|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 № 9** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Поиск записей в файле»** | |
|  | |
| Выполнил студент группы ИКБО-11-22 | Орлов К.Б. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2023

# **Цель работы**

* Получить практический опыт по применению поиска в таблицах данных

# **Постановка задачи**

1. Разработать программу генерации двоичного файла из записей (структура записи определена вариантом). Поле ключа записи в задании варианта подчеркнуто. Файл заполнять данными, используя для датчик псевдослучайных чисел. Ключи записей в файле уникальны.
   1. Для удобства дальнейшей работы в программе выводить количество записанных записей, байт и последнюю запись.
2. Разработать программу поиска записи по ключу в бинарном файле с применением алгоритма линейного поиска.
   1. Провести практическую оценку времени выполнения поиска последней записи в файле объемом 100, 1000, 10 000 и более записей.
   2. Составить таблицу с указанием результатов замера времени.
3. Для оптимизации поиска в файле, разработать программу, создающую в оперативной памяти дополнительную структуру данных, содержащую ключ и ссылку (смещение) на запись в файле.
   1. Разработать функцию, которая принимает на вход ключ, ищет в таблице элемент, соответствующий ключу, и возвращает ссылку на запись в файле. Алгоритм поиска определен в варианте.
   2. Разработать функцию, которая принимает ссылку на запись в файле, считывает ее, применяя механизм прямого доступа к записям файла. И возвращает прочитанную запись как результат.
   3. Провести практическую оценку времени выполнения поиска последней записи в файле объемом 100, 1000, 10 000 и более записей.
   4. Составить таблицу с указанием результатов замера времени. Отдельно учесть время, затраченное на создание дополнительной структуры.
4. Провести анализ эффективности разработанного алгоритма поиска по сравнению с линейным.

Персональный вариант №4. Условие задания:

|  |  |
| --- | --- |
| Алгоритм поиска | Структура записи файла (ключ – подчеркнутое поле) |
| Фибоначчи поиск | Владельцев автомобилей: номер машины – целое число, марка, ФИО владельца |

# **Решение**

# **Вспомогательные функции.**

# **Функция printRecord.**

Функция нужна для вывода значений одной записи. Получает по ссылкам экземпляр структуры Record и экземпляр класса std::string.

|  |
| --- |
| *void* *printRecord*(*const* Record *&*record, std::string *&*label)  {  std::*cout* *<<* label *<<* ": {номер машины: " *<<* record.*\_carId*  *<<* ", бренд машины: " *<<* record.*\_carBrand*  *<<* ", фамилия: " *<<* record.*\_lastName*  *<<* ", имя: " *<<* record.*\_firstName*  *<<* ", отчество: " *<<* record.*\_patronymic* *<<* "}" *<<* '\n';  } |

# **Функция print\_br\_for\_ls.**

Функция используется для вывода строчек таблицы времени для линейного поиска.

|  |
| --- |
| *void* *print\_br\_for\_ls*()  {  *for* (*size\_t* *i* *=* 0; *i* *<* 3 *+* 15 *+* 4 *+* 15 *+* 3; *++i*)  std::*cout* *<<* '=';  std::*cout* *<<* '\n';  } |

# **Функция print\_br\_for\_fs.**

Функция используется для вывода строчек таблицы времени для линейного поиска.

|  |
| --- |
| *int* *\*createRandArrBySize*(*const* *size\_t* *&*size)  {  *int* *\*res* *=* *new* *int*[size];  std::*srand*(std::*time*(nullptr));  *for* (*int* *i* *=* 0; *i* *<* size; *++i*)  *res*[*i*] *=* std::*rand*() *%* 1000000 *+* 1000;  *return* *res*;  } |

# **Функция printMenuForLS.**

Функция используется для вывода шапки таблицы времени выполнения линейного поиска.

|  |
| --- |
| *void* *printMenuForLS*(std::string typeOfSearch)  {  std::*cout* *<<* typeOfSearch *<<* '\n';  *print\_br\_for\_ls*();  std::*cout* *<<* "|| " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* "SIZE"  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* "TIME, s"  *<<* " ||\n";  *print\_br\_for\_ls*();  } |

# **Функция printMenuForFS.**

Функция используется для вывода шапки таблицы времени выполнения поиска Фибоначчи.

|  |
| --- |
| *void* *printMenuForFS*(std::string typeOfSearch)  {  std::*cout* *<<* typeOfSearch *<<* '\n';  *print\_br\_for\_fs*();  std::*cout* *<<* "|| " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* "SIZE"  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(30) *<<* "TIME for creating struct, s"  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(30) *<<* "TIME for search, s"  *<<* " ||\n";  *print\_br\_for\_fs*();  } |

# **Функция timeForLastRecordWithLS.**

Функция timeForLastRecordWithLS используется для произведения измерений по поиску последней записи с помощью алгоритма линейного поиска. На вход принимает объект файлового потока считывания и количество записей в файле. Для поиска последней записи даем на вход ключ, который точно не будет существовать в таблице. Функция измеряет время в наносекундах и приводит результат к секундам.

|  |
| --- |
| *void* *timeForLastRecordWithLS*(std::ifstream *&*fin, *const* *size\_t* *&*countOfRecords)  {  fin.*seekg*(0);  *auto* *begin* *=* std::chrono::steady\_clock::*now*();  *auto* *res* *=* *liniearSearch*(0, fin, countOfRecords);  *auto* *end* *=* std::chrono::steady\_clock::*now*();  *auto* *delta* *=* std::chrono::*duration\_cast*<std::chrono::nanoseconds>(*end* *-* *begin*).*count*();  std::*cout* *<<* "|| " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* countOfRecords  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* std::*fixed* *<<* std::*setprecision*(6) *<<* *double*(*delta*) */* 1000000000  *<<* " ||\n";  *print\_br\_for\_ls*();  } |

