Федеральное агентство связи
Сибирский государственный университет
телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовая работа
«Структуры и алгоритмы обработки данных»
Вариант 126

Выполнил: студентка группы ИП-916
Александрова А. С.
Проверил: преподаватель кафедры ПМиК
Турцев А. А.

Новосибирск 2020

Содержание

1. ПОСТАНОВКА ЗАДАЧИ

Хранящуюся в файле базу данных загрузить в оперативную память компьютера и построить индексный массив, упорядочивающий данные **по названию улицы и номеру дома**, используя **метод Хоара** 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 **сбалансированное по высоте дерево (АВЛ) по названию улицы**, и предусмотреть возможность поиска в дереве по запросу.

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

Программа разрабатывается на языке программирования си.

В = 4 (файл testBase4.dat)

База данных "Населенный пункт"

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

Условие упорядочения и ключ поиска (К):

С = 2 - по названию улицы и номеру дома, К = первые три буквы названия улицы;

2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

2.1. МЕТОД СОРТИРОВКИ

Метод Хоара или метод быстрой сортировки заключается в следующем. Возьмём произвольный элемент массива x. Просматривая массив слева, найдём элемент $a_i \geq x$. Просматривая массив справа, найдём $a_j \leq x$. Поменяем местами a_i и a_J . Будем продолжать процесс просмотра и обмена, до тех пор пока i не станет больше j. Тогда массив можно разбить на две части: в левой части все элементы не больше x, в правой части массива не меньше x. Затем к каждой части массива применяется тот же алгоритм.

Очевидно, трудоёмкость метода существенно зависит от выбора элемента x, который влияет на разделение массива. Максимальные значения M и C для метода быстрой сортировки достигаются при сортировке упорядоченных массивов (в прямом и обратном порядке). Тогда в этом случае в одной части остаётся только один элемент (минимальный или максимальный), а во второй – все остальные элементы. Выражения для M и C имеют следующий вид

$$M=3(n-1),\ C=(n^2+5n+4)/2$$

Таким образом, в случае упорядоченных массивов трудоёмкость сортировки имеет квадратичный порядок.

Элемент a_m называется *медианой* для элементов $a_L \ldots a_R$, если количество элементов меньших a_m равно количеству элементов больших a_m с точностью до одного элемента (если количество элементов нечётно). В примере буква К- медиана для КУРАПОВАЕ.

Минимальная трудоемкость метода Хоара достигается в случае, когда на каждом шаге алгоритма в качестве ведущего элемента выбирается медиана массива. Количество сравнений в этом случае $C=(n+1)\log(n+1)-(n+1)$. Количество пересылок зависит от положения элементов, но не может быть больше одного обмена на два сравнения. Поэтому количество пересылок – величина того же порядка, что и число сравнений. Асимптотические оценки для средних значений M и C имеют следующий вид

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

Метод Хоара неустойчив.

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

Дерево поиска называется *сбалансированным по высоте*, или *АВЛ – деревом*, если для каждой его вершины высоты левого и правого поддеревьев отличаются не более чем на 1.

Адельсон – Вельский и Ландис доказали теорему, гарантирующую, что АВЛ-дерево никогда не будет в среднем по высоте превышать ИСДП более, чем на 45% независимо от количества вершин:

$$\log(n+1) \le h_{АВЛ}(n) < 1{,}44 \log(n+2) - 0{,}328 \text{ при } n \to \infty.$$

Лучший случай сбалансированного по высоте дерева – ИСДП, худший случай – плохое АВЛ – дерево(АВЛ-дерево, которое имеет наименьшее число вершин при фиксированной высоте). Процесс построения плохого АВЛ-дерева: берется фиксированнаю высота h и строится АВЛ – дерево с минимальным количеством вершин(T_h). Ясно, что T_0 – пустое дерево, T_1 – дерево с одной вершиной. Для построения T_h при h > 1 берется корень и два поддерева с минимальным количеством вершин.

Одно поддерево должно быть высотой h–1, а другое высотой h–2.

При включении новой вершины в сбалансированное по высоте дерево. Пусть r – корень АВЛ-дерева, у которого имеется левое поддерево (T_L) и правое поддерево (T_R). Если добавление новой вершины в левое поддерево приведет к увеличению его высоты на 1, то возможны три случая:

1) если $h_L = h_R$, то T_L и T_R станут разной высоты, но баланс не будет нарушен;
2) если $h_L < h_R$, то T_L и T_R станут равной высоты, т. е. баланс даже улучшится;
3) если $h_L > h_R$, то баланс нарушиться и дерево необходимо перестраивать.

Вводится в каждую вершину дополнительный параметр Balance (показатель баланса), принимающий следующие значения:

-1, если левое поддерево на единицу выше правого;
0, если высоты обоих поддеревьев одинаковы;
1, если правое поддерево на единицу выше левого.

Если в какой-либо вершине баланс высот нарушается, то необходимо так перестроить имеющееся дерево, чтобы восстановить баланс в каждой вершине. Для восстановления баланса будем использовать процедуры поворотов АВЛ-дерева.

Добавление новой вершины в АВЛ-дерево: Вначале добавим новую вершину в дерево так же как в случайное дерево поиска (проход по пути поиска до нужного места). Затем, двигаясь назад по пути поиска от новой вершины к корню дерева, будем искать вершину, в которой нарушился баланс (т. е. высоты левого и правого поддеревьев стали отличаться более чем на 1). Если такая вершина найдена, то изменим структуру дерева для восстановления баланса с помощью процедур поворотов.

2.4 МЕТОД КОДИРОВАНИЯ

Код Шеннона позволяет построить почти оптимальный код с длинами кодовых слов $L_i < -\log p_i + 1$. Тогда по теореме Шеннона из п. 5.1

$$L_{cp} < H(p_1,..., p_n) + 1.$$

Код Шеннона, удовлетворяющий этому соотношению, строится следующим образом:

1. Упорядочим символы исходного алфавита A=$\{a_1,a_2,...,a_n\}$ по убыванию их вероятностей: $p_1 \geq p_2 \geq p_3 \geq ... \geq p_n$.
2. Вычислим величины Q_i:, которые называются *кумулятивные вероятности*

$$Q_0=0,\ Q_1=p_1,\ Q_2=p_1+p_2,\ Q_3=p_1+p_2+p_3,\ \dots,\ Q_n=1.$$

3. Представим Q_i в двоичной системе счисления и возьмем в качестве кодового слова первые $\lceil -\log p_i \rceil$ знаков после запятой .

