第3章 栈和队列

3.1 栈

3.1.1 栈的定义

栈(stack)又名堆栈,它是一种运算受限的线性表,限定仅在表尾进行插入和删除操作,如图 3.1 所示。一端称为栈顶,相对地,另一端称为栈底。向一个栈插入新元素又称作进栈、入栈或压栈,是把新元素放到栈顶元素的上面,使之成为新的栈顶元素;从一个栈删除元素又称作出栈或退栈,是把栈顶元素删除,使其相邻的元素成为新的栈顶元素。

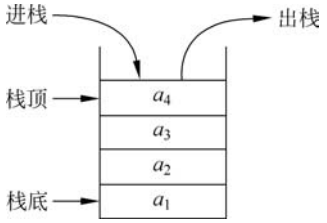

图 3.1 栈的示意图

栈作为一种数据结构,是一种只能在一端进行插入和删除操作的特殊线性表。它按照先进后出的原则存储数据,先进入的数据被压入栈底,最后进入的数据在栈顶,需要读数据时从栈顶开始弹出数据(最后一个数据被第一个读出来)。栈具有记忆作用,对栈的插入与删除操作中,不需要改变栈底指针。

栈是允许在同一端进行插入和删除操作的特殊线性表,栈底固定,而栈顶浮动;栈中元素个数为零时称为空栈。插入一般称为进栈(PUSH),删除则称为退栈(POP)。栈也称为先进后出表。

3.1.2 栈的顺序存储结构

1. 顺序栈的实现

采用顺序存储的栈称为顺序栈,它利用一组地址连续的存储单元存放自栈底到栈顶的数据元素,同时附设一个指针(top)指示当前栈顶元素的位置。

栈的顺序存储类型可描述为

```
class SqStack:
    def __init__(self, maxsize):
```

```
        self.maxSize = maxsize                  # 栈的容量
        self.stack = [None] * self.maxSize      # 顺序栈初始化
        self.top = -1                           # 栈顶指针
```

栈顶指针：self.top，self.top 初始时设置为－1，栈顶元素为 self.stack[self.top]。

进栈操作：栈不满时，栈顶指针先＋1，再给栈顶元素赋值。

出栈操作：栈非空时，先取得栈顶元素值，再将栈顶指针－1。

栈空条件：self.top＝＝－1。

栈满条件：self.top＝＝self.maxSize－1。

栈长：self.top＋1。

顺序栈的入栈操作会受数组上界的约束，当数组空间不足时，会发生栈上溢的情况，此时程序应该能够及时判断并报告错误信息。

2. 顺序栈的基本操作

1）判栈空

判断栈顶指针是否等于－1。

```
def is_empty(self):
    return self.top == -1
```

2）进栈

栈不满时，栈顶指针先＋1，再给栈顶元素赋值。

```
def push(self, x):
    if self.top == self.maxSize - 1:
        raise Exception("栈已满!")
    self.top += 1                   # 栈顶指针+1
    self.stack[self.top] = x
```

3）出栈

栈非空时，先取得栈顶元素值，再将栈顶指针－1。

```
def pop(self):
    if self.top == -1:
        raise Exception("栈为空!")
    e = self.stack[self.top]        # 获取出栈的元素
    self.top -= 1                   # 栈顶指针-1
    return e                        # 返回出栈元素
```

4）读取栈顶元素

栈非空时，栈顶元素为 stack[self.top]。

```
def get_top(self):
    if self.top == -1:
        raise Exception("栈为空!")
    return self.stack[self.top]
```

