

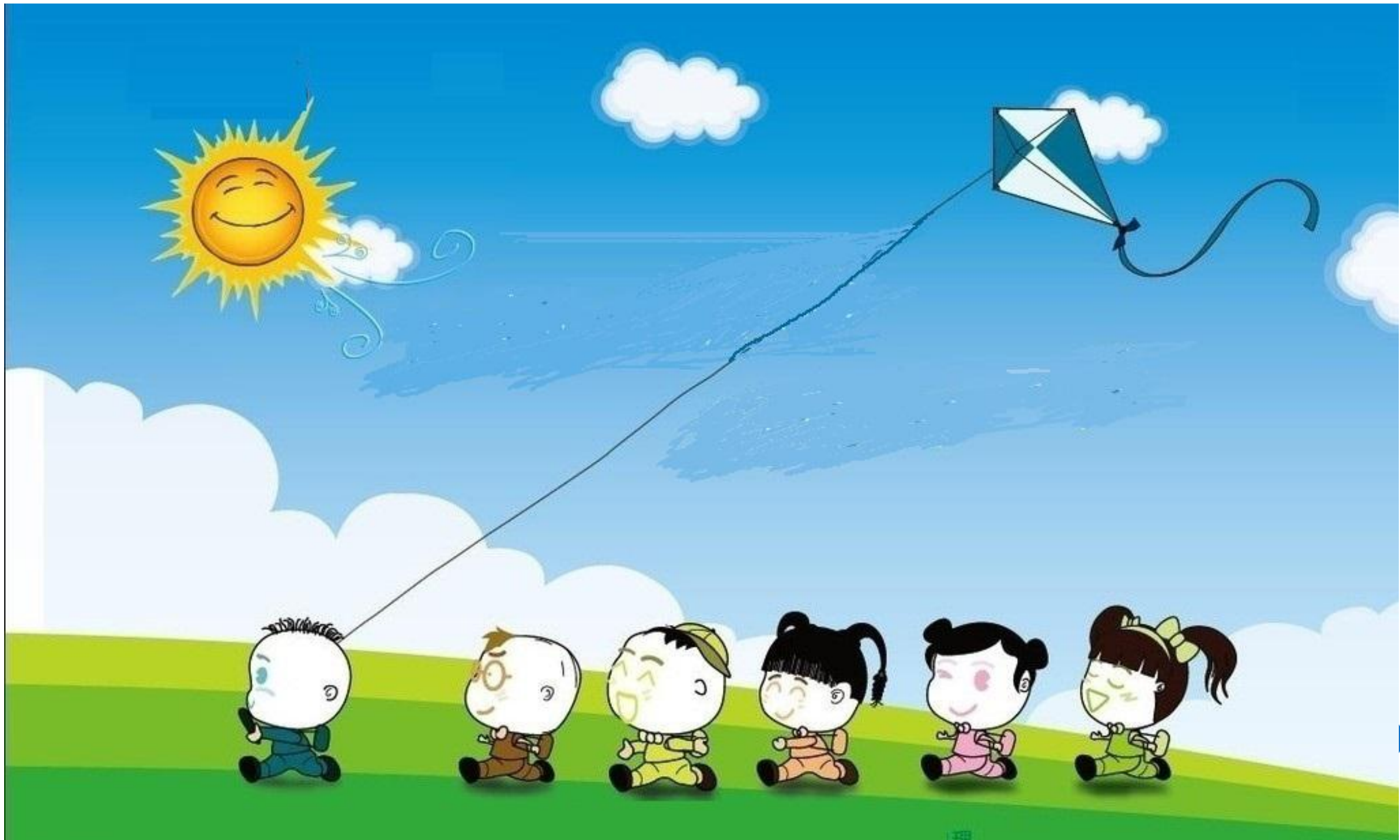


数据结构

DATA STRUCTURE

软件与通信工程学院 郭美

18075531998



○ 线性结构的定义

若结构是非空有限集，则有且仅有一个开始结点和一个终端结点，并且所有结点都最多只有一个直接前趋和一个直接后继。

可表示为： (a_1, a_2, \dots, a_n)

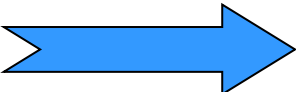


○ 线性结构的特点

- ① 只有一个首结点和尾结点；
- ② 除首尾结点外，其他结点只有一个直接前驱和一个直接后继。

简言之，线性结构反映结点间的逻辑关系是一对一的。

线性结构包括线性表、堆栈、队列、字符串、数组等等，其中，最典型、最常用的是-----

线性表  见第2章



第2章 线性表

教学目标

- 1. 了解线性结构的特点
- 2. 掌握顺序表的定义、查找、插入和删除
- 3. 掌握链表的定义、创建、查找、插入和删除
- 4. 能够从时间和空间复杂度的角度比较两种存储结构的不同特点及其适用场合



教学内容

2.1 线性表的定义和特点

2.2 线性的类型定义

2.3 线性表的顺序表示和实现

2.4 线性表的链式表示和实现

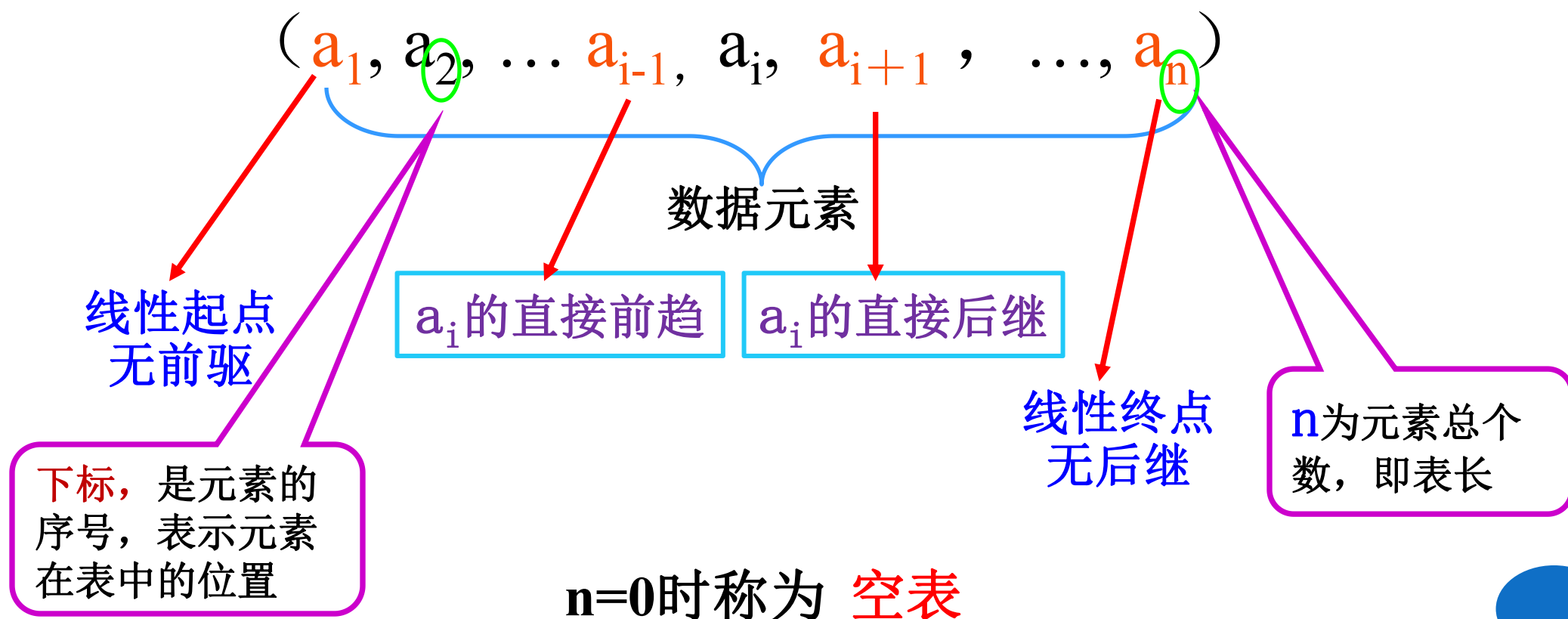
2.5 顺序表和链表的比较

2.6 线性表的应用



2.1 线性表的定义和特点

- 1、线性表的定义：是0个或多个数据元素的有限序列，表示为



2.1 线性表的定义和特点

例1 分析**26** 个英文字母组成的英文表

A, B, C, D, , Z

数据元素都是字母且个数有限；元素间关系是线性

例2 分析学生情况登记表

100条记录

学号	姓名	性别	年龄	班级
2001011810205	于春梅	女	18	2001级电信016班
2001011810260	何仕鹏	男	18	2001级电信017班
2001011810284	王 爽	女	18	2001级通信011班
2001011810360	王亚武	男	18	2001级通信012班
:	:	:	:	:

数据元素都是记录且记录数有限；元素间关系是线性

同一线性表中的元素必定具有相同特性

2.1 线性表的定义和特点

➤ 2、线性表的特点：

- (1) 存在**唯一的一个**被称作“**第一个**”的数据元素；
- (2) 存在**唯一的一个**被称作“**最后一个**”的数据元素；
- (3) 除第一个之外，结构中的每个数据元素均只有一个直接前驱；
- (4) 除最后一个之外，结构中的每个数据元素均只有一个直接后继。
- (5) 线性表里的元素必须**属于同一数据类型**。



2.2 线性表的类型定义（逻辑结构）

ADT List{

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

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

基本操作:

创建与初始化

InistList (&L) //构造空表 L

ClearList (&L) //置 L 为空表

置空

获取元素

GetElem (L, i, &e) //用 e 返回 L 中 第 i 个元素的值

LocatElem (L, e) //返回 L 中第一个值与 e 相同的元素在 L 中的位置, 若无, 则返回 0

查找

求表长度

ListLength (L) //返回 L 中数据元素的个数

ListInsert (&L, i, e) //在 L 中第 i 个位置之前插入新的数据元素 e, L 的长度加 1

插入

ListDelete (&L, i) //删除 L 的第 i 个数据元素, L 的长度减 1

删除

.....

} ADT List

// 完整的线性表的抽象数据类型定义参见教材P23

2.2 线性表的类型定义（逻辑结构）

说明：

- ◆ 抽象数据类型只是一个模型的定义，并不涉及模型的具体实现，因此描述中所涉及的参数不必考虑具体数据类型，在实际应用中可以根据具体需要选择使用不同的数据类型。
- ◆ 抽象数据类型里定义的操作是最基本的，但在实际问题中涉及关于线性表的操作会更复杂，可以用这些基本操作的组合来实现。



2.3 线性表的顺序表示和实现（存储结构）

➤ 1、线性表的顺序表示——又称为顺序存储结构或顺序映像。

顺序存储：把逻辑上相邻的数据元素存储在物理上相邻的存储单元中的存储结构。

简言之，逻辑上相邻，物理上也相邻！

顺序存储方法：用一组地址连续的存储单元依次存储线性表的元素，可通过数组 $V[n]$ 来实现。



2.3 线性表的顺序表示和实现

线性表顺序存储特点：

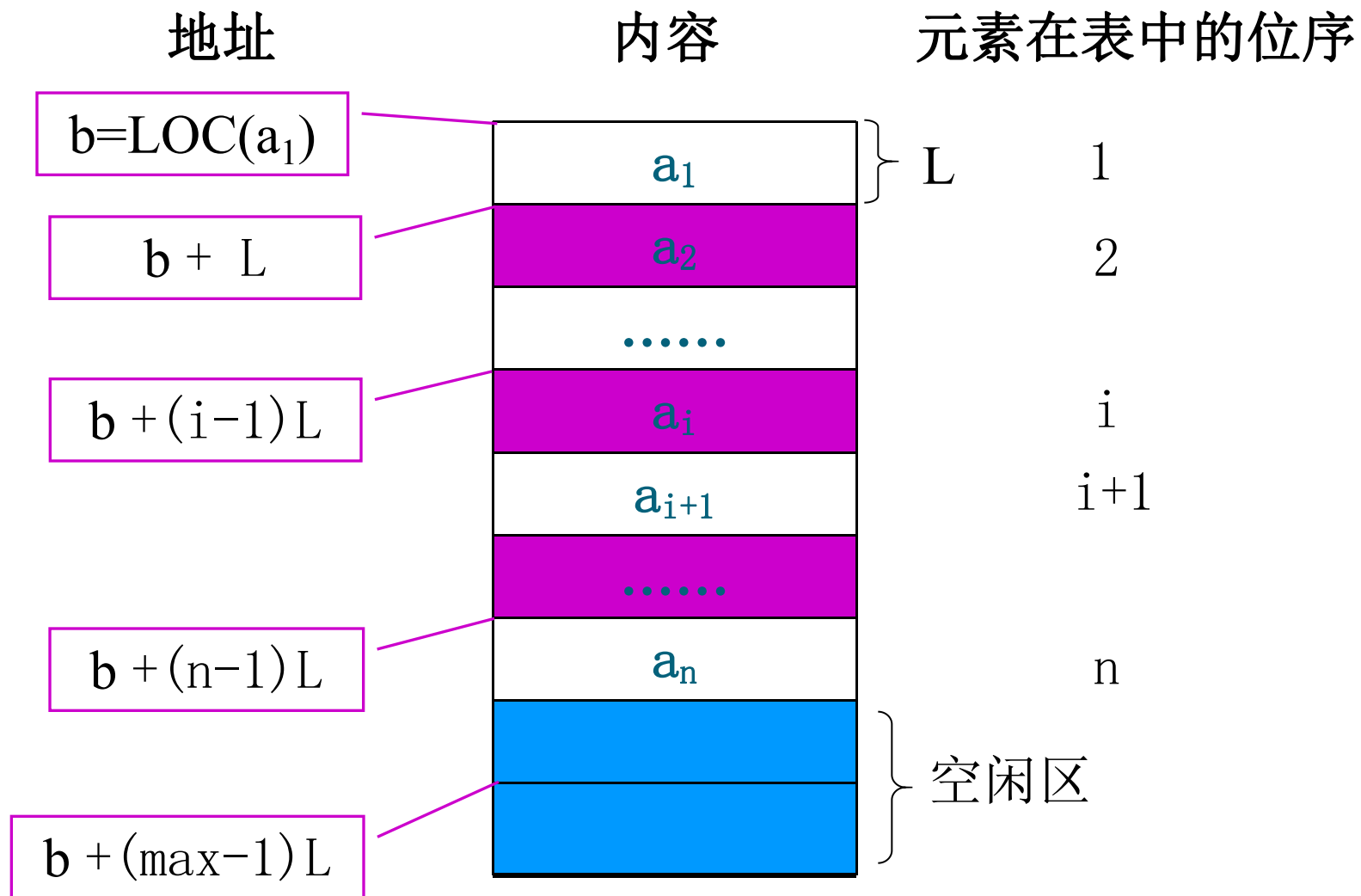
- 1) 逻辑上相邻的数据元素，其物理上也相邻；
- 2) 若已知表中首元素在存储器中的位置，则其他元素存放位置亦可求出（**利用数组下标**）。计算方法是（**参见存储结构示意图**）：

