第二章

线性表

Linear List

主讲：顾为兵

第二章 线性表

目录

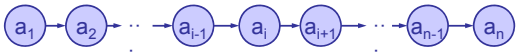
线性表的逻辑结构：

Linear_List=(D,S)

D={a_1, a_2, … , a_n}

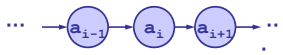
S={s}

s={<a_1,a_2>, <a_2,a_3>, …, <a_{i-1},a_i>, …, <a_{n-1},a_n>}



起始结点
(表头结点,首元结点)

终端结点
(表尾结点)



称：

a_{i-1}是a_i的直接前趋（有时简称"前趋"）

a_{i+1}是a_i的直接后继（有时简称"后继"）

- 线性表是n(n>=0)个结点的有限序列;
- n=0的表称为空表;
- 数据元素呈线性关系。对于非空线性表，必存在唯一的称为“第一个”的数据元素;必存在唯一的称为“最后一个”的数据元素;
- 除首元素a_1没有前趋外，其他结点有且仅有一个直接前驱;
- 除表尾元素a_n没有后继外，其他结点有且仅有一个直接后继

线性表抽象数据类型的定义（P19—20）：

```
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-1,ai∈D, i=2,...,n }
   基本操作:
      InitList (&L)            (初始化)
        操作结果：构造一个空的线性表L
      DestroyList(&L)          (销毁表)
        初始条件：线性表L已经存在;  操作结果：销毁线性表L
      ClearList(&L)            (清空表)
        初始条件：线性表L已经存在;操作结果：清空L
      ListEmpty(L)             (判表空)
        初始条件：线性表L已经存在
        操作结果：若L为空表，则返TRUE，否则返回FALSE
```

（未完，接下页）

返回

(接上页)

ListLength(L)　　(求表长)
初始条件：线性表L已经存在
操作结果：返回L中数据元素的个数

GetElem(L, i, &e)　　(读表元)
初始条件：线性表L已经存在，1≤i≤ListLength(L)
操作结果：用e返回L中第i个元素的值

LocateElem(L, e, compare())　(定位函数)
初始条件：线性表L已经存在，compare()是判定函数
操作结果：返回L中第一个与e满足compare()关系的数据元素的位序。若这样的数据元素不存在，则返回0。

PriorElem(L, current_e, &prior_e)　(求前驱)
初始条件：线性表L已经存在
操作结果：若current_是L中的数据元素，且不是第一个，则用prior_e 返回它的直接前驱的值。否则操作失败，prior_e无定义。

返回

(接上页)

NextElem(L, current_e, &next_e)　(求后继)
初始条件：线性表L已经存在
操作结果：若current_是L中的数据元素，且不是最后一个，则用next_e 返回它的直接后继的值。否则操作失败，next_e无定义

ListInsert(&L, i, e)　　(前插)
初始条件：线性表L已经存在，1≤i≤ListLength(L)+1
操作结果：在L的第i个位置之前插入新的数据元素e

ListDelete(&L, i, &e)　　(删表元)
初始条件：线性表L已经存在，1≤i≤ListLength(L)
操作结果：删除L的第i个数据元素，并用e返回其值

ListTraverse(L, visit())　　(遍历表)
初始条件：线性表L已经存在
操作结果：依次对L的每个元素调用函数visit()。

}ADT List

返回

当线性表的ADT被实现后，通过其基本运算的组合调用，可以实现对线性表的各种复杂运算。

举例：求两个线性表La和Lb的并集　La = La ∪ Lb

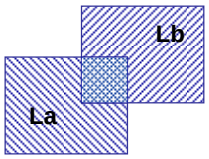
（要求：“就地运算”，运算结果仍然存放在La中）

算法思想：

对Lb中的每个元素bi，检测bi在La中是否已经存在。

若不存在，则将bi插在La的表尾位置，成为La的新表尾；

若存在，则不插。




```
void Union(List &La, List Lb){ //(P20,算法2.1)
  //求线性表La和Lb的并集
   La_len=ListLength(La);   //求La的表长
   Lb_len=ListLength(Lb);   //求Lb的表长
   for(i=1; i<=Lb_len; i++){
      GetElem(Lb, i, bi); //读Lb的第i个元素bi
      if(!LocateElem(La, bi, equal)) //检测
          ListInsert(La, ++La_len, bi);
   }//end for
 }// Union
```

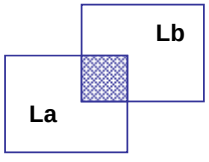
线性表基本运算

举例：求两个线性表La和Lb的交集　La = La ∩ Lb

要求：“就地运算”。

算法思想：

对La中的每个元素a_i，检测a_i在Lb中是否已经存在。若不存在，则将a_i从La表中删除。



```
void Intersect(List &La, List Lb) {
  //求两个线性表的交集：La=La∩Lb
  i=1;
  while(i≤ListLength(La)){
     GetElem( La, i, e );  //读e=ai
     if(!LocateElem(Lb, e, equal))
          ListDelete(La, i, e); //删ai
     else
          i++;
  } // end while
} // Intersect
```

