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

ЛАБОРАТОРНАЯ РАБОТА №5

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Рожков Илья Алексеевич, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

Задание:

Дополнить класс-контейнер из лабораторной работы №2

Дневник отладки:

Были проблемы с созданием умного указателя, так как он не принимает конструктор по умолчанию.

Вывод:

При выполнении работы я на практике освоил основы работы с умными указателями. Они позволяют избежать проблем с утечками памяти, с разыменовыванием нулевого указателя, обращением к неициализированной области памяти, а также с удалением уже удалённого объекта.

Исходный код:

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(Lab1)

set(CMAKE_CXX_STANDARD 23)

add_executable(Lab1 main.cpp figure.h rhombus.cpp rhombus.h pentagon.cpp
pentagon.h GeronFormula.h GeronFormula.cpp hexagon.cpp hexagon.h Node.cpp
Node.h tbinarytree.cpp tbinarytree.h)
```

Figure.h

```
//
// Created by Илья Рожков on 12.09.2021.
//

#ifndef LAB1_FIGURE_H
#define LAB1_FIGURE_H
#include "iostream"
#include <utility>
#include <math.h>
#include <cmath>
```

```
class Figure {

public:
    virtual void Print(std::ostream& os) const = 0;
    virtual size_t VertexesNumber() const = 0;
    virtual double Area() const = 0;
    //virtual ~Figure() = 0;


};


#endif //LAB1_FIGURE_H
```

GeronFormula.cpp

```
//
// Created by Илья Рожков on 16.09.2021.
//
#include "GeronFormula.h"
#include<cmath>


double GeronFormula(double a, double b, double c) {
    double p, s;
    p = (a + b + c) / 2;
    s = sqrt(p * (p - a) * (p - b) * (p - c));
    return s;
}

double getDistance(const std::pair<double, double> &x, const
std::pair<double, double> &y) {
    return sqrt(pow((x.first - y.first), 2) + pow((x.second -
y.second), 2));
}

double GeronFormulaFromCordinates(const Cordinate &a, const
Cordinate &b, const Cordinate &c) {
    double x = getDistance(a, b);
    double y = getDistance(b, c);
    double z = getDistance(c, a);
    return GeronFormula(x, y, z);
}

double AreaOfMultigone(const std::vector<Cordinate>
&cordinates) {
    double s = 0;
    for (int i = 0; i < cordinates.size(); i += 3)
        s += GeronFormulaFromCordinates(cordinates[i],
cordinates[(i + 1) % cordinates.size()], cordinates[(i + 2) %
cordinates.size()]);
    return s;
}
```

hexagon.cpp

```cpp
//
// Created by Илья Рожков on 16.09.2021.
//

#include "hexagon.h"

Hexagon::Hexagon() {
    for (int i = 0; i < 6; i++) {
        Cordinate elemt = std::make_pair(0, 0);
        _cordinates.push_back(elemt);
    }

}

Hexagon::Hexagon(const std::vector<Cordinate> &cordinates) :
_cordinates(cordinates) {
    if (_cordinates.size() != 6) {
        throw "wrong size";
    }

}

size_t Hexagon::VertexesNumber() const {
    return 6;
}

double Hexagon::Area() const {
    return AreaOfMultigone(_cordinates);
}

void Hexagon::Print(std::ostream& os) const {
    os << "Hexagon: ";
    for (int i = 0; i < _cordinates.size(); i++)
        os << '(' << _cordinates[i].first << ", " <<
_cordinates[i].second << ") ";
    os << '\n';
    //return os;

}

std::ostream &operator<<(std::ostream &os, const Hexagon &r) {
    os << "Hexagon: ";
    for (int i = 0; i < r._cordinates.size(); i++)
        os << '(' << r._cordinates[i].first << ", " <<
r._cordinates[i].second << ") ";
    os << '\n';
    return os;
}

std::istream &operator>>(std::istream &in, Hexagon &r) {
    for (int i = 0; i < 6; i++)
        in >> r._cordinates[i].first >>
r._cordinates[i].second;
    return in;
}



