



第2章 线性表





本章重点与难点

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。



第二章 线性表

2.1 线性表概念及基本操作

2.2 线性表的顺序存储和实现

2.3 线性表的链式存储和实现

2.3.1 线性链表

2.3.2 循环链表

2.3.3 双向链表



第二章 线性表

● 本章主要内容

- ✓ 线性表的定义
- ✓ 线性表的顺序存储结构及其基本操作
- ✓ 线性表的链式存储结构及其基本操作

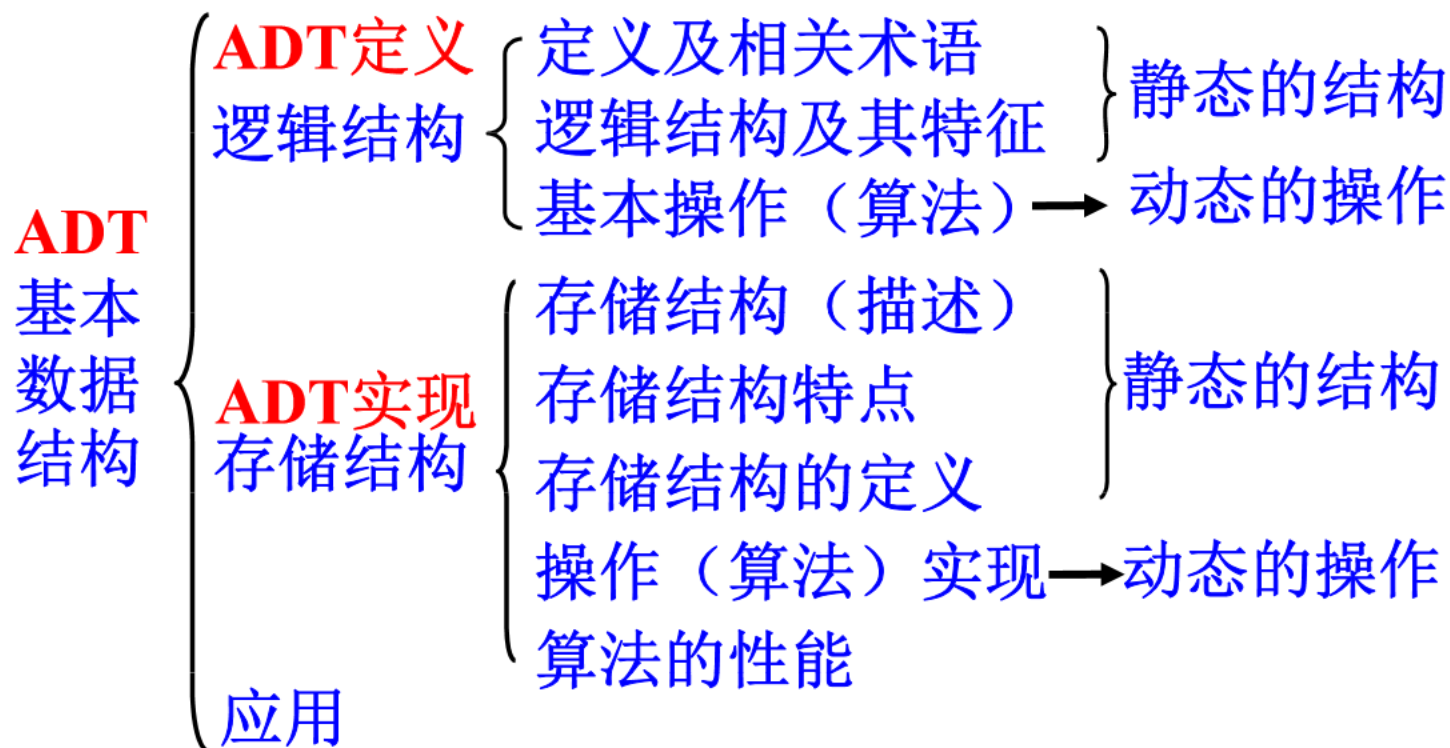
● 学习目的及要求:

- ✓ 掌握线性表的定义
- ✓ 掌握线性表的两种存储结构及其操作



知识点结构梳理

- 基本的数据结构（ADT）
 - 线性表、栈、队列、串、多维数组、广义表
- 知识点结构





第二章 线性表

2.1 线性表概念及基本操作

2.2 线性表的顺序存储和实现

2.3 线性表的链式存储和实现

2.3.1 线性链表


2.3.2 循环链表

2.3.3 双向链表



2.1 线性表概念及基本操作

- 线性表的逻辑结构

- 线性表的定义：是n个**类型相同**数据元素的有限序列。
- 通常记作： $L = (a_1, a_2, a_3, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ ；
- a_i ($1 \leq i \leq n$) 称为数据元素；
 - 下标*i*表示该元素在线性表中的位置或序号。
- n为线性表中元素个数，称为线性表的**长度**；
 - n=0时为空表，记为L= ()。
- L的图示表示：
- 例：
 - 数学中的数列 (11, 13, 15, 17, 19, 21)
 - 英文字母表 (A, B, C, D, E, ..., Z)



2.1 线性表概念及基本操作

- 线性表的逻辑特征 $L = (a_1, a_2, a_3, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$
 - 有限性：
 - 线性表中数据元素的个数是有穷的。
 - 相同性：
 - a_i 为线性表中的元素，元素类型相同。
 - 相继性：
 - a_1 为表中第一个元素，无前驱元素； a_n 为表中最后一个元素，无后继元素；
 - 对于 $\dots a_{i-1}, a_i, a_{i+1} \dots (1 < i < n)$ ，称 a_{i-1} 为 a_i 的直接前驱， a_{i+1} 为 a_i 的直接后继；在线性表中，除第一个元素和最后一个元素之外，其它元素都有且仅有一个直接前驱，有且仅有一个直接后继；
 - 中间不能有缺项。



2.1 线性表概念及基本操作

- 定义在线性表的操作（算法）：

线性表

LIST = (D , R)

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

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

- 设L的型为LIST线性表实例，x 的型为DataType的元素实例，p 为位置变量。线性表的基本操作如下：

- | | |
|-----------------|---------------|
| ✓ 初始化(initial) | ✓ 插入(insert) |
| ✓ 置空(makennull) | ✓ 删除(delete) |
| ✓ 取元素(retrieve) | ✓ 求长度(length) |
| ✓ 定位(locate) | |



2.1 线性表概念及基本操作

- 定义在线性表的操作（算法）：

- 1 $\text{INSERT}(x, p, L)$ ：在表L的位置p处插入x
- 2 $\text{LOCATE}(x, L)$ ：返回x在L中的位置。如果x在表中多次出现，则返回第一次出现的位置。
- 3 $\text{RETRIEVE}(p, L)$ ：返回L中位置p的元素
- 4 $\text{DELETE}(p, L)$ ：删除L中位置p的元素
- 5 $\text{PREVIOUS}(p, L)$ 返回p的前驱位置
- 6 $\text{NEXT}(p, L)$ ：返回p的后继位置
- 7 $\text{MAKENULL}(L)$ ：置空表，并返回 $\text{END}(L)$
- 8 $\text{FIRST}(L)$ ：返回表中第一个位置。
- 9 $\text{END}(L)$ ：返回表中最后一个位置的下一个位置



2.1 线性表概念及基本操作

- 定义在线性表的操作（算法）：

- ✓ 说明

- 上面列出的操作，只是线性表的一些常用的基本操作；
 - 不同的应用，基本操作可能是不同的；
 - 线性表的复杂操作可通过基本操作实现。



2.1 线性表概念及基本操作

例 定义任意线性表类型为LIST，设有线性表L，函数PURGE用以删除线性表L中所有重复出现的元素。

```
Void PURGE ( LIST L )
{ Position p, q ;
  p = FIRST(L) ;//返回表中第一个位置
  while ( p != END(L) )
  { q = NEXT(p, L) ;//返回p的后继位置
    while (q != END(L) )
      if (same(RETRIEVE(p,L),RETRIEVE(q,L)))
        DELETE(q,L) ;
      else
        q = NEXT(q,L) ;//返回q的后继位置
    p = NEXT(p,L) ;
  }
}
```



第二章 线性表

2.1 线性表概念及基本操作

2.2 线性表的顺序存储和实现

2.3 线性表的链式存储和实现

2.3.1 线性链表

2.3.2 循环链表

2.3.3 双向链表



2.2 线性表的顺序存储和实现

- 如何在计算机中存储线性表？
- 如何在计算机中实现线性表的基本操作？

为了存储线性表，至少要保存两类信息：

- (1) 线性表中的**数据元素**；
- (2) 线性表中数据元素的**顺序关系**。



2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表
 - 线性表的顺序存储结构，就是用一组连续的内存单元依次存放线性表的数据元素。
 - 用顺序存储结构存储的线性表——称为顺序表
 - 把线性表的元素按照逻辑顺序依次存放在数组的连续单元内；
 - 再用一个整型量表示最后一个元素所在单元的下标，即表长。



2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表

a_1 的存储地址（简称基地址）为 $\text{Loc}(a_1)$

0	a_1
1	a_2
	...
i	a_i
	...
n-1	a_n

逻辑结构



$\text{Loc}(a_1)$	a_1
$\text{Loc}(a_1)+k$	a_2
	...
$\text{Loc}(a_1)+(i-1)k$	a_i
	...
	a_n

物理存储结构



2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表

说明：在顺序存储结构下

- ✓ 线性表元素之间的逻辑关系通过元素的**存储顺序**反映（表示）出来；
- ✓ 假设线性表中每个数据元素占用 k 个存储单元，则线性表的第 i 个元素的存储位置与第1个元素的**存储位置的关系是**：

$$\text{Loc}(a_i) = \text{Loc}(a_1) + (i - 1)k$$

k 个单元

$\text{Loc}(a_1)$

a_1

a_2

a_{i-1}

a_i

$\text{Loc}(a_i)$

a_{i+1}

a_n



2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表

- 存储结构特点:

- ✓ 元素之间逻辑上的相继关系，用物理上的相邻关系来表示（用物理上的连续性刻画逻辑上的相继性）；
 - ✓ 是一种随机访问存取结构，也就是可以随机存取表中的任意元素，其存储位置可由一个简单直观的公式来表示；
 - ✓ 由于顺序表中任一个存储结点的位置可以通过计算得到，所以对顺序表中的任何一个数据元素的访问，都可以在相同的时间内实现。



2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表

怎样在计算机上实现
线性表的顺序存储结构?

