



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 №5.2

**Тема: Алгоритмы поиска в таблице при работе с данными из файла**

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент  
Группа

Зуев Д.А.  
ИКБО-68-23

**Москва 2025**

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

## ЗАДАНИЕ 1

**Формулировка задачи 1:** Разработать программу поиска записей с заданным ключом в двоичном файле с применением различных алгоритмов. 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. Рекомендация: создайте сначала текстовый файл, а затем преобразуйте его в двоичный.

Персональный вариант:

№	Поиск	Структура записи
13	Бинарный однородный с использованием таблицы смещений	Студент: номер зачетной книжки, номер группы, ФИО

### Определение размера записи в байтах:

В коде структура Student содержит два поля: Keys и Info. Для определения размера записи в байтах, нужно учитывать размер каждого поля.

- Keys: это int, который занимает 4 байта.
- info: это строка, длина которой может варьироваться. В двоичном файле мы сначала записываем длину строки (размер size\_t, который занимает 8 байт), а затем сами символы строки.

Пример расчета:

Для строки info длиной 10 символов размер записи будет:

4 байта (для keys) + 8 байт (для длины строки) + 10 байт (для символов строки)  
= 22 байта.

### Организация прямого доступа к записям в бинарном файле:

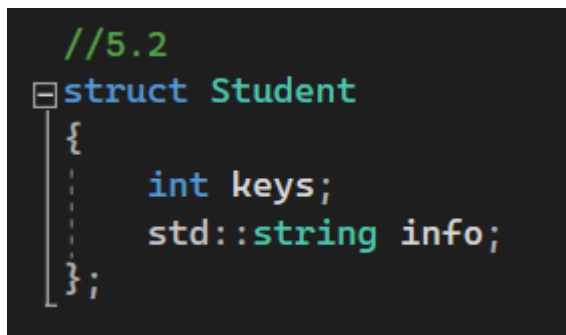
Прямой доступ к записям в бинарном файле означает, что мы можем получить доступ к конкретной записи, зная ее позицию (индекс) в файле. Это достигается с помощью функции `seekg`, которая позволяет перемещать указатель чтения на нужную позицию.

### Перечисление алгоритмов, реализованных в форме функций:

В коде реализованы следующие функции (алгоритмы):


1. **Create\_txt\_file**: Создает текстовый файл с уникальными записями, генерируя случайные ключи и названия городов.
2. **Create\_binary\_file**: Переводит текстовый файл в двоичный формат, записывая ключи и названия городов.
3. **Read\_binary\_file**: Читает данные из двоичного файла и выводит их на консоль (для проверки корректности чтения и перевода данных).

### Код программы



```
//5.2
struct Student
{
    int keys;
    std::string info;
};
```

>Рисунок 1 – реализация структуры для записи данных



```
void Create_txt_file(const std::string& filename, int& n)
{
    std::ofstream file(filename);
    std::set<int> keys;
    int key;

    std::cout << "Filling in the file" << "\n";

    for (auto i = 0; i < n; i++)
    {
        do
        {
            key = std::rand() % 10000;
        } while (keys.find(key) != keys.end());

        keys.insert(key);
        std::string nm = std::to_string(key);
        std::string fio = " surname" + std::to_string(key) + " name" + std::to_string(key) + " " + std::to_string(key);
        file << key << " " << nm << fio << "\n";
    }
    file.close();
}
```

Рисунок 2 – реализация функции create\_txt\_file для создания текстового файла

```
1758 void Create_binary_file(std::string& txt_filename, std::string& binary_filename)
1759 {
1760     std::ofstream out_file(binary_filename, std::ios::binary);
1761     std::ifstream in_file(txt_filename);
1762
1763     Student entry;
1764
1765     while (in_file >> entry.keys)
1766     {
1767         in_file.ignore();
1768         std::getline(in_file, entry.info);
1769
1770         out_file.write(reinterpret_cast<char*>(&entry.keys), sizeof(entry.keys));
1771         size_t info_size = entry.info.size();
1772         out_file.write(reinterpret_cast<char*>(&info_size), sizeof(info_size));
1773         out_file.write(entry.info.c_str(), info_size);
1774     }
1775     out_file.close();
1776     in_file.close();
1777 }
```

