数据结构与算法

Data Structures and Algorithms

第二章 线性表

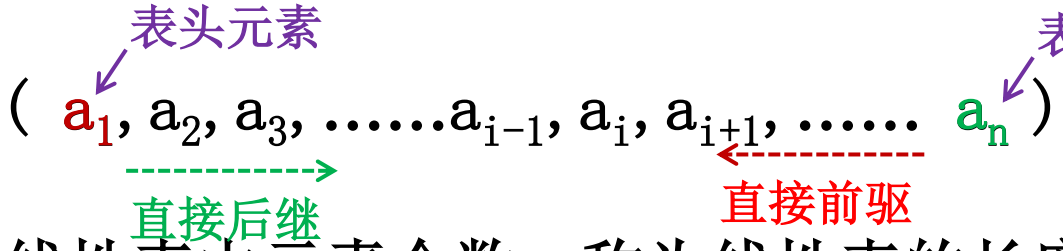
教学要求

- 掌握线性表的逻辑结构，线性表的顺序存储结构和链式存储结构的描述方法；
- 能够从时间和空间复杂性的角度综合比较两种存储结构的不同特点；
- 掌握栈和队列、串和数组的概念、存储及操作；
- 能够利用线性表解决实际应用问题。

2.1 抽象数据型线性表

【定义】线性表是由n(n≥0)个相同类型的元素组成的有序集合。

记为：



其中：① n为线性表中元素个数，称为线性表的长度；

② a_i为线性表中的元素，类型定义为ElementType；

③ 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；

④ 线性表是有限的，也是有序的。

⑤ 元素具有逻辑上的顺序性，各元素排序有先后次序。

数学模型

线性表 LIST = (D , R)

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

$R = \{ H \}$

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

ADT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);　, …..

ADT List {
  数据对象：D = { a_i | a_i ∈ ElemSet, i = 1,2,…,n, n≥0 }
  数据关系：R1 = { <a_{i-1},a_i> | a_{i-1},a_i ∈ D, i = 2,…,n }
  基本操作：
    InitList(&L)
      操作结果：构造一个空的线性表 L。
    DestroyList(&L)
      初始条件：线性表 L 已存在。
      操作结果：销毁线性表 L。
    ClearList(&L)
      初始条件：线性表 L 已存在。
      操作结果：将 L 重置为空表。
    ListEmpty(L)
      初始条件：线性表 L 已存在。
      操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE。
    ListLength(L)
      初始条件：线性表 L 已存在。
      操作结果：返回 L 中数据元素个数。
    GetElem(L, i, &e)
      初始条件：线性表 L 已存在，1≤i≤ListLength(L)。
      操作结果：用 e 返回 L 中第 i 个数据元素的值。
    LocateElem(L, e, compare())
      初始条件：线性表 L 已存在，compare()是数据元素判定函数。
      操作结果：返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。若这样的数据元素不存在，则返回值为 0。
    PriorElem(L, cur_e, &pre_e)
      初始条件：线性表 L 已存在。
      操作结果：若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的前驱，否则操作失败，pre_e 无定义。
    NextElem(L, cur_e, &next_e)
      初始条件：线性表 L 已存在。
      操作结果：若 cur_e 是 L 的数据元素，且不是最后一个，则用 next_e 返回它的后继，否则操作失败，next_e 无定义。
    ListInsert(&L, i, e)
      初始条件：线性表 L 已存在，1≤i≤ListLength(L) + 1。
      操作结果：在 L 中第 i 个位置之前插入新的数据元素 e，L 的长度加 1。
    ListDelete(&L, i, &e)
      初始条件：线性表 L 已存在且非空，1≤i≤ListLength(L)。
      操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1。
    ListTraverse(L, visit())
      初始条件：线性表 L 已存在。
      操作结果：依次对 L 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失败。
} ADT List

【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) ;
        }
 }
```

2.2 线性表的实现

问题：确定数据结构（存储结构）实现型LIST，并在此基础上实现各个基本操作。

存储结构的三种方式：

① 连续的存储空间（数组） → 静态存储

② 非连续存储空间——指针（链表） → 动态存储

③ 游标（连续存储空间+动态管理思想）→ 静态链表

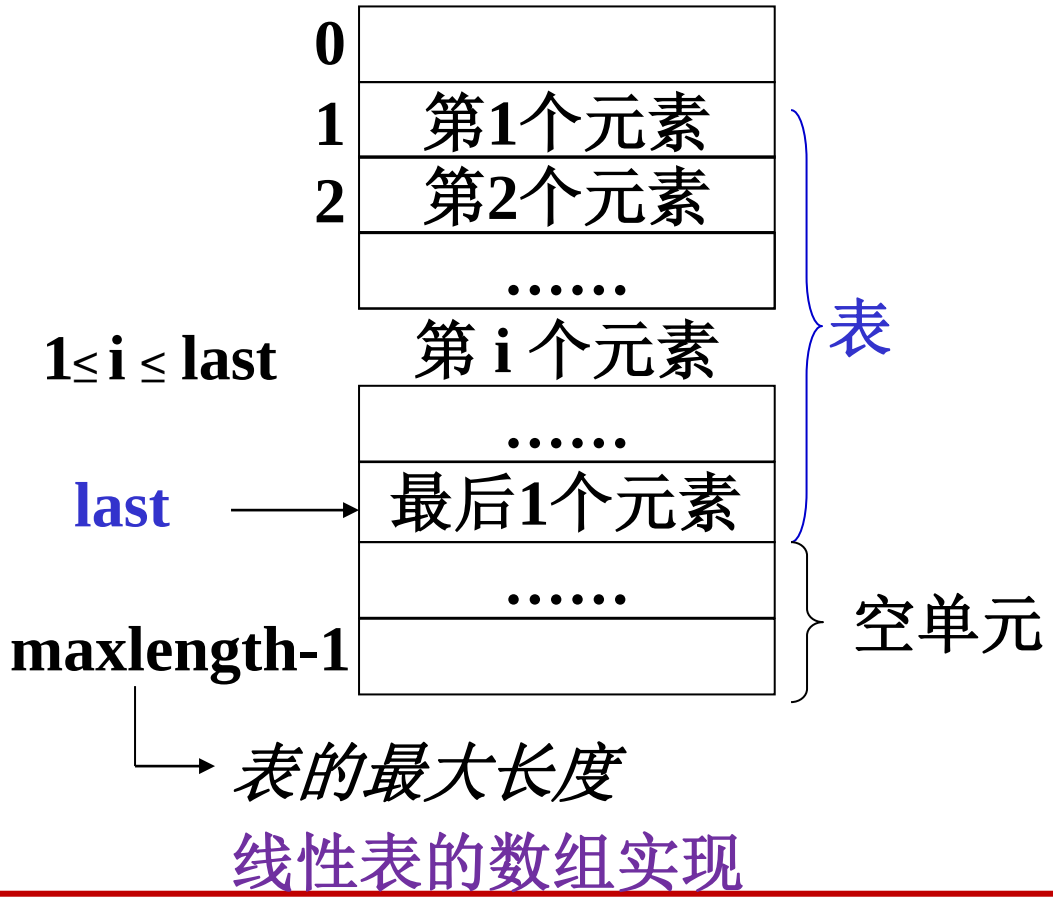
2.2.1 指针和游标

指针：地址量，其值为另一存储空间的地址；

游标：整型量，其值为数组的下标，用以表示指定元素的“地址”或“位置”。

2.2.2　顺序表--线性表的顺序表示

线性表的顺序存储又称顺序表。使用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的数据元素在物理位置上也相邻。



线性表的数组实现

类型定义：

