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: *Пономарев Никита Владимирович, 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 №19:

- Фигура: Прямоугольник(Rectangle)
- Контейнер: Н-дерево (TNaryTree)

Описание программы:

Исходный код разделён на 9 файлов:

1. point.h – описание класса точки
2. point.cpp – реализация класса точки
3. rectangle.h – описание класса квадрата
4. rectangle.cpp – реализация класса квадрата
5. TNaryTree_item.h – описание элемента н-дерева с шаблонами
6. TNaryTree_item.cpp – реализация элемента н-дерева с шаблонами
7. TNaryTree.h – описание н-дерева с шаблонами
8. TNaryTree.cpp – реализация н-дерева с шаблонами
9. Iterator.h – описание и реализация итератора н-дерева с шаблонами
10. main.cpp – основная программа

Дневник отладки:

При выполнении работы ошибок выявлено не было.

Вывод:

В процессе выполнения работы я на практике познакомился с итераторами. Они позволяют легко реализовать обход всех элементов некоторой структыр данных, позволяют использовать цикл range-based-for и для самописных структур. Поэтому я уверен, что знания, полученные в этой лабораторной работе, обязательно пригодятся мне.

Исходный код:

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);
        double X();
        double Y();
        friend std::istream& operator>>(std::istream& is, Point& p);
        friend std::ostream& operator<<(std::ostream& os, const Point& p);

    private:
        double x_;
        double y_;
};

#endif // POINT_H
```

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, const Point& p) {
  os << "(" << p.x_ << ", " << p.y_ << ")";
  return os;
}
```

rectangle.h:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

#include "figure.h"

class Rectangle: Figure {
    public:
        size_t VertexesNumber();
        double Area();
        void Print(std::ostream& os);
        Rectangle();
        Rectangle(Point a_, Point b_, Point c_, Point d_);
        Rectangle(std::istream& is);
        friend std::istream &operator>>(std::istream &is, Rectangle &figure);
        friend std::ostream &operator<<(std::ostream &os, const Rectangle &figure);
    private:
        Point a;
        Point b;
        Point c;
        Point d;
};
```

```
#endif
```

rectangle.cpp:

```
#include "point.h"
#include "rectangle.h"


double Rectangle::Area(){
    return a.dist(b) * b.dist(c);
}


void Rectangle::Print(std::ostream& os){
    os << a << " " << b << " " << c << " " << d << "\n";
}


size_t Rectangle::VertexesNumber(){
    return (size_t)(4);
}


Rectangle::Rectangle() : a(Point()), b(Point()), c(Point()), d(Point()){
}


Rectangle::Rectangle(Point a_, Point b_, Point c_, Point d_):
                                a(a_), b(b_), c(c_), d(d_){
}


Rectangle::Rectangle(std::istream& is){
    is >> a >> b >> c >> d;
}
std::istream &operator>>(std::istream &is, Rectangle &figure){
    is >> figure.a >> figure.b >> figure.c >> figure.d;
    return is;
}


std::ostream &operator<<(std::ostream &os, const Rectangle &figure){
    os << "Rectangle: " << figure.a << " " << figure.b << " " << figure.c << " " <<
figure.d << std::endl;
    return os;

}
```

TNaryTree_item.h:

```
#ifndef TNARYTREE_ITEM_H
#define TNARYTREE_ITEM_H


#include <memory>
```

```
template <class T>
class Item {
    public:
        Item(T a);
        Item(std::shared_ptr<Item<T>> a);
        Item();
        void Set(T a);
        void Set_bro(std::shared_ptr<Item<T>> bro_);
        void Set_son(std::shared_ptr<Item<T>> son_);
        Item Get_data();
        std::shared_ptr<Item<T>> Get_bro();
        std::shared_ptr<Item<T>> Get_son();
        void Print(std::ostream &os);
        double Area();
        ~Item();

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

    private:
        std::shared_ptr<Item<T>> bro = nullptr;
        std::shared_ptr<Item<T>> son = nullptr;
        T data;
};

