第二章 线性表

本章目录

主要内容

- **知识点**
  - 线性表的定义
  - 顺序表
  - 单链表
  - 循环链表
  - 双链表
- **重点难点**
  - **顺序表操作的实现**
  - **单链表操作的实现**
  - **顺序表和链表操作时间复杂度的分析**

线性结构

线性结构特点：在数据元素的非空有限集中

- 存在唯一的一个被称作“第一个”的数据元素
- 存在唯一的一个被称作“最后一个”的数据元素
- 除第一个外，集合中的每个数据元素均只有一个前驱
- 除最后一个外，集合中的每个数据元素均只有一个后继

2.1线性表(Linear List)定义

- 定义： n个具有相同特性的数据元素组成的有限序列；
  - 表示：$\{a_1,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n\}$
  - a_i必须具有相同特性，即属于同一数据对象
  - a_{i-1}是a_i的直接前驱元素，a_{i+1}是a_i的直接后继元素
  - 数据元素a_i在线性表中有确定的位置i，i称为位序
  - 线性表中数据元素的个数n称为线性表的长度,n=0时，线性表称为空表

如 $(a_1, a_2, \ldots\ldots, a_i, \ldots\ldots a_n)$

例　英文字母表（A,B,C,…..Z)是一个线性表

例

学号	姓名	年龄
001	张三	18
002	李四	19
……	……	……

数据元素

- 元素个数n—表长度，n=0空表
- 1<i<n时
  - a_i的直接前驱是a_{i-1}, a_1无直接前驱
  - a_i的直接后继是a_{i+1}, a_n无直接后继
- 元素同构，且不能出现缺项

线性表的抽象数据类型：

ADT List
{
数据对象：D={ai|ai∈ ElemSet,i=1,2,...,n,n≥0}
数据关系：R={<ai-1,ai>|ai,ai-1∈ D,i=2,...,n}
基本操作：
ListInit (&L); //线性表初始化
int ListLength(L); //求线性表长度
ElemType ListGet(L,i) ; //取表元
int ListLocate(L,x) ; //按值查找
ListClear(&L) ; //清空线性表
int ListEmpty(L) ; //判空线性表
ListInsert(&L,i,e) ; //插入
ListDelete(&L,i) ; //删除
ElemType Listprior(L,e) ; //前驱
ElemType ListNext(L,e) ; //后继
ListPrint(L);//遍历
}ADT List

2.2 线性表的顺序存储结构

- 顺序表：
  - 定义：把线性表的结点（元素）按其逻辑次序依次存入一组地址连续的存储单元里，用这种方法存储的线性表称为顺序表。
  - 特点：
    - 以元素在计算机内存中的“物理位置相邻”来表示线性表中数据元素之间的逻辑关系。
    - 只要确定了首地址，线性表中任意数据元素都可以随机存取。

- 元素地址计算方法：
  - $LOC(a_i)=LOC(a_1)+(i-1)*L$
  - $LOC(a_{i+1})=LOC(a_i)+L$
  - 其中：
    - L—一个元素占用的存储单元个数
    - $LOC(a_i)$—线性表第i个元素的地址
- 优点：
  - 实现逻辑上相邻—物理地址相邻
  - 实现随机存取
- 实现：可用C语言的一维数组实现

顺序表的类型定义

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

顺序表基本运算的实现

- **1 顺序表的初始化**
  构造一个空的顺序表，设置length域为0。
- Status InitList(SqList &L) //初始化
- {L.elem=(ElemType*)malloc(LIST_INIT_SIZE*
  sizeof(ElemType));
- if(!L.elem) printf("no succeed!");
- L.length=0;
- L.listsize=LIST_INIT_SIZE;
- return 1;
- }

- 2 插入操作
  - 定义：线性表的插入是指在第i个位置上插入一个新的数据元素x，使长度为n的线性表

$$(a_1, a_2, \ldots\ldots, a_{i-1}, a_i, \ldots\ldots a_n)$$

变成长度为n+1的线性表

$$(a_1, a_2, \ldots\ldots, a_{i-1}, x, a_i, \ldots\ldots a_n)$$

需将第i至第n共（n-i+1)个元素后移

顺序表上基本运算的实现

- 插入运算:在第 i 个位置，插入元素e
- 思想：把从第i个位置开始的元素，依次后移
- 步骤：
  - 1.当前表是否已经满？
  - 2.输入是否有效？
  - 3.依次后移，插入元素
  - 4.长度加1。

```
Status ListInsert_Sq(SqList &L, int i, ElemType e)   //插入
{
ElemType *newbase;  int j;
if(i<1||i>L.length+1)  printf("i is error");
if(L.length>=L.listsize)
{   newbase=(ElemType *)realloc(L.elem,(L.listsize+
    LISTINCREMENT)*sizeof(ElemType));
    if(!newbase) printf("no realloc");
    L.elem=newbase;
    L.listsize+=LISTINCREMENT;
}
for(j=L.length-1;j>=i-1;j--)
   L.elem[j+1]=L.elem[j];
L.elem[i-1]=e;
L.length++;
return 1;
}
```

- 算法时间复杂度T(n)
  - 设P_i是在第i个位置插入一个元素的概率，则在长度为n的线性表中插入一个元素时，所需移动的元素次数的平均次数为：

$$Eis = \sum_{i=1}^{n+1} P_i (n - i + 1)$$

若认为 $P_i = \frac{1}{n+1}$

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

$\therefore T(n) = O(n)$

- 3 删除操作
  - 定义：线性表的删除是指将第i（1≤i ≤ n）个元素删除，使长度为n的线性表

$$(a_1, a_2, \ldots\ldots, a_{i-1}, a_i, \ldots\ldots a_n)$$

变成长度为n-1的线性表

$$(a_1, a_2, \ldots\ldots, a_{i-1}, a_{i+1}, \ldots\ldots a_n)$$

需将第i+1至第n共（n-i)个元素前移

顺序表上基本运算的实现(3)

- 删除运算:删除第 i 个元素，用e返回删除元素值
- 思想：把第i＋1个位置开始的元素，依次前移
- 步骤：
  - 1.要检查删除位置的有效性；
  - 2.用e返回删除元素，依次移动元素；
  - 3.长度减1。

```
void ListDelete_Sq(SqList &L, int i, ElemType &e)
{   int j;
   if(i<1||i>L.length)  printf("i is error");

    e=L.elem[i-1];

   for(j=i;j<=L.length-1;j++)
       L.elem[j-1]=L.elem[j];

   L.length--;
}
```

