第三章 线性表、堆栈和队列

- 3.1 线性表的定义和基本操作
- 3.2 线性表的顺序存储结构
- 3.3 线性表的链接存储结构
- 3.4 复杂性分析
- 3.5 堆栈
- 3.6 队列

3.1 线性表的定义和基本操作

1. 线性表的定义

[例1] 英文字母表 （A, B, C, ……, Z）

整数序列 （1, 78, 9, 12, 10)

[例2] 某班学生健康情况登记表。

学号	姓名	性别	年龄	健康情况
01	张军	男	18	一般
02	陈红	女	17	良好
03	陈军	男	19	神经衰弱
…	…	…	…	…

考虑：这些线性表中的数据元素是什么？

表中每个数据元素由什么域组成？

线性表的定义：一个线性表是由零个或多个具有相同类型的结点组成的有序集合。通常用$(a_1, a_2, \ldots, a_n)$表示一个线性表，$n \geq 0$，a_k表示结点，$1 \leq k \leq n$；

当$n=0$时，线性表中无结点，被称为空表；

当$n>1$时，称a_1为线性表的头结点(简称表头)，称a_n为线性表的尾结点(简称表尾)，a_i为a_{i+1}的前驱结点，a_{i+1}为a_i的后继结点，其中$1 \leq i < n$；

当$n=1$时，线性表中仅有一个结点a_1，a_1既是表头又是表尾。

线性表的逻辑结构：线性结构

线性表记为
$$\begin{cases} (a_1, a_2, \ldots, a_n) & n>0 \\ (\quad) \text{ 空表} & n=0 \end{cases}$$

线性表的基本操作

(1) 创建一个线性表;

(2) 确定线性表的长度;

(3) 判断线性表是否为空;

(4) 存取线性表中第k个结点的字段值;

(5) 查找指定字段值在线性表中的位置;

(6) 删除线性表中第k个结点;

(7) 在线性表中第k个结点后插入一个新结点;

(8) 归并、分拆、复制、排序.....

线性表的存储结构

- 线性表存储结构
  - 顺序存储
  - 链接存储
    - 单向链表
    - 循环链表
    - 双向链表
    - 双向循环链表

第三章 线性表、堆栈和队列

- 3.1 线性表的定义和基本操作
- 3.2 线性表的顺序存储结构
- 3.3 线性表的链接存储结构
- 3.4 复杂性分析
- 3.5 堆栈
- 3.6 队列

3.2 线性表的顺序存储结构

顺序存储：按逻辑顺序将线性表的结点依次存放在一组地址连续的存储单元中。

顺序存储的线性表也被称为顺序表。

顺序表的特点：逻辑顺序与物理顺序相同。

包含4个结点的线性表A[4]在内存中的表示，其中每个结点占2个连续的字节，第一个结点A[1]的首地址为302

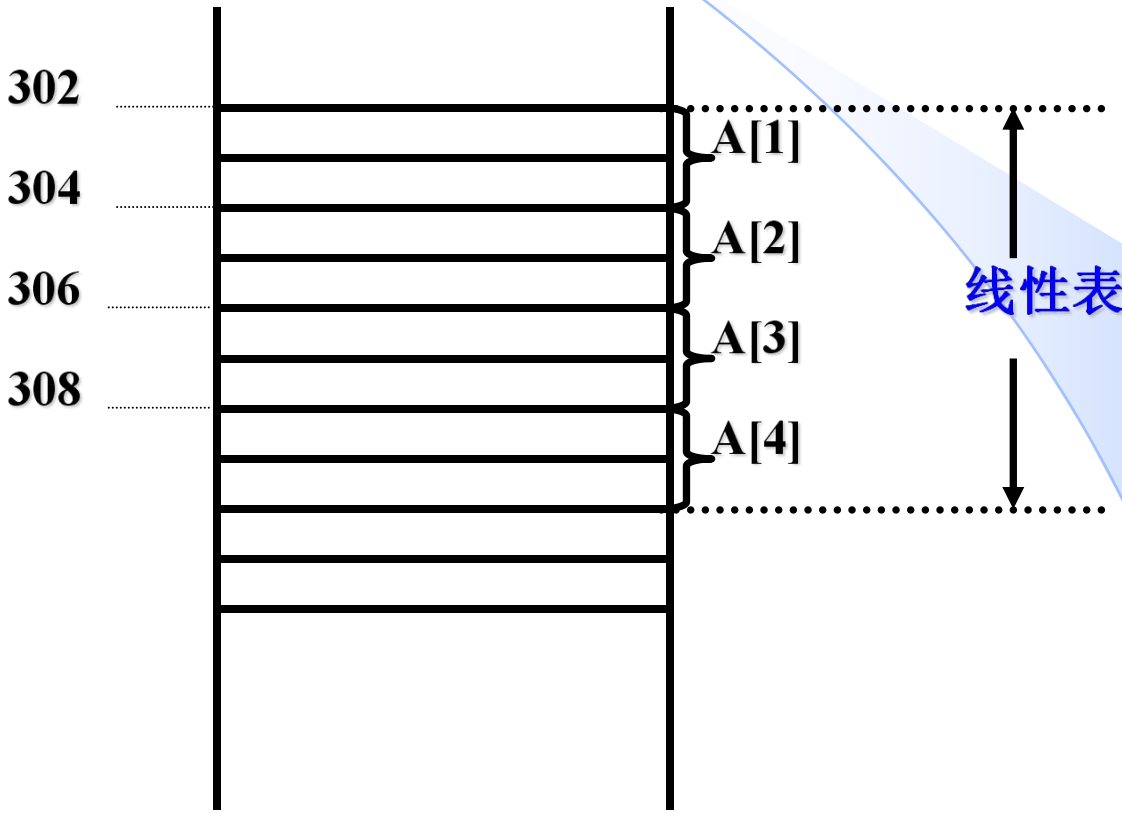
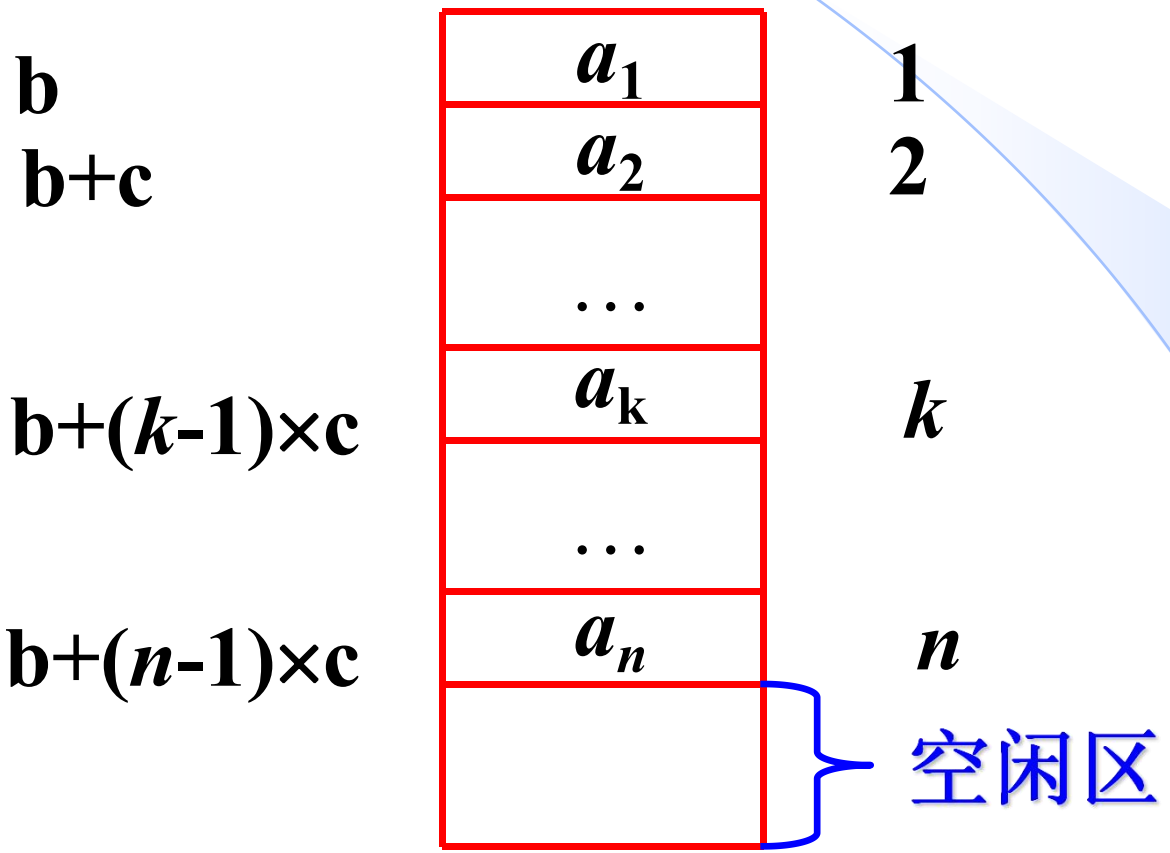


