第二章
线性表

线性表是一种最简单的**线性结构**

线性结构的**基本特征**为：

线性结构 是
一个数据元素的**有序（次序）集**

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 线性表的类型定义

2.2 线性表类型的实现
—— 顺序映象

2.3 线性表类型的实现
—— 链式映象

2.4 一元多项式的表示

2.1
线性表的类型定义

抽象数据类型**线性表**的定义如下：

ADT List {

数据对象：

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

{称 **n** 为线性表的**表长**;

称 **n=0** 时的线性表为**空表**。}

数据关系：

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

{设线性表为 $(a_1,\ a_2,\ \ldots,\ a_i,\ \ldots,\ a_n)$,

称 i 为 a_i 在线性表中的**位序**。}

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≤LengthList(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( ))

（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≤LengthList(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≤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)

（删除数据元素）

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。

2.2 线性表类型的实现----顺序映象

顺序映象

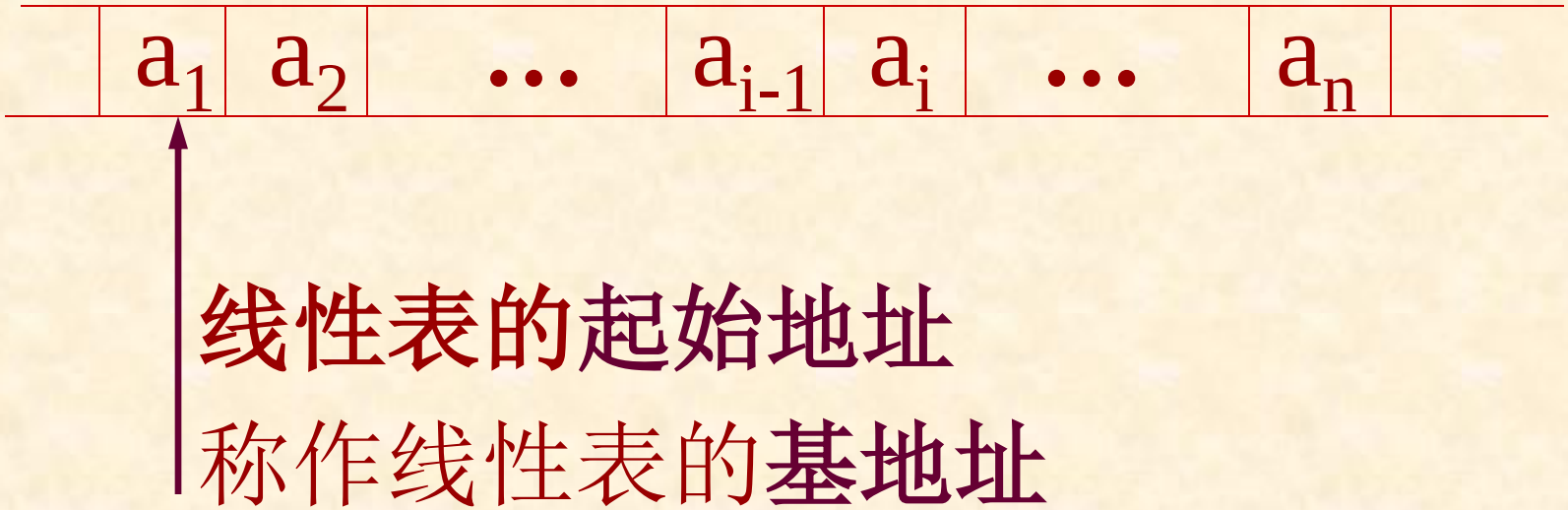
—— 以 x 的存储位置和 y 的存储位置之间某种关系表示逻辑关系<x,y>。

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。

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

即：$LOC(a_i) = LOC(a_{i-1}) + C$

一个数据元素所占存储量↑

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

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

↑基地址

顺序映像的 C 语言描述

```
#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
```

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

```
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);
   L.length = 0;
   L.listsize = LIST_INIT_SIZE
   return OK;
} // InitList_Sq
```

算法时间复杂度：**O(1)**

单选题 1分

设置

在线性结构中，所有结点都有(　　)个直接后继

A 0

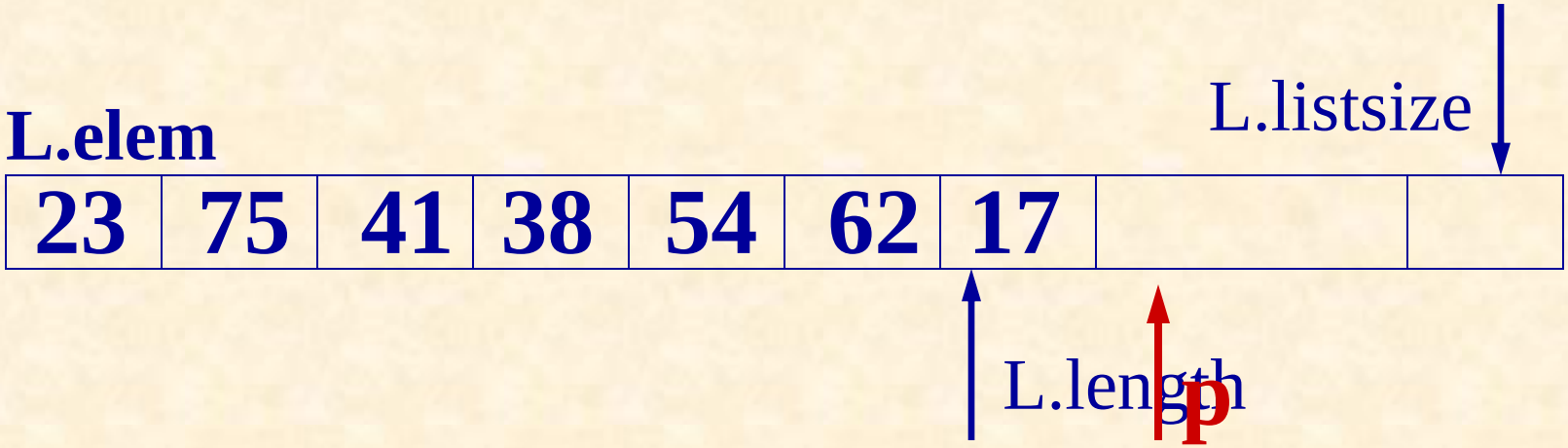
B 0或1

C 1

D 不确定

提交

顺序表的查找



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:

