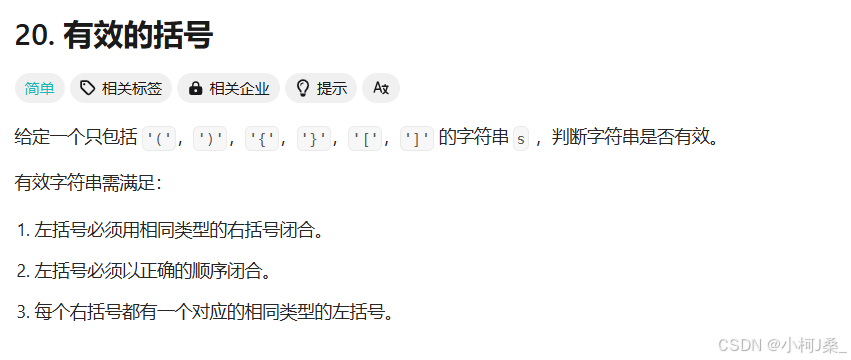
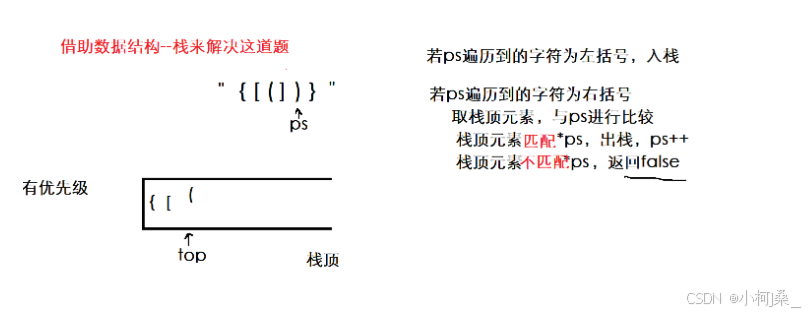
基于对栈和队列知识的了解，我们来一起做几道题目~  
一、有效的括号  
二、用队列实现栈  
三、用栈实现队列  
四、循环队列

一、有效的括号

<https://leetcode.cn/problems/valid-parentheses/description/>

typedef int STDataType;typedef struct Stack{

STDataType\* arr; //栈数组

int capacity; //栈的空间大小

int top; //栈顶位置}ST;

//栈的初始化void STInit(ST\* ps){

assert(ps);

ps->arr = NULL;

ps->capacity = 0;

ps->top = 0;}

//栈的销毁void STDestroy(ST\* ps){

assert(ps);

if(ps->arr)

free(ps->arr);

ps->arr = NULL;

ps->capacity = 0;

ps->top = 0;}

//数据入栈void StackPush(ST\* ps, STDataType x){

assert(ps);

if (ps->capacity == ps->top) //空间满了需要扩容

{

int newcapacity = ps->capacity == 0 ? 4 : 2 \* ps->capacity; //三目运算符如果原本栈为空，就赋初始为4个空间，若不为空，则双倍扩容

STDataType\* tem = (STDataType\*)realloc(ps->arr, newcapacity \* sizeof(ST));

//判断所开空间是否成功

if (tem == NULL)

{

perror("realloc fail!");

exit(1);

}

ps->arr = tem;

ps->capacity = newcapacity;

}

//入栈开始

ps->arr[ps->top++] = x;}

//栈判空bool StackEmpty(ST\* ps){

assert(ps);

return ps->top == 0;}

//数据出栈void StackPop(ST\* ps){

assert(ps);

assert(!StackEmpty(ps));

ps->top--;}

//取栈顶元素

STDataType StackTop(ST\* ps){

assert(ps);

assert(!StackEmpty(ps));

return ps->arr[ps->top - 1];}

//获取栈中有效元素个数int STSize(ST\* ps){

assert(ps);

return ps->top;}

bool isValid(char\* s) {

ST st;

STInit(&st);

char\* ps = s;

while(\*ps != '\0')

{

if(\*ps == '{' || \*ps == '[' || \*ps == '(')

{

StackPush(&st, \*ps);

}

else

{

//右括号，若为空直接返回flase

if(StackEmpty(&st))

{

return false;

}

//看是否匹配

char ch = StackTop(&st);

if((\*ps == '}' && ch == '{')

|| (\*ps == ']' && ch == '[')

|| (\*ps == ')' && ch == '('))

