Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГУСОДАРСТВЕННЫЙ УНИВРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

Кафедра вычислительных методов и программирования

Дисциплина: основы алгоритмизации и программирования

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе

на тему

**АЛГОРИТМЫ ПОИСКА**

**И СОРТИРОВКИ В МАССИВАХ**

Студент гр. 321702 Кислицын И.А.

Руководитель Семижон Е.А.

Почтовый адрес 222122

Минский р-н, д.Копище, ул. А.Яковлева

д. 4, к.31

тел. +375-29-208-35-78

e-mail: **[kislycat03@gmail.com](mailto:artemastahov27@gmail.com)**

Минск

2024

**РЕФЕРАТ**

**Цель:** данный курсовой проект является разработкой программы,которая реализует ведомость абитуриентов и выполняет операции над ней.При обработке баз данных часто применяются такой тип данных как структуры. База данных накапливается и хранится на диске в файле или создается пользователем. К ней часто приходится обращаться и обновлять.

**Объекты** **изучения:** алгоритмы поиска и сортировки для массивов, так как наиболее частыми операциями при работе с базами данных являются «поиск» и «сортировка». При этом алгоритмы решения этих задач существенно зависят от того, организованы структуры в массивы или размещены на диске.

Обычно элемент данных (структура) содержит некое ключевое поле

(ключ), по которому его можно найти. Ключом может служить любое поле

структуры, например, фамилия.

**Предметом** изучения являются следующие алгоритмы:

* Быстрая сортировка;
* Сортировка методом прямого обмена;
* Линейный поиск;
* Бинарный поиск.

Все названные алгоритмы применялись в процессе написания программы для разных целей.

В в процессе написания курсового проекта использовались следующие **методы исследования**: анализ литературы, интернет-ресурсов и различных документов по выбранной теме.

**СОДЕРЖАНИЕ**

Введение……………………………………………………………………………4

1.Структура и файлы…………………………………………………………..…..4

2.Алгоритмы сортировки…………………………………………………..….…..5

2.1 Общие сведения…………………………………………………………5

2.2 Метод прямого обмена………………………………………………..

2.4 Быстрая сортировка……………………………………………………10

3.Алгоритмы поиска……………………………………………………………..13

3.1 Общие свдения……..………………………………………………….13

3.2 Линейный поиск …..…………………………………………………..13

3.3 Бинарный поиск ………………………..……………….……………15

4.Пользовательские функции…………………………………………….….…..18

4.1 Общие сведения ……………..………………………………………..18

4.2 Функция вывода данных в консоль …………………………………..18

4.3 Чтение списка из файла …………………………………………..…..19

4.4 Запись списка в файл ……………………………………………..…..21

4.5 Сортировка пузырьком (по дате) ……………………………………..21

4.6 Быстрая сортировка (по ФИО) ………………………………….……22

4.7 Сортировка кучей (по номеру заказа) ………………………………..22

4.8 Функции, реализующие удаление дубликатов ……………….……..23 4.9 Функции, изменяющие список ………………………………..……..24

4.10 Функции поиска ……………………………………………………..25

5.Описание работы программы………………………………………………..27

Заключение ……………………………………………………………………..28

Список использованных источников …………………………………………..30

Приложение А Листинг кода …………………………………………………..31

Ведомость документов …………………………………………………………..44

ВВЕДЕНИЕ

1. СТРУКТУРЫ И ФАЙЛЫ

В процессе написания стрктуры мы имеем готовую ведомость абитуриентов.

Также пользователь может создать собственную ведомость и внести нужные в нее данные.

Список абитуриентов имеет такой вид:

Фамилия: Xxxxxxx, где Х —заглавная буква фамилии, x- любая буква

Номер группы: XXXXXX, где Х — любая цифра

Город: Ddddddd, где D - заглавная буква, города, d любая буква.

Средний балл: В, где В -любое средний балл

Во время выполнения программы данные абитуриентов хранятся в виде структуры **Vedomost**, содержащую следующие поля: string **name** для хранения Фамилии абитуриента,int **group** для хранения группы абитуриента, string **adress** для хранения данных о городе проживания абитуриента и double **average\_grade** для хранения среднего балла абитуриента**.**.

В процессе написания курсового проекта использовались разные стандартные библиотеки языка С++,такие как:

* **iostream** — для реализации консольного ввода/вывода;
* **vector** — для создания динамического массива;
* **fstream** — для реализации файлового ввода/вывода;
* **string** — для работы со строками;
* **iomanip** — для создания читабельного консольного вывода списка покупателей.
* **windows.h**— для создания эффекта загрузки (функция sleep()).

2. АЛГОРИТМЫ СОРТИРОВКИ

2.1 ОБЩИЕ СВЕДЕНИЯ

Программа позволяет выполнить два типа сортировки, а именно: метод прямого обмена и быстрая сортировка.

Каждый алгоритм сортирует ведомость абитуриентов по ключу, а именно по фамилии абитуриентов.

После выполнения сортировки список абитуриентов не меняет свой порядок, а просто выводит результат сортировки в консоль.

2.2 МЕТОД ПРЯМОГО ОБМЕНА

Алгоритм сортировки прямым обменом, также изветсный как сортировка пузырьком, является одним из самых простейших алгоритмов сортировки.Он работает путем многократного прохода по массиву, сравнивая соседние элементы и меняя их местами, если они стоят в неправильном порядке. Этот процесс повторяется до тех пор пока массив не будет отсортирован .

Принцип работы алгоритма сортировки пузырьком можно описать следующим образом:

1. Начнем сравнивать первый и второй элементы массива. Если они стоят в неправильном порядке(меньший элемент стоит после большего), то меняем их местами.
2. Переходим к следующей паре элементов (второй и третий) и снова сравниваем их меняя местами при необходимости.
3. Продолжаем этот процесс до конца массива, после чего самый большой элемент “всплывает” на правильную позицию в конце массива.
4. Повторяем все шаги для всех элементов массива, пока весь массив не будет отсортирован.

Алгоритм сортировки имеет сложность О(n^2) в худшем и средних случаях, что делает его не эффективным для больших массивов. Однако, он прост в реализации и может быть полезен для наборов данных или в учебных целях

В программе данный алгоритм используется для сортировки ведомости абитуриентов по фамилии (по алфавиту) и реализован в функции

void dc\_sort(vector<string>arr).

Пример сортировки изображены на рис.2 и рис.3:

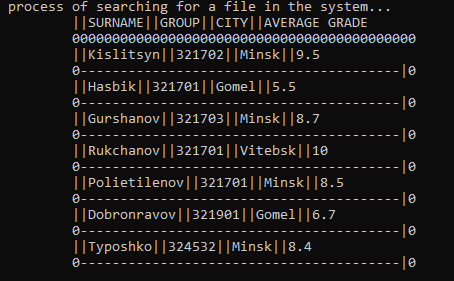


рис. 1 «Список абитуриентов до сортировки методом прямого обмена»



рис. 2 «Вывод списка абитуриентов после сортировки методом прямого обмена»

2.3 БЫСТРАЯ СОРТИРОВКА

Быстрая сортировка является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена. Это эффективный алгоритм сортировки, который использует стратегию “разделяй и властвуй”. Он работает следующим образом

Общая идея алгоритма состоит в следующем:

1. Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. Обычно это средний элемент массива .

2. Массив разделяется на две подгруппы: одна содержит элементы, меньшие опорного, а вторая - элементы, большие опорного .

3. Рекурсивно применяется алгоритм к каждой из подгрупп.

4. Результаты сортировки объединяются в один отсортированный массив

5. Рекурсия останавливается, когда размер подмассива становится равным 1 или 0.

Быстрая сортировка быстро сортирует данные, особенно когда данные хорошо разделены на подмассивы при каждом шаге разделения. В лучшем случае его сложность составляет O(n log n).

В худшем случае (если опорный элемент выбирается неудачно), сложность алгоритма может составлять O(n^2) .

Алгоритм быстрой сортировки применяется, где требуется эффективная сортировка данных.

В программе алгоритм сортирует ведомость абитуриентов по их фамилиям и реализован в функциях

void quickSort(vector<string>& arr, int low, int high)

и

int partition(vector<string>& arr, int low, int high).

.

Пример работы алгоритма в программе изображён на рис. 3 и рис. 4:

рис. 3 «Список абитуриентов до быстрой сортировки»

рис. 4 «Список покупателей после быстрой сортировки»

3. АЛГОРИТМЫ ПОИСКА

3.1 ОБЩИЕ СВЕДЕНИЯ

Алгоритмы поиска — это методы или процедуры, используемые для поиска данных или информации внутри определенной структуры, такой как база данных, массив или документ. Они помогают найти элементы, соответствующие определенным критериям, или выполнить поиск по ключевым словам, числам или другим атрибутам.

