Министерство цифрового развития, связи и массовых коммуникаций
Российской Федерации

Кафедра прикладной математики и кибернетики

Курсовой проект

по курсу

«Структуры и алгоритмы обработки данных»

Вариант 15

Выполнил: студент группы ИП-311
Подкорытова Александра

Проверил: доцент кафедры ПМиК
Янченко Е.В.

Новосибирск, 2024

Содержание

1. ПОСТАНОВКА ЗАДАЧИ

Хранящуюся в файле базу данных загрузить в оперативную память компьютера и построить индексный массив, упорядочивающий данные **по ФИО и названию улицы**, используя **метод прямого слияния** в качестве метода сортировки.

Предусмотреть возможность поиска по ключу в упорядоченной базе, в результате которого из записей с одинаковым ключом формируется очередь, содержимое очереди выводится на экран.

Из записей очереди построить **Двоичное Б-дерево по названию улицы и номеру дома**, и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных статическим **кодом Фано**, предварительно оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран.

Структура записи:

ФИО гражданина: текстовое поле 32 символа
формат <Фамилия>_<Имя>_<Отчество>
Название улицы: текстовое поле 18 символов
Номер дома: целое число
Номер квартиры: целое число
Дата поселения: текстовое поле 10 символов
формат дд-мм-гг

Пример записи из БД:

Петров_Иван_Федорович
Ленина
10
67
29-02-65

Варианты условий упорядочения и ключи поиска (К):

С = 1 - по ФИО и названию улицы, К = первые три буквы фамилии;

2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

2.1. МЕТОД СОРТИРОВКИ

Метод прямого слияния

В основе метода прямого слияния лежит операция слияния серий. p-серией называется упорядоченная последовательность из p элементов. Пусть имеются две упорядоченные серии a и b длины q и r соответственно. Необходимо получить упорядоченную последовательность c, которая состоит из элементов серий a и b. Сначала сравниваем первые элементы последовательностей a и b. Минимальный элемент перемещаем в последовательность c. Повторяем действия до тех пор, пока одна из последовательностей a и b не станет пустой, оставшиеся элементы из другой последовательности переносим в последовательность c. В результате получим (q+r)-серию.

Для алгоритма слияния серий с длинами q и r необходимое количество сравнений 32 и перемещений оценивается следующим образом $\min(q, r) \le C \le q+r-1$, $M=q+r$

Пусть длина списка S равна степени двойки, т.е. 2^k , для некоторого натурального k. Разобьем последовательность S на два списка a и b, записывая поочередно элементы S в списки a и b. Сливаем списки a и b с образованием двойных серий, то есть одиночные элементы сливаются в упорядоченные пары, которые записываются попеременно в очереди c0 и c1. Переписываем очередь c0 в список a, очередь c1 – в список b. Вновь сливаем a и b с образованием серий длины 4 и т. д. На каждом итерации размер серий увеличивается вдвое. Сортировка заканчивается, когда длина серии превысит общее количество элементов в обоих списках. Если длина списка S не является степенью двойки, то некоторые серии в процессе сортировки могут быть короче.

Трудоёмкость метода прямого слияния определяется сложностью операции слияния серий. На каждой итерации происходит ровно n перемещений элементов списка и не более n сравнений. Как нетрудно видеть, количество итераций равно $\lceil \log n \rceil$. Тогда

$$C < n\lceil \log n \rceil,\ M=n\lceil \log n \rceil +n.$$

Дополнительные n перемещений происходят во время начального расщепления исходного списка. Асимптотические оценки для M и C имеют следующий вид

$$C=O(n \log n),\ M=O(n \log n) \text{ при } n \to \infty.$$

Метод обеспечивает устойчивую сортировку. При реализации для массивов, метод требует наличия второго вспомогательного массива, равного по размеру исходному массиву. При реализации со списками дополнительной памяти не требуется.

2.2 ДВОИЧНЫЙ ПОИСК

Алгоритм двоичного поиска в упорядоченном массиве сводится к следующему. Берём средний элемент отсортированного массива и сравниваем с ключом X. Возможны три варианта:

Выбранный элемент равен X. Поиск завершён.

Выбранный элемент меньше X. Продолжаем поиск в правой половине массива.

Выбранный элемент больше X. Продолжаем поиск в левой половине массива.

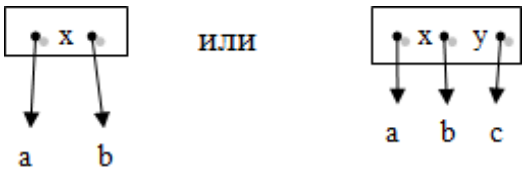
Из-за необходимости найти все элементы соответствующие заданному ключу поиска в курсовой работе использовалась вторая версия двоичного поиска, которая из необходимых элементов находит самый левый, в результате чего для поиска остальных требуется просматривать лишь оставшуюся правую часть массива.

Верхняя оценка трудоёмкости алгоритма двоичного поиска такова. На каждой итерации поиска необходимо два сравнение для первой версии, одно сравнение для второй версии. Количество итераций не больше, чем $\lceil \log_2 n \rceil$. Таким образом, трудоёмкость двоичного поиска в обоих случаях

$$C=O(\log n),\ n \to \infty.$$

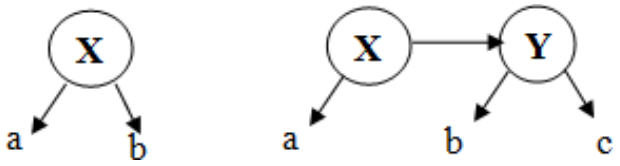
2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ

Двоичное Б-дерево состоит из вершин (страниц) с одним или двумя элементами. Следовательно, каждая страница содержит две или три ссылки на поддеревья. На рисунке ниже показаны примеры страниц Б – дерева при $m = 1$.



Виды вершин ДБД

Классическое представление элементов внутри страницы в виде массива неэффективно, поэтому выбран другой способ представления – динамическое размещение на основе списочной структуры, когда внутри страницы существует список из одного или двух элементов.



Вершины двоичного Б-дерева

2.4 МЕТОД КОДИРОВАНИЯ

Код Фано

Рассмотрим источник с алфавитом $A=\{a_1,a_2,\ldots,a_n\}$ и вероятностями $p_1,\ldots p_n$. Пусть символы алфавита некоторым образом упорядочены, например, $a_1 \le a_2 \le \ldots \le a_n$. *Алфавитным* называется код, в котором кодовые слова лексико-графически упорядочены, т.е. $\varphi(a_1) \le \varphi(a_2) \le \ldots \le \varphi(a_n)$.

Метод Фано построения префиксного почти оптимального кода, для которого $Lcp < H(p1,\ldots,pn) + 1$, заключается в следующем. Упорядоченный по убыванию вероятностей. Список букв алфавита источника делится на две части так, чтобы суммы вероятностей букв, входящих в эти части, как можно меньше отличались друг от друга. Буквам первой части приписывается 0, а буквам из второй части – 1. Далее также поступают с каждой из полученных частей. Процесс продолжается до тех пор, пока весь список не разобьется на части, содержащие по одной букве.

Пример. Пусть дан алфавит A={a1, a2, a3, a4, a5, a6} с вероятностями p1=0.36, p2=0.18, p3=0.18, p4=0.12, p5=0.09, p6=0.07.

Построенный код приведен в таблице.

