

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](#_Toc148679151)

[1.1 Условие задачи 4](#_Toc148679152)

[1.2 Требования по выполнению 4](#_Toc148679153)

[2 Ход решения 5](#_Toc148679154)

[2.1 Двоичный файл из записей фиксированного размера 5](#_Toc148679155)

[2.1.1 Структура записи файла 5](#_Toc148679156)

[2.1.2 Размер записи файла 5](#_Toc148679157)

[2.1.3 Прототипы операций по управления двоичным файлом 6](#_Toc148679158)

[2.2 Хеш-таблица 8](#_Toc148679159)

[2.2.1 Структура элемента таблицы 8](#_Toc148679160)

[2.2.2 Код элемента таблицы 8](#_Toc148679161)

[2.2.3 Описание алгоритмов операций 9](#_Toc148679162)

[2.2.4 Алгоритм поиска записи с заданным ключом в файле посредством хеш-таблицы 10](#_Toc148679163)

[3 Код приложения 11](#_Toc148679164)

[3.1 Модуль управления хэш-таблицей 11](#_Toc148679165)

[3.2 Модуль управления двоичным файлом 15](#_Toc148679166)

[3.3 Модуль управления двоичным файлом посредством хэш-таблицы 22](#_Toc148679167)

[3.4 Функция main 23](#_Toc148679168)

[4 Результаты тестирования 27](#_Toc148679169)

[4.1 Вычисление хэш-функции 27](#_Toc148679170)

[4.2 Тестирование модуля для управления хэш-таблицей 27](#_Toc148679171)

[4.3 Тестирование модуля для управления двоичным файлом с помощью хэш-таблицы 31](#_Toc148679172)

[4.4 Сложность операций над хэш-таблицей 37](#_Toc148679173)

[5 ВЫВОД 40](#_Toc148679174)

# Постановка задачи

## 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: слово, количество вхождений в текст |

## 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.

# Ход решения

## Двоичный файл из записей фиксированного размера

### Структура записи файла

Структура записи файла из кода представлена в Листинге 1.

*Листинг 1 – Структура записи*

struct word {

char name[30];

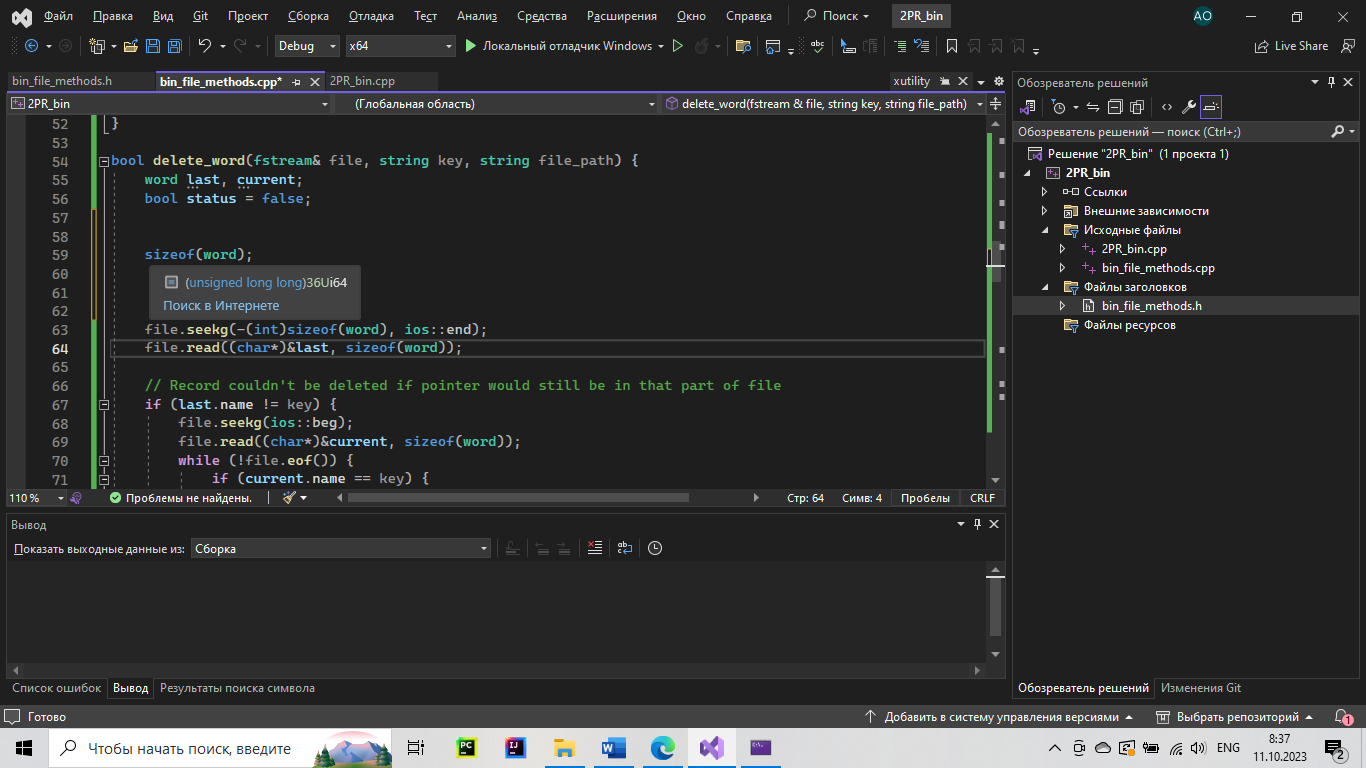
unsigned int count;

};

### 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**

### Прототипы операций по управления 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.

## Хеш-таблица

### 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 |

### Код элемента таблицы

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

*Листинг 3 – Структура элемента и таблицы*

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();

};

### 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.

### 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.

# Код приложения

## 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;

}

}

}

}

## 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 вхоождений слов успешно обновлено в бинарном файле. " << endl;

fout.close();

}

else {

cout << "Файл не найден или не существует." << endl;

}

break;

}

default: {

return 0;

}

}

}

}

## 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;

}

## 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(-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::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;

}

}

}

}

# Результаты тестирования

## Вычисление хэш-функции

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

где mod – функция, возвращающая первый аргумент по модулю второго,

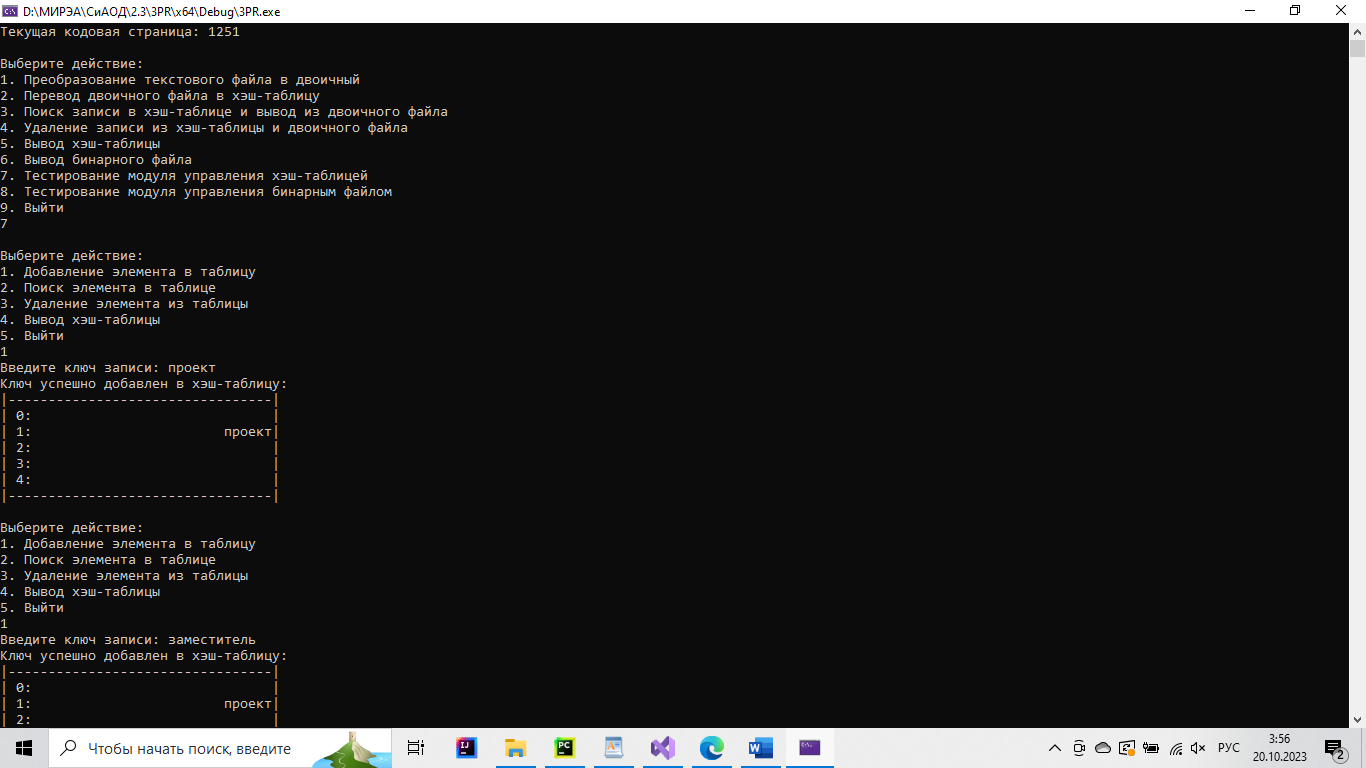
n – текущий размер хэш-таблицы,

ascii – функция, возвращающая ascii-код символа-аргумента,

key – ключ, для которого вычисляется значение хэш-функции.

