

---

# 第2章 线性表

王迪

wangd@sdas.org

---

- 第2章 线性表
- 第3章 栈和队列
- 第4章 串、数组和广义表

线性结构

(逻辑、存储  
和运算)

### 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:  $(a_1, a_2, \dots, a_n)$

线性结构表达式：  $(a_1, a_2, \dots, a_n)$

## 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

## 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 案例引入

2.3 线性的类型定义

2.4 线性表的顺序表示和实现

2.5 线性表的链式表示和实现

2.6 顺序表和链表的比较

2.7 线性表的应用

2.8 案例分析与实现

# 2.1 线性表的定义和特点



线性表的定义：用数据元素的有限序列表示

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

数据元素

线性起点

$a_i$ 的直接前趋

$a_i$ 的直接后继

线性终点

下标，是元素的序号，表示元素在表中的位置

$n=0$ 时称为空表

$n$ 为元素总个数，即表长

## 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； 元素间关系是线性

## 例2 分析学生情况登记表

学号	姓名	性别	年龄	班级
041810205	于春梅	女	18	04级计算机1班
041810260	何仕鹏	男	20	04级计算机2班
041810284	王 爽	女	19	04级计算机3班
041810360	王亚武	男	18	04级计算机4班
:	:	:	:	:

数据元素都是记录； 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

## 2.2 案例引入



### 案例2.1：一元多项式的运算

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

线性表  $P = (p_0, p_1, p_2, \dots, p_n)$

$$P(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$$

数组表示

(每一项的指数*i*隐含在其系数 $p_i$ 的序号中)

指数 (下标 <i>i</i> )	0	1	2	3	4
系数 $p[i]$	10	5	-4	3	2



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



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

稀疏多项式

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



## 案例2.2：稀疏多项式的运算

多项式**非零项**的数组表示

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

下标i	0	1	2	3
系数 a[i]	7	3	9	5
指数	0	1	8	17

(b)  $B(x) = 8x + 22x^7 - 9x^8$

下标i	0	1	2
系数 b[i]	8	22	-9
指数	1	7	8

$$P_n(x) = p_1 x^{e_1} + p_2 x^{e_2} + \dots + p_m x^{e_m}$$

线性表  $P = ((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$

● 创建一个**新数组c**

● 分别从头遍历比较a和b的每一项

✓ **指数相同**，对应系数相加，若其和不为零，则在c中增加一个新项

✓ **指数不相同**，则将指数较小的项复制到c中

● 一个多项式已遍历**完毕**时，将另一个剩余项依次复制到c中即可

●顺序存储结构存在问题

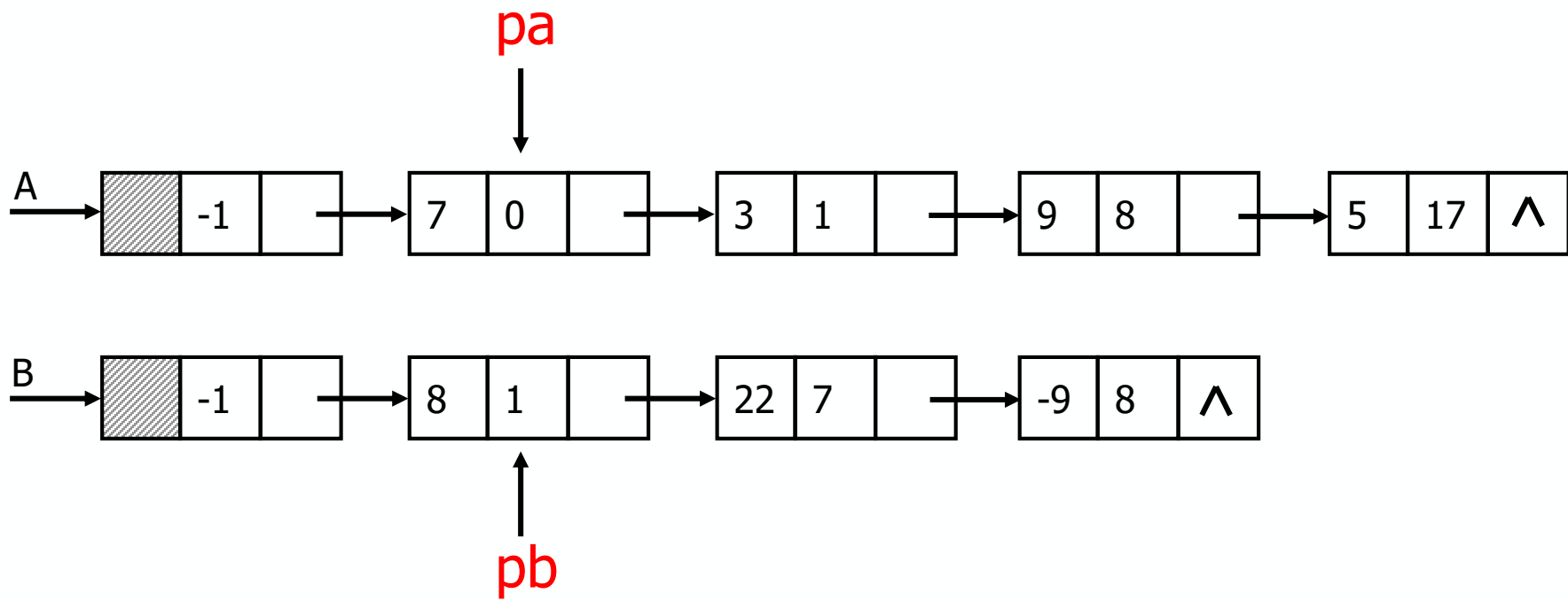
- ✓存储空间分配不灵活
- ✓运算的空间复杂度高



链式存储结构

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

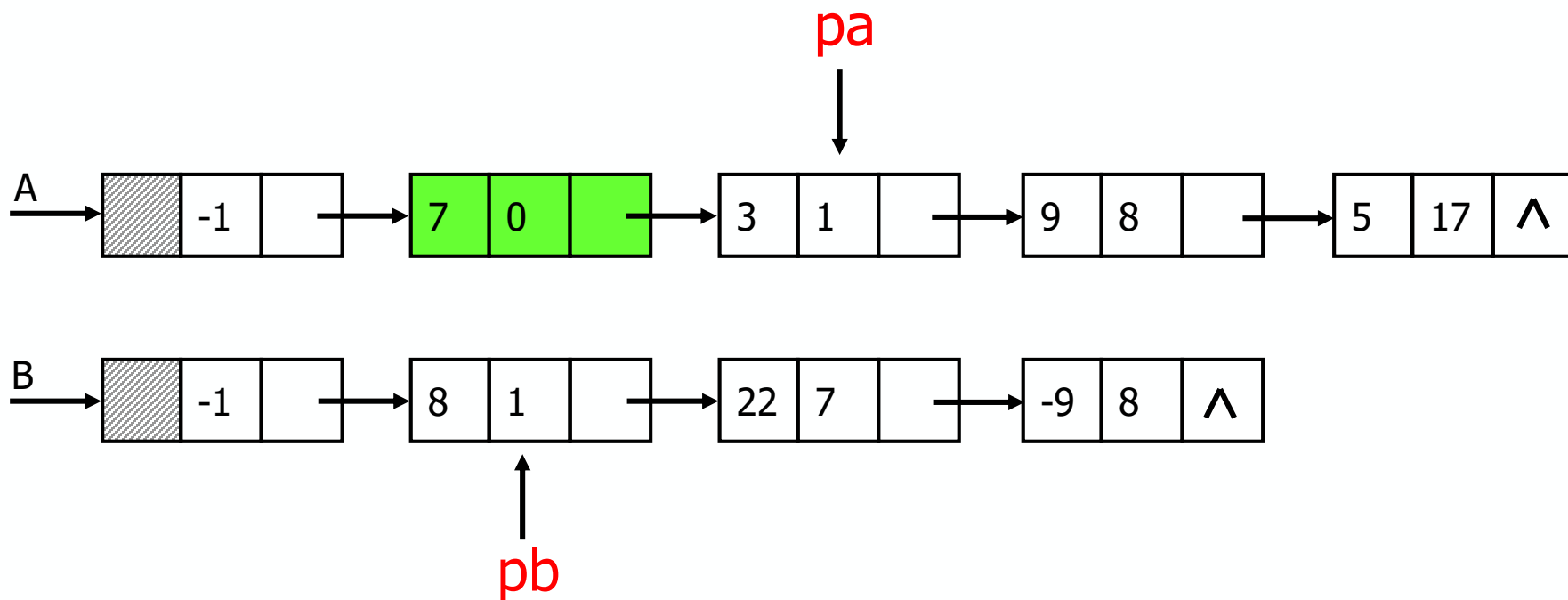
$$B_8(x)=8x+22x^7-9x^8$$



# 多项式相加

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

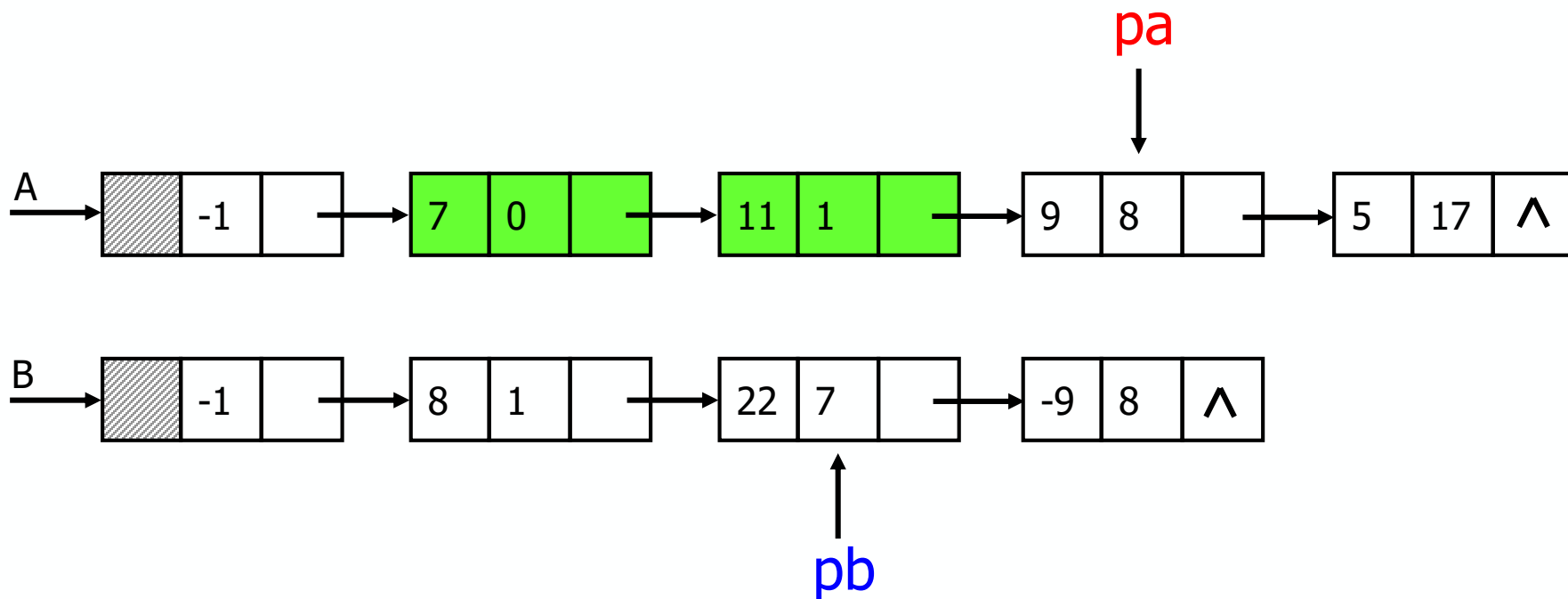
$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加