Таблица 1 Код *Фано*

a_i	P_i	кодовое слово				L_i
a_1	0.36	0	0			2
a_2	0.18	0	1			2
a_3	0.18	1	0			2
a_4	0.12	1	1	0		3
a_5	0.09	1	1	1	0	3
a_6	0.07	1	1	1	1	4

Полученный код является префиксным и почти оптимальным со средней длиной кодового слова Lcp=0.36. 2+0.18. 2+0.18. 2+0.12. 3+0.09. 4+0.07. 4=2.44

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

1. Загрузка и вывод базы данных

Для загрузки базы данных разработана процедура *read_file()*, в которой производится считывание записей типа *record*("Населенный пункт"). Здесь же предусмотрена проверка на наличие файла, откуда выполняется считывание и проверка на выделение памяти для считывания.

За вывод элементов считанной базы данных отвечает процедура *print_data(),* которая предоставляет возможность просмотра базы данных по 20 элементов на странице с возможностью выхода из режима просмотра в главное меню. Также за вывод считанной базы данных отвечает процедура *print_data_all(),* которая предоставляет возможность просмотра всей базы данных с возможностью выхода из режима просмотра в главное меню.

3. Вспомогательные функции и процедуры для сортировки данных

Для сортировки данных используется функция сортировки *MergeSort(),* которая использует вспомогательные процедуры *Split(), from_list_to_line(),Merge*(). Доступ к записям базы данных осуществляется через указатель *next*, для сортировки по ФИО и названию улицы используется процедура *compare_records(),* которая сначала сравнивает ФИО, а затем улицы, если ФИО совпадают, и вызывается в процедуре *Merge*() для сравнения по двум полям структуры. При равенстве суммы вклада происходит сравнение по дате вклада.

4. Особенности реализации бинарного поиска и построения очереди

Бинарный поиск по отсортированной базе осуществляется в функции *binary_searchV2().* Доступ к записям ведётся через индексный массив *indexArray*, который формируется с помощью процедуры *createIndexArray().* При реализации бинарного поиска была использована его вторая версия, так как в результате ее выполнения возвращается номер самого левого из найденных элементов, благодаря чему легко найти и вывести остальные элементы, лишь просмотрев оставшуюся правую часть массива, пока не встретится запись, не удовлетворяющая ключу поиска.

5. Особенности построения дерева, его вывода на экран и поиска

Построение дерева осуществляется в функции *DBD(),* которая вызывается в функции *binary_search_all().* Внутри процедуры построения дерева происходит сравнение записей по названию улицы, с помощью стандартной функции *strcmp(),* сравнение номеров дома происходит как стандартное сравнение целочисленных значений. Для вывода дерева на экран используется процедура *TreeRight(),* которая совершает обход по дереву слева направо и выводит данные на экран. Поиск в дереве осуществляется с помощью рекурсивной функции *Search_DBD(),* которая проверяет данные в корне дерева на соответствие ключу, если ключ поиска и данные в корне дерева различны, то используется рекурсивный вызов процедуры поиска для левого или правого поддерева в зависимости от данных. Если искомые данные меньше корня, то ищем в левом поддереве, иначе – в правом поддереве. Если данные в корне дерева совпадаю с ключом поиска, то на экран выводятся найденные записи.

6. Кодирование данных

Кодирование базы данных начинается с процедуры *CaseFano(),* в которой происходит построение массива встречаемых в базе символов *words* (содержит имя(порядковый номер) и количества встречи символа в базе данных), подсчёт количества всех символов, а также уникальных символов, создание и заполнение динамических массивов для хранения

вероятностей, символов и длин, а также создание двумерного динамического массива для хранения элементарных кодов. Процедура *Sort_probability()*, которая сортирует массив вероятностей по убыванию. Для вычисления энтропии базы данных используется процедура *CalculateEntropy().*

Построение кодовых слов происходит в процедуре Fano(), которая использует дополнительную для кодирования функцию Algoritm_A2(). Процедура *CalculateAVG_L()* используется для подсчёта средней длины кодового слова, а процедура *checkKraft()* вычисляет значения для неравенства Крафта $\sum_{i=1}^{n} 2^{-L_i} \leq 1$ построения кода gilbert().

Вывод результата на экран происходит в процедуре *printFanoCode(),* который включает в себя:
- порядковый номер символа
- символы, встречающиеся в базе данных
- вероятность появления символа
- длину каждого кодового слова
- кодовое слово для каждого символа

А также процедура *printTable(),* которая выводит на экран значение полученное для неравенства Крафта, энтропию, среднюю длину кодового слова и избыточность, также в ней используется вспомогательная функция *Cheak()* для проверки неравенства Lcp < H (p1 ,...,pn) + 1 и вывод этого неравенства на экра

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ

глобальные переменные и константы:

bool flag = false; - сигнализирует, найдены ли подходящие записи в функции Search_DBD().

int СНЕТ = 1; - отслеживает текущий порядковый номер записи при выводе данных.

struct Record - структура, используемая для работы с базой данных «Населенный пункт».

{

char full_name[30]; - поле full_name типа char (используется для хранения ФИО жильца) 32 символа

char street[18]; - поле street типа char (используется для хранения названия улицы) 18 символов

unsigned short int house_number; - поле house_humber типа unsigned short int(используется для хранения номера дома)

unsigned short int apart_number; - поле apart_humber типа unsigned short int(используется для хранения номера квартиры)

char date[10]; - поле date типа char (используется для хранения даты вклада в формате ДД-ММ-ГГ) 10 символов

};

struct List - структура для работы с списком для сортировки и построения двоичного Б-дерева

{

Record data; - поле data типа record(используется для хранения данных типа «Обманутыевкладчики»)

List* next;- поле next, хранящее указатель типа List(указатель на следующий элемент в списке)

List* Right; - поле right, хранящее указатель типа List на правое поддерево

List* Left; - поле left, хранящее указатель типа List на левое поддерево

List* Equal; - поле Equal, хранящее указатель типа List на поддерево с одинаковыми ключами

int Bal; - поле Bal типа int для хранения данных о дереве: 0, если у данной вершины есть только вертикальные ссылки (вершина одна на странице), и 1, если у данной вершины есть правая горизонтальная ссылка.

};

struct line - структура, хранящая указатели на структуру List, для слияния отсортированных очередей

{

List* head = nullptr; - поле, хранящее указатель типа List (указатель на начальный (головной)элемент списка)

List* tail = nullptr; - поле, хранящее указатель типа List (указатель на последний (хвостовой) элемент списка)

};

4.2. ОПИСАНИЕ ПОДПРОГРАММ

Процедуры для обработки базы данных:

1. void read_file(List** head); - чтение базы данных и её запись в список. В качестве параметра принимает указатель на указатель head типа List.
2. void print_data(List* head); - печать базы данных по 20 записей, в качестве параметра принимает List* head(указатель на головной элемент списка).
3. void print_data_all(List* head); - печать всей базы данных, в качестве параметра принимает List* head(указатель на головной элемент списка).

Функции и процедуры сортировки:

4. void MergeSort(List*& S); - основная процедура сортировки методом прямого слияния, принимает в качестве параметра адрес первого элемента в списке S типа List.
5. int Split(List*& S, List*& A, List*& B); - функция разделения списка на два подсписка для дальнейшей

сортировки. В качестве параметров принимает адреса первого элемента списка S типа List и адреса первых элементов в списках типа List A и B для разделения основного.

