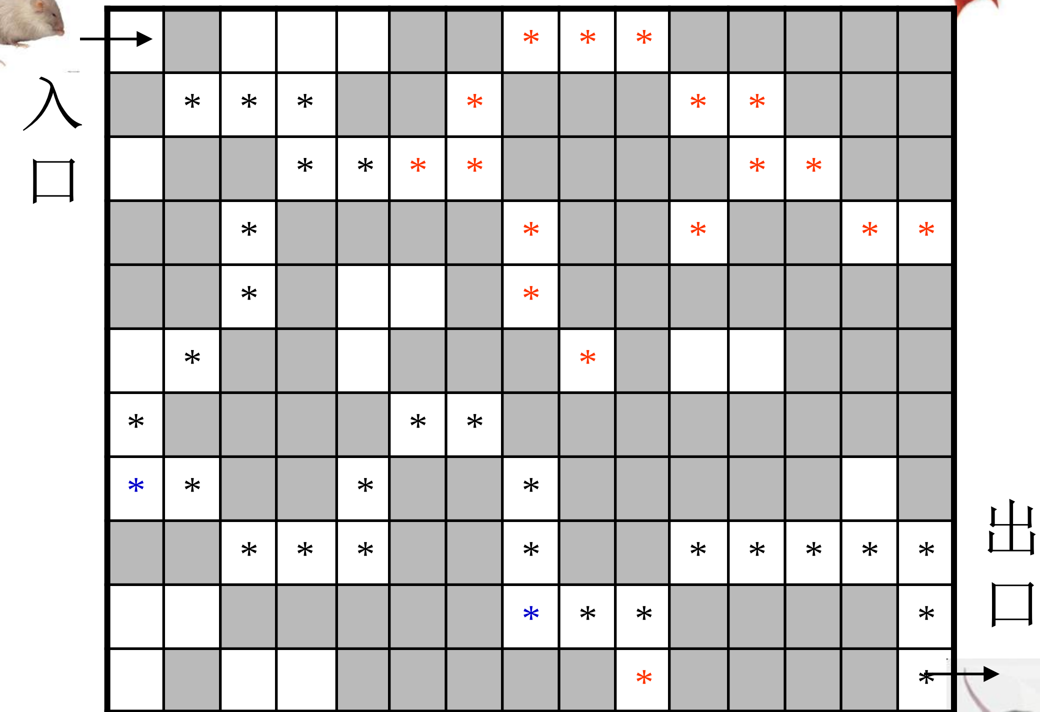
第2章 线性表(Liner List)

问题:

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	1	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	0	1	1	1	1	1	0

$11\times 15 \rightarrow m \times n$

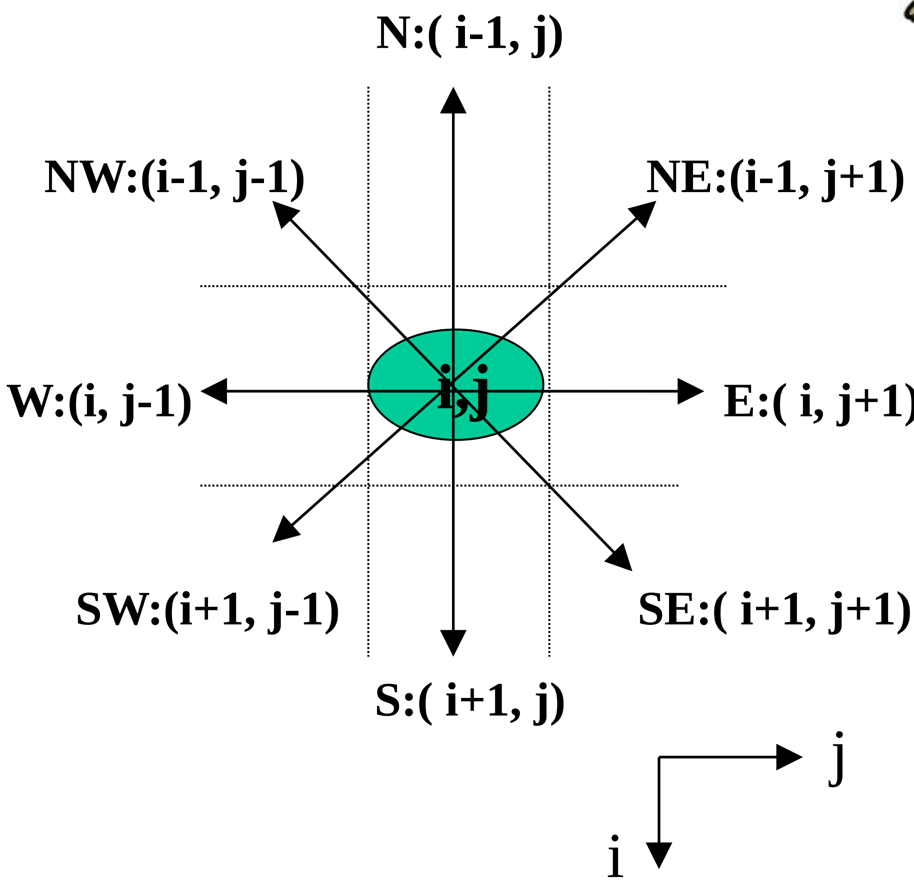


迷宫示例

一个迷宫可用上图所示方阵[m,n]表示，0表示能通过，1 表示不能通过。现假设耗子从左上角[1,1]进入迷宫，编写算法，寻求一条从右下角 [m,n] 出去的路径。

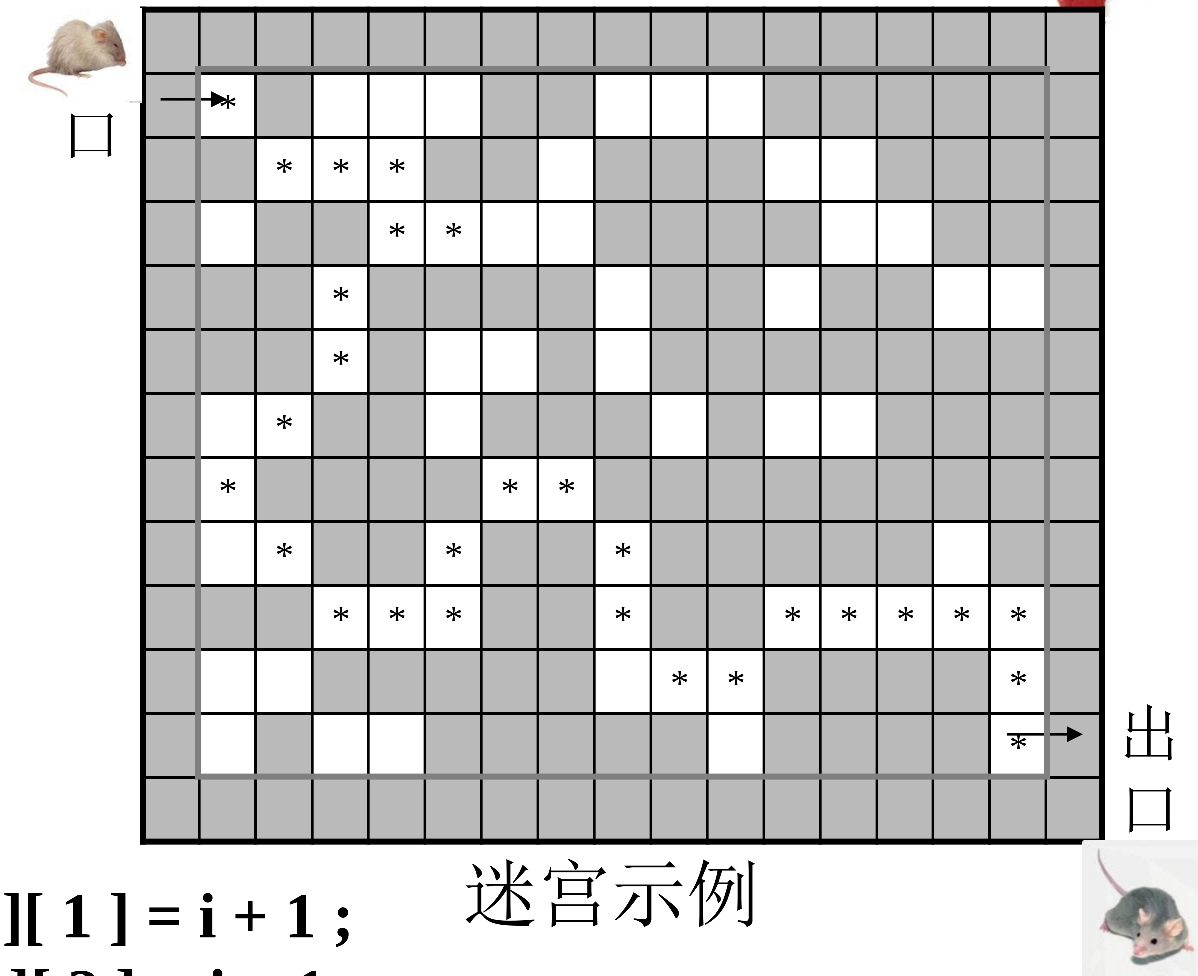
分析:

(1) 迷宫可用二维数组 **Maze[i][j] (1≤i≤m, 1≤j≤n)** 表示，入口**maze[1][1] = 0 ;** 耗子在任意位置可用（**i, j**）坐标表示;



(2) 位置 **(i, j)** 周围有**8**个方向可以走通，分别记为：**E,SE,S,SW,W,NW,N,NE;**如图所示。方向 **v** 按从正东开始且顺时针分别记为**1-8**，**v=1,8;** 设二维数组 **move** 记下八个方位的增量;

V	i	j	说明
1	0	1	E
2	1	1	SE
3	1	0	S
4	1	-1	SW
5	0	-1	W
6	-1	-1	NW
7	-1	0	N
8	-1	1	NE



迷宫示例

从(i , j) 到 (g , h)
且 v = 2 (东南）则
有：　g = i + move[v][1] = i + 1 ;
　　　h = j + move[v][2] = j + 1 ;

(3) 为避免时时监测边界状态，可把二维数组maze[1:m, 1:n]扩大为maze[0:m+1, 0:n+1]，且令0行、0列、m+1行、n+1列的值均为1；

