

# 数据结构与算法

*Data Structure and Algorithms*

西安交通大学自动化系

蔡忠闽 周亚东

# 第三章 栈和队列

---

- 3. 1、栈
- 3. 2、栈的应用举例
- 3. 3、队列

## 3.2 栈的应用——求从起点到终点的简单路径

### 回溯

- 对一个包含有许多结点，且每个结点有多个分支的问题，可以选择一个分支进行搜索。当搜索到某一结点，发现无法再继续搜索下去时，可以沿搜索路径回退到前一结点，沿另一分支继续搜索。
- 如果回退之后没有其他选择，再沿搜索路径回退到更前结点，...。依次执行，直到搜索到问题的解，或搜索完全部可搜索的分支没有解存在为止。
- 回溯法与分治法本质相同，可用递归求解。

## 递归

一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，成为递归函数。递归是程序设计中的强有力工具。

- 1) 很多函数是递归定义的，如阶乘函数；
- 2) 有的数据结构本身具有递归特性，如二叉树，其操作可以采用递归描述；
- 3) 有些问题采用递归求解更为简单，如八皇后问题。

## 自顶向下、逐步分解的策略

- 子问题应与原问题做同样的事情，且更为简单；
- 解决递归问题的策略是把一个规模比较大的问题分解为一个或若干规模比较小的问题，分别对这些比较小的问题求解，再综合它们的结果，从而得到原问题的解。

### —分而治之策略（分治法）

- 这些比较小的问题的求解方法与原来问题的求解方法一样。

# 递归

- 1) 把父问题分解成子问题;
- 2) 有一个可解的子问题。

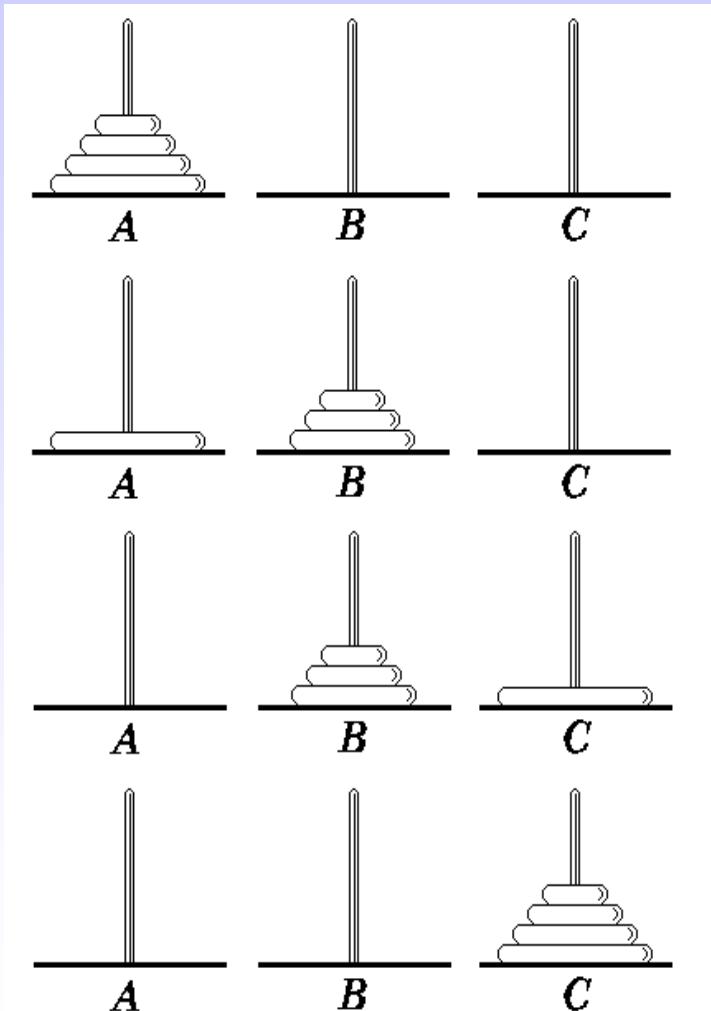
# 递归

- 1) 把父问题分解成子问题;
- 2) 有一个可解的子问题。

## 数学归纳法

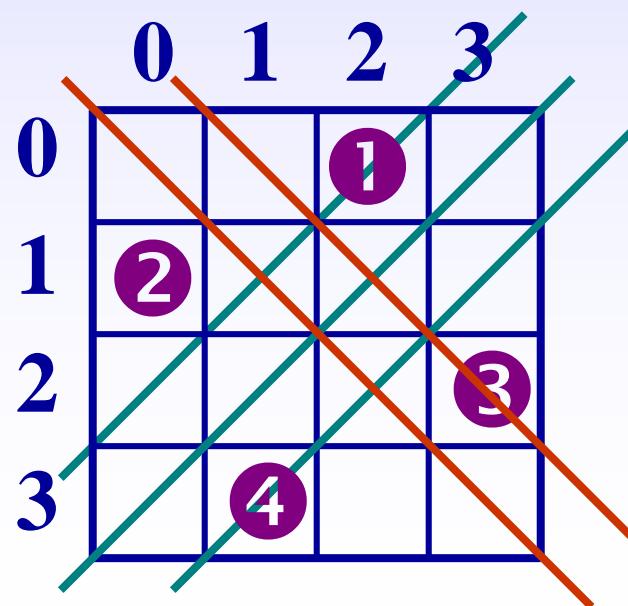
## 3.2 栈的应用—hanoi塔问题

```
void Hanoi (int n, char A, char B, char C)
{
    //解决汉诺塔问题的算法
    if (n == 1)
        Move(A, 1, C)
    else {
        Hanoi(n-1, A, C, B);
        Move(A, n, C)
        Hanoi(n-1, B, A, C);
    }
}
```



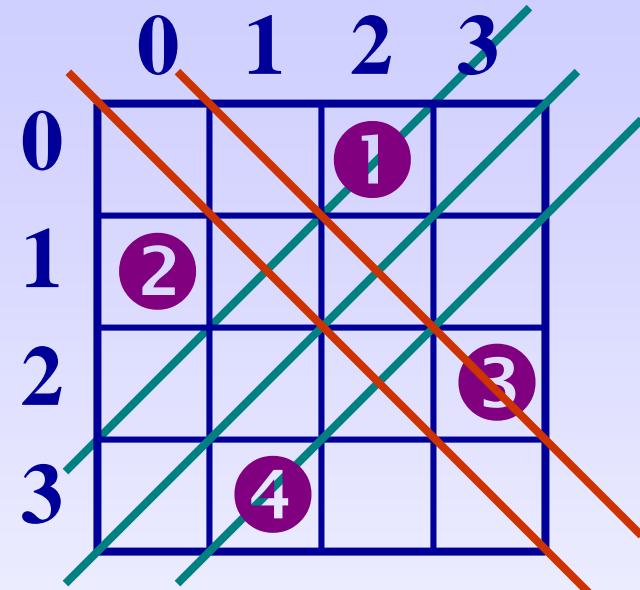
## 3.2 栈的应用--n皇后问题

- 在  $n$  行  $n$  列的国际象棋棋盘上，若两个皇后位于同一行、同一列、同一对角线上，则称为它们为互相攻击。 $n$  皇后问题是指找到这  $n$  个皇后的互不攻击的布局。

