Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовой проект

«Структуры и алгоритмы обработки данных»

Вариант

Выполнил: студент группы

Проверил: ассистент кафедры ПМиК

Новосибирск 2021

Оглавление

[1. Постановка задачи 2](#_Toc64501734)

[2. Основные идеи и характеристики применяемых методов 3](#_Toc64501735)

[2.1. Метод сортировки 3](#_Toc64501736)

[2.2. Двоичный поиск 3](#_Toc64501737)

[2.3. Списки и очереди 4](#_Toc64501738)

[2.4. Дерево и поиск по нему 4](#_Toc64501739)

[2.5. Метод кодирования 6](#_Toc64501740)

[4. Описание программы 10](#_Toc64501741)

[4.1. Основные переменные и структуры 10](#_Toc64501742)

[4.2. Описание подпрограмм 11](#_Toc64501743)

[5. Текст программы 12](#_Toc64501744)

[6. Результаты 28](#_Toc64501745)

[7. Выводы 31](#_Toc64501746)

# Постановка задачи

Хранящуюся в файле базу данных " Населенный пункт" загрузить динамически в оперативную память компьютера в виде списка, вывести на экран по 20 записей на странице с возможностью отказа от просмотра.

Упорядочить данные с помощью цифровой сортировки (Quick sort), построить по отсортированным данным индексный массив. Упорядоченные данные вывести на экран.

Предусмотреть возможность быстрого 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 поиска по ключу, вывести на экран содержимое дерева и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных кодом Шенона, предварительно оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран, вычислить среднюю длину кодового слова и сравнить ее с энтропией исходного файла.

База данных "Населенный пункт"

Стpуктуpа записи:

ФИО гражданина: текстовое поле 32 символа

фоpмат <Фамилия>\_<Имя>\_<Отчество>

Название улицы: текстовое поле 18 символов

Номер дома: целое число

Номер квартиры: целое число

Дата поселения: текстовое поле 10 символов

фоpмат дд-мм-гг

Пpимеp записи из БД:

Петpов\_Иван\_Федоpович\_\_\_\_\_\_\_\_\_\_\_

Ленина\_\_\_\_\_\_\_\_\_\_\_\_\_\_

10

67

29-02-65

Ваpианты условий упоpядочения и ключи поиска (К):

по улице и фио гражданина, К = первые три буквы фамилии. Ключ в дереве = название улицы

# Основные идеи и характеристики применяемых методов

## Метод сортировки

Быстрая сортировка (Quick Sort)

Метод Хоара или метод быстрой сортировки заключается в следующем. Возьмём произвольный элемент массива х. Просматривая массив слева, найдём элемент ai ≥x. Просматривая массив справа, найдём aj ≤x. Поменяем местами ai и aJ . 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 х, в правой части массива не меньше х. Затем к каждой части массива применяется тот же алгоритм.

Очевидно, трудоёмкость метода существенно зависит от выбора элемента х, который влияет на разделение массива. Максимальные значения М и С 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 М и С имеют следующий вид

M=*3(n-1),* C=*(n2+5n+4)/2*

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

Элемент *am* называется *медианой* для элементов *aL…aR,* если количество элементов меньших *am* равно количеству элементов больших *am* 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 М и С имеют следующий вид

С=О(*n* log *n*), М=О(*n* log *n*) при *n → ∞*.

Метод Хоара неустойчив.

## 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 f_02. Таким образом, трудоёмкость двоичного поиска в обоих случаях

f_03

## Списки и очереди

*Списком* называется последовательность однотипных элементов, связанных между собой указателями. Будем считать, что элементы списка имеют тип tLE, указатели на элементы списка имеют тип pLE.

X: tLE p: pLE

|  |
| --- |
| Next |
| Data |

Рисунок 13 Указатель на элемент списка

Поле Next является указателем на элемент списка и может занимать произвольное место в структуре элемента. Однако если оно является первым элементом структуры, то его адрес совпадает с адресом элемента списка, и это позволяет оптимизировать многие операции со списками. Поле Data содержит информацию, которая будет учитываться при сортировке.

Рассмотрим два вида списков: *стек* и *очередь*. *Стек* характеризуется тем, что новый элемент добавляется в начало последовательности, а удаляться может только первый элемент списка. При добавлении в *очередь* новый элемент ставится в конец списка, удаляется первый элемент последовательности.

Рассмотрим основные операции со стеком и очередью. Для работы со стеком необходимо иметь указатель на начало списка. Обозначим его Head. При работе с очередью требуется дополнительный указатель на конец очереди. Обозначим его Tail. Иногда при работе с очередью удобно объединять указатели Head и Tail в виде полей некоторой переменной Queue.

## Дерево и поиск по нему

**ДБД-дерево**

Деревья, имеющие вершины со многими потомками, будем называть сильноветвящимися. Такие деревья могут быть эффективно использованы для решения следующей задачи: формирование и поддержание крупномасштабных деревьев поиска, для которых необходимы включения и удаление элементов, но для которых либо не хватает оперативной памяти, либо она слишком дорога для долговременного использования.

