**第三章 栈和队列**

一、选择题

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1.B | 2.1B | 2.2A | 2.3B | 2.4D | 2.5.C | 3.B | 4.D | 5.D | 6.C | 7.D | 8.B |
| 9.D | 10.D | 11.D | 12.C | 13.B | 14.C | 15.B | 16.D | 17.B | 18.B | 19.B | 20.D |
| 21.D | 22.D | 23.D | 24.C | 25.A | 26.A | 27.D | 28.B | 29.BD | 30.C | 31.B | 32.C |
| 33.1B | 33.2A | 33.3C | 33.4C | 33.5F | 34.C | 35.C | 36.A | 37.AD |  |  |  |

二、判断题

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1.√ | 2.√ | 3. √ | 4. √ | 5.× | 6.√ | 7.√ | 8. √ | 9. √ | 10.× | 11. √ | 12.× |
| 13. × | 14.× | 15. √ | 16.× | 17.√ | 18.× | 19.√ | 20. √ |  |  |  |  |

部分答案解释如下。

1. 尾递归的消除就不需用栈
2. 这个数是前序序列为1,2,3,…,n，所能得到的不相似的二叉树的数目。

三、填空题

1、操作受限（或限定仅在表尾进行插入和删除操作） 后进先出

2、栈 3、3 1 2 4、23 100CH 5、0 n+1 top[1]+1=top[2]

6、两栈顶指针值相减的绝对值为1（或两栈顶指针相邻）。

7、(1)满 (2)空 (3)n (4)栈底 (5)两栈顶指针相邻（即值之差的绝对值为1）

8、链式存储结构 9、S×SS×S×× 10、data[++top]=x；

11、23.12.3\*2-4/34.5\*7/++108.9/+（注：表达式中的点(.)表示将数隔开，如23.12.3是三个数）

12、假溢出时大量移动数据元素。

13、(M+1) MOD N (M+1)% N； 14、队列 15、先进先出 16、先进先出

17、s=(LinkedList)malloc(**sizeof**(LNode))； s->data=x;s->next=r->next；r->next=s；r=s；

18、牺牲一个存储单元 设标记

19、（TAIL+1）MOD M=FRONT (数组下标0到M-1，若一定使用1到M，则取模为0者，值改取M

20、sq.front=(sq.front+1)%(M+1)；**return**(sq.data(sq.front))；(sq.rear+1)%(M+1)==sq.front；

21、栈 22、（rear-front+m）% m； 23、（R-P+N）% N；

24、（1）a[i]或a[1] （2）a[i] （3）pop（s）或s[1]；

25、（1）PUSH（OPTR，w）（2）POP（OPTR）（3）PUSH（OPND，operate（a，theta，b））

26、（1）T>0（2）i<n（3）T>0（4）top<n（5）top+1（6）true（7）i-1（8）top-1（9）T+w[i]（10）false

四、应用题

1、栈是只准在一端进行插入和删除操作的线性表，允许插入和删除的一端叫栈顶，另一端叫栈底。最后插入的元素最先删除，故栈也称后进先出（L**IF**O）表。

2、队列是允许在一端插入而在另一端删除的线性表，允许插入的一端叫队尾，允许删除的一端叫队头。最先插入队的元素最先离开（删除），故队列也常称先进先出（F**IF**O）表。

3、用常规意义下顺序存储结构的一维数组表示队列，由于队列的性质（队尾插入和队头删除），容易造成“假溢出”现象，即队尾已到达一维数组的高下标，不能再插入，然而队中元素个数小于队列的长度（容量）。循环队列是解决“假溢出”的一种方法。通常把一维数组看成首尾相接。在循环队列下，通常采用“牺牲一个存储单元”或“作标记”的方法解决“队满”和“队空”的判定问题。

4、（1）通常有两条规则。第一是给定序列中S的个数和X的个数相等；第二是从给定序列的开始，到给定序列中的任一位置，S的个数要大于或等于X的个数。

（2）可以得到相同的输出元素序列。例如，输入元素为A，B，C，则两个输入的合法序列ABC和BAC均可得到输出元素序列ABC。对于合法序列ABC，我们使用本题约定的S×S×S×操作序列；对于合法序列BAC，我们使用SS××S×操作序列。

5、三个：CDEBA，CDBEA，CDBAE

6、输入序列为123456，不能得出435612，其理由是，输出序列最后两元素是12，前面4个元素（4356）得到后，栈中元素剩12，且2在栈顶，不可能栈底元素1在栈顶元素2之前出栈。

得到135426的过程如下：1入栈并出栈，得到部分输出序列1；然后2和3入栈，3出栈，部分输出序列变为：13；接着4和5入栈，5，4和2依次出栈，部分输出序列变为13542；最后6入栈并退栈，得最终结果135426。

7、能得到出栈序列B、C、A、E、D，不能得到出栈序列D、B、A、C、E。其理由为：若出栈序列以D开头，说明在D之前的入栈元素是A、B和C，三个元素中C是栈顶元素，B和A不可能早于C出栈，故不可能得到D、B、A、C、E出栈序列。

8、借助栈结构，n个入栈元素可得到1/(n+1)((2n）！/（n!\*n!）)种出栈序列。本题4个元素，可有14种出栈序列，abcd和dcba就是其中两种。但dabc和adbc是不可能得到的两种。

9、不能得到序列2，5，3，4，6。栈可以用单链表实现，这就是链栈。由于栈只在栈顶操作，所以链栈通常不设头结点。

10、如果i<j，则对于pi<pj情况，说明pi在pj入栈前先出栈。而对于pi>pj的情况，则说明要将pj压到pi之上，也就是在pj出栈之后pi才能出栈。这就说明，对于i<j<k，不可能出现pj<pk<pi的输出序列。换句话说，对于输入序列1，2，3，不可能出现3，1，2的输出序列。

11、（1）能得到325641。在123依次进栈后，3和2出栈，得部分输出序列32；然后4，5入栈，5出栈，得部分出栈序列325；6入栈并出栈，得部分输出序列3256；最后退栈，直到栈空。得输出序列325641。其操作序列为AAADDAADADDD。

（2）不能得到输出顺序为154623的序列。部分合法操作序列为ADAAAADDAD，得到部分输出序列1546后，栈中元素为23，3在栈顶，故不可能2先出栈，得不到输出序列154623。

12、（1）一个函数在结束本函数之前，直接或间接调用函数自身，称为递归。例如，函数f在执行中，又调用函数f自身，这称为直接递归；若函数f在执行中，调用函数g，而g在执行中，又调用函数f，这称为间接递归。在实际应用中，多为直接递归，也常简称为递归。

（2）递归程序的优点是程序结构简单、清晰，易证明其正确性。缺点是执行中占内存空间较多，运行效率低。

（3）递归程序执行中需借助栈这种数据结构来实现。

（4）递归程序的入口语句和出口语句一般用条件判断语句来实现。递归程序由基本项和归纳项组成。基本项是递归程序出口，即不再递归即可求出结果的部分；归纳项是将原来问题化成简单的且与原来形式一样的问题，即向着“基本项”发展，最终“到达”基本项。

