



第三章 栈与队列

第一部分 栈

BUPT

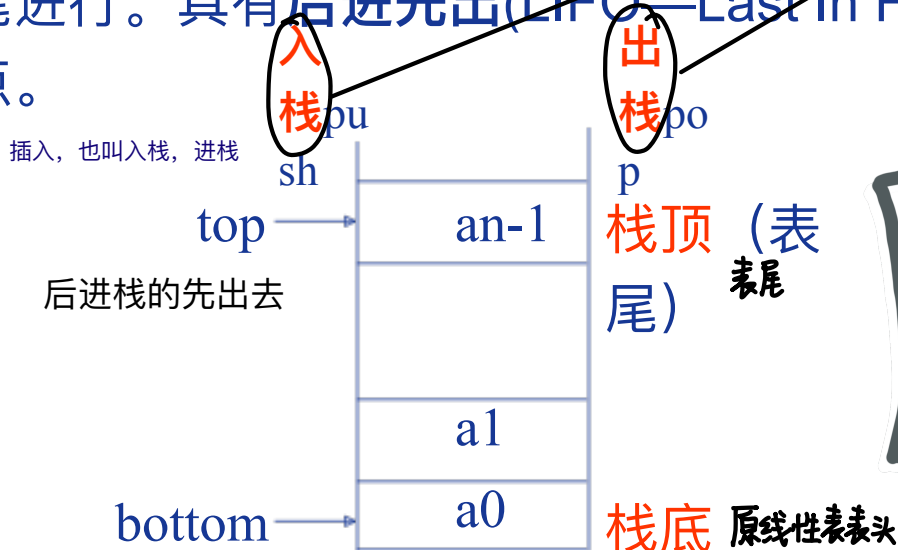
- 1 栈的定义
- 2 顺序栈
- 3 链栈
- 4 栈的应用

3.1 栈的定义

BUPT

[栈的定义]

栈是一种特殊的线性表，限定插入和删除操作只能在表尾进行。具有后进先出(LIFO—Last In First Out)的特点。



定义在栈结构上的基本运算

- (1) 生成空栈操作
- (2) 判栈空函数
- (3) 数据元素入栈操作
- (4) 数据元素出栈函数
- (5) 取栈顶元素函数
- (6) 置栈空操作
- (7) 求当前栈元素个数函数

限定位置

重点讨论特性

3.2. 顺序栈

BUPT

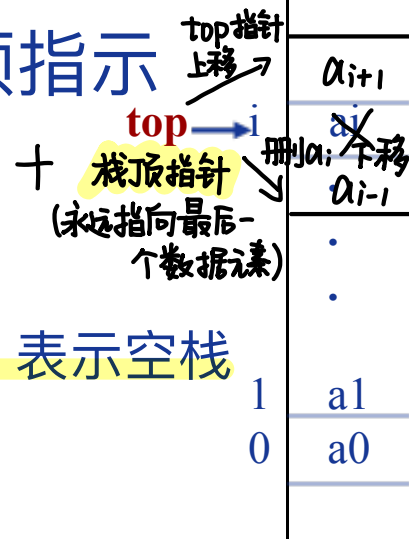
[栈的顺序存储结构]

还是线性表.

一个栈独占一组地址连续的存储单元

[类型定义]

数组(栈空间)+栈顶指示



- 通常0下标端设为栈底， 栈顶指针 top 值为-1，表示空栈

和顺序表相似，顺序栈的类型描述如下：

```
#define MAXSIZE 1024
typedef struct
{datatype data[MAXSIZE]; 数组
  int top; 栈顶指针
}SeqStack;
```



$S \rightarrow top == 0$; 只有1个
出栈
下溢

定义一个指向顺序栈的指针： `SeqStack *s;`

$S \rightarrow top == 0$ (此时出栈下溢) 只有1个数据元素

$S \rightarrow top == MAXSIZE - 1$ 此时入栈上溢

没有空间仍要入栈,则入栈上溢

【算法3-1】置空栈算法

```
SeqStack *Init_SeqStack()
```

```
{ SeqStack *s;
```

```
  s=(SeqStack *) malloc(sizeof(SeqStack)); /*申请栈空间*/
```

```
  if (!s)
```

```
  { printf("空间不足\n");
```

```
    return NULL; /*未申请到足够大的存储空间，返回空指针*/
```

```
  }
```

```
  else
```

```
  { s->top=-1; /*初始化栈顶指针*/
```

```
    return s; /*申请到栈空间，返回栈空间地址*/
```

