Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет информационных технологий и управления

Кафедра вычислительных методов и программирования

Дисциплина: Основы алгоритмизации и программирования

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

на тему:

**«СТРУКТУРА СПИСКА МЕЖДУГОРОДНОЙ АТС»**

Выполнила: гр.32440X Фамилия И.О.

Проверила: Новицкая Л.И.

Минск 2024

**РЕФЕРАТ**

ТЕМА: структура списка междугородной АТС / И.О. Фамилия. – Минск : БГУИР, 2024, – п.з. – 57 с., рисунков – 13, источников – 5, приложений – 2.

*Объект исследования:* структура списка междугородной АТС.

*Предмет исследования:* программное обеспечение, обеспечивающее работу со структурой списка междугородной АТС.

*Цель* *курсовой работы*: разработка программного обеспечения, обеспечивающего работу со структурой списка междугородной АТС.

*Методология проведения работы*: в процессе разработки системы была разработана программная составляющая программного средства.

*Результаты работы*: выполнена постановка задачи и определены основные методы ее решения; описаны основные алгоритмы работы программного средства; выполнено тестирование программного средства, показавшее его соответствие функциональным требованиям, поставленным в задании на разработку. Программный продукт разработан на языке *C++* с применением *MS Visual Studio 2022*.

*Область применения результатов*: разработанная система может быть применена в таких сферах, как частные или государственные АТС. Она способствует эффективному управлению данными о звонках.

**СОДЕРЖАНИЕ**

[Введение 6](#_Toc167181861)

[1 Структуры и файлы 8](#_Toc167181862)

[1.1 Структуры 8](#_Toc167181863)

[1.2 Файлы 9](#_Toc167181864)

[2 Алгоритмы сортировки 12](#_Toc167181865)

[3 Алгоритмы поиска 16](#_Toc167181866)

[4 Пользовательские функции 21](#_Toc167181867)

[5 Описание работы программы 27](#_Toc167181868)

[Заключение 33](#_Toc167181869)

[Список использованных источников 34](#_Toc167181870)

[Приложение А (обязательное) Листинг кода 35](#_Toc167181871)

[Приложение Б (обязательное) Блок-схема работы программы 56](#_Toc167181872)

Введение

Современная жизнь тесно связана с процессами информатизации и компьютеризации, которые активно внедряются во все сферы человеческой деятельности. Одним из таких направлений является работа междугородных автоматических телефонных станций (АТС). Использование программного обеспечения для управления списками звонков на междугородных АТС позволяет эффективно организовать процесс обработки вызовов, улучшить качество обслуживания абонентов и оптимизировать внутренние процессы.

С применением современных информационных технологий АТС получают возможность более точно и оперативно обрабатывать данные о совершаемых звонках. Это способствует улучшению управления ресурсами и повышению конкурентоспособности компании в сфере телекоммуникаций.

Программные решения служат для систематического хранения, управления, обработки и извлечения больших объемов данных. Они находят применение в различных областях, выполняя такие функции, как:

* централизованное хранение данных: программы позволяют собирать данные в одном месте, что облегчает доступ к ним и их управление;
* обеспечение безопасности: программы защищают хранимую информацию с помощью систем контроля доступа и методов шифрования;
* поддержка транзакций: программы обеспечивают выполнение последовательных шагов транзакций, гарантируя их корректное выполнение или откат в случае ошибки;
* управление данными: программы предоставляют возможности для организации и актуализации данных, включая их добавление, обновление и удаление;
* аналитика: программы упрощают анализ данных с помощью запросов, отчетов и средств визуализации;
* поиск и фильтрация: программы позволяют быстро находить и отбирать нужную информацию;
* масштабируемость и производительность: программы могут расширяться для обработки растущего объема данных и обеспечения доступа большого числа пользователей без значительного снижения производительности;
* восстановление данных: программы обеспечивают сохранность информации и ее восстановление после сбоев или ошибок системы с помощью резервного копирования и журналирования.

Существует два основных подхода к взаимодействию с данными:

* прямое изменение и поиск в файле на диске: использование методов работы с файловыми структурами может замедлить обработку данных из-за медленного доступа к дискам, однако такой подход не ограничивает использование оперативной памяти;
* загрузка данных в массив структур для работы в оперативной памяти: этот способ ускоряет обработку за счет более быстрого доступа к данным, однако требует больше памяти.

Изучение структур и алгоритмов сортировки и поиска играет ключевую роль при разработке программы для управления вызовами на междугородной АТС. Данные о звонках включают в себя номер абонента, длительность разговора и другие параметры.

Цель данного курсового проекта заключается в изучении методов сортировки и поиска данных о междугородных вызовах, а также в разработке программного обеспечения на языке C++, соответствующего индивидуальному заданию. В рамках проекта будет реализована функциональность для вывода информации о вызовах по каждому городу в порядке убывания или возрастания их длительности.

Индивидуальное задание представляет собой следующее: на междугородной АТС информация о разговорах содержит дату разговора, код и название города, время разговора, тариф, номер телефона в этом городе и номер телефона абонента. Вывести по каждому городу общее время разговоров с ним и сумму. Ключ: общее время разговоров.

Автоматизация процессов управления данными о вызовах на междугородных АТС значительно повышает эффективность работы предприятия и упрощает анализ телекоммуникационных процессов. Ожидается, что разработанная программа существенно ускорит и упростит процесс анализа вызовов на междугородных АТС.

1 Структуры и файлы

1.1 Структуры

В C++ структуры являются способом объединения переменных различных типов под единым именем. Это полезно для более удобного управления данными и передачи их между функциями. Структуры определяются с использованием ключевого слова struct, за которым следует имя структуры и список её членов в фигурных скобках. Поля структуры доступны через оператор точка [1].

В этом проекте для хранения информации о междугородних телефонных звонках и городах используются две структуры. Их формат представлен ниже:

struct phone\_call

{

unsigned id; // Уникальный идентификатор звонка

string date; // Дата звонка

unsigned city\_code; // Код города

string city\_name; // Название города

unsigned long long time; // Длительность звонка в минутах

double tarif; // Тариф за звонок

string city\_phone\_number; // Номер телефона в городе

string phone\_number; // Номер телефона абонента

};

struct city

{

string name; // Название города

unsigned code; // Код города

unsigned long long time; // Общее время разговоров в минутах

};

Структура phone\_call хранит информацию о каждом телефонном звонке, включая идентификатор, дату, код и название города, время звонка, тариф, городской номер телефона и номер телефона абонента. Структура city используется для хранения информации о городах, таких как название, код и общее время звонков.

Для работы с этими структурами в данном проекте используются векторы. Вектор в C++ – это динамический массив, предоставляемый стандартной библиотекой шаблонов (STL). Вектор автоматически управляет своей памятью и может изменять свой размер при добавлении или удалении элементов. Вектор хранит элементы в непрерывной области памяти, что позволяет эффективно обращаться к элементам [2].

Векторы, используемые в проекте, выглядят следующим образом:

vector<phone\_call> phone\_calls;

vector<city> cities;

Этот подход позволяет легко управлять данными о телефонных звонках и городах. Доступ к элементам вектора осуществляется с помощью оператора квадратных скобок ([]), а доступ к полям структуры — с помощью оператора точки (.).

1.2 Файлы

В программировании файлы представляют собой хранилища данных на компьютере или других устройствах. Они могут содержать различную информацию, например, данные о продуктах, их количестве, номерах цехов и идентификаторах. Файлы имеют расширения, которые указывают на тип данных внутри. Эти файлы могут быть открыты, изменены, перемещены, скопированы или удалены при помощи различных программ и приложений [3].

В этом проекте используются два файла для хранения информации: city\_list.txt, содержащий данные о городах и их кодах, и database.txt, в котором сохраняются все записи о телефонных звонках. Структура данных в этих файлах позволяет удобно организовать и управлять информацией.

Файл city\_list.txt хранит информацию о существующих на АТС городах и их кодах. Каждая строка файла представляет собой отдельную запись о городе, включающую его название и код (см. рисунок 1.1).

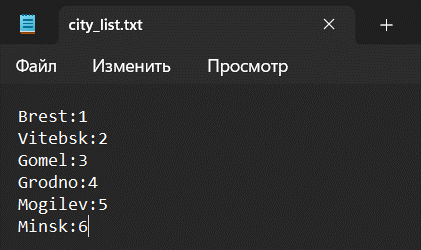


Рисунок 1.1 – Внешний вид файла city\_list.txt.

Файл database.txt содержит данные о произведённых звонках. Формат этого файла позволяет хранить записи о каждом звонке, включая уникальный идентификатор, дату, код города, название города, длительность звонка, тариф, городской номер телефона и номер телефона абонента (см. рисунок 1.2).

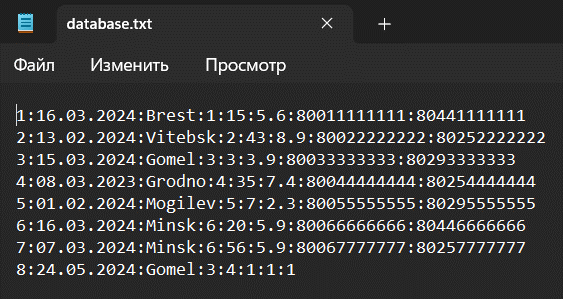


Рисунок 1.2 – Внешний вид файла database.txt.

Для работы с файлами в C++ используется класс fstream из библиотеки <fstream>. В отличие от классов ifstream и ofstream, предназначенных для чтения и записи данных соответственно, fstream объединяет их функциональность, позволяя одновременно читать и записывать данные в файл.

Создание объекта класса fstream и ассоциация его с файлом на диске позволяет производить операции ввода-вывода. Например, можно открыть файл для чтения, извлечь данные и затем закрыть его. Аналогично, файл можно открыть для записи, внести необходимые изменения и затем сохранить их.

