Министерство цифрового развития, связи и массовых

Коммуникаций Российской Федерации

«Сибирский государственный университет телекоммуникаций и информатики»

Кафедра ПМ и К

КУРСОВОЙ ПРОЕКТ

По дисциплине «Структуры и алгоритмы обработки данных»

Вариант 49 (B=1, C=2, S=1, D=3, E=1)

Выполнил: студент группы ИП-012

Николаев Алексей Дмитриевич

Проверил: доцент кафедры ПМ и К

Янченко Е.В.

Новосибирск – 2021

**Оглавление**

[**Постановка задачи** 3](#_Toc90629567)

[**Основные идеи и характеристики применяемых методов** 3](#_Toc90629568)

[Метод сортировки 3](#_Toc90629569)

[Двоичный поиск 4](#_Toc90629570)

[Списки и очереди 4](#_Toc90629571)

[Вид дерева и поиск 4](#_Toc90629572)

[Метод кодирования 5](#_Toc90629573)

[**Описание структур данных и использованных алгоритмов** 5](#_Toc90629574)

[Использованные структуры данных 5](#_Toc90629575)

[Особенности реализации алгоритмов 6](#_Toc90629576)

[**Описание программ** 7](#_Toc90629577)

[Основные переменные и структуры 7](#_Toc90629578)

[Описание подпрограмм 8](#_Toc90629579)

[**Исходный текст программы** 10](#_Toc90629580)

[**Результаты** 24](#_Toc90629581)

[**Выводы** 29](#_Toc90629582)

# **Постановка задачи**

Хранящуюся в файле testBase1.dat базу данных (4000 записей), в соответствии с вариантом 49 (B=1), загрузить динамически в оперативную память компьютера в виде массива, сохранив указатели на каждую запись в индексном массиве указателей, и вывести на экран по 20 записей на странице с возможностью отказа от просмотра.

Упорядочить данные 1-ой базы данных с использованием индексного массива указателей по году издания и автору (C=2), используя пирамидальную сортировку Вилльямса-Флойда в соответствие с вариантом 49 (S=1). Упорядоченные данные вывести на экран.

Предусмотреть возможность быстрого поиска по ключу (K = год издания) в упорядоченной базе, в результате которого из записей с одинаковым ключом формируется очередь, содержимое очереди выводится на экран.

Из записей очереди построить дерево почти оптимального поиска по приближённому алгоритму A1 (D=3) в соответствии с вариантом поиска по ключу, отличному от ключа сортировки, вывести на экран содержимое дерева и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных оптимальным кодом Хаффмана (E=1), предварительно оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран, вычислить среднюю длину кодового слова и сравнить её с энтропией источника.

# **Основные идеи и характеристики применяемых методов**

## Метод сортировки

Метод Вилльямся-Флойда, также известный как пирамидальная сортировка, основан на алгоритме построения пирамиды и их свойств. Пирамиду называют последовательностью элементов, для которых выполняется неравенство

для всех , для которых хотя бы один из элементов существует.

Пирамида обладает 3-мя свойствами, которые широко используются для упорядочивания последовательности данных:

1. Двустороннее усечение (если последовательность является пирамидой, то она останется пирамидой без крайних элементов)
2. Если полная последовательность является пирамидой, то первый элемент последовательности – наименьший.
3. Вторая половина любой последовательности всегда является пирамидой.

Для реализации алгоритма мы достраиваем последовательность до пирамиды, добавляя слева в пирамиду 2-ой части последовательности по элементу. После этого подставляем первый (минимальный) элемент в конец последовательности, а первый подставляем в пирамиду. Мы повторяем процедура пока пирамида не исчезнет, в результате чего мы получим обратно упорядоченную последовательность.

Данный метод сортировки имеет трудоёмкость , так что его целесообразно использовать для сортировки больших объёмов данных.

## Двоичный поиск

Основная идея двоичного поиска заключается в том, чтобы в упорядоченном по возрастанию массиве брать его средний элемент и сравнивать с ключом поиска. Если элемент окажется больше, то искомый элемент должен быть в левой половине массиве, а если меньше – то в правой. Таким образом уменьшаем область для поиска элемента и продолжаем повторять эту процедуру пока элемент не будет найден или весь массив не будет исследован.

Данный метод поиска элемента в массиве имеет трудоёмкость , так поиск элементов происходит очень быстро даже для очень больших последовательностей

## Списки и очереди

Списком называется последовательность однотипных элементов, связанных между собой указателями, которые хранятся в динамической памяти программы. Основное их предназначенное заключается в добавлением элементов, количество которых заранее неизвестно (так же списки используются для сортировки данных с помощью метода прямого слияния и цифровой сортировки, но речь сейчас о них не идёт).

Очередью называется список, каждый новый элемент которого добавляется в конец последовательности и удаляется всегда первый элемент последовательности. Очередь реализует дисциплину обслуживания FIFO (First Input, First Output – Первый вошёл, первым ушёл). Это очень удобно, когда создаёшь список из элементов упорядоченного массива, так как при просмотре списка элементы отображаются в том порядке, в котором они были в него добавлены.