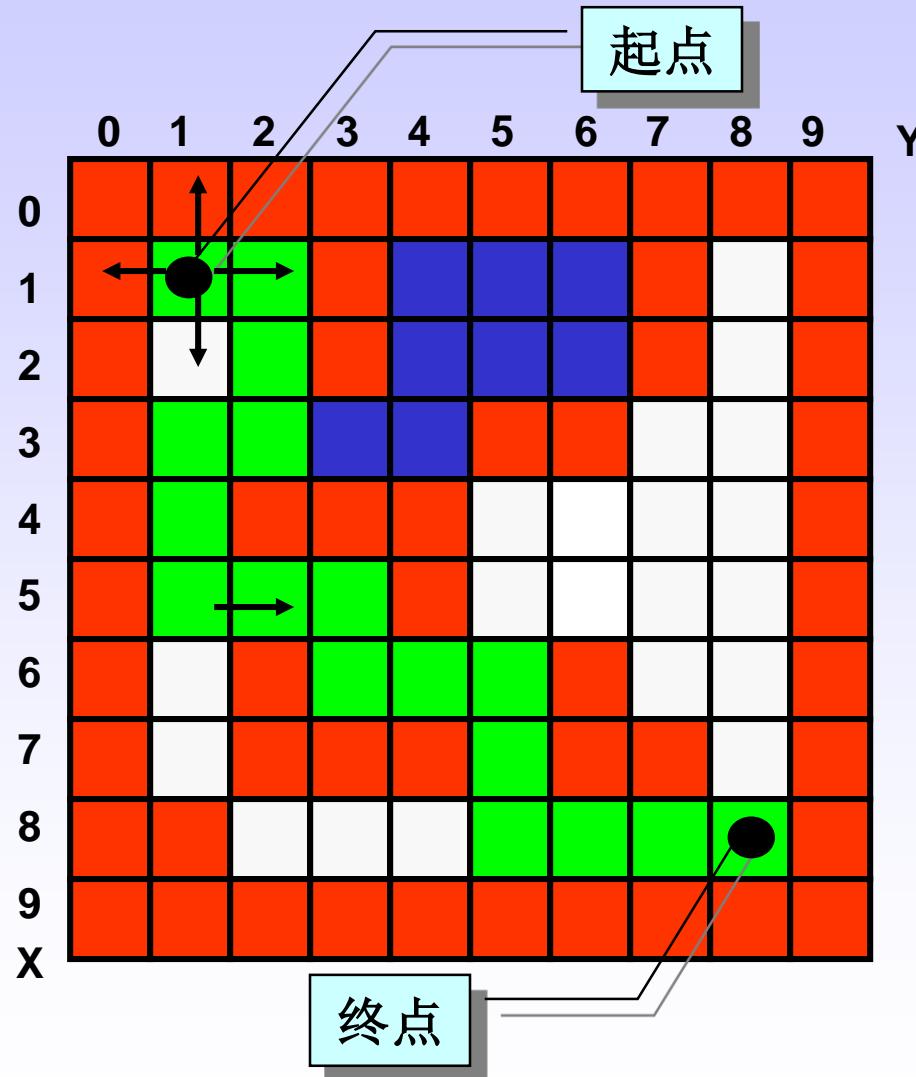


## 3.2 栈的应用——n皇后问题

```
• Boolean NQueen(int i, int * qipan, int n)
• {
•     for (j=1; j<=n;j++)
•     {
•         if Right_position(i, j, qipan)
•         {
•             setposition (i,j,qian)
•             if (i==n) || Nqueen(i+1,qipan,n)
•                 return true;
•             removeposition(i,j,qipan)
•         }
•         Return false
•     }
• }
```



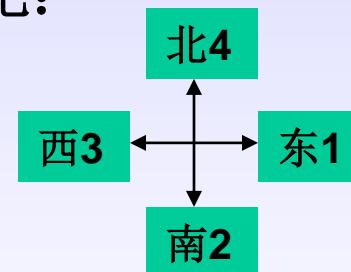
## 3.2 栈的应用--求从起点到终点的简单路径



- 方格的四种类型：

- 非墙且未经试探的方格
- 墙
- 已在路径上的方格
- 已试探过的无发展前途的方格

- 方向标记：



- 起点：(x=1,y=1)；

- 东(1,2) 南(2,1) 西(1,0) 北(0,1)