分析可得，入栈和出栈操作的实现为顺序表的尾插入和尾删除，时间复杂度为 $O(1)$。

【实例操作】

假设有一个容量为 5 的栈,进行如下操作:将 a,b,c,d,e 依次入栈;进行两次出栈操作并输出出栈元素;输出当前栈顶元素;将栈中剩余元素出栈并输出。

```
class SqStack:
    def __init__(self, maxsize):
        self.maxSize = maxsize                  # 栈的容量
        self.stack = [None] * self.maxSize      # 顺序栈初始化
        self.top = -1                           # 栈顶指针

    # 判栈空
    def is_empty(self):
        return self.top == -1

    # 进栈
    def push(self, x):
        if self.top == self.maxSize - 1:
            raise Exception("栈已满!")
        self.top += 1                           # 栈顶指针 + 1
        self.stack[self.top] = x

    # 出栈
    def pop(self):
        if self.top == -1:
            raise Exception("栈为空!")
        e = self.stack[self.top]                # 获取出栈的元素
        self.top -= 1                           # 栈顶指针 - 1
        return e                                # 返回出栈元素

    # 获取栈顶元素
    def get_top(self):
        if self.top == -1:
            raise Exception("栈为空!")
        return self.stack[self.top]

    # 打印出栈顺序
    def clear_print(self):
        while self.top != -1:
            print(self.pop(), end = ' ')


stk = SqStack(5)
stk.push('a')                                   # 入栈
stk.push('b')                                   # 入栈
stk.push('c')                                   # 入栈
stk.push('d')                                   # 入栈
stk.push('e')                                   # 入栈
print(stk.pop())                                # 出栈
print(stk.pop())                                # 出栈
print(stk.get_top())                            # 输出栈顶元素
```

```
stk.clear_print()                                    # 输出剩余元素出栈顺序
```

【输出】

```
e
b
c
c b a
```

3.1.3 栈的链式存储结构

1. 链栈的实现

采用链式存储的栈称为链栈。链栈相比于顺序栈的优点在于便于多个栈共享存储空间和提高其效率，且不存在栈满上溢的情况。链栈通常采用单链表来实现，规定栈的所有操作都在单链表的表头进行，如图 3.2 所示。

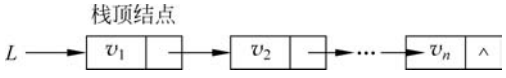

图 3.2 栈的链式存储

由于入栈和出栈只能在栈顶进行，不存在在栈的任意位置进行插入和删除的操作，所以不需要设置空头结点，只需将指针 top 指向栈顶结点，每个结点的指针域指向其后继结点即可。

栈的链式存储类型可描述为

```
class Node:
    def __init__(self, data = None, next = None):
        self.data = data          # 数据域
    self.next = next              # 指针域
```

栈顶指针：self.top，self.top 初始时设置为 None，栈顶元素值为 self.top.data。

进栈操作：将新结点插入链栈的栈顶。

出栈操作：将链栈的栈顶更新成其后继结点。

2. 链栈的基本操作

1）判栈空

top 为 None 时，即为栈空。

```
def is_empty(self):
    return self.top is None
```

2）进栈

将数据域值为 x 的结点插入链栈的栈顶。

```
def push(self, e):
        p = Node(e, self.top)        # 构造新结点，并使其指向栈顶结点
        self.top = p                 # 将新结点更新为栈顶结点
```

3）出栈

删除栈顶结点，即将链栈的栈顶更新成其后继结点。

```
def pop(self):
    if self.is_empty():
        raise Exception("栈为空!")
    p = self.top                    # 获取栈顶元素
    self.top = self.top.next        # 将栈顶元素指向后一位
    return p.data                   # 返回出栈结点的数据域值
```

