栈和队列

```
/**
* 设计循环队列
* 1 牺牲一个位置来判断 空 和 满
* 2 设置 tag，每次删除成功令 tag==0，插入成功令 tag==1；队满条件：front==rear && tag==1 ；
*     队空条件：front==rear && tag==0
*/
class MyCircularQueue {
private:
    vector<int> arr;
    int front;
    int rear;
    int cap;
public:
    /** Initialize your data structure here. Set the size of the queue to be k. */
    MyCircularQueue(int k) {
        cap = k + 1;
        front = 0;
        rear = 0;
        arr.assign(cap,0);
    }

    /** Insert an element into the circular queue. Return true if the operation is successful. */
    bool enQueue(int value) {
        if(isFull()) return false;
        arr[rear] = value;
        rear = (rear+1) % cap;
        return true;
    }

    /** Delete an element from the circular queue. Return true if the operation is successful. */
    bool deQueue() {
        if(isEmpty()) return false;
        front = (front+1) % cap;
        return true;
        }

    /** Get the front item from the queue. */
    int Front() {
        if(isEmpty()) return -1;
        return arr[front];
    }


    /** Get the last item from the queue. */
```

```
    int Rear() {
        if(isEmpty()) return -1;
        // 考虑 cap=4, 入队 1, 2, 3 --> 出队 3 --> 入队 4 --> 此时rear指向0, 而队尾元素在size-1位置
        return arr[(rear - 1 + cap) % cap];
        // 等价于上面
        if(rear==0){
            return arr[szie -1];
        else
            return arr[rear - 1];
        }
    }

    /** Checks whether the circular queue is empty or not. */
    bool isEmpty() {
        return rear==front;
    }

    /** Checks whether the circular queue is full or not. */
    bool isFull() {
        return (rear+1) % cap == front;
    }
};
```

```
/** 给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，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。
 */
class Solution{
public:
    bool isValid(string s){
        int n = s.size();
        if(n % 2 == 1){return false;}
        unordered_map<char,char> pairs = {
            {')','('},
            {']','['},
            {'}','{'}
        };
        stack<char> stk;
        for(char ch : s){
            if(pairs.count(ch)){
                if(stk.empty() || stk.top() != pairs[ch]){
                    return false;
                }else{
                    stk.pop();
                }
            }else{
                stk.push(ch);
            }
        }
        return stk.empty();
    }
};
```

```
/*
    示例 1:
    输入: ["2", "1", "+", "3", "*"]
    输出: 9
    解释: 该算式转化为常见的中缀算术表达式为: ((2 + 1) * 3) = 9
*/

#define MAXSIZE 100
#define INCREASESIZE 100

/* 动态顺序栈结构 */
typedef struct stack {
    int* data;
    int top;
    int stacksize;
}Stack;

/* 入栈，栈满拓展栈空间 */
void Push(Stack* obj, int x) {
    if (obj->top == obj->stacksize - 1) {
        obj->data = (int*)realloc(obj->data, sizeof(int) * (obj->stacksize + INCREASESIZE));
        obj->stacksize += INCREASESIZE;
    }
    obj->data[++obj->top] = x;
}

/* 取栈顶的同时出栈 */
int TopAndPop(Stack* obj) {
    int x = obj->data[obj->top--];
    return x;
}

int evalRPN(char ** tokens, int tokensSize){
    Stack* obj = (Stack*)malloc(sizeof(Stack));
    obj->top = -1;
    obj->stacksize = MAXSIZE;
    obj->data = (int*)malloc(sizeof(int) * MAXSIZE);
    int x, y;
    for (int i = 0; i < tokensSize; i++) {
        //如果是运算符，取两次栈顶，计算，并将结果入栈
        if (!strcmp(tokens[i], "+")) {
            x = TopAndPop(obj); y = TopAndPop(obj);
            Push(obj, y + x);
        }
```

```
        else if(!strcmp(tokens[i], "-")) {
            x = TopAndPop(obj); y = TopAndPop(obj);
            Push(obj, y - x);
        }
        else if(!strcmp(tokens[i], "*")) {
            x = TopAndPop(obj); y = TopAndPop(obj);
            Push(obj, y * x);
        }
        else if(!strcmp(tokens[i], "/")) {
            x = TopAndPop(obj); y = TopAndPop(obj);
            Push(obj, y / x);
        }
        //子字符串为操作数，将其化为整型并入栈
        else {
            Push(obj, atoi(tokens[i]));
        }
    }
    return TopAndPop(obj);    //返回最后栈的唯一一个数
}
```