6. void Merge(List*& A, int& q, List*& B, int& r, line& C); - процедура слияния двух списков в один. В качестве параметров принимает адреса двух первых элементов двухсписков типа List A и B. Длины q и r типа int. Указатель на элемент из очереди C типа line.

7. void from_list_to_line(List*& person, line& queue); - процедура переписывания элементов из очереди в список. В качестве параметров принимает адрес на элемент списка person типа List и адрес элемента из очереди queue типа line.

8. int compare_records(List* a, List* b); - функция для сравнения двух фамилий и улиц.

9. void free_list(List* head); - процедура очистки памяти для списка. В качестве параметра принимает указатель head типа List на первый элемент списка. Вызывается в функции read_file(List** head).

Функции и процедуры для поиска в отсортированной базе данных:

10. void createIndexArray(List* head, List**& indexArray, int& size); - функция заполнения индексного массива указателей на элементы связного списка для двоичного поиска. В качестве параметров принимает указатель head типа List на первый элемент связного списка, ссылку на массив указателей indexArray типа List и ссылку на количество элементов size типа int в индексном массиве указателей.

11. void binary_searchV2(List** indexArray, int size, char* key); - функция двоичного поиска по отсортированным по возрастанию записям (вторая версия). В качестве параметров принимает указатель на массив указателей на элементы списка indexArray типа List, размер индексного массива size типа int и ключ поиска key типа char*.

12. void binary_search_all(List** indexArray, int size, char* key, List*& p); - функция двоичного поиск по отсортированным по возрастанию записям (вторая версия). В качестве параметров принимает указатель на массив указателей на элементы списка indexArray типа List, размер индексного массива size типа int, ключ поиска key типа char* и ссылку на указатель на корень дерева, в которое будут добавляться записи при нахождении совпадений.

Процедуры и функции построения двоичного Б-дерева и поиска в нем

13. bool DBD(record d, List*& p, bool& VR, bool& HR); - функция построения двоичного Б-дерева по названию улицы и номеру дома. В качестве параметров принимает переменную d типа структуры record, хранящую данные, которые нужно добавить в дерево, ссылку на указатель на корень дерева p типа List, в который нужно добавить запись. Указатель будет обновлен в процессе выполнения функции, в случае добавления новой записи. Ссылку на переменную VR типа bool, хранящую в себе информацию о вертикальном росте дерева и ссылку на переменную HR типа bool, хранящую в себе информацию о горизонтальном росте дерева.

14. void TreeRight(List* p, int& i); - функция вывода дерева на экран (обход слева направо). В качестве параметров принимает указатель на корень дерева p типа List и ссылку на переменную i типа int, для вывода порядкового номера записи.

15. void Search_DBD(List* p, const char* X1, unsigned short int X2, int len); - функция поиска по дереву по названию улицы и номеру дома. В качестве параметров принимает указатель на корень дерева p типа List, указатель на ключ поиска X1 типа char, переменную ключа поиска X2 типа unsigned short int и переменную len типа int, хранящую длину ключа.

Процедуры и функции кодирования базы данных:

16. void Sort_probability(char* A, float* P, int L, int R); - функция сортировки символов по убыванию их вероятностей. В качестве параметров принимает динамический массив символов A типа char, динамический массив вероятностей P типа float, индекс левой границы массива L типа int и индекс правой границы R типа int.

17. int Algoritm_A2(float* P, int L, int R); - функция поиска медианы в динамическом массиве вероятностей. В качестве параметров принимает индекс левой границы массива L типа int и индекс правой границы R типа int, а также динамический массив вероятностей P типа float.

18. void Fano(float* P, int L, int R, int k, int* Length, int** C); - основная процедура кодирования, используя код Фано. В качестве параметров принимает динамический массив вероятностей P типа float, левую границу L типа int обрабатываемой части массива P и правую границу R типа int, длину уже построенной части элементарных кодов k типа int, динамический массив длин кодовых слов Length типа int и двумерный динамический массив C типа int (матрица элементарных кодов).

19. float CalculateEntropy(float* P, int n); - функция вычисления энтропии базы данных. В качестве параметров принимает динамический массив P типа float, хранящий вероятности символов, и переменную n типа int(количество уникальных элементов) .

20. float CalculateAVG_L(int* Length, float* P, int n); - функция вычисления средней длины кодового слова. В качестве параметров принимает динамический массив Length типа int, хранящий длины кодовых слов, динамический массив вероятностей P типа float и переменную n типа int(количество уникальных символов).

21. float checkKraft(int* Length, int n); - функция для вычисления неравенства Крафта. В качестве параметров принимает динамический массив длин кодовых слов Lenght типа int и переменную n типа int (количество уникальных символов).

22. bool Cheak(float a, float b); - функция для проверки неравенства Lcp < H (p1 ,...,pn) + 1. В качестве параметров принимает среднюю длину кодового слова a типа float и энтропию b типа float.

23. void printFanoCode(char* symbols, float* P, int* Length, int** C, int n); - функция для печати таблицы закодированных символов. В качестве параметров принимает динамический массив символов symbols типа char, динамический массив вероятностей P типа float, динамический массив длин кодовых слов Lenght типа int, двумерный динамический массив C типа int (матрица элементарных кодов) и переменную n типа int (количество уникальных символов).

24. void printTable(float entropy, float avgLength, float kraftCheck); - функция для вывода таблицы со значением неравенства Крафта, с энтропией базы данных, средней длиной кодового слова и избыточностью кодировки, а также вывод проверки на выполнение неравенства Lcp < H (p1 ,...,pn) + 1. В качестве параметров принимает значение энтропии entropy типа float, средней длины кодового слова avgLength типа float и значение неравенства Крафта kraftCheck типа float.

25. void CaseFano(); - основная функция для кодировки, в которой происходит считывание символов из базы данных, подсчёт вероятностей, вызов процедур Sort_probability(), CalculateEntropy(), Fano(), CalculateAVG_L(), checkKraft(), printFanoCode(), printTable().

Основная программа:

26. int main() – основная программа, в которой последовательно вызываются процедуры для работы с базой данных.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <ctime>
#include <conio.h>
#include <iomanip>
#include <fstream>
#include <cstdio>
#include <sstream>
bool flag = false;
int CHET = 1;

using namespace std;

struct Record
{
   char full_name[32];
   char street[18];
   unsigned short int house_number;
   unsigned short int apart_number;
   char date[10];
};

struct List
{
   Record data;
   List* next;
   List* Right;
   List* Left;
   List* Equal;
   int Bal;
};

struct line
{
   List* head = nullptr;
   List* tail = nullptr;
};

void free_list(List* head)
{
   List* p = head;
   while (p != NULL)
   {
      List* temp = p;
      p = p->next;
      delete temp;
   }
}
```

```
void read_file(List** head)
{
   free_list(*head);
   *head = nullptr;

