МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №5

по дисциплине

«Структуры и алгоритмы обработки данных»

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

Выполнил студент группы ИКБО-04-22 Основин А.И.

Принял старший преподаватель Скворцова Л.А.

Москва 2023

СОДЕРЖАНИЕ

1 ЗАДАНИЕ №1 4
1.1 Постановка задачи 4
1.2 Подход к решению 4
1.2.1 Структура узла бинарного дерева поиска 4
1.2.2 Структура и методы бинарного дерева поиска 5
1.2.3 Алгоритмы операций на псевдокоде 10
1.3 Двоичный файл из записей фиксированного размера 12
1.4 Прототипы операций по управления двоичным файлом 13
1.5 Код приложения 14
1.6 Содержание текстового файла 22
1.7 Результаты тестирования 23
2 ЗАДАНИЕ №2 26
2.1 Постановка задачи 26
2.2 Подход к решению 26
2.2.1 Структура узла АВЛ-дерева 26
2.2.2 Структура и методы АВЛ-дерева 27
2.2.3 Алгоритмы операций на псевдокоде 34
2.3 Двоичный файл из записей фиксированного размера 37
2.4 Прототипы операций по управления двоичным файлом 38
2.5 Код приложения 39
2.6 Содержание текстового файла 48
2.7 Результаты тестирования 49
2.8 Среднее число выполненных поворотов 52
3 ЗАДАНИЕ №3 53

3.1 Постановка задачи 53
3.2 Решение 53
4 ВЫВОД 55

1 ЗАДАНИЕ №1

1.1 Постановка задачи

Разработать приложение, которое использует бинарное дерево поиска (БДП) для поиска записи с ключом в файле, структура записи которого приведена в варианте №17.

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

Разработать класс (библиотеку функций) управления файлом. Включить методы: создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле; поиск записи в файле с использованием БДП.

Дано. Файл двоичный с записями фиксированной длины. Структура записи файла представлена в Таблице 1.

Таблица 1 – Структура записи файла

Поле	char name[30]	unsigned int count
Назначение	Слово из текста	Количество вхождений слова в текст

Результат. Приложение, выполняющее операции поиска, добавления и удаления записей из бинарного файла с помощью бинарного дерева поиска.

1.2 Подход к решению

1.2.1 Структура узла бинарного дерева поиска

Бинарное дерево поиска реализовано как класс с закрытыми вспомогательными методами и открытыми методами-оболочками для непосредственного взаимодействия с ними.

Бинарное дерево поиска состоит из узлов, структура которых представлена в Листинге 1. Для более удобного взаимодействия с узлами БДП и поддержания принципа инкапсуляции в узле БДП определена внутренняя структура, которая используется для хранения ключа и позиции данного элемента в файле.

Листинг 1 – Структура узла бинарного дерева поиска

```
struct Node {
    struct record {
        string key;
        int position;
        record(string _key, int _position) : key(_key), position(_position) {};
    } data;

    Node* left;
    Node* right;
    Node(string _key, int _position) : data(_key, _position), left(nullptr),
right(nullptr) {};
};
```

1.2.2 Структура и методы бинарного дерева поиска

Прототипы основных и вспомогательных методов структуры бинарное дерево поиска представлены в Листинге 2.

Листинг 2 – Структура бинарное дерево поиска

```
struct binTree {
    Node* root;
    binTree() : root(nullptr) {};
    binTree(Node* _root) : root(_root) {};
    ~binTree();

    Node* search(string key);
    int getIndex(string key);
    void add(string _key, int _position);
    int erase(string key);
    void print();

private:
    Node* searchFrom(Node* current, string key);
    Node* addTo(Node* current, Node* new_node);
    Node* remove(Node* current, string key, int& position);
    Node* removeMin(Node* current);

    Node* getMin(Node* from);
    static void printSubTree(Node* current, char from = ' ', size_t from_size
= 0, size_t level = 0);
    void clear(Node* current);
     };
```

- Метод search(string key)
    - Описание: метод-оболочка для метода поиска в БДП, вызывает метод поиска в БДП от корневого узла.
    - Предусловие: key – строковое значение, ключ искомого узла.
    - Постусловие: возвращает указатель на найденный узел, если такой существует в БДП, иначе – нулевой указатель.
- Метод searchFrom(Node* current, string key)
    - Описание: вызывается методом-оболочкой search(string key), реализует поиск элемента в БДП.
    - Предусловие: current указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом-оболочкой передаётся указатель на корень БДП, если корень не пустой; string key – ключ, поиск которого осуществляется в БДП.
    - Постусловие: возвращает указатель на узел, ключ которого равен искомому, если такой существует, в противном случае – вернёт нулевой указатель.
- Метод getIndex(string key)
    - Описание: метод для получения позиции узла с заданным ключом в файле.
    - Предусловие: key – строковое значение, ключ искомого узла.
    - Постусловие: возвращает позицию узла с заданным ключом в файле, если узел с заданным ключом найден, иначе – -1.
- Метод add(string _key, int _position)
    - Описание: метод-оболочка для метода добавления узла в БДП, создаёт новый экземпляр узла БДП и вызывает метод добавления созданного узла в БДП от корневого узла, если корневой узел существует, иначе – добавляет в БДП созданный узел в качестве корневого.

- Предусловие: _key – строковое значение, ключ создаваемого узла, _position – целочисленное значение, позиция создаваемого узла в файле.
- Постусловие: в БДП добавлен узел с ключом, равным _key, и позицией в файле _position.

- Метод addTo(Node* current, Node* new_node)
  - Описание: вызывается методом-оболочкой add(string _key, int _position), реализует добавление элемента в БДП.
  - Предусловие: current – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом-оболочкой передаётся указатель на корень БДП, если корень не пустой; new_node – указатель на новый узел, который необходимо добавить в БДП.
  - Постусловие: в БДП добавлен новый узел; возвращает указатель на текущий узел, для использования текущего узла предыдущим уровнем рекурсии, при отсутствии текущего узла возвращает указатель на новый узел, таким образом, прекращая цепочку рекурсивных вызовов.
- Метод erase(string key)
  - Описание: метод-оболочка для метода удаления узла из БДП, инициализирует целочисленную переменную -1, вызывает метод удаления узла из БДП от корневого узла, передавая созданную целочисленную переменную для записи в неё позиции найденного узла в файле.
  - Предусловие: key – строковое значение, ключ удаляемого узла.
  - Постусловие: из БДП удалён узел с ключом, равным _key, если такой узел существовал в БДП, возвращает позицию удалённого узла в файле.
- Метод remove(Node* current, string key, int& position)

- Описание: вызывается методом-оболочкой erase(string key), реализует удаление узла из БДП.
- Предусловие: current – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом-оболочкой передаётся указатель на корень БДП; string key – ключ, удаление которого осуществляется из БДП, int& position – ссылка на целочисленную переменную, которая создаётся в методе-оболочке, ссылка на переменную предназначается для записи в неё позиции в файле удаляемого элемента.
- Постусловие: из БДП удалён узел с заданным ключом, в целочисленную переменную по ссылке записана позиция удаляемого узла в файле, возвращает указатель на узел, с которым осуществлялась работа на текущем уровне рекурсии для использования текущего узла предыдущим уровнем рекурсии, если текущий узел – удаляемый, возвращает нулевой указатель или указатель на замещающий узел, тем самым осуществляя удаление узла из БДП.

- Метод removeMin(Node* current)
  - Описание: вызывается методом удаления узла из БДП remove(Node* current, string key, int& position), реализует удаление замещающего узла из БДП.
  - Предусловие: current – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом удаления узла из БДП передаётся указатель на правое поддерево узла, который необходимо удалить.
  - Постусловие: из БДП удалён узел с заданным ключом, возвращает указатель на узел, с которым осуществлялась работа на текущем уровне рекурсии для использования текущего узла предыдущим уровнем рекурсии, если текущий узел – удаляемый (наименьший узел в правом поддереве от «реально» удаляемого узла), возвращает

указатель на его правое поддерево, таким образом, переставляя самый меньший узел в правом поддереве от удаляемого узла на место удаляемого узла и перевешивая правое поддерево замещающего узла на родителя замещающего узла.

- Метод getMin(Node* from)
    - Описание: вызывается методом удаления узла из БДП remove(Node* current, string key, int& position), реализует поиск наименьшего узла поддерева с корнем в удаляемом узле; найденный наименьший узел станет замещающим.
    - Предусловие: from – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом удаления узла из БДП передаётся указатель на правое поддерево узла, который необходимо удалить.
    - Постусловие: возвращает указатель на наименьший узел поддерева с корнем в узле from (правом потомке удаляемого узла).
- Метод print()
    - Описание: метод-оболочка для метода вывода БДП, вызывает метод вывода БДП от корневого узла, если дерево не пустое, иначе – выводит сообщение о том, что в дереве нет узлов.
    - Предусловие: отсутствует.
    - Постусловие: в консоль выведено БДП.
- Метод printSubTree(Node* current, char from, size_t from_size, size_t from_level)
    - Описание: вызывается методом-оболочкой print(), реализует вывод БДП в консоль.
    - Предусловие: current – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом-оболочкой передаётся указатель на корень БДП, если БДП не пустое; from – символьное значение для отображения связи между

родительским и дочерним узлом, по умолчанию равно пробелу; from_size – целочисленное беззнаковое – размер информационной части родительского узла в символах, используется для выравнивания вывода, по умолчанию равно нулю; from_level – целочисленное беззнаковое – текущий уровень рекурсии (глубина узла, который выводится при данном рекурсивном вызове, в дереве), используется для выравнивания вывода, по умолчанию равно нулю.

- Постусловие: в консоль выведено БДП.

Также в структуре бинарного дерева поиска определены методы:

- Конструктор по умолчанию для создания пустого дерева.
- Параметризованный конструктор для создания дерева и инициализации корня заданным значением.
- void clear(Node* current) – удаление всех узлов дерева, включая корневой узел.
- Деструктор.

1.2.3 Алгоритмы операций на псевдокоде

Алгоритм поиска узла в БДП на псевдокоде представлен в Листинге 3. Предусловие: current – указатель на корень БДП, key – ключ искомого узла. Постусловие: возвращает указатель на узел, ключ которого равен искомому, если такой существует, в противном случае – вернёт нулевой указатель.

Листинг 3 – Алгоритм поиска узла в БДП

```
searchFrom(current, key){
    if (current = NULL || key = current.data.key) then
        return current;
    else
        if (key < current.data.key) then
            return searchFrom(current.left, key);
        else
            return searchFrom(current.right, key);
        endif
    endif
}
```

Алгоритм вставки узла в БДП на псевдокоде представлен в Листинге 4. Предусловие: current – указатель на корень БДП, _key – строковое значение,

ключ создаваемого узла, _position – целочисленное значение, позиция создаваемого узла в файле. Постусловие: в БДП добавлен узел с ключом, равным _key, и позицией в файле _position, при каждом рекурсивном вызове возвращает указатель на текущий узел (если текущий узел пустой, возвращает указатель на новый узел, тем самым добавляя новый узел в дерево.)

Листинг 4 – Алгоритм вставки узла в БДП

```
addTo(current, _key, _position) {
    if (current = NULL) then
        current <- new Node(_key, _position);
    endif

    if (_key < current.data.key) then
        current.left <- addTo(current.left, _key, _position);
    else
        current.right <- addTo(current.right, _key, _position);
    endif

    return current;
}
```

Алгоритм удаления узла из БДП на псевдокоде представлен в Листинге 5. У метода удаления есть вспомогательные методы: метод нахождения наименьшего узла в поддереве с корнем в узле, который передан в качестве аргумента – getMin(Node* from); метод «вытеснения» замещающего узла из поддерева с корнем в узле, который передан в качестве аргумента removeMin(Node* current). Предусловие: current – указатель на корень БДП, key – ключ, удаление которого осуществляется из БДП, position – ссылка на целочисленную переменную, которая предназначается для записи позиции в файле удаляемого элемента. Постусловие: из БДП удалён узел с заданным ключом, в целочисленную переменную по ссылке записана позиция удаляемого узла в файле.

