МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению самостоятельной работы № 5

Тема:

«Сбалансированные деревья поиска (СДП) и их применение для поиска данных в файле.»

Дисциплина : «Структуры и алгоритмы обработки данных»

Выполнил студент: Руденко Алексей Дмитиевич

Фамилия И.О

Группа: ИКБО-13-22

Номер группы

Москва - 2023

1 ЦЕЛЬ РАБОТЫ

1.1 Цель:

- получить навыки в разработки и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (АВЛ – деревьями);
- получить навыки в применении файловых потоков прямого доступа к данным файла;
- получить навыки в применении сбалансированного дерева поиска для прямого доступа к записям файла

2 ХОД РАБОТЫ

2.1 Задание 1

2.1.1 Постановка задачи:

Разработать приложение, которое использует БДП для организации прямого доступа к записям файла, структура записи которого: Владельцев автомобилей. номер машины, марка.

Дано: Файл двоичный с записями фиксированной длины.

1. Разработать класс (или библиотеку функций) «Бинарное дерево поиска». Тип информационной части узла дерева: ключ и ссылка на запись в файле (как в практическом задании 2). Методы: включение элемента в дерево, поиск ключа в дереве, удаление ключа из дерева, отображение дерева.

2. Разработать класс (библиотеку функций) управления файлом (если не создали в практическом задании 2). Включить методы: создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле; поиск записи в файле с использованием БДП; остальные методы по вашему усмотрению.

2.1.2 Структура данных:

Изображение файла (рис. 1).

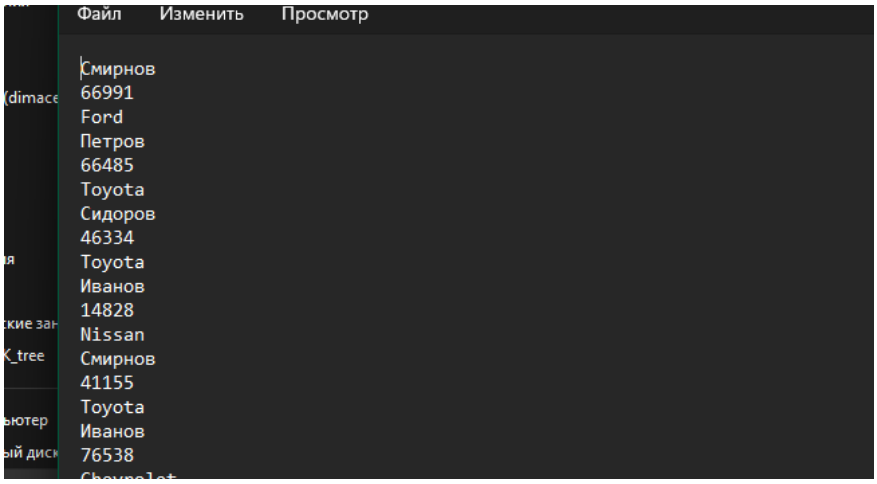

Рисунок 1 – Структура файла с данными.

Реализация структуры данных на языке c++ для хранения в бинарном файле (рис. 2).

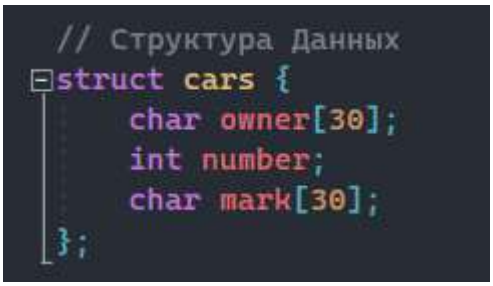

Рисунок 2 – Структура данных фиксированной длины

2.1.3 Подход к решению:

Подход к решению на этом этапе программы включает в себя создание класса **"BinarySearchTree"** и связанных с ним методов для работы с бинарным деревом поиска (БДП). Класс **"BinarySearchTree"** позволяет организовать данные о владельцах автомобилей, используя номер машины в качестве ключа для быстрого поиска и доступа к соответствующим записям в файле.

1. `isInt(const string& str)`: Данный метод проверяет, является ли строка целым числом, используя функцию `isdigit` для проверки каждого символа в строке. Этот метод предназначен для валидации входных данных.

2. `moveForwardNLines(ifstream& file, int n)`: Этот метод перемещает указатель файла на n строк вперед. Он используется, например, для пропуска ненужных строк при чтении файла.

3. `struct Node`: Эта структура представляет узел бинарного дерева поиска. Она содержит поля для номера машины (carNumber), ключа (key), указателей на левого и правого потомков. Каждый узел дерева связан с записью в файле через ключ (позицию в файле).

4. `class BinarySearchTree`: Этот класс представляет бинарное дерево поиска. Он содержит методы для вставки элементов, поиска по ключу,

удаления элементов и отображения дерева. Каждый узел в дереве содержит информацию о номере машины и ключе, который соответствует позиции записи в файле.

5. `getRoot()`: Метод для получения указателя на корень дерева. Это может быть полезно для выполнения операций с деревом.

6. `searchRecursive()`: Рекурсивный метод для поиска ключа (позиции в файле) по номеру машины в бинарном дереве. Он выполняет сравнение ключей и перемещение по дереву в соответствии с результатами сравнения.

7. `insertRecursive()`: Рекурсивный метод для вставки элемента в бинарное дерево. Он также выполняет сравнение ключей для определения, в какую сторону дерева нужно вставить элемент.

8. `removeRecursive()`: Рекурсивный метод для удаления элемента из бинарного дерева. Он осуществляет поиск удаляемого элемента и выполняет операции в зависимости от наличия потомков.

9. `findMinNode()`: Метод для нахождения узла с минимальным ключом (позицией в файле) в поддереве. Этот узел используется при удалении узла с двумя потомками.

10. `display_tree()`: Метод для отображения бинарного дерева в виде дерева, с корнем сверху и визуализацией левых и правых поддеревьев. Это полезно для отладки и визуализации структуры дерева.

Общий подход в этом коде - использование бинарного дерева поиска для эффективного поиска и доступа к данным, организованным в файле. Этот класс и связанные методы предоставляют функциональность для работы с данными о владельцах автомобилей и позволяют выполнять операции вставки, поиска и удаления данных в дереве.

Подход к решению, связанный с бинарными файлами, включает в себя создание функций и структур данных для работы с двоичными файлами, а также реализацию методов для чтения, записи, и удаления записей в файле. Основной упор делается на обработку структуры данных "cars", представляющей информацию о владельцах автомобилей, номерах машин и их марках.

Ключевые элементы подхода к решению:

1. Структура данных "cars":

- В структуре "cars" представлена информация о владельцах автомобилей. Эта структура включает в себя поля для имени владельца, номера машины и марки машины.

2. Создание бинарного файла из текстового:

- Метод `create_bin_file` открывает текстовый файл для чтения и двоичный файл для записи в бинарном режиме. Затем он считывает данные из текстового файла, извлекает информацию о владельцах автомобилей и записывает их в двоичный файл, соответствуя структуре "cars".

3. Вывод записей двоичного файла:

- Метод `print_bin_file` открывает двоичный файл для чтения в бинарном режиме и считывает записи из файла, выводя информацию о владельцах автомобилей, номерах машин и марках на экран.

4. Добавление записи в двоичный файл:

- Метод `add_bin_file` открывает двоичный файл для записи в бинарном режиме с флагом `ios::app`, что позволяет добавлять записи в конец файла. Затем он запрашивает у пользователя информацию о новом владельце автомобиля, номере машины и марке, и добавляет новую запись в файл.

5. Подсчет элементов в бинарном файле:

- Метод `count_elements` подсчитывает количество записей в двоичном файле, опираясь на размер файла и размер каждой записи.

6. Зануление лицензионного номера по индексу:

- Метод `DeleteAtIndex` позволяет занулить (обнулить) лицензионный номер в записи по указанному индексу.

7. Вывод структуры по индексу:

- Метод `printAtIndex` позволяет вывести информацию о владельце автомобиля, номере машины и марке, находящуюся в записи по указанному индексу в файле.

8. Получение индекса последней записи:

- Метод `getLastRecordIndex` определяет индекс последней записи в файле, что может быть полезным при добавлении новых записей.

Этот подход к решению позволяет работать с данными о владельцах автомобилей в бинарных файлах, а также выполнять различные операции, такие как чтение, запись, удаление и поиск записей. Он также предоставляет некоторые дополнительные функции для управления данными в файлах, такие как подсчет записей и вывод информации о записях по индексу.