(4) 采用试探的方法，当到达某个位置且周围八个方向走不通时需要回退到上一个位置，并换一个方向继续试探；为解决回退问题，需设一个栈，当到达一个新位置时将（i, j,v）进栈，回退时退栈。

(5) 每次换方向寻找新位置时，需测试该位置以前是否已经过，对已到达的位置，不能重复试探，为此设矩阵mark[][],其初值为0，一旦到达位置(i , j)时，置 mark[i][j] = 1;

文字描述算法：

(1) 耗子在(1, 1)进入迷宫，并向正东（v=1）方向试探。

(2) 检测下一方位(g, h)。若(g, h)=(m, n)且maze[m][n]=0，则耗子到达出口，输出走过的路径；程序结束。

(3) 若(g, h) ≠(m, n),但(g, h)方位能走通且第一次经过,则记下这一步，并从(g, h)出发，再向东试探下一步。否则仍在(i, j)方位换一个方向试探。

(4) 若(i, j)方位周围8个方位阻塞或已经过，则需退一步，并换一个方位试探。若(i, j)=(1, 1)则到达入口，说明迷宫走不通。

```
Void  GETMAZE ( maze , mark ,move ,S )
{  (i, j, v) = (1,1,1) ;    mark[1][1] = 1 ;   s.top = 0 ;
   do {  g = i+move[v][1] ;  h = j+move[v][2] ;
         if (( g == m) && ( h == n) && (maze[m][n] == 0 ))
             { output( S ) ; return ; }
         if ((maze[g][h] ==0) && mark[g][h] == 0))
             { mark[g][h] = 1 ;  PUSH( i, j, v, S) ; (i, j, v) = (g, h, 1)  ;  }
          else if ( v < 8 )
                 v = v + 1 ;
          else { while (( s.v == 8) && (!EMPTY(S)))  POP( S ) ;
                    if ( s.top > 0 )
                       ( i, j, v++ ) = TOP( S ) ;    } ;
      } while (( s.top ) && ( v != 8 )) ;
   cout << "路径不存在！"   ;  }
```

2.3.4 栈的应用(Cont.)

•波兰逻辑学家J.Lukasiewicz于1929年提出的一种表达式。

- 表达式求值
- 表达式的三种形式

$$\text{表达式:}\begin{cases}\text{前缀表达式(波兰式)}\\ \text{中缀表达式}\\ \text{后缀表达式(逆波兰式)}\end{cases}$$

$$\text{例如，}(a+b)*(a-b)=\begin{cases}* \ + \ a \ b \ - \ a \ b\\ (a+b)*(a-b)\\ a \ b \ + \ a \ b \ - \ *\end{cases}$$

- 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：
  - 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。
  - 后缀表达式中不需要括号规定计算顺序，而由运算操作符）的位置来确定运算顺序。

2.3.4 栈的应用(Cont.)

中缀表达式：$(a+b)*(a-b) \Rightarrow$ 后缀表达式：$a\ b\ +\ a\ b\ -\ *$

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，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，将栈顶操作符依次退栈并输出，直到满足要求为止；
- 遇到“（”进栈，当遇到“）”时，退栈输出直到“（”为止。

2.3.4 栈的应用(Cont.)

$$(a+b)*(a-b) \Rightarrow a\ b\ +\ a\ b\ -\ *$$

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，将操作数进栈保存；
- 当遇到操作符时，从栈中退出两个操作数并作相应运算，将计算结果进栈保存；直到表达式结束，栈中唯一元素即为表达式的值。

•实验一 线性表及应用--算术表达式求值

•要求：

键盘可以重复输入中缀算术表达式

•编程实现转换成后缀表达式输出

•再对该后缀表达式求值计算输出结果。

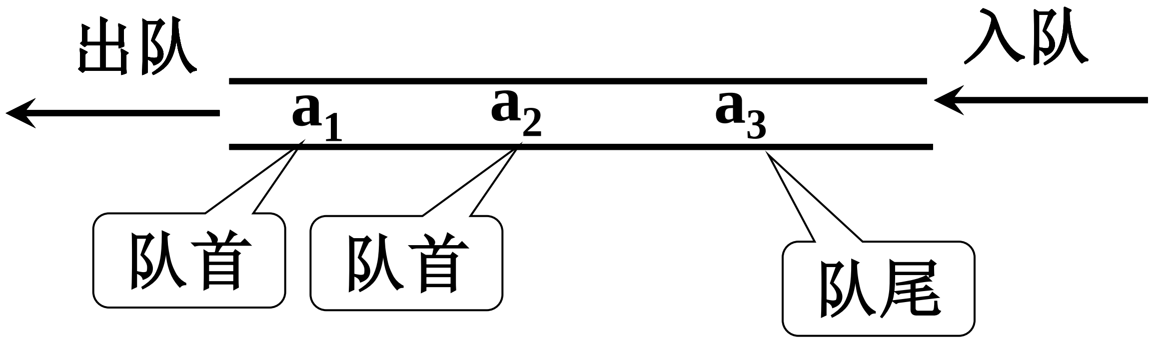
•+ - * / % log

•整数、小数、负数

- 栈的应用
  - 地图四染色
  - 迷宫问题
  - 函数的嵌套调用
  - 递归的实现
    - 斐波那契数列
    - 汉诺塔问题
    - 递归函数
  - 回文游戏
  - 多进制转换
  - 括号匹配的检验
  - 行编辑程序
  - 表达式求值

2.4 特殊的线性表--队列

- 队列：只允许在一端进行插入操作，而另一端进行删除操作的线性表。
- 空队列：不含任何数据元素的队列。
- 队尾和队首：允许插入（也称入队、进队）的一端称为队尾，允许删除（也称出队）的一端称为队首。



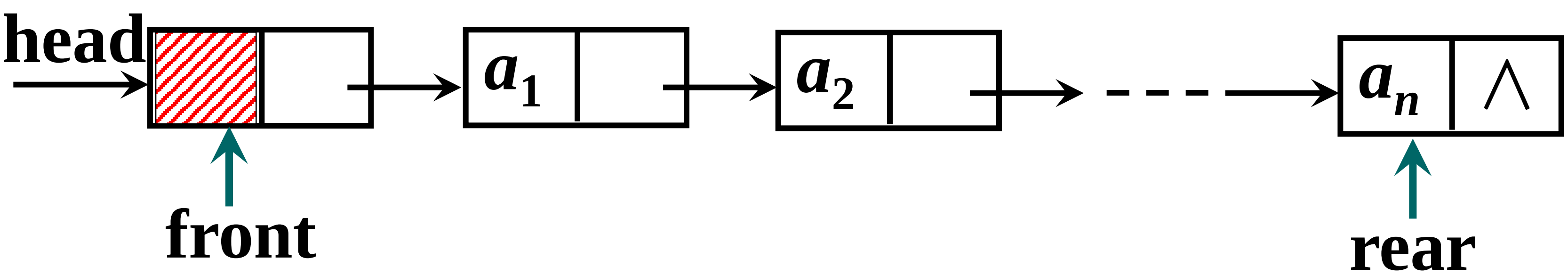
队列的操作特性：先进先出

- 队列的操作：
  - MakeNull(Q)、Front(Q)、EnQueue(x, Q)、DeQueue(Q)、Empty(Q)

2.4.1 队列的指针实现

队列的链接存储结构及实现

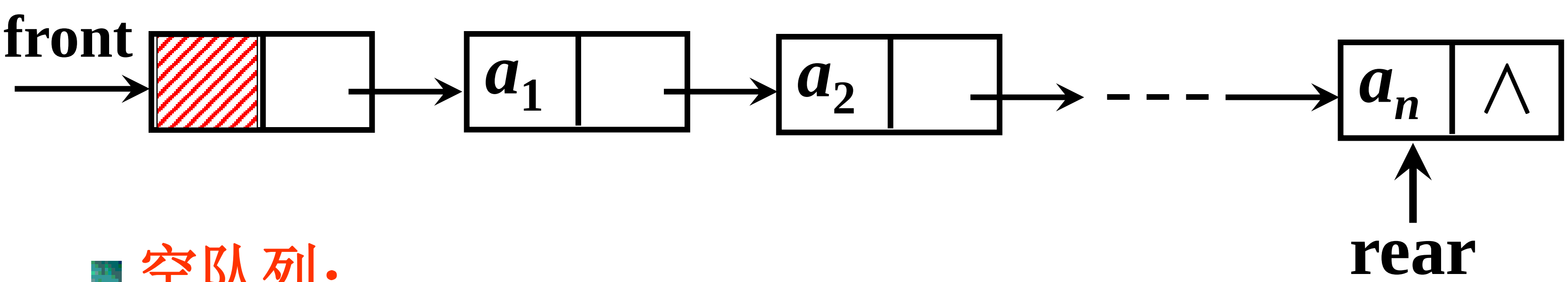
- 链队列：队列的链接存储结构
- 如何改造单链表实现队列的链接存储？



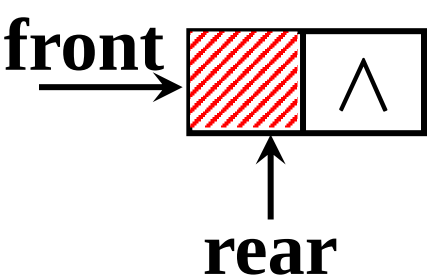
- 队首指针即为链表的头结点指针
- 增加一个指向队尾结点的指针

2.4.1 队列的指针实现(Cont.)

- 队列的链接存储结构及实现
  - 非空队列：



  - 空队列：

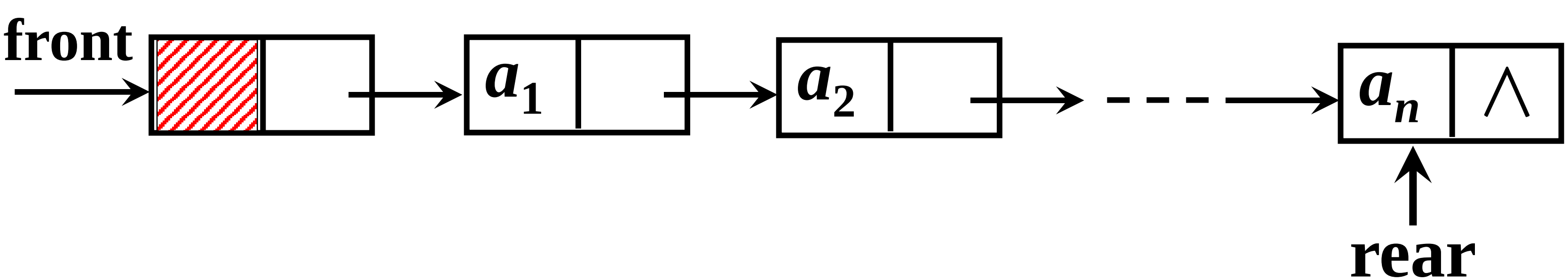


2.4.1 队列的指针实现(Cont.)