# **timeForLastRecordWithFS.**

Функция timeForLastRecordWithFS используется для произведения измерений по поиску последней записи с помощью алгоритма поиска Фибоначчи. На вход принимает объект файлового потока считывания и количество записей в файле. Для поиска последней записи даем на вход ключ, который точно не будет существовать в таблице. Функция измеряет время в наносекундах и приводит результат к секундам. Отдельно замеряет время на создание и сортировку таблицы смещений и время на сам поиск.

|  |
| --- |
| *void* *timeForLastRecordWithFS*(std::ifstream *&*fin, *const* *size\_t* *&*countOfRecords)  {  RecordSearcher *\*rc* *=* nullptr;  *auto* *beg* *=* std::chrono::steady\_clock::*now*();  *createTable*(*rc*, countOfRecords, fin);  *sortTable*(*rc*, countOfRecords);  *auto* *time* *=* std::chrono::*duration\_cast*<std::chrono::nanoseconds>(std::chrono::steady\_clock::*now*() *-* *beg*).*count*();  *auto* *begin* *=* std::chrono::steady\_clock::*now*();  *auto* *res* *=* *fibbonacciSearch*(*rc*, countOfRecords, 0);  *auto* *end* *=* std::chrono::steady\_clock::*now*();  *auto* *delta* *=* std::chrono::*duration\_cast*<std::chrono::nanoseconds>(*end* *-* *begin*).*count*();  std::*cout* *<<* "|| " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* countOfRecords  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(30) *<<* std::*fixed* *<<* std::*setprecision*(6) *<<* *double*(*time*) */* 1000000000  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(30) *<<* std::*fixed* *<<* std::*setprecision*(6) *<<* *double*(*delta*) */* 1000000000  *<<* " ||\n";  *print\_br\_for\_fs*();  *delete[]* *rc*;  }; |

# **Структуры.**

# **Структура записи – Record.**

Структура записи содержит одно поле целого беззнакового типа – carId (ключ, по которому будем искать записи) и четыре поля, которые являются массивами символов на 20 ячеек каждое – \_carBrand, \_firstName, \_lastName, \_patronomyc

|  |
| --- |
| *struct* Record{  *size\_t* *\_carId*;  *char* *\_carBrand*[20];  *char* *\_firstName*[20];  *char* *\_lastName*[20];  *char* *\_patronymic*[20];  }; |

# **Структура таблицы смещений – RecordSearcher.**

Структура таблицы смещений состоит из двух целых беззнаковых полей: \_carId – ключ записи и pos – позиция записи в файле

|  |
| --- |
| *struct* RecordSearcher{  *size\_t* *\_carId*;  *size\_t* *pos*;  }; |

# **Класс генерации уникальной случайной записи – UniqueRecordGenerator**

Класс генерации уникальной случайной записи состоит из нескольких полей. Двух целых беззнаковых: \_idMinVal и \_idMaxVal – минимальное и максимальное значения ключа записи, генератор рандомных чисел Мерсенна – \_mRng, который отвечает за генерацию песвдослучайный чисел, одно поле отип uniform\_int\_distribution - \_mDist, которое отчечает за инициализацию генератора целыми значениями и поля типа unordered\_set<size\_t> – \_setOfUsedId, которое отвечает за хранение использованных id.

|  |
| --- |
| *class* UniqueRecordGenerator  {  *public:*  *UniqueRecordGenerator*(*size\_t* idMinVal *=* 1, *size\_t* idMaxVal *=* *MAX\_SIZE*);  Record *generateUniqueRecord*();  *private:*  *size\_t* *\_idMinVal*;  *size\_t* *\_idMaxVal*;  std::mt19937 *\_mRng*{std::random\_device{}*()*};  std::uniform\_int\_distribution*<size\_t>* *\_mDist*;  std::unordered\_set*<size\_t>* *\_setOfUsedId*;  }; |

# **Метод генерации случайной записи с уникальным ключом.**

Для генерации случайной записи используется метод generateUniqueRecord класса UniqueRecordGenerator. Уникальный ключ записи генерируется с помощью генератора случайных чисел. Сначала мы записываем новое сгенерированное значение в поле \_setOfUsedId и, если вставка в этот контейнер оказалась успешной, то берем из него последнее записанное значение и кладем его в значение ключа. Остальные поля экземпляра структуры записи мы берем из константного набора имен, фамилий и отчеств. Это выбирается путем генерации случайного индекса.

|  |
| --- |
| Record UniqueRecordGenerator::*generateUniqueRecord*()  {  Record *record*;  *while* (*!\_setOfUsedId*.*insert*(*\_mDist(\_mRng)*).*second*)  {  }  *record*.*\_carId* *=* *\**(*\_setOfUsedId*.*cbegin*());  std::*strcpy*(*record*.*\_firstName*, *VEC\_OF\_FIRST\_NAMES[\_mDist(\_mRng)* *%* *VEC\_OF\_FIRST\_NAMES*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_lastName*, *VEC\_OF\_LAST\_NAMES[\_mDist(\_mRng)* *%* *VEC\_OF\_LAST\_NAMES*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_patronymic*, *VEC\_OF\_PATRONOMYCS[\_mDist(\_mRng)* *%* *VEC\_OF\_PATRONOMYCS*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_carBrand*, *VEC\_OF\_CAR\_BRANDS[\_mDist(\_mRng)* *%* *VEC\_OF\_CAR\_BRANDS*.*size*()*]*.*c\_str*());  *return* *record*;  } |

