

# 第三章 栈和队列



## 第三章 栈和队列

本章介绍在程序设计中常用的两种数据结构

——栈和队列

## 第三章 栈和队列

- 3.1 栈
- 3.2 栈的应用举例
- 3.3 栈与递归
- 3.4 队列

## 3.1 栈

3.1.1 栈的概念

3.1.2 栈的顺序存储和实现

3.1.3 栈的链式存储和实现

## 3.1 栈

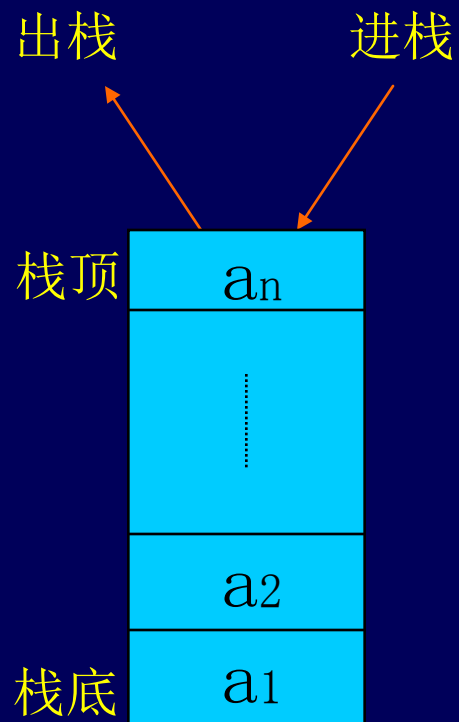
### 3.1.1 栈的概念

#### 一 什么是栈

栈是限定仅能在表尾一端进行插入、删除操作的线性表

$(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$   
删除 ↓ ↑ 插入

### 3.1 栈



栈的特点  
后进先出

第一个进栈的元素在栈底，  
最后一个进栈的元素在栈顶，  
第一个出栈的元素为栈顶元素，  
最后一个出栈的元素为栈底元素

栈的示意图

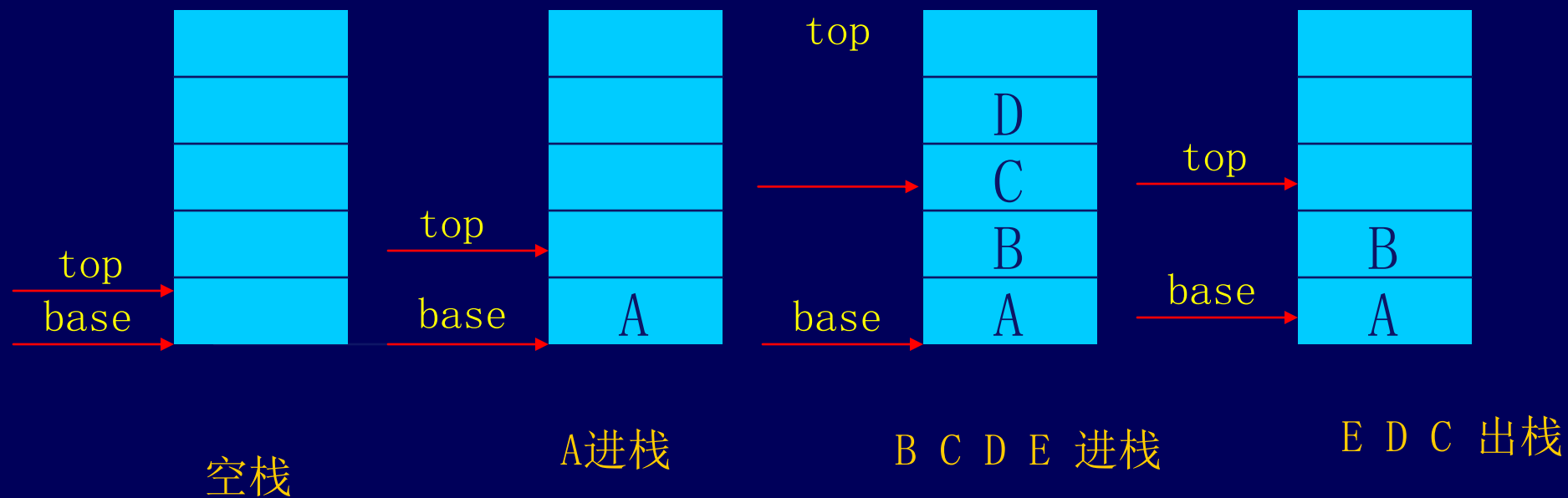


## 3.1 栈

### 二 栈的基本操作

- 1) 构造一个空栈S;
- 2) 进栈操作Push
