МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МИРЭА – РОССИЙСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»
РТУ МИРЭА

Институт ИКБ

Специальность (направление): 09.03.02 (информационные системы и технологии)

Кафедра: КБ-3 «Разработка программных решений и системного программирования»

Дисциплина: «Алгоритмы и структуры данных»

Практическая работа на тему:
Программа по деревьям

Студент: 20.12.2024 Игнатьева М.С.

подпись *Дата* *инициалы и фамилия*

Группа: БСБО-16-23 Шифр: 23Б0088

Преподаватель: 20.12.2024 Филатов В.В.

подпись *дата* *инициалы и фамилия*

Москва 2024 г.

1. Задание (вариант 88)

88.	Оптимальное дерево двоичного поиска	Список сыновей	$C=A \cup_{сим} B$	симметричный обход

2. Термины

Оптимальное дерево двоичного поиска — это дерево, в котором элементы упорядочены таким образом, чтобы минимизировать средневзвешенную высоту, то есть общие затраты на поиск каждого элемента с учетом его частоты.

Список сыновей — это структура, в которой у каждого узла есть два возможных потомка: левый сын и правый брат. Эта структура подходит для представления двоичных деревьев с ограничением на количество детей.

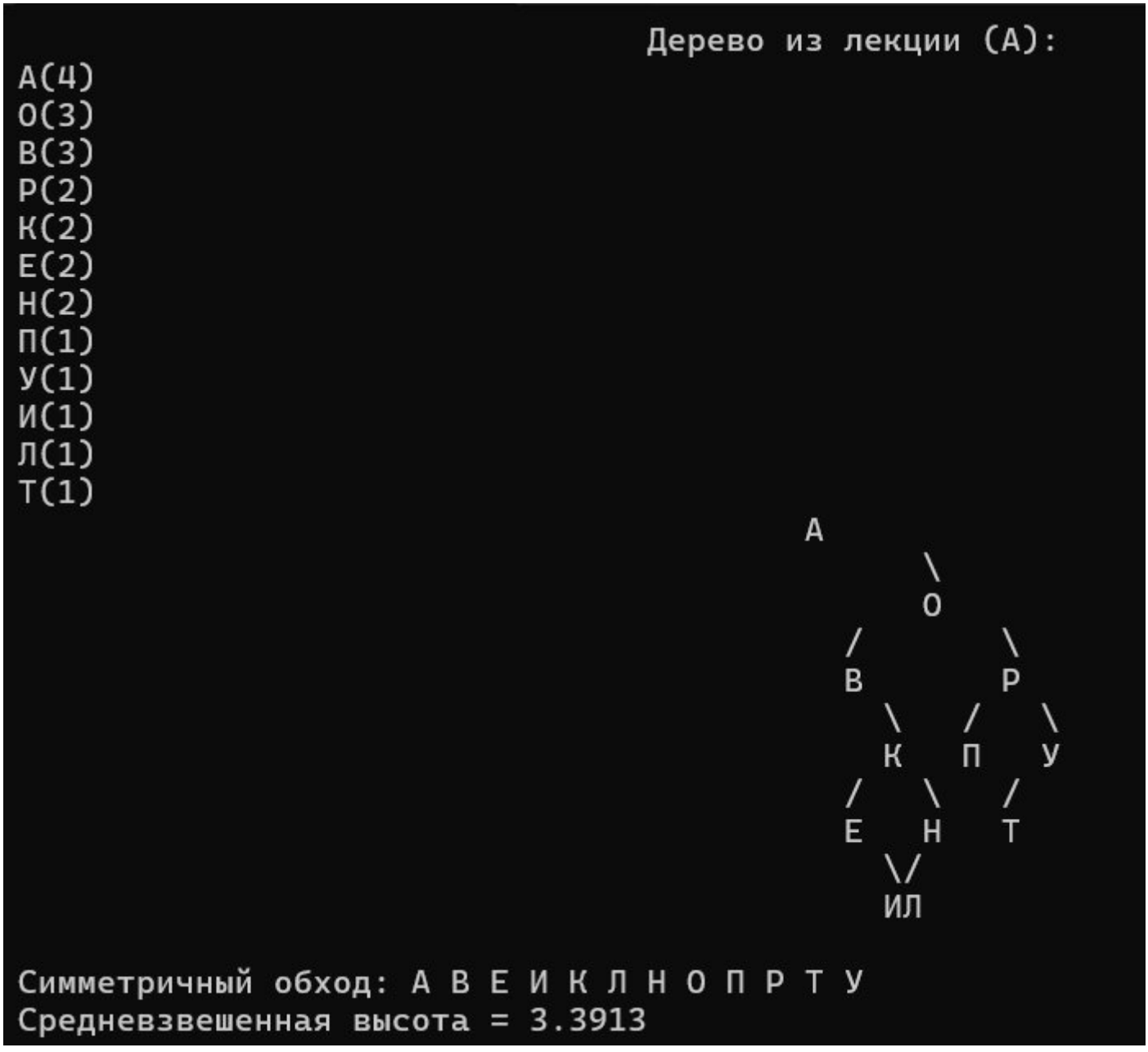
Симметричный обход — это способ обхода двоичного дерева, при котором сначала обрабатывается левый дочерний узел, затем текущий узел, и в конце — правый дочерний узел. Этот обход часто используется для работы с деревьями поиска, так как он посещает элементы в отсортированном порядке.