Рисунок 3 – реализация функции Create\_binary\_file для перевода в двоичный файл

```
1779 void Read_binary_file(std::string& binary_filename)
1780 {
1781     std::ifstream in_file(binary_filename, std::ios::binary);
1782
1783     Student entry;
1784
1785     while (true)
1786     {
1787         if (!in_file.read(reinterpret_cast<char*>(&entry.keys), sizeof(entry.keys)))
1788         {
1789             break;
1790         }
1791         size_t info_size;
1792         in_file.read(reinterpret_cast<char*>(&info_size), sizeof(info_size));
1793
1794         entry.info.resize(info_size);
1795         in_file.read(&entry.info[0], info_size);
1796
1797         std::cout << entry.keys << "\t" << entry.info << "\n";
1798     }
1799 }
1800 }
```

Рисунок 4 – реализация функции read\_binary\_file для чтения и вывода данных из двоичного файла

## Результаты тестирования

Проведем тестирование программы для 10 записей:

```
Filling in the file
41      41 surname41 name41 41
8467    8467 surname8467 name8467 8467
6334    6334 surname6334 name6334 6334
6500    6500 surname6500 name6500 6500
9169    9169 surname9169 name9169 9169
5724    5724 surname5724 name5724 5724
1478    1478 surname1478 name1478 1478
9358    9358 surname9358 name9358 9358
6962    6962 surname6962 name6962 6962
4464    4464 surname4464 name4464 4464
```

Рисунок 5 – тестирование создания файлов для 10 записей

Тестирование показало, что программа работает корректно.

## ЗАДАНИЕ 2

**Формулировка задачи 2:** Разработать программу поиска записи по ключу в бинарном файле с применением алгоритма линейного поиска.

### Алгоритм линейного поиска на псевдокоде:

Функция Search(файл, ключ):

    Открыть файл для чтения

    Пока не достигнут конец файла:

        Прочитать уникальный ключ из файла

        Прочитать длину названия города

        Прочитать название города

        Если уникальный ключ равен ключу:

            Закрыть файл

            Вернуть название города

    Закрыть файл

    Вернуть "Город не найден"

### Код программы

```

std::string Search(std::string& binary_filename, int search_key)
{
    std::ifstream in_file(binary_filename, std::ios::binary);

    Student entry;

    while (true)
    {
        if (!in_file.read(reinterpret_cast<char*>(&entry.keys), sizeof(entry.keys)))
        {
            break;
        }

        size_t info_size;
        in_file.read(reinterpret_cast<char*>(&info_size), sizeof(info_size));

        entry.info.resize(info_size);
        in_file.read(&entry.info[0], info_size);
        if (entry.keys == search_key)
        {
            return entry.info;
        }
    }

    return "NO result";
}

```

Рисунок 6 – реализация функции для линейного поиска по ключу

### Результаты тестирования

Проведем тестирование и замер времени для различного количества записей:

Количество записей	Время(с)
100	0.2545
1000	0.7347
10000	5.4751

Тестирование показало, что программа работает корректно.

### ЗАДАНИЕ 3

**Формулировка задачи 3:** Поиск записи в файле с применением дополнительной структуры данных, сформированной в оперативной памяти.

1. Для оптимизации поиска в файле создать в оперативной памяти структуру 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.

#### **Описание алгоритма доступа к записи в файле посредством таблицы**

Ссылка в таблице (или индексная запись) определяет положение (смещение) записи в двоичном файле. Это позволяет быстро находить записи без необходимости последовательного чтения всего файла. Вместо этого мы можем перейти непосредственно к нужному месту в файле, используя смещение, хранящееся в индексной таблице. Средства C++, которые используются для организации доступа к записи в файле по ссылке:

1. `streampos`: Это тип данных, который используется для хранения позиции в потоке (файле). Он позволяет указать, где в файле находится запись.
2. `seekg`: Метод, который используется для перемещения указателя чтения в потоке на определенное смещение (позицию) в файле.
3. `read`: Метод, который используется для чтения данных из файла, начиная с текущей позиции указателя.

