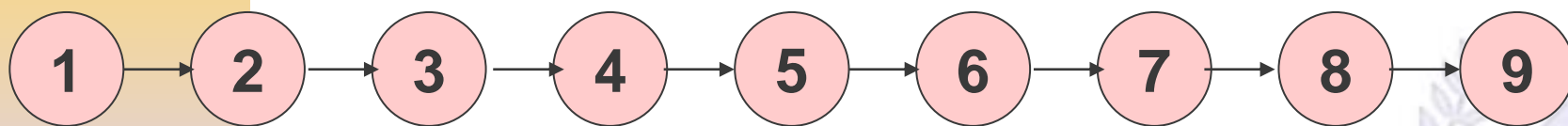




## 第二章 线性表



# 温馨提示: before September. 19

## 数据结构与算法设计 (2021级中文班)

网络教室 / 我的课程 / 2022-2023第一学期本科生 / 计算机学院 / 数据结构与算法设计 (2021级中文班) / 第1章 绪论 (2学时) /

查看

提交

结果

提交历史

测试环境

报表

相似度

### 1. 约瑟夫问题

成绩	10	开启时间	2022年09月5日 星期一 00:00
折扣	0.8	折扣时间	2022年09月18日 星期日 23:55
允许迟交	否	关闭时间	2022年09月19日 星期一 23:55

# 温馨提示: before Sep.25/Oct.11

## 第2章 线性表 (4学时)

本章为线性表, 包括顺序表、单链表和双链表等内容。

### 第2章-线性表-练习

**完成条件:** 达到要求

### 2. 基本操作 (集合合并)

**完成条件:** 完成

### 3. 链表原地逆序

**完成条件:** 完成

### 5. 求循环小数

**完成条件:** 完成

# 线性表及其特征

◆ **线性表**是一种最简单的**线性结构**

◆ **线性结构**一个数据元素的**有序集**

◆ **线性表的特征为：**

‖ 1) 存在唯一的 **第一元素**；

‖ 2) 存在唯一的**最后元素**；

‖ 3) 除第一个元素外，每个元素均有**唯一的前驱**；

‖ 4) 除最后一个元素外，每个元素均有**唯一的后继**；

‖ 5) 同一线性表的元素具有**相同的特性**





# 线性表的实例

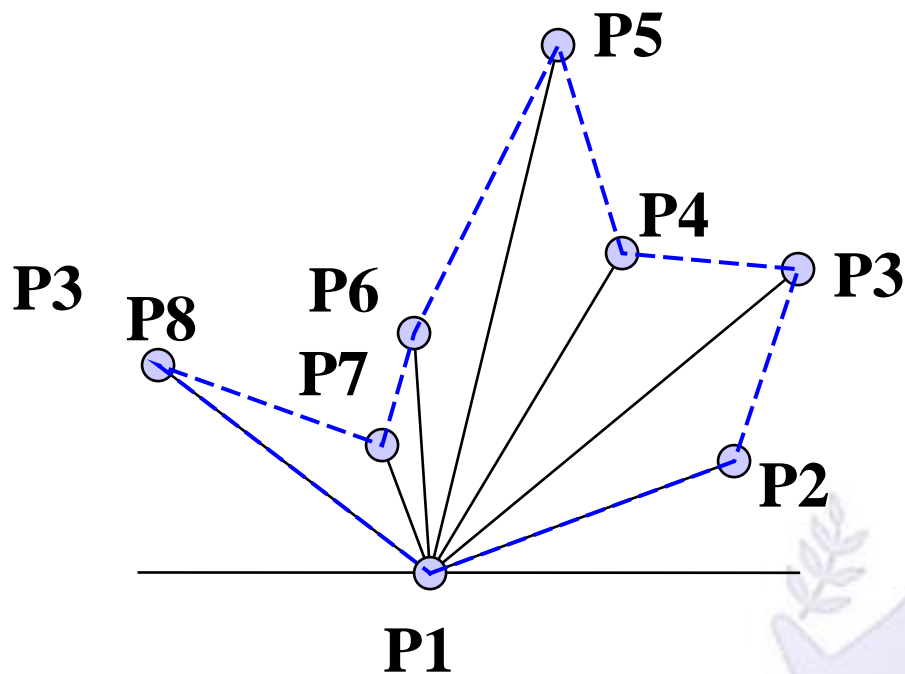
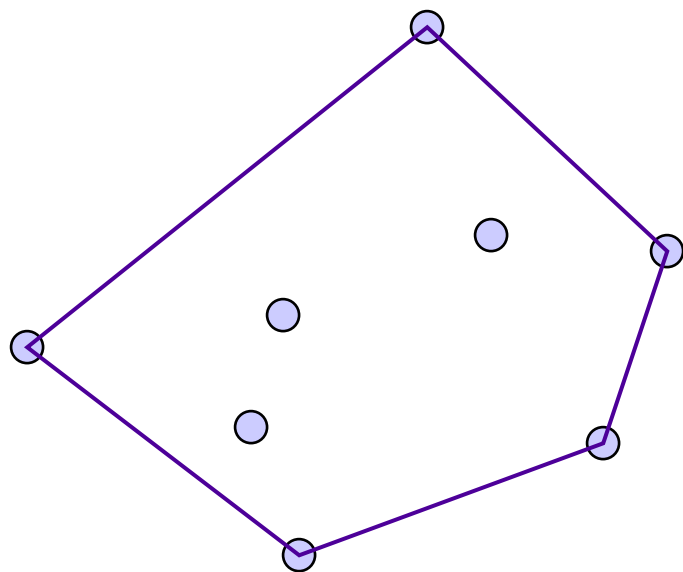
- ◆ 实例1：通讯录、电话簿
- ◆ 实例2：图书管理
- ◆ 实例3：学生成绩管理
- ◆ 实例4：一元多项式的表示及运算

$$p_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$$



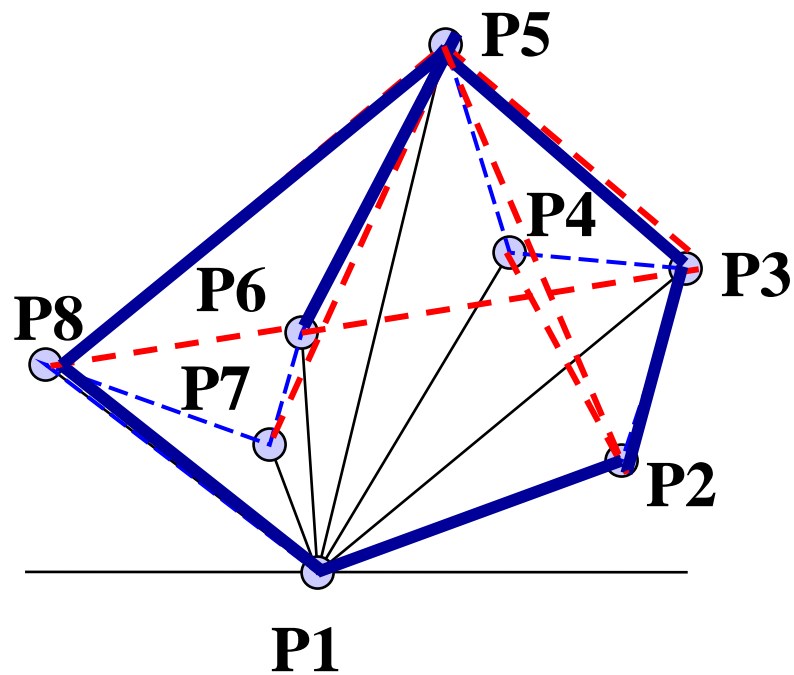
# 线性表的实例

## ◆ 实例5：点的凸包问题





## ◆ 实例5：点的凸包问题



# 线性表的实例

- ◆ 实例6：括号匹配：  $(a+b(u-c*t))$
- ◆ 实例7：停车场安排







# 线性表的实例

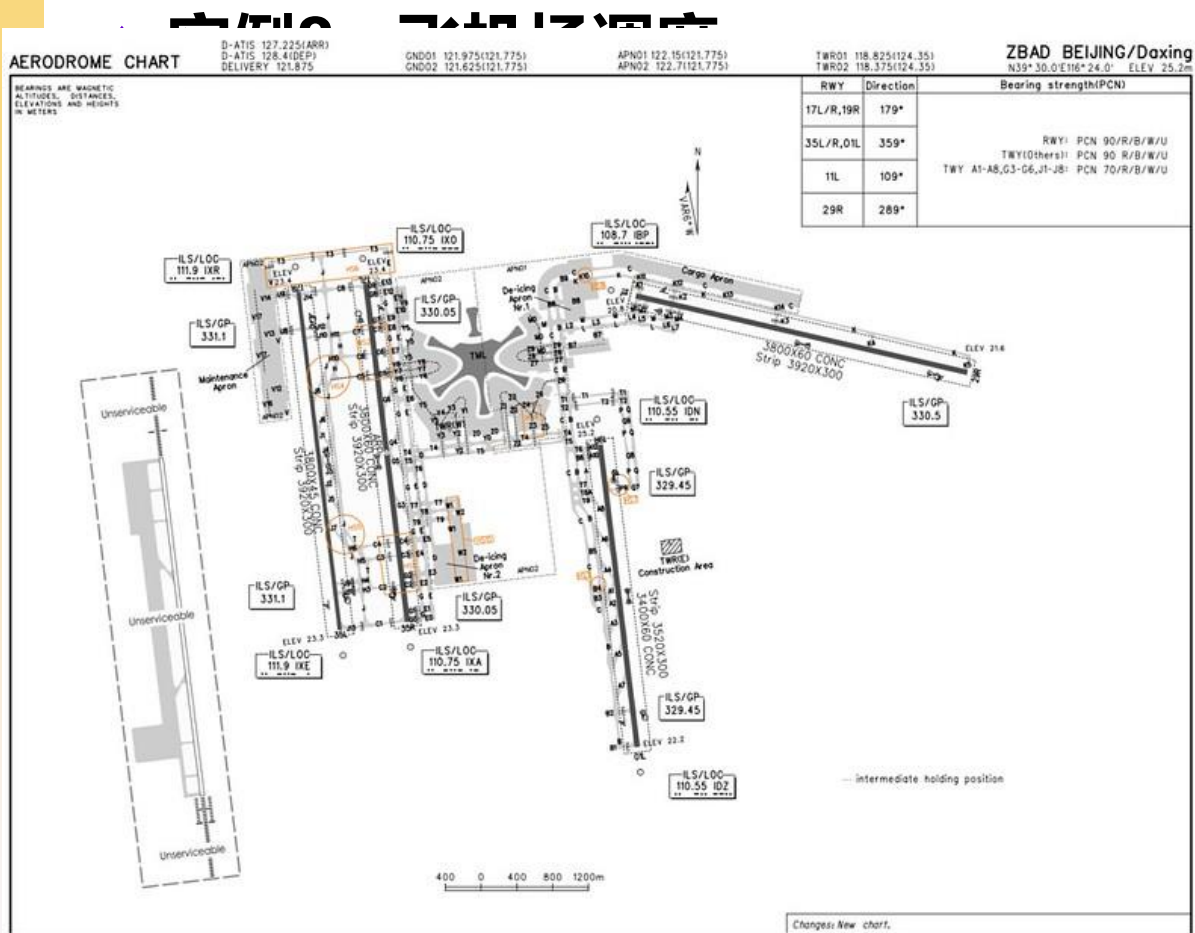
## ◆ 实例8：飞机场调度

### ◆ 原则：

- ✎ **降落优先起飞**，在此原则下按来的顺序排队；
- ✎ 每驾飞机都有一个编号；
- ✎ 每个跑道都有一个编号，都可以用来降落和起飞
- ✎ 跑道同一时间只能被一架飞机占用，占用时间为该飞机降落（起飞）占用跑道时间。



# 线性表的实例

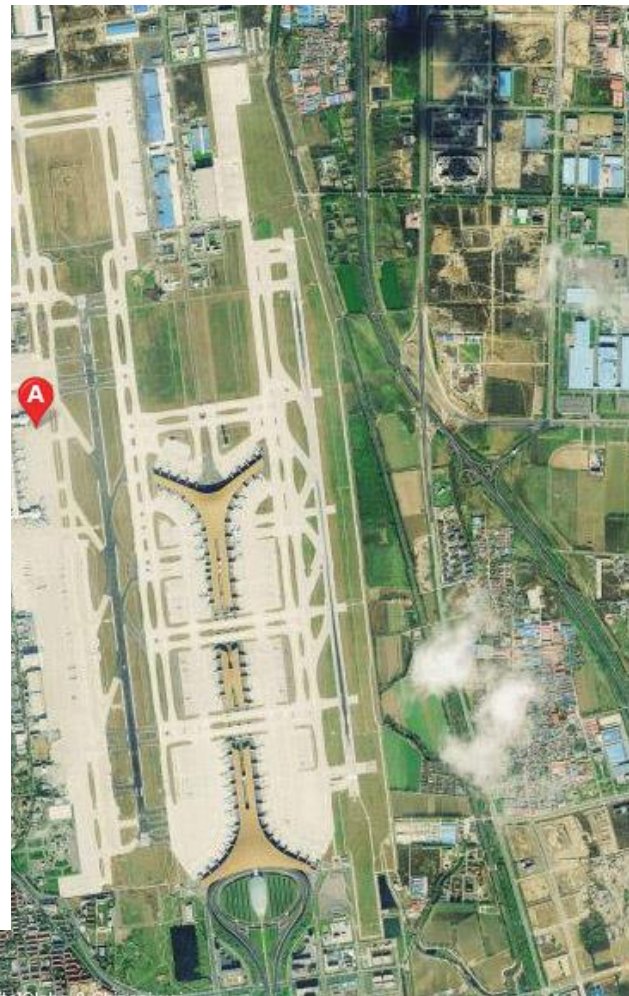


2019-8-20 EFF1910091600

中国民用航空局CAAC

ZBAD AD2.24-1A

yinlei.org 为航空和飞行模拟爱好者整理，本资料请不要用于实际飞行。





# 本章内容

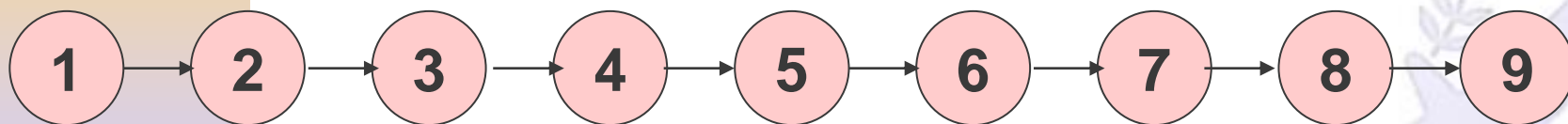
- ◆ 2.1 线性表的类型定义
- ◆ 2.2 线性表类型的实现 — 顺序映象
- ◆ 2.3 线性表类型的实现 — 链式映象
- ◆ 2.4 一元多项式的表示和相加





## 2.1 线性表的类型定义

一个线性表是 $n$ 个数据元素的有限序列



## 2.1.1 抽象数据类型线性表的定义

ADT list {

数据对象:  $D=\{a_i \mid a_i \in \text{Elemset}, i=1, 2, \dots, n, n \geq 0\}$

//n 为线性表的表长;  $n=0$  时的线性表为空表

数据关系:  $R1=\{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2, \dots, n\}$

// i 为  $a_i$  在线性表中的位序

基本操作: .....

