第二章
线性表

※ 教学内容：

线性表的逻辑结构；线性表的存储结构及操作的实现；一元多项式的表示

※ 教学重点：

线性表的逻辑结构；线性表的存储结构；线性表在顺序结构和链式结构上实现基本操作的方法；从时间和空间复杂度的角度比较线性表两种存储结构的不同特点及其适用场合

※ 教学难点：

链式结构上实现基本操作的方法。

线性表是一种最简单的**线性结构**

线性结构是n 个数据元素的**有序(次序)集合**

线性结构的基本特征为:

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. 除最后元素在外，均有唯一的后继；
4. 除第一元素之外，均有唯一的前驱。

2.1 线性表的基本概念

2.2 线性表的顺序表示和实现

2.3 线性表的链式表示和实现

2.4 一元多项式的表示

2.1 线性表的基本概念

一、线性表的逻辑结构

线性表是n个数据元素$a_1, a_2, \ldots, a_n$的有限序列，记为$(a_1, a_2, \ldots a_i, \ldots, a_n)$，

其中，a_i可以是一个数、一个字母、一串字符、一页书，甚至更复杂的信息。

例如：英文字母表（A, B, C, ..., Z）

工资变化情况表

（800，1500，2400，4100，...，20000）

在稍复杂的线性表中，一个数据元素可以由若干个**数据项组成**，如学生情况登记表

学号	姓名	性别	年龄	班级
1	王小林	男	18	计科0601
2	陈红	女	20	计科0601
3	刘建平	男	21	计科0601
4	张立立	男	17	计科0601
……	……	……	……	……

在这种情况下，常把数据元素称为**记录**，含有大量记录的线性表称为**文件**。

综上所述：线性表中的数据元素可以是各种各样的，但同一线性表中的元素必定具有**相同的特性**，即属同一数据对象，相邻数据元素之间存在着序偶关系。

若将线性表记为（$a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$），则

a_{i-1}是a_i的直接前趋；a_{i+1}是a_i的直接后继；

当i=1, 2, …, n-1时，a_i有且仅有一个直接后继；

当i=2, 3，…, n时，a_i有且仅有一个直接前趋；

即：第一个元素无前趋；最后一个元素无后继；

其它元素仅有一个直接前趋和一个直接后继。

线性表中元素的个数n（$n \geqslant 0$）定义为线性表的长度；

n=0时，称为空表；

在非空线性表中的每个数据元素a_i都有一个确定的位置，称i为数据元素a_i在线性表中的位序。

二、抽象数据类型线性表的定义

ADT List {

数据对象:

$D = \{ a_i \mid a_i \in ElemSet, i=1,2,...,n, n \geq 0 \}$

数据关系:

$R1 = \{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n \}$

基本操作:

结构初始化操作

结构销毁操作

引用型操作

加工型操作

} ADT List

初始化操作

InitList(&L)

操作结果： 构造一个空的线性表L。

结构销毁操作

DestroyList(&L)

初始条件： 线性表 L 已存在。

操作结果： 销毁线性表 L。

引用型操作：

ListEmpty(L)

ListLength(L)

PriorElem(L, cur_e, &pre_e)

NextElem(L, cur_e, &next_e)

GetElem(L, i, &e)

LocateElem(L, e, compare( ))

ListTraverse(L, visit( ))

ListEmpty(L)　（线性表判空）

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FALSE。

ListLength(L)　（求线性表的长度）

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)

（求数据元素的前驱）

初始条件： 线性表L已存在。

操作结果： 若cur_e是L的数据元素，但**不是第一个**，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)

（求数据元素的后继）

初始条件： 线性表L已存在。

操作结果： 若cur_e是L的数据元素，但**不是最后一个**，则用next_e返回它的后继，否则操作失败，next_e无定义。

GetElem(L, i, &e)

（求线性表中某个数据元素）

初始条件： 线性表**L**已存在，
且 **1≤i≤ListLength (L)**。

操作结果： 用 e 返回**L**中第 i 个元素的值。

LocateElem(L, e, compare( ))

（定位函数）

初始条件： 线性表**L**已存在，e为给定值，compare( )是数据元素判定函数。

操作结果： 返回**L**中**第1个与e满足关系** compare( )的数据元素的**位序**。若这样的数据元素不存在，则返回值为0。

ListTraverse(L, visit( ))

（遍历线性表）

初始条件： 线性表**L**已存在，
Visit() 为某个访问函数。

操作结果： **依次**对**L**的每个数据元素调用函数visit( )。一旦visit( )失败，则操作失败。

加工型操作

ClearList(&L)

PutElem(&L, i, e)

ListInsert(&L, i, e)

ListDelete(&L, i, &e)

ClearList(&L)

（线性表置空）

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)

（改变数据元素的值）

初始条件：线性表L已存在，
且 1≤i≤ListLength (L) 。

操作结果：L中第i个元素赋值同e的值。

ListInsert(&L, i, e)

（插入数据元素）

初始条件：线性表L已存在，
且 1≤i≤ListLength (L)+1 。

操作结果：在L的第i个数据元素之前插入
新的数据元素e，L的长度增1。

ListDelete(&L, i, &e)

（删除数据元素）

初始条件：　线性表L已存在且非空，
1≤i≤ListLength (L) 。

操作结果：　删除L的第i个数据元素，并
用e返回其值，L的长度减1。

线性表是一个相当灵活的数据结构，它的长度可依据需要进行增减，对数据元素不仅可以访问，还可以进行插入和删除等操作。运算的具体实现一般依赖所采用的存储结构，所以，这里给出的只是定义在逻辑结构上的抽象运算，即只关心这些运算是“做什么”的，至于“怎样实现”则依赖于所选定的存储结构。

对上述定义的抽象数据类型线性表，还可以进行一些更复杂的运算，如：

- 将两个或两个以上的线性表合并成一个线性表；
- 把一个线性表拆分成两个或两个以上的线性表；
- 重新复制一个线性表；
- 对线性表中的数据元素按某个数据项递增或递减的顺序进行重新排序，从而得到有序表。

这些运算均可利用上述基本运算来实现。

利用上述定义的**线性表**

可以实现其它更复杂的操作

例 2-1

例 2-2

例 2-3

例 2-1

假设:有两个集合 A 和 B 分别用两个线性表 LA 和 LB 表示，即：线性表中的数据元素即为集合中的成员。

现要求一个新的集合A = A ∪ B。

上述问题可演绎为

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。

操作步骤：

1.从线性表LB中依次察看每个数据元素；

GetElem(LB, i)→e

2.依值在线性表LA中进行查访；

LocateElem(LA, e, equal( ))

3.若不存在，则插入之。

ListInsert(LA, n+1, e)

```
void union(List &La, List Lb) {
    La_len = ListLength(La);     // 求线性表的长度
    Lb_len = ListLength(Lb);
    for (i = 1;  i <= Lb_len;  i++) {
        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);
            // La中不存在和 e 相同的数据元素，则插入之
    }
} // union
```

例 2-2

已知一个非纯集合(集合中有相同的元素) B，试构造一个纯集合(集合中没有相同的元素) A，使A中只包含B中所有值各不相 同的数据元素。

仍选用线性表表示集合。

集合 B

集合 A

从集合 B 取出物件放入集合 A

要求集合A中同样物件不能有两件以上

因此，算法的策略应该和例2-1相同

```
void union(List &La, List Lb) {
  InitList(La);   // 构造(空的)线性表LA
  La_len=ListLength(La);   Lb_len=ListLength(Lb);
  for (i = 1;  i <= Lb_len;  i++) {
    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);
          // La中不存在和 e 相同的数据元素，则插入之
  }
} // union
```

试改变结构，以**有序表**表示集合。

若线性表中的数据元素相互之间可以比较，并且数据元素在线性表中**依值非递减或非递增有序**排列，即

$$a_i \geqslant a_{i-1} \text{ 或 } a_i \leqslant a_{i-1} (i = 2, 3, \ldots, n)$$

则称该线性表为**有序表**(Ordered List)。

例如：

（2，3，3，5，6，6，6，8，12）

对集合 B 而言，

值相同的数据元素必定相邻；

对集合 A 而言，

数据元素依值从小至大的顺序插入。

```
void purge(List &La, List Lb) {
   InitList(La);    La_len = ListLength(La);
   Lb_len =ListLength(Lb);                    // 求线性表的长度
   for (i = 1;  i <= Lb_len;  i++) {
     GetElem(Lb, i, e); // 取Lb中第i个数据元素赋给 e
      // en为表La中当前最后一个元素
     if (ListEmpty(La) || !equal (en, e)) {
           ListInsert(La, ++La_len, e);
           en = e;
     }      // La中不存在和 e 相同的数据元素，则插入之
}
} // purge
```

例 2-3

已知线性表LA和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中的数据元素仍按值非递减有序排列。

例如，设i，j，k分别指向三个线性表中元素的指针

LA=(3,5,8,11)

↑ i

LB=(2,6,8,9,11,15,20)

↑ j

则 LC=(2, 3, 5,6,8,8,9,11,11,15,20)

↑ k

$$C=\begin{cases} a & \text{当} a<=b \\ b & \text{当} a>b \end{cases}$$

```
Void  MergeList (List La , List Lb, List &Lc)
  {   InitList(Lc);
      i=j=1; k=0;                                          //初始化
      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   {ListInsert(Lc,++k,bj); ++ j} ;
      }
      while (i<=La_len)              // Lb表空, 插入 La 表中剩余元素
          {GetElem(La, i++,ai); ListInsert(Lc,++k,ai)} ;
      while (j<=Lb_len)              // La 表空, 插入 Lb 表中剩余元素
          {GetElem(Lb, j++,bj); ListInsert(Lc,++k,bj) } ;
  } // MergeList
```

2.2 线性表的顺序表示和实现

一、线性表的顺序存储结构——顺序映象

二、线性表的顺序存贮结构示意图及描述

三、线性表的基本操作在顺序表中的实现

四、顺序存储结构的优缺点

一、线性表的顺序存储结构

——顺序映象

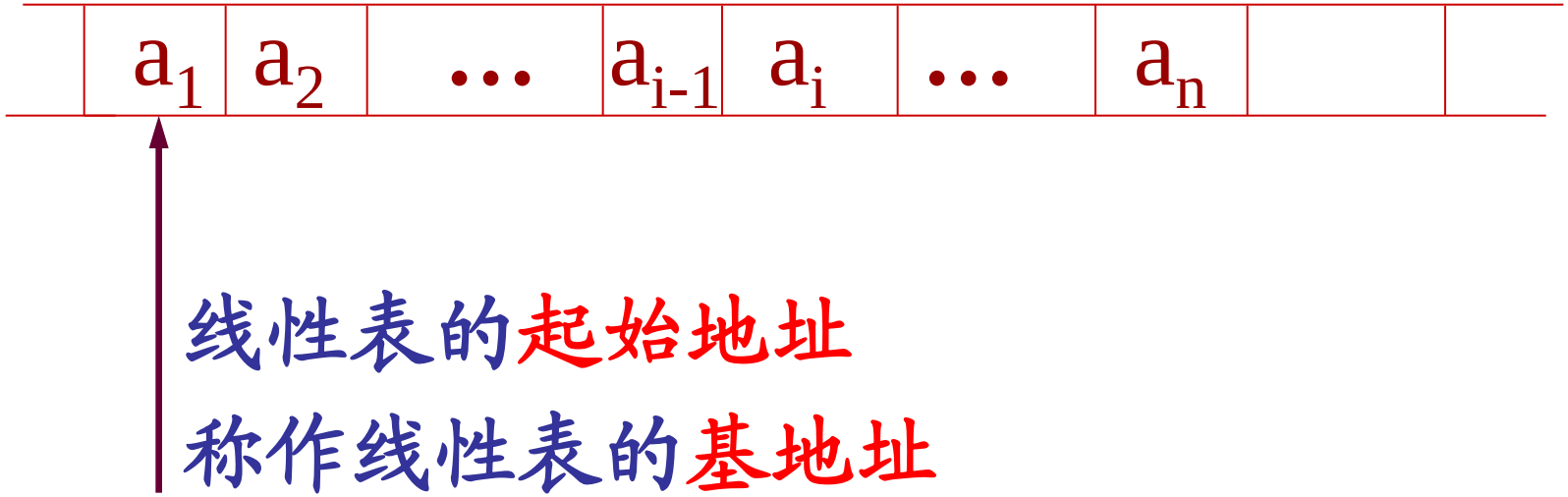
以 x 的存储位置和 y 的存储位置之间某种关系表示逻辑关系<x,y>。

最简单的一种顺序映象方法是:

令y的存储位置和x的存储位置相邻。

顺序存储结构

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。



以“存储位置相邻”表示有序对$<a_{i-1}, a_i>$

假设线性表的每个数据元素要占L个存储单元，则

$$LOC(a_{i+1}) = LOC(a_i) + L$$

所有数据元素的存储位置均取决于第一个数据元素的存储位置，即

$$LOC(a_i) = LOC(a_1) + (i-1)*L$$

↑

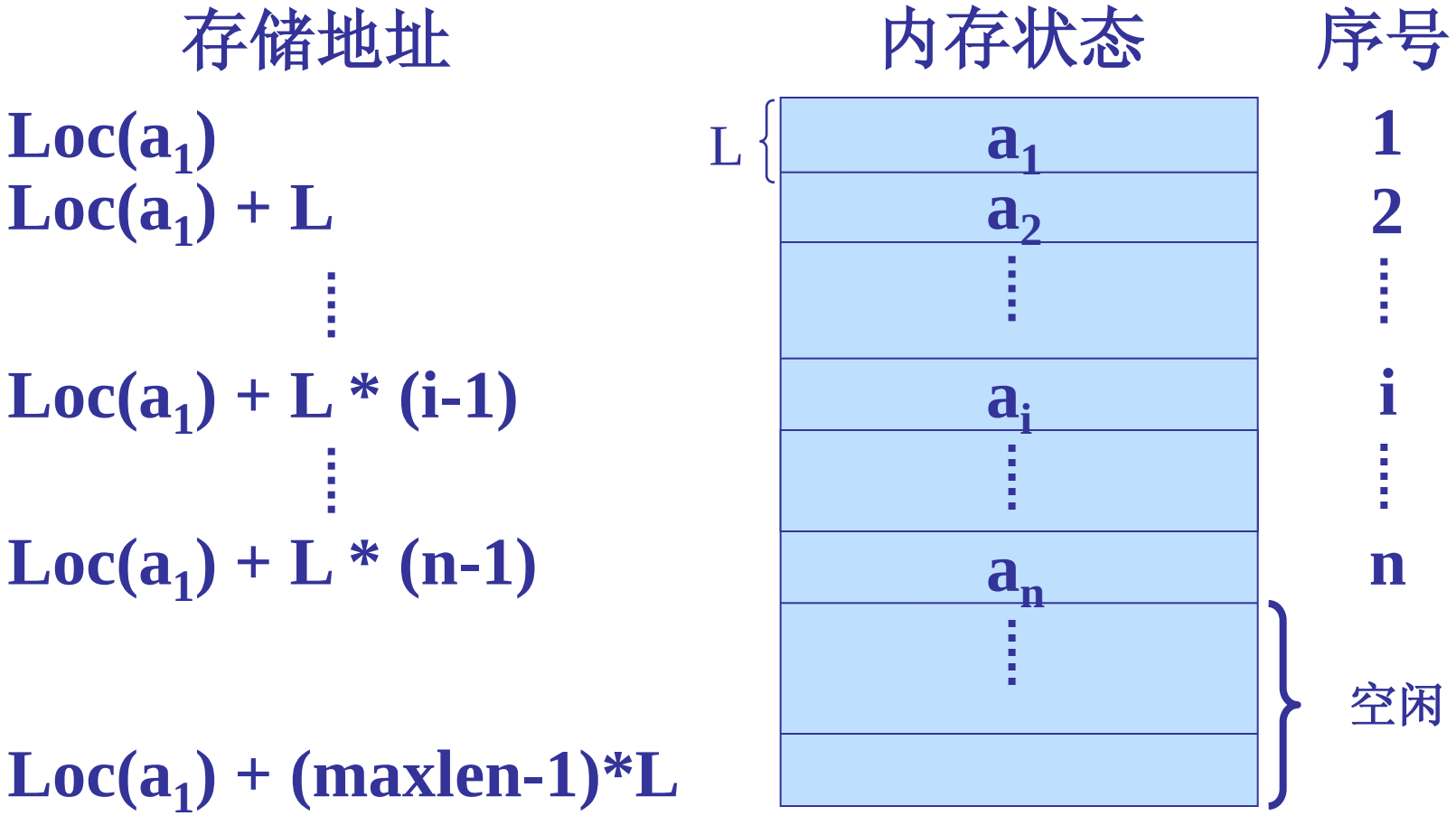
起始地址（基地址）

顺序存储结构的特点

表中相邻的两个元素其物理存储位置也相邻。即以元素在计算机内物理位置上的相邻来表示线性表中数据元素之间相邻的逻辑关系。

每个数据元素的存储位置和线性表的起始位置相差一个和数据元素在线性表中的序号成正比的常数；只要确定了起始位置，线性表中任一数据元素都可随机存取。顺序存储结构是一种随机存取的存储结构。

二、线性表的顺序存贮结构示意图及描述



L是每一个数据元素占用的存储单元

顺序映像的 C 语言描述

通常用数组来描述顺序存储结构，由于线性表的长度可变，所以必须动态分配一维数组。

```
#define  LIST_INIT_SIZE     80
                          // 线性表存储空间的初始分配量
#define  LISTINCREMENT    10
                          // 线性表存储空间的分配增量
 typedef  struct {
     ElemType *elem;          // 存储空间基址
     int      length;         //当前长度
     int      listsize;       // 当前分配的存储容量
                              // (以sizeof(ElemType)为单位)
 } SqList;   //俗称顺序表
```

三、线性表的基本操作在顺序表中的实现

InitList(&L) // 结构初始化

LocateElem(L, e, compare()) // 查找

ListInsert(&L, i, e) // 插入元素

ListDelete(&L, i) // 删除元素

结构初始化—InitList(&L) 的实现

```
Status InitList_Sq( SqList& L ) {
                    // 构造一个空的线性表

    L.elem = (ElemType*) malloc (LIST_
                    INIT_SIZE*sizeof (ElemType));
    if (!L.elem) exit(OVERFLOW);      // 存储分配失败

    L.length = 0;                              // 空表长度为零

    L.listsize = LIST_INIT_SIZE                // 初始存储容量

    return OK;

} // InitList_Sq
```

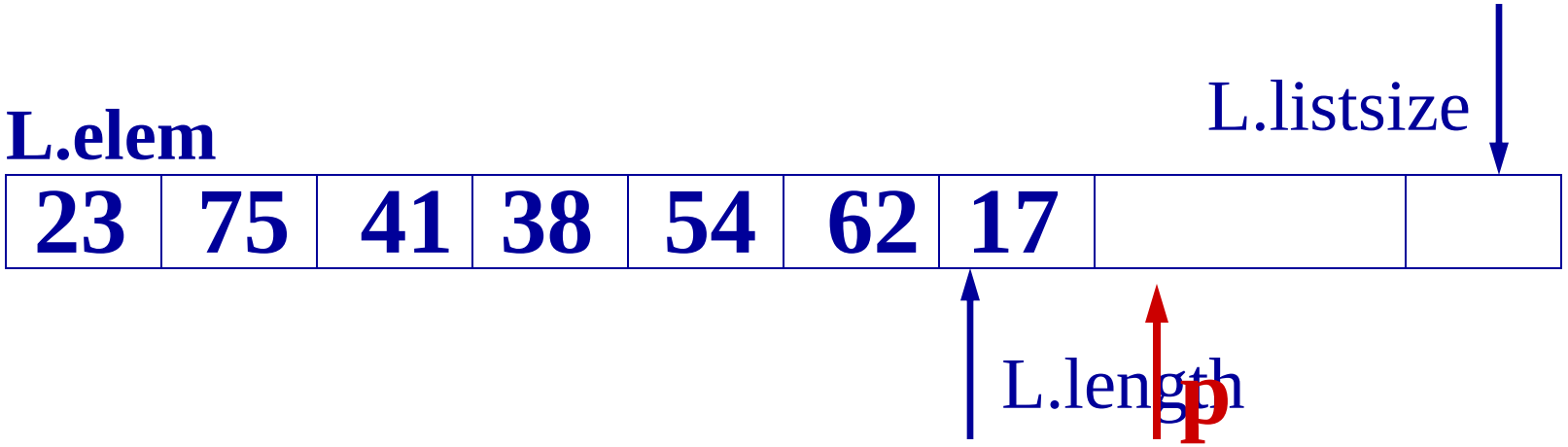
算法时间复杂度：O(1)

扩展线性表L的容量—ExtendList_Sq的实现

```
Status ExtendList_Sq(SqList &L)
    //扩展线性表L的容量

    newbase=(ElemType *)realloc(L.elem,
                    (L.listsize+ LISTINCREMENT) *
                    sizeof(ElemType));
    if (!newbase) exit(OVERFLOW);  //  存储分配失败

   L.elem=newbase;
   L.listsize += LISTINCREMENT
   return OK;

} // ExtendList_Sq
```

算法时间复杂度：O(1)

查找顺序表—LocateElem(L, e, compare()) 的实现



i 8

e = 38 50

可见，基本操作是：将顺序表中的元素逐个和给定值 e 相比较。

```
int LocateElem_Sq(SqList L, ElemType e,
  Status (*compare)(ElemType, ElemType)) {
      // 在顺序表中查询第一个满足判定条件的数据元素，
      // 若存在，则返回它的位序，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))  ++i;
    if (i <= L.length)  return i;
       else  return 0;
} // LocateElem_Sq
```

算法的时间复杂度为：

O(ListLength(L))

线性表插入操作

ListInsert(&L, i, e)的实现

首先分析:

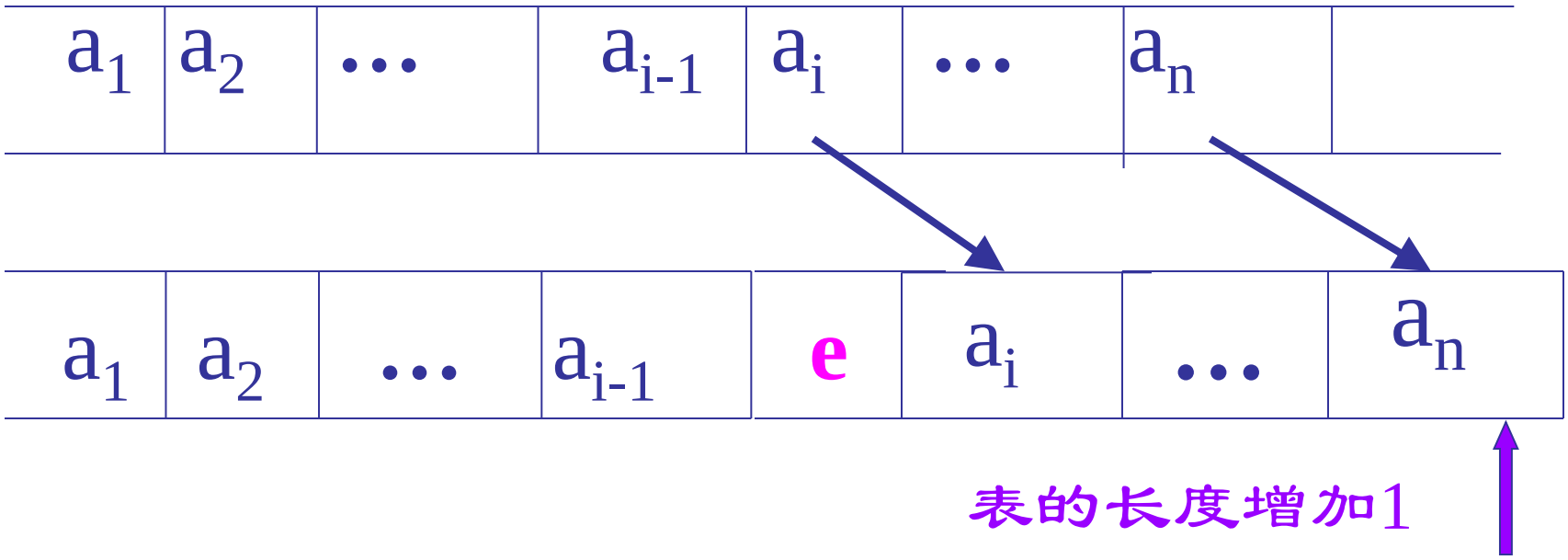
插入元素时，

线性表的逻辑结构发生什么变化?

