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 Альзоаби А.Ф.

подпись *Дата* *инициалы и фамилия*

Группа: БСБО-16-23 Шифр: 23Б0045

Преподаватель: 20.12.2024 Филатов В.В.

подпись *дата* *инициалы и фамилия*

Москва 2024 г.

1. Задание (вариант 45)

45.	Оптимальное дерево двоичного поиска	Левый сын, правый брат (указатели)	C = A \ B	обратный обход

2. Термины

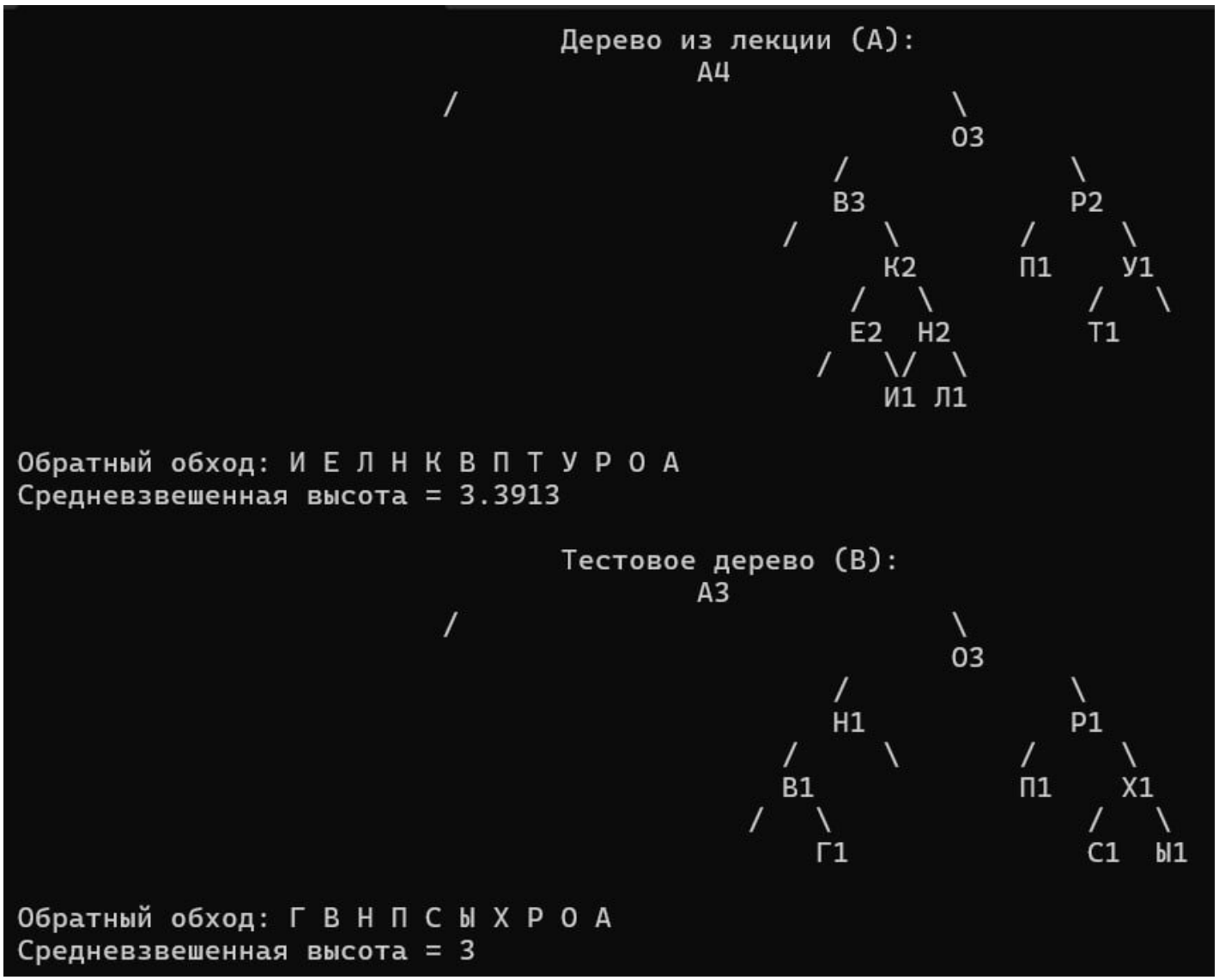
- Оптимальное дерево двоичного поиска — дерево, в котором элементы организованы таким образом, чтобы обеспечить минимальную высоту и максимальную эффективность операций поиска, вставки и удаления. Это дерево приближенно сбалансировано, что минимизирует количество операций при выполнении этих действий.
- Реализация через указатель на левого сына и правого брата — способ представления дерева, где каждый узел хранит два указателя: один на его левого сына (первого потомка), а второй — на правого брата (соседнего узла на одном уровне). Этот метод используется для экономии памяти и упрощения структуры дерева, особенно в случае деревьев с переменным количеством детей.
- Обратный обход — вид обхода дерева, при котором сначала обрабатывается левое поддерево, затем правое, и только после этого корень.