Для вероятностей, представленных в виде десятичных дробей, удобно определить длину кодового слова L_i из соотношения

$$\frac{1}{2^{L_i}} \leq p_i < \frac{1}{2^{L_i-1}},\ i=1,\dots,n.$$

Пример. Пусть дан алфавит A=$\{a_1, a_2, a_3, a_4, a_5, a_6\}$ с вероятностями p_1=0.36, p_2=0.18, p_3=0.18, p_4=0.12, p_5=0.09, p_6=0.07. Построенный код приведен в таблице 6.

Таблица 6 Код Шеннона

a_i	P_i	Q_i	L_i	кодовое слово
a_1	$1/2^2 \leq 0.36 < 1/2$	0	2	00
a_2	$1/2^3 \leq 0.18 < 1/2^2$	0.36	3	010
a_3	$1/2^3 \leq 0.18 < 1/2^2$	0.54	3	100
a_4	$1/2^4 \leq 0.12 < 1/2^3$	0.72	4	1011
a_5	$1/2^4 \leq 0.09 < 1/2^3$	0.84	4	1101
a_6	$1/2^4 \leq 0.07 < 1/2^3$	0.93	4	1110

Построенный код является префиксным. Вычислим среднюю длину кодового слова и сравним ее с энтропией. Значение энтропии вычислено при построении кода Хаффмана в п. 5.2 (H = 2.37), сравним его со значением средней длины кодового слова кода Шеннона

L_{cp}= 0.36·2+(0.18+0.18)·3+(0.12+0.09+0.07)·4=2.92< 2.37+1,

что полностью соответствует утверждению теоремы Шеннона.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

В ходе выполнения курсовой работы, помимо основных алгоритмов, потребовалось реализовать также несколько вспомогательных, необходимых для корректной работы программы.

1. Интерфейс программы

Для организации интерфейса использовалось меню**,** которое обеспечивает многократный выбора различных вариантов обработки базы данных. Визуальное представление пунктов меню вынесено в отдельную процедуру void Draw_Menu(). Также предусмотрено второе меню void Print_Head_Table(). Которое позволяет передвигаться по страницам.

2. Загрузка и вывод базы данных

Для загрузки базы данных разработана процедура *int Read_Data_Base_ To_Array(),* в которой производится считывание записей типа *struct Record*. Здесь же предусмотрена проверка на наличие файла, откуда выполняется считывание.
За вывод элементов считанной базы данных отвечает процедура *void Print_Data_Base().*Она предоставляет возможность просмотра страниц базы данных, здесь же открывается второе меню с выбором перехода на следующую, предыдущую, выбрать определенную страницу или выход к главному меню.

3. Вспомогательные функции и процедуры для сортировки данных

Для сортировки данных используется процедура void void Quick_Sort(). Доступ к записям базы данных осуществляется через индексный массив Less, который был создается процедурой int Less().

4. Особенности реализации бинарного поиска

Бинарный поиск по отсортированной базе осуществляется в процедуре int Search_In_Data_Base(), доступ к записям ведётся через индексный массив Lees, найденные записи заносятся в очередь struct List_Entry, которая очищается перед каждым следующим поиском. Для вывода найденных записей используется процедура void Print_List().

5. Вспомогательные функции и процедуры для построения АВЛ

Построение дерева осуществляется в процедуре void* Add_in_AVL().Поиск осуществляется в процедуре void Search_In_Tree().Обход построенного дерева выполняет процедура void Print_Tree().Поиск по дереву выполняет процедура void Search_In_Tree().

6. Кодирование данных

Кодирование базы происходит в функции void Coding_Symbols(). Кодирование данных начинается с процедуры Quick_Sort_For_Chance();, которая одновременно считает вероятности встречающихся символов и общее их количество. Кодирование происходит в процедуре void Shennon(). Дальше выводятся символы, их вероятности, длины кодовых слов и сами кодовые слова на монитор. В последствие вычисляются энтропия и средняя длина кодового слова.

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. СТРУКТУРЫ

```
struct Record
{
   char citizen[32];
   char street_name[18];
   short int number_of_house;
   short int number_of_flat;
   char check_in_date[10];
};
```
Запись, используемая для работы с базой данных.

```
struct List_Entry
{
        struct List_Entry* next;
        record* data;
};
```

```
struct Queue
{
        list_entry* head;
        list_entry* tail;
};
```
Структуры, использующиеся при сортировке базы данных.

```
struct Vertex
{
        list_entry* head_set;
        int balance;
        struct Vertex *left;
        struct Vertex *right;
};
```
Структура, представляющая АВЛ.

4.2. ОПИСАНИЕ ПОДПРОГРАММ

Процедуры, вывода меню:

1. void Draw_Menu()- выводит меню.
2. void Print_Head_Table()-вывод иформации
3. void Print_Variant_Choice()-меню при просмотре страниц

Процедуры начальной обработки базы данных:

5. int Read_Data_Base_To_Array()-считывает базу данных и формирует индексный массив.
6. void Print_Data_Base()-осуществляет просмотр базы данных.

Функции и процедуры сортировки:

7. void Quick_Sort()-сортирует базу данных по названию улицы и номеру дома.
8. int Less()- сравнение записей по названию улицы и номеру дома.
9. int Comparison_For_Search_Char_Field()- непосредственное сравнение строк.

Функции и процедуры для поиска в отсортированной базе данных:

10. int Search_In_Data_Base()- осуществляет поиск в базе.
11.int Create_Queue_Found_Note()-создается очередь.
12. void Print_Queue()- выводит на экран найденные записи.

Процедуры и функции построения АВЛ:

13. void* Add_in_AVL()- добавление записи в дерево.

Повороты:

14. void *LL_turn()
15. void *LR_turn()
16. void *RL_turn()
17. void *RR_turn()
18. void Print_Tree()-вывод дерева
19. void Search_In_Tree()-осуществляется поиск в дереве
20. void Print_List()-вывод найденного

Процедуры и функции кодирования базы данных:

21. int Reading_Symbols()-считывает символы.
22. void Fill_Chance_Meet_of_Symbols()-вероятность встречи символов.
23. void Quick_Sort_For_Chance()-сортировка вероятности.
24. void Shennon()-кодирование.
25. void Coding_Symbols()-общая процедура кодирования.

Основная программа:

26. main() - основная программа, в которой вызываются процедуры, рисующие меню, а также в зависимости от выбранного пункта меню вызываются соответствующие процедуры.