设首元素 a_1 的存放地址为 $LOC(a_1)$ （称为**首地址**），设每个元素占用存储空间（地址长度）为 L 字节，则表中任一数据元素的**存放地址**为：

$$\begin{aligned} LOC(a_i) &= LOC(a_1) + L * (i-1) \\ LOC(a_{i+1}) &= LOC(a_i) + L \end{aligned}$$



2.3 线性表的顺序表示和实现



注意：C语言中的数组的下标从0开始，即： $V[n]$ 的有效范围是 $V[0] \sim V[n-1]$

2.3 线性表的顺序表示和实现

在顺序表中求取任意位置的一个元素所用的时间都是相等，是一个常数，顺序表的存取时间复杂度为 $O(1)$ ，具有这一特点的存储结构称之为随机存取结构。



2.3 线性表的顺序表示和实现

例：一个一维数组M，下标的范围是0到9，每个数组元素用相邻的5个字节存储。存储器按字节编址，设存储数组元素M[0]的第一个字节的地址是98，则M[3]的第一个字节的地址是？

解：地址计算通式为：

$$\text{LOC}(a_i) = \text{LOC}(a_1) + L * (i-1)$$

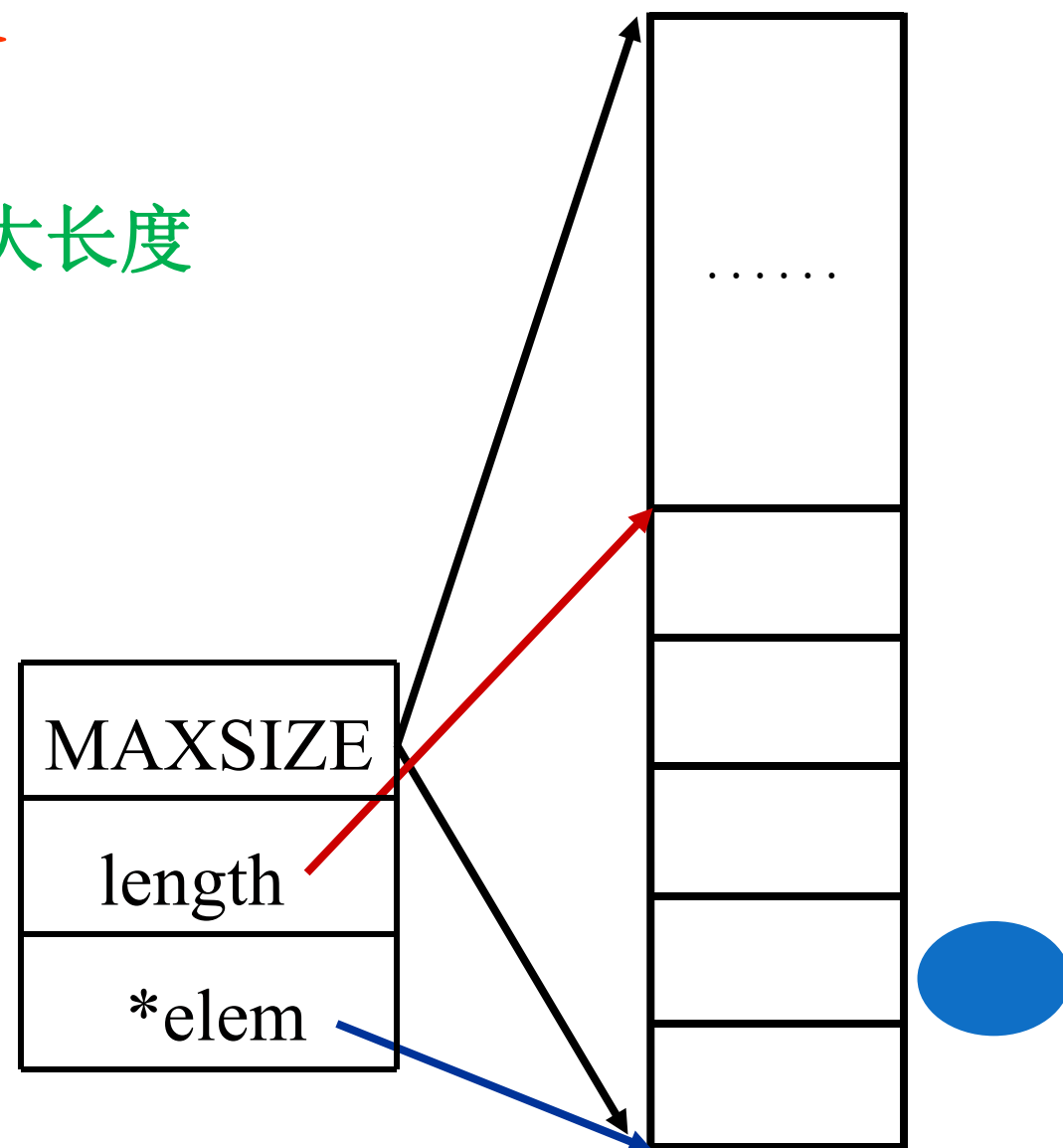
$$\text{因此：} \text{LOC}(M_{[3]}) = 98 + 5 \times (3-0) = 113$$



2.3 线性表的顺序表示和实现

➤ 2、顺序表的存储结构定义

```
#define MAXSIZE 100      //最大长度
typedef struct {
    ElemType *elem;
    //指向数据元素的基地址
    int length;
    //顺序表的当前长度
} SqList;
```



2.3 线性表的顺序表示和实现

➤ 2、顺序表的存储结构定义

区分：最大长度；当前长度

- ✓ 顺序表的最大长度是存放线性表的存储空间长度，存储分配后这个量一般是不变的。
- ✓ 顺序表的当前长度是顺序表中数据元素的个数，随着顺序表插入和删除操作的进行，这个量是变化的。
- ✓ 在任意时刻，顺序表当前长度应该小于等于顺序表的最大长度。



2.3 线性表的顺序表示和实现

➤ 3、顺序表的基本操作实现

○ 顺序表的初始化

//构造一个最多可存放100个整型数据的顺序表

```
int InitList(SqList &L){  
    L.elem=new int[100];  
    if(!L.elem) exit(-1);  
    L.length=0;  
  
    return 1;  
}
```

初始化线性表L （参数用引用）

```
Status InitList(SqList &L){           //构造一个空的顺序表L  
    L.elem=new ElemType[MAXSIZE]; //为顺序表分配空间  
    if(!L.elem) exit(OVERFLOW);      //存储分配失败  
    L.length=0;                       //空表长度为0  
    return OK;  
}
```

使得顺序表结构定义里的三个变量获得相应的值，从而在内存里创建一个具体的顺序表。

回顾引用类型作为函数参数

什么是引用???

引用：它用来给一个对象提供一个替代的名字。

```
#include<iostream.h>
void main(){
    int i=5;
    int &j=i;
    i=7;
    cout<<"i="<<i<<" j="<<j;
}
```

- ✓ **j**是一个引用类型，代表**i**的一个替代名
- ✓ **i**值改变时，**j**值也跟着改变，所以会输出
i=7 j=7



回顾引用类型作为函数参数

```
#include <iostream.h>

void swap(float &m, float &n)
{
    float temp;
    temp=m;
    m=n;
    n=temp;
}
```

```
void main()
{
    float a,b;
    cin>>a>>b;
    swap(a,b);
    cout<<a<<" "<<b<<endl;
}
```



回顾引用类型作为函数参数

引用类型作形参的三点说明：

- (1) 传递引用给函数与传递指针的效果是一样的，**形参变化实参也发生变化**。
- (2) 引用类型作形参，在内存中并没有产生实参的副本，它**直接对实参操作**；而一般变量作参数，形参与实参就占用不同的存储单元，所以形参变量的值是实参变量的副本。因此，当**参数传递的数据量较大**时，用引用比用一般变量传递参数的时间和空间效率都好。
- (3) 指针参数虽然也能达到与使用引用的效果，但在被调函数中需要重复使用“***指针变量名**”的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用**变量的地址作为实参**。



2.3 线性表的顺序表示和实现

算法时间复杂度分析？

➤ 3、顺序表的基本操作实现

$$T(n)=O(1)$$

○ 顺序表的取值

```
Status GetElem(SqList L,int i,ElemType &e){  
    if (i<1||i>L.length) return ERROR;  
    //判断i值是否合理，若不合理，返回ERROR  
    e=L.elem[i-1]; //第i-1的单元存储着第i个数据  
    return OK;  
}
```

通过数组的下标可访问（获取）顺序表中某个特定元素。

2.3 线性表的顺序表示和实现

算法时间复杂度分析？

➤ 3、顺序表的基本操作实现

○ 顺序表的查找

在线性表L中查找值为e的数据元素

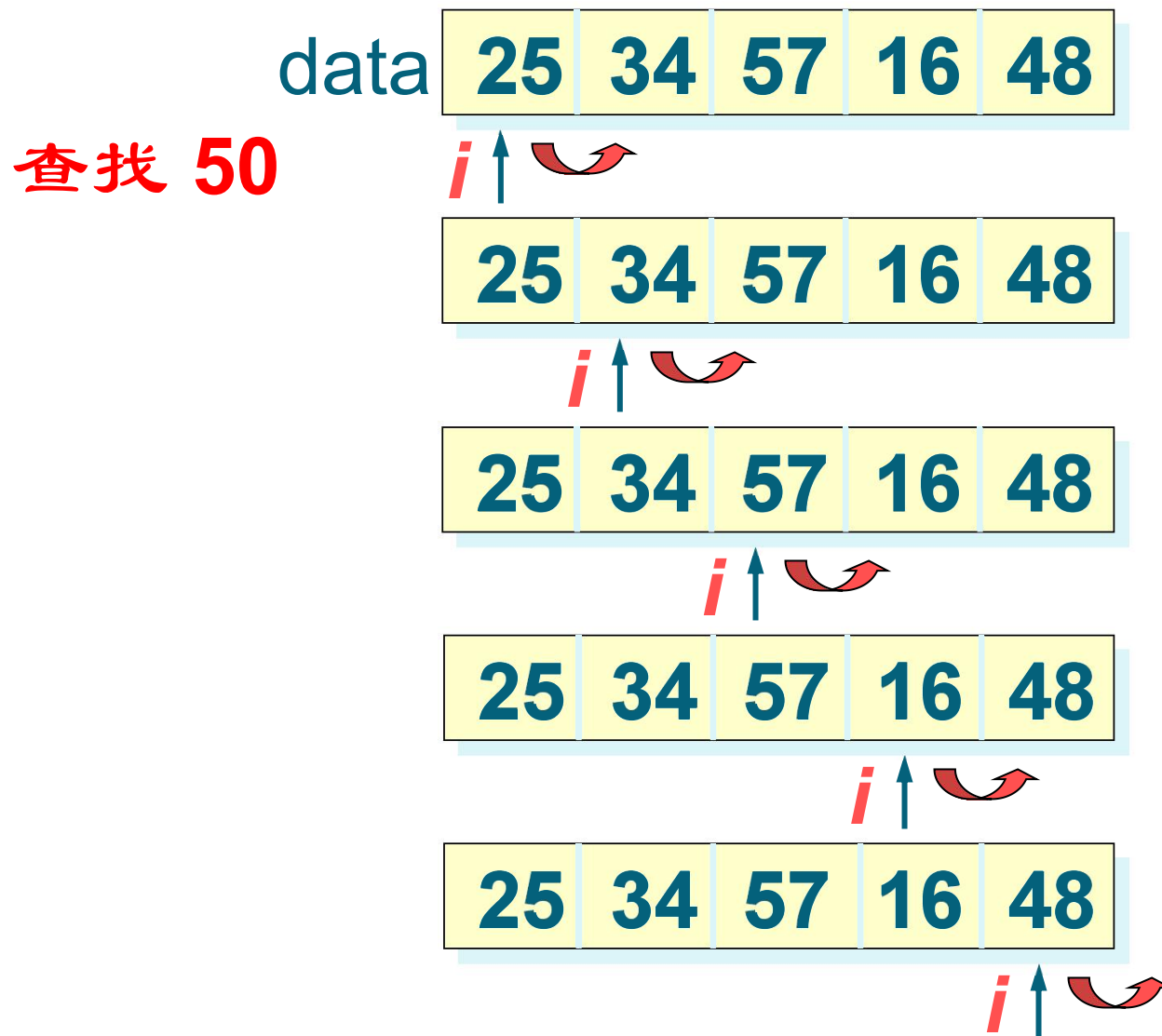
```
int LocateElem(SqList L, ElemType e)
{
    for (i=0; i< L.length; i++)
        if (L.elem[i]==e) return i+1;
    return 0;
}
```

