

# 第三章 栈和队列

- ◆ 实例
- ◆ 栈
- ◆ 队列

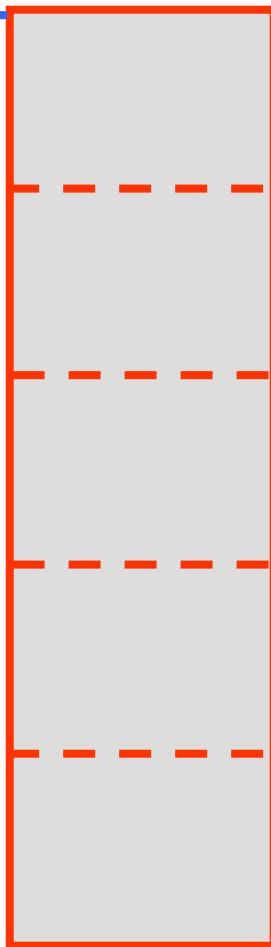
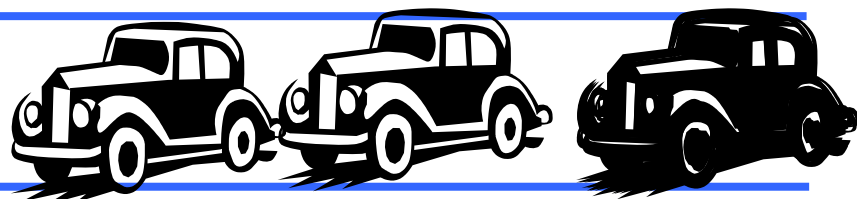


本章小结

课后作业

# 栈和队列实例

## 停车场管理

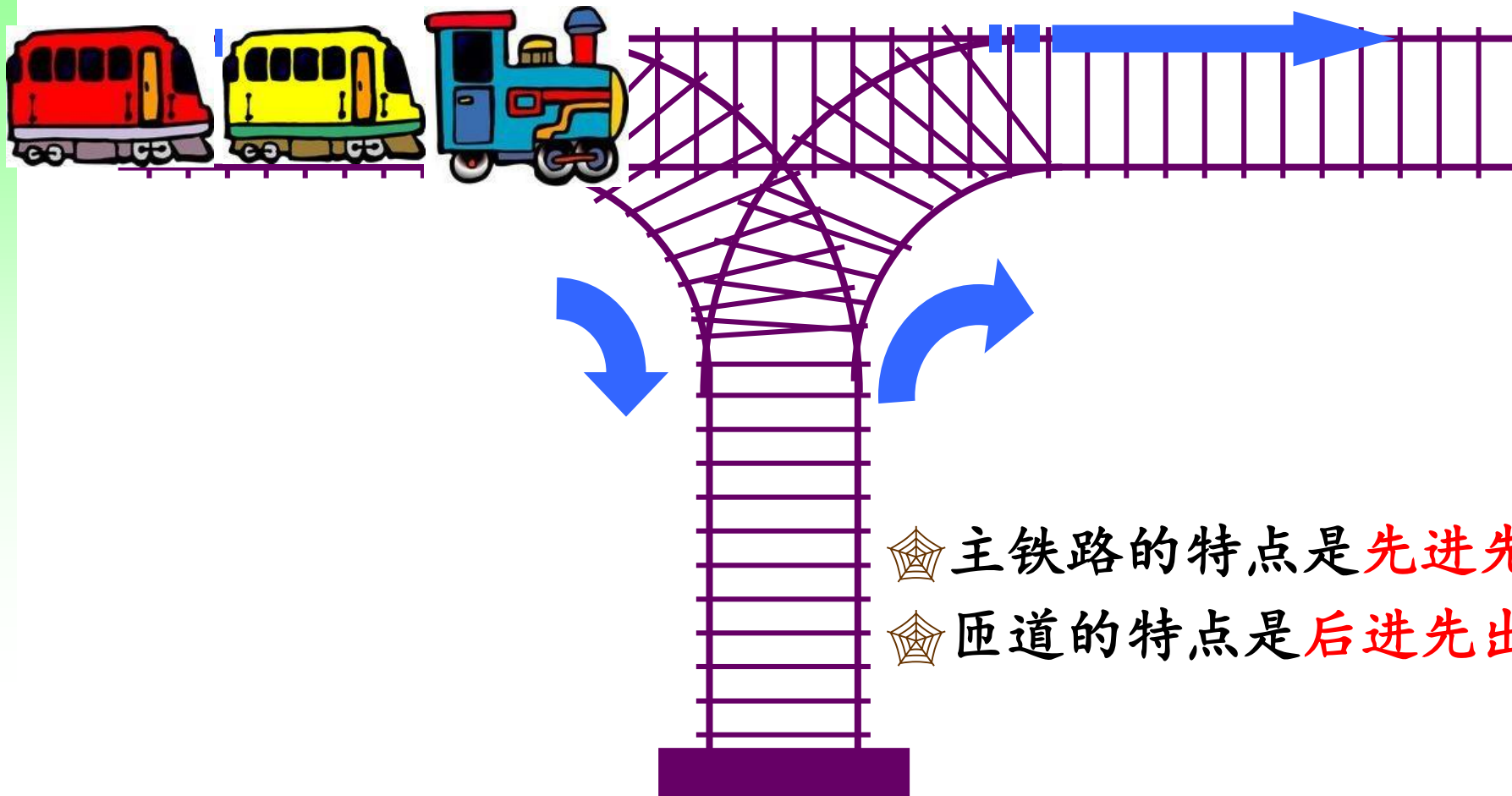


⚡ 停车场的特点是**后进先出**

⚡ 便道的特点是**先进先出**

# 栈和队列实例

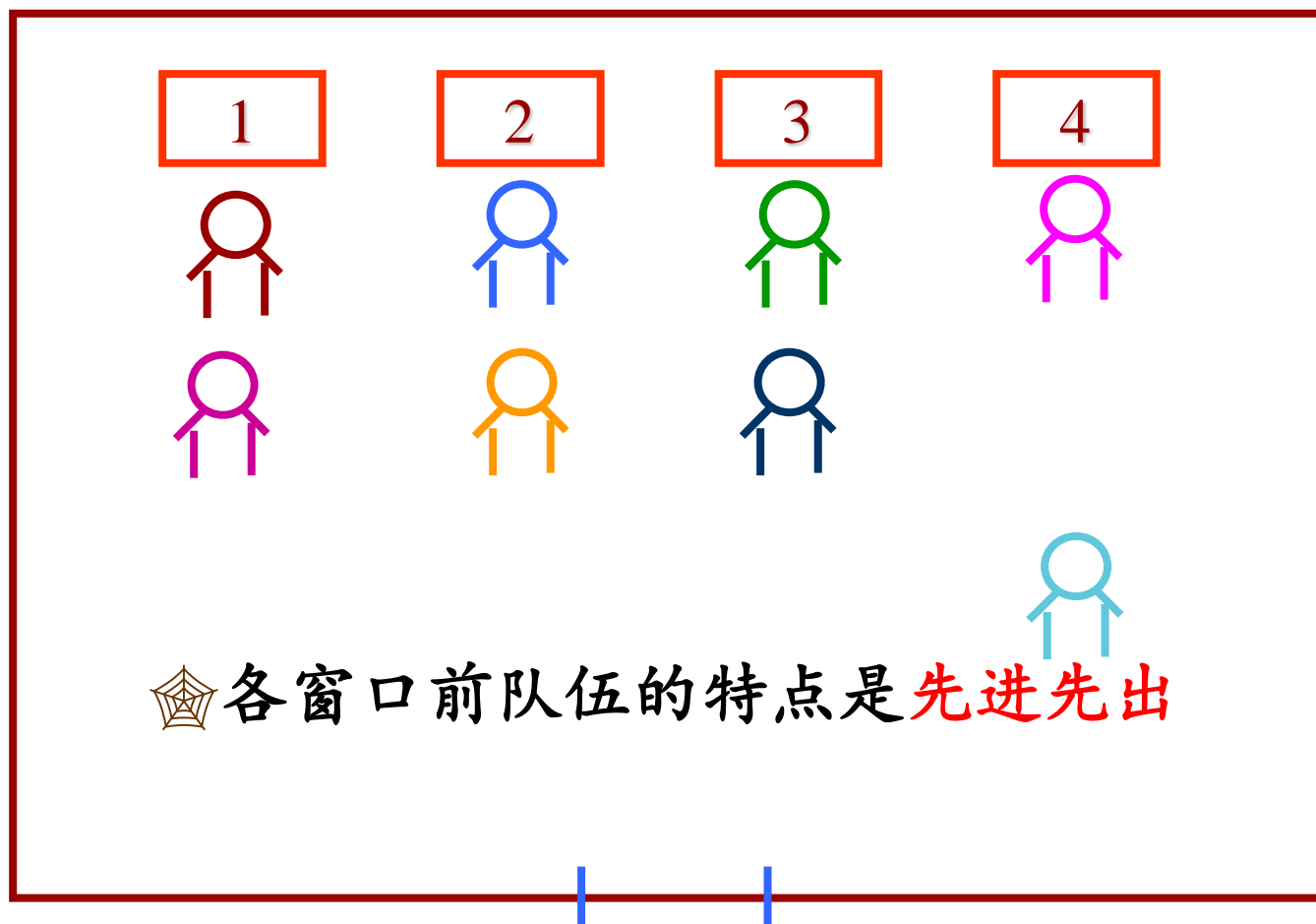
## Railway Switching Network



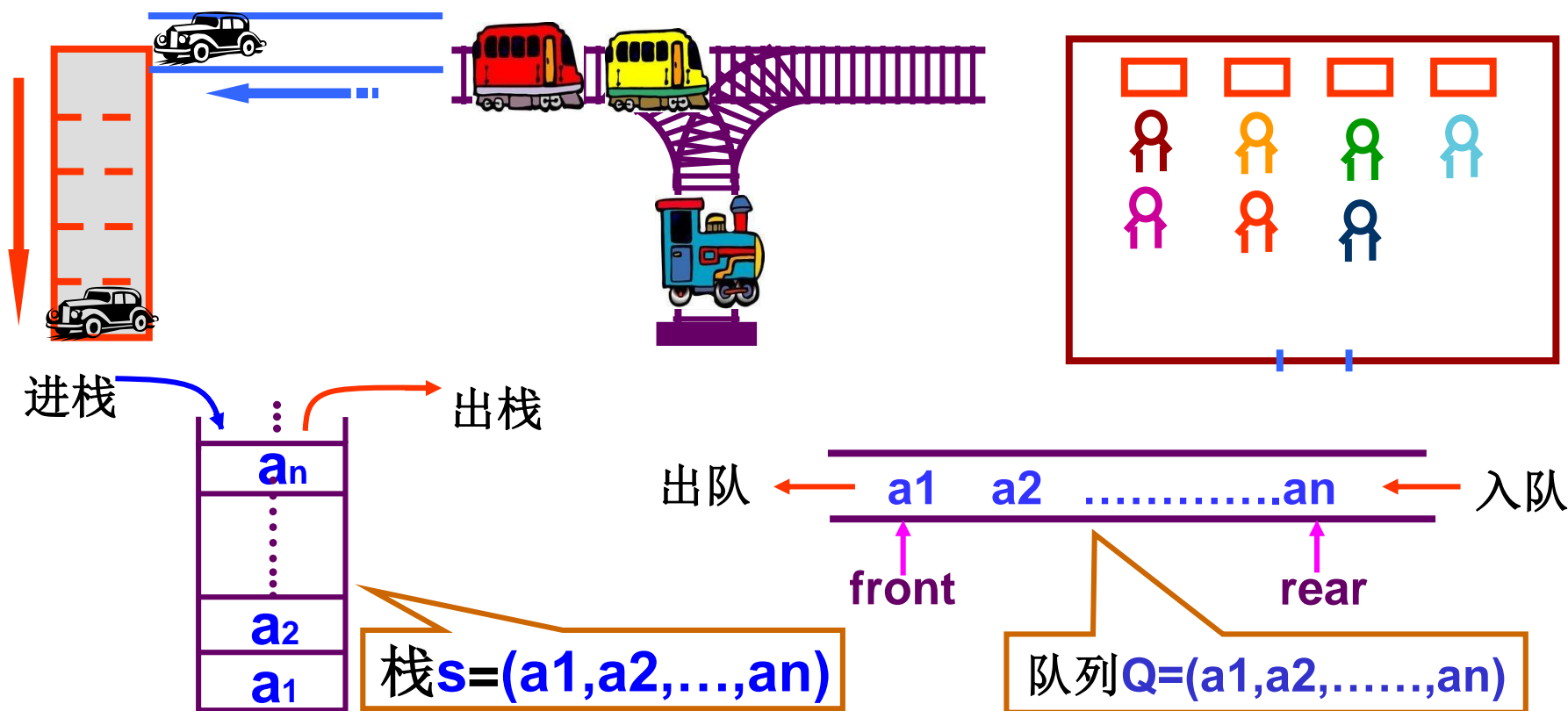
# 栈和队列实例



## 银行业务模拟



# 栈和队列实例



栈和队列是两种特殊的线性表

是操作受限的线性表，称限定性数据结构

限定插入和删除只能在表的“端点”进行的线性表

# 栈(Stack)

---

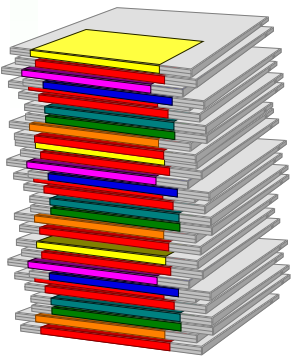
栈的逻辑结构



栈的存储结构



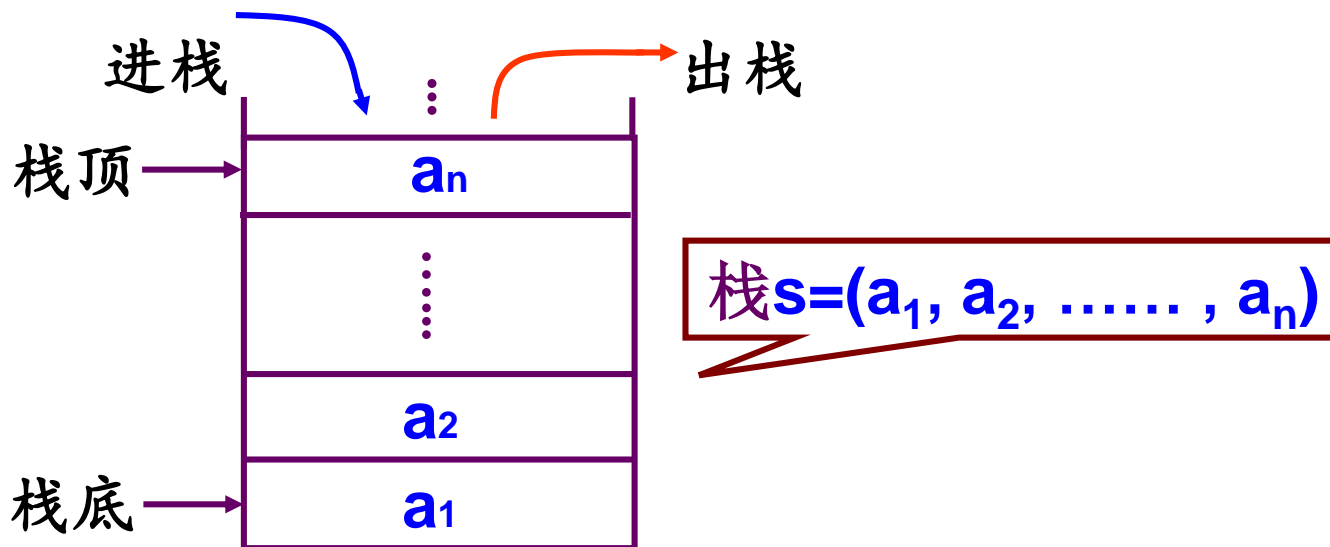
栈的应用



**Back**

# 栈的逻辑结构

🕸 栈——限定只能在表尾进行插入/删除的线性表  
表尾——栈顶，表头——栈底



🕸 特点：先进后出(FILO)或后进先出(LIFO)