Hexagon::Hexagon(std::istream &in) {
    for (int i = 0; i < 6; i++) {
        Cordinate elemt = std::make_pair(0, 0);
        _cordinates.push_back(elemt);
    }
    for (int i = 0; i < 6; i++)
        in >> _cordinates[i].first >> _cordinates[i].second;
    //return in;

}

Hexagon &Hexagon::operator=(const Hexagon &h) {
    if (&h == this)
        return *this;
    _cordinates = h._cordinates;
    return *this;
}

bool Hexagon::operator==(const Hexagon &h) const {
    return _cordinates == h._cordinates;
}

Hexagon::~Hexagon() {

}
```

pentagon.cpp

```
//
// Created by Илья Рожков on 15.09.2021.
//

#include "pentagon.h"
#include <string.h>
#include "GeronFormula.h"



/*
{
    double p, s;
    p = (a + b + c) / 2;
    s = sqrt(p * (p - a) * (p - b) * (p - c));
    return s;
}

double getDistance(const std::pair<double, double>& x , const
std::pair<double, double>& y)
```

```
{
    return sqrt(pow((x.first - y.first), 2) + pow((x.second -
y.second), 2));
}

double GeronFormulaFromCordinates(Cordinate a,Cordinate
b,Cordinate c)
{
    double x = getDistance(a, b);
    double y = getDistance(b, c);
    double z = getDistance(c, a);
    return GeronFormula(x, y, z);
}
*/

Pentagon::Pentagon() {
    for (int i = 0; i < 5; i++) {
        Cordinate elemt = std::make_pair(0,0);
        _cordinates.push_back(elemt);
        //_cordinates[i].first = 0;
        //_cordinates[i].second = 0;
    }

}



size_t Pentagon::VertexesNumber() const {
    return 5;
}

Pentagon::Pentagon(const std::vector<Cordinate> &cordinates) :
_cordinates(cordinates){
    if (_cordinates.size() != 5)
        throw std::out_of_range("wrong number of cordinates");

}

double Pentagon::Area() const {
    return AreaOfMultigone(_cordinates);

}

std::ostream &operator<<(std::ostream &os, const Pentagon &r) {
    os << "Pentagon: ";
    for (int i = 0; i < r._cordinates.size(); i++)
        os << '(' << r._cordinates[i].first << ", " <<
r._cordinates[i].second << ") ";
    os << '\n';
    return os;
}

std::istream &operator>>(std::istream &in, Pentagon &r) {
    for (int i = 0; i < 5; i++)
```

```
        in >> r._cordinates[i].first >> 
r._cordinates[i].second;
    return in;
}

void Pentagon::Print(std::ostream& os) const {
    os << "Pentagon: ";
    for (int i = 0; i < _cordinates.size(); i++)
        os << '(' << _cordinates[i].first << ", " << 
_cordinates[i].second << ") ";
    os << '\n';

}

Pentagon::Pentagon(std::istream &in) {
    for (int i = 0; i < 5; i++) {
        Cordinate elemt = std::make_pair(0,0);
        _cordinates.push_back(elemt);
        //_cordinates[i].first = 0;
        //_cordinates[i].second = 0;
    }
    for (int i = 0; i < 5; i++)
        in >> _cordinates[i].first >> _cordinates[i].second;
}

Pentagon &Pentagon::operator=(const Pentagon &p) {
    if(&p == this)
        return *this;
    _cordinates = p._cordinates;
    return *this;
}

bool Pentagon::operator==(const Pentagon &p) const {
    return _cordinates == p._cordinates;
}

Pentagon::~Pentagon() {

}
```

rhombus.cpp

```
//
// Created by Илья Рожков on 12.09.2021.
//

#include "rhombus.h"
#include <string.h>
#include "GeronFormula.h"


using std::pair;
typedef pair<double, double> Cordinate;
```

```cpp
/*double getDistance(const pair<double, double>& x , const 
pair<double, double>& y)
{
    return sqrt(pow((x.first - y.first), 2) + pow((x.second - 
y.second), 2));
}*/



