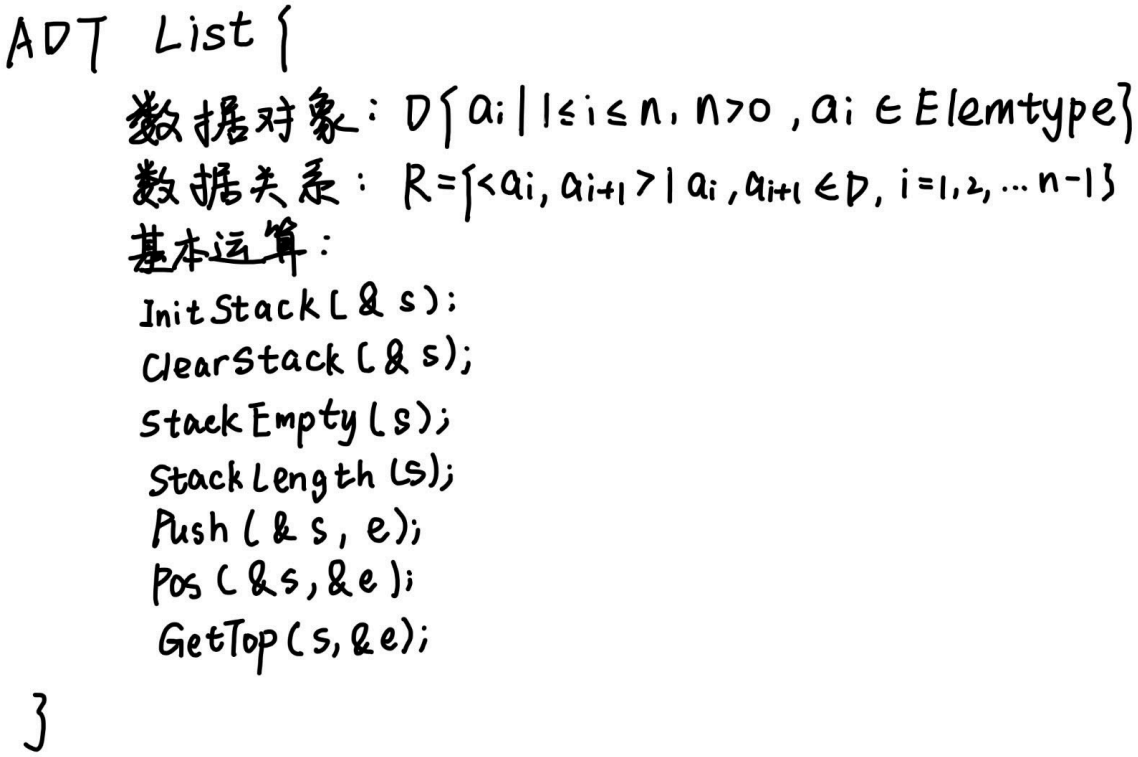
# 栈和队列

## ▼ 栈

### ▼ 抽象数据类型




### ▼ b顺序栈

```
typedef struct{
    int data[MaxSize];
    int top;
}SqStack;
```

基本操作

top指向第一个元素

```
void InitStack(SqStack &S){
    S.top=-1;
}

bool StackEmpty(SqStack &S){
    return (S.top==-1);
}

bool Push(SqStack &S,int x){
    //栈满
    if(S.top==MaxSize-1){
        return false;
    }
    S.data[++S.top]=x;
    return true;
}

bool Pop(SqStack &S,int &x){
    //栈空
    if(StackEmpty(S)){
        return false;
    }
    x=S.data[S.top--];
    return true;
}

bool GetTop(SqStack S,int &x){
    if(S.top==-1) return false;
    x=S.data[S.top];
    return true;
}
```

top初始指向空位置

```
void InitStack_2(SqStack &S){
    S.top=0;
}

bool StackEmpty_2(SqStack &S){
    return (S.top==0);
}

bool Push_2(SqStack &S,int x){
    if(S.top==MaxSize){
        return false;
    }
    S.data[S.top++]=x;
    return true;
}

bool Pop_2(SqStack &S,int &x){
    if(S.top==0) return false;
    x=S.data[--S.top];
    return true;
}

bool GetTop_2(SqStack S,int &x){
    if(S.top==0) return false;
    x=S.data[--S.top];
    return true;
}
```

## 共享栈

栈满条件 $top0 + 1 = top1$

```
typedef struct{
    int data[MaxSize];
    int top0;
```

```
    int top1;
}ShStack;

void InitStack(ShStack &S){
    S.top0=-1;
    S.top1=MaxSize;
}
```