## Тестирование модуля для управления хэш-таблицей

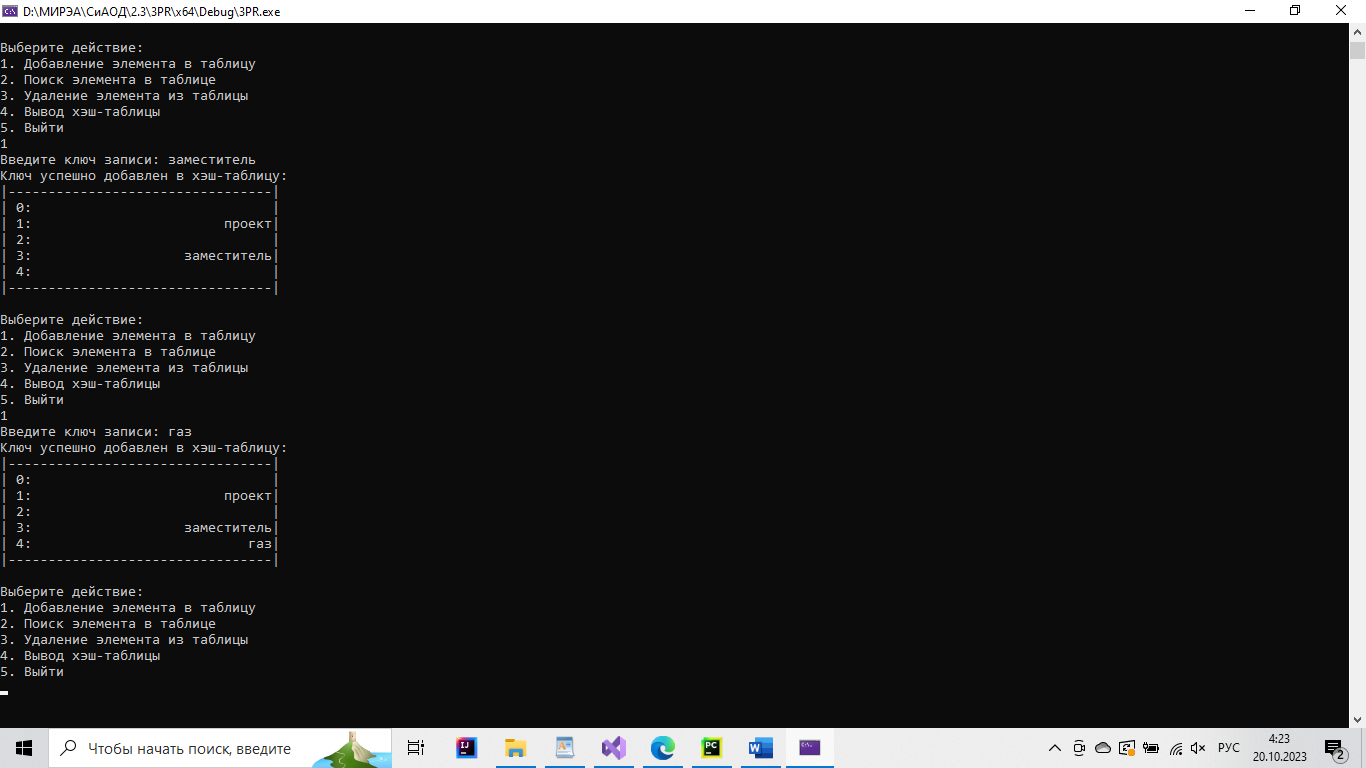
На Рисунке 2 представлен результат тестирования операции вставки ключа в хэш-таблицу без коллизии. Значение хэш-функции вычисляется следующим образом: (239 \* 1 + 240 \* 2 + 238 \* 3 + 229 \* 4 + 234 \* 5 + 242 \* 6) % 5 = 1. Вся таблица свободна, следовательно, вставляем элемент с переданным ключом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



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

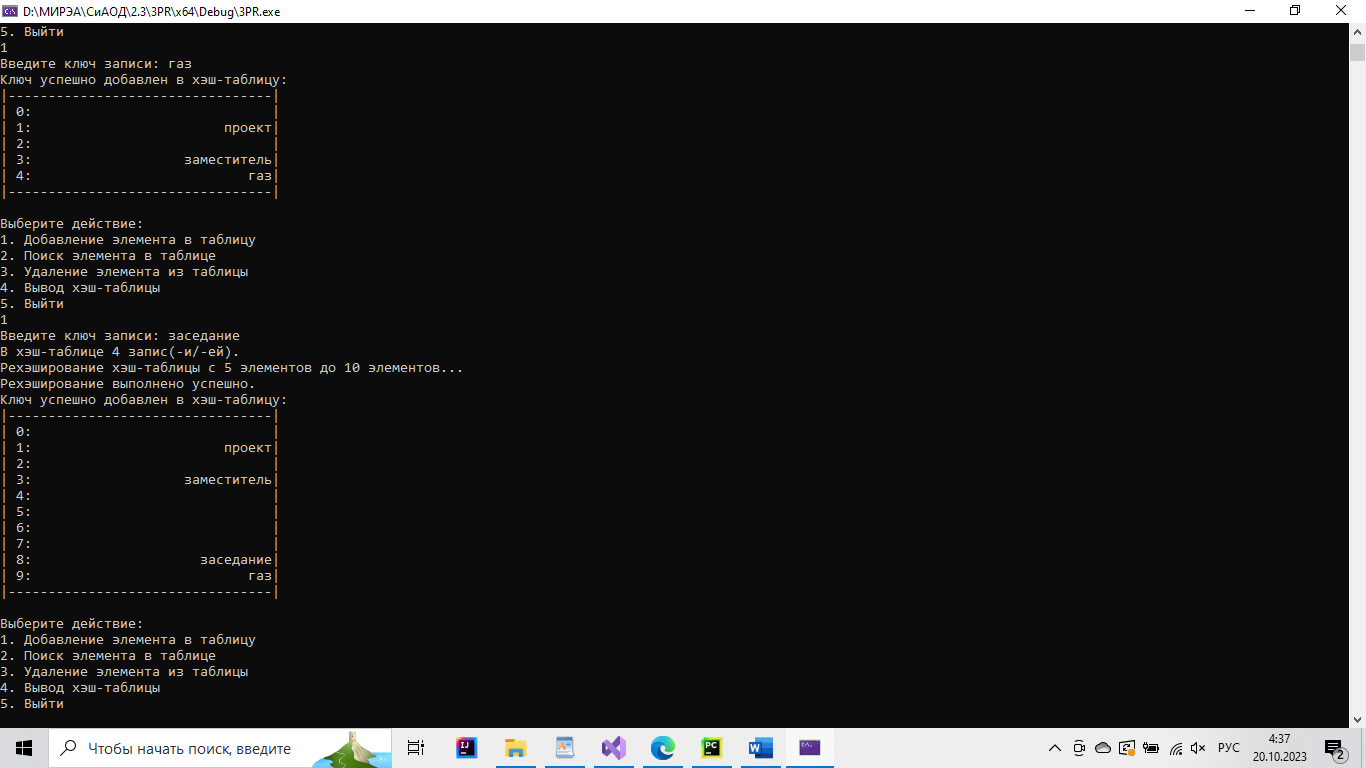
После добавления в хэш-таблицу ключа «проект», был добавлен ключ «заместитель», хэш которого равен: (231 \* 1 + 224 \* 2 + 236 \* 3 + 229 \* 4 + 241 \* 5 + 242 \* 6 + 232 \* 7 + 242 \* 8 + 229 \* 9 + 235 \* 10 + 252 \* 11) % 5 = 3. Данный ключ был также добавлен без коллизии, так как полученный адрес 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.



**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.

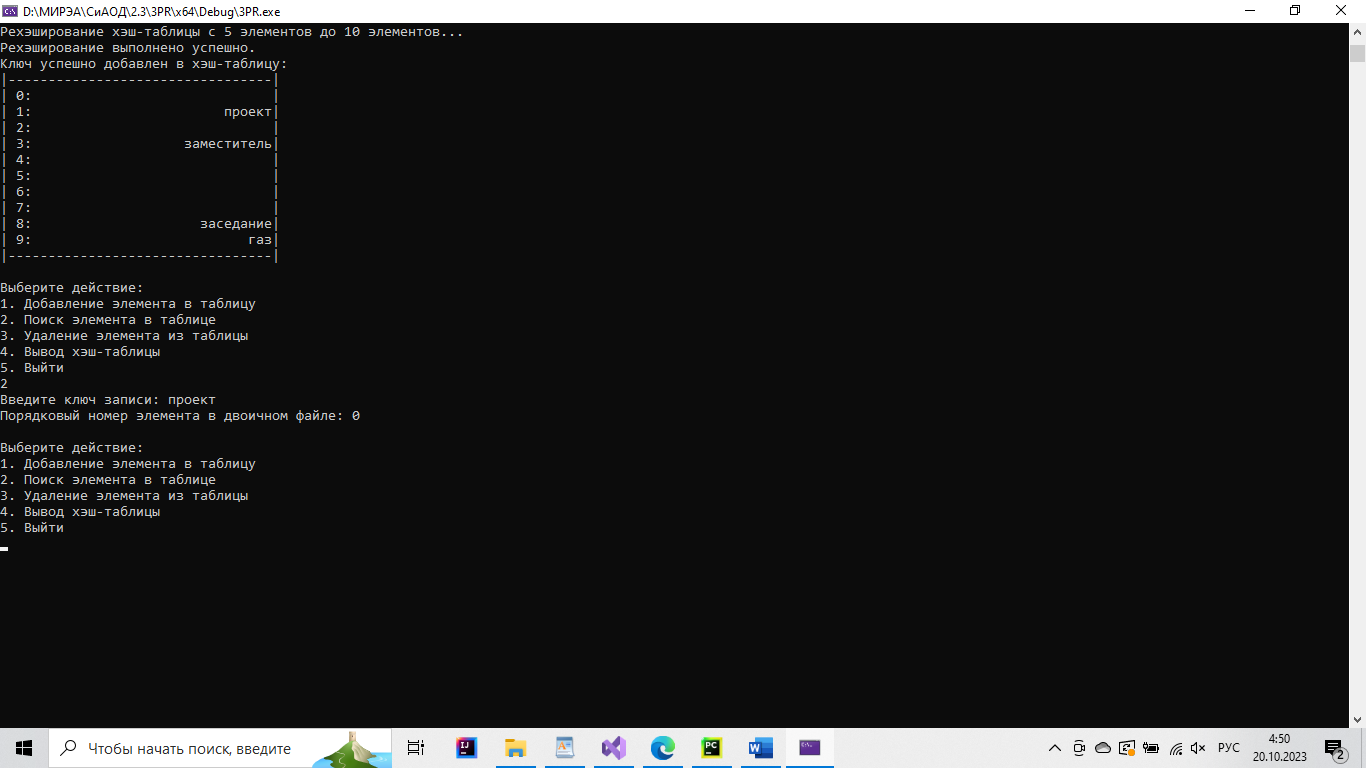


**Рисунок 4 – Тестирование операции вставки ключа в хэш-таблицу с последующим рехэшированием**

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:

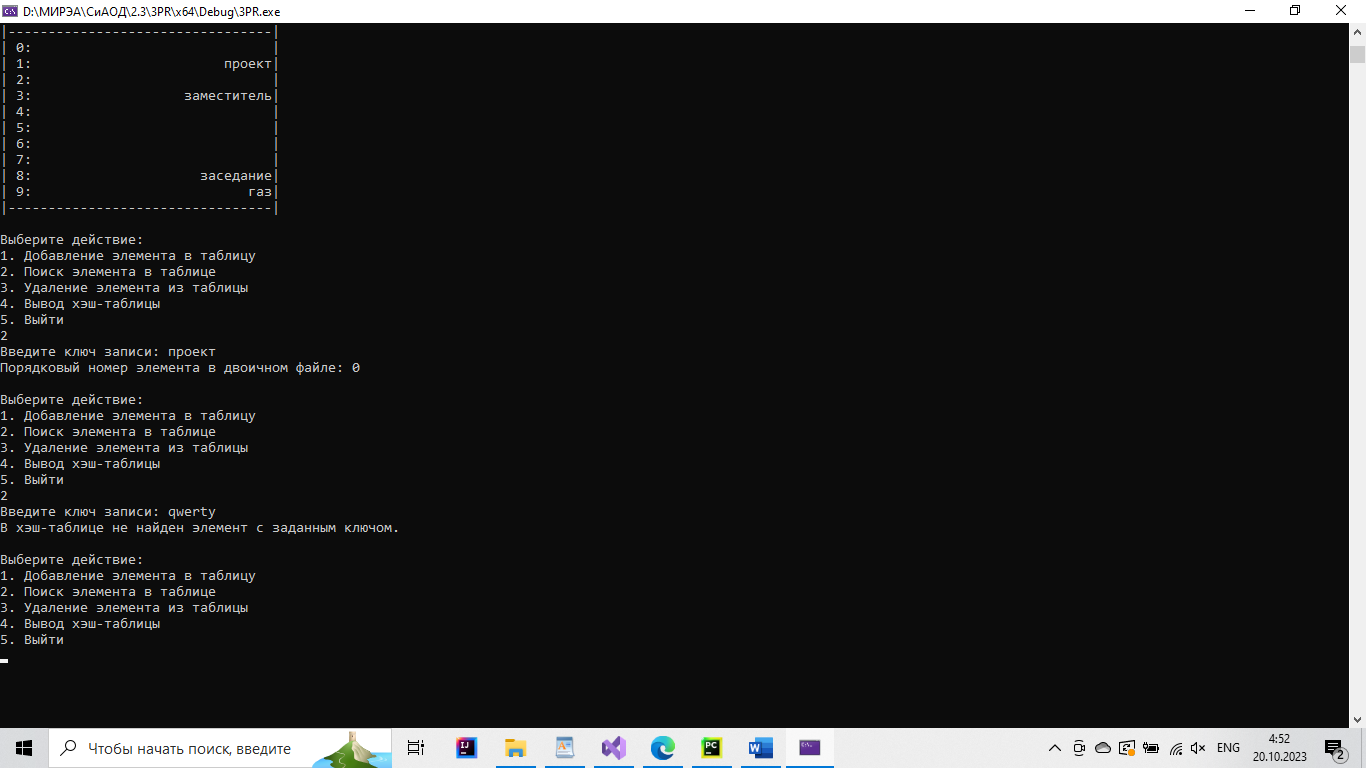
* проект: (239 \* 1 + 240 \* 2 + 238 \* 3 + 229 \* 4 + 234 \* 5 + 242 \* 6) % 10 =1;
* заместитель: (231 \* 1 + 224 \* 2 + 236 \* 3 + 229 \* 4 + 241 \* 5 + 242 \* 6 + 232 \* 7 + 242 \* 8 + 229 \* 9 + 235 \* 10 + 252 \* 11) % 10 = 3;
* газ: (227 \* 1 + 224 \* 2 + 231 \* 3) % 10 = 8;
* заседание: (231 \* 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.



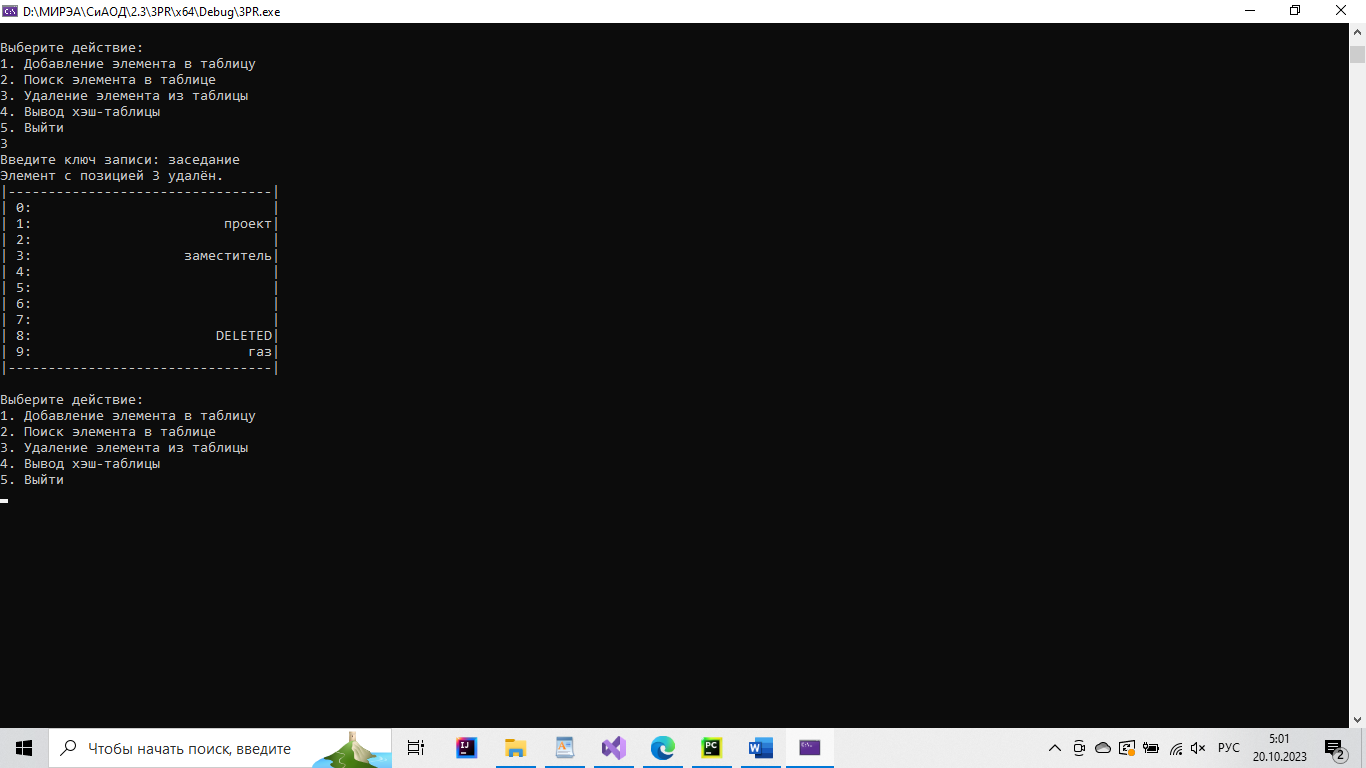
**Рисунок 5 – Тестирование операции поиска существующего ключа в хэш-таблице**

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



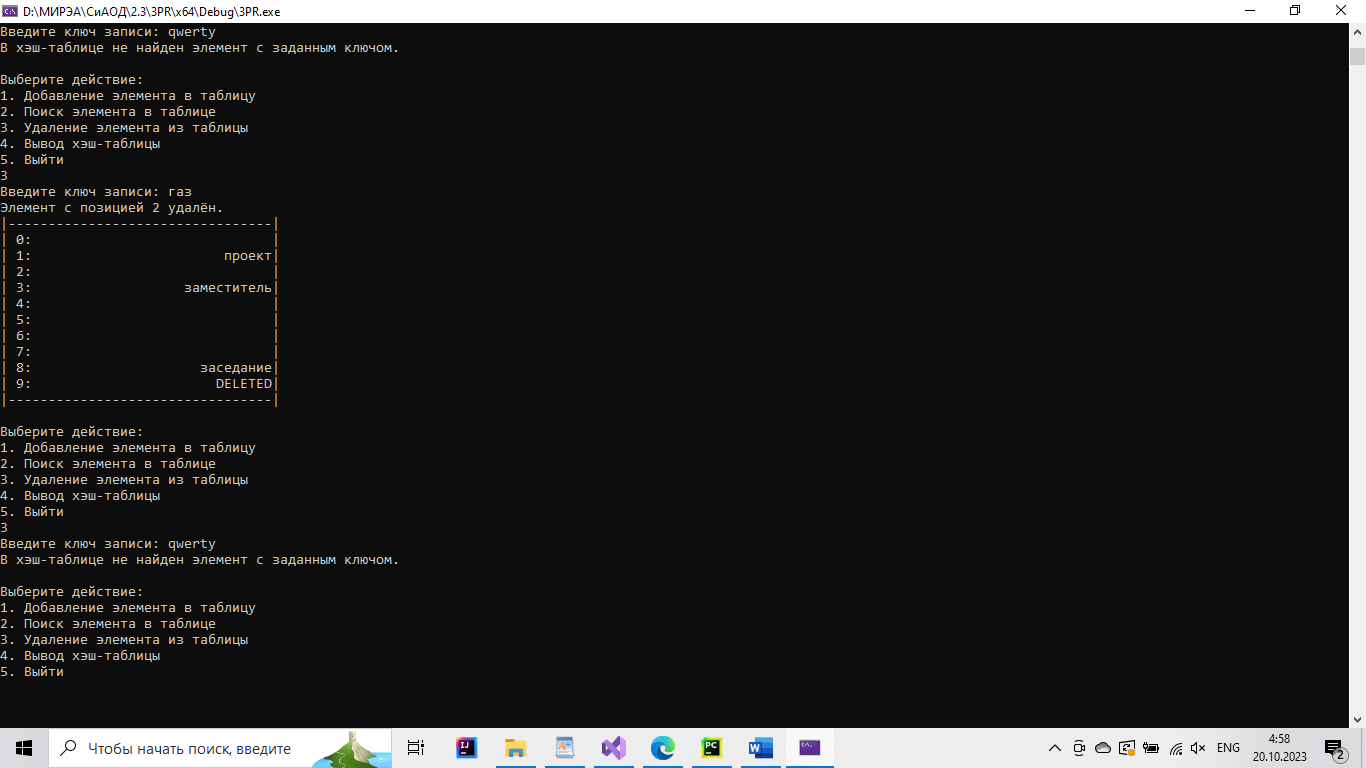
**Рисунок 6 – Тестирование операции поиска несуществующего ключа в хэш-таблице**

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.