Этот подход позволяет эффективно работать с данными, обеспечивая их сохранение и восстановление из файлов. Файлы city\_list.txt и database.txt обеспечивают надежное хранение информации о городах и телефонных звонках, позволяя программе легко обновлять и обрабатывать данные.

2 Алгоритмы сортировки

Сортировка – это процесс упорядочивания элементов массива или другого набора данных по определённому критерию, что упрощает последующий поиск информации. Сортировка называется устойчивой, если при её выполнении элементы с одинаковыми ключами остаются в том же порядке, в каком они были изначально. Важно, чтобы при сортировке изменения производились непосредственно в исходном массиве без использования дополнительных массивов. Сортировки делятся на внутренние, выполняемые в оперативной памяти, и внешние, использующие файловые структуры [4].

Методы внутренней сортировки классифицируются по времени их выполнения и оцениваются по количеству операций сравнения и перестановки элементов. Простые методы сортировки подходят для небольших наборов данных (до 50 элементов), так как они легко реализуются и быстро выполняются. Для больших наборов данных (100 и более элементов) предпочтительнее использовать более сложные алгоритмы, которые хотя и требуют более сложной реализации, но обеспечивают лучшую производительность.

Рассмотрим наиболее известные простые методы сортировки.

Пузырьковая сортировка – это базовый алгоритм, который многократно проходит по списку элементов, сравнивая соседние элементы попарно и меняя их местами, если они расположены в неправильном порядке. На каждом проходе самый большой (или самый маленький) элемент "всплывает" на своё место, наподобие пузырька в воде, что и дало название алгоритму. В данной курсовой работе была использована эта сортировка. Ниже представлен код, реализующий пузырьковую сортировку в курсовой работе:

void bubble\_sort(int type)

{

for (int i = 0; i < cities.size(); i++)

{

for (int j = 0; j < cities.size() - i - 1; j++)

{

if (type == 1)

{

if (cities[j].time > cities[j+1].time)

{

city temp = cities[j];

cities[j] = cities[j + 1];

cities[j + 1] = temp;

}

}

else

{

if (cities[j].time < cities[j + 1].time)

{

city temp = cities[j];

cities[j] = cities[j + 1];

cities[j + 1] = temp;

}

}

}

}

}

Сортировка выбором – в этом алгоритме на каждом шаге выбирается минимальный элемент из оставшихся несортированных элементов и помещается на текущую позицию. Алгоритм последовательно выбирает наименьший (или наибольший) элемент из несортированной части массива и меняет его местами с первым элементом этой части. Этот процесс повторяется до тех пор, пока весь массив не будет отсортирован. Ниже представлен код, реализующий сортировку выбором в курсовой работе:

void selection\_sort(int type)

{

for (int i = 0; i < cities.size(); i++)

{

int m = i;

for (int j = i + 1; j < cities.size(); j++)

{

if (type == 1)

{

if (cities[j].time < cities[m].time) m = j;

}

else

{

if (cities[j].time > cities[m].time) m = j;

}

}

city temp = cities[m];

cities[m] = cities[i];

cities[i] = temp;

}

}

Сортировка вставками – этот метод упорядочивает массив, поочередно вставляя каждый элемент на его правильное место. Алгоритм начинается с первого элемента и последовательно проходит по массиву, вставляя каждый элемент в соответствующее место в уже отсортированной части массива.

Шейкерная сортировка – это улучшенная версия пузырьковой сортировки, при которой элементы перемещаются в обоих направлениях. В отличие от стандартной пузырьковой сортировки, шейкерная сортировка проходит по массиву в обоих направлениях, что ускоряет процесс сортировки.

К более сложным методам сортировки относятся:

* сортировка Шелла, предложенная Дональдом Шеллом в 1959 году, улучшает сортировку вставками за счёт сравнения и перемещения элементов на определённые расстояния друг от друга, которые постепенно уменьшаются до тех пор, пока массив не будет окончательно отсортирован;
* сортировка кучей (HeapSort), разработанная Дж. Уильямсом в 1964 году, использует структуру данных "куча", которая представляет собой бинарное дерево, где каждый узел имеет значение большее или равное значениям его потомков. Максимальный элемент удаляется из кучи и помещается в конец массива, после чего куча перестраивается. Этот процесс повторяется, пока весь массив не будет отсортирован;
* быстрая сортировка (QuickSort), созданная Чарльзом Хоаром в 1962 году, является одним из самых эффективных алгоритмов сортировки. Этот метод, который использован в данной курсовой работе, выбирает опорный элемент, разделяет массив на две части — элементы, меньшие опорного, и элементы, большие опорного. Далее сортировка рекурсивно применяется к каждой части, пока весь массив не будет отсортирован.

Ниже приведен код, который реализует эту сортировку:

void quick\_sort(int low, int high, int type)

{

if (low < high)

{

unsigned x = cities[(low + high) / 2].time;

int i = low, j = high;

while (i <= j)

{

if (type == 1)

{

while (cities[i].time < x) i++;

while (cities[j].time > x) j--;

}

else

{

while (cities[i].time > x) i++;

while (cities[j].time < x) j--;

}

if (i <= j)

{

city temp = cities[i];

cities[i] = cities[j];

cities[j] = temp;

i++;

j--;

}

}

if (low < j) quick\_sort(low, j, type);

if (i < high) quick\_sort(i, high, type);

}

}

Сравнение различных методов сортировки показывает, что при работе с массивами, содержащими более 100 элементов, пузырьковая сортировка оказывается наименее эффективной. В то время как QuickSort работает быстрее HeapSort в 2-3 раза и сортировки Шелла в 3-7 раз, что делает его предпочтительным выбором для больших объемов данных.

3 Алгоритмы поиска

Для поиска нужного элемента в массиве структур типа Ttype, заданного следующим образом:

struct Ttype

{

type key; // ключевое поле типа type

// . . . другие компоненты структуры

} \*a; // указатель на динамический массив таких структур

Дл нам необходимо найти индекс i\_key, такой что a[i\_key].key = f\_key, где key — это интересующее нас поле структуры, а f\_key — значение ключа, которое мы ищем. Найдя индекс i\_key, можно получить доступ к другим полям искомой структуры, используя a[i\_key].

Рассмотрим три метода поиска, используемые в данной курсовой работе: линейный, бинарный и интерполяционный поиск [5].

Линейный поиск используется, когда отсутствует информация о структуре данных. Он заключается в последовательном просмотре каждого элемента массива до нахождения искомого ключа или до достижения конца массива, если ключ не найден. В случае отсутствия уникальности ключа необходимо проверять все элементы массива.

Пример кода для линейного поиска:

void find\_linear(unsigned long long time\_to\_find)

{

int code = 0;

for (int i = 0; i < cities.size(); i++)

{

if (cities[i].time == time\_to\_find)

{

code = 1;

cout << cities[i].name << ' ';

}

}

if (code == 0)

cout << "нет городов с таким временем!\n";

else

cout << '\n';

}

Бинарный поиск применяется для отсортированных массивов и значительно сокращает время поиска. Алгоритм делит массив пополам и сравнивает целевой элемент с серединным элементом массива. Процесс поиска включает следующие шаги:

* определение середины отсортированного массива;
* сравнение целевого значения ключа с ключом серединного элемента;
* если ключи совпадают, поиск завершается;
* если целевое значение ключа меньше ключа серединного элемента, поиск продолжается в левой половине массива;
* если целевое значение ключа больше ключа серединного элемента, поиск продолжается в правой половине массива.

Эти шаги повторяются до тех пор, пока не будет найден элемент с нужным ключом или подмассив не станет пустым. Бинарный поиск достигает логарифмической временной сложности O(log n), что делает его значительно быстрее линейного поиска на больших объемах данных.

Пример кода для бинарного поиска:

void find\_binary(unsigned long long time\_to\_find)

{

selection\_sort(1);

int i\_key = 0, j = cities.size() - 1;

while (i\_key < j)

{

const int m = (i\_key + j) / 2;

if (cities[m].time < time\_to\_find) i\_key = m + 1;

else j = m;

}

if (cities[i\_key].time != time\_to\_find)

{

cout << "нет городов с таким временем!\n";

return;

}

cout << cities[i\_key].name << ' ';

for (int i = i\_key + 1; i < cities.size(); i++)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

for (int i = i\_key - 1; i >= 0; i--)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

cout << '\n';

}

Интерполяционный поиск используется для нахождения определённого значения в отсортированном массиве и является улучшенной версией бинарного поиска. Этот алгоритм основывается на интерполяции, которая предсказывает местоположение искомого значения, используя его числовое значение.

Основная идея заключается в том, что данные в массиве воспринимаются как равномерно распределённые, и используется интерполяционная формула для определения позиции искомого элемента:

pos = lo + ((hi - lo) / (A[hi] - A[lo])) \* (x - A[lo])

где:

pos – предполагаемая позиция искомого элемента.

lo и hi – индексы первого и последнего элементов части массива, в которой выполняется поиск.

A[] – массив, в котором выполняется поиск.

x – значение, которое необходимо найти.

Процесс поиска включает следующие шаги:

* вычисление предполагаемой позиции pos с использованием интерполяционной формулы;
* сравнение значения в позиции pos с искомым значением x;
* если значения равны, поиск завершается;
* если значение в позиции pos меньше x, поиск продолжается в верхней части массива;
* если значение в позиции pos больше x, поиск продолжается в нижней части массива.

Процесс повторяется до тех пор, пока не будет найден искомый элемент или область поиска не станет пустой. Интерполяционный поиск особенно эффективен для массивов с равномерно распределёнными значениями и может достигать временной сложности, близкой к O(log log n), что значительно быстрее, чем O(log n) у бинарного поиска. Однако при неоднородном распределении значений его эффективность может снижаться.

