



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 №3

по дисциплине

«Структуры и алгоритмы обработки данных»

Тема. Применение хеш-таблицы для поиска данных в двоичном файле с
записями фиксированной длины

Выполнил студент группы ИКБО-04-22

Основин А.И.

Принял старший преподаватель

Скворцова Л.А.

Москва 2023

СОДЕРЖАНИЕ

1	Постановка задачи.....	4
1.1	Условие задачи	4
1.2	Требования по выполнению.....	4
2	Ход решения	5
2.1	Двоичный файл из записей фиксированного размера	5
2.1.1	Структура записи файла.....	5
2.1.2	Размер записи файла.....	5
2.1.3	Прототипы операций по управления двоичным файлом	6
2.2	Хеш-таблица	8
2.2.1	Структура элемента таблицы	8
2.2.2	Код элемента таблицы.....	8
2.2.3	Описание алгоритмов операций.....	9
2.2.4	Алгоритм поиска записи с заданным ключом в файле посредством хеш-таблицы	10
3	Код приложения	11
3.1	Модуль управления хэш-таблицей.....	11
3.2	Модуль управления двоичным файлом	15
3.3	Модуль управления двоичным файлом посредством хэш- таблицы	22
3.4	Функция main	23
4	Результаты тестирования.....	27
4.1	Вычисление хэш-функции	27
4.2	Тестирование модуля для управления хэш-таблицей.....	27
4.3	Тестирование модуля для управления двоичным файлом с помощью хэш-таблицы.....	31

4.4	Сложность операций над хэш-таблицей.....	37
5	ВЫВОД.....	40

1 ПОСТАНОВКА ЗАДАЧИ

1.1 Условие задачи

Разработать приложение, которое использует хеш-таблицу для организации прямого доступа к записям файла, структура записи которого представлена в Таблице 1.

Таблица 1 – Задание варианта

№	Тип хеш-таблицы (метод разрешения коллизии)	Структура записи двоичного файла
17	Открытый адрес (смещение на 1)	Частотный словарь: <u>слово</u> , количество вхождений в текст

1.2 Требования по выполнению

Дано: двоичный файл с записями фиксированной длины. Структура записи файла согласно варианту 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: хеш-таблица.

2 ХОД РЕШЕНИЯ

2.1 Двоичный файл из записей фиксированного размера

2.1.1 Структура записи файла

Структура записи файла из кода представлена в Листинге 1.

Листинг 1 – Структура записи

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

2.1.2 Размер записи файла

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.

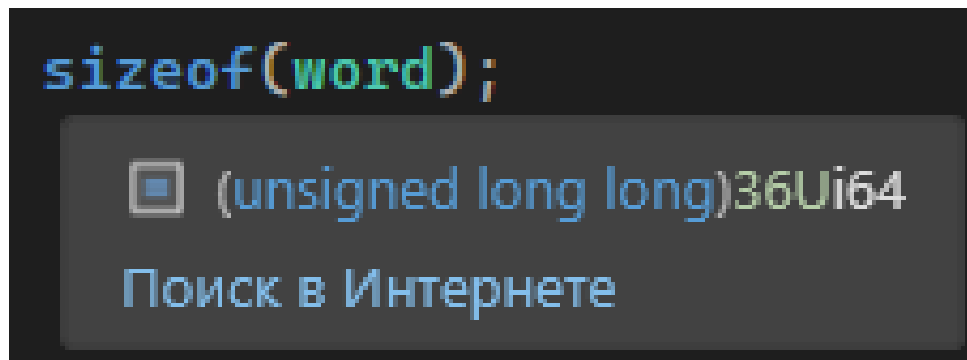


Рисунок 1 – Размер одного экземпляра структуры word

2.1.3 Прототипы операций по управлению двоичным файлом

Прототипы операций по управлению двоичным файлом представлены в Листинге 2.

Листинг 2 – FileMethods.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <io.h>
#include <fcntl.h>
using namespace std;

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

void text2bin(istream& text_file, ostream& bin_file);
void bin2text(istream& bin_file, ostream& text_file);
void print_bin(istream& file);
word get_word(istream& file, int index);
bool delete_word(fstream& file, string key, string file_path);
word get_widest(istream& file);
void add_word(ostream& file, string new_word);
void count_words(fstream& file, vector<string> words);
int testBinF();
```

Функция `text2bin(istream& text_file, ostream& bin_file)` реализует запись данных из файла, связанного с потоком `text_file`, в двоичный файл, связанный с потоком `bin_file`. Предусловие: `text_file` — ссылка на поток чтения из текстового файла; `bin_file` — ссылка на поток записи в бинарный файл. Постусловие: в двоичном файле записаны данные из текстового файла; возвращаемое значение отсутствует.

Функция `bin2text(istream& bin_file, ostream& text_file)` реализует запись данных из файла, связанного с потоком `bin_file`, в текстовый файл, связанный с потоком `text_file`. Предусловие: `bin_file` — ссылка на поток чтения из двоичного файла; `text_file` — ссылка на поток записи в текстовый файл. Постусловие: в текстовом файле записаны данные из двоичного файла; возвращаемое значение отсутствует.

Функция `print_bin(istream& file)` реализует вывод в консоль записей из двоичного файла, связанного с потоком `bin_file`. Предусловие: `file` — ссылка на поток чтения из двоичного файла. Постусловие: в консоль выведены данные из двоичного файла; возвращаемое значение отсутствует.

Функция `get_word(istream& file, int index)` реализует получение записи с порядковым номером `index` из двоичного файла, связанного с потоком `file`. Предусловие: `file` — ссылка на поток чтения из двоичного файла, `index` — целочисленная переменная, порядковый номер. Постусловие: возвращаемое значение — объект структуры `word`.

Функция `delete_word(fstream& file, string key, string file_path)` реализует удаление записи о слове `key` из двоичного файла, связанного с потоком `file`. Предусловие: `file` — ссылка на файловый поток, связанный с двоичным файлом; `key` — строка, хранит искомое слово; `file_path` — строка, хранит путь до двоичного файла. Постусловие: из двоичного файла удалена запись с нужным ключом; возвращаемое значение — булева переменная, показывает успешность удаления записи.

Функция `get_widest(istream& file)` реализует получение самого часто встречающегося в тексте слова. Предусловие: `file` — ссылка на поток чтения из двоичного файла. Постусловие: возвращаемое значение — объект структуры `word`.

Функция `add_word(ostream& file, string new_word)` реализует добавление новой записи о слове в конец двоичного файла. Предусловие: `file` — ссылка на поток записи в двоичный файл, `new_word` — строковая переменная, слово,

которое необходимо записать. Постусловие: в конец двоичного файла записано новое слово; возвращаемое значение отсутствует.

Функция `count_words(fstream& file, vector<string> words)` реализует обновление количества вхождений некоторых слов в текст. Предусловие: `file` — ссылка на файловый поток, связанный с двоичным файлом, `words` — контейнер типа `vector` элементов строкового типа, слова, для которых необходимо обновить количество вхождений. Постусловие: в двоичном файле обновлено количество вхождений переданных слов в текст; возвращаемое значение отсутствует.

Функция `testBinF()` реализует тестирование операций модуля для работы с бинарными файлами. Предусловие: функция вызывается из основной функции программы, передаваемые параметры отсутствуют. Постусловие: возвращаемое значение — вердикт работы программы по завершении тестирования модуля.

2.2 Хеш-таблица

2.2.1 Структура элемента таблицы

В Таблице 3 представлена структура элемента хеш-таблицы.

Таблица 3 – Структура элемента хеш-таблицы

Поле	string key	int position	bool is_deleted
Назначение	Уникальный ключ записи – слово из частотного словаря	Порядковый номер записи в двоичном файле	Состояние записи – удалена или нет

2.2.2 Код элемента таблицы

Код элемента таблицы и реализация структуры таблицы приведены в Листинге 3.