可用C语言中的一维数组来表示, 但数组不是线性表, 数组存放的是线性表, 数组的类型由线性表中的数据元素的性质决定。
如:

```
#define MAX 100  
  
int v[MAX];
```



2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表

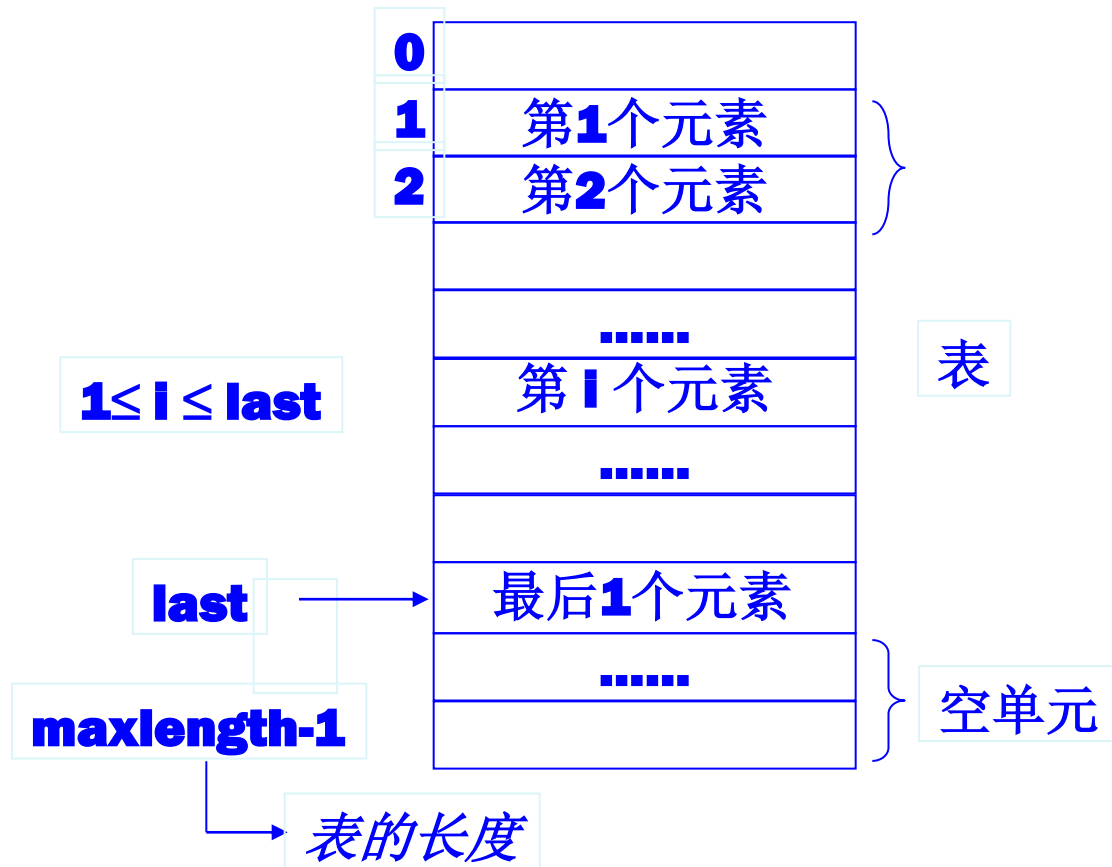


图2-1 线性表的数组实现



2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表

- 顺序表的类型定义:

```
#define MAXSIZE 1000    // 数组容量

typedef int DataType;

        // 将数组定义为整型 typedef struct

{ DataType data[MAXSIZE]; // 数组域

    int last;           // 线性表长域

} Seqlist;           // 结构体类型名
```



2.2 线性表的顺序存储和实现

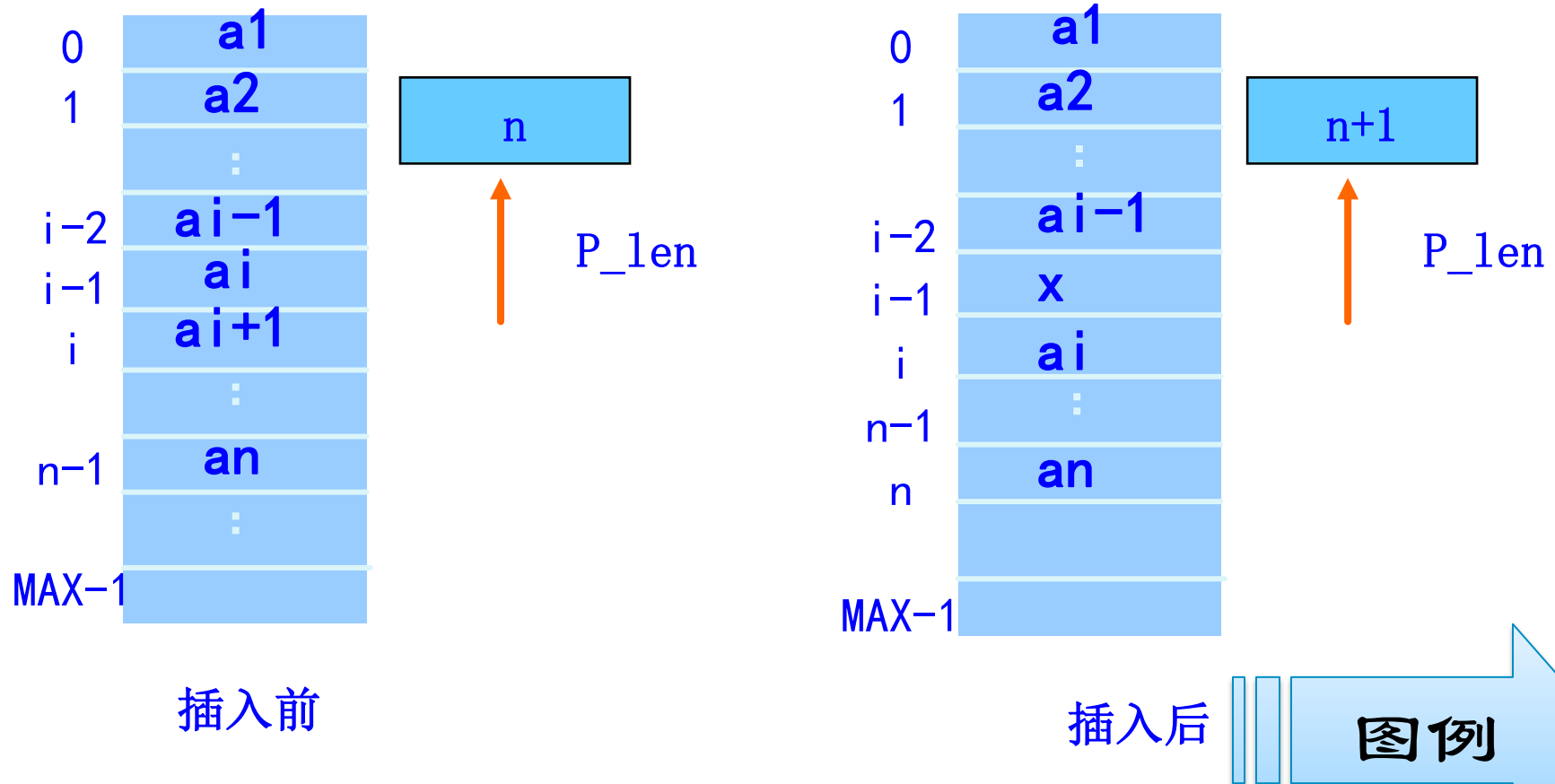
- 创建一个顺序表:

```
void Creat_Seqlist(Seqlist *L)  
{ int i,j;  
    printf("\n n= "); scanf("%d",&(L->last));  
                                /* 线性表的数据个数*/  
    for(i=1; i<=L->last; i++)  
        { printf("\n data= ");  
            scanf("%d",&(L->data[i-1])); /* 输入数据*/  
        }  
    /* Creat_Seqlist end */
```



2.2 线性表的顺序存储和实现

- 顺序表的插入--往顺序表中插入一个新数据元素





2.2 线性表的顺序存储和实现

- 顺序表的插入--往顺序表中插入一个新数据元素

线性表的插入是指在表的第 i 个位置上插入一个值为 x 的新元素，插入后使原表长为 n 的表：

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

成为表长为 $n+1$ 的表：

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

i 的取值范围为 $1 \leq i \leq n+1$ ，



2.2 线性表的顺序存储和实现

- 顺序表的插入操作（算法）

```
int Insert_Seqlist(Seqlist *L, int i, DataType x)
```

```
{ int j; i--;
```

```
    if (L->last==MAXSIZE)
```

```
        {printf("\n Error??\n");return(-1); }
```

```
        /* 表空间已满，不能插入! */
```

```
    if ((i<0) || (i > L->last))
```

```
        {printf("\n Error??");return(-1);}
```

```
        /*检查插入位置的正确性*/
```

注意边界条件



2.2 线性表的顺序存储和实现

- 顺序表的插入操作（算法）

```
else { /* 向后移动数据 */  
    for (j=L->last-1; j>=i; j--)  
        L->data[j+1]=L->data[j];  
    L->data[i]=x; /* 插入数据 */  
    L->last ++; /* 线性表长度加1 */  
    return(1); /* 插入成功，返回 */ }  
}
```



2.2 线性表的顺序存储和实现

- 顺序表的插入算法---时间性能分析
 - 顺序表上的插入运算，时间主要消耗在了数据的移动上，在第*i*个位置上插入 *x*，从 *a_i* 到 *a_n* 都要向下(右)移动一个位置，共需要移动 *n-i+1* 个元素，而 *i* 的取值范围为： **$1 \leq i \leq n+1$** ，即有 *n+1* 个位置可以插入。
 - 设在第*i*个位置上作插入的概率为 *p_i*，则平均移动数据元素的次数：

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



2.2 线性表的顺序存储和实现

- 顺序表的插入算法---时间性能分析

- 假设在线性表的任何位置插入元素的概率 p_i 相等（暂不考虑概率不相等情况），则

$$p_i = \frac{1}{n+1}$$

- 元素插入位置的可能值：

$$i = 1, 2, \dots, n, n+1$$

- 相应向后移动元素次数：

$$n-i+1 = n, n-1, \dots, 1, 0$$

- 对 $n, n-1, \dots, 1, 0$ 求总和，显然为 $n(n+1)/2$ 。所以，插入时数据元素平均移动次数为：



2.2 线性表的顺序存储和实现

- 顺序表的插入算法---时间性能分析

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

- 这说明：在顺序表上做插入操作需移动表中一半的数据元素。显然时间复杂度为 $O(n)$ 。



2.2 线性表的顺序存储和实现

- 删除顺序表中的一个数据元素

线性表的删除运算：

是指将表中第 i 个元素从线性表中去掉，删除后使原表长为 n 的线性表：

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

成为表长为 $n-1$ 的线性表：

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

i 的取值范围为： $1 \leq i \leq n$ ，

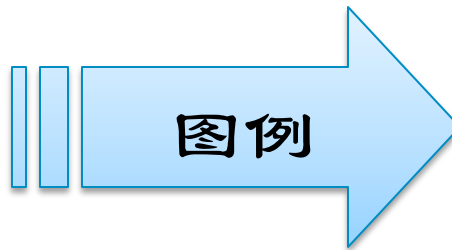


2.2 线性表的顺序存储和实现

- 删除算法的主要步骤:

DELETE(i, L): 删除L中位置i的元素

- ① 若i 不合法或表L空，算法结束，提示错误信息；
- ② 将第i个元素之后的元素（不包括第i个元素）依次向前移动一个位置；
- ③ 表长-1





2.2 线性表的顺序存储和实现

- **删除**顺序表中的一个数据元素-算法

```
void Delete_Seqlist(Seqlist *L,int i)
{ int j; i--;
  if ((i<0) || (i > L->last-1)) printf("\n Not exist! ");
  else
  {for (j=i+1;j<= L->last-1;j++)
    L->data[j-1]=L->data[j]; /* 向前移动数据 */
    L->last--; /* 线性表长度减1 */
  }
}
```




2.2 线性表的顺序存储和实现

- **删除**顺序表中的一个数据元素-算法
 - ✓ 本算法注意以下问题：
 - ① 删除第 i 个元素， i 的取值为 $1 \leq i \leq n$ ，否则第 i 个元素不存在，因此，要检查**删除位置的有效性**。
 - ② 当**表空时不能做删除**，因表空时 $L \rightarrow \text{last}$ 的值为0，条件 $(i < 0 \ || \ i > L \rightarrow \text{last} - 1)$ 也包括了对表空的检查。
 - ③ 删除 a_i 之后，该数据已不存在，**如果需要，先取出 a_i ，再做删除**。



