|  |  |
| --- | --- |
| **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  Пономарев А.Д.  Дроздов А.Д.  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 P\_D\_Tree

{

/\*\*

\* @brief Structure node, for implematation of BST

\*/

template <typename T>

struct Node

{

/\*\*

\* @brief value

\*/

T data;

/\*\*

\* @brief pointer on parent node

\*/

Node\* parent;

/\*\*

\* @brief pointer on left node

\*/

Node\* left;

/\*\*

\* @brief pointer on right node

\*/

Node\* right;

/\*\*

\* @brief

\*/

Node();

/\*\*

\* @brief Constructor with param

\* @param value - value of inserted node

\*/

Node(const T& value);

/\*\*

\* @brief

\* @param node

\*/

Node(const Node& node) = delete;

/\*\*

\* @brief

\* @param node

\*/

Node& operator =(const Node& node) = delete;

/\*\*

\* @brief Move constructor

\* @param node

\*/

Node(Node&& node) noexcept = default;

/\*\*

\* @brief Move operator

\* @param node

\*/

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

/\*\*

\* @brief Destrucor.

\*/

~Node();

/\*\*

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

\* @return true/false root/not root

\*/

bool IsRoot() const noexcept;

/\*\*

\* @brief leaf check

\* @return true/false is leaf or not

\*/

bool IsLeaf() 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& l, const Node& r)

{

if (std::less<T>()(l.data, r.data)) { return -1; }

else 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& l, const Node& 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& l, const Node& 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& node)

{

std::ostringstream buffer{};

buffer << node.data;

stream << buffer.str();

return stream;

};

};

}

template <typename T>

P\_D\_Tree::Node<T>::Node() : data{ 0 }, parent{ nullptr }, left{ nullptr }, right{ nullptr } {}

template <typename T>

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

template <typename T>

P\_D\_Tree::Node<T>::~Node()

{

if (!this->IsRoot())

{

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 P\_D\_Tree::Node<T>::IsRoot() const noexcept { return this->parent == nullptr; }

template <typename T>

bool P\_D\_Tree::Node<T>::IsLeaf() const noexcept { return this->left == nullptr && this->right == nullptr; }

## Код файла BST.h

#pragma once

#include "Node.h"

#include <vector>

#include <sstream>

#include <exception>

namespace P\_D\_Tree

{

/\*\*

\* @brief BST - Binary search tree

\*/

template <typename T>

class BinarySearchTree

{

private:

std::vector<T> values;

void InOrder(Node<T>\* node);

void MakeValues();

void InOrderRemove(Node<T>\* node);

protected:

Node<T>\* root;

size\_t size;

Node<T>\* Insert(Node<T>\* current, Node<T>\* node, Node<T>\* parent);

Node<T>\* Find(Node<T>\* node, const T& value) const noexcept;

Node<T>\* FindMin(Node<T>\* node);

Node<T>\* FindMax(Node<T>\* node);

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

void Swap(BinarySearchTree& other) noexcept;

public:

BinarySearchTree();

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

BinarySearchTree(const BinarySearchTree<T>& other);

BinarySearchTree(BinarySearchTree<T>&& other) noexcept;

BinarySearchTree& operator=(const BinarySearchTree<T>& other);

BinarySearchTree& operator=(BinarySearchTree<T>&& other) noexcept;

virtual ~BinarySearchTree();

bool Add(const T& value);

bool Remove(const T& value);

bool HasValue(const T& value) const noexcept;

bool IsEmpty() const noexcept;

size\_t GetSize() const noexcept;

std::string InOrderPrint() const noexcept;

};

}

template<typename T>

void P\_D\_Tree::BinarySearchTree<T>::InOrder(Node<T>\* node)

{

if (node == nullptr) { return; }

this->InOrder(node->left);

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

this->InOrder(node->right);

}

template<typename T>

void P\_D\_Tree::BinarySearchTree<T>::MakeValues()

{

this->values.clear();

this->InOrder(this->root);

