第3章 栈与队列

CHAPTER 3

在线性结构中，有两种特殊的线性表，即栈和队列，它们是操作受限的线性表，插入和删除操作只能在固定端进行。具体来说，栈的操作是“后进先出”，而队列的操作则是“先进先出”。从抽象数据类型的角度看，它们处理各自元素的方法有很大差别。本章通过一个具体的项目分析，引出栈和队列的结构体定义，初始化、读取数据元素等基本操作，基于这些基本知识的应用，再解决项目的实现问题。

3.1 项目分析引入

在现实生活中，这两种操作特殊的线性表（栈和队列）会被频繁地使用，以解决具体的计算问题。例如：在使用计算机读取字符串形式的算术表达式，判断表达式是否正确并计算相应的结果时，先读取的操作数和操作符不一定要先被计算出结果，例如：3＋4×5，尽管操作符“＋”被先读取出来，也不能先计算 3＋4 的结果。为了有效临时地保存正被读取操作数和操作符，设计算法时常常会引入栈来解决运算符优先级的问题。再如，在设计停车场管理程序时，当停车场停满车辆以后，如果还有车辆需要停车服务，则需要排队等候。当有其他车辆驶离后，再按先来先服务的原则为等待的车辆提供服务。为此，在设计管理程序时，需要使用队列来实现所需的功能。

对于这两个项目来说，关于栈和队列方面的应用，需要分别完成以下功能。

（1）算术表达式的求值。①运算符优先级的定义，包括特殊运算符括号与其他运算符优先级的比较；②读取表达式，分别提取并保存其中的操作数和运算符；③遇到低优先级运算符的处理策略；④遇到高优先级运算符的处理策略。

（2）停车场服务管理程序。①停车场中停车位的标识；②车辆停车后，车牌号与停车位编号的关联；③等待停车车辆的队列管理；④车辆驶离后，停车位回收与再分配。

3.2 项目相关知识点介绍

从上面具体项目的分析中，需要再次强调两种特殊线性表（栈和队列）的本质差别：前者的操作要实现“后进先出”的功能，而后者则是“先进先出”的功能。在程序中创建栈或者队列时，可以先创建一个线性表（顺序表或者链表），然后在这个线性表上，实现“后进先出”或者“先进先出”的操作，就可以完成栈或者队列的创建任务。

3.3 栈的定义

栈(stack)是一种特殊的线性表,它的操作仅限定在线性表尾部进行(如插入、删除、读取栈顶元素等)。线性表的尾部称为栈顶(top),线性表的头部称为栈底(base)。如果栈中不含任何元素,则称该栈为空栈。假设栈 stack=$\{a_1,a_2,\cdots,a_n\}$,则栈底元素是 a_1,栈顶元素为 a_n。栈中元素的进栈次序为$\{a_1,a_2,\cdots,a_n\}$,出栈时只有栈顶元素,即从后数第一个元素 a_n,可出栈。换句话说,栈的操作是按"后进先出"的约束实现的线性表,如图 3-1 所示。因此,栈又称为后进先出的线性表。

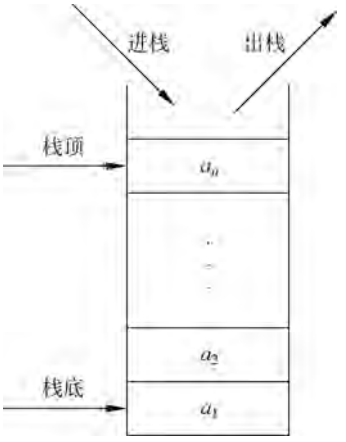

图 3-1 栈的示意图

3.3.1 顺序栈

顺序栈采用顺序存储结构来存储数据元素,即在内存中申请一组连续的地址空间依次存放数据元素,这些元素依次存储于栈底 base 到栈顶 top 指针指向的连续地址空间。使用栈的过程中。

(1) base 指针一直指向栈底元素。

(2) 当数据入栈或者出栈时,该数据存储到 top 指针指向的存储单元或者从 top 的存储单元中被读取。

(3) 当 top 和 base 相等时,表示空栈。

顺序栈的定义如下:

```
//栈的顺序存储结构
#define STACK_INIT_SIZE 10                    //存储空间初始分配量
#define STACKINCREMENT 2                      //存储空间分配增量
typedef struct SqStack{
SElemType *base;                              //栈底指针
SElemType *top;                               //栈顶指针
int stacksize;                                //栈的最大容量
}SqStack;
```

(1) 若 base 的值为 NULL,则表明栈结构不存在。top 为栈顶指针,其初始值指向栈底。每当插入新的栈顶元素时,top 增 1; 删除栈顶元素时,top 减 1。因此,栈空时,top 和 base 相等,都指向栈底。栈非空时,top 始终指向栈顶元素的上一个空位置。

(2) stacksize 指示栈可使用的最大容量,后面算法的初始化操作为顺序栈动态分配 MAXSIZE 大小的数组空间,即将 stacksize 设置为 MAXSIZE。

图 3-2 给出顺序栈中数据元素和指针之间的对应关系。显然,栈底指针 base 在初始化完成以后,始终指向栈底位置。如果初始化栈的操作失败,base=NULL。栈顶指针 top,在栈初始化以后,与栈底指针 base 同时指向栈底元素,即表示空栈。当有新元素入栈时,新元素存放到 top 指向的空间中,然后 top 加 1。当元素出栈时,top 先减 1,然后读取栈顶元素即可。所以,栈非空时,top 始终指向栈顶元素的上一个空位置。

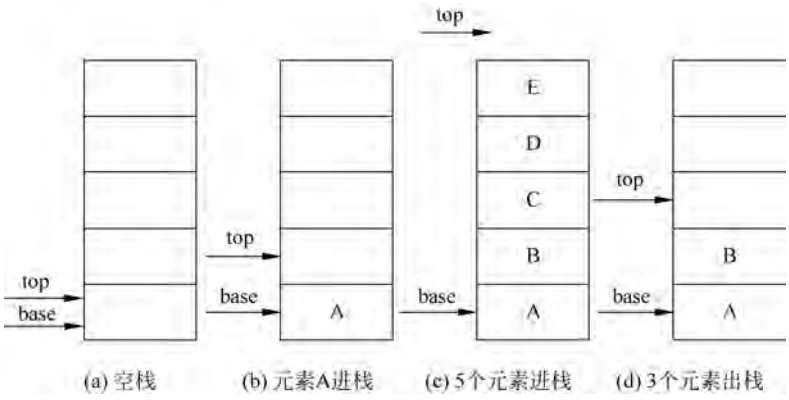

图 3-2 顺序栈的进栈和出栈示意图

另外，成员变量 stacksize 为该结构体的统计型变量，表示栈的最大容量，在顺序栈的初始化操作中，可以将其赋值为宏定义常量 MAXSIZE，然后再动态分配相应大小的地址空间给顺序栈。