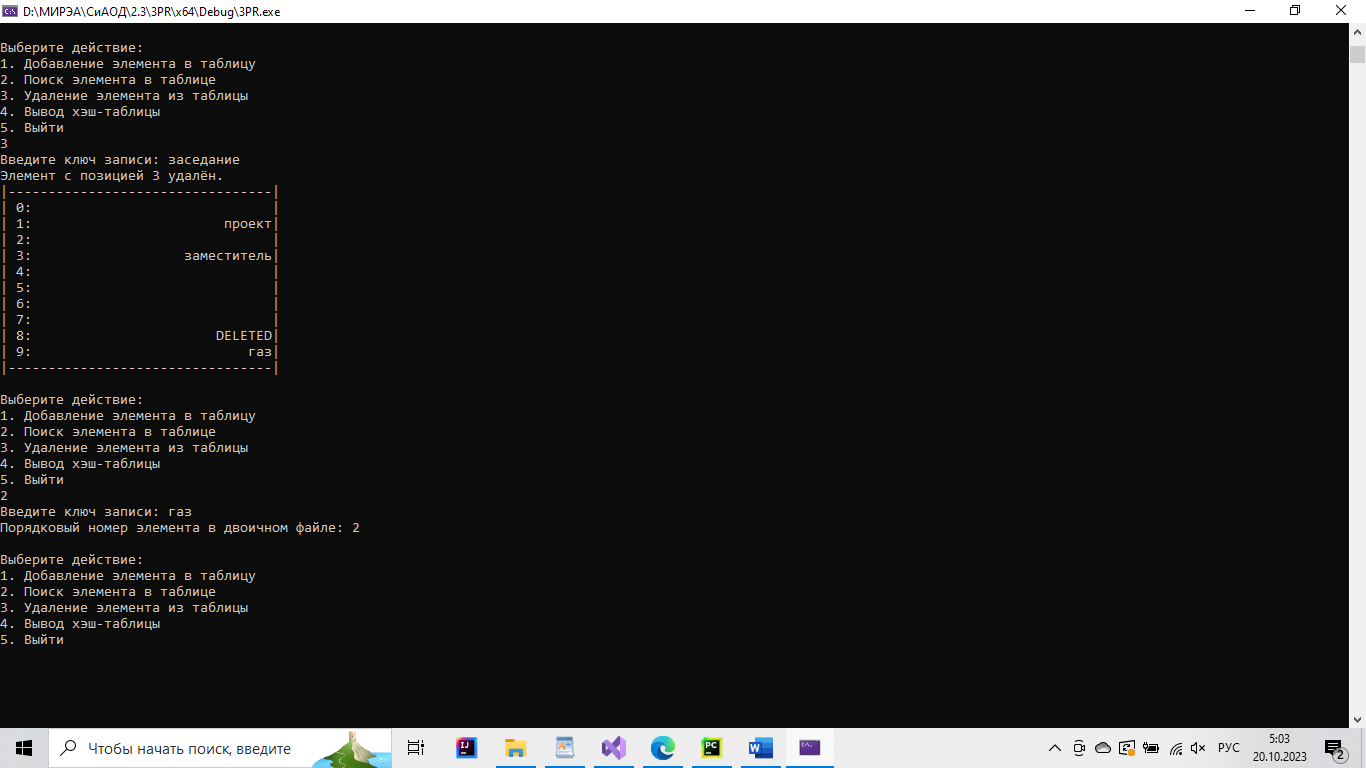
**Рисунок 7 – Тестирование операции удаления существующего ключа из хэш-таблицы**

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



**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.



**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.

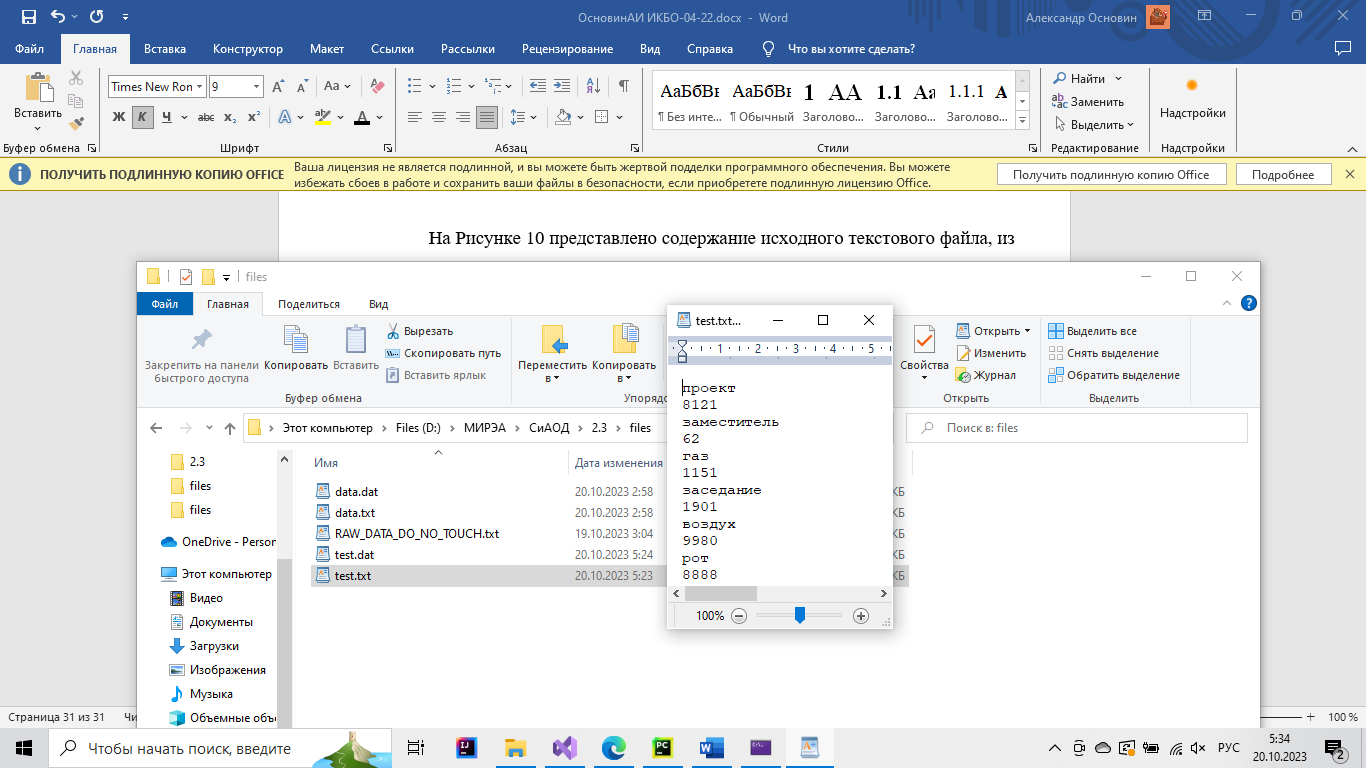
## Тестирование модуля для управления двоичным файлом с помощью хэш-таблицы

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

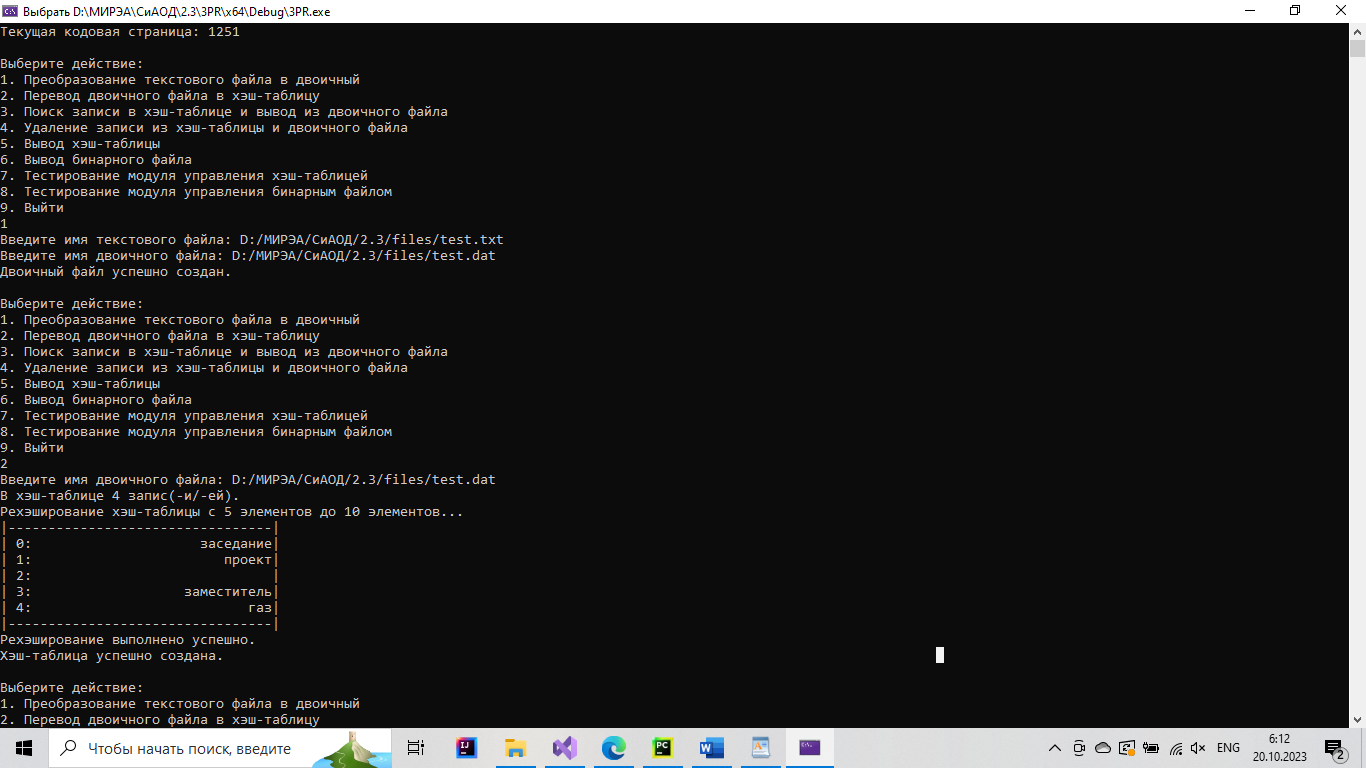
На Рисунке 10 представлено содержание исходного текстового файла, из которого с помощью функции модуля для управления двоичными файлами, будет создан двоичный файл.



