МИНОБРНАУКИ РОССИИ

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

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных технологий

Отчет по практической работе №2

по дисциплине «Структуры и алгоритмы обработки данных»
по теме «Алгоритмы поиска в таблице (массиве). Применение алгоритмов поиска к поиску по ключу записей в файле»

Выполнил:
Студент группы ИКБО-13-22 Ефремова Полина Александровна

Проверил: ассистент Муравьёва Е.А.

МОСКВА 2023 г.

1. ВВЕДЕНИЕ

Цель: получить практический опыт по применению алгоритмов поиска в таблицах данных.

Задание (Вариант 9): разработать программу поиска записей с заданным ключом в двоичном файле с применением различных алгоритмов.

1.1. Задание 1

Создать двоичный файл из записей (структура записи – ***страховой полис: номер полиса, компания, фамилия владельца***). Поле ключа записи подчеркнуто. Заполнить файл данными, используя для поля ключа датчик случайных чисел. Ключи записей в файле уникальны.

Рекомендация: создайте сначала текстовый файл, а затем преобразуйте его в двоичный.

При открытии файла обеспечить контроль существования и открытия файла.

1.2. Задание 2

Поиск в файле с применением линейного поиска:

1) Разработать программу поиска записи по ключу в бинарном файле, созданном в первом задании, 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.

1.3. Задание 3

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

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: ***Фибоначчи поиск***.

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.

2. ХОД РАБОТЫ

2.1. Задание 1

2.1.1. Описание программы

Размер записи в байтах будет равен размеру структуры.

Листинг 1

```
size_t bytes = sizeof(InsurancePolicy);
cout << "Размер записи в байтах: " << bytes << endl << endl;
```

Организация прямого доступа к записям в бинарном файле подразумевает возможность читать и записывать данные в файл без необходимости считывать или записывать всё содержимое файла целиком. Это может быть полезно, когда есть большой файл, и необходимо работать только с конкретными записями, не загружая весь файл в память.

В программе прямой доступ к записям в бинарном файле осуществляется, используя функцию ***write()***.

Полное описание процесса:

1) **Открытие бинарного файла для записи**. Используется флаг ***ios::binary***, чтобы предотвратить форматирование данных файла.

2) **Запись структуры в бинарный файл**. Используется метод ***write***, чтобы записать данные структуры в бинарный файл. Используется ***reinterpret_cast*** для преобразования указателя на структуру в указатель на массив символов, что позволяет записать данные как последовательность байтов.

3) **Закрытие бинарного файла**.

2.1.2. Код программы

Листинг 2

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <chrono>//----> Для измерения времени
#include <ctime>
#include <set>//----> Для набора уникальных элементов
```

```cpp
using namespace std;

//----> Структура записи - Страховой полис: номер полиса, компания, фамилия владельца
struct InsurancePolicy
{
    unsigned long int number = 0;
    string company;
    string surname;
};

//----> Номер полиса, состоящий из 8 чисел
unsigned long int Randomizer()
{
    unsigned long int min = 10000000;
    unsigned long int max = 99999999;
    return min + rand() % (max - min + 1);
}

int main()
{
    //----> ПЕРВОЕ ЗАДАНИЕ

    setlocale(LC_ALL, "ru");
    srand(static_cast<unsigned int>(time(nullptr)));//----> Инициализация генератора

    ofstream textFile("C:\\Users\\efr-p\\Desktop\\clean\\SiAOD\\practice 2\\textFile.txt");//----> Текстовый файл
    if (!textFile)
    {
        cout << "Текстовый файл не удалось открыть для записи +_+" << endl;
        return 1;
    }
    cout << "Текстовый файл успешно открыт для записи ^_^" << endl << endl;