   FILE* file = nullptr;
   errno_t err = fopen_s(&file, "testBase4.dat", "rb");
   if (!file)
   {
      cout << "Error opening file!" << endl;
      return;
   }
   Record temp;
   while (fread(&temp, sizeof(Record), 1, file))
   {
      List* newList = new List;
      newList->data = temp;
      newList->next = *head;
      *head = newList;
   }
   fclose(file);
}

void print_data(List* head)
{
   List* p = head;
   int i = 0;
   while (p != nullptr)
   {
      cout << " " << setw(5) << i + 1 << "|" << " "
         << p->data.full_name << "\t"
         << p->data.street << "\t"
         << p->data.house_number << "\t"
         << p->data.apart_number << "\t"
         << p->data.date << endl;
      p = p->next;
      i++;
      if (i % 20 == 0)
      {
         char ch;
         cout << "\nIf you want to exit, press 0" << endl;
         ch = _getch();
         if (ch == '0') break;
      }
   }
}

void print_data_all(List* head)
{
   List* p = head;
   int i = 0;
   while (p != nullptr)
   {
      cout << " " << setw(5) << i + 1 << "|" << " "
         << p->data.full_name << "\t"
```

```
            << p->data.street << "\t"
            << p->data.house_number << "\t"
            << p->data.apart_number << "\t"
            << p->data.date << endl;
        p = p->next;
        i++;
        if (i % 4000 == 0)
        {
            char ch;
            cout << "\nIf you want to exit, press 0" << endl;
            ch = _getch();
            if (ch == '0') break;
        }
    }
}

int compare_records(List* a, List* b)
{
    int cmp_full_name = strcmp(a->data.full_name, b->data.full_name);
    if (cmp_full_name < 0) return true;
    else if (cmp_full_name > 0) return false;

    int cmp_street = strcmp(a->data.street, b->data.street);
    return (cmp_street < 0);
}

void from_list_to_line(List*& person, line& queue)  //engueue добавление в очередь
{
    if (queue.head == nullptr) queue.head = person;
    else queue.tail->next = person;
    queue.tail = person;
    person = person->next;
    queue.tail->next = nullptr;
}

int Split(List*& S, List*& A, List*& B)
{
    A = S;
    B = S->next;
    int n = 1;
    List* k = A;
    List* p = B;
    while (p != nullptr)
    {
        n++;
        k->next = p->next;
        k = p;
        p = p->next;
    }
    return n;
}

void Merge(List*& A, int& q, List*& B, int& r, line& C)
{
    while ((q != 0) && (r != 0))
```

```
    {
        if (compare_records(A, B))
        {
            from_list_to_line(A, C);
            q--;
        }
        else
        {
            from_list_to_line(B, C);
            r--;
        }
    }
    while (q > 0)
    {
        from_list_to_line(A, C);
        q--;
    }
    while (r > 0)
    {
        from_list_to_line(B, C);
        r--;
    }
}

void MergeSort(List*& S)
{
    int q = 0, r = 0;
    List* A, * B;
    line C[2];

    int length = Split(S, A, B);
    int p = 1;

    while (p < length)
    {
        for (int i = 0; i < 2; i++)
        {
            C[i].head = nullptr;
            C[i].tail = nullptr;
        }

        int i = 0, m = length;
        while (m > 0)
        {
            if (m >= p) q = p; else q = m;
            m -= q;

            if (m >= p) r = p; else r = m;
            m -= r;

            Merge(A, q, B, r, C[i]);
            i = 1 - i;
        }

        A = C[0].head;
```

```
        B = C[1].head;
        p *= 2;
    }

    S = C[0].head;
}

void createIndexArray(List* head, List**& indexArray, int& size)
{
    List* p = head;
    size = 0;
    while (p != nullptr)
    {
        size++;
        p = p->next;
    }

    indexArray = new List * [size];
    p = head;
    for (int i = 0; i < size; i++)
    {
        indexArray[i] = p;
        p = p->next;
    }
}

void binary_searchV2(List** indexArray, int size, char* key)
{
    int L = 0;
    int R = size - 1;
    int found_count = 0;
    while (L < R)
    {
        int m = (L + R) / 2;
        int cmp = strncmp(indexArray[m]->data.full_name, key, 3);
        if (cmp < 0) L = m + 1;
        else R = m;
    }

    if (strncmp(indexArray[R]->data.full_name, key, 3) == 0)
    {
        found_count++;
        cout << found_count << ". "
            << indexArray[R]->data.full_name << " "
            << indexArray[R]->data.street << " "
            << indexArray[R]->data.house_number << " "
            << indexArray[R]->data.apart_number << " "
            << indexArray[R]->data.date << endl;
        int right = R + 1;

        while (right < size && strncmp(indexArray[right]->data.full_name, key, 3) == 0)
        {
            found_count++;
            cout << found_count << ". "
                << indexArray[right]->data.full_name << " "
```

```
            << indexArray[right]->data.street << " "
            << indexArray[right]->data.house_number << " "
            << indexArray[right]->data.apart_number << " "
            << indexArray[right]->data.date << endl;
         right++;
      }
   }
   else
   {
      cout << "Element not found" << endl;
   }
   system("pause");
}

bool DBD(Record d, List*& p, bool& VR, bool& HR)
{
   if (p == NULL)
   {
      p = new List;
      p->data = d;
      p->Left = p->Right = p->Equal = NULL;
      p->Bal = 0;
      VR = true;
      return true;
   }
   else if ((strcmp(p->data.street, d.street) > 0) || (strcmp(p->data.street, d.street) == 0 && d.house_number < p-
>data.house_number))
   {
      if (DBD(d, p->Left, VR, HR))
      {
         if (VR)
         {
            if (p->Bal == 0)
            {
               List* q = p->Left;
               p->Left = q->Right;
               q->Right = p;
               p = q;
               q->Bal = 1;
               VR = false;
               HR = true;
            }
            else
            {
               p->Bal = 0;
               VR = true;
               HR = false;
            }
         }
         else
            HR = false;
      }
      else
         return false;
   }
```

```
    else if ((strcmp(p->data.street, d.street) < 0) || (strcmp(p->data.street, d.street) == 0 && d.house_number > p->data.house_number)) {
        if (DBD(d, p->Right, VR, HR))
        {
            if (VR)
            {
                p->Bal = 1;
                HR = true;
                VR = false;
            }
            else if (HR)
            {
                if (p->Bal == 1)
                {
                    List* q = p->Right;
                    p->Bal = 0;
                    q->Bal = 0;
                    p->Right = q->Left;
                    q->Left = p;
                    p = q;
                    VR = true;
                    HR = false;
                }
                else
                    HR = false;
            }
        }
        else
            return false;
    }
    else
    {
        List* newEqual = new List;
        newEqual->data = d;
        newEqual->next = nullptr;
        newEqual->Equal = p->Equal;
        p->Equal = newEqual;

        return true;
    }
    return true;
}

void binary_search_all(List** indexArray, int size, char* key, List*& p)
{
    bool VR = true;
    bool HR = true;
    int L = 0;
    int R = size - 1;
    int found_count = 0;
    while (L < R)
    {
        int m = (L + R) / 2;
        int cmp = strncmp(indexArray[m]->data.full_name, key, 3);
        if (cmp < 0) L = m + 1;
```

