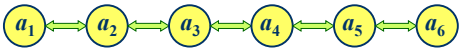
第二章 线性表

- 线性表
- 顺序表
- 链表
- 顺序表与链表的比较
- 单链表的应用：多项式
- 知识扩展：静态链表

线性表 (Linear List)

- 线性表的定义和特点
  - 定义 n（≥0）个数据元素的有限序列，记作
    $(a_1, a_2, \ldots, a_n)$
    a_i是表中数据元素，n 是表长度。
  - 特点 线性排列
    - ✓ 除第一个元素外，其他每一个元素有一个且仅有一个直接前趋。
    - ✓ 除最后一个元素外，其他每一个元素有一个且仅有一个直接后继。



- 理解线性表的要点是
  - a) 表中元素具有逻辑上的顺序性，在序列中各元素排列有其先后次序，有唯一的首元素和尾元素。
  - b) 表中元素个数有限。
  - c) 表中元素都是数据元素。即每一表元素都是原子数据，不允许“表中套表”。
  - d) 表中元素的数据类型都相同。这意味着每一表元素占有相同数量的存储空间。

顺序表 (Sequential List)

- 顺序表的定义和特点
  - 定义 将线性表中的元素相继存放在一个连续的存储空间中，即构成顺序表。
  - 存储 它是线性表的顺序存储表示，可利用一维数组描述存储结构。
  - 特点 元素的逻辑顺序与物理顺序一致。
  - 访问方式 可顺序存取，可按下标直接存取。

	0	1	2	3	4	5
data	25	34	57	16	48	09

顺序表的连续存储方式

$LOC(i) = LOC(i-1) + l = a + i * l,$

LOC 是元素存储位置，l 是元素大小

	0	1	2	3	4	5	6	7	8	9
a	35	27	49	18	60	54	77	83	41	02

l l l l l l l l l l

$a+i*l$

$$LOC(i) = \begin{cases} a, & i = 0 \\ LOC(i-1)+l = a+i*l, & i > 0 \end{cases}$$

顺序表的静态结构定义

```
#define maxSize 100          //最大允许长度
typedef int DataType;        //元素的数据类型

typedef struct {
    dataType data[maxSize];  //存储数组
    int n;                   //当前表元素个数
} SeqList;
```

- 顺序表静态定义，假定 L 是一个类型 SeqList 的顺序表，一般用 L.data[i] 来访问它。
- 表一旦装满，不能扩充。

顺序表的动态结构定义

```
#define initSize 100         //最大允许长度
typedef int DataType;        //元素的数据类型

typedef struct {
    DataType *data;          //存储数组
    int n;                   //当前表元素个数
    int maxSize;             //表的最大长度
} SeqList;
```

- 顺序表动态定义，它可以扩充，新的大小计入数据成员maxSize中。

顺序表基本运算的实现

- 构造一个空的顺序表

```
void InitList (SeqList& L) {
    L.data = ( DataType* ) malloc ( initSize*sizeof
             ( DataType ));
    if ( L.data == NULL)
       { printf ("存储分配失败!\n");  exit (1); }
    L.n = 0;  L.maxSize = initSize;
}
```


- 引用型参数 & 的使用
  - 例如， void InitList (SeqList& L)
  - 引用型参数“&”是把形参 L 看作是实际变量（一个表）的别名，在函数体内对 L 的操作将直接对实际变量的操作。
  - 好处之一是可在函数体内像普通变量那样对 L 操作，使得操作简单。
  - 好处之二是可直接从实际变量得到操作结果。
  - 好处之三是不必创建实际变量的副本空间。

- 按值查找：在顺序表中从头查找结点值等于给定值 x 的结点

```
int Find ( SeqList& L, DataType x ) {
  for ( i = 0; i < L.n; i++ )
    if ( L.data[i] == x ) return i;    //查找成功
  return -1;                           //查找失败
}
```

- 注意，如果表中元素序号从1开始，则第 i 个元素存储于第 $i-1$ 个数组元素位置，函数返回位置比元素序号小1。

查找算法性能分析

- 查找成功的平均比较次数

$$ACN = \sum_{i=0}^{n-1} p_i \times c_i$$

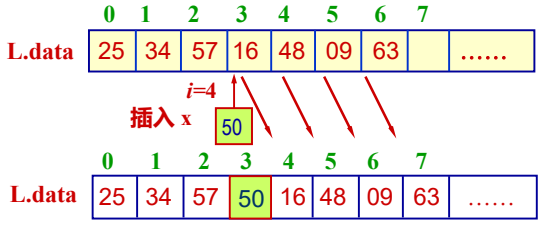
- 若查找概率相等，则

$$ACN = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}(1+2+\cdots+n) = \frac{1}{n} * \frac{(1+n)*n}{2} = \frac{1+n}{2}$$

- 查找不成功　数据比较 n 次。

插入新元素

```
bool Insert ( SeqList& L, DataType x, int i ) {
  //在表中第 i (1≤i≤n+1) 个位置插入新元素 x
    if ( L.n == L.maxSize ) return false;
    if ( i < 1 || i > L.n+1 ) return false;
    for ( int j = L.n-1; j >= i-1; j--)
        L.data[j+1] = L.data[j];
    L.data[i-1] = x;        //实际插在第i-1个位置
    L.n++; return true;  //插入成功
}
```

- 插入时平均移动元素个数AMN

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

删除表元素

```
bool Remove ( SeqList& L, int i, DataType& x ) {
//在表中删除第 i 个元素，通过 x 返回其值
    if ( L.n > 0 && i > 0 && i <= L.n ) {
        x = L.data[i-1];
        for (int j = i; j < L.n; j++)
            L.data[j-1] = L.data[j];
        L.n--; return true;     //删除成功
    }
    else return false;              //删除失败
}
```


