|  |  |
| --- | --- |
| **Российский университет транспорта (МИИТ)**  **Институт транспортной техники и систем управления**  **Кафедра «Управление и защита информации»** | |
| **Отчет**  **по практическому заданию**  **по теме «Структуры данных»**  **по дисциплине «Системы управления базами данных»** | |
|  | Выполнили:  Студенты группы ТКИ-442  Белов С.В.  Проверил:  Доцент кафедры УиЗи, к.т.н., с.н.с.  Васильева М.А. |
| Москва 2023 | |

Оглавление

[Задание 3](#_Toc153372406)

[1. Текст программы на языке C++ 3](#_Toc153372407)

[1.1. Код файла Node.h 3](#_Toc153372408)

[1.2. Код файла BST.h 5](#_Toc153372409)

[1.3. Код файлов Node.cpp и BST.cpp 10](#_Toc153372410)

[1.4. Код файлa main.cpp 10](#_Toc153372411)

[1.5. Код файлa UnitTest1.cpp – тесты для Node.h 11](#_Toc153372412)

[1.6. Код файлa UnitTest2.cpp – тесты для BST.h 13](#_Toc153372413)

[2. Результат работы программы 16](#_Toc153372414)

[3. UML диаграмма классов 18](#_Toc153372415)

[Заключение 19](#_Toc153372416)

# Задание

Разработать структуру данных на языке программирования С++ в ООП парадигме. В нашем случае структура данных – это бинарное дерево поиска (далее - БДП), основные операции которого будут поиск элемента, добавление элемента, удаление узла и обход дерева. Структура способна обрабатывать любые типы данных (template).

# 1. Текст программы на языке C++

## Код файла node.h

#pragma once

#include <iostream>

#include <sstream>

namespace tree {

/\*\*

\* @brief Structure node, for implematation of BST

\*/

template <typename T>

struct Node {

/\*\*

\* @brief value

\*/

T data;

/\*\*

\* @brief pointer on parent node

\*/

Node<T>\* parent;

/\*\*

\* @brief pointer on left node

\*/

Node<T>\* left;

/\*\*

\* @brief pointer on right node

\*/

Node<T>\* right;

/\*\*

\* @brief Constructor with param

\* @param value - value of inserted node

\*/

Node(const T& value);

Node(const Node<T>& node) = delete;

Node& operator =(const Node<T>& node) = delete;

/\*\*

\* @brief Move constructor

\* @param node

\*/

Node(Node<T>&& node) noexcept = default;

/\*\*

\* @brief Move operator

\* @param node

\*/

Node<T>& operator =(Node<T>&& node) noexcept = default;

/\*\*

\* @brief Destrucor.

\*/

~Node();

/\*\*

\* @brief Method for get info is current node root

\* @return true/false root/not root

\*/

bool is\_root() const noexcept;

/\*\*

\* @brief leaf check

\* @return true/false is leaf or not

\*/

bool is\_leaf() const noexcept;

/\*\*

\* @brief Operator to comparison

\* @param left - left node.

\* @param right - right node, with that we compare

\* @return result of comparison

\*/

friend auto operator <=>(const Node<T>& l, const Node<T>& r) {

if (std::less<T>()(l.data, r.data)) {

return -1;

}

if (std::greater<T>()(l.data, r.data)) {

return 1;

}

return 0;

}

/\*\*

\* @brief Operator to comparison

\* @param left - left node.

\* @param right - right node, with that we compare

\* @return result of comparison

\*/

friend bool operator ==(const Node<T>& l, const Node<T>& r) {

return (operator<=>(l, r) == 0);

}

/\*\*

\* @brief Operator not equel

\* @param left - left node.

