第二章　线性表

制作：数据结构在线课程课题组

南京审计大学 信息工程学院

2020.09

内容概要

1、理解线性表的逻辑结构及定义

2、掌握线性表的顺序存储结构及操作实现

3、掌握线性表的链式存储结构及操作实现

4、线性表结构应用举例方法

2.1 线性表的类型定义

☆ 线性表（linear_list）：N个数据元素的有限序列，数据元素之间具有线性关系。

记作：(a1,a2,a3,……an)　　ai是数据元素

☆ 线性关系：除第一个元素外，每个元素有且仅有一个前驱；除最后一个元素外，每个元素有且仅有一个后继；

☆ 特点：数据元素之间的关系是它们在数据集合中的相对位置。

抽象数据类型**线性表**的定义如下

ADT List {

数据对象：

$D=\{ a_i \mid a_i \in ElemSet, i=1,2,...,n, \ n\geq 0 \}$

{ 其中**n** 为线性表的表长; }

数据关系：

$R1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, \ i=2,...,n \}$

{ 设线性表为 $(a_1, a_2, \ldots, a_i, \ldots, a_n)$,

称 i 为 a_i 在线性表中的位序。 }

基本操作：

结构初始化操作

结构销毁操作

引用型操作

加工型操作

} ADT List

初始化操作

InitList(&L)

操作结果：

构造一个空的线性表L。

结构销毁操作

DestroyList(&L)

初始条件：线性表 L 已存在。

操作结果：销毁线性表 L。

引用型操作：

ListEmpty(L)　　ListLength(L)

PriorElem(L, cur_e, &pre_e)

NextElem(L, cur_e, &next_e)

GetElem(L, i, &e)

LocateElem(L, e, compare( ))

ListTraverse(L, visit( ))

线性表判空操作

ListEmpty(L)

初始条件：　线性表**L**已存在。

操作结果：　若**L**为空表，则返回TRUE，否则FALSE。

求线性表的长度操作

ListLength(L)

初始条件：　线性表**L**已存在。

操作结果：　返回**L**中元素个数。

求数据元素的前驱

PriorElem(L, cur_e, &pre_e)

初始条件： 线性表L已存在。

操作结果： 若cur_e是L的元素，但不是第一个，则用pre_e 返回它的前驱，否则操作失败，pre_e无定义。

求数据元素的后继

NextElem(L, cur_e, &next_e)

初始条件： 线性表L已存在。

操作结果： 若cur_e是L的元素，但不是最后一个，则用next_e返回它的后继，否则操作失败，next_e无定义。

求线性表中某个数据元素

GetElem(L, i, &e)

初始条件：线性表L已存在，
且 1≤i≤LengthList(L)

操作结果：用 e 返回L中第 i 个元素的值。

定位函数 **LocateElem(L, e, compare( ))**

初始条件： 线性表L已存在，e为给定值，
compare( )是元素判定函数。

操作结果： 返回L中第1个与e满足关系
compare( )的元素的位序。
若这样的元素不存在，
则返回值为0。

遍历线性表

ListTraverse(L, visit( ))

初始条件： 线性表L已存在。
Visit() 为某个访问函数。

操作结果： 依次对L的每个元素调用函数visit( )。一旦visit( )失败，则操作失败。

加工型操作：

ClearList(&L)

ListInsert(&L, i, e)

ListDelete(&L, i, &e)

线性表置空

ClearList(&L)

初始条件：　线性表**L**已存在。

操作结果：　将**L**重置为空表。

插入数据元素

ListInsert(&L, i, e)

初始条件： 线性表L已存在，
且 1≤i≤LengthList(L)+1

操作结果： 在L的第i个元素之前插入新的元素e，L的长度增1。

删除数据元素

ListDelete(&L, i, &e)

初始条件： 线性表L已存在且非空，
1≤i≤LengthList(L)

操作结果： 删除L的第i个元素，并用e返回其值，L的长度减1。

线性表ADT的应用举例

例1：已知有线性表L，要求删除所有X的出现

```
Status del (List &L, ElemType  x)
  { k=LocateElem(L,x,equal);
      while(k!=0)
      {   Listdelete(L,k,e);
          k=LocateElem (L,x,equal);
      }
   return(OK);
  }
```

2.2 线性表的顺序表示和实现

一、顺序存储结构

1、存储方式：用一组地址连续的存储单元依次存储线性表的各个元素。具体地：假设存储每个元素占用l个存储单元，并且以所占的第一个单元的地址作为该元素的存储地址，于是：

一、顺序存储结构

基地址：首元的起始地址

可随机存取

一、顺序存储结构

2、特点：

☆存储空间必须是连续的，预分配。

☆逻辑顺序与物理顺序一致，用物理上的相邻来表示逻辑上的线性关系。

☆任意相邻元素之间无空闲空间，且相距为 *l* 。

☆已知基地址，可以计算出任意元素的存储地址：

$$LOC(a_i)=base+(i-1)\times l$$

一、顺序存储结构

3、存储实现：

在高级语言中，数组是采用顺序存储的，因而，我们可以用高级语言中的数组类型来说明上面的存储方式。

在对线性表进行存储实现时，我们要定义一个结构体类型，应在其中定义三个域分别用来表示：

（1）存储线性表所有元素的数组

（2）存储线性表元素的数组的长度

（3）存储线性表当前长度的变量

一、顺序存储结构

```
#define LIST_INIT_SIZE    100   //线性表存储空间的初始分配量
#define LISTINCREMENT   10  //线性表存储空间的分配增量
typedef  struct{
      Elemtype    *elem;      //存储空间基址
      int              length;     //当前长度
      int              listsize;    //当前分配存储容量
}SqList;
```

二、顺序存储下的虚拟实现

1、初始化

```
Status InitList_sq(SqList &L)    //构造一个空的线性表
{L.elem=(ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
  if(!L.elem)exit(OVERFLOW);
  L.length=0;  //空表长度为0
  L.listsize=LIST_INIT_SIZE  //初始存储容量
  return OK; }
```

时间复杂性O（1）

2、清空表

```
L.length=0;
```

时间复杂性O（1）

malloc函数和realloc函数

一．malloc函数

【中文名】动态内存分配

【用法】void *malloc(int size);

【功能】malloc向系统申请分配size字节的内存空间，返回类型为void*类型。

【头文件】在Visual C++6.0中可以用malloc.h或者stdlib.h。

【说明】（1）如果分配成功则返回指向被分配内存的指针，否则返回空指针NULL。

（2）当分配的内存空间不再使用时，应使用free()函数将内存块释放。

malloc函数和realloc函数

一．malloc函数

【用法】void *malloc(int size);

