

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 Морозов Артем Борисович, 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 C++, 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"

class Figure {
public:
    virtual double Area() = 0;
    virtual void Print(std::ostream &os) = 0;
    virtual size_t VertexesNumber() = 0;
    virtual ~Figure() {};
```

```
};
```

```
#endif
```

main.cpp

```
#include <iostream>
#include "pentagon.h"
#include "TBinaryTree.h"
#include "TBinaryTreeItem.h"
int main () {
    //lab1
    Pentagon a (std::cin);
    std::cout << "The area of your figure is : " << a.Area() << std::endl;

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

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

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

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

    //lab2
    TBinaryTree<Pentagon> 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<Pentagon>, Pentagon> 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<Pentagon>, Pentagon> first(tree.root->GetLeft());
TIterator<TBinaryTreeItem<Pentagon>, Pentagon> second(tree.root->GetLeft());
if (first == second) {
    std::cout << "YES, YOUR ITERATORS ARE EQUALS" << std::endl;
}
TIterator<TBinaryTreeItem<Pentagon>, Pentagon> third(tree.root->GetRight());
TIterator<TBinaryTreeItem<Pentagon>, Pentagon> fourth(tree.root->GetLeft());
if (third != fourth) {
    std::cout << "NO, YOUR ITERATORS ARE NOT EQUALS" << std::endl;
}
return 0;
}

```

pentagon.cpp

```

#include "pentagon.h"
#include <cmath>

```

```

Pentagon::Pentagon() {}

```

```

Pentagon::Pentagon(std::istream &InputStream)
{
    InputStream >> a;
    InputStream >> b;
    InputStream >> c;
    InputStream >> d;
    InputStream >> e;
    std::cout << "Pentagon that you wanted to create has been created" << std::

```

```

endl;
}

void Pentagon::Print(std::ostream &OutputStream) {
    OutputStream << "Pentagon: ";
    OutputStream << a << " " << b << " " << c << " " << d << " " << e << std::
endl;

}

size_t Pentagon::VertexesNumber() {
    size_t number = 5;
    return number;
}

double Pentagon::Area() {
    double q = abs(a.X() * b.Y() + b.X() * c.Y() + c.X() * d.Y() + d.X() * e.Y() + e.X()
* a.Y() - b.X() * a.Y() - c.X() * b.Y() - d.X() * c.Y() - e.X() * d.Y() - a.X() * e.Y());
    double s = q / 2;
    this->area = s;
    return s;
}

double Pentagon:: GetArea() {
    return area;
}

Pentagon::~~Pentagon() {
    std:: cout << "My friend, your pentagon has been deleted" << std:: endl;
}

bool operator == (Pentagon& p1, Pentagon& p2){
    if(p1.a == p2.a && p1.b == p2.b && p1.c == p2.c && p1.d == p2.d && p1.e
== p2.e) {
        return true;
    }
    return false;
}

std::ostream& operator << (std::ostream& os, Pentagon& p){
    os << "Pentagon: ";
    os << p.a << p.b << p.c << p.d << p.e;
    os << std::endl;
}

```

```
        return os;
    }
}
```

Pentagon.h

```
#ifndef PENTAGON_H
#define PENTAGON_H

#include "figure.h"
#include <iostream>

class Pentagon : public Figure {
public:
    Pentagon(std::istream &InputStream);
    Pentagon();
    double GetArea();
    size_t VertexesNumber();
    double Area();
    void Print(std::ostream &OutputStream);
    friend bool operator == (Pentagon& p1, Pentagon& p2);
    friend std::ostream& operator << (std::ostream& os, Pentagon& p);
    virtual ~Pentagon();
    double area;

private:
    Point a;
    Point b;
    Point c;
    Point d;
    Point e;
};
#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::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);
}

```

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 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
```

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->GetPentagon()));
    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->GetPentagon().GetArea() << ": [";
        Print (os, node->GetLeft());
        if (node->GetRight()){
```

```

        if (node->GetRight()){
            os << ", ";
            Print (os, node->GetRight());
        }
    }
    os << "]";
} else if (node->GetRight()) {
    os << node->GetPentagon().GetArea() << ": [";
    Print (os, node->GetRight());
    if (node->GetLeft()){
        if (node->GetLeft()){
            os << ", ";
            Print (os, node->GetLeft());
        }
    }
    os << "]";
}
else {
    os << node->GetPentagon().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 &pentagon) {
    if (root == NULL) {
        std::shared_ptr<TBinaryTreeItem<T>> help(new TBinaryTreeItem<T>(pentagon));
        root = help;
    }
    else if (root->GetPentagon() == pentagon) {
        root->IncreaseCounter();
    }
    else {
        std::shared_ptr<TBinaryTreeItem<T>> parent = root;
        std::shared_ptr<TBinaryTreeItem<T>> current;
        bool childInLeft = true;
        if (pentagon.GetArea() < parent->GetPentagon().GetArea()) {
            current = root->GetLeft();
        }
        else if (pentagon.GetArea() > parent->GetPentagon().GetArea()) {

```