\* @param right - right node, with that we compare

\* @return result of comparison

\*/

friend bool operator !=(const Node<T>& l, const Node<T>& r) {

return (operator<=>(l, r) != 0);

}

/\*\*

\* @brief Operator for output node

\* @param stream - input stream

\* @param node, that we output

\* @return output stream

\*/

friend std::ostream& operator<<(std::ostream& stream, const Node<T>& node) {

std::ostringstream buffer{};

buffer << node.data;

stream << buffer.str();

return stream;

};

};

template <typename T>

Node<T>::Node(const T& value) : data{ value }, parent{ nullptr }, left{ nullptr }, right{ nullptr } {}

template <typename T>

Node<T>::~Node() {

if (!this->is\_root()) {

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

this->parent->left = nullptr;

}

else {

this->parent->right = nullptr;

}

this->parent = nullptr;

}

this->left = nullptr;

this->right = nullptr;

}

template <typename T>

bool Node<T>::is\_root() const noexcept {

return this->parent == nullptr;

}

template <typename T>

bool Node<T>::is\_leaf() const noexcept {

return this->left == nullptr && this->right == nullptr;

}

}

## Код файла binary\_tree.h

#pragma once

#include "Node.h"

#include <vector>

#include <sstream>

#include <exception>

namespace tree {

/\*\*

\* @brief BST - binary search tree

\*/

template <typename T>

class BinaryTree {

public:

/\*\*

\* @brief default constructor

\*/

BinaryTree();

/\*\*

\* @brief param constructor

\* @param list of values for node in bst

\*/

BinaryTree(std::initializer\_list<T> list);

/\*\*

\* @brief copy constructor

\* @param other - tree from what we copy

\*/

BinaryTree(const BinaryTree& other);

/\*\*

\* @brief move constructor

\* @param other - tree from what we move

\*/

BinaryTree(BinaryTree&& other) noexcept;

/\*\*

\* @brief copy operator

\* @param other - tree from what we copy

\*/

BinaryTree& operator=(const BinaryTree& other);

/\*\*

\* @brief move operator

\* @param other - tree from what we move

\*/

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

/\*\*

\* @brief Destructor

\*/

virtual ~BinaryTree();

/\*\*

\* @brief insert new node

\* @param value - value of new node

\* @return true/false result of working method(sussess or fail)

\*/

bool add\_node(const T& value);

/\*\*

\* @brief remove selected node

\* @param value - value of node for delete

\* @return true/false result of working method(sussess or fail)

\*/

bool remove\_node(const T& value);

/\*\*

\* @brief Exist node with value

\* @param value - value of node.

\* @return true/false result of working method(sussess or fail)

\*/

bool has\_value(const T value) const noexcept;

/\*\*

\* @brief Method for check exist tree or not(exist root or not)

\* @return true/false result of working method(sussess or fail)

\*/

bool is\_empty() const noexcept;

/\*\*

\* @brief get number of nodes in tree

\*/

size\_t get\_size() const noexcept;

/\*\*

\* @brief Output of tree values

\* @return string with values

\*/

std::string in\_order\_print() const noexcept;

protected:

/\*\*

\* @brief root node

\*/

Node<T>\* root;

/\*\*

\* @brief number of nodes in tree

\*/

size\_t size;

/\*\*

\* @brief find node with min data

\* @param current - node from what we start search

\* @return adress of min node

\*/

Node<T>\* find\_min\_node(Node<T>\* current) const noexcept;

/\*\*

\* @brief find node with max data

\* @param current - node from what we start search

\* @return adress of min node

\*/

Node<T>\* find\_max\_node(Node<T>\* current) const noexcept;

/\*\*

\* @brief Change current node on son.

\* @param deleted - old node

\* @param son - new node

\*/

void transplant(Node<T>\* deleted, Node<T>\* son);

/\*\*

\* @brief swap nodes

\* @param other tree

\*/

void swap(BinaryTree& other) noexcept;

/\*\*

\* @brief find node by value

\* @param current - node for start search

\* @param target - value of node target

\* @return adress of node / nullptr, if not exist

\*/

Node<T>\* find\_by\_value(Node<T>\* current, const T& target) const noexcept;

private:

/\*\*

\* @brief Recurse delete tree

\* @param current - by default root

\*/

void in\_order\_remove\_tree(Node<T>\* current);

//

/\*\*

\* @brief Recurse form values

\* @param current node, from what we start values formation

\*/

void in\_order(Node<T>\* current);

/\*\*

\* @brief sort by data

\*/

std::vector<T> values;