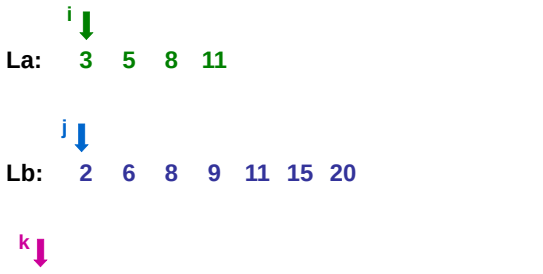
线性表基本运算

举例：将两个有序线性表La和Lb归并成一个线性表Lc

（“异地运算”，运算结果放在第三个线性表Lc中）

算法思想：

- 设La和Lb中的元素都是非递减有序排列。
- 分别从头至尾逐个读取La和Lb中的每个元素ai和bj；比较两个元素的大小：
- 如果ai≤bj，则将ai插入Lc的表尾，i++；
- 如果ai>bj，则将bj插入Lc的表尾，j++；
- 直至一个表被扫描完。此时，将另一个表中的剩余元素一并加入Lc的表尾。




```
void MergeList(List La, List Lb, List &Lc) { //P21算法2.2
  //将有序线性表La和Lb归并成有序线性表Lc
  i=j=1; k=0; InitList(Lc); //初始化
  La_len=ListLength(La);Lb_len=ListLength(Lb);//求表长
  while(i<=La_len && j<=Lb_len){
     GetElem(La,i,ai); GetElem(Lb,j,bj);//读ai和bj
     if(ai<=bj){ ListInsert(Lc, ++k, ai); i++;}
     else{ListInsert(Lc, ++k, bj); j++;}
  }
  while(i<=La_len){ //当La中有未归并完的元素时
     GetElem(La,i++,ai); ListInsert(Lc, ++k, ai);}
  while(j<=Lb_len){ //当Lb中有未归并完的元素时
     GetElem(Lb,j++, bj); ListInsert(Lc, ++k,  bj);}
}// MergeList
```

第二章 线性表

目录

§2.2 线性表的顺序表示和实现----顺序表

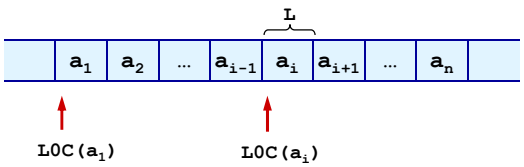
⇨ 用一组地址连续的空间存放线性表的数据元素。各元素按其逻辑次序依次存放；

⇨ 采用顺序存储结构存储的线性表又称为顺序表；

⇨ 顺序表的特点：

逻辑上相邻的元素，其物理位置也相邻；

元素之间的逻辑关系通过物理位置的先后来体现



元素物理地址的计算：

$LOC(a_{i+1}) = LOC(a_i) + L$

$LOC(a_i) = LOC(a_1) + (i-1) \times L$

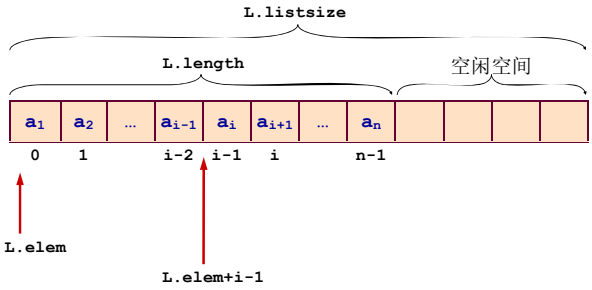
顺序存储结构的类型定义：

```
#define  List_Init_Size 100 //线性表的初始存储空间
#define  ListIncrement  10  //线性表的增量空间

typedef  struct {
   ElemType     *elem;      //顺序表首地址指针
   int          Length;     //顺序表中元素个数
   int          Listsize;   //顺序表的容量
}SqList;
```



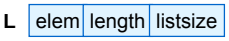
```
SqList L;
```



顺序表存储结构示意图

线性表的基本运算在顺序表上的实现：

举例：初始化空顺序表：




```
Status InitList(SqList &L) {
  L.elem=(ElemType*)
          malloc(sizeof(ElemType)*List_Init_Size);
  if(!L.elem) return OVERFLOW;     //存储分配失败
  L.Length=0;
  L.listsize=List_Init_Size;
  return OK;
}
```

基本运算

读表元操作：

```
Status GetElem(SqList L, int i, ElemType &e){
   if(i<1||i>L.length) return ERROR; //i非法
   e=*(L.elem+i-1);    用下标的写法 e=L.elem[i-1];
   return OK;
}
```

