第2章　线性表

2.1 线性表的类型定义

- 线性结构的特点：

  在数据元素的非空有限集中，

1）有且仅有一个开始结点；

2）有且仅有一个终端结点；

3）除第一个结点外，集合中的每个数据元素均有且只有一个前驱；

4）除最后一个结点外，集合中的每个数据元素均有且只有一个后继。

- 线性序列：线性结构中的所有结点按其关系可以排成一个序列，记为$(a_1, \cdots, a_i, a_{i+1}, \cdots a_n)$

2.1 线性表的类型定义

1. 线性表

1）线性表是n(n ≥0)个数据元素的有限序列。

2）线性表是一种最常用且最简单的数据结构。

含有n个数据元素的线性表是一个数据结构：

List = (D,R)

其中：$D = \{a_i \mid a_i \in D_0, i=1,2,\cdots n, n \geq 0\}$

$R = \{N\}$，$N = \{< a_{i-1}, a_i > \mid a_{i-1}, a_i \in D_0, i = 2,3,\cdots n\}$

D_0 为某个数据对象——数据的子集

- 特性：均匀性，有序性（线性序列关系）

2.1　线性表的类型定义

1. 线性表

3）线性表的长度及空表

- 线性表中数据元素的个数n（n≥0）定义为线性表的长度
- 当线性表的长度为0 时，称为空表。
- a_i 是第i个数据元素，称i为a_i 在线性表中的位序。

2. 线性表的基本操作 p19～p20

1）InitList(&L) 初始化，构造一个空的线性表
2）ListLength(L) 求长度,返回线性表中数据元素个数
3）GetElem(L,i,&e) 取表L中第i个数据元素赋值给e
4）LocateElem(L,e) 按值查找，若表中存在一个或多个值为e的结点，返回第一个找到的数据元素的位序，否则返回一个特殊值。
5）ListInsert(&L,i,e) 在L中第i个位置前插入新的数据元素e，表长加1。
6）ListDelete(&L,i,e) 删除表中第i个数据元素，e返回其值，表长减1。

线性表的基本操作举例

- 例2-1　求A = A ∪ B　P20算法2.1
  - 时间复杂度：LocateElem()执行次数 O(ListLength(A)*ListLength(B))
- 例2-2 合并LA 和 LB 到LC中

  P20-21算法2.2
  - 时间复杂度：ListInsert()执行次数 O(ListLength(LA)+ListLength(LB))

例2-1　求A = A ∪ B　　算法：

```
void union(List &La，List Lb)
{
  La_len=ListLength(La);   Lb_len=ListLength(Lb);
  for ( i=1; i<=Lb_len; i++ )
   {
     GetElem(Lb，i，e);
     if (!LocateElem(La, e, equal))  ListInsert(&La，++La_en，e)
     }
}
```

例2-2 合并LA 和 LB 到LC中：

```
void  MergeList(list La, list Lb, list &Lc)
  {
        InitList(Lc);
        i=j=1;  k=0;
        La_len=ListLength(La);
        Lb_len=ListLength(lb);
```

```
  while((i<=La_len)&&(j<=Lb_len))
    { GetElem(La, i, ai); GetElem(Lb, j, bj);
      if (ai<=bj) { ListInsert(&Lc, ++k, ai); ++i;}
      else { ListInsert(&lc, ++k, bj); ++j;}
     }
  while(i<=La_len)
    { GetElem((La, i++, ai);
      ListInsert(&Lc, ++k, ai);
    }
  while(j<=Lb_len)
    { GetElem((Lb, j++, bj);
      ListInsert(&Lc, ++k, bj);
    }
}
```

2.2 线性表的顺序表示和实现

1. 顺序表——线性表的顺序存储结构

1）在计算机内存中用一组地址连续的存储单元依次存储线性表中的各个数据元素。

2）假设线性表的**每个**元素需占用**L个**存储单元，并以所占的第一个单元的存储地址作为数据元素的**起始存储位置**，则线性表中第i+1个数据元素的存储位置$Loc(a_{i+1})$和第i个数据元素的存储位置$Loc(a_i)$之间满足下列关系：

$$Loc(a_{i+1}) = Loc(a_i) + L$$

一般来说，线性表的第i个元素a_i的存储位置为：

$$Loc(a_i) = Loc(a_1) + (i-1)*L$$

其中$Loc(a_1)$是线性表的第一个数据元素a_1的存储位置，通常称作线性表的**起始位置**或**基地址**。

1. 顺序表—线性表的顺序存储结构

3）线性表的顺序存储结构示意图——p22图2.2

- 用“物理位置”相邻来表示线性表中数据元素之间的逻辑关系。
- 根据线性表的顺序存储结构的特点，只要确定了存储线性表的**起始位置**，线性表中任一数据元素都可随机存取，所以，线性表的顺序存储结构是一种**随机存取**的存储结构。

2. 顺序存储线性表的描述

- C语言中静态分配描述 p22

```
#define LIST_MAX_LENTH    100/或者N/或者是一个常数
typedef  struct ElemType {
      int    *elem;
      int    length;
      } SqList;
SqList  L;
```

2. 顺序存储线性表的描述

- C语言中静态分配描述 p22

- 求置空表

```
Status ClearList( &L )
    {
        L.length=0;
        return  OK;
     }
```

2. 顺序存储线性表的描述

- C语言中静态分配描述 p22

- 求长度

```
Satus List length( SqList  L )
    {
       length= L.length;
       return  OK;
     }
```

2. 顺序存储线性表的描述

❖ C语言中静态分配描述 p22

- 初始化

```
Status InitList_ SqList( SqList  L )
    {
       L.elem=(*int) malloc(LIST_MAX_LENGTH
                    *sizeof(int) );
       if(!L.elem) exit(Overflow) ;
       L.length=0;
       return  OK;
     }
```

2. 顺序表的描述
1）C语言中动态分配描述 p22

```
#define LIST_INIT_SIZE   100
#define LISTINCREMENT  10
typedef  struct {
     ElemType   *elem;
     int        length;
     int        listsize;
     }SqList;
SqList  L;
```

说明：

- elem数组指针指向线性表的基地址
- length指示线性表的当前长度
- listsize指示顺序表当前分配的存储空间大小

当空间不足时，再分配的存储空间增量为LISTINCREMENT*sizeof(ElemType)

2）几个基本操作
①初始化

- p23算法2.3

```
Status InitList_SqList(SqList &L)
{
 L.elem=(ElemType*)malloc(LIST_INIT_SIZE
              *sizeof(ElemType));
    if (!L.elem) exit(OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
}
```

②插入　p24算法2.4

```
Status ListInsert_sq(SqList &L,int i,ElemType e)
{
 if (i<1 || i>L.length+1)  return ERROR;
 if (L.length >= L.listsize)
  {  newbase=(ElemType*)realloc(L.elem,
   (L.listsize+LISTINCREMENT)*sizeof(ElemType));
     if (!newbase) exit(OVERFLOW);
     L.elem = newbase;
     L.listsize+=LISTINCREMENT;
  }
```

- 需将第n(即L.length)至第i个元素向后移动一个位置。注意：C语言中数组下标从0开始，则表中第i个数据元素是L.elem[i-1]。

②插入　p24算法2.4

函数realloc的格式及功能

格式：　void *realloc(void *p,unsigned size)

