



第二章 线性表(Linear List)



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.





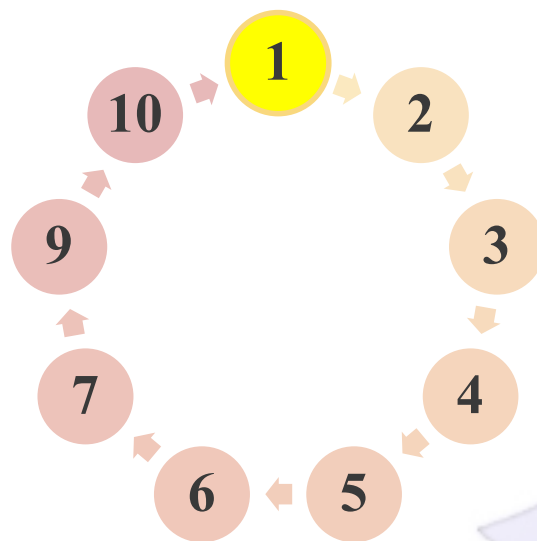
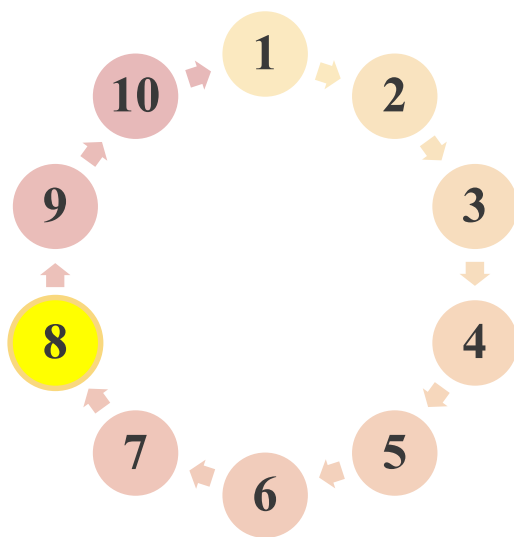
线性表的实例

- ◆ 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

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


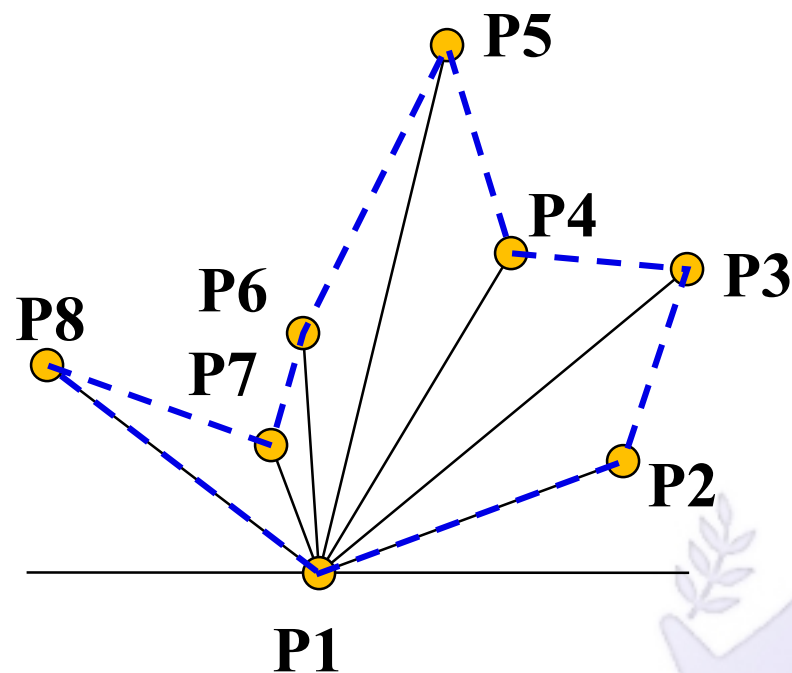
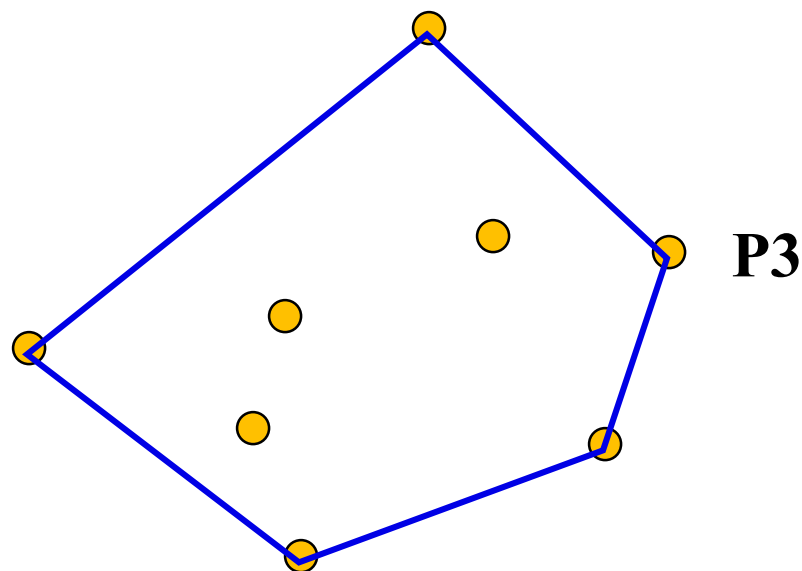
线性表的实例

- ◆ 实例6：约瑟夫环问题：已知 n 个人（以编号1, 2, 3... n 分别表示）围坐在一张圆桌周围。从编号为 k 的人开始报数，数到 m 的那个人出列；他的下一个人又从1开始报数，数到 m 的那个人又出列；依此规律重复下去，直到圆桌周围的人全部出列
- ◆ $n=10; k=6; m=3;$



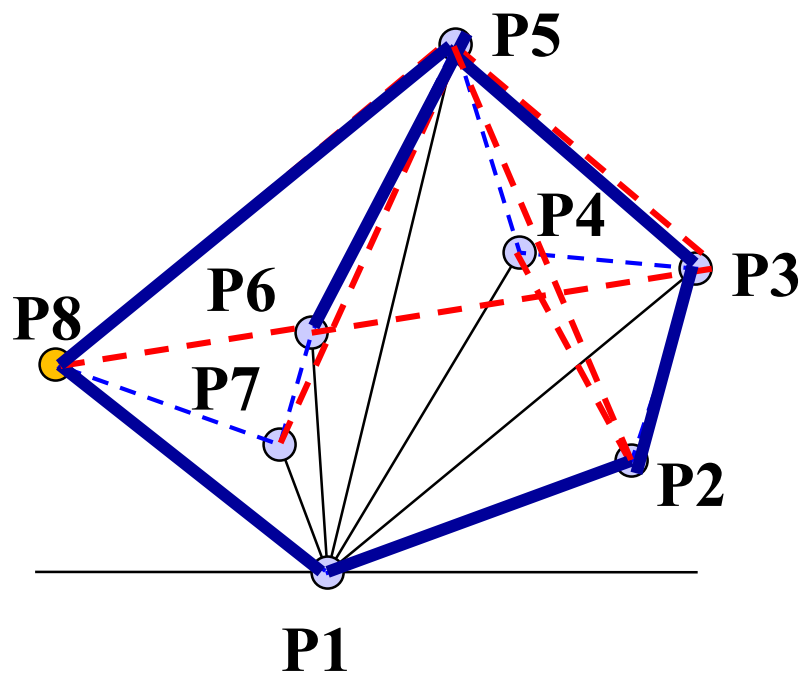
线性表的实例

◆ 实例7：点的凸包问题





◆ 实例7：点的凸包问题



线性表的实例

- ◆ 实例8：括号匹配： $(a+b(u-c*t))$
- ◆ 实例9：停车场安排





线性表的实例

◆ 机场跑道调度

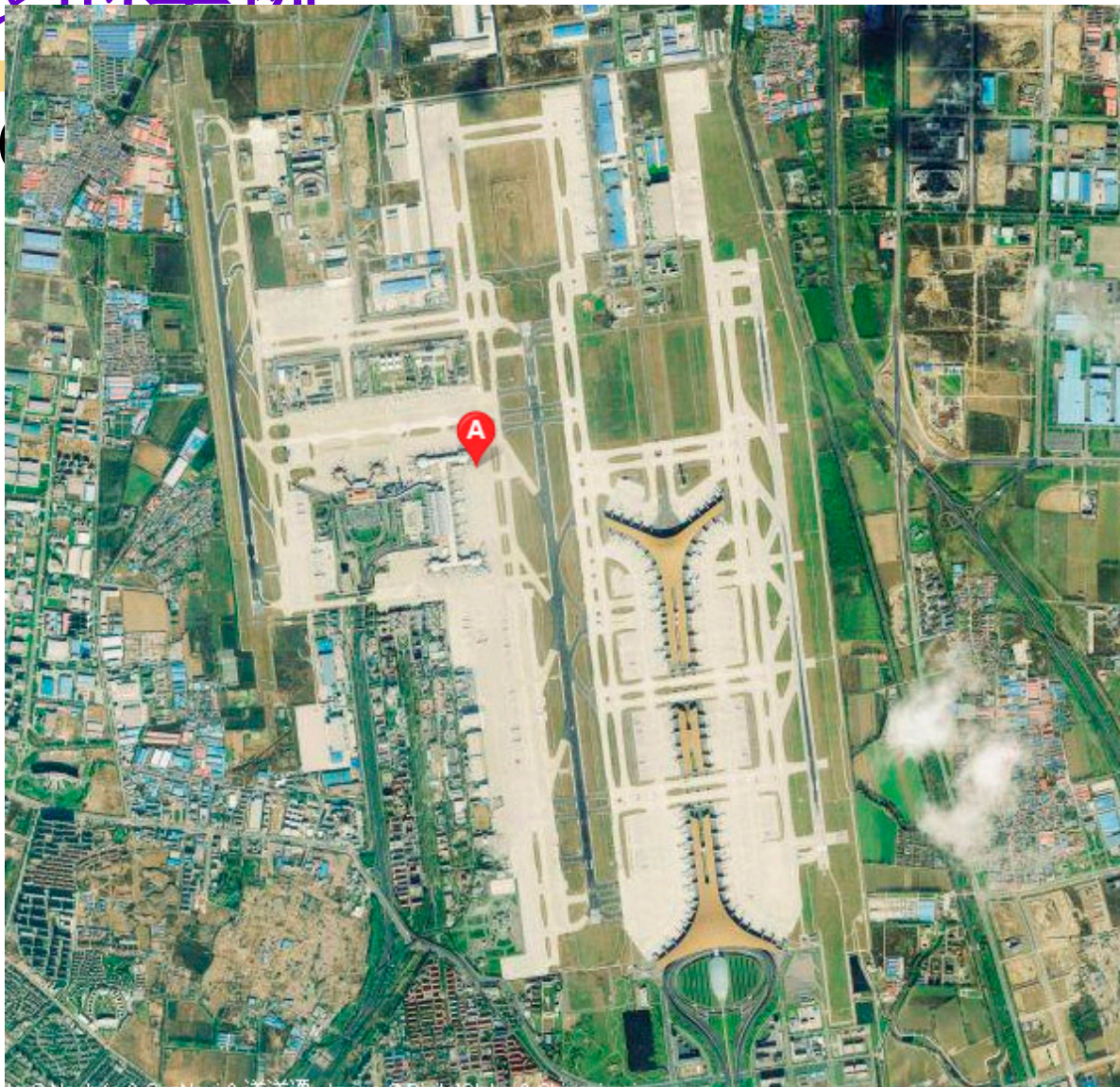
◆ 原则：

- ✎ 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

◆ 实例10



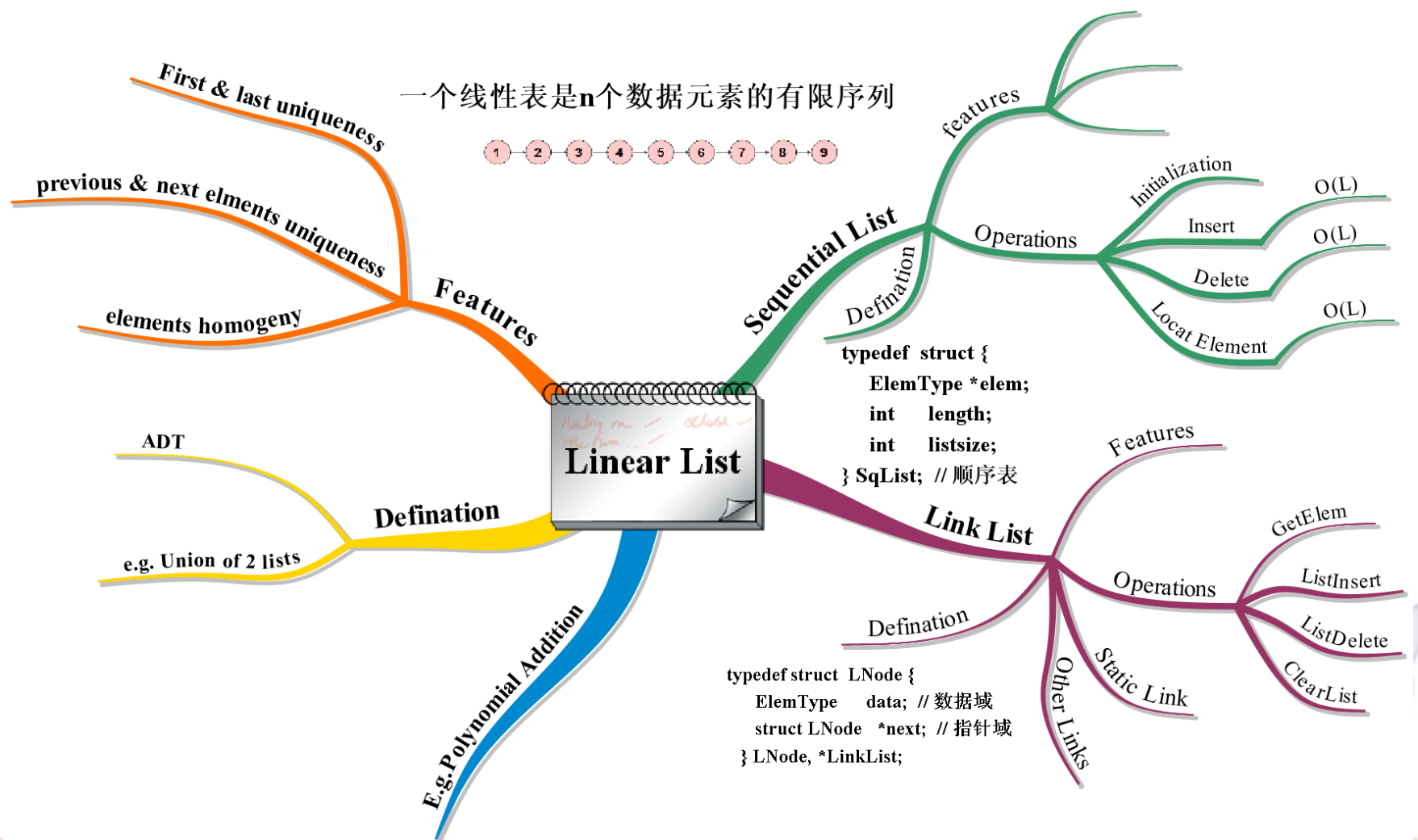


本章内容

- ◆ 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



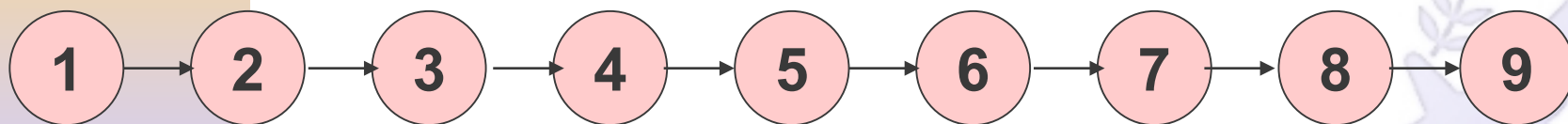
线性表





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[严]