## ▼ 链式栈

```
typedef struct LinkNode{
    int data;
    struct LinkNode *next;
}*LiStack,LiNode;
//typedef struct{
//  Link top;
//  int length;
//}LnkStack;
```

带头结点的

```
bool InitListack(LiStack &L){
    L=(LiNode*)malloc(sizeof(LiStack));
    if(L==NULL) return false;
    L→next=NULL;
    return true;
}

bool Empty(LiStack &L){
    return (L→next==NULL);
}

bool Push(LiStack &L,int x){
    LiNode *s=(LiNode*)malloc(sizeof(LiNode));
    if(s==NULL) return false;
    s→data=x;
```

```
    s->next=L->next;
    L->next=s;
    return true;
}

bool Pop(LiStack &L,int &x){
    if(L->next==NULL) return false;
    x=L->next->data;
    LiNode *p=L->next;
    L->next=L->next->next;
    free(p);
    return true;
}

bool GetTop(LiStack &L,int &x){
    if(L->next==NULL) return false;
    x=L->next->data;
    return true;
}
void ClearListack(LiStack &L){
    if(L->next==NULL) printf("栈为空\n");
    LiNode *s=L->next,*p;
    while(s!=NULL){
        p=s;
        s=s->next;
        free(p);
    }
    L->next=NULL;
}
```

# ▼ 应用

## 函数嵌套调用、递归

### ▼ 数制转换

```
void conversion() {
    // 非负十进制整数转换成八进制数
    InitStack(S); // 构造空栈
    int N;
    scanf("%d", &N); // 输入一个十进制数
    while (N) {
        Push(S, N % 8); // “余数”入栈
        N = N / 8; // 整除，非零“商”继续运算
    }
    while (!StackEmpty(S)) { // 按“求余”所得相逆的顺序输出八进制的各位数
        int e;
        Pop(S, &e);
        printf("%d", e);
    }
}
```

## ▼ 括号匹配

```
bool bracketCheck(char str[],int length){
    SqStack S;
    S.top=-1;
    for(int i=0;i<length;i++){
        if(str[i]=='('||str[i]=='['||str[i]=='{'){
            S.data[++S.top]=str[i];
        }
        else {
            if(!(str[i]==')')&&!(str[i]==']')&&!(str[i]=='}'))
                continue;
            if(S.top==-1) return false;
            char x=S.data[S.top--];
            if(str[i]==')'&& !(x=='(')) return false;
            if(str[i]==']'&& !(x=='[')) return false;
            if(str[i]=='}'&& !(x=='{')) return false;
        }
    }
```

```
    return (S.top==-1);
}
```

## ▼ 迷宫找一条通路

```
bool MazePath(MazeType name, PosType start, PosType end) {
    // 若在迷宫 name 中存在从 start 到 end 的路径，保存在栈中，并返回 TR
    InitStack(S);
    PosType curpos = start;
    int curstep = 1;
    do {
        if (Pass(curpos)) { // 当前位置可行，不是墙、没走过、不是不可行块
            FootPrint(curpos); // 留下足迹，标记为“经过块”
            ElemType e;
            e.seat = curpos;
            e.ord = curstep;
            e.di = 1; // {1,2,3,4} → {east,south,west,north}
            Push(S, e);
            if (curpos == end) {
                return TRUE;
            }
            curpos = NextPos(curpos, 1); // 得到下一个待检测的位置，从东方
            curstep++;
        } else { // 当前位置不可行
            if (!StackEmpty(S)) {
                ElemType e;
                Pop(S, &e);
                while (e.di == 4 && !StackEmpty(S)) {
                    MarkPrint(e.seat); // 标记为不可行点
                    Pop(S, &e);
                }
                if (e.di < 4) {
                    e.di++;
                    Push(S, &e);
                    curpos = NextPos(e.seat, e.di); // 下一个待检测的位置
                }
```

```
        }
      }
    } while (!StackEmpty(S)); // 当栈不空时继续循环
    return FALSE; // 如果栈空了，说明没有路径
}
```

