Министерство цифрового развития, связи и массовых телекоммуникаций Российской Федерации

Сибирский Государственный Университет Телекоммуникаций и Информатики

СибГУТИ

кафедра прикладной информатики и кибернетики

КУРСОВАЯ РАБОТА по дисциплине «Структуры и алгоритмы обработки данных»

Вариант 14

Выполнили: студент группы ИП-011

Игнатьев Д.А.

Проверил: доцент кафедры ПМиК

Янченко Е.В.

Новосибирск 2021

Оглавление

[Постановка задачи 3](#_Toc89865572)

[Основные идеи и характеристики применяемых методов 4](#_Toc89865573)

[Метод Хоара (QuickSort) 4](#_Toc89865574)

[Двоичный поиск 5](#_Toc89865575)

[Списки и очереди 5](#_Toc89865576)

[Дерево оптимального поиска (ДОП) и поиск в нем. 6](#_Toc89865577)

[Кодирование кодом Шеннона 6](#_Toc89865578)

[Описание структур данных и использованных алгоритмов 8](#_Toc89865579)

[Использованные структуры данных 8](#_Toc89865580)

[Особенности реализации алгоритмов 9](#_Toc89865581)

[Описание программы 9](#_Toc89865582)

[Основные переменные и структуры 9](#_Toc89865583)

[процедуры обработки базы данных 9](#_Toc89865584)

[Исходный код программы 11](#_Toc89865585)

[Результаты 40](#_Toc89865586)

[Выводы 42](#_Toc89865587)

# Постановка задачи

Хранящуюся в файле базу данных (4000 элементов) загрузить в оперативную память компьютера и построить индексный массив, упорядочивающий данные **по ФИО и названию улицы**, используя **метод Хоара** в качестве метода сортировки.

Предусмотреть возможность **поиска по ключу (в нашем случае – первые три буквы фамилии) в упорядоченной базе**, в результате которого из записей с одинаковым ключом формируется очередь, содержимое очереди выводится на экран.

Из записей очереди **построить дерево оптимального поиска (приближеный алгоритм А1)**, и предусмотреть возможность **поиска в дереве по запросу**.

Закодировать файл базы данных **статическим кодом Шенона**, предварительно оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран, упакованную базу данных записать в файл, вычислить энтропию и среднюю длину кодов.

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

Условие упорядоченности: по ФИО и названию улицы

Ключ поиска (К): пеpвые тpи буквы фамилии

# Основные идеи и характеристики применяемых методов

## Метод Хоара (QuickSort)

Метод Хоара или метод быстрой сортировки заключается в следующем. Возьмём произвольный элемент массива х. Просматривая массив слева, найдём элемент . Просматривая массив справа, найдём . Поменяем местами и . 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.

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

Элемент называется медианой для элементов , если количество элементов меньших am равно количеству элементов больших с точностью до одного элемента (если количество элементов нечётно). В примере буква К- медиана для КУРАПОВАЕ. В этом случае массив разделяется на 2 равные части.

Минимальная трудоемкость метода Хоара достигается в случае, когда на каждом шаге алгоритма в качестве ведущего элемента выбирается медиана массива. Количество сравнений в этом случае . 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

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 . Таким образом, трудоёмкость двоичного поиска в обоих случаях

## Списки и очереди

Списком называется последовательность однотипных элементов, связанных между собой указателями. Списки делятся на стеки (новый элемент добавляется в начало, удаляться может только первый элемент) и очереди (новый элемент добавляется в конец последовательности, удаляется первый элемент последовательности). В курсовой целесообразнее использовать очереди, поэтому рассмотрим их подробнее.

Операции с очередями:

Исключение первого элемента из очереди

Просмотр очереди

Добавление элемента в конец очереди

Добавление элемента по адресу p в очередь

Добавление элемента в пустую очередь

Иногда удобно рассматривать заголовок очереди как единое целое, образуя структуру. Это удобнее, когда используется много очередей.

## Дерево оптимального поиска (ДОП) и поиск в нем.

До сих пор предполагалось, что частота обращения ко всем вершинам дерева поиска одинакова. Однако встречаются ситуации, когда известна информация о вероятностях обращения к отдельным ключам. Обычно для таких ситуаций характерно постоянство ключей, т.е. в дерево не включаются новые вершины и не исключаются старые, и структура дерева остается неизменной. Эту ситуацию иллюстрирует сканер компилятора, который определяет, относится ли каждое слово программы (идентификатор) к классу ключевых слов. Статистические измерения на сотнях компилируемых программ могут дать информацию об относительных частотах появления в тексте программы конкретных ключевых слов.

Припишем каждой вершине дерева вес , пропорциональный частоте поиска этой вершины (например, если из каждых 100 операций поиска 15 операций приходятся на вершину , то =15). Сумма весов всех вершин дает вес дерева W. Каждая вершина расположена на высоте , корень расположен на уровне 1. Уровень вершины равен количеству операций сравнения, необходимых для поиска этой вершины.

Определим средневзвешенную высоту дерева с n вершинами следующим образом: – среднее время поиска по дереву с учетом частот обращения к его вершинам. Дерево, имеющее минимальную средневзвешенную высоту, называется деревом оптимального поиска (ДОП).

Для минимизации средней длины пути поиска нужно стремиться вершины с наибольшим весом располагать ближе к корню дерева.

ДОП обладает важным свойством. Все его поддеревья также оптимальны

Трудоемкость метода квадратична, благодаря закономерности в матрице Это позволяет сократить поиск до этого диапазона, что дает возможность уменьшить трудоемкость с кубической до квадратичной.

Но при больших объемах деревьев такие алгоритмы становятся неэффективными.

