福州大学至诚学院
FUZHOU UNIVERSITY ZHICHENG COLLEGE

高级语言程序设计（C语言与数据结构）

杨雄

83789047@qq.com

复　习

数据

数据元素　数据元素　数据元素

数据项1　数据项2　数据项1　数据项2　数据项1　数据项2

★数据项是数据的不可分割的最小单位。

复　习

对数据结构，主要讨论如下三方面的问题：

① ★数据的逻辑结构

线性表、树、图。

② ★数据的存储结构(物理结构)：

顺序存储结构、链式存储结构；

③数据的运算

即对数据施加的操作。定义在数据的逻辑结构上的抽象的操作。

复　习

★推导大O阶：

1.用**常数1**取代运行时间中的所有**加法常数**。

2.在修改后的运行次数函数中，只**保留最高阶项**。

3.如果最高阶项存在且不是1，则**去除与这个项相乘的常数**。

得到的结果就是大O阶。

2.线性表

01 线性表的类型定义

02 线性表的顺序表示与实现

03 线性表的链式表示与实现

04 一元多项式的表示及相加

学习目的

- 理解线性表的定义和特点；
- 理解线性表的**顺序表**和**链表**的特点，掌握在这两种存储结构上各种基本运算的**实现算法**以及**效率**的分析
- 掌握一元多项式的**表示**及相加的方法与算法

Part.1

2.1 线性表的定义

2.1 线性表的定义

例1： 英文字母表

(A, B, C, ……, Z)

例2：

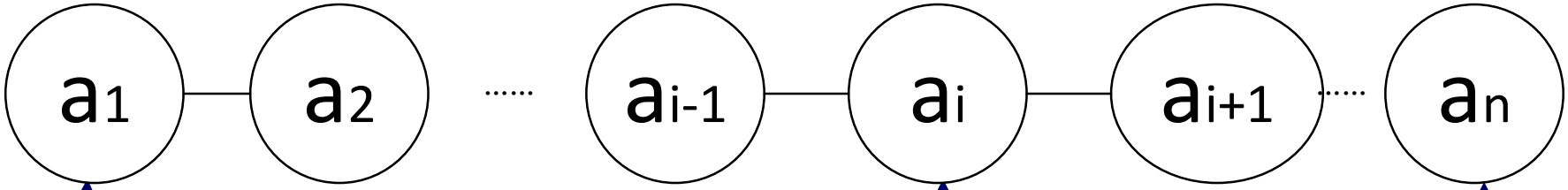
学号	姓名	性别	年龄
1	张三	男	18
2	李四	女	17
3	王五	女	19
……	……	……	……

2.1 线性表的定义

- 线性表：零个或多个数据元素组成的有限序列

线性表是一个灵活的数据结构，长度可以根据需要增长或缩短，对表中的数据元素不仅进行访问，还可以进行插入和删除等。

$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$



- n为线性表的表长，[illegible]空表

第一个元素（没有前驱）

[illegible]线性表中的

第i个元素（有唯一的前驱和唯一的后继）

最后一个元素（没有后继）

2.1 线性表的定义

- 线性结构基本特征：
  - 存在唯一的“第一个”数据元素
  - 存在唯一的“最后一个”数据元素
  - 除第一个外，每个数据元素均有且只有一个前驱元素
  - 除最后一个外，每个数据元素均有且只有一个后继元素。

2.1 线性表的定义

ADT List{

数据对象：**D = { a_i | $a_i \in$ ElemSet, i=1,2,…,n, n≧0 }**

数据关系：**R = {<a_{i-1}, a_i> | a_{i-1}, $a_i \in$ D, i=2,3,…,n }**

基本操作：

InitList(&L);

DestroyList(&L);

ListEmpty(&L);

ListLength(L);

GetElem(L, i, &e);

ListInsert (L, i, &e);

……

} ADT List

- 线性表的抽象数据类型表示

2.1 线性表的定义

➢ 基本操作(一)

抽象数据类型必备的两个操作

- InitList(&L)
  - 操作结果:构造一个空的线性表L。
- DestroyList(&L)
  - 初始条件:线性表L已经存在。
  - 操作结果:销毁线性表L。
- ClearList(&L)
  - 初始条件: 线性表L已存在。
  - 操作结果: 将L重置为空表

➢ 基本操作(二)

- ListLength(L)
  - 初始条件:线性表L已经存在。
  - 操作结果:返回线性表L中的数据元素个数
- GetElem(L,i,&e)
  - 初始条件:线性表L已经存在,1 ⩽ i ⩽ ListLength(L)。
  - 操作结果:用e返回线性表L中第i个数据元素的值。
- LocateElem(L,e)
  - 初始条件:线性表L已经存在。
  - 操作结果:返回L中第1个与e相等的数据元素的位序。若这样的数据元素不存在则返回值为0。

➢ 基本操作(三)

- PriorElem(L,cur_e,&pre_e)
  - 初始条件: 线性表L已经存在。
  - 操作结果: 若cur_e是L的数据元素,且不是第一个,则用pre_e返回它的前驱；否则操作失败， pre_e无意义。

- NextElem(L,cur_e,&next_e)
  - 初始条件: 线性表L已经存在。
  - 操作结果:若cur_e是L的数据元素, 且不是最后一个,则用next_e返回它的后继;否则操作失败, next_e无意义。

2.1 线性表的定义

➢ 基本操作(四)

- 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FALSE

- ListTraverse(&L, visited())
  - 初始条件:线性表已经存在
  - 操作结果：依次对L的每个数据元素调用函数visit()

➢ 基本操作(五)

- ListInsert(&L, i, e)
  - 初始条件:线性表L已经存在，1 ⩽ i ⩽ ListLength(L)+1。
  - 操作结果: 在L的第i个位置之前插入新的数据元素e,L的长度加一。
- 插入前（长度为n）:(a_1,a_2,…, a_{i-1} , a_i,…,a_n)
- 插入后（长度为n+1）： (a_1,a_2,…, a_{i-1},e,a_i, …,a_n)

➢ 基本操作(六)

利用上述定义的线性表可以实现其它更复杂的操作。

- ListDelete(&L,i,&e)
  - 初始条件:线性表L已经存在，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的长度减一。
- 删除前（长度为n）：$(a_1,a_2,\ldots, a_{i-1}, a_i, a_{i+1}, \ldots,a_n)$
- 删除后（长度为n-1）：$(a_1,a_2,\ldots, a_{i-1}, a_{i+1}, \ldots,a_n)$

➢ 线性表的合并--示例1：

- 假设：有两个集合A 和 B ，分别用两个线性表 L_A 和 L_B 表示，现要求一个新的集合A = A∪B。
- 如：集合La=(2,5,7,9), Lb=(4,5,6,7)

  合并结果:La=(2,5,7,9,4,6)

■ **算法思想:**

对线性表作如下操作：扩大线性表L_A，将存在于线性表L_B 中而不存在于线性表 L_A 中的数据元素插入到线性表 L_A 中去。

- 线性表的合并--示例1：

  - 实现步骤

    - 1. 从线性表L_B中依次查看每个数据元素；

      GetElem(L_B, i ,&e)

    - 2. 依次在线性表L_A中进行查找；

      LocateElem(L_A, e)

    - 3. 若不存在，则插入之。

      ListInsert(L_A, n+1, e)

■ 实现代码

```
/*将所有的线性表Lb中但不在La中的数据元素插入到La中*/
void union(List &La, List Lb){
        int La_len, Lb_len, i;
        ElemType e;              //声明与La和Lb相同的数据元素*/
        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)) //La中不存在和 e相同的数据元素，
              ListInsert(La,++La_len, e);  /*插入*/
   }
 }
```

2.1 线性表的定义

➢ 线性表的合并--示例2：

- 已知非递减线性表La、Lb,将所有La和Lb中的数据元素归并到Lc中, 使Lc 的数据元素也是非递减的。
- 如:La=(3,5,8,11), Lb=(2,6,8,9,11,15,20)

  合并结果: Lc=(2,3,5,6,8,8,9,11,11,15,20)

■ 算法思想:

将La、Lb两表中的元素逐一按序加入到一个新表Lc中。

- 线性表的合并--示例2：

■ 实现步骤

- 1. 分别从La和Lb中取得当前元素a_i和b_j；
- 2. 若$a_i \leqslant b_j$，则将a_i插入到Lc中，否则将b_j插入到Lc中。

■ 实现代码

```
// 本算法将非递减的有序表 La 和 Lb 归并为 Lc, La 和 Lb 均不空
void MergeList(List La, List Lb, List &Lc) {
  InitList(Lc);  // 构造空的线性表 Lc
  int i = j = 1, k = 0, La_len, Lb_len;
  ElemType a, b;
  La_len = ListLength(La);
  Lb_len = ListLength(Lb);
  while ((i<=La_len) && (j<=Lb_len)){
    GetElem(La,i,&a); GetElem(Lb,j,&b);
    if(a<=b){
      ListInsert(Lc, ++k, a); ++i;
    }
    else {
      ListInsert(Lc, ++k, b); ++j;
   }}
```

```
  while (i<=La_len) {
    GetElem(La,i++,&a);
    ListInsert(Lc,++k, a);
  } // 当La不空时插入 La 表中剩余元素
  while (j<=Lb_len) {
    GetElem(Lb,j++,&b);
    ListInsert(Lc,++k,b);
   } // 当Lb不空时插入 Lb 表中剩余元素
} // merge_list
```

