第5章 栈和队列

本章将深入探讨栈和队列这两种经典的数据结构，它们在计算机科学中扮演着至关重要的角色。栈和队列都是线性数据结构，其特点在于基本操作的特殊性。栈必须按照“后进先出”的规则进行操作，而队列则必须按照“先进先出”的规则进行操作。相较于线性表，它们的插入和删除操作受到更多的约束和限制。本章将介绍这两种数据结构的基本概念、Python 实现及常见的操作。本章还会列举一些栈和队列的应用，帮助读者建立起对这两种数据结构的深入理解，为将来应用它们解决实际的计算机科学问题奠定基础。

5.1 栈的概念与实现

5.1.1 栈的结构和操作特点

在生活中，经常会遇到“后进先出”(Last In First Out，LIFO)的情形，一个简单的例子就是洗盘子和使用盘子的过程。假设有一堆完全相同的盘子，在清洗它们时会一个个地摞起来，最后一个洗好的盘子自然放在最上面。然而，当需要使用盘子时，会直接取最上面的盘子，而不是随机抽取。这种具有只从顶端进行放入和取出操作的结构，正是栈在现实生活中的一个直观体现。

栈是一种具有特殊性质的线性数据结构，它遵循 LIFO 的原则。如图 5.1 所示，在栈的运作过程中，元素的添加和移除都发生在同一端，这一端通常被称为**栈顶**，另一端被称为**栈底**。当栈中没有任何元素时，称为**空栈**。

由栈的定义出发，可以发现栈具有如下的结构特点：

(1) LIFO 原则：栈最显著的特点是最后添加的元素将会是第一个被移除的。

(2) 单一操作端：所有的操作(如添加、删除、访问等)都在栈顶进行。

(3) 限制性访问：栈不支持随机访问，不能跳过元素或者直接访问栈中间的某个元素。

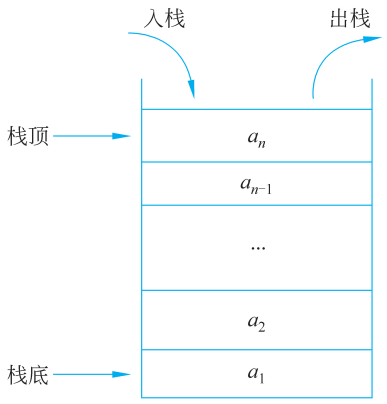

图 5.1　栈的示意图

只能访问栈顶元素，即最后一个添加的元素。

对于栈这种数据结构，可以定义一系列基本操作来管理栈的元素：

(1) is_empty()：检查栈是否为空。如果栈中没有任何元素，返回真(True)，否则返回假(False)。

(2) is_full()：仅适用于具有固定大小的栈，检查栈是否已满。如果栈达到了其最大容量，返回真(True)，否则返回假(False)。

(3) push(e)：将一个元素 e 添加到栈顶，通常称为“入栈”。对于具有固定大小的栈，在执行此操作之前，需要检查栈是否已满。如果栈满，该操作无法执行，这称为栈溢出。

(4) pop()：移除并返回栈顶元素，通常称为“出栈”。在执行此操作之前，需要检查栈是否为空。如果栈空，该操作无法执行，这称为栈下溢。

(5) peek()：返回栈顶元素的值，但不移除它。这个操作允许查看栈顶元素而不影响栈的状态。

通过这些基本操作，可以有效地管理栈中的元素，并利用它们来解决实际问题。例如，在计算机程序中，栈被广泛用于管理函数调用、表达式求值等场景。通过将数据压入栈中，可以在需要时恢复到之前的状态，这正是栈作为一种数据结构的强大之处。

5.1.2　栈的表示和实现

和线性表类似，栈也有两种存储表示方法：顺序栈和链栈。

1. 顺序栈

顺序栈指的是利用顺序存储分配实现的栈，即利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针 top 指示栈顶元素在顺序栈中的位置。可以预先设置顺序栈中数据元素存储区域的最大容量，这样就可以设置一个固定大小的栈。