分析可得,入栈和出栈操作的实现为单链表的头插入和头删除,时间复杂度为 $O(1)$。

【实例操作】

假设有一链栈,进行如下操作:依次将 a,b,c,d 入栈;出栈并输出出栈结点数据域值;将 f 入栈;输出目前链栈中各结点的数据域值。

```
class Node:
    def __init__(self, data = None, next = None):
        self.data = data            # 数据域
        self.next = next            # 指针域


class LinkStack:
    def __init__(self):
        self.top = None             # 指向栈顶元素结点

    # 判空
    def is_empty(self):
        return self.top is None

    # 入栈
    def push(self, e):
        p = Node(e, self.top)       # 构造新结点,并使其指向栈顶结点
        self.top = p                # 将新结点更新为栈顶结点

    # 出栈
    def pop(self):
        if self.is_empty():
            raise Exception("栈为空!")
        p = self.top                # 获取栈顶元素
        self.top = self.top.next    # 将栈顶元素指向后一位
        return p.data               # 返回出栈结点的数据域值

    # 打印栈中所有元素
    def print_stack(self):
        p = self.top
        while p is not None:
            print(p.data, end = ' ')
            p = p.next


l_stack = LinkStack()
l_stack.push('a')                   # 入栈
l_stack.push('b')                   # 入栈
```

```
l_stack.push('c')                    # 入栈
l_stack.push('d')                    # 入栈
print(l_stack.pop())                 # 出栈
l_stack.push('f')                    # 入栈
l_stack.print_stack()                # 输出目前链栈中各结点的数据域值(从栈顶到栈底)
```

【输出】

```
d
f c b a
```

3.2 队　　列

3.2.1 队列的定义

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端(front)进行删除操作，而在表的后端(rear)进行插入操作。和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。

队列的数据元素又称为队列元素。在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出(First In First Out，FIFO)线性表，如图 3.3 所示。

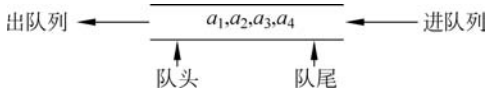

图 3.3　队列示意图

3.2.2 队列的顺序存储结构

1. 队列的顺序存储

顺序队列的存储结构与顺序栈类似，可用列表实现，因为入队和出队操作分别在队尾和队首进行，所以附设两个指针：队头指针 front 指向队头元素，队尾指针 rear 指向队尾元素的下一个位置。

队列的顺序存储类型及基本操作描述如下：

```
class SqQueue:
    def __init__(self, maxsize):
        self.maxSize = maxsize                  # 队列的容量
        self.queue = [None] * self.maxSize      # 队列初始化
        self.front = 0                          # 指向队首元素
        self.rear = 0                           # 指向队尾元素的下一个位置

    # 清空队列
    def clear(self):
        self.front = 0
```

```
        self.rear = 0

    # 判空
    def is_empty(self):
        return self.rear == self.front

    # 统计个数
    def length(self):
        return self.rear - self.front

    # 入队
    def push(self, e):
        if self.rear == self.maxSize:
            raise Exception("队列上溢!")
        self.queue[self.rear] = e              # 送值到队尾元素
        self.rear += 1                         # 队尾指针 + 1

    # 出队
    def pop(self):
        if self.is_empty():
            return
        e = self.queue[self.front]             # 获取出队元素
        self.front += 1                        # 队头指针 + 1
            return e
```

初始状态(队空条件):self . front == self . rear == 0。

进队操作:队不满时,先送值到队尾元素,再将队尾指针+1。

出队操作:队不空时,先取队头元素值,再将队头指针+1。

假设有一容量为 5 的队列,依次将 a,b,c,d,e 入队,然后出队两次,此时 self. rear == self. maxSize,如图 3.4 所示。若还要插入新的元素,则要发生“上溢”,但实际上队列中还有两个空位置,所以这种溢出称为“假溢出”。

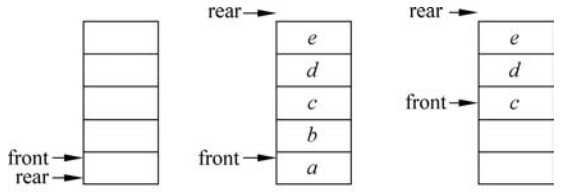

图 3.4　队列的操作

克服假溢出的方法有两种。一种是将队列中的所有元素均向低地址区移动,显然这种方法是很浪费时间的;另一种方法是将数组存储区看成一个首尾相接的环形区域,当存放到 n 地址后,下一个地址就“翻转”为 1。在结构上采用这种方法来存储的队列称为循环队列。