2.2 线性表的顺序存储和实现

- 删除算法的**时间性能**分析
 - 与插入运算相同，其时间主要消耗在了移动表中元素上，删除第*i*个元素时，其后面的元素 $a_{i+1} \sim a_n$ 都要向上移动一个位置，共移动了 $n-i$ 个元素，所以平均移动数据元素的次数：

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



2.2 线性表的顺序存储和实现

• 删除算法的时间性能分析

- 假设在线性表的任何位置删除元素的概率 q_i 相等（暂不考虑概率不相等情况）：

$$q_i = \frac{1}{n}$$

- 元素删除位置的可能值： $i = 1, 2, \dots, n$
- 相应向前移动元素次数： $n-i = n-1, n-2, \dots, 0$
- 对 $n-1, \dots, 1, 0$ 求总和，显然为 $n(n-1)/2$ 。则删除时数据元素平均移动次数为：

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

- 说明顺序表上作删除运算时大约需要移动表中一半的元素，显然该算法的时间复杂度为 $O(n)$ 。



2.2 线性表的顺序存储和实现

- 顺序表中**查找**一个数据元素
 - ✓ 给定数据x，在顺序表L中查找第一个与它相等的数据元素。如果查找成功，则返回该元素在表中的位置；如果查找失败，则返回-1。算法如下：

```
int Location_SeqList(SeqList *L, DataType x)
{ int i=0;
  while(i<=L->last-1 && L->data[i]!= x)i++;
  if (i>L->last-1) return -1;
  else return i;    //返回的是存储位置
}
```



2.2 线性表的顺序存储和实现

- 顺序表中**查找**一个数据元素---算法分析
 - ✓ 该算法的主要运算是**比较**。
 - ✓ 显然比较的次数与 x 在表中的位置有关，也与表长有关。
 - ✓ 当 $a_1=x$ 时，比较一次成功。
 - ✓ 当 $a_n=x$ 时，比较 n 次成功。
 - ✓ 平均比较次数为 $(n+1)/2$ ，时间性能为 **$O(n)$** 。



2.2 线性表的顺序存储和实现

- 顺序表中**查找**一个数据元素---其他操作实现

RETRIEVE : 返回L中位置p的元素

```
elementtype RETRIEVE ( position p , LIST L )
{ if ( p > L.last )
    error( “指定元素不存在” );
  else
    return ( L.elements[ p ] );
} //时间复杂性:O(1)
```



2.2 线性表的顺序存储和实现

- 顺序表中**查找**一个数据元素---其他操作实现

PREVIOUS(p, L): 返回p的前驱

```
position PREVIOUS( position p , LIST L )
{ if ( ( p <= 1 ) || ( p > L.last ) )
    error ( “前驱元素不存在” );
  else
    return ( p - 1 );
} //时间复杂性:O(1)
```



2.2 线性表的顺序存储和实现

- 顺序表中**查找**一个数据元素---其他操作实现

NEXT(p, L): 返回p的后继位置

```
position NEXT( position p , LIST L )
{ if ( ( p < 1 ) || ( p >= L.last ) )
    error ( “前驱元素不存在” );
  else
    return ( p + 1 );
} //时间复杂性:O(1)
```




2.2 线性表的顺序存储和实现

- 顺序表中**查找**一个数据元素---其他操作实现

MAKENULL(L): 置空表, 并返回END(L)

```
position MAKENULL( LIST &L )
```

```
{ L.last = 0 ;
```

```
    return ( L.last +1 );
```

```
}
```

```
//时间复杂性:O(1)
```



2.2 线性表的顺序存储和实现

- 顺序表中**查找**一个数据元素---其他操作实现

FIRST(L): 返回表中第一个位置。

```
position FIRST( LIST L )  
    { return ( 1 ); }  
//时间复杂性:O(1)
```

```
position END( LIST L )  
    { return( L.last + 1 ); }  
//时间复杂性:O(1)
```



2.2 线性表的顺序存储和实现

- 小结:

- ✓ 顺序表的特点:

- ① 通过元素的**存储顺序**反映 线性表中 数据元素之间的逻辑关系;
- ② 可**随机存取**顺序表的元素;
- ③ 顺序表的插入、删除操作要通过**移动元素**实现。



第二章 线性表

2.1 线性表概念及基本操作

2.2 线性表的顺序存储和实现

2.3 线性表的链式存储和实现

2.3.1 线性链表

2.3.2 循环链表

2.3.3 双向链表

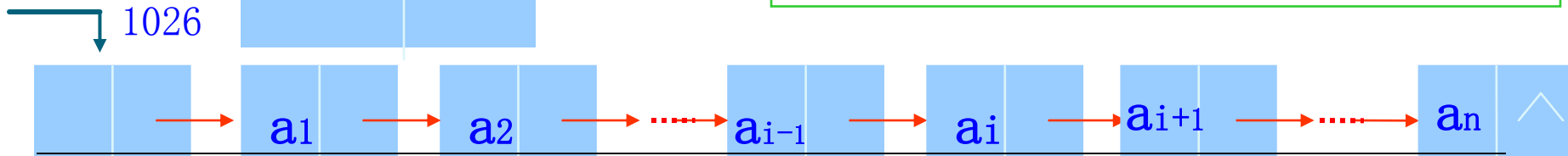


2.3.1 线性链表

- 定义：一个线性表由若干个结点组成，每个**结点**均含有两个域：存放元素的**信息域**和存放其后继结点的**指针域**，这样就形成一个**单向链接式存储结构**，简称**线性链表**或**单链表**。

1010	a₄	0
1012		
1014	a₃	1010
1016		
1018		
1020	a₁	1024
1022		
1024	a₂	1014
1026		

用线性链表存储线性表时，数据元素之间的关系是通过保存直接后继元素的存储位置来表示的





2.3.1 线性链表

- 说明：用一组任意的**存储单元**存储线性表中的数据元素，对每个数据元素除了保存**自身信息**外，还保存了**直接后继元素的存储位置**。
- **存储结构特点：**
 - ✓ 逻辑次序和物理次序不一定相同；
 - ✓ 元素之间的逻辑关系用指针表示；
 - ✓ 需要额外空间存储元素之间的关系；
 - ✓ 非随机访问存取结构（顺序访问）



2.3.1 线性链表

- 线性链表有关术语:

- **结点(node)**: 数据元素及直接后继的存储位置（地址）组成一个数据元素的存储结构，称为一个结点;
- **结点的数据域**: 结点中用于保存数据元素的部分;
- **结点的指针域**: 结点中用于保存数据元素直接后继存储地址的部分;

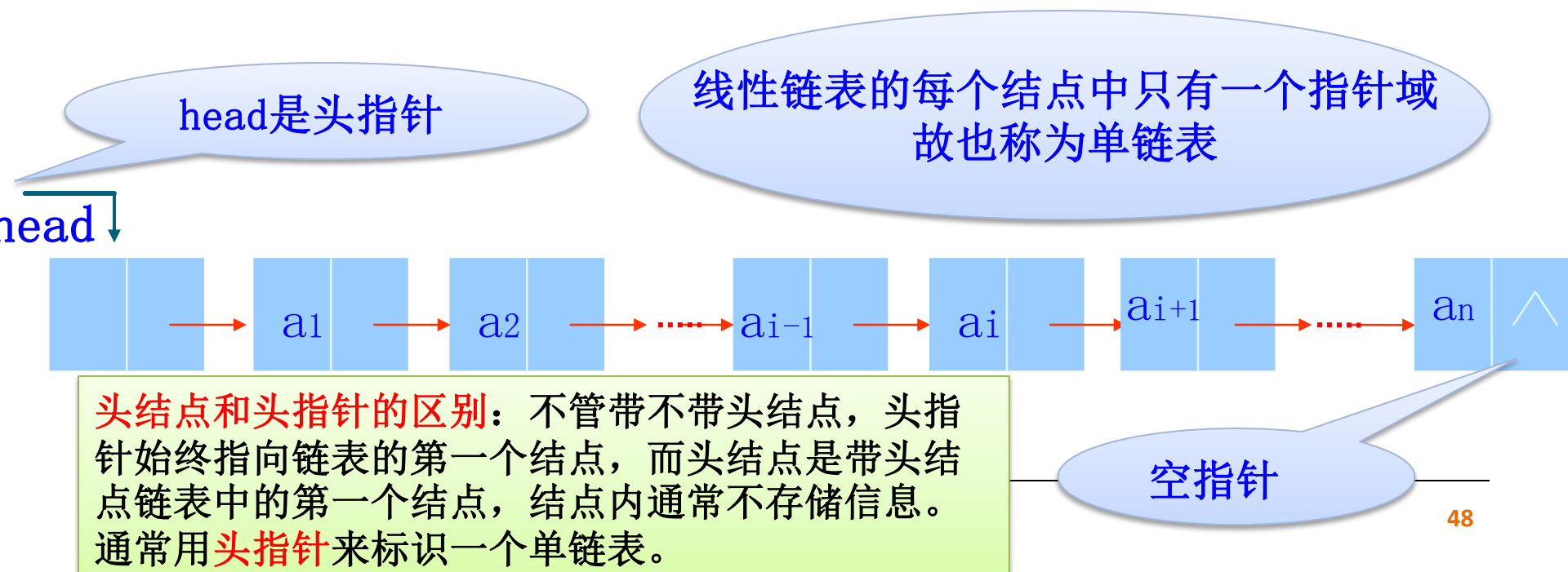




2.3.1 线性链表

• 线性链表有关术语:

- **头指针**: 用于存放线性链表中第一个结点的存储地址;
- **空指针**: 不指向任何结点, 线性链表最后一个结点的指针通常是空指针;
- **头结点**: 线性链表的第一元素结点前面的一个附加结点, 称为头结点;
- **带头结点的线性链表**: 第一元素结点前面增加一个附加结点(头结点)的线性链表称为带头结点的线性链表;