```
struct record {
    string key;
    int position;
    bool is_deleted;

    record() : key(""), position(-1), is_deleted(false) {};
    record(string _key, int _position) : key(_key), position(_position),
is_deleted(false) {};
};

struct HashTable {
    size_t size;
    size_t filled;
    record** records;

    HashTable();
    ~HashTable();
};
```

2.2.3 Описание алгоритмов операций

Функция `insert_key(HashTable& table, string key, int position)` реализует добавление нового элемента в хеш-таблицу при условии сохранения уникальности ключа; при достижении показателя переполнения таблицы (когда более 75% таблицы заполнено), вызывает функцию рехеширования. Предусловие: `table` – ссылка на объект класса `HashTable`; `key` – строковое значение, ключ, который необходимо вставить в таблицу, `position` – порядковый номер ключа в двоичном файле. Постусловие: в таблицу добавлен элемент в соответствии со своим хэшем; возвращает булеву переменную – показатель успешности добавления нового элемента в таблицу.

Функция `get_index(HashTable& table, string key)` реализует получение из таблицы порядкового номера элемента по ключу. Предусловие: `table` – ссылка на объект типа `HashTable`; `key` – строковое значение, ключ, который необходимо найти в таблице. Постусловие: возвращает позицию найденного элемента в файле, если элемент с переданным ключом существует, иначе – возвращает -1.

Функция `delete_key(HashTable& table, string key)` реализует удаление из таблицы элемента по ключу. Предусловие: `table` – ссылка на объект типа `HashTable`; `key` – строковое значение, ключ в таблице. Постусловие:

возвращает позицию удалённого элемента в файле, если в таблице существовал элемент с переданным ключом, иначе – возвращает -1.

2.2.4 Алгоритм поиска записи с заданным ключом в файле посредством хеш-таблицы

Для успешного нахождения записи с заданным ключом в двоичном файле посредством хеш-таблицы необходимо вычислить хэш ключа. Хэш-функция реализована как остаток от деления на текущий размер таблицы суммы произведений ascii-кода каждой буквы слова и индекса буквы в слове.

После вычисления значения хэша для заданного ключа необходимо произвести проверку на совпадение заданного ключа с ключом элемента, находящегося по полученному адресу в хэш-таблице. Если ключи не совпадают, необходимо последовательно перебирать элементы хеш-таблицы, что приводит к линейной сложности. Если найден элемент таблицы с значением ключа, равным заданному, необходимо вернуть его адрес в бинарном файле.

После получения адреса искомого элемента в двоичном файле, необходимо воспользоваться механизмом прямого доступа к записи файла по полученному адресу, предварительно проверив, что найденный индекс элемента не равен -1, то есть что данный элемент существует.

3 КОД ПРИЛОЖЕНИЯ

3.1 Модуль управления хэш-таблицей

В Листинге 4 представлен заголовочный файл модуля управления хэш-таблицей.

Листинг 4 – HashTable.h

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

struct record {
    string key;
    int position;
    bool is_deleted;

    record() : key(""), position(-1), is_deleted(false) {};
    record(string _key, int _position) : key(_key), position(_position),
is_deleted(false) {};
};

struct HashTable {
    size_t size;
    size_t filled;
    record** records;

    HashTable();
    ~HashTable();
};

size_t hash_key(string key, size_t size);
bool insert_key(HashTable &table, string key, int position);
record* find_key(HashTable &table, string key);
int get_index(HashTable &table, string key);
int delete_key(HashTable &table, string key);
void rehash(HashTable &table);
void printHashTable(const HashTable &table);
int testHashT();
```

В Листинге 5 представлен файл модуля управления хэш-таблицей.

Листинг 5 – HashTable.cpp

```
#include "HashTable.h"

HashTable::HashTable() {
    this->size = 5;
    this->filled = 0;

    this->records = new record * [size];
    for (int i = 0; i < this->size; ++i) {
        this->records[i] = nullptr;
    }
}

HashTable::~~HashTable() {
```

```

        delete[] this->records;
    }

    size_t hash_key(string key, size_t size) {
        size_t hash = 0;
        for (size_t i = 0; i < key.size(); ++i) {
            hash += ((unsigned char) key[i]) * (i + 1); // for compiler
independence
        }
        return (hash % size);
    }

    bool insert_key(HashTable &table, string key, int position) {
        bool unique = true;
        size_t index = hash_key(key, table.size);
        size_t stop_index = index;
        while (index < table.size && table.records[index] != nullptr) {
            if (table.records[index]->key == key && !table.records[index]-
>is_deleted) {
                unique = false;
                break;
            }
            index++;
        }

        if (index >= table.size && unique) {
            for (index = 0; index < stop_index; ++index) {
                if (table.records[index] != nullptr) {
                    if (table.records[index]->key == key &&
!table.records[index]->is_deleted) {
                        unique = false;
                        break;
                    }
                }
                else {
                    break;
                }
            }
        }

        if (index < table.size && unique) {
            table.records[index] = new record(key, position);
            if (++table.filled > 0.75 * table.size) {
                rehash(table);
            }
            return true;
        }
        return false;
    }

    record* find_key(HashTable &table, string key) {
        record* found = nullptr;
        size_t index = hash_key(key, table.size);
        size_t stop_index = index;
        while (index < table.size) {
            if (table.records[index] != nullptr) {
                if (table.records[index]->key == key) {
                    if (!table.records[index]->is_deleted) {
                        found = table.records[index];
                        break;
                    }
                }
            }
            ++index;
        }
    }