【注意】（1）形参是无符号整型（不允许为负数）

（2）函数的返回值是void*类型，它表示未确定类型的指针，即不指向任何类型的数据，只提供一个地址。返回的指针指向该内存分配空间的起始位置。

注意，void*类型可以强制转换为其他任何类型的指针。

【例子】 int *p;

p=(int *)malloc（sizeof（int））；

malloc函数和realloc函数

二．realloc函数

【中文名】动态内存调整

【用法】指针名=(void*)realloc(address，newsize)；

【功能】先判断指针address是否有足够的newsize个连续空间，如果有，扩大address指向的地址，并且将address返回；如果空间不够，先按照newsize指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存区域，而后释放原来address所指内存区域（注意：原来指针是自动释放，不需要使用free），同时返回新分配的内存区域的首地址。即重新分配存储器块的地址。

malloc函数和realloc函数

二．realloc函数

【头文件】在Visual C++6.0中可以用malloc.h或者stdlib.h。

【说明】新申请的内存空间大小一般要大于原来的内存空间大小，否则有可能会导致数据丢失！如果不考虑数据内容是否丢失，新申请的内存空间可比原来的内存空间大也可以小。

【返回值】如果重新分配成功则返回指向被分配内存的指针，否则返回空指针NULL。

【例子】
```
char *p，*q;
p=(char*)malloc(10);
q=(char*)realloc(p,20);
```

二、顺序存储下的虚拟实现

3、销毁表　　free (L.elem)

4、求表长　　return (L.length)

5、判断空表　　if (L.length==0)　return　OK;

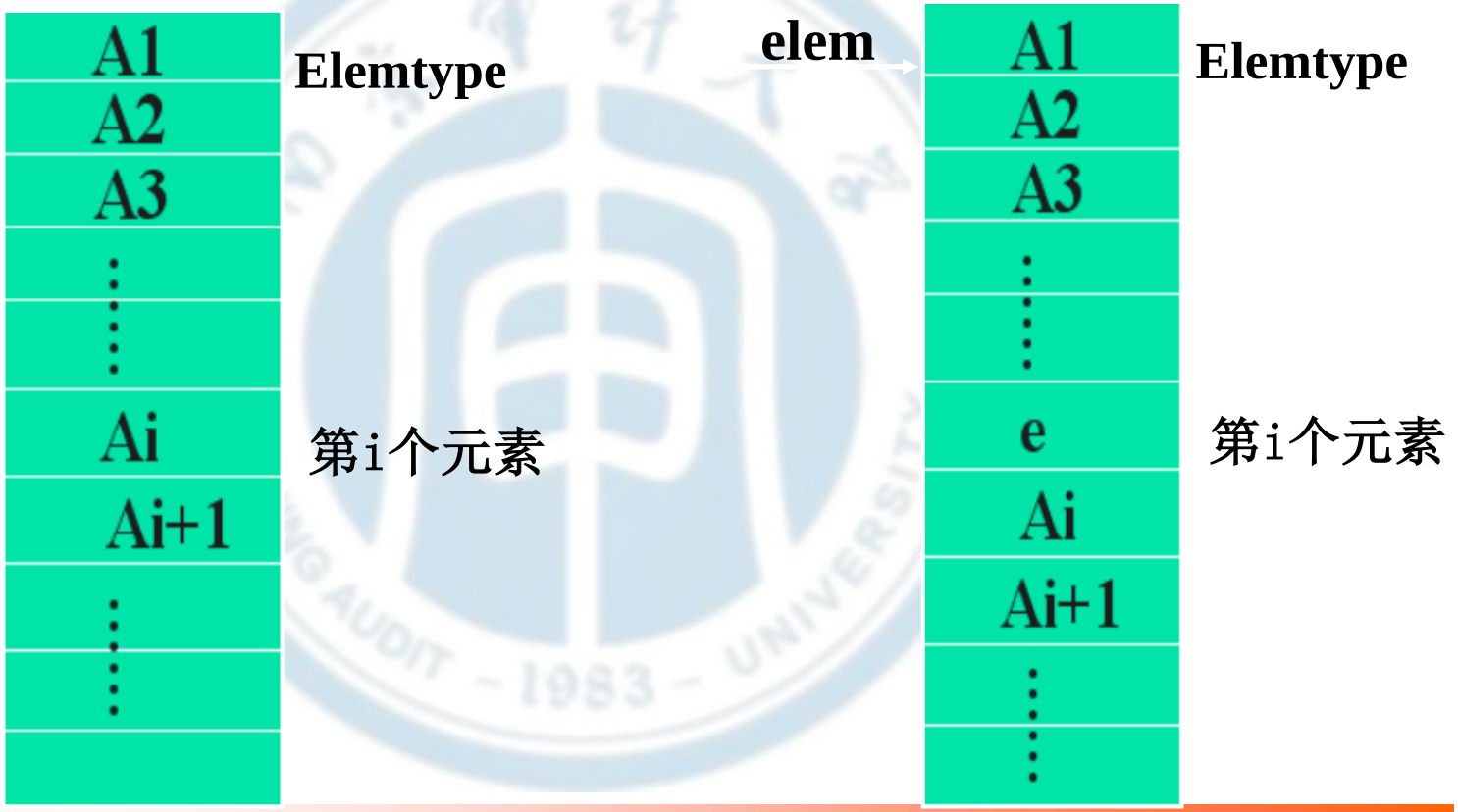
6、取表中第i个元素

```
Status GetElem_sq(SqList L,int i, ElemType &e)
{  if (i>L.length||i<1)  return ERROR;
   p=L.elem+(i-1);  //p为第i个元素的位置
   e=*p;    //取出的第i个元素的值赋给e
   return  OK;
}
```

时间复杂性O（1）

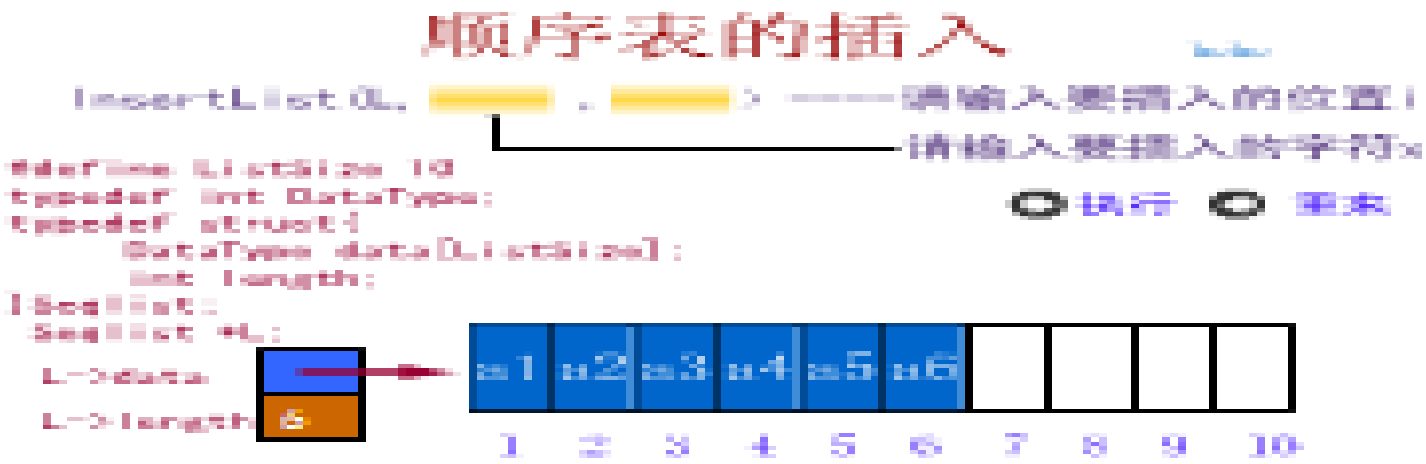
二、顺序存储下的虚拟实现

8、插入 //在顺序线性表L中第i个元素之前插入新元素e



注意：还要考虑空间满的情况

在顺序线性表L中第i个元素之前插入新元素e



在第i个元素之后插入，怎么改？

```
Status Listinsert_Sq(SqList &L,int i, ElemType e) {
        //在顺序线性表L中第i个元素之间插入新元素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);          //存储分配失败
    L.elem=newbase;
    L.listsize+=LISTINCREMENT;
    }
    q=&(L.elem[i-1]);          //q为插入位置
    for (p=&(L.elem[L.length-1]);p>=q;--p)    *(p+1)=*p;
     *q=e; ++L.length;  return OK; }
```