5. ТЕКСТ ПРОГРАММЫ

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#define ERROR_OPEN_BASE 0b00000001
#define ERROR_INCORECT_EXIT_FROM_PROGRAMM 0b00000010
#define ERROR_ALLOCATION_MEMORY 0b00000100
#define ERROR_NOT_FOUND_NOTE 0b00001000

#define SIZE_DATA_BASE 4000
#define START_RECORD_PRINT 20
#define SIZE_KEY_SORT 18
#define SIZE_KEY_SEARCH 3
#define LESS 0
#define EQUALLY 1
#define LARGE 2
#define QUANTITY_SYMBOL_IN_ALPHABET 166

#define FILE_NAME "testBase4.dat"
#define READ_BINARY_ONLY "rb"

bool height = false;

struct Record
{
   char citizen[32];
   char street_name[18];
   short int number_of_house;
   short int number_of_flat;
   char check_in_date[10];
};

typedef struct Record record;

struct List_Entry
{
        struct List_Entry* next;
        record* data;
};

typedef struct List_Entry list_entry;

struct Queue
{
        list_entry* head;
        list_entry* tail;
};

typedef struct Queue queue;

struct Vertex
{
        list_entry* head_set;
        int balance;
        struct Vertex *left;
        struct Vertex *right;
};

typedef struct Vertex vertex;

bool VR,HR;
```

```
void Draw_Menu()
{
	printf("1) Read Data Base\n");
   printf("2) Print Data Base\n");
   printf("3) Sort Data Base\n");
   printf("4) Find in Data Base by Street Name\n");
   printf("5) Find in Data Base by Number of House\n");
   printf("6) Encode Data Base by Shannon\n");
   printf("9) Clear Screen\n");
   printf("0) Exit\n");
}

void Print_Head_Table()
{
	printf("\nFIO\t\t\t\t\t");
	printf("Street name\t");
	printf("Number of house\t");
	printf("Number of flat\t");
	printf("Check in date\n");
}

void Print_Variant_Choice()
{
	printf("\nNext page - n\t");
	printf("Back page - b\t");
	printf("Choice specific page - s\t");
	printf("Change quantity record in one page - q\n");
	printf("Close - c\n");
}

void Print_Data_Base(record *index[])
{
   int Number = 0;
   int quantity_record_print = START_RECORD_PRINT;
   char key_continue = '1';
	while(1)
	{
		Print_Head_Table();
		for(short int i = Number; ((i < Number + quantity_record_print) && (i <SIZE_DATA_BASE)); i++)
		{
			printf("\n%d) %s\t", i + 1, index[i]->citizen);
			printf("%s\t", index[i]->street_name);
			printf("%d\t", index[i]->number_of_house);
			printf("%d\t", index[i]->number_of_flat);
			printf("%s", index[i]->check_in_date);
		}
		Print_Variant_Choice();
		scanf(" %c", &key_continue);
		switch(key_continue)
		{
		case('n'):
		{
			Number = Number + quantity_record_print;
			if(Number >= SIZE_DATA_BASE)
			{
				Number = SIZE_DATA_BASE - quantity_record_print;
			}
			break;
		}
		case('b'):
		{
			Number = Number - quantity_record_print;
			if(Number < 0)
			{
				Number = 0;
```

```
                        }
                        break;
                }
                case('c'):
                {
                        return (void)NULL;
                }
                case('s'):
                {
                        int key_specific = 0;
                        printf("Write need number page\n");
                        scanf("%d",&key_specific);
                        key_specific = key_specific - 1;
                        Number = (key_specific * quantity_record_print);
                        if(Number < 0 || Number >=SIZE_DATA_BASE)
                        {
                                printf("Need page not exist\n");
                                Number = 0;
                        }
                        break;
                }
                case('q'):
                {
                        printf("Write need quantity note in one page\n");
                        scanf("%d",&quantity_record_print);
                        break;
                }
                }
        }
        printf("\n");
}

void Print_Queue(queue* find)
{
        short int i = 1;
        find->tail = find->head;
        do
        {
                printf("\n%d) %s\t", i, find->tail->data->citizen);
                printf("%s\t", find->tail->data->street_name);
                printf("%d\t", find->tail->data->number_of_house);
                printf("%d\t", find->tail->data->number_of_flat);
                printf("%s", find->tail->data->check_in_date);
                i++;
                find->tail=find->tail->next;
        } while(find->tail!=NULL);
        printf("\n");
}

int Read_Data_Base_To_Array(record** note)
{
        FILE* file = NULL;
        file = fopen(FILE_NAME, READ_BINARY_ONLY);
        if(file == NULL)
        {
                return ERROR_OPEN_BASE;
        }
        *note = (record *) malloc (SIZE_DATA_BASE * sizeof(record));
        if(*note == NULL)
        {
                return ERROR_ALLOCATION_MEMORY;
        }
        fread(*note, sizeof(record), SIZE_DATA_BASE, file);
        fclose(file);
        return 0;
}
```

```
int Less(record* A, record* x)
{
	for(int i = 0; i < 18; i++)
	{
		if(A->street_name[i] < x->street_name[i])
		{
			return -1;
		}
		if(A->street_name[i] > x->street_name[i])
		{
			return 1;
		}
	}
	if(A->number_of_house < x->number_of_house)
	{
		return -1;
	}
	if(A->number_of_house > x->number_of_house)
	{
		return 1;
	}
	if(A->number_of_flat < x->number_of_flat)
	{
		return -1;
	}
	if(A->number_of_flat > x->number_of_flat)
	{
		return 1;
	}
	return 0;
}

void Quick_Sort(record** A,int L, int R)
{
	record* x;
	record* q;
	int i=L,j=R;
	 x=A[L];
    while (i<=j)
	 {
		while (Less(A[i],x)==-1) {i++;}
      while (Less(A[j],x)==1) {j--;}
			if (i<=j) {
	    q=A[i];
	    A[i]=A[j];
	    A[j]=q;
	    i++;
	    j--;
	  }
    }
    if (L<j) Quick_Sort(A,L,j);
    if (i<R) Quick_Sort(A,i,R);
}

int Comparison_For_Search_Char_Field(char check_data[], char search_data[])
{
	for(short int i = 0; i < SIZE_KEY_SEARCH; i++)
	{
		if(check_data[i] < search_data[i])
		{
			return LESS;
		}
		if(check_data[i] > search_data[i])
		{
			return LARGE;
```