# **Функция generateBINFile.**

Функция generateBINFile используется для генерации бинарных файлов различного размера. На вход принимает объект файлового потока считывания и количество записей в файле.

|  |
| --- |
| *void* *generateBINFile*(std::ofstream *&*fout, *const* *size\_t* *&*countOfRecords)  {  UniqueRecordGenerator *recordGenerator*(1, countOfRecords);  *for* (*size\_t* *i* *=* 0; *i* *<* countOfRecords; *++i*)  {  Record *record* *=* *recordGenerator*.*generateUniqueRecord*();  fout.*write*((*char* *\**)*&record*, *sizeof*(*record*));  }  } |

# **Функция getRecordByPos.**

Функция getRecordByPos была написана для получения записи из файла по ее позиции. Принимает на вход объект файлового потока считывания и позицию записи в файле

|  |
| --- |
| Record *getRecordByPos*(std::ifstream *&*fin, *const* *size\_t* *&*pos)  {  fin.*seekg*((pos *-* 1) *\** *sizeof*(Record));  Record *res*;  fin.*read*(*reinterpret\_cast<char* *\*>*(*&res*), *sizeof*(Record));  *return* *res*;  } |

# **Функция getLastRecord.**

Функция getLastRecord нужна для получения последней записи в файле. Принимает на вход объект файлового потока считывания и возвращает последнию запись в файле

|  |
| --- |
| Record *getLastRecord*(std::ifstream *&*fin)  {  fin.*seekg*(0, std::ios::*end*);  *int* *file\_size* *=* fin.*tellg*();  *int* *num\_records* *=* *file\_size* */* *sizeof*(Record);  *return* *getRecordByPos*(fin, *num\_records*);  } |

# **Линейный поиск.**

Алгоритм линейного поиска – это обычный перебор всех записей в файле по порядку, пока не будет найдено совпадение. Функция linearSearch принимает в качестве параметров ключ записи, объект файлового потока считывания и количество записей.

|  |
| --- |
| Record *liniearSearch*(*const* *size\_t* *&*carId, std::ifstream *&*fin, *const* *size\_t* *&*countOfRecords)  {  fin.*seekg*(0);  Record *record*;  *for* (*size\_t* *i* *=* 0; *i* *<* countOfRecords; *++i*)  {  fin.*read*(*reinterpret\_cast<char* *\*>*(*&record*), *sizeof*(Record));  *if* (*record*.*\_carId* *==* carId)  {  *return* *record*;  }  }  *return* *record*;  } |

# **Поиск методом Фибоначчи.**

Более оптимальным алгоритмом поиска является алгоритм поиска методом Фибоначчи.

Поиск Фибоначчи является модификацией бинарного поиска. Различия в том, что в отличие от бинарного поиска, где мы делим массив пополам, в поиске Фибоначчи рассматриваются промежутки массива, индексами концов которых являются элементы ряда Фибоначчи.

Сначала рассмотрим функции подготавливающие данные к поиску Фибоначчи

# **Функция createTable.**

Функция createTable используется для создания таблицы смещений. Принимает указатель на массив соответствующей структуры по ссылке, количество записей в файле и объект файлового потока считывания.

|  |
| --- |
| *void* *createTable*(RecordSearcher *\*&*recordSearcher, *const* *size\_t* *&*countOfRecords, std::ifstream *&*fin)  {  fin.*seekg*(0);  recordSearcher *=* *new* *RecordSearcher*[countOfRecords];  Record *tmp*;  *for* (*size\_t* *i* *=* 0; *i* *<* countOfRecords; *++i*)  {  fin.*read*(*reinterpret\_cast<char* *\*>*(*&tmp*), *sizeof*(Record));  recordSearcher[*i*].*\_carId* *=* *tmp*.*\_carId*;  recordSearcher[*i*].*pos* *=* *i+*1;  }  } |

# **Функция сортировки.**

В качестве функции сортировки была выбрана сортировка Шелла со смещениями Шелла.

|  |
| --- |
| *void* *sortTable*(RecordSearcher *\*&*recordSearcher, *const* *size\_t* *&*size)  {  *for* (*int* *gap* *=* size */* 2; *gap* *>* 0; *gap* */=* 2)  {  *for* (*int* *i* *=* *gap*; *i* *<* size; *i++*)  {  *int* *j*;  *auto* *temp* *=* recordSearcher[*i*];  *for* (*j* *=* *i*; *j* *>=* *gap* *&&* recordSearcher[*j* *-* *gap*].*\_carId* *>* *temp*.*\_carId*; *j* *-=* *gap*)  {  recordSearcher[*j*] *=* recordSearcher[*j* *-* *gap*];  }  recordSearcher[*j*] *=* *temp*;  }  }  } |

# **Функция fib.**

Функция fib вычисляет значение N–ого элемента ряда Фибоначчи

|  |
| --- |
| *size\_t* *fib*(*size\_t* n)  {  *int* *a* *=* 0, *b* *=* 1;  *for* (*int* *i* *=* 0; *i* *<* n; *i++*)  {  *a* *+=* *b*;  *b* *=* *a-b*;  }  *return* *a*;  } |

# **Основной алгоритм**

Поиск Фибоначчи предельно прост. Сначла мы инициализируем наши начальные переменные, а затем проверяем попали ли мы в нужное число, и если нет, то куда двигаться дальше – левее текущей позиции или правее.