```
        else R = m;
    }

    if (strncmp(indexArray[R]->data.full_name, key, 3) == 0)
    {
        found_count++;
        cout << found_count << ". "
            << indexArray[R]->data.full_name << " "
            << indexArray[R]->data.street << " "
            << indexArray[R]->data.house_number << " "
            << indexArray[R]->data.apart_number << " "
            << indexArray[R]->data.date << endl;
        DBD(indexArray[R]->data, p, VR, HR);
        int right = R + 1;

        while (right < size && strncmp(indexArray[right]->data.full_name, key, 3) == 0)
        {
            found_count++;
            cout << found_count << ". "
                << indexArray[right]->data.full_name << " "
                << indexArray[right]->data.street << " "
                << indexArray[right]->data.house_number << " "
                << indexArray[right]->data.apart_number << " "
                << indexArray[right]->data.date << endl;
            DBD(indexArray[right]->data, p, VR, HR);
            right++;
        }
    }
    else
        cout << "Element not found" << endl;
    system("pause");
}

void TreeRight(List* p, int& i)
{
    if (p != nullptr)
    {
        TreeRight(p->Left, i);
        cout << " " << setw(5) << ++i << "|" << " "
            << p->data.full_name << "\t"
            << p->data.street << "\t"
            << p->data.house_number << "\t"
            << p->data.apart_number << "\t"
            << p->data.date << endl;

        List* EqualNode = p->Equal;
        while (EqualNode != nullptr)
        {
            cout << " " << setw(5) << ++i << "|" << " "
                << EqualNode->data.full_name << "\t"
                << EqualNode->data.street << "\t"
                << EqualNode->data.house_number << "\t"
                << EqualNode->data.apart_number << "\t"
                << EqualNode->data.date << endl;
            EqualNode = EqualNode->Equal;
```

```
        }
        TreeRight(p->Right, i);
    }

}

void Search_DBD(List* p, const char* X1, unsigned short int X2, int len)
{
    if (p == nullptr)
        return;
    if ((strncmp(p->data.street, X1, len) == 0) && p->data.house_number == X2)
    {
        cout << " " << setw(5) << CHET << "|" << " "
            << p->data.full_name << "\t"
            << p->data.street << "\t"
            << p->data.house_number << "\t"
            << p->data.apart_number << "\t"
            << p->data.date << endl;
        bool Eq = false;
      CHET++;

        List* Equal = p->Equal;
        while (Equal != nullptr)
        {
            cout << " " << setw(5) << CHET << "|" << " "
                << Equal->data.full_name << "\t"
                << Equal->data.street << "\t"
                << Equal->data.house_number << "\t"
                << Equal->data.apart_number << "\t"
                << Equal->data.date << endl;
            Equal = Equal->Equal;
            Eq = true;
        }
        flag = true;
        if (Eq) CHET++;
    }

    Search_DBD(p->Left, X1, X2, len);
    Search_DBD(p->Right, X1, X2, len);
}


void Sort_probability(char* A, float* P, int L, int R)
{
    float x = P[(L + R) / 2];
    int i = L;
    int j = R;
    while (i <= j)
    {
        while (P[i] > x)
        {
            i++;
        }
        while (P[j] < x)
        {
```

```
            j--;
        }
        if (i <= j)
        {
            swap(A[i], A[j]);
            swap(P[i], P[j]);
            i++;
            j--;
        }
    }
    if (L < j)
        Sort_probability(A, P, L, j);
    if (i < R)
        Sort_probability(A, P, i, R);
}

int Algoritm_A2(float* P, int L, int R)
{
    float Summa = 0;
    float sum = 0;
    int i = 0;
    if (L <= R)
    {

        for (i = L; i < R; i++)
        {
            Summa = Summa + P[i];
        }
        for (i = L; i < R; i++)
        {
            if ((sum < Summa / 2) && (sum + P[i] >= Summa / 2))
                break;
            else
                sum = sum + P[i];
        }
    }
    return i;
}

void Fano(float* P, int L, int R, int k, int* Length, int** C)
{
    if (L < R)
    {
        k++;
        int m = Algoritm_A2(P, L, R);

        for (int i = L; i <= R; i++)
        {
            if (i <= m)
            {
                C[i][k] = 0;
                Length[i] = Length[i] + 1;
            }
            else
            {
```

```
            C[i][k] = 1;
            Length[i] = Length[i] + 1;
        }
    }

    Fano(P, L, m, k, Length, C);
    Fano(P, m + 1, R, k, Length, C);
  }
}

float CalculateEntropy(float* P, int n)
{
  float result = 0;
  for (int i = 0; i < n; i++)
  {
    result += P[i] * log2(P[i]);
  }
  return -result;
}

float CalculateAVG_L(int* Length, float* P, int n)
{
  float result = 0;
  for (int i = 0; i < n; i++)
  {
    result += Length[i] * P[i];
  }
  return result;
}

float checkKraft(int* Length, int n)
{
  float kraftSum = 0;
  for (int i = 0; i < n; i++)
  {
    kraftSum += pow(2, -Length[i]);
  }
  return kraftSum;
}

void printFanoCode(char* symbols, float* P, int* Length, int** C, int n)
{
  cout << "-----------------------------------------------------" << endl;
  cout << "| Number | Symbol | Probability | Length|   Code Word   |" << endl;
  cout << "-----------------------------------------------------" << endl;

  for (int i = 0; i < n; i++)
  {
    cout << "| " << setw(6) << i + 1 << " | "
      << setw(6) << symbols[i] << " | "
      << setw(10) << fixed << setprecision(7) << P[i] << setw(4) << " | "
      << setw(5) << Length[i] << " | ";
    for (int j = 0; j < Length[i]; j++)
    {
      cout << C[i][j];
```

```
        }
        cout << setw(15 - Length[i]) << " |" << endl;
    }
    cout << "-----------------------------------------------------" << endl;
    float countP = 0.0;
    for (int i = 0; i < n; i++)
    {
        countP += P[i];
    }
    cout << "\nSumma probabilities = " << fixed << setprecision(4) << countP;
}

void printTable(float entropy, float avgLength, float kraftCheck)
{
    float redundancy = avgLength - entropy;

    cout << endl;
    cout << "-----------------------------------------------------" << endl;
    cout << "|   Kraft   | Entropy | Average lenght | Redundancy |" << endl;
    cout << "-----------------------------------------------------" << endl;
    cout << "| " << setw(9) << fixed << setprecision(5) << kraftCheck << " | "
        << setw(2) << fixed << setprecision(5) << entropy << " | "
        << setw(10) << fixed << setprecision(5) << avgLength << setw(7) << " | "
        << setw(10) << fixed << setprecision(5) << redundancy << " |" << endl;
    cout << "-----------------------------------------------------" << endl;
}

void CaseFano()
{
    int words[256] = { 0 };
    int totalNums = 0, uniqueSymbols = 0;
    char ch;

    ifstream file("testBase4.dat", ios::binary);

    if (!file.is_open())
    {
        cout << "Error#1. File \"testBase4.dat\" not found!" << endl;
    }

    while (file.read(&ch, sizeof(ch)))
    {
        totalNums++;
        words[(unsigned char)ch]++;
    }
    file.close();

    for (int i = 0; i < 256; i++)
    {
        if (words[i] != 0)
            uniqueSymbols++;
    }

    cout << "Unique symbols: " << uniqueSymbols << endl;
    cout << "Total symbols: " << totalNums << endl;
```