```
# define  maxlength  100
Typedef struct   {
  ElementType
          data[maxlength];
           int  last;
} LIST;
```

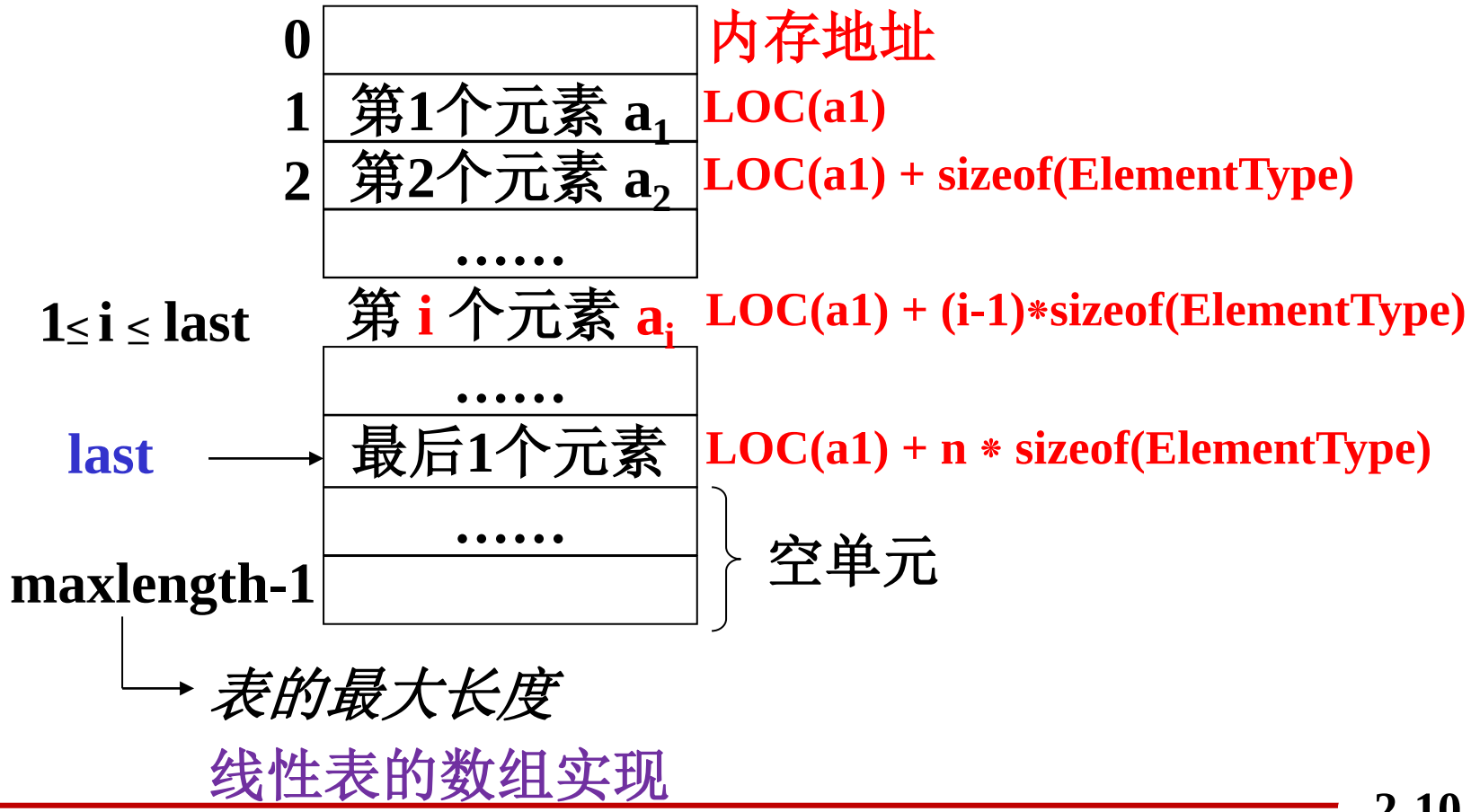
位置类型：

```
      typedef  int  Position;
```

线性表 L:　LIST　L;

2.2.2 顺序表--线性表的顺序表示

线性表的顺序存储又称顺序表。使用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的数据元素在物理位置上也相邻。



线性表的数组实现

线性表的第i个数据元素a_i的存储位置为:

随机访问

$$LOC(a_i)=LOC(a_1)+(i-1)*sizeof(ElemType)$$

线性表动态分配数组实现的顺序存储结构:

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

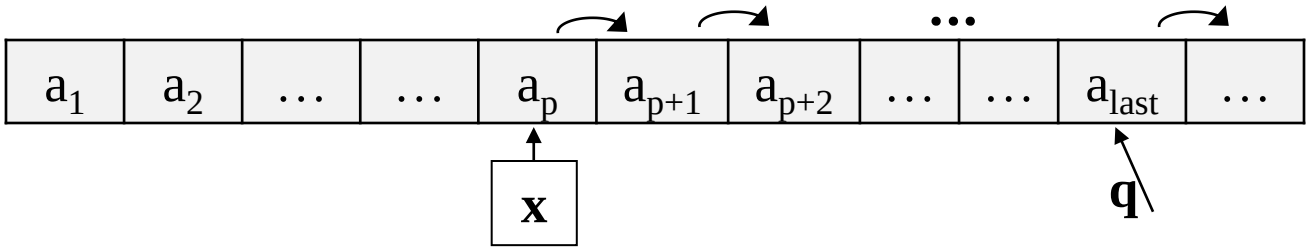
初始化线性表：

```
Status InitList_Sq(SqList  &L) {
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE*sizeof(ElemType));
    if ( !L.elem ) exit(“溢出”）；
    L.length = 0；                          //线性表长度
    L.listsize = LIST_INIT_SIZE ;     //初始存储容量
    return OK;
} //InitList_Sq ;
```

```
if（L.length >= L.listsize)
{   newbase = (ElemType *)realloc(L.elem,
                  (L.listsize+LISTINCREMENT)*sizeof(ElemTYPE));
    if(!newbase) exit(“溢出”);
    L.elem = newbase;                                  //新基址
    L.listsize = L.listsize + LISTINCREMENT ;  //新存储容量
} //if
```

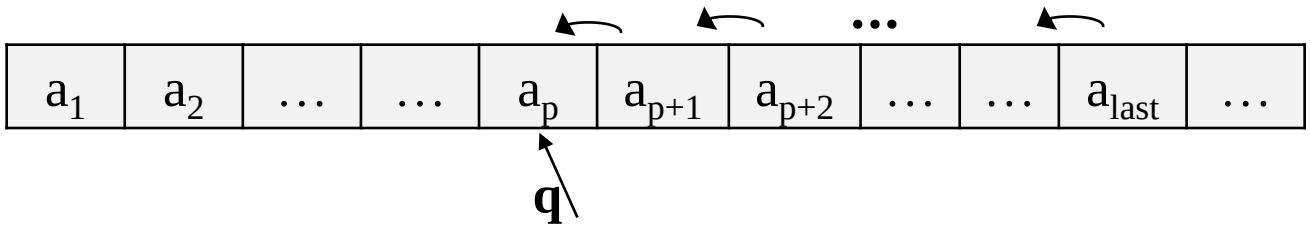
ADT 插入操作

