

# 第3章 栈和队列

3.1 栈

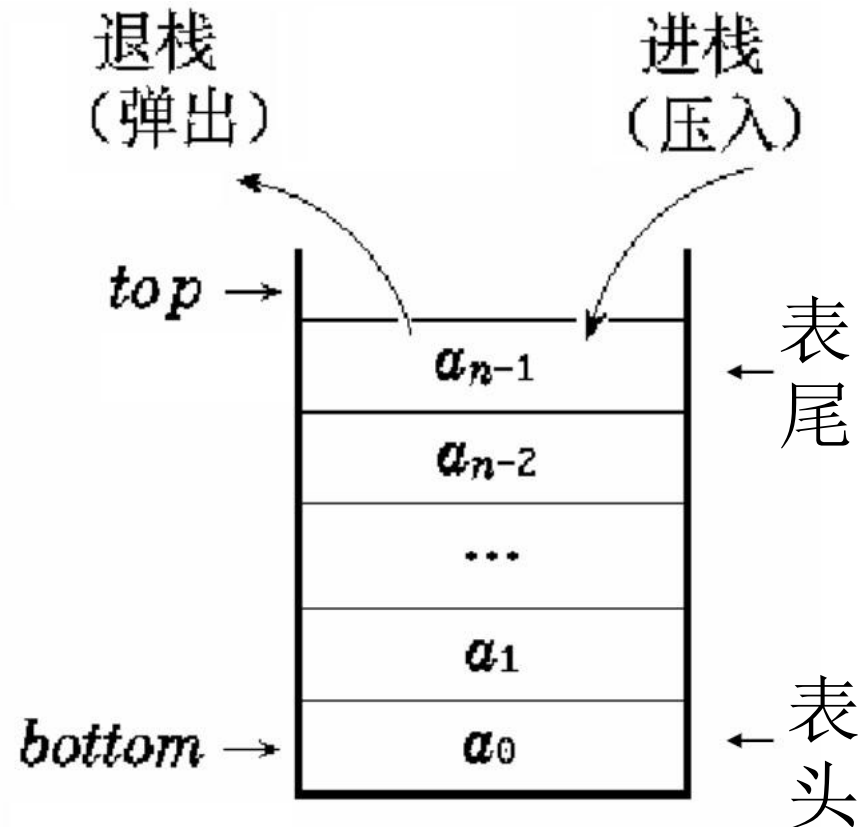
3.2 栈的应用举例

3.3 队列

# 3.1 栈 ( Stack )

■ 1. 定义：限定只在表的一端(表尾)进行插入和删除操作的线性表

- 特点: **后进先出(LIFO)**
- **栈顶(top)**:
- 允许插入和删除的一
- **栈底(bottom)**: 另一端



## 栈的抽象数据类型定义：

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

约定  $a_n$  端为栈顶，  $a_1$  端为栈底

基本操作： 见下页

}ADT Stack

# 栈的基本操作

- ① `InitStack(&s)` 构造一个空栈s。
- ② `StackEmpty(s)` 若s为空栈，返回1，否则返回0。
- ③ `Push(&s, x)` 进栈。插入x为栈顶元素。
- ④ `Pop(&s, x)` 退栈。若s非空，删除栈顶元素，并用x返回其值。
- ⑤ `GetTop(s)` 返回栈顶元素。
- ⑥ `ClearStack (&s)` 将栈s清为空栈。
- ⑦ `StackLength(s)` 返回栈s的元素个数。
- ⑧ `DestroyStack(&s)` 销毁栈
- ⑨ `StackTraverse(S,visit())` 遍历栈

# 3.1 栈

## ■ 2. 栈的表示和实现

1) 顺序栈—栈的顺序存储结构

2) 链栈—栈的链式存储结构

3) 静态分配整型指针

# 顺序栈—栈的顺序存储结构

- 限定在表尾进行插入和删除操作的顺序表

- 类型定义：

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

```
SqStack  s;
```

## ■ base

- 栈底指针，始终指向栈底；
- 当base == NULL时，表明栈结构不存在。

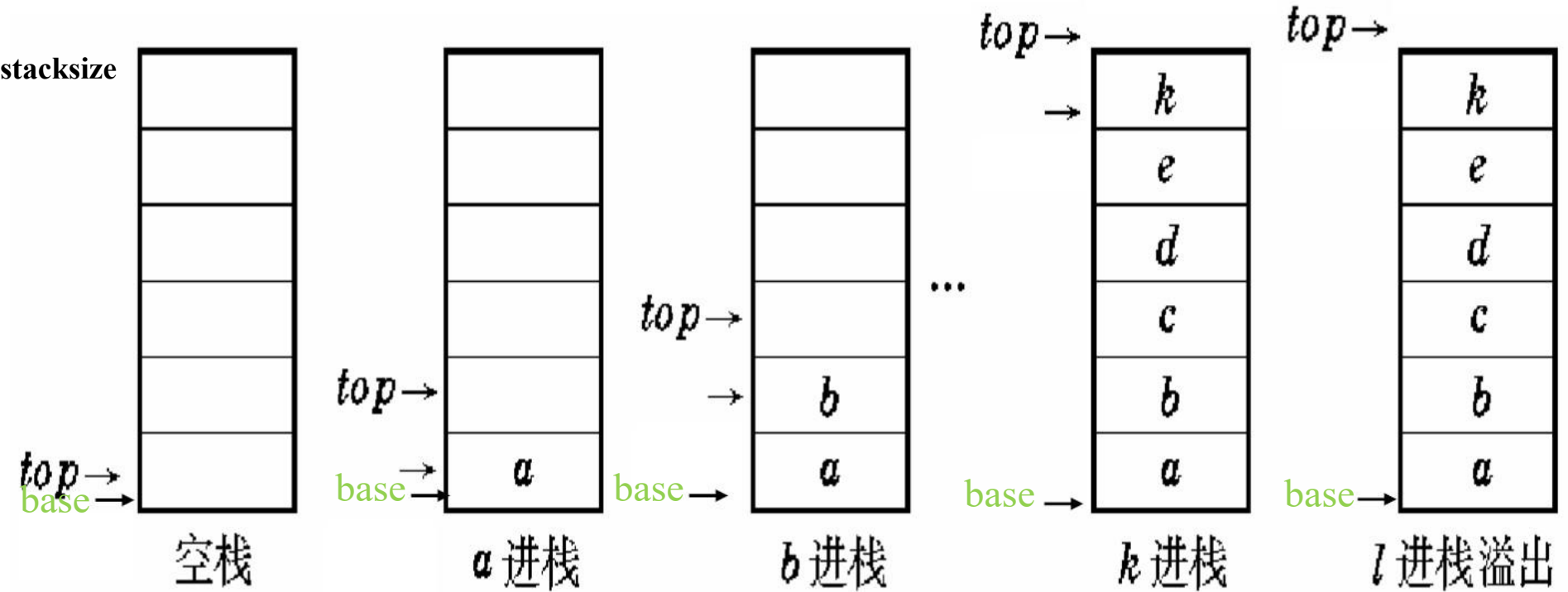
## ■ top

- 栈顶指针
  - a) top的初始值指向栈底，即top=base
  - b) 空栈：当top=base时为栈空的标记
  - c) 当栈非空时，top的位置：指向当前栈顶元素的下一个位置