功能：将p所指向的已分配内存区域的大小改为size。

size可以比原来分配的空间大或小。

②插入（续）

```
    q=&(L.elem[i-1]);
    for (p=&(L.elem[L.length-1]);p>=q;--p)
            *(p+1) = *p;
    *q=e;
    ++L.length;
    return OK;
}
```

③删除 p24～p25算法2.5

```
Status ListDelete_sq(SqList &L,int i,ElemType &e)
{
   if (i<1||i>L.length) return ERROR;
    p=&(L.elem[i-1]);
    e=*p;
    q=L.elem+L.length-1;//表尾元素结点
    for (++p;p<=q;++p) *(p-1)=*p;
    --L.length;
    return OK
 }
```

- 需将第i+1至第L.length个元素向前移动一个位置

插入和删除算法时间分析

- 用“移动结点的次数”来衡量时间复杂度。与表长及插入位置有关。
- 插入：
  - 最坏：i=1，移动次数为n
  - 最好： i=表长+1，移动次数为0
  - 平均：等概率情况下，平均移动次数n/2
- 删除：
  - 最坏：i=1，移动次数为n-1
  - 最好： i=表长，移动次数为0
  - 平均：等概率情况下，平均移动次数(n-1)/2

查找

p25～p26算法2.6

```
int LocateElem_Sq(SqList L, ElemType e)
{
 i=1;
 while ( i<=L.length && e != L.elem[i-1]) ++i;
 if (i<=L.length) return i; else return 0;
}
```

查找的另一种描述

```
int LocateElem_Sq(SqList L, ElemType e)
{ i=1;
  p=L.elem;
  while (i<=L.length && e!=*p++) ++i;
  if (i<=L.length) return i;
  else return 0;
}
```

合并　P26算法2.7的另外一种描述

```
void MergeList_Sq(SqList La,SqList Lb,SqList &Lc)
{ int i=0, j=0,k=0;
  InitList_SqList(Lc);
  while (i<=La.length-1 && j<=Lb.length-1)
        {  if (La.elem[i]<=Lb.elem[j])
                Lc.elem[k++]=La.elem[i++];
           else Lc.elem[k++]=Lb.elem[j++]; }
  while (i<=La.length-1)
           Lc.elem[k++]=La.elem[i++];
  while (j<=Lb.length-1)
           Lc.elem[k++]=Lb.elem[j++];
  Lc.length = k;}
```

合并　P26算法2.7

```
void MergeList_Sq(SqList La,SqList Lb,SqList &Lc)
{ 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);
  pa_last=La.elem+La.length-1;
  pb_last=Lb.elem+Lb.length-1;
  while (pa<=pa_last&&pb<=pb_last)
     {  if(*pa<=*pb) *pc++=*pa++;
         else *pc++=*pb++; }
  while (pa<=pa_last) *pc++=*pa++;
  while (pb<=pb_last) *pc++=*pb++;
}
```

建立

```
#define LIST_INIT_SIZE  100
Status CreateList_Sq(SqList  &L, int n )
{  int i;
   L.elem = (int*) malloc (LIST_INIT_SIZE*sizeof(int));
   if (!L.elem) exit(OVERFLOW);
   L.length = n;
   L.Listsize = LIST_INIT_SIZE;
   for (i=0; i<n;i++) scanf("%d “,&L.elem[i]);
   for (i=0; i<n;i++) printf( “%d,",L.elem[i]);
   printf("\n");
}
```

递增插入

```
Status OrderInsert_Sq(SqList &La, ElemType x)
{ int i=0;
  if (La.length >= La.listsize) return OVERFLOW;
 while (i< La.length && La.elem[i]<x) i++;
 for (int j = La.length-1; j>= i;j--)
     La.elem[j+1] = La.elem[j];
 La.elem[i]= x;
 La.length++;
 return OK;
}
```

递减插入

```
Status DeOrderInsert_Sq(SqList &La, ElemType x)
{  int i, j;
   if (La.length >= La.listsize) return OVERFLOW;
   i=La.length-1;
   while ( i>=0 && La.elem[i]<x ) i--;
   for (j=La.length-1; j>i;j--)
       La.elem[j+1] = La.elem[j];
   La.elem[i+1]= x;
   La.length ++;
   return OK;
}
```

4. 顺序表的分析

1）优点

- 顺序表的结构简单
- 顺序表的存储效率高，是紧凑结构
- **顺序表是一个随机存储结构（直接存取结构）**

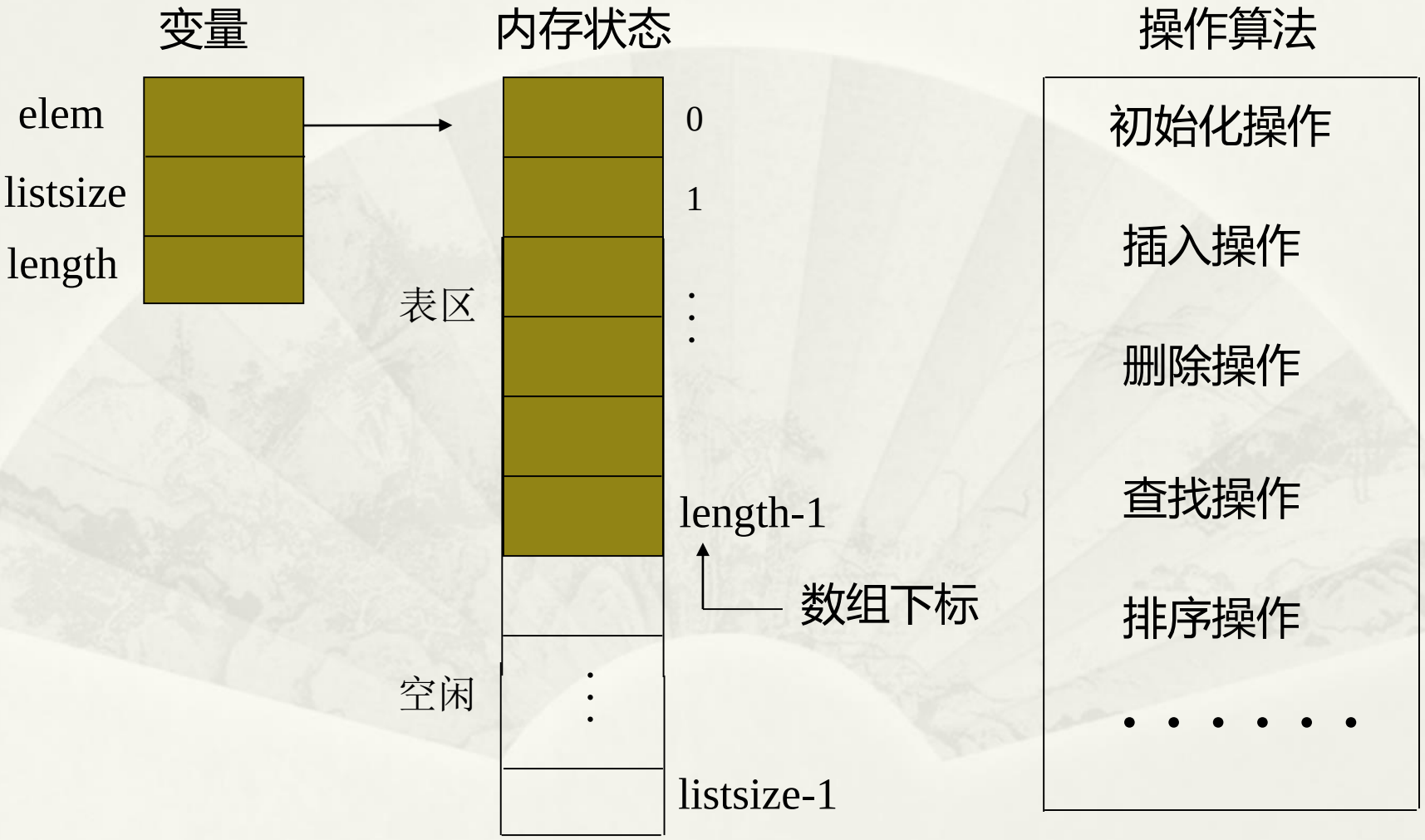
2）缺点

- 在顺序表中进行插入和删除操作时，需要移动数据元素，算法效率较低。
- 对长度变化较大的线性表，或者要预先分配较大空间或者要经常扩充线性表，给操作带来不方便。
- 原因：数组的静态特性造成

