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

Разработать структуру данных на языке программирования C++ 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++

1.1. 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; }
```

1.2. 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();
}
```

1.3. Код файлов Node.cpp и BST.cpp

Файл Node.cpp:

```
#include "Node.h"
```

Файл BST.cpp:

```
#include "BST.h"
```

1.4. Код файла 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;
}
```

1.5. Код файла 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);
        }
    };
```

1.6. Код файла 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

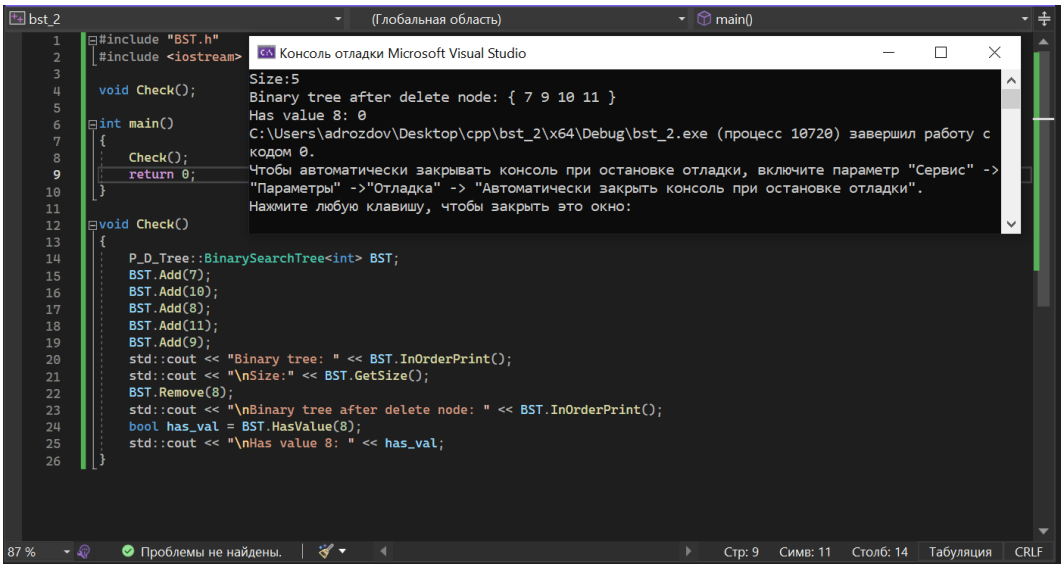

Рисунок 1 – Результат отладки программы

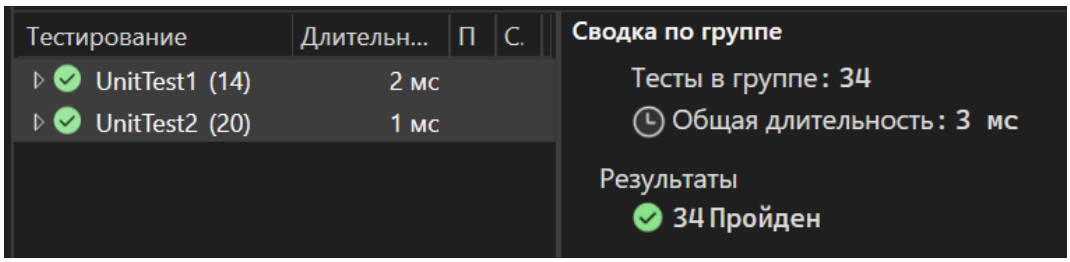

Рисунок 2 – Обозреватель тестов

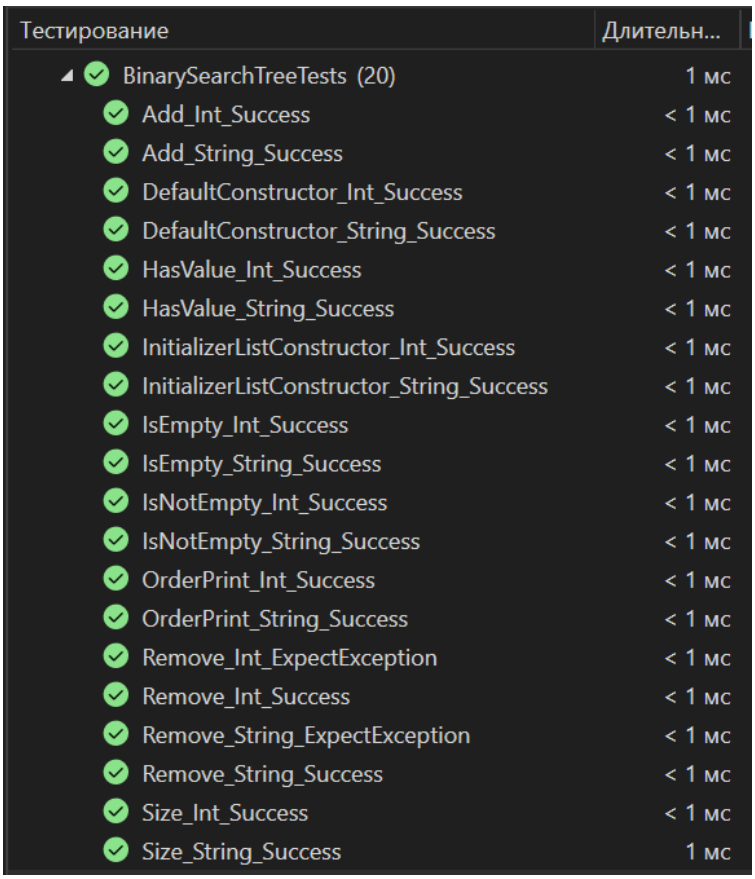

Рисунок 3 – Результат выполнения тестов для BST.h

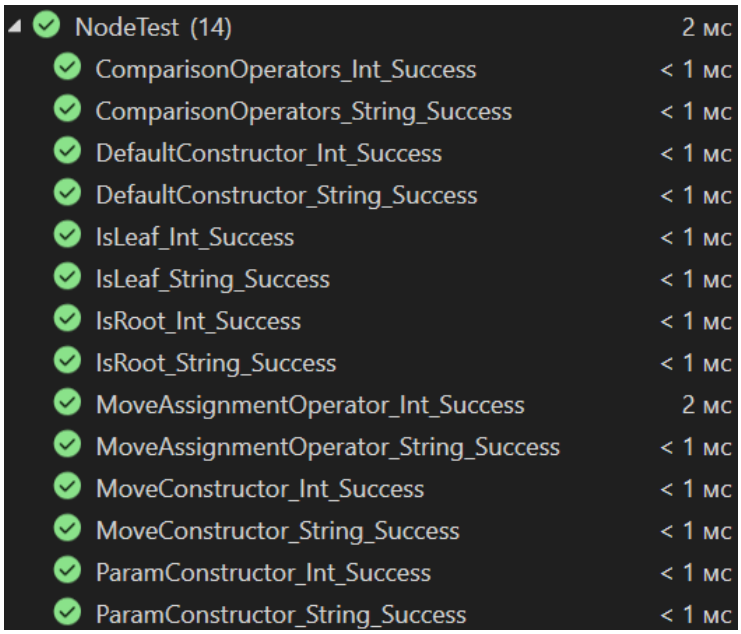

Рисунок 4 – Результат выполнения тестов для Node.h

3. UML диаграмма классов

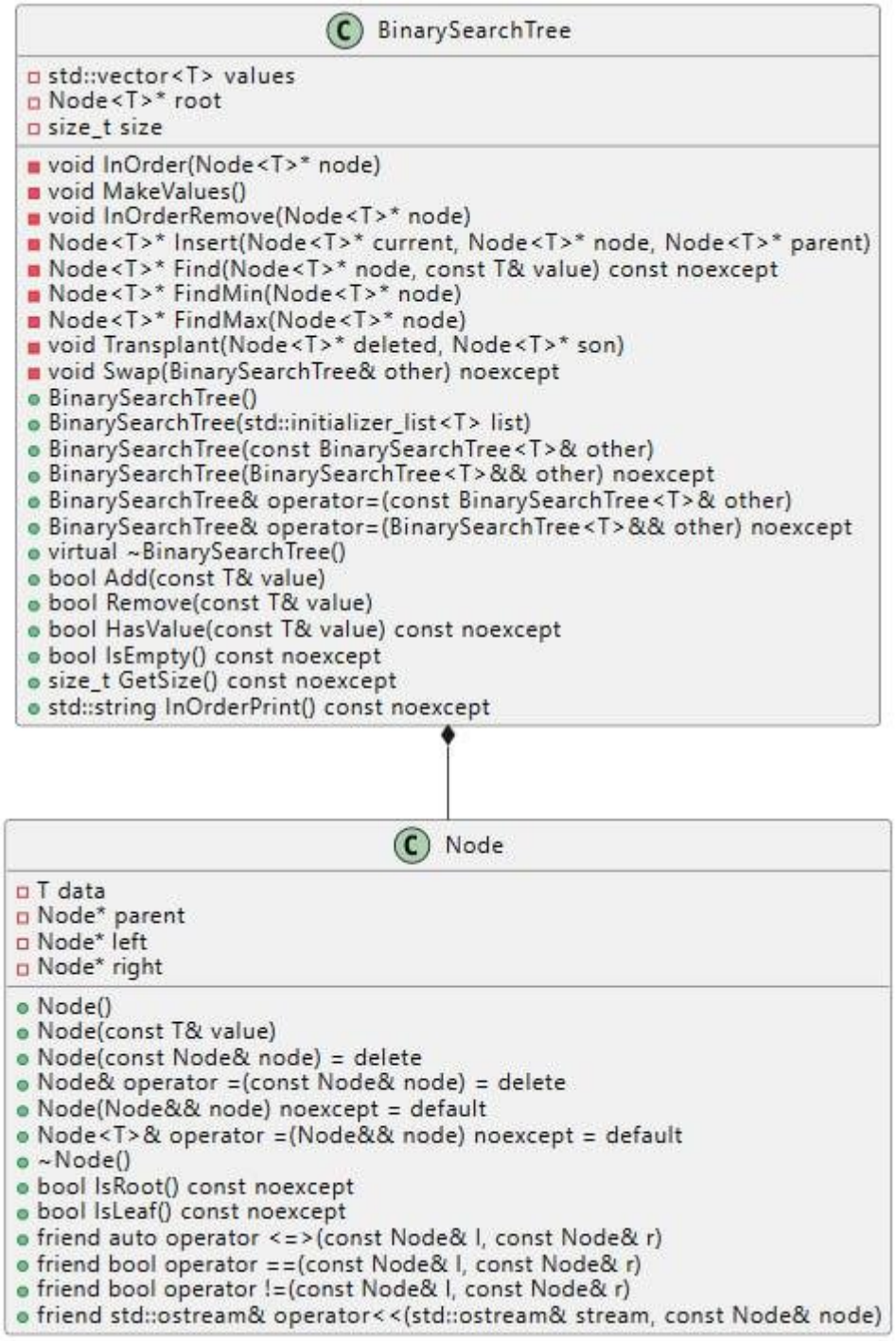

Рисунок 5 – UML диаграмма классов Node и BST

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.