时间复杂性O（n）

考虑平均的情况:

假设在第 i 个元素之前插入的概率为 p_i，则在长度为n 的线性表中**插入一个元素所需移动元素次数的期望值**为：

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

若**假定**在线性表中任何一个位置上进行**插入的概率**都是**相等**的，则**移动元素的期望值为**

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

9、顺序表删除元素

```
Status ListDelete_Sq(SqList &L,int i, ElemType &e) {
        //在顺序线性表L中删除第i个元素，并用e返回其值
        //i的合法值为1<=i<=ListLength_Sq(L)
if (i<1||i>L.length)   return   ERROR;
  p=&(L.elem[i-1]);          //p指向被删除的元素
  e=*p;              //被删除元素的值赋给e
  q=L.elem+L.length-1;
  for (++p;p<=q;++p)      *(p-1)=*p;
  --L.length;
  return OK;
 }                                          时间复杂性O（n）
```

考虑平均的情况:

假设删除第 i 个元素的概率为 q_i，

则在长度为**n** 的线性表中删除一个元素所需移动元素次数的期望值为：

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

若假定在线性表中任何一个位置上进行删除的概率都是相等的，则移动元素的期望值为

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

线性表--顺序存储

特点：以数据元素物理位置的相邻表示逻辑关系的相邻。

优点：随机存取。

缺点：插入删除操作需移动较多数据元素。

顺序存储优点：随机存取，缺点：插入删除操作需移动较多数据元素

作业

指出以下算法中的错误和低效之处，并将它改写为一个既正确又高效的算法。

```
Status DeleteK(SqList &a, int i, int k)
{  //从顺序存储结构的线性表a中删除第i个元素起的k个元素
   if(i<1||k<0||i+k>a.length) return ERROR;//参数不合法
   else {
       for(count=1;count<=k;count++){
       //删除第一个元素
          for(j=a.length;j>=i+1;j--) a.elem[j-1]=a.elem[j];
           a.length--;
        }
   return OK;
}
```

2.3 线性表的链式表示和实现

- 线性表的链式存储结构的特点：
  - 用一组任意的存储单元存储线性表的数据元素。（这组存储单元可以是连续的，也可以是不连续的）
  - 利用指针实现了用不相邻的存储单元存放逻辑上相邻的元素。
  - 每个数据元素a_i，除存储本身信息外，还需存储其直接后继的信息。
  - 数据元素的映象用一个结点来表示。
    - 数据域：元素本身信息
    - 指针域：指示直接后继的存储位置

结点

数据域	指针域

例如：线性表(ZHAO, QIAN, SUN, LI, ZHOU, WU, ZHENG, WANG)

存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25

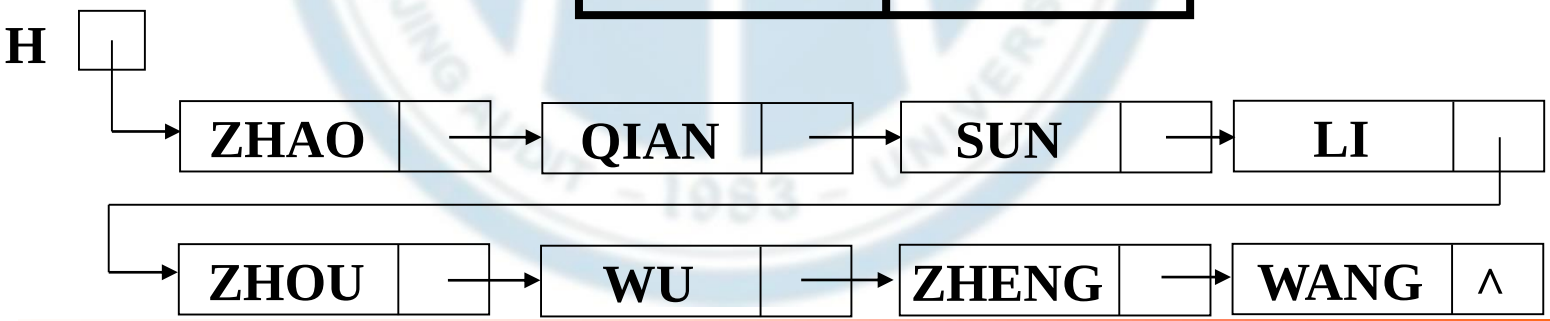
头指针

H

31

数据之间的逻辑关系由结点中的指针指示

最后结点

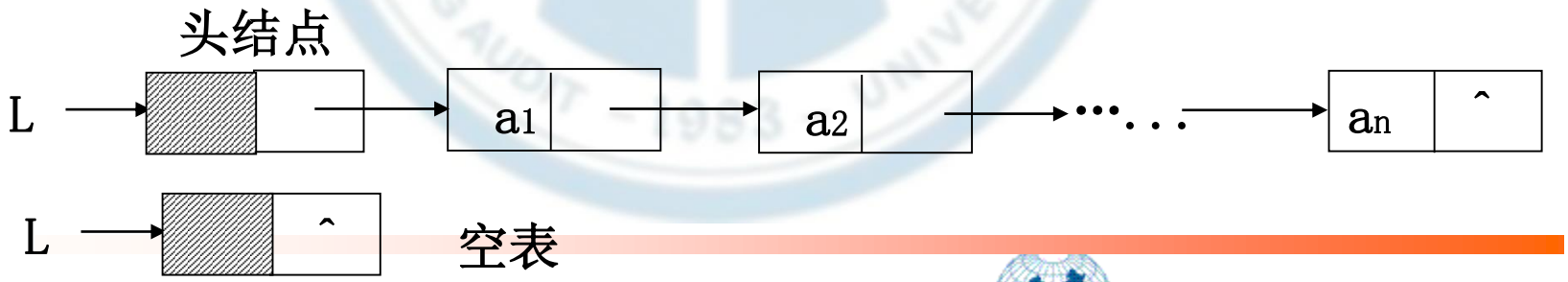


线性链表（单链表）

- 单链表
  - 链表中的每一个结点中只包含一个指针域的称为单链表
- 单链表的存储结构

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

- 头结点：在单链表第一个结点前附设一个结点。

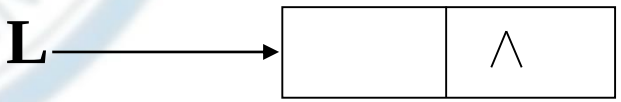


单链表的操作

初始化（带头结点）

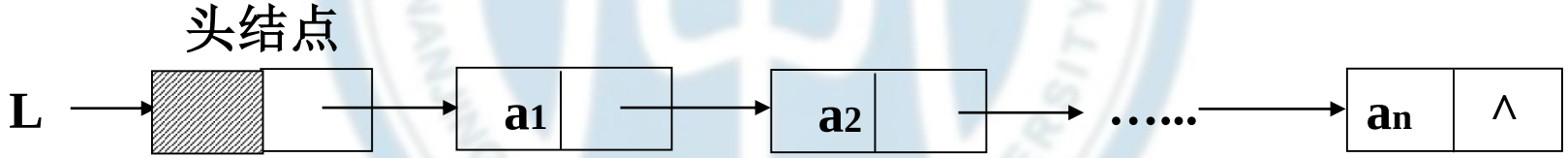
```
status Initlist_L (LinkList  &L)
    {   L=(LNode*) malloc (sizeof(Lnode));

        L->next=NULL;

        return  OK;
    }
```