#endif
```

TNaryTree_item.cpp:

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

template <class T>
Item<T>::Item() {
    data = T();
}

template <class T>
Item<T>::Item(T a){
    data = a;
}

template <class T>
```

```
void Item<T>::Set(T a){
    data = a;
}

template <class T>
Item<T> Item<T>::Get_data(){
    return data;
}

template <class T>
std::shared_ptr<Item<T>> Item<T>::Get_bro(){
    return bro;
}

template <class T>
std::shared_ptr<Item<T>> Item<T>::Get_son(){
    return son;
}

template <class T>
Item<T>::Item(std::shared_ptr<Item<T>> a){
    bro = a->bro;
    son = a->son;
    data = a->data;
}

template <class T>
void Item<T>::Print(std::ostream &os){
    os << data.Area();
}

template <class T>
void Item<T>::Set_bro(std::shared_ptr<Item<T>> bro_){
    bro = bro_;
}

template <class T>
void Item<T>::Set_son(std::shared_ptr<Item<T>> son_){
    son = son_;
}

template <class T>
```

```
double Item<T>::Area(){
    return data.Area();
}

template <class T>
std::ostream &operator<<(std::ostream &os, const Item<T> &obj)
{
    os << "Item: " << obj.data << std::endl;
    return os;
}

template <class T>
Item<T>::~Item() {};

#include "rectangle.h"
template class Item<Rectangle>;
template std::ostream& operator<<(std::ostream& os, const Item<Rectangle> &obj);
```

TNaryTree.h:

```
#ifndef TNARYTREE_H
#define TNARYTREE_H

#include "TnaryTree_item.h"
#include "Iterator.h"

template <class T>
class TNaryTree
{
    public:
        // Инициализация дерева с указанием размера
        TNaryTree(int n);
        // Полное копирование дерева
        TNaryTree(const TNaryTree<T>& other);
        // Добавление или обновление вершины в дереве согласно заданному пути.
        // Путь задается строкой вида: "cbccbccc",
        // где 'c' - старший ребенок, 'b' - младший брат
        // последний символ строки - вершина, которую нужно добавить или обновить.
        // Пустой путь "" означает добавление/обновление корня дерева.
        // Если какой-то вершины в tree_path не существует,
        //  то функция должна бросить исключение std::invalid_argument
        // Если вершину нельзя добавить из за переполнения,
        //  то функция должна бросить исключение std::out_of_range
        void Update(T &&polygon, std::string &&tree_path = "");
```

```
        // Удаление поддерева
        void Clear(std::string &&tree_path = "");
        // Проверка наличия в дереве вершин
        bool Empty();
        // Подсчет суммарной площади поддерева
        double Area(std::string &&tree_path);
        int size();
        // Вывод дерева в формате вложенных списков, где каждый вложенный список
является:
        // "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры
        template <class A>
        friend std::ostream& operator<<(std::ostream& os, const TNaryTree<A>& tree);

        virtual ~TNaryTree();

    private:
        int curr_number;
        int max_number;
        std::shared_ptr<Item<T>> root;
};

#endif
```

