МИНОБРНАУКИ РОСИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания №5.2
Тема:
Алгоритмы поиска в таблице при работе с данными из файла
Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Васильев Б.А.
Группа: ИКБО-20-23

Москва 2024

СОДЕРЖАНИЕ

ЗАДАНИЕ 1 Создать двоичный файл из записей 3
Формулировка задачи 3
Описание подхода к решению 3
Коды программы 4
Результаты тестирования 9
ЗАДАНИЕ 2 Поиск в файле с применением линейного поиска 10
Формулировка задачи 10
Алгоритм 10
Код программы 11
Результаты тестирования 12
ЗАДАНИЕ 3 Поиск записи в файле с применением дополнительной структуры данных, сформированной в оперативной памяти. 14
Формулировка задачи 14
Алгоритм 15
Код программы 16
Результаты тестирования 19
ВЫВОД 22
СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 22

ЦЕЛЬ РАБОТЫ

Получить практический опыт по применению алгоритмов поиска в таблице данных

ЗАДАНИЕ 1

Создать двоичный файл из записей

Формулировка задачи

Создать двоичный файл из записей, структура записи определена вариантом (рис. 1). Поле ключа записи в задании варианта подчеркнуто. Заполнить файл данными, используя для поля ключа датчик случайных чисел. Ключи записей в файле уникальны.

№	Алгоритм поиска	Структура записи файла (ключ – подчеркнутое поле)
1	Бинарный однородный без использования дополнительной таблицы	Читательский абонемент: номер читательского билета - целое пятизначное число, ФИО, Адрес

Рисунок 1 – Индивидуальный вариант задачи

Описание подхода к решению

Создание файла из нужного количества записей можно разделить на два этапа: создание текстового файла, конвертация текстового файла в двоичный.

Структура записи файла определена вариантом (рис. 1). Размер двоичного представления записи в байтах состоит из суммы отведённого количества байтов на каждое из трёх полей. В данной работе на поле номера читательского билета было выделено 5 байт, на поля ФИО и адрес — по 55 байт. Итоговая длина записи равняется 115 байтам. Таким образом, для доступа к i-ой записи, достаточно перейти к байту с номером 115*i.

Коды программы

Реализуем алгоритм на языке программирования C++ (рис. 2-7)

SIAOD > 5_2 > cyrillic_fix > C header_5_2.h > ...

```
#ifndef HEADER_5_2_H
#define HEADER_5_2_H

#include <iostream>
#include <algorithm>
#include <vector>
#include <array>
#include <clocale>
#include <string>
#include <fstream>
#include <set>
#include <random>
#include <sstream>
#include <string.h>
#include <chrono>
#include <fstream>

const std::string MATCHES_FILE_NAME = "matches.txt";
const std::string FILES_FOLDER = "files/";
const std::string SPECIAL_LINE = "99999,Васильев Борис Александрович,пр. Вернадского д. 78";
const char DELIMITER = ',';

// Field lengths
const int ID_SIZE = 5;
const int NAME_SIZE = 55;
const int ADDRESS_SIZE = 55;


#endif
```

Рисунок 2 – Заголовочный файл общий для всех трёх заданий

```cpp
#include "header_5_2.h"

using namespace std;

string last_names[] = {"Иванов", "Петров", "Сидоров", "Кузнецов", "Смирнов",
                       "Попов", "Козлов", "Волков", "Морозов", "Лебедев",
                       "Новиков", "Орлов", "Захаров", "Фёдоров", "Григорьев"};

string first_names[] = {"Александр", "Дмитрий", "Максим", "Алексей", "Иван",
                        "Михаил", "Юрий", "Никита", "Сергей", "Владимир", "Егор",
                        "Павел", "Роман", "Вячеслав", "Олег"};

string father_names[] = {"Александрович", "Дмитриевич", "Максимович", "Алексеевич",
                         "Иванович", "Михайлович", "Юрьевич", "Никитич", "Сергеевич",
                         "Владимирович", "Егорович", "Павлович", "Романович", "Вячеславович",
                         "Олегович"};

string street_names[] = {
        "ул. Ленина", "ул. Сталинская", "ул. Горького", "ул. Пушкина",
        "ул. Чехова", "ул. Тверская", "ул. Маяковского", "ул. Рязанская",
        "пр. Невский", "ул. Арбат", "ул. Суворовская", "пр. Кутузовский",
        "ул. Нагатинская", "ул. Краснопресненская", "пр. Горького"
    };

size_t last_names_size = sizeof(last_names) / sizeof(last_names[0]);
size_t first_names_size = sizeof(first_names) / sizeof(first_names[0]);
size_t father_names_size = sizeof(father_names) / sizeof(father_names[0]);
size_t street_names_size = sizeof(street_names) / sizeof(street_names[0]);

// Padding fields with necessary spaces up to the needed length
void pad_string(const string& str, char* buffer, int size)
{
    memset(buffer, ' ', size);
    strncpy(buffer, str.c_str(), size);
}
```