图3.1 线性表的顺序存储

实现顺序存储的最有效方法是使用一维数组。例如：线性表$(a_1, a_2, \ldots, a_n)$, 可以使用数组$a[n]$来存放。



$$Loc(a[k]) = Loc(a[1]) + (k-1)\times c$$

顺序表中一些变量说明：

用于顺序存储线性表的一维数组A；

用于表示线性表当前长度的length；

用于表示最大长度的MaxSize；

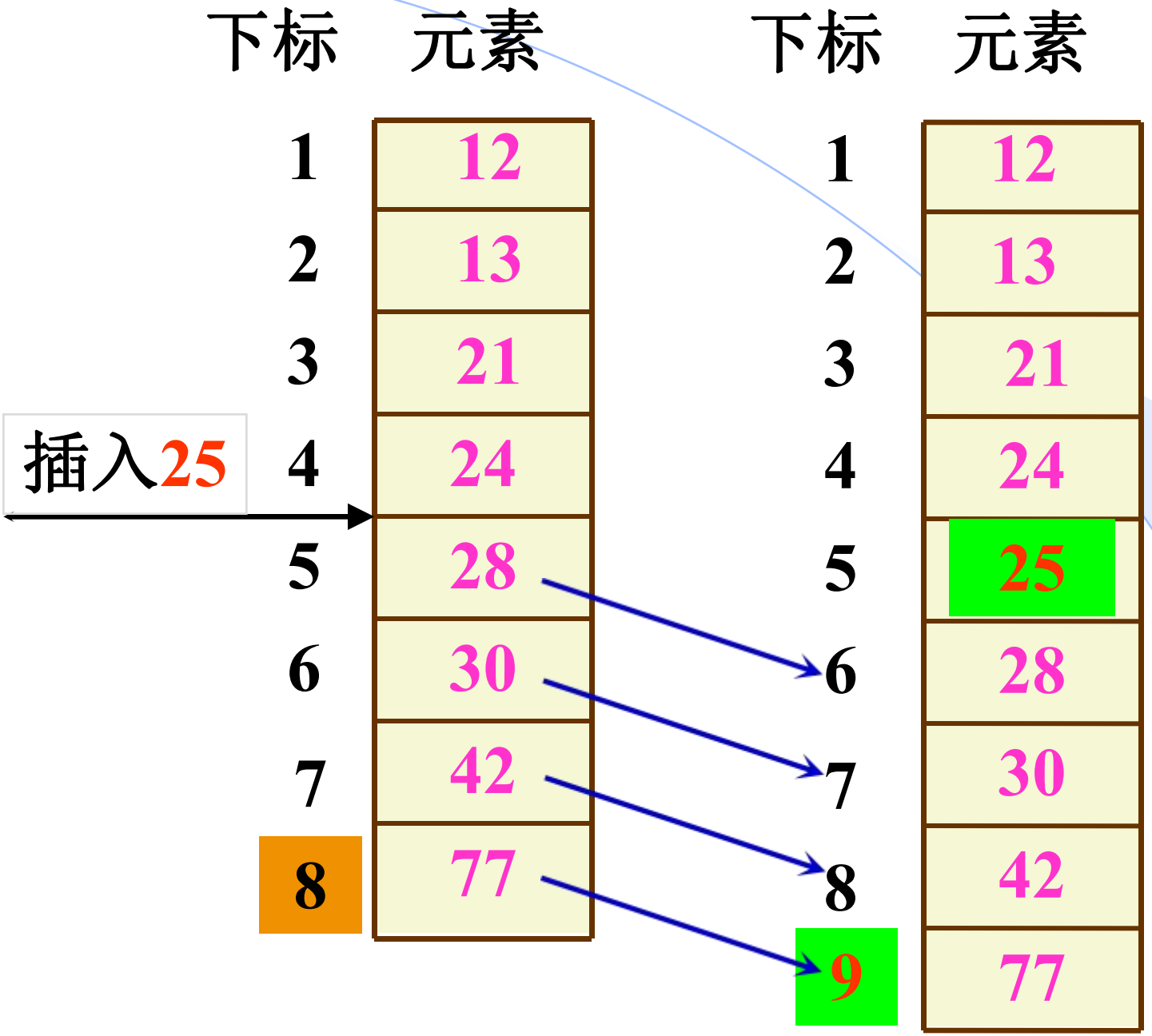
顺序存储的线性表的基本操作

1、插入

[例] 在顺序表（12, 13, 21, 24, 28, 30, 42, 77）中元素24的后面插入元素 25。

考虑：线性表的逻辑结构发生什么变化？

- 某些结点逻辑关系发生变化（结点位置关系变化）
- 线性表长度增加 1


下标
元素
1
12
2
13
3
21
插入25
4
24
5
28
6
30
7
42
8
77
下标
元素
1
12
2
13
3
21
4
24
5
25
6
28
7
30
8
42
9
77

//采用ADL算法描述语言描述算法

算法 **Insert**(A, *k*, *item*. A) /* 在线性表A中下标为 *k* 的结点后插入值为 *item* 的结点，length为线性表的当前长度，MaxSize为最大长度 */

I1.[插入合法？]

IF (*k*<1 OR *k*>*length* OR *length*=MaxSize) THEN

(PRINT“插入不合法”. RETURN).

I2.[插入结点]

FOR *i*=length TO *k*+1 STEP –1 DO

A[*i*+1]←A[*i*].

A[*k*+1] ← *item*.

length ← length+1. ▌

考虑：如何实现在表头插入新结点？

时间复杂性分析：

插入操作的基本运算是：

元素移动

假定线性表的当前长度为 n，输入集合D_n中有多少种可能的合法输入呢？

n种(n个位置可插入元素)