2. 循环队列

循环队列就是将队列存储空间的最后一个位置绕到第一个位置,形成逻辑上的环状空间,供队列循环使用。在循环队列结构中,当存储空间的最后一个位置已被使用而再要进入队运算时,只需要存储空间的第一个位置空闲,便可将元素加入到第一个位置,即将存储空

间的第一个位置作为队尾。循环队列可以更简单防止假溢出的发生，但队列大小是固定的。

当队首指针 front＝MaxSize －1 后，再前进一个位置就自动到了 0，这可以通过除法取余运算(％)来实现。

初始状态：front＝＝rear＝＝0。

队首指针进 1：front＝(front＋1)％ MaxSize。

队尾指针进 1：rear＝(rear＋1)％ MaxSize。

队列长度：(rear＋MaxSize－front)％ MaxSize。

队空条件：front＝＝rear。

假设有一个长度为 6 的循环队列，其初始状态是 front＝＝rear＝＝0，各种操作后队列的头、尾指针的状态变化情况如图 3.5 所示。

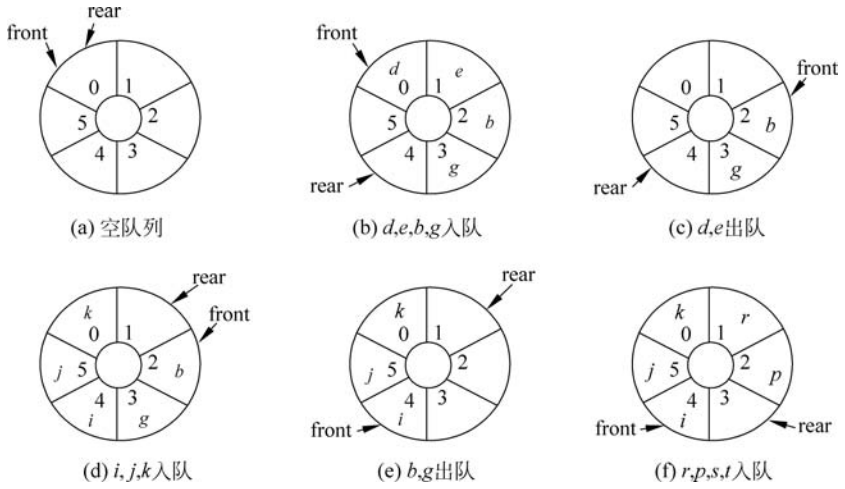

图 3.5　循环队列操作及指针变化情况

入队时尾指针向前追赶头指针，出队时头指针向前追赶尾指针，故队空和队满时头尾指针均相等。因此，无法通过 front＝＝rear 来判断循环队列是空还是满。

为了区分队空还是队满的情况，有以下三种处理方法：

(1) 牺牲一个单元来区分队空和队满，入队时少用一个队列单元，当队头指针在队尾指针的下一个位置时则表示队满。这是一种较为普遍的方法。应用该方法后：

队满条件：(rear＋1)％ MaxSize＝＝front。

队空条件不变：front＝＝rear。

队列中元素的个数：(rear＋MaxSize－front)％ MaxSize。

(2) 类型中增设表示元素个数的数据成员。这样当 front＝＝rear 时，只需根据该数据成员来判断队列队空还是队满。当 size＝＝0 时，则表示队空；当 size＝＝MaxSize 时，则表示队满。

(3) 类型中增设 tag 数据成员。当 tag＝0 时，若因删除导致 front＝＝rear 则为队空；当 tag＝1 时，若因插入导致 front＝＝rear 则为队满。

