МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

АНАЛИЗ ДВОИЧНОГО ДЕРЕВА ПОИСКА

ОТЧЁТ

студента 2 курса 251 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Толстова Роберта Сергеевича

Проверено:

доцент, к. ф.-м. н. _____ М. И. Сафрончик

Саратов 2025

СОДЕРЖАНИЕ

1 Анализ двоичного дерева поиска 3
1.1 Реализация класса на C++ 3

1 Анализ двоичного дерева поиска

1.1 Реализация класса на C++

```
class BinarySearchTree
{
    std::unique_ptr<TreeNode> root;

    // Recursive insertion helper
    void insertRecursive(std::unique_ptr<TreeNode> &node, int val)
    {
        if (!node)
        {
            node = std::make_unique<TreeNode>(val);
            Logger::log(Logger::Level::INFO, "Inserted value: " +
std::to_string(val));
        }
        else if (val < node->value)
        {
            insertRecursive(node->left, val);
        }
        else
        {
            insertRecursive(node->right, val);
        }
    }

    // Recursive deletion helper
    std::unique_ptr<TreeNode>
&deleteRecursive(std::unique_ptr<TreeNode> &node, int val)
    {
        if (!node)
        {
```

```
            Logger::log(Logger::Level::WARNING, "Value not found
for deletion: " + std::to_string(val));
            return node;
        }

        if (val < node->value)
        {
            node->left = std::move(deleteRecursive(node->left,
val));
        }
        else if (val > node->value)
        {
            node->right = std::move(deleteRecursive(node->right,
val));
        }
        else
        {
            if (!node->left)
                return node->right;
            if (!node->right)
                return node->left;

            node->value = findMinimum(node->right)->value;
            node->right = std::move(deleteRecursive(node->right,
node->value));
        }
        return node;
    }

    // Helper to find minimum value node
    std::unique_ptr<TreeNode>
```

```
&findMinimum(std::unique_ptr<TreeNode> &node) noexcept
    {
        return node->left ? findMinimum(node->left) : node;
    }

public:
    BinarySearchTree() = default;

    // Insert interface
    void insert(int val) noexcept
    {
        try
        {
            insertRecursive(root, val);
        }
        catch (const std::bad_alloc &)
        {
            Logger::log(Logger::Level::ERROR, "Memory allocation
failed for value: " + std::to_string(val));
        }
    }

    // Deletion interface
    void remove(int val) noexcept
    {
        if (search(val))
        {
            root = std::move(deleteRecursive(root, val));
            Logger::log(Logger::Level::INFO, "Removed value: " +
std::to_string(val));
        }
```

```
        else
        {
            Logger::log(Logger::Level::WARNING, "Attempted to
remove non-existent value: " + std::to_string(val));
        }
    }

    // Search functionality
    [[nodiscard]] bool search(int val) const noexcept
    {
        const TreeNode *current = root.get();
        while (current)
        {
            if (val == current->value)
                return true;
            current = val < current->value ? current->left.get() :
current->right.get();
        }
        return false;
    }

    // Traversal methods
    void traversePreOrder() const noexcept
    {
        Logger::log(Logger::Level::INFO, "Pre-order traversal:");
        std::function<void(const std::unique_ptr<TreeNode> &)>
traverse =
            [&](const std::unique_ptr<TreeNode> &node)
        {
            if (node)
            {
```

```
                std::cout << node->value << " ";
                traverse(node->left);
                traverse(node->right);
            }
        };
        traverse(root);
        std::cout << "\n";
    }

    void traverseInOrder() const noexcept
    {
        Logger::log(Logger::Level::INFO, "In-order traversal:");
        std::function<void(const std::unique_ptr<TreeNode> &)>
traverse =
            [&](const std::unique_ptr<TreeNode> &node)
        {
            if (node)
            {
                traverse(node->left);
                std::cout << node->value << " ";
                traverse(node->right);
            }
        };
        traverse(root);
        std::cout << "\n";
    }

    void traversePostOrder() const noexcept
    {
        Logger::log(Logger::Level::INFO, "Post-order traversal:");
        std::function<void(const std::unique_ptr<TreeNode> &)>
```

```
traverse =
            [&](const std::unique_ptr<TreeNode> &node)
        {
            if (node)
            {
                traverse(node->left);
                traverse(node->right);
                std::cout << node->value << " ";
            }
        };
        traverse(root);
        std::cout << "\n";
    }

    // File operations
    void loadFromFile(const std::string &filename) noexcept
    {
        try
        {
            std::ifstream file(filename);
            if (!file)
                throw std::ios_base::failure("File open failed");

            int value;
            while (file >> value)
            {
                insert(value);
            }
            Logger::log(Logger::Level::INFO, "Loaded tree from: " +
filename);
        }
```

```
        catch (const std::exception &e)
        {
            Logger::log(Logger::Level::ERROR, "Failed to load file:
" + std::string(e.what()));
        }
    }

    void saveToFile(const std::string &filename) const noexcept
    {
        try
        {
            if (!root)
                Logger::log(Logger::Level::ERROR, "No root node
found");

            auto calculateMaxDepth = [](const
std::unique_ptr<TreeNode> &node)
            {
                short max_depth = 0;
                std::queue<std::pair<const TreeNode *, short>> q;
                if (node)
                    q.emplace(node.get(), 1);

                while (!q.empty())
                {
                    auto [current, depth] = q.front();
                    q.pop();

                    max_depth = std::max(max_depth, depth);
                    if (current->left)
                        q.emplace(current->left.get(), depth + 1);
```

```
                if (current->right)
                    q.emplace(current->right.get(), depth + 1);
            }
            return max_depth;
        };

        const short max_depth = calculateMaxDepth(root);
        const short max_width = 1 << (max_depth - 1);

        std::vector<std::vector<const TreeNode *>>
levels(max_depth,

std::vector<const TreeNode *>(max_width, nullptr));

        std::queue<std::tuple<const TreeNode *, short, short>>
q;
        q.emplace(root.get(), 0, 0);

        while (!q.empty())
        {
            auto [node, level, pos] = q.front();
            q.pop();

            levels[level][pos] = node;
            if (node->left)
                q.emplace(node->left.get(), level + 1, 2 *
pos);
            if (node->right)
                q.emplace(node->right.get(), level + 1, 2 * pos
+ 1);
        }
```

```
            std::ofstream out(filename);
            if (out)
            {
                short offset = (1 << (max_depth - 1)) + 1;
                short current_width = 1;

                for (short level = 0; level < max_depth; ++level)
                {
                    out << std::setw(offset / 2 + 1);
                    levels[level][0] ? out << levels[level][0]-
>value : out << ' ';

                    for (short pos = 1; pos < current_width; ++pos)
                    {
                        out << std::setw(offset);
                        levels[level][pos] ? out << levels[level]
[pos]->value : out << ' ';
                    }

                    offset >>= 1;
                    current_width <<= 1;
                    out << '\n';
                }
            }
            Logger::log(Logger::Level::INFO, "Saved tree to: " +
filename);
        }
        catch (const std::exception &e)
        {
            Logger::log(Logger::Level::ERROR, "Failed to save file:
```

```
" + std::string(e.what()));
        }
    }
};
```

Временная сложность в лучшем и среднем случае для вставки, удаления и поиска элементов O(log n)

Временная сложность в худшем случае (дерево не сбалансированное) вставки, удаления и поиска равна O(n)