**题9.1已经在上次作业交了**

**11.twoStacks类**

在一个数组中实现左右两个栈，两栈均向中间生长。

#pragma once

#include<cstring>

#include<cstdlib>

using namespace std;

/\*twoStacks类

public:

enum stack\_err; 常见错误

protected:

void \_exLength (); 扩展缓冲区长度

int stackTop1; 栈顶下标

int stackTop2; 栈顶下标

int bufferLength; 缓冲区长度

T\* head; 栈数组头部

public:

stack ( int initialCapacity = 10 ); 构造函数，默认长度10

~stack () ; 析构函数，释放缓冲区

bool empty1 () const ; 返回是否空

bool empty2 () const ; 返回是否空

int size1 () const ; 返回长度

int size2 () const ; 返回长度

T top1 (); 返回栈顶元素

T top2 (); 返回栈顶元素

void pop1 (); 弹出栈顶元素，一般不会返回值

void pop2 (); 弹出栈顶元素，一般不会返回值

void push1 ( const T theElement ); 压入元素

void push2 ( const T theElement ); 压入元素

\*/

template<class T>

class twoStacks {

public:

typedef enum { twoStacks1\_empty, twoStacks2\_empty } twoStacks\_err;

protected:

void \_exLength () {

T\* temp = new T[bufferLength \* 2];

memcpy ( temp, head, ( twoStacksTop1 + 1 ) \* sizeof ( T ) );

memcpy ( temp + bufferLength \* 2 - ( bufferLength - twoStacksTop2 ),

head + twoStacksTop2,

( bufferLength - twoStacksTop2 ) \* sizeof ( T ) );

delete[] head;

head = temp;

twoStacksTop2 = bufferLength + twoStacksTop2;

bufferLength = bufferLength \* 2;

}

int twoStacksTop1;

int twoStacksTop2;

int bufferLength;

T\* head;

public:

twoStacks ( int initialCapacity = 50 ) {

bufferLength = initialCapacity;

head = new T[bufferLength];

twoStacksTop1 = -1;

twoStacksTop2 = bufferLength;

}

~twoStacks () { delete[] head; }

bool empty1 () const { return twoStacksTop1 == -1; }

bool empty2 () const { return twoStacksTop2 == bufferLength; }

int size1 () const { return twoStacksTop1 + 1; }

int size2 () const { return bufferLength - twoStacksTop2; }

T top1 () {

//判断是否为空

//有的实现版本中top输入引用，栈空则返回原数值，这样很不符合职责单一原则

if ( empty1 () ) {

throw twoStacks1\_empty;

}

return head[twoStacksTop1];

}

T top2 () {

//判断是否为空

//有的实现版本中top输入引用，栈空则返回原数值，这样很不符合职责单一原则

if ( empty2 () ) {

throw twoStacks2\_empty;

}

return head[twoStacksTop2];

}

void pop1 () {

if ( empty1 () ) {

throw twoStacks1\_empty;

}

head[twoStacksTop1--].~T ();

}

void pop2 () {

if ( empty2 () ) {

throw twoStacks2\_empty;

}

head[twoStacksTop2++].~T ();

}

void push1 ( const T theElement ) {

//判断缓冲区长度并扩大缓冲区

if ( twoStacksTop1 + 1 == twoStacksTop2 ) {

\_exLength ();

}

head[++twoStacksTop1] = theElement;

}

void push2 ( const T theElement ) {

//判断缓冲区长度并扩大缓冲区

if ( twoStacksTop1 + 1 == twoStacksTop2 ) {

\_exLength ();

}

head[--twoStacksTop2] = theElement;

}

};

经测试，运行正常

除改变数组大小外的操作复杂度均为：O(1)

由代码可知，所有方法操作数量固定，都用不到循环与递归，其时间复杂度为O(1)

**第九章：队列**

**7.queueBack变queueSize**

使用queueSize代替queueBack，重新实现队列

#pragma once

#include<cstdlib>

#include<cstdio>

#include<cstring>

using namespace std;

/\*

本题关键：

一在于使用theFront、queueSize、arrayLength这三个量确定theBack的位置

二在于扩大数组的时候考虑队列横跨数组两头的情况。

\*/

/\*queue类

private:

int theFront; 指向左侧

int queueSize; 队列长度

int arrayLength; 数组长度

T\* head; 数组头

void \_exLength (); 扩展数组长度函数

public:

enum queue\_err; 常见错误

queue ( int initialCapacity = 5 ); 构造函数

~queue (); 析构函数

bool empty () const; 返回是否为空

int size () const; 返回长度

T front (); 返回头部元素

T back (); 返回尾部元素

void pop (); 弹出头部元素

void push ( const T& theElement ); 压入尾部

\*/

template<class T>

class queue {

private:

int theFront;

int queueSize;

int arrayLength;

T\* head;

void \_exLength () {

/\*申请新空间\*/

T\* newhead = new T[(arrayLength\*2)];

/\*复制内存\*/

if ( theFront + queueSize <= arrayLength ) {

/\*不跨越头尾时，只需复制整个旧数组\*/

memcpy ( newhead + theFront, head + theFront, queueSize \* sizeof ( T ) );

arrayLength = arrayLength \* 2;

} else {

/\*扩大数组的时候考虑队列横跨数组两头的情况，需要分别复制两头\*/

//复制右侧

memcpy ( newhead + arrayLength + theFront, head + theFront, ( arrayLength - theFront ) \* sizeof ( T ) );