作业

- 2.1 编写程序，建立并显示一个有10个数据元素的顺序线性表。
- 2.2 实现顺序线性表的插入、查找、删除等算法。

顺序表之整体概念


变量
内存状态
操作算法
elem
listsize
length
表区
0
1
length-1
数组下标
空闲
listsize-1
初始化操作
插入操作
删除操作
查找操作
排序操作
. . . . . .

顺序表之整体概念：

顺序表有下列缺点：

（1）插入、删除操作时需要移动大量元素，效率较低；

（2）最大表长难以估计，太大了浪费空间，太小了容易溢出。

因此，在插入和删除操作是经常性操作的应用场合选用顺序存储结构不太合适，此时可以选用链式存储结构，数据元素之间的逻辑关系由结点中的指针来指示。

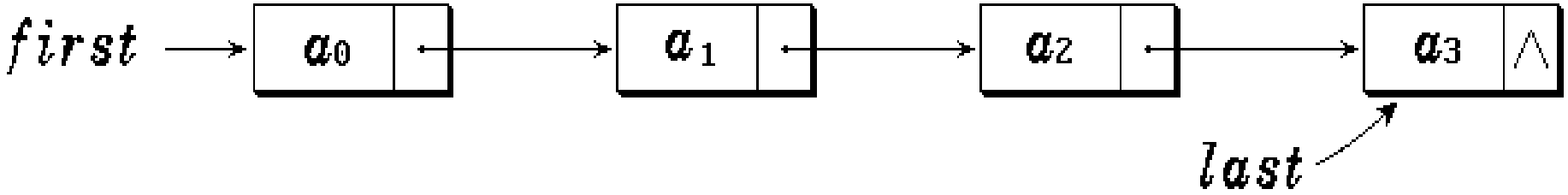
2.3 线性表的链式表示和实现

1. 线性链表

- 特点：在内存中用一组任意的存储单元来存储线性表的数据元素，用每个数据元素所带的指针来确定其后继元素的存储位置。这两部分信息组成数据元素的存储映像，称作结点。
- 结点：数据域 + 指针域（链域）

- 链式存储结构：n个结点链接成一个链表
- 线性链表（单链表）：链表的每个结点只包含一个指针域。

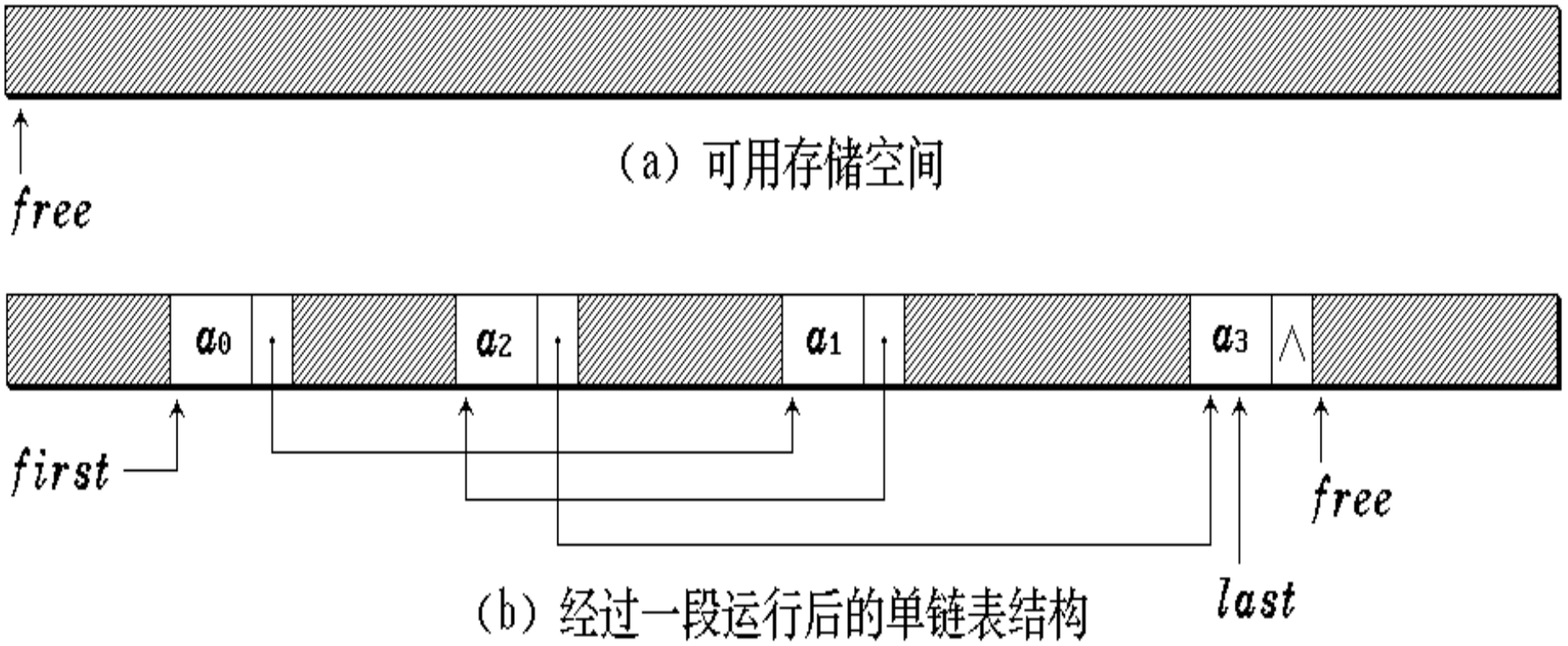
单链表（Singly Linked List）



- first　头指针
- last　尾指针

- ^ 指针为空
- 单链表由头指针唯一确定，因此常用头指针的名字来命名。如表first.

