## 2021.9.28 课后编程练习（队列）：陈鹏宇—20204227

#### Aqueue.h

#ifndef AQUEUE\_H

#define AQUEUE\_H

#define defaultSize 100

#include "queue.h"

void Assert(bool b , std::string s)

{

if (!b) {

std::cout << "Assertion Failed: " << s << std::endl;

exit(-1);

}

}

/\* //原始版本 \*/

template <typename E>

class AQueue:public Queue<E>

{

private:

int maxSize;

int front;

int rear;

E \*listArray;

public:

AQueue(int size = defaultSize){

maxSize = size + 1;

rear = 0 ; front = 1;

listArray = new E[maxSize];

}

~AQueue(){delete [] listArray;}

void clear(){rear = 0 ; front = 1;}

void enqueue(const E& it){

Assert((rear+2) % maxSize != front , "Queue is full");

rear = (rear+1) % maxSize;

listArray[rear] = it;

}

E dequeue(){

Assert(length() != 0 , "Queue is empty");

E it = listArray[front];

front = (front+1) % maxSize;

return it;

}

const E& frontValue() const{

Assert(length() != 0 , "Queue is empty");

return listArray[front];

}

virtual int length() const{

return ((rear + maxSize) - front + 1) % maxSize;

}

};

/\* //使用一个独立的布尔成员记录队列是否为空，而不必在数组中留一个空位置。

template <typename E>

class AQueue:public Queue<E>

{

private:

int maxSize;

int front;

int rear;

E \*listArray;

bool empty; //记录队列是否为空

public:

AQueue(int size = defaultSize){

maxSize = size; //储存size个元素

rear = 0 ; front = 1;

listArray = new E[maxSize];

empty = true; //初始为空

}

~AQueue(){delete [] listArray;}

void clear(){rear = 0 ; front = 1;}

void enqueue(const E& it){

Assert(((rear + 1) % maxSize != front) || empty , "Queue is full"); //空条件用来区分初始状态和满状态

if(((rear + 1) % maxSize != front) || empty) //空条件用来区分初始状态和满状态

rear = (rear + 1) % maxSize;

listArray[rear] = it;

if(empty) //如果为空，则不为空

empty = false;

}

E dequeue(){

Assert(!empty , "Queue is empty");

E it = listArray[front];

front = (front+1) % maxSize;

if((rear + 1) % maxSize == front) //front 重新回到rear前面 为空

empty = true;

return it;

}

const E& frontValue() const{

Assert(!empty , "Queue is empty");

return listArray[front];

}

virtual int length() const{ //数组不留空

if(rear == front)

return 0;

else if(rear > front)

return rear - front + 1;

else

return rear + maxSize - front + 1;

}

};

\*/

/\* //双端队列版本

template <typename E>

class AQueue:public Queue<E>

{

private:

int maxSize;

int front;

int rear;

E \*listArray;

public:

AQueue(int size = defaultSize){

maxSize = size + 1;

rear = 0 ; front = 1;

listArray = new E[maxSize];

}

~AQueue(){delete [] listArray;}

void clear(){rear = 0 ; front = 1;}

void first\_insert(const E& it) //我们认为，第一个元素无论是从队尾还是队首插入，最终效果都一样，通过length来判断是否为第一个元素

{

rear = (rear+1) % maxSize;

listArray[rear] = it;

}

void enqueue(const E& it){ //队尾插入元素

if(length() == 0)

{

first\_insert(it);

}

else

{

Assert((rear+2) % maxSize != front , "Queue is full");

rear = (rear+1) % maxSize;

listArray[rear] = it;

}

}

void enqueue\_front(const E& it) //队首插入元素

{

if(length() == 0)

{

first\_insert(it);

return;

}

else

{

Assert((rear+2) % maxSize != front , "Queue is full");

front = (front - 1) % maxSize;

listArray[front] = it;

}

}

E dequeue() //队首删除元素

{

Assert(length() != 0 , "Queue is empty");

E it = listArray[front];

front = (front+1) % maxSize;

return it;

}

E dequeue\_rear() //队尾删除元素

{

Assert(length() != 0 , "Queue is empty");

E it = listArray[rear];

rear = (rear - 1) % maxSize;

return it;

}

const E& frontValue() const{

Assert(length() != 0 , "Queue is empty");

return listArray[front];

}

virtual int length() const{

return ((rear + maxSize) - front + 1) % maxSize;

}

};