```
                }
        }
        return EQUALLY;
}

int Search_In_Data_Base(record** index, char search_data[], int* index_first_need_note)
{
        short int left_limit = 0;
        short int right_limit = (SIZE_DATA_BASE - 1);
        short int index_midle = 0;
        do
        {
                index_midle = ((left_limit + right_limit) / 2);
                if(Comparison_For_Search_Char_Field(index[index_midle]->street_name, search_data) == LESS)
                {
                        left_limit = (index_midle + 1);
                }
                else
                {
                        right_limit = (index_midle - 1);
                }
        } while(left_limit <= right_limit);
        right_limit++;
        if(Comparison_For_Search_Char_Field(index[right_limit]->street_name, search_data) == EQUALLY)
        {
                *index_first_need_note = (right_limit);
                return 0;
        }
        *index_first_need_note = -1;
        printf("Need note not founded\n");
        return ERROR_NOT_FOUND_NOTE;
}

int Create_Queue_Found_Note(record** index, queue** find, int first_index)
{
        bool continue_flag = true;
        int number = 0;
        number = first_index;
        list_entry* le = NULL;
        le = (list_entry*)malloc(1 * sizeof(list_entry));
        *find = (queue*)malloc(1 * sizeof(queue));
   le->data = index[number];
   le->next = NULL;
   (*find)->head = le;
   (*find)->tail = le;
   number++;
   do
        {
                if(Comparison_For_Search_Char_Field(index[first_index]->street_name, index[number]->street_name) !=
EQUALLY)
                {
                        continue_flag = false;
                        break;
                }
                le = (list_entry*)malloc(1 * sizeof(list_entry));
                le->data = index[number];
        le->next = NULL;
                (*find)->tail->next = le;
                (*find)->tail = le;
                number++;
   } while((true == continue_flag) && (number < SIZE_DATA_BASE));
        return 0;
}

void Print_Tree(vertex *root)
{
```

```
	list_entry* temp = NULL;
	if(root != NULL)
	{
		Print_Tree(root->left);
		{
			temp = root->head_set;
			do
			{
				printf("\n%s\t", temp->data->citizen);
				printf("%s\t", temp->data->street_name);
				printf("%d\t", temp->data->number_of_house);
				printf("%d\t", temp->data->number_of_flat);
				printf("%s", temp->data->check_in_date);
				temp = temp->next;
			}while(temp != NULL);
		}
		Print_Tree(root->right);
	}
}

void Print_List(list_entry* head_set)
{
	int i = 0;
	do
	{
		i++;
		printf("\n%d)%s\t", i, head_set->data->citizen);
		printf("%s\t", head_set->data->street_name);
		printf("%d\t", head_set->data->number_of_house);
		printf("%d\t", head_set->data->number_of_flat);
		printf("%s", head_set->data->check_in_date);
		head_set = head_set->next;
	}while(head_set != NULL);
}

void *LL_turn(vertex **p)
{
	vertex *q;
	q=(*p)->left;
	q->balance=0;
	(*p)->balance=0;
	(*p)->left=q->right;
	q->right=(*p);
	(*p)=q;
}

void *LR_turn(vertex **p)
{
	vertex *q,*r;
	q=(*p)->left;
	r=q->right;
	if (r->balance<0)
	{
		(*p)->balance=1;
	}
	else
	{
		(*p)->balance=0;
	}
	if (r->balance>0)
	{
		q->balance=-1;
	}
	else q->balance=0;
	r->balance=0;
	(*p)->left=r->right;
```

```
	q->right=r->left;
	r->left=q;
	r->right=(*p);
	(*p)=r;
}

void *RR_turn(vertex **p)
{
	vertex *q;
	q=(*p)->right;
	q->balance=0;
	(*p)->balance=0;
	(*p)->right=q->left;
	q->left=(*p);
	(*p)=q;
}

void *RL_turn(vertex **p)
{
	vertex *q,*r;
	q=(*p)->right;
	r=q->left;
	if (r->balance>0)
	{
		(*p)->balance=-1;
	}
	else
	{
		(*p)->balance=0;
	}
	if (r->balance<0)
	{
		q->balance=1;
	}
	else
	{
		q->balance=0;
	}
	r->balance=0;
	(*p)->right=r->left;
	q->left=r->right;
	r->left=(*p);
	r->right=q;
	(*p)=r;
}

void Search_In_Tree(vertex* root, int data)
{
	if(root != NULL)
	{
		if(root->head_set->data->number_of_house < data)
		{
			Search_In_Tree(root->right,data);
		}
		if(root->head_set->data->number_of_house > data)
		{
			Search_In_Tree(root->left,data);
		}
		if(root->head_set->data->number_of_house == data)
		{
			Print_List(root->head_set);
			return (void)0;
		}
	}
}
```

```
void* Add_in_AVL(record *d,vertex **p)
{
	list_entry* temp = NULL;
	if((*p)==NULL)
	{
		(*p) = (vertex*)calloc(1,sizeof(vertex));
    (*p)->head_set = (list_entry*)calloc(1,sizeof(list_entry));
    (*p)->head_set->data = d;
    (*p)->left = NULL;
    (*p)->right = NULL;
    (*p)->balance = 0;
		height=true;
		return NULL;
	}
	else
	{
		if((*p)->head_set->data->number_of_house == d->number_of_house)
		{
			temp = (list_entry*)calloc(1,sizeof(list_entry));
		   temp->data = d;
		   temp->next = (*p)->head_set;
		   (*p)->head_set = temp;
		   return NULL;
		}
		if((*p)->head_set->data->number_of_house>d->number_of_house)
		{
			vertex *v=NULL;
			v=(*p)->left;
			Add_in_AVL(d,&v);
			(*p)->left=v;
			if (height==true)
			{
				if ((*p)->balance>0)
				{
					(*p)->balance=0;
					height=false;
				}
				else
				{
					if ((*p)->balance==0)
					{
						(*p)->balance=-1;
					}
					else
					{
						if ((*p)->left->balance<0)
						{
							LL_turn(p);
							height=false;
						}
						else
						{
							LR_turn(p);
							height=false;
						}
					}
				}
			}
		}
		else
		{
			if ((*p)->head_set->data->number_of_house<=d->number_of_house)
			{
				vertex *v=NULL;
				v=(*p)->right;
				Add_in_AVL(d,&v);
```

```
                                        (*p)->right=v;
                                        if (height==true)
                                        {
                                                if ((*p)->balance<0)
                                                {
                                                        (*p)->balance=0;
                                                        height=false;
                                                }
                                        }
                                        else
                                        {
                                                if ((*p)->balance==0)
                                                {
                                                        (*p)->balance=1;
                                                }
                                                else
                                                {
                                                        if ((*p)->right->balance>0)
                                                        {
                                                                RR_turn(p);
                                                                height=false;
                                                        }
                                                        else
                                                        {
                                                                RL_turn(p);
                                                                height=false;
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }

void Fill_Chance_Meet_of_Symbols(char symbol, double** array_chance, int symbol_alphabet)
{
   for(short int i = 0; i < symbol_alphabet; i++)
           {
      if(array_chance[i][0] == symbol)
                        {
         array_chance[i][1]++;
         return;
      }
                        if(array_chance[i][1] == 0)
                        {
         array_chance[i][0] = symbol;
         array_chance[i][1]++;
         return;
      }
   }
}

void Quick_Sort_For_Chance(double** array_chance, int left_limit, int right_limit)
{
   int j = right_limit, i = left_limit;
   double X = array_chance[(left_limit + right_limit) / 2][1];
   double temp, m;
   while (i <= j)
           {
      while(array_chance[i][1] > X)
                        {
         i++;
      }
      while(array_chance[j][1] < X)
                        {
         j--;
```