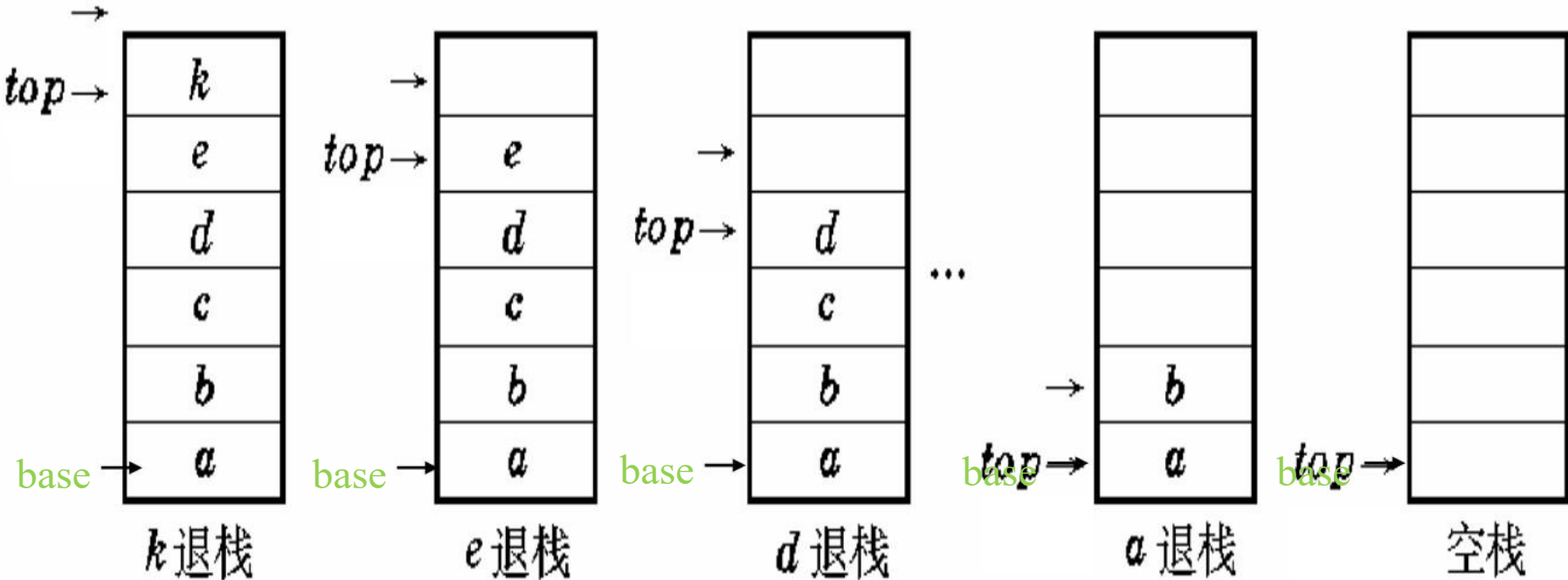
## ■ stacksize

- 当前栈可使用的最大容量

# 进栈示例



# 退栈示例



## ■ 判栈空：

- `s.top == s.base`

## ■ 判栈满：

- `s.top - s.base >= s.stacksize`

## ■ 进栈：

- `*s.top++ = e; 或 *s.top = e; s.top++;`

## ■ 退栈：

- `e = *--s.top; 或 s.top--; e = *s.top;`

## ■ 上溢：

- 当栈满时再做进栈运算必定产生空间溢出

## ■ 下溢：

- 当栈空时,再做退栈运算也将产生溢出

# 基本操作

- 栈的初始化操作

Status InitStack(SqStack &S)

- 取栈顶元素

Status GetTop(SqStack S, SElemType &e)

- 进栈操作

Status Push(SqStack &S, SElemType e)

- 退栈操作

Status Pop(SqStack &S, SElemType &e)

# 栈的初始化操作

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

# 取栈顶元素

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

# 进栈操作

```
Status Push(SqStack &S, SElemType e){
    if (S.top-S.base>=S.stacksize)
        { S.base=(SElemType*)realloc(S.base,
        (S.stacksize+STACKINCREMENT)*sizeof(SElemType
        e));
            if (!S.base) return (OVERFLOW);
            S.top = S.base +S.stacksize;
            S.stacksize += STACKINCREMENT;
        }
    *S.top++ = e;// *S.top=e; S.top=S.top+1;
    return OK;
}
```

# 退栈操作

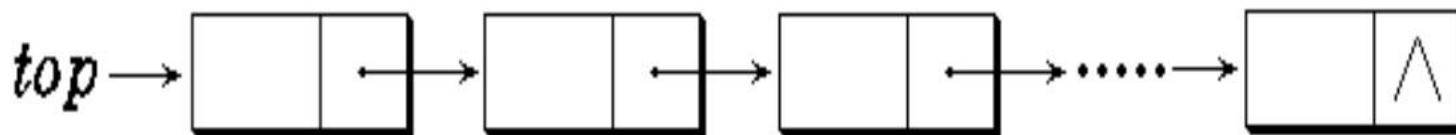
```
Status Pop(SqStack &S, SElemType &e)
{
    if (S.top == S.base) return ERROR;
    e=*--S.top; // S.top--; e=*S.top;
    return OK;
}
```

# 链栈——栈的链式存储结构

- 不带头结点的单链表，其插入和删除操作仅限制在表头位置上进行。
- 链表的头指针即栈顶指针。
- 类型定义：

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

## ■ 链栈示意图



## ■ 栈空条件:

- $s == \text{NULL}$

## ■ 栈满条件:

- 无 / 无Free Memory可申请

# 进栈操作

```
Status Push_L (LinkStack &s, SElemType e)
{
    p=(LinkStack)malloc(sizeof(SNode));
    if (!p) exit(Overflow);
    p->data = e; p->next = s; s=p;
    return OK;
}
```

# 退栈操作

```
Status Pop_L (LinkStack &s, SElemType &e)
{ if (!s) return ERROR;
  e=s->data; p=s; s=s->next;
  free(p);
  return OK;
}
```

# 静态数组

## ■ 类型定义

```
#define MAXSIZE 100
typedef struct {
    SElemType base[MAXSIZE];
    int top;
}SqStack;
SqStack s;
```

# 初始化

```
Status InitStack(SqStack &s)
{
    s.top = 0;
    return OK;
}
```

# 进栈

```
Status Push(SqStack &s,SElemType e)
{ if (s.top == MAXSIZE) return ERROR;
  s.base[s.top] = e; s.top++;
  return OK;
}
```

# 退栈

```
Status Pop(SqStack &s,SElemType &e)
{ if (s.top ==0) return ERROR;
  s.top--;
  e=s.base[s.top];
  return OK;
}
```

# 取栈顶元素

```
Status GetTop(SqStack s, SElemType &e)
{
    if (s.top == 0) return ERROR;
    e=s.base[s.top-1];
    return OK;
}
```

