

第2章 线性表

2.1 线性表的类型定义

2.2 线性表的顺序表示和实现

2.3 线性表的链式表示和实现

2.4 一元多项式的表示及相加

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

例1、26个英文字母组成的字母表

(A, B, C, ..., Z)

例2、 学生健康情况登记表如下：

| 姓 名 | 学 号 | 性 别 | 年 龄 | 健康情况 |
|-------|--------|-------|-------|-------|
| 王小林 | 790631 | 男 | 18 | 健康 |
| 陈 红 | 790632 | 女 | 20 | 一般 |
| 刘建平 | 790633 | 男 | 21 | 健康 |
| 张立立 | 790634 | 男 | 17 | 神经衰弱 |
| | | | | |

从以上例子可看出线性表的逻辑特征是：

- 在非空的线性表，有且仅有一个开始结点 a_1 ，它没有直接前趋，而仅有一个直接后继 a_2 ；
- 有且仅有一个终端结点 a_n ，它没有直接后继，而仅有一个直接前趋 a_{n-1} ；
- 其余的内部结点 $a_i (2 \leq i \leq n-1)$ 都有且仅有一个直接前趋 a_{i-1} 和一个直接后继 a_{i+1} 。

线性表是一种典型的线性结构。

2.1 线性表的类型定义

1. 线性表

1) 线性表是 n ($n \geq 0$) 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：

$$\text{List} = (D, R)$$

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

$$R = \{N\}, N = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D_0, \\ i = 2, 3, \dots, n \}$$

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 ($n \geq 0$) 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置, 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。

说明

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

线性表操作举例

例：假设线性表 $L1=(8, 15, 24, 12, 30)$ ， $i=3$ ， $x_1=12$ ， $x_2=20$ ，求对 $L1$ 进行如下操作的结果。

(1) $ListLengh(L1)$ 结果为5。

(2) $GetElem(L1, i, \&x)$ 结果为 $x=24$ 。

(3) $LocateElem(L1, x_1)$ 结果为4。

(4) $ListInsert(\&L1, i, x_2)$

结果为 $L1=(8, 15, 20, 24, 12, 30)$

(5) $ListDelete(\&L1, i, x)$

结果为 $L1=(8, 15, 24, 12, 30)$ $x=20$

线性表的基本操作举例

* 例2-1 求 $A = A \cup B$ P20算法2.1

- * **分析**：用线性表LA和LB分别表示集合A和B。然后对线性表作如下操作：扩大线性表LA，将存在于线性表LB中而不存在于线性表LA中的数据元素插入到线性表LA中去。
- * **具体操作实现**：从线性表LB中依次取得每个数据元素，并根据它的值在线性表LA中进行查找，若不存在，则插入它。
 - * 时间复杂度：LocateElem()执行次数
 $O(\text{ListLength}(A) * \text{ListLength}(B))$



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)

    }
}
```

线性表的基本操作举例

■ 例2-2 合并LA 和 LB 到LC中

P20-21算法2.2

- ◆ 已知: **LA**、**LB**: 非递减有序排列
- ◆ 要求: **LC**: 非递减有序排列
- ◆ 分析: 先设LC为空表, 然后将LA或LB中的元素逐个插入到LC中即可。
- ◆ 举例: $LA=(3, 5, 8, 11)$; $LB=(2, 6, 8, 9, 11, 15, 20)$
则 $LC=(2, 3, 5, 6, 8, 8, 9, 11, 11, 15, 20)$
- ◆ 时间复杂度: **ListInsert()** 执行次数
 $O(\text{ListLength}(LA)+\text{ListLength}(LB))$



此问题的算法如下：

```
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 $\text{Loc}(a_{i+1})$ 和第 i 个数据元素的存储位置 $\text{Loc}(a_i)$ 之间满足下列关系：

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

一般来说，线性表的第 i 个元素 a_i 的存储位置为：

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

其中 $\text{Loc}(a_1)$ 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。

| 位序 | 数据元素 | 存储地址 |
|----------|----------|------------------|
| 1 | a_1 | b |
| 2 | a_2 | $b+l$ |
| \vdots | \vdots | \vdots |
| i | a_i | $b+(i-1)*l$ |
| \vdots | \vdots | \vdots |
| n | a_n | $b+(n-1)*l$ |
| | \vdots | $b+n*l$ |
| 空闲 | \vdots | \vdots |
| | \vdots | $b+(maxlen-1)*l$ |

线性表的顺序存储结构示意图

2. 顺序存储线性表的描述

❖ C语言中静态分配描述（补充）

```
#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

➤ 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

➤ 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

➤ 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+length-1;//表尾
    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： $i=\text{表长}+1$ ，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： $i=\text{表长}$ ，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

```
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

```
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++;
}
```