    //----> Предусловие: бинарный файл должен быть открыт
    //----> Открываем файл для записи в бинарном режиме (флаг - ios::binary)
    ofstream binaryFileOut("C:\\Users\\efr-p\\Desktop\\clean\\SiAOD\\practice 2\\binaryFile.bin", ios::binary);//----> Бинарный файл
    if (!binaryFileOut)
    {
        cerr << "Бинарный файл не удалось открыть для записи +_+" << endl;
        return 1;
    }
    cout << "Бинарный файл успешно открыт для записи ^_^" << endl << endl;

    int quant = 10;//----> Количество записей
    set<int> unique;

    InsurancePolicy policy;
    for (int i = 0; i < quant; ++i)
    {
        do {
            policy.number = Randomizer();
        } while (unique.count(policy.number) > 0);
        unique.insert(policy.number);
        policy.company = "Компания_" + to_string(i);
        policy.surname = "Фамилия_" + to_string(i);

        textFile << policy.number << ' ' << policy.company << ' ' << policy.surname << '\n';

        //----> Постусловие: записываем данные структуры как последовательность байтов без какой-либо интерпретации (размер структуры в байтах)
```

```
        binaryFileOut.write(reinterpret_cast<const char*>(&policy),
sizeof(policy));
    }
    cout << "Генерация текстового и бинарного файла завершена ^_^" << endl;

    size_t bytes = sizeof(InsurancePolicy);
    cout << "Размер записи в байтах: " << bytes << endl;
    cout << "Количество записей: " << quant << endl << endl;

    textFile.close();
    binaryFileOut.close();
```

2.1.3. Тестирование

```
Текстовый файл успешно открыт для записи ^_^

Бинарный файл успешно открыт для записи ^_^

Генерация текстового и бинарного файла завершена ^_^
Размер записи в байтах: 88
Количество записей: 100
```

Рисунок 1

2.2. Задание 2

2.2.1. Описание программы

Алгоритм линейного поиска — это простой алгоритм поиска элемента в массиве или списке. Он просто перебирает элементы по очереди, начиная с первого элемента, и сравнивает их с ключевым элементом до тех пор, пока не будет найден элемент, который равен ключевому, или пока не будут просмотрены все элементы.

Листинг 3

```
Функция ЛинейныйПоиск(массив, цель):
  Для каждого элемента в массиве с индексом i от 0 до длины массива - 1:
    Если элемент[i] равен ключу:
      Вернуть i (или сам элемент)

  Вернуть -1 (или другое значение, чтобы указать, что элемент не найден)
```

2.2.2. Код программы

Листинг 4

```
//----> ВТОРОЕ ЗАДАНИЕ

    //----> Предусловие: файл должен быть открыт в бинарном режиме для чтения
    ifstream binaryFileIn("C:\\Users\\efr-p\\Desktop\\clean\\SiAOD\\practice
2\\binaryFile.bin", ios::binary);
    if (!binaryFileIn)
    {
```

```
        cout << "Бинарный файл не удалось открыть для чтения +_+" << endl;
        return 1;
    }
    cout << "Бинарный файл успешно открыт для чтения ^_^" << endl << endl;

    cout << "   Добро пожаловать в информационную базу СТРАХпол!" << endl << "
Здесь вы можете проверить действительность страхового полиса юр.лица $_$" << endl
<< endl;
    unsigned long int key;
    cout << "   Введите номер страхового полиса (8 чисел): ";
    cin >> key;

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

    while (binaryFileIn.read(reinterpret_cast<char*>(&policy), sizeof(policy)))
    {
        if (policy.number == key)
        {
            cout << endl << "   Страховой полис найден ^o^" << endl;
            cout << "   Номер полиса ---> " << policy.number << endl;
            cout << "   Компания ---> " << policy.company << endl;
            cout << "   Фамилия владельца ---> " << policy.surname << endl;

