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:

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

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。

数据结构描述如下：

List = (D,R)

其中：D={a_i | a_i∈D0, i=1,2,…n, n≥0}

R = {N},

N ={<a_{i-1}, a_i>| a_{i-1}, a_i∈D0, i=2,3,…n}

D0为某个数据对象（数据的子集）

3）线性表的特性：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

4）线性表的属性：

长度：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。

线性表的长度可根据需要增长或缩短，数据元素不仅可以访问，还可以增加、删除。

2.1 线性表的类型定义

2. 抽象数据类型的定义

- ADT List{

数据对象： D={a_i | a_i∈D0, i=1,2,…n, n≥0}

数据关系： R={<a_{i-1}, a_i> | a_{i-1}, a_i∈D, i=2,3,…n}

基本操作：

1）InitList(&L)初始化,构造一个空的线性表L。

2）ListLength(L) 求长度,返回线性表中元素个数。

3）GetElem(L, i, &e) 取L中第i个数据元素值赋给e。

4）LocateElem(L, e, compare()) 按值查找，返回L中第1个与e满足关系compare()的数据元素的位序，若不存在，返回值为0。

5）ListInsert(&L, i, e) 在L中第i个位置前插入新的数据元素e，L的长度加1。

6）ListDelete(&L, i, &e) 删除L的第i个数据元素，用e返回其值，表长减1。

7）DestroyList(&L) 销毁线性表L。

8）ClearList(&L) 将L重置为空表。

9）ListEmpty(L) 判断L是否为空表，是返回true,否则返回false。

10）PriorElem(L, cur_e, &pre_e) 若cur_e是L的数据元素，且不是第1个，用pre_e返回cur_e的前驱，否则操作失败，pre_e无定义。

11）NextElem(L, cur_e, &next_e) 若cur_e是L的数据元素，且不是最后一个，用next_e返回cur_e的后继，否则操作失败，next_e无定义。

12）ListTraverse(L, visit()) 遍历，依次对L的每一个元素调用函数visit()。

}ADT List

ADT操作举例

操作	输出	线性表组成
InitList(L)		初始化，空表
ListInsert(L,1,8)		8
ListInsert(L,1,4)		4 8
ListInsert(L,2,7)		4 7 8
GetElem(L,3)	8	4 7 8
ListDelete(L,1)		7 8
ListInsert(L,2,9)		7 9 8
LocateElem(L,9)	2	7 9 8
ListInsert(L,4,6)		7 9 8 6
ListInsert(L,5,2)		7 9 8 6 2
ListInsert(L,6,4)		7 9 8 6 2 4
LocateElem(L,5)	0	7 9 8 6 2 4

说明:

a)上面列出的操作是线性表的一些常用的基本操作;

b)不同的应用，基本操作集可能是不同的;

c)线性表的复杂操作可通过基本操作实现。

例如：将两个线性表合并；把一个线性表拆成两个或两个以上的线性表；复制一个线性表等。

2.1 线性表的类型定义

3. 线性表的操作举例

例2-1 求集合的并运算 A = A ∪ B

分析：用线性表LA和LB分别表示集合A和B。

举例：LA=(3, 5, 8, 11); LB=(2, 6, 8, 9, 11, 15, 20)，则合并后 LA=(3, 5, 8, 11, 2, 6, 9, 15, 20)

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，则插入到LA中。

总的时间复杂度：O(ListLength(A)*ListLength(B))

```
void union(List &La，List Lb) {
    la_len=ListLength(La);
    lb_len=ListLength(Lb);
    for(i=1; i<=lb_len; i++)      { //O(m)
      GetElem(lb，i，e); //O(1)
      if (!LocateElem(La, e, equal)) // O(n)
           ListInsert(La，++la_en，e); // O(1)
    }
}//算法2.1   O(m*n)
```

- 例2-2 合并有序表LA 和 LB 到LC中

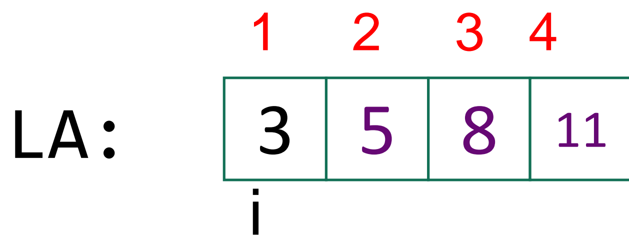
已知：LA、LB：非递减有序排列

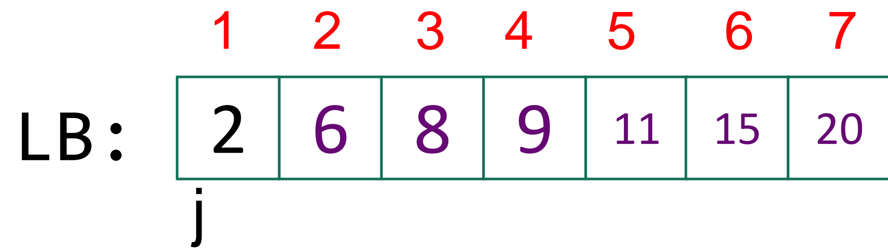
要求：LC：非递减有序排列

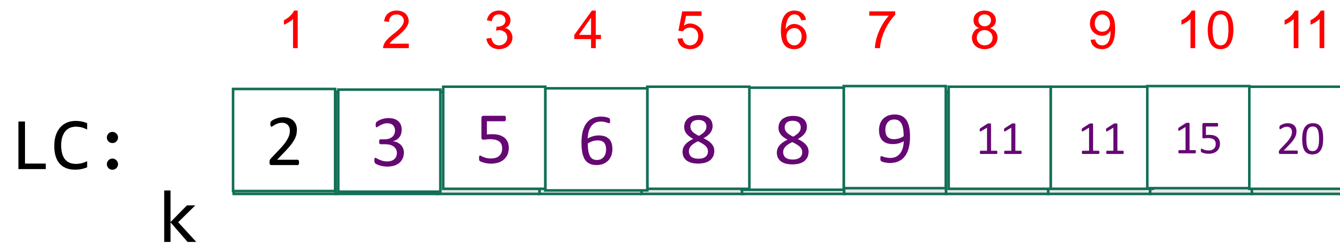
分析：先设LC为空表，然后将LA或LB中的元素逐个插入到LC中即可。

- 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：O(ListLength(LA)+ListLength(LB))


1 2 3 4
LA:
3 5 8 11
i


1 2 3 4 5 6 7
LB:
2 6 8 9 11 15 20
j


1 2 3 4 5 6 7 8 9 10 11
LC:
2 3 5 6 8 8 9 11 11 15 20
k

```
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);        }
}// 算法2.2
```

时间复杂度：ListInsert( )的执行次数
O(ListLength(LA)+ListLength(LB))

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

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。

2.2 线性表的顺序表示和实现

3）线性表的顺序存储结构示意图

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	$b+l$
⋮	⋮	⋮
i	a_i	$b+(i-1)*l$
⋮	⋮	⋮
n	a_n	$b+(n-1)*l$
	⋮	$b+n*l$
空闲	⋮	⋮
	⋮	$b+(maxlen-1)*l$

2. 顺序表的类型定义

- 1) 用C语言中的数组描述 (静态)

```
#define LIST_MAX_LENGTH   100
typedef  struct {
    ElemType   elem[LIST_MAX_LENGTH];
    int   length;//
    } SqList;
SqList  L;
```

2. 顺序表的类型定义

- 2) C语言中动态分配描述（动态）

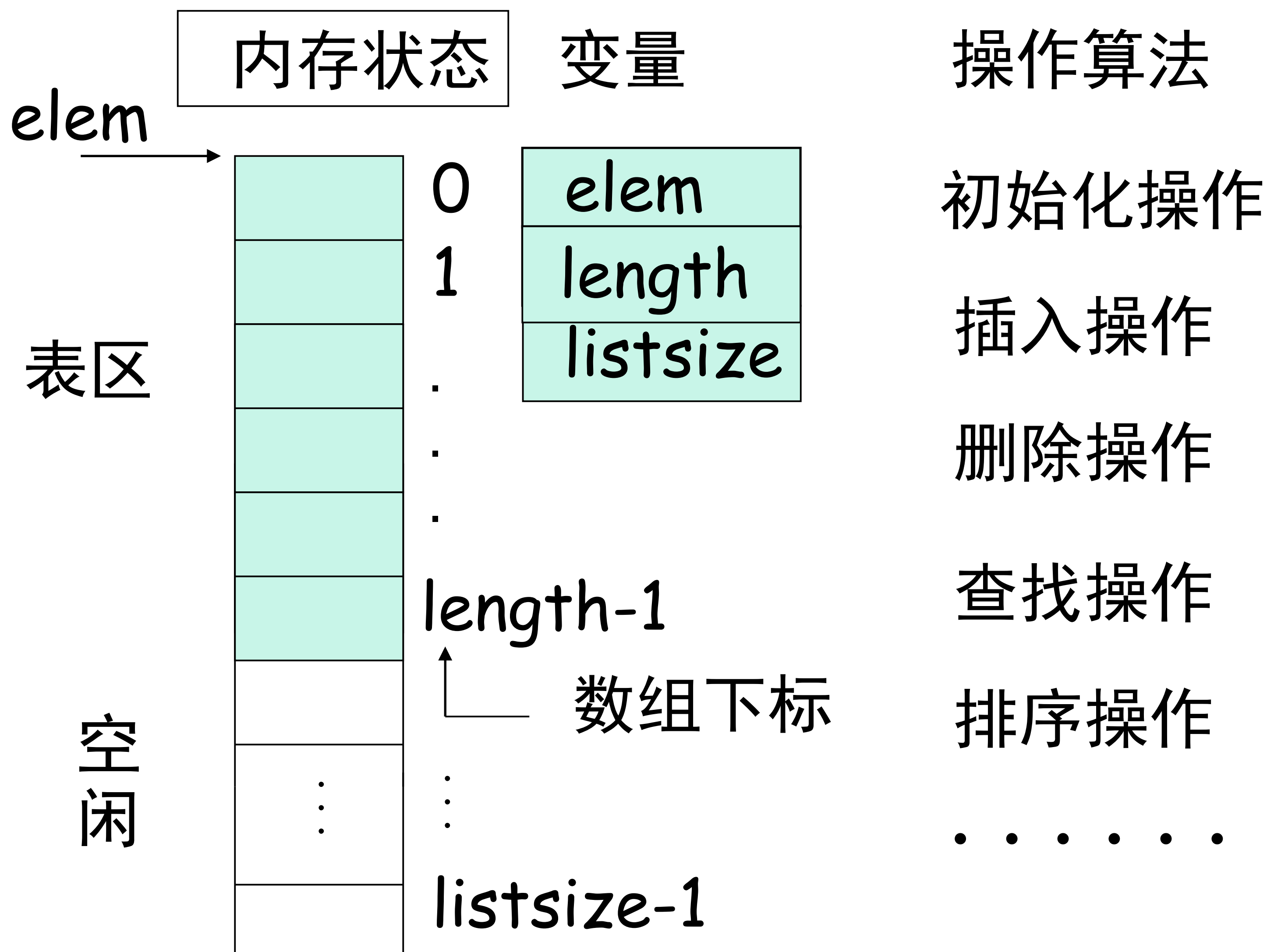
```
typedef  struct {
    ElemType   *elem;
    int        length;
    int        listsize;
    }SqList;
SqList  L;
```