```

        current = root->GetRight();
        childInLeft = false;
    }
    while (current != NULL) {
        if (current->GetPentagon() == pentagon) {
            current->IncreaseCounter();
        }
        else {
            if (pentagon.GetArea() < current->GetPentagon().GetArea()) {
                parent = current;
                current = parent->GetLeft();
                childInLeft = true;
            }
            else if (pentagon.GetArea() > current->GetPentagon().GetArea()) {
                parent = current;
                current = parent->GetRight();
                childInLeft = false;
            }
        }
    }
}

std::shared_ptr <TBinaryTreeItem<T>> item (new TBinaryTreeItem<T>(pentagon));
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 &pentagon) {
    if (root == NULL) {
        return root;
    }
}

```

```

else if (pentagon.GetArea() < root->GetPentagon().GetArea()) {
    root->SetLeft(Pop(root->GetLeft(), pentagon));
}
else if (pentagon.GetArea() > root->GetPentagon().GetArea()) {
    root->SetRight(Pop(root->GetRight(), pentagon));
}
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 vertex 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->GetPentagon().area = pointer->GetPentagon().GetArea();
        root->SetRight(Pop(root->GetRight(), pointer->GetPentagon()));
    }
}
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->GetPentagon().GetArea() &&
current->GetPentagon().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->GetPentagon().GetArea() >= area) {
        return root->GetPentagon();
    }
    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 "pentagon.h"
template class TBinaryTree<Pentagon>;
template std::ostream& operator<<(std::ostream& os, TBinaryTree<Pentagon>& stack);
```

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 "TBinaryTreeItem.h"

template <class T>
```

```
TBinaryTreeltem<T>::TBinaryTreeltem(const T &pentagon) {
    this->pentagon = pentagon;
    this->left = this->right = NULL;
    this->counter = 1;
}
```

```
template <class T>
TBinaryTreeltem<T>::TBinaryTreeltem(const TBinaryTreeltem<T> &other) {
    this->pentagon = other.pentagon;
    this->left = other.left;
    this->right = other.right;
    this->counter = other.counter;
}
```

```
template <class T>
T& TBinaryTreeltem<T>::GetPentagon() {
    return this->pentagon;
}
```

```
template <class T>
void TBinaryTreeltem<T>::SetPentagon(const T& pentagon){
    this->pentagon = pentagon;
}
```

```
template <class T>
std::shared_ptr<TBinaryTreeltem<T>> TBinaryTreeltem<T>::GetLeft(){
    return this->left;
}
```

```
template <class T>
std::shared_ptr<TBinaryTreeltem<T>> TBinaryTreeltem<T>::GetRight(){
    return this->right;
}
```

```
template <class T>
void TBinaryTreeltem<T>::SetLeft(std::shared_ptr<TBinaryTreeltem<T>> item) {
    if (this != NULL){
        this->left = item;
    }
}
```

```
template <class T>
void TBinaryTreeltem<T>::SetRight(std::shared_ptr<TBinaryTreeltem<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.GetPentagon() << std::endl;
    return os;
}

#include "pentagon.h"
template class TBinaryTreeItem<Pentagon>;
template std::ostream& operator<<(std::ostream& os, TBinaryTreeItem<Pentagon> &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->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

```

TBinaryTreeItem.h

```

#ifndef TBINARYTREE_ITEM_H
#define TBINARYTREE_ITEM_H
#include "pentagon.h"

template <class T>
class TBinaryTreeItem {
public:
    TBinaryTreeItem(const T& pentagon);
    TBinaryTreeItem(const TBinaryTreeItem<T>& other);
    T& GetPentagon();
    void SetPentagon(T& pentagon);
    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 SetPentagon(const T& pentagon);
    void IncreaseCounter();
    void DecreaseCounter();
    int ReturnCounter();
    virtual ~TBinaryTreeItem();

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

private:
    T pentagon;
    std::shared_ptr<TBinaryTreeItem<T>> left;
    std::shared_ptr<TBinaryTreeItem<T>> right;
    int counter;
};
#endif

```