3. Описание программы

- fillFreqs(const std::string& string) — заполняет частотный словарь символами из строки, подсчитывая количество каждого символа – веса.
- push() — строит дерево на основе частот символов, добавляя элементы в дерево с учетом их значений и частот.
- calculateP(const Node* node) — рекурсивно рассчитывает переменную P, которая зависит от уровня узла и частоты символа.
- calculateH() — вычисляет средневзвешенную высоту дерева H, используя ранее вычисленные значения P и W.
- printWeights() — выводит символы дерева в порядке убывания их частот.
- printTree() — выводит дерево в виде текстового представления, показывая структуру узлов и их отношения (с помощью слэшей и обратных слэшей для обозначения детей).
- MAKENULL(Node*& node) — рекурсивно очищает память, освобождая все узлы дерева.
- PARENT(const Node* node) — возвращает родителя узла в дереве.
- RIGHT_SIBLING(const Node* node) — возвращает правого брата узла, если он существует.
- LEFT_CHILD(const Node* node) — возвращает левого сына узла, если он существует.
- myOperation(const Tree<T>& A, const Tree<T>& B, Tree<T>& C) — выполняет операцию.

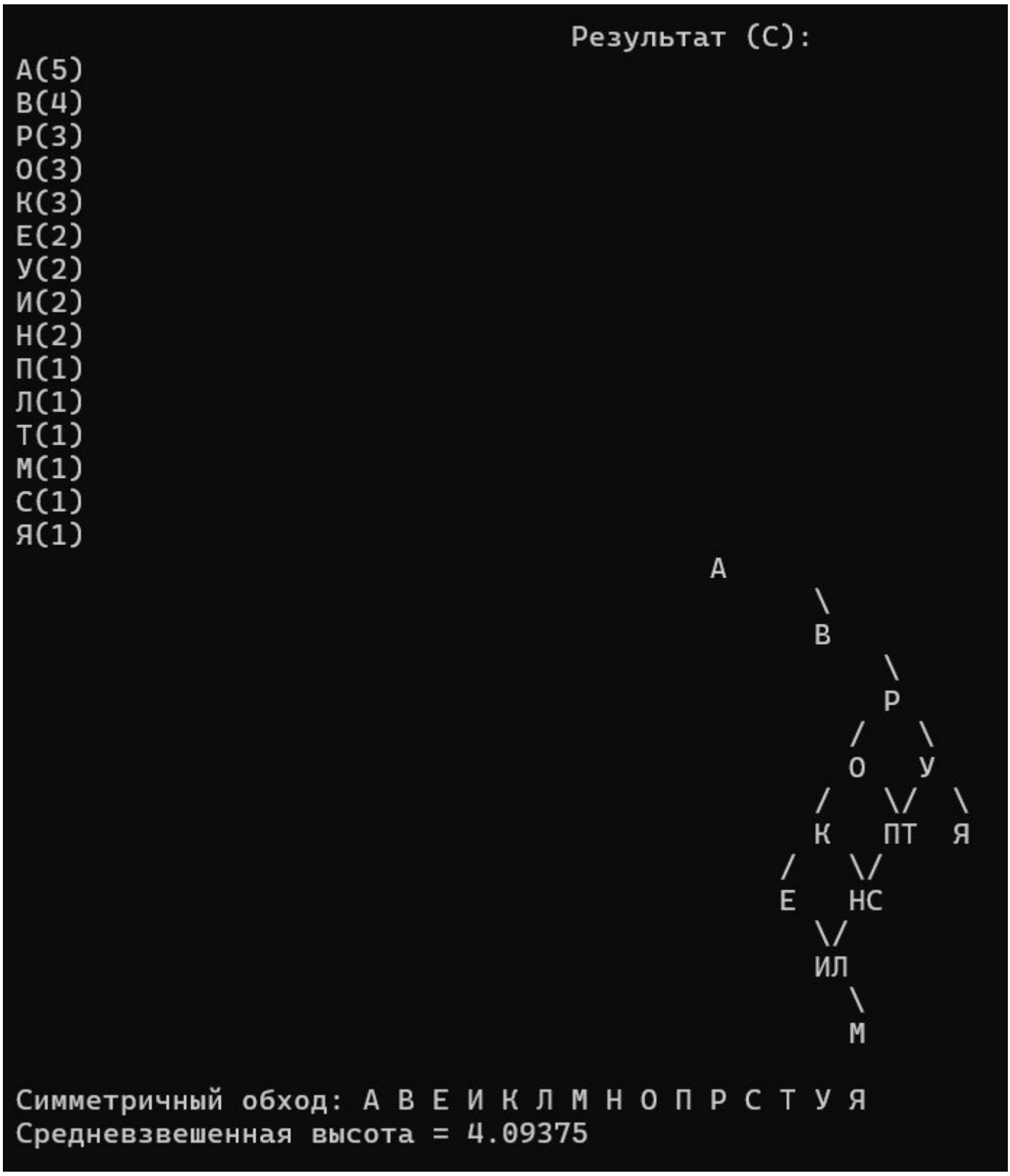
- printInOrder(const Node* start) — выполняет симметричный обход дерева, начиная с указанного узла, и выводит его элементы в отсортированном порядке.
- Iterator — класс для итерации по дереву в симметричном обходе. Реализует операторы ++ и * для прохода по дереву.
- begin() — возвращает итератор на первый элемент дерева для симметричного обхода.
- end() — возвращает итератор, указывающий на конец дерева (после последнего элемента).

