#include<iostream>

#include<assert.h>

using namespace std;

#define ERROR 0

template<class ElemType>

struct cirListNode {//组成链表的结点

ElemType data;

cirListNode<ElemType>\* next;

cirListNode() :next(NULL) {}

cirListNode(const ElemType& value) :data(value), next(NULL) {}

cirListNode(const ElemType& value, cirListNode<ElemType>\* ptr) :data(value), next(ptr) {}

//以下是两个修改操作:

void setListNode(cirListNode<ElemType>\* ptr) { next = ptr; }

void setListNode(const ElemType& value) { data = value; }

};

template<class ElemType>

class cirList {

private:

cirListNode<ElemType>\* head=NULL;//头结点:位于首结点的前面，其data无值

cirListNode<ElemType>\* tail=NULL;//尾指针，指向链表中的最后一个元素，当链表为空时其为next的值为NULL

//其data无值

public:

//构造函数：

cirList<ElemType>() {

head = new cirListNode<ElemType>;

tail = head;

tail->next = head;

}

cirList<ElemType>(const ElemType& e) {

head = new cirListNode<ElemType>;

head->next = tail = new cirListNode<ElemType>(e);

tail->next = head;//tail->next指针永远指向链表的头结点

}

//拷贝构造函数:

cirList(const cirList<ElemType>& L);

//拷贝复制运算符:

cirList<ElemType>& operator=(const cirList<ElemType>& L);

//析构函数://似乎没有问题\*标记要注意

~cirList<ElemType>() {

destroyList();

cout << "调用了析构函数" << endl;

}

//删除释放链表所占内存空间:

void destroyList();

//返回链表的实际长度

int getLength()const;

//取出链表的头结点:

cirListNode<ElemType>\* getHead() { return head; }

//取出链表的尾结点:

cirListNode<ElemType>\* getTail() { return tail; }

//链表清空函数:

void clear();

//检查链表是否为空表:

bool isEmpty() {

if (tail == head)return true;

else return false;

}

//在首结点之前插入一个结点:

bool insertFirst(const ElemType& e);

//设链表的头结点:

bool setHead(cirListNode<ElemType>\* p) {//目前感觉意义不大，因此没有做测试

p = new cirListNode<ElemType>;

head = p;

if (head == p) return true;

else return false;

}

//返回链表的第i个元素:

ElemType getElem(const int& i);

//在链表的pos位置前插入e:

bool insert\_prior(const int& pos, const ElemType& e);

//在链表的pos位置后插入e:

bool insert\_next(const int& pos, const ElemType& e);

//删除链表的首结点:

bool delete\_First();

//头插入的方法创建一个有头结点的单链表

bool createByHead(const int& n);

//尾部插入构造一个有头结点的链表

bool createByTail(const int& n);

//删除链表pos位置的结点:

bool delete\_cirList(const int& pos);

//compare函数:用来判断a和b是否相等

//bool compare(const ElemType& a, const ElemType& b);

//按序号查找，从链表的第一个结点开始，判断当前结点是否是第i个，

//若是，则返回该结点的数据域的值；否则继续后一个，直至表结束。没有第i个结点时返回空。

bool findElem(int k, ElemType& e);

//按值查找，即定位。从链表的第一个结点开始，判断当前结点值是否等于e。

//若是，则返回该结点的序号；否则继续后一个，直至表结束。找不到时返回0。

int searchElem(const ElemType& e) const;

//返回链表给定数据元素的前驱数据元素的值

bool priorElem(ElemType cur\_e, ElemType& pri\_e);

//返回链表给定数据元素的后继数据元素的值

bool nextElem(cirListNode<ElemType>\* p,const ElemType& e);

//返回链表给定数据元素的后继数据元素的值:

bool nextElem(ElemType cur\_e, ElemType& next\_e);

//逆置单链表:

void inversion();

//判断对应的指针所指的链表的结点在不在这个链表中:

//如果在，返回这个结点前驱,若不在或者该结点为头结点或是表尾节点的后一个结点，那么返回NULL

cirListNode<ElemType>\* prior\_pointer(cirListNode<ElemType>\* p);

//删除对应指针所指向的链表的结点:

void delete\_pointer(cirListNode<ElemType>\* p);

//纯化单链表:

void purge();

//链表遍历函数:

void traverse();

};

//删除释放链表函数:

template<class ElemType>

void cirList<ElemType>::destroyList() {

if (head==tail) {

delete head;

