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: Москвин Артём Артурович, группа М80-208Б-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 №15:

- Фигура: Шестиугольник (Hexagon)
- 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
- hexagon.h – описание класса пятиугольника
- pentagon.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: Данная лабораторная работа позволила мне собственноручно реализовать такую важную вещь как итераторы. Итераторы очень похожи на указатели. По сути, они выполняют тот же самый

функционал, только при этом еще и являются средством прохода по контейнеру. Они очень хороши в цикле range-based-for, когда нам нужно пройтись по всем элементам и, например, вывести их. Знания, полученные в ходе выполнения лабораторной работы, считаю очень полезными.

Исходный код:

```
#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 Hexagon;
  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::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;
}
```

```
#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
```

```
#ifndef HEXAGON_H
#define HEXAGON_H

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


class Hexagon : public Figure {
    public:
    Hexagon(std::istream &is);
    Hexagon();
    double GetArea();
    size_t VertexesNumber();
    double Area();
    void Print(std::ostream &os);
    friend bool operator == (Hexagon& p1, Hexagon& p2);
    friend std::ostream& operator << (std::ostream& os, Hexagon& p);
    virtual ~Hexagon();
    double area;
    private:
    Point a;
    Point b;
    Point c;
    Point d;
    Point e;
    Point f;
};
#endif
```

hexagon.cpp:

```
#include "hexagon.h"
#include <cmath>

    Hexagon::Hexagon() {}

    Hexagon::Hexagon(std::istream &is)
  {
      is >> a;
      is >> b;
      is >> c;
      is >> d;
      is >> e;
      is >> f;
      std:: cout << "Hexagon that you wanted to create has been created" << std:: endl;
  }

  void Hexagon::Print(std::ostream &os) {
      os << "Hexagon: ";
      os << a << " " << b << " " << c << " " << d << " " << e << f << std:: endl;

  }

   size_t Hexagon::VertexesNumber() {
       size_t number = 6;
       return number;
   }

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

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

    Hexagon::~Hexagon() {
          std:: cout << "Hexagon has been deleted" << std:: endl;
      }

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

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

TBinaryTreeItem.h:

```
#ifndef TBINARYTREE_ITEM_H
#define TBINARYTREE_ITEM_H
#include "hexagon.h"

template <class T>
class TBinaryTreeItem {
public:
TBinaryTreeItem(const T& hexagon);
TBinaryTreeItem(const TBinaryTreeItem<T>& other);
T& GetHexagon();
void SetHexagon(T& hexagon);
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 SetHexagon(const T& hexagon);
void IncreaseCounter();
void DecreaseCounter();
int ReturnCounter();
virtual ~TBinaryTreeItem();

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

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

TBinaryTreeItem.cpp:

```
#include "TBinaryTreeItem.h"

template <class T>
TBinaryTreeItem<T>::TBinaryTreeItem(const T &hexagon) {
    this->hexagon = hexagon;
    this->left = this->right = NULL;
    this->counter = 1;
```

```
}

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

template <class T>
T& TBinaryTreeItem<T>::GetHexagon() {
    return this->hexagon;
}

template <class T>
void TBinaryTreeItem<T>::SetHexagon(const T& hexagon){
    this->hexagon = hexagon;
}

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.GetHexagon() << std::endl;
    return os;
}

#include "hexagon.h"
template class TBinaryTreeItem<Hexagon>;
template std::ostream& operator<<(std::ostream& os, TBinaryTreeItem<Hexagon> &obj);
```

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 &hexagon);
std::shared_ptr<TBinaryTreeItem<T>> Pop(std::shared_ptr<TBinaryTreeItem<T>> root, T &hexa-
gon);
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->GetHexa-
gon()));
    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->GetHexagon().GetArea() << ": [";
        Print (os, node->GetLeft());
        if (node->GetRight()){
            if (node->GetRight()){
                os << ", ";
                Print (os, node->GetRight());
            }
        }
        os << "]";
    } else if (node->GetRight()) {
       os << node->GetHexagon().GetArea() << ": [";
        Print (os, node->GetRight());
        if (node->GetLeft()){
            if (node->GetLeft()){
                os << ", ";
                Print (os, node->GetLeft());
            }
        }
        os << "]";
    }
```

```
    else {
        os << node->GetHexagon().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 &hexagon) {
    if (root == NULL) {
    std::shared_ptr<TBinaryTreeItem<T>> help(new TBinaryTreeItem<T>(hexagon));
    root = help;
    }
    else if (root->GetHexagon() == hexagon) {
        root->IncreaseCounter();
    }
    else {
        std::shared_ptr <TBinaryTreeItem<T>> parent = root;
        std::shared_ptr <TBinaryTreeItem<T>> current;
        bool childInLeft = true;
        if (hexagon.GetArea() < parent->GetHexagon().GetArea()) {
            current = root->GetLeft();
        }
        else if (hexagon.GetArea() > parent->GetHexagon().GetArea()) {
            current = root->GetRight();
            childInLeft = false;
        }
        while (current != NULL) {
            if (current->GetHexagon() == hexagon) {
                current->IncreaseCounter();
            }
            else {
            if (hexagon.GetArea() < current->GetHexagon().GetArea()) {
                parent = current;
                current = parent->GetLeft();
                childInLeft = true;
                }
            else if (hexagon.GetArea() > current->GetHexagon().GetArea()) {
                parent = current;
                current = parent->GetRight();
                childInLeft = false;
            }
        }
    }
        std::shared_ptr <TBinaryTreeItem<T>> item (new TBinaryTreeItem<T>(hexagon));
        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 <TBi-
naryTreeItem<T>> root, T &hexagon) {
    if (root == NULL) {
        return root;
    }
    else if (hexagon.GetArea() < root->GetHexagon().GetArea()) {
        root->SetLeft(Pop(root->GetLeft(), hexagon));
    }
    else if (hexagon.GetArea() > root->GetHexagon().GetArea()) {
        root->SetRight(Pop(root->GetRight(), hexagon));
    }
    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->GetHexagon().area = pointer->GetHexagon().GetArea();
            root->SetRight(Pop(root->GetRight(), pointer->GetHexagon()));
        }
    }
    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->GetHexagon().GetArea() && current->GetHexagon().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->GetHexagon().GetArea() >= area) {
        return root->GetHexagon();
    }
    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 "hexagon.h"
template class TBinaryTree<Hexagon>;
template std::ostream& operator<<(std::ostream& os, TBinaryTree<Hexagon>& stack);
```

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->GetHexagon();
}

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
```

Результат работы:

```
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;
```

```
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: 1
The result of function named GetItemNotLess is: Pentagon: (3, 23)(23, 212)(32, 3)(23, 1)(1, 2)

The figure that you have put in root is: Pentagon: (3, 23)(23, 212)(32, 3)(23, 1)(1, 2)

The first result of Left-Iter function is: Pentagon: (3, 4)(3, 23)(2, 1)(2, 13)(3, 23)

The first result of Right-Iter function is: Pentagon: (2, 12)(12, 32)(3, 23)(12, 2)(3, 4)

YES, YOUR ITERATORS ARE EQUALS
NO, YOUR ITERATORS ARE NOT EQUALS
```