|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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-08-22 | Сенькевич Г.Д. |
| 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 алгоритмов 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.

Вариант №7. Условие задания:

|  |  |
| --- | --- |
| Упражнение 1 | Бинарный поиск  Регистрация малого предприятия: номер лицензии – целое число, название, учредитель |

# **Решение**

Линейный поиск и бинарный поиск являются двумя наиболее распространенными алгоритмами поиска в массивах или списках данных. В случае двоичных файлов, эти алгоритмы могут использоваться для поиска конкретных значений или записей в файле.

Линейный поиск заключается в том, что мы перебираем все элементы в массиве (или записи в файле) по очереди, начиная с первого, и сравниваем их с искомым значением (или записью). Если элемент (или запись) найден, алгоритм возвращает его позицию в массиве (или файле).

Преимуществом линейного поиска является его простота и универсальность. Однако его сложность составляет O(n), где n - количество элементов в массиве (или записей в файле), что может существенно замедлить поиск в больших файлах.

Бинарный поиск работает иначе. Этот алгоритм предполагает, что элементы в массиве (или записи в файле) упорядочены, то есть отсортированы в определенном порядке (обычно по возрастанию или убыванию). Алгоритм начинает поиск в середине массива (или файла), сравнивая его значение с искомым значением. Если оно равно искомому, алгоритм возвращает его позицию. Если значение элемента в середине меньше искомого, то бинарный поиск продолжает поиск в правой половине массива (или файла), иначе - в левой половине. Алгоритм продолжает делать это до тех пор, пока не найдет искомый элемент (или пока не останется один элемент, который не соответствует искомому).

Преимуществом бинарного поиска является его скорость. В наихудшем случае сложность алгоритма составляет O(log n), где n - количество элементов в массиве (или записей в файле). Таким образом, бинарный поиск значительно быстрее линейного поиска при работе с большими файлами.

Однако для бинарного поиска необходимо, чтобы элементы в массиве (или записи в файле) были упорядочены. Если они не упорядочены, необходимо отсортировать массив (или файл) перед выполнением бинарного поиска, что может занять некоторое время.

В контексте двоичных файлов, бинарный поиск может быть использован для поиска конкретной записи по ключу или другому полю, если записи в файле упорядочены по этому полю. Например, если в двоичном файле хранятся записи о сотрудниках, отсортированные по их идентификатору, то бинарный поиск может быть использован для быстрого поиска записи по идентификатору сотрудника.

Для решения первого упражнения была написана структура specialization, содержащая поля code (код специализации), university\_name (имя ВУЗа) и start\_year (год начала обучения), а также функция fill\_records, генерирующая заданное количество записей (экземпляров структуры specialization) и записывающая их в динамический массив.

|  |
| --- |
| struct specialization  {  int code;  std::string university\_name;  int start\_year;  };  void fill\_records(specialization\* specializations, int size)  {  for (int i = 0; i < size; i++)  {  specializations[i].code = i + 1;  specializations[i].university\_name = "uni №" + std::to\_string(i + 1);  specializations[i].start\_year = 2000 + i;  }  } |

Для решения второго упражнения была написана функция linear\_search, которая реализует алгоритм линейного поиска.

|  |
| --- |
| void linear\_search(FILE\* file, int size, int code)  {  for (int i = 0; i < size; ++i)  {  specialization\* spec = get\_record(file, i);  if (spec->code == code)  {  std::cout << "Запись найдена линейным поиском\n";  return;  }  free(spec);  }  std::cout << "Запись не была найдена линейным поиском\n";  } |

Для решения третьего упражнения были написаны:

* Структура search\_record, имеющая целочисленные поля key (в данном случае код специализации) и position;
* Функция compare\_search\_records, выполняющая сравнение двух экземпляров структуры search\_record (нужна для последующей сортировки таблицы поиска);
* Функция create\_search\_table, создающая таблицу поиска;
* Функция binary\_search, осуществляющая бинарный поиск по заданной таблице поиска.