**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 закончится, и добавлять элементы будет некуда.



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

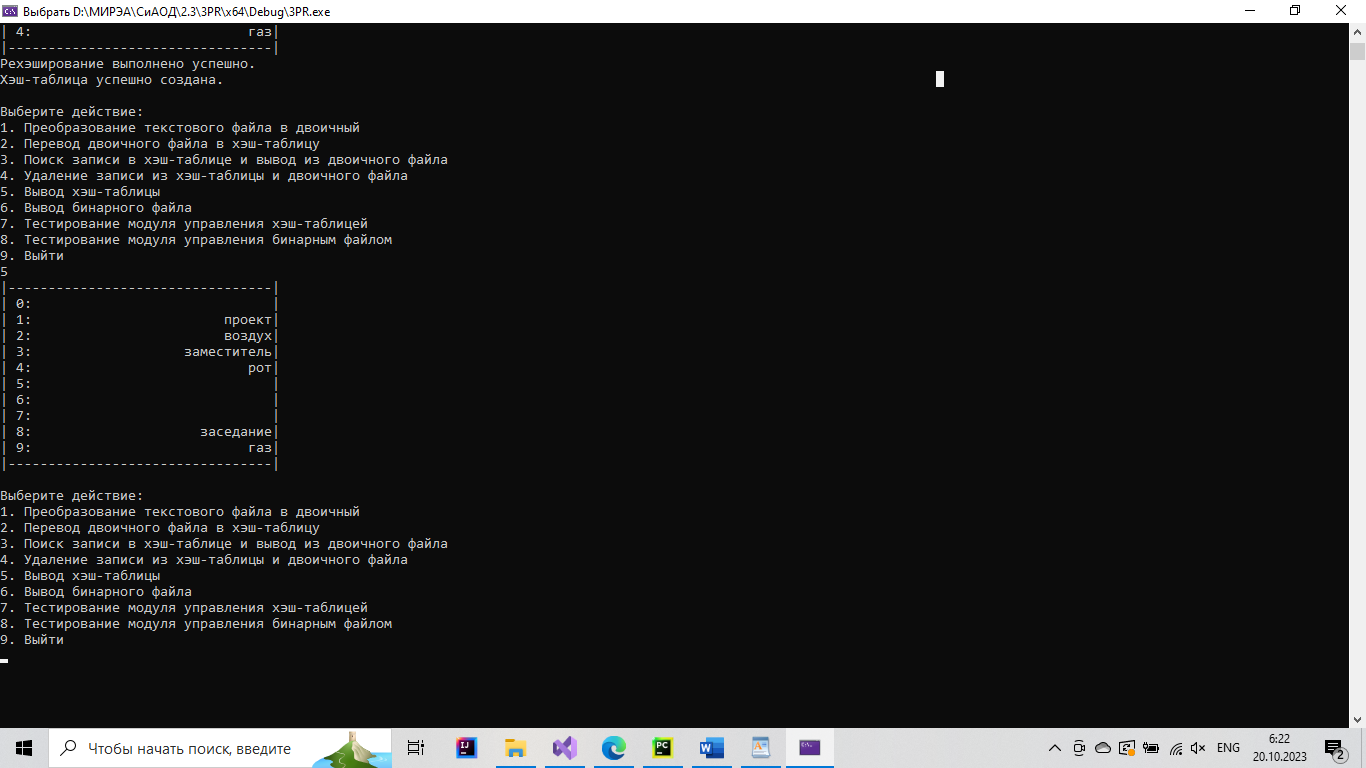
После этого произойдет рехэширование и хэши добавленных в хэш-таблицу записей будут рассчитаны снова (хэши будут вычисляться по тому порядку, в котором записи находятся в таблице), следующим образом:

* 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:

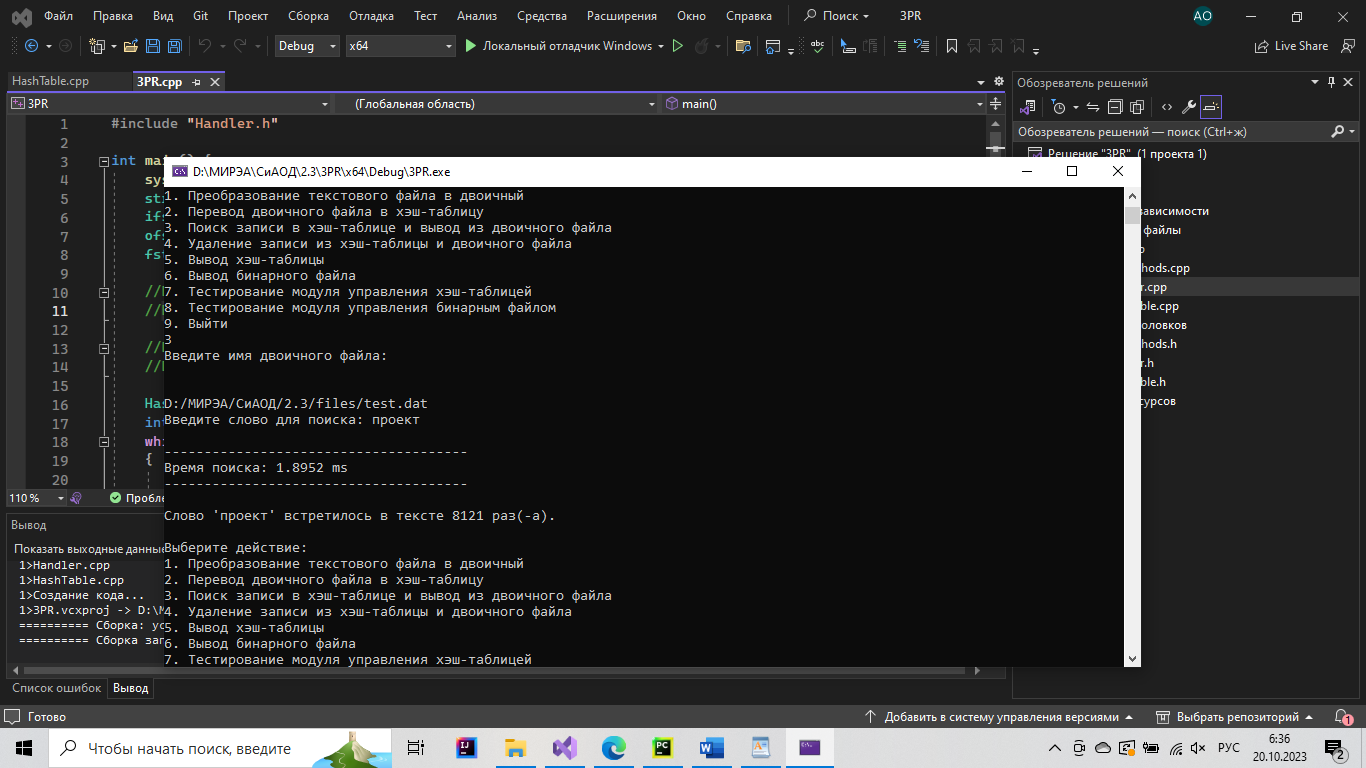
* воздух: (226 \* 1 + 238 \* 2 + 231 \* 3 + 228 \* 4 + 243 \* 5 + 245 \* 6) % 10 = 2, добавление в хэш-таблицу без коллизии;
* рот: (240 \* 1 + 238 \* 2 + 242 \* 3) % 10 = 2, возникновение коллизии, разрешение путём смещения на 1, повторное возникновение коллизии, смещение на 1, добавление в хэш-таблицу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

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



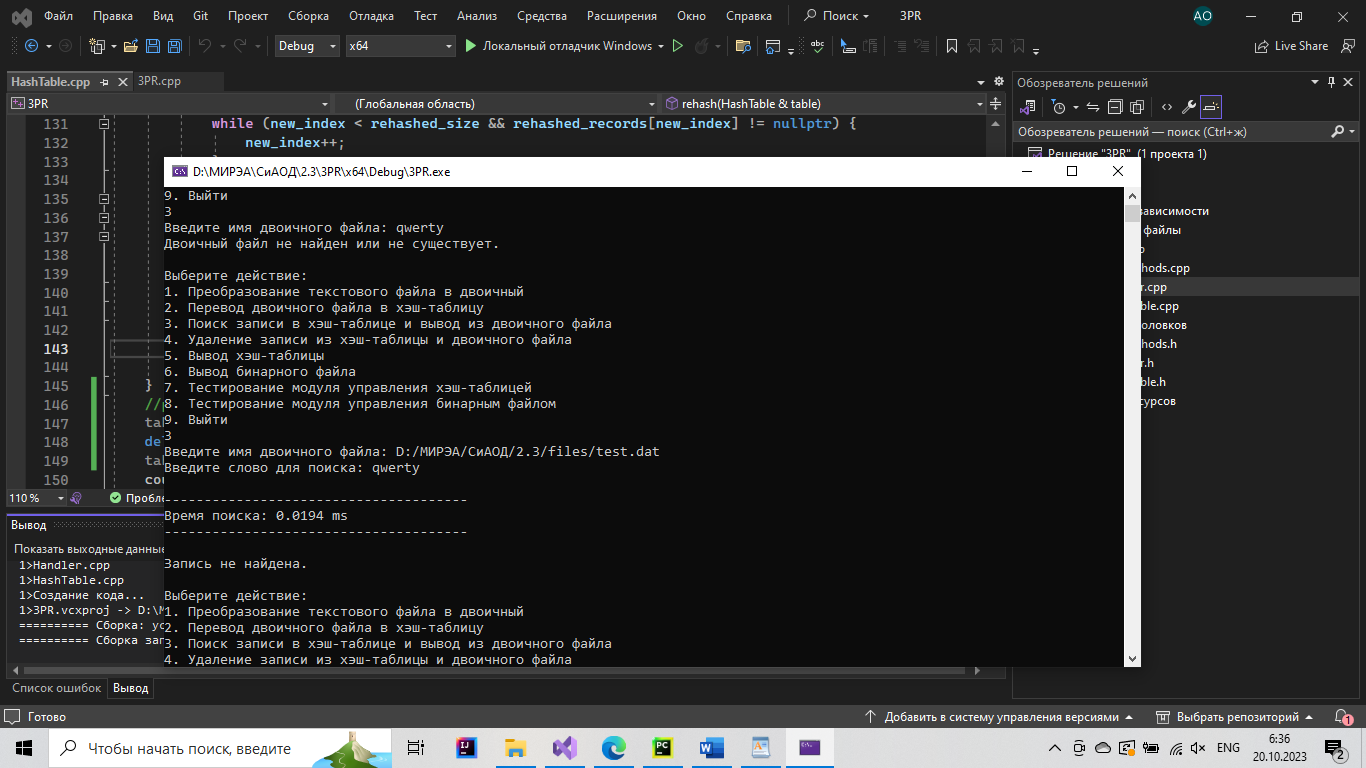
**Рисунок 12 – Хэш-таблица, построенная по содержанию двоичного файла**