13、函数调用结束时vol=14。执行过程图示如下：

vol（4） vol(4)=vol(3)+5

14 =vol(2)+3+5

=vol(1)+4+3+5

vol（3）+5 =vol(0)+2+4+3+5

9 =0+2+4+3+5

=14

vol（2）+3

6

vol（1）+4

2

vol（0）+2

0

14、过程p递归调用自身时，过程p由内部定义的局部变量在p的2次调用期间，不占同一数据区。每次调用都保留其数据区，这是递归定义所决定，用“递归工作栈”来实现。

15、设Hn为n个盘子的Hanoi塔的移动次数。(假定n个盘子从钢针X移到钢针Z，可借助钢针Y)

则 Hn =2Hn-1+1 //先将n-1个盘子从X移到Y，第n个盘子移到Z，再将那n-1个移到Z

=2（2Hn-2+1）+1

=22 Hn-2+2+1

=22（2Hn-3+1）+2+1

=23 Hn-3+22+2+1

·

·

·

= 2k Hn-k+2k-1 +2k-2 +…+21 +20

=2n-1 H1+2n-2+2n-3+…+21+20

因为H1=1，所以原式Hn=2n-1+2n-2+…+21+20=2n-1

故总盘数为n的Hanoi塔的移动次数是2n-1。

16、运行结果为：1 2 1 3 1 2 1（注：运行结果是每行一个数，为节省篇幅，放到一行。）

17、两栈共享一向量空间（一维数组），栈底设在数组的两端，两栈顶相邻时为栈满。设共享数组为S[MAX]，则一个栈顶指针为-1，另一个栈顶指针为MAX时，栈为空。

用C写的入栈操作push（i，x）如下：

const MAX=共享栈可能达到的最大容量

**typedef** **struct** node

{elemtype s[MAX]；

**int**  top[2]；

}anode；

anode ds;

**int** push(**int** i,elemtype x)

//ds为容量有MAX个类型为elemtype的元素的一维数组，由两个栈共享其空间。i的值为0或1，x为类型为elemtype的元素。本算法将x压入栈中。如压栈成功，返回1；否则，返回0。

{**if**（ds.top[1]-ds.top[0]==1）{printf（“栈满\n”）；**return**（0）；}

**switch**（i）

{**case** 0：ds.s[++ds.top[i]]=x；**break**；

**case** 1：ds.s[--ds.top[i]]=x；

**return**（1）；}//入栈成功。

}

18、本程序段查找栈S中有无整数为k的元素，如有，则删除。采用的办法使用另一个栈T。在S栈元素退栈时，若退栈元素不是整数k，则压入T栈。遇整数k，k不入T栈，然后将T栈元素全部退栈，并依次压入栈S中，实现了在S中删除整数k的目的。若S中无整数k，则在S退成空栈后，再将T栈元素退栈，并依次压入S栈。直至T栈空。这后一种情况下S栈内容操作前后不变。

19、中缀表达式8-(3+5)\*(5-6/2）的后缀表达式是： 8 3 5 + 5 6 2 / - \* -

栈的变化过程图略（请参见22题），表达式生成过程为：



中缀表达式exp1转为后缀表达式exp2的规则如下：

设操作符栈s，初始为空栈后，压入优先级最低的操作符‘#’。对中缀表达式从左向右扫描，遇操作数，直接写入exp2；若是操作符（记为w），分如下情况处理，直至表达式exp1扫描完毕。

（1）w为一般操作符（’+’，’-‘，’\*’，’/’等），要与栈顶操作符比较优先级，若w优先级高于栈顶操作符，则入栈；否则，栈顶运算符退栈到exp2，w再与新栈顶操作符作上述比较处理，直至w入栈。

（2）w为左括号（’（’），w入栈。

（3）w为右括号（’）’），操作符栈退栈并进入exp2，直到碰到左括号为止，左括号退栈（不能进入exp2），右括号也丢掉，达到exp2中消除括号的目的。

（4）w为‘#’，表示中缀表达式exp1结束，操作符栈退栈到exp2，直至碰到‘#’，退栈，整个操作结束。

这里，再介绍一种简单方法。中缀表达式转为后缀表达式有三步：首先，将中缀表达式中所有的子表达式按计算规则用嵌套括号括起来；接着，顺序将每对括号中的运算符移到相应括号的后面；最后，删除所有括号。

例如，将中缀表达式8-(3+5)\*(5-6/2）转为后缀表达式。按如上步骤：

执行完上面第一步后为：(8-((3+5)\*(5-(6/2))))；

执行完上面第二步后为：(8((35)+(5(62)/)-)\*)- ；

执行完上面第三步后为：8 3 5 + 5 6 2 / - \* - 。

可用类似方法将中缀表达式转为前缀表达式。

20、中缀表达式转为后缀表达式的规则基本上与上面19题相同，不同之处是对运算符\*\*优先级的规定。在算术运算中，先乘除后加减，先括号内后括号外，相同级别的运算符按从左到右的规则运算。而对\*\*运算符，其优先级同常规理解，即高于加减乘除而小于左括号。为了适应本题中“从右到左计算”的要求，规定栈顶运算符\*\*的级别小于正从表达式中读出的运算符\*\*，即刚读出的运算符\*\*级别高于栈顶运算符\*\*，因此也入栈。

下面以A\*\*B\*\*C为例说明实现过程。

读入A，不是操作符，直接写入结果表达式。再读入\*，这里规定在读入\*后，不能立即当乘号处理，要看下一个符号，若下个符号不是\*，则前个\*是乘号。这里因为下一个待读的符号也是\*，故认为\*\*是一个运算符，与运算符栈顶比较（运算符栈顶初始化后，首先压入‘#’作为开始标志），其级别高于‘#’，入栈。再读入B，直接进入结果表达式。接着读入\*\*，与栈顶比较，均为\*\*，我们规定，后读入的\*\*级别高于栈顶的\*\*，因此\*\*入栈。接着读入C，直接到结果表达式。现在的结果（后缀）表达式是ABC。最后读入‘#’，表示输入表达式结束，这时运算符栈中从栈顶到栈底有两个\*\*和一个‘#’。两个运算符\*\*退栈至结果表达式，结果表达式变为ABC\*\*\*\*。运算符栈中只剩‘#’，退栈，运算结束。

21、（1）sum=21。当x为局部变量时，每次递归调用，都要给局部变量分配存储单元，故x数值4，9，6和2均保留，其递归过程示意图如下：

sum（4）

21

sum（3）+4 （x=4）

17

sum（2）+9 （x=9）

8

sum（1）+6 （x=6）

2

sum（0）+2 （x=2）

0

(2) sum=8,当x为全局变量时，在程序的整个执行期间，x只占一个存储单元，先后读入的4个数(4,9,6,2),仅最后一个起作用。当递归调用结束，逐层返回时sum:=sum(n-1)+x表达式中，x就是2，所以结果为sum=8。