```
① Void ListInsert (LIST  &L, Position  p, ElementType  x)
   {  Position  q ;
      if (L.last >= maxlength – 1)
           error( “ 表满 ” ) ;
      else  if (( p > L.last +1 ) || ( p < 1) )
           error( “ 指定位置不存在 ” ) ;
      else  {
                for ( q = L.last; q >= p; q -- )
                     L.data[ q + 1] = L.data[ q ] ;
                L.last = L.last + 1 ;
                L.data[ p ] = x ;
              }
    }
```



删除操作

```
② Void ListDelete(LIST &L, Position p, ElementType &e)
  { Position q ;
    if ( ( p > L.last ) || ( p < 1 ) )
        error ( “指定位置不存在” ) ;
    else
       { L.last = L.last – 1;
         &e=L.data[p].data;
         for ( q = p ; q <= L.last ; q ++ )
               L.data[ q ] = L.data[ q + 1 ];
       }
  }
```



● 假设 p_i 是在第 i 个元素之前插入一个元素的概率，则在长度为 n 的线性表中插入一个元素时所需移动元素次数的期望值(平均次数)为

$$E_{is} = \sum_{i=1}^{n+1} p_i(n-i+1) \tag{2-3}$$

● 假设 q_i 是删除第 i 个元素的概率，则在长度为 n 的线性表中删除一个元素时所需移动元素次数的期望值(平均次数)为

$$E_{dl} = \sum_{i=1}^{n} q_i(n-i) \tag{2-4}$$

平均查找长度

不失一般性，我们可以假定在线性表的任何位置上插入或删除元素都是等概率的，即

$$p_i = \frac{1}{n+1}, \quad q_i = \frac{1}{n}$$

则式(2-3)和(2-4)可分别简化为式(2-5)和(2-6)：

● $$E_{is} = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2} \tag{2-5}$$

● $$E_{dl} = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2} \tag{2-6}$$

由式(2-5)和(2-6)可见，在顺序存储结构的线性表中插入或删除一个数据元素，平均约移动表中一半元素。若表长为 n，则算法 ListInsert_Sq 和 ListDelete_Sq 的时间复杂度为 $O(n)$。

```
③ int  LocateElem (LIST  L, ElementType e, compare())
    {  int  i ;
       for ( i = 1; i <= L.last ;  i++ )
            if ( compare(L.data[ i ] , e) )
                                  // Equal(L.data[q],x,compare())
                 return ( i ) ;
       return ( L.last + 1 ); // or 0
     }
```

最好情况：在表头，比较1次，O(1);

最坏情况：在表尾或不存在，比较n次， O(n);

平均情况：假设$p_i(1/n)$是查找每个位置的概率，则在长度为n的线性表上查找值为e的元素，平均比较次数为

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

平均查找时间复杂度为 O(n)。

线性表的顺序存储与实现

顺序表的特点：

1. 通过元素的存储顺序反应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. 顺序表的插入和删除操作要通过移动元素实现

```
void MergeList_Sq(SqList La, SqList Lb, SqList &Lc) {
    // 已知顺序线性表 La 和 Lb 的元素按值非递减排列
    // 归并 La 和 Lb 得到新的顺序线性表 Lc,Lc 的元素也按值非递减排列
    pa = La.elem; pb = Lb.elem;
    Lc.listsize = Lc.length = La.length + Lb.length;
    pc = Lc.elem = (ElemType * )malloc(Lc.listsize * sizeof(ElemType));
    if (!Lc.elem)exit(OVERFLOW);  // 存储分配失败
    pa_last = La.elem + La.length - 1;
    pb_last = Lb.elem + Lb.length - 1;
    while (pa <= pa_last && pb <= pb_last) {     // 归并
      if (*pa <= *pb) *pc++ = *pa++;
      else *pc++ = *pb++;
    }
    while (pa <= pa_last) *pc++ = *pa++;         // 插入 La 的剩余元素
    while (pb <= pb_last) *pc++ = *pb++;         // 插入 Lb 的剩余元素
}  // MergeList_Sq
```

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

思考题：

1、数组反向

数组元素首尾交换，实现逆向存储。

2、数组合并

现在给出两个数组A和B，其中：
数组A：“1，7，9，11，13，15，17，19”；
数组B：“2，4，6，8，10”；
两个数组合并为数组C，按升序排列。

3、数组循环移位

将一个含有n个元素的数组向右循环移动k位，要求时间复杂度是O(n)，且只能使用两个额外的变量。

4、在有序数组上的操作，如找给定的数字

输入一个已经按升序排序过的数组和一个数字，在数组中查找两个数，使得它们的和正好是输入的那个数字。

设将 n (n>1) 个整数存放在一个一维数组 R 里。设计一个时间和空间两方面都尽可能高效的算法。将R中保存的序列循环左移 p(0<p<n)个位置。

将R数据由 $(X_0, X_1, \ldots, X_{n-1})$ 变换成 $(X_p, X_{p+1}, \ldots, X_{n-1}, X_0, X_1, \ldots, Xp-1)$。

要求：

1）给出算法的基本设计思想；

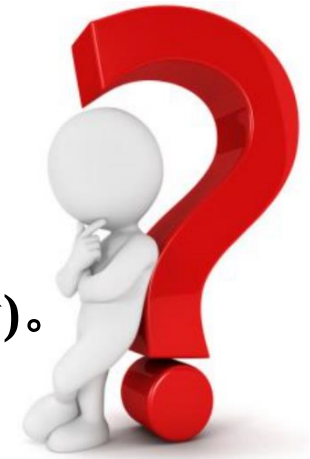
2）根据设计思想，采用C或C++或Java语言描述算法，关键之处给出注释；

3）说明你所设计算法的时间和空间复杂度。

abcdefgh —左移三个→ defghabc

思路一：辅助数组求解。

思路二：视为数据 ab 转换成 ba (a是前p个元素，b是后n-p个元素)。



思考题

给定一个含 n（n>=1）个整数的数组，请设计一个在时间上尽可能高效的算法，找出数组中未出现的最小正整数。例如，数组{-5,3,2,3}中未出现的最小正整数是1；数组{1,2,3}中未出现的最小正整数是4。要求

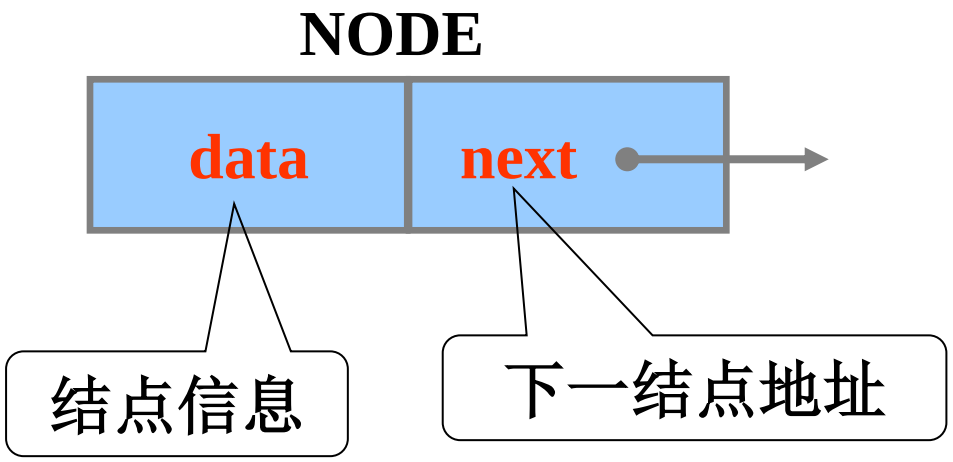
1）给出算法的基本设计思想；

2）根据设计思想，采用C或C++描述算法，关键之处给出注释；

3）说明你所设计算法的时间复杂度和空间复杂度。

2.2.3 单链表--线性表的指针实现

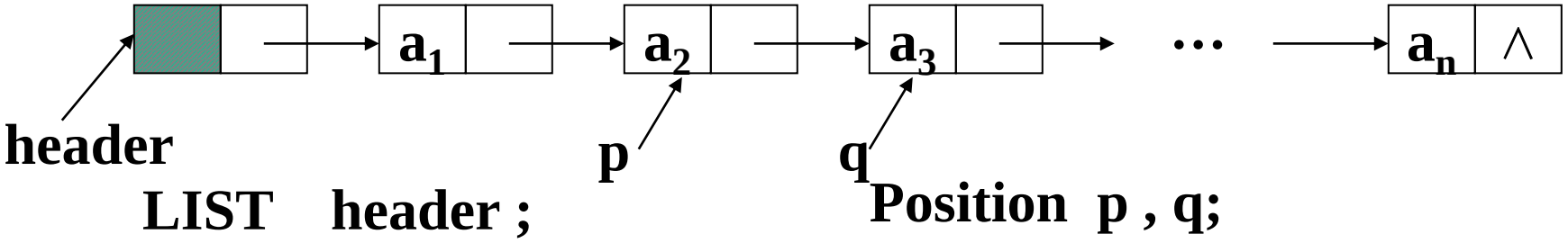
结点形式



结点类型

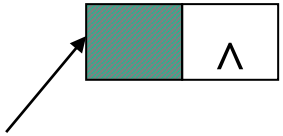
```
struct  NODE  {
    ElementType  data ;
    struct NODE  *next ;
} ;

