МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №8 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:

* Создание сложных динамических структур данных;
* Закрепление принципа OCP.

### Задание

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с ней. «Хранилище объектов» представляет собой контейнер, одного из следующих видов (Контейнер 1-го уровня):

1. Очередь
2. Динамический массив
3. Связный список
4. Бинарное дерево
5. N-Дерево (с ограничением не больше 4 элементов на одном уровне).

Каждым элемент контейнера, в свою очередь, является динамической структурой данных одного из следующих видов (Контейнер 2-го уровня):

1. Очередь
2. Динамический массив
3. Связный список
4. Бинарное дерево
5. N-Дерево (с ограничением не больше 4 элементов на одном уровне).

Таким образом у нас получается контейнер в контейнере. Т.е. для варианта (2,3) это будет массив, каждый из элементов которого – связанный список. А для варианта (1,4) – это очередь из бинарных деревьев. Элементом второго контейнера является объект-фигура, определенная вариантом задания.

При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше **5**. Если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня.

Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию **площади** объекта (в том числе и для деревьев). При удалении объектов должно выполняться правило, что контейнер второго уровня не должен быть пустым. Т.е. если контейнер становится пустым, то он должен удалится.

Нельзя использовать:

* Стандартные контейнеры std.

Программа должна позволять:

* Вводить произвольное количество фигур и добавлять их в контейнер.
* Распечатывать содержимое контейнера (1-го и 2-го уровня).
* Удалять фигуры из контейнера по критериям:
  + По типу (например, все квадраты).
  + По площади (например, все объекты с площадью меньше чем заданная).

**Вариант 21**:

А) Контейнер 1 уровня – N-арное дерево.

Б) Контейнер 2 уровня – Вектор.

В) Фигуры – Прямоугольник, Ромб и Трапеция.

**Дневник отладки**

Так как работа сложная, то делалась она долго. Возникал много помарок, теперь всё вроде работает корректно.

**Недочёты**  
Недочётов вроде нет.

**Выводы**

Лабораторная работа №8 – ООП внутри ООП. Элементами контейнера первого уровня являются контейнеры 2 уровня, элементами которого уже являются фигуры 3 типов. Но одна структура данных может содержать только фигуру одного типа. Эта была очень сложная работа, на которую было потрачено много времени.

**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;

};

#endif

main.cpp  
  
#include <iostream>

#include "tnarytree.h"

int main()

{

TNaryTree<Rectangle> t1(4);

std::cout << t1 << "\n";

t1.Update(Rectangle(std::cin), 0, "");

t1.Update(Rectangle(std::cin), 0);

std::cout << t1 << "\n";

t1.Update(Rectangle(std::cin), 2, "c");

t1.Update(Rectangle(std::cin), 0, "c");

t1.Update(Rectangle(std::cin), 1, "c");

t1.Update(Rectangle(std::cin), 2, "c");

std::cout << t1 << "\n";

t1.Update(Rectangle(std::cin), 4, "c");

t1.Update(Rectangle(std::cin), 3, "c");

std::cout << t1.getItem("c");

std::cout << t1 << "\n";

t1.Update(Rectangle(std::cin), 0, "cb");

t1.Update(Rectangle(std::cin), 1, "cb");

t1.Update(Rectangle(std::cin), 2, "cb");

t1.Update(Rectangle(std::cin), 3, "cb");

std::cout << t1 << "\n";

t1.Update(Rectangle(std::cin), 0, "cbc");

t1.Update(Rectangle(std::cin), 1, "cbc");

t1.Update(Rectangle(std::cin), 2, "cbc");

t1.Update(Rectangle(std::cin), 2, "cbc");

t1.Update(Rectangle(std::cin), 3, "cbc");

std::cout << t1 << "\n";

t1.Update(Rectangle(std::cin), 0, "cbb");

t1.Update(Rectangle(std::cin), 1, "cbb");

t1.Update(Rectangle(std::cin), 2, "cbb");