Листинг 5 – Алгоритм удаления узла из БДП

```
remove(current, key, position) {
    if (current = NULL) then
        return NULL;
    endif

    if (key < current.data.key) then
        current.left <- erase(current.left, key, position);
    else if (key > current.data.key) then
        current.right <- erase(current.right, key, position);
    else
        right <- current.right;
        left <- current.left;
```

```
        position <- current.data.position;
        delete current;

        if (right = NULL) then
            return left;
        endif

        auxiliary <- getMin(right);
        auxiliary.right <- removeMin(right);
        auxiliary.left <- left;
        if (root = NULL) then
            root <- auxiliary;
        endif
        return auxiliary;
    endif

    return current;
}

getMin(from){
    if (from.left = NULL) then
        return from.right;
    endif
    return getMin(from.left);
}

removeMin(current){
    if (current.left = NULL) then
        return current.right;
    endif
    current.left <- removeMin(current.left);
    return current;
}
```

1.3 Двоичный файл из записей фиксированного размера

В Листинге 6 представлена структура записи файла.

Листинг 6 – Структура записи файла

```
struct word {
    char name[30];
    unsigned int count;
};
```

На первый взгляд, размер одной записи данной структуры равен сумме её полей, то есть 30 Байт для массива типа char и 4 Байта для переменной типа int – 34 Байта в сумме, однако это не так. Язык программирования C++ для обеспечения скорости доступа к элементам памяти и безопасности при работе с памятью применяет Байты заполнения: неиспользуемые байты памяти, которые вставляются между переменными в структуре, чтобы гарантировать, что каждая переменная начинается с адреса памяти, выровненного с ее типом данных. В данном случае размер массива name должен быть кратен 4 (размеру

переменной типа int). По этой причине в структуре происходит выравнивание: между массивом name и переменной count есть два неиспользуемых Байта, вследствие чего одна запись данной структуры весит 36 Байтов.

В целях экономии места можно сократить размер массива char на два символа, тогда вес каждой записи такой структуры станет меньше на целых 4 Байта. Также можно увеличить размер массива char на два элемента при этом не увеличивая затраты памяти. Однако, размер структуры принят равным 36 Байтам при любом случае. Система также показывает размер 36 Байт, как показано на Рисунке 1.

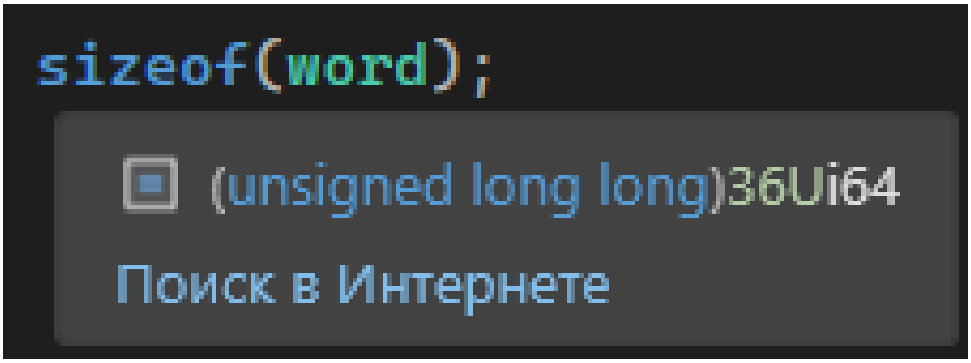

Рисунок 1 – Размер одного экземпляра структуры word

1.4 Прототипы операций по управления двоичным файлом

Прототипы операций по управлению двоичным файлом представлены в Листинге 7.

Листинг 7 – Прототипы операций по управлению двоичным файлом

```
void text2bin(istream& text_file, ostream& bin_file);
void bin2tree(binTree& tree, istream& file, int& position);
word findWord(binTree& tree, istream& file, string key);
void addWord(binTree& tree, ostream& file, string key, unsigned int count,
int& position);
bool eraseWord(binTree& tree, fstream& file, string key, string file_path);
```

Операции по управлению бинарным файлом:

- text2bin(istream& text_file, ostream& bin_file) – перевод текстового файла в двоичный.
- bin2tree(binTree& tree, istream& file, int& position) – добавление записей в существующее БДП из двоичного файла.

- findWord(binTree& tree, istream& file, string key) – поиск записи по ключу в БДП и вывод из двоичного файла.
- addWord(binTree& tree, ostream& file, string key, unsigned int count, int& position) – добавление записи в БДП и двоичный файл.
- eraseWord(binTree& tree, fstream& file, string key, string file_path) – удаление записи из БДП и двоичного файла.

1.5 Код приложения

В Листинге 8 представлен заголовочный файл, определяющий структуру узла БДП.

Листинг 8 – Node.h

```
#pragma once
#include <iostream>
#include <string>
using namespace std;

struct Node {
    struct record {
        string key;
        int position;
        record(string _key, int _position) : key(_key), position(_position)
{};
    } data;

    Node* left;
    Node* right;
    Node(string _key, int _position) : data(_key, _position), left(nullptr),
right(nullptr) {};
};
```

В Листинге 9 представлен заголовочный файл, определяющий структуру БДП.

Листинг 9 – binTree.h

```
#pragma once
#include "Node.h"

struct binTree {
    Node* root;
    binTree() : root(nullptr) {};
    binTree(Node* _root) : root(_root) {};
    ~binTree();

    Node* search(string key);
    int getIndex(string key);
    void add(string _key, int _position);
    int erase(string key);
    void print();
```

```
private:
    Node* searchFrom(Node* current, string key);
    Node* addTo(Node* current, Node* new_node);
    Node* remove(Node* current, string key, int& position);
    Node* removeMin(Node* current);

    Node* getMin(Node* from);
    static void printSubTree(Node* current, char from = ' ', size_t from_size
= 0, size_t level = 0);
    void clear(Node* current);
};
```

В Листинге 10 представлена реализация методов управления БДП.

Листинг 10 – binTree.cpp

```
#include "binTree.h"

Node* binTree::search(string key) {
    return searchFrom(this->root, key);
}

Node* binTree::searchFrom(Node* current, string key) {
    if (current == nullptr || key == current->data.key) {
        return current;
    }
    else {
        if (key < current->data.key) {
            return searchFrom(current->left, key);
        }
        else {
            return searchFrom(current->right, key);
        }
    }
}

int binTree::getIndex(string key) {
    Node* found = search(key);
    if (found == nullptr) {
        return -1;
    }
    return found->data.position;
}

void binTree::add(string _key, int _position) {
    this->root = addTo(this->root, new Node(_key, _position));
}

Node* binTree::addTo(Node* current, Node* new_node) {
    if (current == nullptr) {
        return new_node;
    }

    if (new_node->data.key < current->data.key) {
        current->left = addTo(current->left, new_node);
    }
    else {
        current->right = addTo(current->right, new_node);
    }
    return current;
}

int binTree::erase(string key) {
    int position = -1;
```

```
    this->root = remove(this->root, key, position);
    return position;
}

Node* binTree::remove(Node* current, string key, int& position) {
    if (current == nullptr) {
        return nullptr;
    }

    if (key < current->data.key) {
        current->left = remove(current->left, key, position);
    }
    else if (key > current->data.key){
        current->right = remove(current->right, key, position);
    }
    else {
        Node* right = current->right, * left = current->left;
        position = current->data.position;
        delete current;

        if (right == nullptr) {
            return left;
        }

        Node* auxiliary = getMin(right);
        auxiliary->right = removeMin(right);
        auxiliary->left = left;
        return auxiliary;
    }
    return current;
}

Node* binTree::removeMin(Node* current) {
    if (current->left == nullptr) {
        return current->right;
    }
    current->left = removeMin(current->left);
    return current;
}

Node* binTree::getMin(Node* from) {
    if (from->left == nullptr) {
        return from;
    }
    return getMin(from->left);
}

void binTree::print() {
    if (this->root == nullptr) {
        cout << "The tree is empty!" << endl;
    }
    printSubTree(this->root);
}

void binTree::printSubTree(Node* current, char from, size_t from_size, size_t
level) {
    if (current != nullptr) {
        printSubTree(current->right, '/', from_size + current-
>data.key.size(), level + 1);

        cout << (level > 0 ? (level > 1 ? string(from_size + (level - 1) * 5,
' ') : string(from_size, ' ')) + from + "----" : "");
        cout << current->data.key << endl;
```

```
        printSubTree(current->left, '\\', from_size + current-
>data.key.size(), level + 1);
    }
}

void binTree::clear(Node* current) {
    if (current->left != nullptr) {
        clear(current->left);
    }

    if (current->right != nullptr) {
        clear(current->right);
    }

    delete current;
};

binTree::~binTree() {
    if (this->root != nullptr) {
        clear(this->root);
    }
}
```

В Листинге 11 представлен заголовочный файл модуля управления бинарным файлом.

Листинг 11 – FileMethods.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include <fstream>
#include <io.h>
#include <fcntl.h>
#include "binTree.h"

struct word {
    char name[30];
    unsigned int count;
};

void text2bin(istream& text_file, ostream& bin_file);
void bin2tree(binTree& tree, istream& file, int& position);
word findWord(binTree& tree, istream& file, string key);
void addWord(binTree& tree, ostream& file, string key, unsigned int count,
int& position);
bool eraseWord(binTree& tree, fstream& file, string key, string file_path);
```

В Листинге 12 представлена реализация модуля управления бинарным файлом.

Листинг 12 – FileMethods.cpp

```
#include "FileMethods.h"

void text2bin(istream& text_file, ostream& bin_file) {
    while (!text_file.eof()) {
        word current;
        int i = 0;

        do {
            text_file.get(current.name[i]);
```

```
        } while (current.name[i++] != '\n');
        current.name[i - 1] = '\0';

        text_file >> current.count;
        text_file.get();

        bin_file.write((char*)&current, sizeof(word));
    }
}

void bin2tree(binTree& tree, istream& file, int& position) {
    position = 0;
    word current;

    file.read((char*)&current, sizeof(word));
    while (!file.eof()) {
        tree.add(current.name, position++);
        file.read((char*)&current, sizeof(word));
    }
}

word findWord(binTree& tree, istream& file, string key) {
    word current;

    int index = tree.getIndex(key);
    if (index == -1) {
        current.name[0] = '\0';
    }
    else {
        file.seekg((index) * sizeof(word), ios::beg);
        file.read((char*)&current, sizeof(word));

        if (file.bad() || file.fail()) {
            current.name[0] = '\0';
        }
    }
    return current;
}

void addWord(binTree& tree, ostream& file, string key, unsigned int count,
int& position) {
    word current = word();
    strcpy(current.name, key.c_str());
    current.count = count;

    tree.add(key, position++);
    file.write((char*)&current, sizeof(word));
}

bool eraseWord(binTree& tree, fstream& file, string key, string file_path) {
    int index = tree.erase(key);
    if (index == -1) {
        return false;
    }

    word last;
    file.seekg(-(int)sizeof(word), ios::end);
    file.read((char*)&last, sizeof(word));

    if (last.name != key) {
        // Record couldn't be deleted if pointer would still be in that part
of file
        file.seekg(ios::beg);
```