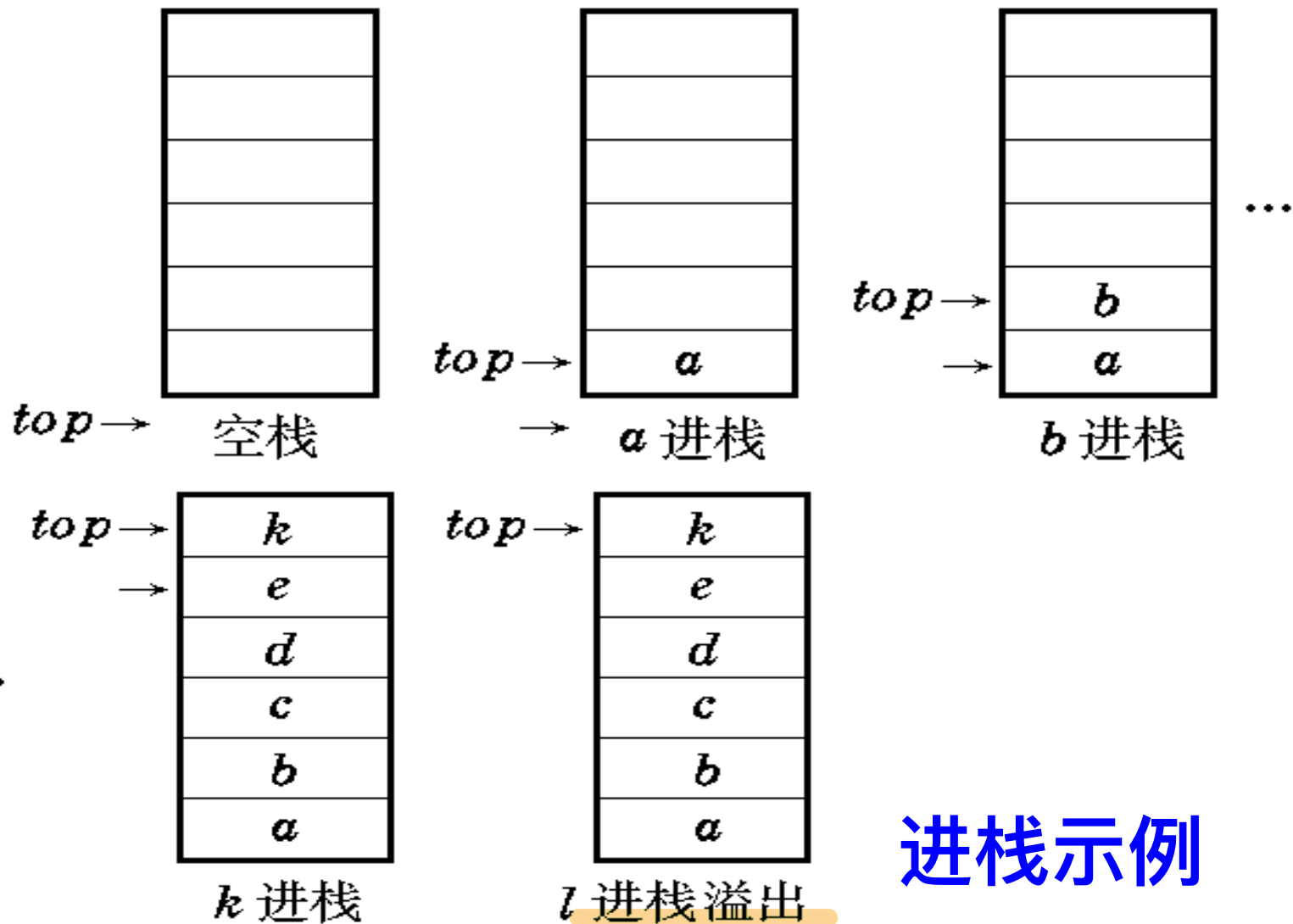
```
  }
```

```
}
```

出错
处理

【算法3-2】 判栈空算法


```
int Empty_SeqStack(SeqStack *s)
{ if (s->top==-1)
    return 1;  /*空栈返回1*/,
  else
    return 0;
}
```

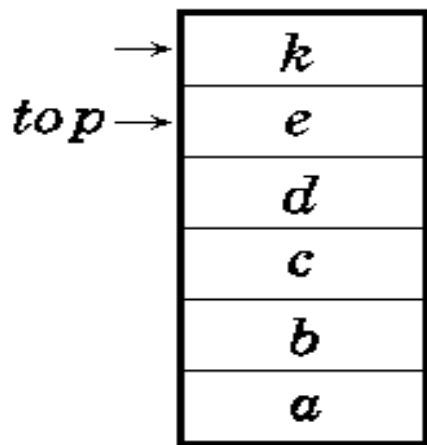



进栈示例

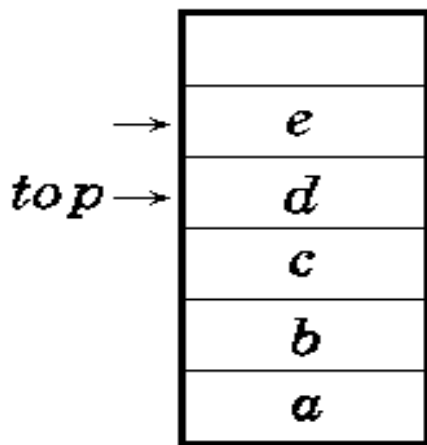
【算法3-3】 入栈算法

```
int Push_SeqStack(SeqStack *s,datatype x)
{ if (s->top==MAXSIZE-1)
    return 0;      /*栈满不能入栈，返回错误代码0*/
else
    { s->top++;      /*栈顶指针向上移动*/
      s->data[s->top]=x; /*将x至入新的栈顶*/
      return 1;      /*入栈成功，返回成功代码1 */
    }
}
```

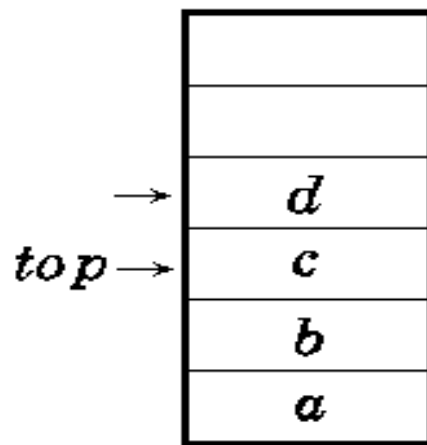
A hand-drawn blue star with an arrow pointing to the line 's->data[s->top]=x; /*将x至入新的栈顶*/'.



k 退栈



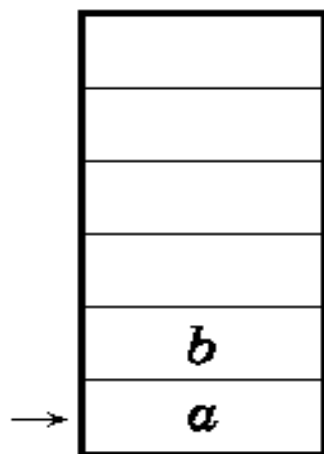
e 退栈



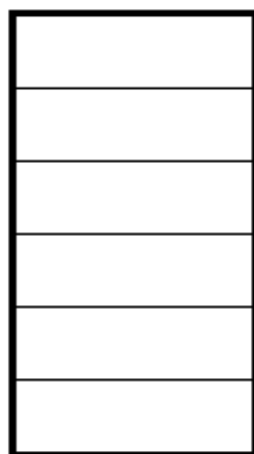
d 退栈

...

