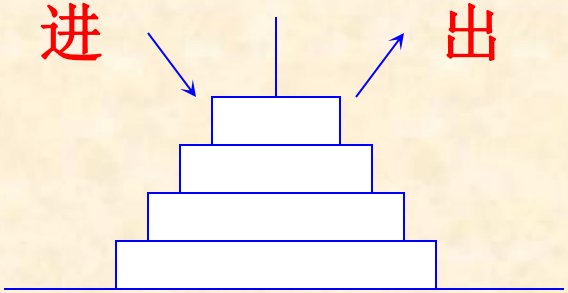
# 第三章

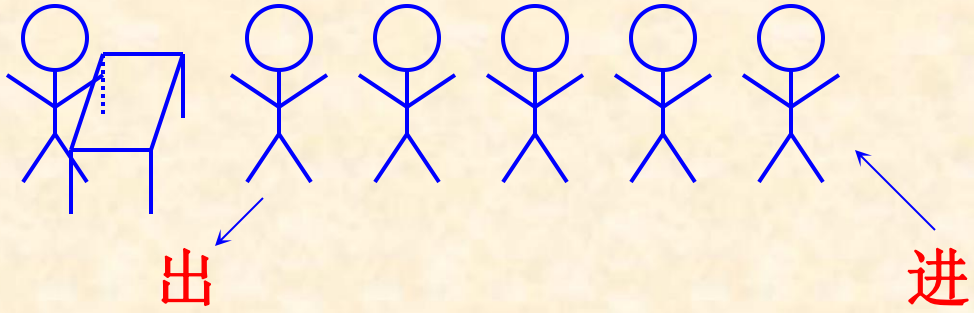
# 栈和队列

# 第三章　　栈和队列

汉诺塔




排队
买票



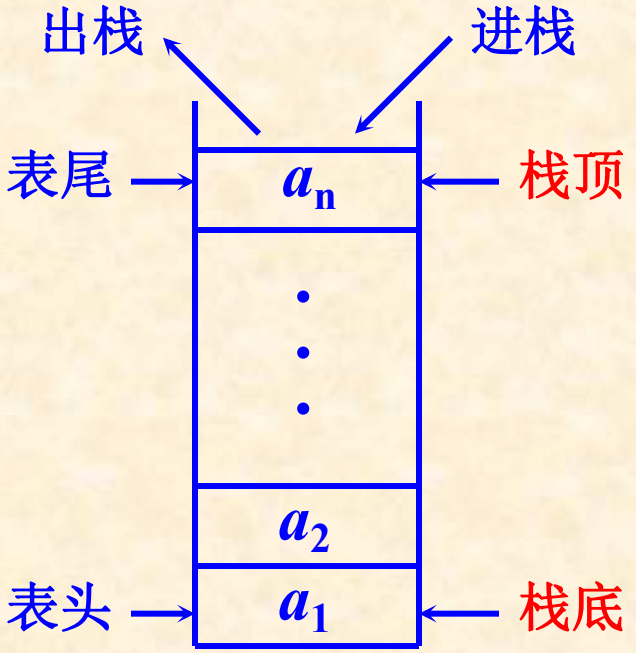

- 栈和队列是两种重要的线性结构
- 栈和队列是操作受限的线性表

# 3.1 栈

## 3.1.1 栈的定义

栈是限定仅在表尾进行插入或删除操作的线性表。

通常，表头端称为栈底，表尾端称为栈顶。




$a_1$为栈底元素

$a_n$为栈顶元素

按$a_1$、$a_2$…、$a_n$顺序进栈

按$a_n$…、$a_2$、$a_1$顺序出栈

栈称为后进先出线性表(LIFO)

# 3.1 栈的类型定义

**ADT Stack {**

**数据对象：**

$D = \{ a_i \mid a_i \in ElemSet,\ i=1,2,...,n,\ n \geq 0 \}$

**数据关系：**

$R1 = \{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\ i=2,...,n \}$

约定$a_n$ 端为栈顶，$a_1$ 端为栈底。

**基本操作：**

**} ADT Stack**

InitStack(**Stack &**S)

DestroyStack(**Stack &**S)

StackLength(**Stack** S)

StackEmpty(**Stack** S)

GetTop(**Stack** S, **&**e)

ClearStack(**Stack &**S)

Push(**Stack &**S, ElemType e)

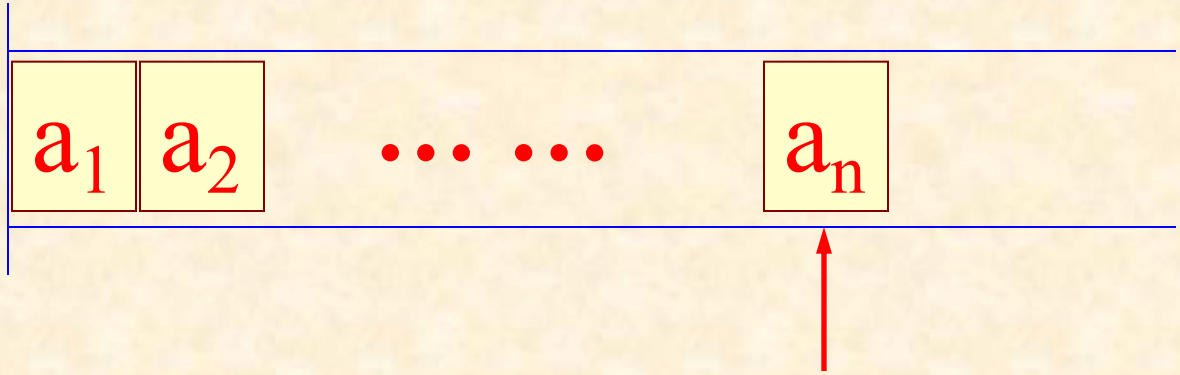
Pop(**Stack &**S, ElemType **&**e)

StackTravers(**Stack** S, visit())

## GetTop(S, &e)

初始条件：栈 S 已存在且非空。

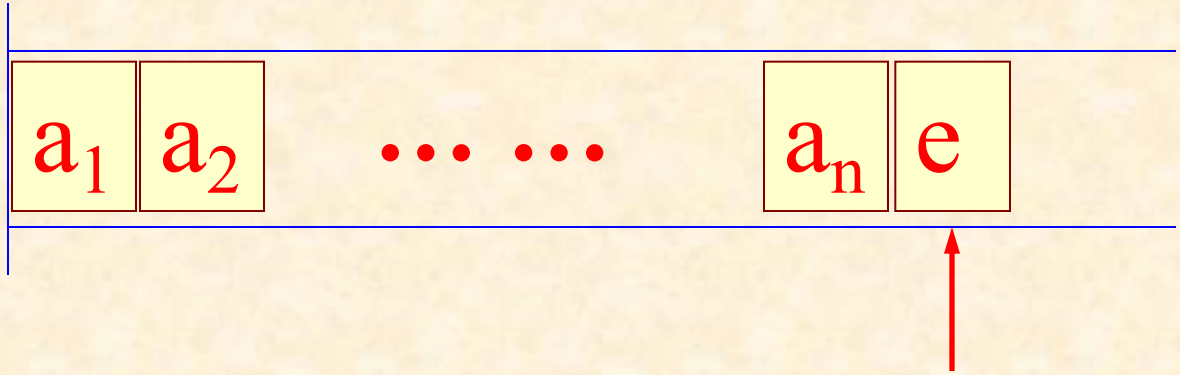
操作结果：用 e 返回 S 的栈顶元素。



- Push(&S, e)

初始条件：栈 S 已存在。

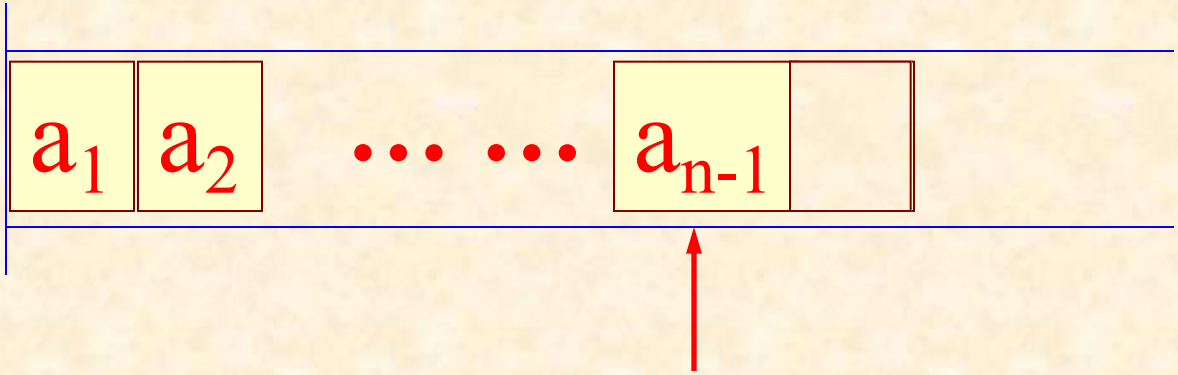
操作结果：插入元素 e 为新的栈顶元素。



- Pop(&S, &e)