设插入成功且插入到各位置的概率相同：$1/n$

则期望复杂性为：

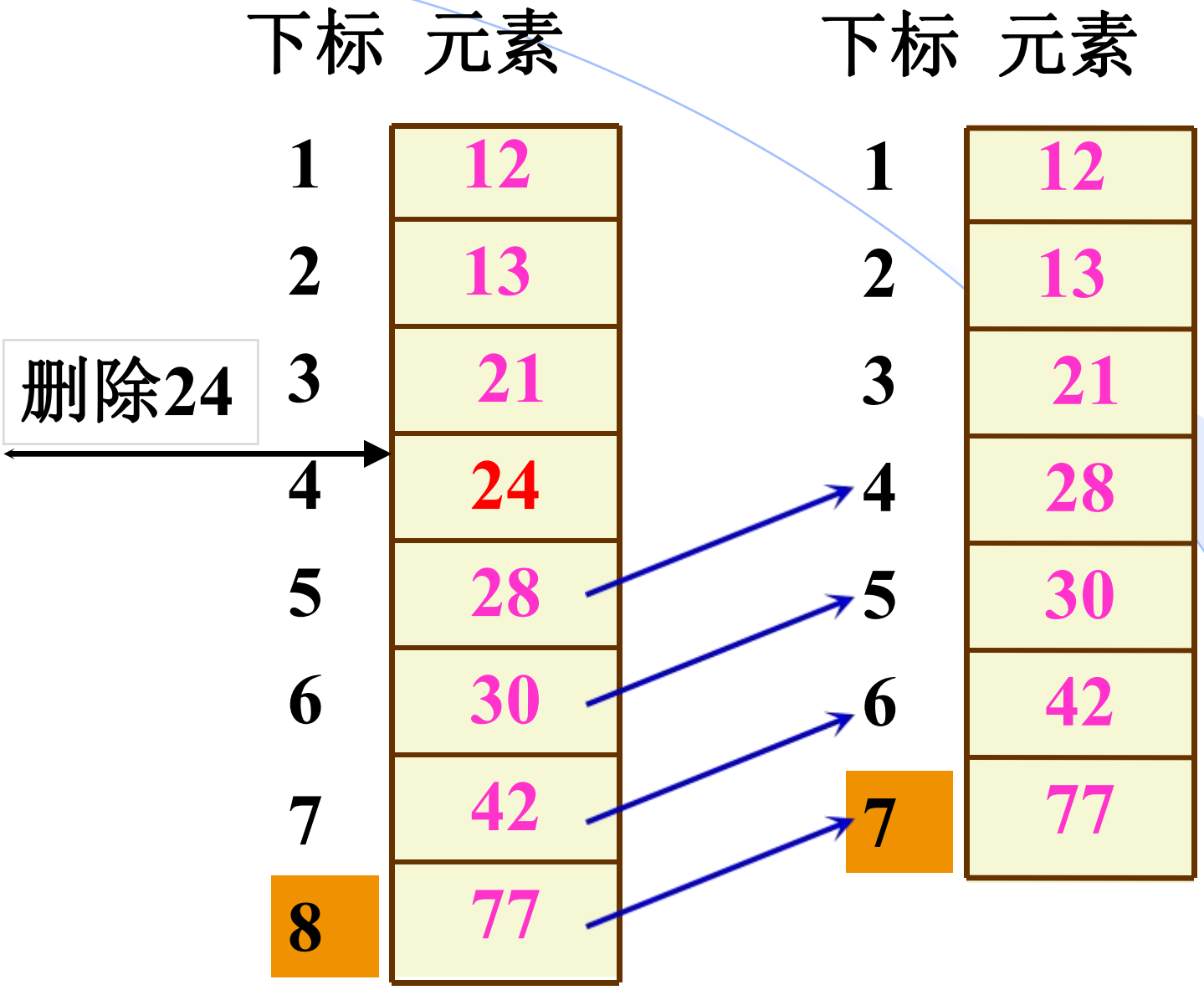
$$E(n)=1+((n-1)+(n-2)+\cdots\cdots+1+0)/n = (n-1)/2+1$$

2、删除

[例] 在顺序表(12, 13, 21, 24, 28, 30, 42, 77)中，删除元素 24。

考虑：线性表的逻辑结构发生什么变化？

- 某些结点逻辑关系发生变化（结点位置关系变化）
- 线性表长度减 1


下标 元素
1 12
2 13
3 21
删除24
4 24
5 28
6 30
7 42
8 77
下标 元素
1 12
2 13
3 21
4 28
5 30
6 42
7 77

算法 Delete(A, *k*. A) /* 删除顺序表 A中下标为 *k* 的结点 */

D1.[*k*合法?]

IF (*k*<1 OR *k*>length OR length=0)

THEN (PRINT“删除不合法”. RETURN).

D2.[删除]

FOR *i*=*k*+1 TO length DO A[*i*-1]←A[*i*].

length ←length-1. ▌

时间复杂性分析

删除操作的基本运算是：元素移动

输入集合D_n中有多少种可能的合法输入呢？

*n*种（*n*个位置可以发生删除）

设删除成功且各位置被删除的概率相等：1/*n*

则期望复杂性为：

$$E(n)=((n-1)+\ldots\ldots+1+0)/n=(n-1)/2$$

讨论

线性表的顺序存储结构

优点：空间利用率高，简单、易于实现，可以随机访问表中的任一元素，存取速度快。

缺点：插入和删除结点时间复杂性高（需移动元素，调整一批结点的地址）。

问题：由于线性表中元素的数目可以改变，因此定义数组时要做如何的考虑呢？

定义足够大的静态数组（可能空间利用率低）

使用动态数组

第三章 线性表、堆栈和队列

- 3.1 线性表的定义和基本操作
- 3.2 线性表的顺序存储结构
- 3.3 线性表的链接存储结构
- 3.4 复杂性分析
- 3.5 堆栈
- 3.6 队列

线性表的链接存储

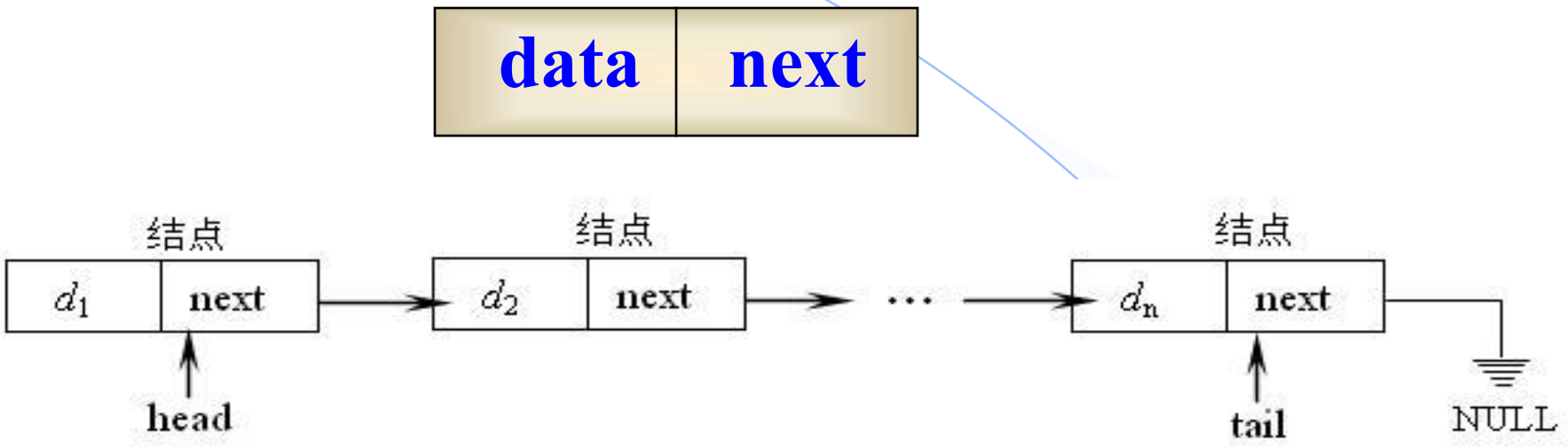
- 链接存储：用任意一组存储单元存储线性表，一个存储单元除了包含结点数据（或信息）字段的值，还必须存放其逻辑相邻结点（前驱或后继结点）的地址信息，即包含指针字段。
- 根据结点指针域的不同，链表主要有三种实现方式：单链表、循环链表和双向链表。