Известны быстрые алгоритмы, строящие почти оптимальные деревья поиска. Назовем эти Алгоритмы А1 и А2. В курсовой использовался алгоритм А1, поэтому распишем его подробнее.

Алгоритм А1:

В качестве корня берем вершину с наибольшим весом, будем поступать так для каждого поддерева.

Алгоритм А1 “плохой”: при дерево равносильно случайному (с точки зрения средневзвешенной высоты)

## Кодирование кодом Шеннона

Теорема (К. Шеннон). Для источника с алфавитом A = {} и вероятностями , и любого разделимого побуквенного кода средняя длина кодового слова всегда не меньше энтропии

И можно построить разделимый побуквенный код, у которого средняя длина кодового слова превосходит энтропию не больше, чем на единицу

Пример и алгоритм построения кода

1. Упорядочим символы исходного алфавита A = {} по убыванию их вероятностей:
2. Вычислим величины , которые называются кумулятивными вероятностями
3. Представим в двоичной системе счисления и возьмем в качестве кодового слова первые знаков после запятой.
4. Проверим, удовлетворяют ли неравенству Крафта?

Неравенство выполняется и, следовательно, префиксный код с таким набором длин кодовых слов существует.

1. Вычислим среднюю длину кодового слова:

и сравним ее с энтропией источника сообщений

Что полностью соответствует утверждению теоремы Шеннона

# Описание структур данных и использованных алгоритмов

## Использованные структуры данных

В курсовой использовались следующие структуры:

Структура, которая используется для записи 1 полной записи с базы данных

typedef struct Locale {

char FIO [32]; //Фамилия\_Имя\_Отчество

char street [18]; //Улица

short int N; //номер дома

short int flatN; //номер квартиры

char date [10]; //Дата заселения дд-мм-гг

} structure;

Структура дерева А1

struct VertexA1 {

Locale \*Key; // поле данных

int Mass; // вес элемента

VertexA1 \*Left = NULL; // указатель на левое поддерево

VertexA1 \*Right = NULL; // указатель на правое поддерево

VertexA1 \*Middle = NULL; // указатель на центральное поддерево

};

## Особенности реализации алгоритмов

1. Интерфейс программы:

Вывод базы оформлен в виде таблицы, что существенно упрощает ее просмотр. Для прорисовки рамки использовался стандартный оператор вывода printf(). Так же, во время вывода добавлены строчки - разделители, чтобы точно не ошибиться во время просмотра информации

1. Вторая версия бинарного поиска

Благодаря использованию второй версии поиска, мы значительно упрощаем поиск нужных элементов, так как после ее выполнения мы получаем готовый диапазон, который просто остается вывести

1. Структура кода

Во время реализации курсовой работы использовался собственно-написанный класс “очередь” для более удобного использования и сохранения данных в коде

1. Использование специальной функции повсеместно с QuickSort

Для упрощения работы с QuickSort, а именно для того, чтобы не приходилось делать несколько прогонов, была использована функция Sravnenie в которой сравниваются несколько структур по конкретно заданным параметрам структур.

1. Использование одного индексного массива для показа части базы данных и постройки по ней дерева, предварительно отсортировав базу.
2. Изменение структуры дерева, для поиска по нему. Для правильного поиска по нему необходимо было добавить указатель на новую ветвь Middle, в которую попадают одинаковые значения

# Описание программы

## Основные переменные и структуры

## процедуры обработки базы данных

void add\_mas(structure \*TH[]); - функция, которая заполняет динамический массив TH структурами, в которых находятся элементы базы данных.

void print\_mas(structure \*TH[], int n); - функция выводит массив TH, в количестве n элементов в соответствии с запросом пользователя

void del\_mas(structure \*TH[]); - функция, которая удаляет массив TH после завершения работы с ним.

Работа с сортировками

int Sravnenie (structure \*One, structure \*Two); // structure \*One и \*Two – два элемента массива TH, которые передаются в эту функцию для последующего сравнения их между собой по некоторым элементам структуры.

int Sravnenie(VertexA1 \*One, VertexA1 \*Two); // VertexA1 \*One и \*Two -два элемента массива bebey с вершинами и весами

во время работы с сортировками приходилось использовать несколько однотипных программ, но с разным наполнением. Так в первом Sravnenie мы сравниваем именно структуры, которые используются в заполнении массива TH. Второе Sravnenie нужно, для верной сортировки для последующего построения и вывода дерева A1

void Quick\_Sort (VertexA1 \*\*Array\_Vertex, int L, int R); // VertexA1 \*\*Array\_Vertex - тип передаваемого массива для сортировки. В данном случае – структуры VertexA1, L – левая граница, R - правая

void Quick\_Sort (structure \*\*A, int L, int R); тип передаваемого массива для сортировки. В данном случае – структуры structure, L – левая граница, R - правая

void Quick\_Sort (long int \*\*A, int L, int R); тип передаваемого массива для сортировки. В данном случае – массива указателей А, L – левая граница, R - правая

Аналогичная ситуация и с QuickSort. Так 1 – нужен для сортировки массива bebey дла последующего дерева А1. 2 – для сортировки массива TH для отсортированного вывода базы данных. 3- для сортировки массива указателей.