В данной курсовой работе был использован модифицированный вариант интерполяционного поиска: если ключ элемента с позицией pos меньше ключа x, то lo сдвигается на одну позицию вправо и поиск продолжается; в противном случае проверяются поля найденного элемента на соответствие условиям — при успешной проверке найденный элемент выводится на экран, затем lo сдвигается на одну позицию вправо и поиск продолжается. Такая модификация необходима, так как условие поиска нестрогое: может существовать несколько элементов с нужным количеством мест, подходящим временем и пунктом назначения.

Пример кода для интерполяционного поиска:

void find\_interpolation(unsigned long long time\_to\_find)

{

selection\_sort(1);

int low = 0;

int high = cities.size() - 1;

while (low <= high && time\_to\_find >= cities[low].time && time\_to\_find <= cities[high].time)

{

if (low == high)

{

if (cities[low].time == time\_to\_find)

{

cout << cities[low].name << ' ';

for (int i = low + 1; i < cities.size(); i++)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

for (int i = low - 1; i >= 0; i--)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].time << ' ';

}

cout << '\n';

return;

}

cout << "нет городов с таким временем!\n";

return;

}

int pos = low + (high - low) / (cities[high].time - cities[low].time) \* (time\_to\_find - cities[low].time);

if (cities[pos].time == time\_to\_find)

{

cout << cities[pos].name << ' ';

for (int i = pos + 1; i < cities.size(); i++)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].time << ' ';

}

for (int i = pos - 1; i >= 0; i--)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

cout << '\n';

return;

}

if (cities[pos].time < time\_to\_find)

low = pos + 1;

else

high = pos - 1;

}

cout << "нет городов с таким временем!\n";

}

4 Пользовательские функции

В работе реализовано и использовано 22 пользовательских функций, не считая функцию main(). Ниже приведено объяснение назначения каждой функции.

bool check\_int\_input(string input) – функция, которая отвечает за правильный ввод переменных целочисленного типа данных. Вызывается в случае того, когда необходимо ввести переменную. Принимает в себя строку, которая является предположительно введённым целым числом. Работает следующим образом:

bool check\_int\_input(string input)

{

try

{

int temp = stoi(input);

}

catch (invalid\_argument)

{

return false;

}

return true;

}

Возвращает true, если введённая строка является числом, или false, если введённая строка не является числом.

void convert\_int\_to\_unsigned\_long\_long(int input, unsigned long long& result) – функция, которая отвечает за правильный перевод из целочисленного типа данных в расширенный целочисленный неотрицательный тип данных. Вызывается в случае того, когда необходимо перевести int в unsigned long long. Принимает в себя число, которое надо перевести и ссылка на итоговую переменную. Работает следующим образом:

void convert\_int\_to\_unsigned\_long\_long(int input, unsigned long long& result)

{

if (input < 0)

input \*= -1;

result = input;

}

Не возвращает никаких значений.

void convert\_int\_to\_unsigned(int input, unsigned& result) – функция, которая отвечает за правильный перевод из целочисленного типа данных в целочисленный неотрицательный тип данных. Вызывается в случае того, когда необходимо перевести int в unsigned. Принимает в себя число, которое надо перевести и ссылка на итоговую переменную. Работает следующим образом:

void convert\_int\_to\_unsigned(int input, unsigned& result)

{

if (input < 0)

input \*= -1;

result = input;

}

Не возвращает никаких значений.

bool check\_city\_input(string info, string& city\_name, unsigned& city\_code) – функция, которая отвечает за правильный ввод города. Вызывается в случае того, когда необходимо ввести город. Принимает в себя две строки и ссылку неотрицательного целочисленного типа данных, которые являются вспомогательными при вводе города. Возвращает true, если введённый город существует, или false, если такого города нет.

bool check\_date\_input(string date) – функция, которая отвечает за правильный ввод даты. Вызывается в случае того, когда необходимо ввести дату. Принимает в себя строку, которая является предположительно введённой датой. Работает следующим образом:

bool check\_date\_input(string date)

{

for (int i = 0; i < date.length(); i++)

{

char c = date[i];

if (!isdigit(c) && c != '.')

return false;

if (c == '.' && i != 2 && i != 5 && i != 10)

return false;

}

return true;

}

Возвращает true, если введённая строка является датой в правильном формате, или false, если введённая строка не является таковой.

end\_of\_function() – функция, которая позволяет выполнить запрос нажатия любой клавиши и очистки консоли. Работает следующим образом:

void end\_of\_function()

{

system("pause");

system("cls");

}

Не возвращает никаких значений.

void create\_new\_phone\_call() – функция создания нового звонка. Запрашивает все необходимые поля для нового звонка, после чего добавляет новый звонок в вектор. Не возвращает никаких значений.

void set\_new\_ids(unsigned id\_to\_delete) – функция, которая является вспомогательной функцией при удалении звонка. На вход принимает значение идентификатора звонка, который удаляется из списка. Работает следующим образом:

void set\_new\_ids(unsigned id\_to\_delete)

{

for (int i = 0; i < phone\_calls.size(); i++)

if (phone\_calls[i].id > id\_to\_delete)

phone\_calls[i].id--;

}

Не возвращает никаких значений.

void print\_phone\_calls() – функция форматированного вывода названия программы и списка звонков на экран. Не возвращает никаких значений.

void print\_phone\_calls\_to\_file() – функция для записи вектора звонков в форматированном виде в файл. Все поля структуры записываются через ‘:’, чтобы не было проблем с чтением структуры из файла. Работает следующим образом:

void print\_phone\_calls\_to\_file()

{

fstream file("database.txt", ios::app);

for (int i = 0; i < phone\_calls.size(); i++)

file << phone\_calls[i].id << ':' << phone\_calls[i].date << ':' << phone\_calls[i].city\_name << ':' << phone\_calls[i].city\_code << ':' << phone\_calls[i].time << ':' << phone\_calls[i].tarif << ':' << phone\_calls[i].city\_phone\_number << ':' << phone\_calls[i].phone\_number << '\n';

}

Не возвращает никаких значений.

void read\_from\_file() – функция для чтения заранее отформатированного файла структур в вектор. Эта функция и функция выше были созданы для сохранения данных о векторе звонков при закрытии программы для последующего перезахода. Не возвращает никаких значений.

void clear\_file() – функция для очистки файла со звонками. Реализована следующим образом:

void clear\_file()

{

fstream file("database.txt", ofstream::out | ofstream::trunc);

file.close();

}

Не возвращает никаких значений.

void create\_city\_vector() – функция для создания вектора городов. Считывает все города из файла с ними, после чего проверяет вектор со звонками и высчитывает общее время разговоров в этом городе. В итоге добавляет все города в вектор. Не возвращает никаких значений.

void delete\_phone\_call() – функция для удаления звонка из списка. Запрашивает идентификатор продукции на удаление, после чего проверяет, существует ли такой идентификатор и, если существует, то удаляет звонок с таким идентификатором и меняет идентификатор остальных звонков, чтобы избежать разрыва. Не возвращает никаких значений.

void bubble\_sort(int type) – функция для сортировки вектора городов по общему времени разговоров методом пузырька. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Не возвращает никаких значений.

void selection\_sort(int type) – функция для сортировки вектора городов по общему времени разговоров методом выбора. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Не возвращает никаких значений.

void quick\_sort(int low, int high, int type) – функция для сортировки вектора городов по количеству методом быстрой сортировки. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Также принимает в себя параметры нижнего индекса и верхнего индекса, которые существуют в связи с тем, что метод быстрой сортировки является рекурсивным. Не возвращает никаких значений.

void sort\_cities() – функция для запроса информации о желаемой сортировки городов. Запрашивает информацию о том, сортировать ли вектор по возрастанию или по убыванию, а также о желаемом методе сортировки (пузырьком, прямым выбором или быстрой). Не возвращает никаких значений.

void find\_linear(unsigned long long time\_to\_find) – функция для проведения линейного поиска в векторе городов. На вход принимает значение времени разговоров, по которому нужно искать продукцию. В случае, если время разговоров было найдено, то выводит идентификатор города на экран, в противном случае выводит на экран сообщение об отсутствии таковых городов. Не возвращает никаких значений.

void find\_binary(unsigned long long time\_to\_find) – функция для проведения бинарного поиска в векторе городов. На вход принимает значение времени разговоров, по которому нужно искать продукцию. Перед началом поиска сортирует вектор, о чём уведомляет пользователя. В случае, если время разговоров было найдено, то выводит идентификатор города на экран, в противном случае выводит на экран сообщение об отсутствии таковых городов. Не возвращает никаких значений.

void find\_interpolation(unsigned long long time\_to\_find) – функция для проведения интерполяционного поиска в векторе городов. На вход принимает значение времени разговоров, по которому нужно искать продукцию. Перед началом поиска сортирует вектор, о чём уведомляет пользователя. В случае, если время разговоров было найдено, то выводит идентификатор города на экран, в противном случае выводит на экран сообщение об отсутствии таковых городов. Не возвращает никаких значений.

void find() – функция для запроса информации о желаемом поиске по времени разговоров. Запрашивает информацию о том, по какому значению количества нужно искать, а также о желаемом методе поиска (линейным, бинарным, интерполяционным). Не возвращает никаких значений.

5 Описание работы программы

Программа представляет собой систему управления списком звонков на междугородной АТС. Код программы содержит структуру phone\_call, которая описывает телефонный звонок, а также структуру city, которая описывает город.

При запуске программы, на экран пользователя выводится меню, в котором он может работать с программой при помощи выбора одного из пунктов меню, а именно:

* добавить запись;
* удалить запись;
* очистить список;
* отсортировать города по времени разговоров;
* вернуть города в исходный вид;
* найти город по времени разговоров;
* выход из программы.