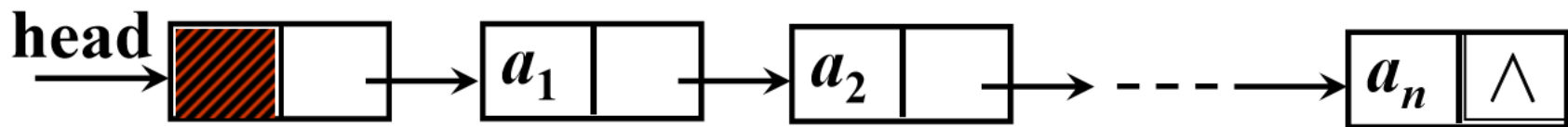




2.3.1 线性链表

- 表头结点的作用：

- 空表和非空表表示统一；
- 在任意位置的插入或者删除的代码统一；
- 注意：是否带表头结点在存储结构定义中无法体现，由操作决定。

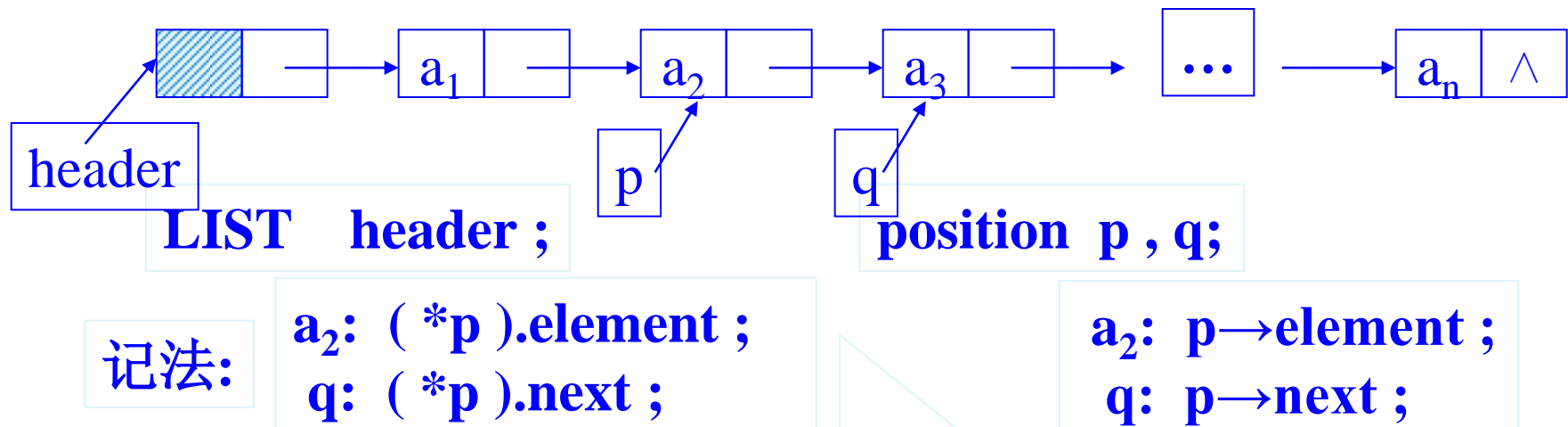
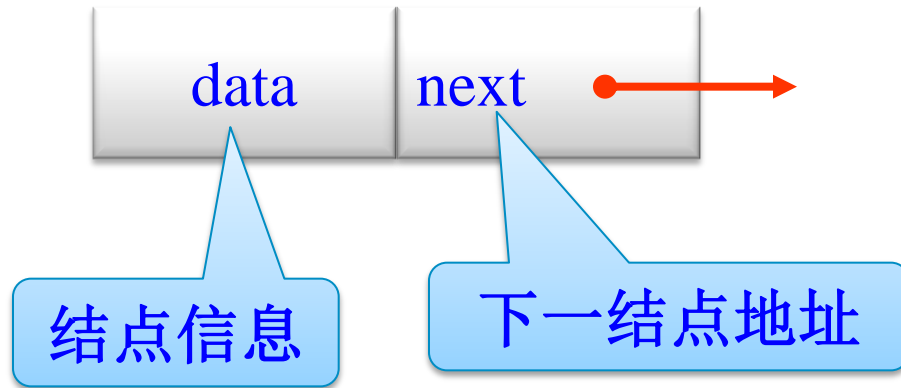


first==NULL;



2.3.1 线性链表

• 结点形式:





2.3.1 线性链表

• 存储结构类型定义:

链表是由一个个结点构成的，结点定义如下：

```
typedef struct node
```

```
{    DataType data;        //数据域
```

```
    struct node *next;    //指针域
```

```
} LNode, *LinkList;
```

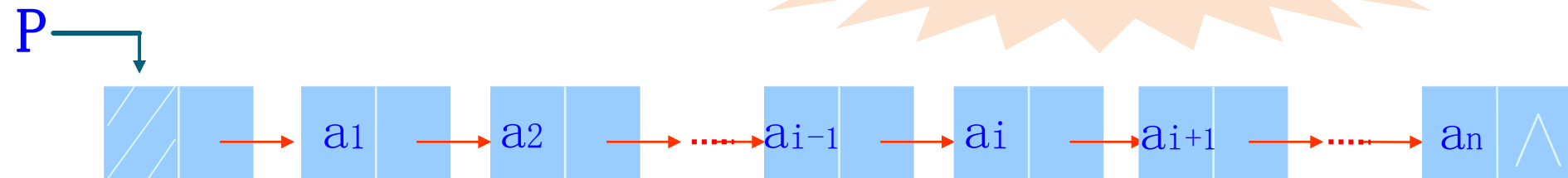
怎样在计算机上实现线性链表？



2.3.1 线性链表

- 单链表的计算机实现:

约定用带头结点的线性链表
存储线性表



建立链表 (C语言用函数: `malloc()`和`free()`)

C++ 中用 操作符`new` 和`delete`

通常, 在设有“指针”数据类型的高级语言中均存在与其对应的过程或函数。

下面以C语言为例



2.3.1 线性链表

- 建空线性链表:

算法:

```
Linklist create_head ( )
```

```
{Linklist head;
```

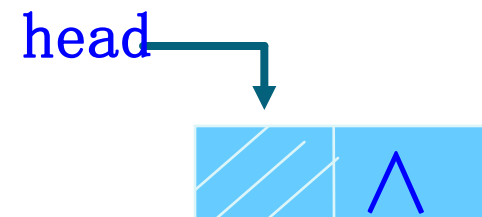
```
head = (linklist)malloc(sizeof(linklist));
```

```
//作用是由系统生成一个linklist型的结点，将该节点的起始位置赋给指针变量head
```

```
head->next = null;
```

```
Return (head);
```

```
}
```





2.3.1 线性链表

- 建立单链表:

- 1、头插法建表

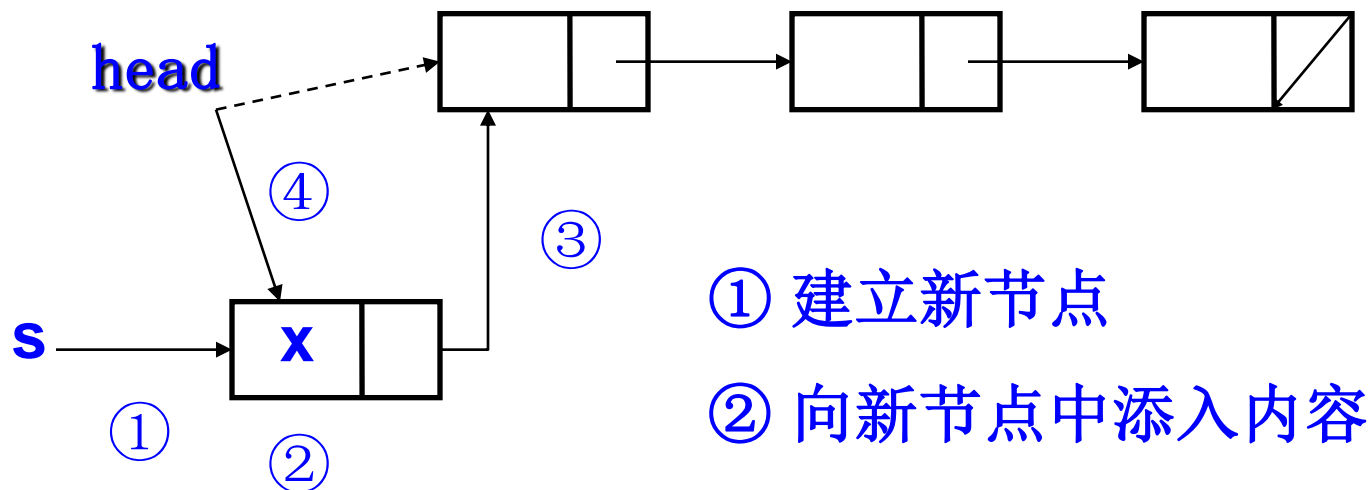
该方法从一个空表开始，重复读入数据，生成新结点，将读入数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上，直到读入结束标志为止。



2.3.1 线性链表

- 建立单链表:

头插法建立单链表



- ① 建立新节点
- ② 向新节点中添入内容
- ③ 使新节点指向链首
- ④ 改变头指针

图例



2.3.1 线性链表

- 建立单链表---头插法:

头插法建立单链表

```
Linklist Createlist()
{char ch;  Linklist *s, *r;
  head=(LinkList)malloc(sizeof(Linklist));
  head=NULL;
  while (ch=getchar())!='\n' )
  { s=(Linklist *)malloc(sizeof(Linklist));
    s->data=ch;
    s->next=head;
    head=s;}
  return(head);      }
```




2.3.1 线性链表

- 建立单链表---头插法:

注意:

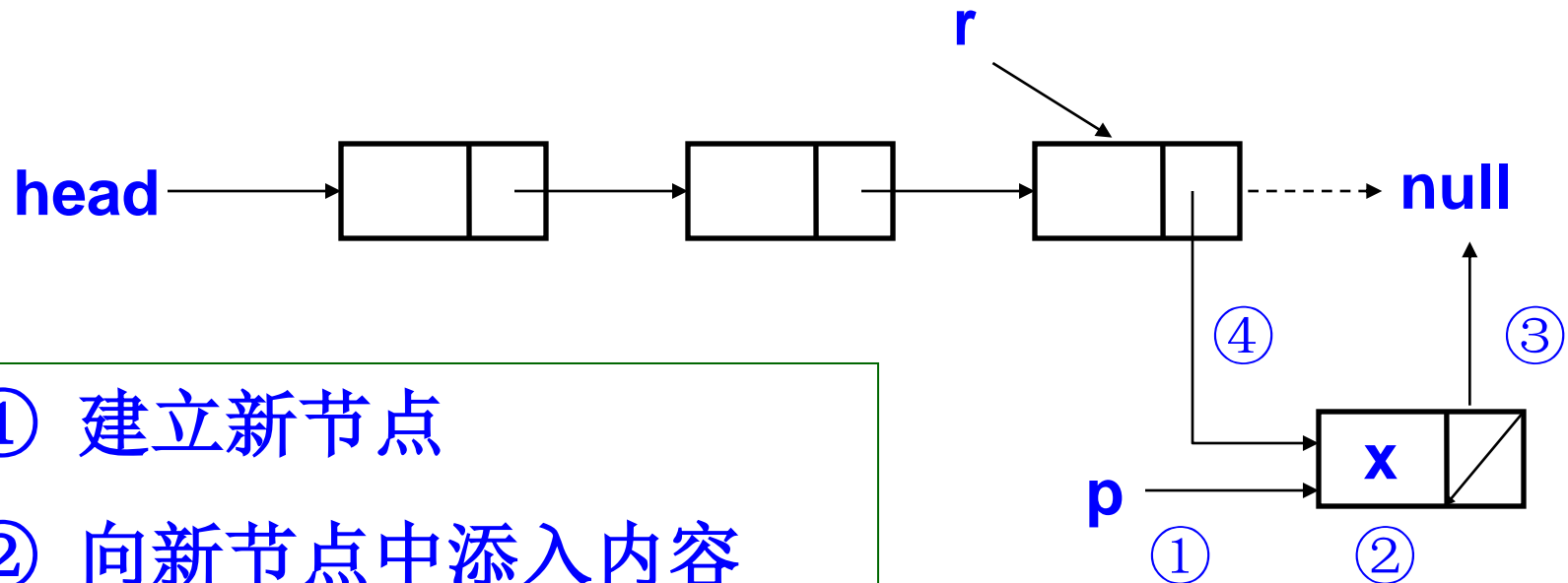
- 在头插法建立的单链表，生成的链表中结点的次序与输入的顺序相反。

如何解决该问题？



2.3.1 线性链表

- 建立单链表---**尾插法建立单链表**:



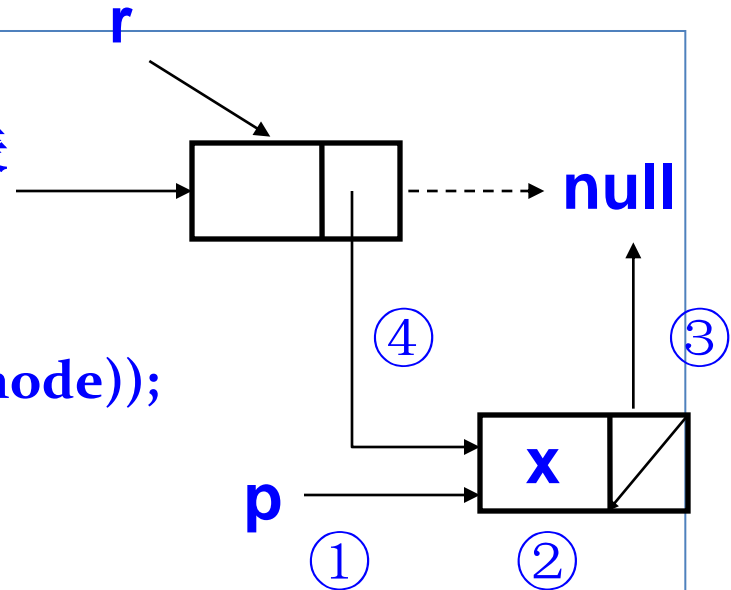


2.3.1 线性链表

• 建立单链表---尾插法建立单链表:

```

linklist creator( )
{.....//不带头结点的尾插法建立单链表
  head=NULL;r=NULL;
  while((ch=getchar())!= '\n' ){
    p=(listnode *)malloc(sizeof(listnode));
    p->data=ch;
    p->next=null
    if(head=NULL)
      { head=p; r=head;} //空表
    else {r->next=p; r=p; } //r为表尾指针
    return(head);}
  
```





2.3.1 线性链表

- 建立单链表---尾插法建立单链表:

```
LinkList Createlist(void)
```

```
{ ListNode *s, *r;
```

```
    Linklist head=(LinkList)malloc(sizeof(ListNode));
```

```
    char ch;
```

```
    r=head;
```

```
    while (ch=getchar())!='\n' )
```

```
    {      s=(ListNode *)malloc(sizeof(ListNode));
```

```
        s->data=ch;
```

```
        r->next=s;
```

```
        r=s;      }
```

```
    r->next=NULL;
```

```
    return(head); }
```

时间复杂度
为 $O(n)$

带头结点的尾插法建立单链表



2.3.1 线性链表

- 建立单链表---方法分析：
 - 带头结点单链表的优点：
 - 由于第一个结点的地址被存放在头结点的next域中，所以链表中每个接点的操作相同，无需特殊处理；
 - 空表和非空表的处理统一了。



2.3.1 线性链表

- 单链表的查找:

- (1) 按序号查找

- 在链表中，即使知道被访问结点的序号，也不能像顺序表那样直接按序号访问结点，只能从链表的头指针出发，顺着链表往下搜索。



2.3.1 线性链表

- 单链表的查找---按序号查找

```
ListNode * GetNode(LinkList head,int i)
```

```
{int j; ListNode *p; p=head;j=0;
```

```
while (p->next&& j<i){
```

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

```
    j++;}
```

```
if (i==j) return p;
```

```
else return NULL;}
```

时间复杂度为 $O(n)$



2.3.1 线性链表

- 单链表的查找---**按值查找**
 - 按值查找是在链表中，查找是否有结点值等于给定值**key**的结点：
 - 若有的话，则返回首次找到的其值为**key**的结点的存储位置；否则返回**NULL**。
 - 查找过程从开始结点出发，顺着链表**逐个**把要查找的结点的值和给定值**key**作比较。其算法如下：



2.3.1 线性链表

- 单链表的查找---按值查找（方法1）

```
linkList *Locate(LinkList head,DataType key)
```

```
{ Listlist *p;
```

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

```
  while (p!=Null&&P->data!=key)
```

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

```
  return p;
```

```
}
```

时间复杂度为 $O(n)$



2.3.1 线性链表

- 单链表的查找---按值查找（方法2）

```
position LOCATE ( elementtype x, LIST L )
{
    position p ;
    p = L ;
    while ( p→next != NULL )
        if ( p→next→element == x )
            return p ;
        else
            p = p→next ;
    return p ;
}
```



2.3.1 线性链表

- 单链表的查找---查找结点P的后继结点

```
position NEXT ( position p, LIST L )
```

```
{ position q ;
```

```
  if ( p→next == NULL )
```

```
    error ( “不存在后继元素” ) ;
```

```
  else
```

```
    { q = p→next;
```

```
      return q ;
```

```
    }
```

```
}//这里注意是结点p，而不是位置
```



2.3.1 线性链表

- 单链表的查找---查找结点P的前驱结点

position PREVIOUS (position p, LIST L)

{ position q ;

if (p == L→next) // L→next为第一个元素

error (“不存在前驱元素”) ;

else

{ q = L ; //顺着表头往下走

while (q→next != p) q = q→next ;

return q ; }

} //没有指向前驱的指针，所有要从头开始



2.3.1 线性链表

- 单链表的查找---查找表尾结点

```
position END ( LIST L)
{
    position q ;
    q = L ;
    while ( q→next != NULL )    q = q→next ;
    return ( q ) ;
};
```

- 单链表的查找---查找表头结点

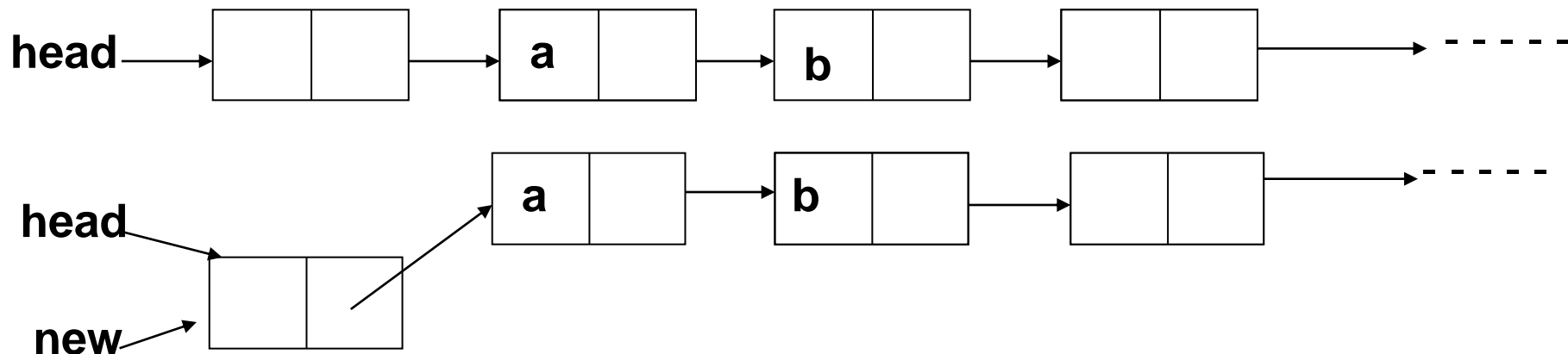
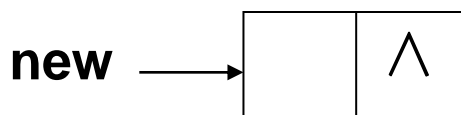
```
position FIRST ( LIST L)
{
    return L;
}
```



2.3.1 线性链表

- 单链表插入操作（分两种情况来看，有头结点和无头结点）

表头插入（无头结点）（无头结点，直接赋值给head）



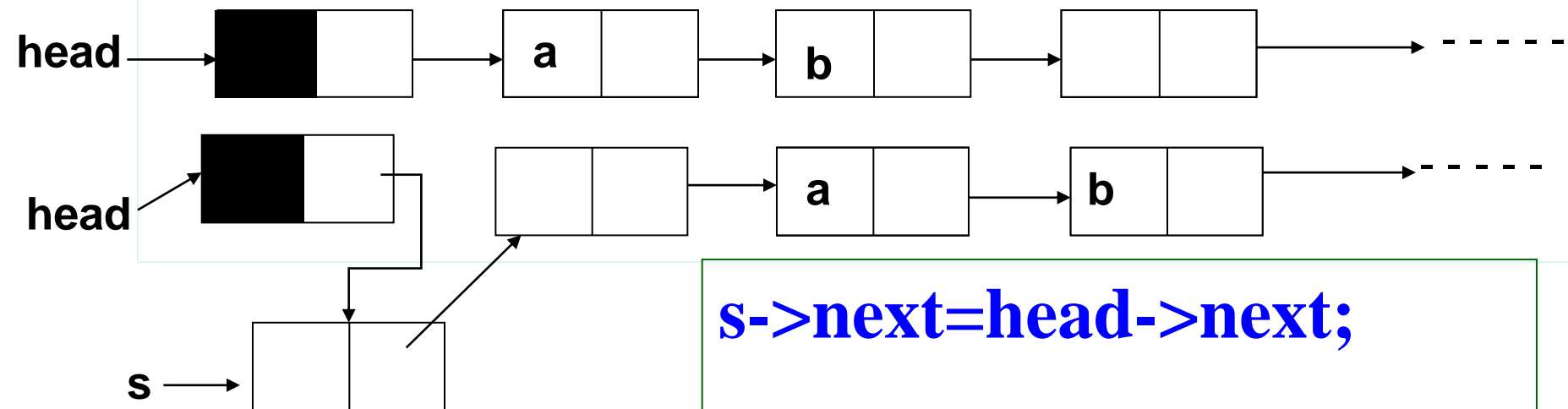
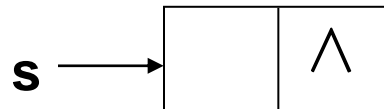
$new \rightarrow next = head; \quad head = new$



2.3.1 线性链表

• 单链表插入操作

表头插入（有头结点）



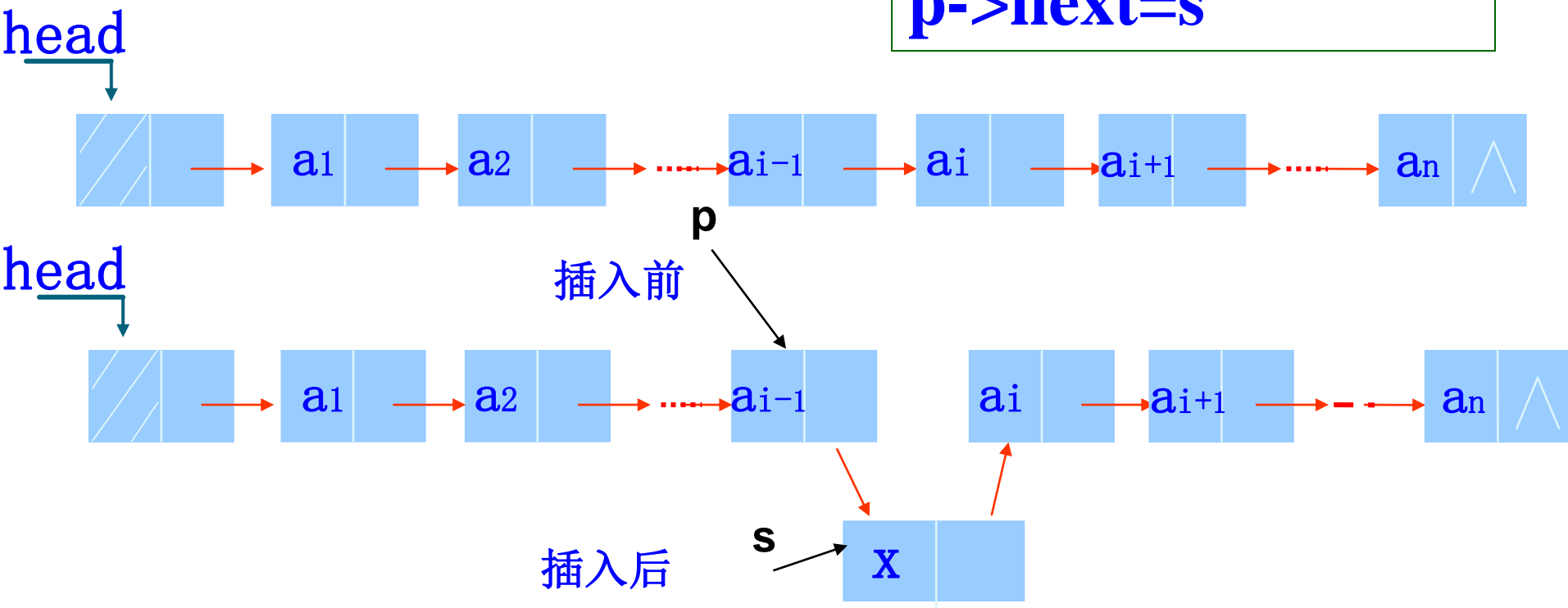
$s \rightarrow next = head \rightarrow next;$