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



- ◆ **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





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**

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, &link_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link_e返回它的后继，否则操作失败，link_e 无定义



- ◆ **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





- ◆ **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





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

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 $A = A \cup B$

- ◆ **基本思想**：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，必要时扩大线性表 **LA**。
- ◆ **控制程序**：for ($i = 1$; $i \leq Lb_len$; $i++$)

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
-----------	---	---	---	---	----	---	---	----	---	---



- 北京理工大学 高春晓

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<=la-len)&&(j<=lb-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

- ◆ 算法步骤:
- ◆ 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LC的长度加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？

3	5	8	8	11	11	11	12
---	---	---	---	----	----	----	----

2	8	8	8	11	11	12	12	15	20
---	---	---	---	----	----	----	----	----	----



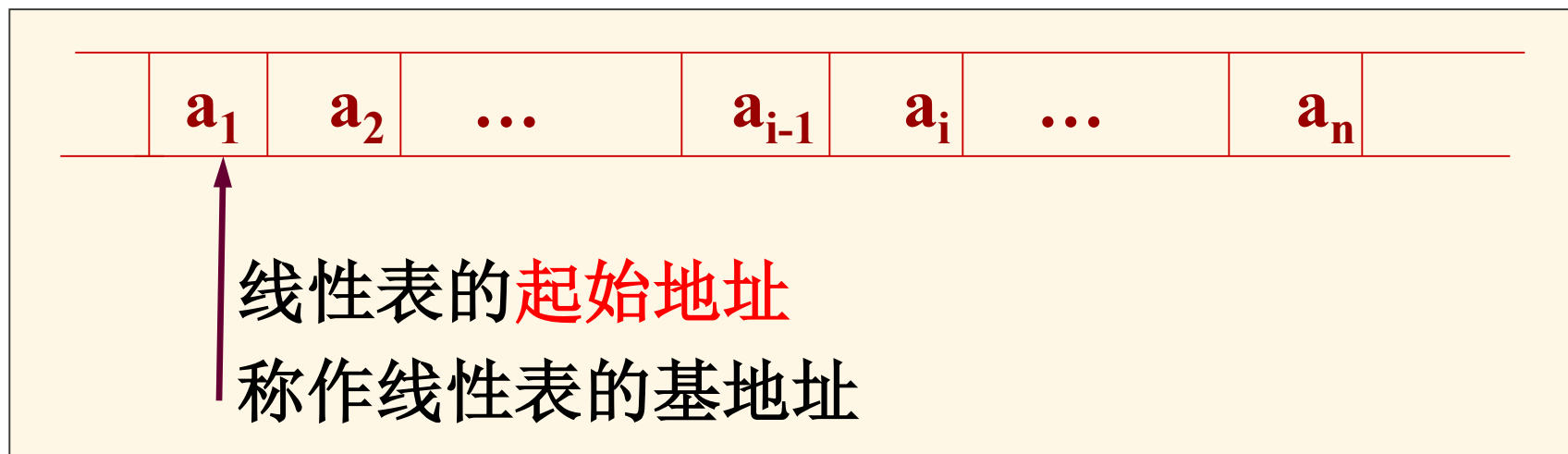
2.2 顺序表(Sequential List)

- ◆ 2.2.1 顺序表示及其特点
- ◆ 2.2.2 数据结构定义及初始化
- ◆ 2.2.3 顺序表的查找操作
- ◆ 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 $LC = LA + LB$



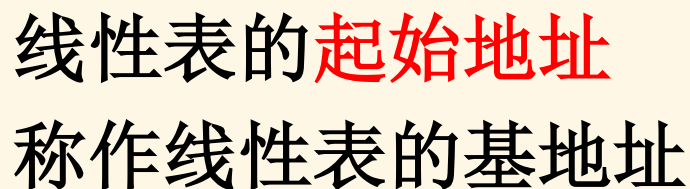
2.2.1 顺序表示及其特点

- ◆ **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。



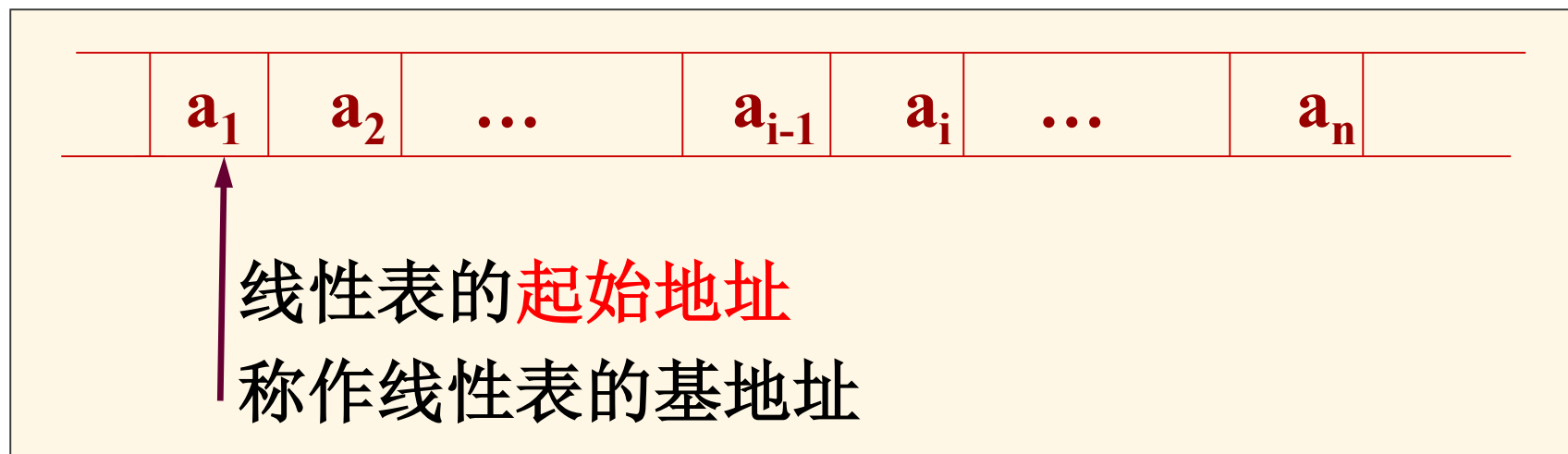


- ◆ 以“**存储位置相邻**”表示有序对 $\langle a_{i-1}, a_i \rangle$
 - ⌈ 即: $LOC(a_i) = LOC(a_{i-1}) + C$
 - ⌈ C 是一个数据元素所占存储量
- ◆ 所有数据元素的存储位置均取决于第一个数据元素的存储位置
 - ⌈ $LOC(a_i) = \underline{LOC(a_1)} + (i-1) \times C$



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。



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）  
} SeqList; // 俗称顺序表
```



2.2.2 顺序表数据结构定义

◆ 动态定义

data	n	maxSize
------	---	---------

```
#define initSize 100
```

```
// 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
```

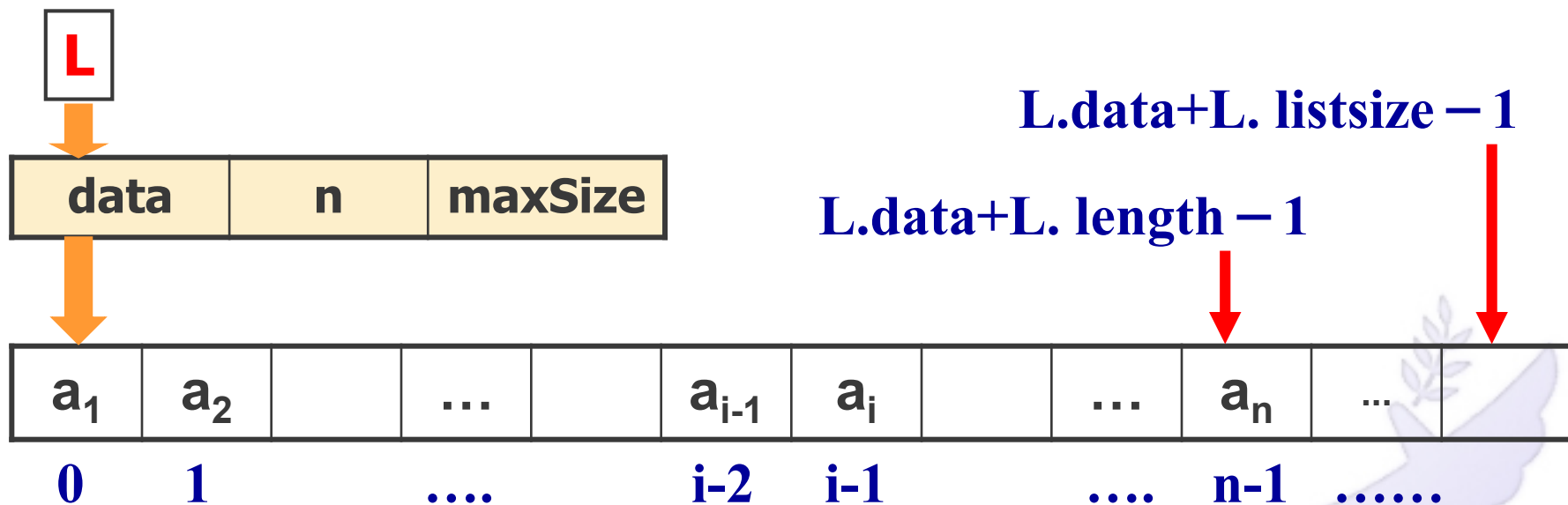
```
typedef struct {  
    DataType *data; // 存储空间基址  
    int      maxSize; // 分配的存储容量  
                // (以sizeof(DataType)为单位)  
    int      n; // 线性表当前长度，即元素个数  
} SeqList; // 俗称顺序表
```

顺序表

```
typedef struct {  
    DataType *data;  
    int      n;  
    int      maxSize;  
} SeqList; // 顺序表
```

SeqList L;

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 $L.data[i - 1]$.



顺序表的初始化操作

```
Status InitList_Sq( SeqList& L ) {  
    // 构造一个空的线性表  
    L.data = (DataType *) malloc  
        (initSize * sizeof (DataType ));  
    if ( L.data == NULL)  
        { printf (“存储分配失败!\n”); exit (1); }  
    L.n = 0;  
    L.maxSize = initSize;  
    return OK;  
    } // InitList_Sq
```

```
typedef struct {  
    DataType *data;  
    int    n;  
    int    maxSize;  
    } SeqList; // 顺序表
```



顺序表的查找操作

- ◆ 按值查找：在顺序表中从头查找结点值等于给定值 x 的结点