В этом случае вершины дерева могут храниться во внешней памяти (например, на диске), тогда ссылки представляют собой адреса на диске, а не в оперативной памяти. Если использовать двоичное дерево для n = 106 элементов, то потребуется log(106) = 20 шагов поиска. Так как каждый шаг включает в себя обращение к диску, то необходимо минимизировать число обращений к диску. Сильноветвящиеся деревья – идеальное решение этой проблемы, т.к. при обращении к диску можно считывать не один элемент, а целую группу, причём размер сектора диска определяет размер минимальной порции (обычно кратен 512 байт). Таким образом, за одно обращение считывается поддерево, которое будем называть страницей. Например, пусть для дерева из 106 элементов размер страницы равен 100 вершин. Поиск будет требовать в среднем log100(106) = 3, а не 20 обращений к диску. Однако если дерево растёт случайным образом, то в худшем случае может потребоваться даже 104 обращений. Поэтому Р. Байером и Е. Маккрейтом был сформулирован критерий управления ростом дерева: каждая страница (кроме одной) должна содержать (при заданном постоянном m) от m до 2m элементов.

Таким образом, для дерева с n элементами и максимальном размером в 2m вершин в худшем случае потребуется logmn обращений к страницам (диску). При этом коэффициент использования памяти не меньше, чем 50%, так как страницы всегда заполнены минимум наполовину. Также эта схема позволяет достаточно просто осуществлять поиск, включения и удаления элементов.

Введем следующее определение. Б – дерево порядка m – это дерево со следующими свойствами:

В каждой странице хранится k элементов данных d1 < d2 < ... < dk и k+1 указатель p0, p1, ...pk. Каждый указатель pi либо равен NIL, либо указывает на вершину, все элементы которой больше di, но меньше di+1.

p0 d1 p1 d2 p2 … pk-1 dk pk

< d1

d1 < d < d2

> dk

Рисунок 49 Страница Б-дерева

Для каждой вершины, кроме корня m ≤ k ≤ 2m, для корня 1 ≤ k ≤ 2m.

Все листья дерева расположены на одном уровне.

Пример Б-дерева при m = 2.

Root

● 10 ● 20 ●

● 11 ● 13 ● 15●

● 21 ● 22 ● 27 ● 29 ●

● 2 ● 6●

Рисунок 50 Пример Б-дерева

Очевидно, количество обращений к диску равно высоте Б-дерева. Легко видеть, что минимальное количество вершин в Б-дереве при заданной высоте h определяется равенством nmin = 2(m+1)h-1 – 1. Отсюда высота Б-дерева порядка m с n элементами данных . Например, при m =255 высота h меньше, чем log n/8, т.е. по сравнению с обычными двоичными деревьями требуется в 8 раз меньше обращений к диску.

**Поиск в Б-дереве**

Если спроецировать Б – дерево на один уровень, то элементы расположатся в возрастающем порядке слева направо. Такое размещение определяет способ поиска элемента с заданным ключом. Страница считывается в оперативную память, а затем производится поиск среди элементов d1, d2, …,dk (упорядоченных!). Если k мало, то можно использовать простой последовательный поиск, если k достаточно большое, то – двоичный поиск.

## Метод кодирования

Код Щеннона

Код Шеннонапозволяет построить почти оптимальный код с длинами кодовых слов . Тогда по теореме Шеннона из п. 5.1

.

Код Шеннона, удовлетворяющий этому соотношению, строится следующим образом:

1. Упорядочим символы исходного алфавита А={*a*1*,a*2*,…,an*} по убыванию их вероятностей: *p*1*≥p*2*≥p*3*≥…≥pn*.
2. Вычислим величины *Qi*:, 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*, … , Qn=*1.

1. Представим *Qi* в двоичной системе счисления и возьмем в качестве кодового слова первые  знаков после запятой .

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

, .

**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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *ai* | *Pi* | *Qi* | *Li* | кодовое слово |
| *a*1  *a*2  *a*3  *a*4  *a*5  *a*6 | 1/22≤0.36<1/2  1/23≤0.18<1/22  1/23≤0.18<1/22  1/24≤0.12<1/23  1/24≤0.09<1/23  1/24≤0.07<1/23 | 0  0.36  0.54  0.72  0.84  0.93 | 2  3  3  4  4  4 | 00  010  100  1011  1101  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

*Lср*= 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*

Интерфейс программы реализован в теле функции Menu. Для выбора пунктов меню используется функция switch( ).

1. *Загрузка и вывод базы данных*

База данных открывается в Menu и считывается в список с помощью метода ReadBase. Считывание производится независимо от желания пользователя, в то время как большинство остальных функций он может выбрать посредствам меню. После считывания в список структур, файл закрывается.

За вывод элементов не отсортированный базы данных отвечает процедура void Print(list \*base) которая представляет возможность просмотра базы данных постранично.

1. *Вспомогательные функции и процедуры для сортировки данных*

База данных сортируется после запуска программы. Для сортировки базы данных используется процедура void QuickSort(list\* mas[], int left, int right, int sort). Она сортирует список по полям улицы и ФИО гражданина. Для быстрого доступа к отсортированной и неотсортированной базе данных, перед вызовом процедуры сортировки, делается индексный массив оригинальной базы данных, который и сортируется.