单链表的存储映像



1）线性链表的描述　p28

```
typedef struct LNode{
    ElemType  data;
    Struct LNode   *next;
} LNode, *LinkList;
LinkList  L;  //L是LinkList类型的变量，表示单链表的头指针
```

2）术语

- 头指针：指向链表中第一个结点
- 第一个数据元素结点（开始结点）
- 头结点：有时在单链表的第一个数据元素结点之前附设一个结点，称之头结点。
  - 说明：头结点的next域指向链表中的第一个数据元素结点。
  - 对于头结点数据域的处理：

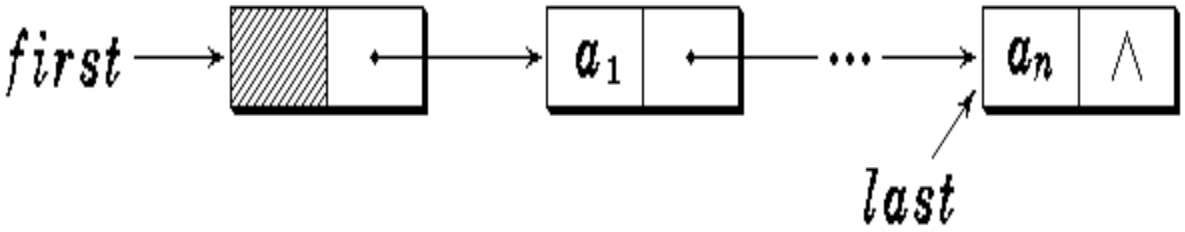
    a. 加特殊信息

    b. 置空

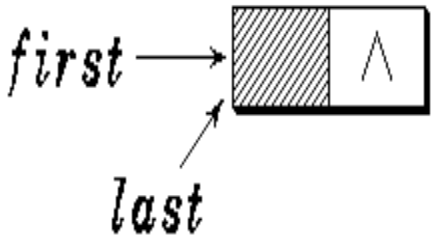
    c. 如数据域为整型，则在该处存放链表长度信息。

3）带头结点的单链表示意图 p28图2.7

- 由于开始结点的位置被存放在头结点的指针域中，所以对链表第一个位置的操作同其他位置一样，无须特殊处理。
- 无论链表是否为空，其头指针是指向头结点的非空指针，因此对空表与非空表的处理也就统一了，简化了链表操作的实现。



非空表



空表

思考

对于带头节点、不带头节点的情况，分别给出：

1. 有三个元素节点的示意图
2. 只有一个元素节点的示意图
3. 没有元素节点的示意图

2. 基本操作

1）取元素　p29 算法2.8

2）插入元素　　p30 算法2.9

3）删除元素　　p30 算法2.10

4）建立链表　　p30～p31 算法2.11

5）有序链表的合并　p31 算法2.12

6）查找（按值查找）

7）求长度

8）集合的并运算

取元素（按序号查找） p29 算法2.8

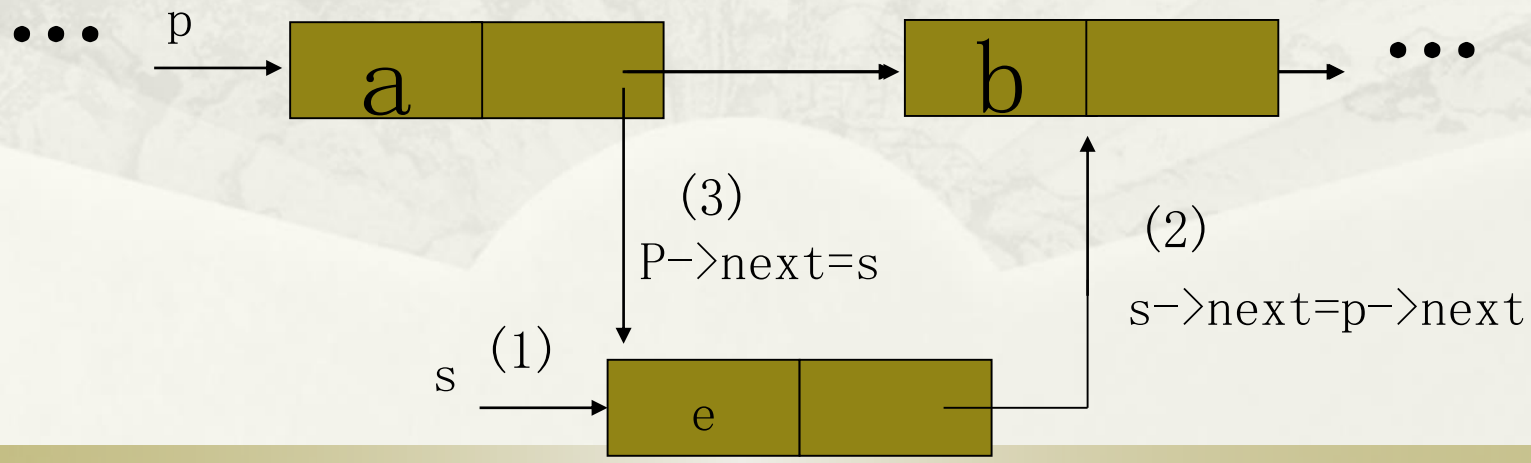
从链表的头指针出发，顺链域next逐个结点往下搜索，直至找到第i个结点为止（j=i）

```
Status GetElem_L(LinkList L,int i,ElemType &e)
{ LinkList p;
  p=L->next;  int j=1;
  while (p && j<i) { p=p->next; ++j; }
  if (!p || j>i) return ERROR;
  e=p->data;
  return OK;
}
```

