第三章 栈和队列

从**数据结构**的角度看：它们和线性表相同

从**数据类型**的角度看：它们和线性表不同

线性表	栈	队列
Insert(L, i, x)(1<=i<=n+1)	Insert(S, n+1, x)	Insert(Q, n+1, x)
Delete(L, i)(1<=i<=n)	Delete(S, n)	Delete(Q, 1)

3.1 栈的类型定义

ADT Stack {

数据对象：D={ a_i | a_i ∈ElemSet, i=1,2,...,n, n≥0 }

数据关系：R1={ <a_{i-1}, a_i >| a_{i-1}, a_i∈D, i=2,...,n }

约定a_n 端为栈顶，a_1 端为栈底。

基本操作：

InitStack(&S)

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。

ClearStack(&S)

初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

StackEmpty(S)

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 FALE。

StackLength(S)

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。

GetTop(S, &e)

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。

Push(&S, e)

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)

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。

} ADT Stack

3.2 栈的应用举例

例一、 数制转换

十进制数 N 和其他 d 进制数的转换是计算机实现计算的基本问题，其解决方法很多，其中一个简单算法基于下列原理：

N = (N div d)×d + N mod d

(其中：div 为整除运算，mod 为求余运算)

例如： $(1348)_{10} = (2504)_8$ ，其运算过程如下：

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5

2 0 2

假设现要编制一个满足下列要求的程序：对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数。由于上述计算过程是从低位到高位顺序产生八进制数的各个数位，而打印输出，一般来说应从高位到低位进行，恰好和计算过程相反。因此，若将计算过程中得到的八进制数的各位顺序进栈，则按出栈序列打印输出的即为与输入对应的八进制数。

```
void conversion () {

// 对于输入的任意一个非负十进制整数，打印输出

// 与其等值的八进制数

InitStack(S); // 构造空栈

scanf ("%d",N);

while (N) {

Push(S, N % 8);

N = N/8;

}

while (!StackEmpty(S)) {

Pop(S,e);

printf ( "%d", e );

}

} // conversion
```

这是利用栈的后进先出特性的最简单的例子。在这个例子中，栈操作的序列是直线式的，即先一味地入栈，然后一味地出栈。也许，有的读者会提出疑问：用数组直接实现不也很简单吗？仔细分析上述算法不难看出，栈的引入简化了程序设计的问题，划分了不同的关注层次，使思考范围缩小了。而用数组不仅掩盖了问题的本质，还要分散精力去考虑数组下标增减等细节问题。

例二、 括号匹配的检验

假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，即（[] ()）或[([] [])]等为正确的格式，[(])或([())

或（( )])均为不正确的格式。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：

1. 到来的右括弧非是所“期待”的；
2. 到来的是“不速之客”；
3. 直到结束，也没有到来所“期待”的。

```
status matching(string& exp) {

// 检验表达式中所含括弧是否正确嵌套，若是，则返回

// OK，否则返回 ERROR

int state = 1;

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

swith of exp[i] {

case 左括弧: { Push(S,exp[i]); i++; break; }

case ")":

{ if (NOT StackEmpty(S) && GetTop(S) = "(")

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

else { state = 0 }

break;

}

…… }

}

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

else return ERROR;

}
```

例三、行编辑程序问题

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

每接受一个字符即存入用户数据区”不恰当。

较好的做法是，**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。允许用户输入出差错，并在发现有误时可以及时更正。**

例如，可用一个退格符“#”表示前一个字符无效；可用一个退行符“@”，表示当前行中的字符均无效。

例如，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++);
```

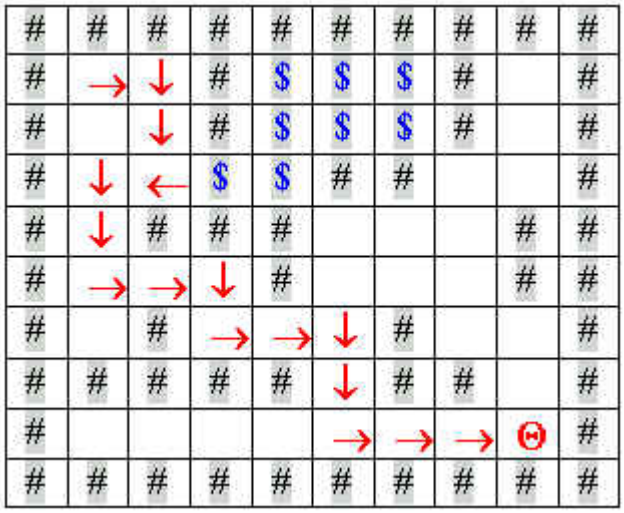
```
void LineEdit() {
// 利用字符栈 S，从终端接收一行并传送至调用过程
// 的数据区。
InitStack(S); //构造空栈 S
ch = getchar(); //从终端接收第一个字符
while (ch != EOF) { //EOF 为全文结束符
while (ch != EOF && ch != '\n') {
switch (ch) {
case '#' : Pop(S, c); break;
// 仅当栈非空时退栈
case '@': ClearStack(S); break; // 重置 S 为空栈
```