## 3.2 栈的应用

- 1. 数制转换
- 2. 行编辑程序
- 3. 表达式求值



## ■ //算法3.1

```
void Conversion(){
    InitStack(s);
    scanf("%d,%d",&N,&base);
    NI=N;
    while (NI){
        Push(s,NI%base);
        NI = NI/base;  }
    while (!(StackEmpty(s)){
        Pop(s,e);
        if (e>9) printf("%c",e+55);
        else printf("%c",e+48);  }
    printf("\n");
}
```

## (2) 行编辑程序

### ■ 简单行编辑程序的功能：

- 接受用户从终端输入的程序或数据，并存入用户的数据区。

### ■ 做法：

- 设立一个输入缓存区，用以接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入时出差错。如：可用一个退格符“#”，删除前一个字符；可用一个退行符“@”，删除一行。

### ■ 实现：

- 设这个缓存区为一个栈结构，每当从终端接受一个字符后作如下判别：

如果既不是退格符也不是退行符，则压栈；

如果是一个退格符，则从栈顶删去一个字符；

如果是一个退行符，则将字符栈清为空栈。

## ■ //LineEdit算法3.2

```
void LineEdit( ){ InitStack(s);
    char ch=getchar( ),c;
    while(ch!=EOF){ //^F或文件尾
        while(ch!=EOF && ch!='\n'){//while 2
            switch(ch) {
                case '#' : Pop(s,c);break;
                case '@' : ClearStack(s); break;
                default : Push(s,ch);
            }//end switch
            ch=getchar( ); }//end while 2
//把从栈底到栈顶的栈内字符传送到调用过程的数据区;
    ClearStack(s);
    if(ch!=EOF)    ch=getchar( );
    }//end while
    DestroyStack(s);
}
```

### (3) 表达式求值

#### ■ 算符优先法

- 假定运算符有：+、-、\*、/、（、）、#
- 运算的规则：  
 $A/B * C + D * E - A * C$  应理解为：  
 $((A/B) * C) + (D * E) - (A * C)$ 
  - 先乘除，后加减；
  - 从左算到右
  - 先括号内，后括号外。
- 算符的优先级：
  - \*/的优先级高于+-；
  - 相同算符先出现的优先级高；
  - 左括号：比括号内的算符的优先级低，比括号外的算符的优先级高
  - 右括号：比括号内的算符的优先级低，比括号外的算符的优先级高
  - #：表达式的结束符，优先级总是最低

## ■算符间的优先级关系：（表3.1）

$\Theta_1 \backslash \Theta_2$	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

$\Theta_1 < \Theta_2$ :  $\Theta_1$ 的优先权低于  $\Theta_2$

$\Theta_1 > \Theta_2$ :  $\Theta_1$ 的优先权高于  $\Theta_2$

$\Theta_1 = \Theta_2$ :  $\Theta_1$ 的优先权等于  $\Theta_2$

## ■ 中缀表达式

- 运算符在操作数中间
- 需要括号改变优先级, 例如:  $4+6*(5+7)/8$

## ■ 后缀表达式 (逆波兰式)

- 运算符在操作数后面
- 不需要括号, 例如:  $4\ 6\ 5\ 7\ +\ *\ 8\ /\ +$

## ■ 前缀表达式 (波兰式)

- 运算符在操作数前面
- 不需要括号, 例如:  $\ +\ 4\ /\ *\ 6\ +\ 5\ 7\ 8$

## ■ 对于中缀表达式, 为实现算符优先算法, 需使用两个工作栈:

- OPND栈: 存数据或运算结果
- OPTR栈: 存运算符

# 算法思想

## ■ 1. 初态:

- 置OPND栈为空；“#”作为OPTR栈的栈底元素

## ■ 2. 依次读入表达式中的每个字符

- 1) 若是操作数，则进入OPND栈；
- 2) 若是运算符，则与OPTR栈的栈顶运算符进行优先权（级）的比较：
  - 若读入运算符的优先权高，则进入OPTR栈；
  - 若读入运算符的优先权低，则OPTR退栈1次（退出栈顶元素），OPND栈退栈2次，（先退出b，再退出a），计算结果再进入OPND栈；继续与optr栈顶元素比较
  - 若读入“）”，OPTR栈的栈顶元素若为“（”，则OPTR退出“（”；
  - 若读入“#”，OPTR栈栈顶元素也为“#”，则OPTR栈退出“#”，结束。

## ■例：3\* (7-2) 求解过程如下：

步骤	OPTR栈	OPND栈	输入字符	主要操作
1	#		3* (7-2) #	PUSH (OPND, '3')
2	#	3	* (7-2) #	PUSH (OPTR, '*' )
3	#*	3	(7-2) #	PUSH (OPTR, '(' )
4	#* (	3	7-2) #	PUSH (OPND, '7')
5	#* (	3 7	-2) #	PUSH (OPTR, '-')
6	#* (-	3 7	2) #	PUSH (OPND, '2')
7	#* (-	3 7 2	) #	operate('7', '-', '2')
8	#*(	3 5	)#	POP(OPTR)
9	#*	3 5	#	operate('3', '*', '5')
10	#	15	#	RETURN(GETTOP(OPND))

```

SElemType EvaluateExpression() {
    InitStack(OPTR);      Push(OPTR,'#'); /* #是表达式结束标志 */
    InitStack(OPND); c=getchar();      GetTop(OPTR,x);
    while(c!='#'||x!='#') {
        if(In(c)) /* 是7种运算符之一 */
            switch(Precede(x,c)) {
                case '<': Push(OPTR,c); /* 栈顶元素优先权低 */
                    c=getchar(); break;
                case '=': Pop(OPTR,x); /* 脱括号并接收下一字符 */
                    c=getchar(); break;
                case '>': Pop(OPTR,theta); /* 退栈并将运算结果入栈 */
                    Pop(OPND,b); Pop(OPND,a);
                    Push(OPND,Operate(a,theta,b));
            } else if(c>='0'&&c<='9') /* c是操作数 */
                { /*将读入字符转换成整数d*/ Push(OPND,d); c=getchar(); }
            else /* c是非法字符,非法输入处理*/
                GetTop(OPTR,x);} /*while*/ GetTop(OPND,x); return x;}

```

Eg.将中缀表达式 “ $1+((2+3)\times 4)-5$ ” 转换为后缀表达式之后  
求值：

扫描到的元素	S2(栈底->栈顶)	S1 (栈底->栈顶)	说明
1	1	空	数字，直接入栈
+	1	+	S1为空，运算符直接入栈
(	1	+ (	左括号，直接入栈
(	1	+ ((	同上
2	1 2	+ ((	数字
+	1 2	+ (( +	S1栈顶为左括号，运算符直接入栈
3	1 2 3	+ (( +	数字
)	1 2 3 +	+ (	右括号，弹出运算符直至遇到左括号
$\times$	1 2 3 +	+ ( $\times$	S1栈顶为左括号，运算符直接入栈
4	1 2 3 + 4	+ ( $\times$	数字
)	1 2 3 + 4 $\times$	+	右括号，弹出运算符直至遇到左括号
-	1 2 3 + 4 $\times$ +	-	-与+优先级相同，因此弹出+，再压入-
5	1 2 3 + 4 $\times$ + 5	-	数字
到达最右端	1 2 3 + 4 $\times$ + 5 - ( 输出结果，注意逆序输出)	空	S1中剩余的运算符