表项的删除

- 删除时平均移动元素个数AMN

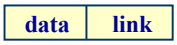
$$AMN = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{1}{n}\frac{(n-1)n}{2} = \frac{n-1}{2}$$

链表（Linked List）

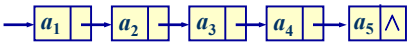
- 线性链表是线性表的链接存储表示。元素之间的逻辑顺序是通过各结点中的链接指针来指示的。
- 线性链表分类：
  - 单链表
  - 循环链表
  - 双向链表
- 链表中第一个元素结点称为首元结点，最后一个元素称为尾结点。首元结点不是头结点。

单链表 (Singly Linked List)

- 特点
  - 每个元素(表项)由结点(Node)构成。

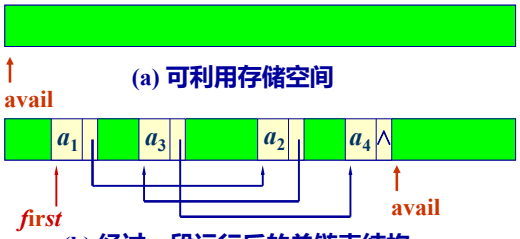


  - 线性结构



  - 结点可以连续，可以不连续存储
  - 结点的逻辑顺序与物理顺序可以不一致
  - 表可扩充

单链表的存储映像



单链表的结构定义

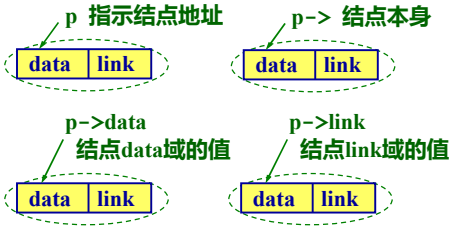
```
typedef char DataType;

typedef struct node {              //链表结点
    DataType data;                 //结点数据域
    struct node * link;            //结点链域
} LinkNode, *LinkList;             //链头指针
```

- 使用时定义实际链表，只需定义链表头指针

```
LinkList first;                    //链表头指针
```


有关指针的一些概念

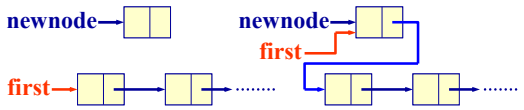


- 在链表中，如果没有定义重载“++”函数，不能使用 p++ 这样的语句进到逻辑上的下一个结点。一般用 p = p->link 进到下一结点。

单链表中的插入与删除

- 插入的考虑
  - 第一种情况：在第 1 个结点前插入：

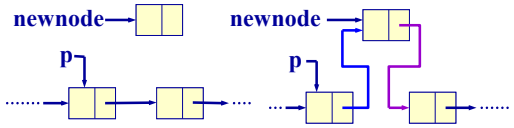
```
newnode->link = first ;
first = newnode;                   //修改头指针
```



- 第二种情况：在链表中间插入：

首先定位指针 p 到插入位置，再将新结点插在其后：

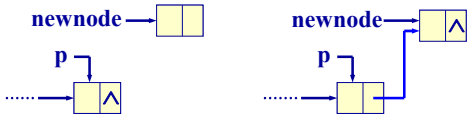
```
newnode->link = p->link;
p->link = newnode;
```



- 第三种情况：在链表末尾插入：

首先定义指针 p 到尾结点位置，再将新结点插在其后，新结点成为新的尾结点。

```
newnode->link = p->link;
p->link = newnode;
```



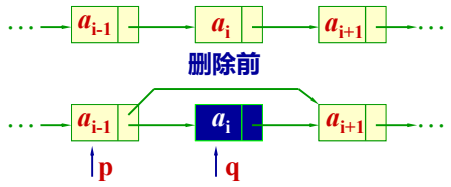
```
bool Insert ( LinkList& first, int i, dataType x ) {
//在链表第 i (≥1)个结点处插入新元素 x
   LinkNode *newnode;
   newnode = (LinkNode *) malloc (sizeof (LinkNode));
   newnode->data = x;
   if ( first == NULL || i == 1 )      //链空或插入首元
      { newnode->link = first;  first = newnode; }
   else {
      LinkNode *p = first, *pr;  int k = 1;
      while ( p != NULL && k < i-1 )
         { pr = p;  p = p->link;  k++; }  //找第 i-1个结点
```



```
      if ( p == NULL && first != NULL ) p = pr;
         //链太短，插在链尾，p收缩到链尾
      newnode->link = p->link;
      p->link = newnode;
   }
   return true;
}
```

- 考虑为何在参数表中 first 定义为引用型。
- 在链表插入时不考虑表满问题，但可能初始时表空，此时新建结点将成为表中的首元结点。

- 删除的考虑
  - 第一种情况: 删除表中第 1 个元素
  - 第二种情况: 删除表中或表尾元素



在单链表中删除含 a_i 的结点

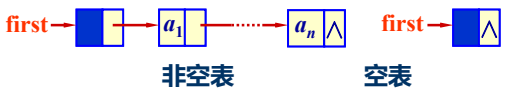

```
bool Remove (LinkList& first,  int i, dataType& x) {
//在链表中删除第 i 个结点。如果要删除表中第 1 个
//结点，需要改变表头指针，所以 first 定义为引用
//型，被删元素的值通过引用型参数 x 返回
   LinkNode *p, *q;   int k;
   if ( i == 0 ) return false;   //无效的删除位置
   if ( i == 1 ) { q = first;  first = first->link; }
   else {                          //删除链中或链尾结点
      p = first;  int k = 1;       //找第 i-1个结点
      while ( p != NULL && k < i-1 )
            { p = p->link; k++; }
```