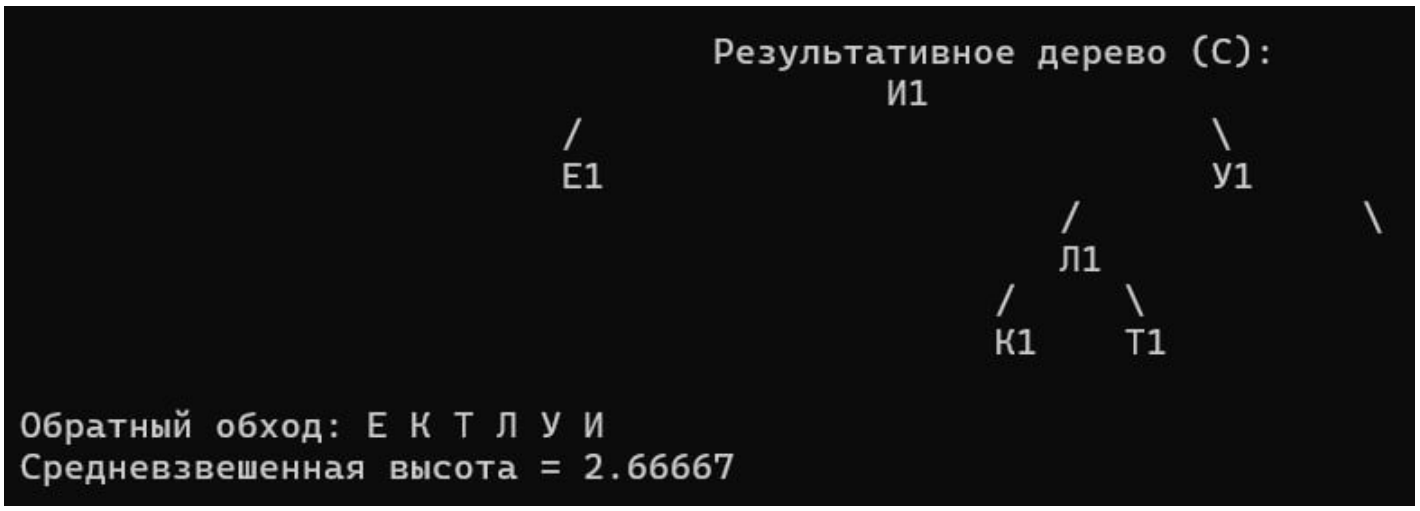
3. Описание программы

- calculateP: Рекурсивно вычисляет сумму P для дерева, используя уровень узла и его частоту.
- calculateH: Вычисляет средневзвешенную высоту H дерева на основе значения P и W (сумма частот элементов).
- getH: Возвращает средневзвешенную высоту дерева (вызывается calculateH для вычислений).
- printTree: Выводит дерево в виде строки с визуальной иерархией (позволяет увидеть структуру дерева).
- parseInputData: Парсит входные данные и подсчитывает частоты символов в строке.
- autoPush: Автоматически строит дерево, вставляя узлы по частотам и организуя их в структуру с указателями на левого сына и правого брата.
- CREATE: Создает новое дерево с заданным узлом и двумя поддеревьями
- PARENT: Возвращает родительский узел для заданного узла.
- LEFT_CHILD: Возвращает левого сына для заданного узла.
- RIGHT_SIBLING: Возвращает правого брата для заданного узла.
- LABEL: Возвращает значение узла (метка).
- MAKENULL: Удаляет узел и его поддеревья (рекурсивно освобождает память).
- printPostorder: Выводит дерево в обратном обходе.

- isContains: Проверяет, содержится ли узел в дереве, используя обратный обход для поиска значения.
- myOperation: Выполняет операцию C = A \ B.
- Iterator: Предоставляет итератор для обхода дерева в обратном порядке, позволяя перебирать элементы дерева (использует стек и множество для отслеживания посещенных узлов).