/\*\*

\* @brief recurse call form values

\*/

void make\_values();

};

template <typename T>

void BinaryTree<T>::in\_order\_remove\_tree(Node<T>\* current) {

if (current == nullptr) {

return;

}

--this->size;

this->in\_order\_remove\_tree(current->left);

this->in\_order\_remove\_tree(current->right);

delete current;

current = nullptr;

}

template <typename T>

void BinaryTree<T>::in\_order(Node<T>\* current) {

if (current == nullptr) {

return;

}

this->in\_order(current->left);

this->values.push\_back(current->data);

this->in\_order(current->right);

}

template <typename T>

void BinaryTree<T>::make\_values() {

this->values.clear();

this->in\_order(root);

}

template <typename T>

Node<T>\* tree::BinaryTree<T>::find\_min\_node(Node<T>\* current) const noexcept {

Node<T>\* min = current;

while (min->left != nullptr) {

min = min->left;

}

return min;

}

template <typename T>

Node<T>\* tree::BinaryTree<T>::find\_max\_node(Node<T>\* current) const noexcept {

Node\* max = current;

while (max->right != nullptr) {

max = max->right;

}

return max;

}

template <typename T>

void BinaryTree<T>::transplant(Node<T>\* deleted, Node<T>\* son) {

if (deleted == son) {

return;

}

int tree\_data = son->data;

Node<T>\* tree\_parent = son->parent;

Node<T>\* tree\_right = son->right;

delete son;

if (tree\_parent == deleted) {

tree\_parent->right = tree\_right;

}

else {

tree\_parent->left = tree\_right;

}

if (tree\_right != nullptr) {

tree\_right->parent = tree\_parent;

}

deleted->data = tree\_data;

}

template <typename T>

void BinaryTree<T>::swap(BinaryTree& other) noexcept {

std::swap(this->root, other.root);

std::swap(this->root->left, other.root->left);

std::swap(this->root->right, other.root->right);

}

template <typename T>

Node<T>\* BinaryTree<T>::find\_by\_value(Node<T>\* current, const T& target) const noexcept {

if (current != nullptr) {

Node<T>\* node = new Node<T>(target);

Node<T>\* ptr = current;

while (\*node != \*ptr && ptr != nullptr) {

if (\*node < \*ptr) {

if (ptr->left != nullptr)

ptr = ptr->left;

else

return nullptr;

}

else if (\*node > \*ptr) {

if (ptr->right != nullptr)

ptr = ptr->right;

else

return nullptr;

}

}

return ptr;

}

return nullptr;

}

template <typename T>

BinaryTree<T>::BinaryTree() : root{ nullptr }, size{ 0 } {

}

template <typename T>

BinaryTree<T>::BinaryTree(std::initializer\_list<T> list) : BinaryTree() {

for (auto& item : list) {

this->add\_node(item);

}

}

template <typename T>

BinaryTree<T>::BinaryTree(const BinaryTree& other) : BinaryTree() {

for (auto& item : other.values) {

this->add\_node(item);

}

}

template <typename T>

BinaryTree<T>::BinaryTree(BinaryTree&& other) noexcept : BinaryTree() {

\*this = other;

}

template <typename T>

BinaryTree<T>& BinaryTree<T>::operator=(const BinaryTree& other) {

if (this != &other) {

BinaryTree node{ other };

this->swap(node);

}

return \*this;

}

template <typename T>

BinaryTree<T>& BinaryTree<T>::operator=(BinaryTree&& other) noexcept {

if (this != &other) {

this->swap(other);

}

return \*this;

}

template <typename T>

BinaryTree<T>::~BinaryTree() {

this->in\_order\_remove\_tree(this->root);

this->root = nullptr;

}

template <typename T>

bool BinaryTree<T>::add\_node(const T& value) {

Node<T>\* new\_node = new Node<T>( value );

if (this->is\_empty()) {

this->root = new\_node;

}

else {

Node<T>\* current = this->root;

Node<T>\* parent = nullptr;

while (current != nullptr && \*current != \*new\_node) {

parent = current;

if (\*new\_node < \*current) {

current = current->left;

