**第二章 线性表**

**一.利用现有操作组成更复杂的操作**

**将在线性表Lb中且不在线性表La中的数据元素，合并到线性La中。**

void union(List &La,List Lb){

La\_len=LengthList(La); //求线性表La的长度

Lb\_len=LengthList(Lb); //求线性表Lb的长度

for (i=1;i<= Lb\_len;i++){

GetElem(Lb,i,e); //取Lb的的第i个数据元素赋给e

//即依次取出Lb中的所有元素

if (!LocateElem(La,e,equal)) //判断e在La中是否存在

ListInsert(La,++La\_len,e); //不存在则插入

}

}

**二 线性表的表示和实现**

**1. 线性表顺序结构在C语言中的定义（静态分配）**

元素所占空间和表长合并为C语言的一个结构类型：

#define maxleng 100

typedef struct

{ ElemType elem[maxleng]；//下标:0,1,...,maxleng-1

int length； //表长

} SqList；

SqList La；

La.length---表长

La.elem[0]----a1

La.elem[La.length-1]---an

**2. 线性表顺序结构在C语言中的定义（动态分配）**

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

typedef struct

{ ElemType \*elem；//存储空间基地址

int length； //表长

int listsize; //当前分配的存储容量

//（以sizeof(ElemType)为单位

} SqList；

SqList Lb；

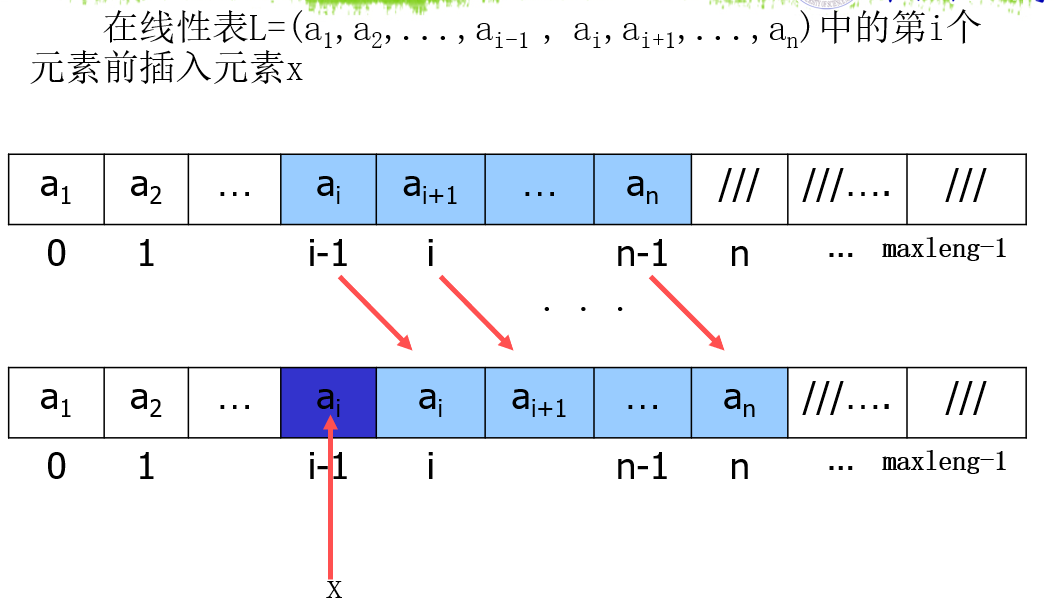
Lb.length---表长

Lb.elem[0]----a1

Lb.elem[Lb.length-1]--- an

**3. 插入算法实现举例—顺序表操作**

**（设 L.elem[0..maxleng-1]中有length个元素，在L.elem[i-1]之前插入新元素e，1<=i<=length）**

****

**A) 算法1：（用指针指向被操作的线性表,静态分配）**

**//设 L.elem[0..maxleng-1]中有length个元素，在**

**L.elem[i-1]之前插入新元素e，(1<=i<=length+1)**

int Insert1(SqList \*L,int i,ElemType e)

