Assignment #6: 链表、栈和排序

Updated 2143 GMT+8 Oct 13, 2025

2025 fall, Complied by 林奕妃、环境科学与工程学院

> **说明：**
>
> 1. **解题与记录：**
>
>   对于每一个题目，请提供其解题思路（可选），并附上使用Python或C++编写的源代码（确保已在OpenJudge， Codeforces，LeetCode等平台上获得Accepted）。请将这些信息连同显示“Accepted”的截图一起填写到下方的作业模板中。（推荐使用Typora https://typoraio.cn 进行编辑，当然你也可以选择Word。）无论题目是否已通过，请标明每个题目大致花费的时间。
>
> 2. **提交安排：**提交时，请首先上传PDF格式的文件，并将.md或.doc格式的文件作为附件上传至右侧的“作业评论”区。确保你的Canvas账户有一个清晰可见的本人头像，提交的文件为PDF格式，并且“作业评论”区包含上传的.md或.doc附件。
>
> 3. **延迟提交：**如果你预计无法在截止日期前提交作业，请提前告知具体原因。这有助于我们了解情况并可能为你提供适当的延期或其他帮助。
>
> 请按照上述指导认真准备和提交作业，以保证顺利完成课程要求。

1. 题目

E24588: 后序表达式求值（25分钟）

Stack, http://cs101.openjudge.cn/practice/24588/

思路：使用栈存放操作数，遍历表达式，判断元素。如果当前元素是数字，将其转换成浮点数，然后压入栈中。如果当前元素是运算符，从栈顶弹出两个操作数，第一个弹出的数是第二个操作数（右操作数），第二个弹出的数是第一个操作数（左操作数）。当表达式中的所有元素都处理完毕后，栈中应该只剩下一个数字，这个数字就是整个表达式的最终计算结果。将它从栈中弹出即可。

代码：

```
def evaluate_postfix(expression):

    # 定义一个集合，方便快速判断一个元素是否为运算符
    operators = {'+', '-', '*', '/'}

    # 使用 Python 的列表来模拟栈
    stack = []

    # 将输入的表达式字符串按空格分割成一个元素列表
    tokens = expression.split()
```

```
    # 从左到右遍历每一个元素
    for token in tokens:
        # 判断当前 token 是否是运算符
        if token in operators:
            # 如果是运算符，就从栈顶弹出两个操作数
            # 注意：先弹出的是第二个操作数 (op2)
            op2 = stack.pop()
            # 后弹出的是第一个操作数 (op1)
            op1 = stack.pop()

            # 根据不同的运算符执行相应的计算
            if token == '+':
                result = op1 + op2
            elif token == '-':
                result = op1 - op2
            elif token == '*':
                result = op1 * op2
            elif token == '/':
                # 题目保证表达式有效，一般不需要考虑除以零的情况
                result = op1 / op2

            # 将计算结果压回栈中
            stack.append(result)
        else:
            # 如果 token 不是运算符，那它就是操作数
            # 将字符串形式的数字转换为浮点数，然后压入栈中
            # 使用 float() 可以同时处理整数和小数
            stack.append(float(token))

    # 遍历完所有元素后，栈中剩下的唯一一个元素就是最终结果
    # 弹出这个结果并返回
    return stack.pop()

# 主程序入口
if __name__ == "__main__":
    try:
        # 读取第一行的整数 n，表示接下来有多少个测试用例
        n = int(input())

        # 循环 n 次，处理每一个测试用例
        for _ in range(n):
            # 读取一整行的后缀表达式
            expression_line = input()

            # 调用函数计算表达式的值
            final_value = evaluate_postfix(expression_line)

            # 按题目要求，格式化输出结果，保留两位小数
            # f-string 的格式化功能 f"{value:.2f}" 非常方便
            print(f"{final_value:.2f}")

    except (IOError, ValueError) as e:
        # 异常处理，防止因输入格式错误等问题导致程序崩溃
        pass
```