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

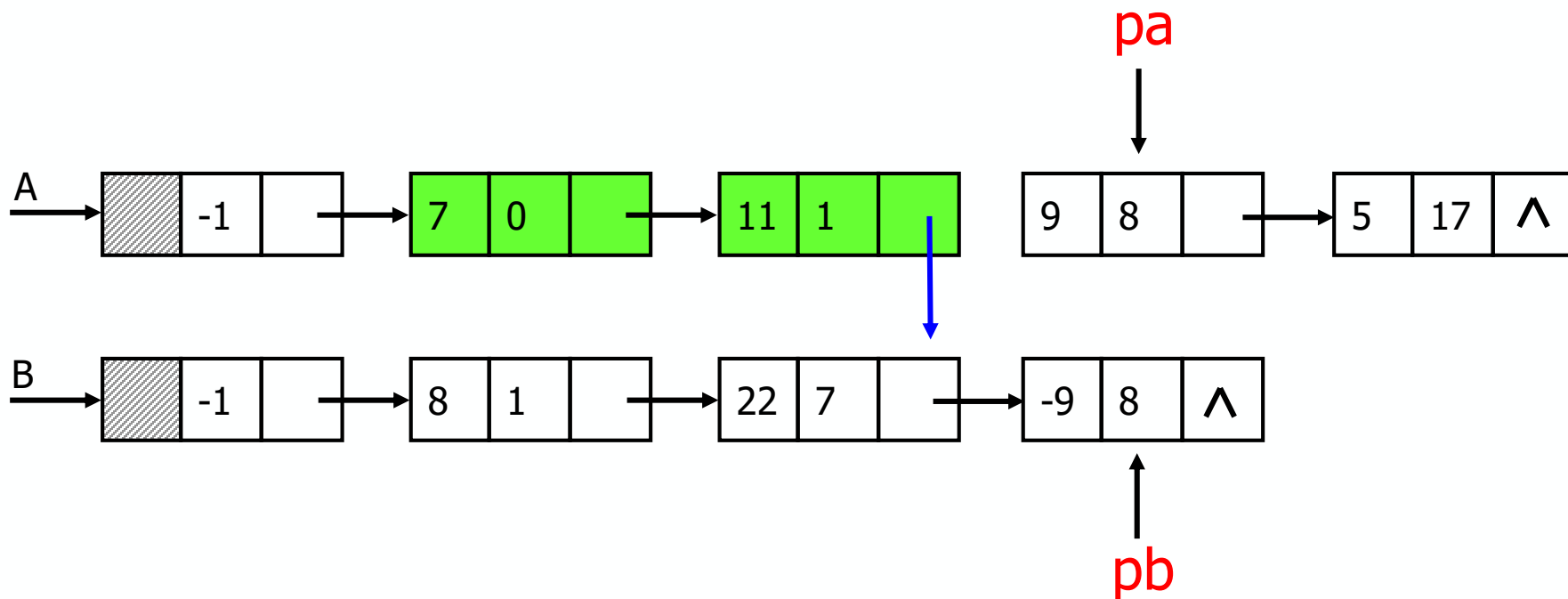
$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加

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

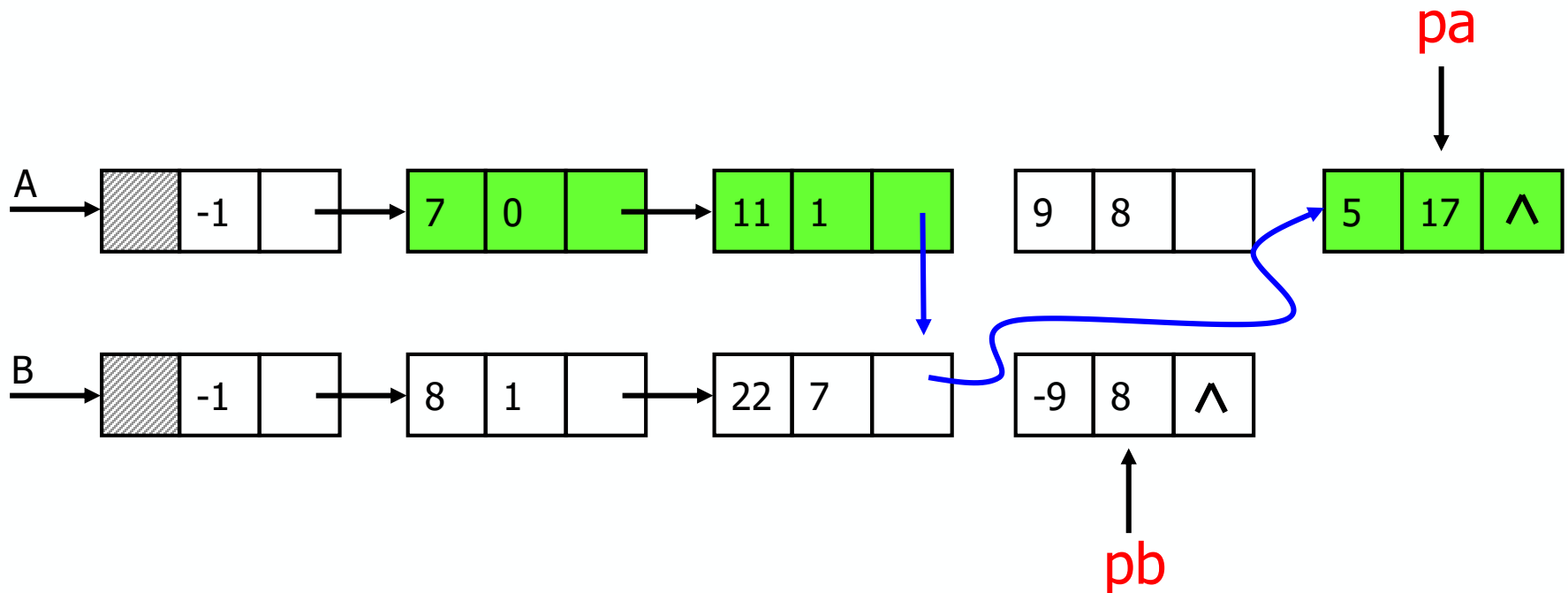
$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加

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

$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 案例2.3：图书信息管理系统

- (1) 查找
- (2) 插入
- (3) 删除
- (4) 修改
- (5) 排序
- (6) 计数

book.txt - 记事本		
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)		
ISBN	书名	定价
9787302257646	程序设计基础	25
9787302219972	单片机技术及应用	32
9787302203513	编译原理	46
9787811234923	汇编语言程序设计教程	21
9787512100831	计算机操作系统	17
9787302265436	计算机导论实验指导	18
9787302180630	实用数据结构	29
9787302225065	数据结构（C语言版）	38
9787302171676	C#面向对象程序设计	39
9787302250692	C语言程序设计	42
9787302150664	数据库原理	35
9787302260806	Java编程与实践	56
9787302252887	Java程序设计与应用教程	39
9787302198505	嵌入式操作系统及编程	25
9787302169666	软件测试	24
9787811231557	Eclipse基础与应用	35

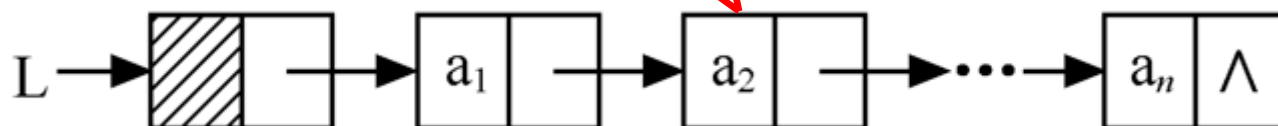


## 图书顺序表

elem[0]	elem[1]	elem[2]	...	elem[length-1]	空闲区	
$a_1$	$a_2$	$a_3$	...	$a_{\text{length}}$		

ISBN	书名	价格
------	----	----

## 图书链表





- 线性表中数据元素的类型可以为简单类型，也可以为复杂类型。
- 许多实际应用问题所涉的基本操作有很大相似性，不应为每个具体应用单独编写一个程序。
- 从具体应用中抽象出共性的逻辑结构和基本操作（抽象数据类型），然后实现其存储结构和基本操作。

## 2.3 线性表的类型定义



### 线性表的重要基本操作

1. 初始化
2. 取值
3. 查找
4. 插入
5. 删除

## 2.4 线性表的顺序表示和实现



线性表的顺序表示又称为顺序存储结构或顺序映像。

**顺序存储定义：**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。

## 顺序存储

存储地址	存储内容
$L_0$	元素1
$L_0+m$	元素2
	.....
$L_0+(i-1)*m$	元素i
	.....
$L_0+(n-1)*m$	元素n

$$\text{Loc(元素i)} = L_0 + (i-1)*m$$

# 顺序表的类型定义

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

# 图书表的顺序存储结构类型定义

```
#define MAXSIZE 10000    //图书表可能达到的最大长度
typedef struct           //图书信息定义
{
    char no[20];          //图书ISBN
    char name[50];        //图书名字
    float price;          //图书价格
}Book;
typedef struct
{
    Book *elem;           //存储空间的基地址
    int length;           //图书表中当前图书个数
}SqList;                 //图书表的顺序存储结构类型为SqList
```

