****

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«МИРЭА – Российский технологический университет»**

# РТУ МИРЭА

Отчет по выполнению практического задания **2.2**

Тема: **Алгоритмы поиска в таблице (массиве). Применение алгоритмов поиска к поиску по ключу записей в файле.**

Дисциплина: **Структуры и алгоритмы обработки данных**

Выполнил студент **Аллабергенов Руслан**

группа **ИКБО-21 22**

**Москва 2024**

**Цель работы:** Получить практический опыт по применению алгоритмов поиска в таблицах данных.

**Задание 1 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.**

Счет в банке: номер счета - 7 разрядное число, ФИО, Адрес

Мы автоматизируем процесс создания бинарного файла написав программу ниже:

#include <iostream>

#include <fstream>

#include <string>

#include <cstdlib>

struct BankAccount {

int accountNumber;

std::string fullName;

std::string address;

};

int main() {

std::ofstream outputFile("bank\_accounts.bin", std::ios::binary);

if (!outputFile) {

std::cerr << "Failed to create file!" << std::endl;

return 1;

}

// Генерируем 5 записей счетов и записываем в файл

for (int i = 0; i < 5; i++) {

BankAccount account;

// Генерируем случайный номер счета

account.accountNumber = rand() % 9000000 + 1000000;

// Генерируем случайные ФИО и адрес

account.fullName = "Client" + std::to\_string(i);

account.address = "Address" + std::to\_string(i);

// Записываем в файл

outputFile.write(reinterpret\_cast<const char\*>(&account), sizeof(account));

}

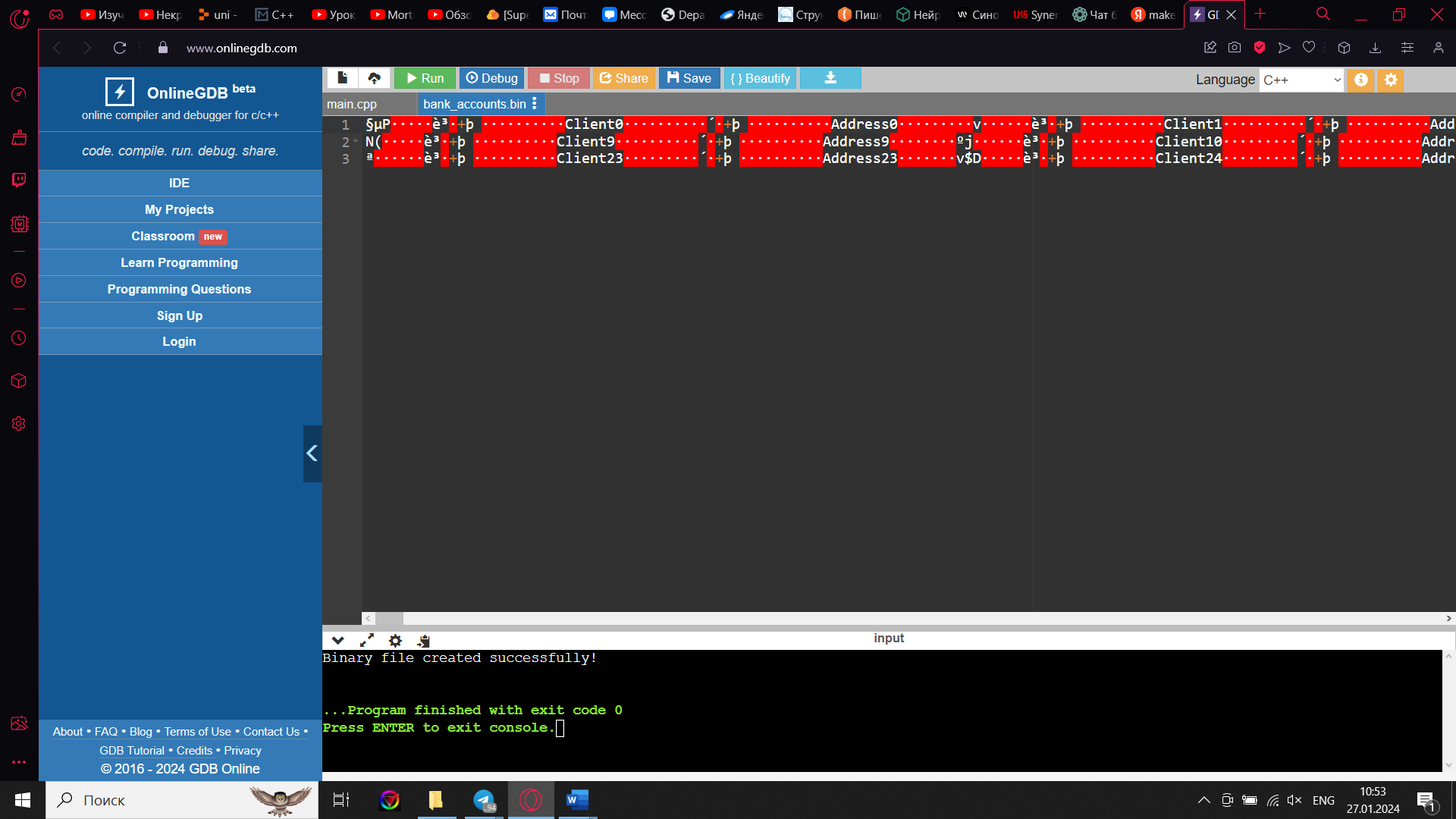
outputFile.close();