}//ADT list



# 第一类操作：结构性操作

## ◆ **InitList( &L )**//初始化一个线性表

‖ 操作结果：构造一个空的线性表L

## ◆ **DestroyList( &L )**//删除线性表

‖ 初始条件：线性表 L 已存在

‖ 操作结果：销毁线性表 L





## 第二类操作：引用型操作

### ◆ ListEmpty( L )//判断线性表是否为空

‖ 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



## 第二类操作：引用型操作（续）

◆ **PriorElem( L, cur\_e, &pre\_e )** //获取当前元素的前驱元素

‣ 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 )** //获取当前元素的后继元素

‣ 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



## 第二类操作：引用型操作（续）

### ◆ **GetElem( L, i, &e )** //访问第i个元素

‣ 初始条件：线性表L已存在；

$1 \leq i \leq \text{LengthList}(L)$ ;

‣ 操作结果：用e 返回L中第i 个元素的值

### ◆ **LocateElem( L, e, compare() )** //寻找是否有值与e相同的元素

‣ 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



## 第二类操作：引用型操作（续）

### ◆ 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



## 第三类操作：加工型操作

### ◆ **ClearList( &L )** //将线性表重置为空表

‖ 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;

$1 \leq i \leq \text{LengthList}(L);$

‖ 操作结果：L中第i个元素赋值同e的值



## 第三类操作：加工型操作

### ◆ ListInsert( &L, i, e ) //插入数据元素

‖ 初始条件：线性表L已存在；

$1 \leq i \leq \text{LengthList}(L) + 1$ ;

‖ 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；

$1 \leq i \leq \text{LengthList}(L)$ ;

‖ 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



# 线性表的ADT-操作

- ◆ **InitList( &L )**           //初始化一个线性表
  - ◆ **DestroyList( &L )**   //删除线性表
  - ◆ **ListEmpty( L )**       //判断线性表是否为空
  - ◆ **ListLength( L )**       //求线性表的长度
  - ◆ **PriorElem( L, cur\_e, &pre\_e )**  
                              //获取当前元素的前驱元素
  - ◆ **NextElem( L, cur\_e, &link\_e )**  
                              //获取当前元素的后继元素
- 

# 线性表的ADT-操作

- ◆ **GetElem( L, i, &e )**                      **//访问第i个元素**
- ◆ **LocateElem( L, e, compare( ) )**  
   **//寻找是否有值与e相同的元素**
- ◆ **ListTraverse (L, visit( ))** **//遍历线性表**
- ◆ **ClearList( &L )**                              **//将线性表重置为空表**
- ◆ **PutElem( &L, i, e )**                        **//改变数据元素的值**
- ◆ **ListInsert( &L, i, e ) )**                   **//插入数据元素**
- ◆ **ListDelete(&L, i, &e) )**                   **//删除数据元素**

## 2.1.2 用线性表实现其它复杂操作

- ◆ 例1、线性表的合并  $A = A \cup B$
- ◆ 设:有两个集合 A 和 B 分别用两个线性表 LA 和 LB 表示。
- ◆ 求一个新的集合  $A = A \cup B$ 。

LA	3	5	1	2	11	9	4
----	---	---	---	---	----	---	---

LB	2	10	6	1	7	4
----	---	----	---	---	---	---

LA	3	5	1	2	11	9	4	10	6	7
----	---	---	---	---	----	---	---	----	---	---

# 例1、线性表的合并 $A = A \cup B$

- ◆ **基本操作：**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
- ◆ **控制程序：**for (i = 1; i <= Lb\_len; i++)

LA	3	5	1	2	11	9	4
----	---	---	---	---	----	---	---

LB	2	10	6	1	7	4
----	---	----	---	---	---	---

LA	3	5	1	2	11	9	4	10	6	7
----	---	---	---	---	----	---	---	----	---	---





## ◆ 1. 初始化参数

**Lb\_len = ListLength(LB)**

◆ 2. 从线性表LB中依次察看每个数据元素;

► **GetElem(LB , i , e)**

- **LocateElem(LA, e, equal( ))**

► **ListInsert(LA, n+1, e)**

# 例1、线性表的合并 $A = A \cup B$

```
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、有序线性表的合并

- ◆ 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
- ◆ 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， $LC = LA + LB$

LA 

3	5	8	11
---	---	---	----

LB 

2	6	8	9	11	15	20
---	---	---	---	----	----	----

LC 

2	3	5	6	8	8	9	11	11	15	20
---	---	---	---	---	---	---	----	----	----	----

1 2 3 4 5 6 7 8 9 10 11

## 例2、有序线性表的合并

- ◆ **基本操作**：构造线性表LC，从LA和LB中依次取出元素，按照非递减的顺序依次插入到表尾
- ◆ **如何确定插入顺序？**
  - ‖ 取出LA中当前的元素a和LB中当前的元素b
  - ‖ 若 $a \leq b$ 则先插入a，并将a的指针后移一位
  - ‖ 否则先插入b，并将b的指针后移一位
- ◆ **控制程序**： `while((i<=/a-len)&&(j<=/b-len))`

LA

3

5

8

11

LB

2

6

8

9

11

15

20

LC

2

3

5

6

8

8

9

11

11

15

20

## 例2、有序线性表的合并

- ◆ 操作步骤:
- ◆ 1. 初始化LC: `InitList(LC);`
- ◆ 2. 设置LA和LB的当前指示:  $i=j=1$ ;
- ◆ 3. 计算LA和LB的长度
- ◆ 4. 依次从线性表LA和LB中取出当前元素, 根据上述标准判断需要插入a或b, 插入后将相应线性表的指示加1;
- ◆ 5. 如果某一个表中的元素取完了, 则将另外一个表的剩余元素之间插入到LC中



```
void mergelist (list la, list lb, list &lc){
```

```
    initlist(lc);           //步骤1
```

```
    i=j=1;k=0;              //步骤2
```

```
    la-len=listlength(la);  lb-len=listlength(lb); //步骤3
```

```
    while((i<=la-len)&&(j<=lb-len)){           //步骤4
```

```
        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){                                           //步骤5
```

```
        getelem((la, i++, ai); listinsert(lc, ++k, ai); }
```

```
    while(j<=lb-len){
```

```
        getelem((lb, j++, bj); listinsert(lc, ++k, bi);}
```

```
}//mergelist
```

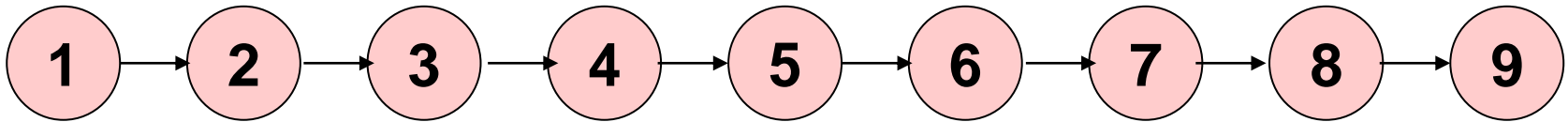


- ◆ **思考：例2中，如果LC中不允许有重复的元素，如何修改上述程序？**



## 2.2 顺序表(Sequential List)

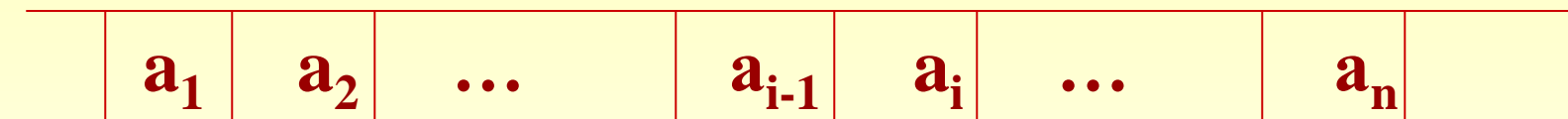
- 2.2.1 顺序表示及其特点
- 2.2.2 数据结构定义
- 2.2.3 顺序表的初始化操作
- 2.2.4 顺序表的插入操作
- 2.2.5 顺序表的删除操作
- 2.2.6 用顺序表实现合并操作  $LC = LA + LB$





## 2.2.1 顺序表示及其特点

- 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  $\langle x, y \rangle$ 。
- 最简单的一种顺序映象方法是：
  - 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。



线性表的起始地址  
称作线性表的基地址

## 2.2.1 顺序表示及其特点

- 以“**存储位置相邻**”表示有序对  $\langle a_{i-1}, a_i \rangle$ 
  - 即:  $LOC(a_i) = LOC(a_{i-1}) + C$
  - $C$  是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
  - $LOC(a_i) = \underline{LOC(a_1)} + (i-1) \times C$

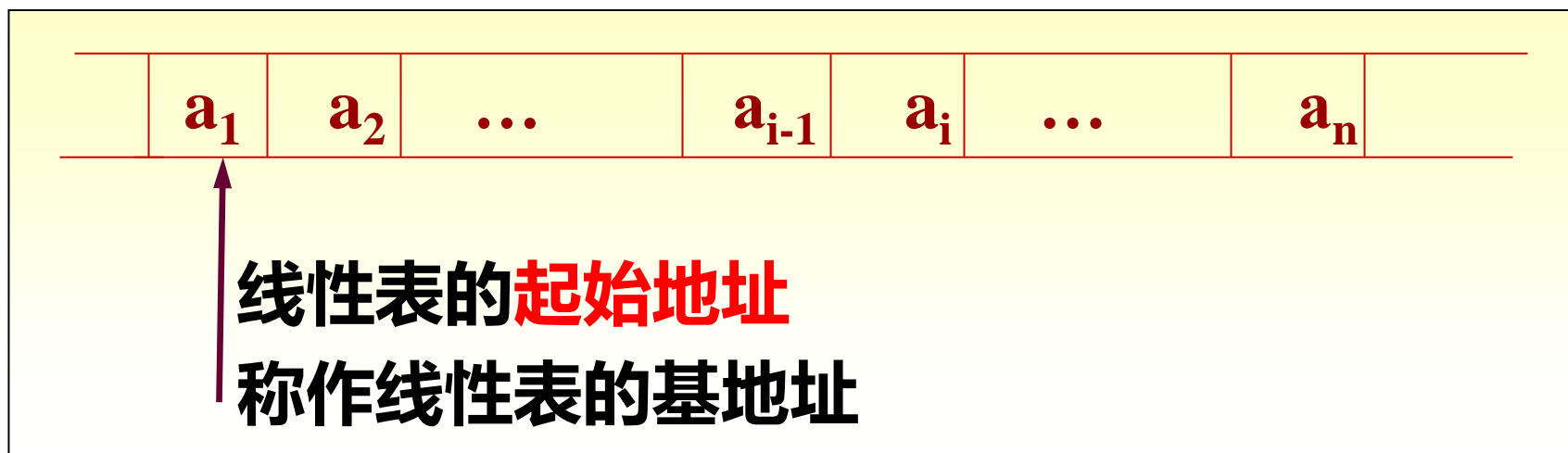


↑  
线性表的**起始地址**  
称作线性表的**基地址**

# 小结：顺序表的特点

- 用连续的存储单元存放线性表的元素。
- 元素存储顺序与元素的逻辑顺序一致。
- 读写元素方便，通过下标即可指定位置。

C语言中采用一维数组存放顺序表



## 2.2.2 顺序表数据结构定义

### ◆ 静态定义

data	n
------	---

**#define maxSize 100** // 线性表最大存储空间

```
typedef struct {  
    DataType data[maxSize]; // 存储空间基址  
    int      n; // 当前长度（元素个数）  
} SqList; // 俗称顺序表
```

## 2.2.2 顺序表数据结构定义

### ◆ 动态定义

elem	length	listsize
------	--------	----------

```
#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
```

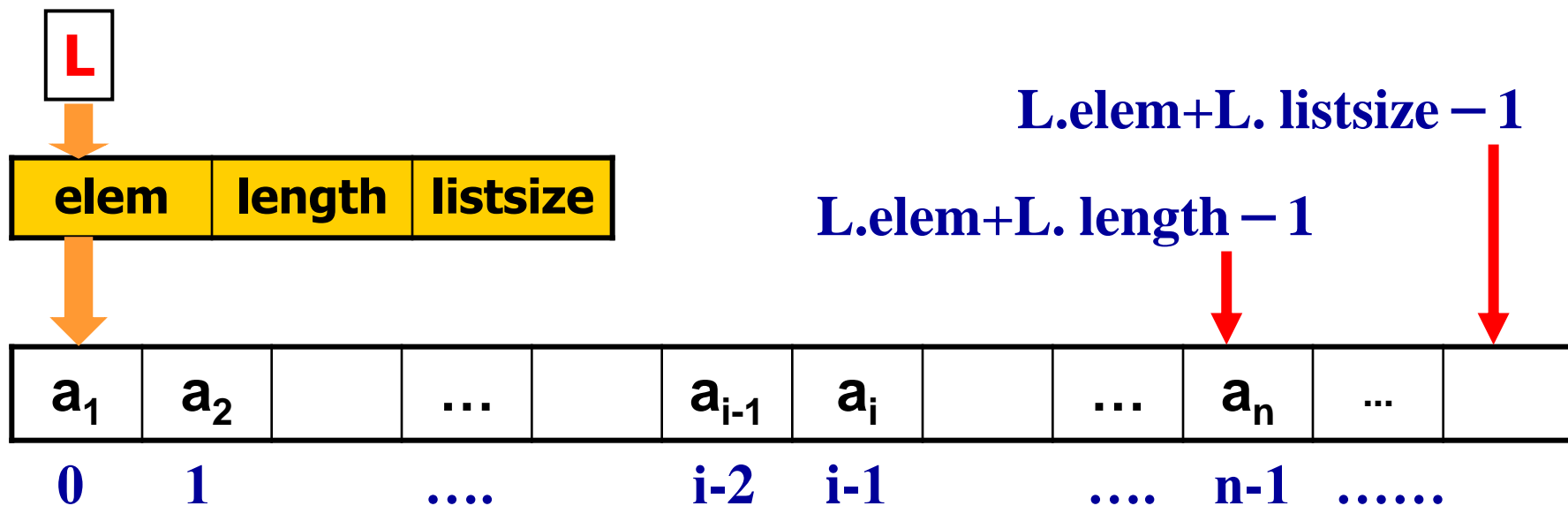
```
typedef struct {
    ElemType *elem;    // 存储空间基址（向量指针）
    int    length;      // 当前长度（元素个数）
    int    listsize;    // 当前表最大存储容量
                    // (以sizeof(ElemType)为单位)
} SqList; // 俗称顺序表
```

# 顺序表

SqList L;

