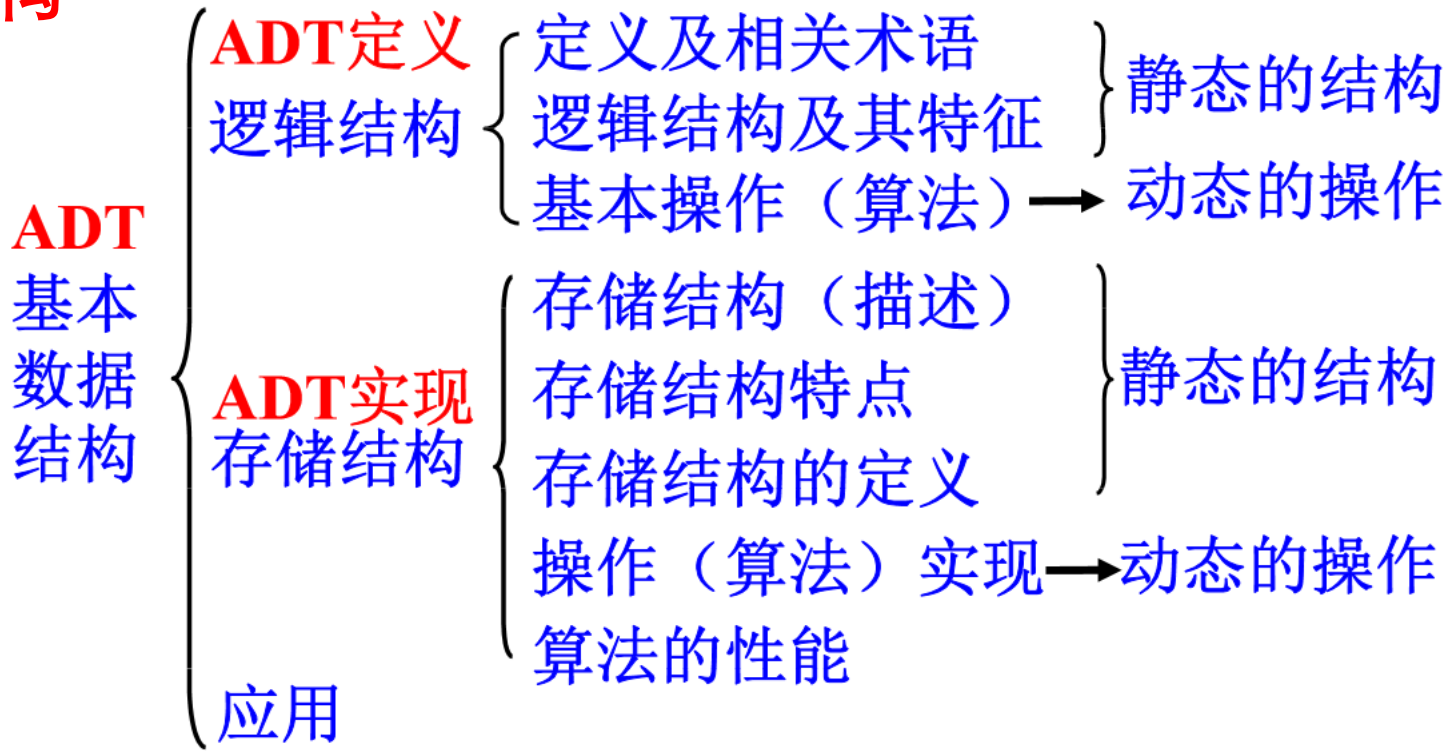
线性结构

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



第2章 线性表

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

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。

2. 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。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

2.1 线性表概念及基本操作

- 【定义】线性表是由n(n≥0)个相同类型的数据元素组成的有限序列。

记为：

$$(a_1, a_2, a_3, \ldots\ldots a_{i-1}, a_i, a_{i+1}, \ldots\ldots a_n)$$

逻辑结构：

其中：n为线性表中元素个数，称为线性表的长度；当n=0时，为空表，记为( )；

a_1为表中第1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…a_{i-1},a_i,a_{i+1}…(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；

线性表是有限的，也是有序的。

2.1 线性表概念及基本操作

数学模型： 线性表 LIST = (D , R)

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

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

操作： 设L的型为LIST线性表实例， e 的型为ElementType的元素实例， p 为位置变量。所有操作描述为：

① ListInsert(&L, p, e);
② LocateElem(L,e, Compare());
③ GetElem(L,p,&e);
④ ListDelete(&L,p,&e);
⑤ PriorElem(L,cur_e, &pre.e), NextElem(L,cur_e,&next_e);
⑥ ClearList(&L);
⑦ ListrFirst(L) ； ⑧ListEnd(L)； , …..

2.1 线性表概念及基本操作

为什么要实现对数据结构的基本操作？

①团队合作编程，你定义的数据结构要让别人能够很方便的使用（封装）
②将常用的操作/运算封装成函数，避免重复工作，降低出错风险

Tips：比起学会“How”，更重要的是想明白“Why”

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2-1】设计函数 Deleval（LIST &L, ElementType d)，其功能为删除 L 中所有值为 d 的元素。

```
Void Deleval( LIST  &L,  ElementType  d )
{  Position  p ;
   p = ListFirst( L ) ;
   while ( p  != ListEnd( L ) )
        {    GetElem(L, p,&e)
             if ( compare( e,d ) )
                ListDelete(&L,p, &e ) ;
             else
                NextList( L ,p,p) ;
        }
 }
```

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

- 线性表是具有相同数据类型的n（n≥0）个数据元素的有限序列
- 如何在计算机中存储线性表？
- 需要保存哪些信息？

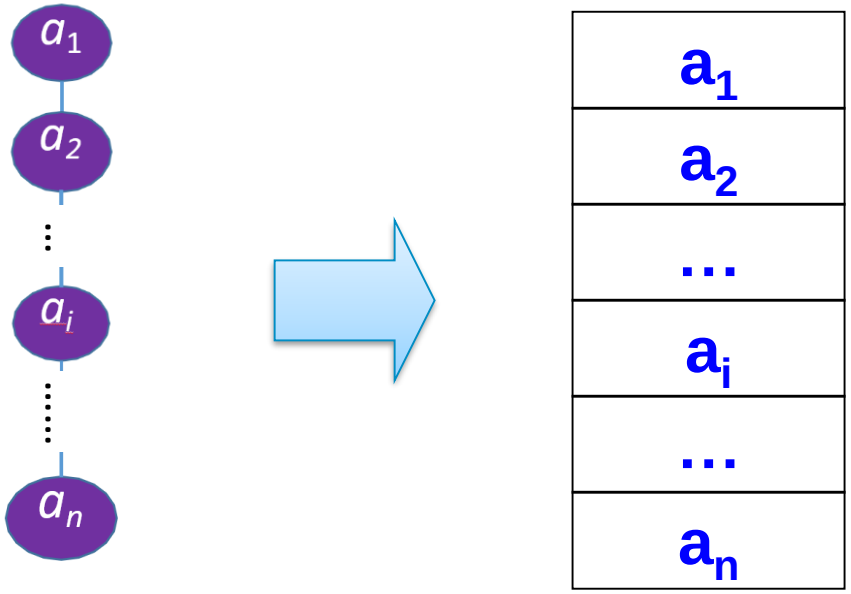
为了存储线性表，至少要保存两类信息：

（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
  - 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.

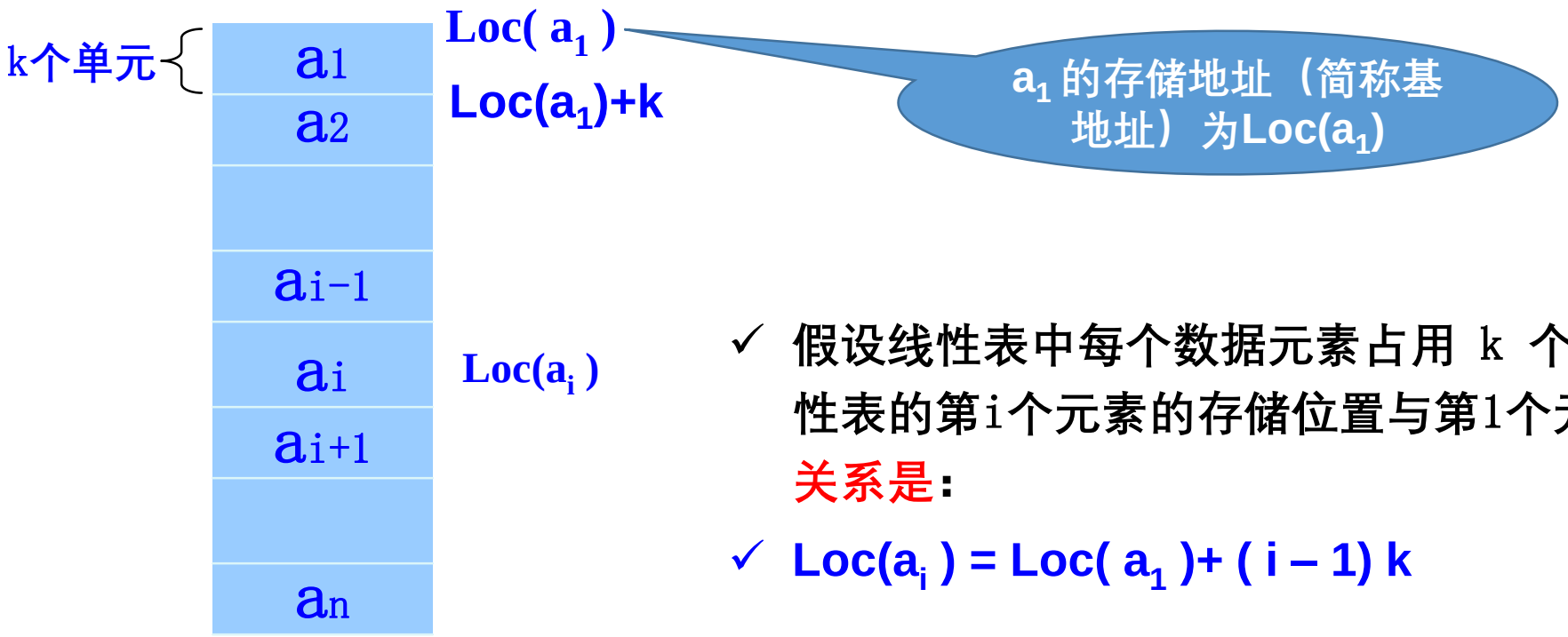


逻辑结构　　物理存储结构

✓ 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；

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**
  - 线性表是具有相同数据类型的n（n≥0）个数据元素的有限序列



- ✓ 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：
- ✓ $Loc(a_i) = Loc(a_1) + (i-1)k$

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，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。