$head \rightarrow next = s$



2.3.1 线性链表

- 单链表插入操作---在第*i*个元素结点之前插入

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




2.3.1 线性链表

- 单链表插入操作---插入运算

插入运算分为前插入运算和后插运算两种。

- 1) 前插既在所指结点之前插入新结点。
- 2) 后插入既在所指结点之后插入新结点。

前插运算：首先要建立一个新结点，但如何找到所指结点的前驱结点q呢？

一般情况，可从链表头进行查找。



2.3.1 线性链表

- 单链表插入操作---按位置插入结点

线性链表的插入Insert(L,x,i)

插入操作主要步骤:

- 1) 查找链表L的第 $i-1$ 个结点;
- 2) 为新元素建立结点;
- 3) 修改第 $i-1$ 个结点的指针和新结点指针完成插入;



2.3.1 线性链表

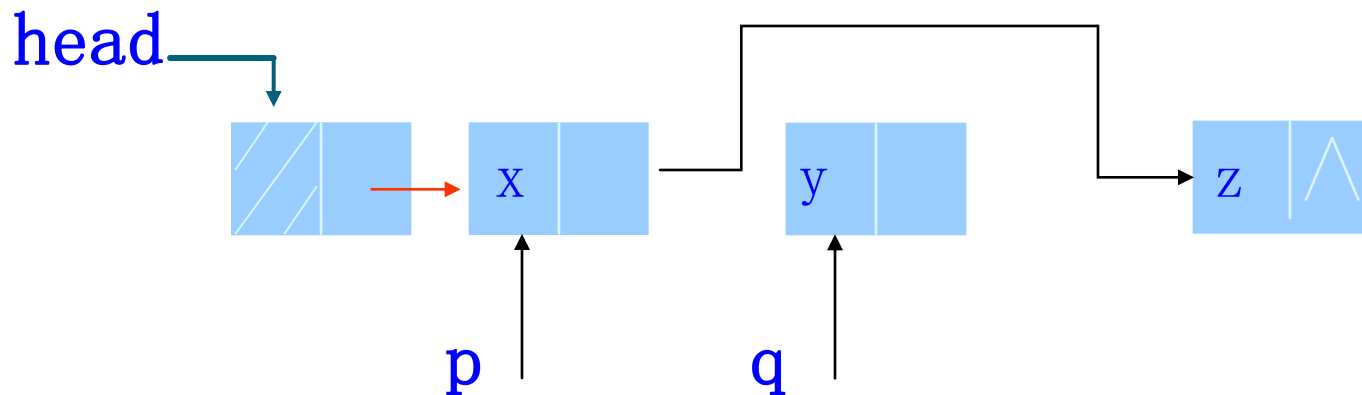
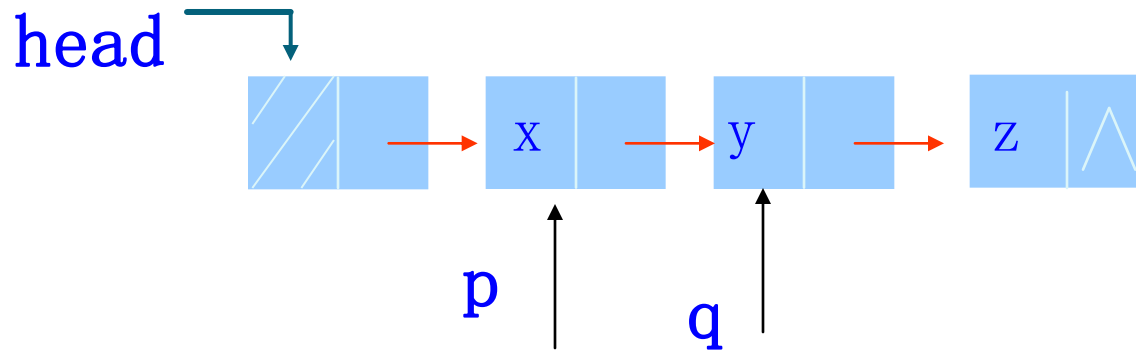
- 单链表插入操作---按位置插入结点
- ```
void Insertlist(LinkList head,DataType x, int i)
{ ListNode *p;
 p=GetNode(head,i-1);//得到i-1位置的结点
 if (p==NULL)
 Error("position error");
 s=(ListNode *(malloc(sizeof(ListNode)));
 s->data=x;
 s->next=p->next;
 p->next=s;
}
```





## 2.3.1 线性链表

- 单链表的删除





## 2.3.1 线性链表

- 单链表的删除---删除后继结点

**Deleteafter(p)//删除\*p的后继结点\*q**

**linklist \*p;**

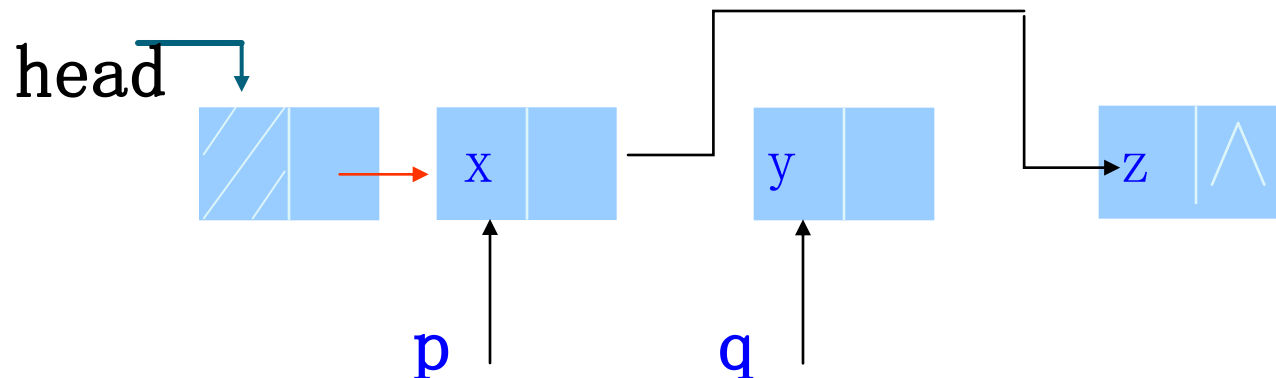
**{linklist \*q;**

**p->next= q;**

**p->next=q->next;**

**free(q);**

**}**





## 2.3.1 线性链表

- 单链表的删除---删除指定位置结点

**Delete(L,i):删除第i个结点**

主要步骤:

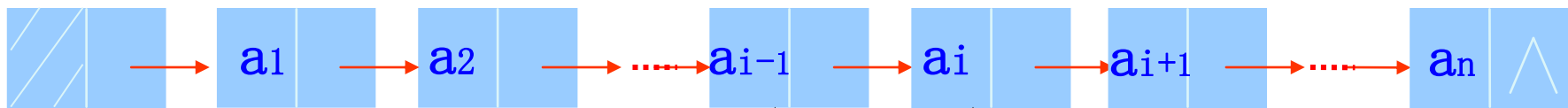
- 1) 查找链表的第  $i-1$  个元素结点,使指针  $p$  指向它, 将指针  $q$  指向第  $i$  个结点;
- 2) 修改第  $i-1$  个元素结点指针, 使其指向第  $i$  个元素结点的后继;
- 3) 回收  $q$  指针所指的第  $i$  个结点空间;



## 2.3.1 线性链表

- 单链表的删除---删除指定位置结点

head

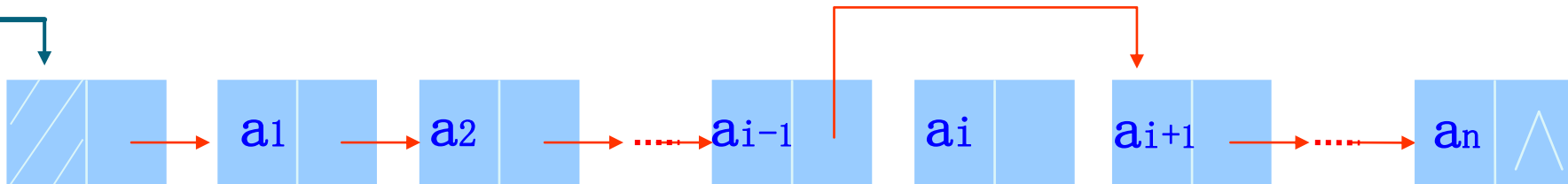


p

r

删除前

head



p

r

删除后



## 2.3.1 线性链表

- 单链表的删除---删除指定位置结点

✓ 在线性链表中删除第i个结点—方法1

■ **void Deletelist(LinkList head, int i)**

**{ ListNode \*p,\*r;**

**p=GetNode (head,i-1);**

**if(p==NULL||p->next==NULL)**

**Error(“position error”);**

**r=p->next; p->next=r->next;**

**free(r);**

**}**





## 2.3.1 线性链表

- 单链表的删除---删除指定位置结点

✓ 在线性链表中删除第*i*个结点—方法2

■ **void Deletelist(LinkList head, int i)**

**{ ListNode \*p,\*r;**

**p=GetNode (head,i-1);**

**if(p != NULL && p->next != NULL)**

**deleteafter(p) //删除p的后继**

**else**

**Print("error")**

**}**



## 单链表小结

### • 优点

① 插入和删除操作容易实现；

✓ 插入时，向系统申请一个结点的存储空间；删除时释放结点的存储空间。是一种动态存储结构。

② 建立空表时，不需要考虑存储空间。线性链表的结点是随需要而定的；

③ 容易合并或分离线性链表；

✓ 由于结点之间是用指针连接的，所以，对线性链表进行合并或分离操作都比较方便，只要改变指针的指向。



# 单链表小结

## • 缺点

- ① 相对顺序表而言，对线性链表中的任一结点进行操作复杂，实现的是顺序存取。
  - 单链表结点中只有一个指向其后继的指针，这使得单链表只能从头结点依次顺序地向后遍历。
- ② 寻找线性链表中的任一结点的前驱比较困难。
  - 若要访问某个结点的前驱结点（删除、插入操作时），只能从头开始遍历，访问后继结点的时间复杂度为 $O(1)$ ，访问前驱结点的时间复杂度为 $O(n)$ 。
- ③ 在单链表的最后一个元素后插入一个元素时，需遍历整个链表。