1. 初始化

顺序栈的初始化操作就是为顺序栈在内存中动态分配一个预定义 MAXSIZE 大小的地址空间。

【算法 3-1】 顺序栈的初始化。

(1) 为顺序栈动态分配一个最大容量为 MAXSIZE 的数组空间，使 base 指向这段空间的起始位置，即栈底。

(2) 栈顶指针 top 初始化时等于 base，表示栈空。

(3) stacksize 设置为 MAXSIZE。

算法具体代码如下：

```
Status InitStack(SqStack &S){
 //构造一个空栈 S
 S.base = new SElemType[MAXSIZE];
 if(!S.base)
     exit(OVERFLOW);
 S.top = S.base;
 S.stacksize = MAXSIZE;
 return OK;
}
```

2. 入栈

入栈操作是指在栈顶插入一个新的元素，同时栈顶指针加 1。

【算法 3-2】 顺序栈的入栈。

(1) 判断栈是否满，若满则返回 ERROR。

(2) 将新元素压入栈顶，栈顶指针加 1。

算法具体代码如下：

```
Status Push(SqStack &S, SElemType e){
 //插入元素 e 为新的栈顶元素
```

```
 if(S.top - S.base == S.stacksize)
     return ERROR;                                    //栈满
 *(S.top++) = e;
 return OK;
}
```

3. 出栈

出栈操作是将栈顶元素移出栈，top 减 1。

【算法 3-3】 顺序栈的出栈。

(1) 判断栈是否为空，若空则返回 ERROR。

(2) 栈顶指针减 1，栈顶元素出栈。

算法具体代码如下：

```
Status Pop(SqStack &S, SElemType &e){
 //删除 S 的栈顶元素，用 e 返回值
 if(S.base == S.top)
     return ERROR;                                    //栈空
 e = *(--S.top);
 return OK;
}
```

4. 取栈顶元素

当栈非空时，此操作返回当前栈顶元素的值，栈顶指针保持不变。

【算法 3-4】 取栈顶元素。

算法具体代码如下：

```
SElemType GetTop(SqStack S){
 if(S.top!= S.base)                                   //栈不为空
     return *(S.top - 1);                             //返回栈顶元素，top 指针不变
}
```

3.3.2 链式栈

链式栈是指采用链式存储结构实现的栈。通常链式栈采用单链表来实现，如图 3-3 所示。因此，链式栈结点的结构体定义和链式栈本身的结构体定义都与链表相似，只是结构体所用名称不同。链式栈的结点被命名为 StackNode，具体定义如下：

```
typedef struct StackNode{
 SElemType data;
 struct StackNode *next;
}StackNode, *LinkStack;
```

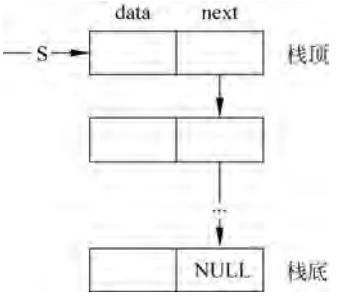

图 3-3 链式栈示意图

由于栈的入栈和出栈操作是在栈顶进行的，显然以链表的头部作为栈顶最为合理，只要设置一个指针变量 s 指向其头部即可，而不用再另设一个变量来存储链表的尾元素地址。

由于头结点在链表中不常用，可在定义单链表时去掉头结点的定义。

下面给出链式栈初始化、入栈、出栈、取栈顶元素等主要操作的实现。

1. 初始化

链式栈的初始化操作是构造一个空的链式栈，因为不需要设置头结点，所以只需要设置栈顶指针等于 NULL 即可。

【算法 3-5】 链式栈的初始化。

算法具体代码如下：

```
Status InitStack(LinkStack & S){
 //构造一个空栈 S,栈顶指针置空
 S->=NULL;
 return OK;
}
```

2. 入栈

和顺序栈的入栈操作不同的是，链式栈可以不需要判断链栈是否满的情况下，直接进行入栈操作，具体来说，只需要为待入栈元素分配一个结点空间，然后将数据存入该结点，再将其链接到链式栈的头部，同时修改头指针即可，如图 3-4 所示。

【算法 3-6】 链式栈的入栈。

(1) 为入栈元素 e 分配空间，用指针 p 指向。

(2) 将新结点数据域置为 e。

(3) 将新结点插入栈顶。

(4) 修改栈顶指针为 p。

算法具体代码如下：

```
Status Push(LinkStack &S, SElemType e){
  //在栈顶插入元素 e
  p=new StackNode;
  p->data=e;
  p->next=S;
  S=p;
  return OK;
}
```

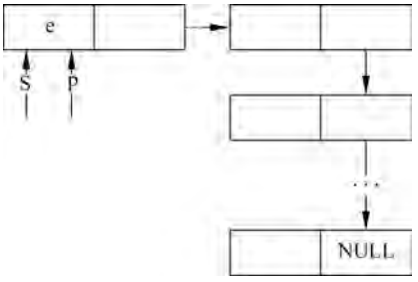

图 3-4 链式栈的入栈操作

3. 出栈

链式栈在出栈操作之前，和顺序栈一样，需要先判断栈是否为空，但是，链式栈在出栈后需要释放刚才出栈元素的地址空间，即原来栈的栈顶空间，如图 3-5 所示。

【算法 3-7】 链式栈的出栈。

(1) 判断栈是否为空，若空则返回 ERROR。

(2) 将栈顶元素赋给 e。

(3) 用指针 p 临时指向栈顶元素的空间，以备释放。

(4) 修改栈顶指针，指向新的栈顶元素。

(5) 释放原栈顶元素的空间。

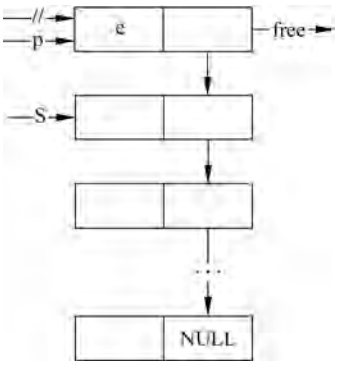

图 3-5　链式栈的出栈过程

算法具体代码如下：

```
Status Pop(LinkStack &S, SElemType &e){
 //删除 S 的栈顶元素,用 e 返回其值
 LinkStack p;
 p = new StackNode;
 if(S == NULL) return ERROR;
 e = S -> data;
 p = S;
 S = S -> next;
 delete p;
 return OK;
}
```

