递归与栈：解决表达式求值

面试题 03.04. 化栈为队

思路：

1.首先举例，我们有上面 [1，2，3，4，5]
这样的一个队列，然后对它进行push/pop/peek/empty四个操作;首先创建两个栈：栈1是stack1 ，栈2是stack2;首先一开始，两个栈都是空的。栈1是stack1 ，栈2是stack2

2.接着，我们执行第一个操作： push to top；然后我们的push操作就是，往 stack1 里面push一个操作,接着，依次往 stac

stack1 里

3.此时，执行第二个操作: pop from top；首先 要判断： stack2 长度是否等于0；如果stack2的长度等于0，就在stack1的长度不为0 的时候，对stack1进行pop删除操作

4.因为栈的特性是，先进后出，后进先出；第一个删除的便是后进去的元素5；按着后进先出的依次删除：4，3，2，1

5.如果stack2的长度不等于0，我们就先将stack2里面的进行删除，一步一步删除后，直到stack2为空，然后再重复stack2

的长度等于0时的操作，就是再将stack1里面的元素删除，添加到是stack2里面，直到，stack1的长度为空，不断重复此操作，直到，stack1的长度为空，最后，stack1的长度为空

6.接着，执行第三个操作: peek from top；peek的操作是查看，查看栈顶的元素，因为它是从0开始数，所以到了这里，需要注意 peek 和 pop 的返回值不同

7.最后一个操作是： empty ,这个方法是最后进行判断队列为空； empty 只是最后返回， stack1和stack2的长度都为0时候

```
/**
 * Initialize your data structure here.
 */
var MyQueue = function () {
    this.stack = []
    this.tempStack = []
}

/**
 * Push element x to the back of queue.
 * @param {number} x
 * @return {void}
 */
MyQueue.prototype.push = function (x) {
    this.stack.push(x)
}

/**
 * Removes the element from in front of queue and returns that element.
```

```
 * @return {number}
 */
MyQueue.prototype.pop = function () {
    if (!this.tempStack.length) {
        while (this.stack.length) {
            this.tempStack.push(this.stack.pop());
        }
    }
    return this.tempStack.pop();
}

/**
 * Get the front element.
 * @return {number}
 */
MyQueue.prototype.peek = function () {
    if (!this.tempStack.length) {
        while (this.stack.length) {
            this.tempStack.push(this.stack.pop());
        }
    }
    let result = this.tempStack.pop();
    this.tempStack.push(result);
    return result;
}

/**
 * Returns whether the queue is empty.
 * @return {boolean}
 */
MyQueue.prototype.empty = function () {
    return !this.stack.length && !this.tempStack.length;
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * var obj = new MyQueue()
 * obj.push(x)
 * var param_2 = obj.pop()
 * var param_3 = obj.peek()
 * var param_4 = obj.empty()
 */
```

682.棒球比赛

思路：

1.首先创建一个栈，用于记录分数；

2.然后遍历整个数组，遇到数字就压入栈；

3.遇到数字就压入栈

4.遇到”C”就弹栈

5.遇到”D”就将栈顶元素乘以2，然后将乘积压入栈

6.遇到”+”就将栈顶元素出栈并记录

7.然后将栈顶元素和弹出的值相加，作为新元素

8.将刚刚弹出的元素重新压入栈

9.将新计算出的和压入栈

10.遍历完成后，最终计算栈中元素的和即可

```
var calPoints = function (ops) {
    let stack = [];
    let len = ops.length;
    for (let i = 0; i < len; i++) {
        if (ops[i] == 'C') {
            stack.pop()
        } else if (ops[i] == 'D') {
            stack.push(stack[stack.length - 1] * 2);
        } else if (ops[i] == '+') {
            stack.push(stack[stack.length - 1] + stack[stack.length - 2]);
        } else {
            stack.push(Number(ops[i]));
        }
    }
    return stack.reduce((a, b) => a + b);
};
```

844.比较含退格的字符串

思路：

1.我们可以利用栈，将两个字符串处理成没有退格的字符串，然后进行比较即可

2.我们创建一个栈，用来处理我们的两个字符串

3.首先处理我们的第一个字符串，我们将字符串的每个元素压入栈。如果遇到#我们就弹出一个元素；
当我们遍历完整个字符串的时候，我们就将栈中的字符输出组成字符串；

4.然后我们按相同的步骤处理第二个字符串

5.最后我们比较两个字符串是否相等

```
/**
 * @param {string} S
 * @param {string} T
 * @return {boolean}
```

```
 */
var backspaceCompare = function (S, T) {
    return processed(S) === processed(T);
};
const processed = (str) => {
    const stackStr = [];
    for (const c of str) {
        if (c === '#') {
            stackStr.pop();
        } else {
            stackStr.push(c);
        }
    }
    return stackStr.join('');
};
```

946.验证栈序列

思路：贪心算法

1.所有的元素一定是按顺序 push 进去的，重要的是思考怎么 pop 出来的

2.假设当前栈顶元素值为 2，同时对应的 popped 序列中下一个要 pop 的值也为 2，那就必须立刻把这个值 pop 出来。因为之后的 push 都会让栈顶元素变成不同于 2 的其他值，这样再 pop 出来的数 popped 序列就不对应了。