```

```

        }
        else {
            break;
        }
    }

    if (found == nullptr) {
        for (index = 0; index < stop_index; ++index) {
            if (table.records[index] != nullptr) {
                if (table.records[index]->key == key) {
                    if (!table.records[index]->is_deleted) {
                        found = table.records[index];
                        break;
                    }
                }
            }
            else {
                break;
            }
        }
    }

    return found;
}

int get_index(HashTable &table, string key) {
    record* found = find_key(table, key);
    if (found != nullptr) {
        return found->position;
    }
    return -1;
}

int delete_key(HashTable &table, string key) {
    record* rubbish = find_key(table, key);
    if (rubbish != nullptr) {
        rubbish->is_deleted = true;
        return rubbish->position;
    }
    return -1;
}

void rehash(HashTable &table) {
    size_t rehashed_size = table.size << 1;
    cout << "В хэш-таблице " << table.filled << " запис(-и/-ей)." << endl;
    cout << "Рехэширование хэш-таблицы с " << table.size << " элементов до "
    << rehashed_size << " элементов..." << endl;

    record** rehashed_records = new record * [rehashed_size];
    for (size_t i = 0; i < rehashed_size; ++i) {
        rehashed_records[i] = nullptr;
    }

    table.filled = 0;
    for (size_t i = 0; i < table.size; ++i) {
        if (table.records[i] != nullptr && !table.records[i]->is_deleted) {
            size_t new_index = hash_key(table.records[i]->key,
rehashed_size);
            size_t stop_index = new_index;
            while (new_index < rehashed_size && rehashed_records[new_index]
!= nullptr) {
                new_index++;
            }

```

```

        if (new_index >= rehashed_size) {
            for (new_index = 0; new_index < stop_index; ++new_index) {
                if (rehashed_records[new_index] == nullptr) {
                    break;
                }
            }
        }
        rehashed_records[new_index] = new record(table.records[i]->key,
table.records[i]->position);
        table.filled++;
    }
}
table.size *= 2;
delete[] table.records;
table.records = rehashed_records;
cout << "Рехэширование выполнено успешно." << endl;
}

void printHashTable(const HashTable &table) {
    cout << "|-----|" << endl;
    for (size_t i = 0; i < table.size; ++i) {
        cout << "| " << i << ": ";
        if (table.records[i] != nullptr) {
            if (!table.records[i]->is_deleted) {
                cout << setw(29) << table.records[i]->key << "|";
            }
            else {
                cout << setw(30) << "DELETED|";
            }
        }
        else {
            cout << setw(30) << "|";
        }
        cout << endl;
    }
    cout << "|-----|" << endl;
}

int testHashT() {
    HashTable table;
    int position = 0;
    string key;

    while (true) {
        cout << endl << "Выберите действие:" << endl;
        cout << "1. Добавление элемента в таблицу" << endl;
        cout << "2. Поиск элемента в таблице" << endl;
        cout << "3. Удаление элемента из таблицы" << endl;
        cout << "4. Вывод хэш-таблицы" << endl;
        cout << "5. Выйти" << endl;

        int action;
        cin >> action;
        switch (action) {
            case 1: {
                cout << "Введите ключ записи: ";
                cin >> key;
                if (insert_key(table, key, position++)) {
                    cout << "Ключ успешно добавлен в хэш-таблицу:" << endl;
                    printHashTable(table);
                }
            }
            else {

```

```

        cout << "Нарушено условие уникальности ключа, дублирование
ключа запрещено." << endl;
    }
    break;
}

case 2: {
    cout << "Введите ключ записи: ";
    cin >> key;
    int index = get_index(table, key);
    if (index == -1) {
        cout << "В хэш-таблице не найден элемент с заданным ключом."
<< endl;
    }
    else {
        cout << "Порядковый номер элемента в двоичном файле: " <<
index << endl;
    }
    break;
}

case 3: {
    cout << "Введите ключ записи: ";
    cin >> key;
    int index = delete_key(table, key);
    if (index == -1) {
        cout << "В хэш-таблице не найден элемент с заданным ключом."
<< endl;
    }
    else {
        cout << "Элемент с позицией " << index << " удалён." << endl;
        printHashTable(table);
    }
    break;
}

case 4: {
    printHashTable(table);
    break;
}

default: {
    return 0;
}
}
}

```

3.2 Модуль управления двоичным файлом

В Листинге 6 представлен заголовочный файл модуля управления хэш-таблицей.

Листинг 6 – FileMethods.h

```

#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>

```

```

#include <string>
#include <vector>
#include <io.h>
#include <fcntl.h>
using namespace std;

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

void text2bin(istream& text_file, ostream& bin_file);
void bin2text(istream& bin_file, ostream& text_file);
void print_bin(istream& file);
word get_word(istream& file, int index);
bool delete_word(fstream& file, string key, string file_path);
word get_widest(istream& file);
void add_word(ostream& file, string new_word);
void count_words(fstream& file, vector<string> words);
int testBinF();

```

В Листинге 7 представлен файл модуля управления хэш-таблицей.