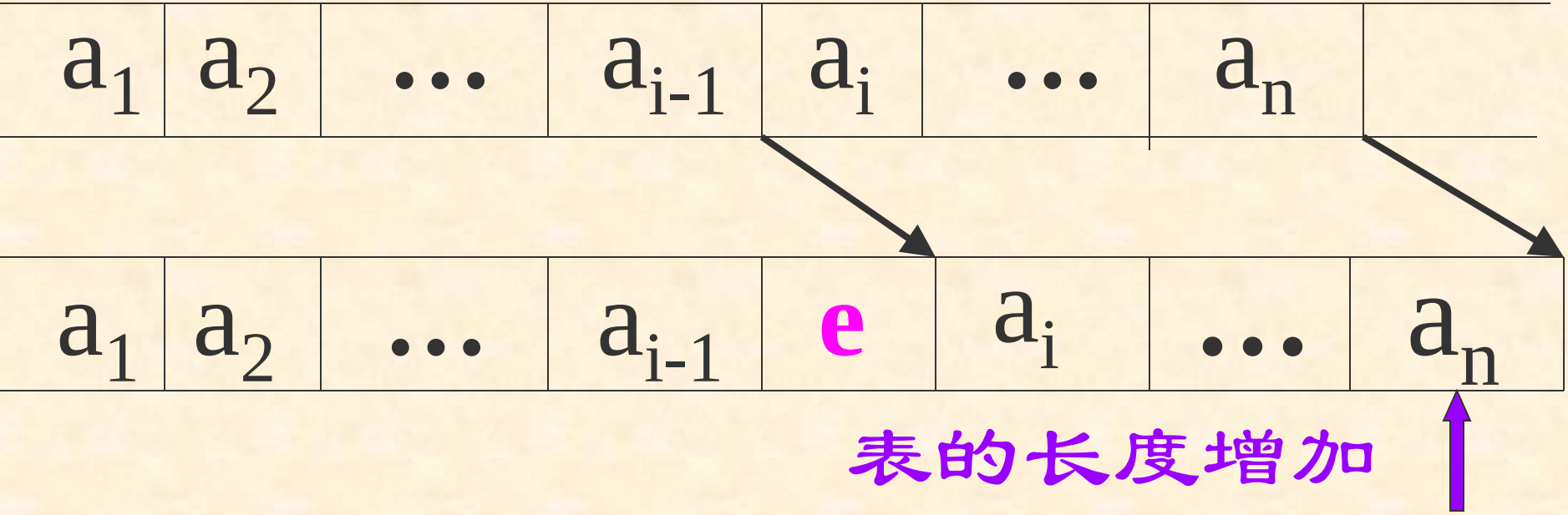
插入元素时，

线性表的**逻辑结构**发生什么变化？

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

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

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



```
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))

```
if (i < 1 || i > L.length+1) return ERROR;
                                    // 插入位置不合法
if (L.length >= L.listsize) {
                                    // 当前存储空间已满，增加分配
    newbase = (ElemType *)realloc(L.elem,
        (L.listsize+LISTINCREMENT)*sizeof (ElemType));
    if (!newbase) exit(OVERFLOW);
                                    // 存储分配失败
    L.elem = newbase;                    // 新基址
    L.listsize += LISTINCREMENT; // 增加存储容量
}
```

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**为：

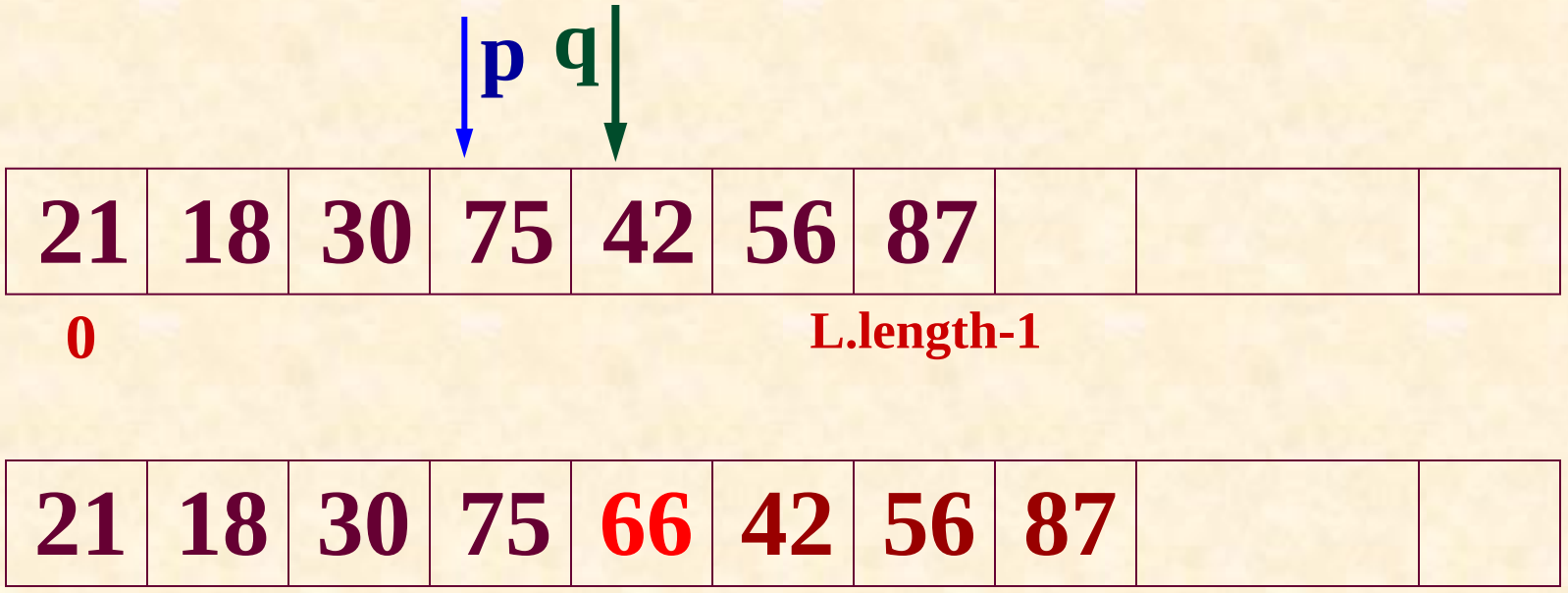
$$E_{is} = \sum_{i=1}^{n+1} p_i (n - i + 1)$$

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，则**移动元素的期望值**为：

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

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;
```