代码运行截图 （至少包含有"Accepted"）

M234.回文链表（20分钟）

linked list, https://leetcode.cn/problems/palindrome-linked-list/

请用快慢指针实现 `O(1)` 空间复杂度。

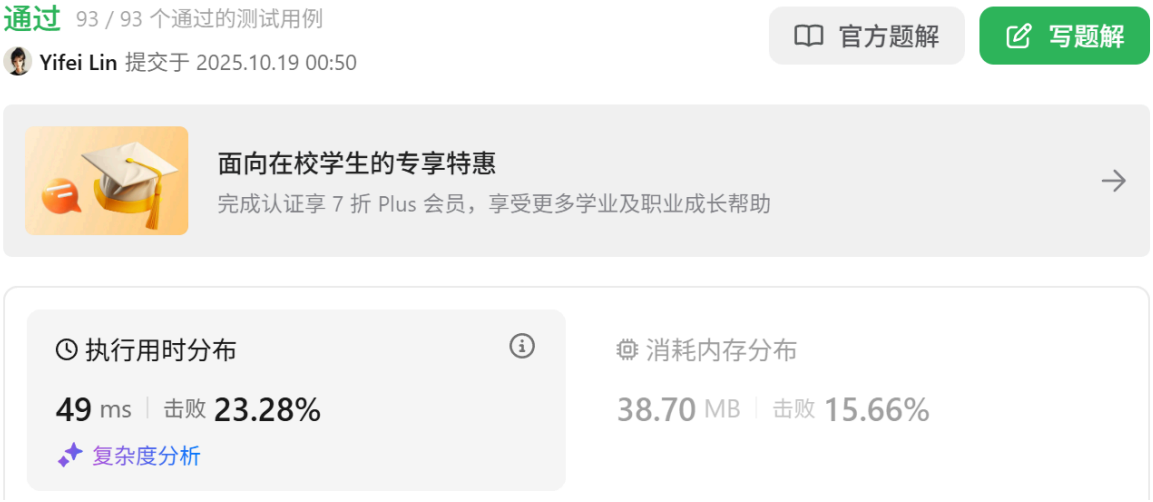
思路：快慢指针，快指针一次走两个，慢指针一次一个，当快指针到达链表尾的时候慢指针位于链表中间，将后半段链表反转，比较前半部分和反转后的后半部分。

代码：

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        """
        判断一个单链表是否为回文链表
        :param head: 链表的头节点
        :return: 如果是回文链表，返回 True；否则，返回 False
        """
        # 边界情况：空链表或只有一个节点的链表一定是回文的
        if not head or not head.next:
            return True

        # 使用快慢指针找到链表的中间节点
        # slow 最终会指向前半部分的末尾节点
        # fast 用于探路，每次走两步
        slow, fast = head, head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next

        # 反转后半部分链表
        # second_half_head 是后半部分的头节点，从 slow 的下一个节点开始
        second_half_head = self.reverse_list(slow.next)
```

```
        # 比较前半部分和反转后的后半部分
        # first_half_pointer 指向前半部分的头
        first_half_pointer = head
        # second_half_pointer 指向反转后后半部分的头
        second_half_pointer = second_half_head

        # 默认结果为 True，如果在比较中发现不一致，则修改为 False
        is_palindrome_result = True
        while second_half_pointer:
            if first_half_pointer.val != second_half_pointer.val:
                is_palindrome_result = False
                break  # 发现不匹配，立即退出循环
            first_half_pointer = first_half_pointer.next
            second_half_pointer = second_half_pointer.next