```
int Find ( SeqList& L, DataType x ) {  
    for ( i = 0; i < L.n; i++ )  
        if ( L.data[i] == x ) return i;    //查找成功  
    return -1;                             //查找失败  
}
```





查找算法性能分析

- ◆ 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

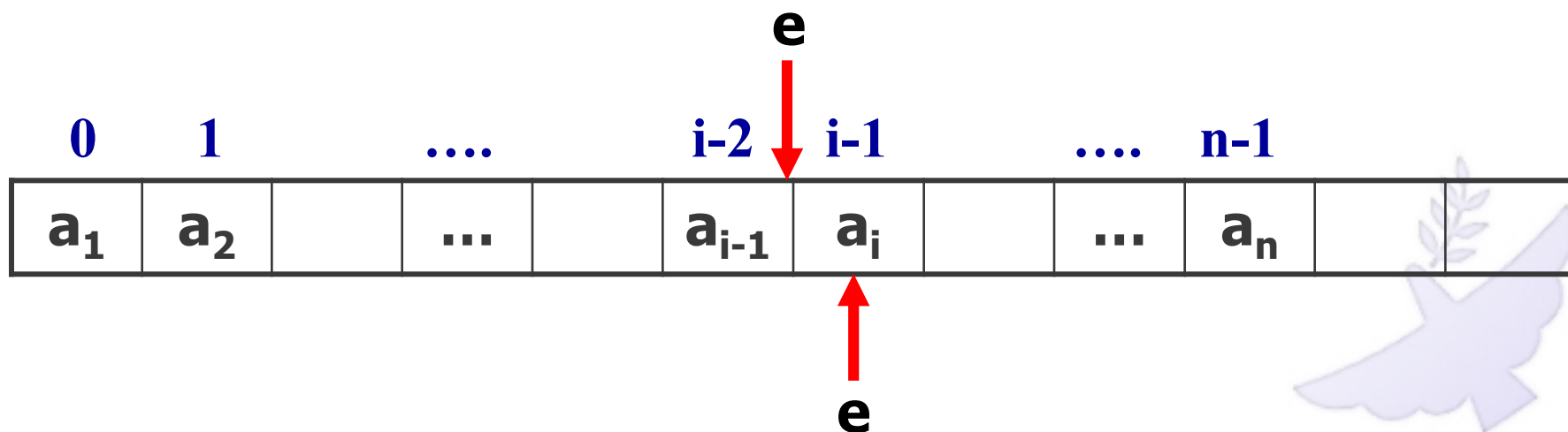
$$\begin{aligned} ACN &= \frac{1}{n} \sum_{i=0}^{n-1} (i+1) = \frac{1}{n} (1+2+\cdots+n) = \\ &= \frac{1}{n} * \frac{(1+n) * n}{2} = \frac{1+n}{2} \end{aligned}$$

- ◆ 查找不成功, 数据比较 n 次。



2.2.4 顺序表的插入操作

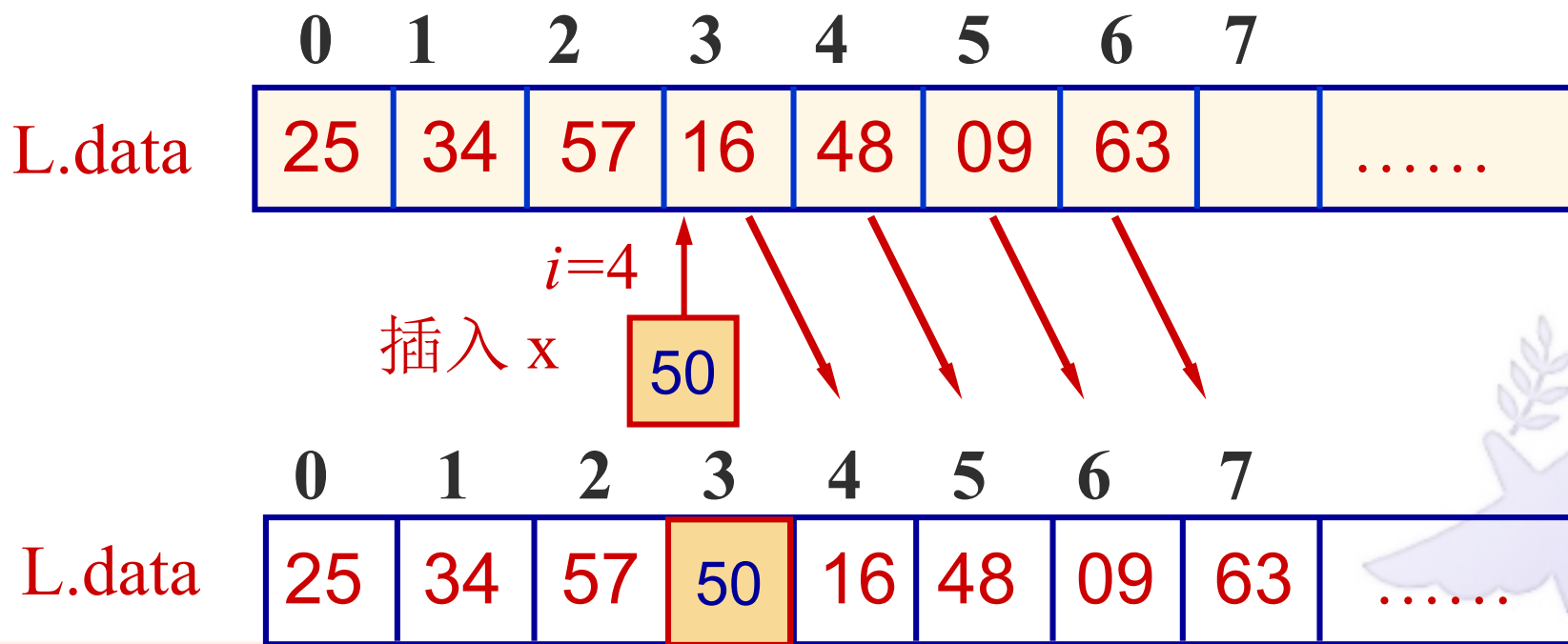
- ◆ **ListInsert(&L, i, e)** // 插入元素
- ◆ 在顺序表L的第 i 个元素之前插入新的元素 e ,
- ◆ 把 e 插入到第 i 个元素的位置
- ◆ i 的合法范围为 $1 \leq i \leq L.n+1$



操作的过程: ListInsert(&L, 4, 50)

◆ 操作步骤:

- 判断插入位置是否合法: $1 \leq i \leq L.n+1$
- 将第n至第i个元素依次后移一位
- 新元素插入第i个位置; 表长度加一

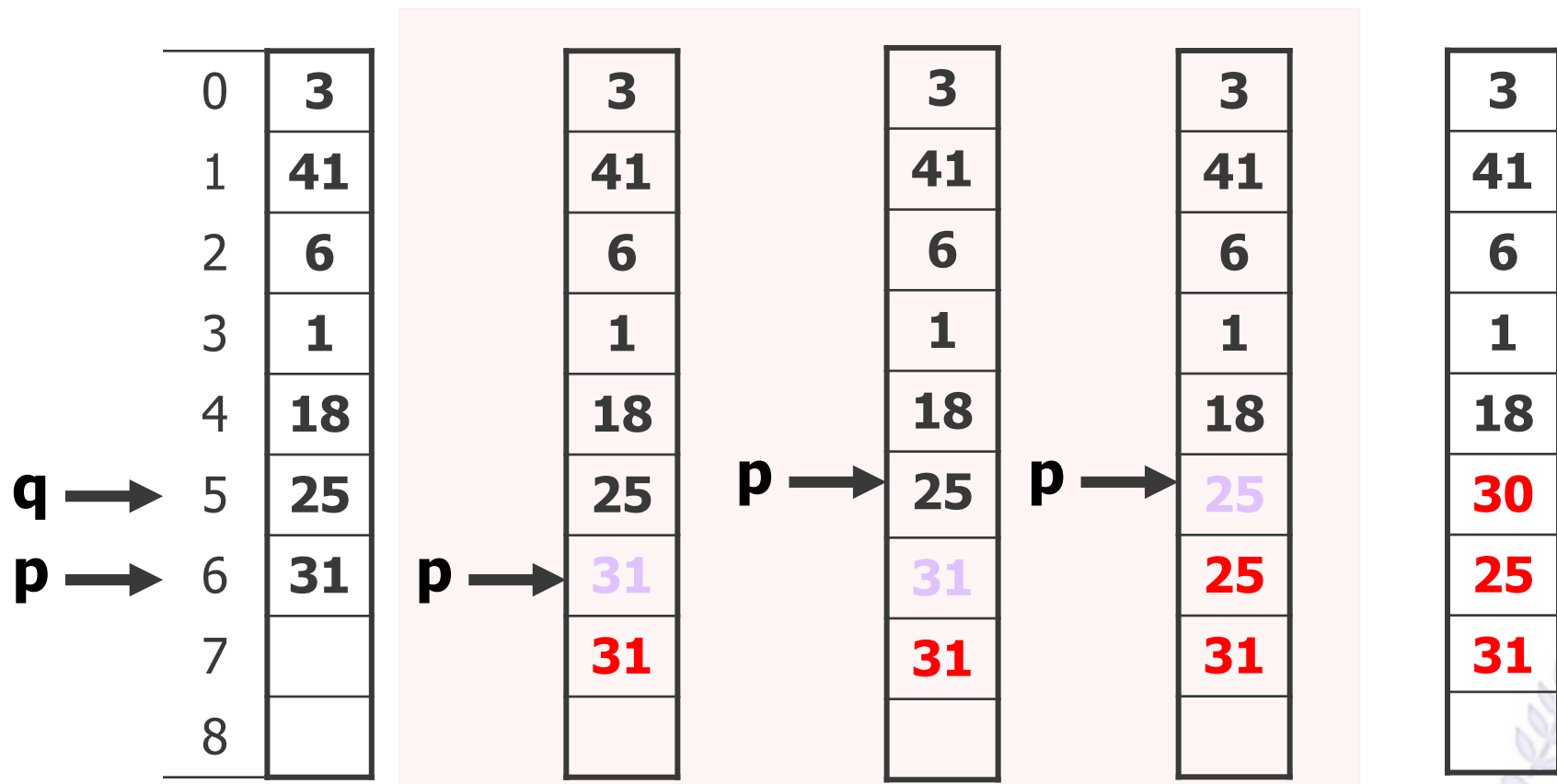




顺序表的插入算法1

```
Status ListInsert ( SeqList& L, DataType x, int i ) {  
    //在表中第 i ( $1 \leq i \leq n+1$ ) 个位置插入新元素 x  
    if ( L.n == L.maxSize ) return false;  
    if ( i < 1 || i > L.n+1 ) return false;  
    for ( int j = L.n-1; j >= i-1; j--) //移动元素  
        L.data[j+1] = L.data[j];  
    L.data[i-1] = x;    //实际插入位置为i-1  
    L.n++; //表长加一  
    return true; //插入成功  
}//ListInsert
```

操作的过程: ListInsert(&L, 6, 30)[严]



$q = \&(L.data[i-1]);$

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

$p--;$

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

$*q=e;$

$p = \&(L.data[L.n-1]);$

$p \geq q;$

插入操作



```
Status ListInsert_Sq(SeqList &L, int i, DataType e) {  
    if (i < 1 || i > L.n+1)           // 步骤1: 位置不合法  
        return ERROR;  
    q = &(L.data[i-1]);               // 步骤2: q 指示插入位置  
  
    for (p = &(L.data[L.n-1]); p >= q; p--)  
        *(p+1) = *p;                 // 步骤3: 元素依次后移  
  
    *q = e;                           // 步骤4: 插入e  
  
    ++L.n;                            // 步骤5: 表长加1  
    return OK;  
} // ListInsert_Sq
```

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：
- ◆ 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.n >= L.maxSize) {  
    // 当前存储空间已满，增加分配  
    newbase = (DataType *)realloc(L.data,  
        (L.maxSize+LISTINCREMENT)*sizeof (DataType));  
    if (!newbase) exit(OVERFLOW);  
    // 存储分配失败  
    L.data = newbase;           // 新基址  
    L.maxSize += LISTINCREMENT; // 增加存储容量  
}
```




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





2.2.5 顺序表的删除操作

删除操作

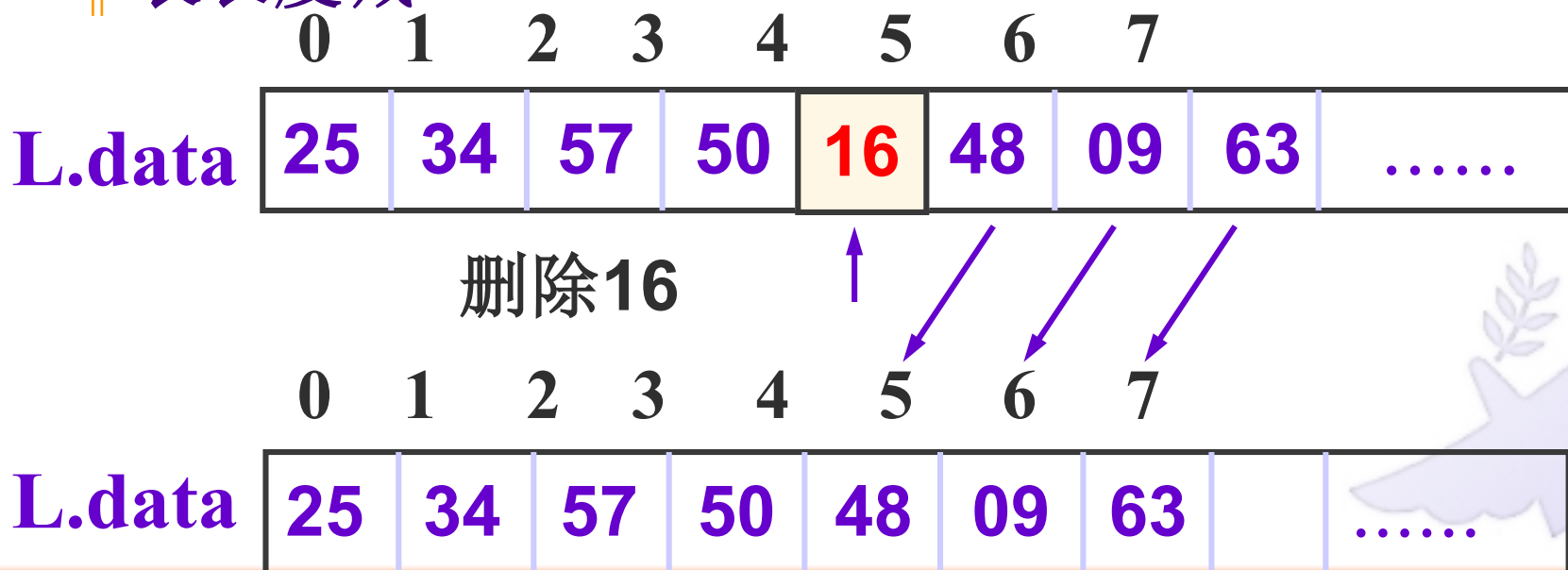
- ◆ `ListDelete(&L, i, &e)` // 删除元素
- ◆ 删除线性表中第*i*个元素，并将删除的元素方在*e*中
- ◆ *i* 的合法范围为 $1 \leq i \leq L.n$



删除操作的过程: ListDelete (&L, 6, &e)

◆ 操作过程:

- 判断参数合法性
- 读出删除元素
- 第i+1个至第n个元素依次前移;
- 表长度减一





```
bool ListDelete ( SeqList& L, int i, DataType& x ) {  
    //在表中删除第 i 个元素，通过 x 返回其值  
    if ( L.n > 0 && i > 0 && i <= L.n ) { //合法性判断  
        x = L.data[i-1]; //读出删除元素  
        for (int j = i; j < L.n; j++) //元素前移  
            L.data[j-1] = L.data[j];  
        L.n--; //表长度减一  
        return true; //删除成功  
    } // if  
    else return false; //删除失败  
} //ListDelete
```


$$*(p-1) = *p;$$

Status ListDelete_Sq

(SeqList &L, int i, DataType &e) {

//步骤1: 位置是否合法

if ((i < 1) || (i > L.n)) return ERROR;

p = &(L.data[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.data+L.n-1; // 表尾的位置

for (++p; p <= tail; ++p) *(p-1) = *p;

--L.n; // 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：
- ◆ 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： $O(n)$





小结：顺序表的优缺点

◆ 优点

- 🔑 不需要附加空间
- 🔑 随机存取任一个元素（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的内存空间
- 🔑 更新操作代价大





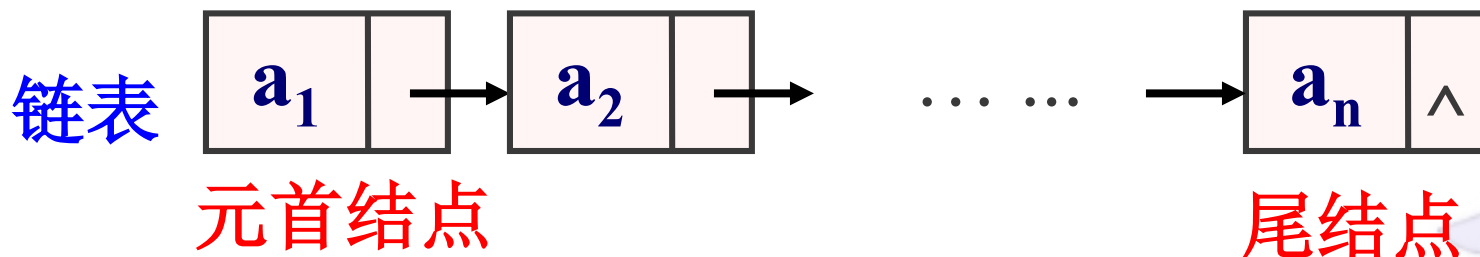
2.3 线性表的链式表示和实现

- ◆ 2.3.1 单链表的定义和特点
- ◆ 2.3.2 单链表的结构定义
- ◆ 2.3.3 单链表中操作的实现
- ◆ 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