# 抽象数据类型栈的定义

## ADT Stack

{ 数据对象:  $D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系:  $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

(约定 $a_n$ 端为栈顶,  $a_1$ 端为栈底)

基本操作:

InitStack( &S );

构造一个空栈S

DestroyStack( &S );

销毁栈S

ClearStack( &S );

将栈S清空

StackEmpty( S );

判断栈S是否为空

StackLength( S );

返回栈S中元素个数

GetTop( S, &e );

用e返回栈S的栈顶元素

Push( &S, e );

将元素e入栈到栈S中

Pop( &S, &e );

用e出栈S的栈顶元素值

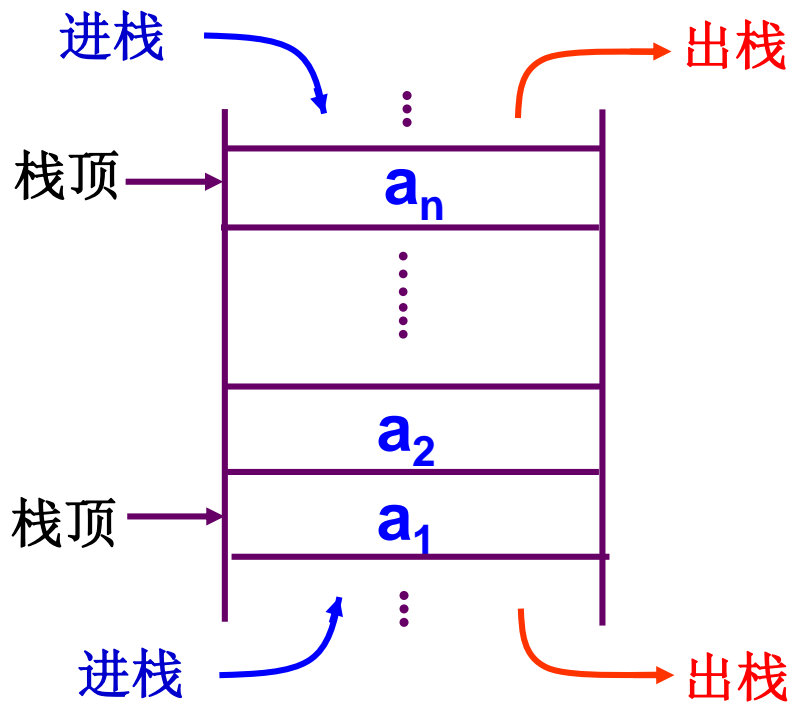
} ADT Stack



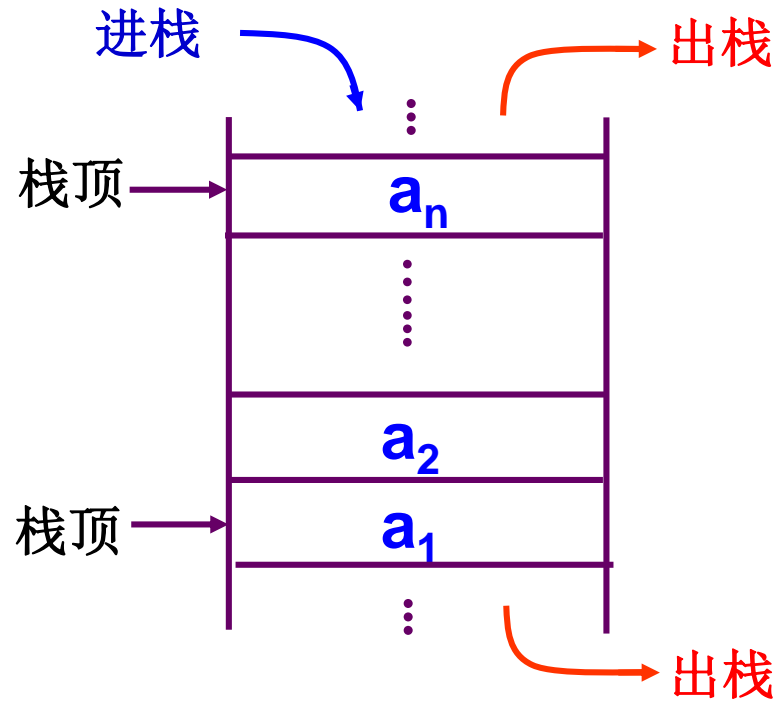
# 栈的逻辑结构



## 双栈和超栈



双栈



超栈



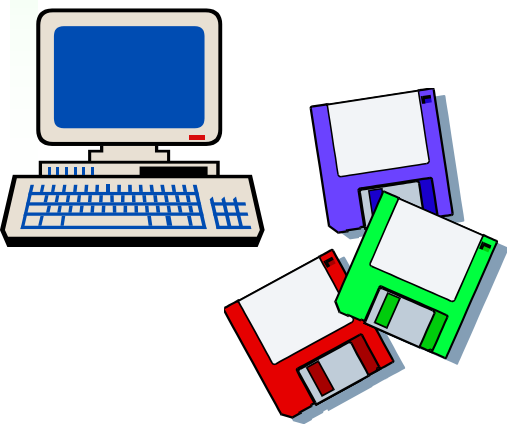
# 栈的存储结构

---

栈的顺序存储结构



栈的链式存储结构

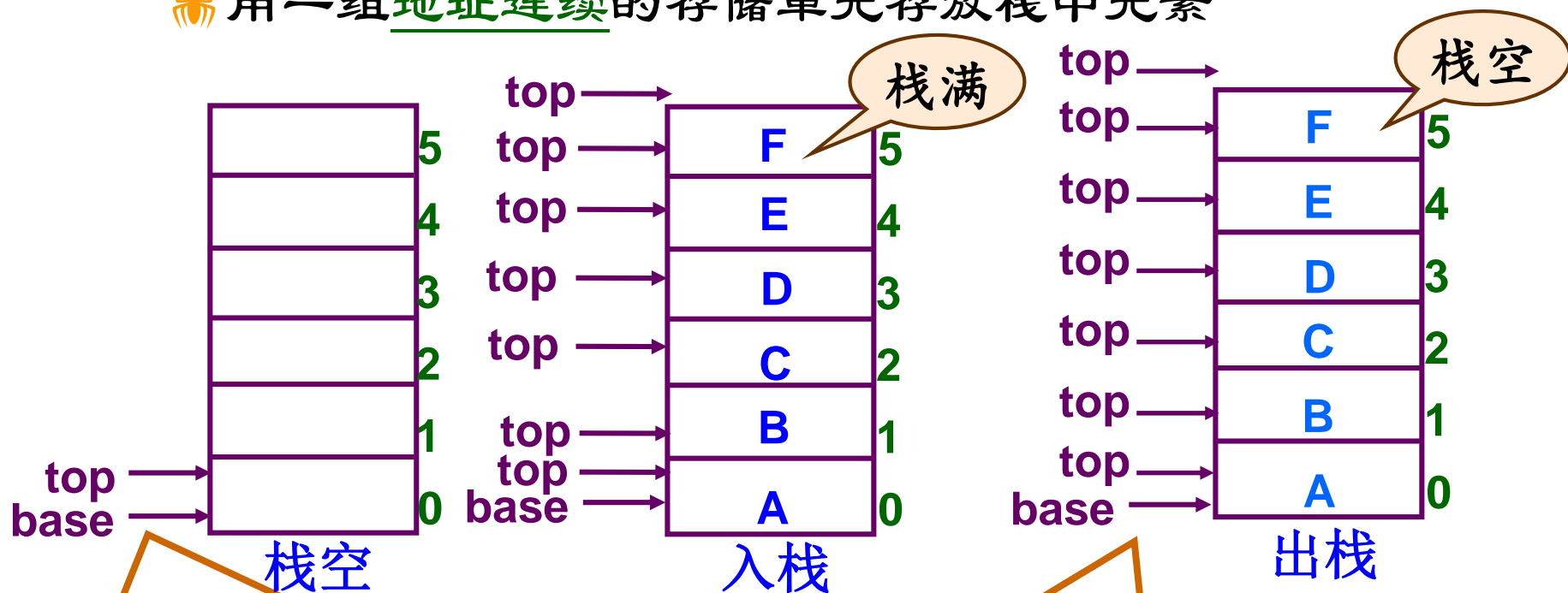


栈

# 栈的顺序存储结构

## 顺序栈

用一组地址连续的存储单元存放栈中元素

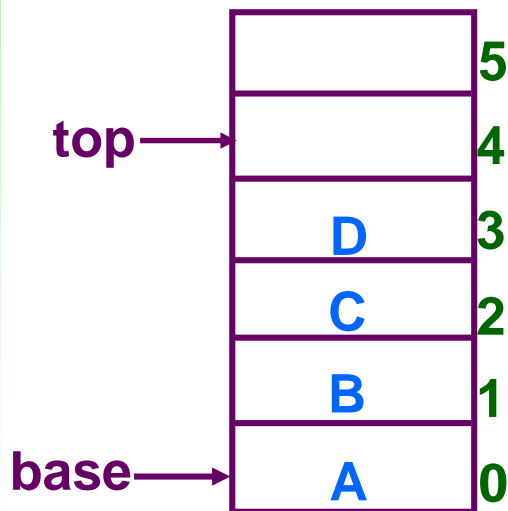


**base**—栈底指针，始终指向栈底的位置  
**top**—栈顶指针，指向实际栈顶后的空位置，初始指向栈底，即 **top=base**

设栈容量为 **stacksize**  
栈满— **top-base >= stacksize**  
栈空— **top==base**

# 栈的顺序存储结构

## 栈的 动态分配 顺序存储结构

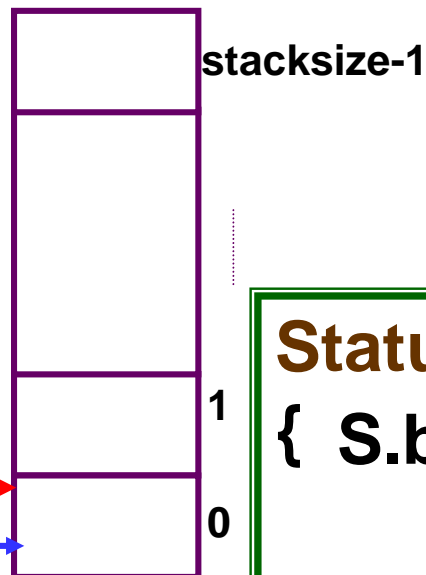


```
#define STACK_INIT_SIZE 100
//顺序栈存储空间的初始分配量
#define STACKINCREMENT 10
//顺序栈存储空间的分配增量
typedef int SElemType; //元素类型

typedef struct
{
    SElemType *base; //栈底指针
    SElemType *top; //栈顶指针
    int stacksize; //当前分配的存储容量
} SqStack;
```

# 栈的顺序存储结构

✎ 构造一个空顺序栈S



```
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10
typedef int SElemType;
typedef struct
{ SElemType *base;
  SElemType *top;
  int stacksize;
} SqStack;
```

动态申请  
顺序栈

```
Status InitStack( SqStack &S )
{ S.base = (SElemType*) malloc(STACK_
    INIT_SIZE*sizeof (SElemType));
  if (!S.base)
    exit(OVERFLOW);
  S.top=S.base;
  S.stacksize = STACK_INIT_SIZE;
  return OK;
}
```

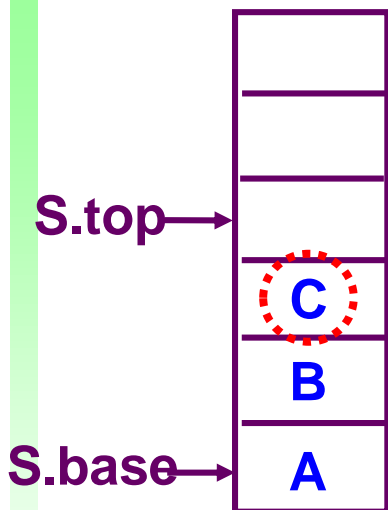
设置栈顶指针  
及栈容量

$T(n)=O(1)$

# 栈的顺序存储结构

~~取~~取栈顶元素

栈成员无变化  
无需引用型参数



```
Status GetTop(SqStack S, SElemType &e)
{
    if(S.top==S.base)
        return ERROR;
    e=*(S.top-1);
    return OK;
}
```

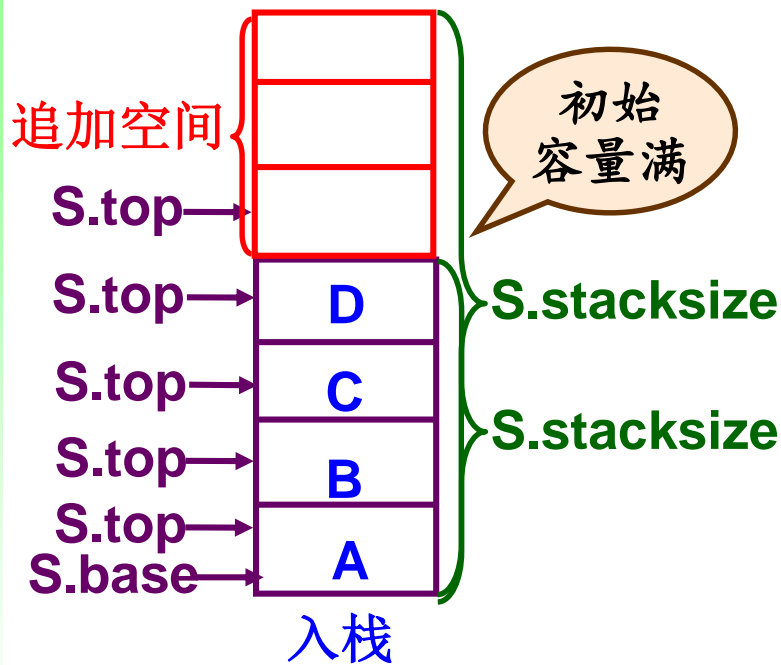
判断栈空

取栈顶元素

$T(n)=O(1)$

# 栈的顺序存储结构

入栈



顺序栈入栈语句:  $*(S.top++)=e;$

追加存储空间:

```
S.base = (SElemType *) realloc  
(S.base, (S.stacksize +  
STACKINCREMENT)  
* sizeof (SElemType));
```