- 队列的链接存储结构及实现
  - 存储结构定义

```
//结点的类型：
struct  celltype  {
     ElemType  data;
      celltype  *next ;
} ;
```

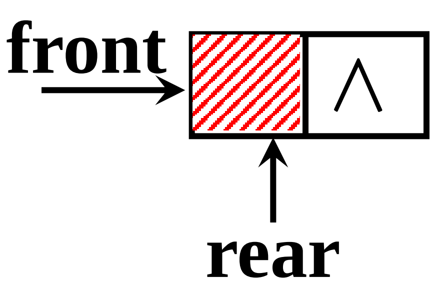
```
队列的 类型：
struct  QUEUE {
      celltype  *front ;
      celltype  *rear ;
};
```



2.4.1 队列的指针实现(Cont.)

- 队列的链接存储结构及实现
  - 操作的实现----初始化和判空

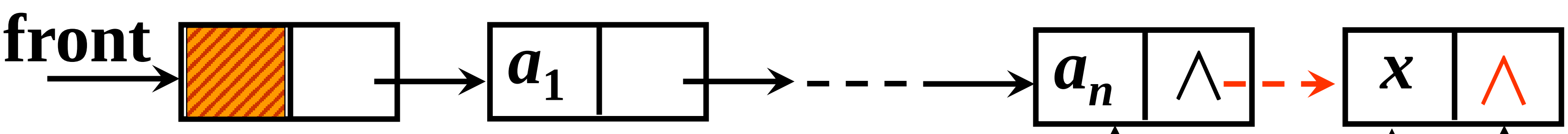
```
① void  MakeNull( QUEUE  &Q )
   {
      Q.front = new celltype ;
      Q.front→next = NULL ;
      Q.rear = Q.front ;
    }
```



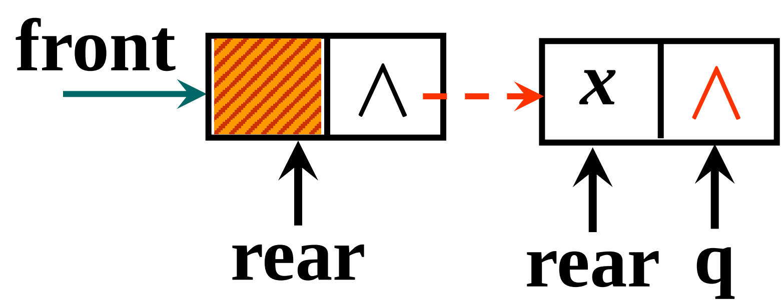

```
② Boolean Empty(QUEUE Q)
   {  if  ( Q.front == Q.rear )
              return  TRUE ;
        else
              return  FALSE ;
   }
```

2.4.1 队列的指针实现(Cont.)

- 队列的链接存储结构及实现
  - 操作的实现----入队



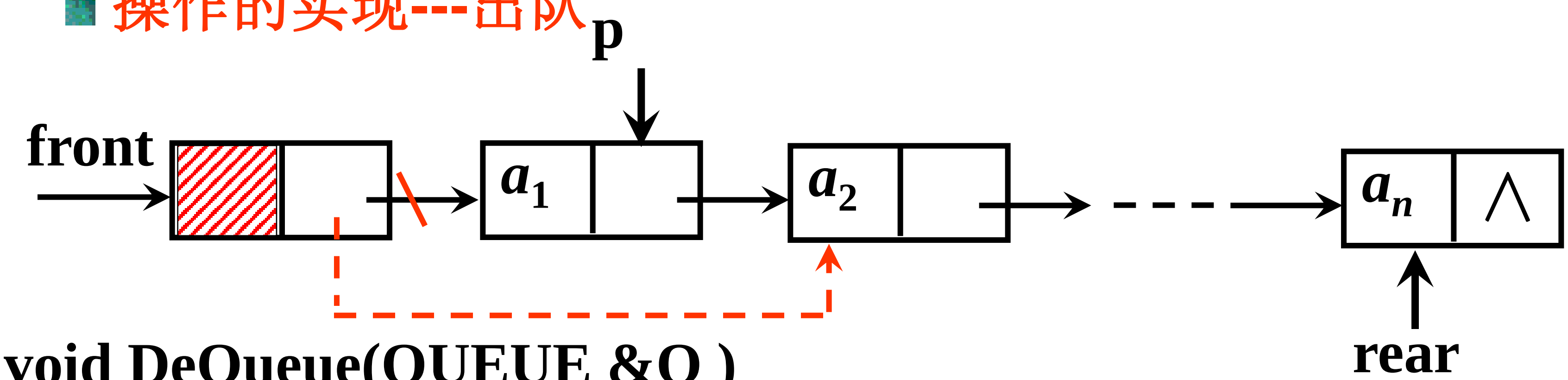

```
③ void  EnQueue(ElemType x, QUEUE  &Q)
  {     q=new cwlltype;
        q->data=x ;
        q->next=NULL;
        //q->next=Q.rear->next;
        Q.rear->next=q;
        Q.rear=q;
  }
```



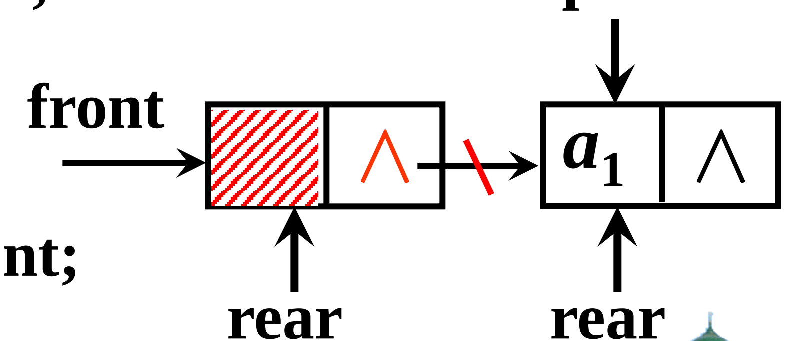
如果没有头结点会怎样?

2.4.1 队列的指针实现(Cont.)

- 队列的链接存储结构及实现
  - 操作的实现---出队



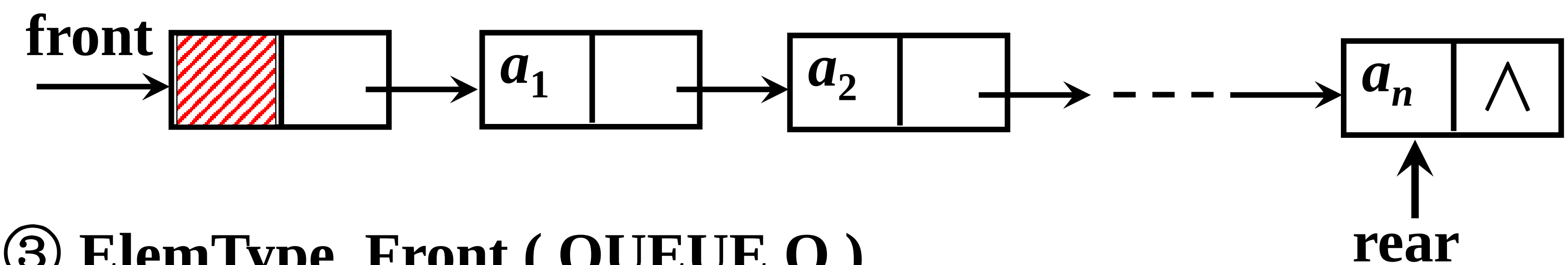

```
void DeQueue(QUEUE &Q )
{    if (Q.rear==Q.front) cout<<"队空";
         p=Q.front->next;
         Q.front->next=p->next;
     if (p->next==NULL) Q.rear=Q.front;
     delete p;
}
```



考虑边界情况：队列中只有一个元素?

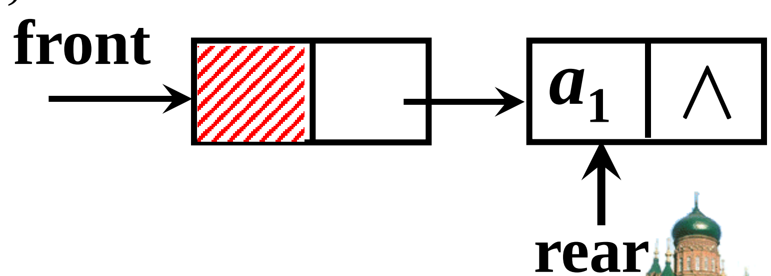
2.4.2 队列的指针实现(Cont.)