初始条件：栈 S 已存在且非空。

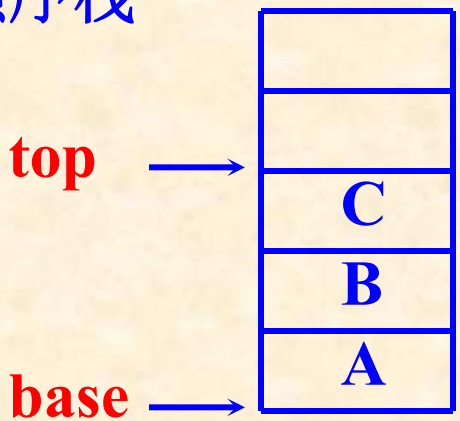
操作结果：删除 S 的栈顶元素，并用 e 返回其值。



## 3.1.2 栈的表示和实现

顺序栈；链式栈。

### 1. 顺序栈

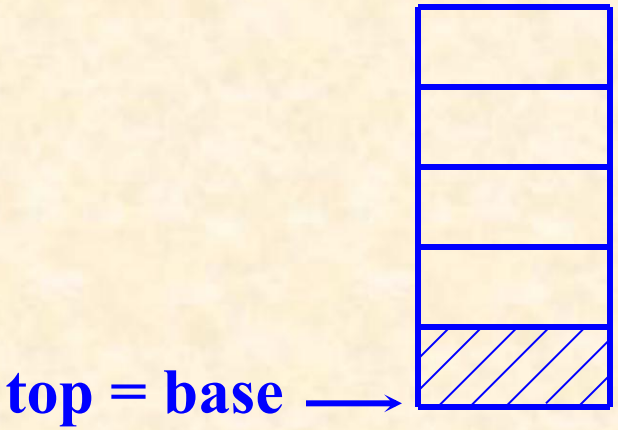



栈底指针——base

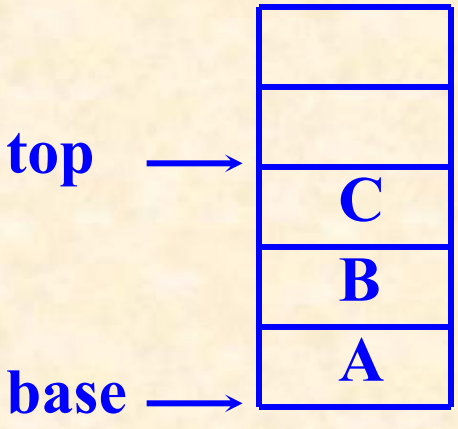
栈顶指针——top

栈容量

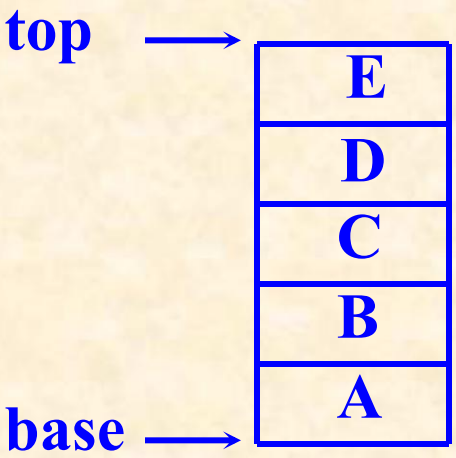
```
typedef   struct  {
      SElemType     * base ;
      SElemType     * top ;
       int                  stacksize ;
} SqStack ;
```




空栈

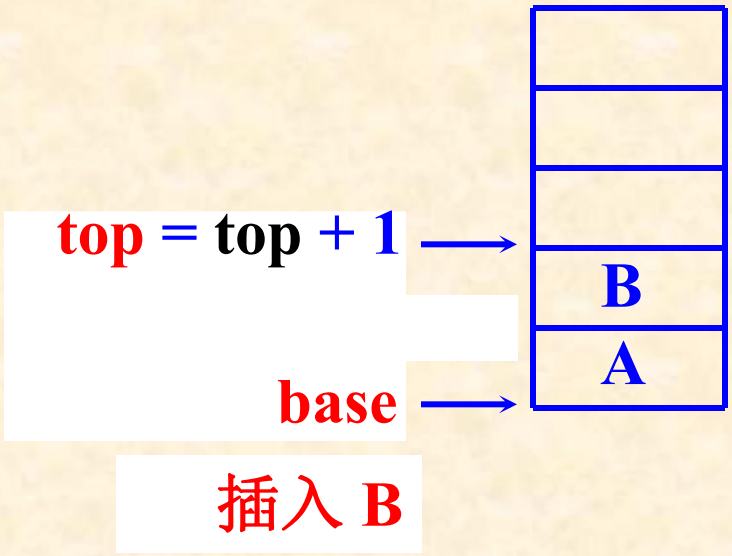



栈中有三个元素

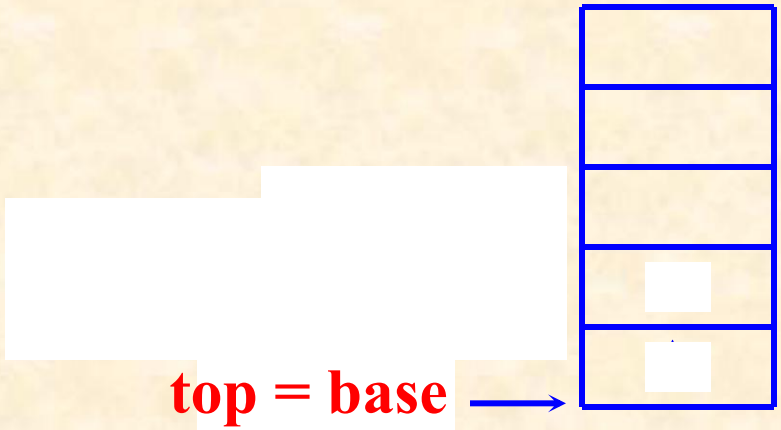



满栈

例，在栈中插入元素 A 和 B



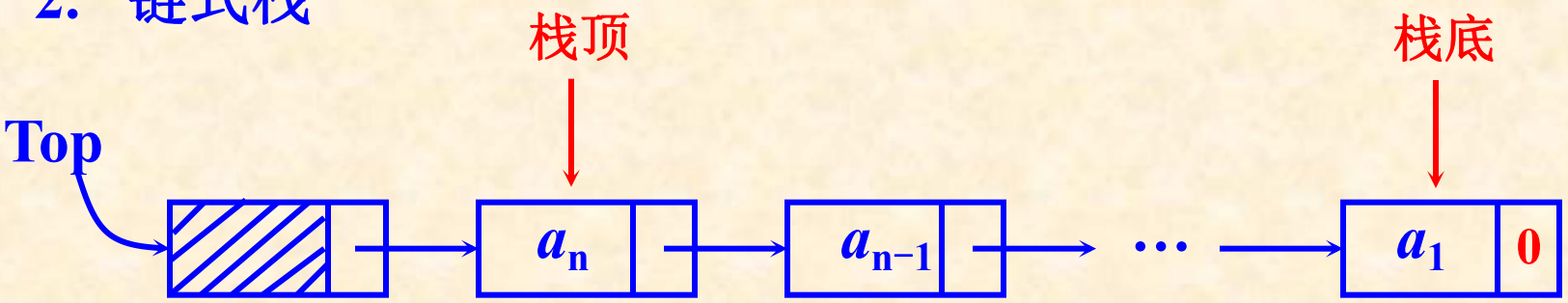
例，删除栈顶元素 B 和 A




删除 C

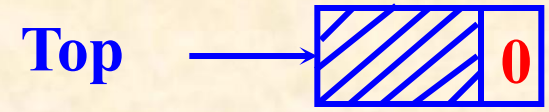
ERROR

## 2. 链式栈




```
typedef   struct   SNode  {
        SElemType          data ;
        struct  SNode      * next ;
} SNode ,  * LinkStack ;
```

空表:



# 例一、 数制转换

算法基于原理：

$$N = (N \text{ div } d) \times d + N \bmod d$$

例如：$(1348)_{10}=(2504)_8$，其运算过程如下：

| N | N div 8 | N mod 8 |
|---|---|---|
| 1348 | 168 | 4 |
| 168 | 21 | 0 |
| 21 | 2 | 5 |
| 2 | 0 | 2 |