- 3) 出栈操作Pop
- 4) 取栈顶元素top

## 3.1 栈



栈操作图示

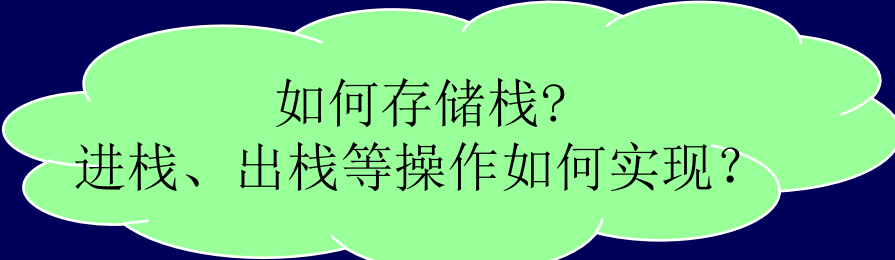


## 3.1 栈

### 3.1.2 栈的顺序存储和实现

#### 一、栈的顺序存储结构

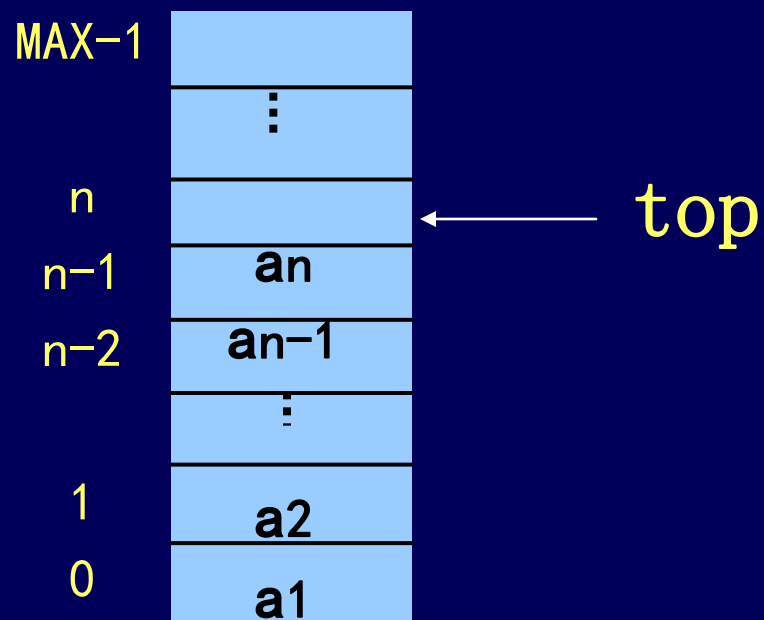
#### 1 栈的顺序存储结构



如何存储栈?  
进栈、出栈等操作如何实现?

```
#define MAX 100  
int stack[MAX];  
int top=0;
```

## 3.1 栈



顺序栈的图示

约定栈顶指针指向

栈顶元素的下一个位置

当栈用顺序结构存储时，  
栈的基本操作如建空栈、  
进栈、出栈等操作如何实现？



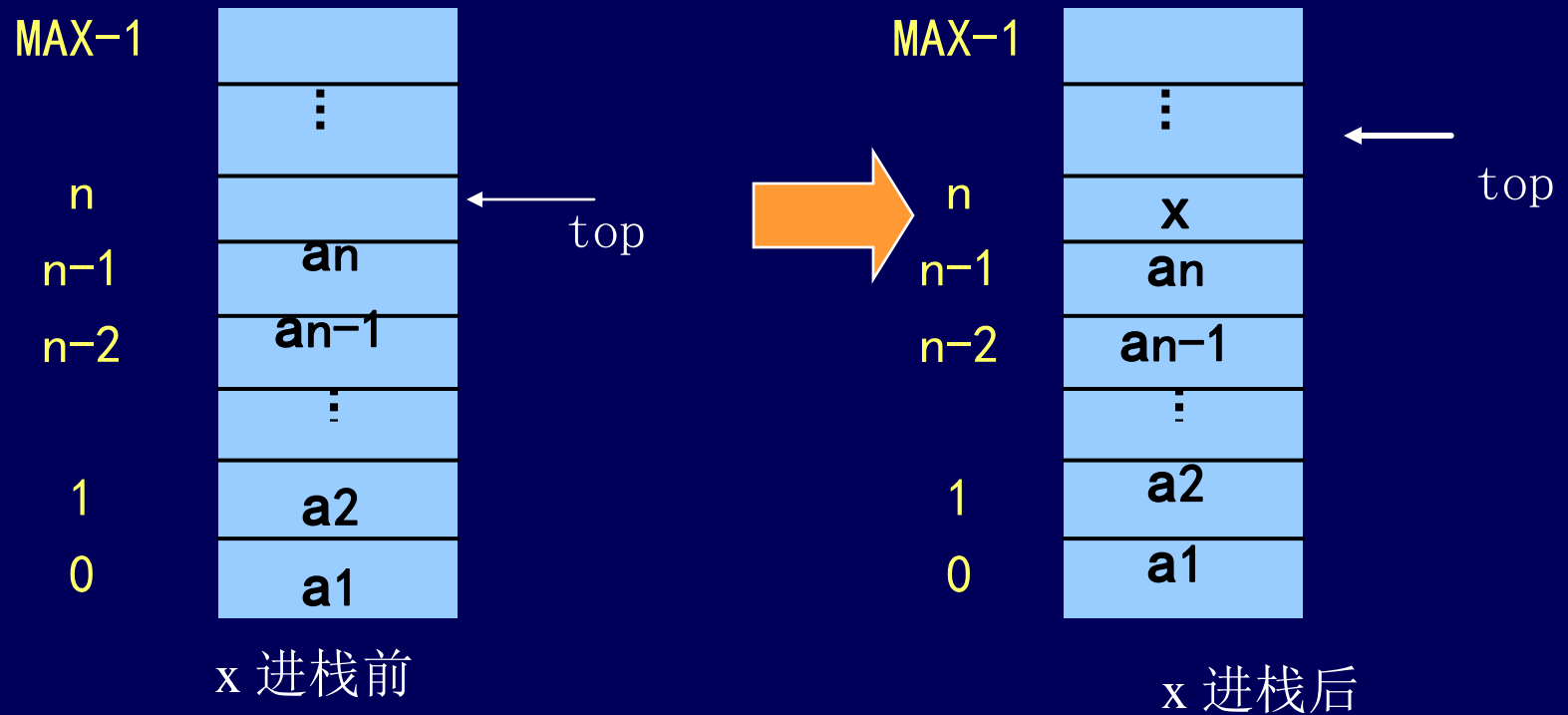
## 1) 进栈操作

```
viod Push( int x )
```

```
{if (top>=MAX) printf(“ overflow” );
```

```
else {stack[top]=x; top++; }
```

```
}
```

