Министерство цифрового развития, связи и массовых коммуникаций РФ

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовая работа

«Структуры и алгоритмы обработки данных»

Вариант 13

Выполнил: студент группы ИП-111

Кузьменок Д.В.

Проверил: доцент кафедры ПМиК

Янченко Е.В.

Новосибирск 2022

Содержание

[1. ПОСТАНОВКА ЗАДАЧИ 3](#_Toc230446878)

[2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ 4](#_Toc230446879)

[2.1. МЕТОД СОРТИРОВКИ 4](#_Toc230446880)

[2.2 ДВОИЧНЫЙ ПОИСК 5](#_Toc230446881)

[2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ 5](#_Toc230446882)

2.4 МЕТОД КОДИРОВАНИЯ………………………………………………………………5

[3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ 6](#_Toc230446883)

[4. ОПИСАНИЕ ПРОГРАММЫ 8](#_Toc230446884)

[4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ 8](#_Toc230446885)

[4.2. ОПИСАНИЕ ПОДПРОГРАММ 9](#_Toc230446886)

[5. ТЕКСТ ПРОГРАММЫ 12](#_Toc230446887)

[6. РЕЗУЛЬТАТЫ 27](#_Toc230446888)

[7. ВЫВОДЫ 32](#_Toc230446889)

# 1. ПОСТАНОВКА ЗАДАЧИ

1) Базу данных, которая хранится в файле под названием “testBase4.dat”, загрузить в память, и с помощью индексного массива упорядочить данные по ФИО и названию улицы, используя сортировку Вильямса-Флойда (пирамидальная сортировка).

2) Далее необходимо реализовать бинарный поиск по отсортированным записям с заданным ключом (первые три буквы фамилии), вследствие чего найденные записи добавляются в очередь с последующим выводом на экран.

3) Имея очередь найденных записей при двоичном поиске необходимо создать дерево оптимального поиска (приближенный алгоритм А2) с уникальным ключом (я выбрал номер дома и квартиры), и предусмотреть поиск по дереву. Если программа запускается несколько раз подряд, то необходимо создать функцию удаления дерева из памяти.

4) Закодировать файл базы данных статическим кодом Хаффмена, вывести вероятности символов, их кодовое слово, энтропию текста и среднюю длину слова.

B = 4 (файл testBase4.dat)

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):

C = 1 - по ФИО и названию улицы, К = пеpвые тpи буквы фамилии.

Ключ в дереве – Номер дома и квартиры.

# 

# 2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

# 2.1. МЕТОД СОРТИРОВКИ

Метод Вильямса-Флойда

Пирамидальная сортировка основана на алгоритме построения пирамиды. Последовательность ai, ai+1,…,ak называется *(i,k)-пирамидой*, если неравенство

aj≤min(a2j, а2j+1)

выполняется для каждого j, j=i,…,k для которого хотя бы один из элементов a2j, a2j+1 существует.

### *Свойства пирамиды:*

1. Если последовательность ai, ai+1,…,аk-1, ak является (*i, k*)-пирамидой, то последовательность ai+1,…,ak-1, полученная усечением элементов с обоих концов последовательности, является (*i+1, k-1*)пирамидой.
2. Если последовательность a1…an – (*1, n*)-пирамида, то а1 – минимальный элемент последовательности.
3. Если a1, a2…,an/2,an/2+1,…an-произвольная последовательность, то последовательность an/2+1,…,an  является (*n/2+1, n*)-пирамидой.

Пирамидальная сортировка производится в два этапа. Сначала строится пирамида из элементов массива. По свойству (3) правая часть массива является (*n/2+1, n*)-пирамидой. Будем добавлять по одному элементу слева, расширяя пирамиду, пока в неё не войдут все элементы массива. Тогда по свойству (2) первый элемент последовательности – минимальный.

Произведём двустороннее усечение: уберём элементы a1,an. Элемент a1 поставим на последнее место, а элемент an добавим к пирамиде a2,…,an-1 слева. Получается новая (1, n-1) пирамида. В ней первый элемент является минимальным. Поставим первый элемент пирамиды на позицию n-1, а элемент an-1 добавим к пирамиде a2,…,an-1, и т.д. В результате получается обратно отсортированный массив.

Общее количество операций сравнений и пересылок для пирамидальной сортировки: C ≤ 2*n* log *n*+*n*+2, M ≤ *n* log *n*+6.5*n*-4. Таким образом, С=O(*n* log *n*), М=O(*n* log *n*) при *n → ∞*.

Отметим некоторые свойства пирамидальной сортировки: метод пирамидальной сортировки неустойчив и не зависит от исходной отсортированности массива.

# 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.

Количество итераций не больше, чем . Таким образом, трудоёмкость двоичного поиска:



# 2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ

*Второй алгоритм* (А2) использует предварительно упорядоченный набор вершин. В качестве корня выбирается такая вершина, что разность весов левого и правого поддеревьев была минимальна. Для этого путем последовательного суммирования весов определим вершину *Vk*, для которой справедливы неравенства:

 и .

Тогда в качестве "центра тяжести" может быть выбрана вершина *Vk, Vk-1* или *Vk+1*, т. е. вершина, для которой разность весов левого и правого поддерева минимальна. Далее действия повторяются для каждого поддерева.

# 2.4 МЕТОД КОДИРОВАНИЯ

Код Хаффмена:

1. Упорядочим символы исходного алфавита А={*a1,…,an*} по убыванию их вероятностей *p1≥p2≥…≥pn*.
2. Если А={*a1,a2*}, то *a1*→0, *a2*→1.
3. Если А={*a1,…,aj,…,an*} и известны коды <*aj* → *bj*>, *j = 1,…,n* ,то для {*a1,…aj/ ,aj//…,an*}, *p(aj)=p(aj/)+ p(aj//), aj/  → bj0, aj// →bj1*.