从第一个元素起，依次和指定元素e比较，若找到与e相等的元素L.elem[i]，则查找成功，返回该元素序号i+1。

顺序查找图示



顺序查找图示



查找失败



○ 顺序表的查找

时间效率分析:

- 算法时间主要耗费在数据的比较上，比较元素的次数取决于被查找元素在顺序表中的位置。

- 假设在查找成功时，所找元素出现在各个位置上的概率相等，则

$$\text{有: } \frac{1 + 2 + 3 + \dots + n}{n} = \frac{(1 + n) \times n}{2n} = \frac{1 + n}{2}$$

顺序表按值查找算法的平均时间复杂度: **O (n)**



2.3 线性表的顺序表示和实现

➤ 3、顺序表的基本操作实现

○ 顺序表的插入

在顺序表的第*i*个位置前插入一个元素。

实现步骤：(n为表长)

- 1) 判断：插入位置*i* 是否合法?表是否已满?
应当有 $1 \leq i \leq n+1$ 或 $i=[1, n+1]$
- 2) 将第n至第*i*位的元素向后移动一个位置;
- 3) 将要插入的元素写到第*i*个位置;
- 4) 表长加1。



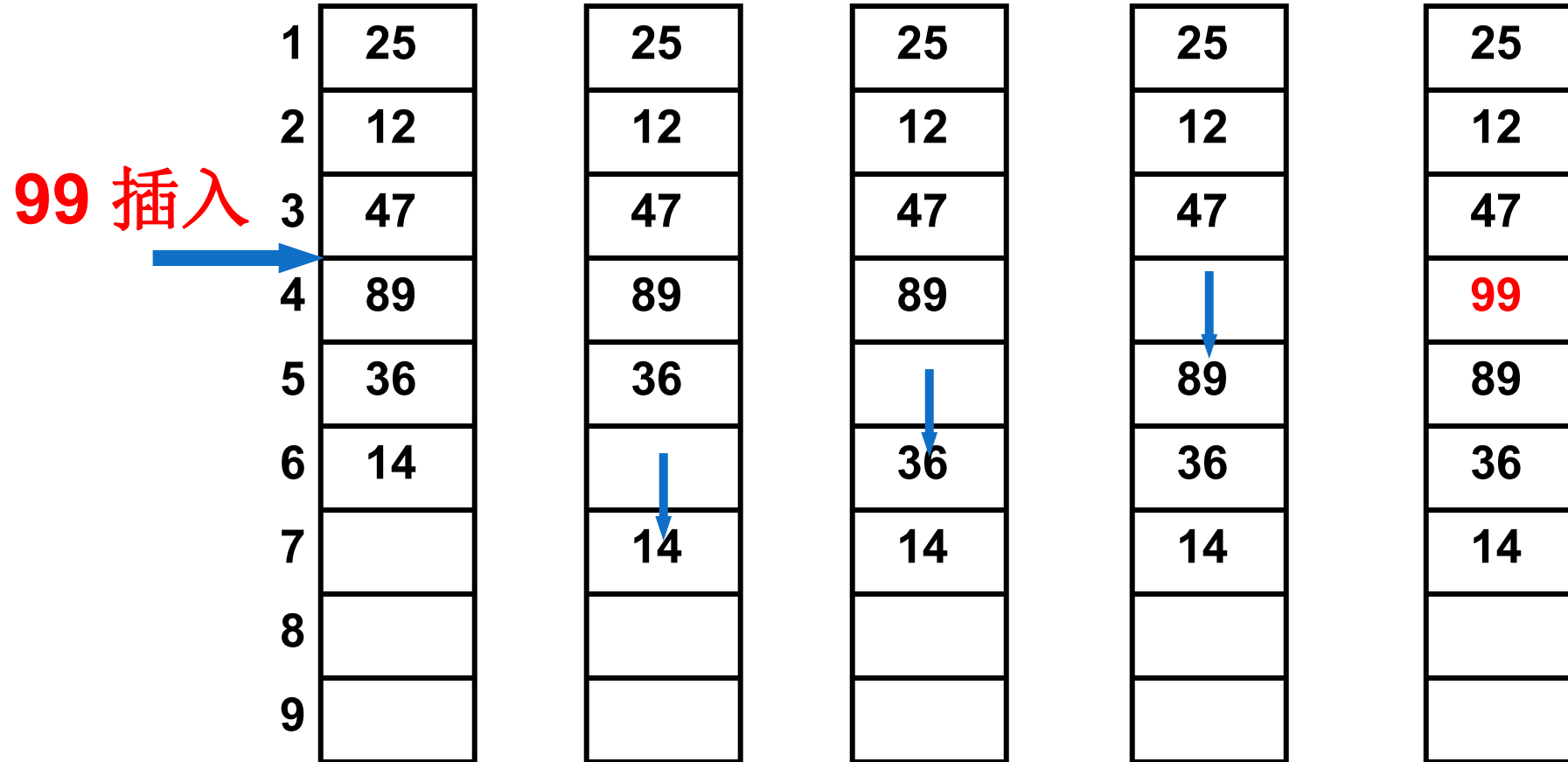
2.3 线性表的顺序表示和实现

算法时间复杂度分析？

➤ 3、顺序表的基本操作实现

○ 顺序表的插入

```
Status ListInsert_Sq(SqList &L,int i ,ElemType e){  
    if(i<1 || i>L.length+1) return ERROR;           //i值不合法  
    if(L.length==MAXSIZE) return ERROR;             //当前存储空间已满  
    for(j=L.length-1;j>=i-1;j--)  
        L.elem[j+1]=L.elem[j];    //插入位置及之后的元素后移  
    L.elem[i-1]=e;                //将新元素e放入第i个位置  
    ++L.length;                   //表长增1  
    return OK;  
}
```



插第 4 个结点之前，移动 $6-4+1$ 次

插在第 i 个结点之前，移动 $n-i+1$ 次



○ 顺序表的插入

时间效率分析:

- 算法时间主要耗费在移动元素的操作上，移动元素的次数取决于插入元素的位置。
- 考虑在各种位置插入（共 $n+1$ 种可能）的平均移动次数，且各个位置上的插入概率相等，则有：

$$\frac{0 + 1 + 2 + \dots + n}{n + 1} = \frac{(0 + n) \times (n + 1)}{2(n + 1)} = \frac{n}{2}$$

顺序表插入算法的平均时间复杂度： **$O(n)$**



2.3 线性表的顺序表示和实现

➤ 3、顺序表的基本操作实现

○ 顺序表的删除

删除顺序表第*i*个位置上的元素。

实现步骤：(*n*为表长)

1) 判断：删除位置*i* 是否合法？

应当有 $1 \leq i \leq n$ 或 $i \in [1, n]$

2) 将第*i*+1至第*n*位的元素向前移动一个位置；

3) 表长减1。



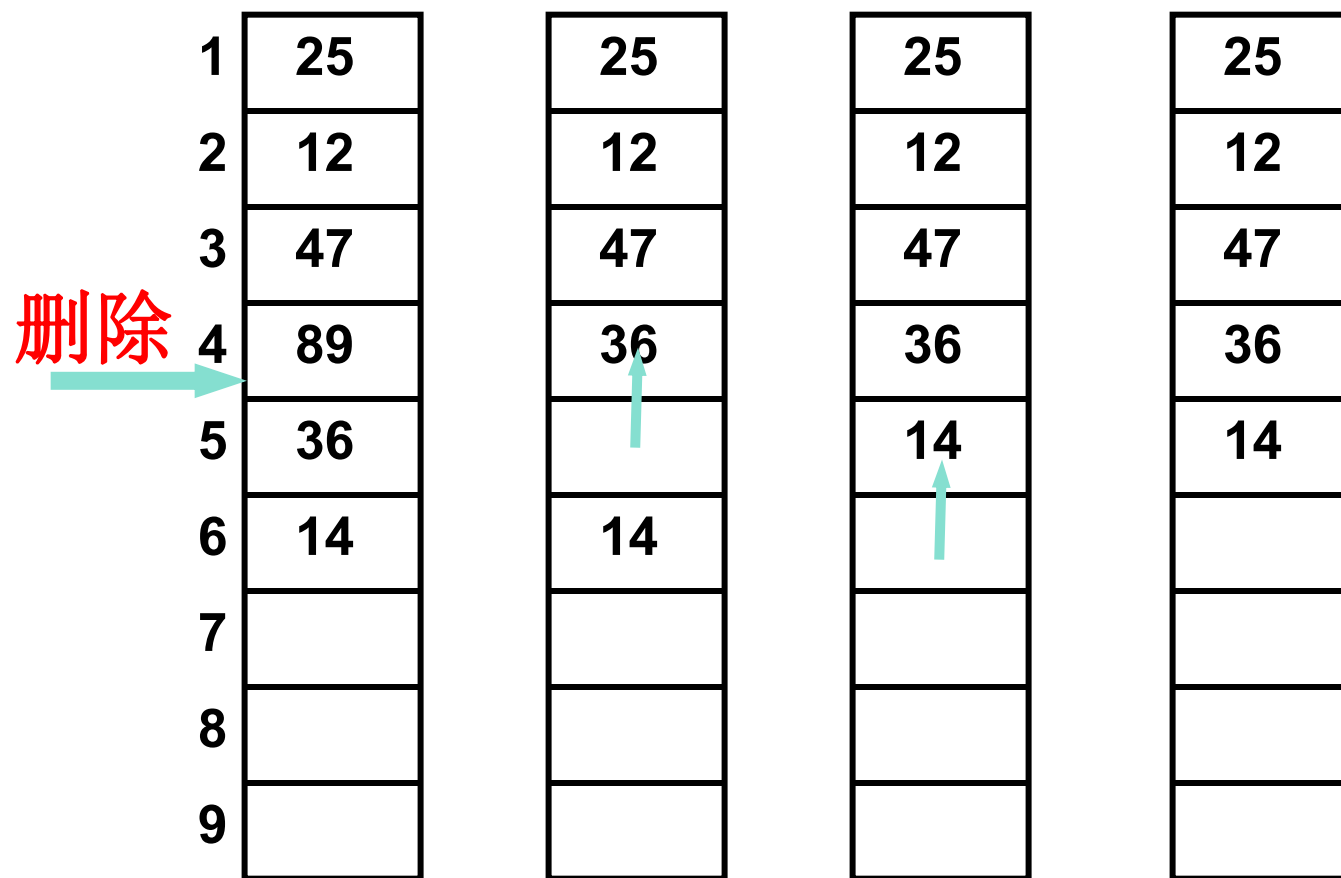
2.3 线性表的顺序表示和实现

算法时间复杂度分析？

➤ 3、顺序表的基本操作实现

○ 顺序表的删除

```
Status ListDelete_Sq(SqList &L,int i){  
    if((i<1)||i>L.length)) return ERROR; //i值不合法  
    for (j=i;j<=L.length-1;j++)  
        L.elem[j-1]=L.elem[j];    //被删除元素之后的元素前移  
    --L.length;    //表长减1  
    return OK;  
}
```



删除第 4 个结点，移动 $6-4$ 次

删除第 i 个结点，移动 $n-i$ 次



○ 顺序表的删除

时间效率分析:

- 算法时间主要耗费在移动元素的操作上，移动元素的次数取决于删除元素的位置。
- 考虑在各种位置删除（共 n 种可能）的平均移动次数，且各个位置上的删除概率相等，则有：

$$\frac{0 + 1 + 2 + \dots + n - 1}{n} = \frac{(n - 1) \times n}{2n} = \frac{n - 1}{2}$$