}

else if (\*new\_node > \*current) {

current = current->right;

}

}

if (\*new\_node < \*parent) {

parent->left = new\_node;

}

else {

parent->right = new\_node;

}

new\_node->parent = parent;

}

++this->size;

this->make\_values();

return true;

}

template <typename T>

bool BinaryTree<T>::remove\_node(const T& value) {

Node<T>\* node = new Node<T>(value);

if (this->is\_empty()) {

throw std::logic\_error("Empty tree");

}

Node<T>\* current = this->root;

while (current != nullptr && \*node != \*current) {

if (\*node < \*current) {

current = current->left;

}

else if (\*node > \*current) {

current = current->right;

}

}

if (current == nullptr) {

throw std::logic\_error("Node with this value doesnt exist");

}

if (current->is\_leaf() == false) {

if (current->right != nullptr && current->left != nullptr) {

Node<T>\* min\_right = this->find\_min\_node(current->right);

transplant(current, min\_right);

}

else {

if (current->right != nullptr) {

current = current->right;

}

else {

current = current->left;

}

T data\_in\_tree = current->data;

Node<T>\* parent\_node = current->parent;

Node<T>\* right\_node = current->right;

Node<T>\* left\_node = current->left;

delete current;

parent\_node->data = data\_in\_tree;

parent\_node->right = right\_node;

parent\_node->left = left\_node;

}

}

else {

delete current;

current = nullptr;

}

--this->size;

this->make\_values();

return true;

}

template <typename T>

bool BinaryTree<T>::has\_value(const T value) const noexcept {

return this->find\_by\_value(this->root, value) != nullptr;

}

template <typename T>

bool BinaryTree<T>::is\_empty() const noexcept {

return this->root == nullptr;

}

template <typename T>

size\_t BinaryTree<T>::get\_size() const noexcept {

return this->size;

}

template <typename T>

std::string BinaryTree<T>::in\_order\_print() const noexcept {

std::ostringstream buffer{};

if (this->size == 0) {

buffer << "Empty tree";

}

else {

buffer << "{ ";

for (auto it = this->values.cbegin(); it != this->values.cend(); ++it) {

buffer << (\*it) << " ";

}

buffer << "}";

}

return buffer.str();

}

}

## Код файлов node.cpp и binary\_tree.cpp

Файл node.cpp:

#include "Node.h"

Файл binary\_tree.cpp:

#include "BST.h"

## Код файлa main.cpp

#include "tree\_container/binary\_tree.h"

#include <iostream>

void show\_work();

int main()

{

show\_work();

return 0;

}

void show\_work()

{

auto node = new tree::Node<int>(8);

auto tree = new tree::BinaryTree<int>();

tree->add\_node(8);

tree->add\_node(10);

tree->add\_node(11);

tree->add\_node(4);

std::cout << "Sorted bst: " << tree->in\_order\_print();

std::cout << "\nSize:" << tree->get\_size();

tree->remove\_node(4);

std::cout << "\nSorted bst(after delete node 4): " << tree->in\_order\_print();

bool has\_val = tree->has\_value(4);

std::cout << "\nHas value 4: " << has\_val;

}

## Код файлa Test\_node.cpp – тесты для node.h

#include "pch.h"

#include "CppUnitTest.h"

#include "../DBMS\_FINAL/New\_BST/tree\_container/binary\_tree.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Nodetest

{

TEST\_CLASS(NODE)

{

public:

TEST\_METHOD(PARAM\_INIT\_NODE\_SUCCESS)

{

//Arrange

auto node = new tree::Node<int>(5);

auto expected = 5;

//Act

auto real\_value = node->data;

//Assert

Assert::AreEqual(expected, real\_value);

Assert::IsTrue(real\_value);

}

TEST\_METHOD(EQUEL\_OPERATOR\_NODE\_SUCCESS)

{

//Arrange

auto left\_node = new tree::Node<int>(5);

auto right\_node = new tree::Node<int>(4);

auto expected = false;

//Act

auto result = (left\_node == right\_node);

//Assert

Assert::IsTrue(result == expected);