Скриншот меню показан на рисунке 5.1.

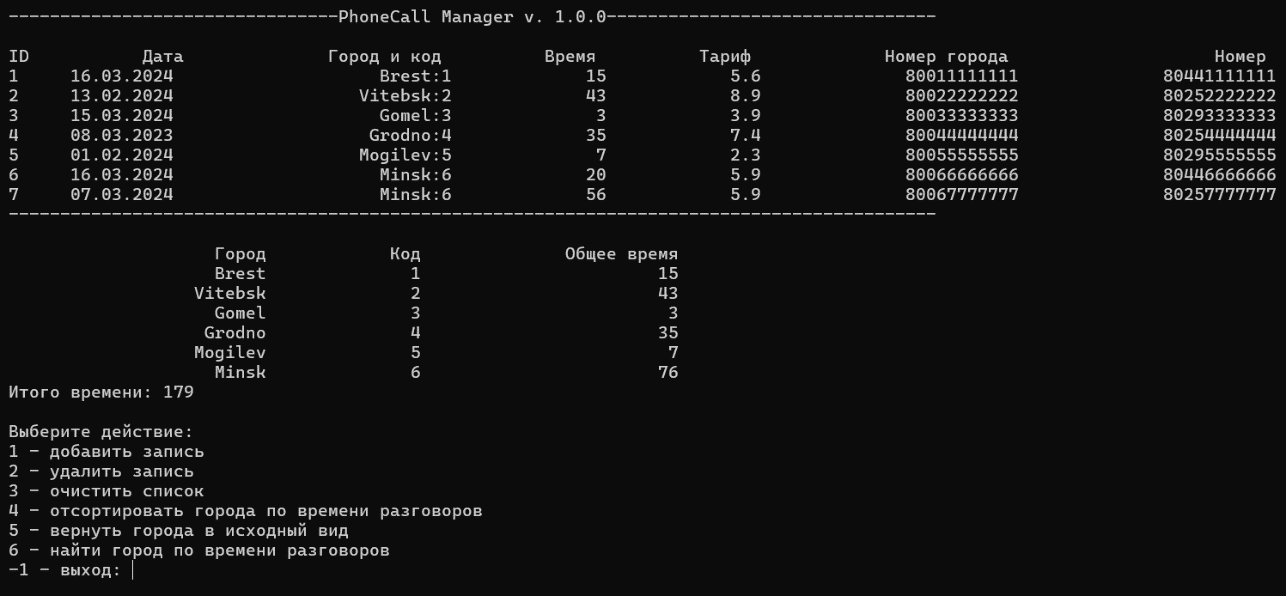


Рисунок 5.1 – Меню программы

При проведении каких-либо операций, изменяющих информацию о звонках, изменения производятся как в векторе структур (в окне приложения соотв.), так и в текстовом файле, содержащем информацию, необходимую для вывода, считывания и сохранения информации, посредством переписывания файла.

Выбор пункта “Добавить запись” запросит список данных, необходимых для создания новой структуры звонка (дату звонка в формате дд.мм.гггг, имя города или его код (в зависимости от того что будет введено второе будет добавлено автоматически), время разговора в минутах, тариф, телефонный номер города, телефонный номер). Идентификатор присваивается звонку автоматически, основываясь на размере нынешнего вектора структур. Новый звонок автоматически добавляется в конец вектора, после чего пользователь возвращается в главное меню программы (см. рисунок 5.2).

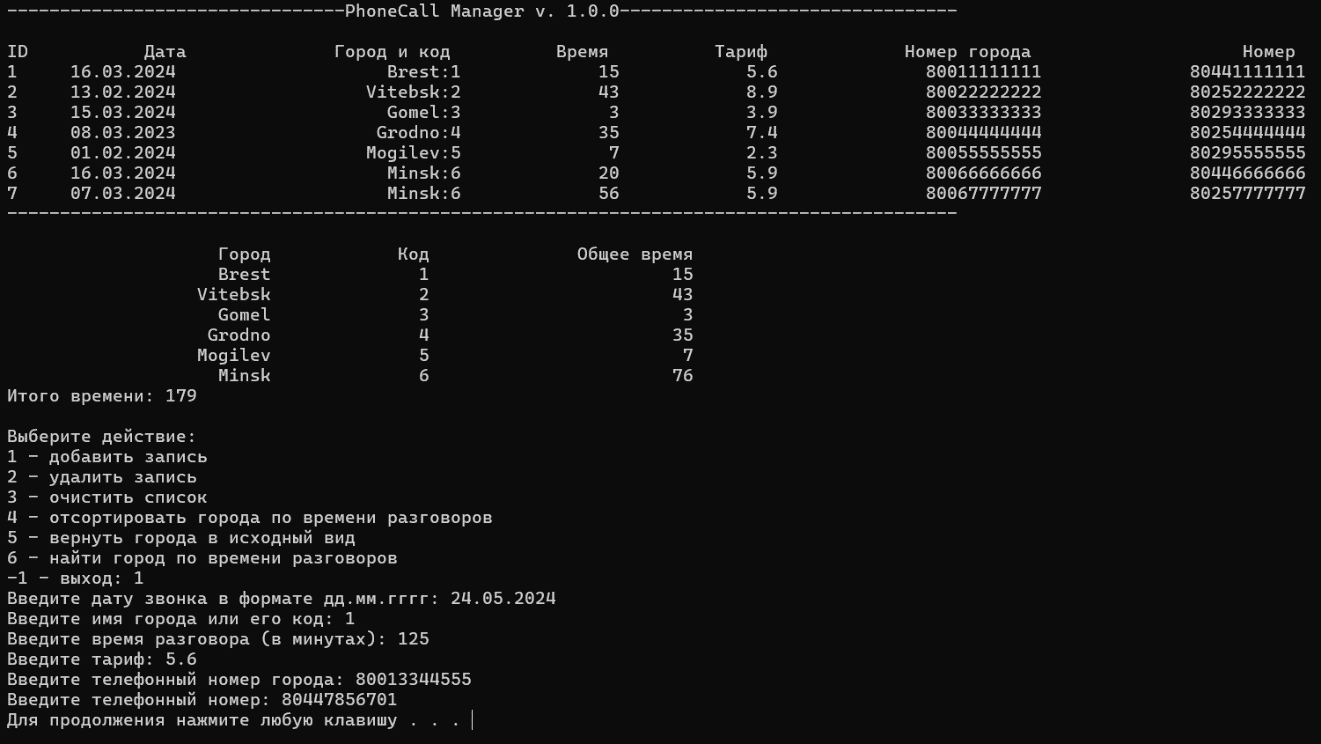


Рисунок 5.2 – Пункт “Добавить запись”

При выборе пункта “Удалить запись”, пользователя попросят ввести идентификатор звонка, который он хочет удалить. После этого звонок будет удален, а идентификаторы остальных звонков будут обновлены. Список звонков обновится автоматически. Если список звонков пустой, то пользователю выведется сообщение об этом. По завершении пользователь вернётся обратно в главное меню (см. рисунок 5.3).

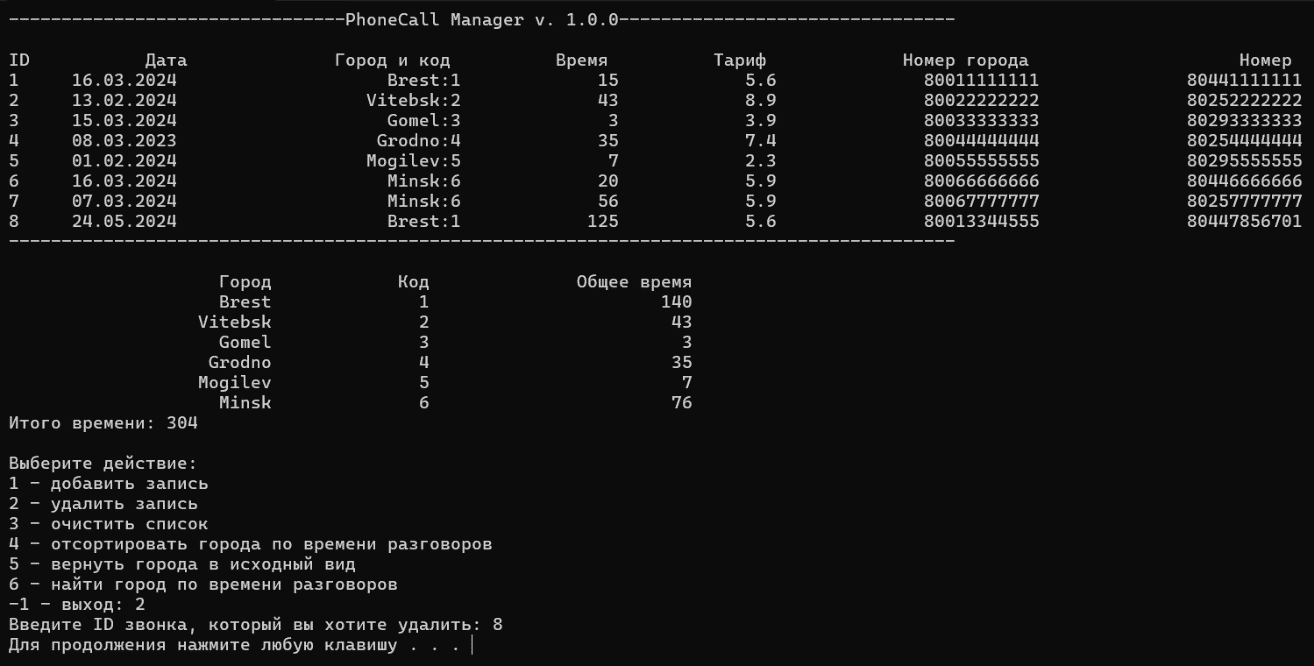


Рисунок 5.3 – Пункт “Удалить запись”

Если пользователь выбирает пункт “Очистить список”, то весь список звонков станет пустым. Если список звонков был и так пустым, то пользователю выведется сообщение об этом. После очищения списка продукции пользователя вернёт обратно в главное меню (см. рисунок 5.4).

