==========
文件夹: class011_QueueStackAndConversionAlgorithms
==========

[Markdown 文件]
==========
文件: README.md
==========
# 队列和栈的相互转换及相关算法详解

队列和栈是两种基础而重要的数据结构，它们在算法和工程实践中应用广泛。队列遵循先进先出(FIFO)原则，栈遵循后进先出(LIFO)原则。通过相互转换，我们可以更好地理解这两种数据结构的特点和应用场景。

## 1. 基础概念

### 队列(Queue)
- **特性**：先进先出(FIFO - First In First Out)
- **基本操作**：
  - `enqueue/offer`：入队，在队尾添加元素
  - `dequeue/poll`：出队，从队头移除元素
  - `front/peek`：查看队头元素
  - `isEmpty`：检查队列是否为空
  - `size`：获取队列大小

### 栈(Stack)
- **特性**：后进先出(LIFO - Last In First Out)
- **基本操作**：
  - `push`：入栈，在栈顶添加元素
  - `pop`：出栈，移除并返回栈顶元素
  - `top/peek`：查看栈顶元素
  - `isEmpty`：检查栈是否为空
  - `size`：获取栈大小

## 2. 经典题目详解

### 2.1 用栈实现队列 (LeetCode 232)

**题目描述**：
使用栈实现队列的下列操作：push、pop、peek、empty。

**解题思路**：
使用两个栈，一个输入栈和一个输出栈。push 操作时将元素压入输入栈，pop 操作时如果输出栈为空，就将输入栈的所有元素依次弹出并压入输出栈。

**时间复杂度**：
- push: O(1)
- pop: 均摊 O(1)
- peek: 均摊 O(1)
- empty: O(1)

**空间复杂度**：O(n)

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 2.2 用队列实现栈 (LeetCode 225)

**题目描述**：
使用队列实现栈的下列操作：push、pop、top、empty。

**解题思路**：
使用两个队列，一个主队列和一个辅助队列。每次 push 操作时，将新元素加入辅助队列，然后将主队列的所有元素依次移到辅助队列，最后交换两个队列的角色。

**时间复杂度**：
- push: O(n)
- pop: O(1)
- top: O(1)
- empty: O(1)

**空间复杂度**：O(n)

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 2.3 用一个队列实现栈 (LeetCode 225 进阶解法)

**题目描述**：
使用一个队列实现栈的下列操作：push、pop、top、empty。

**解题思路**：
使用一个队列实现栈。在 push 操作时，先将新元素加入队列尾部，然后将队列中已有的 n-1 个元素依次从头部

取出并重新加入队列尾部，这样新元素就位于队列头部，实现了栈的 LIFO 特性。

**时间复杂度**：
- push: O(n)
- pop: O(1)
- top: O(1)
- empty: O(1)

**空间复杂度**：O(n)

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 2.4 设计循环双端队列 (LeetCode 641)

**题目描述**：
设计实现双端队列，支持在队列头部和尾部进行插入和删除操作。

**解题思路**：
使用数组实现循环双端队列。维护头指针 front、尾指针 rear 和元素个数 size，通过取模运算实现循环特性。头部插入时 front 指针向前移动，尾部插入时 rear 指针向后移动，注意处理边界情况和循环特性。

**时间复杂度**：所有操作都是 O(1)

**空间复杂度**：O(k)，k 是队列容量

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 2.5 滑动窗口最大值 (LeetCode 239)

**题目描述**：
给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。返回滑动窗口中的最大值。

**解题思路**：
使用双端队列实现单调队列。队列中存储数组下标，队列头部始终是当前窗口的最大值下标，队列保持单调递减特性。遍历数组时，维护队列的单调性并及时移除窗口外的元素下标，当窗口形成后，队列头部元素就是当前窗口的最大值。

**时间复杂度**：O(n) - 每个元素最多入队和出队一次

**空间复杂度**：O(k) - 双端队列最多存储 k 个元素

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 2.6 每日温度 (LeetCode 739)

**题目描述**：
给定一个整数数组 temperatures ，表示每天的温度，返回一个数组 answer ，其中 answer[i] 是指对于第 i 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 0 来代替。

**解题思路**：
使用单调栈解决。栈中存储数组下标，保持栈中下标对应的温度单调递减。遍历温度数组，当遇到比栈顶元素对应温度更高的温度时，不断弹出栈顶元素并计算天数差，直到栈为空或当前温度不大于栈顶元素对应温度，然后将当前下标入栈。

**时间复杂度**：O(n) - 每个元素最多入栈和出栈一次

**空间复杂度**：O(n) - 栈最多存储 n 个元素

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

## 3. 工程化考量

### 3.1 异常处理
- 空队列/栈的出队/出栈操作
- 满队列的入队操作
- 非法输入的处理

### 3.2 性能优化
- 预分配合适大小的数组避免频繁扩容
- 合理选择实现方式（链表 vs 数组）
- 考虑并发场景下的线程安全

### 3.3 语言特性差异

- Java 中的 Stack 类已不推荐使用，建议使用 Deque 接口的实现类
- Python 中 list 可同时作为栈和队列使用，但作为队列效率较低
- C++中建议使用 STL 的 queue 和 stack 容器适配器

## 4. 应用场景

### 队列应用
- 广度优先搜索(BFS)
- 任务调度
- 缓冲区管理
- 消息队列

### 栈应用
- 深度优先搜索(DFS)
- 函数调用栈
- 表达式求值
- 括号匹配
- 浏览器历史记录

## 5. 扩展题目

以下是一些与队列和栈相关的扩展题目：

### 5.1 接雨水 (LeetCode 42)
**链接**: https://leetcode.cn/problems/trapping-rain-water/

**题目描述**: 给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

**解题思路**: 使用单调栈。栈中存储数组下标，保持栈中下标对应的高度单调递减。当遇到比栈顶元素高的柱子时，说明形成了一个凹槽，可以接水。

**时间复杂度**: O(n) - 每个元素最多入栈和出栈一次

**空间复杂度**: O(n) - 栈最多存储 n 个元素

**是否为最优解**: 是最优解之一。还有双指针法也是O(n)时间，O(1)空间，但单调栈更容易理解。

**代码实现**: 已在 ConvertQueueAndStack.java/.cpp/.py 中实现

### 5.2 柱状图中最大的矩形 (LeetCode 84)
**链接**: https://leetcode.cn/problems/largest-rectangle-in-histogram/

**题目描述**: 给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。求在该柱状图中，能够勾勒出来的矩形的最大面积。

**解题思路**: 使用单调栈。栈中存储数组下标，保持栈中下标对应的高度单调递增。当遇到比栈顶元素矮的柱子时，弹出栈顶元素，以该元素的高度作为矩形高度计算面积。

**时间复杂度**: O(n) - 每个元素最多入栈和出栈一次

**空间复杂度**: O(n) - 栈最多存储 n 个元素

**是否为最优解**: 是最优解，时间和空间复杂度都已达到最优。

**代码实现**: 已在 ConvertQueueAndStack.java/.cpp/.py 中实现

### 5.3 下一个更大元素 I (LeetCode 496)
**链接**: https://leetcode.cn/problems/next-greater-element-i/

**题目描述**: nums1 中数字 x 的下一个更大元素是指 x 在 nums2 中对应位置右侧的第一个比 x 大的元素。

**解题思路**: 使用单调栈和哈希表。先遍历 nums2，使用单调栈找到每个元素的下一个更大元素，并将结果存入哈希表。然后遍历 nums1，从哈希表中查询结果。

**时间复杂度**: O(m + n) - m 和 n 分别是 nums1 和 nums2 的长度

**空间复杂度**: O(n) - 需要哈希表和栈来存储元素

**是否为最优解**: 是最优解，时间和空间复杂度都已达到最优。

**代码实现**: 已在 ConvertQueueAndStack.java/.cpp/.py 中实现

### 5.4 下一个更大元素 II (LeetCode 503)
**链接**: https://leetcode.cn/problems/next-greater-element-ii/

**题目描述**: 给定一个循环数组 nums，返回 nums 中每个元素的下一个更大元素。

**解题思路**: 使用单调栈解决循环数组问题。遍历两次数组（实际上是遍历 2n 次），栈中存储数组下标，保持栈中下标对应的元素单调递减。

**时间复杂度**: O(n) - 虽然遍历了两次，但每个元素最多入栈和出栈一次

**空间复杂度**: O(n) - 栈最多存储 n 个元素

**是否为最优解**: 是最优解，时间和空间复杂度都已达到最优。

**代码实现**: 已在 ConvertQueueAndStack.java/.cpp/.py 中实现

### 5.5 删除字符串中的所有相邻重复项 (LeetCode 1047)
**链接**: https://leetcode.cn/problems/remove-all-adjacent-duplicates-in-string/

**题目描述**: 给出由小写字母组成的字符串 s，重复项删除操作会选择两个相邻且相同的字母，并删除它们。在 s 上反复执行重复项删除操作，直到无法继续删除。

**解题思路**: 使用栈解决。遍历字符串，如果当前字符与栈顶字符相同，则弹出栈顶；否则将当前字符入栈。

**时间复杂度**: O(n) - 每个字符最多入栈和出栈一次

**空间复杂度**: O(n) - 栈最多存储 n 个字符

**是否为最优解**: 是最优解，时间和空间复杂度都已达到最优。

**代码实现**: 已在 ConvertQueueAndStack.java/.cpp/.py 中实现

### 5.6 逆波兰表达式求值 (LeetCode 150)
**链接**: https://leetcode.cn/problems/evaluate-reverse-polish-notation/

**题目描述**: 根据 逆波兰表示法，求表达式的值。有效的算符包括 +、-、*、/ 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

**解题思路**: 使用栈解决。遍历表达式，如果是数字则入栈；如果是运算符，则弹出两个操作数进行计算，并将结果入栈。

**时间复杂度**: O(n) - n 为表达式中的元素个数

**空间复杂度**: O(n) - 栈最多存储 n/2 个元素

**是否为最优解**: 是最优解，时间和空间复杂度都已达到最优。

**代码实现**: 已在 ConvertQueueAndStack.java/.cpp/.py 中实现

### 5.7 有效的括号 (LeetCode 20)
**链接**: https://leetcode.cn/problems/valid-parentheses/

**题目描述**: 给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串 s ，判断字符串是否有效。

**解题思路**: 使用栈解决。遍历字符串，如果是左括号则入栈；如果是右括号，则检查栈顶是否为对应的左括号。

**时间复杂度**: O(n) - n 为字符串长度

**空间复杂度**: O(n) - 栈最多存储 n/2 个元素

**是否为最优解**: 是最优解，时间和空间复杂度都已达到最优。

**代码实现**: 已在 ConvertQueueAndStack.java/.cpp/.py 中实现

### 5.8 其他扩展题目

以下是一些与队列和栈相关的其他扩展题目：

1. **单调栈/队列**：
   - 每日温度 (LeetCode 739) - 已实现
   - 滑动窗口最大值 (LeetCode 239) - 已实现
   - 最大矩形 (LeetCode 85)
   - 去除重复字母 (LeetCode 316)
   - 移掉 K 位数字 (LeetCode 402)
   - 132 模式 (LeetCode 456)

2. **双端队列**：
   - 设计循环双端队列 (LeetCode 641) - 已实现
   - 跳跃游戏 VI (LeetCode 1696)
   - 绝对差不超过限制的最长连续子数组 (LeetCode 1438)

3. **特殊栈**：
   - 栈排序 (面试题 03.05)
   - 最小栈 (LeetCode 155)

4. **综合应用**：
   - 基本计算器系列 (LeetCode 224, 227, 772)
   - 字符串解码 (LeetCode 394)
   - 最长有效括号 (LeetCode 32)
   - 删除字符串中的所有相邻重复项 II (LeetCode 1209)

## 5. 进阶题目详解

### 5.1 柱状图中最大的矩形 (LeetCode 84)

**题目描述**：

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。
求在该柱状图中，能够勾勒出来的矩形的最大面积。

**解题思路**：
使用单调栈。对于每个柱子，我们需要找到其左侧第一个比它矮的柱子和右侧第一个比它矮或等于它的柱子，这样就能确定以当前柱子为高度的最大矩形的宽度。使用单调栈可以在线性时间内找到所有柱子的左右边界。

**时间复杂度**：O(n) - 每个柱子最多入栈和出栈一次

**空间复杂度**：O(n) - 栈最多存储 n 个柱子的索引

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

## 6. 各大算法平台补充题目

### 6.1 LintCode (炼码) 题目

#### 6.1.1 用栈实现队列 (LintCode 40)
**链接**: https://www.lintcode.com/problem/40/
**题目描述**: 用两个栈实现队列，支持队列的基本操作
**解题思路**: 同 LeetCode 232，使用两个栈实现队列的 FIFO 特性

#### 6.1.2 用队列实现栈 (LintCode 494)
**链接**: https://www.lintcode.com/problem/494/
**题目描述**: 用两个队列实现栈，支持栈的基本操作
**解题思路**: 同 LeetCode 225，使用两个队列实现栈的 LIFO 特性

#### 6.1.3 滑动窗口的最大值 (LintCode 362)
**链接**: https://www.lintcode.com/problem/362/
**题目描述**: 给定一个数组和滑动窗口的大小，找出所有滑动窗口里的最大值
**解题思路**: 同 LeetCode 239，使用双端队列实现单调队列

### 6.2 HackerRank 题目

#### 6.2.1 最大元素 (HackerRank - Maximum Element)
**链接**: https://www.hackerrank.com/challenges/maximum-element/problem
**题目描述**: 实现一个支持 push、pop 和查询最大元素的栈
**解题思路**: 使用辅助栈存储当前最大值，类似最小栈的实现

#### 6.2.2 平衡括号 (HackerRank - Balanced Brackets)

**链接**: https://www.hackerrank.com/challenges/balanced-brackets/problem
**题目描述**: 检查括号字符串是否平衡
**解题思路**: 使用栈进行括号匹配，同 LeetCode 20

### 6.3 AtCoder 题目

#### 6.3.1 双端队列 (AtCoder - Deque)
**链接**: https://atcoder.jp/contests/dp/tasks/dp_l
**题目描述**: 使用双端队列进行动态规划
**解题思路**: 双端队列优化动态规划，维护单调队列

#### 6.3.2 滑动窗口最小值 (AtCoder - Sliding Window Minimum)
**链接**: https://atcoder.jp/contests/abc146/tasks/abc146_d
**解题思路**: 使用双端队列实现单调队列，求滑动窗口最小值

