

第三章

栈和队列

※ 教学内容:

栈的定义、表示及实现；栈的应用（表达式求值、递归）；队列的定义、表示及实现

※ 教学重点:

栈和队列的特点；在两种存储结构上栈的基本操作的实现；循环队列和链队列的基本运算；递归算法执行过程中栈状态的变化过程

※ 教学难点:

循环队列和链队列的基本运算。

栈和队列是两种常用的数据类型

栈和队列是限定插入和删除只能在表的“端点”进行的线性表。

线性表

Insert(L, **i**, x)

$1 \leq i \leq n+1$

Delete(L, **i**)

$1 \leq i \leq n$

栈

Insert(S, **n+1**, x)

Delete(S, **n**)

队列

Insert(Q, **n+1**, x)

Delete(Q, **1**)

3.1 栈的类型定义

3.2 栈类型的实现

3.3 栈的应用举例

3.4 队列的类型定义

3.5 队列类型的实现

3.1 栈的类型定义

一、栈(stack)的定义

限定仅在表尾进行插入或删除操作的线性表。

其中允许进行插入和删除的一端（表尾）称为栈顶；

另一端（表头）称为栈底。

当表中没有元素时，称为空栈。

假设栈

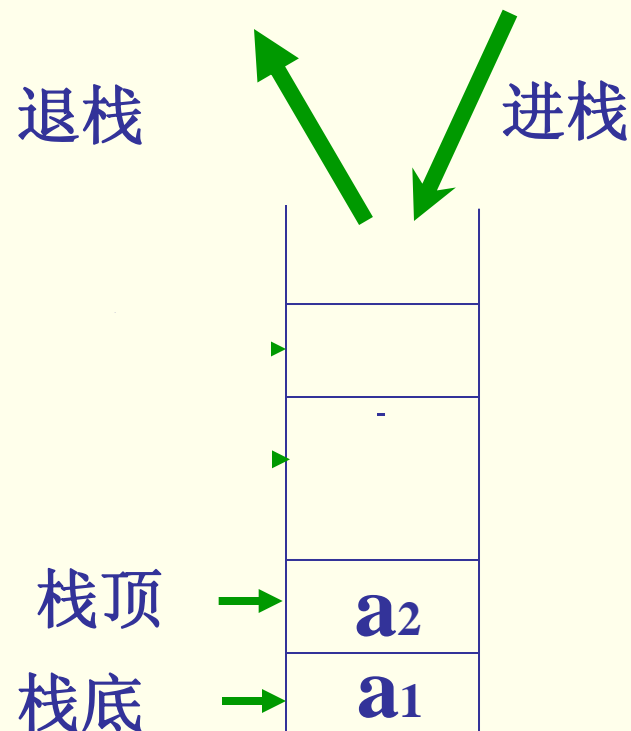
$$S=(a_1, a_2, \dots, a_n)$$

↑ ↑
栈底元素 栈顶元素

进栈次序: a_1, a_2, \dots, a_n

退栈次序: $a_n, a_{n-1}, \dots, a_2, a_1$

栈是按照“**后进先出**”原则处理数据元素的，栈也称为“后进先出”表。简称LIFO表。



二、栈的抽象数据类型的类型定义

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:

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

约定 a_n 端为栈顶, a_1 端为栈底。

基本操作:

} ADT Stack

InitStack(&S)

DestroyStack(&S)

StackLength(S)

StackEmpty(s)

GetTop(S, &e)

ClearStack(&S)

Push(&S, e)

Pop(&S, &e)

StackTravers(S, visit())

InitStack(&S) 初始化操作

操作结果: 构造一个空栈 S。

DestroyStack(&S)

初始条件: 栈 S 已存在。

操作结果: 栈 S 被销毁。

StackLength(S) 求栈的长度

初始条件: 栈 S 已存在。

操作结果: 返回 S 的元素个数, 即栈的长度。

StackEmpty(S) 判定S是否为空栈

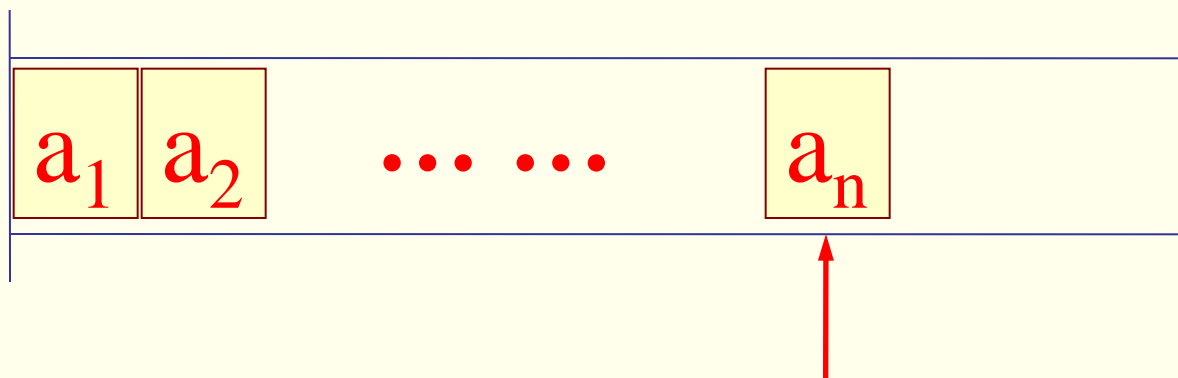
初始条件: 栈 S 已存在。

操作结果: 若栈 S 为空栈, 则返回TRUE,
否则 FALSE。

GetTop(S, &e) 取栈顶元素

初始条件：栈 S 已存在且非空。

操作结果：用 e 返回 S 的栈顶元素。



ClearStack(&S) 栈置空操作

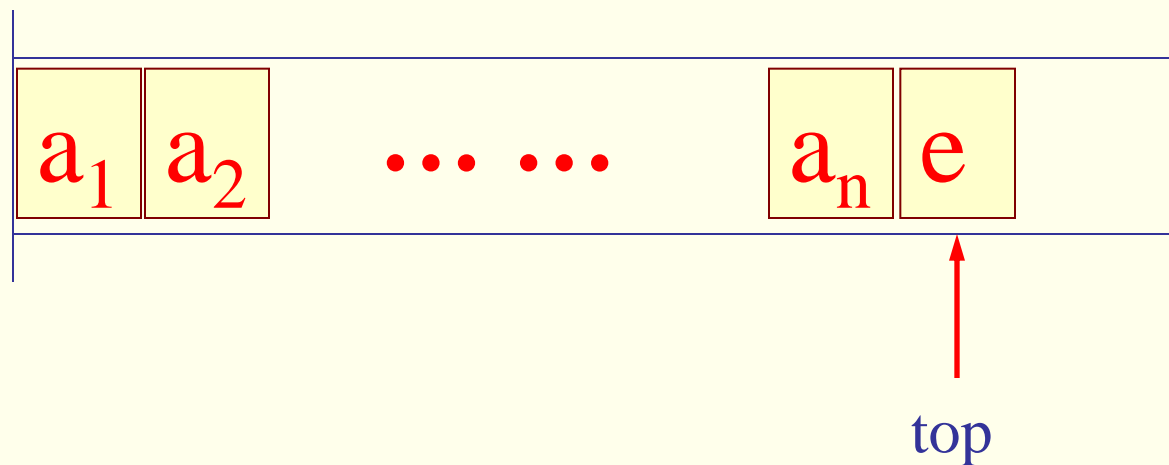
初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

Push(&S, e) 入栈操作

初始条件：栈 S 已存在。

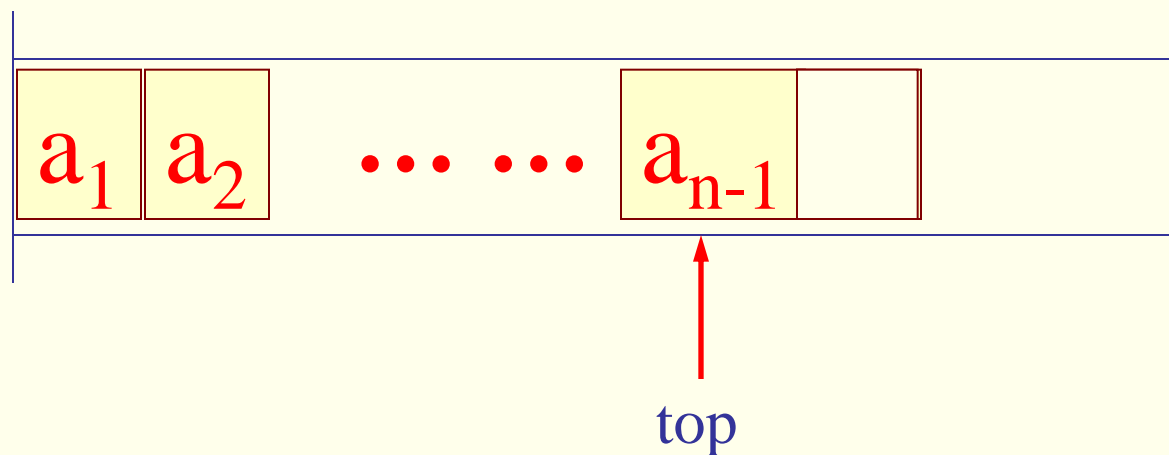
操作结果：插入元素 e 为新的栈顶元素。



Pop(&S, &e) 出栈操作

初始条件：栈 S 已存在且非空。

操作结果：删除 S 的栈顶元素，并用 e 返回其值。



例 3-1

有三个元素的进栈顺序为1、2、3。举出此三个元素可能的出栈序列，并写出相应的进栈和出栈操作序列（假设以s和x分别表示进栈和出栈操作）。

出栈序列

操作序列

1、2、3

S X S X S X

1、3、2

S X S S X X

2、1、3

S S X X S X

2、3、1

S S X S X X

3、2、1

S S S X X X

3.2 栈类型的实现

一、栈的顺序存储结构

- 利用一组地址连续的存储单元（数组）依次存放从栈底到栈顶的若干数据元素，数组的上界maxsize表示栈的最大容量；附设栈顶指针Top来指示栈顶元素在数组中的位置。
- 一个栈的栈底位置是固定的，栈顶位置随着进栈和出栈操作而变化。



栈的顺序存储结构方法一

C语言描述:

类似于线性表的顺序存储结构

```
#define MAXSIZE n
```

/*n为栈中数据元素个数的最大可能值*/

```
typedef struct
```

```
{  
    elemtype stack[MAXSIZE];
```

```
    int top;
```

```
} sqstack;
```

初始化栈算法

```
void Initstack(sqstack &s)
```

```
{
```

```
    s.top= -1;    //设置栈顶指针top, 表示栈空
```

```
}
```

进栈算法

```
status push(sqstack &s, elemtype x)
{ if (s.top>=MAXSIZE-1)
    return ERROR;    //栈满，上溢
  else { s.top++;
        s .stack[top]=x;
        return OK;
      }
} //push
```

出栈算法

```
status pop(sqstack &s, elemtype &e)
{ if (s.top<0)
    return ERROR;    //栈空，下溢
  else { e= s .stack[top];
        s.top--;
        return OK;
      }
} // pop
```

取栈顶元素算法

```
status gettop(sqstack s, elemtype &e)
{ if (s . top < 0)
    return NULL;    //栈空, 返回空值
  else {
    e = s . stack[top];
    return OK;
  }
} // gettop
```

判栈空算法

```
status empty (sqstack s)
```

```
{ if (s . top < 0)
```

```
    return TRUE;    //栈空, 返回TRUE
```

```
else {
```

```
    return FALSE;
```

```
}
```

```
} // empty
```

在实现上述这些操作时，可以不把栈S定义为结构体类型（sqstack s）

而是定义为指向结构体的指针类型：

Sqstack *s;

这样在编程时，只需将

s.top 改为 **s → top**

s.stack[] 改为 **s → stack[]** 即可。

栈的顺序存储结构方法二

类似于线性表的顺序映象实现，指向表尾的指针可以作为栈顶指针。栈顶指针指向最后一个元素的下一个位置。

```
#define STACK_INIT_SIZE 100;
#define STACKINCREMENT 10;

typedef struct {
    SElemType *base;
    SElemType *top; //指向栈顶的下一个位置
    int stacksize;
} SqStack;
```

Status InitStack (SqStack &S)

```
{ // 构造一个空栈S
```

```
S.base=(ElemType*)malloc(STACK_INIT_SIZE*  
                           sizeof(ElemType));
```

```
if (!S.base) exit (OVERFLOW); //存储分配失败
```

S.top = S.base;

```
S.stacksize = STACK_INIT_SIZE;
```

```
return OK;
```

}

```
Status Push (SqStack &S, SElemType e) {  
    if (S.top - S.base >= S.stacksize) { //栈满，追加存储空间  
        S.base = (ElemType *) realloc ( S.base,  
            (S.stacksize + STACKINCREMENT) *  
                sizeof (ElemType));  
        if (!S.base) exit (OVERFLOW); //存储分配失败  
        S.top = S.base + S.stacksize;  
        S.stacksize += STACKINCREMENT;  
    }  
    *S.top++ = e; //先赋值，top后加1  
    return OK  
}
```

入栈
27

出栈

```
Status Pop (SqStack &S, SElemType &e) {  
    // 若栈不空，则删除S的栈顶元素，  
    // 用e返回其值，并返回OK;  
    // 否则返回ERROR  
    if (S.top == S.base) return ERROR;  
    e = *--S.top;        //先top减1 ，后取值给e  
    return OK;  
}
```

顺序栈判满、判空条件

顺序栈满时: $S.top - S.base \geq S.stacksize$

顺序栈空时: $S.top = S.base$

注意:

非空栈中的栈顶指针始终在栈顶元素的下一个位置上。

如果在一个程序中需要使用多个栈，为了充分利用各个栈的空间，不产生栈空间过大造成系统空间紧张的问题，又要保证各个栈都能足够大以保存入栈的元素，不产生“上溢”现象，最好能采用多个栈共享空间的方法，即给多个栈分配一个足够大的数组空间，利用栈的动态特性，使其存储空间互相补充。

为说明简单，假定在程序中同时使用两个栈。给它们分配一个长度为 m 的数组空间，这时可将这两个栈的栈底设在数组的两端，让它们的栈顶向数组中间增长。这样当一个栈中元素较多时，就可以越过数组的中点，延伸到另一个栈的部分空间中去，当然前提是另一个栈中当前的元素不多。只有当两个栈的栈顶相迁时，才会发生“上溢”。显然，这比两个各有 $m/2$ 存储空间的栈产生“上溢”的概率要小。