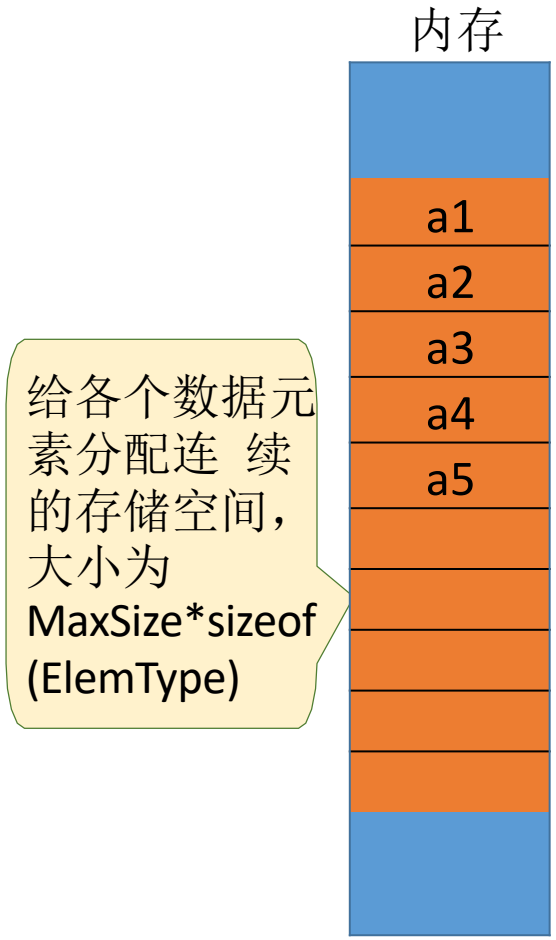
2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表
  - 顺序表的实现-静态分配：

```
#define MAXSIZE 10    //  数组容量

typedef  struct

 {  ElemType data[MAXSIZE];  //  数组域

    int  last;              //  线性表长域

 } Seqlist;             //  结构体类型名
```

位置类型：

typedef int position;



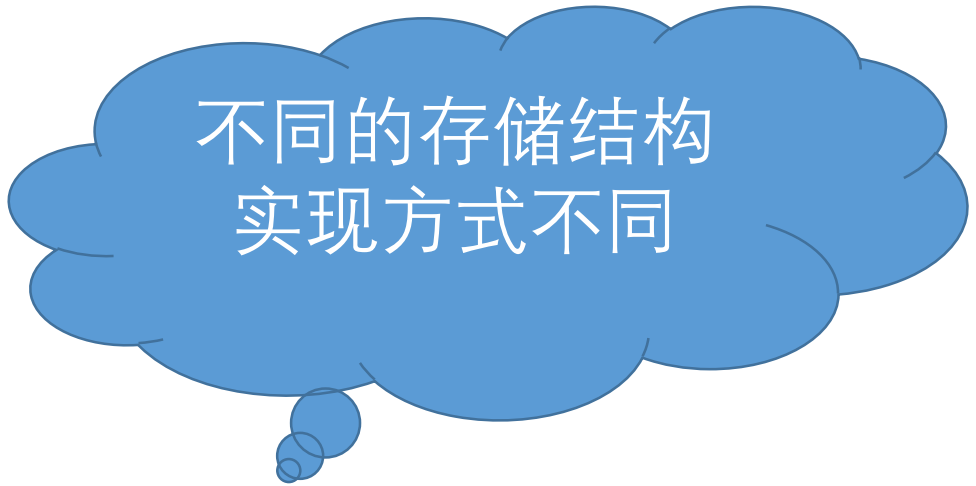
2.2 线性表的顺序存储和实现

- 线性表的顺序存储结构---顺序表
  - 顺序表的实现-动态分配：

```
#define LIST_INIT_SIZE      100       //初始分配空间
#define LISTINCREMENET  10        //分配增量
typedef struct{
    ElemType   *elem;                  //元素空间
    int  length;                       //表的长度
    int listsize;                      //当前容量
}AqList ;
```

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
  - 顺序表的基本操作（创销增删改查）：
    - 创建、销毁
    - 插入、删除
    - 查找、修改



2.2 线性表的顺序存储和实现

- 创建/初始化一个顺序表：

```
typedef  struct
{  int data[MAXSIZE];  //  数组域
   int  last;          //  线性表长域
} Seqlist;             //  结构体类型名
```

```
int Creat_Seqlist(Seqlist *L)
{
  printf("\n n= "); scanf("%d",&(L->last));    /* 线性表的数据个数*/
  if （L->last <0 || L->last >MAXSIZE ） { //检查Input的合法性
    printf(“Wrong input"); return 1;}
  for(int i=0; i<L->last; i++)  {
    printf("\n data= ");
     scanf("%d",&(L->data[i])); /* 输入数据*/
   } return 0;
 }
```

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
  - 顺序表的基本操作：
    - 创建、销毁
    - 插入、删除
    - 查找、修改

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:

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

成为表长为 n+1的表:

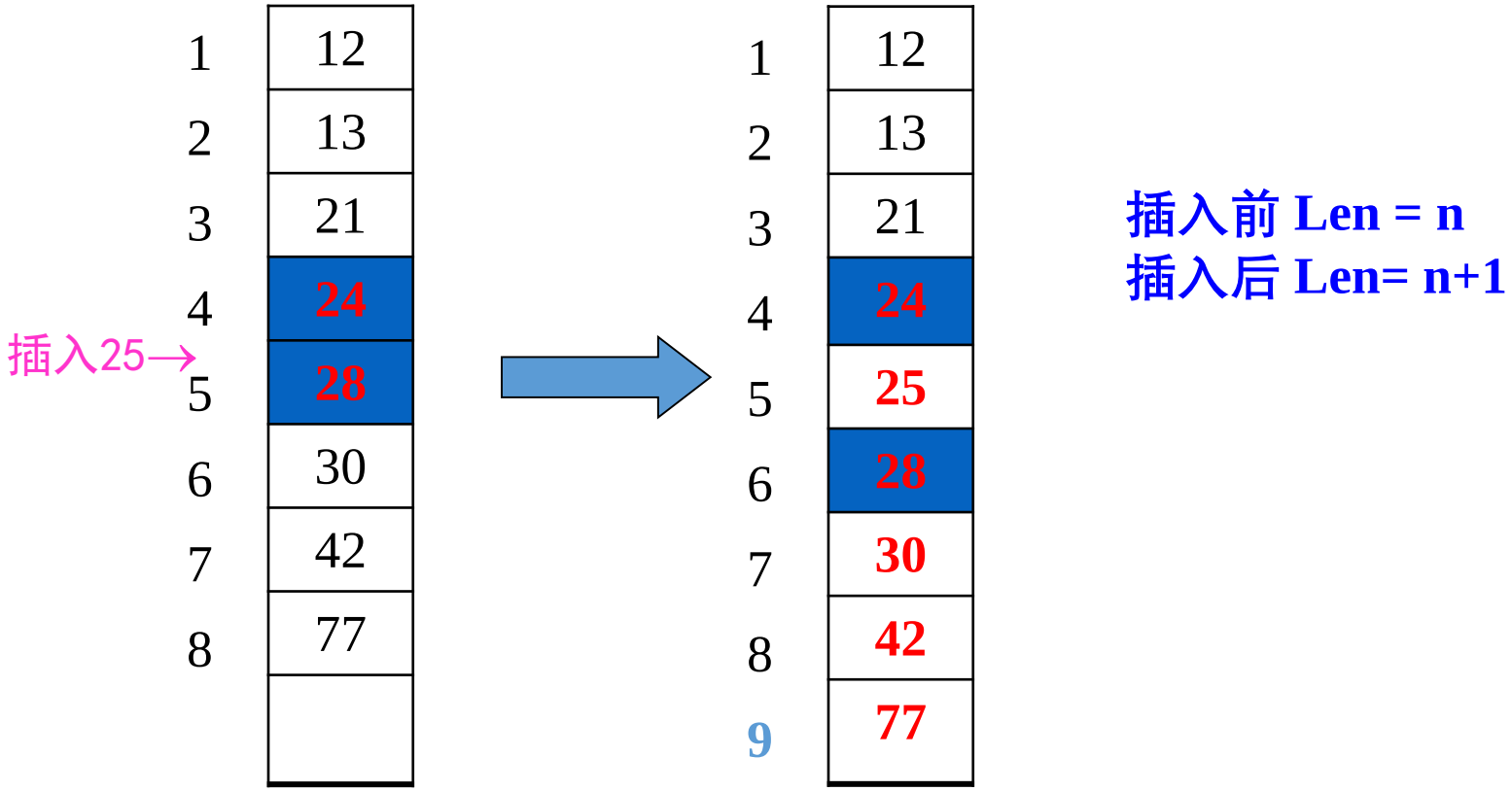
$(a_1, a_2, \ldots, a_{i-1}, x, a_i, a_{i+1}, \ldots, a_n)$。

i 的取值范围为1<=i<=n+1 ，

图例

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



插入前 Len = n
插入后 Len= n+1

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

实现步骤：(n为表长)

•将第n至第i 位的元素向后移动一个位置；

•将要插入的元素写到第i个位置；

•表长加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 都要向后移动一个位置，共需要移动 n-i+1个元素，而 i 的取值范围为： 1<= i<= 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：

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

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

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

  - 则平均移动数据元素的次数：

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

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
  - 元素插入位置的可能值：

    i= 1，2,……, n, n+1
  - 相应向后移动元素次数：

    n-i+1= n, n-1,……, 1, 0
  - 对n,n-1,……,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：

$$E_{in} = \frac{1}{n+1} \cdot \frac{n(n+1)}{2} = \frac{n}{2}$$

  - 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：

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

成为表长为 n-1 的线性表：

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

i 的取值范围为：1<=i<=n ，

图例

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 (L->last ==0 || (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：

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

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

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

2.2 线性表的顺序存储和实现

- 删除算法的时间性能分析
  - 元素删除位置的可能值： i= 1, 2,……, n
  - 相应向前移动元素次数： n-i= n-1, n-2,……, 0
  - 对n-1,……,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：

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

  - 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

- 线性表的顺序存储结构---顺序表
  - 顺序表的基本操作：
    - 创建、销毁
    - 插入、删除
    - 查找、修改

2.2 线性表的顺序存储和实现

- 顺序表的查找-按值查找

✓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:

图例

```
int Location_SeqList(SeqList *L, DataType x)
{  int i=1;
        while(i<=L.last && L->data[i-1]!= x)i++;
        if (i>L->last)  return -1;
            else  return i;}
```

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---按位序查找

RETRIEVE：返回L中位置p的元素

```
elementtype  RETRIEVE ( position  p , LIST  L )
    {  if  ( p<1 || 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 -1) )
            error ( “后继元素不存在” ) ;
        else
            return ( p + 1 );
    }//时间复杂性:O(1)
```

2.2 线性表的顺序存储和实现

- 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 拓展容量不方便

④ 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

2.2 线性表的链式存储和实现

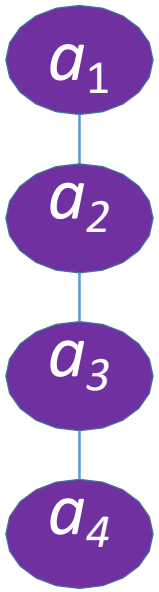
- 线性表是具有相同数据类型的n（n≥0）个数据元素的有限序列
- 如何在计算机中存储线性表？
- 需要保存哪些信息？

