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

ЛАБОРАТОРНАЯ РАБОТА №7 по курсу

объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Прохоров Данила Михайлович, группа М80-208Б-20*
Преподаватель *Дорохов Евгений Павлович*

Цель работы

Целью лабораторной работы является:

Закрепление навыков работы с шаблонами классов;

Построение итераторов для динамических структур данных.

Задание

Используя структуру данных, разработанную для лабораторной работы №4, спроектировать и разработать **итератор** для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа for. Например:

```
for(auto i : stack) {
  std::cout << *i << std::endl;
}
```

Нельзя использовать:

Стандартные контейнеры std.

Программа должна позволять:

Вводить произвольное количество фигур и добавлять их в контейнер;

Распечатывать содержимое контейнера;

Удалять фигуры из контейнера.

Дневник отладки

Циклом никак нельзя вывести дерево общего вида, поэтому функция распечатки рекурсивная. Из-за этого возникали проблемы в отладке, да и программа всё растёт и растёт, конца-края не видно, отлаживать становится всё сложнее.

Недочёты

Недочётов вроде бы нет

Выводы

Лабораторная работа №7 позволила мне реализовать свой итератор, с помощью которого осуществляется вывод контейнера в консоль.

Исходный код

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H
#include "point.h"

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

main.cpp

```
#include <iostream>
#include "tnarytree.h"
int main()
{
    double S = 0.;
    std::string string;
    TNaryTree<Rectangle> t1(3);
    std::cout << t1;
    t1.Update(Rectangle(std::cin), "cbc");
    t1.Update(Rectangle(std::cin), "");
    t1.Update(Rectangle(std::cin));
    t1.Update(Rectangle(std::cin), "c");
    t1.Update(Rectangle(std::cin), "cb");
    std::cout << t1.getItem("cb");
    std::cout << t1.getItem("cbb");
    t1.Update(Rectangle(std::cin), "cc");
    t1.Update(Rectangle(std::cin), "cbb");
    std::cout << t1;
    t1.Update(Rectangle(std::cin), "cbbb");
    if (((S = t1.Area()) == -1))
    {
        std::cout << "There is no such element in tree" << std::endl;
    }
    else
```

```
{
    std::cout << "Area of subtree is " << S << std::endl;
}
if (((S = t1.Area("cbbcccbc")) == -1))
{
    std::cout << "There is no such element in tree" << std::endl;
}
else
{
    std::cout << "Area of subtree is " << S << std::endl;
}
t1.Update(Rectangle(std::cin), "cbc");
std::cout << t1;
t1.Update(Rectangle(std::cin), "ccb");
t1.Update(Rectangle(std::cin), "ccbb");
t1.Update(Rectangle(std::cin), "cbcb");
t1.Update(Rectangle(std::cin), "cbcbb");
std::cout << t1;
if (((S = t1.Area("c")) == -1))
{
    std::cout << "There is no such element in tree" << std::endl;
}
else
{
    std::cout << "Area of subtree is " << S << std::endl;
}
t1.Update(Rectangle(std::cin), "cbbbc");
std::cout << t1;
t1.Update(Rectangle(std::cin), "cbbc");
t1.Update(Rectangle(std::cin), "cbb");
std::cout << t1;
t1.Update(Rectangle(std::cin), "cbbcb");
t1.Update(Rectangle(std::cin), "cbbcbb");
t1.Update(Rectangle(std::cin), "ccbc");
t1.Update(Rectangle(std::cin), "cbbcbc");
t1.Update(Rectangle(std::cin), "cbbd");
t1.Update(Rectangle(std::cin), "cbbcbbc");
t1.Update(Rectangle(std::cin), "cbbcbbcb");
std::cout << t1.getItem("cbbcbbcb");
std::cout << t1.getItem("cbbbbbcbcbcbcccbcc");
std::cout << t1;
TNaryTree<Rectangle> t3(t1);

t3.Update(Rectangle(std::cin));
t3.Update(Rectangle(std::cin), "cbbcbbcbb");
t3.Update(Rectangle(std::cin), "cbbcc");

std::cout << t1 << t3;

t1.RemoveSubTree("ccc");
t1.RemoveSubTree("b");
t1.RemoveSubTree("ccbcbb");
std::cout << t1;
t1.RemoveSubTree("cb");
std::cout << t1;
t1.RemoveSubTree("cbb");
std::cout << t1;
t1.RemoveSubTree("cb");
std::cout << t1;
```

```
    t1.RemoveSubTree("cbbcb");
    std::cout << t1;
    t1.RemoveSubTree("ccb");
    std::cout << t1;
    t1.RemoveSubTree();
    std::cout << t1 << t3;
    TNaryTree<Trapezoid> t2(7);
    t2.Update(Trapezoid(std::cin));
    t2.Update(Trapezoid(std::cin), "c");
    t2.Update(Trapezoid(std::cin), "cb");
    std::cout << t2;
    t2.RemoveSubTree();
    TNaryTree<Rhombus> t4(19);
    t4.Update(Rhombus(std::cin));
    t4.Update(Rhombus(std::cin), "b");
    t4.Update(Rhombus(std::cin), "c");
    t4.Update(Rhombus(std::cin), "cb");
    t4.Update(Rhombus(std::cin), "cbb");
    t4.Update(Rhombus(std::cin), "cbbb");
    t4.Update(Rhombus(std::cin), "cbbbb");
    t4.Update(Rhombus(std::cin), "cbbbbb");
    t4.Update(Rhombus(std::cin), "cbbbbbb");
    t4.Update(Rhombus(std::cin), "cbbbbbbb");
    t4.Update(Rhombus(std::cin), "cbbbbbbbb");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbb");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbc");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbcb");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbb");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbcc");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbccb");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbcccc");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbccc");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbcccc");
    t4.Update(Rhombus(std::cin), "cbbbbbbbbbccccb");
    std::cout << t4;
    t4.RemoveSubTree("cbbbbbbbbbc");
    std::cout << t4;
    t4.RemoveSubTree("cccbcbc");
    t4.RemoveSubTree("c");
    std::cout << t4;
    t4.RemoveSubTree("");
    std::cout << t4;
    system("pause");
    return 0;
}
```