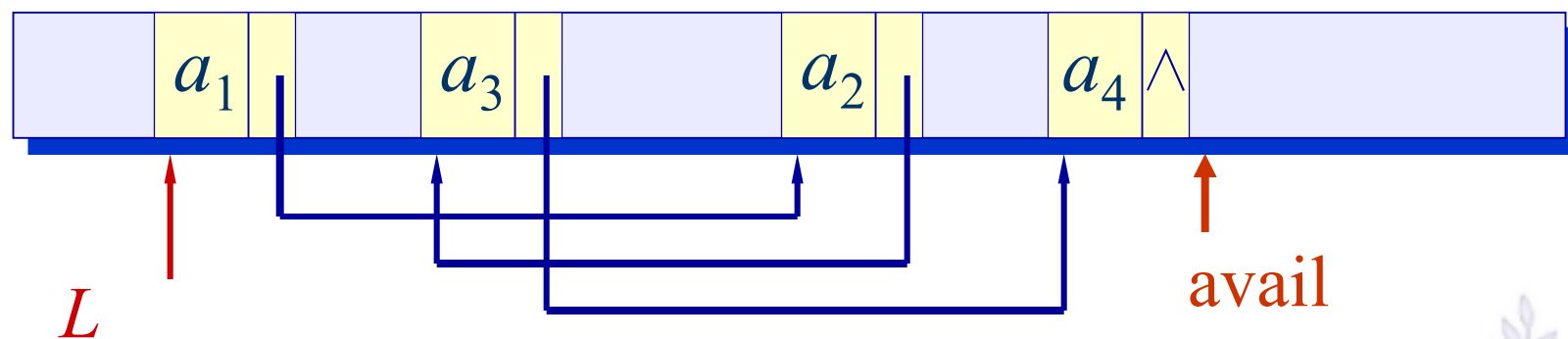
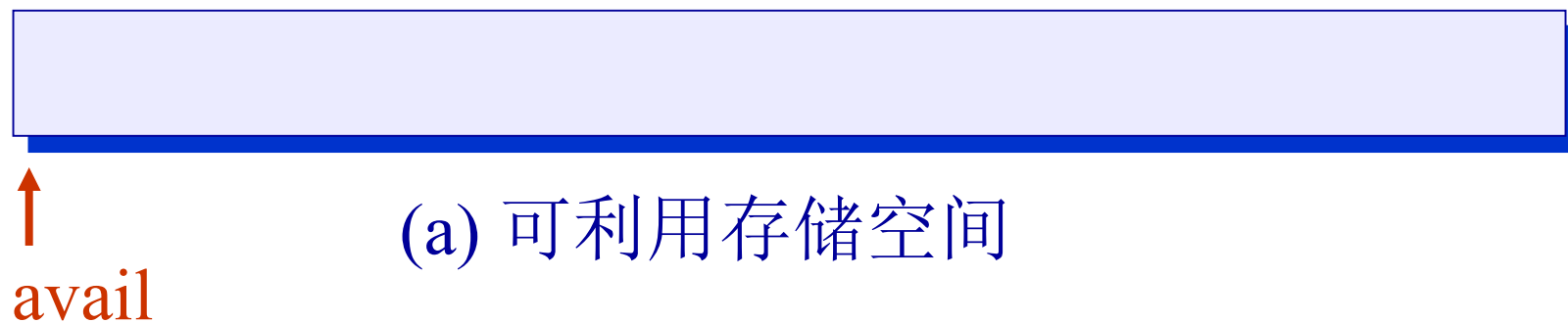
2.3.1 单链表的定义和特点

- ◆ 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)
- ◆ 以“结点的序列”表示线性表——称作链表
- ◆ 链表中第一个元素结点称为元首结点，最后一个元素称为尾结点。首元结点不是头结点。



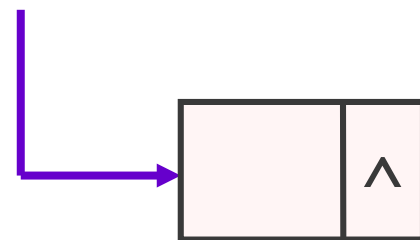


单链表的存储映像

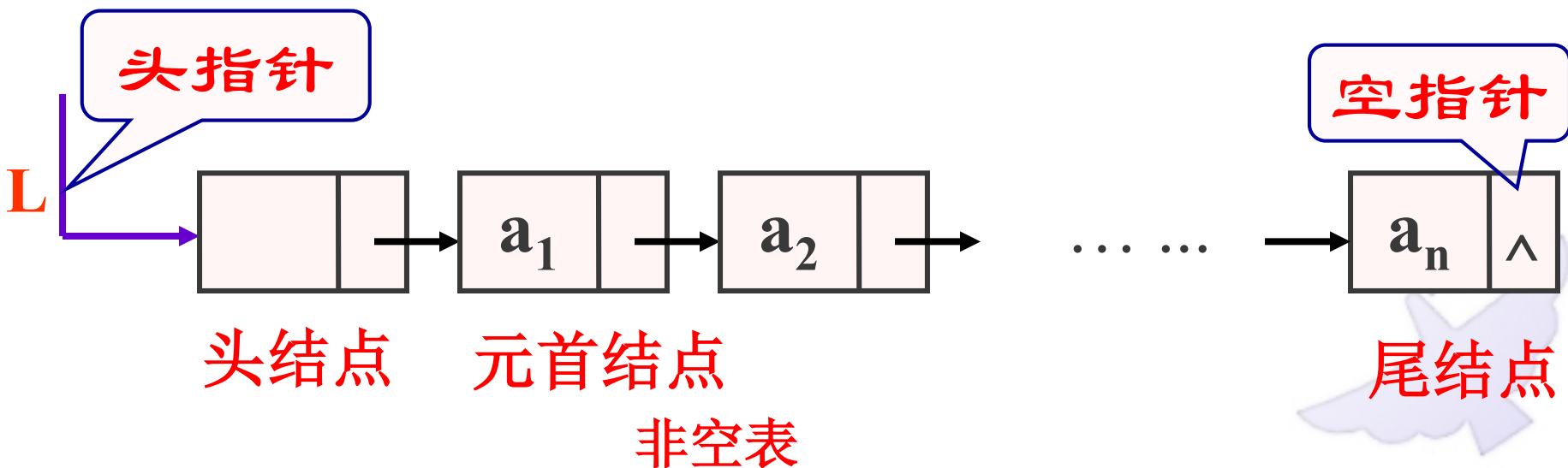


带头结点的单链表

- ◆ **头结点**位于表的最前端，本身不带数据，仅标志表头。
- ◆ 设置头结点的目的是
 - ✎ 统一空表与非空表的操作
 - ✎ 简化链表操作的实现。

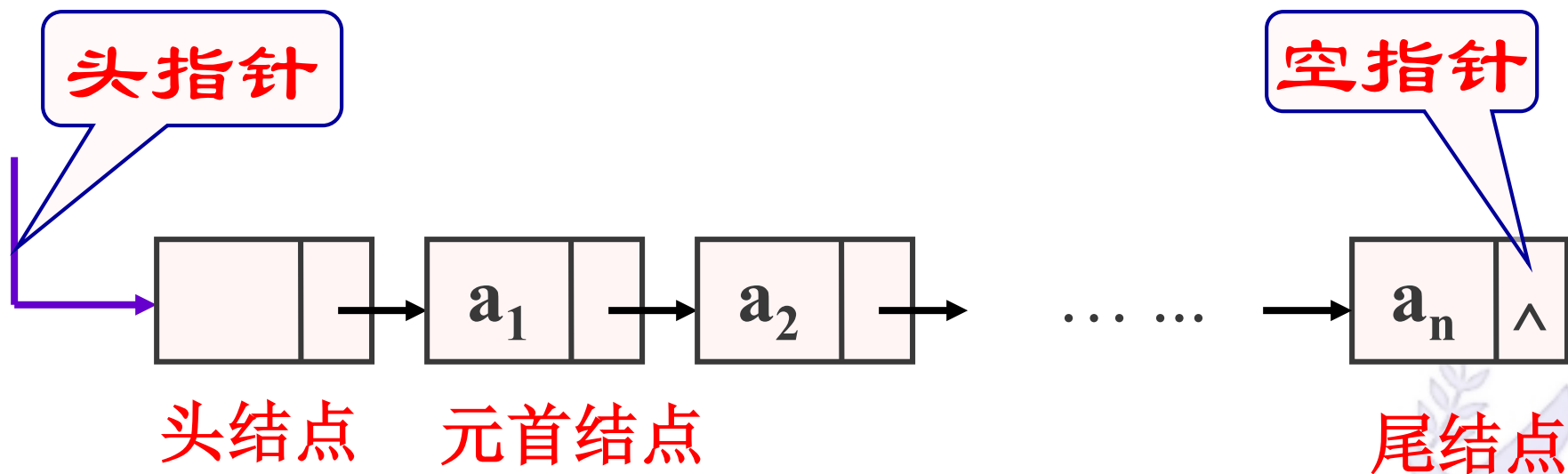


空表



2.3.1 单链表的定义和特点

- ◆ 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。



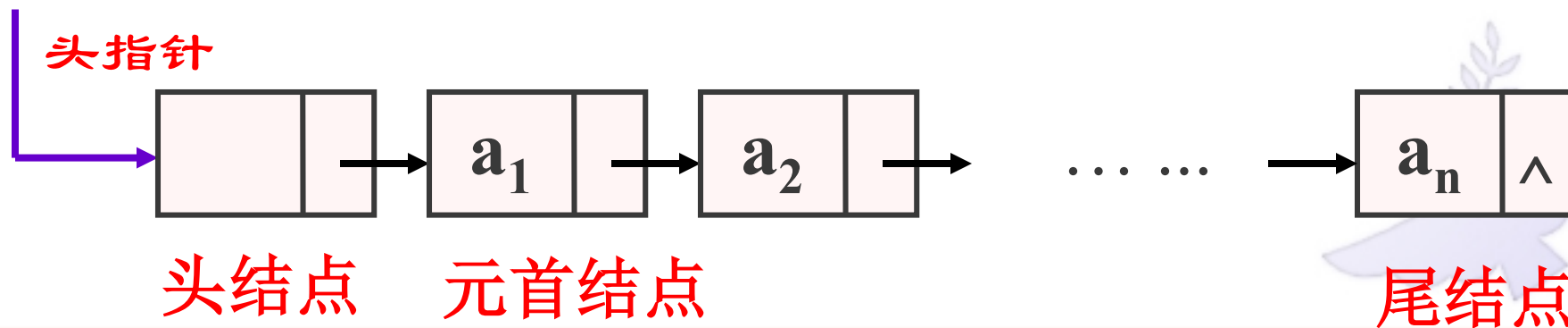
2.3.2 线性链表的定义

data

link

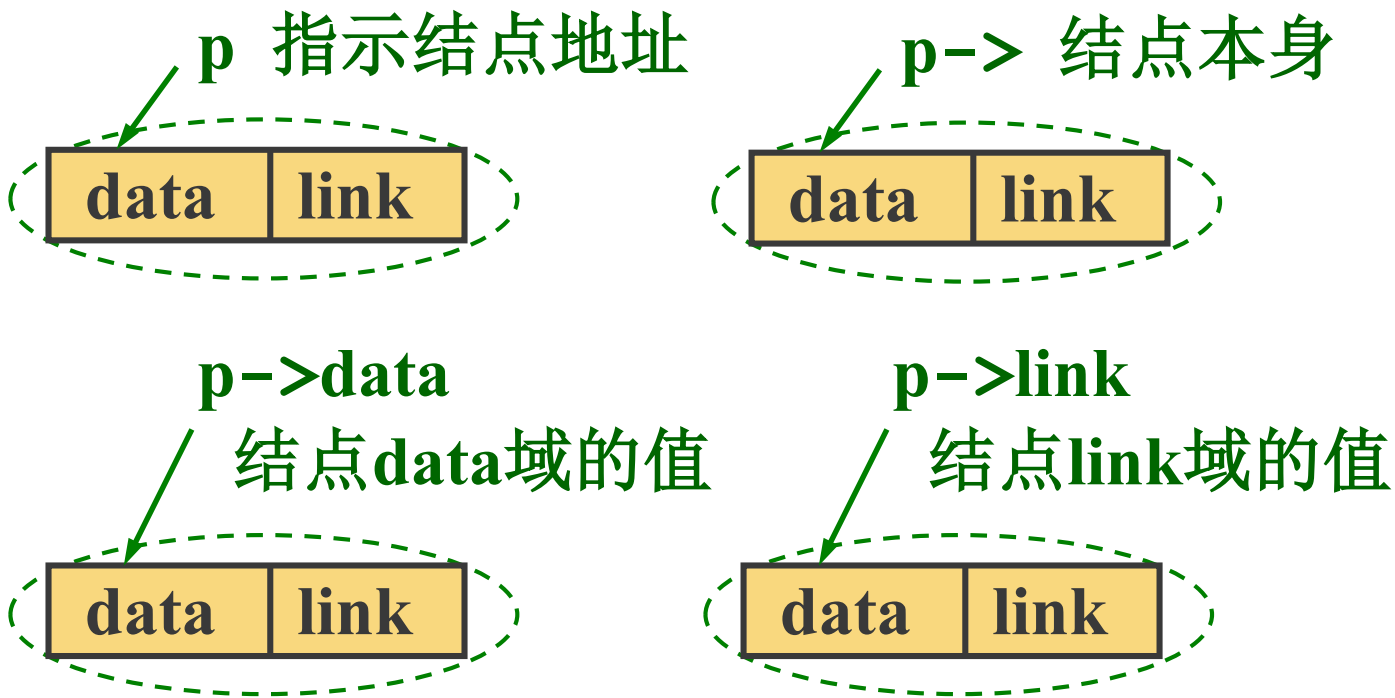
```
typedef struct node {  
    DataType    data; // 数据域  
    struct node *link; // 指针域  
} LinkNode, *LinkList;
```

LinkList L; // L 为单链表的头指针



有关指针的一些概念

LinkNode *p;



- ◆ 在链表中，如果没有定义重载“++”函数，不能使用 **p++** 这样的语句进到逻辑上的下一个结点。
- ◆ 一般用 **p = p->link** 进到下一结点。