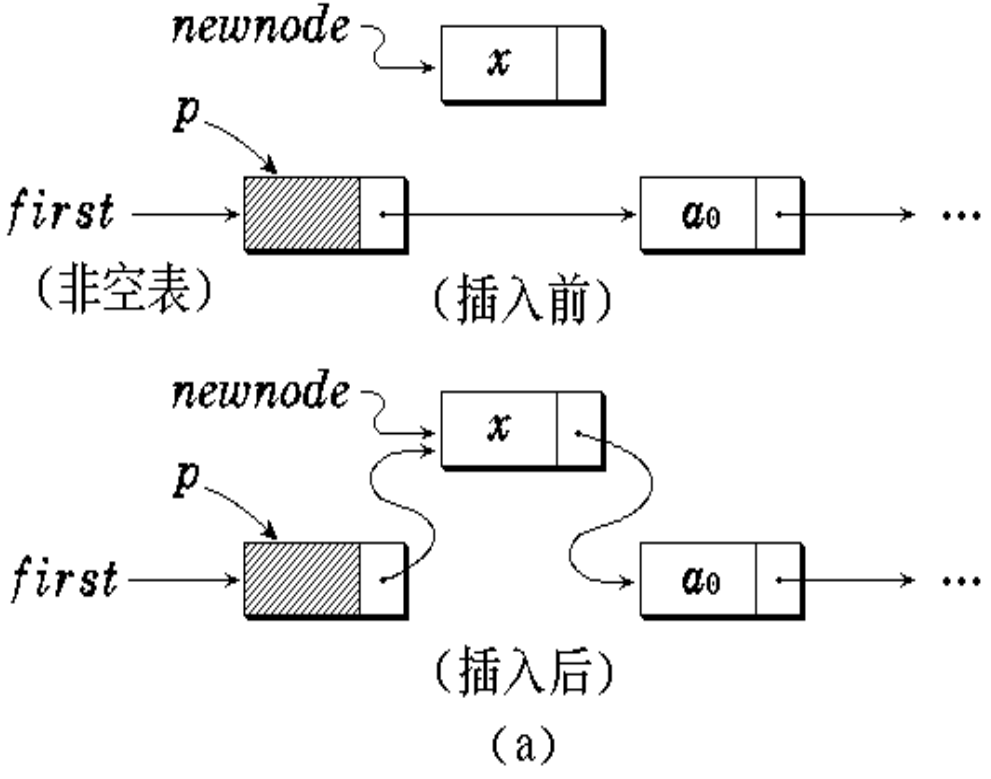
插入元素p30 算法2.9

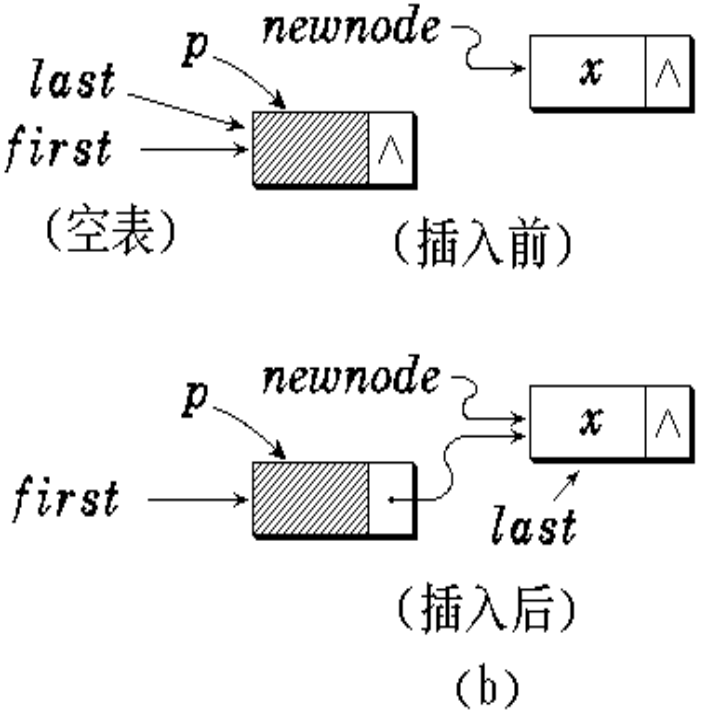
在第i个元素之前插入，先找到第i-1个结点

```
Status ListInsert_L(LinkList &L,int i,ElemType e)
{ LinkList p,s;
   p=L;    int j=0;
   while (p && j<i-1) { p=p->next; ++j;}
   if (!p || j> i-1) return ERROR;
   s = (LinkList) malloc( sizeof (LNode));
   s->data = e; s->next = p->next;
   p->next = s;
   return OK;}
```



在带表头结点的单链表第一个结点前插入新结点

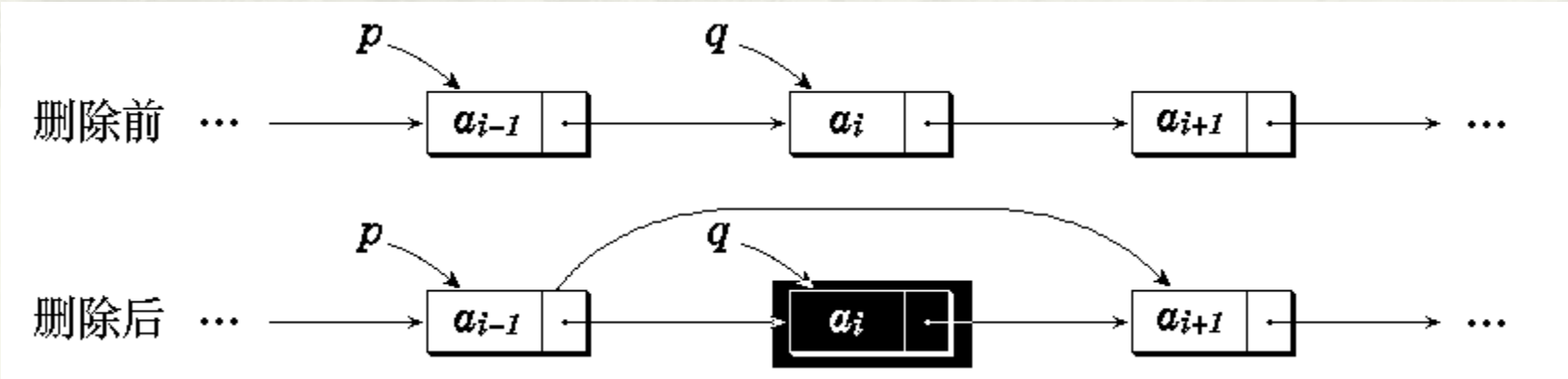




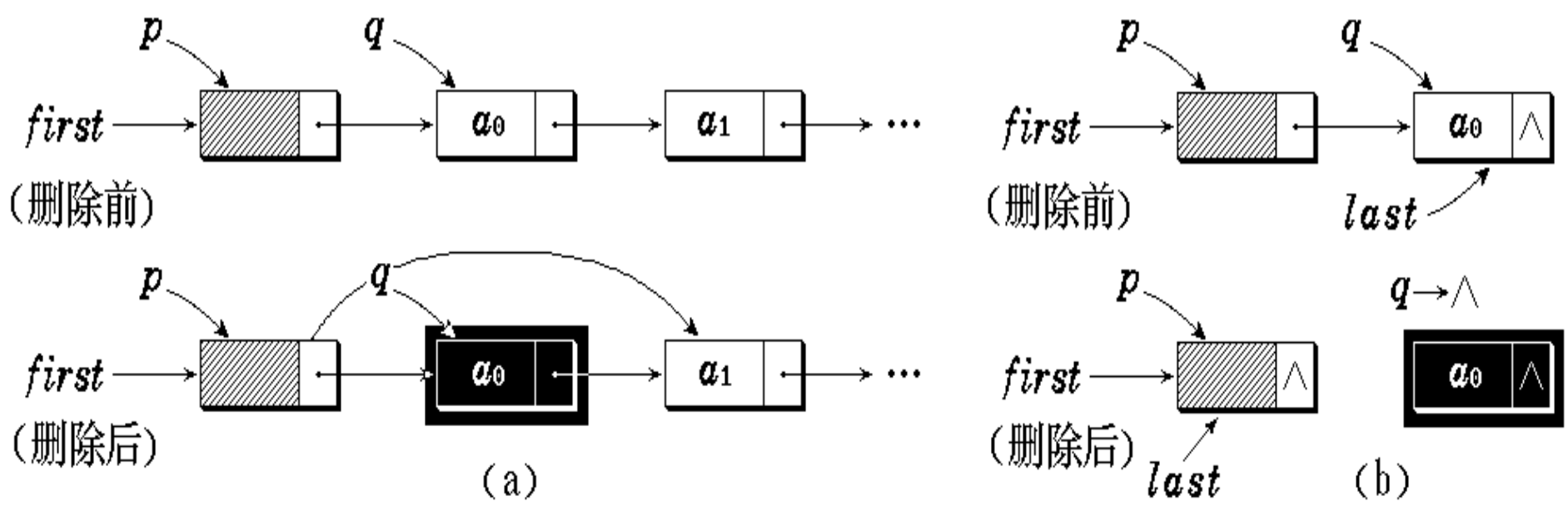

```
newnode→next = p→next;
if ( p→next ==NULL ) last = newnode;
p→next = newnode;
```

删除元素p30 算法2.10

```
Status ListDelete_L(LinkList &L, int i, ElemType &e)
{ LinkList p,q;
  p=L;    int j=0;
  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;          free(q);
  return OK;
}
```



从带表头结点的单链表中删除第一个结点




```
q = p→next;
p→next = q→next;
delete q;
if ( p→next == NULL ) last = p;
```