# 思考题

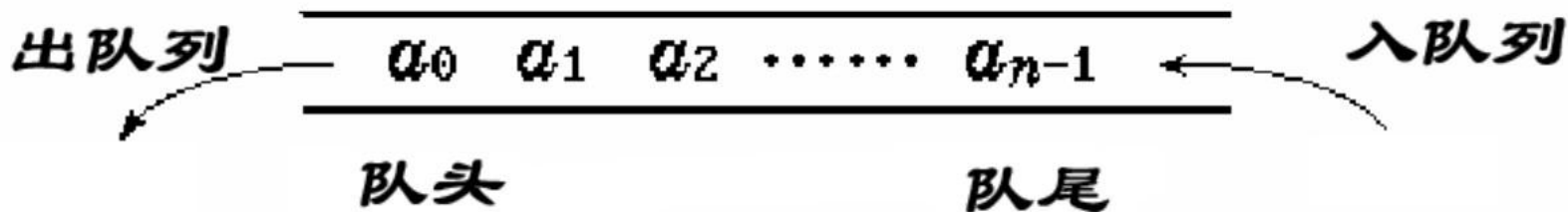
- 1. 如果依次输入A,B,C,D等字符，通过栈的调度，分别实现以下的输出序列：  
(1) BDCA;(1)CBDA.
- 2. 请设计一个栈，除了pop和push函数，还支持min函数，即可返回栈中的最小值。要求执行所有操作的时间复杂度必须是 $O(1)$ 。
- 3. 设 $A=\{1,2,3,\dots,n\}$ 为入栈的顺序，B为A的任一个排列，试证明：若B能通过栈的调度得到，当且仅当对于任意 $1 \leq i < j < k \leq n$ , 该排列中都不存在如下模式：  
 $\{\dots,k,\dots,i,\dots,j,\dots\}$

## 3.3 队列

- 1. 定义
- 2. 链队列——队列的链式存储结构
- 3. 循环队列——队列的顺序存储结构

# 1. 定义

- 队列是限定在表的一端进行删除，在表的另一端进行插入操作的线性表。
- 允许删除的一端叫做队头(front)，允许插入的一端叫做队尾(rear)。
- 特性：FIFO(First In First Out)



队列的抽象数据类型定义：

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

约定  $a_n$  端为队尾尾，  $a_1$  端为队头

基本操作： 见下页

}ADT Queue

# 队列的基本操作

- ① `InitQueue(&Q)` 构造一个空队列Q。
- ② `EmptyStack(Q)` 若Q为空，返回1，否则返回0。
- ③ `EnQueue(&Q,x)` 进队,插入x为队尾元素。
- ④ `DeQueue(&Q,x)` 出队，若Q非空，删除Q的队头元素，并用x返回。
- ⑤ `GetHead(Q,&x)` 用x返回Q的队头元素。
- ⑥ `DestroyQueue(&Q)` 销毁队列Q
- ⑦ `ClearQueue(&Q)` 将队列Q清空
- ⑧ `QueueLength(&Q)` 返回Q的元素个数
- ⑨ `QueueTraverse(Q,visit())` 遍历队列

# 队列的表示和实现

- 1) 链队列——队列的链式存储结构
- 2) 循环队列——队列的顺序存储结构

## 2. 链队列—队列的链式存储结构

- 实质是带头结点的线性链表
- 两个指针：
  - 队头指针Q.front指向头结点
  - 队尾指针Q.rear指向尾结点
- 初始态：队空条件
  - 头指针和尾指针均指向头结点
  - $Q.front = Q.rear$

# 1) 链队列的类型定义

```
typedef struct QNode { //元素结点
    QElemType data;
    struct QNode *next;
}QNode, *QueuePtr;
```

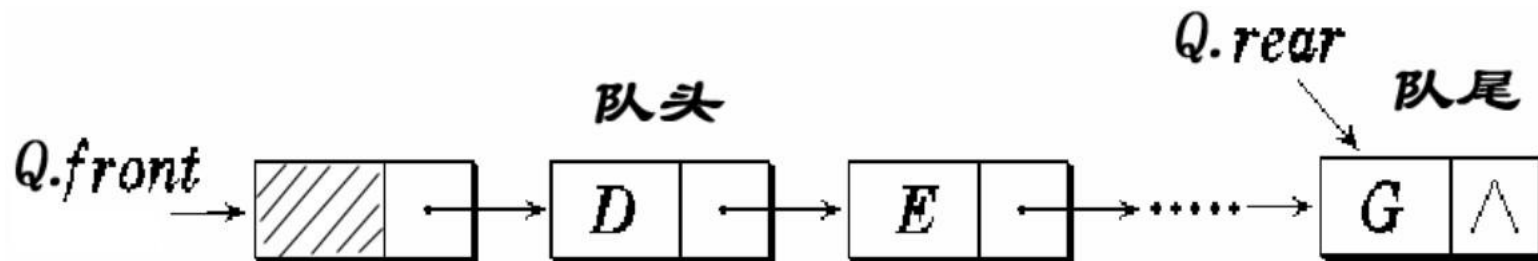
```
typedef struct{           //特殊结点
    QueuePtr front;      //队头指针
    QueuePtr rear;      //队尾指针
}LinkQueue;
```

```
LinkQueue Q;
```

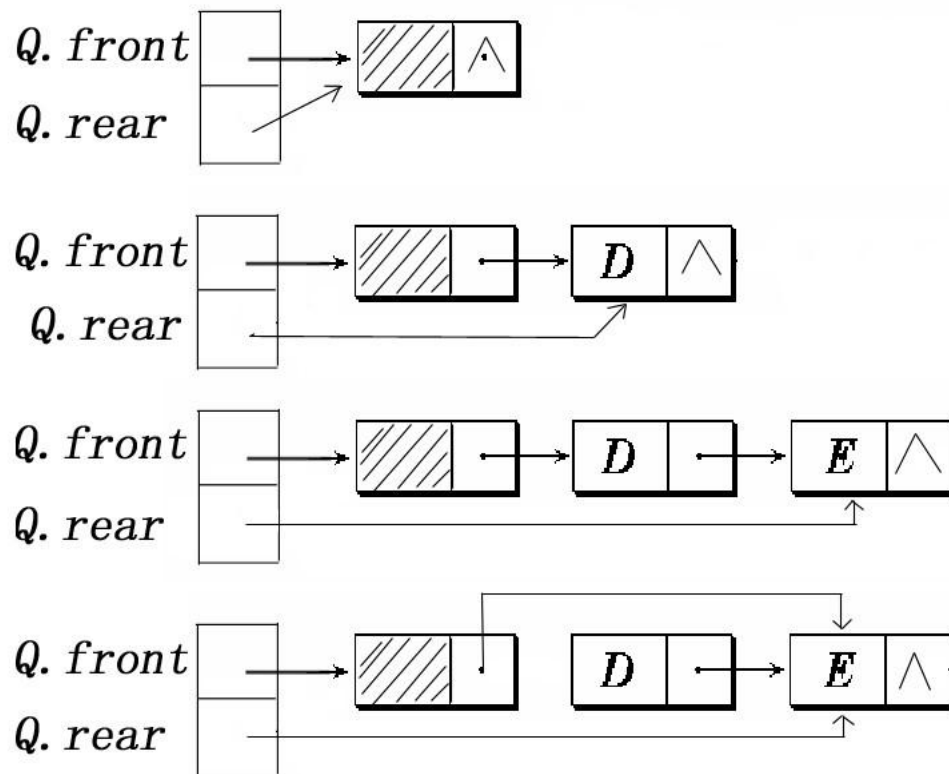
Q.front——指向链头结点

Q.rear ——指向链尾结点

## 2) 链队列示意图



队列运算指针变化状况



### 3) 基本操作与实现

#### ■ 初始化

- Status InitQueue (LinkQueue &Q)