演示

## 3.2 栈的应用--求从起点到终点的简单路径

- **试探方法:** 穷举求解, 试探每一个可能的方向。

- **可能的方向:**

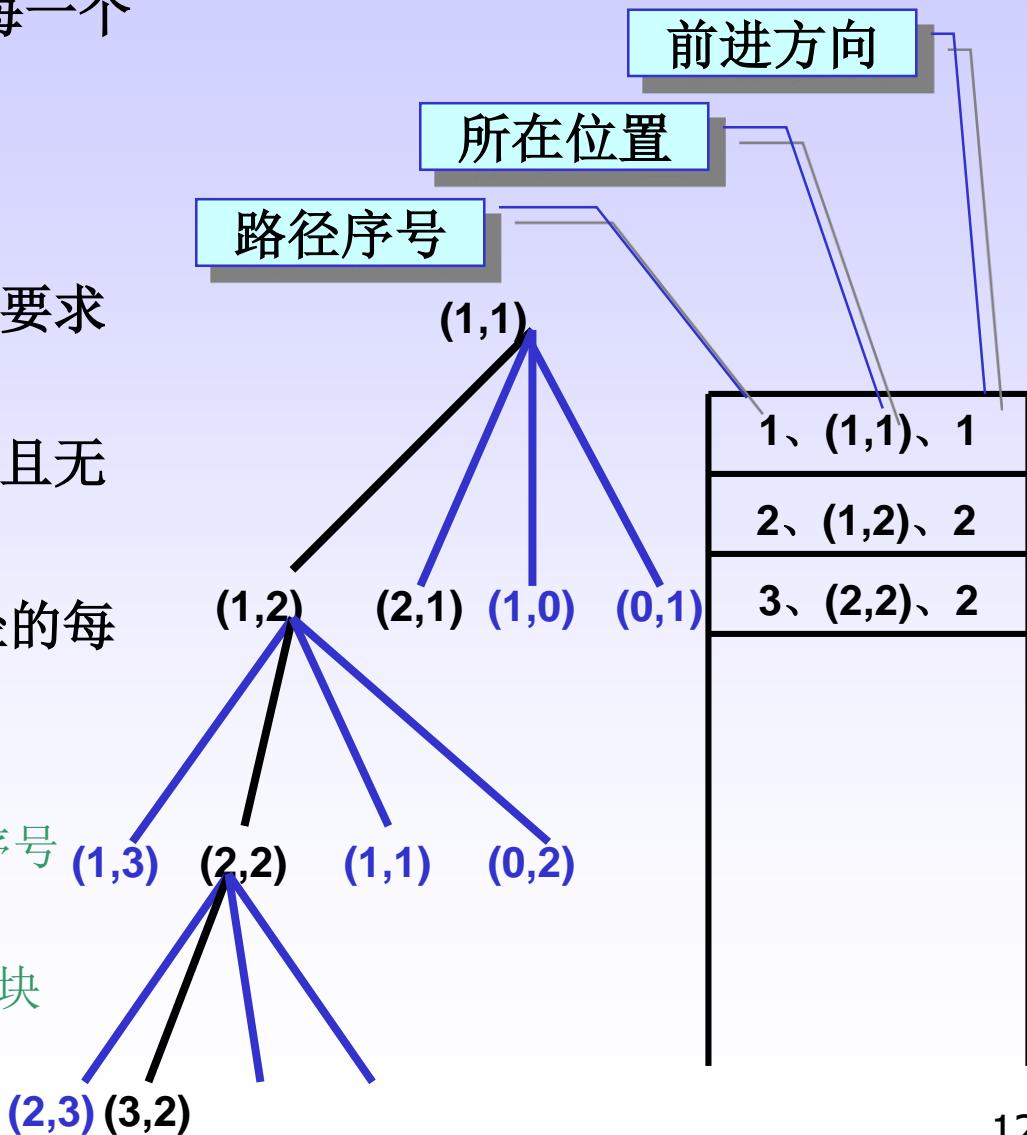
- 非墙新方格

- 不是在路径上的方格 (因为要求从起点到终点的简单路径)

- 不是曾经纳入过路径的方格且无发展前途的

- **堆栈中记录的数据** (组成路径的每个点) 类型:

```
typedef struct
{
    int      step; 块在路径的序号
    PosType seat; 块的坐标位置
    int      di; 从此块走向下一块
                  的方向
} ElemtType;
```



```
Status MazePath ( MazeType maze, PosType start, PosType end )  
{ Initstack(S); curpos = start; curstep = 1;  
  do{ if ( Pass(curpos) )           // 判断当前位置能否通过  
    { FootPrint(curpos);          // 标记已访问过的位置  
      e = ( curstep,curpos,1 ); Push( S, e ); // 加入路径  
      if ( curpos == end ) return ( TRUE ); // 已到达终点  
      curpos = NextPos( curpos, 1); // 得到东邻位置, 注意1代表东。  
      curstep ++;  
    } // if  
  else { // 当前位置不能通过  
    if ( !StackEmpty( S ) ) {  
      Pop( S,e );               // 退回到上次来的位置  
      while ( e.di == 4 && !StackEmpty(S) ) // 当前位置不在路径上  
        { MarkPrint(maze, e.seat); Pop(S,e); } // 设置非路径标志, 后退一步  
      if ( e.di < 4 ) { e.di++; Push(S,e); curpos = NextPos(e.seat, e.di); }  
    } // 在下一个方向继续探索  
  } while ( !StackEmpty(S));  
  return ( FALSE ); // 不存在从起点到终点的路径  
} // MazePath
```

## 3.2 栈的应用—表达式求值

基本思想：

表达式有操作数、运算符和界限符组成，称之为单词。

计算要符合四则运算法则：

- 1) 计算从左到右
- 2) 先乘除，后加减
- 3) 先算括号内，后算括号外

运算符和界限符统称为算法，其集合为**OP**，故算符 $q_1$ 与 $q_2$ 满足以下关系（表3.1）：

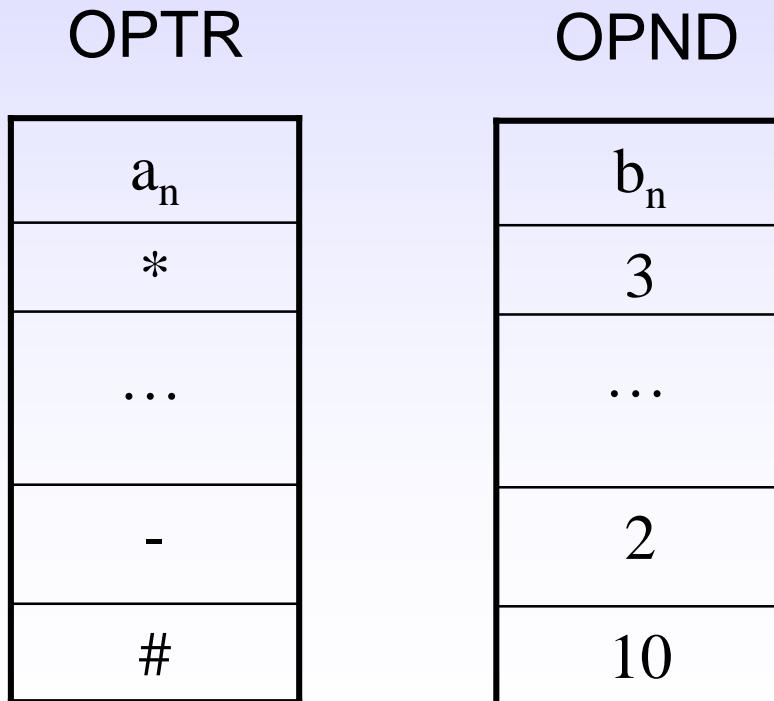
$q_1 < q_2$        $q_1$  的优先权低于  $q_2$

$q_1 = q_2$        $q_1$  的优先权等于  $q_2$

$q_1 > q_2$        $q_1$  的优先权高于  $q_2$

## 3.2 栈的应用—表达式求值

- 1) 置操作数栈OPND为空栈， 运算符栈的栈底元素为#;
- 2) 依次读入字符， 操作数进OPND栈， 运算符则与OPTR栈的栈顶元素比较优先权后进行相应操作， 直至结束。