追加空间后, 设置栈顶指针:

```
S.top = S.base + S.stacksize;
```

设置栈容量:

```
S.stacksize += STACKINCREMENT;
```

设栈(初始)容量为  $S.stacksize$   
栈满 —  $S.top - S.base \geq S.stacksize$

# 栈的顺序存储结构

~~入~~入栈

栈成员有变化  
需引用型参数

**Status** Push (SqStack &S, SElemType e )

{

若栈满，追加存储空间

if( S.top - S.base >= S.stacksize )

{ S.base = (SElemType \*)realloc(S.base, (S.stacksize +  
STACKINCREMENT) \* sizeof(SElemType));

if (!S.base)

exit(OVERFLOW);

S.top = S.base + S.stacksize;

S.stacksize += STACKINCREMENT;

设置新的栈顶  
指针、栈容量

}

\*S.top++ = e;

return OK;

入栈

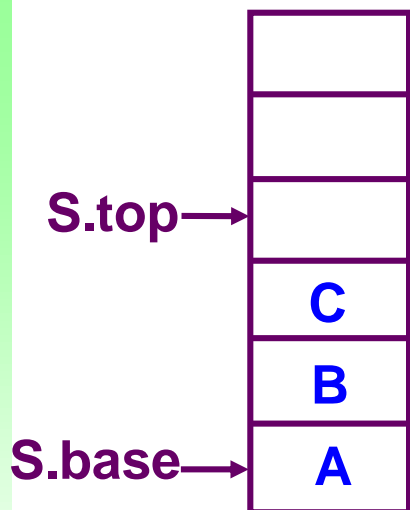
$T(n) = O(1)$

}



# 栈的顺序存储结构

 出栈



栈成员有变化  
需引用型参数

```
Status Pop(SqStack &S, SElemType &e)
{
    if(S.top == S.base)
        return ERROR;
    e = *--S.top;
    return OK;
}
```

判断是否栈空

出栈栈顶元素

$T(n)=O(1)$

# 栈的顺序存储结构

利用栈**后进先出**的特点判断

判断能否得到某出栈序列

例 已知入栈序列为A, B, C, D, E, 判断能否得到出栈序列C, B, D, A, E?

入栈序列:



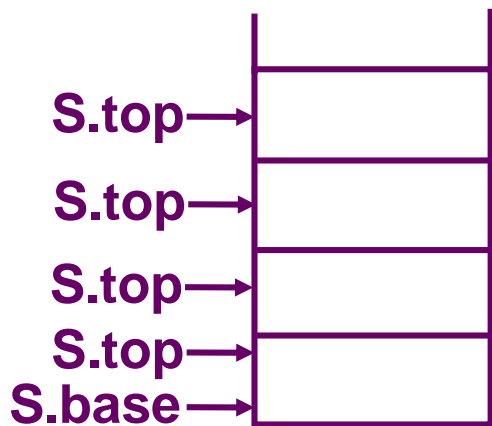
出栈序列:



过程:

- ① 元素A入栈
- ② 元素B入栈
- ③ 元素C入栈
- ④ 元素C出栈
- ⑤ 元素B出栈
- ⑥ 元素D入栈
- ⑦ 元素D出栈
- ⑧ 元素A出栈
- ⑨ 元素E入栈
- ⑩ 元素E出栈

可以得到



存储结构

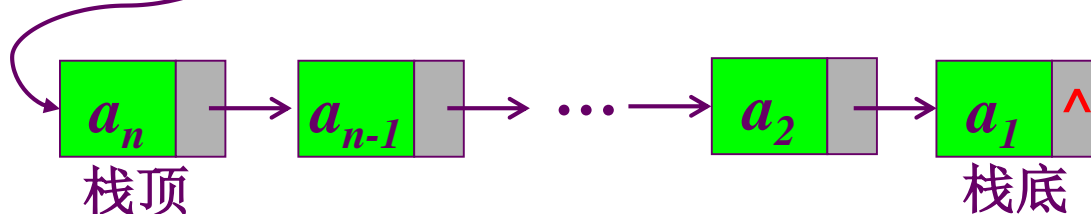
# 栈的链式存储结构



链栈

top

链栈通常不需要头结点



结点结构

data link

```
typedef struct Snode
{
    SElemType data; //数据域
    struct Snode *link; //指针域
}LinkStack;
```

链栈的操作是单链表的特例，易于实现



存储结构

# 栈的应用

解决具有**后进先出**  
特点的问题

函数的嵌套调用

递归的实现

回文游戏

多进制转换

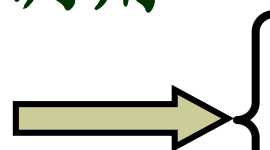
表达式求值

地图四染色

迷宫问题

递归函数

汉诺塔问题



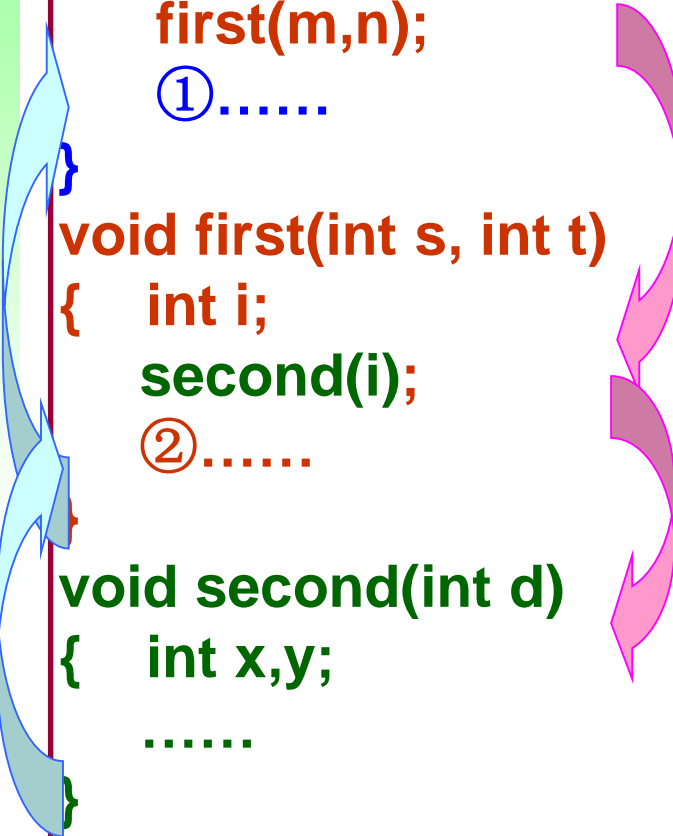
栈



# 栈的应用

## ~~函数~~函数的嵌套调用

```
main( )
{  int m,n;
    .....
    first(m,n);
    ①.....
}
void first(int s, int t)
{  int i;
    second(i);
    ②.....
}
void second(int d)
{  int x,y;
    .....
}
```



调用函数时，编译系统需要——

- ① 为形参及局部变量分配存储空间，并保存返回地址；
- ② 将控制权转移到被调用函数入口；

函数返回时，编译系统需要——

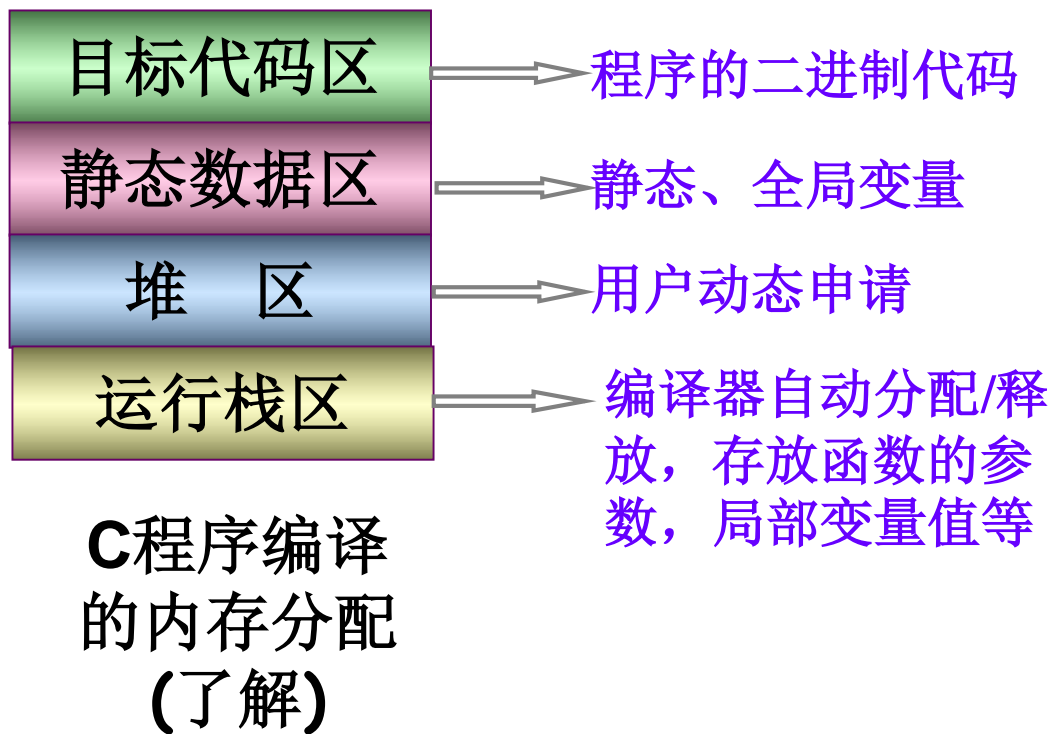
- ① 保存被调用函数计算结果；
- ② 释放被调用函数的数据区；
- ③ 根据返回地址将控制权转回调用函数。

函数之间信息传递和控制权转移必须通过栈来实现。

# 栈的应用

## 函数的嵌套调用

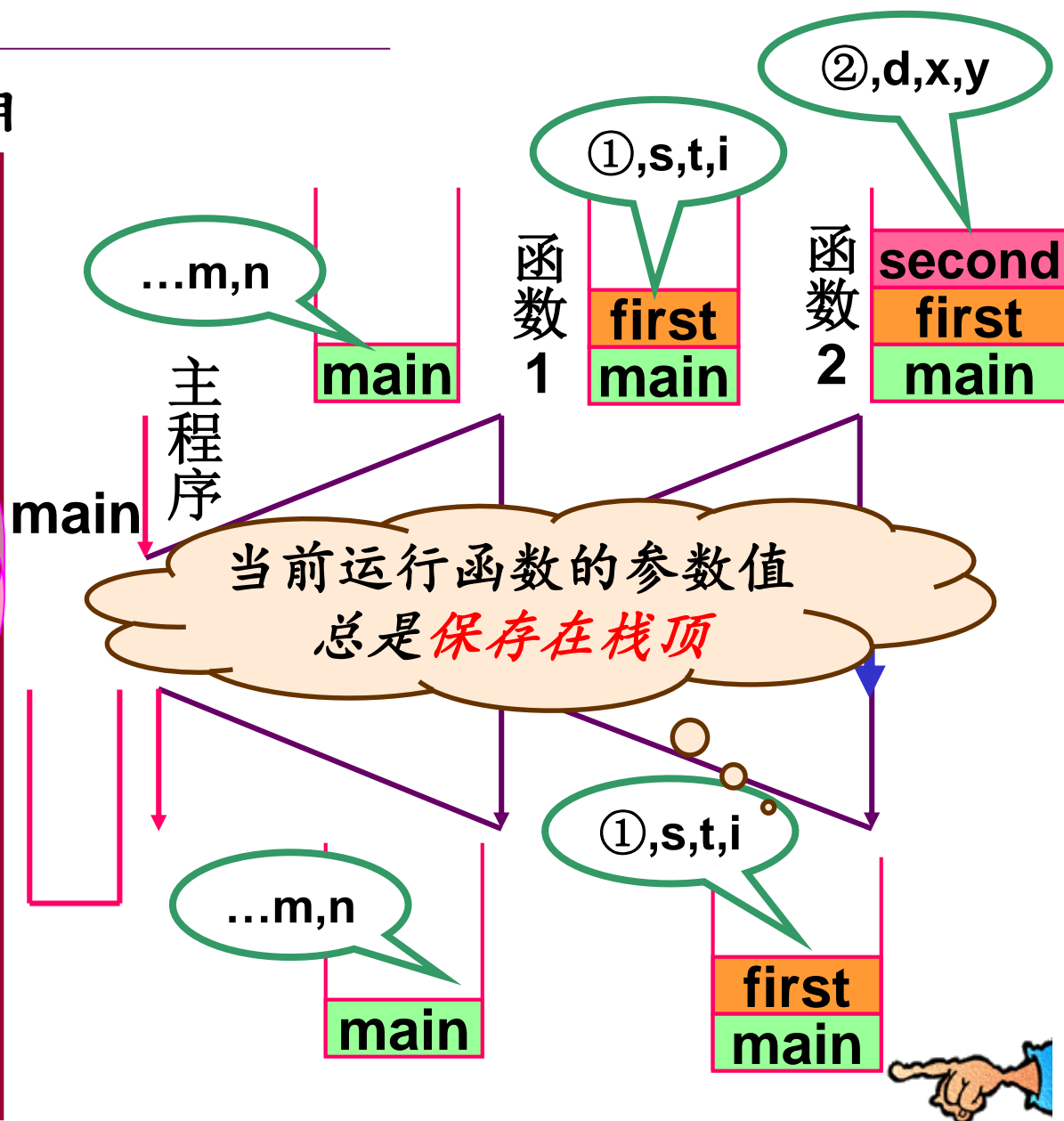
```
main(  
{  int m,n;  
  
    .....  
    first(m,n);  
    ①.....  
}  
void first(int s, int t)  
{  int i;  
    second(i);  
    ②.....  
}  
void second(int d)  
{  int x,y;  
    .....  
}
```



# 栈的应用

## 函数的嵌套调用

```
main()  
{ int m,n;  
  .....  
  first(m,n);  
  ①.....  
}  
  
void first(int s, int t)  
{ int i;  
  second(i);  
  ②.....  
}  
  
void second(int d)  
{ int x,y;  
  .....  
}
```




# 栈的应用

## 递归的实现

 递归函数——直接或间接调用自己的函数称为~

从前有座山，山上有座庙，庙里有个老和尚，老和尚给小和尚讲故事，讲的是：从前有座山，山上有座庙，庙里有个老和尚，老和尚给小和尚讲故事，讲的是……

 递归函数的实现——通过递归工作栈

```
void print(int w)
{
    int i;
    if (w!=0)
    {
        print(w-1);
        for(i=1;i<=w;++i)
            printf("%3d",w);
        printf("\n");
    }
}
```