}

template <typename T>

void P\_D\_Tree::BinarySearchTree<T>::InOrderRemove(Node<T>\* node)

{

if (node == nullptr) { return; }

--this->size;

this->InOrderRemove(node->left);

this->InOrderRemove(node->right);

delete node;

node = nullptr;

}

template<typename T>

P\_D\_Tree::Node<T>\* P\_D\_Tree::BinarySearchTree<T>::Insert(Node<T>\* current, Node<T>\* node, Node<T>\* parent)

{

if (current == nullptr)

{

current = node;

current->parent = parent;

return current;

}

else if (\*node < \*current) { current->left = this->Insert(current->left, node, current); }

else { current->right = this->Insert(current->right, node, current); }

return current;

}

template <typename T>

P\_D\_Tree::Node<T>\* P\_D\_Tree::BinarySearchTree<T>::Find(Node<T>\* node, const T& value) const noexcept

{

if (node == nullptr) { return nullptr; }

else

{

if (std::less<T>()(value, node->data)) { return this->Find(node->left, value); }

else if (std::greater<T>()(value, node->data)) { return this->Find(node->right, value); }

else { return node; }

}

}

template <typename T>

P\_D\_Tree::Node<T>\* P\_D\_Tree::BinarySearchTree<T>::FindMin(Node<T>\* node)

{

while (nullptr != node->left) { node = node->left; }

return node;

}

template <typename T>

P\_D\_Tree::Node<T>\* P\_D\_Tree::BinarySearchTree<T>::FindMax(Node<T>\* node)

{

while (nullptr != node->right) { node = node->right; }

return node;

}

template <typename T>

void P\_D\_Tree::BinarySearchTree<T>::Transplant(Node<T>\* deleted, Node<T>\* son)

{

if (deleted == son) { return; }

T TreeData = son->data;

Node<T>\* TreeParent = son->parent;

Node<T>\* TreeRight = son->right;

delete son;

if (TreeParent == deleted) { TreeParent->right = TreeRight; }

else if (TreeRight != nullptr) { TreeRight->parent = TreeParent; }

else { TreeParent->left = TreeRight; }

deleted->data = TreeData;

}

template <typename T>

void P\_D\_Tree::BinarySearchTree<T>::Swap(BinarySearchTree<T>& other) noexcept

{

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

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

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

}

template <typename T>

P\_D\_Tree::BinarySearchTree<T>::BinarySearchTree() : root{ nullptr }, size{ 0 } {}

template <typename T>

P\_D\_Tree::BinarySearchTree<T>::BinarySearchTree(std::initializer\_list<T> list) : BinarySearchTree()

{

for (auto& item : list) { this->Add(item); }

}

template <typename T>

P\_D\_Tree::BinarySearchTree<T>::BinarySearchTree(const BinarySearchTree<T>& other) : BinarySearchTree()

{

for (auto& item : other) { this->Add(item); }

}

template <typename T>

P\_D\_Tree::BinarySearchTree<T>::BinarySearchTree(BinarySearchTree<T>&& other) noexcept : BinarySearchTree()

{

\*this = other;

}

template <typename T>

P\_D\_Tree::BinarySearchTree<T>& P\_D\_Tree::BinarySearchTree<T>::operator=(const BinarySearchTree<T>& other)

{

if (this != other)

{

BinarySearchTree<T> temp(other);

this->Swap(temp);

}

return \*this;

}

template <typename T>

P\_D\_Tree::BinarySearchTree<T>& P\_D\_Tree::BinarySearchTree<T>::operator=(BinarySearchTree<T>&& other) noexcept

{

if (this != other) { this->Swap(other); }

return \*this;

}

template <typename T>

P\_D\_Tree::BinarySearchTree<T>::~BinarySearchTree()

{

this->InOrderRemove(this->root);

this->root = nullptr;

}

template <typename T>

bool P\_D\_Tree::BinarySearchTree<T>::Add(const T& value)