...



a 退栈



空栈

退栈示例

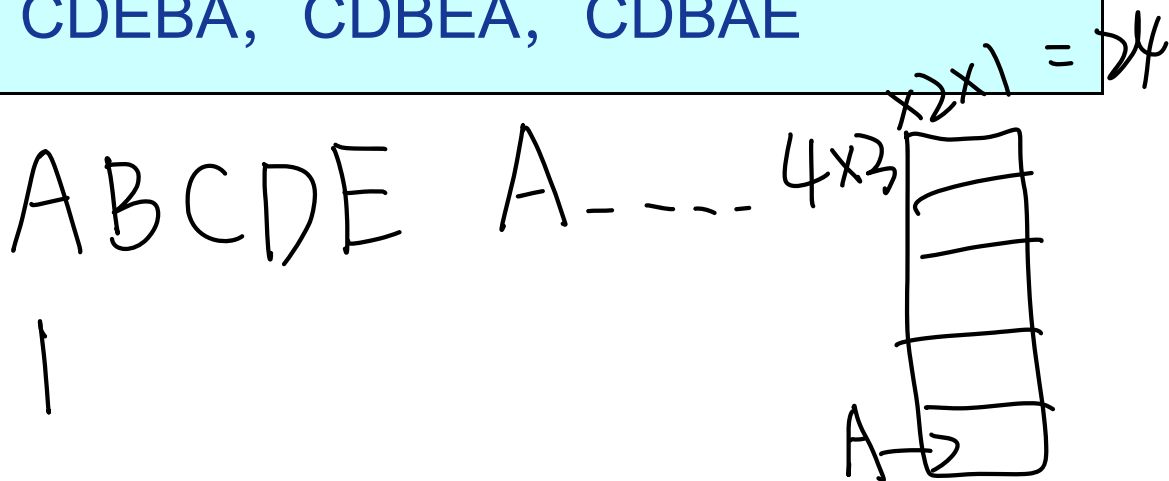
【算法3-4】 出栈算法

```
int Pop_SeqStack(SeqStack *s,datatype *x)
{ /*通过*x返回原栈顶元素*/
    if (Empty_SeqStack(s))
        return 0;    /*栈空不能出栈，返回错误代码0*/
    else
    { *x=s->data[s->top]; /*保存栈顶元素值*/
      s->top--;    /*栈顶指针向下移动*/
      return 1;    /*返回成功代码1 */
    }
}
```

有5个元素，其入栈次序

为：A, B, C, D, E，在各种可能的出栈次序中，以元素C, D最先出栈（即C第一个且D第二个出栈）的次序有哪几个？

CDEBA, CDBEA, CDBAE



思考：n个元素依次入栈，可得到多少个合法的出栈序列
 $(2n)!/[(n+1)!*n!]$

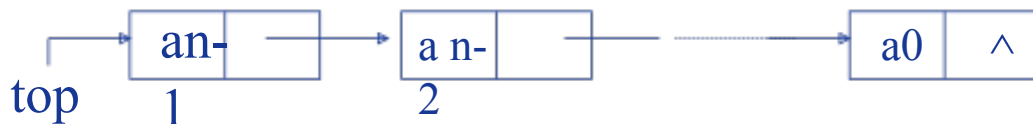
不同的出栈序列实际上对应着不同的入栈出栈操作，以1记为入栈，0为出栈。则问题实际上是求n个1和n个0构成的全排列，其中任意一个位置，它及它此前的数中，1的个数要大于等于0的个数。
 n个1和n个0构成的全排列数为： $(2n)!/[n!*n!]$

在n个0和n个1构成的2n个数的序列中，假设第一次出现0的个数大于1的个数（即0的个数比1的个数大一）的位置为k，则k为奇数，k之前有相等数目的0和1，各为 $(k-1)/2$ 。若把这k个数，0换成1，1换成0，则原序列唯一对应上一个n+1个1和n-1个0的序列。反之，任意一个由n+1个1和n-1个0构成的序列也唯一的对应一个这样不合要求的序列。由于一一对应，故这样不合要求的序列数实际上等于有n+1个1和n-1个0构成的排列数，即 $(2n)!/[(n+1)!(n-1)!]$ 。

因此合法的个数为： $(2n)!/[n!*n!] - (2n)!/[(n+1)!(n-1)!]$
 $= (2n)!/[(n+1)!*n!]$

3.3. 链栈

BUPT



- 链式栈无栈满问题，空间可扩充
- 插入与删除仅在栈顶处执行
- 链式栈的栈顶在链头
- 适合于多栈操作

```
typedef struct node  
{ datatype data;  
  struct node *next;  
} StackNode, * LinkStack;
```

定义top为栈顶指针: LinkStack top ;

栈中的主要运算是在栈顶插入、删除，显然在链表的头部做栈顶是最方便的。

链栈基本操作的实现如下：

(1) 置空栈

仅是需要将栈顶指针置为空即可。

(2) 判栈空

【算法3-6】 判栈空算法

```
int Empty_LinkStack(LinkStack top)
{ if (top==NULL) return 1;
  else return 0;
}
```

(3) 入栈

【算法3-7】 入栈算法

LinkStack Push_LinkStack(LinkStack top, datatype x)

```
{ StackNode *p;  
  p=(StackNode *)malloc(sizeof(StackNode));  
  p->data=x;  
  p->next=top;  
  top=p;  
  return top;  
}
```

(4) 出栈

【算法3-8】 出栈算法

LinkStack Pop_LinkStack (LinkStack top, datatype *x)

```
{ StackNode *p;  
  if (top==NULL) return NULL;  
  else { *x= top->data;  
        p=top;  
        top=top->next;  
        free (p);  
        return top;  
  }  
}
```

3.4.栈的应用

BUPT

- 栈与递归过程

[递归的含义]

函数、过程或者数据结构的内部又直接或者间接地由自身定义。

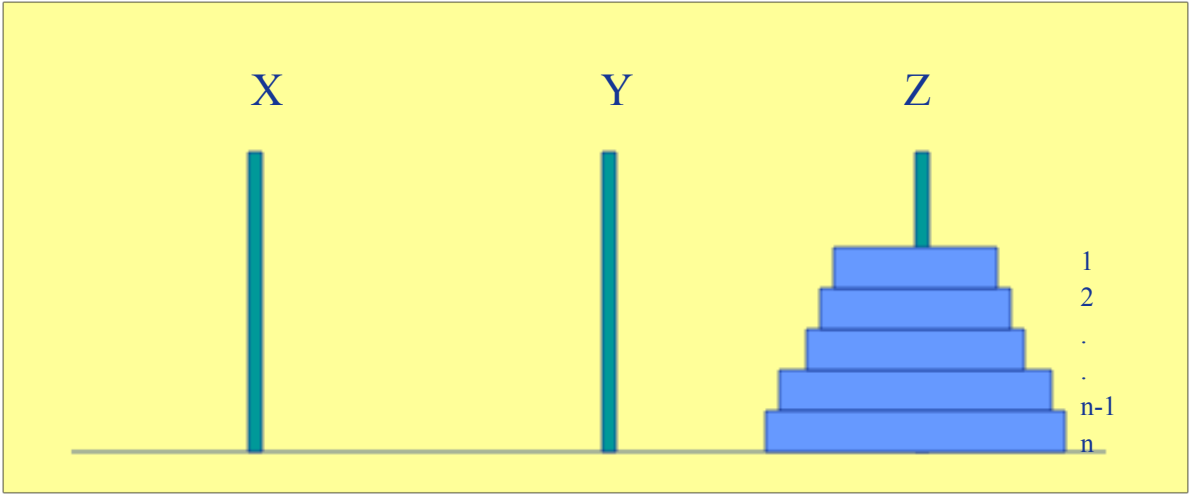
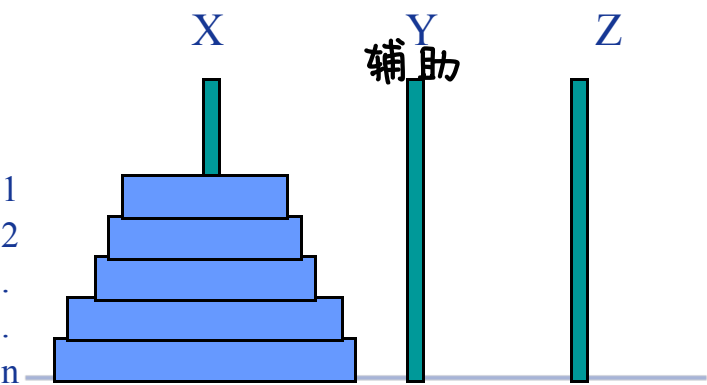
[适合于应用递归的场合]