4. 取栈顶元素

与顺序栈一样，当链式栈非空时，此操作返回当前栈顶元素的值，同时保持栈顶指针 S 不变。

【算法 3-8】 取链式栈的栈顶元素。

算法具体代码如下：

```
SElemType GetTop(LinkStack S){
 //返回 S 的栈顶元素,不修改栈顶指针
 if(S!= NULL)
     return S -> data;
}
```

3.3.3　栈与递归

在程序设计过程中，递归算法的结构常常比非递归算法的结构更简洁和清晰并容易设计，尤其是待解决问题比较复杂时，采用递归算法将更容易设计出有效率的算法。递归算法的设计需要使用栈来实现。这是因为，递归算法通常把一个大型的复杂问题的描述和求解，逐步简化成一系列简单的问题描述和求解。而栈的使用可有效地保存和使用计算过程中的中间结果。为了帮助学习者理解递归算法并提高其设计递归算法的能力，本节将介绍递归算法设计思路，同时详细介绍：在递归算法执行过程中，被中断执行的程序是如何存储于栈

中；这些程序又在何时被弹出栈，继续执行。

1. 递归算法适用情况

递归算法(函数)是指，该算法(函数)在定义的过程中又调用了自己(即定义中需要使用自身的函数功能)。通常，递归算法适用于在以下三种情况。

1) 算法的定义是递归的

有很多数学函数是递归定义的，如大家熟悉的式(3-1)表示的阶乘函数和式(3-2)表示的二阶 Fibonacci 数列等。

$$\mathrm{Fact}(n)=\begin{cases}1, & n=0\\ n\mathrm{Fact}(n-1), & n>0\end{cases} \tag{3-1}$$

$$\mathrm{Fib}(n)=\begin{cases}1, & n=1,2\\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2), & \text{其他}\end{cases} \tag{3-2}$$

对于式(3-1)中的阶乘函数，可以使用递归函数来求解，具体代码如下：

```
long Fact(long n){
 if(n == 0) return 1;
 else return n * Fact(n - 1);
}
```

图 3-6 给出了主程序调用递归函数 Fact(4)的执行过程。在递归过程中，else 语句分别以实际参数 $n=\{3, 2, 1, 0\}$调用自身，而最后一次递归调用，因参数 n 为 0 执行了递归函数的出口程序(if 语句)，递归终止，程序逐步返回，返回时依次计算 1×1、2×1、3×2、4×6，最后将计算结果 24 返回给主程序。

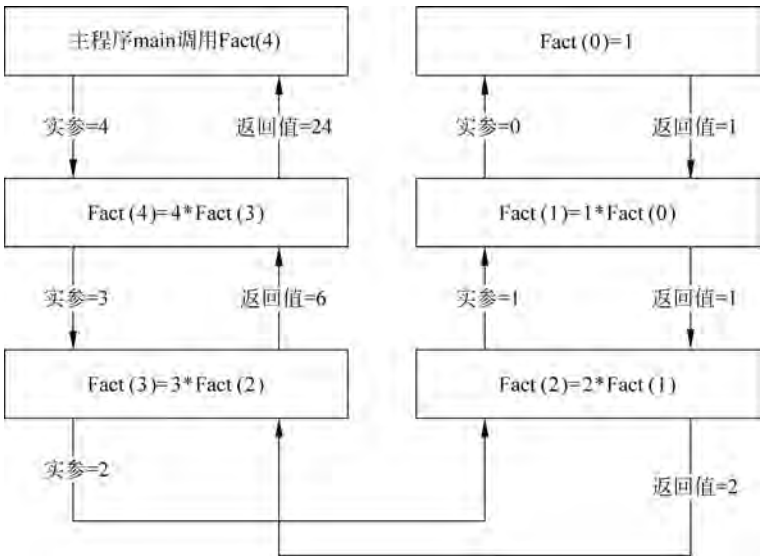

图 3-6　递归求解 4!的过程

类似地，可写出 Fibonacci 数列的递归程序。

```
long Fib(long n){
 if(n == 1||n == 2) return 1;                    //递归终止条件
```

```
 else return Fib(n-1) + Fib(n-2);                    //递归过程
}
```

对于这种类似的递归问题，可以将原来的复杂问题分解成几个逐级简化的且解法相同或者类似的子问题来进行求解，这一过程被称为递归求解，这种分解－求解的策略叫“分治法”。采取分治法进行递归求解的问题需要满足以下 3 个条件。

(1) 能将一个问题转换成一个新的子问题，而且子问题与原问题的解法相同或相似，它们之间的差别在于所传递的参数不同，参数的变化将逐步趋近于可直接求解。

(2) 可以通过上述问题转换实现原有的复杂问题简化，直到子问题可被求解。

(3) 递归函数体中必须有一个明确的递归出口，否则程序将陷入类似于死循环的情况。

2) 数据结构是递归的

基于某些数据结构本身所具有的递归特性，那么基于这类数据结构的一些操作方法(或者函数)可采用递归形式的描述。常见的递归形式的数据结构包含之前所学的“链表”，其结点 LNode 的定义由数据域 data 和指针域 next 组成，而指针 next 是一种指向 LNode 类型的指针，即 LNode 的定义中又用到了自身结构体来定义其成员 next(LNode * next)。第 5 章和第 6 章要讲解的树形结构和图状结构都是非常重要的递归型数据结构。

对于递归型的数据结构来说，如果采用递归算法实现其相应的操作功能，将带来很大的便利。例如，链表结点的遍历输出函数可以采用递归形式的算法。

算法 3-8 是从链表头部开始，依次遍历所有链表结点并打印输出每个结点的 data 域的递归算法。在递归函数被执行时，形式参数 p 得到单链表的首元结点地址；在递归过程中，p 逐步指向每个结点的直接后继结点，同时输出 p 指向结点的 data 域的值，直到链表尾部(p 为 NULL)时，递归过程结束。显然，这个问题的求解过程描述满足上述“分治法”的 3 个基本条件，因此，可以采用递归算法来设计求解过程。