```
        file.seekp(index * sizeof(word), ios::beg);
        file.write((char*)&last, sizeof(word));
        tree.search(last.name)->data.position = index;
    }

    int fh;
    if (_sopen_s(&fh, file_path.c_str(), _O_RDWR, _SH_DENYNO, _S_IREAD |
_S_IWRITE) == 0) {
        if (!(_chsize(fh, (_filelength(fh) - sizeof(word))) == 0)) {
            return false;
        }
        _close(fh);
    }

    /*
        * FOR UNIX: CLOSE FILE, THAN RESIZE IT
        #include <filesystem>
        auto p = filesystem::path(file_path);
        filesystem::resize_file(p, (size - 1) * sizeof(word));
        */

    return true;
}
```

В Листинге 13 представлена реализация приложения.

Листинг 13 – main.cpp

```
#include "fileMethods.h"
#include <chrono>

int main() {
      system("chcp 1251");
      string text_file, bin_file;
      ifstream fin;
      ofstream fout;
      fstream file;

      //D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/test.txt
      //D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/test.dat

      //D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/data.txt
      //D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/data.dat

      binTree* tree = new binTree();
      int position = 0;
      int action;
      while (true)
      {
            cout << endl << "Выберите действие:" << endl;
            cout << "1. Преобразование текстового файла в двоичный" << endl;
            cout << "2. Перевод двоичного файла в БДП" << endl;
            cout << "3. Поиск узла в БДП по ключу и вывод из двоичного файла"
<< endl;
            cout << "4. Добавление узла в БДП и двоичный файл по ключу" <<
endl;
            cout << "5. Удаление записи из БДП и двоичного файла" << endl;
            cout << "6. Вывод БДП" << endl;
            cout << "7. Выйти" << endl;

            cin >> action;
            switch (action) {
            case 1: {
                  cout << "Введите имя текстового файла: ";
```

```
                cin >> text_file;
                cout << "Введите имя двоичного файла: ";
                cin >> bin_file;

                fin.open(text_file, ios::in);
                if (fin.is_open()) {
                        fout.open(bin_file, ios::binary | ios::out);
                        text2bin(fin, fout);
                        if (fin.bad() || fout.bad()) {
                                cout << "Ошибка при создании двоичного файла из
текстового." << endl;
                                return 1;
                        }
                        cout << "Двоичный файл успешно создан." << endl;
                        fin.close();
                        fout.close();
                }

                else {
                        cout << "Текстовый файл не найден или не существует."
<< endl;
                }
                break;
        }

        case 2: {
                cout << "Введите имя двоичного файла: ";
                cin >> bin_file;
                fin.open(bin_file, ios::binary | ios::in);

                if (fin.is_open()) {
                        bin2tree(*tree, fin, position);
                        if (fin.bad()) {
                                cout << "Ошибка создания БДП из данных двоичного
файла." << endl;
                                return 1;
                        }
                        cout << "БДП успешно создано." << endl;
                        fin.close();
                }

                else {
                        cout << "Двоичный файл не найден или не существует."
<< endl;
                }
                break;
        }

        case 3: {
                cout << "Введите имя двоичного файла: ";
                cin >> bin_file;
                fin.open(bin_file, ios::binary | ios::in);

                if (fin.is_open()) {
                        string key;
                        cout << "Введите слово для поиска: ";
                        cin >> key;

                        auto start = chrono::high_resolution_clock::now();
                        word found = findWord(*tree, fin, key);
                        auto end = chrono::high_resolution_clock::now();
```

```
                        cout << endl << "----------------------------------
-" << endl;
                        cout << "Время поиска: " <<
chrono::duration_cast<chrono::nanoseconds>(end - start).count() / 1e6 << "
ms";
                        cout << endl << "----------------------------------
-" << endl << endl;

                        if (fin.bad()) {
                              cout << "Ошибка при поиске записи в файле." <<
endl;
                              return 1;
                        }

                        if (found.name[0] != '\0') {
                              cout << "Слово '" << found.name << "'
встретилось в тексте " << found.count << " раз(-а)." << endl;
                        }
                        else {
                              cout << "Запись не найдена." << endl;
                        }
                        fin.close();
                  }

                  else {
                        cout << "Двоичный файл не найден или не существует."
<< endl;
                  }
                  break;
            }

            case 4: {
                  cout << "Введите имя двоичного файла: ";
                  cin >> bin_file;
                  file.open(bin_file, ios::out | ios::app | ios::binary);
                  if (file.is_open()) {
                        string key;
                        unsigned int count;
                        cout << "Введите слово: ";
                        cin >> key;
                        cout << "Введите количество вхождений в текст: ";
                        cin >> count;

                        addWord(*tree, file, key, count, position);
                        if (fin.bad()) {
                              cout << "Ошибка при добавлении слова в файл." <<
endl;
                              return 1;
                        }
                        cout << "Запись успешно удалена." << endl;
                        file.close();
                  }

                  else {
                        cout << "Двоичный файл не найден или не существует."
<< endl;
                  }
                  break;
            }

            case 5: {
                  cout << "Введите имя двоичного файла: ";
                  cin >> bin_file;
```

```
                  file.open(bin_file, ios::in | ios::out | ios::binary);
                  if (file.is_open()) {
                        string key;
                        cout << "Введите слово: ";
                        cin >> key;
                        bool status = eraseWord(*tree, file, key, bin_file);
                        if (fin.bad()) {
                              cout << "Ошибка при удалении слова из файла." <<
endl;
                              return 1;
                        }

                        if (status) {
                              cout << "Запись успешно удалена." << endl;
                              position--;
                        }
                        else {
                              cout << "Запись с таким ключом не найдена." <<
endl;
                        }
                        file.close();
                  }

                  else {
                        cout << "Двоичный файл не найден или не существует."
<< endl;
                  }
                  break;
            }

            case 6: {
                  tree->print();
                  break;
            }

            default: {
                  delete tree;
                  return 0;
            }
            }
      }
}
```

1.6 Содержание текстового файла

На Рисунке 2 представлено содержимое текстового тестового файла.

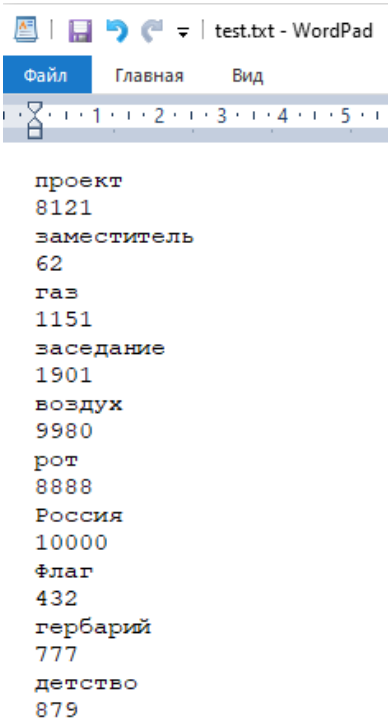

***Рисунок 2* – Содержание тестового файла**

1.7 Результаты тестирования

На Рисунке 3 представлен результат тестирования функции построения БДП по записям из двоичного файла, полученного из заранее подготовленного текстового. БДП успешно построено и выведено в консоль, следовательно, метод добавления элемента в БДП, метод вывода БДП в консоль и функция построения БДП по данным из двоичного файла работают корректно.

```
Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в БДП
3. Поиск узла в БДП по ключу и вывод из двоичного файла
4. Добавление узла в БДП и двоичный файл по ключу
5. Удаление записи из БДП и двоичного файла
6. Вывод БДП
7. Выйти
2
Введите имя двоичного файла: D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/test.dat
БДП успешно создано.

Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в БДП
3. Поиск узла в БДП по ключу и вывод из двоичного файла
4. Добавление узла в БДП и двоичный файл по ключу
5. Удаление записи из БДП и двоичного файла
6. Вывод БДП
7. Выйти
6
      /----рот
проект
                  /----заседание
      \----заместитель
                                    /----детство
                          /----гербарий
                  \----газ
                          \----воздух
                                              /----Флаг
                                    \----Россия
```

Рисунок 3 – Тестирование функции построения БДП по записям из двоичного файла

На Рисунке 4 представлен результат тестирования функции поиска записи в БДП и вывода записи из двоичного файла. Запись успешно найдена и выведена.

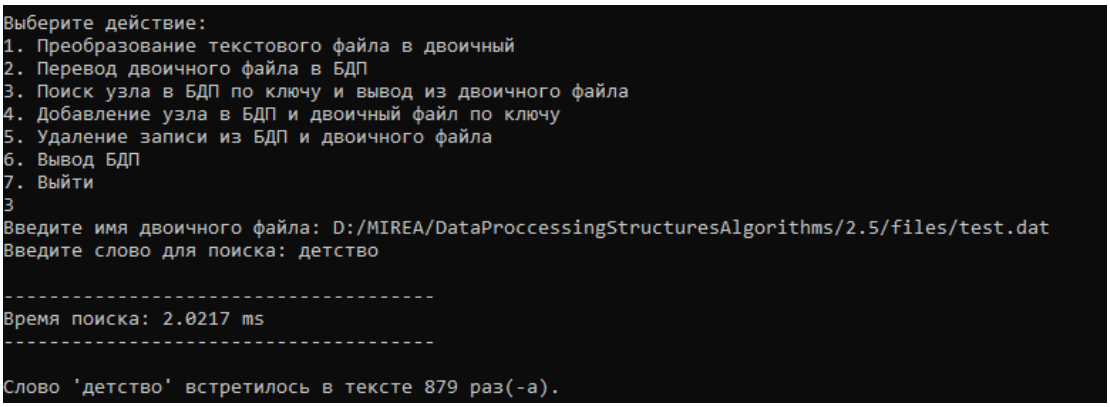

Рисунок 4 – Тестирование функции поиска записи в БДП и вывода записи из двоичного файла

На Рисунке 5 представлен результат тестирования функции добавления записи в БДП и двоичный файл. Как видно из тестирования, запись успешно добавлена и в БДП, и в файл, то есть после добавления, возможно найти запись.

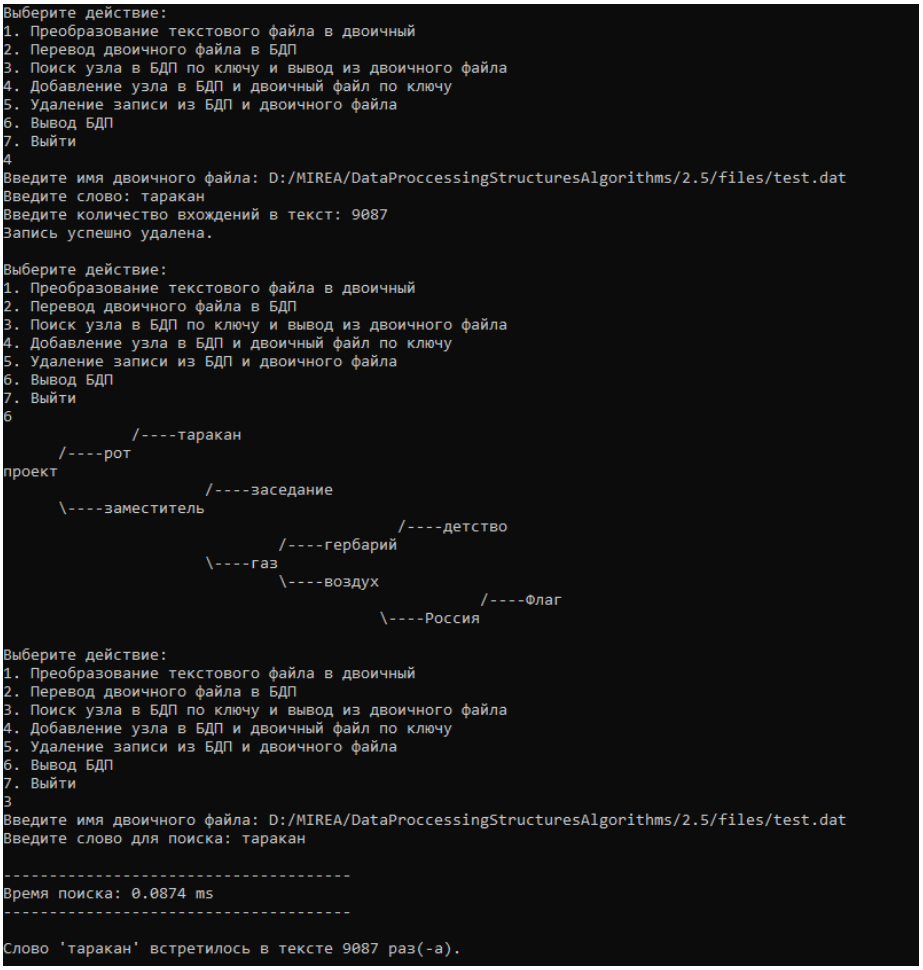

Рисунок 5 – Тестирование функции добавления записи в БДП и двоичный файл

На Рисунке 6 представлен результат тестирования функции удаления записи из БДП и двоичного файла. Удаление корневого узла также обрабатывается программой.

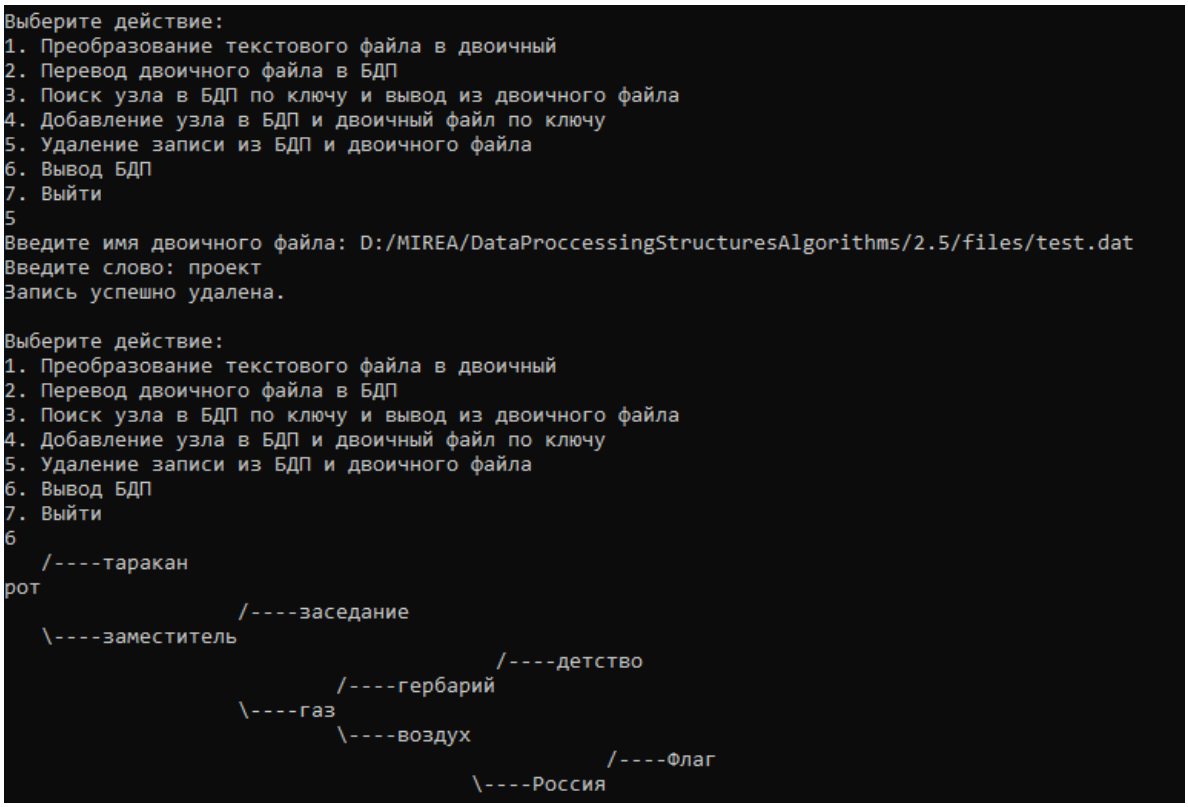

Рисунок 6 – Тестирование функции удаления записи из БДП и двоичного файла

На Рисунке 7 представлен результат тестирования функции поиска записи в БДП и вывода записи из двоичного файла после удаления записи. Изначально запись с ключом «детство» была последней, но после добавления записи «таракан» она стала последней записью в файле, значит на эту запись была заменена в файле удаляемая. После перестановки в дереве и перемещения записи в файле функция поиска работает корректно.

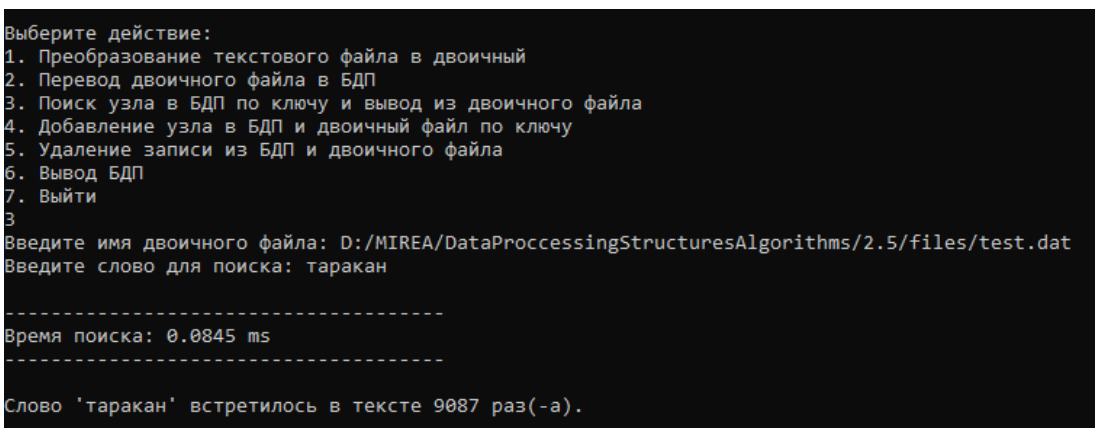

Рисунок 7 – Тестирование функции поиска записи в БДП и вывода записи из двоичного файла после удаления записи

2 ЗАДАНИЕ №2

2.1 Постановка задачи

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

Разработать класс СДП с учетом дерева варианта №17. 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).

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

Дано. Файл двоичный с записями фиксированной длины. Структура записи файла представлена в Таблице 2.

Таблица 2 – Структура записи файла

Поле	char name[30]	unsigned int count
Назначение	Слово из текста	Количество вхождений слова в текст

Результат. Приложение, выполняющее операции поиска, добавления и удаления записей из бинарного файла с помощью СДП (АВЛ).

2.2 Подход к решению

2.2.1 Структура узла АВЛ-дерева

АВЛ-дерево реализовано как класс с закрытыми вспомогательными методами и открытыми методами-оболочками для непосредственного взаимодействия с ними.

АВЛ-дерево состоит из узлов, структура которых представлена в Листинге 14. Для более удобного взаимодействия с узлами АВЛ-дерева и

поддержания принципа инкапсуляции в узле АВЛ-дерева определена внутренняя структура, которая используется для хранения ключа и позиции данного элемента в файле.

Листинг 14 – Структура узла АВЛ-дерева

```
#pragma once
#include <iostream>
#include <string>
using namespace std;

struct Node {
    struct record {
        string key;
        int position;
        record(string _key, int _position) : key(_key), position(_position)
{};
    } data;
    size_t height;

    Node* left;
    Node* right;
    Node(string _key, int _position) : data(_key, _position), height(1),
left(nullptr), right(nullptr) {};
};
```

2.2.2 Структура и методы АВЛ-дерева

Прототипы основных и вспомогательных методов структуры АВЛ-дерево представлены в Листинге 15.

Листинг 15 – Структура АВЛ-дерево

```
struct AVLtree {
     Node* root;
     AVLtree() : root(nullptr) {};
     AVLtree(Node* _root) : root(_root) {};

     Node* search(string key);
     int getIndex(string key);
     void add(string _key, int _position);
     int erase(string key);
     void print();
     ~AVLtree();

private:
     Node* searchFrom(string key, Node* current);
     Node* addTo(Node* current, Node* new_node);
     Node* remove(Node* current, string key, int& position);
     Node* removeMin(Node* current);

     size_t getHeight(Node* current);
     void updateHeight(Node* current);
     int getBalance(Node* current);

     Node* saveBalance(Node* current);
     Node* fixRight(Node* current);
     Node* fixLeft(Node* current);
```