- 删除操作的时间复杂度
  - 设Q_i是删除第i个元素的概率，则在长度为n的线性表中删除一个元素所需移动的元素次数的平均次数为：

$$E_{de} = \sum_{i=1}^{n} Q_i (n - i)$$

若认为 $Q_i = \frac{1}{n}$

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

$\therefore T(n) = O(n)$

•故在顺序表中插入或删除一个元素时，平均移动表的一半元素，当n很大时，效率很低

4 按值查找

- 在L中查找值为x的元素，若有则返回该元素下标，否则返回0。
- int SeqListLocate(SqList L,ElemType x)
- {

```
int i;
i = 1;
while(i<=L.length&&L.data[i-1]!=x)
  i++;
if(i<=L.length) return i;
else return 0;
```

- }

5 输出

```
void disp(SqList L)   //显示或遍历
{    int i;
     if(L.length==0) printf("no element\n");
     for(i=0;i<=L.length-1;i++)
          printf("%d\n",L.elem[i]);
}
```

6 实现顺序表的就地逆置

```
void reverse(SqList &L)
{
    ElemType t; int i;
    for(i=0; i<L.length/2; i++)
    {
        t=L.elem[i];
        L.elem[i]=L.elem[L.length-1-i];
        L.elem[L.length-1-i]=t;
    }
}
```

7 有序顺序表的合并

- 已知两个非递减的有序表La和Lb，求La和Lb合并后的有序表Lc。
- 例如：

  已知：

  La={1，4，5，7，19}

  Lb={2，4，6，7，15，16}

  则La和Lb的合并结果Lc为：

  Lc={1, 2, 4, 4, 5, 6, 7, 7, 15, 16, 19}

```
void MergeList(SqList La, SqList Lb, SqList & Lc)
{       int i=0,j=0,k=0;
        InitList(Lc);

      while((i<La.length)&&(j<Lb.length))
        if(La.elem[i]<Lb.elem[j])
                Lc.elem[k++]=La.elem[i++];
        else
                Lc.elem[k++]=Lb.elem[j++];

       while(i<La.length)
          Lc.elem[k++]=La.elem[i++];
       while(j<Lb.length)
          Lc.elem[k++]=Lb.elem[j++];

        Lc.length=k;
}       时间复杂度：O(La.length+Lb.length)
```

8 有序顺序表表示集合，关于集合的操作---集合并

- 已知两个非递减的有序表La和Lb，La和Lb分别表示两个集合，求La和Lb的并集Lc。
- 例如：

  已知：

  La={1，4，5，7，19}

  Lb={2，4，6，7，15，16}

  则La和Lb的并集结果Lc为：

  Lc={1, 2, 4, 5, 6, 7, 15, 16, 19}

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

9 有序顺序表表示集合，关于集合的操作---集合交

- 已知两个非递减的有序表La和Lb，La和Lb分别表示两个集合，求La和Lb的并集Lc。
- 例如：

  已知：

  La={1，4，5，7，19}

  Lb={2，4，6，7，15，16}

  则La和Lb的交集结果Lc为：

  Lc={4, 7}

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

顺序存储结构的优缺点

- 优点
  - 逻辑相邻，物理相邻
  - 可随机存取任一元素
  - 存储空间使用紧凑
- 缺点
  - 插入、删除操作需要移动大量的元素

思考题

- 1、 判断顺序表中元素是否对称，对称返回1，否则返回0。
- 2、实现把顺序表中所有奇数排在偶数之前，即表的前面为奇数，后面为偶数。
- 3、输入整型元素序列利用有序表插入算法建立一个有序表。

2.3 线性表的链式表示和实现

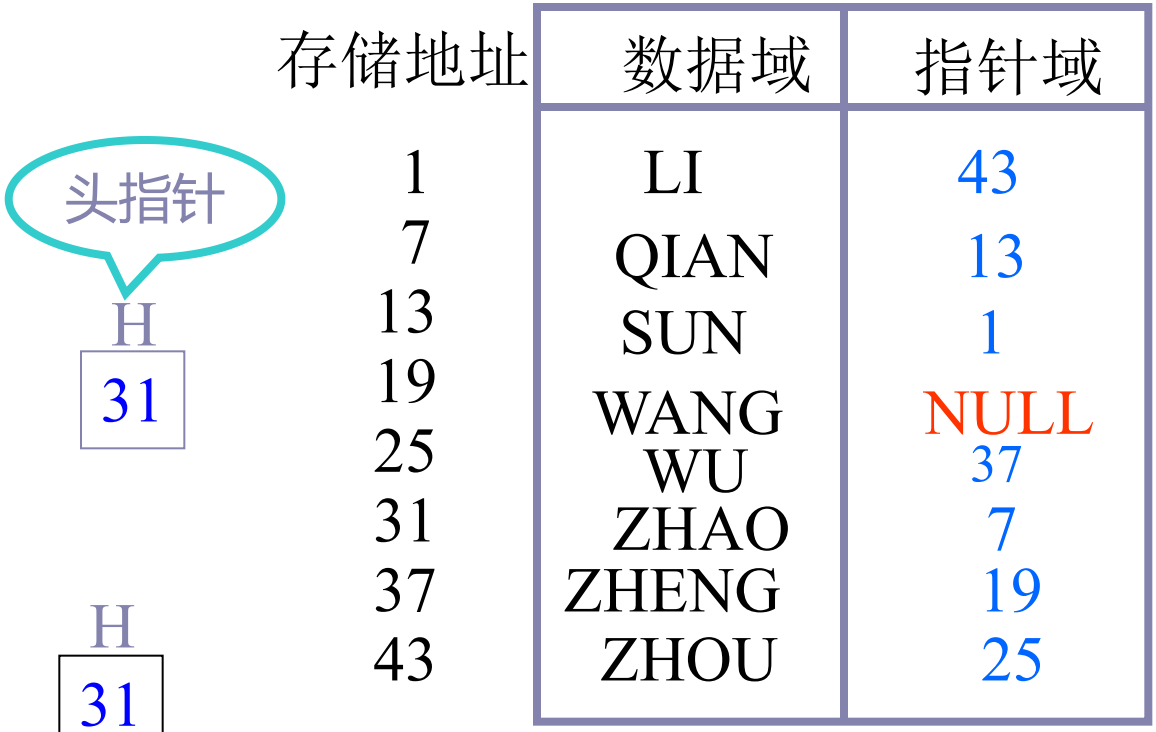
以链式结构存储的线性表称之为线性链表。线性表中的数据元素可以用任意的存储单元来存储，逻辑相邻的两元素的存储空间可以是不连续的。为表示逻辑上的顺序关系，对表的每个数据元素除存储本身的信息之外，还需存储其后继的地址（即用指针表示逻辑关系）。这两部分信息组成数据元素的存储映象，称为结点。