Функции поиска в базе

MyQueue\* Search(structure \*\*TH, int n, Locale \*\*A) // поиск в очереди по ключу structure \*\*TH – массив, в котором хранится база данных, int n – кол-во просматриваемых элементов в базе, Locale \*\*A – массив, в который записываются результаты поиска

int BSearch2(structure \*\*TH, char \*Search, MyQueue \*&Search\_Queue, int n, Locale \*\* A) //бинарный поиск structure \*\*TH – массив, в котором хранится база данных, int n – кол-во просматриваемых элементов в базе, Locale \*\*A – массив, в который записываются результаты поиска

void Delete\_All () // функция удаления очереди

void View\_All () // функция показа очереди

Функции работы с деревьями

void Root\_A1 (VertexA1 \*A, VertexA1 \*&Root); // Создание дерева А1, VertexA1 \*A – указатель на массив, в котором хранятся данные, VertexA1 \*&Root – указатель на корень дерева

void Root\_A1\_Obhod (VertexA1 \*Root, int mode); // обход А1 дерева VertexA1 \*Root – указатель на корень дерева, int mode – выбор, какой именно обход нужен (слева-направо, сверху-вниз, снизу-вверх)

# Исходный код программы

Код программы был разделен на несколько файлов, далее будет идти название файлов в кавычках и текст программы

“main.cpp” – в нем происходит сборка программы

#include <iostream>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

#include <locale.h>

#include <Windows.h>

#include <math.h>

#include "source.h"

#include "tree.h"

using namespace std;

class MyQueue{

private:

struct List {

Locale \*Data;

List \*Next = NULL;

};

List \*Obj = NULL;

int Count = 0;

static int Queue\_Count;

public:

//Добавляем элементы в конец, также как и в материнском классе

void Add (Locale \*data){ // Функция добавления в конец списка

List \*p = Obj;

if(Count == 0){

Obj = new List;

Obj->Data = data;

}

else{

while(p->Next != NULL){

p = p->Next;

}

List \*h = new List;

h->Data = data;

p->Next = h;

}

Count++;

}

Locale Delete\_Element (){ //удалить последний добавленный элемент

if(Count == 0){

cout << endl << "Ошибка!" << endl;

throw "Queue not created!";

}

else {

List \*h = Obj;

Locale \*D;

Obj = Obj->Next;

D = h->Data;

delete h;

Count--;

return \*D;

}

}

Locale Pop (){ // показать последний добавленный элемент

if(Count == 0){

cout << endl << "Ошибка!" << endl;

throw "Queue not created!";

}

else{

Locale \*D;

List \*h = Obj;

while(h->Next != NULL){

h = h->Next;

}

D = Obj->Data;

return \*D;

}

}

MyQueue (){

Count = 0;

Obj = NULL;

Queue\_Count++;

}

static int Get\_Queue\_Count(){

return Queue\_Count;

}

void View\_All (){

if(Count == 0){

cout << endl << "Error!!" << endl;

throw "Objeck not created!";

}

else {

List \*h = Obj;

int i = 1;

while (h->Next != NULL){

printf("%4d %33s %19s %2d %4d %11s\n", i, h->Data->FIO, h->Data->street, h->Data->N, h->Data->flatN, h->Data->date);

h = h->Next;

i++;

}

printf("%4d %33s %19s %2d %4d %11s\n", i, h->Data->FIO, h->Data->street, h->Data->N, h->Data->flatN, h->Data->date);

}

}

void Delete\_All (){ //Удалить всё

if(Count != 0){ // Если список пустой

List \*h;

while(Obj->Next != NULL){

h = Obj;

Obj = Obj->Next;

delete h;

}

delete Obj;

Count = 0;

}

}

int Get\_Count (){

return Count;

}

~MyQueue (){

Delete\_All();

}

};

int MyQueue::Queue\_Count = 0;

int BSearch2(structure \*\*TH, char \*Search, MyQueue \*&Search\_Queue, int n, Locale \*\* A)

{

int L,R,m, i = 0;

L=0;R=n-1;

while(L<R)

{

m=(L+R)/2;

if(strncmp(TH[m]->FIO, Search, 3) == -1){L=m+1;}

else if(strncmp(TH[m]->FIO, Search, sizeof(char)\*3) == 0){

break;

}

else {R=m;}

}

if(strncmp(TH[m]->FIO, Search, sizeof(char)\*3) == 0)

{

while(strncmp(TH[m]->FIO, Search, sizeof(char)\*3) == 0){

if(m == 0){

break;

}

m--;

}

if(m != 0){

m++;

}

while(strncmp(TH[m]->FIO, Search, sizeof(char)\*3) == 0){ // добавляем в очередь элементы удовлетворяющие словию поиска

Search\_Queue->Add(TH[m]);

A[i] = new Locale;

A[i] = TH[m];

m++;

i++;

if(m == 4000){break;}

}

}

else {return -1;}

return 1;

}

MyQueue\* Search(structure \*\*TH, int n, Locale \*\*A){ //функция для поиска по ключу

char \*String\_Seatch = new char [3];

cout << "Search(3): ";

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

String\_Seatch[i] = getch();

cout << String\_Seatch[i];

}

cout << endl;

system("pause");

MyQueue \*Search\_Queue = new MyQueue;

if(BSearch2(TH,String\_Seatch,Search\_Queue,n, A) == 1){

Search\_Queue->View\_All();

}

else {cout << endl << "Entry not found" << endl;}

return Search\_Queue;

}

int main (){

MyQueue \*Aboba = NULL;

setlocale(LC\_ALL, "ru");

Locale \*\*A = NULL;

A = new Locale \*[4000];

structure \*TH[4000]; //объявляю массив указателей

add\_mas(TH); // заполняю массив находится в source

while(1){

system("cls");

int Choice;//выбор

cout << "1: View database" << endl;

cout << "2: View sort database" << endl;

cout << "3: Database search" << endl;

cout << "4: Database coding" << endl;

cout << "-1: Exit" << endl << "Enter: ";

cin >> Choice;

if(Choice == -1){

break;

}

else if (Choice == 1){

while(1){

system("cls");

cout << "how much positions you want?"<<endl;

int n;

cin >> n;

if(n > 0){

print\_mas(TH, n); //просмотр бд

break;

}

else{

cout << "Error!" << endl;

system("pause");