循环队列存储结构的描述如下(使用上述第一种处理方法)：

```
class CircleQueue:
    def __init__(self, maxsize):
```

```
        self.maxSize = maxsize                          # 队列的容量
        self.queue = [None] * self.maxSize              # 队列初始化
        self.front = 0                                  # 指向队首元素
        self.rear = 0                                   # 指向队尾元素的下一个位置
```


3. 循环队列的基本操作

1) 判队空

当 self.rear==self.front 时,队列为空。

```
    def is_empty(self):
        return self.rear == self.front
```

2) 获取表长

在循环队列中,表长为(self.rear−self.front+self.maxSize)% self.maxSize。

```
def get_length(self):
    return (self.rear - self.front + self.maxSize) % self.maxSize
```

3) 入队

```
def push(self, e):
    if (self.rear + 1) % self.maxSize == self.front:
        raise Exception("队列已满!")
    self.queue[self.rear] = e                           # 送值到队尾元素
    self.rear = (self.rear + 1) % self.maxSize          # 队尾指针 + 1
```

4) 出队

```
def pop(self):
    if self.is_empty():
        return None
    e = self.queue[self.front]                          # 获取出队元素
    self.front = (self.front + 1) % self.maxSize        # 队头指针 + 1
    return e
```

【实例操作】

假设有一容量为 5 的循环队列,进行如下操作:将 a,b,c,d,e 入队;进行三次出队操作并输出出队元素;将 f,g,h 入队;输出当前队列中所有的元素。

```
class CircleQueue:
    def __init__(self, maxsize):
        self.maxSize = maxsize                          # 队列的容量
        self.queue = [None] * self.maxSize              # 队列初始化
        self.front = 0                                  # 指向队首元素
        self.rear = 0                                   # 指向队尾元素的下一个位置

    # 判空
    def is_empty(self):
        return self.rear == self.front

    # 获取表长
    def get_length(self):
```

```
        return (self.rear - self.front + self.maxSize) % self.maxSize

    # 入队
    def push(self, e):
        if (self.rear + 1) % self.maxSize == self.front:
            raise Exception("队列已满!")
        self.queue[self.rear] = e                          # 送值到队尾元素
        self.rear = (self.rear + 1) % self.maxSize         # 队尾指针 + 1

    # 出队
    def pop(self):
        if self.is_empty():
            return None
        e = self.queue[self.front]                         # 获取出队元素
        self.front = (self.front + 1) % self.maxSize       # 队头指针 + 1
        return e

    # 输出队列中的所有元素
    def print_queue(self):
        pos = self.front
        while pos != self.rear:
            print(self.queue[pos], end = ' ')
            pos = (pos + 1) % self.maxSize


    cq = CircleQueue(5)
    cq.push('a')                                           # 入队
    cq.push('b')                                           # 入队
    cq.push('c')                                           # 入队
    cq.push('d')                                           # 入队
    print(cq.pop())                                        # 出队
    print(cq.pop())                                        # 出队
    print(cq.pop())                                        # 出队
    cq.push('e')                                           # 入队
    cq.push('f')                                           # 入队
    cq.push('g')                                           # 入队
    cq.print_queue()                                       # 输出队列中的所有元素
```

【输出】

```
a
b
c
d e f g
```

3.2.3 队列的链式存储结构

1. 队列的链式存储

在队列的形成过程中,可以利用线性链表的原理生成一个队列。链式队列可以看成一个同时带有队头指针和队尾指针的单链表。基于链表的队列,要动态创建和删除结点,可以

动态增长队列长度。

队列的链式存储类型的描述如下：

```
class Node:
    def __init__(self, data = None, next = None):
        self.data = data                    # 数据域
        self.next = next                    # 后继指针

class LinkQueue:
    def __init__(self):
        self.front = Node()                 # 队列头结点指针
        self.rear = self.front              # 队尾指针
```

链式队列通常设计成一个带头结点的单链表，这样插入和删除操作可以做到统一。链式队列特别适合于数据元素变动比较大的情况，而且不存在队列满且产生溢出的问题。

2. 链式队列的基本操作

1）判队空

当 front==rear 时，队列为空。

```
def is_empty(self):
    return self.front == self.rear
```