?


```
   if ( p == NULL || p->link == NULL ) {
        printf ( "无效的删除位置!\n" );
        return false;          //链太短，没有删除结点
   }
   else {                      //删除结点p->link
        q = p->link;           //重新链接, 摘下*q
        p->link = q->link;
   }
   x = q->data;  free (q);     //删除*q
   return true;
}
```


带头结点的单链表

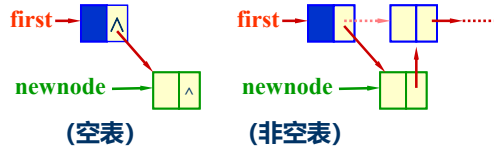
- 头结点位于表的最前端，本身不带数据，仅标志表头。
- 设置头结点的目的是
  - 统一空表与非空表的操作
  - 简化链表操作的实现。



非空表　　空表

前插法建立单链表

- 从一个空表开始，重复读入数据：
  - 生成新结点
  - 将读入数据存放到新结点的数据域中
  - 将该新结点插入到链表的前端
- 直到读入结束符为止。



(空表) (非空表)


```
void insertFront ( LinkList& first, DataType endTag ) {
   DataType val;  LinkNode *s;
   scanf ( "%d", &val );            //读入一数据
   while ( val != endTag ) {        //若不是endTag
      s = (LinkNode *) malloc ( sizeof (LinkNode ));
      s->data = val;                //创建新结点
      s->link = first->link;        //插入到表前端
      first->link = s;
      scanf ( "%d", &val );         //读入下一数据
   }
}
```

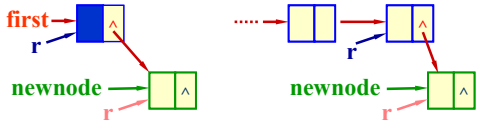


```
void main() {
   LinkList L;  DataType endTag;
   L = (LinkNode *) malloc ( sizeof (LinkNode ));
   L->link = NULL;                  //建立头结点并置空
   scanf ( "%d", endTag );          //序列输入结束标记
   insertFront ( L, endTag );       //前插法建表
   printList ( L );
}
```

- 要求在程序首部使用 #include <stdio.h>实现输入scanf和输出printf操作，使用#include <stdlib.h>实现动态存储分配和回收操作。

后插法建立单链表

- 每次将新结点加在插到链表的表尾；
- 设置一个尾指针 r，总是指向表中最后一个结点，新结点插在它的后面；
- 尾指针 r 初始时置为指向表头结点地址。



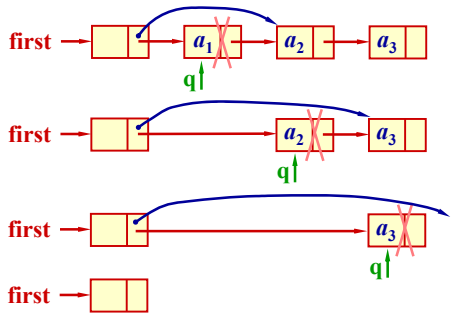
(空表) (非空表)


```
void insertRear ( LinkList& first, DataType endTag ) {
   DataType val;
   LinkNode *s, *rear = first;      //rear指向表尾
   scanf ( "%d", &val );            //读入一数据
   while ( val != endTag ) {
      s = (LinkNode *) malloc ( sizeof (LinkNode ));
      s->data = val;                //创建新结点并赋值
      rear->link = s;  rear = s;    //插入到表尾
      scanf ( "%d", &val );         //读入下一数据
   }
   rear->link = NULL;               //表收尾
}
```


- 主程序与前插法建表类似，只需把insertFront改为insertRear即可。
- 使用前插法建立链表，每次新元素插入在表头，数据元素的链接顺序与输入顺序完全相反。
- 使用后插法建立链表，每次新元素插入在表尾，数据元素的链接顺序与输入顺序完全一致。

```
void printList ( LinkList& first ) {   //输出链表
   for ( LinkNode *p = first; p; p = p->link )
      printf ( "%d ", p->data );
   printf ( "\n" );
}
```

单链表清空

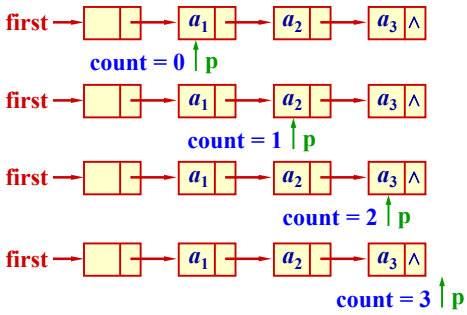


95-37

```
void makeEmpty ( LinkList first ) {
//删去链表中除表头结点外的所有其他结点
   LinkNode *q;
   while ( first->link != NULL ) {        //链不空时
      q = first->link;                    //q指示首元结点
      first->link = q->link;              //摘下q指示结点
      free (q);                           //释放它
   }
}
```

95-38

计算单链表长度



95-39

```
int Length ( LinkList first ) {
   LinkNode *p = first->link;
   //检测指针 p 跳过表头结点指示首元结点
   int count = 0;
   while ( p != NULL ) {          //逐个结点计数
      p = p->link;  count++;
   }
   return count;
}
```

- 结点计数与链表指针前移同步进行。

95-40

在单链表中按值查找

```
LinkNode *Search ( LinkList first, DataType x ) {
//在链表中从头搜索其数据值为 x 的结点
   LinkNode * p = first->link;      //p为检测指针
   while ( p != NULL && p->data != x )
      p = p->link;
   return p;
}
```