}

}

}

else if(Choice == 2){

Quick\_Sort(TH,0,4000-1); // находится в sort

while(1){

system("cls");

cout << "how much positions you want?"<<endl;

int n;

cin >> n;

if(n > 0){

print\_mas(TH, n); //просмотр бд

break;

}

else{

cout << "Error!" << endl;

system("pause");

}

}

}

else if(Choice == 3){

Quick\_Sort(TH,0,4000-1);

system("cls");

Aboba = Search(TH,4000, A);

Aboba->View\_All();

system("Pause");

int m = Aboba->Get\_Count();

VertexA1 \*\*bebey = new VertexA1 \*[m]; //массив вершин с весами

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

bebey[i] = new VertexA1;

bebey[i]->Key = A[i];

bebey[i]->Mass = rand()%100;

}

Quick\_Sort(bebey,0,m-1); //Сортировка по весам вершин

VertexA1 \*A1\_Root = NULL;

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

Root\_A1(bebey[i],A1\_Root);

}

cout <<endl<< "Do a tree search? Y/N" <<endl << "Enter: ";

if(getch() == 'y' || getch() == 'Y'){

do{ //поиск по дереву

system("cls");

cout << endl << endl;

Root\_A1\_Obhod(A1\_Root,2);

int Key = -1;

VertexA1 \*Result = NULL;

cout << endl << "Write Key Search: ";

cin >> Key;

Result = Root\_A1\_Search(Key,A1\_Root);

if(Result != NULL){

int i = 1;

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

printf("| Num| FIO | street | NH|FlNun| date |\n");

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

while(Result != NULL && Result->Key->N == Key){

printf("|%4d|%33s| %19s| %2d| %4d| %11s|\n",i, Result->Key->FIO, Result->Key->street,Result->Key->N,Result->Key->flatN, Result->Key->date);

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

i++;

Result = Result->Middle;

}

cout << endl;

}

else{cout << "Vertex Not Foundet" << endl;}

system("Pause");

cout << endl << "Continue Search? Y/N : ";

}while (getch() == 'y' || getch() == 'Y');

}

Aboba->Delete\_All();

}

else if(Choice == 4){

system("cls");

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

fseek(base,0,SEEK\_SET);

unsigned char \*Symbol = new unsigned char;

long int Chance[256] = {};

long int \*\*Chance\_Symbol; //вероятности встречи каждого символа в БД

Chance\_Symbol = new long int \*[2];

Chance\_Symbol[0] = new long int [256];

Chance\_Symbol[1] = new long int [256];

float All\_Chance = 0; // сумма всех символов в фийле

/\*\* считаем символы\*\*/

fread(Symbol,sizeof( unsigned char),1,base);

while(!feof(base)){

if(((int)(\*Symbol)) < 256){

Chance[(int)(\*Symbol)]++;

}

fread(Symbol,sizeof(unsigned char),1,base);

}

fclose(base);

for(int i = 0; i < 256; i++){

Chance\_Symbol[0][i] = i;

Chance\_Symbol[1][i] = Chance[i];

All\_Chance += (float)Chance[i];

}

Quick\_Sort(Chance\_Symbol,0,255); //сортировка по убыванию

for(int i = 0; i < 256; i++){ //вывод отсортированного массива

cout << Chance\_Symbol[0][i] << ": " << Chance\_Symbol[1][i] << endl;

}

float \*\*Chance\_Symbol\_End = new float \*[2];

Chance\_Symbol\_End[0] = new float [256];

Chance\_Symbol\_End[1] = new float [256];

for(int i = 0; i < 256; i++){

Chance\_Symbol\_End[0][i] = Chance\_Symbol[0][i];

Chance\_Symbol\_End[1][i] = (float)Chance\_Symbol[1][i]/(float)All\_Chance;

//cout << Chance\_Symbol\_End[0][i] << ": " << Chance\_Symbol\_End[1][i] << endl; //вывод вероятностей

}

float \*Q = new float [256];

float \*L = new float [256];

float \*\*C = new float \*[256];

Q[0] = 0;

L[0] = ceil(-log2(Chance\_Symbol\_End[1][0]));

for (int i = 1; i < 256; i++ ){

Q[i] = Q[i-1] + Chance\_Symbol\_End[1][i-1];

if (Chance\_Symbol\_End[1][i] == 0){

L[i] = 0;

}

else{

L[i] = ceil(-log2(Chance\_Symbol\_End[1][i]));

}

}

for (int i = 0; i < 256; i ++){

C[i] = new float [(int)L[i]+1];

C[i][0] = L[i];

for (int j = 1; (float)j < L[i]+1; j++){

Q[i] = Q[i]\*2.f;

C[i][j] = floor(Q[i]);

if (Q[i] > 1.f){

Q[i] -=1.f;

}

}

}

for (int i = 0; i < 256; i++){

if(C[i][0] != 0){

cout << "C" << i <<"(" <<(char)Chance\_Symbol\_End[0][i] << ") = ";

for (int j = 1; j < L[i]+1; j++){

cout << C[i][j];

}

cout << endl;

}

}

float H = 0; //Энтропия

float L\_Sr = 0; //Средняя длина кодового слова

for(int i = 0; i < 256; i++){

if (Chance\_Symbol\_End[1][i] != 0){

H += -(Chance\_Symbol\_End[1][i]\*log2(Chance\_Symbol\_End[1][i]));

}

L\_Sr += Chance\_Symbol\_End[1][i]\*L[i];

}

if(L\_Sr < H+1){

cout <<endl << "H = "<< H << endl;

cout << "L sr = " << L\_Sr << endl;

}

else{ cout << "ERROR! Shannon's theorem does not hold!\n";}

delete[] Chance\_Symbol\_End[0];

delete[] Chance\_Symbol\_End[1];

delete[] Chance\_Symbol;

delete Symbol;

delete []L;

delete[]Q;

delete[]C;

system("pause");