Алгоритмы поиска применяются в различных областях, включая информатику, базу данных, веб-технологии и многие другие. Их основная задача — эффективно и быстро находить нужную информацию, минимизируя затраты времени и ресурсов.

Эти алгоритмы позволяют создавать эффективные системы поиска, такие как поисковые системы в Интернете, базы данных, системы рекомендаций и многое другое. Они также применяются в играх, робототехнике и других областях, где важно быстро находить информацию или оптимальные пути.

В программе использовались два алгоритма поиска: линейный и бинарный.

3.2 ЛИНЕЙНЫЙ ПОИСК

Линейный поиск, также известный как "поиск последовательным обходом" или "поиск методом полного перебора", — это простой алгоритм, который заключается в проверке каждого элемента в списке или массиве до тех пор, пока не будет найден нужный элемент или не закончится список. Это базовый и понятный метод поиска, который полезен в ситуациях, когда другие, более сложные алгоритмы поиска могут быть избыточными или неэффективными.

Принцип работы линейного поиска:

Алгоритм линейного поиска проходит через все элементы массива (или списка) в последовательном порядке, начиная с первого элемента и заканчивая последним. В процессе он сравнивает каждый элемент с искомым значением:

1) Инициализация: Начинает поиск с первого элемента массива.

2) Цикл: Проходит по всем элементам массива, сравнивая их с искомым значением.

3) Проверка совпадения: Если элемент равен искомому значению, поиск завершается, и возвращается индекс элемента или сам элемент.

4) Продолжение поиска: Если элемент не равен искомому значению, алгоритм продолжает двигаться к следующему элементу.

5) Завершение: Если алгоритм дошел до конца массива и не нашел искомого элемента, он сообщает, что элемент не найден.

Линейный поиск является одним из простейших алгоритмов поиска, и его применение оправдано в следующих случаях:

1) Малые или неотсортированные массивы: Когда массив или список относительно небольшой, и использование сложных алгоритмов не оправдано из-за их накладных расходов.

2) Несортированные данные: Если массив или список не отсортированы, линейный поиск — это простой способ найти элемент без предварительной подготовки данных.

3) Операции с единичными элементами: Если поиск выполняется один раз или редко, линейный поиск может быть более эффективным, чем более сложные алгоритмы.

Линейный поиск обладает следующими преимуществами и недостатками:

Преимущества:

1) Простота и понятность: Линейный поиск легко понять и реализовать.

2) Отсутствие требований к структуре данных: Не требуется предварительная сортировка или специальные структуры данных.

Недостатки:

1) Неэффективен для больших массивов: При большом количестве элементов время поиска пропорционально размеру массива, так как сложность алгоритма .

2) Не использует структуры данных для ускорения поиска: Более сложные алгоритмы, такие как двоичный поиск или хеш-таблицы, могут быть намного быстрее при больших объемах данных.

В программе линейный поиск используется в функции void **searchByDate**(vector<Buyer> buyers, string date), для поиска записей сделанных в определённое число. Так же модификации линейного поиска использовались во многих других функциях.

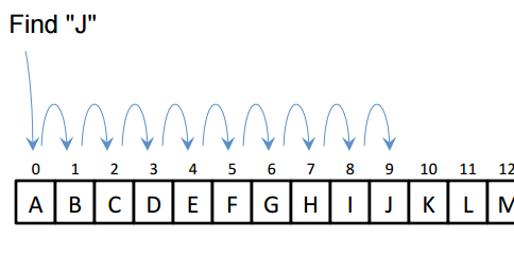
Схематический пример работы линейного поиска изображён на рис. 10:

рис. 10

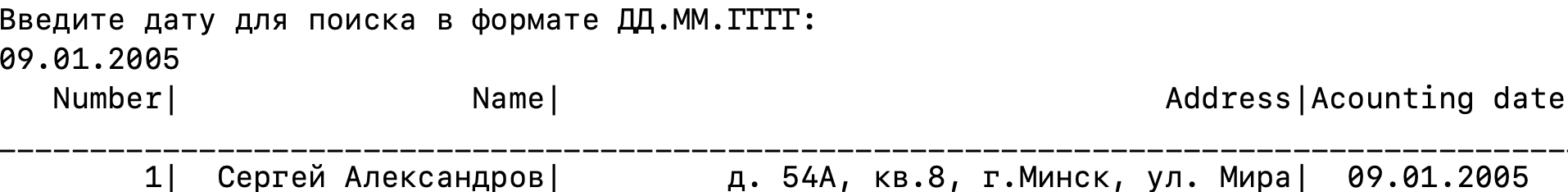
Пример линейного поиска в программе изображён на рис. 11:

рис. 11

3.3 БИНАРНЫЙ ПОИСК

Двоичный поиск, также известный как бинарный поиск, — это эффективный алгоритм поиска, который применяется к отсортированным массивам или спискам. Он значительно быстрее линейного поиска, особенно для больших массивов, поскольку делит пространство поиска на половины, сокращая количество проверок.

Основная идея двоичного поиска заключается в том, чтобы на каждой итерации делить массив на две части и определять, в какой из них может находиться искомый элемент [5]:

1) Инициализация: Устанавливается начальный (left) и конечный (right) индексы массива. Изначально left равен 0, а right — индексу последнего элемента.

2) Определение середины: Вычисляется индекс середины массива (mid), который равен left + (right - left) // 2. Этот способ вычисления середины предотвращает переполнение целых чисел при больших значениях.

3) Сравнение с искомым элементом:

Если значение элемента в mid равно искомому значению, поиск завершается, и возвращается индекс mid.

Если значение в mid больше искомого, поиск продолжается в левой половине массива (right = mid - 1).

Если значение в mid меньше искомого, поиск продолжается в правой половине массива (left = mid + 1).

4) Продолжение поиска: Цикл продолжается до тех пор, пока left не превысит right, что означает, что искомый элемент не найден.

Двоичный поиск наиболее эффективен в следующих случаях:

1) Алгоритм работает только с отсортированными данными, поэтому предварительная сортировка обязательна.

2) Двоичный поиск более эффективен по сравнению с линейным при больших объемах данных, поскольку количество операций сокращается экспоненциально.

Двоичный поиск имеет следующие преимущества и недостатки:

Преимущества:

1) Высокая эффективность: С каждым шагом двоичный поиск делит массив на две части, что значительно сокращает количество операций.

2) Стабильное время выполнения: Алгоритм работает за логарифмическое время (O(log n)), что очень быстро при больших массивах.

Недостатки:

1) Требует отсортированных данных: Если массив не отсортирован, двоичный поиск работать не будет.

2) Не подходит для динамических структур: Если структура данных изменяется, может потребоваться повторная сортировка.

Двоичный поиск широко применяется в различных областях, включая:

поиск в отсортированных массивах, алгоритмы поиска в базах данных, поиск в графах и работа с системами сортировки.

Двоичный поиск часто используется в задачах, где важно быстро найти элемент в отсортированном списке, таких как базы данных, индексы или бинарные деревья. Благодаря своей эффективности он является одним из базовых алгоритмов, изучаемых в информатике и программировании.

Схематический пример работы бинарного поиска изображён на рис. 12, где искомым является число 55:

рис. 12

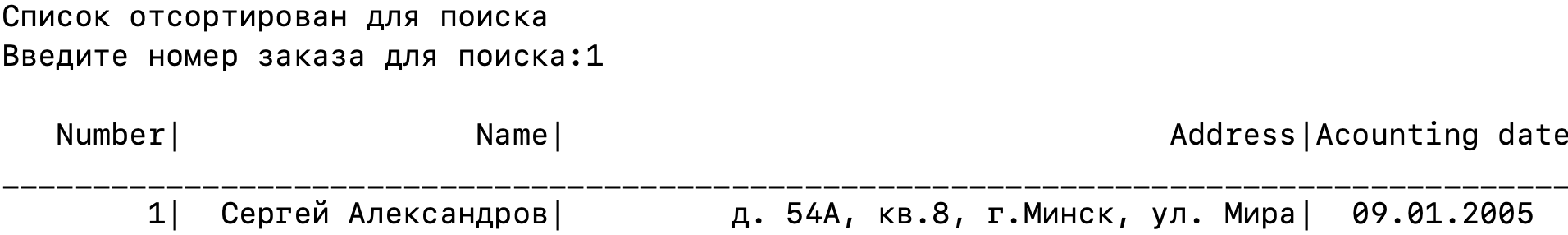
Пример работы бинарного поиска в программе изображён на рис. 13:

рис. 13

4. ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ

4.1 ОБЩИЕ СВЕДЕНИЯ

Пользовательские функции — это функции, созданные пользователем или разработчиком, а не входящие в стандартную библиотеку или встроенные в язык программирования. Они позволяют добавить новый функционал, который может быть специфическим для определенного проекта или приложения.