22、设操作数栈是opnd，操作符栈是optr，对算术表达式A-B\*C/D-E↑F求值，过程如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 步骤 | opnd栈 | optr栈 | 输入字符 | 主要操作 |
| 初始 |  | # | A-B\*C/D-E↑F# | PUSH(OPTR,’#’) |
| 1 | A | # | A-B\*C/D-E↑F# | PUSH(OPND,A) |
| 2 | A | # - | -B\*C/D-E↑F# | PUSH(OPTR,’-’) |
| 3 | AB | # - | B\*C/D-E↑F# | PUSH(OPND,B) |
| 4 | AB | # - \* | \*C/D-E↑F# | PUSH(OPTR,’\*’) |
| 5 | ABC | # - \* | C/D-E↑F# | PUSH(OPND,C) |
| 6 | AT(T=B\*C) | # - / | /D-E↑F# | PUSH(OPND,POP(OPND)\*POP(OPND))  PUSH(OPTR,’/’) |
| 7 | ATD | # - / | D-E↑F# | PUSH(OPND,D) |
| 8 | AT(T=T/D)  T(T=A-T) | # -  # - | -E↑F# | x=POP(OPND);y=POP(OPND)  PUSH(OPND,y/x);  x=POP(OPND);y=POP(OPND);  PUSH(OPND,y-x)  PUSH(OPTR,’-’) |
| 9 | TE | # - | E↑F# | PUSH(OPND,E) |
| 10 | TE | # -↑ | ↑F# | PUSH(OPTR, ‘↑’) |
| 11 | TEF | # -↑ | F# | PUSH(OPND,F) |
| 12 | TE  TS(S=E↑F)    R(R=T-S) | #-    # | # | X=POP(OPND) Y=POP(OPND)  POP(OPTR) PUSH(OPND,y↑x)  x=POP(OPND) y=POP(OPND)  POP(OPTR) PUSH(OPND,y-x) |

23、

|  |  |  |  |
| --- | --- | --- | --- |
| 步骤 | 栈S1 | 栈S2 | 输入的算术表达式（按字符读入） |
| 初始 |  | ® | A-B\*C/D+E/F® |
| 1 | A | ® | A-B\*C/D+E/F® |
| 2 | A | ®- | -B\*C/D+E/F® |
| 3 | AB | ®- | B\*C/D+E/F® |
| 4 | AB | ®-\* | \*C/D+E/F® |
| 5 | ABC | ®-\* | C/D+E/F® |
| 6 | AT1（注：T1=B\*C） | ®-/ | /D+E/F® |
| 7 | AT1D | ®-/ | D+E/F® |
| 8 | AT2（注：T2=T1/D）  T3 （注：T3=A-T2） | ®-  ®+ | +E/F® |
| 9 | T3E | ®+ | E/F® |
| 10 | T3E | ®+/ | /F® |
| 11 | T3EF | ®+/ | F® |
| 12 | T3T4（注：T4=E/F）  T5（注：T5= T3+ T4） | ®+ ® | ® |

24、XSXXXSSSXXSXXSXXSSSS

25、S1和S2共享内存中一片连续空间（地址1到m），可以将S1和S2的栈底设在两端，两栈顶向共享空间的中心延伸，仅当两栈顶指针相邻（两栈顶指针值之差的绝对值等于1）时，判断为栈满，当一个栈顶指针为0，另一个栈顶指针m+1时为两栈均空。

26、设栈S1和栈S2共享向量V[1..m]，初始时，栈S1的栈顶指针top[0]=0，栈S2的栈顶指针top[1]=m+1，当top[0]=0为左栈空，top[1]=m+1为右栈空；当top[0]=0并且top[1]=m+1时为全栈空。当top[1]-top[0]=1时为栈满。

27、（1）每个栈仅用一个顺序存储空间时，操作简便，但分配存储空间小了，容易产生溢出，分配空间大了，容易造成浪费，各栈不能共享空间。

（2）多个栈共享一个顺序存储空间，充分利用了存储空间，只有在整个存储空间都用完时才能产生溢出，其缺点是当一个栈满时要向左、右栈查询有无空闲单元。如果有，则要移动元素和修改相关的栈底和栈顶指针。当接近栈满时，查询空闲单元、移动元素和修改栈底栈顶指针的操作频繁，计算复杂并且耗费时间。

（3）多个链栈一般不考虑栈的溢出（仅受用户内存空间限制），缺点是栈中元素要以指针相链接，比顺序存储多占用了存储空间。

28、设top1和top2分别为栈1和2的栈顶指针

（1）入栈主要语句

**if**(top2-top1==1) {printf(“栈满\n”); exit(0);}

**case**1:top1++；SPACE[top1]=x; //设x为入栈元素。

**case**2:top2--；SPACE[top2]=x;

出栈主要语句

**case**1：**if**（top1==-1） {printf（“栈空\n”）；exit（0）；}

top1--；**return**（SPACE[top1+1]）； //返回出栈元素。

**case**2：**if**（top2==N）{printf（“栈空\n”）；exit（0）；}

top2++；**return**（SPACE[top2-1]）； //返回出栈元素。

（2）栈满条件：top2-top1=1

栈空条件：top1=-1并且top2=N //top1=-1为左栈空，top2=N为右栈空

29、设顺序存储队列用一维数组q[m]表示，其中m为队列中元素个数，队列中元素在向量中的下标从0到m-1。设队头指针为front，队尾指针是rear，约定front指向队头元素的前一位置，rear指向队尾元素。当front等于-1时队空，rear等于m-1时为队满。由于队列的性质（“删除”在队头而“插入”在队尾），所以当队尾指针rear等于m-1时，若front不等于-1，则队列中仍有空闲单元，所以队列并不是真满。这时若再有入队操作，会造成假“溢出”。其解决办法有二，一是将队列元素向前“平移”（占用0至rear-front-1）；二是将队列看成首尾相连，即循环队列（0..m-1）。在循环队列下，仍定义front=rear时为队空，而判断队满则用两种办法，一是用“牺牲一个单元”，即rear+1=front（准确记是（rear+1）%m=front，m是队列容量）时为队满。另一种解法是“设标记”方法，如设标记tag，tag等于0情况下，若删除时导致front=rear为队空；tag=1情况下，若因插入导致front=rear则为队满。

30、见上题29的解答。 31、参见上面29题。

32、**typedef** **struct** node

{elemtype elemcq[m]; //m为队列最大可能的容量。

**int** front ,rear; //front和rear分别为队头和队尾指针。

}cqnode;

cqnode cq;

1. 初始状态

cq.front=cq.rear=0;