#### ■ 销毁队列

- Status DestroyQueue (LinkQueue &Q)

#### ■ 入队

- Status EnQueue(LinkQueue &Q, QElemType e)

#### ■ 出队

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

#### ■ 判队空

- Status QueueEmpty(LinkQueue Q)

#### ■ 取队头元素

- Status GetHead(LinkQueue Q, QElemType &e)

# 链队列初始化

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

# 链队列的入队（插入）

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

# 链队列的出队（删除）

```
Status DeQueue (LinkQueue &Q, ElemType &e){  
    if (Q.front==Q.rear) return ERROR;  
        p=Q.front->next;  
        e=p->data;  
        Q.front->next=p->next;  
        if (Q.rear == p) Q.rear=Q.front;  
        free(p);  
        return OK;  
}
```

# 判断链队列是否为空

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

# 取链队列的第一个元素结点

```
Status GetHead(LinkQueue Q, QElemType &e){  
    if (QueueEmpty(Q)) return ERROR;  
    e=Q.front->next->data;  
    return OK;  
}
```

# 链队列的销毁

```
Status DestroyQueue (LinkQueue &Q){  
    while (Q.front)  
    { Q.rear=Q.front->next;  
      free(Q.front);  
      Q.front=Q.rear;  
    }  
    return OK;  
}
```

# 3. 循环队列—队列的顺序存储结构

## ■ 顺序队列：

- 用一组地址连续的存储单元依次存放从队列头到队列尾的元素

## ■ 需要设两个指针：

- Q.front 指向队列头；
- Q.rear 指向队列尾；

## ■ 初始状态：队空

## ■ 需要设定队列的最大容量

# 类型定义

- 回顾顺序栈的定义：

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

- 思考：能否将上述定义中将\*base当作\*front，\*top当作\*rear来作为队列的类型定义呢？

# 类型定义

- 令front, rear为整型，指示队列头和尾的下标。
- 循环队列的类型定义如下：

```
#define MAXSIZE 100 //最大队列长度
typedef struct {
    QElemType *base;
    int front;
    int rear;
    int queuesize;
}SqQueue;
SqQueue Q;
```

# 顺序队列的初始设定

## ■ 初始态:

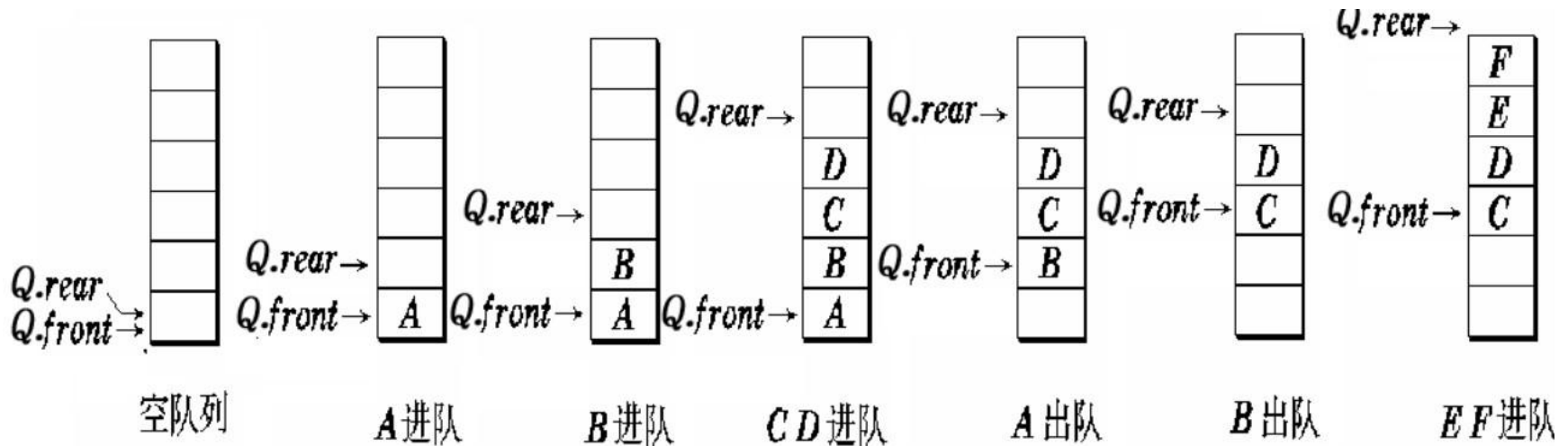
- $Q.front=Q.rear=0$

## ■ 入队: 在队尾添加元素, 并且 $Q.rear$ 后移

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

## ■ 出队: 在队头删除元素, 并且 $Q.front$ 后移

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



# 队列的状态及指针的含义

## ■ 指针的含义：

- Q.rear: 虚指，指向队尾元素的下一个位置
- Q.front: 实指，指向队头元素的位置

## ■ 队列的状态

(1) 队满：当  $Q.rear \geq Q.queue\_size$  时

- 上溢：队满再进队将溢出出错；
- 上图为“假满”