{ if (i<1||i>L->length+1) return ERROR //i值不合法

if (L->length>=maxleng) return OVERFLOW //溢出

for (j=L->length-1；j>=i-1；j--；) //先移动后面元素

L->elem[j+1]=L->elem[j]； //向后移动元素

L->elem[i-1]=e； //插入新元素

L->length++； //长度变量增1

return OK; //插入成功

}

**B)** **算法2：（用引用参数表示被操作的线性表,静态分配）**

**//设 L.elem[0..maxleng-1]中有length个元素，在**

**L.elem[i-1]之前插入新元素e，(1<=i<=length+1)**

Status Insert2(SqList &L,int i,ElemType e)

{ if (i<1||i>L.length+1) return ERROR //i值不合法

if (L.length>=maxleng) return OVERFLOW //溢出

for (j=L.length-1；j>=i-1；j--；)

L.elem[j+1]=L.elem[j]； //向后移动元素

L.elem[i-1]=e； //插入新元素

L.length++； //长度变量增1

return OK //插入成功

} 注：指针用“->”,引用用“.”

**C) 算法3：动态分配线性表空间,用引用参数表示被操作的线性表**

int Insert3(SqList &L,int i, ElemType e)

{int j;

if (i<1 || i>L.length+1) //i的合法取值为1至n+1

return ERROR;

if (L.length>=L.listsize) /\*溢出时扩充\*/

{

ElemType \*newbase;

newbase=(ElemType \*) realloc(L.elem,

(L.listsize+LISTINCREMENT)\*sizeof(ElemType));

if (newbase==NULL) return OVERFLOW; //扩充失败

L.elem=newbase;

L.listsize+=LISTINCREMENT;

}

//向后移动元素，空出第i个元素的分量elem[i-1]

for(j=L.length-1;j>=i-1;j--)

L.elem[j+1]=L.elem[j];

L.elem[i-1]=e; /\*新元素插入\*/

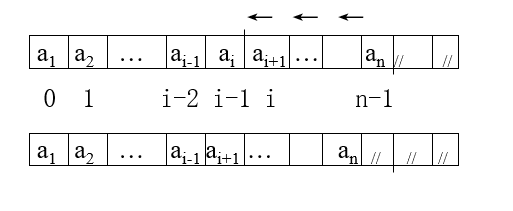
L.length++; /\*线性表长度加1\*/

return OK;

}

注：realloc函数的功能和使用方式

**4. 顺序表删除操作**



int delete(SqList \*L,int i) //删除第i个元素

{ if (i<1 ||i>L->length)

{ printf(“not exist”);

return ERROR;}

else { for(j=i;j<=L->length-1;j++) //先移前面元素

L->elem[j-1]=L->elem[j];

L->length--; return OK;}

}

**5. 例题： 设计一个算法，将顺序表重新排列成以第一个节点为界的两部分，前一部分元素都小于它，后一部分元素都大于或等于它。注：顺序表中不存在于a0相等的其他数据元素。**

数据结构定义如下：

typedef struct

{ ElemType \*elem；//存储空间基地址

int length； //表长

int listsize; //当前分配的存储容量

//（以sizeof(ElemType)为单位

} SqList；

**A）**思路，用pivot存放基准即L->elem[0], 用 i 和 j 分别指向顺序表 L 的第一个元素和最后一个元素，当 i 不等于 j 时循环： 从右向左扫描，找一个关键字小于等于 pivot 的elem[j], 再从左向右扫描，找一个关键字大于等于pivot的记录elem[i], 将两者进行交换。

void move1(SqList \*L)

{ int i=0, j= L->length – 1;

ElemType pivot = L->elem[0]; //以elem[0]为基准

ElemType tmp;

while(i != j) //从区间两端交替向中间扫描直至i=j

{

while(i<j && L->elem[j] > pivot)

j--; //从右向左找第1个小于等于pivot的元素

while(i<j && L->elem[i] < pivot)

i++; //从左向右找第1个大于等于pivot的元素

if(i<j) //交换 L->elem[i]和L->elem[j]

{ tmp = L->elem[i];

L->elem[i] = L->elem[j];

L->elem[j] = tmp;