建立链表（头插法建表）p31 算法2.11

在链表表头插入新结点，结点次序与输入次序相反。

```
void CreateList_L(LinkList &L, int n)
{ LinkList p;
  L=(LinkList)malloc(sizeof(LNode));
  L->next = NULL;
  for (int i=n;i>0;--i) {
      p=(LinkList)malloc(sizeof(LNode));
      scanf("%d",&p->data);
      p->next = L->next; L->next=p; }
  }
```

尾插法建表：将新结点插到链表尾部，须增设一个尾指针last，使其始终指向当前链表的尾结点。

合并有序链表p31 算法2.12

```
void MergeList_L(LinkList &La,LinkList &Lb,LinkList &Lc)
{ LinkList pa,pb,pc;
  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);
}
```

查找（按值查找）

```
int LinkLocate_L (LinkList L, int x)
{  int i;   LinkList p;
   p=L->next;  i=1;
   while (p!=NULL && p->data != x)
                { p= p->next; i++;}
   if (!p)  { printf ("Not found! \n");
                  return(0); }
   else { printf ("i=%d\n",i);return (i); }
}
```

求长度

```
int LinkLength_L (LinkList L)
{  int j;  LinkList p;
   p=L->next;  j=0;
   while (p) {p=p->next;j++;}
   return(j);
 }
```

求长度

注意：p=NULL与p->next =NULL是不同的。

- 总结：

  带头结点：链表置空　　L-〉next = NULL;

  判断是否为空的条件　　if (L->next = NULL)

  不带头结点：则置空 L=NULL;

  判空条件　if (L==NULL)

集合并运算5-2 A=A∪B

```
void UnionList_L(LinkList &La,LinkList Lb)
{   LinkList  p,q,first;  int x;
    first = La->next;  p=Lb->next;
    while (p){
          x=p->data;  p=p->next;  q=first;
          while (q && q->data !=x)  q=q->next;
          if (!q) {  q=(LinkList)malloc(sizeof(LNode));
                     q->data = x;  q->next = La->next;
                     La->next = q; }
                }
}
```

说明：

- first的位置始终不变；
- 插入位置在La表的表头元素之前；
- 查找不会在刚刚插入的结点间进行，只能从first指向的原始La表中进行 （因为每次查找时均有q=first）
- 时间复杂度： O(m*n)；

3. 循环链表

1）循环链表——是一种首尾相接的链表。

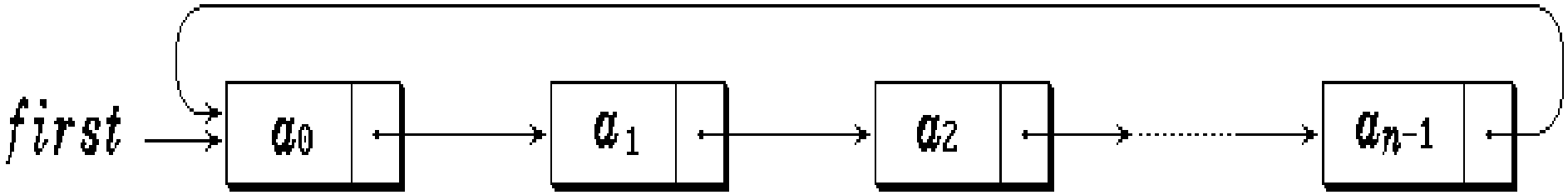
循环链表最后一个结点的next指针不为 0 (NULL)，而是指向了表头结点。在循环链表中没有NULL

为简化操作，在循环链表中往往加入表头结点。

特点：循环链表中，从任一结点出发都可访问到表中所有结点；而在单链表中，必须从头指针开始，否则无法访问到该结点之前的其他结点。

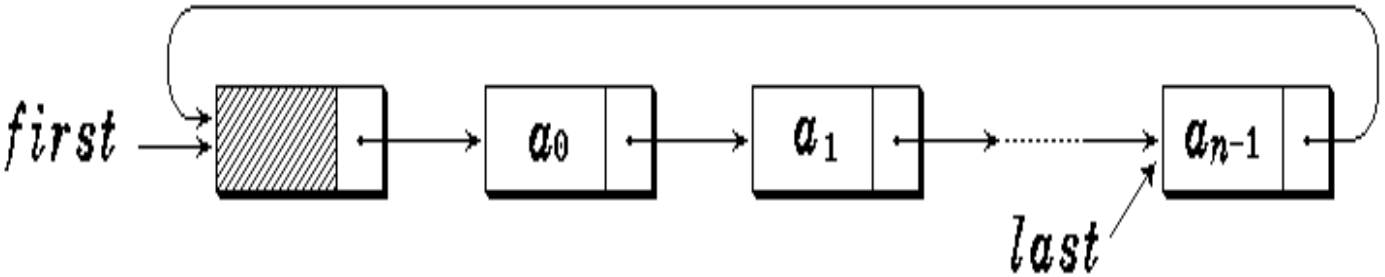
循环链表与单链表不同的是链表中表尾结点的指针域不是NULL，而是链表头结点的指针first（链表指针）

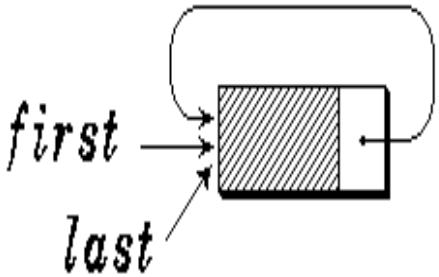
- 循环链表的示例



- 带表头结点的循环链表

（循环条件：p->next != first）





2）循环链表的操作

- 合并两个单链表

```
p=La;
while (p->next)  p=p->next;
p->next = Lb->next;
free(Lb);
```

时间复杂度　O(m)

合并两个循环链表

```
p=La;
while (p->next!=La)  p=p->next;
p->next=Lb->next;
p=p->next;
while (p->next!=Lb) p=p->next;
p->next=La;
free(Lb);
```

时间复杂度 O(m+n)

循环链表的建立算法

```
void CreateList_L(LinkList &L)
{  LinkList p;  int x;
  L= (LinkList)malloc(sizeof(LNode));
   L ->next=L;
  while (scanf("%d,",&x), x!=0 )
       {
         p=(LinkList)malloc(sizeof(LNode));
         p->data=x; p->next = L->next;
         L->next = p;
       }
}
```

显示输出算法（带头结点）——循环链表

```
void PrintList_LC(LinkList L)
 { LinkList p;
   p=L->next;  printf("L->");
   while (p!=L) {
      printf("%d->",p->data);
      p=p->next;  }
     printf("L\n");
 }
```