2.1.4 Алгоритмы операций на псевдокоде

Вставка элемента в БДП:

```
procedure insert(node, key, file_record)
    if node is null
        create a new node with key and file_record
        return the new node
    if key < node.key
        node.left = insert(node.left, key, file_record)
    else if key > node.key
        node.right = insert(node.right, key, file_record)
    return node
end procedure
```

Поиск записи по ключу в БДП и возврат ссылки на запись в файле:

```
function search(node, key)
    if node is null
        return null // Ключ не найден
    if key = node.key
        return node.file_record
    else if key < node.key
        return search(node.left, key)
    else
        return search(node.right, key)
end function
```

Удаление элемента из БДП:

```
procedure delete(node, key)
    if node is null
        return node
    if key < node.key
        node.left = delete(node.left, key)
    else if key > node.key
        node.right = delete(node.right, key)
    else
        // Узел с ключом равным key найден, выполняем удаление
        if node.left is null
            temp = node.right
            destroy node
            return temp
        else if node.right is null
            temp = node.left
            destroy node
            return temp
        // У узла есть два потомка, находим преемника (например, минимальный ключ в
правом поддереве)
        temp = findMinNode(node.right)
        // Копируем данные преемника в текущий узел
        node.key = temp.key
        // Удаляем преемника из правого поддерева
        node.right = delete(node.right, temp.key)
    return node
end procedure

function findMinNode(node)
    while node.left is not null
        node = node.left
    return node
end function
```

Эти алгоритмы позволяют вставлять, искать и удалять элементы в бинарном дереве поиска, где каждый узел содержит ключ и ссылку на запись в файле. Поиск элемента осуществляется с использованием ключа, и при успешном поиске возвращается ссылка на соответствующую запись в файле. Удаление элемента может потребовать обработки нескольких случаев, включая удаление узла с одним или двумя потомками.

2.1.5 Код приложения:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

// Структура Данных
struct LibraryNote {
    char fio[30];
    int number;
    char address[30];
};

int add_bin_file(string filename) {
    ofstream file(filename, ios::out | ios::binary | ios::app);

    LibraryNote note;
    cout << "Введите информацию о читальском абонименте:\n";
    cout << "ФИО: ";
    cin.ignore();
    cin.getline(note.fio, sizeof(note.fio));

    cout << "Номер (5 цифр): ";
    cin >> note.number;

    cout << "Адрес: ";
    cin.ignore();
    cin.getline(note.address, sizeof(note.address));

    file.write(reinterpret_cast<const char*>(&note), sizeof(LibraryNote));
    file.close();

    return note.number; // Возвращаем номер читательского абонемента
}

// Функция для получения индекса последней записи в бинарном файле
int getLastRecordIndex(string filename) {
    ifstream file(filename, ios::binary | ios::ate);
    if (!file.is_open()) {
        return 0; // Если файл не открыт, возвращаем 0
    }

    int size = file.tellg(); // Получаем текущую позицию (размер файла)
    file.close();

    int recordSize = sizeof(LibraryNote); // Размер одной записи
    return size / recordSize; // Возвращаем индекс последней записи
}

// Функция для вывода информации о машине по указанному индексу в бинарном файле
void printAtIndex(string filename, int index) {
    ifstream file(filename, ios::binary);
    if (!file.is_open()) {
        cout << "Ошибка при открытии файла.\n";
        return;
    }

    LibraryNote note;
    file.seekg(index * sizeof(LibraryNote)); // Перемещаем указатель на нужную
позицию
    file.read(reinterpret_cast<char*>(&note), sizeof(LibraryNote));

    cout << "Информация о машине с индексом " << index + 1 << ":\n";
```

```
    cout << "ФИО: " << note.fio << endl;
    cout << "Номер: " << note.number << endl;
    cout << "Адрес: " << note.address << endl;

    file.close();
}

// Функция для вывода содержимого бинарного файла с информацией об абонементах
void print_bin_file(string filename) {
    ifstream file(filename, ios::binary);
    if (!file.is_open()) {
        cout << "Ошибка при открытии файла.\n";
        return;
    }

    LibraryNote note;
    cout << "Содержимое бинарного файла с информацией об абонементах:\n";
    while (file.read(reinterpret_cast<char*>(&note), sizeof(LibraryNote))) {
        cout << "ФИО: " << note.fio << " Номер: " << note.number << " Адрес: " <<
note.address << endl;
    }

    file.close();
}

// Функция для создания бинарного файла с информацией об абонементах из текстового
файла
void create_bin_file(string fText, string fBin) {
    ifstream inputFile(fText);
    if (!inputFile.is_open()) {
        cout << "Ошибка при открытии текстового файла.\n";
        return;
    }

    ofstream outputFile(fBin, ios::binary);
    if (!outputFile.is_open()) {
        cout << "Ошибка при создании бинарного файла.\n";
        return;
    }

    LibraryNote note;
    while (inputFile.getline(note.fio, sizeof(note.fio)) && inputFile >> note.number
&& inputFile.ignore() && inputFile.getline(note.address, sizeof(note.address))) {
        outputFile.write(reinterpret_cast<const char*>(&note), sizeof(LibraryNote));
    }

    inputFile.close();
    outputFile.close();
    cout << "Бинарный файл успешно создан из текстового файла.\n";
}

// Подсчёт элементов в бинарном файле
int count_elements(string filename) {
    ifstream fb(filename, ios::binary);
    if (!fb) {
        cerr << "Unable to open the binary file." << endl;
        return -1; // Возвращаем -1 в случае ошибки
    }

    fb.seekg(0, ios::end); // Перемещаемся в конец файла
    streampos fileSize = fb.tellg(); // Получаем размер файла
    fb.seekg(0, ios::beg); // Перемещаемся в начало файла

    int count = 0;
    while (fb.tellg() < fileSize) {
```

```
        cars x;
        if (fb.read(reinterpret_cast<char*>(&x), sizeof(cars))) {
            count++;
        }
    }

    fb.close();
    return count;
}

// Функция для зануления лицензионного номера по индексу
void DeleteAtIndex(const string& filename, int index) {
    // Открываем бинарный файл для чтения и записи
    fstream file(filename, ios::in | ios::out | ios::binary);

    if (!file) {
        cerr << "Ошибка открытия файла" << endl;
    }

    // Определяем размер одной записи
    size_t record_size = sizeof(struct cars);

    // Перемещаем указатель файла к нужной записи
    file.seekp(index * record_size + sizeof(char) * 30, ios::beg);

    // Зануляем лицензионный номер в записи
    int zero = 0;
    file.write(reinterpret_cast<char*>(&zero), sizeof(int));

    file.close();
}

//====================TREE
bool isInt(const string& str) {
    for (char c : str) {
        if (!isdigit(c)) {
            return false;
        }
    }
    return true;
}

void moveForwardNLines(ifstream& file, int n) {
    for (int i = 0; i < n; ++i) {
        string line;
        if (!getline(file, line)) {
            // Выход из цикла, если достигнут конец файла раньше
            break;
        }
    }
}

// Структура для узла бинарного дерева поиска
struct Node {
    string Number; // Номер
    int key; // Позиция в файле
    Node* left;
    Node* right;

    Node(const string& number, int k) : Number(number), key(k), left(nullptr),
right(nullptr) {}
};

class BinarySearchTree {
public:
```

```
    BinarySearchTree() : root(nullptr) {}

    // Метод для включения элемента в дерево
    void insert(const string& carNumber, int key) {
        root = insertRecursive(root, carNumber, key);
    }

    // Метод для поиска ключа по номеру машины
    int search(const string& Number) {
        return searchRecursive(root, carNumber);
    }

    void print(Node* node, int level = 0) {
        return display_tree(node, level);
    }

    // Метод для получения указателя на корень дерева
    Node* getRoot() {
        return root;
    }

    Node* insert(Node* current, const string& carNumber, int key) {
        return insertRecursive(current, Number, key);
    }

    Node* remove(Node* current, const string& carNumber) {
        return removeRecursive(current, Number);
    }

private:
    Node* root;

    // Рекурсивный метод для поиска ключа по номеру
    int searchRecursive(Node* current, const string& Number) {
        if (current == nullptr) {
            return -1; // Номер не найден
        }

        if (Number == current->carNumber) {
            return current->key; // Возвращаем ключ (индекс)
        }

        else if (Number < current->carNumber) {
            return searchRecursive(current->left, Number);
        }

        else {
            return searchRecursive(current->right, Number);
        }
    }

    // Рекурсивный метод для включения элемента в дерево
    Node* insertRecursive(Node* current, const string& Number, int key) {
        if (current == nullptr) {
            return new Node(Number, key);
        }

        if (carNumber < current->carNumber) {
            current->left = insertRecursive(current->left, Number, key);
        }
        else if (carNumber > current->carNumber) {
            current->right = insertRecursive(current->right, Number, key);
        }

        return current;
```

```
    }

    // Рекурсивный метод для удаления ключа из дерева
    Node* removeRecursive(Node* current, const string& carNumber) {
        if (current == nullptr) {
            return current;
        }

        if (carNumber < current->carNumber) {
            current->left = removeRecursive(current->left, carNumber);
        }
        else if (carNumber > current->carNumber) {
            current->right = removeRecursive(current->right, carNumber);
        }
        else {
            // Найден ключ (номер) который нужно удалить
            if (current->left == nullptr) {
                Node* temp = current->right;
                delete current;
                return temp;
            }
            else if (current->right == nullptr) {
                Node* temp = current->left;
                delete current;
                return temp;
            }