2.2.3 单链表中操作的实现

GetElem(L, i, e) // 取第*i*个数据元素

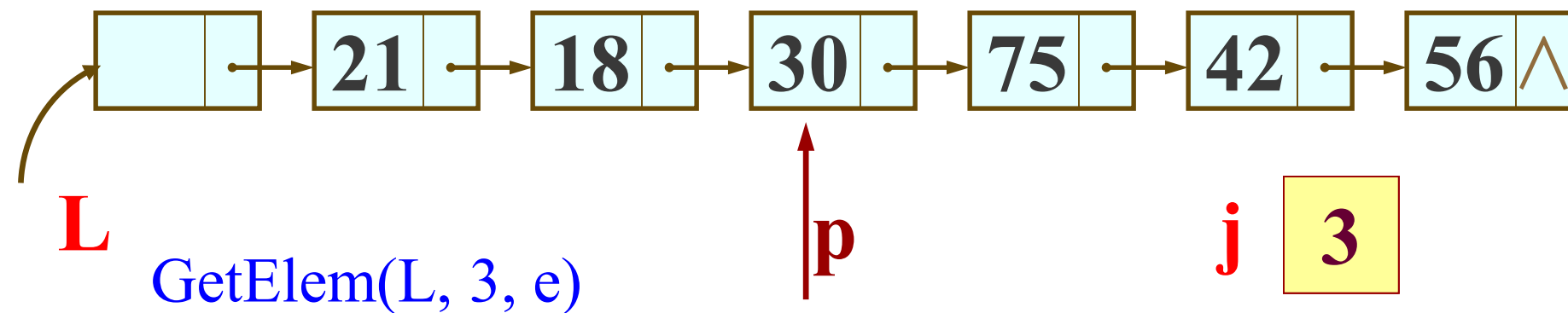
ListInsert(&L, i, e) // 插入数据元素

ListDelete(&L, i, e) // 删除数据元素

ClearList(&L) // 重置线性表为空表



线性表的操作 $\text{GetElem}(L, i, \&e)$



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 $p=\text{null}$ ；



线性表的操作 $\text{GetElem}(L, i, \&e)$

◆ 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**



```

Status GetElem_L(LinkList L, int i, DataType &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→link; j = 1; // 步骤1: 初始化指针和计数器
    //步骤2: 顺指针向后查找，直到p指向第i个元素
    //或 p为空
    while (p!=NULL && j<i) { p = p→link; ++j; }
    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： $O(\text{ListLength}(L))$

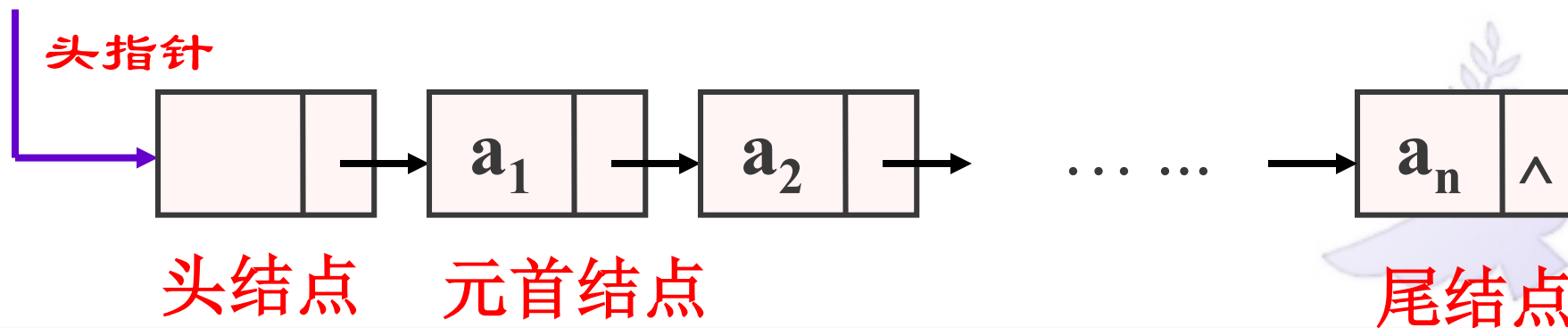
2.3.2 线性链表的定义

data

link

```
typedef struct node {  
    DataType    data; // 数据域  
    struct node *link; // 指针域  
} LinkNode, *LinkList;
```

LinkList L; // L 为单链表的头指针





在单链表中按值查找

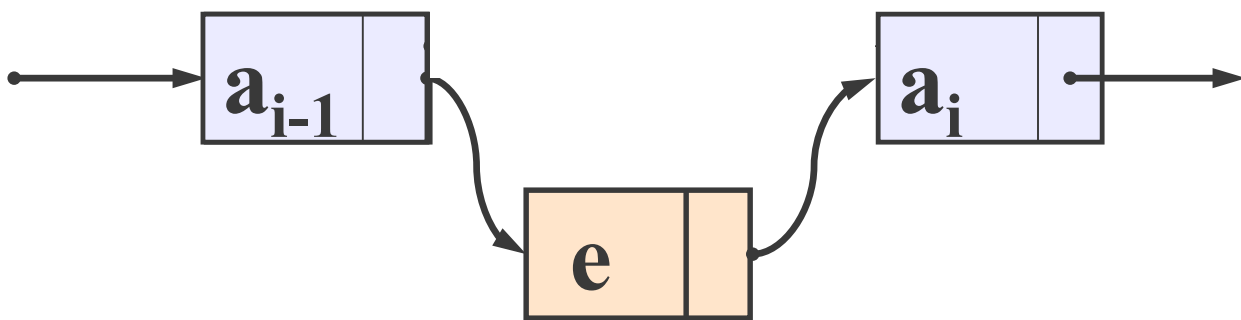
```
LinkNode *Search ( LinkList L, DataType x ) {  
    //在链表中从头搜索其数据值为 x 的结点  
    LinkNode * p = L->link;    //p为检测指针  
    while ( p != NULL && p->data != x )  
        p = p->link;  
    return p;  
}
```

注意，while循环条件**p != NULL**和**p->data != x**不能交换!



线性表的操作 $\text{ListInsert}(\&L, i, e)$

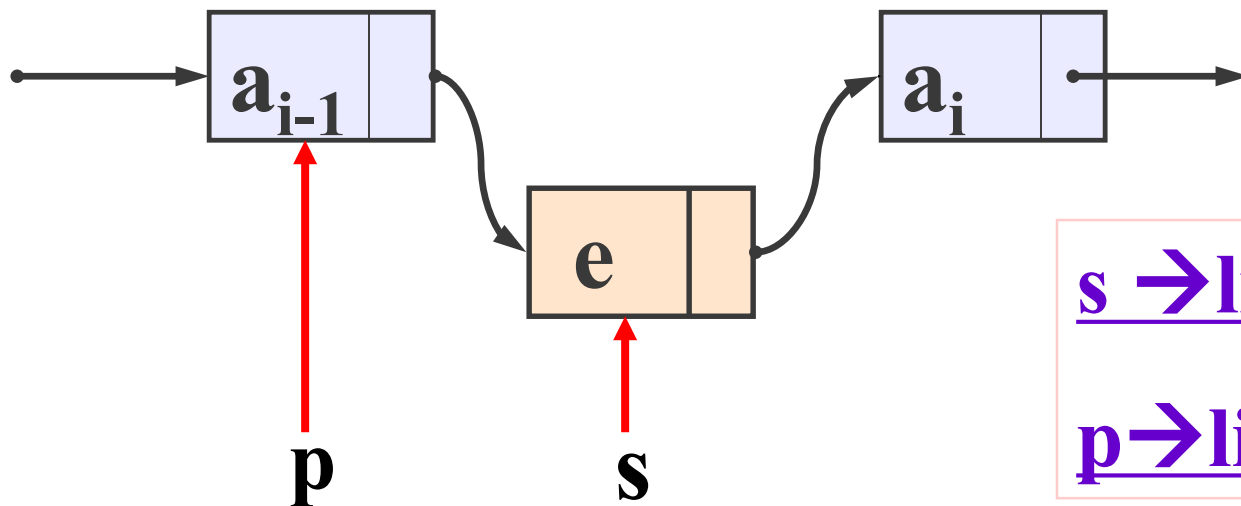
- ◆ 在线性表第 i 个位置插入元素 e
- ◆ 即在第 i 个结点之前插入元素 e
- ◆ 结果：有序对 $\langle a_{i-1}, a_i \rangle$ 改变为 $\langle a_{i-1}, e \rangle$ 和 $\langle e, a_i \rangle$



线性表的操作 $\text{ListInsert}(\&L, i, e)$

- ◆ 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



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

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



线性表操作ListInsert(&L, i, e)基本过程

1. p 指向头结点，初始化计数器 $j=0$;
2. 查找第 $i-1$ 个节点：顺指针向后查找，直到 $j=i-1$ 或 p 为空;
3. 如果找不到第 $i-1$ 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;




```
Status ListInsert_L(LinkList L, int i, DataType 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!=NULL && j < i-1)//步骤2: 寻找第i-1个结点  
    { p = p→link; ++j; }
```

```
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 →link = p→link; p→link = s;
```

```
return OK;
```

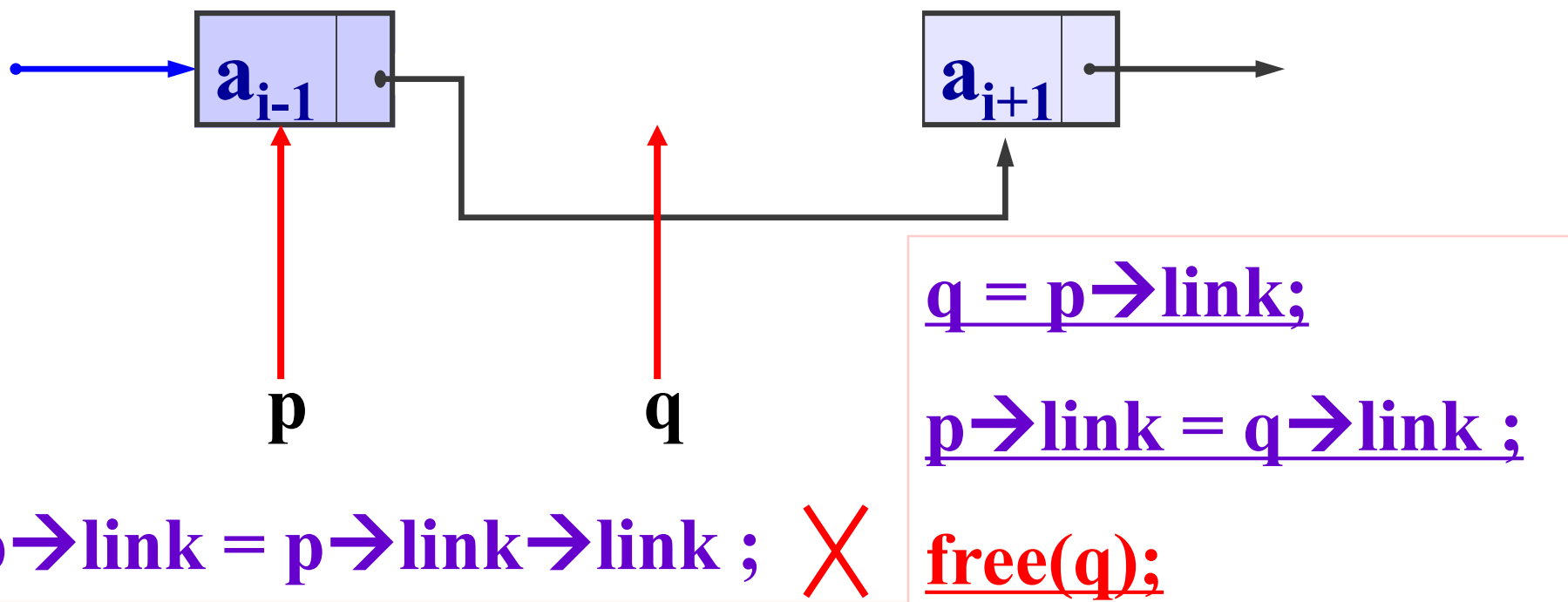
算法的时间复杂度为:

```
} // LinstInsert_L
```

$O(\text{ListLength}(L))$

线性表的操作 **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。





线性表的操作 **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. 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, DataType &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:初始化指针和计数器  
  
    //步骤2: 寻找第i个结点, 并令p指向其前趋  
    while (p → link && j < i-1) { p = p→link; ++j; }  
  
    if (!(p→link) || j > i-1)  
        return ERROR; // 步骤3: 位置不合理  
  
    q = p→link; p→link = q→link; //步骤4:删除节点  
    e = q→data; free(q); //步骤5: 释放结点  
  
    return OK;  
} // ListDelete_L
```

算法的时间复杂度为: $O(\text{ListLength}(L))$



线性表的操作 **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→link;//步骤1:初始化指针  
    while (p != NULL) {//步骤2: 依次释放各个节点  
        q=p→link;  free(p);      p=q;  
    }  
    L→link = NULL; //头节点指向空  
} // ClearList
```

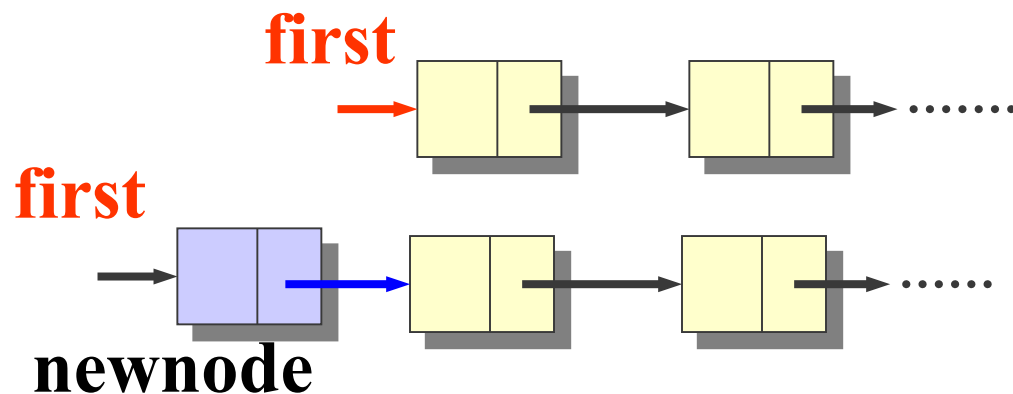
算法时间复杂度: $O(\text{ListLength}(L))$

不带头结点的单链表操作：插入操作

- ◆ **bool Insert (LinkList& first, int i, dataType x)**
- ◆ 第一种情况：在第 1 个结点前插入
- ◆ 条件： $i == 1$