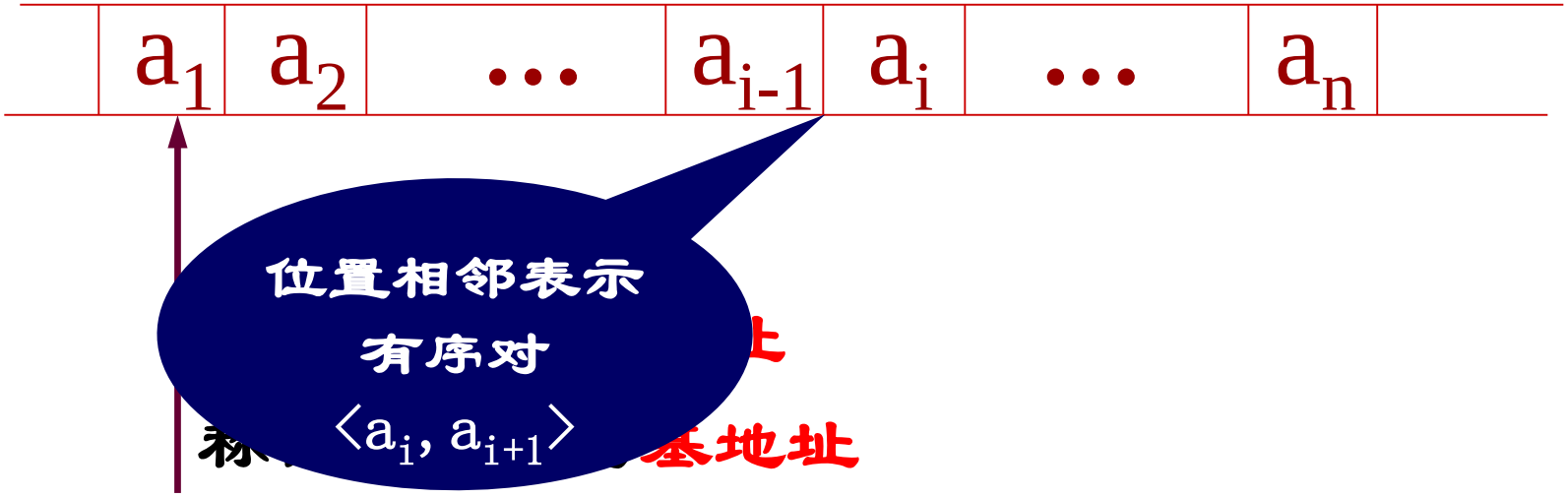
Part.2

2.2 线性表的顺序表示与实现

2.2 线性表的顺序表示与实现

- 线性表$(a_1, a_2, ..., a_i, a_{i+1}, ..., a_n)$的顺序表示：用一组地址连续的存贮单元依次存储线性表的数据元素。

线性表的顺序存储示意图为：



■ 实现：可用C语言的一维数组实现

顺序表类型定义

```
#define LIST_INIT_SIZE 100  // 线性表存储空间的初始分量
#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;
```

2.2 线性表的顺序表示与实现

- **线性表的基本操作在顺序表中的实现：**

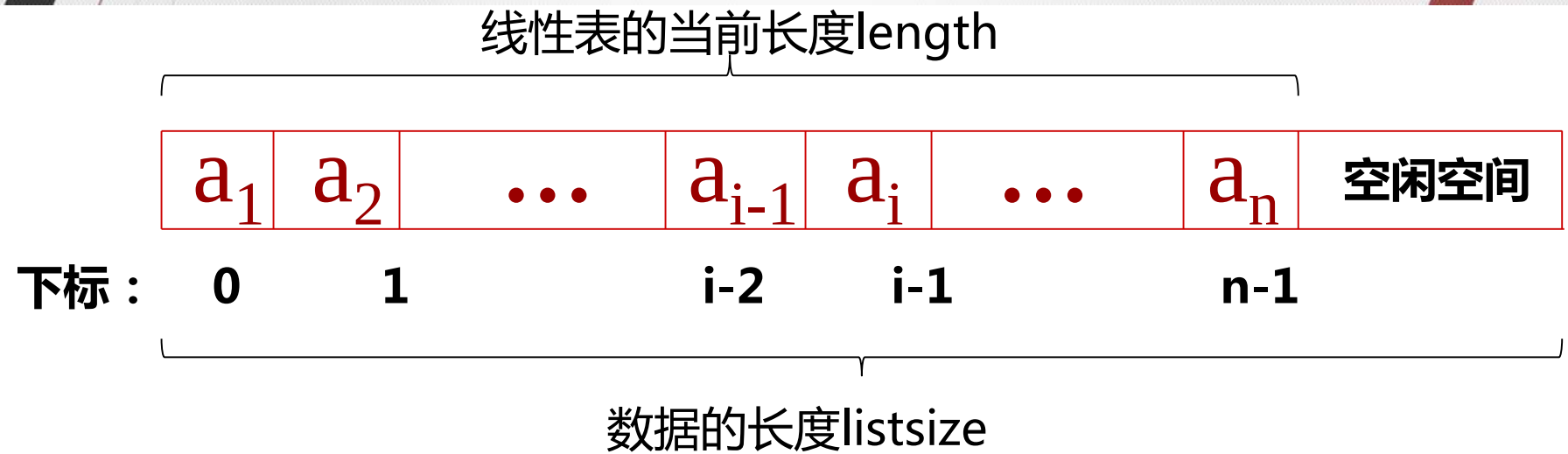
  - InitList(&L)　// 结构初始化
  - GetElem(L, i, &e) //获得元素
  - LocateElem(L, e)　// 查找元素
  - ListInsert(&L, i, e)　// 插入元素
  - ListDelete(&L, i,&e)　// 删除元素

- **操作的实现——InitList(&L)**

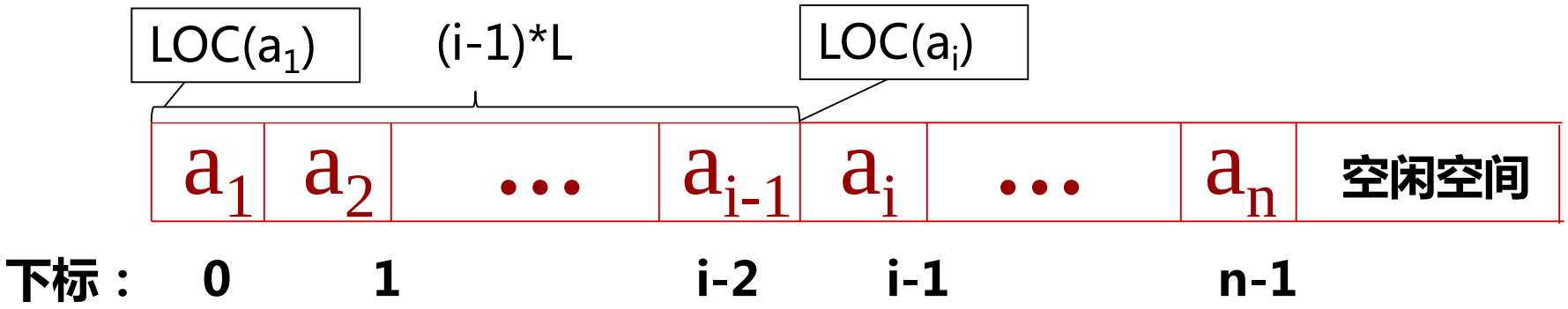
```
Status InitList_Sq( SqList& L ) {
    // 构造一个空的线性表
    L.elem = (ElemType*) malloc
   (LIST_INIT_SIZE*sizeof (ElemType));
    if (!L.elem) exit (OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
} // InitList_Sq
```

算法时间复杂度：O(1)

2.2 线性表的顺序表示与实现



- 元素地址计算方法：
  - $LOC(a_{i+1}) = LOC(a_i) + L$
  - $LOC(a_i) = LOC(a_1) + (i-1)*L$



