第3章 栈和队列

栈和队列都是线性表的特殊形式，普遍存在于日常生活和计算机领域。

在日常生活中，可以看到许多栈和队列的应用。抽屉中，要想拿到先放进去的东西，必须把后放进去的东西先拿出来，最方便使用的是抽屉顶上的东西；米桶中的米，先倒进去的最后才能用到，最后放进去的先被用到。这些是栈的应用。学校食堂在学生中午下课后的高峰用饭时间，各个卖饭菜的窗口都会排队，否则就会拥挤；大型超市等待付款的顾客、银行各柜台前等候存取钱的顾客都会形成排队等待的现象，这些就是队列。

在计算机领域，栈和队列的使用更为广泛，如递归调用、中断、子程序调用中栈的使用；网络中在同一链路上传输的数据报文形成一个队列，网络中共享的打印机中所有申请打印的作业在打印服务器中形成的队列，这些是队列的应用。

本章讨论栈和队列。

3.1 栈

3.1.1 栈的定义及其运算

1. 定义

在表尾一端进行删除和插入操作的线性表称为栈(stack)。

允许插入和删除操作的一端称为栈顶(top)，另一端称为栈底(bottom)。处于栈顶位置的数据元素称为栈顶元素。不含任何数据元素的栈称为空栈。

2. 基本运算

栈的操作不能在栈的中间进行，其插入操作和删除操作都在栈顶进行。对于栈顶元素，可以访问它，也可以弹出栈顶元素。要使用栈底元素必须弹出所有压在它上面的元素才能实现。

设栈为S，下面是栈的基本运算。

(1) **入栈** push(self,x)：将元素插入栈中，使x成为栈S的栈顶元素。

(2) **出栈** pop(self)：取栈顶元素，并从栈S中删除栈顶元素。

(3) **取栈顶元素** GetTop(self)：取栈S的栈顶元素。

(4) **判空** Empty(self)：判断栈S是否为空。

3.1.2 栈的顺序存储结构

1. 定义

使用线性表的顺序存储结构来表示栈称为栈的顺序存储结构，一般称为顺序栈。常用

数组来描述栈的顺序结构。定义 SequenceStack 类,类的初始化函数实现了栈的顺序存储结构定义。具体如下:

定义 3.1 顺序栈

```
def __init__(self):
    self.MaxStackSize = 100
    self.s = [None for x in range(0,self.MaxStackSize)]
    self.top = -1
```

其中,self.s 是一个 Python 列表,长度为 MaxStackSize; top 的值是整数类型,top 表示栈顶指针。top 为-1 时表示空栈。

2. 栈的操作

顺序栈的操作如图 3.1 所示。

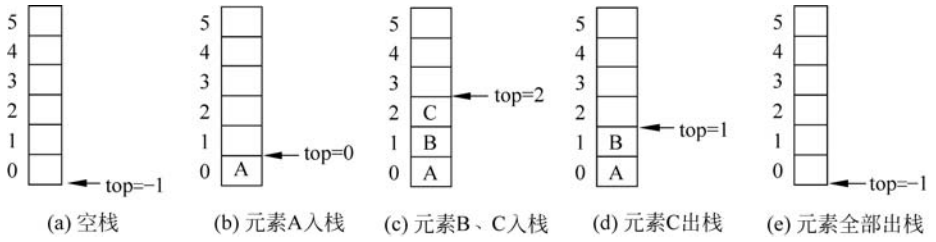

图 3.1 顺序栈的插入和删除操作

图 3.1(a)表示空栈,此时栈顶指针的值 top=-1。如果做出栈操作,则产生“下溢”。

图 3.1(b)表示栈中已经通过 push 操作,在图 3.1(a)表示的栈中压入元素 A,此时栈顶指针加 1,top=0。

图 3.1(c)表示在图 3.1(b)的基础上连续两次对栈进行 push 操作,压入元素 B、C,此时栈顶指针的值 top=2。

图 3.1(d)表示在图 3.1(c)的基础上做 pop 操作弹出元素 C 后的状态,此时栈顶指针在图 3.1(c)的基础上减 1,top=1。

图 3.1(e)表示在图 3.1(d)的基础上连续两次进行 pop 操作,栈已经为空栈,此时栈顶指针的值 top=-1。

3. 顺序栈的基本运算

1) 入栈

算法 3.1 入栈算法

```
def push(self, x):
    if self.top < self.MaxStackSize - 1:
        self.top = self.top + 1
        self.s[self.top] = x
    else:
        print("栈满")
    return
```

2）出栈

算法 3.2 出栈算法

```
def pop(self):
    if self.Empty():
        print("栈为空")
        return
    else:
        iTop = self.top
        self.top = self.top - 1
        return self.s[iTop]
```

3）取栈顶元素

算法 3.3 取栈顶元素算法

```
def GetTop(self):
    if self.Empty():
        print("栈为空")
        return
    else:
        return self.s[self.top]
```

4）判空

算法 3.4 判空算法

```
def Empty(self):
    if self.top == -1:
        iTop = True                          #空栈返回 True
    else:
        iTop = False                         #栈不为空返回 False
    return iTop
```

算法 3.1～算法 3.4 的时间复杂度均为 O(1)。

例 3.1 编写一个算法，将栈 S 的内容反序排列。

解：设 S 是一个顺序栈，要把其中的内容反序排列。设置两个辅助栈 S1 和 S2，用把 S 中的内容装入 S1 和 S2 的办法来实现。

算法如下：

```
def reverse_stack(S):
    S1 = SequenceStack()
    S2 = SequenceStack()
    while !Empty(S):
        x = pop(S)
        push(S1,x)
    while !Empty(S1):
        x = pop(S1)
        push(S2,x)
    while !Empty(S2):
        x = pop(S2)
        push(S,x)
```

3.1.3 栈的链式存储结构

1. 定义

与线性表一样,栈也可以使用链表来表示,此时栈称为链栈。定义链栈的结点类 StackNode。在 StackNode 类中实现链栈结点的初始化。

定义 3.2 链栈的结点定义

```
class StackNode(object):
    def __init__(self):
        self.data = None
        self.next = None
```

接着定义链栈类 LinkStack。

定义 3.3 LinkStack 类

```
class LinkStack(object):
    def __init__(self):
        self.top = StackNode()
```

上面的逻辑定义中定义了一个 top 指针,用来指示链栈的栈顶。

2. 链栈的操作

对一个带有头结点的链栈,进行插入和删除操作如图 3.2 所示。

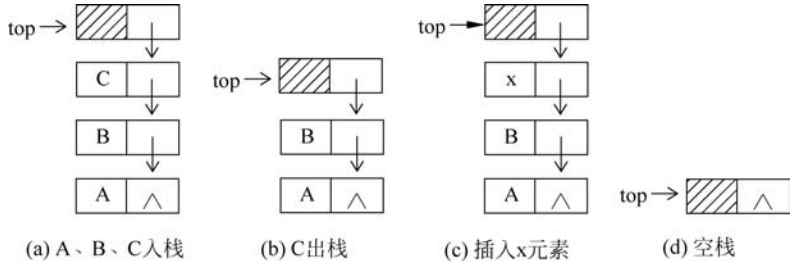

图 3.2 链栈的插入和删除操作

在图 3.2(a)中,栈中有 3 个元素 A、B、C,top 是栈顶指针指向头结点,top.next 指向元素 C。元素 A 是栈底。

在图 3.2(b)中,在图 3.2(a)的基础上进行一次 pop 操作,栈中有两个元素 A、B,top.next 指向元素 B。

在图 3.2(c)中,在图 3.2(b) 的基础上进行一次 push 操作,压入元素 x 后的情况。top.next 指向元素 x。

在图 3.2(d)表示空栈的情况,top.next 指向 None。

3. 链栈的基本运算

对于一个带头结点的链栈,用 top 来指示栈。使用链栈,在元素入栈和出栈时,可以通过直接修改栈头结点的指针值来进行。

1）入栈

算法 3.5 入栈算法

```
def push(self,da):
    tStackNode = StackNode()
    tStackNode.data = da
    tStackNode.next = self.top.next
    self.top.next = tStackNode
    print("当前入栈元素为：",da)
```