1. *Особенности реализации бинарного поиска*

Бинарный поиск по отсортированной базе данных осуществляется в функции int BSearch (list \*\*A, char \*FIO). Результатом работы является индекс элемента в массиве, удовлетворяющий условию ключа.

1. *Вспомогательные функции и процедуры для построения АВЛ-дерева.*

Построение дерева осуществляется в процедуре void addDBD(list\* binList, Vertex\*& p). Все повторяющиеся записи записываются в ветку next. Обход дерева слева направо осуществляется в процедуре void LR\_print(Vertex\* p, int& count). Поиск по дереву выполняется в процедуре void TreeSearch (Vertex\* p, char\* data).

1. *Кодирование данных*

Кодирование данных начинается с процедуры void Decode(), которая открывает файл базы данных для чтения, заполняет массив структур для алфавита кодовых слов всеми возможными символами, считывает из файла символы и считает их вероятности, закрывает файл и сортирует полученный алфавит по вероятностям. В процедуре void shanon() считается длина кодового слова и само кодовое слово.

Затем в процедур void Encode() база данных вновь прочитывается посимвольно и кодируется в новую, с помощью уже полученного кодового алфавита. В процедуре void CodePrint() осуществляется подсчет и вывод средней длины кодового слова, энтропии, а также выводит символы, их вероятности, длины кодовых слов и сами кодовые слова на монитор.

# Описание программы

## Основные переменные и структуры

struct country

{

char FIO[32];

char street[18];

short int number\_home;

short int number\_appart;

char Date[10];

};

Структура для хранения элемента базы данных. Всего таких элементов:

const int N = 4000;

struct list {

country \*data;

list\* prev;

list\* next;

};

Структура для создания списка, в котором хранятся элементы базы данных.

struct Vertex {

country \*data;

int Bal = 0;

Vertex \*Next;

Vertex\* Left;

Vertex\* Right;

};

Структура, представляющая АВЛ-дерево.

Vertex \*root = NULL;; - указатель на корень дерева

struct SN\_code {

float p;

float q;

int l;

char a;

};

Структура, представляющая собой информацию о символе для формирования кодового слова.

SN\_code A[M]; - массив для символов

const int M = 256 – число символов в алфавите

float entropy = 0 - энтропия

float lgm = 0 – средняя длина кодового слова

int v=0; - счётчик уникальных символов в файле

int sum =0 – счётчик всех символов в файле

## Описание подпрограмм

Процедуры работы с базой данных:

1. void Read\_base(FILE \*fp, list \*base) – прочитывание базы данных
2. void copy\_base(list\*, list\*) – дублирование прочитанной БД
3. void Print(list \*Base)- визуальный вывод списка на консоль.

Процедуры сортировки:

1. void QuickSort(list\* mas[], int left, int right, int sort) – сортирует базу данных по улице и ФИО гражданина.
2. int compare\_street(char\* a,char\* b) – сравнение полей улиц
3. int compare\_name(char\* a, char\* b) – сравнение полей имён
4. void Print\_sort\_index\_mas(list \*\*mas) – визуальный вывод отсортированного индексного массива на консоль

Процедуры и функции для поиска в отсортированной базе данных:

1. int BSearch (list \*\*A, char\* depositor) - бинарный поиск по 3 буквам фамилии.

Процедуры построения АВЛ-дерева:

1. void addDBD(list\* binList, Vertex\*& p) – построение ДБД-дерева.
2. void LR\_print(Vertex\* p, int& count) – вывод дерева.
3. void TreeSearch (Vertex\* p, char\* data) – поиск в дереве.

Процедуры и функции кодирования базы данных:

1. void Decode()– считывание символов базы данных.
2. void shanon() – подсчет вероятностей символов, преобразование алфавита и создание кодовых слов.
3. void Encode() – закодирование базы данных
4. void CodePrint()– вывод статистики и алфавита с вероятностями.

# Текст программы

#include <fstream>

#include <iostream>

#include <conio.h>

#include <Windows.h>

#include <iomanip>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <cmath>

#include <iomanip>

using namespace std;

const int N=4000;

const int M=256;

int sum=0;

int code[M][M];

float entropy = 0, lgm = 0;

int fcompression = 0, cfcompression = 0;

bool VR = true;

bool HR = true;

struct country

{

char FIO[32];

char street[18];

short int number\_home;

short int number\_appart;

char Date[10];

};

struct list {

country \*data;

list\* prev;

list\* next;

};

struct Vertex {

country \*data;

int Bal = 0;

Vertex \*Next;

Vertex\* Left;

Vertex\* Right;

};

Vertex \*root = NULL;

struct SN\_code {

float p;

float q;

int l;

char a;

};