1. 队列空

cq.front==cq.rear;

1. 队列满

(cq.rear+1)%m==cq.front;

33、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模拟一个队列的输入：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。

34、（1）队空s.front=s.rear； //设s是sequeuetp类型变量

（2）队满：（s.rear+1）MOD MAXSIZE=s.front //数组下标为0.. MAXSIZE-1

具体参见本章应用题第29题

35、**typedef** **struct**

{elemtp q[m];

**int** front,count; //front是队首指针，count是队列中元素个数。

}cqnode; //定义类型标识符。

(1)判空：**int** Empty(cqnode cq) //cq是cqnode类型的变量

{**if**(cq.count==0) **return**(1)；**else** **return**(0); //空队列}

入队: **int** EnQueue(cqnode cq，elemtp x)

{**if**(count==m){printf(“队满\n”)；exit(0); }

cq.q[(cq.front+count)%m]=x; //x入队

count++; **return**(1); //队列中元素个数增加1,入队成功。

}

出队: **int** DelQueue(cqnode cq)

{**if** (count==0){printf(“队空\n”)；**return**(0);}

printf(“出队元素”，cq.q[cq.front]);

x=cq.q[cq.front];

cq.front=(cq.front+1)%m; //计算新的队头指针。

**return**(x)

}

(2) 队列中能容纳的元素的个数为m。队头指针front指向队头元素。

36、循环队列中元素个数为（REAR-FRONT+N）%N。其中FRONT是队首指针，指向队首元素的前一位置；REAR是队尾指针，指向队尾元素；N是队列最大长度。

37、循环队列解决了用向量表示队列所出现的“假溢出”问题，但同时又出现了如何判断队列的满与空问题。例如：在队列长10的循环队列中，若假定队头指针front指向队头元素的前一位置，而队尾指针指向队尾元素，则front=3，rear=7的情况下，连续出队4个元素，则front==rear为队空；如果连续入队6个元素，则front==rear为队满。如何判断这种情况下的队满与队空，一般采取牺牲一个单元的做法或设标记法。即假设front==rear为队空，而（rear+1）%表长==front为队满，或通过设标记tag。若tag=0，front==rear则为队空；若tag=1，因入队而使得front==rear，则为队满。

本题中队列尾指针rear，指向队尾元素的下一位置,listarray[rear]表示下一个入队的元素。在这种情况下，我们可规定，队头指针front指向队首元素。当front==rear时为队空，当（rear+1）%n=front时为队满。出队操作（在队列不空情况下）队头指针是front=（front+1）%n，

38、既不能由输入受限的双端队列得到，也不能由输出受限的双端队列得到的输出序列是dbca。

39、（1）4132 （2）4213 （3）4231

40、（1）队空的初始条件：f=r=0；

（2）执行操作A3后，r=3；// A3表示三次入队操作

执行操作D1后，f=1；//D1表示一次出队操作

执行操作A5后，r=0；

执行操作D2后，f=3；

执行操作A1后，r=1；

执行操作D2后，f=5；

执行操作A4后，按溢出处理。因为执行A3后，r=4，这时队满，若再执行A操作，则出错。

41．一般说，高级语言的变量名是以字母开头的字母数字序列。故本题答案是:

AP321,PA321,P3A21,P32A1,P321A。

五、算法设计题

1、[题目分析]两栈共享向量空间，将两栈栈底设在向量两端，初始时，s1栈顶指针为-1，s2栈顶为maxsize。两栈顶指针相邻时为栈满。两栈顶相向，迎面增长，栈顶指针指向栈顶元素。

#define maxsize 两栈共享顺序存储空间所能达到的最多元素数

#define elemtp **int**  //假设元素类型为整型

**typedef** **struct**

{elemtp stack[maxsize]; //栈空间

**int** top[2]; //top为两个栈顶指针

}stk;

stk s; //s是如上定义的结构类型变量，为全局变量。

(1)入栈操作：

**int** push(**int** i,**int** x)

//入栈操作。i为栈号，i=0表示左边的栈s1，i=1表示右边的栈s2，x是入栈元素。入栈成功返回1，否则返回0。

{**if**(i<0||i>1){printf(“栈号输入不对”);exit(0);}

**if**(s.top[1]-s.top[0]==1) {printf(“栈已满\n”);**return**(0);}

**switch**(i)

{**case** 0: s.stack[++s.top[0]]=x; **return**(1); **break**;

**case** 1: s.stack[--s.top[1]]=x; **return**(1);

}

}//push

（2） 退栈操作

elemtp pop(**int** i)

//退栈算法。i代表栈号，i=0时为s1栈，i=1时为s2栈。退栈成功返回退栈元素，否则返回-1。

{**if**(i<0 || i>1){printf(“栈号输入错误\n”)；exit(0);}

**switch**(i)

{**case** 0: **if**(s.top[0]==-1) {printf(“栈空\n”)；**return**（-1）；}

**else** **return**(s.stack[s.top[0]--]);

**case** 1: **if**(s.top[1]==maxsize {printf(“栈空\n”); **return**(-1);}

**else** **return**(s.stack[s.top[1]++]);

}

}//算法结束

[算法讨论] 请注意算法中两栈入栈和退栈时的栈顶指针的计算。两栈共享空间示意图略，s1栈是通常意义下的栈，而s2栈入栈操作时，其栈顶指针左移（减1），退栈时，栈顶指针右移（加1）。

2、#define maxsize 栈空间容量

**void** InOutS(**int** s[maxsize])

//s是元素为整数的栈，本算法进行入栈和退栈操作。

{**int** top=0; //top为栈顶指针，定义top=0时为栈空。

**for**(i=1; i<=n; i++) //n个整数序列作处理。

{scanf(“%d”,&x); //从键盘读入整数序列。

**if**(x!=-1) // 读入的整数不等于-1时入栈。

**if**(top==maxsize-1){printf(“栈满\n”);exit(0);}**else** s[++top]=x; //x入栈。

**else** //读入的整数等于-1时退栈。

{**if**(top==0){printf(“栈空\n”);exit(0);} **else** printf(“出栈元素是%d\n”,s[top--])；}}

}//算法结束。

3、[题目分析]判断表达式中括号是否匹配，可通过栈，简单说是左括号时进栈，右括号时退栈。退栈时，若栈顶元素是左括号，则新读入的右括号与栈顶左括号就可消去。如此下去，输入表达式结束时，栈为空则正确，否则括号不匹配。

**int** EXYX(**char** E[],**int** n)

//E[]是有n字符的字符数组，存放字符串表达式，以‘#’结束。本算法判断表达式中圆括号是否匹配。

{**char** s[30]; //s是一维数组，容量足够大，用作存放括号的栈。

**int** top=0; //top用作栈顶指针。

s[top]= ‘#’; //‘#’先入栈，用于和表达式结束符号‘#’匹配。

**int** i=0; //字符数组E的工作指针。

**while**(E[i]!= ‘#’) //逐字符处理字符表达式的数组。

**switch**(E[i])

{**case**‘(’: s[++top]=‘(’; i++ ; **break** ;