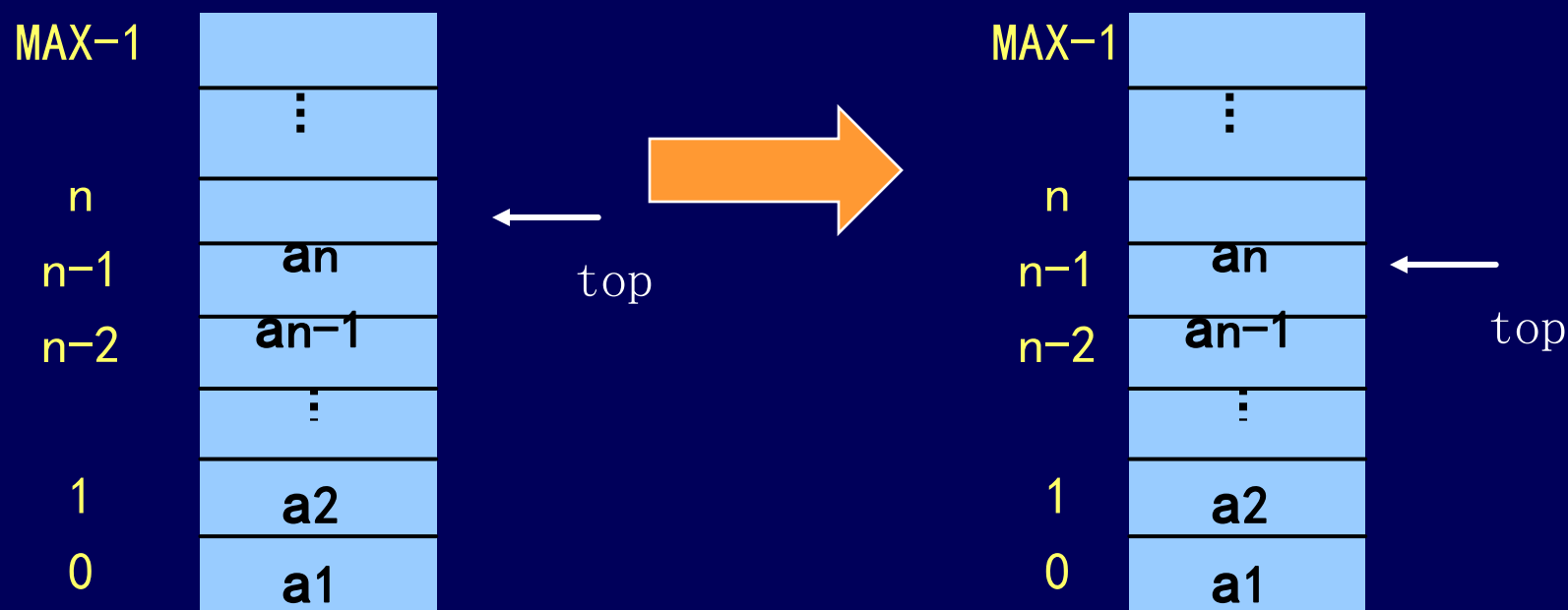


## 2) 出栈操作

```

int pop( )
{if (top==0) {printf("underflow"); return(NULL);}
top--; return(stack[top]);
}

```

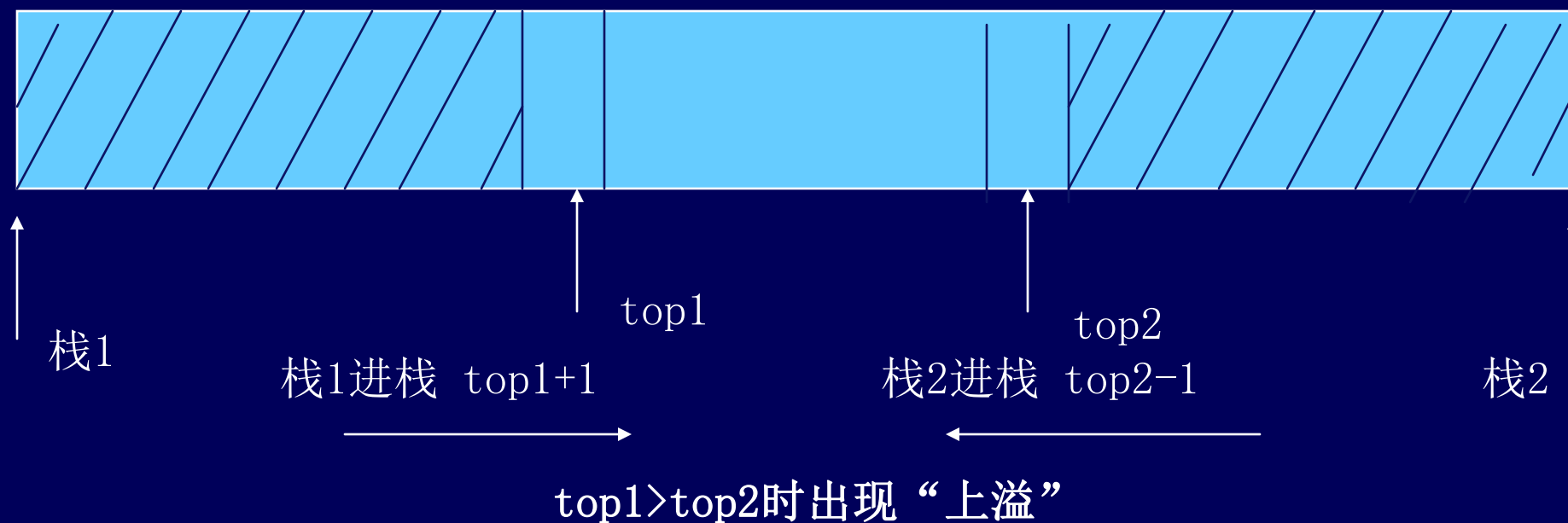


出栈操作前

出栈操作后

# 共享栈

- `#define MAX 100`
- `Int stack[MAX],top1=0,top2=MAX-1;`

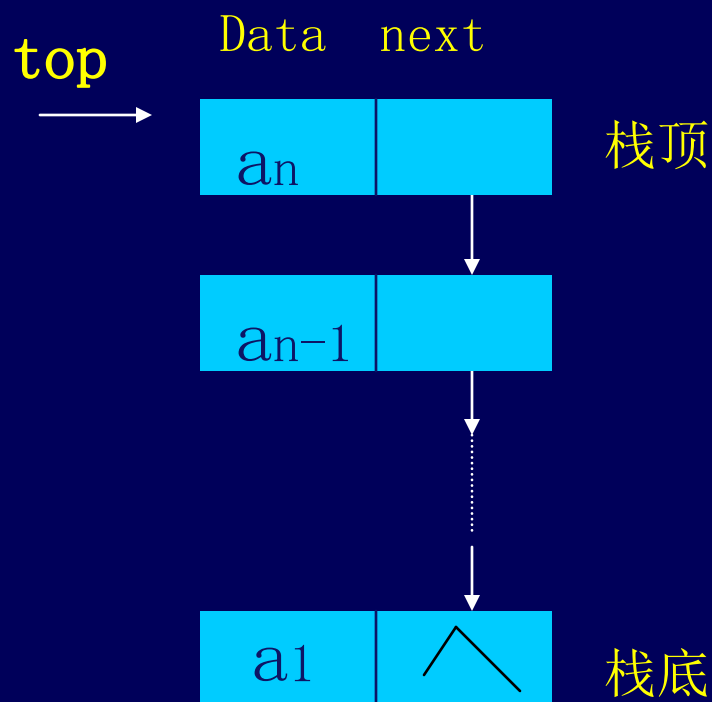


## 3.1 栈

### 2 栈的链式存储和实现

栈的链式存储结构，也称链栈，如图所示：

在前面学习了线性链表的插入删除操作算法，不难写出链式栈初始化、进栈、出栈等操作的算法



### (1) 进栈算法

```
NODE *top;
```

```
Void push(int x)
```

```
{ NODE *p=(NODE *)malloc(sizeof(NODE);
```

```
    p->data=x;
```

```
    p->link=top;
```

```
    top=p;
```

```
}
```

### (2) 出栈算法

```
NODE *pop( );  
{ NODE *p;  
  if (top==NULL)return(NULL);  
  p=top;  
  top=p->link;  
  return(p);  
}
```



### 小 结

- 1 栈是限定仅能在表尾一端进行插入、删除操作的线性表；
- 2 栈的元素具有后进先出的特点；
- 3 栈顶元素的位置由一个称为栈顶指针的变量指示，  
进栈、出栈操作要修改栈顶指针；

## 3.2 栈的应用举例

### 例1 数制转换

对于输入的任意一个非负十进制数，显示输出与其等值的八进制数

数制转换方法

$$N = (N \text{div} 8)_{10} \times 8 + N \bmod 8$$

N: 十进制数, div: 整除运算, mod: 求余运算;

$$(1348)_{10} = 2 \times 8^3 + 5 \times 8^2 + 0 \times 8 + 4 \times 8^0 = (2504)_8$$