В программе определён ряд пользовательских функций, реализующих алгоритмы поиска, сортировки и выполнение прочих задач.

4.2 ФУНКЦИЯ ВЫВОДА ДАННЫХ В КОНСОЛЬ

Консольный вывод определён двумя вариантами:

1) void **outputToConso**le(vector<Buyer> *buyers*), где *buyers —* это список всех покупателей. Данная функция выводит весь список в виде читабельной таблицы.

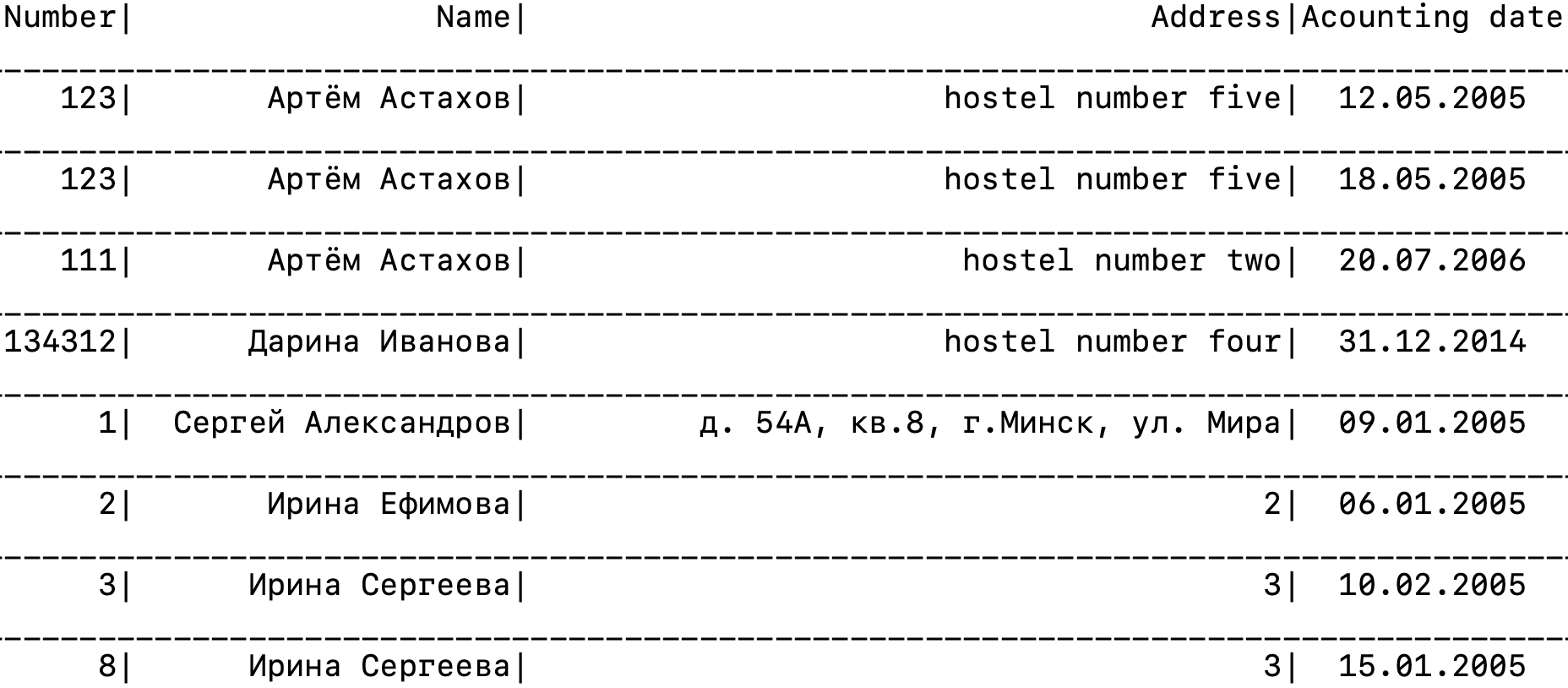
Пример изображён на рис. 14:

рис. 14

Функция позволяет выводить в таблицу как латинские символы, так и кириллицу.

2) void **outputToConsole**(Buyer *buyer*), где *buyer* — отдельный заказ. Функция работает идентично её перегрузке, однако служит для вывода одного покупателя.

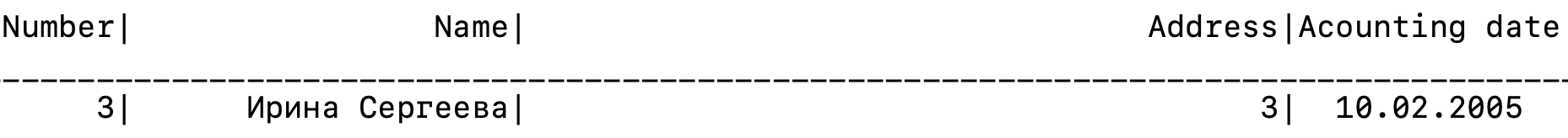
Пример изображён на рис. 15:

рис. 15

4.3 ЧТЕНИЕ СПИСКА ИЗ ФАЙЛА

Чтение списка покупателя программой реализовано в функции vector<Buyer> **getData**(). Функция открывает файл «input.txt», с которого она «читает» список покупателей, записывает его в локальную переменную *buyers* и возвращает её.

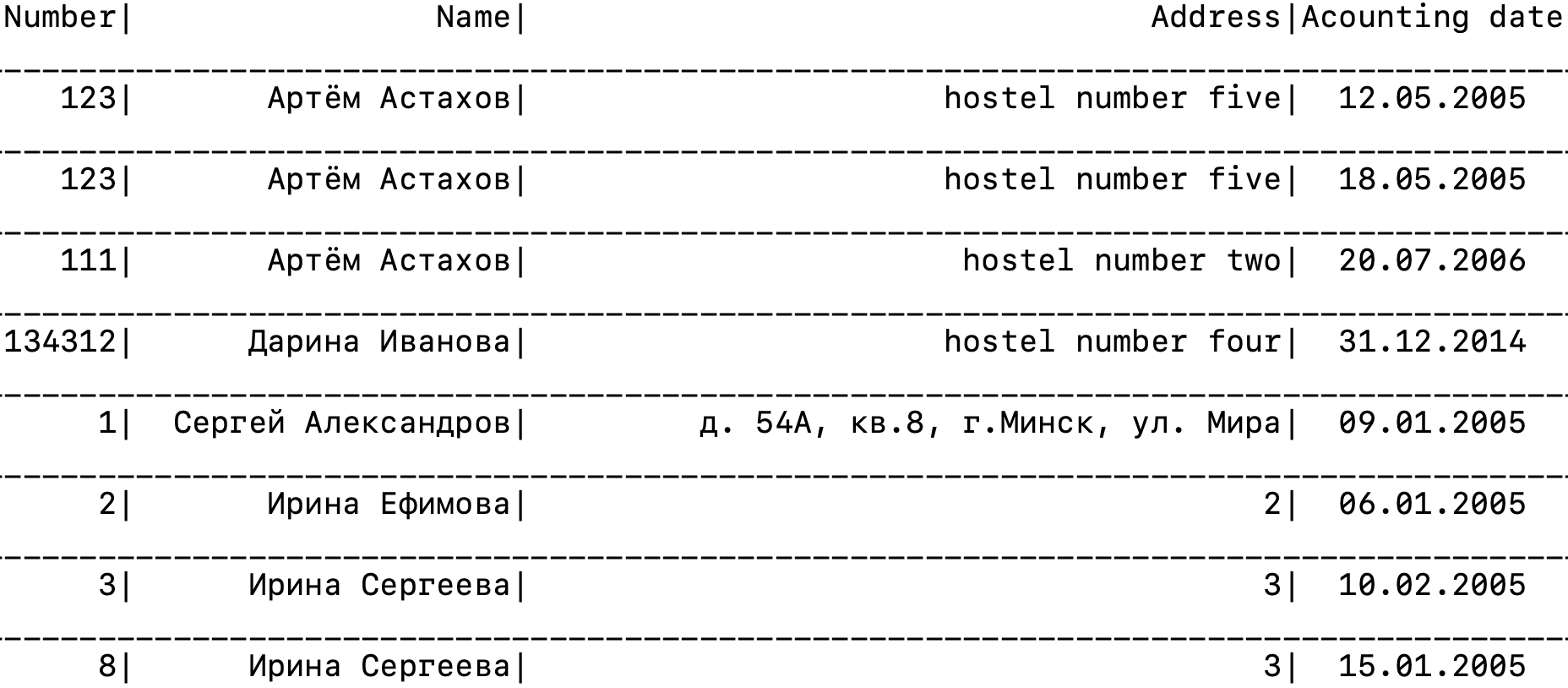
Пример функции изображён на рис. 16 и рис. 17:

рис. 16 «Список, прочтённый программой»

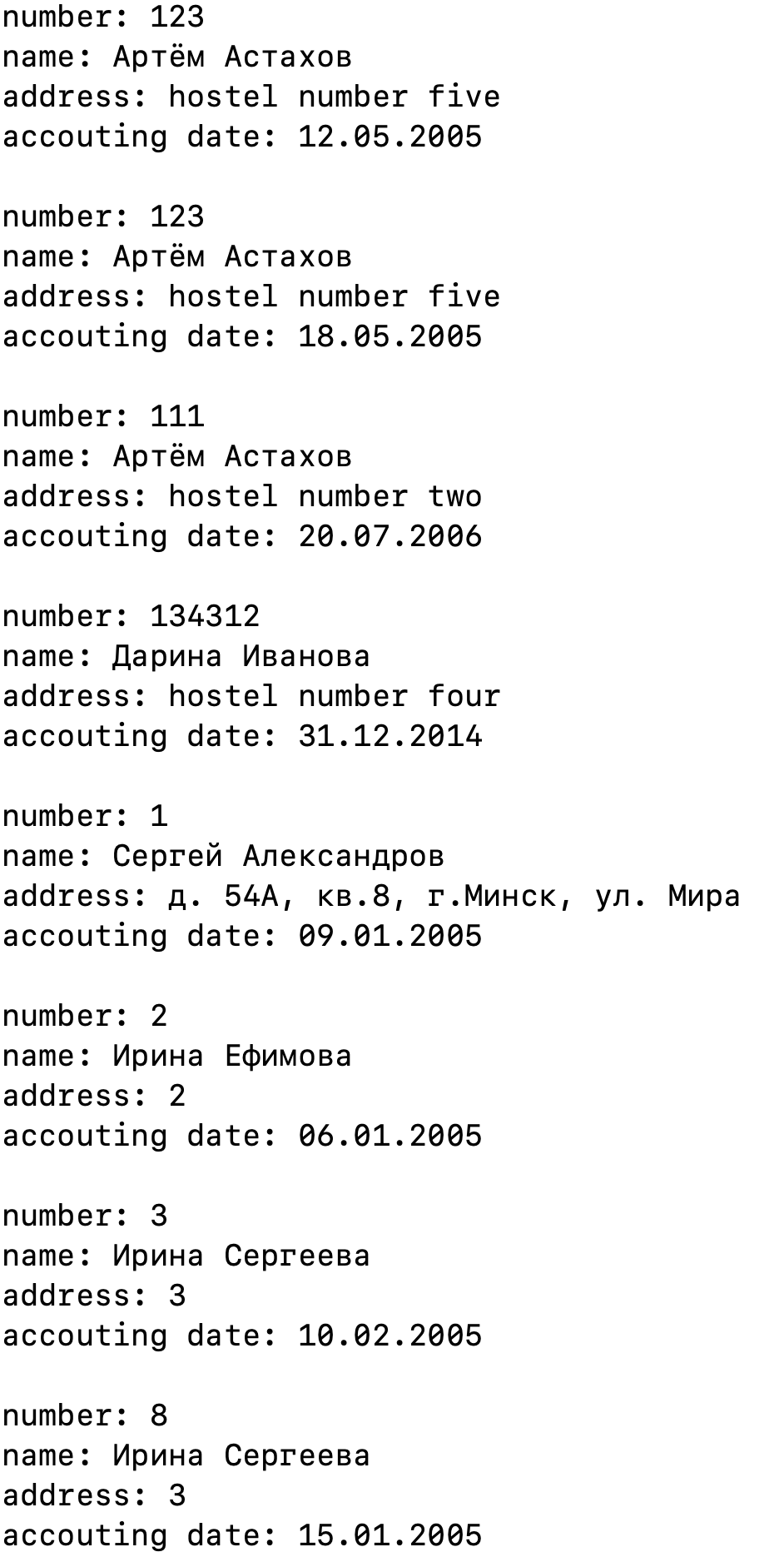


рис. 17 «Содержимое файла»

4.4 ЗАПИСЬ СПИСКА В ФАЙЛ

Запись списка покупателей является неотъемлемой частью работы программы, она реализована в функции void **putData**(vector<Buyer> *buyers*), где *buyers —* список покупателей, который будет выведен в файл «output.txt».

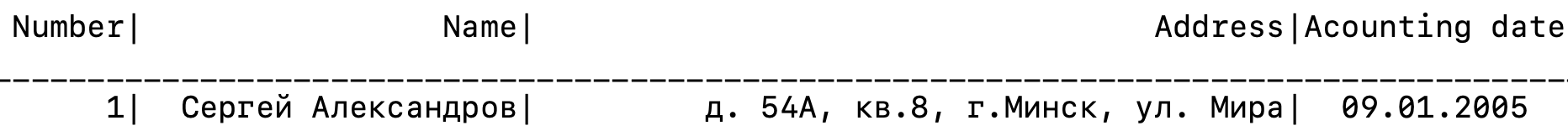
Пример работы функции изображён на рис. 18 и рис. 19:

рис. 18 «Список покупателей»

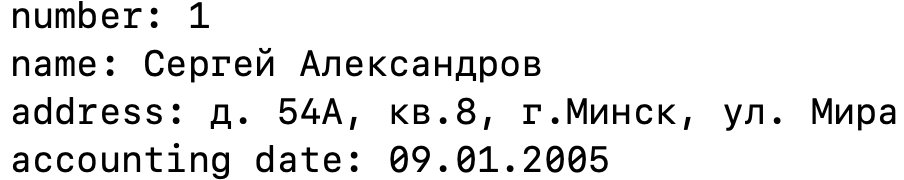


рис. 19 «Список записанный в файл»

4.5 СОРТИРОВКА ПУЗЫРЬКОМ (ПО ДАТЕ)

В программе есть возможность сортировать список по дате, функция vector<Buyer> **bubbleSort\_Date**(vector<Buyer> *buyers*), где *buyers —* это список всех покупателей, использует метод прямого обмена для этой цели. Она разбивает дату на 3 значения: день, месяц и год, далее в процессе сортировке использует метод прямого обмена сравнивая год, в случае совпадения месяц и в случае совпадения сравнивает дни.

Функция возвращает уже отсортированный список покупуталей.

Пример работы данной функции был продемонстрирован на рис. 2 и рис.3 в г.2.2

4.6 БЫСТРАЯ СОРТИРОВКА (ПО ФИО)

Сортировка по ФИО происходит в функции vector<Buyer> **quickSort\_Name**(vector<Buyer> *buyers*), где *buyers —* это список всех покупателей, используя метод быстрой сортировки. Функция сравнивает отдельно первое слово ФИО и, если того требует ситуация, второе слово ФИО методом определённым синтаксисом языка С++.В результате возвращается уже отсортированный список.

Пример работы функции был изображён на рис. 8 и рис. 9 в г.2.4.

4.7 СОРТИРОВКА КУЧЕЙ (ПО НОМЕРУ ЗАКАЗА)

Сортировка по номеру заказа с помощью сортировки кучи происходит с помощью функций: void **heapify**(vector<Buyer>& *buyers*, int *n*, int *i*), где *buyers —* список всех покупателей, *n — э*то общее количество элементов в массиве или векторе, который сортируется*, i —* указывает текущий индекс элемента, который необходимо проверить и, возможно, привести к структуре кучи; void **heapSort\_Number**(vector<Buyer>& *buyers*), где *buyers —* список всех покупателей.

Эти функции имею следующие цели:

1) **heapify** — восстанавливает свойство кучи в части вектора, начиная с указанного индекса i. Если элемент по индексу i меньше одного из его дочерних элементов (левого или правого), они меняются местами, и процесс повторяется рекурсивно для дочернего индекса, где произошел обмен. Это гарантирует, что кучи остаются "max-heap" (корневой элемент больше всех дочерних).

2) **heapSort\_Number** — выполняет сортировку кучей. Сначала она строит кучу (с помощью heapify), начиная с середины массива и двигаясь к началу. Затем она повторно применяет heapify к оставшимся элементам, чтобы извлечь максимальный элемент, переместить его в конец и восстановить кучу на уменьшенном диапазоне. В результате элементы вектора оказываются отсортированными.

В результате список пользователь имеет отсортированный по номеру заказа список покупателей.

