

---

# 第3章 栈和队列

苏智勇

[suzhiyong@njust.edu.cn](mailto:suzhiyong@njust.edu.cn)

<https://zhiyongsu.github.io>

**Visual Computing Group, NJUST**

---

# 教学内容

3.1 栈和队列的定义和特点

3.2 栈的表示和操作的实现

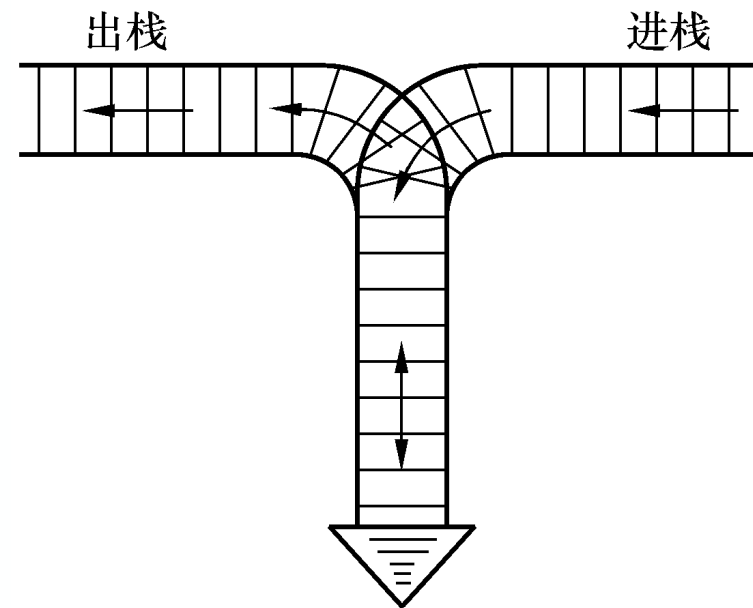
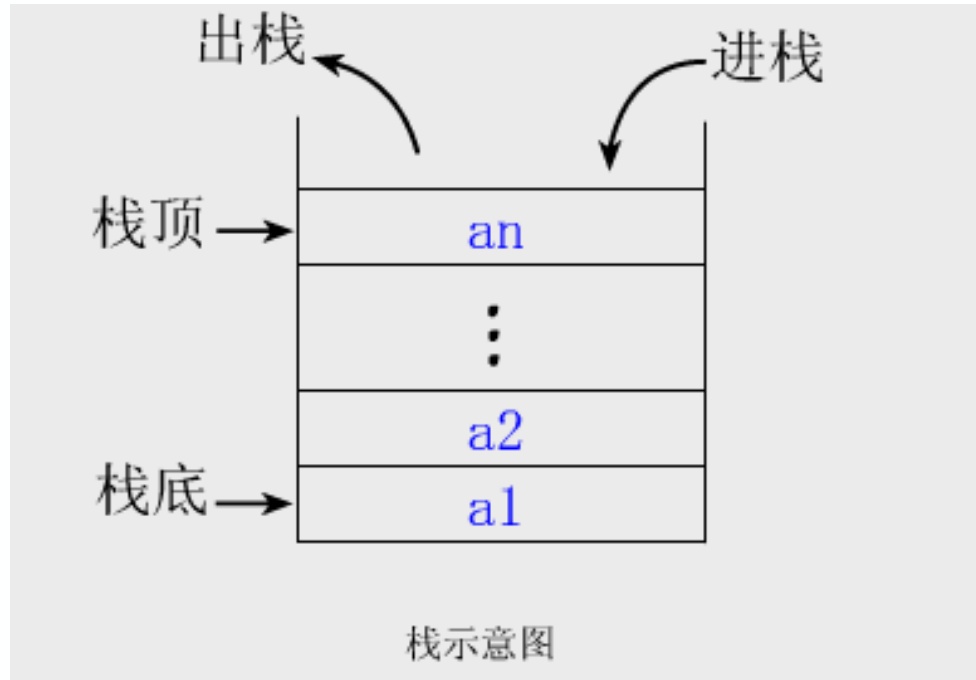
3.3 队列的的表示和操作的实现

3.4 案例分析与实现

# 教学目标

1. 掌握栈和队列的**特点**，并能在相应的应用问题中正确选用
2. 熟练掌握栈的**两种存储结构**的基本操作实现算法，特别注意**栈满和栈空**的条件
3. 熟练掌握**循环队列和链队列**的基本操作实现算法，特别注意**队满和队空**的条件
4. 理解**递归算法**执行过程中栈的状态变化过程
5. 掌握表**表达式求值**方法

# 栈

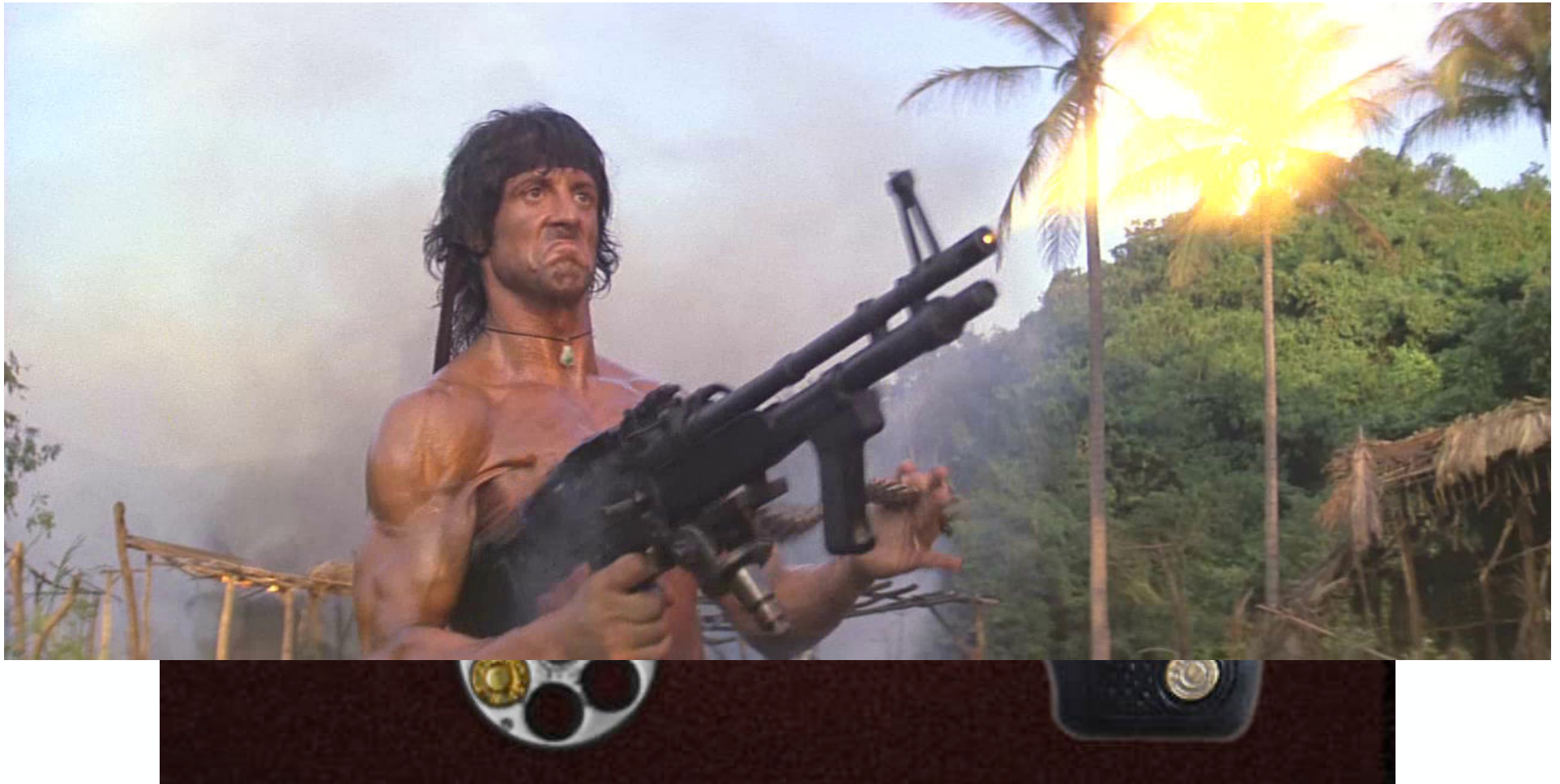


用铁路调度站表示栈



---

# 栈



## 3.1 栈和队列的定义和特点



### 栈

1. 定义 只能在表的一端（栈顶）进行插入和删除运算的线性表
2. 逻辑结构 与线性表相同，仍为一对一关系
3. 存储结构 用顺序栈或链栈存储均可，但以顺序栈更常见