Код Хаффмена обычно строится и хранится в виде двоичного дерева, в листьях которого находятся символы алфавита, а на «ветвях» – 0 или 1. Тогда уникальным кодом символа является путь от корня дерева к этому символу, по которому все 0 и 1 собираются в одну уникальную последовательность.

# 3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

При выполнении курсовой работы для корректной работы программы мне было необходимо создать дополнительные функции.

1. *Интерфейс программы*

Всё взаимодействие с программой реализовано в консоли. При запуске пользователю с помощью функции *Application()* будет предложен выбор: вывод базы данных, вывод отсортированной базы данных, бинарный поиск по базе данных и построение дерева по уникальному ключу (номер дома и квартиры) с возможностью поиска, а так же кодирование базы данных.

2. *Работа с базой данных*

Чтение базы данных происходит в *main()*. Для чтения базы данных я использовал функцию *fopen*. Здесь же создается массив I, в который записываются данные (структуры). Далее создается индексный массив ind для дальнейшей работы при сортировке с двумя ключами (“виртуальная” перестановка записей исходного массива).

За вывод базы данных на экран отвечает функция *Print* (), в которой реализована возможность пролистывать базу данных по 20 элементов, а также обратится к конкретной записи по ее порядковому номеру.

3. *Функции и процедуры для сортировки данных*

Для сортировки базы данных используется функции *CreatePyramid(struct locality\*\* ind, int L, int R)* и *HeapSort(struct locality\*\* ind)*. Доступ к записям базы данных осуществляется через индексный массив ind. Для сортировки по ФИО и названию улицы используется функция *Less(struct locality\* X, struct locality\* Y)* : изначально идет сравнение по ФИО. Если они равны, сравниваются уже символы названия улицы. Функция Less возвращает -1, если запись X меньше Y, и 1, если наоборот.

Функция *Less(struct locality\* X, struct locality\* Y)* вызывается у меня при построении пирамиды, чтобы происходило корректное построение пирамиды при сохранении её условия.

4. *Бинарный поиск*

Бинарный поиск по отсортированной базе осуществляется в процедуре *BiSearch(struct locality\*\* ind, char\* x),* доступ к записям ведётся через индексный массив ind, найденные записи заносятся в очередь *struct Queue*. Для вывода на монитор найденных записей используется процедура *print\_sorted(Queue\* pointers).*

В функции *BiSearch(struct locality\*\* ind, char\* x)* используются две функции *Check1(char\* a, char\* x)* и *Check2(char\* a, char\* x)*, когда первая функция возвращает 0 (если бинарный поиск надо вести в левой части массива), 1 (если записи найдены), а вторая – 1 (если записи одинаковые и последующее добавление этих записей в очередь) и 0 (если записи с одинаковым ключом закончены).

5. *Построение дерева оптимального поиска (приближённый алгоритм А2)*

Построение дерева осуществляется в процедуре *BiSearch(struct locality\*\* ind, char\* x).* Для начала у меня в данной функции считается сколько записей хранится в очереди после двоичного поиска, затем данные записи добавляются из очереди в массив при помощи функции *create\_array(int\*& weights, int N, array\_structs\_for\_tree\*& array\_tree, Queue\* pointers)* добавляются в массив array\_tree. Так же в этой функции каждой вершине задаётся вес, проинициализированный в функции *weight(int\*& weights, int N)* путём рандомного присвоения*.* Массив array\_tree сортируется по возрастанию значений ключей при помощи функции *CreatePyramid\_1(array\_structs\_for\_tree\*& array\_tree, int L, int R)* и *HeapSort\_1(array\_structs\_for\_tree\*& array\_tree, int size),* где реализована функция *Less\_1(struct locality\* X, struct locality\* Y)*, т.к. у меня сортировка идёт по двум ключам и затем передается в функцию *createA2(array\_structs\_for\_tree\* array\_tree, int L, int R),* а окончательное построение дерева поисходит в функции *build\_sdp(locality\* D, vertex\*& p, int W, int flag)*, куда просто заносятся записи. Для вывода построенного дерева на экран используется функция *obhod\_left\_right(vertex\* p, int flag).* Поиск записей по дереву используется в функции *search(vertex\* Root, struct find D, int flag).* Также для многократного использования поиска по дереву реализована функция *delete\_vertex(array\_structs\_for\_tree D, vertex\* Root, int flag)*.

6. *Кодирование*

При помощи функции *Sort(Byte\* arr, int num)*, символы сортируются по вероятностям(в порядке убывания). Кодирования данных происходит в функциях *Huffman(Byte\* P, int num), UP(int num, double q, Byte\* P) и DOWN(int num, int j) (*это вспомогательные функции для кодировки Хаффмена)*.* Также высчитывается средняя длина кодового слова *AverageLengthCodeWordHUFFMAN(Byte\* P, int num)* и энтропия исходного файла *Entropy(Byte\* P, int num)*. Вывод таблицы вида – символ-вероятность-кодовое слово-длина кодового слова происходит в Coding()в цикле.