\*/

#endif // AQUEUE\_H

#### Lqueue.h

#ifndef LQUEUE\_H

#define LQUEUE\_H

#define defaultSize 100

#include "queue.h"

#include "aqueue.h"

#include "link.h"

/\* //原始版本\*/

template <typename E>

class LQueue:public Queue<E>

{

private:

Link<E> \* front;

Link<E> \* rear;

int size;

public:

LQueue(int sz = defaultSize){

front = rear = new Link<E>;

size = 0;

}

~LQueue(){

clear(); delete front;

}

void clear(){

while(front->next != nullptr)

{

rear = front;

front = front->next;

delete rear;

}

rear = front;

size = 0;

}

void enqueue(const E& it){

rear->next = new Link<E>(it,nullptr);

rear = rear->next;

size++;

}

E dequeue(){

Assert(size != 0 , "Queue is empty");

E it = front->next->element;

Link<E>\* ltemp = front->next;

front->next = ltemp->next;

if(rear == ltemp) rear = front;

delete ltemp;

size--;

return it;

}

const E& frontValue() const{

Assert(size != 0 , "Queue is empty");

return front->next->element;

}

virtual int length() const{

return size;

}

};

/\* //双端队列版本

template <typename E>

class LQueue:public Queue<E>

{

private:

Link<E> \* front;

Link<E> \* rear;

int size;

public:

LQueue(int sz = defaultSize){

front = rear = new Link<E>;

size = 0;

}

~LQueue(){

clear(); delete front;

}

void clear(){

while(front->next != nullptr)

{

rear = front;

front = front->next;

delete rear;

}

rear = front;

size = 0;

}

void first\_insert(const E& it) //我们认为，第一个元素无论是从队尾还是队首插入，最终效果都一样，通过length来判断是否为第一个元素

{

rear->next = new Link<E>(it,nullptr);

rear = rear->next;

size++;

}

void enqueue(const E& it){ //队尾插入元素

if(size == 0)

{

first\_insert(it);

}

else

{

rear->next = new Link<E>(it,nullptr);

rear = rear->next;

size++;

}

}

void enqueue\_front(const E& it) //队首插入元素

{

if(size == 0)

{

first\_insert(it);

}

else

{

front->next = new Link<E>(it , front->next);

size++;

}

}

E dequeue()

{

Assert(size != 0 , "Queue is empty");

E it = front->next->element;

Link<E>\* ltemp = front->next;

front->next = ltemp->next;

if(rear == ltemp) rear = front;

delete ltemp;

size--;

return it;

}

E dequeue\_rear() //从队尾删除元素

{

Assert(size != 0 , "Queue is empty");

E it = rear->element;

Link<E> ltemp = rear; //为了让rear指向前一个元素，在单向链表中只有遍历了

if(front->next == ltemp) rear = front;

else

{

Link<E> t = front;

while(t.next != rear)

{

t = t.next;

}

rear = t;

}

delete ltemp;

size--;

return it;

}

const E& frontValue() const{

Assert(size != 0 , "Queue is empty");

return front->next->element;

}

virtual int length() const{

return size;

}

};

\*/

#endif // LQUEUE\_H

#### Queue.h

#ifndef QUEUE\_H

#define QUEUE\_H

#include <iostream>

template <typename E>

class Queue

{

private:

void operator = (const Queue&) {}

Queue(const Queue&) {}

public:

Queue() {}

virtual ~Queue() {}

virtual void clear() = 0;

virtual void enqueue(const E&) = 0;

virtual E dequeue() = 0;

virtual const E& frontValue() const = 0;

virtual int length() const =0;

};

#endif // QUEUE\_H

#### Link.h

#ifndef LINK\_H

#define LINK\_H

#include <cstddef>

/\*单向链表\*/

template <typename E>

class Link

{

private:

static Link<E>\* freelist;

public:

E element;

Link \*next;

Link(const E& elemval , Link\* nextval = nullptr){ element = elemval; next = nextval; }

Link(Link\* nextval = nullptr) { next = nextval; }

void\* operator new(size\_t){

if(freelist == nullptr) return ::new Link;

Link<E>\* temp = freelist;

freelist =freelist->next;

return temp;

}

void operator delete(void\* ptr){

((Link<E>\*)ptr)->next = freelist;

freelist = (Link<E>\*)ptr;

}

};

template <typename E>

Link<E>\* Link<E>::freelist = nullptr;