```
    Node* getMin(Node* from);
    void printSubTree(Node* current, char from = ' ', size_t from_size = 0,
size_t level = 0);
    void clear(Node* current);
};
```

- Метод search(string key)
    - Описание: метод-оболочка для метода поиска в АВЛ-дереве, вызывает метод поиска в АВЛ-дереве от корневого узла.
    - Предусловие: key – строковое значение, ключ искомого узла.
    - Постусловие: возвращает указатель на найденный узел, если такой существует в АВЛ-дереве, иначе – нулевой указатель.
- Метод searchFrom(Node* current, string key)
    - Описание: вызывается методом-оболочкой search(string key), реализует поиск элемента в АВЛ-дереве.
    - Предусловие: current указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом-оболочкой передаётся указатель на корень АВЛ-дерева; string key – ключ, поиск которого осуществляется в АВЛ-дереве.
    - Постусловие: возвращает указатель на узел, ключ которого равен искомому, если такой существует, в противном случае – вернёт нулевой указатель.
- Метод getIndex(string key)
    - Описание: метод для получения позиции узла с заданным ключом в файле.
    - Предусловие: key – строковое значение, ключ искомого узла.
    - Постусловие: возвращает позицию узла с заданным ключом в файле, если узел с заданным ключом найден, иначе – -1.
- Метод getHeight(Node* current)
    - Описание: получает высоту узла, переданного в качестве аргумента.
    - Предусловие: current – указатель на узел, для которого необходимо узнать показатель высоты.

- Постусловие: возвращает целочисленное беззнаковое – высоту переданного в качестве аргумента узла, если такой узел существует.

- Метод updateHeight(Node* current)
  - Описание: обновляет высоту узла, переданного в качестве аргумента.
  - Предусловие: current – указатель на узел, для которого необходимо обновить показатель высоты.
  - Постусловие: высота узла current, переданного в качестве аргумента, обновлена.
- Метод getBalance(Node* current)
  - Описание: получает показатель баланса узла, переданного в качестве аргумента.
  - Предусловие: current – указатель на узел, для которого необходимо узнать показатель баланса.
  - Постусловие: возвращает целое число – баланс узла current, переданного в качестве аргумента.
- Метод add(string _key, int _position)
  - Описание: метод-оболочка для метода добавления узла в АВЛ-дерево, создаёт новый экземпляр узла АВЛ-дерева и вызывает метод добавления созданного узла в АВЛ-дерево от корневого узла, если корневой узел существует, иначе – добавляет в АВЛ-дерево созданный узел в качестве корневого.
  - Предусловие: _key – строковое значение, ключ создаваемого узла, _position – целочисленное значение, позиция создаваемого узла в файле.
  - Постусловие: в АВЛ-дерево добавлен узел с ключом, равным _key, и позицией в файле _position.
- Метод addTo(Node* current, Node* new_node)
  - Описание: вызывается методом-оболочкой add(string _key, int _position), реализует добавление элемента в АВЛ-дерево.

- Предусловие: current – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом-оболочкой передаётся указатель на корень АВЛ-дерева, если корень не пустой; new_node – указатель на новый узел, который необходимо добавить в АВЛ-дерево.
- Постусловие: возвращает указатель на текущий узел, для использования текущего узла предыдущим уровнем рекурсии, при отсутствии текущего узла возвращает указатель на новый узел, таким образом, прекращая цепочку рекурсивных вызовов. В АВЛ-дерево добавлен новый узел.

- Метод saveBalance(Node* current)
  - Описание: вызывает метод обновления высоты для узла current, переданного в качестве аргумента, затем проверяет нужна ли балансировка и, при необходимости, вызывает методы поворотов.
  - Предусловие: current – указатель на узел, показатель высоты которого необходимо обновить.
  - Постусловие: показатель высоты узла current, переданного в качестве аргумента, обновлён; АВЛ-дерево осталось сбалансированным; возвращает указатель на новый узел, стоящий на месте предыдущего в случае перебалансировки АВЛ-дерева (при выполнении поворота место текущего узла занимает один из дочерних узлов).
- Метод fixRight(Node* current)
  - Описание: осуществляет правый поворот.
  - Предусловие: current – указатель на узел, относительно которого необходимо произвести поворот.
  - Постусловие: показатель высоты узла current и его левого наследника обновлены; возвращает узел, который встанет на место узла current, переданного в качестве аргумента (левый дочерний).
- Метод fixLeft(Node* current)

- Описание: осуществляет левый поворот.
- Предусловие: current – указатель на узел, относительно которого необходимо произвести поворот.
- Постусловие: показатель высоты узла current и его правого наследника обновлены; возвращает узел, который встанет на место узла current, переданного в качестве аргумента (правый дочерний).

- Метод erase(string key)
  - Описание: метод-оболочка для метода удаления узла из АВЛ-дерева, инициализирует целочисленную переменную -1, вызывает метод удаления узла из АВЛ-дерева от корневого узла, передавая созданную целочисленную переменную для записи в неё позиции найденного узла в файле.
  - Предусловие: key – строковое значение, ключ удаляемого узла.
  - Постусловие: из АВЛ-дерева удалён узел с ключом, равным _key, если такой узел существовал в АВЛ-дереве, возвращает позицию удалённого узла в файле.
- Метод remove(Node* current, string key, int& position)
  - Описание: вызывается методом-оболочкой erase(string key), реализует удаление узла из АВЛ-дерева.
  - Предусловие: current – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом-оболочкой передаётся указатель на корень АВЛ-дерева; string key – ключ, удаление которого осуществляется из АВЛ-дерева, int& position – ссылка на целочисленную переменную, которая создаётся в методе-оболочке, ссылка на переменную предназначается для записи в неё позиции в файле удаляемого элемента.
  - Постусловие: из АВЛ-дерева удалён узел с заданным ключом, в целочисленную переменную по ссылке записана позиция удаляемого узла в файле, возвращает указатель на узел, с которым

осуществлялась работа на текущем уровне рекурсии для использования текущего узла предыдущим уровнем рекурсии, если текущий узел – удаляемый, возвращает нулевой указатель или указатель на замещающий узел, тем самым осуществляя удаление узла из БДП.

- Метод removeMin(Node* current)
    - Описание: вызывается методом удаления узла из АВЛ-дерева remove(Node* current, string key, int& position), реализует удаление замещающего узла из АВЛ-дерева.
    - Предусловие: current – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом удаления узла из АВЛ-дерева передаётся указатель на правое поддерево узла, который необходимо удалить.
    - Постусловие: из АВЛ-дерева удалён узел с заданным ключом, возвращает указатель на узел, с которым осуществлялась работа на текущем уровне рекурсии для использования текущего узла предыдущим уровнем рекурсии, если текущий узел – удаляемый (наименьший узел в правом поддереве от «реально» удаляемого узла), возвращает указатель на его правое поддерево, таким образом, переставляя самый меньший узел в правом поддереве от удаляемого узла на место удаляемого узла и перевешивая правое поддерево замещающего узла на родителя замещающего узла.
- Метод getMin(Node* from)
    - Описание: вызывается методом удаления узла из АВЛ-дерева remove(Node* current, string key, int& position), реализует поиск наименьшего узла поддерева с корнем в удаляемом узле; найденный наименьший узел станет замещающим.
    - Предусловие: from – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом удаления

узла из АВЛ-дерева передаётся указатель на правое поддерево узла, который необходимо удалить.
    - Постусловие: возвращает указатель на наименьший узел поддерева с корнем в узле from (правом потомке удаляемого узла).
- Метод print()
    - Описание: метод-оболочка для метода вывода АВЛ-дерева, вызывает метод вывода АВЛ-дерева от корневого узла, если дерево не пустое, иначе – выводит сообщение о том, что в дереве нет узлов.
    - Предусловие: отсутствует.
    - Постусловие: в консоль выведено АВЛ-дерево.
- Метод printSubTree(Node* current, char from, size_t from_size, size_t from_level)
    - Описание: вызывается методом-оболочкой print(), реализует вывод АВЛ-дерева в консоль.
    - Предусловие: current – указатель на узел, с которым на данном уровне рекурсии осуществляет работу метод, при вызове методом-оболочкой передаётся указатель на корень АВЛ-дерева, если АВЛ-дерево не пустое; from – символьное значение для отображения связи между родительским и дочерним узлом, по умолчанию равно пробелу; from_size – целочисленное беззнаковое – размер информационной части родительского узла в символах, используется для выравнивания вывода, по умолчанию равно нулю; from_level – целочисленное беззнаковое – текущий уровень рекурсии (глубина узла, который выводится при данном рекурсивном вызове, в дереве), используется для выравнивания вывода, по умолчанию равно нулю.
    - Постусловие: в консоль выведено АВЛ-дерево с показателями баланса для узлов.

Также в структуре АВЛ-дерева определены методы:

- Конструктор по умолчанию для создания пустого дерева.

- Параметризованный конструктор для создания дерева и инициализации корня заданным значением.
- void clear(Node* current) – удаление всех узлов дерева, включая корневой узел.
- Деструктор.

2.2.3 Алгоритмы операций на псевдокоде

Алгоритм поиска узла в АВЛ-дереве на псевдокоде представлен в Листинге 16. Предусловие: current – указатель на корень АВЛ-дерева, key – ключ искомого узла. Постусловие: возвращает указатель на узел, ключ которого равен искомому, если такой существует, в противном случае – вернёт нулевой указатель.

Листинг 16 – Алгоритм поиска узла в АВЛ-дереве

```
searchFrom(current, key){
    if (current = NULL || key = current.data.key) then
        return current;
    else
        if (key < current.data.key) then
            return searchFrom(current.left, key);
        else
            return searchFrom(current.right, key);
        endif
    endif
}
```

Алгоритм вставки узла в АВЛ-дерево на псевдокоде представлен в Листинге 17. Предусловие: current – указатель на корень АВЛ-дерева, _key – строковое значение, ключ создаваемого узла, _position – целочисленное значение, позиция создаваемого узла в файле. Постусловие: в АВЛ-дерево добавлен узел с ключом, равным _key, и позицией в файле _position, при каждом рекурсивном вызове возвращает указатель на текущий узел после проведения для текущего узла операции сохранения баланса.

Листинг 17 – Алгоритм вставки узла в АВЛ-дерево

```
addTo(current, _key, _position) {
    if (current = NULL) then
        current <- new Node(_key, _position);
    endif

    if (_key < current.data.key) then
        current.left <- addTo(current.left, _key, _position);
    else
```

```
        current.right <- addTo(current.right, _key, _position);
    endif

    return saveBalance(current);
}
```

Алгоритм удаления узла из АВЛ-дерева на псевдокоде представлен в Листинге 18. У метода удаления есть вспомогательные методы: метод нахождения наименьшего узла в поддереве с корнем в узле, который передан в качестве аргумента – getMin(Node* from); метод «вытеснения» замещающего узла из поддерева с корнем в узле, который передан в качестве аргумента – removeMin(Node* current). Предусловие: current – указатель на корень АВЛ-дерева, key – ключ, удаление которого осуществляется из АВЛ-дерева, position – ссылка на целочисленную переменную, которая предназначается для записи позиции в файле удаляемого элемента. Постусловие: из АВЛ-дерева удалён узел с заданным ключом, в целочисленную переменную по ссылке записана позиция удаляемого узла в файле.

Листинг 18 – Алгоритм удаления узла из АВЛ-дерева

```
remove(current, key, position) {
    if (current = NULL) then
        return NULL;
    endif

    if (key < current.data.key) then
        current.left <- erase(current.left, key, position);
    else if (key > current.data.key) then
        current.right <- erase(current.right, key, position);
    else
        right <- current.right;
        left <- current.left;
        position <- current.data.position;
        delete current;

        if (right = NULL) then
            return left;
        endif

        auxiliary <- getMin(right);
        auxiliary.right <- removeMin(right);
        auxiliary.left <- left;
        if (root = NULL) then
            root <- auxiliary;
        endif
        return saveBalance(auxiliary);
    endif

    return saveBalance(current);
}
```

```
getMin(from){
    if (from.left = NULL) then
        return from.right;
    endif
    return getMin(from.left);
}

removeMin(current){
    if (current.left = NULL) then
        return current.right;
    endif
    current.left <- removeMin(current.left);
    return saveBalance(current);
}
```

Алгоритм поиска в АВЛ-дереве полностью идентичен алгоритму поиска в БДП, алгоритмы вставки и удаления узла из этих деревьев различаются только вызовом операции сохранения баланса в конце процедуры добавления или удаления узла.

В Листинге 19 представлен алгоритм определения необходимости балансировки дерева после обновления показателя высоты в узле (алгоритм сохранения баланса) на псевдокоде. Предусловие: current – указатель на узел, для которого необходимо обновить баланс и, возможно, провести балансировку. Постусловие: вызывает методы поворотов, необходимые для сохранения баланса в дереве, возвращает указатель на узел, который стоит на месте current, переданного в качестве аргумента.

Листинг 19 – Алгоритм обновления показателя высоты и сохранения баланса

```
saveBalance(current) {
    updateHeight(current);
    if (getBalance(current) = 2) then
        if (getBalance(current.right) < 0) then
            current.right <- fixRight(current.right);
        endif

        return fixLeft(current);

    else if (getBalance(current) = -2) then
        if (getBalance(current.left) > 0) then
            current.left <- fixLeft(current.left);
        endif

        return fixRight(current);
    endif

    return current;
}
```

Алгоритм сохранения баланса использует алгоритмы поворотов для сохранения баланса в дереве. Алгоритм левого поворота на псевдокоде представлен в Листинге 20.

Листинг 20 – Алгоритм левого поворота

```
fixLeft(current) {
    if (current = root) then
        root <- current.right;
    endif

    Node* center <- current.right;
    Current.right <- center.left;
    center.left <- current;
    updateHeight(current);
    updateHeight(center);
    return center;
}
```

Алгоритм правого поворота на псевдокоде представлен в Листинге 21.

Листинг 21 – Алгоритм правого поворота

```
fixRight(current) {
    if (current = root) then
        this.root <- current.left;
    endif