}

}

}

**B)** **思路，用i（初值为0）从左向右查找，j（初值为L->length-1）从右向左查找。以elem[0]为基准（将其先存放在pivot中），当i 不等于 j 时循环（即循环到i和j指向同一元素时为止，该位置为基准元素应放的位置）：从右向左扫描，找一个关键字小于pivot的elem[j], 找到后将elem[j]放到elem[i]处（用elem[j]覆盖elem[i]）,再从左向右扫描，找一个关键字大于等于pivot的记录elem[i],找到后将elem[i]放到elem[j]处（用elem[i]覆盖elem[j]）。最后让elem[i]=pivot。**

**顺序表中可以存在多个等于elem[0]的元素。**

void move2(SqList \*L)

{ int i=0, j=L->lenght-1;

Elemtype pivot = L->elem[0];

while(i != j)

{

while( i<j && L->elem[j] >= pivot)

j--;

L->elem[i] = L->elem[j];

while( i<j && L->elem[i] < pivot)

i++;

L->elem[j] = L->elem[i];

}

L->elem[i] = pivot;

}

**6．顺序存储结构的评价**

**A）.优点：**

(1)是一种随机存取结构，存取任何元素的时间是一个

常数，速度快；

(2)结构简单，逻辑上相邻的元素在物理上也是相邻的；

(3)不使用指针，节省存储空间。

**B）.缺点：**

(1)插入和删除元素要移动大量元素，消耗大量时间；

(2)需要一个连续的存储空间；

(3)插入元素可能发生“溢出”；

(4)自由区中的存储空间不能被其它数据占用(共享)。

**三 线性表的链式表示和实现**

1. **单链表（又称线性链表）（带不带头节点两种形式）的结点结构**

**typedef struct node**

**{ ElemType data； //data为抽象元素类型**

**struct node \*next； //next为指针类型**

**} Lnode, \*Linklist；**

**指针变量head,p,q的说明:**

**Linklist head,p,q； 等同于Lnode \*head, \*p, \*q;**

**2. 生成单链表**

**A）生成“先进先出”单链表（链式队列）**

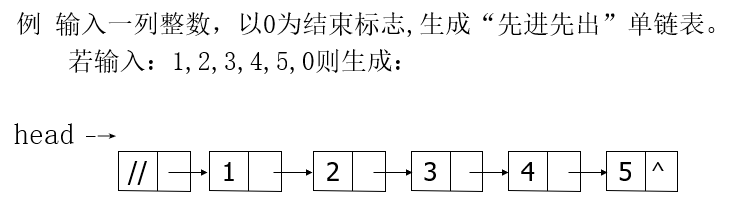
**#define LENG sizeof(struct node) //结点所占的单元数**

