栈和队列

1. StackByQueue.h文件

#pragma once

#include "Queue.h"

typedef struct {

Queue queue;

int maxSize;

} MyStack;

/\*\* Initialize your data structure here. \*/

MyStack\* myStackCreate(int maxSize) {

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

QueueInit(&(stack->queue));

stack->maxSize = maxSize;

return stack;

}

/\*\* Push element x onto stack. \*/

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

QueuePush(&(obj->queue), x);

}

/\*\* Removes the element on top of the stack and returns that element. \*/

int myStackPop(MyStack\* obj) {

int size = QueueSize(&(obj->queue));

for (int i = 0; i < size - 1; i++) {

int val = QueueFront(&(obj->queue));

QueuePop(&(obj->queue));

QueuePush(&(obj->queue), val);

}

int val = QueueFront(&(obj->queue));

QueuePop(&(obj->queue));

return val;

}

/\*\* Get the top element. \*/

int myStackTop(MyStack\* obj) {

int size = QueueSize(&(obj->queue));

for (int i = 0; i < size - 1; i++) {

int val = QueueFront(&(obj->queue));

QueuePop(&(obj->queue));

QueuePush(&(obj->queue), val);

}

int val = QueueFront(&(obj->queue));

QueuePop(&(obj->queue));

QueuePush(&(obj->queue), val);

return val;

}

/\*\* Returns whether the stack is empty. \*/

bool myStackEmpty(MyStack\* obj) {

return QueueEmpty(&(obj->queue));

}

void myStackFree(MyStack\* obj) {

QueueDestroy(&(obj->queue));

free(obj);

}

1. Stack.h文件

#pragma once

typedef struct Stack {

int array[100];

int top;

} Stack;

// 初始化

void StackInit(Stack \*stack) {

stack->top = 0;

}

// 压栈（尾插）

void StackPush(Stack \*stack, int value) {

stack->array[stack->top] = value;

stack->top++;

}

// 出栈、弹出（尾删）

void StackPop(Stack \*stack) {

stack->top--;

}

// 返回栈顶元素

int StackTop(const Stack \*stack) {

return stack->array[stack->top - 1];

}

// 判断是否为空

int StackEmpty(const Stack \*stack) {

return stack->top == 0 ? 1 : 0;

}

// 获取 size

int StackSize(const Stack \*stack) {

return stack->top;

}

// 1. 括号匹配

bool isValid(char\* s) {

Stack stack;

StackInit(&stack);

while (\*s != '\0') {

switch (\*s) {

case '{':

case '[':

case '(':

StackPush(&stack, (int)\*s);

break;

case '}':

case ']':

case ')': {

if (StackEmpty(&stack)) {

return false;

}

char left = (char)StackTop(&stack);

StackPop(&stack);

if (!(left == '(' && \*s == ')'

|| left == '[' && \*s == ']'

|| left == '{' && \*s == '}')) {

return false;

}

break;

}

default:

break;

}

s++;

}

if (!StackEmpty(&stack)) {

return false;

}

return true;

}

1. QueueByStack.h文件

#pragma once

class MyQueue {

public:

/\*\* Initialize your data structure here. \*/

MyQueue() {

}

/\*\* Push element x to the back of queue. \*/

void push(int x) {

left.push(x);

}

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

int pop() {

if (right.empty()) {

while (!left.empty()) {

int top = left.top();

left.pop();

right.push(top);

}

}

int top = right.top();

right.pop();

return top;

}

/\*\* Get the front element. \*/

int peek() {

if (right.empty()) {

while (!left.empty()) {

int top = left.top();

left.pop();

right.push(top);

}

}

int top = right.top();

return top;

}

/\*\* Returns whether the queue is empty. \*/

bool empty() {

return left.empty() && right.empty();

}

stack<int> left;

stack<int> right;

};

1. MinStack\_2.h文件

class MinStack {

public:

/\*\* initialize your data structure here. \*/

MinStack() {

}

void push(int x) {

left.push(x);

if (right.empty() || x <= right.top()) {

right.push(x);

}

}

void pop() {

int top = left.top();

left.pop();

if (top == right.top()) {

right.pop();

}

}

int top() {

return left.top();

}

int getMin() {

return right.top();

}

stack<int> left; // 存正常的数据;

stack<int> right; // 存当前最小的数据;

};

1. MinStack.h文件

class MinStack {

public:

/\*\* initialize your data structure here. \*/

MinStack() {

}

void push(int x) {

left.push(x);

int min;

if (!right.empty() && right.top() <= x) {

min = right.top();

}

else {

min = x;

}

right.push(min);

}

void pop() {

left.pop();

right.pop();

}

int top() {

return left.top();

}

int getMin() {

return right.top();

}

stack<int> left; // 存正常的数据;

stack<int> right; // 存当前最小的数据;

};

1. CircularQueue.h文件

class MyCircularQueue {

public:

/\*\* Initialize your data structure here. Set the size of the queue to be k. \*/

MyCircularQueue(int k) {

array = (int \*)malloc(sizeof(int)\* k);

capacity = k;

size = 0;

front = 0;

rear = 0;

}

/\*\* Insert an element into the circular queue. Return true if the operation is successful. \*/

bool enQueue(int value) {

if (size == capacity) {

return false;

}

array[rear] = value;

rear = (rear + 1) % capacity;

size++;

return true;

}

/\*\* Delete an element from the circular queue. Return true if the operation is successful. \*/

bool deQueue() {

if (size == 0) {

return false;

}

front = (front + 1) % capacity;

size--;

return true;

}

/\*\* Get the front item from the queue. \*/

int Front() {

if (size == 0) {

return -1;

}

return array[front];

}

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

int Rear() {

if (size == 0) {

return -1;

}

return array[(rear + capacity - 1) % capacity];

}

/\*\* Checks whether the circular queue is empty or not. \*/

bool isEmpty() {

return size == 0;

}

/\*\* Checks whether the circular queue is full or not. \*/

bool isFull() {

return size == capacity;

}

int \*array;

int capacity;

int size;

int front;

int rear;

};