```
typedef struct {  
    ElemType *elem;  
    int      length;  
    int      listsize;  
} SqList; // 顺序表
```

注意：由于C语言中数组的下标从“0”开始，表中第  $i$  个元素是  $L.elem[i - 1]$ .



# 顺序表的初始化操作

```
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
```

```
typedef struct {  
    ElemType *elem;  
    int      length;  
    int      listsize;  
} SqList; // 顺序表
```

# 顺序表按值查找操作

- 按在顺序表中从头查找结点值等于给定值  $x$  的结点，依次进行元素比较。

```
int Find( SeqList& L, DataType x )
```

```
{ int i = 1; DataType * p;  
  for ( p=L.data; p; p++ ) //p++将指针 p 进  
    if ( *p != x ) i++;    //到L.data[] 的下  
    else break;           //一个元素位置  
  if ( i <= L.length ) return i; // *p取 p 所指  
  else return 0;           // 元素的值  
}
```



# 顺序表按值查找操作

- 算法性能分析: 查找成功的**平均比较次数**

$$ACN = \sum_{i=0}^{n-1} p_i \times c_i$$

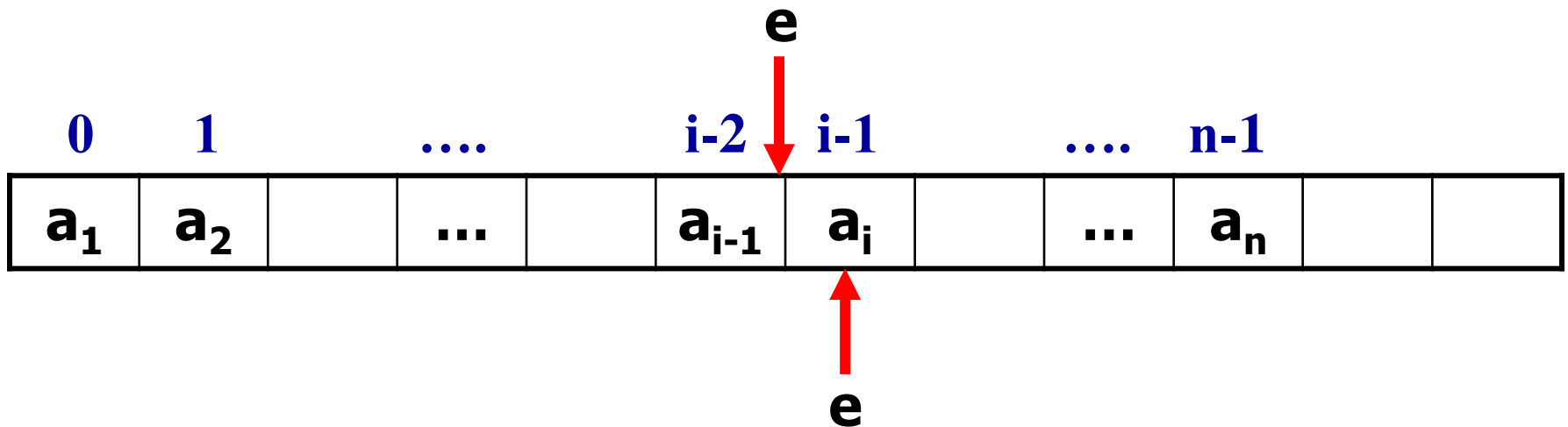
- 若查找概率相等, 则

$$\begin{aligned} ACN &= \frac{1}{n} \sum_{i=0}^{n-1} (i+1) = \frac{1}{n} (1+2+\cdots+n) = \\ &= \frac{1}{n} * \frac{(1+n) * n}{2} = \frac{1+n}{2} \end{aligned}$$

- 查找不成功 数据比较  $n$  次。

# 顺序表的插入操作

- **ListInsert(&L, i, e)** // 插入元素
- 在顺序表L的第  $i$  个元素之前插入新的元素  $e$ ,
- 把  $e$  插入到第  $i$  个元素的位置
- $i$  的合法范围为  $1 \leq i \leq \underline{L.length+1}$

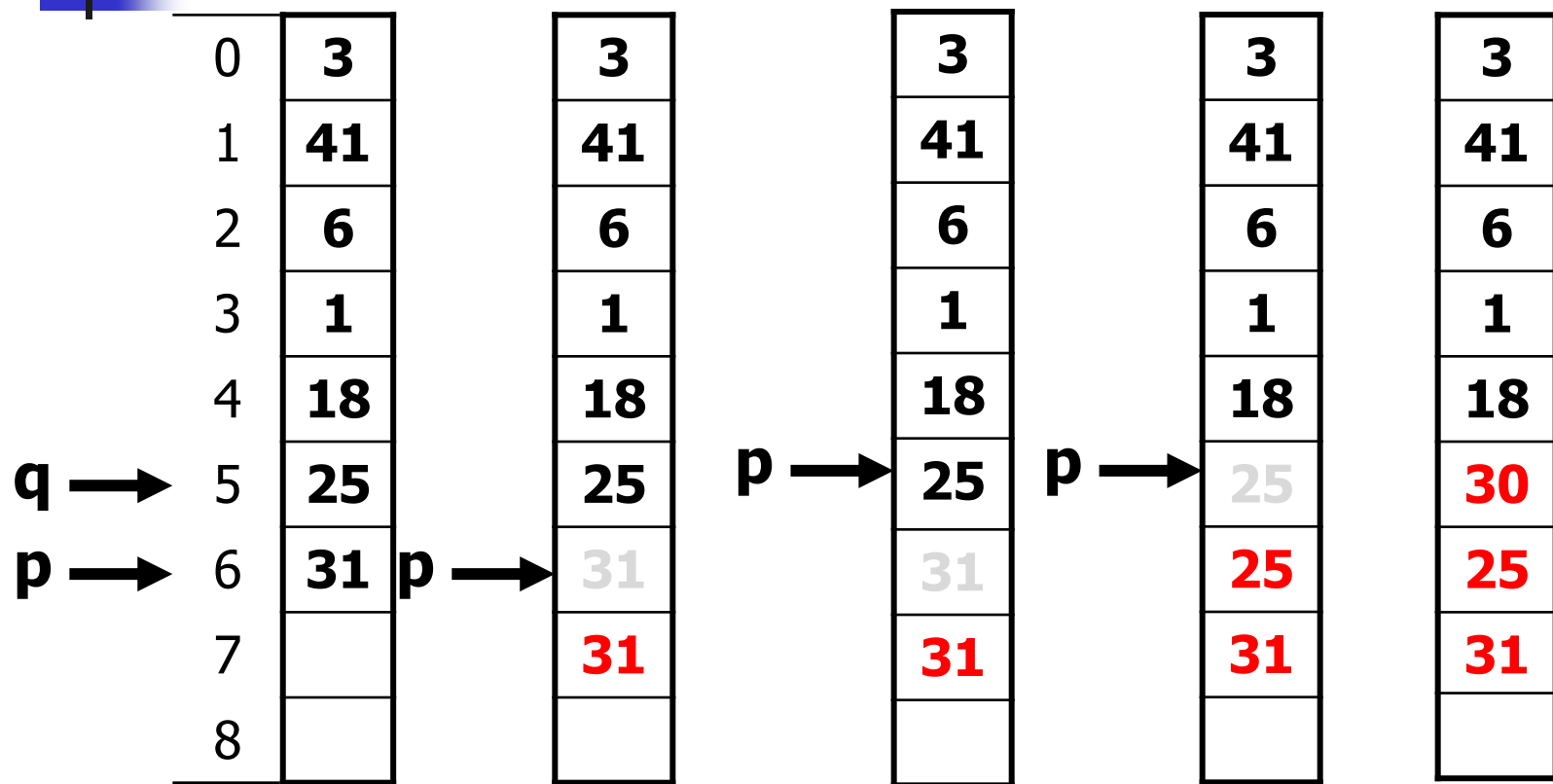


# 线性表插入操作

- 操作的过程: `ListInsert(&L, 6, 30)`

30 →	0	3	3	3	3	3
	1	41	41	41	41	41
	2	6	6	6	6	6
	3	1	1	1	1	1
	4	18	18	18	18	18
	5	25	30	25		30
	6	31	25		25	25
	7		31	31	31	31
	8					

# 操作的过程: ListInsert(&L, 6, 30)



q = &(L.elem[i-1]);    \*(p+1) = \*p;    p--;

p = &(L.elem[L.length-1]);

\*(p+1) = \*p;

\*q=e;

p>=q;

插入操作



# 插入操作步骤： ListInsert

## ■ 操作步骤

1. 判断插入位置是否合法：  $1 \leq i \leq L.length + 1$
2. 初始化指针p和q;
3. 循环：从表尾开始，依次将第i-1个元素之后的元素顺序后移一位
4. 将新元素e写入到第i个位置
5. 将表的长度加1

<b>Status ListInsert_Sq(SqList &amp;L, int i, ElemType e) {</b>	
<b>if (i &lt; 1    i &gt; L.length+1)</b>	<b>// 步骤1: 位置不合法</b>
<b>return ERROR;</b>	
<b>q = &amp;(L.elem[i-1]);</b>	<b>//步骤2: q 指示插入位置</b>
<b>for (p = &amp;(L.elem[L.length-1]); p &gt;= q; p--)</b>	
<b>*(p+1) = *p;</b>	<b>//步骤3: 元素依次后移</b>
<b>*q = e;</b>	<b>// 步骤4: 插入e</b>
<b>++L.length;</b>	<b>// 步骤5: 表长加1</b>
<b>return OK;</b>	
<b>} // ListInsert_Sq</b>	

**算法时间复杂度取决于: 移动元素的次数**

# 插入操作的算法复杂度

- 考虑移动元素的平均情况(Average Moving Number):
  - 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:

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

算法时间复杂度为:  $O(n)$



# 如果存储空间已满怎么办？

```
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;           // 新基址重新赋给线性表  
    L.listsize += LISTINCREMENT;  
    // 增加存储容量，更新表长  
}
```





# 程序设计方法的两点说明

---

- 先考虑一般情况，后考虑特殊情况
- 一般不用基本操作实现其他基本操作

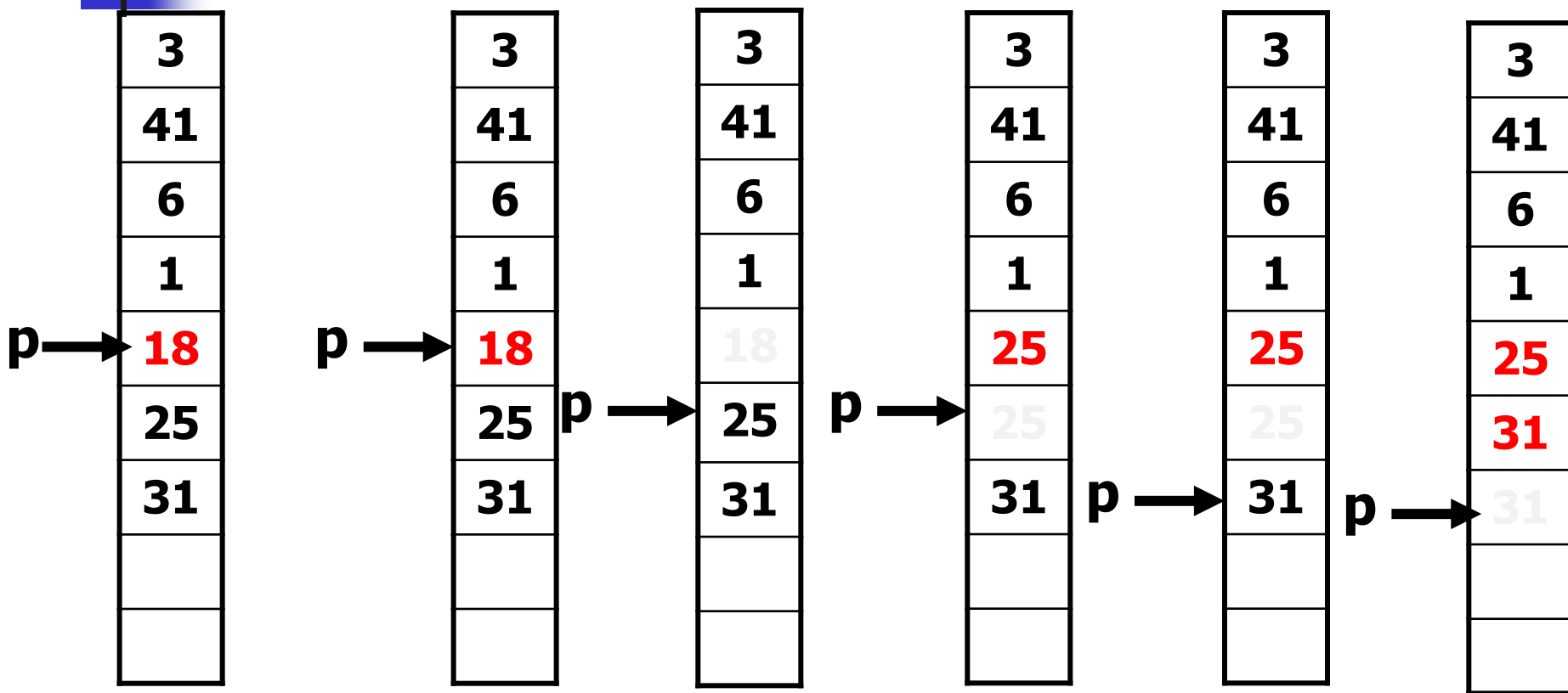


# 顺序表的删除操作

- `ListDelete(&L, i, &e)` // 删除元素
- 删除线性表中第*i*个元素，并将删除的元素方在*e*中
- *i* 的合法范围为  $1 \leq i \leq L.length$

删除操作

## 删除操作的过程: ListDelete (&L, 5, &e)



`p = &(L.elem[i-1]); *e = *p; p++; *(p-1) = *p; p++; *(p-1) = *p;`

`p <= L.elem + L.length - 1;`



# 删除操作步骤：ListDelete

## ■ 操作步骤

1. 判断删除位置是否合法：  $1 \leq i \leq L.length$
2. 初始化指针p;
3. 将第i个元素的值赋给变量e;
4. 循环：从第i+1个元素开始，依次将后面的元素顺序前移一位;
5. 将表的长度减1;

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e)
{
```

```
    if ((i < 1) || (i > L.length)) //步骤1: 位置是否合法
        return ERROR;
```

```
    p = &(L.elem[i-1]); //步骤2: 初始化指针
```

```
    e = *p; //步骤3: 赋给 e
```

```
    //步骤4: 被删除元素之后的元素前移
    tail = L.elem+L.length-1; // 表尾的位置
    for (++p; p <= tail; ++p) *(p-1) = *p;
```

```
    --L.length; // 步骤5: 表长减1
```

```
    return OK;
```

```
} // ListDelete_Sq
```