|  |
| --- |
| struct search\_record  {  int key;  int position;  };  bool compare\_search\_records(search\_record lhs, search\_record rhs)  {  return lhs.key < rhs.key;  }  void create\_search\_table(search\_record\* table, int size, FILE\* file)  {  for (int i = 0; i < size; i++)  {  specialization\* spec = get\_record(file, i);  table[i].key = spec->code;  table[i].position = i;  free(spec);  }  std::sort(table, table + size, compare\_search\_records);  }  void binary\_search(search\_record\* table, int code, int left, int right)  {  while (left <= right)  {  int mid = left + (right - left) / 2;  if (table[mid].key == code)  {  std::cout << "Запись найдена бинарным поиском\n";  return;  }  else if (table[mid].key < code)  {  left = mid + 1;  }  else  {  right = mid - 1;  }  }  std::cout << "Запись не была найдена бинарным поиском\n";  } |

При запуске программы пользователь видит приглашение ввести количество записей для генерации, после чего отображается пользовательское меню, позволяющее выбрать между различными реализованными функциями.

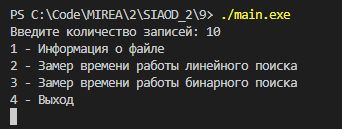


Рисунок 1. Интерфейс программы

# **Тестирование**

Протестируем выполнение программой первого упражнения. Для этого введём количество записей (100), выберем в меню первый пункт (информация о файле), после чего запустим линейный поиск последнего элемента, а потом бинарный. Как видно на рисунке 2, программа успешно сгенерировала 100 записей, записала их в файл, а оба поиска завершились успешно (элемент был найден в обеих случаях).

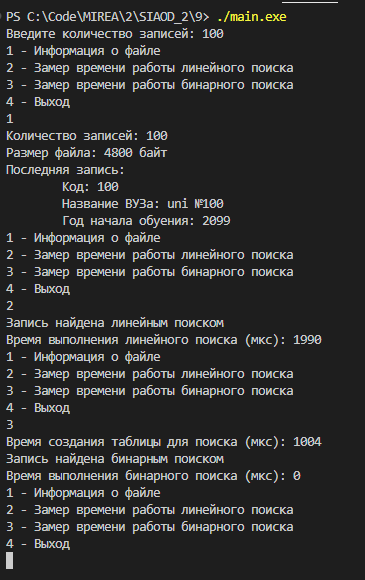


Рисунок 2. Тестирование программы

Таблица 1. Рабочий прогон программы

|  |  |  |  |
| --- | --- | --- | --- |
| **Количество элементов** | **Время работы линейного поиска, мкс** | **Создание структуры для бинарного поиска, мкс** | **Время работы бинарного поиска, мкс** |
| 100 | 1990 | 997 | 962 |
| 1000 | 8000 | 7053 | 1000 |
| 10000 | 50423 | 60035 | 1001 |
| 100000 | 364347 | 340038 | 1006 |
| 1000000 | 3226781 | 3623999 | 1000 |

Как видно из таблицы, алгоритм бинарного поиска работает быстрее линейного поиск, но из-за создания структуры время почти одинаковое. Хотя бинарный поиск все же эффективнее линейного, в файлах с небольшим объемом линейный поиск всё же быстрее.

Из результатов выполнения программы видно, что программа работает корректно, решая все поставленные задачи.

# **Вывод**

В результате выполнения работы я:

1. Освоил алгоритмы поиска в двоичных файлах и их реализацию на C++;
2. Научился программировать автоматическое тестирование простых программ.