其中：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

顺序表之整体概念



3. 顺序表的基本操作实现

① 初始化：Status InitList_SqList(SqList &L)

② 插入：Status ListInsert_sq(SqList &L, int i, ElemType e)

③ 删除：Status ListDelete_sq(SqList &L, int i, ElemType &e)

④ 取元素：int GetElem(SqList L, int i, ElemType &e)

⑤ 查找：int LocateElem_Sq(SqList L, ElemType e, Status (*compare)(ElemType,ElemType))

⑥ 合并: void MergeList_Sq(SqList la, SqList lb, SqList &lc)

常量及类型定义

```
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2 //在math.h中已定义
typedef int Status; /* Status是函数的类型,其值是函数结果状态代码，如OK等*/
typedef int Boolean; /* Boolean是布尔类型,其值是TRUE或FALSE */
```

C语言的头文件

```
#include<string.h>
#include<ctype.h>
#include<malloc.h> /* malloc()等*/
#include<limits.h> /* INT_MAX等*/
#include<stdio.h> /* EOF(=^Z或F6),NULL */
#include<stdlib.h> /* atoi() */
#include<io.h> /* eof()， */
#include<math.h> /* floor(),ceil(),abs() */
#include<process.h> /* exit()， */
```

仅供参考，C++编译器中无<io.h>,<process.h>，exit()在stdio 或stdlib中。

初始化

```
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;
}
```

动态分配内存函数

（1）C语言：malloc和free：动态分配和回收

格式： void *malloc(unsigned size)

功能：分配一块size字节大小的连续内存，返回指向这块内存的起始地址。若分配失败，返回NULL。

格式：void free(void *)

功能：系统回收用malloc分配的内存。

（2）C++：new 和delete

插入

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

函数realloc的格式及功能

- 格式：

  void *realloc(void *p, unsigned size)

- 功能：

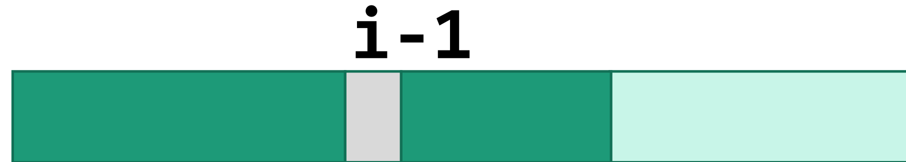
  调整之前调用 **malloc** 或 **calloc** 所分配的 **p**所指向的内存块的大小为size，并返回所分配内存的首地址（void*类型）

  size可以比原来分配的空间大或小。

  例如：L.elem=realloc(L.elem, L.listsize+100);

插入第i个元素的过程

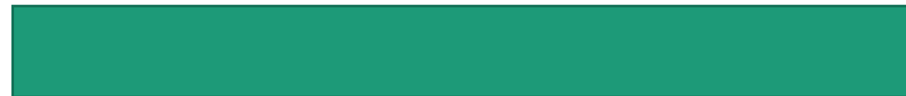
（1）表未满



将下标[i-1,length-1]元素右移

L.elem[i-1]=e;

(2)表已满，先扩充容量

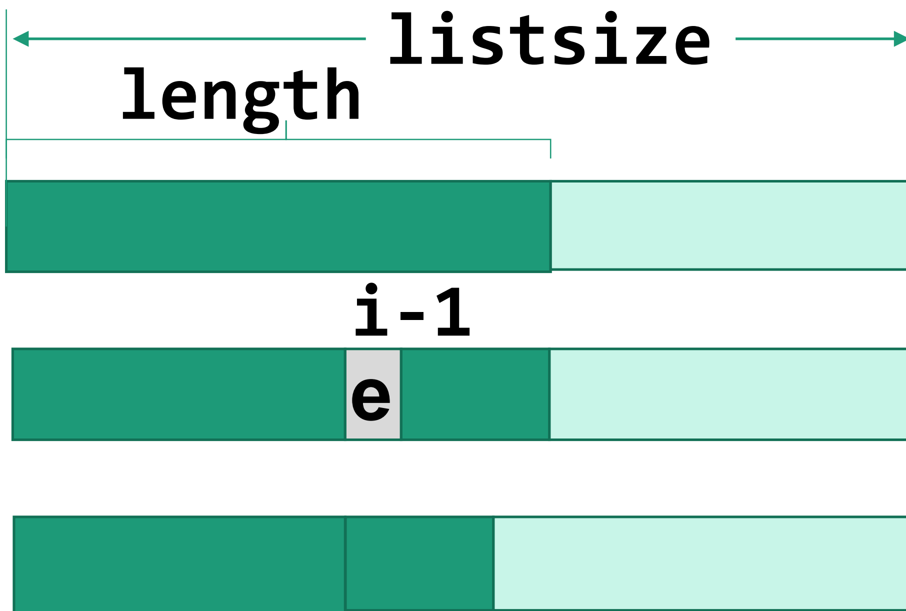


删除

```
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

删除过程



e=L.elem[i-1]

将下标[i,length-1]元素左移

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

取元素

```
int GetElem(SqList L, int i, ElemType &e){
        if (i>L.length | |i<=0) return false;
        e=L.elem[i-1]; return true;
}
```

- 修改元素就可以写成：

```
int PutElem(SqList L, int i, ElemType e){
        if (i>L.length | |i<=0) return false;
        L.elem[i-1] =e; return true;
}
```

查找

```
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;}
```

查找的更一般描述

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

合并(有序表的合并)

```
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++;}
```

例1：创建n个整数的顺序表

```
#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]);
   return OK;
}
```

遍历表

```
Status ListTraverse(SqList La,
Status(*visit)(ElemType)) {
        for (int i=0;i<La.length;i++){
                visit(La.elem[i]);  } return OK;
 }
Status print(ElemType e){ //假定ElemType是int类型
        printf("%d ",e);
        return OK;
}
调用：SqList L;    CreateList_Sq(L , 10);
        ListTraverse(L ,  print);
```

例2：递增插入(有序表)

```
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;
 }//在表中找到第一个不小于x的元素下标i
```

例3：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;
}//从右往左找到第1个不小于x的元素下标
```