        # 将已经反转的后半部分再次反转，恢复原状
        # 然后将其接回到前半部分的末尾
        slow.next = self.reverse_list(second_half_head)

        return is_palindrome_result

    def reverse_list(self, node: ListNode) -> ListNode:
        """
        一个辅助函数，用于反转一个链表，并返回反转后新的头节点。
        这是一个标准的迭代法反转链表的实现。
        """
        prev = None
        current = node
        while current:
            next_node = current.next  # 1. 暂存下一个节点
            current.next = prev       # 2. 当前节点指向前一个节点，完成反转
            prev = current            # 3. prev 指针后移
            current = next_node       # 4. current 指针后移
        return prev # 返回新的头节点 (原始链表的末尾节点)
```

代码运行截图 （至少包含有"Accepted"）



M27217: 有多少种合法的出栈顺序（40分钟）

http://cs101.openjudge.cn/practice/27217/

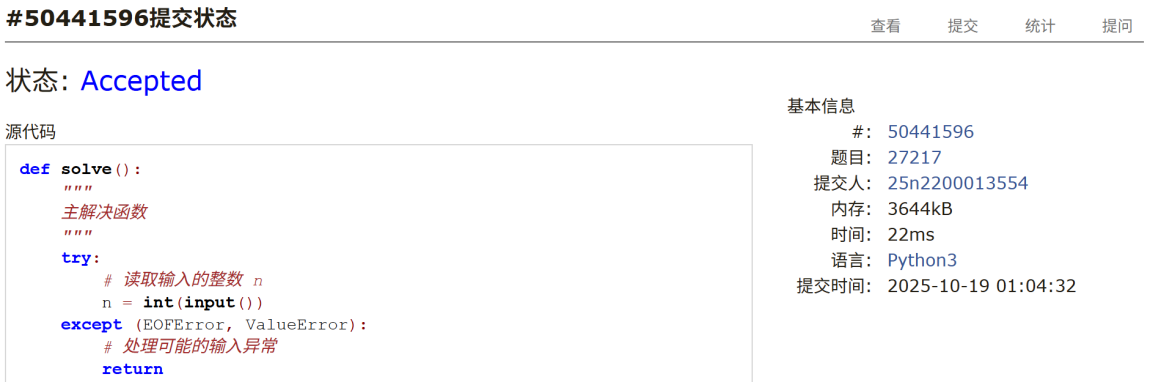
思路：对于 1, 2, ..., n 这个序列，我们总共需要执行 n 次入栈操作和 n 次出栈操作。入栈操作必须按顺序进行，出栈次数不能大于入栈次数。第 n 个卡特兰数 C_n 有多种计算公式，最常用的是：C_n = (1 / (n + 1)) * C(2n, n)。其中 C(2n, n) 是组合数，表示从 2n 个元素中选取 n 个元素的组合数量。C(2n, n) = (2n)! / (n! * n!)。所以，最终公式为：C_n = (2n)! / ((n + 1)! * n!)。如果不使用公式直接计算，还可以使用递归的思想进行动态规划。每进行一步后都可以选择让一个元素入栈或让一个元素出栈，将问题拆解为子问题。

公式直接计算代码：

```
def solve():
    """
    主解决函数
    """
    try:
        # 读取输入的整数 n
        n = int(input())
    except (EOFError, ValueError):
        # 处理可能的输入异常
        return

    # 卡特兰数的计算公式为 C_n = (1 / (n + 1)) * C(2n, n)
    # 分步计算以避免巨大的中间阶乘值
    # Python 的 int 类型可以处理任意大的整数，不必担心溢出

    # 1. 计算组合数 C(2n, n)
    # C(2n, n) = (2n * (2n-1) * ... * (n+1)) / n!

    # 计算分子: (2n) * (2n-1) * ... * (n+1)
    numerator = 1
    for i in range(n + 1, 2 * n + 1):
        numerator *= i

    # 计算分母: n!
    denominator = 1
    for i in range(1, n + 1):
        denominator *= i

    # C(2n, n) 的值
    # 使用整数除法 //
    combination_2n_n = numerator // denominator

    # 2. 计算卡特兰数 C_n
    # C_n = C(2n, n) / (n + 1)
    catalan_number = combination_2n_n // (n + 1)

    # 输出最终结果
    print(catalan_number)

if __name__ == "__main__":
```

```
    solve()
```

递归代码运行截图 （至少包含有"Accepted"）



#50441596提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
def solve():
    """
    主解决函数
    """
    try:
        # 读取输入的整数 n
        n = int(input())
    except (EOFError, ValueError):
        # 处理可能的输入异常
        return
```

基本信息

#: 50441596
题目: 27217
提交人: 25n2200013554
内存: 3644kB
时间: 22ms
语言: Python3
提交时间: 2025-10-19 01:04:32

代码

```
def solve_dp():
    """
    使用动态规划方法解决问题
    """
    try:
        n = int(input())
    except (EOFError, ValueError):
        return

    # 如果 n=0，只有一种可能（什么都不做），但题目保证 1 <= n <= 1000
    if n == 0:
        print(1)
        return

    # 创建一个 dp 数组来存储子问题的解
    # dp[i] 将存储 n=i 时的可能序列总数
    # 数组大小为 n+1，因为我们需要索引从 0 到 n
    dp = [0] * (n + 1)

    # 基础情况：n=0 时，有 1 种可能（空序列）
    dp[0] = 1

    # 迭代计算 dp[1], dp[2], ..., dp[n]
    # 外层循环计算每一个 dp[i]
    for i in range(1, n + 1):
        # 内层循环应用递推公式：
        # dp[i] = Σ (dp[j] * dp[i-1-j]) for j from 0 to i-1
        for j in range(i):
            dp[i] += dp[j] * dp[i - 1 - j]

    # 最终的答案就是 dp[n]
    print(dp[n])

if __name__ == "__main__":
    solve_dp()
```