栈的顺序存储结构方法三（多个栈共享空间）

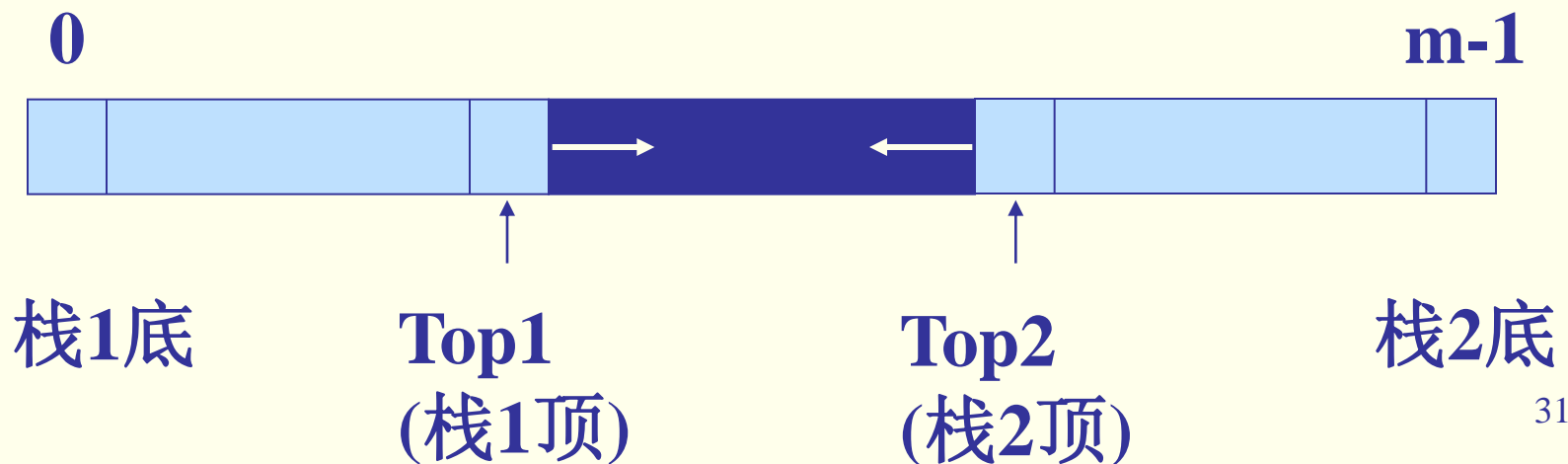
这种栈的数据结构可定义为：

```
typedef struct
```

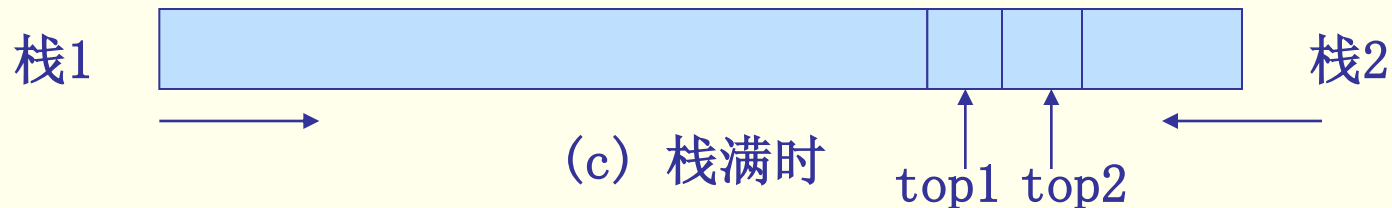
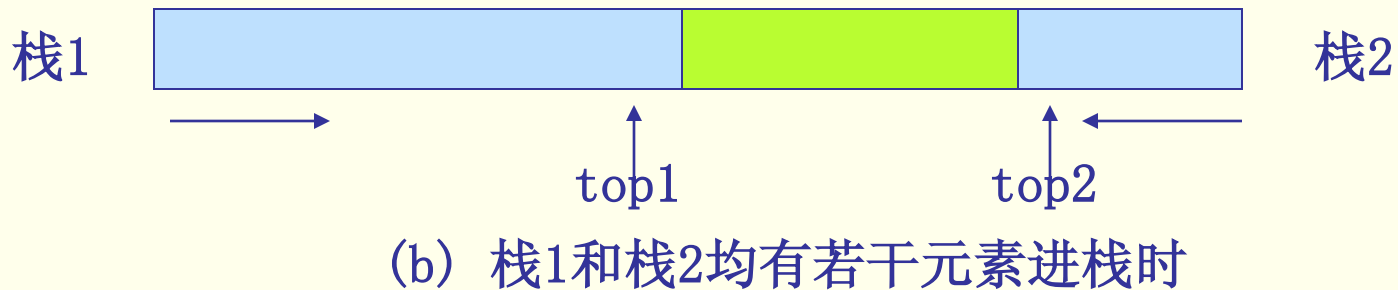
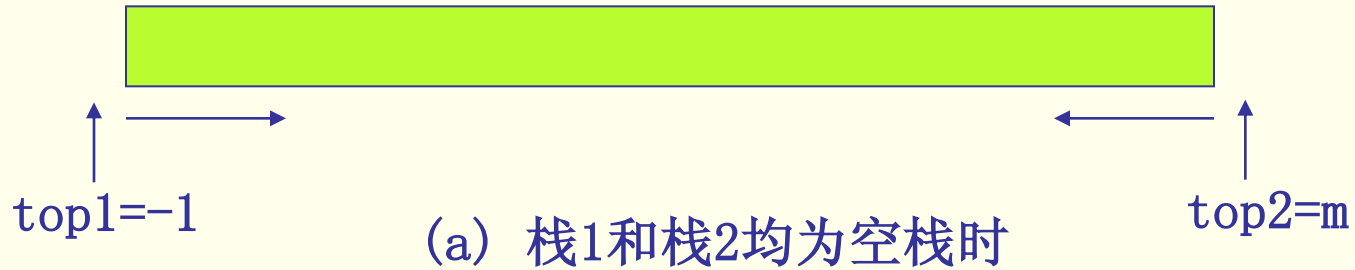
```
{ elemtype stack[m];
```

```
    int top1,top2;
```

```
} dustack;
```



两个栈共享一段存储空间

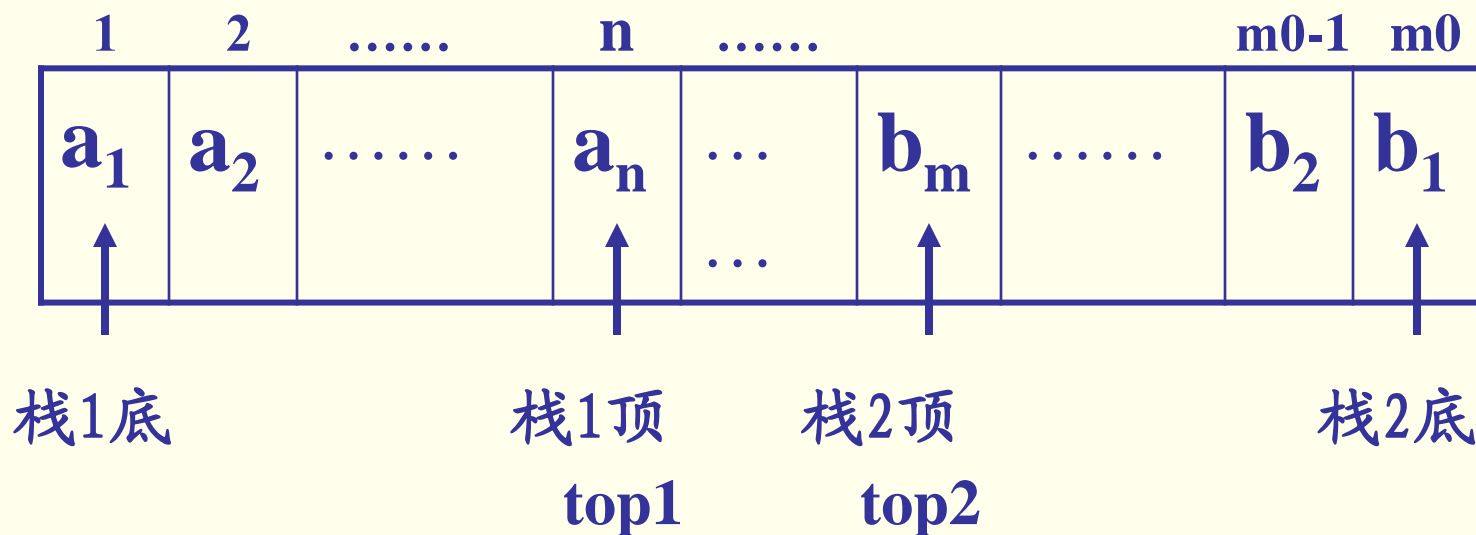


例子： 有两个栈S1和S2共享存储空间 $c[1,m0]$ ，其中一个栈底设在 $c[1]$ 处，另一个栈底设在 $c[m0]$ 处，分别编写S1和S2的进栈 $\text{push}(i,x)$ 、退栈 $\text{pop}(i)$ 和设置栈空 $\text{setnull}(i)$ 的函数，其中， $i=1, 2$ 。

注意： 仅当整个空间 $c[1,m0]$ 占满时才产生上溢。

该共享栈的结构如图所示，两栈的最多元素个数为 m_0 ， $top1$ 是栈1的栈顶指针， $top2$ 是栈2的栈顶指针，当 $top2=top1+1$ 时出现上溢；当 $top1=0$ 时栈1出现下溢，当 $top2=m_0+1$ 时栈2出现下溢。

C的元素序号



根据上述原理得到入栈函数如下:

```
void push (int i ,int x)
{
    if (top1==top2-1 )
        printf(“上溢!\n”);
    else if (i==1);           // 对第一个栈进行入栈操作
        { top1++;c[top1]=x;}
    else                       // 对第二个栈进行入栈操作
        { top2--; c[top2]=x;}
}
```

出栈函数如下:

```
void pop ( int i )
```

```
{ if (i==1);           // 对第一个栈进行出栈操作
```

```
    if (top1==0) printf(“栈1下溢!\n” ) ;
```

```
    else { x=c[top1];top1--;} 
```

```
else           // 对第二个栈进行出栈操作
```

```
    if (top2==m0+1) printf(“栈2下溢!\n” )
```

```
    else { x=c[top2]; top2++;}
```

```
}
```

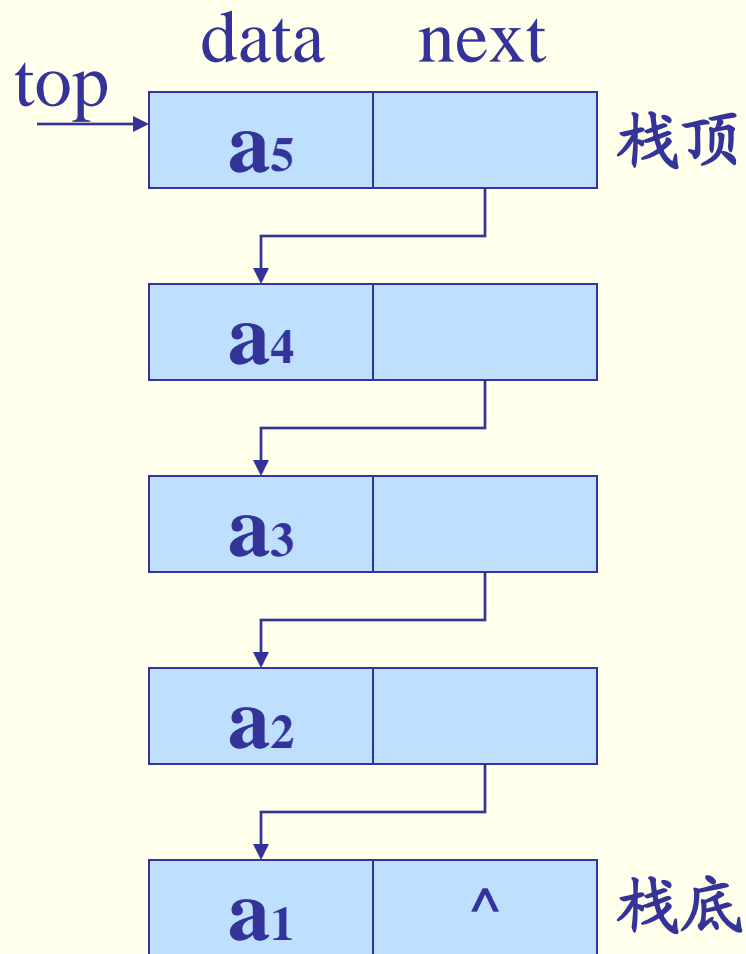
判栈空函数如下:

```
void setnull ( int i )  
    { if (i==1);  
        top1=0;  
    else  
        top2=m0+1;  
    }
```

二、 栈的链式存储结构

当最大需要容量事先不能估计时, 采用链式存储结构是有效的方法, 链栈的操作只能在栈顶处进行。

```
typedef struct snode
{ elemtype data;
  struct snode *next;
}*linkstack;
```



链式进栈操作

```
status push (linkstack top, elemtype x)
{
    t= (linkstack) malloc (sizeof(snode));
    if (t==NULL)
        return ERROR; //内存无可可用空间,栈上溢
    else { t->data=x; t->next=top;
          top=t;
          return OK; }
} //push
```

链式出栈操作

```
status pop (linkstack top, elemtype &e)
{
    if (top == NULL)    //栈空, 栈溢出 (下溢)
        return NULL;
    else {
        p=top;
        top=top->next;
        e=p->data;
        free (p);
        return OK;
    }
} //pop
```


对于链栈，不会产生单个栈满而其余栈空的情形，只有当系统空间全部用完，malloc过程无法实现时才会发生上溢，因此多个链栈共享空间也就是自然的事了。

可见，对栈这样一种元素多变的数据结构来说，链式存储结构似乎更适宜些。

此外初始化栈、栈判空操作也很容易实现。

3.3 栈的应用举例

例一、 数制转换

例二、 括号匹配的检验

例三、 行编辑程序问题

例四、 迷宫求解

例五、 表达式求值

例六、 实现递归

例一、 数制转换

算法基于原理:

$$N = (N \text{ div } d) \times d + N \text{ mod } d$$

例如:

$(1348)_{10} = (2504)_8$, 其运算过程如下:

	N	N div 8	N mod 8	
计算顺序 ↓	1348	168	4	↑ 输出顺序
	168	21	0	
	21	2	5	
	2	0	2	

```
void conversion ( ) {  
    InitStack(S);  
    scanf ("%d",N);  
    while (N) { //N不等于0时循环  
        Push(S, N % 8);  
        N = N/8;  
    }  
  
    while (!StackEmpty(S)) {  
        Pop(S,e);  
        printf ( "%d", e );  
    }  
} // conversion
```

数制转换的完整程序

```
#define stacksize 100;

typedef struct
{
    int base[stacksize];

    int top;
} stack;
```

```
int push(stack *s, int e)  
{  
    if (s->top>=stacksize) return 0;  
    s->base[s->top]=e;  s->top++; return 1;  
}
```

```
int pop(stack *s, int *e)  
{  
    if(s->top==0) return 0;  
    *e=s->base[--s->top];    return 1;  
} //top指向栈顶元素的下一个位置
```

main()

{

stack s1; int m,e,n;

s1.top=0; m=1348; n=8;

while(m)

{ push(&s1, m%n) ; m=m/n; }

while(s1.top!=0)

{ pop(&s1,&e); printf("%d",e);}

}

例二、 括号匹配的检验

假设在表达式中，

$([] ())$ 或 $[([] [])]$

等为正确的格式，

$[(])$ 或 $([())$ 或 $(()])$

均为不正确的格式。

则 检验括号是否匹配的方法可用

“期待的急迫程度” 这个概念来描述。

例如：考虑下列括号序列：

[([] [])]

1 2 3 4 5 6 7 8

分析可能出现的不匹配的情况：

- 到来的右括弧并非在所“期待”的；
- 到来的是“不速之客”；
- 直到结束，也没有到来所“期待”的括弧。

算法的设计思想:

- 1) 凡出现左括弧, 则 进栈;
- 2) 凡出现右括弧, 首先检查栈是否空
若栈空, 则表明该 “右括弧” 多余,
否则和栈顶元素比较,
若相匹配, 则 “左括弧出栈” ,
否则表明不匹配。
- 3) 表达式检验结束时,
若栈空, 则表明表达式中匹配正确,
否则表明 “左括弧” 有余。

Status matching(string& exp) { //只有圆括号，调用基本操作

int state = 1; i=1;

while (i<=Length(exp) && state) {

switch (exp[i]) {

case “(” : { Push(S,exp[i]); i++; **break**; }

case “)” : {

if(**NOT** stackEmpty(S)&&GetTop(S)=“(“)

{ Pop(S,e); i++; }

else state = 0; **break**; } //表示不匹配

default: { state = 0; **break**; } }

if (StackEmpty(S)&&state) **return** **OK**;

else **return** **ERROR**

} //matching

例三、行编辑程序问题

一个简单的行编辑程序的功能是：接收用户从终端输入的程序或数据，并存入用户的数据区。

如何实现？

“每接受一个字符即存入存储器” **?**

由于用户在终端上进行输入时，不能保证不出差错，因此，若在编辑程序中，“每接收一个字符即存入用户数据区”的做法显然不是最恰当的。

合理的作法是：

在用户输入一行的过程中，允许用户输入出差错，并在发现有误时可以及时更正。

实现：设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区，并假设“#”为退格符（前一个字符无效），“@”为退行符。

假设从终端接受了这样两行字符:

```
whli##ilr#e ( s#*s)
```

```
outcha@putchar(*s=#++);
```

则实际有效的是下列两行:

```
while (*s)
```

```
    putchar(*s++);
```

为此，可设这个输入缓冲区为一个栈结构，
每当从终端接收了一个字符之后先作如下判别：

- 如果它既不是退格符也不是退行符，则将该字符
压入栈顶；
- 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。

可用下述算法来描述：


```

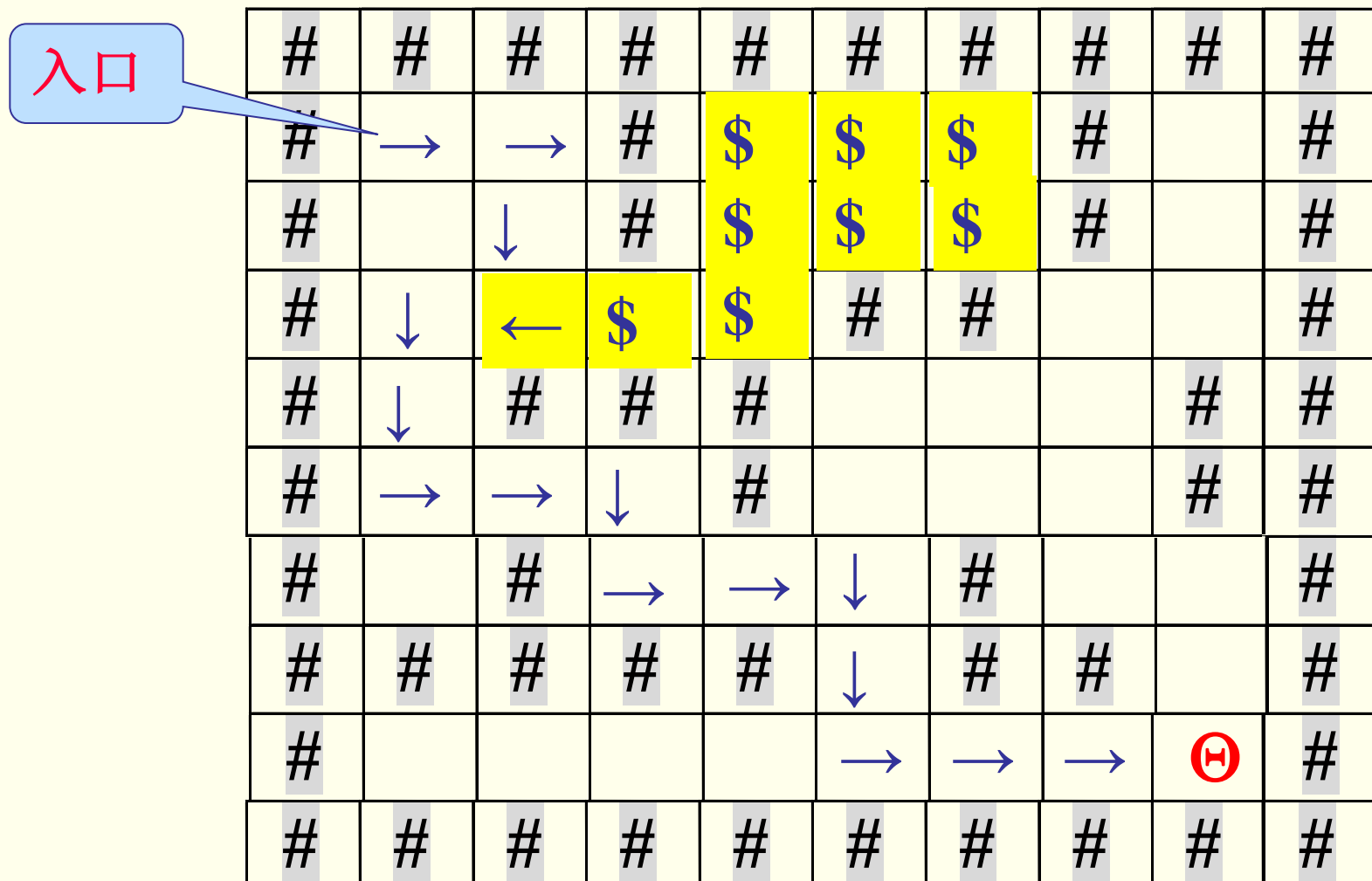
void LineEdit( )
{ InitStack(S);           //构造空栈
  ch=getchar( );          //从终端接收第一个字符
  while (ch!=EOF)         //EOF是全文结束符
  { while (ch!=EOF && ch!='\n')
    { switch (ch)
      { case '#': Pop(S,ch);    break;
        case '@': ClearStack(S); break; //重置S为空栈
        default: Push(S,ch); break;
      }
      ch=getchar( );          //从终端接收下一个字符
    }
    将从栈底至栈顶的栈内字符传送至调用过程的数据区;
    ClearStack (S);           //重置S为空栈
    if (ch!=EOF) ch=getchar( );
  }
  DestroyStack(S);         //栈S被销毁
} //LineEdit

```

例四、迷宫求解

从点的东向开始按顺时针旋转

通常用的是“穷举求解”的方法



求迷宫路径算法的基本思想是：

- 若当前位置“**可通**”，则纳入路径，继续前进；
- 若当前位置“**不可通**”，则后退，换方向继续探索；
- 若四周“**均无通路**”，则将当前位置从路径中删除出去。

求迷宫中一条从入口到出口的路径的算法:

设定当前位置的初值为入口位置;

do {

 若当前位置**可通**,

 则 { 将当前位置**插入栈顶**;

 若该位置是**出口**位置, 则算法**结束**;

 否则切换当前位置的**东**邻方块为

新的当前位置;

 }

 否则 {

A:

 }

} while (栈不空);

A:

若栈不空且栈顶位置尚有其他方向未被探索，
{则设定新的当前位置为：沿顺时针方向旋
转找到的栈顶位置的下一相邻块； }

若栈不空但栈顶位置的四周均不可通，
{则删去栈顶位置； //从路径中删去该通道块
若栈不空，则重新测试新的栈顶位置，
直至找到一个可通的相邻块或出栈至栈空；
}

若栈空，则表明迷宫没有通路。

例五、 表达式求值

表达式求值是程序设计语言编译中的一个基本问题，它的实现是栈应用的又一个典型例子。这里介绍一种简单直观，广为使用的算法，叫“**算符优先法**”。它是根据运算优先关系的规定来实现对表达式的编译或解释执行的。

例如：要对算术表达式 $4+2*3-10/5$ 求值。

算术四则运算规则：

- (1)先括号内，后括号外；
- (2)先乘除，后加减；
- (3)同级运算从左算到右。

“算符优先法”就是根据这个运算优先关系的规定来实现对表达式的编译或解释执行的。

表达式 { 操作数(operand): 常数、变量或常量标识符
运算符(operator): 算术运算符、关系运算符、
逻辑运算符等
界限符(delimiter): (、)、表达式结束符等

为了叙述的简洁，我们仅讨论简单算术表达式的求值问题，这种表达式仅包含加、减、乘、除、括号等四种运算。不难将它推广到更一般的表达式上。

我们把**运算符**和**界限符**统称为**算符**，它们构成的集合命名为 op ，在运算的每一步中，任意两个相继出现的运算符 θ_1 和 θ_2 之间的优先关系至多是下面三种关系之一：

$\theta_1 < \theta_2$: θ_1 的优先权低于 θ_2 ;

$\theta_1 = \theta_2$: θ_1 的优先权等于 θ_2 ;

$\theta_1 > \theta_2$: θ_1 的优先权高于 θ_2 ;

下表定义了算符之间的这种优先关系。

算符优先关系表

02 01 \	+	-	*	/	()	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

为实现算符优先算法可使用两个工作栈：

OPTR栈：用以寄存运算符；

OPND栈：用以寄存操作数或运算结果；

算法基本思想：

(1) 首先置操作数栈为空栈，表达式起始符 “#” 为运算符栈的栈底元素；

(2) 依次读入表达式中的每个字符，若是操作数，则进OPND栈；若是运算符，则和OPTR栈的栈顶运算符比较优先数后作相应操作，直至整个表达式求值完毕（即OPTR栈的栈顶元素和当前读入的字符均为“#”）。

```

OperandType EvaluateExpression( )           //求算术表达式的值
{ InitStack(OPTR); Push (OPTR,'#'); //初始化运算符栈
  InitStack(OPND); c=getchar( );           //初始化操作数栈
  while (c!='#' || GetTop(OPTR))!='#')
  { if (!In(c,OP)                          //读入的c不是运算符
      { Push ((OPND,c);c=getchar( );} //操作数进栈
    else switch (Precede(GetTop(OPTR),c)
    { case '<': Push(OPTR,c); c=getchar( );
      break; //栈顶元素优先权低, 运算符进栈
      case '=': Pop(OPTR,x); c=getchar( );
      break; //脱括号并接收下一字符
      case '>': Pop(OPTR, theta);
      Pop(OPND,b); Pop(OPND,a);
      Push(OPND,Operate(a,theta,b));
      break; //退栈并将运算结果入栈
    } //switch
  } //while
  return GetTop(OPND);
} // EvaluateExpression

```

二元运算符的表达式三种标识方法

表达式 ::= (操作数) + (运算符) + (操作数)

操作数 ::= 简单变量 | 表达式

简单变量 ::= 标识符 | 无符号整数

表达式的三种标识方法:

设 $\text{Exp} = \underline{\text{S1}} + \text{OP} + \underline{\text{S2}}$

则称 $\text{OP} + \underline{\text{S1}} + \underline{\text{S2}}$ 为前缀表示法

$\underline{\text{S1}} + \text{OP} + \underline{\text{S2}}$ 为中缀表示法

$\underline{\text{S1}} + \underline{\text{S2}} + \text{OP}$ 为后缀表示法

例如: $\text{Exp} = \underline{a \times b} + \underline{(c - d / e) \times f}$

前缀式: $+ \underline{\times a b} \underline{\times - c / d e f}$

中缀式: $\underline{a \times b} + \underline{c - d / e \times f}$

后缀式: $\underline{a b \times} \underline{c d e / - f \times} +$

结论:

- 1) 操作数之间的相对次序 **不变**;
- 2) 运算符的相对次序 **不同**;
- 3) 中缀式丢失了括弧信息,
致使运算的次序不确定;

4)前缀式的运算规则为:

连续出现的两个操作数和在他们之前且紧靠它们的运算符构成一个最小表达式;

5)后缀式的运算规则为:

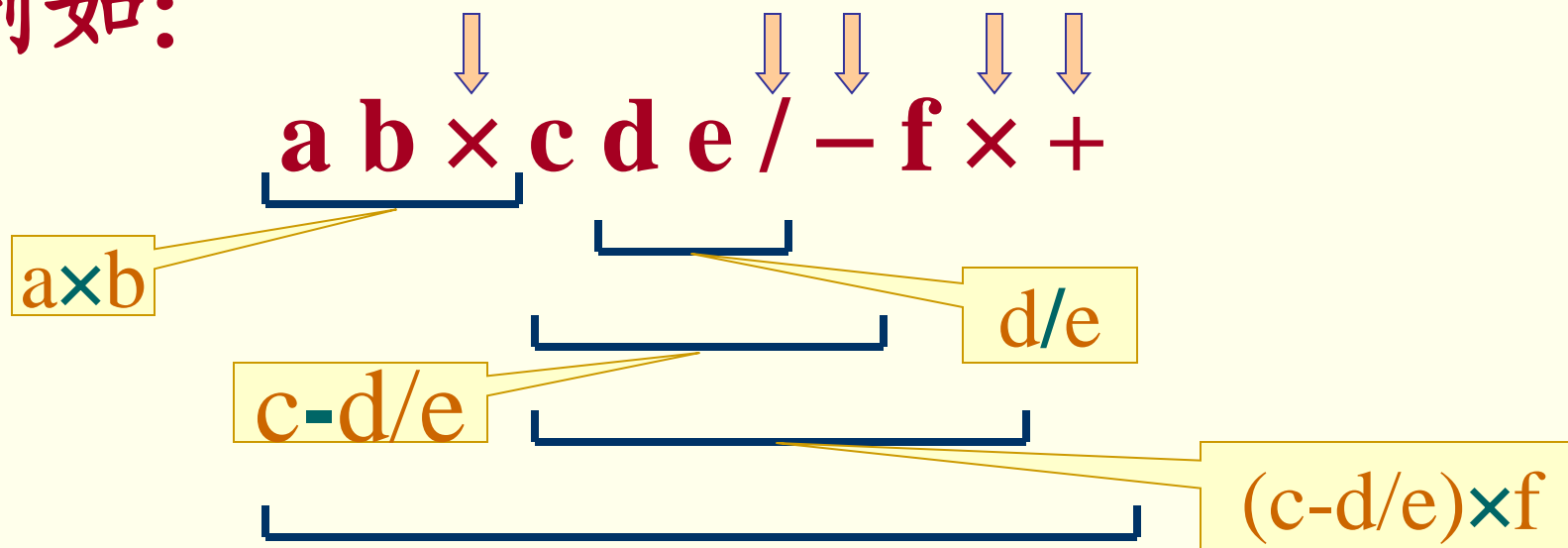
每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式;

运算符在表达式中出现的顺序恰为表达式的运算顺序。

如何从后缀式求值？

先找运算符，
再找操作数

例如：



●如何从原表达式求得后缀式？

分析 “原表达式” 和 “后缀式” 中的运算

原表达式: $a + b \times c - d / e \times f$

后缀式: $a b c \times + d e / f \times -$

每个运算符的运算次序要由它之后的一个运算符来定，在后缀式中，优先数高的运算符领先于优先数低的运算符。

从原表达式求得后缀式的规律为：

- 1) 设立运算符栈;
- 2) 设表达式的结束符为 “#”,
预设运算符栈的栈底为 “#”;
- 3) 若当前字符是操作数,
则直接发送给后缀式。

从原表达式求得后缀式的规律为：

- 4) 若当前运算符的优先数高于栈顶运算符，则进栈；
- 5) 否则，退出栈顶运算符发送给后缀式；
- 6) “(”对它之前后的运算符起隔离作用，“)”可视为自相应左括弧开始的表达式的结束符。

```

void transform(char suffix[ ], char exp[ ] )
    //从原表达式求得后缀式
    //s是运算符栈，s栈底预设 ‘#’， OP是运算符集合
{
    InitStack(S); Push(S, '#');    p = exp; ch = *p;
    while (!StackEmpty(S))
    { if (!IN(ch, OP))                // 若ch是操作数
        Pass( Suffix, ch);
      else { A: ... .. } // 若ch是运算符
      if ( ch!= '#' ) { p++; ch = *p; }
      else { Pop(S, ch); Pass(Suffix, ch); }
    } // while
} // CrtExptree

```

A:

switch (ch)

{ **case** '(' : Push(S, ch); **break**;

case ')' : Pop(S, c);

while (c!= '(')

{ Pass(Suffix, c); Pop(S, c) }

break;

若ch的优先级比c低，为真

default :

while(Gettop(S, c) && (precede(c,ch)))

{ Pass(Suffix, c); Pop(S, c); }

if (ch!= '#') Push(S, ch);

break;

} // switch

求后缀式的值:

- 1) 设立操作数栈S;
- 2) 设表达式的结束符为 “#”;
- 3) 读入表达式一个字符,若当前字符是操作数, 则压入栈中,转4)。

4) 若当前字符是运算符optr, 从栈中弹出2个数, 将运算结果再压入栈, 即:

$x2 = \text{POP}(S); \quad x1 = \text{POP}(S);$
 $\text{PUSH}(S, (x1 \text{ optr } x2));$

5) 读下一字符, 重复3) 和4) 直至读到结束符#;

6) 栈顶, $x = \text{POP}(S)$ 即后缀式相应的表达式的值。

```
int cal(char suffix-exp[ ] )
```

```
//求后缀式表达式的值
```

```
//s是运算数栈，OP是运算符集合
```

```
{  
  InitStack(S); i = 0; ch = suffix-exp[0];
```

```
  while (ch<>'#')
```

```
    { if (!IN(ch, OP))                // 若ch是操作数
```

```
      Push( S, ch);
```

```
    else                               // 若ch是运算符
```

```
      { x2=POP(S);  x1=POP(S); //取出两个操作数
```

```
        PUSH(S,(x1 ch x2)); }
```

```
    i++; ch= suffix-exp[i];
```

```
  } // while
```

```
} //cal
```


例六、实现递归

当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三项任务：

- 将所有的实在参数、返回地址等信息传递给被调用函数保存；
- 为被调用函数的局部变量分配存储区；
- 将控制转移到被调用函数的入口。

从被调用函数返回调用函数之前，应该完成下列三项任务：

- 保存被调函数的计算结果；
- 释放被调函数的数据区；
- 依照被调函数保存的返回地址将控制转移到调用函数。

多个函数嵌套调用的规则是：

后调用先返回！

此时的内存管理实行“栈式管理”

例如：

```
void main( ){    void a( ){    void b( ){  
    ...          ...          ...  
    a( );        b( );  
    ...          ...  
} //main        } // a      } // b
```

函数b的数据区

函数a的数据区

Main的数据区

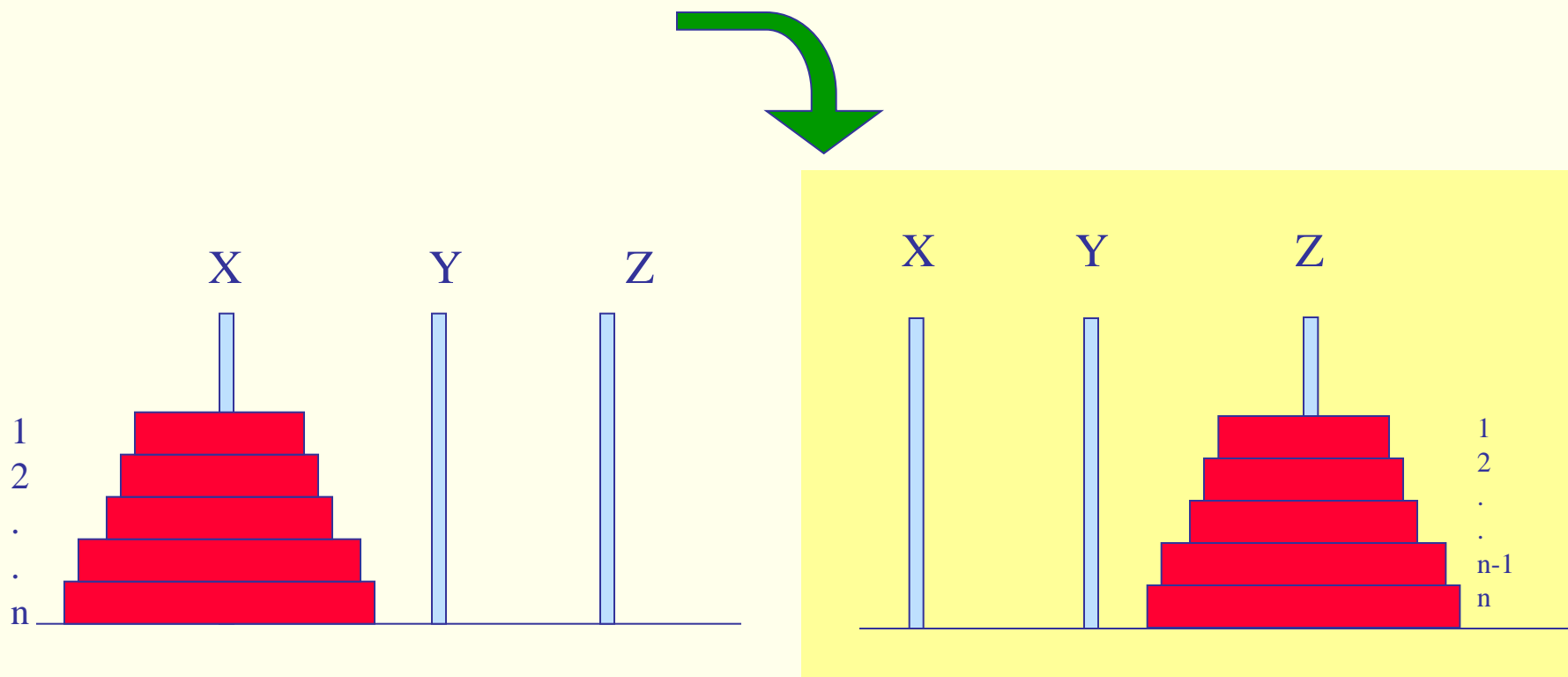
函数之间的信息传递和控制转移必须通过“栈”来实现，系统将整个程序运行时所需要的数据空间安排在一个栈内，每当调用一个函数时，就为它在栈顶分配一个存储区，每当从一个函数退出时，就释放它的存储区。所以**当前运行的函数的数据区必在栈顶**。

一个递归函数的运行过程类似于多个函数的嵌套调用，只是调用函数与被调用函数是同一个函数。注意：递归函数运行的“层次”。主函数是第0层，从主函数调用递归函数为进入第1层。

每进入一层递归，就产生一个新的工作记录（包括上一层的返回地址、实在参数、局部量）压入栈顶。每退出一层递归，就从栈顶弹出一个工作记录。

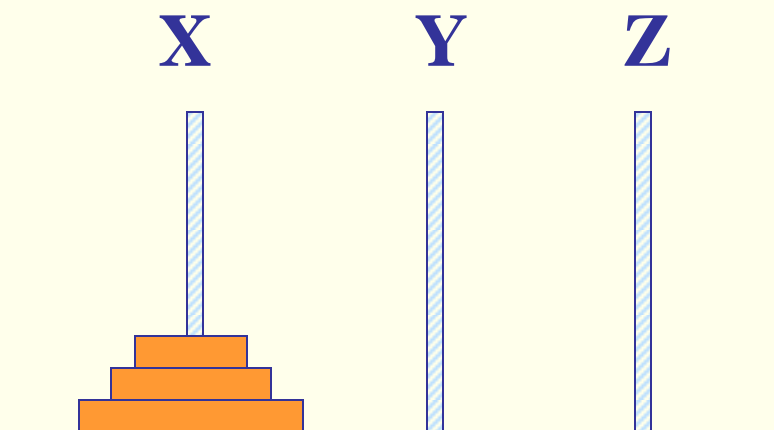
例3-2 n 阶Hanoi问题。

假设有三个分别命名为X、Y和Z的塔座，在塔座X上插有 n 个直径大小各不相同、依小到大编号为1, 2, ..., n 的圆盘。现要求将X轴上的 n 个圆盘移至塔座Z上，并仍按同样顺序叠排。



圆盘移动时必须遵循下列规则：

- 1) 每次只能移动一个圆盘；
- 2) 圆盘可以插在X、Y和Z中的任一塔座上；
- 3) 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

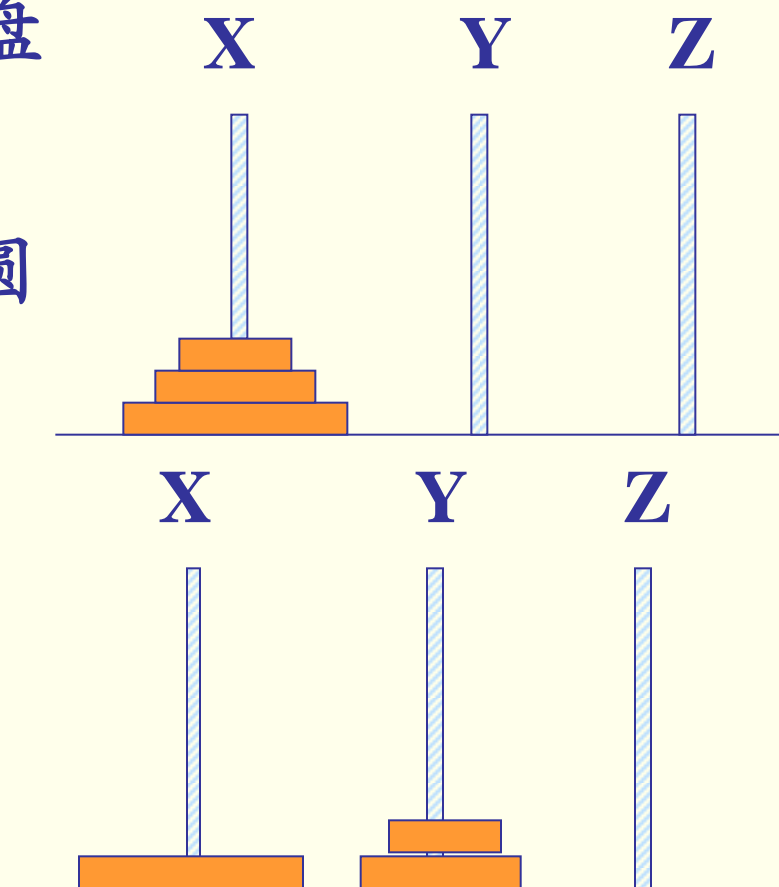


如何实现圆盘的移动操作?