# 4. ОПИСАНИЕ ПРОГРАММЫ.

## 4.1. **ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ**

struct locality

{

char name[32];

char street[18];

short int house;

short int flat;

char date[10];

};

Основная структура для работы с базой данных «Населённый пункт», в которой содержатся поля: ФИО, название улицы, номер дома и квартиры, дата поселения.

struct tLE

{

tLE\* next;

locality\* person;

};

Вспомогательная структура для работы с очередью, содержит поле next (указатель на следующий элемент) и указатель на основную структуру locality, чтобы в дальнейшем строить очередь из записей.

struct Queue

{

tLE\* head;

tLE\* tail;

} pointers;

Структура для добавления записей в очередь, содержит два указателя head и tail, для корректной записи данных в очередь.

struct array\_structs\_for\_tree

{

locality\* person;

int weight;

};

Структура для того, чтобы записи из очереди, которые добавляются при бинарном поиске, формировали массив структур, который потребуется для построения дерева. Более того, каждой ячейке массива будет присваиваться вес.

struct vertex

{

vertex\* left = NULL;

vertex\* right = NULL;

vertex\* mid = NULL;

locality\* person;

int wes;

} \*root;

Основная структура для построения дерева. Содержит указатель на данные из структуры locality, вес вершины, указатели на левое, правое, среднее поддеревья. \*root – указатель на корень дерева.

struct find

{

int house;

int flat;

};

Структура для ввода данных для поиска по дереву.

struct Byte

{

int ASCII\_symbol;

float probability;

};

Структура для работы с кодированием данных, содержит символ и его вероятность.

const int N = 4000 – константа количества записей из базы данных.

int order = 0 – константа для нумерации записей при обходе дерева.

const int max\_number = 256 – константа для максимального количества символов в коде ASCII.

int\* Length – массив длин кодовых слов при кодировке.

int\*\* C – матрица, в которой содержатся кодовые слова при кодировке.

## 

## 4.2. **ОПИСАНИЕ ПОДПРОГРАММ.**

Процедура вывода меню:

*void Application() –* выводит меню, с возможностью выбрать определенный пункт, и т.к. эта функция вызывается в int main() в операторе switch case, то в зависимости от выбора у нас программа будет выполнять следующие действия: вывод базы данных, сортировка базы данных, бинарный поиск и построение дерева и кодировка (в зависимости от введёного пункта будут вызываться определённые функции).

Процедуры вывода базы данных:

*void Print(struct locality\*\* data)* – вывод базы данных на экран, реализована возможность пролистывать базу данных по 20 элементов, а так же обратится к конкретной структуре по ее порядковому номеру.

Функции и процедуры сортировки:

*void CreatePyramid(struct locality\*\* ind, int L, int R)* – процедура, которая из входных данных делает частичную сортировку, т.е. выполняет условие пирамиды.

void HeapSort(struct locality\*\* ind) – окончательная сортировка базы данных по ФИО и названию улицы.

int Less(struct locality\* X, struct locality\* Y) - сравнение записей по ФИО и названию улицы, возвращает 1 или -1, которая в последующем используется в функции *CreatePyramid(struct locality\*\* ind, int L, int R)*.

Функции и процедуры для поиска в отсортированной базе данных:

*void BiSearch(struct locality\*\* ind, char\* x)* – поиск по отсортированной базе данных по первым трем буквам фамилии.

*int Check1(char\* a, char\* x)* - функция возвращает 0 (если бинарный поиск надо вести в левой части массива), 1 (если записи найдены).

*int Check2(char\* a, char\* x)* - 1 (если записи одинаковые и последующее добавление этих записей в очередь) и 0 (если записи с одинаковым ключом закончены).

*void completion(Queue\* pointers, locality\* person)* – процедура добавляет найденные записи из бинарного поиска в очередь Queue.

*int print\_sorted(Queue\* pointers)* – вывод на экран найденных записей при поиске.

Процедуры и функции построения дерева оптимального поиска (А2):

*void weight(int\*& weights, int N)* – создается массив случайных весов, чтобы размер массива совпадал с количеством записей в очереди.

*void create\_array(int\*& weights, int N, array\_structs\_for\_tree\*& array\_tree, Queue\* pointers)* – процедура заполняет массив структур и присваивает этим записям весы из массива весов.

*void CreatePyramid\_1(array\_structs\_for\_tree\*& array\_tree, int L, int R)* – частичная сортировка массива структур по номеру дома и квартиры.

*void HeapSort\_1(array\_structs\_for\_tree\*& array\_tree, int size)* – окончательная сортировка по номеру дома и квартире.

*int Less\_1(struct locality\* X, struct locality\* Y)* – сравнение сначала по номеру дома, а затем по квартире, и возвращаемое значение отправляется в функцию *void CreatePyramid\_1(array\_structs\_for\_tree\*& array\_tree, int L, int R).*

*void createA2(array\_structs\_for\_tree\* array\_tree, int L, int R)* – работа с записями, чтобы они удовлетворяли условию построения дерева А2.

*void obhod\_left\_right(vertex\* p, int flag)* – вывод построенного дерева (обход слева направо).

*void search(vertex\* Root, struct find D, int flag)* – поиск по дереву.

*vertex\* delete\_vertex(array\_structs\_for\_tree D, vertex\* Root, int flag)* – удаление дерева после построения, чтобы при повторном запуске программы у нас в памяти не сохранялось прошлое дерево.

Процедуры и функции кодирования базы данных:

*double Entropy(Byte\* P, int num)* – подсчёт энтропии базы данных.

*float AverageLengthCodeWordHUFFMAN(Byte\* P, int num)* – высчитывается кодовое слово.

*void Huffman(Byte\* P, int num)* – основной алгоритм кодировки Хаффмена.

*int UP(int num, double q, Byte\* P)* – функция поиска подходящей позиции для суммы вероятностей двух последних символов, её вставка и сдвиг остальных элементов.

*void DOWN(int num, int j)* – процедура формирования кодовых слов.

*void Sort(Byte\* arr, int num)* – сортировка символов по убыванию вероятностей.

*void Coding()* – кодирование базы данных, выводы таблицы кодовых слов, энтропии и средней длины.

Основная программа:

*int main()* – основная функция, в которой вызываются все остальные процедуры при помощи switch case.