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

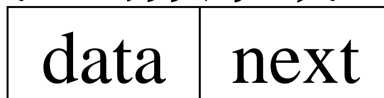
2.3 线性表的链式表示和实现

1. 线性链表

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**。

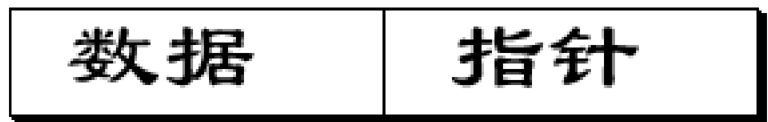
这两部分信息组成数据元素的存储映像，称作结点。

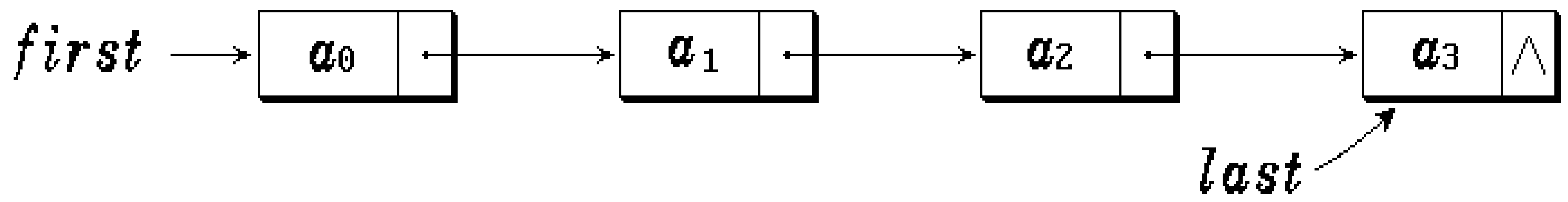
结点：数据域 + 指针域（链域）



链式存储结构：n个结点链接成一个链表

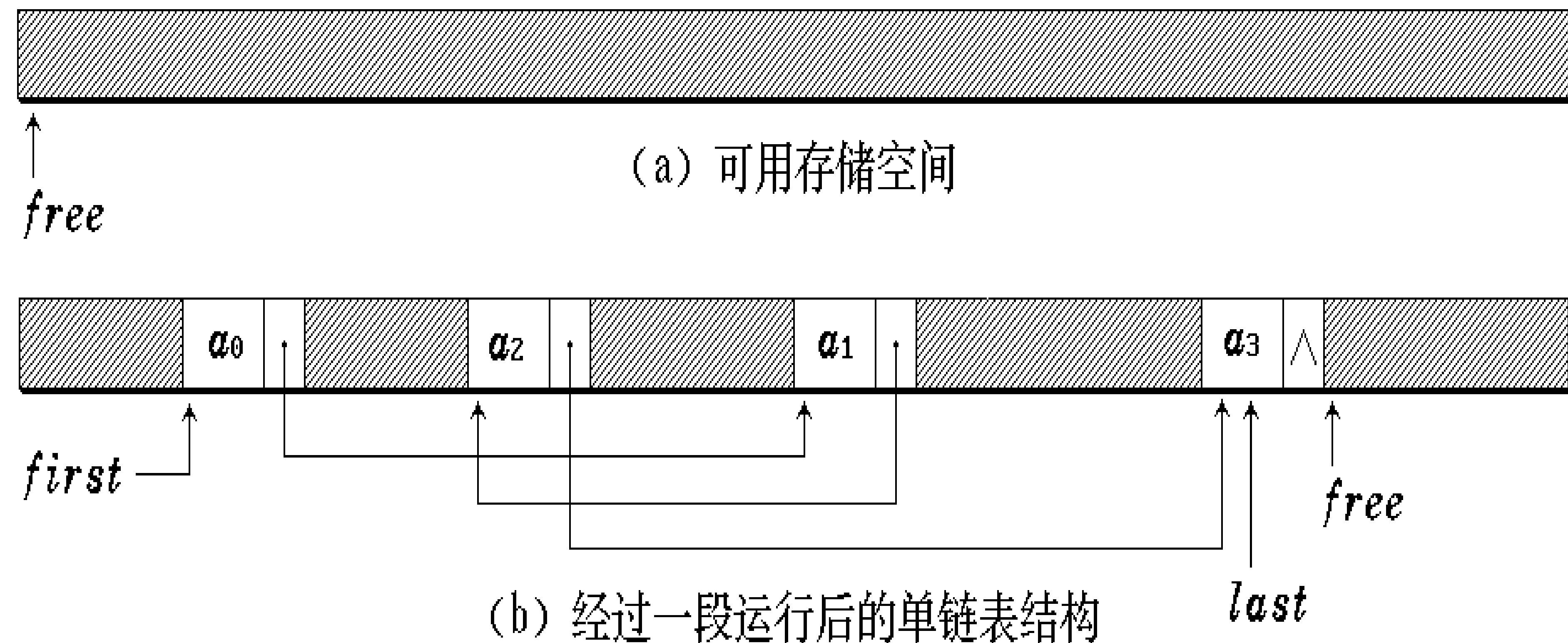
线性链表（单链表）：链表的每个结点只包含一个指针域。





- 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



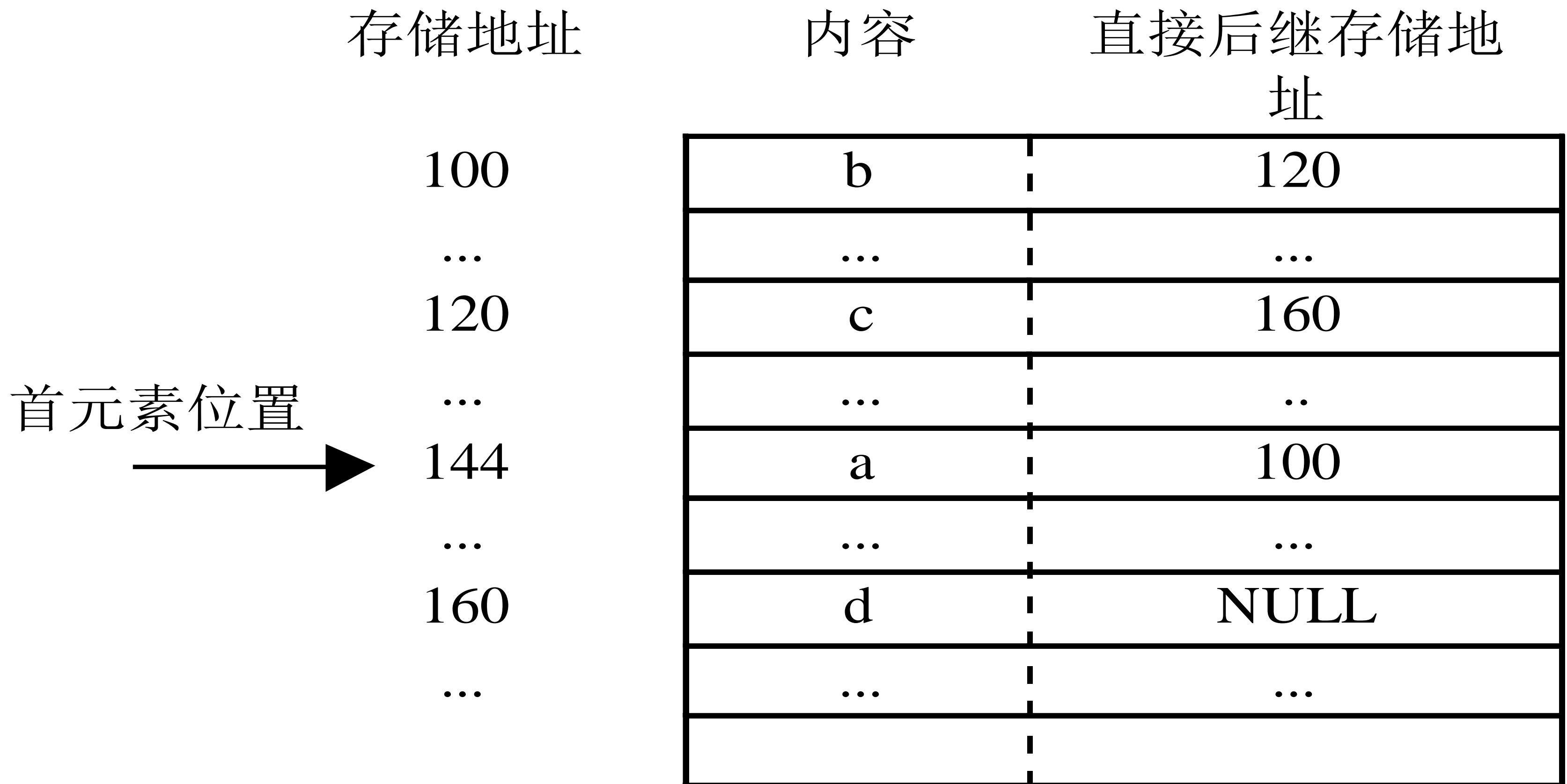


图2-2 线性表链式存储结构示意图

1）线性链表的类型定义

```
typedef struct LNode{
    ElemType  data;
    Struct LNode   *next;
} LNode, *LinkList;
LinkList  L;  //L是LinkList类型的变量
L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。
- 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：

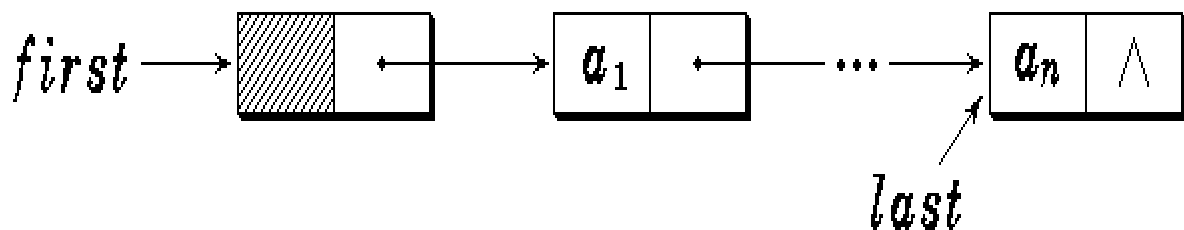
  a.加特殊信息

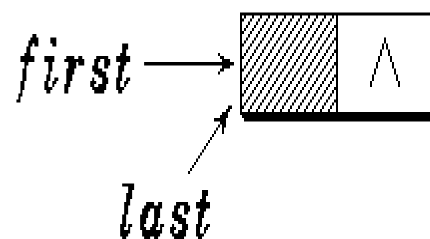
  b.置空

  c.如数据域为整型，则在该处存放链表长度信息。

带头结点的单链表示意图

- 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

3）基本操作

- 取元素：Status GetElem_L(LinkList L, int i, ElemType &e)
- 插入元素：Status ListInsert_L(LinkList &L, int i, ElemType e)
- 删除元素：Status ListDelete_L(LinkList &L, int i, ElemType &e)
- 建立链表：void CreateList_L(LinkList &L, int n)
- 有序链表的合并：void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc)
- 查找（按值查找）：int LocateElem_L(LinkList L, ElemType e, Status(*compare)(ElemType,ElemType))

取元素（按序号查找）

- 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;
}
```