    Node* center <- current.left;
    Current.left <- center.right;
    center.right <- current;
    updateHeight(current);
    updateHeight(center);
    return center;
}
```

2.3 Двоичный файл из записей фиксированного размера

В Листинге 22 представлена структура записи файла.

Листинг 22 – Структура записи файла

```
struct word {
    char name[30];
    unsigned int count;
};
```

На первый взгляд, размер одной записи данной структуры равен сумме её полей, то есть 30 Байт для массива типа char и 4 Байта для переменной типа int – 34 Байта в сумме, однако это не так. Язык программирования C++ для обеспечения скорости доступа к элементам памяти и безопасности при работе с памятью применяет Байты заполнения: неиспользуемые байты памяти, которые вставляются между переменными в структуре, чтобы гарантировать, что каждая переменная начинается с адреса памяти, выровненного с ее типом

данных. В данном случае размер массива name должен быть кратен 4 (размеру переменной типа int). По этой причине в структуре происходит выравнивание: между массивом name и переменной count есть два неиспользуемых Байта, вследствие чего одна запись данной структуры весит 36 Байтов.

В целях экономии места можно сократить размер массива char на два символа, тогда вес каждой записи такой структуры станет меньше на целых 4 Байта. Также можно увеличить размер массива char на два элемента при этом не увеличивая затраты памяти. Однако, размер структуры принят равным 36 Байтам при любом случае. Система также показывает размер 36 Байт, как показано на Рисунке 8.

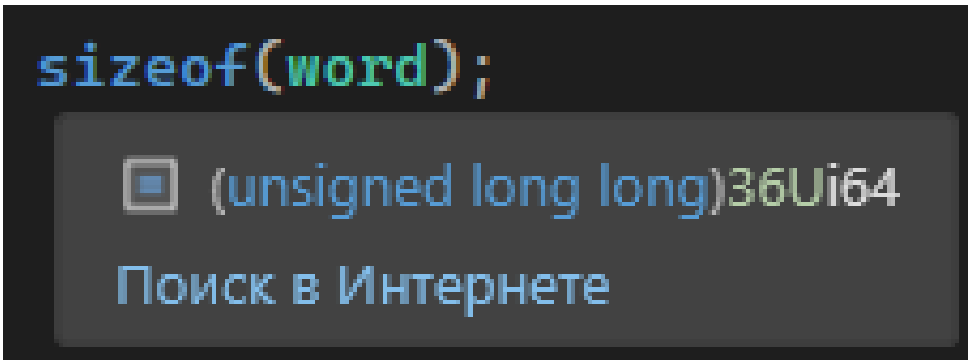

Рисунок 8 **– Размер одного экземпляра структуры word**

2.4 Прототипы операций по управления двоичным файлом

Прототипы операций по управлению двоичным файлом представлены в Листинге 23.

Листинг 23 – Прототипы операций по управлению двоичным файлом

```
void text2bin(istream& text_file, ostream& bin_file);
void bin2tree(AVLtree& tree, istream& file, int& position);
word findWord(AVLtree& tree, istream& file, string key);
void addWord(AVLtree& tree, ostream& file, string key, unsigned int count,
int& position);
bool eraseWord(AVLtree& tree, fstream& file, string key, string file_path);
```

Операции по управлению бинарным файлом:

- text2bin(istream& text_file, ostream& bin_file) – перевод текстового файла в двоичный.
- bin2tree(AVLtree& tree, istream& file, int& position) – добавление записей в существующее АВЛ-дерево из двоичного файла.

- findWord(AVLtree & tree, istream& file, string key) – поиск записи по ключу в АВЛ-дереве и вывод из двоичного файла.
- addWord(AVLtree & tree, ostream& file, string key, unsigned int count, int& position) – добавление записи в АВЛ-дерево и двоичный файл.

eraseWord(AVLtree & tree, fstream& file, string key, string file_path) – удаление записи из АВЛ-дерева и двоичного файла.

2.5 Код приложения

В Листинге 24 представлен заголовочный файл, определяющий структуру узла АВЛ-дерева.

Листинг 24 – Node.h

```
#pragma once
#include <iostream>
#include <string>
using namespace std;

struct Node {
    struct record {
        string key;
        int position;
        record(string _key, int _position) : key(_key), position(_position)
{};
    } data;
    size_t height;

    Node* left;
    Node* right;
    Node(string _key, int _position) : data(_key, _position), height(1),
left(nullptr), right(nullptr) {};
};
```

В Листинге 25 представлен заголовочный файл, определяющий структуру АВЛ-дерева.

Листинг 25 – AVLtree.h

```
#pragma once
#include "Node.h"

struct AVLtree {
    Node* root;
    AVLtree() : root(nullptr) {};
    AVLtree(Node* _root) : root(_root) {};
    Node* search(string key);
    int getIndex(string key);
    void add(string _key, int _position);
    int erase(string key);
    void print();
    ~AVLtree();
private:
```

```
    Node* searchFrom(string key, Node* current);
    Node* addTo(Node* current, Node* new_node);
    Node* remove(Node* current, string key, int& position);
    Node* removeMin(Node* current);
    size_t getHeight(Node* current);
    void updateHeight(Node* current);
    int getBalance(Node* current);
    Node* saveBalance(Node* current);
    Node* fixRight(Node* current);
    Node* fixLeft(Node* current);
    Node* getMin(Node* from);
    void printSubTree(Node* current, char from = ' ', size_t from_size = 0,
size_t level = 0);
    void clear(Node* current);
};
```

В Листинге 26 представлена реализация методов управления АВЛ-деревом.

Листинг 26 – AVLtree.cpp

```
#include "AVLtree.h"

Node* AVLtree::search(string key) {
    return searchFrom(key, this->root);
}

Node* AVLtree::searchFrom(string key, Node* current) {
    if (current == nullptr || key == current->data.key) {
        return current;
    }
    else {
        if (key < current->data.key) {
            return searchFrom(key, current->left);
        }
        else {
            return searchFrom(key, current->right);
        }
    }
}

int AVLtree::getIndex(string key) {
    Node* found = search(key);
    if (found == nullptr) {
        return -1;
    }
    return found->data.position;
}

size_t AVLtree::getHeight(Node* current) {
    if (current == nullptr) {
        return 0;
    }
    return current->height;
}

void AVLtree::updateHeight(Node* current) {
    current->height = max(getHeight(current->left), getHeight(current-
>right)) + 1;
}

int AVLtree::getBalance(Node* current) {
    return getHeight(current->right) - getHeight(current->left);
```

```
}

void AVLtree::add(string _key, int _position) {
    this->root = addTo(this->root, new Node(_key, _position));
}

Node* AVLtree::addTo(Node* current, Node* new_node) {
    if (current == nullptr) {
        return new_node;
    }

    if (new_node->data.key < current->data.key) {
        current->left = addTo(current->left, new_node);
    }
    else {
        current->right = addTo(current->right, new_node);
    }
    return saveBalance(current);
}

Node* AVLtree::saveBalance(Node* current) {
    updateHeight(current);
    if (getBalance(current) == 2) {
        if (getBalance(current->right) < 0) {
            current->right = fixRight(current->right);
        }
        return fixLeft(current);
    }
    else if (getBalance(current) == -2) {
        if (getBalance(current->left) > 0) {
            current->left = fixLeft(current->left);
        }
        return fixRight(current);
    }
    return current;
}

Node* AVLtree::fixRight(Node* current) { // Left node in Left subtree
    if (current == root) {
        this->root = current->left;
    }

    Node* center = current->left;
    current->left = center->right;
    center->right = current;

    updateHeight(current);
    updateHeight(center);
    return center;
}

Node* AVLtree::fixLeft(Node* current) { // Right node in Right subtree
    if (current == root) {
        this->root = current->right;
    }

    Node* center = current->right;
    current->right = center->left;
    center->left = current;

    updateHeight(current);
    updateHeight(center);
    return center;
```

```
}

int AVLtree::erase(string key) {
    int position = -1;
    this->root = remove(this->root, key, position);
    return position;
}

Node* AVLtree::remove(Node* current, string key, int& position) {
    if (current == nullptr) {
        return nullptr;
    }

    if (key < current->data.key) {
        current->left = remove(current->left, key, position);
    }
    else if (key > current->data.key) {
        current->right = remove(current->right, key, position);
    }
    else {
        Node* right = current->right, * left = current->left;
        position = current->data.position;
        delete current;

        if (right == nullptr) {
            return left;
        }

        Node* auxiliary = getMin(right);
        auxiliary->right = removeMin(right);
        auxiliary->left = left;
        return saveBalance(auxiliary);
    }
    return saveBalance(current);
}

Node* AVLtree::removeMin(Node* current) {
    if (current->left == nullptr) {
        return current->right;
    }
    current->left = removeMin(current->left);
    return saveBalance(current);
}

Node* AVLtree::getMin(Node* from) {
    if (from->left == nullptr) {
        return from;
    }
    return getMin(from->left);
}

void AVLtree::print() {
    if (this->root == nullptr) {
        cout << "The tree is empty!" << endl;
    }
    else {
        printSubTree(this->root);
    }
}

void AVLtree::printSubTree(Node* current, char from, size_t from_size, size_t
level) {
    if (current != nullptr) {
```

```
        printSubTree(current->right, '/', from_size + current-
>data.key.size(), level + 1);

        cout << (level > 0 ? (level > 1 ? string(from_size + (level - 1) * 5,
' ') : string(from_size, ' ')) + from + "----" : "");
        cout << current->data.key;
        cout << " (b=" << getBalance(current) << ") " << endl;

        printSubTree(current->left, '\\', from_size + current-
>data.key.size(), level + 1);
    }
}

void AVLtree::clear(Node* current) {
    if (current->left != nullptr) {
        clear(current->left);
    }

    if (current->right != nullptr) {
        clear(current->right);
    }

    delete current;
};

AVLtree::~AVLtree() {
    if (this->root != nullptr) {
        clear(this->root);
    }
}
```

В Листинге 27 представлен заголовочный файл модуля управления бинарным файлом.

Листинг 27 – FileMethods.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include <fstream>
#include <io.h>
#include <fcntl.h>
#include "AVLtree.h"

struct word {
    char name[30];
    unsigned int count;
};

void text2bin(istream& text_file, ostream& bin_file);
void bin2tree(AVLtree& tree, istream& file, int& position);
word findWord(AVLtree& tree, istream& file, string key);
void addWord(AVLtree& tree, ostream& file, string key, unsigned int count,
int& position);
bool eraseWord(AVLtree& tree, fstream& file, string key, string file_path);
```

В Листинге 28 представлен заголовочный файл модуля управления бинарным файлом.

