225 用队列实现栈

Label：栈

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通队列的全部四种操作（push、top、pop 和 empty）。
实现 MyStack 类：
    void push(int x) 将元素 x 压入栈顶。
    int pop() 移除并返回栈顶元素。
    int top() 返回栈顶元素。
    boolean empty() 如果栈是空的，返回 true ；否则，返回 false 。

- 两个队列 实现 栈

```
class MyStack {

    Queue<Integer> inQueue = null;
    Queue<Integer> outQueue = null;
    /** Initialize your data structure here. */
    public MyStack() {
        inQueue = new LinkedList<>();
        outQueue = new LinkedList<>();
    }

    /** Push element x onto stack. */
    public void push(int x) {
        inQueue.add(x);  // 每次 inQueue 中加入的都是队头元素
        while (!outQueue.isEmpty()) {
            inQueue.add(outQueue.poll());  // 将 outQueue 中元素倒给 inQueue ，相当于把 栈顶 与 原栈 接上
        }
        // 相互交换后，outQueue的队列信息就是栈，而 outQueue 是一个空队列，等待下一轮的 栈顶元素
        Queue temp = inQueue;
        inQueue = outQueue;
        outQueue = temp;
    }

    /** Removes the element on top of the stack and returns that element. */
    public int pop() {
        return outQueue.poll();
    }

    /** Get the top element. */
    public int top() {
        return outQueue.peek();
    }

    /** Returns whether the stack is empty. */
    public boolean empty() {
        return outQueue.isEmpty();
    }
}
```

- 一个队列实现栈

```
class MyStack {
    Queue<Integer> queue;

    /** Initialize your data structure here. */
    public MyStack() {
        queue = new LinkedList<Integer>();
    }

    /** Push element x onto stack. */
    public void push(int x) {
        int n = queue.size();
        queue.offer(x);
        for (int i = 0; i < n; i++) {
            queue.offer(queue.poll());
        }
    }

    /** Removes the element on top of the stack and returns that element. */
    public int pop() {
        return queue.poll();
    }

    /** Get the top element. */
    public int top() {
        return queue.peek();
    }

    /** Returns whether the stack is empty. */
    public boolean empty() {
        return queue.isEmpty();
    }
}
```

232 用栈实现队列

Label：栈、队列

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）：
实现 MyQueue 类：
    void push(int x) 将元素 x 推到队列的末尾
    int pop() 从队列的开头移除并返回元素
    int peek() 返回队列开头的元素
    boolean empty() 如果队列为空，返回 true ；否则，返回 false

- 两个栈 实现 队列

```
class MyQueue {

    Stack<Integer> stack = null;
    Stack<Integer> stackQ = null;

    /** Initialize your data structure here. */
    public MyQueue() {
        stack = new Stack<>();
        stackQ = new Stack<>();
    }

    /** Push element x to the back of queue. */
    public void push(int x) {

        while (!stackQ.isEmpty()) {
            stack.add(stackQ.pop());
        }
        stackQ.push(x);
        while (!stack.isEmpty()) {
            stackQ.add(stack.pop());
        }
    }

    /** Removes the element from in front of queue and returns that element. */
    public int pop() {
        return stackQ.pop();
    }

    /** Get the front element. */
    public int peek() {
        return stackQ.peek();
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return stackQ.isEmpty();
    }
}
```

- 优化时间复杂度

```
class MyQueue {

    private Stack<Integer> a = null;  // 用来存储 临时加入队列的元素，b为空了，再将累计的元素倒进B
    private Stack<Integer> b = null;  // 用来存储 队列元素

    public MyQueue() {
         a = new Stack<Integer>();
         b = new Stack<Integer>();
    }

    public void push(int x) {
        a.push(x);
    }

    public int pop() {   // 这样就可以累积一定量之后再倒腾，优化时间复杂度
        //如果栈b不为空，直接pop即可
        if(!b.isEmpty()) {
            return b.pop();
        }
        //如果栈b为空，需要先将栈a中的数据倒腾到栈b中，再pop
        while(!a.isEmpty()) {
            b.push(a.pop());
        }
        return b.pop();
    }

    public int peek() {
        //如果栈b不为空，直接peek即可
        if(!b.isEmpty()) {
            return b.peek();
        }
        //如果栈b为空，需要先将栈a中的数据倒腾到栈b中，再peek
        while(!a.isEmpty()) {
            b.push(a.pop());
        }
        return b.peek();
    }

    public boolean empty() {
        //栈a和栈b需要同时判空
        return a.isEmpty() && b.isEmpty();
    }
}
```

739 每日温度

Label：数组

请根据每日 气温 列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 0 来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