}

else {

cirListNode < ElemType>\* p = head->next;

cirListNode<ElemType>\* q = head;

while (p != head) {

q->next = p->next;

delete p;

p = q->next;

}

delete p;

//delete q;//注意：如果这里是单链表，那么该语句不能少，但是因为这是循环单链表，且p退出while循环时

//已经指向head，故仅有一个delete p就可以把这个链表彻底删除，再次释放同一块空间会引断点

}

}

//拷贝构造函数:意为着可以在定义一个链表的时候直接用另一个链表来初始化

template<class ElemType>

cirList<ElemType>::cirList(const cirList<ElemType>& L) {

head = new cirListNode<ElemType>;

head->next = head;

cirListNode<ElemType>\* q = head;//q用来指向新建的链表待插入的结点的前驱，同时也是现有链表的表尾

tail = head;

if (L.head->next) {

cirListNode<ElemType>\* p = L.head->next;//p专门用来遍历L

while (p != L.head) {

cirListNode<ElemType>\* k = new cirListNode<ElemType>;//k专门用来表示待插入的结点

k->data = p->data;

k->next = q->next;

q->next = k;

q = q->next;

p = p->next;

}

tail = q;

//tail->next = head;

}

cout << "调用了拷贝构造函数" << endl;//测试用句

}

//拷贝复制运算符:

template<class ElemType>

cirList<ElemType>& cirList<ElemType>::operator=(const cirList<ElemType>& L) {

head = new cirListNode<ElemType>;

tail = head;

head->next = head;

if (L.head->next) {

cirListNode<ElemType>\* p = L.head->next;//p用来遍历L

cirListNode<ElemType>\* q = head;//q用来指示现有的待被拷贝链表的最后一个结点，也是待插入的前一个结点

while (p != L.head) {

cirListNode<ElemType>\* k = new cirListNode<ElemType>;

k->data = p->data;

k->next = q->next;

q->next = k;

q = q->next;

p = p->next;

}

tail = q;

//tail->next = head;

}

cout << "使用了拷贝复制运算符" << endl;//测试用句

return \*this;

}

//返回链表的有数据结点个数:

template<class ElemType>

int cirList<ElemType>::getLength()const {

if (!head->next) { return 0; }

else {

cirListNode<ElemType>\* p = head->next;

int num = 0;

while (p != head) {

++num;

p = p->next;

}

return num;

}

}

//链表清空函数:

template<class ElemType>

void cirList<ElemType>::clear() {

if (head->next) {

cirListNode<ElemType>\* p = head->next, \* q = head;//p指向每一个要被删除的结点，q指向p的前驱

while (p != head) {

q->next = p->next;

delete p;

p = q->next;

}

tail = head;

//tail->next = head;

}

else {

cout << "这是一个空的链表" << endl;

}

}

//在首结点之前插入一个结点:

template<class ElemType>

bool cirList<ElemType>::insertFirst(const ElemType& e) {

if (!head->next) { return false; }

else {

cirListNode<ElemType>\* p = new cirListNode<ElemType>;

p->data = e;

p->next = head->next;

head->next = p;

return true;

}

}

//返回链表的第i个元素:

//注:如果输入的序号数不在链表的范围内，那么不返回任何值

template<class ElemType>

ElemType cirList<ElemType>::getElem(const int& i) {

if (i == 0 || tail == head) { return ERROR; }//此处退出函数使用了宏，目前不理解这个东西

//不过它既可以退出函数，又不会像exit那样终止结束程序

else {

int j = 1;

cirListNode<ElemType>\* p = head->next;

while (p != head && j < i) {

p = p->next;

++j;

}

if (j > i || p == head) { return ERROR; }

else {

return p->data;

}

}

}

//在链表的pos位置前插入e:

template<class ElemType>

bool cirList<ElemType>::insert\_prior(const int& pos, const ElemType& e) {

if (pos == 0 && tail == head) {

cirListNode<ElemType>\* p = new cirListNode<ElemType>;

p->data = e;

p->next = head->next;

head->next = p;

tail = p;

//tail->next = head;

return true;

}

else if (tail != head && pos != 0) {

cirListNode<ElemType>\* p = new cirListNode<ElemType>;

p->data = e;

cirListNode<ElemType>\* q = head->next, \* k = head;

int i = 1;

while (q != head && i < pos) {

k = q;

q = q->next;

++i;

}

if ((q == head && i < pos) || i > pos) {

return false;

}

else {

p->next = q;

k->next = p;

if (pos == this->getLength() + 1) {

tail = p;

