

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

ЛАБОРАТОРНАЯ РАБОТА №2
по курсу объектно-ориентированное программирование I семестр, 2021/22
уч. год

Студент Пономарев Никита Владимирович, группа М8О-207Б-20

Преподаватель Дорохов Евгений Павлович

Условие

Задание: Вариант 10:

Фигура: Прямоугольник,

Контейнер: n-дерево .

Спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру. Классы должны удовлетворять следующим правилам:

1. Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
2. Классы фигур должны содержать набор следующих методов:
 - (a) Перегруженный оператор ввода координат вершин фигуры из потока `std::istream` («»). Он должен заменить конструктор, принимающий координаты вершин из стандартного потока.
 - (b) Перегруженный оператор вывода в поток `std::ostream` («»), заменяющий метод `Print` из лабораторной работы 1.
 - (c) Оператор копирования (`=`)
 - (d) Оператор сравнения с такими же фигурами (`==`)
3. Класс-контейнер должен содержать объекты фигур “по значению” (не по ссылке).
4. Класс-контейнер должен содержать набор следующих методов:
 - (a) `Size()` – возвращает количество элементов в контейнере
 - (b) `Empty()` – если контейнер пуст, возвращает 1, иначе – 0
 - (c) `Update()` - вставляет переданный элемент по переданному адресу в n-дерево.
 - (d) `Area()` - вычисляет суммарную площадь прямоугольников, входящих в под-дерево по переданному адресу
 - (e) `operator«` – выводит n-дерево в соответствии с заданным форматом в поток вывода

Нельзя использовать:

1. Стандартные контейнеры `std`.
2. Шаблоны (`template`).
3. Различные варианты умных указателей (`shared_ptr`, `weak_ptr`).

Программа должна позволять:

1. Вводить произвольное количество фигур и добавлять их в контейнер.
2. Распечатывать содержимое контейнера.
3. Удалять фигуры из контейнера.

Описание программы

Исходный код лежит в 11 файлах:

1. `main.cpp`: основная программа, взаимодействие с пользователем посредством команд из меню
2. `figure.h`: описание абстрактного класса фигур
3. `point.h`: описание класса точки
4. `TNaryTree_item.h`: описание класса элемента n-дерева
5. `TNaryTree.h`: описание класса n-дерева
6. `rectangle.cpp`: описание класса прямоугольника, наследующегося от `figures`
7. `point.cpp`: реализация класса точки
8. `rectangle.cpp`: реализация класса прямоугольника, наследующегося от `figures`
9. `TNaryTree.cpp`: реализация класса n-дерева
10. `TNaryTree_item.cpp`: реализация класса элемента n-дерева

Дневник отладки

Возникли проблемы при выводе дерева в заданном формате. Сложно было организовать рекурсию верным способом, чтобы все элементы дерева выводились в верном порядке. Возникли проблемы при добавлении элементов в дерево, так как изначально забывал инициализировать элемент дерева нулевыми ссылками на элемент-сына и элемент-брата. Все эти ошибки были обнаружены в процессе тестирования и успешно исправлены.

Недочёты

Выводы

В процессе выполнения работы я на практике познакомился с работой класса-контейнера n-дерево, реализовал его, а также конструкторы и функции для работы с ним, выполнил перегрузку оператора вывода. Также я освоил работу с выделением и очисткой памяти на языке C++ при помощи команд `new` и `delete`.

Исходный код:

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

#include "point.h"

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

#endif
```

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, 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, 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);  
    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;
}

```

TNaryTree_item.h

```

#ifndef TNARYTREE_ITEM_H
#define TNARYTREE_ITEM_H

#include "rectangle.h"

class Item{
public:
    Item(Point a_, Point b_, Point c_, Point d_);
    Item(Rectangle a);
    Item(Item* a);
    Item();
    void Print(std::ostream &os);
    Item* Get_bro();
    Item* Get_son();
    void Set_bro(Item* a);
    void Set_son(Item* a);
    double Area();
    ~Item();
private:
    Item* bro = nullptr;
    Item* son = nullptr;
    Rectangle data;
};

#endif

```

TNaryTree_item.cpp

```

#include "TNaryTree_item.h"

Item::Item(Point a_, Point b_, Point c_, Point d_){
    data = Rectangle(a_, b_, c_, d_);
}

Item::Item(){
    data = Rectangle();
}