当 $n=1$ 时，从塔座 $X \rightarrow Z$;

当 $n>1$ 时，利用塔座 Y 作辅助塔座，若能设法将压在编号为 n 的圆盘之上的 $n-1$ 个圆盘从塔座 $X \rightarrow Y$ 上，则可先将编号为 n 的圆盘从塔座 $X \rightarrow Z$ ，然后再将塔座 Y 上的 $n-1$ 个圆盘移至塔座 Z 上。

而如何将 $n-1$ 个圆盘从一个塔座移至另一个塔座的问题是一个和原问题具有相同特征属性的问题，只是问题的规模小1，因此可以用同样的方法求解。



```
void hanoi (int n, char x, char y, char z)
```

```
    // 将塔座x上按直径由小到大且至上而下编号为1至n
```

```
    // 的n个圆盘按规则搬到塔座z上, y可用作辅助塔座。
```

```
1 {
```

```
2  if (n==1)
```

```
3     move(x, 1, z);      // 将编号为 1 的圆盘从x移到z
```

```
4  else {
```

```
5     hanoi(n-1, x, z, y); // 将x上编号为 1 至n-1的
                                // 圆盘移到y, z作辅助塔
```

```
6     move(x, n, z);      // 将编号为n的圆盘从x移到z
```

```
7     hanoi(n-1, y, x, z); // 将y上编号为 1 至n-1的
                                // 圆盘移到z, x作辅助塔
```

```
8     }
```

```
9 }
```



```

void hanoi (int n, char x, char y, char z)
{
1  if (n==1)
2    move(x, 1, z);
3  else
4    { hanoi(n-1, x, z, y);
5      move(x, n, z);
6      hanoi(n-1, y, x, z);
7    }
8 }

```

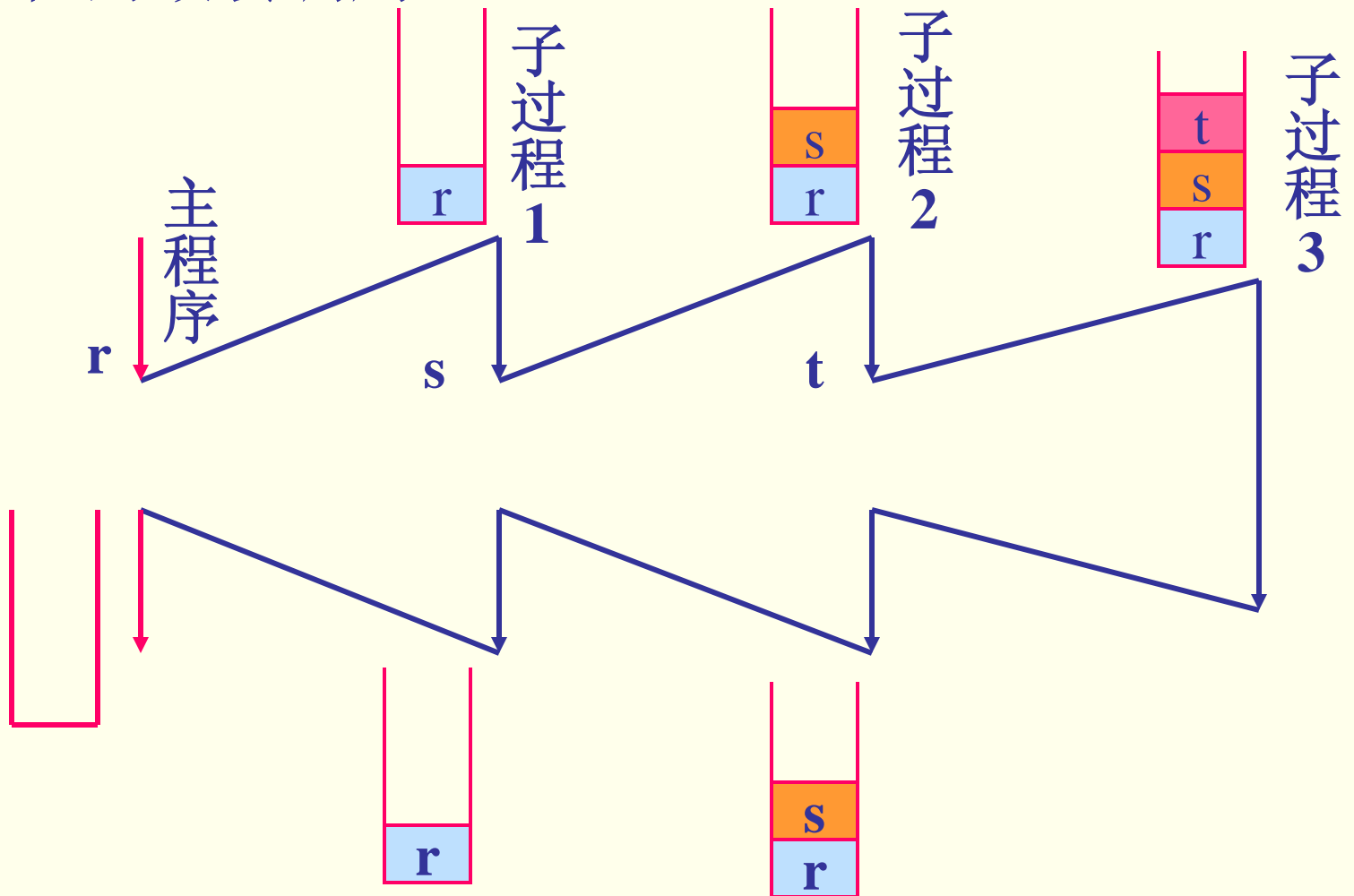
5	2	a	c	b
0	3	a	b	c

语句标号

返址 n x y z

栈的应用

过程的嵌套调用



地图四染色问题

“四染色”定理是计算机科学中著名的定理之一。

使地图中相邻的国家或行政区域不重色，最少可用四种颜色对地图着色。

证明此定理的结论，利用栈采用回溯法对地图着色。

思想：对每个行政区编号：1-7

对颜色编号；a、b、c、d；

从第一号行政区开始逐一染色，每一个区域逐次用四种颜色进行试探，若所取颜色与周围不重，则用栈记下来该区域的色数，否则依次用下一色数进行试探。若出现a-d均与周围发生重色，则需退栈回溯，修改当前栈顶的色数。

地图四染色举例

R[7][7]

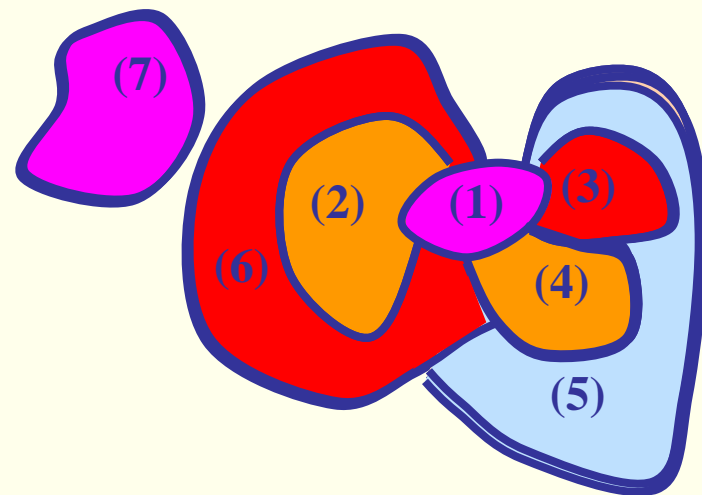
	1	2	3	4	5	6	7
1	0	1	1	1	1	1	0
2	1	0	0	0	0	1	0
3	1	0	0	1	1	0	0
4	1	0	1	0	1	1	0
5	1	0	1	1	0	1	0
6	1	1	0	1	1	0	0
7	0	0	0	0	0	0	0

1# 紫色

2# 黄色

3# 红色

4# 蓝色



S[K]	1	2	3	4	5	6	7
	1	2	3	2	4	3	1

```
Void mapcolor(int R[][],int n,int s[])
```

```
{ s[1]=1; // 1号区域染1色
  a=2; J=1; // a为区域号, J为染色号
  while ( a<=n)
  { while(( J<=4)&&(a<=n))
    { k=1; // k表示已经着色的区域号
      while(( k<a)&&(s[k]*R[a-1][k-1]!=J)) k=k+1;
      // 若不相邻, 或若相邻且不重色, 对下一个区域判断。
      IF (k<a) J=J+1; //相邻且重色
      ELSE{ s[a]=J; a=a+1; J=1; } //相邻且不重色
    }
    IF (J>4) { a=a-1; J=s[a]+1 }
  }
}
```

皇后问题求解

设n皇后问题的解为 $(x_1, x_2, x_3, \dots, x_n)$,

约束条件为：其中任意两个 x_i 和 x_j 不能位于棋盘的同行、同列及同对角线。

如何表示棋盘中放置的棋子？

由于每行、列及对角线上只能有一个棋子，因而对每列来说，只需记录该列中棋子所在的行号，故用一维数组即可。

按四皇后问题求解举例

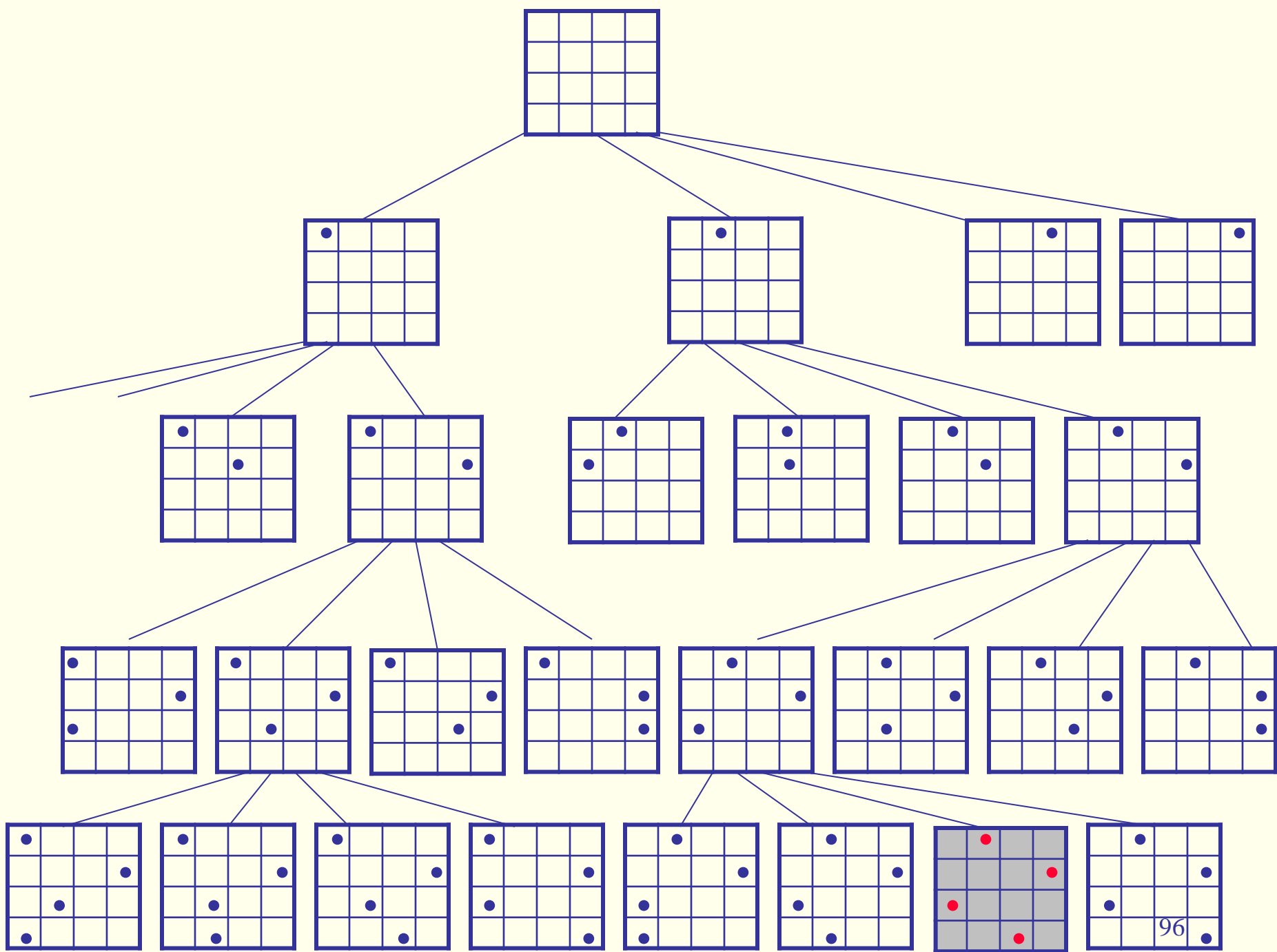
设四皇后问题的解为 (x_1, x_2, x_3, x_4) ,

其中: $x_i (i=1,2,3,4) \in S_i=\{1, 2, 3, 4\}$

约束条件为: 其中任意两个 x_i 和 x_j 不能位于棋盘的同行、同列及同对角线。

按回溯法的定义, 皇后问题求解过程为:

解的初始值为空; 首先添加 $x_1=1$, 之后添加满足条件的 $x_2=3$, 由于对所有的 $x_3 \in \{1,2,3,4\}$ 都不能找到满足约束条件的部分解 (x_1, x_2, x_3) , 则回溯到部分解 (x_1) , 重新添加满足约束条件的 $x_2=4$, 依次类推(按行存列号)。



void queen(int i, int n)

{ // 进入本函数时，在 $n \times n$ 棋盘前 $i-1$ 行已放置了互不攻击的 $i-1$ 个棋子。现从第 i 行起继续为后续棋子选择满足约束条件的位置。当求得 $(i>n)$ 的一个合法布局时，输出之。

if ($i>n$) 输出棋盘的当前布局;

else for ($j=1$; $j \leq n$; $++j$)

{ 在第 i 行第 j 列放置一个棋子;

if (当前布局合法) **queen**($i+1$, n);

移去第 i 行第 j 列的棋子;

}

} // trial

```

#include<stdio.h>
#define n 8 // n为皇后个数, m为摆法计数
int m=0,a[n]; // a[i]存放第i个皇后放置的行号,
int ok(int i, int j) //检查(i,j)上能否放棋子
{ j1=j; i1=i; ok1=1; //检查第i行上能否放棋子
  while( (j1>1)&&ok1)
    {j1--; ok1=a[j1]!=i ;}
  j1=j; i1=i; //检查对角线上能否放棋子
  while( (j1>1)&&(i1>1)&&ok1)
    {j1--; i1--; ok1=a[j1]!=i1 ;}
  j1=j; i1=i; //检查另一对角线上能否放棋子
  while( (j1>1)&&(i1<n)&&ok1)
    {j1--; i1++; ok1=a[j1]!=i1 ;}
  return ok1
}