4. Скриншот работы программы





5. Исходный код

```
/* Программа по деревьям

Название: оптимальное дерево двоичного поиска

Реализация: левый сын, правый брат (указатели)

Обход: обратный

Операция: C = A \ B

*****************************************

Группа: БСБО-16-23

Студент: Альзоаби А. Ф.

Вариант: 45

*/


#include <algorithm>

#include <functional>

#include <iostream>

#include <map>

#include <sstream>

#include <stack>

#include <string>

#include <unordered_map>
```

```cpp
#include <unordered_set>

#include <utility>

#include <vector>


template <typename T>

class Tree {

private:

    enum Mark { PHYSICAL, LOGICAL_DELETED, VIRTUAL };


    struct Node {

        Node* leftSon, * rightSibling, * parent;

        T value;

        unsigned level;

        Mark state;


        Node(const T& v) : leftSon(rightSibling = parent = nullptr), value(v), level(1),
state(PHYSICAL) {}

    };


    std::unordered_map<T, size_t> frequencies;

    double P, W, H;


    void calculateP(const Node* node) {

        if (!node) return;


        calculateP(node->leftSon);

        calculateP(node->rightSibling);


        if (node->state == PHYSICAL) P += node->level * frequencies[node->value];

    }
```

```
    void calculateH() {
        calculateP(root);

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

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

public:
    double getH() {
        calculateH();

        return H;
    }

    void printTree(const Node* root) const {
        if (!root) return;

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

        std::function<void(const Node*, int, int, int, std::map<int, std::string>&)>
buildTreeLines =
            [&](const Node* node, int depth, int position, int offset, std::map<int,
std::string>& levels) {
            if (!node || node->state != PHYSICAL) return;

            std::ostringstream oss;
            oss << node->value;
            std::string value = oss.str() + std::to_string(frequencies.at(node->value));

            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;
            }
        }

        int spacing = std::max(2, offset / 2);
        int leftPosition = position - spacing;
        int rightPosition = position + spacing;

        if (node->leftSon) {
            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->leftSon, depth + 2, leftPosition, spacing, levels);
        }

        const Node* sibling = node->leftSon ? node->leftSon->rightSibling : nullptr;
```

```
            int siblingPosition = rightPosition;

            while (sibling) {
                if (static_cast<int>(levels[depth + 1].size()) < siblingPosition) {
                    levels[depth + 1] += std::string(siblingPosition - levels[depth + 1].size(), ' ') + "\\";
                }
                else {
                    levels[depth + 1] += "\\";
                }

                buildTreeLines(sibling, depth + 2, siblingPosition, spacing, levels);
                sibling = sibling->rightSibling;
                siblingPosition += spacing * 3;
            }
            };

        int initialPosition = 40;
        int initialOffset = 30;
        buildTreeLines(root, 0, initialPosition, initialOffset, levels);

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

    Node* ROOT() const { return root; }

private:
    Node* root;

private:
```

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


    void autoPush() {

        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*&, int)> insertNode = [&](Node*& node, const unsigned& currentLevel = 1) {

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

                        if (!node->leftSon) { // ЛЕВОЕ койко-место никем не занято

                            Node* vRightSon = new Node(T()); // виртуальный

                            vRightSon->state = VIRTUAL;

                            vRightSon->level = currentLevel + 1;

                            vRightSon->parent = node;


                            newNode->parent = node;

                            newNode->level = currentLevel + 1;

                            newNode->rightSibling = vRightSon;


                            node->leftSon = newNode;
```

```
                        }
                        else if (node->leftSon->state == VIRTUAL) { // занято волшебным
духом
                            newNode->level = node->leftSon->level;
                            newNode->parent = node->leftSon->parent;
                            newNode->rightSibling = node->leftSon->rightSibling;

                            node->leftSon = newNode;
                        }
                        else {
                            insertNode(node->leftSon, currentLevel + 1);
                        }
                    }
                    else if (newNode->value > node->value) {
                        if (!node->leftSon) { // свобода слева
                            newNode->level = currentLevel + 1;
                            newNode->parent = node;

                            Node* vLeftSon = new Node(T());
                            vLeftSon->state = VIRTUAL;
                            vLeftSon->level = newNode->level;
                            vLeftSon->parent = node;

                            node->leftSon = vLeftSon;
                            vLeftSon->rightSibling = newNode;
                        }
                        else if (!node->leftSon->rightSibling) {
                            newNode->level = node->leftSon->level;
```

```
                        node->leftSon->rightSibling = newNode;
                        node->leftSon->rightSibling->parent = node;
                    }
                    else if (node->leftSon->rightSibling->state == VIRTUAL) {
                        newNode->level = node->leftSon->level;
                        newNode->parent = node->leftSon->parent;
                        node->leftSon->rightSibling = newNode;
                    }
                    else {
                        insertNode(node->leftSon->rightSibling, currentLevel + 1);
                    }
                }
                };

            insertNode(root, 1);
        }
    }
}

Tree CREATE(Node*& node, const std::vector<Tree>& subTrees) {
    if (!node || subTrees.size() > 2 || subTrees.empty()) return Tree<T>();

    Tree<T> newTree;
    newTree.root = node;

    Node* root1 = subTrees[0]->ROOT();
    Node* root2 = nullptr;
    if (subTrees.size() == 2) root2 = subTrees[1]->ROOT();
```