代码清单 5.1 中给出了一个顺序栈类型 SStack 的 Python 语言实现。其中，使用列表 stack 来描述顺序栈中数据元素的存储区域，其容量由 capacity 决定。鉴于 Python 语言中列表的正向索引值从 0 开始，因此，以 top＝－1 表示空栈。每入栈一个元素，top 指针增加 1，top＝capacity－1 时栈满，此时若需要继续入栈，则应该先扩容。代码清单 5.1 中采用的方式是等比例扩容，直接按当前容量翻倍。也可以采用按照预设的扩容增量进行扩容的方

式，有兴趣的读者可以自行研究。每出栈一个元素，应返回 top 所指示的元素值，然后通过将 top 指针减去 1 的方式来实现“移除”栈顶元素。因此，SStack 的入栈、出栈操作的时间复杂度都是 $O(1)$。

代码清单 5.1　顺序栈类型 SStack 的 Python 语言实现

```
class SStack:
    def __init__(self, capacity=10):     #创建容量为 capacity 的空栈
        self.stack = [None] * capacity    #用列表存放栈中的元素
        self.top = -1                     #栈顶指针，-1 表示当前所含元素个数为 0
        self.capacity = capacity          #该顺序栈可以容纳 capacity 个元素

def is_full(self):                        #判断栈是否已满
    return self.top == self.capacity - 1

def is_empty(self):                       #判断是否为空栈
    return self.top == -1

def push(self, val):                      #将 val 入栈
    if not self.is_full():
        self.top += 1
    else:
        self.stack = self.stack + [None] * self.capacity          #先扩容一倍
        self.top = self.capacity
    self.stack[self.top] = val            #将 val 放在栈顶

def pop(self):                            #出栈
    if not self.is_empty():               #若栈不空，则移除栈顶元素并返回它
        item = self.stack[self.top]
        self.top -= 1                     #栈顶指针“下移”，意味着移除栈顶元素
        return item
    else:                                 #栈空，则输出相应提示信息并返回 None
        print("栈为空，无法移除元素。")
        return None

def peek(self):                           #返回栈顶元素的值，但不移除它
    if not self.is_empty():
        return self.stack[self.top]
    else:
        print("栈为空。")
        return None
```

2. 链栈

链栈指的是利用链式存储实现的栈，链栈的结点结构和单链表的结点结构相同，每个结点包含一个元素域和一个链接域（如图 5.2 所示）。

在单链表中，每个结点的链接域记录了其直接后继的链接，因此，顺着链可以从表头结点逐个遍历到表尾结点。在表头结点前插入一个新结点或者删除表头结点，时间复杂度都是 $O(1)$。但是在表尾结点后插入一个新结点或者删除表尾结点，时间复杂度都是 $O(n)$。

因此，在使用单链表来实现链栈的存储时，若想实现高效的入栈和出栈操作，应让栈顶

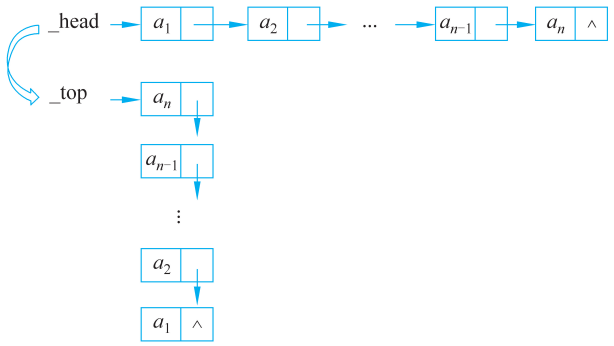

图 5.2　从单链表到链栈

元素成为表头结点。如图 5.2 所示，这一转变无须改变结点的结构，每个结点仍可由一个元素域和链接域构成，只是这里链接域链接的是结点逻辑上的直接前驱。这样，顺着链可以实现从栈顶结点开始逐个遍历到栈底结点。

代码清单 5.2 中给出了一个链栈类型 LStack 的 Python 语言实现。和单链表的实现一样，需要先定义一个表示结点的类型，它含有两个域，分别表示元素域和链接域。这里仍然沿用单链表中的结点类型 ListNode，不做修改。而对于链栈类型 LStack，它只有一个表示栈顶变量的_top 属性，如同单链表类型 LList，仅有一个表示表头变量的_head 属性一样。