2）出栈

算法 3.6 出栈算法

```
def pop(self):
    if self.Empty():
        print("空栈")
        return
    else:
        tStackNode = self.top.next
        self.top.next = tStackNode.next
    return tStackNode.data
```

3）取栈顶元素

算法 3.7 取栈顶元素算法

```
def GetTop(self):
    if self.Empty():
        print("空栈")
        return
    else:
        return self.top.next.data
```

算法 3.7 中调用了判空函数 Empty()，该函数的实现具体见算法 3.8。

4）判空

算法 3.8 判空算法

```
def Empty(self):
    if self.top.next == None:   return True                    #空栈返回 True
    else: return False                                          #非空返回 Flase
```

算法 3.5～算法 3.8 的时间复杂度均为 O(1)。

3.2 栈的应用

3.2.1 数制转换

数制转换是一种常见的计算。在十进制转换为二进制时，根据除 2 取余法，先得到的余数是结果数的低位数，最后得到的余数是结果数的最高位，如果直接打印，会使结果成为倒序数字。为了得到正确的结果，借助栈，可以把每次计算的余数压入栈，最先得到的余数在

栈底,最后得到的余数在栈顶,然后从栈中分别弹出各个余数并打印,可以得到正确的顺序。

十进制数 N 与其他数制 r 的转换,可以用栈来实现。算法基于下列原理:

$$N = int(N \div r) \times r + N \bmod r$$

如十进制数 1024 转换为八进制数 2000 的过程如图 3.3 所示。

N	int(N÷r)	N mod r	位置
1024	128	0	低位
128	16	0	
16	2	0	
2	0	2	高位

图 3.3　十进制数 1024 转换为八进制数 2000

设 s 为一个栈,进行数制转换的算法描述为:

① 初始化栈。

② 输入十进制数 x 和其他进制数 y。

③ 把 x 除以 y 的余数压入栈,商赋给 x;如果 x 为 0,结束计算,否则重复③。

④ 弹出栈顶元素并打印,如果栈空,结束,否则重复④。

算法 3.9　十进制数与其他数制的转换

```
s = LinkStack()
def Digit_Conversion():
    x = int(input("请输入一个十进制正整数:" ))
    y = int(input("请输入一个要转换为的目标进制:" ))
    while x:
        s.push(x % y)
        x = x//y
    while !Empty(s)
        k = s.pop()
        print(" %d " %k)
```

3.2.2　算术表达式转换

1. 算术表达式求值

例 3.2　写出表达式"3+4/25 * 8−6"操作数栈和运算符栈的变化情况。

解:建立操作数栈 S 和算符栈 F,操作数栈 S 存放扫描表达式时的数字,算符栈 F 存放算符。算术运算有如下优先次序。

① 先乘除,后加减。

② 从左到右计算。

③ 先括号内,后括号外。

为操作方便,给表达式的左右两边分别标记符号"#",作为表达式扫描的起点和终点。操作时,从表达式的左边开始扫描,算法如下。

(1) 若是操作数,压入 S,继续扫描下一个元素。

(2) 若是操作符 Δ,则进入循环:

① 若 F 不空，则比较 Δ 与 F 顶元素⊕的优先级。

② 若 Δ＞⊕，则转(3)，否则转③。

③ 使 S 退两栈，退出的数分别为 x1，x2；F 退一栈，退出的运算符为⊕，做运算 x3＝x1⊕x2，x3 压入 S。

④ 重复①，②，③，直到 Δ 的优先级高于 F 的顶元素，或 F 已空。

(3) 若 Δ 是结束符“＃”，则算法结束，此时 S 中只有一个结果元素，否则 Δ 进入 F，继续扫描下一个元素，返回(1)。

入栈和出栈的操作过程如图 3.4 所示。

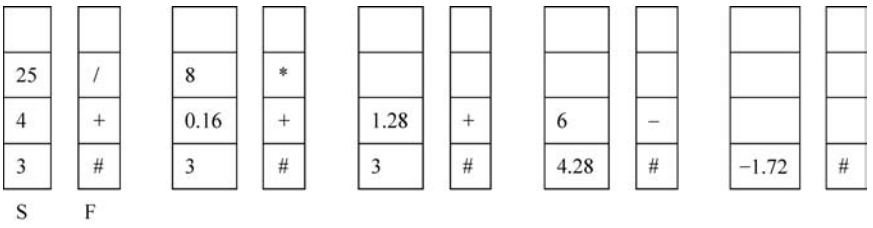

图 3.4　入栈和出栈的操作过程

2. 表达式转换

通常在算术表达式中，运算符放在两个操作数之间称为中序式(infix notation)。计算机程序的编译器在处理表达式时，通常不使用中序式，而是把中序式转换为前序式或后序式。前序式称为波兰式，后序式称为逆波兰式。

1) 中序式转换为前序式

所谓前序式是指运算符放在操作数的前面的表达式，进行转换的规则如下。

① 将算式根据运算优先次序完全括起来。

② 移动所有算符取代所有的左括号，以最近为原则。

③ 删去所有的右括号。

例 3.3　把表达式 A＊B/C 和(A＋B)－C＋D/E 转换为前序式。

解：((A＊B)/C)⟶/＊ABC

(((A＋B)－C)＋(D/E))⟶＋－＋ABC/DE

2) 中序式转换为后序式

所谓后序式是指运算符放在操作数的后面的表达式，进行转换的规则如下。

① 将算式根据运算优先次序完全括起来。

② 移动所有算符取代所有的右括号，以最近为原则。

③ 删去所有的左括号。

例 3.4　把表达式 A＊B/C 和 A＋B)－C＋D/E 转换为后序式。

解：((A＊B)/C)⟶AB＊C/

(((A＋B)－C)＋(D/E))⟶AB＋C－DE/＋

3) 中序式转换为后序式的分析

中序式转换为后序式时栈的算法如下。

(1) 将算式根据运算优先次序完全括起来；左右括号的运算优先级别比运算符低。

(2) 从左到右扫描算术表达式,令读入的符号为 Δ,若 Δ 是操作数,则将 Δ 输出到后序式字符串中;若 Δ 是运算符,则有以下情况要考虑。

① Δ=左括号,Δ 入栈。

② Δ=右括号,弹出栈顶运算符,若不是左括号,则将取出的运算符输出到后序串中,并重复此步直到取出左括号为止,然后将 Δ 与最后弹出的左括号丢弃,返回(2)。

③ Δ=运算符(+ - * /^)。

(i) 设栈顶的算符为⊕,若 Δ 的优先权大于⊕,则将 Δ 入栈。

(ii) 若 Δ 的优先级小于或等于⊕,则弹出栈顶算符⊕并输出到后序串中,返回①。

(3) 若栈不空,则将栈顶算符弹出到后序字串中,直到栈空为止。

例 3.5 以(A+B)-C+D/E 为例,说明中序表达式转换为后序式的操作栈的使用情况。

解:先根据算术运算的优先顺序完善括号(((A+B)-C)+(D/E))。表 3.1 给出了中序表达式转换为后序式的操作栈的使用情况和转换过程。

表 3.1 中序式(A+B)-C+D/E 转换为后序式的操作

步骤	读进字符	栈中内容	当前后序字串	说　明
1	(	(		左括号入栈
2	(	((		左括号入栈
3	(	(((		左括号入栈
4	A	(((	A	操作数 A 直接输出到后序串
5	+	+ ((	A	加号入栈
6	B	+ ((	AB	操作数 B 直接输出到后序串
7	)	((	AB+	")"的优先级小于栈顶符号"+",弹出"+"到后序串中,继续弹出栈顶左括号并丢弃一对括号
8	-	- ((	AB+	减号的优先级比栈顶符号")"高,减号入栈
9	C	- ((	AB+C	操作数 C 直接输出到后序串
10	)	(	AB+C-	")" 的优先级小于栈顶符号"-",弹出"-"到后序串中,继续弹出栈顶左括号并丢弃一对括号