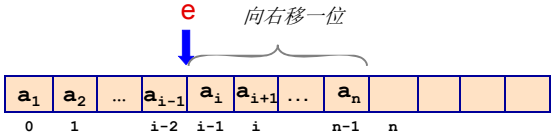
对于顺序存储结构而言，读表元非常容易。因为元素的逻辑位置与物理下标存在简单的线性关系。时间复杂度是O(1)级的。

所以说，顺序表是一种顺序存储，随机存取的结构

基本运算

插入操作（前插）：

- 必须保证逻辑上相邻的元素物理上也相邻
- 合法的插入位置有n+1个，i=1..n+1



基本运算

插入操作算法思想：

1. 检查插入位置i是否合法：1≤i≤n+1，n+1个插入位置；
2. 检查数组中是否有空闲的空间放新元素。如果没有，将数组扩展ListIncrement个结点
3. 将a_n至a_i逐个右移一位
4. 新元素e放入下标为i-1的单元
5. 修改表长：L.length++

顺序表的插入操作必须大量移动元素


```
Status ListInsert(SqList &L, int i, ElemType e){
  //在顺序表L的第i个元素之前插入新结点e
    if(i<1||i>L.length+1) return ERROR; // i非法
    if(L.length>=L.listsize){   //需要扩展数组
       newsize=L.listsize+ ListIncrement;
       newbase=(ElemType *)realloc(L.elem,
                    newsize*sizeof(ElemType));
       if(!newbase)exit(OVERFLOW);
       L.elem=newbase;
       L.listsize += ListIncrement;
    } //end if
```

(未完,接下页)

(续前)

```
    q=L.elem+i-1;       //令q指向ai
    for(p=L.elem+L.length-1; p>=q; p--)
       *(p+1)=*p;     //p指针从表尾向左扫描，元素右移一位
    *q=e;             //将e放入第i个元素的位置上
     L.length ++;
     return OK;
}   //ListInsert
```


插入操作的时间性能分析：

- 在第i个元素之前插入新元素需要移动 n-i+1个元素 (1≤i≤n+1) ；
- 设：在第i个元素之前插入新元素的概率为p_i；
- 做一次插入操作所需移动元素次数的期望值(平均次数)为：

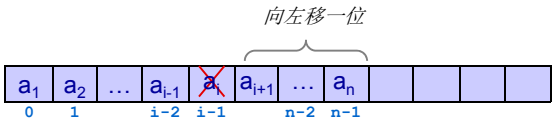
$$E_{ins}=\sum_{i=1}^{n+1}p_i(n-i+1)$$

- 如果在任何位置上做插入操作的概率相等，即$p_i=1/(n+1)$，则

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

删除操作：

必须保证逻辑上相邻的元素物理上也相邻



删除操作的算法：

1. 检查删除位置 i 是否合法，1≤i≤n，n个删除位置；
2. 将a_{i+1}至a_n逐个左移一位
3. 修改表长：L.length--

顺序表的删除操作也必须大量移动元素

```
Status ListDelete(SqList &L, int i, ElemType &e) {
  //删除顺序表的第 i 个元素，用e返回被删元素的值
  if(i<1||i>L.length) return ERROR; //i非法
  p=&(L.elem[i-1]);        //令指针p指向ai
  e=*p;
  for(p++;p<=L.elem+L.length-1; p++)
  *(p-1)=*p;          //ai+1...an左移一位
  L.length-- ;          //表的长度减1
  return OK;
} //ListDelete
```


删除操作的时间性能分析：

- 删除第i(1≤i≤n)个元素需要移动n-i个元素；
- 设删除第i个元素的概率为q_i；
- 做一次删除操作所需移动元素次数的期望值(平均次数)为：

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

- 如果在任何位置上做删除操作的概率都相等，即$q_i=1/n$，则

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

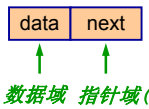
第二章 线性表

目录

§2.3 线性表的链式表示和实现

2.3.1 单链表

- 特点：用一组可能连续，也可能不连续的任意存储单元存放线性表中的数据元素。
- 元素之间的逻辑联系通过指向直接后继的指针来体现。
- 指针next本身并不是数据元素的成份。



数据域 指针域(附加空间)

链表结点的类型定义：

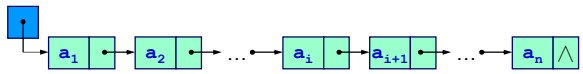
```
typedef struct Node {
   ElemType          data ;
   struct Node       *next ;
}LNode, *LinkList;
```



```
LNode    --- 结构体的类型名
LinkList --- 结构体指针的类型名，专门用来说明头指针变量，
             以增加算法的可读性(头指针代表了一个线性表)。

比如，
      struct Node *p;     //p是普通指针；
      LinkList L;   //L是头指针,引导一个线性链表
```

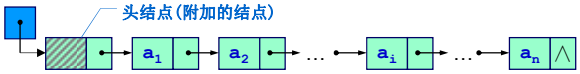
L(头指针)