**case**‘)’: **if**(s[top]==‘(’{top--; i++; **break**;}

**else**{printf(“括号不配对”);exit(0);}

**case**‘#’: **if**(s[top]==‘#’){printf(“括号配对\n”);**return** (1);}

**else** {printf(“ 括号不配对\n”);**return** (0);} //括号不配对

**default** : i++; //读入其它字符，不作处理。

}

}//算法结束。

[算法讨论]本题是用栈判断括号匹配的特例：只检查圆括号的配对。一般情况是检查花括号（‘{’，‘}’）、方括号（‘[’，‘]’）和圆括号（‘（’，‘）’）的配对问题。编写算法中如遇左括号（‘{’，‘[’，或‘（’）就压入栈中，如遇右括号（‘}’，‘]’，或‘）’），则与栈顶元素比较，如是与其配对的括号（左花括号，左方括号或左圆括号），则弹出栈顶元素；否则，就结论括号不配对。在读入表达式结束符‘#’时，栈中若应只剩‘#’，表示括号全部配对成功；否则表示括号不匹配。

另外，由于本题只是检查括号是否匹配，故对从表达式中读入的不是括号的那些字符，一律未作处理。再有，假设栈容量足够大，因此入栈时未判断溢出。

4、[题目分析]逆波兰表达式(即后缀表达式)求值规则如下：设立运算数栈OPND,对表达式从左到右扫描(读入)，当表达式中扫描到数时，压入OPND栈。当扫描到运算符时，从OPND退出两个数，进行相应运算，结果再压入OPND栈。这个过程一直进行到读出表达式结束符$，这时OPND栈中只有一个数，就是结果。

**float** expr( )

//从键盘输入逆波兰表达式，以‘$’表示输入结束，本算法求逆波兰式表达式的值。

｛**float** OPND[30]; // OPND是操作数栈。

　init(OPND); //两栈初始化。

**float** num=0.0; //数字初始化。

scanf (“%c”,&x);//x是字符型变量。

**while**(x!=’$’)

{**switch**

{**case**‘0’<=x<=’9’:**while**((x>=’0’&&x<=’9’)||x==’.’) //拼数

**if**(x!=’.’) //处理整数

{num=num\*10+（ord(x)-ord(‘0’)）; scanf(“%c”,&x);}

**else** //处理小数部分。

{scale=10.0; scanf(“%c”,&x);

**while**(x>=’0’&&x<=’9’)

{num=num+(ord(x)-ord(‘0’)/scale;

scale=scale\*10; scanf(“%c”,&x); }

}//**else**

push(OPND,num); num=0.0;//数压入栈，下个数初始化

**case** x=‘ ’:**break**; //遇空格，继续读下一个字符。

**case** x=‘+’:push(OPND,pop(OPND)+pop(OPND));**break**;

**case** x=‘-’:x1=pop(OPND);x2=pop(OPND);push(OPND,x2-x1);**break**;

**case** x=‘\*’:push(OPND,pop(OPND)\*pop(OPND));**break**;

**case** x=‘/’:x1=pop(OPND);x2=pop(OPND);push(OPND,x2/x1);**break**;

**default**: //其它符号不作处理。

}//结束**switch**

scanf(“%c”,&x);//读入表达式中下一个字符。

}//结束**while**（x！=‘$’）

printf(“后缀表达式的值为%f”,pop(OPND));

}//算法结束。

[算法讨论]假设输入的后缀表达式是正确的，未作错误检查。算法中拼数部分是核心。若遇到大于等于‘0’且小于等于‘9’的字符，认为是数。这种字符的序号减去字符‘0’的序号得出数。对于整数，每读入一个数字字符，前面得到的部分数要乘上10再加新读入的数得到新的部分数。当读到小数点，认为数的整数部分已完，要接着处理小数部分。小数部分的数要除以10（或10的幂数）变成十分位，百分位，千分位数等等，与前面部分数相加。在拼数过程中，若遇非数字字符，表示数已拼完，将数压入栈中，并且将变量num恢复为0，准备下一个数。这时对新读入的字符进入‘+’、‘-’、‘\*’、‘/’及空格的判断，因此在结束处理数字字符的**case**后，不能加入**break**语句。

5、（1）A和D是合法序列，B和C 是非法序列。

（2）设被判定的操作序列已存入一维数组A中。

**int** Judge(**char** A[])

//判断字符数组A中的输入输出序列是否是合法序列。如是，返回true，否则返回false。

{i=0; //i为下标。

j=k=0; //j和k分别为I和字母O的的个数。

**while**(A[i]!=‘\0’) //当未到字符数组尾就作。

{**switch**(A[i])

{**case**‘I’: j++; **break**; //入栈次数增1。

**case**‘O’: k++; **if**(k>j){printf(“序列非法\n”)；exit(0);}

}

i++; //不论A[i]是‘I’或‘O’，指针i均后移。}

**if**(j!=k) {printf(“序列非法\n”)；**return**(false);}

**else** {printf(“序列合法\n”)；**return**(true);}

}//算法结束。

[算法讨论]在入栈出栈序列（即由‘I’和‘O’组成的字符串）的任一位置，入栈次数（‘I’的个数）都必须大于等于出栈次数（即‘O’的个数），否则视作非法序列，立即给出信息，退出算法。整个序列（即读到字符数组中字符串的结束标记‘\0’），入栈次数必须等于出栈次数（题目中要求栈的初态和终态都为空），否则视为非法序列。

6、[题目分析]表达式中的括号有以下三对：‘（’、‘）’、‘[’、‘]’、‘{’、‘}’，使用栈，当为左括号时入栈，右括号时，若栈顶是其对应的左括号，则退栈，若不是其对应的左括号，则结论为括号不配对。当表达式结束，若栈为空，则结论表达式括号配对，否则，结论表达式括号不配对。

**int** Match(LinkedList la)

//算术表达式存储在以la为头结点的单循环链表中，本算法判断括号是否正确配对

{**char** s[]; //s为字符栈，容量足够大

p=la->link; //p为工作指针，指向待处理结点

StackInit(s); //初始化栈s

**while** (p!=la) //循环到头结点为止

{**switch** (p->ch)

{**case** ‘(’:push(s,p->ch); **break**;

**case** ‘)’:**if**(StackEmpty(s)||StackGetTop(s)!=‘(’)

{printf(“括号不配对\n”); **return**(0);} **else** pop(s);**break**;

**case** ‘[’:push(s,p->ch); **break**;

**case** ‘]’: **if**(StackEmpty(s)||StackGetTop(s)!=‘[’)

{printf(“括号不配对\n”); **return**(0);} **else** pop(s);**break**;