代码清单 5.2　链栈类型 LStack 的 Python 语言实现

```
class ListNode:
    def __init__(self, val, next_=None):
        self.val = val
        self.next = next_

class LStack:
    def __init__(self):
        self._top = None

    def is_empty(self):                    #判断是否为空栈
        return self._top is None

    def push(self, val):                   #将 val 入栈
    self._top = ListNode(val, self._top)

    def pop(self):                         #出栈
        if not self.is_empty():
            item = self._top.val
            self._top = self._top.next
            return item
        else:
            print("栈为空，无法移除元素。")
            return None

    def peek(self):                        #返回栈顶元素的值，但不移除它
```

```
        if not self.is_empty():
            return self._top.val
        else:
            print("栈为空。")
            return None
```

上述定义的 SStack 类和 LStack 类的使用方式完全一样。例如，在“借助一个栈来逆序输出列表 ls 中的元素”时，可以将所有元素按照列表中的顺序依次入栈，然后再依次出栈并输出，就可以利用栈操作“后进先出”的特性来达到逆序输出列表中元素的目的。如下代码段给出了基于 SStack 类的实现方案。

```
st = SStack()              #第一行
for x in ls:
    st.push(x)             #依次入栈
while not st.is_empty():
    print(st.pop())        #依次出栈并输出
```

若想基于 LStack 类来完成此任务，仅需将第一行的代码替换为 st＝LStack()即可。从使用的角度来看，这两个类除了类名不同之外，完全可以相互代替。这也是抽象数据类型的功劳。

在基于 Python 语言进行栈相关的算法设计时，也可以直接使用列表类型及其操作来替代栈及其基本操作。建立空栈，就对应于创建一个空列表；那么判断是否为空栈，就对应于判断是否为空表的操作。同时，将表尾元素视为栈顶元素。入栈操作，可以用列表类型的 append 方法来代替；出栈操作，可以用列表类型的 pop 方法来代替，当其参数取默认值时，弹出的就是表尾元素。而取得栈顶元素的值，只需要使用列表的元素索引操作，取索引为－1 的元素即可。在这种设计下，上述“借助一个栈来逆序输出列表 ls 中的元素”也可以采用如下方式实现。

```
st = []                    #利用列表来直接表示栈
for x in ls:
    st.append(x)           #依次入栈
while st != []:            #栈非空则继续循环
    print(st.pop())        #依次出栈并输出
```

例 5.1 （力扣 946）验证栈序列。

【题目描述】

给定 pushed 和 popped 两个序列，每个序列中的值都不重复，设计算法来判定：这两个序列是否为在最初空栈上进行的入栈 push 和出栈 pop 操作序列的结果。如果是，则返回 True；否则，返回 False。

例如，给定 pushed＝[1,2,3,4,5]，popped＝[4,5,3,2,1]，应返回 True。因为可以按以下顺序执行入栈和出栈操作：push(1)，push(2)，push(3)，push(4)，pop()→4，push(5)，pop()→5，pop()→3，pop()→2，pop()→1。

又如，给定 pushed＝[1,2,3,4,5]，popped＝[4,3,5,1,2]，则应返回 False。因为前面的入栈和出栈操作只能按如下顺序进行：push(1)，push(2)，push(3)，push(4)，pop()→4，pop()→3，push(5)，pop()→5，这样一来，由于栈的 LIFO 特性，1 只能在 2 之后弹出，不可

能得到要求的 popped 序列。

【解题思路】

栈的入栈(push)和出栈(pop)操作需要遵循 LIFO 原则。本题要求判断给定的两个序列是否能够通过一个空栈的入栈和出栈操作得到。

首先,根据题目条件“每个序列中的值都不重复”可以推断出如果序列 pushed 和 popped 是有效的栈操作序列,那么在完成所有的入栈和出栈操作后,栈应该为空,因为每个元素都恰好入栈和出栈了一次。