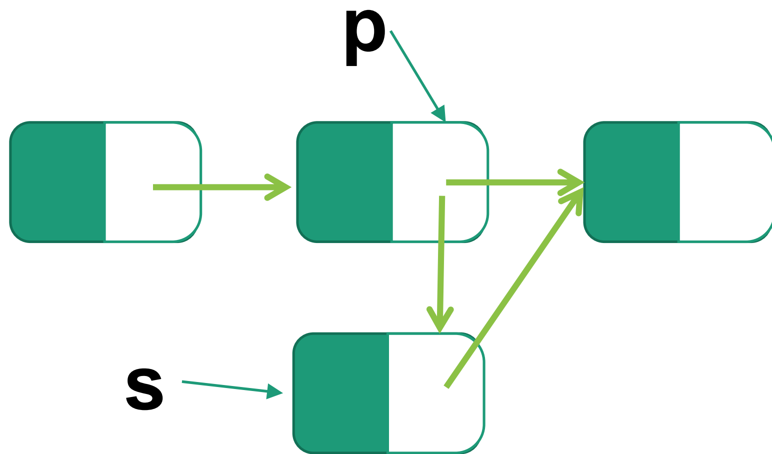
C语言中两个标准函数

- malloc和free：动态分配和回收

假设p是linklist型的变量，
p = (LinkList) malloc(sizeof (LNode))的作用是根据LNode类型的字节数分配内存，返回分配内存的起始地址，赋值给指针变量p

假设 q是linklist型的变量，free(q)的作用是由系统回收一个LNode型的结点内存。

插入结点



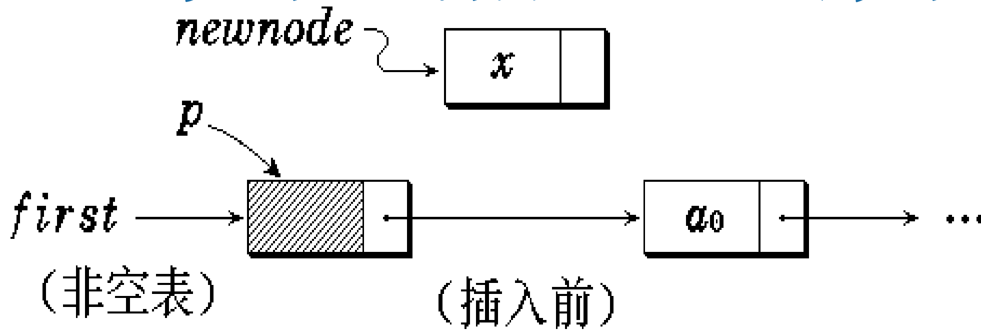

```
s=(LNode*)malloc(sizeof(LNode));
s->next=p->next;
p->next=s;
```

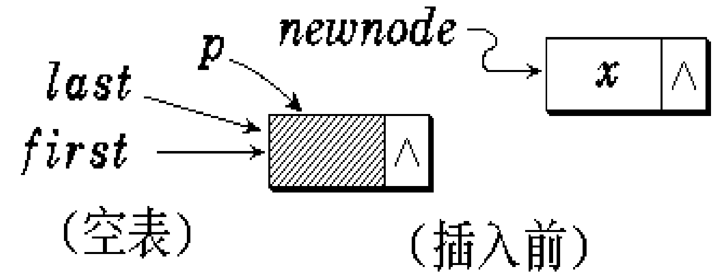
插入元素

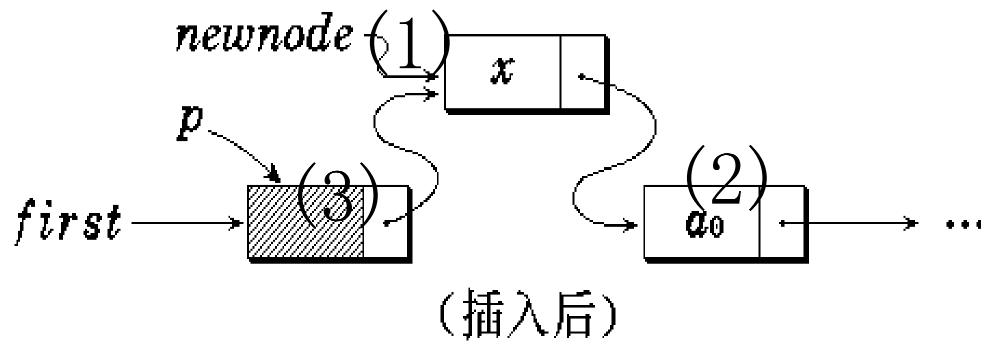
- 在第i个元素之前插入，先找到第i-1个结点

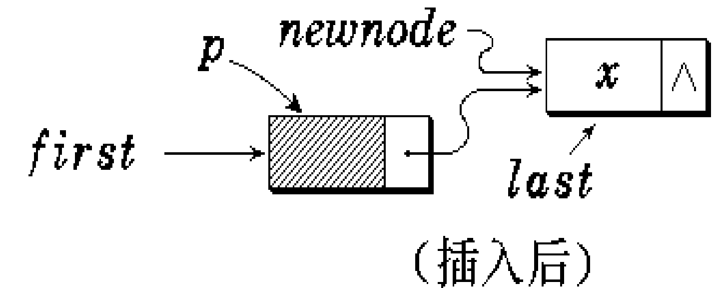
```
Status ListInsert_L(LinkList &L, int i, ElemType e)
{ LinkList p=L,s; int j=0;
        while (p && j<i-1) { p=p->next; ++j;}
        if (!p  | |  j> i-1) return ERROR;
        s = (LinkList) malloc( sizeof (LNode));
        s->data = e; s->next = p->next;
        p->next = s;
        return OK;
}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

在带表头结点（哨兵）的单链表的第一个结点前插入新结点









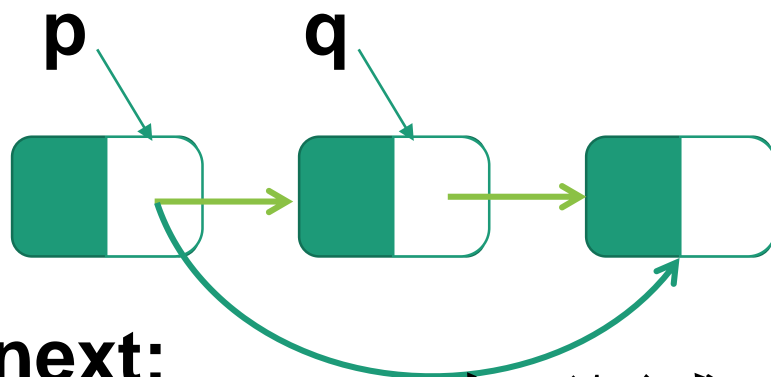
(1) ewnode=(LinkList)malloc(sizeof(LNode));

(2) newnode→next = p→next;

if (p→next ==NULL) last = newnode;

(3) p→next = newnode;

删除结点



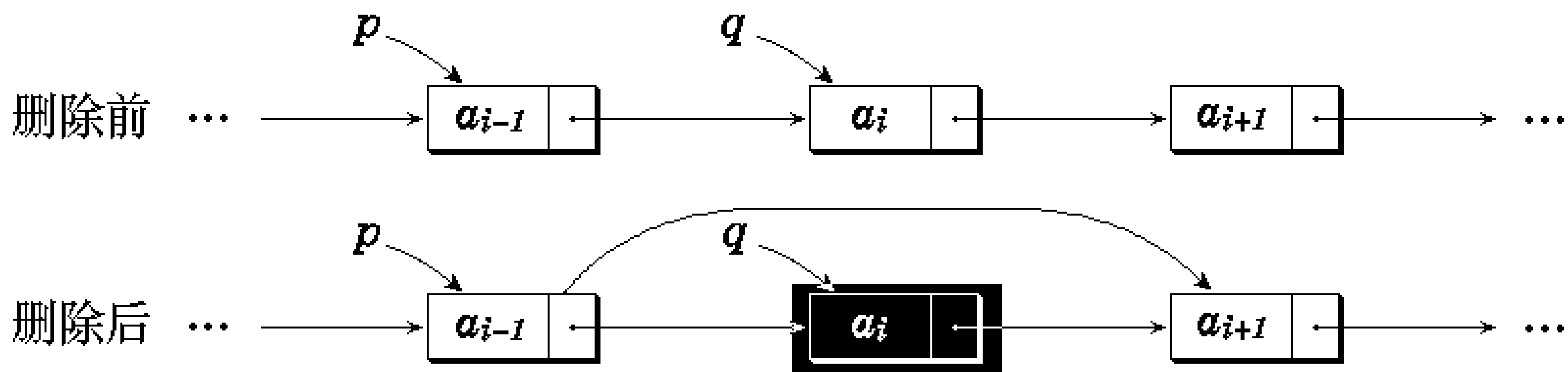
(1) q=p->next;

(2) p->next=q->next;

也可写成：

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

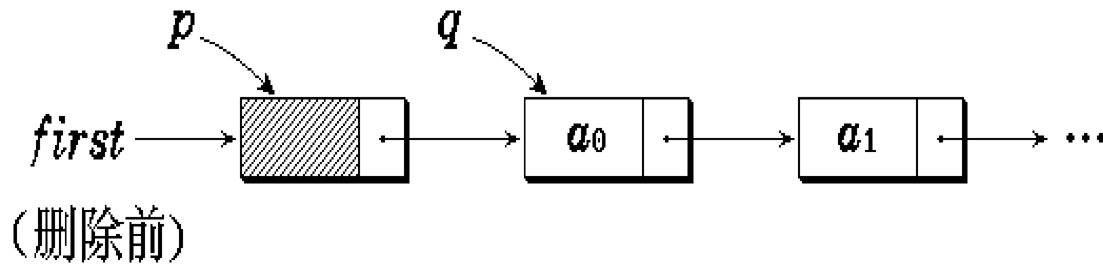
(3) free(q);

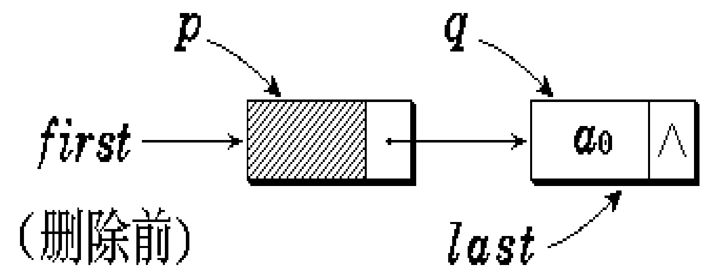


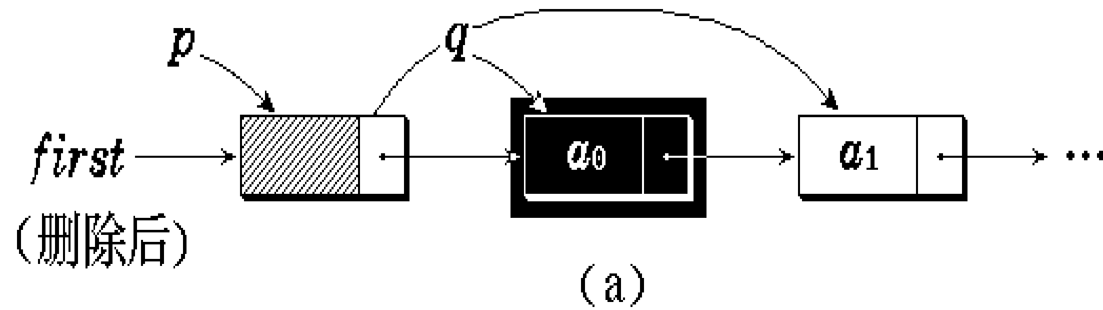
删除元素