{

StackPop(&st);

}

else

{

//不匹配

STDestroy(&st);

return false;

}

}

ps++;

}

bool ret = StackEmpty(&st);

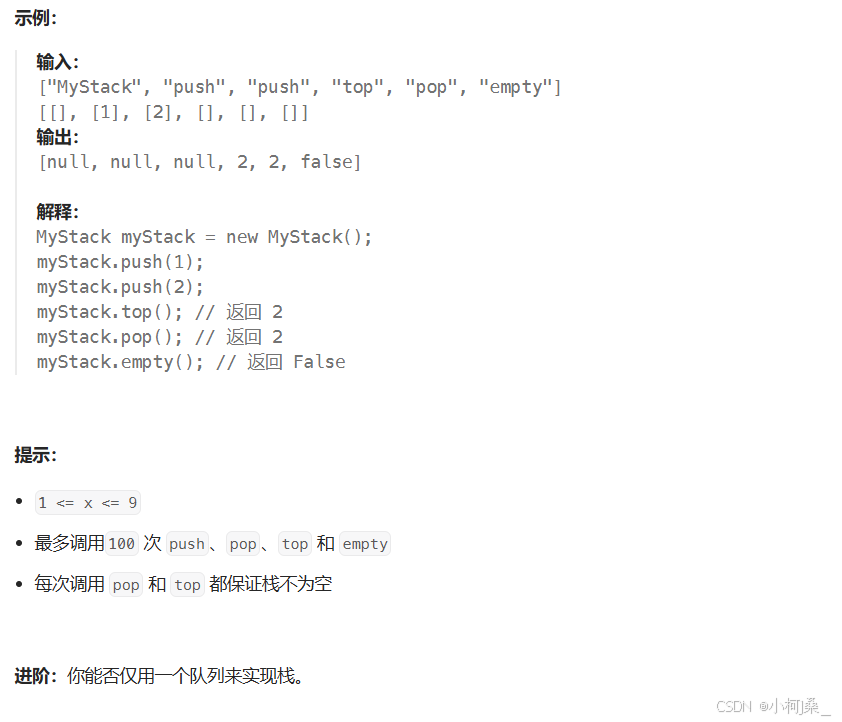
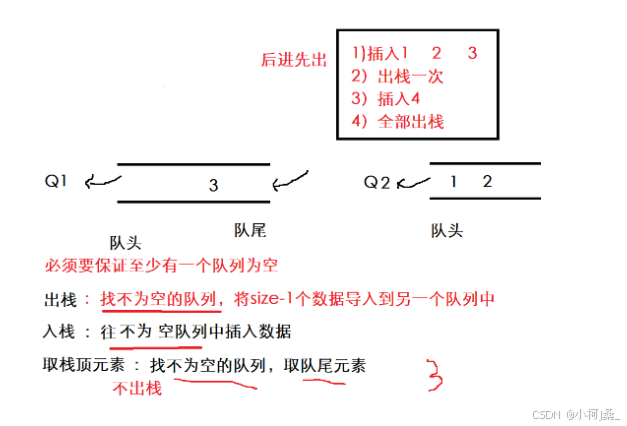
//销毁和返回

STDestroy(&st);

return ret;}

二、用队列实现栈

<https://leetcode.cn/problems/implement-stack-using-queues/description/>

此题两个队列，必须要保证至少一个队列为空。

出栈时，要将不为空的队列的size-1个元素放置在另一个空的队列中，这样剩下的元素就是出栈的，满足了后进先出。

入栈时，要向不为空的队列中插入元素，因为必须要保证有一个队列是空。

注意此题在销毁队列时被销毁的队列可以为NULL

此题在取出栈顶元素时和出栈不一样，如果只是单纯的不进行pop会导致最开始不为空的队列队尾的元素残留下来，因此我们不出栈，遍历取队尾元素。

typedef int QDataType;

typedef struct QueueNode{

QDataType data;

struct QueueNode\* next;}QueueNode;

typedef struct Queue{

QueueNode\* phead;

QueueNode\* ptail;

int size; //记录队列的元素个数}Queue;

//队列的初始化void QueueInit(Queue\* pq){

assert(pq);

pq->phead = pq->ptail = NULL;

pq->size = 0;}

//入队列void QueuePush(Queue\* pq, QDataType x){

assert(pq);