|  |
| --- |
| *size\_t* *fibbonacciSearch*(RecordSearcher *\*&*rc, *const* *size\_t* *&*size, *const* *size\_t* *&*carId)  {  *int* *k* *=* 0, *M*, *i*, *p*, *q*;  *while* (*fib*(*k* *+* 1) *<* (size *+* 1))  *k++*;  *M* *=* *fib*(*k* *+* 1) *-* (size *+* 1);  *i* *=* *fib*(*k*) *-* *M*;  *p* *=* *fib*(*k* *-* 1);  *q* *=* *fib*(*k* *-* 2);  *while* (carId *!=* rc[*i*].*\_carId*)  {  *if* (*i* *<=* 0 *||* carId *>* rc[*i*].*\_carId*)  {  *if* (*p* *==* 1)  *break*;  *else*  {  *i* *+=* *q*;  *p* *-=* *q*;  *q* *-=* *p*;  }  }  *else* *if* (*i* *>=* size *||* carId *<* rc[*i*].*\_carId*)  {  *if* (*q* *==* 0)  *break*;  *else*  {  *int* *tmp* *=* *p*;  *i* *-=* *q*;  *p* *=* *q*;  *q* *=* *tmp* *-* *q*;  }  }  }  *return* *i* *<=* 0 *?* 0 *:* rc[*i*].*pos*;  } |

# **Интерфейс программы**

После запуска программы пользователю предлагается ввести количество записей в рабочем файле. Затем дается выбор из шести опций:

* 0 - Выход из программы
* 1 - Найти запись с помощью линейного поиска
* 2 - Найти запись с помощью поиска Фибоначчи
* 3 - Найти последнюю запись
* 4 - Таблица времени выполнения линейного поиска
* 5 - Таблица времени выполнения поиска Фибоначчи

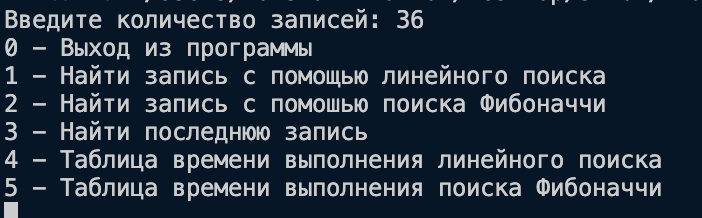


Рисунок 1. Интерфейс программы

# **Тестирование**

# **Тесты и тестовые прогоны**

Тест 1 – тестирование работы функций поиска

Для проверки корректной работы функций поиска, мы найдем последнюю запись в рабочем файле, а затем данные из этой записи внесем в алгоритмы поиска.

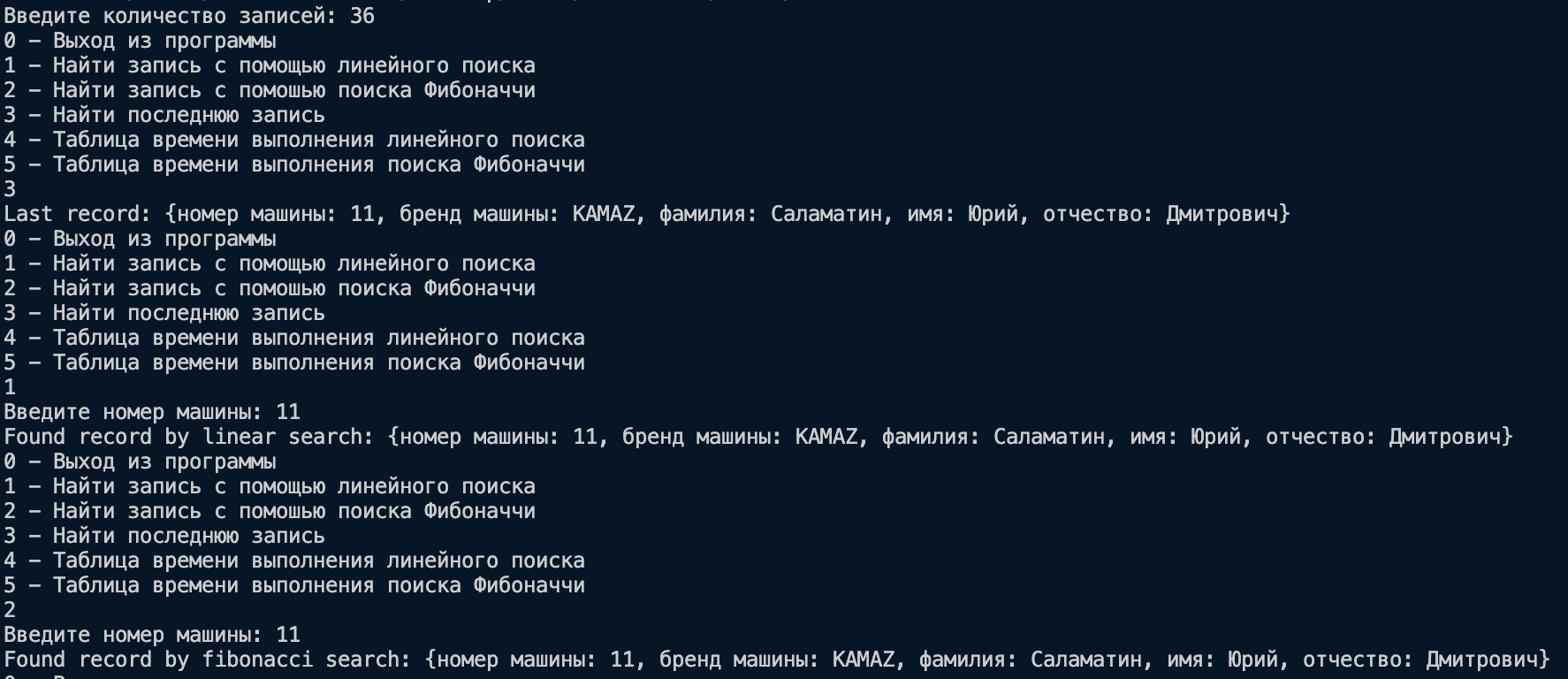


Рисунок 2. Тестирование работы функций поиска.