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

计算时从低位到高位  
顺序产生八进制数的  
各个数位

显示时按从高位到低位的  
顺序输出


结果:     2   5   0   4

## void conversion( )

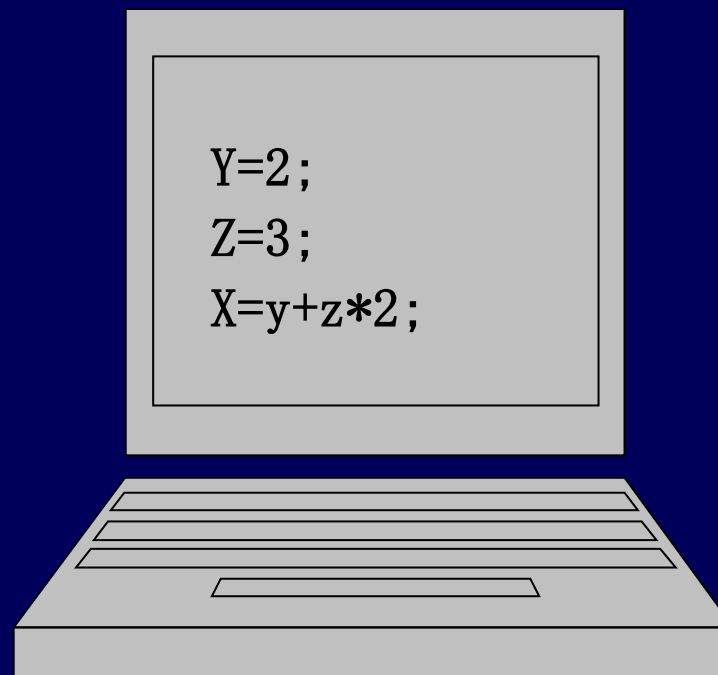
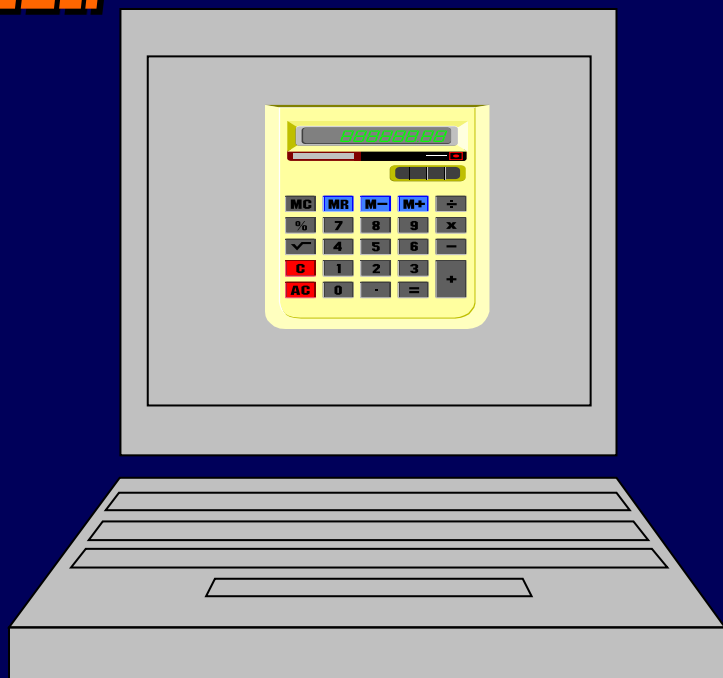
## 3.2 栈的应用举例

```
{    InitStack(s);           //建空栈
    scanf("%d",&x);         //输入一个非负十进制整数
    while(x!=0) {           // x不等于零循环
        push(s, x% 8);      // x/8第一个余数进栈
        x=x/8;              //整除运算
    }
    while(! StackEmpty(s) ) //输出存放在栈中的八制数位
    {    x=pop(s);
        printf("%d",x);
    }
}
```

算法3.1

- 
- `While##ilr#e(s#*s)      // while(*s)`
  - `outcha@putchar(*s=#++)    //putchar(*s++)`

## 3.2 栈的应用举例



### 例2 表达式求值

#### 1) 问题的提出

设计一个小计算器：对键入的表达式进行求值。

高级语言中的赋值语句：变量=表达式；

如何对表达式求值呢？



## 3.2 栈的应用举例

2) 表达式的构成 操作数+运算符+界符（如括号）

3) 表达式的求值:

例  $5+6\times(1+2)-4$

按照四则运算法则，上述表达式的计算过程为：

$$5+6\times(1+2)-4=5+6\times 3-4=5+18-4=23-4=19$$

3 2 1 4

如何确定运算符的运算顺序？

#### 4) 算符优先关系表

表达式中任何相邻运算符 $c_1$ 、 $c_2$ 的优先关系有：

$c_1 < c_2$ :  $c_1$ 的优先级低于 $c_2$

$c_1 = c_2$ :  $c_1$ 的优先级等于 $c_2$

$c_1 > c_2$ :  $c_1$ 的优先级高于 $c_2$

$C_1 \backslash C_2$	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

算符间的优先关系表

注:  $c_1$   $c_2$ 是相邻算符,  $c_1$ 在左,  $c_2$ 在右

## 算符的优先级设置

算符	栈内优先级	栈外优先级
* /	4	3
+ -	2	1
(	0	5
)	5	0
#	-1	-1



## 3.2 栈的应用举例

### 5 算符优先算法

$$5 + 6 \times (1 + 2) - 4$$



从左向右扫描表达式:

遇操作数——保存;

遇运算符 $c_j$ ——与前面的刚扫描过的运算符 $c_i$ 比较

若 $c_i < c_j$  则保存 $c_j$ , ( 因此已保存的运算符的优先关系为  $c_1 < c_2 < c_3 < c_4$ 。 )

若 $c_i > c_j$  则说明 $c_i$ 是已扫描的运算符中优先级最高者, 可进行运算;

若 $c_i = c_j$  则 $c_i$ 为 (,  $c_j$  为 ), 说明括号内的式子已计算完, 需要消去括号;

用两个栈分别保存扫描过程中遇到的操作数和运算符

后面也许有优先级更高的算符



后保存的算符有  
优先级高

## 算符比较算法

```
Char Precede( char c1, char c2)
{int c_temp1, c_temp2;
  switch(c1)
  {case ' * ' :
    case ' / ' : c_temp1=4; break;
    case ' + ' :
    case ' - ' : c_temp1=2; break;
    . . . . . }
  switch(c2)
  {case ' * ' :
    case ' / ' : c_temp2=3; break;
    case ' + ' :
    case ' - ' : c_temp2=1; break;
    . . . . . }
```

续

```
if (c_temp1<c_temp2) return( ' <' );  
if (c_temp1=c_temp2) return( ' =' );  
if (c_temp1>c_temp2) return( ' >' );  
}
```

## 3.2 栈的应用举例

在算符优先算法中，建立了两个工作栈。一个是OPTR栈，用以保存运算符一个是OPND栈，用以保存操作数或运算结果。

```
int express ( )
```

```
{//运算数栈， OP为运算符集合。
```

```
    InitStack(OPTR);  Push (OPTR, '# ');
```

```
    InitStack(OPND);  w=getchar( );
```

```
    While(w!=' #' || GetTop(OPTR)!='#')
```

```
    {if(! In(w,OP)){Push(OPND,w);w=getchar();} //不是运算符则进栈
```

```
        else                                // In(w, OP)判断c是否 是运算符的函数
```