rectangle.cpp

```
#include "rectangle.h"

Rectangle::Rectangle() : a(0.0, 0.0), b(0.0, 0.0), c(0.0, 0.0), d(0.0, 0.0), len1(0),
len2(0), square(0.0)
{
};
```

```
Rectangle::Rectangle(std::istream& is)
{
	is >> a >> b >> c >> d;
	len1 = dist(a, b);
	len2 = dist(b, c);
	square = len1 * len2;
}

Rectangle& Rectangle::operator= (Rectangle rectangle)
{
	a = rectangle.a;
	b = rectangle.b;
	c = rectangle.c;
	d = rectangle.d;
	len1 = rectangle.len1;
	len2 = rectangle.len2;
	square = rectangle.square;
	return rectangle;
};

bool Rectangle::operator== (Rectangle rectangle)
{
	if ((a == rectangle.a) && (b == rectangle.b) && (c == rectangle.c) && (d ==
rectangle.d))
	{
		return true;
	}
	return false;
};

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

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

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

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

double Rectangle::Square()
{
	return  square;
}

Rectangle::~Rectangle()
{
```

```
}
```

rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
#include "figure.h"

class Rectangle : public Figure
{
public:
	Rectangle();
	Rectangle(std::istream& is);
	void Print(std::ostream& os);
	double Square();
	friend std::istream& operator >>(std::istream& is, Rectangle& rectangle);
	friend std::ostream& operator <<(std::ostream& os, Rectangle rectangle);
	Rectangle& operator= (Rectangle rectangle);
	bool operator== (Rectangle rectangle);
	size_t VertexesNumber();
	virtual ~Rectangle();

private:
	Point a, b, c, d;
	double len1, len2;
	double square;
};

#endif
```

rhombus.cpp

```
#include "rhombus.h"

Rhombus::Rhombus() : a(0.0, 0.0), b(0.0, 0.0), c(0.0, 0.0), d(0.0, 0.0), square(0.0),
diag1(0.0), diag2(0.0)
{