```
default : Push(S, ch); break;

// 有效字符进栈，未考虑栈满情形

}

ch = getchar(); // 从终端接收下一个字符

}

将从栈底到栈顶的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);

}
```

例四、 迷宫求解

求迷宫中从入口到出口的所有路径是一个经典的程序设计问题。由于计算机解迷宫时，通常用的是“**穷举求解**”的方法，即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止。为了保证在任何位置上都能沿原路退回，显然**需要用一个后进先出的结构来保存从入口到当前位置的路径**。因此，在求迷宫通路的算法中应用“ 栈 ”也就是自然而然的事了。

假设迷宫如下图所示：



假设“当前位置”指的是“在搜索过程中某一

时刻所在图中某个方块位置”，则求迷宫中一条路径的算法的基本思想是：若当前位置"可通"，则纳入"当前路径"，并继续朝“下一位置”探索，即切换“下一位置”为“当前位置”，如此重复直至到达出口；若当前位置“不可通”，则应顺着“来向”退回到“前一通道块”，然后朝着除“来向”之外的其他方向继续探索；若该通道块的四周四个方块均“不可通”，则应从“当前路径”上删除该通道块。所谓“下一位置”指的是“当前位置”四周四个方向（东、南、西、北）上相邻的方块。假设以栈 S 记录“当前路径”，则栈顶中存放的是“当前路径上最后一个通道块”。由此，“纳入路径”的操作即为“当前位置入栈”；“从当前路径上删除前一通道块”的操作即为“出栈”。

求迷宫中一条从入口到出口的路径的算法可简单描述如下：

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； // 纳入路径

若该位置是出口位置，则结束； // 求得路径存放在栈中

否则切换当前位置的东邻方块为新的当前位置；

}

否则 {

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；

}

} while (栈不空) ；

在此，尚需说明一点的是，所谓当前位置**可通**，指的是**未曾走到过的通道块**，即要求该方块位置不仅是通道块，而且既不在当前路径上（否则所求路径就不是简单路径），也不是曾经纳入过路径后又从路径上删除的通道块（否则只能在死胡同内转圈）。

```
typedef struct {
int ord; // 通道块在路径上的“序号”
PosType seat; // 通道块在迷宫中的“坐标位置”
int di; // 从此通道块走向下一通道块的“方向”
} SElemType; // 栈的元素类型
Status MazePath ( MazeType maze, PosType start,
PosType end ) {
// 若迷宫 maze 中从入口 start 到出口 end 的通道，
// 则求得一条存放在栈中（从栈底到栈顶），并返回
// TRUE；否则返回 FALSE
InitStack(S); curpos = start;
// 设定“当前位置”为“入口位置”
curstep = 1; // 探索第一步
do {
if (Pass (curpos)) {
// 当前位置可以通过，即是未曾走到过的通道块
FootPrint (curpos); // 留下足迹
e = ( curstep, curpos, 1 );
Push (S,e); // 加入路径
if ( curpos == end ) return (TRUE);
// 到达终点（出口）
curpos = NextPos ( curpos, 1 );
```

```
// 下一位置是当前位置的东邻
curstep++; // 探索下一步
}
else { // 当前位置不能通过
if (!StackEmpty(S)) {
Pop (S,e);
while (e.di==4 && !StackEmpty(S)) {
MarkPrint (e.seat); Pop (S,e);
// 留下不能通过的标记，并退回一步
} // while
if (e.di<4) {
e.di++; Push ( S, e);
// 换下一个方向探索
curpos = NextPos (curpos, e.di );
// 设定当前位置是该新方向上的相邻块
} // if
} // if
} // else
} while ( !StackEmpty(S) );
return (FALSE);
} // MazePath
```

例五、 表达式求值

限于二元运算符的表达式定义：

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

在计算机中，表达式可以有三种不同的标识方法

设 Exp = S1 + OP + S2

则称 OP + S1 + S2 为表达式的**前缀表示法**

称 S1 + OP + S2 为表达式的**中缀表示法**

称 S1 + S2 + OP 为表达式的**后缀表示法**

可见，它以运算符所在不同位置命名的。

例如: **Exp** = a×b+(c−d/e)×f
前缀式: +×ab×−c/def
中缀式: a×b+c−d/e×f
后缀式: ab×cde/−f×+

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它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:

- **运算符在式中出现的顺序恰为表达式的运算顺序;**
- **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;**

如何从后缀式求值？

先找运算符，后找操作数

- **如何从原表达式求得后缀式？**

分析 “ 原表达式 ” 和 “ 后缀式 ” 中的运算符:

原表达式: a+b×c-d/e×f
后缀式: abc×+de/f×-

运算符	#	(	+	-	×	/	**
优先数	-1	0	1	1	2	2	3

每个运算符的运算次序要由它之后的一个运算符来定，若当前运算符的优先数小，则暂不进行；否则立即进行。

从原表达式求得后缀式的规律为：

1. 设立操作数栈;
2. 设表达式的结束符为“#”,予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;
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。

算法描述如下：