$(a_1, \ldots, \mathbf{a_{i-1}}, \mathbf{a_i}, \ldots, a_n)$ 改变为

$(a_1, \ldots, \mathbf{a_{i-1}}, \mathbf{e}, \mathbf{a_i}, \ldots, a_n)$

$\mathbf{<a_{i-1}, a_i>}$ → $\mathbf{<a_{i-1}, e>}$, $\mathbf{<e, a_i>}$



```
Status ListInsert_Sq(SqList &L, int i, ElemType e) {
    // 在顺序表L的第 i 个元素之前插入新的元素e,
    // i 的合法范围为 1≤i≤L.length+1

    q = &(L.elem[i-1]);                  // q 指示插入位置
    for (p = &(L.elem[L.length-1]); p >= q; --p)
        *(p+1) = *p;                     // 插入位置及之后的元素右移
    *q = e;                              // 插入e
    ++L.length;                          // 表长增1
    return OK;
} // ListInsert_Sq
```

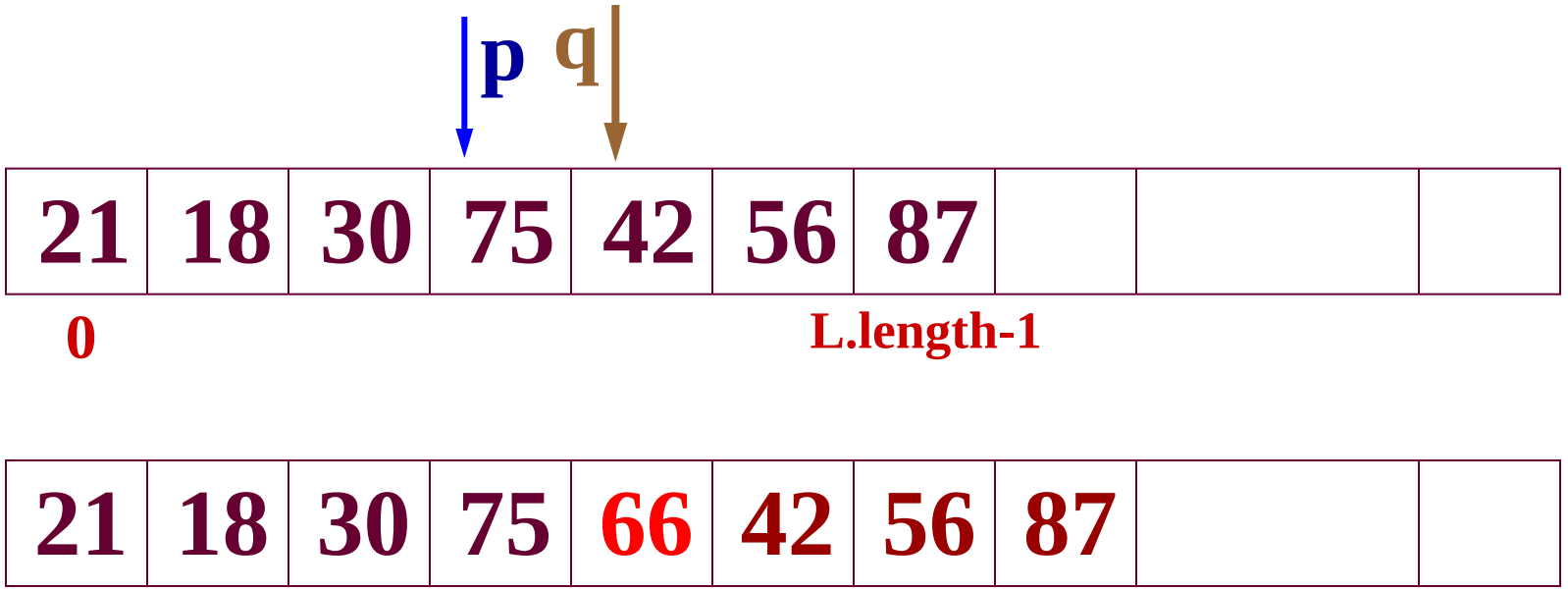
算法时间复杂度为：

O(ListLength(L))

注意：C语言中数组的下标从0开始，线性表L中第i个元素是L.elem[i-1]

例如：ListInsert_Sq(L, 5, 66)

```
q = &(L.elem[i-1]);      // q 指示插入位置
for (p = &(L.elem[L.length-1]); p >= q;  --p)
    *(p+1) = *p;
```



```
Status ListInsert_Sq(SqList &L, int i, ElemType e) {
    // 在顺序表L的第 i 个元素之前插入新的元素e,
    // i 的合法范围为 1≤i≤L.length+1
    ......
    q = &(L.elem[i-1]);                  // q 指示插入位置
    for (p = &(L.elem[L.length-1]); p >= q; --p)
        *(p+1) = *p;        // 插入位置及之后的元素右移
    *q = e;                  // 插入e
    ++L.length;              // 表长增1
    return OK;
} // ListInsert_Sq
```

```
Status ListInsert_Sq(SqList &L, int i, ElemType e) {
    // 在顺序表L的第 i 个元素之前插入新的元素e,
    // i 的合法范围为 1≤i≤L.length+1

    q = &(L.elem[i-1]);                  // q 指示插入位置
    for (p = &(L.elem[L.length-1]); p >= q; --p)
        *(p+1) = *p;          // 插入位置及之后的元素右移
    *q = e;                   // 插入e
    ++L.length;               // 表长增1
    return OK;
} // ListInsert_Sq
```

最坏情况下

算法时间复杂度为：

O(ListLength(L))

考虑移动元素的平均情况：

假设在第 i 个元素之前插入的概率为 p_i，则在长度为n 的线性表中**插入一个元素所需移动元素次数的期望值(平均次数）**为：

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

若**假定**在线性表中任何一个位置上进行**插入的概率**都是**相等**的，$p_i = \frac{1}{n+1}$

则**移动元素的期望值**为：

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

线性表删除操作

ListDelete(&L, i, &e)的实现

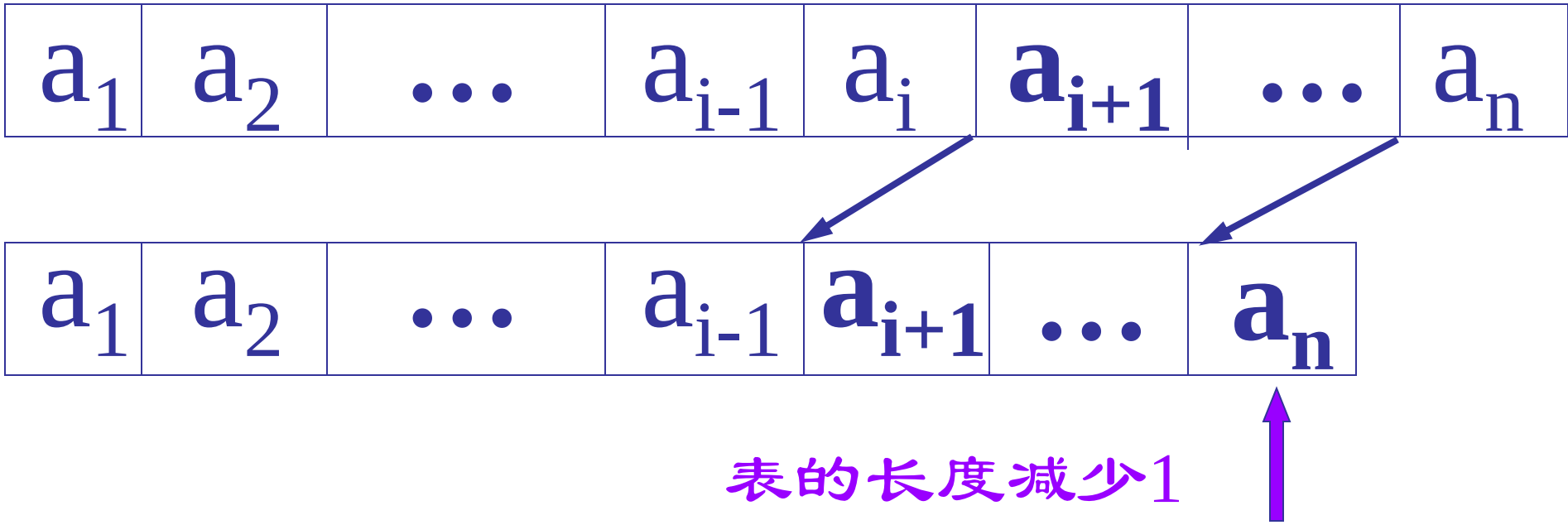
首先分析:

删除元素时，
线性表的逻辑结构发生什么变化?

$(a_1, \ldots, \mathbf{a_{i-1}, a_i, a_{i+1}}, \ldots, a_n)$ 改变为

$(a_1, \ldots, \mathbf{a_{i-1}, a_{i+1}}, \ldots, a_n)$

$\mathbf{<a_{i-1}, a_i>, <a_i, a_{i+1}>}$ ⟶ $\mathbf{<a_{i-1}, a_{i+1}>}$



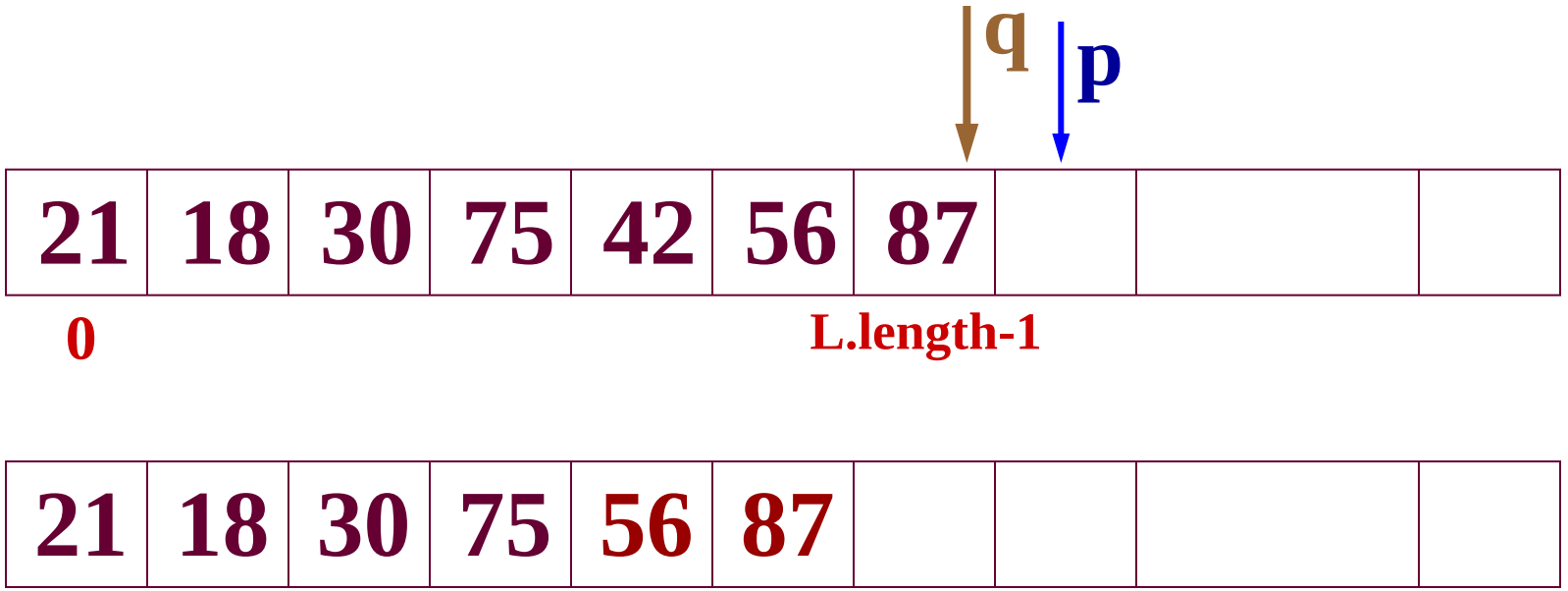
```
Status ListDelete_Sq
                    (SqList &L, int i, ElemType &e) {
    if ((i < 1) || (i > L.length))  return ERROR;
                                          // 删除位置不合法
    p = &(L.elem[i-1]);                  // p 为被删除元素的位置
    e = *p;                              // 被删除元素的值赋给 e
    q = L.elem+L.length-1;               // 表尾元素的位置
    for (++p; p <= q; ++p)  *(p-1) = *p;
                                         // 被删除元素之后的元素左移
    --L.length;                          // 表长减1
    return OK;
} // ListDelete_Sq
```

算法时间复杂度为：

O(ListLength(L))

例如：ListDelete_Sq(L, 5, e)

```
p = &(L.elem[i-1]);
q = L.elem+L.length-1;
for (++p; p <= q; ++p)  *(p-1) = *p;
```