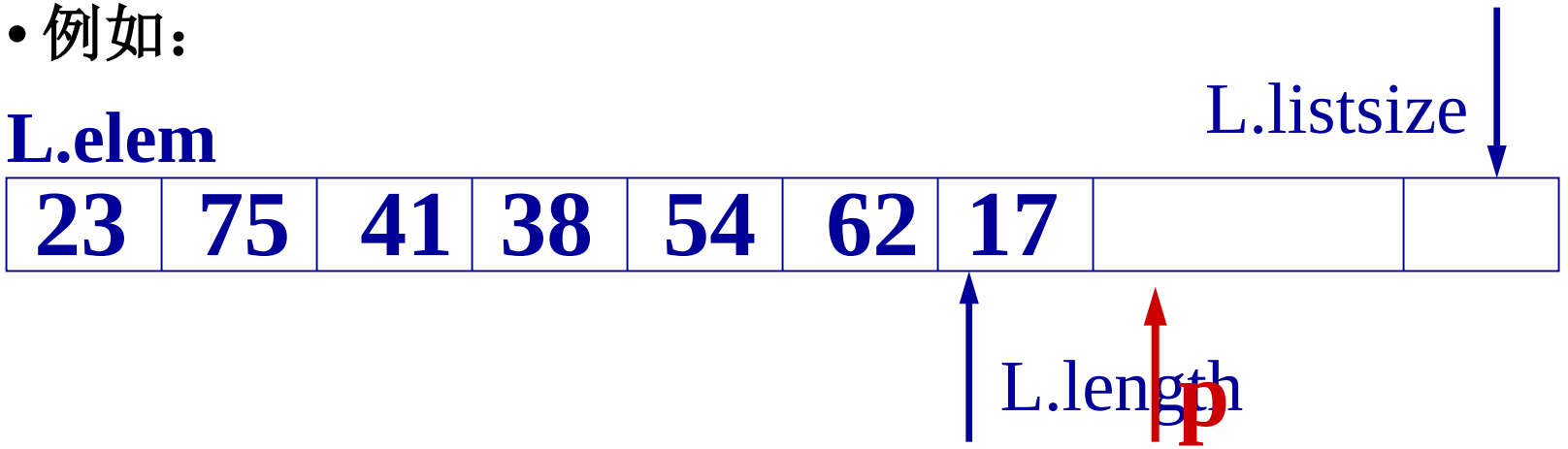
➢ **操作的实现——GetElem(L, I, &e)**

```
/* 初始条件：顺序线性表L已存在, i≤i≤ListLength(L) */
/*操作结构：用e返回L中第i个数据元素的值 */
Status GetElem_Sq(SqList L, int i, ElemType &e)
{
    if (L.length == 0 || i < 1 || i > L.length)
        return ERROR;
    e = L.elem[i-1];
    return OK;
}
```

算法时间复杂度：O(1)

➢ **操作的实现——LocateElem(L, e)**

• 例如：



i 8

e = 38 50

基本操作是：
将顺序表中的元素逐个和给定值 e 相比较。

➢ 操作的实现——LocateElem(L, e)

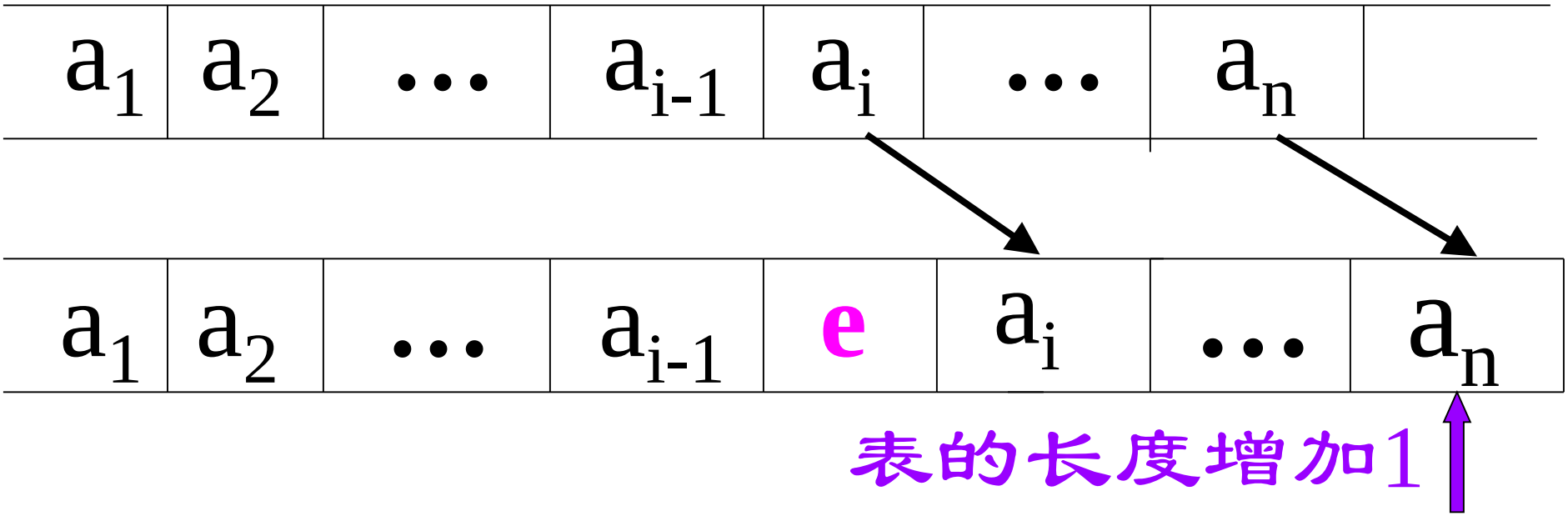
```
// 在顺序表中查询第一个与数据元素e相等的数据元素，
// 若存在，则返回它的位序，否则返回 0
int LocateElem_Sq(SqList L, ElemType e) {
    i = 1;          // i 的初值为第 1 元素的位序
    p = L.elem;     // p 的初值为第 1 元素的存储位置
   while (i <= L.length &&(*p!=e))
   {
      ++i;
      p++;
    }
   if (i <= L.length)  return i;
   else  return 0;
} // LocateElem_Sq
```

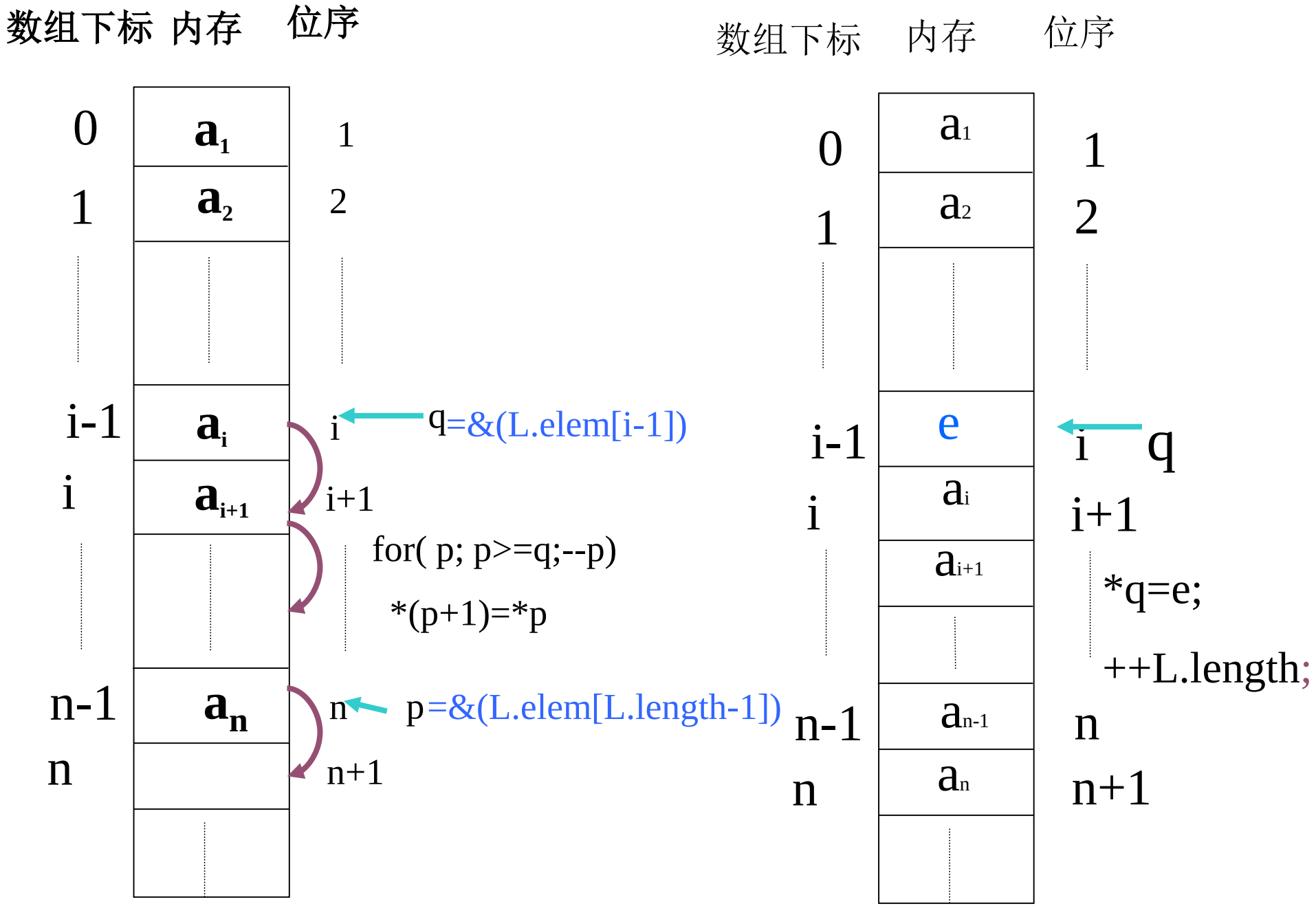
算法时间复杂度：O(n)

➢ **操作的实现——ListInsert_Sq(&L, i, e)**

- **例如：** $(a_1, \ldots, a_{i-1}, a_i, \ldots, a_n)$ **改变为**
  $(a_1, \ldots, a_{i-1}, e, a_i, \ldots, a_n)$