不带头结点的单链表

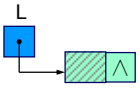
判表空条件：L==NULL

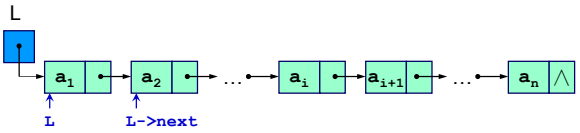
L(头指针)

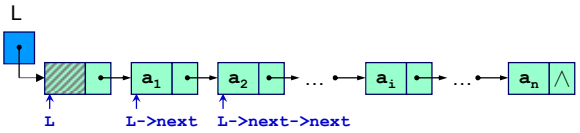


带头结点的单链表

判表空条件空表：L->next==NULL



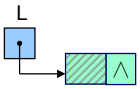




线性表的基本运算在单链表上的实现：

初始化空链表：

```
Status InitList ( LinkList  &L ) {
    //L是带头结点的单链表的头指针
   L=(LNode*)malloc(sizeof(LNode)); //申请头结点
   if(!L) return OVERFLOW;
   L->next=NULL;
   return OK;
}
```



读表元操作：

```
Status  GetElem (LinkList L,  int i,  ElemType  &e) {
 //L是带头结点的单链表，读L的第i个元素，用e返回其值
   j=1; p=L->next;     // p指向首元素a1
  while(p && j<i){p=p->next; j++;}   // p最多右移i－1次
  if(!p||j>i)return ERROR;   //i非法。!p--i太大,j>i--i太小
  e=p->data;
  return OK;
} //GetElem O(n)
```

不能根据i算出a_i的地址直接读，只能从头结点开始一个一个向后”摸”。这是一种随机存储，顺序存取的结构。

基本运算

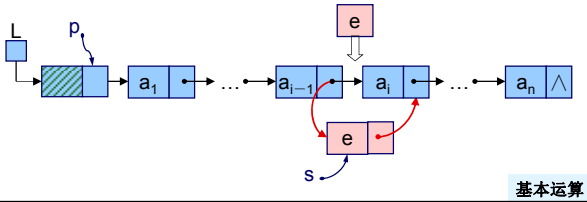
求表长操作：

```
int  ListLength ( LinkList  L ) {
   //L是带头结点的单链表
    n=0;  p=L->next;  //p指向首元素
   while(p)  {n++; p=p->next;}
   return  (n);
} //ListLength O(n)
```

基本运算

插入操作：

1. 指针p从头结点开始，“摸”到a_{i-1};
2. 检查i的合法性；
3. 为新结点申请空间s；
4. 修改指针：s->next=p->next; p->next=s;



基本运算

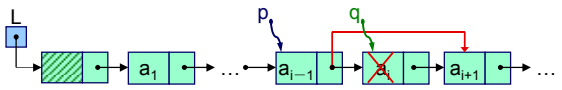
```
Status ListInsert( LinkList L, int i, ElemType e ) {
  // L是带头结点的单链表,在ai之前插入新结点e
   p=L; j=0;  //p指向头结点，j是计数器
   while(p && j<i-1){ p=p->next; j++; } //令p指向ai-1
   if (!p || j>i-1) return ERROR;        //i非法
   s=(LNode*)malloc(sizeof(LNode));      //申请新结点
   s->data=e;                            //填入数据元素
   s->next=p->next;  p->next=s;          //修改指针
   return OK;
} // ListInsert  O(n)
```

特点：不需要大量移动元素，只需局部修改指针。虽然时间复杂度也是O(n)，但时间是消耗在指针运动上。

删除操作：

1. 指针p从头结点开始，“摸到”a_{i-1}；
2. 修改指针域：

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

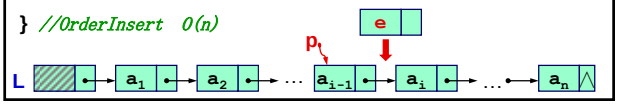



```
Status ListDelete( LinkList L, int i, ElemType &e ) {
  // L是带头结点的单链表,删除ai，用参数e被删结点的值
   p=L; j=0;     //p指向头结点，j是计数器
   while(p->next && j<i-1){p=p->next; j++;}//p指向ai-1
   if (!p->next || j>i-1) return ERROR;// i非法
   q=p->next; e=q->data;                //q指向ai
   p->next=q->next; free(q);            //修改指针
   return OK;
}  // ListInsert  O(n)
```

特点：也不需要大量移动元素，只需修改指针。时间复杂度也是O(n)，是消耗在指针运动上。

举例：在有序单链表L中插入新结点，使得L仍然有序

```
void OrderInsert( LinkList L, ElemType e ) {
 //L是带头结点且非递减有序的单链表，插入新元素e，使L仍然有序
   p=L;
   while(p->next!=NULL && p->next->data<e)
      p=p->next;       //找插入位置
   s=(LNode *)malloc(sizeof(LNode));
   s->next=p->next; s->data=e;
   p->next=s;
} //OrderInsert  O(n)
```