4. Скриншот работы программы



```
                                        Моё дерево (В):
А(2)
С(2)
Я(2)
В(1)
К(1)
И(1)
Р(1)
М(1)
У(1)
```


```
Симметричный обход: А В И К М Р С У Я
Средневзвешенная высота = 3.25
```



5. Исходный код

```
#include <vector>

#include <iostream>

#include <unordered_map>
```

```cpp
#include <sstream>
#include <functional>
#include <map>
#include <string>
#include <algorithm>
#include <utility>
#include <stack>

template <typename T>
class Tree {
private:
    struct Node {
        std::vector<Node*> children;
        T value;
        unsigned level;

        Node(const T& v) : value(v), level(1) {}
    };

    Node* root;
    std::unordered_map<T, size_t> frequencies;
    double P, W, H;

    void calculateP(const Node* node) {
        if (!node) return;

        if (!node->children.empty()) calculateP(node->children[0]);
        if (node->children.size() == 2) calculateP(node->children[1]);
```

```
        P += node->level * frequencies[node->value];
    }

    void calculateH() {
        calculateP(root);

        for (const auto& p : frequencies) W += p.second;

        if (W != 0) H = P / W;
    }

    void printWeights() const {
        std::vector<std::pair<T, size_t>> sortedFrequencies(frequencies.begin(),
frequencies.end());
        std::sort(sortedFrequencies.begin(), sortedFrequencies.end(),
            [](const std::pair<T, size_t>& a, const std::pair<T, size_t>& b) { return
a.second > b.second; });

        for (auto const& p : sortedFrequencies) std::cout << p.first << '(' << p.second <<
')' << std::endl;
    }

public:
    double getH() {
        calculateH();

        return H;
    }

    void printTree() const {
        printWeights();
```

```
        std::function<void(const Node*, int, int, std::map<int, std::string>&)>
buildTreeLines =

            [&](const Node* node, int depth, int position, std::map<int, std::string>&
levels) {

            if (!node) return;


            std::ostringstream oss;

            oss << node->value;

            std::string value = oss.str();


            if (levels.count(depth) == 0) {

                levels[depth] = std::string(position, ' ') + value;

            }

            else {

                if (static_cast<int>(levels[depth].size()) < position) {

                    levels[depth] += std::string(position - levels[depth].size(), ' ') +
value;

                } else {

                    levels[depth] += value;

                }

            }


            if (!node->children.empty()) {

                int spacing = std::max(2, 6 - depth);

                int leftPosition = position - spacing;

                int rightPosition = position + spacing;


                if (node->children.size() >= 1 && node->children[0]) {

                    if (static_cast<int>(levels[depth + 1].size()) < leftPosition) {
```

```
                    levels[depth + 1] += std::string(leftPosition - levels[depth +
1].size(), ' ') + "/";

                }

                else {

                    levels[depth + 1] += "/";

                }

                buildTreeLines(node->children[0], depth + 2, leftPosition, levels);

            }


            if (node->children.size() >= 2 && node->children[1]) {

                if (static_cast<int>(levels[depth + 1].size()) < rightPosition) {

                    levels[depth + 1] += std::string(rightPosition - levels[depth +
1].size(), ' ') + "\\";

                }

                else {

                    levels[depth + 1] += "\\";

                }

                buildTreeLines(node->children[1], depth + 2, rightPosition, levels);

            }

        }

    };


    std::map<int, std::string> levels;

    buildTreeLines(root, 0, 40, levels);


    for (const auto& p : levels) {

        std::cout << p.second << std::endl;

    }
```

```
    }


private:

    void fillFreqs(const std::string& string) { for (const auto& symbol : string)