续表

步骤	读进字符	栈中内容	当前后序字串	说　　明
11	＋	＋ (	AB＋C－	加号的优先级比栈顶符号“)”高，加号入栈
12	(	(＋ (	AB＋C－	左括号入栈
13	D	(＋ (	AB＋C－D	操作数 D 直接输出到后序串
14	/	/ (＋ (	AB＋C－D	“/”的优先级比栈顶符号“(”高，“/”入栈
15	E	/ (＋ (	AB＋C－DE	操作数 E 直接输出到后序串
16	)	＋ (	AB＋C－DE/	“)”的优先级小于栈顶符号“/”，弹出“/”到后序串中，继续弹出栈顶左括号并丢弃一对括号
17	)		AB＋C－DE/＋	“)”的优先级小于栈顶符号“＋”，弹出“＋”到后序串中，继续弹出栈顶左括号并丢弃一对括号，结束

得到的后序序列为 AB＋C－DE/＋，这正是例 3.4 中第二例的结果。

3.2.3　子程序调用

在子程序调用中，当子程序完成后，应该正确地返回主程序调用该子程序的地方。在计算机技术中使用栈来完成此工作。具体做法是在调用子程序时，用栈把主程序调用处的所有参数保存起来，当子程序调用完成时，从栈中弹出调用主程序断点处的所有参数，使主程序能够正确地从断点处继续执行后续程序。

下面是一个用 Python 语言描述的程序多层调用的问题。

```
def B():
    …
    ③C()
…
def A():
    …
    ②B()
    …
def func():
    …
    ①A()
    …
```

图 3.5 是上面程序调用时栈的使用情况，为简单起见，调用 A()、B()、C()处的现场信

息分别用 A、B、C 表示。分析如下。

(1) 开始时,func()在①处调用 A(),程序在调用处的现场信息 A 被压入栈保护起来;当被调用的子程序 A()结束时,返回 func()调用 A()时的现场信息 A,使 func()得以继续进行下去。

(2) A()在执行中调用 B()(②的位置),系统又把程序 A()中调用处的现场信息 B 压入栈保护起来;B()完成后,返回 A()中调用 B()时的现场信息 B,使 A()继续执行直至完成。

(3) B()在执行中又调用了 C()(③的位置),系统再次把 B()中调用处的现场信息 C 压入栈。当 C()执行完成后,从栈中弹出 B()中调用 C()时的现场信息 C,使 B()继续执行直至完成。

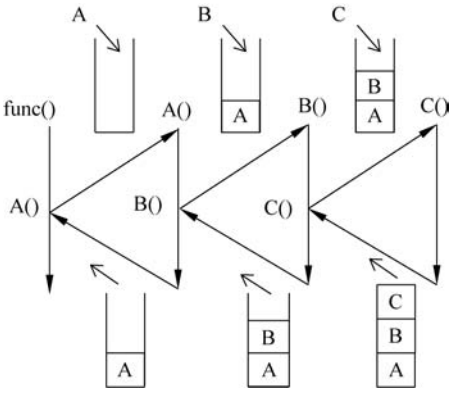

图 3.5　子程序调用时栈的使用情况

这样一层一层的调用,并使用栈来保护调用点的现场信息,是计算机处理程序的重要形式,调用点的现场信息包括(以 X86 处理器为例):

(1) 处理器的标志寄存器信息。

(2) 处理器中的指令寄存器 IP 中的内容。

(3) 处理器中代码段寄存器 CS 中的内容。

3.2.4　递归调用

递归调用是通过自身调用解决问题的一种办法。递归现象是自然界存在的一种奇妙的现象,能够用递归调用解决的问题,必须满足下列条件。

① 问题要有出口,否则递归调用将成为死循环。

② 问题是有限规模的。

递归调用与栈的使用密切相关,下面讨论几个常见的递归调用问题。

1. Hanoi 塔

Hanoi 塔问题这样描述:有 3 个塔座 X、Y、Z,在 X 上有 n 个盘,从顶到底依次编号为 1,2,3,…,n,如图 3.6(a)所示。现在要求把 X 座上的 n 个盘逐个移动到 Z 座,依然是原来在 X 座上的形态,如图 3.6(b)所示。

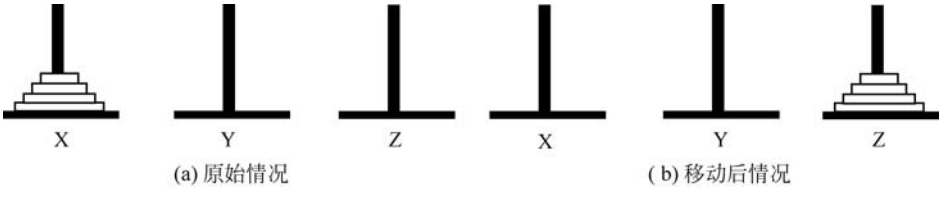

图 3.6　Hanoi 塔问题

盘移动的规则为:

(1) 每次只能移动一个盘。

(2) 盘可以插在 X、Y、Z 中的任何一个上。

(3) 任何时刻都不能将大盘压在小盘上。

求解 Hanoi 塔的算法如下。

当 n=1 时,直接把盘从 X 移到 Z;

当 n>1 时,需要用 Y 作为辅助塔,解法为:

① 把 X 上的 n−1 个盘移到 Y,移动中借助 Z;

② 把 X 上的 n 号盘移到 Z;

③ 把 Y 上的 n−1 个盘移到 Z,移动时借助 X。

算法 3.10 Hanoi 塔递归调用算法

```
def Hanoi(n,x,y,z):
    if n==1:
        move(1,x,z)
    else:
        Hanoi(n-1,x,z,y)
        move(n,x,z)
        Hanoi(n-1,y,x,z)
```

其中,move()的算法可以设计为:

```
def move(m,u,v):
    print("disk %d from %c,to,%c"%(m,u,v))
```

算法 3.10 的时间复杂度为 T(n)。设 Hanoi 函数的执行时间为 f(n),移动盘子的操作时间为 1,当 n=1 时,if 语句执行 1 次,则 f(1)=1。有

$$\begin{aligned} f(n) &= f(n-1)+1+f(n-1)\\ &= 2f(n-1)+1\\ &= \cdots\\ &= 2^i f(n-i)+2^{i-1}+\cdots+2+1 (=2^i f(n-i)+2^i-1)\\ &= \cdots\\ &= 2^{n-1}f(1)+2^{n-1}-1\\ &= 2^n-1 \end{aligned}$$

所以,T(n)=O(2^n),当 n 很大时,执行此算法很困难。在例 1.3 中曾经分析过当 n=1000 时 2^n 算法的情况,Hanoi 塔的时间复杂度正好是这个问题。

三个盘的 Hanoi 塔递归调用情况分析如图 3.7 所示。

系统在进行 Hanoi 塔的递归调用时,自动创建栈来保存递归调用中的函数调用关系。

2. n!