# 线性表的重要基本操作

1. 初始化
2. 取值
3. 查找
4. 插入
5. 删除



## 1. 初始化线性表L （参数用引用）

```
Status InitList_Sq(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;  
}
```

# 补充：几个简单基本操作的算法实现

## 销毁线性表L

```
void DestroyList(SqList &L)
{
    if (L.elem) delete[] L.elem;    //释放存储空间
}
```

## 清空线性表L

```
void ClearList(SqList &L)
{
    L.length=0;    //将线性表的长度置为0
}
```

# 补充：几个简单基本操作的算法实现

## 求线性表L的长度

```
int GetLength(SqList L)
{
    return (L.length);
}
```

## 判断线性表L是否为空

```
int IsEmpty(SqList L)
{
    if (L.length==0) return 1;
    else return 0;
}
```

# 线性表的重要基本操作

1. 初始化
2. 取值
3. 查找
4. 插入
5. 删除

## 2. 取值 (根据位置i获取相应位置数据元素的内容)

获取线性表L中的某个数据元素的内容

```
int 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
```

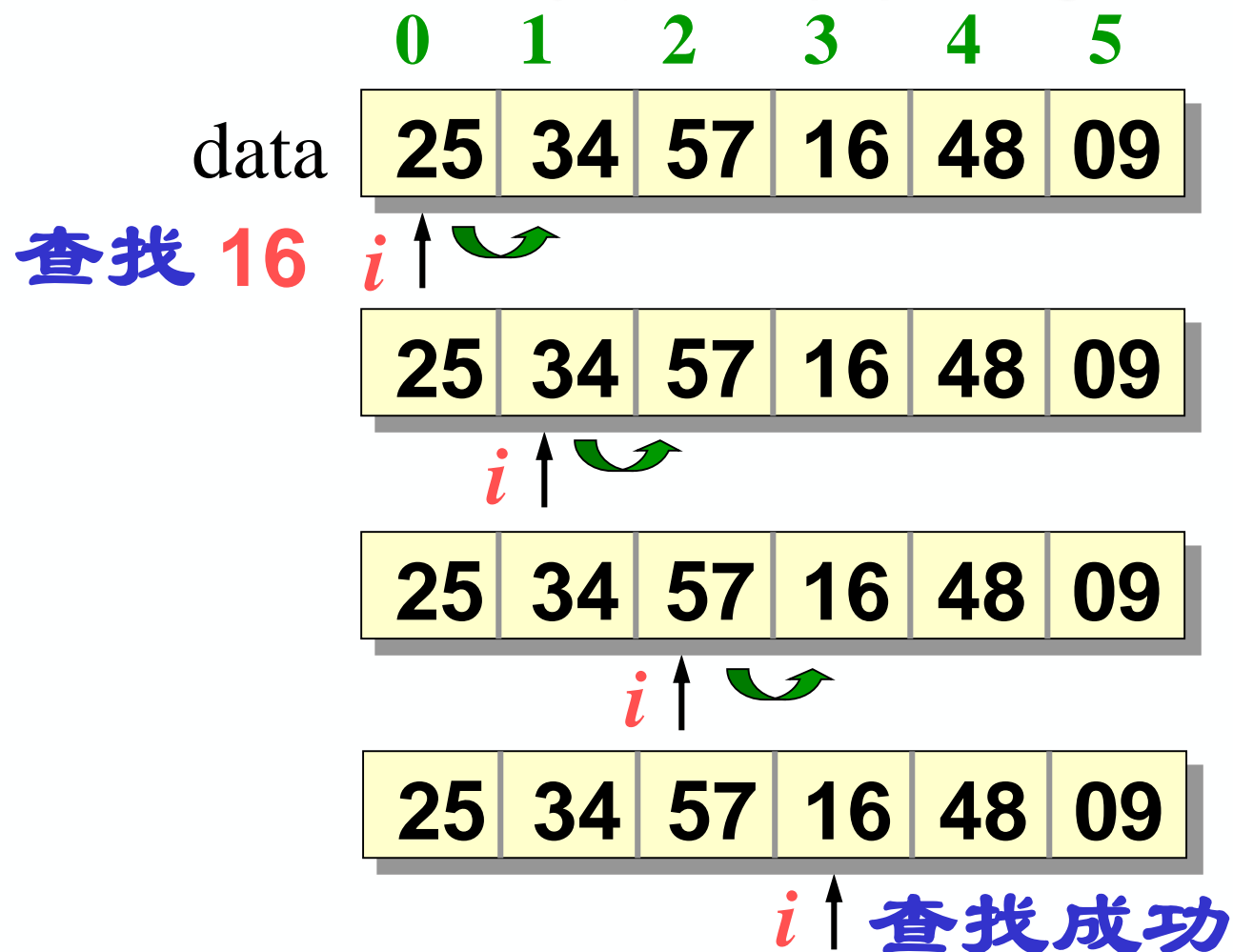
```
    return OK;
```

```
}
```

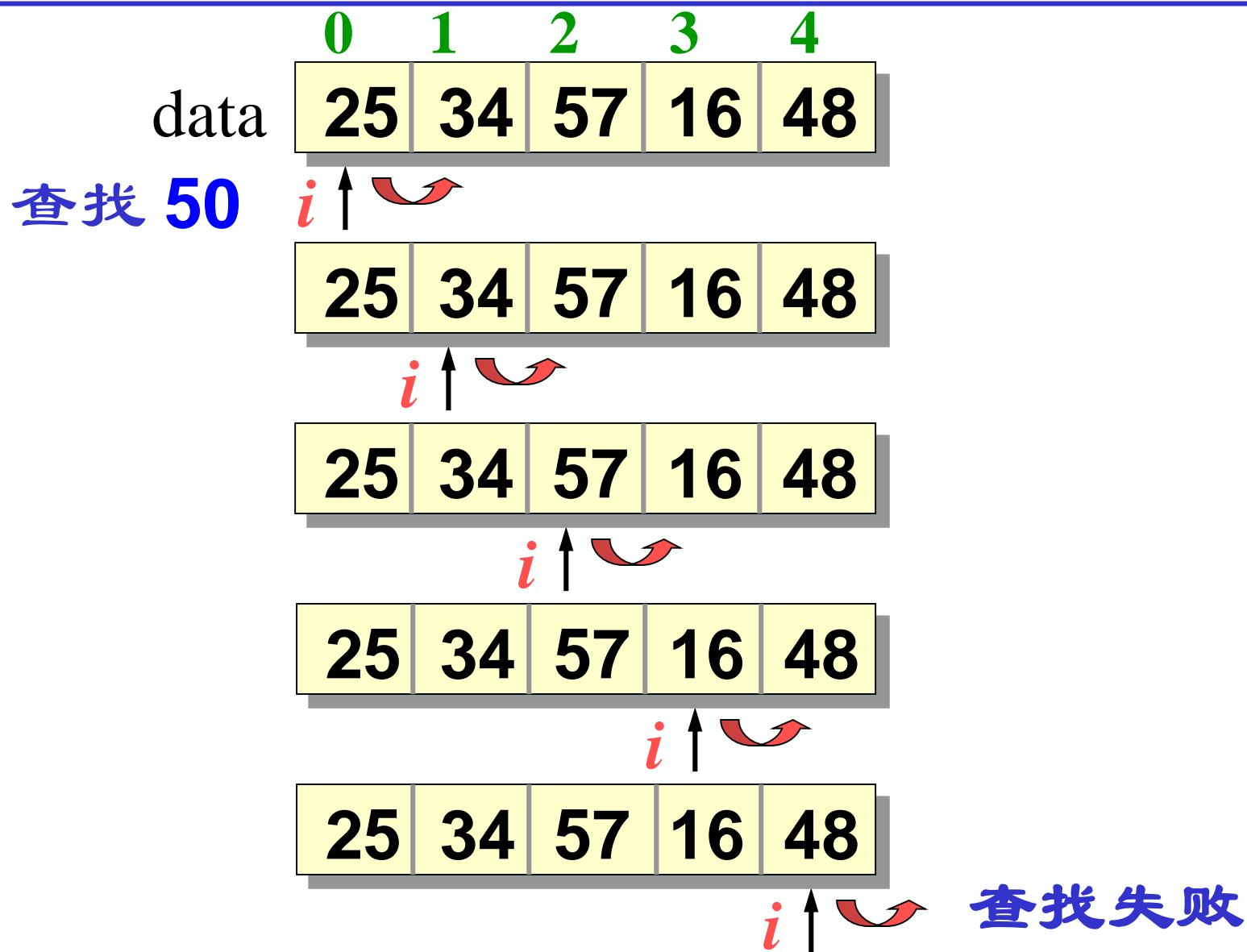
随机存取

### 3. 查找（根据指定数据获取数据所在的位置）

#### 顺序查找图示



### 3. 查找 (根据指定数据获取数据所在的位置)



### 3. 查找 (根据指定数据获取数据所在的位置)

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;
}
```

查找算法时间效率分析 ? ? ?



# 【算法分析】

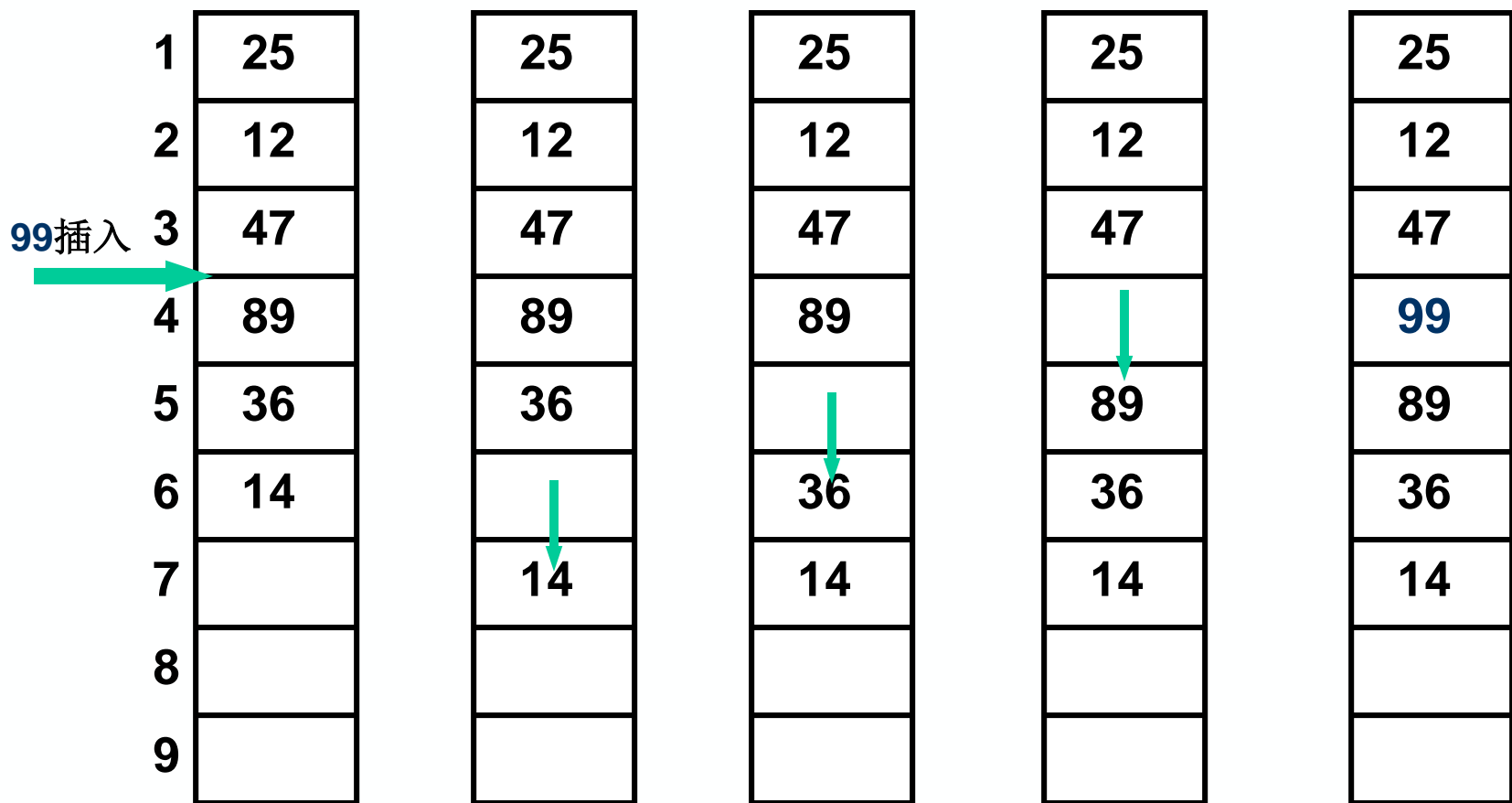
## 平均查找长度

**ASL (Average Search Length) :**

为确定记录在表中的位置，需要给定值进行比较的关键字的个数的期望值叫做查找算法的平均查找长度。  
对含有n个记录的表，查找成功时：

$$ASL = \sum_{i=1}^n P_i C_i$$

## 4. 插入（插在第 $i$ 个结点之前）



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

# 【算法步骤】

- (1) 判断插入位置 $i$  是否合法。
- (2) 判断顺序表的存储空间是否已满。
- (3) 将第 $n$ 至第 $i$  位的元素依次向后移动一个位置，空出第 $i$ 个位置。
- (4) 将要插入的新元素 $e$ 放入第 $i$ 个位置。
- (5) 表长加1，插入成功返回OK。

# 【算法描述】

4. 在线性表L中第i个数据元素之前插入数据元素e

```
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;  
}
```

# 【算法分析】

算法时间主要耗费在移动元素的操作上

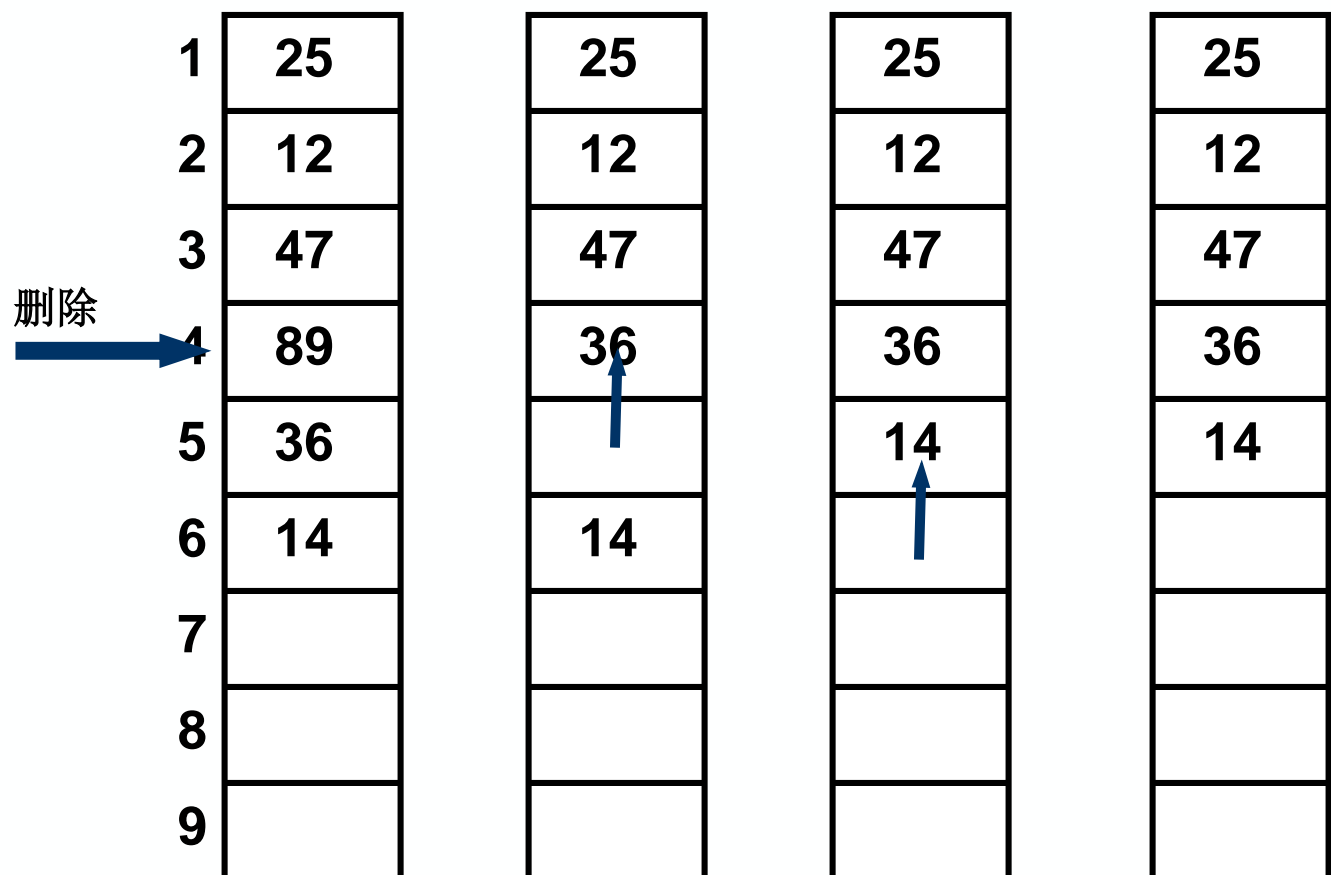
若插入在尾结点之后，则根本无需移动（特别快）；

若插入在首结点之前，则表中元素全部后移（特别慢）；

若要考虑在各种位置插入（共 $n+1$ 种可能）的平均移动次数，该如何计算？

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

# 5. 删除（删除第 i 个结点）



删除第 4 个结点，移动  $6-4$  次

删除第  $i$  个结点，移动  $n-i$  次

# 【算法步骤】

- (1) 判断删除位置 $i$  是否合法（合法值为 $1 \leq i \leq n$ ）。
- (2) 将欲删除的元素保留在 $e$ 中。
- (3) 将第 $i+1$ 至第 $n$  位的元素依次向前移动一个位置。
- (4) 表长减1，删除成功返回OK。

# 【算法描述】

## 5. 将线性表L中第i个数据元素删除