考虑移动元素的平均情况：

假设删除第 i 个元素的概率为 q_i，

则在长度为n 的线性表中删除一个元素所需

移动元素次数的期望值(平均次数）为：

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

若假定在线性表中任何一个位置上进行删除的概率都是相等的，　$p_i = \frac{1}{n}$

则**移动元素的期望值**为：

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

在顺序存储结构的线性表中插入或删除一个数据元素时，其时间主要耗费在移动元素上，移动次数不仅依赖于线性表的长度L. length，还依赖于元素插入或删除的位置i。

当i=1时，全部元素参加移动，移动次数为 n（插入）或 n-1（删除）；

当i= L.length +1（插入）或L.length （删除）时，不需移动元素。平均约移动线性表中一半元素。

算法的时间复杂度为O(n)。

2.1 节中例2-3在以顺序存储结构表示时的实现方法

已知线性表LA和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中的数据元素仍按值非递减有序排列。

例如，设

LA=(3,5,8,11)

↑ i

LB=(2,6,8,9,11,15,20)

↑ j

则 LC=(2, 3, 5,6,8,8,9,11,11,15,20)

↑ k

```
Void  MergeList (List La , List Lb, List &Lc)
  {   InitList(Lc);
      i=j=1; k=0;                                   //初始化
      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   {ListInsert(Lc,++k,bj); ++ j} ;
      while (i<=La_len)                  // 插入 La 表中剩余元素
          {GetElem(La, i++,ai); ListInsert(Lc,++k,ai)} ;
      while (j<=Lb_len)                  // 插入 Lb 表中剩余元素
          {GetElem(Lb, j++,bj); ListInsert(Lc,++k,bj} ;
  } // MergeList
```

对于“顺序表的合并”，A、B的所有元素都放到C中。从上述算法可直接写出形式上极其相似的算法如下。

```
Void  MergeList-sq (SqList La , SqList Lb,
SqList &Lc) {
    //已知顺序线性表La和Lb的元素按值非递减排列
    //归并La和Lb得到新的顺序表Lc，Lc的元素也按值非递减排列
  pa=La.elem; pb=Lb.elem; // pa, pb指示La表和Lb表第一个元素
  Lc.listsize=Lc.length=la.length+Lb.length;
  pc=Lc.elem=(ElemType*)malloc(Lc.listsize*
sizeof(ElemType))                    //为Lc表申请分配存储空间
  if  (!Lc.elem) exit (OVERFLOW);    //存储分配失败
```

```
    pa_last=La.elem+La.length-1;
                    // pa_last指示La表最后一个元素
    pb_last=Lb.elem+Lb.length-1;
                    // pb_last指示Lb表最后一个元素
   while (pa<=pa_last && pb<=pb_last)    //La与 Lb非空
     {  if (*pa<=*pb) *pc++=*pa++;
        else *pc++=*pb++ ;}
   while (pa<=pa_last)
          *pc++=*pa++ ;              // 插入 La 表中剩余元素
   while (pb<=pb_last)
          *pc++=*pb++ ;              // 插入 Lb 表中剩余元素
} // MergeList_sq
```

算法的时间复杂度为：

O(La.length+Lb.length)

四、顺序存储结构的优缺点

(1) 无需为表示数据元素之间的关系而增加额外存储空间，存储密度高；

(2) 可以随机存取表中任一元素，它的存储位置可用一个简单直观的公式来表示；

(3) 插入和删除运算时，必须移动大量元素，效率较低；

(4) 必须预先为线性表分配连续空间。难以准确估计线性表最大长度，估计过小导致溢出，估计过大又会造成存储空间浪费。

2.3 线性表的链式表示和实现

一、单链表

二、结点和单链表的 C 语言描述

三、线性表的操作在单链表中的实现

四、一个带头结点的单链表类型

五、其它形式的链表

六、有序表类型

一、单链表

线性表的链式存储结构的特点：用一组任意的存储单元存储线性表中的数据元素（这组存储单元可以是连续的，也可以是不连续的）。

为了表示每个元素a_i与其直接后继元素a_{i+1}之间的逻辑关系，一个结点包括两部分，数据域data存放元素a_i，指针域next存放一个指针，它指向直接后继元素a_{i+1}所在的结点。这两部分信息组成数据元素ai 的存储映象，称为结点。

以元素(数据元素的映象)

+ 指针(指示后继元素存储位置)

= 结点

n 个结点（a_i(1 ≤ i ≤ n)的存储映象）通过指针链结成一个链表称作单链表

线性表

（ZHAO,QIAN,SUN,LI,ZHOU,WU,ZHENG,WANG)

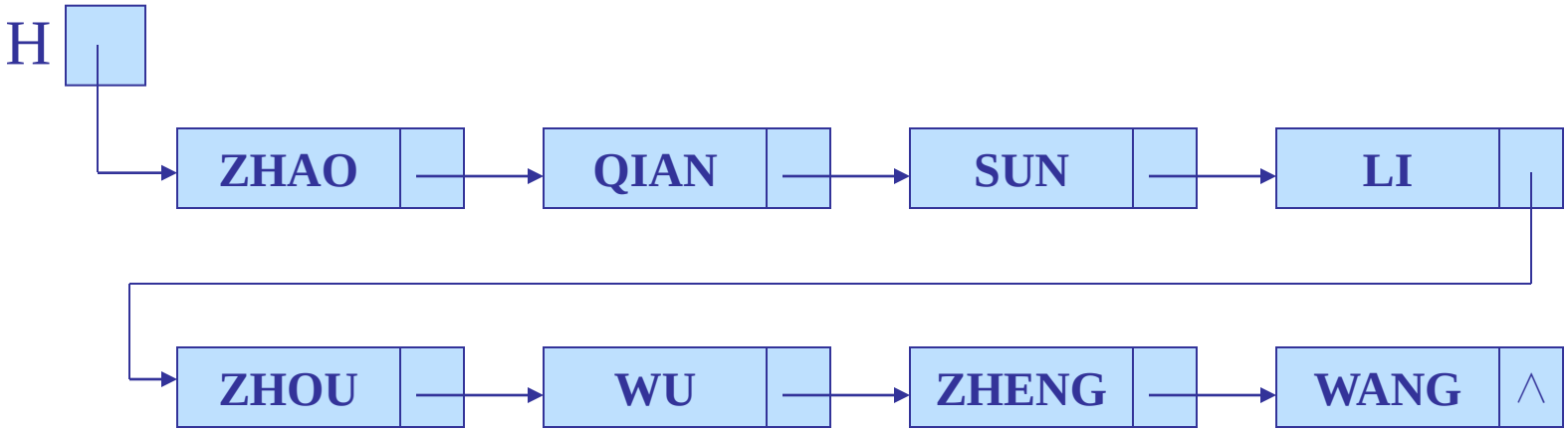
的线性链表存储结构：

头指针
H(head)
31

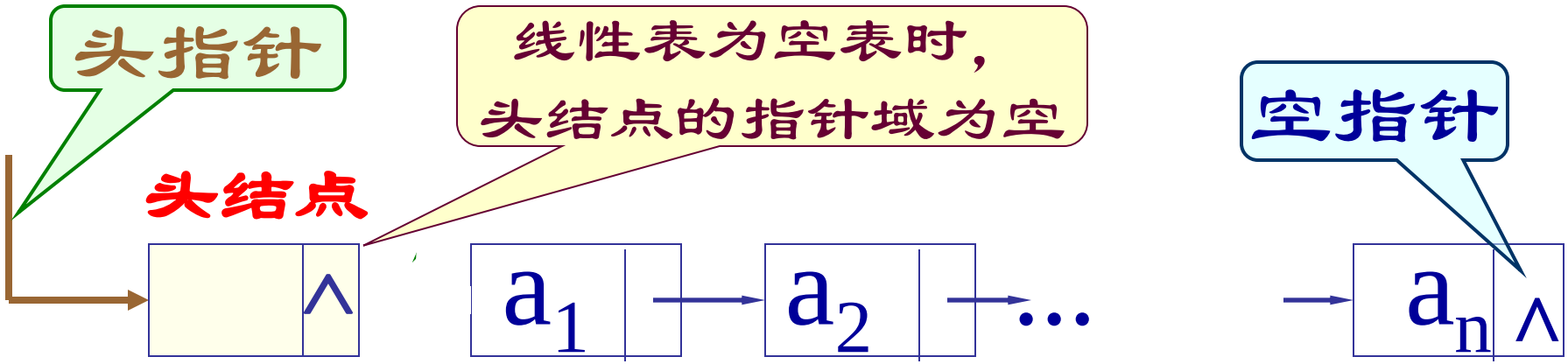
存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	Null
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25

头指针指示链表的第一个结点的存储位置，由于最后一个元素没有直接后继，所以链表最后一个节点的指针为空。

线性链表的逻辑状态



用线性链表表示线性表时，数据元素之间的逻辑关系是由结点中的指针来指示的，逻辑上相邻的两元素其物理位置不要求紧邻。通常把链表画成用箭头相连接的结点的序列，结点之间的箭头表示链域中的指针。由此可见，**单链表可由头指针唯一确定**。



以线性表中第一个数据元素 a_1的存储地址作为线性表的地址，称作线性表的**头指针**。当线性表为空时，头指针为空。

有时为了操作方便，在第一个结点之前虚加一个“**头结点**”，头结点的指针域指向第一个结点的存储位置；以**指向头结点的指针**为链表的**头指针**。

二、结点和单链表的 C 语言描述

```
Typedef struct  LNode {
    ElemType      data;      // 数据域
    struct Lnode   *next;  // 指针域
 } LNode, *LinkList;    （结构指针）

LinkList  L;       // L 为单链表的头指针
```

三、单链表操作的实现

GetElem(L, i, &e)　　// 取第i个数据元素

ListInsert(&L, i, e)　//插入数据元素

ListDelete(&L, i, &e) //删除数据元素

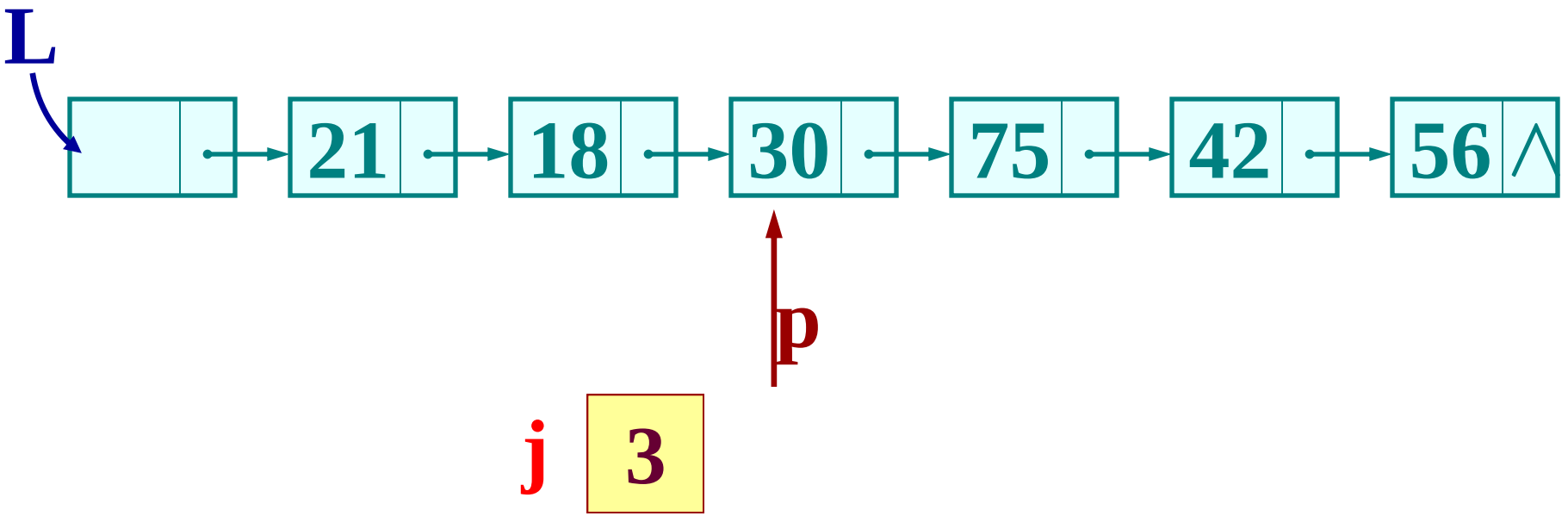
ClearList(&L)　　//重置线性表为空表

CreateList(&L, n)　//生成含n个数据元素的链表

线性表操作 GetElem(L, i, &e)

在单链表中的实现：

在单链表中，任何两个元素的存储位置之间没有固定的联系，然而每个元素的存储位置都包含在其直接前趋结点的信息中。在单链表中，取得第i个数据元素必须从头指针出发寻找。因此，单链表是非随机存取的存储结构。