【算法 3-9】 遍历输出链表中各个结点的递归算法。

(1) 如果 p 为 NULL，递归结束返回。

(2) 否则输出 p-> data，p 指向后继结点，即 p＝p-> next。

算法具体代码如下：

```
void TraverseList(LinkList p){
 if(p == NULL) return ;                              //递归结束
 else{
     cout << p -> data << endl;                      //输出当前结点 data
     TraverseList(p -> next);                        //递归过程
 }
}
```

后面章节要介绍的广义表、二叉树、图等经典数据结构也都具有结构递归特性，基于这些数据结构的算法都可采用递归算法。

3) 问题的解法是递归的

有一类非常复杂的问题，问题的复杂程度与问题的规模呈指数级增长，而且问题规模较小时，解法比较简单。这样的一类问题，使用递归算法求解要比迭代求解更简单，如汉诺(Hanoi)塔问题、迷宫问题、八皇后(8-queen)问题等。

【例 3-1】 n 阶汉诺塔问题。

假设有 3 个塔座，分别被命名为 A、B、C。开始时，在塔座 A 上依次叠放直径大小各不相同的 n 个圆盘。其中，大的在下小的在上，编号从小到大依次为 1，2，3，…，n，图 3-7 所示为 $n=4$。现要求将塔座 A 的 n 个圆盘移至塔座 C 上，并仍按同样顺序叠放，圆盘移动时必须遵循下列规则。

(1) 每次只能移动一个盘子。

(2) 圆盘可以放在 A、B、C 中任一个塔座上。

(3) 任何时刻都不能将一个较大的圆盘压在较小的圆盘上。

显然，当 n 值很小时，盘子的移动问题可以很容易地被解决，但是随着盘子数量的增多，盘子移动问题将变得非常复杂。由于 n 个盘子移动问题的解法可被借鉴于 $n+1$ 个盘子的移动解法，因此可以使用“分治法”来解决这一问题。先来考虑最简单的情况，塔座 A 上最初的盘子数量为 $n=1$ 时，只要将编号为 1 的盘子从塔底 A 直接移到塔座 C 上即可；如果盘子数量增多($n\geqslant 2$)，则可以执行以下 3 个步骤。

(1) 将塔座 A 上的 $n-1$ 个盘子移到塔座 B 上，用塔座 C 过渡；

(2) 将塔座 A 上最后一个盘子直接移动到塔座 C 上；

(3) 将塔座 B 上的 $n-1$ 个盘子移动到塔座 C 上，用塔座 A 过渡。

具体移动过程如图 3-7 所示，图中 $n=4$。从上述的解法来看，将 $n-1$ 个盘子从一个塔座移动到另一个塔座的问题与原问题有很多相似的特征，例如：柱子的数量相同，移动规则相同。那么，不同之处在于前者的问题规模要小，而且 3 个塔座在移动过程中“角色”要互换一下。因此，可以使用递归的方法求解。

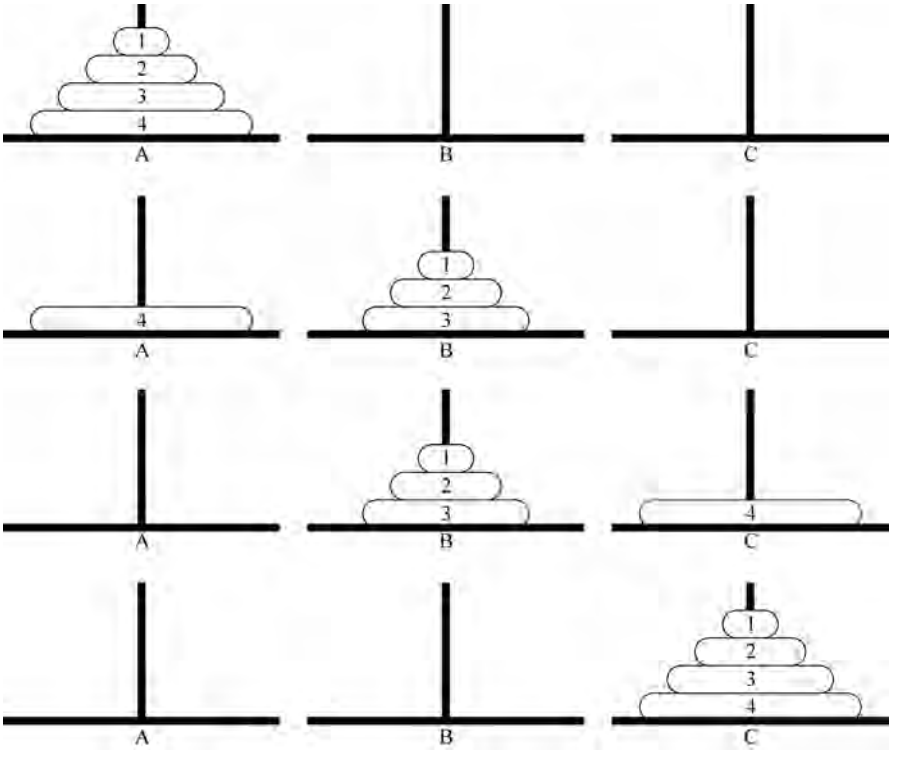

图 3-7 汉诺塔问题

为了实现汉诺塔的移动圆盘方案，即 Hanoi()函数，需要先定义移动一块盘子的基本操作函数：move(A,n,C)，三个形式参数的含义分别是：A 为源柱，n 为盘子编号，C 为目标柱。另外，为了对移动的步数进行计数，在 Hanoi()函数之前设置一个全局变量 m，其初始值为 0，move()函数代码如下所示：

```
int m = 0;
void move(char A, int n, char C){
 cout <<++m <<", "<< n <<", "<< A <<", "<< C << endl;
}
```

【算法 3-10】 汉诺塔问题的递归算法。

(1) 如果 $n=1$，则直接将编号为 1 的圆盘直接从 A 移动到 C，函数结束(出口程序)。

(2) 否则调用递归函数，将 A 上编号 1 到 $n-1$ 的圆盘移动到 B 上，以 C 作为辅助塔；直接将编号为 n 的圆盘从 A 移动到 C；调用递归函数，将 B 上编号 1 到 $n-1$ 的圆盘移动到 C 上，以 A 作为辅助塔。

