МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №7 по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Молчанов Владислав Дмитриевич, группа М80-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

### 

### Цель работы

Целью лабораторной работы является:

Закрепление навыков работы с шаблонами классов;

Построение итераторов для динамических структур данных.

### Задание

Используя структуру данных, разработанную для лабораторной работы №4, спроектировать и разработать **итератор** для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа for. Например:

for(auto i : stack) {

std::cout << \*i << std::endl;

}

Нельзя использовать:

Стандартные контейнеры std.

Программа должна позволять:

Вводить произвольное количество фигур и добавлять их в контейнер;

Распечатывать содержимое контейнера;

Удалять фигуры из контейнера.

**Дневник отладки**

Во время выполнения лабораторной работы были некие неисправности в итерировании по контейнеру в силу нелинейности бинарного дерева. В финальном варианте программы все работает исправно.

**Недочёты**  
Недочётов не было обнаружено.

**Выводы**

Лабораторная работа №7 позволила мне реализовать свой класс Iterator на языке С++, были освоены базовые навыки работы с самописными итераторами и итерирование по созданному контейнеру.

**Исходный код**

figure.h

#ifndef FIGURE\_H

#define FIGURE\_H

#include "point.h"

#include <memory>

class Figure{

private:

double area;

public:

virtual void Print() = 0;

virtual double GetArea() = 0;

virtual size\_t VertexNumber() = 0;

};

#endif

main.cpp  
  
#include <iostream>

#include "rhombus.h"

#include "tbinary\_tree.h"

#include <memory>

using namespace std;

int main(){

Rhombus r1(cin);

Rhombus r2(cin);

TBinaryTree<Rhombus> lol;

lol.Push(r1);

lol.Push(r2);

cout << lol;

lol.root = lol.Pop(lol.root, r1);

cout << lol.Count(0,100) << endl;

cout << lol;

system("pause");

return 0;

}

rhombus.cpp  
  
#include "rhombus.h"

using namespace std;

Rhombus::Rhombus(){

std::cout << "Empty constructor was called\n";

}

Rhombus::Rhombus(istream &is){

cout << "Enter all data: " << endl;

cin >> a;

cin >> b;

cin >> c;

cin >> d;

cout << "Rhombus created via istream" << endl;

}

void Rhombus::Print(){

cout << "Rhombus"<< a << " " << b << " " << c << " " << d << endl;

}

double Rhombus::GetArea(){

return abs(a.x \* b.y + b.x \*c.y + c.x\*d.y + d.x\*a.y - b.x\*a.y - c.x\*b.y - d.x\*c.y - a.x\*d.y)/2;

}

size\_t Rhombus::VertexNumber(){

size\_t h = 4;

return h;

}

bool operator == (Rhombus& r1, Rhombus& r2){

if((r1.a == r2.a) && (r1.b == r2.b) && (r1.c == r2.c) && (r1.d == r2.d)){

return true;

}

else{

return false;

}

}

ostream& operator << (ostream& os, Rhombus& r)

{

os << r.a << " " << r.b << " " << r.c << r.d << endl;

return os;

}

Rhombus::~Rhombus() {}

rhombus.h  
  
#ifndef RHOMBUS\_H

#define RHOMBUS\_H

#include "figure.h"

#include <iostream>

using namespace std;

class Rhombus : public Figure {

public:

Rhombus();

Rhombus(istream &is);

void Print();

double GetArea();

size\_t VertexNumber();

friend bool operator == (Rhombus& r1, Rhombus& r2);

friend ostream& operator << (ostream& os, Rhombus& p);

~Rhombus();

private:

Point a, b, c, d;

};

#endif

Point.cpp

#include "point.h"

#include <cmath>

#include <memory>

std::istream& operator>>(std::istream& is, Point& p) {

is >> p.x >> p.y;

return is;

}

std::ostream& operator<<(std::ostream& os, Point& p) {

os << "(" << p.x << ", " << p.y << ")";

return os;