SN\_code A[M];

int Menu();

void Read\_base(FILE \*fp, list \*base);

void copy\_base(list\*, list\*);

void Print(list \*Base);

void Print\_sort\_index\_mas(list \*\*mas);

void QuickSort(list\* mas[], int left, int right, int sort);

int compare\_street(char\* a,char\* b);

int compare\_name(char\* a, char\* b);

int BSearch (list \*\*A, char\* depositor);

void LR\_print(Vertex\* p, int& count);

void addDBD(list\* binList, Vertex\*& p);

void TreeSearch (Vertex\* p, char\* data);

int size(Vertex\*);

int height(Vertex\*);

int max(int, int);

int sdp(Vertex\*, int);

void Decode();

void shanon();

void CodePrint();

void Encode();

void LR\_print(Vertex\* p, int& count) {

if (p != NULL) {

LR\_print(p->Left, count);

cout << count+1<<") "<< p->data->FIO << "\t" << p->data->street << "\t" << p->data->number\_home

<< "\t"<< p->data->number\_appart << "\t" << p->data->Date << endl;

count++;

LR\_print(p->Next, count);

LR\_print(p->Right, count);

}

}

void addDBD(list\* binList, Vertex\*& p) {

if (p == NULL) {

p = new Vertex;

p->data = binList->data;

p->Bal = 0;

p->Next = NULL;

p->Left = NULL;

p->Right = NULL;

VR = true;

}

else if (compare\_street(p->data->street, binList->data->street) == 0) {

addDBD(binList, p->Next);

}

else if (compare\_street(p->data->street, binList->data->street) > 0) {

addDBD(binList, p->Left);

if (VR) {

if (p->Bal == 0) {

Vertex\* q = p->Left;

p->Left = q->Right;

q->Right = p;

p = q;

q->Bal = 1;

VR = 0;

HR = 1;

}

else {

p->Bal = 0;

VR = 1;

HR = 0;

}

}

else HR = 0;

}

else if (compare\_street(p->data->street, binList->data->street) < 0) {

addDBD(binList, p->Right);

if (VR) {

p->Bal = 1;

HR = 1;

VR = 0;

}

else if (HR) {

if (p->Bal == 1) {

Vertex\* q = p->Right;

p->Bal = 0;

q->Bal = 0;

p->Right = q->Left;

q->Left = p;

p = q;

VR = 1;

HR = 0;

}

else HR = 0;

}

}

}

void TreeSearch (Vertex\* p, char\* data){

if(p!=NULL) ///Поиск улицы

{

if(compare\_street(p->data->street,data) > 0){

TreeSearch(p->Left,data);

}else{

if(compare\_street(p->data->street,data) < 0){

TreeSearch(p->Right,data);

}else{

cout << "Street has been founded: "<< p->data->FIO << "\t" << p->data->street << "\t" << p->data->number\_home

<< "\t"<< p->data->number\_appart << "\t" << p->data->Date << endl;

TreeSearch(p->Next,data);

}

}

}

}

int Menu()

{

system("cls");

setlocale(LC\_ALL, "Russian");

FILE\* fp;

int search\_start=0;

int search=N-1;

fp = fopen("testBase4.dat", "rb");

list\* OriginBase = new list;

list\* IndexBase = new list;

Read\_base(fp,OriginBase);

fclose(fp);

copy\_base(OriginBase, IndexBase);

list \*mas[N];

for(int i=0;i<N;i++){

mas[i]=IndexBase;

IndexBase=IndexBase->next;

}

for(int i=0;i<2;i++)

QuickSort(mas, 0, N - 1, i);

Decode();

Encode();

while(true){

system("cls");

int SoD;

cout << "1.Standard list\n2.Sorted list\n3.BSearch\n4.TreePrint\n5.Coding\n6.Exit" << endl;

cin >> SoD;

switch (SoD)

{

case 1:

{

Print(OriginBase);

break;

}

case 2:

{

Print\_sort\_index\_mas(mas);

break;

}

case 3:

{

//ищем первые 3 буквы фамилии и строим по найденным данным дерево

cout <<endl;

cout << "Chto budem iskat(citizen's sirname')"<<endl;

char sirname[3];

SetConsoleCP(866);

cin >> sirname;

if(strcmp(sirname,"0")!=0){

search = BSearch(mas,sirname);

int fam\_pos=0;

char buffer[32];

if(search == -1){

cout << "Citizen doesn't exists''" << endl;

}

else{

SetConsoleCP(1251);

do{

if(search==0){

break;

}

else{

search--;

}

if(strncmp(mas[search]->data->FIO,sirname,3)!=0){

search++;

break;

}

}

while(1);

search\_start=search;

do{

search++;

if(search==N){

break;

}

if(strncmp(mas[search]->data->FIO,sirname,3)!=0){

break;

}

}

while(1);

cout <<endl <<endl<<"Founded "<<search-search\_start <<" pozitions ("<< search\_start<< " "<< search-1<<")"<<endl;

for(int i=search\_start;i<search;i++){

cout << "Citizen has been founded: "<<i<<" "<< mas[i]->data->FIO << "\t" << mas[i]->data->street << "\t" << mas[i]->data->number\_home

<< "\t"<< mas[i]->data->number\_appart << "\t" << mas[i]->data->Date << endl;

addDBD(mas[i], root);

}

}

}

break;

}

case 4:

{

//выводим дерево и инфу про него, затем ищем улицу

int count =0;

char street[18];

LR\_print(root, count);

cout<<endl;

printf("+------+----------+----------------+----------+\n");

printf("|%6d| UniqSize | Height | Mid. height |\n", N);

printf("+------+----------+----------------+----------+\n");

printf("| DBD |%10d|%10d|%16.2f|\n", size(root), height(root), (double)sdp(root, 1) / size(root));

cout<<endl<< "Street find: "<< endl;

