



# 数据结构

## 华中科技大学计算机学院

郑渤龙

智能大数据管理与分析实验室

主页: <https://bolongzheng.com>

Email: [bolongzheng@hust.edu.cn](mailto:bolongzheng@hust.edu.cn)

QQ: 14060806

QQ群: 684 194 942 →

(加群时请注明班级姓名)



2022年春数据结构

群号: 684194942



扫一扫二维码, 加入群聊。

## 第三章 栈和队列

引言：对线性表  $L=(a_1, a_2, \dots, a_n)$ ,

可在任意第  $i$  ( $i=1, 2, \dots, n, n+1$ ) 个位置插入新元素，或删除任意第  $i$  ( $i=1, 2, \dots, n$ ) 个元素

受限数据结构——插入和删除受限制的线性表。

1. 栈(stack)
2. 队列(queue)
3. 双队列(deque)

都属于插入和删除受限制的线性表。

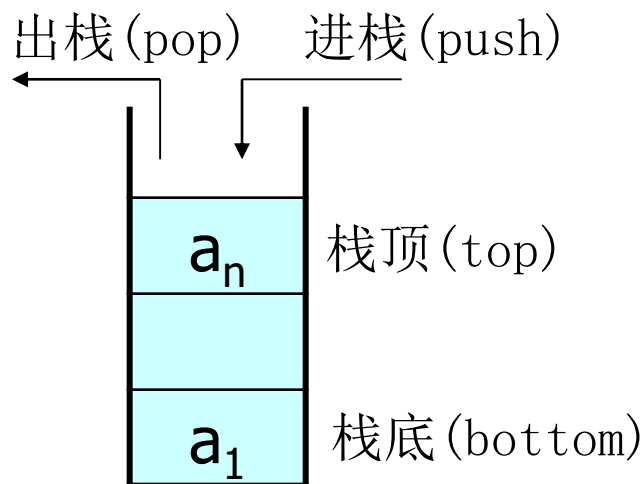


## 3.1 栈(stack)

京张铁路  
青龙桥车站







栈的示意图

**进栈：** 插入一个元素到栈中。

或称：**入栈、推入、压入、push。**

**出栈：** 从栈删除一个元素。

或称：**退栈、上托、弹出、pop。**

**栈顶：** 允许插入、删除元素的一端(表尾)。

**栈顶元素：** 处在栈顶位置的元素。

**栈底：** 表中不允许插入、删除元素的一端。

**空栈：** 不含元素的栈。

**栈的元素的进出原则：**

**“后进先出”，“Last In First Out”。**

**栈的别名：** “后进先出”表、“LIFO”表、  
反转存储器、地窖、堆栈。





## 2. 栈的基本操作

(1)  $\text{Initstack}(s)$ : 置 $s$ 为空栈。

(2)  $\text{Push}(s, e)$ : 元素 $e$ 进栈 $s$ 。

若 $s$ 已满, 则发生溢出。

若不能解决溢出, 重新分配空间失败, 则插入失败。

(3)  $\text{Pop}(s, e)$ : 删除栈 $s$ 的顶元素, 并送入 $e$ 。

若 $s$ 为空栈, 发生“下溢”(underflow);

为空栈时, 表示某项任务已完成。

(4)  $\text{Gettop}(s, e)$ : 栈 $s$ 的顶元素拷贝到 $e$ 。

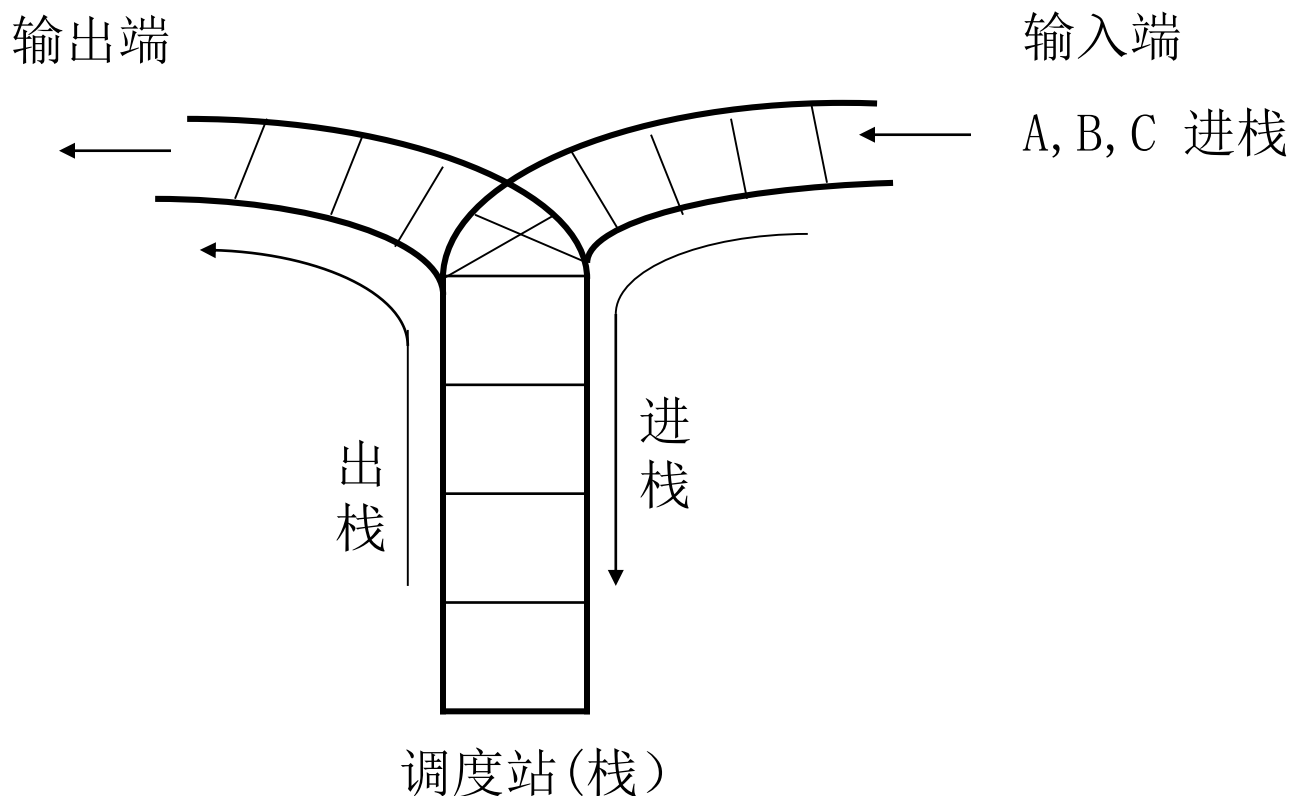
若 $s$ 为空栈, 则结束拷贝。

(5)  $\text{Empty}(s)$ : 判断 $s$ 是否为空栈。

若 $s$ 为空栈, 则 $\text{Empty}(s)$ 为true; 否则为false。



### 3. 理解栈操作（模拟铁路调度站）

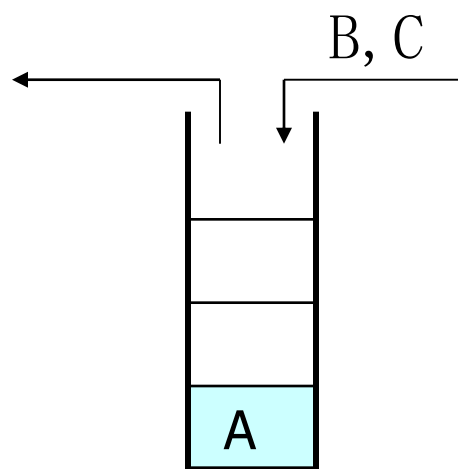


讨论: 😊😊😊

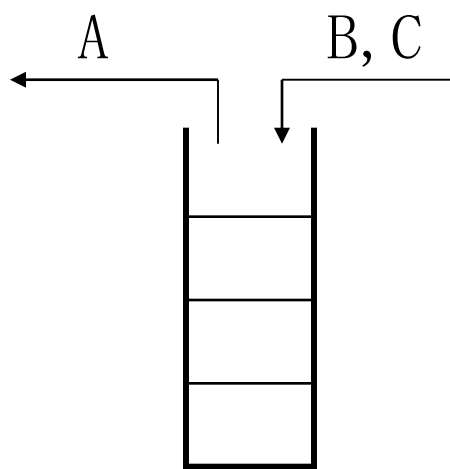
假设依次输入3个元素(车厢)A, B, C到栈(调度站)中,  
可得到哪几种不同输出?



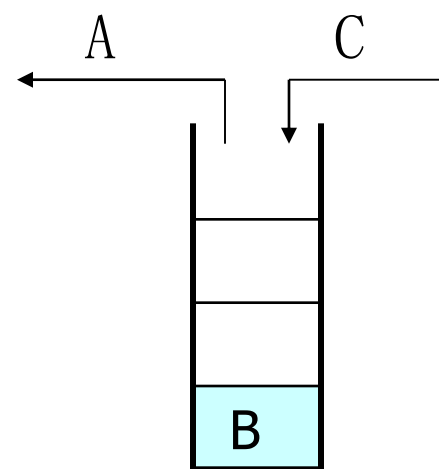
(1) 输入A, B, C, 产生输出A, B, C的过程:



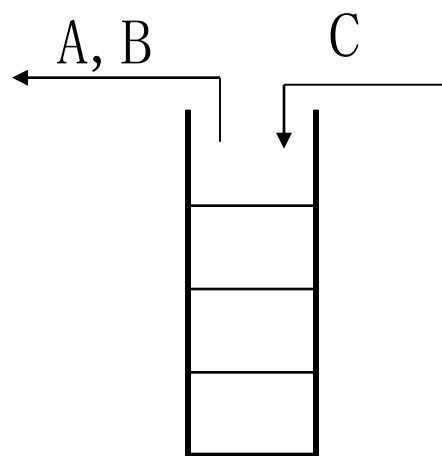
(1) A进栈



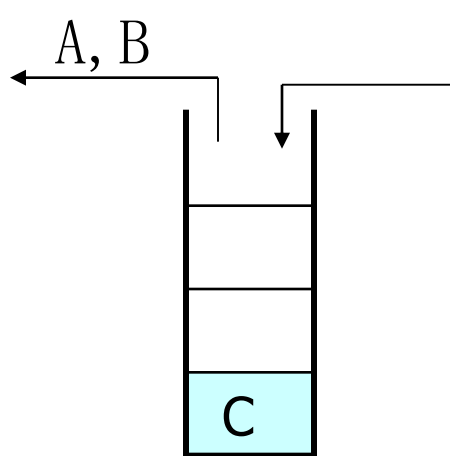
(2) A出栈



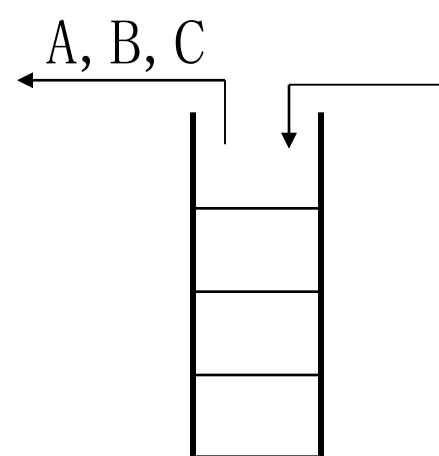
(3) B进栈



(4) B出栈



(5) C进栈

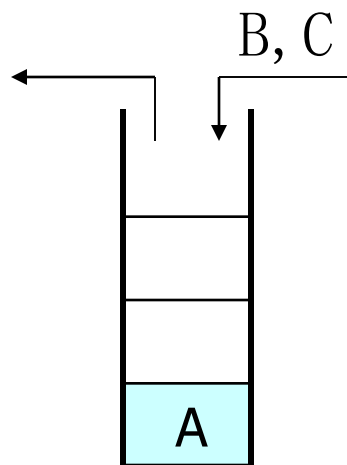


(6) C出栈

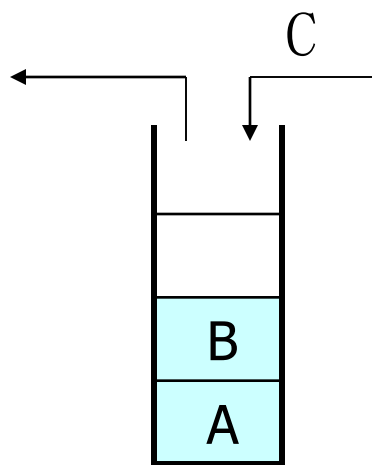




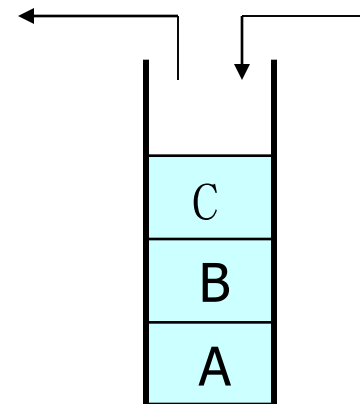
(2) 输入A, B, C, 产生输出C, B, A的过程:



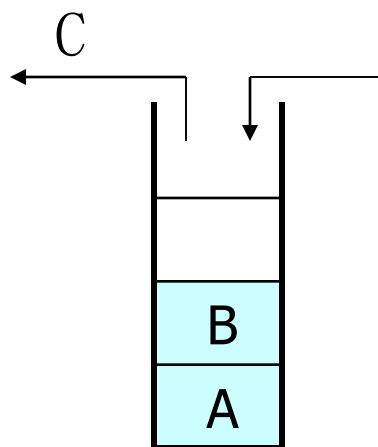
(1) A进栈



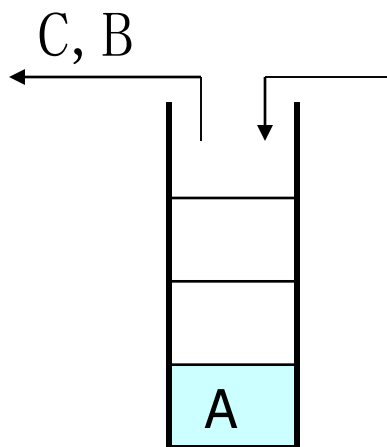
(2) B进栈



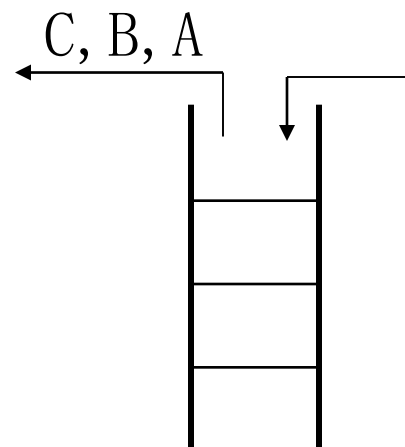
(3) C进栈



(4) C出栈



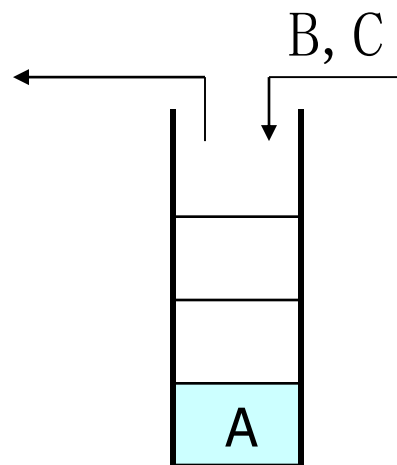
(5) B出栈



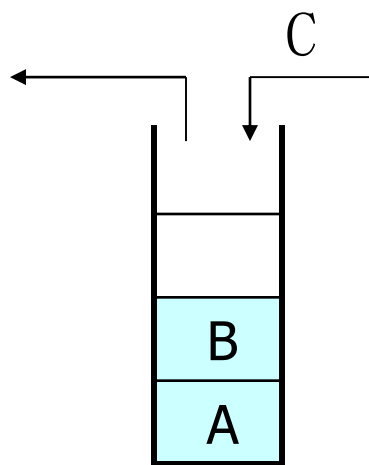
(6) A出栈



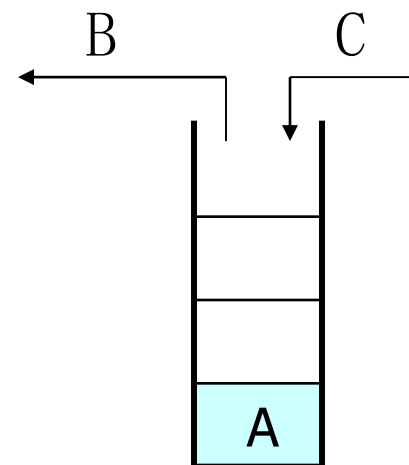
(3) 输入A, B, C, 产生输出B, C, A的过程:



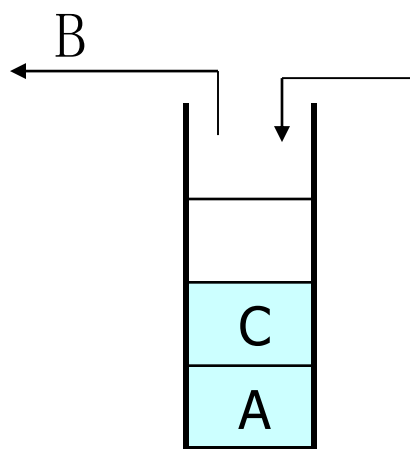
(1) A进栈



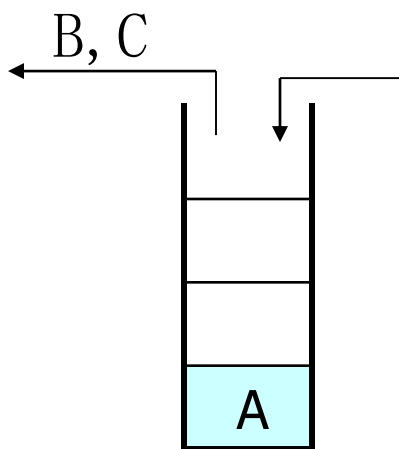
(2) B进栈



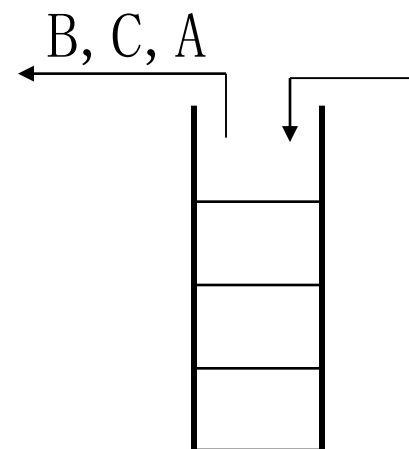
(3) B出栈



(4) C进栈



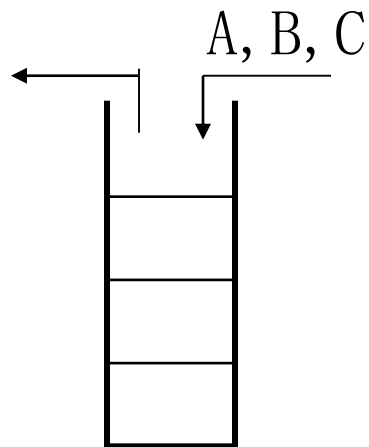
(5) C出栈



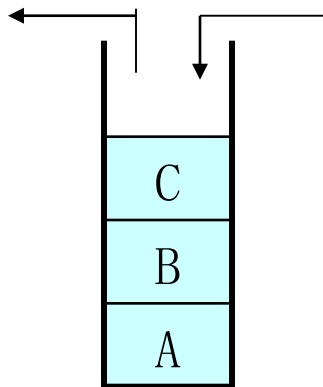
(6) A出栈



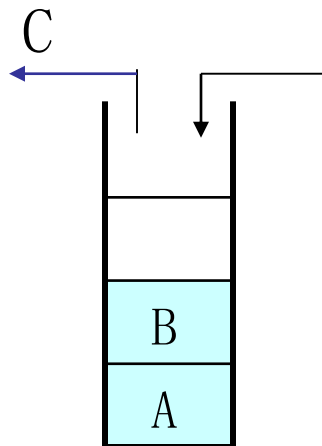
(4) 输入A, B, C, 不能产生输出C, A, B:



(1) 初始状态



(2) A, B, C进栈



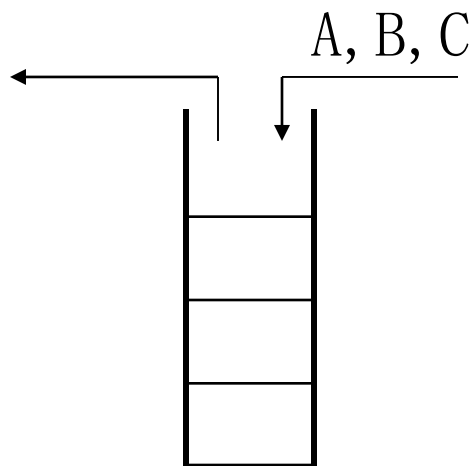
(3) C出栈

当A, B, C依次进栈, C出栈后, 由于栈顶元素是B, 栈底元素是A, 而A不能先于B出栈, 所以不能在输出序列中, 使A成为C的直接后继, 即不可能由输入A, B, C产生输出C, A, B。

一般地, 输入序列 $(\dots, a_i, \dots, a_j, \dots, a_k, \dots)$ 到栈中, 不能得到输出序列 $(\dots, a_k, \dots, a_i, \dots, a_j, \dots)$ 。

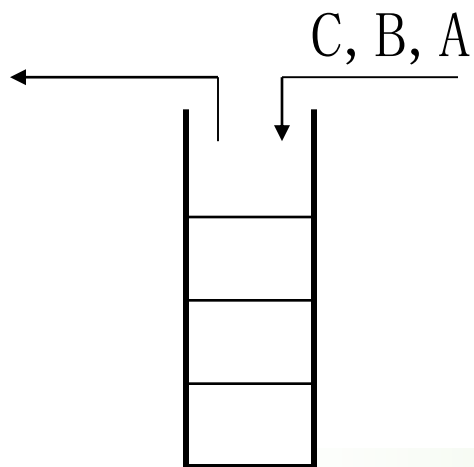


设依次输入元素A, B, C到栈中, 可得哪几种输出?



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

设依次输入元素C, B, A到栈中, 可得哪几种输出?



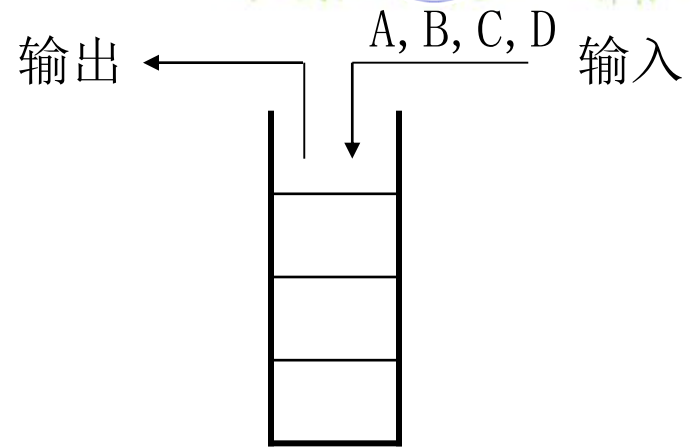
- (1) A, B, C
- (2) A, C, B
- (3) B, A, C
- (4) B, C, A
- (5) C, A, B
- (6) C, B, A





讨论:

假定输入元素 A, B, C, D  
到栈中, 能得当哪几种输出?  
不能得到哪几种输出序列?



- |                |                 |                 |                 |
|----------------|-----------------|-----------------|-----------------|
| (1) A, B, C, D | (7) B, A, C, D  | (13) C, A, B, D | (19) D, B, C, A |
| (2) A, B, D, C | (8) B, A, D, C  | (14) C, A, D, B | (20) D, B, A, C |
| (3) A, C, B, D | (9) B, C, A, D  | (15) C, B, A, D | (21) D, C, B, A |
| (4) A, C, D, B | (10) B, C, D, A | (16) C, B, D, A | (22) D, C, A, B |
| (5) A, D, B, C | (11) B, D, A, C | (17) C, D, A, B | (23) D, A, B, C |
| (6) A, D, C, B | (12) B, D, C, A | (18) C, D, B, A | (24) D, A, C, B |

5种

5种

3种

1种

共5+5+3+1=14种



## 3.1.2 栈的存储表示和操作实现

### 1. 顺序栈： 用顺序空间表示的栈。

设计实现方案时需要考虑的因素：

- 如何分配存储空间

动态分配或静态分配

栈空间范围，如： $s[0..maxleng-1]$

- 如何设置进栈和出栈的标志 $top$

如 $top$ 指向栈顶元素或指向栈顶元素上一空单元等，作为进栈与出栈的依据。

- 分析满栈的条件，用于进栈操作。

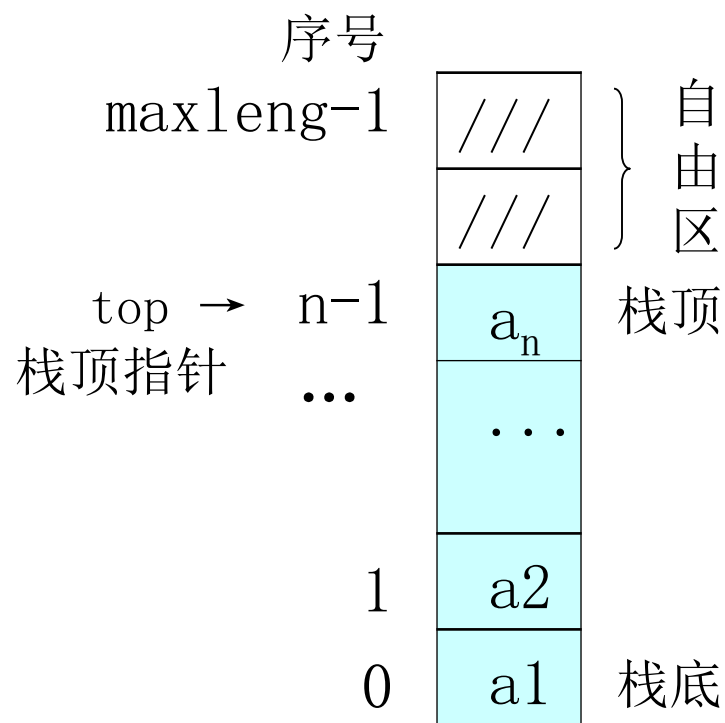
- 分析空栈的条件，用于出栈操作。





(1) 方案1: 栈空间范围为:  $s[0 \dots \text{maxleng}-1]$

顶指针指向顶元素所在位置:



(a) 非空栈示意图

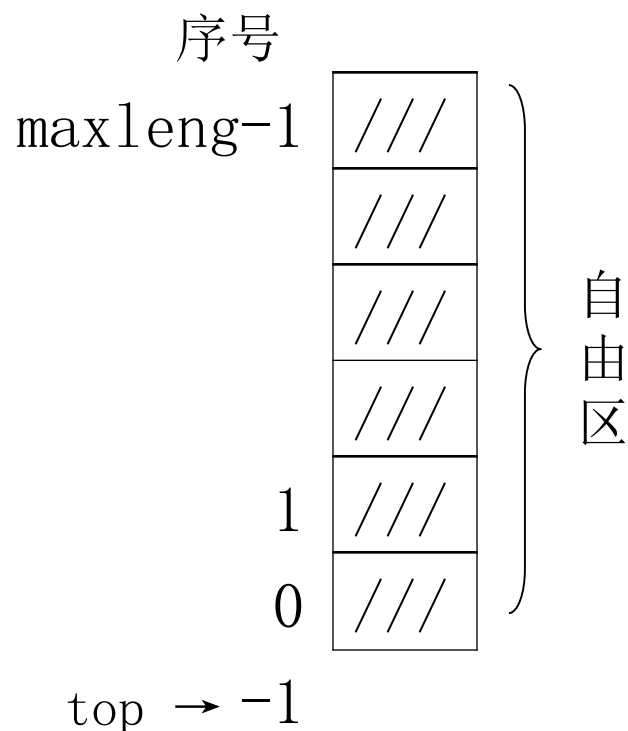
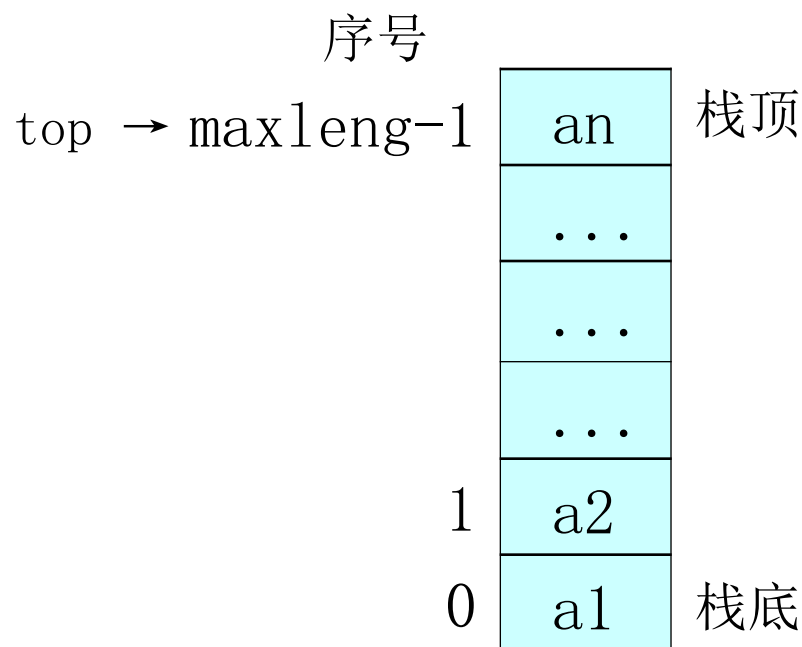
$\text{top} \geq 0$  顶元素 =  $s[\text{top}]$

进栈操作: 先对 $\text{top}$ 加1, 指向下一空位置, 将新数据送入 $\text{top}$ 指向的位置, 完成进栈操作。结束时 $\text{top}$ 指向新栈顶元素所在位置。

出栈操作: 先根据 $\text{top}$ 指向, 取出栈顶数据元素; 再对 $\text{top}$ 减1。完成出栈操作。结束时 $\text{top}$ 指向去掉原栈顶元素后的新栈顶元素所在位置。

(b) 进出栈说明





### (c) 满栈条件

$\text{top} == \text{maxleng}-1$  若插入元素，  
将发生“溢出” “Overflow”

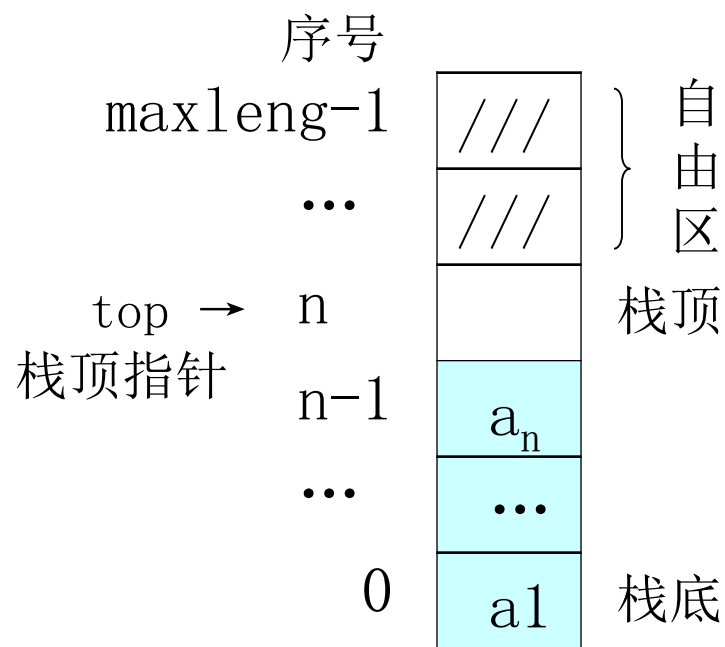
### (d) 空栈条件

$\text{top} == -1$  若删除元素，将发  
生“下溢”



(2) 方案2: 栈空间范围为:  $s[0 \dots \text{maxleng}-1]$

顶指针指向顶元素上的一空位置:



(a) 非空栈示意图

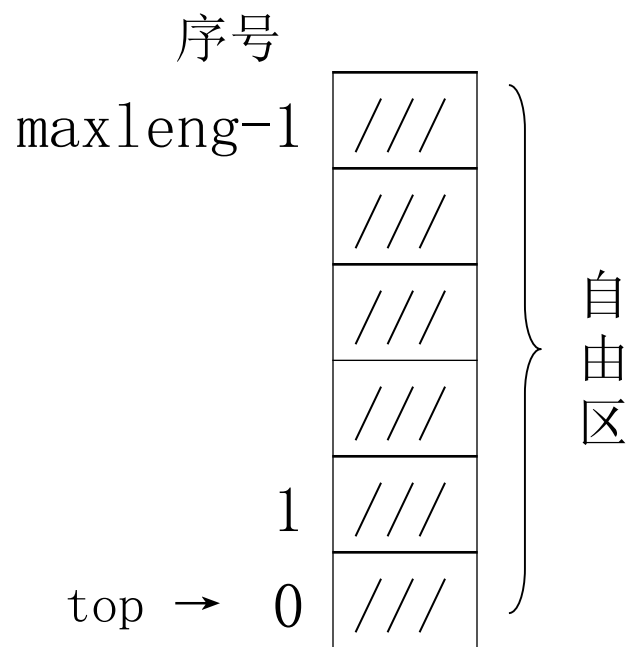
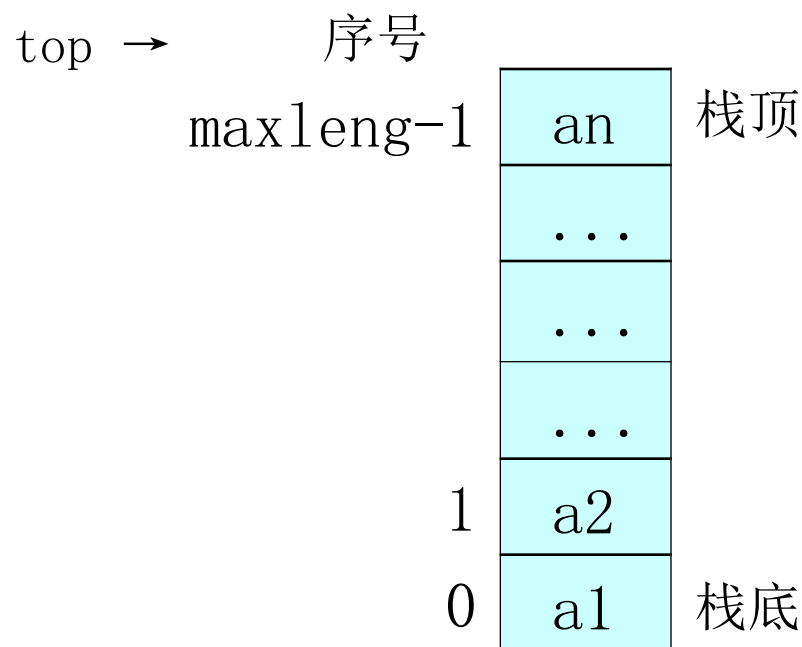
$\text{top} \geq 1$  顶元素 =  $s[\text{top}-1]$

进栈操作: 先将新数据送入  $\text{top}$  指向的位置, 再对  $\text{top}$  加1, 指向下一空位置, 完成进栈操作。结束时  $\text{top}$  正好指向新栈顶元素所在位置的上一空位置。

出栈操作: 先对  $\text{top}$  减1, 根据  $\text{top}$  指向取出栈顶数据元素。完成出栈操作。结束时  $\text{top}$  指向去掉原栈顶元素后的新栈顶元素所在位置的上一空位置。

(b) 进出栈说明





(c) 满栈条件:  $\text{top} == \text{maxleng}$   
若插入元素, 将发生“溢出”

(d) 空栈条件:  $\text{top} == 0$   
若删除元素, 将发生“下溢”



## 2. 顺序栈的描述

栈元素与顶指针合并定义为一个记录(结构)

约定：栈元素空间 $[0..maxleng-1]$

top指向栈元素上一空位置。

\*\* top是栈顶标志，根据约定由top找栈顶元素。

存储空间分配方案：

### (a) 静态分配

```
typedef struct
{ ElemType elem[maxleng];    //栈元素空间
  int top;                    //顶指针
} sqstack;                    //sqstack为结构类型
sqstack s;                    //s为结构类型变量
```

其中：s.top---顶指针；s.elem[s.top-1]---顶元素



## (b) 动态分配

```
#define STACK_INIT_SIZE  100
#define STACKINCREMENT   10
typedef struct
{ ElemType *base;          //指向栈元素空间
  int top;                  //顶指针
  int stacksize;           //当前分配的栈空间大小
} SqStack;                  // SqStack为结构类型
SqStack s;                  //s为结构类型变量
```

其中： s. top--顶指针； s. base[s. top-1]--顶元素





### 3. 顺序栈算法

#### (1) 初始化栈(动态分配)

```
void InitStack(SqStack &S)
{
    S.base=(ElemType *)malloc(STACK_INIT_SIZE*sizeof(ElemType));
    S.top=0;
    S.stacksize= STACK_INIT_SIZE;
}

void main(void)
{
    SqStack S1, S2;
    InitStack(S1);
    S2.base=(ElemType *)malloc(STACK_INIT_SIZE*sizeof(ElemType));
    S2.top=0;
    S2.stacksize= STACK_INIT_SIZE;
}
```



## (2) 进栈算法

```
int push(SqStack &S, ElemType x)
{   if (S.top >= S.stacksize)           //发生溢出，扩充
    {   newbase = (ElemType *)realloc(S.base,
        (S.stacksize + STACKINCREMENT) * sizeof(ElemType));
        if (!newbase) {
            printf("Overflow");
            return ERROR;
        }
        S.base = newbase;
        S.stacksize += STACKINCREMENT;
    }
    S.base[S.top] = x;                   //装入元素x
    S.top++;                             //修改顶指针
    return OK;
}
```



### (3) 出栈算法

```
int pop(SqStack &S, ElemType &x)
```

```
{ if (S.top==0)
```

```
    return ERROR;
```

//空栈

```
else
```

```
    { S.top--;
```

//修改顶指针

```
        x= S.base[S.top];
```

//取走栈顶元素

```
        return OK;
```

//成功退栈，返回OK

```
    }
```

```
}
```





```
main()  
{  
    SqStack    S;  
    ElemType   e;  
    InitStack(S);  
    push(S, 10);  
    if (push(S, 20) == ERROR)           //最好能判断其返回值,  
                                         //做出相应处理  
        printf(“进栈失败!”);  
  
    .....  
    if (pop(S, e) == OK)  
        {退栈成功, 处理e的值}  
    else {退栈失败, 提示错误信息}  
}
```



