МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №7**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: Недосекин Максим Александрович, группа М8О-207Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

**Задание:**

Используя структуры данных, разработанные для лабораторной работы №6, спроектировать и разработать итератор для динамической структуры данных.

Итератор должен быть разработан в виде шаблона, должен работать со всеми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа for, например:

for (auto i : list) {

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

}

**Вариант №18:**

* + Фигура: Трапеция (Trapezoid)
  + Контейнер: Бинарное дерево (Binary Tree)

**Описание программы:**

**Описание программы:**

Исходный код разделён на 11 файлов:

* figure.h – описание класса фигуры
* point.h – описание класса точки
* point.cpp – реализация класса точки
* trapezoid.h – описание класса пятиугольника
* trapezoid.cpp – реализация класса пятиугольника
* TBinaryTreeItem.h – описание элемента бинарного дерева
* TBinaryTreeItem.cpp – реализация элемента бинарного дерева
* TBinaryTree.h – описание бинарного дерева
* TBinaryTree.cpp – реализация бинарного дерева
* main.cpp – основная программа
* Iterator.h – реализация итератора по бинарному дереву

**Дневник отладки:**

Небольшие трудности возникли при реализации итератора. Дело в том, что бинарное дерево – нелинейная структура данных, в связи с чем пройтись по всем элементам не получится. С этим и были связаны проблемы. Однако сейчас все работает отлично.

**Вывод:**

Была написана программа с собственной реализацией контейнера в виде бинарного дерева с содержанием объектов фигур «по назначению», которая может вводить/выводить содержимое контейнера и удалить их при надобности, что дало мне познать преимущества итераторов для динамической структуры данных (в моем случае бинарное дерево), а именно обеспечивает доступ к содержимому объектов, без раскрытия их внутренней реализации, поддержка различных вариантов обхода и реализацию интерфейса для обхода других структур.

**Исходный код:**

**figure.h:**

#ifndef FIGURE\_H

#define FIGURE\_H

#include "point.h"

class Figure {

public:

virtual double Area() = 0;

virtual double GetArea() = 0;

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

virtual size\_t VertexesNumber() = 0;

virtual ~Figure() {};

};

#endif

**Main.cpp:**

#include <iostream>

#include "trapezoid.h"

#include "TBinaryTree.h"

#include "TBinaryTreeItem.h"

int main () {

//lab1

Trapezoid a (std::cin);

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

Trapezoid b (std::cin);

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

Trapezoid c (std::cin);

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

Trapezoid d (std::cin);

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

Trapezoid e (std::cin);

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

//lab2

TBinaryTree<Trapezoid> 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);

tree.Push(d);

tree.Push(e);

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<Trapezoid>, Trapezoid> 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<Trapezoid>, Trapezoid> first(tree.root->GetLeft());

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

if (first == second) {

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

}

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

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

if (third != fourth) {

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

}

return 0;

}

**Point.h:**

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

class Point {

public:

Point();

Point(std::istream &is);

Point(double x, double y);

double dist(Point &other);

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

friend class Pentagon;

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.cpp:**

#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::dist(Point& other) {

double dx = (other.x - x);

double dy = (other.y - y);

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

}

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);

}

**TBinaryTree.cpp:**

#include "TBinaryTree.h"

template <class T>

TBinaryTree<T>::TBinaryTree () {

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>> root\_copy(new TBinaryTreeItem<T>(root->GetTrapezoid()));

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->GetTrapezoid().GetArea() << ": [";

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

if (node->GetRight()){

if (node->GetRight()){

os << ", ";

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

}

}

os << "]";

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

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

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

if (node->GetLeft()){

if (node->GetLeft()){

os << ", ";

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

}

}

os << "]";

}

else {

os << node->GetTrapezoid().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 &trapezoid) {

if (root == NULL) {

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

root = help;

}

else if (root->GetTrapezoid() == trapezoid) {

root->IncreaseCounter();

}

else {

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

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

bool childInLeft = true;

if (trapezoid.GetArea() < parent->GetTrapezoid().GetArea()) {

current = root->GetLeft();

}

else if (trapezoid.GetArea() > parent->GetTrapezoid().GetArea()) {

current = root->GetRight();

childInLeft = false;

}

while (current != NULL) {

if (current->GetTrapezoid() == trapezoid) {

current->IncreaseCounter();

}

else {

if (trapezoid.GetArea() < current->GetTrapezoid().GetArea()) {

parent = current;

current = parent->GetLeft();

childInLeft = true;

}

else if (trapezoid.GetArea() > current->GetTrapezoid().GetArea()) {

parent = current;

current = parent->GetRight();

childInLeft = false;

}

}

}

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

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 &trapezoid) {

if (root == NULL) {

return root;

}

else if (trapezoid.GetArea() < root->GetTrapezoid().GetArea()) {

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

}

else if (trapezoid.GetArea() > root->GetTrapezoid().GetArea()) {

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

}

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->GetTrapezoid().area = pointer->GetTrapezoid().GetArea();