Rhombus::Rhombus() {

}

Rhombus::~Rhombus() {

}

double Rhombus::Area() const {
    return 0.5 * getDistance(_x1, _x3) * getDistance(_x2, _x4);
}

Rhombus::Rhombus(Cordinate &x1, Cordinate &x2, Cordinate &x3, 
Cordinate &x4) : _x1(x1), _x2(x2), _x3(x3), _x4(x4){
    if(!IsRhombus())
        throw  "not correct input";
}

size_t Rhombus::VertexesNumber() const {
    return 4;
}

bool Rhombus::IsRhombus() const {
    if (getDistance(_x1, _x2) == getDistance(_x2, _x3) && 
getDistance(_x2, _x3) == getDistance(_x3, _x4) &&
    getDistance(_x3, _x4) == getDistance(_x4, _x1) && 
getDistance(_x4, _x1) == getDistance(_x1, _x2))
        return true;
    return false;
}



void Rhombus::Print(std::ostream& os) const {
    os << "Rhombus: (" << _x1.first << ", " << _x1.second << ") 
" << '(' << _x2.first << ' ' << _x2.second << ") "
    << '(' << _x3.first << ' ' << _x3.second << ") " << '(' << 
_x4.first << ' ' << _x4.second << ")" << std::endl;

}

std::ostream& operator<<(std::ostream &os, const Rhombus& r)
{
    os << "Rhombus: (" << r._x1.first << ", " << r._x1.second
<< ") " << '(' << r._x2.first << ' ' << r._x2.second << ") "
       << '(' << r._x3.first << ' ' << r._x3.second << ") " <<
'(' << r._x4.first << ' ' << r._x4.second << ")" << std::endl;
    return os;
}

std::istream &operator>>(std::istream &in, Rhombus &r) {
    in >> r._x1.first >> r._x1.second >> r._x2.first >>
r._x2.second >> r._x3.first >> r._x3.second >> r._x4.first >>
r._x4.second;
    if(!r.IsRhombus())
        throw  "not correct input";
    return in;
}

Rhombus::Rhombus(const Rhombus &r) : _x1(r._x1), _x2(r._x2),
_x3(r._x3), _x4(r._x4) {

}

Rhombus::Rhombus(std::istream &in) {
    in >> _x1.first >> _x1.second >> _x2.first >> _x2.second >>
_x3.first >> _x3.second >> _x4.first >> _x4.second;
}

Rhombus &Rhombus::operator=(const Rhombus &r) {
    if (&r == this)
        return *this;
    _x1 = r._x1;
    _x2 = r._x2;
    _x3 = r._x3;
    _x4 = r._x4;
    return *this;

}

bool Rhombus::operator==(const Rhombus &r) const {
    return _x1 == r._x1 && _x2 == r._x2 && _x3 == r._x3 && _x4
== r._x4;
}
```

tbinarytree.cpp

```
//
// Created by Илья Рожков on 30.09.2021.
//

#include "tbinarytree.h"
#include "stdexcept"
```

```cpp
TBinaryTree::TBinaryTree() {
    t_root = nullptr;
}

void TBinaryTree::Push(const Pentagon& octagon) {
    TreeElem* curr = t_root;

    if (curr == nullptr)
        t_root = new TreeElem(octagon);

    while (curr)
    {
        if (curr->get_octagon() == octagon)
        {
            curr->set_count_fig(curr->get_count_fig() + 1);
            return;
        }
        if (octagon.Area() < curr->get_octagon().Area())
            if (curr->get_left() == nullptr)
            {
                curr->set_left(new TreeElem(octagon));
                return;
            }
        if (octagon.Area() >= curr->get_octagon().Area())
            if (curr->get_right() == nullptr && !(curr-
>get_octagon() == octagon))
            {
                curr->set_right(new TreeElem(octagon));
                return;
            }
        if (curr->get_octagon().Area() > octagon.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }
}

const Pentagon& TBinaryTree::GetItemNotLess(double area) {
    TreeElem* curr = t_root;

    while (curr)
    {
        if (area == curr->get_octagon().Area())
            return curr->get_octagon();
        if (area < curr->get_octagon().Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (area >= curr->get_octagon().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    throw std::out_of_range("out_of_range");