SetConsoleCP(866);

cin >> street;

SetConsoleCP(1251);

char \*spc = " ";

strcat(street, spc);

if(strcmp(street,"0")!=0){

TreeSearch(root,street);

}

break;

}

case 5:

{

CodePrint();

break;

}

case 6:

{

system("PAUSE");

return 0;

}

default:

{

cout << "Invalid number entered\nPlease enter numder again" << endl;

system("PAUSE");

system("cls");

break;

}

}

\_getch();

}

return 0;

}

int main()

{

Menu();

return 0;

}

void Encode() {

FILE \*fp, \*fcoded;

fp = fopen("testBase4.dat", "rb");

fcoded = fopen("BaseCoded.dat", "wb");

char buffer;

while (!feof(fp)) {

fscanf(fp, "%c", &buffer);

fcompression++;

for (int i = 0; i < M; i++) {

if (buffer == (char)(i-128)) {

for (int j = 0; j < A[i].l; j++) {

putc(code[i][j], fcoded);

cfcompression++;

}

}

}

}

fclose(fp);

fclose(fcoded);

}

void CodePrint(){

lgm=0;

SetConsoleCP(866);

printf("\n\nКод Шенона: \n");

printf("-------------------------------------------------------------------------------\n");

printf("| Номер Символа | Символ | Вероятность | Кодовое слово | Длина кодового|\n");

printf("| | | | | слова |\n");

printf("|-----------------------------------------------------------------------------|\n");

SetConsoleCP(1251);

for (int i = 0; i < M && A[i].l != 0; i++)

{

printf("| %12d | %c | %2.6f | ",i, A[i].a, A[i].q);

for (int j = 1; j <= A[i].l; j++)

printf("%d", code[i][j]);

for (int j = A[i].l + 1; j < 18; j++)

printf(" ");

printf(" | %7d |\n", A[i].l);

printf("|-----------------------------------------------------------------------------|\n");

lgm += A[i].p \* A[i].l;

}

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

SetConsoleCP(866);

printf("| Энтропия | Средняя длина | Коэф сжатия |\n");

printf("|\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\n");

printf("| %10f | %10.5f | %10.5f |\n", entropy, lgm, (float)fcompression/cfcompression);

printf("|\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\n");

cout << endl << endl << entropy+1 <<" > "<< lgm <<endl <<endl;

}

void Decode(){

int i,j;

FILE \*fp;

fp=fopen("testBase4.dat", "rb");

for (i = 0; i < M; i++) {

A[i].p = 0;

A[i].l = 0;

A[i].q = 0;

A[i].a = (char)(i-128);

}

while (!feof(fp)) {

char c;

fscanf(fp, "%c", &c);

if (feof(fp))

break;

//cout << c<<" - " << (int)c <<endl;

A[c+128].p +=1;

sum++;

}

printf("\n");

fclose(fp);

//сортировка по количеству повторений символов

bool b = true;

while (b)

{

b = false;

for (int i = 1; i < M; i++)

{

if (A[i - 1].p < A[i].p)

{

SN\_code B = A[i-1];

A[i-1]=A[i];

A[i]=B;

b = true;

}

}

}

for (i = 0; i < M && A[i].p != 0; i++){

A[i].p /=sum;

entropy += A[i].p \* abs(log(A[i].p) / log(2));

}

shanon();

}

void shanon()

{

for (int i = 0; i < 166; i++)

{

A[i].q = A[i - 1].q + A[i].p;

A[i].l = ceil(-log(A[i].p) / log(2));

}

for (int i = 1; i < M; i++)

{

for (int j = 1; j <= A[i].l; j++)

{

A[i - 1].q \*= 2;

code[i][j] = floor(A[i - 1].q);

while (A[i - 1].q >= 1)

A[i - 1].q -= 1;

}

}

}

void Read\_base(FILE \*fp, list \*base){

country \*mas = new country();

fread((country \*)mas, sizeof(country), 1, fp);

base->data=mas;

base->prev=NULL;

base->next=NULL;

for (int i = 1; i < N; i++) {

mas = new country();

list \*base\_prev;

fread((country\*)mas, sizeof(country), 1, fp);

base\_prev = base;

base = base->next = new list();

base->data = mas;

base->prev = base\_prev;

base->next = NULL;

}

}

int compare\_street(char\* a,char\* b) {

char bufferA[18];

char bufferB[18];

strcpy(bufferA, a);

strcpy(bufferB, b);

//находим позицию первого пробела, чтоб понять какое слово короче

int space\_pos\_a = strchr(a,' ') - a;

int space\_pos\_b = strchr(b,' ') - b;

//выбираем длину кратчайшего слова

int count;

if(space\_pos\_a <= space\_pos\_b){

count = space\_pos\_a;

}

else{

count = space\_pos\_b;

}

//если слова совпали то первее стоит то, что короче

if(strncmp(a, b,count) == 0){

if(space\_pos\_a == space\_pos\_b){

return 0;