- 队列的链接存储结构及实现
  - 操作的实现---返回队首元素



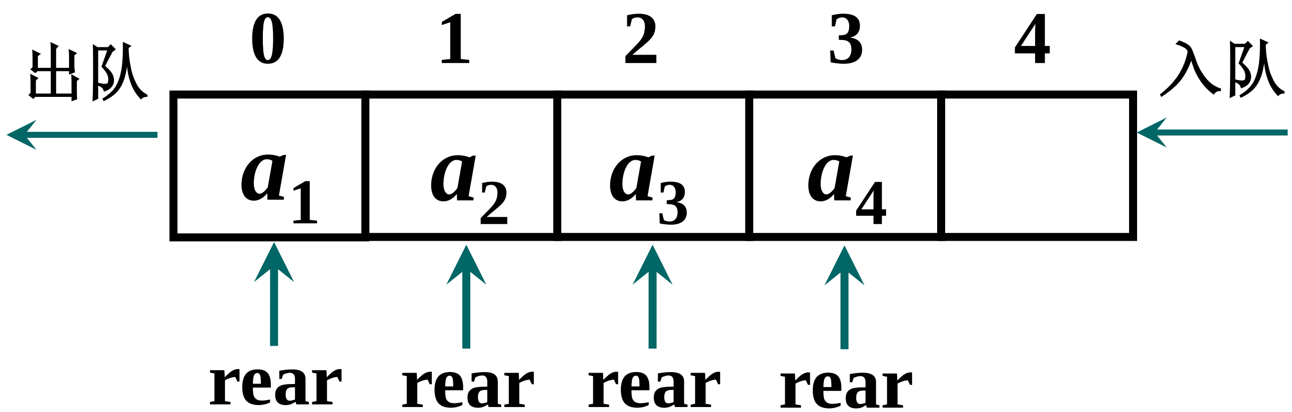
③ ElemType Front (QUEUE Q)

```
{    if ( Q.front→next )
         return  Q.front→next→ data ;
}
```



2.4.3 队列的数组实现

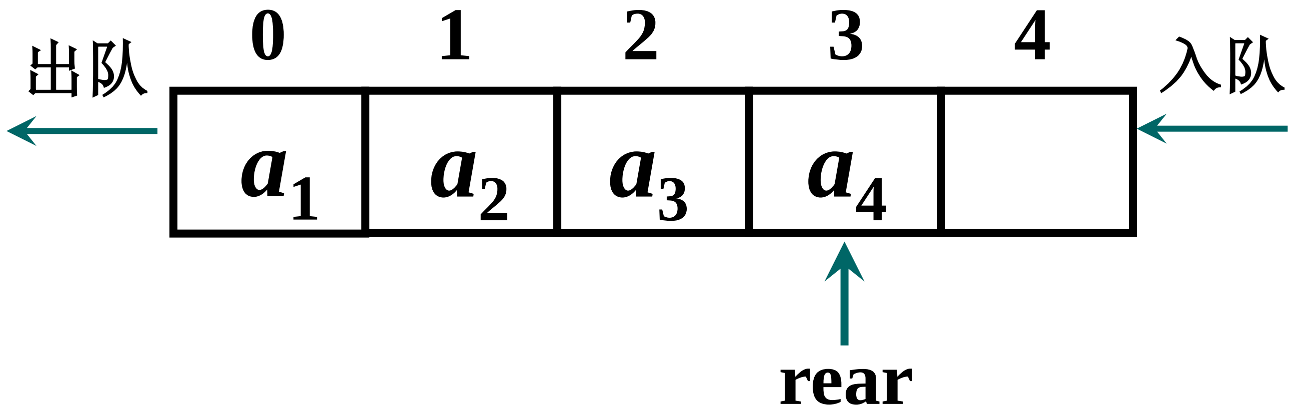
- 队列的顺序存储结构及实现
  - 如何改造数组实现队列的顺序存储?
    - $a_1a_2a_3a_4$依次入队



入队操作时间性能为O(*1*)

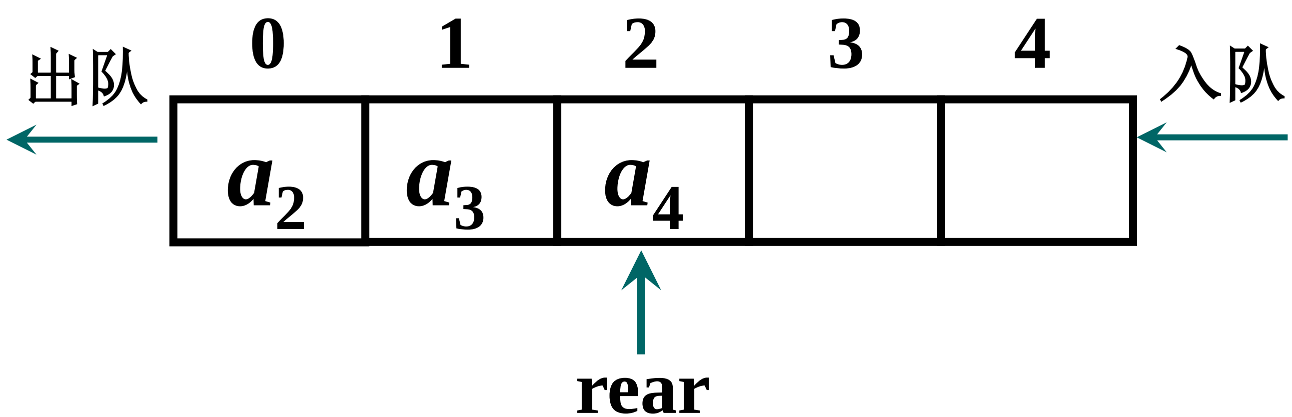
2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 如何改造数组实现队列的顺序存储?
    - a_1a_2依次出队



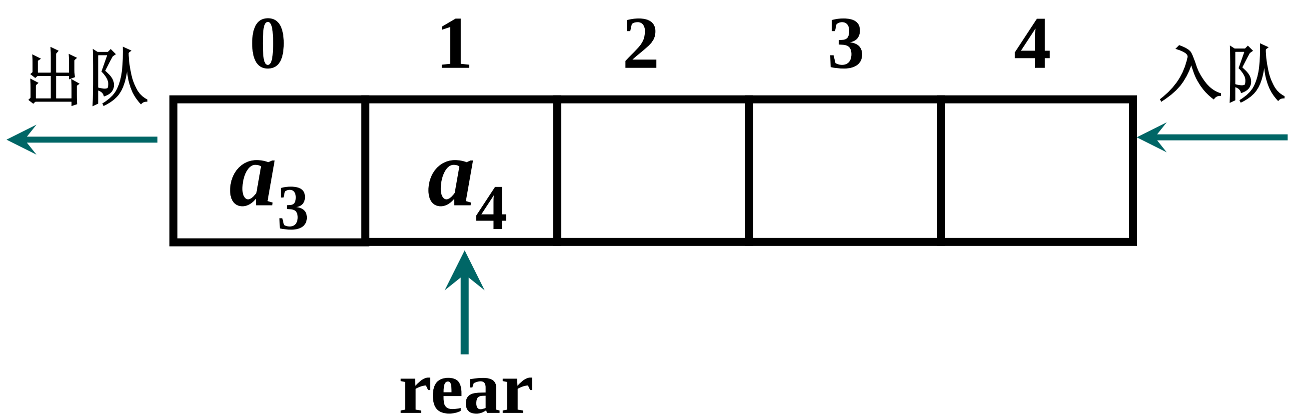
2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 如何改造数组实现队列的顺序存储？
    - a_1a_2依次出队



2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 如何改造数组实现队列的顺序存储?
    - a_1a_2依次出队

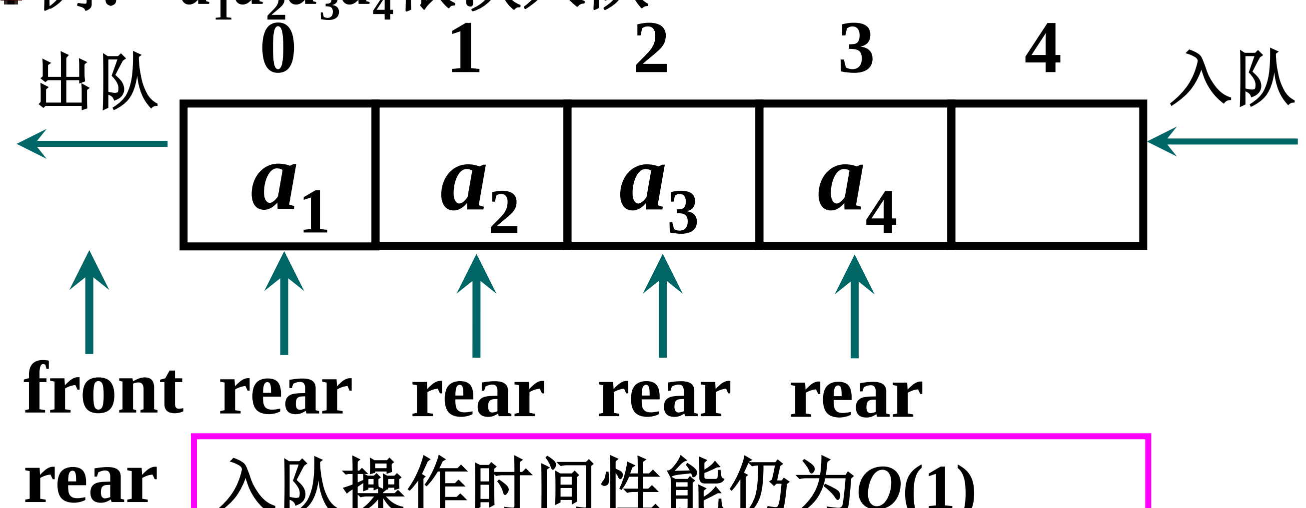


$\boxed{\text{出队操作时间性能为O}(n)}$

2.4.3 队列的数组实现(Cont.)

队列的顺序存储结构及实现

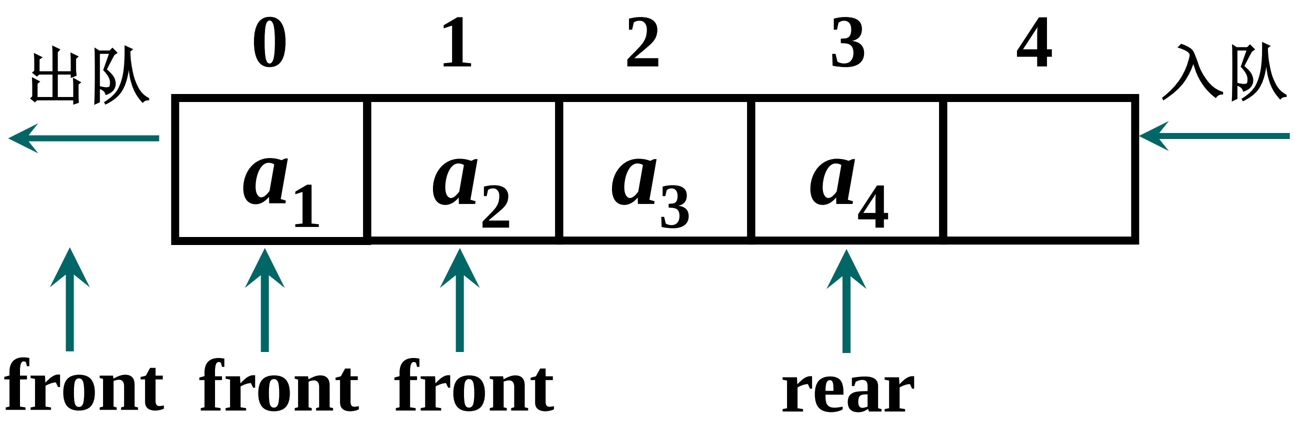
- 如何改进出队的时间性能?
  - 所有元素不必存储在数组的前*n*个单元;
  - 只要求队列的元素存储在数组中连续单元;
  - 设置队头、队尾两个指针.
  - 例: $a_1a_2a_3a_4$依次入队



入队操作时间性能仍为$O(1)$

2.4.3 队列的数组实现(Cont.)