}

else{cout << "Error!\nEnter an item from the suggested menu!!!"; system("pause");}

}

for (int i = 0; i < 400; i++){

delete TH [i];

}

delete TH;

system("pause");

return 0;

}

“tree.h” – здесь хранятся указатели на функции из файла “tree.cpp” и некоторые структуры

struct VertexA1 {

Locale \*Key;

int Mass;

VertexA1 \*Left = NULL;

VertexA1 \*Right = NULL;

VertexA1 \*Middle = NULL;

};

int Sravnenie(VertexA1 \*One, VertexA1 \*Two);

void Quick\_Sort (VertexA1 \*\*Array\_Vertex, int L, int R);

void Root\_A1 (VertexA1 \*A, VertexA1 \*&Root);

void Root\_A1\_Obhod (VertexA1 \*Root, int mode);

int Root\_A1\_Height (VertexA1 \*Root);

VertexA1 \*Root\_A1\_Search (int D, VertexA1 \*&Root);

“tree.cpp” -в нем хранятся основные функции с деревьями

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include "tree.h"

using namespace std;

VertexA1 \*Root\_A1\_Search (int D, VertexA1 \*&Root) //Рекурсивная процедура добавления в случайное древо поиска

{

VertexA1 \*&p = Root;

if( p == NULL ){

return NULL;

}

else if(D < p->Key-> N){

Root\_A1\_Search(D, p->Left);

}

else if(D > p->Key->N){

Root\_A1\_Search(D, p->Right);

}

else {return p;}

}

int Sravnenie(VertexA1 \*One, VertexA1 \*Two){

if( One->Mass > Two->Mass){

return 1;

}

else if(One->Mass < Two->Mass){

return 0;

}

else {return -1;}

}

void Quick\_Sort (VertexA1 \*\*A, int L, int R){

VertexA1 \*x = A[L];

int i = L, j = R;

while( i <= j ){

while( Sravnenie(A[i], x) == 1 ){ //По возрастанию 0

i++;

}

while( Sravnenie(A[j], x) == 0 ){ //По возрастанию 1

j--;

}

if( i <= j ){

VertexA1 \*t = A[i];

A[i] = A[j];

A[j] = t;

i++;

j--;

}

}

if( L < j){

Quick\_Sort(A, L, j);

}

if( i < R ){

Quick\_Sort(A, i, R);

}

}

void Root\_A1 (VertexA1 \*Vertex\_A, VertexA1 \*&Root){

int D = Vertex\_A->Key->N;

VertexA1 \*&p = Root;

if( p == NULL ){

p = Vertex\_A;

}

else if(D < p->Key->N){

Root\_A1(Vertex\_A, p->Left);

}

else if(D > p->Key->N){

Root\_A1(Vertex\_A, p->Right);

}

else {Root\_A1(Vertex\_A, p->Middle);}

}

int Root\_A1\_Height (VertexA1 \*Root){ //вычисление высоты дерева

int h = 0;

if(Root == NULL)

{

h = 0;

}

else {

h = 1 + max(Root\_A1\_Height(Root->Left),Root\_A1\_Height(Root->Right));

}

return h;

}

void Root\_A1\_Obhod (VertexA1 \*Root, int mode){

if(mode == 1){

if(Root != NULL)

{

cout << Root->Key->N << "(" << Root->Mass << ") ";

Root\_A1\_Obhod(Root->Left, mode);

Root\_A1\_Obhod(Root->Right, mode);

}

}

else if(mode == 2){

if(Root != NULL)

{

Root\_A1\_Obhod(Root->Left, mode);

cout << Root->Key->N << "(" << Root->Mass << ") ";

Root\_A1\_Obhod(Root->Middle, mode);

Root\_A1\_Obhod(Root->Right, mode);

}

}

else if(mode == 3){

if(Root != NULL)

{

Root\_A1\_Obhod(Root->Left, mode);

Root\_A1\_Obhod(Root->Right, mode);

cout << Root->Key->N << "(" << Root->Mass << ") ";

}

}

else if(mode == 4){

if(Root != NULL)

{

cout << Root->Mass << " ";

Root\_A1\_Obhod(Root->Left, mode);

Root\_A1\_Obhod(Root->Right, mode);

}

}

else if(mode == 5){

if(Root != NULL)

{

Root\_A1\_Obhod(Root->Left, mode);

cout << Root->Mass << " ";

Root\_A1\_Obhod(Root->Right, mode);

}

}

else if(mode == 6){

if(Root != NULL)

{

Root\_A1\_Obhod(Root->Left, mode);

Root\_A1\_Obhod(Root->Right, mode);

cout << Root->Mass << " ";

}

}

}

“source.h” -в нем хранятся ссылки на функции работы с БД и главная структура

#pragma once

typedef struct Locale {

char FIO [32]; //Фамилия\_Имя\_Отчество

char street [18]; //Улица

short int N; //номер дома

short int flatN; //номер квартиры

char date [10]; //Дата заселения дд-мм-гг

} structure;

void add\_mas(structure \*TH[]);

void print\_mas(structure \*TH[], int n);

void del\_mas(structure \*TH[]);

int Sravnenie (structure \*One, structure \*Two);

void Quick\_Sort (structure \*\*A, int L, int R);

void Quick\_Sort (long int \*\*A, int L, int R);

“source.cpp” – в нем хранятся основные функции для работы с БД

#include <iostream>

#include <stdlib.h>

#include <locale.h>

#include <Windows.h>

#include <conio.h>

#include <string>

#include "source.h"

using namespace std;

void add\_mas(structure \*TH[])