Листинг 7 – 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 bin2text(istream& bin_file, ostream& text_file) {
    word current;
    bin_file.read((char*)&current, sizeof(word));
    while (!bin_file.eof()) {
        text_file << current.name << "\n" << current.count;
        bin_file.read((char*)&current, sizeof(word));

        if (!bin_file.eof()) {
            text_file << "\n";
        }
    }
}

void print_bin(istream& file) {
    word current;
    int n = 1;
    file.read((char*)&current, sizeof(word));
    while (!file.eof()) {

```



```

        cout << n++ << ". " << current.name << ": " << current.count << endl;
        file.read((char*)&current, sizeof(word));
    }
}

word get_word(istream& file, int index) {
    word current;
    file.seekg((index) * sizeof(word), ios::beg);
    file.read((char*)&current, sizeof(word));

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

bool delete_word(fstream& file, string key, string file_path) {
    word last, current;
    bool status = false;

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

    // Record couldn't be deleted if pointer would still be in that part of
    file
    if (last.name != key) {
        file.seekg(ios::beg);
        file.read((char*)&current, sizeof(word));
        while (!file.eof()) {
            if (current.name == key) {
                file.seekp(-(int)sizeof(word), ios::cur);
                file.write((char*)&last, sizeof(word));
                status = true;
                break;
            }
            file.read((char*)&current, sizeof(word));
        }
    }
    else {
        status = true;
    }

    if (status) {
        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)) {
                status = 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 status;
}

word get_widest(istream& file) {
    word current, best;

```

```

        best.count = 0;

        file.read((char*)&current, sizeof(word));
        while (!file.eof()) {
            if (current.count > best.count) {
                best.count = current.count;
                strcpy_s(best.name, current.name);
            }
            file.read((char*)&current, sizeof(word));
        }

        return best;
    }

void add_word(ostream& file, string new_word) {
    word current;
    size_t pos = new_word.size();
    strncpy(current.name, new_word.c_str(), pos);
    current.name[pos] = '\\0';
    current.count = 0;
    file.write((char*)&current, sizeof(word));
}

void count_words(fstream& file, vector<string> words) {
    word current;
    file.read((char*)&current, sizeof(word));
    while (!file.eof()) {
        for (string w : words) {
            if (w == current.name) {
                ++current.count;
                file.seekp(-(int)sizeof(word), ios::cur);
                file.write((char*)&current, sizeof(word));
                file.seekg(sizeof(word), ios::cur);
                cout << current.name << " обновлено: " << current.count << endl;
                break;
            }
        }
        file.read((char*)&current, sizeof(word));
    }
}

int testBinF() {
    string text_file, bin_file;
    ifstream fin;
    ofstream fout;
    fstream fios;

    int action;
    while (true) {
        cout << endl << "Выберите действие:" << endl;
        cout << "1. Преобразовать текстовый файл в двоичный" << endl;
        cout << "2. Вывести записи из двоичного файла в текстовый" << endl;
        cout << "3. Вывести записи из двоичного файла в консоль" << endl;
        cout << "4. Получить запись из двоичного файла по порядковому номеру"
<< endl;
        cout << "5. Удалить запись из двоичного файла по ключу" << endl;
        cout << "6. Определить, какое слово встречалось в тексте чаще всего" <<
endl;
        cout << "7. Добавить в файл новую запись по слову" << endl;
        cout << "8. Обновить количество вхождений некоторых слов, увеличив их
количество на 1" << endl;
        cout << "9. 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 (!fout) {
            cout << "Ошибка при записи в файл." << endl;
            return 1;
        }
        cout << "Двоичный файл успешно записан." << endl;
        fin.close();
        fout.close();
    }
    else {
        cout << "Файл не найден или не существует." << endl;
    }
    break;
}

case 2: {
    cout << "Введите имя двоичного файла: ";
    cin >> bin_file;
    cout << "Введите имя текстового файла: ";
    cin >> text_file;
    fin.open(bin_file, ios::binary | ios::in);
    if (fin.is_open()) {
        fout.open(text_file, ios::out);
        bin2text(fin, fout);
        if (fout.bad() || fout.fail()) {
            cout << "Ошибка при записи в файл." << endl;
            return 1;
        }
        cout << "Текстовый файл успешно записан." << endl;
        fin.close();
        fout.close();
    }
    else {
        cout << "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()) {
        print_bin(fin);
        if (fin.bad()) {
            cout << "Ошибка при чтении файла." << endl;
            return 1;
        }
        fin.close();
    }
    else {
        cout << "Файл не найден или не существует." << endl;
    }
}
}

```

```

        break;
    }

    case 4: {
        cout << "Введите имя двоичного файла: ";
        cin >> bin_file;
        cout << "Введите порядковый номер записи: ";
        int number;
        cin >> number;

        fin.open(bin_file, ios::binary | ios::in);
        if (fin.is_open()) {
            word found = get_word(fin, number);
            if (found.name[0] == '\\0') {
                cout << "Введённый порядковый номер превышает количество
записей в файле." << endl;
            }
            else {
                cout << number << "-ое слово '" << found.name << "'
встречено в тексте " << found.count << " раз(-а)." << endl;
            }

            if (fin.bad() || fin.fail()) {
                cout << "Ошибка при чтении файла." << endl;
                return 1;
            }
            fin.close();
        }
        else {
            cout << "Файл не найден или не существует." << endl;
        }
        break;
    }

    case 5: {
        cout << "Введите имя двоичного файла: ";
        cin >> bin_file;
        cout << "Введите ключ (слово): ";
        string key;
        cin >> key;

        fios.open(bin_file, ios::binary | ios::in | ios::out);
        if (fios.is_open()) {
            bool status = delete_word(fios, key, bin_file);
            if (fios.bad()) {
                cout << "Ошибка при чтении файла." << endl;
                return 1;
            }

            if (!status) {
                cout << "Не удалось удалить запись по ключу." << endl;
            }
            else {
                cout << "Запись удалена." << endl;
            }
            fios.close();
        }
        else {
            cout << "Файл не найден или не существует." << endl;
        }
        break;
    }
}

```

```

case 6: {
    cout << "Введите имя двоичного файла: ";
    cin >> bin_file;
    fin.open(bin_file, ios::binary | ios::in);
    if (fin.is_open()) {
        word best = get_widest(fin);
        if (fin.bad()) {
            cout << "Ошибка при чтении файла." << endl;
            return 1;
        }
        cout << "Слово '" << best.name << "' встречалось в тексте чаще
всего: " << best.count << " раз(-а)." << endl;
        fin.close();
    }
    else {
        cout << "Файл не найден или не существует." << endl;
    }
    break;
}

case 7: {
    cout << "Введите имя двоичного файла: ";
    cin >> bin_file;
    cout << "Введите новое слово для добавления в частотный словарь: ";
    string new_word;
    cin >> new_word;

    fout.open(bin_file, ios::binary | ios::out | ios::app);
    if (fout.is_open()) {
        add_word(fout, new_word);
        if (fout.bad() || fout.fail()) {
            cout << "Ошибка при чтении файла." << endl;
            return 1;
        }

        cout << "Слово " << new_word << " успешно записано в конец
бинарного файла." << endl;
        fout.close();
    }
    else {
        cout << "Файл не найден или не существует." << endl;
    }
    break;
}

case 8: {
    cout << "Введите имя двоичного файла: ";
    cin >> bin_file;
    cout << "Вводите слова, которые необходимо посчитать (чтобы
прекратить ввод введите 'end'): " << endl;
    string new_word = "";
    vector<string> words;
    while (new_word != "end") {
        cin >> new_word;
        words.push_back(new_word);
    }
    words.pop_back();

    fios.open(bin_file, ios::binary | ios::in | ios::out);
    if (fios.is_open()) {
        count_words(fios, words);
        if (fios.bad()) {

```