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



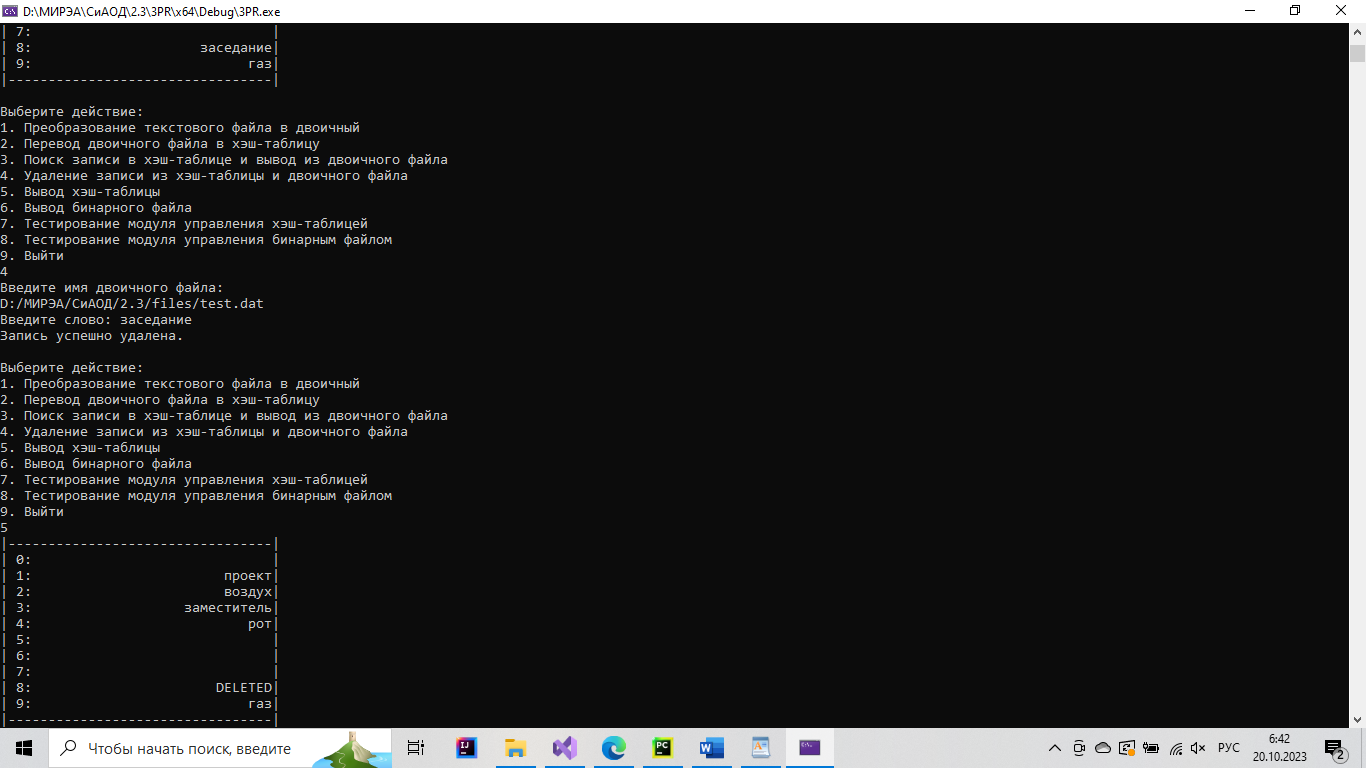
**Рисунок 13 – Тестирование операции поиска существующего ключа в хэш-таблице**

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



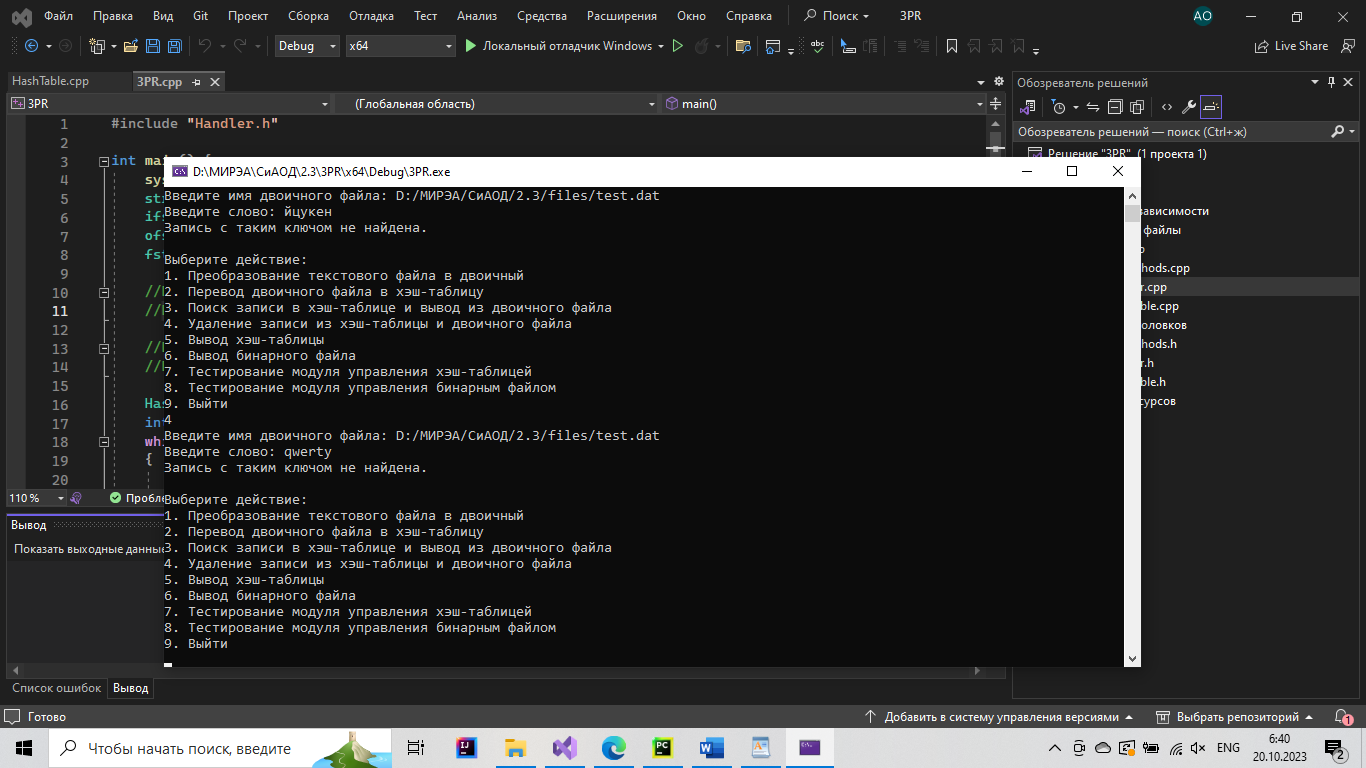
**Рисунок 14 – Тестирование операции поиска несуществующего ключа в хэш-таблице**

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.



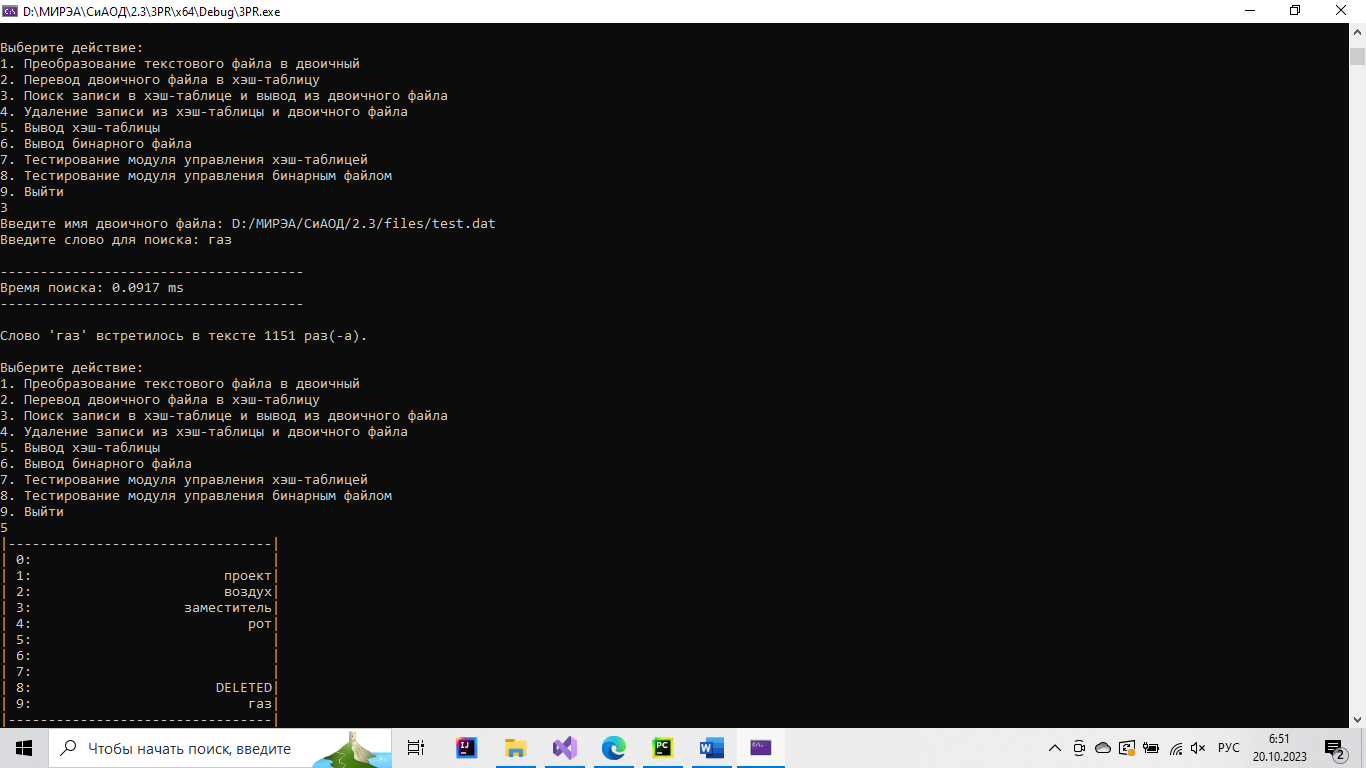
**Рисунок 15 – Тестирование операции удаления существующего ключа из хэш-таблицы**