线性表操作

ListDelete(&L, i, &e)的实现：

首先分析：

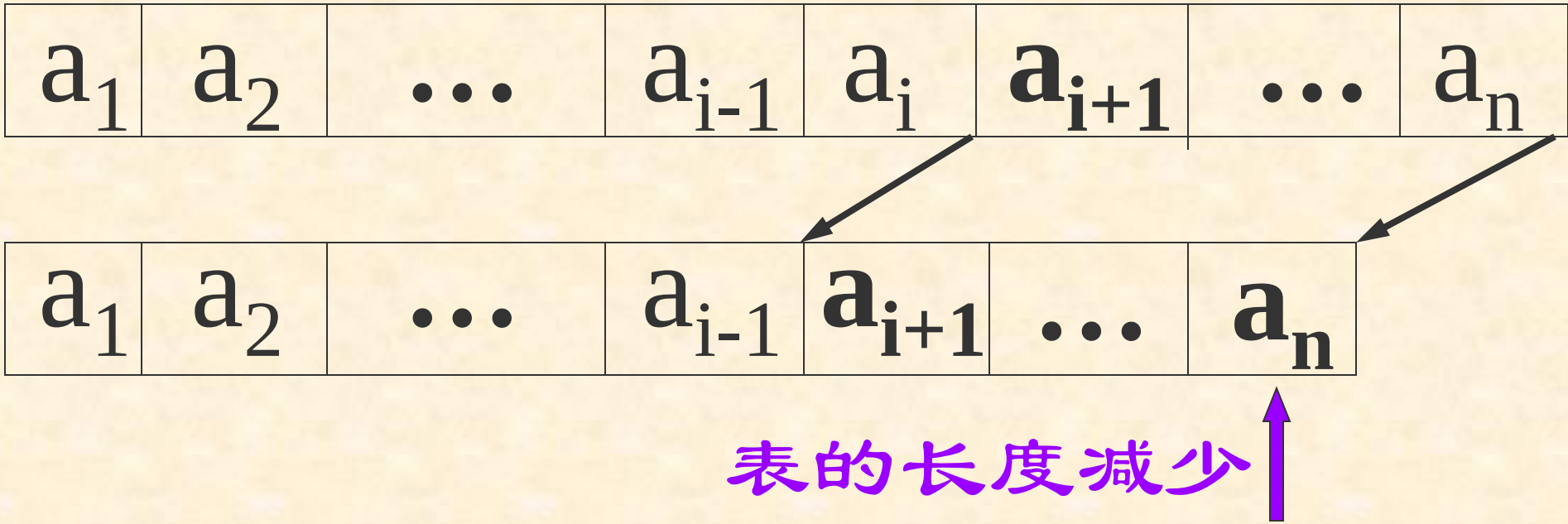
删除元素时，

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))

