|  |  |
| --- | --- |
| Российский университет транспорта (МИИТ)  Институт транспортной техники и систем управления  Кафедра «Управление и защита информации» | |
| Курсовая работа  по теме «Разработка структуры данных»  по дисциплине «Системы управления базами данных и основы построения защищенных баз данных» | |
|  | Выполнил:  Студент группы ТКИ-441 Ковров А.И.  Проверил:  к.т.н. доц. Васильева М.А. |
| Москва 2024 | |

**Оглавление**

[Задание на работу 3](#__RefHeading___Toc7844_1508962542)

[UML-диаграмма классов приложения 5](#__RefHeading___Toc7846_1508962542)

[Проверка утечек памяти 6](#__RefHeading___Toc7846_1508962542_%25D0%)

[Результаты работы приложения 7](#__RefHeading___Toc7848_1508962542)

[Заключение 9](#__RefHeading___Toc7850_1508962542)

[Приложение A 10](#__RefHeading___Toc7852_1508962542)

# Задание на работу

1. Для заданной структуры данных разработать API (программный интерфейс приложения, интерфейс прикладного программирования) (англ. application programming interface) — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой. На языке C++ написать конструктор и деструктор класса разрабатываемой структуры для типа int. Отладить программу.
2. Разработать метод вывода структуры в строку. Отладить приложение. Все изменения выложить в GitHub. Завести новый Pull request. Провести рефакторинг кода в соответствии с Issue.
3. Разработать все необходимые для заданной структуры методы для хранения и поиска (если задано) элемента типа int. Отладить приложение. Все изменения выложить в GitHub. Завести новый Pull request. Провести рефакторинг кода в соответствии с Issue.
4. Структура данных для любого типа данных, реализующая CRUD — акроним, обозначающий четыре базовые функции, используемые при работе с базами данных: создание (англ. create), чтение (read), модификация (update), удаление (delete). Отладить приложение. Все изменения выложить в GitHub. Завести новый Pull request. Провести рефакторинг кода в соответствии с Issue.
5. Переопределение оператора сдвига. Переопределить операторы сдвига для реализации удобного взаимодействия с потоком ввода/вывода. Реализовать (не в библиотеке классов) методы ввода/ вывода структуры из/в консоль и файл, используя операторы сдвига. Отладить приложение. Все изменения выложить в GitHub. Завести новый Pull request. Провести рефакторинг кода в соответствии с Issue.
6. Тесты. Разработать тесты на конструктор и деструктор. Тесты положить в отдельную папку с названием Tests. Отладить приложение. Все изменения выложить в GitHub. Завести новый Pull request. Провести рефакторинг кода в соответствии с Issue.
7. Полное тестирование приложения. Разработать тесты на все публичные методы разрабатываемой структуры данных. Отладить приложение. Все изменения выложить в GitHub. Завести новый Pull request. Провести рефакторинг кода в соответствии с Issue.
8. Отчет по лабораторной работе "Разработка структуры данных" Разработать отчет по лабораторной работе по ГОСТ НИР 2017 (http://docs.cntd.ru/document/1200157208). Отчет должен содержать:
9. Задание на работу
10. UML-диаграмму классов приложения
11. Листинг готового приложения в текстовом формате. Для текста кода использовать шрифт "Courier New" или "Consolas" размером 11 пт с однострочным интервалом. Программу перед этим отформатировать.
12. Результаты работы приложения в виде снимков экрана. Все рисунки должны быть крупными и четкими, иметь подписи в соответствии с ГОСТ НИР 2017. Цвет фона рисунка должен быть белым, цвет шрифта ‑ черным.

# UML-диаграмма классов приложения

Диаграмма классов представлена на рисунке 1.