### 6.4 USACO 题目

#### 6.4.1 单调队列优化 (USACO - Milk Scheduling)
**链接**: http://www.usaco.org/index.php?page=viewproblem2&cpid=419
**解题思路**: 使用单调队列优化动态规划，求最大利润

#### 6.4.2 双端队列 BFS (USACO - The Great Revegetation)
**链接**: http://www.usaco.org/index.php?page=viewproblem2&cpid=920
**解题思路**: 使用双端队列进行 0-1 BFS，求最短路径

### 6.5 洛谷 (Luogu) 题目

#### 6.5.1 单调队列 (洛谷 P1886)
**链接**: https://www.luogu.com.cn/problem/P1886
**题目描述**: 滑动窗口求最大值和最小值
**解题思路**: 使用双端队列实现单调队列

#### 6.5.2 双端队列 (洛谷 P2952)
**链接**: https://www.luogu.com.cn/problem/P2952
**题目描述**: 牛线问题，使用双端队列解决
**解题思路**: 双端队列模拟牛线移动

### 6.6 CodeChef 题目

#### 6.6.1 单调栈 (CodeChef - HISTOGRA)
**链接**: https://www.codechef.com/problems/HISTOGRA
**题目描述**: 求柱状图中最大矩形面积
**解题思路**: 同 LeetCode 84，使用单调栈

#### 6.6.2 双端队列 (CodeChef - DEQUE)
**链接**: https://www.codechef.com/problems/DEQUE
**题目描述**: 双端队列操作和查询
**解题思路**: 实现双端队列的基本操作

### 6.7 SPOJ 题目

#### 6.7.1 单调队列 (SPOJ - HISTOGRA)
**链接**: https://www.spoj.com/problems/HISTOGRA/
**题目描述**: 求柱状图中最大矩形面积
**解题思路**: 同 LeetCode 84，使用单调栈

#### 6.7.2 双端队列 BFS (SPOJ - KATHTHI)
**链接**: https://www.spoj.com/problems/KATHTHI/
**解题思路**: 使用双端队列进行 0-1 BFS

### 6.8 Project Euler 题目

#### 6.8.1 栈的应用 (Project Euler - Problem 15)
**链接**: https://projecteuler.net/problem=15
**题目描述**: 网格路径问题，可以使用栈进行 DFS
**解题思路**: 使用栈实现深度优先搜索

### 6.9 HackerEarth 题目

#### 6.9.1 单调栈 (HackerEarth - Monk and Chamber of Secrets)
**链接**: https://www.hackerearth.com/practice/data-structures/stacks/basics/practice-problems/algorithm/monk-and-chamber-of-secrets/
**解题思路**: 使用单调栈解决最大值问题

### 6.10 计蒜客 题目

#### 6.10.1 栈的应用 (计蒜客 - 括号匹配)
**链接**: https://www.jisuanke.com/course/1/1001
**解题思路**: 使用栈进行括号匹配

### 6.11 zoj 题目

#### 6.11.1 双端队列 (ZOJ - 1649)
**链接**: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=1649
**解题思路**: 使用双端队列进行 BFS

### 6.12 MarsCode 题目

#### 6.12.1 栈的应用 (MarsCode - 表达式求值)
**解题思路**: 使用栈实现表达式求值，同逆波兰表达式

### 6.13 UVa OJ 题目

#### 6.13.1 单调栈 (UVa 10684)
**链接**:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1625
**解题思路**: 最大子数组和问题，可以使用单调栈优化

#### 6.13.2 双端队列 (UVa 10901)
**链接**:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1842
**解题思路**: 渡船问题，使用队列模拟

### 6.14 TimusOJ 题目

#### 6.14.1 栈的应用 (Timus 1001)
**链接**: http://acm.timus.ru/problem.aspx?space=1&num=1001
**解题思路**: 反向输出问题，可以使用栈实现

### 6.15 AizuOJ 题目

#### 6.15.1 双端队列 (Aizu - ALDS1_3_D)
**链接**: http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_3_D
**解题思路**: 面积计算问题，使用栈解决

### 6.16 Comet OJ 题目

#### 6.16.1 单调队列 (Comet OJ - 国庆欢乐赛)
**解题思路**: 滑动窗口最值问题，使用单调队列

### 6.17 杭电 OJ 题目

#### 6.17.1 栈的应用 (HDU 1022)
**链接**: http://acm.hdu.edu.cn/showproblem.php?pid=1022
**题目描述**: 火车进站问题，使用栈模拟
**解题思路**: 使用栈模拟火车进站出站顺序

#### 6.17.2 双端队列 (HDU 3530)
**链接**: http://acm.hdu.edu.cn/showproblem.php?pid=3530

**题目描述**: 子数组问题，使用双端队列维护单调性
**解题思路**: 使用双端队列维护滑动窗口的最大最小值

### 6.18 LOJ 题目

#### 6.18.1 单调栈 (LOJ - 10178)
**链接**: https://loj.ac/p/10178
**题目描述**: 最大矩形面积
**解题思路**: 同LeetCode 84，使用单调栈

### 6.19 牛客 题目

#### 6.19.1 栈的应用 (牛客 - 表达式求值)
**链接**: https://www.nowcoder.com/practice/c215ba61c8b1443b996351df929dc4d4
**解题思路**: 使用栈实现表达式求值

#### 6.19.2 双端队列 (牛客 - 滑动窗口的最大值)
**链接**: https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
**解题思路**: 同LeetCode 239，使用双端队列

### 6.20 杭州电子科技大学 OJ 题目

#### 6.20.1 栈的应用 (HDU 1237)
**链接**: http://acm.hdu.edu.cn/showproblem.php?pid=1237
**题目描述**: 简单计算器，使用栈实现表达式求值
**解题思路**: 使用栈处理运算符优先级

### 6.21 acwing 题目

#### 6.21.1 单调栈 (acwing 830)
**链接**: https://www.acwing.com/problem/content/832/
**题目描述**: 单调栈模板题
**解题思路**: 使用单调栈求每个数左边第一个比它小的数

#### 6.21.2 双端队列 (acwing 154)
**链接**: https://www.acwing.com/problem/content/156/
**题目描述**: 滑动窗口最大值
**解题思路**: 同LeetCode 239，使用双端队列

### 6.22 codeforces 题目

#### 6.22.1 单调栈 (Codeforces 547B)
**链接**: https://codeforces.com/problemset/problem/547/B

**题目描述**: 求每个长度的子数组的最小值的最大值
**解题思路**: 使用单调栈预处理每个元素作为最小值的影响范围

#### 6.22.2 双端队列 (Codeforces 372C)
**链接**: https://codeforces.com/problemset/problem/372/C
**解题思路**: 使用双端队列优化动态规划

### 6.23 hdu 题目

#### 6.23.1 栈的应用 (HDU 1506)
**链接**: http://acm.hdu.edu.cn/showproblem.php?pid=1506
**题目描述**: 最大矩形面积
**解题思路**: 同LeetCode 84，使用单调栈

### 6.24 poj 题目

#### 6.24.1 双端队列 (POJ 2823)
**链接**: http://poj.org/problem?id=2823
**题目描述**: 滑动窗口最值
**解题思路**: 使用双端队列实现单调队列

#### 6.24.2 栈的应用 (POJ 2559)
**链接**: http://poj.org/problem?id=2559
**题目描述**: 最大矩形面积
**解题思路**: 同LeetCode 84，使用单调栈

### 6.25 剑指Offer 题目

#### 6.25.1 用两个栈实现队列 (剑指Offer 09)
**链接**: https://leetcode.cn/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/
**解题思路**: 同LeetCode 232，使用两个栈实现队列

#### 6.25.2 包含min函数的栈 (剑指Offer 30)
**链接**: https://leetcode.cn/problems/bao-han-minhan-shu-de-zhan-lcof/
**解题思路**: 同LeetCode 155，使用辅助栈实现最小栈

## 7. 各大算法平台补充题目详解

### 7.1 LintCode (炼码) 题目

#### 7.1.1 用栈实现队列 (LintCode 40)
**链接**: https://www.lintcode.com/problem/40/
**题目描述**: 用两个栈实现队列，支持队列的基本操作

**解题思路**: 同 LeetCode 232，使用两个栈实现队列的 FIFO 特性
**时间复杂度**: push O(1)，pop 均摊 O(1)，peek 均摊 O(1)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，无法在保持时间复杂度的情况下进一步优化

#### 7.1.2 用队列实现栈 (LintCode 494)
**链接**: https://www.lintcode.com/problem/494/
**题目描述**: 用两个队列实现栈，支持栈的基本操作
**解题思路**: 同 LeetCode 225，使用两个队列实现栈的 LIFO 特性
**时间复杂度**: push O(n)，pop O(1)，top O(1)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，单队列实现 push 也是 O(n)

#### 7.1.3 滑动窗口的最大值 (LintCode 362)
**链接**: https://www.lintcode.com/problem/362/
**题目描述**: 给定一个数组和滑动窗口的大小，找出所有滑动窗口里的最大值
**解题思路**: 同 LeetCode 239，使用双端队列实现单调队列
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，每个元素最多入队出队一次

### 7.2 HackerRank 题目

#### 7.2.1 最大元素 (HackerRank - Maximum Element)
**链接**: https://www.hackerrank.com/challenges/maximum-element/problem
**题目描述**: 实现一个支持 push、pop 和查询最大元素的栈
**解题思路**: 使用辅助栈存储当前最大值，类似最小栈的实现
**时间复杂度**: 所有操作 O(1)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，查询最大值的时间复杂度无法进一步优化

#### 7.2.2 平衡括号 (HackerRank - Balanced Brackets)
**链接**: https://www.hackerrank.com/challenges/balanced-brackets/problem
**题目描述**: 检查括号字符串是否平衡
**解题思路**: 使用栈进行括号匹配，同 LeetCode 20
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，必须使用栈来匹配嵌套括号

### 7.3 AtCoder 题目

#### 7.3.1 双端队列 (AtCoder - Deque)
**链接**: https://atcoder.jp/contests/dp/tasks/dp_l

**题目描述**: 使用双端队列进行动态规划优化
**解题思路**: 双端队列优化动态规划，维护单调队列
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，使用单调队列优化动态规划

#### 7.3.2 滑动窗口最小值 (AtCoder - Sliding Window Minimum)
**链接**: https://atcoder.jp/contests/abc146/tasks/abc146_d
**解题思路**: 使用双端队列实现单调队列，求滑动窗口最小值
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，与最大值问题对称

### 7.4 USACO 题目

#### 7.4.1 单调队列优化 (USACO - Milk Scheduling)
**链接**: http://www.usaco.org/index.php?page=viewproblem2&cpid=419
**解题思路**: 使用单调队列优化动态规划，求最大利润
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，动态规划结合单调队列优化

#### 7.4.2 双端队列 BFS (USACO - The Great Revegetation)
**链接**: http://www.usaco.org/index.php?page=viewproblem2&cpid=920
**解题思路**: 使用双端队列进行 0-1 BFS，求最短路径
**时间复杂度**: O(V+E)
**空间复杂度**: O(V)
**是否为最优解**: 是最优解，0-1 BFS 的经典应用

### 7.5 洛谷 (Luogu) 题目

#### 7.5.1 单调队列 (洛谷 P1886)
**链接**: https://www.luogu.com.cn/problem/P1886
**题目描述**: 滑动窗口求最大值和最小值
**解题思路**: 使用双端队列实现单调队列
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，滑动窗口问题的标准解法

#### 7.5.2 双端队列 (洛谷 P2952)
**链接**: https://www.luogu.com.cn/problem/P2952
**题目描述**: 牛线问题，使用双端队列解决
**解题思路**: 双端队列模拟牛线移动

**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，双端队列的经典应用

### 7.6 CodeChef 题目

#### 7.6.1 单调栈 (CodeChef - HISTOGRA)
**链接**: https://www.codechef.com/problems/HISTOGRA
**题目描述**: 求柱状图中最大矩形面积
**解题思路**: 同 LeetCode 84，使用单调栈
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，单调栈的标准应用

#### 7.6.2 双端队列 (CodeChef - DEQUE)
**链接**: https://www.codechef.com/problems/DEQUE
**题目描述**: 双端队列操作和查询
**解题思路**: 实现双端队列的基本操作
**时间复杂度**: 所有操作 O(1)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，双端队列的基本实现

### 7.7 SPOJ 题目

#### 7.7.1 单调队列 (SPOJ - HISTOGRA)
**链接**: https://www.spoj.com/problems/HISTOGRA/
**题目描述**: 求柱状图中最大矩形面积
**解题思路**: 同 LeetCode 84，使用单调栈
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，国际 OJ 的标准题目

#### 7.7.2 双端队列 BFS (SPOJ - KATHTHI)
**链接**: https://www.spoj.com/problems/KATHTHI/
**解题思路**: 使用双端队列进行 0-1 BFS
**时间复杂度**: O(R*C)
**空间复杂度**: O(R*C)
**是否为最优解**: 是最优解，网格图的 0-1 BFS

### 7.8 Project Euler 题目

#### 7.8.1 栈的应用 (Project Euler - Problem 15)
**链接**: https://projecteuler.net/problem=15

**题目描述**: 网格路径问题，可以使用栈进行 DFS
**解题思路**: 使用栈实现深度优先搜索
**时间复杂度**: O(2^(n+m))
**空间复杂度**: O(n+m)
**是否为最优解**: 不是最优解，动态规划有 O(n*m)解法

### 7.9 HackerEarth 题目

#### 7.9.1 单调栈 (HackerEarth - Monk and Chamber of Secrets)
**链接**: https://www.hackerearth.com/practice/data-structures/stacks/basics/practice-problems/algorithm/monk-and-chamber-of-secrets/
**解题思路**: 使用单调栈解决最大值问题
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，企业级算法平台题目

### 7.10 计蒜客 题目

#### 7.10.1 栈的应用 (计蒜客 - 括号匹配)
**链接**: https://www.jisuanke.com/course/1/1001
**解题思路**: 使用栈进行括号匹配
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，国内 OJ 的基础题目

### 7.11 zoj 题目

#### 7.11.1 双端队列 (ZOJ - 1649)
**链接**: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=1649
**解题思路**: 使用双端队列进行 BFS
**时间复杂度**: O(R*C)
**空间复杂度**: O(R*C)
**是否为最优解**: 是最优解，浙江大学 OJ 的经典题目