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; }
```


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

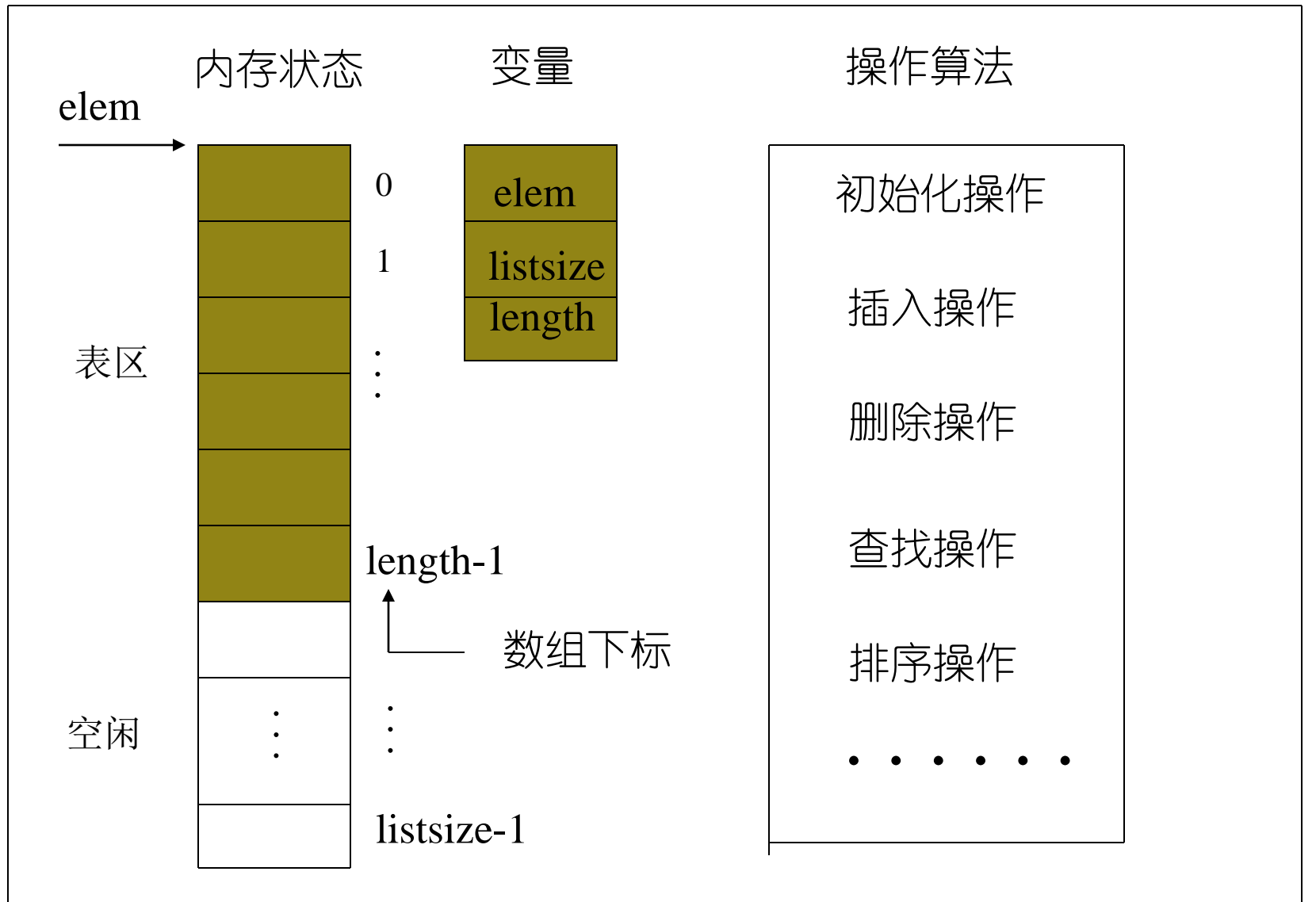
1) 优点

- 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

顺序表之整体概念：



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。

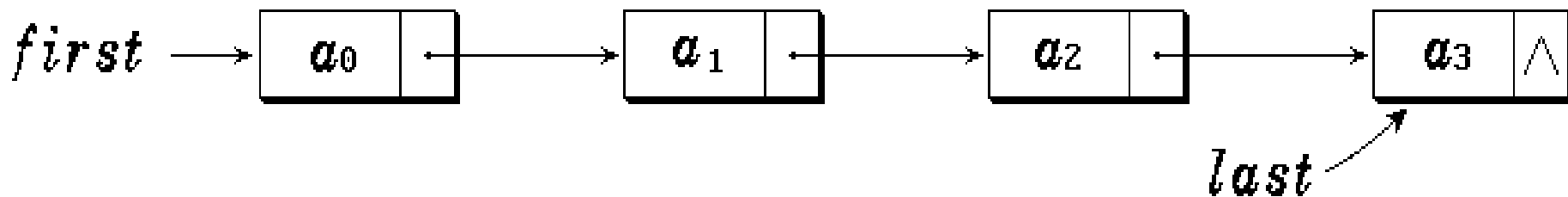
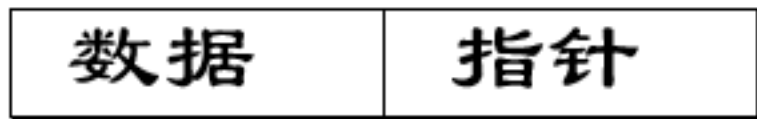
2.3 线性表的链式表示和实现

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）

| | |
|------|------|
| data | next |
|------|------|
- 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。

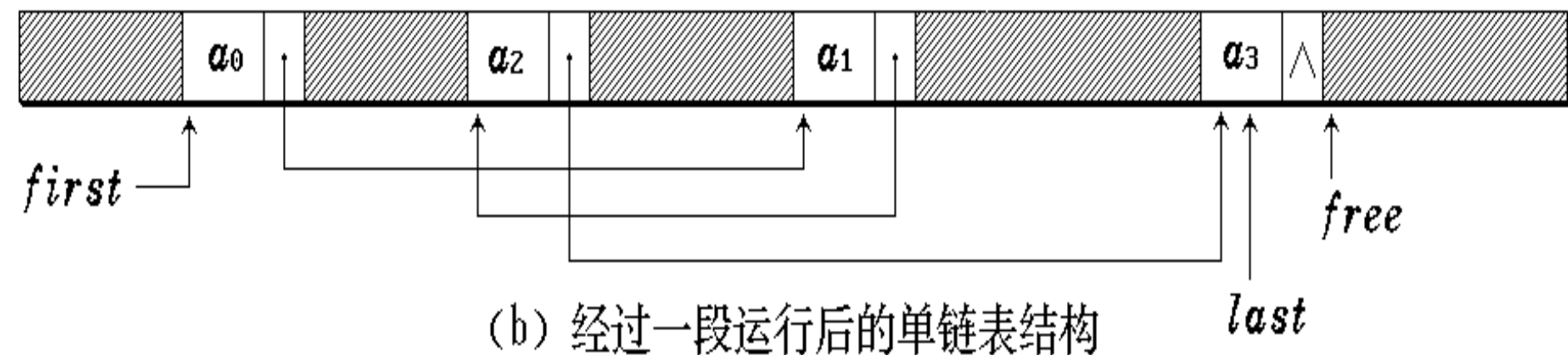
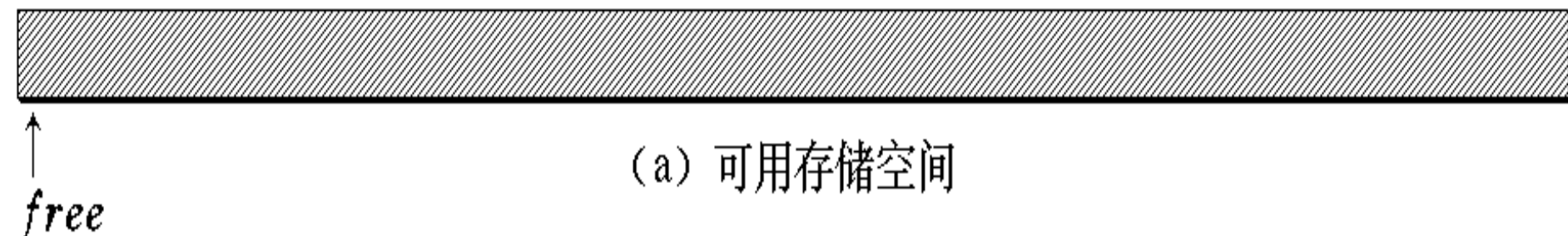
单链表 (Singly Linked List)



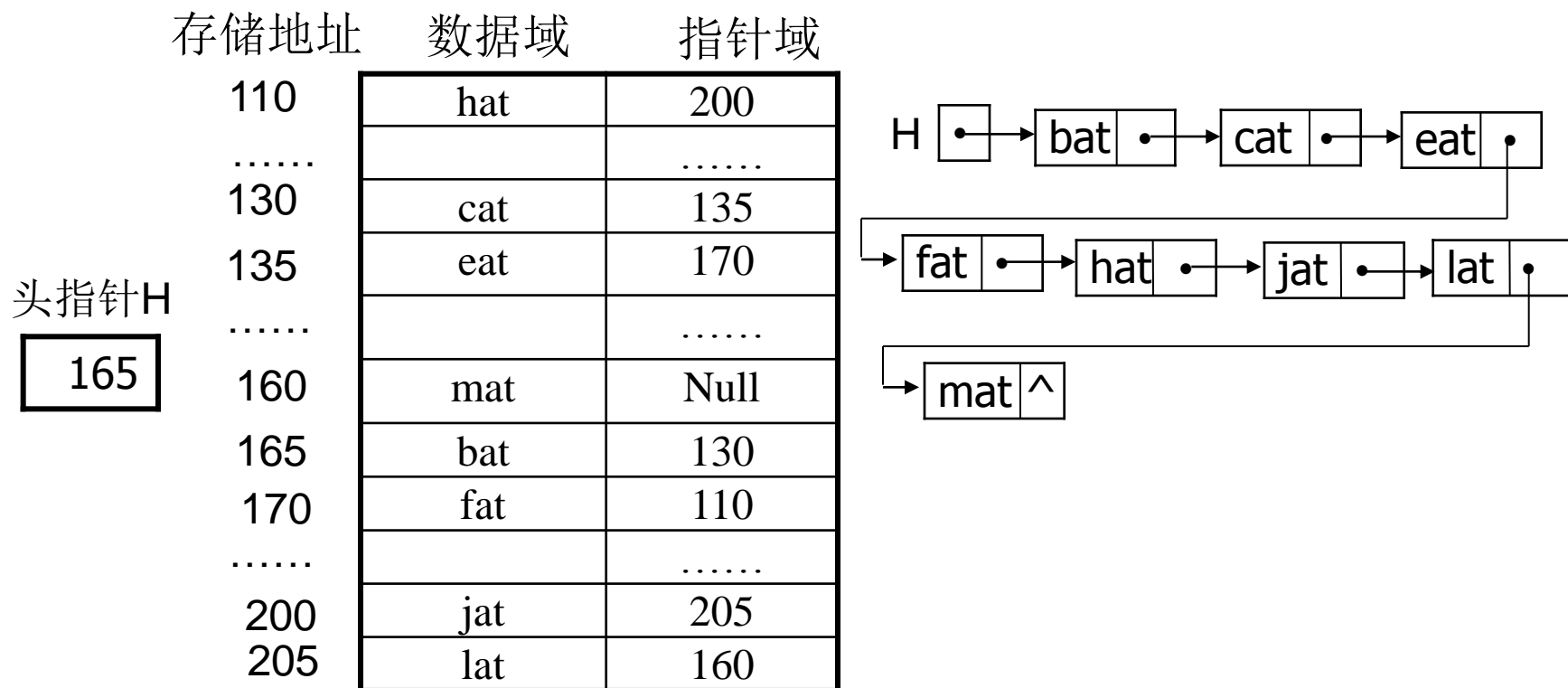
- first 头指针
- last 尾指针

- \wedge 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



例：线性表：(bat, cat, eat, fat, hat, jat, lat, mat)
的线性链表示意图如下：



1) 线性链表的描述 p28