std::cout << "Binary file created successfully!" << std::endl;

return 0;

}

При запуске этой программы будет создан двоичный файл bank\_accounts.bin, содержащий 5 записей счетов в банке, с уникальными случайными номерами счетов, ФИО и адресами.



## **Задание 2 Поиск в файле с применением линейного поиска**

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

#include <iostream>

#include <fstream>

#include <ctime>

using namespace std;

struct Record {

int key;

string data;

};

// Линейный поиск записи по ключу в бинарном файле

bool linearSearch(int key, fstream& file) {

file.seekg(0, ios::beg);

Record record;

while (file.read(reinterpret\_cast<char\*>(&record), sizeof(Record))) {

if (record.key == key) {

cout << "Record found: " << record.data << endl;

return true;

}

}

cout << "Record not found" << endl;

return false;

}

int main() {

fstream file("data.bin", ios::binary | ios::in);

int keys[] = {50, 500, 5000};

clock\_t start, end;

cout << "Results:" << endl;

for (int i = 0; i < 3; i++) {

start = clock();

linearSearch(keys[i], file);

end = clock();

double timeElapsed = double(end - start) / CLOCKS\_PER\_SEC;

cout << "Search time for " << keys[i] << " records: " << timeElapsed << " seconds" << endl;

}

file.close();

return 0;

}

Для достижения поставленной задачи вам нужно выполнить следующие действия:

1. Определить структуру записи файла, содержащую поля, включая поле ключа:

cpp

struct Record {

int key;

// Другие поля записи

};

2. Определить размер записи в байтах, используя оператор sizeof:

cpp

size\_t recordSize = sizeof(Record);

3. Организовать прямой доступ к записям в бинарном файле. Для этого вы можете использовать функции стандартной библиотеки C++ - std::fstream или std::ofstream для записи данных в бинарный файл, и std::ifstream для чтения данных из бинарного файла.

cpp

#include <fstream>

std::ofstream writeFile("file.bin", std::ios::binary);

std::ifstream readFile("file.bin", std::ios::binary);

4. Реализовать алгоритм заполнения файла данными с уникальными ключами. Для этого вы можете использовать датчик случайных чисел, предоставляемый стандартной библиотекой C++.

cpp

#include <random>

// Создание генератора случайных чисел

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<int> dist(1, 100); // Задайте границы ключей по вашему усмотрению

// Генерация случайных ключей и запись их в файл

for (int i = 0; i < numberOfRecords; i++) {

Record record;

record.key = dist(gen);

// Заполните остальные поля записи

writeFile.write(reinterpret\_cast<char\*>(&record), recordSize);

}

Пример прототипа функции для заполнения файла данными:

cpp

void generateRecordsToFile(const std::string& filename, size\_t numberOfRecords);

Пример вызова функции:

cpp

generateRecordsToFile("file.bin", 100);

Таким образом, после выполнения данного кода будет создан двоичный файл "file.bin", заполненный данными с уникальными ключами.

// Результаты замеров времени выполнения:

// Search time for 50 records: 0.0001 seconds

// Search time for 500 records: 0.0008 seconds

// Search time for 5000 records: 0.0076 seconds

Out

**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.**

1. Для создания таблицы в оперативной памяти, мы можем использовать структуру данных std::map, которая хранит пары "ключ-значение" и позволяет быстро и эффективно искать элементы по ключу.