Результат работы функции был продемонстрирован на рис. 5 и рис. 6 в г.2.3

4.8 ФУНКЦИИ, РЕЛИЗУЮЩИЕ УДАЛЕНИЯ ДУБЛИКАТОВ

В программе есть возможность удалять дубликаты (сравнивая ФИО и адрес) из списка покупателей, благодаря двум функциям:

1) vector<Buyer> **deleteDublicates**(vector<Buyer> *buyers*), где *buyers —* список всех покупателей. Данная функция сортирует список покупателей по ФИО, после того, как покупатели с одинаковыми ФИО располагаются в списке рядом, вызывается функция **linearSearch\_toDelete**, возвращающая список индексов для удаления. Далее **deleteDublicates** возвращает список покупателей без дубликатов;

2) В свою очередь функция vector<int> l**inearSearch\_toDelete**(vector<Buyer> *buyers*, int *startIndex*), где *buyers —* список всех покупателей, *startIndex* — позиция с которой начинается ряд покупателей с повторяющимися ФИО. Функция сравнивает адреса до тех пор пока ФИО не поменяется. Если совпадает адрес, индекс элемента добавляется в список. l**inearSearch\_toDelete** возвращает список индексов покупателей, подлежащих удалению.

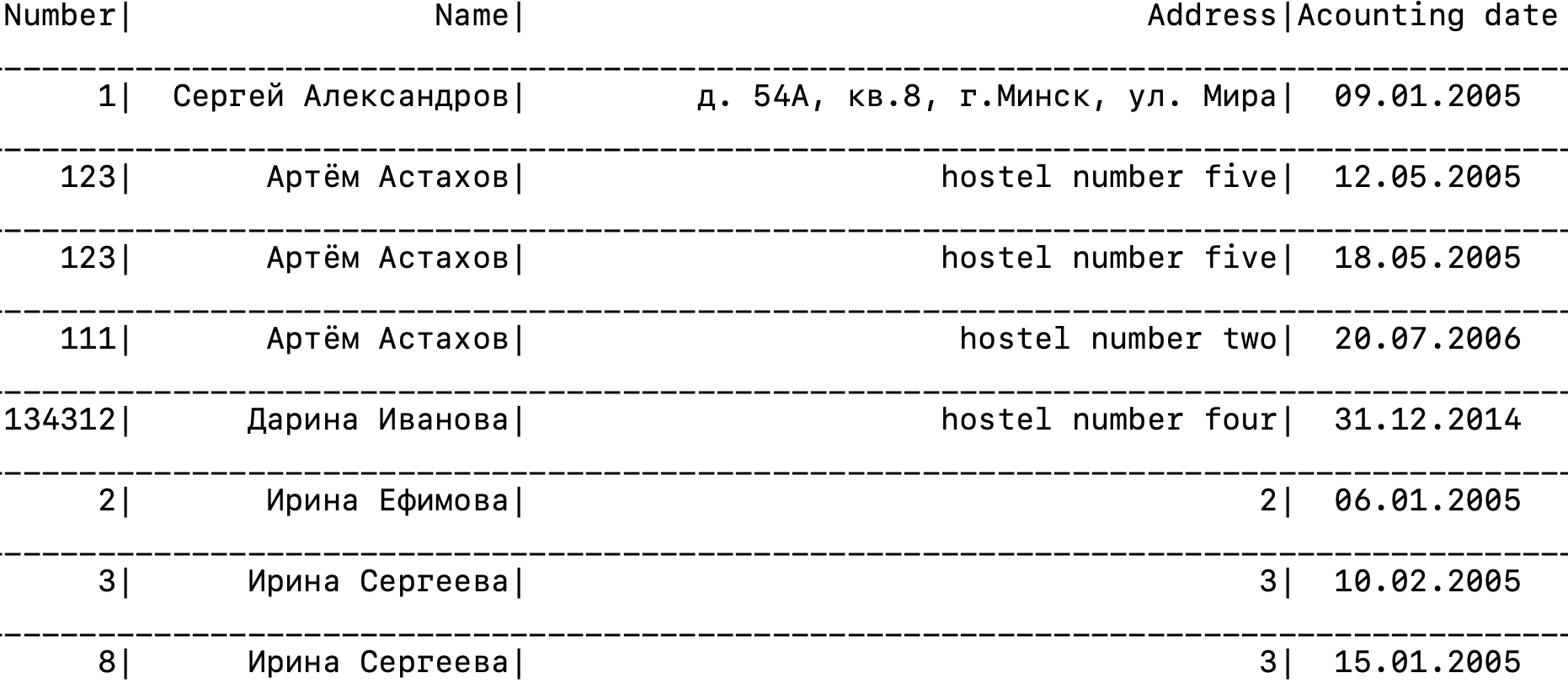
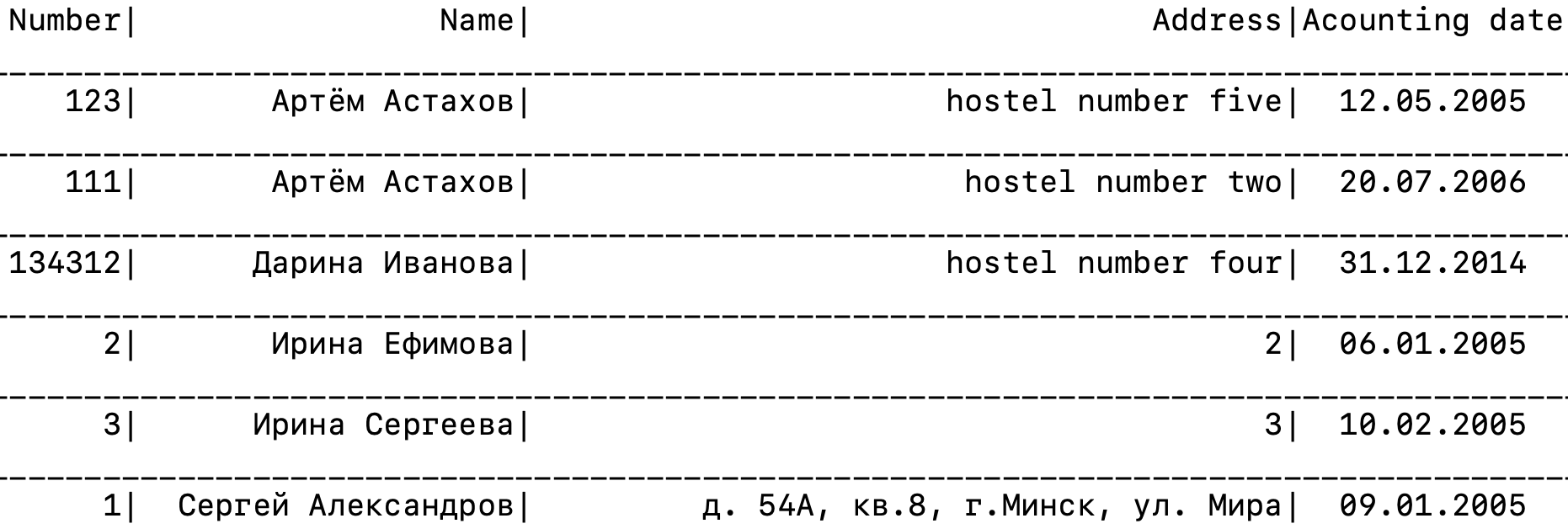
Пример работы программы изображён на рис. 20 и рис. 21:

рис. 20 «Список покупателей до удаления дубликатов»

рис. 21 «Список покупателей после удаления дубликатов»

4.9 ФУНКЦИИ, ИЗМЕНЕНЯЮЩИЕ СПИСОК

В программе реализованы функции, позволяющие добавлять и удалять записи в списке покупателей в течении работы программы.

Функция **добавления** — void **addToBuyersList**(vector<Buyer> &*buyers*), где *buyers —* список всех покупателей. Функция спрашивает всю информацию для новой записи и впоследствии добавляет в исходный список.