为了验证这一点,可以模拟整个入栈和出栈的过程。

(1) 模拟入栈操作:遍历 pushed 序列,将每个元素依次压入模拟栈 st 中。这一步骤直接反映了题目中的入栈操作。

(2) 模拟出栈操作:在模拟入栈的同时,需要检查是否可以模拟出栈操作。由于栈的特性,只有栈顶元素可以被弹出。因此,检查栈顶元素是否与 popped 序列中的当前元素(可定义索引 idx 来指示)相匹配。如果匹配,将栈顶元素弹出,并在 popped 序列中前进到下一个元素(idx+=1)。

(3) 优化出栈检查:在弹出栈顶元素后,继续检查栈顶是否还有与 popped 中当前元素相匹配的元素,如果有,继续弹出操作,直到栈顶元素与 popped 中的当前元素不匹配或栈为空为止。这样做可以确保我们尽可能地模拟连续的出栈操作。

(4) 验证序列有效性:遍历完 pushed 序列后,检查模拟栈 st 是否为空。如果为空,说明所有元素都已按照 popped 序列的顺序成功出栈,返回 True 表示序列有效。如果栈不为空,则说明存在无法按 popped 序列的顺序出栈的元素,返回 False 表示序列无效。

【参考代码】

```
def validateStackSequences(self, pushed: List[int], popped: List[int]) -> bool:
    st = []            #用一个栈 st 来模拟元素推入和弹出过程
    idx = 0            #指示 popped 序列中“当前元素”的位置
    for e in pushed:
        st.append(e)
        while st and st[-1] == popped[idx]:
            st.pop()
            idx += 1
    return st == []
```

5.2 栈的应用举例

因其具有后进先出的固有特性,栈常常被作为算法或程序中的辅助存储结构,临时保存信息,供后面的操作使用。对于那些本质上符合后进先出原则的问题,栈提供了一种自然而直观的解决途径。它是算法和程序设计中的有用工具。本节将通过若干具体的例子,展示栈在各种计算任务中的应用。通过这些实例,读者将了解到栈如何在实际编程中发挥作用,以及如何有效地利用栈来设计解决方案。

5.2.1 括号匹配问题

括号匹配是编程和数学等领域中常见的问题。在编写程序代码或数学公式时，括号的使用至关重要。每种开括号，如圆括号“(”或方括号“[”，都需有一个对应的闭括号，如圆括号“)”或方括号“]”，以形成有效的配对。

在处理括号匹配时，不仅需要确保每一种类型的括号都有对应的闭合，还需要保证它们之间的嵌套关系是正确的。例如，在数学表达式中，复杂的公式可能包含多层嵌套的括号，它们的匹配对于表达式的正确理解和计算至关重要。

简单起见，此处以仅包含圆括号和方括号的表达式来阐述什么是正确的括号匹配。显然，单独的一对圆括号或者方括号是正确匹配的。进一步，如果一个括号序列本身是正确匹配的，那么在其外层添加一对相同类型的括号，或者将两个正确匹配的括号序列首尾相连，所得到的新序列也是正确匹配的。

相反，不正确的括号匹配会表现为以下几种情况。

(1) 类型不匹配：序列中的闭括号与相应的开括号类型不符。例如，序列“[( ]]”中，在第三个位置上，期望的圆括号“)”未出现，而是出现了一个方括号“]”。

(2) 多余闭括号：序列中出现了无法匹配的额外闭括号。例如，在序列“[( )])”中，最后的闭括号“)”没有对应的开括号；再如，在序列“)(”中，一开始就出现了一个不匹配的闭括号。

(3) 缺少闭括号：序列中的开括号没有对应的闭括号。如序列“([ ]( )”中，第一个圆括号“(”缺少闭合。

从刚才的分析中可以发现，当闭括号出现的时候，要么就是匹配了某个开括号，要么就可以得出匹配失败的结论。现在，以下述括号序列为例，来研究一下具体的判定算法。

(	(	[	]	[	]	)	)
1	2	3	4	5	6	7	8

可以考虑从头开始扫描括号序列。对于本例，第 1 个字符是开括号，暂且不管它，继续下一个字符，仍然是开括号，暂且不管它，然后又是开括号，依然不管。接下来是闭括号。当闭括号出现的时候，就应该去检查一下它是不是匹配的，也就是去看看前面离它“最近”的一个开括号是否与它匹配。至此可以发现，之前经过的开括号不仅需要记录下来，而且应该按顺序记录下来，因为现在需要的是最后经过的开括号。对于开括号的存储和使用顺序，具备鲜明的“后进先出”的特征。