---

## 4. 运算规则

只能在栈顶运算，且访问结点时依照后进先出（LIFO）或先进后出（FILO）的原则

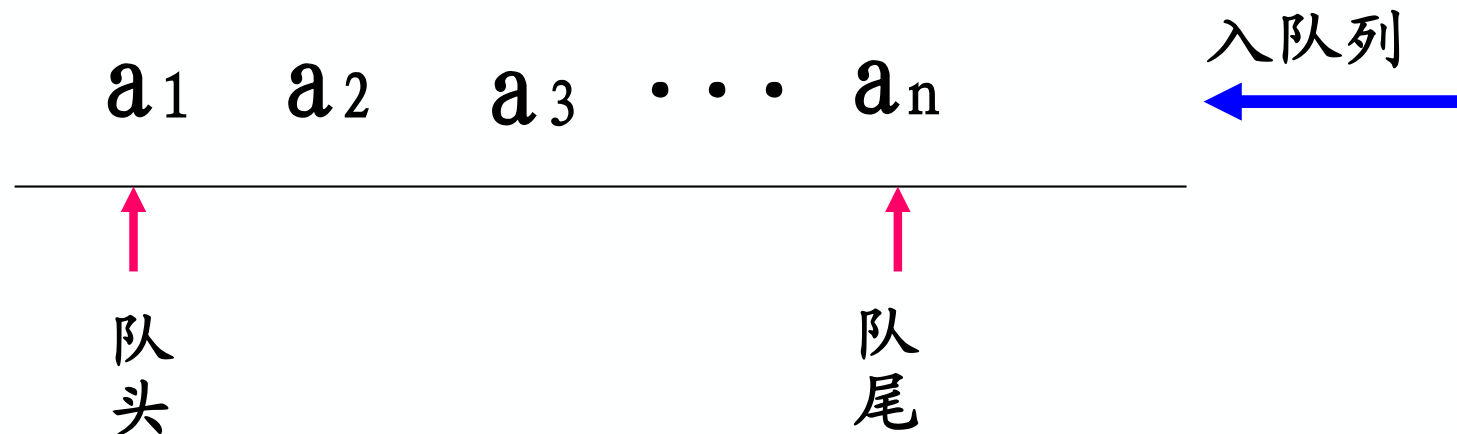
## 5. 实现方式

关键是编写入栈和出栈函数，具体实现依顺序栈或链栈的不同而不同

基本操作有入栈、出栈、读栈顶元素值、建栈、判断栈满、栈空等

队列是一种先进先出(**FIFO**) 的线性表。在表一端插入，在另一端删除。

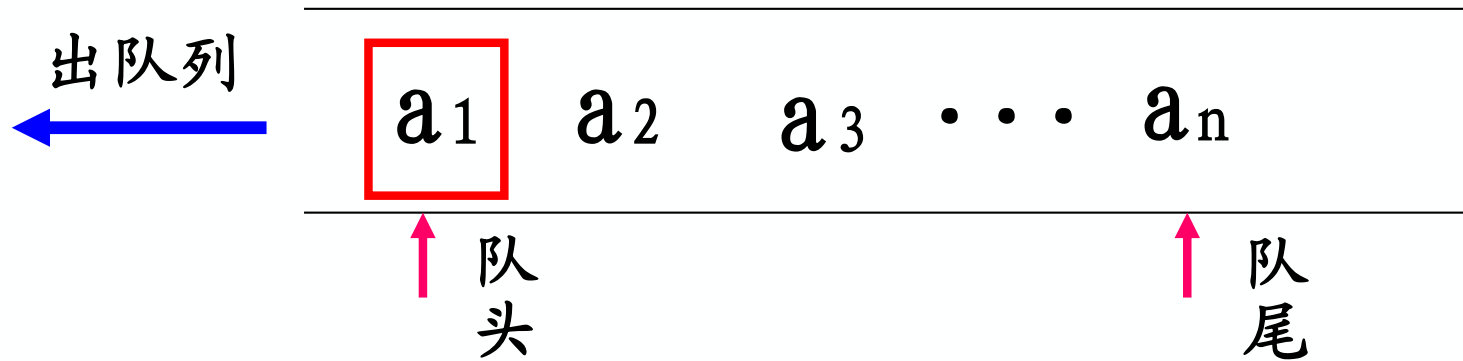
$$q = (a_1, a_2, \dots, a_n)$$



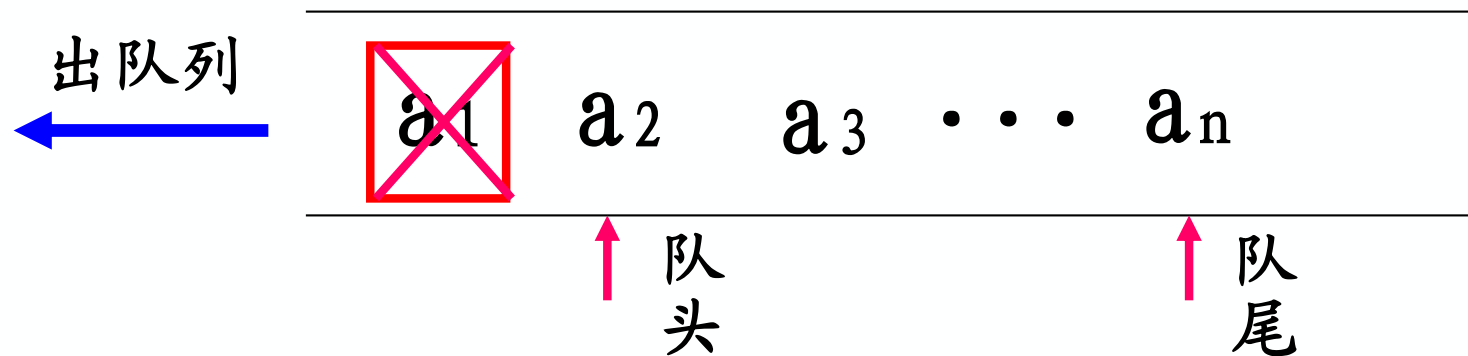


---

$$q = (a_1, a_2, \dots, a_n)$$



$$q = (a_1, a_2, \dots, a_n)$$



## 3.1 栈和队列的定义和特点

### 队列

1. 定义 只能在表的一端（队尾）进行插入，在另一端（队头）进行删除运算的线性表
2. 逻辑结构 与线性表相同，仍为一对一关系
3. 存储结构 用顺序队列或链队存储均可
4. 运算规则 先进先出（FIFO）
5. 实现方式 关键是编写入队和出队函数，具体实现依顺序队或链队的不同而不同

# 栈、队列与一般线性表的区别

栈、队列是一种特殊（**操作受限**）的线性表  
区别：仅在于**运算规则**不同

## 一般线性表

逻辑结构：一对一  
存储结构：顺序表、链表  
运算规则：**随机、顺序存取**

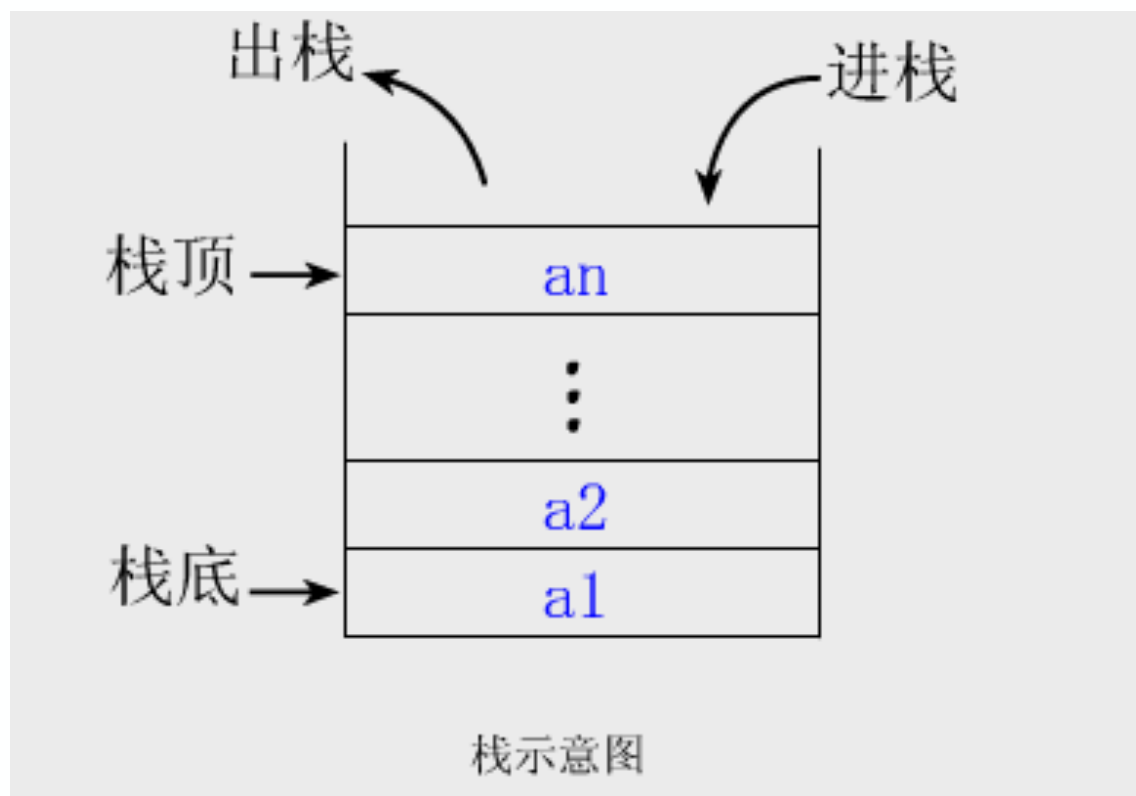
## 栈

逻辑结构：一对一  
存储结构：顺序栈、链栈  
运算规则：**后进先出**

## 队列

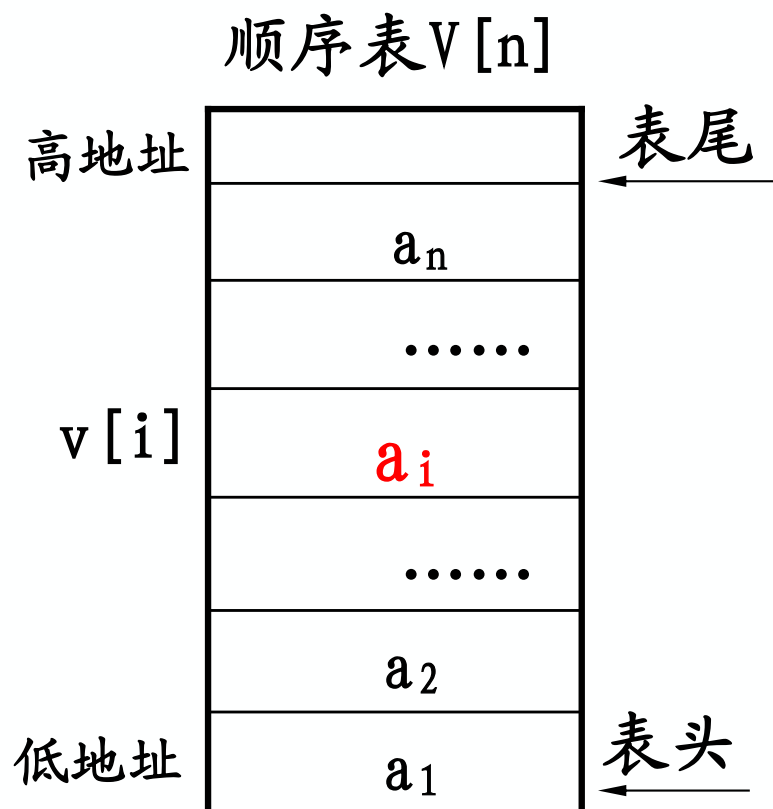
逻辑结构：一对一  
存储结构：顺序队、链队  
运算规则：**先进先出**

## 3.2 栈的表示和操作的实现

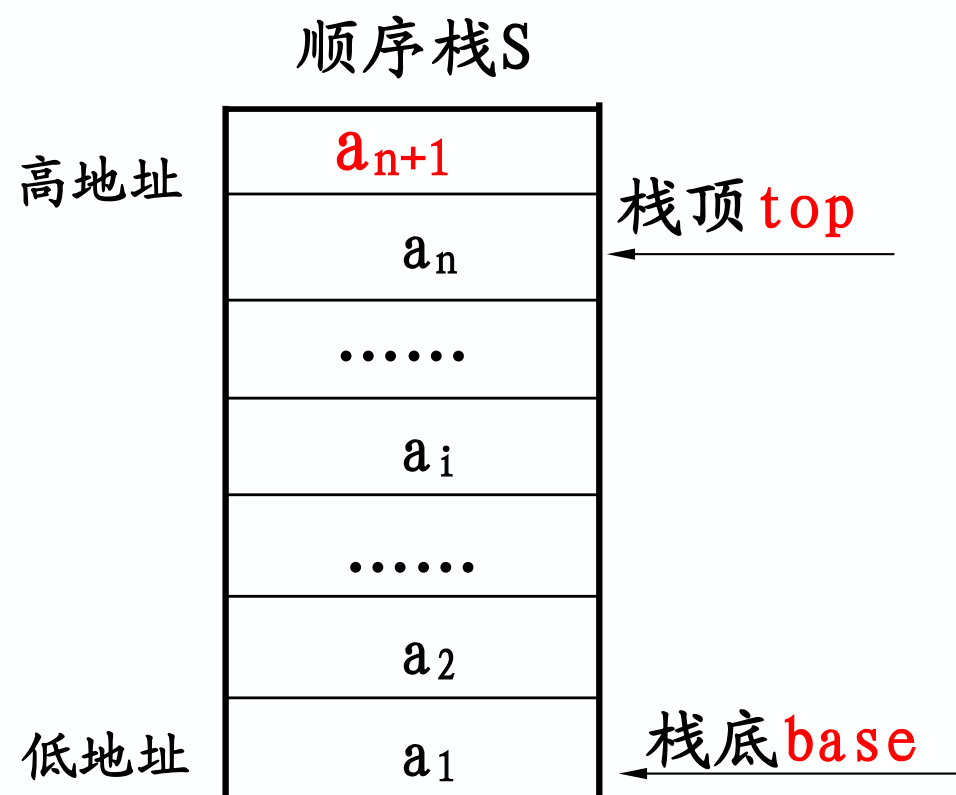


“进” = 压入= PUSH ( )  
“出” = 弹出= POP ( )

# 顺序栈与顺序表



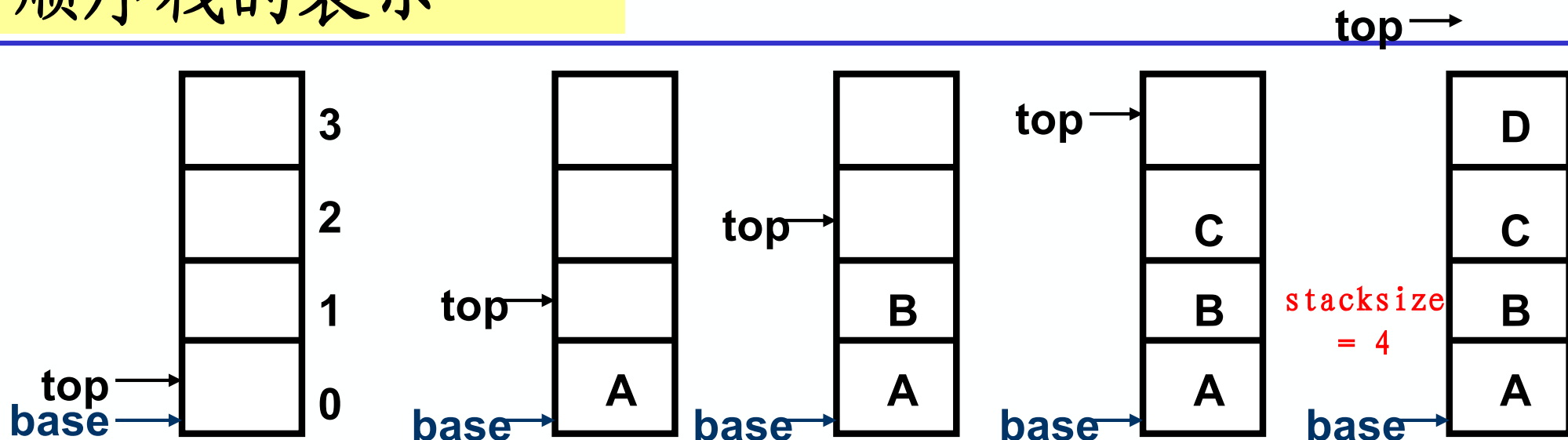
写入:  $v[i] = a_i$   
读出:  $x = v[i]$



压入:  $PUSH(a_{n+1})$   
弹出:  $POP(x)$

前提: 一定要预设栈顶指针  $top$ !