Листинг 28 – FileMethods.cpp

```
#include "FileMethods.h"
```

```
void text2bin(istream& text_file, ostream& bin_file) {
    while (!text_file.eof()) {
        word current;
        int i = 0;

        do {
            text_file.get(current.name[i]);
        } while (current.name[i++] != '\n');
        current.name[i - 1] = '\0';

        text_file >> current.count;
        text_file.get();

        bin_file.write((char*)&current, sizeof(word));
    }
}

void bin2tree(AVLtree& tree, istream& file, int& position) {
    position = 0;
    word current;

    file.read((char*)&current, sizeof(word));
    while (!file.eof()) {
        tree.add(current.name, position++);
        file.read((char*)&current, sizeof(word));
    }
}

word findWord(AVLtree& tree, istream& file, string key) {
    word current;

    int index = tree.getIndex(key);
    if (index == -1) {
        current.name[0] = '\0';
    }
    else {
        file.seekg((index) * sizeof(word), ios::beg);
        file.read((char*)&current, sizeof(word));

        if (file.bad() || file.fail()) {
            current.name[0] = '\0';
        }
    }
    return current;
}

void addWord(AVLtree& tree, ostream& file, string key, unsigned int count,
int& position) {
    word current = word();
    strcpy(current.name, key.c_str());
    current.count = count;

    tree.add(key, position++);
    file.write((char*)&current, sizeof(word));
}

bool eraseWord(AVLtree& tree, fstream& file, string key, string file_path) {
    int index = tree.erase(key);
    if (index == -1) {
        return false;
    }

    word last;
```

```
    file.seekg(-(int)sizeof(word), ios::end);
    file.read((char*)&last, sizeof(word));

    if (last.name != key) {
        // Record couldn't be deleted if pointer would still be in that part
of file
        file.seekg(ios::beg);
        file.seekp(index * sizeof(word), ios::beg);
        file.write((char*)&last, sizeof(word));
        tree.search(last.name)->data.position = index;
    }

    int fh;
    if (_sopen_s(&fh, file_path.c_str(), _O_RDWR, _SH_DENYNO, _S_IREAD |
_S_IWRITE) == 0) {
        if (!(_chsize(fh, (_filelength(fh) - sizeof(word))) == 0)) {
            return false;
        }
        _close(fh);
    }

    /*
        * FOR UNIX: CLOSE FILE, THAN RESIZE IT
        #include <filesystem>
        auto p = filesystem::path(file_path);
        filesystem::resize_file(p, (size - 1) * sizeof(word));
        */

    return true;
}
```

В Листинге 29 представлена реализация приложения.

Листинг 29 – main.cpp

```
#include "fileMethods.h"
#include <chrono>

int main() {
      system("chcp 1251");
      string text_file, bin_file;
      ifstream fin;
      ofstream fout;
      fstream file;

      //D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/test.txt
      //D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/test.dat

      //D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/data.txt
      //D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/data.dat

      AVLtree* tree = new AVLtree();
      int position = 0;
      int action;
      while (true)
      {
            cout << endl << "Выберите действие:" << endl;
            cout << "1. Преобразование текстового файла в двоичный" << endl;
            cout << "2. Перевод двоичного файла в AVL-дерево" << endl;
            cout << "3. Поиск узла в AVL-дереве по ключу и вывод из двоичного
файла" << endl;
            cout << "4. Добавление узла в AVL-дерево и двоичный файл по ключу"
<< endl;
```

```
            cout << "5. Удаление записи из AVL-дерева и двоичного файла" <<
endl;
            cout << "6. Вывод AVL-дерева" << endl;
            cout << "7. Выйти" << endl;

            cin >> action;
            switch (action) {
            case 1: {
                  cout << "Введите имя текстового файла: ";
                  cin >> text_file;
                  cout << "Введите имя двоичного файла: ";
                  cin >> bin_file;

                  fin.open(text_file, ios::in);
                  if (fin.is_open()) {
                        fout.open(bin_file, ios::binary | ios::out);
                        text2bin(fin, fout);
                        if (fin.bad() || fout.bad()) {
                              cout << "Ошибка при создании двоичного файла из
текстового." << endl;
                              return 1;
                        }
                        cout << "Двоичный файл успешно создан." << endl;
                        fin.close();
                        fout.close();
                  }

                  else {
                        cout << "Текстовый файл не найден или не существует."
<< endl;
                  }
                  break;
            }

            case 2: {
                  cout << "Введите имя двоичного файла: ";
                  cin >> bin_file;
                  fin.open(bin_file, ios::binary | ios::in);

                  if (fin.is_open()) {
                        bin2tree(*tree, fin, position);
                        if (fin.bad()) {
                              cout << "Ошибка создания AVL-дерева из данных
двоичного файла." << endl;
                              return 1;
                        }
                        cout << "AVL-дерево успешно создано." << endl;
                        fin.close();
                  }

                  else {
                        cout << "Двоичный файл не найден или не существует."
<< endl;
                  }
                  break;
            }

            case 3: {
                  cout << "Введите имя двоичного файла: ";
                  cin >> bin_file;
                  fin.open(bin_file, ios::binary | ios::in);

                  if (fin.is_open()) {
```

```
                        string key;
                        cout << "Введите слово для поиска: ";
                        cin >> key;

                        auto start = chrono::high_resolution_clock::now();
                        word found = findWord(*tree, fin, key);
                        auto end = chrono::high_resolution_clock::now();

                        cout << endl << "-----------------------------------
-" << endl;
                        cout << "Время поиска: " <<
chrono::duration_cast<chrono::nanoseconds>(end - start).count() / 1e6 << "
ms";
                        cout << endl << "-----------------------------------
-" << endl << endl;

                        if (fin.bad()) {
                              cout << "Ошибка при поиске записи в файле." <<
endl;
                              return 1;
                        }

                        if (found.name[0] != '\0') {
                              cout << "Слово '" << found.name << "'
встретилось в тексте " << found.count << " раз(-а)." << endl;
                        }
                        else {
                              cout << "Запись не найдена." << endl;
                        }
                        fin.close();
                  }

                  else {
                        cout << "Двоичный файл не найден или не существует."
<< endl;
                  }
                  break;
            }

            case 4: {
                  cout << "Введите имя двоичного файла: ";
                  cin >> bin_file;
                  file.open(bin_file, ios::out | ios::app | ios::binary);
                  if (file.is_open()) {
                        string key;
                        unsigned int count;
                        cout << "Введите слово: ";
                        cin >> key;
                        cout << "Введите количество вхождений в текст: ";
                        cin >> count;

                        addWord(*tree, file, key, count, position);
                        if (fin.bad()) {
                              cout << "Ошибка при добавлении слова в файл." <<
endl;
                              return 1;
                        }
                        cout << "Запись успешно удалена." << endl;
                        file.close();
                  }
                  else {
                        cout << "Двоичный файл не найден или не существует."
<< endl;
```

```
                  }
                  break;
            }

            case 5: {
                  cout << "Введите имя двоичного файла: ";
                  cin >> bin_file;
                  file.open(bin_file, ios::in | ios::out | ios::binary);
                  if (file.is_open()) {
                        string key;
                        cout << "Введите слово: ";
                        cin >> key;
                        bool status = eraseWord(*tree, file, key, bin_file);
                        if (fin.bad()) {
                              cout << "Ошибка при удалении слова из файла." <<
endl;
                              return 1;
                        }
                        if (status) {
                              cout << "Запись успешно удалена." << endl;
                              position--;
                        }
                        else {
                              cout << "Запись с таким ключом не найдена." <<
endl;
                        }
                        file.close();
                  }
                  else {
                        cout << "Двоичный файл не найден или не существует."
<< endl;
                  }
                  break;
            }

            case 6: {
                  tree->print();
                  break;
            }

            default: {
                  delete tree;
                  return 0;
            }
            }
      }
}
```

2.6 Содержание текстового файла

На Рисунке 9 представлено содержимое текстового тестового файла.

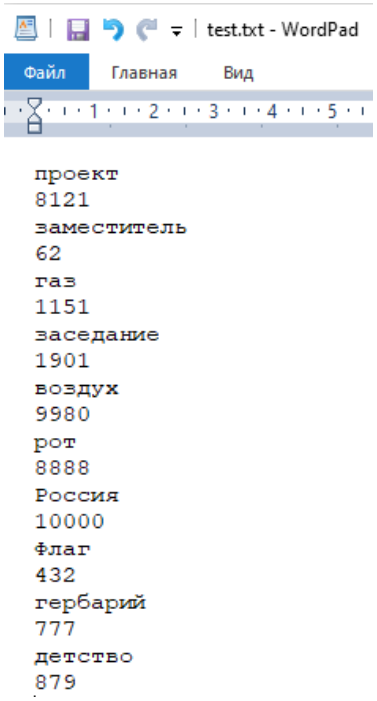

Рисунок 9 – Содержание тестового файла

2.7 Результаты тестирования

На Рисунке 10 представлен результат тестирования функции построения АВЛ-дерева по записям из двоичного файла, полученного из заранее подготовленного текстового. АВЛ-дерево успешно построено и выведено в консоль, следовательно, метод добавления элемента в АВЛ-дерево, метод вывода АВЛ-дерева в консоль и функция построения АВЛ-дерева по данным из двоичного файла работают корректно.

```
Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в AVL-дерево
3. Поиск узла в AVL-дереве по ключу и вывод из двоичного файла
4. Добавление узла в AVL-дерево и двоичный файл по ключу
5. Удаление записи из AVL-дерева и двоичного файла
6. Вывод AVL-дерева
7. Выйти
2
Введите имя двоичного файла: D:/MIREA/DataProccessingStructuresAlgorithms/2.5/files/test.dat
AVL-дерево успешно создано.

Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в AVL-дерево
3. Поиск узла в AVL-дереве по ключу и вывод из двоичного файла
4. Добавление узла в AVL-дерево и двоичный файл по ключу
5. Удаление записи из AVL-дерева и двоичного файла
6. Вывод AVL-дерева
7. Выйти
6
                    /----рот (b=0)
          /----проект (b=0)
                    \----заседание (b=0)
заместитель (b=-1)
                              /----детство (b=0)
                    /----гербарий (b=0)
                              \----газ (b=0)
          \----воздух (b=0)
                         /----Флаг (b=0)
                    \----Россия (b=1)
```

Рисунок 10 – Тестирование функции построения АВЛ-дерева по записям из двоичного файла

На Рисунке 11 представлен результат тестирования функции поиска записи в АВЛ-дереве и вывода записи из двоичного файла. Запись успешно найдена и выведена.

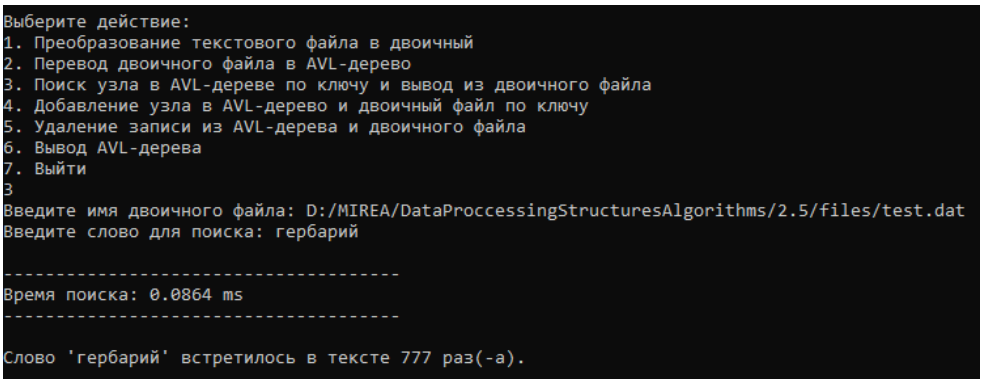

Рисунок 11 – Тестирование функции поиска записи в АВЛ-дереве и вывода записи из двоичного файла

На Рисунке 12 представлен результат тестирования функции добавления записи в АВЛ-дерево и двоичный файл. Как видно из тестирования, запись успешно добавлена и в АВЛ-дерево, и в файл, то есть после добавления, возможно найти запись.