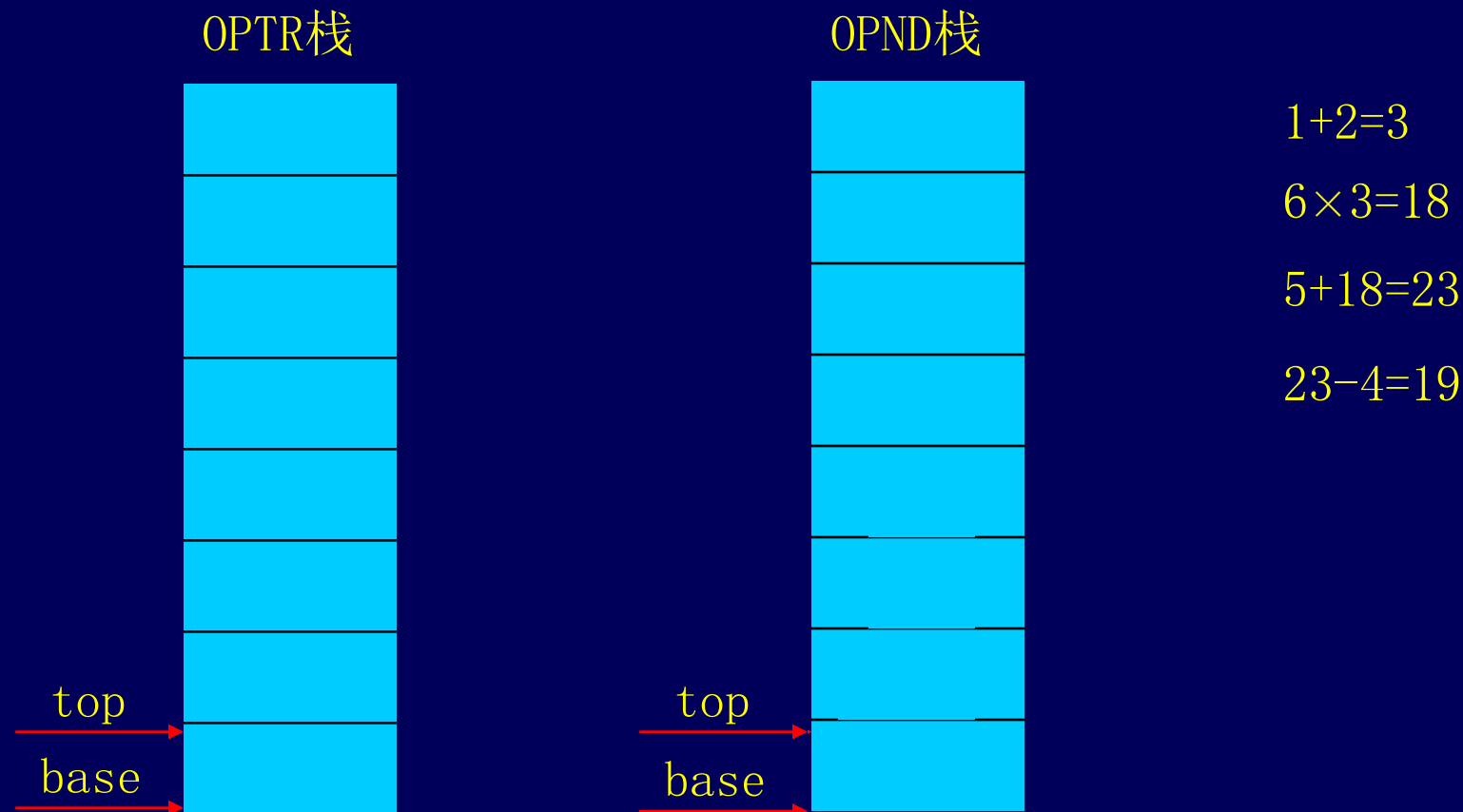
## 3.2 栈的应用举例

续

```
switch (Precede(GetTop(OPTR), w) {  
    case '<':    // 新输入的算符 w 优先级高, w 进栈  
        Push(OPTR, w ); w =getchar( ); break;  
    case '=':    // 去括号并接收下一字符  
        x=Pop(OPTR); w=getchar( ); break;  
    case '>':    //新输入的算符c优先级低, 即栈顶算符优先权高,  
                //出栈并将运算结果入栈OPND  
        op=Pop(OPTR);  
        b=Pop(OPND); a= Pop(OPND);  
        Push(OPND, Operate(a, op, b));  
        break;    }  
} return GetTop(OPND);  
}
```

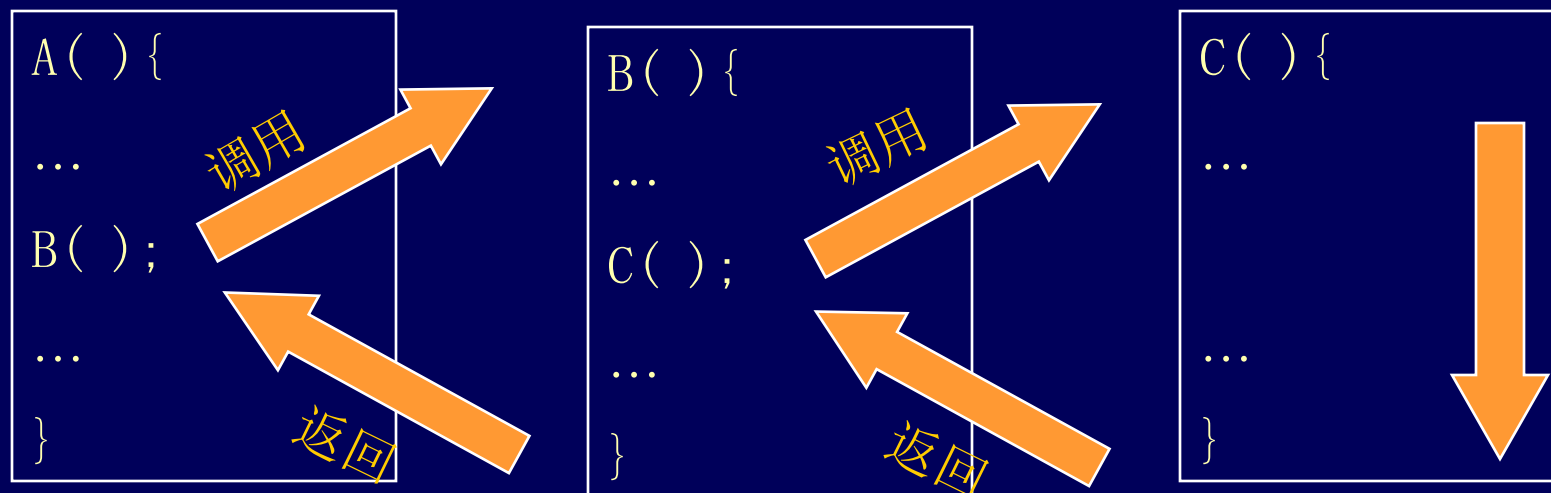
表达式求值示意图:  $5+6\times(1+2)-4=19$

读入表达式过程:  $5+6\times(1+2)-4\#$



### 3.3 栈与递归

#### 一般函数的调用机制



后调用的函数先返回

函数调用顺序 A → B → C

函数返回顺序 C → B → A

计算机正是利用栈来实现函数的调用和返回的

函数调用机制可通过栈来实现

## 3.3 栈与递归

### 1. 什么是递归

递归是一个很有用的工具，在数学和程序设计等许多领域中都用到递归。

递归定义：简单地说，一个用自己定义自己的概念，称为递归定义。

例  $n! = 1 * 2 * 3 * 4 * \dots * (n-1) * n$

$n!$ 递归定义  $n! = 1$  当  $n=0$  时

$n! = n (n-1)!$  当  $n > 0$  时

用  $(n-1)!$  定义  $n!$



### 3.3 栈与递归

2 递归函数：一个直接调用自己或通过一系列调用间接调用自己的函数称为递归函数。

```
A ( ) {  
    ...  
  
    A();  
  
    ...  
}
```

A 直接调用自己

<pre>B() {     ...      C();      ... }</pre>	<pre>C() {     ...      B();      ... }</pre>
---	---