为了存储线性表，至少要保存两类信息：

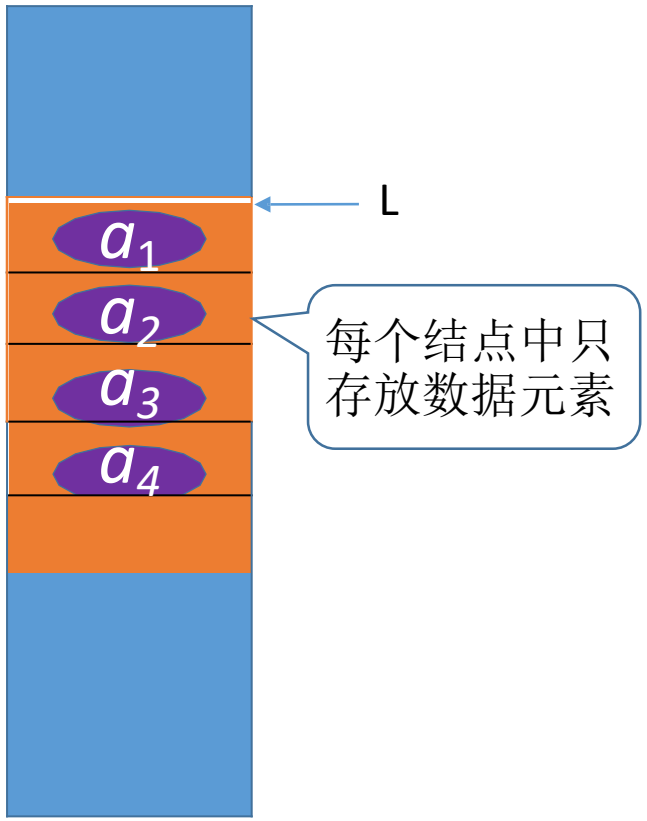
（1）线性表中的数据元素；

（2）线性表中数据元素的顺序关系。

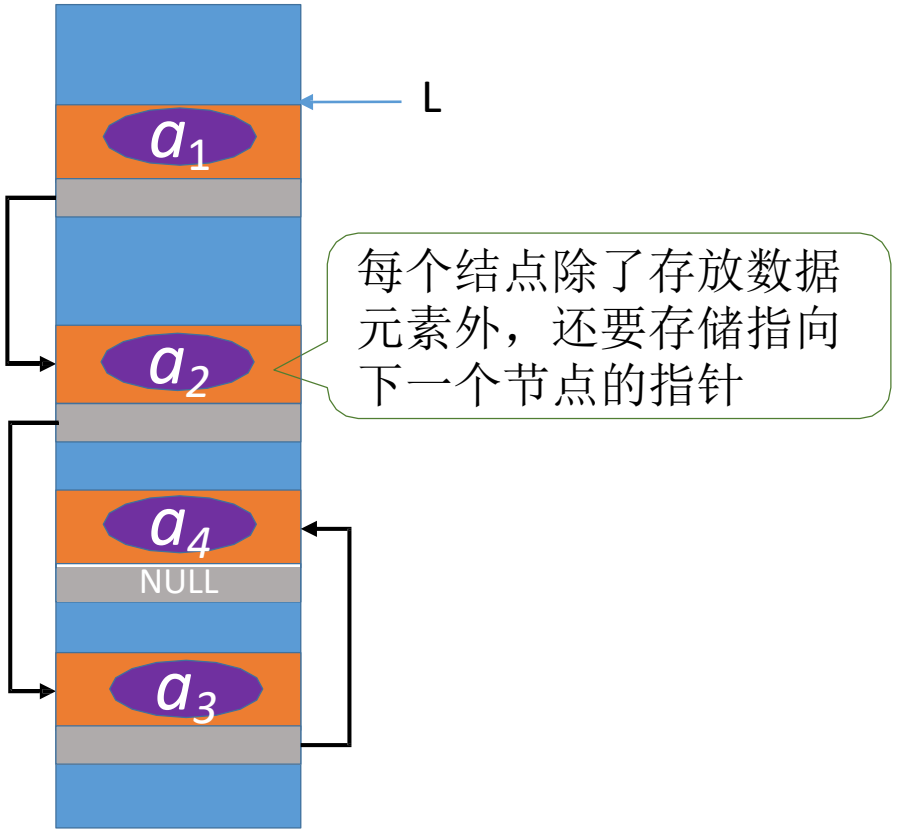
逻辑结构：
线性表



顺序表
（顺序存储）



链表
（链式存储）



优点：可随机存取，存储密度高
缺点：要求大片连续空间，改变容量不方便

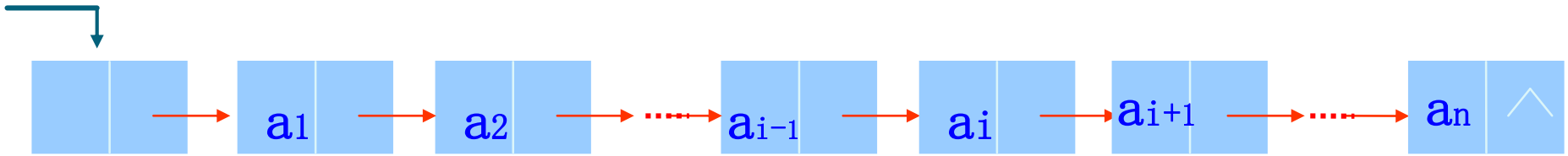
优点：不要求大片连续空间，改变容量方便
缺点：不可随机存取，要耗费一定空间存放指针

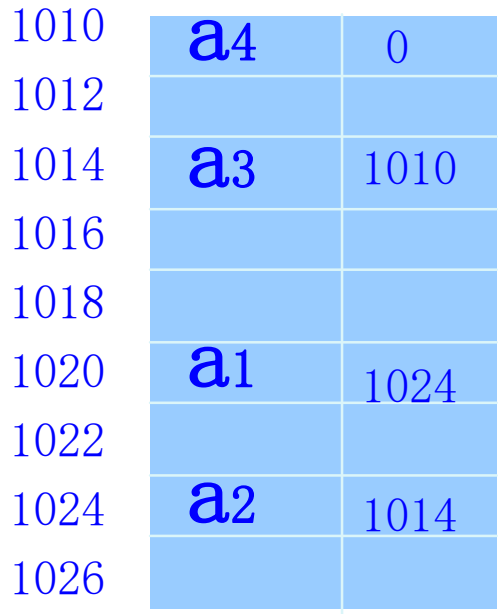
2.3.1 线性链表

- 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

定义：一个线性链表由若干个结点组成，每个结点均含有两个域：存放元素的数据域和存放其后继结点地址的指针域，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。

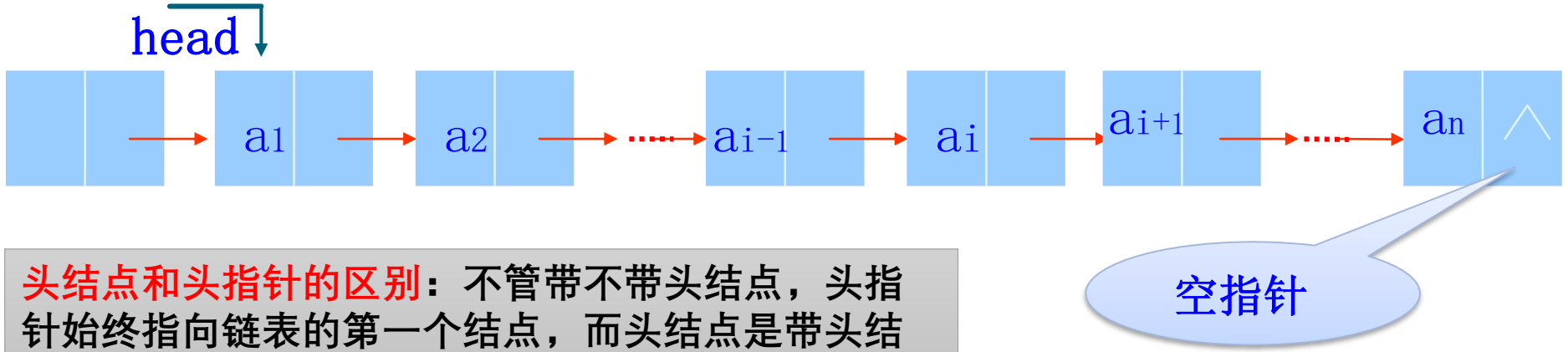




用线性链表存储线性表时，数据元素之间的关系是通过保存直接后继元素的存储位置来表示的

2.3.1 线性链表

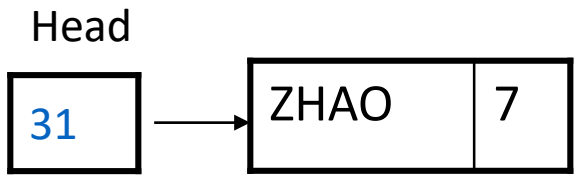
- 线性链表有关术语：
  - 头指针：用于存放线性链表中第一个结点的存储地址；通常用头指针来标识一个单链表。
  - 空指针：不指向任何结点，线性链表最后一个结点的指针通常是空指针；
  - 头结点：线性链表的第一元素结点前面的一个附加结点，称为头结点；
  - 带头结点的线性链表：第一元素结点前面增加一个附加结点(头结点)的线性链表称为带头结点的线性链表；



头结点和头指针的区别：不管带不带头结点，头指针始终指向链表的第一个结点，而头结点是带头结点链表中的第一个结点，结点内通常不存储信息。

例： 一个线性表的逻辑结构为：（ZHAO,QIAN,SUN,LI,ZHOU,WU,ZHENG,WANG），其存储结构用单链表表示如下，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。



∴头指针的值是31

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

2.3.1 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
} ListNode，* LinkList；
typedef struct node ListNode;
typedef struct node* LinkList;
```

为什么要定义两个名字?

```
LNode * GetElem(LinkList L, int i){
    int j=1;
    LNode *p=L->next;
    if(i==0)
        return L;
    if(i<1)
        return  NULL;
    while(p!=NULL && j<i){
        p=p->next;
        j++;
    }
    return  p;
}
```

强调返回的是一个结点

强调这是一 个单链表

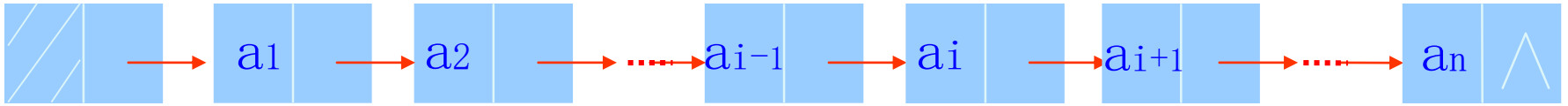
强调这是一个单链表 ——使用 LinkList
强调这是一个结点 ——使用 LNode *

typedef struct node* position;

2.3.1 线性链表

P

- 单链表的计算机实现：



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。

注意：

1、申请内存空间后，记得检查内存分配是否成功

2、不需要使用当不需要再使用申请的内存时，记得释放；释放后应该把指向这块内存的指针指向NULL，防止程序后面不小心使用了它。

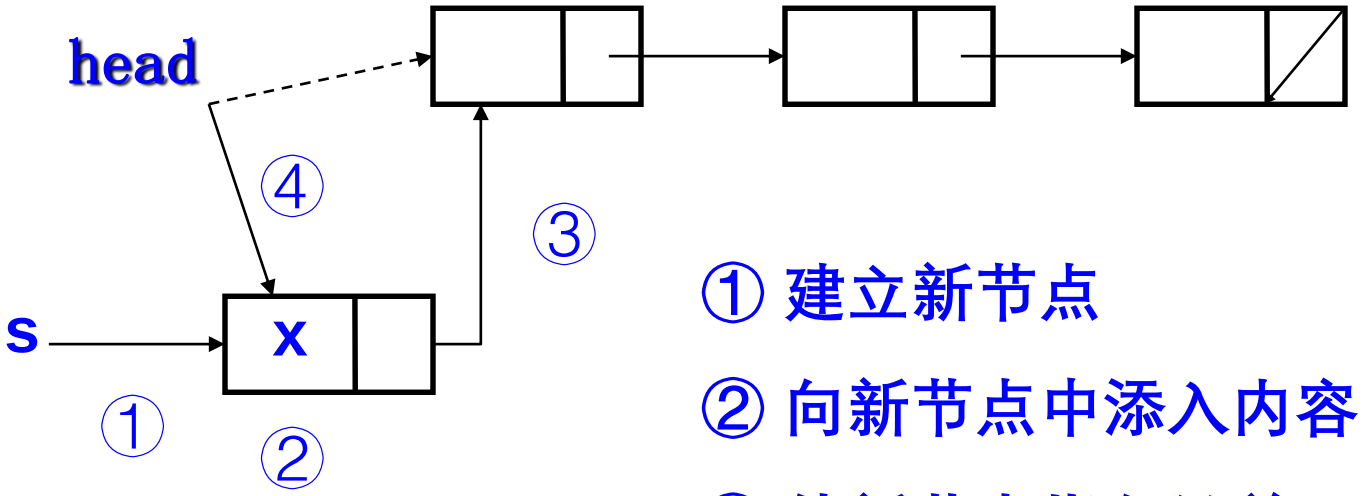
2.3.1 线性链表

- 建空线性链表（带头节点）：