# 

# 5. ТЕКСТ ПРОГРАММЫ

// 13, B:4 C:1 S:1 D:4 E:1

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <cmath>

#include <cstring>

#include <time.h>

#include <fstream>

using namespace std;

const int N = 4000;

int order = 0;

const int max\_number = 256;

struct locality

{

char name[32];

char street[18];

short int house;

short int flat;

char date[10];

};

struct tLE

{

tLE\* next;

locality\* person;

};

struct Queue

{

tLE\* head;

tLE\* tail;

} pointers;

struct array\_structs\_for\_tree

{

locality\* person;

int weight;

};

struct vertex

{

vertex\* left = NULL;

vertex\* right = NULL;

vertex\* mid = NULL;

locality\* person;

int wes;

} \*root;

struct find

{

int house;

int flat;

};

struct Byte

{

int ASCII\_symbol;

float probability;

};

void Print(struct locality\*\* data);

void Application();

void HeapSort(struct locality\*\* ind);

void CreatePyramid(struct locality\*\* ind, int L, int R);

int Less(struct locality\* X, struct locality\* Y);

void BiSearch(struct locality\*\* ind, char\* x);

int Check1(char\* a, char\* x);

int Check2(char\* a, char\* x);

void completion(Queue\* pointers, locality\* person);

int print\_sorted(Queue\* pointers);

void weight(int\*& weights, int N);

void create\_array(int\*& weights, int N, array\_structs\_for\_tree\*& array\_tree, Queue\* pointers);

void CreatePyramid\_1(array\_structs\_for\_tree\*& array\_tree, int L, int R);

void HeapSort\_1(array\_structs\_for\_tree\*& array\_tree, int size);

void createA2(array\_structs\_for\_tree\* array\_tree, int L, int R);

void build\_sdp(locality\* D, vertex\*& p, int W, int flag);

void obhod\_left\_right(vertex\* p, int flag);

void search(vertex\* Root, struct find D, int flag);

vertex\* delete\_vertex(array\_structs\_for\_tree D, vertex\* Root, int flag);

int Less\_1(struct locality\* X, struct locality\* Y);

void Coding();

void Sort(Byte\* arr, int num);

void DOWN(int num, int j);

int UP(int num, double q, Byte\* P);

void Huffman(Byte\* P, int num);

float AverageLengthCodeWordHUFFMAN(Byte\* P, int num);

double Entropy(Byte\* P, int num);

int\* Length;

int\*\* C;

int main() {

int i, j;

char KEY[3];

srand(time(0));

FILE\* fl = fopen("testBase4.dat", "rb");

root = NULL;

struct locality\* I = (struct locality\*)malloc(N \* sizeof(struct locality));

struct locality\*\* ind = (struct locality\*\*)malloc(N \* sizeof(struct locality\*));

for (j = 0; j < N; j++) {

ind[j] = &I[j];

}

fread((locality\*)I, sizeof(locality), N, fl);

while (1) {

Application();

cin >> i;

switch (i) {

case 1:

system("cls");

Print(ind);

system("pause");

system("cls");

break;

case 2:

system("cls");

HeapSort(ind);

Print(ind);

system("pause");

system("cls");

break;

case 3:

system("cls");

cout << "BiSearch" << endl;

cout << endl << "Key of search: ";

cin >> KEY;

BiSearch(ind, KEY);

system("pause");

system("cls");

break;

case 4:

system("cls");

cout << "Coding" << endl;

Coding();

system("pause");

system("cls");

break;

case 5:

cout << "Goodbye!" << endl;

exit(0);

default:

system("cls");

cout << "Wrong value. Enter right number!" << endl;

system("pause");

system("cls");

break;

}

}

delete[] I;

delete[] ind;

return 0;

}

void Application() {

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "|1 | To see the 4 data base |" << endl;

cout << "|--|------------------------------------------|" << endl;

cout << "|2 | To sort data base for the name and street|" << endl;

cout << "|--|------------------------------------------|" << endl;

cout << "|3 | Search and A2-tree (for flats) |" << endl;

cout << "|--|------------------------------------------|" << endl;

cout << "|4 | Coding |" << endl;

cout << "---|------------------------------------------|" << endl;

cout << "|5 | Exit |" << endl;

cout << "-----------------------------------------------" << endl;

}

void Print(struct locality\*\* data) {

int count = 0, ans;

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

printf("%d) ", i + 1);

printf("%32s", data[i]->name);

printf("%19s", data[i]->street);

printf("%7d", data[i]->house);

printf("%7d", data[i]->flat);

printf("%12s\n", data[i]->date);

count++;

if (count == 20) {

count = 0;

cout << "Do you want to see next 20 writings?(1/0)";

cin >> ans;

if (ans == 1) {

continue;

}

else if (ans == 0) {

break;

}

else {

i = ans - 2;

}

}

}

}

void CreatePyramid(struct locality\*\* ind, int L, int R) {

int j = 0, i;

struct locality\* a;

a = ind[L];

i = L;

while (1) {

j = 2 \* i;

if (j > R) break;

if ((j < R) && (Less(ind[j + 1], ind[j]) >= 0)) j++;

if ((Less(a, ind[j]) >= 0)) break;

ind[i] = ind[j];

i = j;

}

ind[i] = a;

}

void HeapSort(struct locality\*\* ind) {

int L, R;

struct locality\* temp;

L = ceil(N / 2);

while (L >= 0) {

CreatePyramid(ind, L, N - 1);

L--;

}

R = N - 1;

while (R >= 0) {

temp = ind[0];

ind[0] = ind[R];

ind[R] = temp;

R--;

CreatePyramid(ind, 0, R);