Рисунок 1 – Диаграмма классов

# Проверка утечек памяти

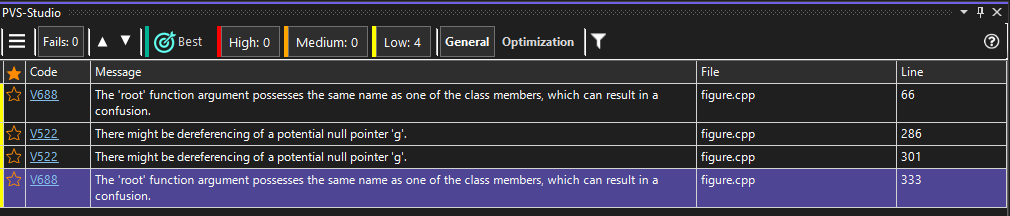


Рисунок 2 – Вывод PVS-Studio

# Результаты работы приложения

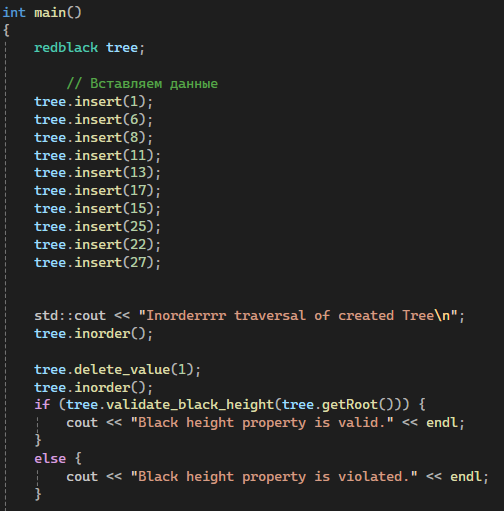
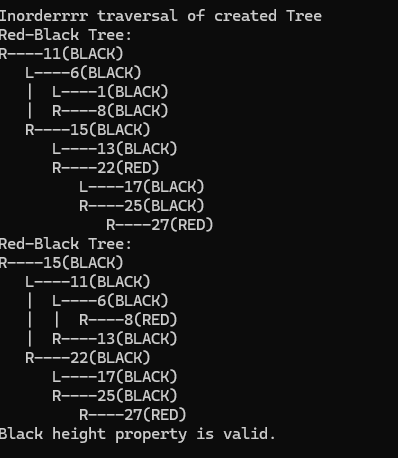


Рисунок 3 – Команды для выполнения

Рисунок 4 – Вывод программы

# Заключение

В ходе курсовой работы была разработана структура данных «Красно-черное дерево» в парадигме ООП на языке С++. Были изучены основы СУБД. В рамках работы также было изучено взаимодействие с тестами в Visual Studio и работа c Git Bash.

# Приложение A

**Листинг приложения**

root.h

#pragma once

struct Node {

int data;

int color; // RED - 0 | BLACK - 1

Node\* left, \* right, \* parent;

Node(int data) : data(data), color(0), left(nullptr), right(nullptr), parent(nullptr) {}

};

main.cpp

// This is a personal academic project. Dear PVS-Studio, please check it.

// PVS-Studio Static Code Analyzer for C, C++, C#, and Java: https://pvs-studio.com

#include <iostream>

#include "../triangle/figure.h"

/\*\*

\* @brief Point of entering the programm

\* @return 0 in case of unluck

\*/

int main()

{

redblack tree;

// Вставляем данные

tree.insert(1);

tree.insert(6);

tree.insert(8);

tree.insert(11);

tree.insert(13);

tree.insert(17);

tree.insert(15);

tree.insert(25);

tree.insert(22);

tree.insert(27);

std::cout << "Inorderrrr traversal of created Tree\n";

tree.inorder();

tree.delete\_value(1);

tree.inorder();

if (tree.validate\_black\_height(tree.getRoot())) {

cout << "Black height property is valid." << endl;

}

else {

cout << "Black height property is violated." << endl;

}

return 0;

}

figure.cpp

// This is a personal academic project. Dear PVS-Studio, please check it.

// PVS-Studio Static Code Analyzer for C, C++, C#, and Java: https://pvs-studio.com

#include <stdexcept>

#include <cmath>

#include "figure.h"

#include <sstream>

#include <iostream>

using namespace std;

/\*

\*@brief конструктор обычный

\*/

redblack::redblack()

{

root = nullptr;

}

/\*

\*@brief деструктор

\*/

redblack::~redblack()

{

delete\_tree(root);

}

/\*

\*@brief подсчет колличества элементов дерева

\* @return колличество элементов

\*/

int redblack::count\_nodes() const

{

return count\_nodes\_helper(root);

}

redblack& redblack::operator=(redblack&& other) noexcept

{

if (this != &other) {

delete\_tree(root);

root = other.root;

other.root = nullptr;