            // У узла есть два детей
            Node* temp = findMinNode(current->right);
            current->carNumber = temp->carNumber;
            current->right = removeRecursive(current->right, temp->carNumber);
        }

        return current;
    }

    Node* findMinNode(Node* node) {
        while (node->left != nullptr) {
            node = node->left;
        }
        return node;
    }


    void display_tree(Node* node, int level = 0) {
        if (node == nullptr) {
            if (level == 0) {
                cout << "Дерево было удалено." << endl;
            }
            return;
        }

        display_tree(node->right, level + 1);
        cout << string(level * 4, ' ') << "-> " << node->carNumber << endl;
        display_tree(node->left, level + 1);
    }

};

int main() {
    setlocale(LC_ALL, "Russian");
    string filename = "LibraryNote.txt";
    BinarySearchTree tree;

    while (true)
```

```
    {

        int c;
        cout << "\n============================== \
            \n[ 1 ] - Создать дерево \
            \n[ 2 ] - Найти владельца по номеру \
            \n 3 ] - Вывести файл \
            \n 4 ] - Показать дерево \
            \n 5 ] - Добавить узел \
            \n 6] - Удалить узел \
            \n 0 ]  Выход\n==============================\n>";
        cin >> c;

        switch (c)
        {
        case 1:
        {
            int key = 0;
            string Number;
            string line;
            // Открываем файл для чтения
            ifstream file(filename);
            if (!file.is_open()) {
                cerr << "\nFailed to open file." << endl;
                return 1;
            }

            // Созадме дерево
            while (getline(file, line)) {
                if (isInt(line))
                {
                    Number = line;
                    key++;
                    tree.insert(Number, key);
                }

            }
            file.close();
            cout << "\nДерево создано.";
            break;
        }
        case 2:
        {
            string searchNumber;
            string line;
            string line2;
            string Number;

            cout << "\nВведите номер машины: ";
            cin >> searchNumber;

            // Поиск номера машины
            int searchKey = tree.search(searchNumber);

            // if для поулчения данных по ключу из файла и вывода
            if (searchKey != -1) {
                cout << "\nНомер " << searchNumber << " найден на позиции " <<
searchKey << endl;

                // Теперь вы можете открыть файл с данными и перейти к нужной
позиции (ключу) для получения данных о владельце машины
                ifstream dataFile("LibraryNote.txt");
                moveForwardNLines(dataFile, searchKey * 3 - 3); // Смещаем к индексу
в файле
                getline(dataFile, line2); // Считать данные о марке машины
```

```
                getline(dataFile, line);
                getline(dataFile, line); // Считать данные о владельце машины
                cout << "О Владельце: " << line << ", " << line2 << endl;
                dataFile.close();
            }
            else {
                cout << "\nNumber " << searchNumber << " not found in the tree." <<
endl;
            }

            break;

        }
        case 3:
        {
            // Открываем файл для чтения
            ifstream file(filename);
            if (!file.is_open()) {
                cerr << "\nFailed to open file." << endl;
                return 1;
            }

            string line;

            cout << endl;

            // Выводим файл в консоль
            while (getline(file, line)) {
                cout << line << endl;
            }
            break;
        }
        case 4:
        {
            tree.print(tree.getRoot());
            break;
        }
        case 5:
        {
            string num;
            string name;
            string addres;
            cout << "\nВведите номер: ";
            cin >> num;
            cout << "\nВведите имя владельца: ";
            cin >> name;
            cout << "\nВведите адрес: ";
            cin >> address;

            // Добавляем в файл:
            ofstream file(filename, ios::app); // Открываем файл для добавления
данных (append)

            if (!file.is_open()) {
                cerr << "Не удалось открыть файл." << endl;
                return 1;
            }

            // Записываем строку в конец файла
            file << name << endl;
            file << num << endl;
            file << mark << endl;
            file.close();

            ifstream file2(filename);
```

```
            if (!file2.is_open()) {
                cerr << "\nFailed to open file." << endl;
                return 1;
            }

            string line;
            int cnt = 0;
            // Создаме дерево
            while (getline(file2, line)) {
                cnt++;
            }

            file2.close();
            tree.insert(tree.getRoot(), num, cnt);

            cout << "\nДобавлен!";
            break;
        }
        case 0:break;
        default:
            break;
        }
    }
}
```

2.1.6 Тестирование:

Построенное дерево для файла показанного выше (рис. 3).

```
================================
1.Создать дерево
2.Найти владельца по номеру
3.Вывести файл
4.Показать дерево
5.Добавить узел
6.Удалить узел
0.Выход
================================
>4
        -> 80814
            -> 77358
    -> 76538
        -> 67641
-> 66991
    -> 66485
            -> 61710
        -> 46334
                -> 41155
            -> 14828
                -> 1234
```

Рисунок 3 – Меню пользователя и построенное дерево

Поиск элемента в дереве с дальнейшим обращением к файлу за данными (рис. 4).

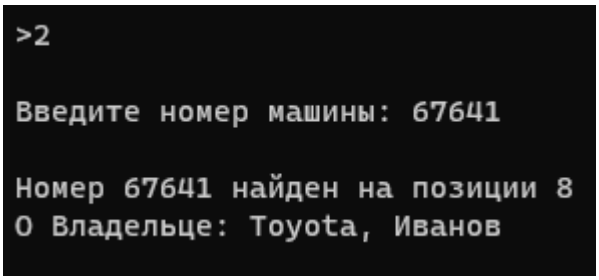

Рисунок 4 – Поиск

Добавление нового узла (рис. 5-6).

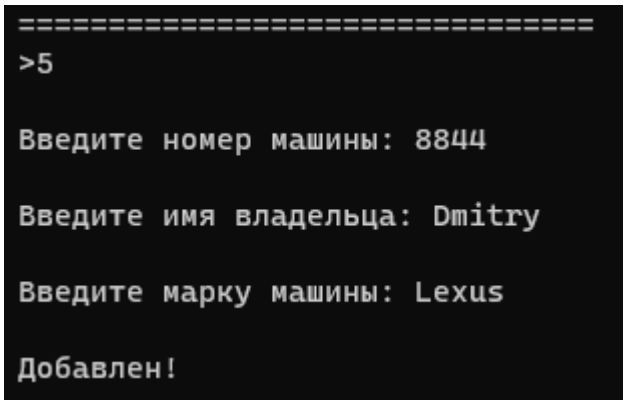

Рисунок 5 – Процесс добавления узла

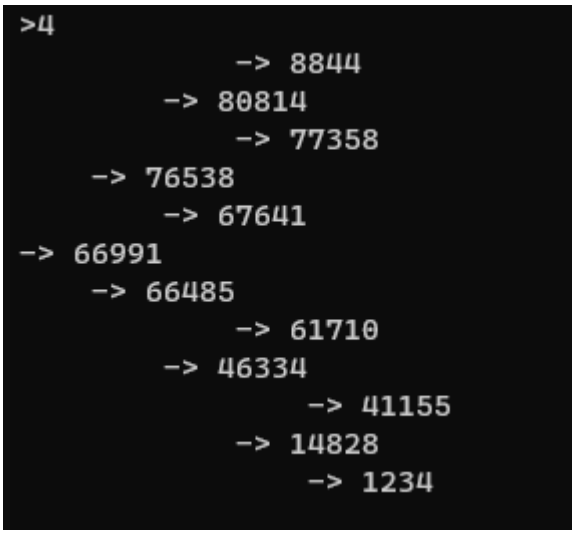

Рисунок 6 – Новый узел в дереве

Удаление узла (рис. 7).

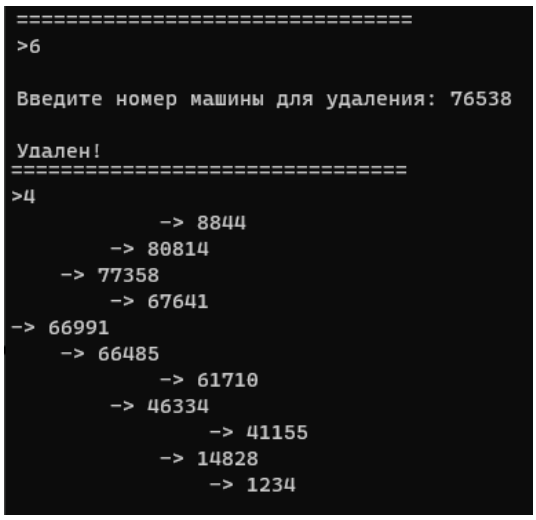

Рисунок 7 – Удаление узла

2.2 Задание 2

2.2.1 Постановка задачи:

Разработать приложение, которое использует сбалансированное дерево поиска (Красно - черное), для доступа к записям файла.

1. Разработать класс СДП с учетом структуры красно-черного дерева. Структура информационной части узла дерева включает ключ и ссылку на запись в файле (адрес места размещения). Основные методы: включение элемента в дерево; поиск ключа в дереве с возвратом ссылки; удаление ключа из дерева; вывод дерева в форме дерева (с отображением структуры дерева).

2. Разработать приложение, которое создает и управляет СДП в соответствии с заданием.

3. Выполнить тестирование.

4. Определить среднее число выполненных поворотов (число поворотов на общее число вставленных ключей) при включении ключей в дерево при формировании дерева из двоичного файла.

2.2.2 Структура данных:

Бинарное дерево является сбалансированным тогда и только тогда, когда для каждого узла ВЫСОТА его двух поддеревьев различается не более чем на 1.

КЧ-деревья (рис. 8) – это двоичные деревья поиска, каждый узел которых хранит дополнительное поле color , обозначающее цвет: красный или черный, и для которых выполнены приведенные ниже свойства. Будем считать, что если left или right равны NULL, то это «указатели» на фиктивные листья. В КЧ-дереве все узлы – внутренние (нелистовые).

Свойства красно-черных деревьев:

1) Каждый узел окрашен либо в красный, либо в черный цвет (в структуре данных узла появляется дополнительное поле – бит цвета).

2) Корень окрашен в черный цвет.

3) Листья (так называемые NULL-узлы) окрашены в черный цвет.

4) Каждый красный узел должен иметь два черных дочерних узла. У черного узла могут быть черные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные.

5) Пути от узла к его листьям должны содержать одинаковое количество черных узлов (это черная высота).

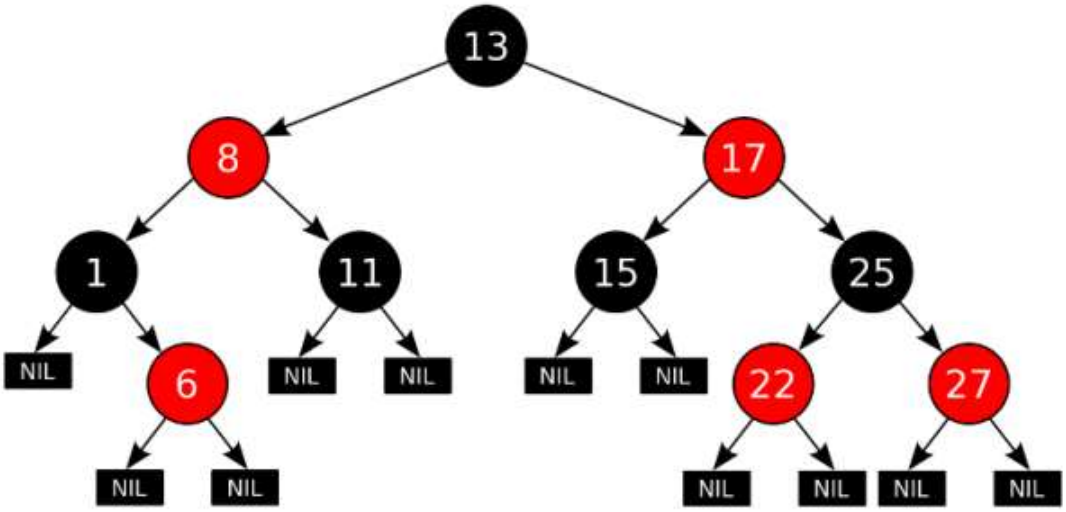

Рисунок 8 – Структура красно-черного дерева

Красно-черные деревья не гарантируют строгой сбалансированности (разница высот двух поддеревьев любого узла не должна превышать 1), как в АВЛ-деревьях. Но соблюдение свойств красно-черного дерева позволяет обеспечить выполнение операций вставки, удаления и выборки за время $O(\log N)$

2.2.3 Подход к решению:

Для решения задачи, необходимо создать класс **`RedBlackTree`**, представляющий собой реализацию красно-чёрного дерева. В данном дереве каждый узел хранит данные (значение) и ключ (позицию в файле), а также имеет цвет (красный или черный), указатели на родителя, левого и правого потомка.

Основные методы класса **`RedBlackTree`** включают:

1. `insert(int value, int key, int& rotations)`: Метод для вставки нового элемента в дерево. Создается новый узел, и в зависимости от значений

вставляемого элемента, он размещается в дереве. Затем выполняется балансировка с помощью метода `fixInsert`, чтобы сохранить свойства красно-чёрного дерева.

2. `remove(int value)`: Метод для удаления элемента из дерева по значению. Если элемент с таким значением найден, он удаляется, и в случае необходимости выполняется балансировка с помощью метода `fixDelete`.

3. `search(int value)`: Метод для поиска элемента в дереве по его значению. Возвращает указатель на найденный узел или `nullptr`, если элемент не найден.

4. `inorderTraversal()`: Метод для выполнения инфиксного обхода дерева и вывода элементов на экран в отсортированном порядке.

5. `print(Node* node, int mode, int level)`: Метод для вывода структуры дерева в форме дерева с отображением цветов узлов. Параметры `mode` и `level` используются для форматирования вывода.

6. `getRoot()`: Метод для получения указателя на корень дерева.

Данный класс `RedBlackTree` предоставляет реализацию сбалансированного дерева поиска, которое может быть использовано для доступа к записям в файле, где ключи соответствуют позициям в файле. Реализация красно-чёрного дерева обеспечивает эффективное выполнение операций вставки, поиска и удаления элементов, а также поддерживает баланс дерева для обеспечения оптимальной производительности.

2.2.4 Алгоритмы операций на псевдокоде:

Вставка в КЧД (Insertion):