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




数组下标 内存 位序
0
1
i-1
i
n-1
n
a_1
a_2
a_i
a_{i+1}
a_n
1
2
i
i+1
n
n+1
q=&(L.elem[i-1])
for(p; p>=q;--p)
*(p+1)=*p
p=&(L.elem[L.length-1])
数组下标 内存 位序
0
1
i-1
i
n-1
n
a_1
a_2
e
a_i
a_{i+1}
a_{n-1}
a_n
1
2
i
q
i+1
*q=e;
++L.length;
n
n+1

2.2 线性表的顺序表示与实现

- **操作的实现——ListInsert_Sq(&L, i, e)**

  - **算法思想:**
    - **如果插入位置不合理，则抛出异常**
    - **如果线性表长度大于等于数组长度，则动态增加容量**
    - **从最后一个元素开始向前遍历到第i 个位置，分别将它们都向后移动一个位置**
    - **将要插入元素填入位置i处**
    - **表长加1**

➢ 操作的实现——ListInsert_Sq(&L, i, e)

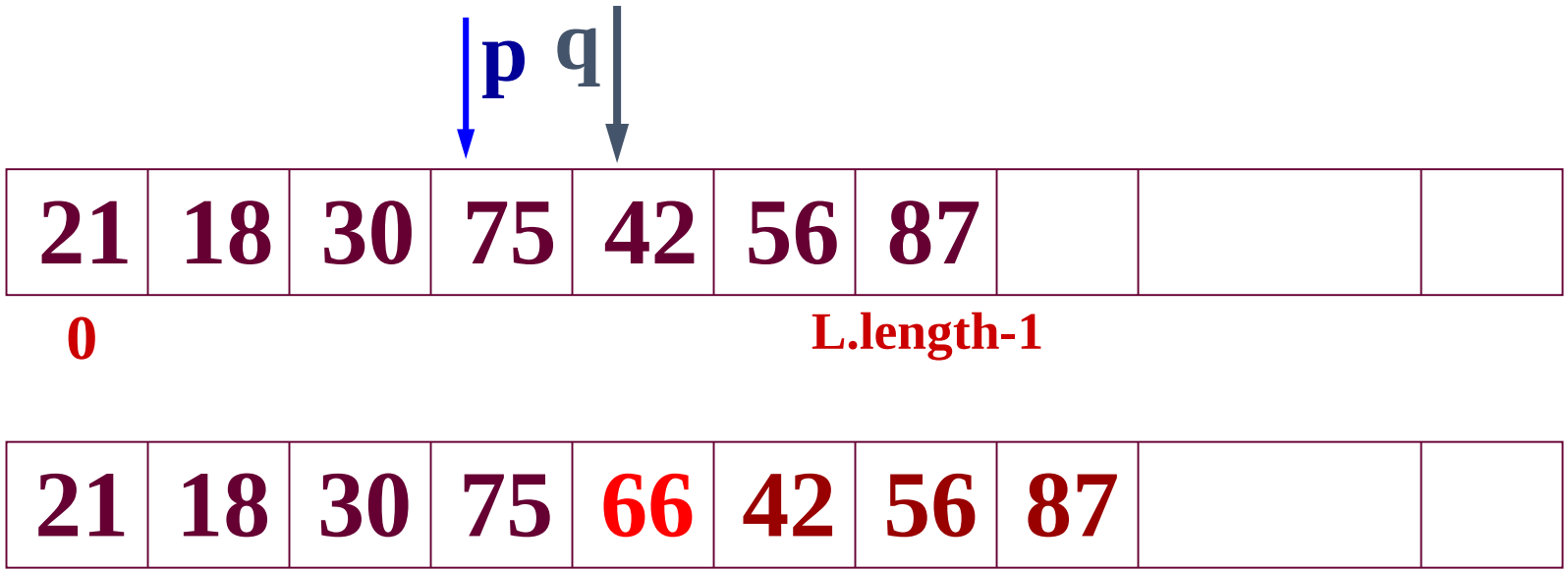
```
Status ListInsert_Sq(Sqlist &L,int i,ElemType e)
{
 /*在顺序线性表L的第i个位置之前插入新的元素e
  i的合法值为1 ≤ i ≤ ListLength_Sq(L) + 1 */
 ElemType *q,*p,*newbase;
 if(i<1 || i>L.length+1) return ERROR;
 if(L.length>=L.listsize){
        newbase=(ElemType*)realloc(L.elem,
        (L.listsize+LISTINCREMENT)*sizeof(ElemType ));
    // 存储分配失败
   if(!newbase) exit(OVERFLOW);
   L.elem=newbase;
   L.listsize+=LISTINCREMENT;
 }
```

➢ 操作的实现——ListInsert_Sq(&L, i, e)

```
    q=&(L.elem[i-1]); // q为插入位置
    // 插入位置及之后的元素后移
    for( p=&(L.elem[L.length-1]); p>=q; --p)
        *(p+1)=*p;
    *q=e; // 插入e
    ++L.length;
    return OK;
}
```