## ▼ 表达式求值

例如：若Exp = a×b+(c−d／e)×f

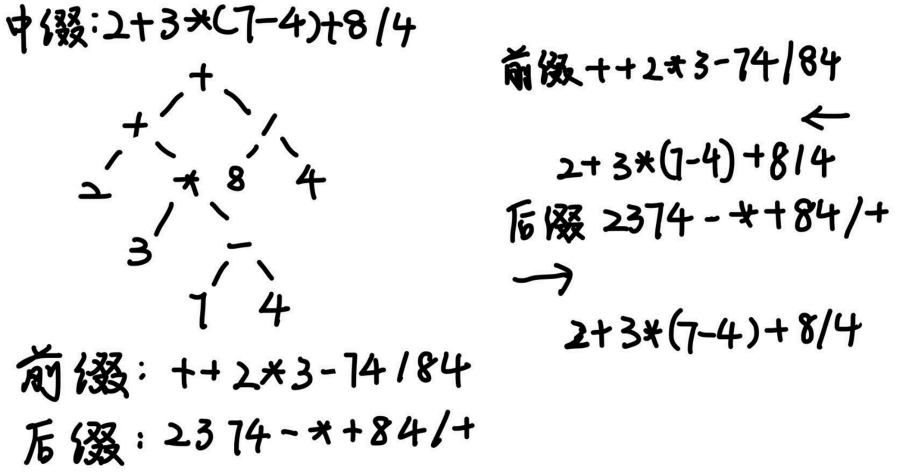
- 前缀式为：+×a b ×−c／d e f
- 中缀式为：a×b +c−d／e×f
- 后缀式为：a b×c d e／−f ×+

**手算**

**前缀表达式**：从右往左扫，计算运算符和其之前的两个操作数，先弹出的是右操作数

**后缀表达式**：从左往右扫，计算运算符和其之前的两个操作数，先弹出的是左操作数

转换手算：中缀画树，前缀从后往前，后缀从前往后



**机算**

▼ **中缀转前缀（右优先）**

1. 遇到操作数直接输出
2. 遇到界限符 ‘ ) ’ 直接入栈，遇到 ‘ ( ‘ 弹出栈内运算符直到 ‘ ) ‘
3. 遇到运算符，依次弹出优先级高于当前运算符的所有，若碰到 ' ) ' 或栈空则停止，再将当前运算符入栈
4. 将剩余运算符依次弹出
5. 将得到的字符串逆置

```
int optim(char op){
    if(op=='+'||op=='-') return 1;
    if(op=='*'||op=='/') return 2;
    return 0;
}
string InToPre(string str){
    SqStack1 S;
    InitStack1(S);
    string num="";
    int len=str.length();
    string poststr="";
    for(int i=len-1;i>=0;i--){
        //提取数字压入栈
        if(str[i]>='0'&&str[i]<='9'){
            num+=str[i];
        }
        //前面的数读入完毕
        else if(str[i+1]>='0' && str[i+1]<='9'){
            poststr+=num+" ";
            num="";
        }
        if(str[i]==')'){
            Push1(S,')');
        }
```

```
        if(str[i]=='('){
            while(S.data[S.top]!=')'){
                poststr+=Pop1(S);
                poststr+=" ";
            }
            Pop1(S);
        }
        //字母直接输出
        if((str[i]>='a'&&str[i]<='z')||(str[i]>='A'&&str[i]<='Z')){
            poststr+=str[i];
            poststr+=" ";
        }
        if(str[i]=='+'||str[i]=='-'||str[i]=='*'||str[i]=='/'){
            while(optim(S.data[S.top])>optim(str[i]) && S.top!=-1){
                poststr+=Pop1(S);
                poststr+=" ";
            }
            Push1(S,str[i]);
        }
    }
    if(num!="") poststr+=num+" ";
    while(S.top!=-1){
        poststr+=Pop1(S);
        poststr+=" ";
    }
    poststr.erase(poststr.length() - 1);
    reverse(poststr.begin(),poststr.end());
    return poststr;
}
//2+3*(7-4)+8/4
//+ + 2 * 3 - 7 4 / 8 4
//(a+b)*c-d/c
//- * + a b c / d c
```