```
Status ListDelete_L(LinkList &L, int i, ElemType &e)
{  LinkList p=L,q; 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;
}//算法2.10
```

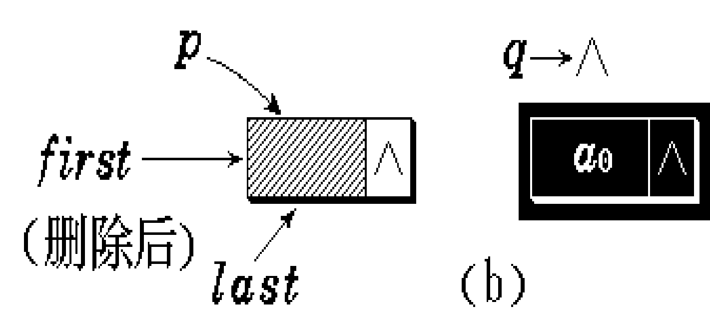
从带表头结点的单链表中删除第一个结点







(a)



(b)

(1) q = p→next;

(2) p→next = q→next;

(3) free(q);

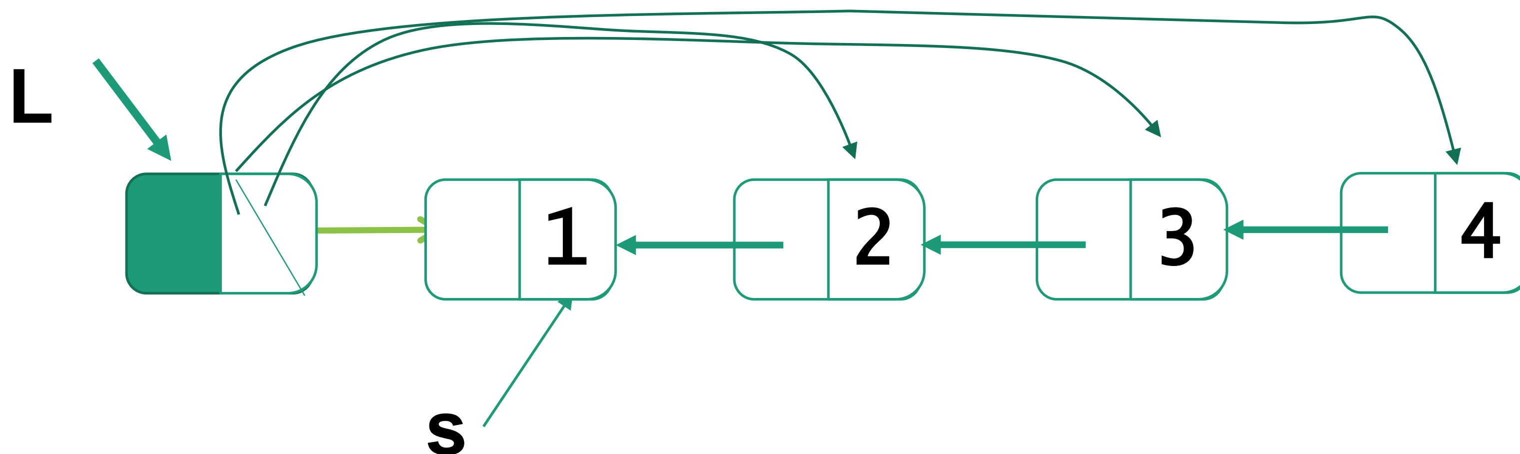
if (p→next == NULL) last = p;

建立链表（头插法建表）

- 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; }  }
```

头插法建表



```
s=(LNode*)malloc(sizeof(LNode));
s->next=L->next;
L->next=s;
```

- 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

```
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);
}
```

递增插入

```
void OrderInsert_L(LinkList &L, ElemType e)
{ LinkList q=L,p=L->next;
  while(p&&p->data<e)  {
    q=p;
    p=p->next;
    }
  q->next= (LinkList)malloc(sizeof(LNode));
  q->next->data=e;
  q->next->next=p;
}
```

查找

```
int LocateElem_L(LinkList L, ElemType e,
Status(*compare)(ElemType,ElemType))
 { int i=0;
   LinkList p=L->next;
   while(p)   {     i++;
     if(compare(p->data,e))
       return i;
     p=p->next;
   }
   return 0;
 }
```

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)
          return 0;
   else return i;
}
```

初始化

```
Status InitList_L(LinkList &L)
 { /* 操作结果：构造一个空的线性表L */
   L=(LinkList)malloc(sizeof(LNode)); /* 产生头结点,并使L指向此头结点*/
   if(!L) /* 存储分配失败*/
     exit(OVERFLOW);
   L->next=NULL; /* 指针域为空*/
   return OK;
 }
```

求长度

```
int ListLength_L(LinkList L) {
  int i=0;
  LinkList p=L->next; /* p指向第一个结点*/
  while(p) /* 没到表尾*/
  {
    i++;
    p=p->next;
  }
  return i;
}
```

遍历

```
Status ListTraverse_L(LinkList L,
void(*vi)(ElemType))  {
  LinkList p=L->next;
  while(p) {
    vi(p->data);
    p=p->next;
  }
  return OK;
}
```

销毁

```
Status DestroyList_L(LinkList &L)
{ /* 初始条件：线性表L已存在。操作结果：销毁线性表L */
   LinkList q;
   while(L)   {
     q=L->next;
     free(L);
     L=q;
   }
   return OK;
}
```

清空

```
Status ClearList_L(LinkList L)
{LinkList p,q;
  p=L->next; /* p指向第一个结点*/
  while(p) /* 没到表尾*/
  {    q=p->next;
    free(p);
    p=q;   }
  L->next=NULL; /* 头结点指针域为空*/
  return OK;
}
```

判空

```
Status ListEmpty_L(LinkList 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 */
  if(L->next) /* 非空*/
    return FALSE;
  else
    return TRUE;
}
```

带头结点和不带头结点的区别

	带头结点	不带头结点
链表置空	L->next = NULL	L=NULL
判空条件	L->next==NULL	L==NULL
头部插入结点N	N->next=L->next; L->next=N	if (L==NULL) L=N; else {N->next=L->next; L=N;}
删除第一个结点	if (L->next){ p=L->next; L->next=L->next->next; free(p);}	if(L){ p=L; L=L->next; free(p);}

找前驱

```
Status PriorElem_L(LinkList L, ElemType cur_e,
ElemType &pre_e) {
   LinkList q,p=L->next; /* p指向第一个结点*/
   if (p==NULL) return ERROR;
   while(p->next)/*p所指结点有后继*/{
     q=p->next; /* q为p的后继*/
     if(q->data==cur_e)     {
       pre_e=p->data;
       return OK;     }
     p=q; /* p向后移*/  }
   return INFEASIBLE;
 }
```

找后继

