МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №8

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: Москвин Артём Артурович, группа М80-208Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

Задание:

Необходимо спроектировать и запрограммировать на языке C++ шаблон класса-контейнера первого уровня, содержащий одну фигуру (колонка фигура 1), согласно вариантам задания. Классы должны удовлетворять следующим правилам:

- Требования к классам фигуры аналогичны требованиям из лабораторной работы №1;
- Требования к классу контейнера аналогичны требованиям из лабораторной работы №2;
- Шаблон класса-контейнера должен содержать объекты используя std::shared_ptr<…>

Вариант №15:

- Фигура: Шестиугольник (Hexagon)
- Контейнер первого уровня: Бинарное дерево (TBinaryTree)

Описание программы:

Исходный код разделён на 13 файлов:

- figure.h – описание класса фигуры
- point.h – описание класса точки
- point.cpp – реализация класса точки
- hexagon.h – описание класса пятиугольника
- hexagon.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
- Titerator.h – реализация итератора по бинарному дереву
- TAllocationBlock.h – реализация аллокатора по заданию
- TAllocationBlock.cpp – описание аллоктаора по заданию

Дневник отладки: При выполнении работы ошибок выявлено не было.

Вывод: В ходе данной лабораторной работы я самостоятельно научился реализовывать аллокатор – очень важную вещь, если мы с вами говорим о низкоуровневых языках программирования типа C или C++. Я считаю, что каждый программист должен уметь реализовывать подобное, ведь работа с памятью – это база C++. Плюс собственноручной реализации аллокатора в том, что мы можем пользоваться другой логикой при выделении памяти, например, в структурах данных. Вполне

вероятно, что во многих случаях именно самостоятельно написанное выделение памяти сможет прийти на помощь программисту.

Исходный код:

point.h:

```
#ifndef LAB6_POINT_H
#define LAB6_POINT_H

#include <iostream>
#include <iomanip>
#include <cmath>
#include <memory>

class Point {

private:
    double x,y;
public:
    Point();
    Point(double x, double y);
    Point(std::istream& is);

    double dist(const Point& other);

    friend std::istream& operator >> (std::istream& is, Point& point);
    friend std::ostream& operator << (std::ostream& os, Point& point);
    friend bool operator == (const Point& p1, const Point& p2);

};

#endif
```

point.cpp:

```
#include "point.h"


std::ostream& operator << (std::ostream& os, Point& point) {
    os << "(" << std::setprecision(1) << point.x << ", " << point.y << ")";
    return os;
}
std::istream& operator >> (std::istream& is, Point& point){
    is >> point.x >> point.y;
    return is;
}

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 == (const Point& p1, const Point& p2){
    return (p1.x == p2.x && p1.y == p2.y);
}
```

figure.h:

```
#ifndef LAB6_FIGURE_H
#define LAB6_FIGURE_H

#include "point.h"

class Figure {
private:
    double area;
public :
    virtual double Area() = 0;
    virtual int VertexesNumber() = 0;

};

#endif
```

hexagon.h:

```
#ifndef LAB6_HEXAGON_H
#define LAB6_HEXAGON_H

#include "figure.h"
#include <vector>
#include <exception>

class Hexagon : public Figure{
private:
    std::vector<Point> points;
    double area;
public:
    Hexagon();
```

```
    Hexagon(std::vector<Point> points);
    friend std::ostream& operator << (std::ostream& os, Hexagon& oct);
    friend std::istream& operator >> (std::istream& is, Hexagon& oct);
    Hexagon& operator= (const Hexagon& copiedOct){
        for (int i = 0; i < 8; ++i){
            this->points[i] = copiedOct.points[i];
        }
        this->area = 0;
        for (int i = 1; i < 7; ++i){
            this->area += Hexagon::triangleArea(copiedOct.points[0], copiedOct.points[i],
copiedOct.points[i + 1]);
        }
        return *this;
    }
    friend bool operator == (const Hexagon& oct1, const Hexagon& oct2);
    double Area();
    double GetArea() const;
    double triangleArea(Point p1, Point p2, Point p3);
    int VertexesNumber();
    ~Hexagon();

    static const int VERTICES_NUM = 8;
};

#endif //LAB6_HEXAGON_H
```