▼ **中缀转后缀（左优先）**

1. 遇到操作数直接输出
2. 遇到界限符 ‘ ( ‘ 压入栈，遇到 ‘ ) ‘ 弹出栈内元素直到 ‘ ( ‘
3. 遇到运算符，弹出优先级高于或**等于**当前运算符的所有，碰到 ‘ ( ‘或栈空则停，再将当前运算符压入栈
4. 将剩余运算符依次弹出

```
string InToPost(string str){
    SqStack1 S;
    InitStack1(S);
    string num="";
    int len=str.length();
    string poststr="";
    for(int i=0;i<len;i++){
        //提取数字压入栈
        if(str[i]>='0'&&str[i]<='9'){
            num+=str[i];
        }
        //前面的数读入完毕
        else if(str[i-1]>='0' && str[i-1]<='9'){
            poststr+=num+" ";
            num="";
        }
        if(str[i]=='('){
            Push1(S,'(');
        }
        if(str[i]==')'){
            while(S.data[S.top]!='('){
                poststr+=Pop1(S);
                poststr+=" ";
            }
            Pop1(S);
        }
        //字母直接入栈
        if((str[i]>='a'&&str[i]<='z')||(str[i]>='A'&&str[i]<='Z')){
```

```
            poststr+=str[i];
            poststr+=" ";
        }
        if(str[i]=='+'||str[i]=='-'||str[i]=='*'||str[i]=='/'){
            while(optim(S.data[S.top])>=optim(str[i]) && S.top!=-1){
                poststr+=Pop1(S);
                poststr+=" ";
            }
            Push1(S,str[i]);
        }
    }
    if(num!="") poststr+=num+" ";
    while(S.top!=-1){
        poststr+=Pop1(S);
        poststr+=" ";
    }
    poststr.erase(poststr.length() - 1);
    return poststr;
}
//((15/(7-(1+1)))*3)-(2+(1+1))
//a+b-a*((c+d)/e-f)+g
//a b + a c d + e / f - * - g +
//1+((1+3)/(2+0)*2+2-1)
//1 1 3 + 2 0 + / 2 * 2 + 1 - +
```

▼ **后缀表达式机算**

```
int calc(char op,int x,int y){
    if(op=='+'){
        return (y+x);
    }
    else if(op=='-'){
        return (y-x);
    }
    else if(op=='*'){
```

```
        return (y*x);
    }
    else if(op=='/'){
        return (y/x);
    }
}

int Postfix(string str){
    SqStack Snum;
    InitStack(Snum);
    int len=str.length(),tmp=0;
    for(int i=0;i<len;i++){
        //提取数字压入栈
        if(str[i]>='0'&&str[i]<='9'){
            tmp=tmp*10+str[i]-'0';
        }
        //遇到' '标志着前面的数读入完毕
        else if(str[i]==' '&& str[i-1]>='0' && str[i-1]<='9'){
            Push(Snum,tmp);
            tmp=0;
        }
        else if(str[i]!=' '){
            //从栈中取两个元素进行运算,运算后压入栈
            int x=Pop(Snum);
            int y=Pop(Snum);
            Push(Snum,calc(str[i],x,y));
        }
    }
    if(Snum.top==0) return Snum.data[0];
    return -1;
}
```

**▼ 中缀表达式机算**

1. 操作数压入栈

2. 运算符或界限符，按中缀转后缀压入栈（期间弹出运算符时弹出两个操作数进行运算）

```
int Infix(string str){
    SqStack1 Sop;
    InitStack1(Sop);
    SqStack Snum;
    InitStack(Snum);
    int len=str.length(),tmp=0;
    char op='#';
    int n1,n2;
    for(int i=0;i<len;i++){
        if(str[i]<='9'&&str[i]>='0'){
            tmp=tmp*10+(str[i]-'0');
        }
        else if(str[i-1]<='9'&&str[i-1]>='0'){
            Push(Snum,tmp);
            tmp=0;
        }
        if(str[i]=='('){
            Push1(Sop,'(');
        }
        if(str[i]==')'){
            while(Sop.data[Sop.top]!='('){
                op=Pop1(Sop);
                n1=Pop(Snum);
                n2=Pop(Snum);
                Push(Snum,calc(op,n1,n2));
            }
            Pop1(Sop);
        }
        if(str[i]=='+'||str[i]=='-'||str[i]=='*'||str[i]=='/'){
            while(optim(Sop.data[Sop.top])>=optim(str[i]) && Sop.top!=
                op=Pop1(Sop);
                n1=Pop(Snum);
```

```
            n2=Pop(Snum);
            Push(Snum,calc(op,n1,n2));
        }
        Push1(Sop,str[i]);
    }
  }
  if(tmp!=0) Push(Snum,tmp);
  while(Sop.top!=-1){
    op=Pop1(Sop);
    n1=Pop(Snum);
    n2=Pop(Snum);
    Push(Snum,calc(op,n1,n2));
  }
  return Pop(Snum);
}
```