时间复杂性：0(1)



单链表的操作

- 查找
  - L为带头结点的单链表的头指针，当第i个元素存在时，其值赋给e并返回OK，否则返回ERROR。
  - 算法思想：设置一个指针变量指向第一个结点，然后，让该指针变量逐一向后指向，直到第i个元素。



单链表的操作

- 查找

```
Status GetElem_L(LinkList L, int i, ElemType &e){
    p=L→next; j=1;          //初始化，p指向第一个结点，j为计数器
    while(p && j<i){        //顺指针向后查找，直到p指向第i个元素或p为空
        p=p→next; ++j;
    }
    if(!p || j>i) return ERROR;          //第i个元素不存在
    e=p→data;                            //取第i个元素
    return OK;
}//GetElem_L
```

顺序存取

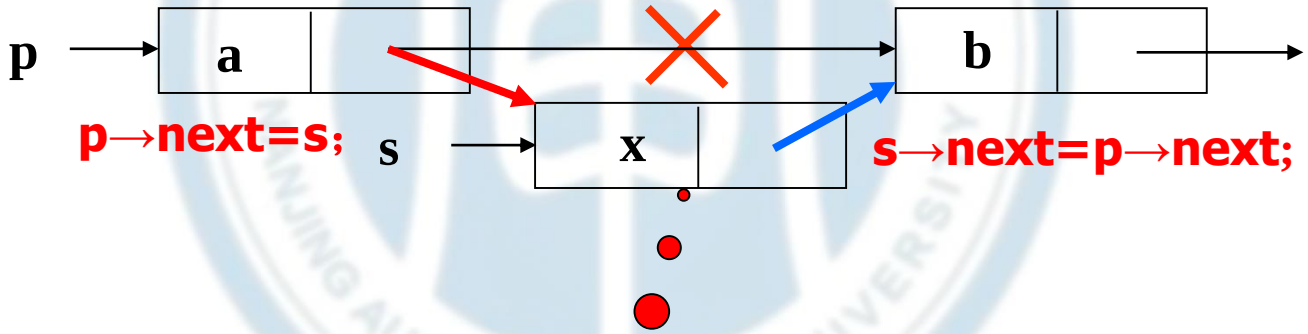
时间复杂性：O(n)

单链表的操作

插入操作：要在数据元素a和b 之间插入元素x

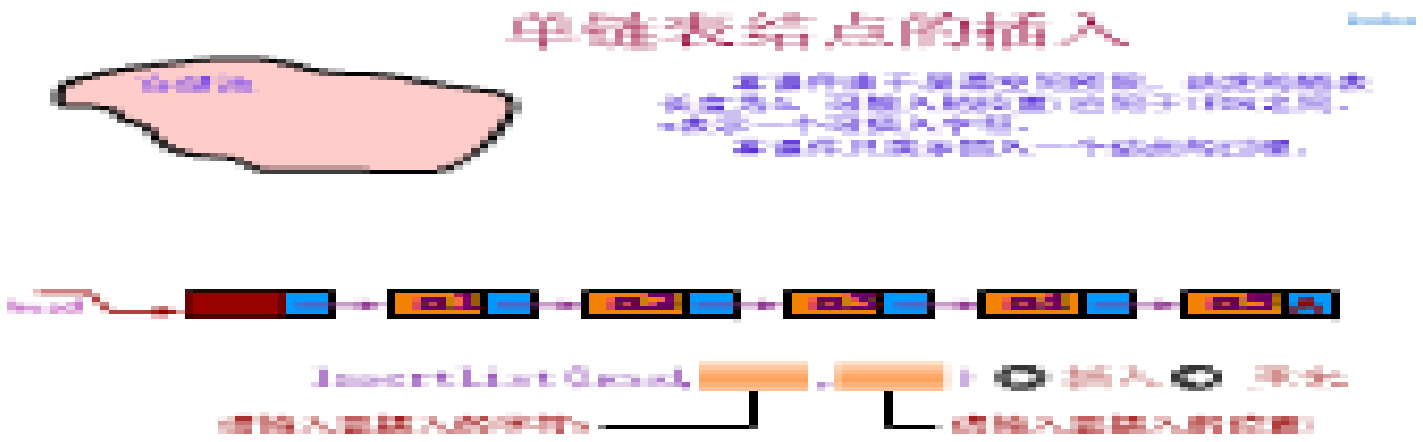
算法思想：

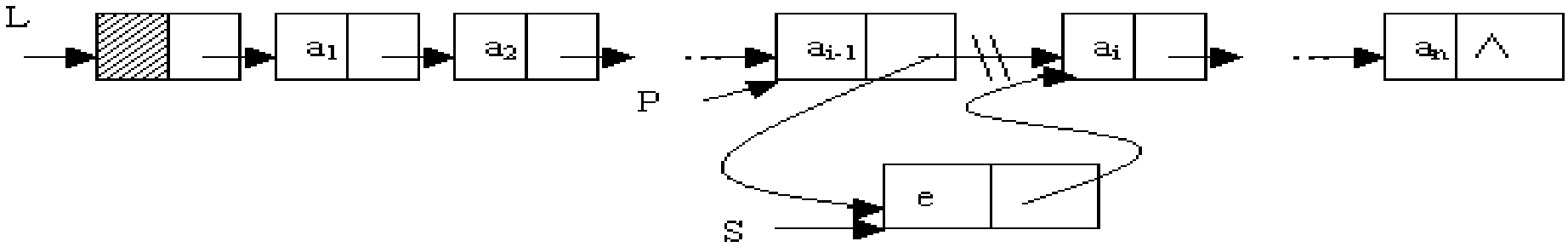
- 生成s结点，数据为x
- s的指针指向b。
- p的指针指向s



可否交换两个指针的修改次序？

- 单链表结点插入





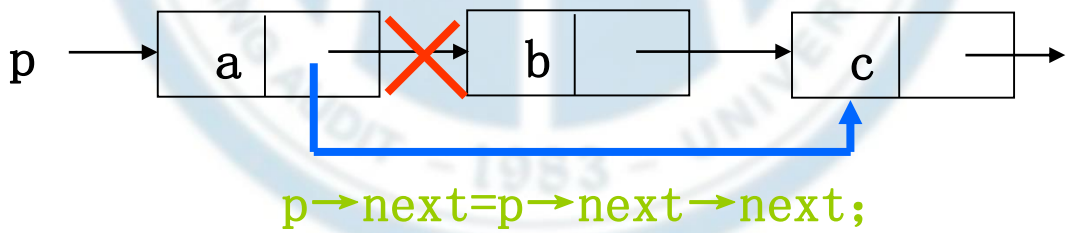

```
Status ListInsert_L(LinkList&L,  int i, ElemType e){
   //在带头结点的单链表L中第i个位置前插入元素e
   p=L;  j=0; //i-1的有效位置从0开始
   while(p && j<i-1) {p=p→next; ++j;}    //寻找第i-1个结点
   if (!p || j>i-1) return ERROR;  //i小于1或者大于表长加1
   s=(LinkList)malloc(sizeof(LNode));
   s→data=e;                                          //生成新结点
   s→next=p-next; p→next=s;                  //插入L中
   return OK;
}// ListInsert_L
```