2）入队

```
def push(self, e):
    p = Node(e)
    self.rear.next = p                      # 将新结点插入队尾
    self.rear = p                           # 更新队尾指针
```

3）出队

```
def pop(self):
    if self.front == self.rear:
        raise Exception("队列已空!")
    p = self.front.next                     # 获取队列第一个元素
    self.front.next = p.next                # 删除该元素
    if self.rear == p:                      # 如果原队列只有一个结点,删除后变空队
        self.rear = self.front
    return p.data                           # 返回出队结点的数据域
```

【实例操作】

假设有一个链式队列，进行如下操作：将 a，b 入队；进行两次出队操作并输出出队元素；将 c，d 入队；输出当前队列中所有的元素。

```
class Node:
    def __init__(self, data = None, next = None):
        self.data = data                    # 数据域
        self.next = next                    # 后继指针


class LinkQueue:
```

```
    def __init__(self):
        self.front = Node()                    # 队列头结点指针
        self.rear = self.front                 # 队尾指针

    # 判空
    def is_empty(self):
        return self.front == self.rear

    # 入队
    def push(self, e):
        p = Node(e)
        self.rear.next = p                     # 将新结点插入队尾
        self.rear = p                          # 更新队尾指针

    # 出队
    def pop(self):
        if self.front == self.rear:
            raise Exception("队列已空!")
        p = self.front.next                    # 获取队列第一个元素
        self.front.next = p.next               # 删除该元素
        if self.rear == p:                     # 如果原队列只有一个结点,删除后变空队
            self.rear = self.front
        return p.data                          # 返回出队结点的数据域

    # 输出队列中的元素
    def print_queue(self):
        if self.front == self.rear:
            return
        cur = self.front.next
        while cur != self.rear:
            print(cur.data, end = ' ')
            cur = cur.next
        print(cur.data)


lq = LinkQueue()
lq.push('a')                                   # 入队
lq.push('b')                                   # 入队
print(lq.pop())                                # 出队
print(lq.pop())                                # 出队
lq.push('c')                                   # 入队
lq.push('d')                                   # 入队
lq.print_queue()                               # 输出队列中的所有元素
```

【输出】

```
a
b
c d
```

3.3 栈与队列相关算法设计与分析

【例 3.1】 括号匹配

给定一个字符串,它只包含'(',')','[',']','{','}',判断该字符串的有效性。有效字符串需满足以下两点:①左括号必须用相同类型的右括号闭合;②左括号必须以正确的顺序闭合。例如,[(])或(()]是无效的字符串,[(())]是有效的字符串。

【分析】

根据题意,如果字符串长度为奇数则一定是无效的,可以在遍历字符串之前,先做奇偶性的判断。在遍历给定的字符串的过程中,当遇到一个左括号时,会期望在后续的遍历中有一个相同类型的右括号将其闭合。而且位置靠后的左括号将会先于前面的左括号被闭合,故该问题符合栈的先进后出特性,适合用栈来解决该问题。

在遍历字符串中,当遇到左括号时,先将这个左括号放入栈中。当遇到一个右括号时,只需取出当前的栈顶元素,判断是否与该右括号匹配。假如不是相同的括号类型,或者栈中并没有左括号,那么该字符串一定是无效的。

在匹配过程中,可以使用 Python 中的字典来存储每一种括号。字典的键为右括号,值为相同类型的左括号。

在遍历结束后,如果栈中没有左括号,说明字符串中的所有左括号已闭合,返回 True,否则返回 False。

【算法】

首先对字符串长度做奇偶性的判断。如果是偶数则接下来遍历字符串:

(1) 如果括号是左括号,则入栈。

(2) 否则通过字典判断括号对应关系,若栈顶出栈的括号与当前遍历括号不对应,则提前返回 False。

遍历结束后,如果栈中没有括号,说明字符串中的所有左括号已闭合,字符串是有效的,返回 True,否则返回 False。

【代码】

```
class Solution(object):
    def isValid(self, s):
        if len(s) % 2 == 1:                              # 奇偶性判断
            return False
        dic = {')': '(', ']': '[', '}': '{'}             # 创建字典
        stack = []                                       # 创建以列表表示的栈
        for ch in s:                                     # 遍历字符串
            if ch in dic:
                if not stack or stack[-1] != dic[ch]:    # 将当前字符与栈顶字符比较
                    return False
                stack.pop()                              # 出栈
            else:
                stack.append(ch)                         # 入栈
        return not stack                                 # 是空栈返回 True,否则返回 False
```

【复杂度分析】

时间复杂度：$O(n)$，其中 n 为字符串的长度。该算法遍历了一次字符串。

空间复杂度：$O(n+|\Sigma|)$，其中栈中的字符数量为 $O(n)$，Σ 为字典的空间数量。本题只包含 6 种括号，故 $|\Sigma|=6$。

【例 3.2】 后缀表达式(逆波兰表达式)求值

根据后缀表达式(逆波兰表达式)，求表达式的值。有效的运算符包括+、-、*、/。例如，有一后缀表达式为 3 9 3/+，转化为中缀表达式为 3+9/3，结果为 6。

【分析】

后缀表达式的特点是运算符总是放在和它相关的操作数之后，并且严格遵循从左到右的运算。由于运算符总是在操作数之后，可以利用栈的先进后出规律，在遇到操作数时先入栈，遇到运算符时再从栈中取出两个操作数进行运算。

【算法】

遍历后缀表达式里的每一个字符：

(1) 如果遇到操作数，则将操作数入栈。

(2) 如果遇到运算符，则从栈中取出两个操作数，其中先出栈的是右操作数，后出栈的是左操作数，使用该运算符对这两个操作数进行运算，将得出的结果重新压入栈中。

遍历完毕后，栈中只剩下一个操作数，该数就是后缀表达式的值。

【代码】

```
class Solution(object):
    def evalRPN(self, tokens):
        stack = []                              # 初始化栈
        for ch in tokens:                       # 遍历表达式
            if ch not in "+-*/":                # 如果是操作数则入栈
                stack.append(int(ch))
            else:                               # 如果是运算符
                b = stack.pop()                 # 右操作数
                a = stack.pop()                 # 左操作数
                if ch == '+':
                    stack.append(a + b)
                elif ch == '-':
                    stack.append(a - b)
                elif ch == '*':
                    stack.append(a * b)
                elif ch == '/':
                    stack.append(int(a / b))
        return stack.pop()                      # 返回栈最后一个元素,即值
```

【复杂度分析】

时间复杂度：$O(n)$，其中 n 是后缀表达式的长度。算法中需要遍历后缀表达式一次。

空间复杂度：$O(n)$，其中 n 是后缀表达式的长度。栈中的元素数量不会超过后缀表达式的长度。

【例 3.3】 用队列模拟栈

使用两个先入先出队列模拟栈。栈应当支持一般栈支持的所有操作(入栈、出栈、获取

栈顶元素、判栈空)。

【分析】

可用对象为两个先入先出的队列,故可以考虑把两个队列分为主队列和副队列。假如入栈顺序为 a,b,首先要保证主队列前端元素一定是 b,即满足队列前端的元素是最后入栈的元素。当将 a 入栈时,可以先把 a 插入到副队列,由于是第一个元素,此时元素在队列里的顺序就是栈的顺序,故可以把该副队列设为主队列,另一个空主队列设为副队列,相当于将两个队列进行了交换,如图 3.6(a)所示。当 b 入栈时,此时主队列有 a,副队列为空,可先把 b 入队到副队列中,再将主队列的元素依次出队并入队到副队列,直到主队列清空,然后交换主队列和副队列,如图 3.6(b)所示。此时主队列元素为 b,a,副队列为空,主队列的元素顺序恰好符合栈的顺序。当栈出栈时,只需将主队列出队一个元素即可,如图 3.6(c)所示。故栈的入栈和出栈操作可以通过两个队列这样的操作来模拟。