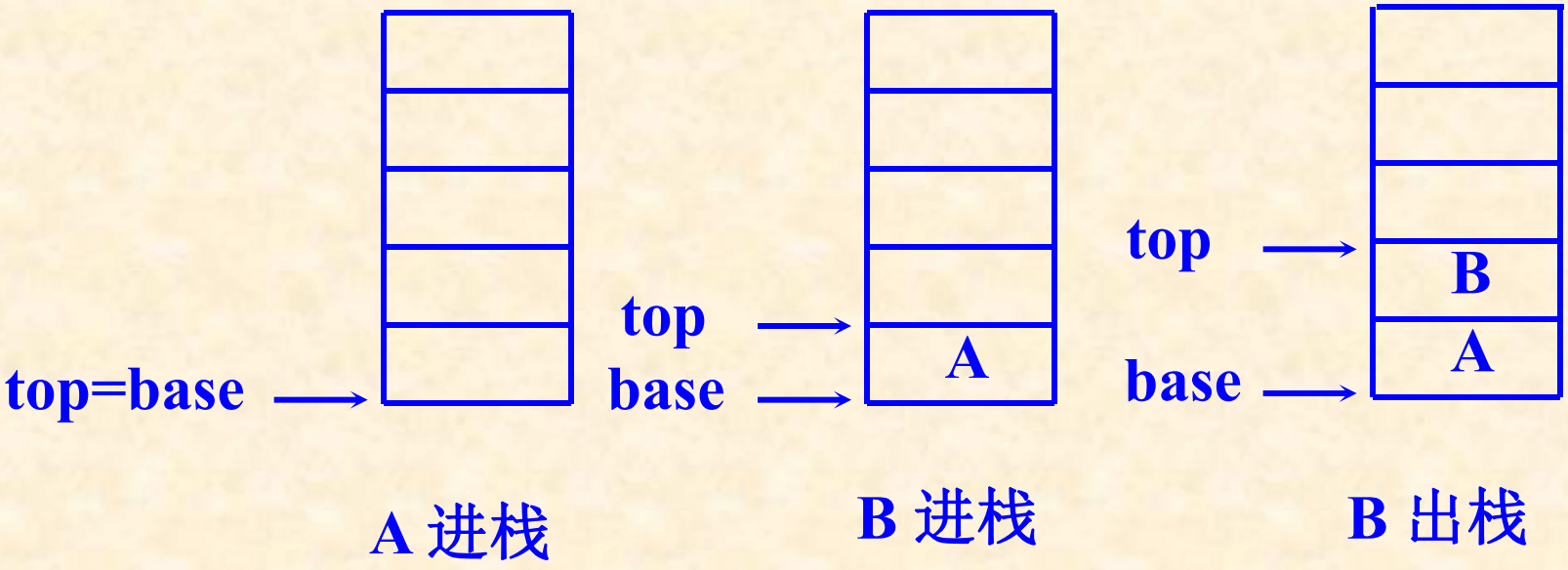
计算顺序

输出顺序

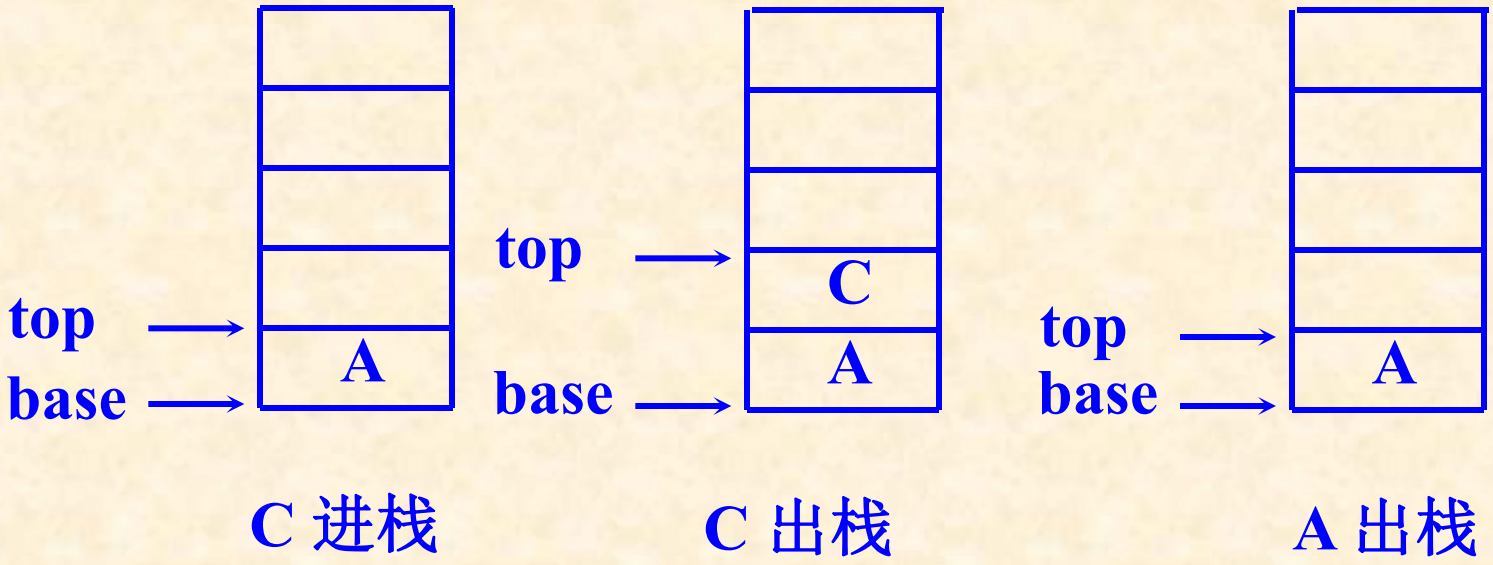
```
void conversion () {
    InitStack(S);
    scanf ("%d",&N);
    while (N) {
      Push(S, N % 8);
      N = N/8;
    }
    while (!StackEmpty(S)) {
      Pop(S,e);
      printf ( "%d", e );
    }
} // conversion
```

# 3.2 栈的应用

假设S和X分别表示进栈和出栈操作，由输入序列“ABC”得到输出序列“BCA”的操作序列为SSXSXX，则由“a*b+c/d”得到“ab*cd/+”的操作序列为?