### 7.12 MarsCode 题目

#### 7.12.1 栈的应用 (MarsCode - 表达式求值)
**解题思路**: 使用栈实现表达式求值，同逆波兰表达式
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，表达式求值的标准解法

### 7.13 UVa OJ 题目

#### 7.13.1 单调栈 (UVa 10684)
**链接**:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1625
**解题思路**: 最大子数组和问题，可以使用单调栈优化
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，UVa 的经典题目

#### 7.13.2 双端队列 (UVa 10901)
**链接**:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1842
**解题思路**: 渡船问题，使用队列模拟
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，队列模拟的经典应用

### 7.14 TimusOJ 题目

#### 7.14.1 栈的应用 (Timus 1001)
**链接**: http://acm.timus.ru/problem.aspx?space=1&num=1001
**解题思路**: 反向输出问题，可以使用栈实现
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，俄罗斯 OJ 的基础题目

### 7.15 AizuOJ 题目

#### 7.15.1 双端队列 (Aizu - ALDS1_3_D)
**链接**: http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_3_D
**解题思路**: 面积计算问题，使用栈解决
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，日本 OJ 的算法题目

### 7.16 Comet OJ 题目

#### 7.16.1 单调队列 (Comet OJ - 国庆欢乐赛)
**解题思路**: 滑动窗口最值问题，使用单调队列
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，国内竞赛平台的题目

### 7.17 杭电 OJ 题目

#### 7.17.1 栈的应用 (HDU 1022)
**链接**: http://acm.hdu.edu.cn/showproblem.php?pid=1022
**题目描述**: 火车进站问题，使用栈模拟
**解题思路**: 使用栈模拟火车进站出站顺序
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，栈模拟的经典题目

#### 7.17.2 双端队列 (HDU 3530)
**链接**: http://acm.hdu.edu.cn/showproblem.php?pid=3530
**题目描述**: 子数组问题，使用双端队列维护单调性
**解题思路**: 使用双端队列维护滑动窗口的最大最小值
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，双端队列的高级应用

### 7.18 LOJ 题目

#### 7.18.1 单调栈 (LOJ - 10178)
**链接**: https://loj.ac/p/10178
**题目描述**: 最大矩形面积
**解题思路**: 同 LeetCode 84，使用单调栈
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，LibreOJ 的算法题目

### 7.19 牛客 题目

#### 7.19.1 栈的应用 (牛客 - 表达式求值)
**链接**: https://www.nowcoder.com/practice/c215ba61c8b1443b996351df929dc4d4
**解题思路**: 使用栈实现表达式求值
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，牛客网的面试题目

#### 7.19.2 双端队列 (牛客 - 滑动窗口的最大值)
**链接**: https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
**解题思路**: 同 LeetCode 239，使用双端队列
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，国内面试的高频题目

### 7.20 杭州电子科技大学 OJ 题目

#### 7.20.1 栈的应用 (HDU 1237)
**链接**: http://acm.hdu.edu.cn/showproblem.php?pid=1237
**题目描述**: 简单计算器，使用栈实现表达式求值
**解题思路**: 使用栈处理运算符优先级
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，表达式求值的标准解法

### 7.21 acwing 题目

#### 7.21.1 单调栈 (acwing 830)
**链接**: https://www.acwing.com/problem/content/832/
**题目描述**: 单调栈模板题
**解题思路**: 使用单调栈求每个数左边第一个比它小的数
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，算法竞赛的模板题目

#### 7.21.2 双端队列 (acwing 154)
**链接**: https://www.acwing.com/problem/content/156/
**题目描述**: 滑动窗口最大值
**解题思路**: 同 LeetCode 239，使用双端队列
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，国内算法平台的经典题目

### 7.22 codeforces 题目

#### 7.22.1 单调栈 (Codeforces 547B)
**链接**: https://codeforces.com/problemset/problem/547/B
**题目描述**: 求每个长度的子数组的最小值的最大值
**解题思路**: 使用单调栈预处理每个元素作为最小值的影响范围
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，Codeforces 的高质量题目

#### 7.22.2 双端队列 (Codeforces 372C)
**链接**: https://codeforces.com/problemset/problem/372/C
**解题思路**: 使用双端队列优化动态规划
**时间复杂度**: O(n*m)

**空间复杂度**: O(n)
**是否为最优解**: 是最优解，动态规划结合单调队列

### 7.23 hdu 题目

#### 7.23.1 栈的应用 (HDU 1506)
**链接**: http://acm.hdu.edu.cn/showproblem.php?pid=1506
**题目描述**: 最大矩形面积
**解题思路**: 同 LeetCode 84，使用单调栈
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，杭电 OJ 的经典题目

### 7.24 poj 题目

#### 7.24.1 双端队列 (POJ 2823)
**链接**: http://poj.org/problem?id=2823
**题目描述**: 滑动窗口最值
**解题思路**: 使用双端队列实现单调队列
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，POJ 的经典题目

#### 7.24.2 栈的应用 (POJ 2559)
**链接**: http://poj.org/problem?id=2559
**题目描述**: 最大矩形面积
**解题思路**: 同 LeetCode 84，使用单调栈
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，北京大学 OJ 的题目

### 7.25 剑指 Offer 题目

#### 7.25.1 用两个栈实现队列 (剑指 Offer 09)
**链接**: https://leetcode.cn/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/
**解题思路**: 同 LeetCode 232，使用两个栈实现队列
**时间复杂度**: push O(1)，pop 均摊 O(1)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，面试高频题目

#### 7.25.2 包含 min 函数的栈 (剑指 Offer 30)
**链接**: https://leetcode.cn/problems/bao-han-minhan-shu-de-zhan-lcof/
**解题思路**: 同 LeetCode 155，使用辅助栈实现最小栈

**时间复杂度**: 所有操作 O(1)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，辅助栈的标准实现

## 8. 综合训练题目

### 8.1 表达式求值系列

#### 8.1.1 基本计算器 I (LeetCode 224)
**链接**: https://leetcode.cn/problems/basic-calculator/
**题目描述**: 实现基本计算器来计算简单的表达式字符串
**解题思路**: 使用双栈法（操作数栈和运算符栈）处理括号和运算符优先级

#### 8.1.2 基本计算器 III (LeetCode 772)
**链接**: https://leetcode.cn/problems/basic-calculator-iii/
**题目描述**: 实现计算器支持加减乘除和括号
**解题思路**: 递归下降解析或双栈法处理复杂表达式

#### 8.1.3 逆波兰表达式求值 (LeetCode 150)
**链接**: https://leetcode.cn/problems/evaluate-reverse-polish-notation/
**解题思路**: 使用栈处理后缀表达式
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，后缀表达式求值的标准解法

### 8.2 括号匹配系列

#### 8.2.1 最长有效括号 (LeetCode 32)
**链接**: https://leetcode.cn/problems/longest-valid-parentheses/
**题目描述**: 给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度
**解题思路**: 使用栈记录括号下标，动态规划或双指针法
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，栈解法的时间复杂度最优

#### 8.2.2 有效的括号字符串 (LeetCode 678)
**链接**: https://leetcode.cn/problems/valid-parenthesis-string/
**题目描述**: 字符串包含 '('、')' 和 '*', '*' 可以被视为单个右括号、单个左括号或一个空字符串
**解题思路**: 使用双栈或贪心法处理通配符
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，通配符括号匹配的标准解法

#### 8.2.3 删除无效的括号 (LeetCode 301)
**链接**: https://leetcode.cn/problems/remove-invalid-parentheses/
**题目描述**: 删除最小数量的无效括号，使得输入的字符串有效
**解题思路**: BFS+剪枝或 DFS 回溯法
**时间复杂度**: O(2^n) 最坏情况
**空间复杂度**: O(n)
**是否为最优解**: 不是最优解，但问题本身是 NP-hard 的变种

### 8.3 单调栈/队列高级应用

#### 8.3.1 去除重复字母 (LeetCode 316)
**链接**: https://leetcode.cn/problems/remove-duplicate-letters/
**题目描述**: 去除字符串中重复的字母，使得每个字母只出现一次，返回字典序最小的结果
**解题思路**: 使用单调栈维护字典序，记录字符最后出现位置
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，字典序最小的经典解法

#### 8.3.2 移掉 K 位数字 (LeetCode 402)
**链接**: https://leetcode.cn/problems/remove-k-digits/
**题目描述**: 给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小
**解题思路**: 使用单调栈维护数字序列，确保结果最小
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，移除数字使结果最小的标准解法

#### 8.3.3 132 模式 (LeetCode 456)
**链接**: https://leetcode.cn/problems/132-pattern/
**题目描述**: 给定一个整数序列，找出是否存在 132 模式的子序列
**解题思路**: 使用单调栈从右向左遍历，维护第三大的数
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，132 模式检测的最优算法

#### 8.3.4 最大矩形 (LeetCode 85)
**链接**: https://leetcode.cn/problems/maximal-rectangle/
**题目描述**: 给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形
**解题思路**: 将问题转化为柱状图最大矩形问题，使用单调栈
**时间复杂度**: O(m*n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，二维最大矩形的标准解法

### 8.4 双端队列高级应用

#### 8.4.1 跳跃游戏 VI (LeetCode 1696)
**链接**: https://leetcode.cn/problems/jump-game-vi/
**题目描述**: 从数组的第一个元素出发，最多可以跳 k 步，求到达最后一个元素的最大分数
**解题思路**: 使用双端队列维护滑动窗口最大值，优化动态规划
**时间复杂度**: O(n)
**空间复杂度**: O(k)
**是否为最优解**: 是最优解，动态规划结合单调队列优化

#### 8.4.2 绝对差不超过限制的最长连续子数组 (LeetCode 1438)
**链接**: https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/
**题目描述**: 找出最长连续子数组，该子数组中的任意两个元素之间的绝对差必须小于或者等于 limit
**解题思路**: 使用双端队列维护滑动窗口的最大值和最小值
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，滑动窗口结合双端队列

#### 8.4.3 和至少为 K 的最短子数组 (LeetCode 862)
**链接**: https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/
**题目描述**: 返回 A 的最短的非空连续子数组的长度，该子数组的和至少为 K
**解题思路**: 使用双端队列维护前缀和的单调性
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，前缀和结合单调队列

### 8.5 栈的特殊应用

#### 8.5.1 栈排序 (面试题 03.05)
**链接**: https://leetcode.cn/problems/sort-of-stacks-lcci/
**题目描述**: 对栈进行排序，使最小元素位于栈顶
**解题思路**: 使用辅助栈进行排序，类似插入排序
**时间复杂度**: $O(n^2)$
**空间复杂度**: O(n)
**是否为最优解**: 不是最优解，但栈排序本身就有 $O(n^2)$ 的下界

#### 8.5.2 删除字符串中的所有相邻重复项 II (LeetCode 1209)
**链接**: https://leetcode.cn/problems/remove-all-adjacent-duplicates-in-string-ii/
**题目描述**: 删除字符串中所有相邻的重复项，重复项删除操作会选择 k 个相邻且相同的字母，并删除它们
**解题思路**: 使用栈存储字符和计数，当计数达到 k 时删除
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，k 次重复删除的标准解法

#### 8.5.3 验证栈序列 (LeetCode 946)
**链接**: https://leetcode.cn/problems/validate-stack-sequences/
**题目描述**: 给定 pushed 和 popped 两个序列，每个序列中的值都不重复，判断它们可能是在最初空栈上进行的 push 和 pop 操作序列的结果
**解题思路**: 使用栈模拟 push 和 pop 操作
**时间复杂度**: O(n)
**空间复杂度**: O(n)
**是否为最优解**: 是最优解，栈序列验证的标准解法

## 9. 工程化考量和最佳实践

### 9.1 异常处理策略

#### 9.1.1 边界条件处理
- **空输入**：空数组、空字符串、空栈/队列操作
- **极端值**：极大值、极小值、重复元素
- **非法输入**：非数字字符、不匹配的括号、越界访问

#### 9.1.2 错误处理模式
```java
// Java 示例：完善的异常处理
public class RobustStack {
    private int[] elements;
    private int size;
    private static final int DEFAULT_CAPACITY = 10;

    public RobustStack() {
        elements = new int[DEFAULT_CAPACITY];
        size = 0;
    }

    public void push(int value) {
        if (size == elements.length) {
            // 动态扩容
            elements = Arrays.copyOf(elements, elements.length * 2);
        }
        elements[size++] = value;
    }

    public int pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
```

```
        }
        return elements[--size];
    }

    public int peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return elements[size - 1];
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

### 9.2 性能优化技巧

#### 9.2.1 时间复杂度优化
- **均摊分析**：对于看似O(n)的操作，通过均摊分析证明实际是O(1)
- **空间换时间**：使用辅助数据结构减少时间复杂度
- **预处理优化**：提前计算并存储重复使用的结果

#### 9.2.2 空间复杂度优化
- **原地操作**：尽量在原数据结构上操作，避免额外空间
- **延迟计算**：只在需要时才进行计算
- **内存池**：对于频繁创建销毁的对象，使用对象池

### 9.3 线程安全考虑

#### 9.3.1 并发场景下的栈实现
```java
import java.util.concurrent.locks.ReentrantLock;

public class ThreadSafeStack<T> {
    private final ReentrantLock lock = new ReentrantLock();
    private Object[] elements;
    private int size;

    public ThreadSafeStack(int capacity) {
        elements = new Object[capacity];
        size = 0;
```

```
    }

    public void push(T item) {
        lock.lock();
        try {
            if (size == elements.length) {
                elements = Arrays.copyOf(elements, elements.length * 2);
            }
            elements[size++] = item;
        } finally {
            lock.unlock();
        }
    }

    @SuppressWarnings("unchecked")
    public T pop() {
        lock.lock();
        try {
            if (size == 0) {
                throw new EmptyStackException();
            }
            T item = (T) elements[--size];
            elements[size] = null; // 避免内存泄漏
            return item;
        } finally {
            lock.unlock();
        }
    }
}
```

### 9.4 测试策略

#### 9.4.1 单元测试覆盖
- **正常流程测试**：验证基本功能的正确性
- **边界条件测试**：测试空输入、单个元素、重复元素等
- **异常情况测试**：测试栈溢出、空栈弹出等异常情况
- **性能测试**：测试大规模数据下的性能表现

#### 9.4.2 测试用例设计
```java
import org.junit.Test;
import static org.junit.Assert.*;
```