```
Insert(root, key, data) :
    if root is null :
Create a new red node with keyand data
Set the new node as the root(make it black)
```

```
    else if key < root.key :
        Recursively insert into the left subtree
    else if key > root.key:
Recursively insert into the right subtree
    else:
Update data for the existing node(optional)

FixInsert(root) // После вставки, выполняем балансировку

FixInsert(node) :
    while node.parent is red :
if node.parent is the left child of node.parent.parent :
    uncle = node.parent.parent's right child
    if uncle is red :
Set node.parentand uncle as black
Set node.parent.parent as red
Set node as node.parent.parent
    else:
if node is the right child of node.parent :
    node = node.parent
    RotateLeft(node)
    Set node.parent as black
    Set node.parent.parent as red
    RotateRight(node.parent.parent)
else:
(Symmetric case for right child)

Set root as black

RotateLeft(node) :
    rightChild = node.right
    node.right = rightChild.left
    if rightChild.left is not null :
        rightChild.left.parent = node
        rightChild.parent = node.parent
        if node.parent is null :
Set root as rightChild
        else if node is the left child of node.parent :
            node.parent.left = rightChild
        else:
node.parent.right = rightChild
rightChild.left = node
node.parent = rightChild

RotateRight(node) :
    (Symmetric to RotateLeft)
```

Удаление из КЧД (Deletion):

```
Delete (root, key):
    node = Search for the node with the given key
    if node is null:
        The key was not found, return

    replacement = node
    isBlack = replacement is black
    if node.left is null:
        replacement = node.right
        Transplant(node, node.right)
    else if node.right is null:
        replacement = node.left
```

```
        Transplant(node, node.left)
    else:
        successor = Find the successor (minimum node in the right subtree)
        replacement = successor
        isBlack = successor is black
        if successor is not node.right:
            Transplant(successor, successor.right)
            successor.right = node.right
            successor.right.parent = successor
        Transplant(node, successor)
        successor.left = node.left
        successor.left.parent = successor
        successor.color = node.color

    if isBlack:
        FixDelete(replacement)

FixDelete(node):
    while node is not root and node is black:
        if node is the left child of node.parent:
            sibling = node.parent's right child
            if sibling is red:
                Set sibling as black
                Set node.parent as red
                RotateLeft(node.parent)
                sibling = node.parent's right child
            if sibling.left is black and sibling.right is black:
                Set sibling as red
                Set node as node.parent
            else:
                if sibling.right is black:
                    Set sibling.left as black
                    Set sibling as red
                    RotateRight(sibling)
                    sibling = node.parent's right child
                Set sibling's color as node.parent's color
                Set node.parent as black
                Set sibling.right as black
                RotateLeft(node.parent)
                Set node as root
        else:
            (Symmetric case for right child)

    Set node as black

Transplant(u, v):
    if u.parent is null:
        Set root as v
    else if u is u.parent's left child:
        u.parent.left = v
    else:
        u.parent.right = v
    if v is not null:
        v.parent = u.parent
```

Поиск в КЧД (Search):