```

```

Void queen(int j) //从第j列开始逐个试探
{ if (j>n)
    { m++; printf("m=%d  ",m);
      for (i=1;i<=n;i++) printf("  %d",a[i]);
      printf( "\n" );
    }
  else for( i=1; i<=n;i++)
      if(ok(i,j)) //检查(i,j)上能否放棋子
      { a[j]=i; //在(i,j)上放一个棋子
        queen(j+1) ;
      }
}

main()
{queen(1);}

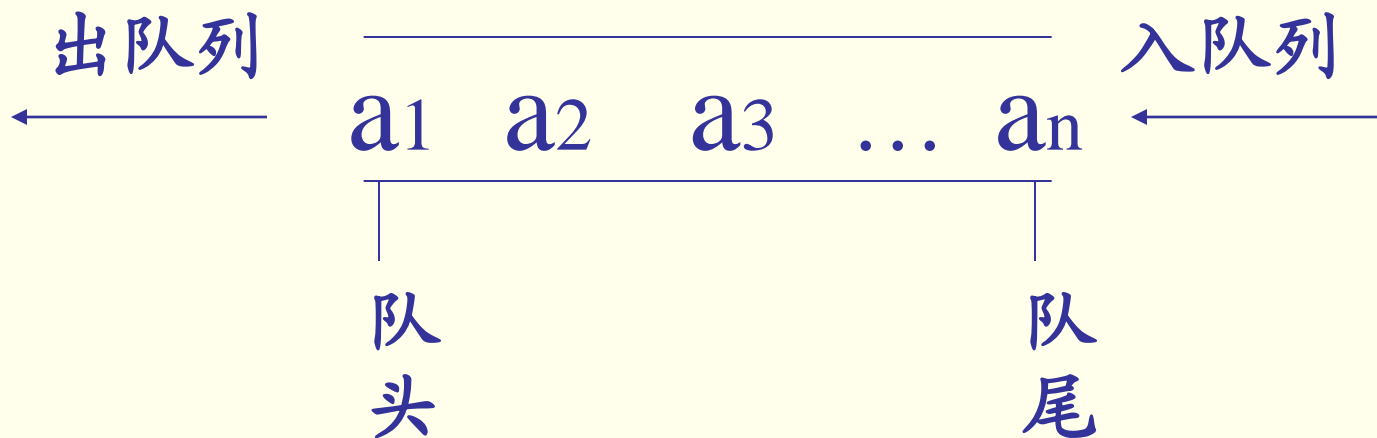
```

3.4 队列的类型定义

一、队列的定义

- 队列是只允许在表的一端进行插入，在另一端删除元素的线性表；
- 允许插入的一端称为队尾(rear)；
- 允许删除的一端称为队头(front)；
- 假设队列为 $Q=(a_1, a_2, \dots, a_n)$ ，则 a_1 是队头元素， a_n 是队尾元素；
- 队列中的元素是按照 a_1, a_2, \dots, a_n 的顺序进入的，退出队列也只能按照这个次序依次退出；
- 当队列中没有元素时称为空队列；
- 队列是一种“先进先出”的线性表，简称FIFO表。

队列示意图



二、队列的抽象数据类型的类型定义

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:

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

约定其中 a_1 端为队列头, a_n 端为队列尾

基本操作:

} ADT Queue

队列的基本操作:

InitQueue(&Q)

DestroyQueue(&Q)

QueueEmpty(Q)

QueueLength(Q)

GetHead(Q, &e)

ClearQueue(&Q)

EnQueue(&Q, e)

DeQueue(&Q, &e)

QueueTravers(Q, visit())

InitQueue(&Q)

操作结果： 构造一个空队列Q。

DestroyQueue(&Q)

初始条件： 队列Q已存在。

操作结果： 队列Q被销毁，
不再存在。

QueueEmpty(Q)

初始条件： 队列Q已存在。

操作结果： 若Q为空队列，则返回
TRUE，否则返回FALSE。

QueueLength(Q)

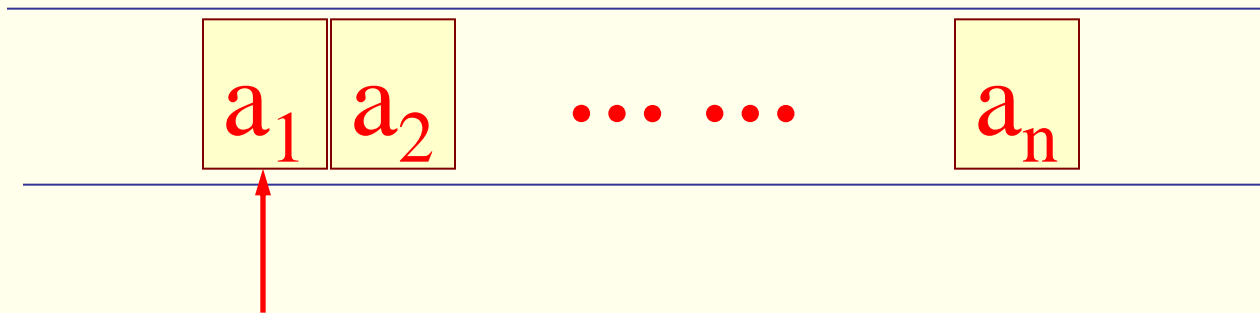
初始条件： 队列Q已存在。

操作结果： 返回Q的元素个数，
即队列的长度。

GetHead(Q, &e)

初始条件： Q为非空队列。

操作结果： 用e返回Q的队头元素。



ClearQueue(&Q)

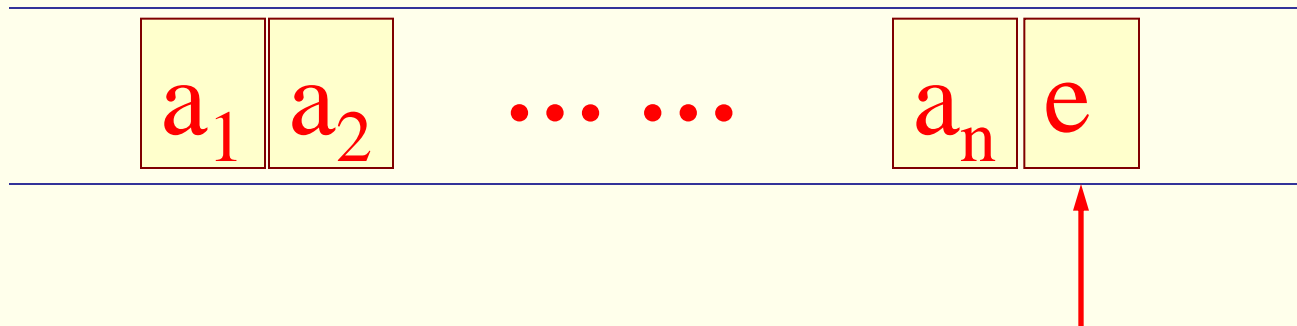
初始条件： 队列Q已存在。

操作结果： 将Q清为空队列。

EnQueue(&Q, e)

初始条件： 队列Q已存在。

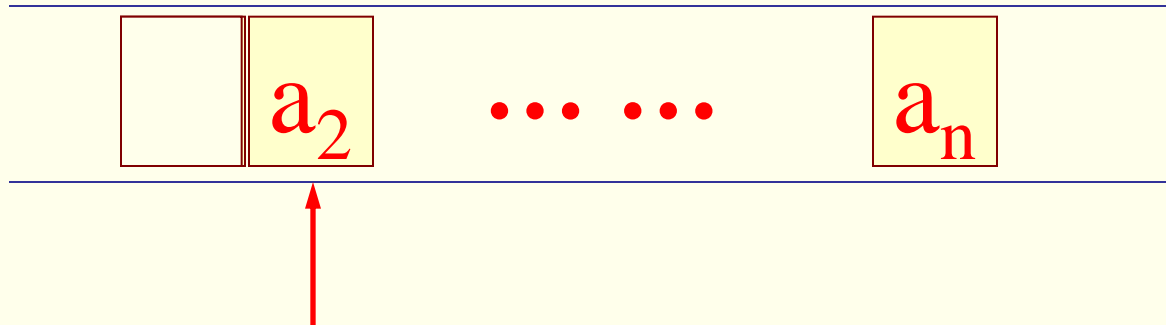
操作结果： 插入元素e为Q的新的
队尾元素。



DeQueue(&Q, &e)

初始条件： Q为非空队列。

操作结果： 删除Q的队头元素，
并用e返回其值。



QueueTravers(Q, visit())

初始条件： Q为非空队列。

操作结果： 从队头到队尾，依次对Q的每个数据元素调用函数visit()。一旦Visit()失败，则操作失败。

3.5 队列类型的实现

循环队列 —— 顺序映像

链 队 列 —— 链式映像

1、队列的顺序存贮结构

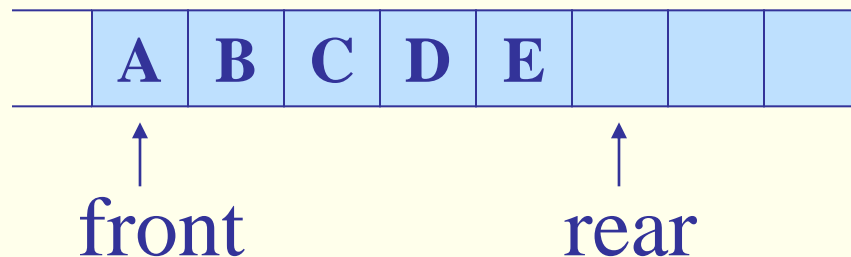
队列的顺序存储结构称为**顺序队列**，采用**一维数组**来存储队列中的每一元素，数组的**上界**表示队列的**最大容量**。另设两个指针**front** 和 **rear**，分别指示队头元素和队尾元素在队列中的位置。

约定：

队头指针**front**总是指向队头元素在队列中的当前位置；

队尾指针**rear**总是指示队尾元素的**下一个**位置；

(1) 一般的队列



- C语言描述:

```
#define MAXQSIZE n
```

```
typedef struct
```

```
{ elemtype queue[MAXQSIZE]; // 静态分配
```

```
    int front , rear ;
```

```
} sequeuetp;
```

- C语言描述:

```
#define MAXQSIZE n
```

```
typedef struct
```

```
{ elemtype *queue;
```

//动态分配

```
int front , rear ;
```

```
} sequeuetp;
```

例：顺序队列Q的C语言描述是：

```
#define MAXQSIZE 8
```

```
typedef struct
```

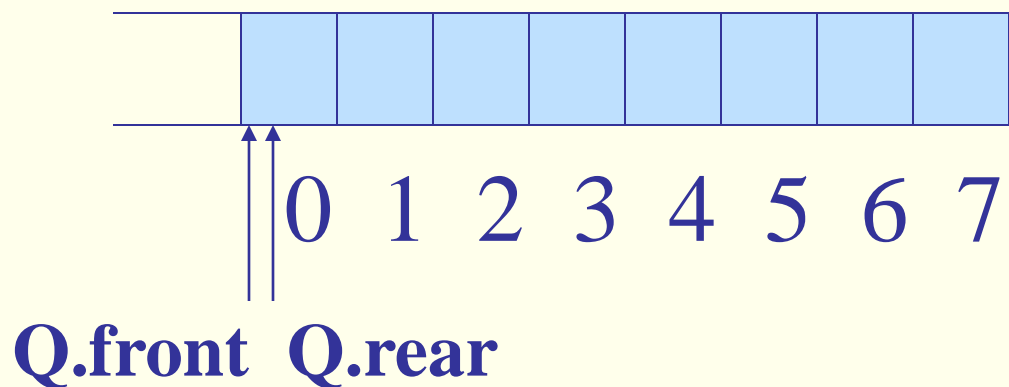
```
{ char queue[MAXQSIZE];
```

```
    int front, rear;
```

```
} sequeuetp;
```

```
sequeuetp Q;
```

(a) 队列的初始状态 (空队列):



状态特征: **$Q.front = Q.rear = 0$**

初始化算法

```
void initqueue (sequeuetp Q)
```

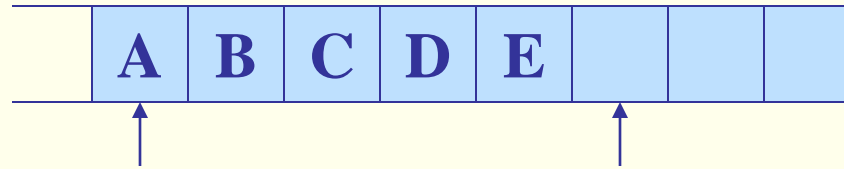
```
{ Q.front= 0;
```

```
  Q.rear= 0;    //设置队头指针和队尾指针均指向
```

```
                //数组下界
```

```
}
```

(b) 元素入队列后，队列的状态：



Q.front

Q.rear

入列操作：

Q.queue [Q.rear]='A';	Q.rear+ +;
Q.queue [Q.rear]='B';	Q.rear+ +;
Q.queue [Q.rear]='C';	Q.rear+ +;
Q.queue [Q.rear]='D';	Q.rear+ +;
Q.queue [Q.rear]='E';	Q.rear+ +;

(c) 元素 A, B, C 出列后的状态:

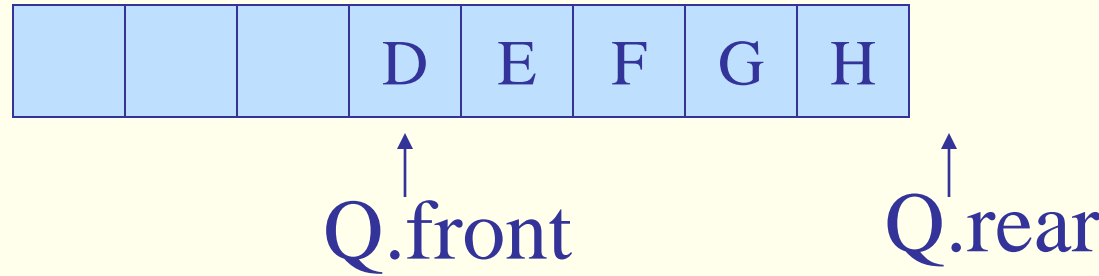


Q.front Q.rear

出列操作:

```
ch= Q.queue [Q.front];    Q.front ++;  
ch= Q.queue [Q.front];    Q.front ++;  
ch= Q.queue [Q.front];    Q.front ++;
```


(d)元素F,G,H,I 继续入列:



入列操作:

`Q.queue[Q.rear]='F';`

`Q.rear ++;`

`Q.queue[Q.rear]='G';`

`Q.rear ++;`

`Q.queue[Q.rear]='H';`

`Q.rear ++;`

‘I’不能进入队列, 因
Q.rear=8,此时队列已满.