t1.Update(Rectangle(std::cin), 3, "cbb");

std::cout << t1 << "\n";

t1.Update(Rectangle(std::cin), 4, "cbb");

t1.Update(Rectangle(std::cin), 5, "cbb");

t1.Update(Rectangle(std::cin), 7, "cbb");

t1.Update(Rectangle(std::cin), 6, "cbb");

std::cout << t1 << "\n";

t1.Update(Rectangle(std::cin), 0, "cbbb");

t1.Remove(false, "cbb", 6);

t1.Remove(false, "cb", 2);

std::cout << t1 << "\n";

t1.Remove(false, "c", 2);

t1.Remove(false, "cbb", 3);

t1.Remove(false, "cb", 0);

std::cout << t1 << "\n";

t1.Remove(true, "cb");

std::cout << t1 << "\n";

t1.Remove(true);

std::cout << t1 << "\n";

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 (square != rectangle.Square())

{

return square < rectangle.Square();

}

if (a != rectangle.a)

{

return a < rectangle.a;

}

if (b != rectangle.b)

{

return b < rectangle.b;

}

if (c != rectangle.c)

{

return c < rectangle.c;

}

return d < rectangle.d;

};

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, const 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, const Rectangle& rectangle);

Rectangle& operator= (Rectangle rectangle);

bool 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 (square != rhombus.Square())

{

return square < rhombus.Square();

}

if (a != rhombus.a)

{

return a < rhombus.a;

}

if (b != rhombus.b)

{

return b < rhombus.b;

}

if (c != rhombus.c)

{

return c < rhombus.c;

}

return d < rhombus.d;

}

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, const 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, const Rhombus& rhombus);

Rhombus& operator= (Rhombus rhombus);

bool 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 (square != trapezoid.Square())

{

return square < trapezoid.Square();

}

if (a != trapezoid.a)

{

return a < trapezoid.a;

}

if (b != trapezoid.b)

{

return b < trapezoid.b;

}

if (c != trapezoid.c)

{

return c < trapezoid.c;

}

return d < trapezoid.d;

};

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, const 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, const Trapezoid& trapezoid);

Trapezoid& operator= (Trapezoid trapezoid);

bool 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, const Point& p)

{

os << "(" << p.x\_ << ", " << p.y\_ << ")";

return os;

}

bool Point::operator == (Point point)

{

return (x\_ == point.x\_) && (y\_ == point.y\_);

}

bool Point::operator!= (Point point)

{

return (x\_ != point.x\_) || (y\_ != point.y\_);

}

bool Point::operator< (Point point)

{

if (x\_ != point.x\_)

{

return x\_ < point.x\_;

}

return y\_ < point.y\_;

}

Point::~Point()

{

}

point.h

#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 Pentagon;

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"

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, const Point& p)

{

os << "(" << p.x\_ << ", " << p.y\_ << ")";

return os;

}

bool Point::operator == (Point point)

{

return (x\_ == point.x\_) && (y\_ == point.y\_);

}

bool Point::operator!= (Point point)

{

return (x\_ != point.x\_) || (y\_ != point.y\_);

}

bool Point::operator< (Point point)

{

if (x\_ != point.x\_)

{

return x\_ < point.x\_;

}

return y\_ < point.y\_;

}

Point::~Point()

{

}

tnarytree.cpp

#include "tnarytree.h"

template TNaryTree<Rectangle>;

template TNaryTree<Rhombus>;

template TNaryTree<Trapezoid>;

template <class T> TNaryTree<T>::TNaryTree()

{

this->N = 2;

this->amount = 1;

root = std::make\_shared<Node<T>>(Node<T>(TVector<T>(), 0, nullptr, nullptr));

}

template <class T> TNaryTree<T>::TNaryTree(int N)

{

try

{

if (N > 4)

{

throw std::invalid\_argument("Number of sons must be lesser than 5\n");