Рисунок 3 – Код к заданию 1 (часть 1)

```
int main()
{
    setlocale(LC_ALL, "ru_RU.UTF-8");

    // setting up random numbers generator
    random_device rd;
    mt19937 eng(rd());

    // define the range for IDs
    uniform_int_distribution<> distr(10000, 99998);

    // opening file to write to
    string output_file_name;
    cout << "Enter output file name (without extension): ";
    cin >> output_file_name;
    output_file_name = FILES_FOLDER + output_file_name;
    string output_file_name_txt = output_file_name + ".txt";
    ofstream output_file(output_file_name_txt);

    if (!output_file.is_open())
    {
        cout << "Unable to open txt file to write.\n";
        return 1;
    }

    // deciding how many entries will be in a file
    int number_of_lines;
    cout << "Enter number of lines to generate: ";
    cin >> number_of_lines;

    set<int> unique_numbers;

    string line;
```

Рисунок 4 – Код к заданию 1 (часть 2)

```
73      for (int i = 0; i < number_of_lines; i++)
74      {
75
76          // making sure generated number wasn't used previously
77          int random_number;
78          do
79          {
80              random_number = distr(eng);
81              unique_numbers.insert(random_number);
82          } while (unique_numbers.size() < i + 1);
83
84          // writing the entry
85          line = "";
86          line += to_string(random_number); // id
87
88          line += DELIMITER;
89
90          line += last_names[distr(eng) % last_names_size];
91          line += " ";
92          line += first_names[distr(eng) % first_names_size];
93          line += " ";
94          line += father_names[distr(eng) % father_names_size];
95
96          line += DELIMITER;
97
98          line += street_names[distr(eng) % street_names_size];
99          line += " д. ";
100         line += to_string(distr(eng) % 120);
101
102         output_file << line << endl;
103     }
104     cout << "Successfully written " << number_of_lines << " lines to " << output_file_name_txt << endl;
105     // putting special entry at the end
106     output_file << SPECIAL_LINE << endl;
107     cout << "Sucessfully added special line as the last entry" << endl;
108
109     output_file.close();
```

Рисунок 5 – Код к заданию 1 (часть 3)

```
112     // opening .txt file to read and .bin file to write
113     ifstream txtfile(output_file_name_txt);
114     ofstream binfile(output_file_name + ".bin", ios::binary);
115 
116     if (!txtfile.is_open() || !binfile.is_open())
117     {
118         cout << "Error opening files to convert txt to bin.\n";
119         return 1;
120     }
121 
122     // converting text fields to binary
123     while (getline(txtfile, line))
124     {
125         istringstream iss(line);
126         string id_str, name, address;
127 
128         getline(iss, id_str, DELIMITER);
129         getline(iss, name, DELIMITER);
130         getline(iss, address);
131 
132         char id_buffer[ID_SIZE];
133         char name_buffer[NAME_SIZE];
134         char address_buffer[ADDRESS_SIZE];
135 
136         pad_string(id_str, id_buffer, ID_SIZE);
137         pad_string(name, name_buffer, NAME_SIZE);
138         pad_string(address, address_buffer, ADDRESS_SIZE);
139 
140         binfile.write(id_buffer, ID_SIZE);
141         binfile.write(name_buffer, NAME_SIZE);
142         binfile.write(address_buffer, ADDRESS_SIZE);
143     }
144     auto end_time = chrono::high_resolution_clock::now();
145     auto time_delta = chrono::duration_cast<chrono::milliseconds>(end_time - start_time);
146     cout << "Completion time in milliseconds: " << time_delta.count() << endl;
147 }
```

Рисунок 6 – Код к заданию 1 (часть 4)

Результаты тестирования

Выполним тестирование программы для создания файла из 100 записей (рис. 7-9).