算法分析:根据定义,n!有下列关系。

$$n=\begin{cases}1 & \text{当 } n=0 \text{ 时}\\ n(n-1)! & \text{当 } n>0 \text{ 时}\end{cases}$$

当 n=0 时,可以结束 n!的计算,这就是出口。根据这个分析,n! 可以利用递归进行计算,算法如下。

算法 3.11 阶乘的递归调用

```
def fact(n):
    if n=0: return 1
    else: return n*fact(n-1)
```

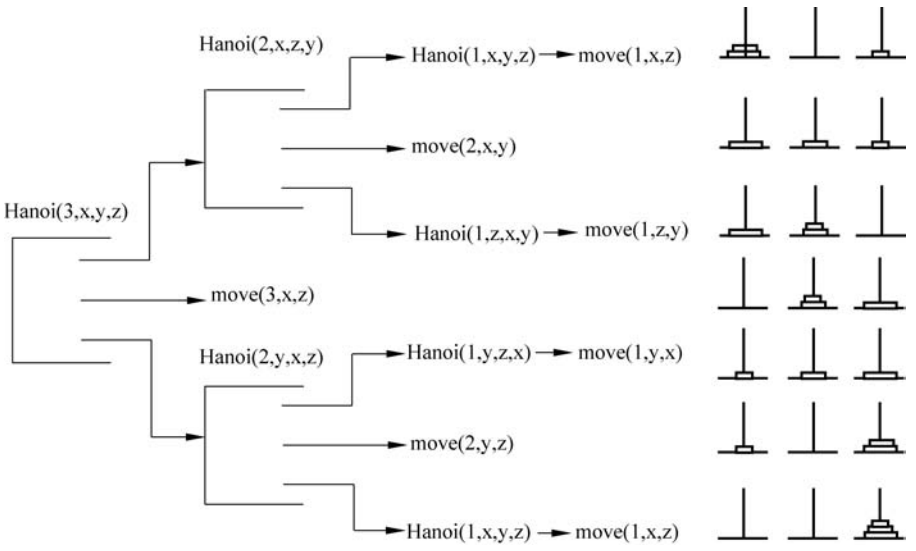

图 3.7　三个盘的 Hanoi 塔递归调用情况分析

n=5 时算法 3.11 的执行过程如图 3.8 所示。

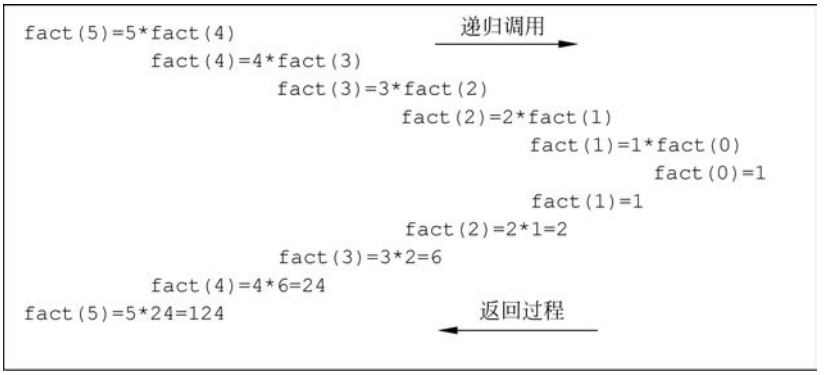

图 3.8　n=5 时算法 3.11 的执行过程

设此算法的时间复杂度为 T(n),fact()函数的执行时间为 f(n),当 n=0 时,递归调用结束,所以 f(0)=0,函数每调用一次,做一次乘法操作,所以有:

$$\begin{cases} f(0)=0 \\ f(n)=f(n-1)+1 \end{cases}$$

很容易得到 f(n)=n,所以 T(n)=O(n)。

3. Fibonacci 数列

Fibonacci 数列是 1202 年由 Fibonacci 发现的。该数列的形式为

0,1,1,2,3,5,8,13,21,34,55,…

其中,每一个数都是其前面相邻两个数的和。Fibonacci 数列有下列关系:

$$F_n=\begin{cases}1 & \text{当 } n=0,1 \text{ 时}\\ F_{n-1}+F_{n-2} & \text{当 } n>1 \text{ 时}\end{cases}$$

当 n=0,1 时,可以结束 Fibonacci 数列的计算,这就是出口。根据这个分析,Fibonacci 数列可以利用递归进行计算,算法如下。

算法 3.12 Fibonacci 数列的递归调用

```
def fib(n):
    if n<=1:
        s=n
    else:
        s=fib(n-1)+fib(n-2)
return s
```

设此算法的时间复杂度为 T(n),f(n)是算法执行的时间。可以得到

$$f(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

当 $n\to\infty$时,$f(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$,所以,$T(n)=O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$[1]

[1] KNUTH D E. The Art of Computer Programming(Volume 1)[M]. 北京:清华大学出版社,2002,79-84.

3.2.5 序列进出栈的排列问题

对于一个有序序列,由于序列入栈与出栈的顺序不同将导致出栈序列与原序列顺序不同。如何解决这个问题?下面先讨论出栈的序列数问题,其次讨论哪些排列是不可能的。

对于 n 个元素的序列顺序进栈,出栈共有多少可能的排列?经过计算可以得到排列数为

$$C_n=\frac{1}{n+1}C_{2n}^n$$

在这些排列中,哪些是可能的输出?哪些是不可能的输出?

对于初始输入序列 1,2,3,…,n,利用栈得到输出序列 $p_1p_2\cdots p_i\cdots p_n$,则在 $p_1p_2\cdots p_i\cdots p_n$ 中,如果有 i<j<k,则对于一个输入序列 $p_i<p_j<p_k$,即

$$\cdots,p_i,\cdots,p_j,\cdots,p_k \quad (p_i<p_j<p_k)$$

不存在这样的输出序列:

$$\cdots,p_k,\cdots,p_i,\cdots,p_j$$

因为 p_k 后进先出,满足栈的特点,而 p_i 在 p_j 的前面进入,却在 p_j 的前面出来,这是不可能的(因为不满足后进先出的原则)。

例 3.6 已知输入序列为 abcde,给出部分不可能的输出序列。

解:根据上面的讨论,不可能的输出序列有 cabde、adbce、abecd、aedbc,因为对于 cabde,c 在 ab 后进入,先出来是正确的,但 a 在 b 先进,却在 b 前先出来,这是不可能的。对于 adbce,a 先进入就出来,d 后进先出,但 b 在 c 先进,却在 c 前先出来,这是不可能的。类似地可以分析 abecd、aedbc。