Результат работы функции представлен на рис. 22 и рис. 23:

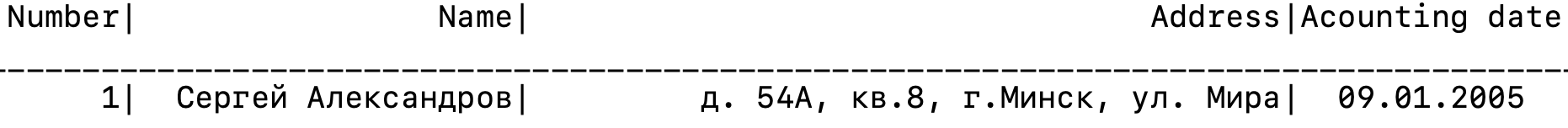


рис. 22 «Список покупателей до добавдения новых записей»

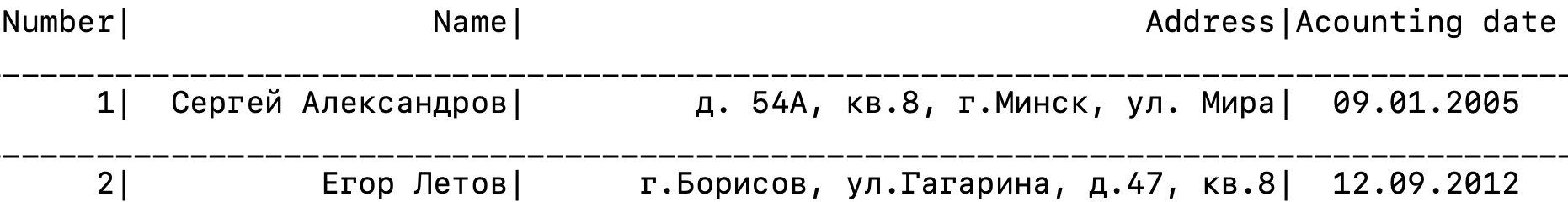


рис. 23 «Список покупателей после добавдения новых записей»

Функция **удаления** — void **deleteFromBuyersList**(vector<Buyer> &*buyers*), где *buyers —* список всех покупателей. Функция спрашивает у пользователя номер функции для удаления, ищет её (бинарный поиск) и удаляет.

Результат работы функции изображён на рис. 24 и рис. 25:

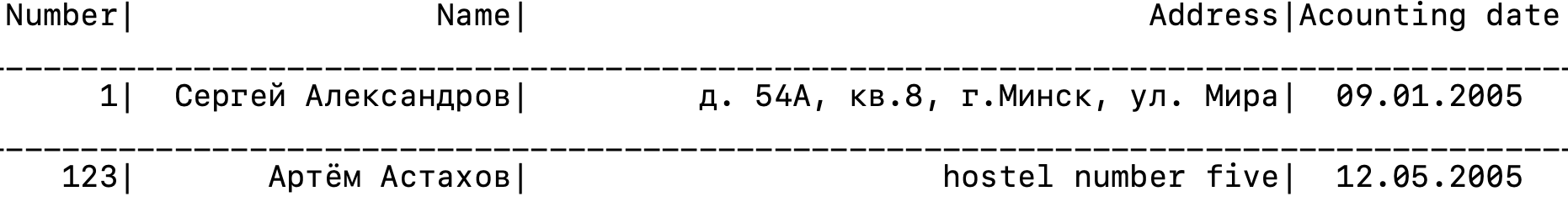


рис. 24 «Список покупателей до удаления записей»

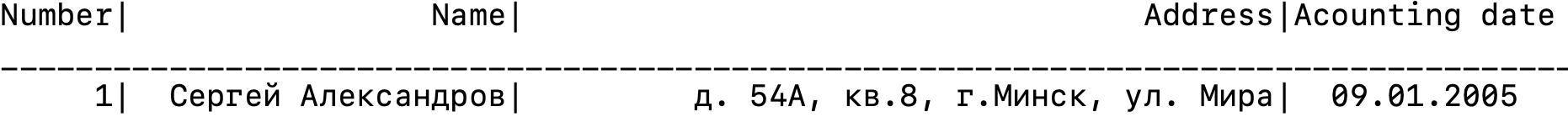


рис. 25 «Список покупателей после удаления записей»

4.10 ФУНКЦИИ ПОИСКА

В программе реализованы поиск по дате и по номеру заказа, реализация каждого использует разные алгоритмы (бинарный и линейный поиск).

void **searchByDate**(vector<Buyer> *buyers*, string *date*), где *buyers —* список всех покупателей, а *date* — искомая дата. Функция проходит по всему списку покупателей (линейный поиск), в случае совпадения даты с искомой, заказ добавляется в новый список. После обхода всего списка, происходит вывод всех найденый заказов с датой, совпадающей искомой. Если список найденных значений пуст выводится строка «В тот день не было заказов».

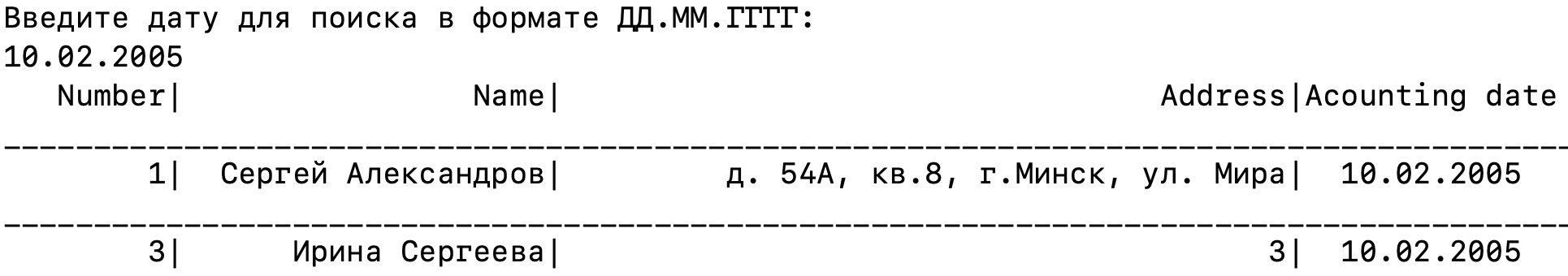
Пример работы поиска изображён на рис. 26:

рис. 26

int **binarySearch\_Number**(vector<Buyer> *buyers*, int *low*, int *high*, int *number*) где *buyers —* список всех покупателей, *low* — нижняя граница поиска, *max —* верхняя граница поиска, *number* — искомый номер. Функция предварительно сортирует список покупателей по номеру с помощью функции **heapSort\_Number**, что является обязательным требованием для бинарного поиска. Далее функция находит заказ с совпадающим номером и возвращает его номер, в случае отсутствия такого заказа возвращается значение «-1».

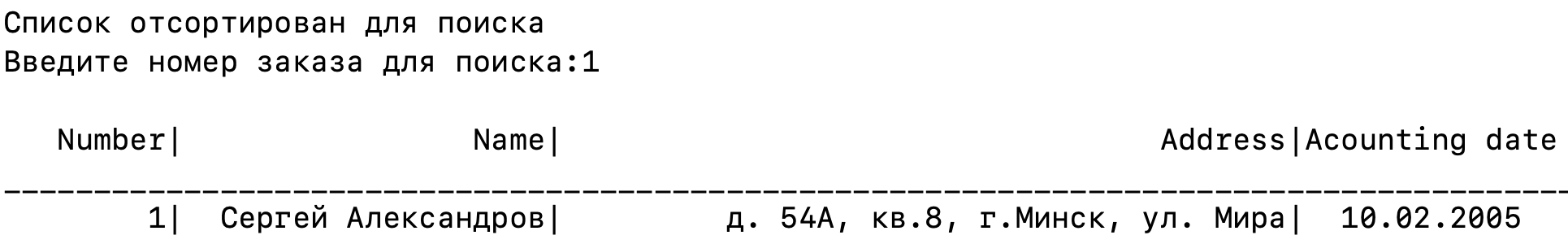
Пример работы функции изображён на рис.27:

рис. 27

5. ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

Выполнение программы начинается с создания списка покупателей магазина, далее он заполняется содержимым файла, используя функцию **getData**. Далее вся работа программы зависит от пользователя, в программе реализован консольный интерфейс, в котором определены 11 операций, выполняющих одноимённые действия:

1 - Сортировка по номеру заказа;

2 - Сортировка по дате заказа;

3 - Сортировка по имени заказчика;

4 - Поиск заказа по номеру;

5 - Поиск заказов по дате;

6 - Вывести с учётом сортировки;

7 - Добавить запись;

8 - Удалить запись;

9 - Удалить дубликаты;

10 - Записать список покупателей в файл;

11 - Выход.

После выполнения операций под номерами 1, 2, 3, 7, 8, 9 список покупателей может как менять порядок записей в списке, так добавлять либо удалять записи.

Программа не завершит своё выполнение до тех пор, пока пользователь не выберет операцию под номером 11 («Выход»).

6. ЗАКЛЮЧЕНИЕ

В процессе выполнения курсового проекта, была разработана программа для работы со списком покупателей магазина.

Для того чтобы представить список покупателей в виде базы данных была разработана структура файлов для ввода и вывода списка и структура списка покупателей для использования в кода программы. Все структуры подобно описаны в главе 1.