```
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;  
}
```



# 【算法分析】

算法时间主要耗费在移动元素的操作上

若删除尾结点，则根本无需移动（特别快）；

若删除首结点，则表中 $n-1$ 个元素全部前移（特别慢）；

若要考虑在各种位置删除（共 $n$ 种可能）的平均移动次数，该如何计算？

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

---

查找、插入、删除算法的平均时间复杂度为  
 $O(n)$

显然，顺序表的空间复杂度 $S(n)=O(1)$   
(没有占用辅助空间)

# 顺序表（顺序存储结构）的特点

- (1) 利用数据元素的存储位置表示线性表中相邻数据元素之间的前后关系，即线性表的**逻辑结构与存储结构一致**
- (2) 在访问线性表时，可以快速地计算出任何一个数据元素的存储地址。因此可以粗略地认为，**访问每个元素所花时间相等**

这种存取元素的方法被称为**随机存取法**

# 顺序表的优缺点

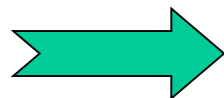
## 优点:

- ✓ **存储密度大** (结点本身所占存储量/结点结构所占存储量)
- ✓ 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
- ✓ 浪费存储空间
- ✓ 属于静态存储形式, 数据元素的个数不能自由扩充

为克服这一缺点



**链表**

## 2.5 线性表的链式表示和实现



### 链式存储结构

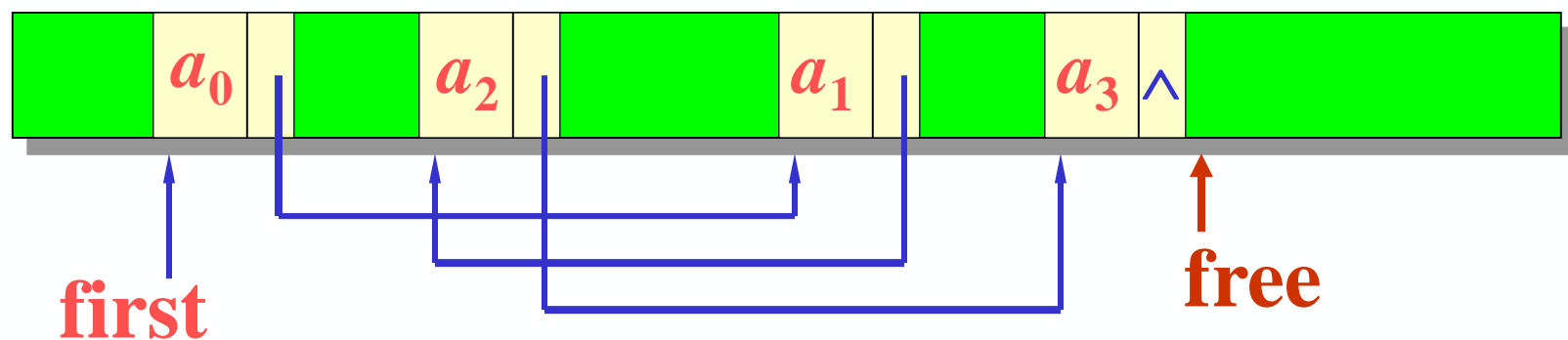
结点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻

线性表的链式表示又称为非顺序映像或链式映像。

如何实现？

通过**指针**来实现

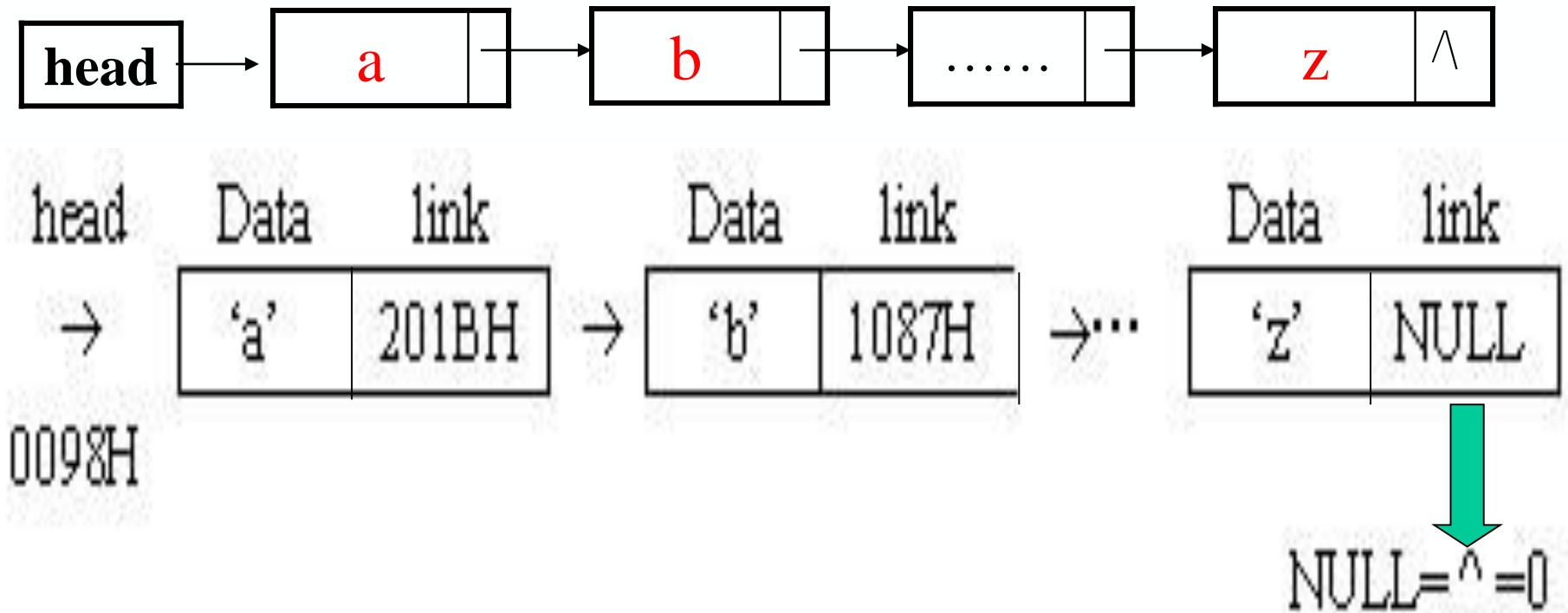
## 单链表的存储映像

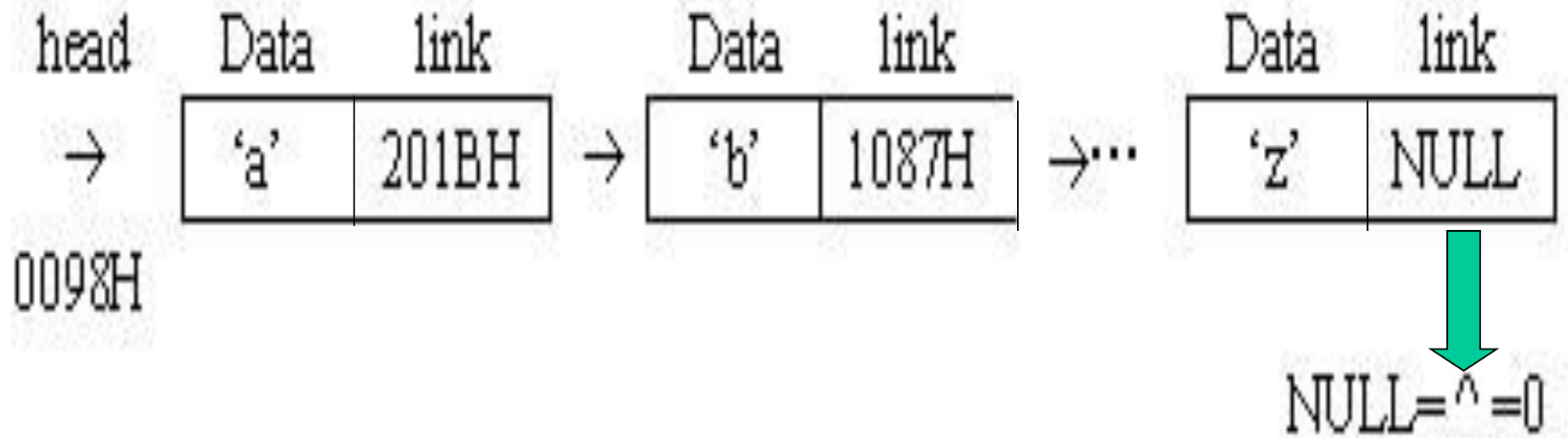


# 例 画出26个英文字母表的链式存储结构

逻辑结构: ( a, b, ..., y, z )

链式存储结构:





各结点由两个域组成：

数据	指针
----	----

**数据域：** 存储元素数值数据

**指针域：** 存储直接后继结点的存储位置



# 与链式存储有关的术语

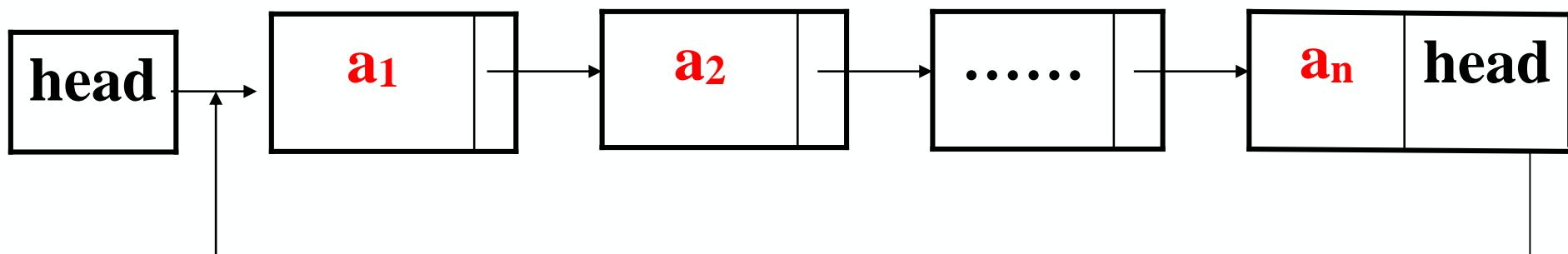
---

- 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

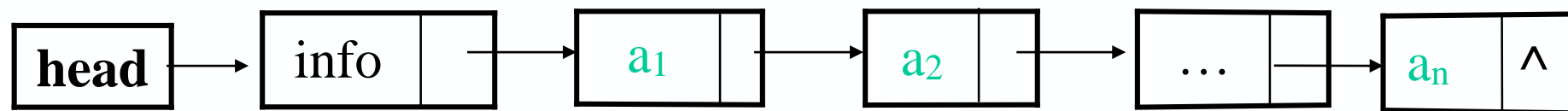
### 3、单链表、双链表、循环链表：

- 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**