}

Point::Point() : x(0.0), y(0.0) {}

Point::Point(double x, double y) : x(x), y(y) {}

Point::Point(std::istream &is) {

is >> x >> y;

}

double Point::dist(const Point& other){

double dx = other.x - x;

double dy = other.y - y;

return sqrt(dx \* dx + dy \* dy);

}

bool operator == (Point& p1, Point& p2){

return (p1.x == p2.x && p1.y == p2.y);

}

Point.h

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

class Point {

public:

friend std::istream& operator>>(std::istream& is, Point& p);

friend std::ostream& operator<<(std::ostream& os, Point& p);

Point();

Point(std::istream &is);

Point(double x, double y);

double dist(const Point& other);

friend bool operator == (Point& p1, Point& p2);

friend class Rhombus;

private:

double x;

double y;

};

#endif // POINT\_H

TBinaryTree.cpp

#include "tbinary\_tree.h"

#include <memory>

template <class T>

TBinaryTree<T>::TBinaryTree(){

this->root = nullptr;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> copy(std::shared\_ptr<TBinaryTreeItem<T>> root){

if(!root){

return nullptr;

}

std::shared\_ptr<TBinaryTreeItem<T>> cur(new TBinaryTreeItem<T>(root->GetRhombus()));

std::shared\_ptr<TBinaryTreeItem<T>> root\_copy = cur;

root\_copy->SetLeft(copy(root->GetLeft()));

root\_copy->SetRight(copy(root->GetRight()));

return root\_copy;

}

template <class T>

TBinaryTree<T>::TBinaryTree(const TBinaryTree &other) {

root = copy(other.root);

}

template <class T>

void rClear(std::shared\_ptr <TBinaryTreeItem<T>> cur){

if (cur != nullptr){

rClear(cur->GetLeft());

rClear(cur->GetRight());

cur = NULL;

}

}

template <class T>

void TBinaryTree<T>::Push(T romb){

if(root == nullptr){

std::shared\_ptr<TBinaryTreeItem<T>> cur(new TBinaryTreeItem<T>(romb));

root = cur;

}

else if(root->GetRhombus() == romb){

root->Increase();

}

else{

std::shared\_ptr<TBinaryTreeItem<T>> parent = root;

std::shared\_ptr<TBinaryTreeItem<T>> cur;

int checkleft = 1;

if(romb.GetArea() < parent->GetRhombus().GetArea()){

cur = root->GetLeft();

}

else{

cur = root->GetRight();

checkleft = 0;

}

while(cur != nullptr){

if(cur->GetRhombus() == romb){

cur->Increase();

}

else{

if(romb.GetArea() < cur->GetRhombus().GetArea()){

parent = cur;

cur = parent->GetLeft();

checkleft = 1;

}

else{

parent = cur;

cur = parent->GetRight();

checkleft = 0;

}

}

}

std::shared\_ptr<TBinaryTreeItem<T>> help(new TBinaryTreeItem<T>(romb));

cur = help;

if(checkleft == 1){

parent->SetLeft(cur);

}

else{

parent->SetRight(cur);

}

}

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> mini(std::shared\_ptr<TBinaryTreeItem<T>> root){

if (root->GetLeft() == NULL){

return root;

}

return mini(root->GetLeft());

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTree<T>::Pop(std::shared\_ptr<TBinaryTreeItem<T>> root, T &romb) {

if (root == NULL) {

return root;

}

else if (romb.GetArea() < root->GetRhombus().GetArea()) {

root->left = Pop(root->left, romb);

}

else if (romb.GetArea() > root->GetRhombus().GetArea()) {

root->right = Pop(root->right, romb);

}

else {

//first case of deleting - we are deleting a list

if (root->left == NULL && root->right == NULL) {

root = NULL;

return root;

}

//second case of deleting - we are deleting a verex with only one child

else if (root->left == NULL && root->right != NULL) {

std::shared\_ptr<TBinaryTreeItem<T>> pointer = root;

root = root->right;

return root;