单链表是一种顺序存取的结构，为找第 i 个数据元素，必须先找到第 i-1 个数据元素。

因此，查找第 i 个数据元素的基本操作为：移动指针，比较 j （当前指针指向的数据元素的位置）和 i 。

令指针 p 始终指向线性表中第 j 个数据元素。

```
Status GetElem_L(LinkList L, int i, ElemType &e) {
    // L是带头结点的链表的头指针，以 e 返回第 i 个元素
  p = L->next;   j = 1;   // p指向第一个结点，j为计数器
  while (p && j<i)  { p = p->next;  ++j;  }
    // 顺指针向后查找，直到 p 指向第 i 个元素或 p 为空
  if ( !p || j>i )
     return ERROR;      // 第 i 个元素不存在
  e = p->data;          // 取得第 i 个元素
  return OK;
} // GetElem_L
```

算法时间复杂度为：

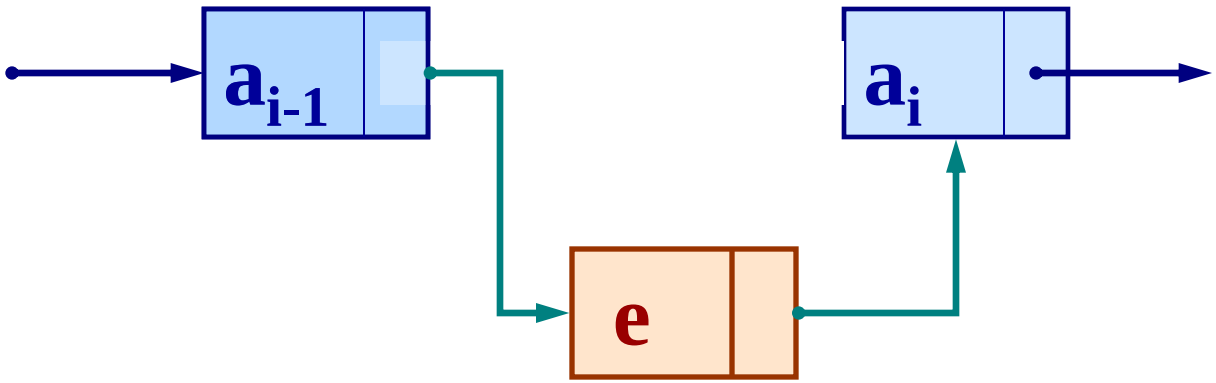
O(ListLength(L))

线性表操作 ListInsert(&L, i, e)

在单链表中的实现：

有序对 $<a_{i-1}, a_i>$

改变为 $<a_{i-1}, e>$ 和 $<e, a_i>$



可见，在链表中插入结点只需要修改指针。但同时，若要在第 i 个结点之前插入元素，修改的是第 i-1 个结点的指针。

因此，在单链表中第 i 个结点之前进行插入的基本操作为:

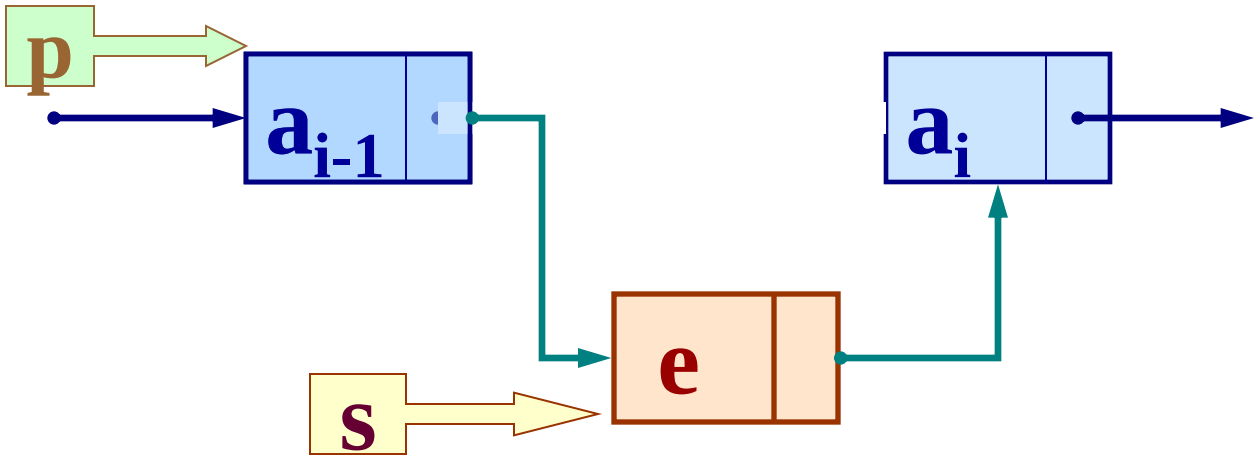
找到线性表中第 i-1 个结点，然后修改其指向后继的指针。

```
Status ListInsert_L(LinkList L, int i, ElemType e) {
        // L 为带头结点的单链表的头指针，本算法
        // 在链表中第i 个结点之前插入新的元素 e
   p = L;    j = 0;
   while (p && j < i-1)
       { p = p->next;  ++j; }      // 寻找第 i-1 个结点
   if (!p || j > i-1)
        return ERROR;              // i 大于表长或者小于1

       ……
} // LinstInsert_L
```

算法的**时间复杂度**为：

O(ListLength(L))

```
s = (LinkList) malloc ( sizeof (LNode));
                                    // 生成新结点
s->data = e;
s->next = p->next;      p->next = s;   //插入
return OK;
```



C语言中的两个标准函数：malloc和free

假设p和q是linklist型的变量，则执行p=(linklist)malloc(sizeof(node))的作用是：由系统生成一个node型的结点，同时将该结点的起始位置赋给指针变量p;执行free(q)的作用是由系统回收一个node型的结点，回收后的结点可以备作再次生成结点时用。

因此，单链表是一种动态结构，整个可用存储空间可为多个链表共同享用，每个链表占用的空间不需预先分配划定，而是可以由系统应需求即时生成。建立线性表的链式存储结构的过程就是一个动态生成链表的过程。即从空表的初始状态起，依次建立各元素结点，并逐个插入链表。

线性表操作 ListDelete (&L, i, &e)

在单链表中的实现：

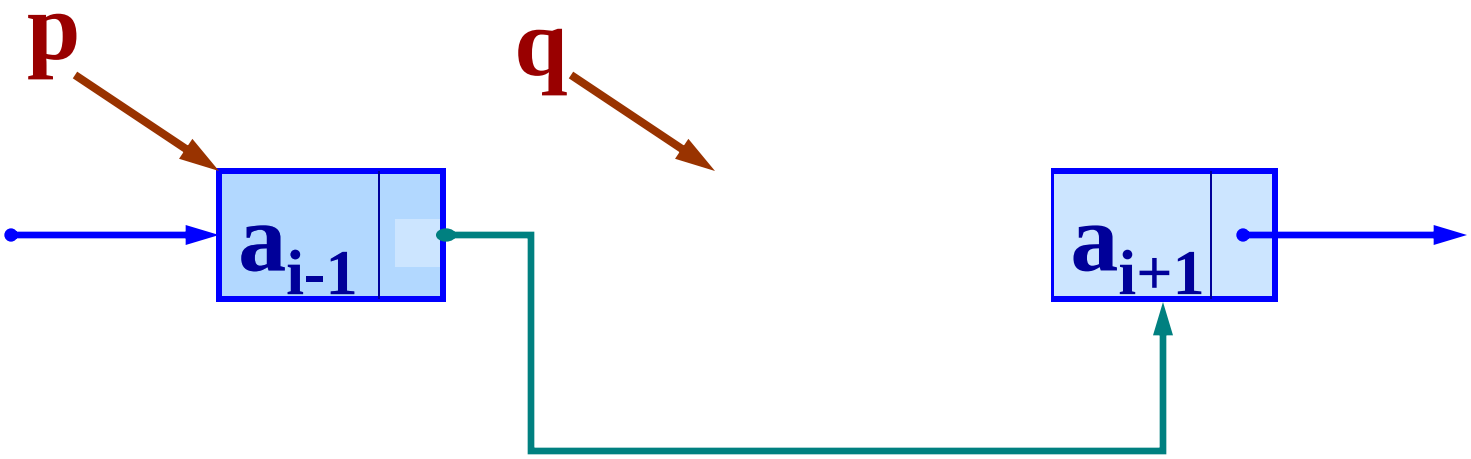
有序对<a_{i-1}, a_i> 和 <a_i, a_{i+1}>

改变为 <a_{i-1}, a_{i+1}>

在单链表中删除第 i 个结点的基本操作为: 找到线性表中第i–1个结点，修改其指向后继的指针。

q = p->next; **p->next = q->next;**

e = q->data; ***free(q);***

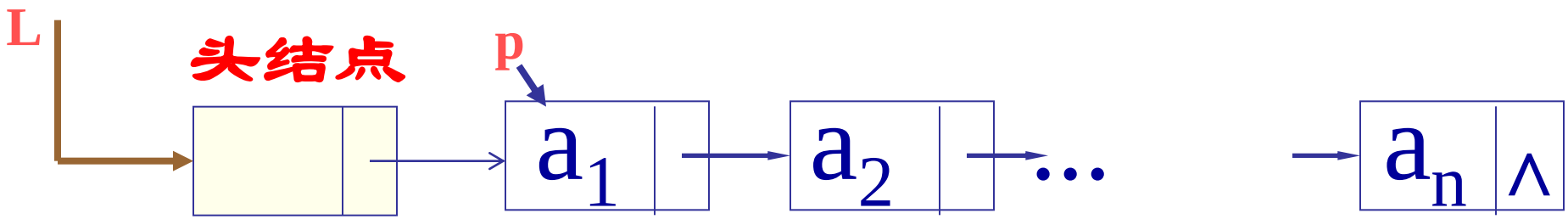


```
Status ListDelete_L(LinkList L, int i, ElemType &e)
{   // 删除以 L 为头指针(带头结点)的单链表中第 i 个结点
 p = L;    j = 0;
 while (p->next && j < i-1) {  p = p->next;   ++j; }
                //寻找第 i-1 个结点，并令 p 指向第i-1个结点
 if  (!(p->next) || j > i-1)
       return ERROR;                  // 删除位置不合理
 q = p->next;   p->next = q->next; // 删除并释放结点
 e = q->data;   free(q);
 return OK;
} // ListDelete_L
```

算法的时间复杂度为：

O(ListLength(L))

线性表操作 ClearList(&L) 在单链表中的实现:




```
void ClearList(&L) {
            // 将单链表重新置为一个空表
   while (L->next) {
     p=L->next;    L->next=p->next;
        free(p);
   }
   L->next=Null;
} // ClearList
```

算法时间复杂度:

O(ListLength(L))

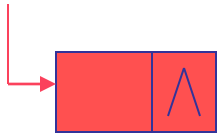
如何从线性表得到单链表？

链表是一个动态的结构，它不需要予分配空间，因此生成链表的过程是一个结点“逐个插入”的过程。

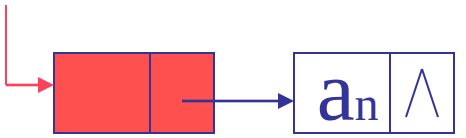
例如：逆位序输入 n 个数据元素的值，建立带头结点的单链表。

操作步骤：

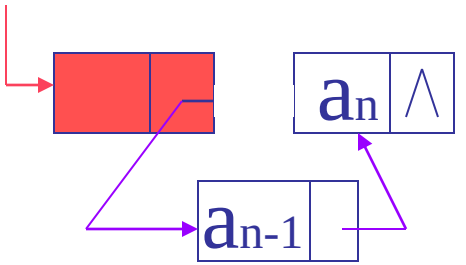
一、建立一个“空表”；



二、输入数据元素a_n，建立结点并插入；



三、输入数据元素a_{n-1}，建立结点并插入；



四、依次类推，直至输入a_1为止。

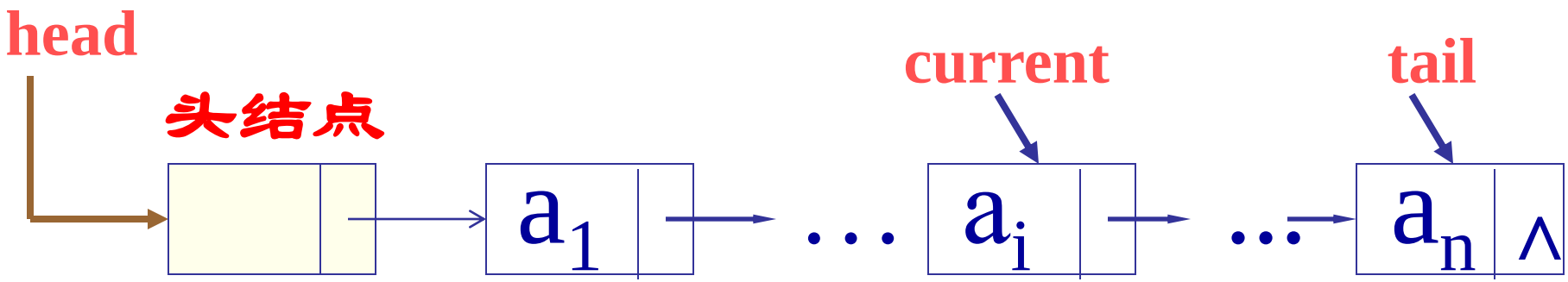
```
void CreateList_L(LinkList &L, int n) {
        // 逆序输入 n 个数据元素，建立带头结点的单链表
   L = (LinkList) malloc (sizeof (LNode));
   L->next = NULL;        //先建立一个带头结点的单链表
    for (i = n; i > 0; --i) {
       p = (LinkList) malloc (sizeof (LNode));
       scanf(&p->data);                          //输入元素值
       p->next = L->next; L->next = p;     //插入
    }
} // CreateList_L
```