}

this->N = N;

this->amount = 1;

root = std::make\_shared<Node<T>>(Node<T>(TVector<T>(), 0, nullptr, nullptr));

}

catch (std::invalid\_argument& error)

{

std::cout << error.what();

return;

}

}

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->vector, 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->vector, 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->vector, 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> TVector<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->vector;

}

}

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->vector;

}

catch (std::invalid\_argument& error)

{

std::cout << error.what() << "Default vector will be displayed\n";

return TVector<T>();

}

catch (std::out\_of\_range& error)

{

std::cout << error.what() << "Default vector will be displayed\n";

return TVector<T>();

}

}

template <class T> void TNaryTree<T>::Update(T&& t, int index, std::string&& tree\_path)

{

try

{

if (index < 0)

{

throw std::invalid\_argument("Index must be whole non-negative number\n");

}

if (!tree\_path.length())

{

if (Empty())

{

root = std::make\_shared<Node<T>>(Node<T>(TVector<T>(t), 0, nullptr, nullptr));

++amount;

}

else

{

if (!root->vector.arr)

{

if (index)

{

throw std::out\_of\_range("There's no such element in vector\n");

}

else

{

root->vector = TVector<T>(t);

}

}

else

{

if (index < root->vector.real\_size)

{

root->vector.Update(t, index);

}

else if ((!root->vector.real\_size) && (!index))

{

root->vector.Update(t, index);

}

else if (index == root->vector.real\_size)

{

root->vector.AppendElement(t);

}

else

{

throw std::out\_of\_range("There's no such element in vector\n");

}

}

}

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)

{

if ((++amount) == 6)

{

--amount;

throw std::invalid\_argument("Number of elements in tree must be lesser than 5\n");

}

if (index)

{

--amount;

throw std::out\_of\_range("There's no such element in vector\n");

}

current\_node->right\_brother = std::make\_shared<Node<T>>(Node<T>(TVector<T>(t), current\_node->remainder - 1, current\_node->parent, current\_node));

}

else

{

if (!current\_node->right\_brother->vector.arr)

{

if (index)

{

throw std::out\_of\_range("There's no such element in vector\n");

}

else

{

current\_node->right\_brother->vector = TVector<T>(t);

}

}

else

{

if (index < current\_node->right\_brother->vector.real\_size)

{

current\_node->right\_brother->vector.Update(t, index);

}

else if (index == current\_node->right\_brother->vector.real\_size)

{

current\_node->right\_brother->vector.AppendElement(t);

}

else

{

throw std::out\_of\_range("There's no such element in vector\n");

}

}

}

break;

}

case 'c':

{

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

if (!current\_node->child)

{

if ((++amount) == 5)

{

throw std::invalid\_argument("Number of elements in tree must be lesser than 5\n");

}

if (index)

{

--amount;

throw std::out\_of\_range("There's no such element in vector\n");

}

current\_node->child = std::make\_shared<Node<T>>(Node<T>(TVector<T>(t), N - 1, current\_node, nullptr));

}

else

{

if (!current\_node->child->vector.arr)

{

if (index)

{

throw std::out\_of\_range("There's no such element in vector\n");

}

else

{

current\_node->child->vector = TVector<T>(t);

}

}

else

{

if (index < current\_node->child->vector.real\_size)

{

current\_node->child->vector.Update(t, index);

}

else if (index == current\_node->child->vector.real\_size)

{

current\_node->child->vector.AppendElement(t);

}

else

{

throw std::out\_of\_range("There's no such element in vector\n");

}

}

}

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)

{

if ((previous->right\_brother) && (previous->right\_brother->vector.arr))

{

previous->right\_brother->vector.Delete();

}

--amount;

previous->right\_brother = nullptr;

previous = previous->left\_brother;

}

node->child->vector.Delete();

--amount;

node->child = nullptr;

}

template <class T> void TNaryTree<T>::Remove(bool v, std::string&& tree\_path, int index)

{

try

{

if (index < 0)

{

throw std::invalid\_argument("Index must be whole non-negative number\n");