- 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;
```



- 操作的实现——ListInsert_Sq(&L, i, e)

■算法时间复杂度T(n)

◆设Pi是在第i个元素之前插入一个元素的概率，则在长度为n的线性表中插入一个元素时，所需移动的元素次数的平均次数为：

$$Eis = \sum_{i=1}^{n+1} P_i (n - i + 1)$$

$$若认为 P_i = \frac{1}{n+1}$$

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

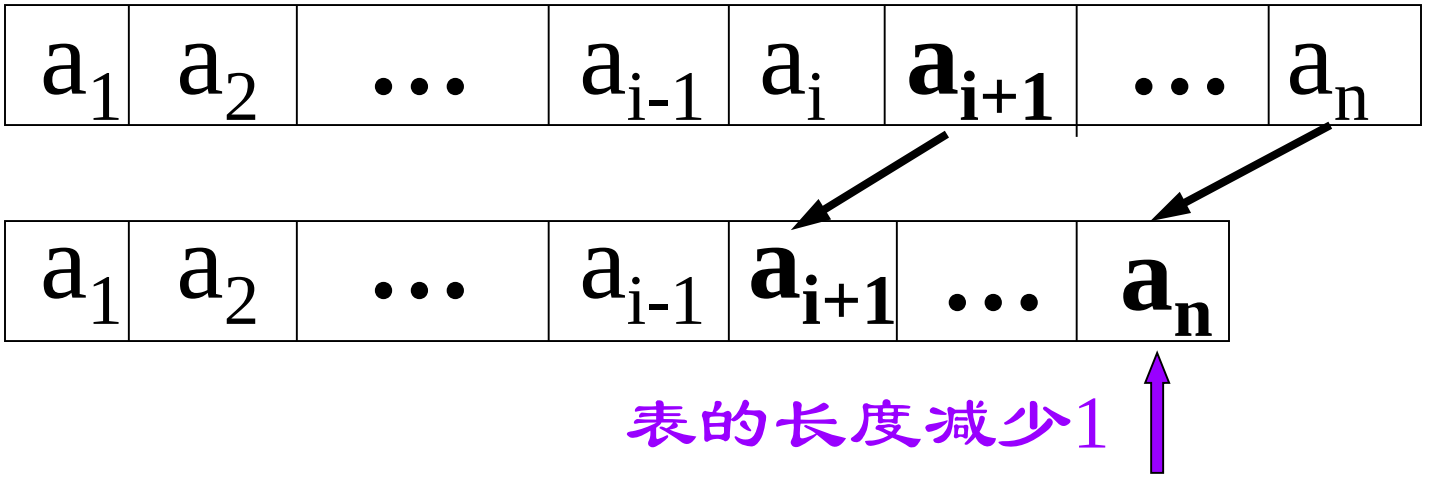
$$\therefore T(n) = O(n)$$

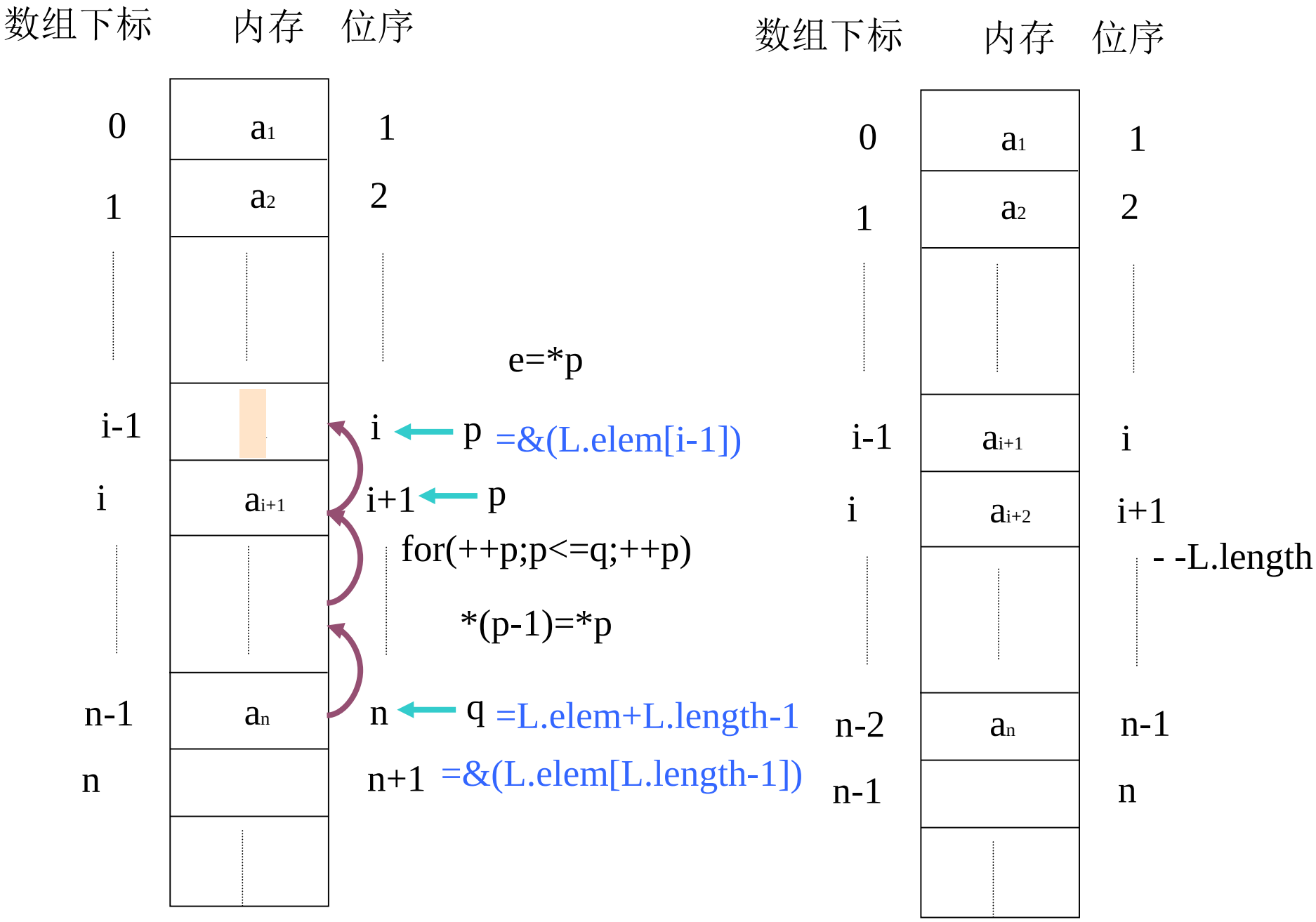
- **操作的实现——ListDelete(&L, i, &e)**

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

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

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




数组下标
内存
位序
0
1
i-1
i
n-1
n
a1
a2
ai+1
an
1
2
i
i+1
n
n+1
e=*p
p =&(L.elem[i-1])
p
for(++p;p<=q;++p)
*(p-1)=*p
q =L.elem+L.length-1
=&(L.elem[L.length-1])
数组下标
内存
位序
0
1
i-1
i
n-2
n-1
a1
a2
ai+1
ai+2
an
1
2
i
i+1
n-1
n
- -L.length

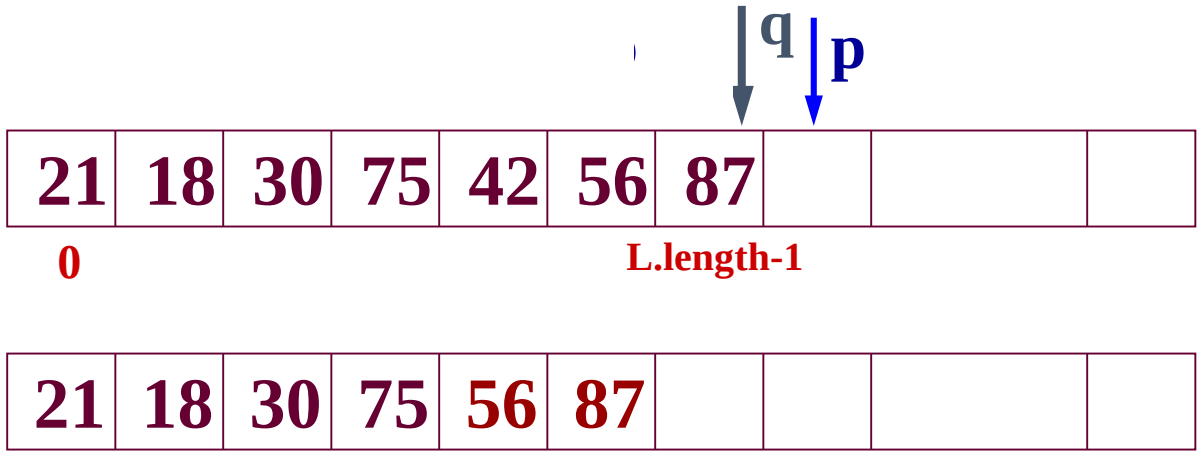
- **操作的实现——ListDelete(&L, i, &e)**

  - **算法思想:**
    - **如果删除位置不合理，则抛出异常**
    - **取出删除元素**
    - **从删除元素位置开始遍历到最后一个元素位置，分别将它们都向前移动一个位置**
    - **表长减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
```