# **Исходный код программы**

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <iostream>  #include <string>  #include <algorithm>  #include <chrono>  struct specialization  {  int code;  std::string university\_name;  int start\_year;  };  void fill\_records(specialization\* specializations, int size)  {  for (int i = 0; i < size; i++)  {  specializations[i].code = i + 1;  specializations[i].university\_name = "uni №" + std::to\_string(i + 1);  specializations[i].start\_year = 2000 + i;  }  }  specialization\* get\_record(FILE\* file, int position)  {  fseek(file, (position) \* sizeof(specialization), SEEK\_SET);  specialization\* spec = (specialization\*)malloc(sizeof(specialization));  fread(spec, sizeof(\*spec), 1, file);  return spec;  }  specialization\* get\_last\_record(FILE\* file, int size)  {  return get\_record(file, size - 1);  }  void linear\_search(FILE\* file, int size, int code)  {  for (int i = 0; i < size; ++i)  {  specialization\* spec = get\_record(file, i);  if (spec->code == code)  {  std::cout << "Запись найдена линейным поиском\n";  return;  }  free(spec);  }  std::cout << "Запись не была найдена линейным поиском\n";  }  struct search\_record  {  int key;  int position;  };  bool compare\_search\_records(search\_record lhs, search\_record rhs)  {  return lhs.key < rhs.key;  }  void create\_search\_table(search\_record\* table, int size, FILE\* file)  {  for (int i = 0; i < size; i++)  {  specialization\* spec = get\_record(file, i);  table[i].key = spec->code;  table[i].position = i;  free(spec);  }  std::sort(table, table + size, compare\_search\_records);  }  void binary\_search(search\_record\* table, int code, int left, int right)  {  while (left <= right)  {  int mid = left + (right - left) / 2;  if (table[mid].key == code)  {  std::cout << "Запись найдена бинарным поиском\n";  return;  }  else if (table[mid].key < code)  {  left = mid + 1;  }  else  {  right = mid - 1;  }  }  std::cout << "Запись не была найдена бинарным поиском\n";  }  int main()  {  int size = 0;  std::cout << "Введите количество записей: ";  std::cin >> size;  specialization\* specializations = new specialization[size];  fill\_records(specializations, size);  FILE\* file = fopen("records.bin", "wb+");  size\_t file\_size = fwrite(specializations, sizeof(specialization), size, file);  fclose(file);  int mode = 0;  while (true)  {  std::cout << "1 - Информация о файле\n"  << "2 - Замер времени работы линейного поиска\n"  << "3 - Замер времени работы бинарного поиска\n"  << "4 - Выход\n";  std::cin >> mode;  file = fopen("records.bin", "rb");  if (mode == 1)  {  specialization\* last\_record = get\_last\_record(file, size);  std::cout << "Количество записей: " << size << '\n'  << "Размер файла: " << size \* sizeof(specialization) << " байт\n"  << "Последняя запись:\n"  << "\tКод: " << last\_record->code << '\n'  << "\tНазвание ВУЗа: " << last\_record->university\_name << '\n'  << "\tГод начала обуения: " << last\_record->start\_year << '\n';  free(last\_record);  }  else if (mode == 2)  {  auto begin = std::chrono::high\_resolution\_clock::now();  linear\_search(file, size, size);  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);  std::cout << "Время выполнения линейного поиска (мкс): " << duration.count() << std::endl;  }  else if (mode == 3)  {  search\_record\* table = new search\_record[size];  auto begin = std::chrono::high\_resolution\_clock::now();  create\_search\_table(table, size, file);  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);  std::cout << "Время создания таблицы для поиска (мкс): " << duration.count() << std::endl;  begin = std::chrono::high\_resolution\_clock::now();  binary\_search(table, size, 0, size - 1);  end = std::chrono::high\_resolution\_clock::now();  duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - begin);    std::cout << "Время выполнения бинарного поиска (мкс): " << duration.count() << std::endl;  delete table;  }  else if (mode == 4)  {  fclose(file);  break;  }  else  {  std::cout << "Такого варианта нет!\n";  }  fclose(file);  }  return 0;  } |