- 规模较大的问题可以化解为规模较小的问题，且二者处理(或定义)的方式一致；
- 当问题规模降低到一定程度时是可以直接求解的(终止条件)

例1. 阶乘 $n! = \begin{matrix} 1 & n=0 \\ n \cdot (n-1)! & n>0 \\ \vdots \\ 1 \end{matrix}$

例2. n阶Hanoi塔问题

每次只能移动一叠.
且大盘必在小盘下



[写递归算法应注意的问题]

- 递归算法本身不可以作为独立的程序运行，需在其它的程序中设置调用初值，进行调用；
- 递归算法应有出口(终止条件)

例1. 求 $n!$ 有句话判定问题规模是否在可以直接解决

```
int Factorial(int n)
{
    if (n==0)
        return(1);
    return(n*Factorial(n-1));
}
```

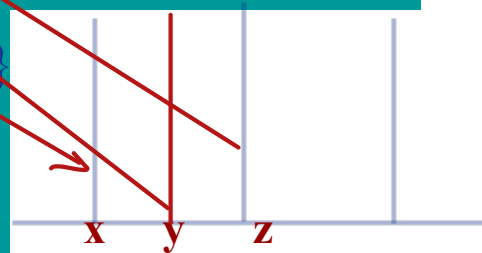
PROCEDURE hanoi()

```

void Hanoi (int n, int x, int y, int z)
{
    if (n==1)
        Move (x, 1, z) {出口条件}
    else{
        Hanoi (n-1, x, z, y);
        Move (x, n, z);
        Hanoi (n-1, y, x, z);}
}

```

盘



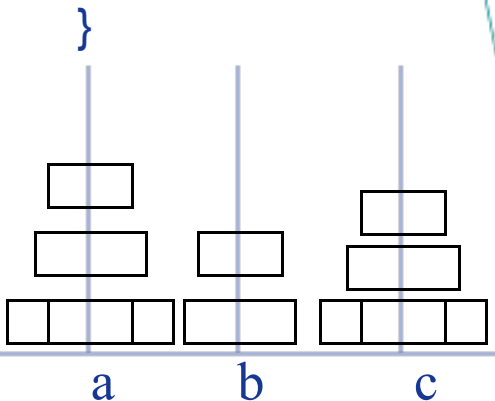
源 辅助 目标



```

void hanoi(3,a,b,c){
  if (3==1)
    move(a,1,c);
  else
  {
    hanoi(2,a,c,b);
    move(a,3,c);
    hanoi(2,b,a,c);
  }
}

```



```

void hanoi(2,a,c,b){
  if (2==1)
    move(a,1,b);
  else
  {
    hanoi(1,a,b,c);
    move(a,2,b);
    hanoi(1,c,a,b);
  }
}

```

```

void hanoi(1,a,b,c){
  if 1=1
    move(a,1,c);
  else {.....}
}

```

```

void hanoi(1,c,a,b){
  if (1==1)
    move(c,1,b);
  else {.....}
}

```

```

void hanoi(2,b,a,c) {
  if (2==1)
    move(b,1,c);
  else
  {
    hanoi(1,b,c,a);
    move(b,2,c);
    hanoi(1,a,b,c);
  }
}

```

```

void hanoi(1,b,c,a) {
  if (1==1)
    move(b,1,a);
  else {.....}
}

```

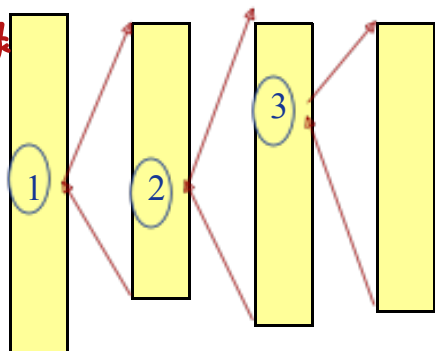
```

void hanoi(1,a,b,c) {
  if (1==1)
    move(a,1,c);
  else {.....}
}

```


[递归算法的实现原理]

用栈
保存工作记录



不用递归
工作栈 \Rightarrow 辅助变量
OS分配
考察空间复杂度应考虑进来。
 $n-1$ 栈的空间
 $O(n-1)$ \leftarrow
时间复杂度。

- 利用栈，栈中每个元素称为工作记录，分成三个部分：
返回地址 实在参数表(变参和值参) 局部变量
- 发生调用时，保护现场，即当前的工作记录入栈，然后转入被调用的过程
- 一个调用结束时，恢复现场，即若栈不空，则退栈，从退出的返回地址处继续执行下去

[递归时系统工作原理示例]

BUPT

```
int Factorial(int n){
    L1:   if (n==0)
    L2:       return(1);
    L3:   return(n*Factorial(n-1));
}
```

3 x 2

```
void main()
```

```
{
    .....
    L0: N=Factorial(3);
    .....
}
```

计算空间复杂度算上栈

的

L3.	1	$\frac{1}{1}$	
L3	2	$\frac{1}{2}$	
L3	3	$\frac{1}{3}$	
L0	3	$\frac{1}{3}$	$\frac{1}{3}$

返回地址 n Factorial N

BUPT SCST

[递归算法的用途]