提示：气温 列表长度的范围是 [1, 30000]。每个气温的值的均为华氏度，都是在 [30, 100] 范围内的整数。

- 迭代

```
class Solution {
    public int[] dailyTemperatures(int[] T) {
        int[] r = new int[T.length];
        for (int i = 0; i < T.length; i++) {
            int day = 1;
            for (int j = i + 1; j < T.length; j++) {
                if (T[i] >= T[j]) {
                    day++;
                }else { // <
                    r[i] = day;
                    break;
                }
            }
        }
        return r;
    }
}
```

- 单调栈

```
class Solution {
    public int[] dailyTemperatures(int[] T) {
        int[] r = new int[T.length];
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < T.length; i++) {
            int currT = T[i];
            while (!stack.isEmpty() && T[stack.peek()] < currT) { // 出栈 即可计算出栈元素的结果
                int index = stack.pop();
                r[index] = i - index;
            }
            stack.push(i);  // 栈中只用存入 index 即可
        }
        return r;
    }
}
```

- 不用栈 直接实现动态倒序更新，有种动态规划的思想

```
class Solution {
    public int[] dailyTemperatures(int[] T) {
        if (null == T || 0 == T.length) return null;
        int[] result= new int[T.length];

        for (int i = T.length - 2; i >= 0; --i) {  // 倒序进行更新
            int j = i + 1;
            while (true) {
                if (T[i] < T[j]) {
                    result[i] = j - i;
                    break;
                } else if (result[j] == 0) {
                    result[i] = 0;
                    break;
                }
                j += result[j];
            }
        }
        return result;
    }
}
```

1047 删除字符串中的所有相邻重复项

Lable：栈、双指针

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。
在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

输入："abbaca"
输出："ca"

- 栈

```
class Solution {
    public String removeDuplicates(String S) {

        Stack<Character> stack = new Stack<>();
        for (int i = 0; i < S.length(); i++) {
            if (!stack.isEmpty() && stack.peek() == S.charAt(i)) {   // 不用考虑连续的情况，题目要求一对一对删除
                stack.pop();
            } else {
                stack.push(S.charAt(i));
            }
        }

        // create string
        StringBuilder stringBuilder = new StringBuilder();
        while (!stack.isEmpty()) {
            stringBuilder.append(stack.pop());
        }
        return stringBuilder.reverse().toString();
    }
}
```

- 修改原数组 （类似于双指针）

```
class Solution {
    public String removeDuplicates(String S) {
        char[] s = S.toCharArray();
        int top = -1;
        for (int i = 0; i < S.length(); i++) {
            if (top == -1 || s[top] != s[i]) {
                s[++top] = s[i];
            } else {
                top--;
            }
        }
        return String.valueOf(s, 0, top + 1);
    }
}
```

1190 反转没对括号间的子串

Label：栈

给出一个字符串 s（仅含有小写英文字母和括号）。
请你按照从括号内到外的顺序，逐层反转每对匹配括号中的字符串，并返回最终的结果。
注意，您的结果中 不应 包含任何括号。

输入：s = "(abcd)"
输出："dcba"
输入：s = "(u(love)i)"
输出："iloveu"
输入：s = "(ed(et(oc))el)"
输出："leetcode"
输入：s = "a(bcdefghijkl(mno)p)q"
输出："apmnolkjihgfedcbq"

- 栈 + 队列

```
class Solution {
    public String reverseParentheses(String s) {
        Stack<Character> stack = new Stack<>();
        Queue<Character> queue = new LinkedList<>();
        for (int i = 0; i< s.length(); i++) {
            if (s.charAt(i) == ')') {
                while (!stack.isEmpty() && stack.peek() != '(') {
                    queue.offer(stack.pop());
                }
                if (!stack.isEmpty()) stack.pop(); // 弹出 (
                // 反向入栈
                while (!queue.isEmpty()) {
                    stack.add(queue.poll());
                }
            }else { // 入栈
                stack.add(s.charAt(i));
            }
        }
        // 形成字符串
        StringBuffer sb = new StringBuffer();
        while (!stack.isEmpty()) {
            sb.append(stack.pop());
        }
        return sb.reverse().toString();
    }
}
```

- 双指针

```
class Solution {
    public String reverseParentheses(String s) {
        char[] arr = s.toCharArray();
        int right = 0, left, len = arr.length;
        while(right < len){
            if(arr[right] != ')'){
                right += 1;
            }
            // 找打右括号，从该点向前遍历
            else{
                left = right;
                while(arr[left] != '('){
                    left -= 1;
                }
                arr[left] = '0';   // 考虑用 0 来占位
                arr[right] = '0';
                reverse(arr, left, right);
            }
        }

        StringBuffer ans = new StringBuffer();
        for(int i = 0; i < len; i++){
            if(arr[i] != '0'){
                ans.append(arr[i]);
            }
        }
        return ans.toString();
    }

    public void reverse(char[] arr, int start, int end){
        while(start < end){
            char temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }
}
```