```
void transform(char suffix[], char exp[] ) {

// 从合法的表达式字符串 exp 求得其相应的后缀式

InitStack(S); Push(S, '#' );

p = exp; ch = *p;

while (!StackEmpty(S)) {

if (!IN(ch, OP)) Pass( Suffix, ch); // 直接发送给后缀式

else {

switch (ch) {

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

case ')' : {

Pop(S, c);

while (c!= '(' )
```

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

break; }

default : {

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

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

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

break;

} // default

} // switch

} // else

if ( ch!= ' #' ) { p++; ch = *p; }

} // while

} // CrtExptree
```

表达式求值的规则：

1. 设立运算符**栈**和操作数**栈**；
2. 设表达式的结束符为“#”，予设运算符栈的栈底为“#”
3. 若当前字符是操作数，则进操作数栈；
4. 若当前运算符的优先数高于栈顶运算符，则进栈；
5. 否则，退出栈顶运算符作相应运算；
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。

算法描述如下：

```
OperandType EvaluateExpression() {

// 算术表达式求值的算符优先算法。设 OPTR 和 OPND

// 分别为运算符栈和运算数栈，OP 为运算符集合。

InitStack (OPTR); Push (OPTR, '#');

initStack (OPND); c = getchar();
```

```
while (c!= '#' || GetTop(OPTR)!= '#') {
if (!In(c, OP)) { Push((OPND, c); c = getchar(); }
// 不是运算符则进栈
else
switch (precede(GetTop(OPTR), c) {
case '<': // 栈顶元素优先权低
Push(OPTR, c); c = getchar();
break;
case '=': // 脱括号并接收下一字符
Pop(OPTR, x); c = getchar();
break;
case '>'# // 退栈并将运算结果入栈
Pop(OPTR, theta);
Pop(OPND, b); Pop(OPND, a);
Push(OPND, Operate(a, theta, b));
break;
} // switch
} // while
return GetTop(OPND);
} // EvaluateExpression
```

例六、实现递归

在高级语言编制的程序中，调用函数与被调用函数之间的链接和信息交换必须通过栈例进行。

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，需先完成三件事：

1．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；
2．为被调用函数的局部变量分配存储区；
3．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：

1．保存被调函数的计算结果；
2．释放被调函数的数据区；
3．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 ”

递归过程指向过程中占用的数据区，称之为**递归工作栈**

每一层的递归参数合成一个记录，称之为**递归工作记录**

栈顶记录指示当前层的执行情况，称之为当前活动记录

栈顶指针， 称之为当前环境指针

例如：

```
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 }
```

3.3 栈类型的实现

顺序栈

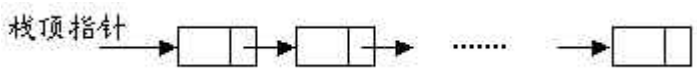
类似于线性表的顺序映象实现，指向表尾的指针可以作为栈顶指针。

```
//----- 栈的顺序存储表示 -----
#define STACK_INIT_SIZE 100; // 存储空间初始分配量
#define STACKINCREMENT 10; // 存储空间分配增量
typedef struct {
SElemType *base; // base 的初值为 NULL
SElemType *top; // 栈顶指针
int stacksize; // 当前已分配的存储空间，以元素为单位
} SqStack;

//----- 基本操作的函数原型说明 -----
Status InitStack (SqStack &S);
// 构造一个空栈 S
Status DestroyStack (SqStack &S);
// 销毁栈 S，S 不再存在
Status ClearStack (SqStack &S);
// 把 S 置为空栈
Status StackEmpty (SqStack S);
// 将栈 S 为空栈，则返回 TRUE，否则返回 FALSE
```

```
int StackLength (SqStack S);
// 返回 S 的元素个数，即栈的长度
Status GetTop (SqStack S, SElemType &e);
// 若栈不空，则用 e 返回 S 的栈顶元素，并返回 OK；
// 否则返回 ERROR
Status Push (SqStack &S, SElemType e);
// 插入元素 e 为新的栈顶元素
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
```