}

if (!tree\_path.length())

{

if (Empty())

{

throw std::invalid\_argument("The root is empty\n");

}

else

{

if (v)

{

DeleteSons(root);

root->vector.Delete();

--amount;

root = nullptr;

return;

}

if (index >= root->vector.real\_size)

{

throw std::invalid\_argument("There's no such element in vector\n");

}

else

{

root->vector.DeleteElement(index);

}

}

}

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");

}

if ((v) || (current\_node->vector.real\_size == 1))

{

if (current\_node->vector.real\_size == 1)

{

current\_node->vector.Delete();

}

DeleteSons(current\_node);

std::shared\_ptr<Node<T>> clone = current\_node->right\_brother;

current\_node->vector.Delete();

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;

}

}

else

{

if (index >= current\_node->vector.real\_size)

{

throw std::out\_of\_range("There's no such element in vector\n");

}

else

{

current\_node->vector.DeleteElement(index);

}

}

}

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();

}

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:

std::shared\_ptr<Node<T>> root;

int N;

int amount;

public:

TNaryTree();

TNaryTree(int);

TNaryTree(TNaryTree<T>&);

void BuildTree(std::shared\_ptr<Node<T>>&, std::shared\_ptr<Node<T>>);

void Update(T&&, int, std::string && = "");

void Remove(bool, std::string && = "", int index = 0);

void DeleteSons(std::shared\_ptr<Node<T>>&);

TVector<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)

{

vector = TVector<T>(t);

this->remainder = remainder;

this->parent = parent;

this->child = child;

this->left\_brother = left\_brother;

this->right\_brother = right\_brother;

}

template <class T> Node<T>::Node(TVector<T> tvector, int remainder, std::shared\_ptr<Node<T>> parent, std::shared\_ptr<Node<T>> left\_brother)

{

vector = tvector;

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.vector;

return os;

}

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

{

}

tnarytreeitem.h  
  
#ifndef TNARY\_TREE\_ITEM\_H

#define TNARY\_TREE\_ITEM\_H

#include "tvector.h"

template <class T>

class Node

{

public:

Node(T, int, std::shared\_ptr<Node<T>>, std::shared\_ptr<Node<T>>);

Node(TVector<T>, int, std::shared\_ptr<Node<T>>, std::shared\_ptr<Node<T>>);

TVector<T> vector;

int remainder;

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  
  
tvector.cpp  
  
#include "tvector.h"

template TVector<Rectangle>;

template TVector<Rhombus>;

template TVector<Trapezoid>;

template <class T> TVector<T>::TVector()

{

real\_size = 0;

size\_of\_memory = 1;

arr = new T[size\_of\_memory];

}

template<class T> TVector<T>::TVector(T t)

{

real\_size = size\_of\_memory = 1;

arr = new T[size\_of\_memory];

arr[0] = t;

}

template <class T> void TVector<T>::Realloc(bool increase)

{

double value;

if (increase)

{

value = 2;

}

else

{

value = 0.5;

}

T\* aux\_arr = new T[int(size\_of\_memory \* value)];

for (int i = 0; i < real\_size; i++)

{

aux\_arr[i] = this->arr[i];

}

delete[] arr;

this->arr = new T[int(size\_of\_memory \* value)];

for (int i = 0; i < real\_size; i++)

{

this->arr[i] = aux\_arr[i];

}

size\_of\_memory = int(size\_of\_memory \* value);

delete[] aux\_arr;

}

template <class T> void TVector<T>::Move(int index, bool forward, int stop, bool updating, T t)

{

if (forward)

{

if (real\_size == size\_of\_memory)

{

Realloc(forward);

}

for (int i = stop; i > index; i--)

{

arr[i] = arr[i - 1];

}

if (!updating)

{

++real\_size;

}

arr[index] = t;

}

else

{

for (int i = index; i < stop - 1; i++)

{

arr[i] = arr[i + 1];

}

if (!updating)