```
Status NextElem_L(LinkList L,ElemType
cur_e,ElemType &next_e) {
   LinkList p=L->next; /* p指向第一个结点*/
   if (p==NULL) return ERROR;
   while(p->next)/* p所指结点有后继*/{
     if(p->data==cur_e){
       next_e=p->next->data;
       return OK;     }
     p=p->next;
   }
   return INFEASIBLE;  }
```

例：用链表实现集合并运算 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)

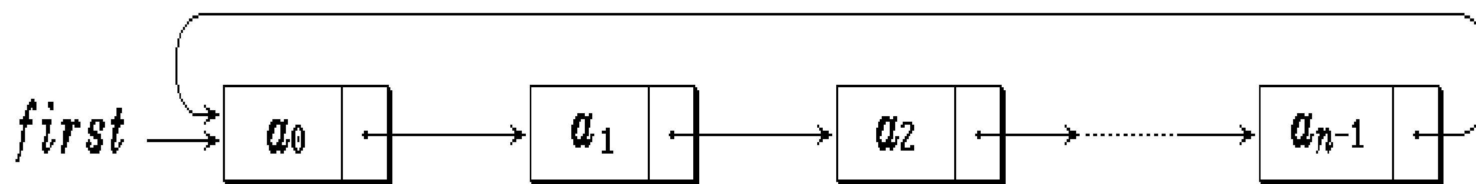
时间复杂度： O(m*n)；

3. 循环链表

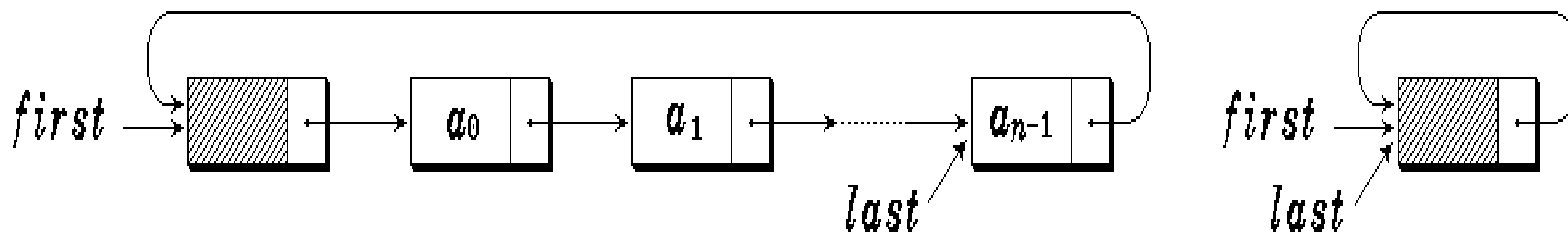
1）定义：循环链表是一种首尾相接的链表。

- 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。
- 循环条件：p->next != H
- 为简化操作，在循环链表中往往加入表头结点

■不带头结点的循环链表



■带头结点的循环链表



单链表 vs 循环链表

单链表	循环链表
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	从任一结点出发都可访问到表中所有结点；
链表中表尾结点的指针域是NULL	链表中表尾结点的指针域是指向链表头结点的指针H，没有指针域为NULL的结点

3. 循环链表

2）循环链表的操作

合并两个循环链表

建立循环链表

输出循环链表

合并两个单链表形成一个单链表

```
void union(LinkList &La, LinkList &Lb){
   LinkList p=La;
   while (p->next)  p=p->next;
   p->next = Lb->next;
   free(Lb);
}
```

La,Lb均是带头结点的单链表。

时间复杂度 O(m)

合并两个循环链表

```
void union(LinkList &La, LinkList &Lb){
    p=La;
    while (p->next!=La)  p=p->next;
    p->next=Lb->next;
    while (p->next!=Lb) p=p->next;
    p->next=La;
    free(Lb);
}
```

时间复杂度 O(m+n)

循环链表的创建

```
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;
       }
}
```

- //带头结点

循环链表的显示输出（带头结点）

```
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");
 }
```

leetcode中有关链表的题目

- 2 两数相加
- 141 判断环形链表
- 142 环形链表2
- 160 相交链表
- 82 删除排序链表中的重复元素II
- 24 两两交换链表中的结点
- 25 K个一组翻转链表
- 143 重排链表
- 707 设计链表

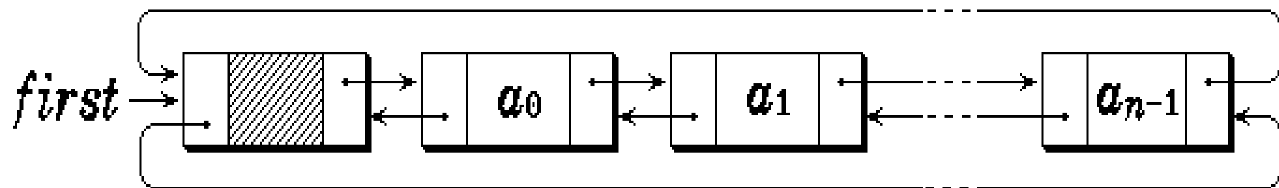
4.双向链表 (Doubly Linked List)

- 双向链表是指在前驱和后继方向都能遍历的线性链表。
- 1) 双向链表的结点结构:

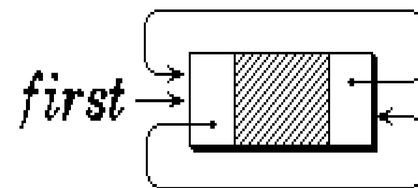
前驱方向 ← (a)结点结构 → 后继方向

- 双向链表通常采用带表头结点的循环链表形式。

- 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;



(b) 非空双向循环链表



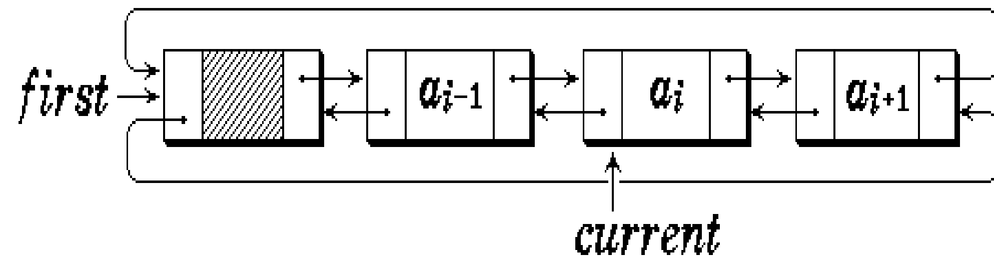
(c) 空表

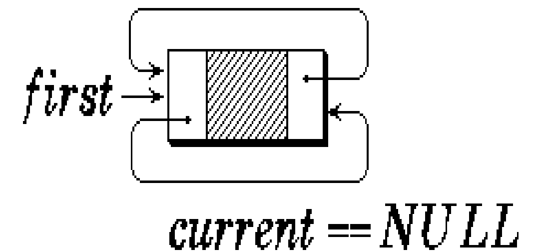
2）双向链表的类型定义

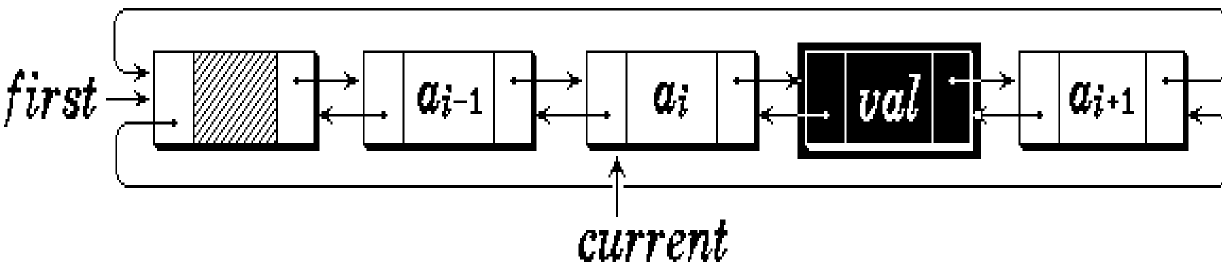
```
typedef struct DuLNode{
    ElemType   data;
    struct DuLNode   *prior;
    struct DuLNode   *next;
  }DuLNode, *DuLinkList;
DuLinkList  d,p,s;
```

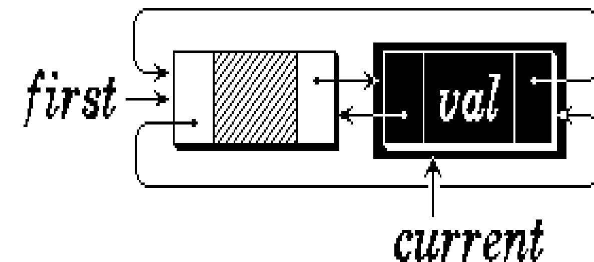
双向循环链表的插入算法

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



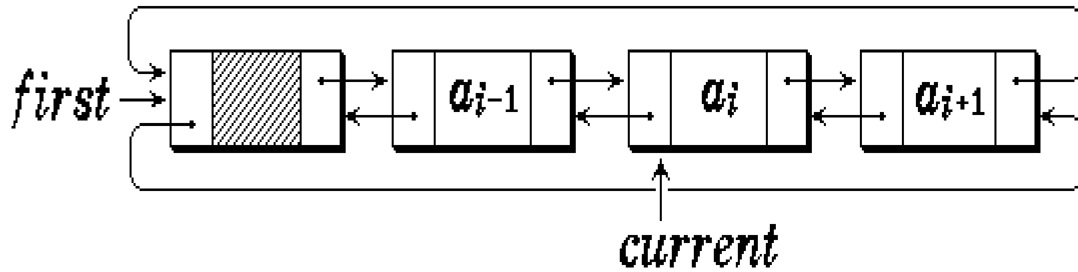


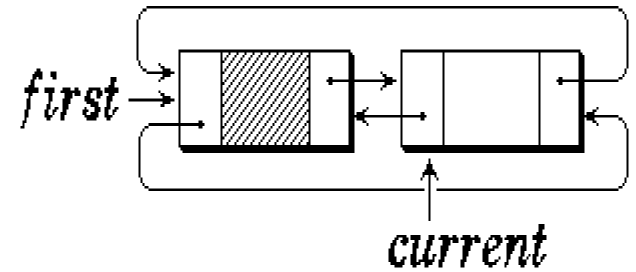


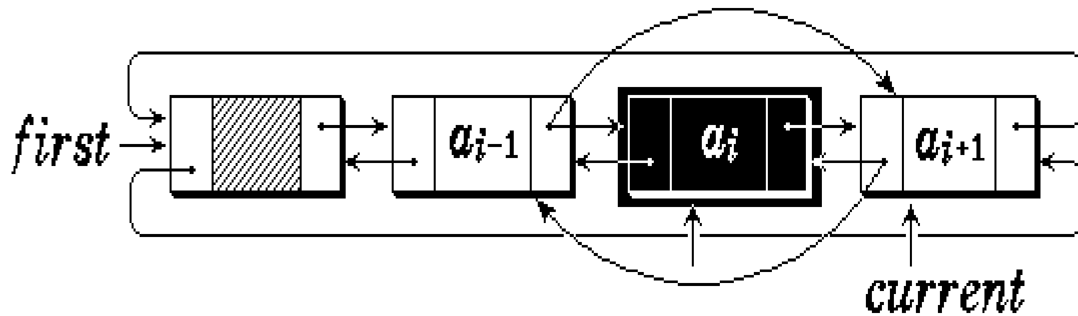


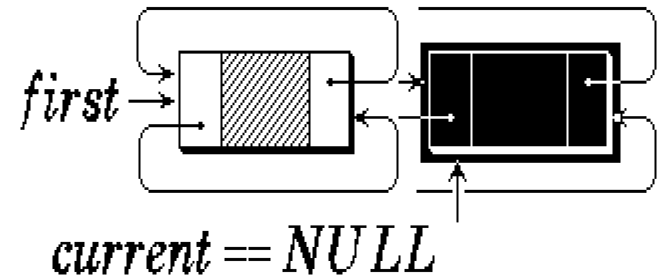
双向循环链表的删除算法

- current→next→prior = current→prior;
- current→prior→next = current→next;









3) 双向链表的基本操作

- 双向链表的建立
- 双向链表的输出

双向循环链表的建立

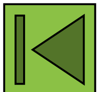
```
void CrtList_DuL(DuLinkList &L)
{   DuLinkList p;    int x;
    L=p=(DuLinkList)malloc(sizeof(DuLNode));
    L ->next=L;   L->prior =L;
    while (scanf("%d",&x),x!=0){
            p-
>next=(DuLinkList)malloc(sizeof(DuLNode));
            p->next->prior =p;   p=p->next;
            p->data=x;
  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