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**为：

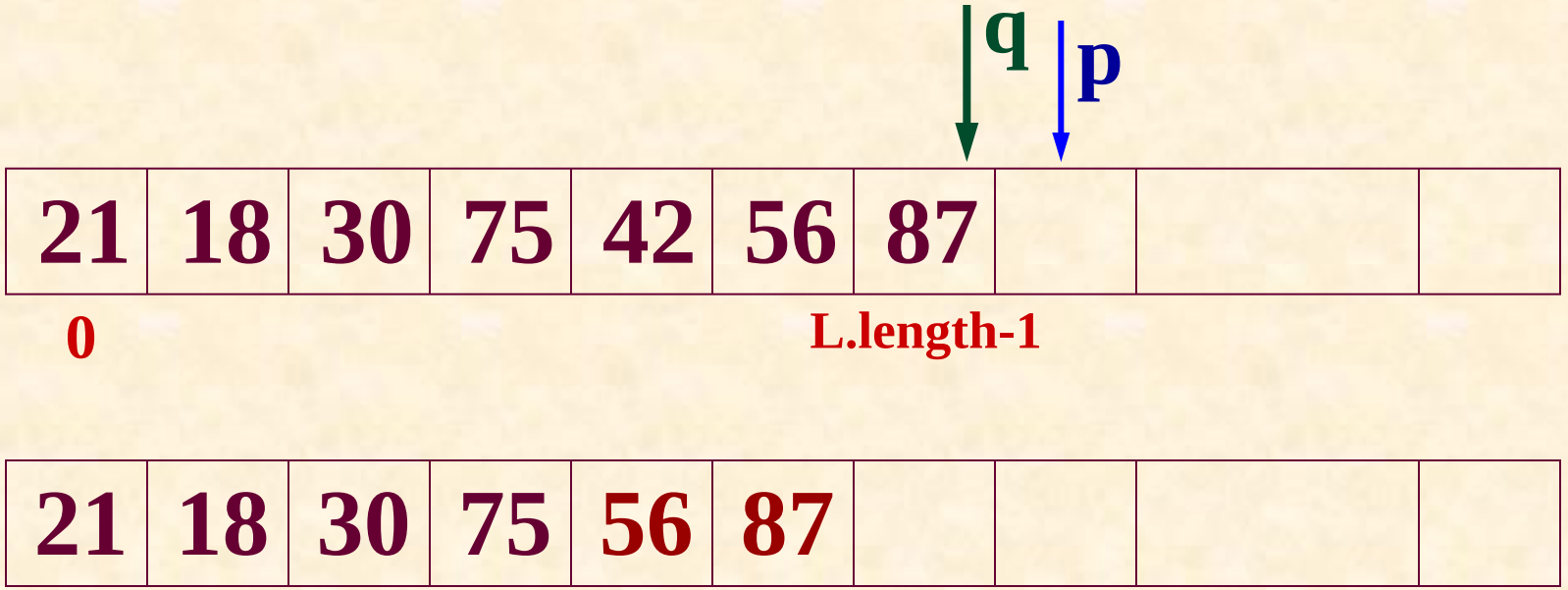
$$E_{dl} = \sum_{i=1}^{n} q_i (n-i)$$

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，则**移动元素的期望值**为：

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

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;
```



2.3 线性表类型的实现---链式映象

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

用一组**地址任意**的存储单元**存放**线性表中的数据元素。

以**元素**(数据元素的映象)

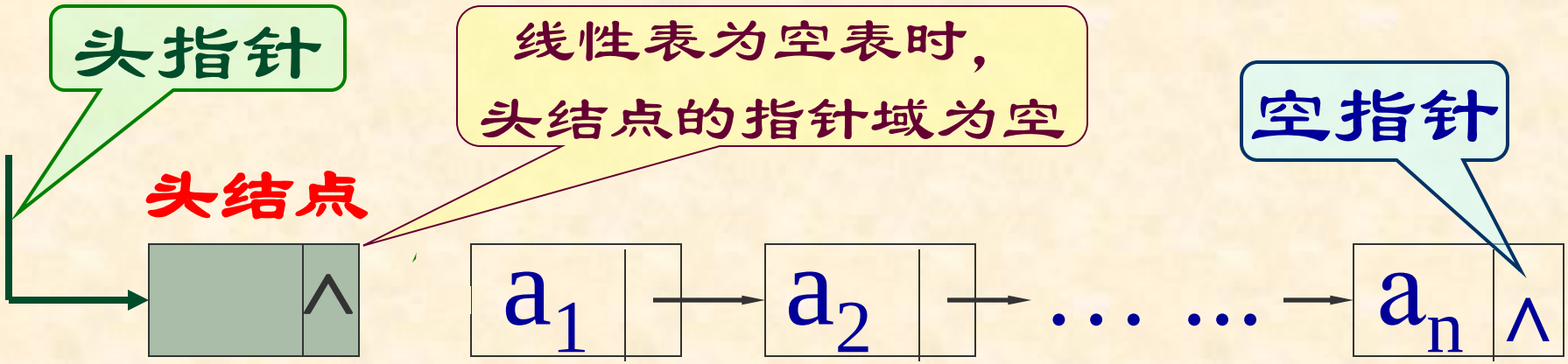
+ **指针**(指示后继元素存储位置)

= **结点**

(表示数据元素 或 **数据元素的映象**)

以“**结点的序列**”表示线性表

——称作**链表**



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。

有时为了操作方便，在第一个结点之前虚加一个“头结点”，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;
```

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

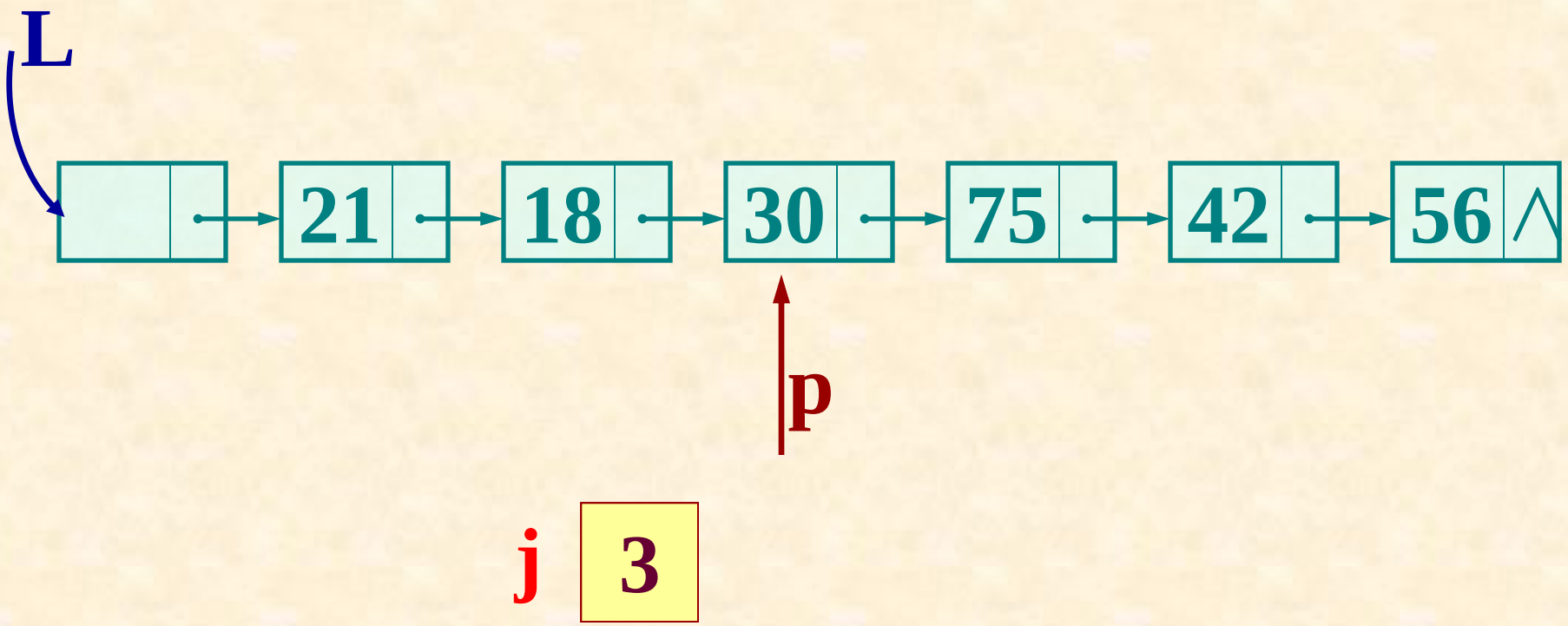
CreateList(&L, n)

// 生成含 ***n*** 个数据元素的链表

线性表的操作

GetElem(L, i, &e)

在单链表中的实现:



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 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：