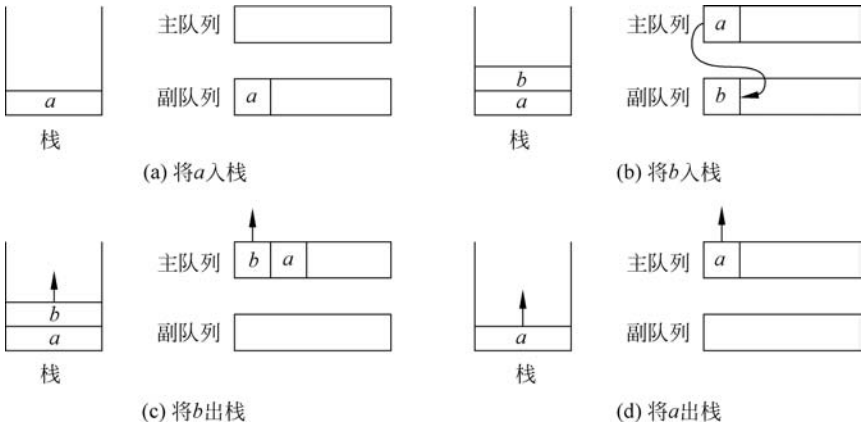

图 3.6　队列模拟栈过程

【算法】

入栈:

(1) 在副队列中入队新元素。

(2) 将主队列中所有元素依次出队并入队到副队列中。

(3) 将主队列和副队列交换。

出栈:将主队列出队。

栈顶元素;即主队列队头元素。

判栈空:即判主队列队空。

【代码】

为简略代码,本代码使用了 Python 里 collections 库中的 deque 结构来实现队列。读者可以查阅相关资料了解 deque 结构的方法。

```
class MyStack:
    def __init__(self):
        self.queue1 = collections.deque()          # 主队列
        self.queue2 = collections.deque()          # 副队列
```

```
# 入栈
def push(self, x):
    self.queue2.append(x)                          # 先入队到副队列
    while self.queue1:                             # 将主队列元素依次出队并入队到副队列
        self.queue2.append(self.queue1.popleft())
    self.queue1, self.queue2 = self.queue2, self.queue1      # 交换主副队列

# 出栈
def pop(self):
    return self.queue1.popleft()                   # 主队列出队

# 栈顶元素
def top(self):
    return self.queue1[0]                          # 返回主队列队头

# 判空
def empty(self):
    return not self.queue1                         # 判主队列队空
```

小　　结

(1) 栈是一种特殊的线性表,它只允许在栈顶进行插入和删除操作,具有后进先出的特性,插入和删除操作的时间复杂度都为 $O(1)$。栈可以采用顺序存储结构和链式存储结构。

(2) 队列也是一种特殊的线性表,它只允许在表头进行删除操作,在表尾进行插入操作,具有先进先出的特性,插入和删除操作的时间复杂度都为 $O(1)$。队列可以采用顺序存储结构和链式存储结构。

(3) 栈和队列的相同点与不同点如表 3.1 所示。

表 3.1　栈和队列的比较

相同点	(1) 都是线性结构,数据元素间是“一对一”的逻辑关系; (2) 都有顺序存储结构和链式存储结构两种实现方式; (3) 都有各自的操作规则,栈是后进先出,队列是先进先出; (4) 插入和删除操作的时间复杂度都为 $O(1)$
不同点	栈的删除操作在表尾进行,具有先进后出的特性;队列的删除操作在表头进行,具有先进先出的特性

(4) 循环队列是将顺序队列的头结点和尾结点相连,能够有效解决“假溢出”现象的发生。