```
        }
        if(i <= j)
                        {
            m = array_chance[j][0];
            array_chance[j][0] = array_chance[i][0];
            array_chance[i][0] = m;
            temp = array_chance[j][1];
            array_chance[j][1] = array_chance[i][1];
            array_chance[i][1] = temp;
            i++;
            j--;
        }
    }
    if(left_limit < j)
            {
        Quick_Sort_For_Chance(array_chance, left_limit, j);
    }
    if(i < right_limit)
            {
        Quick_Sort_For_Chance(array_chance, i, right_limit);
    }
    return;
}

void Shennon(int size, double p[], double length[] ,bool code_word[][20])
{
    float Q[size];
    Q[0] = 0;
    length[0] = (int)ceilf(-log2f(p[0]));
    for (int i = 1; i < size; i++) {
        Q[i] = Q[i - 1] + p[i - 1];
        length[i] = (int)ceilf(-log2f(p[i]));
    }
    for(int i = 0; i < size; i++)
            {
        for (int j = 0; j < length[i]; j++) {
            Q[i] *= 2;
            int dig = (int)Q[i];
            if (dig == 0) code_word[i][j] = 0;
            if (dig == 1) code_word[i][j] = 1;
            if (Q[i] > 1)Q[i]--;
        }
    }
}

void Coding_Symbols(double** array_chance, int symbol_alphabet)
{
    double average_code_length = 0.0;
    double probability[symbol_alphabet + 1], length[symbol_alphabet + 1];
    bool code_word[symbol_alphabet + 1][20];
            for(short int i = 0; i < symbol_alphabet + 1; i++)
            {
        probability[i] = 0;
        length[i] = 0;
    }
    for(short int i = 0; i < symbol_alphabet + 1; i++)
            {
        for(short int j = 0; j < 20; j++)
                        {
            code_word[i][j] = false;
        }
    }
    Quick_Sort_For_Chance(array_chance, 0, symbol_alphabet - 1);
    for(short int i = 1; i < symbol_alphabet + 1; i++)
            {
        probability[i] = array_chance[i - 1][1];
```

```
    }
    printf("so good\n");
    Shennon(symbol_alphabet, probability, length, code_word);
    printf("\nSymbol\tProbability\t\tCode word\tWord length\n");
    for(short int i = 1; i < symbol_alphabet + 1; i++)
            {
        average_code_length += array_chance[i - 1][1] * length[i];
        printf("%hu)%c\t%f\t\t", i, (char)array_chance[i - 1][0], array_chance[i - 1][1]);
        for(short int j = 0; j < length[i]; j++)
                        {
            printf("%d", code_word[i][j]);
        }
        printf("\t\t%0.f\n", length[i]);
    }
    printf("Average length of code words: %f\n", average_code_length);
}

int Reading_Symbols()
{
            double** array_chance;
    FILE* file;
            double entropy = 0.0;
            int symbol_alphabet = QUANTITY_SYMBOL_IN_ALPHABET;
            int sum = 0;
            char symbol = '\0';
            file = fopen(FILE_NAME, READ_BINARY_ONLY);
            if(file == NULL)
            {
                    return ERROR_OPEN_BASE;
            }
    array_chance = (double**)calloc(symbol_alphabet, sizeof(double*));
    if(array_chance == NULL)
    {
            return ERROR_ALLOCATION_MEMORY;
            }
    for(short int i = 0; i < symbol_alphabet; i++)
            {
        array_chance[i] = (double*)calloc(2,sizeof(double));
        if(array_chance[i] == NULL)
        {
            return ERROR_ALLOCATION_MEMORY;
                    }
    }
    while(fread(&symbol, sizeof(char), 1, file) && (!feof(file)))
            {
        Fill_Chance_Meet_of_Symbols(symbol, array_chance, symbol_alphabet);
    }
    fclose(file);
    for(short int i = 0; i < symbol_alphabet; i++)
            {
        sum = (sum + array_chance[i][1]);
    }
    for(short int i = 0; i < symbol_alphabet; i++)
            {
        array_chance[i][1] = (array_chance[i][1] / sum);
    }
    for(short int i = 0; i < symbol_alphabet; i++)
            {
        entropy = (entropy + log(array_chance[i][1]) / log(2) * array_chance[i][1]);
    }
    entropy = entropy * -1;
    Coding_Symbols(array_chance, symbol_alphabet);
    printf("Entropy: %f\n", entropy);
    for(short int i = 0; i < symbol_alphabet; i++)
            {
        free(array_chance[i]);
```

```
        array_chance[i] = NULL;
    }
    free(array_chance);
    array_chance = NULL;
    return 0;
}

int main()
{
            vertex* root = NULL;
            queue *find = NULL;
    record** index = NULL;
    record* note = NULL;
    char* search_data = NULL;
    int error_flag = 0b0;
    int index_first_need_note = 0;
    int find_house = 0;
            char key_choice = '0';
            bool sort_flag = false;
            bool read_flag = false;
    do
    {
        Draw_Menu();
        scanf(" %c", &key_choice);
        switch (key_choice)
                        {
                        case('1'):
                        {
                                error_flag = (error_flag | Read_Data_Base_To_Array(&note));
                                index = (record** )malloc(SIZE_DATA_BASE * sizeof(record));
                                if(index == NULL)
                                {
                                        error_flag = (error_flag | ERROR_ALLOCATION_MEMORY);
                                        return error_flag;
                                }
                                for(short int i = 0; i < SIZE_DATA_BASE; i++)
                                {
                                        index[i] = &note[i];
                                }
                                read_flag = true;
                                sort_flag = false;
                                break;
                        }
                        case('2'):
        {
           if(note != NULL)
                                {
                                        Print_Data_Base(index);
                                }
                                else
                                {
                                        printf("Data Base is not Read\n");
                                }
                                break;
        }
        case('3'):
        {
            if((sort_flag == false) && (read_flag == true))
            {
                        Quick_Sort(index,0,SIZE_DATA_BASE-1);
                        sort_flag = true;
                                }
                                else
                                {
                                        printf("The database has already been sorted or database has not read\n");
                                }
```