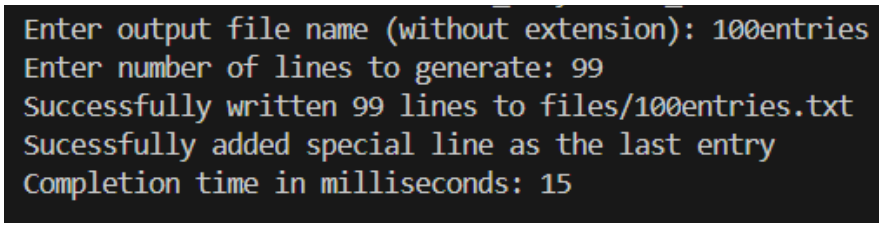


Рисунок 7 – Вывод программы к заданию 1 в консоль

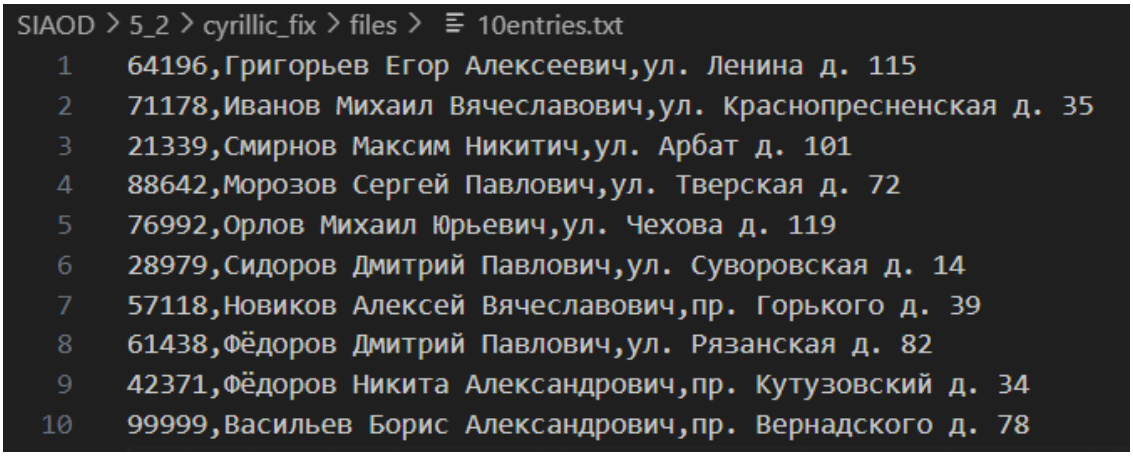


Рисунок 8 – Созданный текстовый файл

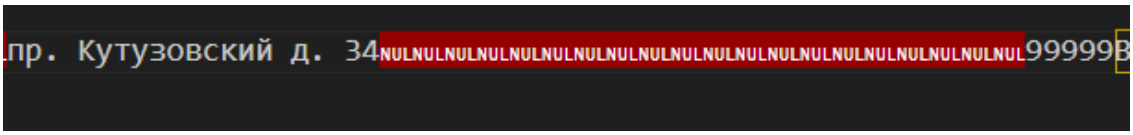


Рисунок 9 – Пример последовательности символов из двоичного файла, просмотренных в кодировке UTF-8

Тестирование показало, что программа работает корректно.

ЗАДАНИЕ 2

Поиск в файле с применением линейного поиска

Формулировка задачи

Разработать программу поиска записи по ключу в бинарном файле с применением алгоритма линейного поиска. Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей. Составить таблицу с указанием результатов замера времени