};

Rhombus::Rhombus(std::istream& is)
{
	is >> a >> b >> c >> d;
	diag1 = dist(a, c);
	diag2 = dist(b, d);
	square = (diag1 * diag2) / 2.;
}

Rhombus& Rhombus::operator= (Rhombus rhombus)
{
	a = rhombus.a;
	b = rhombus.b;
	c = rhombus.c;
```

```
	d = rhombus.d;
	diag1 = rhombus.diag1;
	diag2 = rhombus.diag2;
	square = rhombus.square;
	return rhombus;
};

bool Rhombus::operator== (Rhombus rhombus)
{
	if ((a == rhombus.a) && (b == rhombus.b) && (c == rhombus.c) && (d == rhombus.d))
	{
		return true;
	}
	return false;
};

void Rhombus::Print(std::ostream& os)
{
	os << "Rhombus: " << a << " " << b << " " << c << " " << d << std::endl;
}

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

std::ostream& operator <<(std::ostream& os, Rhombus rhombus)
{
	os << rhombus.a << " " << rhombus.b << " " << rhombus.c << " " << rhombus.d;
	return os;
};

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

double Rhombus::Square()
{
	return square;
}

Rhombus::~Rhombus()
{

}
```

rhombus.h

```
#ifndef RHOMBUS_H
#define RHOMBUS_H
#include "figure.h"

class Rhombus : public Figure
{
```

```
public:
	Rhombus();
	Rhombus(std::istream& is);
	void Print(std::ostream& os);
	double Square();
	friend std::istream& operator >>(std::istream& is, Rhombus& rhombus);
	friend std::ostream& operator <<(std::ostream& os, Rhombus rhombus);
	Rhombus& operator= (Rhombus rhombus);
	bool operator== (Rhombus rhombus);
	size_t VertexesNumber();
	virtual ~Rhombus();


private:
	Point a, b, c, d;
	double diag1, diag2;
	double square;
};

#endif
```

trapezoid.cpp

```
#include "trapezoid.h"

Trapezoid::Trapezoid() : a(0.0, 0.0), b(0.0, 0.0), c(0.0, 0.0), d(0.0, 0.0), square(0.0),
lena(0.0), lenb(0.0), lenc(0.0), lend(0.0)
{

};

Trapezoid::Trapezoid(std::istream& is)
{
	is >> a >> b >> c >> d;
	lena = dist(b, c);
	lenb = dist(a, d);
	lenc = dist(c, d);
	lend = dist(a, b);
	/*if (lena > lenb)
	{
		std::swap(lena, lenb);
		std::swap(lenc, lend);
	}*/
	square = ((lena + lenb) / 2.) * sqrt(pow(lenc, 2) - pow(((pow(lenb - lena, 2) +
pow(lenc, 2) - pow(lend, 2)) / (2. * (lenb - lena))), 2));
}

Trapezoid& Trapezoid::operator= (Trapezoid trapezoid)
{
	a = trapezoid.a;
	b = trapezoid.b;
	c = trapezoid.c;
	d = trapezoid.d;
	lena = trapezoid.lena;
	lenb = trapezoid.lenb;
	lenc = trapezoid.lenc;
```

```
	lend = trapezoid.lend;
	square = trapezoid.square;
	return trapezoid;
};

bool Trapezoid::operator== (Trapezoid trapezoid)
{
	if ((a == trapezoid.a) && (b == trapezoid.b) && (c == trapezoid.c) && (d == 
trapezoid.d))
	{
		return true;
	}
	return false;
};

void Trapezoid::Print(std::ostream& os)
{
	os << "Trapezoid: " << a << " " << b << " " << c << " " << d << std::endl;
}

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

std::ostream& operator <<(std::ostream& os, Trapezoid trapezoid)
{
	os << trapezoid.a << " " << trapezoid.b << " " << trapezoid.c << " " << trapezoid.d;
	return os;
};