}

if(space\_pos\_a < space\_pos\_b){

return 1;

}

return -1;

}

//если слова не совпали то в сортируем в нормальном порядке

if(strncmp(a, b,count) > 0){

return 1;

}

return -1;

}

int compare\_name(char\* a, char\* b) {

char bufferA[32];

char bufferB[32];

strcpy(bufferA, a);

strcpy(bufferB, b);

//находим позицию первого пробела

int space\_pos\_a = strchr(a,' ') - a;

int space\_pos\_b = strchr(b,' ') - b;

//минимальное количество символов в фамилии 3

int count =4;

//если сравниваемые фамилии разных размеров выбираем меньший размер для сравнений

if(space\_pos\_a < space\_pos\_b){

count = space\_pos\_a;

}

else{

count = space\_pos\_b;

}

count -= 1;

//сравниваем фамилии

if(strncmp(bufferA, bufferB,count) > 0){

return 1;

}

if(strncmp(bufferA, bufferB,count) < 0){

return -1;

}

//если сравниваемые фамилии разных размеров выбираем меньший

if(space\_pos\_a > space\_pos\_b){

return 1;

}

if(space\_pos\_a < space\_pos\_b){

return -1;

}

//если фамилии одинаковы, то...

//находим позицию второго пробела

strcpy(bufferA, a);

strcpy(bufferB, b);

bufferA[space\_pos\_a]='a';

bufferB[space\_pos\_b]='a';

space\_pos\_a = strchr(bufferA,' ') - bufferA;

space\_pos\_b = strchr(bufferB,' ') - bufferB;

count = 4;

//если сравниваемые имена разных размеров выбираем меньший размер для сравнений

if(space\_pos\_a < space\_pos\_b){

count = space\_pos\_a;

}

else{

count = space\_pos\_b;

}

count -= 1;

//сравниваем имена

if(strncmp(bufferA, bufferB,count) > 0){

return 1;

}

if(strncmp(bufferA, bufferB,count) < 0){

return -1;

}

//если сравниваемые имена разных размеров выбираем меньший

if(space\_pos\_a > space\_pos\_b){

return 1;

}

if(space\_pos\_a < space\_pos\_b){

return -1;

}

//если имена одинаковы, то...

//находим позицию третьего пробела

bufferA[space\_pos\_a]='a';

bufferB[space\_pos\_b]='a';

space\_pos\_a = strchr(bufferA,' ') - bufferA;

space\_pos\_b = strchr(bufferB,' ') - bufferB;

count = 4;

//если сравниваемые отчества разных размеров выбираем меньший размер для сравнений

if(space\_pos\_a < space\_pos\_b){

count = space\_pos\_a;

}

else{

count = space\_pos\_b;

}

count -= 1;

//сравниваем отчества

if(strncmp(bufferA, bufferB,count) > 0){

return 1;

}

if(strncmp(bufferA, bufferB,count) < 0){

return -1;

}

//если сравниваемые имена разных размеров выбираем меньший

if(space\_pos\_a >= space\_pos\_b){

return 1;

}

return -1;

}

void QuickSort(list\* mas[], int left, int right, int sort)

{

int i, j;

list\* x;

list\* temp;

i = left; j = right;

x = mas[(left + right) / 2];

do {

if(sort==0){

while ((compare\_street(mas[i]->data->street, x->data->street) < 0) && (i < right))

i++;

while ((compare\_street(mas[j]->data->street, x->data->street) >= 0) && (j > left))

j--;

}else{

while ((compare\_name(mas[i]->data->FIO, x->data->FIO) < 0) && (i < right))

i++;

while ((compare\_name(mas[j]->data->FIO, x->data->FIO) > 0) && (j > left))

j--;

}

if (i <= j) {

temp = mas[i];

mas[i] = mas[j];

mas[j] = temp;

i++; j--;

}

} while (i <= j);

if (left < j)

QuickSort(mas, left, j, sort);

if (i < right)

QuickSort(mas, i, right, sort);

}

int BSearch (list \*\*A, char \*FIO) {

int l = 0, r = N, m = 0, fam\_pos=0;

char buffer[32];

while (l <= r) {

m=(l+r)/2;

strncpy(buffer, A[m]->data->FIO,3);

if (strncmp(buffer,FIO,3)==0) {

return m;

}

if (strncmp(buffer,FIO,3) <0)

l = m+1;

else r = m-1;

}

return -1;

}

void Print(list \*Base)

{

int i;

char n;

SetConsoleCP(1251);

cout << "1 for 4000 / 2 for 20 per click/ 3 for nothing"<< endl;

cin>>n;

if(n=='1'){

i=0;

while(i<N){

cout << i + 1 << "\t" << Base->data->FIO << "\t" << Base->data->street << "\t" << Base->data->number\_home

<< "\t"<< Base->data->number\_appart << "\t" << Base->data->Date << endl;

i++;

if(int result=\_kbhit()){

printf( "\nKey '%c' was pressed. Press 'e' key to exit.\n", \_getch());

char something;

cin>>something;

if(something=='e'){

return;

}

}

Base = Base->next;

}

}

if(n=='2'){

i=0;

int g=0;

for(int i=0;i<N;i=i+20){

while((g++)<20)