{

auto newNode = new Node<T>(value);

if (this->root == nullptr) { this->root = newNode; }

else { this->root = this->Insert(this->root, newNode, this->root->parent); }

++this->size;

this->MakeValues();

return true;

}

template <typename T>

bool P\_D\_Tree::BinarySearchTree<T>::Remove(const T& value)

{

auto newNode = new Node<T>(value);

if (this->IsEmpty()) { throw std::logic\_error("Empty tree"); }

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

while (current != nullptr && \*newNode != \*current)

{

if (\*newNode < \*current) { current = current->left; }

else if (\*newNode > \*current) { current = current->right; }

}

if (current == nullptr) { throw std::logic\_error("Node with this value doesnt exist"); }

else if (current->IsLeaf() == false)

{

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

{

Node<T>\* MinRight = this->FindMin(current->right);

Transplant(current, MinRight);

}

else

{

if (current->right != nullptr) { current = current->right; }

else { current = current->left; }

T DataTree = current->data;

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

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

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

delete current;

ParentNode->data = DataTree;

ParentNode->right = RightNode;

ParentNode->left = LeftNode;

}

}

else

{

delete current;

current = nullptr;

}

--this->size;

this->MakeValues();

return true;

}

template <typename T>

bool P\_D\_Tree::BinarySearchTree<T>::HasValue(const T& value) const noexcept

{

return this->Find(this->root, value) != nullptr;

}

template <typename T>

bool P\_D\_Tree::BinarySearchTree<T>::IsEmpty() const noexcept

{

return this->root == nullptr;

}

template <typename T>

size\_t P\_D\_Tree::BinarySearchTree<T>::GetSize() const noexcept

{

return this->size;

}

template <typename T>

std::string P\_D\_Tree::BinarySearchTree<T>::InOrderPrint() const noexcept

{

std::ostringstream buffer{};

buffer << "{ ";

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

{

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

}

buffer << "}";

return buffer.str();

}

## Код файлов Node.cpp и BST.cpp

Файл Node.cpp:

#include "Node.h"

Файл BST.cpp:

#include "BST.h"

## Код файлa main.cpp

#include "BST.h"

#include <iostream>

void Check();

int main()

{

Check();

return 0;

}

void Check()

{

P\_D\_Tree::BinarySearchTree<int> BST;

BST.Add(7);

BST.Add(10);

BST.Add(8);

BST.Add(11);

BST.Add(9);

std::cout << "Binary tree: " << BST.InOrderPrint();

std::cout << "\nSize:" << BST.GetSize();

BST.Remove(8);

std::cout << "\nBinary tree after delete node: " << BST.InOrderPrint();

bool has\_val = BST.HasValue(8);

std::cout << "\nHas value 8: " << has\_val;

}

## Код файлa UnitTest1.cpp – тесты для Node.h

#include "pch.h"

#include "CppUnitTest.h"

#include "../bst\_2/Node.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

TEST\_CLASS(NodeTest)

{

public:

TEST\_METHOD(DefaultConstructor\_Int\_Success)

{

P\_D\_Tree::Node<int> defaultNode;

Assert::IsTrue(defaultNode.IsRoot());

Assert::IsTrue(defaultNode.IsLeaf());

}

TEST\_METHOD(DefaultConstructor\_String\_Success)

{

P\_D\_Tree::Node<std::string> defaultNode;

Assert::IsTrue(defaultNode.IsRoot());

Assert::IsTrue(defaultNode.IsLeaf());

}

TEST\_METHOD(ParamConstructor\_Int\_Success)

{

int value = 42;

P\_D\_Tree::Node<int> paramNode(value);

Assert::IsTrue(paramNode.IsRoot());

Assert::IsTrue(paramNode.IsLeaf());

Assert::AreEqual(paramNode.data, value);

}

TEST\_METHOD(ParamConstructor\_String\_Success)

{

std::string value = "42";

P\_D\_Tree::Node<std::string> paramNode(value);

Assert::IsTrue(paramNode.IsRoot());