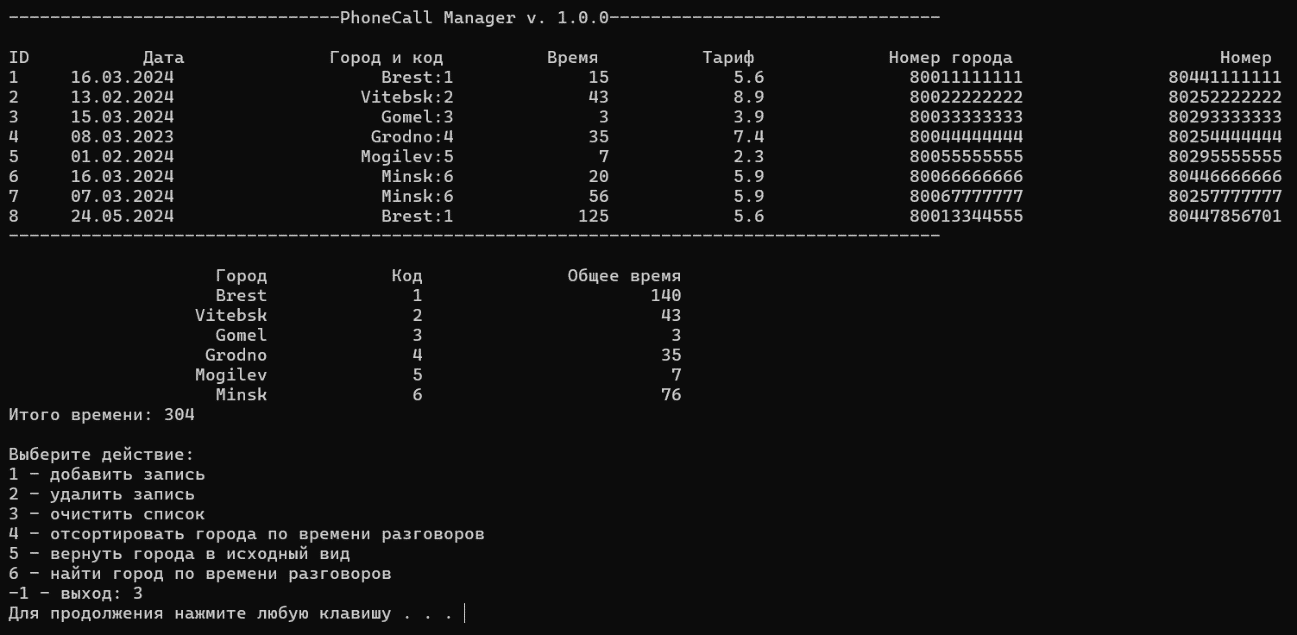


Рисунок 5.4 – Пункт “Очистить список”

При выборе пункта “Отсортировать города по времени разговоров” ему будет предложено выбрать способ сортировки, а также её тип (по возрастанию, по убыванию). Всего существует 3 способа сортировки, а именно:

* сортировка пузырьком;
* сортировка прямым выбором;
* быстрая сортировка.

После выбора способа и типа сортировки будет произведена сортировка списка городов по времени разговоров в зависимости от выбранных параметров, а пользователя вернёт обратно в главное меню (см. рисунки 5.5 и 5.6).

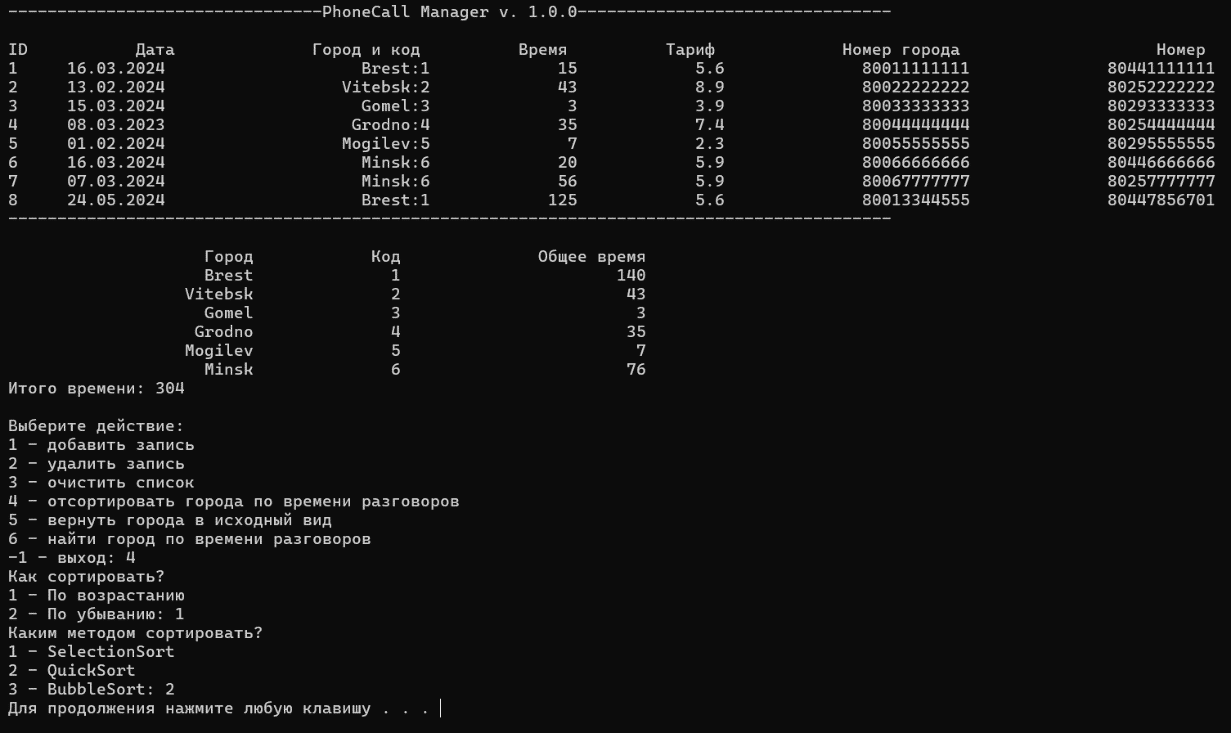


Рисунок 5.5 – Пункт “Отсортировать города по времени разговоров”

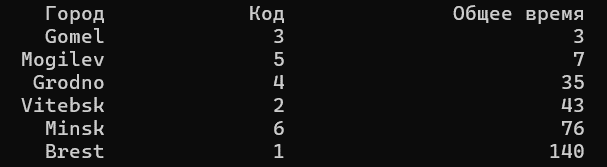


Рисунок 5.6 – Отсортированный список по времени разговоров

При выборе пункта “Вернуть города в исходный вид”, то вектор городов будет автоматически отсортирован по возрастанию кода города, что является исходным видом для вектора городов, а пользователя вернёт обратно в главное меню (см. рисунки 5.7 и 5.8).

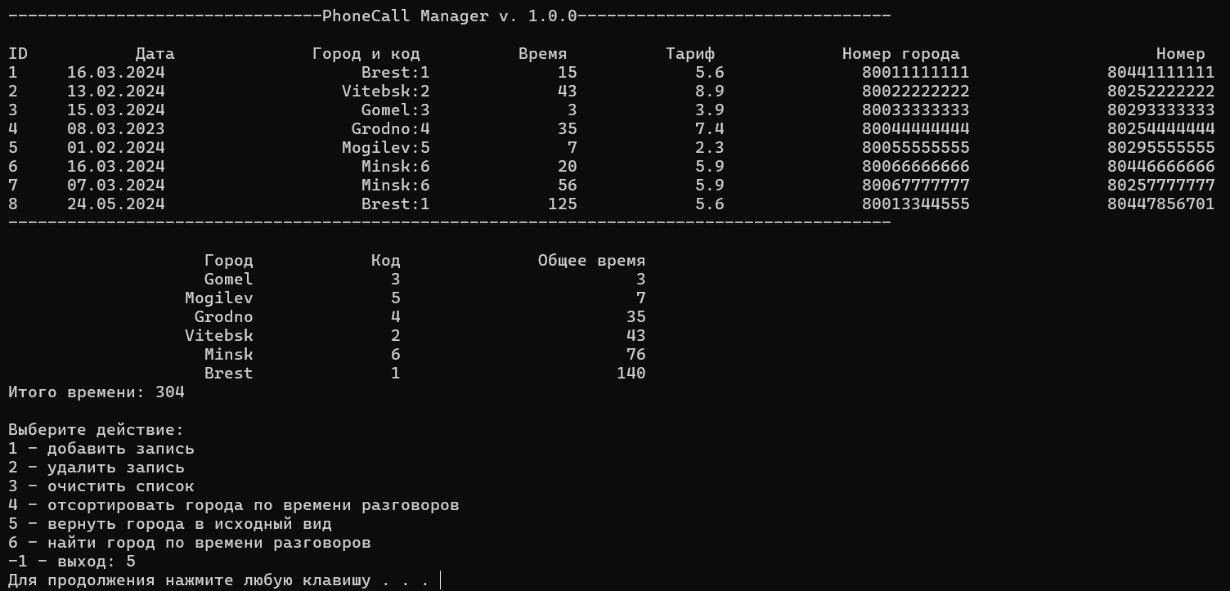


Рисунок 5.7 – Пункт “Вернуть города в исходный вид”

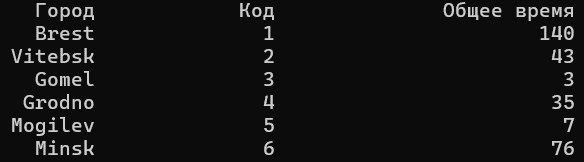


Рисунок 5.8 – Исходный вид вектора городов

При выборе пункта “Найти город по времени разговоров”, пользователь должен ввести значение времени разговоров, по которому он хочет найти город. Затем пользователю будет предложено выбрать один из следующих методов поиска:

* линейный поиск;
* двоичный поиск (список будет автоматически отсортирован по возрастанию, о чём будет выведено соответствующее сообщение);
* интерполяционный поиск (список будет автоматически отсортирован по возрастанию, о чём будет выведено соответствующее сообщение).