```
算法：
Linklist  create_head ( )
 {LinkList   head;
    head = (LinkList)malloc(sizeof(ListNode));
//作用是由系统生成一个LinkNode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---头插法：



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()
 {   Linklist head;  char  ch;   ListNode *s;
     head=create_head ();
     while  ((ch=getchar())!='\n' )
      { s=(ListNode*)malloc(sizeof(ListNode));
          s->data=ch ;
          s->next=head->next;
          head->next=s;}
        return(head);     }
```

2.3.1 线性链表

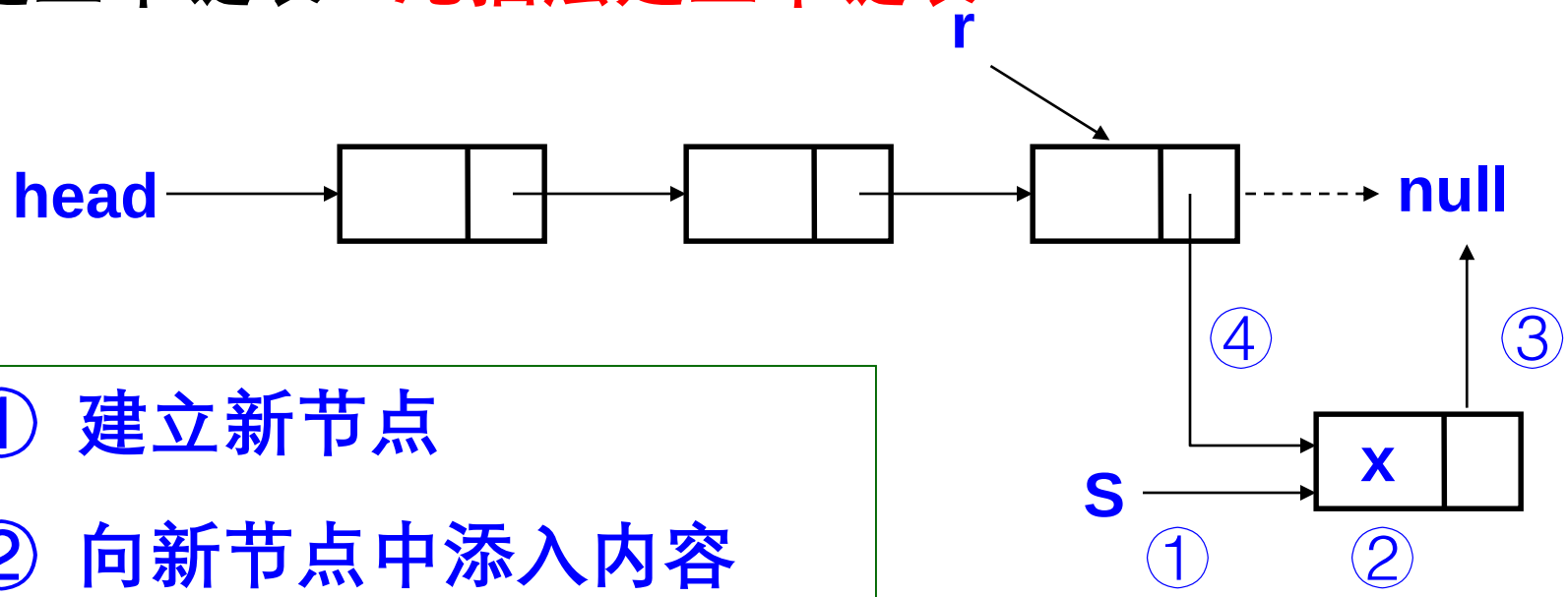
- 建立单链表---头插法：

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。

2.3.1 线性链表

- 建立单链表---尾插法建立单链表：



① 建立新节点

② 向新节点中添入内容

③ 将新节点链入链尾

④ 改变表尾节点的指针

2.3.1 线性链表

- 建立单链表---尾插法建立单链表：

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

图例

2.3.1 线性链表

- 单链表的建立：
  - （1）头插法建表
    - 将新节点插入在表头的位置
    - 生成的线性表的顺序和输入顺序相反
  - （2）尾插法建表
    - 将新节点插入在表尾的位置
    - 需要设立单独的表尾指针

2.3.1 线性链表

- 单链表的查找：
  - （1）按序号查找
  - （2）按值查找

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)
{ListNode *p;  p=head; int j=o;
while (p!=NULL&&j<i){
 p=p->next;
 j++;}
return p;}
```

时间复杂度为O(n)

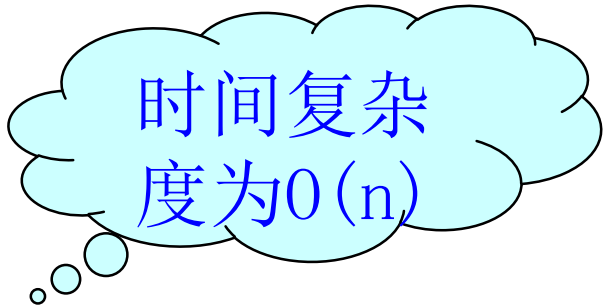
2.3.1 线性链表

- 单链表的查找---按值查找
  - 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

```
LinkNode *Locate( LinkList head ,DataType key)
 {LinkNode * p;
p=head->next;
  while  (p!=NULL&&p->data!=key)
      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

```
LinkNode *  NEXT (LinkList  L，  LinkNode*  p )
    { LinkNode  *q ;
      if  ( p→next == NULL )
          error ( “不存在后继元素” ) ;
      else
          {  q =  p→next;
             return  q ;
           }
     }
```

2.3.1 线性链表

- 单链表的查找---查找结点P的前驱结点

```
LinkNode * PREVIOUS (LinkList L, LinkNode* p)
    { LinkNode* 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

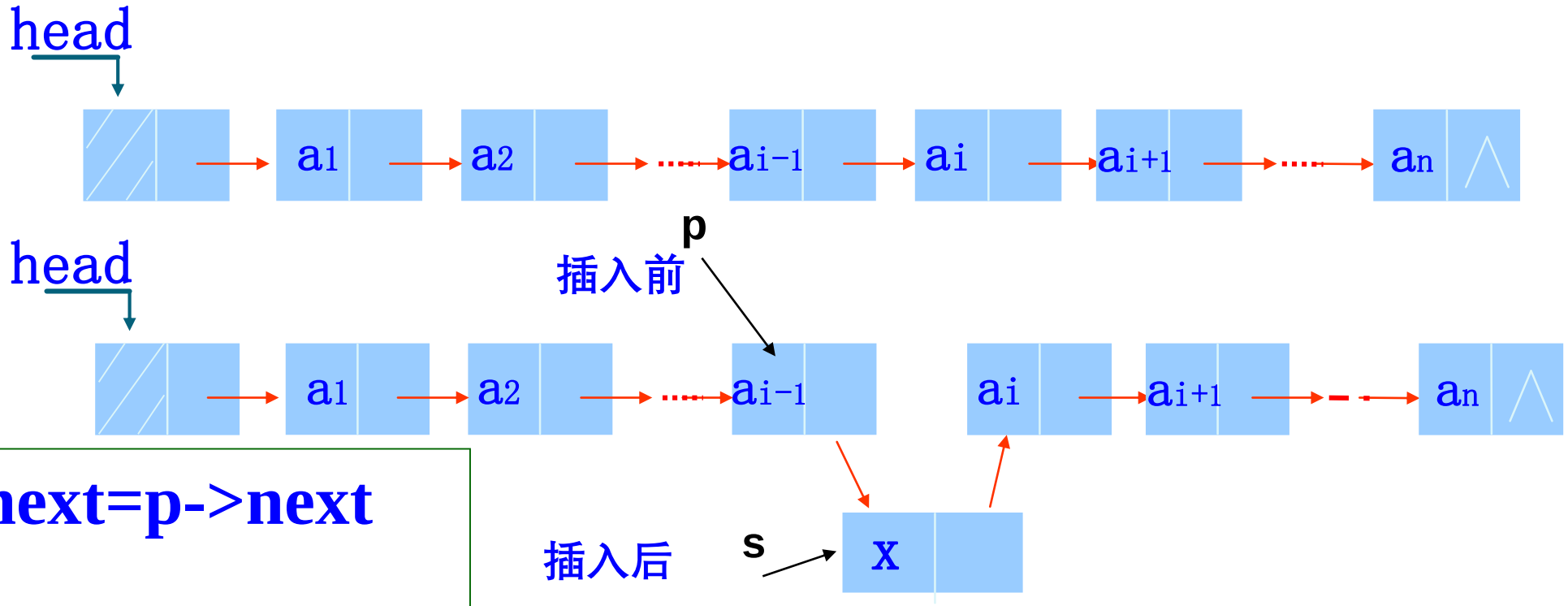
```
position  FIRST ( LIST  L )
     {
         return  L;
      }
```