Assert::IsTrue(paramNode.IsLeaf());

Assert::AreEqual(paramNode.data, value);

}

TEST\_METHOD(MoveConstructor\_Int\_Success)

{

int value = 42;

P\_D\_Tree::Node<int> paramNode(value);

P\_D\_Tree::Node<int> movedNode(std::move(paramNode));

Assert::IsTrue(movedNode.IsRoot());

Assert::IsTrue(movedNode.IsLeaf());

Assert::AreEqual(movedNode.data, value);

}

TEST\_METHOD(MoveConstructor\_String\_Success)

{

std::string value = "42";

P\_D\_Tree::Node<std::string> paramNode(value);

P\_D\_Tree::Node<std::string> movedNode(std::move(paramNode));

Assert::IsTrue(movedNode.IsRoot());

Assert::IsTrue(movedNode.IsLeaf());

Assert::AreEqual(movedNode.data, value);

}

TEST\_METHOD(MoveAssignmentOperator\_Int\_Success)

{

int value = 42;

P\_D\_Tree::Node<int> paramNode(value);

P\_D\_Tree::Node<int> anotherNode;

anotherNode = std::move(paramNode);

Assert::IsTrue(anotherNode.IsRoot());

Assert::IsTrue(anotherNode.IsLeaf());

Assert::AreEqual(anotherNode.data, value);

}

TEST\_METHOD(MoveAssignmentOperator\_String\_Success)

{

std::string value = "42";

P\_D\_Tree::Node<std::string> paramNode(value);

P\_D\_Tree::Node<std::string> anotherNode;

anotherNode = std::move(paramNode);

Assert::IsTrue(anotherNode.IsRoot());

Assert::IsTrue(anotherNode.IsLeaf());

Assert::AreEqual(anotherNode.data, value);

}

TEST\_METHOD(IsRoot\_Int\_Success)

{

P\_D\_Tree::Node<int> rootNode;

Assert::IsTrue(rootNode.IsRoot());

}

TEST\_METHOD(IsRoot\_String\_Success)

{

P\_D\_Tree::Node<std::string> rootNode;

Assert::IsTrue(rootNode.IsRoot());

}

TEST\_METHOD(IsLeaf\_Int\_Success)

{

P\_D\_Tree::Node<int> rootNode;

Assert::IsTrue(rootNode.IsLeaf());

}

TEST\_METHOD(IsLeaf\_String\_Success)

{

P\_D\_Tree::Node<std::string> rootNode;

Assert::IsTrue(rootNode.IsLeaf());

}

TEST\_METHOD(ComparisonOperators\_Int\_Success)

{

P\_D\_Tree::Node<int> equalNode1(42);

P\_D\_Tree::Node<int> equalNode2(42);

P\_D\_Tree::Node<int> lessNode(21);

P\_D\_Tree::Node<int> greaterNode(63);

Assert::IsTrue(equalNode1 == equalNode2);

Assert::IsTrue(equalNode1 != lessNode);

Assert::IsTrue(lessNode < greaterNode);

Assert::IsTrue(greaterNode > lessNode);

}

TEST\_METHOD(ComparisonOperators\_String\_Success)

{

P\_D\_Tree::Node<std::string> equalNode1("42");

P\_D\_Tree::Node<std::string> equalNode2("42");

P\_D\_Tree::Node<std::string> lessNode("21");

P\_D\_Tree::Node<std::string> greaterNode("63");

Assert::IsTrue(equalNode1 == equalNode2);

Assert::IsTrue(equalNode1 != lessNode);

Assert::IsTrue(lessNode < greaterNode);

Assert::IsTrue(greaterNode > lessNode);

}

};

## Код файлa UnitTest2.cpp – тесты для BST.h

#include "pch.h"

#include "CppUnitTest.h"

#include "../bst\_2/BST.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace P\_D\_Tree

{

TEST\_CLASS(BinarySearchTreeTests)

{

public:

TEST\_METHOD(Size\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> bst{ 5, 3, 7 };