3.3.1 线性表的链接存储结构

——————单链表

1. 单链表的定义
2. 单链表的特点
3. 单链表的基本操作

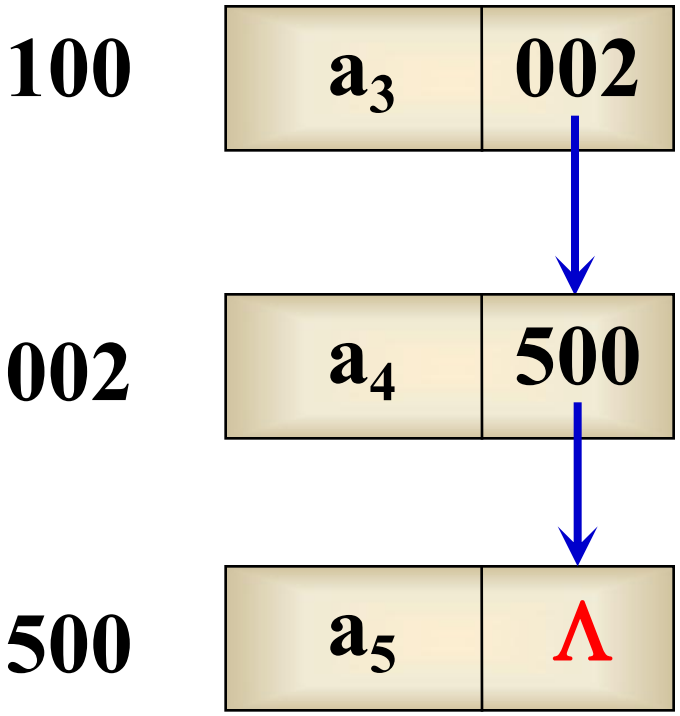
- 单链表的结点结构：



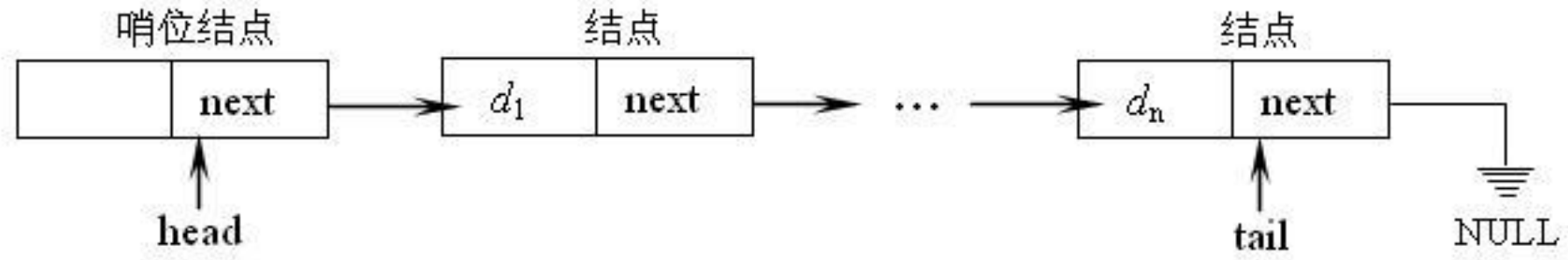
◆ 链表的第一个结点被称为头结点(也称为表头)，指向头结点的指针被称为头指针(head).

◆ 链表的最后一个结点被称为尾结点(也称为表尾)，指向尾结点的指针(如果有)被称为尾指针(tail).