- 查找成功返回结点地址；查找不成功返回空。
- 注意，while循环条件p != NULL 和 p->data != x 不能错位，考虑为什么。

95-41

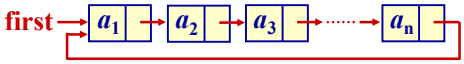
在单链表中按序号查找（定位）

```
LinkNode *Locate ( LinkList first, int i ) {
//返回表中第 i 个元素的地址，头结点为 0 号
   if ( i < 0 ) return NULL;          // i 值不合理
   LinkNode * p = first;  int k = 0;
   while ( p != NULL && k < i )
      { p = p->link;  k++; }          //找第 i 个结点
   return p;
      //当返回地址非空则为第 i 个结点地址
      //否则返回NULL
}
```

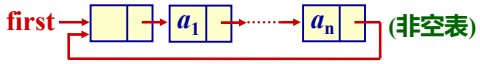
95-42

循环链表 (Circular List)

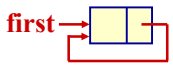
- 循环单链表是单链表的变形。链表尾结点的 link 指针不是 NULL，而是指向了表的前端。



- 为简化操作，在循环单链表中往往加入头结点。



(非空表)

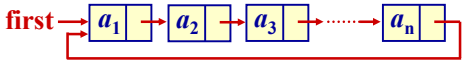


(空表，也要保留表头)

- 循环单链表的判空条件是：first->link == first。
- 循环单链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其他结点的地址。
- 在搜寻过程中，没有一个结点的 link 域为空。

```
for ( p = first->link; p != first; p = p->link )
    do S;
```

- 循环单链表的所有操作的实现类似于单链表，差别在于检测到链尾，指针不为NULL，而是回到链头。



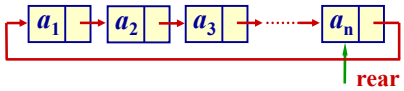
循环单链表的结构定义

```
typedef int DataType;
typedef struct node {          //循环链表定义
    DataType data;             //结点数据
    struct node *link;         //后继结点指针
} CircNode, *CircList;
```

- 在链表中将指针 p 定位于第 i 个结点的操作为

```
CircNode *p = first;  int k = 0;        //first是头结点
while ( p->link != first && k < i )     //回到头结点失败
    { p = p->link;  k++; }              //否则 p 指到目标
```

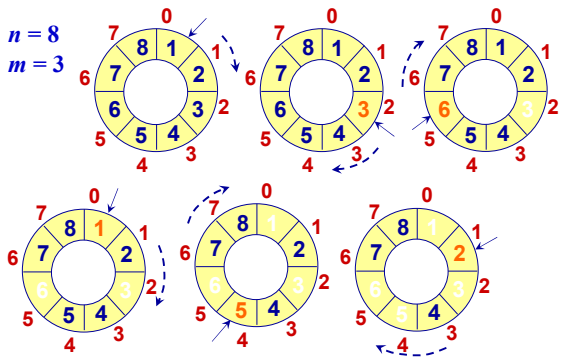

带尾指针的循环链表

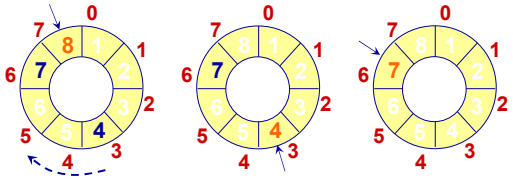


- 如果插入与删除仅在链表的两端发生，可采用带表尾指针的循环链表结构。
  - 在表尾可直接插入新结点，时间复杂性 O(1);
  - 在表尾删除时要找前趋，时间复杂性 O(n);
  - 在表头插入相当于在表尾插入;
  - 在表头可直接删除，时间复杂性 O(1)。

应用:求解约瑟夫问题

- 问题的提法
  - n 个人围成一个圆圈，首先第 1 个人从 1 开始，一个人一个人顺时针报数，报到第 m 个人，令其出列。然后再从下一 个人开始，从 1 顺时针报数，报到第 m 个人，再令其出列，…，如此下去，直到圆圈中只剩一个人为止。此人即为优胜者。
  - 首先用首尾衔接的一维数组来组织。
  - 例如 $n = 8$ $m = 3$





- 如果采用一维数组A[n]存储，每次出列一个人，n 减 1。如果第 k 个人出列，从 k+1 到 n-1，所有人前移一个位置。
- 这样做，不是不可行，但每出列一个人，剩下所有人都要前移，时间花费很多；其次，在数组中循环计数，算法也比较复杂。

95-49

- 如果采用不带头结点的循环单链表，可以不移动元素，仅修改链接指针即可。
- 算法描述如下。

```
#include "CircList.h"
void Josephus ( CircList& L, int n, int m ) {
//在有n个结点的循环单链表中从链头开始报数,
//每次报到第m个人该人出列，然后从下一个人
//开始继续报数……
  CircNode *p = L->link, *pre = NULL;
  int i, j;
```

95-50

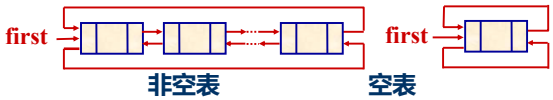
```
  for ( i = 0; i < n-1; i++ ) {          //执行n-1次
    for ( j = 1; j < m; j++ )            //报数m-1
      { pre = p;  p = p->link; }         //沿链前行
    printf ( "出列的人是%d\n", p->data );
    pre->link = p->link;  free (p);   //从链中删去
    p = pre->link;     //p进到被删结点的下一结点
  }
  printf ( "最终剩下的人是%d\n", p->data );
}
```

95-51

双向链表 (Doubly Linked List)