Assert::AreEqual(static\_cast<size\_t>(3), bst.GetSize());

}

TEST\_METHOD(Size\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> bst{ "5", "3", "7" };

Assert::AreEqual(static\_cast<size\_t>(3), bst.GetSize());

}

TEST\_METHOD(Add\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> bst;

bst.Add(5);

bst.Add(3);

bst.Add(7);

Assert::AreEqual(static\_cast<size\_t>(3), bst.GetSize());

}

TEST\_METHOD(Add\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> bst;

bst.Add("5");

bst.Add("3");

bst.Add("7");

Assert::AreEqual(static\_cast<size\_t>(3), bst.GetSize());

}

TEST\_METHOD(Remove\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> bst{ 5, 3, 7 };

Assert::AreEqual(static\_cast<size\_t>(3), bst.GetSize());

Assert::IsTrue(bst.Remove(3));

Assert::AreEqual(static\_cast<size\_t>(2), bst.GetSize());

Assert::IsFalse(bst.HasValue(3));

}

TEST\_METHOD(Remove\_Int\_ExpectException)

{

P\_D\_Tree::BinarySearchTree<int> bst{ 5, 3, 7 };

auto func = [&] { bst.Remove(4); };

Assert::ExpectException<std::logic\_error>(func);

}

TEST\_METHOD(Remove\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> bst{ "5", "3", "7" };

Assert::AreEqual(static\_cast<size\_t>(3), bst.GetSize());

Assert::IsTrue(bst.Remove("3"));

Assert::AreEqual(static\_cast<size\_t>(2), bst.GetSize());

Assert::IsFalse(bst.HasValue("3"));

}

TEST\_METHOD(Remove\_String\_ExpectException)

{

P\_D\_Tree::BinarySearchTree<std::string> bst{ "5", "3", "7" };

auto func = [&] { bst.Remove("4"); };

Assert::ExpectException<std::logic\_error>(func);

}

TEST\_METHOD(OrderPrint\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> bst{ 5, 3, 7, 1, 4 };

Assert::AreEqual(std::string("{ 1 3 4 5 7 }"), bst.InOrderPrint());

}

TEST\_METHOD(OrderPrint\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> bst{ "5", "3", "7", "1", "4" };

Assert::AreEqual(std::string("{ 1 3 4 5 7 }"), bst.InOrderPrint());

}

TEST\_METHOD(IsEmpty\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> emptyBst;

Assert::IsTrue(emptyBst.IsEmpty());

}

TEST\_METHOD(IsEmpty\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> emptyBst;

Assert::IsTrue(emptyBst.IsEmpty());

}

TEST\_METHOD(IsNotEmpty\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> nonEmptyBst{ 1, 2, 3 };

Assert::IsFalse(nonEmptyBst.IsEmpty());

}

TEST\_METHOD(IsNotEmpty\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> nonEmptyBst{ "1", "2", "3" };

Assert::IsFalse(nonEmptyBst.IsEmpty());

}

TEST\_METHOD(DefaultConstructor\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> bst;

Assert::IsTrue(bst.IsEmpty());

Assert::AreEqual(static\_cast<size\_t>(0), bst.GetSize());

}

TEST\_METHOD(DefaultConstructor\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> bst;

Assert::IsTrue(bst.IsEmpty());

Assert::AreEqual(static\_cast<size\_t>(0), bst.GetSize());

}

TEST\_METHOD(InitializerListConstructor\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> bst{ 5, 3, 7, 1, 4 };

Assert::AreEqual(std::string("{ 1 3 4 5 7 }"), bst.InOrderPrint());

}

TEST\_METHOD(InitializerListConstructor\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> bst{ "5", "3", "7", "1", "4" };

Assert::AreEqual(std::string("{ 1 3 4 5 7 }"), bst.InOrderPrint());

}

TEST\_METHOD(HasValue\_Int\_Success)

{

P\_D\_Tree::BinarySearchTree<int> bst{ 5, 3, 7, 1, 4 };