## OperandType EvaluateExpression()

```
// OPTR和OPND分别是运算符栈和运算数栈， OP是运算符集合；  
InitStack (OPTR); Push (OPTR, '#');  
InitStack (OPND); c = getchar ();  
while (c != '#' || GetTop (OPTR) != '#') {  
    if ( ! In(c, OP)) {Push ((OPND, c); c = getchar ();}  
    else  
        switch (Precede(GetTop(OPTR), c)) {  
            case '<': Push (OPTR, c); c = getchar (); break; //栈顶元素优先权低  
            case '=': Pop(OPTR, x); c = getchar (); break;  
            case '>': Pop(OPTR, theta); Pop(OPND, b); //栈顶元素优先权高  
                        Pop(OPND, a); Push(OPND, Operater(a, theta, b));  
                        break;  
        }  
    return GetTop(OPND);  
}
```

# 第三章 栈和队列

---

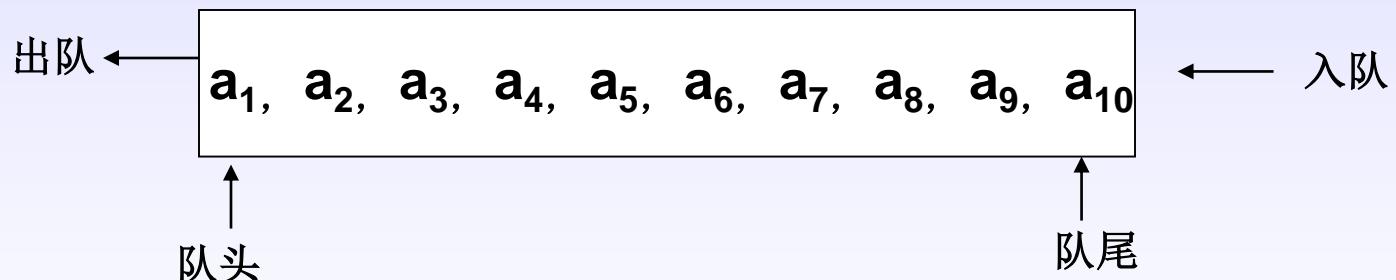
- 3. 1、栈
- 3. 2、栈的应用举例
- 3. 3、队列

## 3.4 队列

### 3.4.1 定义：

队列：在表的一端进行插入，而在另一端进行删除的线性表。

特点：先进先出（First In First Out, FIFO）。



## 3.1 队列 (Queue)

---

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

约定  $a_n$  端为对列尾,  $a_1$  端为对列头

基本操作:

**InitQueue (&Q)** // 构造一个空对列

**DestroyQueue (& Q)** // 销毁对列

**ClearQueue (& Q)** // 将 S 清为空对列

**QueueEmpty(Q)** // 判断是否为空对列, 是则返回 True

**QueueLength(Q)** // 返回对列的长度

**GetHead (Q, &e)** // 返回队头元素

**EnQueue (& Q, e)** // 插入元素 e 为新的队尾元素

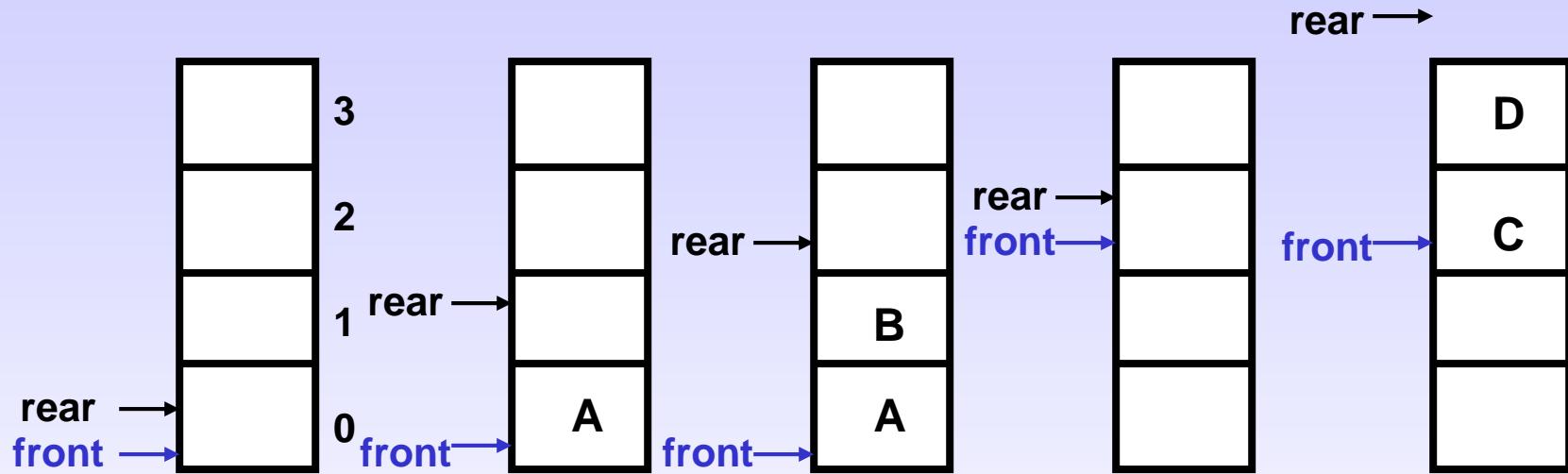
**DeQueue (& Q, &e)** // 删除队头元素, 并用 e 返回