```
newnode->link = first ;
```

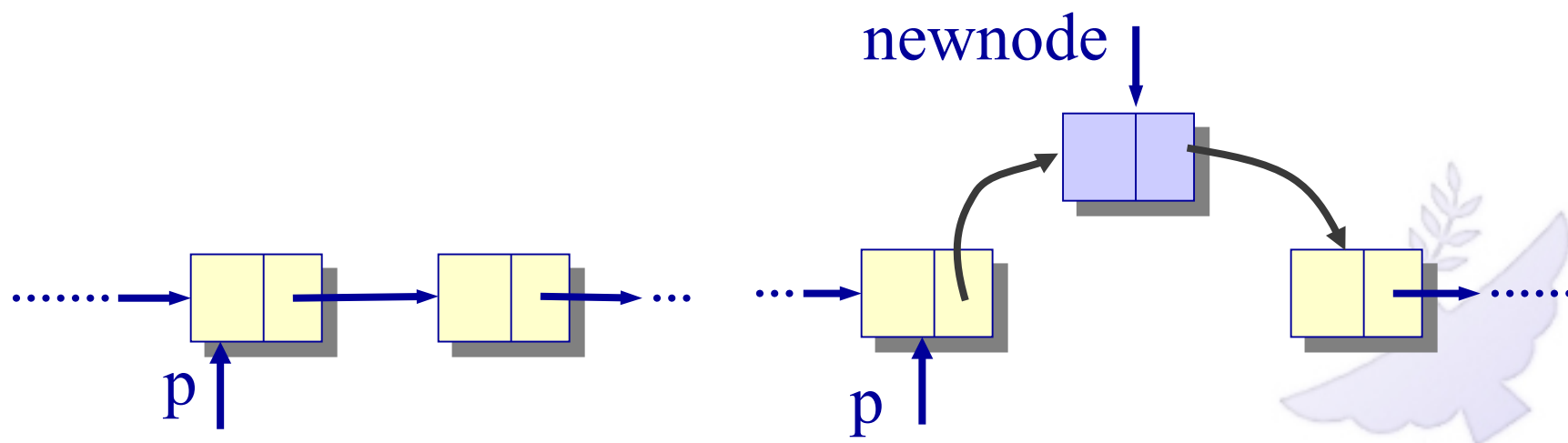
```
first = newnode;           //修改头指针
```



不带头结点的单链表操作：插入操作

- ◆ 第二种情况：在链表中间插入
- ◆ 先定位指针 p 到插入位置，再将新结点插在其后

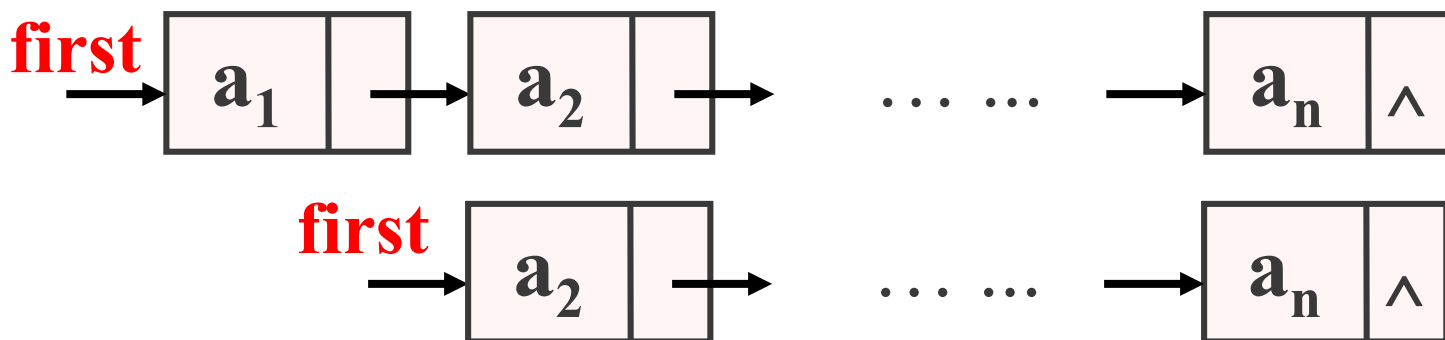
```
newnode->link = p->link;  
p->link = newnode;
```



不带头结点的单链表操作：删除操作

- ◆ **bool Remove (LinkList& first, int i, dataType& x)**
- ◆ //在链表中删除第 i 个结点
 - 第一种情况：删除表中第一个元素
 - 第二种情况：删除表中其它元素，同有头结点情况

```
if ( i == 1 ) { //删除第一个元素  
    q = first; first = first->link; }
```





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

下面用基本操作实现：

CreateList_L(LinkList &L, int n) // 创建线性表



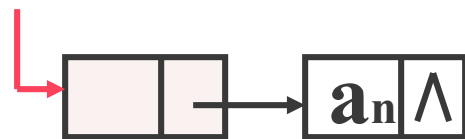
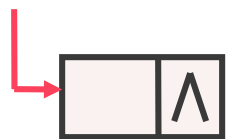
CreateList_L(LinkList &L, int n)

- ◆ 逆序输入 n 个数据元素，建立带头结点的单链表。

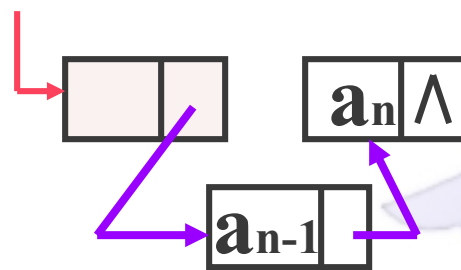
生成链表的过程是一个结点“逐个插入”的过程。

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, 1, e);
3. 依次类推，直至输入 a_1 为止

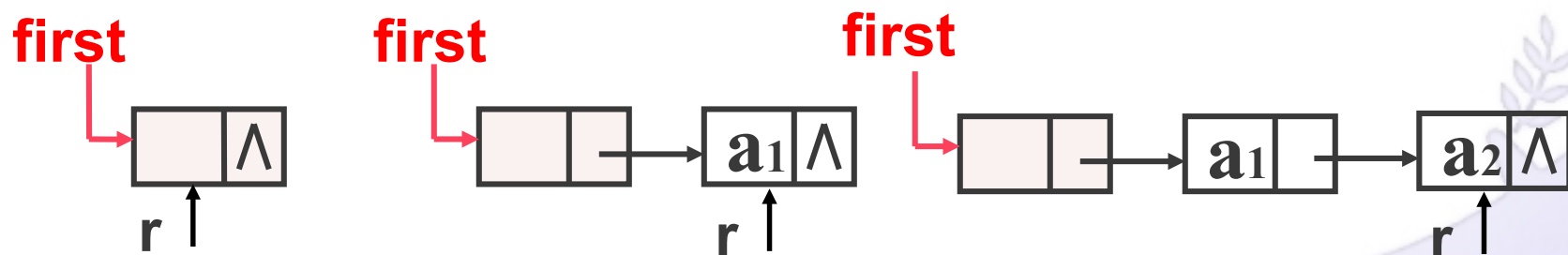


前插法



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
- ◆ 基本思想：
 - 🔧 每次将新结点加在插到链表的表尾；
 - 🔧 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；
 - 🔧 尾指针 r 初始时置为指向表头结点地址。



后插法



单链表练习

◆ 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

◆ 改变链表的设置

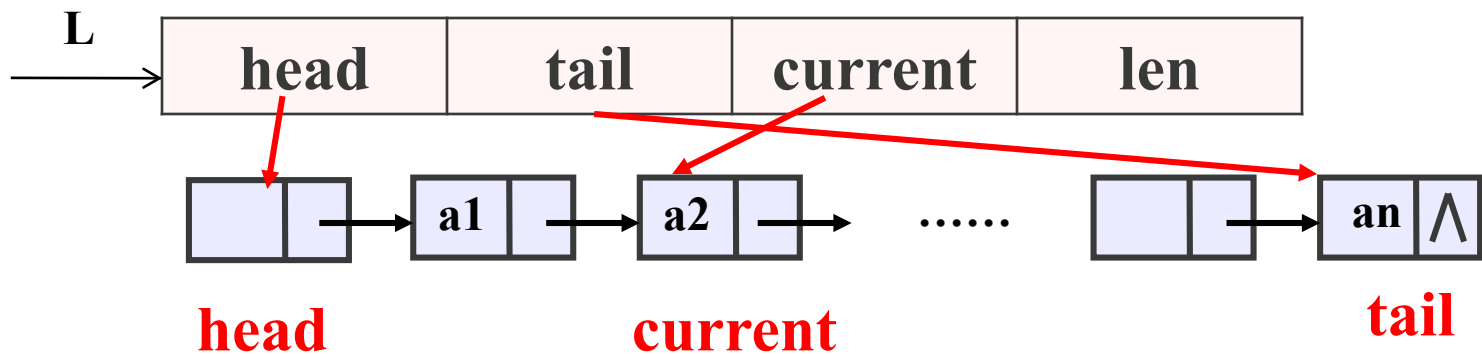
- 增加“表长”、“表尾指针”和“当前位置的指针”三个数据域;



单链表练习

```
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;
```

LinkList L;





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。
```

$O(n)$



head

tail

current

len

增加指针后的链表的基本操作

{引用型操作}

复杂度

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)

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

O(1)



head

tail

current

len

增加指针后的链表的基本操作

{引用型操作}

复杂度

```
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, DataType e,  
    Status (*compare)(DataType, DataType));  
    //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
```

O(n)

```
Status ListTraverse(LinkList L, Status(*visit)() );  
    // 依次对L的每个元素调用函数visit()
```

O(n)



head

tail

current

len

增加指针后的链表的基本操作

{加工型操作}

复杂度

Status ClearList (LinkList &L);

// 重置 L 为空表

O(n)

Status SetCurElem(LinkList &L, DataType 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, DataType e);

// 将元素 e 插入在当前指针之后

O(1)

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

// 删除当前指针之后的结点

O(1)



Status InsAfter(LinkList& L, DataType 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,当前指针指向新结点，
//并返回OK; // 否则返回ERROR。

if (! L.current) return ERROR;//合法性判断

if (! MakeNode(s, e)) return ERROR;//创建新节点

s→link = L.current→link;

L.current→link = s;

if (L.tail == L.current) L.tail = s;//是否修改尾指针

L.current = s;// 修改当前指针

L.len++;//修改表长度

return OK;

} // InsAfter

head	tail	current	len
------	------	---------	-----

2.3.5 静态链表

◆ 用数组实现的链式结构，称为静态链表

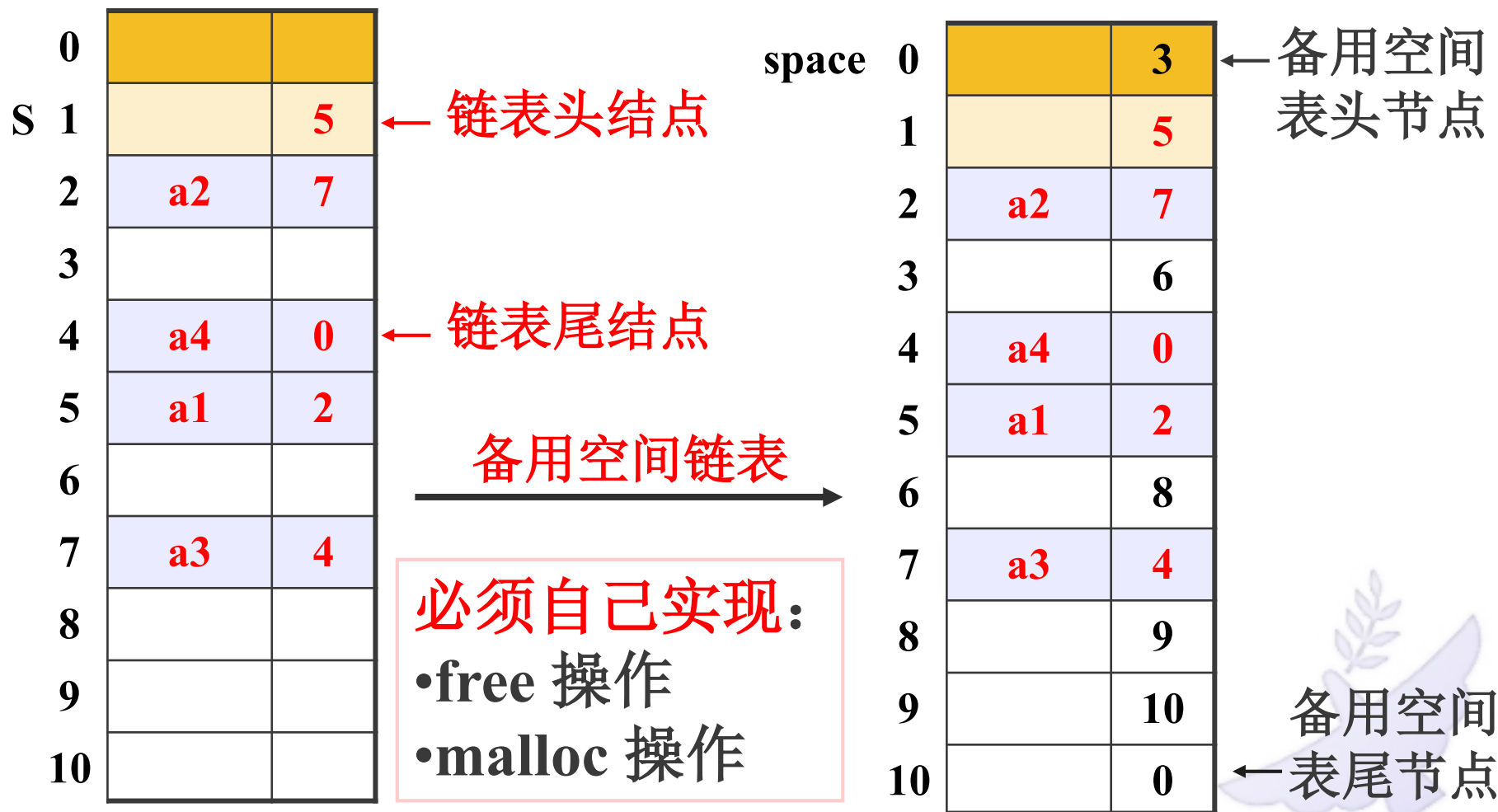
0			
1		5	← 头结点
2	a2	7	
3			
4	a4	0	← 尾结点
5	a1	2	
6			
7	a3	4	
8			
9			
10			

```
#define MAXSIZE 1000
// 链表的最大长度
typedef struct{
    DataType 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
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

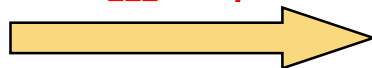
ListInsert_SL(int S, int 5, WAN)

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

插入
WAN

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	WAN	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_SL0;**

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

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

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



静态链表的插入算法

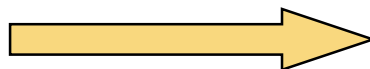
```
Status ListInsert_SL(int S, int i,DataType e)
{ //S静态链表头结点下标, 在第i个元素前插入e
    k=S; j=0; //寻找第i-1个元素结点
    while (k!=0 && j<i-1){k=space[k].cur ; ++j;}
    if(k==0 || j>i-1) return ERROR;
    m=Malloc_SL(); //分配新结点
    if (m == 0) return ERROR;
    space[m].data=e;
    space[m].cur= space[k].cur ; //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	WAN	0
	8		9
	9		10
	10		0