(2) 队空：  $Q.rear == Q.front$

- 下溢：队空时再出队需作队空处理。

# 循环队列

## ■ 如何解决队列“假满”问题？

- 将数组当成一个首尾相接的表处理。
- 取模运算

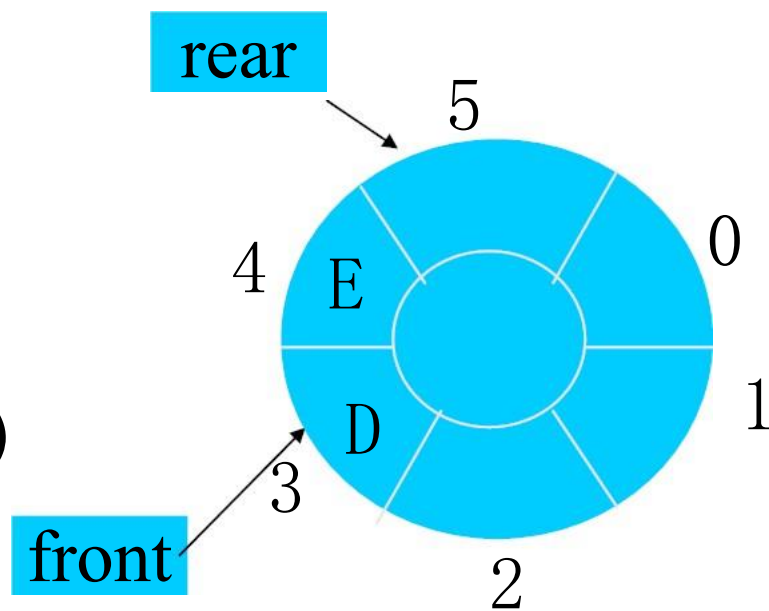
## ■ 初始： $Q.front=Q.rear=0$

## ■ 入队：

- $Q.base[Q.rear]=e;$
- $Q.rear=(Q.rear+1)(\text{模}N)$

## ■ 出队：

- $e=Q.base[Q.front];$
- $Q.front=(Q.front+1)(\text{模}N)$



# 举例

- (1) 初始:  $Q.front=Q.rear=0$
- (2) J1, J2, J3, J4, J5 入队, J1, J2, J3 出队

- $front=3, rear=5$

- (3) 从图2, 继续入队 J6, J7, J8 后

- $front=3, rear=2,$

- 再放入 J9 后,  $front==rear$

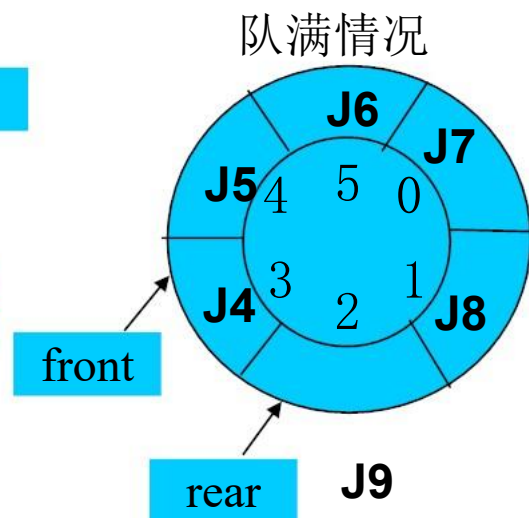
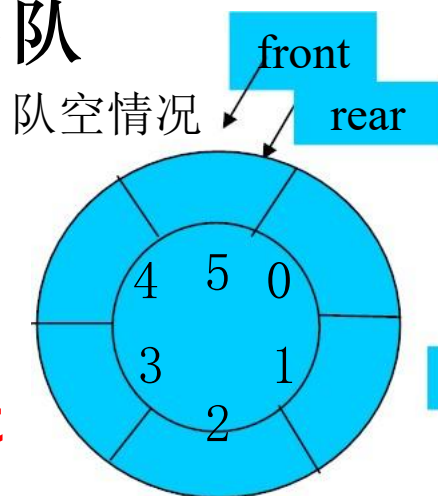
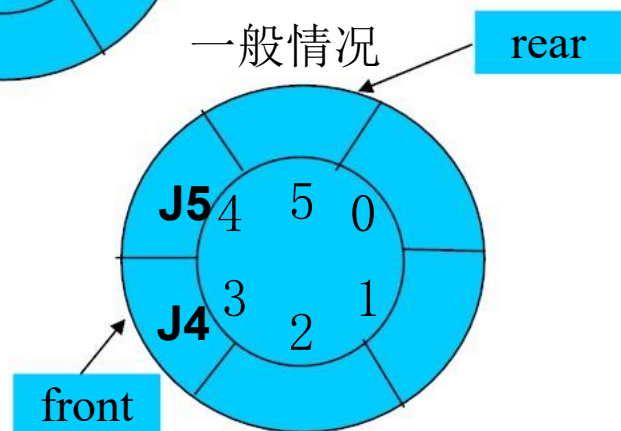
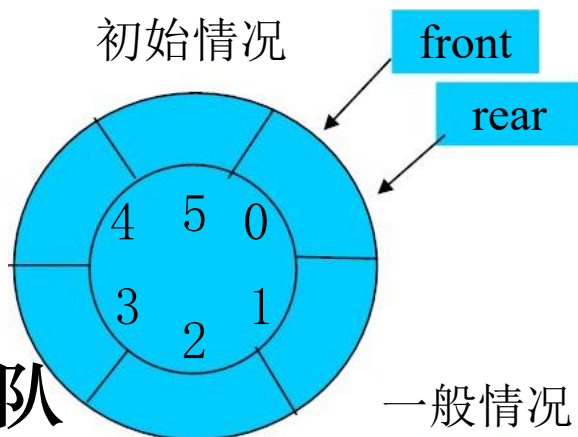
- (4) 从图2, 连续将 J4, J5 出队

- $front=rear=5$

- 如何区分队空和队满?

- $front==rear?$

- $(rear+1) \text{ 模 } N == front$



# 循环队列 (Circular Queue)

- 存储队列的数组被当作首尾相接的表处理。
- 队头、队尾指针加1时从maxsize -1直接进到0，可用语言的取模(余数)运算实现。
- 队头指针进1:  $Q.front = (Q.front + 1) \% MAXSIZE$
- 队尾指针进1:  $Q.rear = (Q.rear + 1) \% MAXSIZE$ ;
- 队列初始化:  $Q.front = Q.rear = 0$ ;
- 队空条件:  $Q.front == Q.rear$ ;
- 队满条件:  $(Q.rear + 1) \% MAXSIZE == Q.front$
- 队列长度:  $(Q.rear - Q.front + MAXSIZE) \% MAXSIZE$

# 说明

- 不能用动态分配的一维数组来实现循环队列，初始化时必须设定一个最大队列长度。
- 循环队列中要有一个元素空间浪费掉
  - 约定：队列头指针在队列尾指针的下一位置上为“满”的标志
- 解决  $Q.front=Q.rear$  不能判别队列“空”还是“满”的其他办法：
  - 使用一个计数器记录队列中元素的总数（即队列长度）
  - 或者，
  - 设一个标志变量以区别队列是空或满

# 基本操作

## ■ 初始化

- Status InitQueue (SqQueue &Q)

## ■ 求队列的长度

- int QueueLength(SqQueue Q)

## ■ 入队

- Status EnQueue (SqQueue &Q, QElemType e)

## ■ 出队

- Status DeQueue(SqQueue &Q, QElemType &e)

## ■ 判队空

- Status QueueEmpty(SqQueue Q)

## ■ 取队头元素

- Status GetHead(SqQueue Q, QElemType &e)