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;
```



➢ **操作的实现——ListDelete(&L, i, &e)**

- 算法评价
  - 设Qi是删除第i个元素的概率，则在长度为n的线性表中删除一个元素所需移动的元素次数的平均次数为：

$$E_{de} = \sum_{i=1}^{n} Q_i (n-i)$$

若认为 $Q_i = \frac{1}{n}$

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

$\therefore T(n) = O(n)$

2.2 线性表的顺序表示与实现

优点

- **逻辑相邻，物理相邻，无须为表示表中元素之间的逻辑关系而增加额外的存储空间**
- **可以快速地存取表中任一位置的元素**

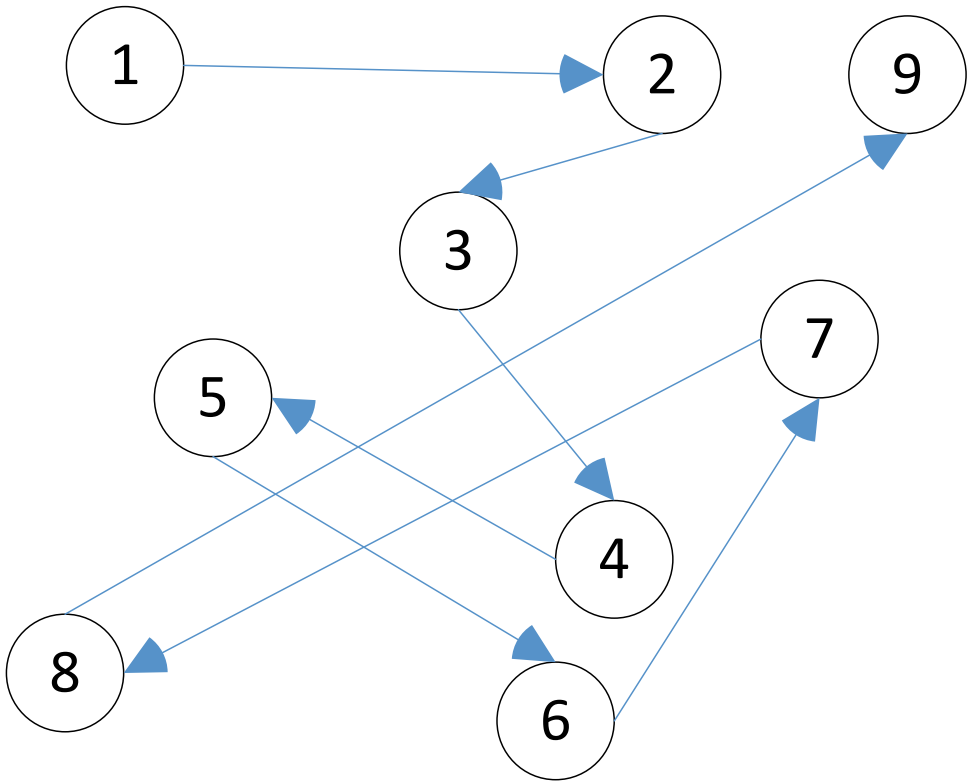
缺点

- **插入和删除操作需要移动大量元素**
- **预先分配空间需按最大空间分配，利用不充分**
- **表容量难以扩充**

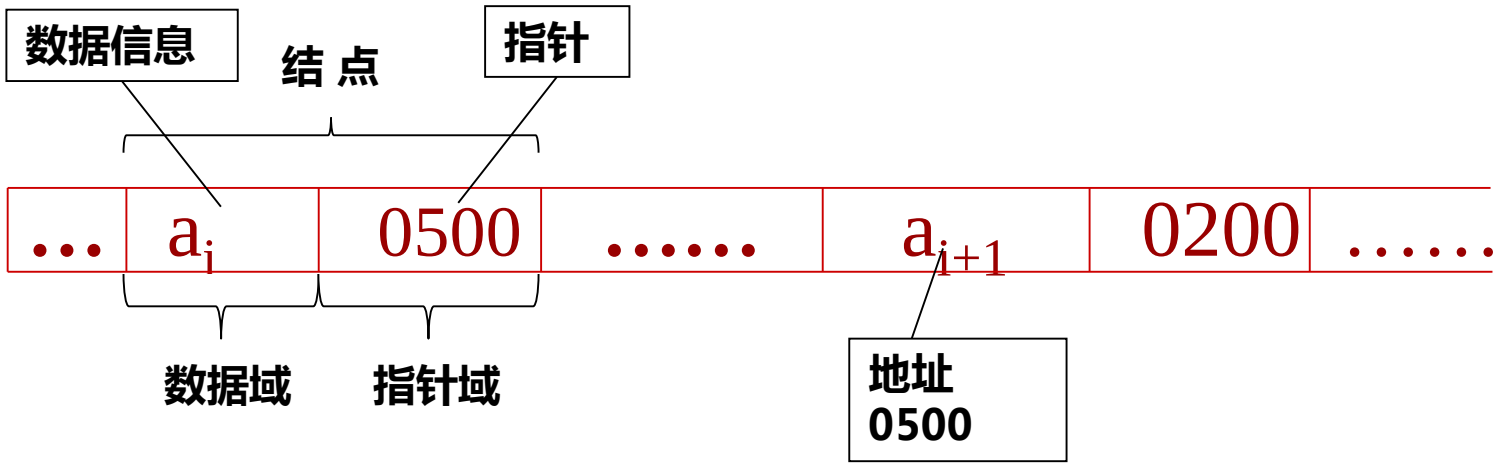
Part.3

2.3 线性表的链式表示与实现

2.3 线性表的链式表示与实现

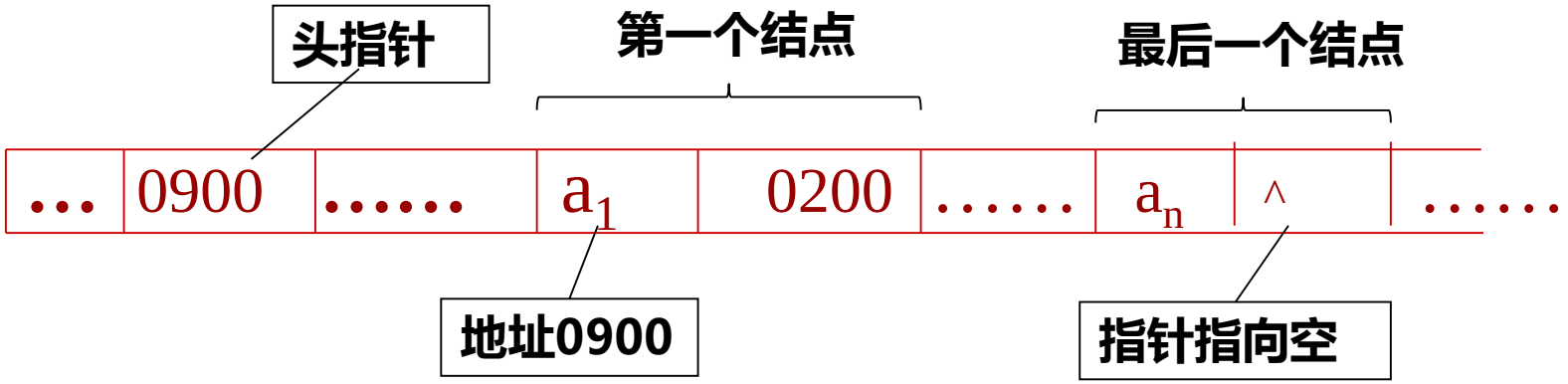


2.3 线性表的链式表示与实现



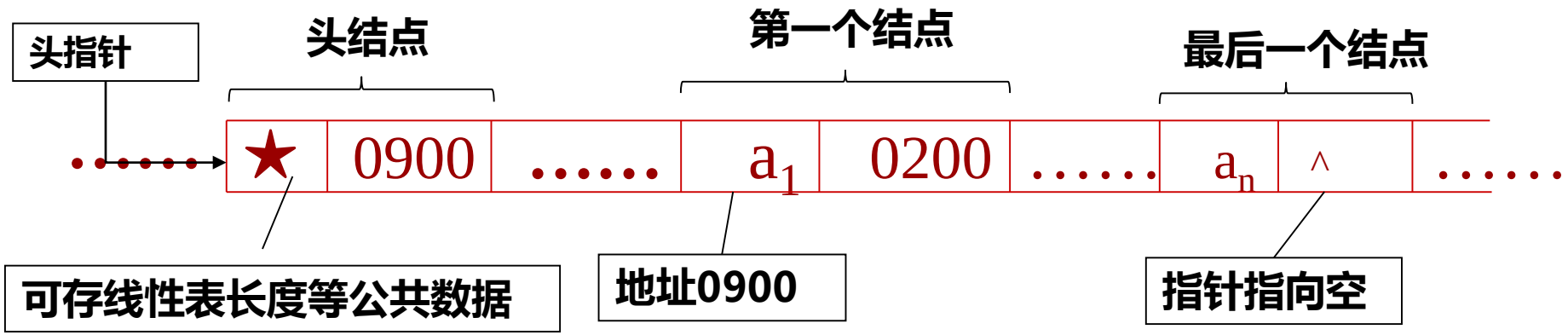
- 线性表(a_1,a_2,…,a_i,a_{i+1},…,a_n)的链式表示:可以用一组任意的存储单元存储线性表的数据元素。存储单元即可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。
- 单链表：链表的每个结点只包含一个指针域。

2.3 线性表的链式表示与实现



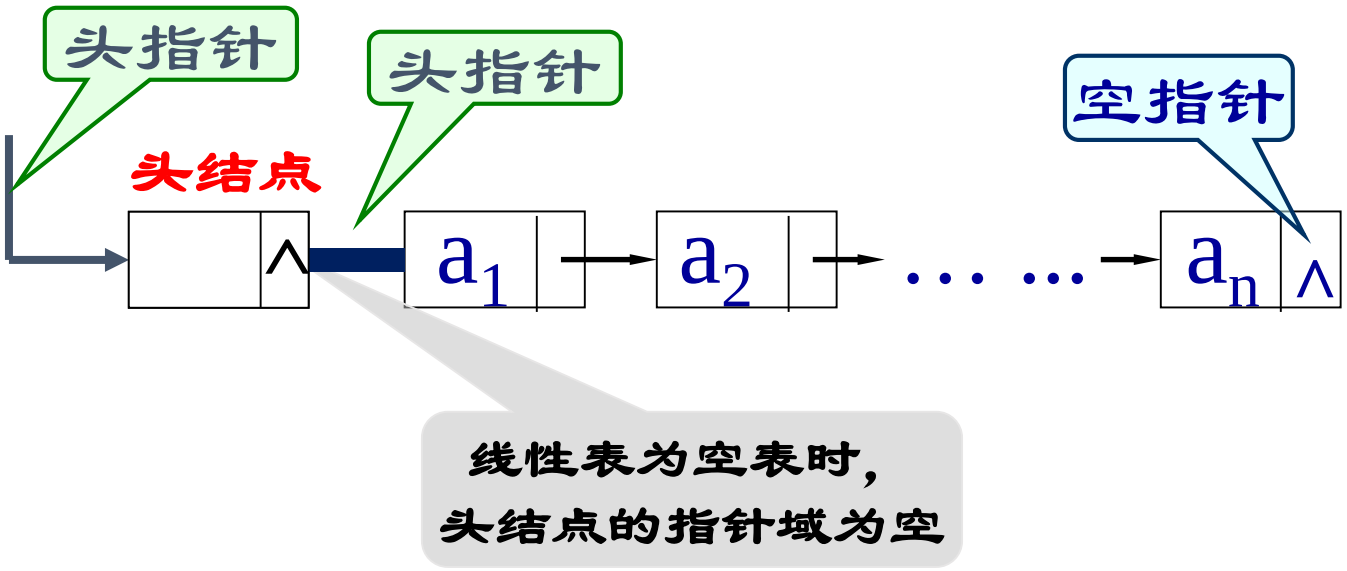
- 链表中第一个结点的存储位置叫做头指针。

2.3 线性表的链式表示与实现



- 头结点：在单链表的第一个结点前附设一个结点。

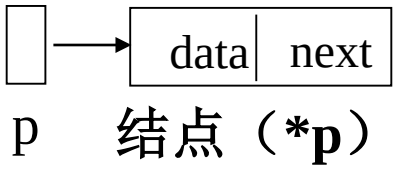
2.3 线性表的链式表示与实现

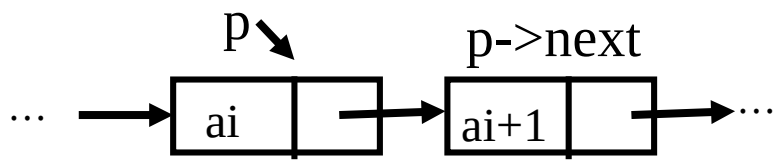


2.3 线性表的链式表示与实现