hexagon.cpp:

```
#include "hexagon.h"

Hexagon::Hexagon(){
    const Point p(0.0, 0.0);
    this->points.assign(6, p);
    this->area = 0;
}

std::istream& operator >> (std::istream& is, Hexagon& oct){
    const Point p(0.0, 0.0);
    oct.points.assign(6, p);
    std::cout << "Enter the coordinates of hexagon: " << std::endl;
    for (int i = 0; i < 6; ++i){
        is >> oct.points[i];
    }
    oct.area = oct.Area();
    //std::cout << "Out of >>\n";
    return is;
}

Hexagon::Hexagon(std::vector<Point> points) : Hexagon(){
    for (int i = 0; i < 6; ++i){
        this->points[i] = points[i];
```

```
    }
    for (int i = 1; i < 5; ++i){
        this->area += Hexagon::triangleArea(points[0], points[i], points[i + 1]);
    }
}

std::ostream& operator << (std::ostream& os, Hexagon& oct){
    os << "Hexagon: ";
    for (int i = 0; i < 6; ++i){
        os << oct.points[i] << ' ';
    }
    return os;
}

bool operator == (const Hexagon& oct1, const Hexagon& oct2){
    for (int i = 0; i < Hexagon::VERTICES_NUM; ++i){
        if (!(oct1.points[i] == oct2.points[i])){
            return false;
        }
    }
    return true;
}

double Hexagon::triangleArea(Point p1, Point p2, Point p3) {
    double a = p1.dist(p2);
    double b = p2.dist(p3);
    double c = p1.dist(p3);
    double p = (a + b + c) / 2.0;
    double s = sqrt(p * (p - a) * (p - b) * (p - c));
    return s;
}

double Hexagon::Area() {
    double s = 0.0;
    for (int i = 1; i < 5; ++i){
        s += Hexagon::triangleArea(points[0], points[i], points[i + 1]);
    }
    return s;
}

int Hexagon::VertexesNumber() {
    return Hexagon::VERTICES_NUM;
}

double Hexagon::GetArea() const {
    return area;
}

Hexagon::~Hexagon() {}
```

TBinaryTreeItem.h:

```
#ifndef LAB6_TBINARY_TREE_ITEM_H
#define LAB6_TBINARY_TREE_ITEM_H

#include "hexagon.h" // checked
#include "TAllocationBlock.h"

template<class T> class TBinaryTreeItem {
private:
    std::shared_ptr<T> data;
    std::shared_ptr<TBinaryTreeItem<T>> left;
    std::shared_ptr<TBinaryTreeItem<T>> right;
    static TAllocationBlock stackitem_allocator;

public:
    TBinaryTreeItem<T>(const std::shared_ptr<T>& data);
    TBinaryTreeItem<T>(std::shared_ptr<TBinaryTreeItem<T>>& other);
    std::shared_ptr<T> GetData();
    void SetData(const std::shared_ptr<T>& data);
    std::shared_ptr<TBinaryTreeItem> GetLeft();
    void SetLeft(std::shared_ptr<TBinaryTreeItem<T>> tBinTreeItem);
    std::shared_ptr<TBinaryTreeItem> GetRight();
    void SetRight(std::shared_ptr<TBinaryTreeItem<T>> tBinTreeItem);
    template<class A> friend std::ostream& operator << (std::ostream& out,
std::shared_ptr<TBinaryTreeItem<A>> treeItem);
    virtual ~TBinaryTreeItem();

    void* operator new(size_t size);
    void operator delete(void* p);

    int counter;
};


#endif //LAB6_TBINARY_TREE_ITEM_H
```

