МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №6 по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Михеева Кристина Олеговна, группа М8О-207Б-20*

Преподаватель *Дорохов Евгений Павлович*

**Условие:**

**Вариант 17:** Используя структуру данных, разработанную для лабораторной работы No4, 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.

**Описание программы:**

Исходный код лежит в 11 файлах:

1. main.cpp: основная программа, взаимодействие с пользователем посредством команд из меню.

2. figure.h: описание абстрактного класса фигуры.

3. point.h: описание класса точки.

4. point.cpp: реализация класса точки.

5. triangle.h: описание класса треугольника, наследующего от figure.

6. triangle.cpp: реализация класса треугольника, наследующего от figure. 7.TbinaryTree.cpp: реализация контейнера (бинарное дерево).

8.TBinaryTree.h: описание контейнера (бинарное дерево).

9. TbinaryTreeItem.cpp: реализация элемента бинарного дерева.

10.iTbinaryTreeItem.h: описание элемента бинарного дерева.

11. Titerator.h: описание итераторов.

**Дневник отладки:**

В данной лабораторной возникли проблемы с утечкой памяти, но при дальнейшем выполнении работы, все утечки были устранены.

**Вывод:**

В данной лабораторной мы снова поработали со шаблонами, а также познакомились с итераторами. Итераторы обеспечивают доступ к элементам контейнера. С помощью итераторов очень удобно перебирать элементы. Итератор описывается типом iterator. Это оказались очень полезными знаниями, которыми я буду пользоваться и в дальнейшей работе.

**Исходный код:**

**figure.h**

#ifndef FIGURE\_H

#define FIGURE\_H

#include <memory>

#include "point.h"

class Figure {

public:

virtual double Area() = 0;

virtual void Print(std::ostream &os) = 0;

virtual size\_t VertexesNumber() = 0;

virtual ~Figure() {};

};

#endif

**point.h**

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

class Point {

public:

Point();

Point(std::istream &is);

Point(double x, double y);

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

friend class Triangle;

double X();

double Y();

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

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

private:

double x;

double y;

};

#endif

**point.срр**

#include "point.h"

#include <cmath>

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::X() {

return x;

};

double Point::Y() {

return y;

};

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;

}

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

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

}

**triangle.h**

#ifndef TRIANGLE\_H

#define TRIANGLE\_H

#include "figure.h"

#include <iostream>

class Triangle : public Figure {

public:

Triangle(std::istream &InputStream);

Triangle();

double GetArea();

size\_t VertexesNumber();

double Area();

void Print(std::ostream &OutputStream);

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

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

virtual ~Triangle();

double area;

private:

Point a;

Point b;

Point c;

};

#endif

**triangle.cpp**

#include "triangle.h"

#include <cmath>

Triangle::Triangle() {}

Triangle::Triangle(std::istream &InputStream)

{

InputStream >> a;

InputStream >> b;

InputStream >> c;

std:: cout << "Triangle that you wanted to create has been created" << std:: endl;

}

void Triangle::Print(std::ostream &OutputStream) {

OutputStream << "Triangle: ";

OutputStream << a << " " << b << " " << c << std:: endl;

}

size\_t Triangle::VertexesNumber() {

size\_t number = 3;

return number;

}

double Triangle::Area() {

double s = 0.5 \* abs(a.getX() \* b.getY() + b.getX() \* c.getY() + c.getX() \* a.getY() - a.getY() \* b.getX() - b.getY() \* c.getX() - c.getY() \* a.getX());

this->area = s;

return s;

}

double Triangle:: GetArea() {

return area;

}

Triangle::~Triangle() {

std:: cout << "My friend, your triangle has been deleted" << std:: endl;

}

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

if(p1.a == p2.a && p1.b == p2.b && p1.c == p2.c) {

return true;

}

return false;

}

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

os << "Triangle: ";

os << p.a << p.b << p.c;

os << std::endl;

return os;

}

**TBinaryTree.h**

#ifndef TBINARYTREE\_H

#define TBINARYTREE\_H

#include "TBinaryTreeItem.h"

#include "TIterator.h"

template <class T>

class TBinaryTree {

public:

TBinaryTree();

TBinaryTree(const TBinaryTree<T> &other);

void Push(T &triangle);