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

double Trapezoid::Square()
{
	return square;
}

Trapezoid::~Trapezoid()
{

}
```

trapezoid.h

```
#ifndef TRAPEZOID_H
#define TRAPEZOID_H
#include "figure.h"
#include <algorithm>
class Trapezoid : public Figure
{
public:
```

```
        Trapezoid();
        Trapezoid(std::istream& is);
        void Print(std::ostream& os);
        double Square();
        friend std::istream& operator >>(std::istream& is, Trapezoid& trapezoid);
        friend std::ostream& operator <<(std::ostream& os, Trapezoid trapezoid);
        Trapezoid& operator= (Trapezoid trapezoid);
        bool operator== (Trapezoid trapezoid);
        size_t VertexesNumber();
        virtual ~Trapezoid();

private:
        Point a, b, c, d;
        double lena, lenb, lenc, lend;
        double square;
};

#endif
```

point.cpp

```
#include "point.h"

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 dist(Point& p1, Point& p2)
{
        double dx = (p1.x_ - p2.x_);
        double dy = (p1.y_ - p2.y_);
        return std::sqrt(dx * dx + dy * dy);
}

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;
}

bool Point::operator == (Point point)
{
        return (x_ == point.x_) && (y_ == point.y_);
}
```

point.h

```
#ifndef POINT_H
#define POINT_H
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <algorithm>
class Point
{
public:
	Point();
	Point(std::istream& is);
	Point(double x, double y);
	double length(Point& p1, Point& p2);
	friend std::istream& operator>>(std::istream& is, Point& p);
	friend std::ostream& operator<<(std::ostream& os, Point& p);
	bool operator== (Point point);
	friend double dist(Point& p1, Point& p2);

private:
	double x_, y_;
};

#endif
```

tnarytree.cpp

```
#include "tnarytree.h"

template TNaryTree<Rectangle>;
template TNaryTree<Rhombus>;
template TNaryTree<Trapezoid>;

template <class T> TNaryTree<T>::TNaryTree()
{
	this->N = 2;
	root = std::make_shared<Node<T>>(Node<T>(T(), 0, nullptr, nullptr));
}
template <class T> TNaryTree<T>::TNaryTree(int N)
{
	this->N = N;
	root = std::make_shared<Node<T>>(Node<T>(T(), 0, nullptr, nullptr));
}

template <class T> TNaryTree<T>::TNaryTree(TNaryTree& other)
{
	N = other.N;
	if (other.Empty())
	{
		root = nullptr;
		return;
	}
```

```
	root = std::make_shared<Node<T>>(Node<T>(other.root->t, 0, nullptr, nullptr));
	BuildTree(root, other.root);
}

template <class T> void TNaryTree<T>::BuildTree(std::shared_ptr<Node<T>>& current_node,
std::shared_ptr<Node<T>> other_node)
{
	if (!other_node->child)
	{
		return;
	}
	current_node->child = std::make_shared<Node<T>>(Node<T>(other_node->child->t,
other_node->child->remainder, current_node, nullptr));
	std::shared_ptr<Node<T>>copy = current_node->child, other_copy = other_node->child;
	while (other_copy)
	{
		BuildTree(copy, other_copy);
		if (other_copy->right_brother)
		{
			copy->right_brother = std::make_shared<Node<T>>(Node<T>(other_copy-
>right_brother->t, other_copy->right_brother->remainder, current_node, copy));
		}
		else
		{
			copy->right_brother = nullptr;
		}
		copy = copy->right_brother;
		other_copy = other_copy->right_brother;
	}
}

template <class T> bool TNaryTree<T>::Empty()
{
	if (root)
	{
		return false;
	}
	return true;
}


template<class T> T TNaryTree<T>::getItem(std::string&& tree_path)
{
	try
	{
		if (!tree_path.length())
		{
			if (Empty())
			{
				throw std::invalid_argument("There's no root\n");
			}
			else
			{
				return root->t;
			}
		}
		std::shared_ptr<Node<T>> current_node = root;
		while (tree_path.length())
		{
```