# 单链表小结

- 顺序表与链表的比较

## 顺序存储

固定，不易扩充  
随机存取  
插入删除费时间  
估算表长度，浪费空间

## 比较参数

←表的容量→  
←存取操作→  
←时间→  
←空间→

## 链式存储

灵活，易扩充  
顺序存取  
访问元素费时间  
实际长度，节省空间



## 2.3.1 线性链表

**例** 遍历线性链表，按照线性表中元素的顺序，依次访问表中的每一个元素，每个元素只能被访问一次。

```
Void VISITE(LIST L)
{ position p ;
 p = L→next ;
 while (p != NULL)
 { cout << p→element ;
 p = p→next ;
 }
}
```



## 2.3.1 线性链表

**例** 已知：la，lb为单链表且元素按值非递减排列。合并后得到新单链表lc也按值非递减排列。

```
Void mergelist(linklist &la,linklist &lb,linklist &lc)
{ linklist *pa,*pb,*rc;
 pa=la->next;pb=lb->next;
 lc=rc=la;//用la的头结点作为lc的头结点
 while(pa!=NULL&&pb!=NULL)
 { if(pa->data<=pb->data)
 {rc->next=pa;rc=pa;pa=pa->next;}
 //rc走到pa指向的结点，pa往下走
 }
```

两个有序链表的合并



## 2.3.1 线性链表

**例** 已知：la，lb为单链表且元素按值非递减排列。合并后得到新单链表lc也按值非递减排列。

两个有序链表的合并

**else**

```
{rc->next=pb;rc=pb;pb=pb->next;}
```

```
}
```

```
rc->next=pa?pa:pb;
```

```
//剩下的表结点连在rc的后面
```

```
free(la);free(lb); }
```



## 2.3.1 线性链表

循环链表

线性链表（单链表）  
如何从表尾找到  
表头？

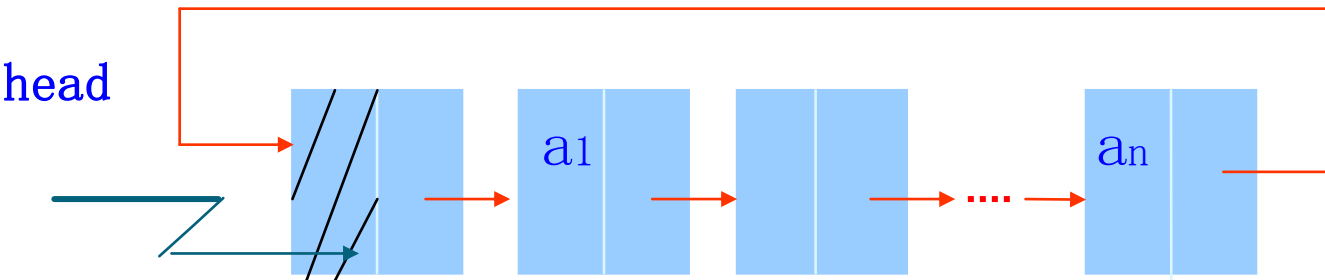




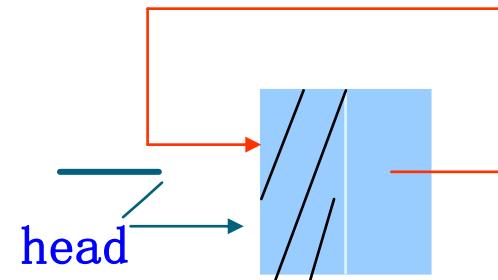
## 2.3.2 循环链表

- 循环链表的概念

- 循环链表是线性表的另一种链式存储结构，它的**特点**是**将线性链表的最后一个结点的指针指向链表的第一个结点**,整个链表**形成一个环**。因此，从表中任一结点出发均可找到表中其它结点。



(a) 非空表



(b) 空表




## 2.3.2 循环链表

- 循环链表与线性链表
  - 循环链表的操作和线性链表基本一致。

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

线性链表:  $p \rightarrow \text{next} == \text{NULL}$

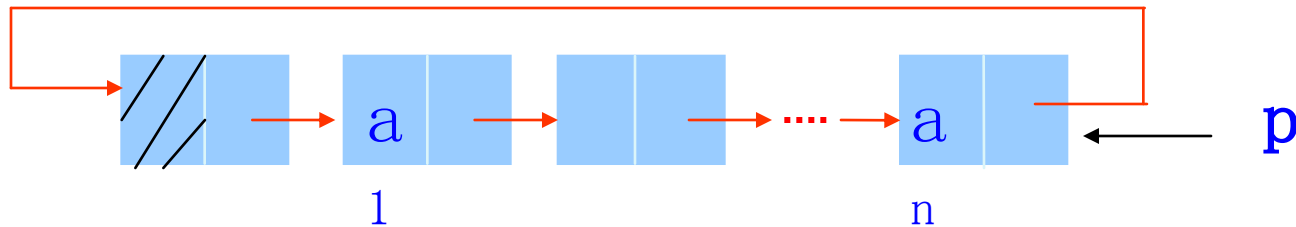


不同是判断表尾的  
条件不同!



## 2.3.2 循环链表

- 给出尾指针的循环链表
  - 对循环链表，有时不给出头指针，而是给出尾指针。



给出尾指针的循环链表



## 2.3.2 循环链表

- 查找：在循环链表中查找值为x的结点的算法

```
linklist *locatenode(linklist *L,datatype x)
```

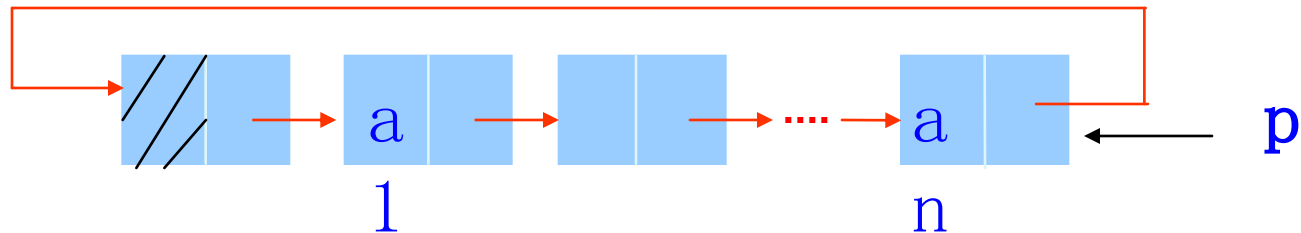
```
{ linklist *p;
```

```
 p=L->next; //带头结点
```

```
 while(p->next!=head&& p->data!=x)
```