++frequencies[symbol]; }


    void push() {

        std::vector<std::pair<T, size_t>> sortedFrequencies(frequencies.begin(),
frequencies.end());

        std::sort(sortedFrequencies.begin(), sortedFrequencies.end(),

            [](const std::pair<T, size_t>& a, const std::pair<T, size_t>& b) { return
a.second > b.second; });


        for (const std::pair<T, size_t>& entry : sortedFrequencies) {

            Node* newNode = new Node(entry.first);


            if (!root) {

                root = newNode;

            }

            else {

                std::function<void(Node*&, unsigned)> insertNode = [&](Node*& node, const
unsigned& currentLevel = 1) {

                    if (newNode->value < node->value) {

                        if (node->children.empty()) node->children.resize(2, nullptr);

                        if (!node->children[0]) {

                            newNode->level = currentLevel + 1;

                            node->children[0] = newNode;

                        }

                        else {

                            insertNode(node->children[0], currentLevel + 1);
```

```
                    }
                }
                else if (newNode->value > node->value) {
                    if (node->children.size() < 2) node->children.resize(2, nullptr);
                    if (!node->children[1]) {
                        newNode->level = currentLevel + 1;
                        node->children[1] = newNode;
                    }
                    else {
                        insertNode(node->children[1], currentLevel + 1);
                    }
                }
                };

            insertNode(root, 1);
        }
    }
}

static Tree CREATE(Node*& node, Tree<T>& T1, Tree<T>& T2) {
    Tree<T> newTree;
    if (!node) return newTree;

    newTree.root = node;

    if (T1.root) node->children.push_back(T1.root);
    if (T2.root) node->children.push_back(T2.root);

    return newTree;
```

```
    }



    void MAKENULL(Node*& node) {
        if (node) {
            if (!node->children.empty()) MAKENULL(node->children[0]);
            if (node->children.size() == 2) MAKENULL(node->children[1]);
            delete node;
            node = nullptr;
        }
    }

public:
    const Node* PARENT(const Node* node) const {
        if (!root || !node || node == root) return nullptr;

        std::function<const Node*(const Node*)> findParent = [&](const Node* current) ->
const Node* {
            if (!current) return nullptr;

            if ((current->children.size() > 0 && current->children[0] == node) ||
                (current->children.size() > 1 && current->children[1] == node)) {
                return current;
            }

            if (current->children.size() > 0) {
                const Node* leftResult = findParent(current->children[0]);
                if (leftResult) return leftResult;
            }
```

```
            if (current->children.size() > 1) {
                return findParent(current->children[1]);
            }
        };

        return findParent(root);
    }

    const Node* RIGHT_SIBLING(const Node* node) const {
        if (!root || !node || root == node) return nullptr;

        const Node* parent = PARENT(node);
        if (!parent) return nullptr;
        if (parent->children.size() == 2 && parent->children[0] == node) return
parent->children[1];

        return nullptr;
    }

    Node* LEFT_CHILD(const Node* node) const { return (root && node
&& !node->children.empty()) ? node->children[0] : nullptr; }
    T LABEL(const Node* node) const { return (root && node) ? node->value : T(); }
    Node* ROOT() const { return root; }

public:
    Tree(const std::string& input) : P(W = H = 0.0), root(nullptr) {
        fillFreqs(input);
        push();
    }

    Tree() : P(W = H = 0.0), root(nullptr) {}
```

```
    ~Tree() { MAKENULL(root); }

    void printInOrder(const Node* start) const {