```
    float* P = new float[uniqueSymbols];
    char* symbols = new char[uniqueSymbols];
    int* Length = new int[uniqueSymbols];
    int** C = new int* [uniqueSymbols];
    for (int i = 0; i < uniqueSymbols; i++)
    {
        C[i] = new int[256](); // Инициализация нулями
    } // Матрица для кодов, 256 - максимальная длина кодового слова

    int index = 0;
    for (int i = 0; i < 256; i++)
    {
        if (words[i] != 0)
        {
            symbols[index] = (char)i;
            P[index] = (float)words[i] / totalNums;
            Length[index] = 0;
            index++;
        }
    }
    // Сортируем массив вероятностей и символов
    Sort_probability(symbols, P, 0, uniqueSymbols - 1);
    float entropy = CalculateEntropy(P, uniqueSymbols);
    Fano(P, 0, uniqueSymbols - 1, -1, Length, C);
    float avgLength = CalculateAVG_L(Length, P, uniqueSymbols);
    float kraftCheck = checkKraft(Length, uniqueSymbols);

    printFanoCode(symbols, P, Length, C, uniqueSymbols);
    printTable(entropy, avgLength, kraftCheck);
}


int main()
{
    List* records = nullptr;
    List** indexArray = nullptr;
    List* p = nullptr;
    int size = 0;
    char key[4];
    char answer_user;
    bool Flag = true;
    int i = 0;
    int Size = 256;
    char* street = new char[Size];
    unsigned short int house_number;
    int len = 0;
    read_file(&records);
    do
    {
        system("CLS");
        cout << "MENU" << endl
            << "1. Output 20 records\n"
            << "2. Sort the database\n"
            << "3. Output all records\n"
```

```
        << "4. Output sorted all records\n"
        << "5. Binary search\n"
        << "6. Binary B-tree by street and house number and search\n"
        << "7. Fano code\n"
        << endl;
    answer_user = _getch();
    switch (answer_user)
    {
    case '1':
        system("CLS");
        cout << "\t\t\tThe database\n" << endl;
        print_data(records);
        break;

    case '2':
        system("CLS");
        cout << "\t\t\tSorted database\n" << endl;
        MergeSort(records);
        print_data(records);
        break;

    case '3':
        system("CLS");
        cout << "\t\t\tThe full database\n" << endl;
        print_data_all(records);
        break;

    case '4':
        system("CLS");
        cout << "\t\t\tSorted full database\n" << endl;
        MergeSort(records);
        createIndexArray(records, indexArray, size);
        print_data_all(records);
        break;

    case '5':
        system("CLS");
        cout << "Enter the key for search: ";
        MergeSort(records);
        cin.getline(key, 4);
        createIndexArray(records, indexArray, size);
        binary_searchV2(indexArray, size, key);
        break;

    case '6':
        MergeSort(records);
        cout << "Enter the key for search: ";
        cin.getline(key, 4);
        cout << "\n\t\t\t\t\tFound records\n\n";

        createIndexArray(records, indexArray, size);
        binary_search_all(indexArray, size, key, p);
        cout << endl << "--------------------------BINARY B-TREE (KEY OF SEARCH: STREET AND HOUSE
NUMBER)----------------------------\n" << endl;
        TreeRight(p, i);
```

```
        cout << "Enter the street for search: ";
        cin.getline(street, Size);

        while (street[len] != '\0')
            len++;
        cout << "Enter the house number for search: ";
        cin >> house_number;
        Search_DBD(p, street, house_number, len);
        CHET = 1;
        if (!flag)
        {
            cout << "No records were found matching the specified criteria.\n";
            flag = false;
        }

        system("pause");
        p = nullptr;
        break;

    case '7':
        CaseFano();
        system("pause");

    }

  } while (Flag);
  free_list(p);