```
Search (root, key):
    if root is null or root.key is equal to key:
        return root
    if key < root.key:
        return Search(root.left, key)
    return Search(root.right, key)
```

2.2.5 Код приложения:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

// Структура Данных
struct LibraryNote {
    char fio[30];
    int number;
    char address[30];
};

// Перечисление для представления цветов узлов дерева
enum Color { RED, BLACK };

// Структура для представления узла красно-чёрного дерева
struct Node {
    int data;        // Значение узла
    int key;         // Позция в файле
    Color color;     // Цвет узла (красный или черный)
    Node* parent;    // Указатель на родительский узел
    Node* left;      // Указатель на левого потомка
    Node* right;     // Указатель на правого потомка
};

// Функция для добавления записи в бинарный файл
int add_bin_file(string filename) {
    ofstream file(filename, ios::out | ios::binary | ios::app);

    LibraryNote note;
    cout << "Введите информацию о читальском абонименте:\n";
    cout << "ФИО: ";
    cin.ignore();
    cin.getline(note.fio, sizeof(note.fio));

    cout << "Номер (5 цифр): ";
    cin >> note.number;

    cout << "Адрес: ";
    cin.ignore();
    cin.getline(note.address, sizeof(note.address));
```

```
    file.write(reinterpret_cast<const char*>(&note), sizeof(LibraryNote));
    file.close();

    return note.number; // Возвращаем номер читательского абонемента
}

// Функция для получения индекса последней записи в бинарном файле
int getLastRecordIndex(string filename) {
    ifstream file(filename, ios::binary | ios::ate);
    if (!file.is_open()) {
        return 0; // Если файл не открыт, возвращаем 0
    }

    int size = file.tellg(); // Получаем текущую позицию (размер файла)
    file.close();

    int recordSize = sizeof(LibraryNote); // Размер одной записи
    return size / recordSize; // Возвращаем индекс последней записи
}

// Функция для вывода информации о машине по указанному индексу в бинарном файле
void printAtIndex(string filename, int index) {
    ifstream file(filename, ios::binary);
    if (!file.is_open()) {
        cout << "Ошибка при открытии файла.\n";
        return;
    }

    LibraryNote note;
    file.seekg(index * sizeof(LibraryNote)); // Перемещаем указатель на нужную позицию
    file.read(reinterpret_cast<char*>(&note), sizeof(LibraryNote));

    cout << "Информация о машине с индексом " << index + 1 << ":\n";
    cout << "ФИО: " << note.fio << endl;
    cout << "Номер: " << note.number << endl;
    cout << "Адрес: " << note.address << endl;

    file.close();
}

// Функция для вывода содержимого бинарного файла с информацией об абонементах
void print_bin_file(string filename) {
    ifstream file(filename, ios::binary);
```

```
    if (!file.is_open()) {
        cout << "Ошибка при открытии файла.\n";
        return;
    }

    LibraryNote note;
    cout << "Содержимое бинарного файла с информацией об абонементах:\n";
    while (file.read(reinterpret_cast<char*>(&note), sizeof(LibraryNote))) {
        cout << "ФИО: " << note.fio << " Номер: " << note.number << " Адрес: " <<
note.address << endl;
    }

    file.close();
}

// Функция для создания бинарного файла с информацией об абонементах из текстового файла
void create_bin_file(string fText, string fBin) {
    ifstream inputFile(fText);
    if (!inputFile.is_open()) {
        cout << "Ошибка при открытии текстового файла.\n";
        return;
    }

    ofstream outputFile(fBin, ios::binary);
    if (!outputFile.is_open()) {
        cout << "Ошибка при создании бинарного файла.\n";
        return;
    }

    LibraryNote note;
    while (inputFile.getline(note.fio, sizeof(note.fio)) && inputFile >> note.number &&
inputFile.ignore() && inputFile.getline(note.address, sizeof(note.address))) {
        outputFile.write(reinterpret_cast<const char*>(&note), sizeof(LibraryNote));
    }

    inputFile.close();
    outputFile.close();
    cout << "Бинарный файл успешно создан из текстового файла.\n";
}



// Класс, представляющий красно-чёрное дерево
class RedBlackTree {
public:
```

```
    RedBlackTree() : root(nullptr) {}

    // Метод для вставки значения в дерево
    void insert(int value, int key, int& rotations) {
        Node* node = new Node{ value, key,  RED, nullptr, nullptr, nullptr };
        if (root == nullptr) {
            root = node;
            root->color = BLACK;
        }
        else {
            insertNode(root, node);
            fixInsert(node, rotations);
        }
    }

    // Метод для удаления значения из дерева
    void remove(int value) {
        Node* node = search(value);
        if (node == nullptr) {
            cout << "Node with value " << value << " not found in the tree." << endl;
            return;
        }
        deleteNode(node);
    }

    // Метод для поиска значения в дереве
    Node* search(int value) {
        return searchN(value, root);
    }

    // Метод для выполнения инфиксного обхода дерева
    void inorderTraversal() {
        inorderTraversal(root);
        cout << endl;
    }

    // Метод вывода
    void print(Node* node, int mode, int level = 0) {
        return display_tree(node, mode, level);
    }

    // Метод для получения указателя на корень дерева
    Node* getRoot() {
```

```
        return root;
    }

private:
    Node* root;

    // Вставка узла в дерево
    void insertNode(Node* root, Node* node) {
        if (node->data < root->data) {
            if (root->left != nullptr) {
                insertNode(root->left, node);
                return;
            }
            else {
                root->left = node;
                node->parent = root;
            }
        }
        else {
            if (root->right != nullptr) {
                insertNode(root->right, node);
                return;
            }
            else {
                root->right = node;
                node->parent = root;
            }
        }
    }



    // Балансировка дерева после вставки
    void fixInsert(Node* node, int& rotations) {
        // Пока текущий узел не является корнем и цвет его родителя RED (красный),
выполняем балансировку
        while (node != root && node->parent->color == RED) {
            if (node->parent == node->parent->parent->left) {
                // Если родитель текущего узла - левый потомок своего родителя (случай
"дядя справа")
                Node* uncle = node->parent->parent->right; // Получаем указатель на дядю
справа

                if (uncle != nullptr && uncle->color == RED) {
                    // Если дядя RED, выполняем перекраску и продвигаемся вверх по
дереву
```

```
                    node->parent->color = BLACK;
                    uncle->color = BLACK;
                    node->parent->parent->color = RED;
                    node = node->parent->parent;
                }
                else {
                    // Если дядя BLACK или nullptr, выполняем соответствующее вращение и
перекраску
                    if (node == node->parent->right) {
                        node = node->parent;
                        rotateLeft(node); rotations++;
                    }
                    node->parent->color = BLACK;
                    node->parent->parent->color = RED;
                    rotateRight(node->parent->parent); rotations++;
                }
            }
            else { // Если родитель текущего узла - правый потомок своего родителя
(случай "дядя слева")
                Node* uncle = node->parent->parent->left; // Получаем указатель на дядю
слева

                if (uncle != nullptr && uncle->color == RED) {
                    // Если дядя RED, выполняем перекраску и продвигаемся вверх по
дереву
                    node->parent->color = BLACK;
                    uncle->color = BLACK;
                    node->parent->parent->color = RED;
                    node = node->parent->parent;
                }
                else {
                    // Если дядя BLACK или nullptr, выполняем соответствующее вращение и
перекраску
                    if (node == node->parent->left) {
                        node = node->parent;
                        rotateRight(node); rotations++;
                    }
                    node->parent->color = BLACK;
                    node->parent->parent->color = RED;
                    rotateLeft(node->parent->parent); rotations++;
                }
            }
        }
        root->color = BLACK; // Убеждаемся, что корень дерева всегда черного цвета
    }
```

```
// Левое вращение
void rotateLeft(Node* node) {
    Node* rightChild = node->right;
    node->right = rightChild->left;
    if (rightChild->left != nullptr) {
        rightChild->left->parent = node;
    }
    rightChild->parent = node->parent;
    if (node->parent == nullptr) {
        root = rightChild;
    }
    else if (node == node->parent->left) {
        node->parent->left = rightChild;
    }
    else {
        node->parent->right = rightChild;
    }
    rightChild->left = node;
    node->parent = rightChild;
}

// Правое вращение
void rotateRight(Node* node) {
    Node* leftChild = node->left;
    node->left = leftChild->right;
    if (leftChild->right != nullptr) {
        leftChild->right->parent = node;
    }
    leftChild->parent = node->parent;
    if (node->parent == nullptr) {
        root = leftChild;
    }
    else if (node == node->parent->left) {
        node->parent->left = leftChild;
    }
    else {
        node->parent->right = leftChild;
    }
    leftChild->right = node;
    node->parent = leftChild;
}
```

```
    // Удаление узла из дерева
    void deleteNode(Node* node) {
        Node* replacement;
        Node* successor = nullptr;
        bool isBlack = node->color == BLACK; // Проверяем, является ли удаляемый узел
черным

        if (node->left != nullptr && node->right != nullptr) {
            successor = minimumNode(node->right); // Находим преемника удаляемого узла
            node->data = successor->data; // Заменяем данные удаляемого узла данными
преемника
            node = successor; // Устанавливаем удаляемый узел на преемника
        }

        if (node->left != nullptr) {
            replacement = node->left; // Устанавливаем замену на левый потомок  (если
есть)
        }
        else {
            replacement = node->right; // Устанавливаем замену на правый потомок
        }

        if (replacement != nullptr) {
            replacement->parent = node->parent; // Обновляем ссылку на родителя для
замены

            if (node->parent == nullptr) {
                root = replacement; // Если удаляемый узел был корнем, заменяем корень
            }
            else if (node == node->parent->left) {
                node->parent->left = replacement; // Обновляем ссылку на левого потомка
родителя
            }
            else {
                node->parent->right = replacement; // Обновляем ссылку на правого
потомка родителя
            }

            delete node; // Удаляем узел

            if (isBlack) {
                fixDelete(replacement); // Если удаляемый узел был черным, исправляем
баланс
            }
        }
        else if (node->parent == nullptr) {
```

```
            root = nullptr; // Если удаляемый узел был корнем, удаляем корень и узел
            delete node;
        }
        else {
            if (node->color == BLACK) {
                fixDelete(node); // Если удаляемый узел был черным, исправляем баланс
            }

            if (node->parent != nullptr) {
                if (node == node->parent->left) {
                    node->parent->left = nullptr; // Обнуляем ссылку на левого потомка у
родителя
                }
                else {
                    node->parent->right = nullptr; // Обнуляем ссылку на правого потомка
у родителя
                }

                delete node; // Удаляем узел
            }
        }
    }

    // Балансировка дерева после удаления
    void fixDelete(Node* node) {
        while (node != root && node->color == BLACK) {
            if (node == node->parent->left) {
                Node* sibling = node->parent->right; // Получаем брата (потомка
родителя)
                if (sibling->color == RED) { // Если брат красный, меняем цвета
                    sibling->color = BLACK;
                    node->parent->color = RED;
                    rotateLeft(node->parent); // Вращение влево с родителем
                    sibling = node->parent->right;
                }
                if (sibling->left->color == BLACK && sibling->right->color == BLACK) {
                    sibling->color = RED; // Если оба потомка брата черные, брат
становится красным
                    node = node->parent; // Поднимаемся выше
                }
                else {
                    if (sibling->right->color == BLACK) { // Если правый потомок брата
черный
                        sibling->left->color = BLACK;
                        sibling->color = RED;
```

```
                        rotateRight(sibling); // Вращение вправо с братом
                        sibling = node->parent->right;
                    }
                    sibling->color = node->parent->color; // Брат получает цвет родителя
                    node->parent->color = BLACK; // Родитель становится черным
                    sibling->right->color = BLACK; // Правый потомок брата становится
черным
                    rotateLeft(node->parent); // Вращение влево с родителем
                    node = root; // Завершаем цикл, так как закончили балансировку
                }
            }
            else {
                Node* sibling = node->parent->left; // Аналогично для правых и левых
случаев
                if (sibling->color == RED) {
                    sibling->color = BLACK;
                    node->parent->color = RED;
                    rotateRight(node->parent);
                    sibling = node->parent->left;
                }
                if (sibling->right->color == BLACK && sibling->left->color == BLACK) {
                    sibling->color = RED;
                    node = node->parent;
                }
                else {
                    if (sibling->left->color == BLACK) {
                        sibling->right->color = BLACK;
                        sibling->color = RED;
                        rotateLeft(sibling);
                        sibling = node->parent->left;
                    }
                    sibling->color = node->parent->color;
                    node->parent->color = BLACK;
                    sibling->left->color = BLACK;
                    rotateRight(node->parent);
                    node = root;
                }
            }
        }
        node->color = BLACK; // Устанавливаем корень в черный цвет, чтобы сохранить
свойства КЧ-дерева
    }


    // Поиск узла с заданным значением в дереве
```

```
Node* searchN(int value, Node* node) {
    if (node == nullptr || node->data == value) {
        return node;
    }
    if (value < node->data) {
        return searchN(value, node->left);
    }
    else {
        return searchN(value, node->right);
    }
}

// Нахождение узла с минимальным значением в дереве
Node* minimumNode(Node* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

// Рекурсивный инфиксный обход дерева
void inorderTraversal(Node* node) {
    if (node != nullptr) {
        inorderTraversal(node->left);
        cout << node->data << " ";
        inorderTraversal(node->right);
    }
}

// Вывод дерева
void display_tree(Node* node, int mode, int level = 0) {
    if (node == nullptr) {
        if (level == 0) {
            cout << "Длина дерева ноль -_-" << endl;
        }
        if (mode)
        {
            cout << string(level * 4, ' ') << level << "-> null B " << endl;
        }

        return;
    }
```

```
        display_tree(node->right, mode, level + 1);
        cout << (node->color == BLACK ? "" : "\033[31m") << string(level * 4, ' ') <<
level << "-> " << node->data << " " << (node->color == BLACK ? "B" : "R\033[0m") <<
endl;
        display_tree(node->left, mode, level + 1);
    }
};


int main()
{
    setlocale(LC_ALL, "Russian");
    string fText = "LibraryNote_data.txt";
    string fBin = "LibraryNote_data.dat";
    RedBlackTree tree;

    while (true) {
        int c;
        cout << "\n================== \
            \n[ 1 ] - Создать бинарный файл  \
            \n[ 2 ] - Вывести бинарный файл  \
            \n[ 3 ] - Создать КЧД по .dat файлу \
            \n[ 4 ] - Вывод КЧД \
            \n  L[ 41 ] - Вывод КЧД с пустыми листьями \
            \n[ 5 ] - Удалить элемент из дерева по номеру \
            \n[ 6 ] - Добавить элемент \
            \n[ 7 ] - Поиск элемента \
            \n==================\n> ";
        cin >> c;

        switch (c)
        {
        case 1:
        {
            create_bin_file(fText, fBin);
            break;
        }
        case 2:
        {
            print_bin_file(fBin);
            break;
        }
        case 3:
        {
            int rotations = 0;
```

```
            ifstream fb(fBin, ios::in | ios::binary);
            LibraryNote x;
            int key = 0;
            fb.read((char*)&x, sizeof(LibraryNote)); key++;
            while (!fb.eof())
            {
                int rotations2 = 0;
                tree.insert(x.number, key, rotations2); key++;
                rotations += rotations2;
                fb.read((char*)&x, sizeof(LibraryNote));
            }
            cout << "\nДерево построено:\n  кол-во поворотов: ";
            cout << rotations << "\n  кол-во элементов: " << key << endl;
            cout << "\n  среднее кол-во поворотов: " << (double)rotations / (double)key
<< endl;
            break;
        }
        case 4:
        {
            cout << endl;
            tree.print(tree.getRoot(), 0);
            break;
        }
        case 41:
        {
            cout << endl;
            tree.print(tree.getRoot(), 1);
            break;
        }
        case 5:
        {
            cout << "\nВведите номер для удалению: ";
            int num;
            cin >> num;
            tree.remove(num);
            cout << "\nЭлемент удалён.\n";
            break;
        }
        case 6:
        {
            int rotations = 0;
            tree.insert(add_bin_file(fBin), getLastRecordIndex(fBin), rotations);
            cout << "\nЭлемент добавлен. Кол-во поворотов: ";
            cout << rotations << endl;
```

```
            break;
        }
        case 7:
        {
            cout << "\nВведите номер для поиска: ";
            int num;
            cin >> num;
            printAtIndex(fBin, tree.search(num)->key - 1);
            break;
        }
        default:
            break;
        }
    }
}
```