}

else if (root->right == NULL && root->left != NULL) {

std::shared\_ptr<TBinaryTreeItem<T>> pointer = root;

root = root->left;

return root;

}

//third case of deleting

else {

std::shared\_ptr<TBinaryTreeItem<T>> pointer = mini(root->right);

root->Set(pointer->GetRhombus().GetArea());

root->right = Pop(root->right, pointer->GetRhombus());

}

}

}

template <class T>

void rCount(double minArea, double maxArea, std::shared\_ptr<TBinaryTreeItem<T>> curItem, int& ans){

if (curItem != nullptr){

rCount(minArea, maxArea, curItem->GetLeft(), ans);

rCount(minArea, maxArea, curItem->GetRight(), ans);

if (minArea <= curItem->GetRhombus().GetArea() && curItem->GetRhombus().GetArea() < maxArea){

ans += curItem->Cnt();

}

}

}

template <class T>

int TBinaryTree<T>::Count(double minArea, double maxArea){

int ans = 0;

rCount(minArea, maxArea, root, ans);

return ans;

}

template <class T>

bool TBinaryTree<T>::Empty(){

return root == nullptr;

}

template <class T>

void TBinaryTree<T>::Clear(){

rClear(root);

}

template <class T>

void Print (std::ostream& os, std::shared\_ptr<TBinaryTreeItem<T>> node){

if (!node){

return;

}

if( node->left){

os << node->GetRhombus().GetArea() << ": [";

Print (os, node->left);

if (node->right){

if (node->right){

os << ", ";

Print (os, node->right);

}

}

os << "]";

} else if (node->right) {

os << node->GetRhombus().GetArea() << ": [";

Print (os, node->right);

if (node->left){

if (node->left){

os << ", ";

Print (os, node->left);

}

}

os << "]";

}

else {

os << node->GetRhombus().GetArea();

}

}

template <class T>

std::ostream& operator<< (std::ostream& os, TBinaryTree<T>& tree){

Print(os, tree.root);

os << "\n";

return os;

}

template <class T>

T& TBinaryTree<T>::GetItemNotLess(double area, std::shared\_ptr<TBinaryTreeItem<T>> root) {

if (root->GetRhombus().GetArea() >= area) {

return root->GetRhombus();

}

else {

GetItemNotLess(area, root->right);

}

}

template <class T>

TBinaryTree<T>::~TBinaryTree() {

Clear();

}

template class TBinaryTree<Rhombus>;

template ostream& operator<<(ostream& os, TBinaryTree<Rhombus>& tree);

TBinaryTree.h

#ifndef TBINARYTREE\_H

#define TBINARYTREE\_H

#include "TBinaryTreeItem.h"

template <class T>

class TBinaryTree {

public:

TBinaryTree();

TBinaryTree(const TBinaryTree<T> &other);

void Push(T &pentagon);

std::shared\_ptr<TBinaryTreeItem<T>> Pop(std::shared\_ptr<TBinaryTreeItem<T>> root, T &pentagon);

T& GetItemNotLess(double area, std::shared\_ptr<TBinaryTreeItem<T>> root);

void Clear();

bool Empty();

int Count(double minArea, double maxArea);

template <class A>

friend std::ostream& operator<<(std::ostream& os, TBinaryTree<A>& tree);

virtual ~TBinaryTree();

std::shared\_ptr <TBinaryTreeItem<T>> root;

};

#endif

TBinaryTreeItem.cpp

#include "tbinary\_tree\_item.h"

#include <memory>

template <class T>

TBinaryTreeItem<T>::TBinaryTreeItem(const T& romb) {

this->rhombus = romb;

this->left = nullptr;

this->right = nullptr;

this->cnt = 1;

}

template <class T>

TBinaryTreeItem<T>::TBinaryTreeItem(const TBinaryTreeItem<T>& other) {

this->rhombus = other.rhombus;

this->left = other.left;

this->right = other.right;

this->cnt = other.cnt;