实参=3，结果？



运行结果：

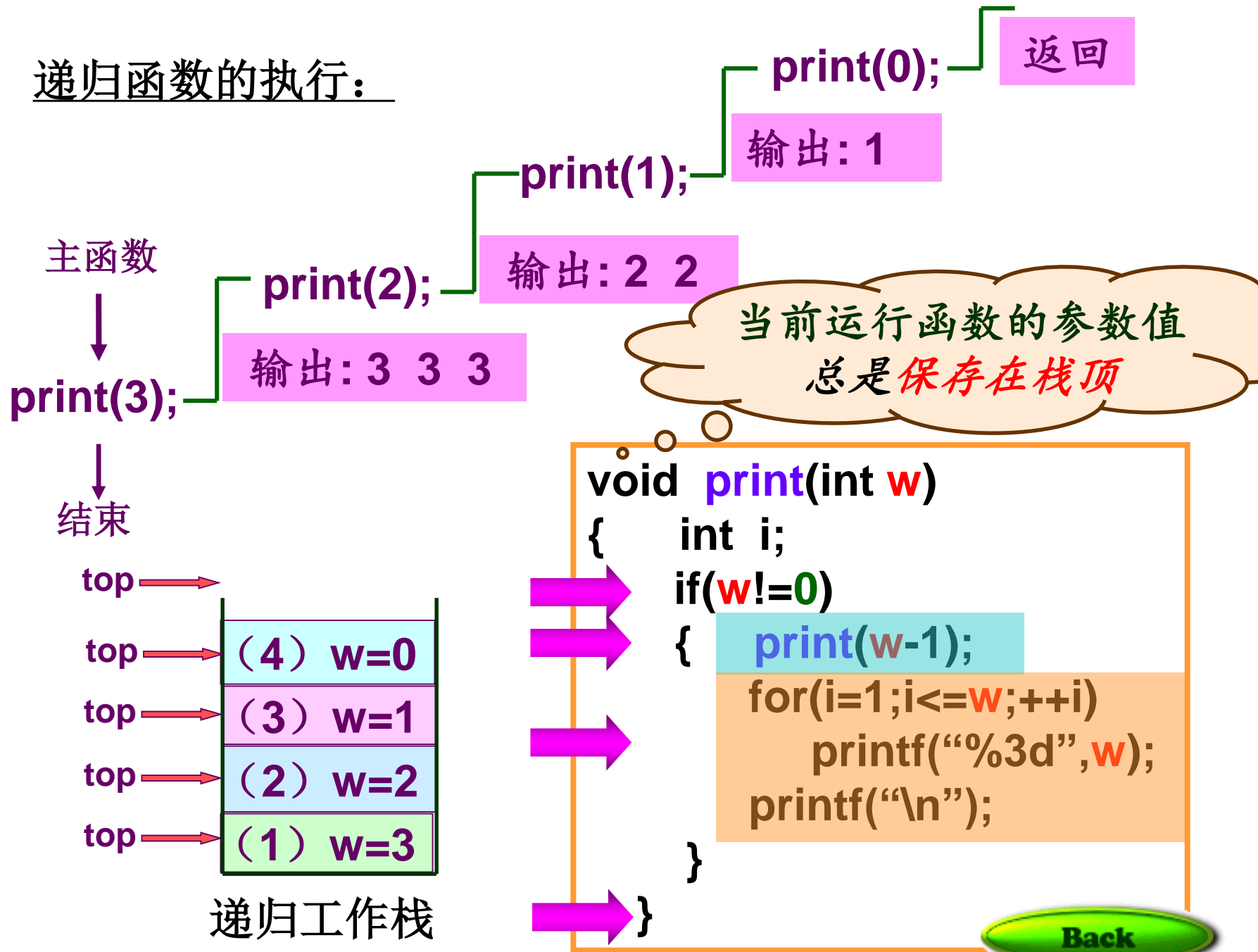
```
1
2  2
3  3  3
```



Hanoi



## 递归函数的执行:



# 栈的应用

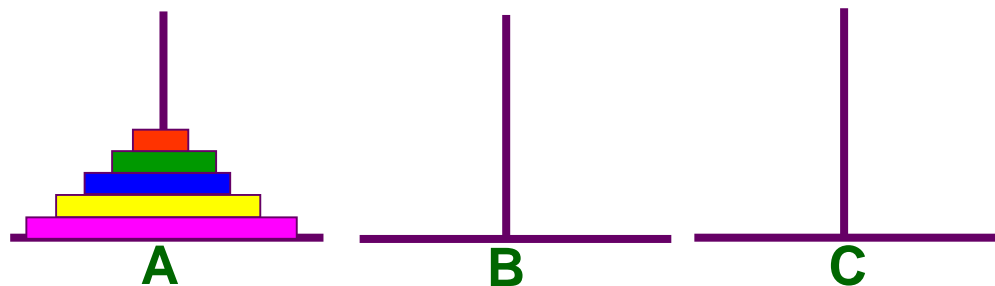
递归的实现

## Tower of Hanoi

相传在很久很久以前，在中东地区的一个寺庙里，几个和尚日夜不停地移动着64个盘子，日复一日，年复一年。移盘的规则是：……。据说这些盘子移完的那一天就是世界末日。

问题：有A,B,C三个塔座，A上套有n个直径不同的盘子，按直径从小到大叠放，编号1,2,...,n。要求将n个盘子从A借助B移到C，叠放顺序不变。移动原则为：

- ① 每次只能移1个圆盘；
- ② 圆盘可在三个塔座上任意移动；
- ③ 任何时刻，不能将大盘压到小盘上。



# 栈的应用

## 递归的实现

### Tower of Hanoi

将 $n$ 个盘子从A借助B移到C

✓ 解决方法：

分治法——将规模为 $n$ 的问题分解为若干个规模更小、与原问题形式相同的子问题，直至子问题的规模足够小，易于求解为止。

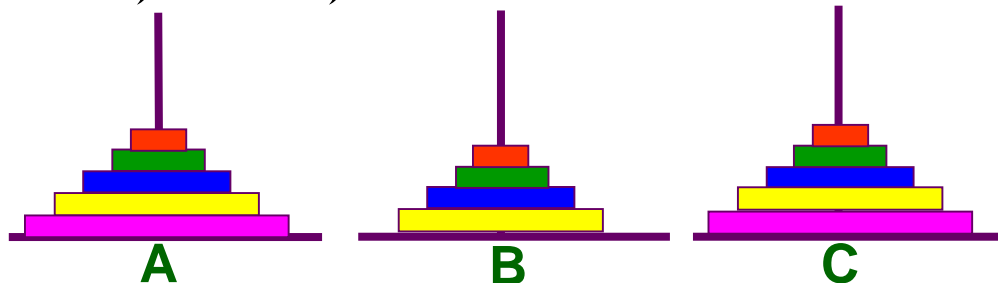
✱ $n=1$ 时，把1号盘子从A移到C；

✱ $n>1$ 时，①先把前 $n-1$ 个盘子从A借助C移到B，

②然后将 $n$ 号盘子从A移到C，

③再将 $n-1$ 个盘子从B借助A移到C。

——把 $n$ 个盘子的Hanoi问题转化为 $n-1$ 个盘子的Hanoi问题，……，最终转化为1个盘子的Hanoi问题。



# 栈的应用

递归的实现

## Tower of Hanoi

将n个盘子从A借助B移到C

```
void move(int h, char x,
          char z)
{
    printf("%d:%c->%c\n",
           h,x,z);
}
```

```
void hanoi(int n,char x,char y,char z)
{ if(n==1)
    move(1,x,z);
  else
  { hanoi(n-1,x,z,y);
    move(n,x,z);
    hanoi(n-1,y,x,z);
  }
```

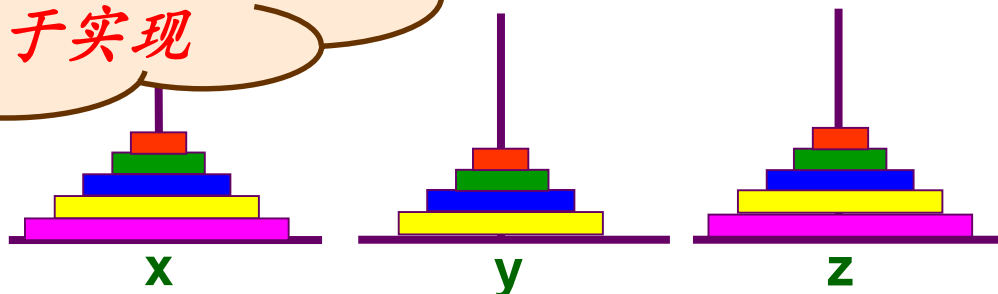
n=1时

① n-1个盘子从  
x借助z移到y

② 将n号盘子从x移到z

③ n-1个盘子从  
y借助x移到z

递归函数的特点——  
简单、易于实现



# 栈的应用

递归的实现

## Tower of Hanoi

将n个盘子从A借助B移到C

```
void hanoi(int n, char x, char y, char z)
{ if(n==1)
    move(1,x,z);
  else
  { hanoi(n-1,x,z,y);
    move(n,x,z);
    hanoi(n-1,y,x,z);
  }
}
```

n=1时

① n-1个盘子从  
x借助z移到y

② 将n号盘子从x移到z

③ n-1个盘子从  
y借助x移到z

递归函数的执行:

- ✓ 需使用递归工作栈
- ✓ 递归工作栈中保存:

n	x	y	z	返回地址
---	---	---	---	------

可用行号表示

算法演示

😊 n个盘子需要移动  $2^n - 1$  次, 64个盘子大约需  $1.8 \times 10^{19}$  次。  
假设和尚们每秒移动一次盘子, 大约需要一万亿年; 一微秒移动一次盘子, 大约需要一百万年。



栈的应用

# 栈的应用

## 回文游戏

有趣的对联

客上天然居 居然天上客  
人过大钟寺 寺钟大过人

例：原串 **madamimadam**  
**madamimadam**

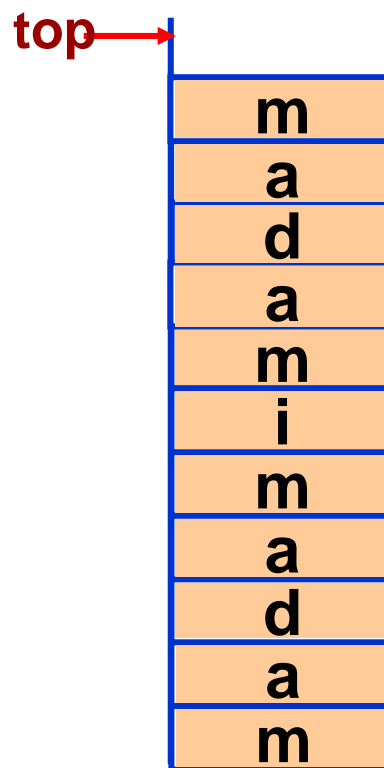


 回文——顺读与逆读相同的字符串。



如何判断某字符串  
是否是回文？

- ① 读入字符串
- ② 将字符依次入栈
- ③ 依次出栈字符，将出栈字符与原串中相应字符进行比较——  
若某字符对应不相等，**非回文**  
若直到栈空字符均对应相等，**回文**



栈的应用

# 栈的应用

## 多进制转换

### 转换原理

$$N = (a_n a_{n-1} \dots a_1 a_0)_2$$

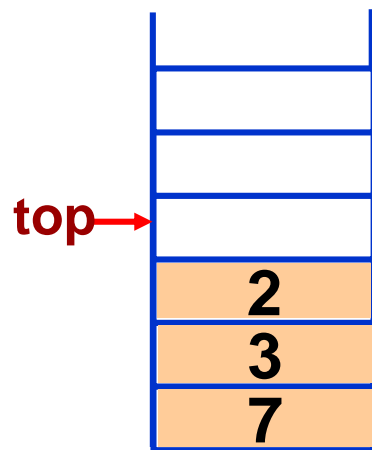
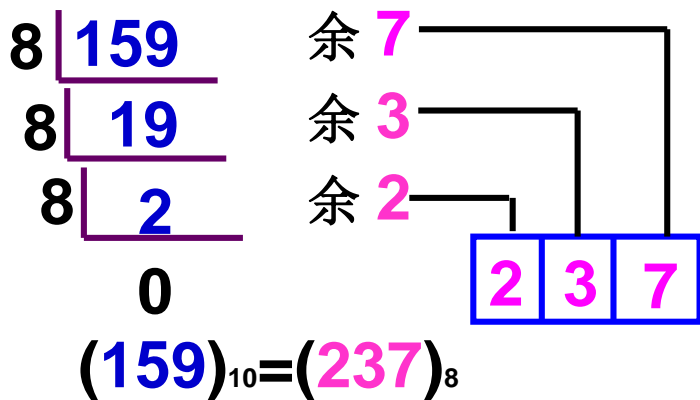
$$= a_n \times 2^n + a_{n-1} \times 2^{n-1} + \dots + a_1 \times 2^1 + a_0 \times 2^0$$

$$= 2 \times (a_n \times 2^{n-1} + a_{n-1} \times 2^{n-2} + \dots + a_1 \times 2^0) + a_0$$