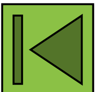
2.3.4 改进的线性链表

- 用上述定义的单链表实现线性表的操作存在的**问题**：
  1. 单链表的表长是一个隐含的值
  2. 在单链表的最后一个元素后面插入元素时，需遍历整个链表
  3. 在链表中，元素的“位序”概念淡化，结点的位置概念强化。
- **改进链表的设置**：
  1. 增加“表长”、“表尾指针”和“当前位置指针”三个数据域
  2. 将基本操作由“位序”改变为“指针”



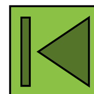
一个改进的带头结点的线性链表定义

```
typedef  struct LNode{ //结点类型
    ElemType  data;
    struct LNode *next;
    struct LNode *prev;
 }*Link，*Position;
typedef struct {  //链表类型
     Link  head, tail; /* 分别指向线性链表中的头结点和最后一个结点 */
     int     len;    //指示链表长度
     Link  current;  //指向当前访问的结点指针
     }LinkList
```



结点的基本操作

Status MakeNode(Link &p,ElemType e);
//分配由p指向值为e的结点，并返回OK;若分配失败，则返回ERROR
void FreeNode(Link &p);
//释放p所指结点



链表的基本操作

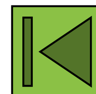
- 见课表p37~38
- 结构初始化和销毁结构

  Status InitList (LinkList &L);

  //构造一个空的链表，头指针、尾指针、当前指针都指向头结点，表长设为0。

  Status DestroyList(LinkList &L);
- 其他操作：

  表头插入、表头删除、表尾添加、表尾删除、结点前插入、结点后插入、取结点值、更新结点值、找结点的前驱、找结点的后继、查找元素等



顺序表和链表的对比

	顺序表	链表
从存储空间利用率角度出发	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连续的存储空间。	存储空间是动态分配的，无需预先分配，存储空间是不连续的。
	可能导致存储空间的浪费或溢出。	不会发生溢出。但指针域需占额外的存储空间。
从时间效率出发，按操作方式不同	若对线性表进行的主要操作是查找，插入和删除操作只在尾部进行时，宜采用顺序表存储结构。	若对线性表任意位置进行插入和删除操作频繁时，宜采用链表作为存储结构。
访问特性	随机存取（直接存取）	顺序存取

2.4 一元多项式的表示和相加

- n阶多项式Pn(x)有n+1项。

  系数 a0, a1, a2, …, an

  指数 0, 1, 2, …, n 按升幂排列

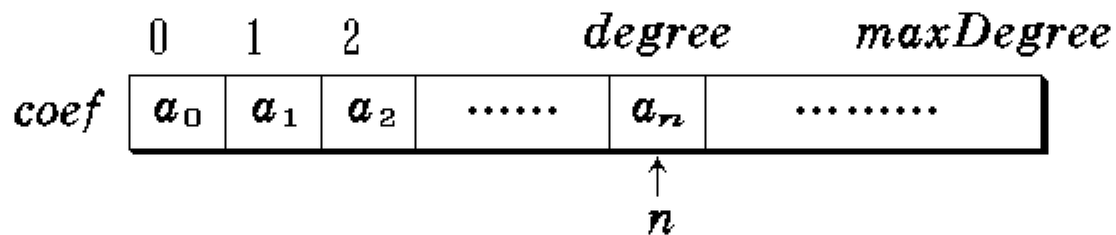
- 在计算机中，可以用一个线性表来表示

  P=(a0,a1, … , an)

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

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

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: $D=\{a_i \mid a_i \in TermSet, i=1,2,\ldots,m, \ m\geq 0$
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： $R1=\{<a_{i-1},a_i> \mid a_{i-1},a_i \in D$,且$a_{i-1}$中的指数值$<a_i$中的指数值, $i=2,\ldots n\}$

基本操作：；

void CreatPolyn(Polynomial&, int);
void PrintPolyn(Pplynomial);
void AddPolyn(Polynomial &,Polynomial &P);
void SubtractPolyn(Polynomial &,Polynomial &);
void MultiplyPolyn(Polynomial &,Polynomial &);

}

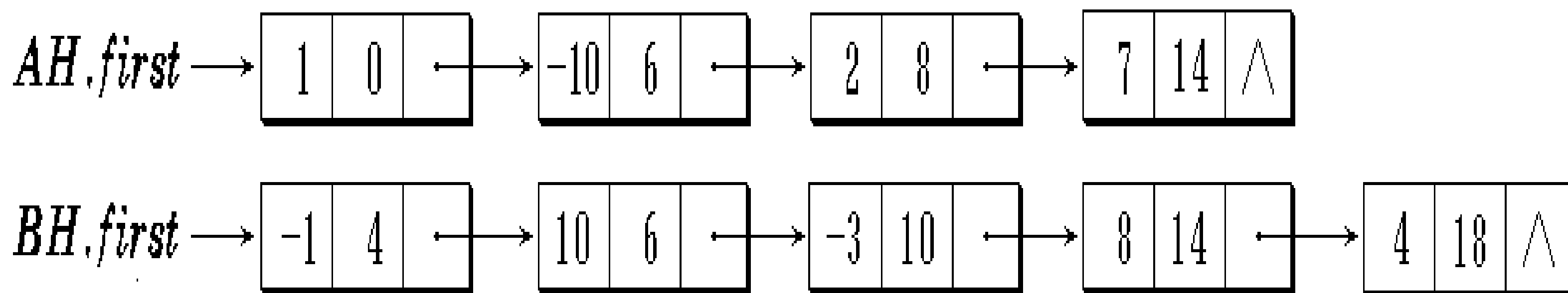
4. Polynomial抽象数据类型的实现

```
typedef struct /* 项的表示 */
  {   double coef; /* 系数 */
      int expn; /* 指数 */
    }term,ElemType;
    /* term用于本ADT, ElemType为LinkList的数据对象名 */
typedef LinkList polynomial;/*用带表头结点的有序链表表示多项式*/
```

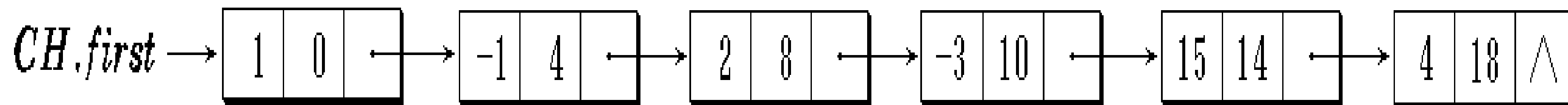
两个多项式的相加（链表形式）

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

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



(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。

查找和有序插入

1）查找

Status LocateElemP(LinkList L, ElemType e, Position &q, int(*compare)(ElemType, ElemType))