- 首尾相接的链表称为**循环链表**

循环链表示意图：



## 4、头指针、头结点和首元结点



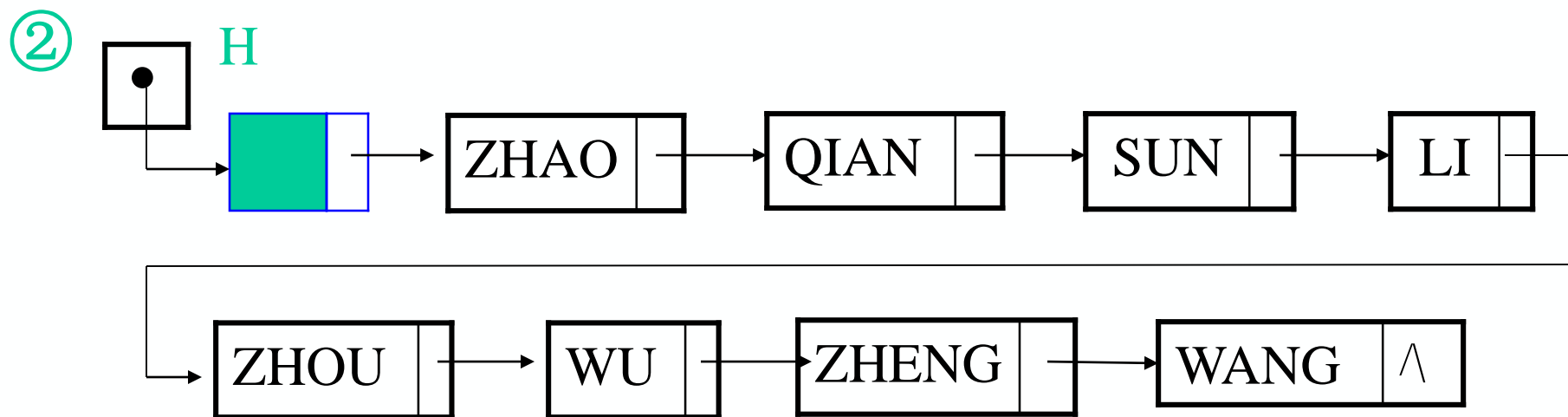
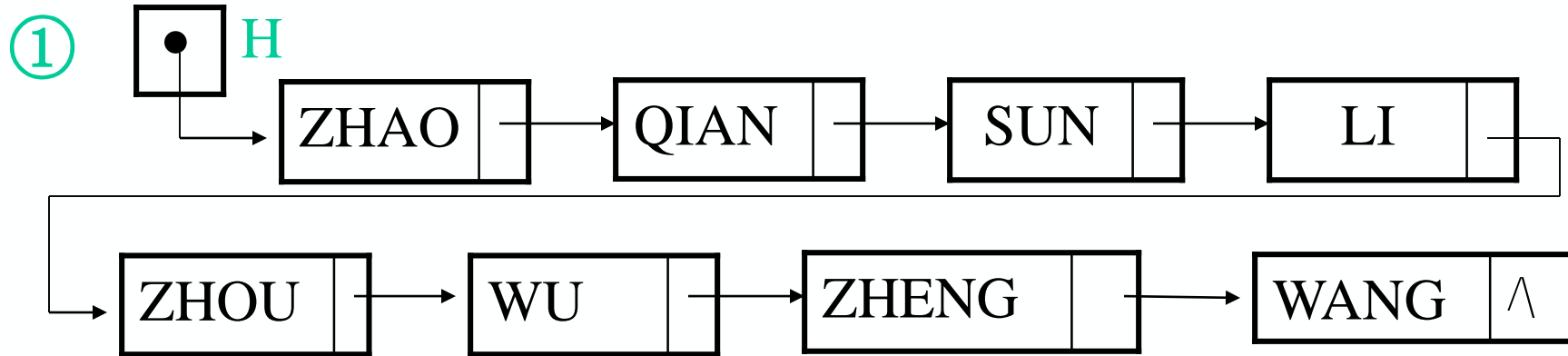
头指针 头结点 首元结点

头指针是指向链表中第一个结点的指针

首元结点是指链表中存储第一个数据元素 $a_1$ 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

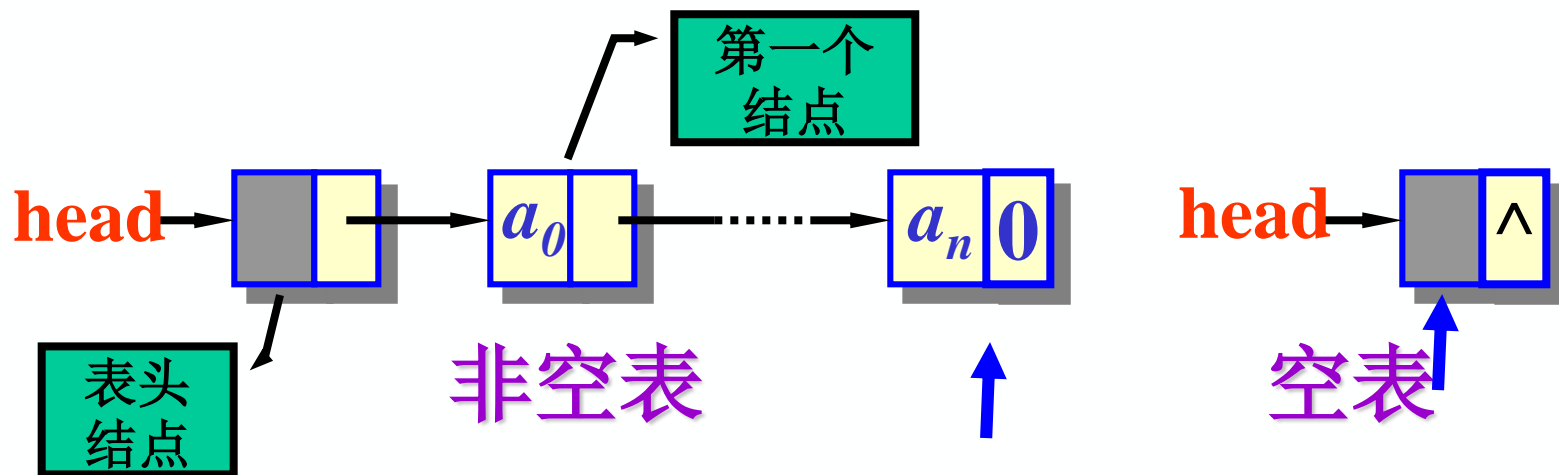
上例链表的逻辑结构示意图有以下两种形式：



区别：① 无头结点    ② 有头结点

## 讨论1. 如何表示空表?

有头结点时, 当头结点的指针域为空时表示空表



## 讨论2. 在链表中设置头结点有什么好处？

### 1. 便于首元结点的处理

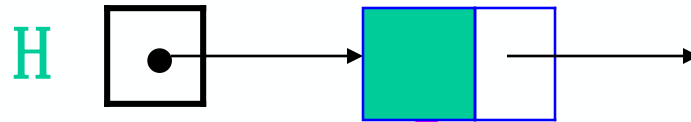
首元结点的地址保存在头结点的指针域中，所以在链表的第一个位置上的操作和其它位置一致，无须进行特殊处理；

### 2. 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。

### 讨论3. 头结点的**数据域**内装的是什么？

头结点的**数据域**可以为空，也可存放线性表**长度**等附加信息，但此结点不能计入链表长度值。



头结点的数据域

# 链表（链式存储结构）的特点

- （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



# 链表的优缺点

---

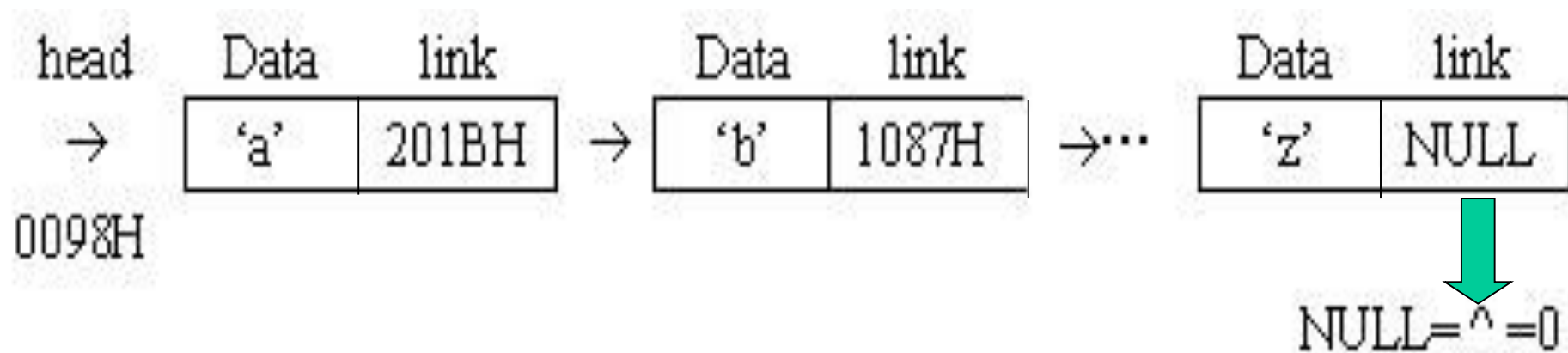
## 优点

- 数据元素的个数可以自由扩充
- 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


# 链表的优缺点

## 缺点

- 存储密度小
- 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**）



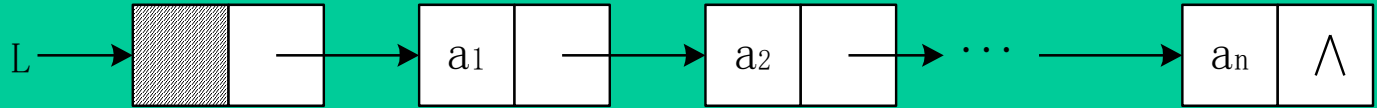
# 练习

- 
1. 链表的每个结点中都恰好包含一个指针。
  2. 顺序表结构适宜于进行顺序存取，而链表适宜于进行随机存取。
  3. 顺序存储方式的优点是存储密度大，且插入、删除运算效率高。
  4. 线性表若采用链式存储时，结点之间和结点内部的存储空间都是可以不连续的。
  5. 线性表的每个结点只能是一个简单类型，而链表的每个结点可以是一个复杂类型

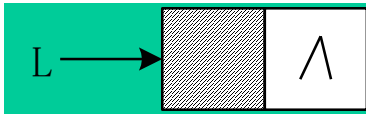
## 2.5.1 单链表的定义和实现



非空表



空表



- ✓ 单链表是由表头唯一确定，因此单链表可以用头指针的名字来命名
- ✓ 若头指针名是L，则把链表称为表L

# 单链表的存储结构定义