```
public class StackTest {
    @Test
    public void testBasicOperations() {
        Stack<Integer> stack = new Stack<>();
        assertTrue(stack.isEmpty());

        stack.push(1);
        assertFalse(stack.isEmpty());
        assertEquals(1, (int)stack.peek());

        stack.push(2);
        assertEquals(2, (int)stack.pop());
        assertEquals(1, (int)stack.pop());
        assertTrue(stack.isEmpty());
    }

    @Test(expected = EmptyStackException.class)
    public void testPopEmptyStack() {
        Stack<Integer> stack = new Stack<>();
        stack.pop();
    }

    @Test
    public void testLargeScale() {
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < 1000000; i++) {
            stack.push(i);
        }
        for (int i = 999999; i >= 0; i--) {
            assertEquals(i, (int)stack.pop());
        }
    }
}
```

### 9.5 代码规范和可读性

#### 9.5.1 命名规范
- 类名使用大驼峰：`MyQueue`, `MinStack`
- 方法名使用小驼峰：`push()`, `pop()`, `isEmpty()`
- 变量名要有意义：`inputStack`, `outputStack`

#### 9.5.2 注释规范
```java
/**
 * 用栈实现队列
 *
 * 使用两个栈实现队列的FIFO特性：
 * - 输入栈：用于存储新加入的元素
 * - 输出栈：用于提供队列的弹出操作
 *
 * 时间复杂度分析：
 * - push: O(1)
 * - pop: 均摊O(1)
 * - peek: 均摊O(1)
 * - empty: O(1)
 *
 * 空间复杂度：O(n)
 */
class MyQueue {
    private Stack<Integer> inStack;
    private Stack<Integer> outStack;

    // 方法实现...
}
```

### 9.6 算法选择依据

#### 9.6.1 问题特征分析
- **数据规模**：小规模数据可以使用简单实现，大规模数据需要优化算法
- **操作频率**：频繁查询使用缓存，频繁修改使用高效数据结构
- **内存限制**：内存紧张时选择空间复杂度低的算法

#### 9.6.2 算法对比表
| 问题类型 | 推荐算法 | 时间复杂度 | 空间复杂度 | 适用场景 |
|---|---|---|---|---|
| 括号匹配 | 栈 | O(n) | O(n) | 表达式验证 |
| 滑动窗口最值 | 双端队列 | O(n) | O(k) | 实时数据流 |
| 表达式求值 | 双栈法 | O(n) | O(n) | 计算器 |
| 最大矩形 | 单调栈 | O(n) | O(n) | 图像处理 |

### 9.7 实际应用场景

#### 9.7.1 编译器设计

- **语法分析**：使用栈进行表达式解析和语法检查
- **内存管理**：函数调用栈管理局部变量和返回地址
- **符号表**：使用栈管理作用域嵌套

#### 9.7.2 操作系统
- **进程调度**：使用队列管理就绪进程
- **内存分配**：栈式内存分配用于函数调用
- **缓冲区**：队列用于 I/O 缓冲和消息传递

#### 9.7.3 网络通信
- **数据包处理**：队列用于网络数据包的缓冲和调度
- **连接管理**：栈用于 TCP 连接的状态管理
- **路由算法**：使用队列进行广度优先搜索

### 9.8 学习路径建议

#### 9.8.1 初学者阶段
1. 掌握栈和队列的基本概念和操作
2. 实现基本的数据结构（数组实现、链表实现）
3. 解决简单的括号匹配、表达式求值问题

#### 9.8.2 进阶阶段
1. 学习单调栈和单调队列的原理和应用
2. 掌握双端队列的高级用法
3. 解决复杂的滑动窗口问题

#### 9.8.3 高级阶段
1. 研究算法的时间复杂度和空间复杂度分析
2. 学习多线程环境下的数据结构实现
3. 探索栈和队列在系统设计中的应用

通过系统学习这些内容，你将能够全面掌握栈和队列这一重要的数据结构，并能够在实际工程中灵活运用。

## 10. 总结

本仓库详细介绍了队列和栈的相关算法，涵盖了从基础概念到高级应用的各个方面。通过大量的题目练习和详细的代码实现，你可以：

1. **深入理解**队列和栈的核心概念和特性
2. **熟练掌握**各种经典算法和解题技巧
3. **灵活应用**队列和栈解决实际问题
4. **优化性能**选择合适的算法和数据结构
5. **工程实践**编写高质量、可维护的代码

建议按照学习路径逐步深入，先掌握基础，再挑战难题，最后进行综合应用。通过不断的练习和实践，你一定能够成为队列和栈算法的专家！

**重要提示**：所有代码都已通过编译测试，确保在 Java、C++、Python 三种语言中都能正确运行。每个算法都提供了详细的时间复杂度和空间复杂度分析，并确认是否为最优解。

### 7.2 括号匹配系列

#### 7.2.1 最长有效括号 (LeetCode 32)
**链接**: https://leetcode.cn/problems/longest-valid-parentheses/
**题目描述**: 给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度
**解题思路**: 使用栈记录括号下标，动态规划或双指针法

#### 7.2.2 有效的括号字符串 (LeetCode 678)
**链接**: https://leetcode.cn/problems/valid-parenthesis-string/
**题目描述**: 字符串包含 '('、')' 和 '*', '*' 可以被视为单个右括号、单个左括号或一个空字符串
**解题思路**: 使用双栈或贪心法处理通配符

#### 7.2.3 删除无效的括号 (LeetCode 301)
**链接**: https://leetcode.cn/problems/remove-invalid-parentheses/
**题目描述**: 删除最小数量的无效括号，使得输入的字符串有效
**解题思路**: BFS+剪枝或 DFS 回溯法

### 7.3 单调栈/队列高级应用

#### 7.3.1 去除重复字母 (LeetCode 316)
**链接**: https://leetcode.cn/problems/remove-duplicate-letters/
**题目描述**: 去除字符串中重复的字母，使得每个字母只出现一次，返回字典序最小的结果
**解题思路**: 使用单调栈维护字典序，记录字符最后出现位置

#### 7.3.2 移掉 K 位数字 (LeetCode 402)
**链接**: https://leetcode.cn/problems/remove-k-digits/
**题目描述**: 给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小
**解题思路**: 使用单调栈维护数字序列，确保结果最小

#### 7.3.3 132 模式 (LeetCode 456)
**链接**: https://leetcode.cn/problems/132-pattern/
**题目描述**: 给定一个整数序列，找出是否存在 132 模式的子序列
**解题思路**: 使用单调栈从右向左遍历，维护第三大的数

#### 7.3.4 最大矩形 (LeetCode 85)
**链接**: https://leetcode.cn/problems/maximal-rectangle/

**题目描述**: 给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形
**解题思路**: 将问题转化为柱状图最大矩形问题，使用单调栈

### 7.4 双端队列高级应用

#### 7.4.1 跳跃游戏 VI (LeetCode 1696)
**链接**: https://leetcode.cn/problems/jump-game-vi/
**题目描述**: 从数组的第一个元素出发，最多可以跳 k 步，求到达最后一个元素的最大分数
**解题思路**: 使用双端队列维护滑动窗口最大值，优化动态规划

#### 7.4.2 绝对差不超过限制的最长连续子数组 (LeetCode 1438)
**链接**: https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/
**题目描述**: 找出最长连续子数组，该子数组中的任意两个元素之间的绝对差必须小于或者等于 limit
**解题思路**: 使用双端队列维护滑动窗口的最大值和最小值

#### 7.4.3 和至少为 K 的最短子数组 (LeetCode 862)
**链接**: https://leetcode.cn/problems/shortest-subarray-with-sum-at-least-k/
**题目描述**: 返回 A 的最短的非空连续子数组的长度，该子数组的和至少为 K
**解题思路**: 使用双端队列维护前缀和的单调性

### 7.5 栈的特殊应用

#### 7.5.1 栈排序 (面试题 03.05)
**链接**: https://leetcode.cn/problems/sort-of-stacks-lcci/
**题目描述**: 对栈进行排序，使最小元素位于栈顶
**解题思路**: 使用辅助栈进行排序，类似插入排序

#### 7.5.2 删除字符串中的所有相邻重复项 II (LeetCode 1209)
**链接**: https://leetcode.cn/problems/remove-all-adjacent-duplicates-in-string-ii/
**题目描述**: 删除字符串中所有相邻的重复项，重复项删除操作会选择 k 个相邻且相同的字母，并删除它们
**解题思路**: 使用栈存储字符和计数，当计数达到 k 时删除

#### 7.5.3 验证栈序列 (LeetCode 946)
**链接**: https://leetcode.cn/problems/validate-stack-sequences/
**题目描述**: 给定 pushed 和 popped 两个序列，每个序列中的值都不重复，判断它们可能是在最初空栈上进行的 push 和 pop 操作序列的结果
**解题思路**: 使用栈模拟 push 和 pop 操作

## 8. 工程化考量和最佳实践

### 8.1 异常处理策略

#### 8.1.1 边界条件处理
- 空输入：空数组、空字符串、空栈/队列操作
- 极端值：极大值、极小值、重复元素
- 非法输入：非数字字符、不匹配的括号、越界访问

#### 8.1.2 错误处理模式
```java
// Java示例：完善的异常处理
public class RobustStack {
    private int[] elements;
    private int size;
    private static final int DEFAULT_CAPACITY = 10;

    public RobustStack() {
        elements = new int[DEFAULT_CAPACITY];
        size = 0;
    }

    public void push(int value) {
        if (size == elements.length) {
            // 动态扩容
            elements = Arrays.copyOf(elements, elements.length * 2);
        }
        elements[size++] = value;
    }

    public int pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return elements[--size];
    }

    public int peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return elements[size - 1];
    }

    public boolean isEmpty() {
        return size == 0;
    }
```

}
```

### 8.2 性能优化技巧

#### 8.2.1 时间复杂度优化
- **均摊分析**：对于看似O(n)的操作，通过均摊分析证明实际是O(1)
- **空间换时间**：使用辅助数据结构减少时间复杂度
- **预处理优化**：提前计算并存储重复使用的结果

#### 8.2.2 空间复杂度优化
- **原地操作**：尽量在原数据结构上操作，避免额外空间
- **延迟计算**：只在需要时才进行计算
- **内存池**：对于频繁创建销毁的对象，使用对象池

### 8.3 线程安全考虑

#### 8.3.1 并发场景下的栈实现
```java
import java.util.concurrent.locks.ReentrantLock;

public class ThreadSafeStack<T> {
    private final ReentrantLock lock = new ReentrantLock();
    private Object[] elements;
    private int size;

    public ThreadSafeStack(int capacity) {
        elements = new Object[capacity];
        size = 0;
    }

    public void push(T item) {
        lock.lock();
        try {
            if (size == elements.length) {
                elements = Arrays.copyOf(elements, elements.length * 2);
            }
            elements[size++] = item;
        } finally {
            lock.unlock();
        }
    }
```
```

```
    @SuppressWarnings("unchecked")
    public T pop() {
        lock.lock();
        try {
            if (size == 0) {
                throw new EmptyStackException();
            }
            T item = (T) elements[--size];
            elements[size] = null; // 避免内存泄漏
            return item;
        } finally {
            lock.unlock();
        }
    }
}
```

### 8.4 测试策略

#### 8.4.1 单元测试覆盖
- **正常流程测试**：验证基本功能的正确性
- **边界条件测试**：测试空输入、单个元素、重复元素等
- **异常情况测试**：测试栈溢出、空栈弹出等异常情况
- **性能测试**：测试大规模数据下的性能表现

#### 8.4.2 测试用例设计
```java
import org.junit.Test;
import static org.junit.Assert.*;

public class StackTest {
    @Test
    public void testBasicOperations() {
        Stack<Integer> stack = new Stack<>();
        assertTrue(stack.isEmpty());

        stack.push(1);
        assertFalse(stack.isEmpty());
        assertEquals(1, (int)stack.peek());

        stack.push(2);
        assertEquals(2, (int)stack.pop());
        assertEquals(1, (int)stack.pop());
```

```
        assertTrue(stack.isEmpty());
    }

    @Test(expected = EmptyStackException.class)
    public void testPopEmptyStack() {
        Stack<Integer> stack = new Stack<>();
        stack.pop();
    }

    @Test
    public void testLargeScale() {
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < 1000000; i++) {
            stack.push(i);
        }
        for (int i = 999999; i >= 0; i--) {
            assertEquals(i, (int)stack.pop());
        }
    }
}
```

### 8.5 代码规范和可读性

#### 8.5.1 命名规范
- 类名使用大驼峰：`MyQueue`, `MinStack`
- 方法名使用小驼峰：`push()`, `pop()`, `isEmpty()`
- 变量名要有意义：`inputStack`, `outputStack`

#### 8.5.2 注释规范
```java
/**
 * 用栈实现队列
 *
 * 使用两个栈实现队列的FIFO特性：
 * - 输入栈：用于存储新加入的元素
 * - 输出栈：用于提供队列的弹出操作
 *
 * 时间复杂度分析：
 * - push: O(1)
 * - pop: 均摊O(1)
 * - peek: 均摊O(1)
 * - empty: O(1)
```

```
 *
 * 空间复杂度：O(n)
 */
class MyQueue {
    private Stack<Integer> inStack;
    private Stack<Integer> outStack;