{

cout << i + 1 << "\t" << Base->data->FIO << "\t" << Base->data->street << "\t" << Base->data->number\_home

<< "\t"<< Base->data->number\_appart << "\t" << Base->data->Date << endl;

Base = Base->next;

}

if(g>=20){

cout << "y for continue" << endl;

cin >> n;

if (n=='y')

g=0;

else

return;

}

}

}

}

void Print\_sort\_index\_mas(list \*\*mas){

int i;

char n;

SetConsoleCP(1251);

cout << "1 for 4000 / 2 for 20 per click/ 3 for nothing"<< endl;

cin>>n;

if(n=='1'){

i=0;

while(i<N){

cout << i + 1 << "\t" << mas[i]->data->FIO << "\t" << mas[i]->data->street << "\t" << mas[i]->data->number\_home

<< "\t"<< mas[i]->data->number\_appart << "\t" << mas[i]->data->Date << endl;

i++;

if(int result=\_kbhit()){

printf( "\nKey '%c' was pressed. Press 'e' key to exit.\n", \_getch());

char something;

cin>>something;

if(something=='e'){

return;

}

}

}

}

if(n=='2'){

i=0;

int g=0;

for(int i=0;i<N;i=i+20){

while((g++)<20)

{

cout << i + 1 << "\t" << mas[i+g]->data->FIO << "\t" << mas[i+g]->data->street << "\t" << mas[i+g]->data->number\_home

<< "\t"<< mas[i+g]->data->number\_appart << "\t" << mas[i+g]->data->Date << endl;

}

if(g>=20){

cout << "y for continue" << endl;

cin >> n;

if (n=='y')

g=0;

else

return;

}

}

}

}

void copy\_base(list \*a, list \*b) {

b->prev = NULL;

b->data = a->data;

for (int i = 1; i < N; i++) {

a = a->next;

b->next = new list;

b->next->prev = b;

b = b->next;

b->data = a->data;

}

b->next = NULL;

}

int size(Vertex \*x)

{

if (x == NULL) {

return 0;

}

else {

return 1 + size(x->Left) + size(x->Right);

}

}

int max(int x, int y)

{

if (x > y) return x;

return y;

}

int height(Vertex \*x)

{

if (x == NULL) {

return 0;

}

else {

return 1 + max(height(x->Left), height(x->Right));

}

}

int sdp(Vertex \*x, int l)

{

if (x == NULL) {

return 0;

}

else {

return l + sdp(x->Left, l + 1) + sdp(x->Right, l + 1);

}

}

# Результаты

Рисунок 1. Неотсортированная база данных.

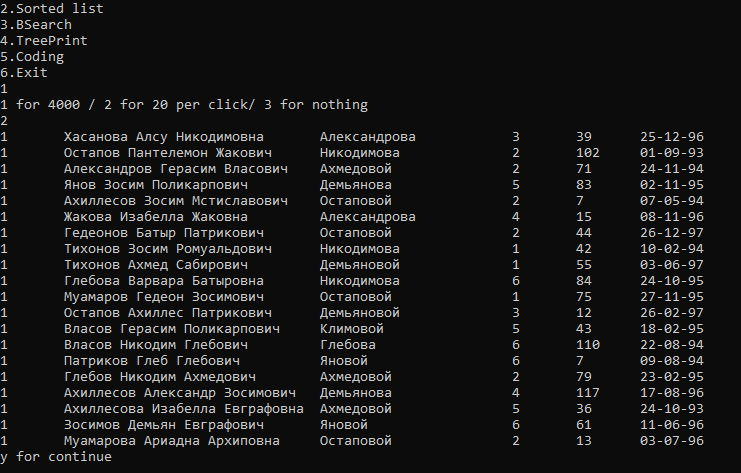


Рисунок 2. Отсортированная база данных по улице и ФИО.