(至少包含有"Accepted")

#50441612提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
def solve_dp():
    """
    使用动态规划方法解决问题
    """
    try:
        n = int(input())
    except (EOFError, ValueError):
        return
```

基本信息

#: 50441612
题目: 27217
提交人: 25n2200013554
内存: 3676kB
时间: 240ms
语言: Python3
提交时间: 2025-10-19 01:08:51

M24591:中序表达式转后序表达式（25分钟）

http://cs101.openjudge.cn/practice/24591/

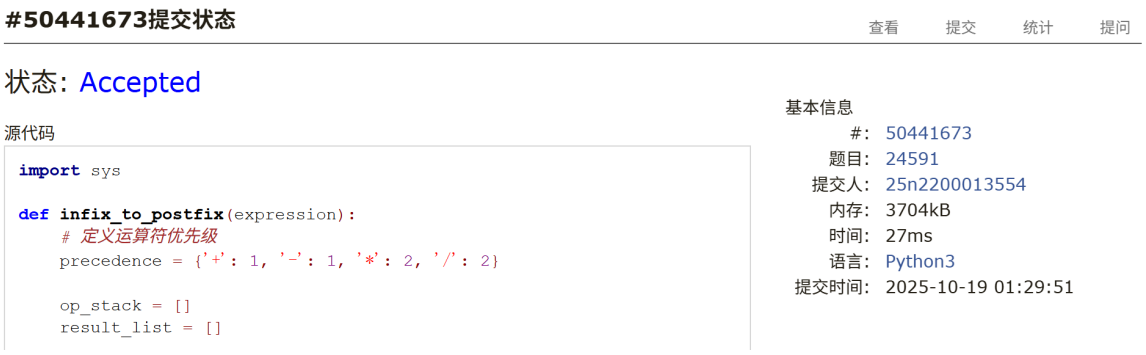
思路：当遇到括号时，需要将括号内的运算符号弹出。

代码

```
import sys

def infix_to_postfix(expression):
    # 定义运算符优先级
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}

    op_stack = []
    result_list = []

    i = 0
    while i < len(expression):
        char = expression[i]

        if char.isdigit() or char == '.':
            # --- 处理数字（可能是多位或小数）---
            num_str = ""
            while i < len(expression) and (expression[i].isdigit() or
expression[i] == '.'):
                num_str += expression[i]
                i += 1
            result_list.append(num_str)
            continue # 继续下一个循环，因为 i 已经前进了

        elif char == '(':
            op_stack.append(char)

        elif char == ')':
            # 弹出直到遇到 '('
            top_op = op_stack.pop()
            while top_op != '(':
                result_list.append(top_op)
```

```
                top_op = op_stack.pop()

        else: # 是运算符 + - * /
            # 弹出所有优先级更高或相等的运算符
            while (op_stack and op_stack[-1] != '(' and
                   precedence.get(op_stack[-1], 0) >= precedence.get(char, 0)):
                result_list.append(op_stack.pop())
            op_stack.append(char)

        i += 1

    # 将栈中剩余的运算符全部弹出
    while op_stack:
        result_list.append(op_stack.pop())

    return " ".join(result_list)

# --- 主程序 ---
def main():
    try:
        n = int(sys.stdin.readline())
        for _ in range(n):
            expression = sys.stdin.readline().strip()
            # 移除所有空格，方便处理
            expression = expression.replace(" ", "")
            postfix_expr = infix_to_postfix(expression)
            print(postfix_expr)
    except (ValueError, IndexError):
        return

if __name__ == "__main__":
    main()
```

（至少包含有"Accepted"）



#50441673提交状态

查看　提交　统计　提问

状态: Accepted

源代码

```
import sys

def infix_to_postfix(expression):
    # 定义运算符优先级
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}

    op_stack = []
    result_list = []
```

基本信息

#: 50441673
题目: 24591
提交人: 25n2200013554
内存: 3704kB
时间: 27ms
语言: Python3
提交时间: 2025-10-19 01:29:51

M02299:Ultra-QuickSort（30分钟）

merge sort, http://cs101.openjudge.cn/practice/02299/

思路：最小交换次数等于其逆序对的数量，利用归并排序计算逆序对。

代码