}

}

int Less(struct locality\* X, struct locality\* Y) {

if (strcmp(X->name, Y->name) > 0) { return 1; }

else if (strcmp(X->name, Y->name) < 0) { return -1; }

else if (strcmp(X->street, Y->street) > 0) { return 1; }

else if (strcmp(X->street, Y->street) < 0) { return -1; }

else { return 0; }

}

void BiSearch(struct locality\*\* ind, char\* x) {

int m, L = 0, R = N - 1, count = 0, \* weights;

struct find number;

pointers.tail = (tLE\*)&pointers.head;

while (L < R) {

m = (L + R) / 2;

if (Check1(ind[m]->name, x)) L = m + 1;

else R = m;

}

if (Check2(ind[R]->name, x)) {

for (int i = L; Check2(ind[R]->name, x) == 1;) {

completion(&pointers, ind[i]);

i++;

R++;

if (R + 1 > N) break;

}

}

else {

cout << "Key not found!" << endl;

system("pause");

system("cls");

}

count = print\_sorted(&pointers);

array\_structs\_for\_tree\* info = (struct array\_structs\_for\_tree\*)malloc(sizeof(struct array\_structs\_for\_tree) \* count);

weight(weights, count);

create\_array(weights, count, info, &pointers);

HeapSort\_1(info, count);

createA2(info, 0, count - 1);

obhod\_left\_right(root, 1);

order = 0;

cout << "Enter the house that you want to search: ";

cin >> number.house;

cout << "Enter the flat that you want to search: ";

cin >> number.flat;

search(root, number, 0);

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

root = delete\_vertex(info[i], root, 0);

}

}

int Check1(char\* a, char\* x) {

for (int i = 0; i < 3; i++) {

if (a[i] == x[i]) {

continue;

}

else if (a[i] > x[i]) {

return 0;

}

else {

return 1;

}

}

return 0;

}

int Check2(char\* a, char\* x) {

for (int i = 0; i < 3; i++) {

if (a[i] == x[i]) {

continue;

}

else if (a[i] != x[i]) {

return 0;

}

}

return 1;

}

void completion(Queue\* pointers, locality\* person) {

tLE\* p;

p = new tLE;

p->next = 0;

p->person = person;

pointers->tail->next = p;

pointers->tail = p;

}

int print\_sorted(Queue\* pointers) {

int count = 0;

tLE\* p;

p = pointers->head;

while (p != NULL) {

cout << order + 1 << ") ";

cout << "Name: " << p->person->name;

cout << " Street: " << p->person->street;

cout << " House: " << p->person->house;

cout << " Flat: " << p->person->flat;

cout << " Date: " << p->person->date << endl;

cout << "---------------------------------" << endl;

order++;

count++;

p = p->next;

}

order = 0;

cout << endl;

return count;

}

void weight(int\*& weights, int N) {

int i;

weights = (int\*)malloc(sizeof(int) \* N);

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

weights[i] = 1 + rand() % 100;

}

}

void create\_array(int\*& weights, int N, array\_structs\_for\_tree\*& array\_tree, Queue\* pointers) {

tLE\* p = pointers->head;

int i;

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

array\_tree[i].person = p->person;

array\_tree[i].weight = weights[i];

p = p->next;

}

}

void CreatePyramid\_1(array\_structs\_for\_tree\*& array\_tree, int L, int R) {

int j = 0, i;

array\_structs\_for\_tree a;

a = array\_tree[L];

i = L;

while (1) {

j = 2 \* i;

if (j > R) break;

if ((j < R) && (Less\_1(array\_tree[j + 1].person, array\_tree[j].person) >= 0)) j++;

if ((Less\_1(a.person, array\_tree[j].person) >= 0)) break;

array\_tree[i] = array\_tree[j];

i = j;

}

array\_tree[i] = a;

}

void HeapSort\_1(array\_structs\_for\_tree\*& array\_tree, int size) {

int L, R;

array\_structs\_for\_tree temp;

L = ceil(size / 2);

while (L >= 0) {

CreatePyramid\_1(array\_tree, L, size - 1);

L--;

}

R = size - 1;

while (R >= 0) {

temp = array\_tree[0];

array\_tree[0] = array\_tree[R];

array\_tree[R] = temp;

R--;

CreatePyramid\_1(array\_tree, 0, R);

}

}

int Less\_1(struct locality\* X, struct locality\* Y) {

if (X->house > Y->house) { return 1; }

else if (X->house < Y->house) { return -1; }

else if (X->flat > Y->flat) { return 1; }

else if (X->flat < Y->flat) { return -1; }

else { return 0; }

}

void createA2(array\_structs\_for\_tree\* array\_tree, int L, int R)

{

int i, wes = 0, sum = 0;

if (L <= R)

{

for (i = L; i <= R; i++) wes += array\_tree[i].weight;

for (i = L; i <= R; i++)

{

if ((sum < wes / (float)2) && (sum + array\_tree[i].weight >= wes / (float)2)) break;

sum += array\_tree[i].weight;

}

build\_sdp(array\_tree[i].person, root, array\_tree[i].weight, 0);

createA2(array\_tree, L, i - 1);

createA2(array\_tree, i + 1, R);

}

}

void build\_sdp(locality\* D, vertex\*& p, int W, int flag)

{

if (p == NULL)

{

p = (struct vertex\*)malloc(sizeof(struct vertex));

p->person = D;

p->wes = W;

p->left = NULL;

p->right = NULL;

p->mid = NULL;

}

else if (flag == 0) {

if (D->house < p->person->house)

{

build\_sdp(D, p->left, W, 0);

}

else if (D->house > p->person->house)

{

build\_sdp(D, p->right, W, 0);

}

else if (D->house == p->person->house) {

build\_sdp(D, p->mid, W, 1);