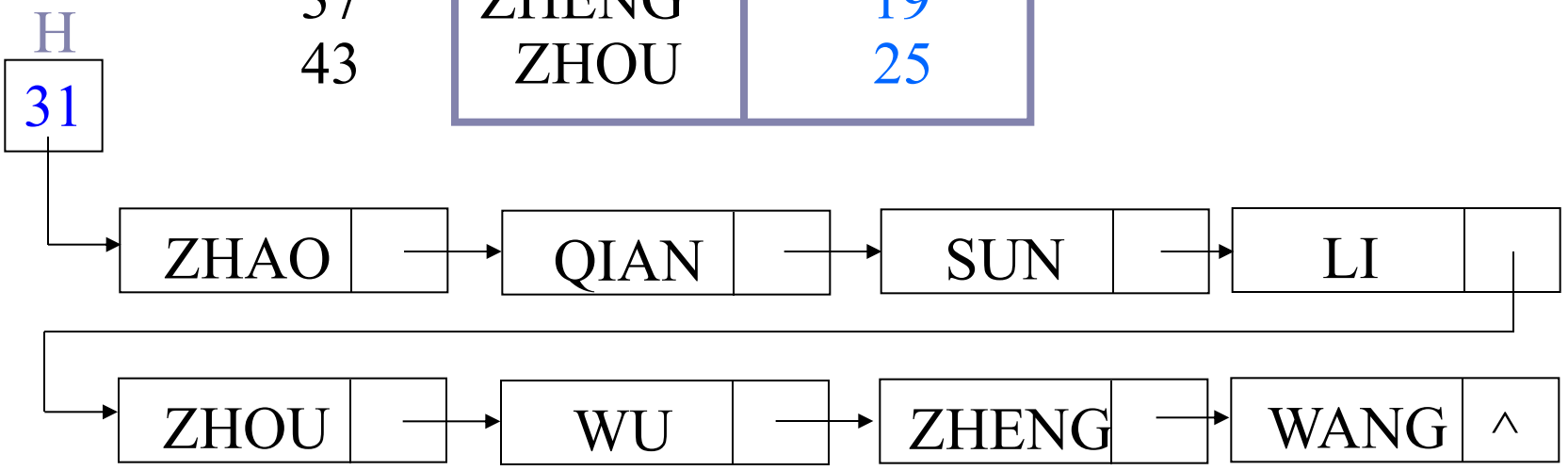
单链表结构图示

- 结点a_i

- 链表

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





- 单链表结点的类型定义

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

LinkList   h,p;
```

结点（*p）

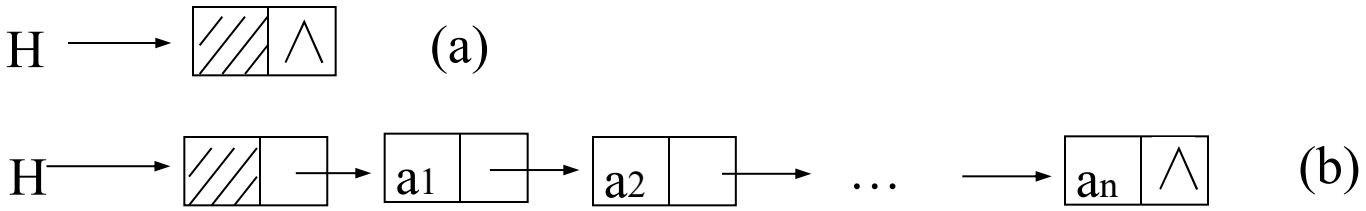
(*p)表示p所指向的结点
(*p).data⇔p->data表示p指向结点的数据域
(*p).next⇔p->next表示p指向结点的指针域

指针变量使用前需要分配内存空间
p=(LinkList)malloc(sizeof(LNode));

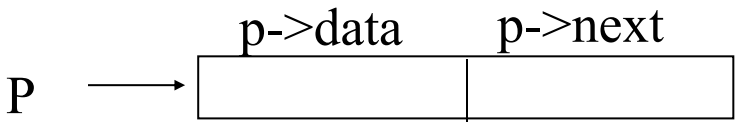
h=NULL;表示h是空指针。

带头结点的单链表

- 通常情况下，为了运算的统一，常在第一个结点前附设一个结点，称为“头结点”，头指针具有标识作用，因而，常用作链表的名字



LinkList p;
p=(LinkList)malloc(sizeof(LNode));
则完成了申请一块LNode类型的存储单元的操作，并将其地址赋值给变量p。
释放结点：free(p)



单链表操作的实现(1)

- 带头结点的单链表的初始化：
  - int LinkedListInit(LinkList &L)
  - {//建立一个空的单链表
  - L=(LinkList)malloc(sizeof(LNode));
  - if (L==NULL)
  - {printf(“无内存空间可分配”);retutn 0;}
  - L->next=NULL;
  - return 1;
  - }

单链表操作的实现(2)

求表长:

```
int  LinkedListLength(LinkList L)
{//求带头结点的单链表的长度
LinkList p; int j;
p=L->next;      //p指向第一结点
 j=0;
 while(p!=NULL)
  {j++;p=p->next; } //移动p指向下一结点
 return  j;
}
```

单链表操作的实现(3)

- 取第i个元素:
  - int LinkListGet(LinkList L, int i, ElemType &e)
  - {//在单链表L中查找第i个元素结点，用e返回
  - LinkList p; int j;
  - p=L->next;　　j=1;
  - while (p!=NULL && j<i)
  - {j++; p=p->next;}
  - if(!p||j>i) return 0; //第i个元素不存在
  - e=p->data;
  - return 1;
  - }

单链表操作的实现(4)

- 按值查找:
  - LinkList LinkedListLocate(LinkList L,ElemType x)
  - {//在带头结点的单链表L中查找值为x的结点，找到后返回其指针，否则返回空
  - LinkList p;
  - p=L->next;
  - while(p!=NULL && p->data!=x)
  - p=p->next;
  - if(!p)　{printf(“无值为X的结点”);return NULL;}
  - else return p;
  - }

单链表操作的实现(5)

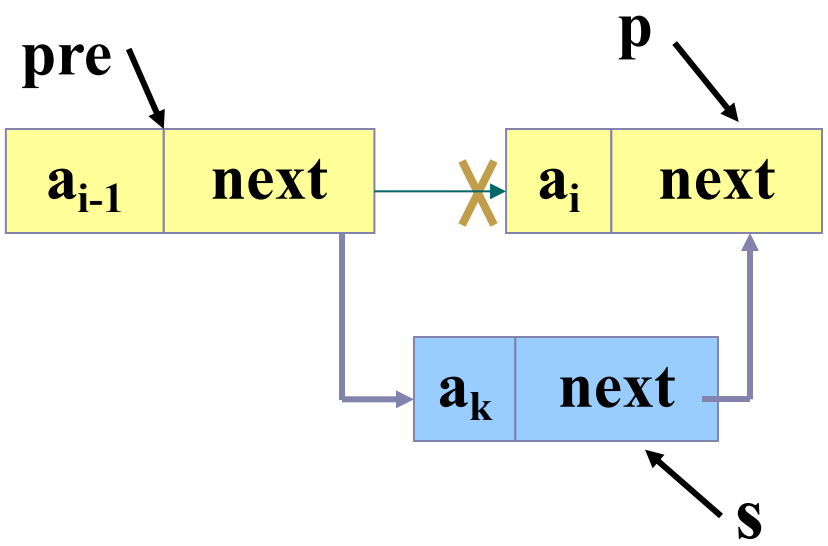
- 查找p结点的前驱:
  - LinkList LinkedListLocate(LinkList L, LinkList p)
  - {//在单链表L中求p指向的结点的前驱
  - LinkList pre;
  - if(L->next==p)
  - {printf(“p指向第一元素结点，无前驱”);return NULL;}
  - pre=L->next;
  - while(pre!=NULL && pre->next!=p)
  - pre=pre->next;
  - return pre;
  - }

单链表操作的实现(6)

- 查找值为e的结点的后继:
  - LinkList LinkedListLocate(LinkList L, ElemType e)
  - {//在单链表L中求元素值为e的结点的后继
  - LNode *p;
  - p=L->next;
  - while(p!=NULL && p->data!=e)
  - p=p->next;
  - if(p==NULL)
  - {printf(“不存在值为e的结点”);return NULL;}
  - else if(p->next==NULL)
    - {printf(“值为e的结点是最后一个结点，无后继”);return NULL;}
  - else return p->next;
  - }

单链表操作的实现(7)

插入元素：



p表示当前结点，pre表示前一个结点(的指针)。

在p前插入元素s

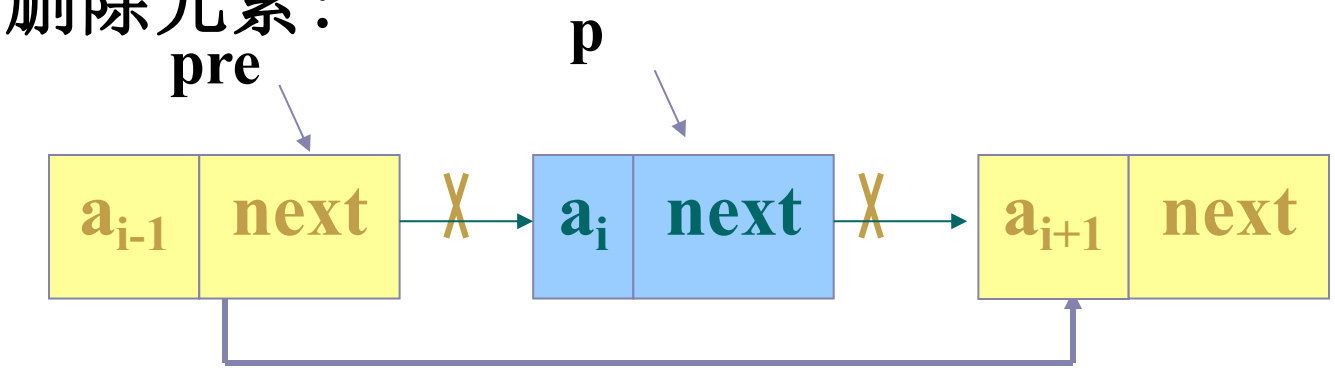
```
s->next  = pre->next;
pre->next = s;
```

单链表操作的实现(7)—插入

```
int ListInsert_L(LinkList &L, int i, ElemType e)
{  //在带头结点的单链表L中第i个位置之前插入元素e
      int  j;       LinkList  p, s; j=0; p=L;
          while(j<i-1&&p)
          {j++;p=p->next;}
          if(j>i-1||!p)   //不存在第i-1个元素
            return 0;
   s=(LinkList)malloc(sizeof(LNode));
   s->data=e;
   s->next=p->next;
   p->next=s;
   return 1;
}
```

单链表操作的实现(8)

删除元素：



p表示当前结点，pre表示前一个结点。

删除p结点的语句

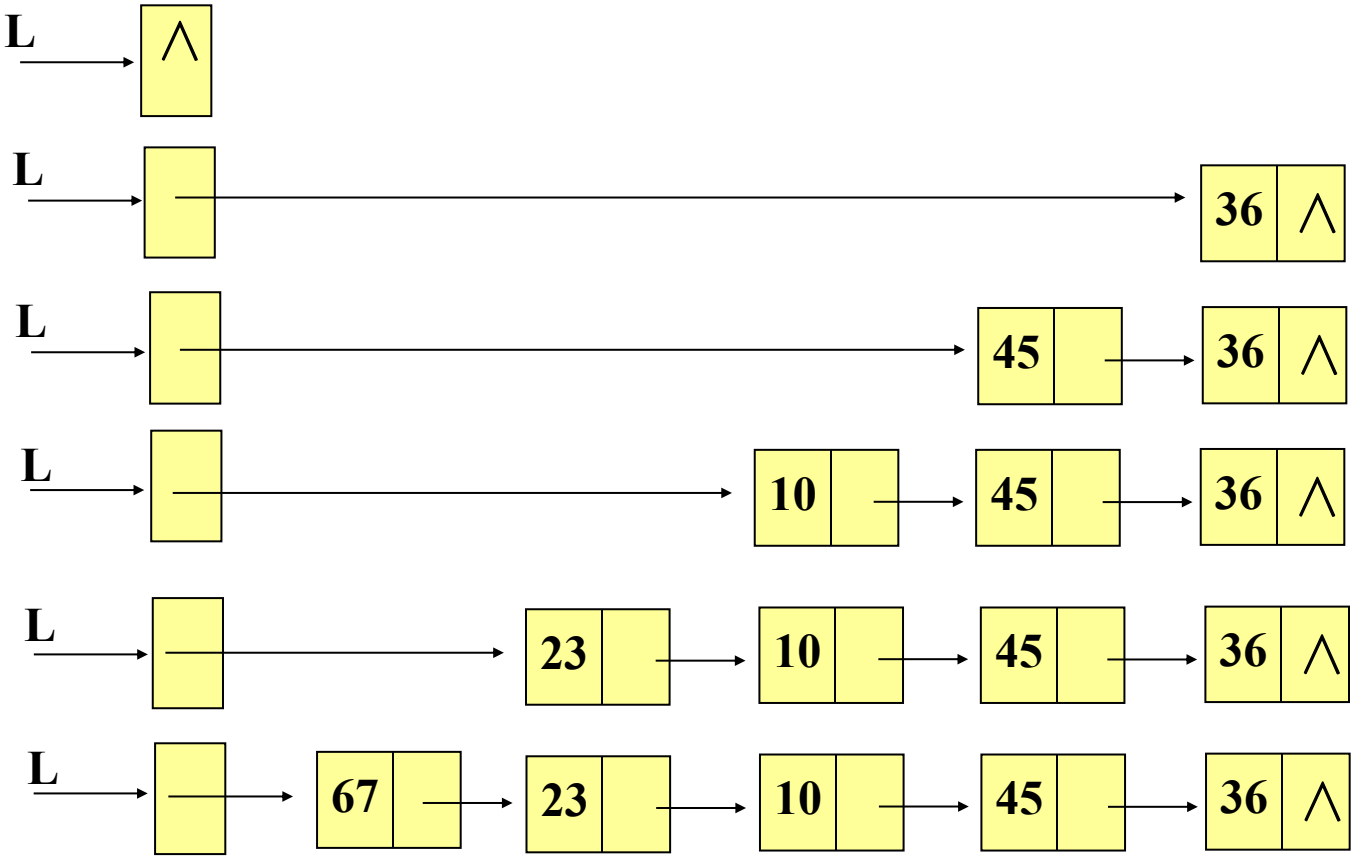
```
pre->next = p->next;
free( p );
```

单链表操作的实现(8)—删除