Алгоритм

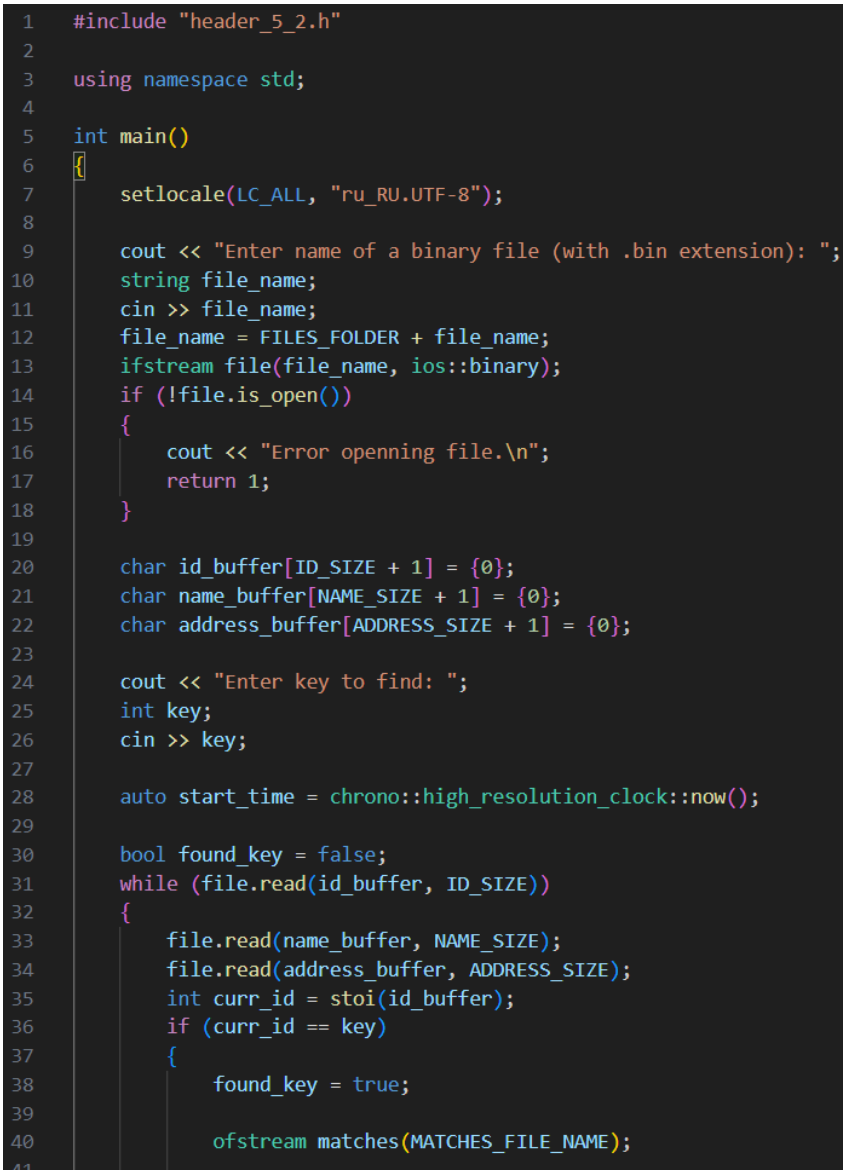
Приведём алгоритм линейного поиска, написанный на псевдокоде (рис. 10).

```
LINEAR_SEARCH
i ← 1 // инициализация параметра цикла
While (i<n) And (a[i]<>x)
Do i ← i + 1
If (i = n) Then <Элемент не найден>
Else <Элемент найден>
```

Рисунок 10 – псевдокод алгоритма линейного поиска

Код программы

Реализуем алгоритм на языке программирования C++ (рис. 11-12)



```cpp
#include "header_5_2.h"

using namespace std;

int main()
{
    setlocale(LC_ALL, "ru_RU.UTF-8");

    cout << "Enter name of a binary file (with .bin extension): ";
    string file_name;
    cin >> file_name;
    file_name = FILES_FOLDER + file_name;
    ifstream file(file_name, ios::binary);
    if (!file.is_open())
    {
        cout << "Error openning file.\n";
        return 1;
    }

    char id_buffer[ID_SIZE + 1] = {0};
    char name_buffer[NAME_SIZE + 1] = {0};
    char address_buffer[ADDRESS_SIZE + 1] = {0};

    cout << "Enter key to find: ";
    int key;
    cin >> key;

    auto start_time = chrono::high_resolution_clock::now();

    bool found_key = false;
    while (file.read(id_buffer, ID_SIZE))
    {
        file.read(name_buffer, NAME_SIZE);
        file.read(address_buffer, ADDRESS_SIZE);
        int curr_id = stoi(id_buffer);
        if (curr_id == key)
        {
            found_key = true;

            ofstream matches(MATCHES_FILE_NAME);
```