顺序表删除算法的平均时间复杂度： **$O(n)$**



顺序表的小结

线性表**顺序存储结构特点**：逻辑关系上相邻的两个元素在物理存储位置上也相邻。

优点：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**随机存取表中任一元素**。

缺点：在插入、删除某一元素时，需要移动大量元素。空间估计不明时，按最大空间分配。

为克服这一缺点，我们引入另一种存储形式：**链式存储结构**



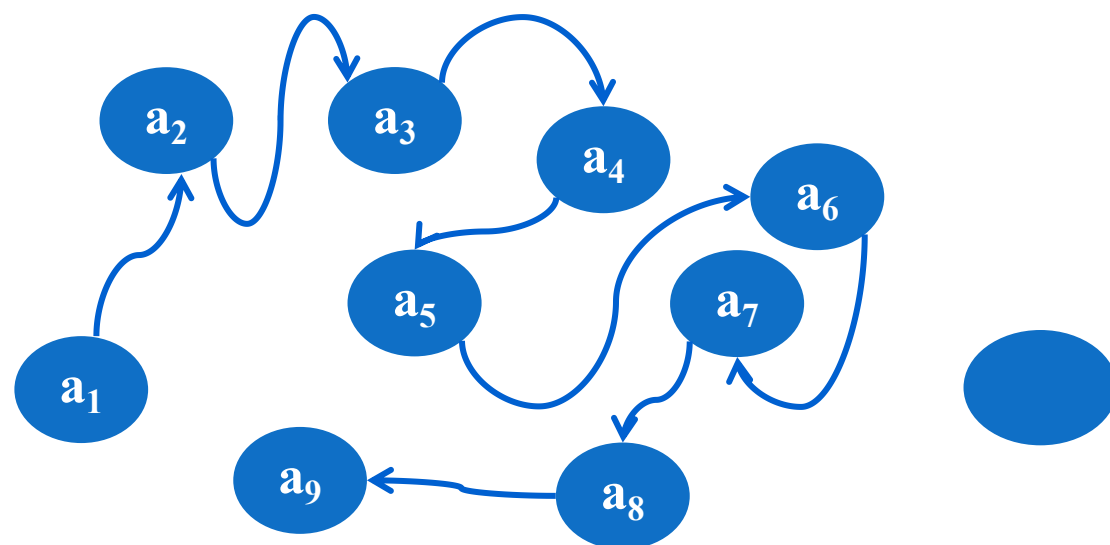
2.4 线性表的链式表示和实现

➤ 1、线性表的链式表示——又称为链式存储结构或非顺序映像。

链式存储：用一组任意的存储单元存储线性表的数据元素，这组存储单元可以是连续的，也可以是不连续的。

在链式存储中，数据元素除了需要存其自身的信息外，还要存储它后继元素的存储地址！

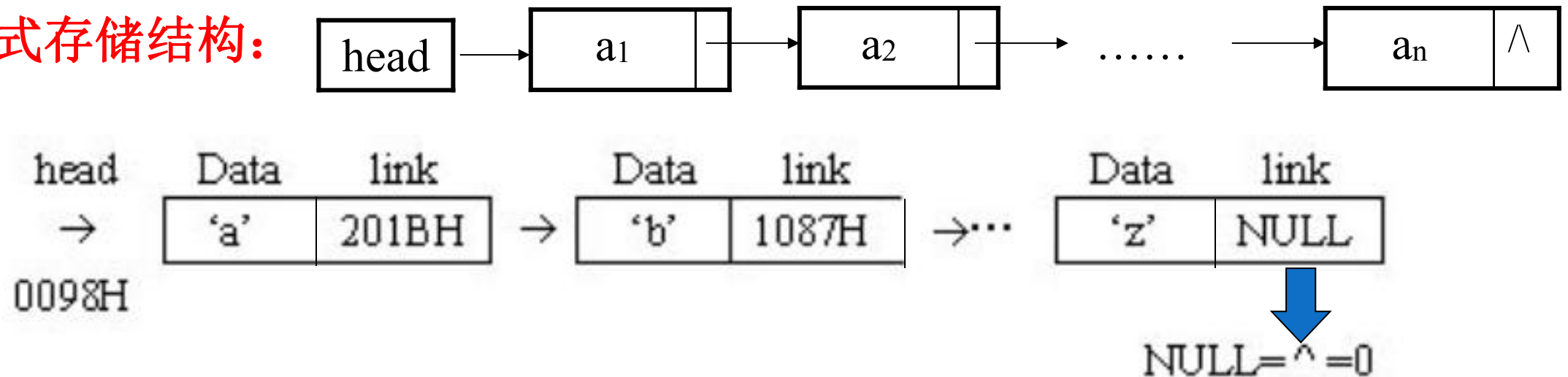
链式存放示意图



例：画出26个英文字母表的链式存储结构。

逻辑结构：（ a, b, ... , y, z）

链式存储结构：



讨论1：每个存储结点都包含两部分：数据和指针域(链域)。

讨论2：在单链表中，除了首元结点外，任一结点的存储位置由其直接前驱结点的链域的值指示。

2.4 线性表的链式表示和实现

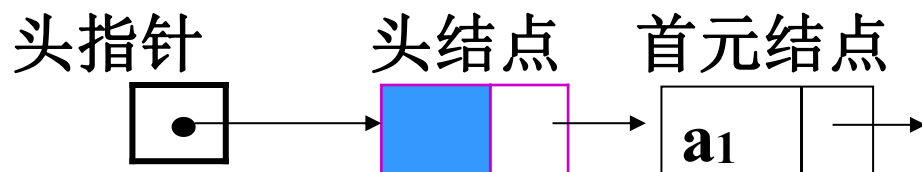
➤ 2、与链式存储有关术语

- 1) **结点**：数据元素的存储映像。由**数据域**和**指针域**两部分组成；
- 2) **链表**：n 个结点由指针链组成一个链表。它是线性表的链式存储映像，称为线性表的链式存储结构。
- 3) **单链表、双链表、多链表、循环链表**：
 - ◆ 结点只有一个指针域的链表，称为**单链表**或**线性链表**；
 - ◆ 有两个指针域的链表，称为**双链表**；
 - ◆ 有多个指针域的链表，称为**多链表**；
 - ◆ 首尾相接的链表称为**循环链表**。
- 4) **首元结点、头结点和头指针**



2.4 线性表的链式表示和实现

何谓**首元结点**、**头结点**和**头指针**？



首元结点是指链表中存储线性表**第一个数据元素** a_1 的结点。

头结点是在链表的首元结点之前附设的一个结点；数据域内只放空表标志和表长等信息。

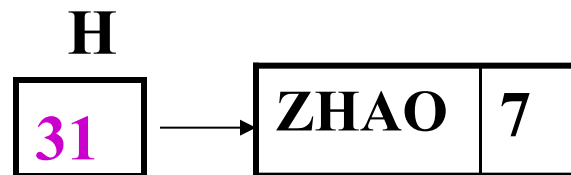
头指针是指向链表中第一个结点（或为头结点或为首元结点）的指针。**单链表可由一个头指针唯一确定。**

2.4 线性表的链式表示和实现

例：一个线性表的逻辑结构为：

(ZHAO, QIAN, SUN, LI, ZHOU, WU, ZHENG, WANG)，其存储表示如下，
请问其头指针的值是多少？

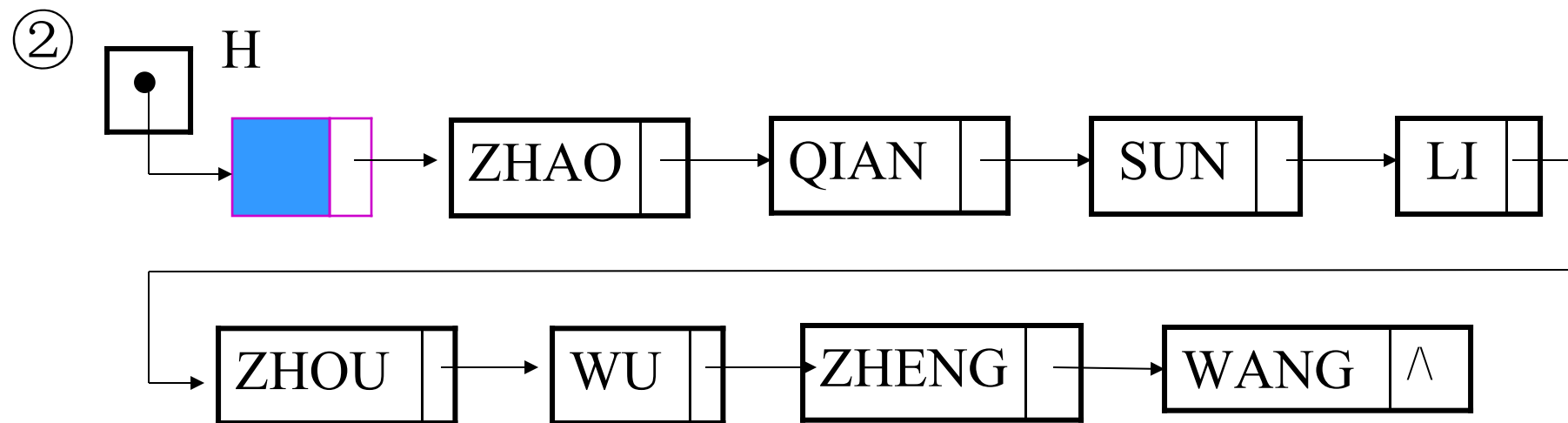
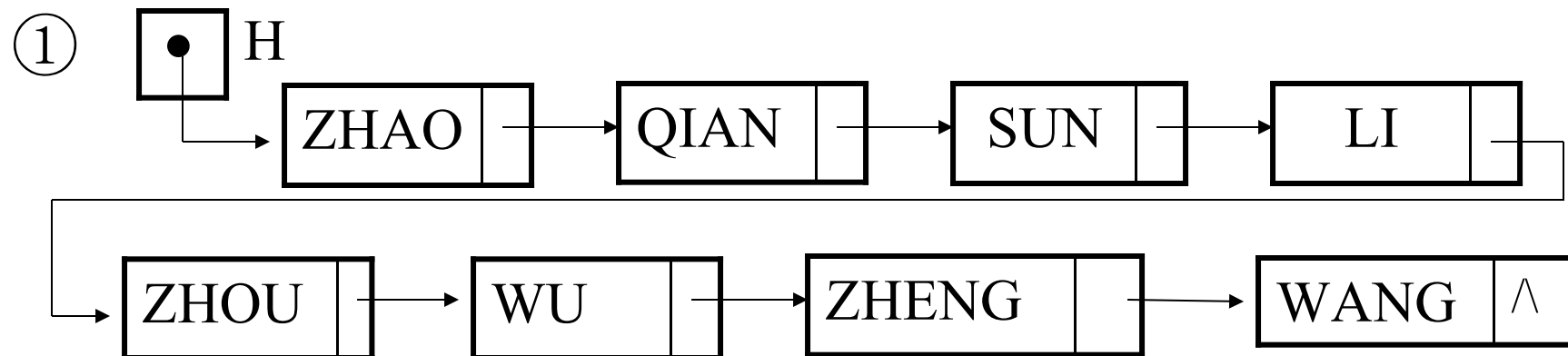
答：头指针是指向链表中第一个结点的指针，
因此关键是要寻找第一个结点的地址。



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

2.4 线性表的链式表示和实现

上例链表的逻辑结构示意图有以下两种形式：

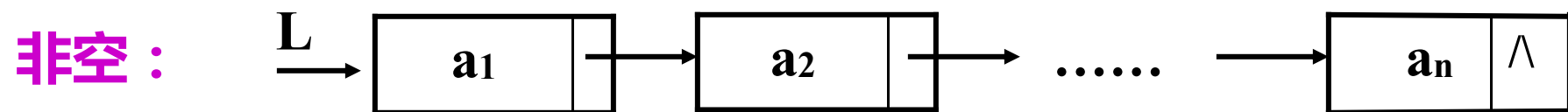


区别：① 无头结点 ② 有头结点

2.4 线性表的链式表示和实现

请思考：如何表示非空单链表和空单链表？

无头结点时：



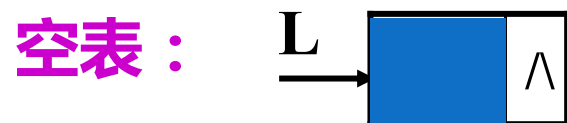
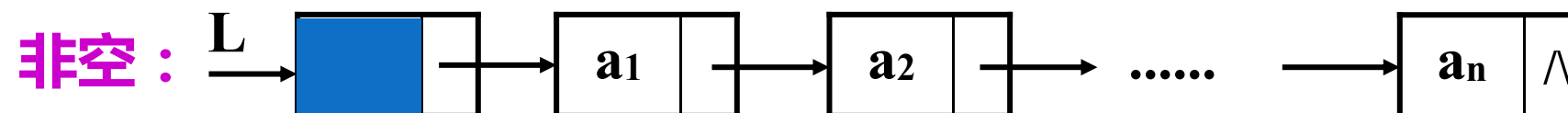
空表： $L = \text{NULL}$



2.4 线性表的链式表示和实现

请思考：如何表示非空单链表和空单链表？

有头结点时：



2.4 线性表的链式表示和实现

在链表中设置头结点的好处？

◆ 便于首元结点的处理

首元结点的地址保存在头结点的指针域中，所以在链表的第一个位置上的操作和其它位置一致，无须进行特殊处理。

◆ 便于空表和非空表的统一处理

无论链表是否为空，头指针都是指向头结点的非空指针，因此空表和非空表的处理也就统一了。



2.4 线性表的链式表示和实现

线性表链式存储特点：

- 1) 结点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻。
- 2) 访问时只能通过头指针进入链表，并通过每个结点的指针域向后扫描其余结点，所以寻找第一个结点和最后一个结点所花费的时间不等。

这种存取元素的方法被称为：**顺序存取法**



2.4 线性表的链式表示和实现

线性表链式存储优点：

- 1) 数据元素的个数可以自由扩充。
- 2) 插入、删除等操作不必移动数据，只需修改链接指针，修改效率较高。

线性表链式存储缺点：

- 1) 存储密度小
- 2) 存取效率不高，必须采用顺序存取，即存取数据元素时，只能按链表的顺序进行访问（顺藤摸瓜）

2.4 线性表的链式表示和实现

➤ 3、链表的存储结构定义

用C语言的结构指针来描述单链表的存储结构

```
Typedef struct LNode {  
    ElemType    data;           //数据域  
    struct LNode *next;        //指针域  
}LNode, *LinkList;           // *LinkList为LNode类型的指针
```

LNode *p; ↔ **LinkList p;**



2.4 线性表的链式表示和实现

➤ 3、链表的存储结构定义

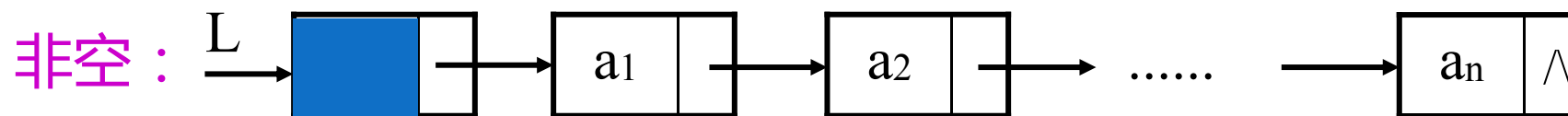
注意区分指针变量和结点变量两个不同的概念

```
LNode *p;
```

- 指针变量 p : 表示结点地址
- 结点变量 $*p$: 表示一个结点



2.4 线性表的链式表示和实现



结点a1的位置： $L \rightarrow next$ ；

结点a2的位置： $L \rightarrow next \rightarrow next$ ；以此类推

结点a1的值： $L \rightarrow next \rightarrow date$ ；

结点a2的值： $L \rightarrow next \rightarrow next \rightarrow date$ ；以此类推

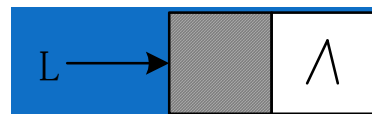


在单链表中，获取某个数据元素必须从**头指针出发**遍历链表，
故单链表是**非随机存取结构**。

2.4 线性表的链式表示和实现

➤ 4、单链表的基本操作实现

○ 单链表的初始化（构造一个空表）



【算法步骤】

- (1) 生成新结点作头结点，用头指针 L 指向头结点。
- (2) 头结点的指针域置空。

【算法描述】

```
Status InitList_L(LinkList &L){  
    L=new LNode;  
    L->next=NULL;  
    return OK;  
}
```