单链表的操作

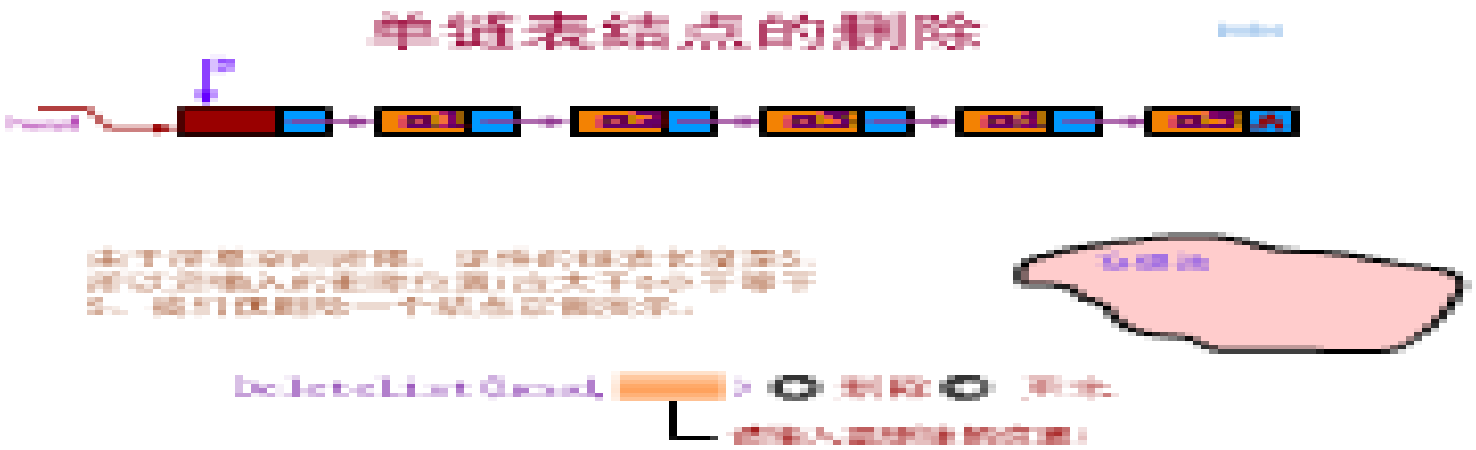
删除操作：在单链表数据元素a、b、c三个相邻的元素中删除b

算法思想：

- 就是要让a的指针直接指向c，使b从链表中脱离。
- 释放b所分配的资源。

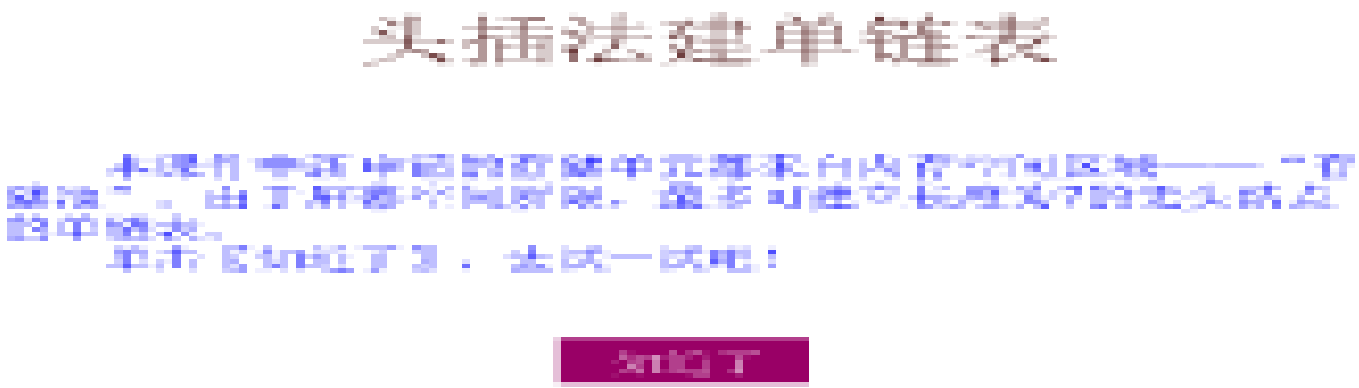


- 单链表结点删除




```
Status ListDelete_L(LinkList&L,  int i, ElemType &e){
   //在带头结点的单链表L中，删除第i个元算，并由e返回其值
   p=L;  j=0; //i-1的有效位置从0开始
   while(p->next && j<i-1) {  //寻找第i个结点，并令p指向其前趋
         p=p→next; ++j;
   }
   if (!(p->next) || j>i-1) return ERROR; //删除位置不合理
   q=p->next; p->next=q->next;              //删除结点
   e=q->data; free(q);                      //释放结点
   return OK;
}// ListDelete_L
```

单链表的操作:头插法建立单链表



先进后出

单链表的操作:头插法建立单链表

- 建立线性表的链式存储结构的过程就是一个动态生成链表的过程
- 从空表开始，依次建立各个元素结点，并逐个插入链表

```
void  CreateList_L(LinkList &L,  int n){
  //逆位序输入n个元素的值，建立带表头结点的单链表L。
  L=(LinkList) malloc (sizeof(LNode));
  L->next=NULL;                          //建一个带头结点的单链表
  for (i=n; i>0 ; --i ) {
        p= (LinkList) malloc (sizeof(LNode));     //生成新结点
        scanf(&p->data);                                  //输入元素值
        p->next=L->next; L->next =p;                //插入到表头
  }
}// CreateList_L
```

两个指针

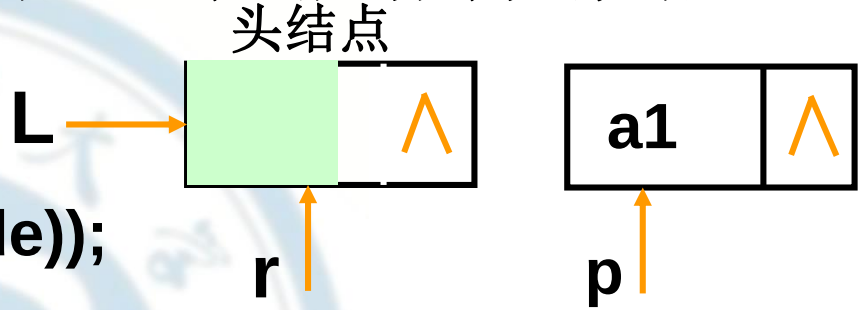
这是一个逆序建立

单链表的操作:尾插法建立单链表

单链表的操作:尾插法建立单链表

方法：从头到尾，即从第一个元素结点逐个创建各个元素结点。每次都是链到当前链表的最后。

创建头结点：




```
L＝(LinkList)malloc(sizeof(LNode));
L->next=NULL; r=L;
```

读入一个元素，链入其中：

```
p＝(LinkList)malloc(sizeof(LNode));
    scanf(&p->data);
    p->next=NULL;
    r->next=p; r=r->next;
```