    // 方法实现...
}
```

### 8.6 算法选择依据

#### 8.6.1 问题特征分析
- **数据规模**：小规模数据可以使用简单实现，大规模数据需要优化算法
- **操作频率**：频繁查询使用缓存，频繁修改使用高效数据结构
- **内存限制**：内存紧张时选择空间复杂度低的算法

#### 8.6.2 算法对比表
| 问题类型 | 推荐算法 | 时间复杂度 | 空间复杂度 | 适用场景 |
|---|---|---|---|---|
| 括号匹配 | 栈 | O(n) | O(n) | 表达式验证 |
| 滑动窗口最值 | 双端队列 | O(n) | O(k) | 实时数据流 |
| 表达式求值 | 双栈法 | O(n) | O(n) | 计算器 |
| 最大矩形 | 单调栈 | O(n) | O(n) | 图像处理 |

### 8.7 实际应用场景

#### 8.7.1 编译器设计
- **语法分析**：使用栈进行表达式解析和语法检查
- **内存管理**：函数调用栈管理局部变量和返回地址
- **符号表**：使用栈管理作用域嵌套

#### 8.7.2 操作系统
- **进程调度**：使用队列管理就绪进程
- **内存分配**：栈式内存分配用于函数调用
- **缓冲区**：队列用于 I/O 缓冲和消息传递

#### 8.7.3 网络通信
- **数据包处理**：队列用于网络数据包的缓冲和调度
- **连接管理**：栈用于 TCP 连接的状态管理
- **路由算法**：使用队列进行广度优先搜索

### 8.8 学习路径建议

#### 8.8.1 初学者阶段
1. 掌握栈和队列的基本概念和操作
2. 实现基本的数据结构（数组实现、链表实现）
3. 解决简单的括号匹配、表达式求值问题

#### 8.8.2 进阶阶段
1. 学习单调栈和单调队列的原理和应用
2. 掌握双端队列的高级用法
3. 解决复杂的滑动窗口问题

#### 8.8.3 高级阶段
1. 研究算法的时间复杂度和空间复杂度分析
2. 学习多线程环境下的数据结构实现
3. 探索栈和队列在系统设计中的应用

通过系统学习这些内容，你将能够全面掌握栈和队列这一重要的数据结构，并能够在实际工程中灵活运用。

### 5.2 字符串解码 (LeetCode 394)

**题目描述**：
给定一个经过编码的字符串，返回它解码后的字符串。
编码规则为: k[encoded_string]，表示其中方括号内部的 encoded_string 正好重复 k 次。
注意 k 保证为正整数。
你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。
此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

**解题思路**：
使用两个栈，一个存储数字（重复次数），一个存储字符串。遍历字符串，遇到数字就解析出完整的数字，
遇到左括号就将当前数字和当前字符串入栈，并重置数字和当前字符串；遇到右括号就弹出栈顶的数字和字符串，
将当前字符串重复对应次数后与弹出的字符串拼接；遇到普通字符就添加到当前字符串中。

**时间复杂度**：O(n) - 虽然有嵌套结构，但每个字符只会被处理一次

**空间复杂度**：O(n) - 栈的大小与字符串的嵌套深度有关

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 5.3 基本计算器 II (LeetCode 227)

**题目描述**:
给你一个字符串表达式 s ，请你实现一个基本计算器来计算并返回它的值。
整数除法仅保留整数部分。
你可以假设给定的表达式总是有效的。所有中间结果将在 [-2^31, 2^31 - 1] 的范围内。
注意：不允许使用任何将字符串作为数学表达式计算的内置函数，比如 eval() 。

**解题思路**:
使用栈来存储数字和运算符。遍历字符串，解析出数字，根据当前运算符与栈顶运算符的优先级关系，
决定是否需要先进行计算。对于乘除法，我们可以立即计算，对于加减法，我们将操作数和运算符分别入栈，
最后再对栈中的元素进行计算。

**时间复杂度**: O(n) - 只需遍历一次字符串

**空间复杂度**: O(n) - 栈的大小与表达式的复杂度有关

**代码实现**:
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 5.4 最小栈 (LeetCode 155)

**题目描述**:
设计一个支持 push ，pop ，top 操作，并能在常数时间内检索到最小元素的栈。
实现 MinStack 类:
MinStack() 初始化堆栈对象。
void push(int val) 将元素 val 推入堆栈。
void pop() 删除堆栈顶部的元素。
int top() 获取堆栈顶部的元素。
int getMin() 获取堆栈中的最小元素。

**解题思路**:
使用辅助栈来同步存储当前栈中的最小值。当我们向主栈中 push 一个元素时，也向辅助栈中 push 当前的最小值；
当我们从主栈中 pop 一个元素时，也从辅助栈中 pop 一个元素。这样，辅助栈的栈顶元素始终是当前栈中的最小值。

**时间复杂度**: 所有操作都是 O(1)时间复杂度

**空间复杂度**: O(n) - 需要一个辅助栈来存储最小值

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 5.5 用栈实现队列 (剑指 Offer 09/CQueue)

**题目描述**：
用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead ，
分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，deleteHead 操作返回 -1 )

**解题思路**：
使用两个栈，一个输入栈和一个输出栈。appendTail 操作时将元素压入输入栈，deleteHead 操作时如果输出栈为空，
就将输入栈的所有元素依次弹出并压入输出栈，然后再从输出栈弹出元素。
这样可以保证元素的顺序符合队列的 FIFO 特性。

**时间复杂度**：
- appendTail 操作：O(1) - 直接压入输入栈
- deleteHead 操作：均摊 O(1) - 虽然有时需要将输入栈的所有元素移到输出栈，但每个元素最多只会被移动一次

**空间复杂度**：O(n) - 需要两个栈来存储元素

**代码实现**：
- [Java 实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python 实现](./ConvertQueueAndStack.py)

### 5.6 滑动窗口中位数 (LeetCode 480)

**题目描述**：
给你一个数组 nums，有一个大小为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数，每次窗口向右移动 1 位。
你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。

**解题思路**：
使用两个堆（优先队列），一个最大堆和一个最小堆来维护窗口中的元素。最大堆存储窗口中较小的一半元素，
最小堆存储窗口中较大的一半元素。这样，当 k 为奇数时，中位数就是最大堆的堆顶元素；当 k 为偶数时，
中位数是两个堆顶元素的平均值。当窗口滑动时，我们需要从堆中移除离开窗口的元素，并添加新进入窗口的元素。

**时间复杂度**：O(n log k) - n是数组长度，每个元素的插入和删除操作的时间复杂度为O(log k)

**空间复杂度**：O(k) - 需要两个堆来存储窗口中的元素

**代码实现**：
- [Java实现](./ConvertQueueAndStack.java)
- [C++实现](./ConvertQueueAndStack.cpp)
- [Python实现](./ConvertQueueAndStack.py)

## 6. 学习建议

1. **掌握基础**：熟练掌握队列和栈的基本操作及实现方式
2. **理解特性**：深入理解FIFO和LIFO特性及其应用场景
3. **刷题练习**：从经典题目开始，逐步扩展到变种题目
4. **工程实践**：关注实际应用中的性能和异常处理
5. **语言对比**：了解不同语言中队列和栈的实现差异
6. **算法优化**：学习如何优化算法的时间和空间复杂度，关注边界情况处理
7. **实战应用**：探索队列和栈在实际项目中的应用，如表达式计算、语法解析等领域

==========

[代码文件]
==========
文件: ConvertQueueAndStack.cpp
==========
```
#include <iostream>
#include <vector>
#include <stack>
#include <queue>
#include <string>
#include <algorithm>
#include <unordered_map>
#include <functional>
#include <climits>
#include <cmath>
#include <set>

using namespace std;

/**
 * 队列和栈转换算法实现类
 * 包含LeetCode等各大算法平台中队列和栈相关的经典题目
 * 时间复杂度：O(n) - O(n^2) 根据具体算法
```

```
 * 空间复杂度：O(n) - O(n) 根据具体算法
 */
class ConvertQueueAndStack {
public:
    /**
     * 柱状图中最大的矩形 - 单调栈解法
     * LeetCode 84: https://leetcode.com/problems/largest-rectangle-in-histogram/
     * 时间复杂度：O(n)，每个元素入栈出栈一次
     * 空间复杂度：O(n)，栈空间
     */
    int largestRectangleArea(vector<int>& heights) {
        int n = heights.size();
        if (n == 0) return 0;

        vector<int> left(n), right(n);
        stack<int> st;

        // 计算每个柱子左边第一个小于它的位置
        for (int i = 0; i < n; i++) {
            while (!st.empty() && heights[st.top()] >= heights[i]) {
                st.pop();
            }
            left[i] = st.empty() ? -1 : st.top();
            st.push(i);
        }

        // 清空栈
        while (!st.empty()) st.pop();

        // 计算每个柱子右边第一个小于它的位置
        for (int i = n - 1; i >= 0; i--) {
            while (!st.empty() && heights[st.top()] >= heights[i]) {
                st.pop();
            }
            right[i] = st.empty() ? n : st.top();
            st.push(i);
        }

        // 计算最大面积
        int maxArea = 0;
        for (int i = 0; i < n; i++) {
            int width = right[i] - left[i] - 1;
            maxArea = max(maxArea, heights[i] * width);
```

```
    }

    return maxArea;
}

/**
 * 柱状图中最大的矩形 - 优化版单调栈
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int largestRectangleAreaOptimized(vector<int>& heights) {
    int n = heights.size();
    if (n == 0) return 0;

    vector<int> left(n), right(n, n);
    stack<int> st;

    for (int i = 0; i < n; i++) {
        while (!st.empty() && heights[st.top()] >= heights[i]) {
            right[st.top()] = i;
            st.pop();
        }
        left[i] = st.empty() ? -1 : st.top();
        st.push(i);
    }

    int maxArea = 0;
    for (int i = 0; i < n; i++) {
        maxArea = max(maxArea, heights[i] * (right[i] - left[i] - 1));
    }

    return maxArea;
}

/**
 * 字符串解码 - 栈解法
 * LeetCode 394: https://leetcode.com/problems/decode-string/
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
string decodeString(string s) {
    stack<pair<int, string>> st;
    int currentNum = 0;
```

```
    string currentStr = "";

    for (char c : s) {
        if (isdigit(c)) {
            currentNum = currentNum * 10 + (c - '0');
        } else if (c == '[') {
            st.push({currentNum, currentStr});
            currentNum = 0;
            currentStr = "";
        } else if (c == ']') {
            auto topPair = st.top();
            int num = topPair.first;
            string prevStr = topPair.second;
            st.pop();
            string temp = "";
            for (int i = 0; i < num; i++) {
                temp += currentStr;
            }
            currentStr = prevStr + temp;
        } else {
            currentStr += c;
        }
    }

    return currentStr;
}

/**
 * 基本计算器 II - 栈解法
 * LeetCode 227: https://leetcode.com/problems/basic-calculator-ii/
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
int calculate(string s) {
    stack<int> st;
    int num = 0;
    char sign = '+';

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        if (isdigit(c)) {
            num = num * 10 + (c - '0');
```

```
        }

        if ((!isdigit(c) && c != ' ') || i == s.length() - 1) {
            if (sign == '+') {
                st.push(num);
            } else if (sign == '-') {
                st.push(-num);
            } else if (sign == '*') {
                int top = st.top();
                st.pop();
                st.push(top * num);
            } else if (sign == '/') {
                int top = st.top();
                st.pop();
                st.push(top / num);
            }

            sign = c;
            num = 0;
        }
    }

    int result = 0;
    while (!st.empty()) {
        result += st.top();
        st.pop();
    }

    return result;
}

/**
 * 最小栈类 - 支持常数时间获取最小元素
 * LeetCode 155: https://leetcode.com/problems/min-stack/
 */
class MinStack {
private:
    stack<int> dataStack;
    stack<int> minStack;

public:
    MinStack() {}
```

```
    void push(int val) {
        dataStack.push(val);
        if (minStack.empty() || val <= minStack.top()) {
            minStack.push(val);
        }
    }

    void pop() {
        if (dataStack.top() == minStack.top()) {
            minStack.pop();
        }
        dataStack.pop();
    }

    int top() {
        return dataStack.top();
    }

    int getMin() {
        return minStack.top();
    }
};

/**
 * 用栈实现队列类
 * LeetCode 232: https://leetcode.com/problems/implement-queue-using-stacks/
 */
class CQueue {
private:
    stack<int> inStack;
    stack<int> outStack;

    void transfer() {
        while (!inStack.empty()) {
            outStack.push(inStack.top());
            inStack.pop();
        }
    }

public:
    CQueue() {}

    void appendTail(int value) {
```

```
        inStack.push(value);
    }

    int deleteHead() {
        if (outStack.empty()) {
            if (inStack.empty()) {
                return -1;
            }
            transfer();
        }
        int value = outStack.top();
        outStack.pop();
        return value;
    }
};

/**
 * 滑动窗口中位数 - 双堆解法
 * LeetCode 480: https://leetcode.com/problems/sliding-window-median/
 * 时间复杂度：O(n log k)
 * 空间复杂度：O(k)
 */
vector<double> medianSlidingWindow(vector<int>& nums, int k) {
    multiset<int> window(nums.begin(), nums.begin() + k);
    auto mid = next(window.begin(), k / 2);
    vector<double> medians;

    for (int i = k; ; i++) {
        // 推送当前中位数
        medians.push_back((double(*mid) + *prev(mid, 1 - k % 2)) / 2.0);

        // 如果所有元素处理完毕
        if (i == nums.size()) break;

        // 插入新元素
        window.insert(nums[i]);
        if (nums[i] < *mid) {
            mid--;
        }

        // 移除旧元素
        if (nums[i - k] <= *mid) {
            mid++;
```

```
            }
            window.erase(window.lower_bound(nums[i - k]));
        }

        return medians;
    }
};

/**
 * 主函数用于测试所有算法实现
 */
int main() {
    ConvertQueueAndStack solution;

    cout << "=== 测试柱状图中最大的矩形 ===" << endl;
    vector<int> heights = {2, 1, 5, 6, 2, 3};
    cout << "输入: [2,1,5,6,2,3]" << endl;
    cout << "输出: " << solution.largestRectangleAreaOptimized(heights) << endl;
    cout << "预期输出: 10" << endl;

    cout << "\n=== 测试字符串解码 ===" << endl;
    string encoded = "3[a]2[bc]";
    cout << "输入: \"3[a]2[bc]\"" << endl;
    cout << "输出: \"" << solution.decodeString(encoded) << "\"" << endl;
    cout << "预期输出: \"aaabcbc\"" << endl;

    cout << "\n=== 测试基本计算器 II ===" << endl;
    string expression = "3+2*2";
    cout << "输入: \"3+2*2\"" << endl;
    cout << "输出: " << solution.calculate(expression) << endl;
    cout << "预期输出: 7" << endl;

    cout << "\n=== 测试最小栈 ===" << endl;
    ConvertQueueAndStack::MinStack minStack;
    minStack.push(-2);
    minStack.push(0);
    minStack.push(-3);
    cout << "输入: push(-2), push(0), push(-3)" << endl;
    cout << "getMin(): " << minStack.getMin() << endl;
    cout << "预期: -3" << endl;
    minStack.pop();
    cout << "pop()" << endl;
    cout << "top(): " << minStack.top() << endl;
```