2.4 线性表的链式表示和实现

➤ 4、单链表的基本操作实现

○ 单链表的取值

难点分析：单链表中想取得第*i*个元素，必须从头指针出发寻找（顺藤摸瓜），不能随机存取。

实现步骤：

- （1）令指针

指向链表的首元结点，*j*作为计数器；
- （2）当*j*<*i*时，遍历链表，*p*不断指向下一结点（*p*=*p*->*next*），*j*累加1；
- （3）若到达表尾

为空，则第*i*个结点不存在；
- （4）否则查找成功，返回结点的数据（*p*->*date*）。



○ 单链表的取值

【算法描述】

```
Status GetElem_L(LinkList L, int i, ElemType &e){
```

```
    p=L->next; j=1; //初始化
```

```
    while(p&& j<i){ //向后扫描，直到p指向第i个元素或p为空
```

```
        p=p->next; ++j;
```

```
    }
```

```
    if(!p || j>i) return ERROR; //第i个元素不存在
```

```
    e=p->data; //取第i个元素
```

```
    return OK;
```

```
}
```

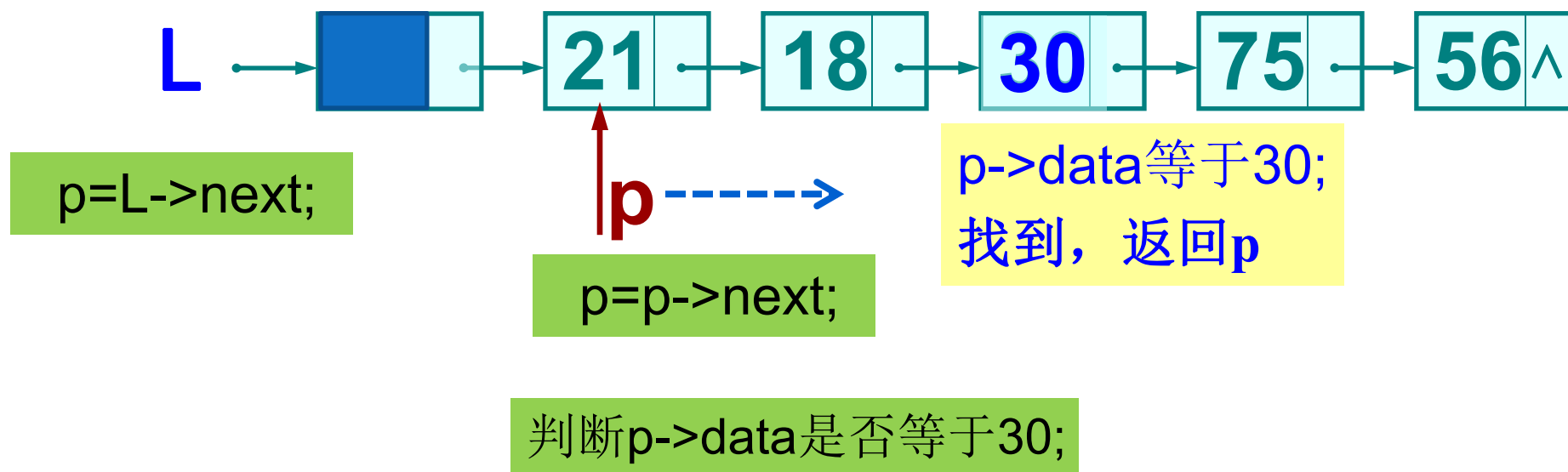
讨论：该算法的时间复杂度是多少？ $O(n)$

2.4 线性表的链式表示和实现

➤ 4、单链表的基本操作实现

○ 单链表的按值查找

e=30

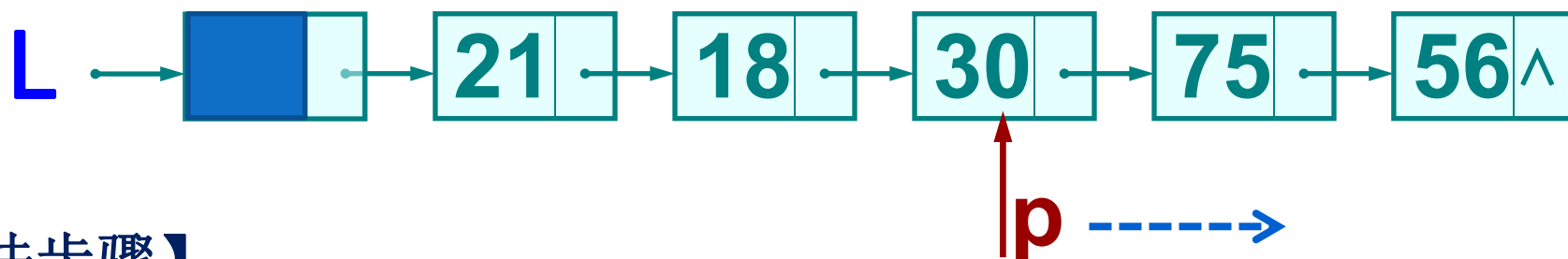


2.4 线性表的链式表示和实现

➤ 4、单链表的基本操作实现

○ 单链表的按值查找

e=50



p==NULL;
没找到，返回p

【算法步骤】

- (1) 从第一个结点起，依次和**e**相比较。
- (2) 如果找到一个其值与**e**相等的数据元素，则返回其在链表中的地址；
- (3) 如果查遍整个链表都没有找到其值和**e**相等的元素，则返回0或“NULL”。

p=p->next;

判断p->data是否等于50;

○ 单链表的按值查找

【算法描述】

```
LNode *LocateElem_L (LinkList L, Elemtype e) {
```

```
//返回L中值为e的数据元素的地址，查找失败返回NULL
```

```
    p=L->next;
```

```
    while(p && p->data!=e)
```

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

```
    return p;
```

```
}
```

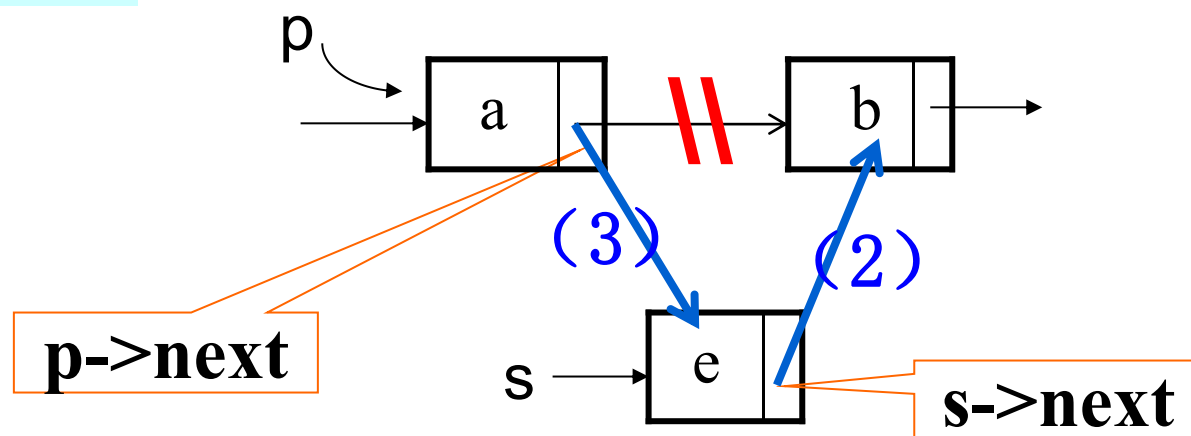
讨论：该算法的时间复杂度是多少？ **$O(n)$**

2.4 线性表的链式表示和实现

➤ 4、单链表的基本操作实现

○ 单链表的插入

在 a 和 b 中插入值为 e 的结点的示意图如下：



(1) 元素x结点应预先生成: `s=new LNode; s->data=e;`

(2) `s->next=p->next;`

思考: 步骤2和3能互换么?

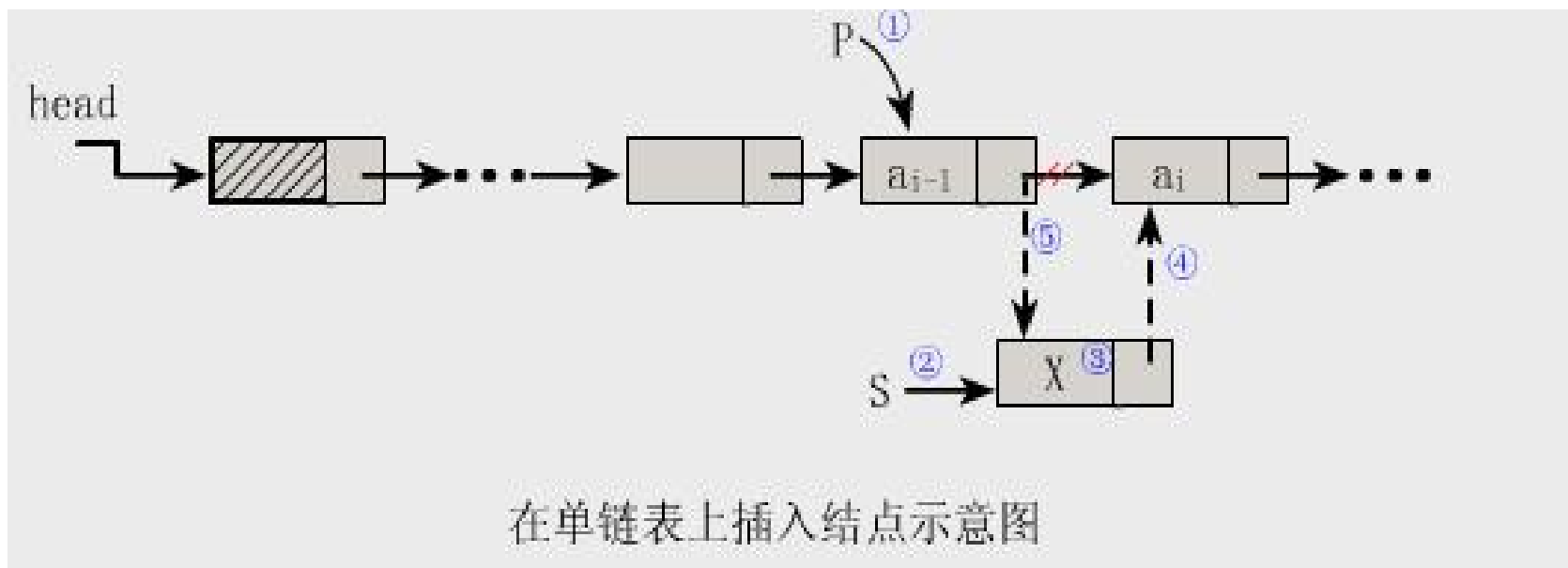
(3) `p->next=s ;`

○ 单链表的插入

在单链表L中第i个元素之前插入数据元素e

【算法步骤】

- (1) 找到 a_{i-1} 存储位置p
- (2) 生成一个新结点*s
- (3) 将新结点*s的数据域置为x
- (4) 新结点*s的指针域指向结点 a_i
- (5) 令结点*p的指针域指向新结点*s



○ 单链表的插入

在单链表L中第i个元素之前插入数据元素e

【算法描述】

```
Status ListInsert_L(LinkList &L,int i,ElemType e){
```

```
    p=L;j=0;
```

```
    while(p&& j<i-1){p=p->next;++j; }           //寻找第i-1个结点
```

```
    if(!p||j>i-1) return ERROR;                //i大于表长+1或者小于1
```

```
    s=new LNode;                                //生成新结点s
```

```
    s->data=e;                                   //将结点s的数据域置为e
```

```
    s->next=p->next;                             //将结点s插入L中
```

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

```
    return OK;    讨论：该算法的时间复杂度是多少？  $O(n)$ 
```

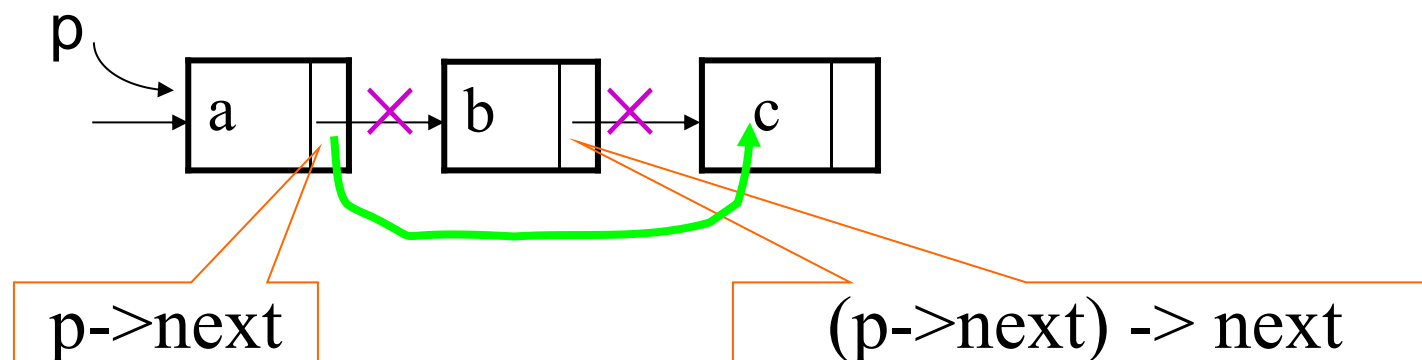
```
}//ListInsert_L
```

2.4 线性表的链式表示和实现

➤ 4、单链表的基本操作实现

○ 单链表的删除

在单链表中删除某个结点的示意图如下：



删除步骤（即核心语句）：

```
q = p->next;           //保存b的地址，靠它才能找到c
p->next=q->next;        //a、c两结点相连
free(q);                //删除b结点，彻底释放
```