По окончании поиска на экран будут выведены все города, которые были найдены по запросу пользователя. Если город не был найден, то будет выведено соответствующее сообщение. После этого пользователя перенесёт обратно в главное меню (см. рисунки 10 и 11).

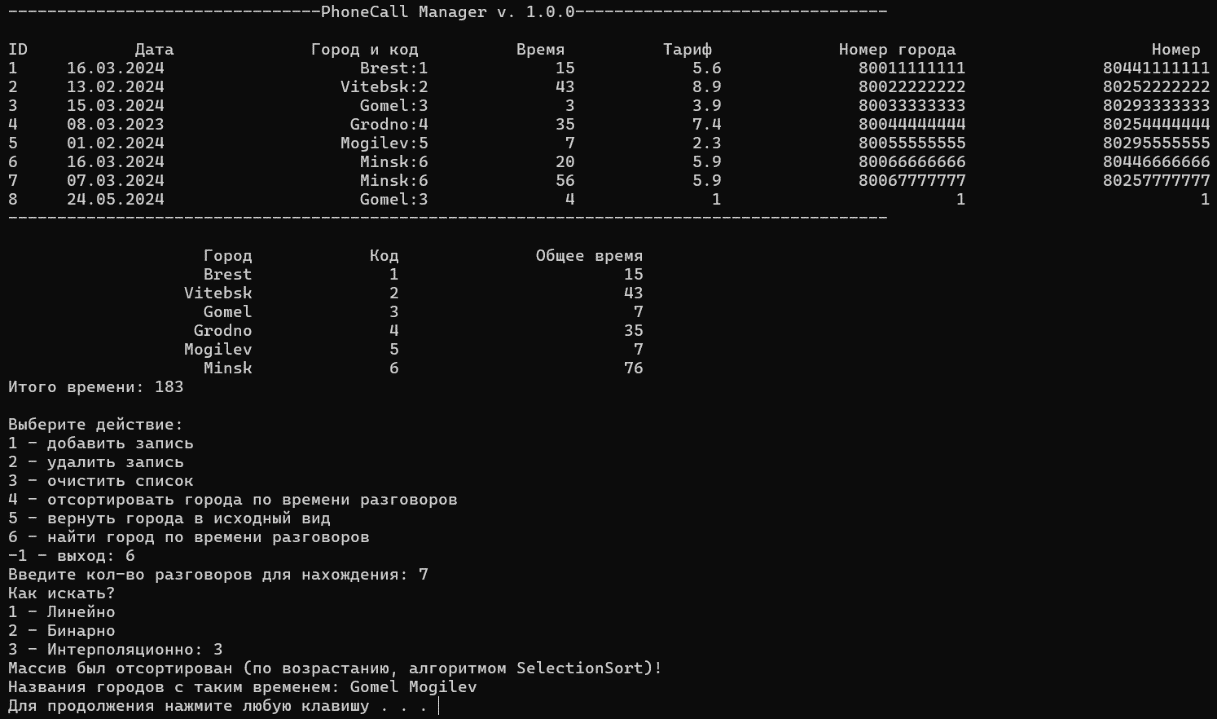


Рисунок 5.9 – Пункт “Найти город по времени разговоров” (город был найден)

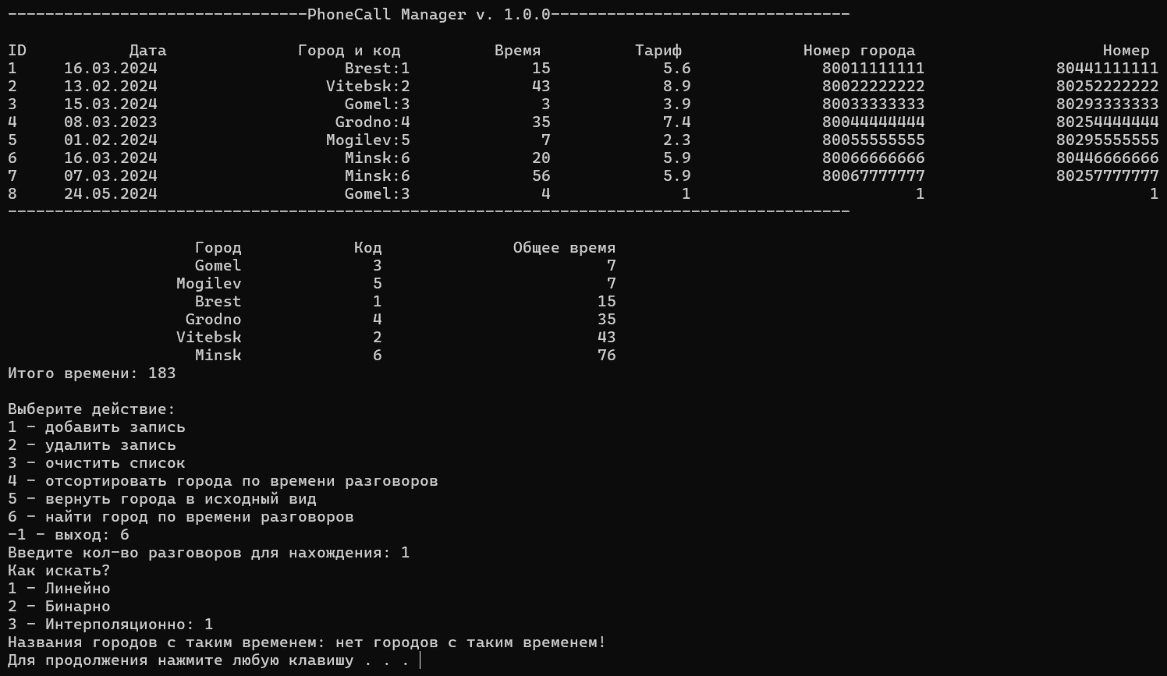


Рисунок 5.10 – Пункт “Найти город по времени разговоров” (город не был найден)

Пункт “Выход” завершает работу программы.

Заключение

В итоге, создание программы потребовало выполнения ряда задач:

* интерпретировать требования задания для их реализации в коде;
* разработать модель программы с учетом необходимого функционала;
* написать код, реализующий все функции;
* отладить и протестировать работу программы.

Требования к проекту были тщательно изучены и проанализированы. На их основе был составлен перечень функциональных задач, которые необходимо реализовать. Далее была создана общая модель программы, описывающая её принцип работы и структуру кода. После этого началась работа над написанием кода на языке программирования C++.

При создании перечня необходимого функционала учитывались все требования задания. Эти требования были интерпретированы и адаптированы для их реализации в рамках программирования, затем они были разбиты на отдельные функции или блоки кода. После составления списка функциональных требований, была продумана их взаимосвязь и интерфейс работы программы. Код программы разрабатывался итерационно, что подразумевало создание работающей версии программы с минимальным функционалом и последующее поэтапное добавление новых функций. Каждый этап сопровождался тестированием корректности работы программы. Этот подход позволил проводить написание и отладку кода одновременно.

В результате выполнения курсового проекта и решения поставленных задач была создана программа на языке C++ для управления списком в междугородных АТС. Программа включает функционал для просмотра, сортировки и редактирования информации о совершённых звонках из заранее заданных городов, что существенно упрощает и ускоряет работу с данными о звонках на междугородных АТС.

Список использованных источников

[1] Классы и структуры (C++) | Microsoft Learn [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/cpp/cpp/classes-and-structs-cpp?view=msvc-170. – Дата доступа: 17.03.2024.

[2] Шилдт, Г. С++ для начинающих / Г. Шилдт. – Москва : Эком, 2013. – 640 с.

[3] Как работает файл на компьютере и что это такое, его характеристики [Электронный ресурс]. – Режим доступа: https://it-tehnik.ru/windows10/fayl-chto-eto.html. – Дата доступа: 20.03.2024.

[4] 13 основных алгоритмов сортировки на С/С++ [Электронный ресурс]. – Режим доступа: https://cyberkesa.ru/algo/. – Дата доступа: 01.04.2024.

[5] Способы поиска элемента в массиве С: просто о сложном [Электронный ресурс]. – Режим доступа: https://codernet.ru/articles/drugoe/sposobyi\_poiska\_elementa\_v\_massive\_s\_prosto\_o\_slozhnom/. – Дата доступа: 05.04.2024.