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



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

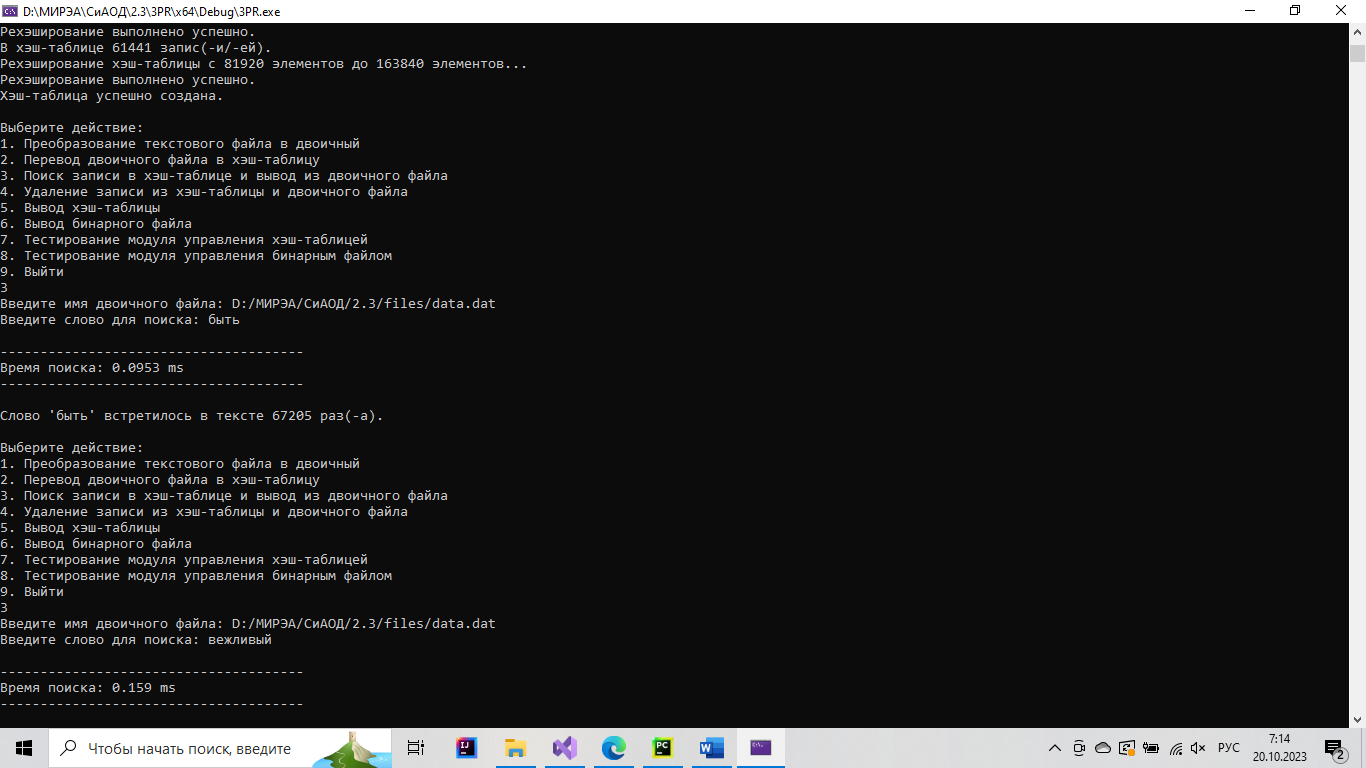
Тестирование операции поиска ключа («газ»), который размещен в хэш-таблице после удаленного ключа («заседание»), с одним значением хэша для этих ключей (8) представлен на Рисунке 17.



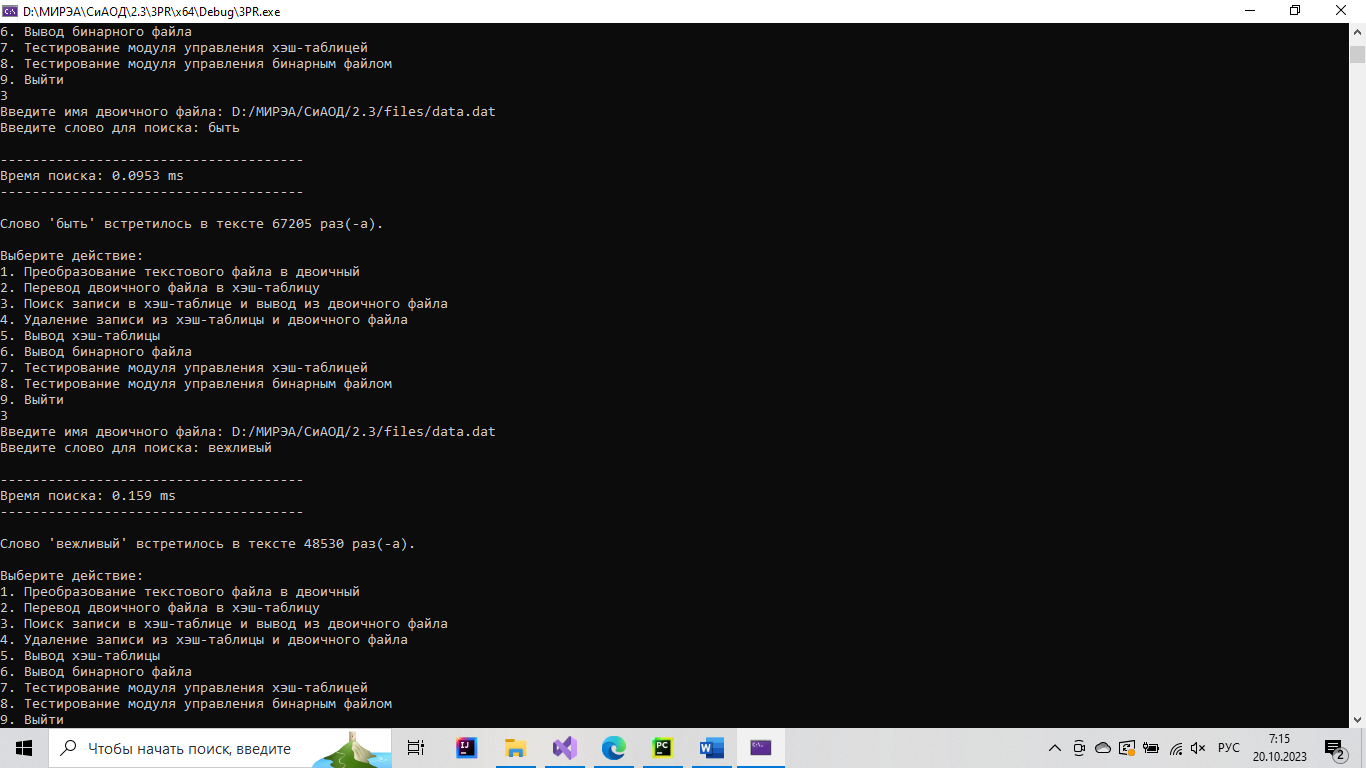
**Рисунок 17 – Тестирование операции поиска ключа, который размещен в хэш-таблице после удаленного ключа, с одним значением хэша для этих ключей**

## Сложность операций над хэш-таблицей

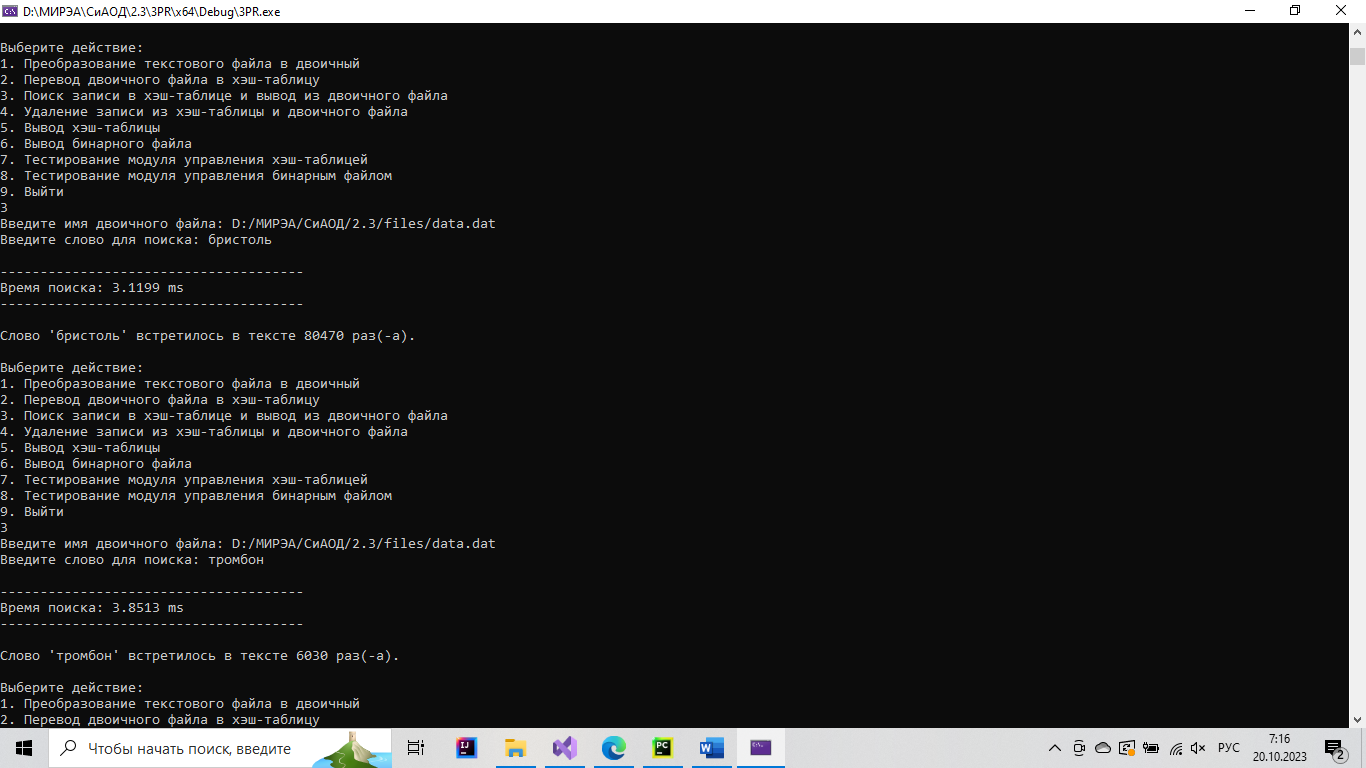
Три основных операции (вставка, поиск и удаление ключа) имеют константную сложность O(1), не зависящую от количества элементов в хэш-таблице. По этой причине все операции даже над огромной хэш-таблицей будут осуществляться быстро. Чтобы это доказать был взят файл со 100.000 записей, их которого была 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.