下面的输出序列都是可行的,如 abcde、abced、abdce、abdec、edcba。

对于具有 5 个元素的输入序列,共有 42 种排列,所以只给出了部分输出结果。读者可以自己尝试寻找其他的可能或不可能的序列。

3.3 队　　列

3.3.1 队列的定义及运算

1. 定义

在表的一端进行删除操作而在另一端进行插入操作的线性表称为队列(queue)。

允许删除操作的一端称为队头,允许插入操作的一端称为队尾,如图 3.9 所示。新插入的元素只能被加到队尾,被删除的元素只能是队头元素。

为队头和队尾分别设置指针,队头指针指向队头元素的前一个单元,队尾指针指向队尾元素,当队头和队尾指针相等时,队列为空。

性质:特殊的线性表,先进先出(first in first out,FIFO)结构。

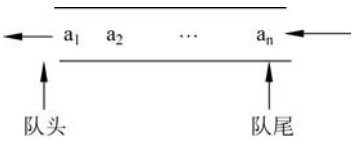

图 3.9　队列示意

2. 基本运算

设队列为 Q,下面是队列的基本运算。

(1) **初始化__init__(self)**:创建一个空队列 Q。

(2) **入队列 AddQ(self,x)**:入队操作,将元素插入队列中,Q 使 x 成为队列 Q 的元素。

(3) **出队列 DelQ(self)**:出队操作,取队头元素,并从队列 Q 中删除队头元素。

(4) **取队头元素 GetFront(self)**:取队头元素,该元素不出队列。

(5) **判空 Empty(self)**:判断队列 Q 是否为空。

3.3.2 队列的顺序存储结构

1. 定义

队列的顺序存储是指采用一组物理上连续的存储单元来存放队列中的所有元素。接下来用 Python 语言来实现顺序队列的定义和相关操作。首先定义顺序队列的类 SequenceQueue,并实现该类对象的初始化函数__init__(self)。具体如下。

定义 3.4　顺序队列的类 SequenceQueue

```
class SequenceQueue:
    def __init__(self):
        self. MAXSIZE = 10
        self.data = [None for x in range(0,self. MAXSIZE)]
        self.front = 0
        self.rear = 0
```

其中,data 是一个 Python 列表类型数据,长度为 MAXSIZE;front 指示队头元素的位置,rear 指向队尾元素的位置;队列空时,front=rear。

2. 队列的操作

顺序队列的操作如图 3.10 所示。

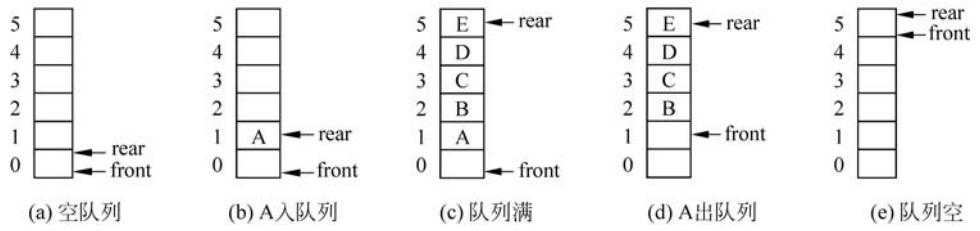

图 3.10　顺序队列的操作

图 3.10(a)表示空队列，此时队列的 front＝rear＝0。

图 3.10(b)表示元素 A 入队列，rear 加 1，使 rear＝1。

图 3.10(c)表示队列中连续进入元素 B、C、D、E，此时队尾指针 rear＝5。

图 3.10(d)表示在图 3.10(c)的基础上，元素 A 出队列，此时，front 加 1，front＝1。

图 3.10(e)表示在图 3.10(d)的基础上多次进行出队操作，队列已经为空队列，rear＝front＝5。

3. 循环队列

在图 3.10(e)中，因为 rear 和 front 已经达到队列尾部，rear＝front＝MAXSIZE－1，要再插入元素，队列已满，所以无法插入，因而产生了溢出，但实际上队列为空。这种现象称为假“溢出”。解决这个问题的办法是把队列首尾连接起来，构成循环队列。

循环队列的操作如图 3.11 所示。

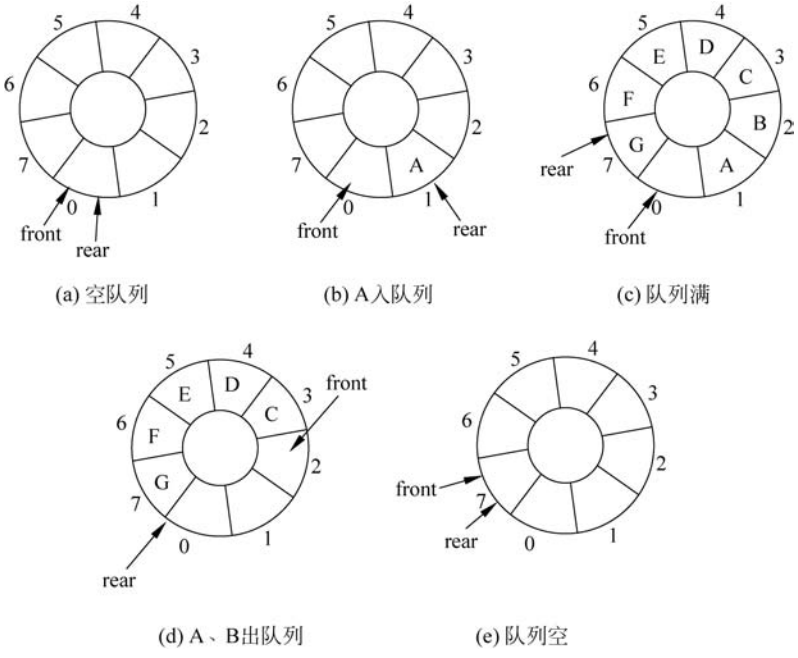

图 3.11　循环队列的操作

图 3.11(a)表示空队列，此时队列的 front＝rear＝0。

图 3.11(b)表示队列中已经有元素 A，rear 加 1，使 rear＝1。

图 3.11(c)表示队列中连续进入元素 B、C、D、E、F、G，此时队尾指针 rear＝7。

图 3.11(d)表示在图 3.11(c)的基础上,元素 A、B 出队列,此时,front=2。

图 3.11(e)表示在图 3.11(d)的基础上多次进行出队操作,队列已经为空队列,rear=front=7。

在图 3.11(e)的基础上,rear 和 front 均达到队尾,即 front=rear=MAXSIZE-1 时,该怎么办?此时再前进一个位置就是 0,那么利用除法取余数的运算,就可以实现 rear、front 变量顺利从 MAXSIZE-1 滑动到 0。即

```
rear = (rear + 1) % MAXSIZE
front = (front + 1) % MAXSIZE
```

这样就可以解决循环队列的指针在 MAXSIZE-1 和 0 之间的顺利转换。

当使用循环队列时,虽然解决了假溢出的问题,但是,又该怎样判断队列是空还是满呢?当出队的速度快于入队的速度时,队头指针就会赶上队尾指针,使得 front=rear,则队列已空。如果入队的速度快于出队的速度,则队尾 rear 指针很快赶上队头指针 front,当 rear=front 时,队列已满。队列空和满都有 rear=front。这使判断难以进行。约定,使用 front=rear 作为空队列条件,而使用(rear+1) % MAXSIZE=front 作为队列满条件(见图 3.11(c))。

4. 基本运算

创建循环队列类 CircularSequenceQueue,在该类定义中实现循环队列的基本运算。设循环队列为 Q。

1) 初始化

算法 3.13 初始化算法

```
def __init__(self):
    self.MAXSIZE = 10
    self.data = [None for x in range(0,self. MAXSIZE)]
    self.front = 0
    self.rear = 0
```