$\therefore a_0$  是  $\frac{N}{2}$  的余数

连续除以“基”，取余数，直到商为0

例 把十进制数159转换成八进制数



$$(159)_{10} = (237)_8$$



栈的应用

# 栈的应用

后缀表达式中  
操作数的顺序不变

## 表达式求值

🐜 程序设计语言编译中的一个基本问题

🐜 表达式的三种标识方法：

✳️ 中缀表达式：运算符放在两个运算对象之间；

✳️ 后缀表达式：不包含括号，运算符放在两个运算对象的后面，所有的计算按运算符出现的顺序，严格从左向右进行（不考虑运算符优先级）；

✳️ 前缀表达式：类似于后缀表达式，不包含括号，运算符放在两个运算对象的前面。

中  
缀  
表  
达  
式

$a*b+c$

$a+b*c$

$a+(b*c+d)/e$



$ab*c+$

$abc*+$

$abc*d+e/+$

后  
缀  
表  
达  
式



# 栈的应用

## 表达式求值

中缀表达式求值：操作数栈和运算符栈

编译程序自左向右扫描至表达式结束：

- ✓ 若遇到操作数，一律进操作数栈；
- ✓ 若遇到运算符，需比较当前运算符与栈顶运算符优先级

① 优先级相同：

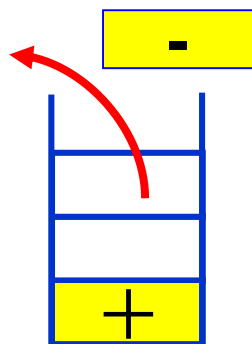
② 当前运算符优先级低：

③ 当前运算符优先级高：

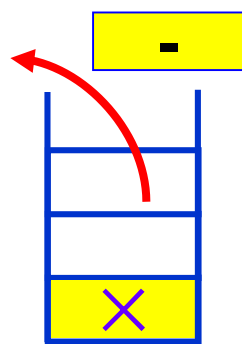
出栈栈顶运算符，从操作数栈中取2个操作数，运算并将结果入栈

当前运算符入栈

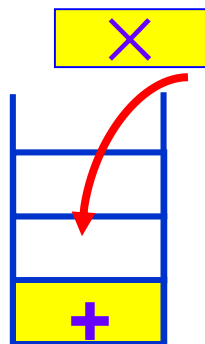
① 如  $5+3-2$



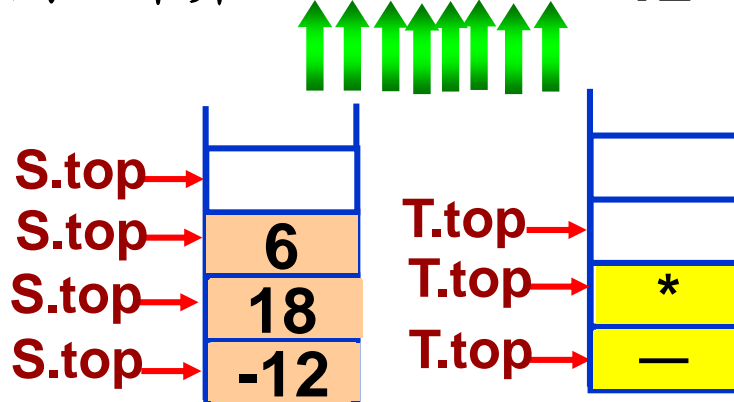
② 如  $5*3-2$



③ 如  $5+3*2$



例 计算  $2+4-3*6 = -12$



# 栈的应用

## 表达式求值

中缀表达式求值：操作数栈和运算符栈

分析：

算符优先级

当前	+	-	*	/	(	)	#
栈顶							
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

#表示表达式开始和结束

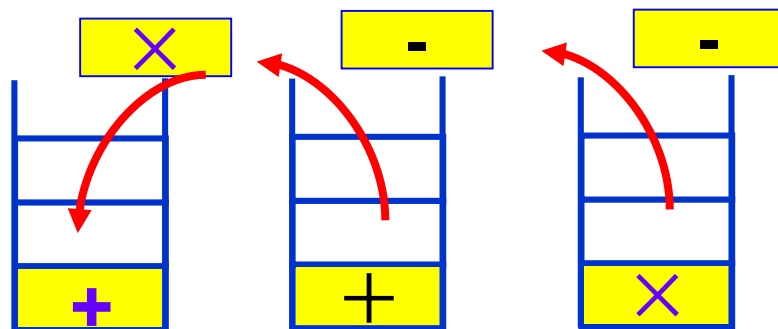
- ✓ 若栈顶运算符优先级低，当前运算符入栈；
- ✓ 若栈顶运算符优先级相同或较大，则栈顶运算符出栈

✓ 当前运算符为(，一律入栈；

✓ 当前运算符为)，一律出栈栈顶运算符；

✓ =表示(与)、#与#相遇，即括号内或表达式运算完毕，应脱括号或脱#(出栈栈顶)；

✓ 栈顶运算符为(，当前运算符一律入栈；



# 栈的应用

## 表达式求值

中缀表达式求值：操作数栈和运算符栈

分析：

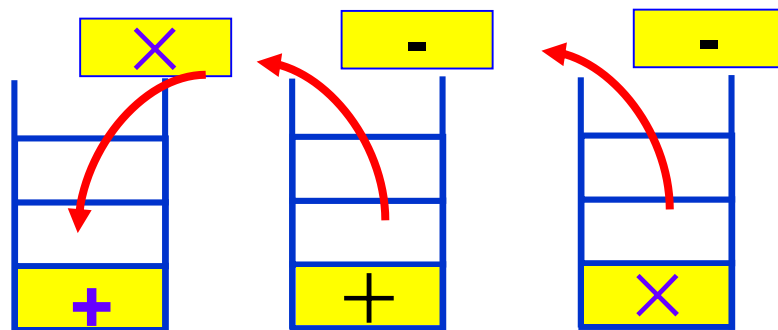
算符优先级

当前 \ 栈顶	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

#表示表达式开始和结束

- ✓ 若栈顶运算符优先级低，当前运算符入栈；
- ✓ 若栈顶运算符优先级相同或较大，则栈顶运算符出栈

- ✓ 栈顶运算符为)，一律出栈；
- ✓ 栈顶运算符为#，当前运算符一律入栈；
- ✓ 当前运算符为#，栈顶运算符一律出栈；
- ✓ 语法正确的表达式中，)与(、#与)、(与#不会相遇。



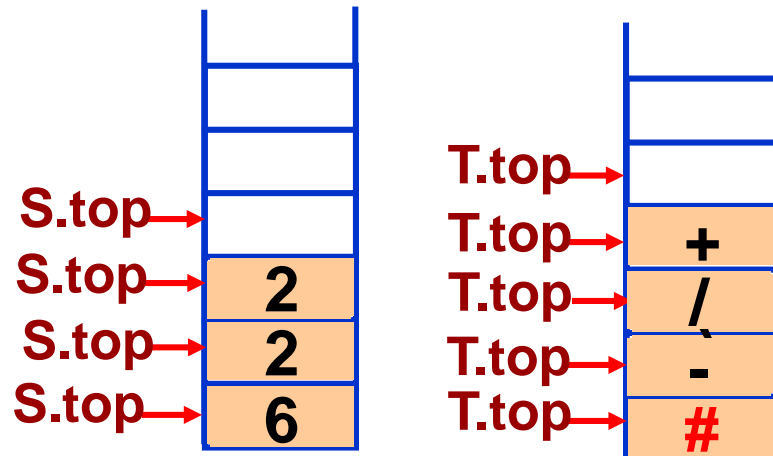
# 栈的应用

## ❌ 表达式求值

🕷 中缀表达式求值：操作数栈和运算符栈

例 计算  $2*(1+3)-4/2 = 6$

$\#2*(1+3)-4/2\#$



当前	+	-	*	/	(	)	#
栈顶							
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

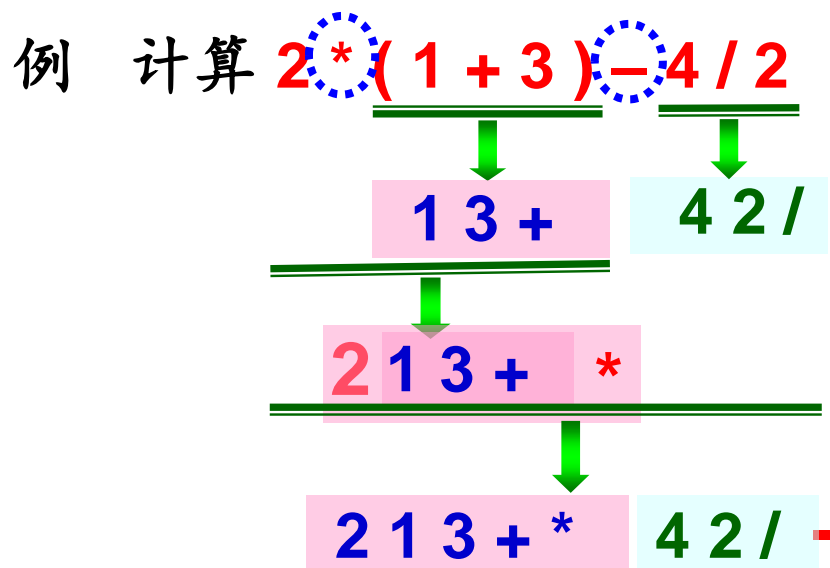
# 栈的应用

## ❌ 表达式求值

🐜 后缀表达式求值：只需要操作数栈

🐜 编译程序自左向右扫描至表达式结束：

- ① 读入表达式一个字符
- ② 若是操作数，入栈，转④
- ③ 若是运算符，从栈中出栈2个操作数进行运算，将运算结果入栈，转④
- ④ 若表达式未结束，转①；若表达式结束，栈顶即表达式值



# 栈的应用

## ❌ 表达式求值

🐜 后缀表达式求值：只需要操作数栈

🐜 编译程序自左向右扫描至表达式结束：

① 读入表达式一个字符

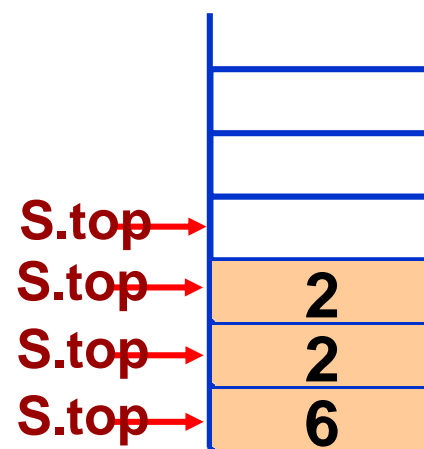
② 若是操作数，入栈，转④

③ 若是运算符，从栈中出栈2个操作数进行运算，将运算结果入栈，转④

④ 若表达式未结束，转①；若表达式结束，栈顶即表达式值

例 计算  $2 * (1 + 3) - 4 / 2 = 6$

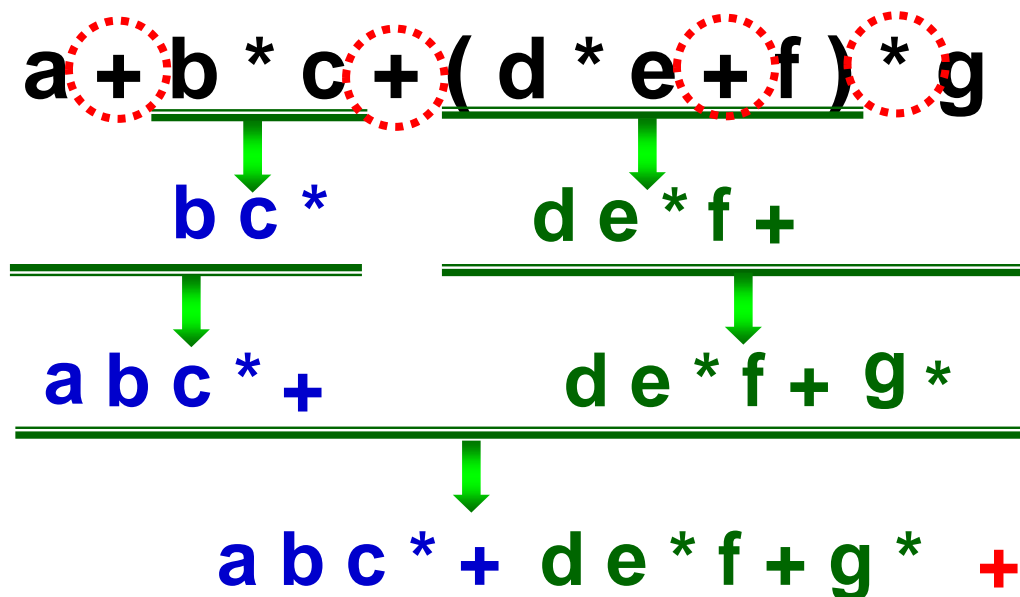
后缀表达式：2 1 3 + \* 4 2 / -



# 栈的应用

## 表达式求值

将中缀表达式转换为后缀表达式



后缀表达式中  
操作数的顺序不变

$abc* + de*f + g* +$

# 栈的应用

## ❌ 表达式求值

🕷 将中缀表达式转换为后缀表达式

# a + b \* c + ( d \* e + f ) \* g #    abc\*+de\*f+g\*+

后缀表达式中

操作数的顺序不变

↑  
(  
+  
#

运算符栈

后缀表达式:

a b c \* + d e \* f + g \* +

结束

转换算法步骤：依次读入中缀表达式字符

- ① 当前字符是操作数时，直接发送给后缀式
- ② 当前字符是运算符时，

若当前运算符的优先数高于栈顶运算符，则进栈  
否则，出栈栈顶运算符发送给后缀式



栈的应用



# 栈的应用

## ✎ 地图四染色问题

📖 1852年，刚从伦敦大学毕业的**费南西斯·古色利**望着地图发呆，突然发现了这一问题。

📖 之后这一问题引起了越来越多数学家的兴趣，但仍未得到该问题在数学上的严格证明。

📖 直到1976年，在美国伊利诺伊大学的**两台计算机上**，耗时**1200小时、经过100亿次判断**，最终证明了四色猜想。

📖 但证明还未止步，数学家们还在寻求更简洁优美的证明方法。在该问题的研究过程中，很多数学理论随之产生，对**拓扑学**和**图论**的发展起到了重要的推动作用。



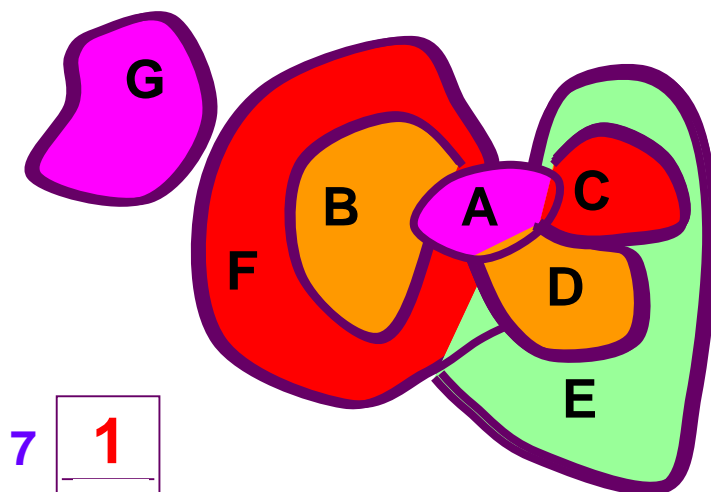
# 栈的应用

## ✂️ 地图四染色问题

R[7][7]

	1	2	3	4	5	6	7
1	0	1	1	1	1	1	0
2	1	0	0	0	0	1	0
3	1	0	0	1	1	0	0
4	1	0	1	0	1	1	0
5	1	0	1	1	0	1	0
6	1	1	0	1	1	0	0
7	0	0	0	0	0	0	0