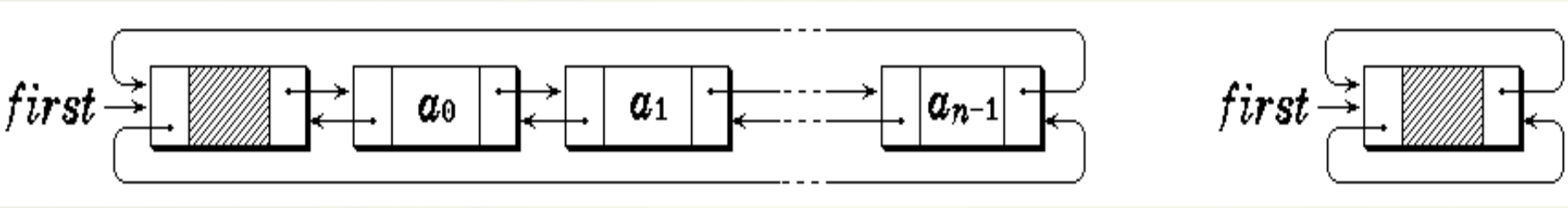
4.双向链表（Doubly Linked List）

- 双向链表是指在前驱和后继方向都能游历(遍历)的线性链表。

1） 双向链表的结点结构：

前驱方向 ← (a)结点结构 → 后继方向

双向链表通常采用带表头结点的循环链表形式。



(b) 非空双向循环链表　　(c) 空表

- 对双向循环链表中任一结点的指针，有：

  p == p→prior→next == p→next→prior

- 置空表：

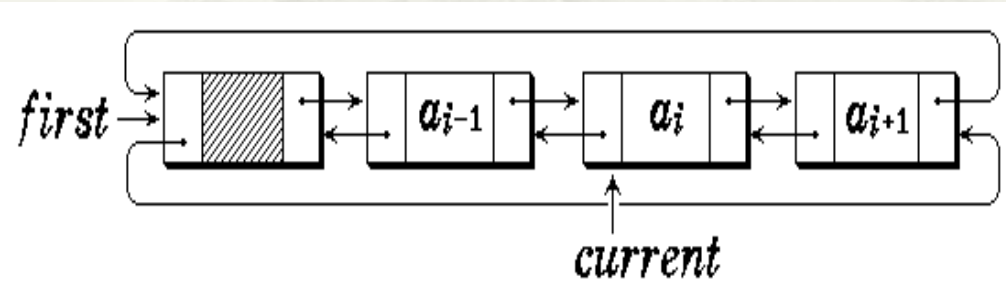
  p→prior = p ; p→next = p;

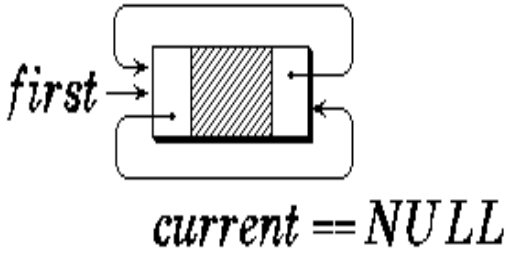
2）双向循环链表存储结构的描述　p35～p36

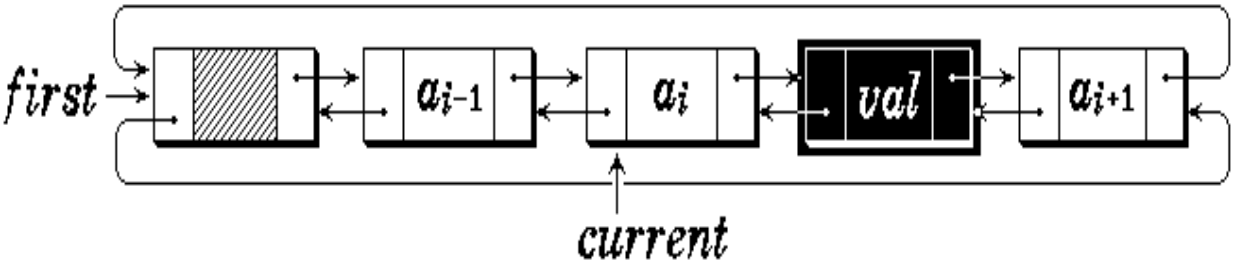
```
typedef struct DuLNode{
    ElemType   data;
    struct DuLNode   *prior;
    struct DuLNode   *next;
  }DuLNode, *DuLinkList;
DuLinkList  d,p,s;
```

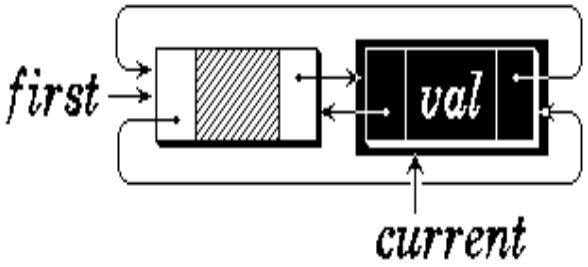
双向循环链表的插入算法

```
s→prior = current; (1)
s→next =current→next;    (2)
current→next = s;    (3)
s→next→prior = s;    (4)
```



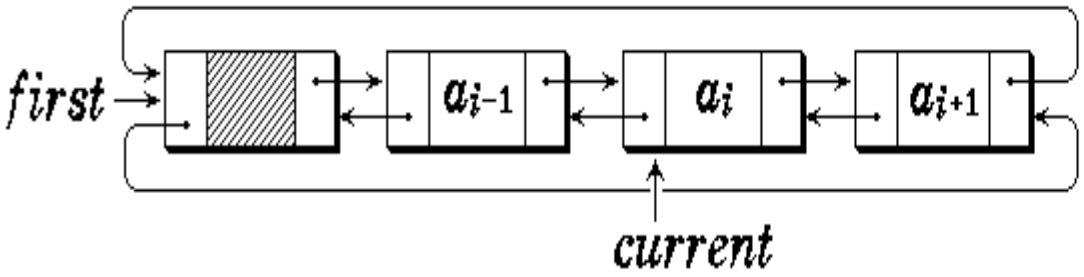


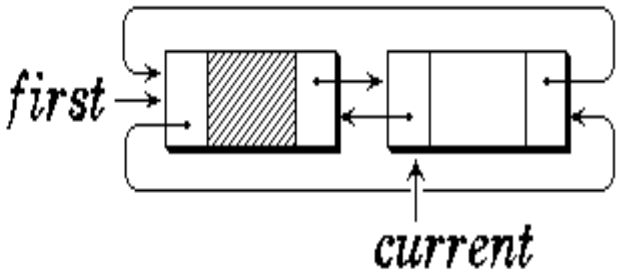


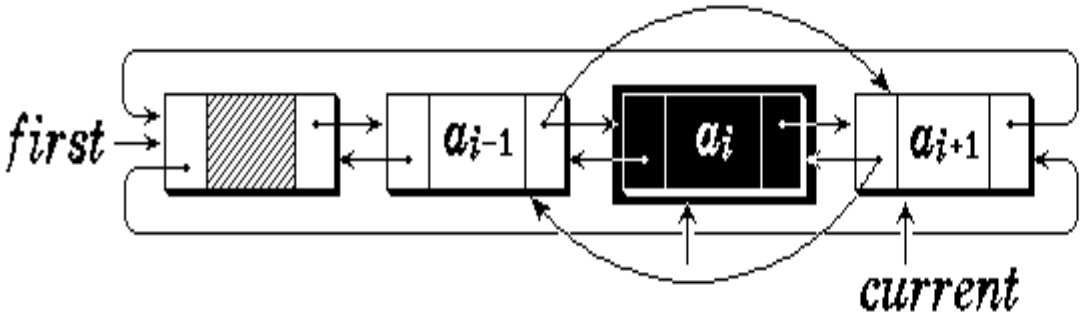


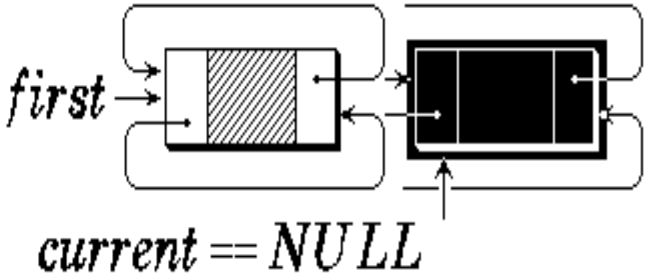
双向循环链表的删除算法

```
current→next→prior = current→prior;
current→prior→next = current→next;
```