**算法时间复杂度取决于: 移动元素的次数**

# 删除操作的算法复杂度

- 考虑移动元素的平均情况(Average Moving Number):

- 假设在删除第  $i$  个元素的概率为  $p_i$ , 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:

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

- 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:

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

算法时间复杂度为:  $O(n)$



# 删除操作的算法复杂度

- 思考：删除算法最好情况下的时间复杂度是什么？

算法时间复杂度为： $O(1)$

# 定位操作: LocateElem\_Sq

**LocateElem\_Sq(SqList L, ElemType e,  
Status (\*compare)(ElemType, ElemType))**

L.elem

23	75	41	38	54	62	17		
----	----	----	----	----	----	----	--	--



L.length-1



L.listsize-1

i

8

e = 38 50

**基本操作：**将顺序表中的元素逐个和给定值 e 相比较。

**循环结束标志：**找到e或者p超过表尾的地址





# 定位操作

- 请大家课下自行练习顺序表的定位操作
  - 写出操作步骤和程序
  - 要求：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

算法时间复杂度为： $O(n)$



# 小结：顺序表的优缺点

---

## ■ 优点

- 不需要附加空间
- 随机存取任何一个元素（根据下标）

## ■ 缺点

- 很难估计所需空间的大小
- 开始就要分配足够大的一片连续内存空间
- 更新操作代价大



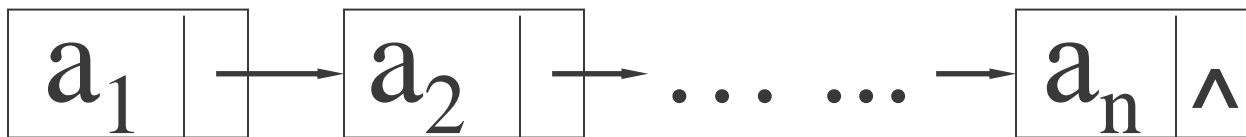
## 2.3 线性表的链式表示和实现

- ◆ 2.3.1 什么是线性链表
- ◆ 2.3.2 线性链表的定义
- ◆ 2.3.3 线性表的操作在链表中的实现
- ◆ 2.3.4 线性链表数据结构
- ◆ 2.3.5 静态链表
- ◆ 2.3.6 其它链表

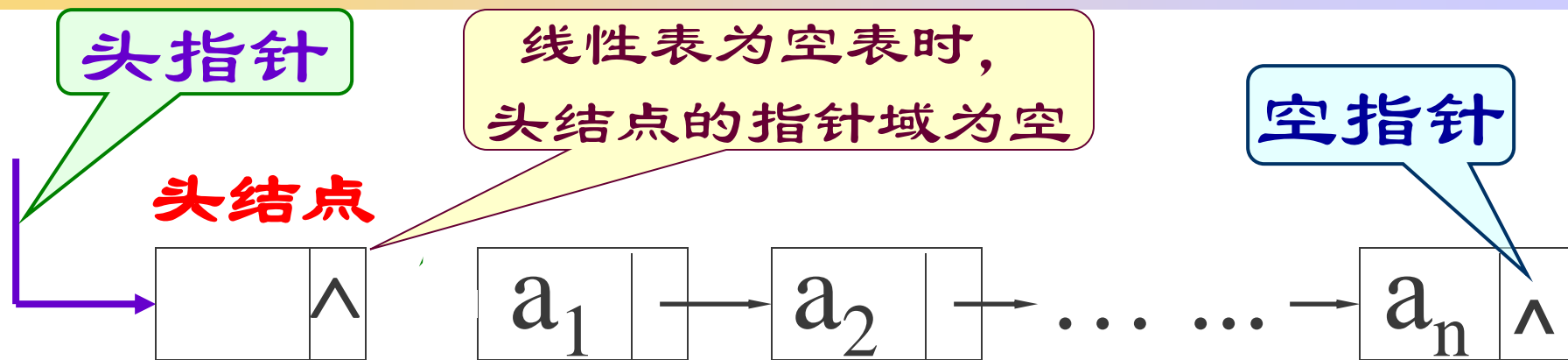


## 2.3.1 什么是线性链表

- ◆ 用一组**地址任意**的存储单元**存放**线性表中的**数据元素**。
- ◆ **元素** + **指针** (指示后继元素存储位置) = **结点** (表示数据元素的映象)
- ◆ 存储结构是非顺序映像, 即**链式映像**
- ◆ 以 “**结点的序列**” 表示线性表——称作**线性链表**



## 2.3.1 什么是线性链表

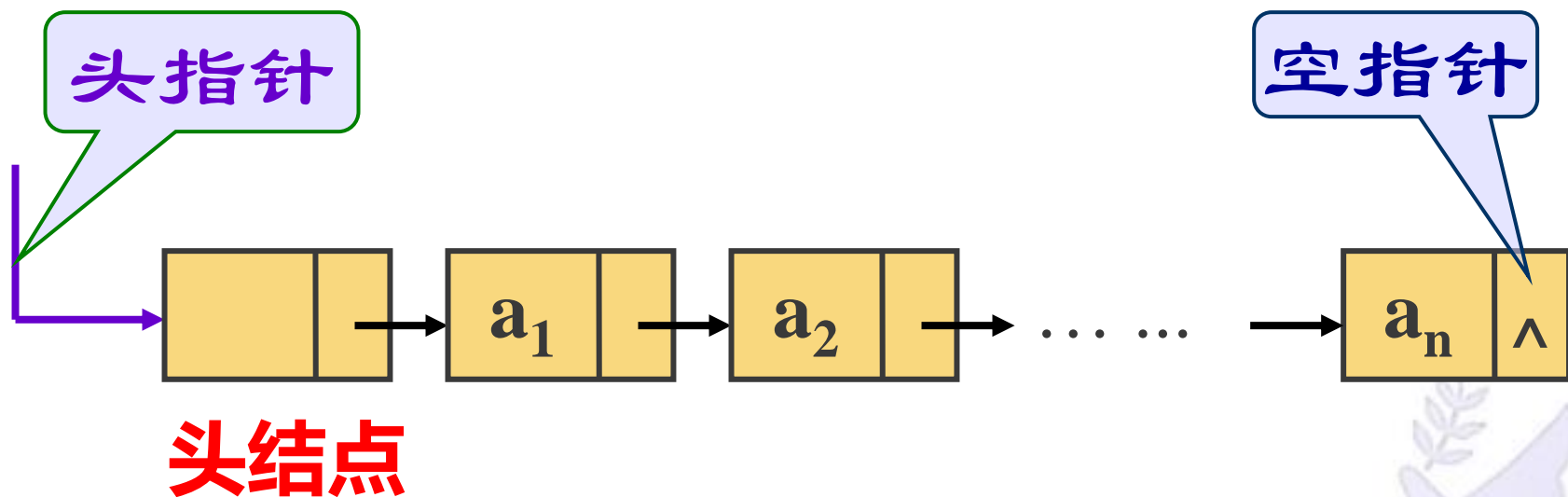


以线性表中第一个数据元素 $a_1$ 存储地址作为线性表的地址，称作线性表的**头指针**。

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。

## 2.3.1 什么是线性链表

- ◆ 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。



## 2.3.2 线性链表的定义

```
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

## 2.2.3 线性表操作在链表中的实现

**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**

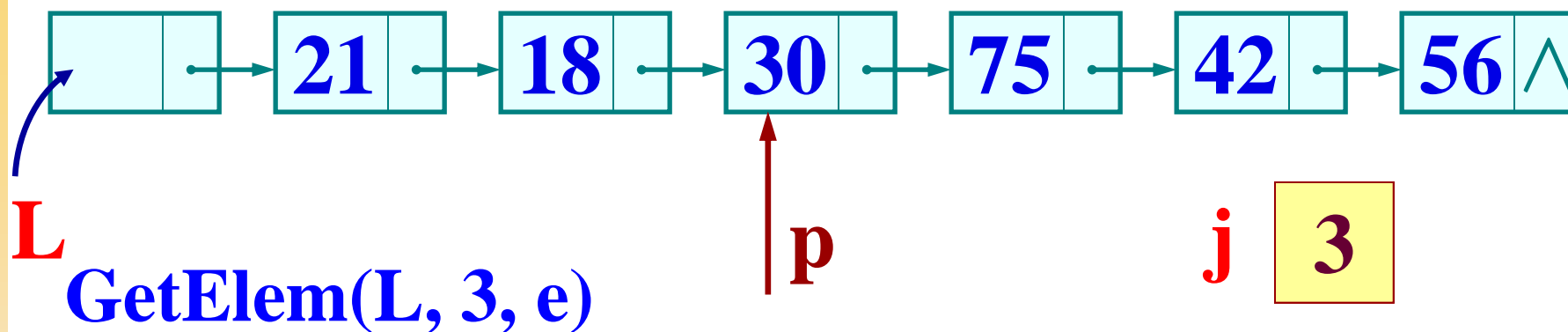
**ClearList(&L) // 重置线性表为空表**





# 单链表操作 GetElem(L, i, &e)

LinkNode \*p;



基本操作为：移动指针 **i** 次。

令指针 **p** 始终指向线性表中第 **j** 个数据元素。

结束条件：找到第 **i** 个结点，即 **j=i**；  
或者 **i** 大于链表长度，即 **p == null**;

# 单链表操作 GetElem(L, i, &e)

## ◆ 算法的基本过程

1. p指向第一个结点, 初始化计数器j为1
2. 顺指针向后查找, 直到  $j = i$  或 p 为空
3. 如果找不到第i个结点 ( $j > i$  或  $p = \text{null}$ ), 则返回**ERROR**
4. 取出第i个结点的数据, 放在e中, 返回**OK**

1.  $p = L \rightarrow \text{next}; j = 1;$
2.  $\text{while } (p \neq \text{NULL} \ \&\& \ j < i) \{ p = p \rightarrow \text{next}; ++j; \}$
3.  $\text{if } (p = \text{NULL} \ || \ j > i) \text{ return ERROR};$
4.  $e = p \rightarrow \text{data}; \text{ return OK};$

```
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; // 步骤1:初始化指针, 指向1st节点
    while (p!=NULL && j<i) { p = p→next; ++j; }
    //步骤2: 顺指针向后查找, 直到p指向第i个元素
    //或 p为空
    if ( p==NULL || j>i )
        return ERROR; // 步骤3:第 i 个元素不存在
    e = p→data; // 步骤4: 取得第 i 个元素
    return OK;
} // GetElem_L
```

算法时间复杂度为:  $O(\text{ListLength}(L))$



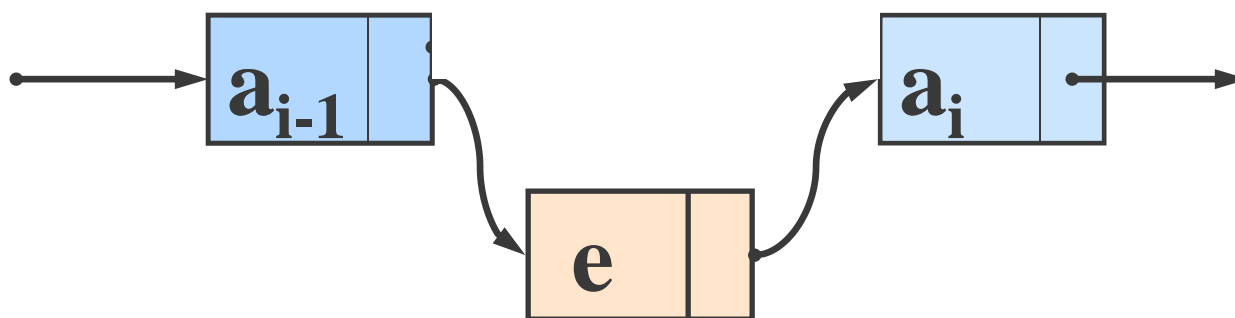
# 在单链表中按值查找

```
LinkNode *Search ( LinkList L, ElemType x ) {  
    //在链表中从头搜索其数据值为 x 的结点  
  
    LinkNode * p = L->next; //p为检测指针  
  
    while ( p != NULL && p->data != x )  
  
        p = p->next;  
  
    return p;  
  
} // Search
```

**while循环条件p != NULL 和 p->data != x不能错位!**

# 单链表操作 ListInsert(&L, i, e)

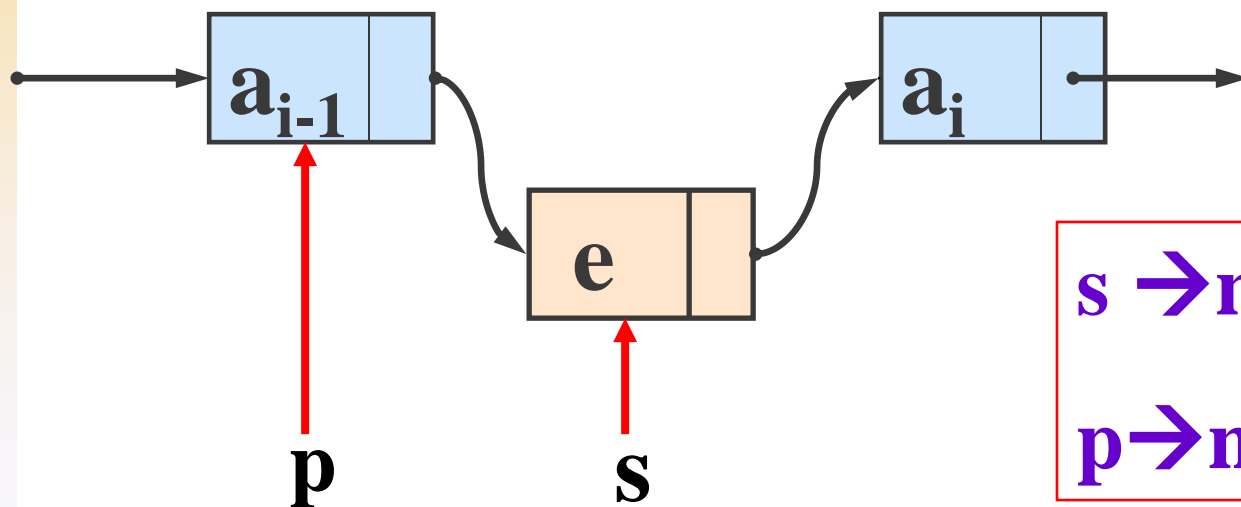
- ◆ 在第  $i$  个结点之前插入元素  $e$
- ◆ 有序对  $\langle a_{i-1}, a_i \rangle$  改变为  $\langle a_{i-1}, e \rangle$  和  $\langle e, a_i \rangle$



## 单链表操作 ListInsert(&L, i, e)

- ◆ 在链表中插入结点只需要修改指针。
- ◆ 修改第  $i-1$  个结点的指针。
- ◆ 基本操作:

¶ 找到线性表中第  $i-1$  个结点, 修改其后继指针



$s \rightarrow \text{next} = p \rightarrow \text{next};$   
 $p \rightarrow \text{next} = s;$

# 单链表操作ListInsert(&L, i, e) 过程总结

1. //p指向头结点, 初始化计数器 $j = 0$ ;
2. //顺指针向后查找, 直到  $j = i - 1$  或 p 为空;
3. //如果找不到第i个结点, 则返回**ERROR**;
4. //创建新结点, 若创建失败, 则返回**ERROR**;
5. //将新结点插入到i结点之前;

1.  $p = L; j = 0;$
2.  $\text{while } (p \neq \text{NULL} \ \&\& \ j < i - 1) \{ p = p \rightarrow \text{next}; ++j; \}$
3.  $\text{if } (p == \text{NULL} \ || \ j > i - 1) \text{ return ERROR};$
4.  $s = (\text{LinkList}) \text{ malloc } ( \text{sizeof } (\text{LNode}));$   
 $\text{If } (!s) \text{ exit(OVERFLOW)};$
5.  $s \rightarrow \text{data} = e; s \rightarrow \text{next} = p \rightarrow \text{next}; p \rightarrow \text{next} = s;$