[例] 将线性表（a_3，a_4，a_5），以单链表的形式存储在内存中。



为了对表头结点插入、删除等操作的方便，通常在表的前端增加一个特殊的表头结点，称其为哨位结点。



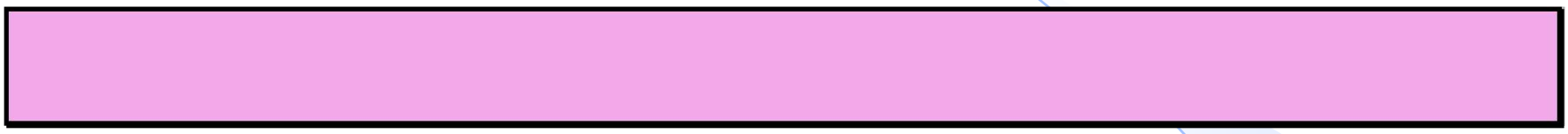
3.3.1 线性表的链接存储结构

——单链表

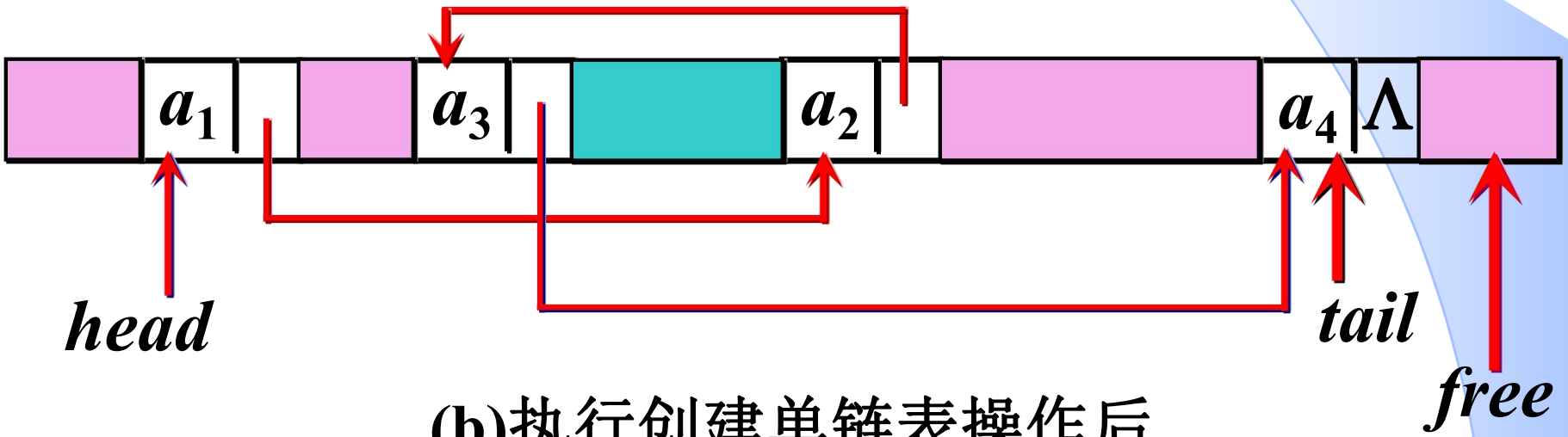
1. 单链表的定义
2. 单链表的特点
3. 单链表的基本操作

特点:逻辑顺序与物理顺序可以相同也可不同，
即逻辑上相邻的结点在物理上不必相邻。

单链表的存储映像



(a)可用存储空间



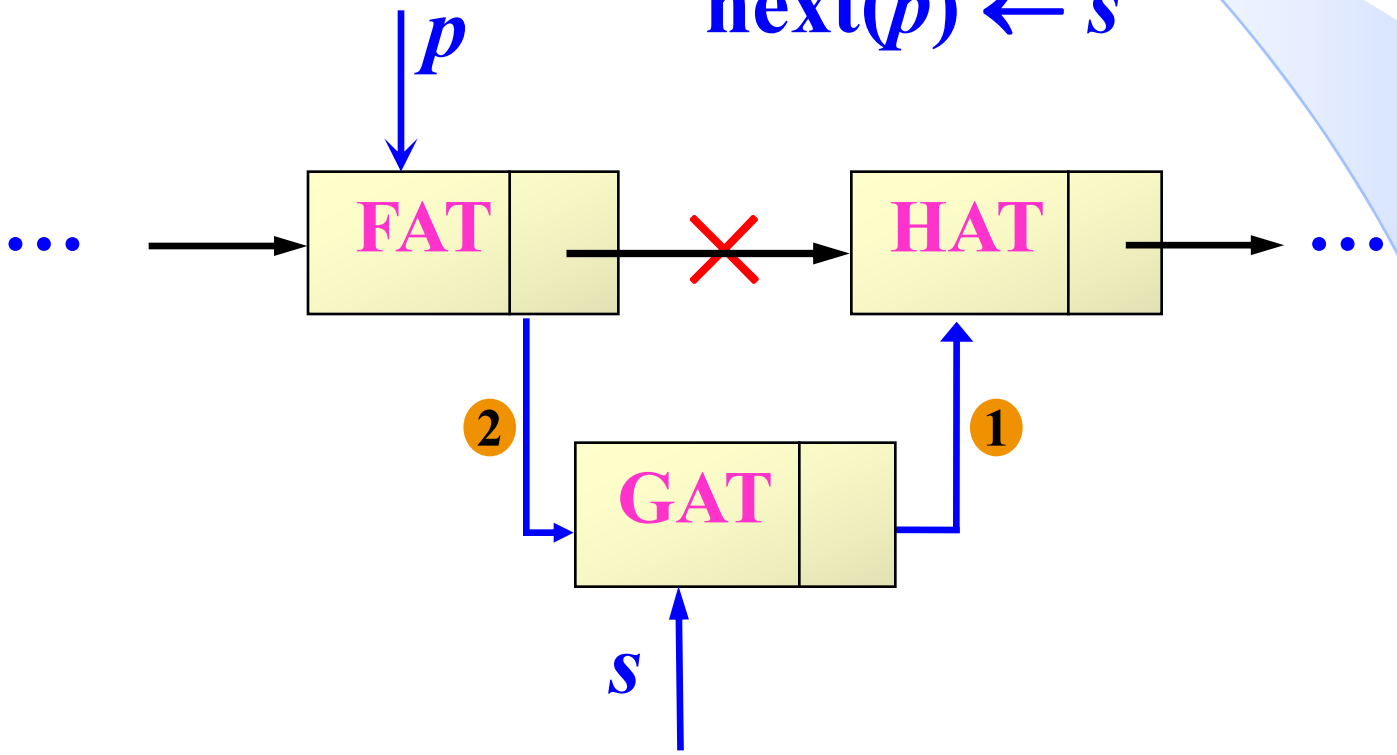
(b)执行创建单链表操作后

在单链表中，插入或删除一个结点，只需改变一个或两个相关结点的指针域，不对其它结点产生影响。

- 插入操作：在结点 *p* 之后插入结点 *s*

$$next(s) \leftarrow next(p)$$

$$next(p) \leftarrow s$$



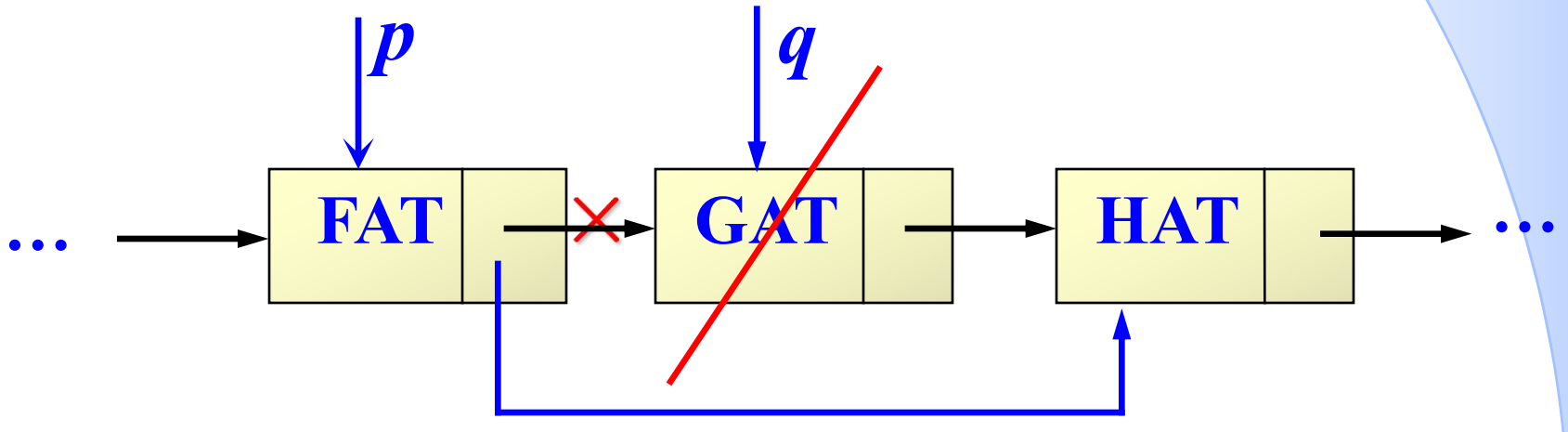
在单链表中，插入或删除一个结点，只需改变一个或两个相关结点的指针域，不对其它结点产生影响。

- 删除：将结点*p*的后继结点删除

  $q \leftarrow \text{next}(p)$.

  $\text{next}(p) \leftarrow \text{next}(q)$.

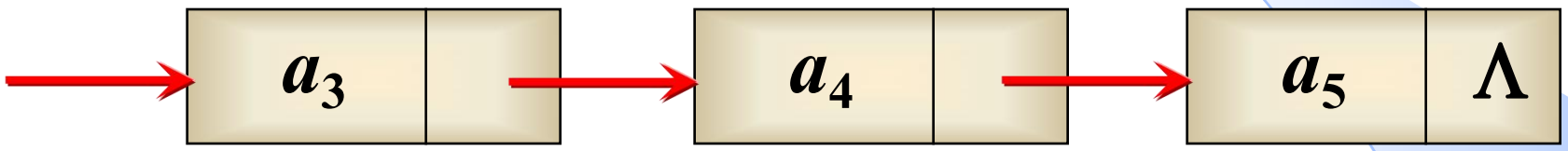
  $\text{AVAIL} \Leftarrow q$.



- 单链表的特性：

① 利用链接域实现线性表元素的逻辑关系。

② 单链表有头结点、尾结点、头指针。



- 单链表的优点：

① 插入、删除方便；

② 共享空间好：结点可以不连续存储，易于扩充。

3.3.1 线性表的链接存储结构

——单链表

1. 单链表的定义
2. 单链表的特点
3. 单链表的基本操作

单链表基本操作的实现

算法 Find (*k*. *item*) // 将链表中第*k*个结点的字段值赋给*item*

F1. [*k*合法？]

IF (*k* < 1) THEN (PRINT “存取位置不合法”.
RETURN.)

F2. [初始化]

p←*head* . *i*←0. //令指针*p*指向哨位结点，计数器初始值为0

F3. [找第*k*个结点]

WHILE (*p*≠NULL AND *i*<*k*) DO
(*p*←next(*p*) . *i*←*i*+1.)
// 若找到第*k*个结点或已到达表尾，则循环终止
IF *p* = NULL THEN (PRINT “无此结点”. RETURN.)
item←data(*p*). ▌

- 算法Find，最好情况下的时间复杂度为$O(1)$；最坏情况下的时间复杂度为$O(n)$；
- 平均情况下，假设$k<1, k=1, \ldots, k=n, k>n$的概率相同，即每种情况的发生概率为$1/(n+2)$，则WHILE循环的执行次数平均为

$$\frac{0+1+\ldots+n+n+1}{n+2}=\frac{1}{n+2}\frac{(n+1)(n+2)}{2}=\frac{n+1}{2}=O(n)$$

- 因此，存取操作的时间复杂度为$O(n)$.

算法 **Search** (*item*. *i*) /* 在链表中查找字段值为*item*的结点并返回其在表中的位置*/

S1. [初始化]

p←next(*head*) . *i*←1. /* 令指针p指向哨位结点的后继结点，计数器初始值为1*/

S2. [逐点访问]

WHILE (p ≠ NULL AND data(*p*) ≠ *item*) DO

(*p*←next(*p*) . *i*←*i*+1.)

/* 令指针p指向下一个结点，且计数器加1*/

IF *p*≠NULL THEN RETURN .

PRINT “无此结点”. *i* ← -1. RETURN. ▌

- Search算法最好情况下的时间复杂度为*O*(1)，最坏情况和平均情况下的时间复杂度皆为*O*(*n*).

算法Delete (*k*. *item*) // 删除链表中第*k*个结点并将
// 其字段值赋给*item*

D1. [*k*合法?]

IF (*k* < 1) THEN (PRINT “删除不合法”. RETURN.)

D2. [初始化]

p←*head* . *i*←0. // 令指针*p*指向哨位结点，计数器初始值为0

D3. [找第*k*个结点]

WHILE (*p*≠NULL AND *i*<*k* –1) DO

(*p*←next(*p*) . *i*←*i*+1.)

IF *p* = NULL THEN (PRINT “无此结点”. RETURN.)