            //----> Постусловие: ключ найден, функция должна вернуть 0 и вывести
информацию
            binaryFileIn.close();

            auto end_time = chrono::high_resolution_clock::now();
            auto duration = chrono::duration_cast<chrono::microseconds>(end_time -
start_time);
            cout << "   Время выполнения: " << duration.count() << " [мкс]" <<
endl;

            return 0;
        }
    }
    //----> Постусловие: ключ не найден, функция должна вернуть 0 и вывести
сообщение
    cout << "   К сожалению страховой полис с номером " << key << " не был найден
X_X" << endl;
    binaryFileIn.close();

    auto end_time = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end_time -
start_time);
    cout << "   Время выполнения: " << duration.count() << " [мкс]" << endl;

    return 0;
}
```

2.2.3. Тестирование

```
Текстовый файл успешно открыт для записи ^_^

Бинарный файл успешно открыт для записи ^_^

Генерация текстового и бинарного файла завершена ^_^
Размер записи в байтах: 88
Количество записей: 100

Бинарный файл успешно открыт для чтения ^_^

   Добро пожаловать в информационную базу СТРАХпол!
   Здесь вы можете проверить действительность страхового полиса юр.лица $_$

   Введите номер страхового полиса (8 чисел): 10018924

   Страховой полис найден ^o^
   Номер полиса ---> 10018924
   Компания ---> Компания_50
   Фамилия владельца ---> Фамилия_50
   Время выполнения: 1762 [мкс]
```

```
Текстовый файл успешно открыт для записи ^_^

Бинарный файл успешно открыт для записи ^_^

Генерация текстового и бинарного файла завершена ^_^
Размер записи в байтах: 88
Количество записей: 1000

Бинарный файл успешно открыт для чтения ^_^

   Добро пожаловать в информационную базу СТРАХпол!
   Здесь вы можете проверить действительность страхового полиса юр.лица $_$