}

return \*this;

}

redblack& redblack::operator=(const redblack& other)

{

if (this != &other) {

delete\_tree(root);

root = copy\_tree(other.root, nullptr);

}

return \*this;

}

bool redblack::operator==(const redblack& other) const

{

return compare\_nodes(root, other.root);

}

bool redblack::validate\_black\_height(Node\* root)

{

int expected\_black\_count = -1;

return check\_black\_height(root, 0, expected\_black\_count);

}

bool redblack::compare\_nodes(const Node\* a, const Node\* b) const

{

if (a == nullptr && b == nullptr) {

return true;

}

if (a == nullptr || b == nullptr) {

return false;

}

return (a->data == b->data) && (a->color == b->color) && compare\_nodes(a->left, b->left) && compare\_nodes(a->right, b->right);

}

Node\* redblack::getRoot()

{

return root;

}

/\*

\*@brief функция для вставки новых корней дерева

\*/

void redblack::insert(int value) {

Node\* newNode = new Node(value);

if (root == nullptr) {

root = newNode;

insert\_case1(newNode);

return;

}

Node\* current = root;

Node\* parent = nullptr;

while (current != nullptr) {

parent = current;

if (value == current->data) {

cout << "Duplicate value: " << value << endl;

delete newNode;

return;

}

else if (value < current->data) {

current = current->left;

}

else {

current = current->right;

}

}

newNode->parent = parent;

if (value < parent->data) {

parent->left = newNode;

}

else {

parent->right = newNode;

}

insert\_case1(newNode);

}

/\*

\*@brief функция для удаления корня дерева

\*/

void redblack::delete\_value(int value)

{

Node\* nodeToDelete = root;

while (nodeToDelete != nullptr && nodeToDelete->data != value) {

if (value < nodeToDelete->data) {

nodeToDelete = nodeToDelete->left;

}

else {

nodeToDelete = nodeToDelete->right;

}

}

if (nodeToDelete == nullptr) {

cout << "Value " << value << " not found in the tree." << endl;

return;

}

if (!is\_leaf(nodeToDelete->left) && !is\_leaf(nodeToDelete->right))

{

Node\* successor = nodeToDelete->right;

while (successor->left != nullptr)

{

successor = successor->left;

}

nodeToDelete->data = successor->data;

nodeToDelete = successor;

}

delete\_one\_child(nodeToDelete);

}

/\*

\*@brief функция для поиска деда корня (корень->parent->parent)

\*@return Node \* n->parent->parent

\*/

Node\* redblack::grandparent(Node\* n) {

if ((n != nullptr) && (n->parent != nullptr)) {

return n->parent->parent;

}

return nullptr;

}

/\*

\*@brief поиск дяди корня (брат отца)

\* @return Node \* n->parent->other\_child

\*/

Node\* redblack::uncle(Node\* n) {

Node\* g = grandparent(n);

if (g == nullptr) return nullptr;

if (n->parent == g->left) {

return g->right;

}

return g->left;

}

/\*

\*@brief поворот дерева относительно узла n против часовой стрелки (налево)

\*/

void redblack::rotate\_left(Node\* n) {

Node\* pivot = n->right;

pivot->parent = n->parent;

if (n->parent == nullptr) {

root = pivot;

}

else if (n->parent->left == n) {

n->parent->left = pivot;

}

else {

n->parent->right = pivot;

}

n->right = pivot->left;

if (pivot->left != nullptr) {

pivot->left->parent = n;

}

n->parent = pivot;

pivot->left = n;

}

/\*

\*@brief поворот дерева относительно узла n по часовой стрелке (направо)

\*/

void redblack::rotate\_right(Node\* n) {

Node\* pivot = n->left;

pivot->parent = n->parent;

if (n->parent == nullptr) {

root = pivot;

}

else if (n->parent->left == n) {

n->parent->left = pivot;

}

else {

n->parent->right = pivot;

}

n->left = pivot->right;

if (pivot->right != nullptr) {

pivot->right->parent = n;

}

n->parent = pivot;

pivot->right = n;

}

/\*

\*@brief вспомогательная функция для удаления дерева

\*/

void redblack::delete\_tree(Node\* node)

{

if (node != nullptr) {

delete\_tree(node->left);

delete\_tree(node->right);

delete node;