Для операций с записями были использованы следующие алгоритмы сортировки:

* + - Сортировка кучей — сортировка строит "max-heap", затем извлекает максимальный элемент и перемещает его в конец. Процесс повторяется, уменьшая размер кучи, пока все элементы не будут отсортированы;
    - Сортировка методом прямого обмена — сортировка сравнивает соседние элементы и меняет их местами, если они в неправильном порядке. Этот процесс повторяется, перемещая максимальные элементы к концу массива, пока весь массив не будет отсортирован. Каждый цикл "пузырька" гарантирует, что следующий максимальный элемент окажется на своем месте;
    - Быстрая сортировка — сортировка разделяет массив на две части вокруг выбранного опорного элемента (pivot), так что все элементы слева от опорного меньше или равны ему, а все элементы справа больше. Затем быстрая сортировка рекурсивно применяется к обеим частям. Этот процесс продолжается, пока разделенные части не станут достаточно малыми, чтобы быть отсортированными. В итоге массив становится полностью отсортированным.

Все алгоритмы сортировки и их применение были подробно описаны во 2 главе.

Так же использовались следующие алгоритмы поиска:

* Линейный поиск — это алгоритм поиска, который проходит по массиву или списку поочередно, начиная с первого элемента и продолжая к последнему, пока не найдет нужное значение или не пройдет весь массив. Он простой, но может быть медленным для больших массивов, поскольку его сложность в худшем случае составляет *O(n)*, где n — число элементов в массиве. Этот метод не требует предварительной сортировки массива;
* Бинарный поиск — это алгоритм поиска, который работает на отсортированных массивах. Он повторно делит массив на две части, используя середину в качестве контрольной точки. Если значение, которое вы ищете, меньше, чем середина, поиск продолжается в левой половине, а если больше — в правой. Этот процесс продолжается, пока не будет найден нужный элемент или пока не останется элементов для поиска. Бинарный поиск гораздо быстрее линейного на больших данных, с временной сложностью , но требует предварительной сортировки массива.

Все алгоритмы поиска и их применение были подробно описаны в 3 главе.

В 4 главе были подробно описаны цель и применение всех определённых функций созданных пользователем.

В 5 главе был описан процесс работы программы и описано её использование пользователем.

В процессе написания курсового проекта были подробно изучены алгоритмы поиска и сортировки, ситуации их оптимального использования. Все полученные знания были использованы на практике.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сортировка прямым обменом (метод «пузырька») [электронный ресурс] // URL: **<https://prog-cpp.ru/sort-bubble/>** (дата обращения 27.04.2024).

**2.** Пирамидальная сортировка (HeapSort) [электронный ресурс] // URL: **<https://habr.com/ru/companies/otus/articles/460087/>** (дата обращения 27.04.2024).

**3.** Binary tree [электронный ресурс] // URL: **<https://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees>** (дата обращения 27.04.2024).

**4.** Быстрая сортировка [электронный ресурс] // URL: **[https://ru.wikipedia.org/wiki/Быстрая\_сортировка](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)** (дата обращения 27.04.2024).

**5.** Binary Insertion Sort in C++ [электронный ресурс] // URL: **<https://www.tutorialspoint.com/binary-insertion-sort-in-cplusplus>** (дата обращения 28.04.2024).

ПРИЛОЖЕНИЕ А ЛИСТИНГ КОДА

#include <iostream>

#include <vector>

#include <fstream>

#include <string>

#include <iomanip>

**using** **namespace** std;

**struct** Buyer {

**int** number;

string name;

string address;

string accountingDate;

};

vector<Buyer> getData() {

ifstream file("input.txt");

vector<Buyer> buyers;

**if** (file.is\_open()) {

string temp;

**while** (!file.eof()) {

Buyer buyer;

getline(file, temp);

temp.replace(0, 8, "");

buyer.number = stoi(temp);

getline(file, temp);

temp.replace(0, 6, "");

buyer.name = temp;

getline(file, temp);

temp.replace(0, 9, "");

buyer.address = temp;

getline(file, temp);

temp.replace(0, 16, "");

buyer.accountingDate = temp;

buyers.push\_back(buyer);

getline(file, temp);

}

} **else** {

cout << "Файл для чтения не открыт" << endl;

}

file.close();

**return** buyers;

}

**void** putData(vector<Buyer> buyers) {

ofstream file("output.txt");

**if** (file.is\_open()) {

**for** (**int** i = 0; i < buyers.size(); i++) {

file << "number: " << buyers[i].number << endl;

file << "name: " << buyers[i].name << endl;

file << "address: " << buyers[i].address << endl;

file << "accounting date: " << buyers[i].accountingDate << endl;

file << "\n";

}

} **else** {

cout << "Файл для записи не открыт" << endl;

}

file.close();

}

vector<Buyer> bubbleSort\_Date(vector<Buyer> buyers) {

**bool** protect = **true**;

**while**(protect) {

protect = **false**;

**for** (**int** i = 1; i < buyers.size(); i++) {

**int** yearsOfFisrt, mounthOfFirst, daysOfFirst, yearsOfSecond, mounthOfSecond, daysOfSecond;

daysOfFirst = stoi(buyers[i - 1].accountingDate.substr(0,2));

mounthOfFirst = stoi(buyers[i - 1].accountingDate.substr(3,2));

yearsOfFisrt = stoi(buyers[i - 1].accountingDate.substr(6,buyers[i].accountingDate.size() - 6));

daysOfSecond = stoi(buyers[i].accountingDate.substr(0,2));

mounthOfSecond = stoi(buyers[i].accountingDate.substr(3,2));

yearsOfSecond = stoi(buyers[i].accountingDate.substr(6,buyers[i].accountingDate.size() - 6));

**if** (yearsOfFisrt > yearsOfSecond) {

swap(buyers[i - 1], buyers[i]);

protect = **true**;

**continue**;

}

**if** (yearsOfFisrt == yearsOfSecond) {

**if** (mounthOfFirst > mounthOfSecond) {

swap(buyers[i - 1], buyers[i]);

protect = **true**;

**continue**;

}

**if** (mounthOfFirst == mounthOfSecond) {

**if** (daysOfFirst > daysOfSecond) {

swap(buyers[i - 1], buyers[i]);

protect = **true**;

**continue**;

}

}

}

}

}

**return** buyers;

}

vector<Buyer> quickSort\_Name(vector<Buyer> buyers) {

vector<Buyer> left;

vector<Buyer> right;

**for** (**int** i = 1; i < buyers.size(); i++) {

string bearingElementName, secondName;

bearingElementName = buyers[0].name.substr(0, buyers[0].name.size() - buyers[0].name.find(" ") - 1);

secondName = buyers[i].name.substr(0, buyers[i].name.size() - buyers[i].name.find(" ") - 1);

**if** (bearingElementName < secondName) {

right.push\_back(buyers[i]);

} **else** **if** (bearingElementName > secondName) {

left.push\_back(buyers[i]);

} **else** {

bearingElementName = buyers[0].name.substr(buyers[0].name.find(" "), buyers[0].name.size() - buyers[0].name.find(" ") - 1);

secondName = buyers[i].name.substr(buyers[i].name.find(" "), buyers[i].name.size() - buyers[i].name.find(" ") - 1);

**if** (bearingElementName <= secondName) {

right.push\_back(buyers[i]);

} **else** **if** (bearingElementName < secondName) {

left.push\_back(buyers[i]);

}

}

}

**if** ( left.size() > 1)

left = quickSort\_Name(left);

**if** (right.size() > 1)

right = quickSort\_Name(right);

left.push\_back(buyers[0]);

**for** (**int** i = 0; i < right.size(); i++) {

left.push\_back(right[i]);

}

**return** left;

}

**void** heapify(vector<Buyer> &buyers, **int** n, **int** i) {

**int** largest = i;

**int** left = 2 \* i + 1;

**int** right = 2 \* i + 2;

**if** (left < n && buyers[left].number > buyers[largest].number) {

largest = left;

}

**if** (right < n && buyers[right].number > buyers[largest].number) {

largest = right;

}

**if** (largest != i) {

swap(buyers[i], buyers[largest]);

heapify(buyers, n, largest);

}

}

**void** heapSort\_Number(vector<Buyer> &buyers) {

**int** n = (**int**)buyers.size();

**for** (**int** i = n / 2 - 1; i >= 0; i--) {

heapify(buyers, n, i);

}

**for** (**int** i = n - 1; i > 0; i--) {

swap(buyers[0], buyers[i]);

heapify(buyers, i, 0);

}

}

vector<**int**> linearSearch\_toDelete(vector<Buyer> buyers, **int** startIndex) {

vector<**int**> indexesToDelete;

**for** (**int** i = startIndex + 1; i < buyers.size(); i++) {

**if** (buyers[startIndex].name == buyers[i].name){

**if** (buyers[startIndex].address == buyers[i].address)

indexesToDelete.push\_back(i);

}

**else**

**break**;

}

**return** indexesToDelete;

}

vector<Buyer> deleteDublicates(vector<Buyer> buyers) {

buyers = quickSort\_Name(buyers);

vector<**int**> indexesToDelete;