B间接调用自己

### 3.3 栈与递归

#### 2. 递归算法的编写

- 1) 将问题用递归的方式描述（定义）
- 2) 根据问题的递归描述（定义）编写递归算法

问题的递归描述（定义）

递归定义包括两项

基本项（终止项）：描述递归终止时问题的求解；

递归项：将问题分解为与原问题性质相同，但规模较小的问题；

例  $n!$  的递归定义

基本项：  $n!=1$             当  $n=0$

递归项：  $n!=n (n-1)!$     当  $n>0$

用  $(n-1)!$  定义  $n!$

### 3.3 栈与递归

例1 编写求解 阶乘 $n!$  的递归算法

首先给出阶乘 $n!$  的递归定义

$n!$ 的递归定义

基本项:	$n!=1$	当 $n=1$
递归项:	$n!=n (n-1)!$	当 $n> 1$

```
int fact(int n)
{
    //算法功能是求解并返回n的阶乘
    if (n=1) return (1) ;
    else return (n*fact(n-1)) ;
}
```

### 3.3 栈与递归

我们看一下 $n=3$  阶乘函数 $\text{fact}(n)$ 的执行过程

```

Main( ) {
    int n=3, y;
    L y= fact(n);
}
    
```



$\text{fact}(n)$

```

int fact (int n) {
    If (n=1) return 1;
    Else
    L1 return (n*fact(n-1));
}
    
```



```

int fact (n) {
    If (n=1) return (1);
    Else
    L1 return (n*fact(n-1));
}
    
```

$n=3$



$n * \text{fact}(n-1)$

```

int fact (int n) {
    If (n=1) return (1);
    Else
    L1 return (n*fact(n-1));
}
    
```

$n=2$

$n * \text{fact}(n-1)$

返回地址 实参


注意递归调用中  
栈的变化

## 3.4 队列

3.4.1 队列的概念

3.4.2 循环队列

——队列的顺序存储和实现

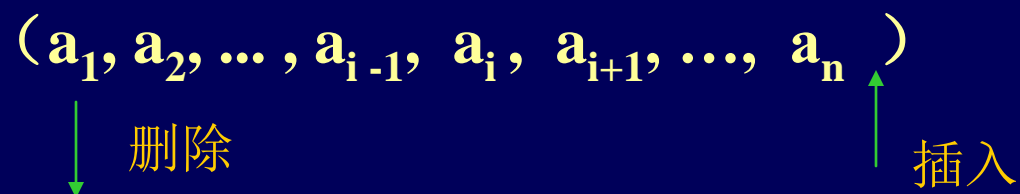
3.4.3 队列的链式存储和实现

## 3. 4 队列

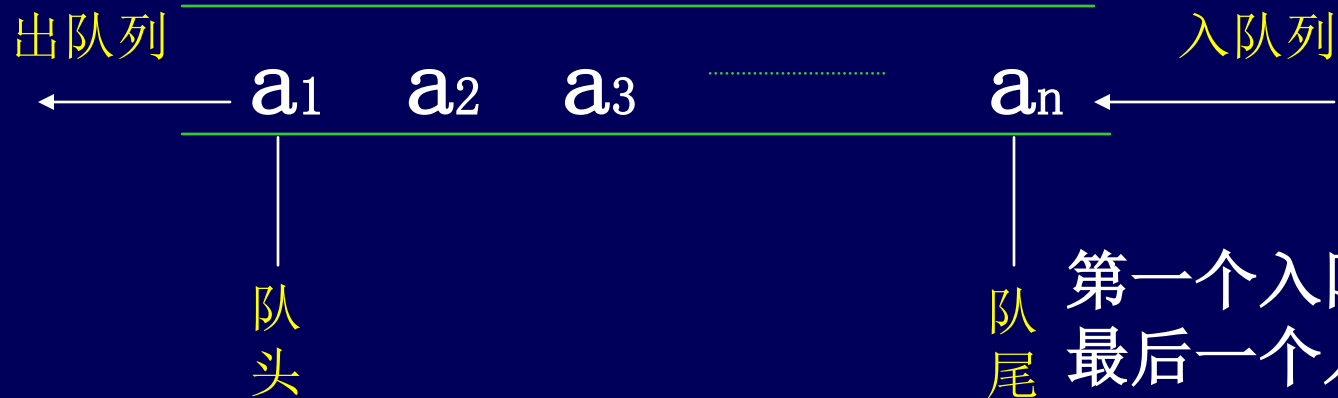
### 3.4.1 队列的概念

#### 一 什么是队列

队列是限定仅能在表头进行删除，表尾进行插入的线性表



### 3. 4 队列



队列的特点  
先进先出

队列的示意图

第一个入队的元素在队头，  
最后一个入队的元素在队尾，  
第一个出队的元素为队头元素，  
最后一个出队的元素为队尾元素



## 3. 4 队列

### 二 队列的基本操作

- 1) 初始化操作
- 2) 销毁操作
- 3) 置空操作
- 4) 判空操作
- 5) 取队头元素操作
- 6) 入队操作
- 7) 出队操作



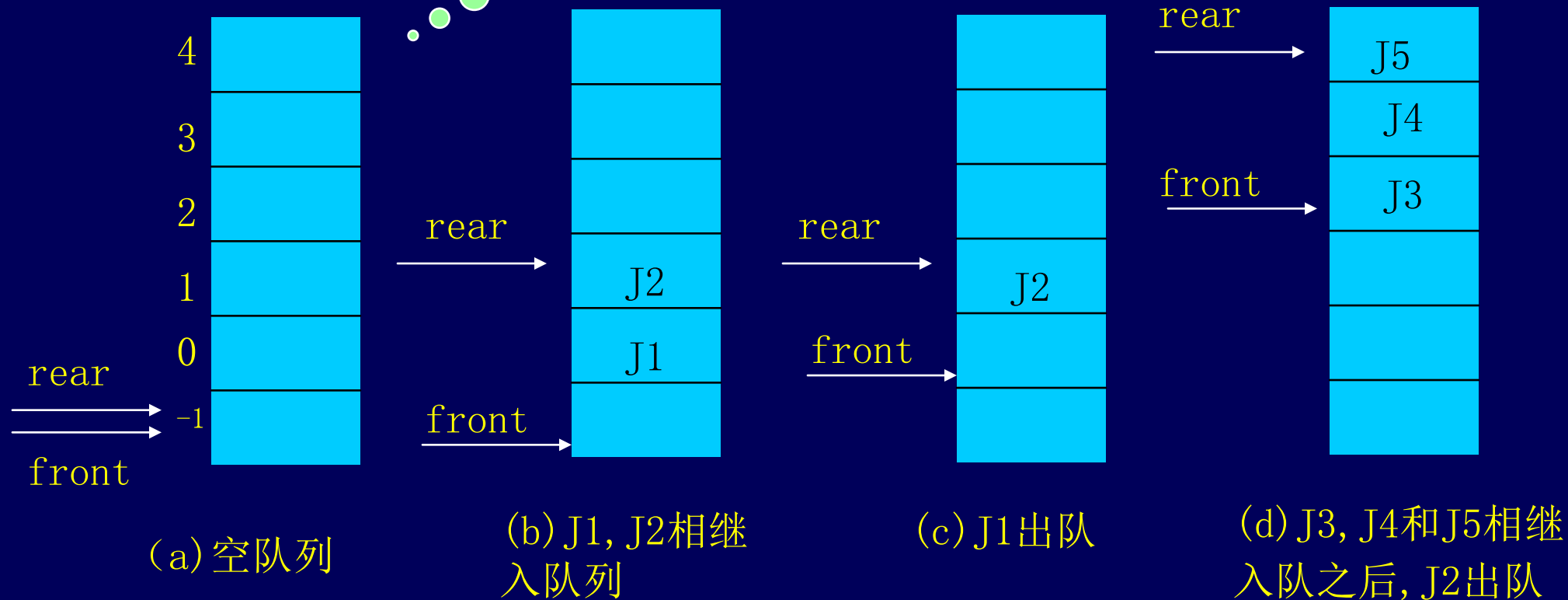
### 3. 4 队列

```
#define MAX 50 /*数据结构定义*/  
int queue[MAX]; int front=-1; rear=-1;  
  
int insert_queue(int x) /*入队列*/  
{if(rear==MAX-1)return(0);  
  rear++;  
  queue[rear]=x;return(1);  
}  
  
int del_queue() /*入队列*/  
{ if(rear==front) return(0);  
  front++;return(queue[front]);  
}
```