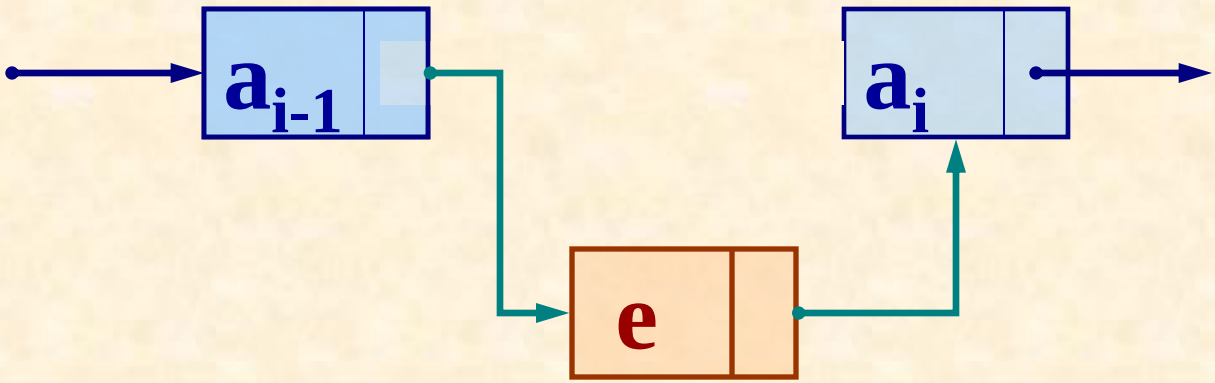
O(ListLength(L))

线性表的操作 **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+1或者小于1
      ......
} // LinstInsert_L
```

算法的时间复杂度为: O(ListLength(L))

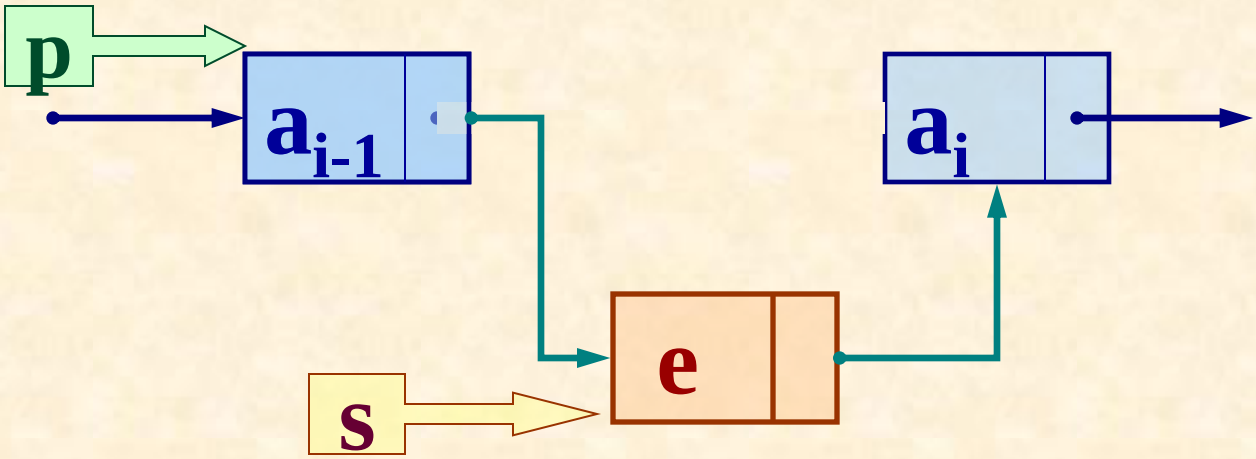
s = (LinkList) **malloc** (**sizeof** (LNode));

// 生成新结点

s->data = e;

s->next = p->next; p->next = s; // 插入

return OK;



线性表的操作ListDelete (&L, i, &e)

在链表中的实现:

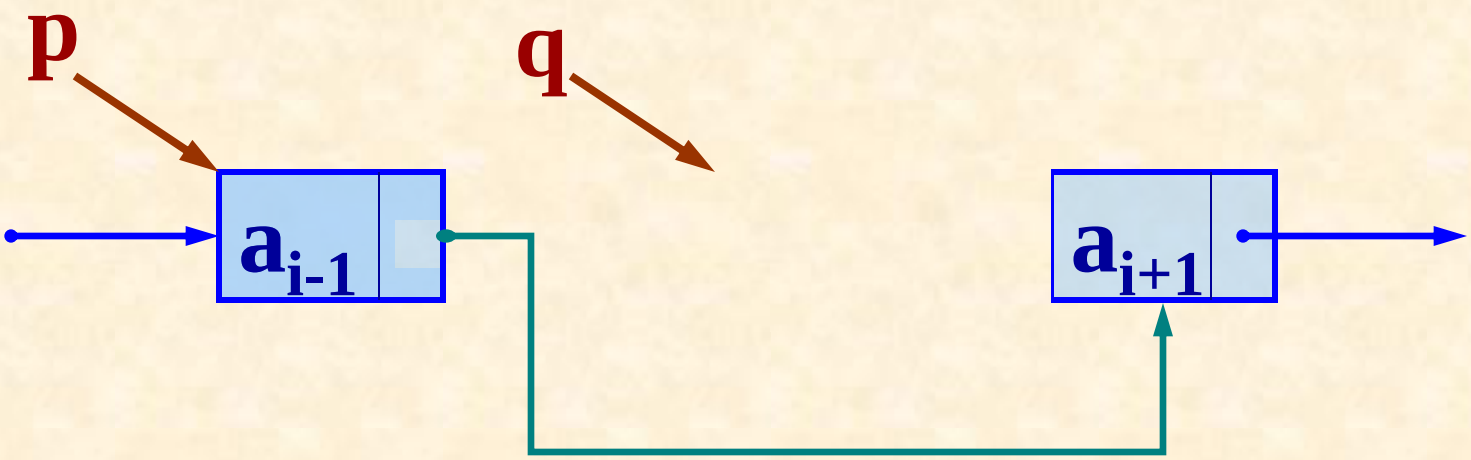
有序对〈a_{i-1}, a_i〉和〈a_i, a_{i+1}〉

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);***



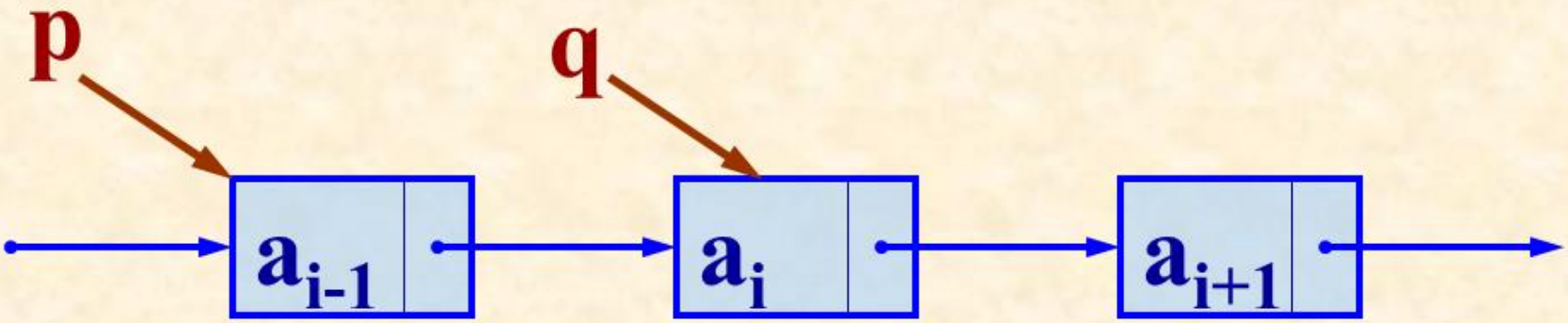
```
Status ListDelete_L(LinkList L, int i, ElemType &e) {
  // 删除以 L 为头指针(带头结点)的单链表中第 i 个结点
    p = L;    j = 0;
    while (p->next && j < i-1) {  p = p->next;   ++j; }
                                  // 寻找第 i 个结点，并令 p 指向其前趋
    if  (!(p->next) || j > i-1)
        return ERROR;  // 删除位置不合理
    q =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