**struct node //定义结点类型**

**{ int data； //data为整型**

**struct node \*next； //next为指针类型**

**}；**

****

**struct node \*creat1( )**

**{ struct node \*head,\*tail,\*p； //变量说明**

**int e；**

**head=(struct node \*)malloc(LENG)； //生成表头结点**

**tail=head； //尾指针指向表头**

**scanf(“%d”,&e)； //输入第一个数**

**while (e!=0)； //不为0**

**{ p=(struct node \*)malloc(LENG)；//生成新结点**

**p->data=e； //装入输入的元素e**

**tail->next=p； //新结点链接到表尾**

**tail=p； //尾指针指向新结点**

**scanf(“%d”,&e)； //再输入一个数**

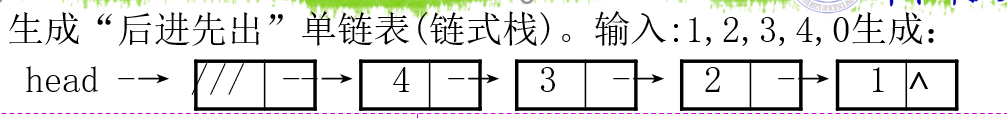
**}**

**tail->next=NULL； //尾结点的next置为空指针**

**return head； //返回头指针**

**}**

**B）生成“后进先出”单链表(链式栈)。**

****

**struct node \*creat2( ) //生成“后进先出”单链表**

**{ struct node \*head,\*p；**

**int e;**

**head=(struct node \*)malloc(LENG)； //生成表头结点**

**head->next=NULL； //置为空表**

**scanf(“%d”,&e)； //输入第一个数**

**while (e!=0)； //不为0**

**{ p=(struct node \*)malloc(LENG)；//生成新结点**

**p->data=e； //输入数送新结点的data**

**p->next=head->next； //新结点指针指向原首结点**

**head->next=p； //表头结点的指针指向新结点**

**scanf(“%d”,&e)； //再输入一个数**

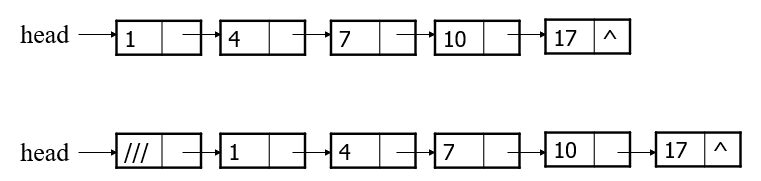
**}**

**return head； //返回头指针**

**}**

**3.** **: 输入一列整数,以0为结束标志,生成递增有序单链表。(不包括0）**

**递增有序单链表的两种形式：**

****

**A)** **假定有一个无表头结点的递增排序的链表，插入一个新结点，使其仍然递增。**

main()

{

struct node \*head； //定义头指针

head=NULL； //置为空表

scanf(“%d”,&e)； //输入整数

while (e!=0)； //不为 0，未结束

{

head=creat3\_1(head,e); //插入递增有序单链表

scanf(“%d”,&e)； //输入整数

}

}

**生成不带头结点的递增有序单链表。(不包括0）**

struct node \* creat3\_1(struct node \*head，int e)

{ q=NULL； p=head； //q,p扫描,查找插入位置

while (p && e>p->data) //未扫描完,且e大于当前结点

{ q=p； p=p->next；} //q,p后移,查下一个位置

f=(struct node \*)malloc(LENG)； //生成新结点

f->data=e； //装入元素e

if (p==NULL){

f->next=NULL;

if (q==NULL) //(1)对空表的插入

head=f;

else q->next=f;} //(2)作为最后一个结点插入

else if (q==NULL) //(3)作为第一个结点插入

{f->next=p; head=f;}

else

{f->next=p； q->next=f；} //(4)一般情况插入新结点

return head; }

**B) 假定有一个有表头结点的递增排序的链表，插入一个新结点，使其仍然递增。**

main()

{

head=(struct node \*)malloc(LENG)； //生成表头结点

head->next=NULL； //置为空表

scanf(“%d”,&e)； //输入整数

while (e!=0)； //不为 0，未结束

{

creat3\_2(head,e); //插入递增有序单链表head

scanf(“%d”,&e)； //输入整数

}

}

**生成带头结点的递增有序单链表。(不包括0）**

void creat3\_2(struct node \*head，int e)

**{** q=head；

p=head->next； //q,p扫描,查找插入位置

while (p && e>p->data) //未扫描完,且e大于当前结点

{ q=p；

p=p->next； //q,p后移,查下一个位置

}

f=(struct node \*)malloc(LENG)； //生成新结点

f->data=e； //装入元素e

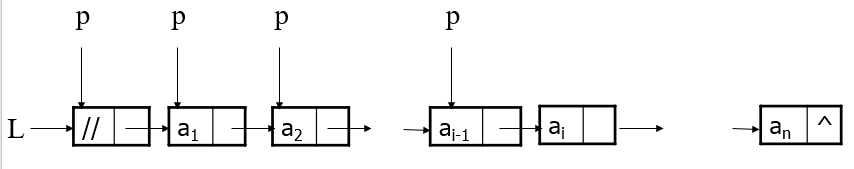
f->next=p； q->next=f； //插入新结点

**}**

**4. 在单链表的指定位置插入新元素**

**输入：头指针L、位置i、数据元素e**

**输出：成功返回OK，否则ERROR**

****

程序：

int insert( Linklist &L，int i, ElemType e)

{p=L；

j=1;

while (p && j<i)