   Введите номер страхового полиса (8 чисел): 10015564

   Страховой полис найден ^o^
   Номер полиса ---> 10015564
   Компания ---> Компания_500
   Фамилия владельца ---> Фамилия_500
   Время выполнения: 2091 [мкс]
```

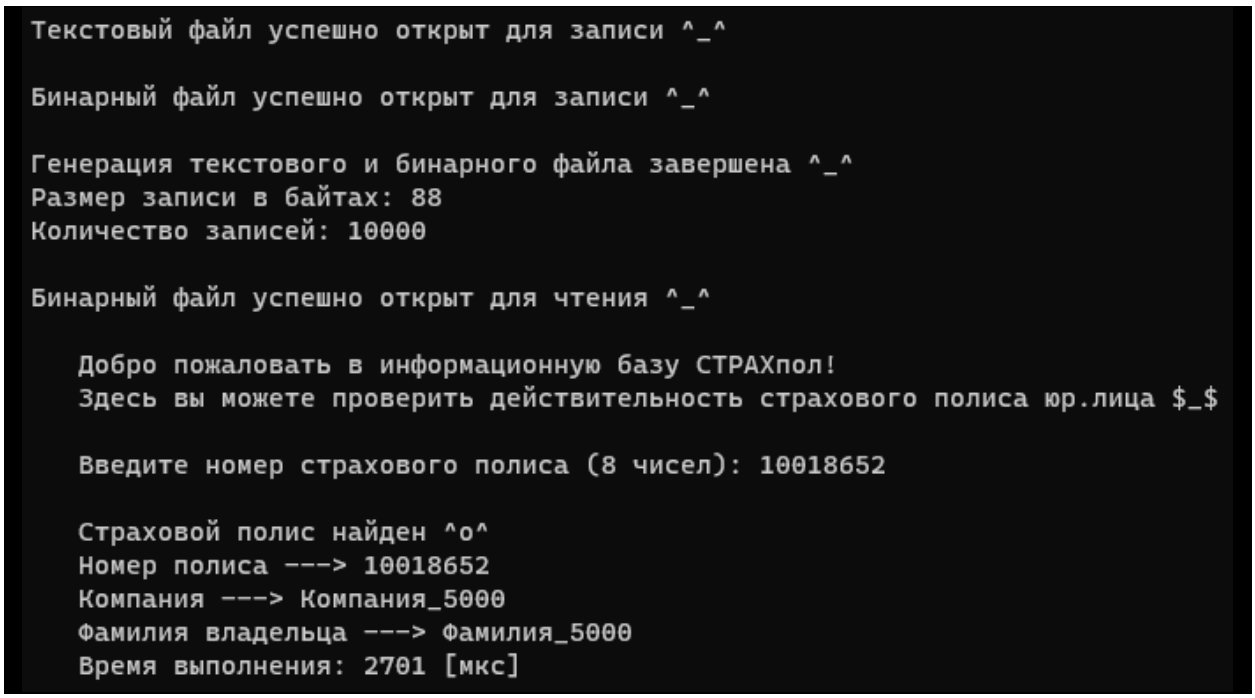


Рисунок 2

2.2.4. Практическая оценка времени выполнения

Количество записей	Время выполнения [мкс]/[мс]	Размер записи [Байт]
100	1762 / 1,76	88
1 000	2091 / 2,09	
10 000	2701 / 2,70	

2.3. Задание 3

2.3.1. Описание алгоритма доступа к записи в файле посредством таблицы

Создание пустой таблицы:

В начале кода, создается пустой вектор, который будет представлять таблицу данных.

Открытие бинарного файла для чтения:

Код пытается открыть бинарный файл для чтения в режиме бинарного ввода. Если открытие файла не удалось, программа выводит сообщение об ошибке и завершает выполнение.

Чтение записей из бинарного файла:

1) Код создает переменную “*ссылку*”, которая используется для отслеживания смещения (позиции) в бинарном файле.

2) Затем с помощью цикла, начинается чтение записи из бинарного файла, при этом предполагается, что каждая запись имеет размер, соответствующий размеру структуры (в байтах). Внутри этого **цикла**:

- Создается экземпляр структуры таблицы.
- В этот экземпляр записывается ключ, который представляет собой номер полиса, считанный из бинарного файла.
- Затем записывается смещение текущей записи в файле, которое равно текущему значению смещения.
- Экземпляр записи таблицы добавляется в конец вектора.
- Смещение увеличивается на 1.

Завершение чтения и закрытие файла:

После завершения цикла чтения записей, бинарный файл закрывается.

Ссылка (или смещение) в таблице определяет на какой позиции в файле начинается запись данных для каждой записи в таблице данных. При этом каждая запись будет иметь свое уникальное смещение. ***Смещение используется*** для того, чтобы можно было легко найти и прочитать соответствующую запись в бинарном файле, если известен ключ (номер полиса). Это позволяет быстро перейти к нужной записи в файле, необходимой для выполнения поиска или чтения данных из него.

2.3.2. Описание алгоритма поиска

Фибоначчи-поиск — это один из алгоритмов поиска, который использует числа Фибоначчи для определения местоположения искомого элемента в ***упорядоченном*** массиве данных.

Основная идея Фибоначчи-поиска заключается в том, чтобы делить массив на подмассивы согласно числам Фибоначчи, и сравнивать искомый элемент с элементами, находящимися на позициях чисел Фибоначчи. Если искомый элемент больше текущего элемента, то поиск продолжается в правой части массива, иначе - в левой части. Этот процесс продолжается до тех пор, пока не будет найден искомый элемент или массив станет пустым.

Преимущество Фибоначчи-поиска заключается в том, что он имеет ***логарифмическую сложность*** по времени, что делает его эффективным для больших упорядоченных массивов данных. Однако для его реализации требуется предварительное вычисление чисел Фибоначчи и дополнительное использование памяти для их хранения.

Листинг 5

```
Функция ФибоначчиПоиск(массив, цель):
    n = ДлинаМассива(массив)
    FnMinus2 = 0
    FnMinus1 = 1
    Fn = FnMinus1 + FnMinus2

    Пока Fn < n:
        FnMinus2 = FnMinus1
        FnMinus1 = Fn
        Fn = FnMinus1 + FnMinus2

    offset = -1

    Пока Fn > 1:
        i = Минимум(offset + FnMinus2, n - 1)