- 求解递归定义的数学函数
- 在以递归方式定义的数据结构上的运算/操作
- 可用递归方式描述的解决过程

[递归转换为非递归的方法] 从大规模到小规模

- 1) 采用迭代算法 递归—从顶到底 迭代—从底到顶
- 例：求n!
- ```
int fact(int n){
 m=1;
 for (i=1;i<=n;i++) m=m*i;
 return(m); }
```
- 递归过程发生在程序最尾部
- 尾递归：重赋参数 (覆盖) 后方无语句需执行。

## 2) 消除尾递归

BUPT

例：顺序输出单链表中的结点数据

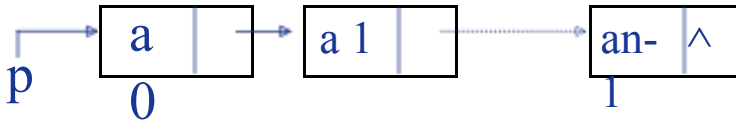
```
void linklist_output(pointer p){
 IF (p!=NULL)
 { write(p->data);
 linklist_output(p->next);
 }
}
```

使用跳转语句：

```
void linklist_output1(pointer p){
```

使用循环语句：

```
void linklist_output2(pointer p){
 while (p!=nil)
 { write(p->data);
 p:=p->next;
 }
}
```



### 3) 利用栈模拟递归—通用方法

BUPT

如果是递归函数，需改写为递归过程

例：求 $n!$

```
void fact(int n; int &f){
```

↑  
加  
个栈初始化语句

```
 init_stack(s);
```

```
 0: if (n==0) f=1;
```

```
 else
```

```
 { 1: fact(n-1,f);
```

```
 2: f:=n*f;
```

```
 }
```

```
}
```

用栈的方式改写

```
top=top+1; s[top].rd=2;
```

```
s[top].n=n; s[top].f=f;
```

```
n=n-1; goto 0;
```

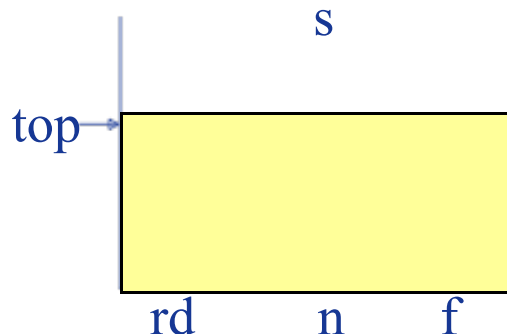
```
if (top!=0)
```

```
{ s[top].f=f; {还原本层变参值}
```

```
top=top-1; n=s[top+1].n;
```

```
f=s[top+1].f; goto s[top+1].rd }
```

## [自设栈模拟系统工作栈]



### 改写算法

在程序开头增加栈的初始化语句

改写递归调用语句 入栈处理；

确定调用的参数值；

转至调用的开始语句

改写所有递归出口 退栈还原参数；

转至返回地址处

```

void fact(int n, int & f)
{
 init_stack(s);
0: if (n==0) f=1;
 {
 1: top=top+1;
 s[top].rd=2;
 s[top].n=n;
 s[top].f=f;
 n=n-1; goto 0;
 2: f=n*f }
 if (top!=0)
 {
 s[top].f=f;
 top=top-1;
 n=s[top+1].n;
 f=s[top+1].f;
 goto s[top+1].rd; }
}

```



```

void fact(int n,int &f)
{
 init_stack(s);
 while (n!=0)
 {
 top=top+1;
 s[top].n=n;
 n=n-1;}

 f=1;
 while (top!=0)
 {
 top=top-1;
 n=s[top+1].n;
 f=n*f; }
}

```

注：此时可确定栈中只  
存放n值即可。

# 栈的应用2-整型数简单表达式求值

[前提假设] 表达式语法正确；  
表达式结束符为‘#’