  return 0;
}
```

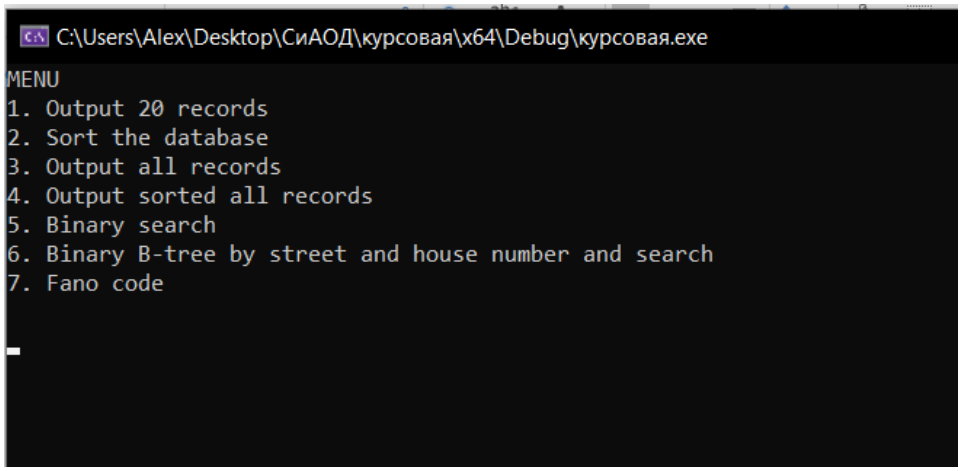


Рисунок 1. Меню программы

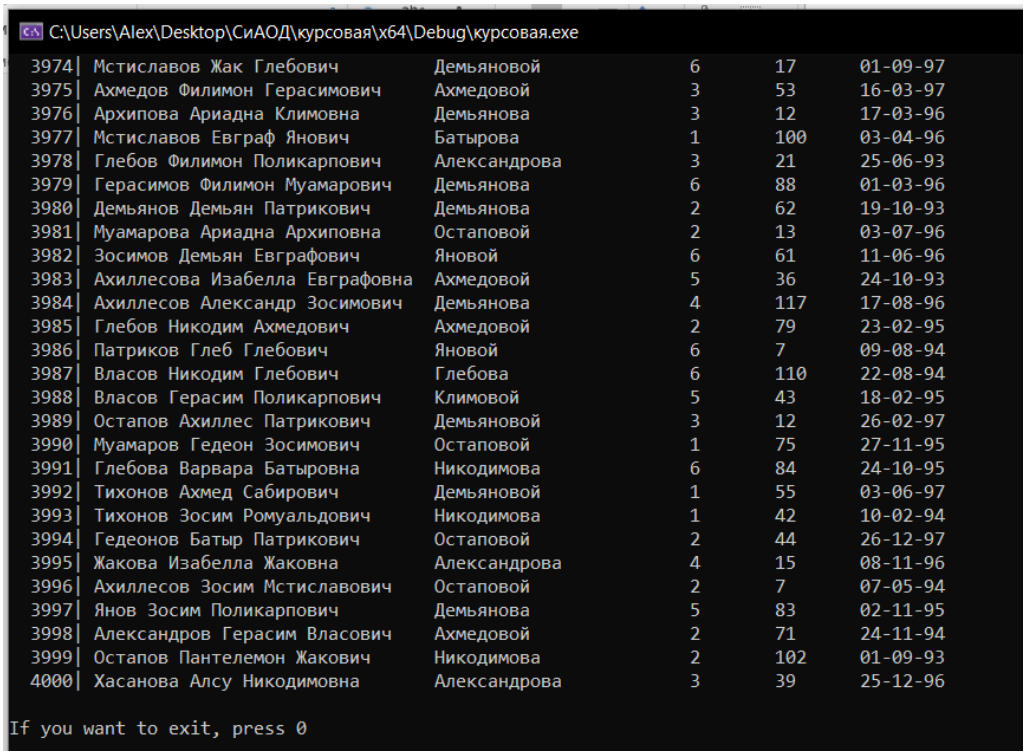


Рисунок 2. Неотсортированная база данных (вывод всех записей).

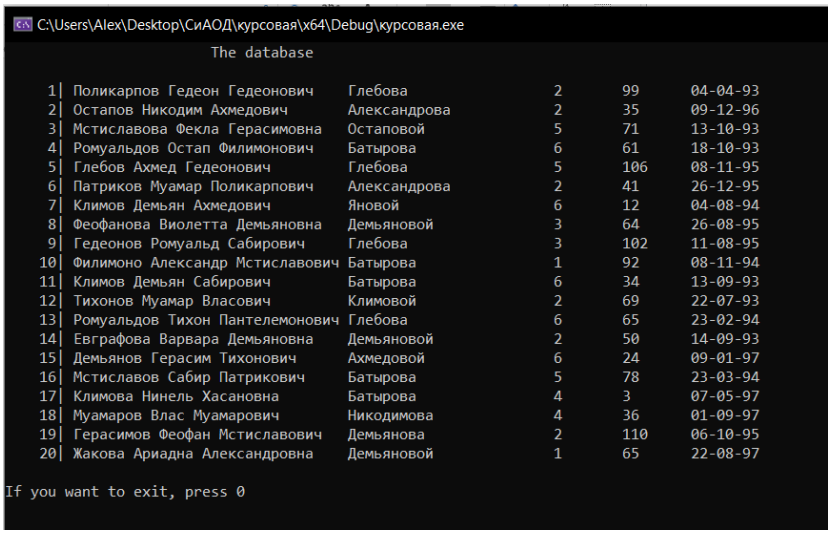


Рисунок 3. Неотсортированная база данных (вывод по 20 записей).

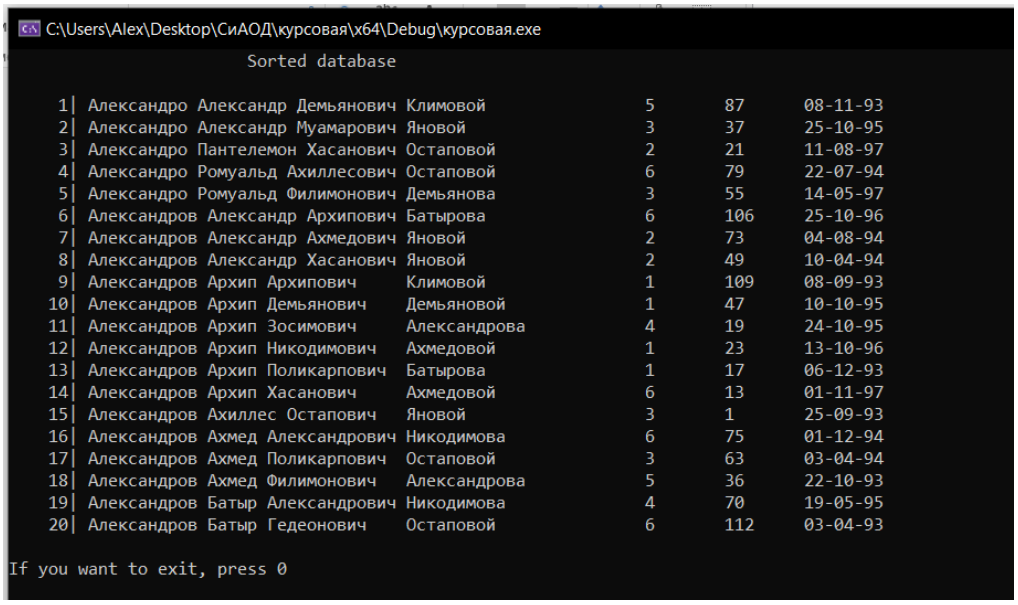


Рисунок 4. Отсортированная по ФИО и названию улицы база данных (вывод по 20 записей).

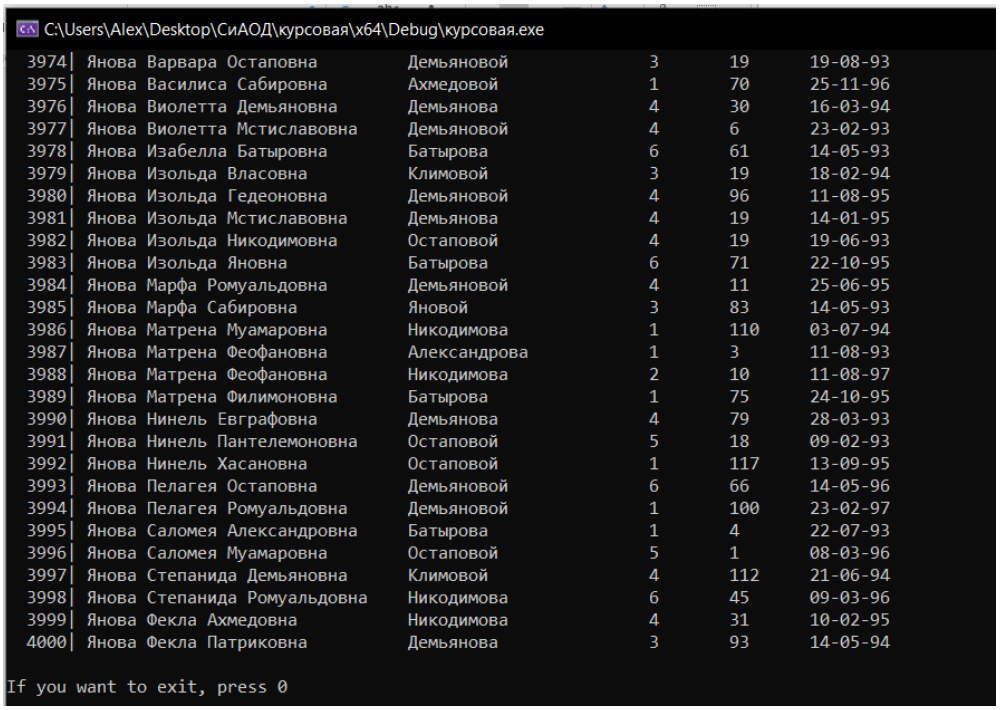


Рисунок 5. Отсортированная по ФИО и названию улицы база данных (вывод всех записей).

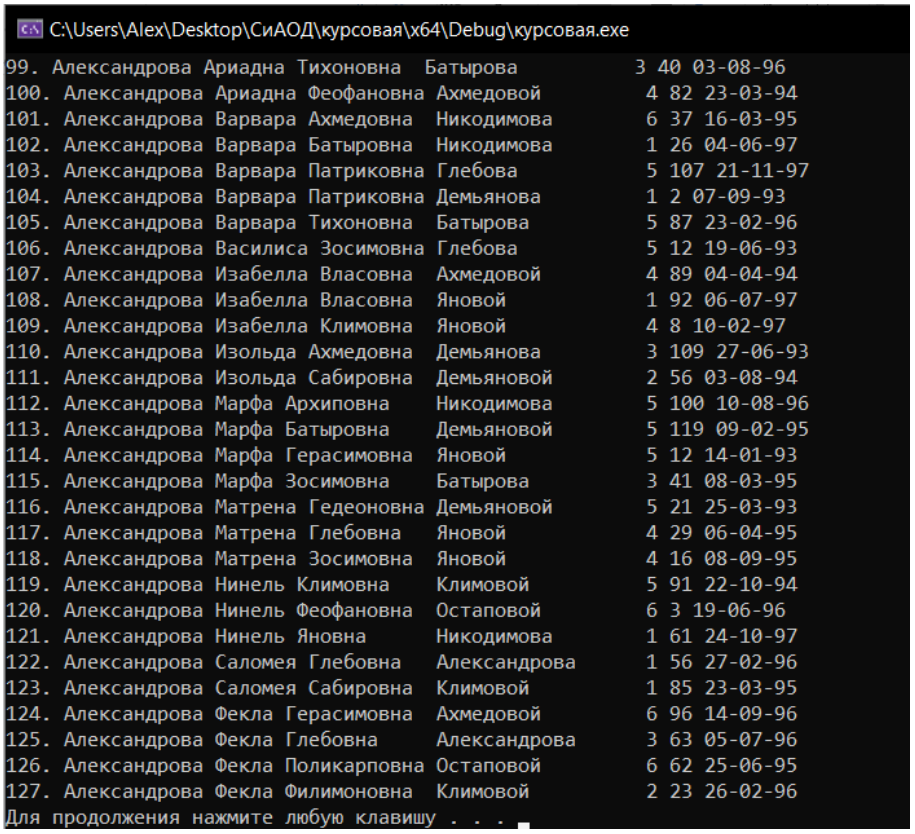


Рисунок 6. Записи, полученные в результате двоичного поиска (ключ: первые три буквы фамилии = «Але»)

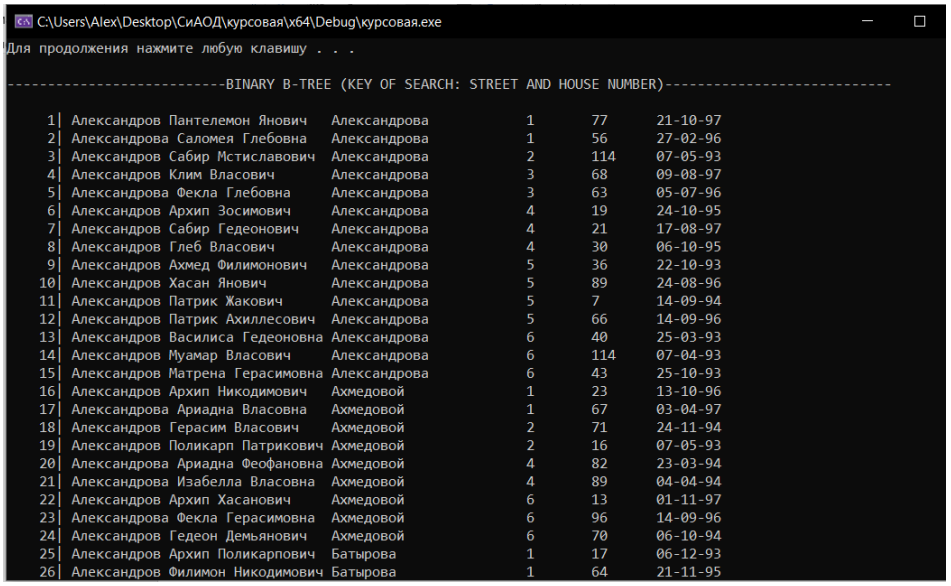


Рисунок 7. Вывод двоичного Б-дерева (дерево строится по названию улицы и номеру дома)