```

        cout << "Ошибка при чтении файла или записи в него." <<
endl;
        return 1;
    }
    cout << "Количество вхождений слов успешно обновлено в
бинарном файле. " << endl;
    fout.close();
}
else {
    cout << "Файл не найден или не существует." << endl;
}
break;
}

default: {
    return 0;
}
}
}

```

3.3 Модуль управления двоичным файлом посредством хэш-таблицы

В Листинге 8 представлен заголовочный файл модуля управления двоичным файлом посредством хэш-таблицы.

Листинг 8 – Handler.h

```

#pragma once
#include <chrono>
#include "FileMethods.h"
#include "HashTable.h"

void bin2hash(HashTable &table, istream& file);
word find_word(HashTable &table, istream& file, string key);
bool erase_word(HashTable &table, fstream& file, string key, string
file_path);

```

В Листинге 9 представлен файл модуля управления двоичным файлом посредством хэш-таблицы.

Листинг 9 – Handler.cpp

```

#include "Handler.h"

void bin2hash(HashTable &table, istream& file) {
    int position = 0;
    word current;

    file.read((char*)&current, sizeof(word));
    while (!file.eof()) {
        if (!insert_key(table, current.name, position++)) {
            cout << "Провалена попытка вставки слова '" << current.name << "'
в хэш-таблицу" << endl;
        }
        file.read((char*)&current, sizeof(word));
    }
}

```

```

word find_word(HashTable &table, istream& file, string key) {
    word current;
    int index = get_index(table, key);
    if (index == -1) {
        current.name[0] = '\\0';
    }
    else {
        current = get_word(file, index);
    }
    return current;
}

bool erase_word(HashTable &table, fstream& file, string key, string
file_path) {
    int index = delete_key(table, 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) {
        file.seekg(ios::beg);
        file.seekp(index * sizeof(word), ios::beg);
        file.write((char*)&last, sizeof(word));
        find_key(table, last.name)->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);
    }
    return true;
}

```

3.4 Функция main

В Листинге 10 представлена реализация диалогового интерфейса на основе текстового меню в функции main.

Листинг 10 – main.cpp

```

#include "Handler.h"

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

    HashTable table = HashTable();
    int action;

```

```

while (true)
{
    cout << endl << "Выберите действие:" << endl;
    cout << "1. Преобразование текстового файла в двоичный" << endl;
    cout << "2. Перевод двоичного файла в хэш-таблицу" << endl;
    cout << "3. Поиск записи в хэш-таблице и вывод из двоичного файла"
<< endl;
    cout << "4. Удаление записи из хэш-таблицы и двоичного файла" <<
endl;
    cout << "5. Вывод хэш-таблицы" << endl;
    cout << "6. Вывод бинарного файла" << endl;
    cout << "7. Тестирование модуля управления хэш-таблицей" << endl;
    cout << "8. Тестирование модуля управления бинарным файлом" <<
endl;
    cout << "9. 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()) {
            bin2hash(table, fin);
            if (fin.bad()) {
                cout << "Ошибка создания хэш-таблицы из данных
двоичного файла." << endl;
                return 1;
            }
            cout << "Хэш-таблица успешно создана." << 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 = find_word(table, 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(-a)." << 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::in | ios::out | ios::binary);
    if (file.is_open()) {
        string key;
        cout << "Введите слово: ";
        cin >> key;
        bool status = erase_word(table, file, key, bin_file);
        if (fin.bad()) {
            cout << "Ошибка при удалении слова из файла." <<
endl;

```

```

        return 1;
    }

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

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

case 5: {
    printHashTable(table);
    break;
}

case 6: {
    cout << "Введите имя двоичного файла: ";
    cin >> bin_file;
    fin.open(bin_file, ios::binary | ios::in);
    if (fin.is_open()) {
        print_bin(fin);
        if (fin.bad()) {
            cout << "Ошибка при чтении файла." << endl;
            return 1;
        }
        fin.close();
    }

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

case 7: {
    return testHashT();
    break;
}

case 8: {
    return testBinF();
    break;
}

default: {
    return 0;
}
}
}

```

4 РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

4.1 Вычисление хэш-функции

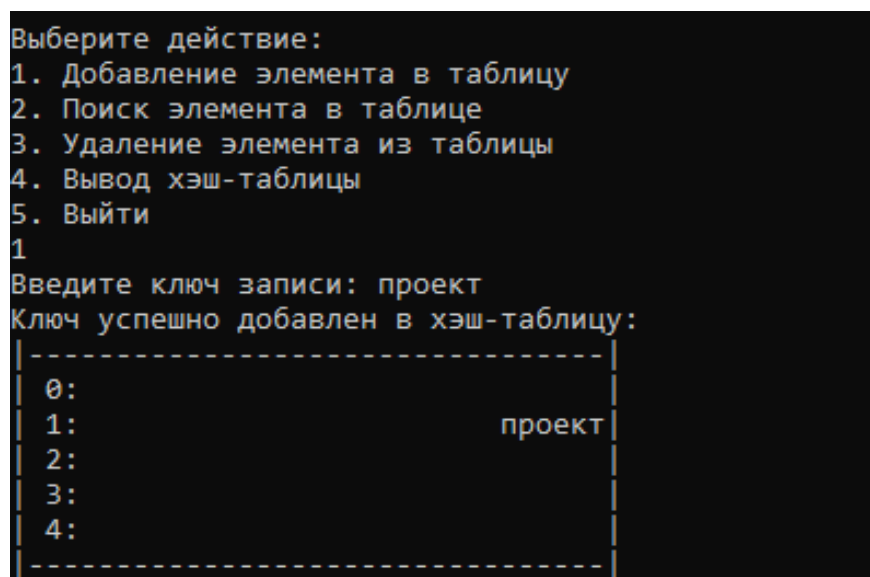
Изначальный размер хэш-таблицы равен 5. Значение хэш-функции вычисляется по формуле:

$$\text{mod}(\sum_{i=1}^n \text{ascii}(\text{key}(i)) * i),$$

где `mod` – функция, возвращающая первый аргумент по модулю второго,
`n` – текущий размер хэш-таблицы,
`ascii` – функция, возвращающая `ascii`-код символа-аргумента,
`key` – ключ, для которого вычисляется значение хэш-функции.

4.2 Тестирование модуля для управления хэш-таблицей

На Рисунке 2 представлен результат тестирования операции вставки ключа в хэш-таблицу без коллизии. Значение хэш-функции вычисляется следующим образом: $(239 * 1 + 240 * 2 + 238 * 3 + 229 * 4 + 234 * 5 + 242 * 6) \% 5 = 1$. Вся таблица свободна, следовательно, вставляем элемент с переданным ключом в хэш-таблицу по адресу 1, поле «позиции в файле» заполняется автоматически.