Все проверяемы функции работают корректно.

Теперь проведем тестовые прогоны алгоритмов поиска.

Воспользуемся опциями 4 и 5 (линейный поиск и поиск Фибоначчи соответственно). Они создают файлы различного размера и отличные от рабочего файла.

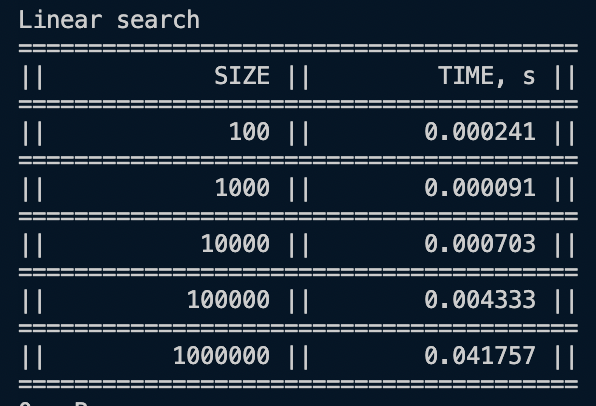


Рисунок 3. Тестовый прогон алгоритма линейного поиска.

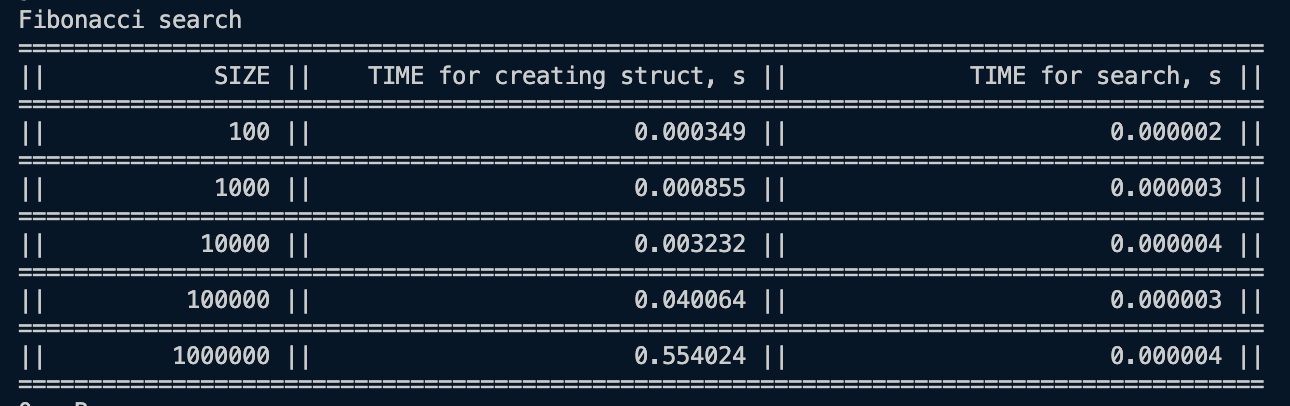


Рисунок 4. Тестовый прогон алгоритма поиска Фибоначчи.

Теперь перенесем полученные данные в таблицы.

Таблица 1. Результаты тестового прогона алгоритма линейного поиска

|  |  |
| --- | --- |
| N | t, с |
| 100 | 0,000241 |
| 1000 | 0,000091 |
| 10000 | 0,000703 |
| 100000 | 0,004333 |
| 1000000 | 0,041757 |

Таблица 2. Результаты тестового прогона алгоритма поиска Фибоначчи

|  |  |  |
| --- | --- | --- |
| N | t (создание структуры), с | t (время поиска), c |
| 100 | 0,000349 | 0,000002 |
| 1000 | 0,000855 | 0,000003 |
| 10000 | 0,003232 | 0,000004 |
| 100000 | 0,040064 | 0,000003 |
| 1000000 | 0,554024 | 0,000004 |

# **Анализ результатов тестовых прогонов**

При сравнении данных с таблицы наглядно видно насколько поиск Фибоначчи превосходит по времени выполнения линейный поиск, однако это только в случае, если брать только время самого поиска, а для него, опять же, потребовалось создать и заполнить отдельную структуру и отсортировать массив, что сделало общее время выполнения гораздо больше, чем у линейного. Таким образом можно сделать вывод, что алгоритм нужно выбирать в зависимости от поставленной задачи и условий, так, например, при работе с большими и частично отсортированными массивами поиск Фибоначчи проявит себя в разы лучше, однако при работе с маленькими массивами или частично отсортированными массивами, скорость выполнения зависит от нескольких условий, таких, как набор входных данных и аппаратура.

# **Вывод**

В результате выполнения работы я:

1. В ходе данной практической работы я получил практический опыт по применению алгоритмов поиска в таблицах данных.