1# 紫色  
2# 黄色  
3# 红色  
4# 绿色



7	1
6	3
5	4
4	2
3	3
2	2
1	1

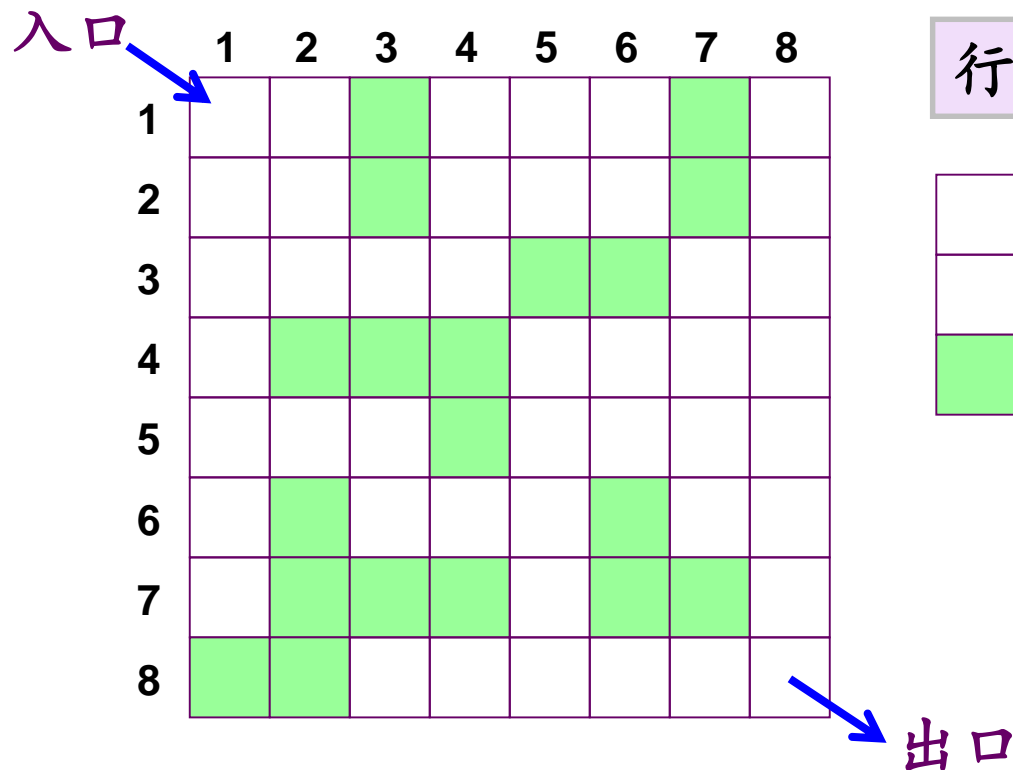


栈的应用

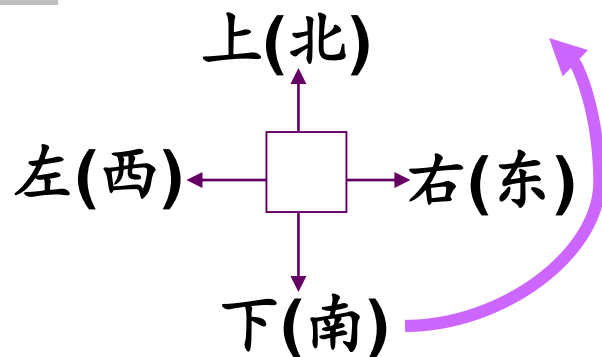
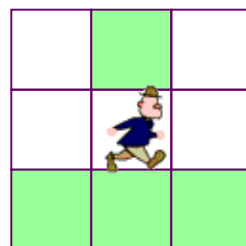
# 栈的应用

## 迷宫问题

问题：寻找一条从入口到出口的通路。



行走规则



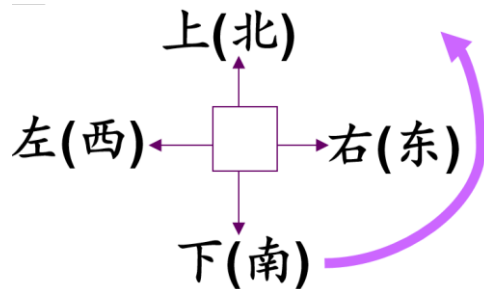
从向下开始，按逆时针方向搜索下一步可能前进的位置


# 栈的应用

为了尽量保留之前的走法，需要一个后进先出的结构来保存从入口到当前位置的路径。

## 迷宫问题

**问题：寻找一条从入口到出口的通路。**



	1	2	3	4	5	6	7	8
1	i							
2	i							
3	i							
4	i							
5	i							
6	i							
7								
8								

# 栈

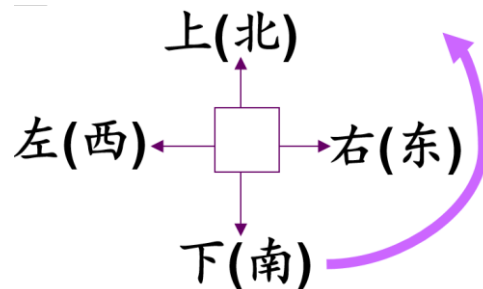
(7, 1)
(6, 1)
(5, 1)
(4, 1)
(3, 1)
(2, 1)
(1, 1)

# 栈的应用

为了尽量保留之前的走法，需要一个后进先出的结构来保存从入口到当前位置的路径。

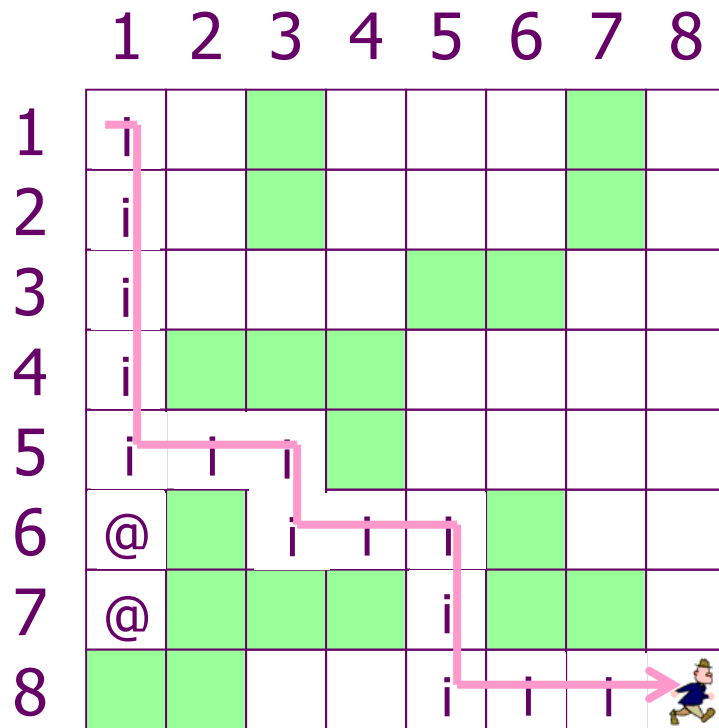
## 迷宫问题

问题：寻找一条从入口到出口的通路。



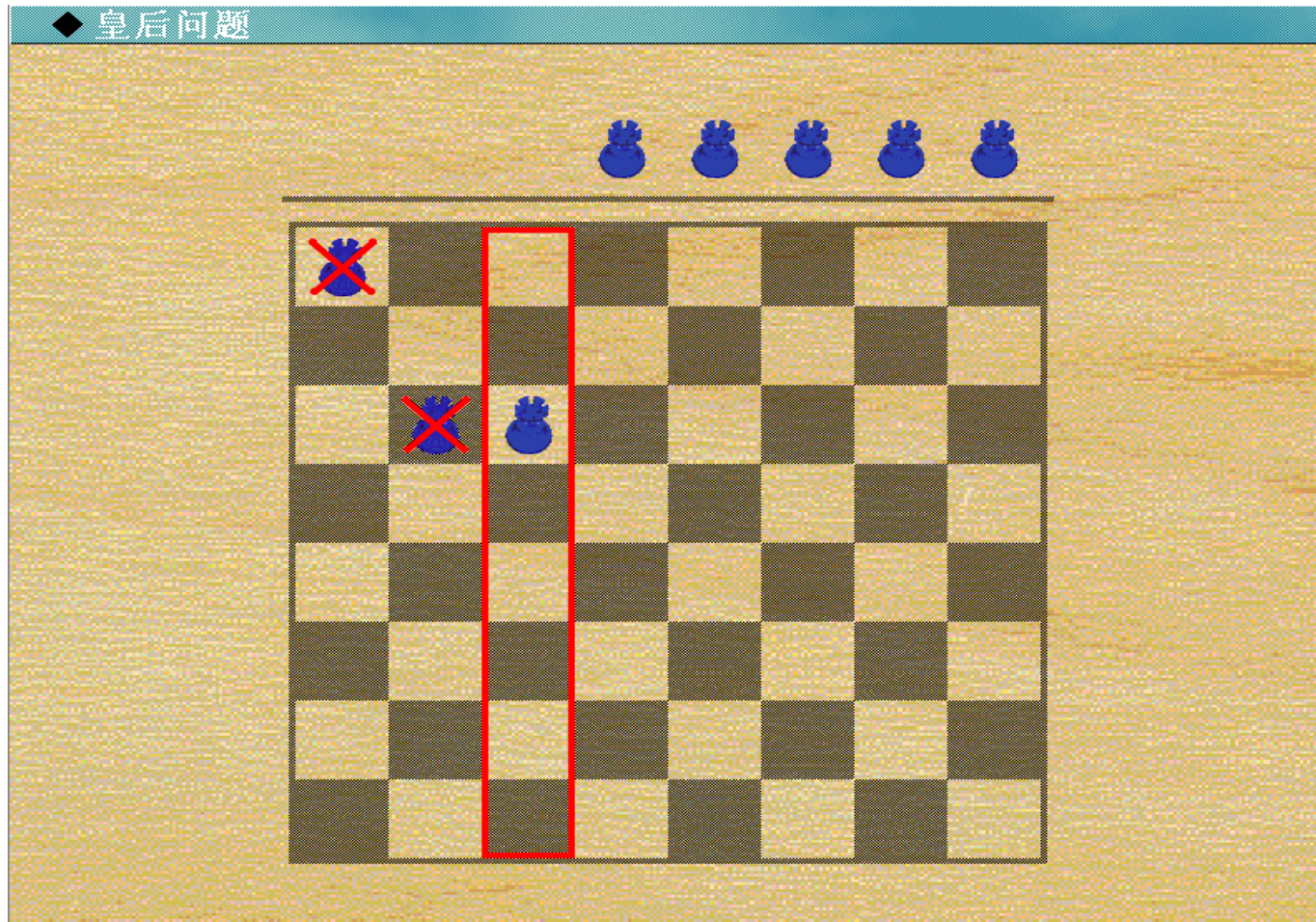
(8, 8)
⋮
(6, 4)
(6, 3)
(5, 3)
(5, 2)
(5, 1)
(4, 1)
(3, 1)
(2, 1)
(1, 1)

栈



# 栈的应用

## ✂皇后问题



# 队列(Queue)

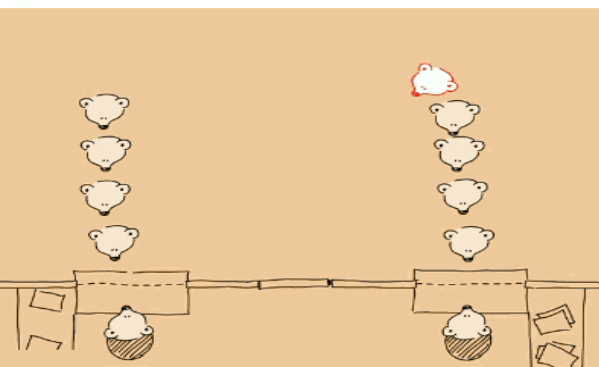
队列的逻辑结构



队列的存储结构



队列的应用

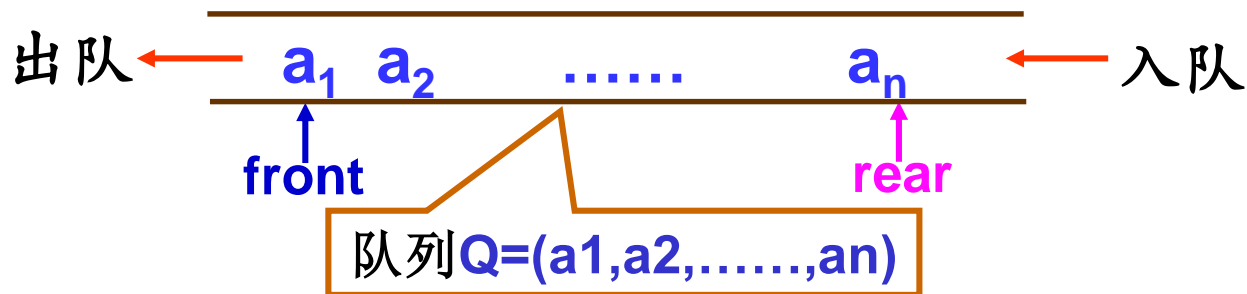


Back



# 队列的逻辑结构

❖ 队列——限定只能在表的一端进行插入，在表的另一端进行删除的线性表。

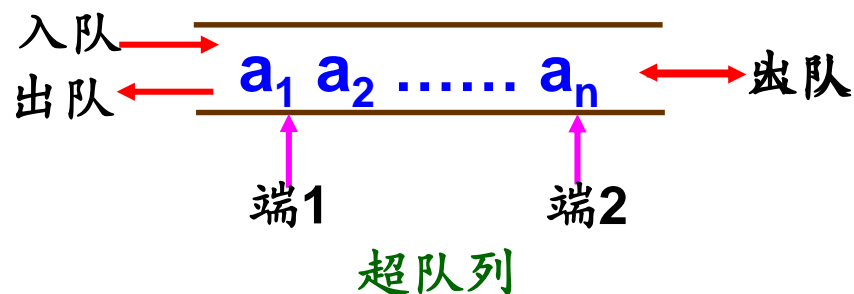
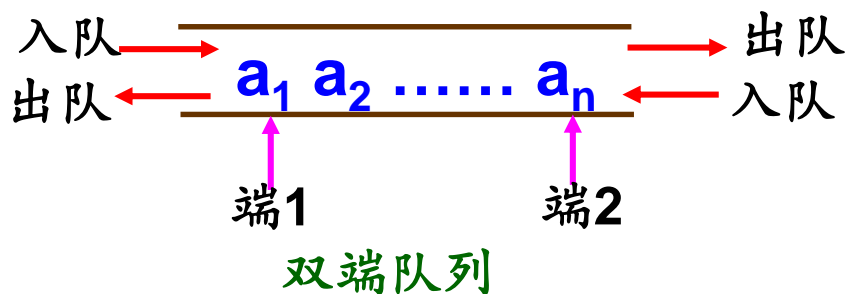


❖ 队尾(rear)——允许插入的一端

❖ 队头(front)——允许删除的一端

❖ 特点：先进先出(FIFO)

❖ 双端队列和超队列





# 抽象数据类型队列的定义

## ADT Queue

{ 数据对象:  $D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系:  $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$   
(约定 $a_1$ 端为队头,  $a_n$ 端为队尾)

基本操作:

InitQueue( &Q );	构造一个空队列Q
DestroyQueue( &Q );	销毁队列Q
ClearQueue( &Q );	将队列Q清空
QueueEmpty( Q );	判断队列Q是否为空
QueueLength( Q );	返回队列Q中元素个数
GetHead( Q, &e );	取队头元素
EnQueue( &Q, e );	元素e入队
DeQueue( &Q, &e );	出队