```
        if (root1) root1->parent = node;

        if (root2) root2->parent = node;


        node->leftSon = root1;

        node->leftSon->rightSibling = root2;


        return newTree;

    }


    Node* PARENT(const Node* node) const { return (root && node && node != root) ?
node->parent : nullptr; }

    Node* LEFT_CHILD(const Node* node) const { return (root && node) ? node->leftSon :
nullptr; }

    Node* RIGHT_SIBLING(const Node* node) const { return (root && node) ?
node->rightSibling : nullptr; }

    T LABEL(const Node* node) const { return (root && node) ? node->value : T(); }


    void MAKENULL(Node*& node) {

        if (node) {

            MAKENULL(node->leftSon);

            MAKENULL(node->rightSibling);

            delete node;

            node = nullptr;

        }

    }


public:

    Tree(const std::string& input) : P(W = H = 0.0), root(nullptr) {

        parseInputData(input);

        autoPush();
```

```
    }

    Tree() : P(W = H = 0.0), root(nullptr) {}
    ~Tree() { MAKENULL(root); }

    void printPostorder(const Node* currentNode) const {
        if (!currentNode) return;

        printPostorder(currentNode->leftSon);
        if (currentNode->leftSon) printPostorder(currentNode->leftSon->rightSibling);

        if (currentNode->state == PHYSICAL) std::cout << currentNode->value << " ";
    }



private:
    bool isContains(const Node* node) const {
        if (!node) return false;

        bool found = false;
        const Node* currentNode = ROOT();
        std::function<void(const Node*)> reversePostOrder = [&](const Node* currentNode) {
            if (!currentNode) return;

            reversePostOrder(currentNode->leftSon);
            if (currentNode->leftSon) reversePostOrder(currentNode->leftSon->rightSibling);

            if (currentNode->value == node->value) found = true;
            };
```

```
        reversePostOrder(ROOT());

        return found;
    }

public:
    static void myOperation(const Tree<T>& A, const Tree<T>& B, Tree<T>& C) {
        std::string values;
        std::function<void(const Node*)> reversePostOrder = [&](const Node* currentNode) {
            if (!currentNode) return;

            reversePostOrder(currentNode->leftSon);
            if (currentNode->leftSon) reversePostOrder(currentNode->leftSon->rightSibling);

            if (!B.isContains(currentNode)) values.push_back(currentNode->value);
            };

        reversePostOrder(A.ROOT());
        C.parseInputData(values);
        C.autoPush();
    }

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

        Iterator(const Node* root) : current(nullptr) {
```

```
            if (root) nodes.push(root);
            moveToNext();
        }

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

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

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

    private:
        void moveToNext() {
            while (!nodes.empty()) {
                const Node* node = nodes.top();

                if (visited.find(node) == visited.end()) {
                    visited.insert(node);

                    if (node->leftSon && node->leftSon->rightSibling &&
node->leftSon->rightSibling->state == PHYSICAL) nodes.push(node->leftSon->rightSibling);
                    if (node->leftSon && node->leftSon->state == PHYSICAL)
nodes.push(node->leftSon);
                }
                else {
                    current = node;
                    nodes.pop();
                    return;
```

```
                }

            }


            current = nullptr;

        }

    };


public:

    Iterator begin() const { return Iterator(root); }

    Iterator end() const { return Iterator(nullptr); }

};


int main() {

    setlocale(LC_ALL, "Russian");


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

    const std::string example = "РОВПОВАЕЕКУВИЛРКТОАНАНА";

    Tree<char> A(example);

    A.printTree(A.ROOT());

    std::cout << "\nОбратный обход: "; A.printPostorder(A.ROOT());

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


    ////////////////////////////////////////////////////////////////


    std::cout << "\n\t\t\t\tТестовое дерево (B):\n";

    const std::string test = "АХНАПРОЫСВАООГ";

    Tree<char> B(test);

    B.printTree(B.ROOT());

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

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


    ////////////////////////////////////////////////////////////////


    Tree<char> result;

    Tree<char>::myOperation(example, test, result);

    std::cout << "\n\t\t\t\tРезультативное дерево (C):\n";

    result.printTree(result.ROOT());

    std::cout << "\nОбратный обход: "; result.printPostorder(result.ROOT());

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


    return 0;

}
```

6. Вывод

В этой практической работе я научился работать с деревьями, где используется структура "левый сын - правый брат", что помогает более эффективно представлять и обрабатывать иерархические данные. Я также изучил, как рассчитывать средневзвешенную высоту дерева, используя частоты элементов и уровни узлов. Практическая реализация операции слияния деревьев позволила мне понять важность правильного обхода и организации структуры данных для оптимизации алгоритмов.

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.