数据结构与算法

Data Structures and Algorithms

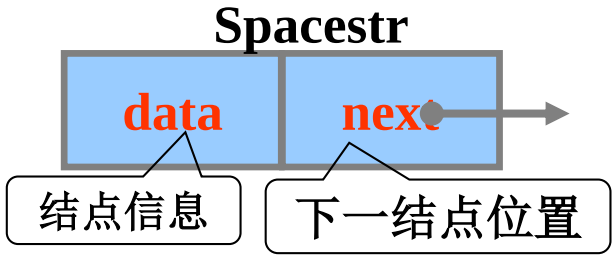
第二部分 线性表

第 三 章

栈和队列

回顾：线性链表 和 栈

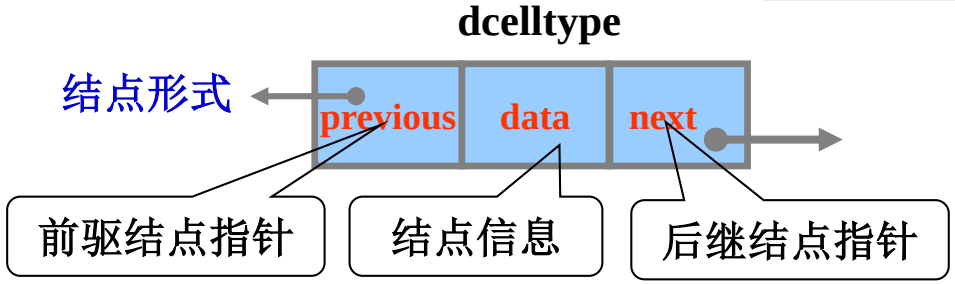
结点形式



1. 线性链表 – 线性表的游标实现

   连续存储空间+动态管理的思想

2. 双向链表



3. 环形链表及双向环形链表

4. 栈

   （1）顺序存储

   （2）链式存储

   （3）栈的而应用

   递归

   数制变换

   迷宫求解

3、表达式求值

表达式：
- 前缀表达式（逆波兰式）
- 中缀表达式
- 后缀表达式（波兰式）

如：

(a + b) * (a － b)
- *+ab－ab
- (a + b) * (a － b)
- ab+ab－*

高级语言中，采用类似自然语言的中缀表达式，但计算机对中缀表达式的处理是很困难的，而对后缀或前缀表达式则显得非常简单。

后缀表达式的特点：

① 在后缀表达式中，变量（操作数）出现的顺序与中缀表达式顺序相同。

② 后缀表达式中不需要括弧定义计算顺序，而由运算（操作符）的位置来确定运算顺序。

I. 将中缀表达式转换成后缀表达式

对中缀表达式从左至右依次扫描每一个字符，由于操作数的顺序保持不变，当遇到操作数时直接输出；为调整运算顺序，设立一个栈用以保存操作符，扫描到操作符时，将操作符压入栈中，进栈的原则是保持栈顶操作符的优先级要高于栈中其他操作符的优先级，否则，栈顶操作符依次退栈并输出，直到表达式结束。遇到“(”进栈，当遇到“)”时，退栈输出直到“)”为止，括号内部依然按照前述操作。

II. 由后缀表达式计算表达式的值

对后缀表达式从左至右依次扫描，与I相反，遇到操作数时，将操作数进栈保留；当遇到操作符时，从栈中退出两个操作数并作相应运算，将计算结果进栈保留；直到表达式结束，栈中唯一元素即为表达式的值。

A+B*(C-D)-E/F ⇨ ABCD-*+EF/-

(a+b)*c-(b+d)/e ⇨ ab+c*bd+e/-

a+b-c ⇨ **ab+c-**

abc-+

哪一个更好呢？ **a+(b-c)**

a*b/c ⇨ **ab*c/**

abc/*

哪一个更好呢？ **a*(b/c)**

例题 已知操作符包括+、-、*、/、（和）。将中缀表达式a+b-a*((c+d)/e-f)+g 转换成等价的后缀表达式ab+acd+e/f-*-g+时，用栈来存放暂时还不能确定运算次序的操作符。若栈初始为空，则转换过程中同时保存在栈中的操作符的最大个数是（A）

A. 5　　B. 7　　C. 8　　D. 11

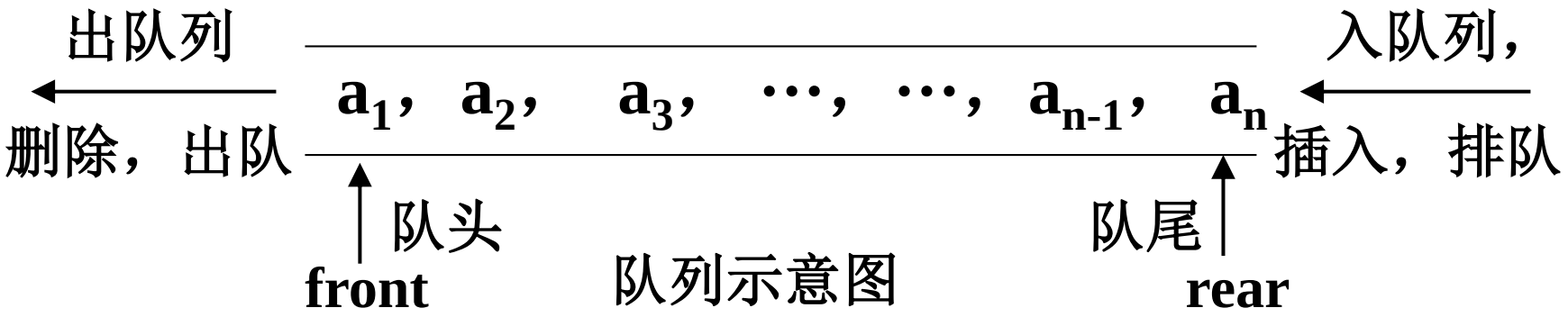
例题 假设栈初始为空，将中缀表达式a/b+(c*d-e*f)/g 转换成等价的后缀表达式时，当扫表到f时，栈中元素依次是（B）

A. +(*-　　B. +(-*　　C. /+(*-*　　D. /+-*

2.4 队列（Queue）

队列是对线性表的插入和删除操作加以限定的另一种限定型数据结构。

【定义】将线性表的插入和删除操作分别限制在表的两端进行，和栈相反，队列是一种先进先出（First In First Out，简称 FIFO 结构）的线性表。



队列示意图

ADT操作： MakeNull(Q)、Front(Q)、EnQueue(x, Q)、DeQueue(Q)、Empty(Q)