## [算法思想]

OPTR栈—寄存运算符 OPND栈—寄存操作数

- 1)放入起始符‘#’作为OPTR栈底元素
- 2)依次读入表达式字符，

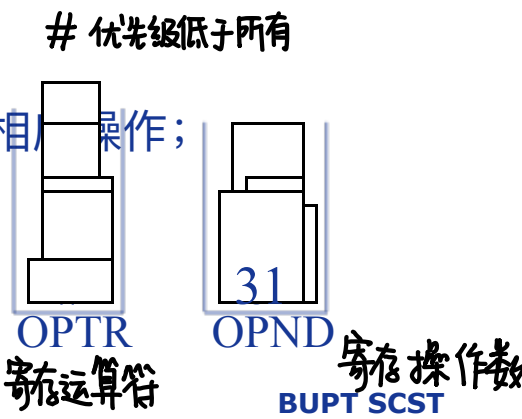
操作数：进OPND栈；

运算符：与OPTR.top元素比较优先级后做相应操作；

直至读入‘#’且OPTR栈只剩‘#’ 两#：操作结束

3)OPND栈底所留元素即为所求

[例] 5+3\*(7-2)+11# 简单表达式运算  
↑结束表达式





# 第二部分 队列

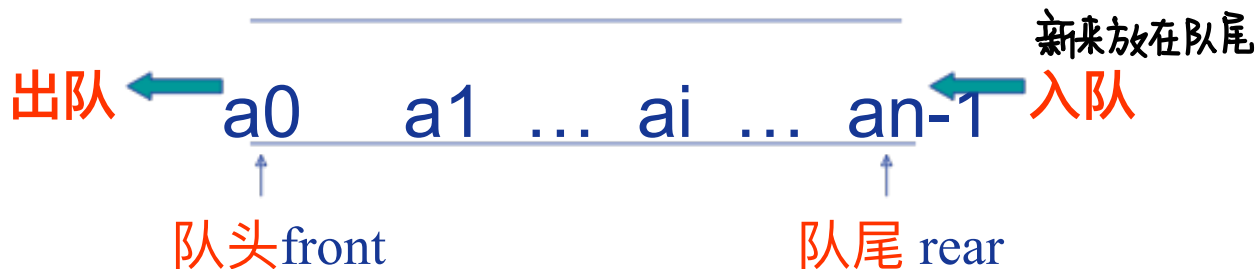
BUPT

- 1 队列的定义
- 2 链队列
- 3 循环队列

## 3.5 队列的定义

BUPT

**队列**是一种特殊的线性表，限定插入和删除操作分别在表的两端进行。具有**先进先出**(FIFO—First In First Out)的特点。



# 定义在队列结构上的基本运算

(1)构造空队列操作

(2)判队空否函数

(3)元素入队操作

(4)元素出队函数

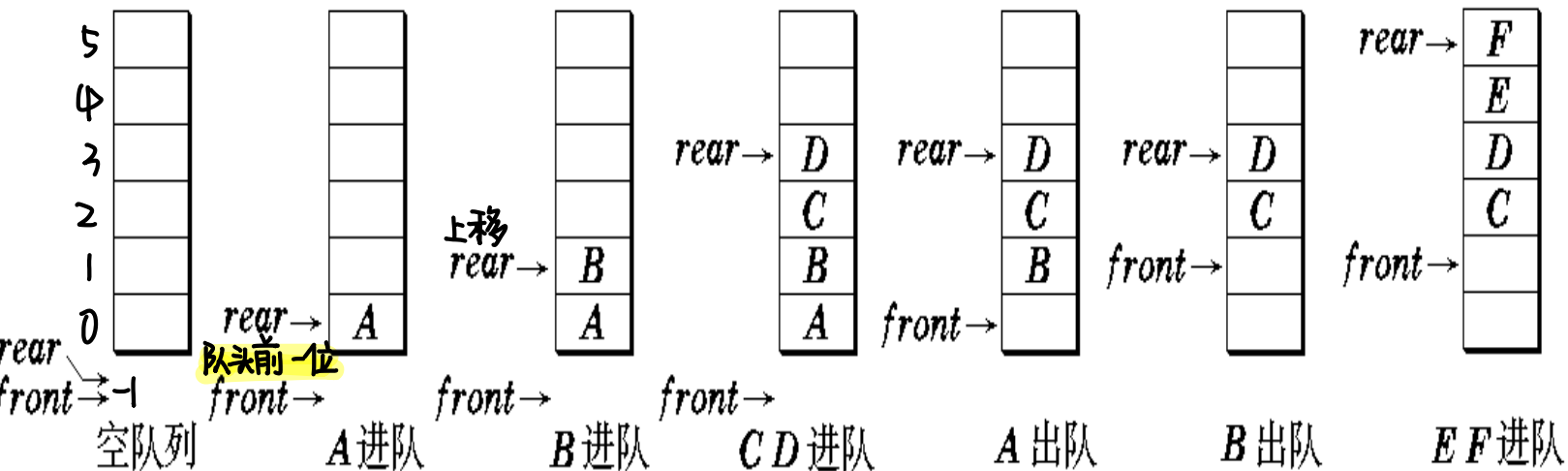
限定位置作插入,删除

(5)取队头元素函数

(6) 队列置空操作

(7)求队中元素个数函数

# 队列的进队和出队



- **进队时队尾指针先进一  $rear = rear + 1$ ，再将新元素按  $rear$  指示位置加入。**
- **出队时队头指针先进一  $front = front + 1$ ，再将  $front$  指示的元素取出。**
- **队满时再进队将溢出出错；队空时再出队将队空处理。**

**思考：可否用两个栈实现一个队列？如何实现？**

## 利用两个栈s1、s2（等容量）模拟一个队列,实现队列的插入,删除以及判队空运算

- 栈的特点是后进先出, 队列的特点是先进先出。初始时设栈s1和栈s2均为空。
- (1) 用栈s1和s2模拟队列的输入: 设s1和s2容量相等。分以下三种情况讨论: 若s1未满, 则元素入s1栈; 若s1满, s2空, 则将s1全部元素退栈, 再压栈入s2, 之后元素入s1栈; 若s1满, s2不空 (已有出队列元素), 则不能入队。
- (2) 用栈s1和s2模拟队列出队 (删除): 若栈s2不空, 退栈, 即是队列的出队; 若s2为空且s1不空, 则将s1栈中全部元素退栈, 并依次压入s2中, s2栈顶元素退栈, 这就是相当于队列的出队。若栈s1为空并且s2也为空, 队列空, 不能出队。
- (3) 判队空 若栈s1为空并且s2也为空, 才是队列空。

讨论: s1和s2容量之和是队列的最大容量。其操作是, s1栈满后, 全部退栈并压栈入s2 (设s1和s2容量相等)。再入栈s1直至s1满。这相当于队列元素“入队”完毕。出队时, s2退栈完毕后, s1栈中元素依次退栈到s2, s2退栈完毕, 相当于队列中全部元素出队。

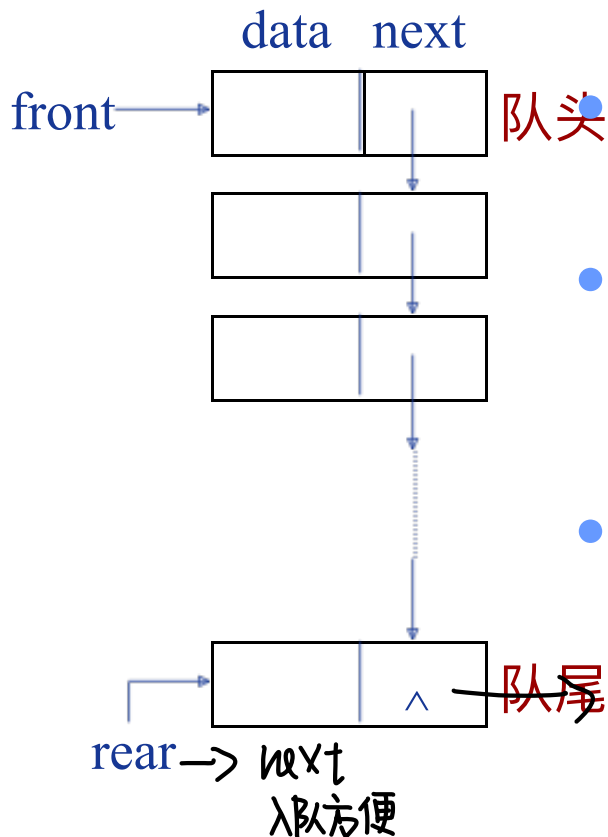
在栈s2不空情况下, 若要求入队操作, 只要s1不满, 就可压入s1中。若s1满和s2不空状态下要求队列的入队时, 按出错处理。