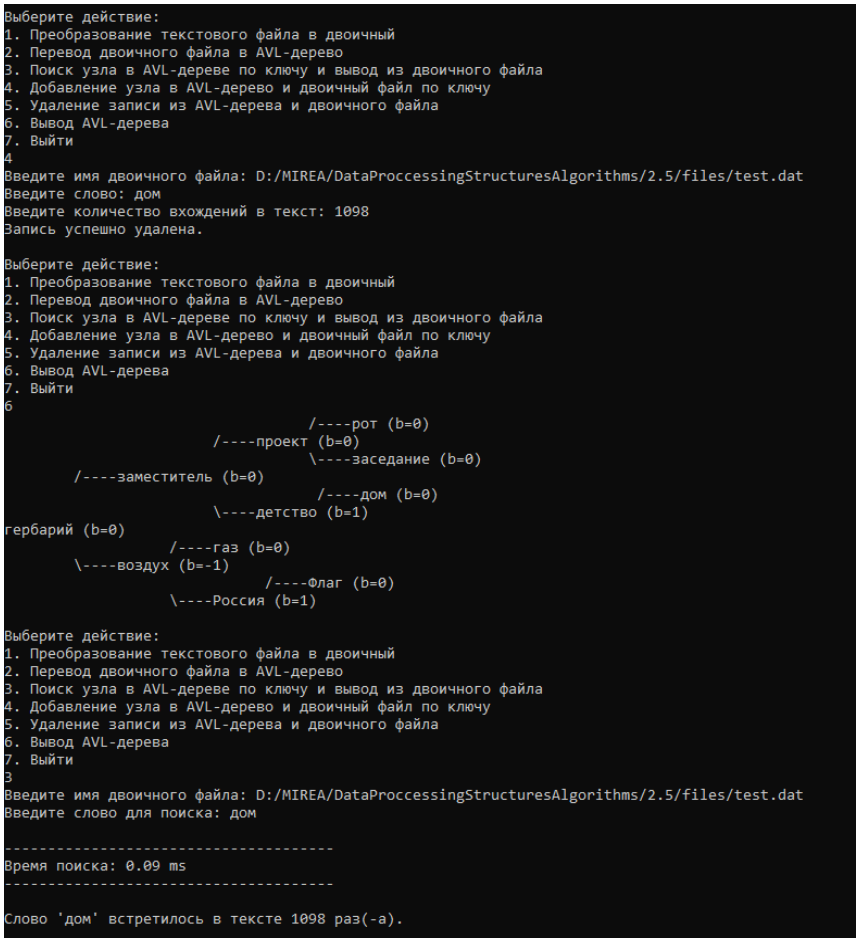

Рисунок 12 – Тестирование функции добавления записи в АВЛ-дерево и двоичный файл

В предыдущем тестовом примере прямо во время добавления нового узла в АВЛ-дерево и двоичный файл понадобилось произведение операции восстановления баланса – операция произвела лево-правый поворот: левый поворот для левого дочернего узла («воздух»), в котором был нарушен баланс («заместитель»), а затем правый попорот для узла, в котором был нарушен баланс. Новым корнем дерева впоследствии поворотов стал узел «гербарий».

На Рисунке 13 представлен результат тестирования функции удаления записи из АВЛ-дерева и двоичного файла.

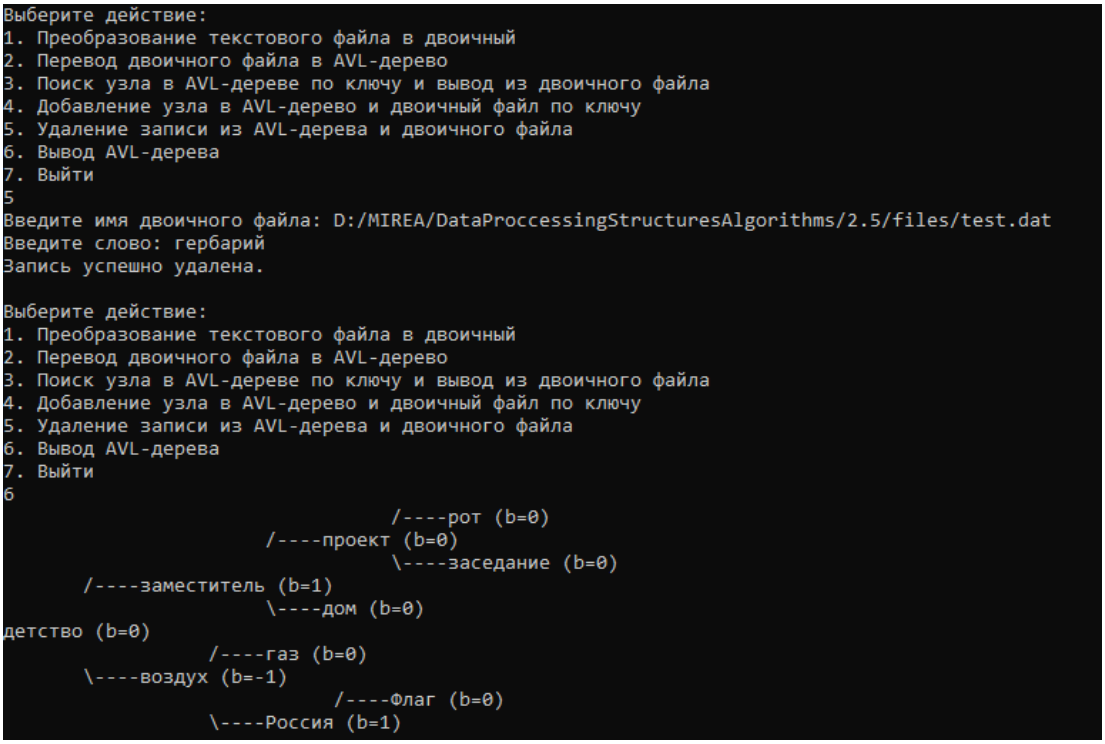

Рисунок 13 – Тестирование функции удаления записи из АВЛ-дерева и двоичного файла

Вследствие удаления узла из дерева баланс не был нарушен, поэтому выполнения операции сохранения баланса не требуется. Приложение обработало удаление корневого узла дерева.

На Рисунке 14 представлен результат тестирования функции поиска записи в АВЛ-дереве и вывода записи из двоичного файла после удаления записи. Изначально запись с ключом «детство» была последней, но после добавления записи «дом» стал последней записью в файле, значит на эту запись была заменена в файле удаляемая. После перестановки в дереве и перемещения записи в файле функция поиска работает корректно.

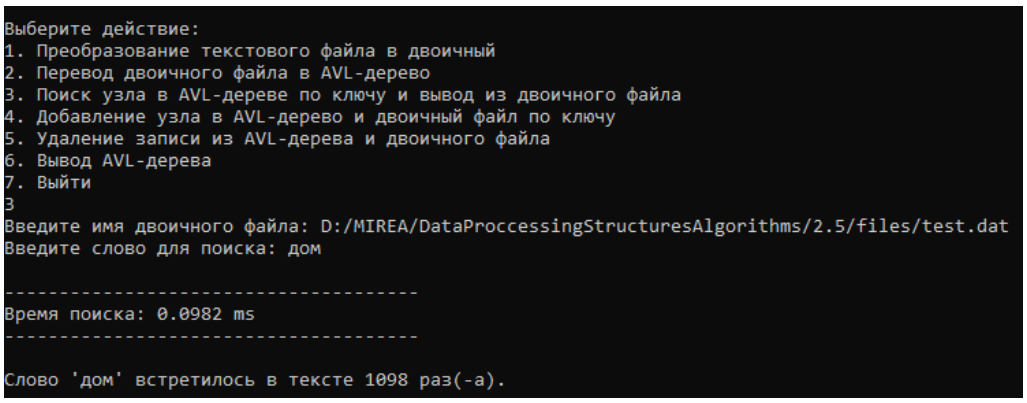

Рисунок 14 – Тестирование функции поиска записи в АВЛ-дереве и вывода записи из двоичного файла после удаления записи

2.8 Среднее число выполненных поворотов

Количество выполненных поворотов при включении ключей в дерево при формировании дерева из двоичного файла представлено на Рисунке 15.

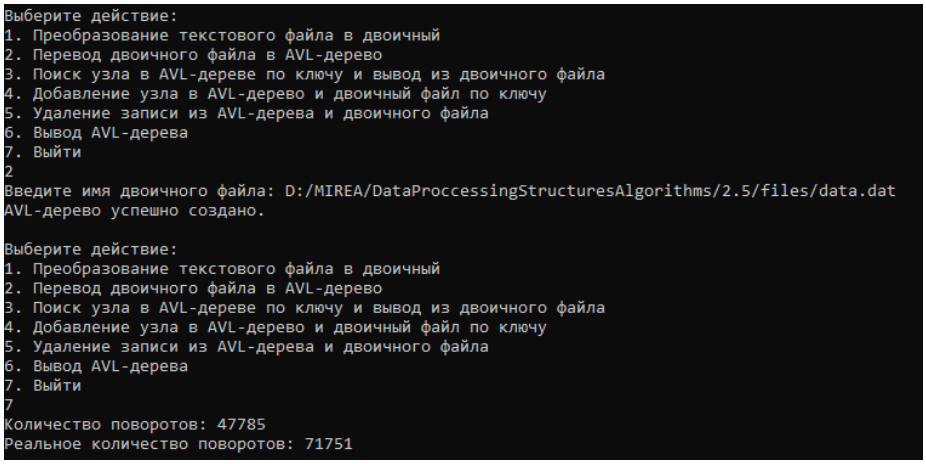

Рисунок 15 – Количество выполненных поворотов

В выводе программы «количество поворотов» — это количество раз, когда программе пришлось прибегнуть к осуществлению одного из четырёх видов поворота для сохранения баланса в дереве; «Реальное количество поворотов» — это количество раз, когда программе пришлось вызвать функцию правого или левого поворота непосредственно. Для более справедливой оценки при вычислении среднего числа поворотов будет использоваться «реальное число поворотов».

Среднее число выполненных поворотов вычисляется как отношение числа поворотов к количеству записей в двоичном файле, из которого построено дерево. Таким образом, среднее число поворотов равно: 71751 / 100.000 = 0.7175.

3 ЗАДАНИЕ №3

3.1 Постановка задачи

Анализ алгоритма поиска записи с заданным ключом при применении структур данных:

- Хеш-таблица;
- бинарное дерево поиска;
- СДП.

Требования по выполнению задания:

- Протестировать на данных:
  - небольшого объема;
  - большого объема.
- Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице.
- Оформить таблицу результатов по форме

3.2 Решение

В Таблице 3 представлен анализ эффективности алгоритма поиска записи с заданным ключом при применении различных структур данных.

Таблица 3 – Анализ алгоритма поиска записи с заданным ключом

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 (мс)
Хэш-таблица	500	O(n)	8, 0.1245
БДП			42, 0.2034
АВЛ-дерево			35, 0.0892
Хэш-таблица	100.000		3, 0.1287
БДП			103, 0.2943
АВЛ-дерево			49, 0.1181

Из приведённой таблицы следует, что самой эффективной структурой для поиска является хэш-таблица, а использование АВЛ-дерева ускоряет поиск по сравнению с обычными БДП. Плюсом АВЛ-дерева по сравнению с Хэш-таблицей является «постоянство» - дерево постоянно балансируется, практически не оказывая нагрузки на вычислительную машину, а хэш-таблица с увеличением размера может требовать большой вычислительной мощности для рехэширования.

4 ВЫВОД

В процессе выполнения данной практической работы были изучены принципы работы с БДП и АВЛ-деревом, была реализована программа управления бинарным файлом посредством БДП и АВЛ-дерева. Программа прошла тестирование и работает корректно.

Анализ эффективности алгоритма поиска записи с заданным ключом при применении различных структур данных показал, что наилучше решение для уменьшения времени поиска – Хэш-таблица, однако АВЛ-дерево тоже показало неплохие результаты, особенно в совокупности с маленькой нагрузкой на вычислительную машину.