{

FILE \*base1 = fopen("testBase4.dat", "rb");

for(int i = 0; i < 4000; i++)

if(base1 != NULL){

TH[i] = new structure;

fread(TH[i]->FIO, sizeof(char), 32, base1);

fread(TH[i]->street, sizeof(char), 18, base1);

fread(&TH[i]->N, sizeof(short int), 1, base1);

fread(&TH[i]->flatN, sizeof(short int), 1, base1);

fread(TH[i]->date, sizeof(char), 10, base1);

}

else{

cout <<"Not founded";

}

fclose(base1);

}

void print\_mas(structure \*TH[], int n)

{

int k =0;

do

{

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

printf("| Num| FIO | street | NH|FlNun| date |\n");

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

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

{

printf("|%4d|%33s| %19s| %2d| %4d| %11s|\n",k+1, TH[k]->FIO, TH[k]->street,TH[k]->N,TH[k]->flatN,TH[k]->date);

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

k++;

}

cout << "continue? Y/N"<<endl;

int t = getch();

if (t =='n' || t =='N')

break;

} while(1);

}

void del\_mas(structure \*TH[])

{

for(int i =0; i <4000; i++)

{

delete TH[i];

}

}

void Quick\_Sort (long int \*\*A, int L, int R){ //Упорядочим по убыванию

long int x = A[1][L];

int i = L, j = R;

while( i <= j ){

while( A[1][i] > x ){

i++;

}

while( A[1][j] < x ){

j--;

}

if( i <= j ){

long int t = A[1][i];

long int Ind = A[0][i];

A[1][i] = A[1][j];

A[0][i] = A[0][j];

A[1][j] = t;

A[0][j] = Ind;

i++;

j--;

}

}

if( L < j){

Quick\_Sort(A, L, j);

}

if( i < R ){

Quick\_Sort(A, i, R);

}

}

“sort.spp” – в нем хранится сортировка структуры

#include <iostream>

#include <stdlib.h>

#include <locale.h>

#include <Windows.h>

#include <conio.h>

#include <string>

#include "source.h"

using namespace std;

int Sravnenie (structure \*One, structure \*Two){

if( strcmp(One->FIO, Two->FIO) > 0){

return 1;

}

else if(strcmp(One->FIO, Two->FIO) < 0){

return 0;

}

else {

if(One->street > Two->street){

return 1;

}

else if(One->street < Two->street){

return 0;

}

else{return -1;}

}

}

void Quick\_Sort (structure \*\*A, int L, int R){ //быстрая сортировка массива указателей

structure \*x = A[L];

int i = L, j = R;

while( i <= j ){

while( Sravnenie(A[i], x) == 0 ){

i++;

}

while( Sravnenie(A[j], x) == 1 ){

j--;

}

if( i <= j ){

structure \*t = A[i];

A[i] = A[j];

A[j] = t;

i++;

j--;

}

}

if( L < j){

Quick\_Sort(A, L, j);

}

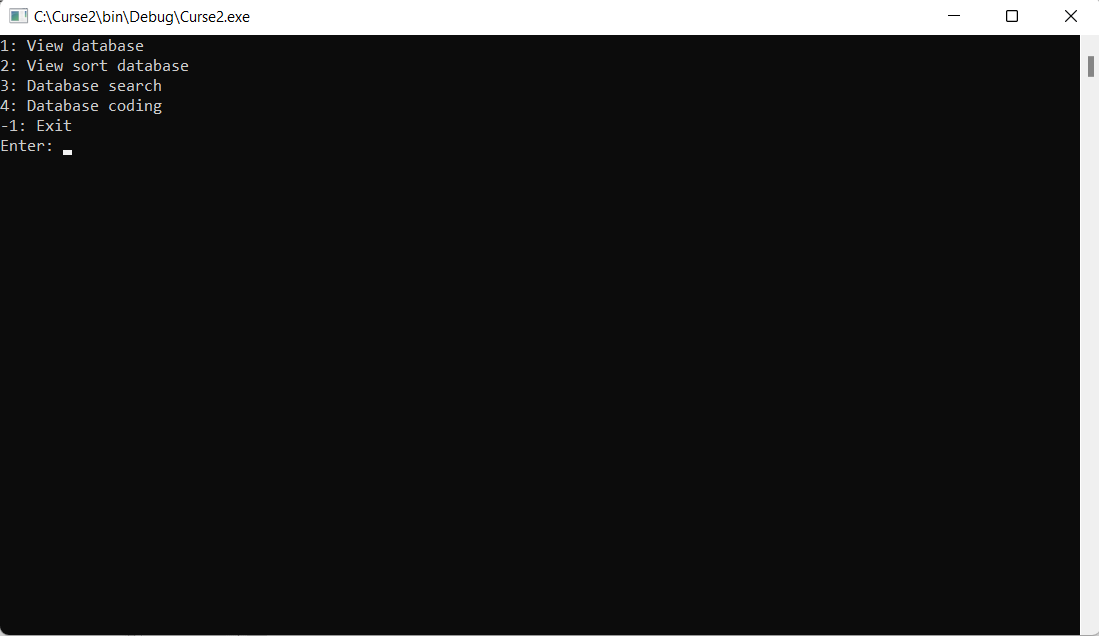
if( i < R ){

Quick\_Sort(A, i, R);