```

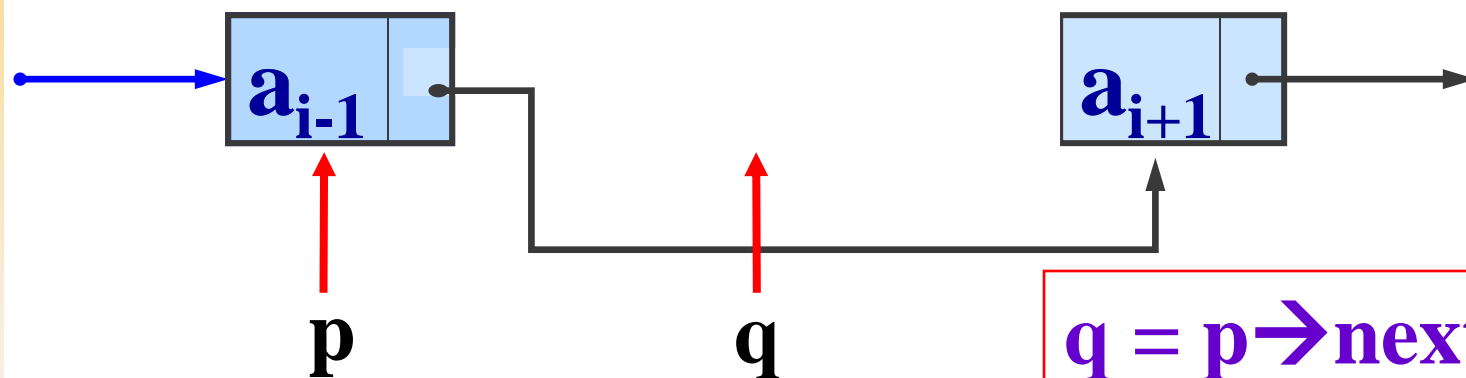
Status ListInsert_L(LinkList L, int i, ElemType e) {
    // L 为带头结点的单链表的头指针
    p = L;   j = 0;           //步骤1：初始化指针和计数器
    while (p && j < i-1)
        { p = p→next; ++j; }    //步骤2：寻找第i-1个结点
    if (!p || j > i-1) return ERROR; //步骤3：找不到i-1
    s = (LinkList) malloc ( sizeof (LNode));
    If (!s) exit(OVERFLOW);    //步骤4：创建新节点
    s→data = e;
                                //步骤5插入新节点
    s →next = p→next; p→next = s;
    return OK;
} // LinstInsert_L  算法的时间复杂度: O(ListLength(L))

```



# 单链表操作ListDelete (&L, i, &e)

- ◆ 在单链表中删除第  $i$  个结点
- ◆ 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。



$p = p \rightarrow \text{next} \rightarrow \text{next};$  X

$P \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next}$  ✓

$q = p \rightarrow \text{next};$   
 $p \rightarrow \text{next} = q \rightarrow \text{next};$   
 $\text{free}(q);$

# 单链表操作ListDelete (&L, i, &e)

## ◆ 算法的基本过程

1. p指向第一个结点, 初始化计数器j
2. 顺指针向后查找第i个结点, 并令p指向其前趋
3. 如果找不到第i个结点, 则返回ERROR
4. 从表中删除结点i(修改指针)
5. 将结点i的值赋给变量e, 并释放结点i所占的内存
6. 返回OK





```
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;           //步骤1:初始化指针和计时器
```

```
        //步骤2: 寻找第i个结点, 并令p指向其前趋
```

```
    while (p → next && j < i-1) { p = p→next; ++j; }
```

```
    if (!(p→next) || j > i-1)
```

```
        return ERROR;
```

```
        // 步骤3: 位置不合理
```

```
    q = p→next; p→next = q→next; //步骤4:删除节点
```

```
    e = q→data; free(q);
```

```
        //步骤5: 释放结点
```

```
    return OK;
```

```
} // ListDelete_L
```

算法的时间复杂度:  $O(\text{ListLength}(L))$

# 单链表操作 ClearList(&L)

```
void ClearList(LinkList L) {  
    // 将单链表L (带头结点)重新置为一个空表  
    p=L→next;           //步骤1:初始化指针  
    while (p) {          //步骤2:依次释放各个节点  
        q=p→next;   free(p);   p=q;  
    }  
    L→next = NULL;      //头节点指向空,即指针域置空  
} // ClearList
```

算法时间复杂度:  $O(\text{ListLength}(L))$



# 线性表的其它操作

**DestroyList\_L(LinkList &L) // 删除线性表**

**InitList\_L(LinkList &L) // 初始化线性表**

**ListTraverse\_L // 遍历线性表**

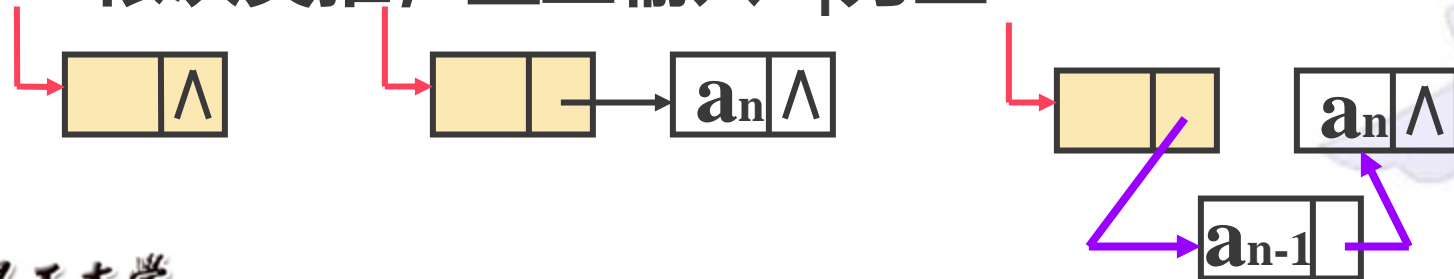
下面利用基本操作实现一个线性表的创建:

**CreateList\_L(LinkList &L, int n) // 创建线性表**



# CreateList\_L(LinkList &L, int n)

- ◆ 从表尾到表头逆序输入n个数据元素，建立带头结点的单链表
- ◆ 生成链表的过程是一个结点“逐个插入”的过程
- ◆ “前插法”操作步骤：
  1. 建立一个“空表”； InitList\_L(L);
  2. 输入数据元素 $a_n$ ，建立结点并插入；  
ListInsert\_L(L,  $a_n$ , e);
  3. 依次类推，直至输入 $a_1$ 为止

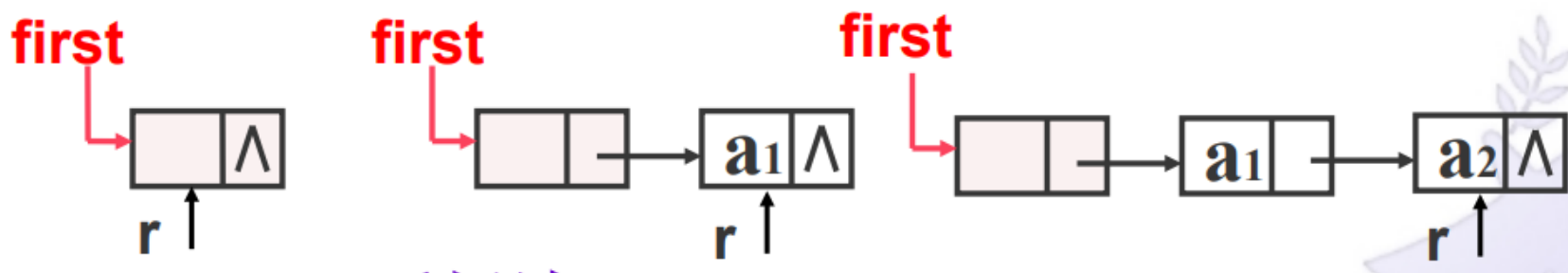


```
void CreateList_L(LinkList &L, int n) {  
    // 逆序输入 n 个数据，前插法建带头结点的单链表  
    ElemType e;  
    int i;  
    InitList_L(L);  
    printf("please input %d integer", n);  
    for (i = 1; i <= n; i++) {  
        scanf("%d", &e); // 输入元素值  
        ListInsert_L(L, 1, e); // 插入  
    }  
} // CreateList_L
```

算法的时间复杂度为:  $O(\text{Listlength}(L))$

# 后插法建立单链表

- ◆ 顺序输入  $n$  个数据元素，建立带头结点的单链表
- ◆ “后插法” 操作步骤：
  1. 每次将新结点加在插到链表的表尾；
  2. 设置一个尾指针  $r$ ，总是指向表中最后一个结点，新结点插在它的后面；
  3. 尾指针  $r$  初始时置为指向表头结点地址。





# 后插法建立单链表

```
void insertRear ( LinkList& first, ElemType endTag ) {  
    ElemType val;  
    LinkNode *s, *rear = first;           //rear指向表尾  
    scanf ( "%d", &val );                 //读入一数据  
    while ( val != endTag ) {  
        s = (LinkNode *) malloc ( sizeof (LinkNode ));  
        s->data = val;                     //创建新结点并赋值  
        rear->next = s; rear = s;         //插入到表尾  
        scanf ( "%d", &val );             //读入下一数据  
    } //while  
    rear->next = NULL;                     //表收尾  
} //insertRear
```

北京理工大学 算法的时间复杂度为:  $O(\text{Listlength}(L))$



# 上述单链表存在的问题

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

可以增加“表长”、“表尾指针”和“当前位置的指针”三个数据域



## 2.3.4 增加指针后的链表数据结构

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

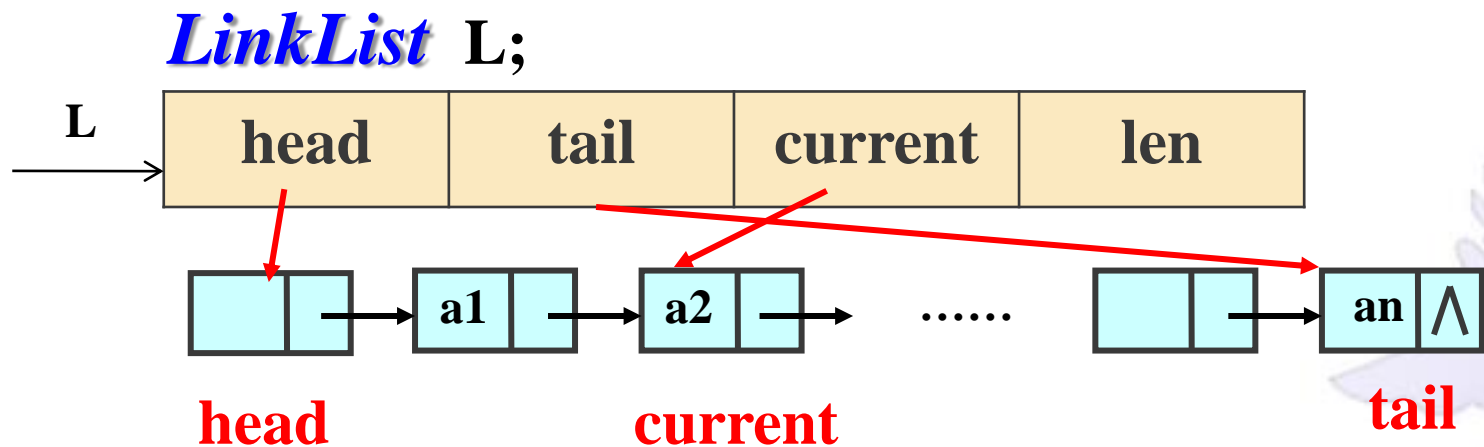
```
Status MakeNode( Link &p, ElemType e );  
// 分配由 p 指向的值为e的结点, 并返回OK,  
// 若分配失败, 则返回 ERROR
```

```
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;

```



# 增加指针后的链表的基本操作

head

tail

current

len

{结构初始化和销毁结构}

复杂度

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

```
// 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。
```

$O(1)$

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