算法的**时间复杂度**为：

O(Listlength(L))

四、改进的线性链表类型

```
typedef struct LNode {    // 结点类型
    ElemType      data;
    struct LNode   *next;
} *Link, *Position;

typedef struct {      // 链表类型
    Link  head, tail;
                  // 分别指向头结点和最后一个结点的指针
    int   len;                // 指示链表长度
    Link  current;            // 指向当前被访问的结点的
                              // 指针，初始位置指向头结点
} LinkList;
```



Status MakeNode(Link **&**p, ElemType e);

// 分配由 **p** 指向的值为**e**的结点，并返回**OK**，

// 若分配失败，则返回 **ERROR**

void FreeNode(Link **&**p);

// 释放 **p** 所指结点

链表的基本操作: (P37)

{结构初始化和销毁结构}

Status InitList(LinkList **&**L);

O(1)

// 构造一个空的线性链表 L，其头指针、
// 尾指针和当前指针均指向头结点，表长
// 为零。

Status DestroyList(LinkList **&**L);

O(n)

// 销毁线性链表 L，L不再存在。

{引用型操作}

Status ListEmpty (LinkList L); //判表空 O(1)

int ListLength(LinkList L); //求表长 O(1)

Status Prior(LinkList L); O(n)

//改变当前指针指向其前驱

Status Next (LinkList L); O(1)

//改变当前指针指向其后继

ElemType GetCurElem (LinkList L); O(1)

//返回当前指针所指数据元素

Status LocatePos(LinkList L, int i);

//改变当前指针指向第i个结点

O(n)

Status LocateElem (LinkList L, ElemType e,
Status (*compare)(ElemType, ElemType));

//若存在与e 满足函数compare( )判定关系的
//元素，则移动当前指针指向第1个满足条件的
//元素，并返回OK；否则返回ERROR

O(n)

Status ListTraverse(LinkList L, Status(*visit)());

//依次对L的每个元素调用函数visit()

O(n)

{加工型操作}

Status ClearList (LinkList &L);

//重置 L 为空表

O(n)

Status SetCurElem(LinkList &L, ElemType e);

//更新当前指针所指数据元素

O(1)

Status Append (LinkList &L, Link s);

//在表尾结点之后链接一串结点

O(s)

Status InsAfter (LinkList &L, Elemtype e);

//将元素 e 插入在当前指针之后

O(1)

Status DelAfter (LinkList &L, ElemType& e);

//删除当前指针之后的结点

O(1)

```
Status InsAfter( LinkList& L, ElemType e ) {
    //若当前指针在链表中，则将数据元素e插入在线性
    //链表L中当前指针所指结点之后，并返回OK;
    //否则返回ERROR。
    if ( ! L.current )  return ERROR;
    if (! MakeNode( s, e) )  return ERROR;
    s->next = L.current->next;
    L.current->next = s;
    if (L.tail = L.current)  L.tail = s;
    L.current = s;  len++;     return OK;
} // InsAfter
```

```
Status DelAfter( LinkList& L, ElemType& e ) {
    //若当前指针及其后继在链表中，则删除线性链表L中当前
    //指针所指结点之后的结点，并返回OK；否则返回ERROR。
    if ( !(L.current && L.current->next ) )
        return ERROR;
    q = L.current->next;
    L.current->next = q->next;
    if (L.tail = q)  L.tail = L.current;
                                    //被删结点恰为尾结点
    e=q->data;  FreeNode(q);
    return OK;
} //DelAfter
```

利用上述定义的线性链表的基本操作如何完成线性表的其它操作？

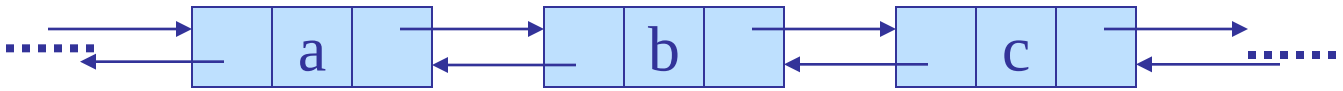
例：

```
Status ListInsert_L(LinkList L, int i, ElemType e) {
      //在带头结点的单链线性表L的第i个元素之前插入元素e
   if (!LocatePos (L, i-1))  return ERROR;
                              //i值不合法，第i-1个结点不存在
   if (InsAfter (L, e))  return OK;      //完成插入
   else return ERROR;
} // ListInsert_L
```

五、其它形式的链表

1. 双向链表

链表中的结点有两个指针域，其一指向直接后继，另一指向直接前趋。

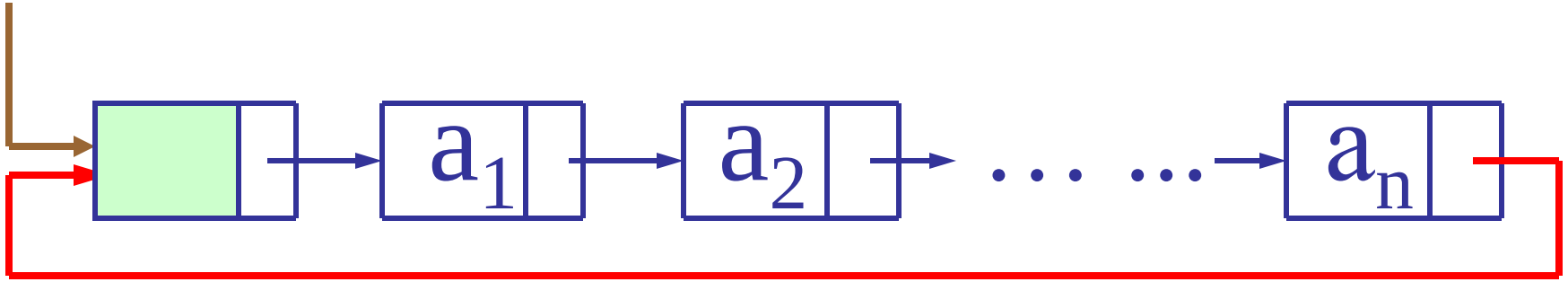


用C语言可描述如下：


```
typedef struct  DuLNode {
    ElemType          data;    // 数据域
    struct DuLNode  *prior;
                                // 指向前驱的指针域
    struct DuLNode  *next;
                                // 指向后继的指针域
} DuLNode, *DuLinkList;
```

2. 循环链表

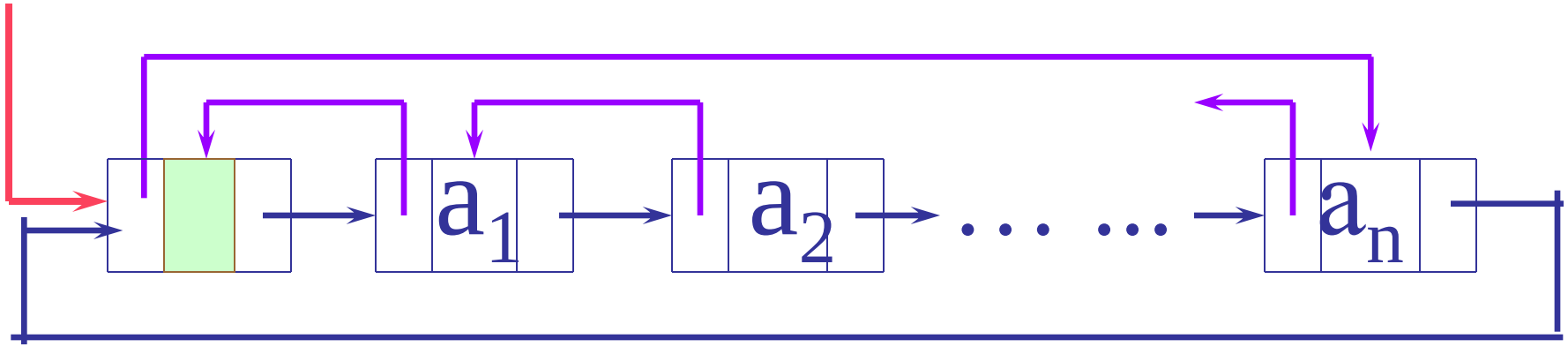
最后一个结点的指针域的指针又指回第一个结点的链表。



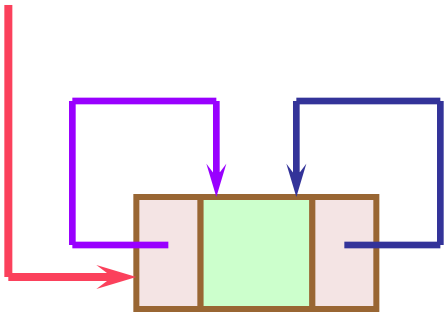
和单链表的差别仅在于，判别链表中最后一个结点的条件不再是“后继是否为空”，而是“后继是否为头结点”。