TBinaryTreeItem.cpp:

```
#include "TbinaryTreeItem.h"
template<class T>
TBinaryTreeItem<T>::TBinaryTreeItem(const std::shared_ptr<T>& data){
    //std::cout << "In TBItem constructor\n";
    this->data = data;
    this->left = nullptr;
    this->right = nullptr;
    this->counter = 1;
}

template<class T>
TBinaryTreeItem<T>::TBinaryTreeItem(std::shared_ptr<TBinaryTreeItem<T>>& other) {
    this->data = other->data;
    this->left = other->left;
    this->right = other->right;
```

```
    this->counter = other->counter;
}

template <class T>
TAllocationBlock TBinaryTreeItem<T>::stackitem_allocator(sizeof(TBinaryTreeItem<T>), 100);

template<class T>
std::shared_ptr<T> TBinaryTreeItem<T>::GetData() {
    return this->data;
}

template<class T>
void TBinaryTreeItem<T>::SetData(const std::shared_ptr<T>& data) {
    this->data = data;
}

template<class T>
std::shared_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetLeft() {
    if (this != nullptr){
        return this->left;
    }
    else
        return nullptr;
}

template<class T>
void TBinaryTreeItem<T>::SetLeft(std::shared_ptr<TBinaryTreeItem<T>> tBinTreeItem) {
    if (this != nullptr){
        this->left = tBinTreeItem;
    }
}

template<class T>
std::shared_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetRight() {
    if (this != nullptr){
        return this->right;
    }
    else
        return nullptr;
}

template<class T>
void TBinaryTreeItem<T>::SetRight(std::shared_ptr<TBinaryTreeItem<T>> tBinTreeItem) {
    if (this != nullptr){
        this->right = tBinTreeItem;
    }
}

template<class T>
std::ostream& operator << (std::ostream& out, std::shared_ptr<TBinaryTreeItem<T>>
treeItem){
    if (treeItem != nullptr){
        out << treeItem->counter << '*' << std::setprecision(5) << treeItem->GetData()-
>GetArea();
```

```
    }
    else {
        out << "null";
    }
    return out;
}

template <class T>
void* TBinaryTreeItem<T>::operator new(size_t size) {
    return stackitem_allocator.allocate();
}

template <class T>
void TBinaryTreeItem<T>::operator delete(void* p) {
    stackitem_allocator.deallocate(p);
}

template<class T>
TBinaryTreeItem<T>::~TBinaryTreeItem<T>() {
    //std::cout << "Destructor TBinaryTreeItem was called\n";
}

template class TBinaryTreeItem<Hexagon>;
template std::ostream& operator <<(std::ostream& out, std::shared_ptr<TBinaryTreeItem<Hex-
agon>> treeItem);
```

TBinaryTree.h:

```
#ifndef LAB6_TBINARY_TREE_H
#define LAB6_TBINARY_TREE_H

#include "TBinaryTreeItem.h"
#include "Titerator.h"

template<class T> class TBinaryTree {
private:
    std::shared_ptr<TBinaryTreeItem<T>> root;
    std::shared_ptr<TBinaryTreeItem<T>> treeEnd;
public:
    TBinaryTree<T>();
    TBinaryTree(TBinaryTree<T>& otherBinTree);
    void Push(const std::shared_ptr<T>& data);
    void Pop(const std::shared_ptr<T>& data);
    void Clear();
    bool Empty();
    int Count(const std::shared_ptr<T>& data);
    std::shared_ptr<T> GetItemNotLess(double area);
    template<class A> friend std::ostream& operator << (std::ostream &out, TBinaryTree<A>*
tree);
    std::shared_ptr<TBinaryTreeItem<T>> GetRoot();

    TIterator<TBinaryTreeItem<T>, T> begin();
```

```
    TIterator<TBinaryTreeItem<T>, T> end();

    virtual ~TBinaryTree();

};


#endif //LAB6_TBINARY_TREE_H
```