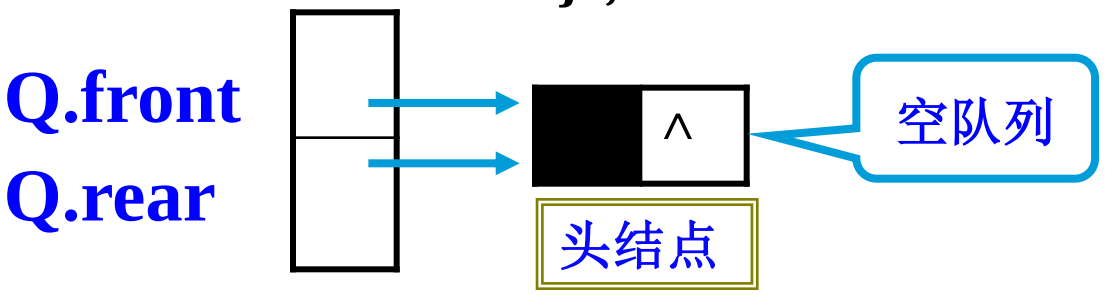
2.4.1 链队列--队列的指针实现

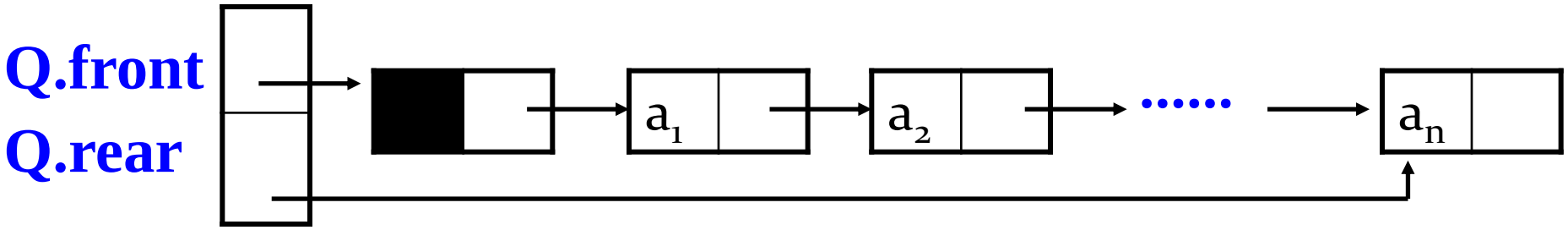
元素结构：

```
struct NODE {
    ElementType data ;
    struct NODE *next ;
};
```

队列的“型”：

```
struct QUEUE {
    struct NODE *front ;
    struct NODE *rear ;
};
```





队列的指针实现示意图

ADT操作：

① MakeNull(&Q)
② boolean Empty(Q) ;
③ ElementType Front (Q) ;
④ EnQueue (x, &Q)
⑤ DeQueue (&Q)

```
④ Void EnQueue (x, &Q )
   { Q.rear→next = New NODE ;
     Q.rear = Q.rear→next ;
     Q.rear→data = x ;
     Q.rear→next = Null ;
   } ;
```

```
⑤ Void DeQueue (&Q )
   { struct NODE *p ;
     if ( Empty( Q ) )
          error ( "空队列" ) ;
     else { p = Q.front→next ;
            Q.front→next = p→next ;
            Delete p ;
      if ( Q.front→next == Null )
                    Q.rear = Q.front ; }
} ;
```

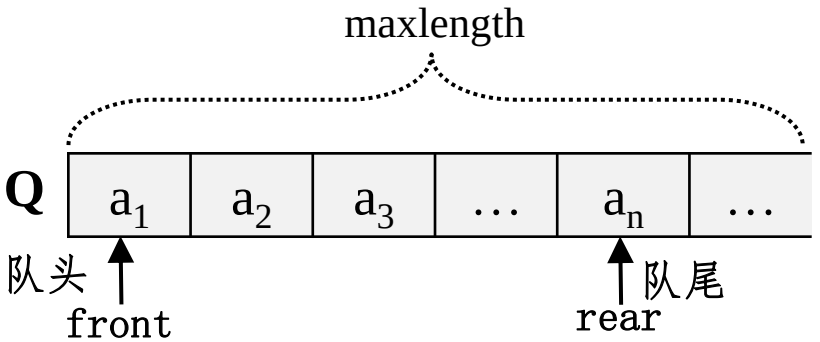
入队和出队分别
修改的是哪一个指针？

(Q.rear == p)?

2.4.2 顺序队列--队列的数组实现

队列的“型”：

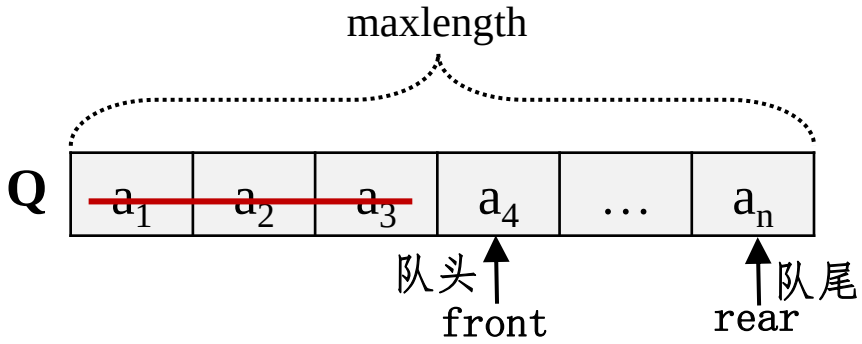
```
struct QUEUE {
   ElementType data [ maxlength ] ;
   int front ;
   int rear ;
  }
```