```
int ListDelete_L(LinkList &L,int i, ElemType &e)
{   int j;   LinkList p, q;
    j=0; p=L;
    while(j<i-1&&p)
    {     j++;p=p->next;   }
    if(j>i-1||!(p->next))
        return 0;
    q=p->next;
    e=q->data;
    p->next=q->next;
    free(q);
    return 1;
}
```

单链表操作的实现(9)

建立单链表——头插法: 从一个空表开始，读入一个数据，生成新结点，将新结点插入到单链表的表头上，直到读入n个数据为止。



单链表操作的实现(9)

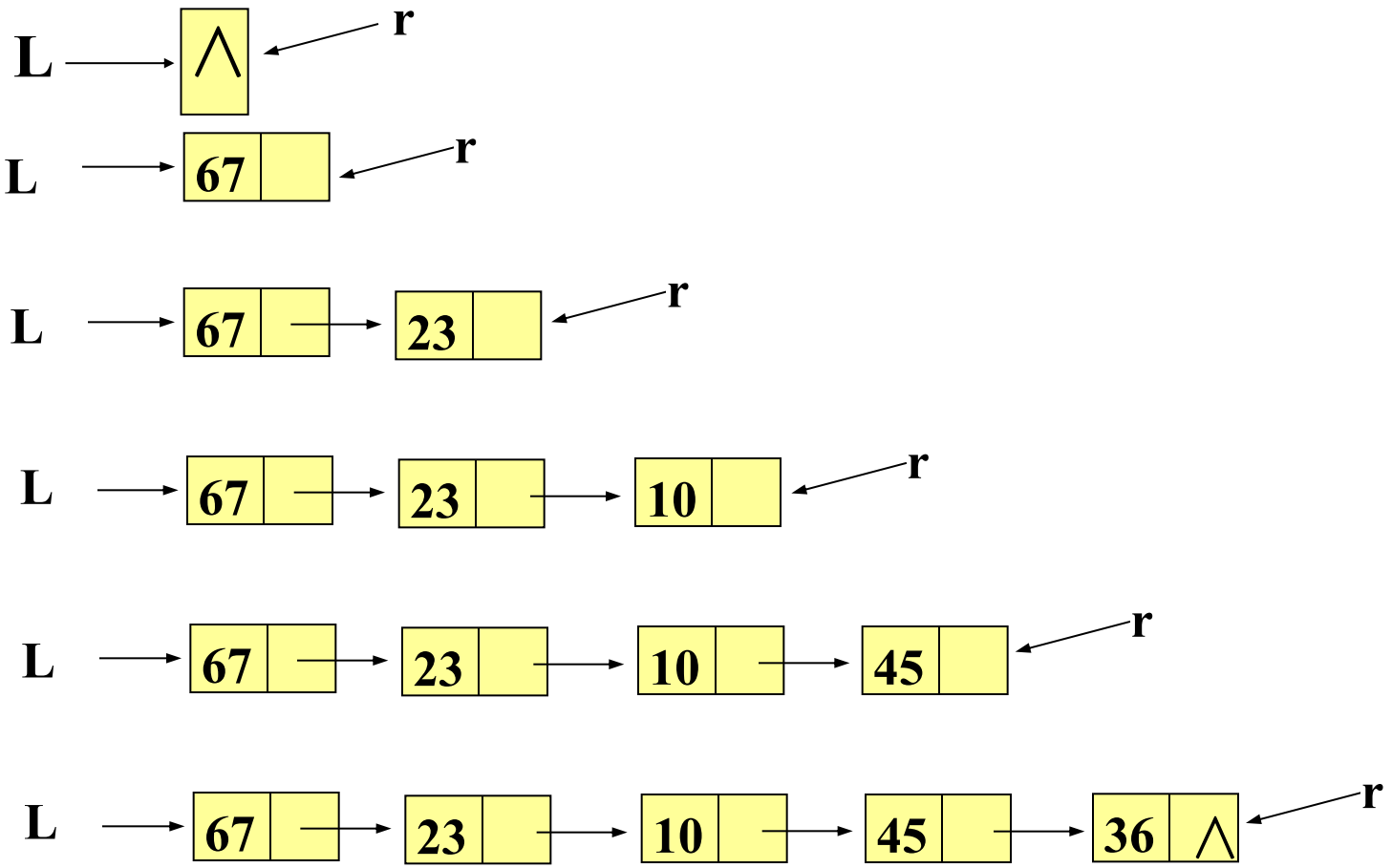
头插法建立单链表算法：

```
void CreateList_L(LinkList &L, int n)  //创建单链表，逆序
{
        LinkList p;     int i;
        L=(LinkList)malloc(sizeof(LNode));
        L->next=NULL;

        for(i=n;i>=1;i--)
        {  p=(LinkList)malloc(sizeof(LNode));
           scanf("%d",&(p->data));
                p->next=L->next;
                L->next=p;
        }
}
```

单链表操作的实现(10)

建立单链表——尾插法:从一个空表开始，读入一个数据，生成新结点，将新结点插入到单链表的表尾上，直到读入结束标识符为止。



单链表操作的实现(10)

尾插法建立单链表算法：

```
void LinkedListCreat2(LinkList &L)
{//用尾插法建立带头结点的单链表
 LinkedList  r;
 L=(LNode*)malloc(sizeof(LNode));
 L->next=NULL; r=L;
 scanf("%d",&x);
 while (x!=flag)          //设置结束标志
    {p=(LNode*)malloc(sizeof(LNode);
     p->data=x;            //赋值元素值
     r->next=p;            //在尾部插入新结点
     r=p;                  //r 指向新的尾结点
  scanf("%d",&x);}
 r->next=NULL;       //最后结点的指针域放空指针
}
```

带头结点的单链表的就地逆置

```
void reverse(LinkList &L)
{
    LinkList   p, q;
    p=L->next;    L->next=NULL;
    while(p!=NULL)
    {
        q=p->next;
        p->next=L->next;
        L->next=p;
        p=q;
    }
}
```

单链表的遍历