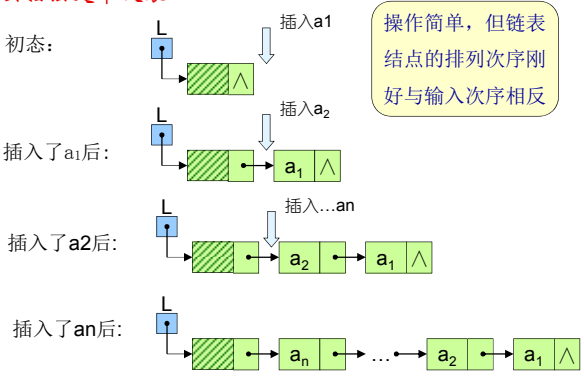


举例：创建单链表

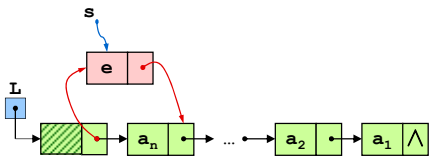
两种处理方法：

1. 头插法： 插入点始终在表头位置，被插元素总是新的表头；
2. 尾插法： 插入点始终在表尾位置，被插元素总是新的表尾。

头插法建单链表



操作简单，但链表结点的排列次序刚好与输入次序相反



头插时执行的语句：

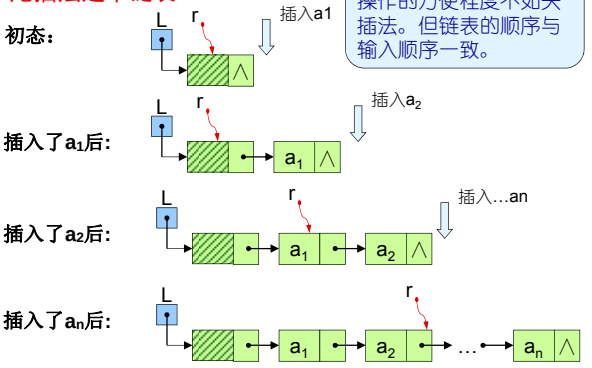
```
(1) s->next=L->next;
(2) L->next=s;
```

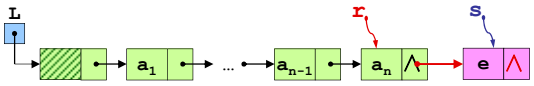

头插法算法：

```
void Head_Insert( LinkList &L ) {
//从键盘输入结点，按头插法创建带头结点的单链表L
  L=(LinkList)malloc(sizeof(LNode));L->next=NULL;
    //初始化空链表L
  scanf(&e);        //从键盘输入一个结点
  while(e != EndMark){   //EndMark是某个结束标志
     s=(LNode*)malloc(sizeof(LNode)); //申请新结点
     s->data=e;
     s->next=L->next; L->next=s;     //新结点插入链表
     scanf(&e);      //继续读
   } //end while
}  //end Head_Insert;
```


尾插法建单链表

需要增设一个尾指针r，操作的方便程度不如头插法。但链表的顺序与输入顺序一致。





尾插时执行的语句：

```
(1)  s->next=NULL
(2)  r->next=s
(3)  r=s
```


尾插法算法：

```
void Tail_Insert( LinkList &L ) {
//从键盘输入结点，按尾插法创建带头结点的单链表L
  L=(LinkList)malloc(sizeof(LNode));L->next=NULL;
  r=L;     //尾指针r初始时指向头结点
  scanf(&e);        //从键盘输入一个结点
  while(e != EndMark){   //EndMark是某个结束标志
     s=(LNode*)malloc(sizeof(LNode)); //申请新结点
     s->data=e;
     s->next=NULL; r->next=s; r=s;   //新结点插入链表
     scanf(&e);      //继续读
   } //end while
}  //end Tail_Insert;
```

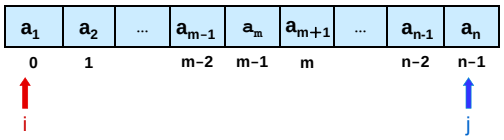

举例：逆置线性表

逆置前：$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{1+1}, \ldots, a_{n-1}, a_n)$

逆置后：$(a_n, a_{n-1}, \ldots, a_{i+1}, a_i, a_{i-1}, \ldots, a_2, a_1)$

对顺序表和单链表，实现逆置的方法是不一样的

逆置顺序表:



- 初始时在表头和表尾各设一个指针i和j;
- 互换i、j所指的元素;
- i++; j--;
- 重复执行，直至i和j相遇。

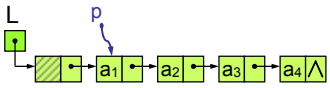
逆置顺序表:

```
void invert ( ElemType *E, int i, int j ) {
  //将数组E自下标i至下标j之间的元素逆置
  while(i<j) E[i++] ↔ E[j--] ;
}

void InvertSqList( SqList L ) {
  //将顺序表L中的所有元素逆置
  invert(L.elem, 0, L.length-1);
}
```