## 4. 链式栈:

### 使用不带表头结点的单链表

#### (1) 结点和指针的定义

```
struct node
```

```
{ ElemType data;
```

```
    struct node *next;
```

```
} *top=NULL;
```

//data为抽象元素类型

//next为指针类型

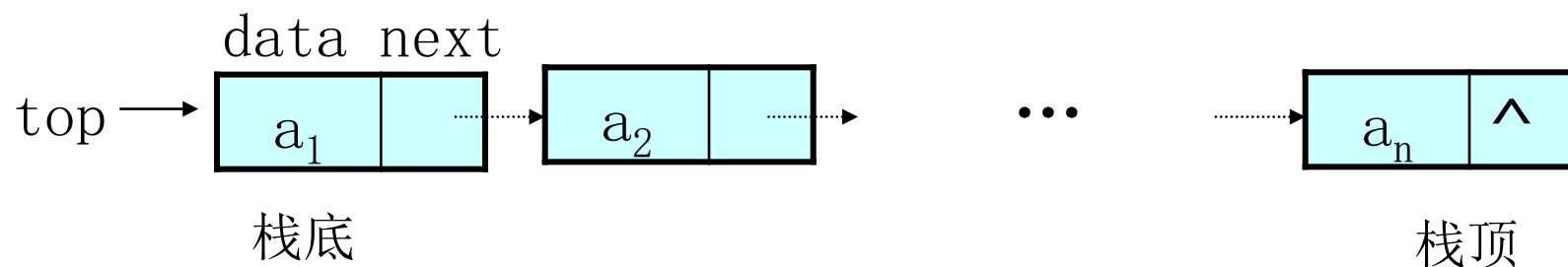
//初始化, 置top为空



## (2) 非空链式栈的一般形式

假定元素进栈次序为： $a_1$ 、 $a_2$ 、 $\dots a_n$ 。

用普通无头结点的单链表：



进栈需要找到最后一个结点。

出栈时删除最后一个结点。

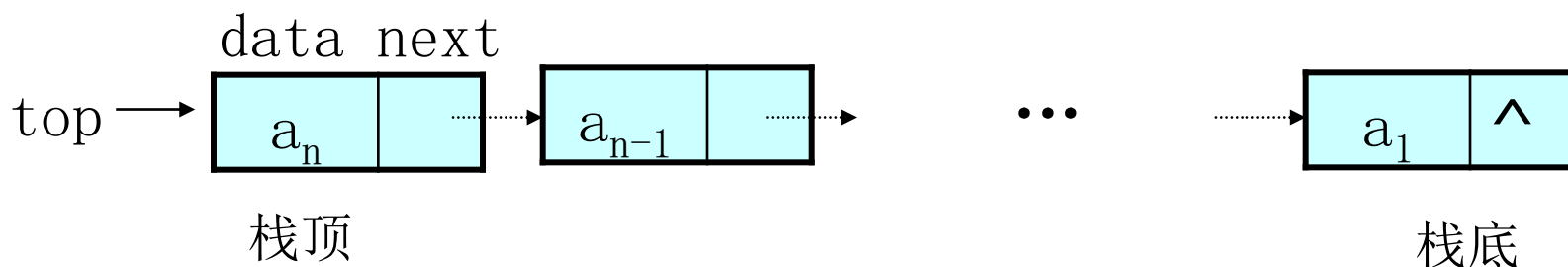
**缺点：** 进出栈时间开销大： $O(n)$





## (2) 非空链式栈的一般形式（续）

解决方案：将指针次序颠倒过来，top指向 $a_n$ 。



进栈将新结点作为首结点。

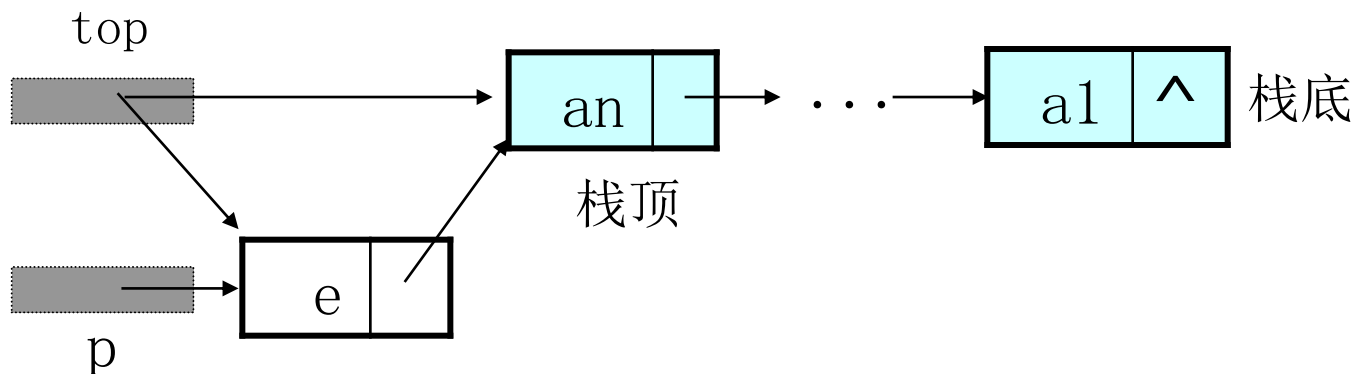
出栈时删除首结点。

**优点：**进出栈时间为常数:  $O(1)$



### (3) 链式栈的进栈:

压入元素e到top为顶指针的链式栈



```
p=(struct node *)malloc(sizeof(struct node));
```

```
p->data=e;
```

```
p->next=top;
```

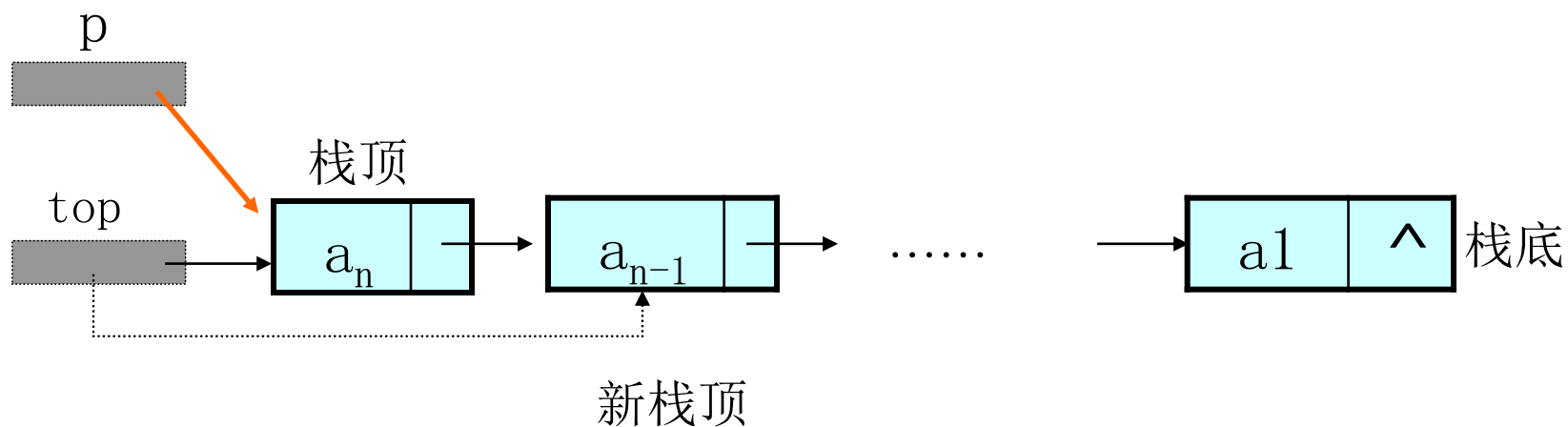
```
top=p;
```

## 进栈算法:

```
struct node *push_link(struct node *top, Elemtyp e)
{ struct node *p;
  int leng=sizeof(struct node); //确定新结点空间的大小
  p=(struct node *)malloc(leng); //生成新结点
  p->data=e;                      //装入元素e
  p->next=top;                    //插入新结点
  top=p;                         //top指向新结点
  return top;                    //返回指针top
}
```



## (4). 链式栈的退栈



```
p=top;
```

```
top=top->next;
```

```
free(p);
```



## 退栈算法

```
struct node *pop(struct node *top, Elemtyp *e)
{ struct node *p;
  if (top==NULL) return NULL;      //空栈, 返回NULL
  p=top;                          //p指向原栈的顶结点
  (*e)=p->data;                    //取出原栈的顶元素送 (*e)
  top=top->next;                   //删除原栈的顶结点
  free(p);                         //释放原栈顶结点的空间
  return top;                      //返回新的栈顶指针top
}
```



## 3.2 栈的应用举例

栈的基本用途——保存暂时不用的数或存储地址。

### 3.2.1 数制转换

例. 给定十进制数  $N=1348$ , 转换为八进制数  $R=2504$

1. 依次求余数, 并送入栈中, 直到商为0。

(1)  $r1=1348\%8=4$  //求余数

$n1=1348/8=168$  //求商

(2)  $r2=168\%8=0$  //求余数

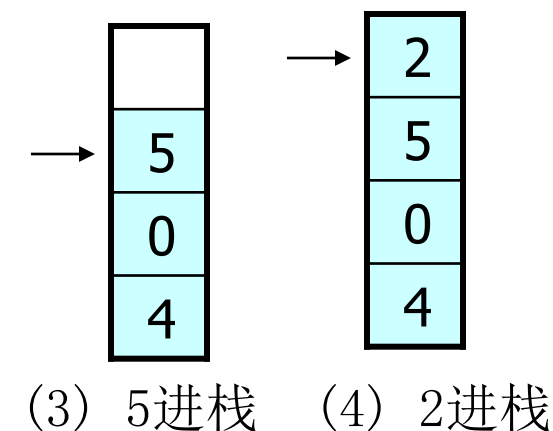
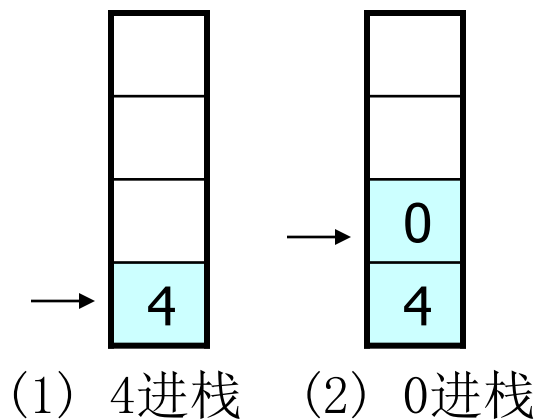
$n2=168/8=21$  //求商

(3)  $r3=21\%8=5$  //求余数

$n3=21/8=2$  //求商

(4)  $r4=2\%8=2$  //求余数

$n4=2/8=0$  //求商



2. 依次退栈, 得 $R=2504$







## 3.2.2 判定表达式中的括号匹配

### 1. 括号匹配的表达式

例.  $\{ \dots ( \dots ( ) \dots ) \dots \}$   
 $[ \dots \{ \dots ( ) \dots ( ) \dots \} \dots ]$

### 2. 括号不匹配的表达式

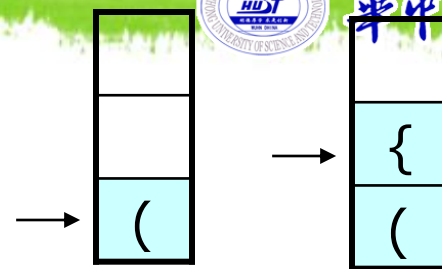
例.  $\{ \dots [ \} \dots ]$   
 $[ \dots ( \dots ( ) \dots ) \dots ]$



### 3. 判定括号不匹配的方法

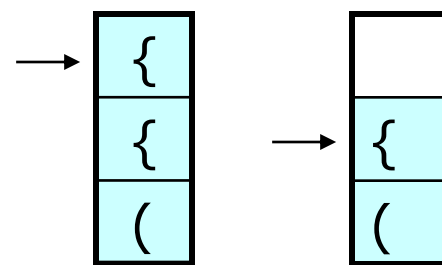
例.  $( \dots \{ \dots \{ \dots \} \dots ]$

$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$   
 (1)   (2)   (3)   (4)   (5)



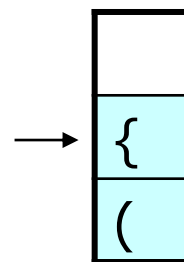
(1) “(” 进栈

(2) “{” 进栈



(3) “{” 进栈

(4) “{” 退栈



(5) “]” 与 “{” 不匹配



### 3.2.3 表达式求值

例:  $4 + 2 * 3 - 12 / (7 - 5)$

① ② ③ ④ ⑤

算符优先关系表

#### 求值规则:

1. 先乘除, 后加减;
2. 先括号内, 后括号外;
3. 同类运算, 从左至右。

#### 约定:

θ1----左算符

θ2----右算符

θ1=#, 为开始符

θ2=#, 为结束符

θ1 \ θ2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=



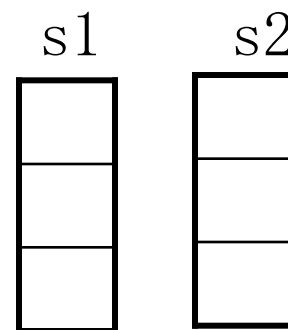


## 算法思想:

设: s1----操作数栈, 存放暂不运算的数和中间结果

s2----算符栈, 存放暂不运算的算符

1. 置s1, s2为空栈; 开始符#进s2;
2. 从表达式读取“单词”w----操作数/算符
3. 当 $w \neq \text{'\#'}$  || s2的顶算符 $\neq \text{'\#'}$  时, 重复:



3.1 若w为操作数, 则w进s1, 读取下一“单词”w;

3.2 若w为算符, 则:

3.2.1 若  $\text{prio}(\text{s2的顶算符}(\theta_1)) < \text{prio}(w(\theta_2))$ , 则:  
w进s2; 读取下一“单词”w;

3.2.2 若  $\text{prio}(\text{s2的顶算符}(\theta_1)) = \text{prio}(w(\theta_2))$ , 且 $w = \text{'\)'}$ , 则:  
去括号,  $\text{pop}(\text{s2})$ ; 读取下一“单词”w;

3.2.3 若  $\text{prio}(\text{s2的顶算符}(\theta_1)) > \text{prio}(w(\theta_2))$ , 则:  
{  $\text{pop}(\text{s1}, a)$ ;  $\text{pop}(\text{s1}, b)$ ;  $\text{pop}(\text{s2}, \text{op})$ ;  
   $c = b \text{ op } a$ ;  $\text{push}(\text{s1}, c)$ ;                   /\*op为 $\theta_1$ \*/  
}

4. s1的栈顶元素为表达式的值。





例. 求表达式的值:  $\# 4 + 2 * 3 - 12 / (7 - 5) \#$

步骤	操作数栈	运算符栈	输入串	下步操作说明
0		#	4+2*3-12/(7-5)#	操作数进栈
1	4	#	+2*3-12/(7-5)#	$p(\#) < p(+)$ , 进栈
2	4	# +	2*3-12/(7-5)#	操作数进栈
3	42	# +	*3-12/(7-5)#	$p(+)<p(*)$ , 进栈
4	42	# + *	3-12/(7-5)#	操作数进栈
5	423	# + *	-12/(7-5)#	$p(*)>p(-)$ , 退栈 $op=*$
6	423	# +	-12/(7-5)#	操作数退栈 $b=3$
7	42	# +	-12/(7-5)#	操作数退栈 $a=2$
8	4	# +	-12/(7-5)#	$a*b$ 得 $c=6$ 进栈





步骤	操作数栈	运算符栈	输入串	下步操作说明
8	4	# +	-12 / (7-5) #	a*b得6进栈
9	46	# +	-12 / (7-5) #	p(+)>p(-),退栈op=+
10	46	#	-12 / (7-5) #	操作数退栈b=6
11	4	#	-12 / (7-5) #	操作数退栈a=4
12		#	-12 / (7-5) #	a+b得c=10进栈
13	10	#	-12 / (7-5) #	p(#)<p(-),进栈
14	10	# -	12 / (7-5) #	操作数进栈
15	1012	# -	/ (7-5) #	p(-)<p(/),进栈
16	1012	# - /	(7-5) #	p(/)<p(,),进栈
17	1012	# - / (	7-5) #	操作数进栈





步骤	操作数栈	运算符栈	输入串	下步操作说明
17	1012	# - / (	7-5) #	操作数进栈
18	10127	# - / (	-5) #	$p(()) < p(-)$ , 进栈
19	10127	# - / ( -	5) #	操作数进栈
20	101275	# - / ( -	) #	$p(-) > p())$ , 退栈 $op = -$
21	101275	# - / (	) #	操作数退栈 $b = 5$
22	10127	# - / (	) #	操作数退栈 $a = 7$
23	1012	# - / (	) #	$a - b$ 得 $c = 2$ 进栈
24	10122	# - / (	) #	$p(()) = p())$ , 去括号
25	10122	# - /	#	$p(/) > p(\#)$ , 退栈 $op = /$
26	10122	# -	#	操作数退栈 $b = 2$





步骤	操作数栈	运算符栈	输入串	下步操作说明
26	10122	# -	#	操作数退栈b=2
27	1012	# -	#	操作数退栈a=12
28	10	# -	#	a/b得c=6进栈
29	106	# -	#	p(-)>p(#),退栈op=-
30	106	#	#	操作数退栈b=6
31	10	#	#	操作数退栈a=10
32		#	#	a-b得c=4进栈
33	4	#	#	p(#)=p(#),算法结束