```
                        break;
                }
                case('4'):
                {
                        if(sort_flag == true)
                        {
                                printf("Write need note to found\n");
                                search_data = (char*)calloc(SIZE_KEY_SEARCH,sizeof(char));
                                if(search_data == NULL)
                                {
                                        return (error_flag | ERROR_ALLOCATION_MEMORY);
                                }
                                fflush(stdin);
                                fread(search_data,sizeof(char),(SIZE_KEY_SEARCH + 1),stdin);
                                error_flag = (error_flag | Search_In_Data_Base(index, search_data,&index_first_need_note));
                                free(search_data);
                                search_data = NULL;
                                if(index_first_need_note != -1)
                                {
                                        error_flag = (error_flag | Create_Queue_Found_Note(index, &find,
index_first_need_note));
                                        Print_Queue(find);
                                }
                        }
                        else
                        {
                                printf("Need sort base\n");
                        }
                        break;
                }
                case('5'):
                {
                        if(find == NULL)
                        {
                                printf("Need Search by street name\n");
                                break;
                        }
                        find->tail = find->head;
                        do
                        {
                                Add_in_AVL(find->tail->data,&root);
                                find->tail=find->tail->next;
                        } while(find->tail != NULL);
                        Print_Tree(root);
                        printf("\nWrite need number of house\n");
                        scanf("%d",&find_house);
                        Search_In_Tree(root, find_house);
                        printf("\n");
                        if(root != NULL)
                        {
                                free(root);
                                root = NULL;
                        }
                        break;
                }
                case('6'):
                {
                        error_flag = (error_flag | Reading_Symbols());
                        break;
                }
                case('9'):
                {
                        system("CLS");
                        break;
                }
        case('0'):
```