#endif // LINK\_H

#### Stack.h

#ifndef STACK\_H

#define STACK\_H

template <typename E>

class Stack

{

private:

void operator = (const Stack&) {}

Stack(const Stack&) {}

public:

Stack() {};

virtual ~Stack() {};

virtual void clear() = 0;

virtual void push(const E& it) = 0;

virtual E pop() = 0;

virtual const E& topValue() const = 0;

virtual int length() const = 0;

};

#endif // STACK\_H

#### Astack.h

#ifndef ASTACK\_H

#define ASTACK\_H

#define defaultSize 100

#include "stack.h"

#include <iostream>

template <typename E>

class AStack : public Stack<E>

{

private:

int maxSize;

int top;

E \*listArray;

public:

void Assert(bool val, std::string s)

{

if (!val) {

std::cout << "Assertion Failed: " << s << std::endl;

exit(-1);

}

}

AStack(int size = defaultSize)

{

maxSize = size;

top = 0;

listArray = new E[size];

}

~AStack()

{

delete [] listArray;

}

void clear()

{

top = 0;

}

void push(const E& it)

{

Assert(top != maxSize , "Stack is full");

listArray[top++] = it;

}

E pop()

{

Assert(top != 0 , "Stack is empty");

return listArray[--top];

}

const E& topValue() const

{

//Assert(top >= 1 , "Stack is empty");

return listArray[top-1];

}

int length() const

{

return top;

}

};

#endif // ASTACK\_H

#### 1. 修改图4.26顺序队列的定义，使用一个独立的布尔成员记录队列是否为空，而不必在数组中留一个空位置。

修改aqueue.h为

#ifndef AQUEUE\_H

#define AQUEUE\_H

#define defaultSize 100

#include "queue.h"

void Assert(bool b , std::string s)

{

if (!b) {

std::cout << "Assertion Failed: " << s << std::endl;

exit(-1);

}

}

/\* //使用一个独立的布尔成员记录队列是否为空，而不必在数组中留一个空位置。\*/

template <typename E>

class AQueue:public Queue<E>

{

private:

int maxSize;

int front;

int rear;

E \*listArray;

bool empty; //记录队列是否为空

public:

AQueue(int size = defaultSize){

maxSize = size; //储存size个元素

rear = 0 ; front = 1;

listArray = new E[maxSize];

empty = true; //初始为空

}

~AQueue(){delete [] listArray;}

void clear(){rear = 0 ; front = 1;}

void enqueue(const E& it){

Assert(((rear + 1) % maxSize != front) || empty , "Queue is full"); //空条件用来区分初始状态和满状态

if(((rear + 1) % maxSize != front) || empty) //空条件用来区分初始状态和满状态

rear = (rear + 1) % maxSize;

listArray[rear] = it;

if(empty) //如果为空，则不为空

empty = false;

}

E dequeue(){

Assert(!empty , "Queue is empty");

E it = listArray[front];

front = (front+1) % maxSize;

if((rear + 1) % maxSize == front) //front 重新回到rear前面 为空

empty = true;

return it;

}

const E& frontValue() const{

Assert(!empty , "Queue is empty");

return listArray[front];

}

virtual int length() const{ //数组不留空

if(rear == front)

return 0;

else if(rear > front)

return rear - front + 1;

else

return rear + maxSize - front + 1;

}

};

#endif // AQUEUE\_H

Main函数

#include <iostream>

#include "aqueue.h"

#include "lqueue.h"

#include "astack.h"

using namespace std;

int main()

{

AQueue<int> a(6);

for(int i = 0 ; i < 6 ; i++)

{

int temp ; cin >> temp;

a.enqueue(temp);

}

cout<<a.length()<<endl;

for(int i = 0 ; i < 6 ; i++)

{

cout<<a.frontValue()<<endl;

a.dequeue();

}

}

文本

描述已自动生成

#### 2. 双端队列（deque）是一种队列，但是它的元素同时可以从队首和队尾进行插入和删除操作，编写一个双端队列，分别用数组和链表实现。（可以参考用一个数组实现2个栈的实现）.

##### 数组版本：

#ifndef AQUEUE\_H

#define AQUEUE\_H

#define defaultSize 100

#include "queue.h"

void Assert(bool b , std::string s)

{

if (!b) {

std::cout << "Assertion Failed: " << s << std::endl;

exit(-1);

}

}

/\* //双端队列版本 \*/

template <typename E>

class AQueue:public Queue<E>

{

private:

int maxSize;

int front;

int rear;