/* 无第*k*–1个结点*/

IF next(*p*)=NULL THEN (PRINT “无此结点”. RETURN.)

/* 无第*k*个结点*/

D4. [删除]

q←next(*p*). next(*p*) ←next(*q*). // 修改*p*的next指针

item←data(*q*). AVAIL⇐*q*. ▌ //存取*q*的值并释放其存储空间

算法Delete (*k*. *item*) // 删除链表中第*k*个结点并将
// 其字段值赋给*item*

D1. [*k*合法?]

IF (*k* < 1) THEN (PRINT “删除不合法”. RETURN.)

D2. [初始化]

p←*head* . *i*←0. // 令指针*p*指向哨位结点，计数器初始值为0

D3. [找第*k*个结点]

WHILE (next(*p*) ≠NULL AND *i*<*k* –1) DO

(*p*←next(*p*) . *i*←*i*+1.)

IF next(*p*)=NULL THEN (PRINT “无此结点”. RETURN.)
/* 无第*k*个结点*/

D4. [删除]

q←next(*p*). next(*p*) ←next(*q*). // 修改*p*的*next*指针

item←data(*q*). AVAIL⇐*q*. ▌ //存取*q*的值并释放其存储空间

算法Insert (*k*, *item*. *head*) // 在链表中第*k*个结点后插入
// 字段值为*item*的结点

I1. [*k*合法？]

IF (*k* < 0) THEN (PRINT“插入不合法”. RETURN.)

I2. [初始化]

p←*head* . *i*←0. //令指针p指向哨位结点，计数器初始值为0

I3. [*p*指向第*k*结点]

WHILE (*p*≠NULL AND *i*<*k*) DO

(*p*←next(*p*) . *i*←*i*+1.)

IF *p*=NULL THEN (PRINT “插入不合法”. RETURN.)

I4. [插入]

s⇐AVAIL. data(*s*) ←*item*. next(*s*) ←next(*p*). /* 生成新结点*s*，
其next指针指向*p*的后继结点*/

next(*p*) ←*s*. ▌ // 修改*p*的next指针，令其指向新插入结点*s*

插入、删除操作的时间复杂性分析

- 插入、删除操作的最好情况的时间复杂度为$O(1)$;
- 插入、删除操作最坏情况下的时间复杂度为$O(n)$;
- 平均情况下，时间复杂度也是$O(n)$。

单链表的不足：

① 单链表虽然克服了顺序存储的缺点，但却不能进行随机访问。

② 对单链表来说，只有从头结点开始才能扫描链表中的全部结点 。

③ 从一个结点出发，只能访问到链接在它后面的结点，而无法访问位于它前面的结点。

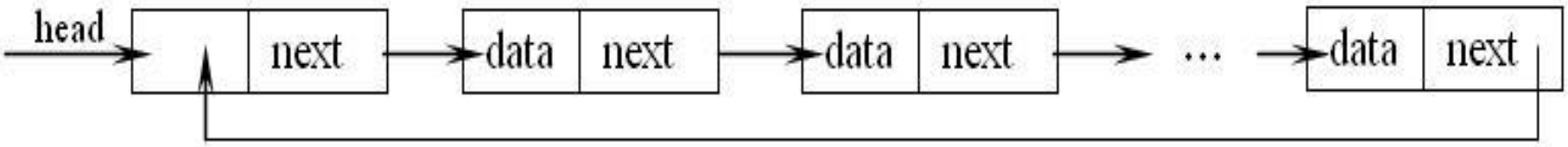
3.3 线性表的链接存储结构

3.3.1 单链表

3.3.2 循环链表

3.3.3 双向链表

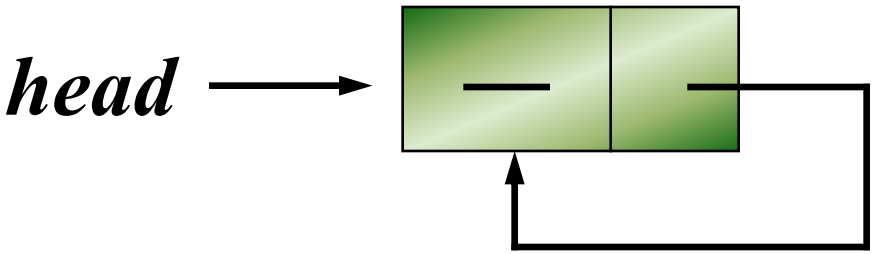
- 把链接结构“循环化”，即让表尾结点的next域存放指向哨位结点的指针，而不是存放空指针NULL（即Λ），这样的单链表被称为循环链表。
- 循环链表使我们可以从链表的任何位置开始，访问链表中的任一结点。