Assert::IsTrue(bst.HasValue(3));

Assert::IsFalse(bst.HasValue(10));

}

TEST\_METHOD(HasValue\_String\_Success)

{

P\_D\_Tree::BinarySearchTree<std::string> bst{ "5", "3", "7", "1", "4" };

Assert::IsTrue(bst.HasValue("3"));

Assert::IsFalse(bst.HasValue("10"));

}

};

}

# 2. Результат работы программы

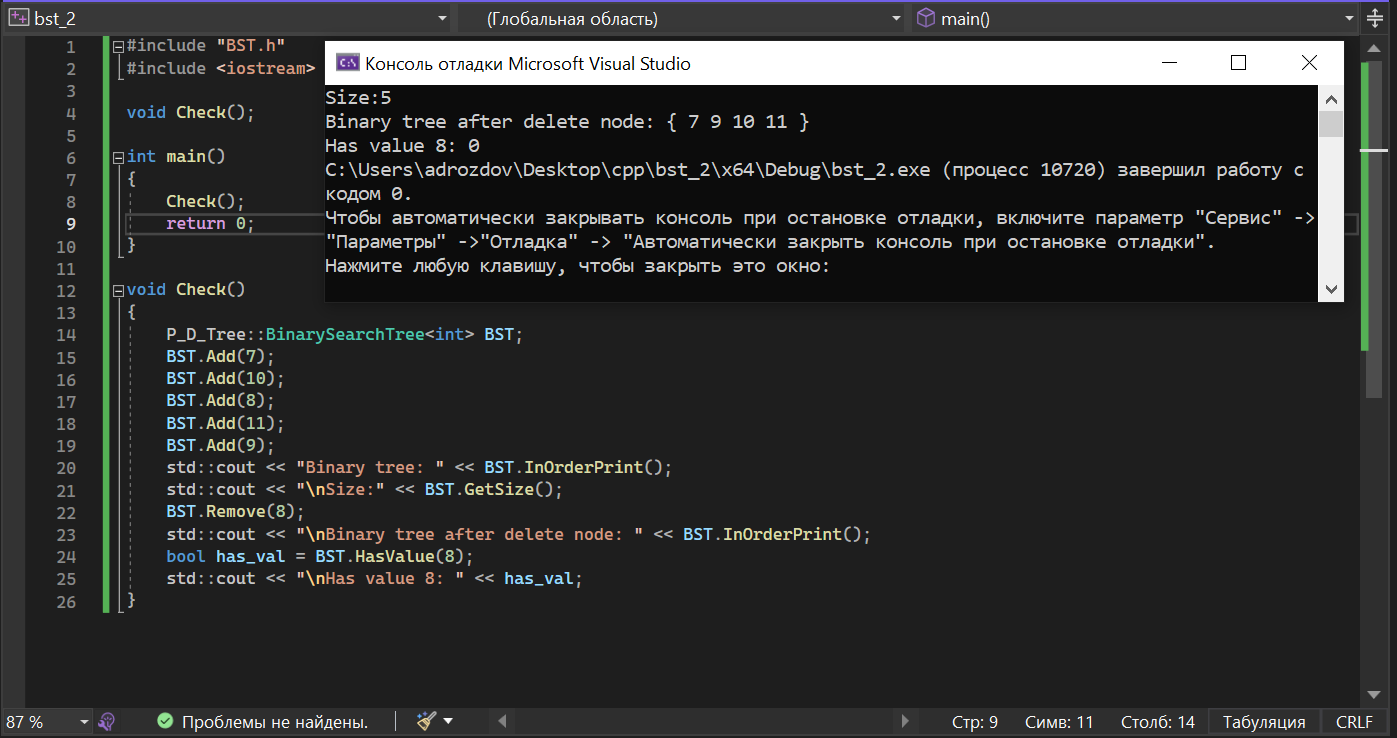


Рисунок 1 – Результат отладки программы

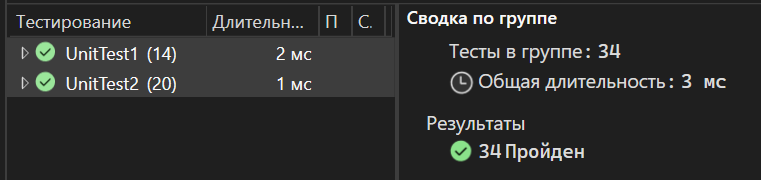