E \*listArray;

public:

AQueue(int size = defaultSize){

maxSize = size + 1;

rear = 0 ; front = 1;

listArray = new E[maxSize];

}

~AQueue(){delete [] listArray;}

void clear(){rear = 0 ; front = 1;}

void first\_insert(const E& it) //我们认为，第一个元素无论是从队尾还是队首插入，最终效果都一样，通过length来判断是否为第一个元素

{

rear = (rear+1) % maxSize;

listArray[rear] = it;

}

void enqueue(const E& it){ //队尾插入元素

if(length() == 0)

{

first\_insert(it);

return;

}

else

{

Assert((rear+2) % maxSize != front , "Queue is full");

rear = (rear+1) % maxSize;

listArray[rear] = it;

}

}

void enqueue\_front(const E& it) //队首插入元素

{

if(length() == 0)

{

first\_insert(it);

return;

}

else

{

Assert((rear+2) % maxSize != front , "Queue is full");

front = (front - 1) % maxSize;

listArray[front] = it;

}

}

E dequeue() //队首删除元素

{

Assert(length() != 0 , "Queue is empty");

E it = listArray[front];

front = (front+1) % maxSize;

return it;

}

E dequeue\_rear() //队尾删除元素

{

Assert(length() != 0 , "Queue is empty");

E it = listArray[rear];

rear = (rear - 1) % maxSize;

return it;

}

const E& frontValue() const{

Assert(length() != 0 , "Queue is empty");

return listArray[front];

}

virtual int length() const{

return ((rear + maxSize) - front + 1) % maxSize;

}

};

#endif // AQUEUE\_H

Main函数

#include "lqueue.h"

#include "astack.h"

using namespace std;

int main(){

AQueue<int> a(6);

a.enqueue(1); //这里交叉插入元素

a.enqueue\_front(2);

a.enqueue(3);

a.enqueue\_front(4);

a.enqueue(5);

a.enqueue\_front(6);

cout<<"expected answer: 6 4 2 1 3 5"<<endl;

cout<<"answer: ";

for(int i = 0 ; i < 6 ; i++)

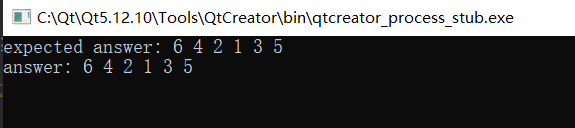
{

cout<<a.frontValue()<<" ";

a.dequeue(); //含顺序删除理论输出为642135

}

}



##### 链表版本：

#ifndef LQUEUE\_H

#define LQUEUE\_H

#define defaultSize 100

#include "queue.h"

#include "aqueue.h"

#include "link.h"

/\* //双端队列版本 \*/

template <typename E>

class LQueue:public Queue<E>

{

private:

Link<E> \* front;

Link<E> \* rear;

int size;

public:

LQueue(int sz = defaultSize){

front = rear = new Link<E>;

size = 0;

}

~LQueue(){

clear(); delete front;

}

void clear(){

while(front->next != nullptr)

{

rear = front;

front = front->next;

delete rear;

}

rear = front;

size = 0;

}

void first\_insert(const E& it) //我们认为，第一个元素无论是从队尾还是队首插入，最终效果都一样，通过length来判断是否为第一个元素

{

rear->next = new Link<E>(it,nullptr);

rear = rear->next;

size++;

}

void enqueue(const E& it){ //队尾插入元素

if(size == 0)

{

first\_insert(it);

}

else

{

rear->next = new Link<E>(it,nullptr);

rear = rear->next;

size++;

}

}

void enqueue\_front(const E& it) //队首插入元素

{

if(size == 0)

{

first\_insert(it);

}

else

{

front->next = new Link<E>(it , front->next);

size++;

}

}

E dequeue()

{

Assert(size != 0 , "Queue is empty");

E it = front->next->element;

Link<E>\* ltemp = front->next;

front->next = ltemp->next;

if(rear == ltemp) rear = front;

delete ltemp;

size--;

return it;

}

E dequeue\_rear() //从队尾删除元素

{

Assert(size != 0 , "Queue is empty");

E it = rear->element;

Link<E> ltemp = rear; //为了让rear指向前一个元素，在单向链表中只有遍历了

if(front->next == ltemp) rear = front;

else

{

Link<E> t = front;

while(t.next != rear)

{

t = t.next;

}

rear = t;