表达式的值





## 包含min函数的栈

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是  $O(1)$

```
MinStack minStack = new MinStack();  
minStack.push(-2);  
minStack.push(0);  
minStack.push(-3);  
minStack.min(); --> 返回 -3.  
minStack.pop();  
minStack.top(); --> 返回 0.  
minStack.min(); --> 返回 -2.
```

```
1 typedef struct {  
2  
3 } MinStack;  
4  
5 /** initialize your data structure here. */  
6  
7 MinStack* minStackCreate() {  
8  
9 }  
10  
11 void minStackPush(MinStack* obj, int x) {  
12  
13 }  
14  
15 void minStackPop(MinStack* obj) {  
16  
17 }  
18  
19 int minStackTop(MinStack* obj) {  
20  
21 }  
22  
23 int minStackMin(MinStack* obj) {  
24  
25 }  
26  
27 void minStackFree(MinStack* obj) {  
28  
29 }  
30  
31 /**  
32 * Your MinStack struct will be instantiated and called as such:  
33 * MinStack* obj = minStackCreate();  
34 * minStackPush(obj, x);  
35  
36 * minStackPop(obj);  
37  
38 * int param_3 = minStackTop(obj);  
39  
40 * int param_4 = minStackMin(obj);  
41  
42 * minStackFree(obj);  
43 */
```





## 3.4 队列（排队, queue）

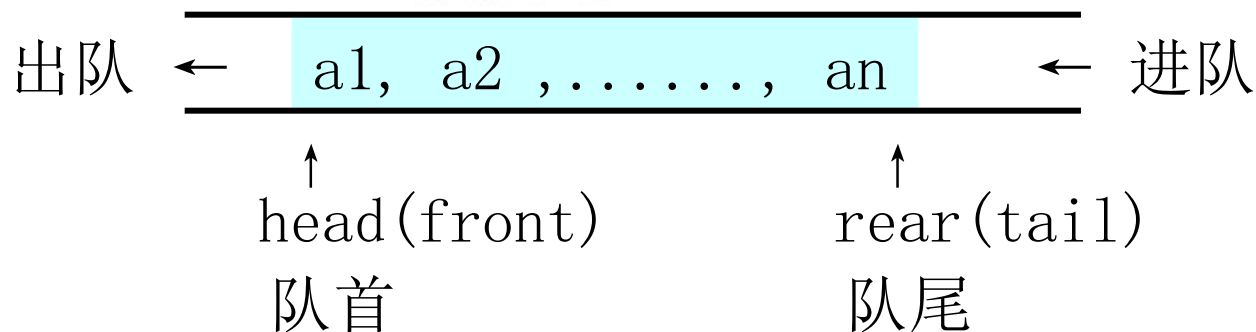
### 3.4.1 队列及其操作

#### 1. 定义和术语

- 队列——只允许在表的一端删除元素, 在另一端插入元素的线性表。
- 空队列——不含元素的队列。
- 队首——队列中只允许删除元素的一端。head, front
- 队尾——队列中只允许插入元素的一端。rear, tail
- 队首元素——处于队首的元素。
- 队尾元素——处于队尾的元素。
- 进队——插入一个元素到队列中。又称：入队。
- 出队——从队列删除一个元素。

2. 元素的进出原则：“先进先出”，“First In First Out”  
队列的别名：“先进先出”表，“FIFO”表，排队, queue





队列示意图

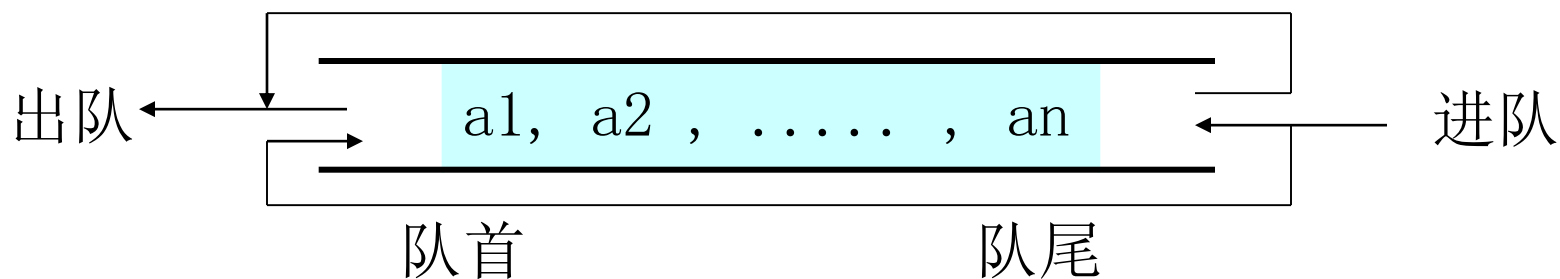
### 3. 队列的基本操作:

- (1) InitQueue(q) ---- 初始化, 构造一个空队列q。
- (2) QueueEmpty(q) ---- 判断q是否为空队列。
- (3) EnQueue(q, e) ---- 将e插入队列q的尾端。
- (4) DeQueue(q, e) ---- 取走队列q的首元素, 送e。
- (5) GetHead(q, e) ---- 读取队列q的首元素, 送e。
- (6) QueueClear(q) ---- 置q为空队列。



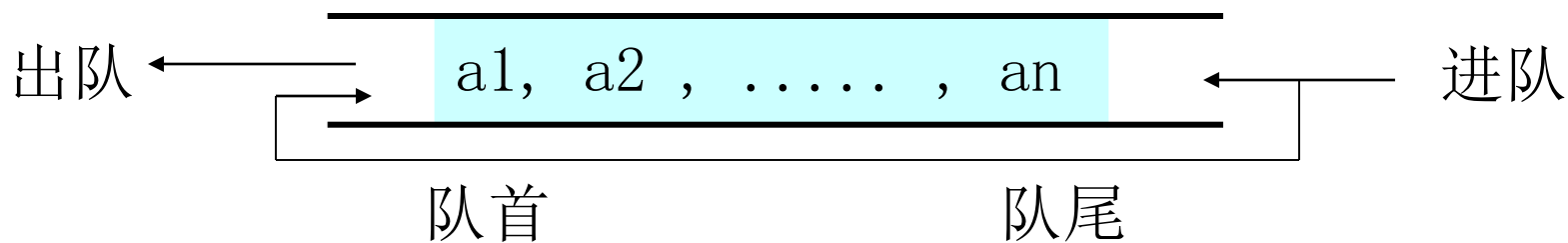
#### 4. 双队列（双端队列，deque——double ended queue）

(1) 双队列——允许在表的两端插入、删除元素的线性表。



双队列示意图

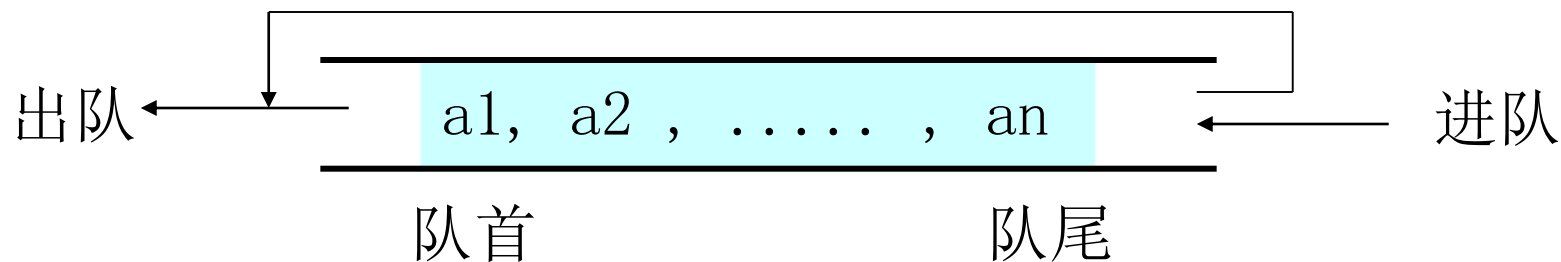
(2) 输出受限双队列——只许在表的两端插入、在一端删除元素的线性表。



输出受限双队列示意图



(3) 输入受限双队列——只允许在表的一端插入、在两端删除元素的线性表。



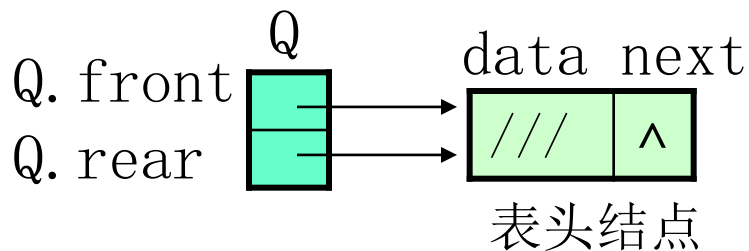
输入受限双队列示意图



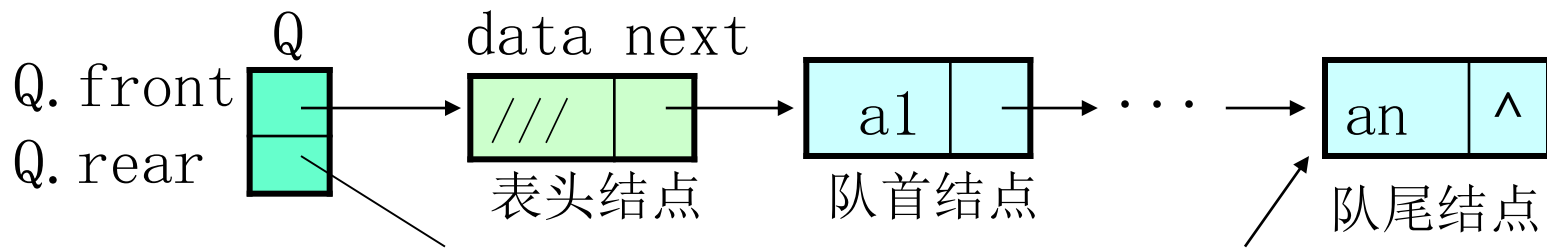
## 3.4.2 链式队列：用带表头结点的单链表表示队列

### 1. 一般形式

#### (1) 空队列：



#### (2) 非空队列：



其中： Q. front——队头(首)指针，指向表头结点。

Q. rear——队尾指针，指向队尾结点。

Q. front->data 不放元素。

Q. front->next 指向队首结点a1。



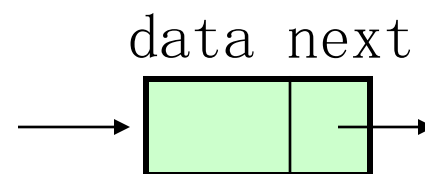
## 2. 定义结点类型

### (1) 存放元素的结点类型

```
typedef struct Qnode
{ ElemType data;           //data为抽象元素类型
  struct Qnode *next;      //next为指针类型
} Qnode, *QueuePtr;        //结点类型, 指针类型
```

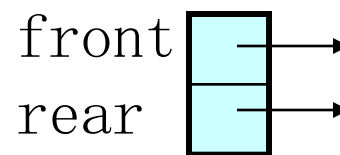
其中: Qnode----结点类型

QueuePtr----指向Qnode的指针类型



### (2) 由头、尾指针组成的结点类型

```
typedef struct
{ Qnode *front; //头指针
  Qnode *rear;  //尾指针
} LinkQueue;    //链式队列类型
```



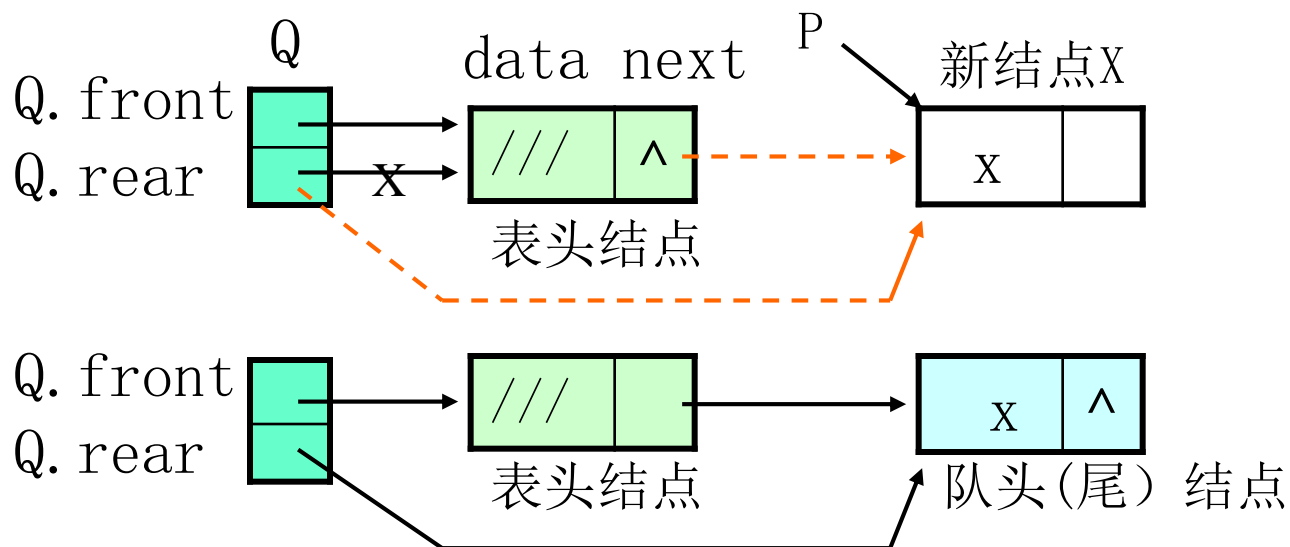
### 3. 生成空队列算法

```
#define LENG sizeof(Qnode)    //求结点所占的单元数
LinkQueue InitQueue( )        //生成仅带表头结点的空队列Q
{ LinkQueue Q;                //说明变量Q
  Q.front=Q.rear=(QueuePtr)malloc(LENG); //生成表头结点
  Q.front->next=NULL;          //表头结点的next为空指针
  return Q;                    //返回Q的值
}
```