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

}

}

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->GetTrapezoid().GetArea() && current->GetTrapezoid().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->GetTrapezoid().GetArea() >= area) {

return root->GetTrapezoid();

}

else {

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

}

}

template <class T>

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

if (current!= nullptr){

RecursiveClear(current->GetLeft());

RecursiveClear(current->GetRight());

current = nullptr;

}

}

template <class T>

void TBinaryTree<T>::Clear(){

RecursiveClear(root);

root = nullptr;

}

template <class T>

bool TBinaryTree<T>::Empty() {

if (root == nullptr) {

return true;

}

return false;

}

template <class T>

TBinaryTree<T>::~TBinaryTree() {

Clear();

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

}

#include "trapezoid.h"

template class TBinaryTree<Trapezoid>;

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

**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 &trapezoid);

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

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 &trapezoid) {

this->trapezoid = trapezoid;

this->left = this->right = nullptr;

this->counter = 1;

}

template <class T>

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

this->trapezoid = other.trapezoid;

this->left = other.left;

this->right = other.right;

this->counter = other.counter;

}

template <class T>

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

return this->trapezoid;

}

template <class T>

void TBinaryTreeItem<T>::SetTrapezoid(const T& trapezoid){

this->trapezoid = trapezoid;

}

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 != nullptr){

this->left = item;

}

}

template <class T>

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

if (this != nullptr){

this->right = item;

}

}

template <class T>

void TBinaryTreeItem<T>::IncreaseCounter() {

if (this != nullptr){

counter++;

}

}

template <class T>

void TBinaryTreeItem<T>::DecreaseCounter() {

if (this != nullptr){

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.GetTrapezoid() << std::endl;

return os;

}

template class TBinaryTreeItem<Trapezoid>;

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

**TBinaryTreeItem.h:**

#ifndef TBINARYTREE\_ITEM\_H

#define TBINARYTREE\_ITEM\_H

#include <memory>

#include "trapezoid.h"

template <class T>

class TBinaryTreeItem {

public:

TBinaryTreeItem(const T& trapezoid);

TBinaryTreeItem(const TBinaryTreeItem<T>& other);

T& GetTrapezoid();

void SetTrapezoid(T& trapezoid);

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

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

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

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

void SetTrapezoid(const T& trapezoid);

void IncreaseCounter();

void DecreaseCounter();

int ReturnCounter();

virtual ~TBinaryTreeItem();

template<class A>

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

private:

T trapezoid;

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

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

int counter;

};

#endif

**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->GetTrapezoid();

}

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

**Trapezoid.cpp:**

#include "trapezoid.h"

#include <cmath>

Trapezoid::Trapezoid() {}

Trapezoid::Trapezoid(std::istream &InputStream)

{

InputStream >> a;

InputStream >> b;

InputStream >> c;

InputStream >> d;

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

}

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

OutputStream << "Trapezoid: ";

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

}

size\_t Trapezoid::VertexesNumber() {

size\_t number = 4;

return number;

}

double Trapezoid::Area() {

double k = (a.Y() - d.Y()) / (a.X() - d.X());

double m = a.Y() - k \* a.X();

double h = abs(b.Y() - k \* b.X() - m) / sqrt(1 + k \* k);

return 0.5 \* (a.dist(d) + b.dist(c)) \* h;

}

double Trapezoid:: GetArea() {

return area;

}

Trapezoid::~Trapezoid() {

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

}

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

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

return true;

}

return false;

}

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

os << "Trapezoid: ";

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

os << std::endl;

return os;

}

**Trapezoid.h:**

#ifndef TRAPEZOID\_H

#define TRAPEZOID\_H

#include "figure.h"

#include <iostream>

class Trapezoid : public Figure {

public:

Trapezoid(std::istream &InputStream);

Trapezoid();

double GetArea();

size\_t VertexesNumber();

double Area();

void Print(std::ostream &OutputStream);

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

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

virtual ~Trapezoid();

double area;

private:

Point a;

Point b;

Point c;

Point d;

};

#endif

**Результат работы:**

0 0 1 1 2 1 3 0

Trapezoid that you wanted to create has been created

The area of your figure is : 2

0 0 1 1 2 1 3 0

Trapezoid that you wanted to create has been created

The area of your figure is : 2

0 0 1 1 2 1 3 0

Trapezoid that you wanted to create has been created

The area of your figure is : 2

0 0 1 1 2 1 3 0

Trapezoid that you wanted to create has been created

The area of your figure is : 2

0 0 1 1 2 1 3 0

Trapezoid that you wanted to create has been created

The area of your figure is : 2

Is tree empty? 1

And now, is tree empty? 0

The number of figures with area in [minArea, maxArea] is: 5

The result of searching the same-figure-counter is: 5

The result of function named GetItemNotLess is: Trapezoid: (0, 0)(1, 1)(2, 1)(3, 0)

The figure that you have put in root is: Trapezoid: (0, 0)(1, 1)(2, 1)(3, 0)

The first result of Left-Iter function is: Trapezoid: (

Process finished with exit code -1073741819 (0xC0000005)