## 3.6 链队列

BUPT

### 用单链表表示的队列

类型定义：队头指针 + 队尾指针



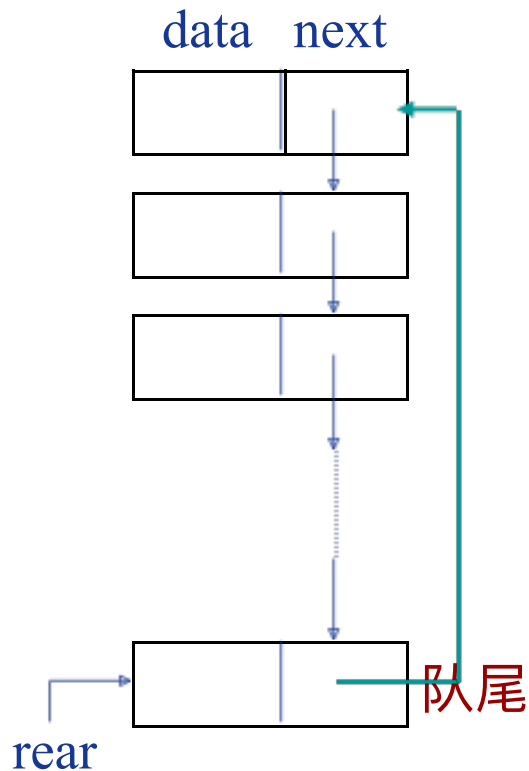
队头在链头，队尾在链尾。

- 链式队列在进队时无队满问题，但有队空间问题。
- 队空条件为

$front \rightarrow next == NULL$

# 用单循环链表定义的队列

类型定义：队尾指针



## 3.7 循环队列

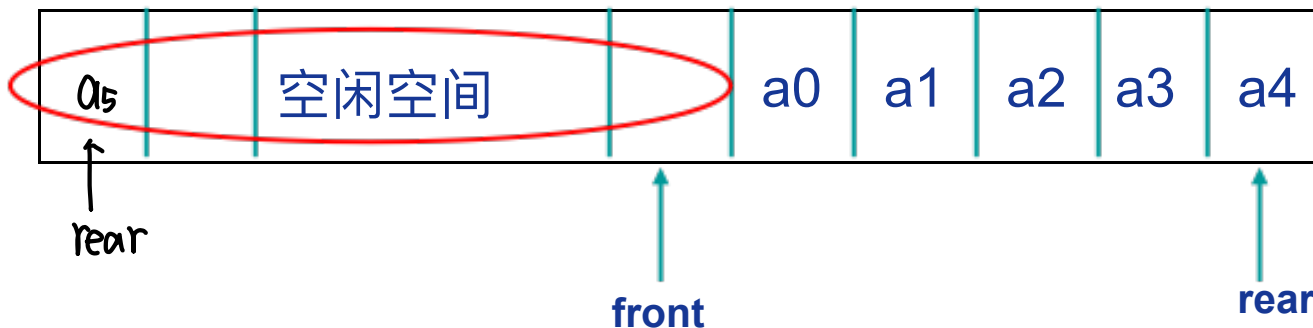
BUPT

### 一般用法的顺序存储结构之缺陷

出队列操作后大量移动数据或存在“假上溢”现象：即存在空闲空间但无法入队。

### 解决方法

- 平移元素法，当发生假溢出时，就把整个队列的元素平移到存储区的首部，然后再插入新元素。这种方法需移动大量的元素，因而效率是很低的。

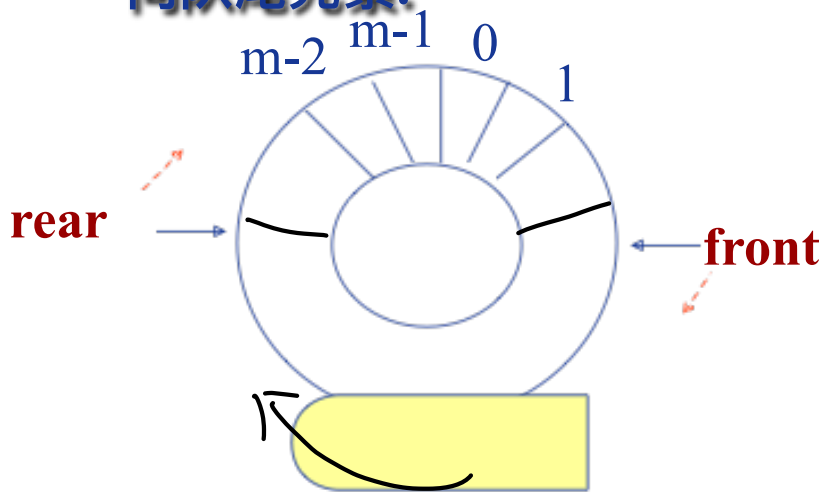




**[循环队列]**将顺序队列的存储区假想为一个环状空间

**[类型定义]** 一维数组(队列空间) + 头指针 + 尾指针

队头指针`front`指向队头元素的前一位置，而队尾指针`rear`指向队尾元素。



**优点：**不需要移动元素，操作效率高，空间的利用率也很高。

```

typedef struct {
 datatype data[MAXSIZE]; /*数据的存
 储区*/
 int front,rear; /*队头队尾指针*/
 int num; /*队中元素的个数*/
}c_SeQueue; /*循环队*/

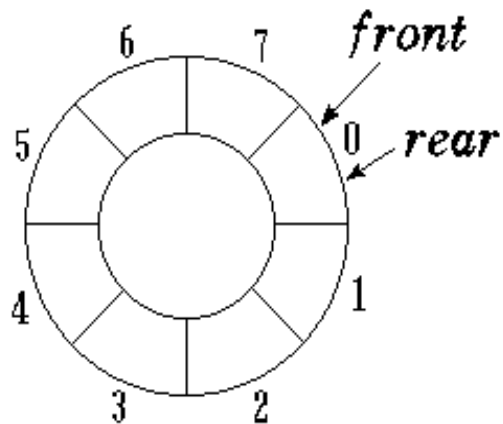
c_SeQueue *q;