```
                        switch (tree_path[0])
                        {
                        case 'b':
                        {
                                if (!current_node)
                                {
                                        throw std::invalid_argument("There's no such element in
tree\n");
                                }
                                current_node = current_node->right_brother;
                                break;
                        }
                        case 'c':
                        {
                                if (!current_node)
                                {
                                        throw std::invalid_argument("There's no such element in
tree\n");
                                }
                                current_node = current_node->child;
                                break;
                        }
                        default:
                        {
                                throw std::invalid_argument("String must contain only 'b' or
'c' characters\n");
                        }
                        }
                        tree_path.erase(tree_path.begin());
                }
                if (!current_node)
                {
                        throw std::invalid_argument("There's no such element in tree\n");
                }
                return current_node->t;
        }
        catch (std::invalid_argument& error)
        {
                std::cout << error.what();
                return T();
        }
        catch (std::out_of_range& error)
        {
                std::cout << error.what();
                return T();
        }
}

template <class T> void TNaryTree<T>::Update(T&& t, std::string&& tree_path)
{
        try
        {
                if (!tree_path.length())
                {
                        if (Empty())
                        {
                                root = std::make_shared<Node<T>>(Node<T>(t, 0, nullptr,
nullptr));
                        }
```

```
                else
                {
                        root->t = t;
                }
                return;
        }
        std::shared_ptr<Node<T>> current_node = root;
        while (tree_path.length() > 1)
        {
                switch (tree_path[0])
                {
                case 'b':
                {
                        if (!current_node)
                        {
                                throw std::invalid_argument("There's no such element in
tree\n");
                        }
                        current_node = current_node->right_brother;
                        break;
                }
                case 'c':
                {
                        if (!current_node)
                        {
                                throw std::invalid_argument("There's no such element in
tree\n");
                        }
                        current_node = current_node->child;
                        break;
                }
                default:
                {
                        throw std::invalid_argument("String must contain only 'b' or
'c' characters\n");
                }
                }
                tree_path.erase(tree_path.begin());
        }
        switch (tree_path[0])
        {
        case 'b':
        {
                if ((!current_node) || (!current_node->remainder))
                {
                        throw std::out_of_range("Node already has " + std::to_string(N)
+ " sons, so it's imposible to add another one\n");
                }
                if (!current_node->right_brother)
                {
                        current_node->right_brother =
std::make_shared<Node<T>>(Node<T>(t, current_node->remainder - 1, current_node->parent,
current_node));
                }
                else
                {
                        current_node->t = t;
                }
                break;
```

```
			}
			case 'c':
			{
				if (!current_node)
				{
					throw std::invalid_argument("There's no such element in
tree\n");
				}
				if (!current_node->child)
				{
					current_node->child = std::make_shared<Node<T>>(Node<T>(t, N -
1, current_node, nullptr));
				}
				else
				{
					current_node->child->t = t;
				}
				break;
			}
			default:
			{
				throw std::invalid_argument("String must contain only 'b' or 'c'
characters\n");
			}
			}
			tree_path.erase(tree_path.begin());
		}
		catch (std::invalid_argument& error)
		{
			std::cout << error.what();
			return;
		}
		catch (std::out_of_range& error)
		{
			std::cout << error.what();
			return;
		}
}

template <class T> void TNaryTree<T>::DeleteSons(std::shared_ptr<Node<T>>& node)
{
		std::shared_ptr<Node<T>> copy = node->child, previous = copy;
		while (copy)
		{
			if (copy->child)
			{
				DeleteSons(copy);
			}
			previous = copy;
			copy = copy->right_brother;
		}
		while (previous)
		{
			previous->right_brother = nullptr;
			previous = previous->left_brother;
		}
		node->child = nullptr;
}
```