队列Q状态1

随着不断有元素出队和进队（插入和删除），队列的状态由1变成2；此时a_n占据队列的最后一个位置；第 n+1 个元素无法进队,但实际上，前面部分位置空闲。

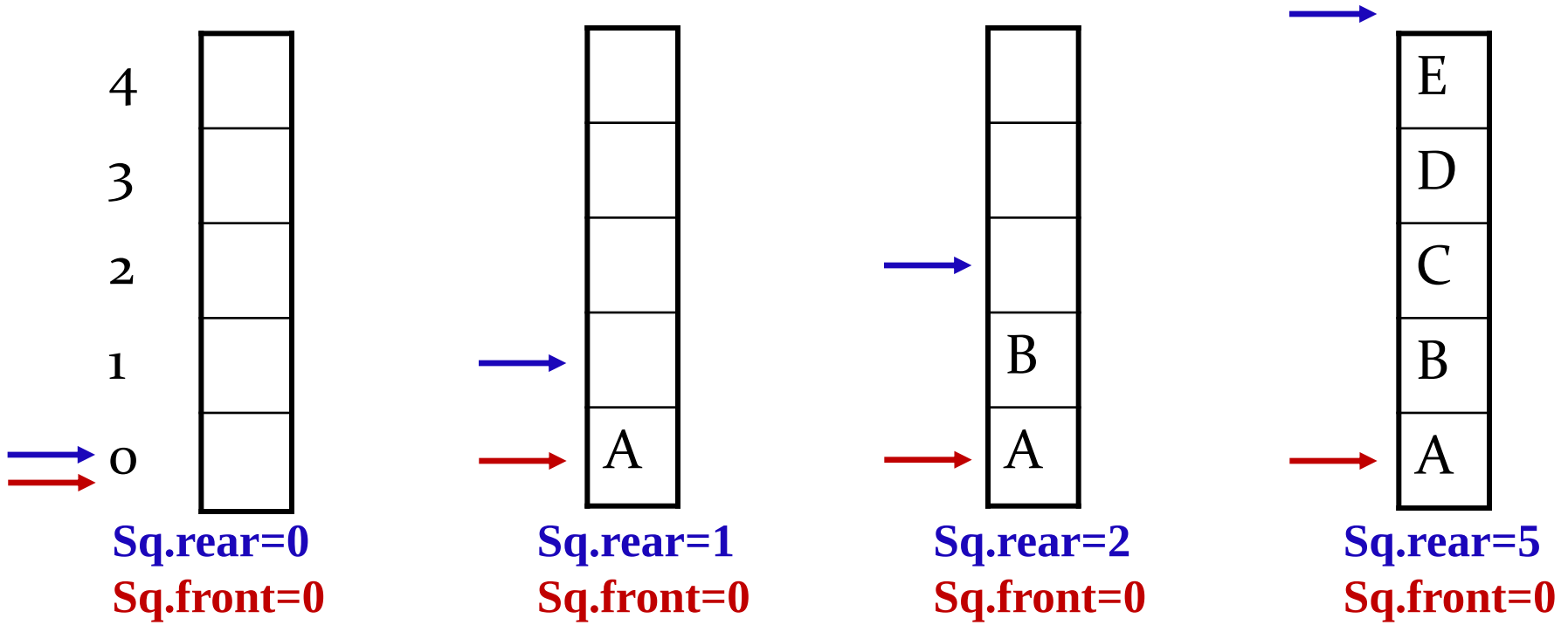
——假溢出



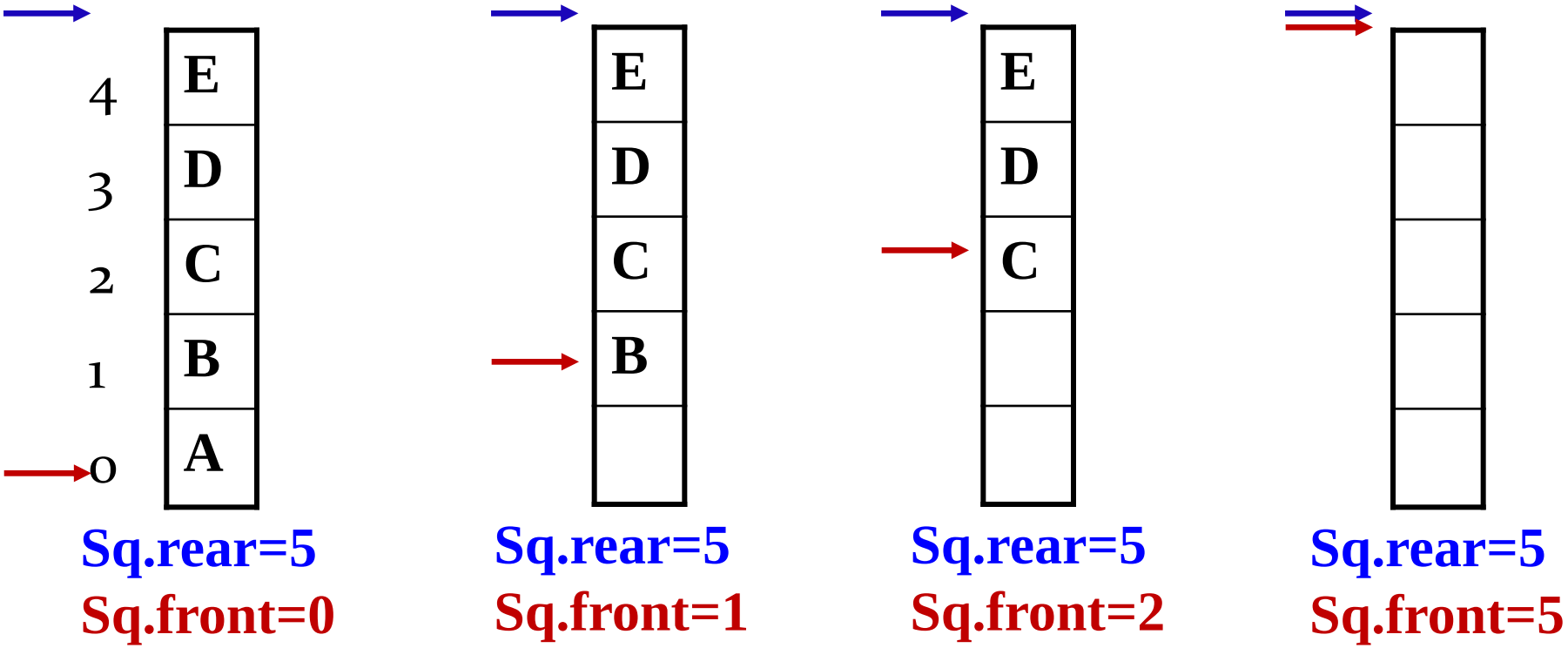
队列Q状态2

顺序队列讨论：

我们约定初始化建空队列时，令front=rear=0，每当插入新的队列尾元素时，尾指针增加1；每当删除队列头元素时，头指针增加1；因此，在非空队列中，头指针始终指向队列头元素，而尾指针始终指向队列尾元素的下一个位置。



- 顺序队列讨论：



讨论结论：在采用一般顺序队列时出现假溢出现象？

当元素被插入到数组中下标最大的位置上之后，队列的空间就用尽了，但此时数组的低端还有空闲空间，这种现象叫做假溢出。

解决假溢出的方法有多种；一是通过不断移动元素位置，每当有元素出队列时，后面的元素前移一个位置，使队头元素始终占据队列的第一个位置。

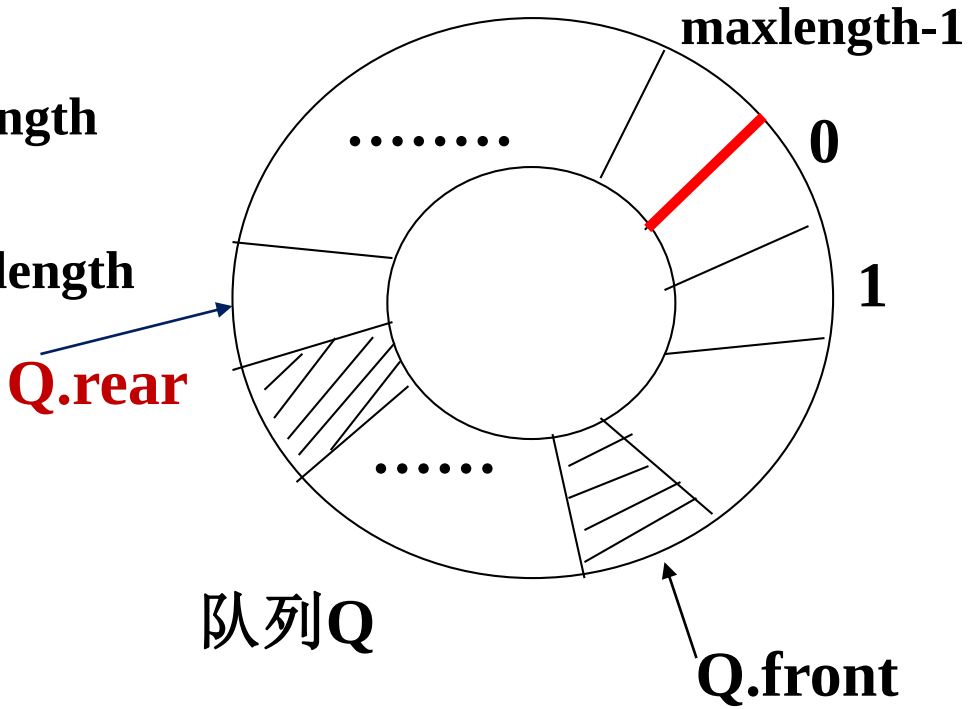
第二种方法是采用循环队列。

插入元素：

Q.rear = (Q.rear + 1） % maxlength

删除元素：

Q.front = (Q.front + 1） % maxlength



队空：

Q.rear == Q.front

?