```
Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
1
Введите ключ записи: проект
Ключ успешно добавлен в хэш-таблицу:
|-----|
| 0:      |
| 1:      | проект
| 2:      |
| 3:      |
| 4:      |
|-----|
```

Рисунок 2 – Тестирование операции вставки ключа в хэш-таблицу без коллизии

После добавления в хэш-таблицу ключа «проект», был добавлен ключ «заместитель», хэш которого равен: $(231 * 1 + 224 * 2 + 236 * 3 + 229 * 4 + 241 * 5 + 242 * 6 + 232 * 7 + 242 * 8 + 229 * 9 + 235 * 10 + 252 * 11) \% 5 = 3$. Данный ключ был также добавлен без коллизии, так как полученный адрес хэш-таблицы свободен. После добавления в хэш-таблицу второго ключа, происходит вычисление хэша для ключа «газ»: $(227 * 1 + 224 * 2 + 231 * 3) \% 5 = 3$. Полученный для ключа «газ» адрес уже занят в хэш-таблице – коллизия, она разрешается согласно заданию варианта смещением на 1. Ключ «газ» добавлен в хэш-таблицу на место первого свободного адреса, следующего за его хэшем – 4.

На Рисунке 3 представлен результат тестирования операции вставки ключа в хэш-таблицу с коллизией.

```

Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
1
Введите ключ записи: заместитель
Ключ успешно добавлен в хэш-таблицу:
|-----|
| 0: |
| 1: |                проект |
| 2: |
| 3: |                заместитель |
| 4: |
|-----|

Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
1
Введите ключ записи: газ
Ключ успешно добавлен в хэш-таблицу:
|-----|
| 0: |
| 1: |                проект |
| 2: |
| 3: |                заместитель |
| 4: |                газ |
|-----|

```

Рисунок 3 – Тестирование операции вставки ключа в хэш-таблицу с коллизией

На Рисунке 4 представлен результат тестирования операции вставки ключа в хэш-таблицу с последующим рехэшированием.

```

Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
1
Введите ключ записи: заседание
В хэш-таблице 4 запис(-и/-ей).
Рехэширование хэш-таблицы с 5 элементов до 10 элементов...
Рехэширование выполнено успешно.
Ключ успешно добавлен в хэш-таблицу:
-----
0:
1:                                проект
2:
3:                                заместитель
4:
5:
6:
7:
8:                                заседание
9:                                газ
-----

```

Рисунок 4 – Тестирование операции вставки ключа в хэш-таблицу с последующим рехэшированием

При добавлении в хэш-таблицу с исходным размером, равным 5, четвёртого ключа «заседание» выполняется условие для рехэширования таблицы: $4 > 0.75 * 5$. После добавления ключа «заседание» в хэш-таблицу размером 5 по адресу: $(231 * 1 + 224 * 2 + 241 * 3 + 229 * 4 + 228 * 5 + 224 * 6 + 237 * 7 + 232 * 8 + 229 * 9) \% 5 = 3$, происходит пересчёт хэша каждого элемента, то есть перераспределение элементов по хэш-таблице большего размера. Новый размер хэш-таблицы равен 10. А адреса всех элементов вычисляются следующим образом:

- проект: $(239 * 1 + 240 * 2 + 238 * 3 + 229 * 4 + 234 * 5 + 242 * 6) \% 10 = 1$;
- заместитель: $(231 * 1 + 224 * 2 + 236 * 3 + 229 * 4 + 241 * 5 + 242 * 6 + 232 * 7 + 242 * 8 + 229 * 9 + 235 * 10 + 252 * 11) \% 10 = 3$;
- газ: $(227 * 1 + 224 * 2 + 231 * 3) \% 10 = 8$;
- заседание: $(231 * 1 + 224 * 2 + 241 * 3 + 229 * 4 + 228 * 5 + 224 * 6 + 237 * 7 + 232 * 8 + 229 * 9) \% 10 = 8$, коллизия, смещение на 1 – адрес свободен: 9.

На Рисунке 5 представлен результат тестирования операции поиска существующего ключа в хэш-таблице.

```
Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
2
Введите ключ записи: проект
Порядковый номер элемента в двоичном файле: 0
```

Рисунок 5 – Тестирование операции поиска существующего ключа в хэш-таблице

На Рисунке 6 представлен результат тестирования операции поиска несуществующего ключа в хэш-таблице.

```
Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
2
Введите ключ записи: qwerty
В хэш-таблице не найден элемент с заданным ключом.
```

Рисунок 6 – Тестирование операции поиска несуществующего ключа в хэш-таблице

На Рисунке 7 представлен результат тестирования операции удаления существующего ключа из хэш-таблицы. После удаления ключа из хэш-таблицы до рехэширования он отображается как «DELETED» для наглядности.

```
Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
3
Введите ключ записи: заседание
Элемент с позицией 3 удалён.
```

0:	
1:	проект
2:	
3:	заместитель
4:	
5:	
6:	
7:	
8:	DELETED
9:	газ

Рисунок 7 – Тестирование операции удаления существующего ключа из хэш-таблицы

На Рисунке 8 представлен результат тестирования операции удаления несуществующего ключа из хэш-таблицы.

```
Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
3
Введите ключ записи: qwerty
В хэш-таблице не найден элемент с заданным ключом.
```

Рисунок 8 – Тестирования операции удаления несуществующего ключа из хэш-таблицы

Тестирование операции поиска ключа, который размещен в хэш-таблице после удаленного ключа, с одним значением хэша для этих ключей представлен на Рисунке 9.