### 3.4 队列

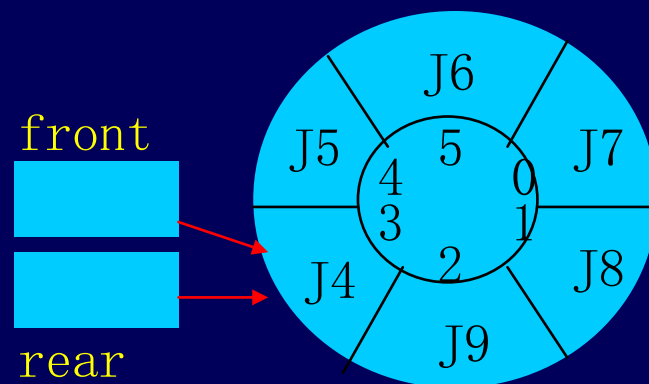
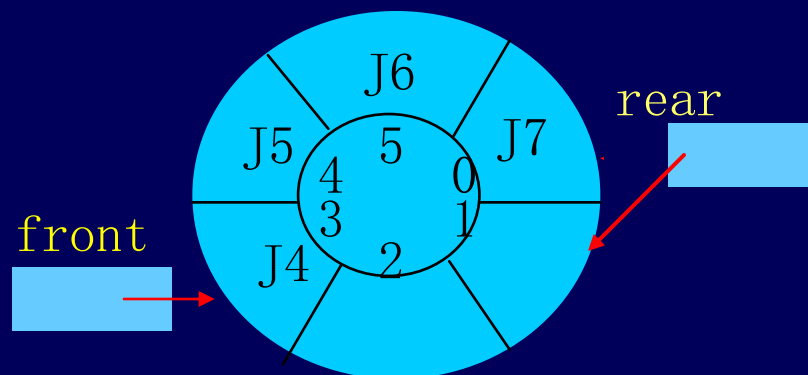
front, rear均为整数  
用箭头指示只是为了直观

又有J6入队, 怎么办?

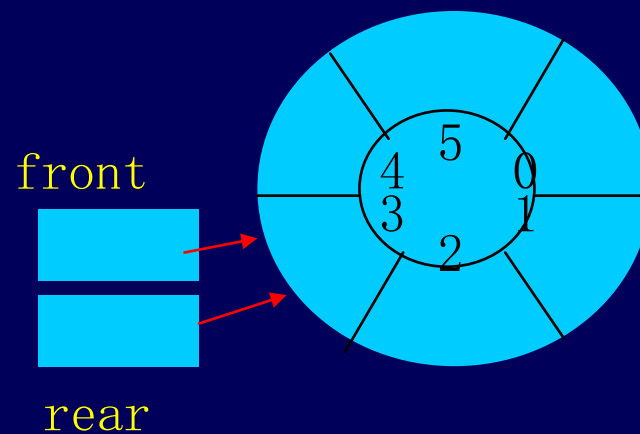


## 3.4 队列

### 3. 循环队列



(c) 队满



(b) 队空

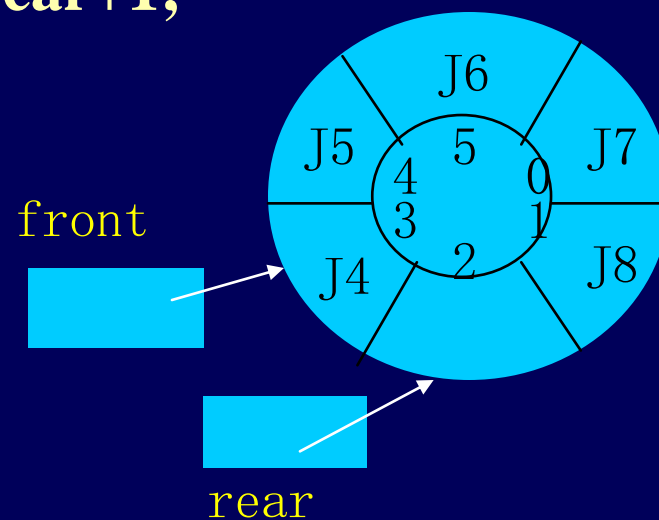
队空、队满  
都有  $\text{front} = \text{rear}$

如何判断循环队列  
队空、队满？

### 3.4 队列

有两种方法:

- 1) 另设一个标志位以区分队空、队满。
- 2) 少用一个存储单元, 队满条件:  $\text{front} = \text{rear} + 1$ ;



(d)

## 3.4 队列

### 二 循环队列的基本操作算法

1) 初始化操作 功能：建一个空队列Q；

算法：

```
Int queue[MAX];
```

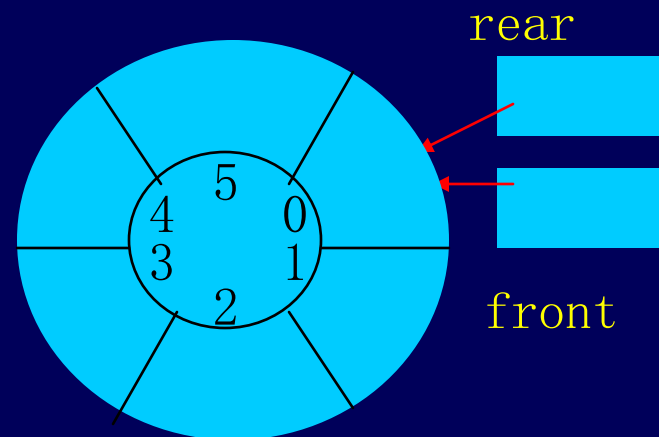
```
Int front, rear;
```

```
int InitQueue_Sq() {
```

```
    //构造一个空队列Q
```

```
    front = 0; rear=0;
```