        if (!start) return;

        std::function<void(const Node*)> inOrder = [&](const Node* current) {
            if (!current) return;

            if (!current->children.empty()) inOrder(current->children[0]);
            std::cout << current->value << ' ';
            if (current->children.size() == 2) inOrder(current->children[1]);
        };

        inOrder(start);
    }

    static void myOperation(const Tree<T>& A, const Tree<T>& B, Tree<T>& C) {
        if (!A.ROOT() || !B.ROOT()) return;

        std::string nodesFromB;
        std::function<void(const Node*)> inOrder = [&](const Node* current) {
            if (!current) return;

            if (!current->children.empty()) inOrder(current->children[0]);
            nodesFromB.push_back(current->value);
            if (current->children.size() == 2) inOrder(current->children[1]);
        };

        inOrder(B.ROOT());
```

```
        C.frequencies = A.frequencies;
        //C.push();
        //C.clearFreqs();
        C.fillFreqs(nodesFromB);
        C.push();
    }

private:
    class Iterator {
    public:
        const Node* current;
        std::stack<const Node*> nodes;

        Iterator(const Node* root) : current(nullptr) {
            pushLeft(root);
            moveToNext();
        }

        T operator*() const { return current->value; }

        Iterator& operator++() {
            moveToNext();
            return *this;
        }

        bool operator!=(const Iterator& other) const { return current != other.current; }

    private:
        void pushLeft(const Node* node) {
```

```
            while (node) {
                nodes.push(node);
                node = (node->children.empty() ? nullptr : node->children[0]);
            }
        }

        void moveToNext() {
            if (nodes.empty()) {
                current = nullptr;
                return;
            }

            current = nodes.top();
            nodes.pop();

            if (current->children.size() == 2) {
                pushLeft(current->children[1]);
            }
        }
    };

public:
    Iterator begin() const { return Iterator(root); }
    Iterator end() const { return Iterator(nullptr); }
};

int main() {
    setlocale(LC_ALL, "Russian");
```

```
    Tree<char> A("РОВПОВАЕЕКУВИЛРКТОАНАНА");

    std::cout << "\t\t\t\tДерево из лекции (A):\n"; A.printTree();

    std::cout << "\nСимметричный обход: "; A.printInOrder(A.ROOT());

    std::cout << "\nСредневзвешенная высота = " << A.getH() << std::endl;


    std::cout << std::endl;


    Tree<char> B("КРАСИВАЯМУСЯ");

    std::cout << "\t\t\t\tМоё дерево (B):\n"; B.printTree();

    std::cout << "\nСимметричный обход: "; for (auto it = B.begin(); it != B.end(); ++it)
std::cout << *it << ' ';

    std::cout << "\nСредневзвешенная высота = " << B.getH() << std::endl;


    std::cout << std::endl;


    Tree<char> C; Tree<char>::myOperation(A, B, C);

    std::cout << "\t\t\t\tРезультат (C):\n"; C.printTree();

    std::cout << "\nСимметричный обход: "; C.printInOrder(C.ROOT());

    std::cout << "\nСредневзвешенная высота = " << C.getH() << std::endl;


    return 0;

}
```

6. Вывод

В ходе выполнения практической работы я реализовала оптимальное дерево двоичного поиска с использованием списка сыновей. В процессе разработки я познакомилась с различными методами работы с деревьями, такими как симметричный обход, а также освоение операций слияния деревьев, что позволило мне углубить знания в области структур данных. Особое внимание было уделено вычислению средневзвешенной высоты дерева, что стало полезным инструментом для оценки эффективности работы с деревьями.

7. Литература

Кормен Т. Х., Лейзерсон Ч. Е., Ривест Р. L., Штайн К. — *Введение в алгоритмы*, стр. 539, Глава 12. Деревья поиска.

Хиршберг Д. С., Чьенг В. В. — *Деревья и алгоритмы на них*, стр. 186, Глава 7. Алгоритмы обхода и модификации деревьев.

Лекции и практики – преподаватель Филатов В. В.