typedef  NODE *LIST;
typedef  NODE  *Position;
```



LIST header ;

Position p , q;



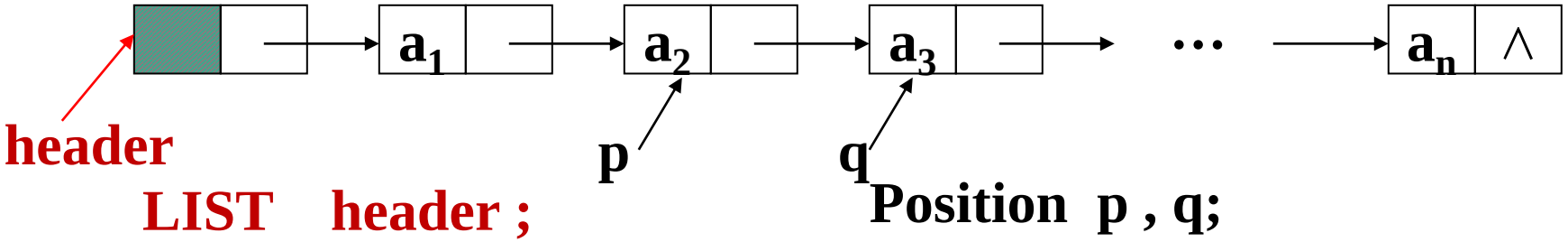
记法:

a_2: (*p).data ;
q: (*p).next ;

⇒

a_2: p->data ;
q: p->next ;

2.2.3 单链表--线性表的指针实现



头结点与头指针的区别：

不管有没有头结点，头指针始终指向链表的第一个节点；

而头结点是带有头结点的链表中第一个节点，通常不存储任何信息。

引入头结点的好处：

① 头结点的指针域存放首结点的位置，链表第一个位置和其他位置上的操作一致，无需特殊处理。

② 无论链表是否为空，其头指针都指向头结点的非空指针（空表中头结点的指针域为空），空表与非空表处理统一。

操作讨论：

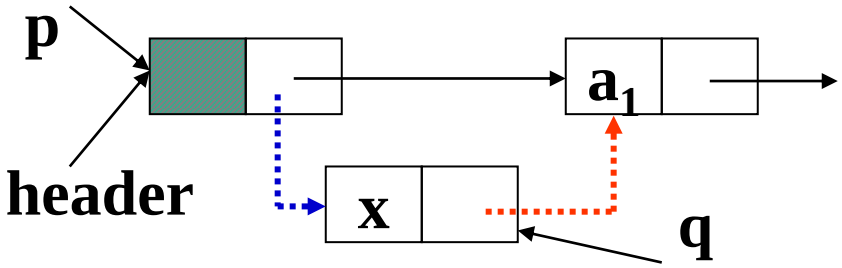
(1)插入元素

```
q = New  NODE ;
q->data = x ;
q->next = p->next ;
p→next = q ;
```

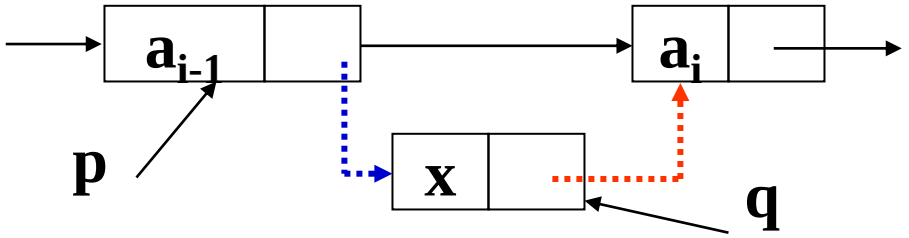
或:

```
temp = p->next ;
p->next = New  NODE ;
p->next->data = x ;
p->next->next = temp ;
```

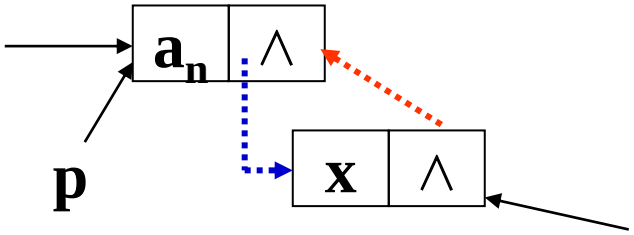
讨论表头结点的作用



(a) 表头插入元素（头插法）



(b) 中间插入元素



(c) 表尾插入元素（尾插法）

操作讨论：

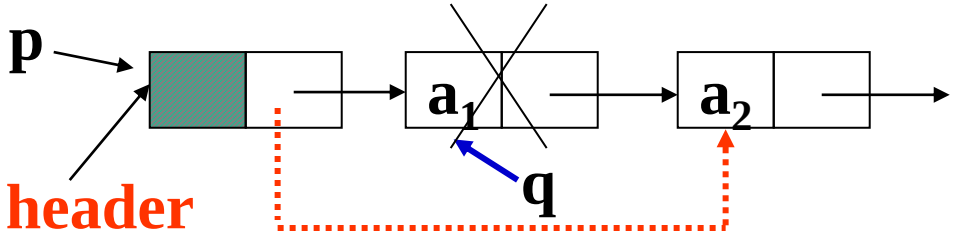
(2)删除元素

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

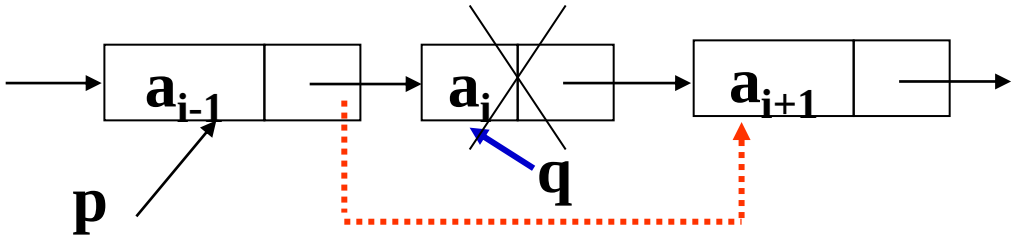
或：

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

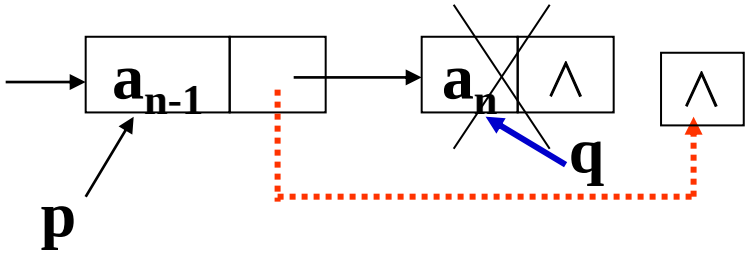
讨论结点 a_i 的位置 p



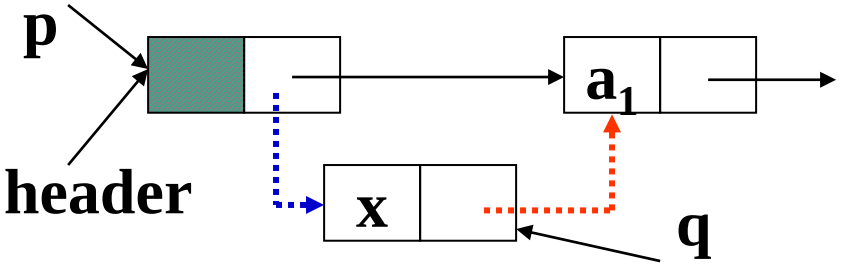
(a) 删除第一个元素

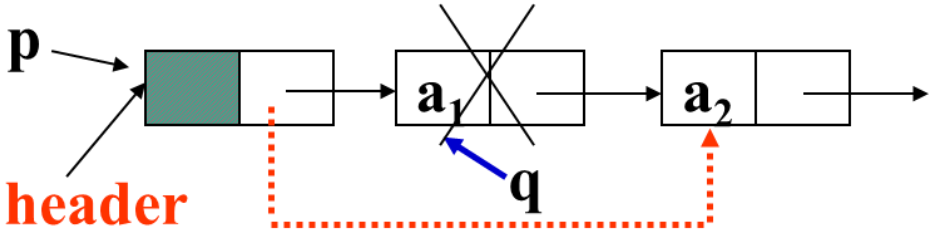


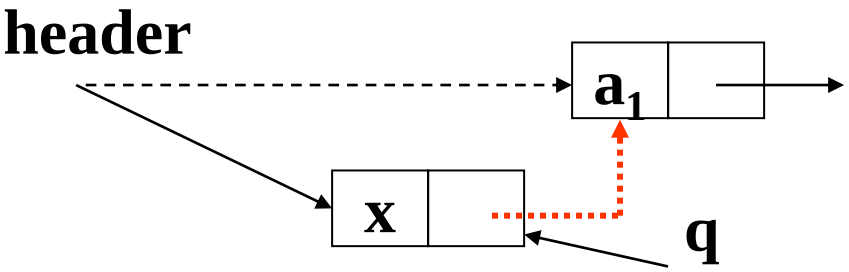
(b) 删除中间元素



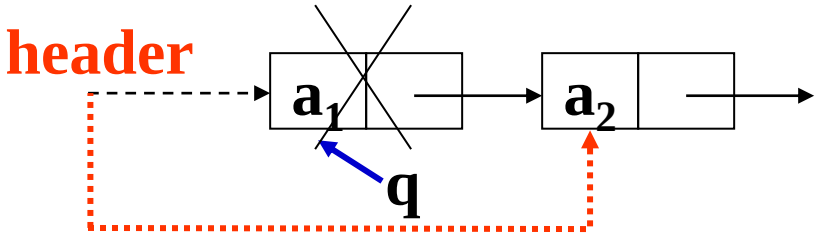
(c) 删除表尾元素







(a) 表头插入元素



(a) 删除第一个元素

- 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；
- 通常，将线性链表中的元素位置超前一个结点，即指向结点前驱的指针；
- 头结点指针相当于一个地址常量。

ADT操作

① Position LocateElem (LIST L , ElementType x, int compare())//值

```
{ position p ;
  p = L ;
  while ( p→next != Null )
        if ( compare(p→next→data ,x ) )
            return p ;
        else
            p = p→next ;
   return p ;
}
```

② Void ListInsert (LIST &L , int i, ElementType x) // 第i个元素

```
{ position p, q ; int j=0;
  p=L;
  while(p&&j<i-1) { j++; p=p->next; }// 找到第 i-1 个位置
  if(!p||j>i) return Error;
  q = New NODE ;
  q →data = x ;
  q →next = p →next ;
  p →next = q ;
}
```

③ GetElem (LIST L, int i, ElementType &e) // 第i个元素

```
{   position p; int j=1;
    p=L->next;
    while(p&& j<i) { p=p->next; j++;  }
    if(!p||j>i)   return Error;
     &e=p ->data;
}
```

④ ListDelete (LIST &L, int i, ElementType &e) // 第i个元素