```
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**

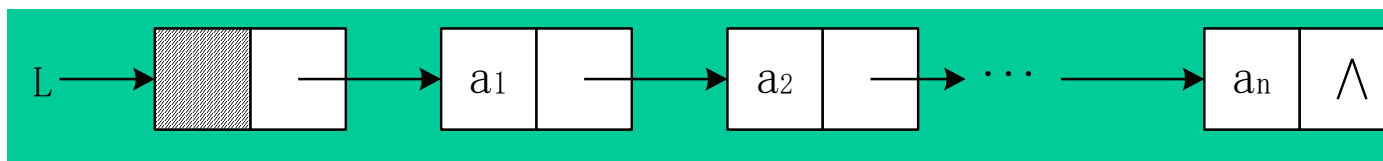


**LinkList p**

## 注意区分指针变量和结点变量两个不同的概念

- 指针变量p: 表示结点地址
- 结点变量\*p: 表示一个结点

# LNNode \*p



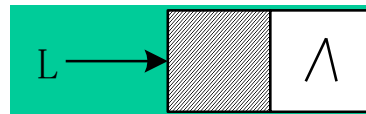
若  $p \rightarrow data = a_i$ , 则  $p \rightarrow next \rightarrow data = a_{i+1}$

## 2.5.2 单链表基本操作的实现



1. 初始化
2. 取值
3. 查找
4. 插入
5. 删除

## 1. 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;  
}
```



# 补充：几个简单基本操作的算法实现

## 销毁

```
Status DestroyList_L(LinkList &L){  
    LinkList p;  
    while(L)  
    {  
        p=L;  
        L=L->next;  
        delete p;  
    }  
    return OK;  
}
```

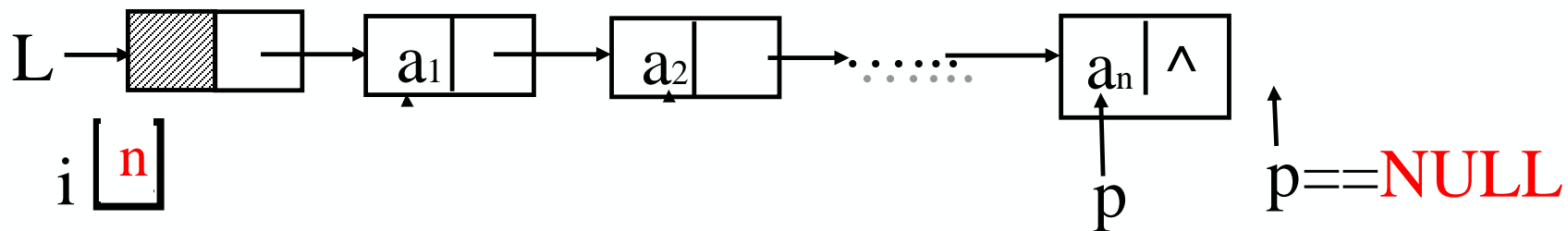
# 补充：几个简单基本操作的算法实现

## 清空