}

size_t TBinaryTree::Count(const Pentagon& octagon) {
    size_t count = 0;
    TreeElem* curr = t_root;

    while (curr)
    {
        if (curr->get_octagon() == octagon)
            count = curr->get_count_fig();
        if (octagon.Area() < curr->get_octagon().Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (octagon.Area() >= curr->get_octagon().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    return count;
}

void Pop_List(TreeElem* curr, TreeElem* parent);
void Pop_Part_of_Branch(TreeElem* curr, TreeElem* parent);
void Pop_Root_of_Subtree(TreeElem* curr, TreeElem* parent);
void TBinaryTree::Pop(const Pentagon& octagon) {

    TreeElem* curr = t_root;
    TreeElem* parent = nullptr;

    while (curr && curr->get_octagon() != octagon)
    {
        parent = curr;
        if (curr->get_octagon().Area() > octagon.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }

    if (curr == nullptr)
        return;

    curr->set_count_fig(curr->get_count_fig() - 1);

    if(curr->get_count_fig() <= 0)
    {
        if (curr->get_left() == nullptr && curr->get_right() ==
nullptr)
        {
            Pop_List(curr, parent);
            return;
        }
        if (curr->get_left() == nullptr || curr->get_right() == 
nullptr)
        {
            Pop_Part_of_Branch(curr, parent);
            return;
        }
        if (curr->get_left() != nullptr && curr->get_right() != 
nullptr)
        {
            Pop_Root_of_Subtree(curr, parent);
            return;
        }
    }
}

void Pop_List(TreeElem* curr, TreeElem* parent) {
    if (parent->get_left() == curr)
        parent->set_left(nullptr);
    else
        parent->set_right(nullptr);
    delete(curr);
}

void Pop_Part_of_Branch(TreeElem* curr, TreeElem* parent) {
    if (parent) {
        if (curr->get_left()) {
            if (parent->get_left() == curr)
                parent->set_left(curr->get_left());

            if (parent->get_right() == curr)
                parent->set_right(curr->get_left());

            curr->set_right(nullptr);
            curr->set_left(nullptr);
            delete(curr);
            return;
        }

        if (curr->get_left() == nullptr) {
            if (parent && parent->get_left() == curr)
                parent->set_left(curr->get_right());

            if (parent && parent->get_right() == curr)
                parent->set_right(curr->get_right());

            curr->set_right(nullptr);
            curr->set_left(nullptr);
            delete(curr);
            return;
        }
    }
```

```
}

void Pop_Root_of_Subtree(TreeElem* curr, TreeElem* parent) {
    TreeElem* replace = curr->get_left();
    TreeElem* rep_parent = curr;
    while (replace->get_right())
    {
        rep_parent = replace;
        replace = replace->get_right();
    }

    curr->set_octagon(replace->get_octagon());
    curr->set_count_fig(replace->get_count_fig());

    if (rep_parent->get_left() == replace)
        rep_parent->set_left(nullptr);
    else
        rep_parent->set_right(nullptr);
    delete(replace);
    return;
}

bool TBinaryTree::Empty() {
    return t_root == nullptr ? true : false;
}

void Tree_out (std::ostream& os, TreeElem* curr);
std::ostream& operator<<(std::ostream& os, const TBinaryTree& 
tree) {
    TreeElem* curr = tree.t_root;
    Tree_out(os, curr);
    return os;
}

void Tree_out (std::ostream& os, TreeElem* curr) {
    if (curr)
    {
        if(curr->get_octagon().Area() >= 0)
            os << curr->get_count_fig() << "*" << curr-
>get_octagon().Area();
        if(curr->get_left() || curr->get_right())
        {
            os << ": [";
            if (curr->get_left())
                Tree_out(os, curr->get_left());
            if(curr->get_left() && curr->get_right())
                os << ", ";
            if (curr->get_right())
                Tree_out(os, curr->get_right());
            os << "]";
        }
    }
}
```