```
void print(LinkedList L)
{
LinkedList  p=L->next;

while (p)
    {
    printf(“%d”,p->data);
     p=p->next;
    }

}
```

链表算法举例----合并单链表

- Union(LinkList la, LinkList lb, LinkList lc)
- {‖ 将非递减有序的单链表la和lb合并成新的非递减有序单链表lc，并要求利用原表空间
- lc=(LNode*)malloc(sizeof(LNode); ‖ 申请结点
- lc->next=NULL; ‖ 初始化链表lc
- pa=la->next; ‖ pa是链表la的工作指针
- pb=lb->next; ‖ pb是链表lb的工作指针
- pc=lc; ‖ pc是链表lc的工作指针
- while(pa && pb) ‖ la和lb均非空
- if(pa->data<=pb->data)
- {pc->next=pa; pc=pa; pa=pa->next; }
- ‖ la中元素插入lc(接下页)

链表算法举例

- （接上页）
- else
- {pc->next=pb;　pc=pb; pb=pb->next; } ‖ lb中元素插入lc
- if(pa) pc->next=pa;
  - ‖ 若pa未到尾，将pc指向pa
- else pc->next=pb;
  - ‖ 若pb未到尾，将pc指向pb
  - free(la);
  - free(lb);
- }

思考题

- 上页的合并单链表如果保留la和lb不变，应该如何操作呢？

思路：再定义一个Linklist变量s，为s分配存储空间，给s赋值后加入lc链表中。

- 将非递减有序的单链表la和lb合并成新的非递增有序单链表lc，并要求利用原表空间。

思路：往lc中插入元素时，采用头插法，而不是尾插法。

删除递增有序的带头结点单链表L中值大于min小于max的元素

```
void delminmax(LinkList &L, int min, int max)
{
   LinkList p,q,t;
   q=L;     p=L->next;
   while(p!=NULL&&p->data<=min)
   {       q=p;p=p->next;    }//从p开始删除

   while(p!=NULL&&p->data<max)
   {
      t=p;
      q->next=p->next;
      p=p->next;
      free(t);
   }
}
```

链式结构的特点

- 非随机存贮结构，所以取表元素要慢于顺序表。
  - **节约了内存(结点现用现申请，不会浪费)**
- 适合于插入和删除操作
  - **实际上用空间换取了时间，结点中加入了指针，使得这两种操作转换为指针操作;**

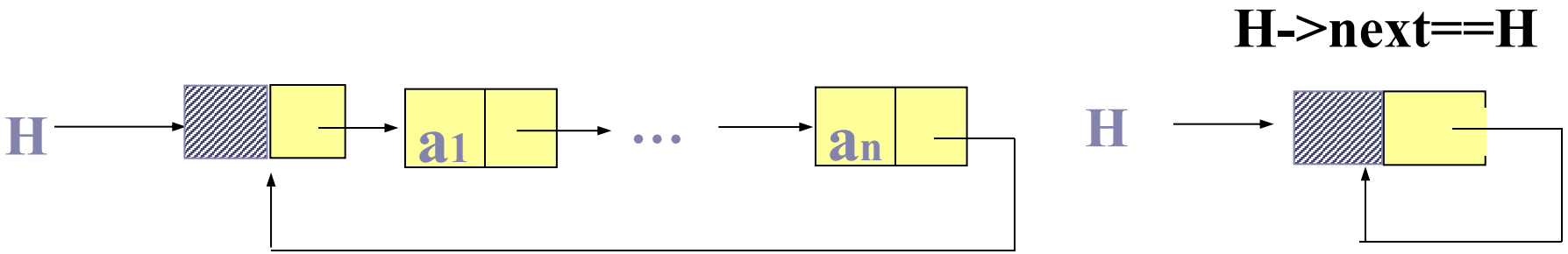
线性表实现方法的比较

- 顺序存储可以随机存取，插入删除操作需要移动元素，要求存储单元连续。
- 链式存储不能随机存取，适用于插入删除操作比较频繁的情况，不需要移动元素，存储单元可以不连续。

循环链表

循环链表：链表尾结点的指针域指向头结点。
这样形成的链表我们叫做循环链表。

单循环链表：



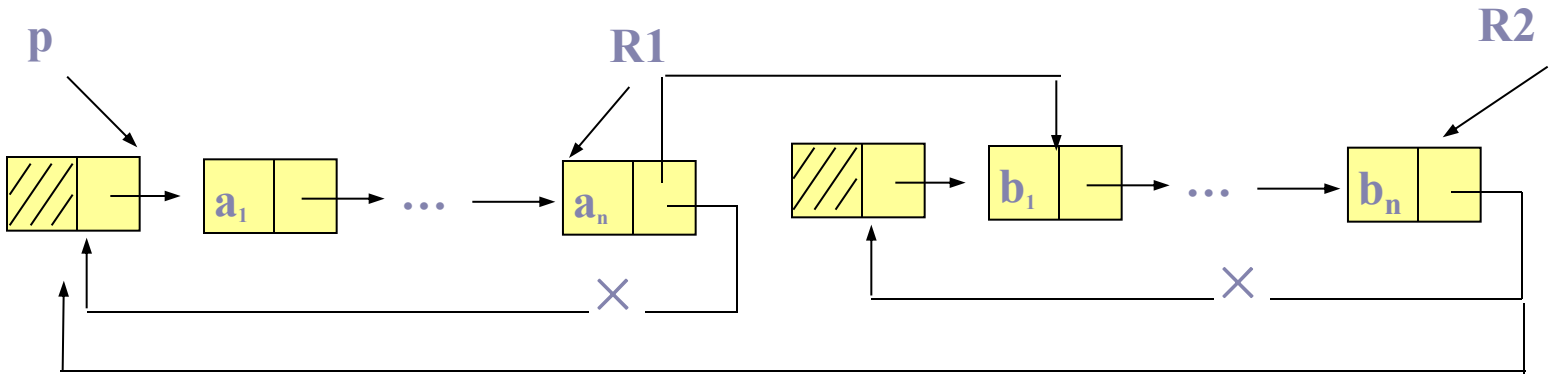
(a)非空表　　(b)空表

循环链表有时只设尾指针

连接两个只设尾指针的单循环链表L1和L2

语句段如下：