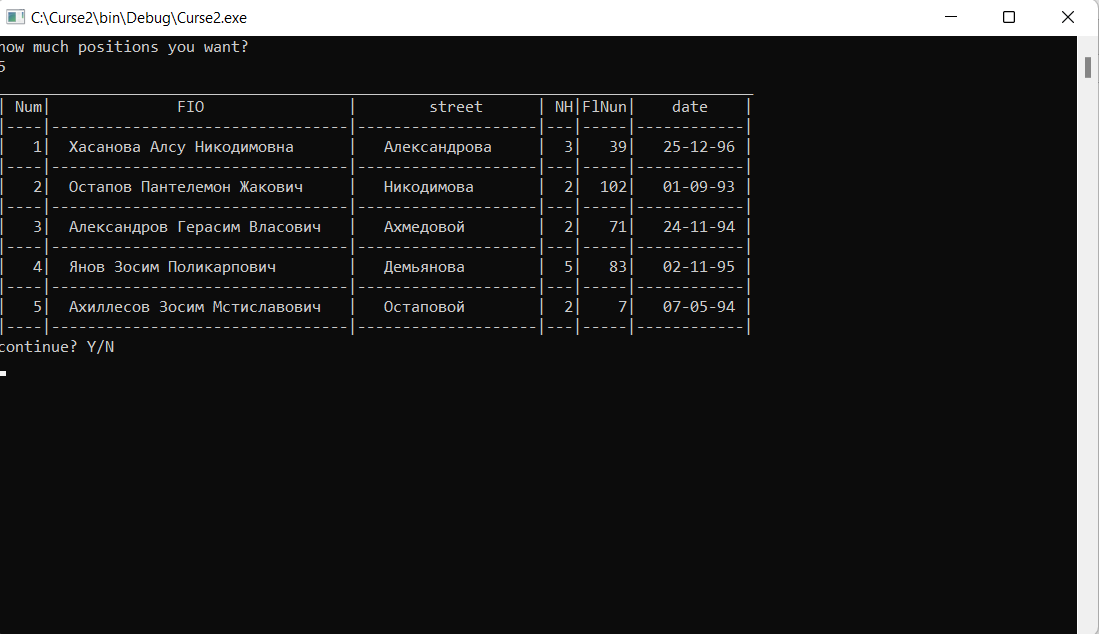
}

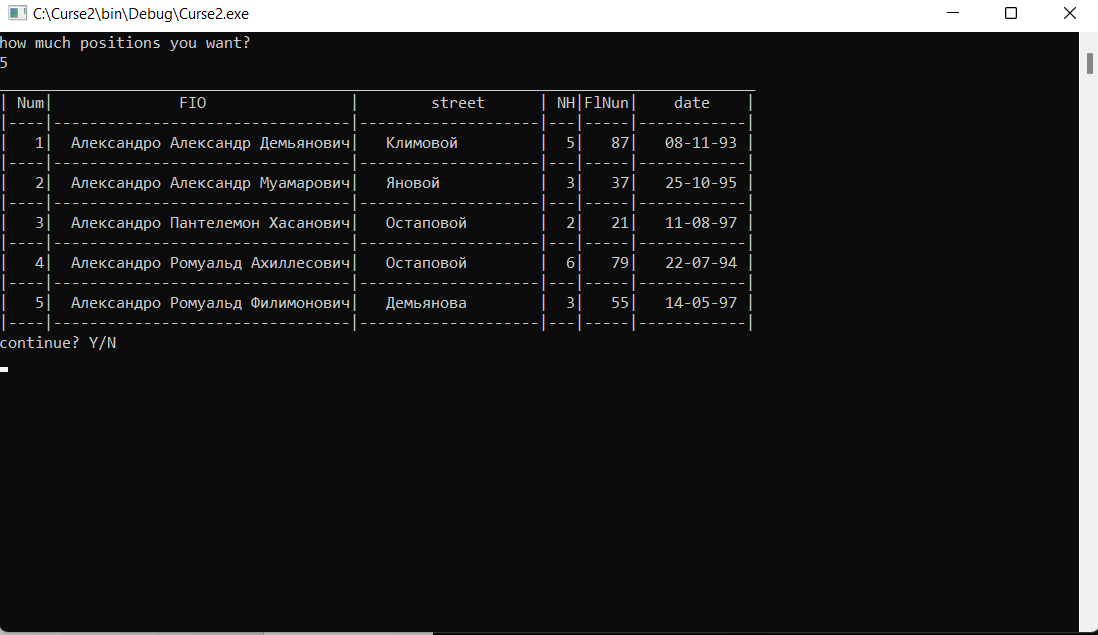
}

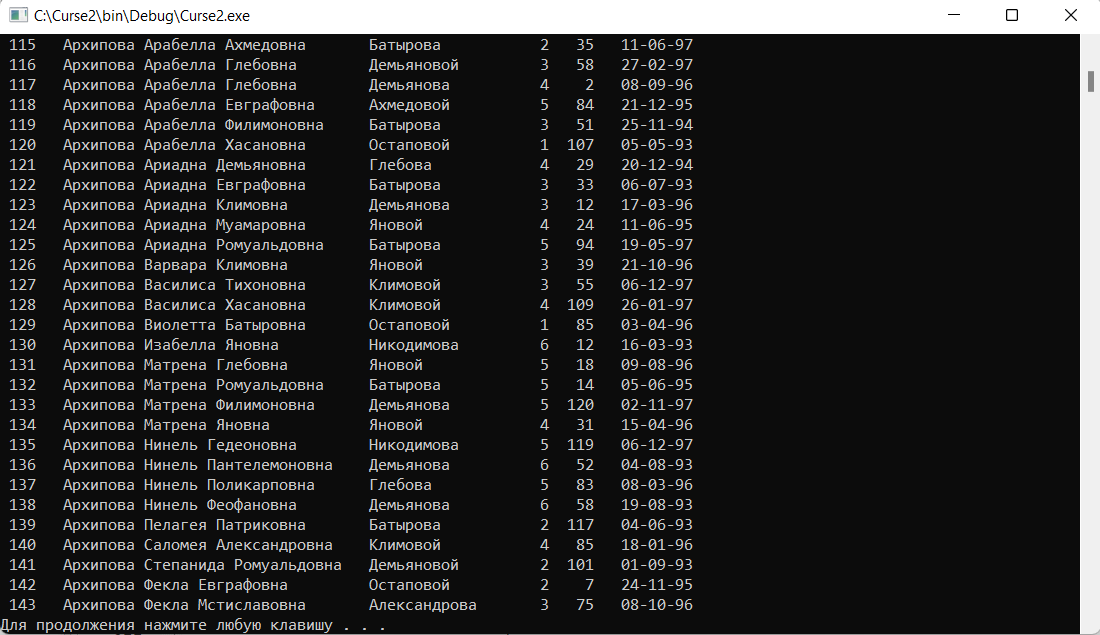
# Результаты

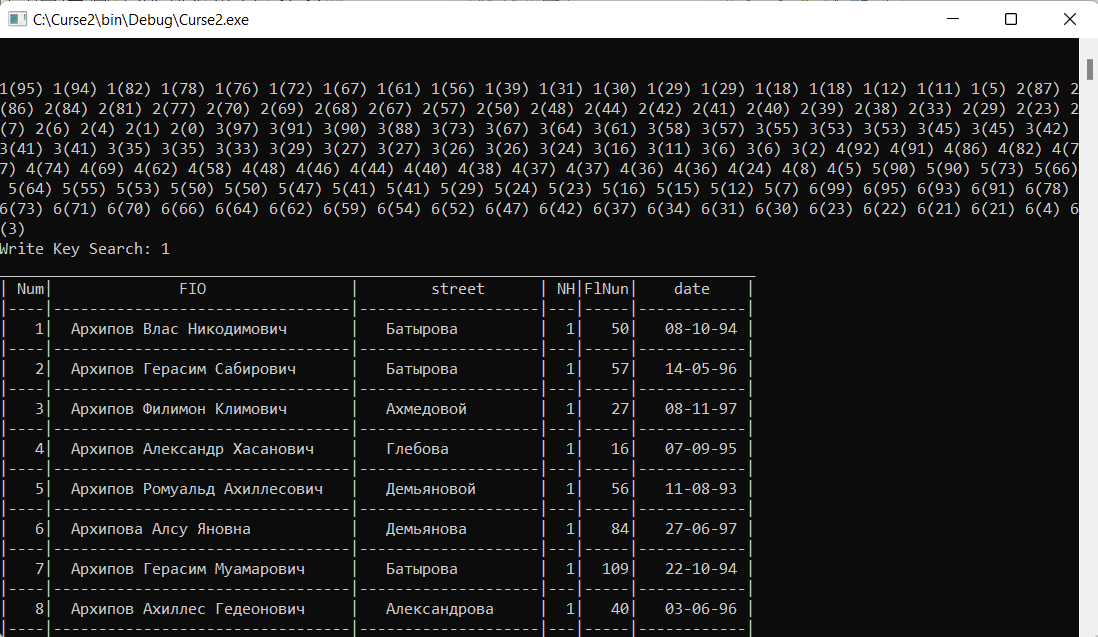


Начальный экран программы

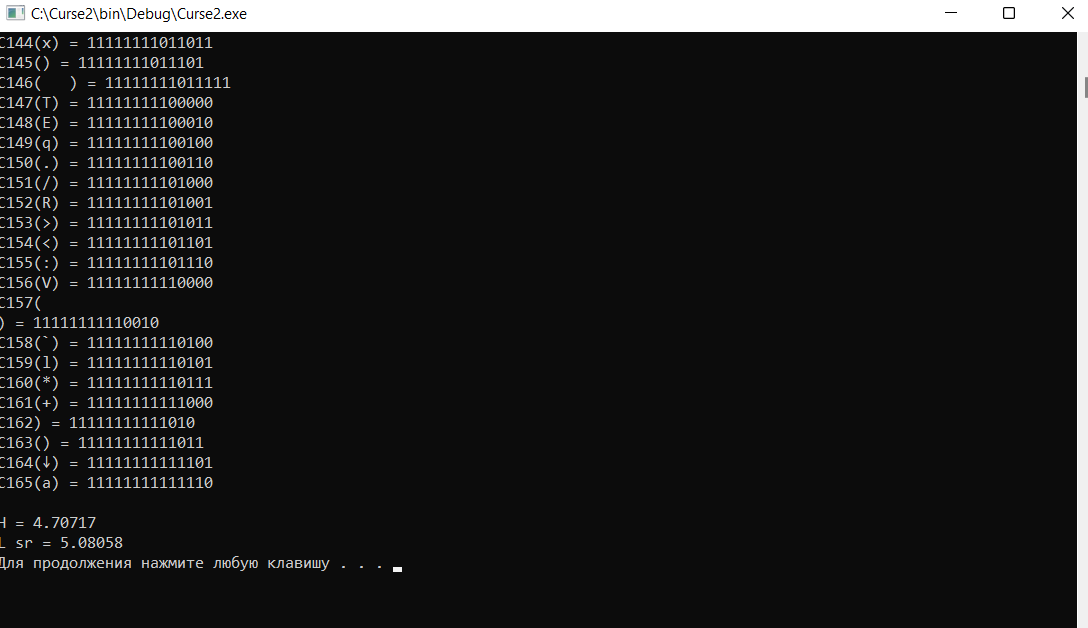
вывод неотсортированной базы

вывод отсортированной базы

вывод после поиска



А1 дерево поиска

вывод после кодирования

# Выводы

В ходе работы над курсовой я научился работе с базой данных, а именно: считывание, сортировка, вывод. Так же, я, благодаря полученным знаниям о кодировании смог закодировать базу данных, для возможной последующей работы. Не менее важным была работа с деревьями, которая составляла большую часть курсовой. Из-за разнообразных заданий, я стал лучше ориентироваться в деревьях, их структурах и назначениях.