队满：

Q.rear == Q.front

采用循环队列，rear指针指向队尾元素会怎么样？

插入元素：

Q.rear = (Q.rear + 1） % maxlength

删除元素：

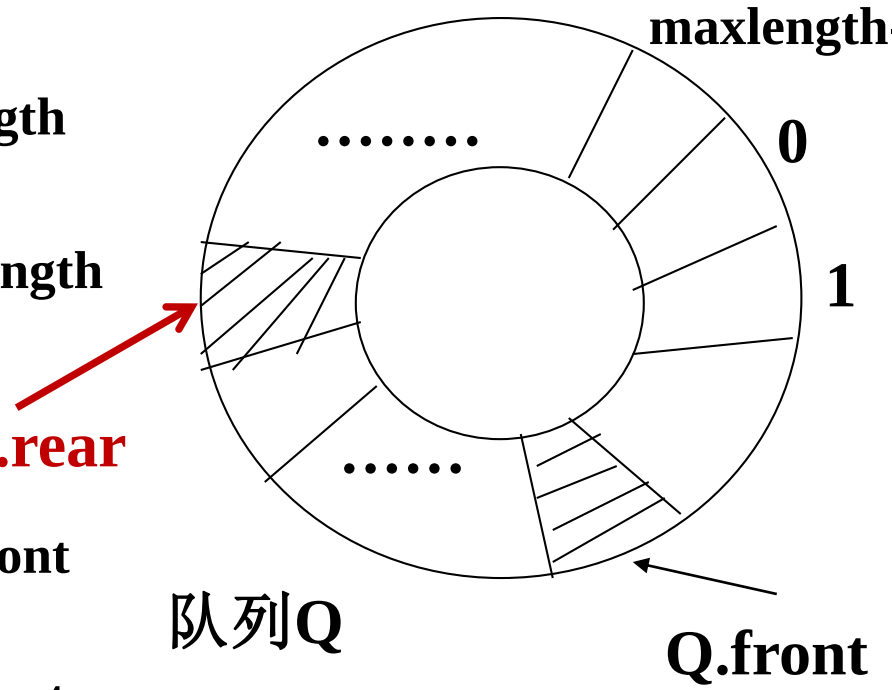
Q.front = (Q.front + 1） % maxlength

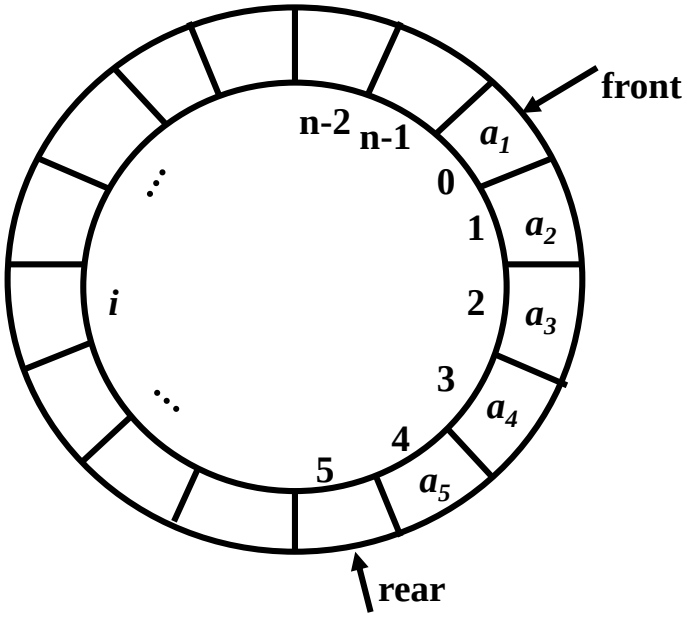
队空：

(Q.rear + 1）% maxlength == Q.front

队满：

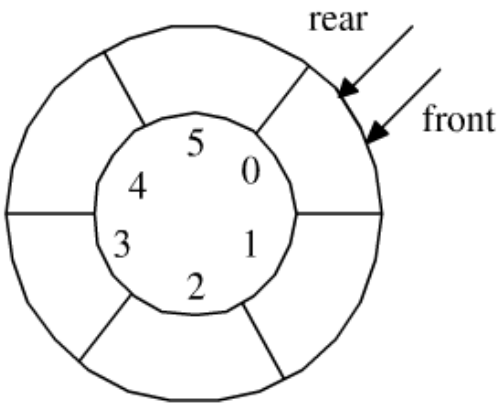
(Q.rear + 1) % maxlength == Q.front



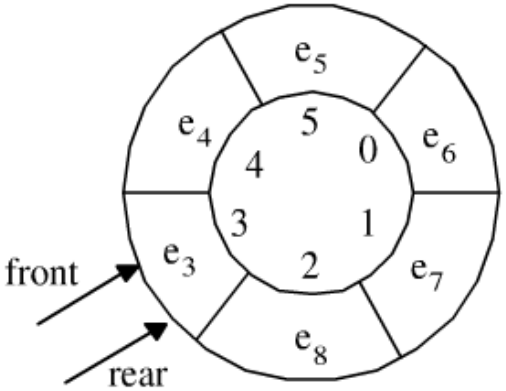


Q.front = (Q.front + 1） % n

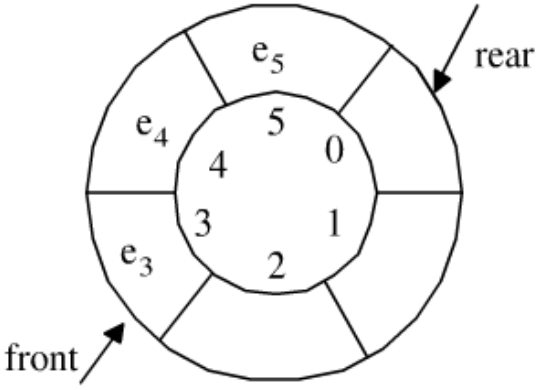
Q.rear = (Q.rear + 1） % n



(a) 空队列



(b) 队列满



(b) 一般情况

问题：如何解决循环队列中队空与队满状态相同？

方法一：另设一个标志位用以区别队空与队满两种状态(count)；

方法二：约定队列头指针在队列尾指针的下一位置上(空位)；

队空：Q.rear == Q.front

队满：(Q.rear+1) mod maxlength == Q.front

结论：两种方法的代价是相同的。

ADT操作：

```
int  Addone( int i )
{  return ( ( i + 1)% maxlength ) ; }
```

①
```
Void  MakeNull ( QUEUE  &Q)
    {  Q.front = 0 ;
       Q.rear = maxlength – 1; }
```