//tail->next = head;

}

return true;

}

}

else { return false; }

}

//在链表的pos位置后插入e:

template<class ElemType>

bool cirList<ElemType>::insert\_next(const int& pos, const ElemType& e) {

if (tail == head) {

if (pos == 0) {

cirListNode<ElemType>\* p = new cirListNode<ElemType>;

p->data = e;

p->next = head->next;

head->next = p;

tail = p;

//tail->next = head;

return true;

}

else { return false; }

}

else {

cirListNode<ElemType>\* p = new cirListNode<ElemType>;

p->data = e;

cirListNode<ElemType>\* q = head->next, \* k = head;;

int i = 0;

while (q != head && i < pos) {

k = q;

q = q->next;

++i;

}

if ((q == head && i < pos) || i > pos) {

return false;

}

else {

p->next = k->next;

k->next = p;

if (pos == this->getLength()) {

tail = p;

//tail->next = head;

}

return true;

}

}

}

//删除链表的首结点:

template<class ElemType>

bool cirList<ElemType>::delete\_First() {

if (tail != head) {

cirListNode<ElemType>\* p = head->next;

if (p == tail) {

tail = head;

//tail->next = head;

}

head->next = head->next->next;

delete p;

return true;

}

else return false;

}

//头插入的方法创建一个有头结点的单链表

template<class ElemType>

bool cirList<ElemType>::createByHead(const int& n) {

ElemType e;

cirListNode<ElemType>\* q = head;

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

cin >> e;

cirListNode<ElemType>\* p = new cirListNode<ElemType>;

p->data = e;

p->next = q->next;

q->next = p;

if (i == 0) {

tail = p;

//tail->next = head;

}

}

if (tail != head) return true;

else return false;

}

//尾部插入构造一个有头结点的链表

template<class ElemType>

bool cirList<ElemType>::createByTail(const int& n) {

ElemType e;

cirListNode<ElemType>\* q = head;

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

cin >> e;

cirListNode<ElemType>\* p = new cirListNode<ElemType>;

p->data = e;

p->next = q->next;

q->next = p;

q = p;

if (i == n - 1) {

tail = p;

//tail->next = head;

}

}

if (tail != head) return true;

else return false;

}

//删除链表pos位置的结点:

template<class ElemType>

bool cirList<ElemType>::delete\_cirList(const int& pos) {

if (head == tail || pos == 0) { return false; }

else {

int i = 1;

cirListNode<ElemType>\* p = head->next, \* q = head;

while (p != head && i < pos) {

q = p;

p = p->next;

++i;

}

if ((p == head && i <= pos) || i > pos) {

return false;

}

else if (p != head) {

q->next = p->next;

if (q->next == head) {

tail = q;

//tail=tail->next;

}

delete p;

return true;

}

}

}

//compare函数:用来判断a和b是否相等

//此处比较的是链表中两个结点的data的值

/\*template<class ElemType>

bool cirList<ElemType>::compare(const ElemType& a, const ElemType& b) {

if (a == b) return true;

else return false;

}\*/

//按序号查找，从链表的第一个结点开始，判断当前结点是否是第i个，

//若是，则返回该结点的数据域的值；否则继续后一个，直至表结束。没有第i个结点时返回空。

template<class ElemType>

bool cirList<ElemType>::findElem(int k, ElemType& e) {//注意此程序中e有可能为随机值，因此使用时尽量给e

if (tail == head || k == 0) return false;//赋一个初始值

else {

int i = 1;

cirListNode<ElemType>\* p = head->next;

while (p && i < k) {

p = p->next;

++i;

}

if ((p == head && i <= k) || i > k) {

return false;

}

else {

e = p->data;

return true;

}

}

}

//按值查找，即定位。从链表的第一个结点开始，判断当前结点值是否等于e。

//若是，则返回该结点的序号；否则继续后一个，直至表结束。找不到时返回0。

template<class ElemType>

int cirList<ElemType>::searchElem(const ElemType& e) const {

if (tail == head) { return 0; }

else {

int i = 1;

cirListNode<ElemType>\* p = head->next;

while (p != head) {

if (p->data == e) {

return i;

}

else {

if (p->next == head) {

return 0;

}

p = p->next;

++i;

}

}

}

}

//返回链表给定数据元素的前驱数据元素的值

template<class ElemType>

bool cirList<ElemType>::priorElem(ElemType cur\_e, ElemType& pri\_e) {

if (tail == head) { return false; }

else {

cirListNode<ElemType>\* p = head->next, \* q = head;

while (p != head) {

if (p->data == cur\_e) {

if (q == head) return false;

pri\_e = q->data;

return true;