TNaryTree.cpp:

```
#include "TNaryTree.h"
#include <string>
#include <memory>
#include <stdexcept>
#include <iostream>

template <class T>
TNaryTree<T>::TNaryTree(int n) {
    max_number = n;
    curr_number = 0;
    root = nullptr;
};

template <class T>
bool TNaryTree<T>::Empty() {
    return curr_number ? 0 : 1;
}

template <class T>
```

```
void TNaryTree<T>::Update(T &&polygon, std::string &&tree_path){
    if(tree_path != "" && curr_number == 0){
        throw std::invalid_argument("Error, there is not a root value\n");
        return;
    } else if(tree_path == "" && curr_number == 0){
        std::shared_ptr<Item<T>> q(new Item<T>(polygon));
        root = q;
        curr_number++;
    } else if(curr_number + 1 > max_number){
        throw std::out_of_range("Current number of elements equals maximal number of
elements in tree\n");
        return;
    } else {
        std::shared_ptr<Item<T>> tmp = root;
        for(size_t i = 0; i < tree_path.length() - 1; i++) {
            if(tree_path[i] == 'b'){
                std::shared_ptr<Item<T>> q((*tmp).Get_bro());
                if(q == nullptr){
                    throw std::invalid_argument("Path does not exist\n");
                    return;
                }
                tmp = q;
            } else if(tree_path[i] == 'c'){
                std::shared_ptr<Item<T>> q = (*tmp).Get_son();
                if(q == nullptr){
                    throw std::invalid_argument("Path does not exist\n");
                    return;
                }
                tmp = q;
            } else {
                throw std::invalid_argument("Error in path\n");
                return;
            }
        }
        std::shared_ptr<Item<T>> item(new Item<T>(polygon));
        if(tree_path.back() == 'b'){
            /*std::shared_ptr<Item> p = (*tmp).Get_bro();
            p = item;*/
            (*tmp).Set_bro(item);
            curr_number++;
        } else if(tree_path.back() == 'c'){
            /*std::shared_ptr<Item> p = (*tmp).Get_son();
```

```
            p = item;*/
            (*tmp).Set_son(item);
            curr_number++;
        } else {
            throw std::invalid_argument("Error in path\n");
            return;
        }
    }
}

template <class T>
std::shared_ptr<Item<T>> copy(std::shared_ptr<Item<T>> root){
    if(!root){
        return nullptr;
    }
    std::shared_ptr<Item<T>> root_copy(new Item<T>(root));
    (*root_copy).Set_bro(copy((*root).Get_bro()));
    (*root_copy).Set_son(copy((*root).Get_son()));
    return root_copy;
}

template <class T>
TNaryTree<T>::TNaryTree(const TNaryTree<T>& other){
    curr_number = 0;
    max_number = other.max_number;
    root = copy(other.root);
    curr_number = other.curr_number;
;}

template <class T>
int TNaryTree<T>::size(){
    return curr_number;
}

template <class T>
int clear(std::shared_ptr<Item<T>> node) {
    if (!node) {
        return 0;
    }
    int temp_res = clear((*node).Get_bro()) + clear((*node).Get_son()) + 1;
    return temp_res;
}
```

```
template <class T>
void TNaryTree<T>::Clear(std::string &&tree_path){
    std::shared_ptr<Item<T>> prev_tmp = nullptr;
    std::shared_ptr<Item<T>> tmp;
    tmp = root;
    if (tree_path.empty()) {
        clear(root);
        curr_number = 0;
        root = nullptr;
        return;
    }
    for(size_t i = 0; i < tree_path.length(); i++) {
        if(tree_path[i] == 'b'){
            std::shared_ptr<Item<T>> q((*tmp).Get_bro());
            if(q == nullptr){
                throw std::invalid_argument("Path does not exist\n");
                return;
            }
            prev_tmp = tmp;
            tmp = q;
        } else if(tree_path[i] == 'c'){
            std::shared_ptr<Item<T>> q((*tmp).Get_son());
            if(q == nullptr){
                throw std::invalid_argument("Path does not exist\n");
                return;
            }
            prev_tmp = tmp;
            tmp = q;
        } else {
            throw std::invalid_argument("Error in path\n");
            return;
        }
    }
    if (tmp == (*prev_tmp).Get_son()) {
        (*prev_tmp).Set_son(nullptr);
    } else {
        (*prev_tmp).Set_bro(nullptr);
    }
    curr_number -= clear(tmp);
}
```

```
template <class T>
double area(std::shared_ptr<Item<T>> node){
    if(!node){
        return 0;
    }
    return node->Area() + area((*node).Get_bro()) + area((*node).Get_son());
}

template <class T>
double TNaryTree<T>::Area(std::string &&tree_path){
    std::shared_ptr<Item<T>> tmp;
    tmp = root;
    for(size_t i = 0; i < tree_path.length(); i++) {
        if(tree_path[i] == 'b'){
            std::shared_ptr<Item<T>> q((*tmp).Get_bro());
            if(q == nullptr){
                throw std::invalid_argument("Path does not exist\n");
                return -1;
            }
            tmp = q;
        } else if(tree_path[i] == 'c'){
            std::shared_ptr<Item<T>> q((*tmp).Get_son());
            if(q == nullptr){
                throw std::invalid_argument("Path does not exist\n");
                return -1;
            }
            tmp = q;
        } else {
            throw std::invalid_argument("Error in path\n");
            return -1;
        }
    }
    return area(tmp);
}

// Вывод дерева в формате вложенных списков, где каждый вложенный список является:
// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры

template <class T>
void print(std::ostream& os, std::shared_ptr<Item<T>> node){
    if(!node){
        return;
```

```
    }
    if((*node).Get_son()){
        //os <<  <<node->pentagon.GetArea() <<  : ]" <<
        os << node->Area() << ": [";
        print(os, (*node).Get_son());
        os << "]";
        if((*node).Get_bro()){
            os << ", ";
            print(os, (*node).Get_bro());
        }
    } else if ((*node).Get_bro()) {
        os << node->Area() << ", ";
        print(os, (*node).Get_bro());
        if((*node).Get_son()){
            os << ": [";
            print(os, (*node).Get_son());
            os << "]";
        }
    }
    else {
        os << node->Area();
    }
}

template <class T>
std::ostream& operator<<(std::ostream& os, const TNaryTree<T>& tree){
    print(os, tree.root);
    os << "\n";
    return os;
}

template <class T>
TNaryTree<T>::~TNaryTree(){
    Clear();
};

#include "rectangle.h"
template class TNaryTree<Rectangle>;
template std::ostream& operator<<(std::ostream& os, const TNaryTree<Rectangle>& stack);
```