TBinaryTree.cpp:

```
#include "TBinaryTree.h"
#include "string"

template<class T>
TBinaryTree<T>::TBinaryTree() {
    this->root = nullptr;
    std::shared_ptr<Hexagon> octEnd = std::make_shared<Hexagon>();
    //std::cout << "In TBTree: in constructor before NULL_OCT\n";
    //this->treeEnd = std::make_shared<TBinaryTreeItem<T>>(TBinaryTreeItem<T>(octEnd));
}

template<class T>
void recursiveCopying(std::shared_ptr<TBinaryTreeItem<T>> parItem, std::shared_ptr<TBi-
naryTreeItem<T>> curItem,
                      std::shared_ptr<TBinaryTreeItem<T>> otherItem, bool isLeftChild){
    if (otherItem != nullptr){
        curItem = std::make_shared<TBinaryTreeItem<T>>(TBinaryTreeItem<T>(otherItem));
        if (isLeftChild){
            parItem->SetLeft(curItem);
        }
        else{
            parItem->SetRight(curItem);
        }
        recursiveCopying(curItem, curItem->GetLeft(), otherItem->GetLeft(), true);
        recursiveCopying(curItem, curItem->GetRight(), otherItem->GetRight(), false);
    }
}

template<class T>
TBinaryTree<T>::TBinaryTree(TBinaryTree<T>& otherBinTree){
    this->root = std::make_shared<TBinaryTreeItem<T>>(TBinaryTreeItem<T>(otherB-
inTree.root));
    recursiveCopying(this->root, this->root->GetLeft(), otherBinTree.root->GetLeft(),
true);
    recursiveCopying(this->root, this->root->GetRight(), otherBinTree.root->GetRight(),
false);
}

template<class T>
void TBinaryTree<T>::Push(const std::shared_ptr<T>& data){
    bool needChangeEnd = true;
```

```
    if (this->root == nullptr){
        this->root = std::make_shared<TBinaryTreeItem<T>>(TBinaryTreeItem<T>(data));
        //root->SetRight(treeEnd);
    }
    else if (*this->root->GetData() == *data){
        ++this->root->counter;
    }
    else{
        std::shared_ptr<TBinaryTreeItem<T>> parent = this->root;
        std::shared_ptr<TBinaryTreeItem<T>> curItem;
        bool childInLeft = true;
        if (data->GetArea() < parent->GetData()->GetArea()) {
            curItem = this->root->GetLeft();
            needChangeEnd = false;
        }
        else {
            curItem = this->root->GetRight();
            childInLeft = false;
        }
        while (curItem != nullptr){ // while we are not in needed place in tree
            if (*curItem->GetData() == *data){ // if all points are same
                ++curItem->counter;
                return;
            }
            else { // compare with area
                if (data->GetArea() < curItem->GetData()->GetArea()){ // go to left child
                    parent = curItem;
                    curItem = parent->GetLeft();
                    childInLeft = true;
                    needChangeEnd = false;
                }
                else { // go to right child
                    parent = curItem;
                    curItem = parent->GetRight();
                    childInLeft = false;
                }
            }
        }
        curItem = std::make_shared<TBinaryTreeItem<T>>(TBinaryTreeItem<T>(data));
        //if (needChangeEnd)
            //curItem->SetRight(treeEnd);
        if (childInLeft){ // set the pointers
            parent->SetLeft(curItem);
        }
        else{
            parent->SetRight(curItem);
        }
    }
}

template<class T>
void TBinaryTree<T>::Pop(const std::shared_ptr<T>& data){
    std::shared_ptr<TBinaryTreeItem<T>> deletedItem = root;
    bool needChangeEnd = true;
```

```
    if (root != nullptr) { // if tree isn't empty
        std::shared_ptr<TBinaryTreeItem<T>> parentDelItem = root;
        bool isLeftLeaf = true; // will need this var in delete leaf
        while (deletedItem != nullptr && !(*deletedItem->GetData() == *data)) {
            if (deletedItem != root){
                parentDelItem = deletedItem;
            }
            if (data->GetArea() < deletedItem->GetData()->GetArea()) {
                deletedItem = deletedItem->GetLeft();
                isLeftLeaf = true;
                needChangeEnd = false;
            } else {
                deletedItem = deletedItem->GetRight();
                isLeftLeaf = false;
            }

        }
        if (deletedItem == nullptr) {
            throw std::invalid_argument("There isn't such hexagon in tree!");
        }
        if (deletedItem->counter > 1){
            --deletedItem->counter;
            return;
        }
        else {
            if (deletedItem->GetLeft() != nullptr){ // check left subtree
                std::shared_ptr<TBinaryTreeItem<T>> largestChild = deletedItem->GetLeft();
                if (largestChild->GetRight() != nullptr) { // if he isn't the largest
child himself
                    std::shared_ptr<TBinaryTreeItem<T>> parent = largestChild;
                    largestChild = parent->GetRight();
                    while (largestChild->GetRight() != nullptr) {
                        parent = largestChild;
                        largestChild = largestChild->GetRight();
                    }
                    // here we swap the values in deleted item and largest child and
change pointers to children
                    deletedItem->counter = largestChild->counter;
                    deletedItem->SetData(largestChild->GetData());
                    // in fact, we don't delete deletedItem, we delete the largest child
and put his values to deletedItem
                    parent->SetRight(largestChild->GetLeft());
                    //delete largestChild;
                    largestChild = nullptr;
                }
                else{ // if he is the largest child himself. Parent is unnecessary
                    deletedItem->counter = largestChild->counter;
                    //largestChild->counter = tmpCounter;
                    deletedItem->SetData(largestChild->GetData());