```
p= R1->next;                    //保存L1 的头结点指针
R1->next=R2->next->next;//头尾连接
free(R2->next);   //释放第二个表的头结点
   R2->next=p;
```



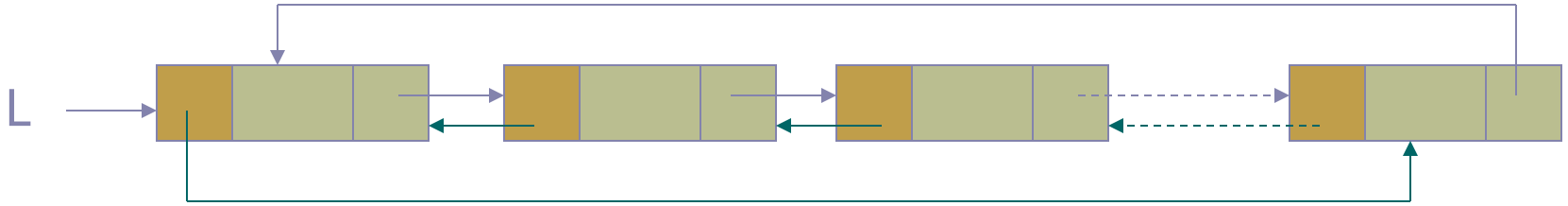
双链表

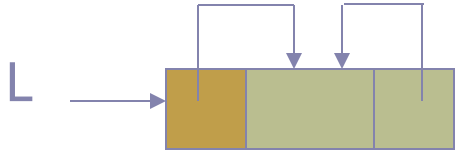
如果希望查找前驱的时间复杂度达到O(1)，我们可以用空间换时间，每个结点再加一个指向前驱的指针域，使链表可以进行双方向查找。用这种结点结构组成的链表称为双向链表。

结点的结构图：

双向链表的逻辑表示

双向循环链表



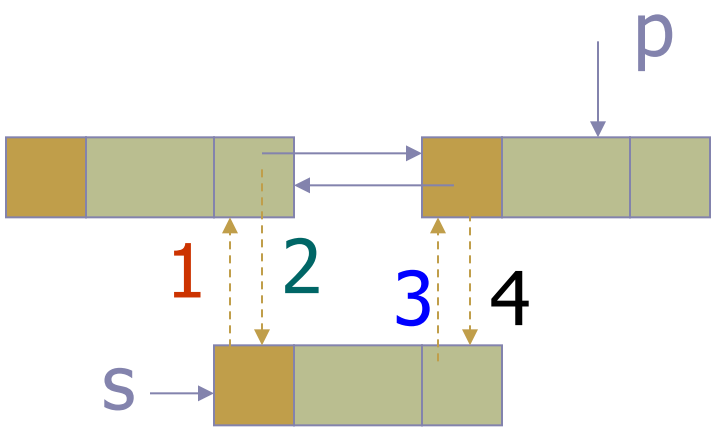


双向链表的类型定义

双向链表结点的类型定义如下：

```
typedef int ElemType;
typedef struct  DLNode
{ElemType data;
 struct DLNode *prior,*next;
}DLNode,*DLinkList;
```

双向链表的插入



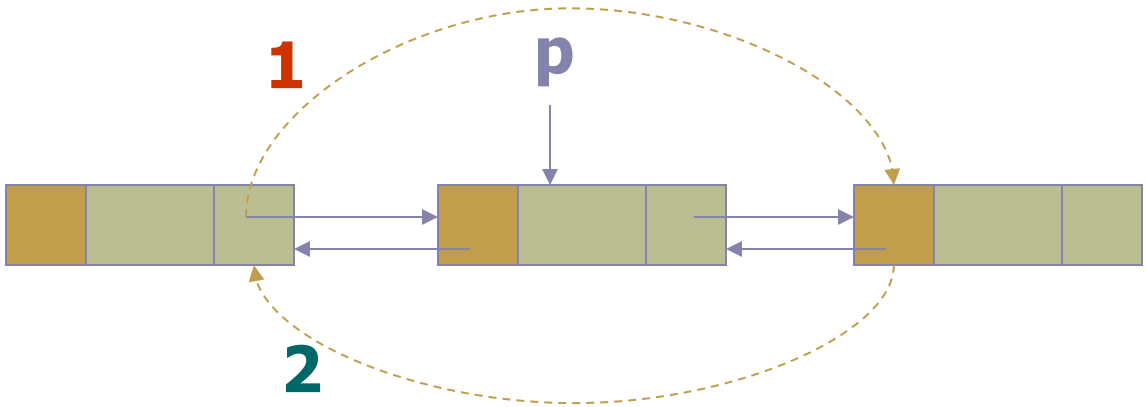
s->prior = p->prior;

p->prior->next = s;

s->next = p;

p->prior = s;

双向链表的删除




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

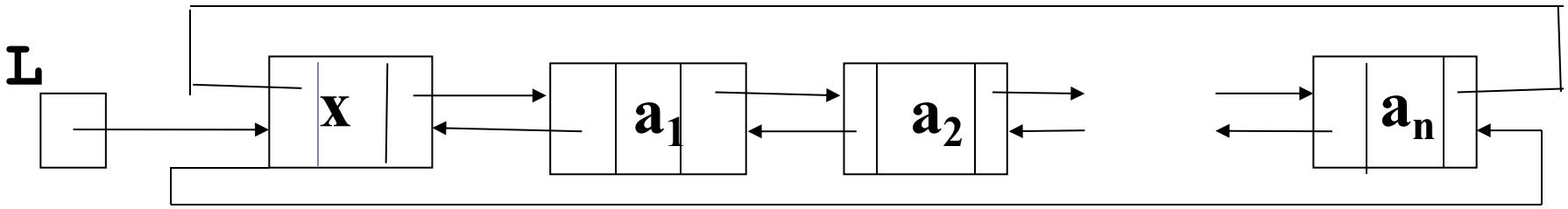
单循环链表算法举例(1)

已知一个带头结点的单循环链表，元素按从大到小排序，试写一个算法，插入一个元素x至循环链表的适当位置，使之保持链表的有序性。

```
void  insertsort(LinkList  L, ElemType x）
{  LinkList p, q, s;
   p=L;    q=L->next;
   s=(ElemType *) malloc (sizeof(ElemType));
   s->data=x;
   while(q!=L&&q->data>x)
   { p=q;  q=q->next; }
   p->next=s;
   s->next=q;
}
```

循环链表算法举例（2）

已知一带头结点的双向循环链表，从第二个结点至表尾递增有序，（设a1<x<an）如下图。试编写程序，将第一个结点删除并插入表中适当位置，使整个链表递增有序（带头结点）



循环链表算法举例（2）