```
template <class T> void TNaryTree<T>::RemoveSubTree(std::string&& tree_path)
{
	try
	{
		if (!tree_path.length())
		{
			if (Empty())
			{
				throw std::invalid_argument("The root is empty\n");
			}
			else
			{
				DeleteSons(root);
				root = nullptr;
				return;
			}
		}
		std::shared_ptr<Node<T>> current_node = root;
		while (tree_path.length())
		{
			switch (tree_path[0])
			{
			case 'b':
			{
				if (!current_node)
				{
					throw std::invalid_argument("There's no such element in
tree\n");
				}
				current_node = current_node->right_brother;
				break;
			}
			case 'c':
			{
				if (!current_node)
				{
					throw std::invalid_argument("There's no such element in
tree\n");
				}
				current_node = current_node->child;
				break;
			}
			default:
			{
				throw std::invalid_argument("String must contain only 'b' or
'c' characters\n");
			}
			}
			tree_path.erase(tree_path.begin());
		}
		if (!current_node)
		{
			throw std::invalid_argument("There's no such element in tree\n");
		}
		DeleteSons(current_node);
		std::shared_ptr<Node<T>> clone = current_node->right_brother;
		if (current_node->left_brother)
		{
			if (current_node->right_brother)
```

```
                        {
                                current_node->right_brother->left_brother = current_node-
>left_brother;
                        }
                        current_node->left_brother->right_brother = current_node-
>right_brother;
                }
                else
                {
                        current_node->parent->child = current_node->right_brother;
                }
                current_node = nullptr;
                while (clone)
                {
                        ++(clone->remainder);
                        clone = clone->right_brother;
                }
        }
        catch (std::invalid_argument& error)
        {
                std::cout << error.what();
                return;
        }
        catch (std::out_of_range& error)
        {
                std::cout << error.what();
                return;
        }
}

template <class T> double TNaryTree<T>::AreaOfSubtree(std::shared_ptr<Node<T>> node)
{
        double S = node->t.Square();
        std::shared_ptr<Node<T>> current_node = node->child;
        while (current_node)
        {
                S += AreaOfSubtree(current_node);
                current_node = current_node->right_brother;
        }
        return S;
}

template <class T> double TNaryTree<T>::Area(std::string&& tree_path)
{
        try
        {
                if (Empty())
                {
                        throw std::invalid_argument("The root is empty\n");
                }
                if (!tree_path.length())
                {
                        return AreaOfSubtree(root);
                }
                std::shared_ptr<Node<T>> current_node = root;
                while (tree_path.length())
                {
                        switch (tree_path[0])
                        {
```

```
                        case 'b':
                        {
                                if (!current_node)
                                {
                                        throw std::invalid_argument("There is no such element in
tree\n");
                                }
                                current_node = current_node->right_brother;
                                tree_path.erase(tree_path.begin());
                                break;
                        }
                        case 'c':
                        {
                                if (!current_node)
                                {
                                        throw std::invalid_argument("There is no such element in
tree\n");
                                }
                                current_node = current_node->child;
                                tree_path.erase(tree_path.begin());
                                break;
                        }
                        default:
                        {
                                throw std::invalid_argument("String must contain only 'b' or
'c' characters\n");
                        }
                        }
                        tree_path.erase(tree_path.begin());
                }
                if (!current_node)
                {
                        throw std::invalid_argument("There's no such element in tree\n");
                }
                return AreaOfSubtree(current_node);
        }
        catch (std::invalid_argument& error)
        {
                std::cout << error.what();
                return -1.;
        }
        catch (std::out_of_range& error)
        {
                std::cout << error.what();
                return -1.;
        }
}

template std::ostream& operator<<(std::ostream& os, TNaryTree<Rectangle>& tree);
template std::ostream& operator<<(std::ostream& os, TNaryTree<Rhombus>& tree);
template std::ostream& operator<<(std::ostream& os, TNaryTree<Trapezoid>& tree);

template <typename T> std::ostream& operator<<(std::ostream& os, TNaryTree<T>& tree)
{
        try
        {
                if (tree.Empty())
                {
                        throw std::invalid_argument("The root is empty");
```

```
                }
                std::shared_ptr<Node<T>> current_node = tree.root;
                tree.root->PrintSubTree(os);
        }
        catch (std::invalid_argument& error)
        {
                os << error.what();
        }
        os << "\n";
        return os;
};

template <class T> TNaryTree<T>::~TNaryTree()
{
        if (!Empty())
        {
                DeleteSons(root);
                root = nullptr;
        }
}
```

tnarytree.h