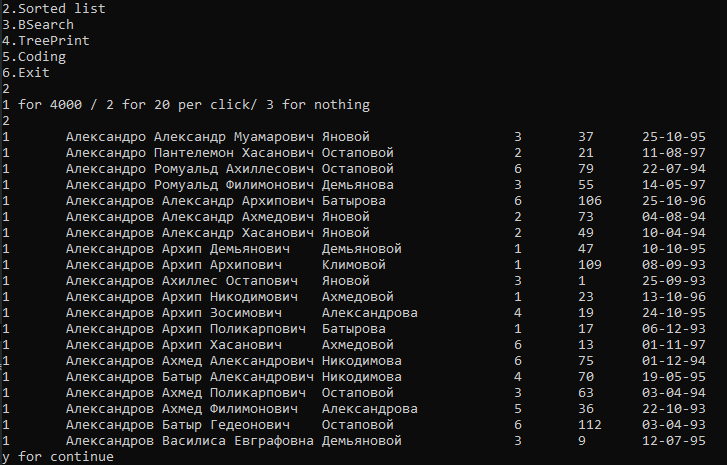


Рисунок 3. Бинарный поиск по ФИО

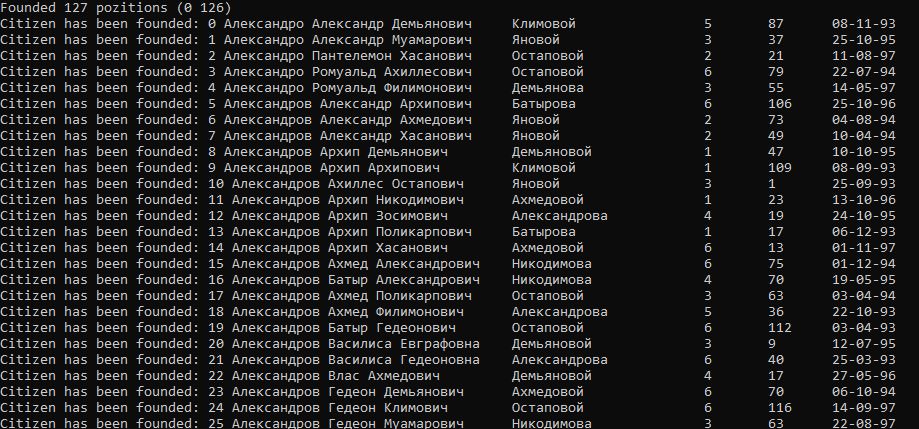


Рисунок 4. Дерево, ключ в дереве – улица.

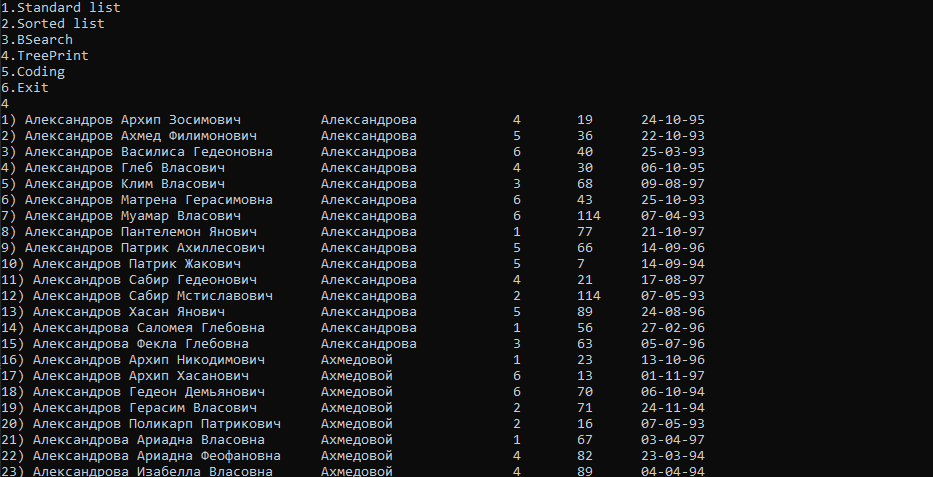


Рисунок 5. Поиск по дереву (элементы с одинаковым ключом).

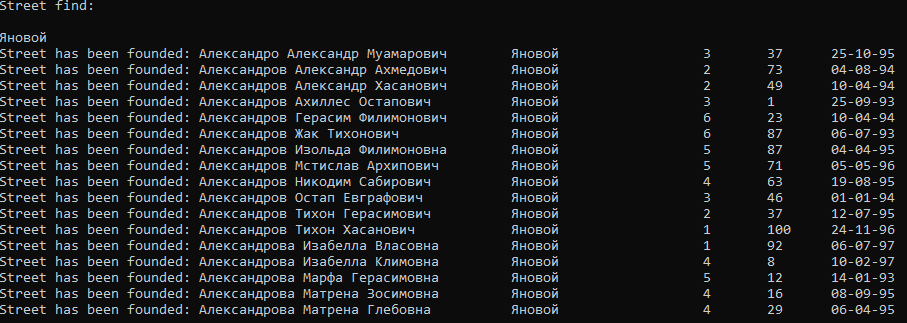
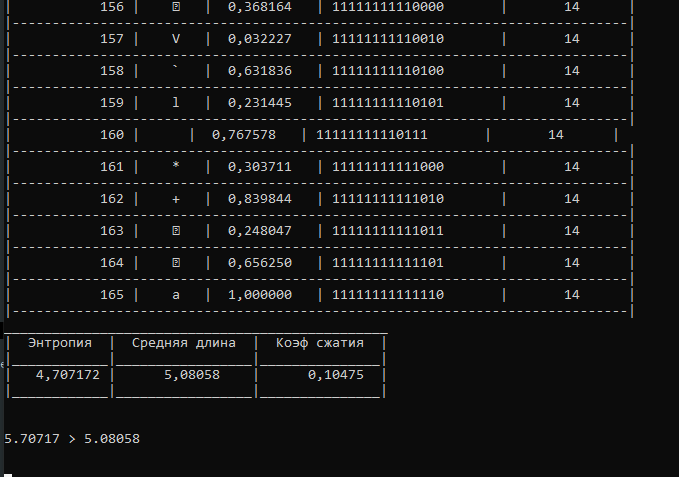


Рисунок 6. Кодовые слова, энтропия и средняя длина кодового слова.



# Выводы

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, поиска, построения ДБД-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 (Lср ≤ H + 1).

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.