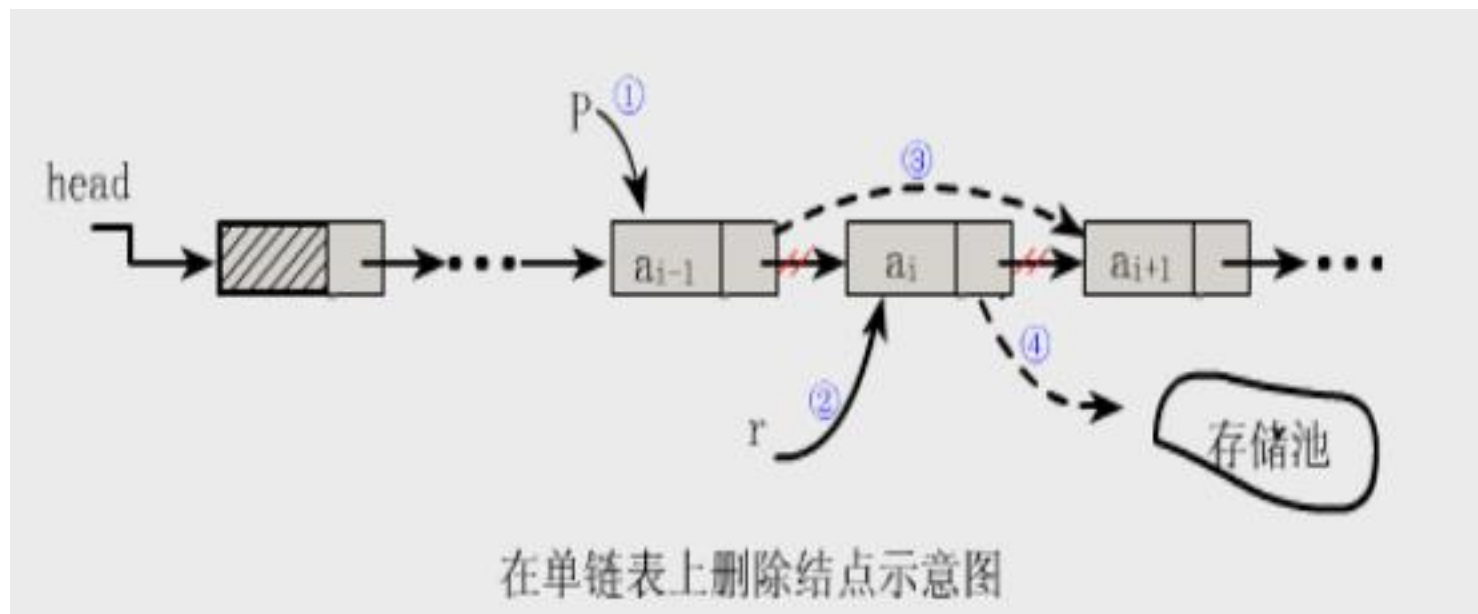
思考：省略free(q)语句行不行？

○ 单链表的删除

在单链表L中删除第 i 个数据元素 e

【算法步骤】

- (1) 找到 a_{i-1} 存储位置 p
- (2) 保存要删除的结点的值
- (3) 令 $p \rightarrow \text{next}$ 指向 a_i 的直接后继结点
- (4) 释放结点 a_i 的空间



○ 单链表的删除

在单链表L中删除第 i 个数据元素 e

【算法描述】

```
Status ListDelete_L(LinkList &L,int i,ElemType &e){
```

```
    p=L;j=0;
```

```
    while(p->next &&j<i-1){ //寻找第 $i$ 个结点，并令 $p$ 指向其前驱
```

```
        p=p->next; ++j;
```

```
    }
```

```
    if(!(p->next)||j>i-1) return ERROR; //删除位置不合理
```

```
    q=p->next; //临时保存被删结点的地址以备释放
```

```
    p->next=q->next; //改变删除结点前驱结点的指针域
```

```
    e=q->data; //保存删除结点的数据域
```

```
    free q; //释放删除结点的空间
```

```
    return OK;
```

```
}//ListDelete_L
```

讨论：该算法的时间复杂度是多少？ $O(n)$

单链表的运算时间效率分析

1. 取值、查找：因线性链表只能顺序存取，即在取值、查找时要从头指针找起，时间复杂度为 $O(n)$ 。
2. 插入和删除：因线性链表不需要移动元素，只要修改指针，一般情况下时间复杂度为 $O(1)$ 。
3. 但是，如果要在单链表中进行前插或删除操作，由于要从头查找前驱结点，所耗时间复杂度为 $O(n)$ 。



2.4 线性表的链式表示和实现

➤ 4、单链表的基本操作实现

○ 单链表的建立

难点分析：链表中的每个数据元素在内存中是“零散”存放的，链表所占用空间的大小和位置没有事先固定。

实现思路：动态生成过程，即从空表的初始状态起，依次建立各结点，并逐个**插入**链表中。



单链表的创建

单链表整表创建，实现步骤：

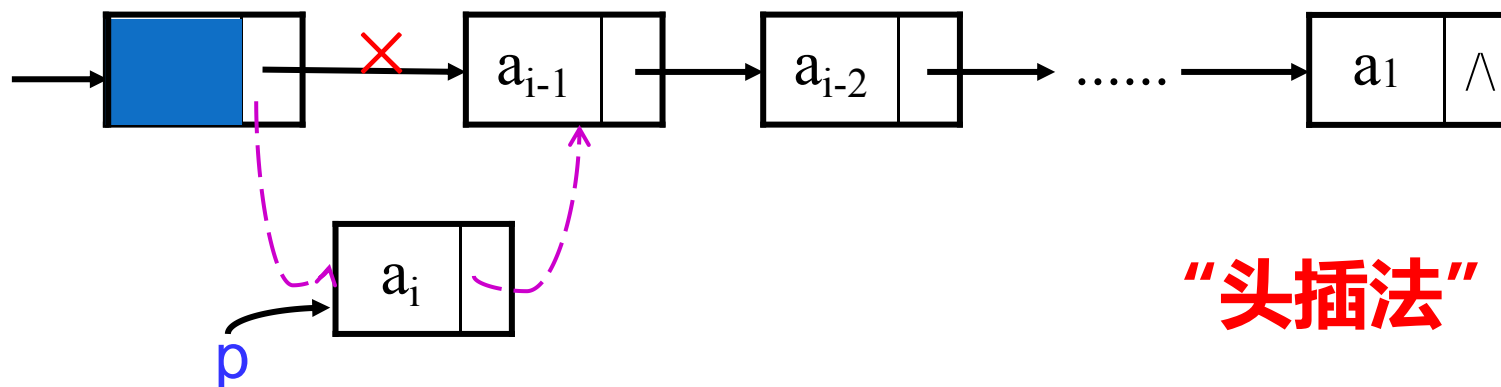
1) 建立一个带头结点的单链表，此时为空表；

2) 循环

◆ 生成一新结点 p ；

◆ 给该结点输入数据；

◆ 将 p 插入到头结点与前一新结点之间。



○ 单链表的创建(前插法)

“前插法”创建单链表，逆位输入n个元素的值

【算法描述】

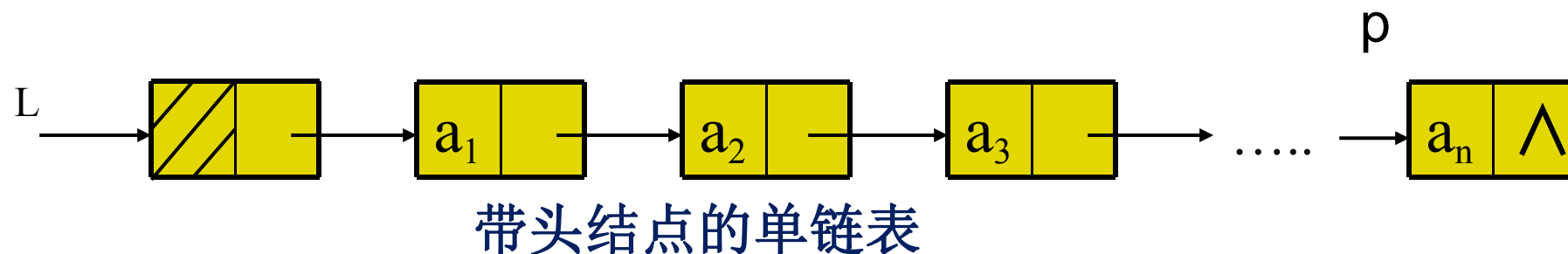
课后思考：“尾插法”如何实现？

```
void CreateList_F(LinkList &L,int n){  
    L=new LNode;  
    L->next=NULL;    //先建立一个带头结点的单链表  
    for(i=n;i>0;--i){  
        p=new LNode;    //生成新结点  
        cin>>p->data;    //输入元素值  
        p->next=L->next; L->next=p;    //插入到表头  
    }  
} //CreateList_F
```