} ADT Queue



队列

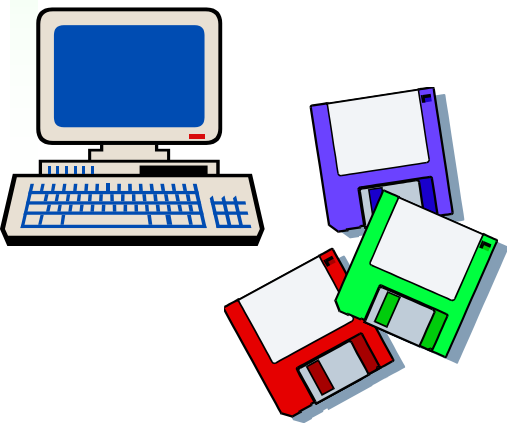
# 队列的存储结构

---

链队列



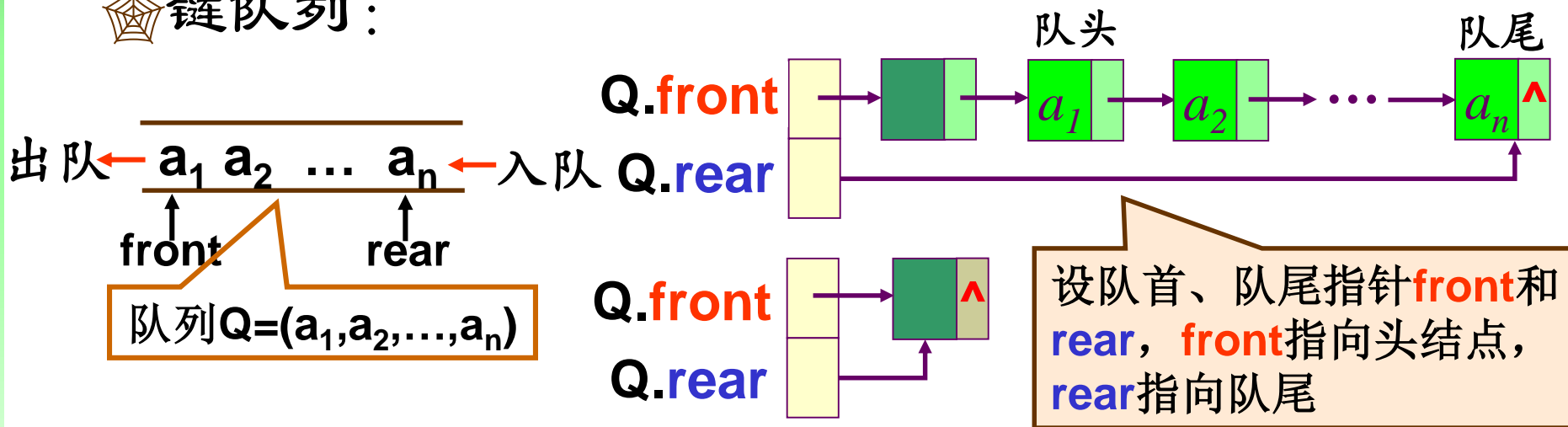
顺序队列



队列

# 队列的链式存储结构

链队列：



结点结构

```
typedef struct Qnode
{
    QElemType data; //数据域
    struct Qnode *link; //指针域
}QNode, *QueuePtr;
```

链队列结构

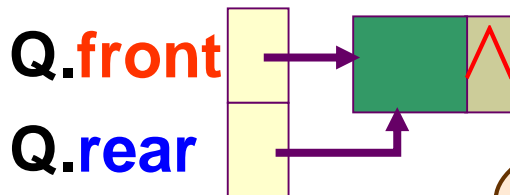
```
typedef struct
{
    QueuePtr front; //队头指针
    QueuePtr rear; //队尾指针
}LinkQueue;
```

LinkQueue Q;

# 队列的链式存储结构

✎ 构造一个空链队列Q

LinkQueue Q;



队列成员有变化  
需引用型参数

```
Status InitQueue ( LinkQueue &Q)
```

```
{
```

```
    Q.front=Q.rear= (QueuePtr)malloc(sizeof(QNode));
```

```
    if (!Q.front)
```

```
        exit (OVERFLOW);
```

```
    Q.front->link= NULL;
```

```
    return OK;
```

```
}
```

建立链队列头结点

设置头结点指针域

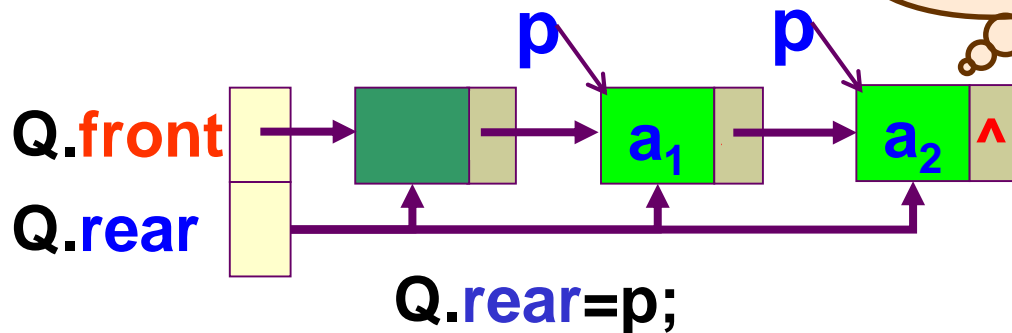
$T(n)=O(1)$

# 队列的链式存储结构

❌ 入队

$Q.rear \rightarrow link = p;$

$Q.front$ 、 $Q.rear$   
如何改变?



**Status** EnQueue( LinkQueue &Q, QElemType **e** )

{ QueuePtr p;

$p = (\text{QueuePtr}) \text{malloc} (\text{sizeof} (\text{QNode}));$

生成新结点

$\text{if} (!p) \quad \text{exit}(\text{OVERFLOW});$

$p \rightarrow \text{data} = e; \quad p \rightarrow \text{link} = \text{NULL};$

新结点赋值

$Q.rear \rightarrow \text{link} = p;$

$Q.rear = p;$

入队

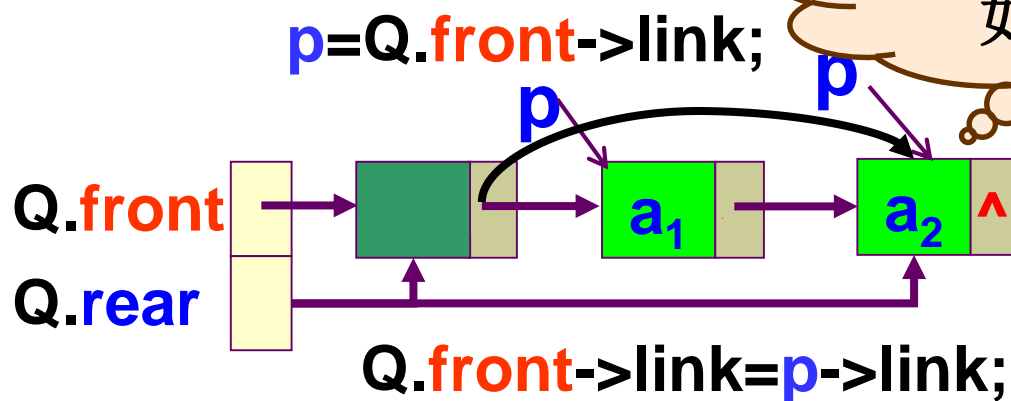
return OK;

$T(n) = O(1)$

}

# 队列的链式存储结构

❌ 出队



```
Status DeQueue( LinkQueue &Q, QElemType &e)
```

```
{ QueuePtr p;  
  if(Q.front == Q.rear) return ERROR;
```

判断队列是否为空

```
  p = Q.front->link; e = p->data;
```

出队

```
  Q.front->link = p->link;
```

```
  if (Q.rear == p) Q.rear = Q.front;
```

出队结点是否是最后结点

```
  free(p); return OK;
```

$T(n) = O(1)$

```
}
```



# 队列的顺序存储结构

顺序队列：

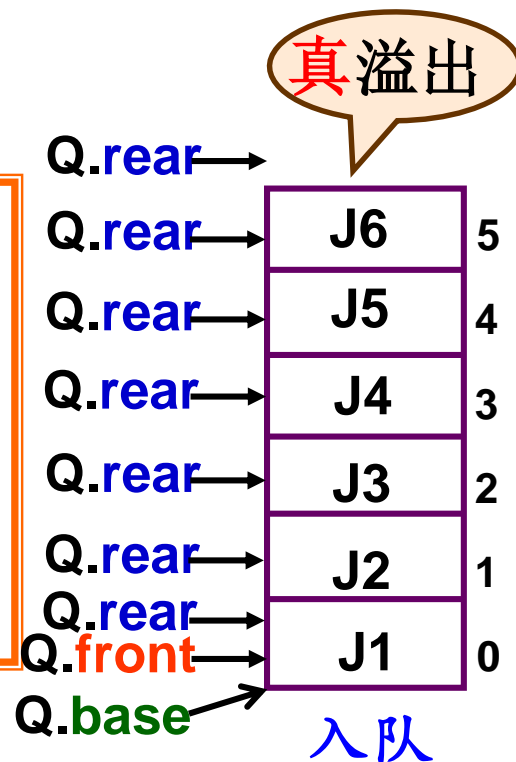
```
#define MAXQSIZE 100 //队列长度
typedef struct
{ QElemType *base;    //存储空间基址
  int front; //头指针，指向队头元素
  int rear;  //尾指针，指向队尾元素的下一个位置
} SqQueue;
```

**SqQueue** Q;

初值： **Q.front**=**Q.rear**=0;

队空： **Q.front**==**Q.rear**;

入队： **Q.base**[**Q.rear**++]=**e**;



真溢出条件： **Q.front**==0;  
**Q.rear**==MAXQSIZE;

# 队列的顺序存储结构

顺序队列：

```
#define MAXQSIZE 100 //队列长度
typedef struct
{ QElemType *base; //存储空间基址
  int front; //头指针，指向队头元素
  int rear; //尾指针，指向队尾元素的下一个位置
} SqQueue;
```

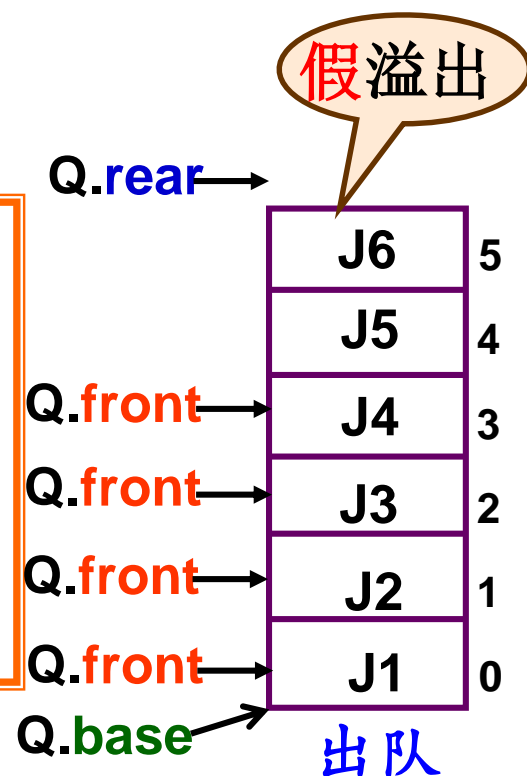
**SqQueue** Q;

初值： **Q.front**=**Q.rear**=0;

队空： **Q.front**==**Q.rear**;

入队： **Q.base**[**Q.rear**++]=**e**;

出队： **e**=**Q.base**[**Q.front**++];



真溢出条件： **Q.front**==0;  
**Q.rear**==MAXQSIZE;

假溢出条件： **Q.front** ≠ 0;  
**Q.rear**==MAXQSIZE;