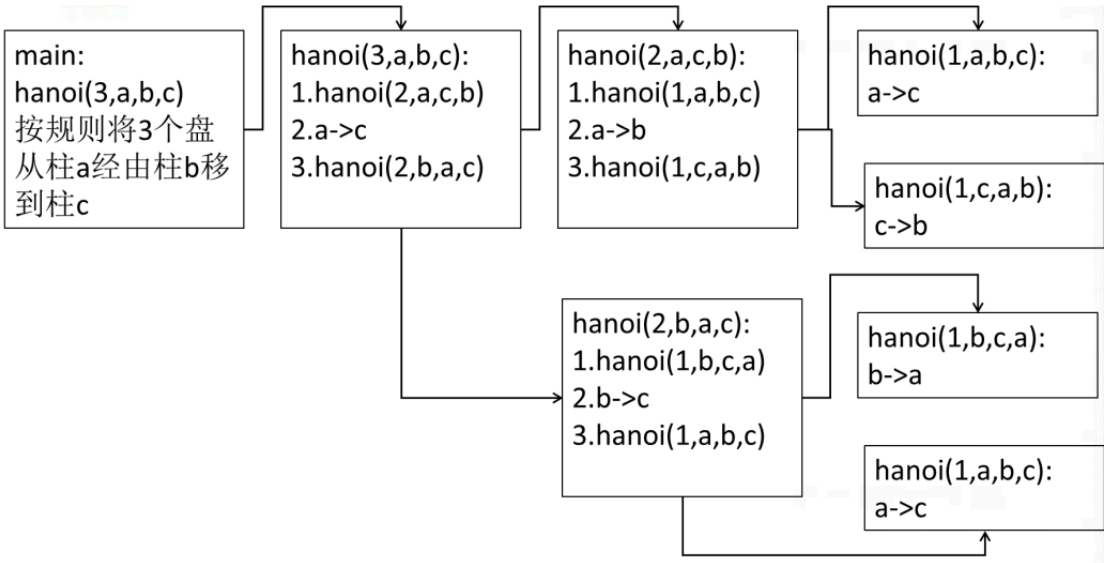
▼ **前缀表达式机算**

```
int Prefix(string str){
  SqStack Snum;
  InitStack(Snum);
  int len=str.length(),tmp=0,pow=1;
  for(int i=len-1;i>=0;i--){
    if(str[i]>='0'&&str[i]<='9'){
      tmp+=pow*(str[i]-'0');
      pow*=10;
    }
    //遇到' '标志着前面的数读入完毕
    else if(str[i]==' '&& str[i+1]>='0' && str[i+1]<='9'){
      Push(Snum,tmp);
      tmp=0;
      pow=1;
    }
    else if(str[i]!=' '){
      //从栈中取两个元素进行运算,运算后压入栈
```

```
            int x=Pop(Snum);
            int y=Pop(Snum);
            Push(Snum,calc(str[i],y,x));
        }
    }
    if(Snum.top==0) return Snum.data[0];
    return -1;
}
```

## ▼ 汉诺塔游戏

```
void hanoi(int n, char x, char y, char z, int &i) {
    // 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
    if (n == 1) {
        move(x, 1, z); // 将编号为1的圆盘从 x 移到 z
        i++;
    } else {
        hanoi(n - 1, x, z, y, i); // 将 x 上编号为1至 n-1 的圆盘移到 y, z 作辅助塔
        move(x, n, z); // 将编号为 n 的圆盘从 x 移到 z
        i++;
        hanoi(n - 1, y, x, z, i); // 将 y 上编号为1至 n-1 的圆盘移到 z, x 作辅助塔
    }
}
```




# ▼ 队列

## ▼ 抽象数据类型

ADT List {

数据对象：$D\{a_i \mid 1 \le i \le n, n>0, a_i \in Elemtype\}$

数据关系：$R=\{<a_i, a_{i+1}> \mid a_i, a_{i+1} \in D, i=1,2,\dots n-1\}$

基本运算：

```
InitQueue(&q);
ClearQueue(&q);
QueueEmpty(q);
QueueLength(q);
EnQueue(&q,e);
DeQueue(&q,e);
GetFront(q,&e);
```

}

## ▼ 顺序队列

普通队列会出现"假溢出"