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

```
 return(p);}
```

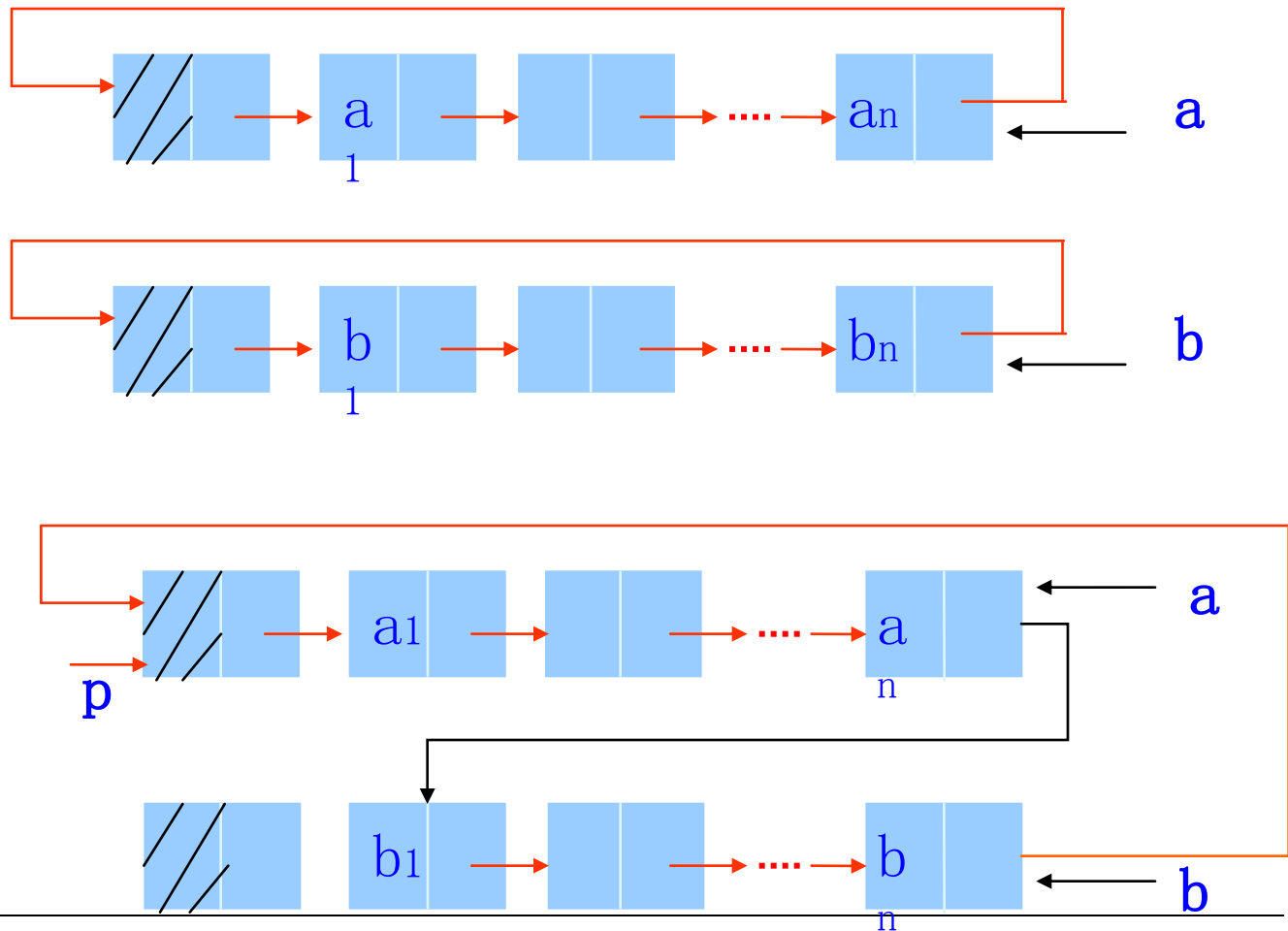


给出尾指针的循环链表



## 2.3.2 循环链表

- 例：将两个线性表链接成一个线性表





## 2.3.2 循环链表

- **注意：**由于单链表结点的定义，使得在单链表的操作中，对指定结点的前驱结点操作需要从表头开始查找。

如何解决该问题？

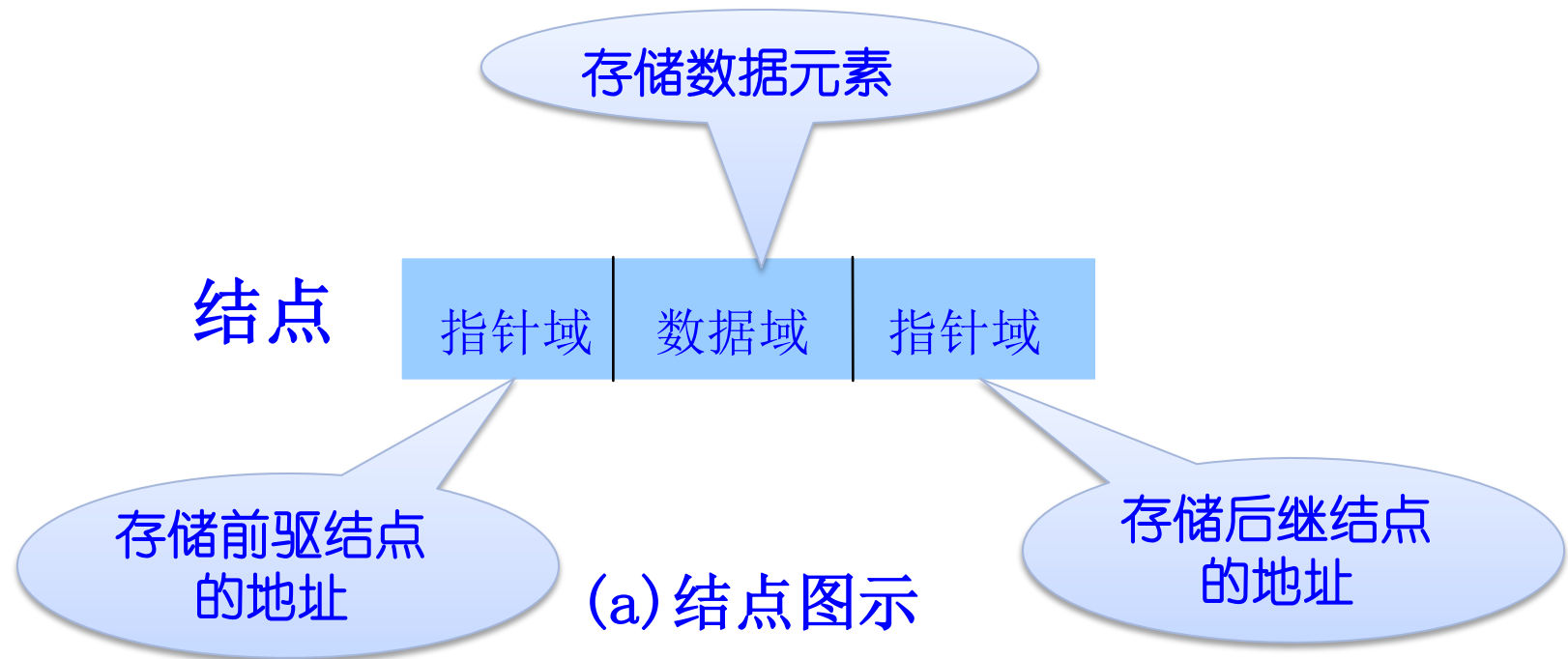
双向链表



## 2.3.3 双向链表

- 双向链表的概念

- 双向链表中，每个结点有**两个指针域**，一个指向直接后继元素结点，另一个指向直接前驱元素结点。





## 2.3.3 双向链表

- 双向链表的结点结构定义

```
Typedef Struct dnode
{ DataType data;

 struct dnode * prior, *next;

}dlistlist;

dlinklist *head;
```





## 2.3.3 双向链表

- 插入操作算法 --后插
  - 将值为 $x$ 的结点插入 $p$  结点之后的过程





## 2.3.3 双向链表

- 插入操作算法 – 前插(将值为 $x$ 的结点插入 $p$  结点之前)

```
void Dinsertbefore(dlinklist *p,DataType x)
```

```
{dlinklist *s=malloc(sizeof (dlinklist));
```

```
s->data=x;
```

```
s->prior=p->prior; //(1)
```

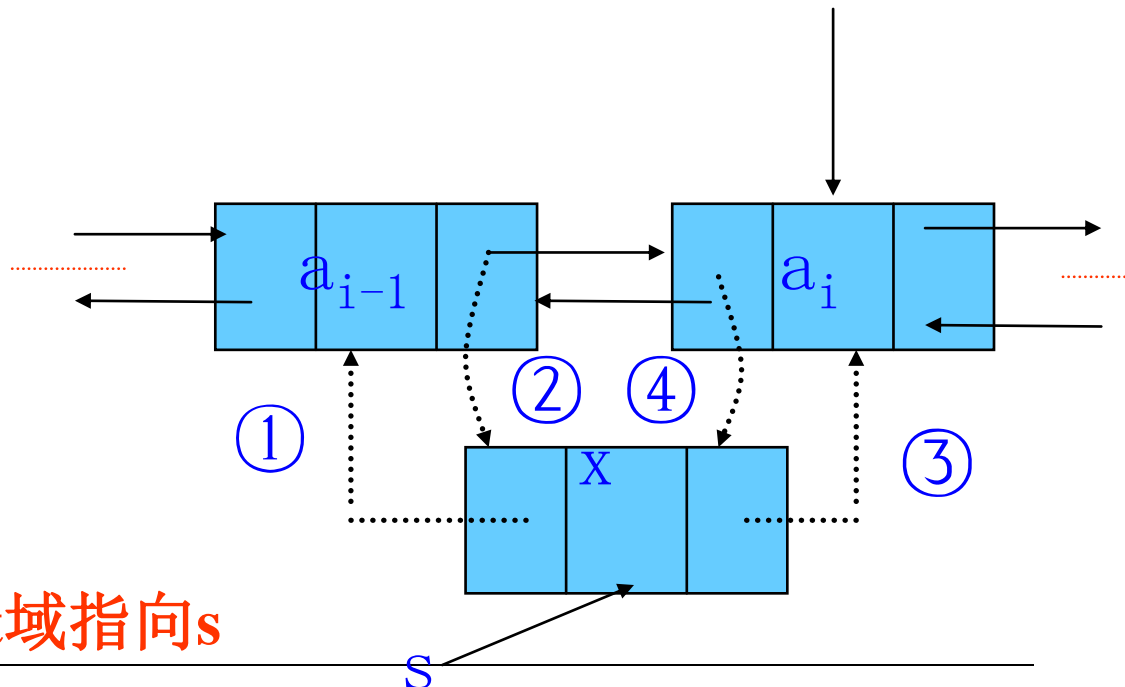
```
p->prior->next=s; //(2)
```

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

```
p->prior=s; //(4)
```

```
}
```

(2)是 $p$ 的前驱结点的 $next$ 域指向 $s$





## 2.3.3 双向链表

- 删除当前结点

....

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

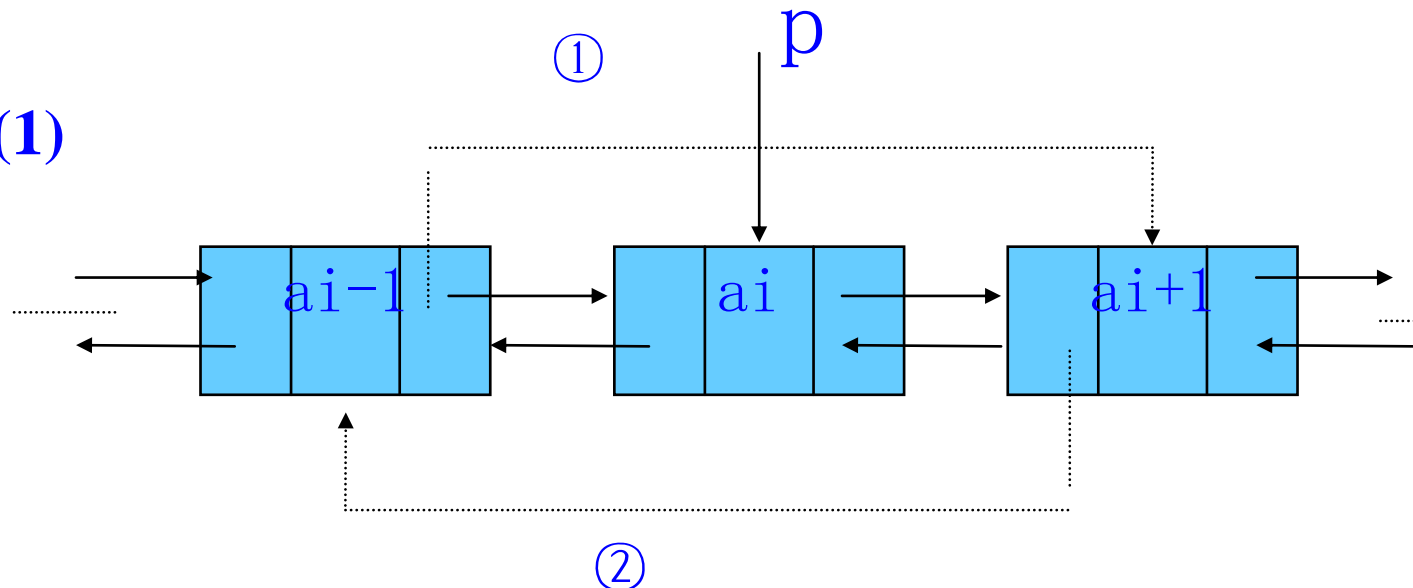
//p的前驱结点的next域指向

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

//p的后继结点的prior域指向

`free(p);`

//时间复杂度O(1)

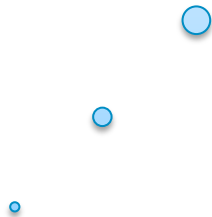




## 2.3.3 双向链表

- 删除后继结点

图示





## 2.3.3 双向链表

- 双向循环链表

- 双向环形链表的结构与双向链表结构相同，只是将表头元素的空previous域指向表尾，同时将表尾的空next域指向表头结点，从而形成向前和向后的两个环形链表，对链表的操作变得更加灵活。
- 很明显，在双向链表中不仅能直接找到结点的前驱，也能即刻找到结点的后继。
- 假设指针p指向双向链表中某个结点，则
$$p \rightarrow \text{prior} \rightarrow \text{next} = p$$
$$p \rightarrow \text{next} \rightarrow \text{prior} = p$$
- 操作特点：
  - “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。



图示



# 实际中应该怎样选取存储结构？

## ① 基于存储的考虑

- ✓ 对线性表的长度或存储**规模难以估计**时，**不宜采用顺序表**；链表**不用事先估计存储规模**，链表的存储密度较低，显然链式存储结构的存储密度是小于1的。

## ② 基于运算的考虑

- ✓ 在**顺序表**中**按序号访问** $a_i$ 的时间复杂度为 **$O(1)$** ，而**链表**中按序号访问的时间复杂度为 **$O(n)$** 。所以如果经常做的运算是按序号访问数据元素，显然顺序表优于链表。
- ✓ 在**顺序表**中做插入、删除操作时，**平均移动表中一半的元素**，当数据元素的信息量较大且较长时，这一点是不应忽视的；在链表中插入、删除操作时，虽然也要找插入位置，但**操作主要是比较操作**，从这个角度考虑后者优于前者。



# 实际中应该怎样选取存储结构？

## ③ 基于环境的考虑

✓ **顺序表**容易实现，任何高级语言中都有**数组类型**；**链表**的操作是基于**指针**的，相对来讲，前者较为简单，这也是用户考虑的一个因素。

■ 总之，两种存储结构各有长短，选择哪一种由实际问题的主要因素决定。通常较稳定的线性表选择顺序存储，而频繁做插入、删除操作的线性表（即动态性较强）宜选择链式存储。

■ **注意：**只有熟练掌握顺序存储和链式存储，才能深刻理解它们各自的优缺点。



# 本章小结

- ✓ 本章学习了线性表的顺序存储结构——顺序表
- ✓ 链式存储结构-线性链表,循环链表,双向链表,
- ✓ 在这两种存储结构下如何实现线性表的基本操作。
- ✓ 如何在计算机上存储线性表, 如何在计算机上实现线性表的操作。
- ✓ 在不同的存储结构下, 线性表的同一操作的算法是不同的。
- ✓ 在顺序表存储结构下, 线性表的插入删除操作, 通过移动元素实现, 在线性链表存储结构下, 线性表的插入删除操作, 通过修改指针实现。
- ✓ 对于某一实际问题, 如何选择合适的存储结构, 如何在某种存储结构下实现对数据对象的操作。