2) 入队列

算法 3.14 入队列算法

```
def AddQ(self,x):
    if (self.rear + 1) % self.MAXSIZE!= self.front:
        self.rear = (self.rear + 1) % self.MAXSIZE          #队尾指针加 1
        self.data[self.rear] = x                            #元素入队
         print("当前入队元素为: ",x)
    else:
        print("队列已满,无法入队")
        return
```

3) 出队列

算法 3.15 出队列算法

```
def DelQ(self):
    if self.Empty():
        print("队列为空,无法出队")
```

```
        return
    else:
        self.front = (self.front + 1) % self.MAXSIZE        #队头指针加 1
        return self.data[self.front]                        #队头元素出队
```

4）取队头元素

算法 3.16 取队头元素算法

```
def GetFront(self):
    if self.Empty():
        print("队列为空,无法输出队头元素")
        return
    else:
        #返回队头元素
        return self.data[(self.front + 1) % self.MAXSIZE]
```

5）判空

算法 3.17 判空算法

```
def Empty(self):
    if self.front == self.rear:
        return True                                         #空队列返回 True
    else:
        return False                                        #非空返回 False
```

算法 3.13～算法 3.17 的时间复杂度均为 O(1)。

例 3.7 编写一个算法，将队列 Q 的内容反序排列。

解：设 Q 是一个顺序队列，要把其中的内容反序排列。设置一个辅助栈 S，先把 Q 中的内容压入 S，然后弹出栈中内容的办法来实现。

算法核心部分如下：

```
def reverse_queue(Q):
    S = SequenceStack()                 #S 是一个栈对象
    while !Empty(Q):
        x = DelQ(Q)
        S.push(x)
    SetNONE(Q)                          #将队列 Q 清空
    while !Empty(S):
        x = S.pop()                     #将 S 中的元素依次出栈
        AddQ(Q,x)                       #将出栈元素加入队列 Q,实现元素反序排列
    return Q
```

3.3.3 队列的链式存储结构

1. 定义

队列的链式存储结构简称链队列。此时，每个结点增加一个链域。为了描述方便，首先定义链队列的结点类。

定义 3.5 链队列的结点类

```
class QueueNode:
    def __init__(self):
        self.data = None
        self.next = None
```


其中,data 是一个任意类型的数据元素,next 为链域。

接着定义链队列类 LinkQueue,在该类中首先实现了链队列的初始化函数,具体如下:

定义 3.6 链队列类

```
class LinkQueue:
    def __init__(self):
        tQueueNode = QueueNode()
        self.front = tQueueNode
        self.rear = tQueueNode
```

代码中的 front 和 rear 在后续算法实现中分别用来指向链队列的头和尾。

设队列为带头结点的链队列,Q 是队列的指针结点,图 3.12 说明了链队列的操作。

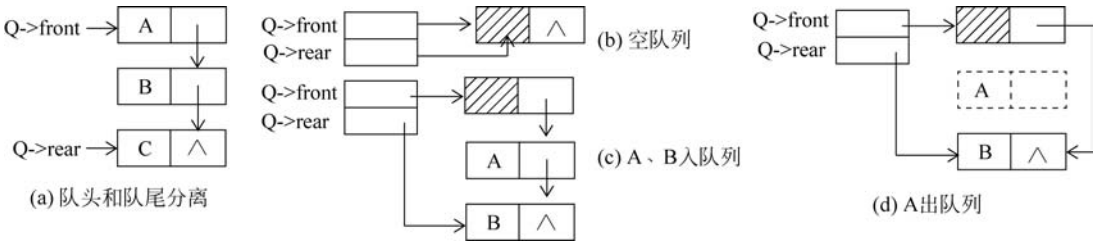

图 3.12 链队列的操作

图 3.12(a)表示链队列的一般情况(不带头结点),队头与队尾分离。

图 3.12(b)表示前面逻辑定义的链队列的情况,只有一个头结点的队列,即空队列。

图 3.12(c)表示链队列中连续进入元素 A、B 两个元素的情形,此时 Q.front 指向队列的头结点,而 Q.front.next 指向队列中的第一个元素 A,Q.rear 指向队尾元素 B。

图 3.12(d)表示在图 3.12(c)的基础上,元素 A 出队列后的情形,此时 Q.front.next 指向队列中的第一个元素 B,Q.rear 指向队尾元素 B。

2. 链队列的基本运算

设链队列为 Q,算法 3.18～算法 3.21 的时间复杂度均为 O(1)。

1) 入队列

算法 3.18 入队列算法

```
def AddQ(self, x):
    p = QueueNode()
    p.data = x
    self.rear.next = p                    # 头结点指针指向新入队结点
    self.rear = p                         # 队列尾指针指向新入队结点
    print("当前进队的元素为: ",x)
```

2) 出队列

算法 3.19 出队列算法

```
def DelQ(self):
    if self.Empty():
        print("队列为空")
        return
    elif self.front.next.next == None:      # 队列中只有一个元素时
        self.front.next = None
        self.rear = self.front
    else:
        p = self.front.next                 # 指针 p 指向队列第一个元素
        self.front.next = p.next            # 队头指针指向第二个元素
        return p.data                       # 第一个元素出队
```

3）取队头元素

算法 3.20 取队头元素算法

```
def GetFront(self):
    if self.Empty():
        print("队列为空")
        return
    else:
        return self.front.next.data
```

4）判空

算法 3.21 判空算法

```
def Empty(self):
    if self.front == self.rear:
        return True                         # 空队列返回 True
    else:
        return False                        # 非空队列返回 False
```

3.3.4 队列的应用

在计算机领域，队列有十分广泛的应用。网络中在同一链路上传输的数据报文形成一个队列，在网络环境下各个客户申请打印的作业在打印服务器中形成一个队列，在操作系统中存在临界资源的共享和互斥的问题等也使用了队列。

下面是操作系统中使用队列的问题示例——**生产者与消费者问题**。这是一个利用循环队列解决临界资源互斥的经典问题。

假定在生产者和消费者之间的公用缓冲池中具有 n 个缓冲区，用循环队列 buffer[n]表示（如图 3.13 所示，图中阴影部分表示存放产品区，空白部分为空闲缓冲区），用于存放生产的产品。又假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将产品送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个产品。

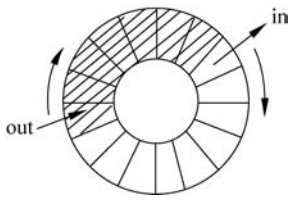

图 3.13 生产者与消费者问题

设信号量 mutex 表示公共信号量，实现诸进程对 buffer[n]的互斥使用，初值为 1；empty 为生产者信号量，表示队列是否为空，初值为 n；full 表示消费者信号量，表示队列是否为满，初值为 0。

首先,在每个程序中用于实现互斥的wait(mutex)和signal(mutex)必须成对出现;其次,对资源信号量empty和full的wait和signal操作,同样需要成对出现,但它们分别处于不同的算法中;最后,在每个算法中的多个wait操作顺序不能颠倒,应先执行对资源信号量的wait操作,再执行对互斥信号量的wait操作,否则可能引起进程死锁。

关于信号量S的wait(S)和signal(S)操作的定义分别为:

wait(S):while S≤0 do no-op;

S=S-1;

若S≤0,则在信号量S上的操作等待

signal(S):S=S+1;