所以，可以考虑在程序中设置一个栈，用来记录扫描过程中经过的开括号。在这个例子中，先依次把 1、2、3 处的开括号都入栈，等待配对使用。然后当碰到 4 处的闭括号时，直接将它和栈顶的开括号进行配对检查。配对成功，就意味着栈顶的这个开括号无须再等待配对了，因此把它出栈。这时，2 处的开括号就成了新的栈顶，表示接下来最急着配对的开括号就是它了。

接下来，继续扫描。遇到的是 5 处的开括号，把它入栈，它成为新的栈顶，现在若想使序列配对成功，最需要的是与它配对的闭括号“]”。原来的栈顶所期待的圆闭括号的等待急迫程度就下降了一级。如果这时来了圆闭括号，就会导致匹配失败。所以，在这个问题中使用

栈，会自动地调整处于等待中的开括号们的配对急迫程度。

继续扫描到 6 处闭括号，配对成功，出栈；继续扫描到 7 处闭括号，配对成功，出栈；最后扫描到 8 处闭括号，配对成功，出栈。至此，字符串扫描完毕，栈也空了，意味着刚刚好，既没有多余的闭括号，也没有仍在等待的开括号，括号匹配成功。

再思考以下两种情况：

(1) 如果在扫描的过程中，扫描到一个闭括号，但是此时栈已空，意味着什么？

(2) 如果字符串扫描完毕了，但栈不是空的，意味着什么？

显然，上述第(1)种情况意味着“多余闭括号”，而第(2)种情况意味着“缺少闭括号”，它们都应被判定为括号匹配失败。

综上，可以得到括号匹配问题求解的基本算法如下：

(1) 从头开始顺序遍历表达式中的所有字符，对于每个遇到的开括号，将其推入栈中。

(2) 对于每个闭括号，检查栈顶元素是否与其匹配。如果遇到栈空或类型不匹配的情况，括号匹配失败；否则弹出栈顶元素。

(3) 如果字符串遍历结束时栈为空，则表明所有括号都已正确匹配，括号匹配成功；否则表明栈中剩余的开括号没有对应的闭括号，括号匹配失败。

给定一个只包括“(”,“)”,“{”,“}”,“[”,“]”的括号字符串 s，检验其是否为有效括号串的具体算法实现可以参考代码清单 5.3。

代码清单 5.3　判断括号字符串是否有效

```
def isValid(s):
    d = {")": "(", "]": "[", "}": "{"}
    st = []
    for c in s:
        if c in "([{":
            st.append(c)
        else:
            if st and d[c] == st[-1]:
                st.pop()
            else:
                return False
    if st:
        return False
    else:
        return True
```

例 5.2　(力扣 1249)移除无效的括号。

【题目描述】

给出一个由“(”,“)”和小写字母组成的字符串 s，设计算法从 s 中删除最少数目的“(”或者“)”，可以删除任意位置的括号，使得剩下的字符串中的括号是匹配的，返回操作后的字符串。

【解题思路】

本问题可以分解为如下两个子问题。

(1) 确定 s 中所有需要删除的括号的索引，存放在一个集合 pos 中。

(2) 遍历 s 中所有的字符，如果其索引不在 pos 中，则将其添加到结果字符串中。

可以把第一个子问题转换为：在判定字符串 s 中的括号是否匹配的过程中，当出现匹

配失败的情况时，就往 pos 中添加失配括号的索引。在本题中，因为仅有圆括号这种类型的括号，不正确的括号匹配只会表现为“多余闭括号”或“缺少闭括号”。在多余闭括号时，失配的括号是当前到来的闭括号；而在“缺少闭括号”时，失配的括号是栈中剩余的开括号。因此，栈中应存放括号的索引，而不是括号本身，这样有利于获得失配括号的索引。