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

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

**TBinaryTree.cpp**

#include "TBinaryTree.h"

template <class T>

TBinaryTree<T>::TBinaryTree () {

root = NULL;

}

template <class T>

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

if (!root) {

return NULL;

}

std::shared\_ptr<TBinaryTreeItem<T>> root\_copy(new TBinaryTreeItem<T>(root->GetTriangle()));

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

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

return root\_copy;

}

template <class T>

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

root = copy(other.root);

}

template <class T>

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

if (!node){

return;

}

if(node->GetLeft()){

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

Print (os, node->GetLeft());

if (node->GetRight()){

if (node->GetRight()){

os << ", ";

Print (os, node->GetRight());

}

}

os << "]";

} else if (node->GetRight()) {

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

Print (os, node->GetRight());

if (node->GetLeft()){

if (node->GetLeft()){

os << ", ";

Print (os, node->GetLeft());

}

}

os << "]";

}

else {

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

}

}

template <class T>

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

Print(os, tree.root);

os << "\n";

return os;

}

template <class T>

void TBinaryTree<T>::Push (T &triangle) {

if (root == NULL) {

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

root = help;

}

else if (root->GetTriangle() == triangle) {

root->IncreaseCounter();

}

else {

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

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

bool childInLeft = true;

if (triangle.GetArea() < parent->GetTriangle().GetArea()) {

current = root->GetLeft();

}

else if (triangle.GetArea() > parent->GetTriangle().GetArea()) {

current = root->GetRight();

childInLeft = false;

}

while (current != NULL) {

if (current->GetTriangle() == triangle) {

current->IncreaseCounter();

}

else {

if (triangle.GetArea() < current->GetTriangle().GetArea()) {

parent = current;

current = parent->GetLeft();

childInLeft = true;

}

else if (triangle.GetArea() > current->GetTriangle().GetArea()) {

parent = current;

current = parent->GetRight();

childInLeft = false;

}

}

}

std::shared\_ptr <TBinaryTreeItem<T>> item (new TBinaryTreeItem<T>(triangle));

current = item;

if (childInLeft == true) {

parent->SetLeft(current);

}

else {

parent->SetRight(current);

}

}

}

template <class T>

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

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

return root;

}

return FMRST(root->GetLeft());

}

template <class T>

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

if (root == NULL) {

return root;

}

else if (triangle.GetArea() < root->GetTriangle().GetArea()) {

root->SetLeft(Pop(root->GetLeft(), triangle));

}

else if (triangle.GetArea() > root->GetTriangle().GetArea()) {

root->SetRight(Pop(root->GetRight(), triangle));

}

else {

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

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

root = NULL;

return root;

}

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

else if (root->GetLeft() == NULL && root->GetRight() != NULL) {

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

root = root->GetRight();

return root;

}

else if (root->GetRight() == NULL && root->GetLeft() != NULL) {

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

root = root->GetLeft();

return root;

}

//third case of deleting

else {

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

root->GetTriangle().area = pointer->GetTriangle().GetArea();

root->SetRight(Pop(root->GetRight(), pointer->GetTriangle()));

}

}

return root;

}

template <class T>

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

if (current != NULL) {

RecursiveCount(minArea, maxArea, current->GetLeft(), ans);

RecursiveCount(minArea, maxArea, current->GetRight(), ans);

if (minArea <= current->GetTriangle().GetArea() && current->GetTriangle().GetArea() < maxArea) {

ans += current->ReturnCounter();

}

}

}

template <class T>

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

int ans = 0;

RecursiveCount(minArea, maxArea, root, ans);

return ans;

}

template <class T>

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

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

return root->GetTriangle();

}

else {

return GetItemNotLess(area, root->GetRight());

}

}

template <class T>

void RecursiveClear(std::shared\_ptr <TBinaryTreeItem<T>> current){

if (current!= NULL){

RecursiveClear(current->GetLeft());

RecursiveClear(current->GetRight());

current = NULL;

}

}

template <class T>

void TBinaryTree<T>::Clear(){

RecursiveClear(root);

root = NULL;

}

template <class T>

bool TBinaryTree<T>::Empty() {

if (root == NULL) {

return true;

}

return false;