状态特征为:

Q.rear=MAXSIZE

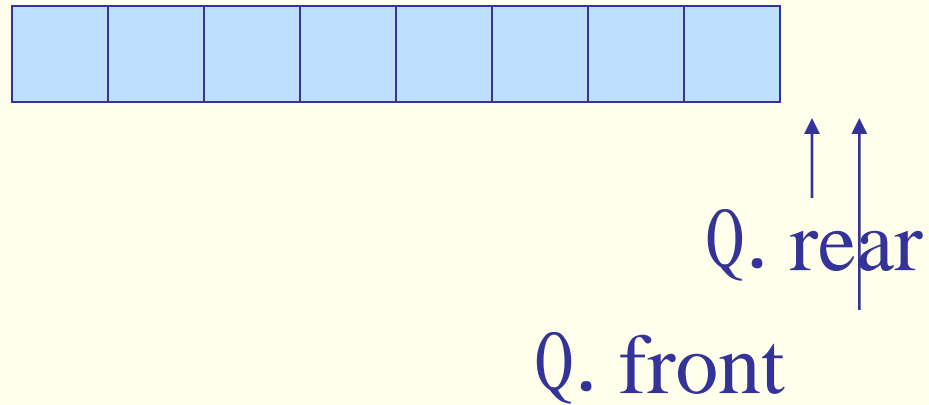
入队列算法

```
status enqueue(sequeuetp &Q, elemtype x)
{ if (Q.rear==MAXSIZE)
    return error;           //队列已满
else { Q.queue[Q.rear]=x;
      Q.rear++;
      return ok;
    }
}
```

出队列算法

```
status delqueue(sequeuetp &Q, elemtype &e)
{ if (Q.front==Q.rear)
    return (NULL);           //队列空
else { e=Q.queue[Q.front];
      Q.front++;
      return OK;
    }
}
```

(e) 元素D,E,F,G,H出列后, 队列之状态:



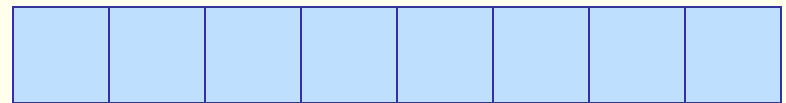
状态特征: $Q. front = Q. rear$

- 1) 队列中无元素, 是空队列;
- 2) 因 $rear = maxsize$, 此时队列被看作满, 不能执行入队列操作, 即溢出。

假溢出

- $\text{rear}=\text{maxsize}$ 不一定说明队列中有 maxsize 个元素, 如上例实际上还有可用空间, 称**假溢出**;
- 解决“**假溢出**”的办法一般有两种:
 - 将整个**队列左移**, 使队列的第一个元素重新位于 $\text{queue}[0]$, 且置 $\text{front}=0$; 但会引起大量元素的移动, 增加算法的运行时间;
 - 设想 $\text{queue}[0]$ 接在 $\text{queue}[\text{maxsize}-1]$ 之后, 使一维数组 queue 成为一个**首尾相接的环**, 使得:

if $\text{Q.rear} + 1 = \text{maxsize}$
 $\text{Q.rear}=0$;



Q.rear ↑
 Q.front ↑

(2) 循环队列 入队列算法

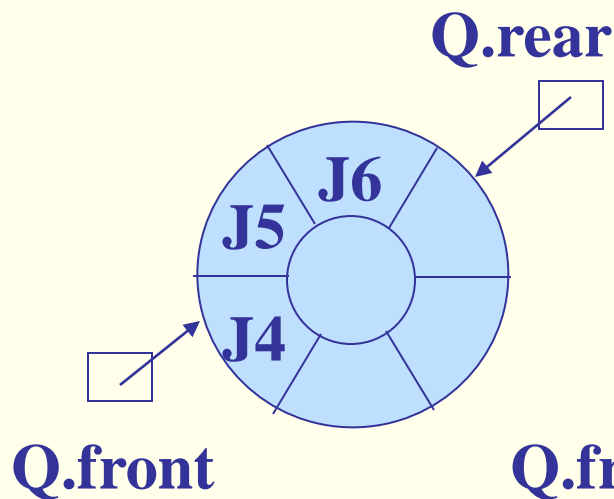
```
status enqueue(sequeuetp &Q, elemtype x)
{ Q.queue[Q.rear]=x;
  Q.rear=(Q.rear+1)%maxsize;
  return ok;
} //没有判断队列是否已满
```

循环队列

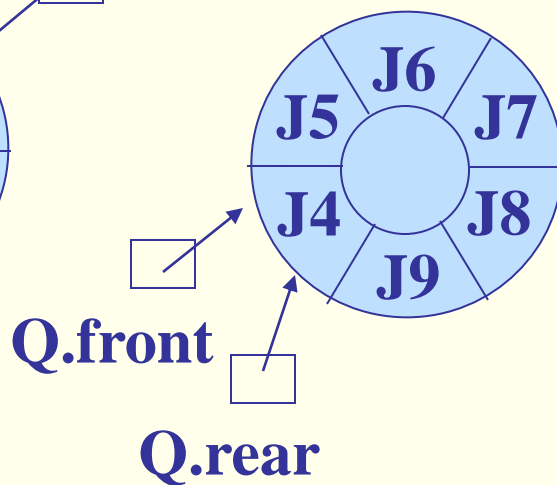
出队列算法

```
status delqueue(sequeue_t Q, elemtype &e)
{
    e=Q.queue[Q.front];
    Q.front=(Q.front+1)%maxsize;
    return OK;
} //没有判断队列是否已空
```

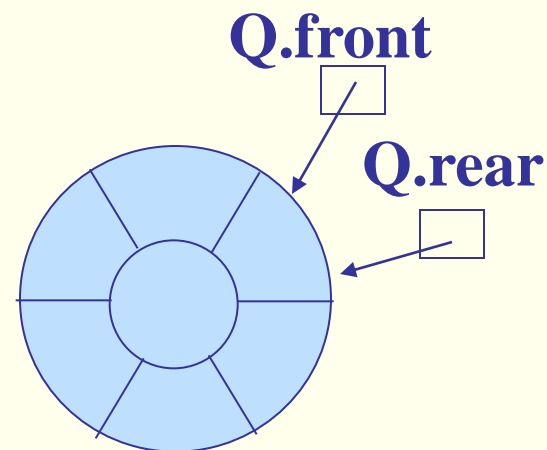
在循环队列中如何判定队满和队空？



一般情况



队列满



队列空

$Q.front = Q.rear$ $Q.front = Q.rear$

两种方法:

(1) 设置一个**标志flag**，以区别队列是满还是空。其基本思想是：该标志用于标记在队列中执行的最后一次操作。**flag**初始化为“**delete**”。在每一次**插入**操作后，该**标志**被置为“**entry**”；在每一次**删除**操作后，该**标志**被置为“**delete**”。当出现 $\text{front}=\text{rear}$ 时，通过检查**flag**的值即可确定此时队列是满还是空。如果**flag**此时为“**entry**”，则可判断队列是满的，否则队列是空的。

不过标志**flag**的使用**减慢了**队列插入和删除的**速度**。由于在采用队列的问题中，进队列和出队列操作是很频繁用到的，减少时间的花费是很重要的，因此常常采用后一种方法。

(2) 少用一个数据元素空间，以队尾指针加1等于队头指针来表示队列**满**。判别公式为：

$$(\text{rear}+1)\% \text{MAXSIZE} = \text{front}$$

这样规定后，在数组中队尾指针所指的位置始终是空的，即有MAXSIZE个分量的循环空间只能表示长度不超过MAXSIZE-1的队列，但可以避免另外再设标志。此时**rear=front**的条件还可用来判断队列是否为**空**。这时，入队列和出对列算法如下：

循环队列的入队算法

```
status encycqueue(sequeue_t Q, elemtype x)
{ if ((Q.rear+1)%MAXSIZE==Q.front)
    return error;      // 队列满
else { Q.queue[Q.rear]=x;
      Q.rear=(Q.rear+1)%MAXSIZE;
                        // 队尾指针循环加1
    return OK;
  }
}
```

循环队列的出队算法

```
status delcycque(sequeuetp &Q, elemtype &e)
{ if (Q.front==Q.rear)
    return(NULL);           //队列空
else
    { e= Q.queue[Q.front]
      Q.front=(Q.front+1)%MAXSIZE;
                                //队头指针循环加1
      return OK;
    }
}
```

2、队列的链式存储结构

对于在使用中数据元素变动较大的数据结构来说，用链式存储结构比用顺序存储结构更有利。显然，队列也是这样一种数据结构。

用链表表示的队列简称链队列。一个链队列显然需要两个分别指示队头和队尾的指针才能唯一确定。为了操作方便起见，我们也给链队列添加一个头结点，并令头指针指向头结点。

由此，空的链队列的判定条件为头指针与尾指针均指向头结点。

链队列的操作即为单链表的插入和删除操作的特殊情况，只是尚需修改尾指针和头指针。

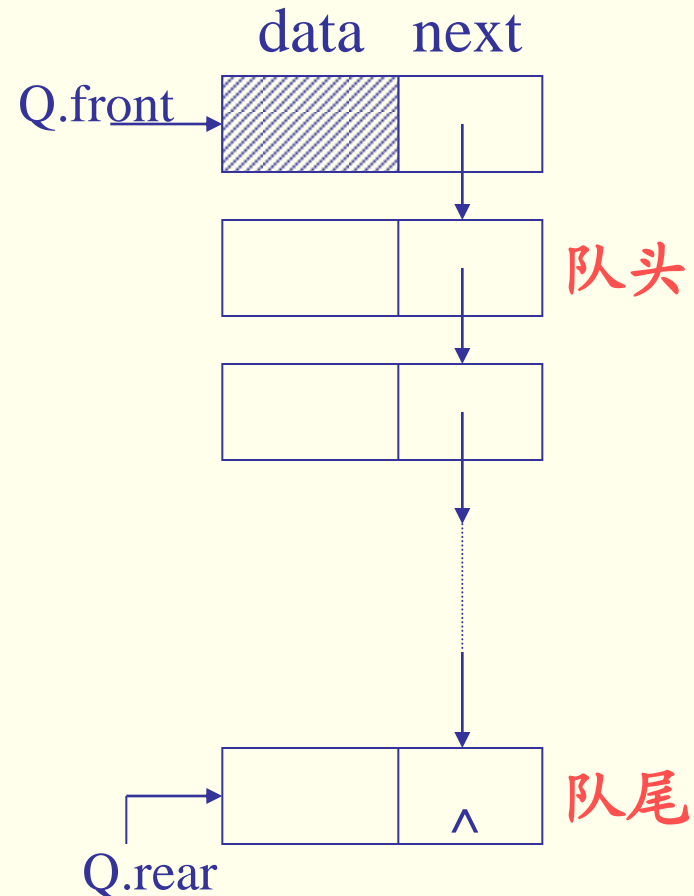
链队列的c语言描述:

结点结构: //定义链队列中的一个结点结构

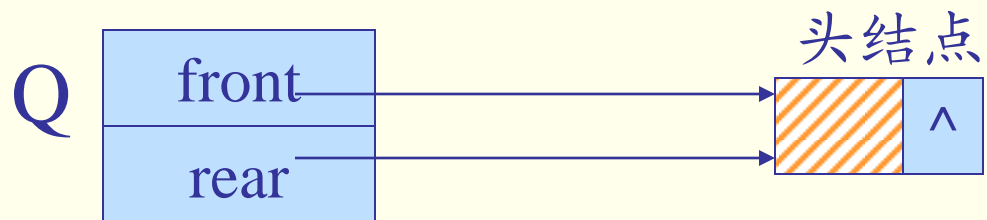
```
typedef struct qnode
{ elemtype data ;
  struct qnode * next ;
} queueptr;
```

队列结构://定义链队列结构

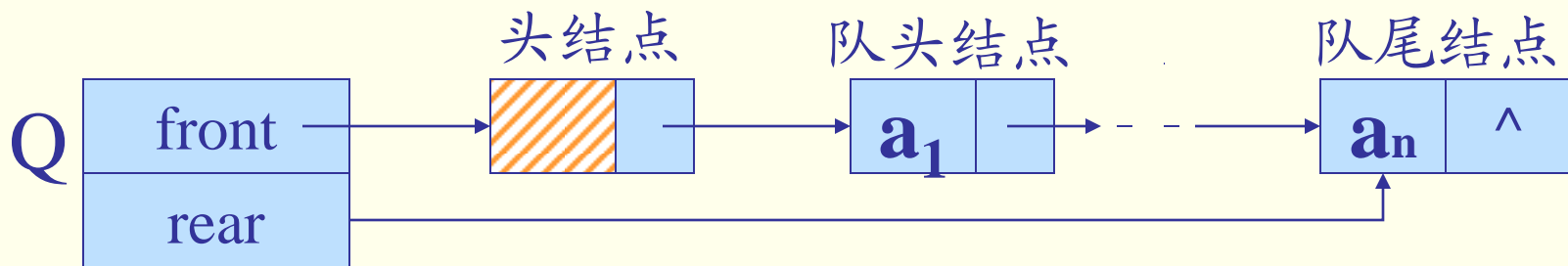
```
typedef struct
{ queueptr *front ;
  queueptr *rear ;
} *linkqueue;
```



链队列的示意图:



(a) 空链队列



(b) 非空链队列

链队列的初始化算法

```
void initqueue(linkqueue Q)
```

```
{  Q.front=(queueptr*)malloc(sizeof(qnode));
```

//产生一个头结点

```
if(!Q.front) exit(OVERFLOW);    //存储分配失败
```

```
Q.rear= Q.front;    //头、尾指针均指向头结点
```

```
Q.front → next=NULL;
```

```
}
```