算法具体实现代码如下：

```
void Hanoi(int n, char A, char B, char C){
 if(n == 1) move(A,1,C);
 else{
     Hanoi(n - 1,A,C,B);
     move(A,n,C);
     Hanoi(n - 1,B,A,C);
 }
}
```

2. 递归算法分析

关于递归算法分析，主要包含两方面的分析：时间复杂度分析和空间复杂度分析。

1) 时间复杂度的分析

在算法分析中，当一个算法中包含递归调用时，其时间复杂度的分析可转化为一个递归方程求解。实际上，这个问题是数学上求解渐近阶的问题，迭代法是求解递归方程的一种常用方法，其基本步骤是迭代地展开递归方法的右端，使之成为一个非递归的和式，然后通过对和式的估计来达到对方程左端的估计。

下面以求解阶乘的递归函数 Fact(n)为例，说明如何通过迭代法来求解递归方程的时间复杂度。

设 Fact(n)的执行时间是 $T(n)$。此递归函数中语句“if(n==0) return 1;”的执行时间是 $O(1)$，递归调用 Fact($n-1$)的执行时间是 $T(n-1)$，所有 y 语句“else return n * Fact(n-1);”的执行时间是 $O(1)+T(n-1)$。求得递归方程的解为 $T(n)=O(n)$。采用这种方法计算斐波那契(Fibonacci)数列和汉诺塔问题递归算法的时间复杂度是 $O(2^n)$。

2) 空间复杂度的分析

递归函数在执行时，系统需设立一个“递归工作栈”存储每一层递归所需的信息，此工作栈是递归函数执行的辅助空间，因此分析递归算法空间复杂度需要分析工作栈的大小。所以，空间复杂度函数为：$S(n)=O(f(n))$，其中 $f(n)$为递归工作栈中工作记录的个数与问题规模 n 的函数关系。根据这种分析方法得到阶乘问题、斐波那契数列问题、汉诺塔的递

归算法空间复杂度函数为 $O(n)$。

通过对递归函数的详细分析，可以发现，在执行递归函数时需要系统提供一个隐式栈的数据结构，以存储递归调用前的程序以及状态。对于结构较为清晰的递归函数来说，可以通过程序的基本语句模拟递归函数的执行过程。这一过程中，需要将递归工作栈的状态变化用程序的基本语句直接写出，汇总之后即可得到相应的非递归算法。

总结，这种利用栈将递归函数改写成非递归函数的步骤如下。

(1) 设置一个工作栈存放递归工作记录。

(2) 进入非递归调用入口，将调用程序传来的实际参数和返回地址入栈。

(3) 进入递归调用入口，当不满足递归结束条件时，逐层递归，将实参、返回地址及局部变量入栈，这一过程可用循环语句来实现。

(4) 递归结束条件满足，将到达递归出口的给定常数作为当前的函数值。

(5) 返回处理。在栈不空的情况下，反复退出栈顶记录，根据记录中的返回地址进行题意规定的操作，即逐层计算当前函数值，直到栈空为止。

通过上述步骤，可以将任何递归函数改写成非递归函数。但改写后的非递归函数结构比较混乱，可读性不好，有的函数还需要经过一系列优化。由于递归函数结构清晰，程序易读，而且其正确性容易得到证明。由于递归问题编程时，系统隐式递归栈的存在，用户不需要自己管理递归工作栈，因此，使用递归函数将给用户编制程序和调试程序带来很大的方便。

3.4 队列的定义

3.4.1 队列的定义和特点

队列(queue)也是一种特殊的线性表，它的操作限定于：只允许在表的一端进行插入元素的操作，而另一端只能进行删除元素的操作。这一原理与日常生活中的排队现象是一致的，越早进入队列的元素离开得就越早，反之亦然。在队列中，可插入元素的一端称为队尾(rear)，可删除元素的一端称为队头(front)。例如，队列 $\{a_1, a_2, \cdots, a_n\}$ 中，a_1 就是队头元素，a_n 是队尾元素。队列中元素的进队列次序为 $\{a_1, a_2, \cdots, a_n\}$，退出队列也只能按照这个次序依次进行，图 3-8 是队列的示意图。

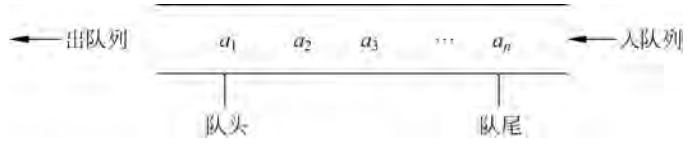

图 3-8 队列的示意图

3.4.2 队列的基本操作

队列的基本操作主要包括队列的初始化、元素入队、元素出队以及取队头元素等。函数名称，形式参数，初始条件，操作结果如下所述。

(1) 函数 InitQueue(&Q)的操作结果：构造一个空队列 Q。

(2) 函数 DestroyQueue(&Q)的初始条件：队列 Q 已存在；操作结果：队列 Q 被销毁。

(3) 函数 EnQueue(&Q，e)的初始条件：队列 Q 已存在；操作过程：将 e 值赋给队尾元素，队尾指针指向下一个空位置；操作结果：元素 e 插入 Q 的队尾。

(4) 函数 DeQueue(&Q，&e)的初始条件：Q 为非空队列；操作过程：将对头元素的值赋给 e，队头指针指向下一个位置；操作结果：找出 Q 的队头元素，用 e 接受其值。

(5) 函数 GetHead(Q)的初始条件：Q 为非空队列；操作结果：返回 Q 的队头元素。

3.4.3 循环队列

队列的顺序存储结构，采用一组地址连续的存储单元依次存放从队列头到队列尾的元素，这一点与顺序栈很相似，但是顺序队列须设置两个整型的游标变量 front 和 rear 分别指示队列头元素和队列尾元素的位置。队列的顺序存储结构如下：

```
#define MAXQSIZE 100
typedef struct{
 QElemType * base;
 int front;
 int rear;
}SqQueue;
```

队列的初始化是在内存中申请一片连续地址空间，以创建一个空队列，空间成功申请以后，令游标 front＝rear＝0 即可。

如果需要插入新的元素入队列时，将新元素存储于尾指针指向的单元格，然后尾指针 rear 加 1；如果要删除元素出队列时，只需要令头指针 front 加 1 即可，这样游标 front 刚才指向的位置，可被再次写入数据。由此可见，在一个非空的队列中，头指针始终指向队列头元素，而队尾指针始终指向队尾元素的下一个位置，如图 3-9 所示。

