得线性表($a_1,\ldots, a_{i-1}, a_i, a_{i+1},\ldots, a_n$) 改变为 ($a_1,\ldots, a_{i-1}, a_{i+1},\ldots, a_n$) 即：
    - (1) 改变了表中元素之间的关系，使$< a_{i-1}, a_i >$和$< a_i, a_{i+1} >$改变为 $< a_{i-1}, a_{i+1} >$
    - (2) 表长减1
  - 对顺序表而言，需要改变从第 i+1 个元素起到第 n 个元素的存储位置，即进行“**从第 i+1 到第 n 个元素往前移动一个位置**”。
  - 演示2-2-4.swf

顺序表中基本操作的实现

- 四、**删除元素操作（课堂）**
  - ➔ 元素$a_{i+1}, \cdots, a_n$每个均向前移一个位置
  1. 应该是$a_k=a_{k+1}$，k的取值范围i升序到n-1
  2. 或者，应该是$a_{k-1}=a_k$，k的取值范围i+1升序到n
  3. i=1时，循环n-1次；
  4. i=n时，循环0次；
  - ➔ 表长减1

顺序表中基本操作的实现

- 算法2.7 Status ListDelete(SqList &L， int pos， ElemType &e)

```
{       // 若1≤pos≤Listlength(L)，则以 e 带回从顺序表 L 中删除的
        // 第 pos 个元素且返回 OK，否则返回 错误值
        if ((pos < 1) || (pos > L.length))
                return ERROR ；                          // 删除位置不合法
        e =L.elem[pos-1];
        for (j = pos； j<L.length；++j)
           L.elem[j-1] = L.elem[j]；                     // 被删除元素之后的元素左移
        --L.length；                                     // 表长减1
        return OK；
} // ListDelete
```

- 此算法的时间复杂度为 **O (ListLength(L))**
  - 算法中的基本操作是“移动元素”，for循环的执行次数在最坏的情况下(pos=1即删除第一个元素时)为L.length-1。
- 演示2-2-5.swf

顺序表中基本操作的实现

- 五、销毁结构操作

和“初始化”操作分配空间相对应，销毁结构的实质是释放它所占的全部空间，以便使存储空间得到充分的利用。

顺序表中基本操作的实现

- 算法2.8

```
void DestroyList( SqList &L )
{
        // 释放顺序表 L 所占存储空间
        free(L.elem)；
        L.listsize = 0；
        L.length = 0；
} // DestroyList_Sq
```

- 此算法的**时间复杂度为 O(1)**

插入和删除操作的性能分析

●插入和删除操作的性能分析

- 在顺序表中任何一个合法位置上进行“一次”插入或删除操作时，需要移动的元素个数的平均值是多少？
- 令$E_{in}(n)$表示在长度为 n 的顺序表中进行一次插入操作时所需进行“移动”元素个数的期望值（即平均移动个数），则

$$E_{in} = \sum_{i=1}^{n+1} p_i \cdot (n-i+1)$$

- 其中，pi是在第 i 个元素之前插入一个元素的概率，n-i+1 是在第 i 个元素之前插入一个元素时所需移动的元素个数。由于可能插入的位置 i=1,2,3,…,n+1 共 n+1 个，假设在每个位置上进行插入的机会均等，则

$$p_i = \frac{1}{n+1}$$

插入和删除操作的性能分析

- 由此，在上述等概率假设的情况下，进行一次插入操作时需要移动的元素个数的平均值为

$$E_{in} = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{1}{n+1} \times \frac{n(n+1)}{2} = \frac{n}{2}$$

插入和删除操作的性能分析

➔同理，令$E_{dl}(n)$表示在长度为 n 的顺序表中进行一次**删除操作**时所需进行“移动”元素个数的期望值（即平均移动个数），则

$$E_{dl}=\sum_{i=1}^{n}q_i(n-i)$$

其中，q_i是删除第 i 个元素的概率，n-i 是删除第 i 个元素时所需移动元素的个数。同样假设在 n 个可能进行删除的位置 i=1,2,…,n 机会均等，则

$$q_i=\frac{1}{n}$$

- 由此，在上述等概率假设的情况下，进行一次**删除**操作时**需要移动的元素个数的平均值**为

$$E_{dl}=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{1}{n}\times\frac{n(n-1)}{2}=\frac{n-1}{2}$$

插入和删除操作的性能分析

- 由此可见，在**顺序存储**表示的线性表中**插入**或**删除**一个数据元素，**平均约需移动表中一半元素**。这个数目在线性表的长度较大时是很可观的。
- 这个**缺陷**完全是由于顺序存储要求线性表的元素**依次紧挨存放**造成的。
- 因此，这种顺序存储表示**仅适用于不常进行插入和删除操作、表中元素相对稳定的线性表**。

顺序表其它算法举例

- 例2-4 试编写算法“比较”两个顺序表的大小。
  - 首先，何谓顺序表的“大”和“小”？现作如下规定：
    - 设 A=(a_1,…,a_m)和 B=(b_1,…,b_n)均为顺序表
    - A’ 和 B’ 分别为 A 和 B中除去最大共同前缀后的子表
      - （例如，A=(y,x,x,z,x,z)，B=(y,x,x,z,y,x,x,z)，则两者中最大的共同前缀为(y,x,x,z)，
      - 在两表中除去最大共同前缀后的子表分别为A’=(x,z)和B’=(y,x,x,z)）。
    - 若 A’=B’=空表，则 A=B；若 A’=空表，而 B’≠空表，或者两者均不为空表，且 A‘ 的首元小于 B’ 的首元，则 A<B；否则 A>B。