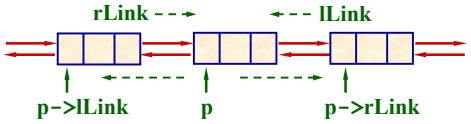
- 双向链表是指在前趋和后继方向都能遍历的线性链表。双向链表每个结点的结构为：

- 双向链表通常采用带头结点的循环双链表形式。每一个结点处于两个链中。



95-52

- 结点指向
  - p->lLink 指示结点 p 的前趋结点
  - p->rLink 指示结点 p 的后继结点
  - p->lLink->rLink指示结点 p 的前趋结点的后继结点，即结点 p 本身
  - p->rLink->lLink指示结点 p 的后继结点的前趋结点，即结点 p 本身



95-53

循环双链表的定义

```
typedef int DataType;          //每个元素的类型
typedef struct node {          //结点定义
  DataType data;               //数据
  int freq;                    //访问计数
  struct node *lLink, *rLink;  //指针
} DblNode, *DblList;           //双向链表
```

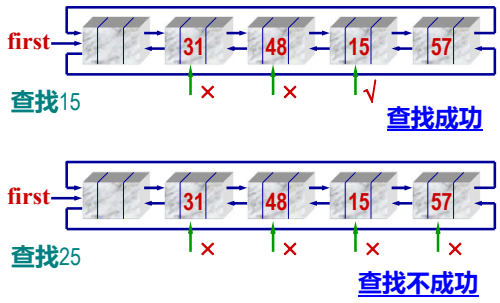
- 单链表寻找结点后继的时间复杂度是$O(1)$，寻找结点前趋的时间复杂度是$O(n)$；而双向链表寻找后继和前趋的时间复杂度都是$O(1)$。

95-54

建立空的循环双链表

```
void initDblList ( DblList &first ) {
    first = (DblNode *) malloc ( sizeof (DblNode));
    if ( first == NULL ) {            //建立头结点
        printf ("存储分配错!\n" );
        exit (1);
    }
    first->lLink = first->rLink = first;
    first->freq = 0;
}
```


循环双链表的查找图示



循环双链表的查找

- 若在以 first 为头结点的循环双链表中搜寻含 x 的结点，要区分是在后继方向还是在前趋方向。当查找结果是 p 指向头结点，则查找失败，否则函数返回找到结点的地址。
- 后继方向

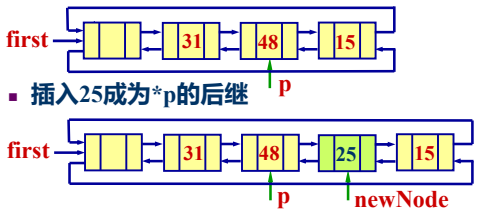
```
DblNode *p = first->rLink;
while ( p != first && p->data != x ) p = p->rLink;
return p;
```

- 前趋方向类似，只需把 rLink 换成 lLink。

定位：查找第 i 个结点在链表中的位置

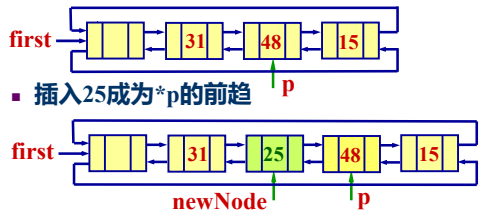
```
DblNode *Locate ( DblList first, int i, int d ) {
    if ( i < 0 ) return NULL;
    if ( i == 0 ) return first;
    DblNode *p = ( d == 0 ) ? first->lLink : first->rLink;
    for ( int j = 1; j < i; j++ )
        if ( p == first ) break;
        else p = ( d == 0 ) ? p->lLink : p->rLink;
                //d = 0前趋方向, d = 1后继方向
    return ( p != first ) ? p : NULL;
}
```


循环双链表的插入 (后继链)




```
newNode->rLink = p->rLink;
p->rLink = newNode;
newNode->lLink = p;
newNode->rLink->lLink = newNode;
```


循环双链表的插入 (前趋链)




```
newNode->lLink = p->lLink;
p->lLink = newNode;
newNode->rLink = p;
newNode->lLink->rLink = newNode;
```

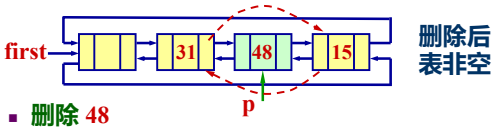
```
bool Insert (DblList first, DataType x, int i, int d ) {
    DblNode *p = Locate (first, i-1, d);
    if ( p == NULL ) return false;
    DblNode *newNode = (DblNode *) malloc
        (sizeof (DblNode));        //分配结点
    newNode->data = x;
    if ( d != 0 ) {              //在后继方向插入到*p右方
        newNode->rlink = p->rlink;
        p->rlink = newNode;
        newNode->llink = p;
        newNode->rlink->llink = newNode;
```



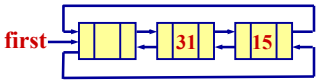
```
    }
    else {                       //在前趋方向插入到*p左方
        newNode->llink = p->llink;
        p->llink = newNode;
        newNode->rlink = p;
        newNode->llink->rlink = newNode;
    }
    return true;
}
```

- 两个方向的插入语句类似，只是 lLink 与 rLink 互换了一下。

循环双链表的删除

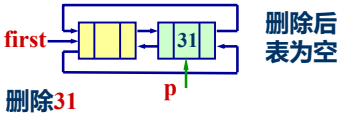


- 删除 48

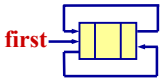



```
p->rlink->llink = p->llink;
p->llink->rlink = p->rlink;
```