Рисунок 11 – Код к заданию 2 (часть 1)

```
            string entry = id_buffer;
            entry += DELIMITER;

            string curr_name(name_buffer);
            string curr_address(address_buffer);

            curr_name.erase(curr_name.find_last_not_of(' ') + 1);
            curr_address.erase(curr_address.find_last_not_of(' ') + 1);

            entry += curr_name;
            entry += DELIMITER;

            entry += curr_address;
            matches << entry << endl;
            matches.close();
            file.close();
            break;
        }
    }
    auto end_time = chrono::high_resolution_clock::now();
    auto time_delta = chrono::duration_cast<chrono::microseconds>(end_time - start_time);

    if (found_key) cout << "Match saved to " << MATCHES_FILE_NAME << endl;
    else cout << "No matches found.\n";

    cout << "Completion time in microseconds: " << time_delta.count();
    return 0;
}
```

Рисунок 12 – Код к заданию 2 (часть 2)

Результаты тестирования

```
Enter name of a binary file (with .bin extension): 100entries.bin
Enter key to find: 99999
Match saved to matches.txt
Completion time in microseconds: 371
```

Рисунок 13 – Тестирование программы на файле из 100 записей

```
Enter name of a binary file (with .bin extension): 1000entries.bin
Enter key to find: 99999
Match saved to matches.txt
Completion time in microseconds: 836
```

Рисунок 14 – Тестирование программы на файле из 1000 записей

```
Enter name of a binary file (with .bin extension): 10000entries.bin
Enter key to find: 99999
Match saved to matches.txt
Completion time in microseconds: 4124
```

Рисунок 15 – Тестирование программы на файле из 10000 записей

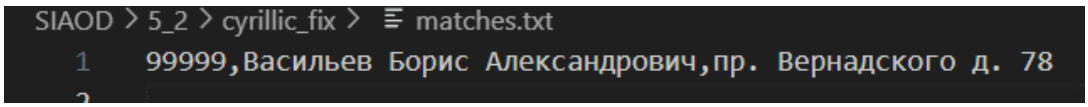


Рисунок 16 – Файл с найденной по ключу записью

Тестирование показало, что программа работает корректно (рис.13-16). Запишем полученные в результате тестирования данные о времени выполнения поиска в таблицу 1.

Таблица 1 — замер времени выполнения линейного поиска

Количество записей в файле	Время поиска в микросекундах
100	371
1000	836
10000	4124

ЗАДАНИЕ 3

Поиск записи в файле с применением дополнительной структуры данных, сформированной в оперативной памяти.

Формулировка задачи

1. Для оптимизации поиска в файле создать в оперативной памяти структур данных – таблицу, содержащую ключ и ссылку (смещение) на запись в файле.
2. Разработать функцию, которая принимает на вход ключ и ищет в таблице элемент, содержащий ключ поиска, а возвращает ссылку на запись в файле. Алгоритм поиска определен в варианте.
3. Разработать функцию, которая принимает ссылку на запись в файле, считывает ее, применяя механизм прямого доступа к записям файла. Возвращает прочитанную запись как результат.
4. Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей.
5. Составить таблицу с указанием результатов замера времени.

Алгоритм

Сформируем в оперативной памяти структуру для прямого доступа к записям из файла, каждый элемент которой будет содержать пару данных: ключ и позицию в байтах, к которой необходимо перейти. Отсортировав элементы данной структуры по значению поля ключа, воспользуемся однородным бинарным поиском, чтоб найти ключ искомый.

Однородный бинарный поиск — это алгоритм, который, подобно классическому бинарному поиску, делит массив на части, но делает это равномерными шагами с использованием степеней двойки. Начальная позиция устанавливается в определённой точке (например, в начале), а затем вычисляется шаг смещения, который уменьшается с каждым циклом, пока не станет достаточно малым для точного нахождения элемента. В зависимости от того, больше или меньше текущий элемент целевого значения, алгоритм перемещает позицию вперёд или назад на рассчитанное смещение.