```
{  position  q; int j=0;
    p=L;
    while(p->next && j<i-1) { p=p->next; j++;  }
     if  (!( p->next)||j>i-1)  return Error ;
    q = p →next ;
    p ->next = q->next ;
    &e=q->data;
    Delete  q ;
} ;
```

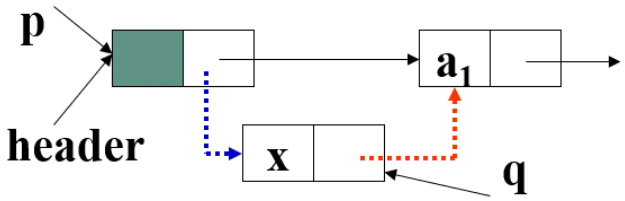
按值查找又如何呢？

【例2-1】输入*n*个元素的值，建立带表头结点的单向线性链表。

（教材）单向链表存储结构：

```
typedef  struct LNode {
        ElemType data ;
        struct LNode  *next ;
} LNode  *LinkList ;
```

```
void CreateList ( LinkList &L, int  n )
{       L = (LinkList)malloc(sizeof(Lnode)) ;        //表头
        L->next = NULL ;
        for (i=0; i<n; i++)
            {   p = (LinkList)malloc(sizeof(LNode));
                scanf(&p->data);
                p->next = L->next ;
                L->next = p ;
             } //for
}//CreateList
```

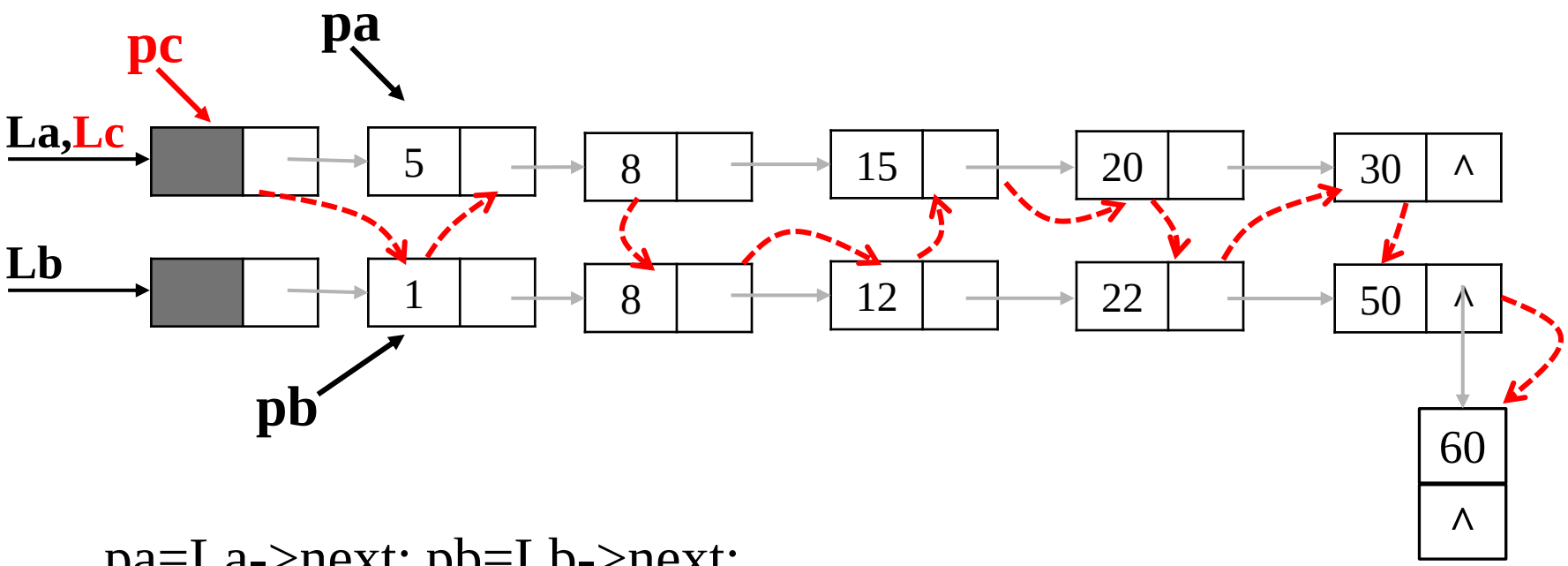


(a) 表头插入元素（头插法）

【例2-2】已知单链线性表La和Lb的元素按值非递减排列，设计算法归并La和Lb得到新的单链线性表Lc，使Lc的元素也按非递减排列。要求算法空间复杂度为O(1)。