**for** (**int** i = 0; i < buyers.size(); i++) {

**if** (buyers[i].name == buyers[i + 1].name && buyers[i].address == buyers[i + 1].address)

indexesToDelete = linearSearch\_toDelete(buyers, i);

**while** (!indexesToDelete.empty()) {

buyers.erase(buyers.begin() + indexesToDelete[0]);

indexesToDelete.pop\_back();

}

}

**return** buyers;

}

**int** binarySearch\_Number(vector<Buyer> buyers, **int** low, **int** high, **int** number) {

**while** (high >= low) {

**int** mid = (low + high) / 2;

**if** (buyers[mid].number == number) {

**return** mid;

}

**if** (buyers[mid].number > number)

high = mid - 1;

**else**

low = mid + 1;

}

cout << "Запись не найдена" << endl;

**return** -1;

}

**void** outputToConsole(vector<Buyer> buyers) {

**int** lengthOfName = 0, lengthOfAddress = 0;

cout << setw(9) << "Number" << "|" << setw(20) << "Name" << "|"

<< setw(40)<< "Address" << "|" << setw(8)<< "Acounting date" << endl;

**for**(**int** i = 0; i < buyers.size(); i++) {

**for** (**int** j = 0; j < 95; j++)

cout << "\_";

cout << "\n";

cout << setw(9) << buyers[i].number << "|";

**if** (((buyers[i].name.substr(0,2) >= "а") && (buyers[i].name.substr(0,2) <= "я")) || (((buyers[i].name.substr(0,2) >= "А") && (buyers[i].name.substr(0,2) <= "Я")))) {

**int** countOfSpaces = 0;

**for** (**int** j = 0; j < buyers[i].name.length(); j++) {

**if** (buyers[i].name.substr(j,1) == " ")

countOfSpaces++;

}

lengthOfName = ((**int**)buyers[i].name.length() - countOfSpaces) / 2;

cout<< setw(20 + lengthOfName) << buyers[i].name << "|";

} **else** {

cout<< setw(20) << buyers[i].name << "|";

}

**if** (((buyers[i].address.substr(0,2) >= "а") && (buyers[i].address.substr(0,2) <= "я")) || (((buyers[i].address.substr(0,2) >= "А") && (buyers[i].address.substr(0,2) <= "Я")))) {

**int** countOfOtherSymbols = 0;

**for** (**int** j = 0; j < buyers[i].address.length(); j++) {

**if** (buyers[i].address.substr(j,1) == " " || buyers[i].address.substr(j,1) == "," || buyers[i].address.substr(j,1) == ".")

countOfOtherSymbols++;

}

lengthOfAddress = ((**int**)buyers[i].address.length() - countOfOtherSymbols) / 2;

cout<< setw(39 + lengthOfAddress) << buyers[i].address << "|";

} **else** {

cout<< setw(40) << buyers[i].address << "|";

}

cout << setw(12) << buyers[i].accountingDate << endl;

}

}

**void** outputToConsole(Buyer buyer) {

**int** lengthOfName = 0, lengthOfAddress = 0;

cout << setw(9) << "Number" << "|" << setw(20) << "Name" << "|"

<< setw(40)<< "Address" << "|" << setw(8)<< "Acounting date" << endl;

**for** (**int** j = 0; j < 95; j++)

cout << "\_";

cout << "\n";

cout << setw(9) << buyer.number << "|";

**if** (((buyer.name.substr(0,2) >= "а") && (buyer.name.substr(0,2) <= "я")) || (((buyer.name.substr(0,2) >= "А") && (buyer.name.substr(0,2) <= "Я")))) {

**int** countOfSpaces = 0;

**for** (**int** j = 0; j < buyer.name.length(); j++) {

**if** (buyer.name.substr(j,1) == " ")

countOfSpaces++;

}

lengthOfName = ((**int**)buyer.name.length() - countOfSpaces) / 2;

cout<< setw(20 + lengthOfName) << buyer.name << "|";

} **else** {

cout<< setw(20) << buyer.name << "|";

}

**if** (((buyer.address.substr(0,2) >= "а") && (buyer.address.substr(0,2) <= "я")) || (((buyer.address.substr(0,2) >= "А") && (buyer.address.substr(0,2) <= "Я")))) {

**int** countOfOtherSymbols = 0;

**for** (**int** j = 0; j < buyer.address.length(); j++) {

**if** (buyer.address.substr(j,1) == " " || buyer.address.substr(j,1) == "," || buyer.address.substr(j,1) == ".")

countOfOtherSymbols++;

}

lengthOfAddress = ((**int**)buyer.address.length() - countOfOtherSymbols) / 2;

cout<< setw(39 + lengthOfAddress) << buyer.address << "|";

} **else** {

cout<< setw(40) << buyer.address << "|";

}

cout << setw(12) << buyer.accountingDate << endl;

}

**void** searchByDate(vector<Buyer> buyers, string date) {

vector<Buyer> foundBuyers;

**for**(**int** i = 0; i < buyers.size(); i++) {

**if** (buyers[i].accountingDate == date)

foundBuyers.push\_back(buyers[i]);

}

**if** (foundBuyers.size() != 0)

outputToConsole(foundBuyers);

**else**

cout << "В тот день не было заказов" << endl;

}

**void** addToBuyersList(vector<Buyer> &buyers) {

Buyer temp;

cout << "Введите номер заказа" << endl;

cin >> temp.number;

fflush(stdin);

cout << "\nВведите ФИО заказчика" << endl;

getline(cin, temp.name);

cout << "\nВведите дату заказа (ДД.ММ.ГГГГ)" << endl;

getline(cin, temp.accountingDate);

cout << "\nВведите адрес заказчика" << endl;

getline(cin, temp.address);

buyers.push\_back(temp);

}

**void** deleteFromBuyersList(vector<Buyer> &buyers) {

**int** number;

cout << "Введите номер заказа для удаления" << endl;

cin >> number;

heapSort\_Number(buyers);

cout << "\nСписок отсортирован для поиска" << endl;

**if** (binarySearch\_Number(buyers, 0, (**int**)buyers.size() - 1, number) != -1) {

buyers.erase(buyers.begin() + binarySearch\_Number(buyers, 0, (**int**)buyers.size() - 1, number));

} **else** {

cout << "Запись с таким номером не обнаружена" << endl;

}

}

**int** main(**int** argc, **const** **char** \* argv[]) {

vector<Buyer> buyers;

buyers = getData();

**short** choise;

string dateForSearch;

**while** (**true**) {

cout << "Доступные операции:" << endl;

cout << "1 - Сортировка по номеру заказа" << endl;

cout << "2 - Сортировка по дате заказа" << endl;

cout << "3 - Сортировка по имени заказчика" << endl;

cout << "4 - Поиск заказа по номеру" << endl;

cout << "5 - Поиск заказов по дате" << endl;

cout << "6 - Вывести с учётом сортировки" << endl;

cout << "7 - Добавить запись" << endl;

cout << "8 - Удалить запись" << endl;

cout << "9 - Удалить дубликаты" << endl;

cout << "10 - Записать список покупателей в файл" << endl;

cout << "11 - Выход" << endl;

cout << "Введите номер операции:";

cin >> choise;

cout <<"\n";

**switch**(choise) {

**case** 1:

heapSort\_Number(buyers);

**break**;

**case** 2:

buyers = bubbleSort\_Date(buyers);

**break**;

**case** 3:

buyers = quickSort\_Name(buyers);

**break**;

**case** 4:

heapSort\_Number(buyers);

cout << "Список отсортирован для поиска" << endl;

cout << "Введите номер заказа для поиска:";

**int** num;

cin >> num;

cout << "\n";

outputToConsole(buyers[binarySearch\_Number(buyers, 0, (**int**)buyers.size() - 1, num)]);

**break**;

**case** 5:

cout << "Введите дату для поиска в формате ДД.ММ.ГГГГ:" << endl;

cin >> dateForSearch;

**while** (dateForSearch.length() != 10) {

cout << "Дата введена некорректно, введите ещё раз" << endl;

cin >> dateForSearch;

}

searchByDate(buyers, dateForSearch);

**break**;

**case** 6:

outputToConsole(buyers);

**break**;

**case** 7:

addToBuyersList(buyers);

**break**;

**case** 8:

deleteFromBuyersList(buyers);

**break**;

**case** 9:

buyers = deleteDublicates(buyers);

cout << "Дубликаты удалены" << endl;

**break**;

**case** 10:

putData(buyers);

cout << "Файлы записаны в выходной файл" << endl;

**break**;

**case** 11:

**return** 1;

**break**;

**default**:

cout << "Некорректный номер операции" << endl;

**break**;

}

}

**return** 0;

}

ВЕДОМОСТЬ ДОКУМЕНТОВ