**QueueTraverse(Q, visit())** // 对每个元素都调用 visit 函数, 如调用失败, 则操作失效

}

## 3.4 队列

### 3.4.2 顺序表示的队列：



```
typedef struct {
    QElemType *base;
    int front;
    int rear;
} SqQueue;
```

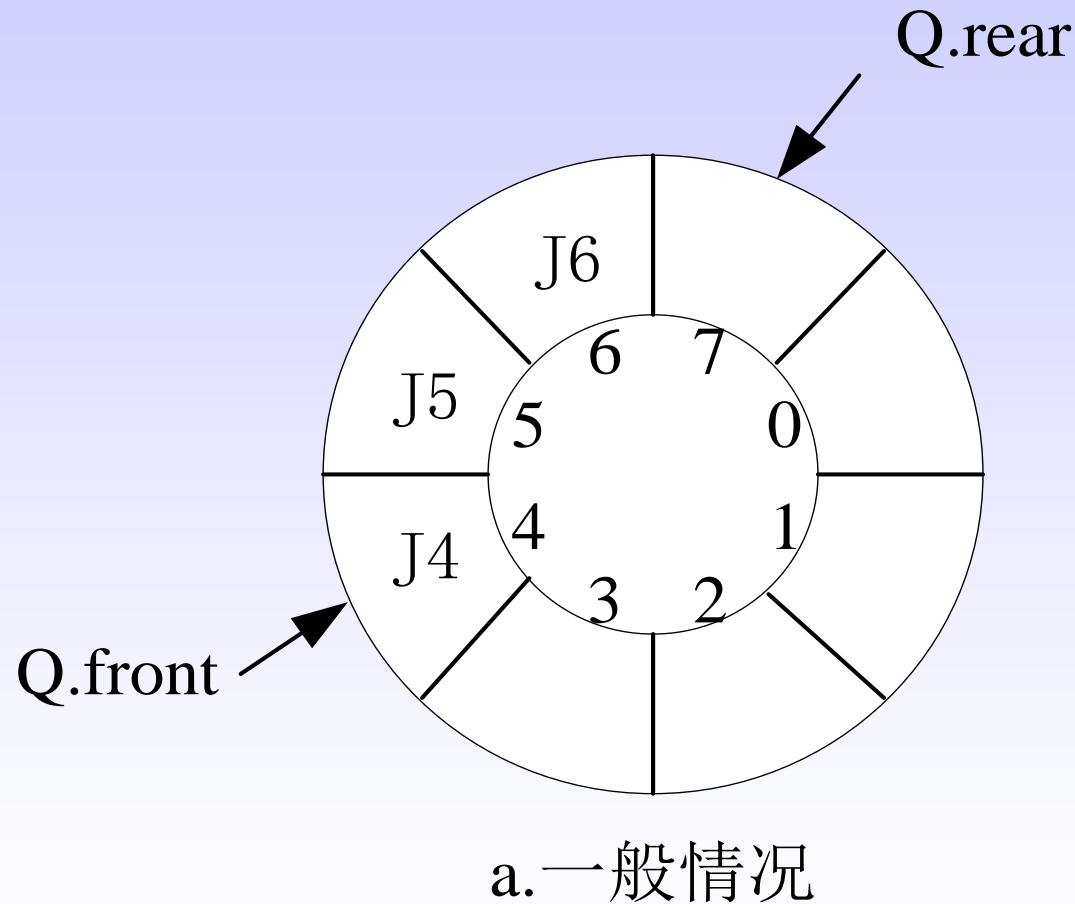
入队: **base[rear++]=x;**

出队: **e=base[front++];**

队空标志: **front == rear**

队满标志: **rear>=MAXQSIZE (假溢出)**

## 3.4 队列



## 3.4 队列

---

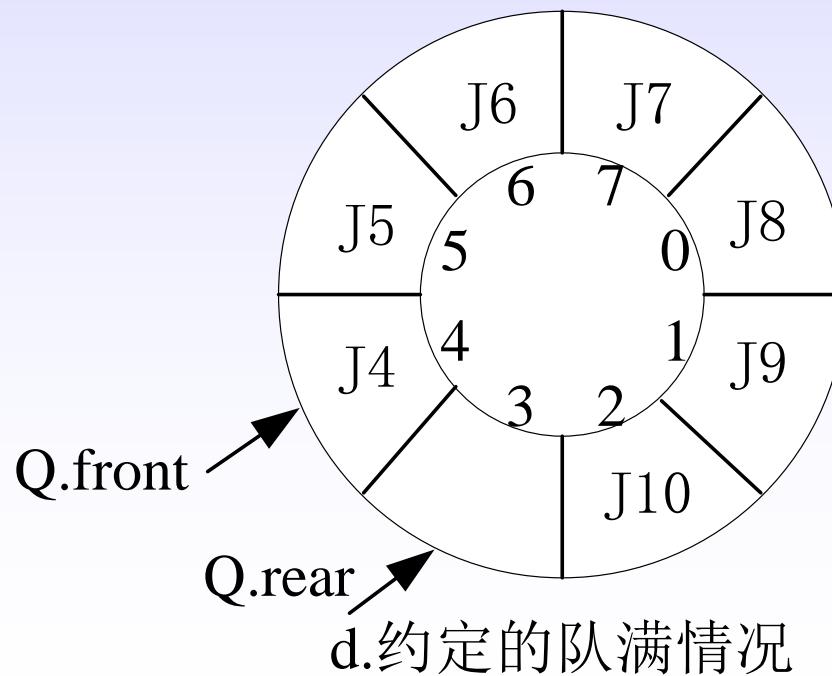
- 为充分利用空间，克服上述假溢出现象的方法是将空间想象为一个首尾相接的圆环（图3.13），并称这种队列为**循环队列**。
- 在循环队列中进行出队、入队操作时，头尾指针仍要加1，朝前移动。只不过当头尾指针指向向量上界（MAXQSIZE-1）时，其加1操作的结果是指向向量的下界0。这种循环意义下的加1操作可以描述为：

```
if (Q. rear+1 == MAXQSIZE)           Q. rear=0;  
else      Q. rear++;
```

利用模运算可简化为： **Q. rear = (Q. rear+1) % MAXQSIZE**

## 3.4 队列

解决此问题的方法是少用一个元素的空间，约定入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等则认为队满（图d，注意：Q. rear所指的单元始终为空）。



## 3.4 队列的状态判断

---

队满：

条件  $((Q.\text{rear} + 1) \% \text{MAXQSIZE}) == Q.\text{front}$

注意  $Q.\text{rear}$  是否会由最高下标跳至最低下标(循环)

队空：

条件  $Q.\text{rear} == Q.\text{front}$

注意  $Q.\text{front}$  是否会由最高下标跳至最低下标(循环)

## 3.4 队列

---

基本操作的实现程序：

```
Status InitQueue (SqQueue &Q )  
{ Q.base = (QElemType *)  
        malloc(MAXQSIZE * sizeof(QElemType));  
        // 分配队列的存储空间;  
    if ( !Q.base ) exit( OVERFLOW );  
    Q.front = Q.rear = 0; // 队头、尾指针清0  
    return OK;  
}
```

## 3.4 队列

入队时应先判队是否满：

条件  $((Q.\text{rear} + 1) \% \text{MAXQSIZE}) == Q.\text{front}$

注意  $Q.\text{rear}$  是否会由最高下标跳至最低下标(循环)

Status EnQueue (SqQueue &Q, QELEMType e )

{ if ((( Q.rear + 1) % MAXQSIZE ) == Q.front )

    return( ERROR ) ; // 思考：此处能否采用realloc函数

    Q.base [ Q.rear ] = e; // 扩大队列的存储空间?