3.双向循环链表

非空表



空表



双向链表的操作特点：

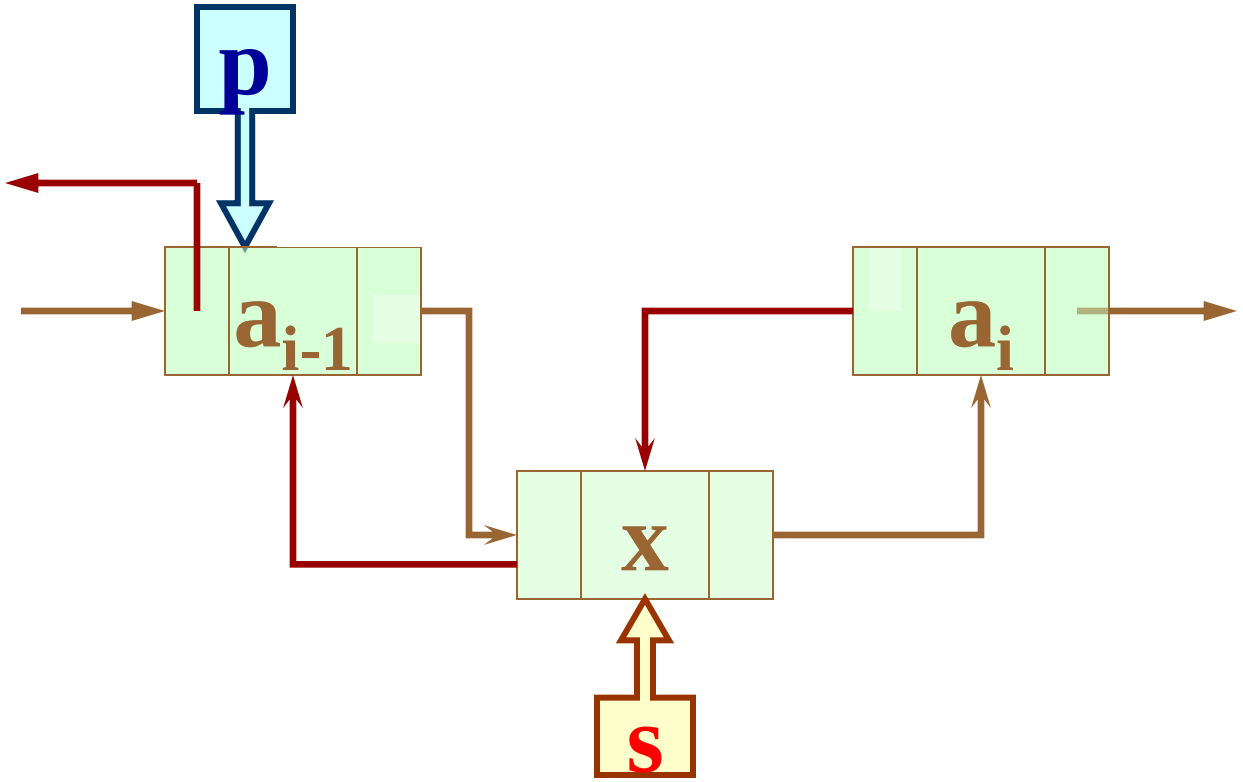
1.“查询”和单链表相同

Length 、Get 、Locate 操作仅涉及一个方向的指针，则算法描述与单向线性链表的操作相同。

2.“插入”和“删除”时与单向线性链表有区别

双向链表中需要同时修改两个方向上的指针。

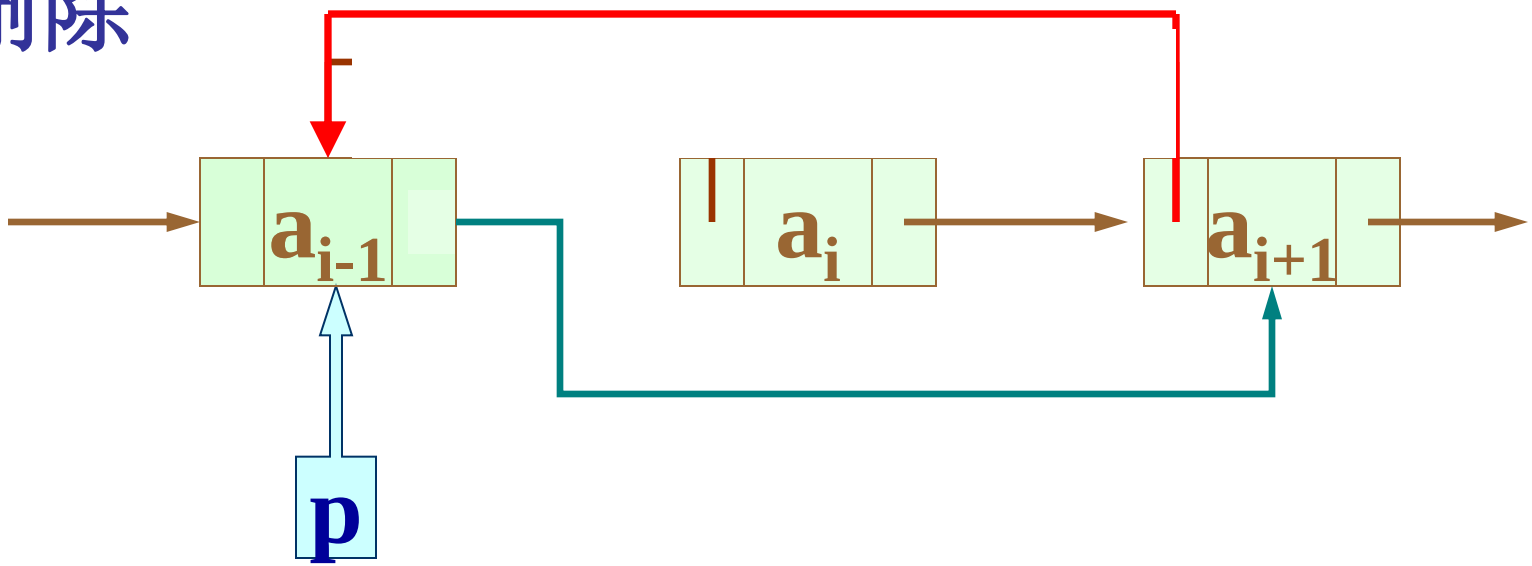
插入



s->next = p->next; p->next = s;

s->next->prior = s; s->prior = p;

删除



p->next = p->next->next;

p->next->prior = p;

```
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e) {
                    // 在带头结点的双向循环链表L中第i 个结点之前
                    // 插入新的元素 e，1≤i ≤表长+1。
   if  (!(p=GetElem_DuL(L,i)))  // 第i 个元素的位置指针p
         return ERROR;          //  p=NULL，即第i个元素不存在
   if (!(s=(DuLinkList)malloc(sizeof(DuLNode))))
            return ERROR;          // 存储空间分配失败
   s->data=e;
   s-> prior = p-> prior;    p-> prior ->next = s;
   s->next= p;    p->prior = s;
   return OK;
} // ListInsert_DuL
```

算法的**时间复杂度**为：

O(ListLength(L))

```
Status ListDelete_DuL(DuLinkList &L, int i, ElemType &e) {
        // 删除带头结点的双向循环链表L中第i 个元素
        // i的合法值为 1≤i ≤表长。
    if  (!(p=GetElem_DuL(L,i)))  // 第i 个元素的位置指针p
        return ERROR;        //  p=NULL，即第i个元素不存在
    e=p->data;
    p-> prior ->next = p->next;
    p->next->prior= p->prior;
    free(p);
    Return OK;
} // ListDelete_DuL
```

算法的**时间复杂度**为：

O(ListLength(L))

4.静态链表

借用一维数组来描述线性链表，称静态链表。这种描述方法便于在不设“指针”类型的高级程序语言中使用链表结构。 其类型说明如下:

```
#define MAXSIZE 1000          // 链表的最大长度
Typedef struct {
      elemtype  data;
      int cur;                    //指示结点在数组中的相对位置
} SlinkList[MAXSIZE];
```

静态链表示例

0		1
1	ZHAO	2
2	QIAN	3
3	SUN	4
4	LI	5
5	ZHOU	6
6	WU	7
7	ZHENG	8
8	WANG	0
9		
10		

原始状态

0		1
1	ZHAO	2
2	QIAN	3
3	SUN	4
4	LI	9
5	ZHOU	6
6	WU	8
7	ZHENG	8
8	WANG	0
9	SHI	5
10		

在”LI”后插入“SHI”和删除“ZHENG”之后的状

如上描述的链表中，数组的一个分量表示一个结点，同时用游标（指示器cur）代替指针指示结点在数组中的位置。数组的第0个分量可看成头结点，其指针域指示链表的第一个结点。

这种存储结构仍需要预先分配一个较大的存储空间，但在作线性表的插入和删除操作时不需移动元素，仅需修改指针，故仍具有链式存储结构的主要优点。

为了和指针型描述的线性链表相区别，我们给这种用数组描述的链表起名为静态链表。

假设S为SLinkList型的变量，则S[0].cur指示第一个结点在数组中的位置。

若设i=S[0].cur，则S[i].data存储线性表的第一个元素，且S[i].cur指示第二个结点在数组中的位置。

一般情况，若第i个分量表示链表的第k个结点，则S[i].cur指示第k+1个结点的位置。

因此，在静态链表中实现线性表的操作和动态链表相似，以整型游标i代替动态指针p，

在静态链表中指针后移操作（类似于p=p->next）

用i= S[i].cur 实现。

静态链表中定位函数LocateElem的实现

```
int LocateElem_SL(SLinkList S,ElemType e) {
        //在静态单链线性表S中查找第1个值为e的元素。
        //若找到，则返回它在S中的位序；否则返回0。
    i=S[0].cur;                          // i 指示表中第一个结点
    while (i&&S[i].data!=e) i=S[i].cur;
                                         //在表中顺链查找
     return  i;
} // LocateElem_SL
```

类似地可写出在静态链表中实现插入和删除操作的算法。静态链表中指针修改的操作和单链表基本相同，所不同的是，需由用户自己实现malloc和free这两个函数。为了辨明数组中哪些分量未被使用，解决的办法是将所有未被使用过的以及被删除的分量用游标链成一个备用的链表，每当进行插入时便可从备用链表上取得第一个结点作为待插入的新结点；反之，在删除时将从链表中删除下来的结点链接到备用链表上。

将整个数组空间初始化成一个链表

```
Void InitSpace_SL(SLinkList &space) {
    // 将一维数组space中各分量链成一个备
    // 用链表，space[0].cur为头指针，“0”表示空指针
  for (i=0; i<MAXSIZE-1;++i)  space[i].cur=i+1;
  space[MAXSIZE-1].cur=0;
}
```

从备用链表space中取得一个结点

```
int Malloc_SL(SLinkList &space)    {
                        //若备用链表space非空，则返回
                        //分配的结点下标，否则，返回0
    i=space[0].cur;              //备用链表的头结点
    if(space[0].cur)    space[0].cur=space[i].cur;
    return i;                    //指向结点的位置
}
```

将空闲结点链结到备用链表space的头部

```
void Free_SL(SLinkList &space,int k) {
    //将下标为K的空闲结点回收到备用链表space
    space[k].cur=space[0].cur;
    space[0].cur=k;
}
```

六、有序表类型

集合中
任意两个
元素之间
均可以
进行比较

ADT Ordered_List {

数据对象：$S = \{ x_i | x_i \in \textbf{OrderedSet}, i=1,2,\ldots,n, n\geq 0 \}$

数据关系：$R = \{ \langle x_{i-1}, x_i \rangle \mid x_{i-1}, x_i \in S, x_{i-1} \leq x_i, i=2,3,\ldots,n \}$

基本操作：

Status LocateElem(LinkList L, ElemType e, Position &q,

int(*compare)(ElemType,ElemType))

初始条件：有序表L已存在。

操作结果：若有序表L中存在元素e，则q指示L中第一个值为e 的元素的位置，并返回函数值TRUE；否则q 指示第一个大于e 的元素的前驱的位置，并返回函数值FALSE。

例如:

(12, 23, 34, 45, 56, 67, 78, 89, 98)

若 e = 45,
则 q 指向

若 e = 88,
则 q 指向

表示值为 88 的元素应插入在该指针所指结点之后。

Status OrderInsert(LinkList L, ElemType e, int(*compare)(ElemType,ElemType))

初始条件：有序表L已存在。

操作结果：按有序判定函数compare()的约定，将值为e的结点插入到有序链表L的适当位置。

有序链表的其他操作与一般线性链表的其他操作都一致，只有上述两个操作有区别。

回顾例2-2(用Lb中值不同的元素构造La):

```
void union(List &La, List Lb) {
    InitList(La);                          // 构造(空的)线性表La
    La_len=ListLength(La);   Lb_len=ListLength(Lb);
    for (i = 1;  i <= Lb_len;  i++) {
        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);
                    // La中不存在和 e 相同的数据元素，则插入之
    }// for
} // union
```

算法时间复杂度： $O(n^2)$

```
void purge(List &La, List Lb) {          // Lb为有序表
    InitList(LA);    La_len = ListLength(La);
    Lb_len =ListLength(Lb);                    // 求线性表的长度
    for (i = 1;  i <= Lb_len;  i++) {
        GetElem(Lb, i, e);          // 取Lb中第i个数据元素赋给 e
        if (ListEmpty(La) || !equal (en, e)) {
            ListInsert(La, ++La_len, e);
            en = e;
        }              // La中不存在和 e 相同的数据元素，则插入之
    }
} // purge
```

算法时间复杂度：O(n)

链式存储结构的优缺点

(1) 插入和删除运算时，无须移动表中元素的位置，只需修改有关结点的指针内容;

(2)不需要一块连续的存储空间，只要能存放一个数据元素的空闲结点就可以被利用;

(3)表的规模易扩充;

(4) 不能随机访问表中元素，访问时间与元素在表中的位置有关。

在实际应用中采用哪一种存储结构更合适？

① 存储空间

顺序表要求预先分配存储空间，一般在程序执行之前是难以估计存储空间大小，估计过大会造成浪费，估计过小又会产生空间溢出。而链式存储结构的存储空间是动态分配，只要内存空间有空间，就可动态申请内存空间，不会产生溢出。对于存储空间的考虑也可以用存储密度的大小来衡量。其中存储密度的大小定义为一个结点数据本身所占用的存储量与结点结构所占用的存储量的比值。一般地，存储密度越大，存储空间的利用率就越高。显然，顺序表的存储密度为1，而链式存储结构的存储密度则小于1。

② 运算时间

顺序存储结构是一种随机存取的结构，便于元素的随机访问。即表中任一元素都可在O(1)时间复杂度情况下迅速而直接地存取。而链式存储结构，必须从头指针开始顺着链扫描才能取得，一般情况下其时间复杂度为O(n)，所以对于那些只进行查找运算而很少做插入和删除等的运算，宜采用顺序存储结构。但在顺序表中进行元素的插入和删除运算时，需移动大量元素，平均要移动约半数的元素。而采用链式存储结构，由于进行元素的插入或删除，只需修改指针并结合一定的查找。所以，对于那些需要经常频繁地进行元素的插入和删除运算的线性表，其存储结构应采用链式存储结构。