        Если массив[i] < цель:
            Fn = FnMinus1
```

```
        FnMinus1 = FnMinus2
        FnMinus2 = Fn - FnMinus1
        offset = i
    Иначе Если массив[i] > цель:
        Fn = FnMinus2
        FnMinus1 = FnMinus1 - FnMinus2
        FnMinus2 = Fn - FnMinus1
    Иначе:
        Вернуть i // Элемент найден

  Если FnMinus1 == 1 И массив[offset + 1] == цель:
    Вернуть offset + 1

  Вернуть -1 // Элемент не найден
```

2.3.3. Код программы

Листинг 6

```
#include <algorithm>
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <vector>
#include <string>
#include <chrono>//----> Для измерения времени
#include <ctime>
#include <set>//----> Для набора уникальных элементов

using namespace std;

//----> Структура записи – Страховой полис: номер полиса, компания, фамилия
владельца
struct InsurancePolicy
{
    unsigned long int number = 0;
    string company;
    string surname;
};

//----> Структура данных для хранения ключа и смещения
struct TableContent
{
    unsigned long int key;
    int fileOffset;//----> Ссылка на запись в файле (смещение)
};

//----> Номер полиса, состоящий из 8 чисел
unsigned long int Randomizer()
{
    unsigned long int min = 10000000;
    unsigned long int max = 99999999;
    return min + rand() % (max - min + 1);
}

//----> Фибоначчи поиск
int Fibonacci(const vector<TableContent>& Table, unsigned long int key)
{
    //----> Предусловие: Должен быть создан объект, содержащий n элементов, и ключ

    int n = Table.size();
    //----> Создание последовательности чисел Фибаначчи
```

```
    int FnMinus2 = 0;
    int FnMinus1 = 1;
    int Fn = FnMinus1 + FnMinus2;

    while (Fn < n)//----> Пока текущее число Фибаначчи меньше n
    {
        //----> Продвижение вперед
        FnMinus2 = FnMinus1;
        FnMinus1 = Fn;
        Fn = FnMinus1 + FnMinus2;
    }

    int offset = -1;

    while (Fn > 1)//----> Пока текущее число Фибаначчи больше 1
    {
        int i = min(offset + FnMinus2, n - 1);//----> Индекс

        if (Table[i].key < key)//----> Поиск в правой части массива
        {
            Fn = FnMinus1;
            FnMinus1 = FnMinus2;
            FnMinus2 = Fn - FnMinus1;
            offset = i;
        }
        else if (Table[i].key > key)//----> Поиск в левой части массива
        {
            Fn = FnMinus2;
            FnMinus1 = FnMinus1 - FnMinus2;
            FnMinus2 = Fn - FnMinus1;
        }
        else
        {
            //----> Постусловие: Найдена позиция с данным ключом
            return i;
        }
    }
    if (FnMinus1 == 1 && Table[offset + 1].key == key)
    {
        return offset + 1;
    }
    //----> Постусловие: Позиция с заданным ключом не найдена
    return -1;
}

//----> Считывание записи в файле по заданной ссылке
void Reading(const string& file, int offset, InsurancePolicy& policy)
{
    //----> Предусловие: Должен существовать файл и быть доступным для чтения

    ifstream binaryFileIn(file, ios::binary);
    if (!binaryFileIn)
    {
        cout << endl << "Бинарный файл не удалось открыть для чтения +_+" << endl;
        exit(1);
    }
    cout << endl << "Бинарный файл успешно открыт для чтения ^_^" << endl << endl;

    binaryFileIn.seekg(offset * sizeof(InsurancePolicy), ios::beg);//---->
Перемещение указателя файла к заданному смещению (относительно начала
последовательности)

    if (binaryFileIn.read(reinterpret_cast<char*>(&policy),
sizeof(InsurancePolicy)))
    {
```

```
        if (policy.number != 0)
        {
            cout << "   Номер полиса ---> " << policy.number << endl;
            cout << "   Компания ---> " << policy.company << endl;
            cout << "   Фамилия владельца ---> " << policy.surname << endl;
        }

        //----> Постусловие: Информация о полисе будет выведена на экран, файл
закроется
    }
    else
    {
        cout << endl << "Бинарный файл не удалось прочитать +_+" << endl;