2.3.1 线性链表

- 单链表的查找：
  - 一般来说，除了查找给定节点的后继节点之外，都需要从表头开始找

2.3.1 线性链表

- 单链表插入操作 Insertlist(L, x, i) ---在位置i插入新的结点



s->next=p->next

p->next=s

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Insertlist (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");
  ListNode *s;
  s=(ListNode  *)malloc(sizeof(ListNode));
  s->data=x;
  s->next=p->next;
  p->next=s;
    }
```

图例

2.3.1 线性链表

- 单链表插入操作---指定指针的插入运算
  - Insertlist (L,x,p) 指定指针的插入，分为前插和后插操作

1）后插既在所指结点之后插入新结点。

- O（1）

2）前插入既在所指结点之前插入新结点。

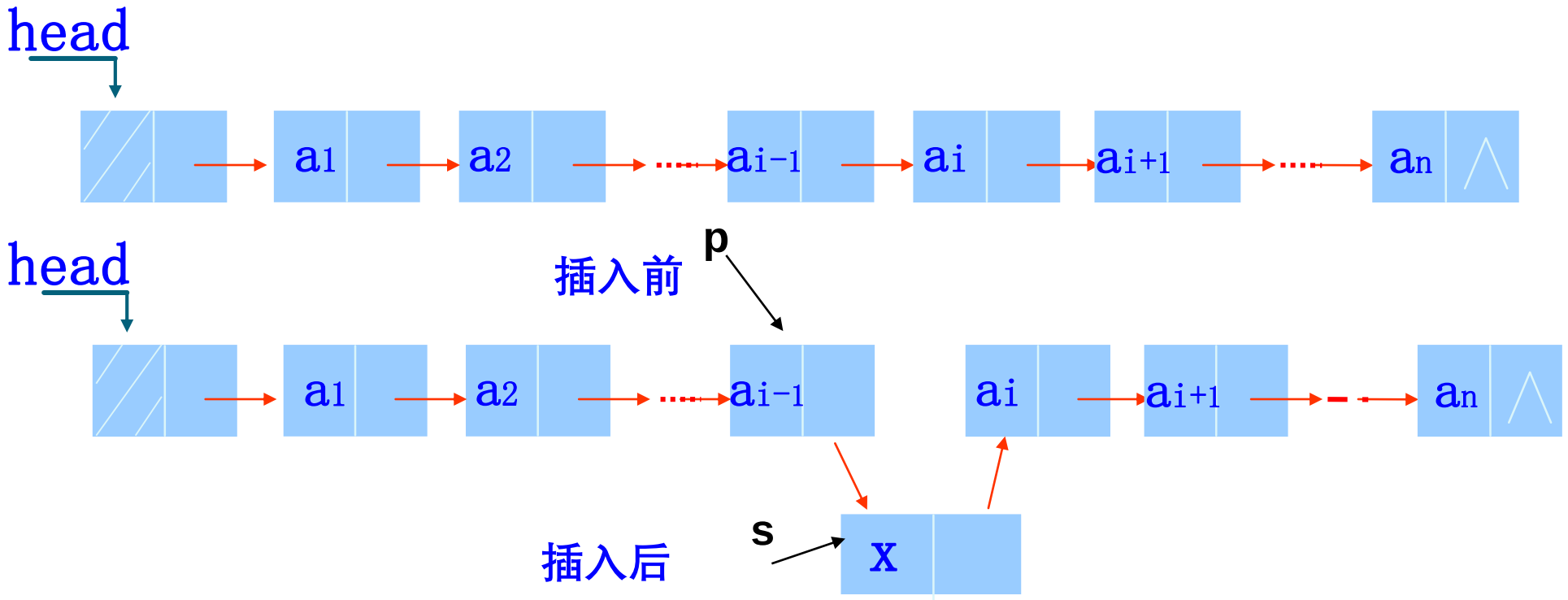
- 首先建立一个新结点，

  从链表头查找结点的前驱结点q

- O（n）

2.3.1 线性链表

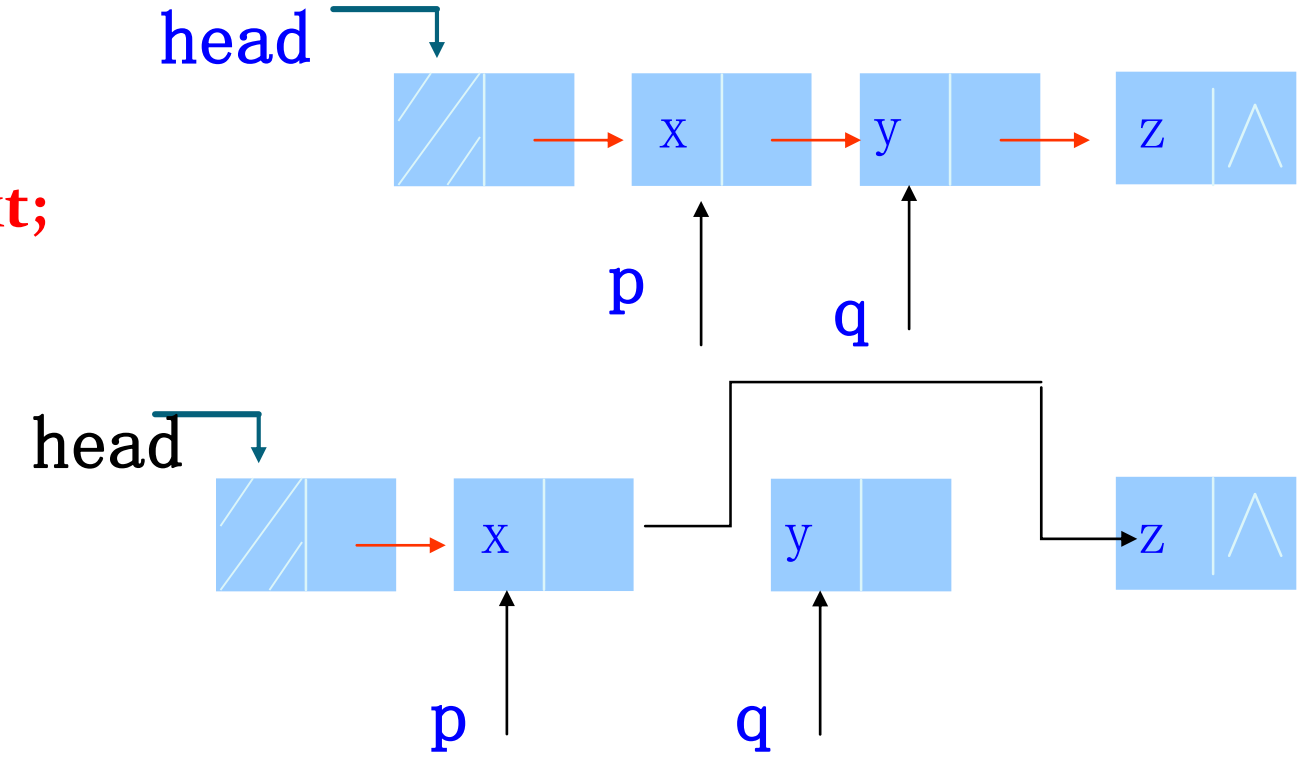
- 单链表的插入和删除：
  - 单链表插入操作 Insertlist(L, x, p) ---在位置p前插入新的结点



2.3.1 线性链表

- 单链表的删除---删除后继结点

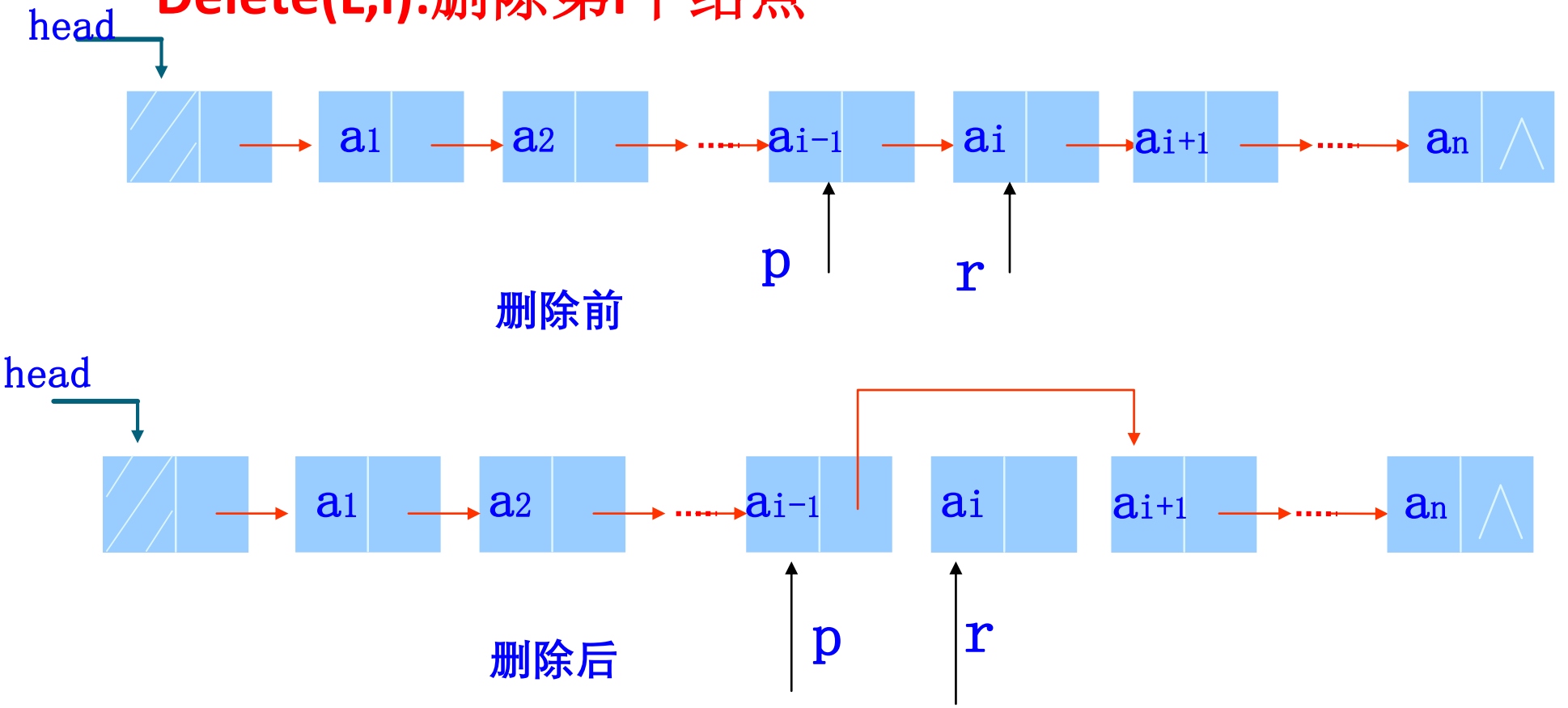
```
Deleteafter(ListNode * p)//删除*p的后继结点*q
{ListNode *q;
   q=p->next;
   p->next=q->next;
   free(q);
}
```



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



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；

图例

2.3.1 线性链表

- 单链表的删除---删除指定位置结点