}

template <class T>

T& TBinaryTreeItem<T>::GetRhombus() {

return this->rhombus;

}

template <class T>

void TBinaryTreeItem<T>::SetRhombus(const T& romb){

this->rhombus = romb;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetLeft(){

return this->left;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetRight(){

return this->right;

}

template <class T>

void TBinaryTreeItem<T>::SetLeft(std::shared\_ptr<TBinaryTreeItem<T>> tBinTreeItem) {

if (this != nullptr){

this->left = tBinTreeItem;

}

}

template <class T>

void TBinaryTreeItem<T>::SetRight(std::shared\_ptr<TBinaryTreeItem<T>> tBinTreeItem) {

if (this != nullptr){

this->right = tBinTreeItem;

}

}

template <class T>

void TBinaryTreeItem<T>::Increase() {

if (this != nullptr){

++cnt;

}

}

template <class T>

void TBinaryTreeItem<T>::Decrease() {

if (this != nullptr){

cnt--;

}

}

template <class T>

int TBinaryTreeItem<T>::Cnt() {

return this->cnt;

}

template <class T>

void TBinaryTreeItem<T>::Set(int a){

cnt = a;

}

template <class T>

TBinaryTreeItem<T>::~TBinaryTreeItem() {

std::cout << "Destructor TBinaryTreeItem was called\n";

}

template <class T>

std::ostream &operator<<(std::ostream &os, TBinaryTreeItem<T> t)

{

os << t.rhombus << std::endl;

return os;

}

#include "rhombus.h"

template class TBinaryTreeItem<Rhombus>;

template std::ostream& operator<<(std::ostream& os, TBinaryTreeItem<Rhombus> t);

TBinaryTreeItem.h  
  
#ifndef LAB2\_TBINARY\_TREE\_ITEM\_H

#define LAB2\_TBINARY\_TREE\_ITEM\_H

#include "rhombus.h"

#include <memory>

template <class T>

class TBinaryTreeItem {

public:

TBinaryTreeItem(const T& romb);

TBinaryTreeItem(const TBinaryTreeItem<T>& other);

T& GetRhombus();

void SetRhombus(const T& romb);

std::shared\_ptr<TBinaryTreeItem<T>> GetLeft();

void SetLeft(std::shared\_ptr<TBinaryTreeItem<T>> tBinTreeItem);

std::shared\_ptr<TBinaryTreeItem<T>> GetRight();

void SetRight(std::shared\_ptr<TBinaryTreeItem<T>> tBinTreeItem);

void Increase();

void Decrease();

int Cnt();

void Set(int a);

virtual ~TBinaryTreeItem();

T rhombus;

std::shared\_ptr<TBinaryTreeItem<T>> left;

std::shared\_ptr<TBinaryTreeItem<T>> right;

int cnt;

template<class A>

friend std::ostream &operator<<(std::ostream &os, TBinaryTreeItem<A> t);

};

#endif //LAB2\_TBINARY\_TREE\_ITEM\_H

TIterator.h  
  
#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <iostream>

#include <memory>

template <class T, class A>

class TIterator {

public:

TIterator(std::shared\_ptr<T> iter) {

node\_ptr = iter;

}

A& operator\*() {

return node\_ptr->GetPentagon();

}

void GoToLeft() { //переход к левому поддереву, если существует

if (node\_ptr == NULL) {

std:: cout << "Root does not exist" << std:: endl;

}

else {

node\_ptr = node\_ptr->GetLeft();

}

}

void GoToRight() { //переход к правому поддереву, если существует

if (node\_ptr == NULL) {

std:: cout << "Root does not exist" << std:: endl;

}

else {

node\_ptr = node\_ptr->GetRight();

}

}

bool operator == (TIterator &iterator) {

return node\_ptr == iterator.node\_ptr;

}

bool operator != (TIterator &iterator) {

return !(\*this == iterator);

}

private:

std::shared\_ptr<T> node\_ptr;

};

#endif