# **Исходный код программы**

# **Листинг 1 – файл main.cpp**

|  |
| --- |
| *#include* "../Headers/UniqueRecordGenerator.hpp"  *#include* "../Headers/RecordSearcher.hpp"  Record *linearSearch*(*const* *size\_t* *&*carId, std::ifstream *&*fin, *const* *size\_t* *&*countOfRecords)  {  fin.*seekg*(0);  Record *record*;  *for* (*size\_t* *i* *=* 0; *i* *<* countOfRecords; *++i*)  {  fin.*read*(*reinterpret\_cast<char* *\*>*(*&record*), *sizeof*(Record));  *if* (*record*.*\_carId* *==* carId)  {  *return* *record*;  }  }  *return* *record*;  }  *size\_t* *fib*(*size\_t* n)  {  *int* *a* *=* 0, *b* *=* 1;  *for* (*int* *i* *=* 0; *i* *<* n; *i++*)  {  *a* *+=* *b*;  *b* *=* *a* *-* *b*;  }  *return* *a*;  }  *size\_t* *fibbonacciSearch*(RecordSearcher *\*&*rc, *const* *size\_t* *&*size, *const* *size\_t* *&*carId)  {  *int* *k* *=* 0, *M*, *i*, *p*, *q*;  *while* (*fib*(*k* *+* 1) *<* (size *+* 1))  *k++*;  *M* *=* *fib*(*k* *+* 1) *-* (size *+* 1);  *i* *=* *fib*(*k*) *-* *M*;  *p* *=* *fib*(*k* *-* 1);  *q* *=* *fib*(*k* *-* 2);  *while* (carId *!=* rc[*i*].*\_carId*)  {  *if* (*i* *<=* 0 *||* carId *>* rc[*i*].*\_carId*)  {  *if* (*p* *==* 1)  *break*;  *else*  {  *i* *+=* *q*;  *p* *-=* *q*;  *q* *-=* *p*;  }  }  *else* *if* (*i* *>=* size *||* carId *<* rc[*i*].*\_carId*)  {  *if* (*q* *==* 0)  *break*;  *else*  {  *int* *tmp* *=* *p*;  *i* *-=* *q*;  *p* *=* *q*;  *q* *=* *tmp* *-* *q*;  }  }  }  *return* *i* *<=* 0 *?* 0 *:* rc[*i*].*pos*;  }  Record *getRecordByPos*(std::ifstream *&*fin, *const* *size\_t* *&*pos)  {  fin.*seekg*((pos *-* 1) *\** *sizeof*(Record));  Record *res*;  fin.*read*(*reinterpret\_cast<char* *\*>*(*&res*), *sizeof*(Record));  *return* *res*;  }  Record *getLastRecord*(std::ifstream *&*fin)  {  fin.*seekg*(0, std::ios::*end*);  *int* *file\_size* *=* fin.*tellg*();  *int* *num\_records* *=* *file\_size* */* *sizeof*(Record);  *return* *getRecordByPos*(fin, *num\_records*);  }  *void* *generateBINFile*(std::ofstream *&*fout, *const* *size\_t* *&*countOfRecords)  {  UniqueRecordGenerator *recordGenerator*(1, countOfRecords);  *for* (*size\_t* *i* *=* 0; *i* *<* countOfRecords; *++i*)  {  Record *record* *=* *recordGenerator*.*generateUniqueRecord*();  fout.*write*((*char* *\**)*&record*, *sizeof*(*record*));  }  }  *void* *print\_br\_for\_ls*()  {  *for* (*size\_t* *i* *=* 0; *i* *<* 3 *+* 15 *+* 4 *+* 15 *+* 3; *++i*)  std::*cout* *<<* '=';  std::*cout* *<<* '\n';  }  *void* *printMenuForLS*(std::string typeOfSearch)  {  std::*cout* *<<* typeOfSearch *<<* '\n';  *print\_br\_for\_ls*();  std::*cout* *<<* "|| " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* "SIZE"  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* "TIME, s"  *<<* " ||\n";  *print\_br\_for\_ls*();  }  *void* *print\_br\_for\_fs*()  {  *for* (*size\_t* *i* *=* 0; *i* *<* 3 *+* 15 *+* 4 *+* 15 *+* 3 *+* 4 *+* 30 *+* 15; *++i*)  std::*cout* *<<* '=';  std::*cout* *<<* '\n';  }  *void* *printMenuForFS*(std::string typeOfSearch)  {  std::*cout* *<<* typeOfSearch *<<* '\n';  *print\_br\_for\_fs*();  std::*cout* *<<* "|| " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* "SIZE"  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(30) *<<* "TIME for creating struct, s"  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(30) *<<* "TIME for search, s"  *<<* " ||\n";  *print\_br\_for\_fs*();  }  *void* *timeForLastRecordWithLS*(std::ifstream *&*fin, *const* *size\_t* *&*countOfRecords)  {  fin.*seekg*(0);  *auto* *begin* *=* std::chrono::steady\_clock::*now*();  *auto* *res* *=* *linearSearch*(0, fin, countOfRecords);  *auto* *end* *=* std::chrono::steady\_clock::*now*();  *auto* *delta* *=* std::chrono::*duration\_cast*<std::chrono::nanoseconds>(*end* *-* *begin*).*count*();  std::*cout* *<<* "|| " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* countOfRecords  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* std::*fixed* *<<* std::*setprecision*(6) *<<* *double*(*delta*) */* 1000000000  *<<* " ||\n";  *print\_br\_for\_ls*();  }  *void* *timeForLastRecordWithFS*(std::ifstream *&*fin, *const* *size\_t* *&*countOfRecords)  {  RecordSearcher *\*rc* *=* nullptr;  *auto* *beg* *=* std::chrono::steady\_clock::*now*();  *createTable*(*rc*, countOfRecords, fin);  *sortTable*(*rc*, countOfRecords);  *auto* *time* *=* std::chrono::*duration\_cast*<std::chrono::nanoseconds>(std::chrono::steady\_clock::*now*() *-* *beg*).*count*();  *auto* *begin* *=* std::chrono::steady\_clock::*now*();  *auto* *res* *=* *fibbonacciSearch*(*rc*, countOfRecords, 0);  *auto* *end* *=* std::chrono::steady\_clock::*now*();  *auto* *delta* *=* std::chrono::*duration\_cast*<std::chrono::nanoseconds>(*end* *-* *begin*).*count*();  std::*cout* *<<* "|| " *<<* std::*setfill*(' ') *<<* std::*setw*(15) *<<* countOfRecords  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(30) *<<* std::*fixed* *<<* std::*setprecision*(6) *<<* *double*(*time*) */* 1000000000  *<<* " || " *<<* std::*setfill*(' ') *<<* std::*setw*(30) *<<* std::*fixed* *<<* std::*setprecision*(6) *<<* *double*(*delta*) */* 1000000000  *<<* " ||\n";  *print\_br\_for\_fs*();  *delete[]* *rc*;  };  *int* *main*()  {  *short* *com*;  *size\_t* *carId*, *pos*, *countOfRec*;  std::string *label*;  Record *record*;  std::*cout* *<<* "Введите количество записей: ";  std::*cin* *>>* *countOfRec*;  std::ofstream *fout*("records.bin", std::ios::*binary*);  *generateBINFile*(*fout*, *countOfRec*);  *fout*.*close*();  std::ifstream *fin*("records.bin", std::ios::*binary*);  *while* (true)  {  std::*cout* *<<* "0 - Выход из программы\n";  std::*cout* *<<* "1 - Найти запись с помощью линейного поиска\n";  std::*cout* *<<* "2 - Найти запись с помошью поиска Фибоначчи\n";  std::*cout* *<<* "3 - Найти последнюю запись\n";  std::*cout* *<<* "4 - Таблица времени выполнения линейного поиска\n";  std::*cout* *<<* "5 - Таблица времени выполнения поиска Фибоначчи\n";  std::*cin* *>>* *com*;  *if* (*!com*)  {  *break*;  }  *else* *if* (*com* *==* 1)  {  std::*cout* *<<* "Введите номер машины: ";  std::*cin* *>>* *carId*;  *record* *=* *linearSearch*(*carId*, *fin*, *countOfRec*);  *if* (*record*.*\_carId* *!=* *carId*)  {  std::*cout* *<<* "В файле нет такого элемента\n";  }  *else*  {  *label* *=* "Found record by linear search";  *printRecord*(*record*, *label*);  }  }  *else* *if* (*com* *==* 2)  {  std::*cout* *<<* "Введите номер машины: ";  std::*cin* *>>* *carId*;  RecordSearcher *\*rc* *=* nullptr;  *createTable*(*rc*, *countOfRec*, *fin*);  *sortTable*(*rc*, *countOfRec*);  *pos* *=* *fibbonacciSearch*(*rc*, *countOfRec*, *carId*);  *if* (*!pos*)  {  std::*cout* *<<* "В файле нет такого элемента\n";  }  *else*  {  *record* *=* *getRecordByPos*(*fin*, *pos*);  *label* *=* "Found record by fibonacci search";  *printRecord*(*record*, *label*);  }  *delete[]* *rc*;  }  *else* *if* (*com* *==* 3)  {  *record* *=* *getLastRecord*(*fin*);  *label* *=* "Last record";  *printRecord*(*record*, *label*);  }  *else* *if* (*com* *==* 4)  {  *printMenuForLS*("Linear search");  *for* (*size\_t* *i* *=* *MIN\_SIZE*; *i* *<=* *MAX\_SIZE*; *i* *\*=* 10)  {  std::ofstream *foutForTables*("recordsForTest.bin", std::ios::*binary*);  *generateBINFile*(*foutForTables*, *i*);  *foutForTables*.*close*();  std::ifstream *finForTables*("recordsForTest.bin", std::ios::*binary*);  *timeForLastRecordWithLS*(*finForTables*, *i*);  *finForTables*.*close*();  }  }  *else* *if* (*com* *==* 5)  {  *printMenuForFS*("Fibonacci search");  *for* (*size\_t* *i* *=* *MIN\_SIZE*; *i* *<=* *MAX\_SIZE*; *i* *\*=* 10)  {  std::ofstream *foutForTables*("recordsForTest.bin", std::ios::*binary*);  *generateBINFile*(*foutForTables*, *i*);  *foutForTables*.*close*();  std::ifstream *finForTables*("recordsForTest.bin", std::ios::*binary*);  *timeForLastRecordWithFS*(*finForTables*, *i*);  *finForTables*.*close*();  }  }  *else*  {  std::*cout* *<<* "ошибка ввода\n";  }  }  *fin*.*close*();  } |