先进先出，正序建立，三根指针

重复n次

这是一个正序建立

单链式存储结构下的其他操作

求长度

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

时间复杂性：O(n)

单链式存储结构下的其他操作

逆转：将一单链表逆转，要求逆转在原链表上进行，不允许重新构造一个链表。(就地逆转)

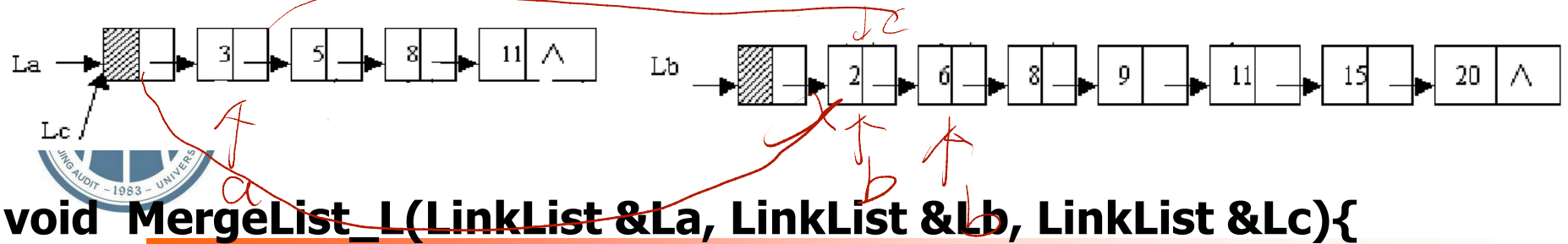
```
void reverse (LinkList  &head)
{ p=head;
   q=p->next;
   while(q!=NULL)      /*当L没有后继结点时终止*/
   {  r=q->next;  q->next=p;  p=q;  q=r;  }
head->next=NULL;
head=p;     /*p指向L的最后一个结点，现改为头结点*/
}
```

三根指针，pq用来交换，为了防止丢失下一个节点地址，用r记录

单链式存储结构下的其他操作

合并：将两个有序链表合并为一个有序链表

- 算法思想：
  - 设立三个指针pa、pb和pc分别用来指向两个有序链表和合并表的当前元素。
  - 比较两个表的当前元素的大小，将小的元素链接到合并表中，即，让合并表的当前指针指向该元素，然后，修改指针。
  - 在归并两个链表为一个链表时，不需要另建新表的结点空间，而只需将原来两个链表中结点之间的关系解除，重新建立关系。



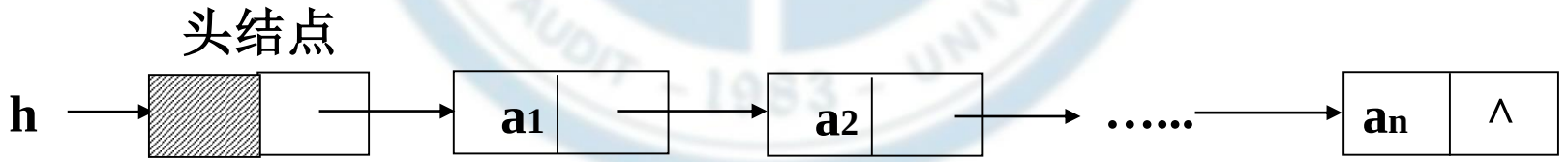

```
void  MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc){
//已知单链表La和Lb的元素按非递减排列，归并La和Lb得到新的非递减单链表 Lc 。
    pa=La->next; pb=Lb->next;
    Lc=pc=La; //用La的头结点作为Lc的头结点
    while (pa && pb) {
          if (pa->data<=pb->data){
                pc->next=pa; pc=pa; pa=pa->next;
          }
        else{pc->next=pb; pc=pb; pb=pb->next; }
    }
    pc->next=pa?pa:pb; //插入剩余段
    free(Lb); //释放Lb的头结点
}// MergeList_L
```

单链表特点

- 它是一种动态结构，整个存储空间为多个链表共用。
- 不需预先分配空间。
- 插入、删除操作的速度快。
- 指针占用额外存储空间。
- 不能随机存取，查找速度慢。
- 只能沿一个方向查找节点

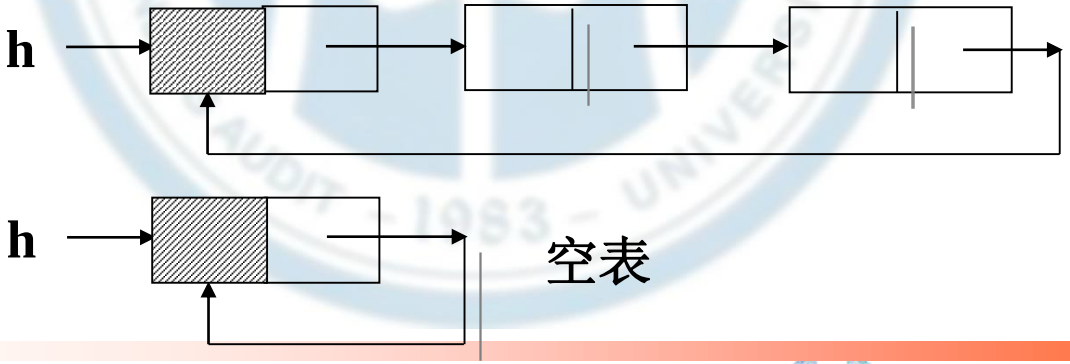
单链表的改进

- 它是一种动态结构，整个存储空间为多个链表共用。
- 不需预先分配空间。
- 插入、删除操作的速度快。
- 指针占用额外存储空间。
- 不能随机存取，查找速度慢。
- 查找受限，从某点出发，只能查找后面的节点