**case** ‘{’:push(s,p->ch); **break**;

**case** ‘}’: **if**(StackEmpty(s)||StackGetTop(s)!=‘{’)

{printf(“括号不配对\n”); **return**(0);} **else** pop(s);**break**;

} p=p->link; 后移指针

}//**while**

**if** (StackEmpty(s)) {printf(“括号配对\n”); **return**(1);}

**else**{printf(“括号不配对\n”); **return**(0);}

}//算法match结束

[算法讨论]算法中对非括号的字符未加讨论。遇到右括号时，若栈空或栈顶元素不是其对应的左圆（方、花）括号，则结论括号不配对，退出运行。最后，若栈不空，仍结论括号不配对。

7、[题目分析]栈的特点是后进先出，队列的特点是先进先出。所以，用两个栈s1和s2模拟一个队列时，s1作输入栈，逐个元素压栈，以此模拟队列元素的入队。当需要出队时，将栈s1退栈并逐个压入栈s2中，s1中最先入栈的元素，在s2中处于栈顶。s2退栈，相当于队列的出队，实现了先进先出。显然，只有栈s2为空且s1也为空，才算是队列空。

(1) **int** enqueue(stack s1,elemtp x)

//s1是容量为n的栈，栈中元素类型是elemtp。本算法将x入栈，若入栈成功返回1，否则返回0。

{**if**(top1==n && !Sempty(s2)) //top1是栈s1的栈顶指针，是全局变量。

{printf(“栈满”);**return**(0);} //s1满s2非空,这时s1不能再入栈。

**if**(top1==n && Sempty(s2)) //若s2为空，先将s1退栈,元素再压栈到s2。

{**while**(!Sempty(s1)) {POP(s1,x);PUSH(s2,x);}

PUSH(s1,x); **return**(1); //x入栈，实现了队列元素的入队。

}

(2) **void** dequeue(stack s2,s1)

//s2是输出栈，本算法将s2栈顶元素退栈，实现队列元素的出队。

{**if**(!Sempty(s2)) //栈s2不空，则直接出队。

{POP(s2,x); printf(“出队元素为”,x); }

**else** //处理s2空栈。

**if**(Sempty(s1)) {printf(“队列空”);exit(0);}//若输入栈也为空，则判定队空。

**else** //先将栈s1倒入s2中，再作出队操作。

{**while**(!Sempty(s1)) {POP(s1,x);PUSH(s2,x);}

POP(s2,x); //s2退栈相当队列出队。

printf(“出队元素”，x);

}

}//结束算法dequue。

(3) **int** queue\_empty()

//本算法判用栈s1和s2模拟的队列是否为空。

{**if**(Sempty(s1)&&Sempty(s2)) **return**(1);//队列空。

**else** **return**(0); //队列不空。

}

[算法讨论]算法中假定栈s1和栈s2容量相同。出队从栈s2出，当s2为空时，若s1不空，则将s1倒入s2再出栈。入队在s1，当s1满后，若s2空，则将s1倒入s2，之后再入队。因此队列的容量为两栈容量之和。元素从栈s1倒入s2，必须在s2空的情况下才能进行，即在要求出队操作时，若s2空，则不论s1元素多少（只要不空），就要全部倒入s2中。

类似本题叙述的其它题的解答：

1. 该题同上面题本质相同，只有叙述不同，请参考上题答案。

8、[题目分析]本题要求用链接结构实现一个队列，我们可用链表结构来实现。一般说，由于队列的先进先出性质，所以队列常设队头指针和队尾指针。但题目中仅给出一个“全局指针p”，且要求入队和出队操作的时间复杂性是O（1），因此我们用只设尾指针的循环链表来实现队列。

1. **PROC** addq(VAR p:linkedlist,x:elemtp);

//p是数据域为data、链域为link的用循环链表表示的队列的尾指针，本算法是入队操作。

new(s); //申请新结点。假设有内存空间，否则系统给出出错信息。

s↑.data:=x; s↑.link:=p↑.link;//将s结点入队。

p↑.link:=s; p:=s; //尾指针p移至新的队尾。

**ENDP**;

1. **PROC** deleq(VAR p:linkedlist,VAR x:elemtp);

// p是数据域为data、链域为link的用循环链表表示的队列的尾指针，本算法实现队列元素的出队，若出队成功，返回出队元素，否则给出失败信息。

**IF** （p↑.link=p）**THEN**[writeln(“空队列”);**return**(0);]//带头结点的循环队列。

**ELSE**[s:=p↑.link↑.link; //找到队头元素。

p↑.link↑.link:=s↑.link; //删队头元素。

x:=s↑.data; //返回出队元素。

**IF** (p=s) **THEN** p:=p↑.link; //队列中只有一个结点，出队后成为空队列。

dispose(s); //回收出队元素所占存储空间。

]

**ENDP**;

[算法讨论]上述入队算法中，因链表结构，一般不必考虑空间溢出问题，算法简单。在出队算法中，首先要判断队列是否为空，另外，对出队元素，要判断是否因出队而成为空队列。否则，可能导致因删除出队结点而将尾指针删掉成为“悬挂变量”。

9、本题与上题本质上相同，现用类C语言编写入队和出队算法。

（1）**void** EnQueue (LinkedList rear, ElemType x)

// rear是带头结点的循环链队列的尾指针，本算法将元素x插入到队尾。

{ s= (LinkedList) malloc (**sizeof**(LNode)); //申请结点空间

s->data=x; s->next=rear->next; //将s结点链入队尾

rear->next=s; rear=s; //rear指向新队尾

}

（2）**void** DeQueue (LinkedList rear)

// rear是带头结点的循环链队列的尾指针，本算法执行出队操作，操作成功输出队头元素；否则给出出错信息。

{ **if** (rear->next==rear) { printf(“队空\n”); exit(0);}

s=rear->next->next; //s指向队头元素，

rear->next->next=s->next; //队头元素出队。

printf (“出队元素是”，s->data);

**if** (s==rear) rear=rear->next; //空队列

free(s);

}

10、[题目分析] 用一维数组 v[0..M-1]实现循环队列，其中M是队列长度。设队头指针 front和队尾指针rear，约定front指向队头元素的前一位置，rear指向队尾元素。定义front=rear时为队空，(rear+1)%m=front 为队满。约定队头端入队向下标小的方向发展，队尾端入队向下标大的方向发展。

（1）#define M 队列可能达到的最大长度

**typedef** **struct**

{ elemtp data[M];

**int** front,rear;

} cycqueue;

（2）elemtp delqueue ( cycqueue Q)

//Q是如上定义的循环队列，本算法实现从队尾删除，若删除成功，返回被删除元素，否则给出出错信息。

{ **if** (Q.front==Q.rear) {printf(“队列空”); exit(0);}

Q.rear=(Q.rear-1+M)%M; //修改队尾指针。

**return**(Q.data[(Q.rear+1+M)%M]); //返回出队元素。

}//从队尾删除算法结束

**void** enqueue (cycqueue Q, elemtp x)

// Q是顺序存储的循环队列，本算法实现“从队头插入”元素x。

{**if** (Q.rear==(Q.front-1+M)%M) {printf(“队满”; exit(0);)