**循环队列**

- 牺牲一个空间（rear指向队尾的下一个，front指向队头）
- 若要利用全部空间：1. struct里加size　2. struct加tag（记录最近一次是删除还是插入）
- 空：$rear = front$
- 满：$(rear+1)\%MAXSIZE = front$
- 长度：$(rear+MAXSIZE-front)\%MAXSIZE$

```
typedef struct{
    int data[MaxSize];
    int front,rear;
}SqQueue;

void InitQueue(SqQueue &Q){
    Q.rear=Q.front=0;
}
bool QueueEmpty(SqQueue Q){
    return (Q.rear==Q.front);
}
int GetLen(SqQueue Q){
    return (Q.rear+MaxSize-Q.front)%MaxSize;
}

bool EnQueue(SqQueue &Q,int x){
    if(QueueFull(Q)) return false;
    Q.data[Q.rear]=x;
    Q.rear=(Q.rear+1)%MaxSize;
    return true;
}
bool DeQueue(SqQueue &Q,int &x){
    if(Q.rear==Q.front) return false;
    x=Q.data[Q.front];
    Q.front=(Q.front+1)%MaxSize;
    return true;
}
```

```
bool GetHead(SqQueue Q,int &x){
    if(Q.rear==Q.front) return false;
    x=Q.data[Q.front];
    return true;
}
```

## ▼ 链式队列

```
typedef struct LinkNode{
    int data;
    struct LinkNode *next;
}LinkNode;
typedef struct{
    LinkNode *front,*rear;
}LinkQueue;
//如果需要可以增加length
```

带头结点

```
void InitQueue(LinkQueue &Q){
    Q.front=Q.rear=(LinkNode*)malloc(sizeof(LinkNode));
    Q.front→next=NULL;
}
bool IsEmpty(LinkQueue Q){
    return (Q.front==Q.rear);
}
void EnQueue(LinkQueue &Q,int x){
    LinkNode *s=(LinkNode *)malloc(sizeof(LinkNode));
    s→data=x;
    s→next=NULL;
    Q.rear→next=s;
    Q.rear=s;  //改尾指针
}
bool DeQueue(LinkQueue &Q,int &x){
    if(IsEmpty(Q)) return false;
    LinkNode *p=Q.front→next;
```

```
    x=p→data;
    Q.front→next=p→next;
    //最后一个结点出队
    if(Q.rear==p){
        Q.rear=Q.front;
    }
    free(p);
    return true;
}

bool GetFront(LinkQueue Q,int &x){
    if(IsEmpty(Q)) return false;
    x=Q.front→next→data;
    return true;
}
void ClearQueue(LinkQueue &Q){
    if(IsEmpty(Q)){
        printf("队列已为空\n");
        return;
    }
    LinkNode *p=Q.front→next,*q;
    while(p!=Q.rear){
        q=p;
        p=p→next;
        free(q);
    }
    free(p);
    Q.rear=Q.front;
    Q.rear→next=NULL;
    Q.front→next=NULL;
}
```

**用链接方式存储的队列，在进行删除运算时。**
**A. 仅修改头指针**
**B. 仅修改尾指针**
**C. 头、尾指针都要修改**
**D. 头、尾指针可能都要修改**

解析：本题考点是队列的基本操作。链接方式存储的队列，一般都是在队头进行删除运算，头指针需要修改，但当删除队列中最后一个元素时，头、尾指针都需要修改。因此，本题参考答案是D。

# ▼ 应用

## 杨辉三角计算

```
void yanghui(int n) {
   // 打印输出杨辉三角的前 n( n>0 )行
   Queue Q;
   int i, s, e, k = 1;

   for (i = 1; i <= n; i++) {
      cout << ' ';
   }
   cout << '1' << endl; // 在中心位置输出杨辉三角最顶端的"1"

   InitQueue(Q, n + 2); // 设置最大容量为 n+2 的空队列
   EnQueue(Q, 0); // 添加行界值
   EnQueue(Q, 1);
   EnQueue(Q, 1); // 第一行的值入队列

   while (k < n) { // 通过循环队列输出前 n-1 行的值
      for (i = 1; i <= n - k; i++) {
         cout << ' '; // 输出n-k个空格以保持三角型
      }
      EnQueue(Q, 0); // 行界值“0”入队列
      do { // 输出第 k 行，计算第 k+1 行
         Dequeue(Q, s);
         GetHead(Q, e);
         if (e) cout << e << ' '; // 若e为非行界值0，则打印输出 e 的值并加一空
         else cout << endl; // 否则回车换行，为下一行输出做准备
         EnQueue(Q, s + e);
```

```
        } while (e != 0);
        k++;
    }
    Dequeue(Q, e); // 行界值“0”出队列
    while (!QueueEmpty(Q)) { // 单独处理第 n 行的值的输出
        Dequeue(Q, e);
        cout << e << ' ';
    }
}
```