若S≤0,则唤醒信号量S上等待的操作。

wait(S)和signal(S)是两个原子操作,因此,它们在执行时是不可中断的。也即当一个进程在修改某信号量时,没有其他进程可同时对该信号量进行修改。

整型量in、out表示生产和消费一个产品的下标变量。in表示生产一个产品并加入缓冲池的下标,初值为0,当生产者生产一个产品并加入buffer[n]时,in=(in+1) mod n。out表示从buffer[n]中取走一个产品时的下标,初值为0,当消费者从缓冲池取走一个产品时,out=(out+1) mod n。

根据上述分析,生产者与消费者共用缓冲池的核心算法如算法3.22所示。

算法3.22 生产者与消费者算法

```
buffer = [item] * n                    #buffer为一个循环队列
mutex = 1
empty = n
full = 0
in = 0
out = 0
def proceducer():
    while True:
        produce an item                #生产一个产品
        wait(empty)                    #empty - 1
        wait(mutex)                    #mutex - 1
        buffer(in) = item              #产品入队
        in = (in + 1) % n              #修改队头指针的下标
        signal(mutex)                  #mutex + 1
        signal(full)                   #full + 1
def consumer():
    while True:
        wait(full)                     #full + 1
        wait(mutex)                    #mutex - 1
        item = buffer(out)             #产品出队
        out = (out + 1) % n            #修改队头指针的下标
        signal(mutex)                  #mutex + 1
        signal(empty)                  #empty + 1
        consumer the item              #消费一个产品
```

本章小结

本章讨论了栈和队列的定义、逻辑结构、存储结构和它们的运算，并简单讨论了它们的应用。

栈是一种特殊的线性表，采用后进先出或先进后出的结构。在栈中插入或删除元素只能在其一端进行，能够进行插入和删除操作的一端称为栈顶，另一端称为栈底。栈的存储结构与线性表类似，分为顺序结构和链表结构。栈的运算主要有初始化、入栈、出栈、取栈顶元素和判空。

队列是一种特殊的线性表，采用先进先出的结构。在队列中插入和删除元素只能在队列的不同端进行，能够进行插入操作的一端称为队尾，能进行删除操作的一端称为队头。队列的存储结构与线性表类似，分为顺序结构和链表结构。队列的运算主要有初始化、入队、出队、取队头元素和判空。

习 题 3

一、单项选择题

1. 将整数 1,2,3,4,5 依次进栈，最后都出栈，出栈可以在任何时刻(只要栈不空)进行，则出栈序列不可能是_____。

A. 2,3,4,1,5　　B. 2,3,1,4,5

C. 5,4,1,3,2　　D. 1,5,4,3,2

2. 若一个栈的进栈序列为 1,2,3,…,n，输出序列的第 1 个元素是 n，则第 i 个元素是_____。

A. n−1　　B. n−i+1　　C. i　　D. n−i−1

3. 设数组 data[n]作为循环队列 queue 的存储空间，front 为队头指针，rear 为队尾指针，则执行出队操作后，其头指针的 front 值为_____。

A. front=front+1　　B. front=(front+1)%(n−1)

C. front=(front−1)%n　　D. front=(front+1)%n

4. 向一个栈顶指针为 top 的链栈(不带头结点)，压入一个 s 所指的结点时，执行_____。

A. top.next=s;　　B. s.next=top.next; top.next=s;

C. s.next=top; top=s;　　D. s.next=top; top=top.next;

5. 若用一个大小为 6 的数组来实现循环队列，且当前 rear 和 front 的值分别为 0 和 3。当从队列中删除一个元素，再加入两个元素后，rear 和 front 的值分别为_____。

A. 1 和 5　　B. 2 和 4　　C. 4 和 2　　D. 5 和 1

6. 栈和队列的共同特点是_____。

A. 都是先进后出　　B. 都是先进先出

C. 没有共同点　　D. 只允许在端点处插入和删除元素

7. 判断一个栈 S(最大元素数为 MAX)为空的条件是_____。

A. S.top!=1　　B. S.top==−1

C. S.top !=MAX　　D. S.top !=MAX-1

8. 判断一个栈S(最大元素数为MAX)为满的条件是_____。

A. S.top!=1　　B. S.top==-1

C. S.top !=MAX　　D. S.top !=MAX-1

9. 最不适合用于链栈的链表是_____。

A. 只有表头指针没有表尾指针的循环双链表

B. 只有表尾指针没有表头指针的循环双链表

C. 只有表头指针没有表尾指针的循环单链表

D. 只有表尾指针没有表头指针的循环单链表

10. 从一个栈顶指针为top的链栈中删除一个结点时,用x保存被删除的结点,则执行_____。

A. x=top; top=top.next　　B. x=top.data

C. top=top.next;x=top.data　　D. x=top.data; top=top.next;

11. 判断一个队列Q(最大元素数为MAX)为空的条件是_____。

A. Q.rear-Q.front=MAX　　B. Q.rear-Q.front-1=MAX

C. Q.front= Q.rear　　D. Q.front= Q.rear+1

12. 判断一个队列Q(最大元素数为MAX)为满的条件是_____。

A. Q.rear-Q.front=MAX　　B. Q.rear-Q.front-1=MAX

C. Q.front= Q.rear　　D. Q.front= Q.rear+1

13. 在循环队列中是否可以插入下一个元素,_____。

A. 与队头指针和队尾指针的值有关

B. 只与队尾指针的值有关,与队头指针的值无关

C. 只与数组大小有关,与队头指针和队尾指针的值无关

D. 与曾经进行过多少次插入操作有关

14. 判断一个循环队列Q(最大元素数为MAX)为空的条件是_____。

A. Q.front==Q.rear

B. Q.front!=Q.rear

C. Q.front==(Q.rear+1)%MAX

D. Q.front!=(Q.rear+1)%MAX

15. 循环队列用数组A[n]表示,已知其头尾指针分别为front和rear,则当前队列中的元素个数为_____。

A. (rear-front+n)%n　　B. rear-front+1

C. rear-front-1　　D. rear-front

16. 中序表达式A-(B+C/D)*E的后序式是_____。

A. AB-C+D/E*　　B. ABC+D/-E*

C. ABCD/E*+-　　D. ABCD/+E*-

二、填空题

1. 仅允许在表的同一端进行插入和删除操作的线性表称为_____。

2. 队列的插入操作在_____进行,删除操作在_____进行;栈的插入操作在

_____进行，删除操作在_____进行。

3. 栈和队列的差别仅在于_____。

4. 通常元素进栈的操作是_____。

5. 通常元素出栈的操作是_____。

6. 设栈采用顺序存储结构，若已知 i－1 个元素进栈，则将第 i 个元素进栈时，进栈的时间复杂度为_____。

7. 若用不带头结点的单链表来表示链栈 S，则创建一个空栈要执行的操作是_____。

8. 从循环队列删除一个元素时，通常的操作是_____。

9. 向循环队列插入一个元素时，通常的操作是_____。

10. 在具有 MAX 个单元的循环队列中，队满时共有_____个元素。

11. 在 Q 的链队列中，判定只有一个元素的条件是_____。

三、简答题

1. 跟踪下列代码，显示每次调用后栈中的内容。

```
st = SequenceStack()
st.push('A')
st.push('B')
st.push('C')
st.pop()
st.pop()
st.push('D')
st.push('E')
st.push('F')
st.pop()
st.push('G')
st.pop()
st.pop()
st.pop()
```

2. 对于一个栈，输入序列为 A，B，C。如果输出序列由 A，B，C 所组成，试给出全部可能的输出序列和不可能的输出序列。

3. 跟踪下列代码，显示每次调用后队列中的内容。

```
Q = SequenceQueue()
Q.AddQ('A')
Q.AddQ('B')
Q.AddQ('C')
Q.DelQ()
Q.DelQ()
Q.AddQ('D')
Q.AddQ('E')
Q.AddQ('F')
Q.DelQ()
Q.AddQ('G')
Q.DelQ()
Q.DelQ()
Q.DelQ()
```