```
// 销毁线性链表 L，L不再存在。
```

$O(n)$

# 增加指针后的链表的基本操作

head

tail

current

len

## {引用型操作}

**Status** ListEmpty ( LinkList L ); //判表空

**O(1)**

**int** ListLength( LinkList L ); // 求表长

**O(1)**

**Status** Next ( LinkList L );  
// 改变当前指针指向其后继

**O(1)**

**Status** Prior( LinkList L );  
// 改变当前指针指向其前驱

**O(n)**

**ElemType** GetCurElem ( LinkList L );  
// 返回当前指针所指数据元素

**O(1)**

复杂度

# 增加指针后的链表的基本操作

{加工型操作}

head

tail

current

len

复杂度

**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)**

## 2.3.5 静态链表

### ◆ 用数组实现的链式结构，称为静态链表

约定：下标为0的是备用空间

0			
1		5	← 头结点
2	a2	7	
3			
4	a4	0	← 尾结点
5	a1	2	
6			
7	a3	4	
8			
9			
10			

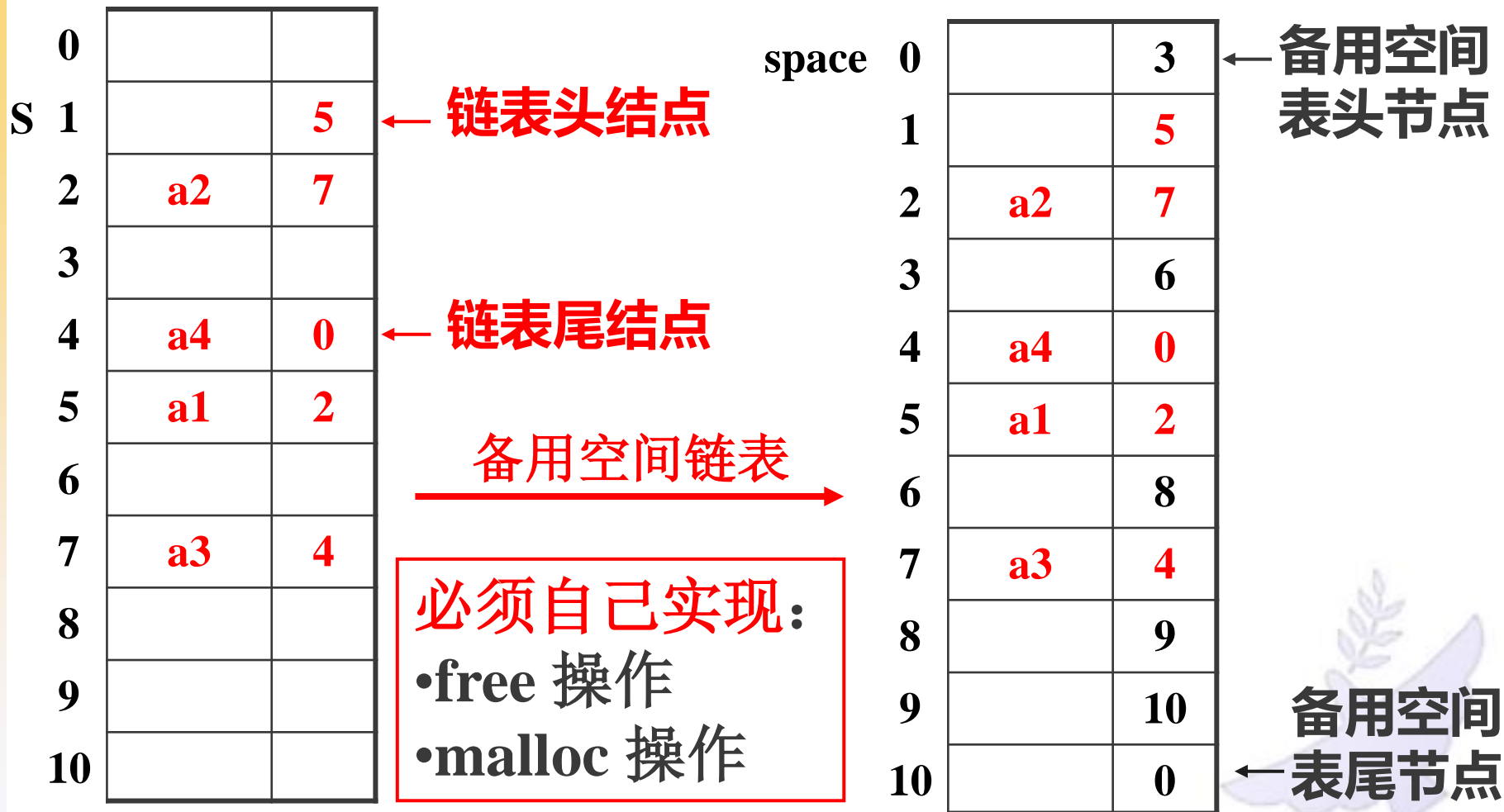
```
#define MAXSIZE 1000
    // 链表的最大长度

typedef struct{
    ElemType data;
    int cur;
} component,
    SLinkList[MAXSIZE];

SLinkList *space
```



# 备用空间链表



# 建备用空间链表

space 0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

```
void InitSpace_SL( )
```

```
{ //建备用空间链表
```

```
    // space为头指针， 0为空指针
```

```
    for(i=0;i< MAXSIZE-1; ++i)
```

```
        space[i].cur=i+1;
```

```
    space[MAXSIZE-1 ].cur=0;
```

```
} // InitSpace_SL
```

初始化备用空间

# 从备用链表中获取一个结点

space 0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

从备用链表中  
获取一个结点

space 0		2
<b>S=1</b> 1		0
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

# 从备用链表中获取一个结点

```
Status Malloc_SL( )
```

```
{ //若备用空间链表非空，返回分配结点的下标，  
  否则返回 0
```

```
    if (!space[0].cur) return 0;
```

```
    i=space[0].cur; //取出第一个备用结点
```

```
    space[0].cur=space[i].cur;//修改备用表
```

```
    return i;
```

```
} //Malloc_SL
```

# 将结点回收到备用链表中

```
void Free_SL(int k)
{ //将下标为k的结点回收到备用空间链表（头）

    space[k].cur=space[0].cur;
    space[0].cur=k;

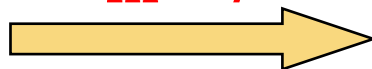
} //Free_SL
```

# 静态链表的插入 ListInsert\_SL(int S, int 5, WANG)

space	0		7
S=1	1		2
	2	ZHAO	3
	3	QIAN	4
	4	SUN	5
	5	LI	6
	6	ZHOU	0
	7		8
	8		9
	9		10
	10		0

Li之后插入  
WANG

分配新结点  
m=7



space	0		8
S=1	1		2
	2	ZHAO	3
	3	QIAN	4
	4	SUN	5
	5	LI	7
	6	ZHOU	0
	7	WANG	6
	8		9
	9		10
	10		0



◆ 1、寻找第 $i-1$ 个元素结点，设其下标为 $k$

¶ **m=Malloc\_SL();**

```
space[m].data=e;
```

```
space[m].cur= space[k].cur ;
```

```
space[k].cur=m ;
```



# 静态链表的插入算法

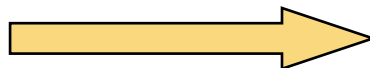
```
Status ListInsert_SL(int S, int i, ElemType e)
{ //S静态链表头结点下标,在第i个元素前插入e
  k=S; j=0; //寻找第i-1个元素结点
  while (k&& j<i-1){k=space[k].cur ; ++j;}
  if(!k||j>i-1) return ERROR;
  m=Malloc_SL(); //分配新结点
  if (m == 0) return ERROR;
  space[m].data=e;
  space[m].cur= space[k].cur ; //插入新结点
  space[k].cur=m ;
  return OK;
} // ListInsert_SL
```



# 静态链表的删除 ListDelete\_SL(int S, int 3, e)

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

删除  
SUN

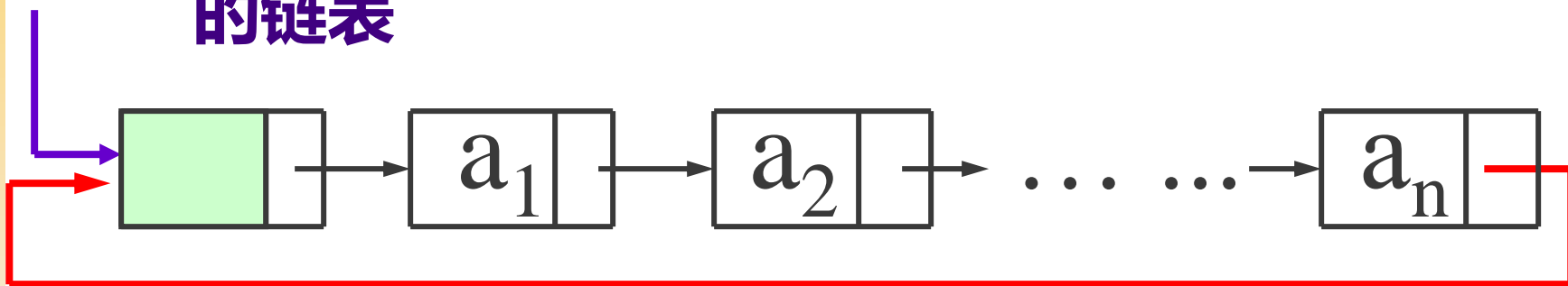


space	0		4
S	1		2
	2	ZHAO	3
	3	QIAN	5
	4		8
	5	LI	6
	6	ZHOU	7
	7	WANG	0
	8		9
	9		10
	1		0
	0		

# 其它链表

## ◆ 1) 单向循环链表

📌 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

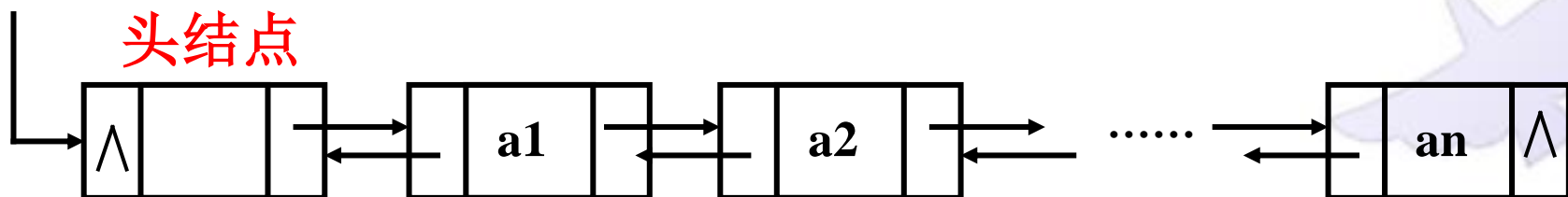
“判别链表中**最后一个结点的条件**”：

- ✓ 单链表中是“后继是否为空”
- ✓ 单向循环链表中是“后继是否为头结点”。

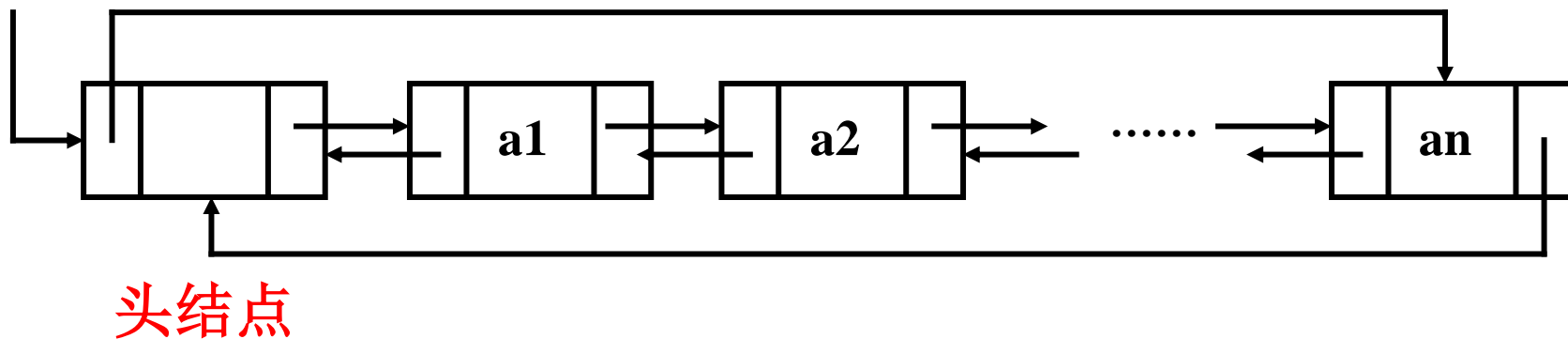
# 其它链表

## 2) 双向链表

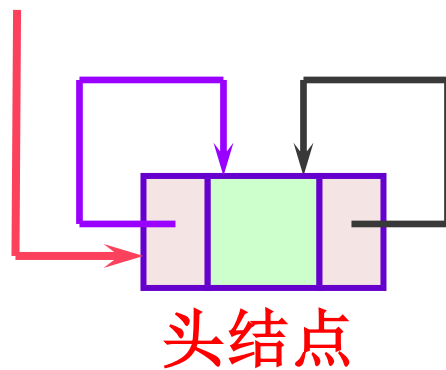
```
typedef struct DuLNode {  
    ElemType      data; // 数据域  
    struct DuLNode *prior;  
    // 指向前驱的指针域  
    struct DuLNode *next;  
    // 指向后继的指针域  
} DuLNode, *DuLinkList;
```



### 3) 双向循环链表：循环双链表



空表



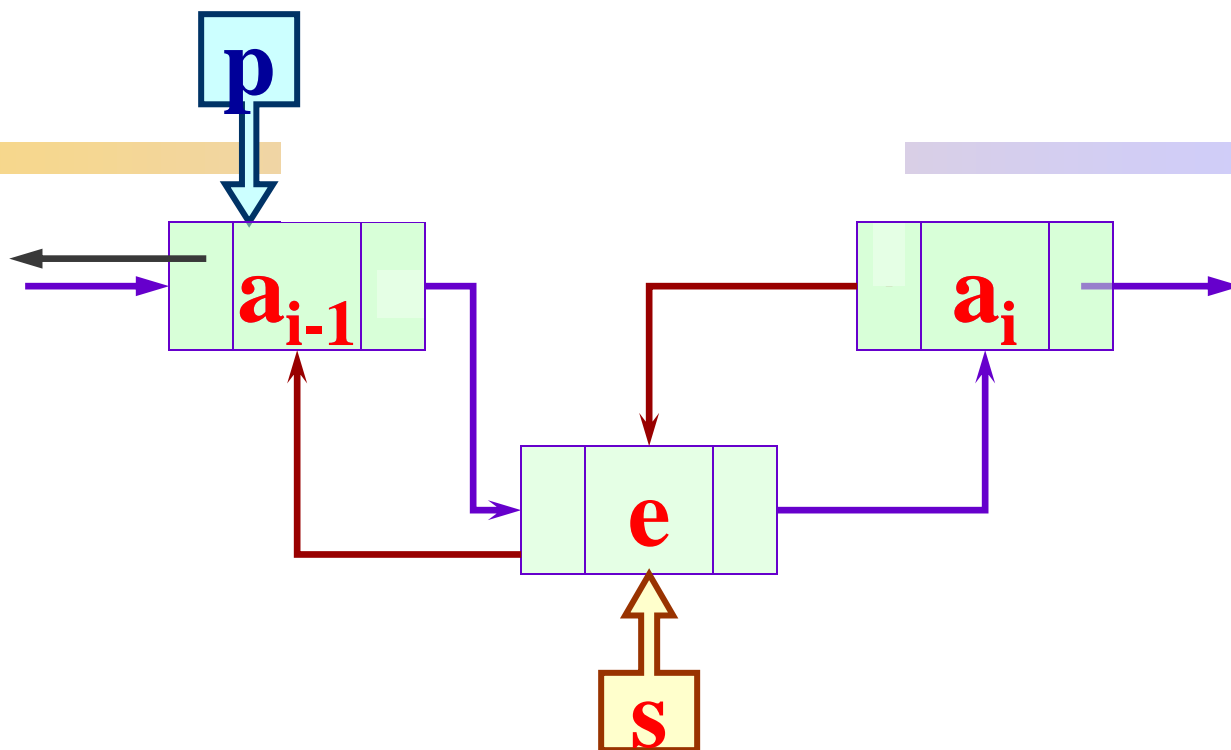


# 双向链表的操作特点

- ◆ “查询” 和单链表相同（可以从两个方向进行）。
- ◆ “插入” 和 “删除” 时需要同时修改两个方向上的指针。



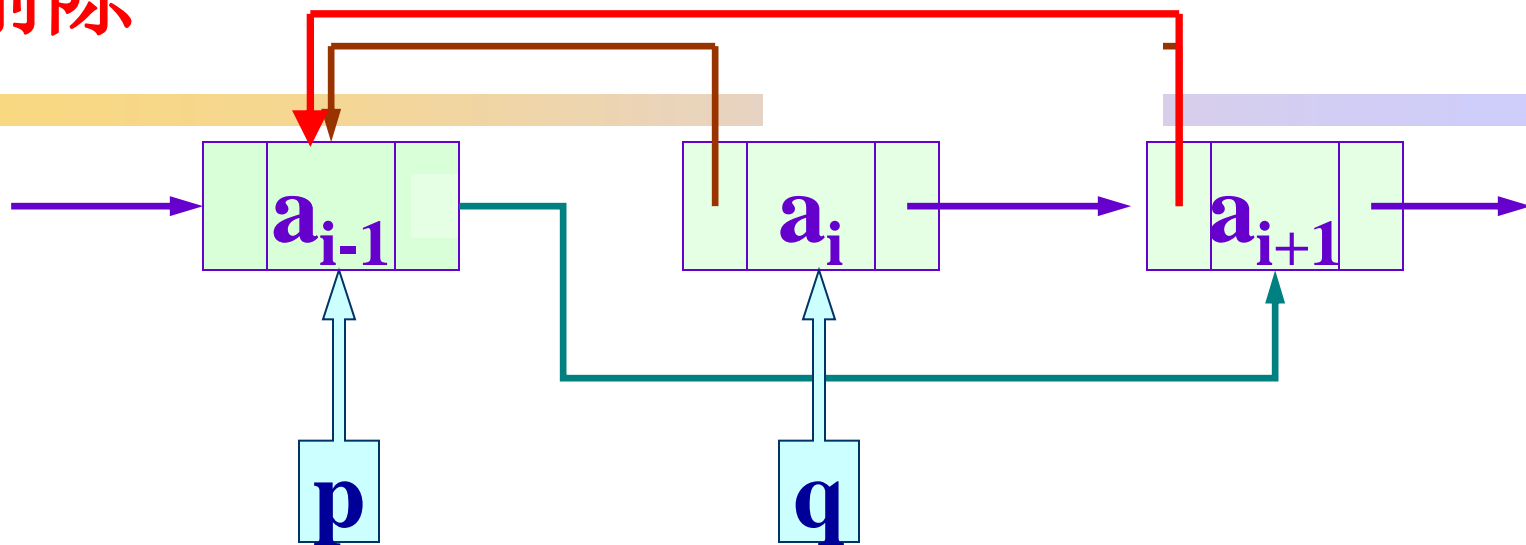
# 插入



$s \rightarrow \text{next} = p \rightarrow \text{next}; \quad p \rightarrow \text{next} = s; //$ 修正后继

$s \rightarrow \text{next} \rightarrow \text{prior} = s; \quad s \rightarrow \text{prior} = p; //$ 修正前驱

# 删除



$q = p \rightarrow \text{next}$

$p \rightarrow \text{next} = q \rightarrow \text{next};$

$p \rightarrow \text{next} \rightarrow \text{prior} = p;$

$\text{free}(q);$



# 顺序表与链表的比较

## 1、基于空间的比较

### ◆ 存储分配的方式

🔑 顺序表的存储空间可以是静态分配的，也可以是动态分配的。

🔑 链表的存储空间是动态分配的。

◆ 存储密度 = 结点数据本身所占的存储量 / 结点结构所占的存储总量

🔑 顺序表的存储密度 = 1

🔑 链表的存储密度 < 1







# 顺序表与链表的比较

## 2、基于时间的比较

### ◆ 存取方式

‖ 顺序表可以随机存取，也可以顺序存取。

‖ 链表只能顺序存取。

### ◆ 插入/删除时移动元素个数

‖ 顺序表平均需要移动近一半元素。

‖ 链表不需要移动元素，只需要修改指针。

‖ 若插入/删除仅发生在表的两端，宜采用带尾指针的循环链表。



# 线性表应用

## ——一元多项式表示及计算



# 一元多项式

$$p_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$$

- ◆ 在计算机中，可以用一个**线性表**来表示：

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

- ◆ 但对于  $S(x) = 1 + 3x^{10000} - 2x^{20000}$ ，该表示方法是否合适？

- ◆ **一元稀疏多项式**

$$p_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_nx^{e_n}$$

其中： $p_i$  是指数为  $e_i$  的项的非零系数，

$$0 \leq e_1 < e_2 < \dots < e_m = n$$



# 一元多项式

$$p_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_nx^{e_n}$$

◆ 可以下列线性表表示：

$$\uparrow ((p_1, e_1), (p_2, e_2), \dots, (p_n, e_n))$$

◆ 例如：

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

↑ 可用线性表((7, 3), (-2, 12), (-8, 999))表示



# 一元多项式的ADT

**ADT Polynomial {**

**数据对象:**

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

$\text{TermSet}$  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:**

$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n$

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

**基本操作: .....**

**}// ADT Polynomial**

# 一元多项式的ADT—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  $P$ 。



# 一元多项式的ADT—基本操作

## ◆ 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：  
$$Pa = Pa + Pb。$$

## ◆ SubtractPolyn ( &Pa, &Pb )

## ◆ MultiplyPolyn(&Pa, &Pb)



# 一元多项式的实现

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

//以带表头结点的有序链表表示多项式为例

## 单链表的结点定义