②
```
boolean Empty( Q )
   QUEUE  Q ;
  {  if (Q.rear == Q.front  )
         return  TRUE ;
    else        return  FALSE ;
  }
```

③
```
ElementType  Front( QUEUE  Q )
    { if  ( Empty( Q ) )
         return  Null ;
      else
         return  (Q.data[ Q. front ] );
    } ;
```

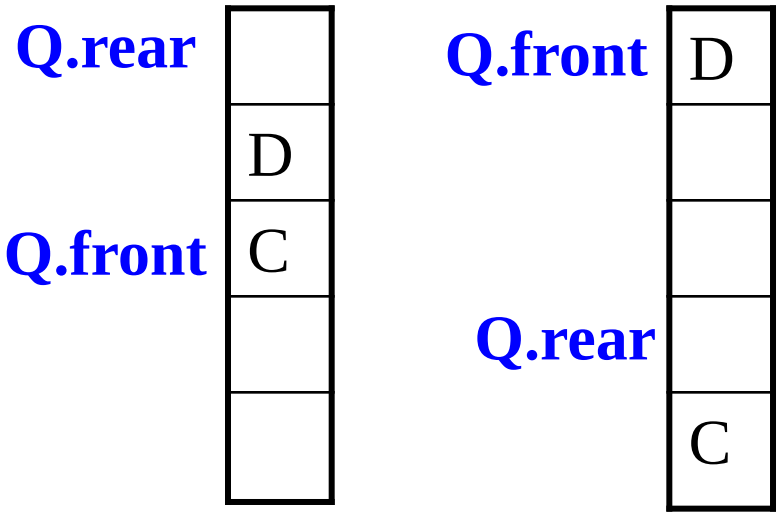
④ Void EnQueue (ElementType x, QUEUE Q)

```
{   if ( Addone( Q.rear ) == Q.front )
        error (“队列满”) ;
    else {
            Q.rear = Addone ( Q.rear ) ;
            Q.datas[ Q.rear ] = x ;                }
}
```

⑤ Void DeQueue (QUEUE Q) ;

```
{    if ( Empty ( Q ) )
        error (“空队列”) ;
     else
        Q.front = Addone ( Q.front ) ;
} ;
```

分析以下两种状态如何求队列长度



Q.rear = 4 **Q.rear=1**

Q.front =2 **Q.front=4**

队列长度 **(Q.rear-Q.front + maxlength)% maxlength;**

循环队列知识点概括

- 队列存放数组被当作首尾相接的线性表处理。
- 队头、队尾指针加1时从maxlength-1直接进到0，可用c语言的取模(余数)运算实现。

☞ 出队： front = (front+1) % maxlength;

☞ 入队： rear = (rear+1) % maxlength;

☞ 队列初始化：front = rear = 0;

☞ 队空： front == rear;

☞ 队满： front == (rear+1) % maxlength;

☞ 求队长： (Q.rear-Q.front+ maxlength)% maxlength;

2.4.3 队列的应用

队列在计算机系统中的应用比较广泛，两个方面简述：

1. 解决主机和外部设备之间速度不匹配的问题

- 内外速度不一致
- 设置数据缓冲区
- 依次从缓冲区取出数据处理

2. 解决由多用户引起的资源竞争问题

- CPU资源的竞争为典型例子
- 多终端计算系统上，多用户发起占用CPU
- 按照请求的先后顺序排成队列
- 依次为队首请求安排动作
- 处理完成安排出队列，再安排用户

- **队列使用的原则：** 凡是符合先进先出原则的
  - 服务窗口和排号机、打印机的缓冲区、分时系统、树型 结构的层次遍历、图的广度优先搜索等等 结构的层次遍历、图的广度优先搜索等等
- **举例**
  - 约瑟夫出圈问题：n个人排成一圈，从第一个开始报数， 报到m的人出圈，剩下的人继续开始从1报数，直到所有 的人都出圈为止。
    - Josephus有过的故事：39 个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人抓。于是决定了自杀方式，41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀。然后下一个重新报数，直到所有人都自杀身亡为止。然而Josephus 和他的朋友并不想遵从，Josephus要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏。
  - 舞伴问题：假设在周末舞会上，男士们和女士们进入舞厅 时，各自排成一队。跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。若两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲。现要求写算法模拟上述舞伴配对问题。

例：某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作。若元素a, b, c, d, e依次入此队列后再进行出队操作，则不可能得到的出队序列是（ ）

（A） b, a, c, d, e　　（B）d, b, a, c, e

（C√） d, b, c, a, e　　（D）e, c, b, a, d

例：现有队列Q与栈S，初始化Q中的元素依次是1, 2, 3, 4, 5, 6,（1在队头），S为空。若仅允许下列3种操作：(1)出队并输出出队元素；(2)出队并将出队元素压入栈；(3)出栈并输出出栈元素，则不能得到的输出序列是（ ）

（A） 1, 2, 5, 6, 4, 3　　（B）2, 3, 4, 5, 6, 1

（C√） 3, 4, 5, 6, 1, 2　　（D）6, 5, 4, 3, 2, 1

例：已知循环队列顺序存储A[0…n-1]，队列非空时front和rear分别指向队头和队尾元素。若初始为空队列，且要求第一个进入队列的元素存在A[0]的位置，那么front和rear初始值（ ）

（A） 0,0 （B√）0，n-1 （C） n-1,0（D）n-1,n-1

例：数组Q[n]用来表示一个循环队列，f为当前队列头元素的前一位置，r为队尾元素的位置。假定队列中元素的个数小于n，计算队列中元素的公式为：

（A）r－f　　（B）（n＋f－r）％n

（C）n＋r－f　　（D）（n＋r－f）％n √

分析：4种公式哪种合理？

当 r ≥f 时（A）合理；

当 r < f 时（C）合理；

} 综合2种情况，以（D）的表达最为合理

例：在一个循环队列中，若约定队首指针指向队首元素的前一个位置。那么，从循环队列中删除一个元素时，其操作是

先 移动队首指针 ，后 取出元素 。

注意变换思维方式

例：多项式的代数运算

$$P(x) = \sum_{i=n-1}^{0} a_i \, x^{\,i}$$

方案1：数组一

n-1	a_{n-1}
…	…
i	a_i
…	…
3	a_3
2	a_2
1	a_1
0	a_0

coeftype p[N};