```
typedef struc LNode{                    // 定义单链表结点
    ElemType    data;
    struct LNode   *next ;    // 指向后继的指针域
}LNode, *LinkList;
```

LNode *p;





(*p)表示p所指向的结点

(*p).data⇔p->data表示p指向结点的数据域

(*p).next⇔p->next表示p指向结点的指针域

2.3 线性表的链式表示与实现

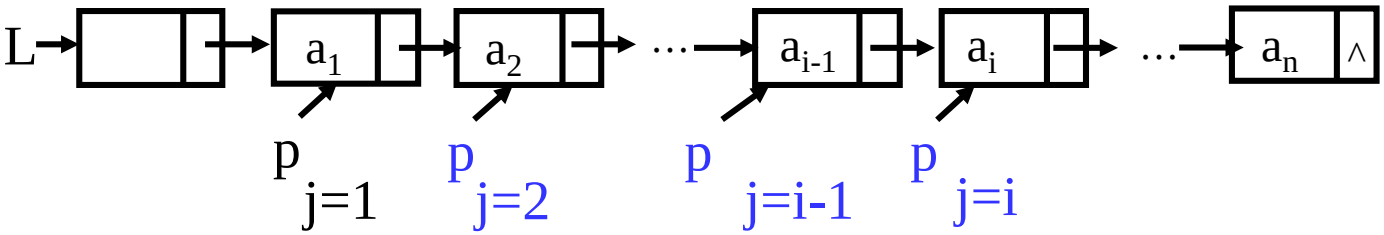
单链表操作的实现：

- GetElem(L, i, &e)　　//取第i个数据元
- ListInsert(&L, i, e) //插入数据元
- 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

2.3 线性表的链式表示与实现

➢ 单链表表的查找操作GetElem_L(L,i,&e)

- **算法思想:**
  - **声明一个指针p指向链表第一个结点，初始化j从1开始；**
  - **当j<i时，就遍历链表，让p的指针向后移动，不断指向下一结点，j累加1**
  - **若到链表末尾p为空，则说明第i个结点不存在；**
  - **否则查找成功，返回结点p的数据。**



while (p&&j<i) {p=p→ next; ++j;}

2.3 线性表的链式表示与实现

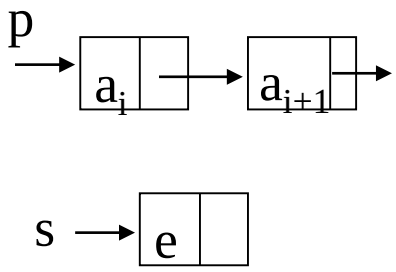
```
/* 初始条件：顺序线性表L已存在，1≤i≤ListLength(L) */
/* 操作结果：用e返回L中第i个数据元素的值 */

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(n)

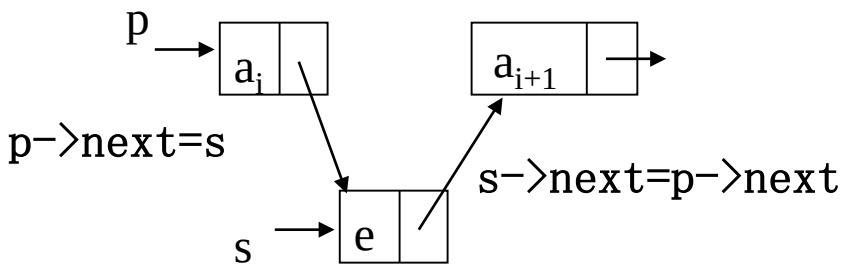
➢ 单链表的插入操作ListInsert_L

- 插入结点:指针p所指的结点后插入指针s所指的结点
- s->next = p->next; p->next = s;

插入前:



插入后:



➢ 单链表的插入操作ListInsert_L

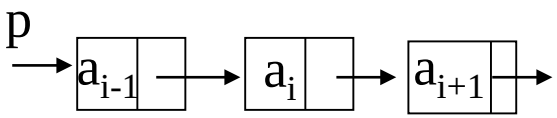
- **算法思想:**
  - **声明一个指针p指向链表头结点，初始化j从1开始；**
  - **当j<i时，就遍历链表，让p的指针向后移动，不断指向下一结点，j累加1；**
  - **若到链表末尾p为空，则说明第i个结点不存在；**
  - **否则查找成功，在系统中生成一个空结点s；**
  - **将数据元素e赋值给s->data；**
  - **单链表的插入标准语句s->next=p->next; p->next=s；**
  - **返回成功。**

```
/* 初始条件：顺序线性表L已存在, i≤i≤ListLength(L) */
/* 操作结果：在L中第i个结点位置之前插入新的数据元素e，L的长度加1*/
Status ListInsert_L(LinkList &L, int i, ElemType e)
{
  LinkList s,p;
  int j;
  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;                    /* 第i个结点不存在*/
  s = (Lnode *)malloc(sizeof(Lnode));     /* 生成新结点 */
  if(!s) return OVERFLOW;
  s->data = e;
  s->next = p->next; p->next = s;        /* 将p的后续结点赋值给s的后继,将s赋值
  给p的后续 */
  return OK;
}
```

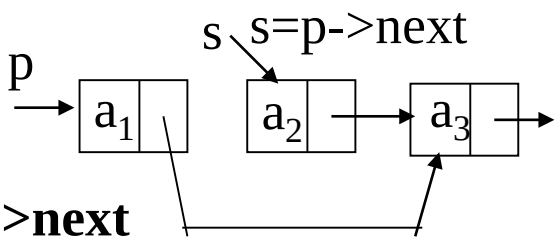
➢ 单链表的删除操作ListDelete_L

- 删除结点：删除指针p所指的结点后的结点
- s=p->next; p->next = p->next->next;

删除前：

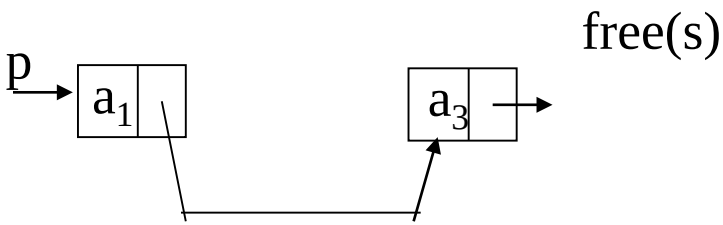


删除后：



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

释放结点后：



➢ 单链表的删除操作ListDelete_L

- 算法思想:
  - 声明一个指针p指向链表头结点，初始化j从1开始；
  - 当j<i时，就遍历链表，让p的指针向后移动，不断指向下一结点，j累加1；
  - 若到链表末尾p为空，则说明第i个结点不存在；
  - 否则查找成功，将欲删除结点p->next赋值给s：s=p->next；
  - 单链表的删除标准语句p->next=s->next；
  - 将q结点中的数据赋值给e，作为返回；
  - 释放q结点
  - 返回成功。

```
/* 初始条件：顺序线性表L已存在, i≤i≤ListLength(L) */
/* 操作结果：删除L的第i个结点，并用e返回其值，L的长度减1*/
Status ListDelete_L(LinkList L, int i, ElemType &e)
{
  LinkList s,p;
  int j;
  p = L; j = 0;
  while(p->next && j<i-1){p=p->next;++j} /*遍历寻找第i-1个结点*/
  if(!(p->next)||j>i-1) return ERROR;/*第i个结点不存在*/
  s = p->next;
  p->next = s->next;   /*将s的后续赋值给p的后继*/
  e = s->data;         /*将s结点中的数据赋给e*/
  free(s);                    /*让系统回收此结点，释放内存*/
  return OK;
}
```

单链表的插入和删除的算法时间复杂度：O(n)

1. 遍历查找第i个结点 O(n)
2. 插入或删除结点 O(1)

那批量插入或删除呢？

对于插入或删除数据越频繁的操作，单链表的效率优势就越是明显 。

- 单链表表的生成操作CreateList_L

如何生成单链表？

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。

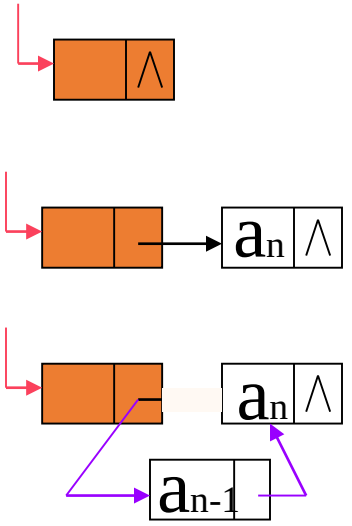
操作步骤：

1. 建立一个“空表”；

2. 输入数据元素a_n，
   建立结点并插入；

3. 输入数据元素a_{n-1}，
   建立结点并插入；

4. 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
    }
```

算法的时间复杂度为:O(n)

```
} // CreateList_L
```

➢ 单链表的清除操作 ClearList_L(&L)

```
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(n)

➢例：合并两个有序链表

```
void MergeList(LinkList&La,LinkList& Lb,LinkList& lc)
{
    ListNode* pa = La->next ;
    ListNode* pb = Lb->next ;
    Lc=pc=pa;
    while( pa && pb ) {
        if ( pa->data <= pb->data ) {
            pc->next = pa ; pc = pa ; pa = pa->next;
        }
         else { pc->next=pb; pc=pb; pb=pb->next; }
    }
    pc->next=pa ? pa : pb ; //链接剩余元素
    free(Lb); //释放头结点
}
```