        //----> Постусловие: Будет выведена информация об ошибке, файл закроется
    }
    binaryFileIn.close();
}

int main()
{
    setlocale(LC_ALL, "ru");
    srand(static_cast<unsigned int>(time(nullptr)));

    ofstream textFile("C:\\Users\\efr-p\\Desktop\\clean\\SiAOD\\practice
2\\textFile.txt");//----> Текстовый файл
    if (!textFile)
    {
        cout << "Текстовый файл не удалось открыть для записи +_+" << endl;
        return 1;
    }
    cout << "Текстовый файл успешно открыт для записи ^_^" << endl << endl;

    //----> Открываем файл для записи в бинарном режиме (флаг - ios::binary)
    ofstream binaryFileOut("C:\\Users\\efr-p\\Desktop\\clean\\SiAOD\\practice
2\\binaryFile.bin", ios::binary);//----> Бинарный файл
    if (!binaryFileOut)
    {
        cerr << "Бинарный файл не удалось открыть для записи +_+" << endl;
        return 1;
    }
    cout << "Бинарный файл успешно открыт для записи ^_^" << endl << endl;

    int quant = 100;//----> Количество записей
    set<int> unique;

    InsurancePolicy policy;
    for (int i = 0; i < quant; ++i)
    {
        do {
            policy.number = Randomizer();
        } while (unique.count(policy.number) > 0);
        unique.insert(policy.number);
        policy.company = "Компания_" + to_string(i);
        policy.surname = "Фамилия_" + to_string(i);

        textFile << policy.number << ' ' << policy.company << ' ' <<
policy.surname << '\n';

        binaryFileOut.write(reinterpret_cast<const char*>(&policy),
sizeof(policy));
    }
    cout << "Генерация текстового и бинарного файла завершена ^_^" << endl;

    size_t bytes = sizeof(InsurancePolicy);
```

```
    cout << "Размер записи в байтах: " << bytes << endl;
    cout << "Количество записей: " << quant << endl << endl;

    textFile.close();
    binaryFileOut.close();

    //----> ТРЕТЬЕ ЗАДАНИЕ

    const string file = "C:\\Users\\efr-p\\Desktop\\clean\\SiAOD\\practice
2\\binaryFile.bin";

    //----> Создание таблицы
    vector<TableContent> Table;
    cout << "Начинается создание таблицы данных UwU" << endl << endl;
    ifstream binaryFileIn(file, ios::binary);
    if (!binaryFileIn)
    {
        cout << "Бинарный файл не удалось открыть для чтения +_+" << endl;
        return 1;
    }
    cout << "Бинарный файл успешно открыт для чтения ^_^" << endl << endl;

    int offset = 0;

    while (binaryFileIn.read(reinterpret_cast<char*>(&policy), sizeof(policy)))
    {
        TableContent content;
        content.key = policy.number;//----> Генерация ключа (номера полиса)
        content.fileOffset = offset;
        Table.push_back(content);
        offset += 1;
    }
    cout << "Таблица успешно создана UwU" << endl << endl;
    binaryFileIn.close();

    //----> Сортировка таблицы
    sort(Table.begin(), Table.end(), [](const TableContent& a, const TableContent&
b) {return a.key < b.key; });
    cout << "Таблица успешно отсортирована UwU" << endl << endl;

    //----> Поиск по таблице
    cout << "   Добро пожаловать в информационную базу СТРАХпол!" << endl << "
Здесь вы можете проверить действительность страхового полиса юр.лица $_$" << endl
<< endl;
    unsigned long int key;
    cout << "   Введите номер страхового полиса (8 чисел): ";
    cin >> key;

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

    int result = Fibonacci(Table, key);

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

    if (result != -1)
    {
        //----> Получена ссылка на запись в файле
        cout << endl << "   Полис с номером " << key << " найден в позиции " <<
result << endl;

        //----> Считываем запись из файла по ссылке
        Reading(file, Table[result].fileOffset, policy);
    }
    else
    {
```