判断链表为空的条件：

单链表：next(*head*) = NULL

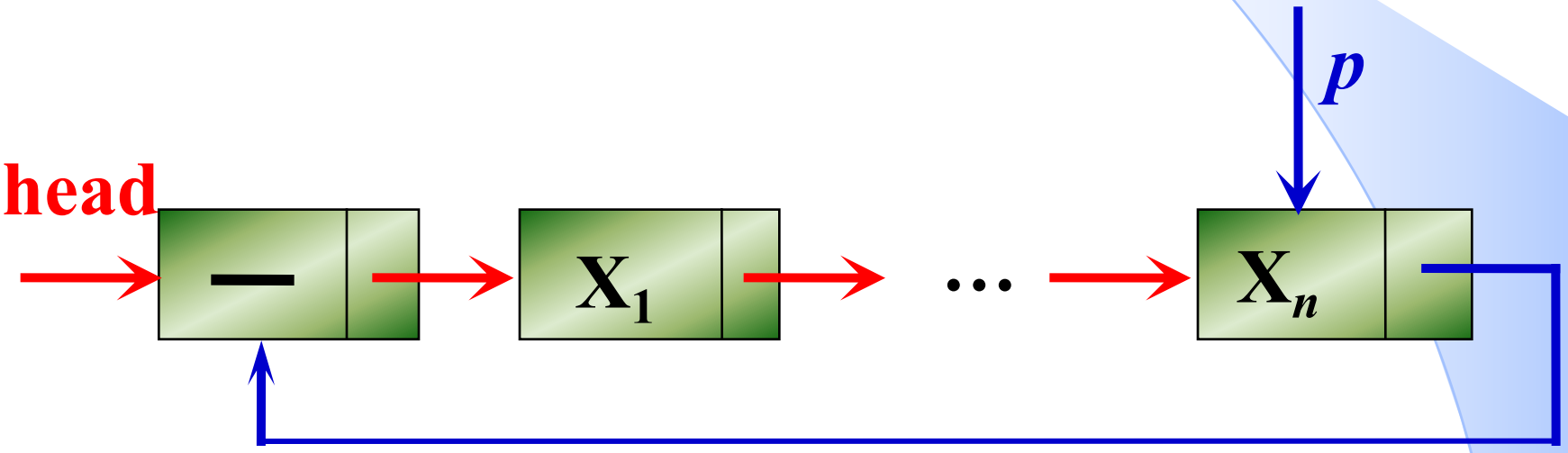
循环链表：next(*head*) = *head*



判断表尾的条件：

单链表： next (*p*) = NULL

循环链表： next (*p*) = *head*



3.3 线性表的链接存储结构

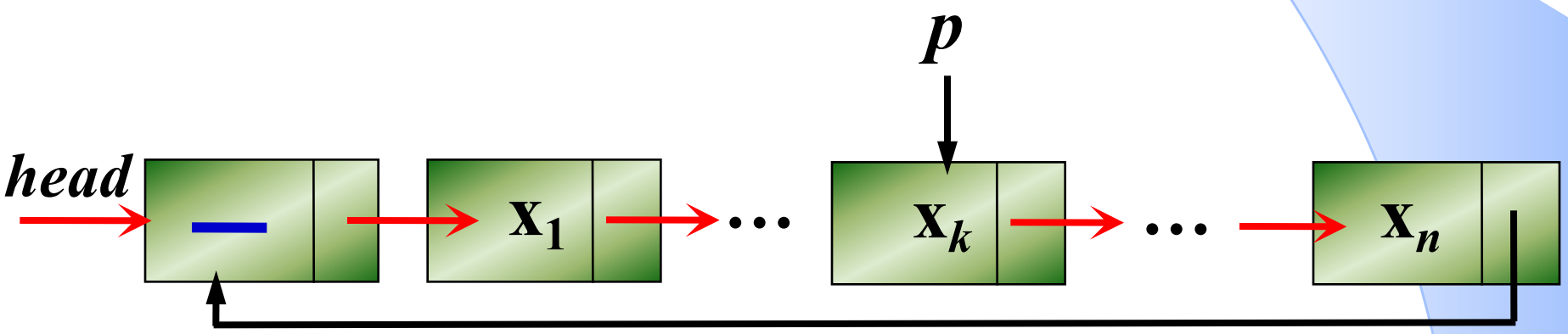
3.3.1 单链表

3.3.2 循环链表

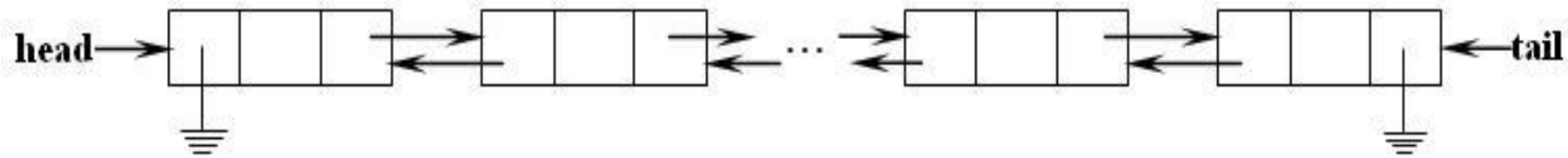
3.3.3 双向链表

循环链表遇到的问题：

在循环链表中访问某结点p的前驱结点，需遍历整个链表，其时间复杂性为$O(n)$.



- 所谓双向链表(Double-Linked List)，指链表中任一结点*P*都是由data域、左指针域left和右指针域right构成的，左指针域和右指针域分别存放*P*的左右两边相邻结点的地址信息，链表中表头结点的left指针和表尾结点的right指针均为NULL. 指针*head*和*tail*分别指向双向链表的头结点和尾结点，双向链表也被称为双重链表。

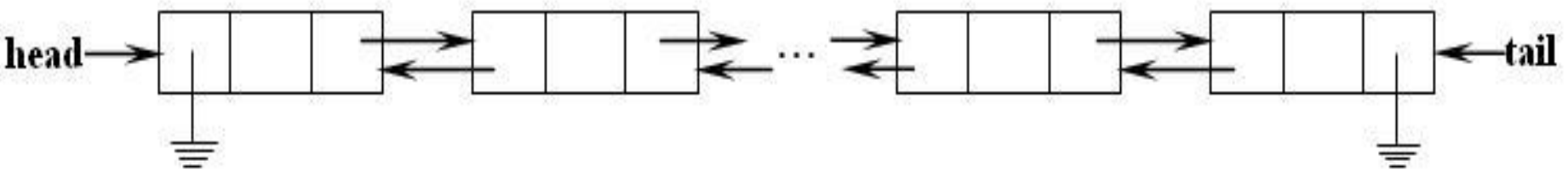


双向链表的特点：

- 每个结点有两个指针域left和right
- 左指针指向其前驱结点，右指针指向其后继结点
- 头结点和尾结点的指针域特殊

优点：

方便找某个结点的前驱结点。



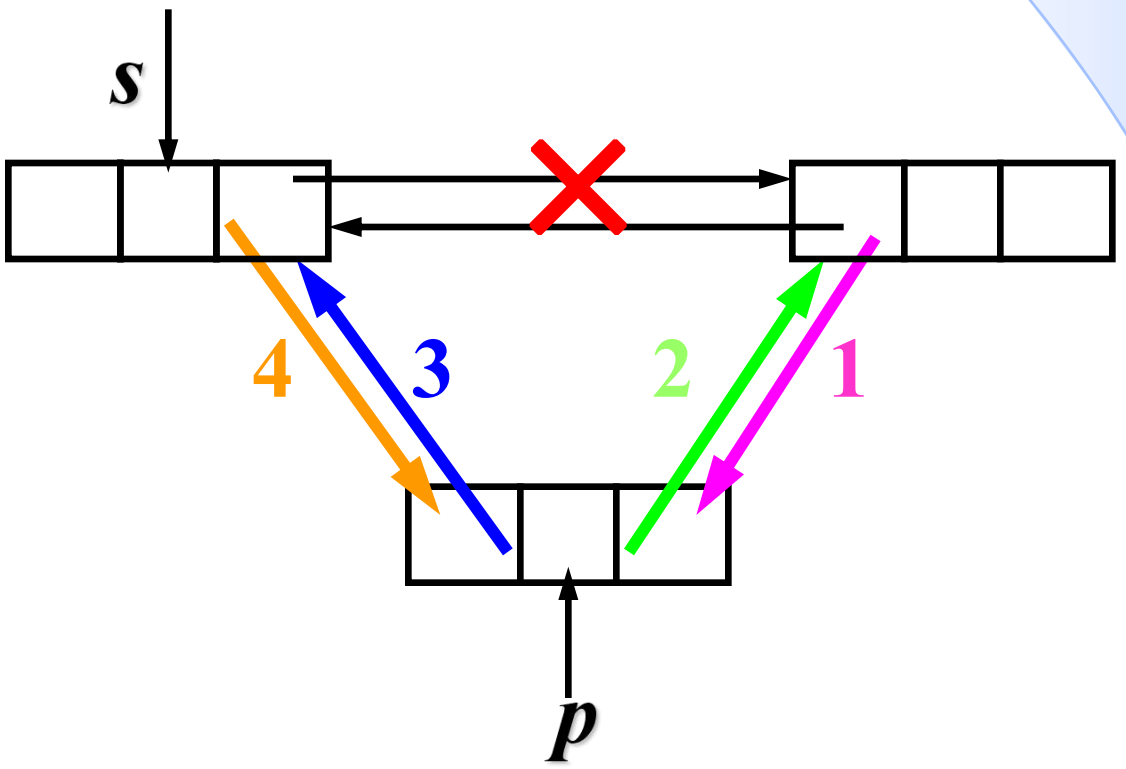
结点插入：在结点s之后插入结点 p

left(right(*s*)) ← *p* .

right(*p*) ← right(*s*).

left(*p*) ← *s*.

right(*s*) ← *p* .



算法**DLInsert**（*s* , *p*. *tail*）// 在结点*s*的右边插入结点*p*

DLI1.[s为尾结点] // 结点*s*是否为尾结点？

IF right (*s*) = Λ THEN ((right (*p*) ←Λ . left (*p*) ← *s*.
right (*s*)←*tail* ← *p*. RETURN.)

DLI2. [插入]

right (*p*) ← right (*s*).

left (*p*) ← *s*.

left (right (*s*)) ← *p* .