删除
SUN

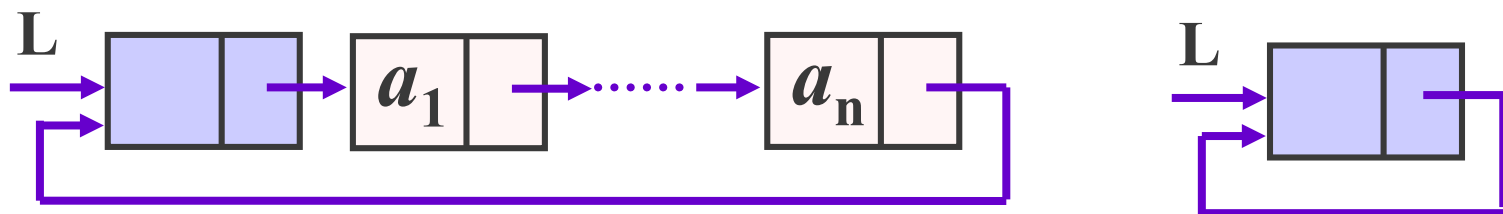


space	0		4
S	1		2
	2	ZHAO	3
	3	QIAN	5
	4	SUN	8
	5	LI	6
	6	ZHOU	7
	7	WAN	0
	8		9
	9		10
	10		0

2.3.6 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”：

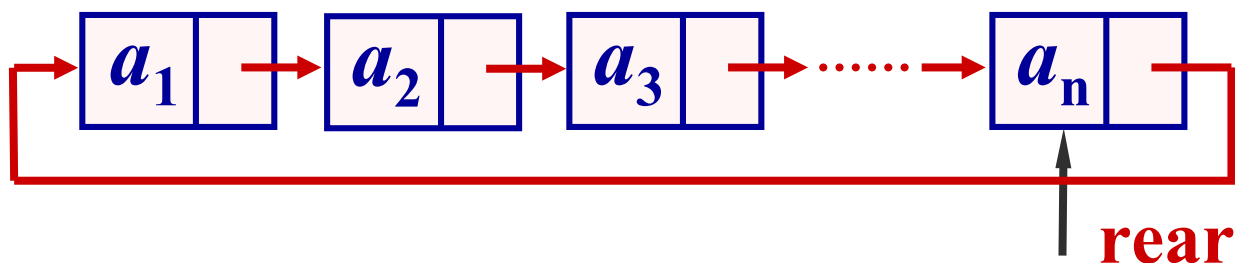
- ✓ 单链表中是“后继是否为空”
- ✓ 单向循环链表中是“后继是否为头结点”。



◆ 带尾指针的循环链表

◆ 如果插入与删除仅在链表的两端发生，可采用带表尾指针的循环链表结构。

- 🔊 在表尾可直接插入新结点，时间复杂度 $O(1)$;
- 🔊 在表尾删除时要找前趋，时间复杂度 $O(n)$;
- 🔊 在表头插入相当于在表尾插入;
- 🔊 在表头可直接删除，时间复杂度 $O(1)$ 。



2) 双向链表

llink

data

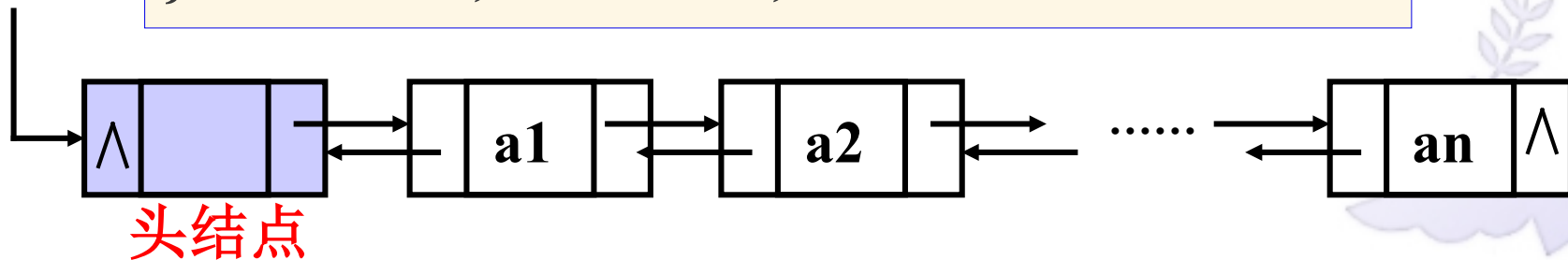
rlink

前趋方向



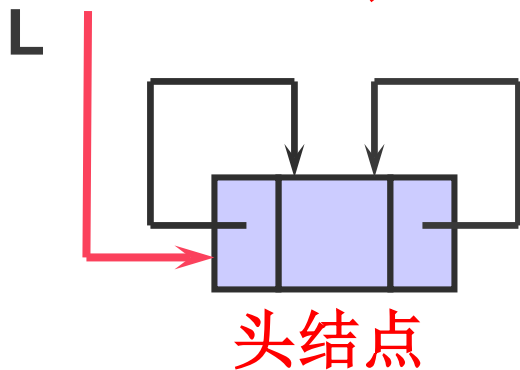
后继方向

```
typedef struct DuLNode {  
    DataType    data; // 数据域  
    struct DuLNode *llink;  
                // 指向前驱的指针域  
    struct DuLNode *rlink;  
                // 指向后继的指针域  
    int freq;    // 访问计数  
} DblNode, *DblList;
```

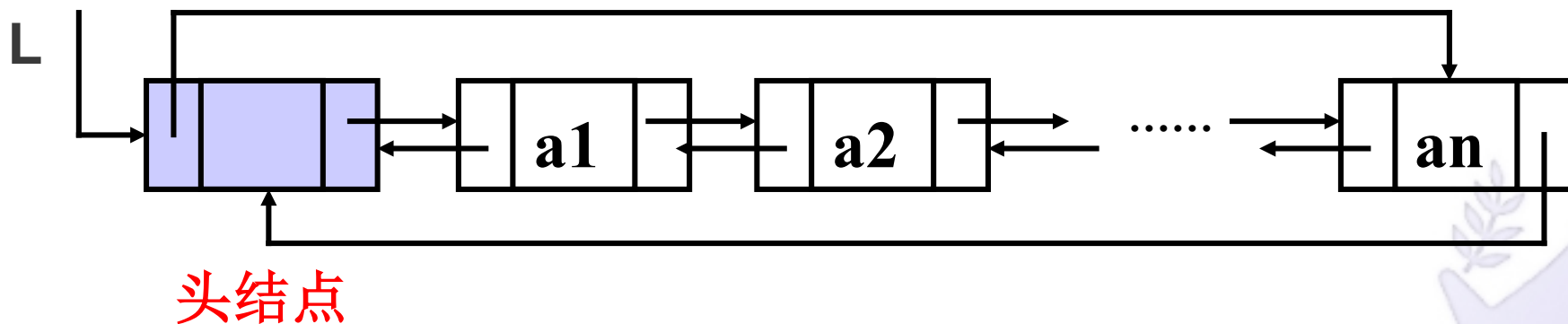


3) 双向循环链表

空表



前趋方向 ← → 后继方向



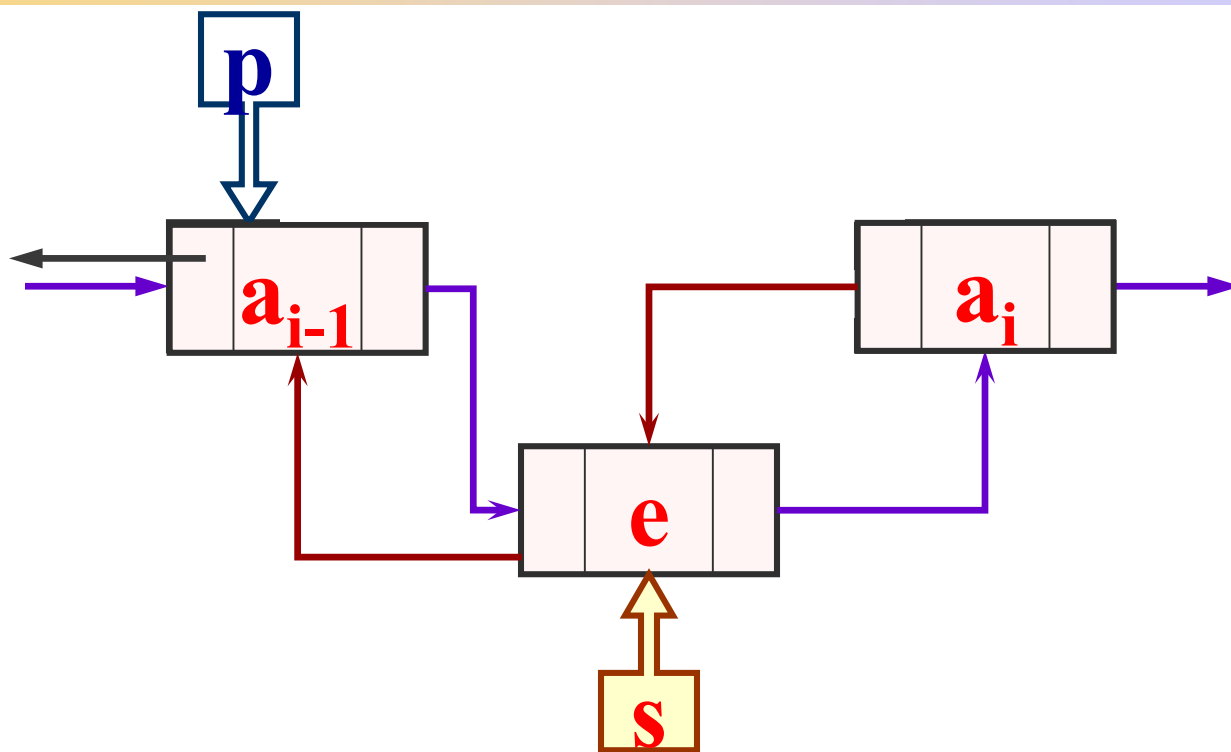


双向链表的操作

- ◆ “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。



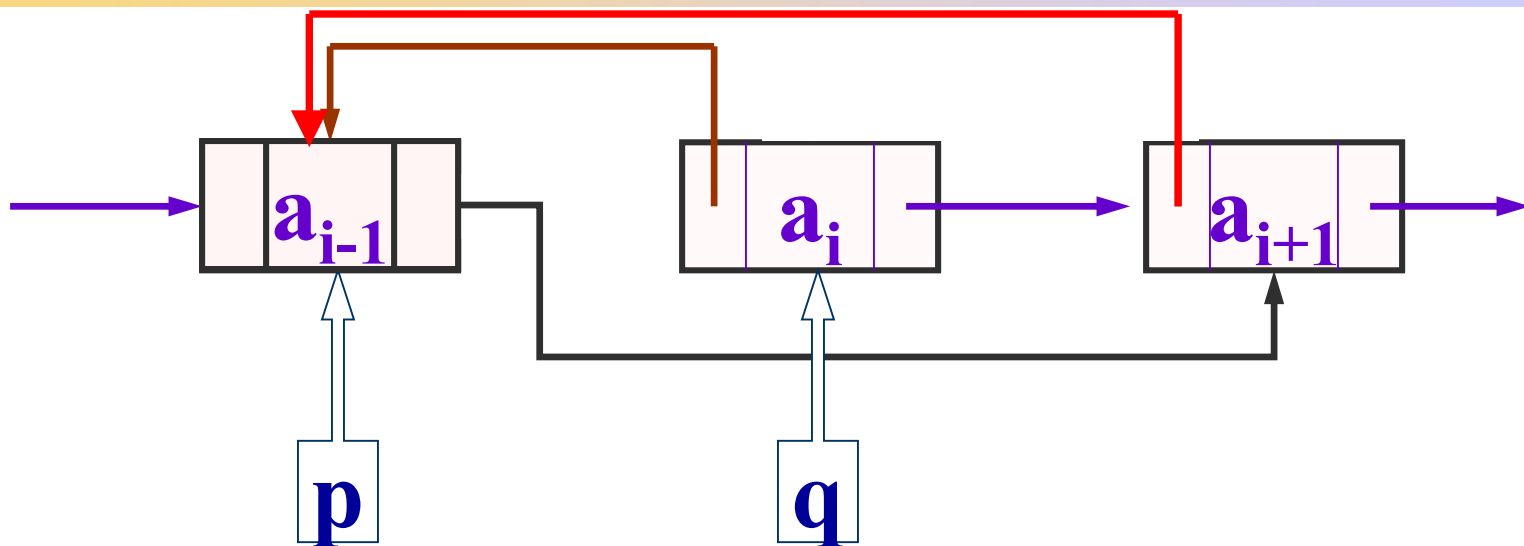
双向链表的插入操作



$s \rightarrow rLink = p \rightarrow rLink; \quad p \rightarrow rLink = s;$

$s \rightarrow rLink \rightarrow lLink = s; \quad s \rightarrow lLink = p;$

双向链表的删除操作



$q = p \rightarrow rLink$

$p \rightarrow rLink = q \rightarrow rLink;$

$p \rightarrow rLink \rightarrow lLink = p;$

$free(q);$





2.4 一元多项式的表示



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：

$$\text{P} = (p_0, p_1, \dots, p_n)$$

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

◆ 称为一元稀疏多项式

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

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:

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

◆ 例如:

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

¶ 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:

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

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)

◆ MuliplyPolyn(&Pa, &Pb)





一元多项式的实现

```
typedef LinkList 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