                    deletedItem->SetLeft(largestChild->GetLeft());
                    //delete largestChild;
                    largestChild = nullptr;
                }
            }
```

```
            else if (deletedItem->GetRight() != nullptr){ // check right subtree
                std::shared_ptr<TBinaryTreeItem<T>> leastChild = deletedItem->GetRight();
                if (leastChild->GetLeft() != nullptr) { // if he isn't the least child
himself
                    std::shared_ptr<TBinaryTreeItem<T>> parent = leastChild;
                    leastChild = parent->GetLeft();
                    while (leastChild->GetLeft() != nullptr) {
                        parent = leastChild;
                        leastChild = leastChild->GetLeft();
                    }
                    //if (needChangeEnd && deletedItem->GetRight() == leastChild){
                        //deletedItem->SetRight(treeEnd);
                    //}
                    // here we swap the values in deleted item and largest child and
change pointers to children
                    deletedItem->counter = leastChild->counter;
                    deletedItem->SetData(leastChild->GetData());
                    // in fact, we don't delete deletedItem, we delete the largest child
and put his values to deletedItem
                    parent->SetLeft(leastChild->GetRight());
                    //delete leastChild;
                    leastChild = nullptr;
                }
                else{ // if he is the least child himself
                    deletedItem->counter = leastChild->counter;
                    deletedItem->SetData(leastChild->GetData());
                    deletedItem->SetRight(leastChild->GetRight());
                    //delete leastChild;
                    leastChild = nullptr;
                }
            }
            else{ // if deleted item is a leaf
                if (deletedItem == root) {
                    root = nullptr;
                    //delete root;
                }
                else {
                    deletedItem = nullptr;
                    //delete deletedItem;
                    if (isLeftLeaf)
                        parentDelItem->SetLeft(nullptr);
                    else{
                        //if (needChangeEnd)
                        //    parentDelItem->SetRight(treeEnd);
                        //else
                            parentDelItem->SetRight(nullptr);
                    }

                    //deletedItem = nullptr;
                }
            }
        }
    }
    else {
```

```
        std::cout << "Tree is empty!\n";
    }
}

template<class T>
void recursiveCount(const std::shared_ptr<T>& data, std::shared_ptr<TBinaryTreeItem<T>>
curItem, int& ans){
    if (curItem != nullptr){
        recursiveCount(data, curItem->GetLeft(), ans);
        recursiveCount(data, curItem->GetRight(), ans);
        if (*curItem->GetData() == *data){
            ans += curItem->counter;
        }
    }
}

template<class T>
int TBinaryTree<T>::Count(const std::shared_ptr<T>& data){
    int ans = 0;
    recursiveCount(data, root, ans);
    return ans;
}

template<class T>
void recursiveClear(std::shared_ptr<TBinaryTreeItem<T>> curItem){
    if (curItem != nullptr){
        recursiveClear(curItem->GetLeft());
        recursiveClear(curItem->GetRight());
        //delete curItem;
        curItem = nullptr;
    }
}

template<class T>
void TBinaryTree<T>::Clear(){
    recursiveClear(root);
    //delete root;
    root = nullptr;
}

template<class T>
bool TBinaryTree<T>::Empty(){
    return root == nullptr;
}

template<class T>
std::shared_ptr<T> TBinaryTree<T>::GetItemNotLess(double area) {
    std::shared_ptr<TBinaryTreeItem<T>> curItem = root;
    while (curItem != nullptr){
        if (curItem->GetData()->GetArea() >= area){
            return curItem->GetData();
        }
        else {
            curItem = curItem->GetRight();
```

```
        }
    }
    throw std::out_of_range("Passed area is bigger then maximum in tree!");

}

template<class T>
void recursivePrint(std::ostream& out, std::shared_ptr<TBinaryTreeItem<T>> curItem){
    if (curItem != nullptr){
        out << curItem->counter << "*" << std::setprecision(5) << curItem->GetData()->Get-
Area();
        if (curItem->GetLeft() != nullptr || curItem->GetRight() != nullptr){
            out << ": [";
        }

        recursivePrint(out, curItem->GetLeft());

        if (curItem->GetLeft() != nullptr && curItem->GetRight() != nullptr){
            out << ", ";
        }
        recursivePrint(out, curItem->GetRight());
        if (curItem->GetLeft() != nullptr || curItem->GetRight() != nullptr)
            out << "]";
    }
}

template<class T>
std::ostream& operator << (std::ostream& out, TBinaryTree<T>* tree){
    if (tree == nullptr){
        out << "Tree is null" << std::endl;
    }
    else if (tree->root == nullptr){
        out << "Tree is empty\n";
    }
    else{
        recursivePrint(out, tree->root);
        out << std::endl;
    }
    return out;
}

template<class T>
std::shared_ptr<TBinaryTreeItem<T>> TBinaryTree<T>::GetRoot(){
    return root;
}

template<class T>
TIterator<TBinaryTreeItem<T>, T> TBinaryTree<T>::begin(){
    std::shared_ptr<TBinaryTreeItem<T>> beginItem = root;
    while(beginItem->GetLeft() != nullptr){
        beginItem = beginItem->GetLeft();
    }
    return TIterator<TBinaryTreeItem<T>, T>(beginItem);
}
```

```
template<class T>
TIterator<TBinaryTreeItem<T>, T> TBinaryTree<T>::end(){
    return TIterator<TBinaryTreeItem<T>, T>(treeEnd);
}

template<class T>
TBinaryTree<T>::~TBinaryTree<T>() {
    std::cout << "Destructor TBinaryTree was called\n";
    Clear();
}

template class TBinaryTree<Hexagon>;
template std::ostream& operator <<(std::ostream &out, TBinaryTree<Hexagon>* tree);
```