Приложение А  
(обязательное)  
Листинг кода

#include <fstream>

#include <iomanip>

#include <iostream>

#include <sstream>

#include <vector>

using namespace std;

struct phone\_call

{

unsigned id;

string date;

unsigned city\_code;

string city\_name;

unsigned long long time;

double tarif;

string city\_phone\_number;

string phone\_number;

phone\_call() = default;

phone\_call(unsigned id, string date, unsigned city\_code, string city\_name, unsigned long long time, double tarif, string city\_phone\_number, string phone\_number)

{

this->id = id;

this->date = date;

this->city\_code = city\_code;

this->city\_name = city\_name;

this->time = time;

this->tarif = tarif;

this->city\_phone\_number = city\_phone\_number;

this->phone\_number = phone\_number;

}

};

**Продолжение приложения А**

struct city

{

string name;

unsigned code;

unsigned long long time;

city() = default;

city(string n, unsigned c, unsigned long long t)

{

name = n;

code = c;

time = t;

}

};

vector<phone\_call> phone\_calls;

vector<city> cities;

bool check\_int\_input(string);

bool check\_double\_input(string);

void convert\_int\_to\_unsigned(int, unsigned&);

void convert\_int\_to\_unsigned\_long\_long(int, unsigned long long&);

bool check\_date\_input(string);

bool check\_city\_input(string);

void create\_new\_phone\_call();

void set\_new\_ids(unsigned);

void delete\_phone\_call();

void print\_phone\_calls();

void print\_phone\_calls\_to\_file();

void clear\_file();

void end\_of\_function();

void create\_city\_vector();

void read\_from\_file();

void selection\_sort(int);

void bubble\_sort(int);

void quick\_sort(int, int, int);

void find();

void find\_linear(unsigned long long);

void find\_binary(unsigned long long);

void find\_interpolation(unsigned long long);

**Продолжение приложения А**

bool check\_date\_input(string date)

{

for (int i = 0; i < date.length(); i++)

{

char c = date[i];

if (!isdigit(c) && c != '.')

return false;

if (c == '.' && i != 2 && i != 5 && i != 10)

return false;

}

return true;

}

bool check\_city\_input(string info, string& city\_name, unsigned& city\_code)

{

fstream file("city\_list.txt", ios::in);

if (!file.is\_open())

{

cout << "Файл с городами был создан по умолчанию, поскольку файла не существует.\n";

file.close();

file.open("city\_list.txt", ios::app);

file << "Brest:1\nVitebsk:2\nGomel:3\nGrodno:4\nMogilev:5\nMinsk:6";

file.close();

file.open("city\_list.txt", ios::in);

}

int type = 0;

int input\_city\_code = 0;

try

{

input\_city\_code = stoi(info);

}

catch (invalid\_argument)

{

type = 1;

}

**Продолжение приложения А**

while(true)

{

string buf;

string name;

getline(file, name, ':');

if (name[0] == '\n' || name[0] == '\0')

break;

getline(file, buf, '\n');

int code = stoi(buf);

if (type == 0)

{

if (code == input\_city\_code)

{

city\_code = code;

city\_name = name;

return true;

}

}

else

{

if (name == info)

{

city\_code = code;

city\_name = name;

return true;

}

}

}

file.close();

cout << "Города с таким годом или названием не существует!\n";

return false;

}

bool check\_int\_input(string input)

**Продолжение приложения А**

{

try

{

int temp = stoi(input);

}

catch (invalid\_argument)

{

return false;

}

return true;

}

bool check\_double\_input(string input)

{

try

{

double temp = stod(input);

}

catch (invalid\_argument)

{

return false;

}

return true;

}

void convert\_int\_to\_unsigned\_long\_long(int input, unsigned long long& result)

{

if (input < 0)

input \*= -1;

result = input;

}

void convert\_int\_to\_unsigned(int input, unsigned& result)

{

if (input < 0)

input \*= -1;

result = input;

}

**Продолжение приложения А**

void create\_new\_phone\_call()

{

phone\_call add;

string buf;

do

{

cout << "Введите дату звонка в формате дд.мм.гггг: ";

getline(cin, add.date);

} while (!check\_date\_input(add.date));

do

{

cout << "Введите имя города или его код: ";

getline(cin, buf);

} while (!check\_city\_input(buf, add.city\_name, add.city\_code));

do

{

cout << "Введите время разговора (в минутах): ";

getline(cin, buf);

} while (!check\_int\_input(buf));

convert\_int\_to\_unsigned\_long\_long(stoi(buf), add.time);

do

{

cout << "Введите тариф: ";

getline(cin, buf);

} while (!check\_double\_input(buf));

add.tarif = stod(buf);

cout << "Введите телефонный номер города: ";

getline(cin, add.city\_phone\_number);

cout << "Введите телефонный номер: ";

getline(cin, add.phone\_number);

add.id = phone\_calls.size() + 1;

**Продолжение приложения А**

phone\_calls.push\_back(add);

}

void set\_new\_ids(unsigned id\_to\_delete)

{

for (int i = 0; i < phone\_calls.size(); i++)

if (phone\_calls[i].id > id\_to\_delete)

phone\_calls[i].id--;

}

void delete\_phone\_call()

{

if (phone\_calls.empty())

{

cout << "У вас нет телефонных звонков!\n";

return;

}

unsigned id\_to\_delete;

string buf;

do

{

cout << "Введите ID звонка, который вы хотите удалить: ";

getline(cin, buf);

} while (!check\_int\_input(buf));

convert\_int\_to\_unsigned(stoi(buf), id\_to\_delete);

if (id\_to\_delete > phone\_calls.size())

{

cout << "Не существует элемента с таким ID!\n";

return;

}

for (auto it = phone\_calls.cbegin(); it != phone\_calls.cend(); ++it)

{

if (it->id == id\_to\_delete)

{

phone\_calls.erase(it);

**Продолжение приложения А**

set\_new\_ids(id\_to\_delete);

break;

}

}

}

void print\_phone\_calls()

{

cout << "--------------------------------PhoneCall Manager v. 1.0.0--------------------------------\n\n";

if (phone\_calls.empty())

{

cout << "Список звонков пустой!\n";

return;

}

cout << "ID" << setw(15) << "Дата" << setw(25) << "Город и код" << setw(15) << "Время" << setw(15) << "Тариф" << setw(25) << "Номер города" << setw(25) << "Номер" << '\n';

for (int i = 0; i < phone\_calls.size(); i++)

cout << phone\_calls[i].id << setw(15) << phone\_calls[i].date << setw(25) << phone\_calls[i].city\_name << ':' << phone\_calls[i].city\_code << setw(15) << phone\_calls[i].time << setw(15) << phone\_calls[i].tarif << setw(25) << phone\_calls[i].city\_phone\_number << setw(25) << phone\_calls[i].phone\_number << '\n';

cout << "------------------------------------------------------------------------------------------\n\n";

cout << setw(25) << "Город" << setw(15) << "Код" << setw(25) << "Общее время" << '\n';

unsigned long long time = 0;

for (int i = 0; i < cities.size(); i++)

{

cout << setw(25) << cities[i].name << setw(15) << cities[i].code << setw(25) << cities[i].time << '\n';

time += cities[i].time;

}

**Продолжение приложения А**

cout << "Итого времени: " << time << '\n';

}

void print\_phone\_calls\_to\_file()

{

fstream file("database.txt", ios::app);

for (int i = 0; i < phone\_calls.size(); i++)

file << phone\_calls[i].id << ':' << phone\_calls[i].date << ':' << phone\_calls[i].city\_name << ':' << phone\_calls[i].city\_code << ':' << phone\_calls[i].time << ':' << phone\_calls[i].tarif << ':' << phone\_calls[i].city\_phone\_number << ':' << phone\_calls[i].phone\_number << '\n';

}

void clear\_file()

{

fstream file("database.txt", ofstream::out | ofstream::trunc);

file.close();

}

void end\_of\_function()

{

system("pause");

system("cls");

}

void create\_city\_vector()

{

fstream file("city\_list.txt", ios::in);

while (true)

{

string buf;

string name;

unsigned long long time = 0;

getline(file, name, ':');

if (name[0] == '\n' || name[0] == '\0')

break;

**Продолжение приложения А**

getline(file, buf, '\n');

unsigned code = stoi(buf);

for (int i = 0; i < phone\_calls.size(); i++)

{

if (phone\_calls[i].city\_code == code)

time += phone\_calls[i].time;

}

cities.emplace\_back(name, code, time);

}

file.close();

}

void read\_from\_file()

{

fstream file("database.txt", ios::in);

while (true)

{

string buf;

getline(file, buf, ':');

if(buf[0] == '\n' || buf[0] == '\0')

break;

unsigned id = stoi(buf);

string date;

getline(file, date, ':');

string city\_name;

getline(file, city\_name, ':');

getline(file, buf, ':');

unsigned city\_code = stoi(buf);

getline(file, buf, ':');

unsigned long long time = stoi(buf);

getline(file, buf, ':');

**Продолжение приложения А**

double tarif = stod(buf);

string city\_phone\_number;

getline(file, city\_phone\_number, ':');

string phone\_number;

getline(file, phone\_number, '\n');

phone\_calls.emplace\_back(id, date, city\_code, city\_name, time, tarif, city\_phone\_number, phone\_number);

}

file.close();

}

void bubble\_sort(int type)

{

for (int i = 0; i < cities.size(); i++)

{

for (int j = 0; j < cities.size() - i - 1; j++)

{

if (type == 1)

{

if (cities[j].time > cities[j+1].time)

{

city temp = cities[j];

cities[j] = cities[j + 1];

cities[j + 1] = temp;

}

}

else

{

if (cities[j].time < cities[j + 1].time)

{

city temp = cities[j];

cities[j] = cities[j + 1];

cities[j + 1] = temp;

}

}

}

}

}

**Продолжение приложения А**

void selection\_sort(int type)

{

for (int i = 0; i < cities.size(); i++)

{

int m = i;

for (int j = i + 1; j < cities.size(); j++)

{

if (type == 1)

{

if (cities[j].time < cities[m].time) m = j;

}

else

{

if (cities[j].time > cities[m].time) m = j;

}

}

city temp = cities[m];

cities[m] = cities[i];

cities[i] = temp;

}

}

void quick\_sort(int low, int high, int type)

{

if (low < high)

{

unsigned x = cities[(low + high) / 2].time;

int i = low, j = high;

while (i <= j)

{

if (type == 1)