{

--real\_size;

}

if ((real\_size \* 2) == size\_of\_memory)

{

Realloc(forward);

}

}

}

template <class T> int TVector<T>::LowerBound(T t)

{

int l = 0, r = real\_size, m;

while (l < r)

{

m = l + (r - l) / 2;

if (!(arr[m] < t))

{

r = m;

}

else

{

l = m + 1;

}

}

return l;

}

template <class T> void TVector<T>::AppendElement(T t)

{

if (!real\_size)

{

if (!size\_of\_memory)

{

arr = new T[++size\_of\_memory];

}

arr[0] = t;

++real\_size;

}

else

{

//double square = t.Square();

int index = LowerBound(t);

Move(index, true, real\_size, false, t);

}

}

/\*

template <class T> int TVector<T>::BinarySearch(T t)

{

int l = -1, r = real\_size, m

while (l < r - 1)

{

m = (l + r) / 2;

if (arr[m] < t)

{

l = m;

}

else

{

r = m;

}

}

return ((r < real\_size) && (arr[r] == t)) ? r : -1;

}

\*/

template <class T> void TVector<T>::DeleteElement(int index)

{

if (index >= real\_size)

{

std::cout << "There's no such element in vector\n";

return;

}

if (real\_size == 1)

{

real\_size = size\_of\_memory = 0;

delete[] arr;

arr = nullptr;

return;

}

Move(index, false, real\_size, false);

}

template <class T> void TVector<T>::Delete()

{

if (arr)

{

delete[] arr;

arr = nullptr;

}

real\_size = size\_of\_memory = 0;

}

template <class T> void TVector<T>::Update(T t, int index)

{

if ((!index) && (!real\_size))

{

++real\_size;

arr[index] = t;

return;

}

if (index >= real\_size)

{

std::cout << "There's no such element in vector\n";

return;

}

else

{

int new\_index = LowerBound(t);

if ((new\_index >= index) && (index + 1 >= new\_index))

{

arr[index] = t;

return;

}

else if (new\_index < index)

{

Move(new\_index, true, index, true, t);

}

else

{

Move(index, false, new\_index, true);

arr[new\_index - 1] = t;

}

}

}

template <typename T> void TVector<T>::operator= (const TVector<T>& tvector)

{

if (arr)

{

delete[] arr;

}

real\_size = tvector.real\_size;

size\_of\_memory = tvector.size\_of\_memory;

arr = new T[size\_of\_memory];

for (int i = 0; i < real\_size; i++)

{

arr[i] = tvector.arr[i];

}

}

template std::ostream& operator<<(std::ostream& os, const TVector<Rectangle>& tvector);

template std::ostream& operator<<(std::ostream& os, const TVector<Rhombus>& tvector);

template std::ostream& operator<<(std::ostream& os, const TVector<Trapezoid>& tvector);

template <typename T> std::ostream& operator<< (std::ostream& os, const TVector<T>& tvector)

{

os << "{";

for (int i = 0; i < tvector.real\_size - 1; i++)

{

os << tvector.arr[i] << "; ";

}

if (tvector.real\_size)

{

os << tvector.arr[tvector.real\_size - 1];

}

os << "}";

return os;

}

template <class T> TVector<T>::~TVector()

{

}

tvector.h  
  
#ifndef TVECTOR\_H

#define TVECTOR\_H

#include "rhombus.h"

#include "rectangle.h"

#include "trapezoid.h"

template <class T>

class TVector

{

public:

TVector();

TVector(T);

int real\_size, size\_of\_memory;

T\* arr = nullptr;

void AppendElement(T);

//int BinarySearch(T);

void Delete();

void DeleteElement(int);

int LowerBound(T);

void Move(int, bool, int, bool, T = T());

void Realloc(bool);

void Update(T, int);

void operator= (const TVector<T>&);

template <typename A>

friend std::ostream& operator<<(std::ostream&, const TVector<A>&);

~TVector();

};

#endif