2.2.6 Тестирование

Структура бинарного файла и пользовательское меню приложения (рис. 9).

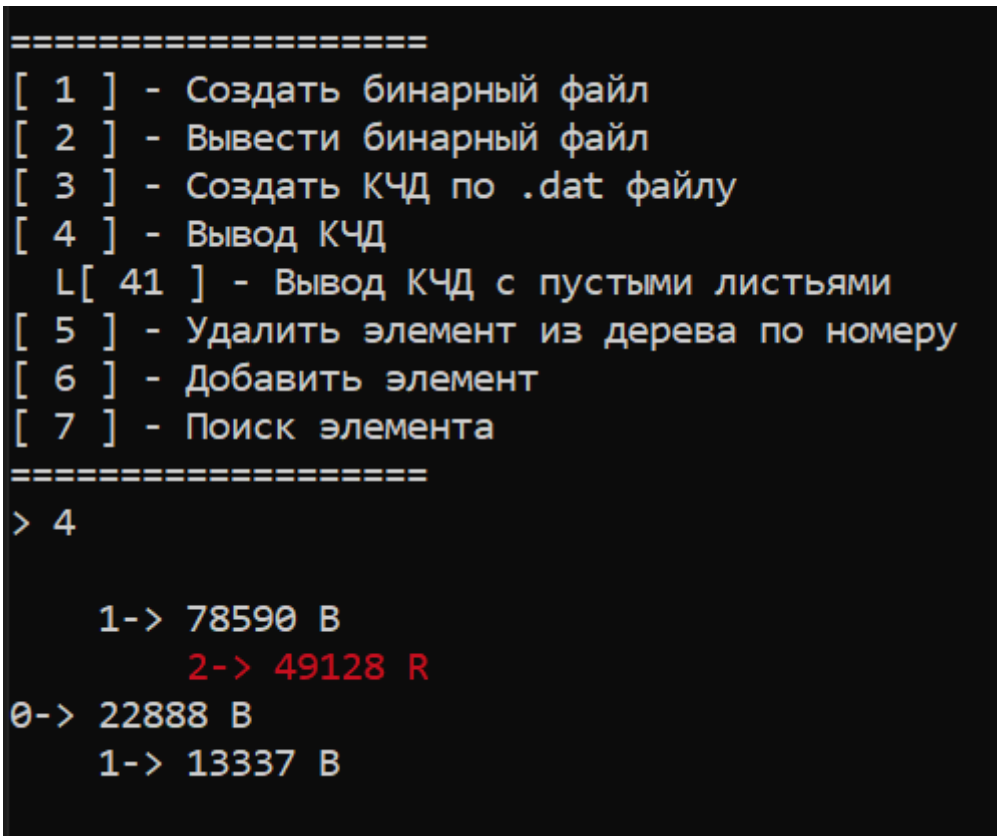

Рисунок 9 – Текстовый интерфейс приложения и данные файла

Процесс строение дерева с подсчётом поворотов и его отображение (рис. 10).