    Q.rear = (( Q.rear + 1) % MAXQSIZE );

    return OK;

} // EnQueue ;

## 3.4 队列

---

出队时应先判断队是否空：

条件  $Q.\text{rear} == Q.\text{front}$

注意  $Q.\text{front}$  是否会由最高下标跳至最低下标(循环)

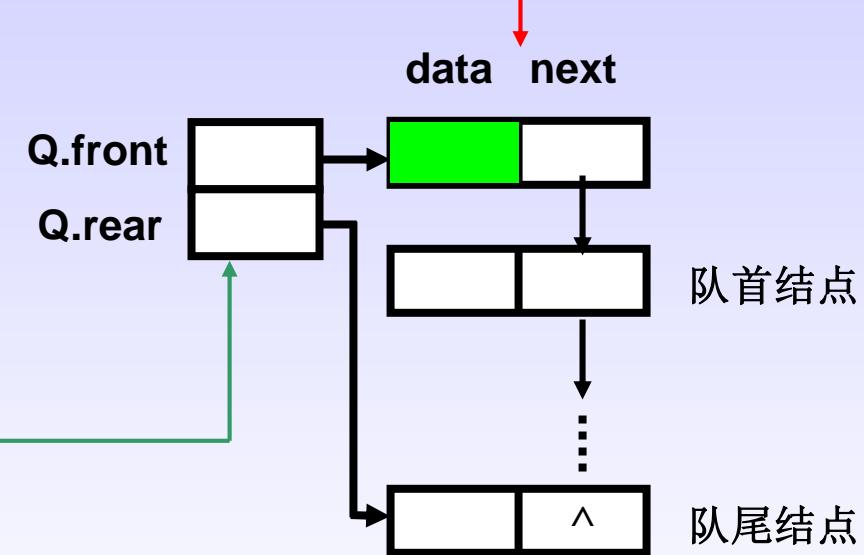
```
Status DeQueue (SqQueue &Q, QElemType &e )  
{ if ( Q.rear == Q.front)  
    return( ERROR ) ;  
    e = Q.base [ Q.front ] ;  
    Q.front = ( Q.front + 1) % MAXQSIZE ;  
    return OK;  
} // DeQueue ;
```

## 3.4 队列

### 3.4.3 链式表示的队列：

```
typedef struct Qnode {
    QElemType      data;
    struct Qnode *next;
} Qnode;
```

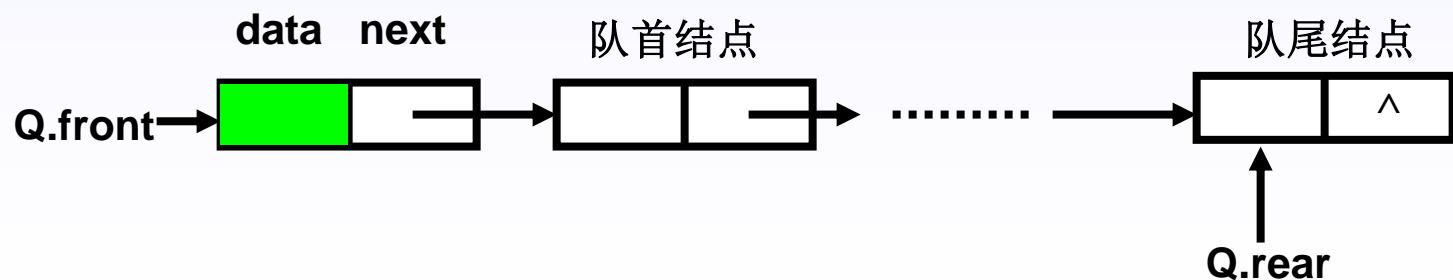
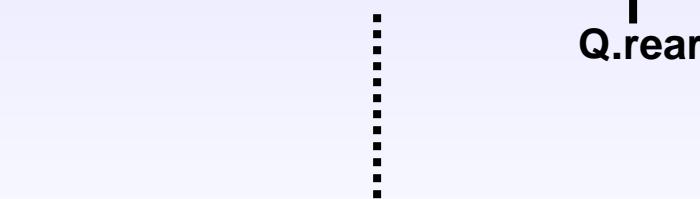
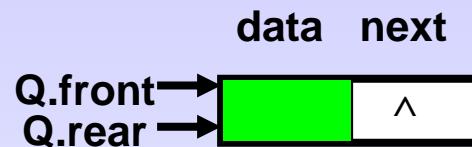
```
typedef struct {
    Qnode *front;
    Qnode *rear;
} LinkQueue;
```



`Q.front` 和 `Q.rear` 分别是队首和队尾指针。它们指示着队首的前一结点和队尾结点。

## 3.4 队列

链接队列的操作：



## 3.4 队列

---

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

## 3.4 队列

---

```
Status DeQueue (LinkQueue &Q, QElemType &e )  
{ if ( Q.rear == Q.front)  
    return( ERROR ) ;  
p = Q.front->next;  
e = p->data;  
Q.front->next = p->next;  
if( Q.rear == p )      // 出队前队列中只有一个元素时,  
    Q.rear = Q.front ; // 需修改队尾指针, 指向头结点。  
free( p );  
return OK;  
} // DeQueue ;
```