3.将 pushed 队列中的每个数都 push 到栈中，同时检查这个数是不是 popped 序列中下一个要 pop 的值，如果是就把它 pop 出来。

4.最后，检查是不是所有的该 pop 出来的值都 pop 出来了。

```
 var validateStackSequences = function (pushed, popped) {
    var stack = [];
    var j=0;//索引
    for (let cur of pushed) {
        stack.push(cur); //存
        while (stack[stack.length - 1] === popped[j] && stack.length > 0) { //匹配弹出
            stack.pop();
            j++;
        }
    }
    return !stack.length;
};
```

20.有效括号

思路：

1.给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。
有效字符串需满足：左括号必须用相同类型的右括号闭合。左括号必须以正确的顺序闭合。

2.我们遍历给字符串s。当我们遇到一个左括号时，我们会希望在后面的字符中，有一个相同类型的右括号将其闭合。

3.当我们遇到一个右括号时，我们就可以去出栈顶的左括号，判断两个括号是否能闭合。
如果不是相同类型的括号或者栈中没有括号，我们这个字符串就是无效的。

4.为了能快速的判断括号的类型，我们可以创建一个Map来帮助我们。 创建一个栈来存储我们的左括号。

5.最后开始遍历我们的字符串，如果是左括号，我们就将该字符加入到栈中。
如果是右括号，我们就将判断栈顶的元素和当前的元素是否能完全闭合。

```
var isValid = function (s) {
    let n = s.length;
    if (n % 2 === 1) return false;
    let map = new Map([
        [')', '('],
        [']', '['],
        ['}', '{']
    ]);
    let stack = [];
    for (ch of s) {
        if (map.has(ch)) {
            if (!stack.length || stack[stack.length - 1] !== map.get(ch)) {
                return false;
            }
            stack.pop()
        } else {
            stack.push(ch);
        }
    }
    return stack.length === 0;
};
```

```
var isValid = function (s) {
    let stack = [];
    for (ch of s) {
        switch (ch) {
            case '(':
            case '[':
            case '{':
                stack.push(ch);
                break;
            case ')':
                if(stack.pop()!== '(') return false;
                break;
```

```
            case '}':
                if(stack.pop()!== '{') return false;
                break;
            case ']':
                if(stack.pop()!== '[') return false;
                break;
        }
    }
    return !stack.length;
};
```

1021.删除最外层的括号

思路：

1.首先我们解决这道题的关键在于需要知道哪些是需要去除的外层括号；

2.为了找到这些需要去除的外层括号，我们需要用到计数器；

3.遇到左括号，我们的计数器 +1+1，遇到右括号，我们的计数器

-1−1，这样，一对有效保留的括号，总共让计数器归零，若不归零，则代表当前遍历到的括号中，一定存在多余括号；

4.最后的规律就是：遇到左括号，当前计数值大于 00，则属于有效的左括号；遇到右括号，当前计数值大于 11，则属于有效的右括号。

```
 var removeOuterParentheses = function (S) {
    let res = '';
    let opened = 0;
    for (ch of S) {
        // opend >0  我们已经有了一个左括号
        //我们新找到的这个左括号，就认为他不是最外层的括号，就给他拼接起来，然后这个数量进行加一；
        if (ch === '(' && opened++ > 0) res += ch;
        // opend >1  我们已经有了两个及以上左括号
        //我们新找到的这个右括号，就认为他不是最外层的括号，就给他拼接起来，然后这个左括号的数量进行减一；
        if (ch === ')' && opened-- > 1) res += ch;
    }
    return res;
};
```

1249.移除无效括号

思路：

1.首先分析 有效 字符串的含义；

2.当且仅当 满足以下条件时，字符串中的括号是平衡的：（1）字符串中有相同数量的 "(" 和 ")"；（2）从左至右遍历字符串，统计当前 "(" 和 ")" 的数量，永远不会出现 ")" 的数量大于 "(" 的数量，称 count("(") - count(")") 为字符串的余量。

3.从左向右扫描字符串，每次遇到 "(" 时，余量递增，遇到 ")" 时，余量递减；

4.在任何时候如果余量为负（")" 的数量多于 "("），则返回 false；

5.如果扫描到字符串末尾时余量为 0，说明 ")" 的数量等于 "(" 的数量，返回 true；

```
var minRemoveToMakeValid = function (s) {
    // 记录需要删除的多余括号的索引
    // leftDel, rightDel分别存放'(', ')'
    const n = s.length,
        leftDel = [],
        rightDel = [];
    for (let i = 0; i < n; i++) {
        const char = s[i];
        if (char === '(') {
            leftDel.push(i);
        } else if (char === ')') {
            // 如果有对应的'(', 从删除列表中移除
            // 否则')'是多余的，加入右括号的删除列表
            if (leftDel.length > 0) {
                leftDel.pop();
            } else {
                rightDel.push(i);
            }
        }
    }
    // 根据记录删除
    const res = [...s],
        filter = leftDel.concat(rightDel),
        l = filter.length;
    for (let i = 0; i < l; i++) {
        res[filter[i]] = '';
    }
    return res.join('');
};
```