}

else {

if (p->next == head) {

return false;

}

q = p;

p = p->next;

}

}

}

}

//返回链表给定数据元素的后继数据元素的值

/\*template<class ElemType>//感觉莫名其妙的函数，没有进行测试

bool cirList<ElemType>::nextElem(cirListNode<ElemType>\* p,const ElemType& e) {

if (!head->next) { return false; }

else {

cirListNode<ElemType>\* q = head->next;

while (q->next) {

if (q->data == e) {

p->data = q->next->data;

return true;

}

else {

if (!q->next->next)return false;

q = q->next;

}

}

}

}\*/

//返回链表给定数据元素的后继数据元素的值:

template<class ElemType>

bool cirList<ElemType>::nextElem(ElemType cur\_e, ElemType& next\_e) {

if (tail == head) { return false; }//链表为空

else {

cirListNode<ElemType>\* p = head->next;

while (p != head) {

if (p->data == cur\_e) {

if (p->next != head) {

next\_e = p->next->data;

return true;

}

else return false;//给定的值为链表最后一个结点的值，因为最后一个结点的下一位指向

//head，取其值没有什么用，故返回false

}

p = p->next;

}

if (p == head) {//所传入的值不在链表中

return false;

}

}

}

//逆置循环链表:

template<class ElemType>

void cirList<ElemType>::inversion() {

if (tail == head) { return; }

else {

cirListNode<ElemType>\* p = head->next, \* q = p;//p为遍历原表L的指针，而q则记录原表L的首结点

//即q也是逆置后链表的尾结点

while (p != tail) {

head->next = p->next;//将p拿下来

p->next = tail->next;

tail->next = p;//将p插到tail后，同时tail不要动

p = head->next;

}//连续这样操作，head就会最终指向原表的尾结点，即逆置后的首结点

tail = q;//所有的操作后，切莫忘掉尾指针

tail->next = head;

}

}

//判断对应的指针所指的链表的结点在不在这个链表中:

//如果在，返回这个结点前驱,若不在或者该结点为头结点或是表尾节点的后一个结点，那么返回NULL

template<class ElemType>

cirListNode<ElemType>\* cirList<ElemType>::prior\_pointer(cirListNode<ElemType>\* p) {

if (tail == head) { return NULL; }//没有研究的必要

else {

cirListNode<ElemType>\* q = head;

while (q->next != head) {

if (q == p) {//注意在循环链表中，头结点的

return tail;

}

else if (q->next == p) return q;//q有可能是head，可以方便我们进行一些插入操作，但是不方便我们进行取值

else q = q->next;

}

return NULL;

}

}

//删除对应指针所指向的链表的结点:

template<class ElemType>

void cirList<ElemType>::delete\_pointer(cirListNode<ElemType>\* p) {

if (tail == head) { return; }

else {

cirListNode<ElemType>\* q = this->prior\_pointer(p);

if (q == tail) {

return;

}

else {

q->next = p->next;

if (p == tail) {

tail = q;

tail->next = head;

}

delete p;

}

}

}

//单链表的纯化:保留一个链表中先出现的元素的值，而删除其后与其相同的所有值

//无序有序表均可以纯化，且对于相同的值保留从前到后第一个出现的值

template<class ElemType>

void cirList<ElemType>::purge() {

if (this->isEmpty()) { return; }//没有纯化的必要

else {

cirListNode<ElemType>\* k = head->next;

while (k != head && k->next != head) {

cirListNode<ElemType>\* q = k, \* p = q->next;

while (p != head) {

if (p->data == k->data) {

delete\_pointer(p);

p = q->next;

}

else {

q = p;

p = p->next;

}

}

k = k->next;

}

}

}

//链表遍历函数:

template<class ElemType>

void cirList<ElemType>::traverse() {

if (tail != head) {

cirListNode<ElemType>\* p = head->next;

while (p != head) {

cout << p->data << ' ';

//if (p->next != NULL) cout << ' ';

p = p->next;

}

}

/\*else {

cout << "这是一个空链表" << endl;

}\*/

}

template<class ElemType>

void F(cirList<ElemType>& L, const int& order) {

cout << "L" << order << "的元素为:";

if (!L.isEmpty()) {

cout << endl;

L.traverse();

}

cout << endl;

cout<< "L"<<order<<"的元素个数为:"<<L.getLength()<< endl;

}