{

while (cities[i].time < x) i++;

while (cities[j].time > x) j--;

}

else

{

while (cities[i].time > x) i++;

**Продолжение приложения А**

while (cities[j].time < x) j--;

}

if (i <= j)

{

city temp = cities[i];

cities[i] = cities[j];

cities[j] = temp;

i++;

j--;

}

}

if (low < j) quick\_sort(low, j, type);

if (i < high) quick\_sort(i, high, type);

}

}

void sort\_cities()

{

if (cities.empty())

{

cout << "Список городов пустой!\n";

return;

}

int sort\_parameter1 = 0, sort\_parameter2 = 0;

string buf;

cout << "Как сортировать?\n1 - По возрастанию\n2 - По убыванию: ";

while (sort\_parameter1 != 1 && sort\_parameter1 != 2)

{

getline(cin, buf);

if (!check\_int\_input(buf))

{

cout << "Введите правильное число: ";

continue;

}

sort\_parameter1 = stoi(buf);

if (sort\_parameter1 != 1 && sort\_parameter1 != 2)

**Продолжение приложения А**

cout << "Введите правильное число: ";

}

cout << "Каким методом сортировать?\n1 - SelectionSort\n2 - QuickSort\n3 - BubbleSort: ";

while (sort\_parameter2 < 1 || sort\_parameter2 > 3)

{

getline(cin, buf);

if (!check\_int\_input(buf))

{

cout << "Введите правильное число: ";

continue;

}

sort\_parameter2 = stoi(buf);

if (sort\_parameter2 < 1 || sort\_parameter2 > 3)

cout << "Введите правильное число: ";

}

switch (sort\_parameter2)

{

case 1:

selection\_sort(sort\_parameter1);

break;

case 2:

quick\_sort(0, cities.size() - 1, sort\_parameter1);

break;

case 3:

bubble\_sort(sort\_parameter1);

break;

default:

break;

}

}

void find()

{

if (cities.empty())

{

cout << "Список городов пуст!!\n";

return;

**Продолжение приложения А**

}

string buf;

while (!check\_int\_input(buf))

{

cout << "Введите кол-во разговоров для нахождения: ";

getline(cin, buf);

}

unsigned long long time\_to\_find = stoi(buf);

int find\_parameter = 0;

cout << "Как искать?\n1 - Линейно\n2 - Бинарно\n3 - Интерполяционно: ";

while (find\_parameter < 1 || find\_parameter > 3)

{

getline(cin, buf);

if (!check\_int\_input(buf))

{

cout << "Введите правильное число: ";

continue;

}

find\_parameter = stoi(buf);

if (find\_parameter < 1 || find\_parameter > 3)

cout << "Введите правильное число: ";

}

if (find\_parameter != 1)

cout << "Массив был отсортирован (по возрастанию, алгоритмом SelectionSort)!\n";

cout << "Названия городов с таким временем: ";

switch (find\_parameter)

{

case 1:

find\_linear(time\_to\_find);

**Продолжение приложения А**

break;

case 2:

find\_binary(time\_to\_find);

break;

case 3:

find\_interpolation(time\_to\_find);

break;

default:

break;

}

}

void find\_linear(unsigned long long time\_to\_find)

{

int code = 0;

for (int i = 0; i < cities.size(); i++)

{

if (cities[i].time == time\_to\_find)

{

code = 1;

cout << cities[i].name << ' ';

}

}

if (code == 0)

cout << "нет городов с таким временем!\n";

else

cout << '\n';

}

void find\_binary(unsigned long long time\_to\_find)

{

selection\_sort(1);

int i\_key = 0, j = cities.size() - 1;

while (i\_key < j)

{

const int m = (i\_key + j) / 2;

if (cities[m].time < time\_to\_find) i\_key = m + 1;

**Продолжение приложения А**

else j = m;

}

if (cities[i\_key].time != time\_to\_find)

{

cout << "нет городов с таким временем!\n";

return;

}

cout << cities[i\_key].name << ' ';

for (int i = i\_key + 1; i < cities.size(); i++)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

for (int i = i\_key - 1; i >= 0; i--)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

cout << '\n';

}

void find\_interpolation(unsigned long long time\_to\_find)

{

selection\_sort(1);

int low = 0;

int high = cities.size() - 1;

while (low <= high && time\_to\_find >= cities[low].time && time\_to\_find <= cities[high].time)

{

if (low == high)

{

if (cities[low].time == time\_to\_find)

**Продолжение приложения А**

{

cout << cities[low].name << ' ';

for (int i = low + 1; i < cities.size(); i++)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

for (int i = low - 1; i >= 0; i--)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

cout << '\n';

return;

}

cout << "нет городов с таким временем!\n";

return;

}

int pos = low + (high - low) / (cities[high].time - cities[low].time) \* (time\_to\_find - cities[low].time);

if (cities[pos].time == time\_to\_find)

{

cout << cities[pos].name << ' ';

for (int i = pos + 1; i < cities.size(); i++)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

**Продолжение приложения А**

for (int i = pos - 1; i >= 0; i--)

{

if (cities[i].time != time\_to\_find)

break;

cout << cities[i].name << ' ';

}

cout << '\n';

return;

}

if (cities[pos].time < time\_to\_find)

low = pos + 1;

else

high = pos - 1;

}

cout << "нет городов с таким временем!\n";

}

int main()

{

setlocale(LC\_NUMERIC, "C");

setlocale(LC\_CTYPE, "Russian");

int choice;

read\_from\_file();

create\_city\_vector();

do

{

print\_phone\_calls();

choice = 0;

cout << "\nВыберите действие:\n1 - добавить запись\n2 - удалить запись\n3 - очистить список\n4 - отсортировать города по времени разговоров\n5 - вернуть города в исходный вид\n6 - найти город по времени разговоров\n-1 - выход: ";

**Продолжение приложения А**

while ((choice < 1 || choice > 6) && choice != -1)

{

string buf;

getline(cin, buf);

if (!check\_int\_input(buf))

cout << "Введите подходящее число: ";

else

{

choice = stoi(buf);

if ((choice < 1 || choice > 6) && choice != -1)

cout << "Введите подходящее число: ";

}

}

switch (choice)

{

case 1:

create\_new\_phone\_call();

cities.clear();

create\_city\_vector();

end\_of\_function();

break;

case 2:

delete\_phone\_call();

cities.clear();

create\_city\_vector();

end\_of\_function();

break;

case 3:

phone\_calls.clear();

cities.clear();

create\_city\_vector();

end\_of\_function();

break;

case 4:

sort\_cities();

end\_of\_function();

break;

case 5:

**Продолжение приложения А**

cities.clear();

create\_city\_vector();

end\_of\_function();

break;

case 6:

find();

end\_of\_function();

break;

default:

break;

}

clear\_file();

print\_phone\_calls\_to\_file();

} while (choice != -1);

return 0;

}

Приложение Б  
(обязательное)  
Блок-схема работы программы

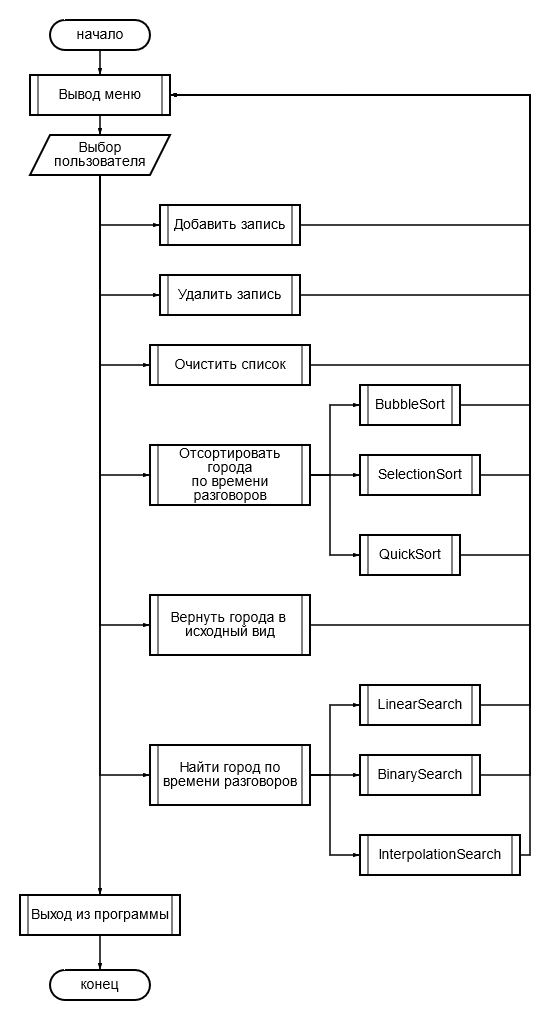


Рисунок Б.1 – Блок-схема работы программы

**ВЕДОМОСТЬ ДОКУМЕНТОВ**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | | | | | *Наименование* | | | | *Дополнитель-ные сведения* | |
|  | | | | | *Текстовые документы* | | | |  | |
|  | | | | |  | | | |  | |
| *БГУИР КП 1-40 05 01-12 063ПЗ* | | | | | *Пояснительная записка* | | | | *57 с.* | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  |  |  |  |  | ***БГУИР КП 1-40 05 01 12 XXX ВД*** | | | | | |
|  |  |  |  |  |
|  |  |  |  |  |
| *Изм.* | *Л.* | *№ докум.* | *Подп.* | *Дата* | *Структура списка междугородной АТС* | *Лит* | | | *Лист* | *Листов* |
| *Разраб.* | | *Фамилия* |  |  |  | *У* |  | *57* | *57* |
| *Пров.* | | *Новицкая* |  |  | *Кафедра ВМиП*  *гр. 32440X* | | | | |
|  | |  |  |  |
|  | |  |  |  |
|  | |  |  |  |