逆置单链表

方法1：用头插法重组单链表

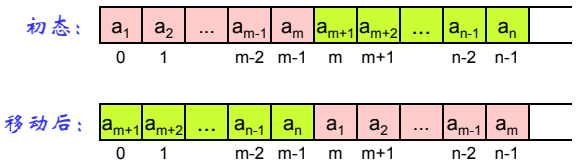


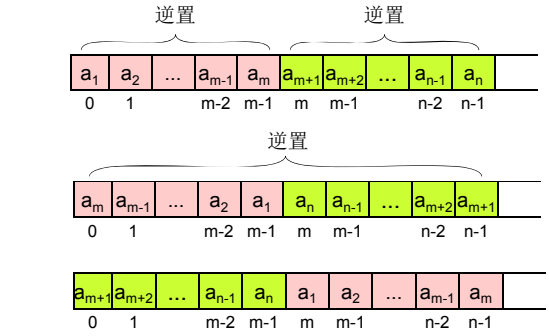

```
void InvertLinkList1(LinkList L){
      // L是带头结点的单链表。用头插法逆置L
   p=L->next;          //p指向a1
   L->next=NULL;       //剥离头结点
   while(p){
      q=p->next;       //用q保留p后继的指针
      p->next=L->next; L->next=p; //将p头插在L中
      p=q;
   }//while
}//InvertLinkList1
```


方法2：将所有结点的next指针逆转

```
void InvertLinkList2 ( LinkList L ) {
  // L是带头结点的单链表的头指针
   if(!L->next)  return;          //空表
   p=L->next; q=p->next;
   p->next=NULL; //a1的next置空指针，剥离头结点
   while(q){ r=q->next ;
             q->next=p ; //将q->next指向q的前驱
             p=q ; q=r ;
   } //while
   L->next=p;
} //InvertLinkList1
```


举例：将线性表中的元素整体移动



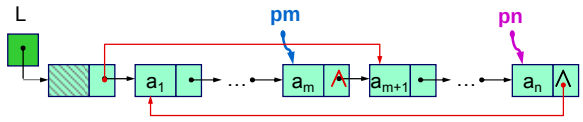


$T(n) = 3m/2 + 3(n-m)/2 + 3n/2 = 3n = O(n)$

将顺序表元素整体移动:

```
void  Move ( SqList L, int m ) {
  //将顺序表L中前m个元素与后n-m个元素互换位置
    invert(L.elem, 0, m-1);              //互换左区间
    invert(L.elem, m, L.length-1);       //互换右区间
    invert(L.elem, 0, L.length-1);       //互换整个区间
}
```


整体移动在单链表上的实现:



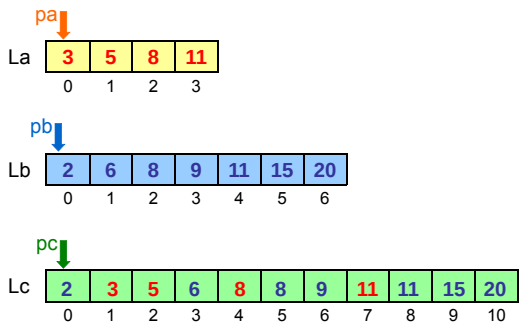
修改指针的操作:

1) pn->next = L->next;
2) L->next = pm->next;
3) Pm->next = NULL;

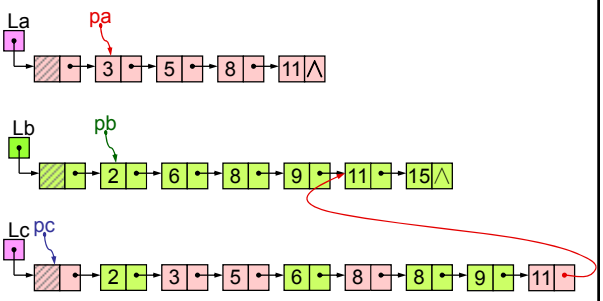
将链表的元素整体移动:

```
Status Move ( LinkList  L,  int m ) {
//将单链表L中前m个元素与后n-m个元素互换位置
  j=1; pm=L->next;
  while(p && j<m){pm=pm->next;j++;} //令pm指向a_m
  if(!pm||j>m) return ERROR;        //m非法
  pn=pm;
  while(pn->next) pn=pn->next;      //令pn指向a_n
  if(pm!=pn){      //修改指针
      pn->next=L->next; L->next=pm->next;
      pm->next=NULL;
  }//if
  return OK; } //Move
```


举例：两个有序顺序表的归并(P26，算法2.7)



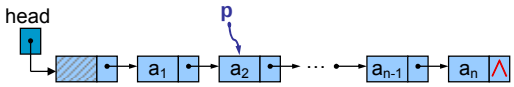
例5：两个有序单链表的归并(P31，算法2.12)