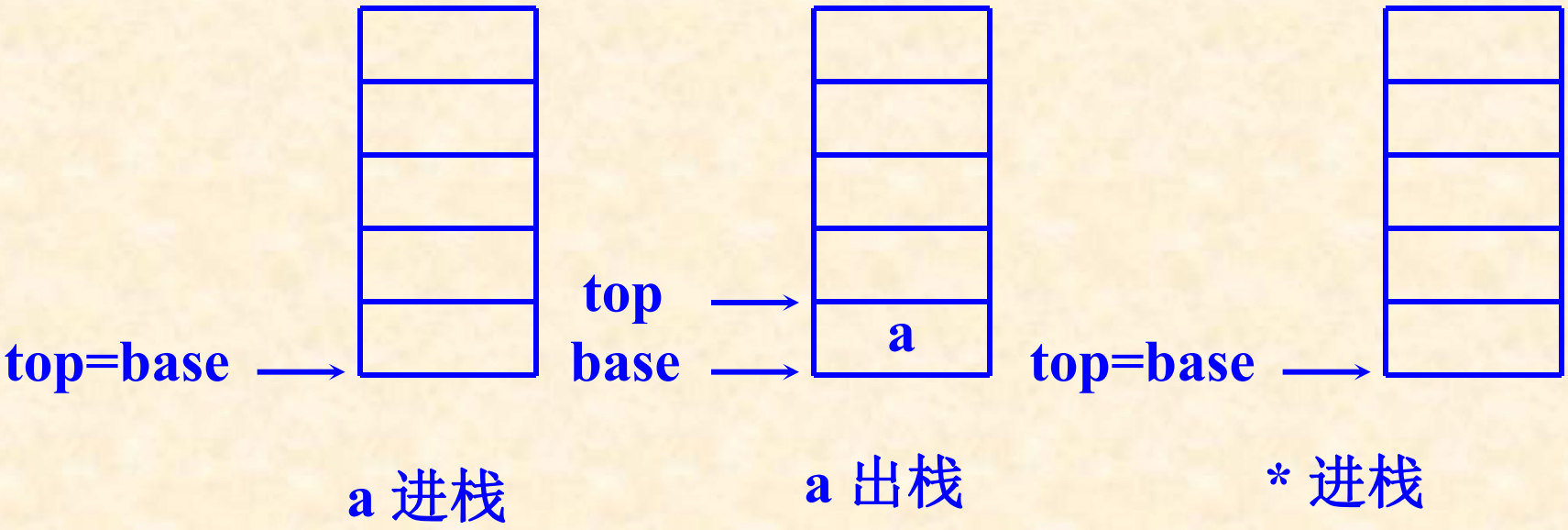



操作序列：S　S　X



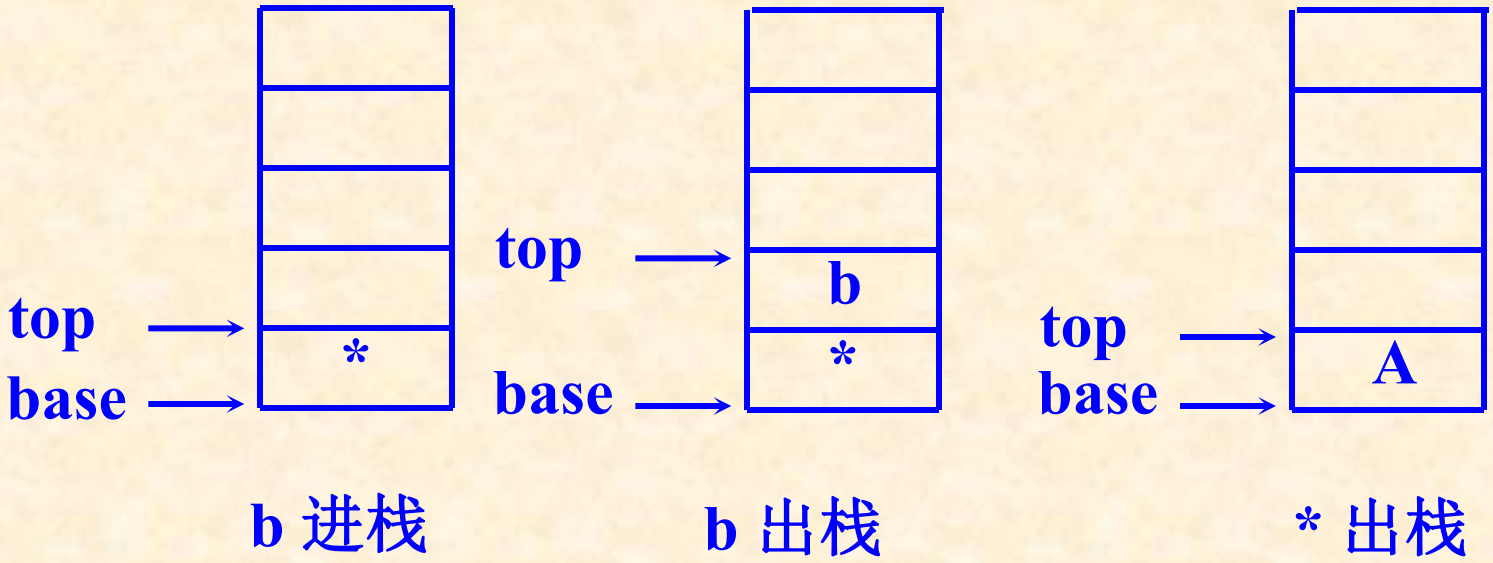

操作序列：S S X S X X

输出：B C A




操作序列：S X S

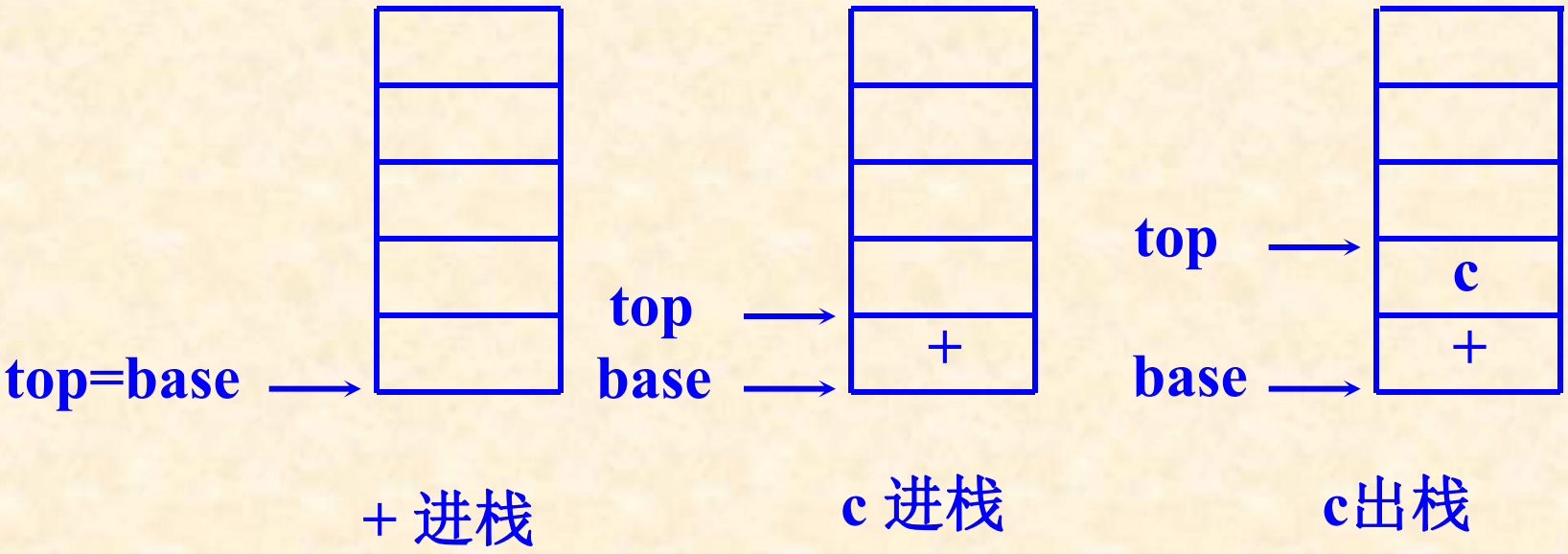
输出：a




b 进栈　　b 出栈　　* 出栈

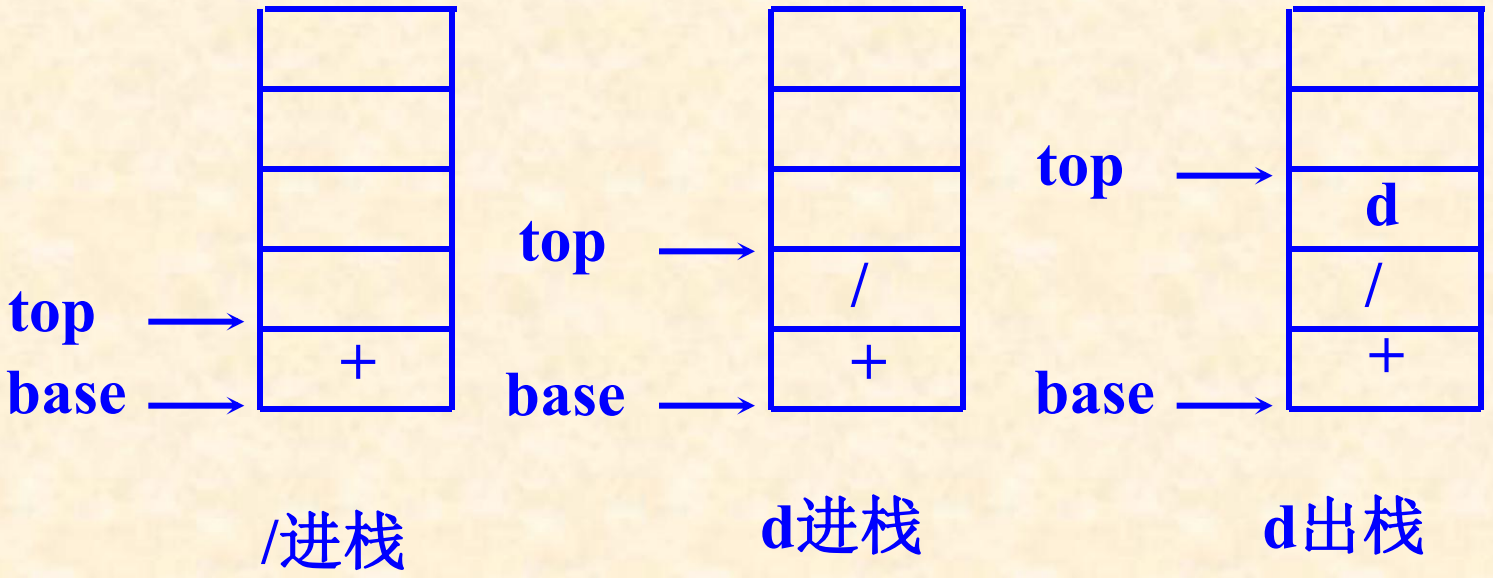
操作序列：S X S S X X

输出：a b *



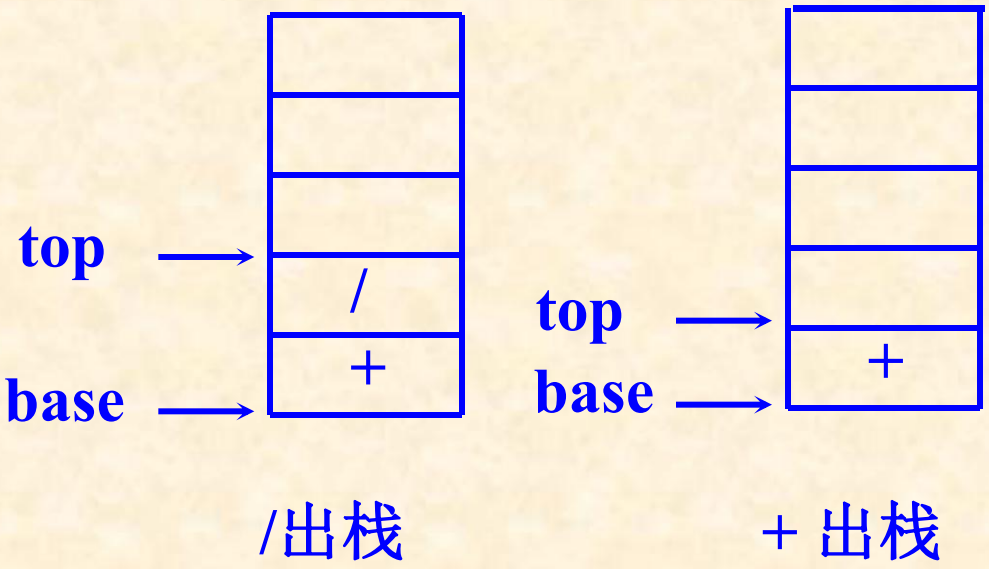

操作序列：S X S S X X S S X

输出：a b * c




操作序列：S X S S X X S S X S S X

输出：a b * c d




操作序列：S X S S X X S S X S S X X X

输出：a b * c d / +

# 例二、括号匹配的检验

假设在表达式中

（［］（））或［（［］［］）］

等为正确的格式，

［（］）或（［（））或((()])

均为不正确的格式。

则 检验括号是否匹配的方法可用

“期待的急迫程度”这个概念来描述。

例如：考虑下列括号序列：

[ ( [ ] [ ] ) ]

1 2 3 4 5 6 7 8

## 算法的设计思想：

1）凡出现左括弧，则进栈；

2）凡出现右括弧，首先检查栈是否空

若栈空，是则表明该“右括弧”多余否则和栈顶元素比较，若相匹配，则“左括弧出栈”否则表明不匹配

3）表达式检验结束时，

若栈空，则表明表达式中匹配正确

否则表明“左括弧”有余

```
Status matching(string& exp) {
  int state = 1;
  while (i<=Length(exp) && state) {
    switch (exp[i]) {
      case 左括弧:{Push(S,exp[i]); i++; break;}
      case"右括弧" : {
        if(!StackEmpty(S)&&GetTop(S)="(")
          {Pop(S,e);  i++;}
        else {state = 0;}
        break;  }   … …
  }
  if (StackEmpty(S)&&state) return OK; ……
```