循环双链表的删除



- 删除31




```
p->rlink->llink = p->llink;
p->llink->rlink = p->rlink;
```



```
bool Remove (DblList first, int i, int d, DataType& x) {
//删除在 d 指明方向的第 i 个结点, x 返回其值
    DblNode *p = Locate (first, i, d);
        //指针定位于删除结点位置
    if ( p == NULL) return false;        //不能删除
    p->rLink->lLink = p->lLink;
    p->lLink->rLink = p->rLink;
        //将被删结点 p 从链上摘下
    x = p->data;  free (p);              //删去
    return true;
}
```


顺序表与链表的比较

- 基于空间的比较
  - 存储分配的方式
    - 顺序表的存储空间可以是静态分配的，也可以是动态分配的。
    - 链表的存储空间是动态分配的。
  - 存储密度 = 结点数据本身所占的存储量/结点结构所占的存储总量
    - 顺序表的存储密度 = 1
    - 链表的存储密度 < 1

- 基于时间的比较
  - 存取方式
    - 顺序表可以随机存取，也可以顺序存取。
    - 链表只能顺序存取。
  - 插入/删除时移动元素个数
    - 顺序表平均需要移动近一半元素。
    - 链表不需要移动元素，只需要修改指针。
    - 若插入/删除仅发生在表的两端，宜采用带尾指针的循环链表。

95-67

线性表链式存储方式的比较

操作名称 / 链表名称	找表头结点	找表尾结点	找P结点前驱结点
带头结点单链表L	L->next 时间耗费O(1)	一重循环 时间耗费O(n)	顺P结点的next域无法找到P结点的前驱
带头结点循环单链表（头指针）L	L->next 时间耗费O(1)	一重循环 时间耗费O(n)	顺P结点的next域可以找到P结点的前驱时间耗费O(n)
带尾指针的循环单链表R	R->next O(1)	R 时间耗费O(1)	顺P结点的next域可以找到P结点的前驱时间耗费O(n)
带头结点双向循环链表L	L->next O(1)	L->prior 时间耗费O(1)	P->prior 时间耗费O(1)

一元多项式 (Polynomial)

$$P_n(x) = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n = \sum_{i=0}^{n} c_ix^i$$

- n 阶一元多项式 $P_n(x)$ 有 n+1 项。
  - 系数 $c_0, c_1, c_2, \ldots, c_n$
  - 指数 0, 1, 2, …, n。按升幂排列
- 多项式求值通常使用如下方式

$$P_n(x) = c_nx^n + c_{n-1}x^{n-1} + \cdots + c_1x + c_0$$
$$= ((\ldots((c_nx + c_{n-1})x + c_{n-2})x + \cdots)x + c_1)x + c_0$$

从最内层括号开始，逐层向外计算。

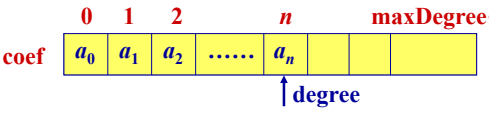
95-69

多项式的存储表示

第一种：静态数组表示

```
const int maxDegree = 20;        //最大允许阶数
typedef struct Polynomial {      //多项式结构定义
    int degree;                  //实际阶数
    float coef [maxDegree+1];    //系数数组
}
```

- 例 $P_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ 的静态表示



95-70

- 在这种存储表示中，x^i 的系数 c_i 存放于 coef[i]，适用于指数连续排列的多项式。
- 它的优点是可以简化操作;
- 缺点是对于指数不全的稀疏多项式（大于n / 2的项空）如

$$P_{101}(x) = 3 + 5x^{50} - 14x^{101}$$

coef 数组长度达到 102，实际只有 3 个非零项，不经济。

第二种：只保存非零系数项

- 这种存储表示适用于稀疏多项式。
- 在保留非零系数的同时必须保留它的指数。

95-71

```
#define DefaultSize 20;          //默认数组大小
typedef struct data {            //多项式的项定义
    float coef;                  //系数
    int exp;                     //指数
};
typedef struct Polynomial {      //多项式结构定义
    int maxSize;                 //数组最大保存项数
    int n;                       //实际项数
    data *Term;                  //项数组
}
```

- 动态定义项数组，使用前必须初始化，分配空间。

95-72

	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

- 对于每一个非零（系数）项，保存它的系数 a_i 和指数 e_i。这样，若有一个多项式 pl，可初始化为

```
pl.maxSize = DefaultSize;
pl.Term = new data[pl.maxSize];
pl.count = n;
pl.Term[i].coef = a_i, pl.Term[i].exp = e_i,
                  0≤i≤n
```


多项式的链表表示

- 这是第三种存储表示。
- 用数组定义的多项式存储结构不易扩充，在执行多项式运算时可能会增减许多项，采用链表表示将会提高运算的效率。
- 在多项式的链表表示中每个结点 data 的构成为：

  Term = { coef, exp }
- 每个结点实际上有三个域，如图：

多项式链表的结构定义

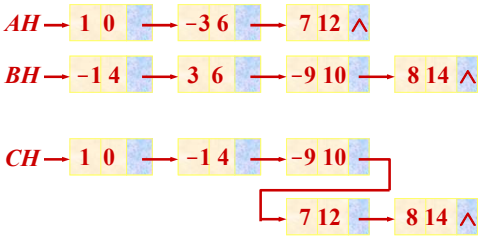
```
typedef struct node {          //多项式数据定义
    float coef;                //系数
    int exp;                   //指数
    struct node * link;        //链接指针
} Term, *Polynomial;
```