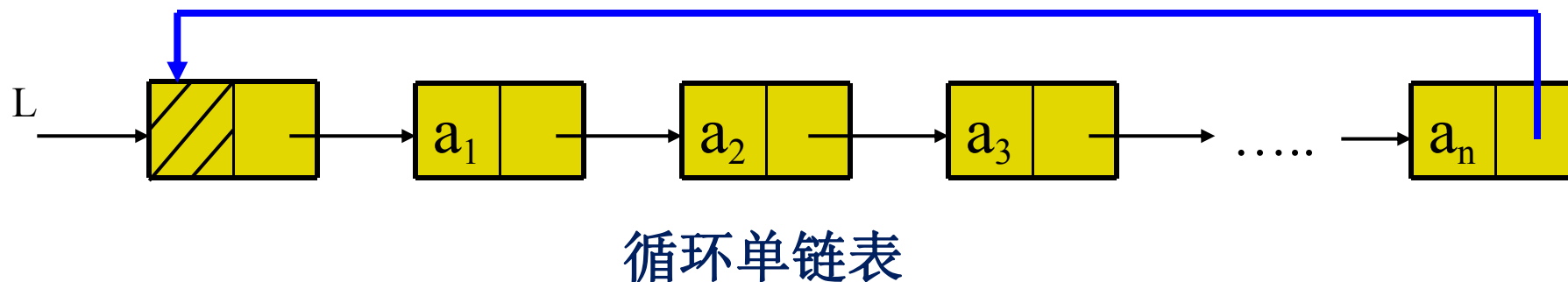
讨论：该算法的时间复杂度是多少？ $O(n)$

2.4 线性表的链式表示和实现

➤ 5、**循环链表**——将表中最后一个结点的指针域指向头结点 ($p \rightarrow \text{next} = L$;)，整个链表形成一个环。



若将链表中最后一个结点的next域指向头结点，则得到：



2.4 线性表的链式表示和实现

➤ 5、循环链表

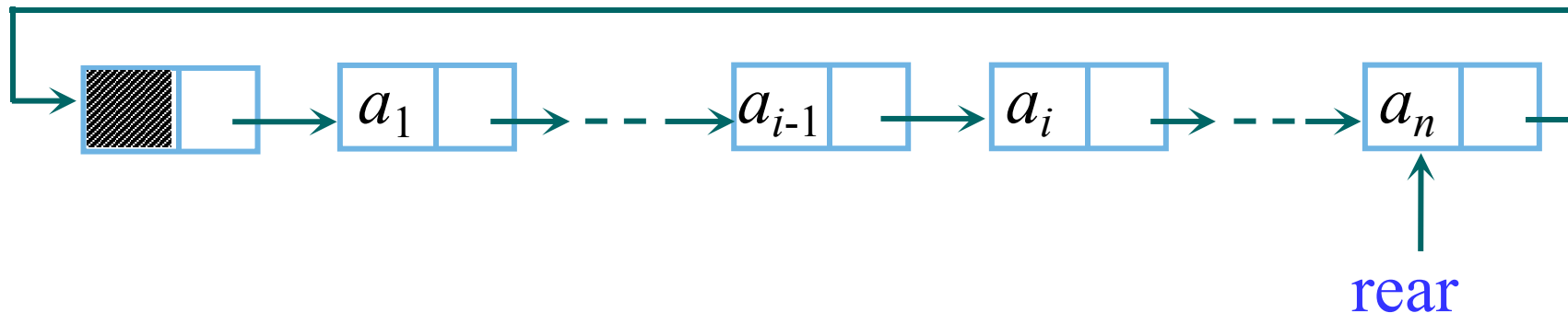
A. 特点：从任一结点出发均可找到表中其他结点。

B. 与单链表的区别：循环条件的判断

单链表 ----- $p = \text{NULL}$ 或 $p \rightarrow \text{next} = \text{NULL}$

循环链表 ----- $p = \text{head}$ 或 $p \rightarrow \text{next} = \text{head}$

C. 设立尾指针：可以使链表合并简化。

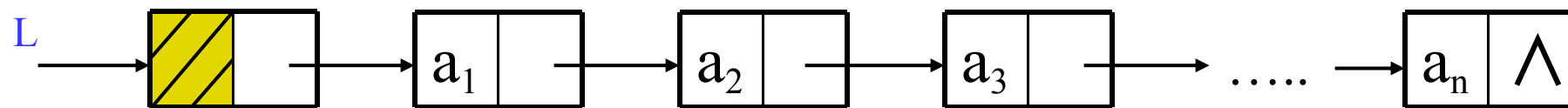




如何查找开始结点和终端结点

2.4 线性表的链式表示和实现

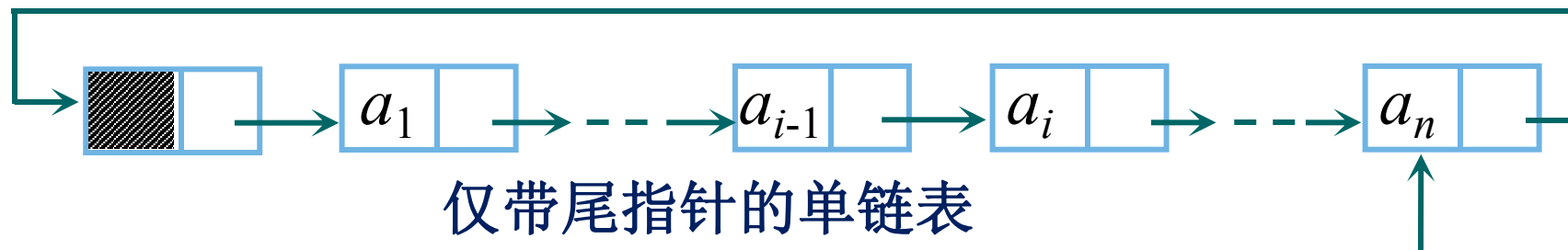
➤ 5、循环链表



带头结点的单链表

开始结点: $L \rightarrow \text{next}$, 时间为 $O(1)$

终端结点: 将单链表扫描一遍, 时间为 $O(n)$



仅带尾指针的单链表

开始结点: $\text{rear} \rightarrow \text{next} \rightarrow \text{next}$, 时间为 $O(1)$

终端结点: rear , 时间为 $O(1)$

rear



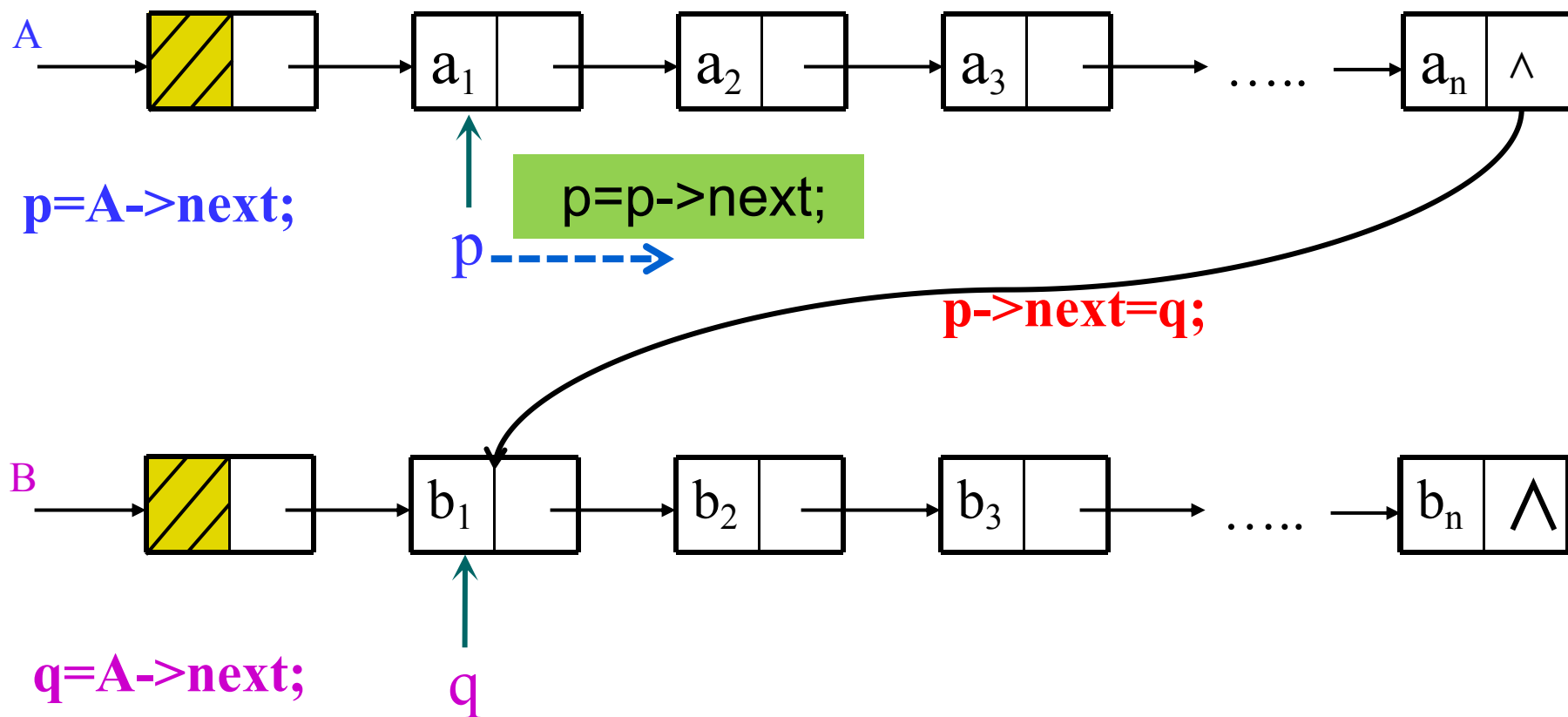
2.4 线性表的链式表示和实现



➤ 5、循环链表

带头结点的单链表情况

如何合并两个链表



时间为 $O(n)$

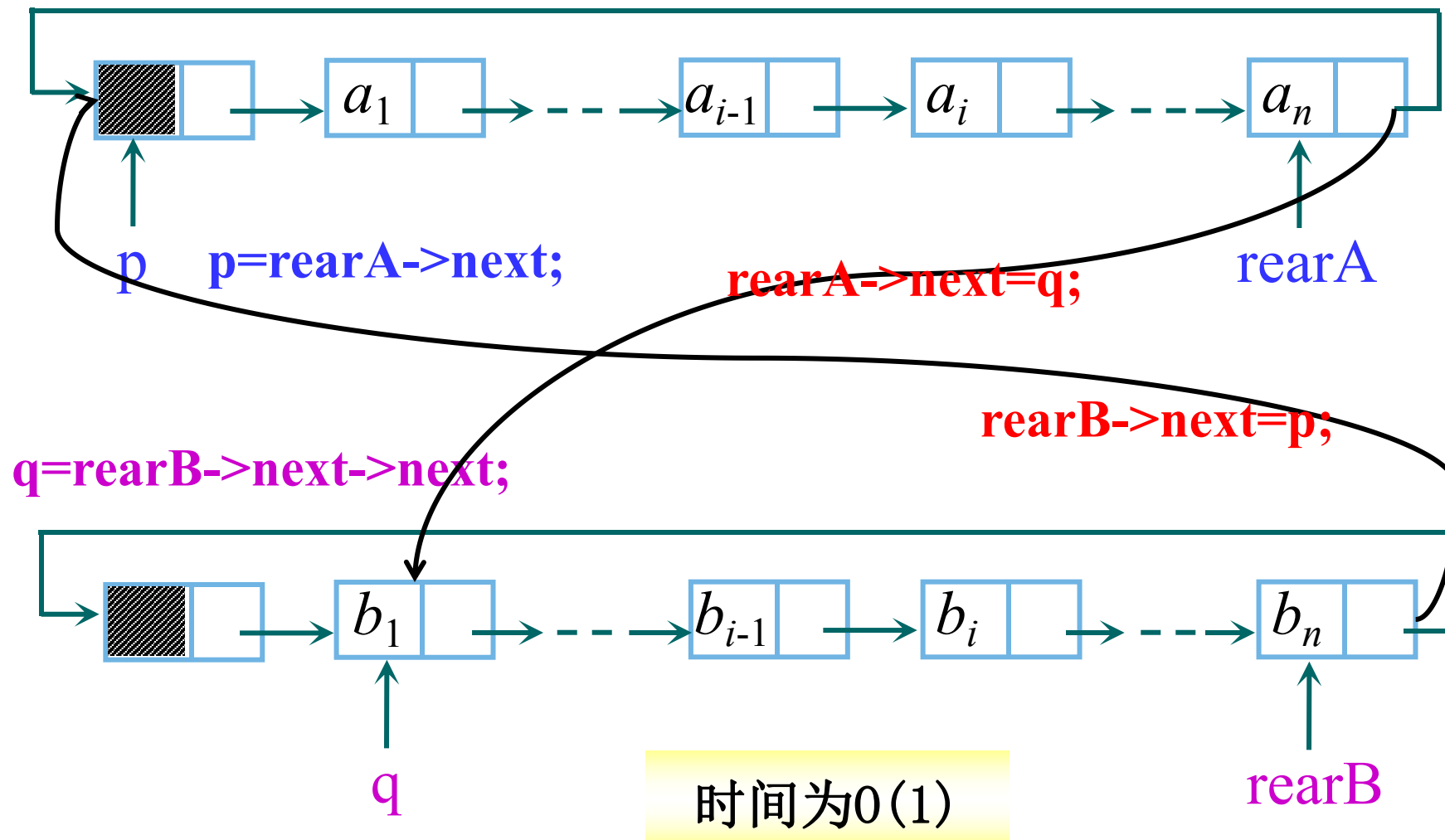
2.4 线性表的链式表示和实现



➤ 5、循环链表

仅带尾指针的单链表情况

如何合并两个链表



2.4 线性表的链式表示和实现

➤ 5、循环链表

实现循环链表的所有操作也都与单链表类似。只是判断链表结束的条件有所不同。下面列举两个循环链表操作的算法示例。

初始化循环链表CL

```
Status InitList(LinkList &CL) {  
    CL=(LinkList)malloc(sizeof(LNode));  
    if (CL) {CL-->next=CL; return OK;} //让next域指向它自身  
    else return ERROR ;  
}
```



2.4 线性表的链式表示和实现

➤ 5、循环链表

在循环链表CL中检索值为e的数据元素

```
Status GetElem_L(LinkList CL, int i, ElemType &e)
{
    P = CL->next; j=1;    // 初始化 , p指向第一个结点, j为计数器
    while(p!=CL&&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==CL||j>i) return ERROR;    // 第i个元素不存在
    e=p->data;    // 取第i个元素
    return OK;
} // GetElem_L
```



2.4 线性表的链式表示和实现

➤ 5、循环链表

在循环链表中，访问结点的特点：

访问后继结点，只需要向后走一步，而访问前驱结点，就需要转一圈。

结论：循环链表并不适用于经常访问前驱结点的情况。

解决方法：在需要频繁地同时访问前驱和后继结点的时候，使用双向链表。

双向链表 就是每个结点有两个指针域。一个指向后继结点，另一个指向前驱结点。



2.4 线性表的链式表示和实现

➤ 6、双向链表——有两个指针域的链表。

A. 结点结构:



B. 特点: 在每个结点中设置两个指针, 一个指向直接后继, 一个指向直接前驱。可直接确定一个结点的前驱和后继结点。可提高效率。

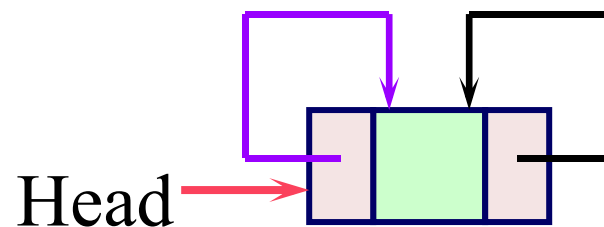
```
typedef struct DuLNode{  
    ElemType  data;  
    struct DuLNode *prior;  
    struct DuLNode *next;  
}DuLNode, *DuLinkList
```

注: 双向链表在非线性结构 (如树结构) 中将大量使用。

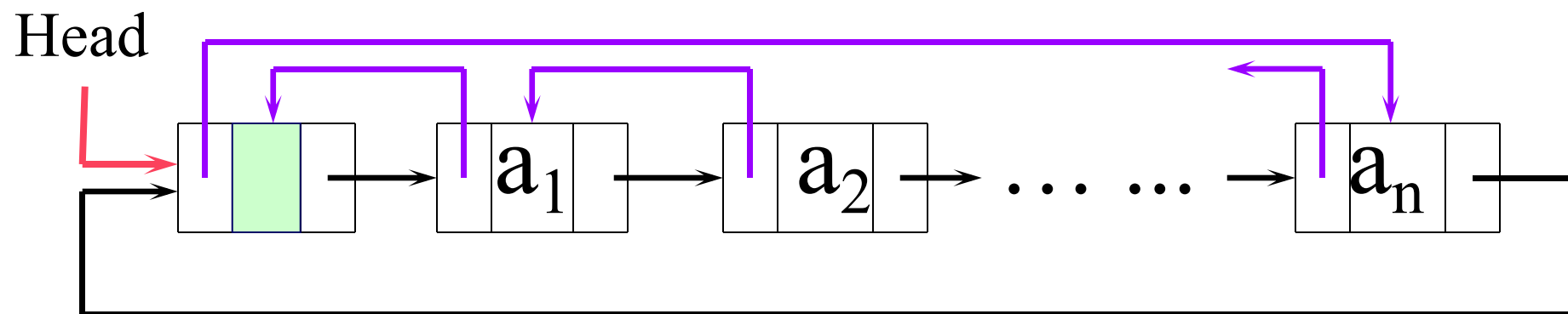
2.4 线性表的链式表示和实现

➤ 6、双向循环链表——将双向链表的头结点和尾结点链接起来。

空双向循环链表



非空双向循环链表



2.4 线性表的链式表示和实现

➤ 6、双向循环链表

○ 双向循环链表的建立

```
Status InitList (DuLinkList &L) {  
    L=new DuLNode;  
    if (L) { L->next=L->prior=L;  
            return OK;  
    }else  
        return OVERFLOW;  
}
```