# 顺序栈的表示



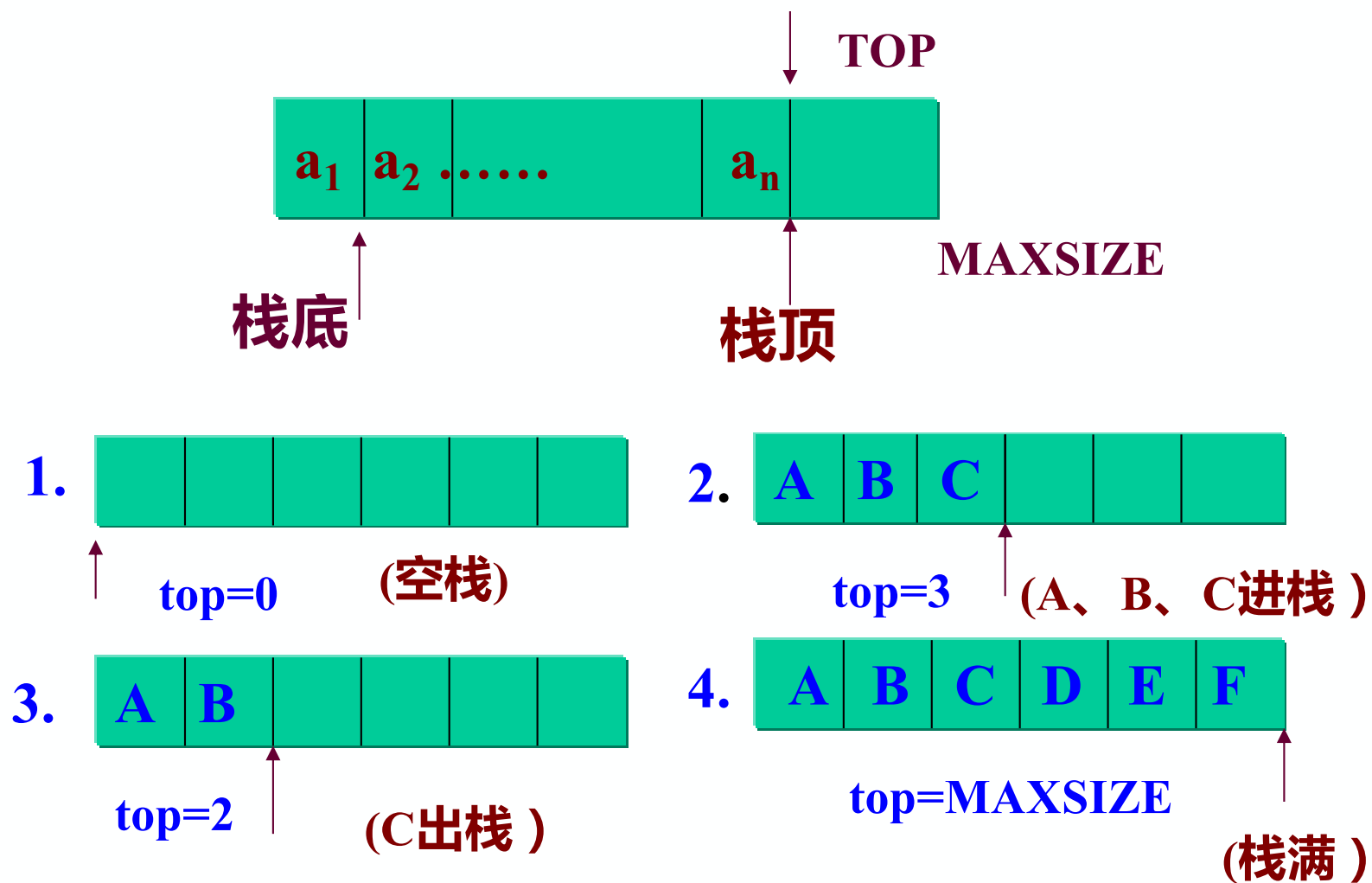
top 指示真正的栈顶元素之上的下标地址  
栈满时的处理方法:

- 1、报错, 返回操作系统。
- 2、分配更大的空间, 作为栈的存储空间, 将原栈的内容移入新栈。

空栈  $base == top$   
是栈空标志

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

# 顺序栈的表示



# 顺序栈的表示

---

(1) 栈的顺序存储结构：用一维数组作为存储空间。

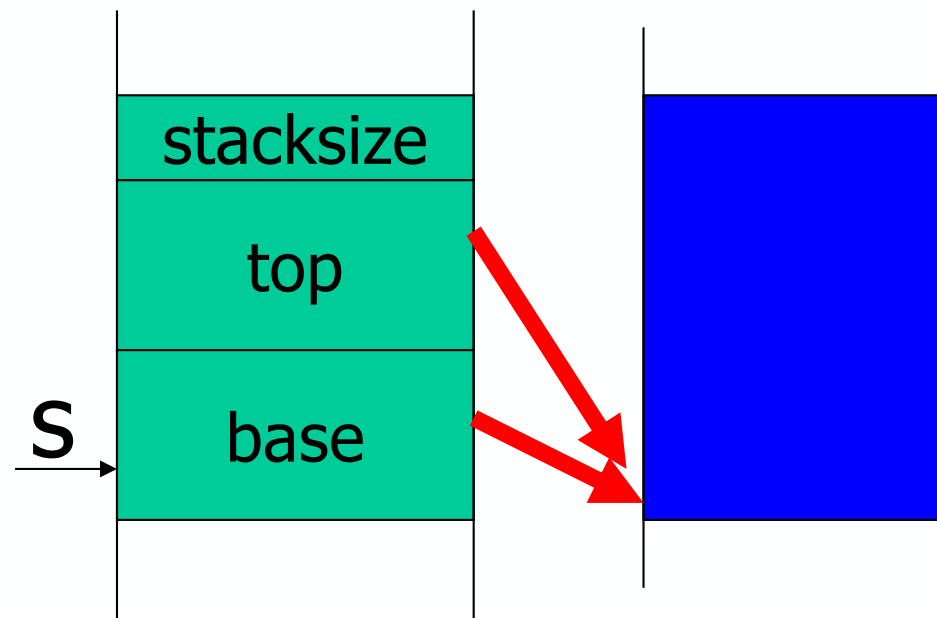
(2) 顺序栈：栈的**顺序存储结构**称为顺序栈。

栈的操作只能在一端进行；即栈顶位置随进栈和出栈而变化。



# 顺序栈初始化

- 构造一个空栈
- 步骤:
  - (1) 分配空间并检查空间是否分配失败，若失败则返回错误
  - (2) 设置栈底和栈顶指针  
 **$S.top = S.base;$**
  - (3) 设置栈大小



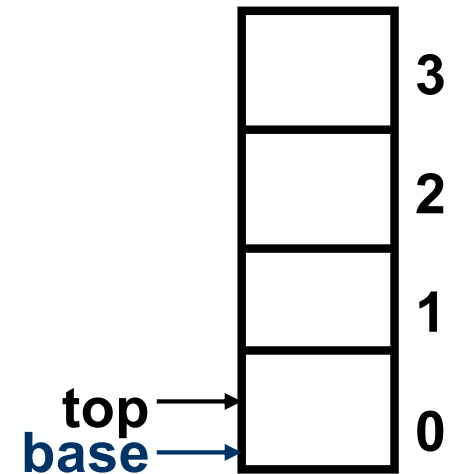
# 顺序栈初始化

---

```
Status InitStack( SqStack *pS )
{
    pS->base=(SElemType*)malloc(MAXSIZE*sizeof(SElemType));
    if( !pS->base )    return OVERFLOW;
    pS->top = pS->base;
    pS->stacksize = MAXSIZE;
    return OK;
}
```

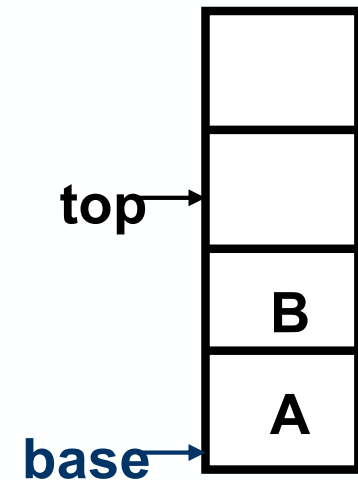
## 判断顺序栈是否为空

```
bool StackEmpty( SqStack S )  
{  
    if(S.top == S.base) return true;  
    else return false;  
}
```



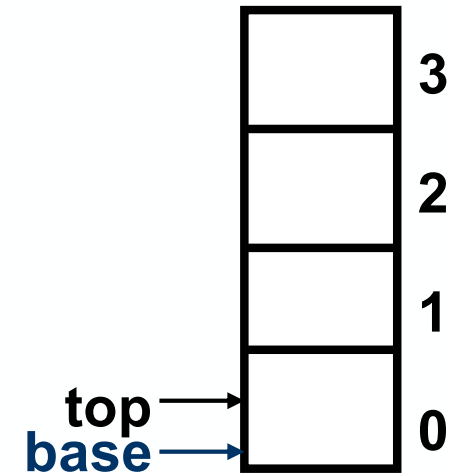
## 求顺序栈的长度

```
int StackLength( SqStack S )  
{  
    return S.top - S.base;  
}
```



## 清空顺序栈

```
Status ClearStack( SqStack *pS )  
{  
    if(pS->base ) pS-> top = pS->base ;  
    return OK;  
}
```

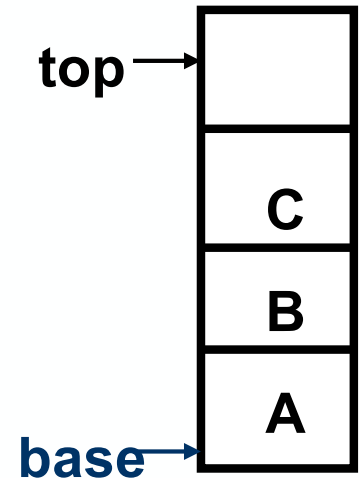


## 销毁顺序栈

```
Status DestroyStack( SqStack *pS ){  
    if( pS->base ){ free(pS->base) ;  
        pS->stacksize = 0;  
        pS->base = pS->top = NULL;}  
    return OK;}
```

# 顺序栈进栈

- (1) 判断是否栈满，若满则出错
- (2) 元素e压入栈顶
- (3) 栈顶指针加1



```
Status Push( SqStack *pS, SElemType e)
```

```
{
```

```
    if( pS->top - pS->base == pS->stacksize ) // ? 几种方式?
```

```
        return ERROR;
```

```
        *(pS->top++) = e;
```

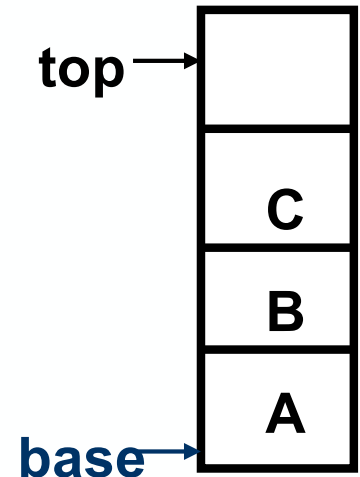
```
        return OK;
```

```
}
```

**\*pS->top = e;**  
**pS->top++;**

# 顺序栈出栈

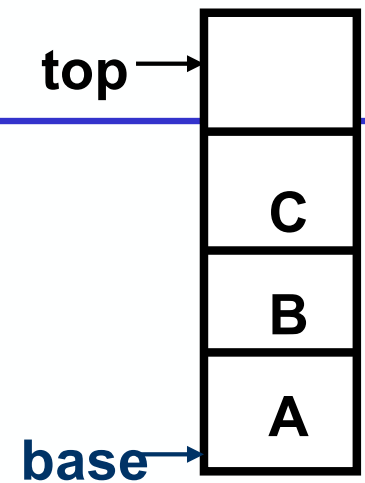
- (1) 判断是否栈空，若空则出错
- (2) 获取栈顶元素e
- (3) 栈顶指针减1



```
Status Pop( SqStack *pS, SElemType *pe)
{
    if( pS->top == pS->base ) // ?几种方式?
        return ERROR;
    (*pe)= *(--pS->top);
    return OK;
}
```

## 取顺序栈栈顶元素

- (1) 判断是否空栈，若空则返回错误
- (2) 否则通过栈顶指针获取栈顶元素



```
SElemType GetTop( SqStack S)
```

```
{
```

```
    if( S.top != S.base )
```

```
        return *( S.top-1 );
```

```
}
```