```
typedef struct LNode{
```

```
    ElemType  data;
```

```
    Struct LNode  *next;
```

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

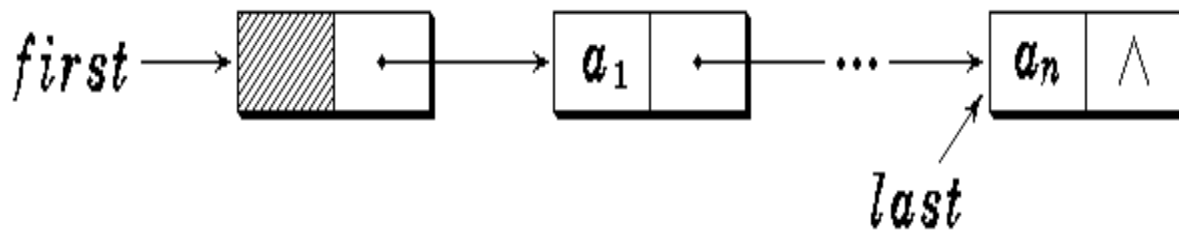
LinkList L; //L是LinkList类型的变量,
表示单链表的头指针

2) 术语

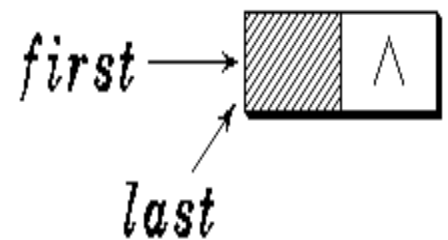
- **头指针**：指向链表中第一个结点
- 第一个数据元素结点（**开始结点**）
- **头结点**：有时在单链表的第一个数据元素结点之前附设一个结点，称之头结点。
 - 说明：头结点的next域指向链表中的第一个数据元素结点。
 - 对于头结点数据域的处理：
 - a. 加特殊信息
 - b. 置空
 - c. 如数据域为整型，则在该处存放链表长度信息。

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。



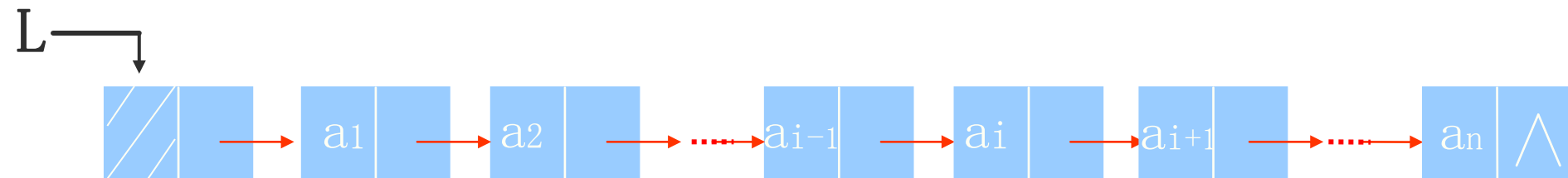
非空表



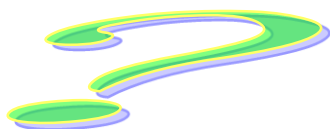
空表

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



如何在线性链表
上实现线性表的基本操作？
如何建表？如何插入？删除？



约定用带头结点的线性链表
存储线性表

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;
}
```