函数功能：　若升序链表L中存在与e满足判定函数compare()取值为0的元素，则q指示L中第一个满足compare()取值=0的结点的位置，并返回TRUE；否则q指示第一个与e满足判定函数compare()取值>0的元素的前驱的位置。并返回FALSE。*/

2）将值为e的结点插入或合并到升序链表L中

Status OrderInsertMerge(LinkList &L, ElemType e, int(* compare)(term,term))

查找

```
Status LocateElemP(LinkList L, ElemType e,
Position &q,
nt(*compare)(ElemType,ElemType))
 {Link pp=L.head,p=L.head->next;
   while(p&&(compare(p->data,e)<0)) {
     pp=p;     p=p->next;   };
    if(!p||compare(p->data,e)>0) {
     q=pp;     return FALSE;   }
   else /* 找到 */   {
     q=p;
     return TRUE;   }
 }
```

插入或合并到链表中

```
Status OrderInsertMerge(LinkList &L, ElemType e,
int(* compare)(term,term)) {Position q,s;
  if(LocateElemP(L,e,q,compare)) /*L中存在该指
数项 */
  {    q->data.coef+=e.coef;
    if(!q->data.coef) /* 系数为0 */
    { /* 删除多项式L中当前结点 */
      s=PriorPos(L,q); /* s为当前结点的前驱 */
```

```
if(!s) /* q无前驱 */
        s=L.head;
    DelFirst(L,s,q);    FreeNode(q);
    }     return OK;   }
  else /* 生成该指数项并插入链表 */
    if(MakeNode(s,e)) /* 生成结点成功 */
    {
      InsFirst(L,q,s);
      return OK;
    }
    else /* 生成结点失败 */
      return ERROR;
 }
```

创建多项式

```
void CreatPolyn(polynomial &P,int m) {
   Position q,s;   term e;   int i;
   InitList(P);
   for(i=1;i<=m;++i)
   {    scanf("%f,%d",&e.coef,&e.expn);
     if(!LocateElemP(P,e,q,cmp)) /* 当前链表中不存在该指数项,cmp是实参 */
       if(MakeNode(s,e)) /* 生成结点*/
         InsFirst(P,q,s); /*q后插入*/
   }
}
```

本章小结

- 1.掌握线性表的抽象数据类型的定义。（包括了逻辑结构和基本操作集）。
- 2.熟练掌握线性表的顺序存储结构的定义和实现。
- 3.熟练掌握线性表的链式存储结构的定义和实现。
- 4.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。

1.线性表的逻辑结构和基本操作集。

逻辑结构：数据元素之间是线性结构，一对一的关系。

特点：均匀的、有序的。

基本操作：

1）InitList(List &)

2）GetElem(List ,int , ElemType &)

3）LocateElem(List ,ElemType ,(*compare)())

4）ListInsert(List &,int,ElemType)

5）ListDelete(List &,int,ElemType &)

6）DestroyList(List &) 销毁线性表L

7）OrderInsert(List &, ElemType) 有序表的插入

8) MergeList(List , List, List&) 有序表的合并

9) UnionList(List&, List) 集合的合并

10)Traverse(List)

- 2. 线性表的顺序存储结构（顺序表）

  顺序表的类型定义

  基本操作的算法实现，以及时间复杂度和空间复杂度的分析（插入、删除、查找$O(n)$）

  取元素$O(1)$ 随机存取结构

- 3. 线性表的链式存储结构（链表）

  链表的类型定义（单链表、双向链表）

  头指针、头结点的含义

  单链表基本操作的算法实现，以及时间复杂度和空间复杂度的分析（取元素、查找、按位序插入和删除均是$O(n)$）

  循环链表与单链表算法的异同，为什么说循环链表用尾指针比头指针要好?

  双向链表的插入和删除

  了解改进的链表的类型定义及一些操作。*

附：用c++实现的一个链表类的定义：

```
template<class ElemType>
class LinkList{
private:
  struct node{
        ElemType data;
        node *next,*prev;
        node(const ElemType &x,node *p=NULL,
  node *n=NULL){      data=x;next=n;prev=p;}
        node():next(NULL),prev(NULL){}
        ~node(){}
  };
  node *head,*tail;
  int len;
  node* move(int i) const; //返回第i个结点的地址
```

```
public:
        LinkList();
        ~LinkList();
        void clear();
        int length() const{
                return len;
        }
        void insert(int i,const ElemType &x);
        void remove(int i,ElemType &e);
        int search(const ElemType &x) const;
        ElemType get(int i) const;
        void traverse() const;
        …
};
```

**STL（标准模板库）中的线性表

容器：C++把每种数据结构的实现称为一个容器，是为了保存一组类型相同的对象而设计的类。

迭代器：用于表示容器中的对象位置的类型，迭代器对象相当于指向容器中对象的指针，是将容器中对象的位置信息封装起来

STL中的线性表有3种实现：

vector：用动态增长的数组存储数据元素，类似顺序表，随机存取，尾部添加O(1)

list：用双向链表实现，插入删除是O(1)

deque：优化的存储结构

vector类和list类的共同操作：

int size() const	元素个数
void clear()	清空
bool empty()	判空
void push_back(const object &x)	添加到表尾
void pop_back()	删除表尾
const object & back() const	返回表尾
const object & front() const	返回表头

vector类和list类中与迭代器相关的操作：

iterator begin()	返回表头位置
iterator end()	返回表尾位置
iterator insert(iterator pos, const object &x)	pos位置前插入x
iterator erase(iterator pos,const object &x)	删除pos位置的元素
iterator erase(iterator start,iterator end)	删除区间的元素

迭代器类的常用操作:

函数	作用
itr++,++itr	迭代器指向下一元素
*itr	取迭代器指向的元素
itr1==itr2	判断两个迭代器是否指向同一元素,是返回true,否则false
itr1!=itr2	判断是否指向不同元素

vector类特有的操作：

object &operator[](int idx)	返回下标位置的对象
object &at(int idx)	返回表尾位置
int capacity()	数组容量
void reserve(int newCapacity)	设置数组容量

list类的操作

void push_front(const object &x)	表头插入
void pop_front()	表头删除

扩展：容量扩充策略

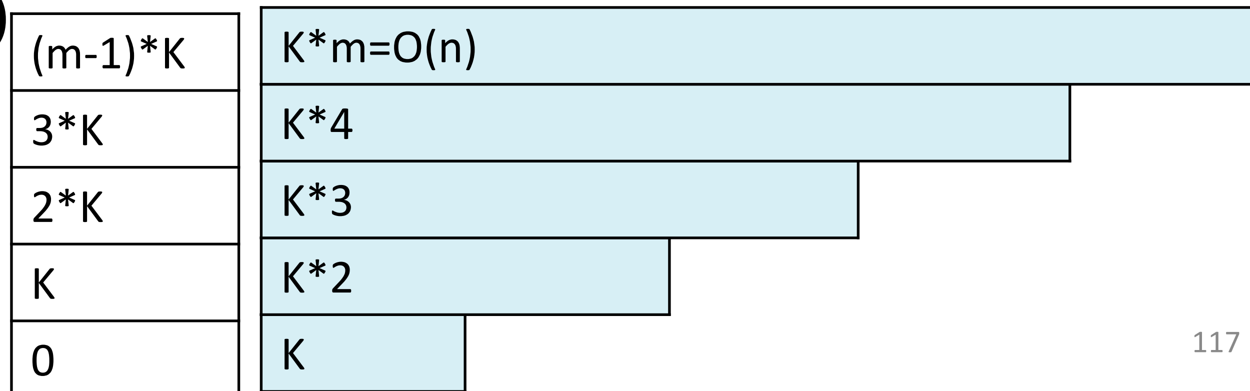
- 1.递增策略

  扩容时每次增量为固定值K(常数)，假定初始容量设置为0，连续插入n=m*K个元素

  于是，在第1、K+1、2K+1、。。。次插入时，都需扩容。

  每次扩容过程中复制原顺序表的时间成本依次为：0, K, 2K, ... ，(m-1)*K

  总耗时=m*(m-1)*K/2 =$O(n^2)$，每次的分摊成本为$O(n)$



扩展：容量扩充策略

- 2. 加倍策略

  扩容时容量为扩容前的2倍。假定在初始容量为1的满顺序表中，连续插入n=2^m个元素。

  于是，在第1、2、4、8、16。。。次插入时，都需扩容。

  每次扩容过程中复制原顺序表的时间成本依次为：1, 2, 4, 8, 16, ... , 2^m-=n

  总耗时=2^(m+1)-1 =$O(n)$每次的分摊成本为

2^(m-1)	2^m =O(n)
4	8
2	4
1	2
0	1

扩展：两种扩充策略对比

	递增策略	倍增策略
累计增容时间	$O(n^2)$	$O(n)$
分摊增容时间	$O(n)$	$O(1)$
装填因子	=100%	>50%

思考：反复执行删除操作后，需要考虑回收部分空间，你会选择什么策略？

扩展：分摊分析*

- 分摊时间复杂度（amortized complexity）：连续实施足够多次操作，所需成本摊还到单次操作
  - 从实际可行的角度，对一系列操作做整体的考量
  - 更加忠实地刻画了可能出现的操作序列
- 平均时间复杂度：根据各种操作出现概率的分布，将对应成本加权平均
  - 各种可能的操作，作为独立事件分别考查
  - 割裂了操作之间的相关性和连贯性