```
Выберите действие:
1. Добавление элемента в таблицу
2. Поиск элемента в таблице
3. Удаление элемента из таблицы
4. Вывод хэш-таблицы
5. Выйти
2
Введите ключ записи: газ
Порядковый номер элемента в двоичном файле: 2
```

Рисунок 9 – Тестирование операции поиска ключа, который размещен в хэш-таблице после удаленного ключа, с одним значением хэша для этих ключей

На Рисунке 9 ключ «газ» успешно найден в хэш-таблице несмотря на то, что он располагается после удалённого элемента с таким же значением хэша, как показано на Рисунке 7.

4.3 Тестирование модуля для управления двоичным файлом с помощью хэш-таблицы

В Листинге 11 приведено содержание текстового тестового файла:

Листинг 11 – Содержание тестового файла

```
1) проект
2) 8121
3) заместитель
4) 62
5) газ
6) 1151
7) заседание
8) 1901
9) воздух
10) 9980
11) рот
12) 8888
```

На Рисунке 10 представлено содержание исходного текстового файла, из которого с помощью функции модуля для управления двоичными файлами, будет создан двоичный файл.

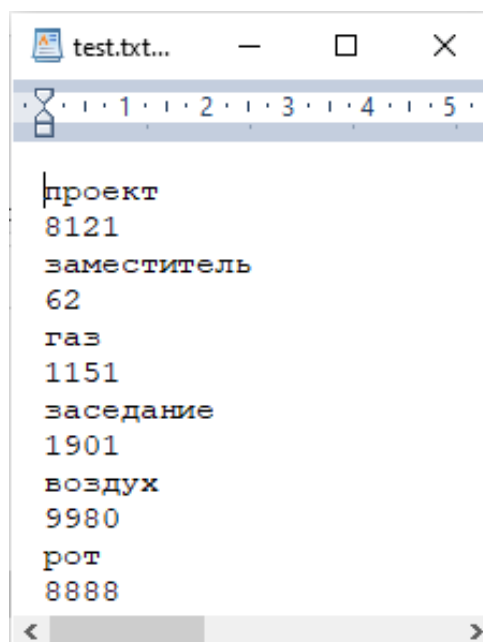


Рисунок 10 – Содержание тестового файла

На Рисунке 11 представлен процесс создания двоичного файла на основе текстового. После создания двоичного файла на его основе создаётся хэш-таблица. Изначальный размер хэш-таблицы равен 5, то есть рехэширование понадобится после добавления в хэш-таблицу четвёртого элемента ($4 > 0.75 * 5$). В тестовом двоичном файле 6 записей, также, как и в текстовом, поэтому рехэширование произойдет во время заполнения хэш-таблицы из файла. При добавлении первых четырёх записей в хэш-таблицу из двоичного файла, их адреса будут рассчитаны с помощью хэш-функции следующим образом:

- проект: $(239 * 1 + 240 * 2 + 238 * 3 + 229 * 4 + 234 * 5 + 242 * 6) \% 5 = 1$, добавление в хэш-таблицу без коллизии;
- заместитель: $(231 * 1 + 224 * 2 + 236 * 3 + 229 * 4 + 241 * 5 + 242 * 6 + 232 * 7 + 242 * 8 + 229 * 9 + 235 * 10 + 252 * 11) \% 5 = 3$, добавление в хэш-таблицу без коллизии;
- газ: $(227 * 1 + 224 * 2 + 231 * 3) \% 5 = 3$, возникновение коллизии, разрешение путём смещения на 1, добавление в хэш-таблицу под адресом 4;

- заседание: $(231 * 1 + 224 * 2 + 241 * 3 + 229 * 4 + 228 * 5 + 224 * 6 + 237 * 7 + 232 * 8 + 229 * 9) \% 5 = 3$, возникновение коллизии, разрешение путём смещения на 1, повторное возникновение коллизии, переход к началу хэш-таблицы (смещение на 1), добавление в хэш-таблицу под адресом 0. Таким образом обработано неоднократное возникновение коллизии с учётом того, что место в конце таблицы может закончиться, и добавлять элементы будет некуда.

```

Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
1
Введите имя текстового файла: D:/МИРЭА/СиАОД/2.3/files/test.txt
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/test.dat
Двоичный файл успешно создан.

Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
2
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/test.dat
В хэш-таблице 4 записи(-и/-ей).
Рехэширование хэш-таблицы с 5 элементов до 10 элементов...
-----|
0:      заседание|
1:      проект   |
2:      |         |
3:      заместитель|
4:      газ      |
-----|
Рехэширование выполнено успешно.
Хэш-таблица успешно создана.

```

Рисунок 11 - Хэш-таблица, построенная по содержанию двоичного файла до рехэширования

После этого произойдет рехэширование и хэши добавленных в хэш-таблицу записей будут рассчитаны снова (хэши будут вычисляться по тому порядку, в котором записи находятся в таблице), следующим образом:

- заседание: $(231 * 1 + 224 * 2 + 241 * 3 + 229 * 4 + 228 * 5 + 224 * 6 + 237 * 7 + 232 * 8 + 229 * 9) \% 10 = 8$, добавление в хэш-таблицу без коллизии;
- проект: $(239 * 1 + 240 * 2 + 238 * 3 + 229 * 4 + 234 * 5 + 242 * 6) \% 10 = 1$, добавление в хэш-таблицу без коллизии;

- заместитель: $(231 * 1 + 224 * 2 + 236 * 3 + 229 * 4 + 241 * 5 + 242 * 6 + 232 * 7 + 242 * 8 + 229 * 9 + 235 * 10 + 252 * 11) \% 10 = 3$, добавление в хэш-таблицу без коллизии;
- газ: $(227 * 1 + 224 * 2 + 231 * 3) \% 10 = 8$, возникновение коллизии, разрешение путём смещения на 1, добавление в хэш-таблицу под адресом 9.

По завершении рехэширования размер таблицы увеличится вдвое ($5 * 2 = 10$), и будет продолжен процесс добавления записей в хэш-таблицу из файла, оставшиеся записи будут добавлены в хэш-таблицу следующим образом:

- воздух: $(226 * 1 + 238 * 2 + 231 * 3 + 228 * 4 + 243 * 5 + 245 * 6) \% 10 = 2$, добавление в хэш-таблицу без коллизии;
- рот: $(240 * 1 + 238 * 2 + 242 * 3) \% 10 = 2$, возникновение коллизии, разрешение путём смещения на 1, повторное возникновение коллизии, смещение на 1, добавление в хэш-таблицу под адресом 4.

Таким образом, хэш-таблица, полученная из двоичного файла, будет выглядеть как показано на Рисунке 12.

Выберите действие:

1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти

5

0:	
1:	проект
2:	воздух
3:	заместитель
4:	рот
5:	
6:	
7:	
8:	заседание
9:	газ

Рисунок 12 – Хэш-таблица, построенная по содержанию двоичного файла

На Рисунке 13 представлен результат тестирования операции поиска существующего ключа в хэш-таблице.