```cpp
void recursive_clear(TreeElem* curr);
void TBinaryTree::Clear() {
    if (t_root->get_left())
        recursive_clear(t_root->get_left());
    t_root->set_left(nullptr);
    if (t_root->get_right())
        recursive_clear(t_root->get_right());
    t_root->set_right(nullptr);
    delete t_root;
    t_root = nullptr;
}

void recursive_clear(TreeElem* curr){
    if(curr)
    {
        if (curr->get_left())
            recursive_clear(curr->get_left());
        curr->set_left(nullptr);
        if (curr->get_right())
            recursive_clear(curr->get_right());
        curr->set_right(nullptr);
        delete curr;
    }
}

TBinaryTree::~TBinaryTree() {
}
```

tree_elem.cpp

```cpp
#include <iostream>
#include <memory>
#include "tree_elem.h"


TreeElem::TreeElem() {
    octi = nullptr;
    count_fig = 0;
    t_left = nullptr;
    t_right = nullptr;
}

TreeElem::TreeElem(const Octagon octagon) {
    octi = MakeSPTR(Octagon)(octagon);
    count_fig = 1;
    t_left = nullptr;
    t_right = nullptr;
}

const Octagon& TreeElem::get_octagon() const{
    return *octi;
}
int TreeElem::get_count_fig() const{
    return count_fig;
}
```

```cpp
SPTR(TreeElem) TreeElem::get_left() const{
    return t_left;
}
SPTR(TreeElem) TreeElem::get_right() const{
    return t_right;
}

void TreeElem::set_octagon(const Octagon& octagon){
    octi = MakeSPTR(Octagon)(octagon);
}
void TreeElem::set_count_fig(const int count) {
    count_fig = count;
}
void TreeElem::set_left(SPTR(TreeElem) to_left) {
    t_left = to_left;
}
void TreeElem::set_right(SPTR(TreeElem) to_right) {
    t_right = to_right;
}

TreeElem::~TreeElem() {
}
```

```
tbinarytree.cpp
```

```cpp
#include "tbinarytree.h"
#include <stdexcept>

TBinaryTree::TBinaryTree() {
    t_root = nullptr;
}

void TBinaryTree::Push(const Octagon& octagon) {
    SPTR(TreeElem) curr = t_root;
    SPTR(TreeElem) OctSptr(new TreeElem(octagon));

    if (!curr)
    {
        t_root = OctSptr;
    }
    while (curr)
    {
        if (curr->get_octagon() == octagon)
        {
            curr->set_count_fig(curr->get_count_fig() + 1);
            return;
        }
        if (octagon.Area() < curr->get_octagon().Area())
            if (curr->get_left() == nullptr)
            {
                curr->set_left(OctSptr);
                return;
            }
        if (octagon.Area() >= curr->get_octagon().Area())
```

```
            if (curr->get_right() == nullptr && !(curr-
>get_octagon() == octagon))
            {
                curr->set_right(OctSptr);
                return;
            }
        if (curr->get_octagon().Area() > octagon.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }
}

const Octagon& TBinaryTree::GetItemNotLess(double area) {
    SPTR(TreeElem) curr = t_root;
    while (curr)
    {
        if (area == curr->get_octagon().Area()) 
            return curr->get_octagon();
        if (area < curr->get_octagon().Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (area >= curr->get_octagon().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    throw std::out_of_range("out of range");
}

size_t TBinaryTree::Count(const Octagon& octagon) {
    size_t count = 0;
    SPTR(TreeElem) curr = t_root;

    while (curr)
    {
        if (curr->get_octagon() == octagon)
            count = curr->get_count_fig();
        if (octagon.Area() < curr->get_octagon().Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (octagon.Area() >= curr->get_octagon().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    return count;
}
```

```cpp
void Pop_List(SPTR(TreeElem) curr, SPTR(TreeElem) parent);
void Pop_Part_of_Branch(SPTR(TreeElem) curr, SPTR(TreeElem) 
parent);
void Pop_Root_of_Subtree(SPTR(TreeElem) curr, SPTR(TreeElem) 
parent);
void TBinaryTree::Pop(const Octagon& octagon) {