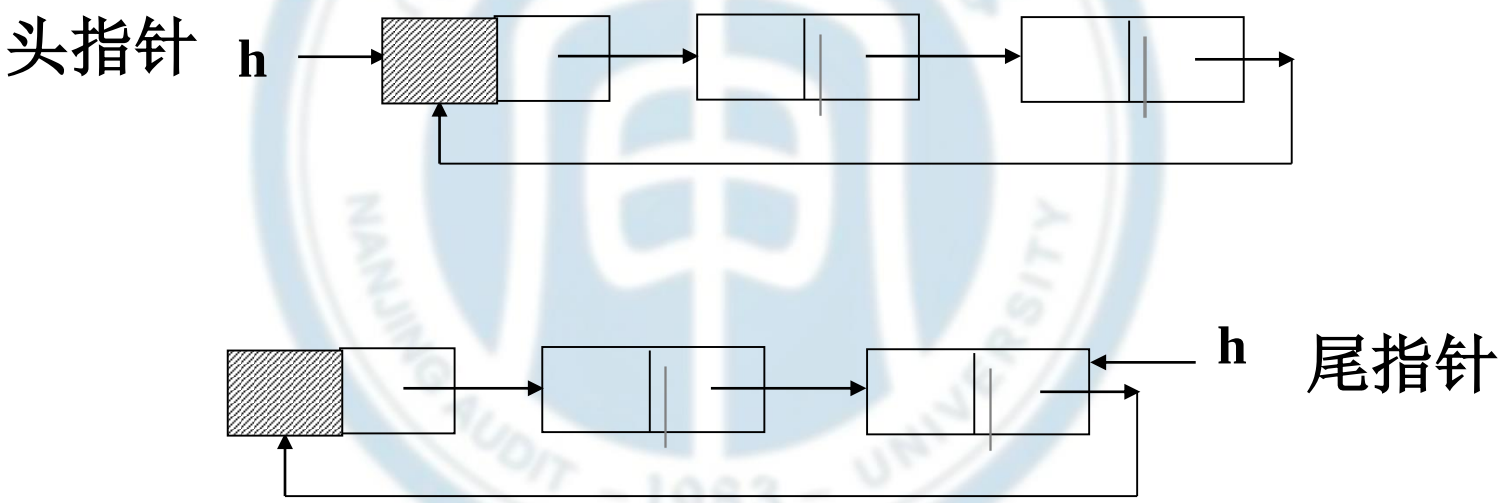


2.3.2 循环链表

- 表中最后一个结点的指针指向头结点，使链表构成环状。
- 特点：从表中任一结点出发均可找到表中其他结点，提高查找效率。
- 操作与单链表基本一致，判表尾条件不同。
  - 单链表p或p->next=NULL
  - 循环链表p或p->next=h



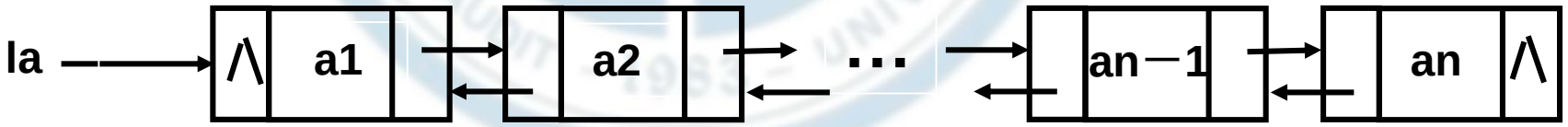
- 在循环链表中设计尾指针而不设头指针，可以使某些操作简单。



2.3.3 双向链表

双链式存储结构

（a1,a2,…ai,ai+1,…an）



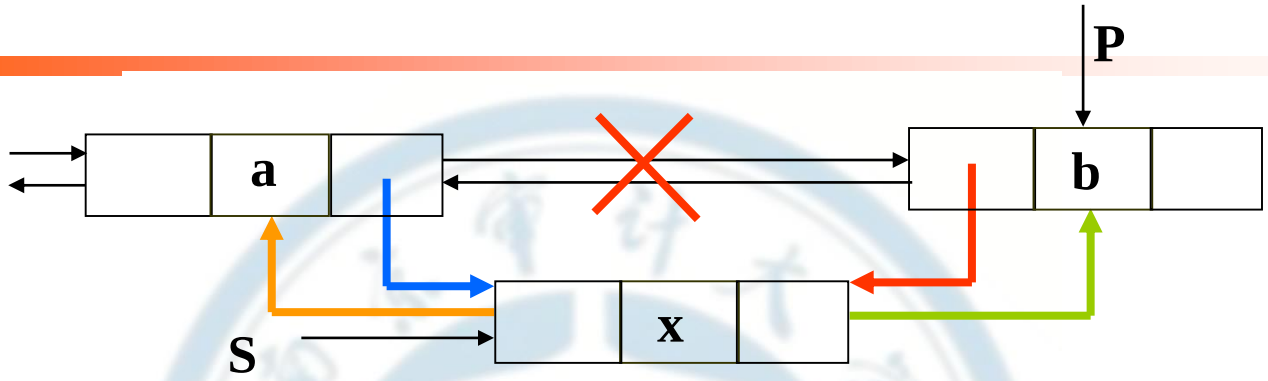
```
//---线性表的双链式存储结构----
typedef   struct  DuLNode{
        ElemType  data;
        struct DuLNode *next;
        struct DuLNode *prior;
        }  DuLNode,*DuLinkList;
```

双向链表的操作

- 双指针使得双向循环链表中，前趋结点和后继结点的查找更为方便、快捷。NextElem和PriorElem的执行时间为O(1)。
- 仅需涉及一个方向的指针的操作和单链表的操作相同。
- 插入和删除需同时修改两个方向的指针。

插入操作：

要在数据元素a和b 之间插入元素x



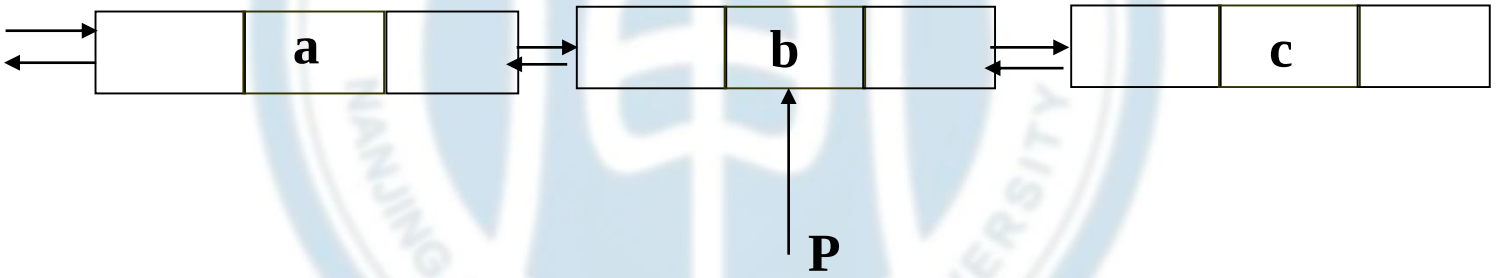

```
s=(DuLinkList)malloc(sizeof(DuLNode));
s->data=x;
s->prior=p->prior;
p->prior->next=s;
s->next=p;
p->prior=s;
```