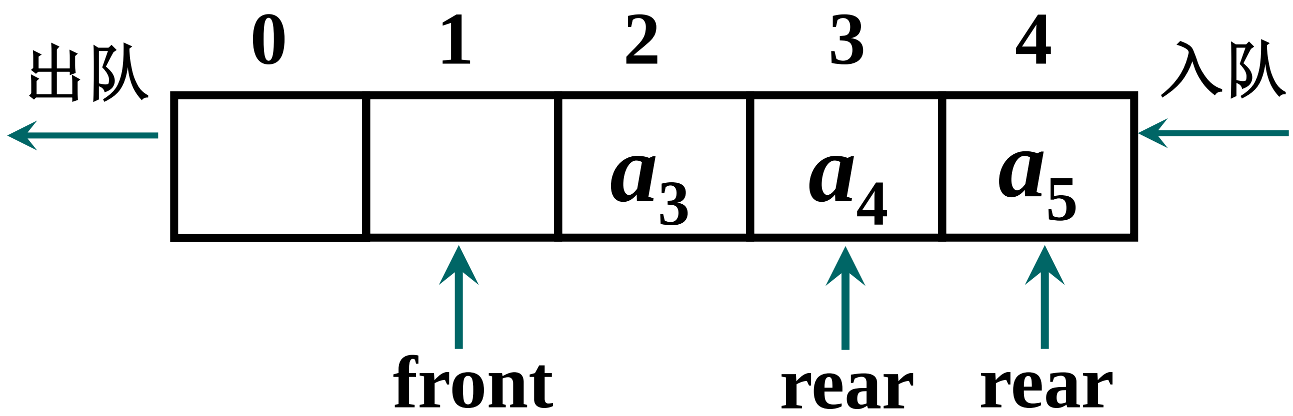
- 队列的顺序存储结构及实现
  - 如何改进出队的时间性能？
    - a_1a_2依次出队



出队操作时间性能提高为$O(1)$

2.4.3 队列的数组实现(Cont.)

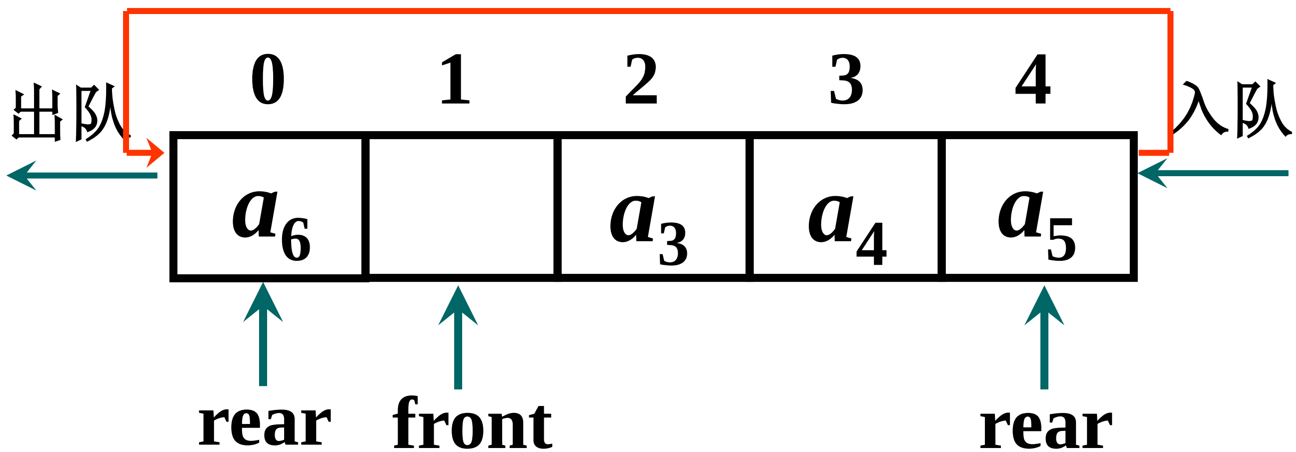
- 队列的顺序存储结构及实现
  - 队列的移动有什么特点？



  - 继续入队会出现什么情况？
  - 假溢出：
    - 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。

2.4.3 队列的数组实现(Cont.)

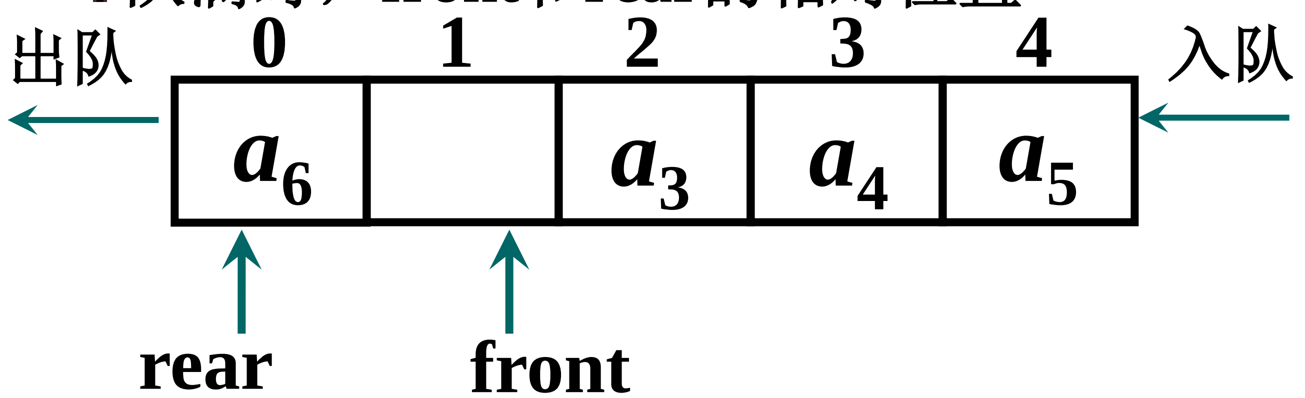
- 队列的顺序存储结构及实现
  - 如何避免假溢出？

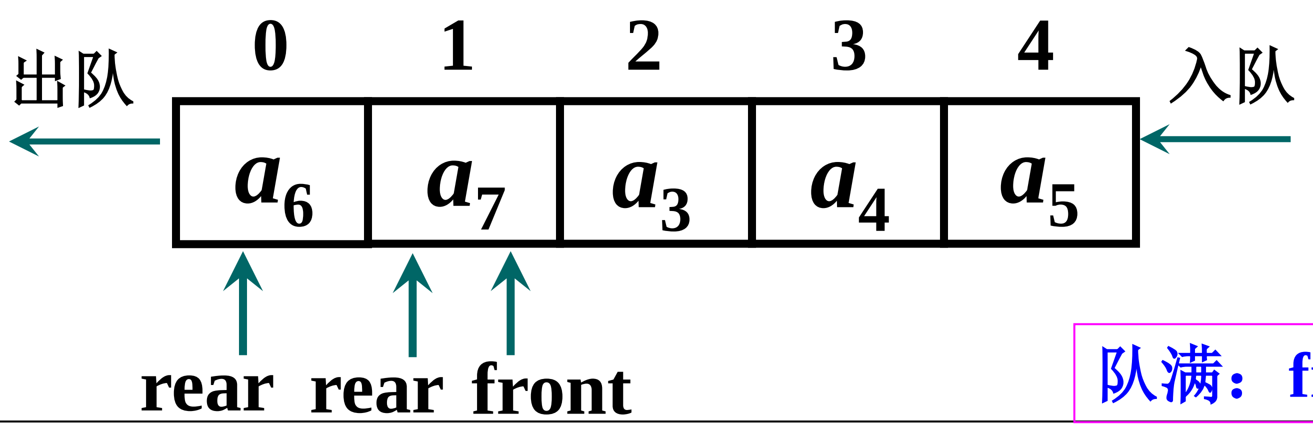


  - 循环数组：将数组最后一个单元的下一个单元看成是0号单元，即把数组头尾相接----按模加一。
  - 循环队列：用循环数组表示的队列。

2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 如何判断循环队列队空和队满？
    - 队满时，front和rear的相对位置