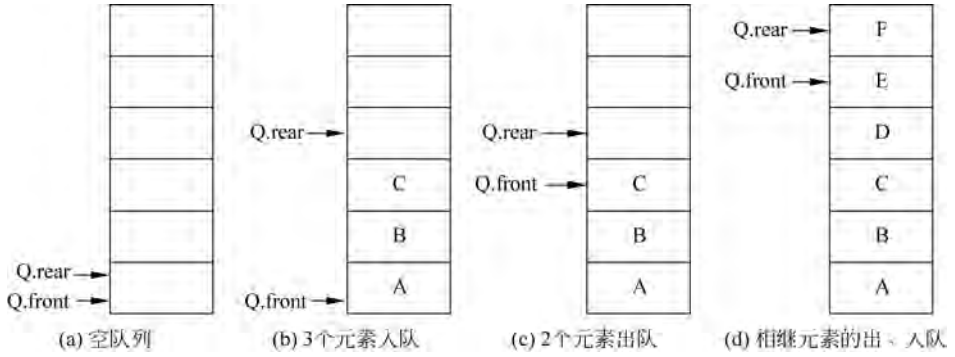

图 3-9 顺序队列初始化和数据操作后的游标状态

在顺序队列进行一系列增删数据操作以后，会出现一个非常严重的问题。以图 3-9 为例，当前队列容量为 6 个元素，当队列处于图 3-9(d)所示的状态时，队列虽然未满，但是队列不能再插入新元素入队尾，因为队尾指针 Q. rear 已经指向顺序队列的上界。这一现象被称为“假溢出”现象，描述的是虽然队列未满，但队尾指针已到边界，如果再插入数据，则队尾指针越界的现象。产生这种现象的根本原因在于，在顺序队列上增删数据操作时，队尾指针 Q. rear 和队头指针 Q. front 是单向的(只增不减)。

为了有效解决这种“假溢出”问题，一种可行的办法是在逻辑上改变顺序队列的结构，即将顺序队列转变为一个环状数据结构，即循环队列，如图 3-10 所示。

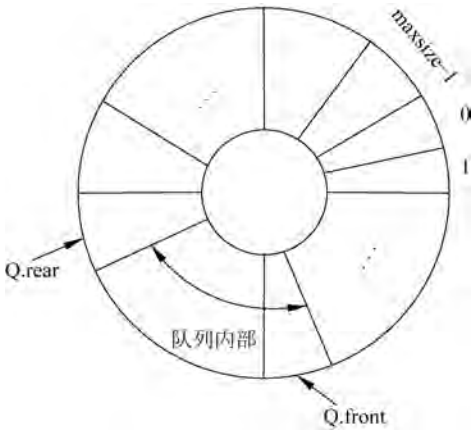

图 3-10 循环队列结构示意图

在循环队列中增删数据元素时，其队尾和队头指针分别要做“加 1 取模”的运算。求模时用到的除数为循环队列的容量(maxsize)。显然，通过这种运算方式，可以不用改变顺序队列的存储结构，队列的头指针和尾指针依然可以在顺序存储空间内以头尾衔接的方式循环移动，从而避免了“假溢出”的问题。

对于循环队列来说，如何判断队列是满状态还是空状态的问题，通常有以下两种处理方式。

(1) 队列中预留一个存储空间，不能用于存储数据。具体来说，假设循环队列容量为 m 时，当 $m-1$ 个元素入队以后，就判定循环队列为满队状态，不能再有新元素入队。那么，判断循环队列为空状态的条件是：队头和队尾指针相同；判断循环队列为满状态的条件是：队尾指针在“加 1 求模”运算后等于队头指针。因此，用循环队列的队尾和队头指针运算表达式表示循环队列为空状态的条件为 Q. front==Q. rear，表示循环队列为满状态的条件为(Q. rear+1)%MAXQSIZE==Q. front。

(2) 另设一个标志位 flag，以区别循环队列处于“空状态”还是“满状态”。当初始化循环队列时，可以设置 flag=-1，以表示空的循环队列；当有元素进入循环队列时，flag 被设置为 0；当元素入队以后，如果队尾指针加 1 取模后等于队头指针，即满足队满条件((Q. rear+1)%MAXQSIZE==Q. front)时，先允许元素入队，同时将 flag 设置为 1，以表示队列已满，这时循环队列不允许其他元素入队；当元素出队操作时，如果队头指针加 1 取模后等于队尾指针，即满足队空条件((Q. front+1)%MAXQSIZE==Q. rear)时，先允许元素出队，同时将 flag 设置为-1，表示循环队列已经为空。

下面给出第一种策略下循环队列的主要操作(初始化，求长度，入队以及出队等)。

1. 初始化

循环队列的初始化操作就是动态分配一个预定义大小为 MAXQSIZE 的数组空间。

【算法 3-11】 循环队列的初始化。

(1) 为循环队列申请一个最大容量为 MAXQSIZE 的数组空间，指针 base 指向数组空间的首地址。

(2) 头指针和尾指针设置为 0，初始化的循环队列为空状态。

算法具体代码如下：

```
StatusInitQueue(SqQueue &Q){
 Q.base = new QElemType[MAXQSIZE];
 if(!Q.base) exit(OVERFLOW);
 Q.front = Q.rear = 0;
 return OK;
}
```

2. 求队列长度

对于循环队列来说，尾指针和头指针的差值可能为负数，所以需要将差值加上 MAXQSIZE，然后再用 MAXQSIZE 求模。

【算法 3-12】 求循环队列的长度。

算法的具体代码如下：

```
int QueueLength(SqQueue Q){
 return (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
}
```

3. 入队

入队操作是指在循环队列的队尾插入一个新的元素。

【算法 3-13】 循环队列的入队。

(1) 判断循环队列是否为满状态，若“是”，则返回 ERROR，以结束。

(2) 将新元素插入队尾的位置。

(3) 队尾指针做加 1 求模运算，即(Q.rear+1)%MAXQSIZE。

算法具体代码如下：

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

4. 出队

出队操作是将循环队列的队头指向的元素删除。

【算法 3-14】 循环队列的出队。

(1) 判断队列是否为空状态，若“是”，则返回 ERROR，以结束。

(2) 保存队头元素为临时变量。

(3) 队头指针做加 1 求模运算，即(Q.front+1)%MAXQSIZE。

算法具体代码如下：

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

5. 取出队头元素

当队列为非空状态时，此操作返回当前队头指针指向元素的值，且队头指针保持不变。

【算法 3-15】 取循环队列的队头元素。

算法具体代码如下：

```
SElemType GetHead(SqQueue Q){
 if(Q.front!= Q.rear)
     return Q.base[Q.front];
}
```