```
        cout << endl << "   Полис с номером " << key << " не найден" << endl;
    }

    auto duration = chrono::duration_cast<chrono::microseconds>(end_time -
start_time);
    cout << "   Время выполнения: " << duration.count() << " [мкс]" << endl;

    return 0;
}
```

2.3.4. Тестирование

```
Текстовый файл успешно открыт для записи ^_^

Бинарный файл успешно открыт для записи ^_^

Генерация текстового и бинарного файла завершена ^_^
Размер записи в байтах: 88
Количество записей: 100

Начинается создание таблицы данных UwU

Бинарный файл успешно открыт для чтения ^_^

Таблица успешно создана UwU

Таблица успешно отсортирована UwU

   Добро пожаловать в информационную базу СТРАХпол!
   Здесь вы можете проверить действительность страхового полиса юр.лица $_$

   Введите номер страхового полиса (8 чисел): 10029663

   Полис с номером 10029663 найден в позиции 87

Бинарный файл успешно открыт для чтения ^_^

   Номер полиса ---> 10029663
   Компания ---> Компания_50
   Фамилия владельца ---> Фамилия_50
   Время выполнения: 1300 [нс]
```

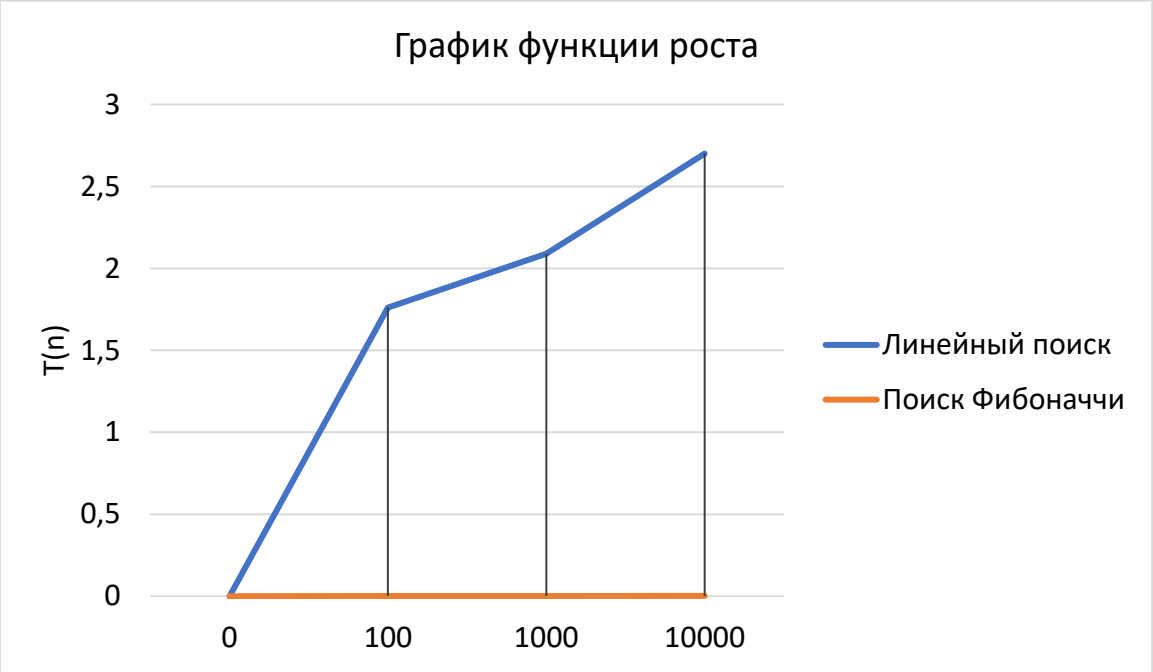
```
Текстовый файл успешно открыт для записи ^_^

Бинарный файл успешно открыт для записи ^_^

Генерация текстового и бинарного файла завершена ^_^
Размер записи в байтах: 88
Количество записей: 1000