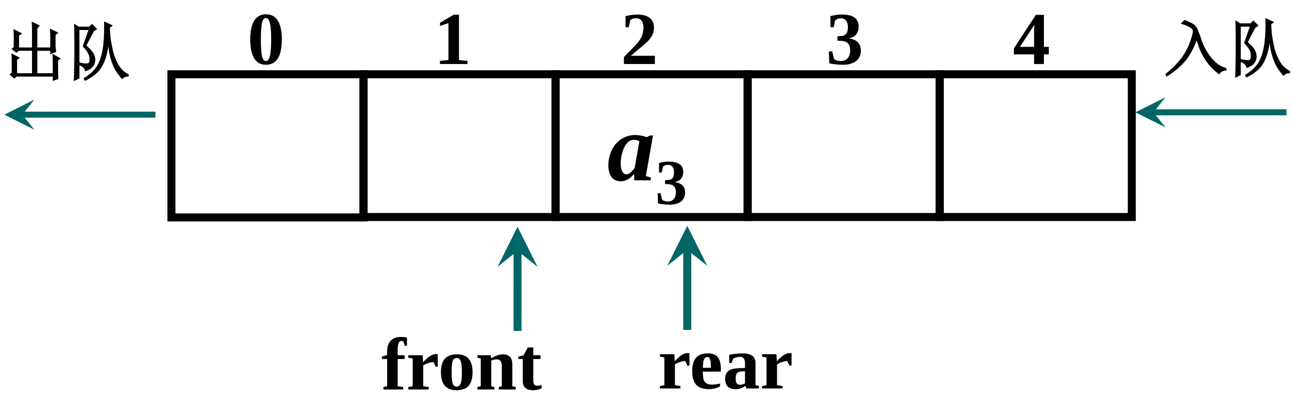


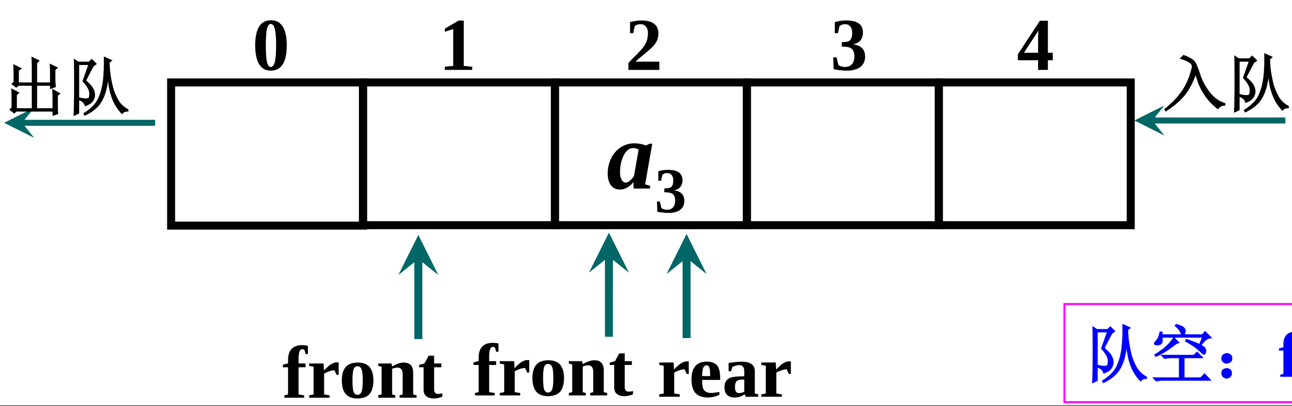


队满：front==rear

2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 如何判断循环队列队空和队满？
    - 队空时，front和rear的相对位置





队空：front==rear

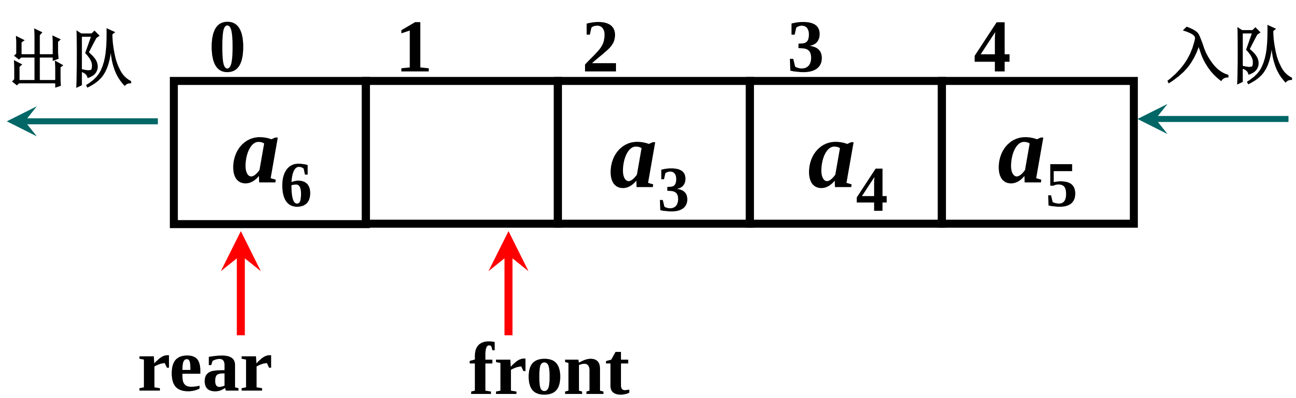
2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 如何区分队空、队满的判定条件？
    - 方法一：增设一个存储队列中元素个数的计数器count:
      - 当front==rear 且count==0时，队空；
      - 当front==rear 且count==MaxSize时，队满；
    - 方法二：设置标志flag,
      - 当front==rear且flag==0时为队空；
      - 当front==rear且flag==1时为队满。
    - 方法三：
      - 保留队空的判定条件：front==rear;
      - 队满判定条件修改为：((rear+1)%MaxSize==front)
      - 代价：浪费一个元素空间，队满时数组中有一个空闲单元；

2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 存储结构的定义

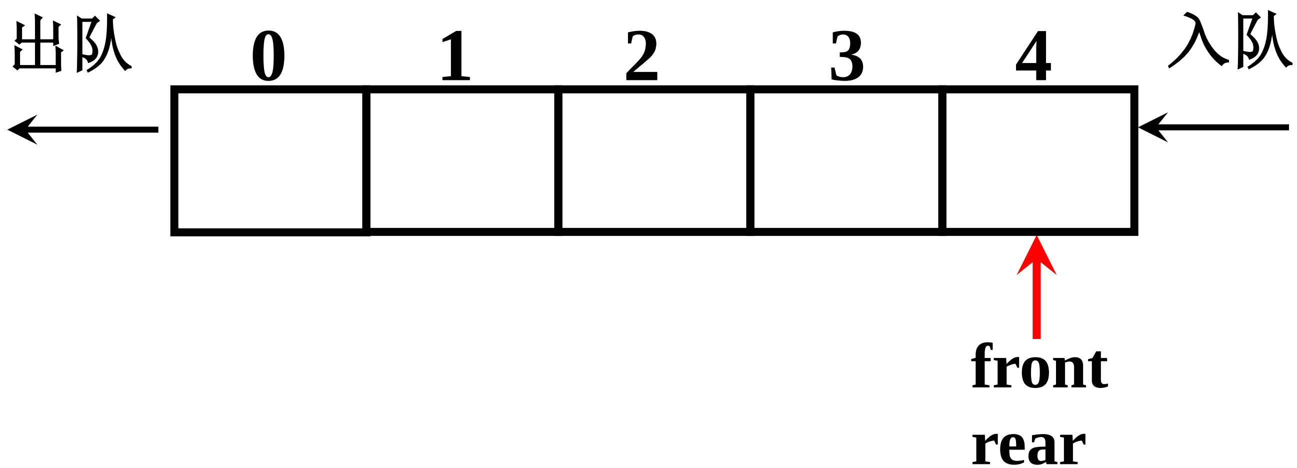
```
struct  QUEUE {
      ElemType  data [ MaxSize ] ;
      int   front ;
      int   rear ;
};   //队列的类型
```



2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 操作的实现---- ①队列初始化

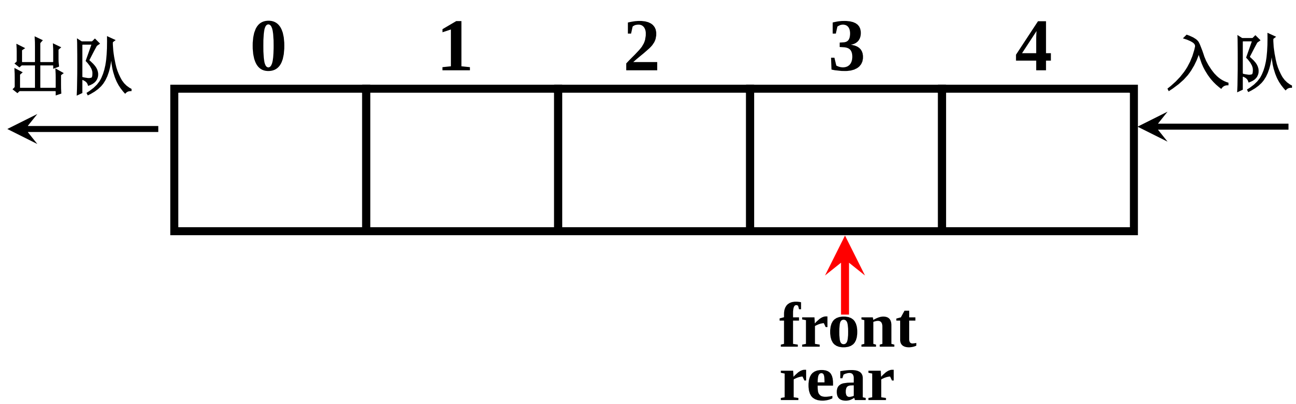
```
void  MakeNull ( QUEUE  &Q)
{
     Q.front = MaxSize-1;
     Q.rear   = MaxSize-1;
}
```



2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 操作的实现---- ②队列判空

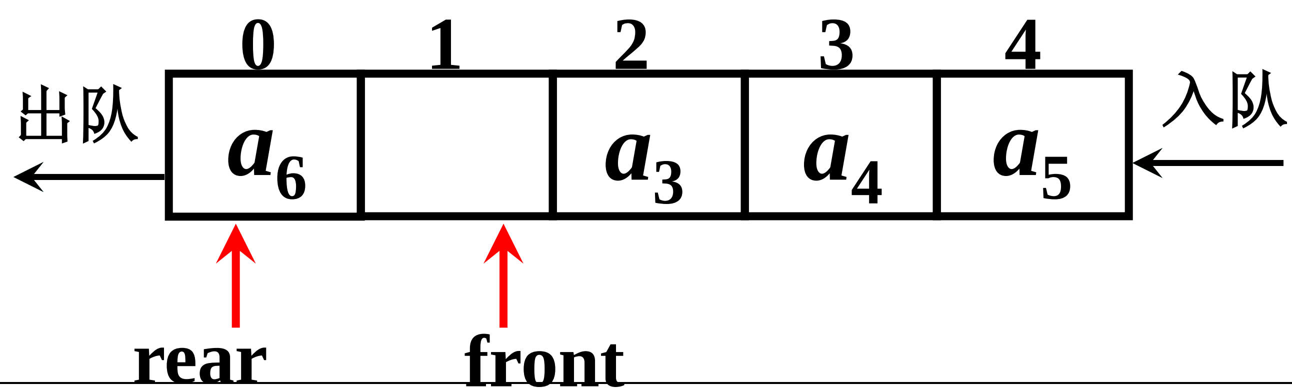
```
bool Empty( QUEUE  Q )
{    if ( Q.rear ==  Q.front  )
          return  TRUE ;
     else
          return  FALSE ;
}
```



2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 操作的实现---- ③返回队首元素

```
ElemType  Front( QUEUE  Q )
{   if  ( Empty( Q ) )    return  NULLESE ;
      else {
               return  (Q.data[(Q.front+1)%MaxSize ] );
      }
}
```

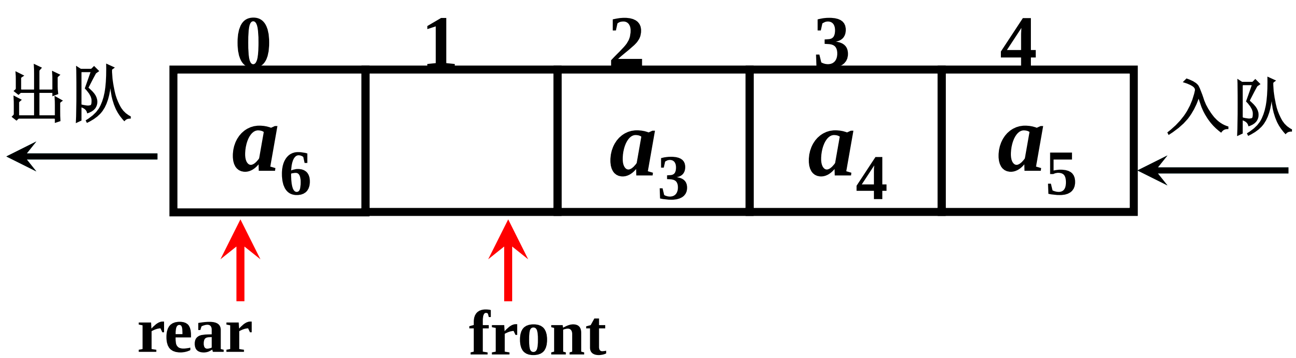


2.4.3 队列的数组实现(Cont.)