# 队列的顺序存储结构

蜘蛛网 顺序队列：

蜘蛛 假溢出问题的解决方案

真溢出条件：

$Q.front == 0;$

$Q.rear == MAXQSIZE;$

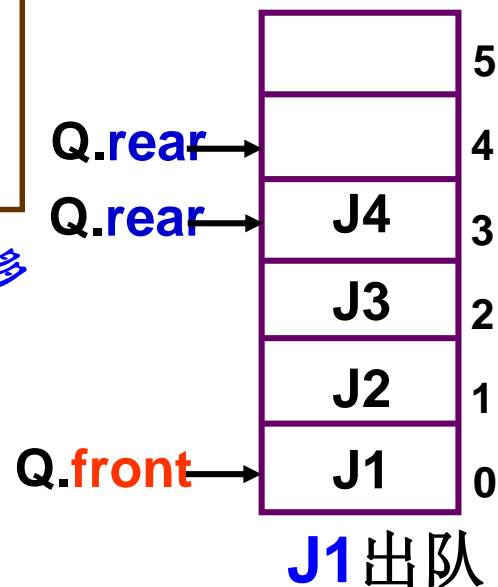
假溢出条件：

$Q.front \neq 0;$

$Q.rear == MAXQSIZE;$

方案①：队首固定，出队元素后剩余元素下移

缺点——需要移动大量元素



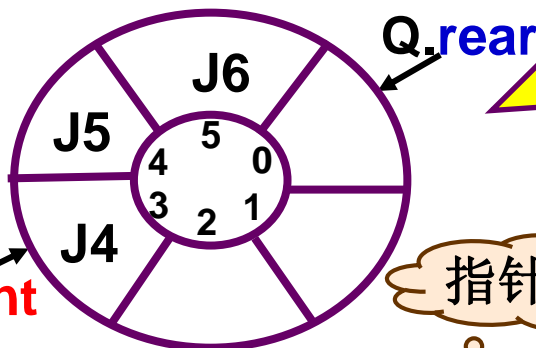
## 假溢出问题的解决方案

**Q.rear==MAXQSIZE;**

**Q.rear==MAXSIZE**, 则令**Q.rear=0**



**Q.front**



## 满足下标范围、 首尾相连

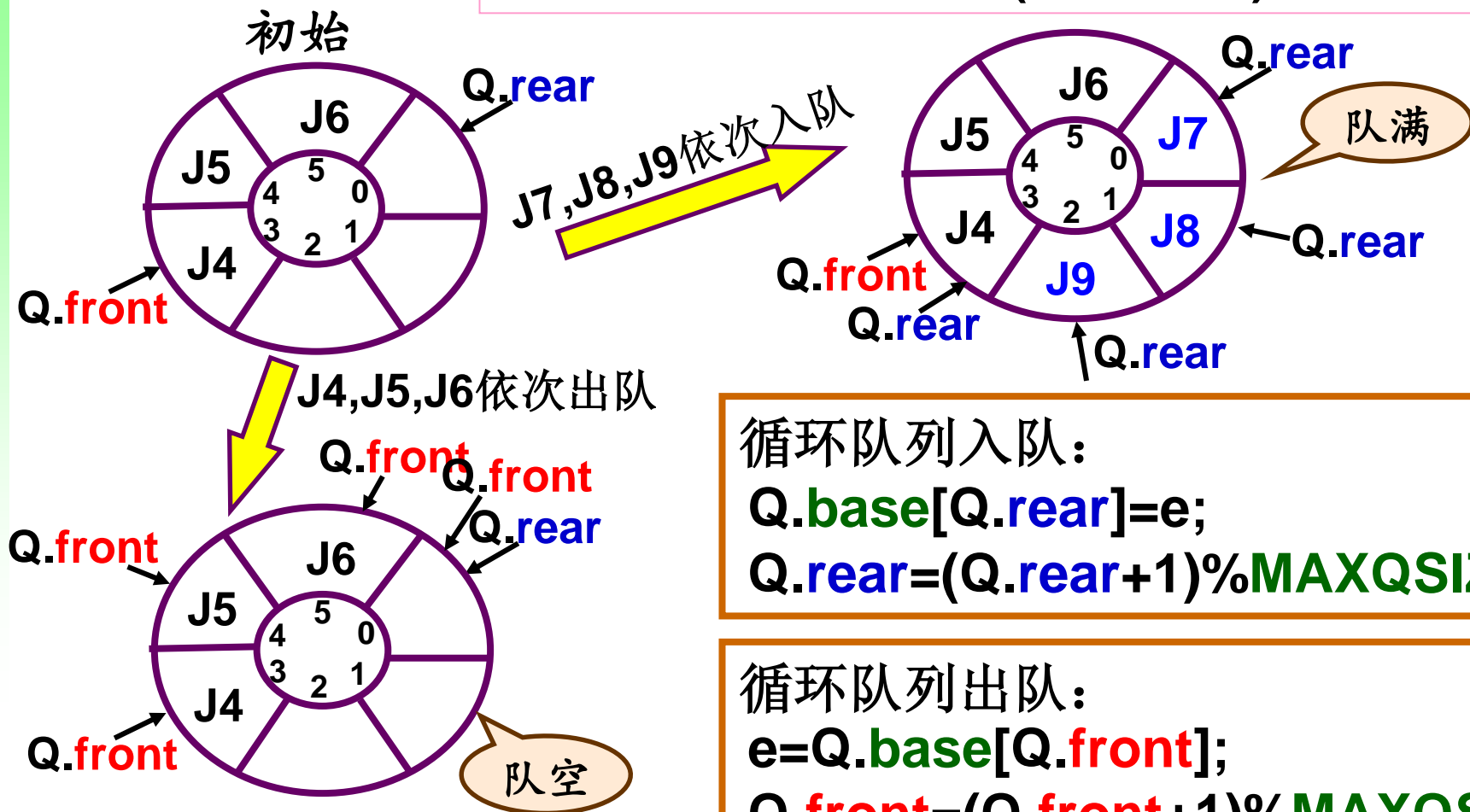
**Q.front后移:  $Q.front = (Q.front + 1) \% MAXSIZE$ ;**

# 队列的顺序存储结构



循环队列：  $Q.rear$ 后移：  $Q.rear=(Q.rear+1)\%MAXSIZE$ ;

$Q.front$ 后移：  $Q.front=(Q.front+1)\%MAXSIZE$ ;



循环队列入队：

$Q.base[Q.rear]=e$ ;

$Q.rear=(Q.rear+1)\%MAXQSIZE$ ;

循环队列出队：

$e=Q.base[Q.front]$ ;

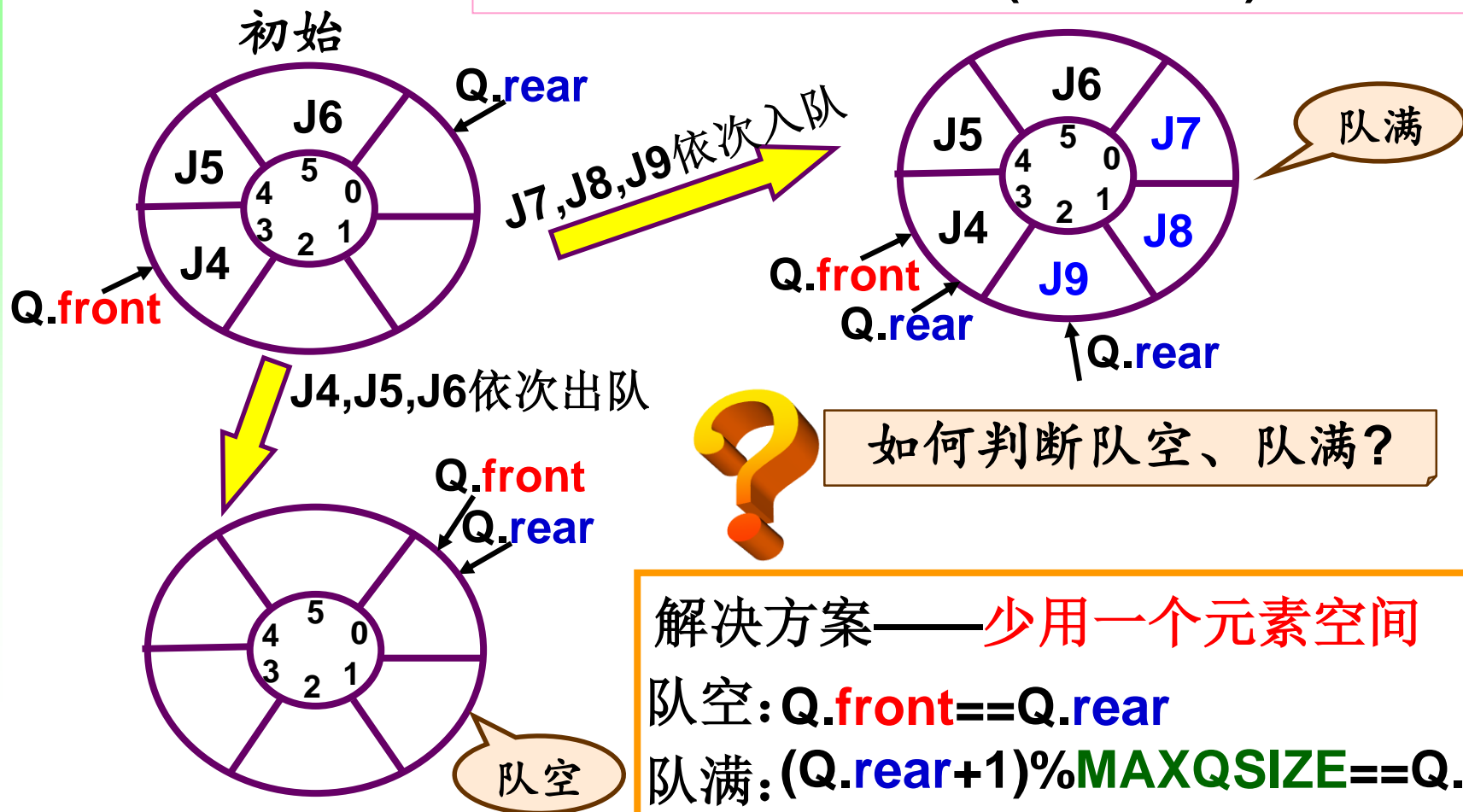
$Q.front=(Q.front+1)\%MAXQSIZE$ ;

# 队列的顺序存储结构



循环队列：  $Q.rear$  后移：  $Q.rear = (Q.rear + 1) \% MAXSIZE$ ;

$Q.front$  后移：  $Q.front = (Q.front + 1) \% MAXSIZE$ ;



解决方案——少用一个元素空间

队空：  $Q.front == Q.rear$

队满：  $(Q.rear + 1) \% MAXQSIZE == Q.front$

# 队列的顺序存储结构



循环队列：

构造空的循环队列Q

```
#define MAXQSIZE 100
typedef struct
{ QElemType *base;
  int front;
  int rear;
} SqQueue;
```

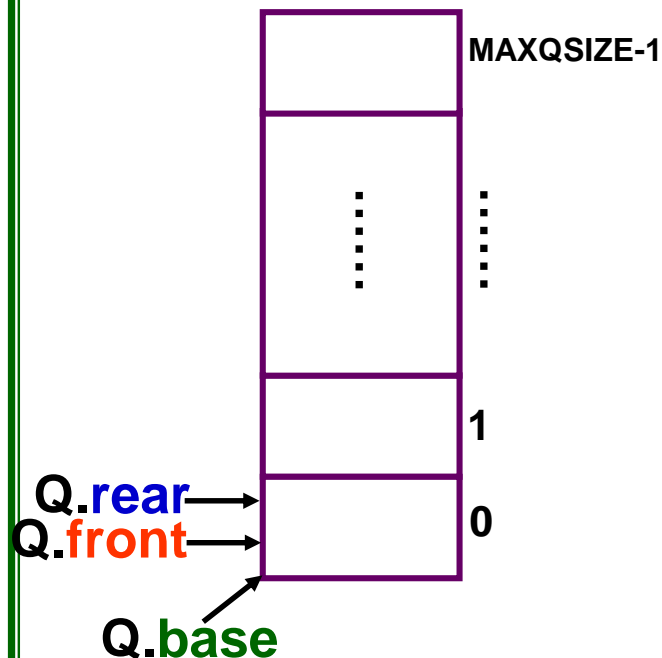
```
Status InitQueue(SqQueue &Q )
```

```
{
  Q.base=(QElemType *)malloc(
    MAXQSIZE*sizeof(QElemType));
  if(!Q.base)
    exit(OVERFLOW);
  Q.front=Q.rear =0;
  return OK;
}
```

申请存储空间

设置队头和  
队尾指针

$T(n)=O(1)$



# 队列的顺序存储结构

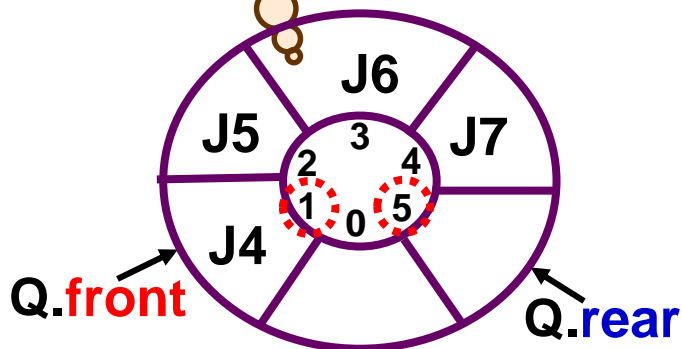
🕸 循环队列：

🗑 求循环队列中的元素个数

```
int QueueLength(SqQueue Q)
{
    return (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
}
```

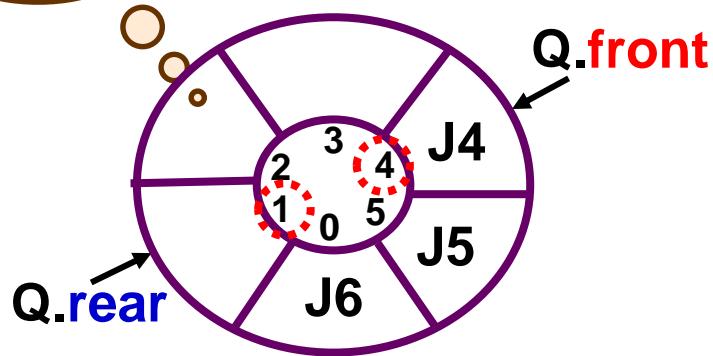
$$(5-1+6)\%6=4$$

+6和%6抵消



$$(1-4+6)\%6=3$$

%6未起作用



$$T(n)=O(1)$$

# 队列的顺序存储结构

🕸 循环队列：

🗑 循环队列入队

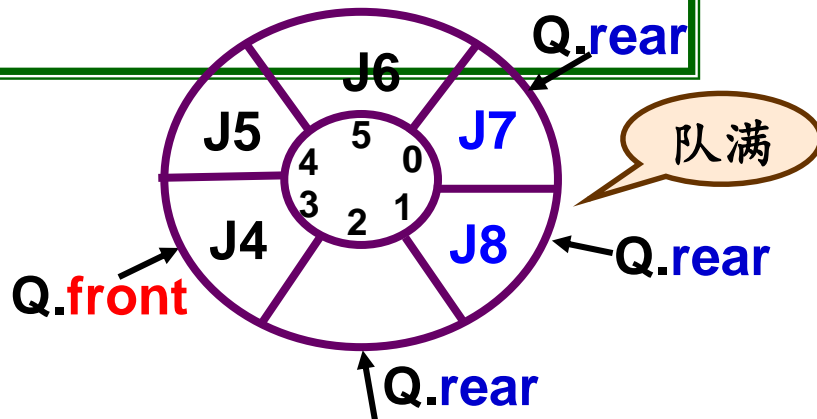
```
Status EnQueue( SqQueue &Q, QElemType e)
{
    if((Q.rear+1)%MaxQSize==Q.front)
        return ERROR;
    Q.base[Q.rear] = e;
    Q.rear=(Q.rear+1)%MaxQSize;
    return OK;
}
```

是否队满

入队

修改队尾指针

$T(n)=O(1)$



# 队列的顺序存储结构

🕸 循环队列：

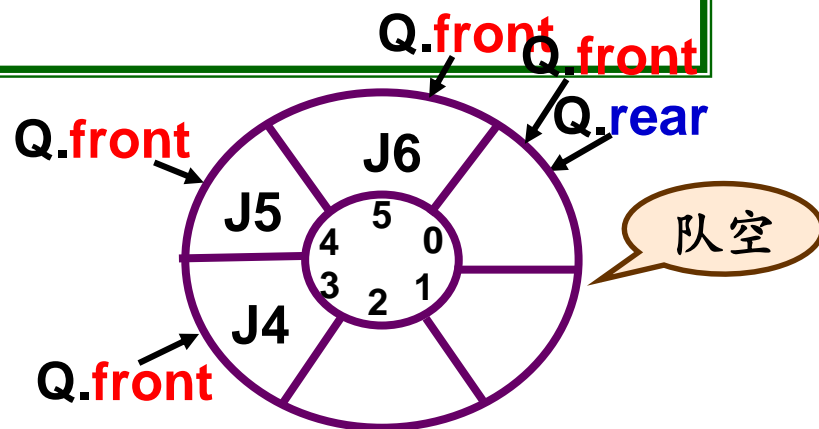
🗑 循环队列出队

```
Status DeQueue( SqQueue &Q, QElemType &e )  
{  
    if(Q.front==Q.rear) 是否队空  
        return ERROR;  
    e=Q.base[Q.front]; 出队  
    Q.front=(Q.front+1)%MaxQSize; 修改队头指针  
    return OK;  
}
```

$$T(n)=O(1)$$



队列存储





# 队列的应用

解决具有**先进先出**  
特点的问题

划分子集问题

图的广度优先遍历

农夫过河问题

离散事件模拟

优先级队列

作业调度

迷宫问题



队列



# 本章小结

- 掌握栈和队列的基本概念
- 掌握栈和队列的顺序、链式存储结构及基本操作的实现
- 掌握栈的应用实例
- 了解双栈、双端队列等特殊的数据结构