链栈

利用链表实现栈，注意链表中指针的方向是从栈顶到栈底。



3.4 队列的类型定义

ADT Queue {

数据对象：D＝{a_i | a_i∈ElemSet, i=1,2,...,n, n≥0}

数据关系：R1＝{ <a_{i-1},a_i > | a_{i-1}, a_i ∈D, i=2,...,n}

约定其中a_1 端为队列头，a_n 端为队列尾。

基本操作：

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。

ClearQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

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。

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。

} ADT Queue

3.5 队列类型的实现

链队列——链式映象

```
typedef struct QNode {
QElemType data;
struct QNode *next;
} QNode, *QueuePtr;
typedef struct {
QueuePtr front; // 队头指针
QueuePtr rear; // 队尾指针
} LinkQueue;
//----- 基本操作的函数原型说明 -----
Status InitQueue (LinkQueue &Q) {
// 构造一个空队列 Q
Status DestroyQueue (LinkQueue &Q) {
// 销毁队列 Q，Q 不再存在
Status ClearQueue (LinkQueue &Q) {
// 将 Q 清为空队列
Status QueueEmpty (LinkQueue Q) {
// 若队列 Q 为空队列，则返回 TRUE，否则返回 FALSE
int QueueLength (LinkQueue Q) {
// 返回 Q 的元素个数，即为队列的长度
Status GetHead (LinkQueue Q, QElemType &e) {
// 若队列不空，则用 e 返回 Q 的队头元素，并返回 OK；
// 否则返回 ERROR
Status EnQueue (LinkQueue &Q, QElemType e) {
```

```
// 插入元素 e 为 Q 的新的队尾元素

Status DeQueue (LinkQueue &Q, QElemType &e) {

// 若队列不空，则删除 Q 的队头元素，用 e 返回其值，

// 并返回 OK；否则返回 ERROR
```

循环队列——顺序映象

```
#define MAXQSIZE 100 //最大队列长度

typedef struct {

QElemType *base; // 动态分配存储空间

int front; // 头指针，若队列不空，指向队列头元素

int rear; // 尾指针，若队列不空，指向队列尾元素

// 的下一个位置

} SqQueue;
```

3.6 队列应用举例——排队问题的系统仿真

一、需求分析和规格说明

编制一个事件驱动仿真程序以模拟理发馆内一天的活动，要求输出在一天的营业时间内，到达的顾客人数、顾客在馆内的平均逗留时间和排队等候理发的平均人数以及在营业时间内空椅子的平均数。假设理发馆内设有 N 把理发椅，一天内连续营业 T 小时。在 T 小时内顾客到达的时间及顾客理发所需时间均随机生成，过了营业时间，不再有顾客进门，但仍需继续为已进入店内的顾客理发，直至最后一名顾客离开为止。

二、概要设计

1. 主程序为：

```
void BarberShop_Simulation( int chairNum, int CloseTime) {

// 理发店业务模拟，统计一天内顾客在店内逗留的平均时间

// 和顾客在等待理发时排队的平均长度以及空椅子的平均数
```

```
OpenForDay; // 初始化
while MoreEvent do {
EventDrived(OccurTime, EventType); // 事件驱动
switch (EventType) {
case 'A' : CustomerArrived; break; // 处理到达事件
case 'D' : CustomerDeparture; break; // 处理离开事件
default : Invalid;
} // switch
} // while
CloseForDay; // 计算平均逗留时间和排队的平均长度
} // BarberShop_Simulation
```

2. 数据类型为:

ADT Event (事件) {

数据对象: D = { 事件的发生时刻，事件类型 }

数据关系: R = { }

基本操作:

```
Initiate(&E,oT, eT);
```

操作结果: 产生一个发生时间为 oT，事件类型为 eT 的事件 E;

```
GetOccurTime(ev);
```

初始条件: 事件 ev 已存在，

操作结果: 返回该事件的发生时间;

```
GetEventType(ev);
```

初始条件: 事件 ev 已存在;

操作结果: 返回该事件的类型;

}

ADT customer(顾客) {

数据对象：D = { 进门时刻，理发所需时间};

数据关系：R = { }

基本操作：

Initiate(&P,aT, cT);

操作结果：产生一个进门时刻为 aT，理发时间为 cT 的顾客 P;

GetArrivalTime(Ps);