队列的顺序存储结构及实现

- 操作的实现---- ④入队

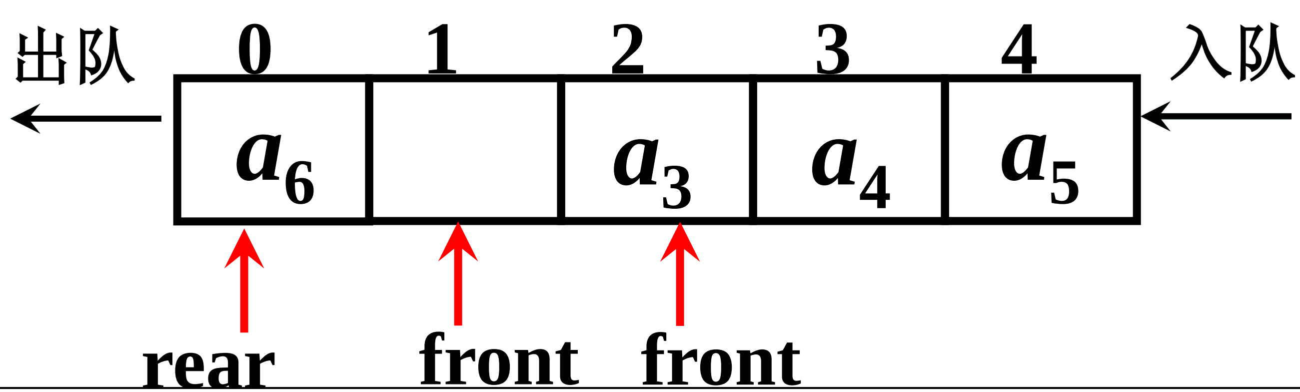
```
void  EnQueue ( ElemType x, QUEUE &Q )
{       if ( (Q.rear+1)%MaxSize ==Q.front )
                cout<< “队列满”；
        else{       Q.rear=(Q.rear+1)%MaxSize ;
                    Q.data[ Q.rear ] = x ;
 }
```



2.4.3 队列的数组实现(Cont.)

- 队列的顺序存储结构及实现
  - 操作的实现---- ⑤出队

```
void  DeQueue (  QUEUE   Q  ) ;
{     if (  Empty ( Q ) )
              cout<< "空队列"  <<endl;
      else
              Q.front = (Q.front+1)%MaxSize ;
}
```



2.4.4 队列的应用

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

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。
- 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。

2.5 特殊线性表—字符串

2.5.1 串的逻辑结构

- 串：零个或多个字符组成的有限序列。
- 串长度：串中所包含的字符个数。
- 空串：长度为0的串，记为：“ ”。
- 非空串通常记为： ***S*=" *s*1 *s*2 …… *sn* "**
  - 其中：*S*是串名，双引号是定界符，双引号引起来的部分是串值，*si*（1≤*i*≤*n*）是一个任意字符。
  - 字符集：ASII码、扩展ASII码、Unicode字符集
- 子串：串中任意个连续的字符组成的子序列。
- 主串：包含子串的串。
- 子串的位置：子串的第一个字符在主串中的序号。

2.5.1 串的逻辑结构

- 串的操作
  - string MakeNull( ) ;
  - bool IsNull (S) ;
  - void In(S, a) ;
  - int   Len(S) ;
  - void Concat(S1, S2) ;
  - string Substr(S, m, n) ;
  - int Index(S, S1) ;
- 与其他线性结构相比，串的操作对象有什么特点？
  - 串的操作通常以串的整体作为操作对象。

例一：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 ) );
            }
}
```

例二：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 串的存储结构

- 顺序串：
  - 用数组来存储串中的字符序列。
- 非压缩形式
  -

……	*C*	*h*	*i*	*n*	*s*	*e*	……

- 压缩形式
  -

……		*C* *h* *i* *n*	*e* *s* *e*			……

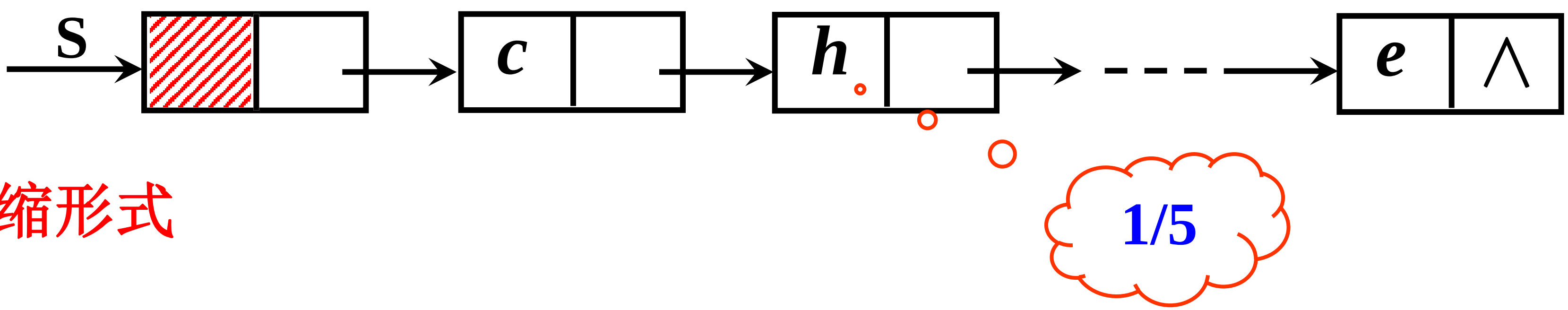
2.5.2 串的存储结构(Cont.)

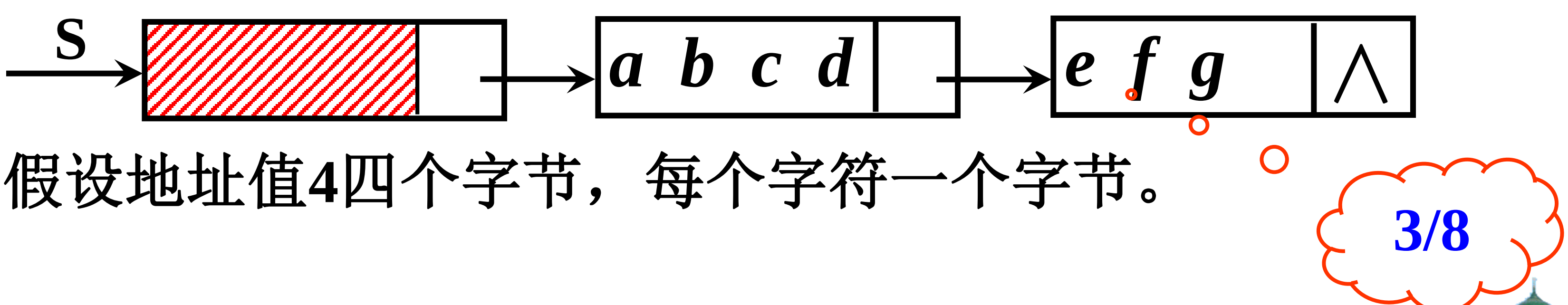
- 如何表示串的长度？
  - 方法一：用一个变量来表示串的实际长度，同一般线性表
  - 方法二：在串尾存储一个不会在串中出现的特殊字符作为串的终结符，表示串的结尾。

0	1	2	3	4	5	6	…… …… Max−1	
C	*h*	*i*	*n*	*e*	*s*	*e*	空 闲	7

0	1	2	3	4	5	6	7	…… …… Max−1
C	*h*	*i*	*n*	*e*	*s*	*e*	\0	空 闲

2.5.2 串的存储结构(Cont.)

- 链接串：
  - 用链接存储结构来存储串。
- 非压缩形式
  - 

- 压缩形式
  - 

  - 假设地址值4四个字节，每个字符一个字节。
  - 如何实现压缩形式的字符串的插入和删除等操作？

2.5.3 模式匹配

- 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)
  - 基本思想
    - 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。

2.5.3 模式匹配(Cont.)

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，模式串指针需复位（j=0）。

2.5.3 模式匹配(Cont.)

- 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。

2.5.3 模式匹配(Cont.)