}

}

else if (flag == 1) {

if (D->flat < p->person->flat) build\_sdp(D, p->left, W, 1);

else if (D->flat >= p->person->flat) build\_sdp(D, p->right, W, 1);

}

else

{

cout << "The vertex exists in the tree!" << endl;

}

}

void obhod\_left\_right(vertex\* p, int flag)

{

if (p != NULL)

{

obhod\_left\_right(p->left, flag);

if (p->mid != NULL)

{

build\_sdp(p->person, p->mid, p->wes, 1);

flag = 1;

obhod\_left\_right(p->mid, flag);

flag = 0;

}

if (flag == 1)

{

cout << order + 1 << ") ";

order++;

cout << "Name: " << p->person->name;

cout << " Street: " << p->person->street;

cout << " House: " << p->person->house;

cout << " Flat: " << p->person->flat;

cout << " Date: " << p->person->date << endl;

cout << "---------------------------------" << endl;

}

flag = 1;

obhod\_left\_right(p->right, flag);

}

}

void search(vertex\* Root, struct find D, int flag)

{

vertex\* p = (struct vertex\*)malloc(sizeof(struct vertex));

p = Root;

while (p != NULL)

{

if (flag == 0)

{

if (D.house < p->person->house) p = p->left;

else if (D.house > p->person->house) p = p->right;

else if (D.house == p->person->house) p = p->mid, flag = 1;

}

else if (flag == 1)

{

if (D.flat < p->person->flat) p = p->left;

else if (D.flat > p->person->flat) p = p->right;

else if (D.flat == p->person->flat) break;

}

else break;

}

if (p != NULL) {

cout << order + 1 << ") ";

order++;

cout << "Name: " << p->person->name;

cout << " Street: " << p->person->street;

cout << " House: " << p->person->house;

cout << " Flat: " << p->person->flat;

cout << " Date: " << p->person->date << endl;

cout << "---------------------------------" << endl;

if (p->right != NULL) search(p->right, D, 1);

}

}

vertex\* delete\_vertex(array\_structs\_for\_tree D, vertex\* Root, int flag)

{

vertex\*\* p;

p = &Root;

vertex\* q, \* r, \* s;

while (\*p != NULL)

{

if (flag == 0)

{

if (D.person->house < (\*p)->person->house) p = &((\*p)->left);

else if (D.person->house > (\*p)->person->house) p = &((\*p)->right);

else if (D.person->house == (\*p)->person->house) break;

}

else break;

}

if (\*p != NULL)

{

q = \*p;

if (q->left == NULL) \*p = q->right;

else if (q->right == NULL) \*p = q->left;

else

{

r = q->left;

s = q;

if (r->right == NULL)

{

r->right = q->right;

\*p = r;

}

else

{

while (r->right != NULL)

{

s = r;

r = r->right;

}

s->right = r->left;

r->left = q->left;

r->right = q->right;

\*p = r;

}

}

free(q);

}

return Root;

}

double Entropy(Byte\* P, int num)

{

double sum = 0;

for (int i = 0; P[i].probability != 0; i++)

{

sum += -P[i].probability \* log2(P[i].probability);

}

return sum;

}

float AverageLengthCodeWordHUFFMAN(Byte\* P, int num)

{

float sum = 0;

for (int i = 1; i <= num; i++) sum += P[i].probability \* Length[i];

return sum;

}

void Huffman(Byte\* P, int num)

{

double q;

int j;

if (num == 2)

{

C[1][1] = 0;

Length[1] = 1;

C[2][1] = 1;

Length[2] = 1;

}

else

{

q = P[num - 1].probability + P[num].probability;

j = UP(num, q, P);

Huffman(P, num - 1);

DOWN(num, j);

}

}

int UP(int num, double q, Byte\* P)

{

int j = 1;

for (int i = num - 1; i >= 2; i--)

{

if (P[i - 1].probability < q) P[i] = P[i - 1];

else

{

j = i;

break;

}

}

P[j].probability = q;

return j;

}

void DOWN(int num, int j)

{

int\* S = new int[1024 + 1];

for (int k = 1; k <= 1024; k++) S[k] = -1;

for (int k = 1; C[j][k] != -1; k++) S[k] = C[j][k];

int length\_temp = Length[j];

for (int i = j; i <= num - 2; i++)

{

for (int k = 1; C[i + 1][k] != -1; k++) C[i][k] = C[i + 1][k];

Length[i] = Length[i + 1];

}

for (int k = 1; S[k] != -1; k++) C[num - 1][k] = S[k];

for (int k = 1; S[k] != -1; k++) C[num][k] = S[k];

C[num - 1][length\_temp + 1] = 0;

C[num][length\_temp + 1] = 1;

Length[num - 1] = length\_temp + 1;

Length[num] = length\_temp + 1;

delete[] S;

}

void Sort(Byte\* arr, int num)

{

int j, k, i;

Byte temp;

for (i = 0; i < num - 1; i++)

{

k = i;

for (j = i + 1; j < num; j++)

if (arr[j].probability > arr[k].probability) k = j;

temp = arr[i];

arr[i] = arr[k];

arr[k] = temp;

}

}

void Coding()

{

Byte P[max\_number];

int bytes[max\_number] = { 0 };

int sum = 0;

unsigned char x;

ifstream infile;

infile.open("testBase4.dat", ios::binary | ios::in);

while (1)

{

infile.read((char\*)&x, 1);

if (infile.eof())

break;

bytes[x]++;

sum++;

}

for (int i = 0; i < max\_number; ++i)

{

P[i].ASCII\_symbol = i;

P[i].probability = (float)bytes[i] / (float)sum;

}

Sort(P, max\_number);

double entropy\_text = Entropy(P, max\_number);

int num = 0;

for (int i = 0; i < max\_number; i++)