# 初始化

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

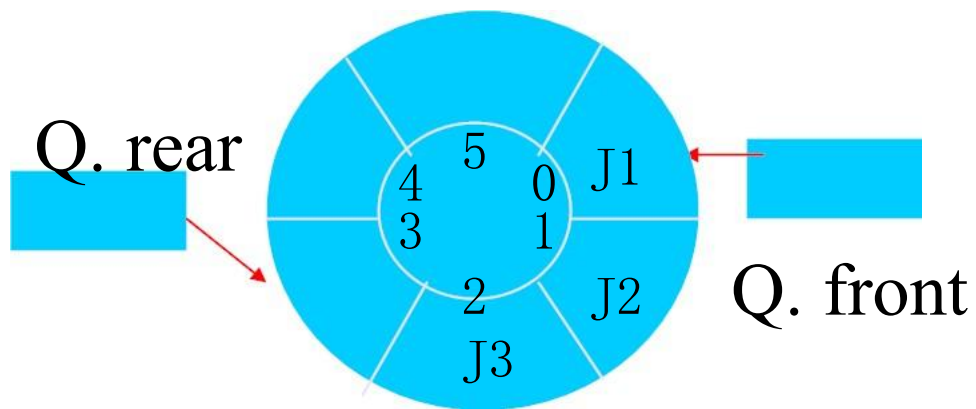
# 求队列长度

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

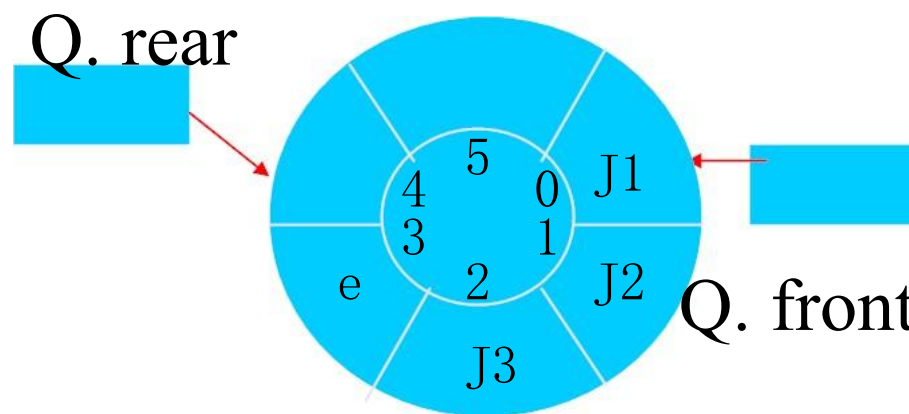
# 入队

```
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;  
}
```

入队操作：将元素  $e$  插入队尾



元素  $e$  入队前

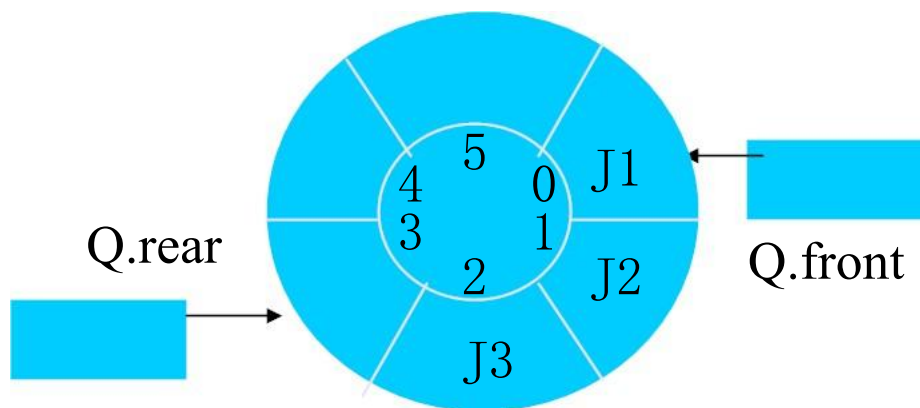


元素  $e$  入队后

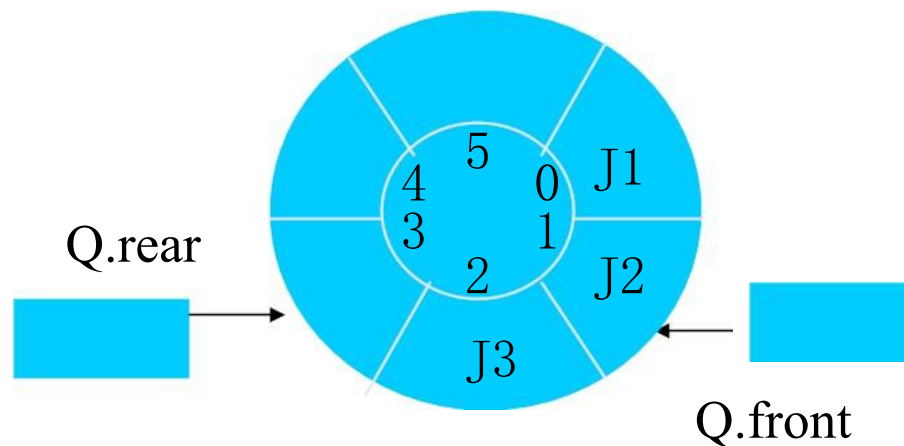
# 出队

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

出队操作：删除队头元素



出队操作前



出队操作后

# 判队空

```
Status QueueEmpty(SqQueue Q){  
    if (Q.front==Q.rear) return TRUE;  
    return FALSE;  
}
```

# 取队头元素

```
Status GetHead(SqQueue Q, QElemType &e){  
    if QueueEmpty(Q) return ERROR;  
    e=Q.base[Q.front];  
    return OK;  
}
```

# 非循环队列

- 类型定义：与循环队列完全一样
- 关键：修改队尾/队头指针
  - $Q.rear = Q.rear + 1;$
  - $Q.front = Q.front + 1;$
- 在判断时，有 $\%MAXQSIZE$ 为循环队列，否则为非循环队列
- 队空条件：  $Q.front == Q.rear$
- 队满条件：  $Q.rear \geq MAXQSIZE$
- 注意“假上溢”的处理
- 长度：  $Q.rear - Q.front$

# 4. 队列的应用举例

- 1) 解决计算机主机与外设不匹配的问题，如解决高速CPU与低速打印设备之间速度不匹配问题；
- 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

# 银行服务事件模拟

## ■ 银行排队问题模拟

- 程序由**事件驱动**，事件包括顾客到达事件和离开事件；所有事件按照时间顺序存放在一个有序列表中；
- 初始状态：事件表中含有一个事件，即第一个顾客到达事件，时间为0。
- 到达事件的处理：随机产生该顾客的**服务时间**以及下一个顾客的**到达时间**，将该顾客加入队列长度最小的队列中等待处理，若该顾客是队头元素，则产生一个离开事件加入到事件列表中，将下一个顾客的到达事件插入到事件有序列表中。

# 银行服务事件模拟

- 离开事件的处理：计算客户逗留时间，从队列中删除该客户，产生下一个队头元素的离开事件。
- 因此需要用到2种数据结构：
1. 事件列表(有序表):主要操作是插入和删除；
  2. 队列:队列中的主要信息是客户到达时刻和办理事务的时间

# 数据结构

```
typedef struct{
    int OccurTime;//事件发生时刻
    int NType; //0:到达，其他是窗口号，离开
}Event,ElemType; //放入有序链表中
typedef LinkList EventList;//事件列表
typedef struct{
    int ArriveTime; //下一个顾客的到达时刻
    int Duration; //事务处理时间
    int no;//客户编号
}QElemType;//客户信息，进入队列
EventList ev; Event en; //事件列表和事件
queue<QElemType> q[5]; //4个客户队列
QElemType customer; //客户记录
int TotalTime, CustomerNum; //累计逗留时间
int intertime, durtime, CloseTime;
```

事件驱动代码

```
void Bank_Simulation(int CloseTime){
    OpenForDay();//设定第一个客户到达事件
    while (!ListEmpty(ev)){//事件表不空
        DelFirst(GetHead(ev),p);
        en=GetCurElem(p);//删除第一个事件
        if (en.NType==0)//处理到达事件
            CustomerArrived();
        else //处理离开事件
            CustomerDeparture();
    }
    //计算并输出平均逗留时间
    cout<<"The average time is
    "<<(double)TotalTime/CustomerNum;
}
```

