



МИНОБРНАУКИ РОССИИ

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)

ОТЧЁТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ

«Работа с данными из файла»

**по дисциплине «Структуры и алгоритмы обработки данных (часть
2/2)»**

Выполнил студент группы ИКБО-41-23

Попов А.В.

Принял
Ассистент

Рысин М.Л.

Практические работы выполнены

«__»_____2024 г.

(подпись студента)

«Зачтено»

«__»_____2024 г.

(подпись преподавателя)

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ.....	3
2 ЗАДАНИЕ 1.....	4
2.1 Формулировка задачи.....	4
2.2 Реализация задачи.....	4
3 ЗАДАНИЕ 2.....	7
3.1 Формулировка задачи.....	7
3.2 Реализация задачи.....	7
3.3 Тестирование программы.....	8
4 ЗАДАНИЕ 3.....	10
3.1 Формулировка задачи.....	10
3.2 Реализация задачи.....	11
3.1 Тестирование программы.....	14
5 ВЫВОД.....	15

1 ПОСТАНОВКА ЗАДАЧИ

Разработать программу поиска записей с заданным ключом в двоичном файле с применением различных алгоритмов.

2 ЗАДАНИЕ 1

2.1 Формулировка задачи

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

Персональный вариант №10:

Структура записи файла: регистрационный номер – шестизначное число, название страховой компании.

2.2 Реализация задачи

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <random>

using namespace std;

struct Insurance {
    int registration_number;
    char insurance_company[50];
};

int generateRandomNumber(int min, int max) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dist(min, max);
    return dist(gen);
}

string generateRandomCompany() {
    string companies[] = { "Росгосстрах", "Ингосстрах", "РЕСО-Гарантия", "АльфаСтрахование", "Согласие", "ВТБ Страхование" };
    int randomIndex = generateRandomNumber(0, 5);
    return companies[randomIndex];
}
```

Рисунок 1 — Структура записи в файл и функции для генерации случайных регистрационных номеров и названий компаний

```

void createFiles(const string& binary_filename, const string& text_filename, int record_count) {
    ofstream bin_file(binary_filename, ios::binary);
    ofstream txt_file(text_filename);

    if (!bin_file.is_open() || !txt_file.is_open()) {
        cout << "Ошибка при открытии файлов!" << endl;
        return;
    }

    vector<int> used_numbers;

    for (int i = 0; i < record_count; ++i) {
        Insurance insurance;
        int temp_number;

        do {
            temp_number = generateRandomNumber(100000, 999999);
        } while (find(used_numbers.begin(), used_numbers.end(), temp_number) != used_numbers.end());
        used_numbers.push_back(temp_number);
        insurance.registration_number = temp_number;
        string company = generateRandomCompany();
        strncpy_s(insurance.insurance_company, company.c_str(), sizeof(insurance.insurance_company));
        insurance.insurance_company[sizeof(insurance.insurance_company) - 1] = '\0';

        bin_file.write(reinterpret_cast<char*>(&insurance), sizeof(Insurance));

        txt_file << insurance.registration_number << " " << insurance.insurance_company << endl;
    }

    bin_file.close();
    txt_file.close();
}

int main() {
    string binary_filename = "insurance.bin";
    string text_filename = "insurance.txt";
    int record_count = 100;

    createFiles(binary_filename, text_filename, record_count);
    return 0;
}

```

Рисунок 2 — функция создания текстового и бинарного файлов и основная функция программы

Программа предназначена для генерации данных о страховых полисах и их записи в два файла: бинарный (insurance.bin) и текстовый (insurance.txt). Она генерирует записи, каждая из которых содержит уникальный шестизначный регистрационный номер и название страховой компании.

Названия страховых компаний выбираются случайным образом из заранее заданного списка. Для генерации случайных чисел и выбора

компаний используются инструменты стандартной библиотеки C++, что гарантирует равномерное распределение псевдослучайных чисел.

В бинарном файле запись хранится в компактной форме с использованием побайтной записи структуры Insurance. Это позволяет в будущем быстро считывать данные программно. Размер одной записи вычисляется как сумма размеров всех полей структуры:

- int registration_number: 4 байта.
- char insurance_company[50]: 50 байт.

Общий размер записи составляет 54 байта. Это значение используется для организации прямого доступа к записям в бинарном файле.

3 ЗАДАНИЕ 2

3.1 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 и составить таблицу с указанием результатов замера времени.

3.2 Реализация задачи

```
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>

using namespace std;
using namespace chrono;

struct Insurance {
    int registration_number;
    char insurance_company[50];
};

int main() {
    setlocale(LC_ALL, "Russian");

    ifstream file("insurance.bin", ios::binary);
    if (!file.is_open()) {
        cerr << "Ошибка при открытии файла!" << endl;
        return 1;
    }

    int search_key;
    cout << "Введите регистрационный номер для поиска: ";
    cin >> search_key;