Рисунок 2 – Обозреватель тестов

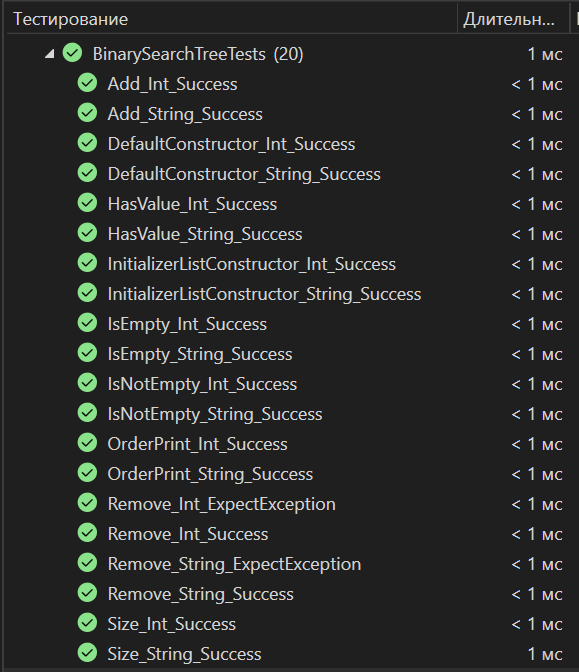


Рисунок 3 – Результат выполнения тестов для BST.h

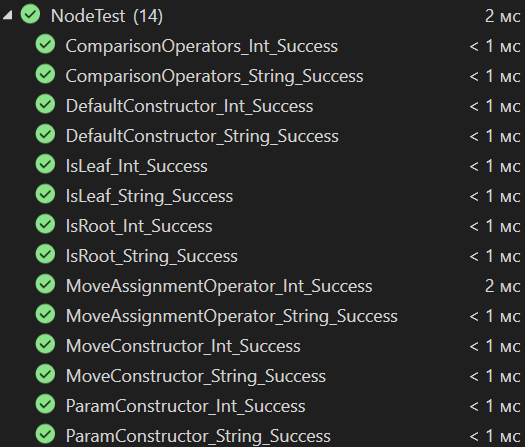


Рисунок 4 – Результат выполнения тестов для Node.h

# 3. UML диаграмма классов

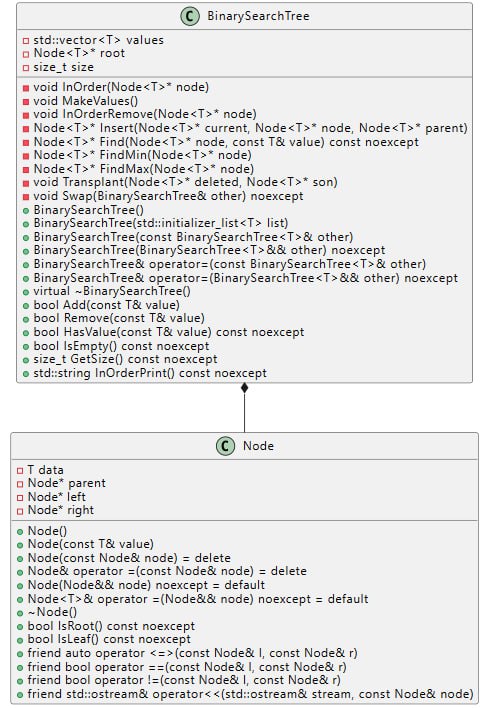


Рисунок 5 – UML диаграмма классов Node и BST

# 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.