//申请新节点

QueueNode\* newnode = (QueueNode\*)malloc(sizeof(QueueNode));

if (newnode == NULL)

{

perror("malloc fail!");

exit(1);

}

//申请成功

newnode->data = x;

newnode->next = NULL;

if (pq->phead == NULL)

{

pq->phead = pq->ptail = newnode;

}

else

{

pq->ptail->next = newnode;

pq->ptail = newnode;

}

pq->size++;}

//队列判空bool QueueEmpty(Queue\* pq){

assert(pq);

return pq->phead == NULL && pq->ptail == NULL;}

//出队列void QueuePop(Queue\* pq){

assert(pq);

assert(!QueueEmpty(pq));

//只有一个结点的情况

if (pq->phead == pq->ptail)

{

free(pq->phead);

pq->phead = pq->ptail = NULL;

}

else

{

//有多个节点的情况

QueueNode\* next = pq->phead->next;

free(pq->phead);

pq->phead = next;

}

pq->size--;}

//取队头数据

QDataType QueueFront(Queue\* pq){

assert(pq);

assert(!QueueEmpty(pq));

return pq->phead->data;}

//取队尾数据

QDataType QueueBack(Queue\* pq){

assert(pq);

assert(!QueueEmpty(pq));

return pq->ptail->data;}

//取队列元素个数int QueueSize(Queue\* pq){

assert(pq);

return pq->size;}

//队列销毁void QueueDestroy(Queue\* pq){

assert(pq);

//assert(!QueueEmpty(pq)); 此题可以为空

QueueNode\* pcur = pq->phead;

while (pcur)

{

QueueNode\* next = pcur->next;

free(pcur);

pcur = next;

}

pq->phead = pq->ptail = NULL;

pq->size = 0;}

//////////////////////////两个队列来实现栈typedef struct {

Queue q1;

Queue q2;} MyStack;

MyStack\* myStackCreate() {

//初始化栈就是初始化两个队列

MyStack\* pst = (MyStack\* )malloc(sizeof(MyStack));

QueueInit(&pst->q1);

QueueInit(&pst->q2);

return pst;}

//入栈void myStackPush(MyStack\* obj, int x) {

if(!QueueEmpty(&obj->q1))

{

QueuePush(&obj->q1, x);

}

else

{

QueuePush(&obj->q2, x);

}}

//出栈int myStackPop(MyStack\* obj) {

//找不为空的队列

Queue\* empQ = &obj->q1; //为空队列

Queue\* noneQ = &obj->q2; //不为空队列

if(!QueueEmpty(empQ))

{

noneQ = &obj->q1;

empQ = &obj->q2;

}

//将不为空的队列数据size-1个导出

while(QueueSize(noneQ) > 1)

{

int front = QueueFront(noneQ);

QueuePush(empQ ,front);

QueuePop(noneQ);

}

//非空队列中只剩下要出栈的数据

int pop = QueueFront(noneQ);

QueuePop(noneQ);

return pop;

}

//取栈顶元素———取不为空队列的队尾元素int myStackTop(MyStack\* obj) {

if(!QueueEmpty(&obj->q1))

{

return QueueBack(&obj->q1);

}

else

{

return QueueBack(&obj->q2);

}

}

//判断栈是否为空,也就是判断这两个队列是否为空bool myStackEmpty(MyStack\* obj) {

return QueueEmpty(&obj->q1) && QueueEmpty(&obj->q2);}

//栈的销毁，就是两个队列的销毁void myStackFree(MyStack\* obj) {

QueueDestroy(&obj->q1);