    Insurance insurance;
    bool found = false;

    auto start_time = high_resolution_clock::now();

    while (file.read(reinterpret_cast<char*>(&insurance), sizeof(Insurance))) {
        if (insurance.registration_number == search_key) {
            cout << "Запись найдена!" << endl;
            cout << "Регистрационный номер: " << insurance.registration_number << endl;
            cout << "Название страховой компании: " << insurance.insurance_company << endl;
            found = true;
            break;
        }
    }

    auto end_time = high_resolution_clock::now();
    auto elapsed_time = duration_cast<microseconds>(end_time - start_time).count();

    if (!found) {
        cout << "Запись с регистрационным номером " << search_key << " не найдена." << endl;
    }

    cout << "Время выполнения поиска: " << float(elapsed_time) / 1000000.0 << " секунд." << endl;

    file.close();
    return 0;
}
```

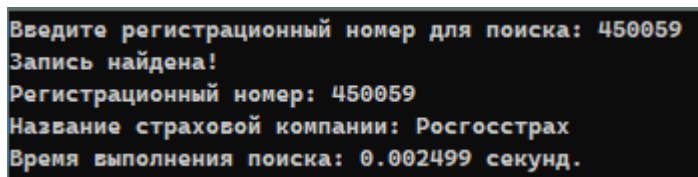
Рисунок 3 — реализация задания 2 на C++

После открытия файла в режиме бинарного чтения (ios::binary) программа проверяет, удалось ли открыть файл. Если файл недоступен, выводится сообщение об ошибке, и выполнение завершается. Пользователь вводит искомый регистрационный номер, который будет сравниваться с номерами в файле.

Записи читаются из файла последовательно, блоками размера структуры Insurance. Каждая запись сравнивается с введённым регистрационным номером. Если запись с указанным номером найдена, выводятся её данные.

Если запись найдена, поиск завершается досрочно, и программа переходит к выводу результатов. Для повышения информативности программа измеряет время выполнения поиска.

3.3 Тестирование программы



```
Введите регистрационный номер для поиска: 450059
Запись найдена!
Регистрационный номер: 450059
Название страховой компании: Росгосстрах
Время выполнения поиска: 0.002499 секунд.
```

Рисунок 4 — Результат выполнения программы

Проведём практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей. Во всех случаях ключевое значение будет в конце файла, то есть будет воссоздан худший случай для данного алгоритма. Приведём результаты тестирования в таблице 1.

Кол-во записей	Время выполнения (секунды)
100	0.002873
1 000	0.003499
10 000	0.005686

Таблица 1 - Практическая оценка времени выполнения линейного поиска

В результате проделанных тестов можно сделать вывод о том, что сложность алгоритма линейного поиска в худшем случае, то есть когда ключевое значение находится на последнем месте, равна $O(n)$. В лучшем случае, то есть когда ключевое значение находится на первом месте, сложность алгоритма будет равна $O(1)$.

4 ЗАДАНИЕ 3

3.1 Формулировка задачи

Для оптимизации поиска в файле создать в оперативной памяти структур данных – таблицу, содержащую ключ и ссылку (смещение) на запись в файле. 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. Разработать функцию, которая принимает ссылку на запись в файле, считывает ее, применяя механизм прямого доступа к записям файла. Возвращает прочитанную запись как результат. Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей.

Персональный вариант №10:

Алгоритм поиска: Интерполяционный поиск.

3.2 Реализация задачи

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace chrono;

struct Insurance {
    int registration_number;
    char insurance_company[50];
};

struct IndexEntry {
    int registration_number;
    streampos offset;
};

bool compare(const IndexEntry& a, const IndexEntry& b) {
    return a.registration_number < b.registration_number;
}

vector<IndexEntry> createIndexTable(const string& filename) {
    ifstream file(filename, ios::binary);
    if (!file.is_open()) {
        cerr << "Ошибка открытия файла!" << endl;
        exit(1);
    }

    vector<IndexEntry> index_table;
    Insurance insurance;
    streampos offset = 0;

    while (file.read(reinterpret_cast<char*>(&insurance), sizeof(Insurance))) {
        index_table.push_back({ insurance.registration_number, offset });
        offset = file.tellg();
    }

    file.close();

    sort(index_table.begin(), index_table.end(), compare);

    return index_table;
}
```

Рисунок 5 - Структуры данных, функции сравнения и создания в оперативной памяти таблицы

Функция `createIndexTable` предназначена для создания индексационной таблицы, которая позволяет ускорить поиск записей в бинарном файле. Основная идея заключается в том, чтобы собрать информацию о каждой записи в файле, включая её регистрационный номер и смещение (позицию) в файле, и сохранить эти данные в виде вектора структур `IndexEntry`. Затем таблица сортируется по регистрационным номерам, чтобы обеспечить возможность реализовать интерполяционный алгоритм поиска.

```

streampos interpolationSearch(const vector<IndexEntry>& index_table, int key) {
    int low = 0, high = index_table.size() - 1;

    while (low <= high && key >= index_table[low].registration_number && key <= index_table[high].registration_number) {
        if (low == high) {
            if (index_table[low].registration_number == key)
                return index_table[low].offset;
            break;
        }

        if (key == index_table[high].registration_number) {
            return index_table[high].offset;
        }

        if (key == index_table[low].registration_number) {
            return index_table[low].offset;
        }

        int pos = low + ((key - index_table[low].registration_number) * (high - low)) /
            (index_table[high].registration_number - index_table[low].registration_number);

        if (pos < low) pos = low;
        if (pos > high) pos = high;

        if (index_table[pos].registration_number == key)
            return index_table[pos].offset;

        if (index_table[pos].registration_number < key) {
            low = pos + 1;
        }
        else {
            high = pos - 1;
        }
    }

    return -1;
}