2.3 线性表的链式表示与实现

优点

- 它是一种动态结构，整个存储空间为多个链表共用
- 不需预先分配空间
- 插入、删除操作

缺点

- 指针占用额外存储空间
- 不能随机存取，查找速度慢

2.3 线性表的链式表示与实现

存储分配方式

- 顺序存储结构用一段连续的存储单元依次存储线性表的数据元素
- 单链表采用链式存储结构，用一组做任意的存储单元存放线性表元素

时间性能

- 查找
  - 顺序存储结构O(1)
  - 单链表O(n)
- 插入和删除
  - 顺序存储结构需要平均移动表长一半的元素，时间为O(n)
  - 单链表在找出某位置的指针后，插入和删除时间为O(1)

空间性能

- 顺序存储结构需要预先分配存储空间，分大了，浪费，分小了易发生上溢，需要重新分配
- 单链表不需要预先分配存储空间，只要有就可以分配，元素个数不受限制

Part.4

2.4 一元多项式的表示及相加

2.4 一元多项式的表示及相加

$P_n(x)=p_0+p_1x+p_2x^2+\ldots+p_nx^n$

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

$Q_m(x)=q_0+q_1x+q_2x^2+\ldots+q_mx^m$

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

设m<n,则

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

$=(p_0+q_0)+(p_1+q_1)x+(p_2+q_2)x^2+\ldots+(p_m+q_m)x^m+p_{m+1}x^{m+1}+\ldots+p_nx^n$

可用线性表R表示

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

2.4 一元多项式的表示及相加

- 顺序存储结构线性表:很难处理多项式次数很高且变化很大的情况，存储结构的最大长度很难确定。

例:$S(x)=1+3x^{10000}+2x^{20000}$

2.4 一元多项式的表示及相加

$$P_n(x)=p_1x^{e_1}+p_2x^{e_2}+\ldots + p_mx^{e_m}$$

满足：$0 \leqslant e_1<e_2<\cdots\cdots<e_{m=n}$

■可用长度为m且每个元素有两个数据项(系数项和指数项)的线性表表示一元多项式$P_n(x)$

◆最坏情况下：需要多存储一倍的数据

◆普遍情况下，对于多数多项式，将大大节约存储空间。

2.4 一元多项式的表示及相加

ADT Polynomial {

数据对象：略

数据关系：略

基本操作：

CreatePolyn(&P, m)

操作结果:输入m项的系数和指数，建立一元多项式

DestroyPolyn(&P)

操作结果：销毁一元多项式P

AddPolyn(&Pa, &Pb)

操作结果：完成多项式的加法运算,即Pa=Pa+Pb,并销毁一元多项式Pb

SubtractPolyn(&Pa,&Pb)

操作结果：完成多项式的想减运算,即Pa=Pa-Pb,并销毁一元多项式Pb

MultiplyPolyn(&Pa,&Pb)

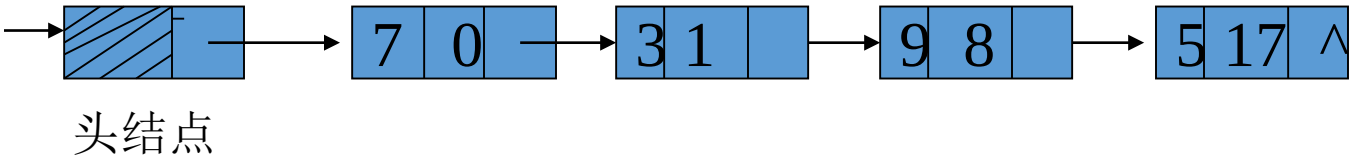
操作结果：完成多项式的相乘运算,即Pa=PaxPb,并销毁一元多项式Pb

} ADT Polynomial ;

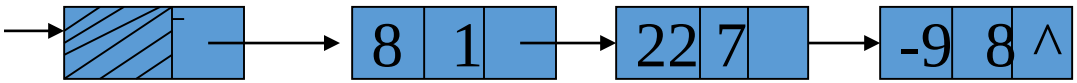
2.4 一元多项式的表示及相加

$A_{17}(x)=7+3x+9x^{8}+5x^{17}$

$B_{8}(x)=8x+22x^{7}-9x^{8}$



头结点



头结点

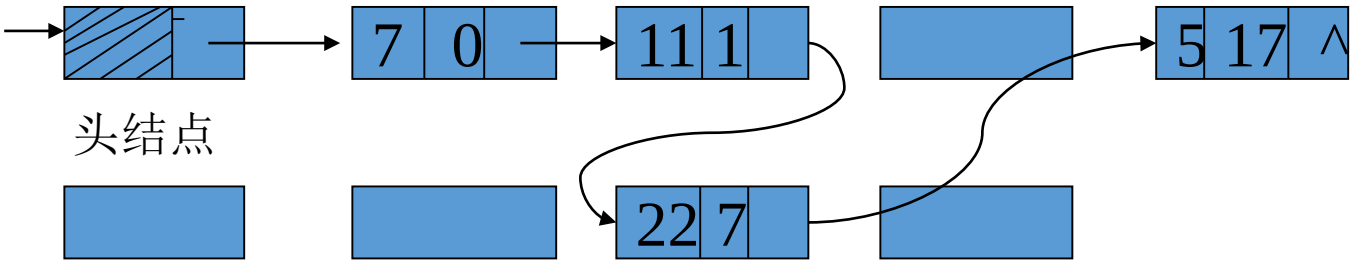
2.4 一元多项式的表示及相加

假设指针qa和qb分别指向多项式A和多项式B中当前进行比较的某个结点运算规则为：

①指针qa所指结点的指数值<指针qb所指结点的指数值，则将qa指针所指结点插入到“和多项式”链表

②指针qa所提结点的指数值>指针qb所指结点的指数值，则将qb指针所指指点插入到“和多项式”链表

③指针qa所指结点的指数值=指针qb所指结点的指数值，则将两个结点中的系数相加，若和不为零，则修改qa所指结点的系数值，同时释放qb所指结点； 反之，释放qa和qb所指结点。



算法描述

链式存储表示的类型

```
typedef struct ploy
   {   float coef ;     /*系数部分*/
       int    expn ;    /*指数部分*/
       struct ploy  *next ;
   } Ploy ;
```

算法描述

```
Ploy  *add_ploy(ploy  *La，  ploy  *Lb)

  /*  将以La ，Lb为头指针表示的一元多项式相加，生成一个新的结果多项式  */
   {  ploy  *Lc , *pc , *pa , *pb , *p ;  float  x ;
      Lc=pc=(ploy *)malloc(sizeof(ploy)) ;
      pa=La->next ; pb=Lb->next ;
      while (pa!=NULL&&pb!=NULL)
         {  if  (pa->expn<pb->expn)
               {  p=(ploy *)malloc(sizeof(ploy)) ;
                  p->coef=pa->coef ; p->expn=pa->expn ;
                  p->next=NULL ;
```

```
        /* 生成一个新的结果结点并赋值 */
      pc->next=p ; pc=p ; pa=pa->next ;
    }
     /* 生成的结点插入到结果链表的最后，pa指向下一个结点 */
if  (pa->expn>pb->expn)
  {  p=(ploy *)malloc(sizeof(ploy)) ;
      p->coef=pb->coef ; p->expn=pb->expn ;
      p->next=NULL ;
        /* 生成一个新的结果结点并赋值 */
       pc->next=p ; pc=p ; pb=pb->next ;
    }   /* 生成的结点插入到结果链表的最后，pb指向下一个结点 */
```

```
if  (pa->expn==pb->expn)
   {  x=pa->coef+pb->coef ;
      if  (abs(x)<=1.0e-6)
          /*  系数和为0，pa, pb分别直接后继结点  */
         {  pa=pa->next ; pb=pb->next ;  }
      else   /*  若系数和不为0，生成的结点插入到结果链表的最后，  pa, pb分别直接后继结点  */
          {  p=(ploy *)malloc(sizeof(ploy)) ;
             p->coef=x ; p->expn=pb->expn ;
             p->next=NULL ;
              /*  生成一个新的结果结点并赋值  */
             pc->next=p ; pc=p ;
             pa=pa->next ; pb=pb->next ;
```

```
            }
        }
    }    /* end of while */
if  (pb!=NULL)
    while(pb!=NULL)
      {   p=(ploy *)malloc(sizeof(ploy)) ;
          p->coef=pb->coef ; p->expn=pb->expn ;
          p->next=NULL ;
              /*  生成一个新的结果结点并赋值  */
            pc->next=p ; pc=p ; pb=pb->next ;
      }
```

```
    if  (pa!=NULL)
        while(pa!=NULL)
            {   p=(ploy *)malloc(sizeof(ploy)) ;
                p->coef=pb->coef ; p->expn=pa->expn ;
                p->next=NULL ;
                    /*  生成一个新的结果结点并赋值  */
                  pc->next=p ; pc=p ; pa=pa->next ;
            }
    return (Lc) ;
}
```

Part.5

总结

总 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（顺序表）和链式存储结构（链表）。
2. 熟练掌握这两类存储结构的描述方法。
3. 熟练掌握线性表在顺序存储结构上实现基本操作：查找、插入和删除的算法。
4. 熟练掌握在单链表上实现基本操作：创建、查找、插入和删除。
5. 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**。