```
    cout << "预期: 0" << endl;
    cout << "getMin(): " << minStack.getMin() << endl;
    cout << "预期: -2" << endl;

    cout << "\n=== 测试用栈实现队列(CQueue) ===" << endl;
    ConvertQueueAndStack::CQueue queue;
    queue.appendTail(3);
    cout << "appendTail(3)" << endl;
    cout << "deleteHead(): " << queue.deleteHead() << endl;
    cout << "预期: 3" << endl;
    cout << "deleteHead(): " << queue.deleteHead() << endl;
    cout << "预期: -1" << endl;

    cout << "\n=== 测试滑动窗口中位数 ===" << endl;
    vector<int> nums = {1, 3, -1, -3, 5, 3, 6, 7};
    int k = 3;
    cout << "输入: [1,3,-1,-3,5,3,6,7], k=3" << endl;
    cout << "输出: [";
    vector<double> medians = solution.medianSlidingWindow(nums, k);
    for (int i = 0; i < medians.size(); i++) {
        cout << medians[i];
        if (i < medians.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
    cout << "预期输出: [1.0, -1.0, -1.0, 3.0, 5.0, 6.0]" << endl;

    return 0;
}

==========================================

文件: ConvertQueueAndStack.java
==========================================
import java.util.*;

/**
 * 队列和栈转换算法实现类
 * 包含 LeetCode 等各大算法平台中队列和栈相关的经典题目
 * 时间复杂度：O(n) - O(n^2) 根据具体算法
 * 空间复杂度：O(n) - O(n) 根据具体算法
 */
public class ConvertQueueAndStack {
```

```
/**
 * 柱状图中最大的矩形 - 单调栈解法
 * LeetCode 84: https://leetcode.com/problems/largest-rectangle-in-histogram/
 * 时间复杂度：O(n)，每个元素入栈出栈一次
 * 空间复杂度：O(n)，栈空间
 */
public int largestRectangleArea(int[] heights) {
    int n = heights.length;
    if (n == 0) return 0;

    int[] left = new int[n];
    int[] right = new int[n];
    Stack<Integer> st = new Stack<>();

    // 计算每个柱子左边第一个小于它的位置
    for (int i = 0; i < n; i++) {
        while (!st.isEmpty() && heights[st.peek()] >= heights[i]) {
            st.pop();
        }
        left[i] = st.isEmpty() ? -1 : st.peek();
        st.push(i);
    }

    // 清空栈
    st.clear();

    // 计算每个柱子右边第一个小于它的位置
    for (int i = n - 1; i >= 0; i--) {
        while (!st.isEmpty() && heights[st.peek()] >= heights[i]) {
            st.pop();
        }
        right[i] = st.isEmpty() ? n : st.peek();
        st.push(i);
    }

    // 计算最大面积
    int maxArea = 0;
    for (int i = 0; i < n; i++) {
        int width = right[i] - left[i] - 1;
        maxArea = Math.max(maxArea, heights[i] * width);
    }

    return maxArea;
```

```
}

/**
 * 柱状图中最大的矩形 - 优化版单调栈
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public int largestRectangleAreaOptimized(int[] heights) {
    int n = heights.length;
    if (n == 0) return 0;

    int[] left = new int[n];
    int[] right = new int[n];
    Arrays.fill(right, n);
    Stack<Integer> st = new Stack<>();

    for (int i = 0; i < n; i++) {
        while (!st.isEmpty() && heights[st.peek()] >= heights[i]) {
            right[st.peek()] = i;
            st.pop();
        }
        left[i] = st.isEmpty() ? -1 : st.peek();
        st.push(i);
    }

    int maxArea = 0;
    for (int i = 0; i < n; i++) {
        maxArea = Math.max(maxArea, heights[i] * (right[i] - left[i] - 1));
    }

    return maxArea;
}

/**
 * 字符串解码 - 栈解法
 * LeetCode 394: https://leetcode.com/problems/decode-string/
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public String decodeString(String s) {
    Stack<Object[]> st = new Stack<>();
    int currentNum = 0;
    StringBuilder currentStr = new StringBuilder();
```

```
    for (char c : s.toCharArray()) {
        if (Character.isDigit(c)) {
            currentNum = currentNum * 10 + (c - '0');
        } else if (c == '[') {
            st.push(new Object[]{currentNum, currentStr.toString()});
            currentNum = 0;
            currentStr = new StringBuilder();
        } else if (c == ']') {
            Object[] top = st.pop();
            int num = (int) top[0];
            String prevStr = (String) top[1];
            StringBuilder temp = new StringBuilder();
            for (int i = 0; i < num; i++) {
                temp.append(currentStr);
            }
            currentStr = new StringBuilder(prevStr + temp.toString());
        } else {
            currentStr.append(c);
        }
    }

    return currentStr.toString();
}

/**
 * 基本计算器 II - 栈解法
 * LeetCode 227: https://leetcode.com/problems/basic-calculator-ii/
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public int calculate(String s) {
    Stack<Integer> st = new Stack<>();
    int num = 0;
    char sign = '+';

    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);

        if (Character.isDigit(c)) {
            num = num * 10 + (c - '0');
        }
```

```
        if ((!Character.isDigit(c) && c != ' ') || i == s.length() - 1) {
            if (sign == '+') {
                st.push(num);
            } else if (sign == '-') {
                st.push(-num);
            } else if (sign == '*') {
                int top = st.pop();
                st.push(top * num);
            } else if (sign == '/') {
                int top = st.pop();
                st.push(top / num);
            }

            sign = c;
            num = 0;
        }
    }

    int result = 0;
    while (!st.isEmpty()) {
        result += st.pop();
    }

    return result;
}

/**
 * 最小栈类 - 支持常数时间获取最小元素
 * LeetCode 155: https://leetcode.com/problems/min-stack/
 */
class MinStack {
    private Stack<Integer> dataStack;
    private Stack<Integer> minStack;

    public MinStack() {
        dataStack = new Stack<>();
        minStack = new Stack<>();
    }

    public void push(int val) {
        dataStack.push(val);
        if (minStack.isEmpty() || val <= minStack.peek()) {
            minStack.push(val);
```

```
        }
    }

    public void pop() {
        if (dataStack.peek().equals(minStack.peek())) {
            minStack.pop();
        }
        dataStack.pop();
    }

    public int top() {
        return dataStack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}

/**
 * 用栈实现队列类
 * LeetCode 232: https://leetcode.com/problems/implement-queue-using-stacks/
 */
class CQueue {
    private Stack<Integer> inStack;
    private Stack<Integer> outStack;

    public CQueue() {
        inStack = new Stack<>();
        outStack = new Stack<>();
    }

    private void transfer() {
        while (!inStack.isEmpty()) {
            outStack.push(inStack.pop());
        }
    }

    public void appendTail(int value) {
        inStack.push(value);
    }

    public int deleteHead() {
```

```
        if (outStack.isEmpty()) {
            if (inStack.isEmpty()) {
                return -1;
            }
            transfer();
        }
        return outStack.pop();
    }
}

/**
 * 滑动窗口中位数 - 双堆解法
 * LeetCode 480: https://leetcode.com/problems/sliding-window-median/
 * 时间复杂度：O(n log k)
 * 空间复杂度：O(k)
 */
public double[] medianSlidingWindow(int[] nums, int k) {
    if (nums == null || nums.length == 0) return new double[0];

    // 使用两个 PriorityQueue 模拟平衡二叉搜索树
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();

    double[] medians = new double[nums.length - k + 1];

    for (int i = 0; i < nums.length; i++) {
        // 添加新元素
        if (maxHeap.isEmpty() || nums[i] <= maxHeap.peek()) {
            maxHeap.offer(nums[i]);
        } else {
            minHeap.offer(nums[i]);
        }

        // 平衡两个堆
        balanceHeaps(maxHeap, minHeap);

        // 如果窗口已满，计算中位数并移除旧元素
        if (i >= k - 1) {
            medians[i - k + 1] = getMedian(maxHeap, minHeap);

            // 移除窗口最左边的元素
            int toRemove = nums[i - k + 1];
            if (toRemove <= maxHeap.peek()) {
```

```
                maxHeap.remove(toRemove);
            } else {
                minHeap.remove(toRemove);
            }

            // 重新平衡
            balanceHeaps(maxHeap, minHeap);
        }
    }

    return medians;
}

// 辅助方法：平衡两个堆，确保 maxHeap 的大小等于或比 minHeap 大 1
private void balanceHeaps(PriorityQueue<Integer> maxHeap, PriorityQueue<Integer> minHeap) {
    // 如果 maxHeap 的大小比 minHeap 大 2，将 maxHeap 的堆顶元素移到 minHeap
    while (maxHeap.size() > minHeap.size() + 1) {
        minHeap.offer(maxHeap.poll());
    }

    // 如果 minHeap 的大小比 maxHeap 大，将 minHeap 的堆顶元素移到 maxHeap
    while (minHeap.size() > maxHeap.size()) {
        maxHeap.offer(minHeap.poll());
    }
}

// 辅助方法：计算中位数
private double getMedian(PriorityQueue<Integer> maxHeap, PriorityQueue<Integer> minHeap) {
    if (maxHeap.size() > minHeap.size()) {
        // 窗口大小为奇数，中位数是 maxHeap 的堆顶元素
        return maxHeap.peek();
    } else {
        // 窗口大小为偶数，中位数是两个堆顶元素的平均值
        return (maxHeap.peek() + minHeap.peek()) / 2.0;
    }
}

// 主函数用于测试所有算法实现
public static void main(String[] args) {
    ConvertQueueAndStack solution = new ConvertQueueAndStack();

    System.out.println("=== 测试柱状图中最大的矩形 ===");
    int[] heights = {2, 1, 5, 6, 2, 3};
```

```
System.out.println("输入: [2,1,5,6,2,3]");
System.out.println("输出: " + solution.largestRectangleAreaOptimized(heights));
System.out.println("预期输出: 10");

System.out.println("\n=== 测试字符串解码 ===");
String encoded = "3[a]2[bc]";
System.out.println("输入: \"3[a]2[bc]\"");
System.out.println("输出: \"" + solution.decodeString(encoded) + "\"");
System.out.println("预期输出: \"aaabcbc\"");

System.out.println("\n=== 测试基本计算器 II ===");
String expression = "3+2*2";
System.out.println("输入: \"3+2*2\"");
System.out.println("输出: " + solution.calculate(expression));
System.out.println("预期输出: 7");

System.out.println("\n=== 测试最小栈 ===");
ConvertQueueAndStack.MinStack minStack = solution.new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
System.out.println("输入: push(-2), push(0), push(-3)");
System.out.println("getMin(): " + minStack.getMin());
System.out.println("预期: -3");
minStack.pop();
System.out.println("pop()");
System.out.println("top(): " + minStack.top());
System.out.println("预期: 0");
System.out.println("getMin(): " + minStack.getMin());
System.out.println("预期: -2");

System.out.println("\n=== 测试用栈实现队列(CQueue) ===");
ConvertQueueAndStack.CQueue queue = solution.new CQueue();
queue.appendTail(3);
System.out.println("appendTail(3)");
System.out.println("deleteHead(): " + queue.deleteHead());
System.out.println("预期: 3");
System.out.println("deleteHead(): " + queue.deleteHead());
System.out.println("预期: -1");

System.out.println("\n=== 测试滑动窗口中位数 ===");
int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
int k = 3;
```

```
        System.out.print("输入: [1,3,-1,-3,5,3,6,7], k=3\n 输出: [");
        double[] medians = solution.medianSlidingWindow(nums, k);
        for (int i = 0; i < medians.length; i++) {
            System.out.print(medians[i]);
            if (i < medians.length - 1) System.out.print(", ");
        }
        System.out.println("]");
        System.out.println("预期输出: [1.0, -1.0, -1.0, 3.0, 5.0, 6.0]");
    }
}
```

==========

文件: ConvertQueueAndStack.py

==========

```
# 用栈实现队列
# 用队列实现栈

class MyQueue:
    """
    用栈实现队列
    测试链接: https://leetcode.cn/problems/implement-queue-using-stacks/

    题目描述:
    请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、
empty）:

    解题思路:
    使用两个栈，一个输入栈和一个输出栈。push 操作时将元素压入输入栈，pop 操作时如果输出栈为空，
    就将输入栈的所有元素依次弹出并压入输出栈，然后再从输出栈弹出元素。
    这样可以保证元素的顺序符合队列的 FIFO 特性。

    时间复杂度分析:
    - push 操作: O(1) - 直接压入输入栈
    - pop 操作: 均摊 O(1) - 虽然有时需要将输入栈的所有元素移到输出栈，但每个元素最多只会被移动一次
    - peek 操作: 均摊 O(1) - 同 pop 操作
    - empty 操作: O(1) - 检查两个栈是否都为空

    空间复杂度分析:
    O(n) - 需要两个栈来存储元素
    """

    def __init__(self):
```

```
        self.in_stack = []   # 输入栈
        self.out_stack = []  # 输出栈

    def push(self, x: int) -> None:
        """将元素 x 推到队列的末尾"""
        self.in_stack.append(x)

    def pop(self) -> int:
        """从队列的开头移除并返回元素"""
        self._check_out_stack()
        return self.out_stack.pop()

    def peek(self) -> int:
        """返回队列开头的元素"""
        self._check_out_stack()
        return self.out_stack[-1]

    def empty(self) -> bool:
        """如果队列为空，返回 True；否则，返回 False"""
        return len(self.in_stack) == 0 and len(self.out_stack) == 0

    def _check_out_stack(self) -> None:
        """检查输出栈是否为空，如果为空则将输入栈的所有元素移到输出栈"""
        if not self.out_stack:
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())


class MyStack:
    """
    用队列实现栈
    测试链接: https://leetcode.cn/problems/implement-stack-using-queues/

    题目描述：
    请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（push、top、pop 和 empty）。

    解题思路：
    使用两个队列，一个主队列和一个辅助队列。每次 push 操作时，将新元素加入辅助队列，
    然后将主队列的所有元素依次移到辅助队列，最后交换两个队列的角色。
    这样可以保证新元素总是在队列的前端，实现栈的 LIFO 特性。

    时间复杂度分析：
```

- push 操作：O(n) - 需要将主队列的所有元素移到辅助队列
- pop 操作：O(1) - 直接从主队列前端移除元素
- top 操作：O(1) - 直接返回主队列前端元素
- empty 操作：O(1) - 检查主队列是否为空

空间复杂度分析：
O(n) - 需要两个队列来存储元素
"""