```
typedef struct { // 项的表示
```

```
    float coef; // 系数
```

```
    int expn; // 指数
```

```
} term, DataType;
```

$$P_1(x) = 7x^3 - 2x^{12} - 8x^{999}$$

$$P_2(x) = 4x + 5x^{12} - 34x^{39}$$





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
- 📏 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



线性表存储密度

- ‖ **n**表示线性表中当前元素的数目
- ‖ **P**表示指针存储单元大小
- ‖ **E**表示数据元素存储单元大小
- ‖ **D**表示可在数组中存储的线性表元素的最大数目

◆ 空间需求

- ‖ 顺序表的空间需求为 $D * E$
- ‖ 链表的空间需求为 $n(P + E)$

◆ **n**的临界值：使得 $DE = n(P + E)$

- ‖ **n**大于临界值，顺序表的空间效率就更高
- ‖ 如果 $P = E$ ，则临界值为 $D/2$





线性表习题

- ◆ 3、若线性表最常用的操作是存取第 i 个元素及其前趋和后继元素的值，为节省时间应采用的存储方式是（ ）。
 - **A** 顺序表 **B** 单链表 **C** 双向链表 **D** 单向循环链表
- ◆ 4、在 p 所指向的结点后插入由 q 指向的新结点的语句序列是
 - **A** $p \rightarrow \text{link} = q;$ $q \rightarrow \text{link} = p \rightarrow \text{link};$
 - **B** $q = p \rightarrow \text{link};$ $p \rightarrow \text{link} = q;$
 - **C** $q \rightarrow \text{link} = p \rightarrow \text{link};$ $p \rightarrow \text{link} = q;$
 - **D** $p = p \rightarrow \text{link};$ $q \rightarrow \text{link} = p;$

答案：A C





线性表习题

- ◆ 5、若长度为 n 的线性表采用顺序存储结构，在其
- ◆ 第 i ($1 \leq i \leq n+1$) 个位置之前插入一个新元素的算法的时间复杂度为 ()。
 - ‖ A) $O(n/2)$ B) $O(n)$ C) $O(n^2)$ D) $O(\log_2 n)$
- ◆ 6、若某线性表最常用的操作是在最后一个结点之后插入一个结点或删除最后一个结点，则采用存储结构算法的时间效率最高的是 ()。
 - ‖ A) 单链表 B) 给出表尾指针的单循环链表
 - ‖ C) 双向链表 D) 给出表尾指针双向循环链表

答案： B、 D

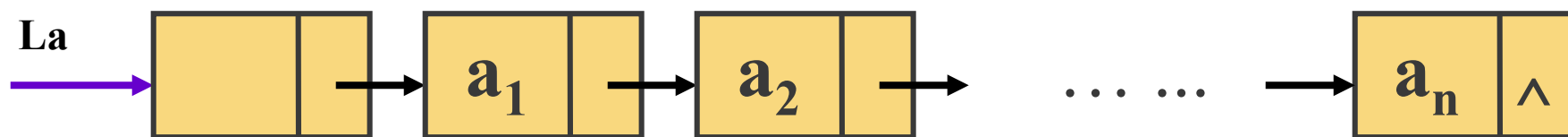


线性表习题-算法题

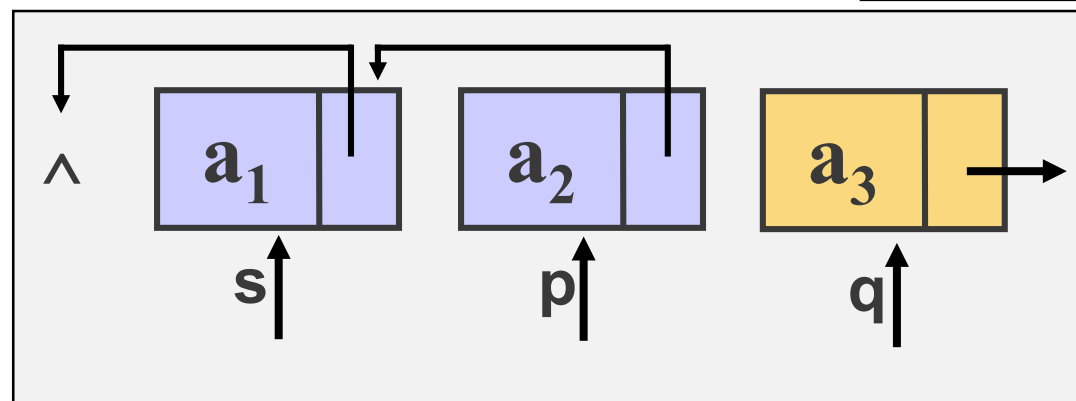
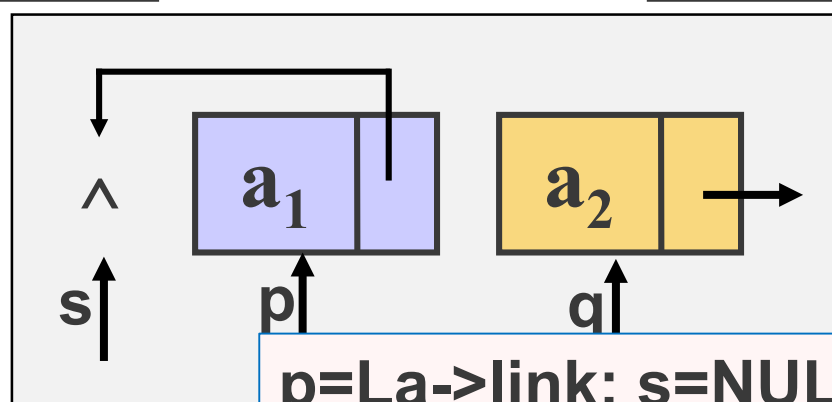
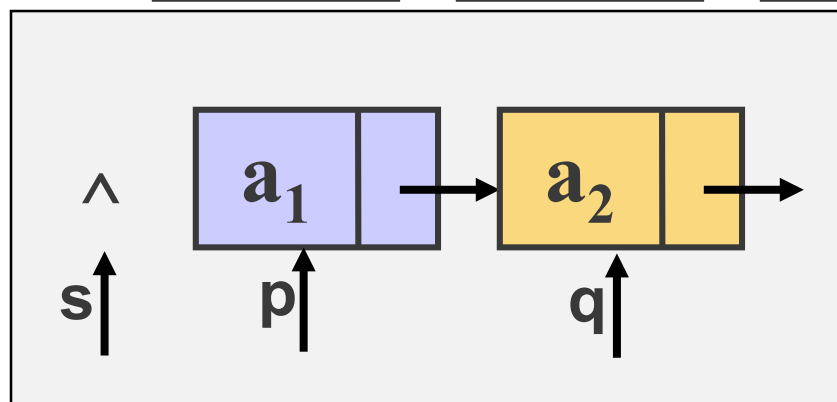
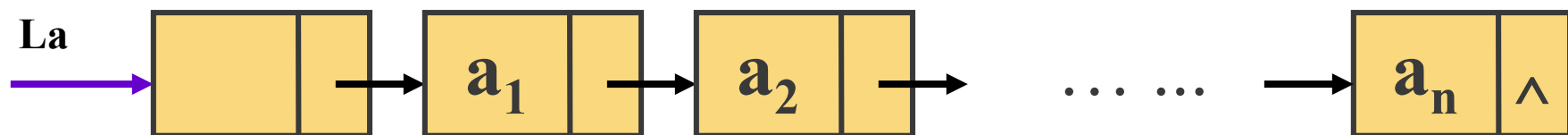
◆ 1、给定一个带头节点的单链表，请将该链表置逆。

思路1：原地置逆

思路2：逆序插入新表中

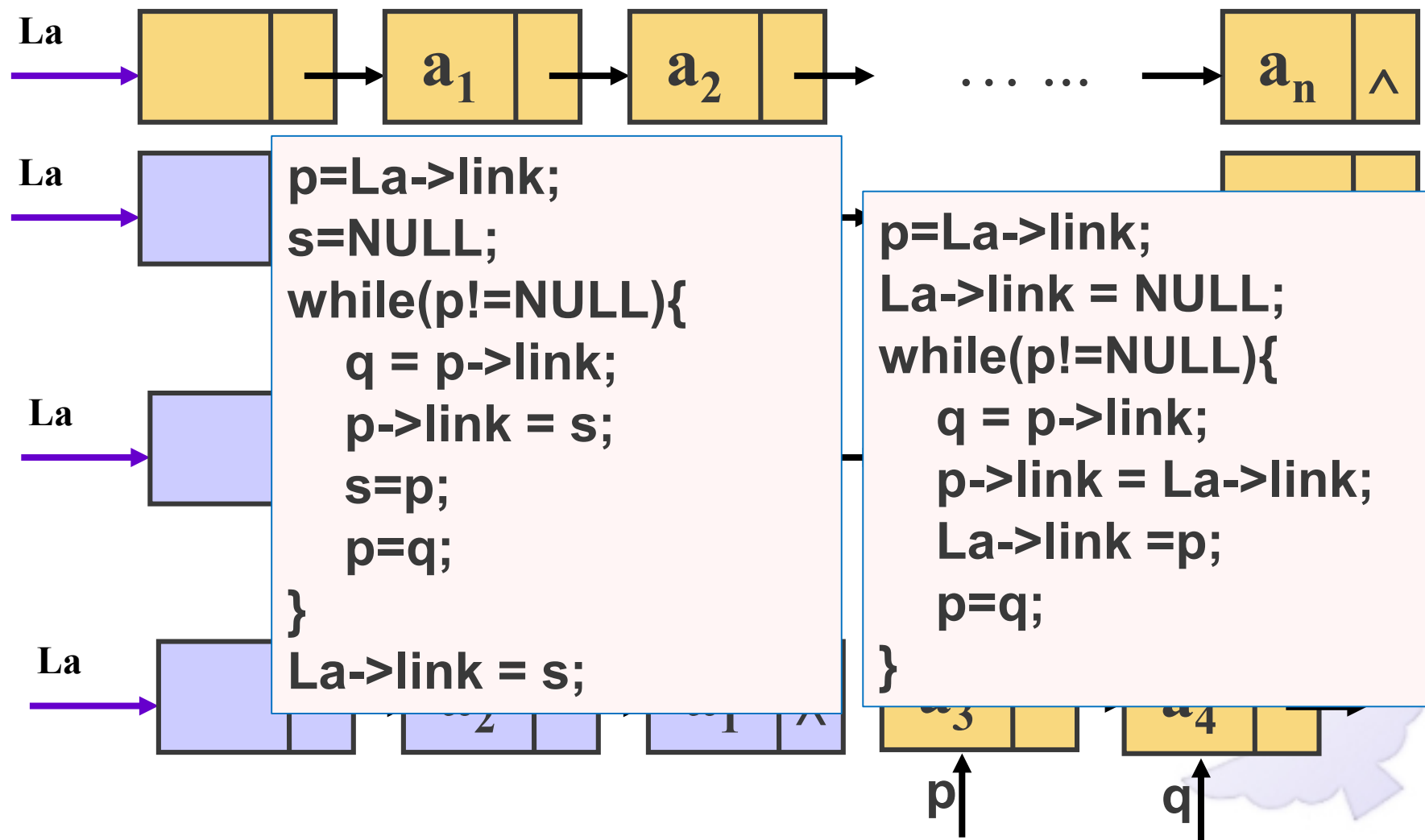


思路1：原地置逆



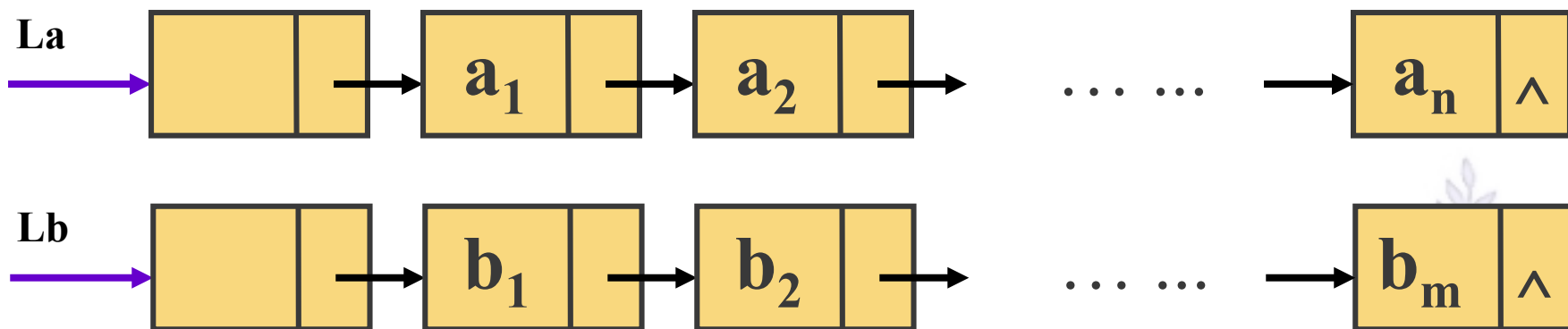
```
p=La->link; s=NULL;
while(p!=NULL){
    q = p->link;
    p->link = s;
    s=p;    p=q;
}
La->link = s;
```

思路2：逆序插入新表中



线性表习题-算法题

- ◆ 2、将两个非递减有序链表归并为一个非递增的有序链表（利用原表结点）。
- ◆ 3、将两个非递减有序链表归并为一个非递减的有序链表（利用原表结点）。





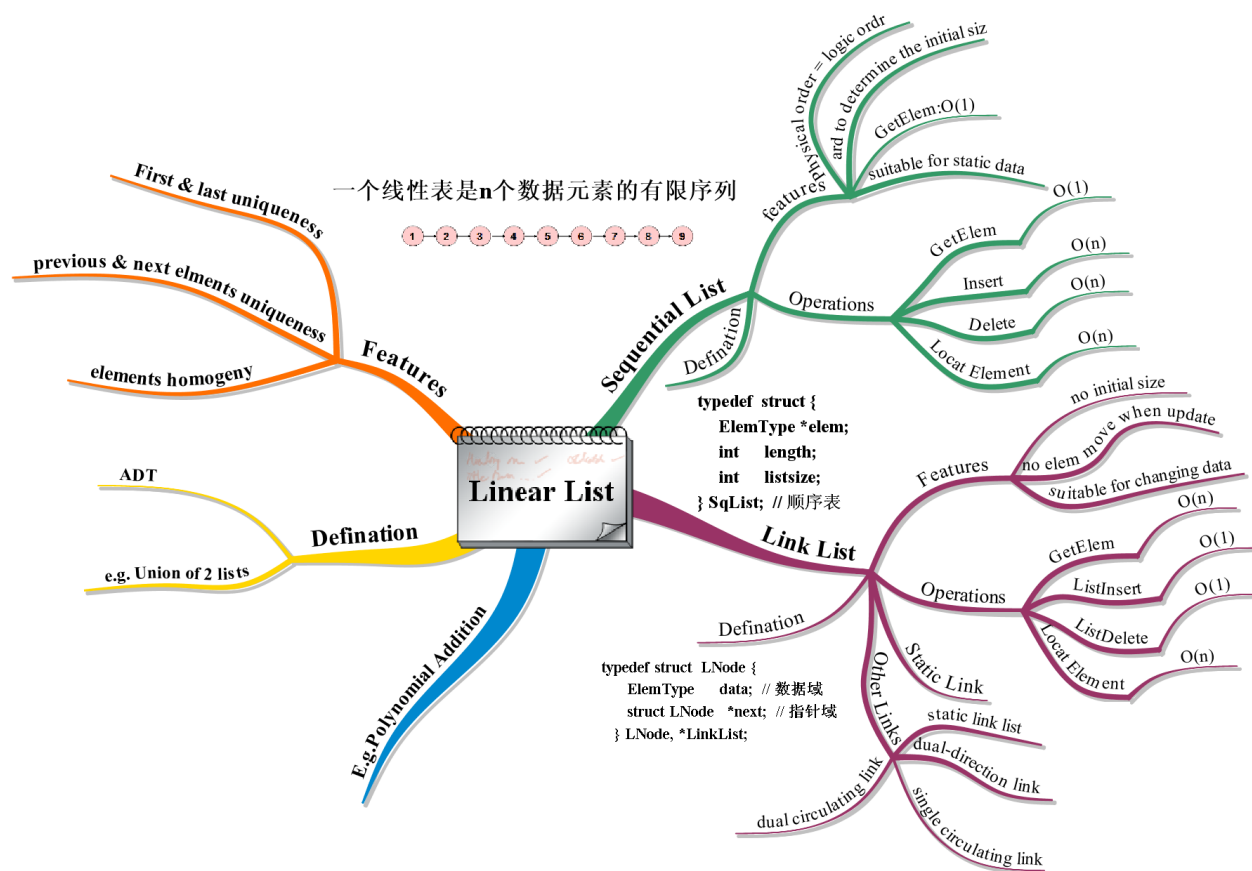
线性表习题-算法题

- ◆ 4、试设计一个对**双向链表**的进行查找的算法，要求在查找的同时，按访问频度重排链表（访问频度越高的结点越靠近链头）。
- ◆ [算法设计]先从左向右搜索需要查找的结点，如找到，则改变其访问频度，然后反向搜索其新的位置，最后从链中摘下此结点并将其插入到新的位置。为方便起见，在头结点的freq域中存入一极大的值。链表结点的结构为：

```
typedef struct DLinkNode {  
    int data, freq;  
    struct DLinkNode *llink, *rlink;  
}DLinkNode;
```



Review





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。





END of CHAPTER II