{ p=p->next； //p后移,指向下一个位置

j++；}

if (i<1 || p==NULL) //插入点错误

return ERROR;

f=(LinkList) malloc(LENG)； //生成新结点

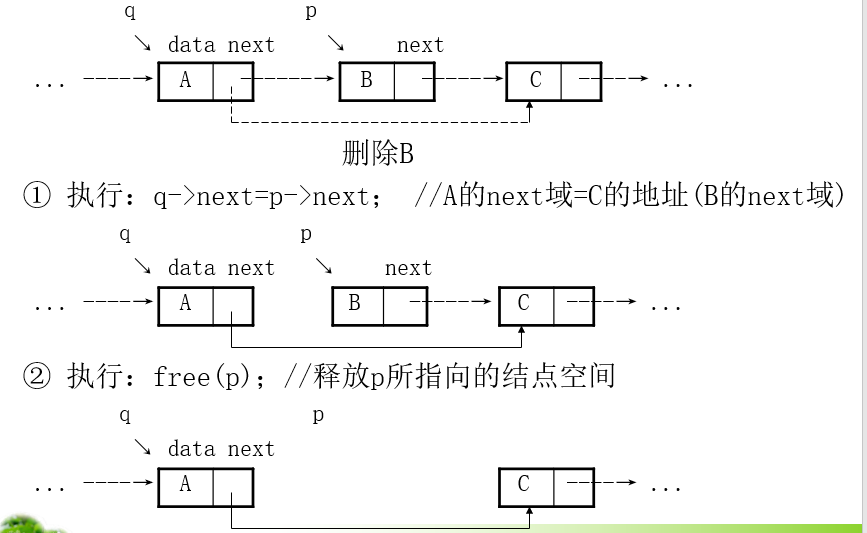
f->data=e； //装入元素e

f->next=p->next； p->next=f； //插入新结点

return OK;

}

**5. 在单链表中删除一个结点**



**A）在带表头结点的单链表中删除元素值为e的结点(可能有多个)**

int Delete1(Linklist head, ElemType e)

{ struct node \*q,\*p;

int result=Yes；

q=head； p=head->next； //q,p扫描

while(p)

{

while(p&&p->data!=e) //查找元素为e的结点

{

q=p； //记住前一个结点

p=p->next； //查找下一个结点

}

if (p) //有元素为e的结点

{

q->next=p->next； //删除该结点

free(p)； //释放结点所占的空间

p=q->next; //p指向下一个结点

result=Yes;

}

}

return result;

}

**B）在单链表中删除指定位置的元素**

**输入：头指针L、位置i**

**输出：将i位置结点数值域赋值给e，返回删除结果**

int Delete2( Linklist &L，int i,ElemType &e)

{p=L；

j=1;

while (p->next && j<i) //循环结束时p不可能为空

{ p=p->next； //p后移,指向下一个位置

j++；}

if (i<1 || p->next==NULL) //删除点错误

return ERROR;

q=p->next； //q指向删除结点

p->next=q->next； //从链表中摘出

e=q->data; //取走数据元素值

free(q)； //释放结点空间

return OK;

}

**6. 将两个有序单链表La和Lb合并为有序单链表Lc:**

**（该算法利用原单链表的结点）**

**输入：两单链表的头指针**

**输出：合并后的单链表的头指针**

struct node \*merge(struct node \*La, struct node \*Lb)

{

struct node \*pa,\*pb,\*pc；

pa=La->next； //pa指向表La的首结点

pb=Lb->next； //pb指向表Lb的首结点

pc=La； //使用表La的头结点,pc为尾指针

free(Lb)； //释放表Lb的头结点

while (pa && pb) //表La表Lb均有结点

if (pa->data<=pb->data) //取表La的一个结点

{ pc->next=pa； //插在表Lc的尾结点之后

pc=pa； //变为表Lc新的尾结点

pa=pa->next； //移向表La下一个结点

}

else //取表Lb的一个结点

{ pc->next=pb； //插在表Lc的尾结点之后

pc=pb； //变为表Lc新的尾结点

pb=pb->next； //移向表Lb下一个结点

}

if (pa) pc->next=pa； //插入表La的剩余段

else pc->next=pb； //插入表Lb的剩余段

return La;

}

**7）给定一个带头结点的单链表 L，设计一个算法使 L 中的元素递增有序排列**

void Sort(LinkList L){ //本算法实现将单链表L的结点重排，使其递增有序

LinkList p=L->next, pre;