}

template <class T>

TBinaryTree<T>::~TBinaryTree() {

Clear();

std:: cout << "Your tree has been deleted" << std:: endl;

}

#include "triangle.h"

template class TBinaryTree<Triangle>;

template std::ostream& operator<<(std::ostream& os, TBinaryTree<Triangle>& stack);

**TBinaryTreeItem.h**

#ifndef TBINARYTREE\_H

#define TBINARYTREE\_H

#include "TBinaryTreeItem.h"

#include "TIterator.h"

template <class T>

class TBinaryTree {

public:

TBinaryTree();

TBinaryTree(const TBinaryTree<T> &other);

void Push(T &triangle);

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

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 "TBinaryTreeItem.h"

template <class T>

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

this->triangle = triangle;

this->left = this->right = NULL;

this->counter = 1;

}

template <class T>

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

this->triangle = other.triangle;

this->left = other.left;

this->right = other.right;

this->counter = other.counter;

}

template <class T>

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

return this->triangle;

}

template <class T>

void TBinaryTreeItem<T>::SetTriangle(const T& triangle){

this->triangle = triangle;

}

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>> item) {

if (this != NULL){

this->left = item;

}

}

template <class T>

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

if (this != NULL){

this->right = item;

}

}

template <class T>

void TBinaryTreeItem<T>::IncreaseCounter() {

if (this != NULL){

counter++;

}

}

template <class T>

void TBinaryTreeItem<T>::DecreaseCounter() {

if (this != NULL){

counter--;

}

}

template <class T>

int TBinaryTreeItem<T>::ReturnCounter() {

return this->counter;

}

template <class T>

TBinaryTreeItem<T>::~TBinaryTreeItem() {

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

}

template <class T>

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

{

os << "Item: " << obj.GetTriangle() << std::endl;

return os;

}

#include "triangle.h"

template class TBinaryTreeItem<Triangle>;

template std::ostream& operator<<(std::ostream& os, TBinaryTreeItem<Triangle> &obj);

**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->GetTriangle();

}

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;

};

**main.cpp**

#include <iostream>

#include "triangle.h"

#include "TBinaryTree.h"

#include "TBinaryTreeItem.h"

int main () {

Triangle a (std::cin);

std:: cout << "The area of your figure is : " << a.Area() << std:: endl;

Triangle b (std::cin);

std:: cout << "The area of your figure is : " << b.Area() << std:: endl;

Triangle c (std::cin);

std:: cout << "The area of your figure is : " << c.Area() << std:: endl;

//lab2

TBinaryTree<Triangle> tree;

std:: cout << "Is tree empty? " << tree.Empty() << std:: endl;

tree.Push(a);

std:: cout << "And now, is tree empty? " << tree.Empty() << std:: endl;

tree.Push(b);

tree.Push(c);

std:: cout << "The number of figures with area in [minArea, maxArea] is: " << tree.Count(0, 100000) << std:: endl;

std:: cout << "The result of searching the same-figure-counter is: " << tree.root->ReturnCounter() << std:: endl;

std:: cout << "The result of function named GetItemNotLess is: " << tree.GetItemNotLess(0, tree.root) << std:: endl;

//lab5

TIterator<TBinaryTreeItem<Triangle, Triangle> iter(tree.root);

std:: cout << "The figure that you have put in root is: " << \*iter << std:: endl;

iter.GoToLeft();

std:: cout << "The first result of Left-Iter function is: " << \*iter << std:: endl;

iter.GoToRight();

std:: cout << "The first result of Right-Iter function is: " << \*iter << std:: endl;

TIterator<TBinaryTreeItem<Triangle>, Triangle> first(tree.root->GetLeft());

TIterator<TBinaryTreeItem<Triangle>, Triangle> second(tree.root->GetLeft());

if (first == second) {

std:: cout << "YES, YOUR ITERATORS ARE EQUALS" << std::endl;

}

TIterator<TBinaryTreeItem<Triangle>, Triangle third(tree.root->GetRight());

TIterator<TBinaryTreeItem<Triangle>, Triangle> fourth(tree.root->GetLeft());

if (third != fourth) {

std:: cout << "NO, YOUR ITERATORS ARE NOT EQUALS" << std::endl;

}

return 0;

}

3

4