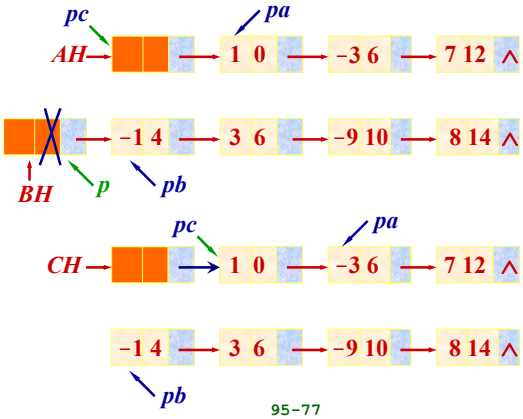
- 多项式的链表表示的优点是：
  - 多项式的项数可以动态地增长，不存在存储溢出问题。
  - 插入、删除方便，不移动元素。

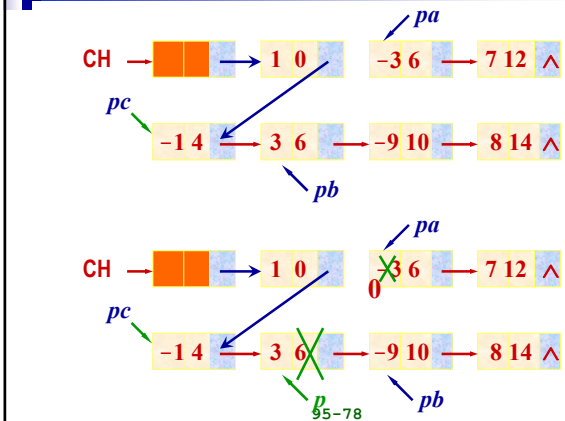
多项式相加的链表运算

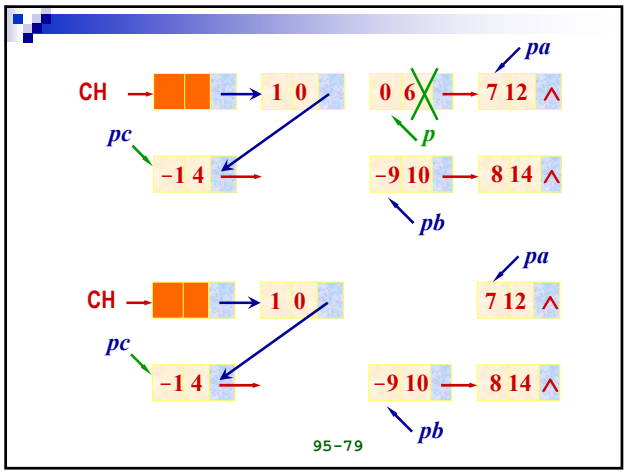
$AH = 1 - 3x^6 + 7x^{12}$

$BH = -x^4 + 3x^6 - 9x^{10} + 8x^{14}$

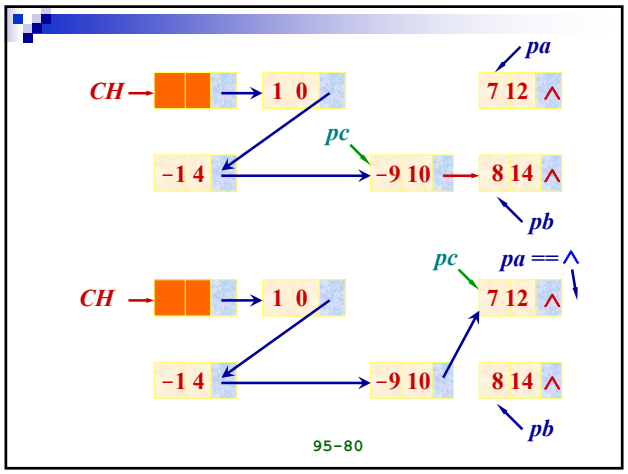




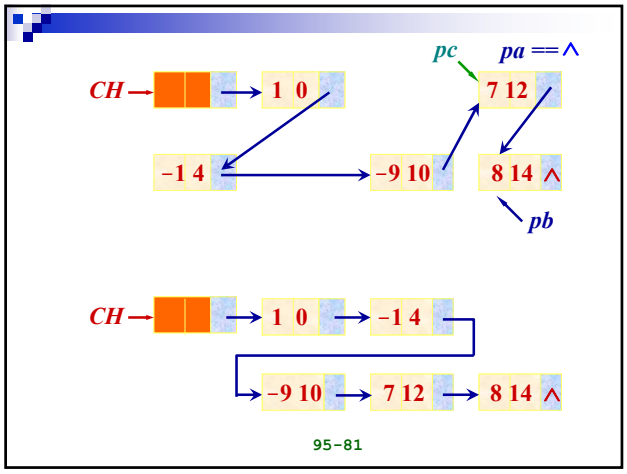




95-79



95-80



95-81

多项式相加的链表实现

```
void AddPolynomial ( Polynomial& A,
    Polynomial& B, Polynomial& C ) {
//两个带头结点的按升幂排列的多项式A与B相加，
//返回结果多项式链表的表头指针 C，结果不另外
//占用存储, 覆盖 A 和 B 链表
    Term *pa, *pb, *pc, *p, *s;  double a;  int b;
    pc = C;                  //设结果链表C只有头结点
    pa = A->link;            //多项式 A 的检测指针
    pb = B->link;            //多项式 B 的检测指针
```

95-82

```
while ( pa != NULL && pb != NULL ) {
    a = pa->coef;
    if ( pa->exp == pb->exp ) { //对应项指数相等
        a = pa->coef+pb->coef; //系数相加
        p = pb;  pb = pb->link;  free (p);
        //指数相等的结点仅保留一个加入结果链
        if ( fabs (a) > 0.001 ) {     //相加不为零
            pa->coef = a;  pc->link = pa;  pc = pa;
            pa = pa->link;            //加入结果链
        }
        else                          //相加为零,该项不要
```