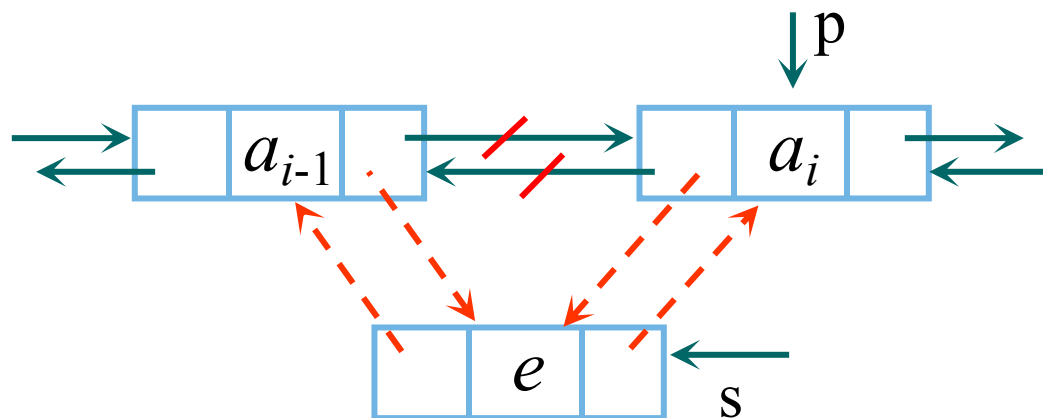
2.4 线性表的链式表示和实现

➤ 6、双向循环链表

在双向循环链表DL中，第 i 个数据元素

○ 双向循环链表的插入

之前插入数据元素 e 。



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

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

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

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

注意指针修改的相对顺序

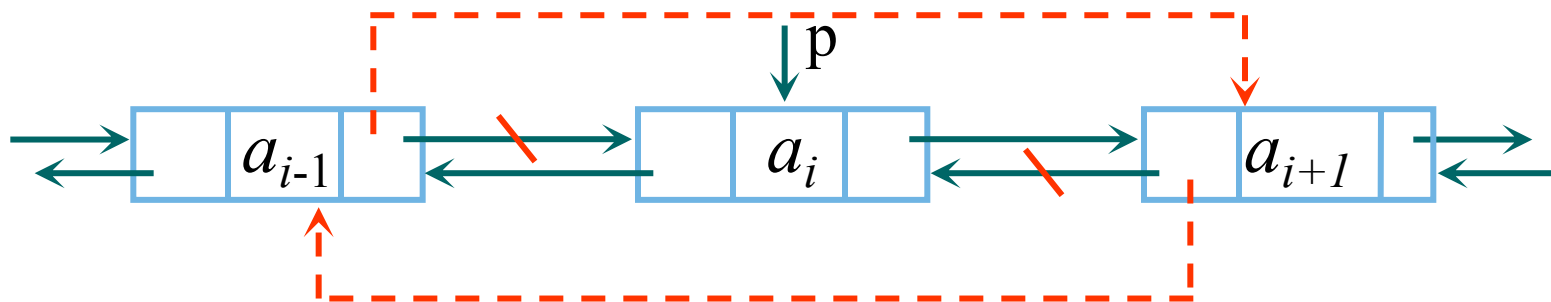
2.4 线性表的链式表示和实现

➤ 6、双向循环链表

在双向循环链表DL中，删除第*i*个数据

○ 双向循环链表的删除

元素e。



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

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

$\text{free}(p);$



2.4 线性表的链式表示和实现

线性表链式存储优点：

- 1) 数据元素的个数可以自由扩充。
- 2) 插入、删除等操作不必移动数据，只需修改链接指针，修改效率较高。

线性表链式存储缺点：

- 1) 存储密度小
- 2) 存取效率不高，必须采用顺序存取，即存取数据元素时，只能按链表的顺序进行访问（顺藤摸瓜）

2.5 顺序表和链表的比较

存储结构 比较项目		顺序表	链表
空间	存储空间	预先分配，会导致空间闲置或溢出现象	动态分配，不会出现存储空间闲置或溢出现象
	存储密度	不用为表示结点间的逻辑关系而增加额外的存储开销，存储密度等于1	需要借助指针来体现元素间的逻辑关系，存储密度小于1
时间	存取元素	随机存取，按位置访问元素的时间复杂度为 $O(1)$	顺序存取，按位置访问元素时间复杂度为 $O(n)$
	插入、删除	平均移动约表中一半元素，时间复杂度为 $O(n)$	不需移动元素，确定插入、删除位置后，时间复杂度为 $O(1)$
适用情况		<ul style="list-style-type: none">① 表长变化不大，且能事先确定变化的范围② 很少进行插入或删除操作，经常按元素位置序号访问数据元素	<ul style="list-style-type: none">① 长度变化较大② 频繁进行插入或删除操作



2.6 线性表的应用

➤ 1、线性表的合并

问题描述：

假设利用两个线性表La和Lb分别表示两个集合A和B，现要求一个新的集合

$$A=A\cup B$$

La=(7, 5, 3, 11)

Lb=(2, 6, 3)

La=(7, 5, 3, 11, 2, 6)



2.6 线性表的应用

➤ 1、线性表的合并

【算法步骤】

(1) 从线性表LB中依次取得每个数据元素。

GetElem(LB, i, &e);

(2) 在La中查找该元素。

LocateElem(LA, e);

(3) 如果找不到，则将其插入La的最后。

ListInsert(&LA, e);

La = (7 , 5 , 3 , 11)

Lb = (2 , 6 , 3)

La = (7 , 5 , 3 , 11 , 2 , 6)

}

2.6 线性表的应用

➤ 1、线性表的合并

【算法步骤】

(1) 从线性表LB中依次取得每个数据元素。

GetElem(LB, i, &e);

(2) 在La中查找该元素。

LocateElem(LA, e);

(3) 如果找不到，则将其插入La的最后。

ListInsert(&LA, e);

【算法描述】

```
void union(List &La, List Lb){  
    La_len=ListLength(La);  
    Lb_len=ListLength(Lb);  
    for(i=1;i<=Lb_len;i++){  
        GetElem(Lb,i,e);  
        if(!LocateElem(La,e))  
            ListInsert(&La,++La_len,e);  
    }  
}
```

2.6 线性表的应用

➤ 1、线性表的合并

○ 上机实现（采用顺序表）

// 顺序表类型的结构体定义

```
#define MAXSIZE 20
```

```
typedef struct{
```

```
    int *elem;
```

```
    int length;
```

```
}SqList;
```

// 创建顺序表 L（空表）

```
void InitList(SqList &L){
```

```
    L.elem = new int[MAXSIZE];
```

```
    L.length = 0;
```

```
}
```

// 求顺序表 L 的长度

```
int ListLength(SqList L){
```

```
    return L.length;
```

```
}
```

2.6 线性表的应用

➤ 1、线性表的合并

- 上机实现（采用顺序表）

// 求顺序表 L 的第i个元素，并以 e 返回

```
void GetElem(SqList L, int i, int &e){
```

```
    e = L.elem[i - 1];
```

```
}
```

// 判断 L 里有没有 e 这个元素

```
int LocateElem(SqList L, int e) {
```

```
    int i;
```

```
    for (i = 0; i < L.length; i++)
```

```
        if (e == L.elem[i])
```

```
            return 1;
```

```
    return 0;
```

```
}
```

2.6 线性表的应用

➤ 1、线性表的合并


○ 上机实现（采用顺序表）

// 将 e 插入到 L 的最后

```
void ListInsert(SqList &L, int e) {  
    L.elem[L.length] = e;  
    L.length++;  
}
```

// 线性表的合并（顺序表）

```
void unionList(SqList &LA, SqList LB) {  
    int LA_len, LB_len, i, e;  
    LA_len = ListLength(LA);  
    LB_len = ListLength(LB);  
    for (i = 1; i <= LB_len; i++) {  
        GetElem(LB, i, e);  
        if (!LocateElem(LA, e))  
            ListInsert(LA, e);  
    }  
}
```



2.6 线性表的应用

➤ 1、线性表的合并

○ 上机实现（采用顺序表）

// 输出顺序表

```
void ListOutput(SqList L){  
    int i;  
    for (i = 0; i < L.length; i++)  
        cout << L.elem[i] << " ";  
    cout << endl;  
}
```

```
int main() {  
    SqList LA, LB;  
    int d; InitList(LA);  InitList(LB);  
    cout<<"请输入LA表中的元素: "<<endl;  
    for(int i=0;i<4;i++){  
        cin>>x;  
        ListInsert(LA,x);  }  
    cout<<"请输入LB表中的元素: "<<endl;  
    for(i=0;i<3;i++){  
        cin>>x;  
        ListInsert(LB,x);  }  
    unionList(LA, LB);  
    cout << "LA和LB合并后的集合为: \n";  
    ListOutput(LA);  
    return 0;  
}
```

课后思考：该算法的时间复杂度是多少？

2.6 线性表的应用

➤ 2、有序表的合并

问题描述:

已知线性表La 和Lb中的数据元素按值非递减有序排列, 现要求将La和Lb归并为一个新的线性表Lc, 且Lc中的数据元素仍按值非递减有序排列。

La=(1 ,7, 8)

Lb=(2, 4, 8, 10, 11)

Lc=(1, 2, 4, 7 , 8, 8, 10, 11)



2.6 线性表的应用

➤ 2、有序表的合并 (以单链表为例)

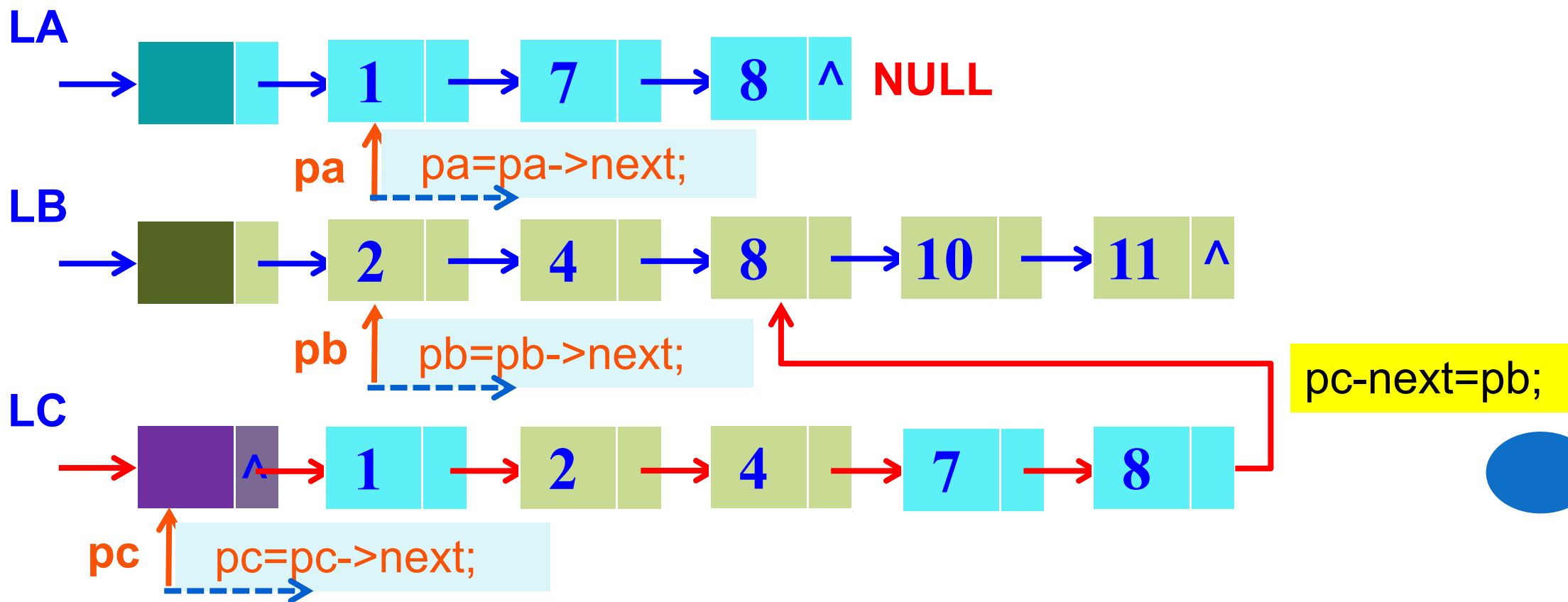
【算法步骤】

(1) 创建一个空表Lc。

(2) 依次从 La 或 Lb 中“摘取”元素值较小的结点插入到 Lc 表的最后，直至其中一个表变空为止。

(3) 继续将 La 或 Lb 其中一个表的剩余结点插入在 Lc 表的最后。

`pa->data <= pb->data;`



课后思考：该算法的时间、空间复杂度是多少？

➤ 2、有序表的合并（以单链表为例）

【算法描述】

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc){  
    pa=La->next; pb=Lb->next;  
    pc=Lc=La;           //用La的头结点作为Lc的头结点  
    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;   //插入剩余段  
    delete Lb;          //释放Lb的头结点  
}
```

小结

- 1、掌握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，掌握链表中的头结点、头指针和首元结点的区别及循环链表、双向链表的特点等。

- 3、熟练掌握顺序表的查找、插入和删除算法。
- 4、熟练掌握链表的查找、插入和删除算法。
- 5、能够从时间和空间复杂度的角度比较两种存储结构的不同特点及其适用场合。

作业

- 1、完成教材P54-56的选择题。
- 2、完成教材P56的算法设计第1题和第6题。（作业本）