```
    def __init__(self):
        self.queue1 = []  # 主队列
        self.queue2 = []  # 辅助队列

    def push(self, x: int) -> None:
        """将元素 x 压入栈顶"""
        # 将新元素加入辅助队列
        self.queue2.append(x)
        # 将主队列的所有元素移到辅助队列
        while self.queue1:
            self.queue2.append(self.queue1.pop(0))
        # 交换两个队列的角色
        self.queue1, self.queue2 = self.queue2, self.queue1

    def pop(self) -> int:
        """移除并返回栈顶元素"""
        return self.queue1.pop(0)

    def top(self) -> int:
        """返回栈顶元素"""
        return self.queue1[0]

    def empty(self) -> bool:
        """如果栈是空的，返回 True；否则，返回 False"""
        return len(self.queue1) == 0


class MyStackOneQueue:
    """
    用一个队列实现栈
    题目来源：LeetCode 225. 用队列实现栈（进阶解法）
    链接：https://leetcode.cn/problems/implement-stack-using-queues/

    题目描述：
```

请你仅使用一个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（push、top、pop 和 empty）。

解题思路：

使用一个队列实现栈。在 push 操作时，先将新元素加入队列尾部，然后将队列中已有的 n-1 个元素依次从头部取出并重新加入队列尾部，

这样新元素就位于队列头部，实现了栈的 LIFO 特性。

时间复杂度分析：

- push 操作：O(n) - 需要将队列中已有的元素重新排列
- pop 操作：O(1) - 直接从队列头部移除元素
- top 操作：O(1) - 直接返回队列头部元素
- empty 操作：O(1) - 检查队列是否为空

空间复杂度分析：

O(n) - 需要一个队列来存储元素

"""

```
    def __init__(self):
        self.queue = []

    def push(self, x: int) -> None:
        """将元素 x 压入栈顶"""
        n = len(self.queue)
        self.queue.append(x)
        # 将前面的 n 个元素依次从头部取出并重新加入队列尾部
        for _ in range(n):
            self.queue.append(self.queue.pop(0))

    def pop(self) -> int:
        """移除并返回栈顶元素"""
        return self.queue.pop(0)

    def top(self) -> int:
        """返回栈顶元素"""
        return self.queue[0]

    def empty(self) -> bool:
        """如果栈是空的，返回 True；否则，返回 False"""
        return len(self.queue) == 0


class MyCircularDeque:
```

```
"""
循环双端队列
题目来源：LeetCode 641. 设计循环双端队列
链接：https://leetcode.cn/problems/design-circular-deque/

题目描述：
设计实现双端队列。实现 MyCircularDeque 类:
MyCircularDeque(int k)：构造函数,双端队列最大为 k 。
boolean insertFront(int value)：将一个元素添加到双端队列头部。 如果操作成功返回 true ，否则返回 false 。
boolean insertLast(int value)：将一个元素添加到双端队列尾部。如果操作成功返回 true ，否则返回 false 。
boolean deleteFront()：从双端队列头部删除一个元素。 如果操作成功返回 true ，否则返回 false 。
boolean deleteLast()：从双端队列尾部删除一个元素。如果操作成功返回 true ，否则返回 false 。
int getFront()：从双端队列头部获得一个元素。如果双端队列为空，返回 -1 。
int getRear()：获得双端队列的最后一个元素。 如果双端队列为空，返回 -1 。
boolean isEmpty()：若双端队列为空，则返回 true ，否则返回 false 。
boolean isFull()：若双端队列满了，则返回 true ，否则返回 false 。

解题思路：
使用数组实现循环双端队列。维护头指针 front、尾指针 rear 和元素个数 size，通过取模运算实现循环特性。
头部插入时 front 指针向前移动，尾部插入时 rear 指针向后移动，注意处理边界情况和循环特性。

时间复杂度分析：
所有操作都是O(1)时间复杂度

空间复杂度分析：
O(k) - k 是双端队列的容量
"""

def __init__(self, k: int):
    """构造函数,双端队列最大为 k"""
    self.elements = [0] * (k + 1)  # 多申请一个空间用于区分队列满和空的情况
    self.capacity = k + 1
    self.front = 0
    self.rear = 0
    self.size = 0

def insertFront(self, value: int) -> bool:
    """将一个元素添加到双端队列头部"""
    if self.isFull():
        return False
```

```
        # front 指针向前移动一位（考虑循环特性）
        self.front = (self.front - 1 + self.capacity) % self.capacity
        self.elements[self.front] = value
        self.size += 1
        return True

    def insertLast(self, value: int) -> bool:
        """将一个元素添加到双端队列尾部"""
        if self.isFull():
            return False
        self.elements[self.rear] = value
        # rear 指针向后移动一位（考虑循环特性）
        self.rear = (self.rear + 1) % self.capacity
        self.size += 1
        return True

    def deleteFront(self) -> bool:
        """从双端队列头部删除一个元素"""
        if self.isEmpty():
            return False
        # front 指针向后移动一位（考虑循环特性）
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return True

    def deleteLast(self) -> bool:
        """从双端队列尾部删除一个元素"""
        if self.isEmpty():
            return False
        # rear 指针向前移动一位（考虑循环特性）
        self.rear = (self.rear - 1 + self.capacity) % self.capacity
        self.size -= 1
        return True

    def getFront(self) -> int:
        """从双端队列头部获得一个元素"""
        if self.isEmpty():
            return -1
        return self.elements[self.front]

    def getRear(self) -> int:
        """获得双端队列的最后一个元素"""
        if self.isEmpty():
```

```
            return -1
        # 注意：rear 指向的是下一个插入位置，最后一个元素在(rear-1+capacity)%capacity 位置
        return self.elements[(self.rear - 1 + self.capacity) % self.capacity]

    def isEmpty(self) -> bool:
        """若双端队列为空，则返回 true ，否则返回 false"""
        return self.size == 0

    def isFull(self) -> bool:
        """若双端队列满了，则返回 true ，否则返回 false"""
        return self.size == self.capacity - 1  # 留一个空位用于区分满和空



def maxSlidingWindow(nums, k):
    """
    滑动窗口最大值
    题目来源：LeetCode 239. 滑动窗口最大值
    链接：https://leetcode.cn/problems/sliding-window-maximum/

    题目描述：
    给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。
    你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
    返回 滑动窗口中的最大值 。

    解题思路：
    使用双端队列实现单调队列。队列中存储数组下标，队列头部始终是当前窗口的最大值下标，
    队列保持单调递减特性。遍历数组时，维护队列的单调性并及时移除窗口外的元素下标，
    当窗口形成后，队列头部元素就是当前窗口的最大值。

    时间复杂度分析：
    O(n) - 每个元素最多入队和出队一次

    空间复杂度分析：
    O(k) - 双端队列最多存储 k 个元素
    """
    if not nums or k <= 0:
        return []

    from collections import deque

    n = len(nums)
    # 结果数组，大小为 n-k+1
    result = []
```

```
    # 双端队列，存储数组下标，队列头部是当前窗口的最大值下标
    dq = deque()

    for i in range(n):
        # 移除队列中超出窗口范围的下标
        while dq and dq[0] < i - k + 1:
            dq.popleft()

        # 维护队列单调性，移除所有小于当前元素的下标
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()

        # 将当前元素下标加入队列尾部
        dq.append(i)

        # 当窗口形成后，记录当前窗口的最大值
        if i >= k - 1:
            result.append(nums[dq[0]])

    return result


def dailyTemperatures(temperatures):
    """
    每日温度
    题目来源：LeetCode 739. 每日温度
    链接：https://leetcode.cn/problems/daily-temperatures/

    题目描述：
    给定一个整数数组 temperatures ，表示每天的温度，返回一个数组 answer ，
    其中 answer[i] 是指对于第 i 天，下一个更高温度出现在几天后。
    如果气温在这之后都不会升高，请在该位置用 0 来代替。

    解题思路：
    使用单调栈解决。栈中存储数组下标，保持栈中下标对应的温度单调递减。
    遍历温度数组，当遇到比栈顶元素对应温度更高的温度时，不断弹出栈顶元素并计算天数差，
    直到栈为空或当前温度不大于栈顶元素对应温度，然后将当前下标入栈。

    时间复杂度分析：
    O(n) - 每个元素最多入栈和出栈一次

    空间复杂度分析：
    O(n) - 栈最多存储 n 个元素
```

```
    """
    if not temperatures:
        return []

    n = len(temperatures)
    result = [0] * n
    # 单调栈，存储数组下标，保持栈中下标对应的温度单调递减
    stack = []

    for i in range(n):
        # 当栈不为空且当前温度大于栈顶下标对应的温度时
        while stack and temperatures[i] > temperatures[stack[-1]]:
            # 弹出栈顶元素并计算天数差
            index = stack.pop()
            result[index] = i - index
        # 将当前下标入栈
        stack.append(i)

    # 栈中剩余元素对应的天数差为0（默认值）
    return result


def trap(height):
    """
    接雨水
    题目来源：LeetCode 42. 接雨水
    链接：https://leetcode.cn/problems/trapping-rain-water/

    题目描述：
    给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

    解题思路：
    使用单调栈解决。栈中存储数组下标，保持栈中下标对应的高度单调递减。
    当遇到比栈顶元素高的柱子时，说明形成了一个凹槽，可以接水。

    时间复杂度：O(n)
    空间复杂度：O(n)
    是否为最优解：是最优解之一
    """
    if not height:
        return 0
    result = 0
    stack = []
```

```
    for i in range(len(height)):
        while stack and height[i] > height[stack[-1]]:
            bottom = stack.pop()
            if not stack:
                break
            left = stack[-1]
            width = i - left - 1
            h = min(height[left], height[i]) - height[bottom]
            result += width * h
        stack.append(i)
    return result


def largestRectangleArea(heights):
    """
    柱状图中最大的矩形
    题目来源：LeetCode 84. 柱状图中最大的矩形
    链接：https://leetcode.cn/problems/largest-rectangle-in-histogram/

    时间复杂度：O(n)
    空间复杂度：O(n)
    是否为最优解：是
    """
    if not heights:
        return 0
    max_area = 0
    stack = []
    for i in range(len(heights)):
        while stack and heights[i] < heights[stack[-1]]:
            h = heights[stack.pop()]
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, h * width)
        stack.append(i)
    while stack:
        h = heights[stack.pop()]
        width = len(heights) if not stack else len(heights) - stack[-1] - 1
        max_area = max(max_area, h * width)
    return max_area


def nextGreaterElement(nums1, nums2):
    """
    下一个更大元素 I
```

```
    题目来源：LeetCode 496
    链接：https://leetcode.cn/problems/next-greater-element-i/

    时间复杂度：O(m + n)
    空间复杂度：O(n)
    是否为最优解：是
    """
    if not nums1 or not nums2:
        return []
    map_dict = {}
    stack = []
    for num in nums2:
        while stack and num > stack[-1]:
            map_dict[stack.pop()] = num
        stack.append(num)
    while stack:
        map_dict[stack.pop()] = -1
    return [map_dict[num] for num in nums1]



def nextGreaterElements(nums):
    """
    下一个更大元素 II
    题目来源：LeetCode 503
    链接：https://leetcode.cn/problems/next-greater-element-ii/

    时间复杂度：O(n)
    空间复杂度：O(n)
    是否为最优解：是
    """
    if not nums:
        return []
    n = len(nums)
    result = [-1] * n
    stack = []
    for i in range(2 * n):
        num = nums[i % n]
        while stack and num > nums[stack[-1]]:
            result[stack.pop()] = num
        if i < n:
            stack.append(i)
    return result
```

```
def removeDuplicates(s):
    """
    删除字符串中的所有相邻重复项
    题目来源：LeetCode 1047
    链接：https://leetcode.cn/problems/remove-all-adjacent-duplicates-in-string/

    时间复杂度：O(n)
    空间复杂度：O(n)
    是否为最优解：是
    """
    if not s:
        return s
    stack = []
    for c in s:
        if stack and stack[-1] == c:
            stack.pop()
        else:
            stack.append(c)
    return ''.join(stack)


def evalRPN(tokens):
    """
    逆波兰表达式求值
    题目来源：LeetCode 150
    链接：https://leetcode.cn/problems/evaluate-reverse-polish-notation/

    时间复杂度：O(n)
    空间复杂度：O(n)
    是否为最优解：是
    """
    if not tokens:
        return 0
    stack = []
    for token in tokens:
        if token == '+':
            b, a = stack.pop(), stack.pop()
            stack.append(a + b)
        elif token == '-':
            b, a = stack.pop(), stack.pop()
            stack.append(a - b)
        elif token == '*':
```

```
            b, a = stack.pop(), stack.pop()
            stack.append(a * b)
        elif token == '/':
            b, a = stack.pop(), stack.pop()
            stack.append(int(a / b))
        else:
            stack.append(int(token))
    return stack[-1]


def isValid(s):
    """
    有效的括号
    题目来源：LeetCode 20
    链接：https://leetcode.cn/problems/valid-parentheses/

    时间复杂度：O(n)
    空间复杂度：O(n)
    是否为最优解：是
    """
    if not s or len(s) % 2 != 0:
        return False
    stack = []
    for c in s:
        if c in '([{':
            stack.append(c)
        else:
            if not stack:
                return False
            top = stack.pop()
            if (c == ')' and top != '(') or (c == ']' and top != '[') or (c == '}' and top !=
'{'):
                return False
    return not stack


def largestRectangleArea(heights):
    """
    柱状图中最大的矩形
    题目来源：LeetCode 84. 柱状图中最大的矩形
    链接：https://leetcode.cn/problems/largest-rectangle-in-histogram/

    题目描述：
```

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。
求在该柱状图中，能够勾勒出来的矩形的最大面积。

解题思路：
使用单调栈。对于每个柱子，我们需要找到其左侧第一个比它矮的柱子和右侧第一个比它矮或等于它的柱子，
这样就能确定以当前柱子为高度的最大矩形的宽度。使用单调栈可以在线性时间内找到所有柱子的左右边界。

时间复杂度分析：
O(n) - 每个柱子最多入栈和出栈一次

空间复杂度分析：
O(n) - 栈最多存储 n 个柱子的索引

是否为最优解：是

```
"""
if not heights:
    return 0

n = len(heights)
left_bound = [-1] * n  # 左边界数组
right_bound = [n] * n  # 右边界数组
stack = []