LinkList r=p->next; //r保持p后继结点指针，以保证不断链

p->next=NULL; //构造只含一个数据结点的有序表

p=r;

while(p!=NULL){

r=p->next; //保存p的后继结点指针

pre=L;

while (pre->next!=NULL && pre->next->data<p->data)

pre=pre->next; //在有序表中查找插入的前驱结点pre

p->next=pre->next; //将p 插入到pre 之后

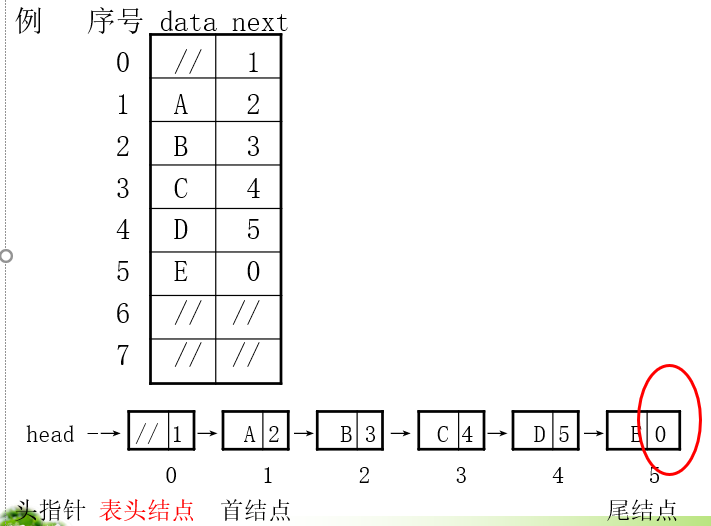
pre->next=p;