```
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")
   }
```

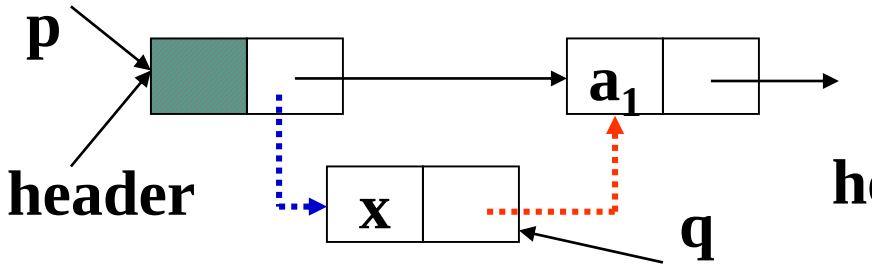
2.3.1 线性链表

- 表头结点的作用：

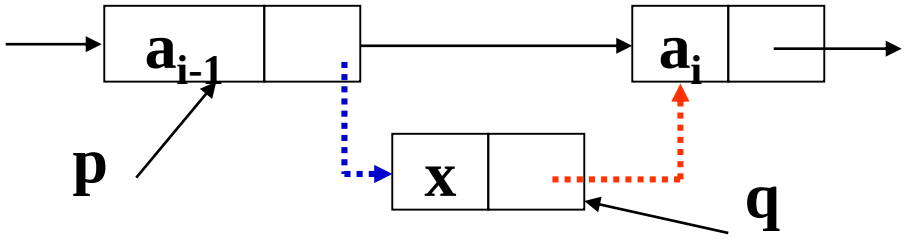
(1)插入元素

q = New NODE ;
q->data = x ;

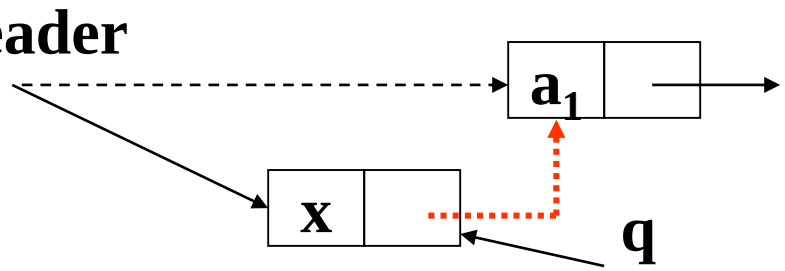
q->next = p->next ;
p→next = q ;



(a) 表头插入元素



(b) 中间插入元素



(c) 表头插入元素

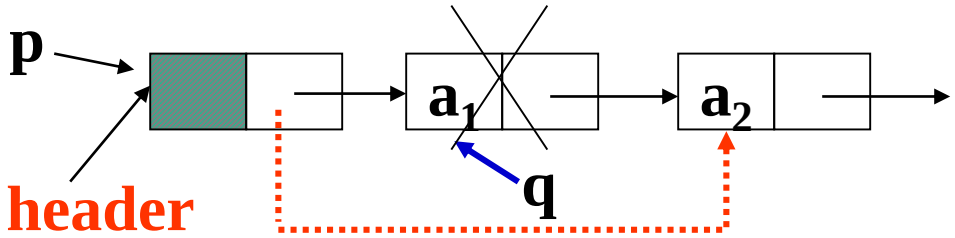
q->next = head ;
head = q ;

2.3.1 线性链表

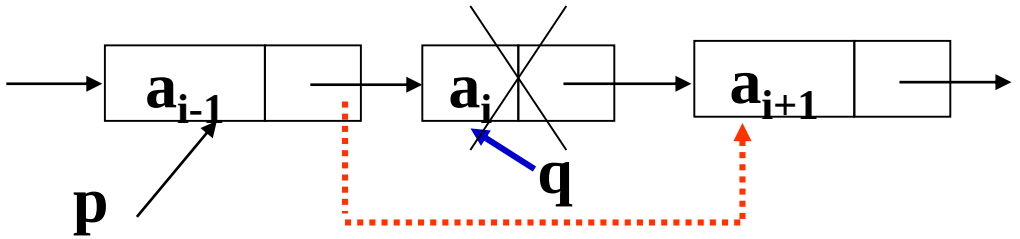
- 表头结点的作用：

(1)删除元素

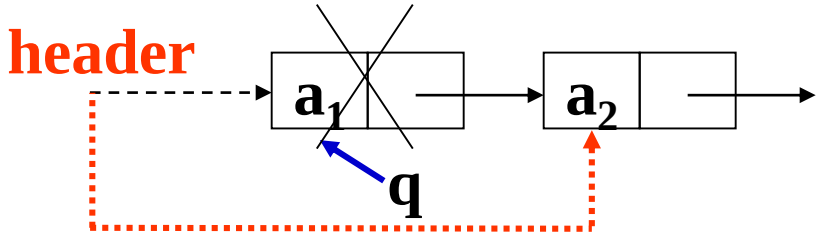
```
q = p->next ;
p->next = q->next ;
Delete  q ;
```



(a) 删除第一个元素



(b) 删除中间元素



(c) 删除第一个元素

```
q = head;
head= q->next ;
Delete  q ;
```

2.3.1 线性链表

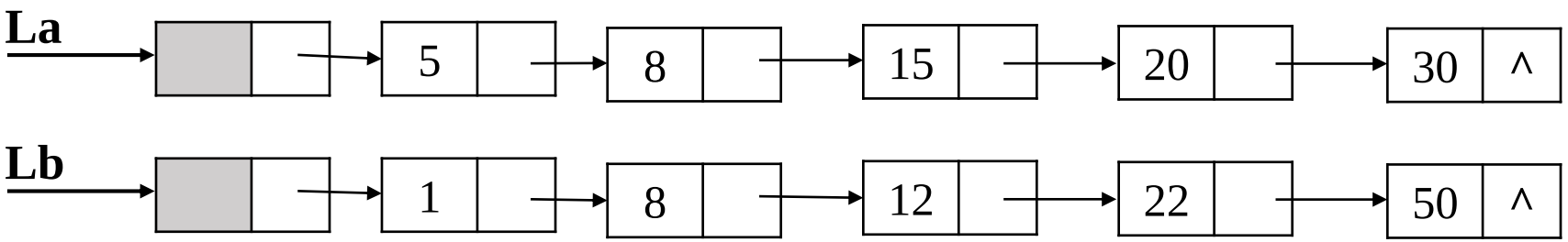
- 表头结点的作用：
  - 表头结点是线性链表第1个结点的前驱，使线性链表的插入和删除结点的操作统一起来；
  - 空表和非空表的处理统一了。
  - 注意：是否带表头结点在存储结构定义中无法体现，由操作决定。

操作：设L的型为LIST线性表实例，e 的型为ElementType的元素实例，p 为位置变量。所有操作描述为：

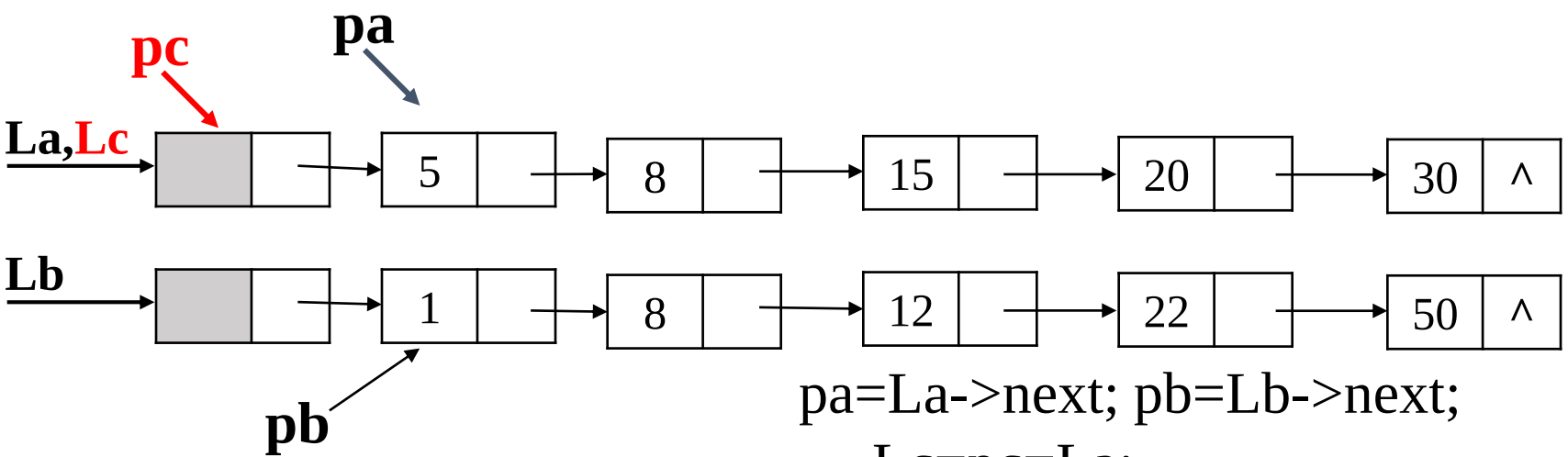
① ListInsert(&L, p, e);
② LocateElem(L,e, Compare());
③ GetElem(L,p,&e);
④ ListDelete(&L,p,&e); …

2.3.1 线性链表

例 已知：la，lb为线性表且元素按值非递减排列。
合并后得到新线性表lc也按值非递减排列。



2.3.1 线性链表



图例

```
pa=La->next; pb=Lb->next;
    Lc=pc=La;
    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 ;
```

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, Link &Lc)
{   pa=La->next; pb=Lb->next;
    Lc=pc=La;
    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 ;
      free(Lb) ;
} //MergeList_L
```

T(n)=O(La.length+Lb.Length)

2.3.1 线性链表

【例2-3】线性表的遍历与元素个数统计（带表头结构）

遍历：按照一定的原则或顺序，依次访问线性表中的每一个元素，且每个元素只能被访问一次。

原则或顺序：从前向后。

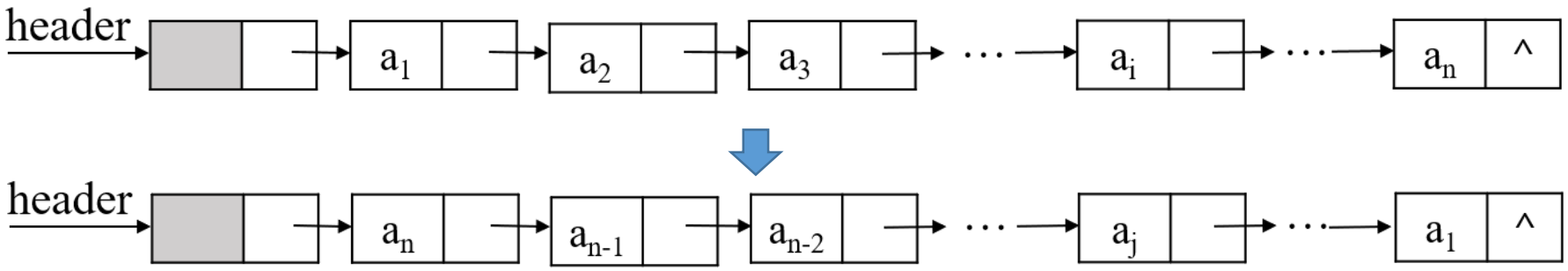
```
int List（LIST header)
{
   Position p;
   int n=0;
   p=header->next;
   while(p)
   {  visit(p->data);
      n++;
      p=p->next;
   }
   return(n);
}
```