Начинается создание таблицы данных UwU

Бинарный файл успешно открыт для чтения ^_^

Таблица успешно создана UwU

Таблица успешно отсортирована UwU

   Добро пожаловать в информационную базу СТРАХпол!
   Здесь вы можете проверить действительность страхового полиса юр.лица $_$

   Введите номер страхового полиса (8 чисел): 10024998

   Полис с номером 10024998 найден в позиции 779

Бинарный файл успешно открыт для чтения ^_^

   Номер полиса ---> 10024998
   Компания ---> Компания_500
   Фамилия владельца ---> Фамилия_500
   Время выполнения: 1300 [нс]
```

```
Текстовый файл успешно открыт для записи ^_^

Бинарный файл успешно открыт для записи ^_^

Генерация текстового и бинарного файла завершена ^_^
Размер записи в байтах: 88
Количество записей: 10000

Начинается создание таблицы данных UwU

Бинарный файл успешно открыт для чтения ^_^

Таблица успешно создана UwU

Таблица успешно отсортирована UwU

   Добро пожаловать в информационную базу СТРАХпол!
   Здесь вы можете проверить действительность страхового полиса юр.лица $_$

   Введите номер страхового полиса (8 чисел): 10010546

   Полис с номером 10010546 найден в позиции 3198

Бинарный файл успешно открыт для чтения ^_^

   Номер полиса ---> 10010546
   Компания ---> Компания_5000
   Фамилия владельца ---> Фамилия_5000
   Время выполнения: 1900 [нс]
```

2.3.5. Практическая оценка времени выполнения

Количество записей	Время выполнения [нс]/[мс]	Размер записи [Байт]
100	1300 / 0,0013	88
1 000	1300 / 0,0013	
10 000	1900 / 0,0019	

2.4. Анализ эффективности алгоритмов поиска в файле



Временная сложность:

Линейный поиск имеет временную сложность **O(n)**, где n - количество элементов в массиве. Это означает, что время выполнения линейного поиска линейно зависит от размера данных.

Поиск Фибоначчи имеет временную сложность **O(log n)**, где n - количество элементов в массиве. Это делает его ***более эффективным*** для больших наборов данных.

Отсортированность данных:

Линейный поиск не требует отсортированных данных и может быть использован для поиска в неупорядоченных списках.

Поиск Фибоначчи требует, чтобы данные были упорядочены по ключу, поскольку он использует бинарный поиск (дробление на половину на каждом шаге новой итерации).

Структура данных:

Линейный поиск прост и может быть использован для любой структуры данных, включая массивы, списки и т. д.

Поиск Фибоначчи, как правило, используется для поиска в упорядоченных массивах.

Сложность реализации:

Линейный поиск очень прост в реализации и требует минимума дополнительных вычислений.

Поиск Фибоначчи более сложен в реализации из-за необходимости поддерживать последовательность чисел Фибоначчи и проверять элементы в массиве на каждой итерации.

Итог:

Линейный поиск обычно подходит для небольших наборов данных или когда данные не отсортированы, а поиск Фибоначчи может быть более эффективным для больших объемов данных и данных, упорядоченных по ключу. Выбор между ними зависит от конкретных требований задачи и характеристик данных.

3. ВЫВОД

В рамках данного задания, целью было получение практического опыта в применении алгоритмов поиска в таблицах данных, используя двоичные файлы. Задание предполагало разработку программы для поиска записей с заданным ключом в таких файловых структурах, применяя различные алгоритмы поиска.

Программа была успешно реализована, включая открытие и чтение бинарного файла, а также применение линейного алгоритма поиска для поиска записей по заданному ключу. Каждый этап программы был задокументирован, включая предусловия и постусловия для функций, а также измерение времени выполнения операции поиска.

Это практическое задание позволило на практике изучить принципы организации и работы с бинарными файлами, а также применить алгоритмы поиска для поиска записей с заданным ключом. Этот опыт может быть полезным при разработке приложений, которые требуют эффективного доступа и поиска данных в больших объемах информации, например, баз данных или файловых хранилищ.