```
import sys

# 设置递归深度限制，防止大数据量时栈溢出
sys.setrecursionlimit(1000000)

def count_inversions_and_sort(arr):
    # 用来存储总逆序对数量的全局（或可变）变量
    # 这里用一个列表来传递引用，以便在递归中修改
    inversion_count = [0]

    def merge_sort(sub_arr):
        if len(sub_arr) <= 1:
            return sub_arr

        mid = len(sub_arr) // 2
        left = merge_sort(sub_arr[:mid])
        right = merge_sort(sub_arr[mid:])

        return merge(left, right)

    def merge(left, right):
        merged = []
        i, j = 0, 0

        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                merged.append(left[i])
                i += 1
            else: # left[i] > right[j]
                # 发现了逆序对
                # right[j] 比 left[i] 及之后的所有元素都小
                inversion_count[0] += len(left) - i

                merged.append(right[j])
                j += 1

        # 将剩余的元素加入
        merged.extend(left[i:])
        merged.extend(right[j:])

        return merged

    merge_sort(arr)
    return inversion_count[0]

# --- 主程序 ---
def main():
    while True:
        try:
            line = sys.stdin.readline()
            if not line:
                break
            n = int(line)
```

```
            if n == 0:
                break

            sequence = [int(sys.stdin.readline()) for _ in range(n)]

            result = count_inversions_and_sort(sequence)
            print(result)

        except (ValueError, IndexError):
            break

if __name__ == "__main__":
    main()
```

(至少包含有"Accepted")

#50441718提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys

# 设置递归深度限制，防止大数据量时栈溢出
sys.setrecursionlimit(1000000)

def count_inversions_and_sort(arr):
    # 用来存储总逆序对数量的全局（或可变）变量
    # 这里用一个列表来传递引用，以便在递归中修改
    inversion_count = [0]
```

基本信息

#: 50441718
题目: 02299
提交人: 25n2200013554
内存: 36396kB
时间: 3535ms
语言: Python3
提交时间: 2025-10-19 01:55:36

M146.LRU缓存（1小时）

hash table, doubly-linked list, https://leetcode.cn/problems/lru-cache/

思路：用一个哈希表 cache 来存储 key 到节点的映射。哈希表提供查找功能。双向链表用来实现操作功能。链表头部存放最近最少使用的节点。链表尾部存放最近刚刚使用的节点。

代码：

```
class DLinkedNode:
    """双向链表节点"""
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:

    def __init__(self, capacity: int):
        self.cache = {}
        self.capacity = capacity
        # 使用伪头部和伪尾部节点简化边界处理
        self.head = DLinkedNode()
```

```
        self.tail = DLinkedNode()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_to_tail(self, node: DLinkedNode):
        """将节点添加到双向链表的尾部"""
        node.prev = self.tail.prev
        node.next = self.tail
        self.tail.prev.next = node
        self.tail.prev = node

    def _remove_node(self, node: DLinkedNode):
        """从双向链表中删除一个节点"""
        node.prev.next = node.next
        node.next.prev = node.prev

    def _move_to_tail(self, node: DLinkedNode):
        """将一个已存在的节点移动到尾部"""
        self._remove_node(node)
        self._add_to_tail(node)

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1

        # 如果 key 存在，通过哈希表定位，然后将节点移动到尾部
        node = self.cache[key]
        self._move_to_tail(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            # 如果 key 存在，更新值并将节点移动到尾部
            node = self.cache[key]
            node.value = value
            self._move_to_tail(node)
        else:
            # 如果 key 不存在，创建一个新节点
            new_node = DLinkedNode(key, value)

            # 添加到哈希表和链表尾部
            self.cache[key] = new_node
            self._add_to_tail(new_node)

            # 检查是否超出容量
            if len(self.cache) > self.capacity:
                # 删除链表头部的节点（最久未使用）
                head_node = self.head.next
                self._remove_node(head_node)
                # 从哈希表中删除对应的 key
                del self.cache[head_node.key]

# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

代码运行截图（至少包含有"Accepted"）

2. 学习总结和个人收获

对栈的使用还不是很熟练，尤其是M27217: 有多少种合法的出栈顺序一开始没有读懂题目，后面用递归方法做感觉已经能找到一些感觉了，还是要多练习。后面发现因为输入输出简单只有一个数字而且这是卡特兰数，可以代入公式直接计算输出结果也满足题目的要求，但这个解法可能不是作业想要练习的。不过其公式的推导与迭代法的求解是相通的，也启发我有时候从数学角度也能把问题简化，进而实现代码的简化。例如后面M02299:Ultra-QuickSort思考如何交换，就需要知道最小交换次数等于其逆序对的数量，进而把问题转化为求逆序对数量。