```

1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
3
Введите имя двоичного файла:

D:/МИРЭА/СиАОД/2.3/files/test.dat
Введите слово для поиска: проект

-----
Время поиска: 1.8952 ms
-----

Слово 'проект' встретилось в тексте 8121 раз(-а).

```

Рисунок 13 – Тестирование операции поиска существующего ключа в хэш-таблице

На Рисунке 14 представлен результат тестирования операции поиска несуществующего ключа в хэш-таблице.

```

Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
3
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/test.dat
Введите слово для поиска: qwerty

-----
Время поиска: 0.0194 ms
-----

Запись не найдена.

```

Рисунок 14 – Тестирование операции поиска несуществующего ключа в хэш-таблице

На Рисунке 15 представлен результат тестирования операции удаления существующего ключа из хэш-таблицы. После удаления ключа из хэш-таблицы до рехэширования он отображается как «DELETED» для наглядности.

```
Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
4
Введите имя двоичного файла:
D:/МИРЭА/СиАОД/2.3/files/test.dat
Введите слово: заседание
Запись успешно удалена.

Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
5
|-----|
| 0:      |
| 1:      |      проект |
| 2:      |      воздух  |
| 3:      |      заместитель |
| 4:      |      рот     |
| 5:      |      |
| 6:      |      |
| 7:      |      |
| 8:      |      DELETED |
| 9:      |      газ    |
|-----|
```

Рисунок 15 – Тестирование операции удаления существующего ключа из хэш-таблицы

На Рисунке 16 представлен результат тестирования операции удаления несуществующего ключа из хэш-таблицы.

```
Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
4
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/test.dat
Введите слово: qwerty
Запись с таким ключом не найдена.
```

Рисунок 16 – Тестирование операции удаления несуществующего ключа из хэш-таблицы

Тестирование операции поиска ключа («газ»), который размещен в хэш-таблице после удаленного ключа («заседание»), с одним значением хэша для этих ключей (8) представлен на Рисунке 17.

```

Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
3
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/test.dat
Введите слово для поиска: газ

-----
Время поиска: 0.0917 ms
-----

Слово 'газ' встретилось в тексте 1151 раз(-а).

Выберите действие:
1. Преобразование текстового файла в двоичный
2. Перевод двоичного файла в хэш-таблицу
3. Поиск записи в хэш-таблице и вывод из двоичного файла
4. Удаление записи из хэш-таблицы и двоичного файла
5. Вывод хэш-таблицы
6. Вывод бинарного файла
7. Тестирование модуля управления хэш-таблицей
8. Тестирование модуля управления бинарным файлом
9. Выйти
5
-----
| 0: |
| 1: |      проект |
| 2: |      воздух |
| 3: |   заместитель |
| 4: |      рот |
| 5: |
| 6: |
| 7: |
| 8: |   DELETED |
| 9: |      газ |
|-----|

```

Рисунок 17 – Тестирование операции поиска ключа, который размещен в хэш-таблице после удаленного ключа, с одним значением хэша для этих ключей

4.4 Сложность операций над хэш-таблицей

Три основных операции (вставка, поиск и удаление ключа) имеют константную сложность $O(1)$, не зависящую от количества элементов в хэш-таблице. По этой причине все операции даже над огромной хэш-таблицей будут осуществляться быстро. Чтобы это доказать был взят файл со 100.000 записей, их которого была составлена хэш-таблица. В этой таблице было проведено по несколько операций поиска в начале, середине и конце файла. Результаты проведённых операций – время выполнения каждого запроса – представлены на Рисунках 18 – 23.

```
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/data.dat
Введите слово для поиска: быть

-----
Время поиска: 0.0953 ms
-----

Слово 'быть' встретилось в тексте 67205 раз(-а).
```

Рисунок 18 – Поиск записи в начале файла

```
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/data.dat
Введите слово для поиска: вежливый

-----
Время поиска: 0.159 ms
-----

Слово 'вежливый' встретилось в тексте 48530 раз(-а).
```

Рисунок 19 – Поиск записи в начале файла

```
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/data.dat
Введите слово для поиска: бристоль

-----
Время поиска: 3.1199 ms
-----

Слово 'бристоль' встретилось в тексте 80470 раз(-а).
```

Рисунок 20 – Поиск записи в середине файла

```
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/data.dat
Введите слово для поиска: тромбон

-----
Время поиска: 3.8513 ms
-----

Слово 'тромбон' встретилось в тексте 6030 раз(-а).
```

Рисунок 21 – Поиск записи в середине файла

```
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/data.dat
Введите слово для поиска: публик

-----
Время поиска: 7.7233 ms
-----

Слово 'публик' встретилось в тексте 85176 раз(-а).
```

Рисунок 22 – Поиск записи в конце файла

```
Введите имя двоичного файла: D:/МИРЭА/СиАОД/2.3/files/data.dat
Введите слово для поиска: сухпай

-----
Время поиска: 8.6653 ms
-----

Слово 'сухпай' встретилось в тексте 16047 раз(-а).
```

Рисунок 23 – Поиск записи в конце файла

Важно отметить, что исходный размер таблицы – 5, а количество записей в файле – 100.000, поэтому таблица подвергается рехэшированию несколько раз. Финальный размер таблицы равен 163.840 элементам, то есть коэффициент нагрузки примерно равен 0.61, кроме того, в таблице уже достаточно много кластеров.

5 ВЫВОД

Тестирование операции поиска в хэш-таблице с 100.000 записей показало, что поиск в хэш-таблице происходит за одно и то же время, и не зависит от расположения записи в файле. Это доказывает, что операция поиска, а также удаления и вставки записи без рехэширования, имеет сложность $O(1)$. Такая сложность у алгоритмов операций будет сохраняться при правильно подобранной хэш-функции и вовремя проводимом рехэшировании, в противном случае сложность алгоритма может возрасти до линейной.