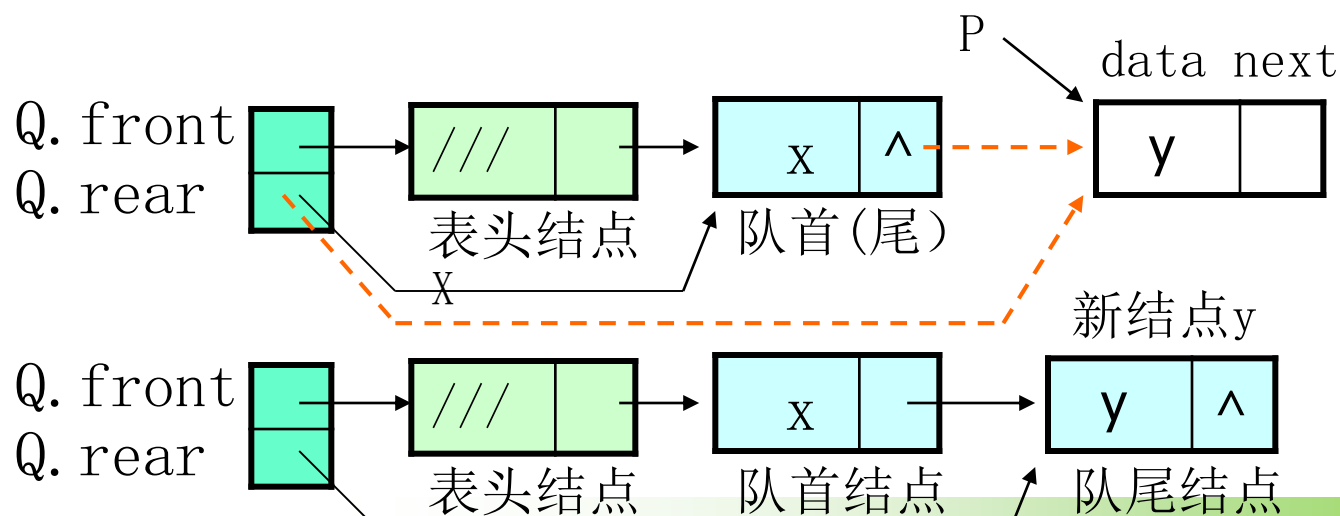
```
main()
{
  LinkQueue que;               /*定义一个队列*/
  que=InitQueue();             /*初始化队列*/
  .....
}
```



## 4. (空队列时) 插入新元素x



## (非空队列时) 插入新元素y





## 插入新元素e的的算法(1)

```
LinkQueue EnQueue(LinkQueue Q, ElemType e)
{ Qnode *p;                //说明变量p
  p=(Qnode *)malloc (LENG); //生成新元素结点
  p->data=e;                //装入元素e
  p->next=NULL;             //为队尾结点
  Q.rear->next=p;           //插入新结点
  Q.rear=p;                 //修改尾指针
  return Q;                 //返回Q的新值
}
```

```
main()
```

```
{
  LinkQueue que;            /*定义一个队列*/
  que=InitQueue();          /*初始化队列*/
  que=EnQueue(que, 10);     /*插入新元素*/
}
```



## 插入新元素e的的算法(2)

```
int EnQueue(LinkQueue *Q, ElemType e)
{ Qnode *p;                                //说明变量p
  p=(Qnode *)malloc (LENG);                //生成新元素结点
  if (!p) {printf(“OVERFLOW”);            //新结点生成失败
    return ERROR;}

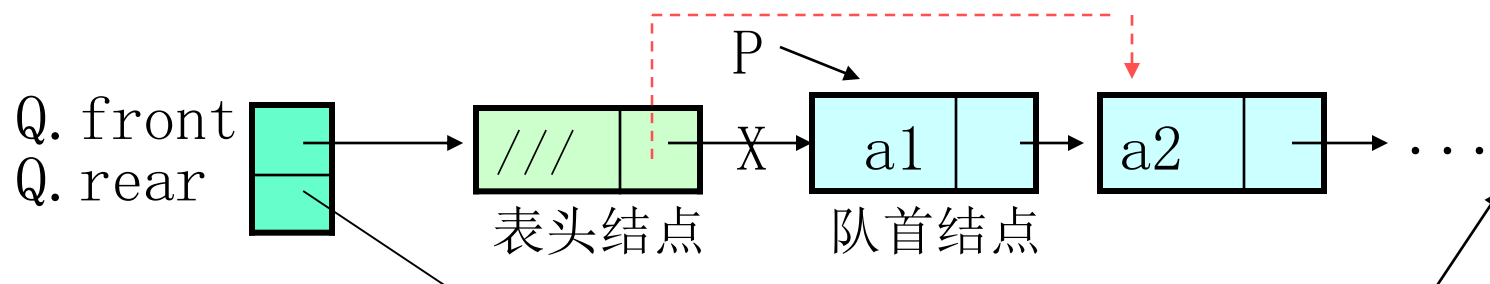
  p->data=e;                                //装入元素e
  p->next=NULL;                             //为队尾结点
  Q->rear->next=p;                          //插入新结点
  Q->rear=p;                                //修改尾指针
  return OK;                                //成功返回
}

main()
{ LinkQueue que;                          /*定义一个队列*/
  que=InitQueue();                        /*初始化队列*/
  EnQueue(&que, 10);                      /*插入新元素*/
}
```

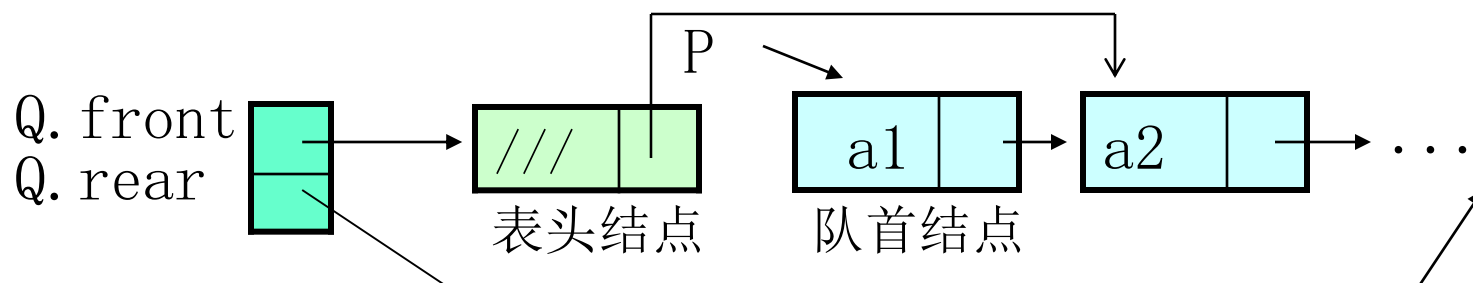


## 5. 出队——删除队头结点

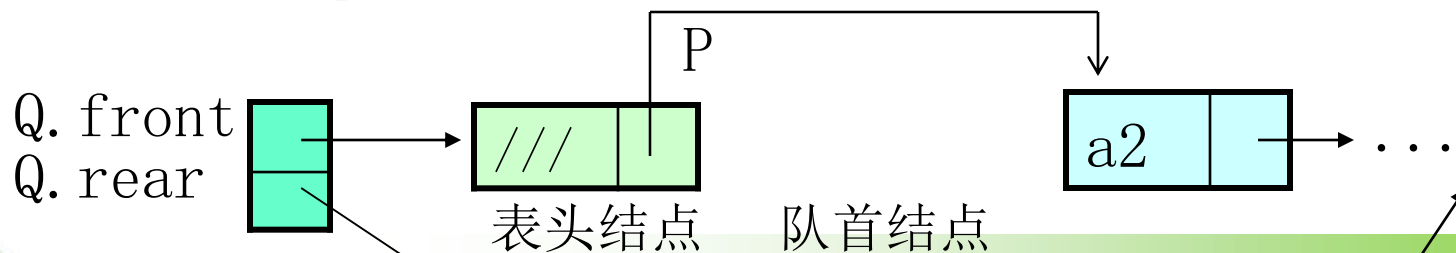
(1) 若原队列有2个或2个以上的结点



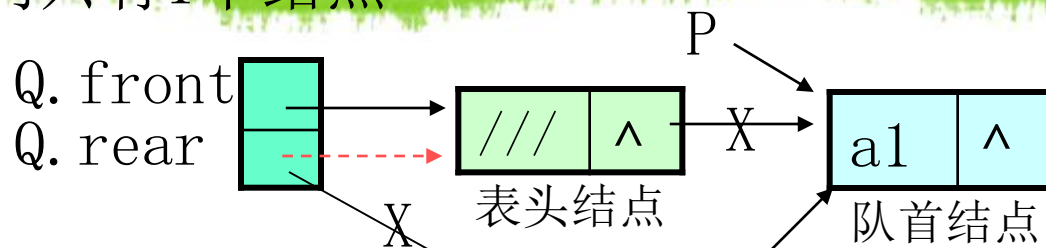
(a) 执行:  $Q.front \rightarrow next = p \rightarrow next$ ;



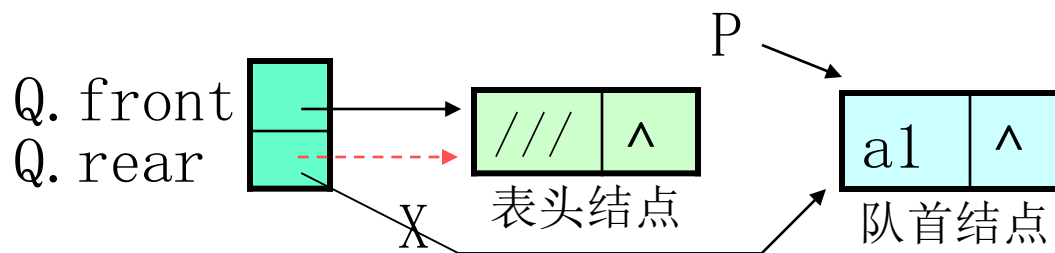
(b) 执行:  $free(p)$ ;



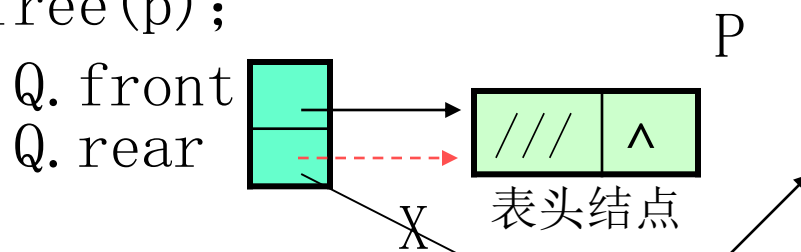
## (2) 若原队列只有1个结点



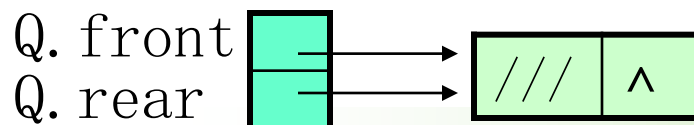
(a) 执行:  $Q.front \rightarrow next = p \rightarrow next;$



(b) 执行:  $free(p);$



(c) 执行:  $Q.rear = Q.front;$



## 出队算法:

```
Status DelQueue(LinkQueue &Q, ElemType &e)
{ Qnode *p;                                //说明变量p
  if (Q.front==Q.rear)                      //若原队列为空
    {printf( "Empty queue" );              //空队列
      return Q; }
  p = Q.front->next;                        //P指向队首结点
  e = p->data;                              //取出元素, e指向它
  Q.front->next=p->next;                    //删除队首结点
  if (Q.rear==p)                           //若原队列只有1个结点
    Q.rear=Q.front;                        //修改尾指针
  free(p);                                 //释放被删除结点的空间
  return OK;
}
```





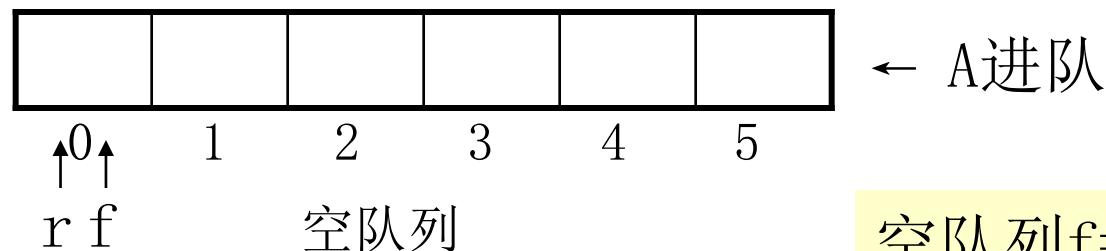
### 3.4.3 队列的顺序表示和实现

假设用一维数组 $Q[0..5]$ 表示顺序队列

#### 1. 顺序队列与“假溢出”

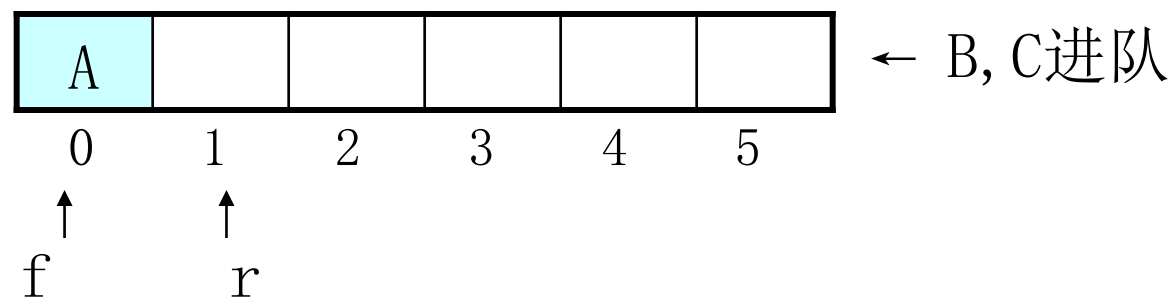
设 $f$ 指向队头元素， $r$ 指向队尾元素后一空单元

(1) 初始化后:

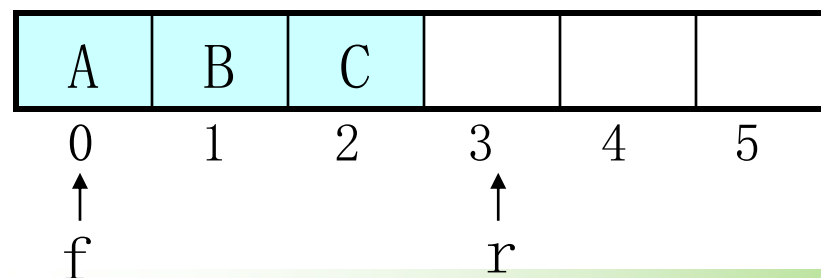


空队列 $f=r$

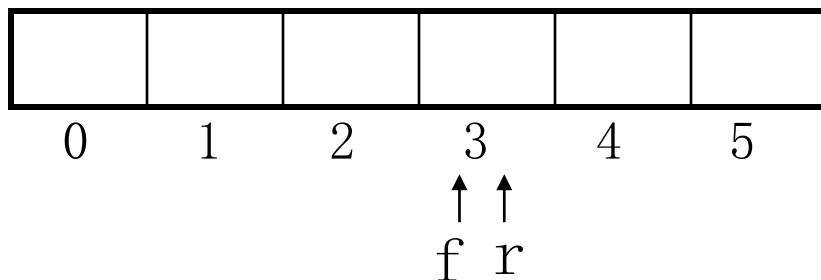
(2) A进队后:



(3) B, C进队后:



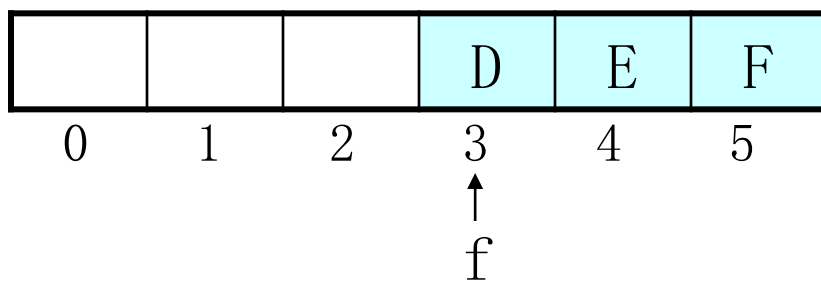
(4) A, B, C 出队之后:



← D, E, F 进队

此时空队列  $f=r$

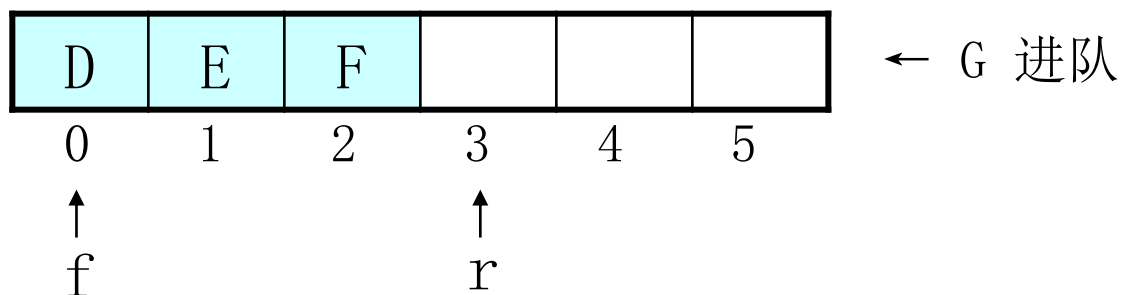
(5) D, E, F 依次进队之后:



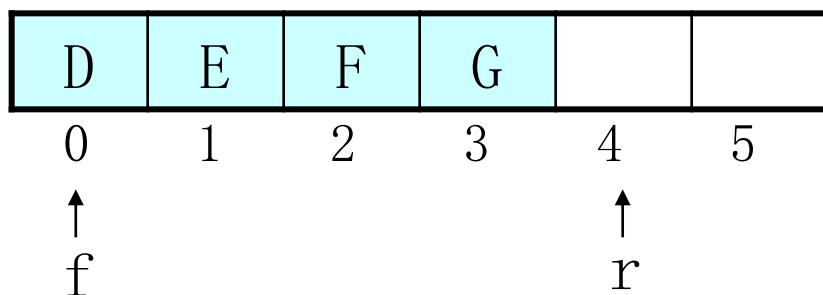
← G进队, “假溢出

解决假溢出的方法一： 移动元素。

(6) D, E, F移到前端后：

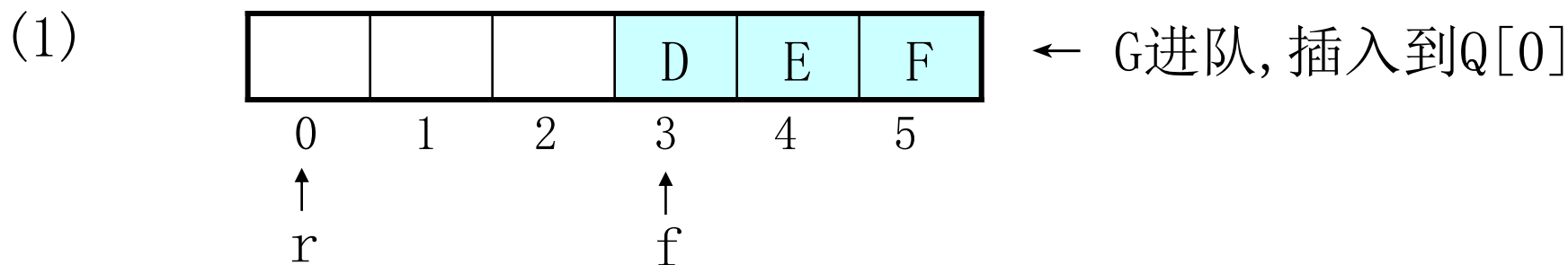


(7) G进队之后：

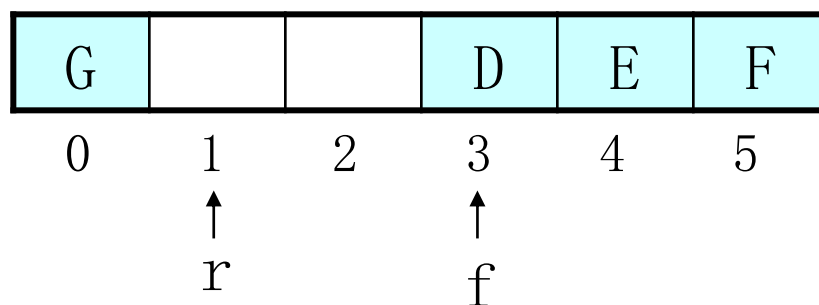


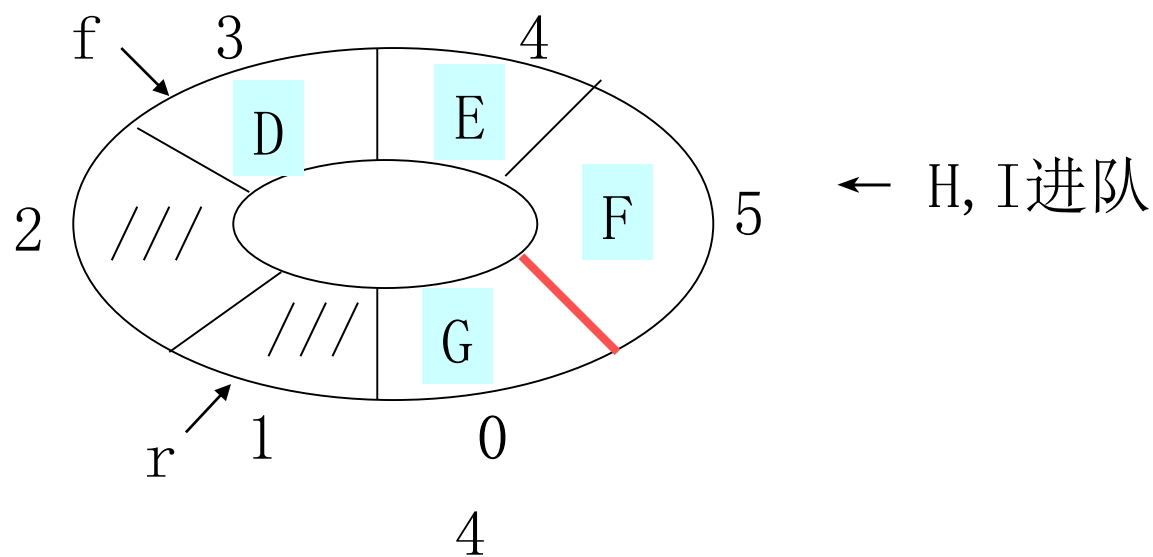


2. 解决假溢出的方法二： 将Q当循环表使用(循环队列)：



(2) G进Q[0]之后：

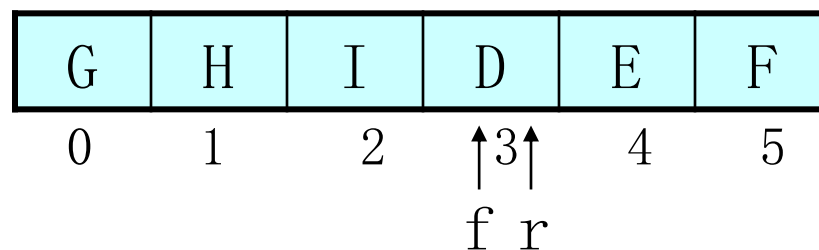
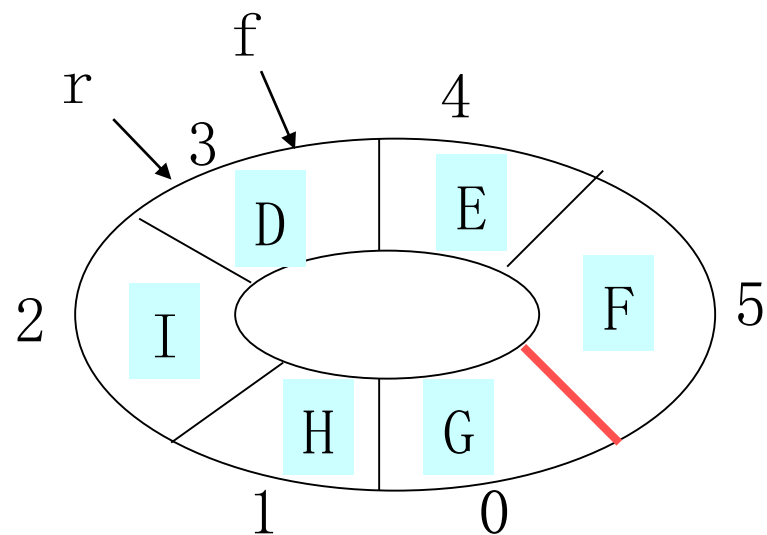




将 $Q[0..5]$ 解释为循环队列的示意图

(3) H, I进队之后

“满队列:



$f=r$ 时  
空队列?  
满队列?

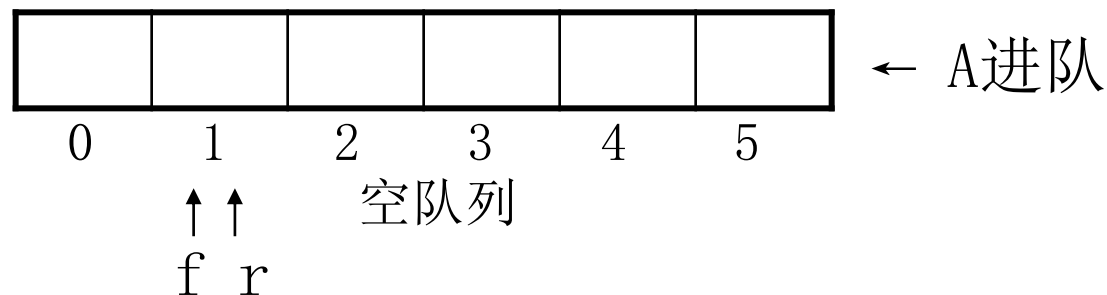


### 3. 方法二的实现方法:

设 $f$ 指向队头元素;  $r$ 指向队尾元素后一空单元。 $Q[0..5]$ 为循环队列。

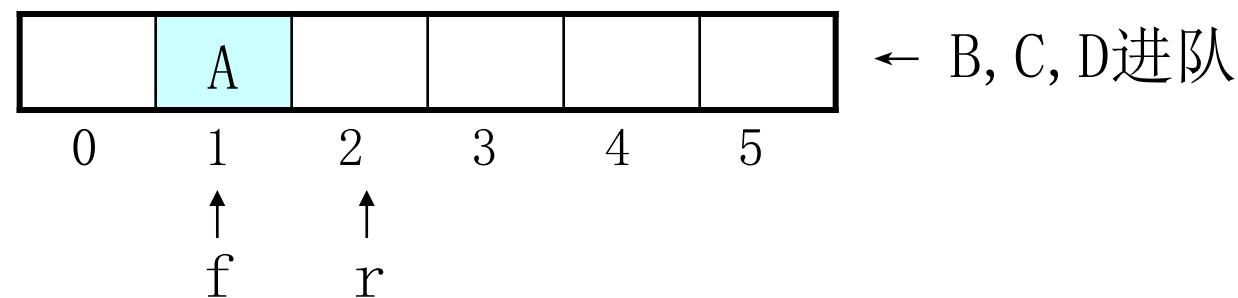
#### (1) 初始化

$f=r=1$ ;  
(只要在0到5的  
范围内相等即可)



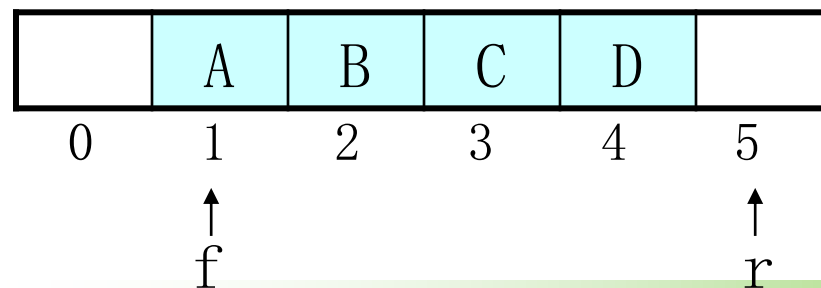
#### (2) A进队

$Q[r]=A$ ;  
 $++r$ ;