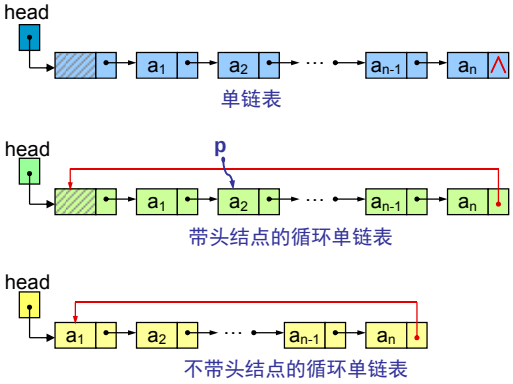
§ 2.3 线性表的链式表示和实现

2.3.2 循环链表

在普通单链表中，从某个结点开始，沿着向后链，不能遍历完线性表的全部结点。



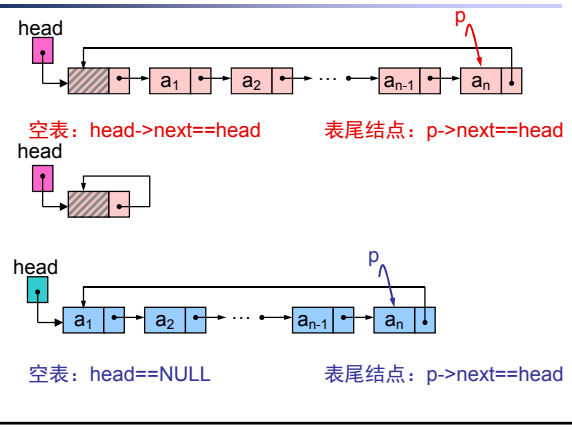
单链表

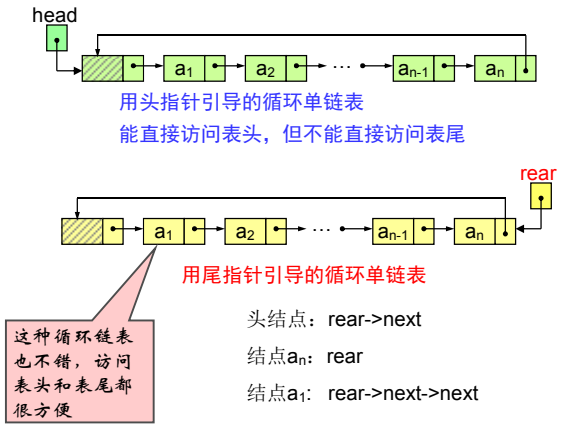


- 在循环单链表中，从任意一个结点出发，沿着向后链能够访问到表中的所有元素；
- 对循环链表的操作与普通单链表基本相同，只是判表空和判断表尾元素的条件有所不同:

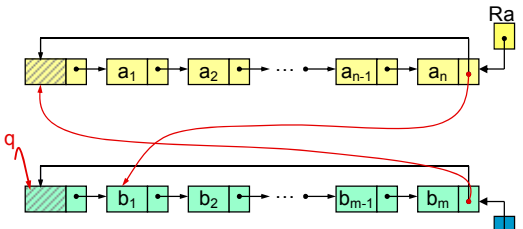
判断表尾：p->next = = head

判断表空：head->next = = head





例：将两个循环单链表合并成一个循环单链表

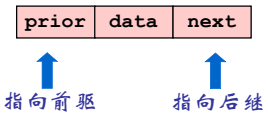



```
void  connect( LinkList &Ra,  LinkList Rb ) {
   q=Rb->next;
   Rb->next=Ra->next;  Ra->next=q->next; Ra=Rb;
   free(q);  }
```


§ 2.3 线性表的链式表示和实现

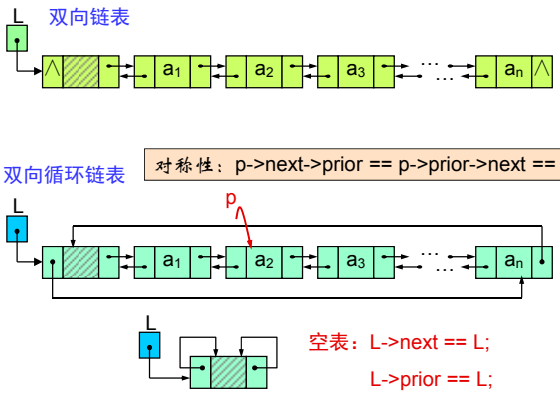
2.3.3 双向链表

在单链表中，通过向后链可以直接访问一个结点的后继，但是不能直接访问到一个结点的前驱，只能从头结点开始逐个向后摸。为了能直接访问前驱，可以在每个结点上增加一个指向前驱的指针域prior。



双向链表的结点类型定义：

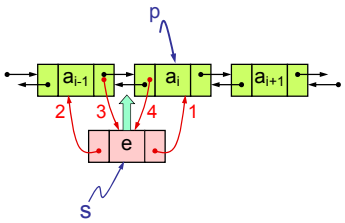

```
typedef struct  DuLNode  {
      ElemType          data;
      struct  DuLNode   *prior;
      struct  DuLNode   *next;
} DuLNode,  *DuLinkList;
```