```
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
```

Pc=Pa=Pa+Pb




```
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 ;
```

合并算法总结：

两个有序的线性链表合并成一个新的有序的线性链表

① 顺序存储的合并算法时间复杂度为O(m+n)或O(max(m,n))，空间复杂度为O(m+n);

② 链式存储的合并算法时间复杂度同顺序存储，但空间复杂度为O(1)；

③ 合并算法是内部排序的一种方法，但对外部排序来讲，这是唯一方法。

思考题：

已知单链线性表La和Lb的元素按值非递减排列，设计算法归并La和Lb得到新的单链线性表Lc，使Lc的元素按非递增排列。要求算法空间复杂度为O(1)。

【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。

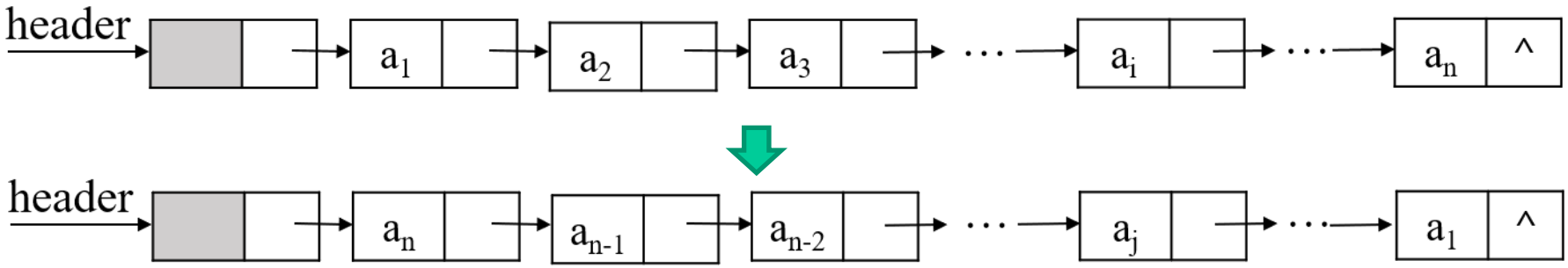
```
int List（LIST header)
{
   position p;
   int n=0;
   p=header->next;
   while(p)
   {  visit(p->data);
      n++;
      p=p->next;
   }
   return(n);
}
```

复杂度怎么样呢？

问题：

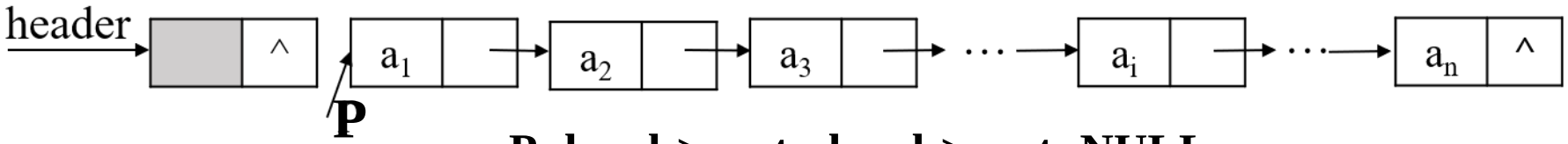
如何实现按逆序输出链表中的每一个元素？

【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。



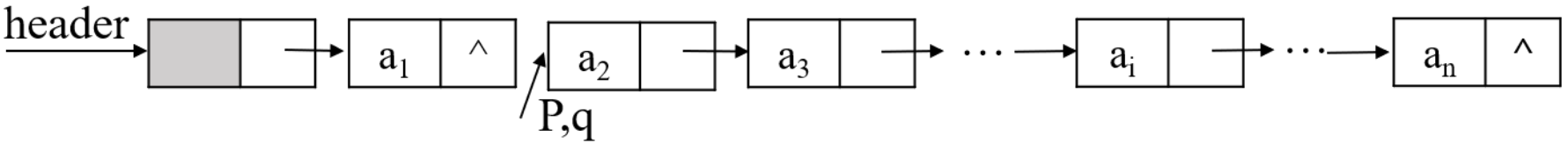
分析：

第1步：

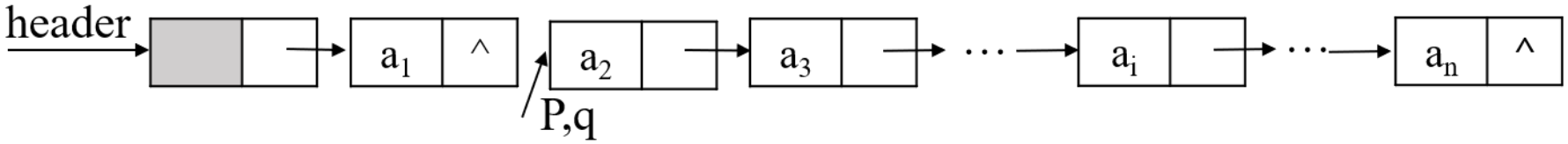


P=head->next; head->next=NULL

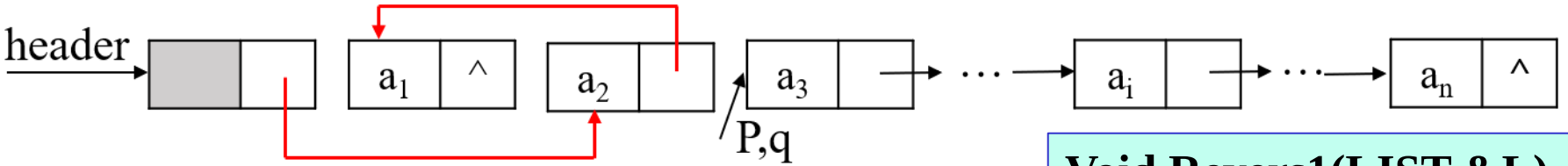
第2步：



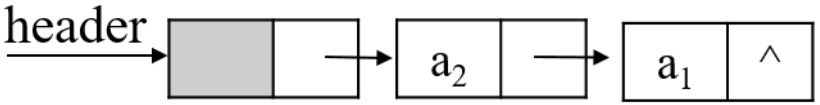
q=P->next; P->next=head->next; head->next=P; P=q;



第3步： q=P->next; P->next=head->next; head->next=P; P=q;



q=P->next; P->next=head-

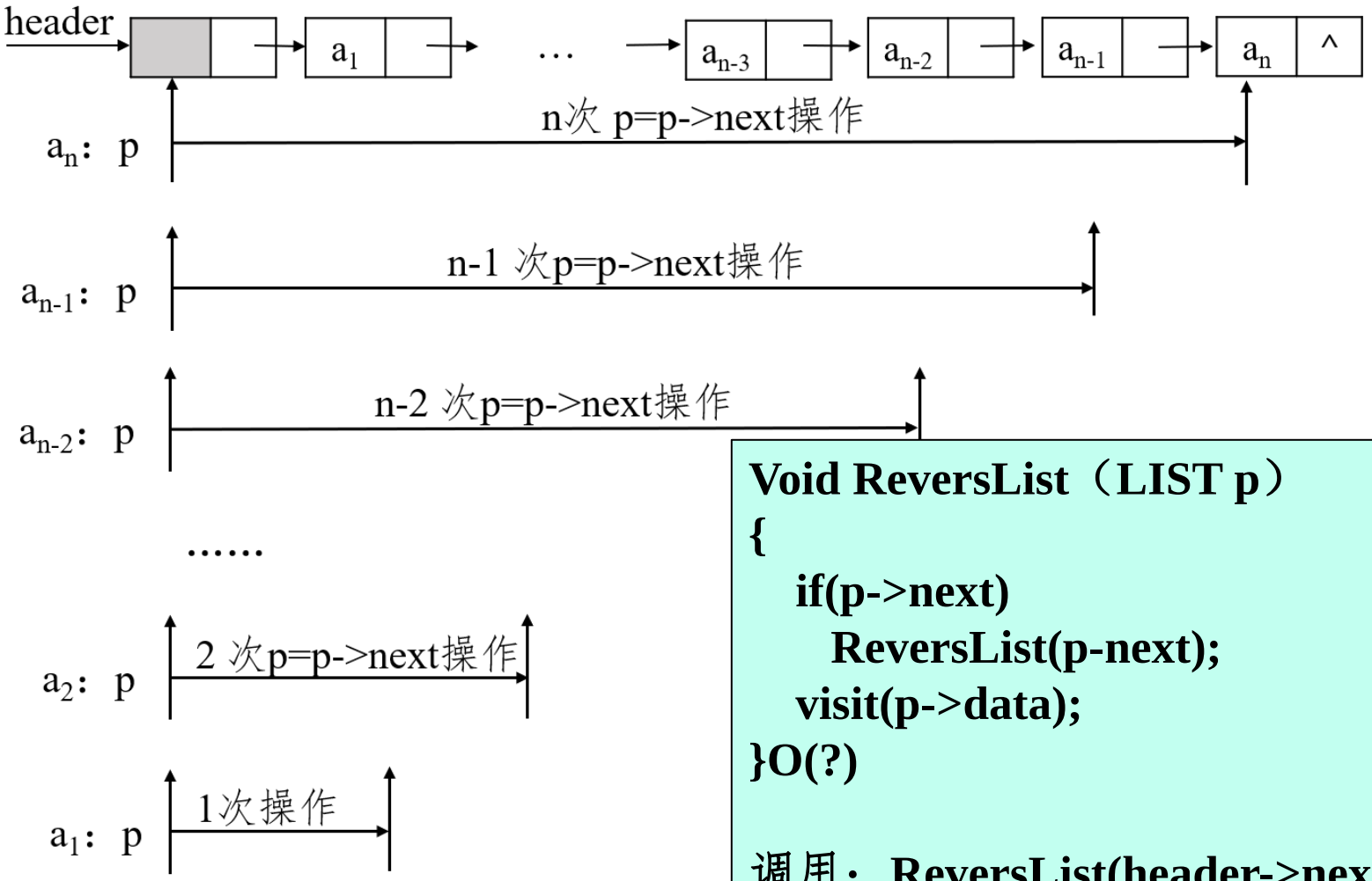


重复上述操作，把链表p的每一个结点依次插入到header的头节点之后，每次都形成一个新的第一个节点……，直到p=NULL为止。这个过程可以用下面循环完成：

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