{

if (P[i].probability != 0) num++;

}

Length = new int[num + 1];

C = new int\* [num + 1];

for (int i = 1; i < num + 1; i++) C[i] = new int[num + 1];

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

{

for (int j = 1; j < num + 1; j++) C[i][j] = -1;

}

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

{

Length[i] = 0;

}

Byte P\_Huffman\_Work[max\_number + 1], P\_Huffman\_Print[max\_number + 1];

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

{

P\_Huffman\_Work[i].ASCII\_symbol = P[i - 1].ASCII\_symbol;

P\_Huffman\_Work[i].probability = P[i - 1].probability;

}

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

{

P\_Huffman\_Print[i].ASCII\_symbol = P[i - 1].ASCII\_symbol;

P\_Huffman\_Print[i].probability = P[i - 1].probability;

}

Huffman(P\_Huffman\_Work, num);

float avg\_length\_code\_word\_HUFFMAN = AverageLengthCodeWordHUFFMAN(P\_Huffman\_Print, num);

cout << "HUFFMAN" << endl;

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "| Symbol | Probability | Code Word | Length |" << endl;

cout << "|----------|----------------|---------------|----------|" << endl;

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

{

printf("|%6c - %d | ", P\_Huffman\_Print[i].ASCII\_symbol, P\_Huffman\_Print[i].ASCII\_symbol);

printf(" %5lf | ", P\_Huffman\_Print[i].probability);

for (int j = 1; j <= Length[i]; j++) printf("%d", C[i][j]);

printf(" | %5d |", Length[i]);

printf("\n");

}

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "Entropy: " << entropy\_text << endl;

cout << "Average length of code word: " << avg\_length\_code\_word\_HUFFMAN << endl;

for (int i = 1; i <= num; i++) delete C[i];

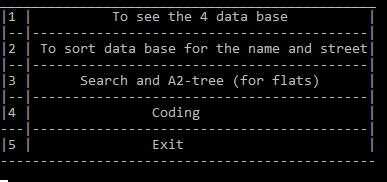
delete[] Length;

delete[] C;

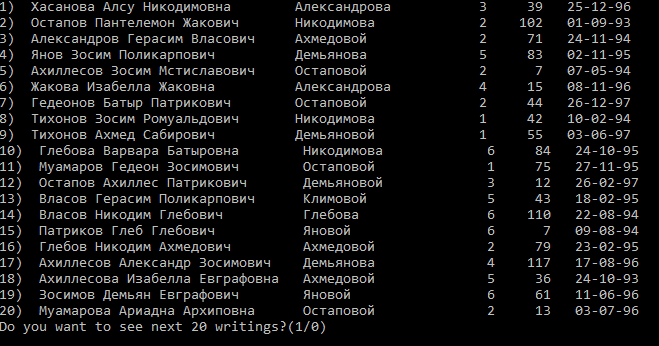
}

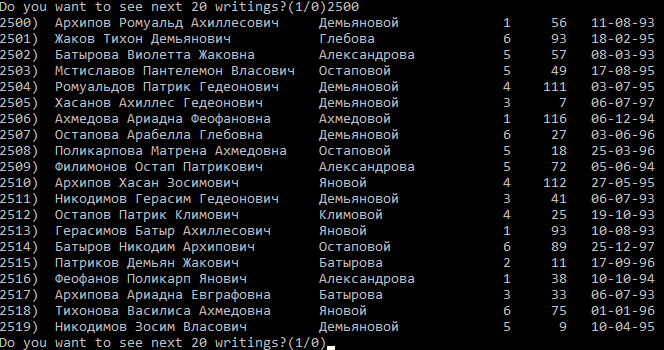
# 6. РЕЗУЛЬТАТЫ

1. Меню выбора (пользователь вводит с клавиатуры цифру для дальнейшей работы программы).

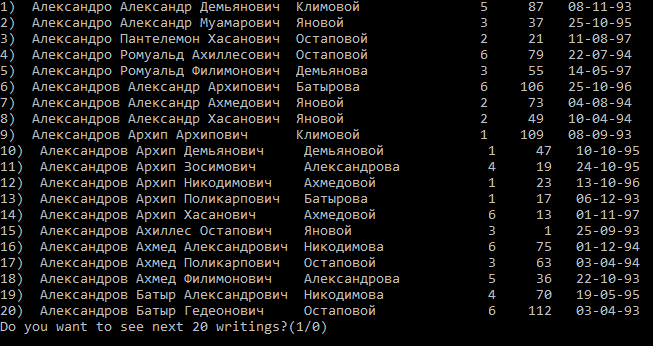


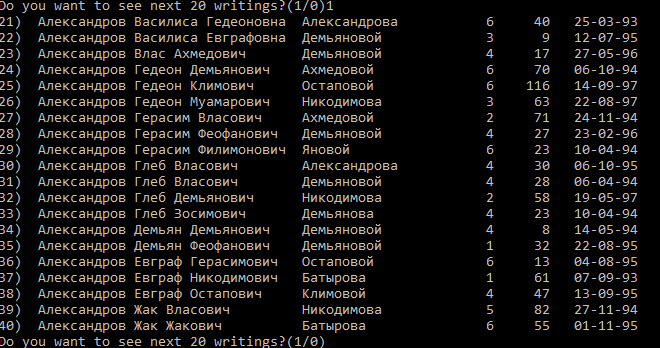
2. Чтение базы данных из файла (неотсортированная). На второй фотографии показано, что можно ввести номер определенной записи, и тогда вместе с ней выведется следующие 20 записей.