# 例三、行编辑程序问题

如何实现？

“每接受一个字符即存入存储器”？

并不恰当！

在用户输入一行的过程中，允许用户输入出差错，并在发现有误时可以及时更正。

合理的作法是：

设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区；并假设“#”为退格符，“@”为退行符。

假设从终端接受了这样两行字符：

**wh**li##**ilr**#**e**（s#*s)

outcha@**putchar**(*s=#++);

则实际有效的是下列两行：

**while** (*s)

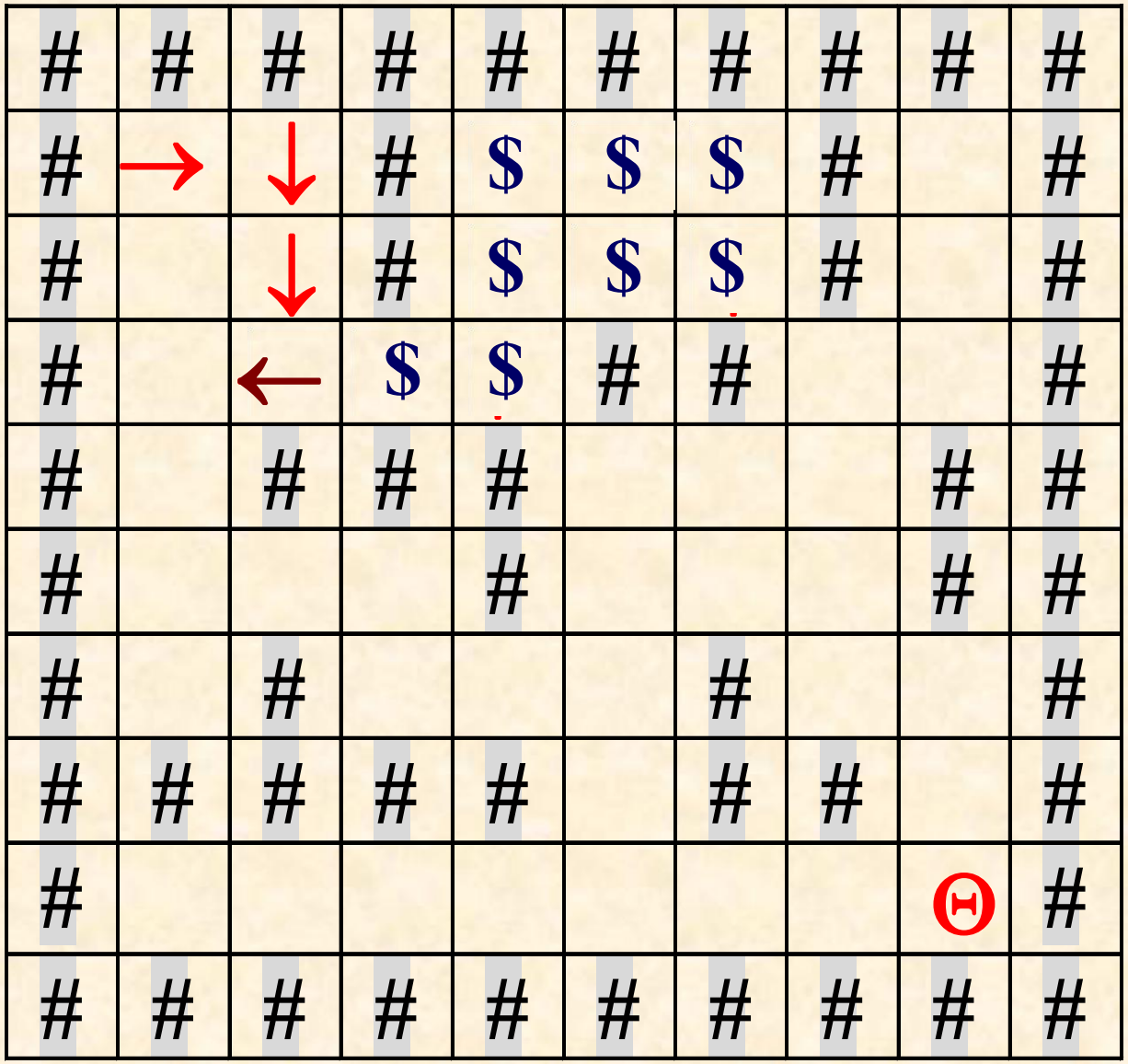
**putchar**(*s++);

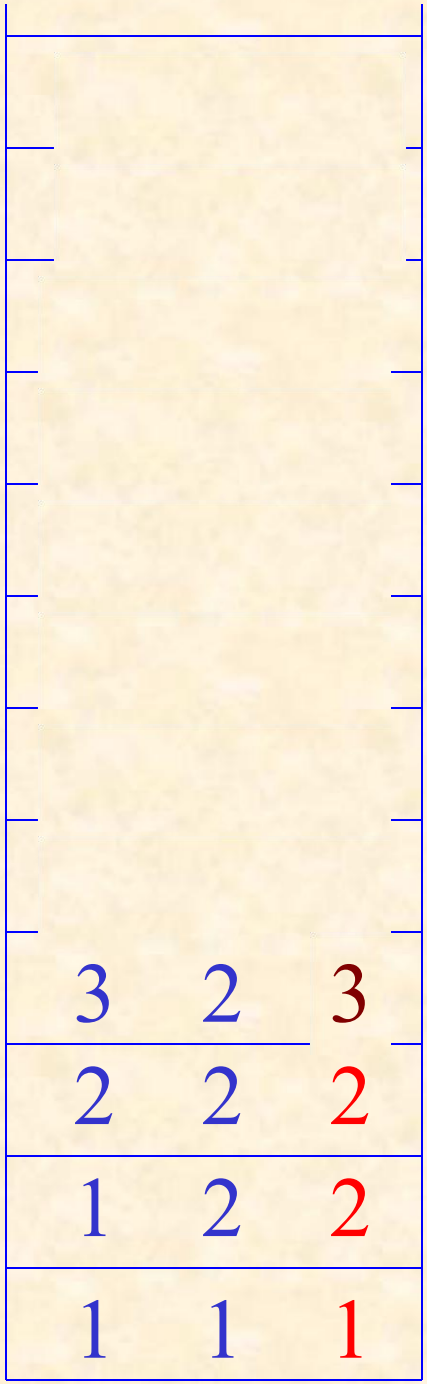
```
while (ch != EOF) { //EOF为全文结束符
    while (ch != EOF && ch != '\n') {
      switch (ch) {
       case '#' : Pop(S, c); break;
       case '@': ClearStack(S); break;// 重置S为空栈
       default : Push(S, ch);  break;
      }
      ch = getchar();  // 从终端接收下一个字符
    }
    将从栈底到栈顶的字符传送至调用过程的
    数据区;
    ClearStack(S);      // 重置S为空栈
    if (ch != EOF)  ch = getchar(); }
```

# 例四、 迷宫求解

通常用的是“穷举求解”的方法

| # | # | # | # | # | # | # | # | # | # |
|---|---|---|---|---|---|---|---|---|---|
| # | → | ↓ | # | $ | $ | $ | # |  | # |
| # |  | ↓ | # | $ | $ | $ | # |  | # |
| # | ↓ | ← | $ | $ | # | # |  |  | # |
| # | ↓ | # | # | # |  |  |  | # | # |
| # | → | → | ↓ | # |  |  |  | # | # |
| # |  | # | → | → | ↓ | # |  |  | # |
| # | # | # | # | # | ↓ | # | # |  | # |
| # |  |  |  |  | → | → | → | Θ | # |
| # | # | # | # | # | # | # | # | # | # |


#
$
Θ


3 2 3
2 2 2
1 2 2
1 1 1

# 求迷宫路径算法的基本思想是：

- 若当前位置“可通”，则纳入路径，继续前进；
- 若当前位置“不可通”，则换方向继续探索；
- 若四周“均无通路”，则将当前位置从路径中删除出去。

# 求迷宫中一条从入口到出口的路径的算法：

设定当前位置的初值为入口位置；

do {

若当前位置可通，

则 {将当前位置插入栈顶；

若该位置是出口位置，则算法结束；

否则切换当前位置的东邻方块为

新的当前位置；

}

否则 { … …

}

} while (栈不空）；

若栈不空且栈顶位置尚有其他方向未被探索，

则设定新的当前位置为: 沿顺时针方向旋转

找到的栈顶位置的下一相邻块；

若栈不空但栈顶位置的四周均不可通，

则｛删去栈顶位置；// 从路径中删去该通道块

若栈不空，则重新测试新的栈顶位置，

直至找到一个可通的相邻块或出栈至栈空；

｝

若栈空，则表明迷宫没有通路。

# 例五、 表达式求值

限于二元运算符的表达式定义:

表达式 ::= (操作数) + (运算符) + (操作数)