C语言中两个标准函数

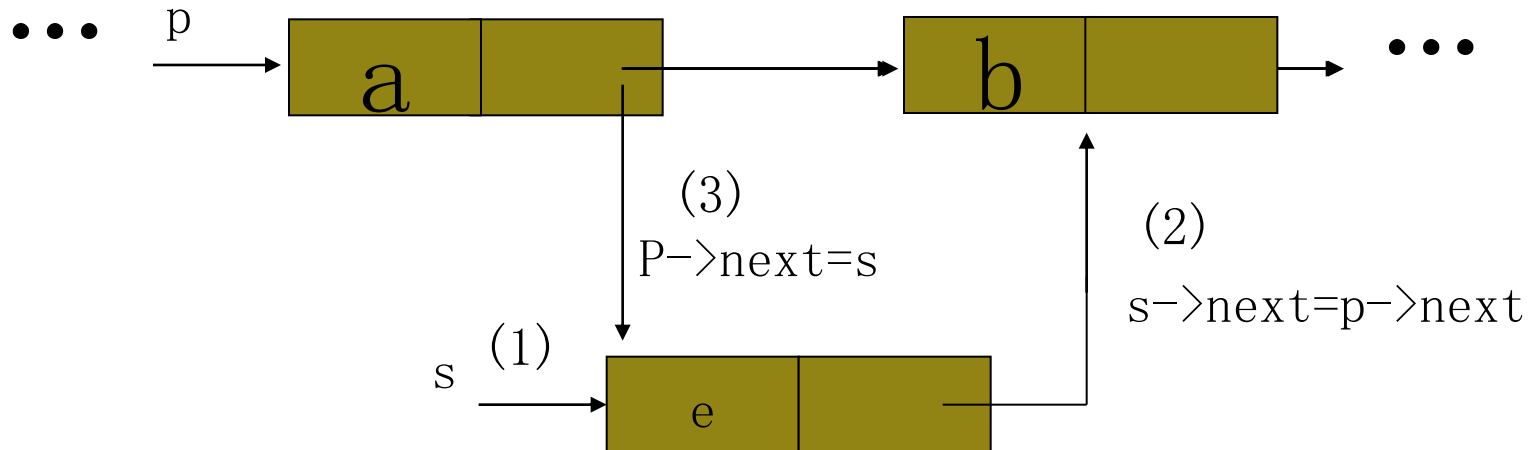
- * Malloc和free： 动态分配和回收
- * 假设p是linklist型的变量，`p = (LinkedList) malloc(sizeof (LNode))`的作用是由系统生成一个Lnode型的结点，同时将该结点的起始位置赋给指针变量p
- * 假设 q是linklist型的变量，`free(q)`的作用是由系统回收一个Lnode型的结点。

插入元素_{p30} 算法2.9

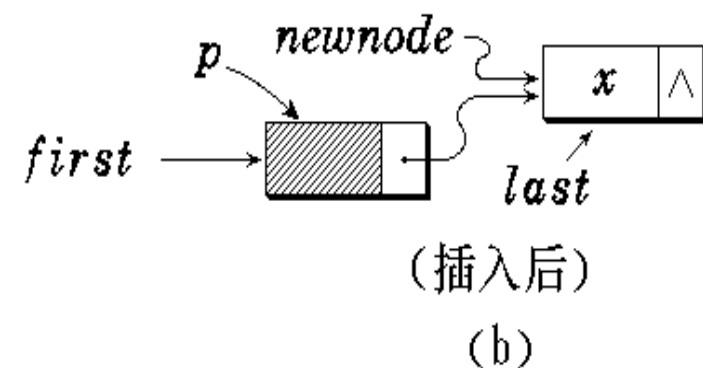
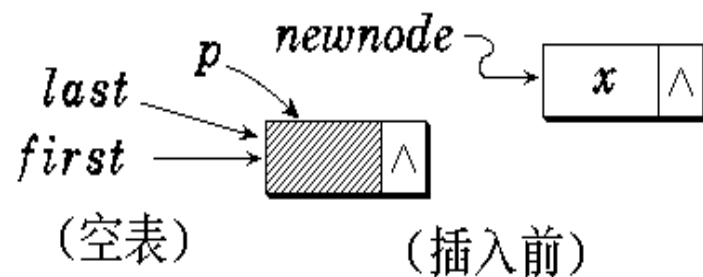
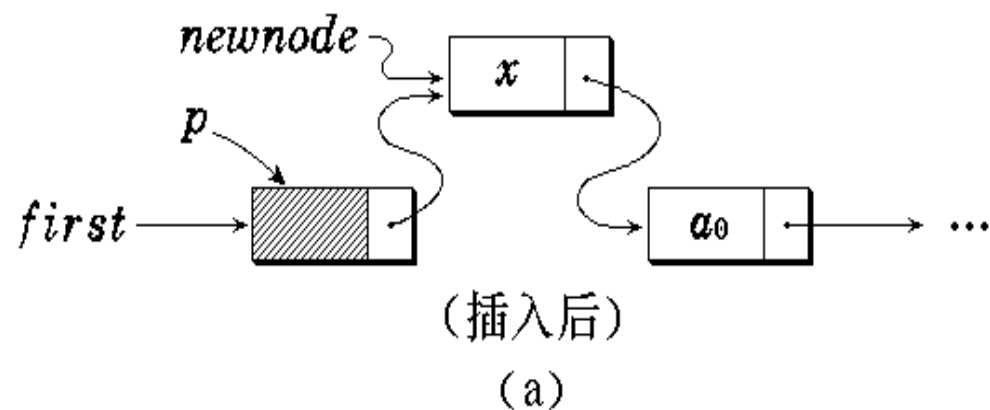
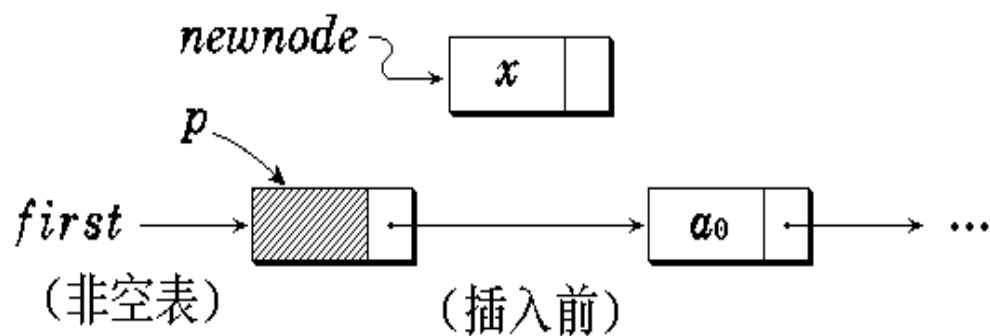
在第 i 个元素之前插入，先找到第 $i-1$ 个结点