# **Листинг 2 – файл UniqueRecordGenerator.cpp**

|  |
| --- |
| *#include* <../Headers/UniqueRecordGenerator.hpp>  UniqueRecordGenerator::*UniqueRecordGenerator*(*size\_t* idMinVal, *size\_t* idMaxVal)  : *\_idMinVal*(idMinVal),  *\_idMaxVal*(idMaxVal),  *\_mDist*(idMinVal, idMaxVal)  {  }  Record UniqueRecordGenerator::*generateUniqueRecord*()  {  Record *record*;  *while* (*!\_setOfUsedId*.*insert*(*\_mDist(\_mRng)*).*second*)  {  }  *record*.*\_carId* *=* *\**(*\_setOfUsedId*.*cbegin*());  std::*strcpy*(*record*.*\_firstName*, *VEC\_OF\_FIRST\_NAMES[\_mDist(\_mRng)* *%* *VEC\_OF\_FIRST\_NAMES*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_lastName*, *VEC\_OF\_LAST\_NAMES[\_mDist(\_mRng)* *%* *VEC\_OF\_LAST\_NAMES*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_patronymic*, *VEC\_OF\_PATRONOMYCS[\_mDist(\_mRng)* *%* *VEC\_OF\_PATRONOMYCS*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_carBrand*, *VEC\_OF\_CAR\_BRANDS[\_mDist(\_mRng)* *%* *VEC\_OF\_CAR\_BRANDS*.*size*()*]*.*c\_str*());  *return* *record*;  } |