■ //初始设置，产生第一个到达事件

```
void OpenForDay(){  
    TotalTime=0;CustomerNum=0;  
    InitList(ev);  
    en.OccurTime =0;en.NType=0;  
    OrderInsert(ev,en,cmp);  
}
```

■ 第0时刻到达第一个客户，插入事件列表

客户到达事件处理

```
void CustomerArrived(){
    ++CustomerNum;
    intertime=rand()%5+1;//1~5
    durtime=rand()%30+1;//1~30
    int t=en.OccurTime+intertime;
    if (t<CloseTime)
        OrderInsert(ev,{t,0},cmp);
    int i=Minimum();//最小队列
    q[i].push({en.OccurTime,durtime,Customer
Num});
    if (q[i].size()==1)/*队头元素，得到一个离开事件*/
        OrderInsert(ev,{en.OccurTime
+durtime,i},cmp);
}
```

客户离开事件处理

```
void CustomerDeparture(){
    int i=en.NType;
    customer=q[i].front();//取队头元素
    q[i].pop();//出队
    TotalTime +=en.OccurTime-
customer.ArriveTime;//一个客户任务完成
    if (!q[i].empty()){
        customer=q[i].front();

        OrderInsert(ev,{en.OccurTime+customer
.Duration,i},cmp);
    }
}
```

## ■ 运行结果:

- ①  $t = 0$ 分钟时, 第1个Customer Arrived 进入第 1 个队列
- ②  $t = 2$ 分钟时, 第2个Customer Arrived 进入第 2 个队列
- ③  $t = 7$ 分钟时, 第3个Customer Arrived 进入第 3 个队列
- ④  $t = 12$ 分钟时, 第3个客户Departure 第3个队列, 用时 5
- ⑤  $t = 12$ 分钟时, 第4个Customer Arrived 进入第 3 个队列
- ⑥  $t = 13$ 分钟时, 第2个客户Departure 第2个队列, 用时 11
- ⑦  $t = 16$ 分钟时, 第5个Customer Arrived 进入第 2 个队列
- ⑧  $t = 18$ 分钟时, 第1个客户Departure 第1个队列, 用时 18
- ⑨ .....
- ⑩  $t = 93$ 分钟时, 第20个客户Departure 第3个队列, 用时 38
- ⑪  $t = 94$ 分钟时, 第21个客户Departure 第4个队列, 用时 38
- ⑫ The average time is 22.2857

# 练习题

- 1. 设循环队列用数组 $A[0..m-1]$ 表示，队首、队尾指针分别是 $front$ 和 $rear$ ，判定队满的条件？
- 2. 循环队列用数组 $A[0..m-1]$ 存放其元素值，已知其头尾指针分别是 $front$ 和 $rear$ ，则当前队列的元素个数是？
- 3. 循环队列存储在数组 $A[0..m]$ 中，则入队时的操作？
- 4. 若用一个大小为6的数组来实现循环队列，且当前 $rear$ 和 $front$ 的值分别为0和3，当从队列中删除一个元素，再加入两个元素后， $rear$ 和 $front$ 的值分别为多少？
- 5. 用链表方式存储的队列，在进行删除运算时头、尾指针可能都要修改？

6. 阅读代码，说出以下代码的功能。（栈和队列的元素类型均为int）

```
void sf2(Queue &Q){
    stack S;
    int d;
    initstack(S);
    while(! QueueEmpty(Q)){
        DeQueue(Q,d);
        Push(S,d);
    }
    while(!StackEmpty(S))    {
        Pop(S,d);
        EnQueue(Q,d);
    }
}
```

# 小结

- 1. 掌握栈和队列这两种抽象数据类型的特点，并能在相应的应用问题中正确选用它们。
- 2. 熟练掌握顺序栈和链栈的类型定义及基本操作实现。
- 3. 熟练掌握循环队列和链队列的类型定义及基本操作实现。
- 4. \*\* 理解递归算法执行过程中栈的状态变化过程。

## ■ 栈

- 操作受限的线性表，只在栈顶进行插入(push)和删除(pop)
- 先进后出的特点
- 链栈是不带头结点的单链表，插入和删除都在头指针处进行
- 顺序栈中top指针的用法
- \* 栈的应用: 函数调用、括号匹配、数制转换、表达式求值等
- 能够判断正确的出栈序列
- \* 能够将中缀表达式转换成后缀表达式，会计算后缀表达式

## ■ 队列

- 操作受限的线性表，只在队尾插入(enqueue)，队头删除(dequeue)。
- 先进先出的特点
- 链队列是包含头指针和尾指针的单链表
- 队列的顺序存储结构是循环队列
  - 头尾指针的实指和虚指，指针加一要取模，队列长度的计算，队满的判定，队空的判定。
- 队列的应用：解决共享资源的分配，离散事件的模拟
- 双栈当队

# 扩展题

1. 双端队列deque是对队列的扩展，该结构允许在队列的两端分别进行插入和删除操作。接口如下（见下页）。请实现如下所定义的双向队列结构（可考虑从已有结构中派生）；分析这些接口的时间复杂度各为多少？

```
template<typename T> class Deque{
public:
    T& front();//读取首元素
    T& rear();//读取末元素
    void insertFront(T const& e); /*e插入至队列前
端*/
    void insertRear(T const& e); /*e插入至队列末
端*/
    T removeFront();//删除队列首元素
    T removeRear();//删除队列末元素
};
```

## 附：栈的c++定义举例

```
template<class ElemType>
```

```
class seqStack{ //顺序栈类的定义
```

```
private:
```

```
    ElemType *elem;
```

```
    int top;    int maxSize;
```

```
public:
```

```
    seqStack(int initSize=100){
```

```
        elem=new ElemType[initSize];
```

```
        maxSize=initSize; top=-1; }
```

```
    ~seqStack(){ delete [] elem; }
```

```
    void push(const ElemType &x);
```

```
    ElemType pop();
```

```
    ElemType top() const;
```

```
    bool isEmpty() const{ return top==-1; }
```

```
};
```

## 附：C++STL（标准模板库）中的栈和队列

**容器适配器**：借助于其他容器存储数据的容器。栈和队列都是容器适配器。

定义一个栈(或队列)对象，要指明栈中元素的类型以及借助于哪个容器，因此栈的类模板有两个模板参数，分别表示栈元素类型和所借助的容器类型。栈可以借助的容器有：vector, list, deque。栈的类模板的第2个参数缺省值是deque,说明底层容器是deque。

容器适配器无**迭代器**。

```
#include <stack>
```

```
stack<int, vector<int> > s1;//用vector保存数据的整数栈
```

```
stack<int, list<int> > s2;//用list保存数据的整数栈
```

```
stack<int> s3;//用deque保存数据的整数栈
```

# STL中栈和队列的标准函数

■ #include <queue>

函数	功能
push	入队,
pop	出队
front	取队头元素
back	取队尾元素
empty	判队空
size	队列长度

■ #include <stack>

函数	功能
push	入栈,
pop	出栈
top	取栈顶元素
empty	判栈空