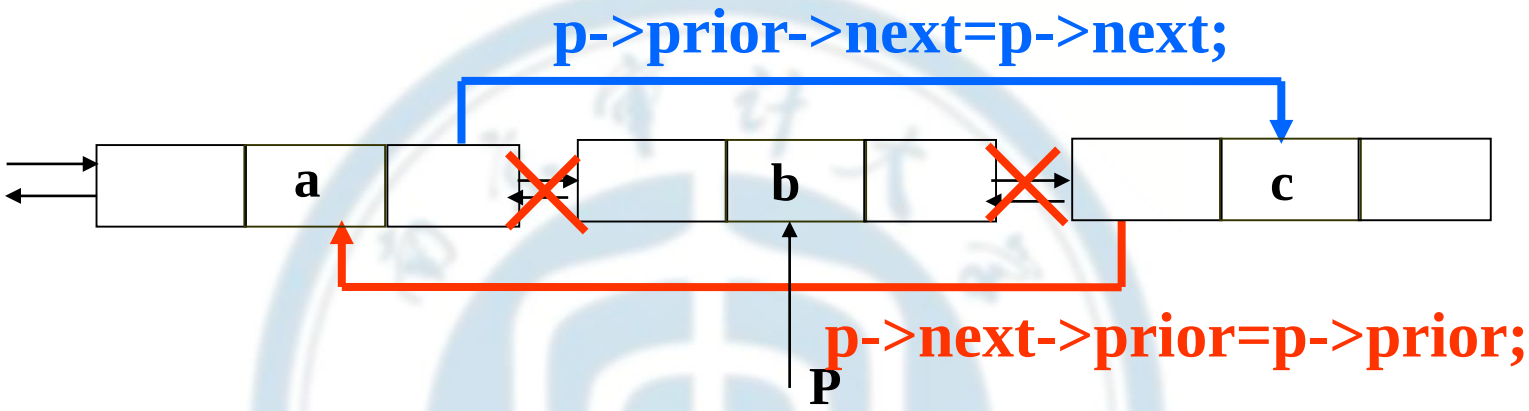
思考：四个指针的修改顺序可否任意改变？

插入算法2. 18

```
Status ListInsert_DuL(DuLinkList&L,  int i, ElemType e){
    //在带头结点的双向循环链表L中第i个位置前插入元素e
    //i的合法值为1≤i≤表长+1
    if (!(p=GetElemP_DuL(L,i)))   //在L中确定第i个元素的指针p
        return ERROR;                          //p=NULL,即第i个元素不存在
    if(!(s=(DuLinkList)malloc(sizeof(DuLNode))))
        return ERROR;                                  //申请空间失败
    s->data=e;                                 //生成新结点
    s->prior=p->prior;  p->prior->next=s;
    s->next=p;  p->prior=s;                            //插入L中
    return OK;
}// ListInsert_DuL
```

删除操作：
在单链表数据元素a、b、c三个相邻的元素中删除b





free(p);

删除算法2. 19

```
Status ListDelete_DuL(DuLinkList&L,  int i, ElemType &e){
  //在带头结点的双向循环链表L中，删除第i个元算，并由e返回其值
  //i的合法值为1≤i≤表长
  if (!(p=GetElemP_DuL(L,i)))  //在L中确定第i个元素的指针p
        return ERROR;                    //p=NULL,即第i个元素不存在
  e=p->data;
  p->prior->next=p->next;
  p->next->prior=p->prior;               //删除结点
  free(p);                                 //释放结点
  return OK;
}// ListDelete_DuL
```

双向链表的操作

删除p的直接后继结点的语句序列

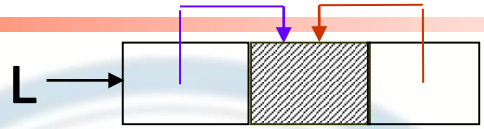
```
q = p->next;
p->next = q->next;
q->next->prior = p;
free(q);
```

删除p的直接前驱结点的语句序列

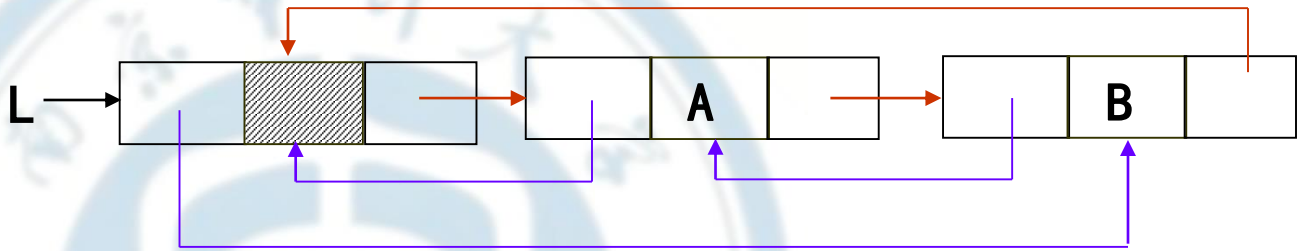
```
q = p->prior;
p->prior = q->prior;
q-> prior->next = p;
free(q);
```

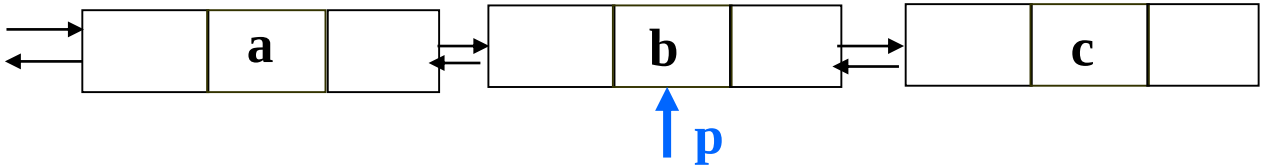
双向循环链表

空双向循环链表：



非空双向循环链表：





p->prior->next= p= p->next->proir;

双向循环链表

2、特点：

☆ 相当于两个循环单链

3、实现：

与双向链表完全相同

练习

不带头结点的单链表head为空的判定条件是__A__。

A. head= =NULL

B. head—＞next= =NULL

C. head—＞next= =head

D. head!=NULL

带头结点的单链表head为空的判定条件是__B__。

A. head= =NULL

B. head—＞next= =NULL

C. head—＞next= =head

D. head!=NULL

非空的循环单链表head的尾结点（由p所指向）满足__C__。

A. p—＞next= =NULL

B. p= =NULL

C. p—＞next= =head

D. p= =head

在循环双链表的p所指结点之后插入s所指结点的操作是__D__。

- A. **p->next** =s; s->**prior**=p; **p->next->prior**=s; s ->**next** = **p->next;**
- B. **p->next** =s; **p->next->prior**=s; s->**prior**=p; s->**next** =**p->next;**
- C. s->**priort**=p; s->**next** =**p->next;** **p->next** =s; **p->next->prior**=s;
- D. s->**prior**=p; s->**next** = **p->next;** **p->next->prior**=s; **p->next** =s;

本章小结（一）

- 线性表是n(n≥0)个数据元素的序列，通常写成

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

- 线性表中除了第一个和最后一个元素之外，都只有一个前驱和一个后继。
- 线性表中每个元素都有自己确定的位置，即“位序”。
- n=0时的线性表称为“空表”，在写线性表的操作算法时一定要考虑你的算法对空表的情况是否也正确。

本章小结（二）

- 顺序表
  - 是线性表的顺序存储结构的一种别称。
  - 特点是以“存储位置相邻”表示两个元素之间的前驱、后继关系。
  - 优点是可以随机存取表中任意一个元素。
  - 缺点是每作一次插入或删除操作时，平均来说必须移动表中一半元素。
  - 常应用于主要是为查询而很少作插入和删除操作，表长变化不大的线性表。

本章小结（三）

- 链表
  - 是线性表的链式存储结构的别称。
  - 特点是以“指针”指示后继元素，因此线性表的元素可以存储在存储器中任意一组存储单元中。
  - 优点是便于进行插入和删除操作。
  - 缺点是不能进行随机存取，每个元素的存储位置都存放在其前驱元素的指针域中，为取得表中任意一个数据元素都必须从第一个数据元素起查询。
  - 由于它是一种动态分配的结构，结点的存储空间可以随用随取，并在删除结点时随时释放，以便系统资源更有效地被利用。这对编制大型软件非常重要，作为一个程序员在编制程序时必须养成这种习惯。