Titerator.h:

```
#ifndef LAB6_TITERATOR_H
#define LAB6_TITERATOR_H


#include <iostream>

template <class node, class T>
class TIterator {
public:
    TIterator(std::shared_ptr<node> n) {
        nodePtr = n;
    }

    std::shared_ptr<node> operator*() {
        return nodePtr;
    }

    std::shared_ptr<T> operator->() {
        return nodePtr->GetData();
    }

    bool operator==(TIterator const& i) {
        return nodePtr == i.nodePtr;
    }

    bool operator!=(TIterator const& i) {
        return !(*this == i);
    }

    void GoToLeft() {
        if (nodePtr == NULL) {
            std:: cout << "Node doesn't exist" << std:: endl;
        }
        else {
            nodePtr = nodePtr->GetLeft();
        }
```

```
    }
    void GoToRight() {
        if (nodePtr == NULL) {
            std:: cout << "Node doesn't exist" << std:: endl;
        }
        else {
            nodePtr = nodePtr->GetRight();
        }
    }

private:
    std::shared_ptr<node> nodePtr;
};

#endif //LAB5_TITERATOR_H
```

main.cpp:

```
#include <iostream>
#include "TBinaryTree.h"
int main() {
    int command;
    std::vector<std::shared_ptr<Hexagon>> addedHexagons;
    TBinaryTree<Hexagon>* tree = nullptr;
    int numOfItem = 0;
    while(true){
        std::cout << "----------------MENU----------------\n";
        std::cout << "0 : Exit the program" << "\n";
        std::cout << "1 : Add hexagon in tree\n";
        std::cout << "2 : Get first item with area not less than entered\n";
        std::cout << "3 : Get number of entries of hexagon by the queue number of your in-
put\n";
        std::cout << "4 : Get the first item with area not less entered and delete it\n";
        std::cout << "5 : Clear tree\n";
        std::cout << "6 : Create tree from another tree\n";
        std::cout << "7 : Print tree\n";
        std::cout << "8 : Create tree\n";
        std::cout << "9 : Delete tree\n";
        std::cout << "10 : Use iterators\n";
        std::cin >> command;

        switch (command) {
            case 0:{
                delete tree;
                return 0;
            }
            case 1:{
                std::cout << numOfItem + 1 << ".\n";
                std::shared_ptr<Hexagon> oct = std::make_shared<Hexagon>();
                std::cin >> *oct;
                std::cout << oct->GetArea() << std::endl;
                addedHexagons.push_back(oct);
                ++numOfItem;
```

```
                tree->Push(oct);
                break;
            }
            case 2:{
                double area;
                std::cout << "Enter the area:\n";
                std::cin >> area;
                try{
                    std::shared_ptr<Hexagon> oct = tree->GetItemNotLess(area);
                    std::cout << *oct << "(its area = " << std::setprecision(5) << oct-
>GetArea() << ")" << std::endl;
                }
                catch(std::exception& ex){
                    std::cout << ex.what() << std::endl;
                }
                break;
            }
            case 3:{
                int num;
                std::cout << "Enter the index number of entered items:\n";
                std::cin >> num;
                std::cout << *addedHexagons[num - 1] << "with area = " << std::setpreci-
sion(4) <<
                          addedHexagons[num - 1]->GetArea() << " meets " <<
                          tree->Count(addedHexagons[num - 1]) << " times in tree\n";
                break;
            }
            case 4:{
                double area;
                std::cout << "Enter the area:\n";
                std::cin >> area;
                try{
                    std::shared_ptr<Hexagon> deletedOct = tree->GetItemNotLess(area);
                    tree->Pop(deletedOct);
                }
                catch(std::exception& ex){
                    std::cout << ex.what() << std::endl;
                }
                break;
            }
            case 5:{
                tree->Clear();
                numOfItem = 0;
                break;
            }
            case 6:{
                TBinaryTree<Hexagon>* otherTree = new TBinaryTree<Hexagon>;
                std::cout << "Copied: " << otherTree;
                delete otherTree;
                break;
            }
            case 7:{
                std::cout << tree;
                break;
```

```
            }
            case 8:{
                tree = new TBinaryTree<Hexagon>();
                break;
            }
            case 9:{
                delete tree;
                tree = nullptr;
                numOfItem = 0;
                break;
            }
            case 10:{
                TIterator<TBinaryTreeItem<Hexagon>, Hexagon> iterator(tree->GetRoot());
                std:: cout << "Iterator points on hexagon: " << *iterator << std:: endl;
                iterator.GoToLeft();
                std:: cout << "Its left descendant: ";
                std:: cout << *iterator << std:: endl;
                iterator.GoToRight();
                std:: cout << "Its right descendant: " << *iterator << std:: endl;
                TIterator<TBinaryTreeItem<Hexagon>, Hexagon> iterA(tree->GetRoot()-
>GetLeft());
                TIterator<TBinaryTreeItem<Hexagon>, Hexagon> iterB(tree->GetRoot()-
>GetLeft());
                if (iterA == iterB) {
                    std:: cout << "Comparison of iterators is working: 1" << std::endl;
                }
                TIterator<TBinaryTreeItem<Hexagon>, Hexagon> iterC(tree->GetRoot()-
>GetRight());
                if (iterC != iterA) {
                    std:: cout << "Comparison of iterators is working: 2" << std::endl;
                }
                break;
            }
        }
    }
}
```