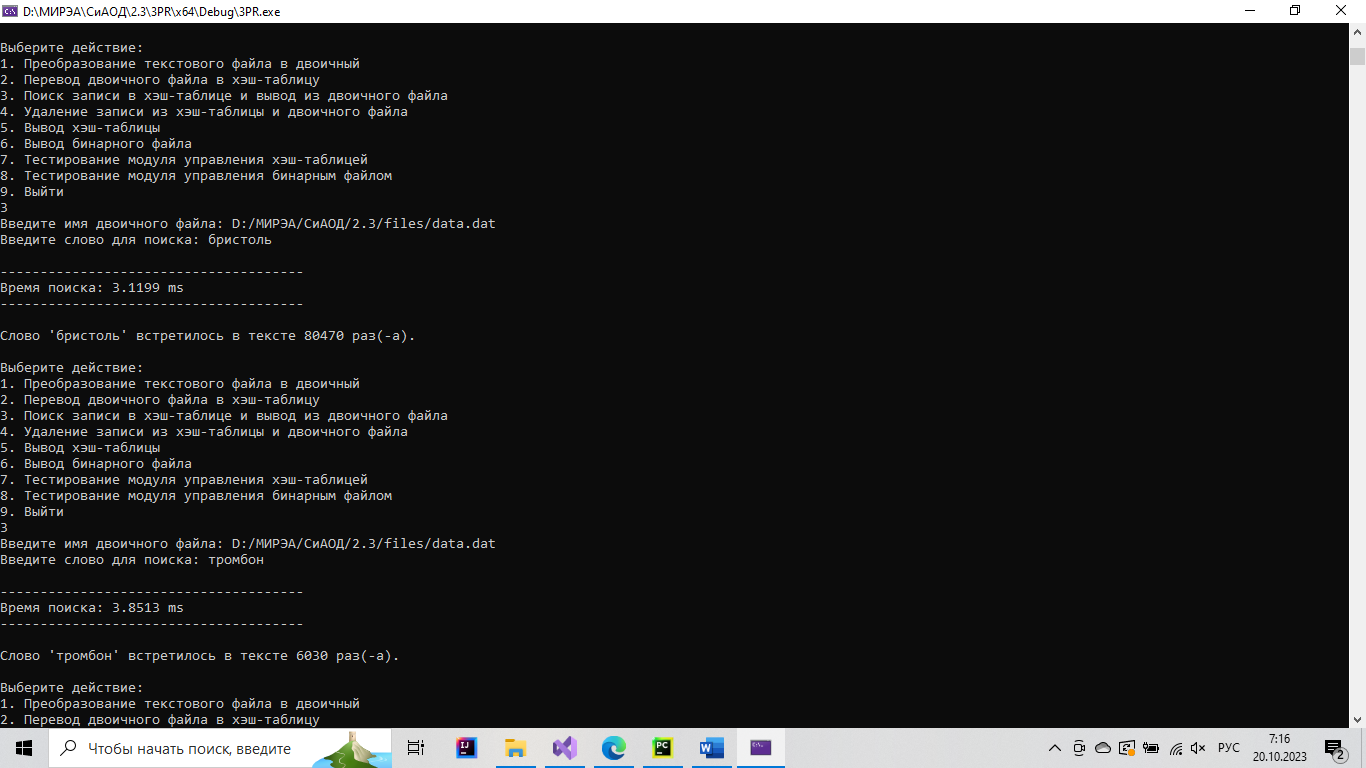
**Рисунок 18 – Поиск записи в начале файла**



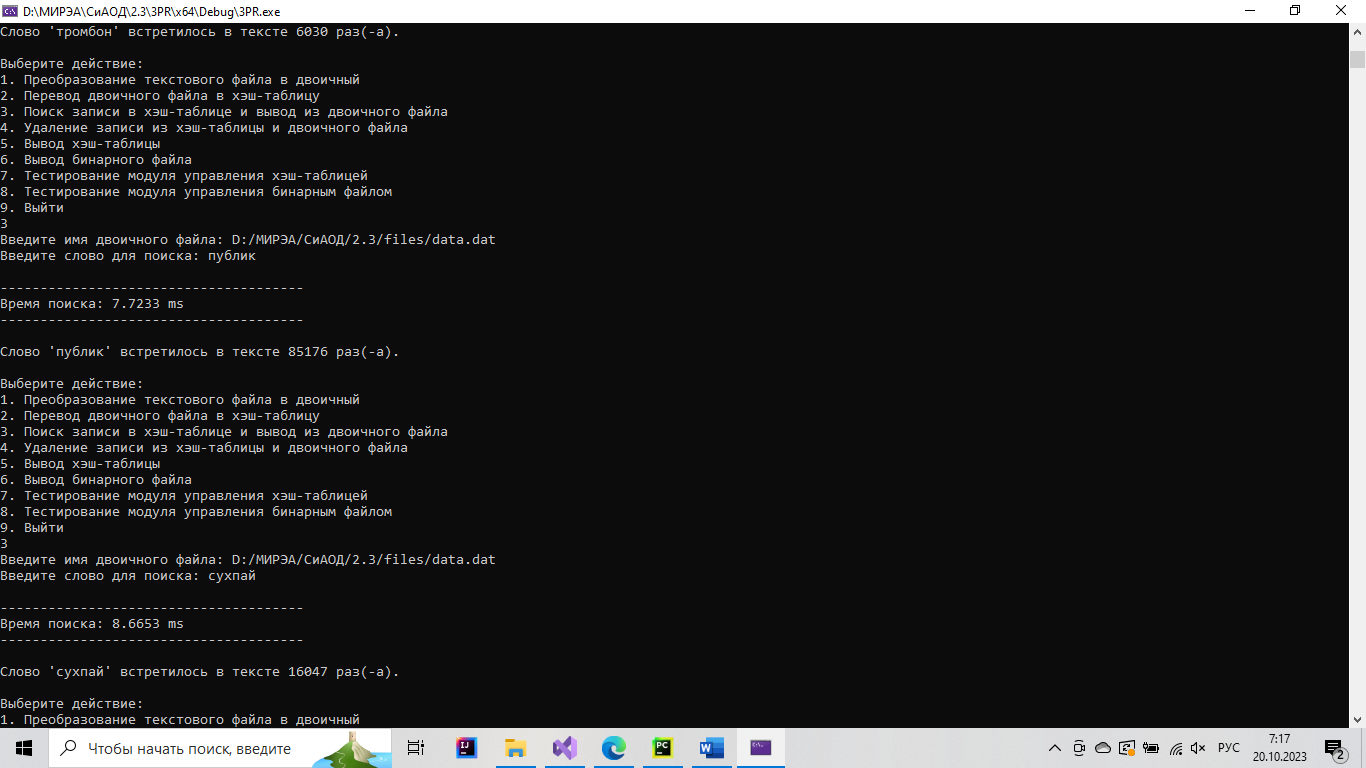
**Рисунок 19 – Поиск записи в начале файла**



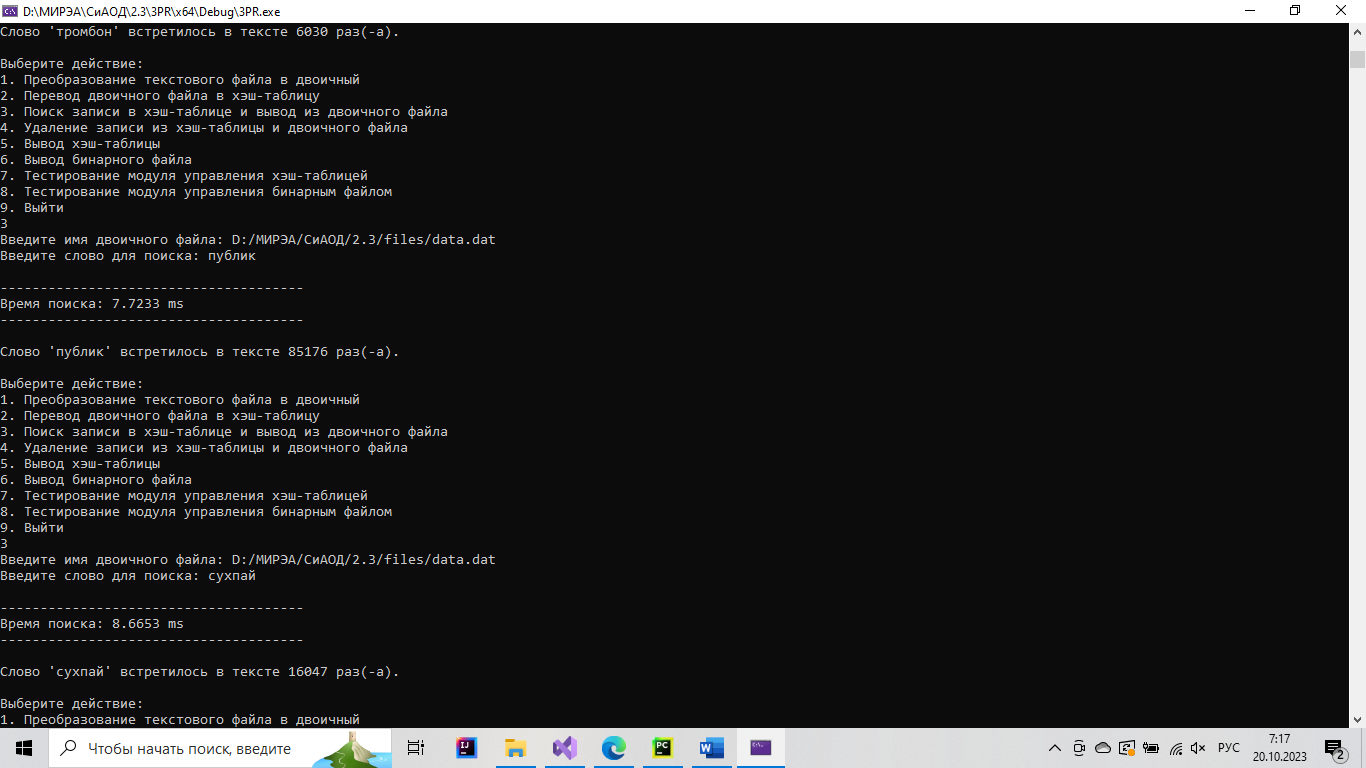
**Рисунок 20 – Поиск записи в середине файла**



**Рисунок 21 – Поиск записи в середине файла**



**Рисунок 22 – Поиск записи в конце файла**



**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.

# 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.