操作数 ::= 简单变量 | 表达式

简单变量 :: = 标识符 | 无符号整数

# 计算机中表达式的三种表示方法：

设 Exp = S1 + **OP** + S2

则称 **OP** + S1 + S2 为前缀表示法

S1 + **OP** + S2 为中缀表示法

S1 + S2 + **OP** 为后缀表示法

例如：　**Exp** = a × b + (c – d / e) × f

前缀式：　+ × a b × – c / d e f

中缀式：　a × b + c – d / e × f

后缀式：　a b × c d e / – f × +

结论：　5)后缀式的运算规则为：

1)运算符在式中出现的顺序恰为运算的先后次序；

2)操作数之间的相对次序不变；

3)运算符总是跟在与之相邻的操作数之后；

4)前缀式运算规则为：从右至左扫描，将出现的连续两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式；

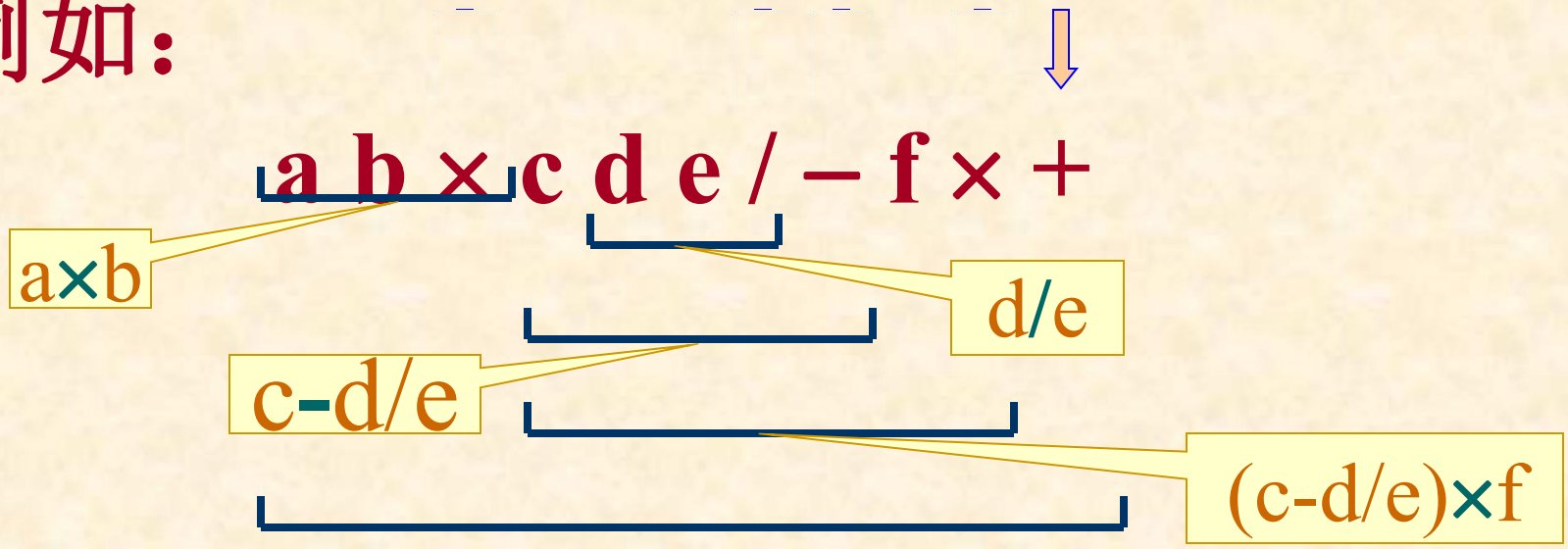
5)后缀式的运算规则为：从左至右扫描，将出现的紧靠在一个运算符之前的两个操作数构成一个最小表达式；

# 如何从后缀式求值？

先找运算符，

再找操作数

例如：



## ●如何从原表达式求得后缀式?

分析“原表达式”和“后缀式”中的运算符:

原表达式: **a + b × c – d / e × f**

后缀式: **a b c × + d e / f × –**

每个运算符的运算次序要由它之后的一个运算符来定，在后缀式中,优先数高的运算符领先于优先数低的运算符。

# 从原表达式求得后缀式的规律为：

1) 设立暂存运算符的**栈**;

**2) 设表达式的结束符为“#”，**
**预设运算符栈的栈底为“#”**

3) 若当前字符是操作数，
则直接发送给后缀式;

## 从原表达式求得后缀式的规律为：

4) 若当前运算符的优先数高于栈顶运算符，则进栈;

5) 否则，退出栈顶运算符发送给后缀式;

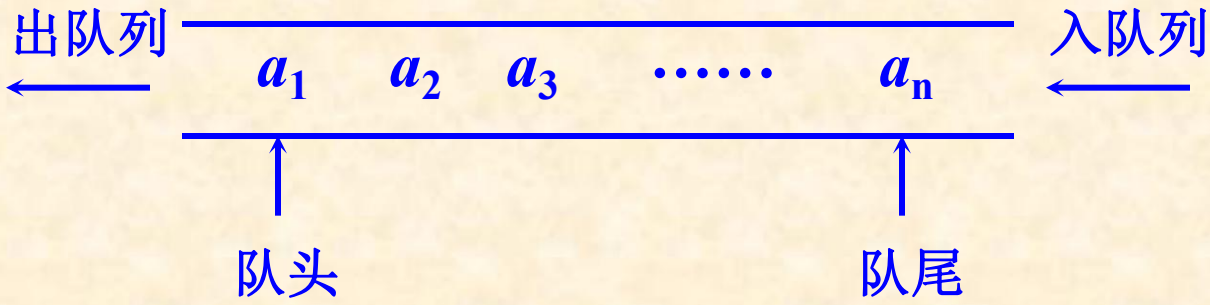
6) “(”对它之前后的运算符起隔离作用，“)”可视为自相应左括弧开始的表达式的结束符。

# 3.4 队 列

## 3.4.1 队列的定义

队列是一种先进先出(FIFO)的线性表。

队列只允许在一端进行插入，而在另一端进行删除。



主要基本操作包括:

**QueueEmpty ( Q )**

**QueueLength ( Q )**

**GetHead ( Q，&e )**

**EnQueue ( &Q，e )**

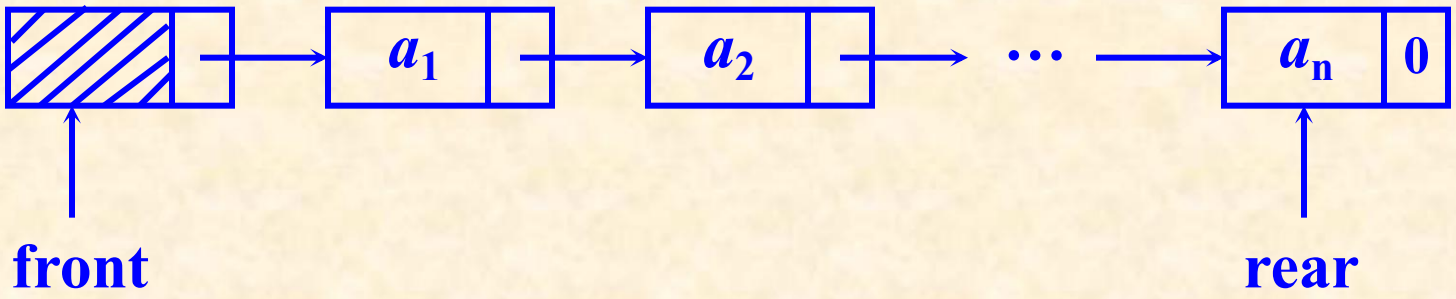
**DeQueue ( &Q，&e )**

## 3.4.2 队列的表示和实现

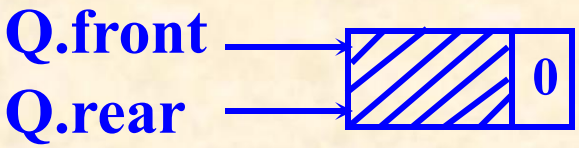
链队列；顺序队列>循环队列

### 1. 链队列—链式存储结构

必须具备指示队头和队尾的指针(头指针、尾指针)。




空队列

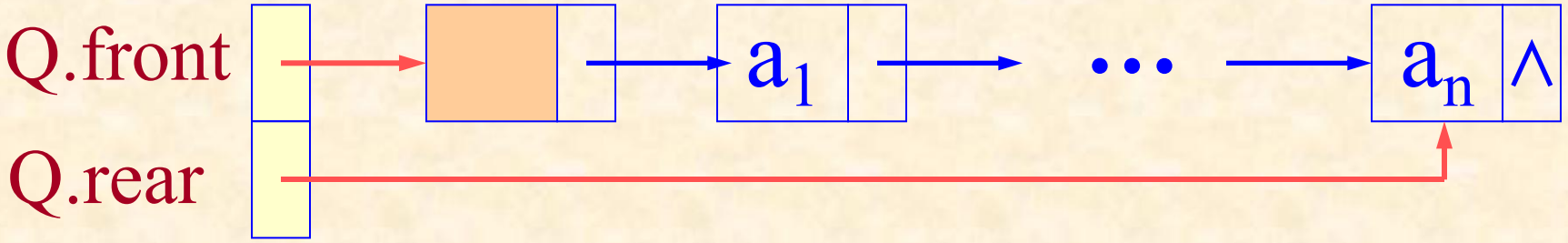


# 链队列——链式映象

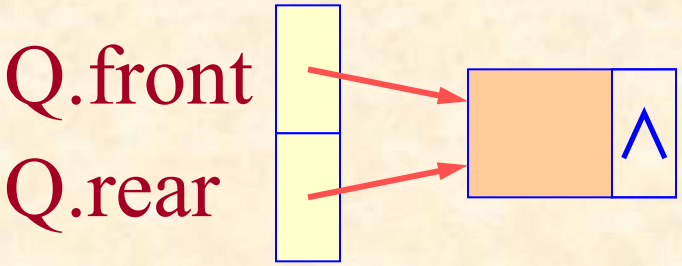
```
typedef struct QNode { // 结点类型
  QElemType      data;
  struct QNode  *next;
} QNode, *QueuePtr;
```

```
typedef struct {      // 链队列类型
    QueuePtr  front;  // 队头指针
    QueuePtr  rear;   // 队尾指针
} LinkQueue;
```

LinkQueue Q;