#### **Алгоритм**

Сначала отсортируем подготовим нашу таблицу. Нам надо её отсортировать, для этого воспользуемся сортировкой пузырьком. После создадим таблицу смещения для бинарного поиска, а после

## Код программы

```
1833 void Create_binary_file_with_table(std::string& txt_filename, std::string& binary_filename, std::vector<Index_Entry>& table)
1834 {
1835     std::ofstream out_file(binary_filename, std::ios::binary);
1836     std::ifstream in_file(txt_filename);
1837
1838     Student entry;
1839
1840     while (in_file >> entry.keys)
1841     {
1842         in_file.ignore();
1843         std::getline(in_file, entry.info);
1844
1845         std::streampos pos = out_file.tellp();
1846         out_file.write(reinterpret_cast<char*>(&entry.keys), sizeof(entry.keys));
1847         size_t info_size = entry.info.size();
1848         out_file.write(reinterpret_cast<char*>(&info_size), sizeof(info_size));
1849         out_file.write(entry.info.c_str(), info_size);
1850
1851         table.push_back({ entry.keys, pos });
1852     }
1853     out_file.close();
1854     in_file.close();
1855 }
```

Рисунок 8 – реализация функции Create\_binary\_file с добавлением строки для записи в индексную таблицу

```
1857 int Binary_search_with_table(std::vector<Index_Entry>& table, int search_key)
1858 {
1859     int n = table.size();
1860     //Сортируем массив
1861     for (int i = 0; i < n; i++)
1862     {
1863         bool ind = false;
1864         for (int j = 0; j < n - i - 1; j++)
1865         {
1866             if (table[j].keys > table[j + 1].keys)
1867             {
1868                 std::swap(table[j], table[j + 1]);
1869                 ind = true;
1870             }
1871         }
1872         if (ind == false)
1873         {
1874             break;
1875         }
1876     }
1877
1878     // Функция для построения таблицы смещений
1879     std::vector<int> shiftTable(10001, -1); // Здесь предполагается, что значения в массиве лежат в диапазоне [0, 100]
1880     for (int i = 0; i < n; i++) {
1881         if (shiftTable[table[i].keys] == -1) {
1882             shiftTable[table[i].keys] = i;
1883         }
1884     }
1885
1886     // Функция бинарного поиска с использованием таблицы смещений
1887     int left = 0, right = n - 1;
1888     while (left <= right) {
1889         int mid = left + (right - left) / 2;
1890         if (table[mid].keys == search_key) {
1891             return mid;
1892         }
1893         else if (table[mid].keys < search_key) {
1894             left = mid + 1;
1895         }
1896         else {
1897             right = mid - 1;
1898         }
1899     }
1900     return shiftTable[search_key]; // Если элемент найден, возвращаем индекс -1
1901 }
1902 }
```

Рисунок 9 – реализация функции Binary\_search\_with\_table для поиска



```

1903 Student Read_position(std::string& binary_filename, std::streampos position)
1904 {
1905     std::ifstream in_file(binary_filename, std::ios::binary);
1906     Student entry;
1907
1908     in_file.seekg(position);
1909     in_file.read(reinterpret_cast<char*>(&entry.keys), sizeof(entry.keys));
1910
1911     size_t info_size;
1912     in_file.read(reinterpret_cast<char*>(&info_size), sizeof(info_size));
1913     entry.info.resize(info_size);
1914     in_file.read(&entry.info[0], info_size);
1915     in_file.close();
1916     return entry;
1917 }