```
        {
            return error_flag;
        }
        default:
        {
            printf("Incorrect choice\n");
            break;
                    }
        }
    } while(key_choice != '0');
    error_flag = (error_flag | ERROR_INCORECT_EXIT_FROM_PROGRAMM);
    for(short int i = 0; i < SIZE_DATA_BASE; i++)
            {
                    free(index[i]);
                    index[i] = NULL;
            }
            free(index);
            index = NULL;
            free(note);
            note = NULL;
            free(find);
            find = NULL;
    free(root);
    root = NULL;
            return error_flag;
}
```

6. РЕЗУЛЬТАТЫ

Рисунок 1.Меню.

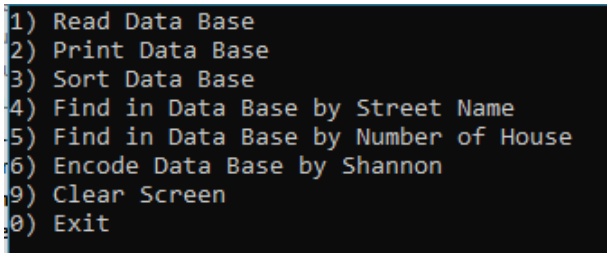


Рисунок 2. Неотсортированная база данных.

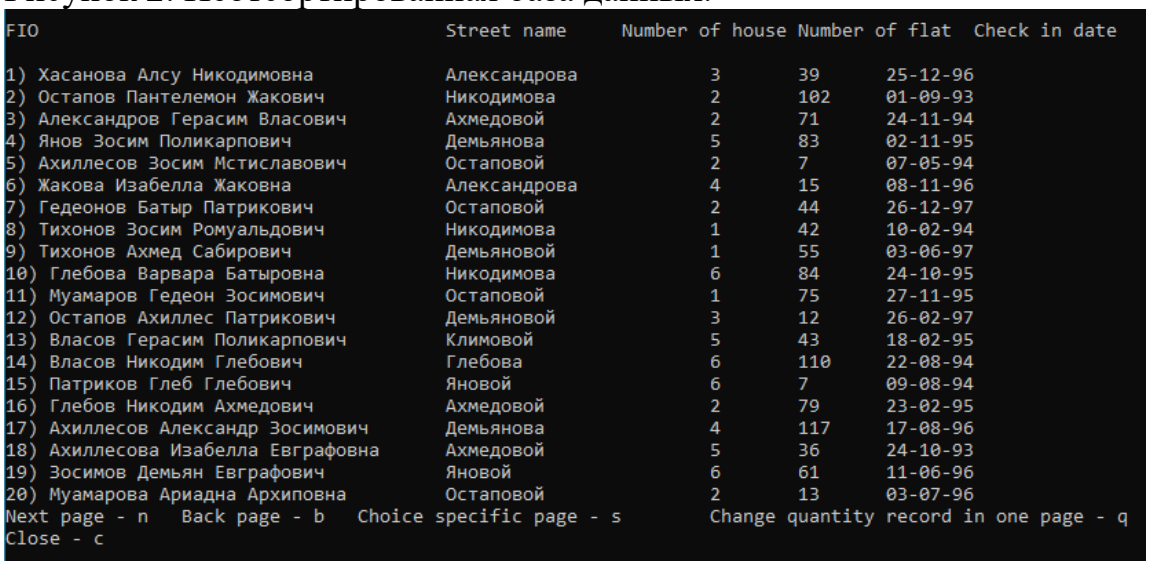


Рисунок 3. Отсортированная база данных.

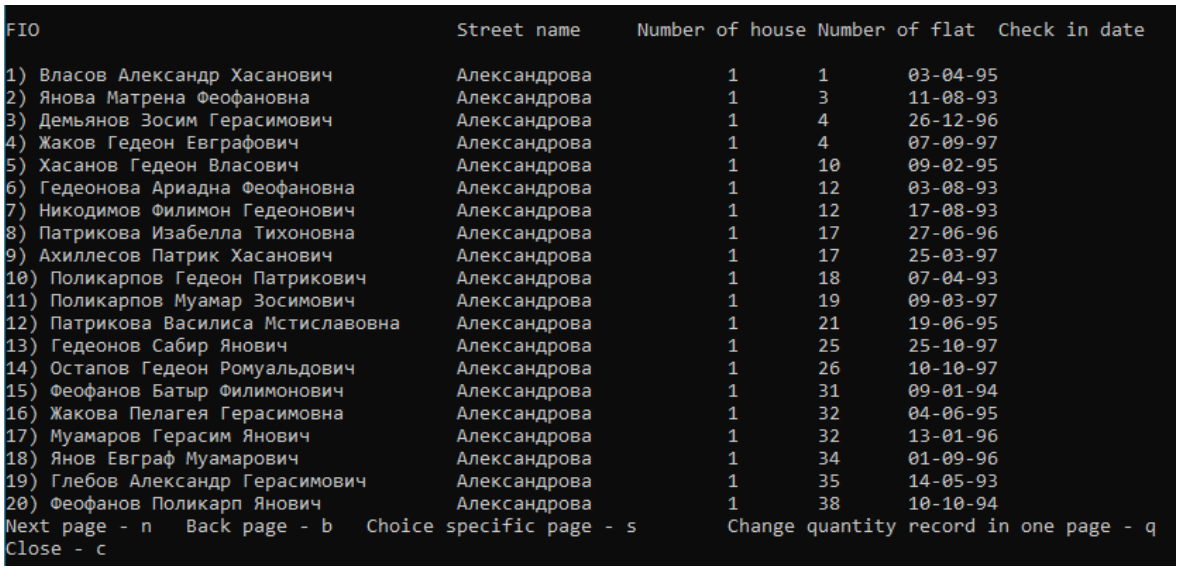
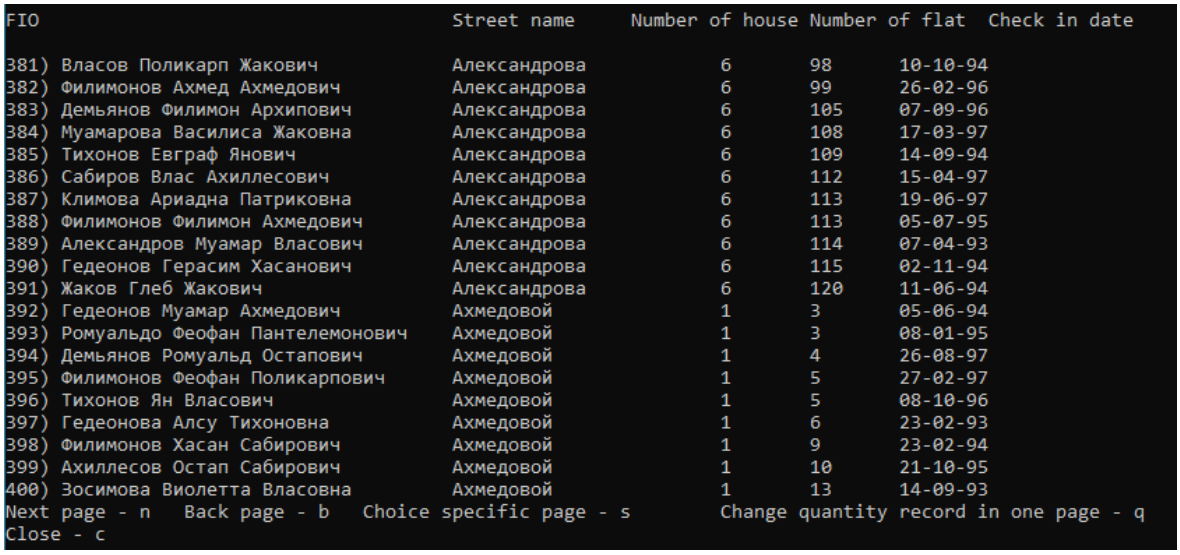


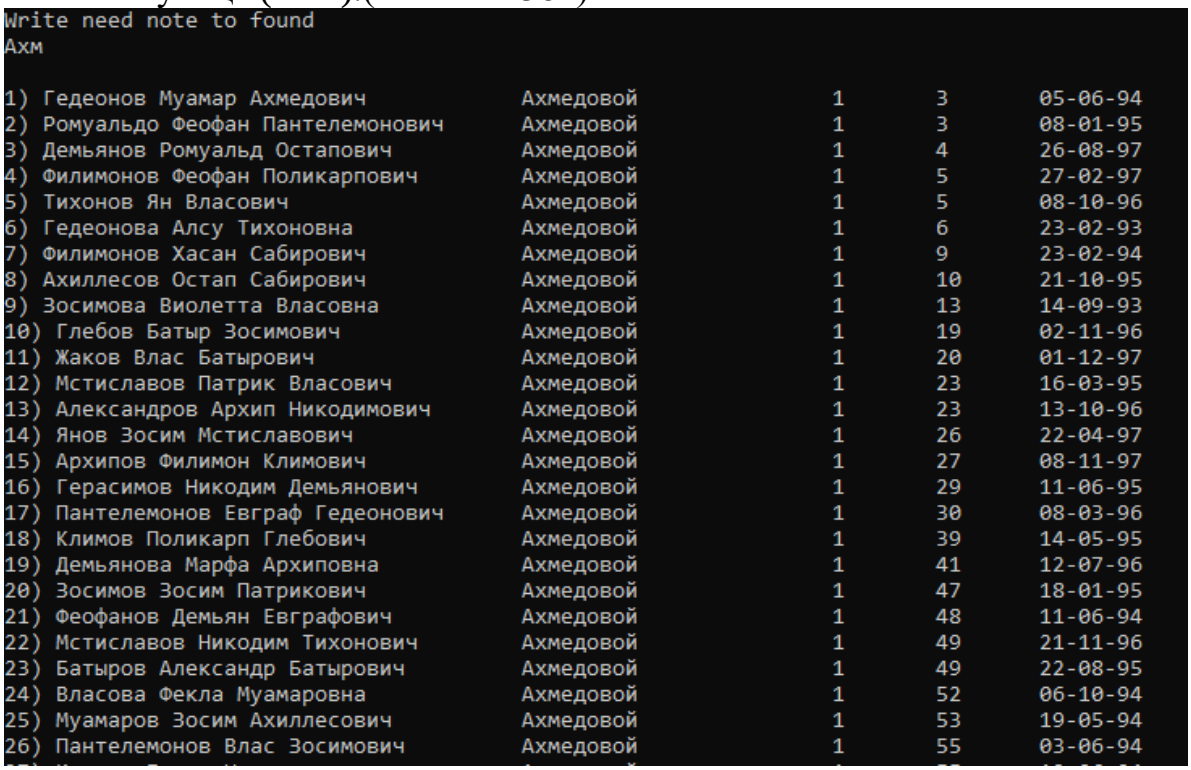
Рисунок 4. Страница, где более наглядно видно, что база данных отсортированная



FIO	Street name	Number of house	Number of flat	Check in date
381) Власов Поликарп Жакович	Александрова	6	98	10-10-94
382) Филимонов Ахмед Ахмедович	Александрова	6	99	26-02-96
383) Демьянов Филимон Архипович	Александрова	6	105	07-09-96
384) Муамарова Василиса Жаковна	Александрова	6	108	17-03-97
385) Тихонов Евграф Янович	Александрова	6	109	14-09-94
386) Сабиров Влас Ахиллесович	Александрова	6	112	15-04-97
387) Климова Ариадна Патриковна	Александрова	6	113	19-06-97
388) Филимонов Филимон Ахмедович	Александрова	6	113	05-07-95
389) Александров Муамар Власович	Александрова	6	114	07-04-93
390) Гедеонов Герасим Хасанович	Александрова	6	115	02-11-94
391) Жаков Глеб Жакович	Александрова	6	120	11-06-94
392) Гедеонов Муамар Ахмедович	Ахмедовой	1	3	05-06-94
393) Ромуальдо Феофан Пантелемонович	Ахмедовой	1	3	08-01-95
394) Демьянов Ромуальд Остапович	Ахмедовой	1	4	26-08-97
395) Филимонов Феофан Поликарпович	Ахмедовой	1	5	27-02-97
396) Тихонов Ян Власович	Ахмедовой	1	5	08-10-96
397) Гедеонова Алсу Тихоновна	Ахмедовой	1	6	23-02-93
398) Филимонов Хасан Сабирович	Ахмедовой	1	9	23-02-94
399) Ахиллесов Остап Сабирович	Ахмедовой	1	10	21-10-95
400) Зосимова Виолетта Власовна	Ахмедовой	1	13	14-09-93

Next page - n Back page - b Choice specific page - s Change quantity record in one page - q
Close - c

Рисунок 5. Очередь из записей, полученных в результате поиска по первым трем буквам названия улицы(Ахм).(нашлось 384)



Write need note to found
Ахм

1) Гедеонов Муамар Ахмедович	Ахмедовой	1	3	05-06-94
2) Ромуальдо Феофан Пантелемонович	Ахмедовой	1	3	08-01-95
3) Демьянов Ромуальд Остапович	Ахмедовой	1	4	26-08-97
4) Филимонов Феофан Поликарпович	Ахмедовой	1	5	27-02-97
5) Тихонов Ян Власович	Ахмедовой	1	5	08-10-96
6) Гедеонова Алсу Тихоновна	Ахмедовой	1	6	23-02-93
7) Филимонов Хасан Сабирович	Ахмедовой	1	9	23-02-94
8) Ахиллесов Остап Сабирович	Ахмедовой	1	10	21-10-95
9) Зосимова Виолетта Власовна	Ахмедовой	1	13	14-09-93
10) Глебов Батыр Зосимович	Ахмедовой	1	19	02-11-96
11) Жаков Влас Батырович	Ахмедовой	1	20	01-12-97
12) Мстиславов Патрик Власович	Ахмедовой	1	23	16-03-95
13) Александров Архип Никодимович	Ахмедовой	1	23	13-10-96
14) Янов Зосим Мстиславович	Ахмедовой	1	26	22-04-97
15) Архипов Филимон Климович	Ахмедовой	1	27	08-11-97
16) Герасимов Никодим Демьянович	Ахмедовой	1	29	11-06-95
17) Пантелемонов Евграф Гедеонович	Ахмедовой	1	30	08-03-96
18) Климов Поликарп Глебович	Ахмедовой	1	39	14-05-95
19) Демьянова Марфа Архиповна	Ахмедовой	1	41	12-07-96
20) Зосимов Зосим Патрикович	Ахмедовой	1	47	18-01-95
21) Феофанов Демьян Евграфович	Ахмедовой	1	48	11-06-94
22) Мстиславов Никодим Тихонович	Ахмедовой	1	49	21-11-96
23) Батыров Александр Батырович	Ахмедовой	1	49	22-08-95
24) Власова Фекла Муамаровна	Ахмедовой	1	52	06-10-94
25) Муамаров Зосим Ахиллесович	Ахмедовой	1	53	19-05-94
26) Пантелемонов Влас Зосимович	Ахмедовой	1	55	03-06-94

Рисунок 6. Дерево, ключ в дереве – номер дома.(3)