}

}

/\*

\*@brief балансировка дерева после добавления нового узла случай 1 (теория для этого взята из википедии)

\*/

void redblack::insert\_case1(Node\* n) {

if (n->parent == nullptr) {

n->color = 1; // Корень всегда черный

root = n;

}

else {

insert\_case2(n);

}

}

/\*

\*@brief балансировка дерева после добавления нового узла случай 2 (теория для этого взята из википедии)

\*/

void redblack::insert\_case2(Node\* n) {

if (n->parent->color == 1) return; // Дерево корректно

insert\_case3(n);

}

/\*

\*@brief балансировка дерева после добавления нового узла случай 3 (теория для этого взята из википедии)

\*/

void redblack::insert\_case3(Node\* n) {

Node\* u = uncle(n);

Node\* g;

if ((u != nullptr) && (u->color == 0)) {

n->parent->color = 1;

u->color = 1;

g = grandparent(n);

g->color = 0;

insert\_case1(g);

}

else {

insert\_case4(n);

}

}

/\*

\*@brief балансировка дерева после добавления нового узла случай 4 (теория для этого взята из википедии)

\*/

void redblack::insert\_case4(Node\* n) {

Node\* g = grandparent(n);

if ((n == n->parent->right) && (n->parent == g->left)) {

rotate\_left(n->parent);

n = n->left;

}

else if ((n == n->parent->left) && (n->parent == g->right)) {

rotate\_right(n->parent);

n = n->right;

}

insert\_case5(n);

}

/\*

\*@brief балансировка дерева после добавления нового узла случай 5 (теория для этого взята из википедии)

\*/

void redblack::insert\_case5(Node\* n) {

Node\* g = grandparent(n);

if (g == nullptr) return;

n->parent->color = 1;

g->color = 0;

if ((n == n->parent->left) && (n->parent == g->left)) {

rotate\_right(g);

}

else {

rotate\_left(g);

}

}

/\*

\*@brief вспомогательная функция для красивого вывода целого дерева

\*/

void redblack::inorderHelper(Node\* root, string indent, bool last) {

if (root != nullptr) {

cout << indent;

if (last) {

cout << "R----";

indent += " ";

}

else {

cout << "L----";

indent += "| ";

}

string sColor = (root->color == 0) ? "RED" : "BLACK";

cout << root->data << "(" << sColor << ")" << endl;

inorderHelper(root->left, indent, false);

inorderHelper(root->right, indent, true);

}

}

/\*

\*@brief функция для поиска брата узла

\* @return Node\* брат узла

\*/

Node\* redblack::sibling(Node\* n)

{

if (n == n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

void redblack::replace\_node(Node\* n, Node\* child)

{

if (n->parent == nullptr) {

root = child;

}

else if (n == n->parent->left) {

n->parent->left = child;

}

else {

n->parent->right = child;

}

if (child != nullptr) {

child->parent = n->parent;

}

}

void redblack::delete\_one\_child(Node\* n)

{

Node\* child = (n->right != nullptr) ? n->right : n->left;

if (child != nullptr) {

replace\_node(n, child);

if (n->color == 1) {

if (child->color == 0) {

child->color = 1;

}

else {

delete\_case1(child);

}

}

}

else if (n->parent == nullptr) {

root = nullptr;

}

else {

if (n->color == 1) {

delete\_case1(n);

}

replace\_node(n, nullptr);

}

delete n;

}

/\*

\*@brief балансировка дерева после удалении узла случай 1 (теория для этого взята из википедии)

\*/

void redblack::delete\_case1(Node\* n)

{

if (n->parent != nullptr)

delete\_case2(n);

}

/\*

\*@brief балансировка дерева после удалении узла случай 2 (теория для этого взята из википедии)

\*/

void redblack::delete\_case2(Node\* n)

{

Node\* s = sibling(n);

if (s->color == 0) {

n->parent->color = 0;

s->color = 1;

if (n == n->parent->left)

rotate\_left(n->parent);

else

rotate\_right(n->parent);

}

delete\_case3(n);

}

/\*

\*@brief балансировка дерева после удалении узла случай 3 (теория для этого взята из википедии)

\*/

void redblack::delete\_case3(Node\* n)