```

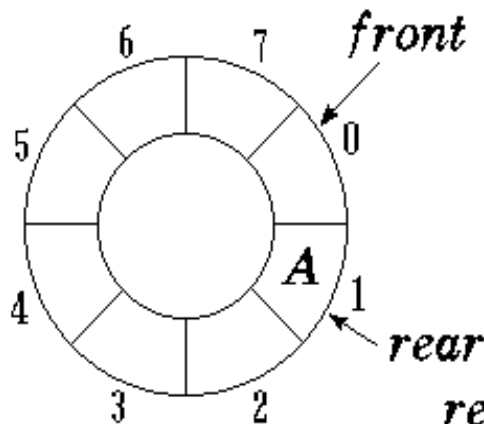
可以不用操作 用头.尾指针求即可

# 循环队列的进队和出队实例

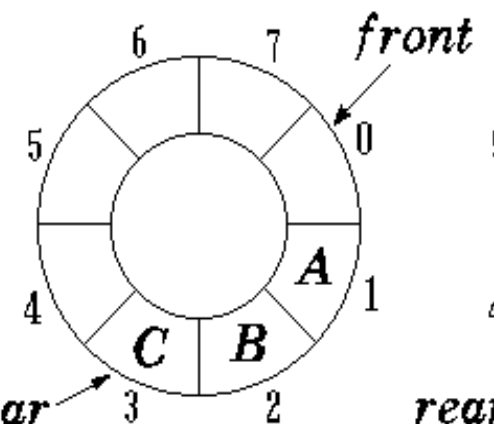
BUPT



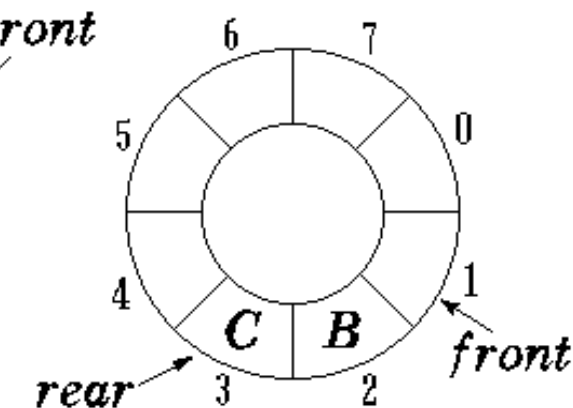
空队列



A 进队

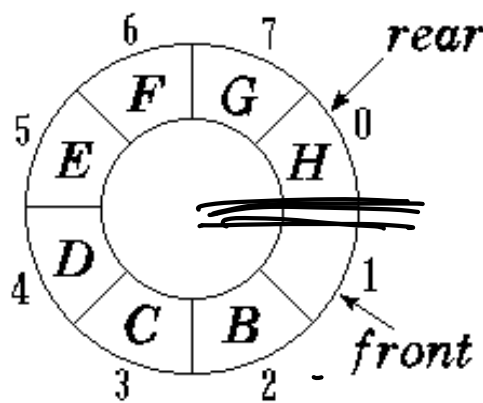


B C 进队



A 出队

G 出队同理



DEFGH 进队

DEFG

队尾指针+1  
H: . . . .

1. 若  $(rear \neq m-1)$   
 $rear \rightarrow next ++;$   
 或  
 $rear = 0;$

2. 从 0 到  $m-1$

作取余

$rear = (rear + 1) \% m$

✓ 存储队列的数组被当作首尾相接的表处理。  
 为解决如何判断队列的满与空间问题：**牺牲一个存储单元**。解决队满、队空条件冲突  $rear+1$  对空间作取余恰好  $= front$

✓ 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。

出队:  $q \rightarrow front = (q \rightarrow front + 1) \% maxSize;$

入队:  $q \rightarrow rear = (q \rightarrow rear + 1) \% maxSize;$

队列初始化:  $q \rightarrow front = q \rightarrow rear = 0;$

队空条件:  $q \rightarrow front == q \rightarrow rear;$   $\text{mod } (rear - front + 8)$

队满条件:  $(q \rightarrow rear + 1) \% maxSize == q \rightarrow front$

思考: 循环队列中当前元素的个数为多少?

$rear - front$  如果是负数,  $+ 8 = \text{元素个数}$

当前元素个数  $(q \rightarrow rear - q \rightarrow front + maxSize) \% maxSize$

线性表.  
顺序表.  
链表  
栈

队列  
循环队列  
链栈

BUPT

## (1) 置空队

【算法3-14】 构建一个空的循环队算法

```
CSeQueue* Init_SeQueue()
```

```
{ CSeQueue *q;
```

```
 q=(CSeQueue *)malloc(sizeof(CSeQueue));
```

```
 q->front=q->rear=MAXSIZE-1;
```

```
 q->num=0;
```

```
 return q;
```

```
}
```

## (2) 入队

【算法3-15】 循环队入队算法

```
int In_SeQueue (CSeQueue *q , datatype x)
{ if(q->num==MAXSIZE)
 { printf("队满");
 return -1; /*队满不能入队*/
 }
 else
 { q->rear=(q->rear+1) % MAXSIZE;
 q->data[q->rear]=x;
 q->num++;
 return 1; /*入队完成*/
 }
}
```

### (3) 出队

【算法3-16】 循环队出队算法

```
int Out_SeQueue (CSeQueue *q, datatype *x)
{
 if (q->num==0)
 {
 printf("队空");
 return -1; /*队空不能出队*/
 }
 else {q->front=(q->front+1) % MAXSIZE;
 *x=q->data[q->front]; /*读出队头元素*/
 q->num--;
 return 1; /*出队完成*/
 }
}
```

## (4) 判队空

【算法3-17】 判循环队空队算法

```
int Empty_SeQueue(CSeQueue *q)
{ if (q->num==0) return 1;
 else return 0;
}
```



**[例]**假设以数组sq[8]存放循环队列元素,变量front指向队头元素的前一位置,变量rear指向队尾元素, 如用A和D分别表示入队和出队操作。

请给出下列执行操作序列A3D1A5D2A1D2A1时的状态

队空的初始条件:  $\text{front}=\text{rear}=0$ ;

执行操作A3后,  $\text{rear}=3$ ; // A3表示三次入队操作

执行操作D1后,  $\text{front}=1$ ; //D1表示一次出队操作

执行操作A5后,  $\text{rear}=0$ ;

执行操作D2后,  $\text{front}=3$ ;

执行操作A1后,  $\text{rear}=1$ ;

执行操作D2后,  $\text{front}=5$ ;

执行操作A4后, 溢出。因为执行A3后,  $\text{rear}=4$ , 这时队满, 若再执行A操

# 本章作业

BUPT

1. 试推导求解  $n$  阶hanoi塔问题至少要执行的移动操作  $\text{move}$  次数。
2. 试证明：若借助栈由输入序列  $1, 2, \dots, n$  得到输出序列为  $P_1, P_2, \dots, P_n$ （它是输入序列的一个排列），则在输出序列中不可能出现这样的情形：存在着  $i < j < k$ , 使  $P_j < P_k < P_i$ 。

3. 假设以I和O分别表示入栈和出栈操作。栈的初态和终态均为空，入栈和出栈的操作序列可表示为仅由I和O组成的序列，称可以操作的序列为合法序列，否则称为非法序列。

(1) 下面所示的序列中哪些是合法的？

A. IOIIOIOO    B. IOOIIOIO    C.  
IIIOIOIO    D. IIIOOIOO