问题：

如何实现按逆序输出链表中的每一个元素？

2.3.1 线性链表

【例2-4】线性表的反向，即线性表的最后一个元素变成第一个元素，而第一个元素变成最后一个元素。

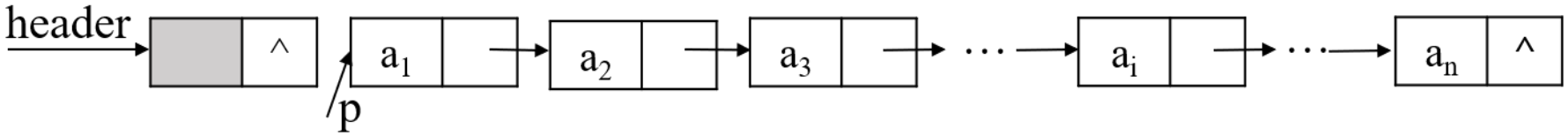


2.3.1 线性链表

【例2-4】线性表的反向，即线性表的最后一个元素变成第一个元素，而第一个元素变成最后一个元素。

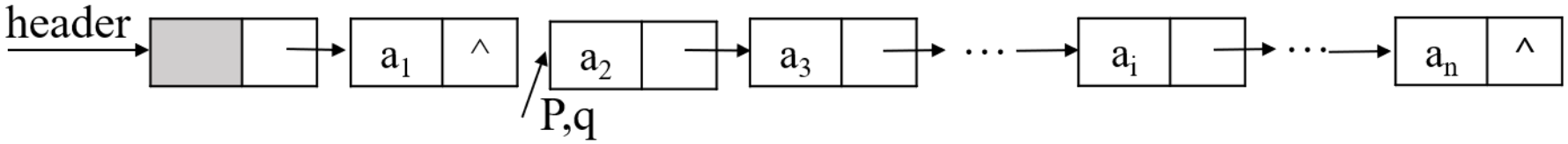
分析：

第1步：



p=head->next; head->next=NULL

第2步：

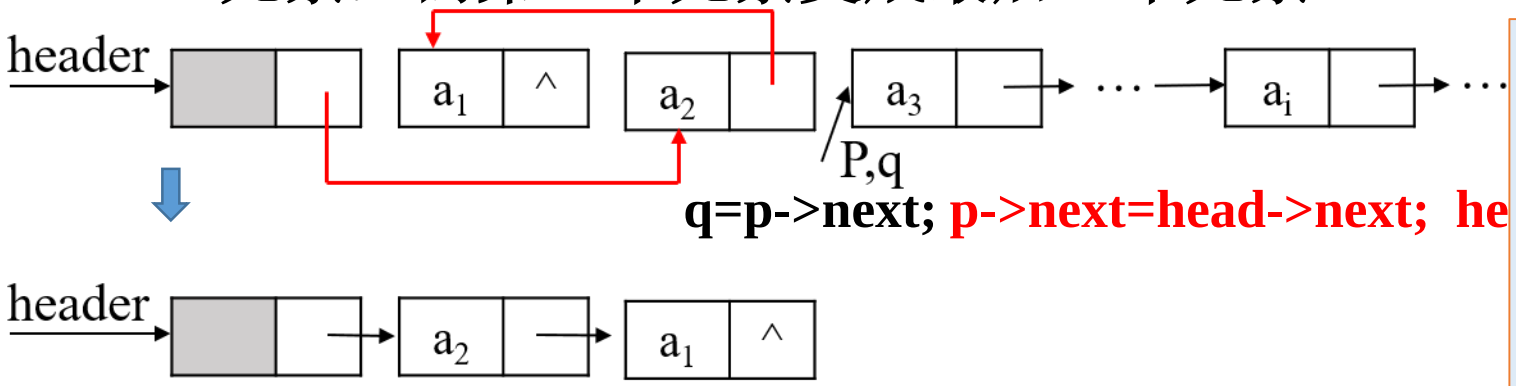


q=p->next; p->next=head->next; head->next=p; p=q;

2.3.1 线性链表

【例2-4】线性表的反向，即线性表的最后一个元素变成第一个元素，而第一个元素变成最后一个元素。

第3步：



重复上述操作，把链表p的每一个结点依次插入到header的头节点之后，每次都形成一个新的第一个节点……，直到p=NULL为止。这个过程可以用下面循环完成：

While(p) {q=p->next; p->next=head->next; head->next=p; p=q;}

```
Void Revers1(LIST &L)
{
  Position p,q;
  p=head->next;
  head->next=NULL;
  While(p)
  {  q=p->next;
     P->next=head->next;
     head->next=p;
     p=q;
   }
}// O(n)
```

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**
  - 相对顺序表而言，对线性链表中的任一结点进行操作复杂，只能实现的是顺序存取。
    - 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。
  - 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）。
  - 每创建一个结点，都要进行一次系统调用分配一块空间，这会浪费一定时间；由于创建结点的时间不固定，会导致结点分配的空间不连续，容易形成离散的内存碎片

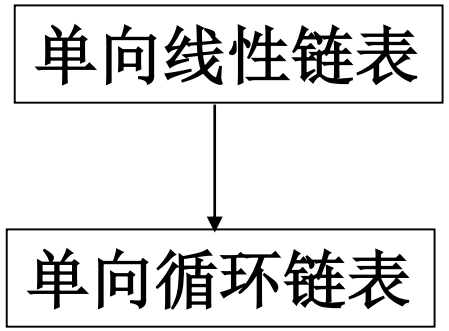
单链表小结

- 顺序表与链表的比较

顺序存储	比较参数	链式存储
固定，不易扩充	←表的容量→	灵活，易扩充
随机存取	←存取操作→	顺序存取
插入删除费时间	←时间→	访问元素费时间
估算表长度，浪费空间	←空间→	实际长度，节省空间

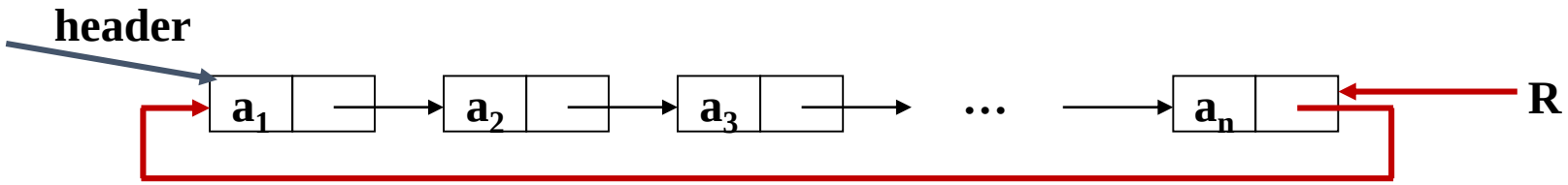
第二章 线性表

2.3.1 循环链表



对线性链表的改进，解决“单向操作”的问题；改进后的链表，能够从任意位置元素开始访问表中的每一个元素。

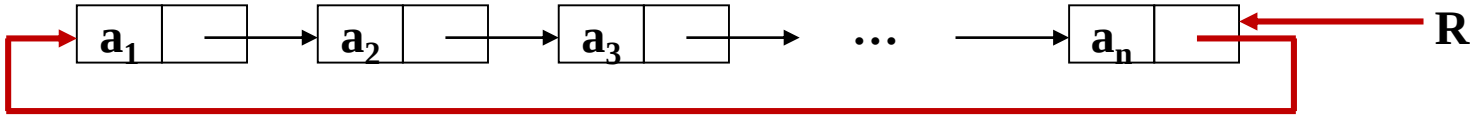
单向环形链表：



其结构与单向线性链表相同，一般用表尾指针标识链表，可以省去表头结点。表头：R→next　亦即表中的一个元素a_1

好处：能够在O（1）时间复杂度内同时找到表尾和表头

2.3.1循环链表



ADT操作：①在表左端插入结点Insert(x,First(R),R);

```
Void  LInsert( ElementType  x , LIST  &R )
{   celltype  *p ;
    p = New  NODE ;
    p→data = x ;
    if ( R == Null )
        {   p→next = p ;   R = p ;  }
    else
        {   p→next = R→next ;  R→next = p ;  }
 } ;
```

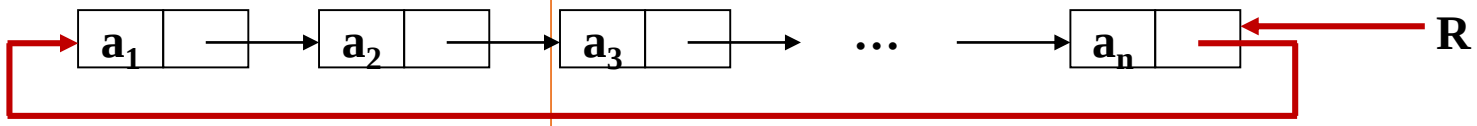
②在表右端插入结点Insert(x,End(R),R); →RInsert(x,R)

```
Void   RInsert( ElementType  x , LIST  R )
{
    LInsert ( x , R ) ;  R = R→next ;
 }
```

2.3.1循环链表

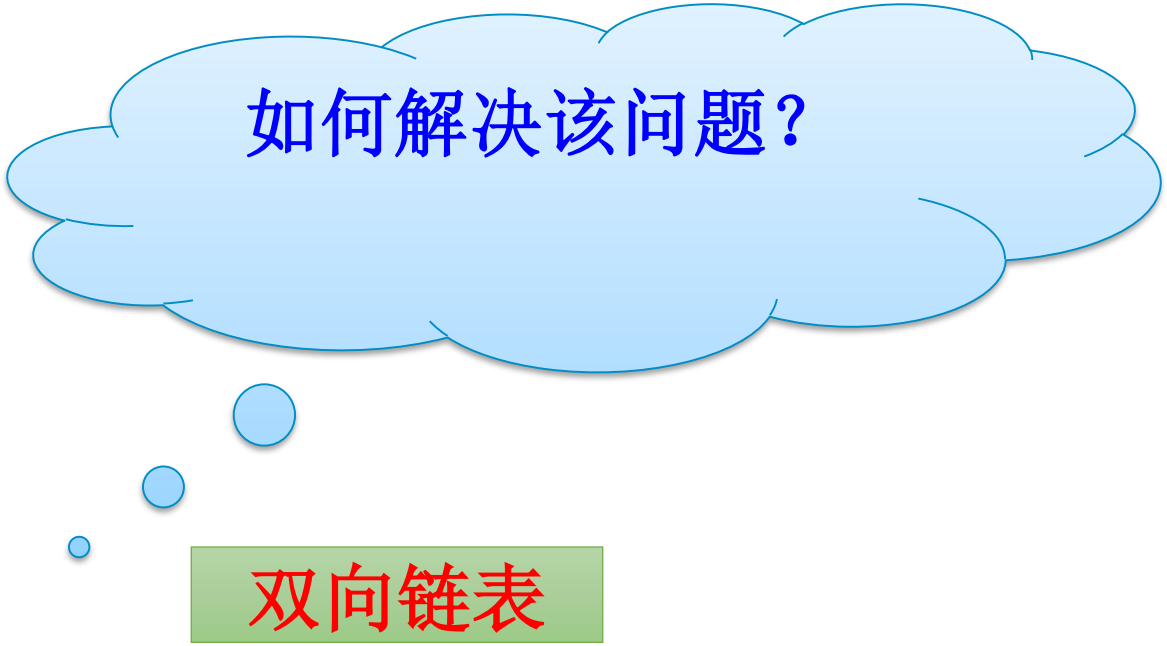
③从表左端删除结点Delete(First(R),R); →LDelete(R)

```
Void  LDelete( LIST  &R )
{   struct NODE  *p ;
    if ( R == Null )
        error ( “空表” );
    else
        {   p = R→next ;
            R→next = p→next ;
            if ( p == R )
               R=Null;
            Delete  p ;
        }
} ;
```