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;}
```



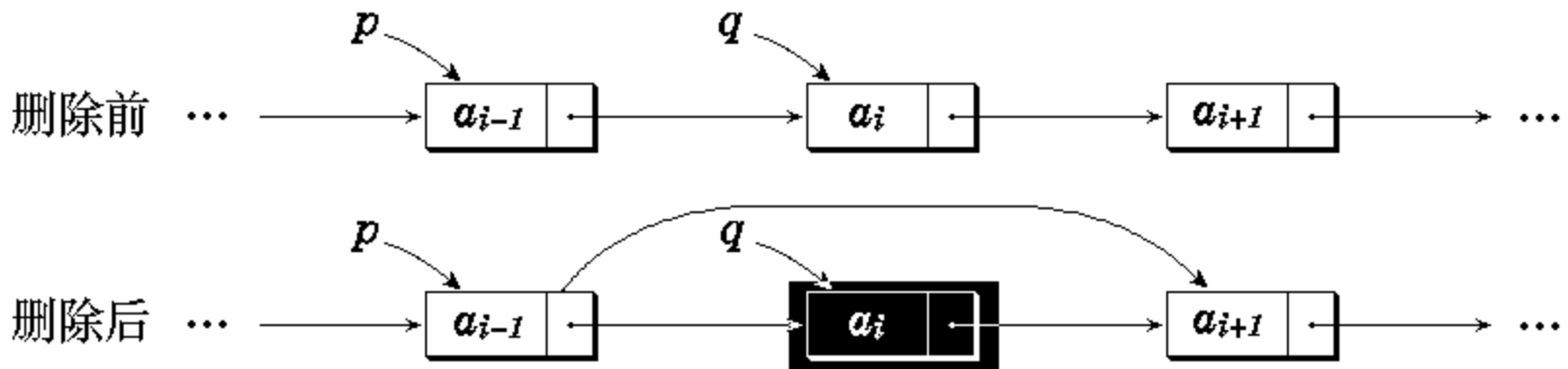
在带表头结点的单链表 第一个结点前插入新结点



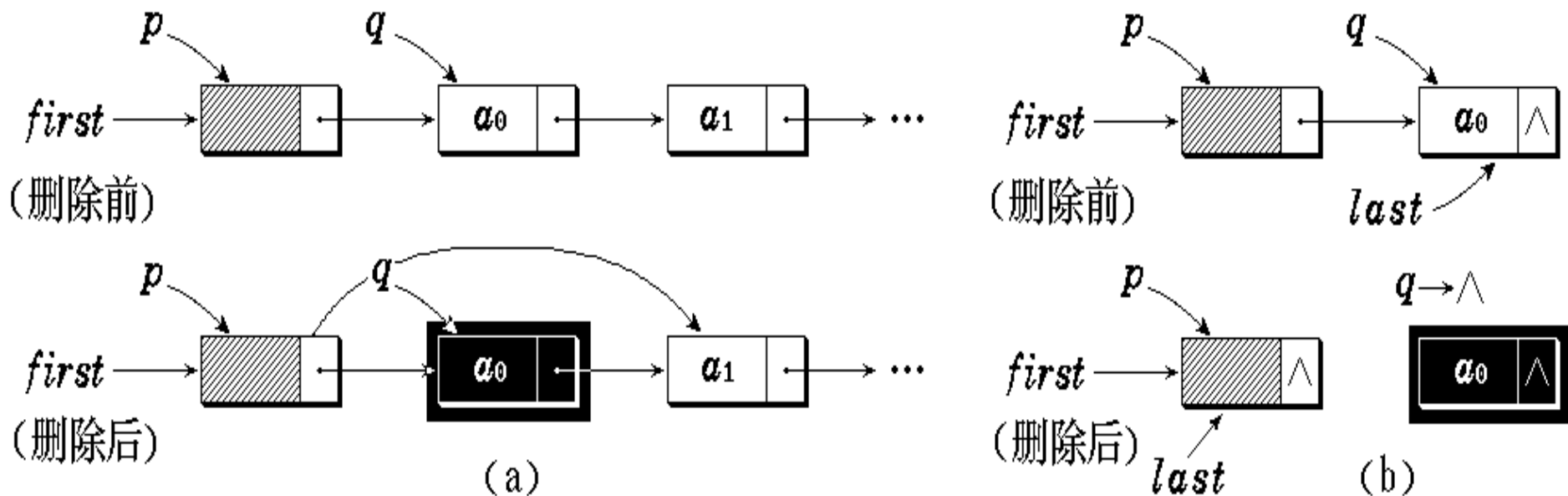
```
newnode→next = p→next;  
if ( p→next ==NULL ) last = newnode;  
p→next = newnode;
```

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; }
}
```