```
Status ClearList(LinkList & L){  
    // 将L重置为空表  
    LinkList p,q;  
    p=L->next; //p指向第一个结点  
    while(p) //没到表尾  
        { q=p->next; delete p; p=q; }  
    L->next=NULL; //头结点指针域为空  
    return OK;  
}
```

# 补充：几个简单基本操作的实现

## 求表长



## “数”结点：

- 指针  $p$  依次指向各个结点
- 从第一个元素开始“数”
- 一直“数”到最后一个结点

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

```
i=0;
```

```
while(p){i++;p=p->next;}
```

## 求表长

```
int ListLength_L(LinkList L){
```

```
//返回L中数据元素个数
```

```
    LinkList p;
```

```
    p=L->next; //p指向第一个结点
```

```
    i=0;
```

```
    while(p){//遍历单链表, 统计结点数
```

```
        i++;
```

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

```
    return i;
```

```
}
```

## “数”结点:

- 指针p依次指向各个结点
- 从第一个元素开始“数”
- 一直“数”到最后一个结点

---

## 判断表是否为空

```
int ListEmpty(LinkList L){  
//若L为空表，则返回1，否则返回0  
    if(L->next) //非空  
        return 0;  
    else  
        return 1;  
}
```

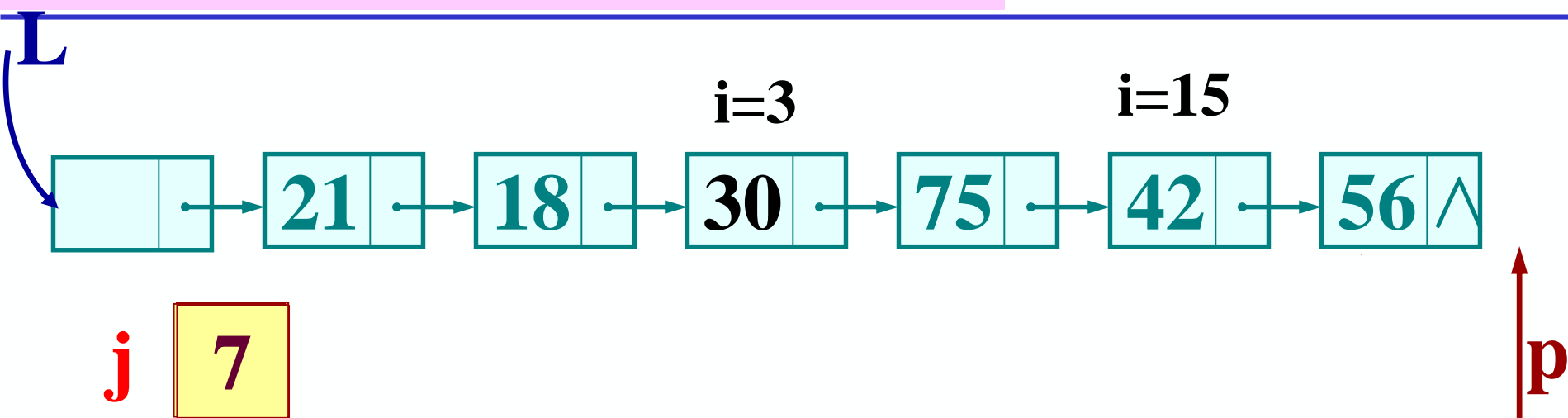
# 线性表的重要基本操作

1. 初始化
2. 取值
3. 查找
4. 插入
5. 删除

## 2. 取值 (根据位置*i*获取相应位置数据元素的内容)

- 思考：顺序表里如何找到第*i*个元素？
- 链表的查找：要从链表的头指针出发，顺着链域next逐个结点往下搜索，直至搜索到第*i*个结点为止。因此，链表不是随机存取结构

例：分别取出表中 $i=3$ 和 $i=15$ 的元素



## 【算法步骤】

- ✓从第1个结点 ( $L \rightarrow \text{next}$ ) 顺链扫描，用指针 $p$ 指向当前扫描到的结点， $p$ 初值 $p = L \rightarrow \text{next}$ 。
- ✓ $j$ 做计数器，累计当前扫描过的结点数， $j$ 初值为1。
- ✓当 $p$ 指向扫描到的下一结点时，计数器 $j$ 加1。
- ✓当 $j = i$ 时， $p$ 所指的结点就是要找的第 $i$ 个结点。



## 2. 取值 (根据位置i获取相应位置数据元素的内容)

//获取线性表L中的某个数据元素的内容

```
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;
```

```
    }
```

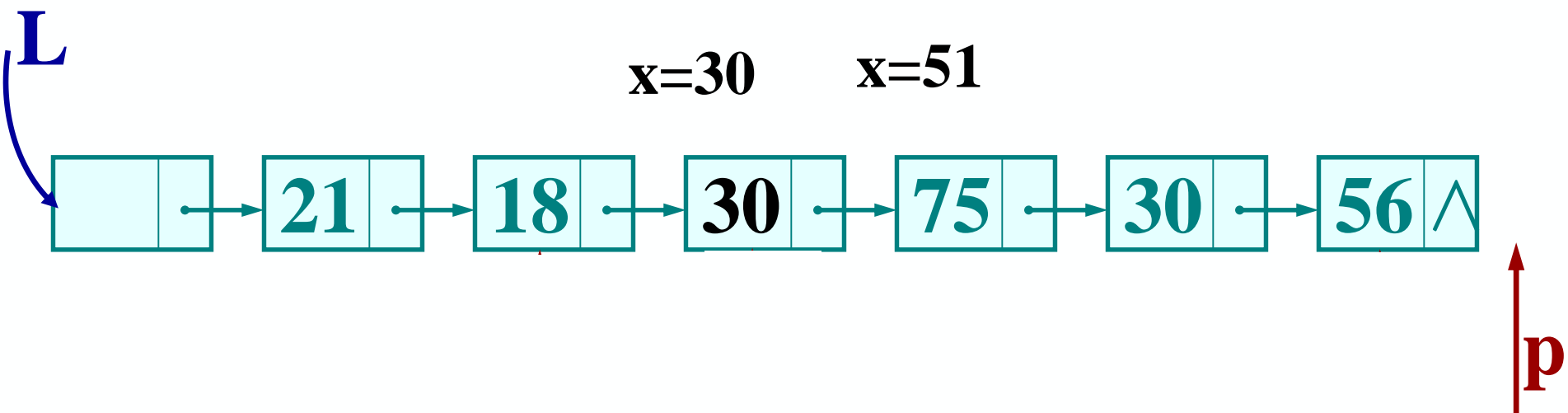
```
    if(!p || j>i) return ERROR; //第i个元素不存在
```

```
    e=p->data; //取第i个元素
```

```
    return OK;
```

```
}//GetElem_L
```

### 3. 查找 (根据指定数据获取数据所在的位置)



**j** **7** 未找到，返回0

- ✓从第一个结点起，依次和e相比较。
- ✓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“位置”或地址；
- ✓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”。

# 【算法描述】

//在线性表L中查找值为e的数据元素

```
LNode *LocateELem_L (LinkList L, Elemtyp 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;  
}
```

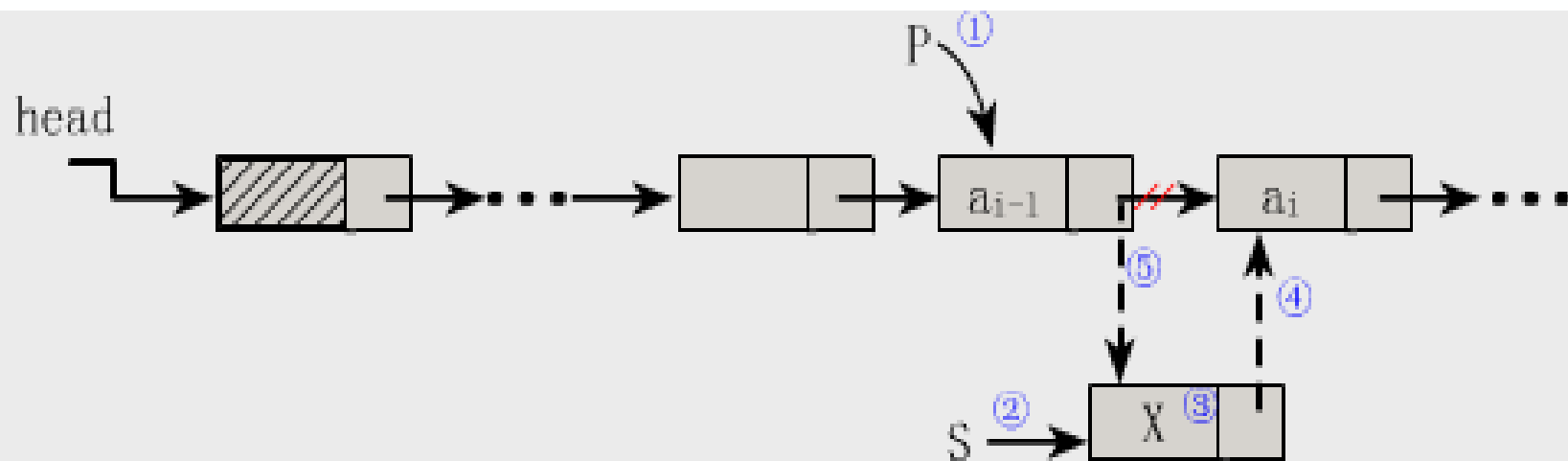
# 【算法描述】

//在线性表L中查找值为e的数据元素