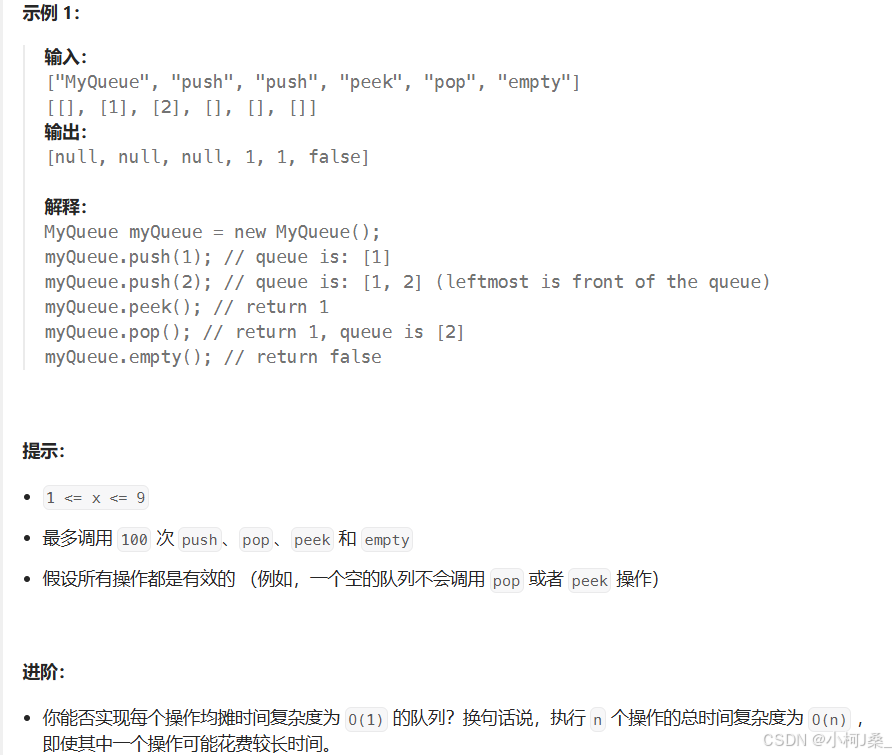
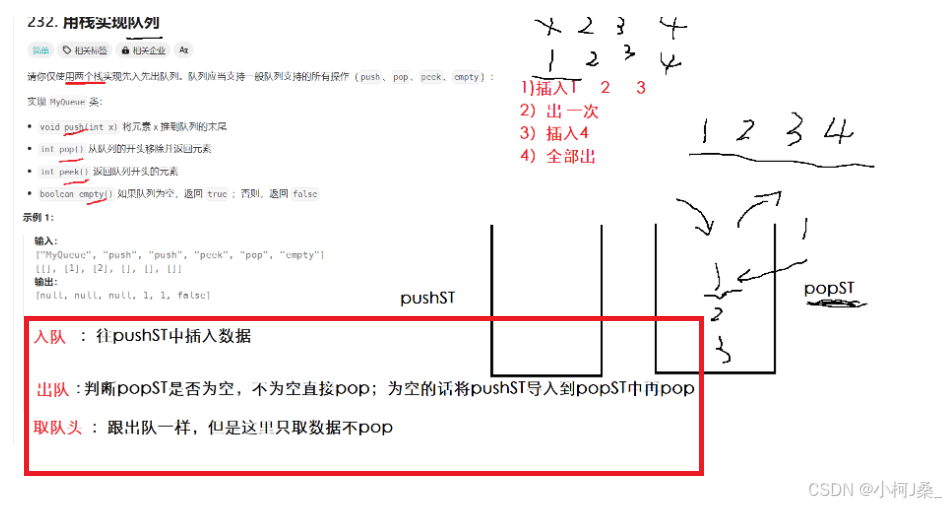
QueueDestroy(&obj->q2);

free(obj);

obj = NULL;}

三、用栈实现队列

<https://leetcode.cn/problems/implement-queue-using-stacks/description/>

typedef int STDataType;typedef struct Stack{

STDataType\* arr; //栈数组

int capacity; //栈的空间大小

int top; //栈顶位置}ST;

//栈的初始化void STInit(ST\* ps){

assert(ps);

ps->arr = NULL;

ps->capacity = 0;

ps->top = 0;}

//栈的销毁void STDestroy(ST\* ps){

assert(ps);

if(ps->arr)

free(ps->arr);

ps->arr = NULL;

ps->capacity = 0;

ps->top = 0;}

//数据入栈void StackPush(ST\* ps, STDataType x){

assert(ps);

if (ps->capacity == ps->top) //空间满了需要扩容

{

int newcapacity = ps->capacity == 0 ? 4 : 2 \* ps->capacity; //三目运算符如果原本栈为空，就赋初始为4个空间，若不为空，则双倍扩容

STDataType\* tem = (STDataType\*)realloc(ps->arr, newcapacity \* sizeof(STDataType));

//判断所开空间是否成功

if (tem == NULL)

{

perror("realloc fail!");

exit(1);

}

ps->arr = tem;

ps->capacity = newcapacity;