③ 程序设计语言

这主要是指依据某种高级语言是否提供指针类型或者依据实际需要决定存储结构是选用静态链表还是动态链表。

总之，线性表的顺序实现和链式实现各有优缺点，是无法笼统地认定哪种优，哪种劣。只能根据实际问题的具体实现需要，对各方面的优缺点加以综合平衡来确定适宜的存储结构。

2.4 一元多项式的表示

在数学上，一元n次多项式

$$P_n(x)= p_0 + p_1 x+ p_2 x^2 + \ldots + p_n x^n$$

由n+1个系数唯一确定，可用线性表P来表示

$$P=(p_0, p_1, p_2, \ldots, p_n)$$

每一项的指数 i 隐含在其系数P_i的序号里。

假设$Q_m(x)$是一元m次多项式，同样可用线性表Q来表示

$$Q=(q_0, q_1, q_2, \ldots, q_m)$$

设m<n，则两个多项式相加的结果

$$R_n(x)= P_n(x)+ Q_m(x)$$

可用线性表R表示

$$R=(p_0+q_0, p_1+q_1, p_2+q_2, \ldots, p_m+q_m, p_{m+1}, \ldots, p_n)$$

在通常应用中，多项式的次数很高且变化很大，比如下面的一元稀疏多项式：

$$S(x) = 1 + 3x^{10000} - 2x^{20000}$$

如果用顺序存储结构，就要用长度为20001的线性表来表示，而表中仅有三个非零元素，这种对内存空间的浪费是应该避免的。

解决的办法是,只存储非零项。

如果只存储非零项，则显然必须同时存储系数项以及相应的指数。

一般情况下的一元稀疏多项式可写成

$$P_n(x) = p_1x^{e1} + p_2x^{e2} + \text{---} + p_mx^{em}$$

其中：p_i 是指数为 e_i 的项的非零系数，

$$0 \le e_1 < e_2 < \text{---} < e_m = n$$

可用一个长度为m且每个元素有两个数据项（系数项和指数项）的线性表表示:

$$((p_1, e_1), (p_2, e_2), \cdots\cdots, (p_m, e_m))$$

例如:

$$P_{999}(x) = 7x^3 - 2x^{12} - 8x^{999}$$

可用线性表

((7, 3), (-2, 12), (-8, 999))表示

在最坏情况下，n+1(=m)个系数都不为零，则比只存储每项系数的方案要多存储一倍的数据。但对于S(x)类（稀疏一元多项式）的多项式，这种表示将大大节省空间。

对于像

$((p_1, e1), (p_2, e2), \ldots\ldots, (p_m, em))$

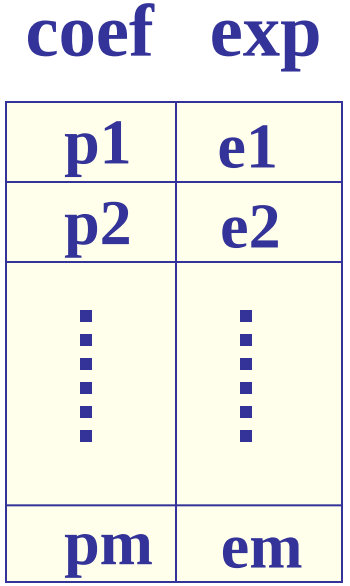
的线性表，有两种存储结构：顺序存储结构和链式存储结构。在实际的应用程序中取用哪一种，则要视多项式作何种运算而定。

(1)若只求多项式的值，运算中无须修改多项式的系数和指数值，采用顺序存储结构为宜。

(2)若求两个多项式之和，采用链式存储结构为宜。

多项式选择顺序存储结构:

```
#define  maxlen  maxsize;
typedef struct // 定义结点（元素）类型
      {  float coef;   // 系数域
         int   exp;    // 指数域
      } elemtp;
typedef  struct   // 定义线性表类型
      {  elemtp  vec[maxlen];
                        // 定义线性表为数组
         int  len;       // 线性表长度
      } sequenlist;
```



多项式选择链式存储结构:

```
typedef struct  node                  // 定义结点类型
    {  float coef;                    // 系数域
       int   exp;                     // 指数域
       struct node *next;             // 指针域
    } polynode;
polynode *p,*q;                       // 定义链表（指针）类型
```

抽象数据类型一元多项式的定义如下：

ADT Polynomial {

数据对象：

$D=\{ a_i \mid a_i \in TermSet, i=1,2,...,m, m\geq 0$

TermSet 中的每个元素包含一个

表示系数的实数和表示指数的整数 }

数据关系：

$R1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, \quad i=2,...,n$

且a_{i-1}中的指数值＜a_i中的指数值 }

基本操作：

CreatPolyn (&P, m)

操作结果：输入 m 项的系数和指数，
建立一元多项式 P。

DestroyPolyn (&P)

初始条件：一元多项式 P 已存在。

操作结果：销毁一元多项式 P。

PrintPolyn (&P)

初始条件：一元多项式 P 已存在。

操作结果：打印输出一元多项式 P。

PolynLength(P)

初始条件：一元多项式 P 已存在。

操作结果：返回一元多项式 P 中的项数。

AddPolyn (&Pa, &Pb)

初始条件：一元多项式 **Pa** 和 **Pb** 已存在。

操作结果：完成多项式相加运算，即：

Pa = Pa + Pb，并销毁一元多项式 **Pb**。

SubtractPolyn(&Pa， &Pb)

初始条件：一元多项式Pa 和Pb已经存在

操作结果：完成多项式相减运算，Pa=Pa-Pb，
并销毁一元多项式Pb。

MultiplyPolyn(&Pa，&Pb)

初始条件：一元多项式Pa 和Pb已经存在

操作结果：完成多项式相乘运算，Pa=Pa*Pb，
并销毁一元多项式Pb。

} ADT Polynomial

一元多项式的实现（采用链表存储）

结点的数据元素类型定义为：

```
typedef struct {                    // 项的表示
    float  coef;                    // 系数
    int    expn;                    // 指数
} ElemType;

typedef  OrderedLinkList   polynomial;
                    // 用带表头结点的有序链表表示多项式
```

```
Status CreatPolyn ( polynomail &P, int m ) {
    // 输入m项的系数和指数，建立表示一元多项式的有序链表P
    InitList (P); h=GetHead(P); e.coef = 0.0;  e.expn = -1;
    SetCurElem (h, e);                    // 设置头结点的数据元素
    for ( i=1; i<=m; ++i ) {              // 依次输入 m 个非零项
        scanf (e.coef, e.expn);
        if (!LocateElem ( P, e, (*cmp)()) ) // 当前链表中不存
                                            在该指数项
        if ( !InsAfter ( P, e ) )   return ERROR;
    }
    return OK;
} // CreatPolyn
```

注意： 1. 输入次序不限;

2. 指数相同的项只能输入一次。

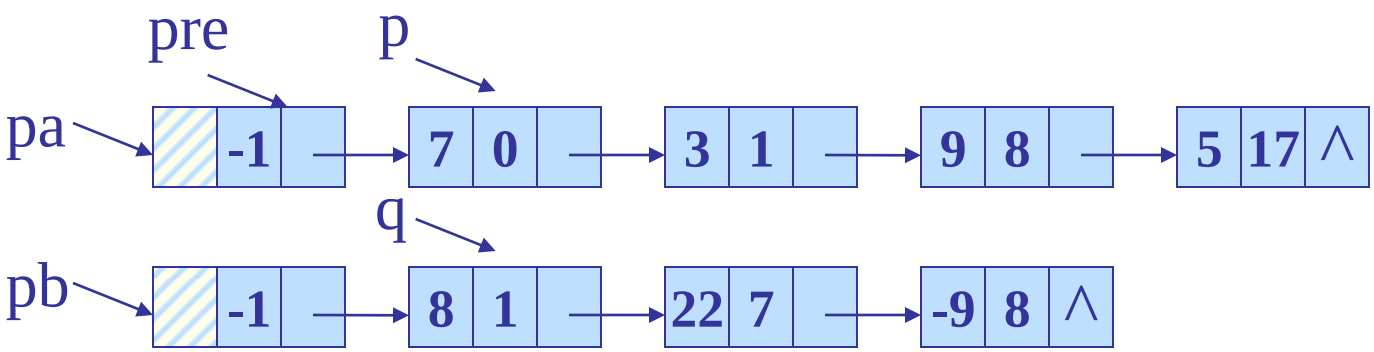
设An (x)和Bm (x)都是形如

$$P_n(x)= p_1 x^{e1}+ p_2 x^{e2} + \ldots + p_m x^{em}$$

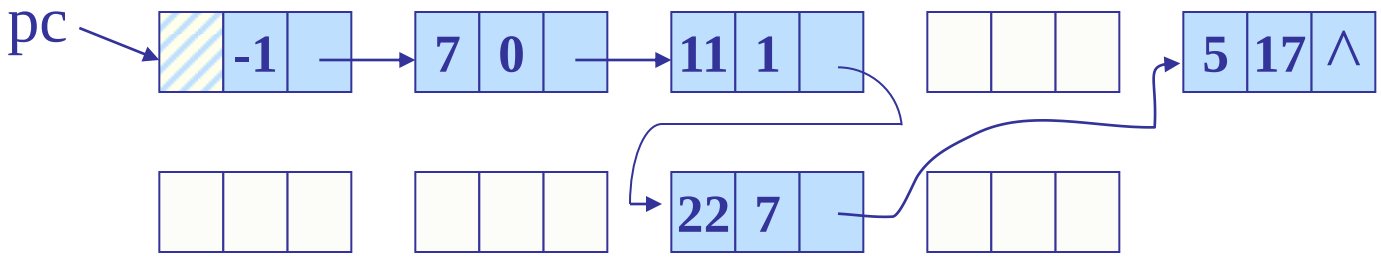
的一元多项式，现求两多项式之和

$$C_n(x)= A_n(x)+B_m(x) \quad (m<n)$$

显然，应采用链式存储结构。用两个线性链表分别表示两个一元多项式，链表中的每个结点表示多项式中的一项。



多项式表的单链存储结构



相加得到的和多项式

两个一元多项式的运算规则:

指数相同的项，对应系数相加，若和不为零，则生成和多项式的一项；指数不相同的项，则复制到和多项式中。

实现方法:

设两个指针p, q分别指向两个多项式中的某一个结点，则可以比较结点的指数项，

①若p→exp<q→exp, *p结点是多项式中的一项，p指针在原来的链表上后移;

② 若p→exp=q→exp, 对应项的系数相加，若系数和不为零，只要修改*p结点的系数域，释放q结点，否则应删除*p结点，释放p、q结点； p、q指针在原来的链表上分别后移;

③ 若p→exp>q→exp, *q结点是多项式中的一项，*q结点应插在*p结点之前，且q指针在原来的链表上后移;

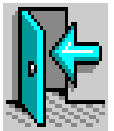
```
void polyadd (polynomial &pa, polynomial &pb)
 {   pc=pa;p=pa→next; q=pb→next; pre=pa;
     while (p && q)
        switch ( compare(p->expn, q->expn))
         {    case -1: {pre=p;p=p→next; break; }   // p指针后移
              case 0:
                        { x=p→coef+q→coef;
                          if (x!=0)
                            {p→coef=x; pre=p;}          // 修改p结点
                          else { pre→next=p→next;
                                 free(p);}                  // 删除p结点
                          p=pre→next; r=q; q=q→next; free( r ); break; }
              case 1: {  r=q→next; q→next=p; pre→next=q;
                         pre=q; q=r; break;}     // q结点插入在p结点之前
          }
     if (q)  pre→next=q;
     free(pb);
 }// polyadd
```

本章小结

1. 了解线性表的逻辑结构特性：是数据元素之间存在着线性关系，在计算机中表示这种关系的两类不同的存储结构是顺序存储结构和链式存储结构。用前者表示的线性表简称为顺序表，用后者表示的线性表简称为链表。

2. 熟练掌握这两类存储结构的描述方法，以及线性表的各种基本操作的实现。

3. 能够从时间和空间复杂度的角度综合比较线性表两种存储结构的不同特点及适用场合。



习题：2.19　　2.22　　2.38

2.19 已知线性表中的元素以值递增有序排列，并以单链表作存储结构。试写一高效的算法，删除表中所有值大于mink且小于maxk的元素（若表中存在这样的元素），同时释放被删结点空间，并分析你的算法的时间复杂度（注意： mink和maxk是给定的两个参变量，他们的值可以和表中相同，也可以不同）

2.22 试写一算法，对单链表实现就地逆置。

注意：没有特别说明，链表均带头结点

2.38　设有一个双向循环链表，每个结点中除有prior,data和next三个域外，还增设了一个访问频度域freq。在链表被起用之前，频度域freq的值均初始化为零，而每当对链表进行一次locate(L,x)的操作后，被访问的结点（即元素值等于x的结点）中的频度域freq的值便增1，同时调整链表中结点之间的次序，使其按访问频度非递减的次序顺序排列，以便始终保持被频繁访问的结点总是靠近表头结点。试编写符合上述要求的locate操作的算法。