```
Write need number of house
3

1)Янов Архип Ахиллесович              Ахмедовой        3       120     14-09-95
2)Жакова Пелагея Батыровна            Ахмедовой        3       117     21-10-93
3)Зосимова Алсу Феофановна            Ахмедовой        3       117     05-07-93
4)Архипов Архип Климович              Ахмедовой        3       116     15-04-97
5)Гедеонов Архип Поликарпович         Ахмедовой        3       112     01-01-95
6)Янов Батыр Филимонович              Ахмедовой        3       109     05-11-96
7)Жаков Никодим Никодимович           Ахмедовой        3       109     25-09-95
8)Остапов Мстислав Филимонович        Ахмедовой        3       105     22-07-97
9)Архипов Ахмед Ромуальдович          Ахмедовой        3       103     03-08-95
10)Власов Герасим Зосимович           Ахмедовой        3       101     09-08-93
11)Ахмедов Мстислав Патрикович        Ахмедовой        3       99      07-09-97
12)Ахиллесов Остап Пантелемонович     Ахмедовой        3       98      24-11-93
13)Ахиллесова Ариадна Глебовна        Ахмедовой        3       94      14-05-96
14)Герасимова Ада Зосимовна           Ахмедовой        3       94      19-10-94
15)Климов Муамар Сабирович            Ахмедовой        3       93      05-05-95
16)Ахмедов Муамар Власович            Ахмедовой        3       86      06-12-93
17)Сабиров Зосим Гедеонович           Ахмедовой        3       86      05-05-97
18)Ахиллесов Зосим Архипович          Ахмедовой        3       84      17-08-97
19)Зосимов Ахиллес Сабирович          Ахмедовой        3       82      09-03-97
20)Остапов Остап Никодимович          Ахмедовой        3       76      28-03-95
21)Пантелемонов Влас Гедеонович       Ахмедовой        3       75      26-01-97
22)Феофанов Евграф Власович           Ахмедовой        3       74      25-10-96
23)Янов Муамар Глебович               Ахмедовой        3       74      25-09-94
24)Глебов Ян Филимонович              Ахмедовой        3       73      24-08-95
25)Сабиров Александр Герасимович      Ахмедовой        3       71      08-10-94
```

Рисунок 7. Примеры кодовых слов.

```
Symbol  Probability             Code word       Word length
1)      0.253816                00              2
2)      0.078125                0100            4
3)о     0.070414                0101            4
4)а     0.055051                01100           5
5)в     0.049977                01110           5
6)и     0.038797                10000           5
7)-     0.031422                10001           5
8)е     0.026910                100100          6
9)н     0.026867                100110          6
10)л    0.024012                101000          6
11)м    0.023395                101001          6
12)0    0.019676                101011          6
13)9    0.019070                101100          6
14)р    0.018594                101101          6
15)с    0.017047                101111          6
16)1    0.013828                1100000         7
17)д    0.013445                1100010         7
18)т    0.011289                1100011         7
19)ч    0.011027                1100101         7
```

Рисунок 8. Энтропия и средняя длина кодового слова.

```
Average length of code words: 5.079430
Entropy: 4.707176
```

7. ВЫВОДЫ

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, поиска, АВЛ дерева, поиска по дереву и кодирование базы данных.

В результате кодирования были получены данные, подтверждающие теоретические сведения. К таковым относятся: величины средней длины кодового слова и энтропии ($Lcp \leq H + 2$).

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

Реализованные алгоритмы представляют минимальный набор процедур для представления и обработки базы данных, а также отличаются достаточно высоким быстродействием и эффективностью.