## 3.4 队列

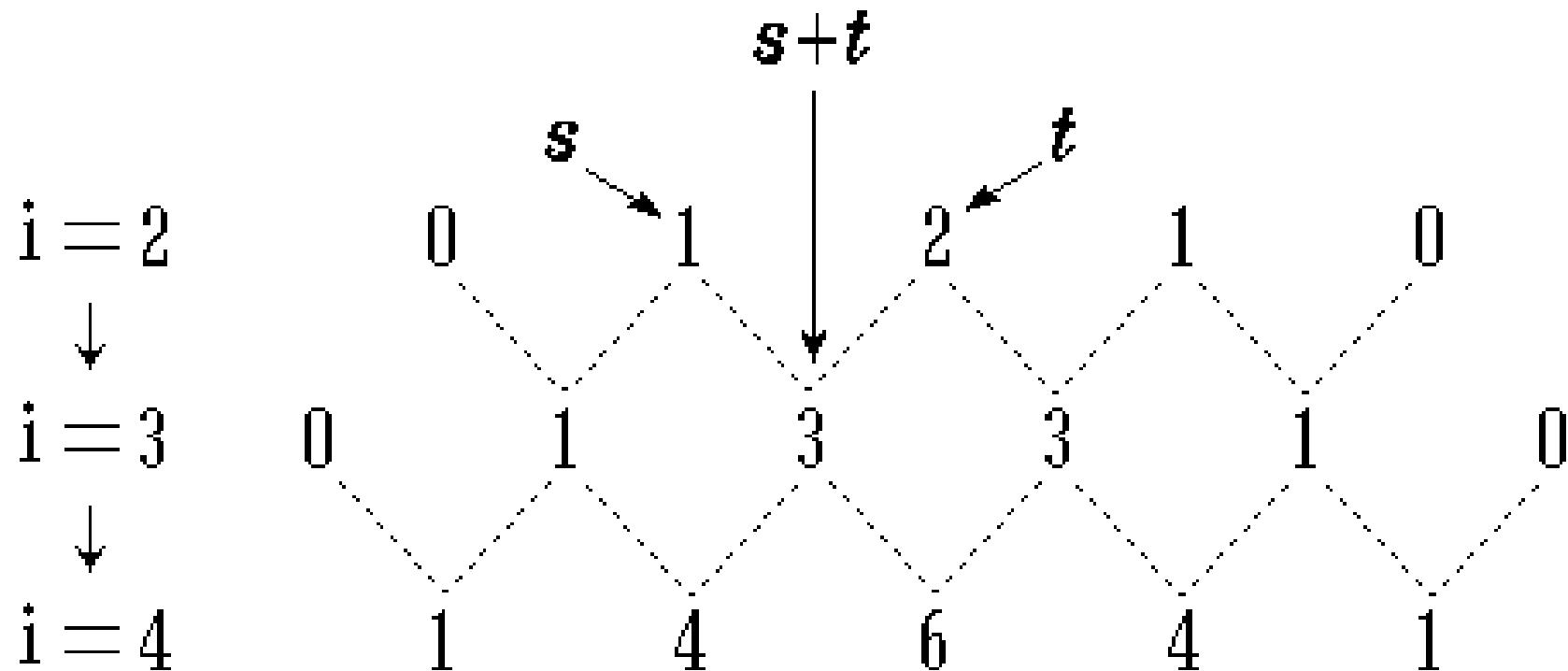
例1 逐行打印二项展开式  $(a + b)^i$  的系数

杨辉三角形 (Pascal's triangle)

|   |   |   |   |    |   |    |   |         |
|---|---|---|---|----|---|----|---|---------|
|   | 1 |   | 1 |    |   |    |   | $i = 1$ |
|   | 1 |   | 2 |    | 1 |    |   | 2       |
|   | 1 |   | 3 |    | 3 |    | 1 | 3       |
|   | 1 |   | 4 |    | 6 |    | 4 | 4       |
| 1 |   | 5 |   | 10 |   | 10 |   | 5       |
| 1 |   | 6 |   | 15 |   | 20 |   | 15      |
|   |   |   |   |    |   |    |   | 6       |