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。



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

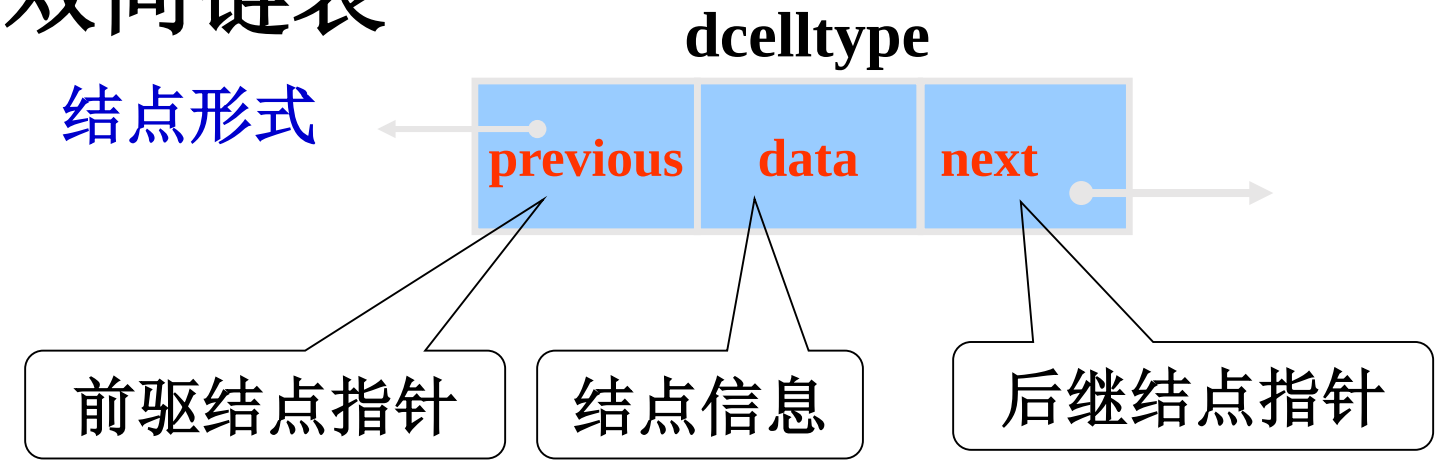
2.3.1 单链表

2.3.2 循环链表

2.3.3 双向链表

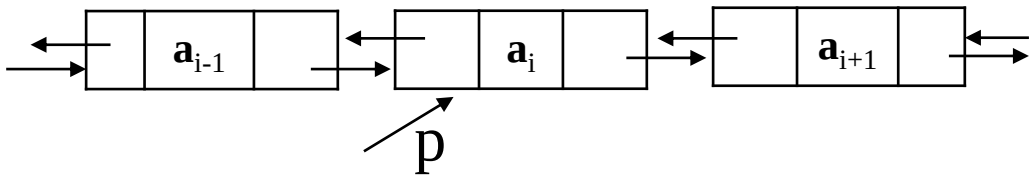
2.3.1 双向链表

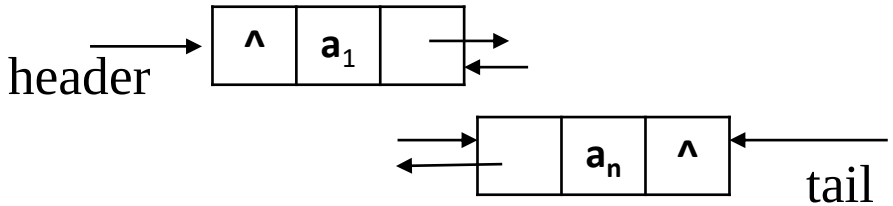
结点形式



结点类型

```
struct  DnodeType  {
    ElementType  data ;
    DnodeType  *next, *previous ;
} ;

typedef struct DnodeType  *DLIST ;
typedef struct DnodeType  *Position ;
```

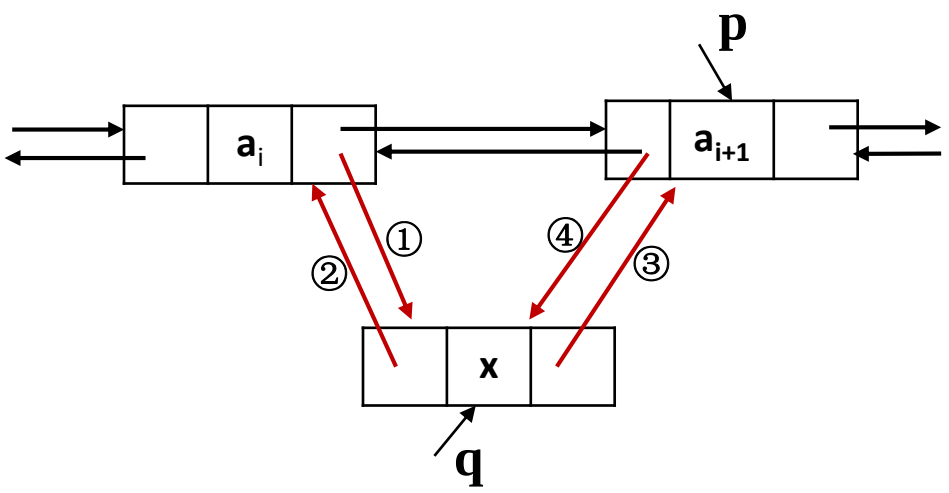




2.3.1 双向链表

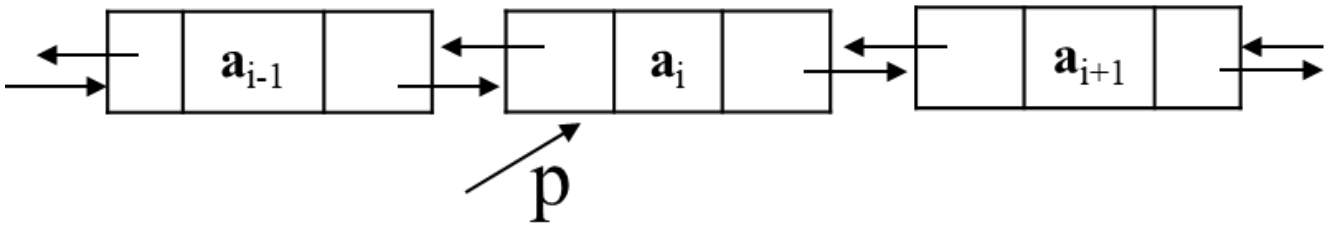
ADT操作：插入操作算法 –前插(将值为x的结点插入p 结点之前）

```
Void  Insert( ElementType x, Position p,  DLIST  &L ) ;
{
   Position  q ;
   q = new DNodeType ;
   q->data = x ;
   ①  P->previous->next = q ;
   ②  q->previous = p->previous ;
   ③  q->next = p ;
   ④  p->previous = q ;
} ;
```



2.3.1 双向链表

ADT操作：删除当前结点p



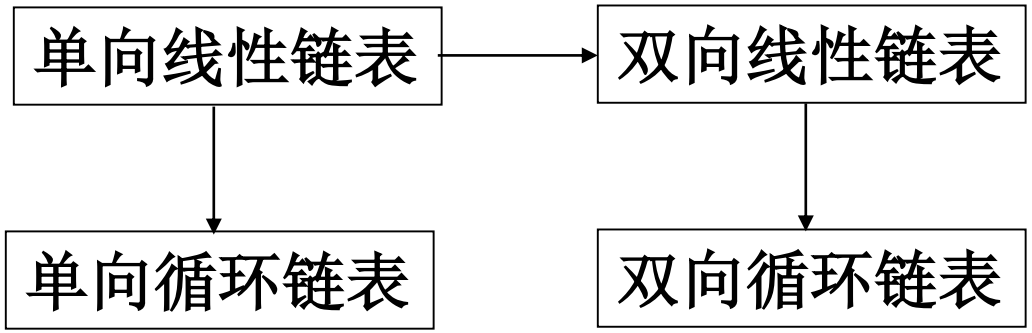
删除位置p的元素：

```
P->previous->next = p->next;
P->next->previous = p->previous ;
free(p) / Delete  p ;
```

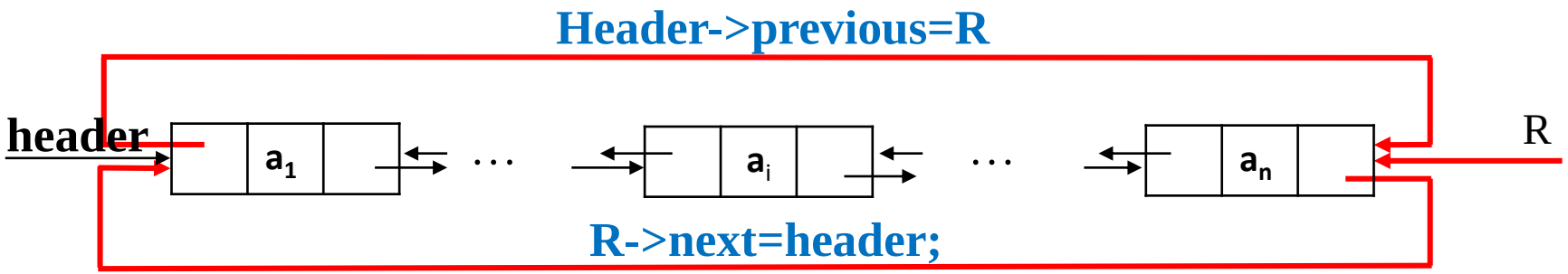
图例

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。



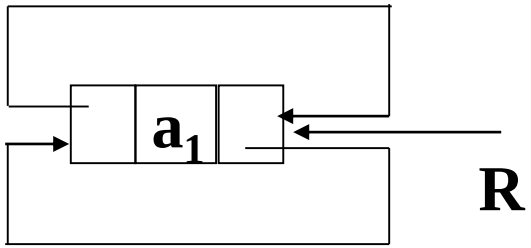
2.3.3 双向链表

- 双向循环链表

空双向环形链表：

R=Null：

单一结点双向环形链表：



R->next=R;
R->previous=R;

- 很明显，在双向循环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。
- 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。

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，
- ✓ 顺序表存储密度高，链表的存储密度较低

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。

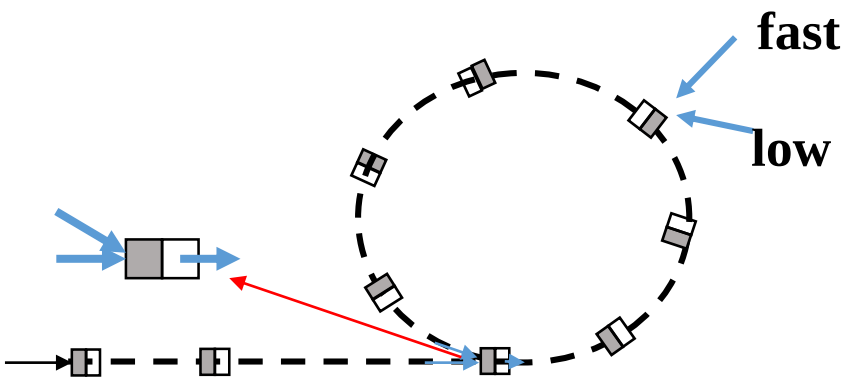
思考题：

1、线性链表，求倒数第K个数

2、线性链表，求中间位置的元素

3、单向链表环的问题

如何判断一个单向链表是否有环？



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。