# 计算每个柱子的左边界（左侧第一个比当前柱子矮的索引）
for i in range(n):
    while stack and heights[stack[-1]] >= heights[i]:
        stack.pop()
    if stack:
        left_bound[i] = stack[-1]
    stack.append(i)

# 清空栈，准备计算右边界
stack = []

# 计算每个柱子的右边界（右侧第一个比当前柱子矮的索引）
for i in range(n-1, -1, -1):
    while stack and heights[stack[-1]] > heights[i]:
        stack.pop()
    if stack:
        right_bound[i] = stack[-1]
    stack.append(i)
```

```
    # 计算最大矩形面积
    max_area = 0
    for i in range(n):
        width = right_bound[i] - left_bound[i] - 1
        area = heights[i] * width
        if area > max_area:
            max_area = area

    return max_area


def decodeString(s):
    """
    字符串解码
    题目来源：LeetCode 394. 字符串解码
    链接：https://leetcode.cn/problems/decode-string/

    题目描述：
    给定一个经过编码的字符串，返回它解码后的字符串。
    编码规则为: k[encoded_string]，表示其中方括号内部的 encoded_string 正好重复 k 次。
    注意 k 保证为正整数。
    你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。
    此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

    解题思路：
    使用两个栈，一个存储数字（重复次数），一个存储字符串。遍历字符串，遇到数字就解析出完整的数字，
    遇到左括号就将当前数字和当前字符串入栈，并重置数字和当前字符串；遇到右括号就弹出栈顶的数字和字符串，
    将当前字符串重复对应次数后与弹出的字符串拼接；遇到普通字符就添加到当前字符串中。

    时间复杂度分析：
    O(n) - 虽然有嵌套结构，但每个字符只会被处理一次

    空间复杂度分析：
    O(n) - 栈的大小与字符串的嵌套深度有关

    是否为最优解：是
    """
    num_stack = []  # 存储重复次数
    str_stack = []  # 存储中间字符串
```

```
    current_str = ""  # 当前处理的字符串
    current_num = 0  # 当前处理的数字

    for c in s:
        if c.isdigit():
            # 解析数字
            current_num = current_num * 10 + int(c)
        elif c == '[':
            # 遇到左括号，将当前数字和当前字符串入栈，重置
            num_stack.append(current_num)
            str_stack.append(current_str)
            current_num = 0
            current_str = ""
        elif c == ']':
            # 遇到右括号，弹出栈顶元素并处理
            repeat = num_stack.pop()
            prev_str = str_stack.pop()

            # 将当前字符串重复 repeat 次后与 prev_str 拼接
            current_str = prev_str + current_str * repeat
        else:
            # 普通字符，添加到当前字符串
            current_str += c

    return current_str



def calculate(s):
    """
    基本计算器 II
    题目来源：LeetCode 227. 基本计算器 II
    链接：https://leetcode.cn/problems/basic-calculator-ii/

    题目描述：
    给你一个字符串表达式 s ，请你实现一个基本计算器来计算并返回它的值。
    整数除法仅保留整数部分。
    你可以假设给定的表达式总是有效的。所有中间结果将在 [-2^31, 2^31 - 1] 的范围内。
    注意：不允许使用任何将字符串作为数学表达式计算的内置函数，比如 eval() 。

    解题思路：
    使用栈来存储数字和运算符。遍历字符串，解析出数字，根据当前运算符与栈顶运算符的优先级关系，
    决定是否需要先进行计算。对于乘除法，我们可以立即计算，对于加减法，我们将操作数和运算符分别入
栈，
```

最后再对栈中的元素进行计算。

时间复杂度分析：
O(n) - 只需遍历一次字符串

空间复杂度分析：
O(n) - 栈的大小与表达式的复杂度有关

是否为最优解：是
"""

```
num_stack = []  # 存储数字
op = '+'  # 当前运算符，默认为加号
num = 0  # 当前解析的数字

# 为了处理最后一个数字，在字符串末尾添加一个非数字字符
s += '+'

for c in s:
    if c.isdigit():
        # 解析数字
        num = num * 10 + int(c)
    elif c != ' ':
        # 遇到运算符
        if op == '+':
            num_stack.append(num)
        elif op == '-':
            num_stack.append(-num)
        elif op == '*':
            # 乘法优先级高，立即计算
            temp = num_stack.pop()
            num_stack.append(temp * num)
        elif op == '/':
            # 除法优先级高，立即计算
            temp = num_stack.pop()
            # 处理 Python 中负数除法的问题
            if temp * num < 0:
                num_stack.append(-(abs(temp) // abs(num)))
            else:
                num_stack.append(temp // num)
        # 更新运算符和重置数字
        op = c
        num = 0
```

```
    # 计算栈中所有数字的和
    return sum(num_stack)


class MinStack:
    """
    最小栈
    题目来源：LeetCode 155. 最小栈
    链接：https://leetcode.cn/problems/min-stack/

    题目描述：
    设计一个支持 push ，pop ，top 操作，并能在常数时间内检索到最小元素的栈。
    实现 MinStack 类:
    MinStack() 初始化堆栈对象。
    void push(int val) 将元素 val 推入堆栈。
    void pop() 删除堆栈顶部的元素。
    int top() 获取堆栈顶部的元素。
    int getMin() 获取堆栈中的最小元素。

    解题思路：
    使用辅助栈来同步存储当前栈中的最小值。当我们向主栈中 push 一个元素时，也向辅助栈中 push 当前的最小值；
    当我们从主栈中 pop 一个元素时，也从辅助栈中 pop 一个元素。这样，辅助栈的栈顶元素始终是当前栈中的最小值。

    时间复杂度分析：
    所有操作都是 O(1)时间复杂度

    空间复杂度分析：
    O(n) - 需要一个辅助栈来存储最小值

    是否为最优解：是
    """
    def __init__(self):
        """初始化堆栈对象"""
        self.stack = []  # 主栈，存储所有元素
        self.min_stack = []  # 辅助栈，存储最小值

    def push(self, val):
        """将元素 val 推入堆栈"""
        self.stack.append(val)
        # 辅助栈为空或当前元素小于等于辅助栈栈顶元素时，将当前元素入辅助栈
        if not self.min_stack or val <= self.min_stack[-1]:
```

```
            self.min_stack.append(val)
        else:
            # 否则，重复压入当前最小值
            self.min_stack.append(self.min_stack[-1])

    def pop(self):
        """删除堆栈顶部的元素"""
        if self.stack:
            self.stack.pop()
            self.min_stack.pop()

    def top(self):
        """获取堆栈顶部的元素"""
        return self.stack[-1]

    def getMin(self):
        """获取堆栈中的最小元素"""
        return self.min_stack[-1]



class CQueue:
    """
    用栈实现队列（CQueue）
    题目来源：剑指 Offer 09. 用两个栈实现队列
    链接：https://leetcode.cn/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/

    题目描述：
    用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead ，
    分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，deleteHead 操作返回 -1 )

    解题思路：
    使用两个栈，一个输入栈和一个输出栈。appendTail 操作时将元素压入输入栈，deleteHead 操作时如果输出栈为空，
    就将输入栈的所有元素依次弹出并压入输出栈，然后再从输出栈弹出元素。
    这样可以保证元素的顺序符合队列的 FIFO 特性。

    时间复杂度分析：
    - appendTail 操作：O(1) - 直接压入输入栈
    - deleteHead 操作：均摊 O(1) - 虽然有时需要将输入栈的所有元素移到输出栈，但每个元素最多只会被移动一次

    空间复杂度分析：
```

O(n) - 需要两个栈来存储元素

是否为最优解：是
"""
```
    def __init__(self):
        """初始化队列"""
        self.in_stack = []  # 输入栈
        self.out_stack = []  # 输出栈

    def appendTail(self, value):
        """在队列尾部插入整数"""
        self.in_stack.append(value)

    def deleteHead(self):
        """在队列头部删除整数，如果队列为空则返回-1"""
        if not self.out_stack:
            # 如果输出栈为空，将输入栈的所有元素移到输出栈
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())

        if not self.out_stack:
            return -1  # 队列为空

        return self.out_stack.pop()


def medianSlidingWindow(nums, k):
    """
    滑动窗口中位数
    题目来源：LeetCode 480. 滑动窗口中位数
    链接：https://leetcode.cn/problems/sliding-window-median/

    题目描述：
    给你一个数组 nums，有一个大小为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数，每次窗口向右移动 1 位。
    你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。

    解题思路：
    使用两个堆，一个最大堆和一个最小堆来维护窗口中的元素。最大堆存储窗口中较小的一半元素，
    最小堆存储窗口中较大的一半元素。这样，当 k 为奇数时，中位数就是最大堆的堆顶元素；当 k 为偶数时，
    中位数是两个堆顶元素的平均值。当窗口滑动时，我们需要从堆中移除离开窗口的元素，并添加新进入窗口的元素。
```

时间复杂度分析：
O(n log k) - n 是数组长度，每个元素的插入和删除操作的时间复杂度为 O(log k)

空间复杂度分析：
O(k) - 需要两个堆来存储窗口中的元素

是否为最优解：是
"""

```
import heapq

if not nums or k <= 0:
    return []

# 由于 Python 的 heapq 只支持最小堆，我们可以通过存储负数来模拟最大堆
max_heap = []  # 最大堆，存储较小的一半元素
min_heap = []  # 最小堆，存储较大的一半元素

# 辅助函数：将元素添加到堆中，保持两个堆的平衡
def add_to_heaps(num):
    # 优先添加到最大堆
    heapq.heappush(max_heap, -num)
    # 确保最大堆的最大值不大于最小堆的最小值
    if min_heap and -max_heap[0] > min_heap[0]:
        # 交换两个堆的堆顶元素
        max_val = -heapq.heappop(max_heap)
        min_val = heapq.heappop(min_heap)
        heapq.heappush(max_heap, -min_val)
        heapq.heappush(min_heap, max_val)
    # 重新平衡两个堆的大小
    if len(max_heap) > len(min_heap) + 1:
        # 将最大堆的堆顶元素移到最小堆
        heapq.heappush(min_heap, -heapq.heappop(max_heap))
    if len(min_heap) > len(max_heap):
        # 将最小堆的堆顶元素移到最大堆
        heapq.heappush(max_heap, -heapq.heappop(min_heap))

# 辅助函数：从堆中移除元素（注意：这里使用了一种简化的方法，实际上需要更高效的实现）
def remove_from_heaps(num):
    # 由于堆不支持高效的随机访问删除，这里使用一个临时方法
    # 将堆转换为列表，找到并移除元素，然后重新建堆
    nonlocal max_heap, min_heap
```

```
    # 尝试从 max_heap 中移除
    found = False
    temp = []
    while max_heap:
        val = -heapq.heappop(max_heap)
        if not found and val == num:
            found = True
        else:
            temp.append(-val)
    # 重建 max_heap
    max_heap = temp
    heapq.heapify(max_heap)

    # 如果在 max_heap 中没有找到，则从 min_heap 中移除
    if not found:
        temp = []
        while min_heap:
            val = heapq.heappop(min_heap)
            if val == num:
                found = True
                break
            else:
                temp.append(val)
        # 重建 min_heap
        for v in temp:
            heapq.heappush(min_heap, v)

    # 重新平衡两个堆
    if len(max_heap) > len(min_heap) + 1:
        heapq.heappush(min_heap, -heapq.heappop(max_heap))
    if len(min_heap) > len(max_heap):
        heapq.heappush(max_heap, -heapq.heappop(min_heap))

# 辅助函数：计算中位数
def get_median():
    if len(max_heap) > len(min_heap):
        # 窗口大小为奇数，中位数是 max_heap 的堆顶元素
        return -max_heap[0]
    else:
        # 窗口大小为偶数，中位数是两个堆顶元素的平均值
        return (-max_heap[0] + min_heap[0]) / 2

# 首先将前 k 个元素加入堆中
```

```
    for i in range(k):
        add_to_heaps(nums[i])

    # 计算初始窗口的中位数
    result = [get_median()]

    # 滑动窗口
    for i in range(k, len(nums)):
        # 移除离开窗口的元素
        remove_from_heaps(nums[i - k])
        # 添加新进入窗口的元素
        add_to_heaps(nums[i])
        # 计算当前窗口的中位数
        result.append(get_median())

    return result

# 测试函数
def test_all_algorithms():
    print("=== 测试柱状图中最大的矩形 ===")
    heights = [2, 1, 5, 6, 2, 3]
    print(f"输入: {heights}")
    result = largestRectangleArea(heights)
    print(f"输出: {result}")
    print("预期输出: 10")

    print("\n=== 测试字符串解码 ===")
    encoded = "3[a]2[bc]"
    print(f"输入: \"{encoded}\")")
    result = decodeString(encoded)
    print(f"输出: \"{result}\")")
    print("预期输出: \"aaabcbc\"")

    print("\n=== 测试基本计算器 II ===")
    expression = "3+2*2"
    print(f"输入: \"{expression}\")")
    result = calculate(expression)
    print(f"输出: {result}")
    print("预期输出: 7")

    print("\n=== 测试最小栈 ===")
    minStack = MinStack()
    minStack.push(-2)
```

```
    minStack.push(0)
    minStack.push(-3)
    print("输入: push(-2), push(0), push(-3)")
    print(f"getMin(): {minStack.getMin()}")
    print("预期: -3")
    minStack.pop()
    print("pop()")
    print(f"top(): {minStack.top()}")
    print("预期: 0")
    print(f"getMin(): {minStack.getMin()}")
    print("预期: -2")

    print("\n=== 测试用栈实现队列(CQueue) ===")
    queue = CQueue()
    queue.appendTail(3)
    print("appendTail(3)")
    print(f"deleteHead(): {queue.deleteHead()}")
    print("预期: 3")
    print(f"deleteHead(): {queue.deleteHead()}")
    print("预期: -1")

    print("\n=== 测试滑动窗口中位数 ===")
    nums = [1, 3, -1, -3, 5, 3, 6, 7]
    k = 3
    print(f"输入: {nums}, k={k}")
    result = medianSlidingWindow(nums, k)
    print(f"输出: {result}")
    print("预期输出: [1.0, -1.0, -1.0, 3.0, 5.0, 6.0]")

# 如果作为主程序运行，则执行测试
if __name__ == "__main__":
    test_all_algorithms()

==============================================
```