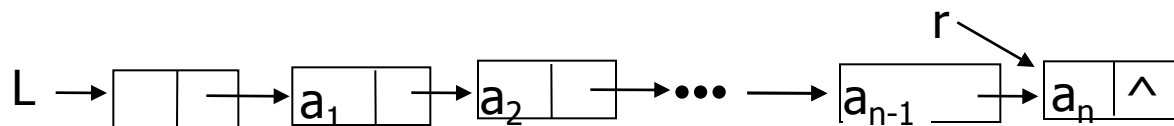
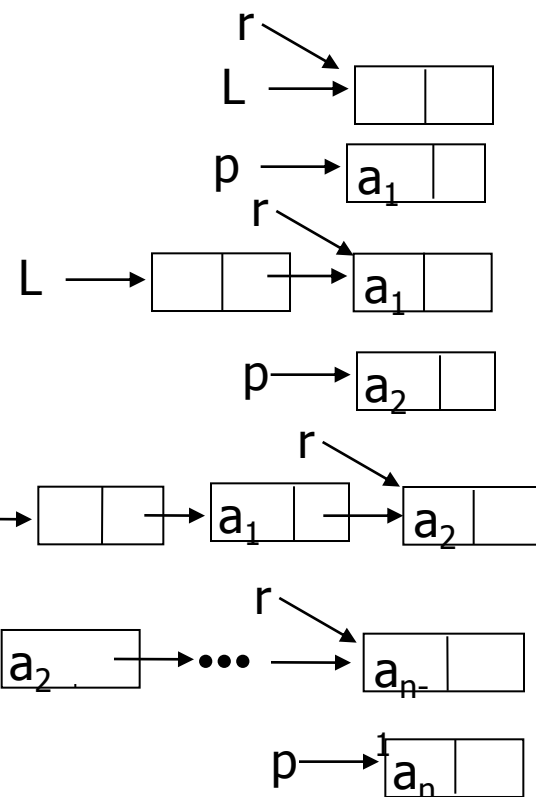
思考题： 建立链表（尾插法建表）

思考题：建立链表（尾插法建表）

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

思考题：线性链表的建立算法（尾插法）

```
L=(LinkedList)malloc(sizeof(LNode));  
r=L;  
for (i=1; i<=n; ++i) {  
    p=(LinkedList)malloc(sizeof(LNode));  
    scanf(&p->data);  
    r->next=p; r=p;  
}  
p->next=NULL  
}
```



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; //插入剩余段
    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);
}
```

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 $A=A \cup B$

```
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: $O(m*n)$;

线性链表小结

线性链表是线性表的一种链式存储结构

线性链表的特点

- 1 通过保存直接后继元素的存储位置来表示数据元素之间的逻辑关系；
- 2 插入、删除操作通过修改结点的指针实现；
- 3 不能随机存取元素。

3. 循环链表

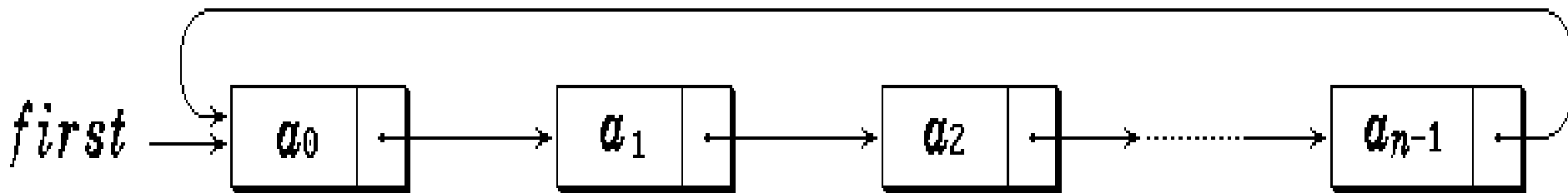
1) 循环链表——是一种首尾相接的链表。

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。

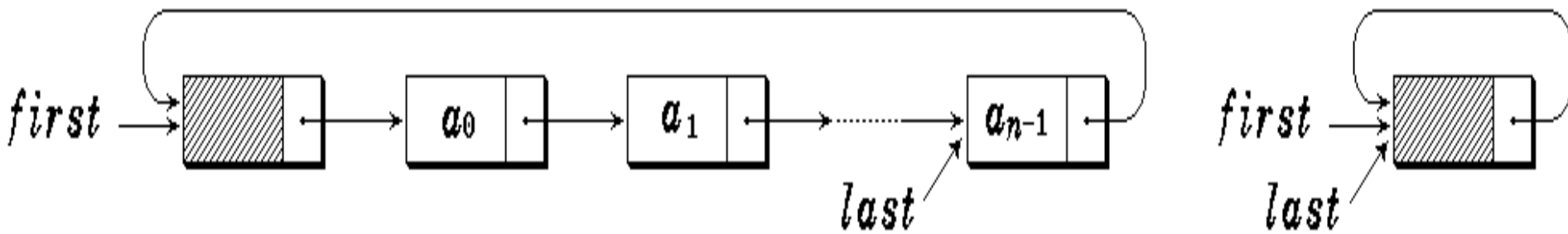
循环链表与单链表不同的是链表中表尾结点的指针域不是NULL，而是链表头结点的指针H（链表指针）

- 循环链表的示例



- 带表头结点的循环链表

（循环条件： $p \rightarrow \text{next} \neq H$ ）



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");
}
```

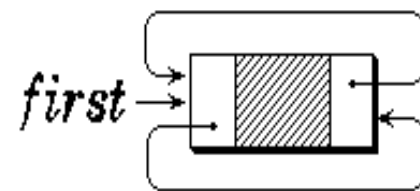
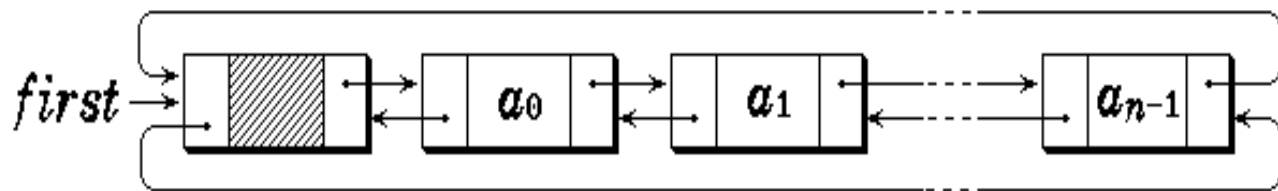
4. 双向链表 (Doubly Linked List)

- 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:

| | | |
|------------------------|---------------------|-----------------------|
| <i>prior</i> (左链指针) | <i>data</i> (数据) | <i>next</i> (右链指针) |
|------------------------|---------------------|-----------------------|

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：

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

- 置空表：

$$p \rightarrow \text{prior} = p ; p \rightarrow \text{next} = p ;$$

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;
```