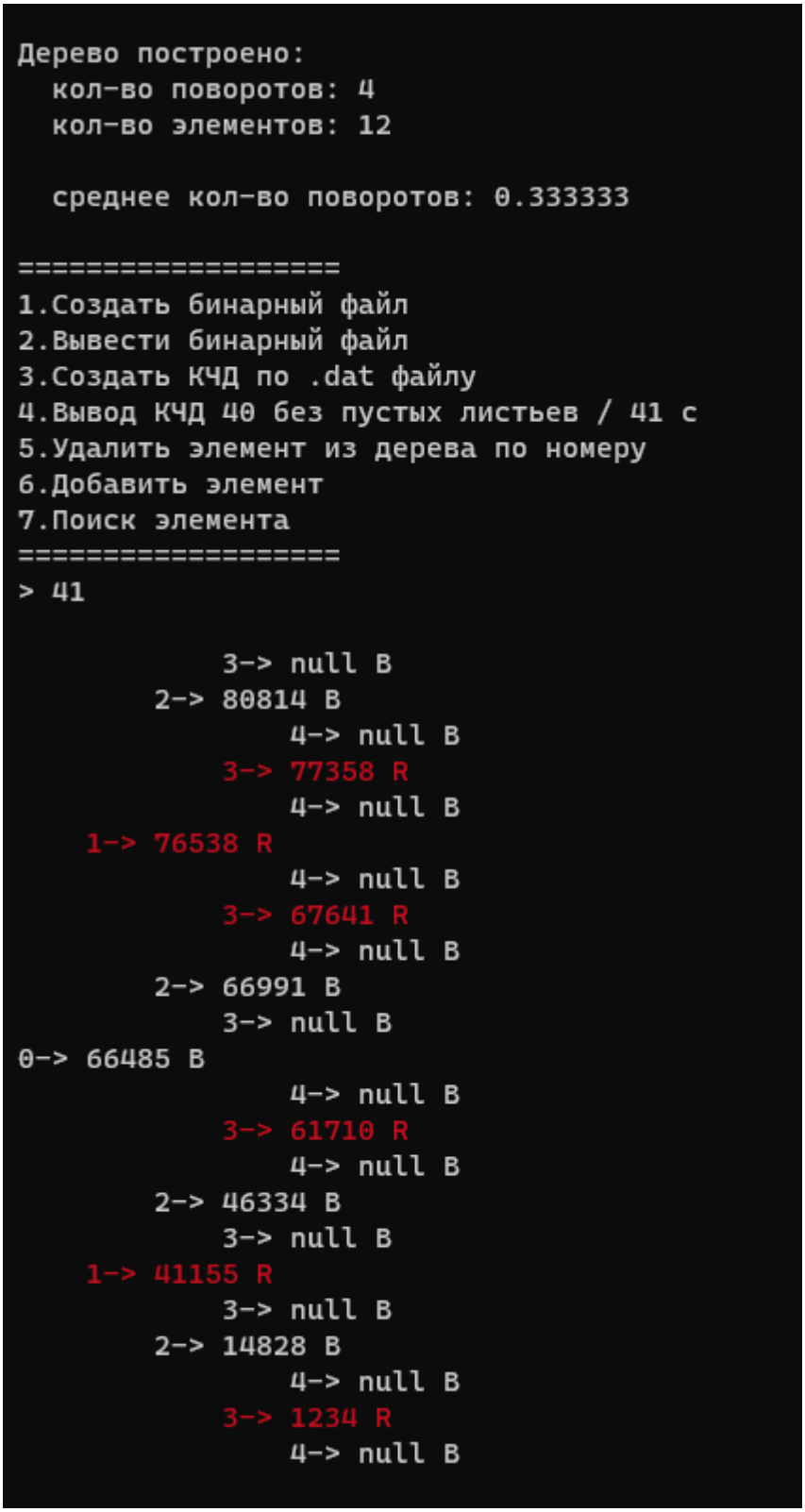

Рисунок 10 – Построенное КЧД

Удаление чёрного узла с номером 66991 (рис. 11)

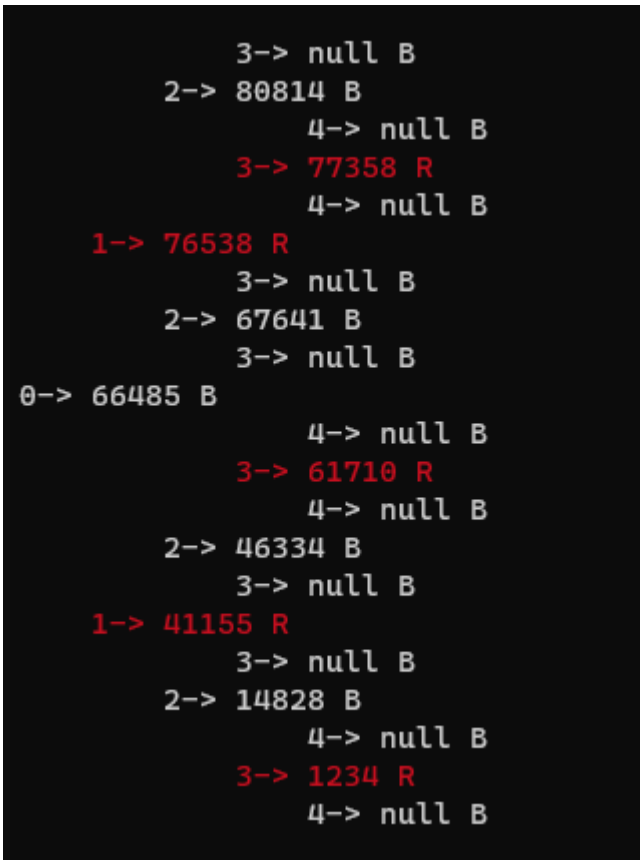

Рисунок 11 – Удаление чёрного элемента элемента 67641 ранее был красный, стал чёрным

Удаление красного элемента с 2 детьми 76538 (рис. 12).

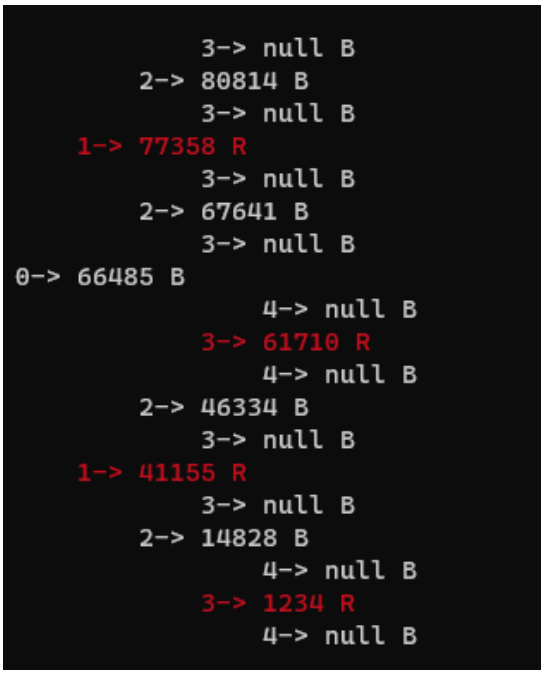

Рисунок 12 – Удаление красного элемента

Добавление нового элемента (рис. 13-15)