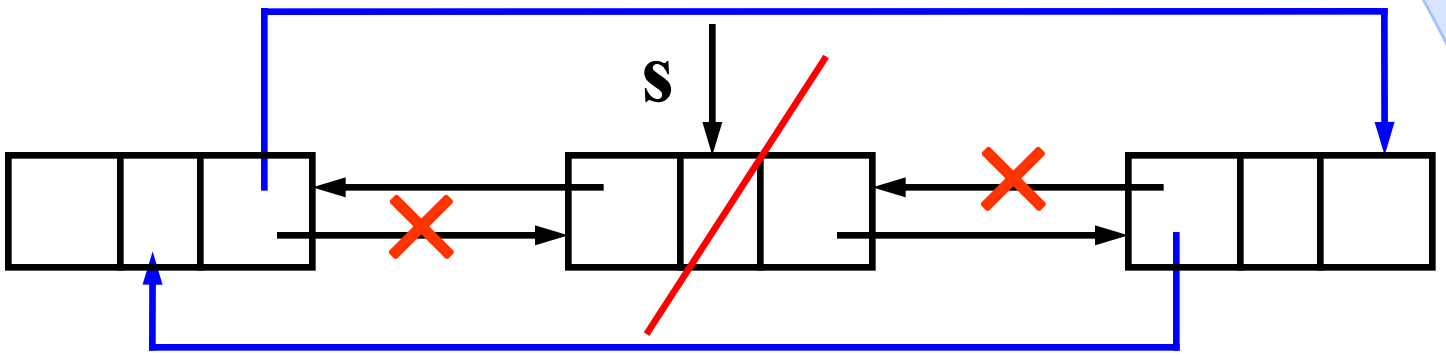
right (*s*) ← *p* . ▌

删除结点

right(left(*s*)) ← right(*s*)

left(right(*s*)) ← left(*s*)

AVAIL⇐ *s*



算法DeleteNode (*s. head, tail*) // 删除双向链表中的结点*s*

DN1.[*s*为链表中的唯一结点] // 双向链表中只有一个结点

IF left (*s*) = right (*s*) = Λ

THEN (*head*← *tail* ← Λ. GOTO DN4.).

DN2.[*s*是头结点]

IF left (*s*) = Λ

THEN (left (right (*s*)) ←Λ . *head*← right (*s*) . GOTO DN4.).

DN3.[*s*是尾结点]

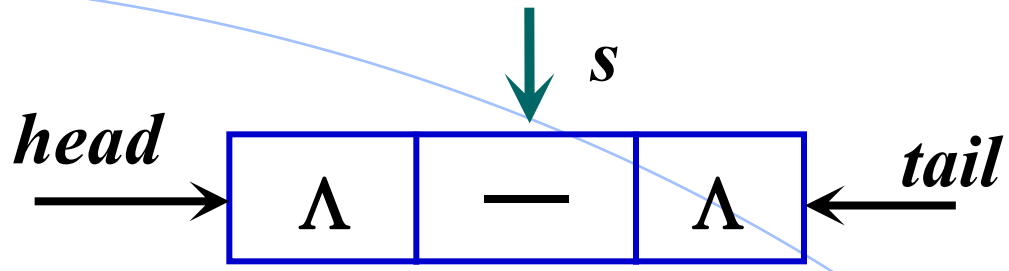
IF right (*s*) = Λ

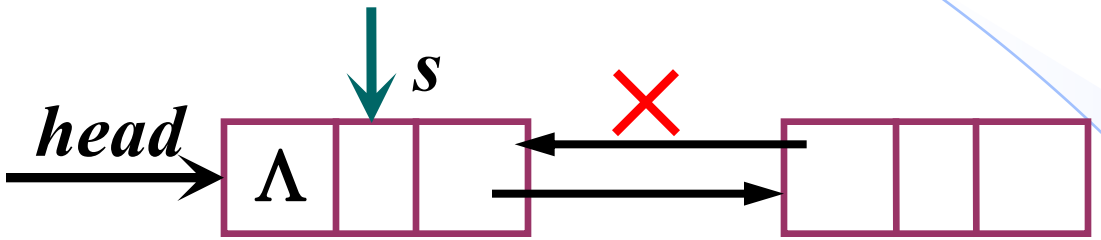
THEN (right (left (*s*)) ← Λ . *tail* ← left (*s*) . GOTO DN4.).

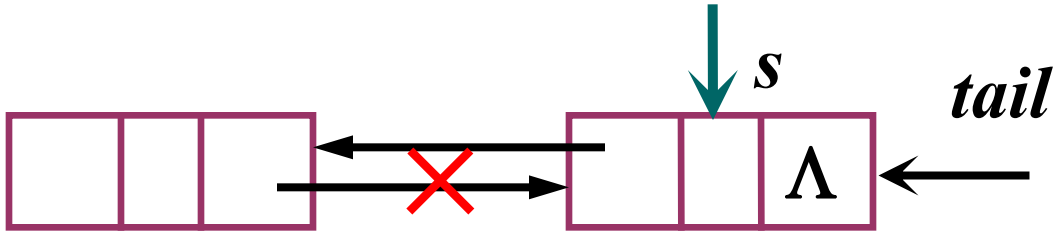
ELSE (right (left (*s*)) ← right (*s*) . left (right (*s*)) ← left (*s*).)

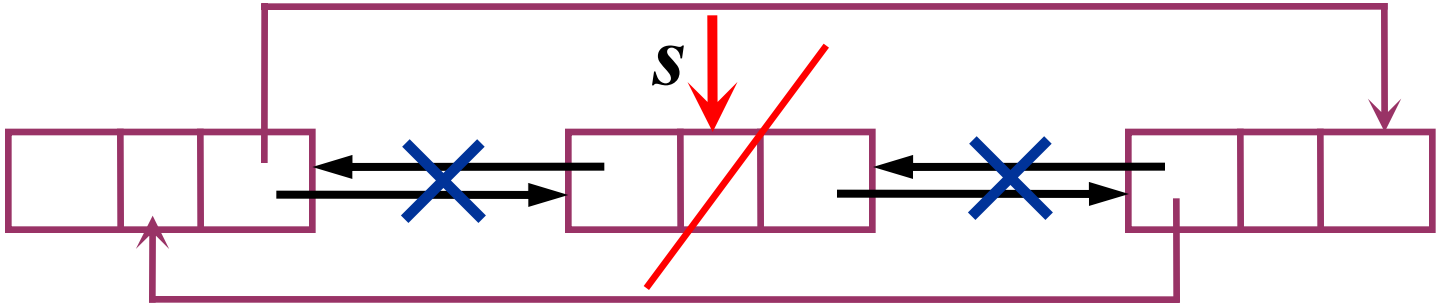
DN4.[释放结点*s*]

AVAIL⇐ *s*. ▌


s
head
tail
Λ
—
Λ


s
head
Λ


s
tail
Λ


s

由于在某些应用中链表的头结点和尾结点使用频繁，在这种场合使用带尾指针的循环链表比较合适。

双向链表的特点是寻找某个结点的前驱和后继结点都很容易。在需要经常查找结点的前驱和后继的场合，使用双向链表比较合适。

还可以结合循环链表和双向链表的特点，构造双向循环链表。

静态单链表

静态单链表是一种借助数组来实现的线性链表，它将数据元素可能的存储范围局限于一维数组内，在数组内数据元素可以任意存放，即逻辑上相邻的数据元素可以是不相邻的数组元素。

在静态单链表中，对于一个数据元素而言，除了存储该元素的值以外，还需要存储其直接后继在一维数组中的下标。

为了便于为新插入的数据元素分配所需的存储空间，需要维护一个由空闲数组单元构成的静态单链表。

第三章 线性表、堆栈和队列

- 3.1 线性表的定义和基本操作
- 3.2 线性表的顺序存储结构
- 3.3 线性表的链接存储结构
- 3.4 复杂性分析
- 3.5 堆栈
- 3.6 队列

3.4 顺序存储和链式存储的复杂性分析

1、空间效率的比较

- 顺序表所占用的空间来自于申请的数组空间，数组大小是事先确定的，很明显，当表中的元素较少时，顺序表中的很多空间处于闲置状态，造成了空间的浪费；
- 链表所占用的空间是根据需要动态申请的，不存在空间浪费的问题，但是链表需要在每个结点上附加一个或多个指针，从而产生额外开销。

2、时间复杂性的比较

- 线性表的基本操作是存取、插入和删除。对于顺序表，随机存取是非常容易的，但是每插入或者删除一个元素，都需要移动若干元素。对于链表，无法实现随机存取，必须要从表头开始遍历链表，直到找到要存取的元素，但是链表的插入和删除操作却非常简便，只需要修改一个或者两个指针值。
- 当线性表经常需要进行插入、删除操作时，链表的时间复杂性较小，效率较高；当线性表经常需要存取，且存取操作比插入删除操作频繁的情况下，则顺序表的时间复杂性较小，效率较高。