```

Рисунок 6 — Функция интерполяционного поиска

Функция `interpolationSearch` реализует интерполяционный поиск в отсортированном массиве индексов. Этот алгоритм оптимизирует процесс поиска за счёт использования математической интерполяции, которая определяет позицию искомого элемента, основываясь на значении ключа. В отличие от бинарного поиска, интерполяционный более эффективен для данных с равномерным распределением.

```

Insurance readInsuranceByOffset(const string& filename, streampos offset) {
    ifstream file(filename, ios::binary);
    if (!file.is_open()) {
        cerr << "Ошибка открытия файла!" << endl;
        exit(1);
    }

    Insurance insurance;
    file.seekg(offset);
    file.read(reinterpret_cast<char*>(&insurance), sizeof(Insurance));
    file.close();

    return insurance;
}

```

Рисунок 7 — Функция прямого доступа к данным

Функция `readInsuranceByOffset` предназначена для чтения конкретной записи из бинарного файла на основе её смещения (позиции в файле). Она используется для прямого доступа к данным, что особенно полезно в сочетании с индексной таблицей, где хранится информация о смещениях всех записей.

```

int main() {
    setlocale(LC_ALL, "Russian");

    string filename = "insurance.bin";

    int search_key;
    cout << "Введите регистрационный номер для поиска: ";
    cin >> search_key;

    auto start_time = high_resolution_clock::now();

    vector<IndexEntry> index_table = createIndexTable(filename);

    streampos offset = interpolationSearch(index_table, search_key);

    auto end_time = high_resolution_clock::now();
    auto total_time = duration_cast<microseconds>(end_time - start_time).count();

    if (offset != -1) {
        Insurance insurance = readInsuranceByOffset(filename, offset);

        cout << "Запись найдена!" << endl;
        cout << "Регистрационный номер: " << insurance.registration_number << endl;
        cout << "Название страховой компании: " << insurance.insurance_company << endl;
    }
    else {
        cout << "Запись с регистрационным номером " << search_key << " не найдена." << endl;
    }

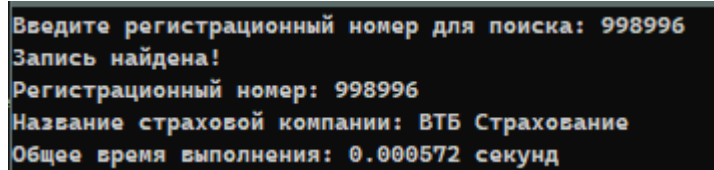
    cout << "Общее время выполнения: " << float(total_time) / 1000000.0 << " секунд" << endl;

    return 0;
}

```

Рисунок 8 — основная функция программы

3.1 Тестирование программы



```
Введите регистрационный номер для поиска: 998996
Запись найдена!
Регистрационный номер: 998996
Название страховой компании: ВТБ Страхование
Общее время выполнения: 0.000572 секунд
```

Рисунок 9 - Результат выполнения программы

Проведём практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей. Во всех случаях ключевое значение будет в с наибольшим ключевым значением, то есть будет воссоздан худший случай для данного алгоритма.

Приведём результаты тестирования в таблице 2.

Кол-во записей	Время выполнения (секунды)
100	0.000572
1 000	0.002138
10 000	0.018679

Таблица 2 - Практическая оценка времени выполнения интерполяционного поиска

В результате проделанных тестов можно сделать вывод о том, что сложность алгоритма линейного поиска в худшем случае, то есть когда ключевое значение находится на последнем месте, равна $O(n)$. В лучшем случае, то есть когда ключевое значение находится на первом месте, сложность алгоритма будет равна $O(1)$. В среднем случае его сложность будет равна $O(\log n * \log n)$, что лучше, чем $O(\log n)$ для бинарного поиска.

5 ВЫВОД

В ходе выполнения данной работы были изучены алгоритмы поиска в таблице при работе с данными из файла. Было разработано и реализовано два эффективный алгоритма внешней сортировки: линейный и интерполяционный. Линейный алгоритм был лёгок в реализации, его можно использовать и с отсортированными данными, и с не отсортированными. Однако, имеет в среднем случае сложность $O(n)$, что, в свою очередь, замедляет его выполнение для больших наборов данных. Интерполяционный алгоритм поиска является более сложным в реализации, его можно использовать только с отсортированными данными. Но он имеет в среднем случае сложность $O(\log n * \log n)$, что делает его очень достаточно для поиска в отсортированных данных и на большом их наборе.