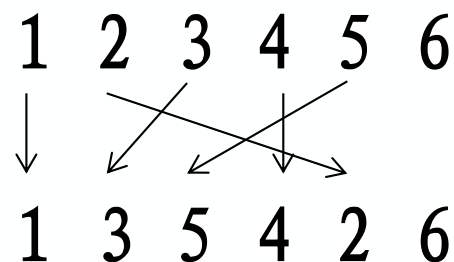
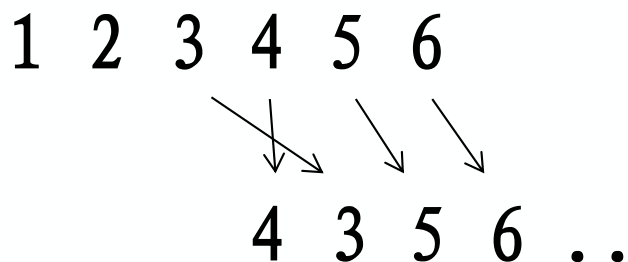
参考代码: SqStackTestDev

调用: TestCreatStack 函数

# 课堂练习

1. 如果一个栈的输入序列为123456，能否得到435612和135426的出栈序列？

435612中到了12顺序不能实现；  
135426可以实现。





## 课堂练习

2. 若已知一个栈的入栈序列是 $1, 2, 3, \dots, n$ , 其输出序列为 $p_1, p_2, p_3, \dots, p_n$ , 若 $p_1=n$ , 则 $p_i$ 为

A.  $i$

B.  $n-i$

C.  $n-i+1$

D. 不确定

## 课堂练习

3. 在一个具有 $n$ 个单元的顺序栈中，假设以地址高端作为栈底，以 $top$ 作为栈顶指针，则当作进栈处理时， $top$ 的变化为

A.  $top$ 不变

B.  $top=0$

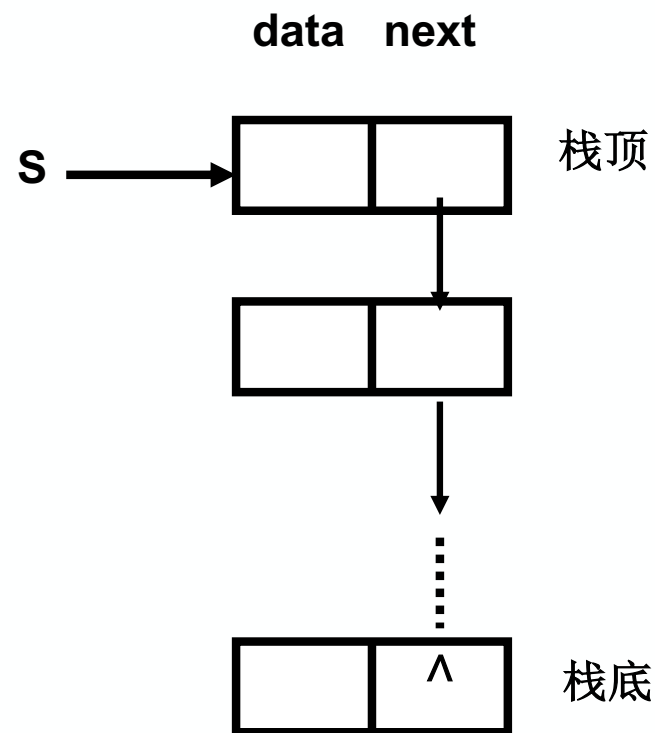
C.  $top++$

D.  $top--$

# 链栈的表示

- ✓ 运算是受限的单链表，只能在链表头部进行操作，故没有必要附加头结点。栈顶指针就是链表的头指针

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



# 链栈的初始化

$S \longrightarrow \wedge$

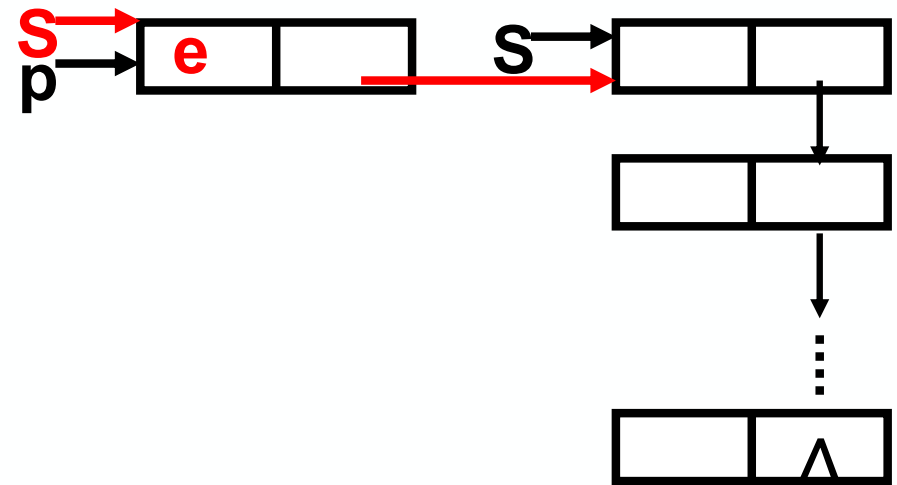
```
void InitStack(LinkStack *pS )  
{  
    *pS=NULL;  
}
```

## 判断链栈是否为空

---

```
Status StackEmpty(LinkStack S)  
{  
    if (S==NULL) return TRUE;  
    else return FALSE;  
}
```

# 链栈进栈

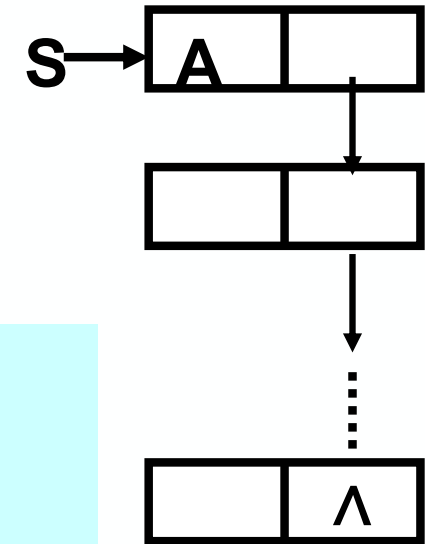


```
Status Push(LinkStack *pS , SElemType e)
{ StackNode* p=(StackNode*)malloc(sizeof(StackNode));
  if (!p) exit(OVERFLOW);
  p->data=e; p->next=*pS; *pS=p;
  return OK;
}
```

# 链栈出栈

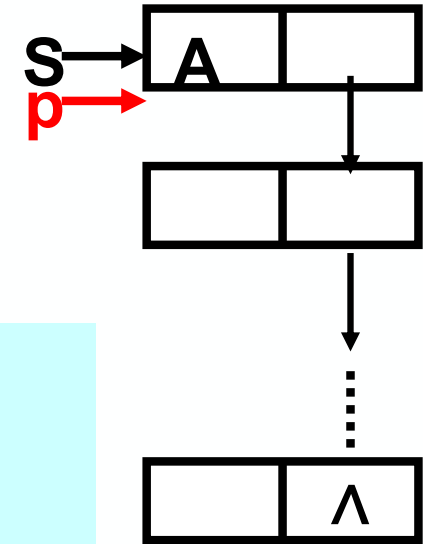
**e = 'A'**

```
Status Pop (LinkStack *pS,SElemType *pe)
{ StackNode* p=NULL;
  if ((*pS)==NULL) return ERROR;
  *pe = (*pS)->data;
  p = (*pS);
  (*pS) = (*pS)->next;
  free(p);
  return OK;
}
```



# 链栈出栈

$e = 'A'$



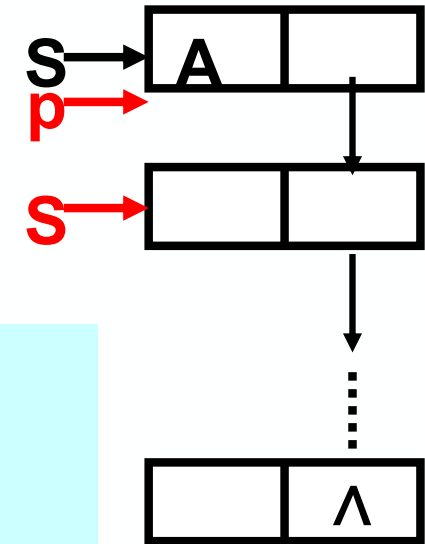
```
Status Pop (LinkStack *pS,SElemType *pe)
{ StackNode* p=NULL;
  if ((*pS)==NULL) return ERROR;
  *pe = (*pS)->data;
  p = (*pS);
  (*pS) = (*pS)->next;
  free(p);
  return OK;
}
```



# 链栈出栈

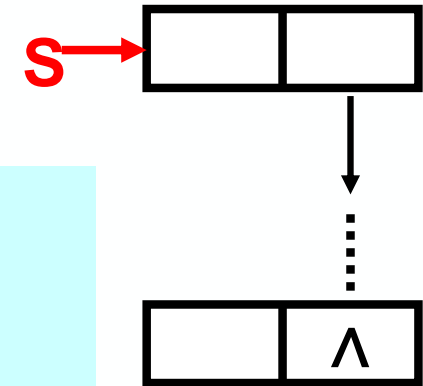
$e = 'A'$

```
Status Pop (LinkStack *pS,SElemType *pe)
{ StackNode* p=NULL;
  if ((*pS)==NULL) return ERROR;
  *pe = (*pS)->data;
  p = (*pS);
  (*pS) = (*pS)->next;
  free(p);
  return OK;
}
```



# 链栈出栈

**e = 'A'**



```
Status Pop (LinkStack *pS,SElemType *pe)
{ StackNode* p=NULL;
  if ((*pS)==NULL) return ERROR;
  *pe = (*pS)->data;
  p = (*pS);
  (*pS) = (*pS)-> next;
  free(p);
  return OK;
}
```

**SElemType GetTop(LinkStack S)**

```
{  
    if (S==NULL) exit(1);  
    else return S->data;  
}
```

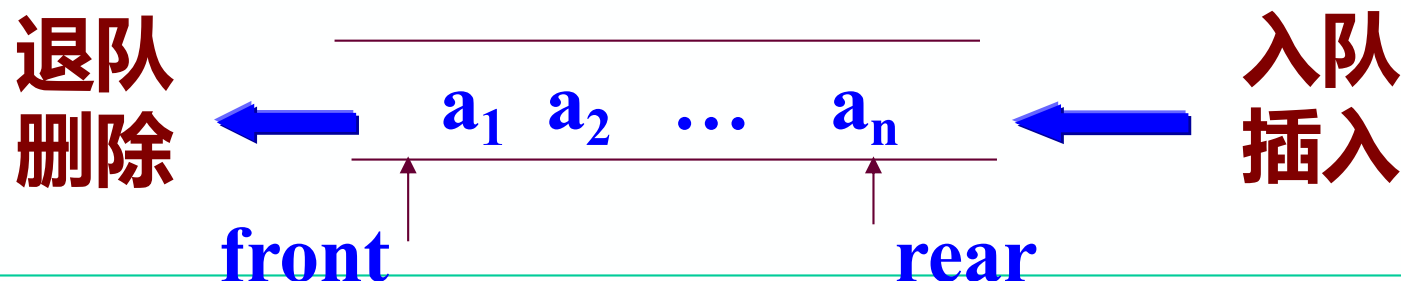
链栈代码参考

**LinkStackTestDev 目录**

**调用TestCreating()函数**

### 3.3 队列的表示和操作的实现

- 队列是一种特殊的线性表，它只允许在表的前端（**front**）进行删除操作，而在表的后端（**rear**）进行插入操作。
- 进行插入操作的端称为队尾，进行删除操作的端称为队头。
- 队列中没有元素时，称为空队列。
- 队列具有先进先出（**FIFO**）的特点。



- 
- **front**为队头指针，指示队头元素。
  - **rear** 为队尾指针，指示队尾元素的下一个位置。

---

# 队列的操作

清空队列

判别队列是否为空；空，取T；

非空，取值为F。

插入操作

删除操作

取队头元素

# 队列的抽象数据类型

**ADT Queue {**

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

数据关系:  $R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 1, 2, \dots, n \}$

基本操作: 约定 $a_1$ 端为队列头, $a_n$ 端为队列尾

- (1) **InitQueue ()** //构造空队列
- (2) **DestroyQueue ()** //销毁队列
- (3) **ClearQueue ()** //清空队列
- (4) **QueueEmpty()** //判空. 空--TRUE,