## 3.4 队列

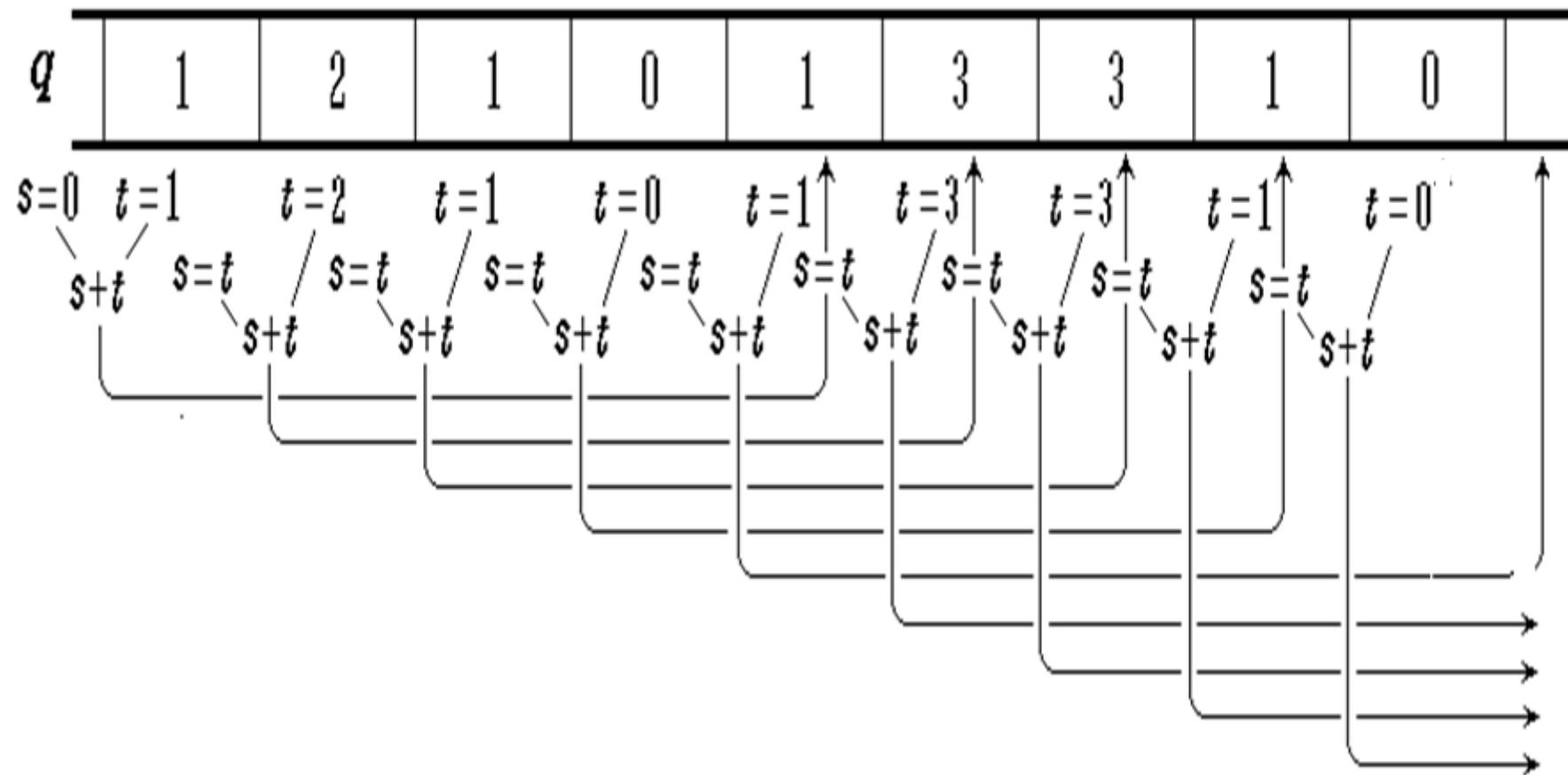
分析第  $i$  行元素与第  $i+1$  行元素的关系



目的是从前一行的数据可以计算下一行的数据

### 3.4 队列

从第  $i$  行数据计算并存放第  $i+1$  行数据



## 3.4 队列

```
void YangHui ( int n ) // n为需要输出的杨辉三角形的行数
{ SqQueue q;    int s=0, t;
  InitQueue(q);  EnQueue (q, 1); EnQueue (q, 1);
  for( int i=1; i<=n; i++ ) // 逐行计算
  { printf("\n");
    EnQueue (q, 0);
    for ( int j=1; j<=i+2; j++ ) // 根据上行系数求下行系数
    { DeQueue (q, t);
      EnQueue (q, s+t);
      s = t;
      if ( j != i+2 ) printf("%3d", s); // 不输出每行结尾的0
    }
  }
}
```

## 3.4 队列

### 例2 图元识别

数字化图像是一个 $m \times m$  的像素矩阵。在单色图像中，每个像素的值要么为0，要么为1，值为0的像素表示图像的背景，而值为1的像素则表示图元上的一个点，称其为图元像素。

如果一个像素在另一个像素的左、上、右或下部，则称这两个像素为相邻像素。识别图元就是对相邻像素进行标记，当且仅当两个像素相邻时，称这两个像素属于同一图元，它们的标号相同。

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 |



|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
|   |   | 2 |   |   |   |   |
|   |   | 2 | 2 |   |   |   |
|   |   |   |   | 3 |   |   |
|   |   |   |   | 3 | 3 |   |
|   |   |   |   |   | 4 | 5 |
| 4 | 4 | 4 |   |   | 3 | 5 |
| 4 | 4 | 4 |   |   | 5 | 5 |

## 3.4 队列

---

### 图元识别算法：

- 1 为了简化运算，在图像周围包上一圈空白像素（即0像素）。
- 2 逐行逐列扫描像素，当遇到一个没有图元编号的图元像素时，就给它指定一个图元编号（使用数字2, 3, ...作为图元编号），该像素就成为一个新图元的种子。
- 3 识别和标记与种子相邻的所有图元像素（1-间距像素）。
- 4 识别和标记与1-间距像素相邻的所有无标记图元像素（2-间距像素）。
- 5 继续识别和标记与2-间距像素相邻的无标记图元像素。这个过程一直持续到再也找不到新的、相邻的无标记图元像素为止。
- 6 检查是否还存在无图元编号的图元像素，如果有，返回到2继续执行，否则算法结束。

## 3.4 队列

---

```
void Label(int pixel[][], int m)
    // pixel存图象, m为图象的高和宽
{
    SqQueue q;
    PosType here, nbr, cpos; // 类型说明见“迷宫问题”
    int i, j, id=1; //图元编号
    InitQueue(q);
    for ( i=0; i<=m+1; i++) { // 初始化“围墙”
        pixel[0][i] = pixel[m+1][i] = 0; // 底和顶
        pixel[i][0] = pixel[i][m+1] = 0; // 左和右
    }
}
```

```
for ( cpos.x=1; cpos.x<=m; cpos.x++) // 图像的第r行
    for (cpos.y=1; cpos.y<=m; cpos.y++) // 图像的第c列
        if (pixel[cpos.x][cpos.y] == 1) // 新图元
        { pixel[cpos.x][cpos.y] = ++id; // 得到下一个id
            here = cpos;
            do{ // 寻找其余图元
                for(j=1; j<=4; j++) // 检查当前像素的所有相邻像素
                { nbr=NextPos(here, j); // NextPos见“迷宫问题”
                    if (pixel[nbr.x][nbr.y] == 1) {
                        pixel[nbr.x][nbr.y] = id;
                        EnQueue(q, nbr);
                    }
                }
            }while(DeQueue(q,here)); // 还有未探索的像素吗?
        } // 结束if和for
    } //end Label
```

## 3.5 栈的进一步讨论

当进栈元素的编号为 $1, 2, \dots, n$ 时，可能的出栈序列有多少种？

设进栈元素数为 $n$ ，可能出栈序列数为 $m_n$ :

✓  $n = 0, m_0 = 1$ : 出栈序列{}。

✓  $n = 1, m_1 = 1$ : 出栈序列{1}。

✓  $n = 2, m_2 = 2: = m_0 * m_1 + m_1 * m_0$

a) 出栈序列中1在第1位。1进 1出 2进 2出，出栈序列为{1, 2}。 $= m_0 * m_1 = 1$

b) 出栈序列中1在第2位。1进 2进 2出 1出，出栈序列为{2, 1}。 $= m_1 * m_0 = 1$

✓  $n = 3, m_3 = 5: = m_0 * m_2 + m_1 * m_1 + m_2 * m_0$

a) 出栈序列中1在第1位。后面2个元素有 $m_2 = 2$ 个出栈序列：{1, 2, 3}, {1, 3, 2}。

## 3.5 栈的进一步讨论

$$= m_0 * m_3 = 5$$

- b) 出栈序列中1在第2位。前面有2，后面3、4有 $m_2 = 2$ 个出栈序列: {2, 1, 3, 4}, {2, 1, 4, 3}。 =  $m_1 * m_2 = 2$
- c) 出栈序列中1在第3位。前面2、3有 $m_2 = 2$ 个出栈序列，后面有4: {3, 2, 1, 4}, {2, 3, 1, 4}。 =  $m_2 * m_1 = 2$
- d) 出栈序列中1在第4位。前面3个元素有 $m_3 = 5$ 个出栈序列: {2, 3, 4, 1}, {2, 4, 3, 1}, {3, 2, 4, 1}, {3, 4, 2, 1}, {4, 3, 2, 1}。  
=  $m_3 * m_0 = 5$

## 3.5 栈的进一步讨论

- ✓ 一般地，设有  $n$  个元素按序号  $1, 2, \dots, n$  进栈，轮流让  $1$  在出栈序列的第  $1, 2, \dots, n$  位，则可能的出栈序列数为：

$$m_n = \sum_{i=0}^{n-1} m_i * m_{n-i} = m_0 * m_{n-1} + m_1 * m_{n-2} + \dots + m_{n-1} * m_0$$

推导结果为：

$$m_n = \sum_{i=0}^{n-1} m_i * m_{n-i} = \frac{1}{n+1} C_{2n}^n$$