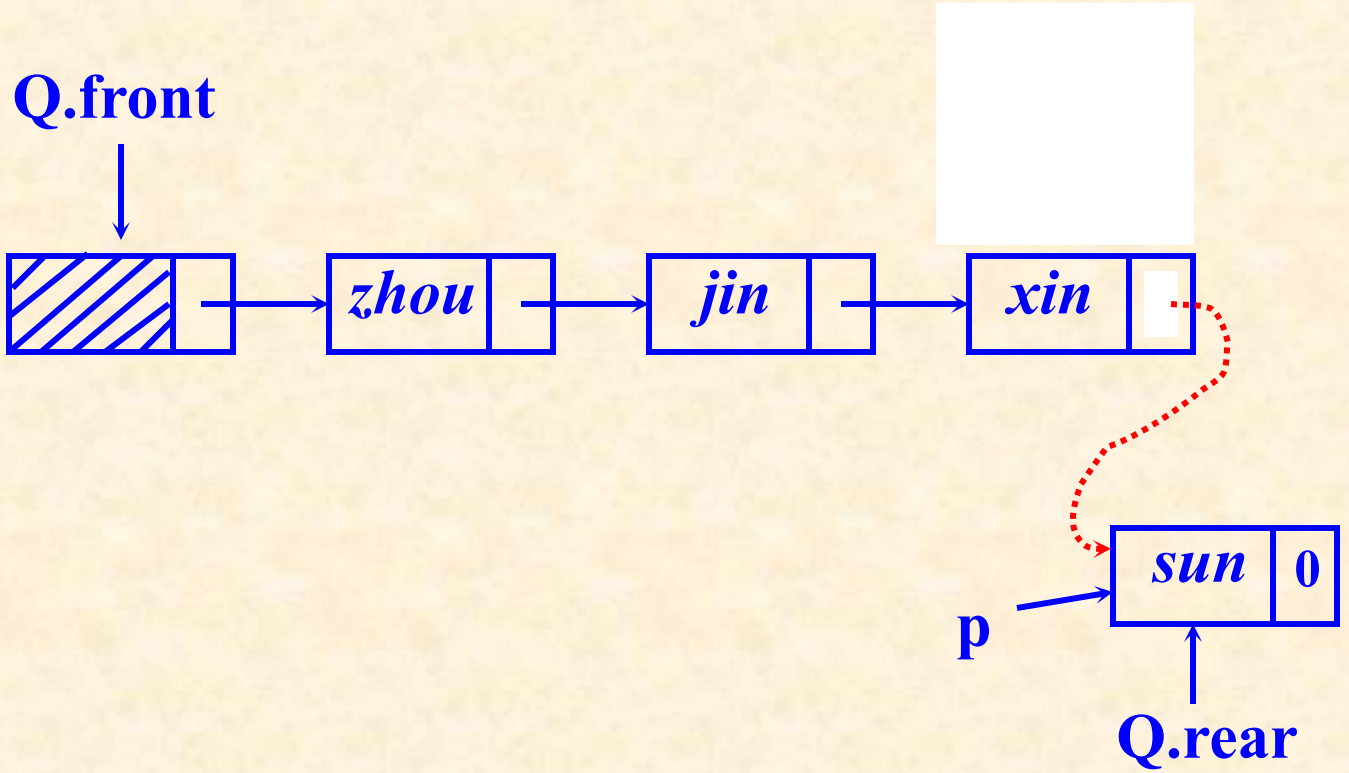



空队列



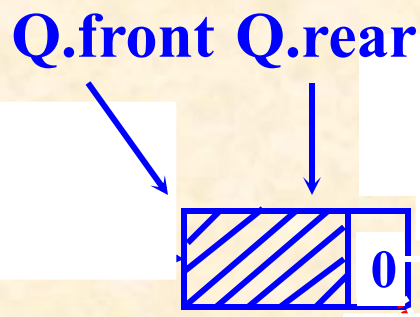
在队尾插入新元素 e

```
Status  EnQueue ( LinkQueue  &Q ,  QElemType  e ) {
    p = ( QueuePtr ) malloc ( sizeof(QNode) ) ;
    if ( ! p )   exit (OVERFLOW) ;
    p->data = e ;  p->next = NULL ;   //申请新结点
    Q.rear->next = p ;
    Q.rear = p ;                                  //插入在队尾
    return  OK ;
}
```


Q.front
zhou
jin
xin
sun
0
p
Q.rear

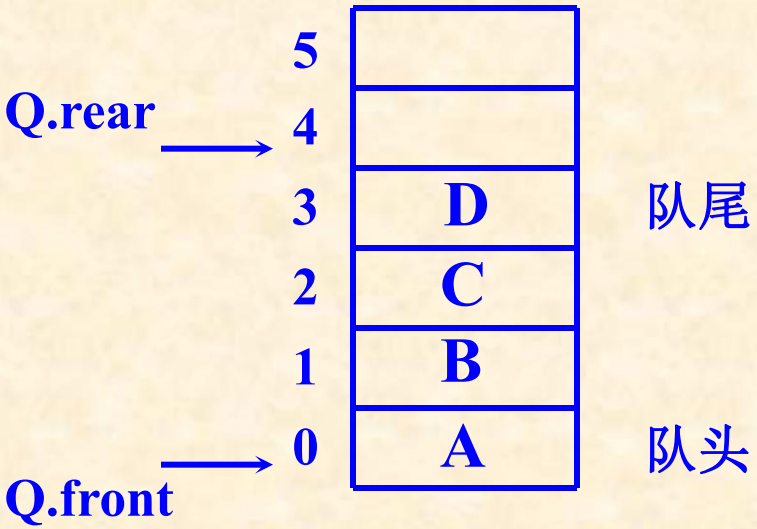
## 删除队头元素

```
Status DeQueue ( LinkQueue &Q , QElemType &e ) {
    if ( Q.front == Q.rear ) return ERROR ;
    p = Q.front->next ;
    e = p->data ;                          //取第一个结点
    Q.front->next = p->next ;   //删除第一个结点
    if ( Q.rear == p ) Q.rear = Q.front ;
                          //若需要删除的队头结点就是尾结点
    free ( p ) ;
    return OK ;
}
```