# 队列的抽象数据类型

(5) **QueueLength()**      //取队列长度

(6) **GetHead ()**      //取队头元素,

(7) **EnQueue ()**      //入队列

(8) **DeQueue ()**      //出队列

(9) **QueueTraverse()**      //遍历

**}ADT Queue**



## 队列的顺序表示 - - 用一维数组base[M]

```
#define M 100 //最大队列长度
```

```
Typedef struct {
```

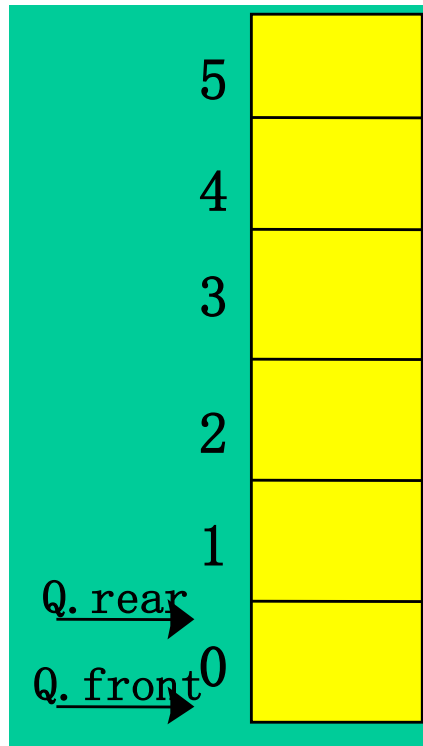
```
    QElemType *base; //初始化的动态分配存储空间
```

```
    int front;        //头指针
```

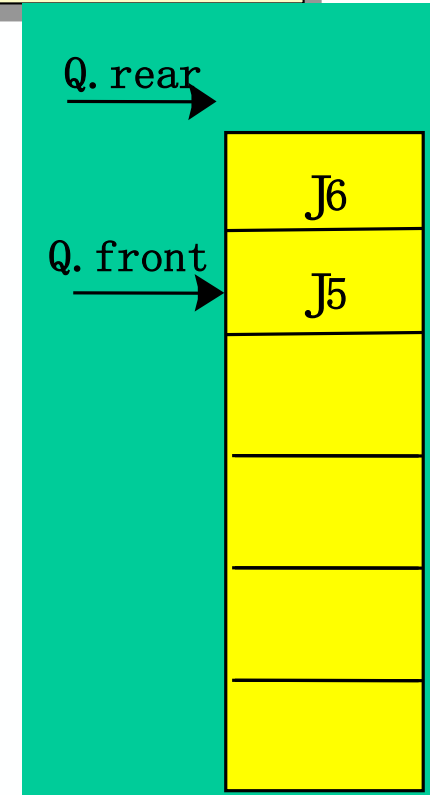
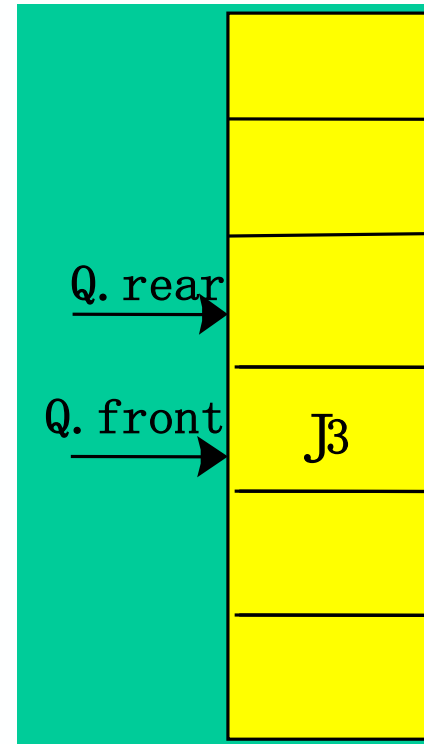
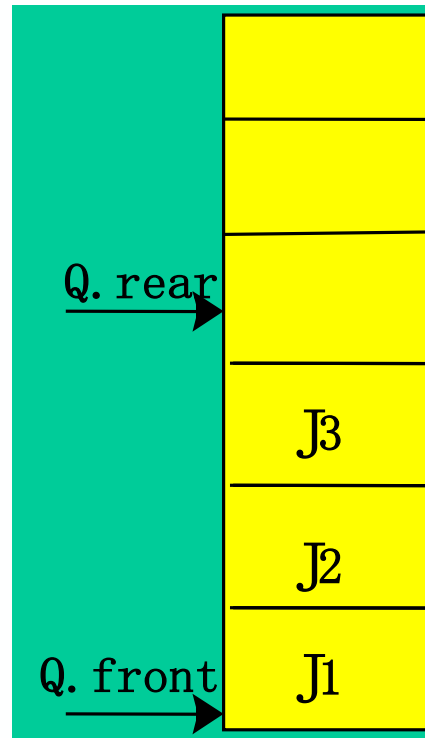
```
    int rear;         //尾指针
```

```
}SqQueue;
```

# 队列的顺序表示 - - 用一维数组base[M]



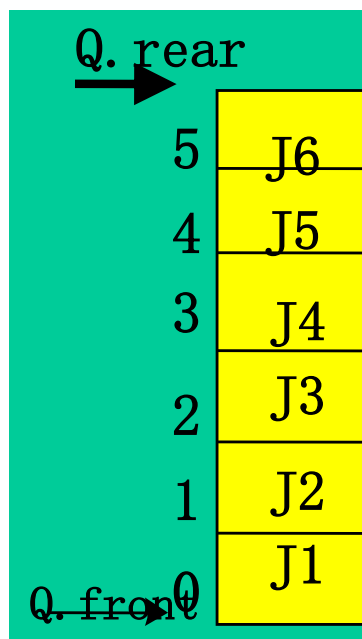
**front=rear=0**



空队标志: `front == rear`  
入队: `base[rear++] = x;`  
出队: `x = base[front++];`

# 存在的问题

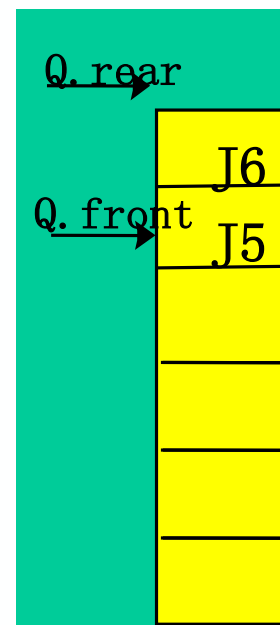
设大小为M



**front=0**

**rear=M时**

再入队——真溢出



**front≠0**

**rear=M时**

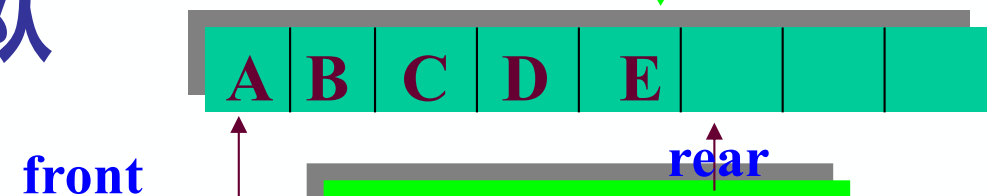
再入队——假溢出

## (A) 空队列



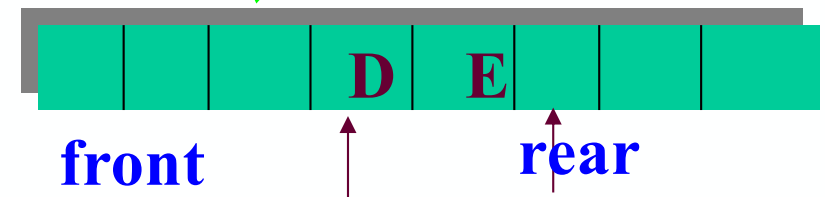
入队时，rear在变

## (B) A、B、C、D、E入队



出队时，front在变

## (C) A、B、C出队



---

( D ) F、 G、 H入队



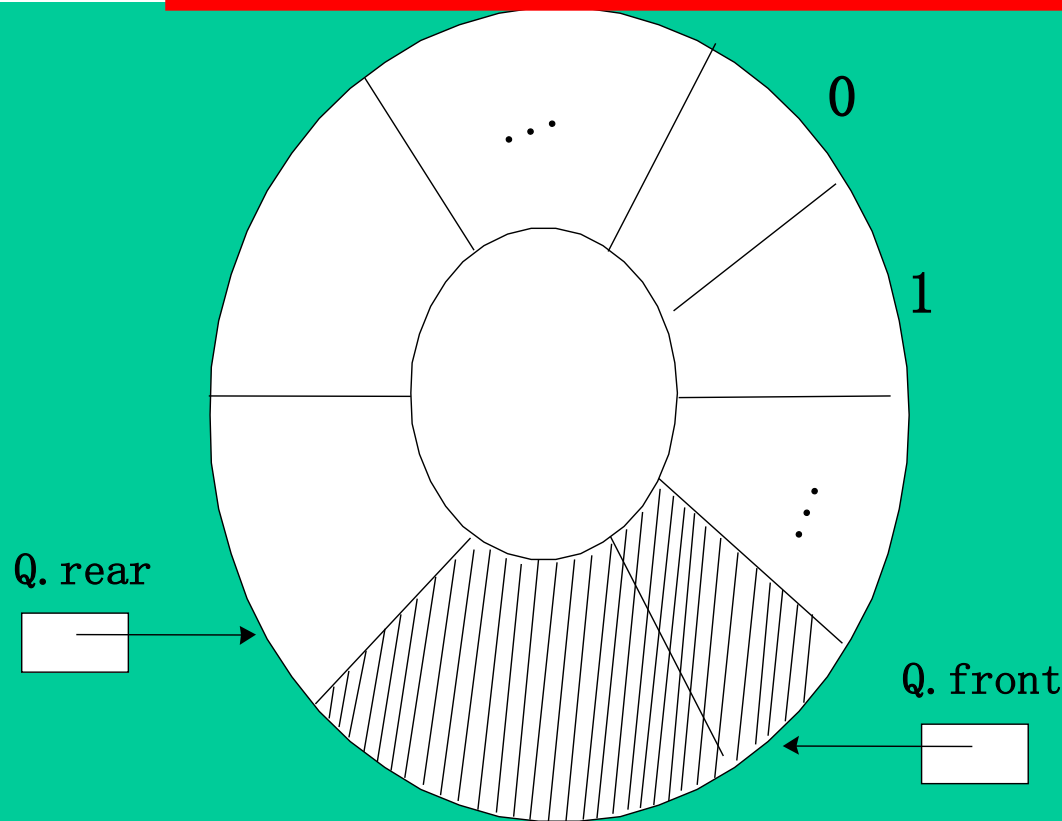
(E ) D、 E、 F、 G、 H出队 , 出现假 “溢出”



注：一方面队列中是空的，另一方面又出现溢出。  
显然，这是逻辑设计上的问题。

# 解决的方法 —— 循环队列

base[0]接在base[M-1]之后  
若 $\text{rear}+1==M$   
则令 $\text{rear}=0$ ;



实现：利用“模”运算

入队：

**$\text{base}[\text{rear}]=x;$**

**$\text{rear}=(\text{rear}+1)\%M;$**

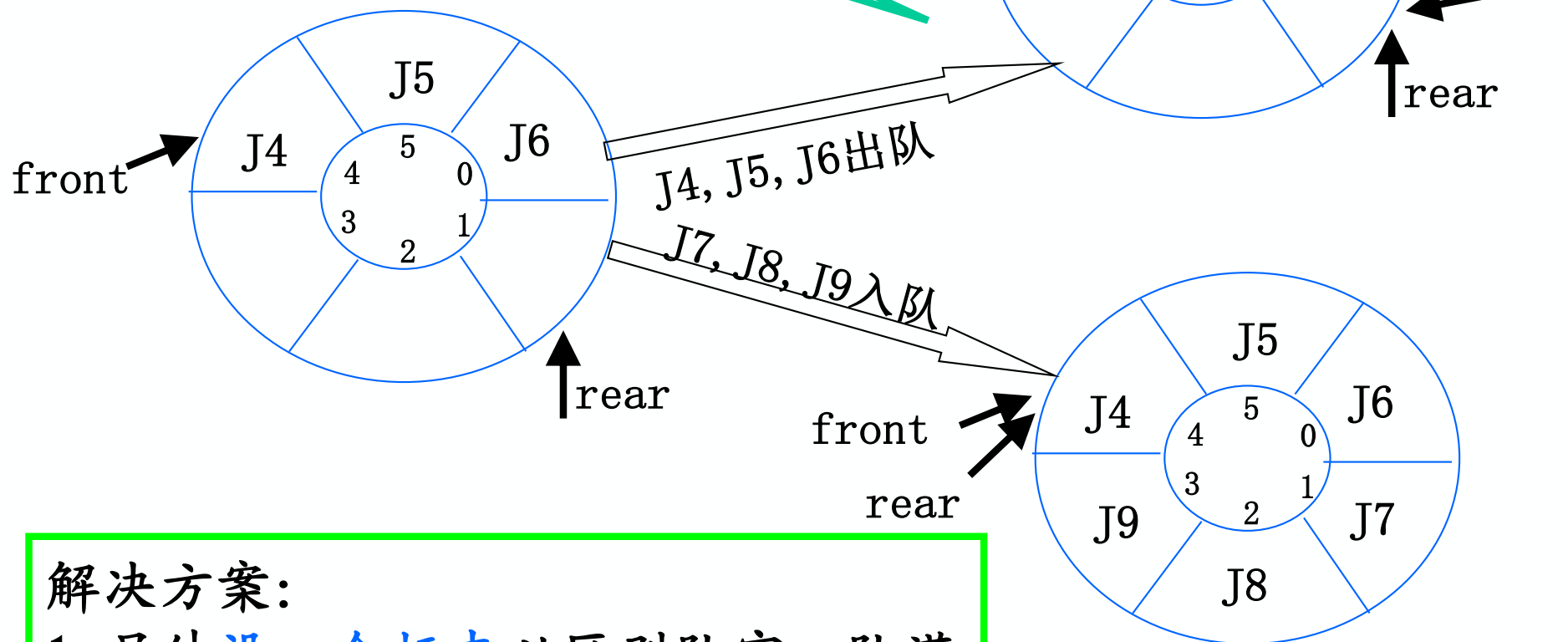
出队：

**$x=\text{base}[\text{front}];$**

**$\text{front}=(\text{front}+1)\%M;$**

队空:  $front == rear$

队满:  $front == rear$



解决方案:

1. 另外设一个标志以区别队空、队满

2. 少用一个元素空间:

队空:  $front == rear$

队满:  $(rear + 1) \% M == front$

# 循环队列

```
#define MAXQSIZE 100 //最大长度
```

```
Typedef struct {
```

```
    QElemType *base; //初始化的动态分配存储空间
```

```
    int front;        //头指针
```

```
    int rear;         //尾指针
```

```
}SqQueue;
```



## 循环队列初始化

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

## 求循环队列的长度

---

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

## 循环队列入队

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

## 循环队列出队

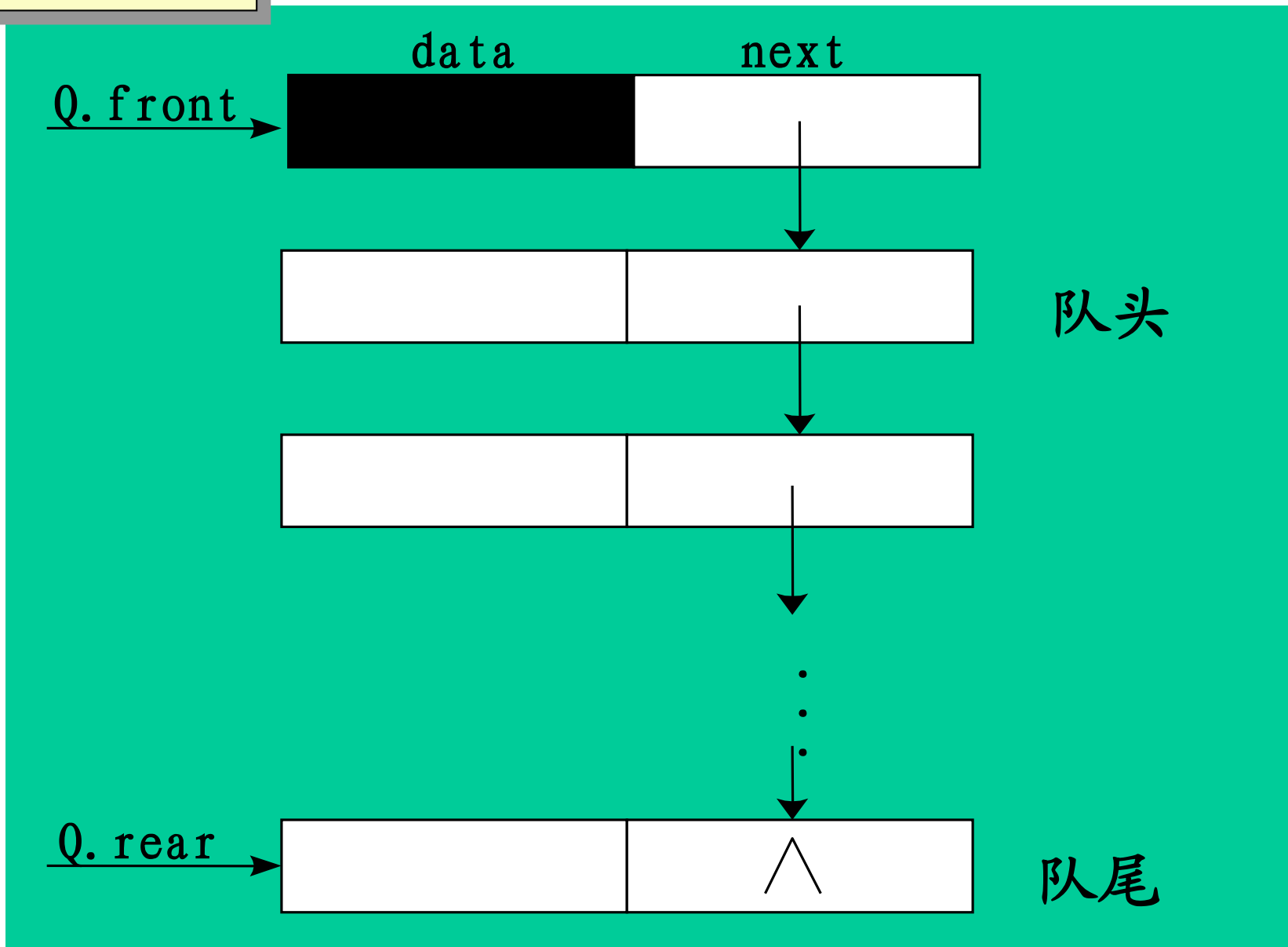
```
Status DeQueue (SqQueue *pQ, QElemType *pe){  
    if((*pQ).front==(*pQ).rear) return ERROR;  
    *pe=(*pQ).base[(*pQ).front];  
    (*pQ).front=((*pQ).front+1)%MAXQSIZE;  
    return OK;  
}
```

---

```
QElemType GetHead(SqQueue Q)
{
    if(Q.front!=Q.rear)
        return Q.base[Q.front];
}
```

代码参考SqQueueTestDev  
TestCreatSqQueue函数

# 链队列



# 链队列

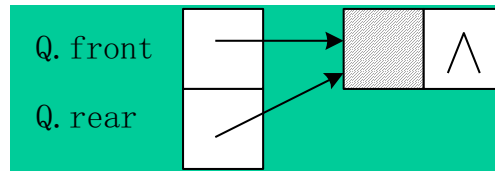
---

```
typedef struct QNode{  
    QElemType  data;  
    struct Qnode *next;  
}Qnode, *QueuePtr;
```

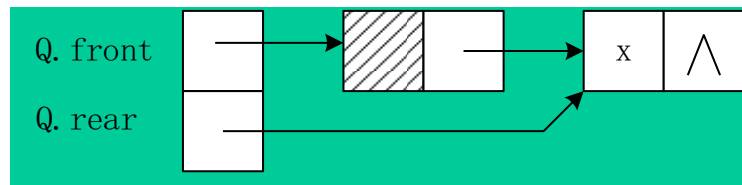
```
typedef struct {  
    QueuePtr front;    //队头指针  
    QueuePtr rear;    //队尾指针  
}LinkQueue;
```

# 链队列

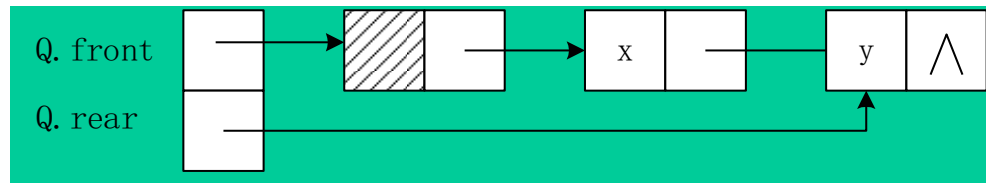
(a) 空队列



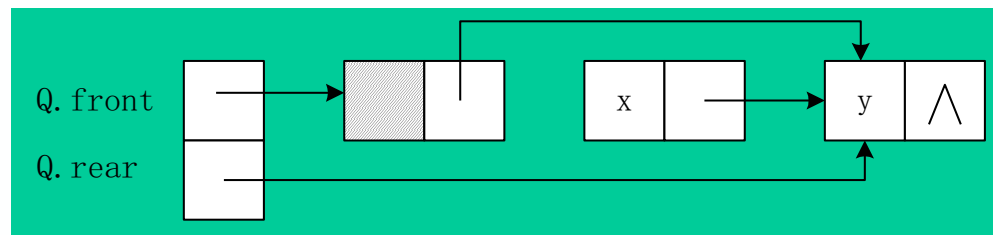
(b) 元素x入队列



(c) 元素y入队列



(d) 元素x出队列





## 链队列初始化

---

```
Status InitQueue (LinkQueue *pQ){  
    (*pQ).front=(*pQ).rear=(QueuePtr)  
    malloc(sizeof(QNode));  
    if(!(*pQ).front) exit(OVERFLOW);  
    (*pQ).front->next=NULL;  
    return OK;  
}
```

## 销毁链队列

```
Status DestroyQueue (LinkQueue *pQ){  
    while((*pQ).front){  
        (*pQ).rear=(*pQ).front->next;  
        free((*pQ).front);  
        (*pQ).front=(*pQ).rear; }//巧用rear指针  
    return OK;  
}
```

## 判断链队列是否为空

---

```
Status QueueEmpty (LinkQueue Q){  
    return (Q.front==Q.rear);  
}
```

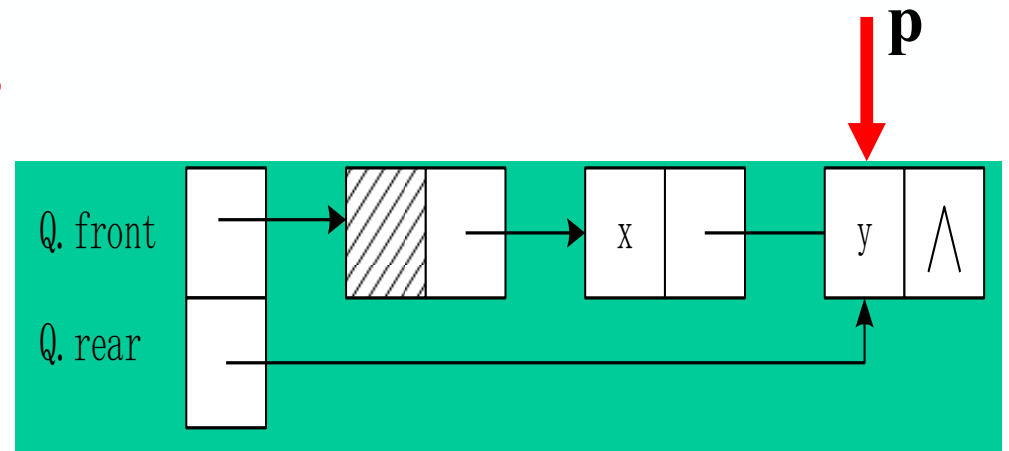
## 求链队列的队头元素

---

```
Status GetHead (LinkQueue Q, QElemType *pe){  
    if(Q.front==Q.rear) return ERROR;  
    *pe=Q.front->next->data;  
    return OK;  
}
```

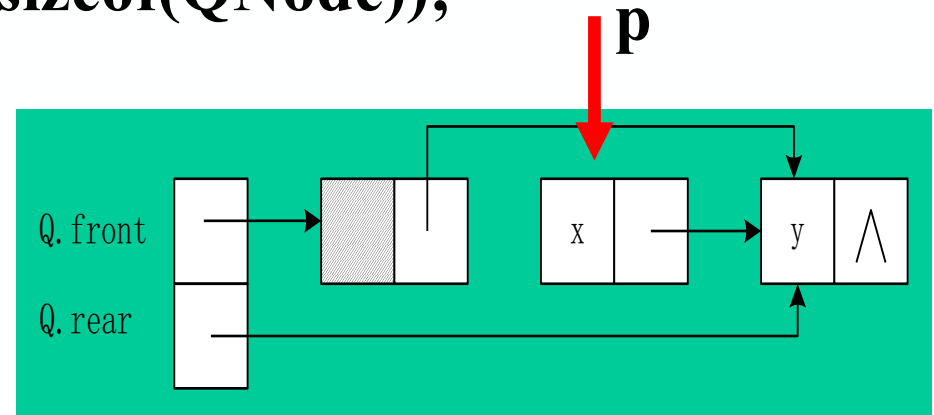
# 链队列入队

```
Status EnQueue(LinkQueue *pQ, QElemType e){  
    QueuePtr p=(QueuePtr)malloc(sizeof(QNode));  
    if(!p) exit(OVERFLOW);  
    p->data=e; p->next=NULL;  
    (*pQ).rear->next=p;  
    (*pQ).rear=p;  
    return OK;  
}
```