}

delete ltemp;

size--;

return it;

}

const E& frontValue() const{

Assert(size != 0 , "Queue is empty");

return front->next->element;

}

virtual int length() const{

return size;

}

};

#endif // LQUEUE\_H

Main函数

#include <iostream>

#include "aqueue.h"

#include "lqueue.h"

#include "astack.h"

using namespace std;

int main(){

LQueue<int> a(6);

a.enqueue(1); //这里交叉插入元素

a.enqueue\_front(2);

a.enqueue(3);

a.enqueue\_front(4);

a.enqueue(5);

a.enqueue\_front(6);

cout<<"expected answer: 6 4 2 1 3 5"<<endl;

cout<<"answer: ";

for(int i = 0 ; i < 6 ; i++)

{

cout<<a.frontValue()<<" ";

a.dequeue(); //含顺序删除理论输出为642135

}

}

文本

描述已自动生成

#### 3.回文是指一个字符串从前面读和从后面读都一样。仅使用若干栈和队列ADT函数及若干个int类型和char类型的变量，编写一个算法，判断一个字符串是否为回文。假设字符串从标准输入设备一次读入一个字符，算法的输出结果应为true或者false。

#include <iostream>

#include "aqueue.h"

#include "lqueue.h"

#include "astack.h"

using namespace std;

template <typename E>

bool isPalindrome(AQueue<E> &a , AStack<E> &b)

{

int n = a.length();

for(int i = 0 ; i < n/2 ; i++)

{

if(a.frontValue() != b.topValue())

{

return false;

}

else

{

a.dequeue();

b.pop();

}

}

return true;

}

int main(){

string s ; getline(cin,s);

while(s != "q") //输入q停止程序

{

AQueue<char> aq ; AStack<char> as;

int n = s.size();

for(int i = 0 ; i < n ; i++)

{

char t = s[i] ; aq.enqueue(t) ; as.push(t);

}

if(isPalindrome(aq,as))

cout<<"true"<<endl;

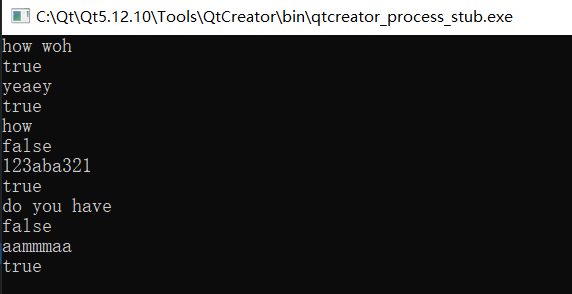
else

cout<<"false"<<endl;

getline(cin,s);

}

}



#### 4.编程实现使用循环队列作为存储结构解决约瑟夫问题。Josehus问题的描述：数字1，2，…，n，n个人顺时针围着一个圆圈坐着，每个人都持有一个密码（正整数）。选择一个正整数开头作为报数m的限制，从第一人开始按顺时针方向从一个报数开始，报数m停止报数。谁报了m则出列，他的密码作为新的m值，从下一个人开始按顺时针方向重新报出一个数字，一直到所有人都出列为止。设计一个程序使用循环队列，按出列顺序打印每个数字。要求提交完整的可运行的源程序和运行结果截图。

#include <iostream>

#include "aqueue.h"

#include "lqueue.h"

#include "astack.h"

using namespace std;

struct person{

int password;

int origin\_id;

};

int reset(int m , int n)

{

return m % n ;

}

int main(){ //这里的aqueue采用的是原始版本

int n , m ; cin >> n >> m;//输入多少个人，以及初始密码

AQueue<person> a(n);

for(int i = 0 ; i < n ; i++)//输入每个人的密码，初始化

{

int tpass ; cin >> tpass ; person tperson;

tperson.password = tpass ; tperson.origin\_id = i + 1;

a.enqueue(tperson);

}

person now;

while(n != 1)

{

m = reset(m,n);//减少循环

if(m == 0) m += n;//避免m为0的情况

for(int i = 0 ; i < m-1 ; i++)//将非说出密码者出队再入队

{

person temp = a.frontValue();

a.dequeue();

a.enqueue(temp);

} //此时，front指向nowperson

now = a.frontValue() ; m = now.password ; cout<<now.origin\_id<<" ";//重置密码，输出id

a.dequeue();//删除

n--;

}

cout<<a.frontValue().origin\_id;//只剩最后一位，直接输出即可

}