```
int LocateELem_L (LinkList L, Elemtype e) {  
    //返回L中值为e的数据元素的位置序号，查找失败返回0  
    p=L->next; j=1;  
    while(p && p->data!=e)  
        {p=p->next; j++;}  
    if(p) return j;  
    else return 0;  
}
```

## 4. 插入（插在第 $i$ 个结点之前）

- 将值为 $x$ 的新结点插入到表的第 $i$ 个结点的位置上，即插入到 $a_{i-1}$ 与 $a_i$ 之间



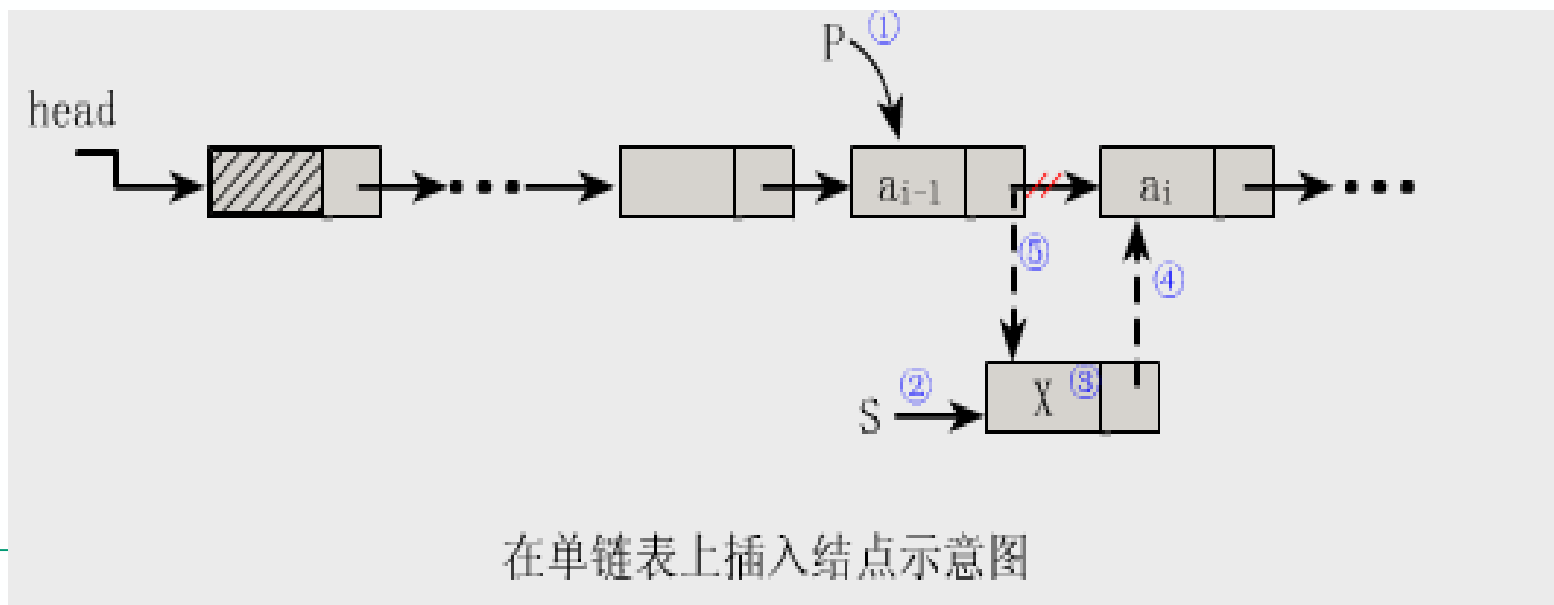
在单链表上插入结点示意图

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

思考：步骤1和2能互换么？

# 【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 $*s$



# 【算法描述】

//在L中第i个元素之前插入数据元素e

```
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;
```

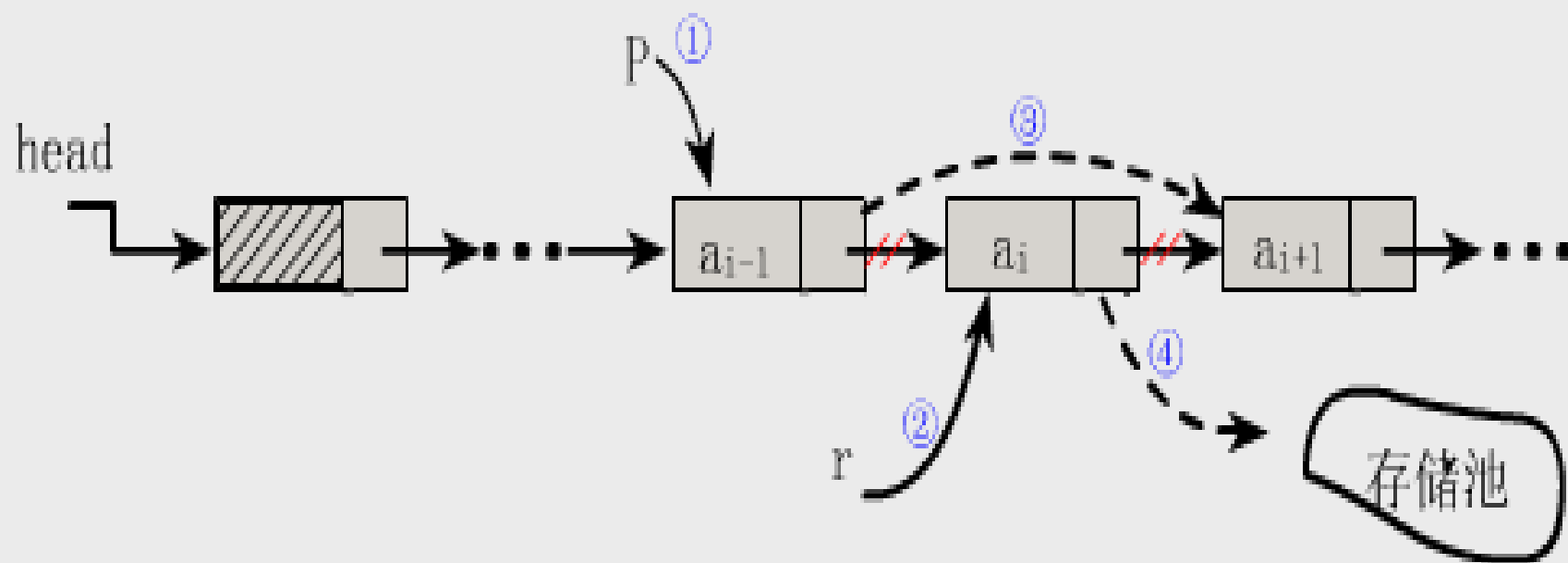
```
}//ListInsert_L
```

## 5. 删除（删除第 $i$ 个结点）

- 将表的第 $i$ 个结点删去
- 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

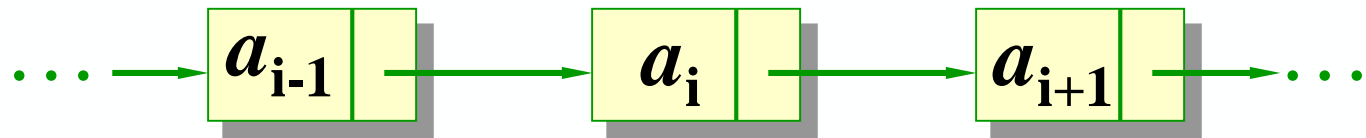


## 5. 删除（删除第 $i$ 个结点）

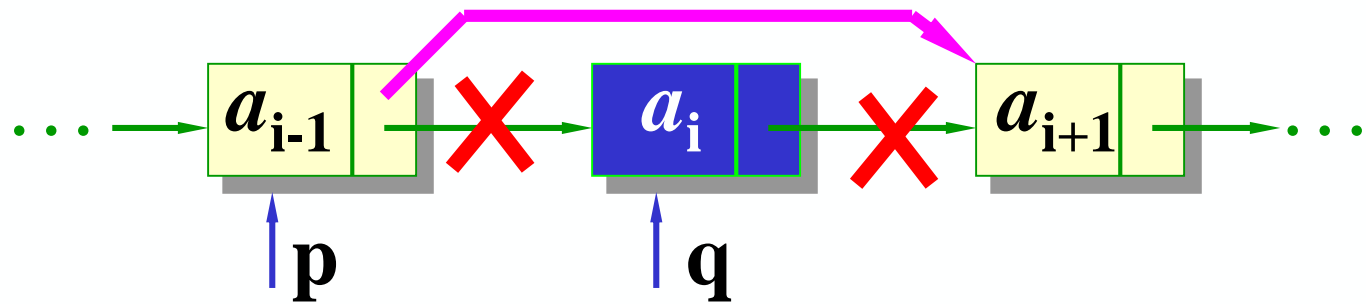


在单链表上删除结点示意图

## 5. 删除 (删除第 $i$ 个结点)



删除前

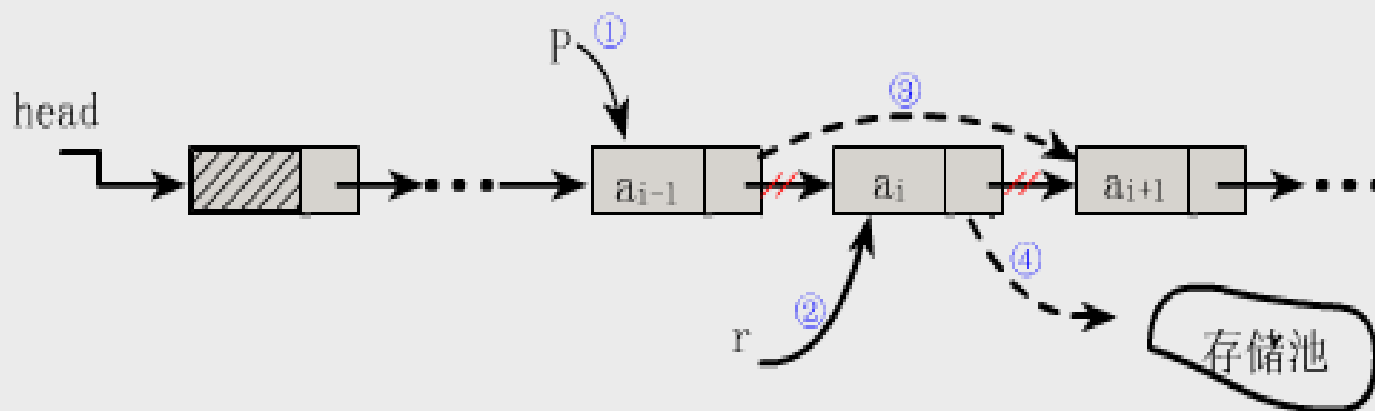


删除后

$p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next} \quad ???$

# 【算法步骤】

- (1) 找到 $a_{i-1}$ 存储位置 $p$
- (2) 临时保存结点 $a_i$ 的地址在 $q$ 中，以备释放
- (3) 令 $p \rightarrow \text{next}$ 指向 $a_i$ 的直接后继结点
- (4) 将 $a_i$ 的值保留在 $e$ 中
- (5) 释放 $a_i$ 的空间



在单链表上删除结点示意图

# 【算法描述】

//将线性表L中第i个数据元素删除

```
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  
    delete q;    //释放删除结点的空间  
    return OK;  
}  
//ListDelete_L
```

# 链表的运算时间效率分析

1. **查找**：因线性链表只能顺序存取，即在查找时要从头指针找起，查找的时间复杂度为  $O(n)$ 。
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  $O(1)$ 。

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  $O(n)$ 。