Q.data[Q.front]=x; //x 入队列

Q.front=(Q.front-1+M)%M; //修改队头指针。

}// 结束从队头插入算法。

11、参见9。

12、[题目分析] 双端队列示意图如下（设maxsize =12）

0 1 2 3 4 5 6 7 8 9 10 11

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |

↑ ↑

end1 end2

用上述一维数组作存储结构，把它看作首尾相接的循环队列。可以在任一端（end1或end2）进行插入或删除。初始状态end1+1=end2被认为是队空状态；end1=end2被认为是队满状态。（左端队列）end1指向队尾元素的前一位置。end2指向（右端队列）队尾元素的后一位置。入队时判队满，出队（删除）时判队空。删除一个元素时，首先查找该元素，然后，从队尾将该元素前的元素依次向后或向前（视end1端或end2端而异）移动。

**FUNC** add (Qu:deque; var x:datatype;tag 0..1):integer;

//在双端队列Qu中插入元素x，若插入成功，返回插入元素在Qu中的下标；插入失败返回-1。tag=0表示在end1端插入；tag=1表示在end2端插入。

**IF** Qu.end1=Qu.end2 **THEN** [writeln(“队满”);**return**(-1);]

**CASE** tag OF

0: //在end1端插入

[Qu.end1:=x; //插入x

Qu.end1:=(Qu.end1-1) MOD maxsize; //修改end1

**RETURN**(Qu.end1+1) MOD maxsize); //返回插入元素的下标。

1: //在end2端插入

[Qu.end2:=x;

Qu.end2:=(Qu.end2+1) MOD maxsize;

**RETURN**(Qu.end2-1) MOD maxsize);

]

ENDC; //结束**CASE**语句

**ENDF**; //结束算法add

**FUNC** delete (Qu: deque; VAR x:datatype; tag:0..1):integer;

//本算法在双端队列Qu中删除元素x，tag=0时从end1端删除，tag=1时从end2端删除。删除成功返回1，否则返回0。

**IF** (Qu.end1+1) MOD maxsize=Qu.end2 **THEN** [writeln(“队空”);**return**(0);]

**CASE** tag OF

0: //从end1端删除

[i:=(Qu.end1+1) MOD maxsize; //i是end1端最后插入的元素下标。

**WHILE**(i<>Qu.end2) AND (Qu.elem[i]<>x) **DO**

i=(i+1) MOD maxsize;//查找被删除元素x的位置

**IF** (Qu.elem[i]=x) AND (i<>Qu.end2) **THEN**

[ j:=i;

**WHILE**((j-1+maxsize) MOD maxsize <>Qu.end1) **DO**

[Qu.elem[j]:=Qu.elem[(j-1+maxsize) MOD maxsize];

j:=(j-1+maxsize) MOD maxsize;

]//移动元素，覆盖达到删除

Qu.end1:=(Qu.end1+1) MOD maxsize; //修改end1指针

**RETURN**(1);

]

**ELSE** **RETURN**(0);

]//结束从end1端删除。

1: //从end2端删除

[i:=(Qu.end2-1+maxsize) MOD maxsize; //i是end2端最后插入的元素下标。

**WHILE**(i<>Qu.end1) AND (Qu.elem[i]<>x) **DO**

i=(i-1+maxsize) MOD maxsize;//查找被删除元素x的下标

**IF** (Qu.elem[i]=x) AND (i<>Qu.end1) **THEN** //被删除元素找到

[ j:=i;

**WHILE**((j+1) MOD maxsize <>Qu.end2) **DO**

[Qu.elem[j]:=Qu.elem[(j+1) MOD maxsize];

j:=(j+1) MOD maxsize;

]//移动元素，覆盖达到删除

Qu.end2:=(Qu.end2-1+maxsize) MOD maxsize; //修改end2指针

**RETURN**(1);//返回删除成功的信息

]

**ELSE** **RETURN**(0);//删除失败

]//结束在end2端删除。

ENDC;//结束**CASE**语句

**ENDF**;//结束delete

[算法讨论]请注意下标运算。(i+1) MOD maxsize容易理解，考虑到i-1可能为负的情况，所以求下个i时用了(i-1+maxsize) MOD maxsize。

13、[题目分析] 本题与上面12题基本相同，现用类C语言给出该双端队列的定义。

#define maxsize 32

**typedef** **struct**

{datatype elem[maxsize];

**int** end1,end2; //end1和end2取值范围是0..maxsize-1

} deque;

14、[题目分析] 根据队列先进先出和栈后进先出的性质，先将非空队列中的元素出队，并压入初始为空的栈中。这时栈顶元素是队列中最后出队的元素。然后将栈中元素出栈，依次插入到初始为空的队列中。栈中第一个退栈的元素成为队列中第一个元素，最后退栈的元素（出队时第一个元素）成了最后入队的元素，从而实现了原队列的逆置。

**void** Invert(queue Q)

//Q是一个非空队列，本算法利用空栈S和已给的几个栈和队列的ADT函数，将队列Q中的元素逆置。

{makempty(S); //置空栈

**while** (!isEmpty(Q)) // 队列Q中元素出队

{value=deQueue(Q); push(S,value); }// 将出队元素压入栈中

**while**(!isEmpty(S)) //栈中元素退栈

{value=pop(S); enQueue(Q,value); }//将出栈元素入队列 Q

}//算法invert 结束

15、为运算方便，设数组下标从0开始，即数组v[0..m-1]。设每个循环队列长度（容量）为L，则循环队列的个数为n=⎡m/L⎤。为了指示每个循环队列的队头和队尾，设如下结构类型

**typedef** **struct**

{**int** f,r;

}scq;

scq q[n];

（1）初始化的核心语句

**for**(i=1;i<=n;i++) q[i].f=q[i].r=(i-1)\*L; //q[i]是全局变量

（2）入队 **int** addq(**int** i;elemtp x)

//n个循环队列共享数组v[0..m-1]和保存各循环队列首尾指针的q[n]已经定义为全局变量，数组元素为elemtp类型，本过程将元素插入到第i个循环队列中。若入队成功，返回1，否则返回队满标记0（入队失败）。

{ **if** (i<1||i>n) {printf(“队列号错误”);exit(0);}

**if** (q[i].r+1)%L+(i-1)\*L==q[i].f) {printf(“队满\n”);exit(0);}

q[i].r=(q[i].r+1)%L+(i-1)\*L; // 计算入队位置

v[q[i].r]=x; **return**(1);//元素x入队

}

（3）出队 **int** deleteq (**int** i)