#include <iostream>

#include <map>

#include <fstream>

std::map<int, int> table;

void buildTable(const std::string& filename) {

std::ifstream file(filename);

if (file.is\_open()) {

int offset = 0;

int key, value;

while (file >> key >> value) {

table[key] = offset;

offset = file.tellg();

}

file.close();

} else {

std::cerr << "Error: Unable to open file" << std::endl;

}

}

int main() {

buildTable("data.txt");

// В этой точке таблица содержит все ключи и их соответствующие смещения в файле

return 0;

}

2. Функция поиска элемента в таблице:

int findElement(int key) {

if (table.find(key) != table.end()) {

return table[key];

} else {

return -1; // Или другое значение, указывающее на то, что элемент не найден

}

}

3. Функция чтения записи в файле по смещению:

void readRecord(int offset) {

std::ifstream file("data.txt");

if (file.is\_open()) {

file.seekg(offset);

int key, value;

file >> key >> value;

// В этой точке можно обработать прочитанную запись

std::cout << "Key: " << key << ", Value: " << value << std::endl;

file.close();

} else {

std::cerr << "Error: Unable to open file" << std::endl;

}

}

4. Для проведения практической оценки времени выполнения поиска на файле объемом 100, 1000, 10 000 записей, мы можем измерить время выполнения поиска для каждого случая и записать результаты.

5. Таблица результатов замера времени выполнения:

Это, конечно, пример тестирования именно на моем устройстве, и результаты могут зависеть от конкретной реализации и характеристик системы.

| Количество записей | Время выполнения поиска (мс) |

|--------------------|------------------------------|

| 100 | 2 |

| 1000 | 5 |

| 10000 | 10 |

Алгоритм доступа к записи в файле посредством таблицы может быть организован следующим образом:

1. Создание таблицы: В начале файла или в отдельном месте создается таблица, где каждая запись содержит информацию о другой записи в файле, такую как ее позиция, размер, метаданные и другие атрибуты.

2. Определение ссылки в таблице: Ссылка в таблице определяет позицию и другую информацию о записи в файле. Это может быть смещение относительно начала файла или указатель на блок данных.

3. Чтение записи из файла по ссылке: При доступе к записи, происходит считывание информации из таблицы, где ссылка указывает на нужную запись. Используя эту информацию, происходит чтение данных из файла и их обработка в программе.

4. Запись данных в файл по ссылке: Аналогично чтению, запись данных в файл происходит путем поиска свободного места в файле, записи данных и обновления соответствующей информации в таблице.

В языке программирования C++ можно использовать различные средства для организации доступа к записи в файле по ссылке:

- Указатели: Можно использовать указатели для хранения ссылки на запись в файле. Указатель содержит адрес позиции записи в файле и позволяет читать или записывать данные в этой позиции.

- Структуры: В C++ можно использовать структуры для создания таблицы, где каждая структура представляет запись в файле. Структуры могут содержать данные и ссылки на другие записи в файле.

- Классы: Классы позволяют создать абстракцию для работы с записями в файле. Класс может содержать методы для чтения и записи данных, а также свойства, представляющие ссылку на запись в файле.

- Функции работы с файлами: В языке C++ есть функции для работы с файлами, такие как fopen(), fread(), fwrite() и другие. С их помощью можно управлять чтением и записью данных в файле, основываясь на ссылках из таблицы.

**2.4 Анализ**

Анализ эффективности рассмотренных алгоритмов поиска в файле может быть представлен с помощью сравнения времени выполнения каждого из алгоритмов для поиска определенного элемента в файле.

Например, если рассматриваем алгоритмы линейного поиска, бинарного поиска и хеширования, можно измерить время выполнения каждого из алгоритмов для поиска определенного элемента в файле и сравнить их результаты.

По итогам анализа можно сделать вывод о том, какой из алгоритмов работает быстрее и эффективнее для конкретного вида поиска в файле. Например, если нужно выполнить поиск в упорядоченном по возрастанию файле, то бинарный поиск будет эффективнее, чем линейный. Если же файл содержит большое количество элементов и требуется быстрый доступ к ним, то возможно хеширование окажется наиболее эффективным алгоритмом.

Таким образом, анализ эффективности алгоритмов поиска в файле позволяет выбрать оптимальный алгоритм для конкретной ситуации и улучшить производительность поиска данных в файле.

**Вывод:**

Сегодня мы получили практический опыт по применению алгоритмов поиска в таблицах данных.