# 链队列出队

```
Status DeQueue (LinkQueue *pQ, QElemType *pe){  
    QueuePtr p=(QueuePtr)malloc(sizeof(QNode));  
    if(!p) exit(OVERFLOW);  
    if((*pQ).front==(*pQ).rear)  
        return ERROR;  
    p=(*pQ).front->next;  
    *pe=p->data;  
    (*pQ).front->next=p->next;  
    if((*pQ).rear==p)  
        (*pQ).rear=(*pQ).front;  
    free(p);  
    return OK;  
}
```



代码参考:

LinkQueueTestDev

TestCreateLinkQueue()

## 3.4 案例分析与实现

### 案例1：数制的转换

#### 【算法步骤】

- ① 初始化一个空栈S。
- ② 当十进制数N非零时，循环执行以下操作：
  - 把N与8求余得到的八进制数压入栈S；
  - N更新为N与8的商。
- ③ 当栈S非空时，循环执行以下操作：
  - 弹出栈顶元素e；
  - 输出e。

## 案例1：数制的转换

### 【算法描述】

```
void conversion(int N)
{ // 对于任意一个非负十进制数，打印输出与其等值的八进制数
    SqStack myStack;
    InitStack(&myStack);          //
    SElemType elem;
    while(N)                       //
    {
        Push(&myStack, N%8);
        N=N/8;                     //
    }
    while(!StackEmpty(myStack)) //
    {
        Pop(&myStack, &elem);      //
        printf("%d ", elem);       //
    }
}
```

参考代码SqStackTestDev  
调用函数conversion(28);



## 案例2：括号的匹配

### 【算法步骤】

- ① 初始化一个空栈S。
- ② 设置一标记性变量flag，用来标记匹配结果以控制循环及返回结果，1表示正确匹配，0表示错误匹配，flag初值为1。
- ③ 扫描表达式，依次读入字符ch，如果表达式没有扫描完毕或flag非零，则循环执行以下操作：
  - 若ch是左括号 “[”或 “(”，则将其压入栈；
  - 若ch是右括号 “)””，则根据当前栈顶元素的值分情况考虑：若栈非空且栈顶元素是 “(”，则正确匹配，否则错误匹配，flag置为0；
  - 若ch是右括号 “]”，则根据当前栈顶元素的值分情况考虑：若栈非空且栈顶元素是 “[”，则正确匹配，否则错误匹配，flag置为0。
- ④ 退出循环后，如果栈空且flag值为1，则匹配成功，返回true，否则返回false。

## 案例2：括号的匹配

### Status Matching( )

{

//检验表达式中所含括号是否正确匹配，如果匹配，则返回true，否则返回false

char ch;

SElemType x;

LinkStack S;

InitStack(&S); //初始化空栈

int flag = 1; //标记匹配结果以控制循环及返回结果

ch=getchar();

## 案例2：括号的匹配

///**假设表达式以 “#” 结尾**

**switch (ch) {**

**case '[' :**

**case '(' :** //若是左括号，则将其压入栈

**Push(&S, ch); break;**

**case ')' :** //若是 “)”，则根据当前栈顶元素的值分情况考虑

**if (!StackEmpty(S) && GetTop(S) == '(')**

**{ Pop(&S, &x);** //若栈非空且栈顶元素是 “(”，则正确匹配

**//printf("match )\n");      }**

**else**

**flag = 0;** //若栈空或栈顶元素不是 “(”，错误失败

**break;**

**case ']' :** //若是 “]”，则根据当前栈顶元素的值分情况考虑

**if (!StackEmpty(S) && GetTop(S) == '[')**

**Pop(&S, &x);** //若栈非空且栈顶元素是 “[”，则正确匹配

**else flag = 0;** //若栈空或栈顶元素不是 “[”，则错误匹

**break;**

**}**

**ch=getchar();**

**}**

## 案例2：括号的匹配

...

```
if (StackEmpty(S) && flag)
    return true; //匹配成功
else
    return false; //匹配失败
}
```

代码参考 **LinkStackTestDev**  
调用函数  
**TestMatch()**  
运行时输入  
**4+3\*(3-2)#**

## 案例2：括号的匹配

---

```
if (StackEmpty(S) && flag)
    return 1; //匹配成功
else
    return 0; //匹配失败
}
```

### 算术四则运算规则

- (1) 先乘除, 后加减
- (2) 从左算到右
- (3) 先括号内, 后括号外

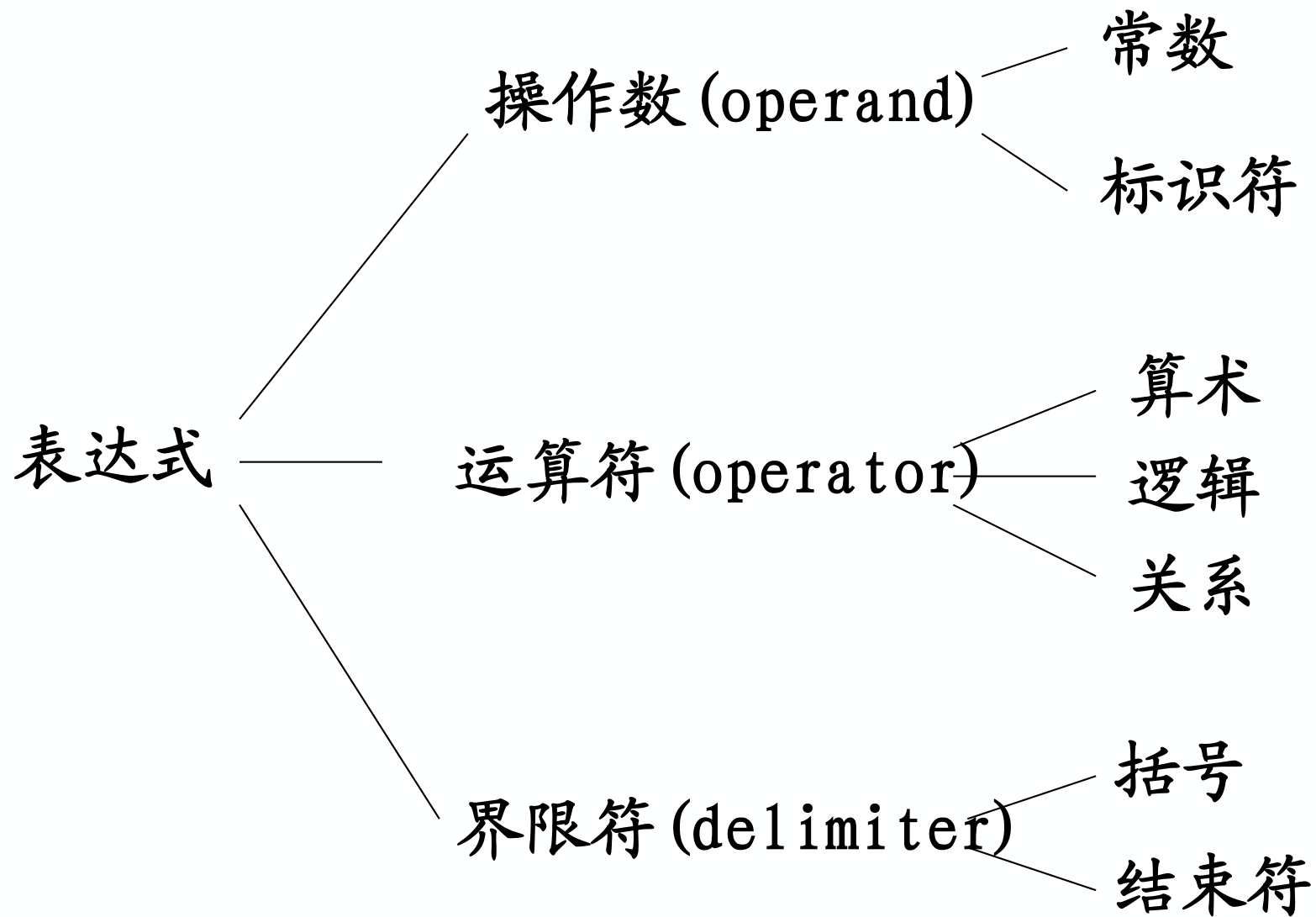


表3.1 算符间的优先关系

$\theta_1 \backslash \theta_2$	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	X
)	>	>	>	>	X	>	>
#	<	<	<	<	<	X	=



# 算法步骤

设定两栈：OPND-----操作数或运算结果    OPTR-----运算符

- ① 初始化OPTR栈和OPND栈，将表达式起始符“#”压入OPTR栈。
- ② 扫描表达式，读入第一个字符ch，如果表达式没有扫描完毕至“#”或OPTR的栈顶元素不为“#”时，则循环执行以下操作：
  - 若ch不是运算符，则压入OPND栈，读入下一字符ch；
  - 若ch是运算符，则根据OPTR的栈顶元素和ch的优先级比较结果，做不同的处理：
    - 若是小于，则ch压入OPTR栈，读入下一字符ch；
    - 若是大于，则弹出OPTR栈顶的运算符，从OPND栈弹出两个数，进行相应运算，结果压入OPND栈；
    - 若是等于，则OPTR的栈顶元素是“(”且ch是“)”，这时弹出OPTR栈顶的“(”，相当于括号匹配成功，然后读入下一字符ch。
- ③ OPND栈顶元素即为表达式求值结果，返回此元素。

```
//测试输入4+7 + 3*(5-2)#
```

```
char EvaluateExpression( ) {  
    SqStack myOPTR,myOPND;  
    SqStack *OPTR=&myOPTR,*OPND=&myOPND;  
    char ch;  
    SElemType b,a,x;  
    char theta;  
    InitStack (OPTR); Push (OPTR,'#') ;  
    InitStack (OPND); ch = getchar( );  
    while (ch!= '#' || GetTop(myOPTR)!= '#')  
    {  
        //printf("%c\n",ch);  
        //处理空格  
        while(ch==' ') {ch = getchar(); }  
        if (!In(ch)) // ch运算数则进栈  
        { Push(OPND,ch-'0'); ch = getchar();}  
        else  
        switch (Precede(GetTop(myOPTR),ch))  
        { //比较优先权  
            case '<' : //当前字符ch压入OPTR栈, 读入下一字符ch  
                Push(OPTR, ch); ch = getchar(); break;  
            case '>' : //弹出OPTR栈顶的运算符运算, 并将运算结果入栈  
                Pop(OPTR, &theta);  
                Pop(OPND, &b); Pop(OPND, &a);  
                Push(OPND, Operate(a, theta, b)); break;  
            case '=' : //脱括号并接收下一字符  
                Pop(OPTR,&x); ch = getchar(); break;  
        } // switch  
    } // while  
    return GetTop(myOPND);  
} // EvaluateExpression
```

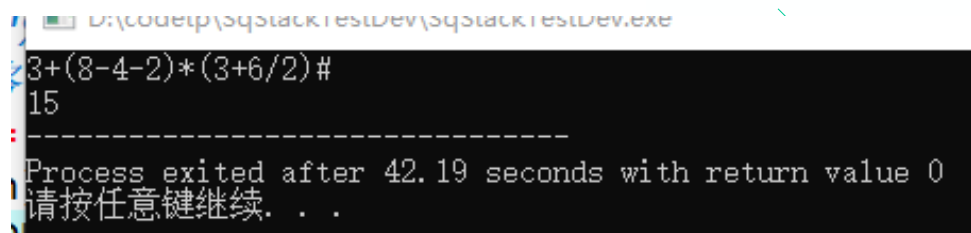
表达式运算,  
参考代码:

SqStackTestDev

函数EvaluateExpression()

运行时输入

4+7 + 3\*(5-2)#



```
D:\code\p\sqstacktestdev\sqstacktestdev.exe  
3+(8-4-2)*(3+6/2)#  
15  
-----  
Process exited after 42.19 seconds with return value 0  
请按任意键继续. . .
```

OPTR	OPND	INPUT	OPERATE
#		3*(7-2)#	Push(opnd,'3')
#	3	*(7-2)#	Push(optr,'*')
#,*	3	(7-2)#	Push(optr,'(')
#,*,(	3	7-2)#	Push(opnd,'7')
#,*,(	3,7	-2)#	Push(optr,'-')
#,*,(,—	3,7	2)#	Push(opnd,'2')
#,*,(,—	3,7,2	)#	Operate(7-2)
#,*,(	3,5	)#	Pop(optr)
#,*	3,5	#	Operate(3*5)
#	15	#	GetTop(opnd)