{

Node\* s = sibling(n);

if (s == nullptr) return;

if ((n->parent->color == 1) && (s->color == 1) && (s->left == nullptr || s->left->color == 1) && (s->right == nullptr || s->right->color == 1)) { s->color = 0;

delete\_case1(n->parent);

}

else {

delete\_case4(n);

}

}

/\*

\*@brief балансировка дерева после удалении узла случай 4 (теория для этого взята из википедии)

\*/

void redblack::delete\_case4(Node\* n)

{

Node\* s = sibling(n);

if ((n->parent->color == 0) && (s->color == 1) && (s->left->color == 1) && (s->right->color == 1))

{

s->color = 0;

n->parent->color = 1;

}

else

delete\_case5(n);

}

/\*

\*@brief балансировка дерева после удалении узла случай 5 (теория для этого взята из википедии)

\*/

void redblack::delete\_case5(Node\* n)

{

Node\* s = sibling(n);

if (s->color == 1) {

if ((n == n->parent->left) && (s->right->color == 1) && (s->left->color == 0))

{

s->color = 0;

s->left->color = 1;

rotate\_right(s);

}

else if ((n == n->parent->right) && (s->left->color == 1) && (s->right->color == 0))

{

s->color = 0;

s->right->color = 1;

rotate\_left(s);

}

}

delete\_case6(n);

}

/\*

\*@brief балансировка дерева после удалении узла случай 6 (теория для этого взята из википедии)

\*/

void redblack::delete\_case6(Node\* n)

{

Node\* s = sibling(n);

s->color = n->parent->color;

n->parent->color = 1;

if (n == n->parent->left) {

s->right->color = 1;

rotate\_left(n->parent);

}

else {

s->left->color = 1;

rotate\_right(n->parent);

}

}

/\*

\*@brief проверка на то, вляется ли узел листом (последним в ветви)

\*/

bool redblack::is\_leaf(Node\* n)

{

return n == nullptr || (n->left == nullptr && n->right == nullptr);

}

/\*

\*@brief вывод дерева в консоль

\*/

void redblack::inorder() {

if (root == nullptr)

cout << "Tree is empty." << endl;

else {

cout << "Red-Black Tree:" << endl;

inorderHelper(root, "", true);

}

}

/\*

\*@brief Конструктор с инициализатором списка

\*/

redblack::redblack(const initializer\_list<int>& values) : root(nullptr) {

for (int value : values) {

insert(value);

}

}

/\*

\*@brief Конструктор копирования

\*/

redblack::redblack(const redblack& other) : root(nullptr) {

root = copy\_tree(other.root, nullptr);

}

/\*

\*@brief Вспомогательная функция для копирования дерева

\*/

Node\* redblack::copy\_tree(Node\* other\_root, Node\* parent) {

if (!other\_root) return nullptr;

Node\* new\_node = new Node(other\_root->data);

new\_node->color = other\_root->color;

new\_node->parent = parent;

new\_node->left = copy\_tree(other\_root->left, new\_node);

new\_node->right = copy\_tree(other\_root->right, new\_node);

return new\_node;

}

/\*

\*@brief Вспомогательная функция для подсчета узлов дерева

\*/

int redblack::count\_nodes\_helper(Node\* node) const

{

if (node == nullptr) {

return 0;

}

return 1 + count\_nodes\_helper(node->left) + count\_nodes\_helper(node->right);

}

bool redblack::check\_black\_height(Node\* node, int current\_black\_count, int& expected\_black\_count)

{

if (node == nullptr) {

if (expected\_black\_count == -1) {

expected\_black\_count = current\_black\_count;

return true;

}

return current\_black\_count == expected\_black\_count;

}

if (node->color == 1) {

current\_black\_count++;

}

bool left\_check = check\_black\_height(node->left, current\_black\_count, expected\_black\_count);

bool right\_check = check\_black\_height(node->right, current\_black\_count, expected\_black\_count);

return left\_check && right\_check;

}

/\*

\*@brief Конструктор перемещения

\*/

redblack::redblack(redblack&& other) noexcept : root(other.root) {

other.root = nullptr;

}

figure.h

// This is a personal academic project. Dear PVS-Studio, please check it.