//复制左侧

memcpy ( newhead, head, ( ( theFront + arrayLength ) % arrayLength )\* sizeof ( T ) );

//复制右侧时，theFront的位置向右移动，需要确定新的位置

theFront = arrayLength + theFront;

}

delete[] head;

head = newhead;

}

public:

typedef enum { queue\_empty } queue\_err;

queue ( int initialCapacity = 5 ) {

arrayLength = initialCapacity;

head = new T[arrayLength];

theFront = 0;

queueSize = 0;

}

~queue () { delete[] head; }

bool empty () const { return queueSize == 0; }

int size () const { return queueSize; }

T front () {

if ( empty () )

throw queue\_empty;

return head[theFront];

}

T back () {

if ( empty () )

throw queue\_empty;

return head[( theFront + queueSize ) % arrayLength - 1];

}

void pop () {

if ( empty () )

throw queue\_empty;

head[theFront].~T ();

theFront = ( theFront + 1 ) % arrayLength;

queueSize--;

}

void push ( const T& theElement ) {

/\*进行长度判断\*/

if ( queueSize == arrayLength ) {

\_exLength ();

}

head[(theFront+queueSize)%arrayLength] = theElement;

queueSize++;

}

};

经测试，运行正常

**8.加入lastOp**

使用lastOp判断满空

#pragma once

#include<cstdlib>

#include<cstdio>

#include<cstring>

using namespace std;

/\*

本题关键：

上一题我采用front与back均指向头尾元素的形式，本题中为了出现front==back的情况，应当是front指向队列头左侧的空位置。

这样一来，本题关键：

一在于在front==back时使用lastOp来确定队列的空与满，为pop则空，为push则满。

二在于考虑数组复制时“复制两头”的情况（同上题）。

同时，由于theFront指向的位置改变，其他所有相关代码也要改变

\*/

/\*queue

public:

typedef enum { queue\_empty } queue\_err; 代表常见错误

typedef enum { POP, PUSH } lastOp\_enum; 代表pop与push两种状态

private:

int theFront; 队列头，在数组中指向队列头左侧空位

int theBack; 队列尾，指向队列尾元素

lastOp\_enum lastOp; 保存上次操作

int arrayLength; 数组长度

T\* head; 数组头部

void \_exLength (); 数组扩展函数，较为重要

public:

queue ( int initialCapacity = 5 ); 构造函数

~queue (); 析构函数

bool isfull () const; 判断是否满，是题目重点

bool isempty ()const; 判断是否空，是题目重点

int size () const; 返回队列长度

T front (); 返回头部元素

T back (); 返回尾部元素

void pop (); 弹出头部元素

void push ( const T& theElement ); 元素入队列

\*/

template<class T>

class queue {

public:

typedef enum { queue\_empty } queue\_err;

typedef enum { POP, PUSH } lastOp\_enum;

private:

int theFront;//为了完成题意，theFront指向队列左侧减一的位置

int theBack;

lastOp\_enum lastOp;

int arrayLength;

T\* head;

void \_exLength () {

T\* newhead = new T[2 \* arrayLength];

if ( theFront == arrayLength - 1 ) {

/\*整体复制到新数组左侧\*/

memcpy ( newhead + theFront + 1, head + theFront + 1,( theBack - theFront) \* sizeof ( T ) );

theFront = arrayLength \* 2 - 1;

} else {

/\*分别复制到新数组两边\*/

//右边

int newf = arrayLength + theFront;

memcpy ( newhead + newf + 1, head + theFront + 1, ( arrayLength - 1 - theFront )\*sizeof(T) );

//左边

memcpy ( newhead, head, (theBack + 1) \* sizeof ( T ) );

theFront = newf;

//一开始不小心忘记加sizeof，导致debug一段时间；

//一开始-1-写成--1，造成VS出现从未见过的报错，复制到dev cpp中调试，发现是-1-写成了--1

}

arrayLength = arrayLength \* 2;

delete[] head;

head = newhead;

}

public:

queue ( int initialCapacity = 5 ) {

arrayLength = initialCapacity;

head = new T[arrayLength];

theFront = 0;

theBack = 0;

lastOp = POP;

}

~queue () { delete[] head; }

/\*如果push到 theBack == theFront 则满，如果pop到 theBack == theFront 则空\*/

bool isfull () const { return ( theBack == theFront ) && ( lastOp == PUSH ); }

bool isempty ()const { return ( theBack == theFront ) && ( lastOp == POP ); }

int size () const {

/\*返回长度也要判断当前处于 theBack == theFront 的哪种种情况\*/

if ( isfull () ) {

return arrayLength;

} else {

return ( theBack + arrayLength - theFront ) % arrayLength;

}

}

T front () {

if ( isempty () )

throw queue\_empty;

return head[( theFront + 1 ) % arrayLength];

}

T back () {

if ( isempty () )

throw queue\_empty;

return head[theBack];

}

void pop () {

if ( isempty () )

throw queue\_empty;

head[( theFront + 1 ) % arrayLength].~T ();

theFront = ( theFront + 1 ) % arrayLength;

lastOp = POP;

}

void push ( const T& theElement ) {

if ( isfull () ) {

\_exLength ();

}

theBack = ( theBack + 1 ) % arrayLength;

head[theBack] = theElement;

lastOp = PUSH;

}

};

经测试，运行正常