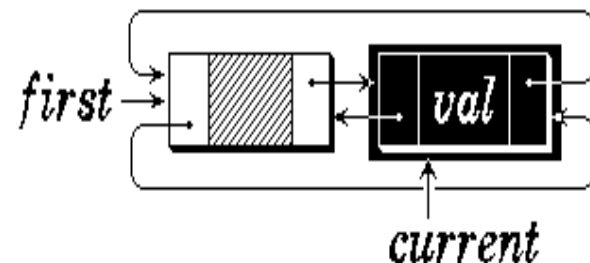
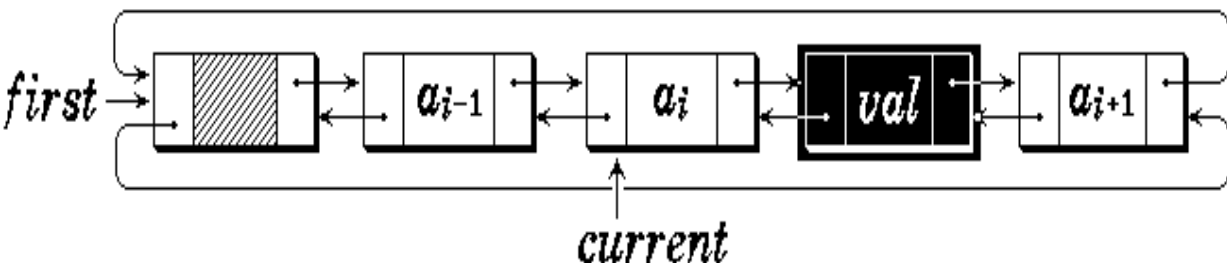
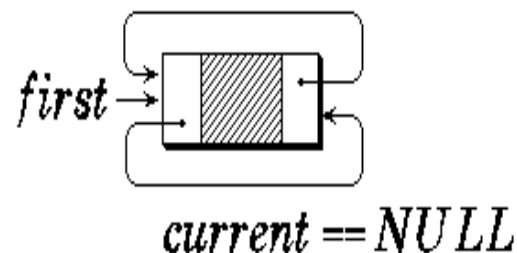
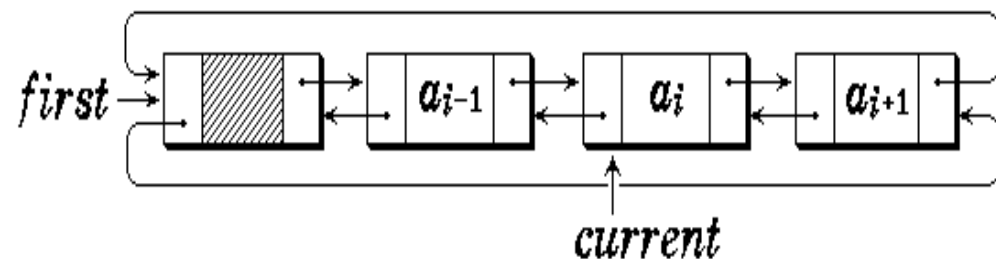

双向循环链表的插入算法

$p \rightarrow \text{prior} = \text{current};$ (1)

$p \rightarrow \text{next} = \text{current} \rightarrow \text{next};$ (2)

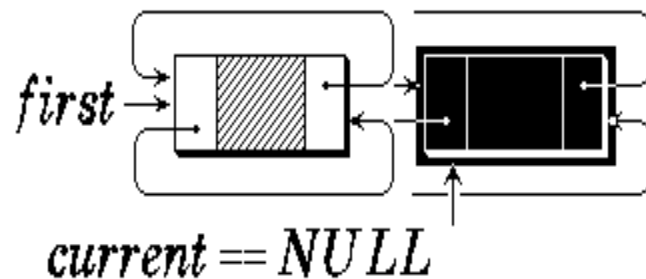
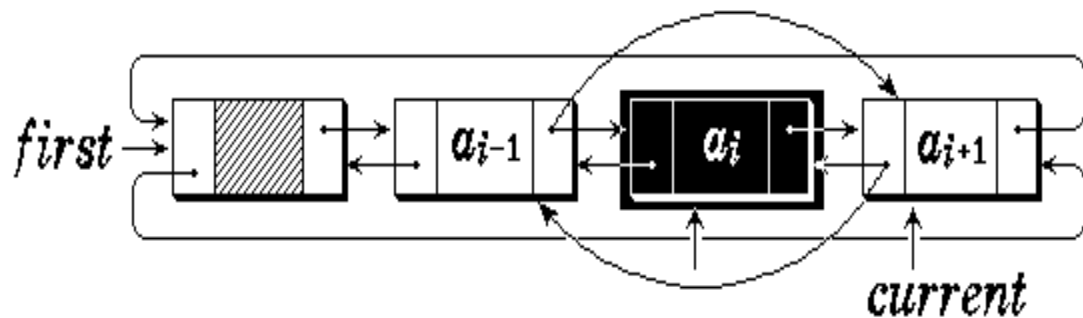
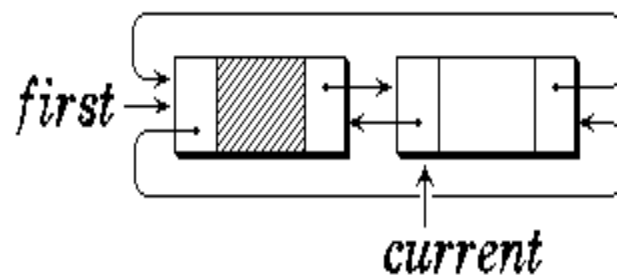
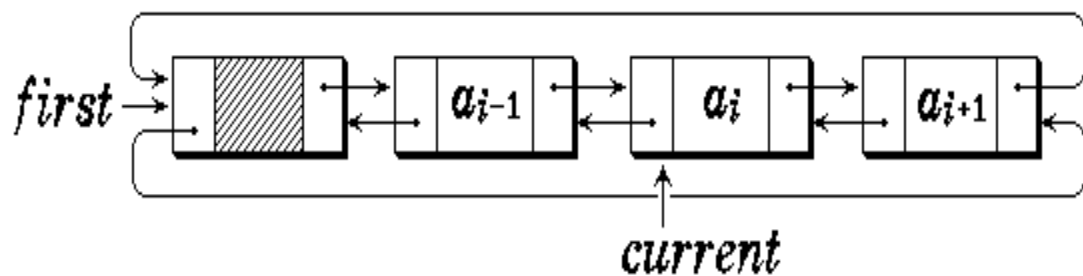
$\text{current} \rightarrow \text{next} = p;$ (3)