【参考代码】

```
def minRemoveToMakeValid(self, s: str) -> str:
    st = []
    pos = set()
    for i, c in enumerate(s):
        if c not in "()":
            continue
        if c == "(":
            st.append(i)
        else:
            if st:        #栈非空，则括号匹配
                st.pop()
            else:         #栈空，说明此处到来的闭括号是多余的，添加其索引至 pos
                pos.add(i)
    if st:                #栈非空，说明缺少闭括号，将栈中剩余的开括号索引合并到 pos 中
        pos = set(st) | pos
    ans = ""
    for i, c in enumerate(s):
        if i not in pos: #仅保留串 s 中位置不在 pos 中的字符
            ans += c
    return ans
```

5.2.2 后缀表达式求值

任何一个表达式都由操作数、运算符和界限符组成。其中，操作数可以是常数，也可以是被说明为变量或常量的标识符。运算符主要有算术运算符、关系运算符和逻辑运算符三类。基本界限符有左右括号和表达式结束符等。为了叙述简洁，在此仅限于讨论只含加、减、乘、除 4 种二元运算符的算术表达式。例如：

```
(3 - 5) * (6 + 17 * 4) / 3
```

书写这种式子，我们从小到大就有一种习惯：将两个操作数分别写在运算符的左右两侧，即运算符位于两个操作数的中间。在计算机科学中，将这种式子称为中缀表达式。在中缀表达式中，可以通过引入括号来改变运算的优先级（括号内的先计算）。

中缀表达式适合于人类阅读，但在使用计算机来求值时，括号的引入会增加处理的难度。在计算机处理时，更适用的是被称为后缀表达式（也称为逆波兰式）的写法——在这种写法中，运算符总写在它们的运算对象之后。对于上面的例子，其后缀表达式为

```
3 5 - 6 17 4 * + * 3 /
```

可以看到，在后缀表达式中没有括号。在后缀表达式中，运算符总写在它们的运算对象之后。在对后缀表达式进行计算求值时，对于每个运算符，就需要用它前面、最靠近它的两个数作为操作数进行运算。因此，计算需要从前往后进行。碰到操作数时先记录下来，碰到

运算符就需要用刚刚记录的两个操作数进行计算。计算的结果也需要记录下来，因为它可能是下一个运算符的某个操作数。

因此，在这个算法中，需要存储碰到的操作数或者计算结果。这些存储下来的数据，在使用时，具有“后进先出”的特点。因此，宜选用栈来存放这些数据。

例 5.3 （力扣 150）逆波兰式求值。

【题目描述】

给定一个字符串数组 tokens，其中元素从前到后依次代表一个逆波兰式由左到右的各个部分。设计算法计算该逆波兰式的值并返回。

注意：

（1）有效的运算符为'+'、'-'、'*'和'/'。

（2）每个操作数（运算对象）都可以是一个整数或者另一个表达式。

（3）两个整数之间的除法总是向零截断。

（4）表达式中不含除零运算。

（5）答案及所有中间计算结果可以用整数表示。

例如，当 tokens=["2","1","+","3","*"]时，其对应的中缀算术表达式为(2+1)*3，计算的结果为 9。

又如，当 tokens=["10","6","9","3","+","-11","*","/","*","17","+","5","+"]时，其对应的中缀算术表达式为((10*(6/((9+3)*-11)))+17)+5，计算的结果为 22。

【解题思路】

准备一个空栈（可以是 SStack 或 LStack 类的对象，也可以直接使用列表来表示）。从头开始，依次处理列表中的每个元素：如果是操作数，就仅需将它入栈；如果是运算符，则需要连续两次出栈，第一次出栈的元素作为右操作数，第二次出栈的元素作为左操作数，根据运算符的类型选择进行相应的计算，将计算结果入栈。这样循环操作直至处理完列表中的所有元素。此时，栈中一定有且仅有一个元素，它的值就是所求的表达式的值。

【参考代码】

```
def evalRPN(self, tokens: List[str]) -> int:
    st = []
    for x in tokens:
        if x not in "+-*/":
            st.append(int(x))  #栈中存放的是整数形式的操作数
        else:
            a = st.pop()
            b = st.pop()
            if x == "+":
                res = b + a
            elif x == "-":
                res = b - a
            elif x == "*":
                res = b * a
            else:
                res = int(b / a)
            st.append(res)
    return st.pop()
```