填空题 3分

设置

如图所示，删除元素a_i的语句为 [填空1] [填空2] [填空3]



正常使用填空题需3.0以上版本雨课堂

作答

操作 **ClearList(&L)** 在链表中的实现:

```
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);
   }
} // ClearList
```

算法时间复杂度： **O(ListLength(L))**

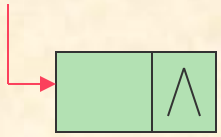
如何生成单链表？

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

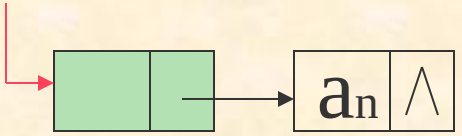
例如：逆位序输入 n 个数据元素的值，建立带头结点的单链表。

操作步骤：

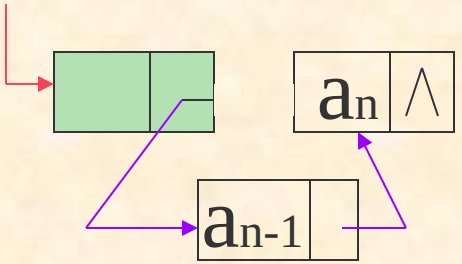
一、建立一个“空表”；



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))**

用上述定义的单链表实现线性表的操作时，存在的问题：

1. 单链表的表长是一个隐含的值；
2. 在单链表的最后一个元素之后插入元素时，需遍历整个链表；
3. 在链表中，元素的“位序”概念淡化，结点的“位置”概念加强。

改进链表的设置：

1. 增加“表长”、“表尾指针”和“当前位置的指针”三个数据域；
2. 将基本操作中的“位序 i ”改变为“指针 p ”。

四、一个带头结点的线性链表类型

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

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
```

```
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;
```

链表的基本操作：

{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); O(n)

// 改变当前指针指向第i个结点

Status LocateElem (LinkList L, ElemType e,

tatus (*compare)(ElemType, ElemType)); O(n)

// 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

// 元素的前驱，并返回OK; 否则返回ERROR

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;     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;
    e=q->data;  FreeNode(q);
    return OK;
} //DelAfter
```

{利用上述定义的线性链表可以完成
线性表的其他操作}

例一

```
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
   if(InsAfter(L,e)) return OK;//插入在第i-1个结点之后
   else return ERROR;
} // ListInsert_L
```

例二

```
Status MergeList_L(LinkList &Lc, LinkList &La,
                   LinkList &Lb ， int (*compare)
                                  (ElemType,ElemType))) {
  // 归并有序表 La 和 Lb , 生成新的有序表 Lc,
  //  并在归并之后销毁La 和 Lb,
  // compare 为指定的元素大小判定函数

        ……

} // MergeList_L
```

```
if ( !InitList(Lc))  return ERROR;    // 存储空间分配失败
LocatePos (La, 0);  LocatePos (Lb, 0); // 当前指针指向头结点
if ( DelAfter( La, e))  a = e;        // 取得 La 表中第一个元素 a
else a = MAXC;       // MAXC为常量最大值
if ( DelAfter( Lb, e))  b = e;        // 取得 Lb 表中第一个元素 b
else b = MAXC;       // a 和 b 为两表中当前比较元素
while (!( a=MAXC && b=MAXC)) {  // La 或 Lb 非空
        … …
}
DestroyList(La);  DestroyList(Lb);  // 销毁链表 La 和 Lb
return OK;
```

```
if ((*compare)(a, b) <=0) {  //  a≤b
    InsAfter(Lc, a);
    if ( DelAfter( La, e1) )    a = e1;
    else a = MAXC;
}
else {   //  a＞b
    InsAfter(Lc, b);
    if ( DelAfter( Lb, e1) )    b = e1;
    else  b = MAXC;
}
```