顺序表其它算法举例

- 算法要求对两个顺序表进行“比较”，是一种“引用型”操作，因此在算法中不应该破坏已知表。
- 按注解中的规定，只有在两个表的长度相等，且每个对应元素都相同时才相等；否则两个顺序表的大小主要取决于两表中除去最大公共前缀后的第一个元素。
- 算法的基本思想为：若 $a_j = b_j$，则 j 增 1，之后继续比较后继元素；否则即可得出比较结果。显然，j 的初值应为 0，循环的条件是 j 不超出其中任何一个表的范围。若在循环内不能得出比较结果，则循环结束时有三种可能出现的情况需要区分。
- 具体算法如下页算法2.9所示。

顺序表其它算法举例

❊算法2.9

```
int compare( SqList A，SqList B )
{       // 若 A<B，则返回 -1；若 A=B，则返回 0；若 A>B，则返回 1
        j=0；
        while ( j<A.length && j<B.length )
           if ( A.elem[j] < B.elem[j] ) return(-1)；
           else if ( A.elem[j] > B.elem[j] ) return(1)；
           else j++；
// 至此，有3种情况，(1)A,B中所有元素相等(2)A.length>B.length (3)A.length<B.length
        if ( A.length == B.length ) return (0)；
        else if ( A.length < B.length ) return(-1)；
      else return(1)；
} // compare
```

✸ 此算法的时间复杂度为 ***O* (Min(A.length, B.length))**。

顺序表其它算法举例

- 例2-5　试设计一个算法，用尽可能少的辅助空间将顺序表中前 m 个元素和后 n 个元素进行互换，即将线性表(a_1, a_2, …, a_m, b_1, b_2, …, b_n) 改变成(b_1, b_2, …, b_n, a_1, a_2, …, a_m)。（课堂）

- 此题的一种比较简单的算法是，从表中第 m+1 个元素起依次插入到元素 a_1 之前。则首先需将该元素 b_k (k=1，2，…，n)暂存在一个辅助变量中，然后将它之前的 m 个元素(a_1, a_2, …, a_m)依次后移一个位置。

- 演示2-2-6.swf

✸ 显然，由于对每一个 b_k 都需要移动 m 个元素，因此算法的时间复杂度为 $O(m \times n)$。

顺序表其它算法举例

- 可采用另一种算法为，对顺序表进行三次“逆置”，第一次是对整个顺序表进行逆置，之后分别对前 n 个和后 m 个元素进行逆置。

- 演示2-2-7.swf

- 具体算法如下页算法2.11所示。

顺序表其它算法举例

❊算法2.10

```
void invert( ElemType &R[]， int s，int t )
{       // 本算法将数顺序表R 中下标自 s 到 t 的元素逆置，即将
        // （Rs，Rs+1，…，Rt-1，Rt）改变为（Rt，Rt-1，…，Rs+1，Rs）
        for ( k=s； k<=(s+t)/2； k++ )
        {
           w = R[k]；
           R[k] = R[t-k+s]；
           R[t-k+s] = w；
        } // for
} // invert
```

顺序表其它算法举例

❊算法2.11

```
void exchange ( SqList &A，int m )
{       // 本算法实现顺序表中前 m 个元素和后 n 个元素的互换
        if ( m > 0 && m < A.length ){
           n = A.length – m；
           invert( A.elem，0，m+n-1 )；
           invert( A.elem，0，n-1 )；
           invert( A.elem，n，m+n-1 )；
        }
} // exchange
```

✸ 由于逆置顺序表可以利“交换”相应元素进行，其时间复杂度为线性级别，则三次调用逆置算法完成的操作的时间复杂度仍然是线性级别的，即为***O* (m+n)**。

本章（上）小结

- 在这一章，介绍了**线性表**的**抽象数据类型**的定义以及它的**顺序存储结构**的实现。
- 线性表是 n(n≥0) 个数据元素的序列，通常写成$(a_1, a_2, \ldots, a_n)$；
- 线性表中除了**第一个**和**最后一个**元素之外都只有一个**前驱**和一个**后继**。线性表中每个元素都有自己**确定的位置**，即“**位序**”，因此，线性表可以看成是由 n 个 (i, a_i) 构成的集合。
- **n=0**时的线性表称为“**空表**”，它是线性表的一种**特殊状态**，因此，在写线性表的操作算法时一定要考虑算法对空表的情况是否也正确。

本章（上）小结（续）

- **顺序表**是线性表的顺序存储结构的一种别称。它的**特点**是以“**存储位置相邻**”表示两个元素之间的**前驱后继关系**。
- 顺序表的**优点**是可以**随机存取**表中任意一个元素。
- **缺点**是每作一次插入或删除操作时，**平均**来说必须**移动表中一半元素**。
- 常应用于主要是为**查询**而很少作**插入和删除**操作，表长变化不大的线性表。

本章（上）知识点与重点

- **知识点**

  线性表、顺序表

- **重点和难点**

  顺序表的表示与实现、顺序表各类操作的时间复杂度等。

本章（上）作业

- 1. 设顺序表va中的数据元素递增有序。试写一算法，将x插入到顺序表的适当位置上，且保持表的有序性。
- 2. 假设以两个元素值非递减有序排列的线性表A和B分别表示两个集合，在同一表（A或B）中可能存在值相同的元素，求A和B的交集，存放在A表空间中，要求新生成的交集A表中的元素值各不相同且按非递减有序排列。
- 3. 编写c程序，实现顺序表中的基本操作，并进行测试。