    SPTR(TreeElem) curr = t_root;
    SPTR(TreeElem) parent = nullptr;

    while (curr && curr->get_octagon() != octagon)
    {
        parent = curr;
        if (curr->get_octagon().Area() > octagon.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }

    if (curr == nullptr)
        return;

    curr->set_count_fig(curr->get_count_fig() - 1);

    if(curr->get_count_fig() <= 0)
    {
        if (curr->get_left() == nullptr && curr->get_right() == 
nullptr)
        {
            Pop_List(curr, parent);
            return;
        }
        if (curr->get_left() == nullptr || curr->get_right() == 
nullptr)
        {
            Pop_Part_of_Branch(curr, parent);
            return;
        }
        if (curr->get_left() != nullptr && curr->get_right() != 
nullptr)
        {
           Pop_Root_of_Subtree(curr, parent);
           return;
        }
    }
}

void Pop_List(SPTR(TreeElem) curr, SPTR(TreeElem) parent) {
    if (parent->get_left() == curr)
                parent->set_left(nullptr);
            else
                parent->set_right(nullptr);
}
```

```cpp
void Pop_Part_of_Branch(SPTR(TreeElem) curr, SPTR(TreeElem) 
parent) {
    if (parent) {
        if (curr->get_left()) {
            if (parent->get_left() == curr)
                parent->set_left(curr->get_left());

            if (parent->get_right() == curr)
                parent->set_right(curr->get_left());

            curr->set_right(nullptr);
            curr->set_left(nullptr);
            return;
        }

        if (curr->get_left() == nullptr) {
            if (parent && parent->get_left() == curr)
                parent->set_left(curr->get_right());

            if (parent && parent->get_right() == curr)
                parent->set_right(curr->get_right());

            curr->set_right(nullptr);
            curr->set_left(nullptr);
            return;
        }
    }
}

void Pop_Root_of_Subtree(SPTR(TreeElem) curr, SPTR(TreeElem) 
parent) {
    SPTR(TreeElem) replace = curr->get_left();
    SPTR(TreeElem) rep_parent = curr;
    while (replace->get_right())
    {
        rep_parent = replace;
        replace = replace->get_right();
    }

    curr->set_octagon(replace->get_octagon());
    curr->set_count_fig(replace->get_count_fig());

    if (rep_parent->get_left() == replace)
        rep_parent->set_left(nullptr);
    else
        rep_parent->set_right(nullptr);
    return;
}

bool TBinaryTree::Empty() {
    return t_root == nullptr ? true : false;
}
```

```cpp
void Tree_out (std::ostream& os, SPTR(TreeElem) curr);
std::ostream& operator<<(std::ostream& os, const TBinaryTree& 
tree) {
    SPTR(TreeElem) curr = tree.t_root;
    Tree_out(os, curr);
    return os;
}

void Tree_out (std::ostream& os, SPTR(TreeElem) curr) {
    if (curr)
    {
        if(curr->get_octagon().Area() >= 0)
            os << curr->get_count_fig() << "*" << curr-
>get_octagon().Area();
        if(curr->get_left() || curr->get_right())
        {
            os << ": [";
            if (curr->get_left())
                Tree_out(os, curr->get_left());
            if(curr->get_left() && curr->get_right())
                os << ", ";
            if (curr->get_right())
                Tree_out(os, curr->get_right());
            os << "]";
        }
    }
}

void recursive_clear(SPTR(TreeElem) curr);
void TBinaryTree::Clear() {
    if (t_root->get_left())
        recursive_clear(t_root->get_left());
    t_root->set_left(nullptr);
    if (t_root->get_right())
        recursive_clear(t_root->get_right());
    t_root->set_right(nullptr);
    t_root = nullptr;
}

void recursive_clear(SPTR(TreeElem) curr){
    if(curr)
    {
        if (curr->get_left())
            recursive_clear(curr->get_left());
        curr->set_left(nullptr);
        if (curr->get_right())
            recursive_clear(curr->get_right());
        curr->set_right(nullptr);
    }
}

TBinaryTree::~TBinaryTree() {
}
```