```
typedef struct node
{ int    coef, exp; // 系数, 指数
  struct node *next;
} ITEM;
```

coef	exp	next
------	-----	------





# 一元多项式加法

$$A(x) = 7 + 3x^2 + 9x^8$$

$$B(x) = 8x + 22x^2 - 9x^8 + 5x^{17}$$

$$C(x) = A(x) + B(x) = 7 + 11x + 22x^2 + 5x^{17}$$

运算规则：假设：p、q 分别指向A、B中某一结点，p、q初值指向第一结点，结果放入A链

**比较**  
p->exp 与 q->exp

- $p \rightarrow \text{exp} < q \rightarrow \text{exp}$ : 结点 p 是**和多项式**中的一项，指针 p 后移一个结点，q 不动
- $p \rightarrow \text{exp} > q \rightarrow \text{exp}$ : 结点 q 是**和多项式**中的一项，将q插在p之前，q后移，p不动
- $p \rightarrow \text{exp} = q \rightarrow \text{exp}$ :
  - =0**: 从A表中删去p, 释放p,q, p,q后移
  - ≠0**: 修改p系数域, 释放q, p,q后移

**直到 p 或 q == NULL**

若q=NULL, 结束

若p=NULL, 将B中剩余部分连到A上

# 一元多项式加法

$$A(x) = 7 + 3x^2 + 9x^8$$

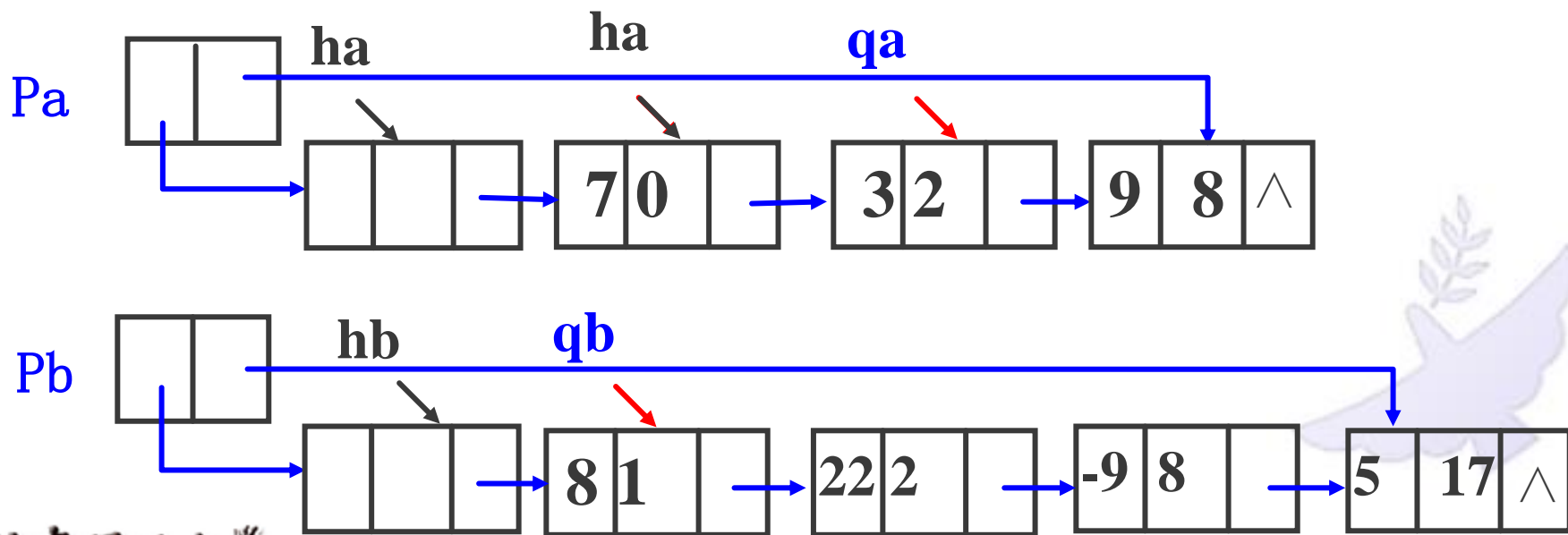
$$B(x) = 8x + 22x^2 - 9x^8 + 5x^{17}$$

$$C(x) = A(x) + B(x) = 7 + 8x + 25x^2 + 5x^{17}$$