Ниже приведён алгоритм бинарного однородного поиска в таблице на псевдокоде.

```
i ← 0 // начальная позиция
pow ← 1 // инициализация степени двойки
n ← размер(vec)
Do
    pow ← pow * 2 // увеличиваем степень двойки
    offset ← (n + (pow / 2)) / pow

    If (vec[i][0] = x) Then
        <Элемент найден в позиции i>
        Exit
    Else If (vec[i][0] < x) Then
        i ← i + offset
    Else
        i ← i - offset
While (offset > 0)

<Элемент не найден>
```

Код программы

Реализуем алгоритм на языке программирования C++ (рис. 17-20)

```cpp
#include "header_5_2.h"

using namespace std;

// Necessary function for sorting lookup table
bool cmpr_columns(const array<int, 2>& a, const array<int, 2>& b, int col_index)
{
    return a[col_index] < b[col_index];
}

// The search in lookup table
int uniform_binary_search(vector<array<int, 2>>& vec, int target)
{
    int n = vec.size();
    int pow = 1;
    int offset;
    int curr_position = 0;
    do
    {
        pow <<= 1;

        offset = (n + (pow >> 1)) / pow;

        if (vec[curr_position][0] == target)
        {
            return curr_position;
        }
        else if (vec[curr_position][0] < target)
        {
            curr_position += offset;
        }
        else
        {
            curr_position -= offset;
        }
    } while (offset > 0);
    return -1;
}
```

Рисунок 17 – Код к заданию 3 (часть 1)

```
// Getting formatted entry
string get_entry(string file_name, streampos position_in_file)
{
    char id_buffer[ID_SIZE + 1] = {0};
    char name_buffer[NAME_SIZE + 1] = {0};
    char address_buffer[ADDRESS_SIZE + 1] = {0};

    ifstream file(file_name, ios::binary);
    file.seekg(position_in_file);
    file.read(id_buffer, ID_SIZE);
    file.read(name_buffer, NAME_SIZE);
    file.read(address_buffer, ADDRESS_SIZE);
    file.close();

    string entry = id_buffer;
    entry += DELIMITER;

    string curr_name(name_buffer);
    string curr_address(address_buffer);

    curr_name.erase(curr_name.find_last_not_of(' ') + 1);
    curr_address.erase(curr_address.find_last_not_of(' ') + 1);

    entry += curr_name;
    entry += DELIMITER;

    entry += curr_address;
    return entry;
}
```

Рисунок 18 – Код к заданию 3 (часть 2)

```cpp
int main()
{
    setlocale(LC_ALL, "ru_RU.UTF-8");

    // Opening relevant data file
    cout << "Enter name of a binary file (with .bin extension): ";
    string file_name;
    cin >> file_name;
    file_name = FILES_FOLDER + file_name;
    ifstream file(file_name, ios::binary | ios::ate);
    if (!file.is_open())
    {
        cout << "Error openning file.\n";
        return 1;
    }

    const streampos file_size = file.tellg(); // Getting file size
    const size_t line_length = ID_SIZE + NAME_SIZE + ADDRESS_SIZE;
    const size_t number_of_entries = file_size / line_length;

    // Declaring a lookup table
    vector<array<int, 2>> lookup_table(number_of_entries);

    // Creating char buffers for each of the fields of an entry
    char id_buffer[ID_SIZE + 1] = {0};


    // Filling lookup table with data {ID, byte_position in file}
    for (int i = 0; i < number_of_entries; i++)
    {
        streampos curr_position = i*line_length;
        file.seekg(curr_position);
        file.read(id_buffer, ID_SIZE);
        lookup_table[i][0] = stoi(id_buffer);
        lookup_table[i][1] = curr_position;
    }
    file.close();
```

Рисунок 19 – Код к заданию 3 (часть 3)

```
    // Sorting lookup table
    const int col_index = 0;
    sort(lookup_table.begin(), lookup_table.end(), [col_index](const array<int, 2>& a, const array<int, 2>& b) {
        return cmpr_columns(a, b, col_index);
    });

    // Getting key value to look for
    cout << "Enter key to find: ";
    int key;
    cin >> key;

    // Starting timer
    auto start_time = chrono::high_resolution_clock::now();

    // Search in lookup table
    int position = uniform_binary_search(lookup_table, key);
    auto end_time = chrono::high_resolution_clock::now();
    auto time_delta = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
    if (position == -1)
    {
        cout << "Key not found.\n";
        return 1;
    }

    // Accessing relevant entry from a file
    streampos position_in_file = lookup_table[position][1];
    string entry = get_entry(file_name, position_in_file);

    ofstream matches(MATCHES_FILE_NAME);
    matches << entry << endl;
    cout << "Match saved to " << MATCHES_FILE_NAME << endl;

    // Showing search time
    cout << "Completion time in nanoseconds: " << time_delta.count();

    matches.close();
}
```

Рисунок 20 – Код к задаче 3 (часть 4)