p=r; //扫描原单链表中剩下的结点

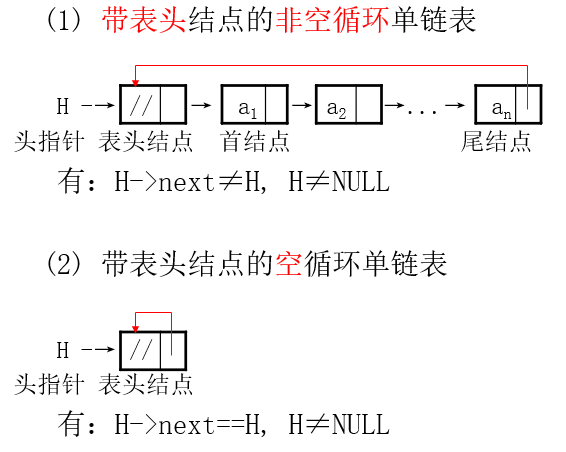
}

}

**静态链表----用数组描述的链表**

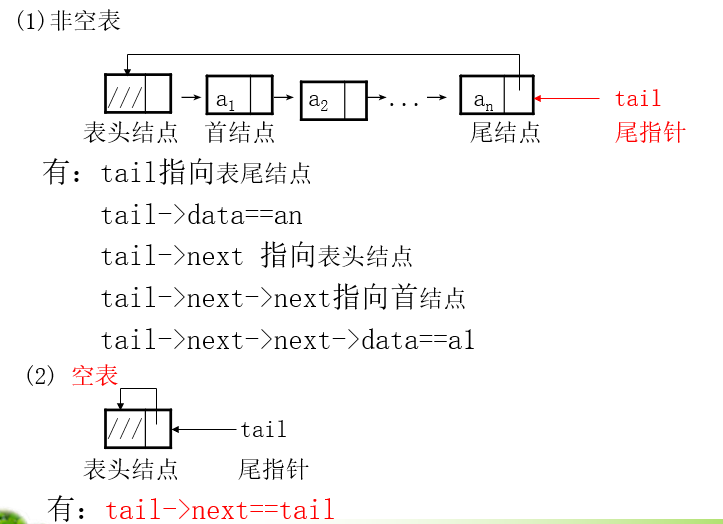
**  
循环链表——一般带头结点**

1. **一般形式（单链的循环链表）**

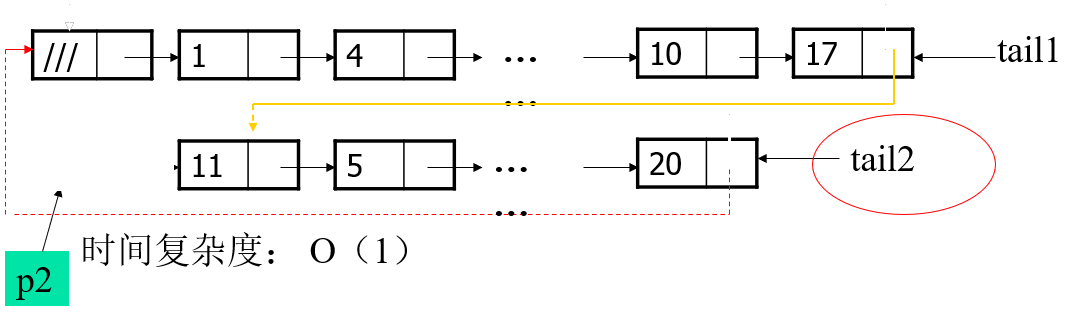
****

1. **只设尾指针的循环链表**

**A）一般形式**

****

**B）两循环链表首尾相连（p2指向第二个链表的头结点）**

****

**(1) p2=tail2->next;**

**(2) tail2->next=tail1->next;**

**(3) tail1->next=p2->next;**

**(4) free(p2);**

**3.求以head为头指针的循环单链表的长度,**

**并依次输出结点的值。 算法如下：**

int length(struct node \*head)

{ int leng=0； //长度变量初值为0

struct node \*p；

p=head->next； //p指向首结点

while (p!=head) //p未移回到表头结点

{ printf(“%d”,p->data)；//输出

leng++； //计数

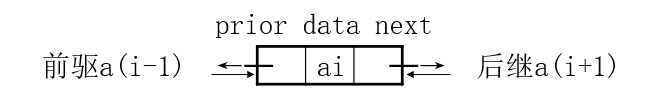
p=p->next；} //p移向下一结点

return leng； //返回长度值

}

**双向链表**

1. **结点结构**

****

结点类型定义

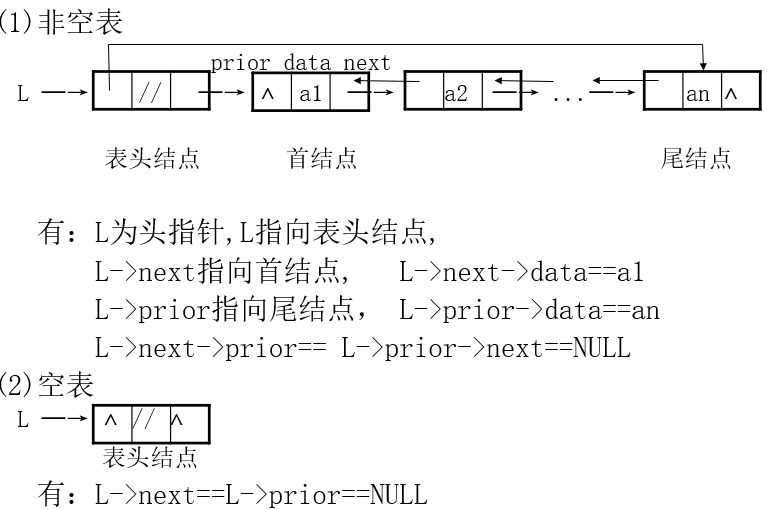
typedef struct Dnode

{ ElemType data； //data为抽象元素类型

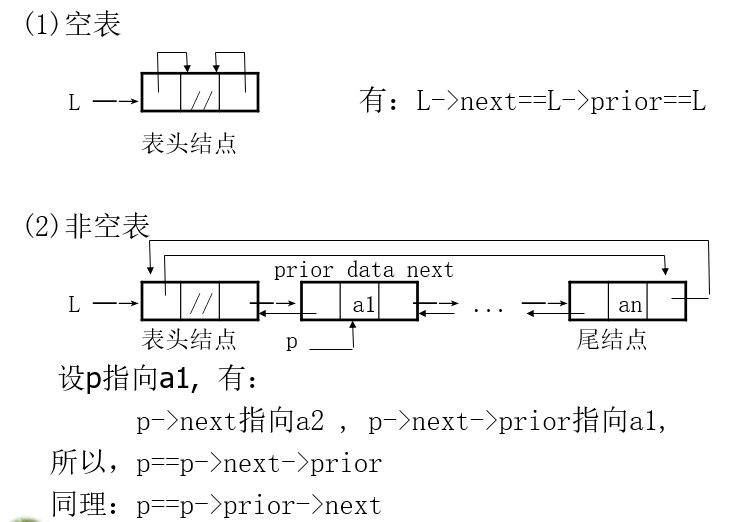
struct Dnode \*prior,\*next； //prior,next为指针类型