Iterator.h:

```
#ifndef ITERATOR_H
```

```
#define ITERATOR_H

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

template <class node, class T>
class Iterator{
    public:
        Iterator(std::shared_ptr<node> n){
            it = n;
            it_prev = nullptr;
        }
        Iterator& operator=(const Iterator& it_){
            it = it_.it;
            return *this;
        }
        bool operator==(const Iterator& it_){
            return it == it_.it;
        }
        bool operator!=(const Iterator& it_){
            return !(it == it_.it);
        }
        node operator*(){
            return it->Get_data();
        }
        Iterator& operator++ (){
            if(it->Get_son() != nullptr && it_prev != it->Get_bro()){
                it_prev = it;
                it = it->Get_son();
            } else if (it->Get_bro() != nullptr && it_prev != it->Get_bro()){
                it_prev = it;
                it = it->Get_bro();
            } else {
                while(it->Get_bro() == nullptr || (it->Get_bro() == it_prev)){
                    it_prev = it;
                    if(it->Get_older() == nullptr){
                        return *this;
                    }
                    it = it->Get_older();
                }
                if(it->Get_bro() == nullptr && it->Get_son() == it_prev){
                    it_prev = it;
```

```
                    it = it->Get_older();
                    if(it->Get_older() == nullptr){
                        return *this;
                    }
                    if(it->Get_bro() == nullptr){
                        while(it->Get_bro() == nullptr){
                            it_prev = it;
                            it = it->Get_older();
                        }
                        it_prev = it;
                        it = it->Get_bro();
                    } else {
                    it = it->Get_bro();
                    }
                } else {
                    it = it->Get_bro();
                }
            }
            return *this;
        }
        Iterator& operator++ (int){
            Iterator tmp(*it);
            ++(*this);
            return it;
        }
        Iterator& operator-- (){
            it = it.Get_older();
            return *this;
        }
        Iterator& operator-- (int){
            Iterator tmp(*it);
            --(*this);
            return it;
        }
    public:
        std::shared_ptr<node> it_prev;
        std::shared_ptr<node> it;
};

#endif
```

main.cpp:

```
#include "TNaryTree.h"
```

```cpp
#include "rectangle.h"

int main(void)
{
    TNaryTree<Rectangle> t(5);
    t.Update(Rectangle(Point(0, 0), Point(1, 0),Point(1, 2), Point(0, 2)), "");
    t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "b");
    t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "bb");
    t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "bbc");
    t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "c");
    std::cout << t.size() << "\n";
    std::cout << t.Area("") << "\n";
    std::cout << t.size() << "\n";
    std::cout << t;
    TNaryTree<Rectangle> q(t);
    t.Clear();
    std::cout << q.size() << " " << q.Area("") << "\n";
    std::cout << q;
    return 0;
}
```

Результат работы:

```
5
17
5
5 17
1: [4], 4, 4: [4]

1: [4], 4, 4: [4]
0
```