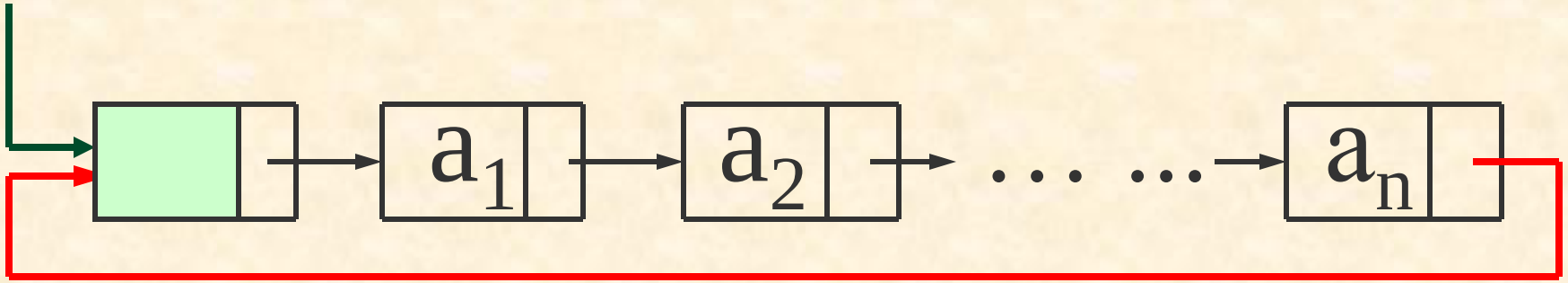
五、其它形式的链表

1. 双向链表

```
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

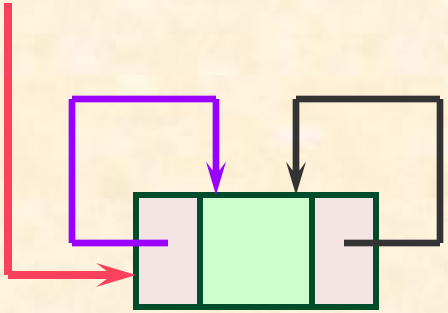
最后一个结点的指针域的指针又指回第一个结点的链表



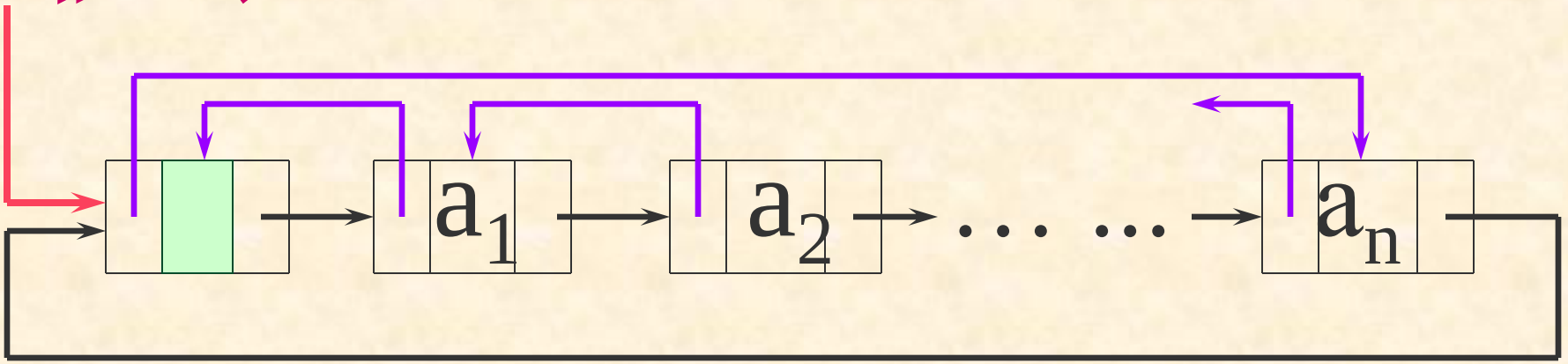
和单链表的差别仅在于，判别链表中最后一个结点的条件不再是“后继是否为空”，而是“后继是否为头结点”。

双向循环链表

空表



非空表

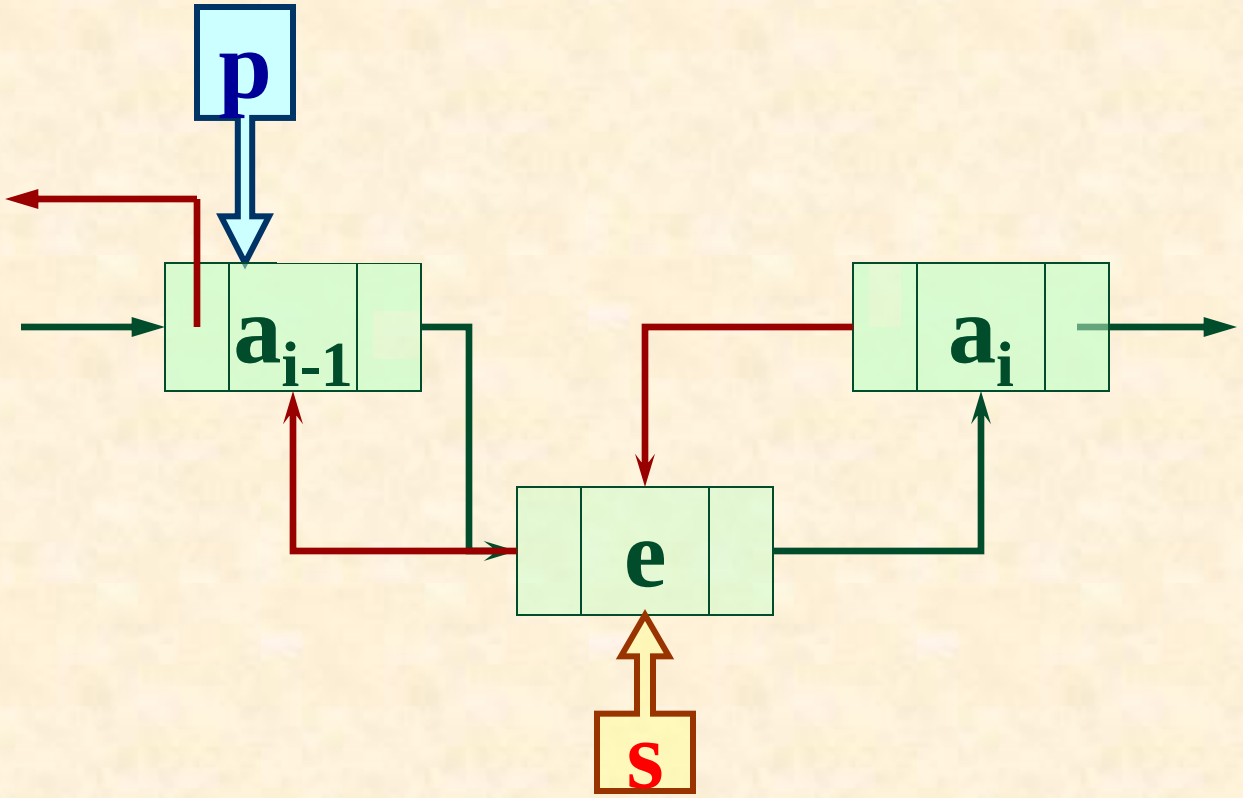


双向链表的操作特点：

“查询” 和单链表相同。

“插入” 和“删除”时需要同时修改两个方向上的指针。

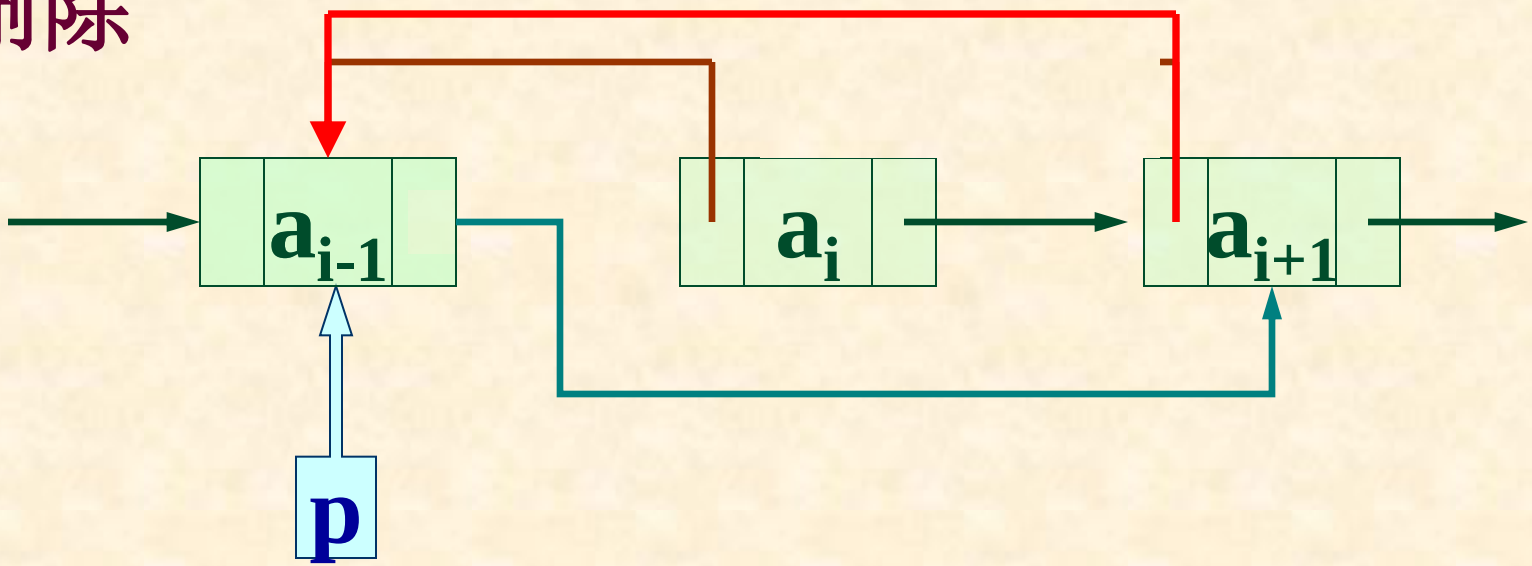
插入



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;

主观题 10分

设置

在_____链表中，从任何一结点出发都能访问到表中的所有结点。

正常使用主观题需2.0以上版本雨课堂

作答

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：$S = \{ x_i | x_i \in \text{OrderedSet}, i=1,2,\ldots,n, n\geq 0 \}$

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：

Compare是一个有序判定函数

…… ……

LocateElem(L, e, &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, 45)

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。