14	3
13	0
12	0
11	0
10	0
9	0
8	2
7	0
6	0
5	0
4	0
3	0
2	0
1	0
0	1

$P(x)=3x^{14}+2x^{8}+1$

方案2：数组二

$coef_{n-1}$	exp_{n-1}
$coef_{n-2}$	exp_{n-2}
…	…
$coef_i$	exp_i
…	…
$coef_2$	exp_2
$coef_1$	exp_1
$coef_0$	exp_0
…	…

```
Struct {
        coeftype  coef;
        exptype exp;
} p[N]
```

$P(x)=3x^{14}+2x^{8}+1$

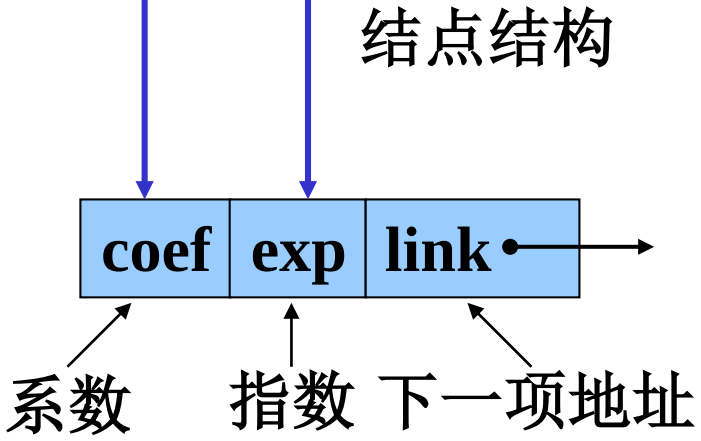
3	14
2	8
1	0
	…

方案3：链表

$$P(x) = \sum_{i=n}^{0} a_i x^i$$

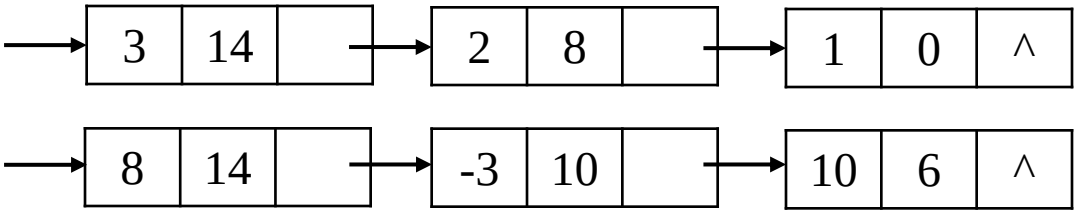
结点类型：

```
struct PolyNode {
    int   coef ;
    int   exp ;
    polylink  *link ;
}
typedef PolyNode
*PolyPointer ;
```