// n个循环队列共享数组v[0..m-1]和保存各循环队列首尾指针的q[n]已经定义为全局变量，数组元素为elemtp类型，本过程将第i个循环队列出队。若出队成功，打印出队列元素，并返回1表示成功；若该循环队列为空，返回0表示出队失败。

{**if** (<1||>n) {printf(“队列号错误\n”);exit(0);}

**if** (q[i].r==q[i].f) {printf(“队空\n”); **return**(0);}

q[i].f=(q[i].f+1)%L+(i-1)\*L;

printf(“出队元素”,q[i].f); **return**(1);

}

（4）讨论，上述算法假定最后一个循环队列的长度也是L，否则要对最后一个循环队列作特殊处理。另外，未讨论一个循环队列满而相邻循环队列不满时，需修改个循环队列首尾指针的情况（即各循环队列长度不等）。

n个循环队列共享数组v[0..m-1]的示意图如下：

0 L-1 2L-1 3L-1 (n-1)L-1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | … |  |

第i个循环队列从下标 (i-1)L 开始，到iL-1为止。设每个循环队列均用牺牲一个单元的办法来判断队满，即为（q[i].r+1）%L+(i-1)\*L=q[i].f时，判定为队满。

16、**int** MaxValue (**int** a[],**int** n)

//设整数序列存于数组a中，共有n个，本算法求解其最大值。

{**if** (n==1) max=a[1];

**else** **if** a[n]>MaxValue(a,n-1) max=a[n];

**else** max=MaxValue(a,n-1);

**return**(max);

}

17、本题与上题类似，只是这里是同时求n个数中的最大值和最小值的递归算法。

**int** MinMaxValue(**int** A[],**int** n,**int** \*max,**int** \*min)

//一维数组A中存放有n个整型数，本算法递归的求出其中的最小数。

{**if** (n>0)

{**if**(\*max<A[n]) \*max=A[n];

**if**(\*min>A[n]) \*min=A[n];

MinMaxValue(A,n-1,max,min);

}//算法结束

[算法讨论]调用本算法的格式是MinMaxValue(arr,n,&max,&min);其中，arr是具有n个整数的一维数组，max=-32768是最大数的初值,min=32767是最小数的初值。

18、[题目分析] 求两个正整数m和n的最大公因子，本题叙述的运算方法叫辗转相除法，也称欧几里德定理。其函数定义为：

gcd(m,n)=

**int** gcd (**int** m,n)

//求正整数m和n的最大公因子的递归算法

{**if**(m<n) **return**(gcd(n,m))； //若m<n，则m和n互换

**if**(n==0) **return**(m); **else** **return**(gcd(n,m%n));

}//算法结束

使用栈，消除递归的非递归算法如下：

**int** gcd(**int** m,n)

{**int** s[max][2]; //s是栈，容量max足够大

top=1; s[top][0]=m; s[top][1]=n;

**while** (s[top][1]!=0)

**if** (s[top][0]<s[top][1]) //若m<n，则交换两数

{t=s[top][0]; s[top][0]=s[top][1]; s[top][1]=t;}

**else**{t=s[top][0]%s[top][1]; top++; s[top][0]=s[top-1][1]; s[top][1]=t; }

**return**(s[top][0]);

}//算法结束

由于是尾递归，可以不使用栈，其非递归算法如下

**int** gcd (**int** m,n)

//求正整数m和n的最大公因子

{**if** (m<n){t=m;m=n;n=t;}// 若m<n，则m和n互换

**while** (n!=0) {t=m; m=n; n=t%n;}

**return**(m);

} //算法结束

19、[题目分析]这是以读入数据的顺序为相反顺序进行累乘问题，可将读入数据放入栈中，到输入结束，将栈中数据退出进行累乘。累乘的初值为1。

**PROC** test;

CONST maxsize=32;

VAR s:ARRAY[1..maxsize] OF integer, top,sum,a:integer;

[top:=0; sum:=1;//

read(a);

**WHILE** a<>0 **DO**

[top:=top+1; s[top]:=a; read(a); ]

write(sum:5);

**WHILE** top>0 **DO**

[sum:=sum\*s[top]; top:=top-1; write(sum:5);]

**ENDP**;

20、[题目分析] 本题与第19题基本相同，不同之处就是求和，另外用C描述。

**int** test;

{**int** x,sum=0,top=0,s[];

scanf(“%d”,&x)

**while** (x<>0)

{s[++top]:=a; scanf(“%d”,&x); }

printf(sum:5);

**while** (top)

{sum+=s[top--]; printf(sum:5); }

};

21、**int** Ack(**int** m,n)

{**if** (m==0) **return**(n+1);

**else** **if**(m!=0&&n==0) **return**(Ack(m-1,1));

**else** **return**(Ack(m-1,Ack(m,m-1));

}//算法结束

（1）Ack(2,1)的计算过程

Ack(2,1)=Ack(1,Ack(2,0)) //因m<>0,n<>0而得

=Ack(1,Ack(1,1)) //因m<>0,n=0而得

=Ack(1,Ack(0,Ack(1,0))) // 因m<>0,n<>0而得

= Ack(1,Ack(0,Ack(0,1))) // 因m<>0,n=0而得

=Ack(1,Ack(0,2)) // 因m=0而得

=Ack(1,3) // 因m=0而得

=Ack(0,Ack(1,2)) //因m<>0,n<>0而得

= Ack(0,Ack(0,Ack(1,1))) //因m<>0,n<>0而得

= Ack(0,Ack(0,Ack(0,Ack(1,0)))) //因m<>0,n<>0而得

= Ack(0,Ack(0,Ack(0,Ack(0,1)))) //因m<>0,n=0而得

= Ack(0,Ack(0,Ack(0,2))) //因m=0而得

= Ack(0,Ack(0,3)) //因m=0而得

= Ack(0,4) //因n=0而得

=5 //因n=0而得

（2）**int** Ackerman( **int** m, **int** n)

{**int** akm[M][N];**int** i,j;

**for**(j=0;j<N;j++) akm[0][j];=j+1;

**for**(i=1;i<m;i++)

{akm[i][0]=akm[i-1][1];

**for**(j=1;j<N;j++)

akm[i][j]=akm[i-1][akm[i][j-1]];

}

**return**(akm[m][n]);

}//算法结束

22、[题目分析]从集合（1..n）中选出k(本题中k=2)个元素，为了避免重复和漏选，可分别求出包括1和不包括1的所有组合。即包括1时，求出集合（2..n）中取出k-1个元素的所有组合；不包括1 时，求出集合（2..n）中取出k个元素的所有组合。，将这两种情况合到一起，就是题目的解。

**int** A[],n; //设集合已存于数组A中。

**void** comb(**int** P[],**int** i,**int** k)

//从集合（1..n）中选取k（k<=n）个元素的所有组合

{**if** (k==0) printf(P);

**else** **if**(k<=n) {P[i]=A[i]; comb(P,i+1,k-1); comb(P,i+1,k); }

}//算法结束