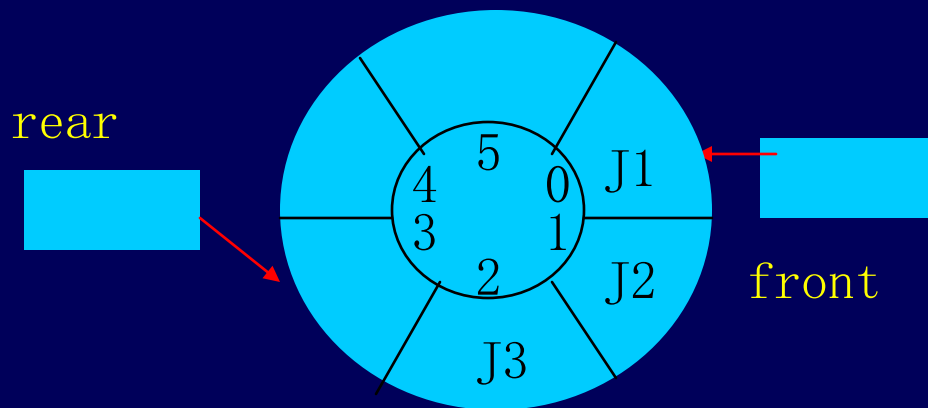
```
    Return (1)}
```



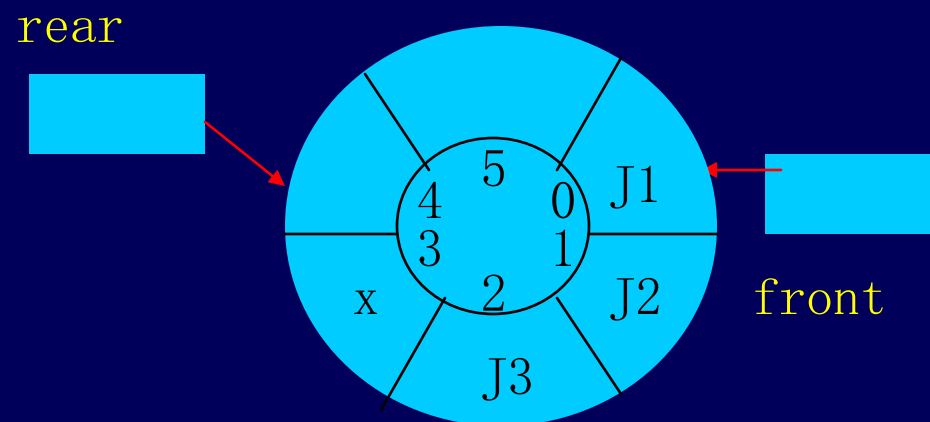
建一个空队列Q

### 3.4 队列

6) 入队操作 功能：将元素  $x$  插入队尾



元素  $x$  入队前



元素  $x$  入队后

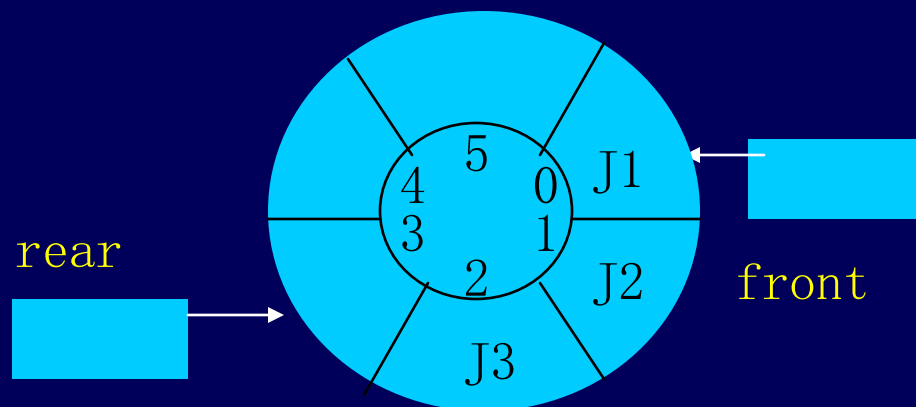
## 3.4.2 循环队列——队列的顺序存储和实现

```
int insert_queue(int x)    /*入队列*/  
{if((rear+1)%MAX==front)return(0);  
  rear=(rear+1)%MAX;  
  queue[rear]=x;return(1);  
}
```

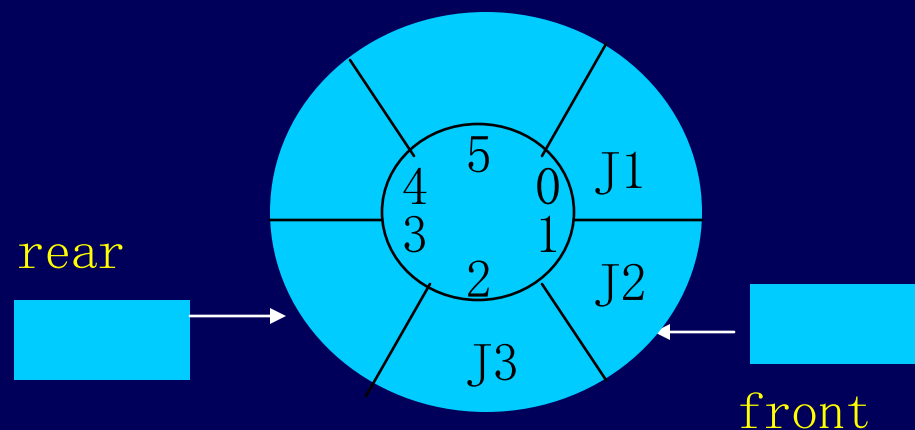
求余  
运算

### 3.4 队列

7) 出队操作 功能：删除队头元素；



出队操作前



出队操作后



### 3.4 队列

```
int del_queue()    /*出队列*/
{
    if(rear==front) return(0);
    front=(front+1)%MAX;
    return(queue[front]);
}
```

## 3.4 队列

### 3.4.3 链队列——队列的链式存储结构和实现

#### 一 链队列



链队列图示

### 3.4 队列

#### 二 链队列的类型定义

```
struct node //链队列结点的类型定义
{int data;
  struct node *link;
};typedef struct node NODE
NODE *front, *rear;
```

### 3.4 队列 入列运算

```
Void addqlink(int x)
{NODE *p;
  p=(NODE *)malloc(sizeof(NODE));
  p->data=x;p->link=NULL;
rear->link=p;rear=p;
}
```

### 3.4 队列 出列运算

```
NODE *deleqlink( )  
{ NODE *p;  
  if (front==rear) return(NULL);  
  p=front->link;  
  front->link=p->link;  
  if (front->link==NULL)rear=front;  
  return(p);  
}
```

### 3.4 队列 元素记数运算

```
int count( )  
{ int i; NODE *p;  
  if (front==rear)return(0);  
  for(p=front->link;i=1;p!=p->link)i++;  
  return(1);  
}
```

## 3.4 队列

### 三 队列的应用

- 1) 解决计算机主机与外设不匹配的问题;
  - 2) 解决由于多用户引起的资源竞争问题;
  - 3) 离散事件的模拟----模拟实际应用中的各种排队现象;
  - 4) 用于处理程序中具有先进先出特征的过程;
- } 在操作系统课程  
中会讲到

## 小 结

- 1 队列是限定仅能在表尾一端进行插入，表头一端删除操作的线性表；
- 2 队列中的元素具有先进先出的特点；
- 3 队头、队尾元素的位置分别由称为队头指针和队尾指针的变量指示，
- 4 入队操作要修改队尾指针，出队操作要修改队头指针；





# 第三章结束