```
int StrMatch_BF ( char* S, char* T, int pos=0)
{ /*S为主串T为模式，长度分别为了lenS和lenT；串采用顺序存储结构*/
   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;
   else
       return -1;               // 匹配不成功
}
```

2.5.3 模式匹配(Cont.)

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)。

2.5.3 模式匹配(Cont.)

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).

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)。

2.5.3 模式匹配(Cont.)

KMP算法----改进的模式匹配算法

- 为什么BF算法时间性能低？
  - 在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果。

- 简单的匹配算法：一旦某个字符匹配失败，从头开始。
- （本次匹配起点后一个字符开始）
- 主串 S=000000000000000000000000000000001

- 52个零

-
- 模式串 T=00000001，指针要回朔45次。

2.5.3 模式匹配(Cont.)

- KMP算法----改进的模式匹配算法
  - 为什么BF算法时间性能低？
    - 在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果。
  - 如何在匹配不成功时主串不回溯？
    - 主串不回溯，模式就需要向右滑动一段距离。
  - 如何确定模式的滑动距离？
    - 利用已经得到的“部分匹配”的结果
    - 将模式向右“滑动”尽可能远的一段距离后，继续进行比较

2.5.3 模式匹配(Cont.)

- 假设主串ababcabcacbab，模式abcac，KMP算法的匹配过程示例：
- 第1趟匹配

```
  ↓ i=2
a b a b c a b c a c b a b
a b c
  ↑j=2
```

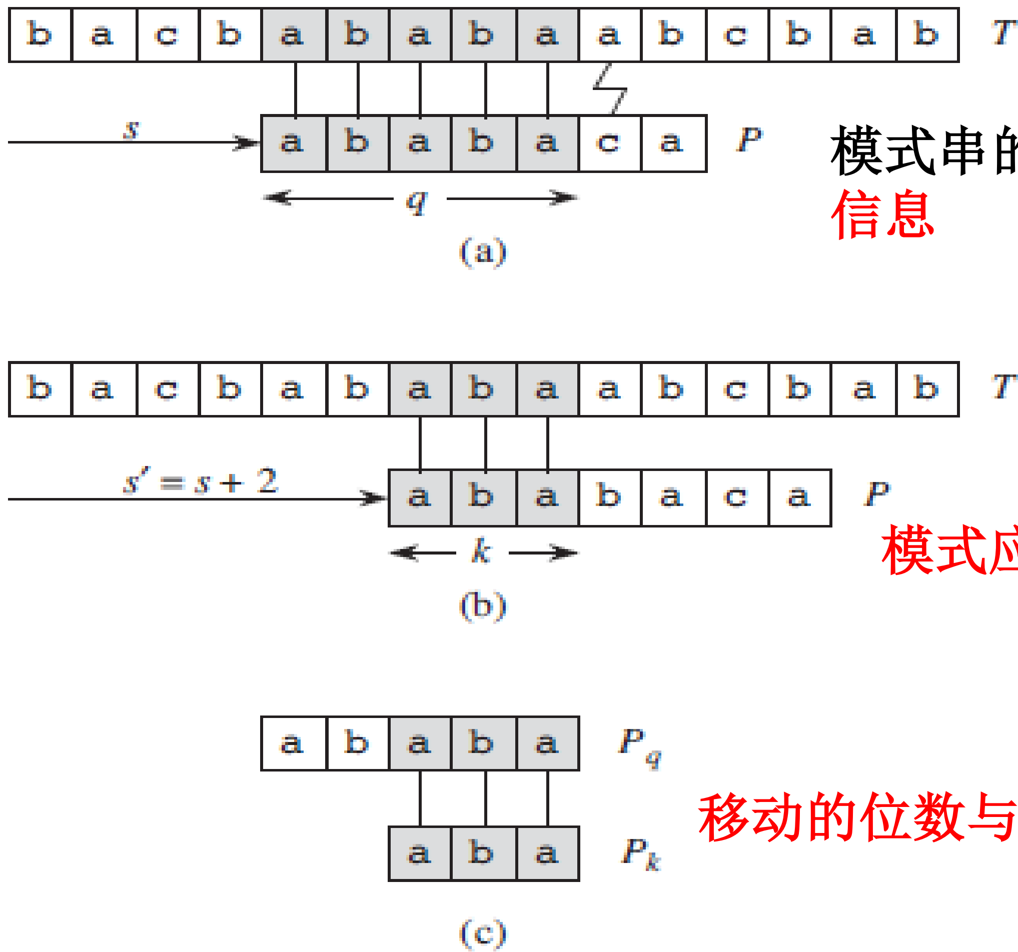
第2趟匹配

```
  ↓ i=2----6
a b a b c a b c a c b a b
    a b c a c
    ↑ j=0
```

第3趟匹配

```
            ↓ i=6
a b a b c a b c a c b a b
          a b c a c
            ↑j=1
```

KMP算法（图解）

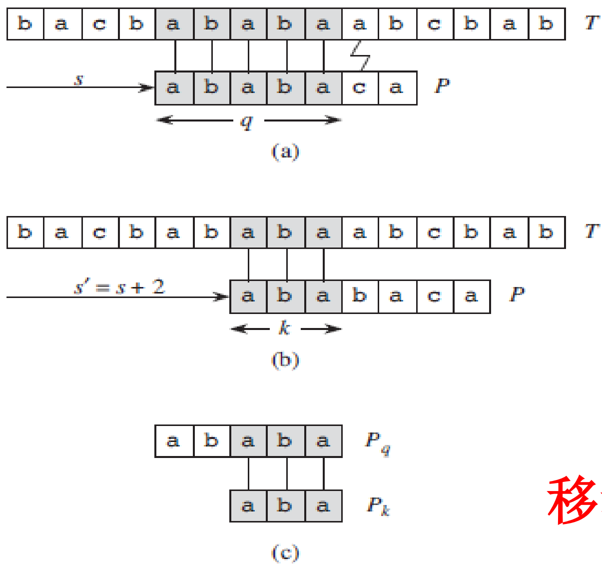


模式串的部分匹配为下次匹配位置提供信息

模式应该向右滑多远才是最高效率的?

移动的位数与模式串某位的自身最大前缀有关

KMP算法（图解）



模式串的部分匹配为下次匹配位置提供信息

模式应该向右滑多远才是最高效率的?

移动的位数与模式串某位的自身最大前缀有关

- （a图）：**前5个字符已经匹配成功**，**naive**算法接着**移到 s + 1**。但是明显的 s + 1 处是明显无效的。
- （b）图：**s + 2前三个字符都可以匹配**，所以**很可能是匹配点**。
- 数组**π** 记录的就是这些信息，比如对于P，上边的例子**π [5] = 3**，则下一个可能的位移是s′= s + （q – π [q]），即s′= s + 2。也就是在匹配过程中，**用π 数组记录下一次可能匹配位置的信息**。

KMP算法（前缀函数）

[前缀函数]：给定模式串P[1..m]，P的前缀函数

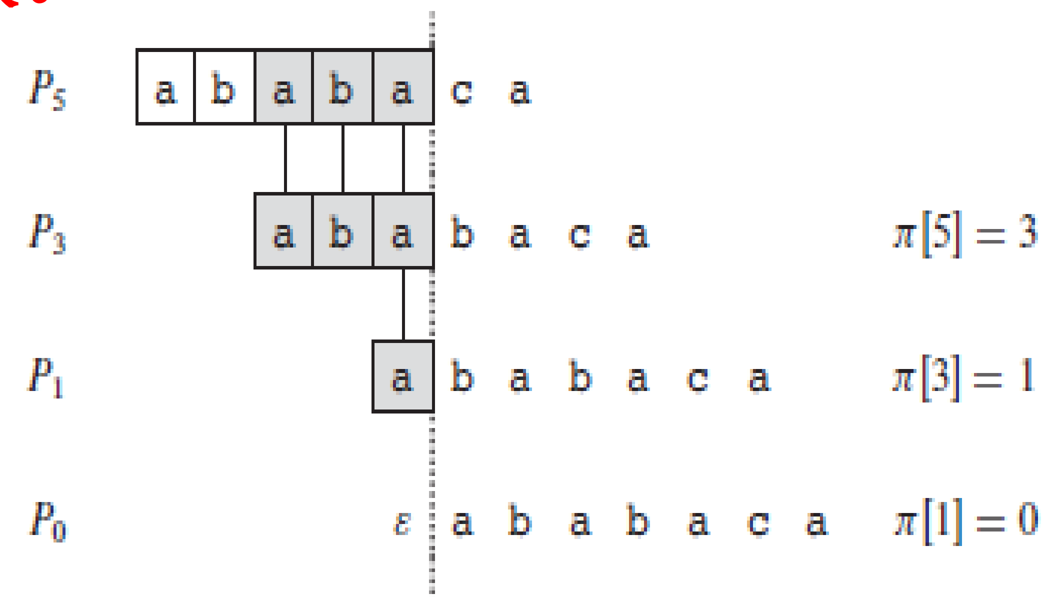
π： $\{1, 2, \cdots, m\} \rightarrow \{0, 1, \cdots, m-1\}$，满足

$\pi[q]=\max\{k: k<q$，并且P_k是P_q的后缀$\}$

也有称为失配函数。

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

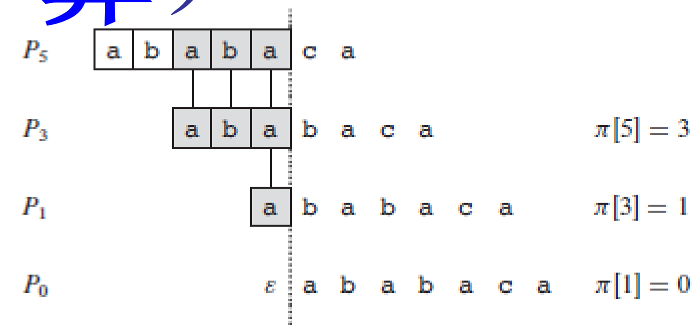
KMP算法（匹配算法）

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0                            //Number of characters matched.
for i ← 1 to n                   //Scan the text from left to right.
       do while q > 0 and P[q + 1] ≠ T[i]
            do q ← π[q]   //end while //Next character does not match.
       if P[q + 1] = T[i]
           then q ← q + 1       //Next character matches.
       if q = m                  //Is all of P matched?
            then print "Pattern occurs with shift" i – m
                     q ← π[q]    //Look for the next match.
```

KMP算法（前缀函数计算）

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

$$\pi(q)=\begin{cases} 0, & q=0 \\ \pi^m(q-1)+1, & \text{其中m是满足等式} P[\pi^l(q-1)+1] = P[q] \text{ 的最小整数} l \\ 0, & \text{没有满足上式的} l \end{cases}$$

$\Pi^1(q) = \pi(q)$， $\pi^1(q) = \pi(\pi^{1-1}(q))$

```
COMPUTE-PREFIX-FUNCTION(P)
 m ← length[P]
 π[1] ← 0
 k ← 0
 for q ← 2 to m
     do while k > 0 and P[k + 1] ≠ P[q]
            do k ← π[k]
        if P[k + 1] = P[q]
           then k ← k + 1
        π[q] ← k
 return π
```