```
void DInsert（DLinkList &L）
 {s=L->next；   x=s->data; ‖ s暂存第一结点的指针
   p=s->next；  ‖ 将第一结点从链表上摘下
   p->prior=L；L->next=p；
    while（p->data<x&&p!=L）
          p=p->next；  ‖ 查插入位置，插入在p的前面
    s->next=p；s->prior=p->prior；‖ 插入s
    p->prior->next=s；p->prior=s；
   }‖ 算法结束
```

- 一元多项式的表示及相加
  - 一元多项式的表示：

$$P_n(x) = P_0 + P_1 x + P_2 x^2 + \cdots\cdots + P_n x^n$$

可用线性表P表示 $P = (P_0, P_1, P_2, \cdots\cdots, P_n)$

但对S(x)这样的多项式浪费空间 $S(x) = 1 + 3x^{1000} + 2x^{20000}$

一般 $P_n(x) = P_1 x^{e1} + P_2 x^{e2} + \cdots\cdots + P_m x^{em}$

其中 $0 \le e1 \le e2 \cdots\cdots \le em$ （P_i为非零系数）

用数据域含两个数据项的线性表表示

$$((P_1, e1), (P_2, e2), \cdots\cdots (P_m, em))$$

其存储结构可以用顺序存储结构，也可以用单链表

- 单链表的结点定义

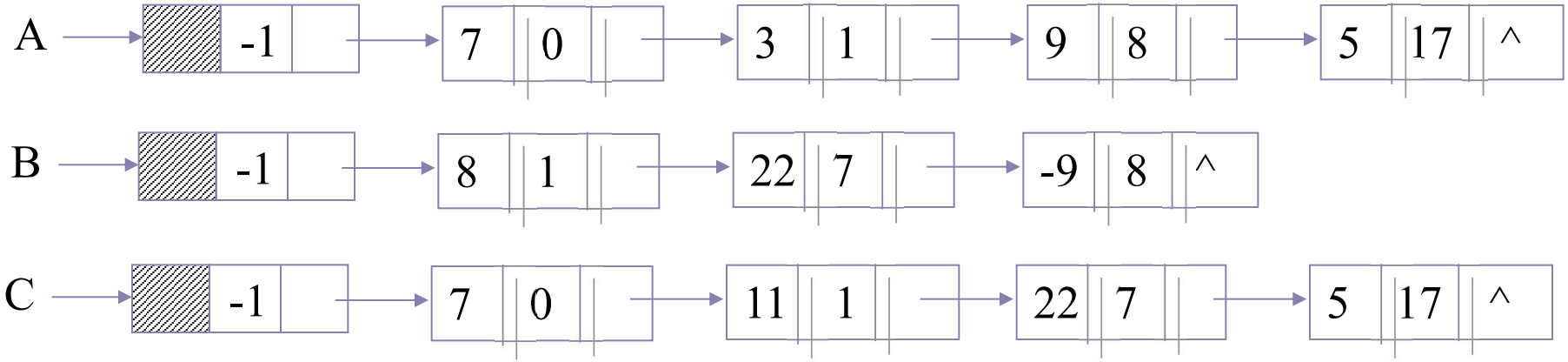
```
typedef   struct node
{   int coef,exp;
    struct node  *next;
}JD;
```

- 一元多项式相加

$$A(x) = 7 + 3x + 9x^{8} + 5x^{17}$$

$$B(x) = 8x + 22\ x^{7} - 9x^{8}$$

$$C(x) = A(x) + B(x) = 7 + 11\ x + 22\ x^{7} + 5x^{17}$$



- 运算规则 A=A+B

设p,q分别指向A,B中某一结点，p,q初值是第一结点，

比较 p->exp与q->exp

- p->exp < q->exp: p结点是和多项式中的一项 p后移,q不动
- p->exp > q->exp: q结点是和多项式中的一项 将q插在p之前,q后移,p不动
- p->exp = q->exp: 系数相加
  - 0：从A表中删去p, 释放p,q，p,q后移
  - ≠0：修改p系数域, 释放q，p,q后移

直到p或q为NULL

- 若q==NULL，结束
- 若p==NULL，将B中剩余部分连到A上即可

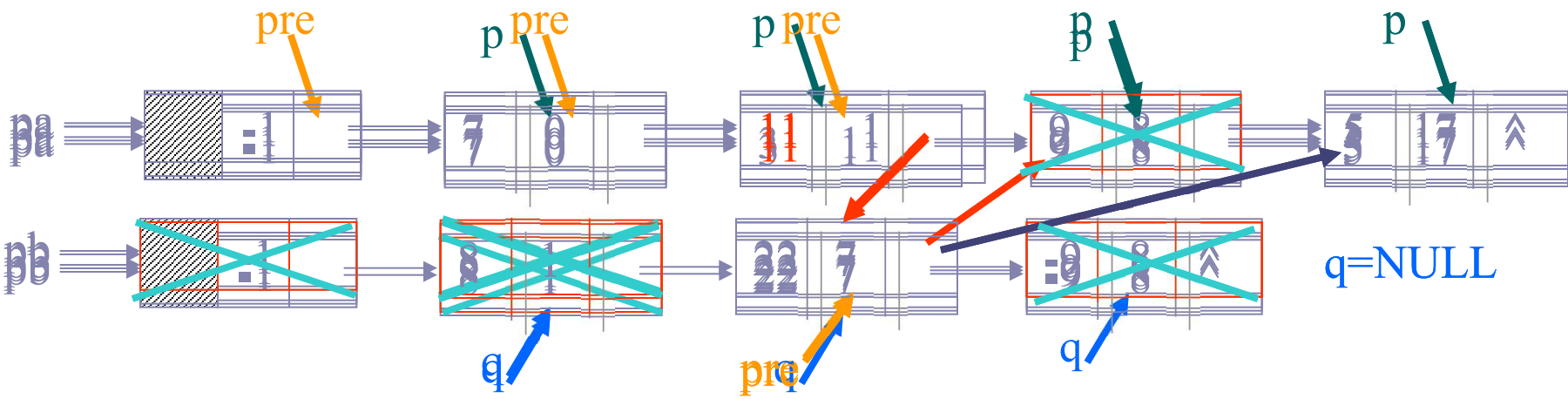
算法描述

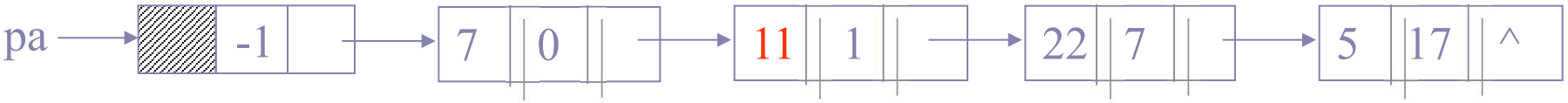
```
void  add_poly(JD * &pa,JD *pb)
{  JD *p,*q,*u,*pre;
   int x;
   p=pa->next;   q=pb->next;   pre=pa;
   while((p!=NULL) && ((q!=NULL))
   {   if(p->exp<q->exp)   //p后移
       {   pre=p;  p=p->next;}
       else if(p->exp==q->exp)
            {  x=p->coef+q->coef;  //计算系数和
               if(x!=0){ p->coef=x;  pre=p;}
               else    { pre->next=p->next;  free(p); //删除并释放p}
               p=pre->next; //p后移
               u=q;          q=q->next;  //q后移，并释放原来的q
               free(u);
            }
```

接下页

接上页

```
        else  // p->exp>q->exp的情况
        {    u=q->next;
             q->next=p;pre->next=q;  //将q插入在p前面
             pre=q;  q=u;
        }
    }  //while
    if(q!=NULL)
       pre->next=q;
    free(pb);
}
```


pre
p
pre
p
pre
p
p
pa
pb
q
pre
q
q=NULL


pa
-1
7
0
11
1
22
7
5
17
^