```
void union(List &La, List Lb) {// Lb 为线性表
  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
```

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)

2.4 一元多项式的表示

一元多项式

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

在计算机中，可以用一个线性表来表示：

$$P = (p_0, p_1, \cdots, p_n)$$

但是对于形如

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

的多项式，上述表示方法是否合适？

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，

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

可以下列线性表表示：

$$（（p_1, e_1）, (p_2, e_2), \text{---}, (p_m, e_m)）$$

例如:

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

可用线性表

((7, 3), (-2, 12), (-8, 999))

表示

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

且$\mathbf{a_{i-1}}$中的指数值＜$\mathbf{a_i}$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)

… …

} ADT Polynomial

一元多项式的实现：

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

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
} term, ElemType;
```

```
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); 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)()) )
            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。

```
Status AddPolyn ( polynomial &Pc,
                  polynomial &Pa, polynomial &Pb) {
  // 利用两个多项式的结点构成“和多项式” Pc = Pa + Pb
                        … …
  if (DelAfter(Pa, e1))  a=e1.expn   else  a=MAXE;
  if (DelAfter(Pb, e2))  b=e2.expn   else  b=MAXE;
  while (!(a=MAXE && b=MAXE)) {
              … …
   }
              … …
} // AddPolyn
```

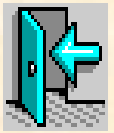
```
switch (*cmp(e1, e2)) {
    case -1: {  // 多项式PA中当前结点的指数值小
                        … …       break; }
    case 0: {   //  两者的指数值相等
        e1.coef= a.coef + b.coef ;
        if ( e1.coef != 0.0 ) InsAfter(Pc, e1);
                    … …                 break;
    }
    case 1: {  //多项式PB中当前结点的指数值小
                        … …       break; }
}
```

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其适用场合。



单选题 1分

设置

对线性表进行二分查找时，要求线性表必须

A 以顺序方式存储

B 以链接方式存储

C 以顺序方式存储，且数据元素有序

D 以链接方式存储，且数据元素有序

提交