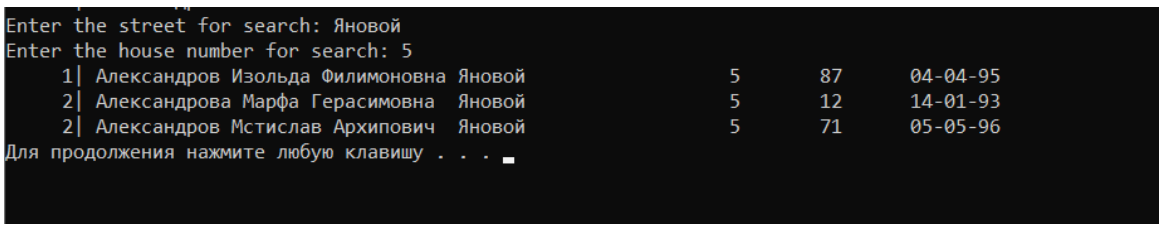


Рисунок 8. Поиск по дереву (данные для поиска: название улицы – Яновой, номер дома - 5)

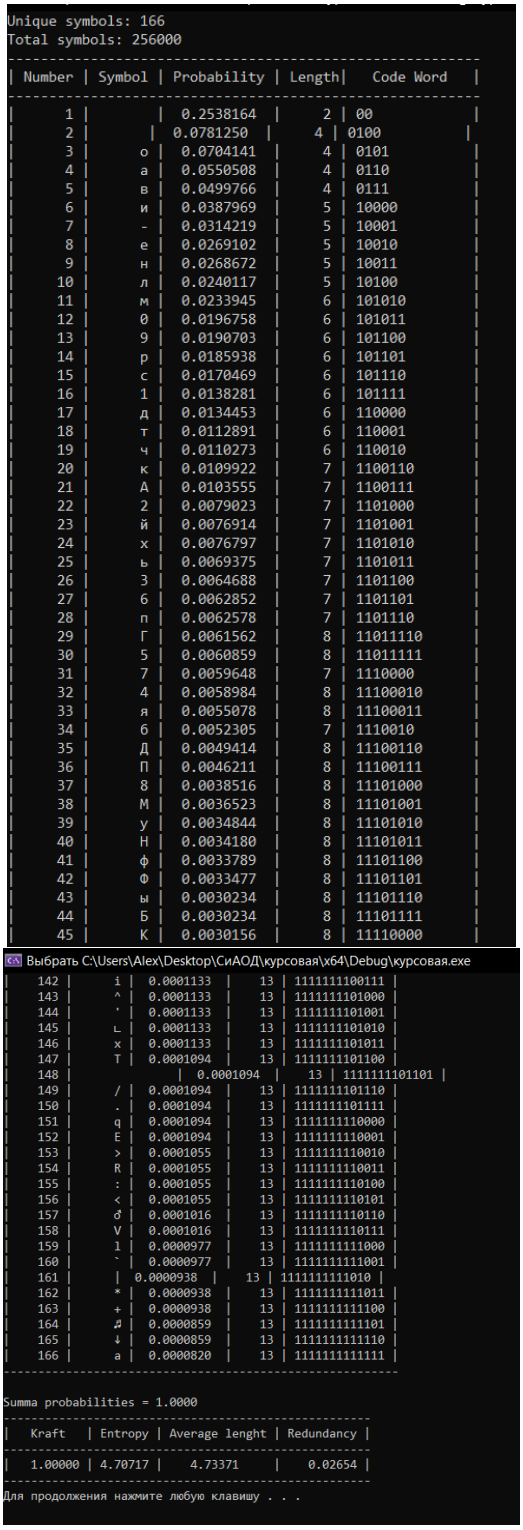


Рисунок 9. Вывод закодированных символов из базы данных методом кодировки Фано.

7. ВЫВОДЫ

В ходе выполнения курсового проекта были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, двоичного поиска, создания очереди, построения двоичного бинарного дерева, поиска по дереву, кодирования данных.

Четкая структуризация кода и грамотно подобранные имена переменных, структур данных, функций и процедур способствуют удобочитаемости программы.

Все разработанные алгоритмы расширяют возможности работы с данными и способствуют улучшению эффективности анализа и обработки данных и представляют собой минимальный набор процедур для представления и обработки базы данных.