}

//入栈开始

ps->arr[ps->top++] = x;}

//栈判空bool StackEmpty(ST\* ps){

assert(ps);

return ps->top == 0;}

//数据出栈void StackPop(ST\* ps){

assert(ps);

assert(!StackEmpty(ps));

ps->top--;}

//取栈顶元素

STDataType StackTop(ST\* ps){

assert(ps);

assert(!StackEmpty(ps));

return ps->arr[ps->top - 1];}

//获取栈中有效元素个数int STSize(ST\* ps){

assert(ps);

return ps->top;}

///////////////////////////////////////////////用栈实现队列//定义队列，包含出数据的栈和入数据的栈typedef struct {

ST pushST;

ST popST;} MyQueue;

//队列的初始化

MyQueue\* myQueueCreate() {

//初始化队列就是初始化两个栈

MyQueue\* pst = (MyQueue\* )malloc(sizeof(MyQueue));

STInit(&pst->pushST);

STInit(&pst->popST);

return pst;}

//向队列中插入数据void myQueuePush(MyQueue\* obj, int x) {

//就是想pushST中插入数据

StackPush(&obj->pushST ,x);}

//出队列int myQueuePop(MyQueue\* obj) {

//若popSt为空,将pushSt的数据导入popST

if(StackEmpty(&obj->popST))

{

while(!StackEmpty(&obj->pushST))

{

StackPush(&obj->popST ,StackTop(&obj->pushST));

StackPop(&obj->pushST);

}

}

int x = StackTop(&obj->popST);

StackPop(&obj->popST);

return x;}

//获取对头元素，与出队列一样，只是不用popint myQueuePeek(MyQueue\* obj) {

//若popSt为空,将pushSt的数据导入popST

if(StackEmpty(&obj->popST))

{

while(!StackEmpty(&obj->pushST))

{

StackPush(&obj->popST ,StackTop(&obj->pushST));

StackPop(&obj->pushST);

}

}

int x = StackTop(&obj->popST);

return x;}

//判断队列是否为空，也就是两个栈是或否为空bool myQueueEmpty(MyQueue\* obj) {

return StackEmpty(&obj->popST) && StackEmpty(&obj->pushST);}

//队列的销毁，就是销毁栈void myQueueFree(MyQueue\* obj) {

STDestroy(&obj->popST);