$a(x) = 3x^{14} + 2x^8 + 1$

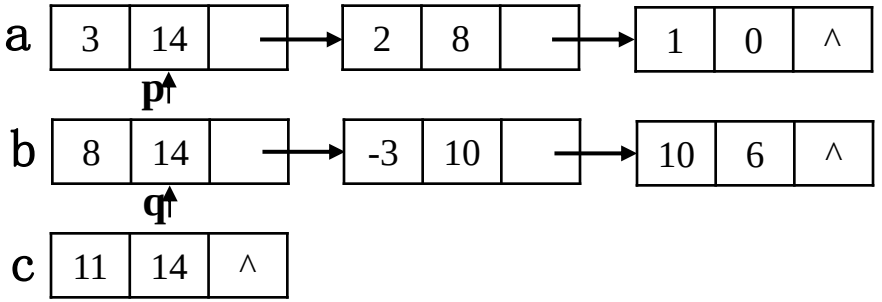
$b(x) = 8x^{14} - 3x^{10} + 10x^6$



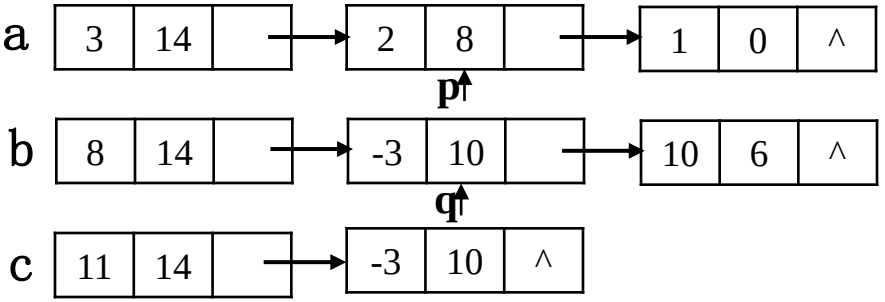
$c(x) = a(x) + b(x)$

$= 11x^{14} - 3x^{10} + 2x^8 + 10x^6 + 1$

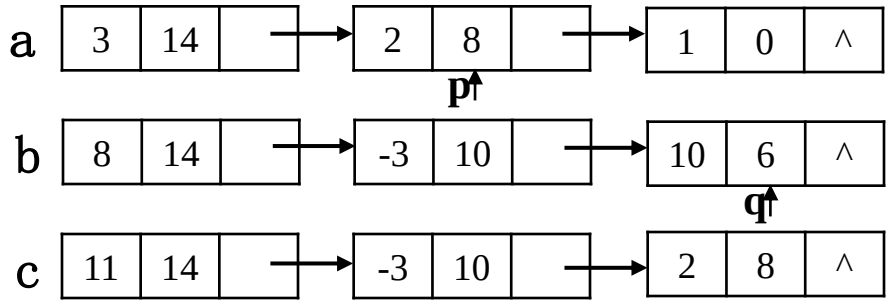
(1) p->exp == q->exp



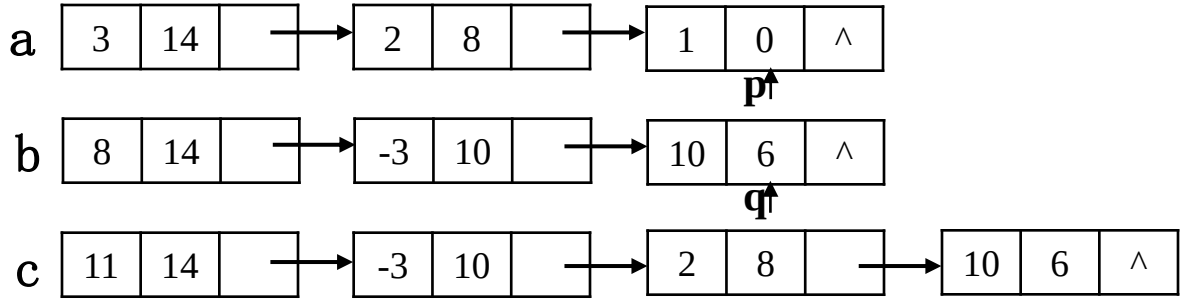
(2) p->exp < q->exp



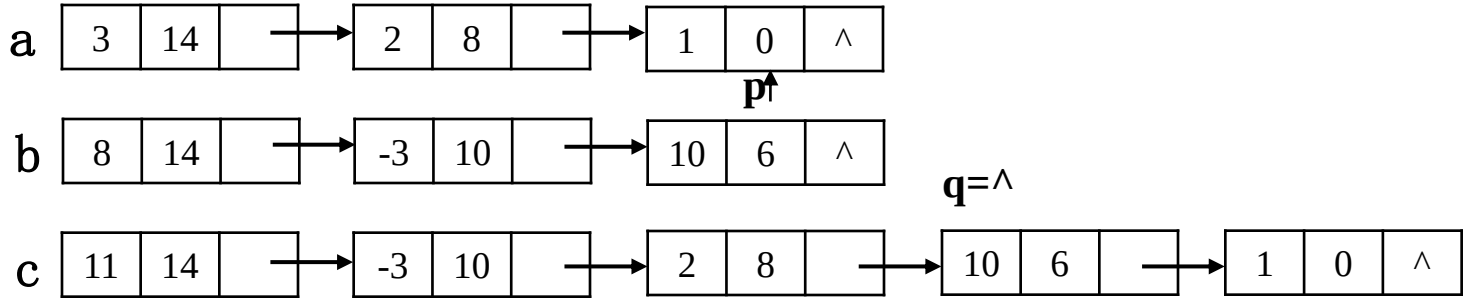
(3) p->exp > q->exp



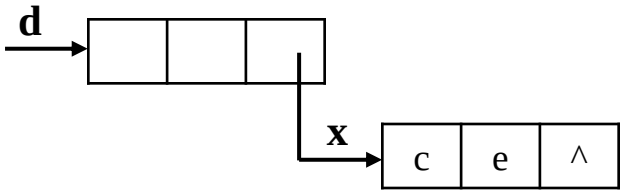
(4) p->exp < q->exp



(5) p != null

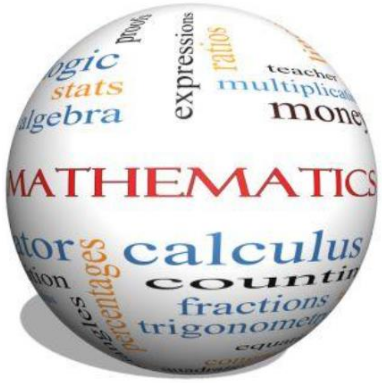


Attch(int c, int e, PolyPointer d)




```
PolyPointer  Attch ( int  c , int e ,
PolyPointer  d )
{     PolyPointer   x  ;
       x = new   PolyNode ;
       x->coef = c ;
       x->exp = e ;
       d->link = x ;
       return  x ;
}
```

Compare(int x , int y) =
- '=' 当x=y
- '>' 当x>y
- '<' 当x<y




```
char  Compare ( int x, int y)
{
   char c;
   if( x == y )
        c = '=';
    else if( x > y )
        c = '>';
    else
        c = '<';
    return( c );
}
```

表达式加法算法：

```
PolyPointer PolyAdd ( PolyPointer a ,
                      PolyPointer b )
{ PolyPointer p, q, d, c;
  int y ;
  p = a->link;  q = b->link;
  c = new PolyNode; d = c ;
  while ( (p != NULL) && (q != NULL) )
     switch ( Compare ( p->exp, q->exp ) )
     {  case '=' :
           y = p->coef + q->coef ;
           if ( y )  d = Attch( y, p->exp, d );
           p = p->link ; q = q->link ;
           break;
        case '>':
           d = Attch( p->coef, p->exp, d );
           p = p->link ;
          break;
        case '<':
   d = Attch( q->coef, q->exp, d );
   q = q->link ;
    break;
 }
 while ( p != NULL )
 { d = Attch( p->coef, p->exp, d );
   p = p->link ;
 }
 while ( q !=NULL )
 { d = Attch( q->coef, q->exp, d );
   q = q->link ;
 }
 d->link = NULL ;
 p = c;  c = c->link;
 delete  p;
 return  c;
}
```

2.5 串(String)

2.5.1 抽象数据型串

串是线性表的一种特殊形式，表中每个元素的类型为字符型，是一个有限的字符序列。

串的基本形式可表示成：$S = ‘ a_1a_2a_3\cdots\cdots a_n ’$;

其中： char a_i ； $0 \le i \le n$ ； $n \ge 0$ ； 当 $n = 0$ 时，为空串。

n 为串的长度 ;

C 语言中串有两种实现方法：

(1) 字符数组，如：char str1[10] ;

(2) 字符指针，如：char *str2 ;

ADT操作：

string Null( ) ;

boolean IsNull (S) ;

Void In(S, a) ;

int Len(S) ;

Void ConcatT(S1, S2) ;

string Substr(S, m, n) ;

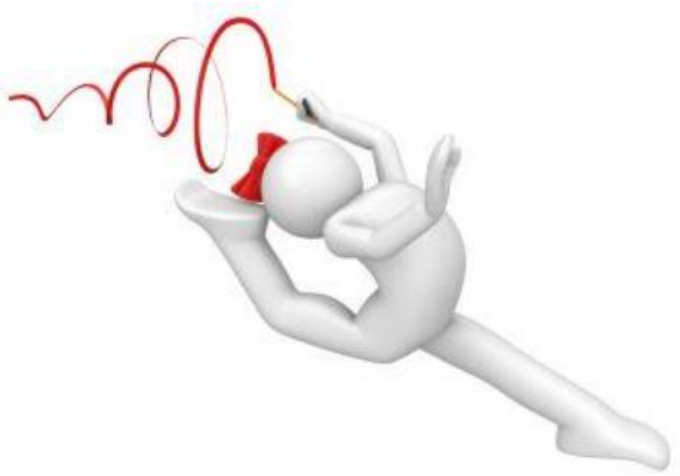
boolean Index(S, S1) ;

【例2-15】将串T 插在串S 中第 i 个字符之后Insert(S, T, i)。

```
Void  Insert( STRING  &S, STRING  T, int  i )
{   STRING  t1,  t2 ;
    if  ( ( i < 0 ) || ( i > LEN( S ) )
        error ‘指定位置不对’  ;
    else
        if ( IsNull( S ) )  S = T ;
        else
            if ( IsNull ( T ) )
               {   t1 = Substr( S, 1, i ) ;
                   t2 = Substr( S, i + 1, LEN( S ) );
                   S = Concat( t1, Concat( T, t2 ) );
                }
}
```

【例2-16】从串 S 中将子串 T 删除Delete(S, T)。

```
Void  Delete( STRING  &S, STRING  T )
{   STRING  t1,  t2 ;
    int  m, n ;
    m = Index( S, T );
    if  ( m==0 )
            error ‘串S中不包含子串T’ ;
    else
        {   n = Len( T ) ;
            t1 = Substr( S, 1, m - 1 ) ;
            t2 = Substr( S, m + n, LEN( S ) );
            S = Concat( t1, t2);
         }
}
```