# **Листинг 3 – файл UniqueRecordGenerator.hpp**

|  |
| --- |
| *#ifndef* *UNIQUE\_RECORD\_GENERATOR\_H*  *#define* *UNIQUE\_RECORD\_GENERATOR\_H*  *#include* "Record.hpp"  *#include* <random>  *#include* <unordered\_set>  *constexpr* *size\_t* *MAX\_SIZE* *=* 1000000;  *constexpr* *size\_t* *MIN\_SIZE* *=* 100;  *const* std::vector*<*std::string*>* *VEC\_OF\_FIRST\_NAMES* *=* {"Иван", "Василий", "Семен", "Арсений", "Артем", "Лев", "Макар", "Сергей", "Михаил", "Максим", "Евгений", "Николай", "Юрий", "Илья", "Аристарх", "Виталий", "Егор"};  *const* std::vector*<*std::string*>* *VEC\_OF\_LAST\_NAMES* *=* {"Высоцкий", "Пулеметов", "Мельников", "Кузнцеов", "Чупков", "Малютин", "Волков", "Морозов", "Рылов", "Минаков", "Головин", "Чернов", "Смирнов", "Карпов", "Саламатин"};  *const* std::vector*<*std::string*>* *VEC\_OF\_PATRONOMYCS* *=* {"Артемович", "Иванович", "Львович", "Ильич", "Семнович", "Макарович", "Олегович", "Павлович", "Игоревич", "Юрьевич", "Дмитрович", "Егорович", "Сергеевич"};  *const* std::vector*<*std::string*>* *VEC\_OF\_CAR\_BRANDS* *=* {"BMW", "CHEVROLET", "DODGE", "MINI COOPER", "FERRARI", "LAMBORGINI", "NISSAN", "KIA", "MARK", "MERCEDES", "MAZDA", "FIAT", "KAMAZ", "LIAZ"};  *class* UniqueRecordGenerator  {  *public:*  *UniqueRecordGenerator*(*size\_t* idMinVal *=* 1, *size\_t* idMaxVal *=* *MAX\_SIZE*);  Record *generateUniqueRecord*();  *private:*  *size\_t* *\_idMinVal*;  *size\_t* *\_idMaxVal*;  std::mt19937 *\_mRng*{std::random\_device{}*()*};  std::uniform\_int\_distribution*<size\_t>* *\_mDist*;  std::unordered\_set*<size\_t>* *\_setOfUsedId*;  };  *#endif* |

# **Листинг 4 – файл RecordSearcher.cpp**

|  |
| --- |
| *#include* <../Headers/UniqueRecordGenerator.hpp>  UniqueRecordGenerator::*UniqueRecordGenerator*(*size\_t* idMinVal, *size\_t* idMaxVal)  : *\_idMinVal*(idMinVal),  *\_idMaxVal*(idMaxVal),  *\_mDist*(idMinVal, idMaxVal)  {  }  Record UniqueRecordGenerator::*generateUniqueRecord*()  {  Record *record*;  *while* (*!\_setOfUsedId*.*insert*(*\_mDist(\_mRng)*).*second*)  {  }  *record*.*\_carId* *=* *\**(*\_setOfUsedId*.*cbegin*());  std::*strcpy*(*record*.*\_firstName*, *VEC\_OF\_FIRST\_NAMES[\_mDist(\_mRng)* *%* *VEC\_OF\_FIRST\_NAMES*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_lastName*, *VEC\_OF\_LAST\_NAMES[\_mDist(\_mRng)* *%* *VEC\_OF\_LAST\_NAMES*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_patronymic*, *VEC\_OF\_PATRONOMYCS[\_mDist(\_mRng)* *%* *VEC\_OF\_PATRONOMYCS*.*size*()*]*.*c\_str*());  std::*strcpy*(*record*.*\_carBrand*, *VEC\_OF\_CAR\_BRANDS[\_mDist(\_mRng)* *%* *VEC\_OF\_CAR\_BRANDS*.*size*()*]*.*c\_str*());  *return* *record*;  } |

# **Листинг 5 – файл RecordSearcher.hpp**

|  |
| --- |
| *#ifndef* *RECORD\_SEARCHER\_H*  *#define* *RECORD\_SEARCHER\_H*  *#include* "Record.hpp"  *#include* <fstream>  *#include* <chrono>  *struct* RecordSearcher{  *size\_t* *\_carId*;  *size\_t* *pos*;  };  *void* *createTable*(RecordSearcher *\*&*recordSearcher,*const* *size\_t&* countOfRecords ,std::ifstream *&*fin);  *void* *sortTable*(RecordSearcher *\*&*RecordSearcher, *const* *size\_t&* size);  *#endif* |

# **Листинг 6 – файл Record.cpp**

|  |
| --- |
| *#include* <../Headers/Record.hpp>  *void* *printRecord*(*const* Record *&*record, std::string *&*label)  {  std::*cout* *<<* label *<<* ": {номер машины: " *<<* record.*\_carId*  *<<* ", бренд машины: " *<<* record.*\_carBrand*  *<<* ", фамилия: " *<<* record.*\_lastName*  *<<* ", имя: " *<<* record.*\_firstName*  *<<* ", отчество: " *<<* record.*\_patronymic* *<<* "}" *<<* '\n';  } |

# **Листинг 7 – файл Record.hpp**

|  |
| --- |
| *#pragma* *once*  *#ifndef* *RECORD\_H*  *#define* *RECORD\_H*  *#include* <iostream>  *struct* Record{  *size\_t* *\_carId*;  *char* *\_carBrand*[20];  *char* *\_firstName*[20];  *char* *\_lastName*[20];  *char* *\_patronymic*[20];  };  *void* *printRecord*(*const* Record*&* record, std::string*&* label);  *#endif* |