```

```

}

Item::Item(Rectangle a){
    data = a;
}

Item::Item(Item* a){
    bro = a->bro;
    son = a->son;
    data = a->data;
}

Item* Item::Get_bro(){
    return bro;
}

Item* Item::Get_son(){
    return son;
}

void Item::Set_bro(Item* a){
    bro = a;
}

void Item::Set_son(Item* a){
    son = a;
}

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

double Item::Area(){
    return data.Area();
}

Item::~Item(){};

```

TNaryTree.h

```

#ifndef TNARYTREE_H
#define TNARYTREE_H

```



```

#include "TNaryTree_item.h"
#include "point.h"
#include "rectangle.h"
#include "figure.h"

class TNaryTree {
public:
    // Инициализация дерева с указанием размера
    TNaryTree(int n);
    // Полное копирование дерева
    TNaryTree(const TNaryTree& other);
    // Добавление или обновление вершины в дереве согласно заданному пути.
    // Путь задается строкой вида: "сбссбссс",
    // где 'с' - старший ребенок, 'б' - 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(Rectangle &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
    friend std::ostream& operator<<(std::ostream& os, const TNaryTree& tree);
    virtual ~TNaryTree();
private:
    int curr_number;
    int max_number;
    Item* root;
};

#endif

```

TNaryTree.cpp

```

#include "TNaryTree.h"
#include <string>
#include <stdexcept>

TNaryTree::TNaryTree(int n) {
    max_number = n;
    curr_number = 0;
    root = nullptr;
};

bool TNaryTree::Empty(){
    return curr_number ? 0 : 1;
}

void TNaryTree::Update(Rectangle &&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){
        Item* q = (new Item(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 ele
        return;
    } else {
        Item* tmp = root;
        for(int i = 0; i < tree_path.length() - 1; i++){
            if(tree_path[i] == 'b'){
                Item* 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'){
                Item* 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;
        }
    }
    Item* item(new Item(polygon));
    if(tree_path.back() == 'b'){
        tmp->Set_bro(item);
        curr_number++;
    } else if(tree_path.back() == 'c'){
        tmp->Set_son(item);
        curr_number++;
    } else {
        throw std::invalid_argument("Error in path\n");
        return;
    }
}

Item* copy(Item* root){
    if(!root){
        return nullptr;
    }
    Item *root_copy = new Item(root);
    root_copy->Set_bro(copy(root->Get_bro()));
    root_copy->Set_son(copy(root->Get_son()));
    return root_copy;
}

TNaryTree::TNaryTree(const TNaryTree& other){
    curr_number = 0;
    max_number = other.max_number;
    root = copy(other.root);
    curr_number = other.curr_number;
};

int TNaryTree::size(){
    return curr_number;
}

int clear(Item* node) {
    if (!node) {

```

```

        return 0;
    }
    int temp_res = clear(node->Get_bro()) + clear(node->Get_son()) + 1;
    delete node;
    return temp_res;
}

void TNaryTree::Clear(std::string &&tree_path){
    Item* prev_tmp = nullptr;
    Item* tmp = root;
    if (tree_path.empty()) {
        clear(root);
        curr_number = 0;
        root = nullptr;
        return;
    }
    for(int i = 0; i < tree_path.length(); i++){
        if(tree_path[i] == 'b'){
            Item* 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'){
            Item* 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);
}

double area(Item* node){
    if(!node){
        return 0;
    }
    return node->Area() + area(node->Get_bro()) + area(node->Get_son());
}

double TNaryTree::Area(std::string &&tree_path){
    Item* tmp = root;
    for(int i = 0; i < tree_path.length(); i++){
        if(tree_path[i] == 'b'){
            Item* 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'){
            Item* 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*

```

void print(std::ostream& os, Item* node){
    if(!node){
        return;
    }
}

```

```

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

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

TNaryTree::~TNaryTree(){
    Clear();
};

```

main.cpp

```

#include <iostream>
#include "TNaryTree_item.h"
#include "point.h"
#include "rectangle.h"

```

```

#include "figure.h"
#include "TNaryTree.h"

int main(){
    TNaryTree t(5);
    t.Update(Rectangle(Point(0, 0), Point(1, 0),Point(1, 1), Point(0, 1)), "");
    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";
    TNaryTree q(t);
    std::cout << q.size() << " " << q.Area("") << "\n";
    std::cout << t << '\n' << q;
    // std::cout << q.root; //<< " " << q.root->bro << " " << q.root->bro->bro << " " <<
    // std::cout << t.root << " " << t.root->bro << " " << t.root->bro->bro << " " << t.
    // std::cout << t.root->bro->son << "\n";
    // delete (t.root);
    // t.root->bro->Print(std::cout);
    t.Clear("");
    std::cout << t.Area("") << "\n";
}

```