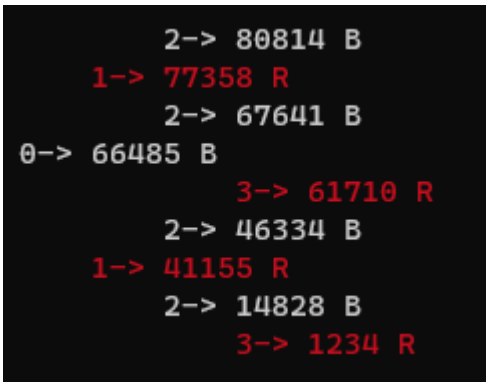

Рисунок 13 – Дерево без пустых листьев

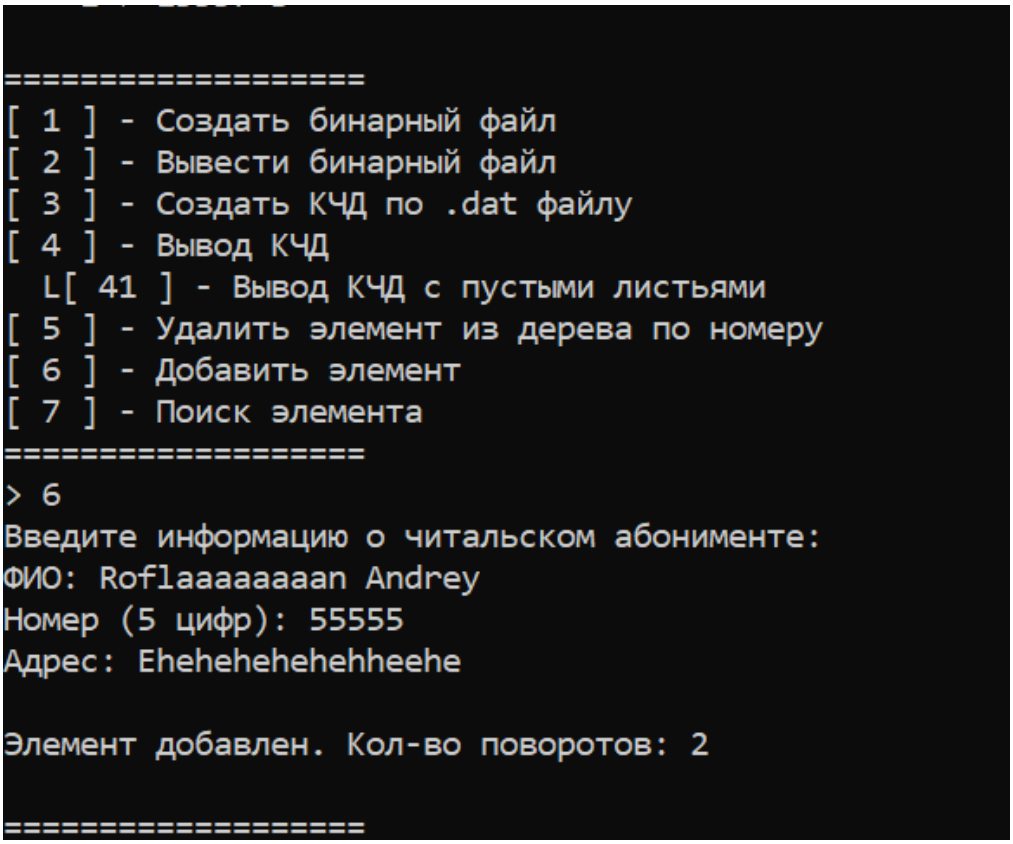

Рисунок 14 – Добавление элемента

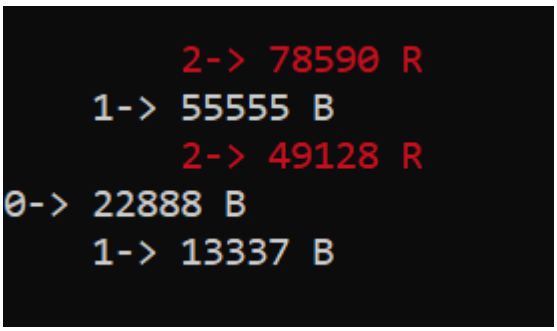

Рисунок 15 – Дерево после добавления элемента

Поиск элемента в дереве по номеру с дальнейшим обращением к бинарному файлу за данными (рис. 16).

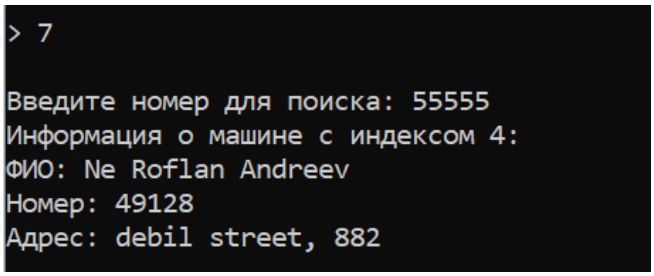

Рисунок 16 – Поиск элемента

2.3 Задание 3

2.3.1 Постановка задачи:

Выполнить анализ алгоритма поиска записи с заданным ключом при применении структур данных:

- хеш – таблица;

- бинарное дерево поиска;

- СДП Требования по выполнению задания

1. Протестировать на данных: а) небольшого объема; б) большого объема.

2. Построить хеш-таблицу из чисел файла.

3. Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Оформить таблицу результатов по форме: Вид поисковой структуры Количество элементов,

загруженных в структуру в момент выполнения поиска Емкостная сложность: объем памяти для структуры Количество выполненных сравнений, время на поиск ключа в структуре

4. Провести анализ алгоритма поиска ключа на исследованных поисковых структурах на основе данных, представленных в таблице.

2.3.2 Ход работы:

Таблица 1 – Асимптотика структур данных

Структура данных	Лучший случай	Худший случай
Хэш-таблица	$O(1)$	$O(n)$
БДП	$O(\lg(n))$	$O(n)$
СДП (КЧД)	$O(\log(n))$	O(log(n))

Эти сравнения структур данных оценивают производительность (временную сложность) лучшего и худшего случая для операций в различных структурах данных.

Хэш-таблица:

Лучший случай (O(1)): В лучшем случае, при правильной хэшировании и равномерном распределении данных, операции вставки, поиска и удаления выполняются за постоянное время, что делает хэш-таблицы очень эффективными.

Худший случай (O(n)): В худшем случае, когда возникают коллизии (когда два элемента хэшируются в одну и ту же ячейку), производительность может ухудшиться до O(n), где n - количество элементов в таблице.

Бинарное дерево поиска (БДП):

Лучший случай (O(log(n)): В лучшем случае, когда БДП сбалансировано, операции поиска, вставки и удаления выполняются за логарифмическое время от числа элементов n в дереве.

Худший случай (O(n)): В худшем случае, если дерево несбалансировано (например, в виде списка), операции могут потребовать O(n) времени.

Сбалансированное двоичное дерево поиска (СДП или КЧД):

Лучший случай (O(log(n)): Сбалансированные двоичные деревья поиска обеспечивают быстрый доступ к данным, и операции выполняются в среднем за логарифмическое время.

Худший случай (O(log(n)): Даже в худшем случае (если дерево всегда сбалансировано), операции все равно выполняются за логарифмическое время.

Таблица 2 – тестирование структур на данных разного объема на процесс поиска

Вид поисковой структуры	Количество элементов, загруженных в структуру в момент выполнения поиска	Емкостная сложность: объём памяти для структуры	Время на поиск ключа в структуре в мс.
Хэш-таблица	1000	const	0.1
БДП	1000	const	3.0
СДП (КЧД)	1000	const	1.5
Хэш-таблица	1000000	const	0.1
БДП	1000000	const	24.7
СДП (КЧД)	1000000	const	18.2

***const** т.к в процессе поиска размер структур не изменяется. В общем виде для всех структур составит 30 байт (owner) + 4 байта (number) + 30 байт (mark) = 64 байта * n, где – n – кол-во элементов. Так, хэш таблица при рехешировании может увеличить размер структуры данных зарезервировав место под новые пустые ячейки таблицы для увеличения размера. А бинарные деревья будут динамически менять свой размер при добавлении/удалении элементов.

Вывод по результатам сравнения:

Для всех тестов были взяты одни и те же наборы данных, что доказало на практике асимптотическую оценку сложности поиска в данных структурах.

Хэш-таблица для данного набора данных оказалась наиболее эффективной, но использование КЧД гарантировано дает логарифмическую сложность в любой ситуации, тогда как хэш-таблица может “завязнуть” в коллизиях и ухудшить свою ситуацию до линейной сложности.

3 ВЫВОД

В рамках практической работы было проведено исследование сбалансированных деревьев поиска, а именно, сосредоточено внимание на сбалансированных бинарных деревьях поиска (КЧД) и их применении для эффективного поиска данных в файлах. Целью данного исследования было приобрести навыки разработки и реализации алгоритмов управления бинарным деревом поиска, а также применения сбалансированных деревьев поиска для прямого доступа к записям в файлах.

В ходе исследования были получены следующие результаты:

1. Сравнение производительности различных структур данных, включая хэш-таблицы, бинарные деревья поиска и сбалансированные бинарные деревья поиска (КЧД), позволяет выделить их основные характеристики.

2. Хэш-таблица оказалась наиболее эффективной в лучшем случае благодаря постоянному времени выполнения операций. Однако в худшем случае, при коллизиях, ее производительность может ухудшиться, что следует учитывать при выборе этой структуры.

3. Бинарные деревья поиска (БДП) имеют логарифмическое время выполнения в среднем случае, но могут оказаться неэффективными в худшем случае, особенно когда дерево дегенерируется в список.

4. Сбалансированные бинарные деревья поиска (КЧД) продемонстрировали стабильно хорошую производительность как в лучшем, так и в худшем случае, что делает их надежным выбором для большинства сценариев.

Важно отметить, что выбор структуры данных зависит от конкретных требований приложения и ожидаемых сценариев использования. Подходящая структура данных может значительно повысить эффективность операций поиска и доступа к данным.

Это исследование позволило лучше понять преимущества и ограничения различных структур данных в различных сценариях, что является ценным знанием для разработки и оптимизации программных приложений.

4 СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. СиАОД Самостоятельная работа 5 (сбалансированные деревья поиска).pdf
2. Рысин М.Л. и др. Введение в структуры и алгоритмы обработки данных. Ч. 1 - учебное пособие, 2022
3. Рысин М.Л. и др. Основы программирования на языке C++. Учебное пособие, 2022

5. metanit.co

6. ru.cppreference.com/w/

7. habr.com

8. techiedelight.com

9. overcoder.net

10. tree - Non-recursive depth first search algorithm - Stack Overflow. https://stackoverflow.com/questions/5278580/non-recursive-depth-first-search-algorithm.

11. Non-recursive Depth First Search Algorithm: Explained for Data .... https://saturncloud.io/blog/nonrecursive-depth-first-search-algorithm-explained-for-data-scientists/.

12. Tree Traversal Techniques - Data Structure and Algorithm Tutorials .... https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/.