```
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)
```

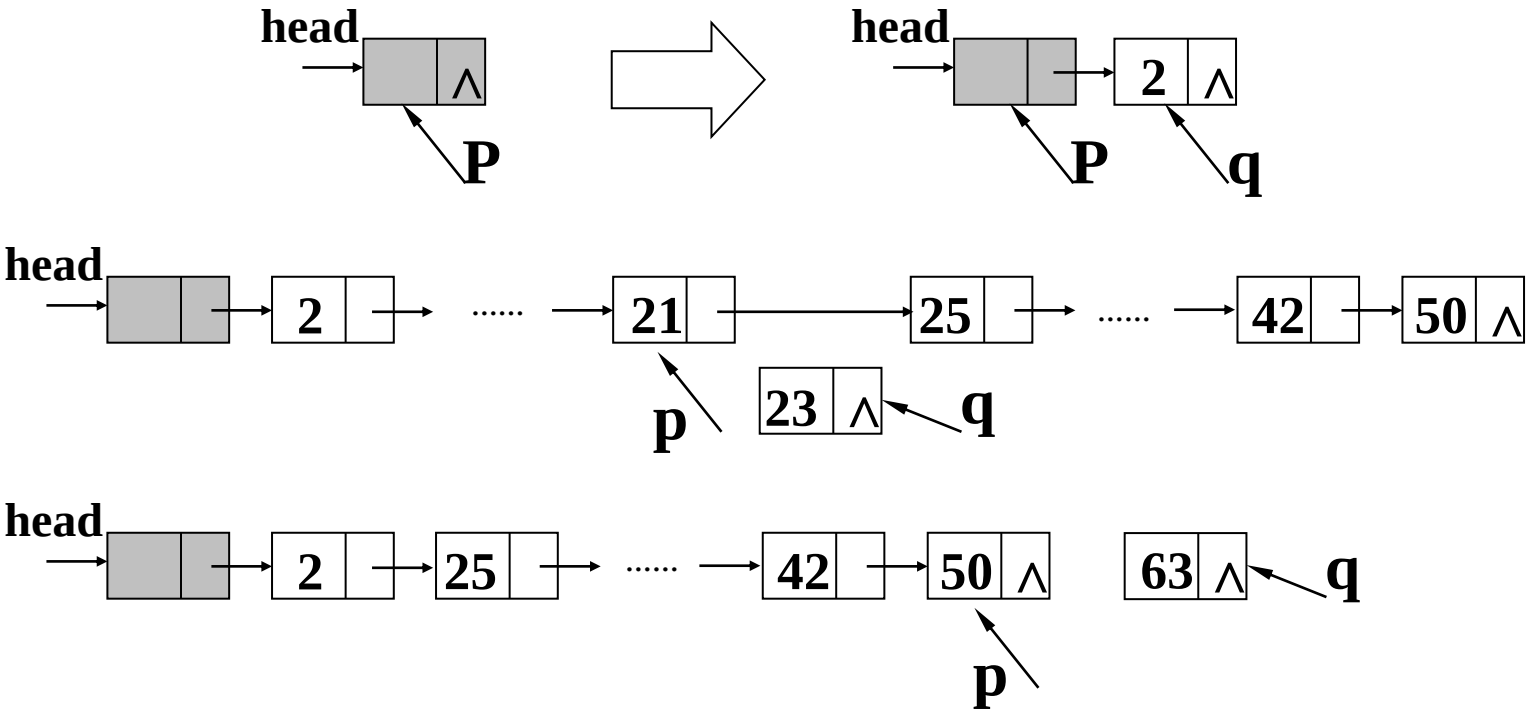
【例2-5】按逆序遍历链表




```
Void ReversList（LIST p）
{
   if(p->next)
      ReversList(p-next);
   visit(p->data);
}O(?)

调用：ReversList(header->next);
```

【例2-6】建立有序链表




```
P=head;
while((p->next!=Null)&&(p->next->data<=x))
        p=p->next;
Insert(x, p, &L )
```

```
typedef  struct NODE {
                    int   data ;
        struct NODE  *next ;
} ;
```

建立有序链表算法：

```
NODE *Create_Link()
{
     NODE  *head,*p;
     int  x;
     head=(NODE *)malloc(sizeof(NODE *));
     head->next=Null;                              //表头结点
     scanf("%d",&x);
     while(x!=-999)
          {  p=head;
             while((p->next!=Null)&&(p->next->data<=x))
                  p=p->next;
             ListInsert(&L,p,x)                    //插入
             scanf("%d",&x);
           }
      return(head);
}
```

【例2-7】 已知一个带有表头结点的单链表，结点结构为：

data	link

假设该链表只给出了头指针list。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点（k为正整数）。若查找成功，算法输出该结点的data域的值，并返回1；否则，只返回0。要求：

1）描述算法的基本设计思想；

2）描述算法的详细实现步骤；（ADT或文字表述）

3）采用C或C++描述算法，关键之处给出注释。

4）说明所设计算法的时间和空间复杂度。

算法思想：高效的算法只遍历一遍链表。

1）设置两个指针p和q，均指向头结点的下一个结点（首结点）。

2）p指针向后移动k个位置（即指向第k个结点）；

3）两个指针一起向后移动，当p指向最后一个结点时，q所指即是所求。

线性表 静态存储 与 动态存储的 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
…	…	…

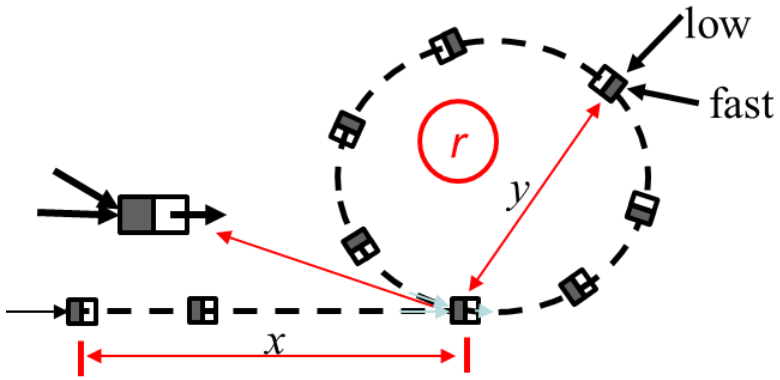
思考题：

1、单向链表环的问题

如何判断一个单向链表是否有环？找到环的入口结点。

2、线性链表，求倒数第K个数

3、线性链表，求中间位置的元素



4、单向链表交叉问题

如何判断两个单向链表是否交叉？找到交叉结点。