}

TEST\_METHOD(NOT\_EQUEL\_OPERATOR\_NODE\_SUCCESS)

{

//Arrange

auto left\_node = new tree::Node<int>(5);

auto right\_node = new tree::Node<int>(6);

auto expected = true;

//Act

auto result = (left\_node != right\_node);

//Assert

Assert::AreEqual(expected, result);

Assert::IsTrue(result);

}

TEST\_METHOD(LESS\_OPERATOR\_NODE\_SUCCESS)

{

//Arrange

auto left\_node = new tree::Node<int>(5);

auto right\_node = new tree::Node<int>(60);

auto expected = true;

//Act

auto result = (left\_node < right\_node);

//Assert

Assert::IsTrue(result == expected);

}

TEST\_METHOD(GREATER\_OPERATOR\_NODE\_SUCCESS)

{

//Arrange

auto left\_node = new tree::Node<int>(4);

auto right\_node = new tree::Node<int>(6);

auto expected = false;

//Act

auto result = (left\_node > right\_node);

//Assert

Assert::IsTrue(result == expected);

}

};

}

## Код файлa Test.cpp – тесты для binary\_tree.h

#include "pch.h"

#include <list>

#include "CppUnitTest.h"

#include "../DBMS\_FINAL/New\_BST/tree\_container/binary\_tree.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Test

{

TEST\_CLASS(BST)

{

public:

TEST\_METHOD(IS\_EMPTY\_SUCCESS)

{

//Arrange

auto tree = new tree::BinaryTree<int>();

auto expected = true; // default init - empty tree

//Act

auto is\_empty = tree->is\_empty();

//Assert

Assert::AreEqual(expected, is\_empty);

Assert::IsTrue(is\_empty);

}

TEST\_METHOD(GET\_SIZE\_SUCCESS)

{

//Arrange

std::initializer\_list<int> list{ 1, 5 };

auto tree = new tree::BinaryTree<int>(list);

auto expected = 2;

//Act

auto actual\_size = tree->get\_size();

//Assert

Assert::AreEqual(expected, int(actual\_size));

Assert::IsTrue(actual\_size);

}

TEST\_METHOD(PRINT\_SUCCESS)

{

//Arrange

std::initializer\_list<int> list{ 1, 5, 9 };

auto tree = new tree::BinaryTree<int>(list);

std::string expected = "{ 1 5 9 }";

//Act

auto print = tree->in\_order\_print();

//Assert

Assert::AreEqual(expected, print);

Assert::IsTrue(tree);

}

TEST\_METHOD(HAS\_INT\_VALUE\_SUCCESS)

{

//Arrange

std::initializer\_list<int> list{ 1, 5, 9 };

auto tree = new tree::BinaryTree<int>(list);

auto expected = true; // has value - true

//Act

auto actual = tree->has\_value(5);

//Assert

Assert::AreEqual(expected, actual);

Assert::IsTrue(actual);

}

TEST\_METHOD(ADD\_INT\_NODE\_SUCCESS)

{

//Arrange

auto tree = new tree::BinaryTree<int>();

auto expected = true; // has value - true

//Act

auto added = tree->add\_node(5);

auto actual = tree->has\_value(5); // tested upper

//Assert

Assert::AreEqual(expected, actual);

Assert::IsTrue(added);

}

TEST\_METHOD(REMOVE\_INT\_NODE\_SUCCESS)

{

//Arrange

std::initializer\_list<int> list{ 1, 5 };

auto tree = new tree::BinaryTree<int>(list);

auto expected = false; // has value - false

//Act

auto remove = tree->remove\_node(5);

auto actual = tree->has\_value(5);

//Assert

Assert::AreEqual(expected, actual);

Assert::IsTrue(remove);

}

TEST\_METHOD(HAS\_DOUBLE\_VALUE\_SUCCESS)

{

//Arrange

std::initializer\_list<double> list{ 1.1, 5.1, 9.1 };

auto tree = new tree::BinaryTree<double>(list);

auto expected = true; // has value - true

//Act

auto actual = tree->has\_value(5.1);

//Assert

Assert::AreEqual(expected, actual);