}\*DLList //DLList为指针类型

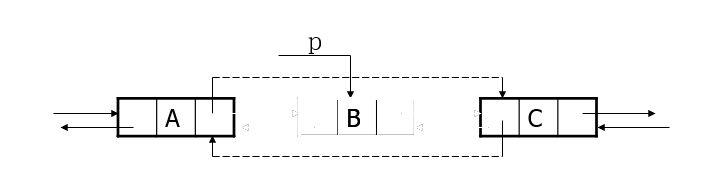
1. **双向链表的一般形式**

****

1. **双向循环链表的一般形式**

****

1. **已知指针p指向结点B，删除B**

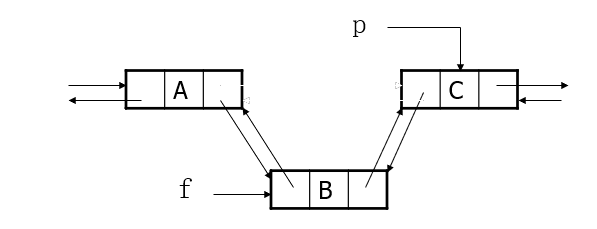
****

p->prior->next=p->next； //结点A的next指向结点C

p->next->prior=p->prior； //结点C的prior指向结点A

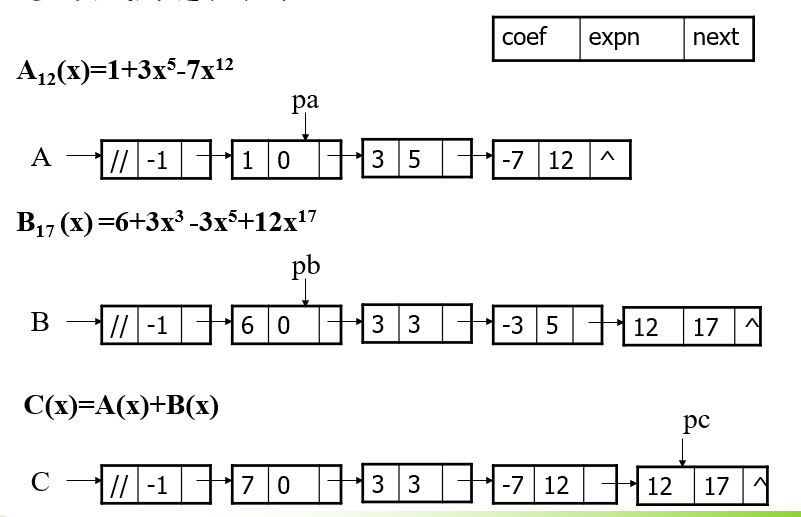
free(p)； //释放结点B占有的空间

**5.** **已知指针p指向结点C，在A、C之间插入结点B**

****

* 1. f->prior=p->prior； //结点B的prior指向结点A
  2. f->next=p； //结点B的next指向结点C
  3. p->prior->next=f； //结点A的next指向结点B
  4. p->prior=f； //结点C的prior指向结点B

1. **多项式的链表表示**



**C(x)=A(x)+B(x)的算法步骤：**

1、 pa、pb分别指向首元素结点,产生C(x)的空链表,pc指向头结点；

2、 pa不为空并且pb不为空,重复下列操作：

2-1 pa->expn等于pb->expn

（a） pa->coef+pb->coef不等于零： 产生新结点，添加到pc后， pc指向新结点。 pa、pb后移

（b） pa->coef+pb->coef等于零： pa、pb后移

2-2 pa->expn小于pb->expn：根据pa产生新结点，添加到pc后， pc指向新结点pa后移

2-3 pa->expn大于pb->expn：根据pb产生新结点，添加到pc后， pc指向新结点pb后移

3、pa为空，取pb剩余结点产生新结点， pb为空，取pa剩余结点产生新结点，依次添加到pc的后面。