3）基本操作：
双向循环链表的建立

```
void CrtList_DuL(DuLinkList &L)
{ DuLinkList p;     int x;
  L=p=(DuLinkList)malloc(sizeof(DuLNode));
    L ->next=L;   L->prior =L;
  while (scanf("%d",&x),x!=0){
      p->next=(DuLinkList)malloc(sizeof(DuLNode));
      p->next->prior =p;    p=p->next;
      p->data=x;
      }
  p->next=L;       L->prior =p;
}
```

显示输出

```
void PrtList_DuL(DuLinkList L)
{ DuLinkList p;
  p=L->next;  printf("L->");
  while (p!=L){
      printf("%d->",p->data);
      p=p->next;
      }
  printf("\n");
  }
```

2.4 一元多项式的表示和相加

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

$$= \sum_{i=0}^{n} a_i x^i$$

- n阶多项式$P_n(x)$有n+1项。
  - 系数 a_0，a_1，a_2，…，a_n
  - 指数 0，1，2，…，n。按升幂排列
- 在计算机中，可以用一个线性表来表示
  - P=(a_0, a_1，… ，a_n)

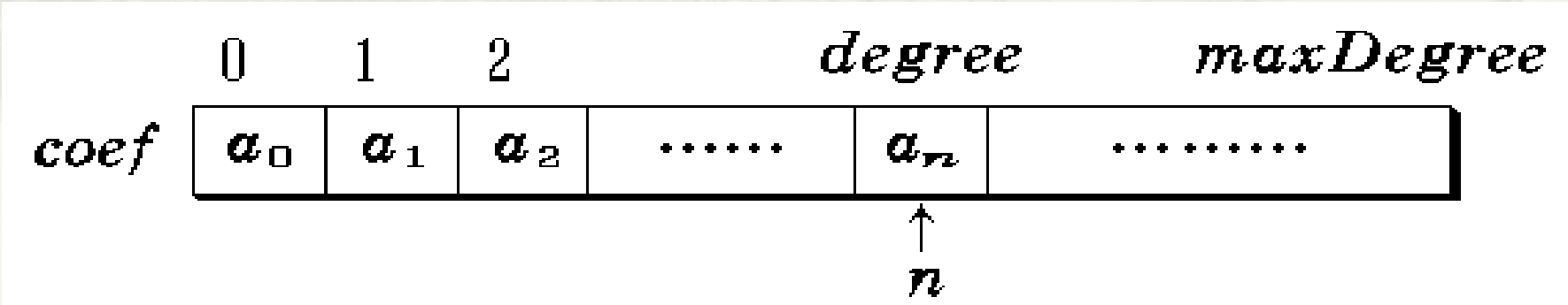
1. 第一种表示方法

$$P_n=(a_0, a_1, \cdots, a_n)$$

适用于指数连续排列、“0”系数较少的情况。但对于指数不全的多项式，如

$$P_{20000}(x) = 3 + 5x^{50} + 14x^{20000}$$

会造成系统空间的巨大浪费。



2. 第二种表示方法

一般情况下，一元多项式可写成：

$P_n(x)=p_1x^{e1}+p_2x^{e2}+\cdots+p_mx^{em}$

其中：p_i是指数为ei的项的非零系数，

$0\leqslant e1 \leqslant e2 \leqslant \cdots \leqslant em \leqslant n$　可用二元组表示

$((p_1,e1),(p_2,e2),\ \cdots\ ,(p_m,em))$

例：$P_{999}(x) = 7x^3 - 2x^{12} -8x^{999}$

表示成：((7,3),(-2,12),(-8,999))

	0	1	2		i		m
coef	a_0	a_1	a_2	……	a_i	……	a_m
exp	e_0	e_1	e_2	……	e_i	……	e_m

3. 一元多项式的抽象数据类型定义

ADT *Polynomial* {

数据对象： D=$\{a_i | a_i \in TermSet, i=1,2,\cdots,m, \quad m \geqslant 0\}$ TermSet中的每个元素包含一个表示系数的实数和表示指数的整数}

数据关系： R1=$\{\langle a_{i-1}, a_i\rangle | a_{i-1}, a_i \in D$, 且$a_{i-1}$中的指数值<$a_i$中的指数值, $i=2,\cdots n\}$

基本操作：

}ADT Polynomial

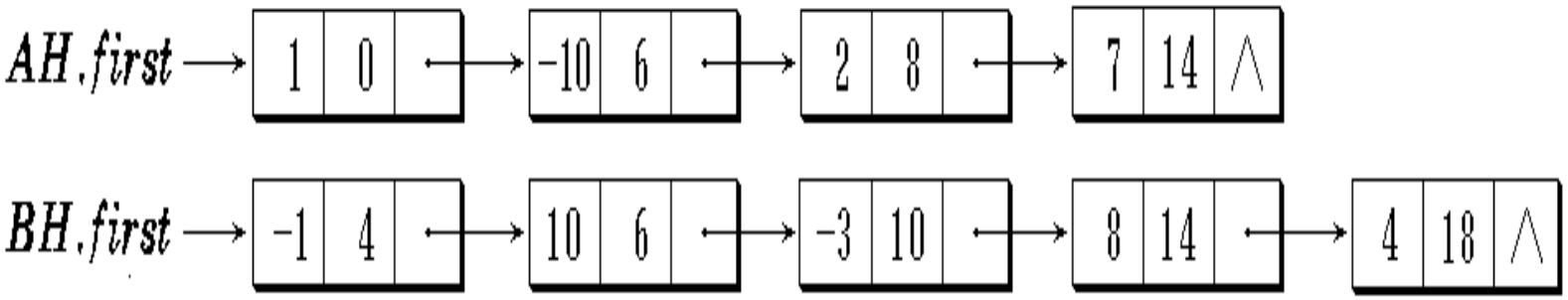
4. 抽象数据类型(Polynomial)的实现

```
  typedef struct {
      float coef;
      int expn;
}term, ElemType;
//term用于本ADT, ElemType为LinkList
的数据对象名
typedef LinkList polynomial;
```

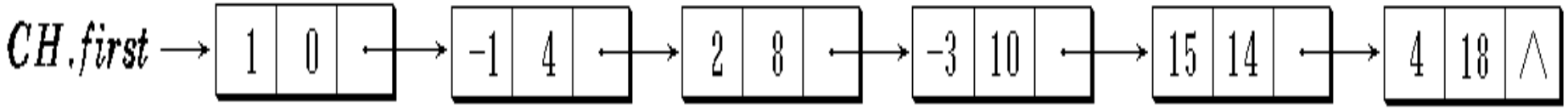
多项式链表的相加

$$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$$

$$BH = -x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$$



(a) 两个相加的多项式



(b) 相加结果的多项式

两个多项式的相加

- 结果多项式另存
- 扫描两个相加多项式，若都未检测完：
  - 若当前被检测项指数相等，系数相加。若未变成0，则将结果加到结果多项式。
  - 若当前被检测项指数不等，将指数小者加到结果多项式。
- 若有一个多项式已检测完，将另一个多项式剩余部分复制到结果多项式。