**e = p −> data = *xin***

## 2. 顺序队列—顺序存储结构



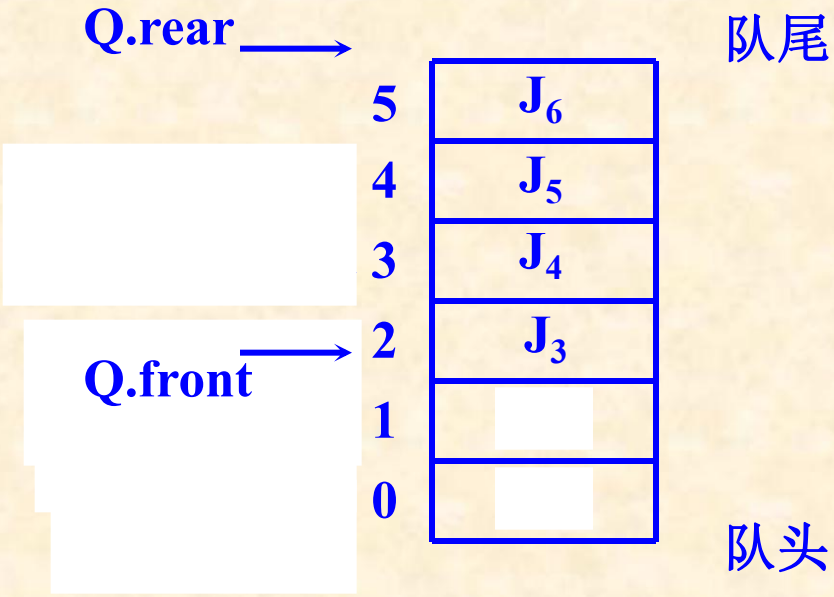

用一组地址连续的存储单元依次存放队头到队尾的元素。

指针 front 、rear 分别指示队头元素和队尾下一个元素。

令 front = rear = 0 表示空队列，rear=MAXSIZE 表示队满。

每插入一新元素，rear 增 1，每删除一元素，front 增 1。

# 插入、删除操作




插入元素 $J_1$；

插入元素 $J_2$、$J_3$；

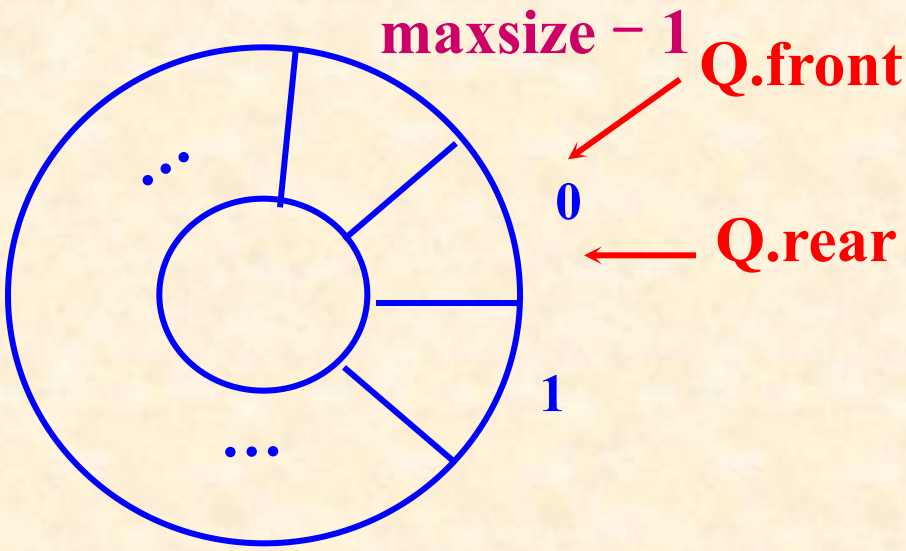
删除元素 $J_1$、$J_2$；

插入元素 $J_4$、$J_5$、$J_6$；

此时，队满，无法再插入新的元素，但实际队列中的可用空间并未真的被占满。

## 3. 循环队列—顺序存储结构

将顺序队列改造为一个环状的空间。




指针 front 、rear 分别指示队头元素和队尾下一个元素。

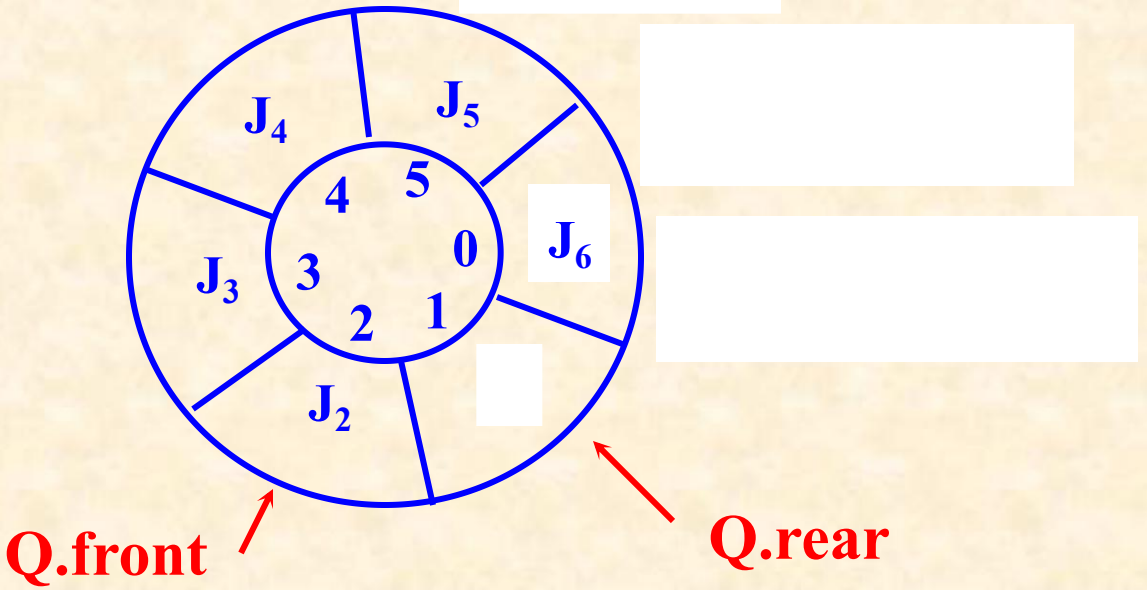
令 front = rear = 0 表示空队列。

每插入一新元素，rear = (rear + 1) % maxsize，

每删除一元素，front = (front + 1) % maxsize 。　// %：求余

# 插入、删除操作

maxsize = 6




初始，Q.front = Q.rear = 0，空队列。

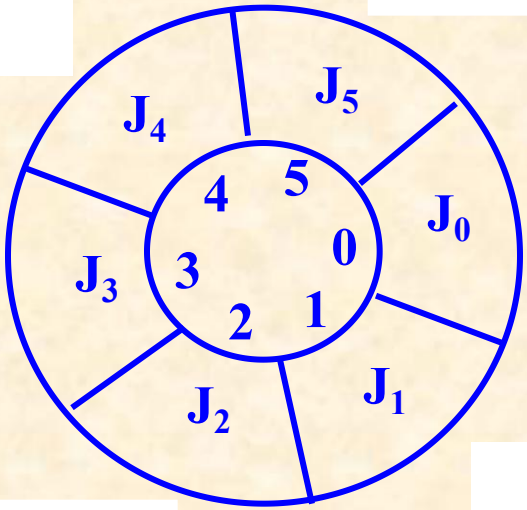
插入元素 $J_0$、$J_1$、$J_2$、$J_3$、$J_4$；

删除元素 $J_0$、$J_1$；

插入元素 $J_5$、$J_6$；

问题：

maxsize = 6




Q.front

Q.rear

初始，Q.front = Q.rear = 0 ，空队列。

插入元素 $J_0$、$J_1$、$J_2$、$J_3$、$J_4$、$J_5$；

Q.front = Q.rear = 0 ，满队列。

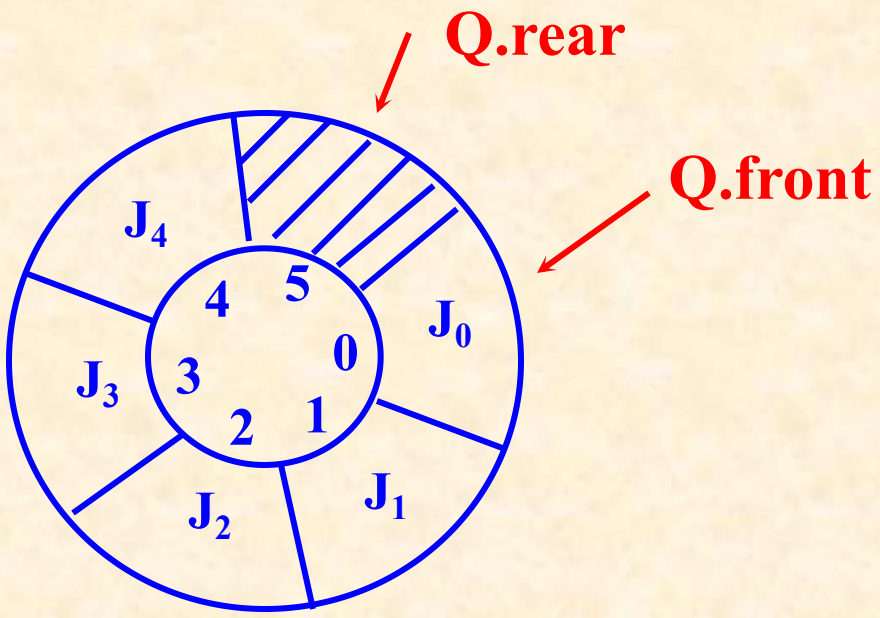
故无法通过 front = rear 来分辨队空或队满。

解决方案:

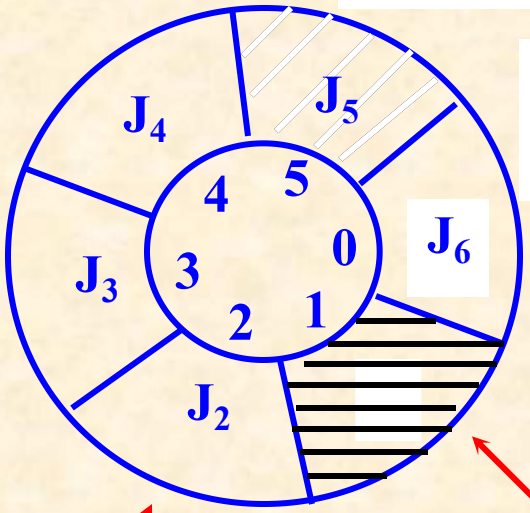
特殊空间，规定 front 与rear 之间总空出一个空间。

队空: Q.front == Q.rear

队满: Q.front == (Q.rear + 1) % maxsize



maxsize = 6




Q.front

Q.rear

初始，Q.front = Q.rear = 0，空队列。

插入元素 $J_0$、$J_1$、$J_2$、$J_3$、$J_4$；

Q.front == (Q.rear + 1) % maxsize 队满

删除元素 $J_0$、$J_1$；

插入元素 $J_5$、$J_6$；

Q.front == (Q.rear + 1) % maxsize 队满

- 队列通常采用两种存储结构是(　　)。

A．顺序存储结构和链表存储结构

B．散列方式和索引方式

C．链表存储结构和数组

D．线性存储结构和非线性存储结构

- 若让元素1，2，3依次进栈，则出栈次序不可能出现（　）种情况。

A．3，2，1　　B．2，1，3

C．3，1，2　　D．1，3，2

- 栈和队列都是（　　）。

A．链式存储的线性结构

B．顺序存储的线性结构

C．限制存取位置的线性结构

D．限制存取位置的非线性结构

- 对于给定的结点序列abcdef，规定进栈只能从序列的左端开始。通过栈的操作，能得到的序列为（　）。

A．abcfed　　B．cabfed

C．abcfde　　D．cbafde

- 栈的插入与删除操作在（　）进行。

A．栈顶　　B．栈底

C．任意位置　　D．指定位置

- 设一个栈的输入序列为A、B、C、D，则借助一个栈所能得到的输出序列不可能是（　）。

A．ABCD　　B．DCBA

C．ACDB　　D．DABC

- 一个队列的入队序列是a、b、c、d，则队列的输出序列为_____。
- 栈结构通常采用的两种存储结构是_____和_____。
- 中缀算术表达式5+6/(23-(6+15))*8 所对应的后缀算术表达式为_____。
- 中缀表达式3+x*(2.4/5-6)所对应的后缀表达式为_____。

- 队列通常采用两种存储结构是(　　)。

  A．顺序存储结构和链表存储结构 B．散列方式和索引方式

  C．链表存储结构和数组　　D．线性存储结构和非线性存储结构

- 设一个栈的输入序列为A、B、C、D，则借助一个栈所能得到的输出序列不可能是（　）。

  A．ABCD　　B．DCBA　　C．ACDB

  D．DABC

中缀表达式6+y*(2.5/5-4)所对应的后缀表达式为_____。