TAllocationBlock.h:

```
#ifndef LAB6_TALLOCATION_BLOCK_H
#define LAB6_TALLOCATION_BLOCK_H
#include <cstdlib>

class TAllocationBlock {
public:
    TAllocationBlock(size_t size, size_t count);
    void* allocate();
    void deallocate(void* pointer);
    bool has_free_blocks();

    virtual ~TAllocationBlock();

private:
    size_t _size;
```

```
    size_t _count;

    char* _used_blocks;
    void** _free_blocks;

    size_t _free_count;
};
#endif //LAB6_TALLOCATION_BLOCK_H
```

TAllocationBlock.cpp:

```
#include "TAllocationBlock.h"
#include <iostream>

TAllocationBlock::TAllocationBlock(size_t size, size_t count)
        : _size(size), _count(count) {
    _used_blocks = (char*) malloc(_size * _count);
    _free_blocks = (void**)malloc(sizeof(void*) * _count);

    for (size_t i = 0; i < _count; ++i) {
        _free_blocks[i] = _used_blocks + i * _size;
    }
    _free_count = _count;
    std::cout << "TAllocationBlock: Memory init" << std::endl;
}

void* TAllocationBlock::allocate() {
    void* result = nullptr;

    if (_free_count > 0) {
        result = _free_blocks[_free_count - 1];
        _free_count--;
        std::cout << "TAllocationBlock: Allocate " << (_count - _free_count);
        std::cout << " of " << _count << std::endl;

    } else {
        std::cout << "TAllocationBlock: No memory" << std::endl;
    }

    return result;
}

void TAllocationBlock::deallocate(void* pointer) {
    std::cout << "TAllocationBlock: Deallocate block " << std::endl;

    _free_blocks[_free_count] = pointer;
    _free_count++;
}

bool TAllocationBlock::has_free_blocks() {
    return _free_count > 0;
}

TAllocationBlock::~TAllocationBlock() {
```

```
    if (_free_count < _count) {
        std::cout << "TAllocationBlock: Memory leak?" << std::endl;
    } else {
        std::cout << "TAllocationBlock: Memory freed" << std::endl;
    }
    delete _free_blocks;
    delete _used_blocks;
}
```

Результат работы:

Такой же, как и в лабораторной работе №5.