$p \rightarrow \text{next} \rightarrow \text{prior} = p;$ (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");
}
```

顺序表和链表的比较

■ 从存储空间利用率角度出发

顺序表：存储空间是**静态分配**的，在执行程序前需预先分配一定长度的存储空间。当线性表的长度 n 不能预先确定时，可能会分配的过大或过小，则导致存储空间的浪费或溢出。

链表：存储空间是**动态分配**的，无需预先分配存储空间，只要内存中尚有可分配空间就不会发生溢出。但指针域需占额外的存储空间。

∴

当线性表**长度变化较大**或难以估计时，宜采用链表存储结构；

当线性表**长度变化不大**，且能预先估计存储量大小时，为节省存储空间，宜采用顺序表存储结构。

■从时间效率角度考虑

若对线性表进行的主要操作是查找，很少进行插入和删除操作时，宜采用顺序表存储结构。

若对线性表进行插入和删除操作频繁时，宜采用链表作为存储结构。

2.4 一元多项式的表示和相加

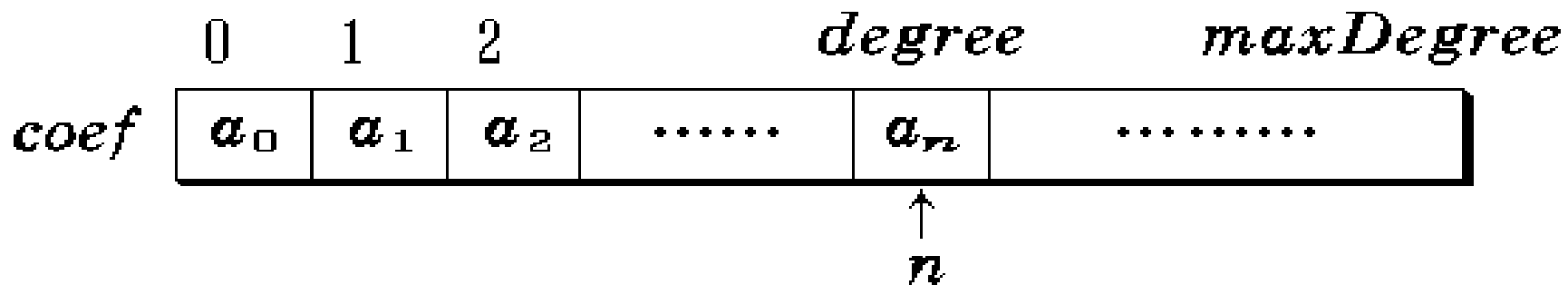
$$\begin{aligned} P_n(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \\ &= \sum_{i=0}^n a_i x^i \end{aligned}$$

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

1. 第一种表示方法

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

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：

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \cdots + p_mx^{e_m}$$

其中： p_i 是指数为 e_i 的项的非零系数，

$0 \leq e_1 \leq e_2 \leq \cdots \leq e_m \leq n$ 二元组表示

$((p_1, e_1), (p_2, e_2), \cdots, (p_m, e_m))$

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

表示成： $((7, 3), (-2, 12), (-8, 999))$

| | 0 | 1 | 2 | | i | | m |
|-------------|-------|-------|-------|-------|-------|-------|-------|
| <i>coef</i> | a_0 | a_1 | a_2 | | a_i | | a_m |
| <i>exp</i> | 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: $D = \{a_i \mid a_i \in \text{TermSet}, i=1, 2, \dots, m, m \geq 0\}$ 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: $R1 = \{\langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, \text{且 } a_{i-1} \text{中的指数值} < a_i \text{中的指数值}, i=2, \dots, n\}$

基本操作:

} ADT *Polynomial*

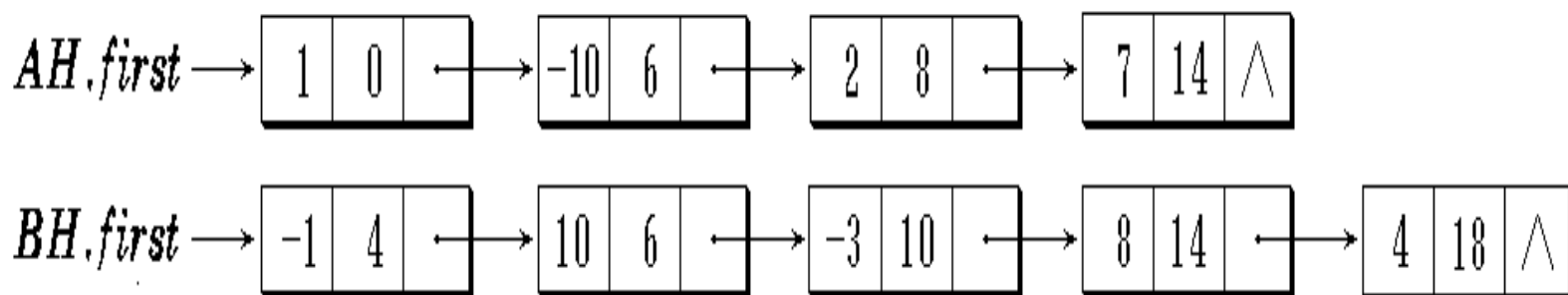
4. 抽象数据类型 (Polynomial) 的实现

```
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;
```

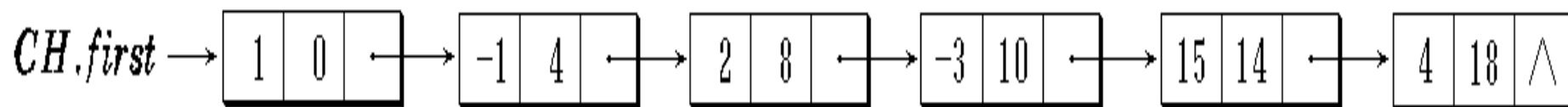
多项式链表的相加

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

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



(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。



本章小结

■ 线性表的逻辑结构

线性表的逻辑结构特征

线性表上定义的基本运算，并能利用基本运算构造出较复杂的运算

■ 线性表的顺序存储结构

顺序表的含义及特点

顺序表上的插入、删除操作及其平均时间复杂度分析
利用顺序表设计算法解决简单的应用问题

■ 线性表的链式存储结构

- ◆ 链表中头指针和头结点的使用
- ◆ 单链表、双链表、循环链表链接方式上的区别
- ◆ 单链表上实现的查找（按序号、按值）、插入和删除（注意修改指针的顺序）等基本算法，并分析其时间复杂度（都为 $O(n)$ ）。
- ◆ 循环链表上尾指针取代头指针的作用，以及单循环链表上的算法与单链表上相应算法的异同点。
- ◆ 双链表的定义特点，求前驱较方便
- ◆ 利用链表设计算法解决简单的应用问题：如两个链表的合并等

■顺序表和链表的比较

◆顺序表和链表的主要优缺点（从空间和时间这两方面来考虑，其基本操作实现的难易程度）

◆顺序表：随机存取结构，适宜于静态查找

◆链表：适宜于动态的插入、删除操作

◆针对线性表上所需要执行的主要操作，知道选择顺序表还是链表作为其存储结构才能取得较优的时空性能。