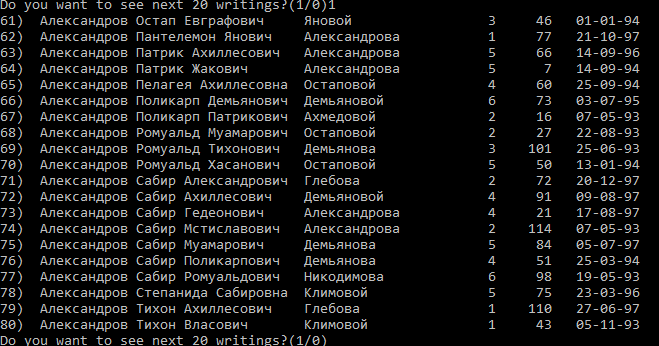




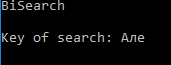
3. Отсортированная база данных (первичный ключ ФИО, а затем название улицы). Например, 30 и 31 запись: ФИО одинаковое, значит записи сортируются по названию улицы.

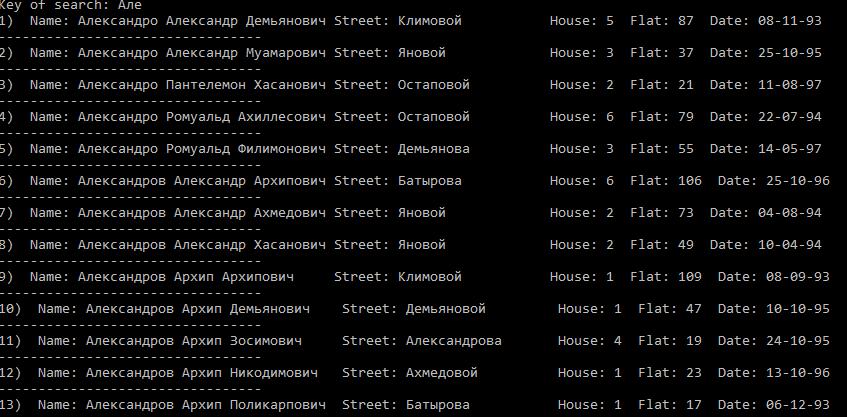




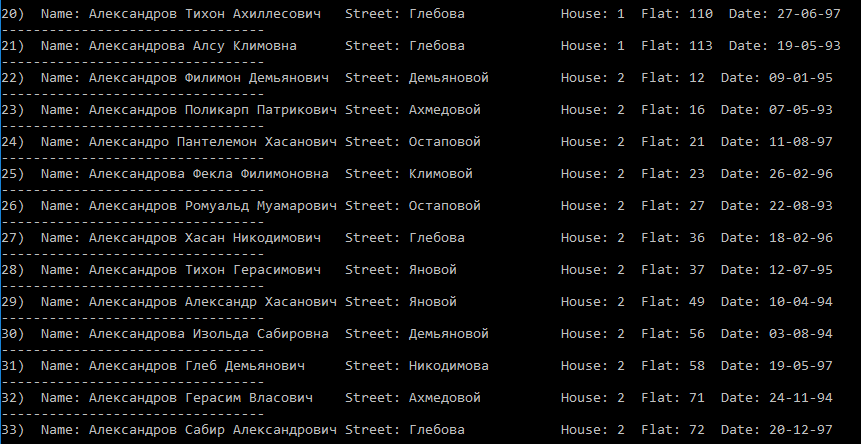


4. Очередь из записей, полученных в результате поиска (первые три буквы фамилии: Але).

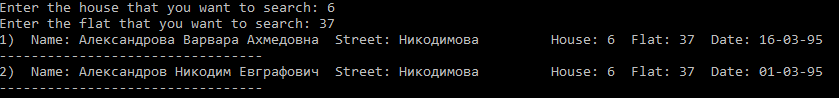




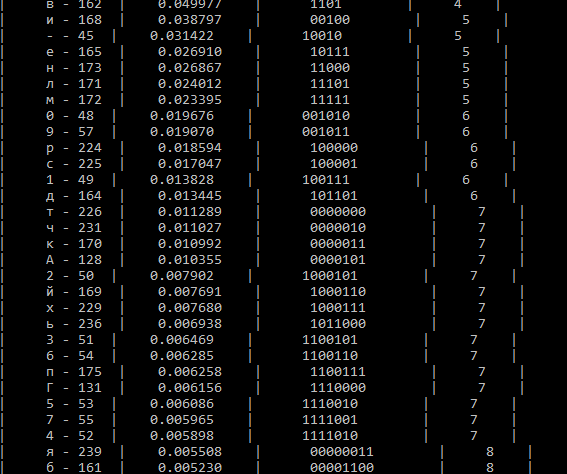
5. Обход дерева оптимального поиска А2 (первичный ключ для построения номер дома, а вторичный – номер квартиры).



6. Поиск по дереву (ключ – номер дома и квартиры). Если значения одинаковые, то в дереве они помещаются в средний указатель.



7. Кодирование (символ и его порядок по таблице ASCII, вероятность, кодовое слово и длина).



8. Энтропия файла базы данных, средняя длина.



# 7. ВЫВОДЫ

При завершении выполнения курсовой работы я выполнил следующие поставленные задачи: вывод базы данных, сортировка базы данных по двум ключам, бинарный поиск и занесение найденных записей в очередь, построение дерева (приближенный алгоритм дерева оптимального поиска А2), его обход слева направо и поиск по нему с двумя ключами, кодирование исходного файла базы данных (тем более теоретические значения были подтверждены на практике, т.е. код Хаффмена оказался действительно оптимальным и главное условие кодировки было сохранено Lср ≤ H+1).

Тем более, выполненные мной алгоритмы представляют собой минимальный набор для работы с базами данных, а также они обладают высоким быстродействием и эффективностью.