// PVS-Studio Static Code Analyzer for C, C++, C#, and Java: https://pvs-studio.com

#pragma once

#include "../struckk/rooot.h"

using namespace std;

/\*

\* @brief Class Figure

\*/

class redblack

{

private:

Node\* root;

/\*

\*@brief функция для поиска деда корня (корень->parent->parent)

\*@return Node \* n->parent->parent

\*/

Node\* grandparent(Node\* n);

/\*

\*@brief поиск дяди корня (брат отца)

\* @return Node \* n->parent->other\_child

\*/

Node\* uncle(Node\* n);

/\*

\*@brief поворот дерева относительно узла n против часовой стрелки (налево)

\*/

void rotate\_left(Node\* n);

/\*

\*@brief поворот дерева относительно узла n по часовой стрелке (направо)

\*/

void rotate\_right(Node\* n);

/\*

\*@brief вспомогательная функция для удаления дерева

\*/

void delete\_tree(Node\* node);

/\*

\* @brief Вставка, случай 1 из википедии: N - корень

\* @param n Вставляемый узел

\*/

void insert\_case1(Node\* n);

/\*

\* @brief Вставка, случай 2 из википедии: P - чёрный

\* @param n Вставляемый узел

\*/

void insert\_case2(Node\* n);

/\*

\* @brief Вставка, случай 3 из википедии: U - красный

\* @param n Вставляемый узел

\*/

void insert\_case3(Node\* n);

/\*

\* @brief Вставка, случай 4 из википедии: N - правый потомок P, P - левый потомок G

\* @param n Вставляемый узел

\*/

void insert\_case4(Node\* n);

/\*

\* @brief Вставка, случай 5 из википедии: N - левый потомок P, P - левый потомок G

\* @param n Вставляемый узел

\*/

void insert\_case5(Node\* n);

/\*

\*@brief вспомогательная функция для красивого вывода целого дерева

\*@param root - корень дерева, indent - строка для вывода, last - для прехеода налево или направо

\*/

void inorderHelper(Node\* root, string indent, bool last);

/\*

\*@brief функция для поиска брата узла

\* @return Node\* брат узла

\*/

Node\* sibling(Node\* n);

void replace\_node(Node\* n, Node\* child);

/\*

\* @brief Удаляет одного потомка узла

\* @param n Узел, у которого необходимо удалить потомка

\*/

void delete\_one\_child(Node\* n);

/\*

\* @brief Удаление, случай 1 из википедии: N - новый корень

\* @param n Удаляемый узел

\*/

void delete\_case1(Node\* n);

/\*

\* @brief Удаление, случай 2 из википедии: S - красный

\* @param n Удаляемый узел

\*/

void delete\_case2(Node\* n);

/\*

\* @brief Удаление, случай 3 из википедии: P, S и дети S - черные

\* @param n Удаляемый узел

\*/

void delete\_case3(Node\* n);

/\*

\* @brief Удаление, случай 4 из википедии: S и его дети черные, но P - красный

\* @param n Удаляемый узел

\*/

void delete\_case4(Node\* n);

/\*

\* @brief Удаление, случай 5 из википедии: S - черный, левый потомок S - красный, правый потомок S - черный, N является левым потомком своего отца

\* @param n Удаляемый узел

\*/

void delete\_case5(Node\* n);

/\*

\* @brief Удаление, случай 6 из википедии: S - черный, правый потомок S - красный, N является левым потомком своего отца

\* @param n Удаляемый узел

\*/

void delete\_case6(Node\* n);

/\*

\*@brief проверка на то, вляется ли узел листом (последним в ветви)

\* @return true or false

\*/

bool is\_leaf(Node\* n);

/\*

\* @brief Вспомогательная функция для копирования дерева

\* @return копия узла

\*/

Node\* copy\_tree(Node\* other\_root, Node\* parent);

/\*

\* @brief вспомогательная функция для подсчета узлов дерева

\* @param текущий узел

\* @return результат сравнения

\*/

int count\_nodes\_helper(Node\* node) const;

/\*

\* @brief вспомогательная функция для подсчета черных корней

\* @param текущий узел счетчик и текущий максимум

\* @return true or false

\*/

bool check\_black\_height(Node\* node, int current\_black\_count, int& expected\_black\_count);

public:

/\*

\*@brief вывод дерева в консоль

\*/

void inorder();

