МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИИТ)
Кафедра МОиСИТ

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №7.1

«Балансировка дерева поиска»
по дисциплине
«Структуры и алгоритмы обработки данных»

Выполнил студент группы ИКБО-10-24 Бикташев И. И.

Практическую работу выполнил «__»_____2025 г.

«Зачтено» «__»_____2025 г.

Москва 2025

Цель работы: составить программу создания двоичного дерева поиска и реализовать процедуры для работы с деревом согласно варианту.

Тип двоичного дерева: АВЛ-дерево.

Реализуемые алгоритмы: Вставка элемента, балансировка элементов, симметричный обход, обход в ширину, поиск суммы значений листьев, поиск высоты дерева, удаления элемента.

Описание решения: АВЛ-дерево — это сбалансированное двоичное дерево поиска, в котором для каждой его вершины высота её двух поддеревьев различается не более чем на 1. Для поддержания баланса после операций вставки или удаления выполняется поворот вокруг узла, где нарушено условие сбалансированности.

Решение: в качестве звеньев дерева будет использована структура Node (листинг 1)

Листинг 1 - Структура Node

```
// Звено дерева
struct Node {
    char key;
    Node* left;
    Node* right;
    int height;
    Node(char k) : key(k), left(nullptr), right(nullptr), height(1) {}
};

// Поиск высоты поддерева
int height(Node* N) {
    return N ? N->height : 0;
}

// Поиск разницы высот поддеревьев
int getBalance(Node* node) {
    if (!node) return 0;
    return height(node->left) - height(node->right);
}
```

Для балансировки дерева были созданы функции левого и правого повтора поддерева (листинг 2)

Листинг 2 - Повороты поддеревьев

```
// Правый поворот
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* temp = x->right;

    x->right = y;
```

```
    y->left = temp;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}

// Левый поворот
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* temp = y->left;

    y->left = x;
    x->right = temp;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}
```

Сама функция балансировка использует как левый и правый повороты, так и их сочетания, так называемые большой левый и большой правый повороты. В зависимости от разности высот поддерева и его поддеревьев функция балансировки использует один из 4 поворотов для балансировки всего поддерева. (листинг 3)

Листинг 3 - Функция балансировки

```
// Балансировка
Node* balance(Node* node){
    if (!node)
        return nullptr;

    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);

    if (balance > 1 && getBalance(node->left) >= 0)
        return rightRotate(node);

    if (balance < -1 && getBalance(node->right) <= 0)
        return leftRotate(node);

    if (balance > 1 && getBalance(node->left) < 0) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && getBalance(node->right) > 0) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}
```

Функции вставки и удаления элемента из дерева сначала его ищут в бинарном дереве и после, вместе с удалением или вставкой элемента выполняют полную балансировку дерева (листинг 4)

Листинг 4 - Функция вставки и удаления

```
// Вставка звена в поддерево
Node* insert(Node* node, char key) {
    if (!node)
        return new Node(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // равные ключи не допускаются
        return node;

    return balance(node);
}
// Поиск наименьшего звена в поддереве
Node* findMinNode(Node* node) {
    Node* current = node;
    while (current && current->left) {
        current = current->left;
    }
    return current;
}
// Удалить ноду из поддерева
Node* deleteNode(Node* node, char value) {
    if (!node)
        return nullptr;

    // Поиск удаляемого узла
    if (value < node->key) {
        node->left = deleteNode(node->left, value);
    } else if (value > node->key) {
        node->right = deleteNode(node->right, value);
    } else {
        // Узел найден - выполняем удаление
        // Узел с одним потомком или без потомков
        if (!node->left || !node->right) {
            Node* temp = node->left ? node->left : node->right;

            // Нет потомков
            if (!temp) {
                node = nullptr;
            } else { // Один потомок
                *node = *temp; // Копируем содержимое потомка
            }
            delete temp;
        } else { // Узел с двумя потомками
            Node* temp = findMinNode(node->right);
            node->key = temp->key;
            node->right = deleteNode(node->right, temp->key);
        }
    }
    // Балансировка дерева
    return balance(node);
}
```

Были также реализован алгоритмы симметричного обхода, обхода в ширину и поиска суммы значений звеньев (листинг 5)

Листинг 5 - Дополнительные функции

```
// Симметричный обход
void inOrder(Node* root) {
    if (root) {
        inOrder(root->left);
        cout << root->key << ' ';
        inOrder(root->right);
    }
}
// Обход в ширину
void breadthFirst(Node* root) {
    if (!root) return;
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        cout << node->key << ' ';
        if (node->left) q.push(node->left);
        if (node->right) q.push(node->right);
    }
}
// Сумма звеньев
int sumLeafValues(Node* root) {
    if (!root) return 0;
    if (!root->left && !root->right)
        return (int)root->key;
    return sumLeafValues(root->left) + sumLeafValues(root->right);
}
```

Результат работы программы предоставлен на рисунке 1

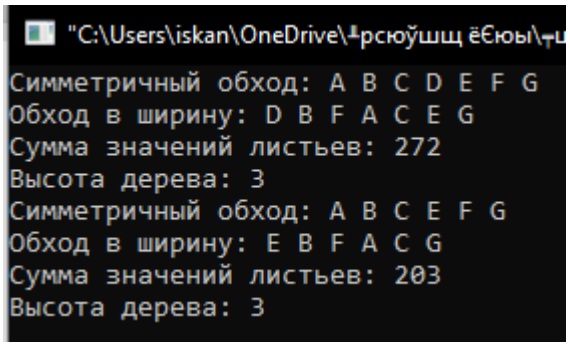

Рисунок 1 - Результат работы программы

Сначала создается АВЛ-дерево и в него добавляются все символы латиницы от A до G и выводится основная информация о дереве. После из дерева удаляется символ D и так же выводится информация о дереве.

Вывод

Поставленные задачи выполнены: изучены и реализованы алгоритмы по работе с АВЛ-деревом, для удаления и вставки в него элементов, алгоритмы симметричного обхода и обхода в ширину элементов дерева.

Литература

Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

Документация по языку C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 31.09.2025)

Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 31.09.2025)