初始条件：顾客 Ps 已存在，

操作结果：返回该顾客的进门时刻;

GetCutTime(Ps);

初始条件：顾客 Ps 已存在;

操作结果：返回该顾客的理发时间;

}

ADT OrderedList(有序表) {

数据对象：是上述定义的事件的集合

数据关系：按事件的发生时刻自小至大有序

基本操作：

在线性表的操作的基础上添加一个有序表的插入

Insert(&OL, x)

初始条件：有序表 OL 已存在;

操作结果：在有序表中插入 x，并保持表的有序性;

}

ADT (数据元素为上述定义的顾客的)队列

学 习 要 点

1．掌握栈和队列这两种抽象数据类型的特点，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，即两种存储结构表示时的基本操作实现算法，特别应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．理解递归算法执行过程中栈的状态变化过程。

思考题

3.1 若按教科书 3.1.1 节中图 3.1(b)所示铁道进行车厢调度(注意：两侧铁道均为单向行驶道)，则请回答：

(1) 如果进站的车厢序列为 123，则可能得到的出站车厢序列是什么?

(2) 如果进站的车厢序列为 123456，则能否得到 435612 和 135426 的出站序列，并请说明为什么不能得到或者如何得到(即写出以‘S’表示进栈和以‘X’表示出栈的栈操作序列)。

3.2 假设以 S 和 X 分别表示入栈和出栈的操作，则初态和终态均为栈空的入栈和出栈的操作序列可以表示为仅由 S 和 X 组成的序列。称可以操作的序列为合法序列(例如，SXSX 为合法序列，SXXS 为非法序列)。试给出区分给定序列为合法序列或非法序列的一般准则，并证明:两个不同的合法(栈操作)序列(对同一输入序列)不可能得到相同的输出元素(注意：在此指的是元素实体，而不是值)序列。

3.3 试证明:若借助栈由输入序列 12…n得到的输出序列为$p_1p_2\cdots p_n$(它是输入序列的一个排列)，则在输出序列中不可能出现这样的情形：存在着i<j<k使$p_j<p_k<p_i$。

3.4 试推导求解 n 阶梵塔问题至少要执行的 move 操作的次数。

3.5 试将下列递推过程改写为递归过程。

```
status ditui(int n){

int i;

i=n;
```

```
while(i>1 ){
printf(i);
i-- ;
}
}
```

3.6 试将下列递归过程改写为非递归过程。

```
status test(int sum){
int x ;
scanf(x);
if( !x ) sum=0;
else {test(sum); }
sum+= x ;
printf(sum);
}
```

3.7 假设以顺序存储结构实现一个双向栈，即在一维数组的存储空间中存在着两个栈，它们的栈底分别设在数组的的两个端点。试编写实现这个双向栈 tws 的三个操作:初始化 inistack(tws)、入栈 push(tws,i,x) 和出栈 pop(tws,i) 的算法，其中 i 为 0 或 1，用以分别指示设在数组两端的两个栈，并讨论按过程(正/误状态变量可设为变参)或函数设计这些操作算法各有什么优缺点。

3.8 试写一个算法，识别依次读入的一个以@为结束符的字符序列是否为形如‘序列$_1$&序列$_2$’模式的字符序列。其中序列$_1$和序列$_2$中都不含字符‘&’，且序列$_2$是序列$_1$的逆序列。例如，‘a+b&b+a’是属该模式的字符序列，而‘1+3&3-1’则不是。

3.9 试写一个判别表达式中开、闭括号是否配对出现的算法。

3.10 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“]”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)。

3.11 试写一个算法，对以逆波兰式表示的表达式求值。

3.12 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点(注意不设头指针)，试编写相应的队列初始化、入队列和出队列的算法。

3.13 如果希望循环队列中的元素都能得到利用，则需设置一个标志域 tag，并以 tag 的值为 0 或 1 来区分，尾指针和头指针值相同时的队列状态是“空”还是“满”。试编写与此结构相应的入队列和出队列的算法，并从时间和空间角度讨论设标志和不设标志这两种方法的使用范围(如当循环队列容量较小而队列中每个元素占的空间较多时，那一种方法较好)。

3.14 假设将循环队列定义为：以域变量 rear 和 length 分别指示循环队列中队尾元素的位置和内含元素的个数。试给出此循环队列的队满条件，并写出相应的入队列和出队列的算法(在出队列的算法中要返回队头元素)。

3.15 假设称正读和反读都相同的字符序列为“回文”，例如，'abba' 和 'abcba'是回文，'abcde' 和 'ababab' 则不是回文。试写一个算法判别读入的一个以'@'为结束符的字符序列是否是“回文”。