由上述分析可见，如果用户的应用程序中设有循环队列，则必须为它分配一个最大容量MAXSIZE；如果这一容量无法满足用户的需求，则应该采用链式队列的数据结构，以避免反复申请更大容量内存空间的操作。

3.4.4 链式队列

链式队列是指采用链式存储结构的队列。通常使用单链表来作为链式队列物理存储结构，如图 3-11 所示。与顺序队列一样，链式队列也需要两个指针(rear 和 front)，分别指向队尾和队头。为了操作上的便利，可以为整个链式队列添加一个头结点，并且设置队头指针始终指向头结点，而不是队头元素。

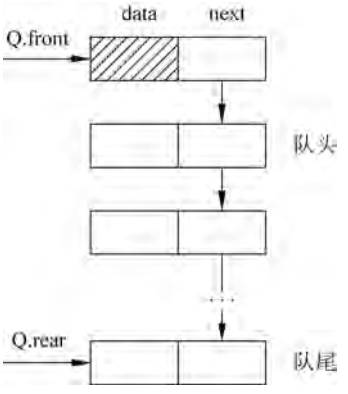

图 3-11 链式队列示意图

```
typedef struct QNode{
 QElemType data;
 struct QNode * next;
}QNode, * QueuePtr;
typedef struct{
 QueuePtr front;
 QueuePtr rear;
}LinkQueue;
```

下面给出链式队列的基本操作，包含链式队列的初始化、入队、出队和取队头元素操作的实现。

1. 初始化

链式队列的初始化操作主要包含两个步骤：构造一个只有头结点的空队列；将队尾指针和队头指针指向头结点，如图 3-12(a)所示。

【算法 3-16】 链式队列的初始化。

(1) 生成新结点作为头结点，头结点的指针域 next 置空。

(2) 队尾指针和队头指针指向该结点。

算法具体代码如下：

```
Status InitQueue(LinkQueue &Q){
 Q.front = Q.rear = new QNode;
 Q.front -> next = NULL;
 return OK;
}
```

2. 入队

与循环队列的入队操作不同，链式队列在入队操作之前不需要判断队列是否为满状态，

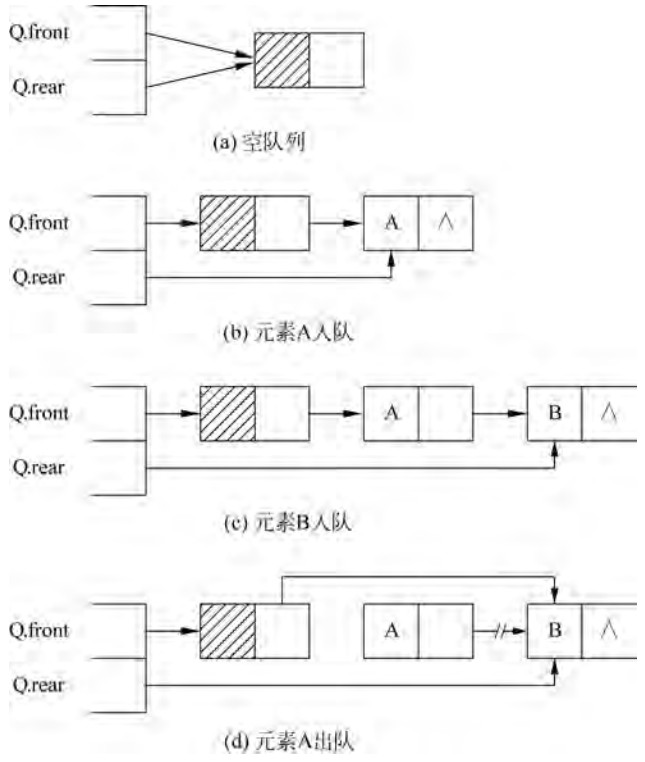

图 3-12　队列运算过程中的指针变化情况

只需要为入队元素动态分配一个结点空间，然后将数据存储于结点的 data 域，如图 3-12(b)和图 3-12(c)所示。

【算法 3-17】 链式队列的入队。

(1) 为入队元素分配结点空间，用指针 p 指向。

(2) 将 e 存储于新结点的数据域。

(3) 将新结点插入队尾。

(4) 修改队尾指针为 p。

算法具体代码如下：

```
Status EnQueue(LinkQueue &Q, QElemType e){
QueuePtr p;
 p = new QNode;
 p->data = e;
 p->next = NULL;
 Q.rear->next = p;
 Q.rear = p;
 return OK;
}
```

3. 出队

和所有队列一样，链式队列在出队前需要先判断队列是否为空状态，但是，链式队列出队操作的特别之处在于，在队头元素出队以后需要释放出队元素所占空间，如图 3-12(d)所示。

【算法 3-18】 链式队列的出队。

(1) 判断队列是否为空状态,若空则返回 ERROR,出队函数结束。

(2) 临时保存队头元素的地址,以备接下来的释放空间操作。

(3) 修改头结点的指针域,指向下一个结点。

(4) 判断出队元素是否为最后一个元素,若是,则将队尾指针重新赋值,指向头结点。

(5) 释放出队元素的地址空间。

算法具体代码如下:

```
Status DeQueue(LinkQueue &Q, QElemType &e){
    QueuePtr p;
    if(Q.front == Q.rear)
       return ERROR;
    p = Q.front -> next;
    e = p -> data;
    Q.front -> next = p -> next;
    if(Q.rear == p)
       Q.rear = Q.front;
    delete p;
    return OK;
}
```

在执行链式队列的出队函数时,还要考虑一个特殊情况,即当队列中最后一个元素被删除后,队列尾指针将没有合适的结点可指向,因此需对队尾指针重新赋值,以指向头结点(链式队列的初始化状态)。

4. 取队头元素

与循环队列一样,当队列处于非空状态时,取队头元素的操作将返回当前队头指针所指向元素的值,队头指针不发生变化。

【算法 3-19】 取队头元素。

算法具体代码如下:

```
QElemType GetHead(LinkQueue Q){
 if(Q.front!= Q.rear)
     return Q.front -> next -> data;
}
```