/\*

\*@brief функция для вставки новых корней дерева

\*/

void insert(int value);

/\*

\*@brief функция для удаления узла дерева

\*/

void delete\_value(int value);

/\*

\*@brief конструктор обычный

\*/

redblack();

/\*

\*@brief конструктор с инициализатором списка

\*/

redblack(const initializer\_list<int>& values);

/\*

\*@brief конструктор копирования

\*/

redblack(const redblack& other);

/\*

\*@brief конструктор перемещения

\*/

redblack(redblack&& other) noexcept; // Конструктор перемещения

/\*

\*@brief деструктор

\*/

~redblack();

/\*

\*@brief подсчет колличества элементов дерева

\*@return колличество элементов

\*/

int count\_nodes() const;

/\*

\*@brief перегрузка оператора присваивания

\* @return сам объект для множественного присваивания

\*/

redblack& operator=(redblack&& other) noexcept;

/\*

\*@brief перегрузка оператора присваивания

\* @return сам объект для множественного присваивания

\*/

redblack& operator=(const redblack& other);

/\*

\* @brief Переопределение оператора равенства

\* @param other Объект, с которым сравнивается данный

\* @return результат сравнения

\*/

bool operator==(const redblack& other) const;

/\*

\* @brief функция для проверки высоты дерева, что количество черных узлов одинакого

\* @param root - корень дерева

\* @return true or false

\*/

bool validate\_black\_height(Node\* root);

/\*

\* @brief функция для сравнения двух деревьев

\* @param a,b - ноды деревьев

\* @return true or false

\*/

bool compare\_nodes(const Node\* a, const Node\* b) const;

/\*

\* @brief функция для сравнения двух деревьев

\* @return корень дерева

\*/

Node\* getRoot();

};

Geo.Test.cpp

// This is a personal academic project. Dear PVS-Studio, please check it.

// PVS-Studio Static Code Analyzer for C, C++, C#, and Java: https://pvs-studio.com

#include "CppUnitTest.h"

#include "../triangle/figure.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace GeoTests

{

TEST\_CLASS(GeoTests)

{

public:

public:

TEST\_METHOD(Insert\_SingleElement\_RootIsCorrect)

{

redblack tree;

tree.insert(10);

Assert::IsNotNull(tree.getRoot());

Assert::AreEqual(10, tree.getRoot()->data);

Assert::AreEqual(1, tree.getRoot()->color);

}

TEST\_METHOD(Delete\_LeafNode\_TreeRemainsValid)

{

redblack tree;

tree.insert(10);

tree.insert(5);

tree.insert(15);

tree.delete\_value(5);

Assert::IsNotNull(tree.getRoot());

Assert::AreEqual(2, tree.count\_nodes());

}

TEST\_METHOD(Insert\_Duplicates\_NotAdded)

{

redblack tree;

tree.insert(10);

tree.insert(10);

Assert::AreEqual(1, tree.count\_nodes());

}

TEST\_METHOD(BlackHeight\_Validation)

{

redblack tree{ 10, 5, 15, 1, 7, 12, 18 };

bool isValid = tree.validate\_black\_height(tree.getRoot());

Assert::IsTrue(isValid, L"Tree does not maintain black-height property");

}

TEST\_METHOD(CopyConstructor\_CreatesIdenticalTree)

{

redblack tree1{ 10, 5, 15, 1, 7, 12, 18 };

redblack tree2 = tree1;

Assert::IsTrue(tree1 == tree2);

}

TEST\_METHOD(MoveConstructor\_TreeTransferredCorrectly)

{

redblack tree1{ 10, 5, 15, 1, 7, 12, 18 };

redblack tree2 = std::move(tree1);

Assert::IsTrue(tree1.getRoot() == nullptr);

Assert::AreEqual(7, tree2.count\_nodes());

}

TEST\_METHOD(DeleteValue\_NotFound\_NoCrash)

{

redblack tree{ 10, 5, 15 };

tree.delete\_value(20);

Assert::AreEqual(3, tree.count\_nodes());

}

TEST\_METHOD(Constructor\_InitializerList\_BuildsTree)

{

redblack tree{ 10, 5, 15, 1, 7, 12, 18 };

Assert::AreEqual(7, tree.count\_nodes());

}

};

}