对双向链表而言，大部分操作与单链表差不多，只是插入和删除操作的差别比较大。

大家通过对比教材上算法2.9和2.18以及2.10和2.19，可以体会其中的差别。

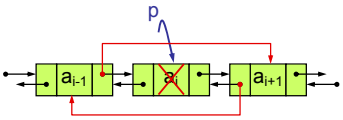
插入操作：



修改指针的语句：

```
1. s->next=p;
2. s->prior=p->prior;
3. p->prior->next=s;
4. p->prior=s
```


删除操作：



修改指针的语句：

(1) p->prior->next=p->next;

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

(3) free(p)

由于线性链表中结点之间序偶关系是用后继指针来表示的，因此数据元素在线性表中的“位序”的概念已淡化，而代之以元素在链表中的“位置”，即结点的指针。

为此，可以针对线性链表的特点定义其基本运算的集合。见教材P37－38页。

线性链表类型定义：　(见P37)

```
typedef struct LNode{     //结点类型
      ElemType        data;
      struct  LNode   *next;
}LNode, *Link, *Position;    //Link用来说明头指针
                             //Position用来说明普通指针变量

typedef struct LNode{     //链表类型
      Link head, tail;    //头指针和尾指针
      int  len;                 //表长
}LinkList;   //注意，与P28定义的LinkList不同
```


线性链表基本运算集合（见P37-38）

```
Status MakeNode(Link &p, ElemType e);
      //构造一个新结点
Status FreeNode(Link &p);
      //释放p所指结点
Status InitList(LinkList &L);
      //构造一个空的线性链表L
Status DestroyList(LinkList &L);
      //销毁线性链表L
Status ClearList(LinkList &L);
      //清空线性链表L
```

接下页


```
Status InsFirst(Link h, Link s);
      //h指向头结点，将s所指结点插在首元结点之前
Status DelFirst(Link h, Link &q);
      //h指向头结点，删除首结点并以q返回其指针
Status Append(LinkList &L, Link s);
      //将指针s所引导的一串结点链接到表尾结点之后
Status Remove(LinkList &L, Link &q);
      //删除线性链表L的表尾结点并以q返回
Status InsBefore(LinkList &L,Link &p,Link s);
      //将s所指结点插在p所指结点之前，
      //修改p指针，令其指向新插入的结点
```

接下页

§2.3 线性表的链式表示和实现

```
Status InsAfter(LinkList &L,Link &p,Link s);
      //将s所指结点插在p所指结点之后,
      //修改p指针，令其指向新插入的结点
Status SetCurElem(Link &p, ElemType e);
      //用e更新p所指结点的值
Status ListEmpty(LinkList L);
      //判表空
int ListLength(LinkList L);
      //求表长
Position GetHead(LinkList L);
      //返回线性链表L头结点指针
```

接下页

§2.3 线性表的链式表示和实现

```
Position GetLast(LinkList L);
      //返回线性链表L尾结点指针
Position PriorPos(LinkList L, Link p);
      //返回p所指结点的直接前驱的指针
Position NextPos(LinkList L, Link p);
      //返回p所指结点的直接后继的指针
Status LocatePos(LinkList L, int i, Link &p);
      //用p返回L中第i个结点的指针，若i非法则返回ERROR
```

接下页

§2.3 线性表的链式表示和实现

```
Position LocateElem(LinkList L, ElemType e,
        Status (*compare)(ElemType,ElemType));
      //返回线性链表L中第一个与e满足compare关系的
      //元素的位置。如果这样的结点不存在，则返回

Status ListTraverse(LinkList L,
                    Status (*visit)());
      //用访问函数visit()遍历L中的每个元素
```

线性链表基本运算完

§2.3 线性表的链式表示和实现 • 双向链表

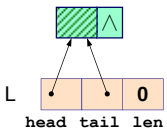
运算举例：构造一个结点

```
Status MakeNode ( Position &p， ElemType e ) {
  //申请一个链表结点，data域赋以e值，
  //变参p传回结点的指针
   p=(Link)malloc(sizeof(struct LNode));
   if (!p) return ERROR;
   p->data=e;
   return OK ;
}
```

§2.3 线性表的链式表示和实现

运算举例：初始化空表

```
Status InitList(LinkList &L){
   L.head=L.tail=(Link)malloc(sizeof(LNode));
   if(!L.head) return OVERFLOW;
   L.head->next=NULL;
   L.len=0;
   return  OK;
}//InitList
```



§2.3 线性表的链式表示和实现

运算举例：销毁一个线性链表

```
Status DestroyList ( LinkList &L ) {
  //销毁L，L将不再存在
   p=L.head;  //p指向头结点
   while(p){
        q=p->next; free(p);
        p=q;
   }
   L.head=L.tail=NULL; L.len=0;
   return OK ;
}
```