95-83

```
            { p = pa;  pa = pa->link;  free (p); }
    }
    else if ( pa->exp > pb->exp ) {    //pa指数大
        pc->link = pb;  pc = pb;
        pb = pb->link;
    }
    else {                             //pb指数大
        pc->link = pa;  pc = pa;
        pa = pa->link;
    }
} //while结束
```

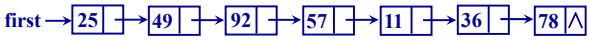
95-84

```
    if ( pa != NULL ) pc->link = pa;
    else pc->link = pb;        //剩余部分链入 C 链
}
```

- 本算法与教材上的算法略有差别，因为它利用了原来两个多项式链表的空间，没有另外占用存储空间。相加后两个链表都已破坏。
- 算法轮流检测两个链表，对每个结点处理一次。若两个链表长度分别为m和n，算法时间复杂度为$O(m+n)$。

知识扩展：静态链表

- 如果为数组中每一个元素附加一个链接指针，就形成静态链表结构。
- 它允许不改变各元素的物理位置，只要重新链接就能够改变这些元素的逻辑顺序。
- 由于它是利用数组定义的，在整个运算过程中存储空间的大小不会变化，因此称之为静态链表。
- 静态链表的每个结点由两个数据成员构成：data域存储数据，link域存放链接指针。所有结点形成一个结点数组，它也可以带有头结点。



动态链表

	0	1	2	3	4	5	6	7
data		25	92	57	36	78	11	49
link	1	7	3	6	5	-1	4	2

静态链表

- 链表的头结点在A[0]，从A[1]起，后面都是链表结点的存放空间，链接指针用数组元素的下标（序号）表示。A[0].link给出链表第一个结点的位置。

静态链表的结构

```
#define maxSize 100        //静态链表大小
typedef int DataType;
typedef struct {
    DataType data;         //结点数据
    int link;              //结点链接指针
} SLNode;
typedef struct {
    SLNode elem[maxSize+1];
    int avail;             //当前可分配空间首地址
} StaticLinkList;
```


静态链表操作的实现

```
void InitList ( StaticLinkList& A ) {
//将链表空间初始化
    A.elem[0].link = -1;
    A.avail = 1;          //当前可分配空间从 1 开始
    for ( int i = 1; i < maxSize-1; i++ )
        A.elem[i].link = i+1;          //构成空闲链接表
    A.elem[maxSize-1].link = -1;  //链表收尾
};
```

- 通过初始化操作，把全部链表空间链接成可利用空间表，将来可按照 avail 指针分配和回收结点。


```
int Length ( StaticLinkList& A ) {
//计算静态链表的长度
    int p = A.elem[0].link;  int count = 0;
    while ( p != -1 ) {
        p = A.elem[p].link;  count++;
    }
    return count;
};
bool IsEmpty ( StaticLinkList& A ) {
//判链表空否?
    return ( A.elem[0].link == -1);
};
```

```
int Search ( StaticLinkList& A, DataType x ) {
//在静态链表中查找具有给定值的结点
   int p = A.elem[0].link;   //指针 p 指向表的首元素
   while ( p != -1 )          //逐个结点检测
      if ( A.elem[p].data == x ) break;
      else p = A.elem[p].link;
   return p;
};
```

95-91

```
int Locate ( StaticLinkList& A, int i ) {
//在静态链表中查找第 i 个结点
   if ( i < 0 ) return -1;        //参数不合理
   if ( i == 0 ) return 0;
   int j = 1,  p = A.elem[0].link;
   while ( p != -1 && j < i )
      { p = A.elem[p].link;  j++; }
                        //循链查找第 i 号结点
   return p;
};
```

95-92

```
int AllocNode ( StaticLinkList& A ) {
//从可利用空间表分配一个可利用结点
   if ( A.avail == -1 ) return -1;
   int q = A.avail;              //分配结点
   A.avail = A.elem[A.avail].link;
   return q;
};
void FreeNode ( StaticLinkList& A, int i ) {
//将结点 i 回收到可利用空间表
   A.elem[i].link = A.avail;
   A.avail = i;
};
```

95-93

```
bool InsertFront ( StaticLinkList& A, DataType x ) {
//在静态链表的表头插入一个新结点
   int p = AllocNode (A);
   if ( p == -1 ) return false;    //结点分配失败
   A.elem[p].data = x;
   A.elem[p].link = A.elem[0].link;
   A.elem[0].link = p;             //在链头链入结点
   return true;
};
```

95-94

```
bool Insert ( StaticLinkList& A, int i, DataType x ) {
//在静态链表A第 i 个结点后面插入新结点
   int p = Locate(A, i);
   if ( p == -1 ) return false;     //找不到结点
   int q = AllocNode (A);           //分配结点
   if ( q == -1 ) return false;     //分配结点失败
   A.elem[q].data = x;
   A.elem[q].link = A.elem[p].link;  //链入
   A.elem[p].link = q;
   return true;
};
```

95-95

```
bool Remove ( StaticLinkList& A, int i,
      DataType& x ) {
//在静态链表A中释放第 i 个结点, 通过 x 返回其值
   int p = Locate(A, i-1);          //找第i-1号结点
   if ( p == -1 ) return false;     //找不到结点
   if ( A.elem[p].link == -1 ) return false;
   int q = A.elem[p].link;          //第i号结点
   x = A.elem[q].data;
   A.elem[p].link = A.elem[q].link;  //摘下第i号结点
   FreeNode (A, q);                 //释放
   return true;
};
```

95-96