销毁链队列算法

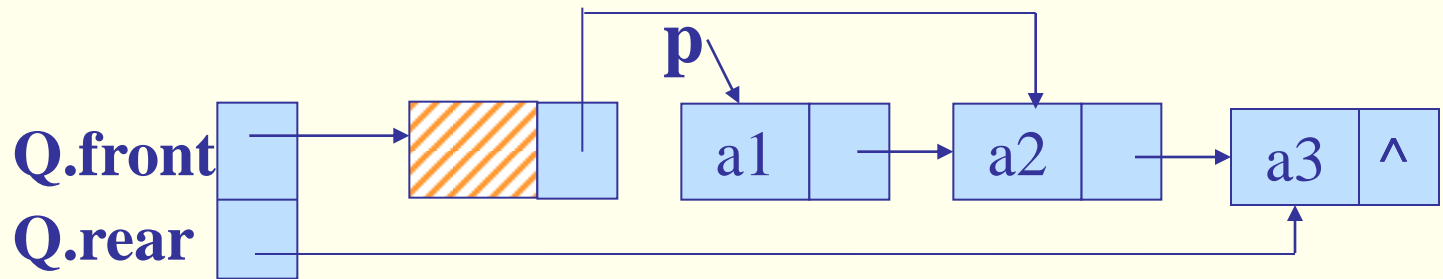
```
Status DestroyQueue(LinkQueue &Q) {  
    while(Q.front){           //从头结点开始往后删结点  
        Q.rear=Q.front->next;  
        free(Q.front);  
        Q.front=Q.rear;  
    }  
    return OK;  
}
```

链队列的入列算法

```
void enqueue (linkqueue Q, elemtype x)
{
    p=(queueptr*)malloc(sizeof(qnode));
    if(!p) exit(OVERFLOW);    //存储分配失败
    p →data=x;
    p →next=NULL;
    Q.rear →next =p;
    Q.rear =p;
}
```

链队列的出列操作

- 当链队列的长度大于1时，出队列操作只要修改头结点的next域内容即可，队尾指针不用变化。



`p = Q.front → next;`

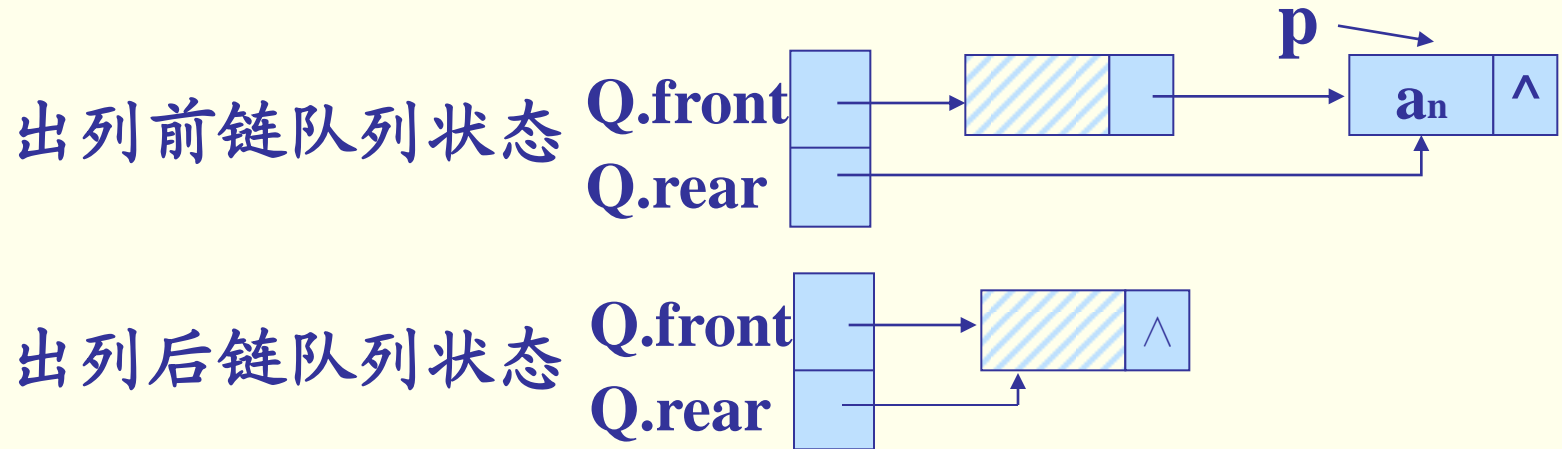
//p指向队头元素

`Q.front → next = p → next;` //修改头结点指针域，指向新的队头元素

`x = p → data;`

`free(p);` //释放被删掉的原队头元素

- 当链队列的长度等于1时，即队列中只有一个数据元素，执行出队列操作后，就成为一个空队列，这时，除修改头结点的next域外，还需修改队尾指针，这是因为此时队尾指针也指向被删结点。



`p = Q.front → next;` //p指向队头元素

`Q.front → next = p → next;` //修改头结点指针域，指向新的队头元素

`x = p → data;`

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

`free(p);` //释放被删掉的原队头元素

链队列的出列算法

```
status delqueue (linkqueue Q)
```

```
{ if (Q.front==Q.rear)
```

```
    return NULL;           //队空
```

```
    p = Q.front → next;    //p指向队头元素
```

```
    Q.front → next = p → next;
```

```
        //修改头结点指针域，指向新的队头元素
```

```
    x = p → data;
```

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

```
    free(p);               //释放被删掉的原队头元素
```

```
    return OK;
```

```
}
```

k阶斐波那契序列

试利用循环队列编写求k阶斐波那契序列中前 $n+1$ 项 ($f_0, f_1, f_2, \dots, f_n$) 的算法, 要求满足 $f_n \leq \max$ 而 $f_{n+1} > \max$, 其中 \max 为某个约定的常数。

(注意本题所用循环队列的容量仅为 k , 则在算法执行结束时, 留在循环队列中的元素应是 k 阶斐波那契序列中的最后 k 项 f_{n-k+1}, \dots, f_n)。

k阶斐波那契序列

$$f_0=0, f_1=0, \dots, f_{k-2}=0, f_{k-1}=1,$$

$$f_n = f_{n-1} + f_{n-2} + \dots + f_{n-k}$$

$$(n=k, k+1, \dots)$$

$$f_i = f_{i-1} + f_{i-2} + \dots + f_{i-k}$$

$$f_{i+1} = f_i + f_{i-1} + f_{i-2} + \dots + f_{i-k+1}$$

$$\text{两式相减得: } f_{i+1} = 2*f_i - f_{i-k}$$

```

void fb(int k,int max)    //方法一,队列的容量为k
{ for(i=0;i<=k-2;i++) {f[i]=0; cq.elem[i]=0;}
  cq.elem[k-1]=1;  cq.rear=k-1;  n=k;
  while(cq.elem[cq.rear]<max)
    {f[n]=0;
      for(j=0;j<k;j++) f[n]=f[n]+ cq.elem[j];
      cq.rear=(cq.rear +1) % k;
      cq.elem[cq.rear]=f[n];    n++;
    }
  if(cq.elem[cq.rear]>max) n=n-2; else n=n-1;
  if (max==1) {n=k;f[k]=1;}
}

```


方法二 利用 $f_{i+1} = 2*f_i - f_{i-k}$, 队列的容量为 $k+1$

Void fb(int k,int max)

```
{ for(i=0;i<=k-2;i++) {f[i]=0; cq.elem[i]=0;}
  cq.elem[k-1]= cq.elem[k]= 1;
  cq.rear=k;    n=k+1;    f[k-1]=f[k]=1;
  while(cq.elem[cq. rear]<max)
    {j= (cq. rear+1) % (k+1);
      f[n]= cq.elem[cq. rear]*2- cq.elem[j];
      cq.elem[j]=f[n];    cq.rear=j;    n++;
    }
  if(cq.elem[cq.rear]>max) n=n-2; else n=n-1;
  if (max==1) {n=k;f[k]=1;}
  if (max==0)  n=k-2;
}
```

队列应用举例

划分子集问题

❖ 问题描述：已知集合

$A = \{a_1, a_2, \dots, a_n\}$ ，及集合上的关系

$R = \{ (a_i, a_j) \mid a_i, a_j \in A, i \neq j \}$ ，其中 (a_i, a_j)

表示 a_i 与 a_j 间存在冲突关系。要求将

A 划分成互不相交的子集

$A_1, A_2, \dots, A_k, (k \leq n)$ ，使任何子集中的

的元素均无冲突关系，同时要求分

子集个数尽可能少

例 $A=\{1,2,3,4,5,6,7,8,9\}$

$R=\{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9),$
 $(5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

可行的子集划分为：

$A1=\{ 1,3,4,8 \}$

$A2=\{ 2,7 \}$

$A3=\{ 5 \}$

$A4=\{ 6,9 \}$

❖ 算法思想：利用循环筛选。从第一个元素开始，凡与第一个元素无冲突的元素划归一组；再将剩下的元素重新找出互不冲突的划归第二组；直到所有元素进组

计算机实现:

❖ 所用数据结构

● 冲突关系矩阵

◆ $r[i][j]=1$, i,j 有冲突

◆ $r[i][j]=0$, i,j 无冲突

● 循环队列cq[n]

● 数组result[n]存放每个元素分组号

● 工作数组newr[n]

工作过程

- **初始状态**：A中元素放于cq中，result和newr数组清零，组号group=1
- **第一个元素出队**，将r矩阵中第一行“1”拷入newr中对应位置，这样，凡与第一个元素有冲突的元素在newr中对应位置处均为“1”，下一个元素出队
 - ◆ 若其在newr中对应位置为“1”，**有冲突**，重新插入cq队尾，参加下一次分组
 - ◆ 若其在newr中对应位置为“0”，**无冲突**，可划归本组；再将r矩阵中该元素对应行中的“1”拷入newr中

●如此反复，直到9个元素依次出队，由newr中为“0”的单元对应的元素构成第1组,将组号group值“1”写入result对应单元中

●令group=2,newr清零，对cq中元素重复上述操作，直到cq中front==rear,即队空，运算结束

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	1	2	3	4	5	6	7	8	9
初始									
	0	1	2	3	4	5	6	7	8
newr	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8
result	0	0	0	0	0	0	0	0	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq		2	3	4	5	6	7	8	9
	\uparrow								\uparrow
	f								r
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8
result	1	0	0	0	0	0	0	0	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	2		3	4	5	6	7	8	9
	\uparrow r	\uparrow f							
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8
result	1	0	0	0	0	0	0	0	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	2			4	5	6	7	8	9
	r		f						
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	0	1	1	0	0
	0	1	2	3	4	5	6	7	8
result	1	0	1	0	0	0	0	0	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	2				5	6	7	8	9
	\uparrow r			\uparrow f					
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	1	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	0	1	1	0	0	0	0	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	2	5				6	7	8	9
		\uparrow r			\uparrow f				
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	1	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	0	1	1	0	0	0	0	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9),$
 $(5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	2	5	6				7	8	9
			\uparrow r			\uparrow f			
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	1	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	0	1	1	0	0	0	0	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	2	5	6	7				8	9
				\uparrow r			\uparrow f		
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	1	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	0	1	1	0	0	0	0	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	2	5	6	7					9
				\uparrow r				\uparrow f	
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	1	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	0	1	1	0	0	0	1	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$



$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	2	5	6	7	9				
				\uparrow r					\uparrow f
	0	1	2	3	4	5	6	7	8
newr	0	1	0	0	1	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	0	1	1	0	0	0	1	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq		5	6	7	9				
									
	f				r				
	0	1	2	3	4	5	6	7	8
newr	1	0	0	0	1	1	0	1	1
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	0	0	0	1	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq			6	7	9	5			
		\uparrow			\uparrow				
		f			r				
	0	1	2	3	4	5	6	7	8
newr	1	0	0	0	1	1	0	1	1
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	0	0	0	1	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq				7	9	5	6		
			\uparrow f				\uparrow r		
	0	1	2	3	4	5	6	7	8
newr	1	0	0	0	1	1	0	1	1
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	0	0	0	1	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9),$
 $(5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq					9	5	6		
				f			r		
	0	1	2	3	4	5	6	7	8
newr	1	0	1	0	1	1	0	1	1
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	0	0	2	1	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq						5	6	9	
					f			r	
	0	1	2	3	4	5	6	7	8
newr	1	0	1	0	1	1	0	1	1
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	0	0	2	1	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq							6	9	
						f		r	
	0	1	2	3	4	5	6	7	8
newr	0	1	0	1	0	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	3	0	2	1	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq								9	6
							f		r
	0	1	2	3	4	5	6	7	8
newr	0	1	0	1	0	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	3	0	2	1	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), \\ (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	9								6
	\uparrow							\uparrow	
	r							f	
	0	1	2	3	4	5	6	7	8
newr	0	1	0	1	0	1	1	0	1
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	3	0	2	1	0

- 算法描述

$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8
cq	9								
	\uparrow r							\uparrow f	
	0	1	2	3	4	5	6	7	8
newr	0	1	1	0	1	0	1	0	0
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	3	4	2	1	0

- 算法描述

$$R = \{ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9), (5,6), (5,4), (7,5), (7,6), (3,7), (6,3) \}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

可行的子集划分为：

$$A1 = \{ 1, 3, 4, 8 \}$$

$$A2 = \{ 2, 7 \}$$

$$A3 = \{ 5 \}$$

$$A4 = \{ 6, 9 \}$$

	0	1	2	3	4	5	6	7	8
cq									
	0	1	2	3	4	5	6	7	8
newr	0	1	1	1	1	0	1	0	0
	0	1	2	3	4	5	6	7	8
result	1	2	1	1	3	4	2	1	4

小 结

栈和队列是两种常见的数据结构，它们都是运算受限的线性表。栈的插入和删除均是在栈顶进行，它是“**后进先出**”的线性表；队列的插入在队尾，删除在队头，它是“**先进先出**”的线性表。在具有后进先出（或先进先出）特性的实际问题中，我们可以使用栈（或队列）这种数据结构来求解。

和线性表类似，依照存储表示的不同，栈有顺序栈和链栈之分，队列有顺序队列和链队列两种，而实际中使用的队列是循环队列。

栈和队列的“**上溢**”和“**下溢**”概念及判别条件应**重点领会**，正确判别栈或队列的空间满而产生的溢出，正确使用栈空或队列空来控制返回。

本章学习要点

1. 掌握栈和队列类型的特点，并能在相应的应用问题中正确选用它们。
2. 熟练掌握栈类型的两种实现方法，特别注意栈满和栈空的条件以及它们的描述方法。
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描述方法。
4. 了解栈和队列的应用，理解递归算法执行过程中栈状态的变化过程。

作业： 3.19

3.19 假设一个算术表达式中可以包含三种括号：圆括号“(”和“)”、方括号“[”和“]”、花括号“{”和“}”，且这三种括号可按任意的次序嵌套使用

(如：... [... { ... } ... [...] ...] ... [...] ... (...) ...)。编写判别给定表达式中所含括号是否正确配对出现的算法（已知表达式已存入数据元素为字符的顺序表中）。