4. 设 Q[10]是一个顺序队列,初始状态为 front=rear=0,画出做完下列操作后队列的头尾的状态变化情况。如果不能入队,指出其元素,并说明理由。

d,e,b,g,h 入队
d,e 出队
i,j,k,l,m 入队
b 出队
n,o,p 入队

5. 设 Q[10]是一个循环顺序队列,初始状态为 front=rear=0,画出做完下列操作后队列的头尾的状态变化情况,如果不能入队,指出其元素,并说明理由。

d,e,b,g,h 入队
d,e 出队
i,j,k,l,m 入队
b 出队
n,o,p 入队

四、算法分析

1. 编写返回栈底元素的算法。

2. 编写算法,计算栈顶指针为 top 的栈中元素的个数。

3. 编写算法,利用栈和队列的运算将队列的内容反序排列。

4. 编写算法,返回队列中的最后一个元素。

5. 有两个栈 s1 和 s2 共享存储空间 c[1..m_0],其中一个栈底设在 c[1]处,另一个栈底设在 c[m_0]处,分别编写 s1 和 s2 的进栈 push(i,x)、退栈 pop(i)和设置栈空 SetNone(i)的函数,其中 i=1,2。

注意,仅当整个空间 c[1.. m_0]占满时才产生溢出。

6. 假定用一个循环单链表表示队列,该队列只设一个队尾指针 rear,不设队头指针,编写函数:①向循环队列中插入一个元素为 x 的结点;②从循环队列中删除一个元素。

7. 在 top 指向的链队列中,编写算法,求链队列中结点的个数。

8. 已知 n 为大于或等于零的正整数,试写出下列函数的递归算法。

$$f(n)=\begin{cases} n+1 & \text{当 } x=0 \text{ 时} \\ n*f(n/2) & \text{当 } x\neq 0 \text{ 时} \end{cases}$$

9. 写出顺序栈中,完整的十进制数转换为任意进制数的算法(包括压入、弹出、初始化栈、判断栈空及主程序)。

实　　训

一、实训目标

1. 掌握栈存储结构及其算法。

2. 掌握队列的存储结构及其算法。

二、实训要求

1. 编程实现栈的典型算法。

2. 编程实现队列的典型算法。

三、实训内容

1. 栈的存储及其算法。

(1) 顺序栈定义、生成、弹出与压入。

(2) 链栈的定义、生成、弹出与压入。

(3) 栈的应用。

2. 队列的存储及其算法。

(1) 顺序队列的定义、生成、入队、出队。

(2) 链队列的定义、生成、入队、出队表元素的插入。

(3) 循环队列。

四、实训案例

1. 顺序栈操作。

(1) 栈的类定义。

```
class SequenceStack(object):
    def __init__(self):
        self.MaxStackSize = 10
        self.s = [None for x in range(0,self.MaxStackSize)]
        self.top = -1
```

(2) 入栈操作。

```
def push(self,x):
    if self.top < self.MaxStackSize - 1:
        self.top = self.top + 1
        self.s[self.top] = x
    else:
        print("栈满")
    return
```

(3) 出栈操作。

```
def pop(self):
    if self.Empty():
        print("栈为空")
        return
    else:
        iTop = self.top
        self.top = self.top - 1
        return self.s[iTop]
```

(4) 获取栈顶元素。

```
def GetTop(self):
    if self.Empty():
        print("栈为空")
        return
    else:
        return self.s[self.top]
```

(5) 栈的判空。

```
def Empty(self):
    if self.top == -1:
        iTop = True
    else:
        iTop = False
    return  iTop
```

(6) 遍历栈中元素。

```
def StackTraverse(self):
    if self.Empty():
        print("栈为空")
        return
    else:
        for i in range(0,self.top+1):
            print(self.s[i],end = '  ')
print("\n")
```

(7) 输出栈元素个数。

```
def GetStackLength(self):
    if self.Empty():
        print("栈为空")
        return
    else:
        return self.top+1
```

(8) 输入栈内元素。

```
def CreateStackByInput(self):
    data = input("请输入元素(继续输入请按 Enter 键,结束输入请按"#": ")
    while data!= '#':
        self.push(data)
        data = input("请输入元素: ")
```

(9) 测试主函数。

```
SStack = SequenceStack()
SStack.CreateStackByInput()
print("栈内的元素为: ",end = '\n')
SStack.StackTraverse()
x = input("请输入入栈元素值: ")
SStack.push(int(x))
print("栈内的元素为: ",end = '\n')
SStack.StackTraverse()
x = SStack.pop()
print("弹出栈的元素值%d: ",x,end = '\n')
print("栈内的元素为: ",end = '\n')
SStack.StackTraverse()
```

(10) 运行结果。

```
请输入元素(继续输入请按 Enter 键,结束输入请按"#": 1
请输入元素: 2
请输入元素: 3
请输入元素: 4
请输入元素: #
栈内的元素为:
1  2  3  4
请输入入栈元素值: 6
栈内的元素为:
1  2  3  4  6
弹出栈的元素值: 6
栈内的元素为:
1  2  3  4
>>>
```

2. 链队列操作。

(1) 定义类结点。

```
class QueueNode(object):                    #定义类结点名称: QueueNode
    def __init__(self):
        self.data = None
        self.next = None
```

(2) 定义链队列表类。

```
class LinkQueue(object):                    #定义一个链式队列类: LinkQueue
    def __init__(self):
        tQueueNode = QueueNode()
        self.front = tQueueNode
        self.rear = tQueueNode
```

(3) 入队算法。

```
def AddQ(self,x):
    p = QueueNode()
    p.data = x
    self.rear.next = p
    self.rear = p
    print("当前进队的元素为: ",x)
```

(4) 出队算法。

```
def DelQ(self):
    if self.Empty():
        print("队列为空")
        return
    elif self.front.next.next == None:
        self.front.next = None
        self.rear = self.front
    else:
        p = self.front.next
```

```
        self.front = p.next
        return p.data
```

(5) 取对头元素。

```
def GetFront(self):
    if self.Empty():
        print("队列为空")
        return
    else:
        return self.front.next.data
```

(6) 判断队列是否为空的函数。

```
def Empty(self):
    if self.front == self.rear:
        return 1
    else:
        return 0
```

(7) 创建队列函数。

```
def CreateQueueByInput(self):
    data = input("请输入元素(继续输入请按 Enter 键,结束输入请按"#": ")
    while data!= '#':
        self.AddQ(data)
        data = input("请输入元素: ")
def PrintQueue(self):
    if self.Empty():
        print("队列为空")
        return
    else:
        p = self.front.next
        while p != None:
            if p.next!= None:
                print(p.data, end = '—>')
            else:print(p.data)
            p = p.next
```

(8) 测试主函数。

```
lq = LinkQueue()
lq.CreateQueueByInput()
print("当前的队列如下: ")
lq.PrintQueue()
```

(9) 运行结果。

```
请输入元素(继续输入请按 Enter 键,结束输入请按"#" : 1
当前进队的元素为: 1
请输入元素: 2
当前进队的元素为: 2
请输入元素: 3
```

```
当前进队的元素为：3
请输入元素：4
当前进队的元素为：4
请输入元素：5
当前进队的元素为：5
请输入元素：#
当前的队列如下：
1->2->3->4->5
```