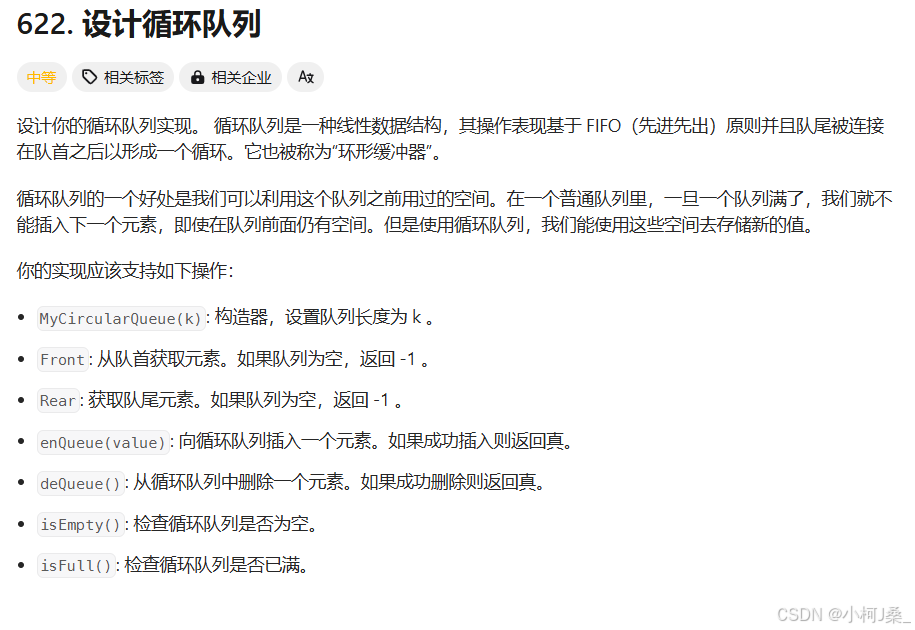
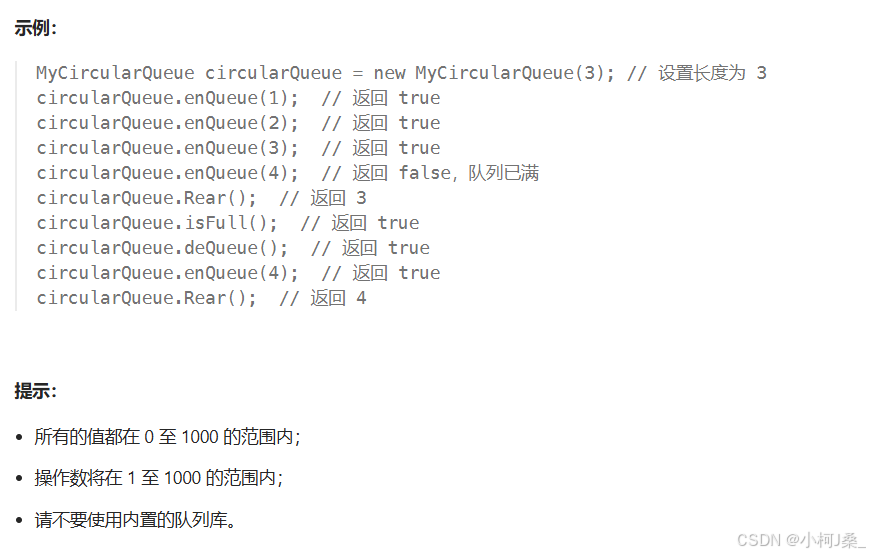
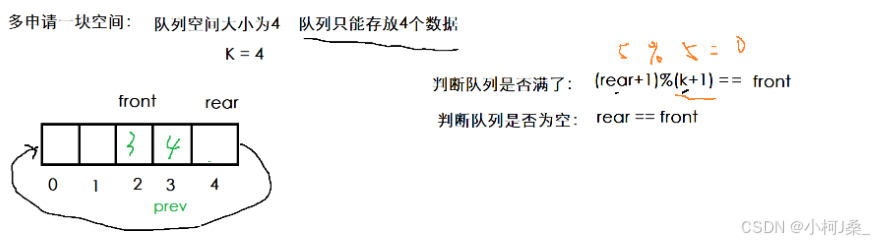
STDestroy(&obj->pushST);

//还要将创建的obj置空

free(obj);

obj = NULL;}

四、循环队列

<https://leetcode.cn/problems/design-circular-queue/description/>  
  
  


typedef struct {

int\* arr;

int front;

int rear;

int capacity;} MyCircularQueue;

//循环队列初始化

MyCircularQueue\* myCircularQueueCreate(int k) {

MyCircularQueue\* pst = (MyCircularQueue\* )malloc(sizeof(MyCircularQueue));

pst->arr = (int\* )malloc(sizeof(int)\*(k+1));

pst->front = pst->rear = 0;

pst->capacity = k;

return pst;}

//队列是否满了？bool myCircularQueueIsFull(MyCircularQueue\* obj) {

return (obj->rear + 1)%(obj->capacity + 1) == (obj->front);}

bool myCircularQueueEnQueue(MyCircularQueue\* obj, int value) {

//队列满了不能插入数据

if(myCircularQueueIsFull(obj))

{

return false;

}

//队列未满，开始插入数据

obj->arr[obj->rear++] = value;

obj->rear %=obj->capacity + 1;

return true;

}

//判断队列是否为空bool myCircularQueueIsEmpty(MyCircularQueue\* obj) {

return obj->front == obj->rear;}

//删除队列元素bool myCircularQueueDeQueue(MyCircularQueue\* obj) {

//如果队列为空不能删除队列元素

if(myCircularQueueIsEmpty(obj))

{

return false;

}

//开始删除队列元素

obj->front++;

obj->front %= obj->capacity + 1;

return true;}

//获取队列队头元素int myCircularQueueFront(MyCircularQueue\* obj) {

if(myCircularQueueIsEmpty(obj))

{

return -1;

}

return obj->arr[obj->front];}

int myCircularQueueRear(MyCircularQueue\* obj) {

if(myCircularQueueIsEmpty(obj))

{

return -1;

}

int prev = obj->rear - 1;

if(obj->rear == 0)

{

prev = obj->capacity;

}

return obj->arr[prev];}

//队列销毁void myCircularQueueFree(MyCircularQueue\* obj) {

free(obj->arr);

free(obj);

obj = NULL;}