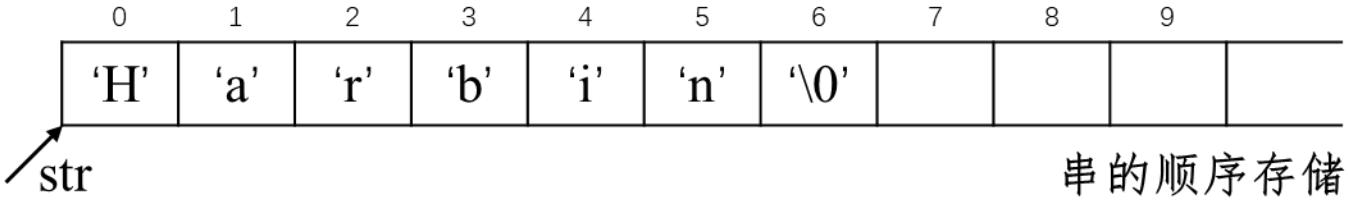


2.5.2 串的实现

1、串的顺序存储

采用连续的存储空间（数组），自第一个元素开始，依次存储字符串中的每一个字符。

char str[10] =“Harbin”;



操作：**Null，IsNull，In，Len，Concat，Substr，Index**

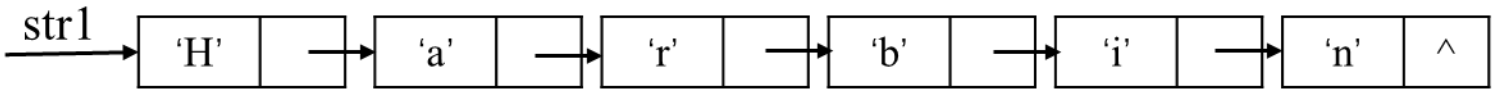
2、串的链式存储

构造线性链表，**element**类型为**char**，自第一个元素开始，依次存储字符串中的每一个字符。

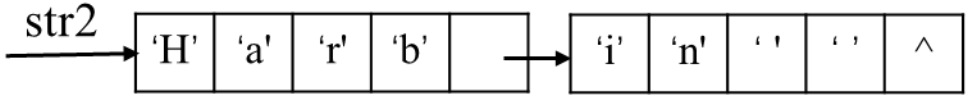
2、串的链式存储（续）

```
struct node {
    char data;
    node *link;
};
typedef node *STRING1;
STRING1 str1;
```

```
struct node {
    char data[4];
    node *link;
};
typedef node *STRING2;
STRING2 str2;
```



分配空间利用率为1/5（20%）



串的块链存储

分配空间利用率约为1/2（50%）

两种方式的对比分析
（空间利用率，操作速度）

假设地址量（link）占用4个字节

ADT操作：

Index(S,S1)

若S1是S的子串则返回S1首字符在S中的位置，否则返回0；

```
int  Index ( STRING1  S, S1 )
{
     struct node *p, *q, *i ;
     int  t ;
     if  ( ( S1 != Null ) && ( S != Null) )
        {   t = 1 ;  i = S ;  q = S1 ;
            do {
                  if ( p→data == q→data )
                     {   q = q→link ;
                         if ( q == Null )  return( t ) ;
                         p = p→link ;
                      }
                  else
                     {   t = t + 1 ;  i = i→link ;
                         p = i ;  q = S1 ;
                      }
               } while ( p != Null ) ;
        }
      return  0 ;
} ;
```

3、模式匹配(朴素模式匹配算法)

模式匹配(字符串匹配是计算机的基本任务之一)

给定S=“S_0 S_1… S_{n-1}”(主串)和T=“T_0 T_1…T_{m-1}”(模式)，在S 中寻找T 的过程称为模式匹配。如果匹配成功，返回T 在S 中的位置；如果匹配失败，返回-1。

假设串采用顺序存储结构。

朴素模式匹配算法(Brute-Force算法) ：枚举法(回溯)

从主串S 的第一个字符开始和模式T 的第一个字符进行比较，若相等，则继续比较两者的后续字符；否则，从主串S 的第二个字符开始和模式T 的第一个字符进行比较。

重复上述过程，直到T 中的字符全部比较完毕，说明本趟匹配成功；或S 中字符全部比较完，则说明匹配失败。

设主串S=“ababcabcacbab”，模式串T=“abcac”

第1趟匹配	主串	ababcabcacbab	i=2
	模式串	abc	j=2 匹配失败
第2趟匹配	主串	ababcabcacbab	i=1
	模式串	abc	j=0 匹配失败
第3趟匹配	主串	ababcabcacbab	i=6
	模式串	abcac	j=4 匹配失败
第4趟匹配	主串	ababcabcacbab	i=3
	模式串	abc	j=0 匹配失败
第5趟匹配	主串	ababcabcacbab	i=4
	模式串	abc	j=0 匹配失败
第6趟匹配	主串	ababcabcacbab	i=9 //返回i-lenT+1
	模式串	abcac	j=4 匹配成功

特点:主串指针需回溯（i-j+1），模式串指针需复位（j=0）。

BF算法实现的详细步骤：

1.在串S和串T中设比较的起始下标i和j；

2.循环直到S或T的所有字符均比较完；

2.1 如果S[i]=T[j]，继续比较S和T的下一个字符；

2.2 否则，将i回溯(i=i-j+1)，j复位，准备下一趟比较；

3.如果T中所有字符均比较完，则匹配成功，返回主串起始比较下标；

否则，匹配失败，返回-1。

```
int StrMatch_BF ( char* S, char* T, int pos=0)
{   /*为主串S、模式T长度分别为lenS和lenT，且字符串采用顺序存储*/
    i = pos;    j = 0;                          // 从第一个位置开始比较
   while (i<lenS && j<lenT) {
       if (S[i] == T[j]) {++i; ++j;}    // 继续比较后继字符
       else {i = i – j + 1;    j = 0;  }   // 指针后退重新开始匹配
    }                                          // 返回与模式第一字符相等的字符在主串
中的序号
    if ( j >= lenT)
         return i- lenT+1;                                  i,j=1开始又怎么样？
   else
         return -1;                          // 匹配不成功
}
```

Brute-Force算法的时间复杂度

主串S长n,模式串T长m。可能匹配成功的(主串)位置(0 ~ n-m)。

①最好的情况下，模式串的第0个字符失配

设匹配成功在S的第i个字符，则在前i趟匹配中共比较了i次，第i趟成功匹配共比较了m次，总共比较了（i+m）次。所有匹配成功的可能共有n-m+1种，所以在等概率情况下的平均比较次数：

$$\sum_{i=0}^{n-m} p_i(i+m) = \frac{1}{n-m+1}\sum_{i=0}^{n-m}(i+m) = \frac{1}{2}(n+m)$$

最好情况下算法的平均时间复杂度O(n+m)。

②最坏的情况下，模式串的最后1个字符失配

设匹配成功在S的第 i 个字符，则在前i趟匹配中共比较了i*m次，第i趟成功匹配共比较了m次，总共比较了(i+1)*m次。所有匹配成功的可能共有n-m+1种，所以在等概率情况下的平均比较次数：

$$\sum_{i=0}^{n-m} p_i(i+1)m = \frac{m}{n-m+1}\sum_{i=0}^{n-m}(i+1) = \frac{1}{2}m(n-m+2)$$

最坏情况下的平均时间复杂度为O(n*m)。