Результаты тестирования

Проведём тестирование программы бинарного однородного поиска на файлах из 100, 1000 и 10000 записей (рис. 21-29)

```
Enter name of a binary file (with .bin extension): 100entries.bin
Enter key to find: 58194
Match saved to matches.txt
Completion time in nanoseconds: 1100
```

Рисунок 21 – Тестирование программы на файле из 100 записей

```
SIAOD > 5_2 > cyrillic_fix > matches.txt
1    58194,Григорьев Дмитрий Романович,ул. Маяковского д. 57
```

Рисунок 22 – Файл с найденной по ключу записью

```
Enter name of a binary file (with .bin extension): 1000entries.bin
Enter key to find: 49620
Match saved to matches.txt
Completion time in nanoseconds: 2300
```

Рисунок 23 – Тестирование программы на файле из 100 записей

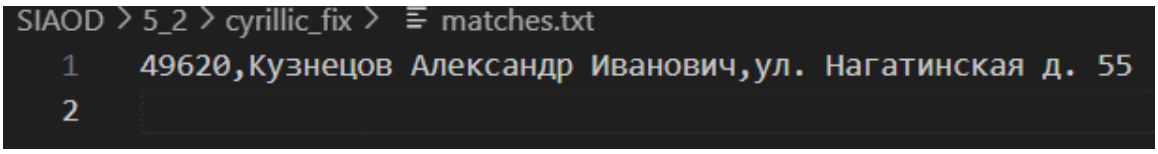


Рисунок 24 – Файл с найденной по ключу записью

```
Enter name of a binary file (with .bin extension): 10000entries.bin
Enter key to find: 11113
Match saved to matches.txt
Completion time in nanoseconds: 1700
```

Рисунок 25 – Тестирование программы на файле из 100 записей

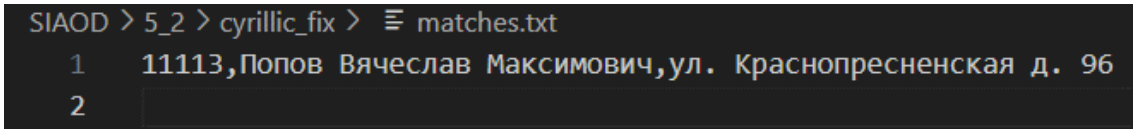


Рисунок 26 – Файл с найденной по ключу записью

```
Enter name of a binary file (with .bin extension): 100entries.bin
Enter key to find: 99999
Match saved to matches.txt
Completion time in nanoseconds: 600
```

Рисунок 27 – Тестирование программы на файле из 100 записей с поиском последней записи

```
Enter name of a binary file (with .bin extension): 1000entries.bin
Enter key to find: 99999
Match saved to matches.txt
Completion time in nanoseconds: 1100
```

Рисунок 28 – Тестирование программы на файле из 1000 записей с поиском последней записи

```
Enter name of a binary file (with .bin extension): 10000entries.bin
Enter key to find: 99999
Match saved to matches.txt
Completion time in nanoseconds: 1900
```

Рисунок 29 – Тестирование программы на файле из 10000 записей с поиском последней записи

Тестирование показало, что программа работает корректно. Запишем полученные в результате тестирования данные о времени выполнения поиска в таблицу 2.

Таблица 2 — замер времени выполнения бинарного однородного поиска

Количество записей в файле	Время поиска в наносекундах
100	600
1000	1100
10000	1900

Как видно из сравнения данных второго столбца таблиц 1 и 2, алгоритм бинарного однородного поиска является значительно более эффективным по сравнению с алгоритмом линейного поиска, особенно при большом количестве данных.

ВЫВОД

В результате выполнения работы был получен практический опыт по применению алгоритмов поиска в таблицах данных. Было сделано сравнение алгоритмов линейного и бинарного однородного поиска, второй оказался более чем в десятки раз эффективнее даже на относительно небольшом наборе данных.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Рысин, М. Л. Введение в структуры и алгоритмы обработки данных : учебное пособие / М. Л. Рысин, М. В. Сартаков, М. Б. Туманова. — Москва : РТУ МИРЭА, 2022 — Часть 2 : Поиск в тексте. Нелинейные структуры данных. Кодирование информации. Алгоритмические стратегии — 2022. — 111 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/310826 (дата обращения: 16.09.2024).

2. Документация по языку C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 16.09.2024).