```
#ifndef TNARYTREE_H
#define TNARYTREE_H
#include"tnarytreeitem.h"
#include <exception>
#include <string>

template<class T>
class TNaryTree
{
private:
        /*
        struct Node
        {
                Node(T, int, std::shared_ptr<Node>, std::shared_ptr<Node>);
                int remainder;
                T t;
                std::shared_ptr<Node> parent;
                std::shared_ptr<Node> child;
                std::shared_ptr<Node> left_brother;
                std::shared_ptr<Node> right_brother;
                void PrintSubTree(std::ostream& os);
                ~Node();
        };*/
        std::shared_ptr<Node<T>> root;
        int N;
public:
        TNaryTree();
        TNaryTree(int);
        TNaryTree(TNaryTree<T>&);
        void BuildTree(std::shared_ptr<Node<T>>&, std::shared_ptr<Node<T>>);
        void Update(T&&, std::string && = "");
        void RemoveSubTree(std::string && = "");
```

```
        void DeleteSons(std::shared_ptr<Node<T>>&);
        T getItem(std::string && = "");
        bool Empty();
        double Area(std::string && = "");
        double AreaOfSubtree(std::shared_ptr<Node<T>>);
        template <typename A>
    friend std::ostream& operator<<(std::ostream&, TNaryTree<A>&);
        virtual ~TNaryTree();
};
#endif
```

tnarytreeitem.cpp

```
#include "tnarytreeitem.h"
template Node<Rectangle>;
template Node<Rhombus>;
template Node<Trapezoid>;

template <class T> Node<T>::Node(T t, int remainder, std::shared_ptr<Node<T>> parent,
std::shared_ptr<Node<T>> left_brother)
{
        this->t = t;
        this->remainder = remainder;
        this->parent = parent;
        this->child = child;
        this->left_brother = left_brother;
        this->right_brother = right_brother;
}

template <typename T> std::shared_ptr<Node<T>> Node<T>::getChild()
{
        return this->child;
}

template <typename T> std::shared_ptr<Node<T>> Node<T>::getBrother()
{
        return this->right_brother;
}

template <typename T> std::shared_ptr<Node<T>> Node<T>::getParent()
{
        return this->parent;
}

template <class T> void Node<T>::PrintSubTree(std::ostream& os)
{
        os << *this;
        if (!this->child)
        {
                return;
        }
        std::shared_ptr<Node<T>> current_node = this->getChild();
        os << ": [";
        while (current_node)
```

```
	{
		current_node->PrintSubTree(os);
		if (current_node->getBrother())
		{
			os << ", ";
		}
		current_node = current_node->getBrother();
	}
	os << "]";
}

template std::ostream& operator<<(std::ostream& os, const Node<Rectangle>& node);
template std::ostream& operator<<(std::ostream& os, const Node<Rhombus>& node);
template std::ostream& operator<<(std::ostream& os, const Node<Trapezoid>& node);

template <typename T> std::ostream& operator<< (std::ostream& os, const Node<T>& node)
{
	os << node.t;
	return os;
}

template<class T> Node<T>::~Node()
{

}
```

tnarytreeitem.h

```
#ifndef TNARY_TREE_ITEM_H
#define TNARY_TREE_ITEM_H

#include "rhombus.h"
#include "rectangle.h"
#include "trapezoid.h"

template <class T>
class Node
{
public:
	Node(T, int, std::shared_ptr<Node<T>>, std::shared_ptr<Node<T>>);
	int remainder;
	T t;
	std::shared_ptr<Node> parent;
	std::shared_ptr<Node> child;
	std::shared_ptr<Node> left_brother;
	std::shared_ptr<Node> right_brother;
	void PrintSubTree(std::ostream& os);
	template <typename A>
	friend std::ostream& operator<<(std::ostream&, const Node<A>&);
	virtual ~Node();
	std::shared_ptr<Node<T>> getChild();
	std::shared_ptr<Node<T>> getBrother();
	std::shared_ptr<Node<T>> getParent();
};
#endif
```