Assert::IsTrue(actual);

}

TEST\_METHOD(ADD\_DOUBLE\_NODE\_SUCCESS)

{

//Arrange

auto tree = new tree::BinaryTree<double>();

auto expected = true; // has value - true

//Act

auto added = tree->add\_node(5.1);

auto actual = tree->has\_value(5.1); // tested upper

//Assert

Assert::AreEqual(expected, actual);

Assert::IsTrue(added);

}

TEST\_METHOD(REMOVE\_DOUBLE\_NODE\_SUCCESS)

{

//Arrange

std::initializer\_list<double> list{ 1.1, 5.1};

auto tree = new tree::BinaryTree<double>(list);

auto expected = false; // has value - false

//Act

auto remove = tree->remove\_node(5.1);

auto actual = tree->has\_value(5.1);

//Assert

Assert::AreEqual(expected, actual);

Assert::IsTrue(remove);

}

};

}

# 2. Результат работы программы

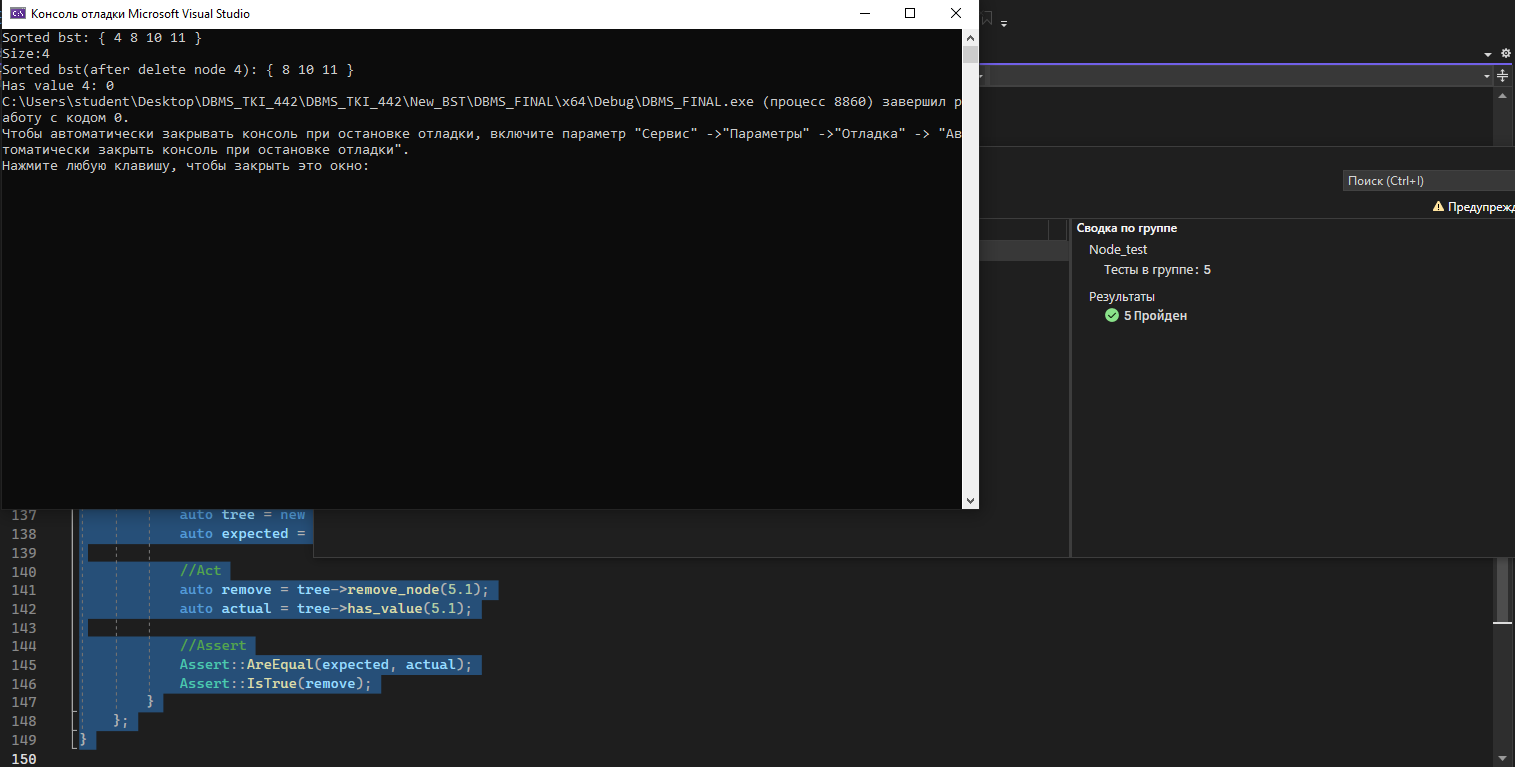


Рисунок 1 – Результат отладки программы

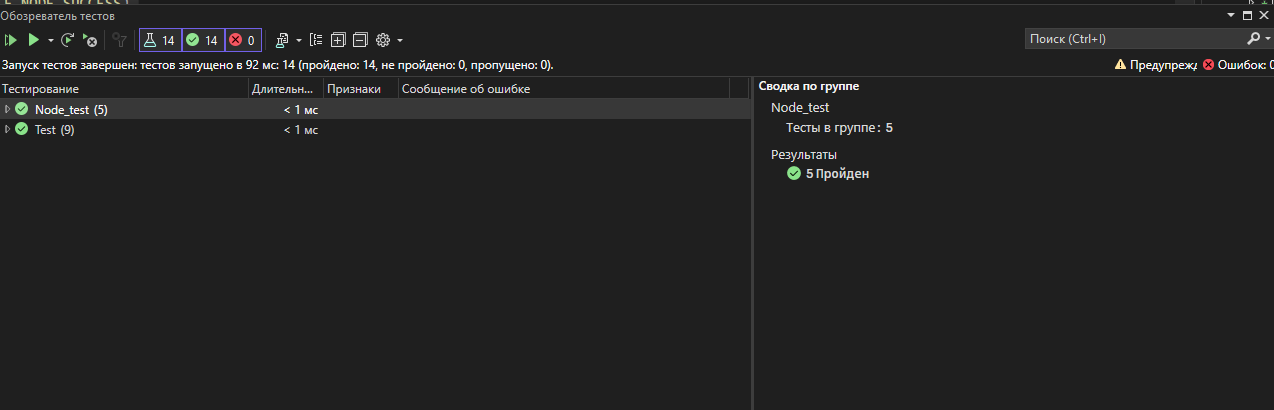


Рисунок 2 – Обозреватель тестов

# 3. UML диаграмма классов

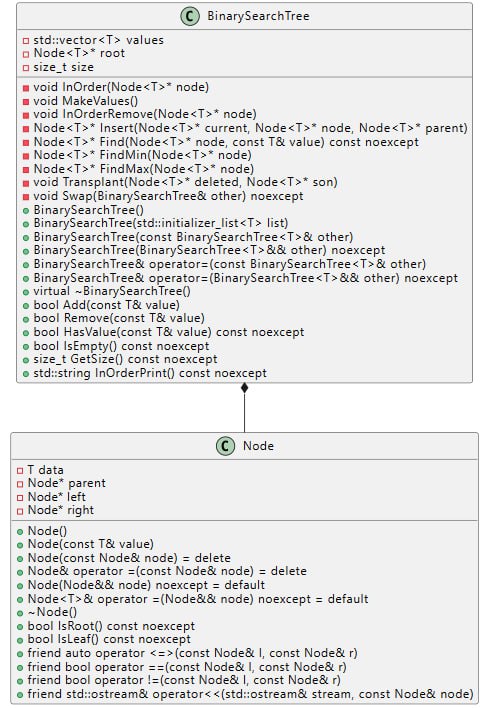


Рисунок 3 – UML диаграмма классов Node и BST

# Заключение

В результате выполнения практического задания была разработана структура данных - бинарное дерево поиска. Для обеспечения обработки различных типов данных была использована техника шаблонов, что позволило использовать данную структуру с любым типом данных.