#### (3) B, C, D进队

$Q[r++] = B$ ;  
 $Q[r++] = C$ ;  
 $Q[r++] = D$ ;

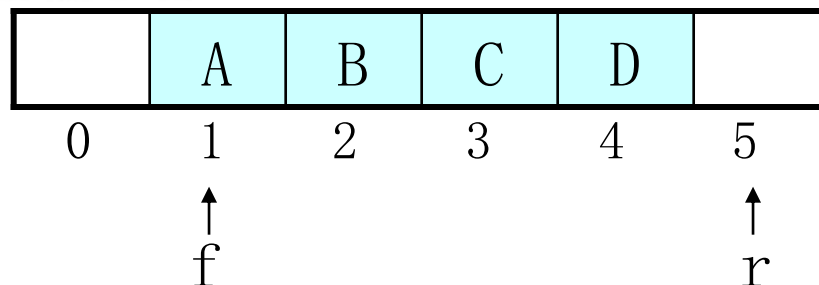


(3) B, C, D进队

$Q[r++] = B;$

$Q[r++] = C;$

$Q[r++] = D;$

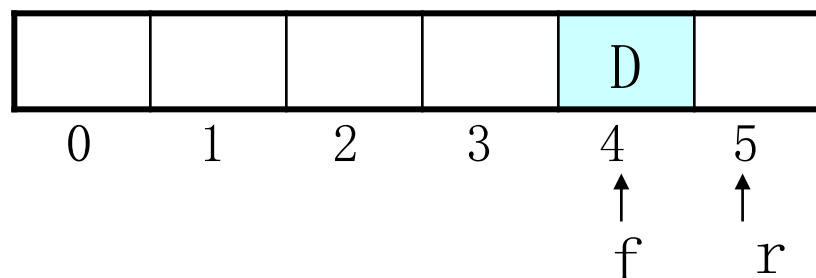


(4) A, B, C出队

$e1 = Q[f++];$

$e2 = Q[f++];$

$e3 = Q[f++];$

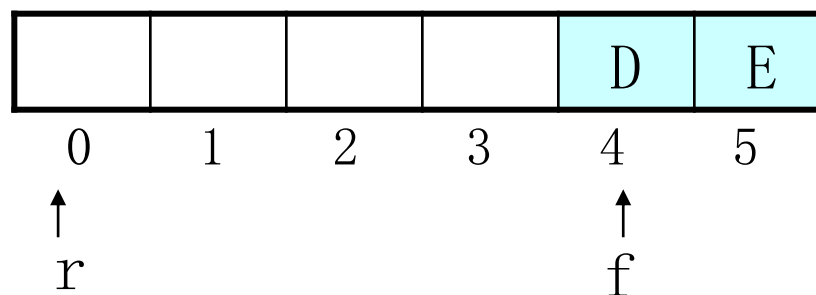


← E进队

(5) E进队

$Q[r] = E;$

$r = (r+1) \% 6$

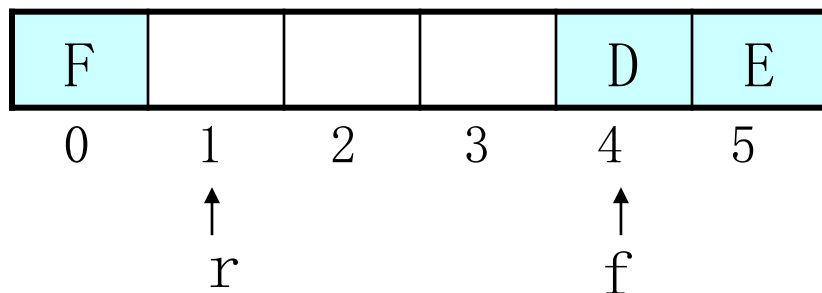


← F进队



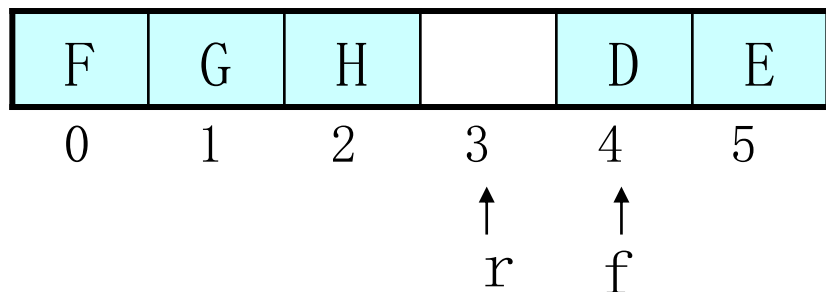
(6) F进队

$Q[r++] = F;$



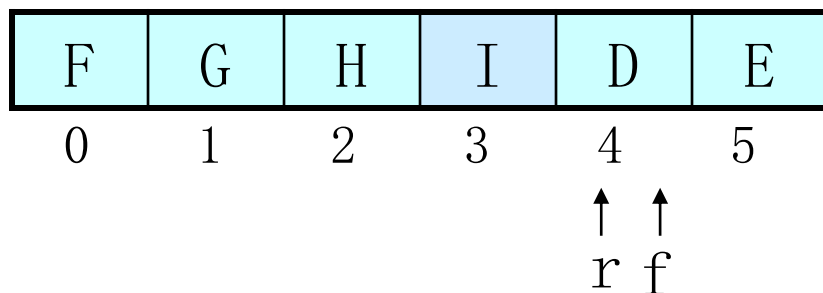
← G, H进队

(7) G, H进队后:



← I进队

(8) I进队后，导致 $r=f$ ，产生二义性。



## 解决方案:

- 1、方案一：增加一标识变量。
- 2、方案二：还剩最后一个单元不使用，可避免满队列时出现的二义性，即：进队前测试：若 $r+1==f$ ，表明还剩最后一个单元，认为此时就是满队列。

若队列为 $Q[0..maxleng-1]$ ，则共有 $maxleng-1$ 个元素



方案二的空，满队列条件：

(1) 满队列条件：

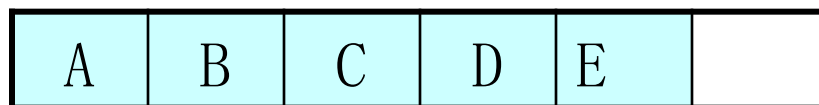
若A, B, C, D, E  
依次进队后：



0 1 2 3 4 5

↑ ↑  
r f

← G, H进队



0 1 2 3 4 5

↑ ↑  
f r

当  $r+1==f$  或  $(f==0) \&\& (r==\text{maxleng}-1)$

即：  $(r+1) \% \text{maxleng} == f$  为满队列

(2) 空队列条件：

A, B, C, D, E

依次出队后：



0 1 2 3 4 5

↑ ↑  
f r

当  $(f==r)$  为空队列



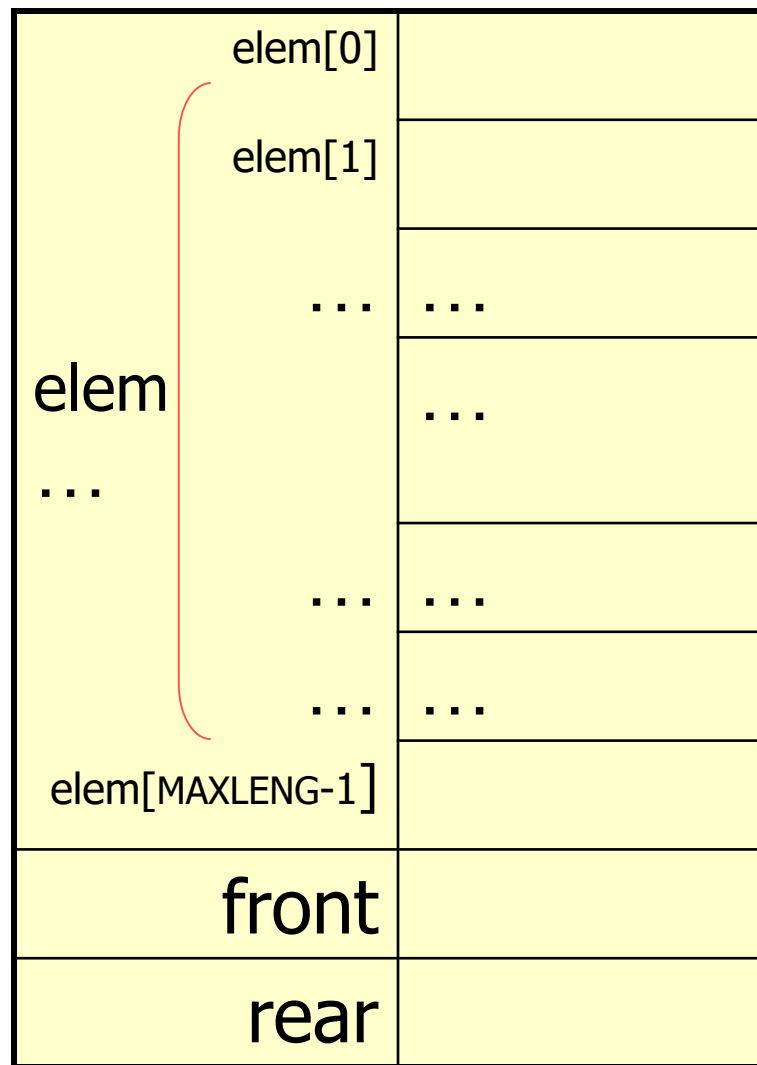


## 4. 顺序队列算法举例

定义队列的C类型

```
#define MAXLENG 100
Typedef struct
{
    ElemType elem[MAXLENG];
    int front, rear;
} SeQueue;
```

```
//定义结构变量Q表示队列
SeQueue Q;
```



队列Q的存储结构示意图

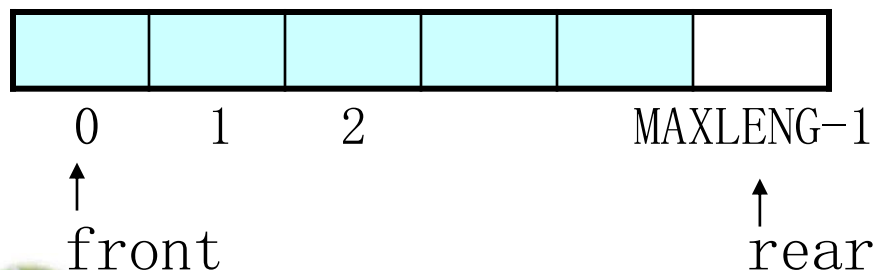


## (1) 进队算法:

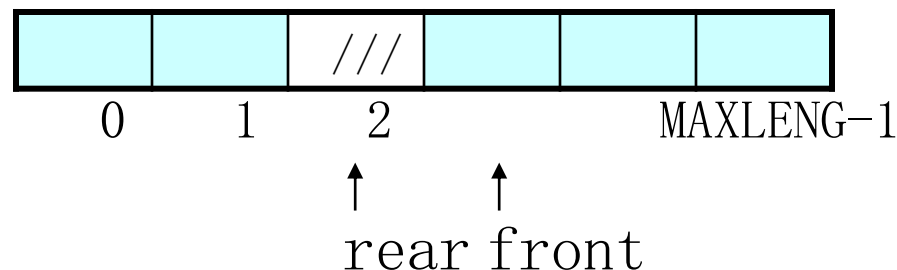
假设用Q表示顺序队列, 头指针front指向队头元素, rear指向尾元素的后一个空位, e为进队元素。

```
int En_Queue( SeQueue &Q, Elemtype e)
{ if ((Q.rear+1)% MAXLENG==Q.front)    //若Q已满, 退出
    return ERROR;
    Q.elem[Q.rear]=e;                    //装入新元素e
    Q.rear++;                            //尾指针后移一个位置
    Q.rear = Q.rear % MAXLENG;          //为循环队列
    return OK;
}
```

Q[0..MAXLENG-1] 已满



Q[0..MAXLENG-1] 已满

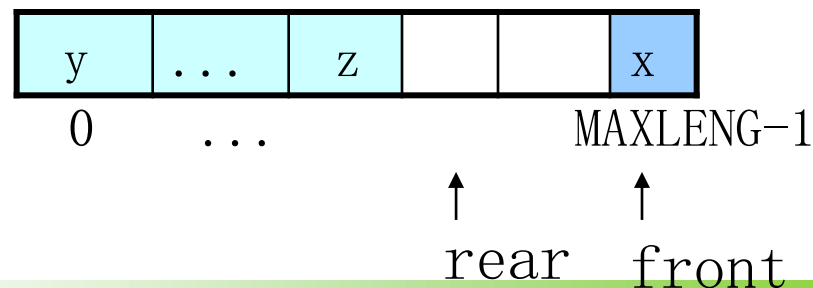
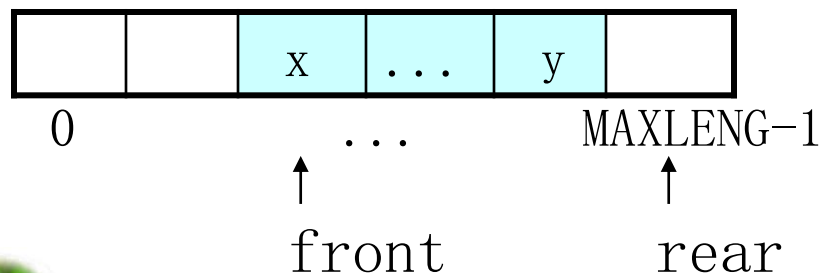


## (2) 出队算法

```
int De_Queue (SeQueue &Q, Elemtyp e &e)
{
    if (Q.front==Q.rear)    //Q为空队列，退出
        return ERROR;
    e=Q.elem[Q.front];    //取走队首元素，送e

    Q.front=(Q.front+1)% MAXLENG; //循环后移到一个位置

    return OK;
}
```





## 思考题

1. 如何根据front, rear, maxlength求队列的长度。

$$(Q.rear - Q.front + maxlength) \% maxlength$$