3.5 项目实现

本节的项目实现共分两个部分:算术表达式求值和停车场管理小程序。

1. 算术表达式求值

一个表达式主要由操作数和运算符组成,这里将括号这种界限符视为一种特殊运算符来处理。运算法则可定义为:先计算括号内,再计算括号外;先乘除,后加减。

基于上面总结的基本运算法则,将运算符之间的优先级关系定义出来,并存储于一张二维表中,详见表 3-1。

表 3-1 运算符之间的优先级关系

运算符 1	运算符 2						
	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

表达式求值的算法步骤。

(1) 初始化用于存储操作数和运算符的数据栈：OPTER 和 OPND，并将表达式起始符号“#”压入 OPTR 栈。

(2) 扫描表达式，读取的操作数或者运算符存储变量 ch，如果表达式没扫描完或者 OPTR 的栈顶元素不是“#”，则循环执行下面的操作：①若 ch 是操作数，则压入 OPND 栈，再继续扫描字符串。②若 ch 是运算符，则比较 OPTR 栈顶元素和 ch 之间的优先级。如果小于，则 ch 压入 OPTR 栈，继续扫描表达式；如果大于，则弹出 OPTR 栈顶的运算符，从 OPND 栈弹出两个操作数，进行相应的计算，结果再压入 OPND 栈中；如果等于，则表明 OPTR 的栈顶元素为“(”，且 ch 为“)”，这时直接弹出栈顶元素与 ch 匹配，然后继续扫描。

(3) 如果表达式没有错误，直接返回 OPND 栈顶元素作为表达式的计算结果。

表达式求值的函数代码实现如下：

```
char Expression(){
 InitStack(OPND);
 InitStack(OPTR);
 Push(OPTR, '#');
 cin >> ch;
 while(ch!= '#'||GetTop(OPTR)!= '#'){
     if(!In(ch)){
         Push(OPND,ch);
         cin >> ch;
     }
     else{
         switch(Precede(GetTop(OPTR), ch)){
             case: '<':
                 Push(OPTR, ch);
                 cin >> ch;
                 break;
             case: '>':
                 Pop(OPTR, theta);
                 Pop(OPND, b);
                 Pop(OPND, a);
                 Push(OPND, Operate(a, theta, b));
                 break;
```

```
            case: '=':
                Pop(OPTR, x);
                cin >> ch;
                break;
        }
    }
 return GetTop(OPND);
}
```

2. 停车场管理算法

注意,停车场管理小程序只关注于与队列操作相关的部分,只要解决当已满的停车场有车辆驶离,如何为其他等待的车辆提供停车服务的问题。

停车场管理算法步骤如下所述。

(1) 为停车场的每个停车位编号,并统计停车位的数量。

(2) 在停车场运营中,实时统计空余停车位。

(3) 当停车场没有空位时,为需要提供停车服务的车辆建立并管理等待队列。

(4) 当停车场有空位时,为等待服务的车辆合理分配停车位置。

停车函数的代码实现如下:

```
#define MAXSIZE 255
void Parking(){
 string str;
 int i, count = 0;
 string park[MAXSIZE];
 InitQueue(waiting);
 cin >> str;
 while(str!= '#'){
    if(!IsParked(str)){                          //进入停车场
        if(count!= 255){
            i = search(park);                    //在 park 中找空位,返回第一个空位的下标
            park[i] = str;
            count++;
        }
        else EuQueue(waiting, str);
    }
    else{                                        //出停车场
        i = search(park, str);
        park[i] = "";
        count--;
        if(!IsEmpty(waiting))
            DeQueue(waiting, str);
    }
 }
 Print(park);
}
```

在算法中,通过重载方式 search()函数可以实现两种功能。

(1) search(park)的功能是遍历整个停车场,直到出现第一个空位停止,并将位置编号

返回，如果停车场已满，则返回－1。

(2) search(park, str)的功能是在停车场中查找车牌为 str 的车辆，如果找到，则表明该车准备驶离，否则，表明该车准备停车。

3.6 习题

1. 选择题

(1) 元素 a,b,c,d,e 依次进栈，则出栈次序不可能的是(　　)。

A. e,d,c,b,a　　B. b,a,e,d,c

C. d,c,a,b,e　　D. b,c,e,d,a

(2) 如果一个栈的入栈序列为 1,2,3,…, n，输出序列为 $a_1,a_2,a_3,\cdots,a_n$，若 $a_1=n$，则 pi 为(　　)。

A. i　　B. $n-i$

C. $n-i+1$　　D. 不确定

(3) 循环队列 Q 的大小为 n，f 和 r 分别为队头和队尾指针，计算队列中元素个数的公式为(　　)。

A. $r-f$　　B. $(n+f-r)\%n$

C. $n+r-f$　　D. $(n+r-f)\%n$

(4) 一个递归算法必须包含(　　)。

A. 递归部分　　B. 终止条件和递归部分

C. 迭代部分　　D. 终止条件和迭代部分

(5) 判定一个循环队列 QU(最多元素为 m0)为满队列的条件是(　　)。

A. QU-> rear-QU-> front==m0

B. QU-> rear-QU-> front－1==m0

C. QU-> front==QU-> rear

D. QU-> front==QU-> rear+1

2. 简答题

(1) 说明线性表、栈与队的异同点。

(2) 设有编号为 1、2、3、4 的四辆列车，顺序进入一个栈式结构的车站，具体写出这四辆列车开出车站的所有可能的顺序。

3. 程序阅读题

(1) 写出下列程序段的输出结果(队列中的元素类型 QElem Type 为 char)。

```
void main( ){
 Queue Q; Init Queue (Q);
Char x = 'e'; y = 'c';
EnQueue (Q,'h'); EnQueue (Q,'r'); EnQueue (Q, y);
DeQueue (Q,x); EnQueue (Q,x);
DeQueue (Q,x); EnQueue (Q,'a');
while(!QueueEmpty(Q)){ DeQueue (Q,y);printf(y); };
Printf(x);
}
```

(2) 简述以下算法的功能(栈和队列的元素类型均为 int)。

```
void algo3(Queue &Q){
Stack S; int d;
InitStack(S);
while(!QueueEmpty(Q)){
DeQueue (Q,d); Push(S,d);
};
while(!StackEmpty(S)){
Pop(S,d); EnQueue (Q,d);
}}
```