```

Рисунок 10 – реализация функции read\_position для чтения записи по ссылке

```

// Основная функция
int main() {
    setlocale(LC_ALL, "rus");
    string textFilename = "cities.txt"; // Имя текстового файла
    string binaryFilename = "cities.bin"; // Имя двоичного файла
    int recordCount;

    cout << "Введите количество записей для создания в файле: ";
    cin >> recordCount;

    createTextFile(textFilename, recordCount);
    vector<IndexEntry> indexTable; // Индексная таблица
    convertToBinary(textFilename, binaryFilename, indexTable);

    // Сортируем индексную таблицу по уникальному ключу
    sort(indexTable.begin(), indexTable.end(), [](const IndexEntry& a, const IndexEntry& b) {
        return a.uniqueKey < b.uniqueKey;
    });

    cout << "Данные успешно записаны в двоичный файл." << endl;

    // Проверка и вывод данных из двоичного файла
    readAndVerifyBinary(binaryFilename);

    // Поиск города по коду
    int searchKey;
    cout << "Введите код города для поиска: ";
    cin >> searchKey;

    // Замер времени для поиска Фибоначчи
    auto start = chrono::high_resolution_clock::now(); // Начало замера времени
    int index = fibonacciSearch(indexTable, searchKey);
    auto end = chrono::high_resolution_clock::now(); // Конец замера времени

    if (index != -1) {
        CityRecord foundRecord = readRecordAtPosition(binaryFilename, indexTable[index].filePosition);
        cout << "Результат поиска (Поиск Фибоначчи): " << foundRecord.cityName << endl;
    }
    else {
        cout << "Город не найден (Поиск Фибоначчи)" << endl;
    }

    // Вычисление и вывод времени выполнения
    chrono::duration<double, milli> duration = end - start; // Время в миллисекундах
    cout << "Время выполнения поиска Фибоначчи: " << duration.count() << " мс" << endl;

    return 0;
}

```

Рисунок 11 – реализация функции основной функции программы

## Результаты тестирования

Проведем тестирование и замер времени для различного количества записей:

Количество записей	Время(с)
100	0.0014
1000	0.0016
10000	0.0022

## Анализ

**Линейный поиск** — это простой алгоритм, который проходит через каждую запись в файле последовательно, сравнивая ключи, пока не найдет нужный или не достигнет конца файла.

Сложность:

- Временная сложность:  $O(n)$ , где  $n$  — количество записей в файле. В худшем случае алгоритм должен просмотреть все записи.
- Пространственная сложность:  $O(1)$ , так как алгоритм не требует дополнительной памяти для хранения данных.

Преимущества:

- Простота реализации.
- Не требует предварительной сортировки данных.

Недостатки:

- Низкая эффективность для больших объемов данных, так как требует линейного прохода.

**Бинарный поиск** использует индексную таблицу, которая содержит уникальные ключи и ссылки на позиции записей в файле.

Сложность:

- Временная сложность:  $O(\log n)$  в среднем случае, так как алгоритм использует деление и сокращение диапазона поиска.
- Пространственная сложность:  $O(n)$  для хранения индексной таблицы.

Преимущества:

- Более высокая эффективность по сравнению с линейным поиском для больших объемов данных.
- Позволяет быстро находить записи, используя ссылки на позиции в файле.

#### Недостатки:

- Требуется предварительной сортировки данных и создания индексной таблицы, что увеличивает время на подготовку.
- Сложнее в реализации по сравнению с линейным поиском.

## **ВЫВОД**

В ходе выполнения практической работы получен практический опыт по применению алгоритмов поиска в таблицах данных. Разработана программа поиска записей с заданным ключом в двоичном файле с применением различных алгоритмов. Реализован поиск в файле с применением линейного поиска, а также Бинарный поиск записи в файле с применением дополнительной структуры данных, сформированной в оперативной памяти.

Бинарный поиск является более эффективным методом для поиска записей в больших объемах данных, особенно когда данные уже отсортированы и существует возможность создания индексной таблицы. Линейный поиск, хотя и проще в реализации, не подходит для больших наборов данных из-за своей линейной временной сложности. Выбор метода поиска зависит от конкретных требований к производительности и структуре данных.