计算 $3*(7-2)$

---

# 队列的其它应用

## 【例】汽车加油站

结构：入口和出口为单行道，加油车道若干条 $n$

每辆车加油都要经过三段路程，三个队列

- 1. 入口处排队等候进入加油车道
- 2. 在加油车道排队等候加油
- 3. 出口处排队等候离开

若用算法模拟，需要设置 $n+2$ 个队列。

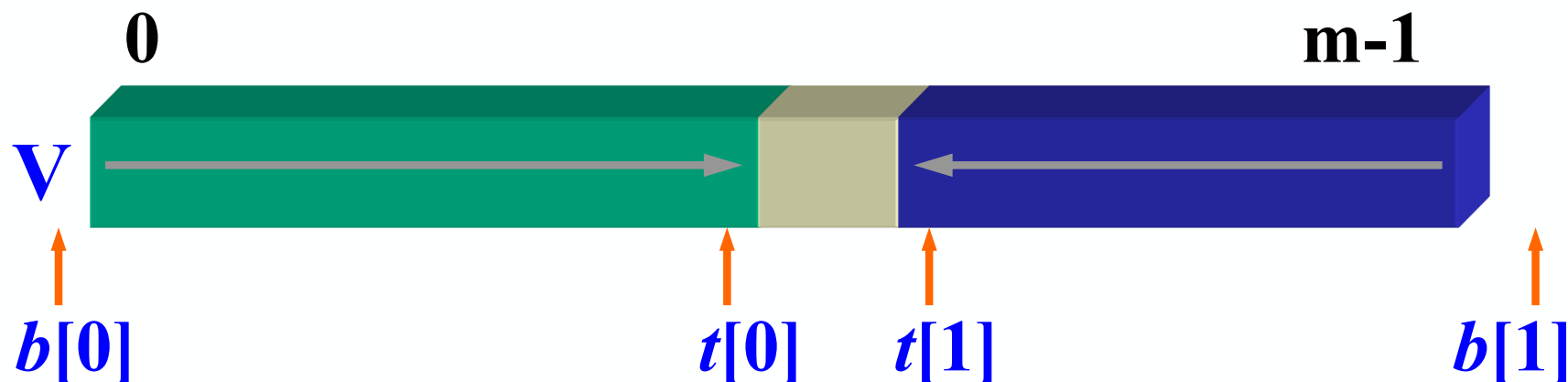
## 【例】模拟打印机缓冲区

- ✓ 在主机将数据输出到打印机时，主机速度与打印机的打印速度不匹配
- ✓ 为打印机设置一个**打印数据缓冲区**，当主机需要打印数据时，先将数据依次写入缓冲区，写满后主机转去做其他的事情
- ✓ 而打印机就从缓冲区中按照**先进先出**的原则依次读取数据并打印



1. 掌握栈和队列的**特点**，并能在相应的应用问题中正确选用
2. 熟练掌握栈的**顺序栈**和链栈的进栈出栈算法，特别注意**栈满和栈空**的条件
3. 熟练掌握**循环队列**和链队列的进队出队算法，特别注意**队满和队空**的条件
4. 掌握**表达式求值**方法

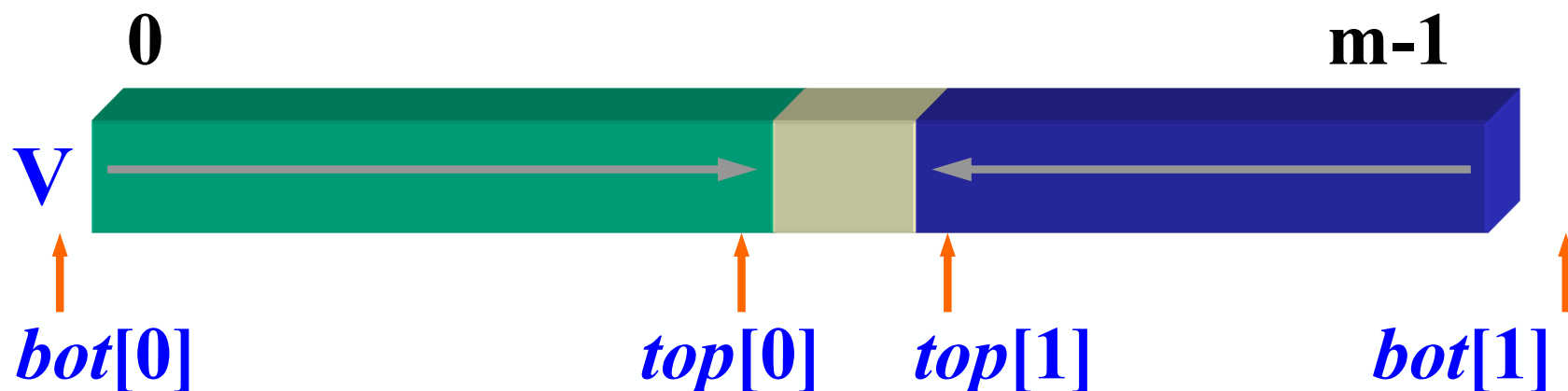
## 双栈共享一个栈空间



优点：互相调剂，灵活性强，减少溢出机会

# 课堂练习

✓将编号为0和1的两个栈存放于一个数组空间  $V[m]$  中，栈底分别处于数组的两端。当第0号栈的栈顶指针  $top[0]$  等于  $-1$  时该栈为空，当第1号栈的栈顶指针  $top[1]$  等于  $m$  时该栈为空。两个栈均从两端向中间增长（如下图所示）。





# 课堂练习

✓数据结构定义如下

```
typedef struct
{
    int top[2], bot[2];    //栈顶和栈底指针
    SElemType *V; //栈数组
    int m;                //栈最大可容纳元素个数
} Db1Stack;
```

# 课堂练习

✓试编写判断栈空、栈满、进栈和出栈四个算法的函数(函数定义方式如下)

```
void Dbllpush(DblStack &s,SElemType x,int i) ;
```

```
//把x插入到栈i的栈
```

```
int Dbllpop(DblStack &s,int i,SElemType &x) ;
```

```
//退掉位于栈i栈顶的元素
```

```
int IsEmpty(DblStack s,int i) ;
```

```
//判栈i空否,空返回1,否则返回0
```

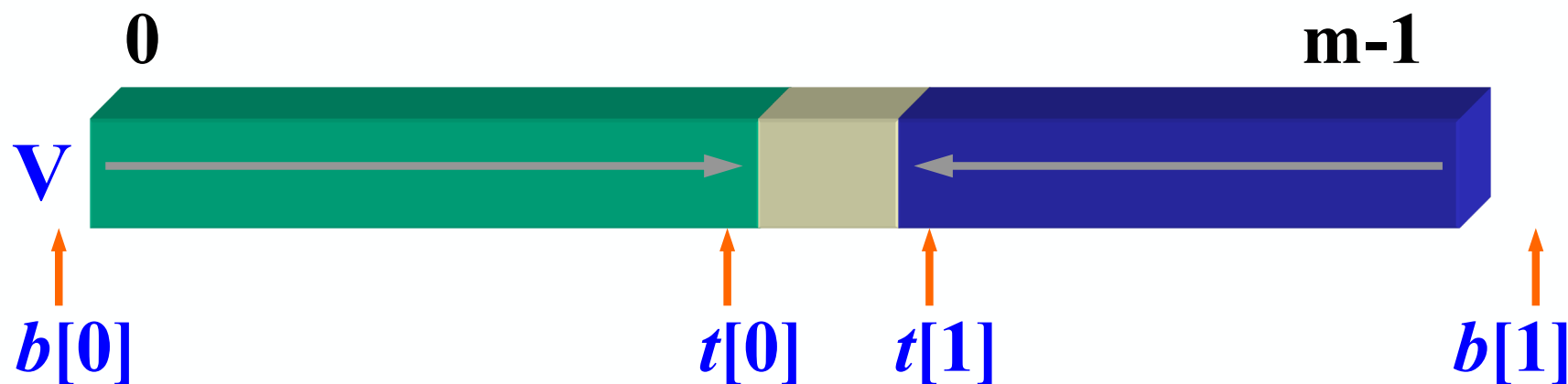
```
int IsFull(DblStack s) ;
```

```
//判栈满否,满返回1,否则返回0
```

# 提示

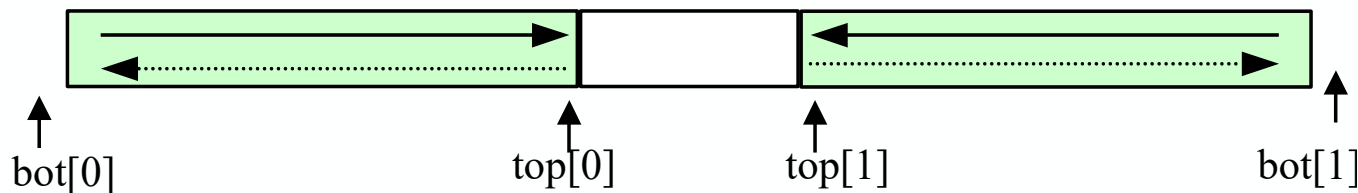
栈空:  $\text{top}[i] == \text{bot}[i]$   $i$ 表示栈的编号

栈满:  $\text{top}[0] + 1 == \text{top}[1]$  或  $\text{top}[1] - 1 == \text{top}[0]$



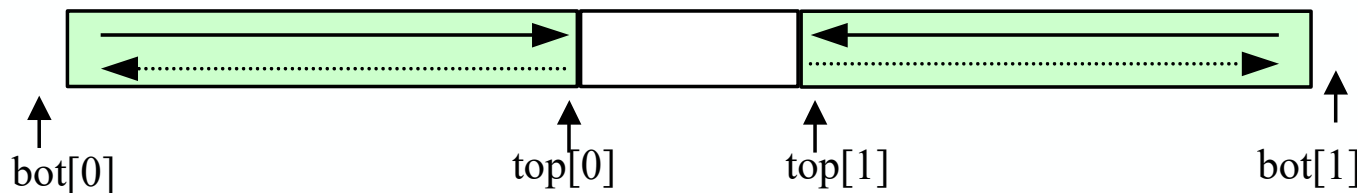
(1) 将编号为0和1的两个栈存放于一个数组空间 $V[m]$ 中，栈底分别处于数组的两端。当第0号栈的栈顶指针 $top[0]$ 等于-1时该栈为空；当第1号栈的栈顶指针 $top[1]$ 等于 $m$ 时，该栈为空。两个栈均从两端向中间增长。试编写双栈初始化，判断栈空、栈满、进栈和出栈等算法的函数。双栈数据结构的定义如下：

```
typedef struct {  
    int top[2], bot[2];    //栈顶和栈底指针  
    SElemType *V;          //栈数组  
    int m;                 //栈最大可容纳元素个数  
} Db1Stack;
```



(1) 将编号为0和1的两个栈存放于一个数组空间 $V[m]$ 中，栈底分别处于数组的两端。当第0号栈的栈顶指针 $top[0]$ 等于-1时该栈为空；当第1号栈的栈顶指针 $top[1]$ 等于 $m$ 时，该栈为空。两个栈均从两端向中间增长。试编写双栈初始化，判断栈空、栈满、进栈和出栈等算法的函数。双栈数据结构的定义如下：

```
typedef struct {  
    int top[2], bot[2];    //栈顶和栈底指针  
    SElemType *V;          //栈数组  
    int m;                 //栈最大可容纳元素个数  
} Db1Stack;
```



//初始化一个大小为m的双向栈s

**Status Init\_Stack(DblStack &s,int m)**

```
{  
    s.V=new SElemType[m];  
    s.bot[0]=-1;  
    s.bot[1]=m;  
    s.top[0]=-1;  
    s.top[1]=m;  
    return OK;  
}
```

//判栈i空否,空返回1,否则返回0

**int IsEmpty(DblStack s,int i)**

**{return s.top[i] == s.bot[i]; }**

//判栈满否,满返回1,否则返回0

**int IsFull(DblStack s)**

**{ if(s.top[0]+1==s.top[1]) return 1;**

**else return 0;}**

```
void Dbllpush(DblStack &s,SElemType x,int i)
{
    if( IsFull (s ) ) exit(1);
        // 栈满则停止执行
    if ( i == 0 ) s.V[ ++s.top[0] ] = x;
        //栈0情形： 栈顶指针先加1, 然后按此地址进栈
    else s.V[--s.top[1]]=x;
        //栈1情形： 栈顶指针先减1, 然后按此地址进栈
}
```



```
int Dbllpop(DblStack &s,int i,SElemType &x)
{if ( IsEmpty ( s,i ) ) return 0;
    //判栈空否,若栈空则函数返回0
    if ( i == 0 ) s.top[0]--; //栈0情形: 栈顶指针减1
    else s.top[1]++; //栈1情形: 栈顶指针加1
    return 1;
}
```