```
typedef struct P
{ ITEM * head, * tail;
} Ps;
Ps Pa, Pb;
```

// 指向头结点和表尾结点

qa->exp < qb->exp 成立

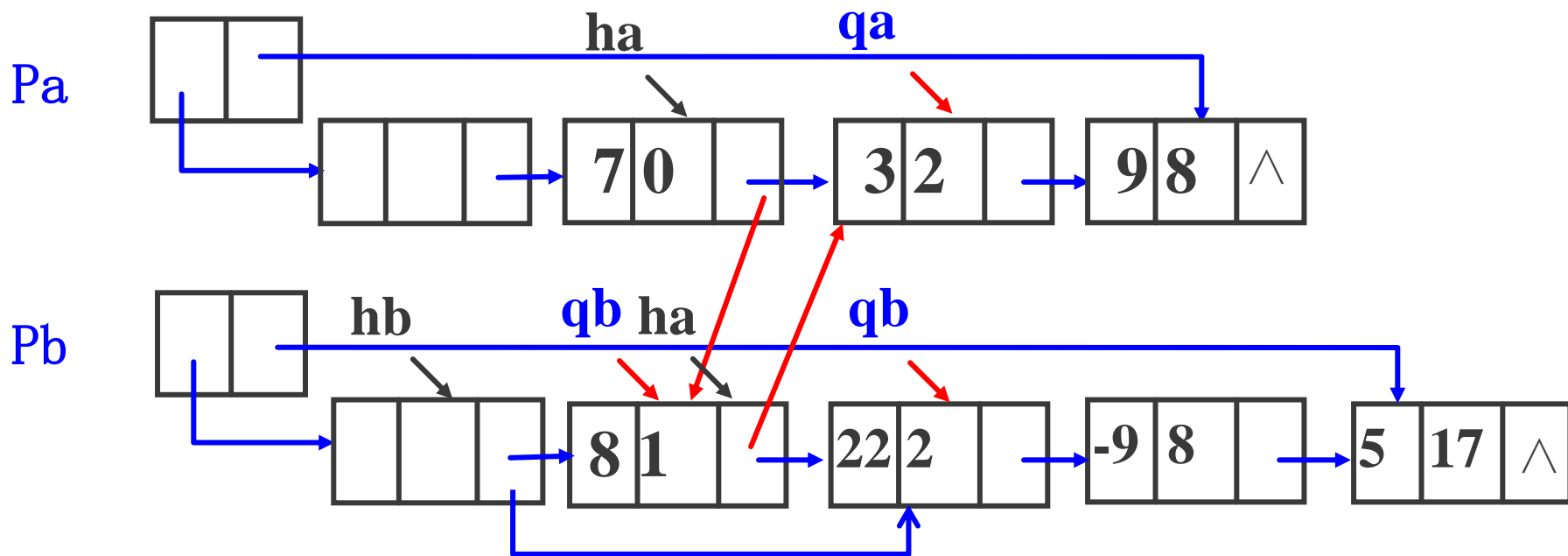


# 一元多项式加法

$$A(x) = 7 + 3x^2 + 9x^8$$

$$B(x) = 8x + 22x^2 - 9x^8 + 5x^{17}$$

$$C(x) = A(x) + B(x) = 7 + 8x + 25x^2 + 5x^{17}$$



$qa \rightarrow \text{exp} > qb \rightarrow \text{exp}$  成立

$ha \rightarrow \text{next} = qb$

$ha = qb$

$qb = qb \rightarrow \text{next}$

$ha \rightarrow \text{next} = qa$

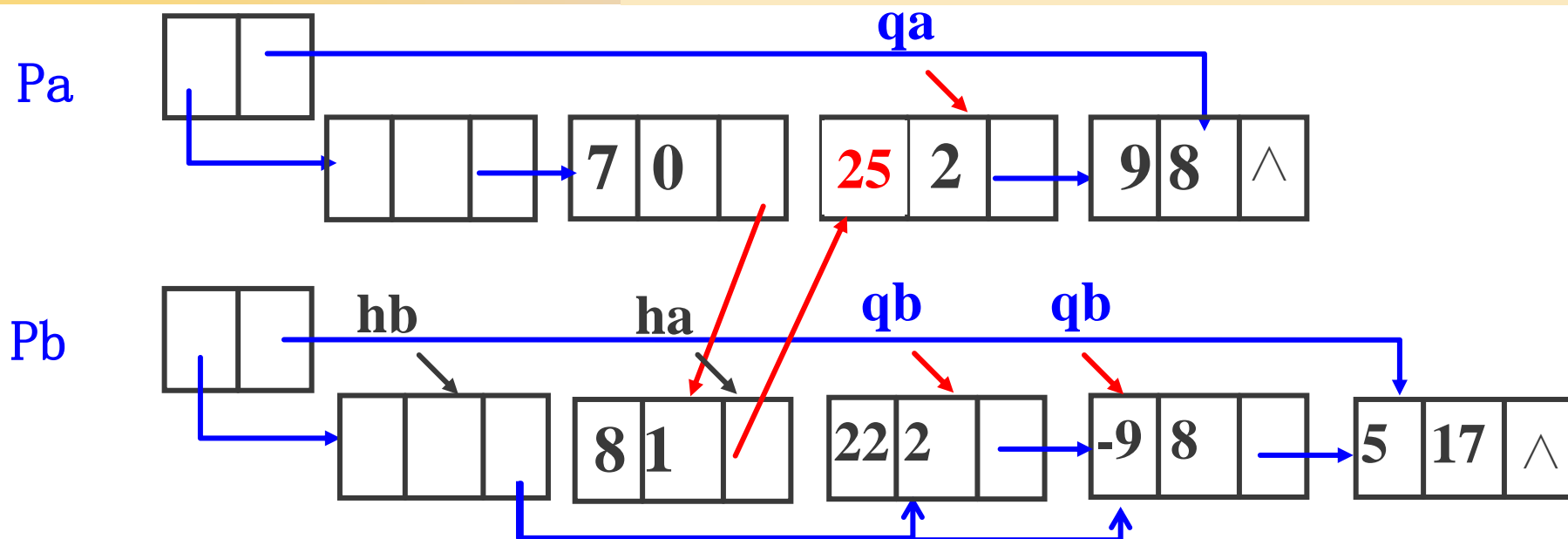
$hb \rightarrow \text{next} = qb$

# 一元多项式加法

$$A(x) = 7 + 3x^2 + 9x^8$$

$$B(x) = 8x + 22x^2 - 9x^8 + 5x^{17}$$

$$C(x) = A(x) + B(x) = 7 + 8x + 25x^2 + 5x^{17}$$



**qa->exp = qb->exp 成立**

**qa->coef += qb->coef**

**qb = qb->next**

**free( hb->next )**

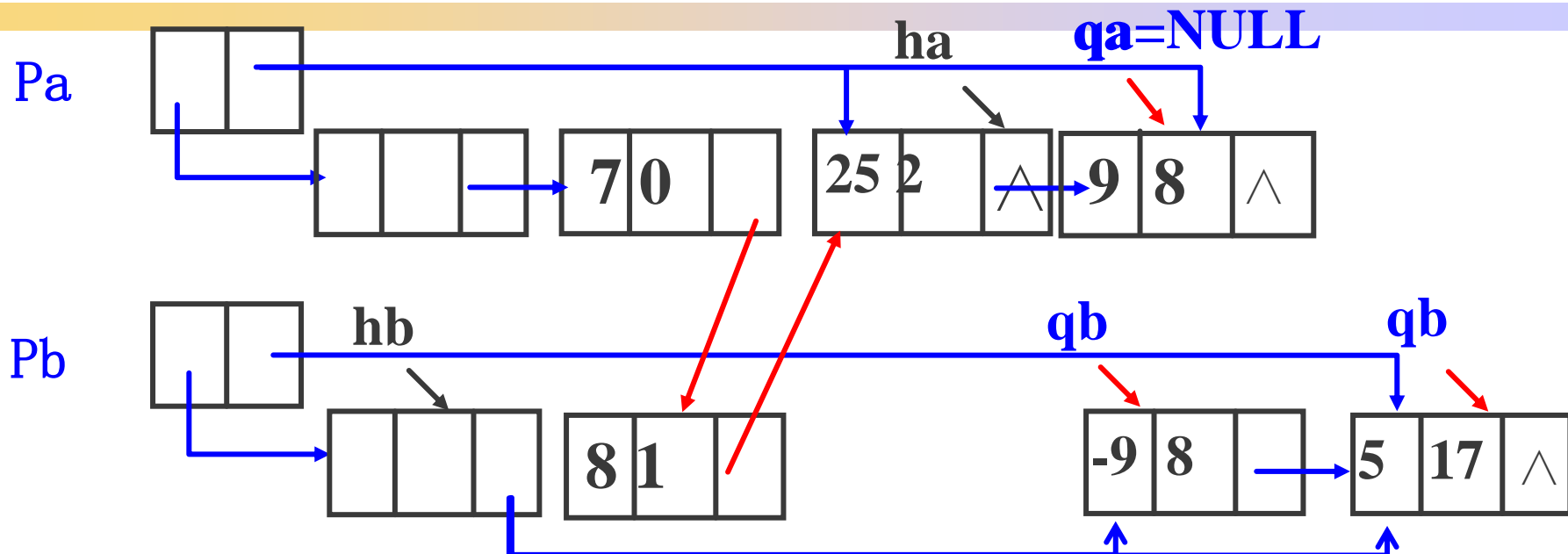
**hb->next = qb**

# 一元多项式加法

$$A(x) = 7 + 3x^2 + 9x^8$$

$$B(x) = 8x + 22x^2 - 9x^8 + 5x^{17}$$

$$C(x) = A(x) + B(x) = 7 + 8x + 25x^2 + 5x^{17}$$



$qa \rightarrow exp = qb \rightarrow exp$  成立

$qa = qa \rightarrow next$

$Pa.tail = ha$

$free(ha \rightarrow next)$

$ha \rightarrow next = qa$

$qb = qb \rightarrow next$

$free(hb \rightarrow next)$

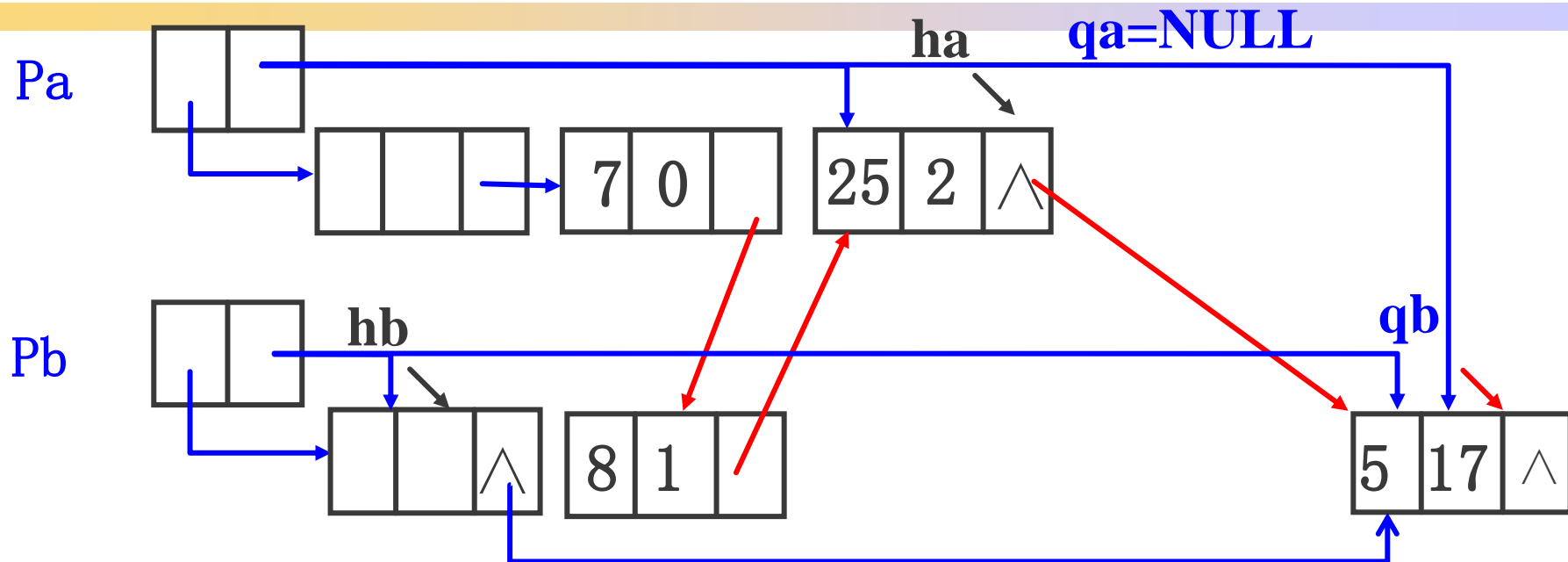
$hb \rightarrow next = qb$

# 一元多项式加法

$$A(x) = 7 + 3x^2 + 9x^8$$

$$B(x) = 8x + 22x^2 - 9x^8 + 5x^{17}$$

$$C(x) = A(x) + B(x) = 7 + 8x + 25x^2 + 5x^{17}$$



qa 结束

**ha->next = qb**

**hb->next = NULL**

**Pa.tail = Pb.tail**

**Pb.tail = hb**

# 顺序表和链表的比较

## ◆ 顺序表的主要特点

- ┆ 没有使用指针，不用花费额外开销
- ┆ 线性表元素的读访问非常简洁便利
- ┆ 插入、删除运算时间代价 $O(n)$

## ◆ 链表的主要特点

- ┆ 无需事先了解线性表的长度，允许线性表的长度动态变化
- ┆ 插入、删除运算时间代价 $O(1)$ ，但找第  $i$  个元素运算时间代价 $O(n)$
- ┆ 每个元素都有结构性存储开销

## ◆ 结论

- ┆ 顺序表：适合存储静态数据（插入、删除不频繁的情况）
- ┆ 链表：适合存储动态变化数据



# 线性表应用场合的选择

## ◆ 顺序表不适用的场合

- 🔧 经常插入删除时，不宜使用顺序表
- 🔧 线性表的最大长度也是一个重要因素

## ◆ 链表不适用的场合

- 🔧 当读操作远多于插入、删除操作时，不应选择链表
- 🔧 当指针的存储开销远大于数据内容时，应该慎重选择







# 本章学习要点

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。



# 线性表基本内容

## 逻辑结构

基本特性，基本操作与效率

## 基本概念

表长，空表，元素的位序

## 线性表

## 实现

### 顺序存储

静态  
动态

### 数据对象

数据关系  
基本操作

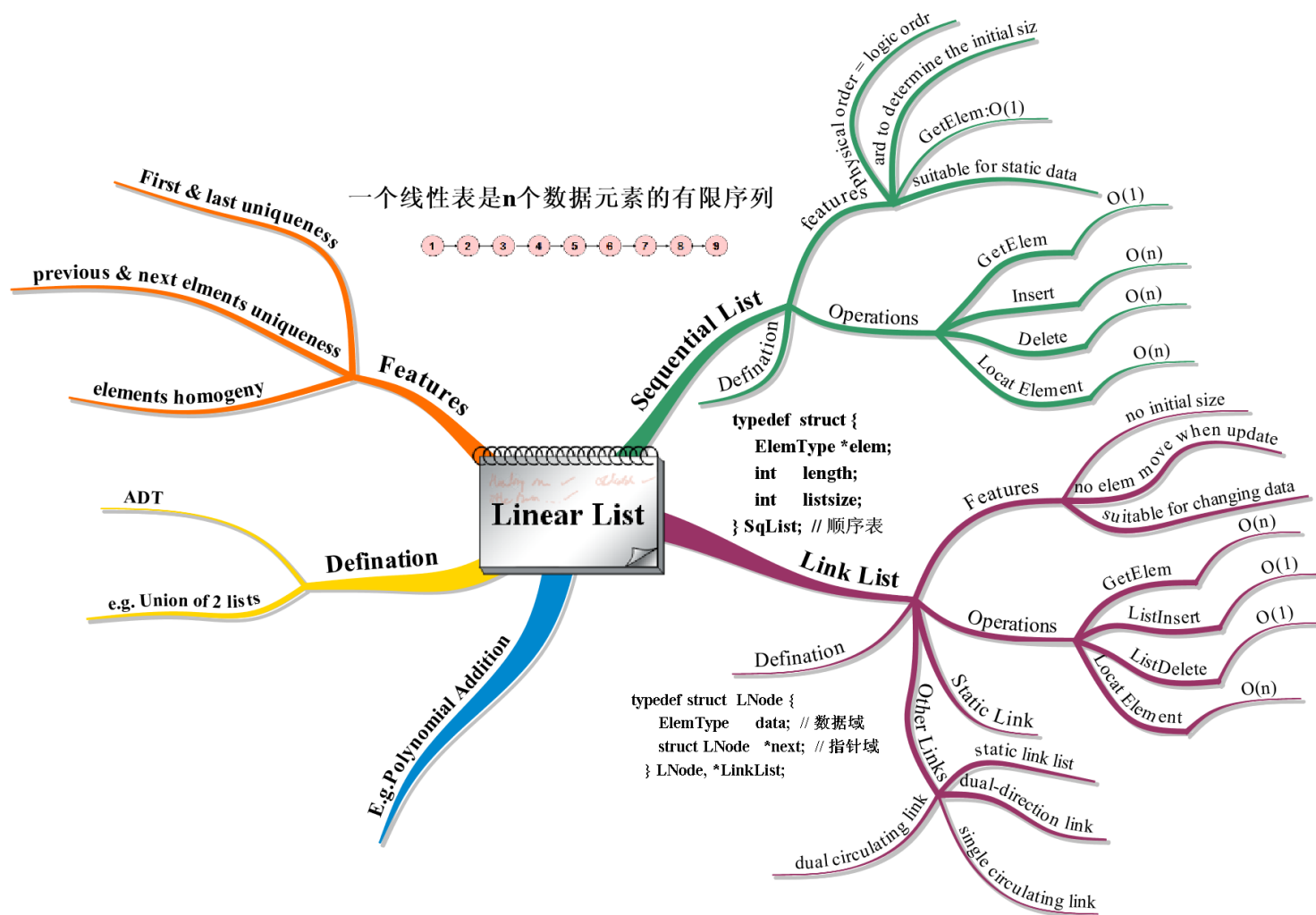
### 链式存储

静态  
动态

### 链表基本形态

数据对象/关系/基本  
操作  
特殊形态链表

# Review





## End of Chapter 2

**史树敏**  
**计算机学院**