## Вид дерева и поиск

Вид дерева, которое будет использовано для решения поставленной задачи, будет двоичное дерево оптимального поиска. Особенностью двоичных деревьев является то, что каждая вершина дерева соединена с возможно пустыми левым и правым двоичными поддеревьями. Двоичное дерево называется двоичным деревом поиска, если ключ поиска в каждой его вершине больше ключей всех вершин в левом поддереве и меньше ключей всех вершин в правом поддереве.

Идея построения дерева оптимального поиска, заключается в том, что часто нужно строить дерево не просто, чтобы эффективно реализовать в нём поиск и хранение информации, но и чтобы самые часто встречаемые данные находились как можно ближе к корню дерева. Так как точный алгоритм ДОП по трудоёмкости и памяти слишком много затратный, использую обычно приближённые алгоритмы А1 и А2.

Приближенный алгоритм А1 заключается в том, чтобы обратно упорядочить последовательность, из которой будет формироваться дерево, по их весу (показателю встречаемости данных) и добавлять элементы в дерево по пути, как если бы мы искали этот элемент в дереве. В результате поиск не будет нарушен, а самые часто встречаемые элементы будут близко корню.

Используя низко трудоёмкую сортировку, можно получить приближённый алгоритм А1 трудоёмкостью .

## Метод кодирования

Метод кодирования, который будет использован для кодирования базы данных с целью дальнейшего его сжатия, будет оптимальный код Хаффмана. Он признан самым эффективным статическим методом кодирования исходного алфавита, так как средняя длина кодовых слов, порождаемая этим методом, наиболее близко подошла к среднему значению количества информации, необходимого для кодирования сообщений (энтропии источника).

Идея метода Хаффмана заключается в сумме наименьших вероятностей букв алфавита источника, с целью получения последовательности двух вероятностей. Затем при обратном процессе (одна вероятность разбивается на две предыдущих) одному из элементов присваивается ноль в конце кодового слова, а другому – единица. В результате мы получаем кодовые слова для каждого символа алфавита, где ни одно кодовое слово не является началом другого (префиксный код), а значит его можно однозначно декодировать.

# **Описание структур данных и использованных алгоритмов**

## Использованные структуры данных

Для реализации курсового проекта были использованы только 5 структур:

1. Struct record;
2. Struct List;
3. Struct Element;
4. Struct Vertex;
5. Class Byte\_Chance;

Структура типа record хранит в себе данные о записях, которые были занесены в файл базы данных, включая статический массивы author[12], title[32], publisher[16] типа char и переменные year и num\_of\_page типа short int. Она используется для хранения записей из базы данных в оперативной памяти компьютера, чтобы каждый раз не считывать из файла всю базу данных.

Структура типа List хранит в себе указатель Data на одну из записей типа record и указатель Next на следующий элемент типа List. Она используется для создания очереди из найденных по ключу поиска элементов.

Структура типа Element хранит в себе указатель Data на одну из записей типа record и переменную W типа int, которая показывает какой случайный вес у данной записи. Она используется для более удобного хранения информации о весе определённой записи во время сортировки для построения ДОП.

Структура типа Vertex хранит в себе указатель на элемент типа Element, указатели Left и Right типа Vertex, которые указывают на левые и правые поддеревья данной вершины, и указатель Down типа Vertex, который создаёт список из вершин дерева с одинаковым ключом. Она широко используется для построения ДОП, храня информацию о каждой вершине: её весе, записи из базы данных и указателей на поддеревья.

Класс Byte\_Chance хранит в себе переменную Chance типа double, которая показывает какой шанс встречи определённого байта в программе, и value типа int, которая показывает целочисленное значение данного байта. Он нужен для эффективного хранения информации о шансе определённого байта во время сортировки. (В C# переменные класса имеют ссылочный тип, а структуры – физический, так что эффективные во время сортировки использовать класс)

## Особенности реализации алгоритмов

Пирамидальная сортировка (Метод Вилльямса-Флойда):

Для того, чтобы пирамидальная сортировка прямо, а не обратно упорядочивала входной массив, необходимо изменить принцип построения пирамиды. Вместо условия, что каждый элемент пирамиды меньше минимального из двух элементов, которые стоят в два раза дальше от начала массива, необходимо чтобы каждый элемент был больше максимального. Таким образом, в начале всей последовательности, которая является пирамидой, будет стоят наибольший элемент, и он будет отправляться в конец последовательности, а не наименьший.

Двоичный поиск:

Для реализации курсового проекта лучше всего использовать 2-ой вариант реализации алгоритма двоичного поиска, так как из элементов, совпадающих по ключу, алгоритм возвращает самый левый из них, благодаря чему найти все равные элементы начиная с первого из них очень легко.

Дерево оптимального поиска:

Для реализации курсового проекта необходимо произвольным образом распределить весы для каждой найденной по ключу записи, так как заранее неизвестно, как часто встречается та или иная запись. Также для того, чтобы записи, совпадающие по ключи, не пропадали и находились в дереве, они будут добавляться в конец очереди из вершины, которая одинакова по ключу.

Оптимальный код Хаффмана:

Так как длина кодовых слов для алфавита источника увеличивается по мере выполнения кодировки, то невозможно использовать статические или динамические массивы символов, так что необходимо использовать строковые переменные типа string (С ними мне было комфортнее всего работать на языке C#, именно поэтому последняя часть курсового проекта написана не на C++).

# **Описание программ**

## Основные переменные и структуры

Рассмотрим сначала код программы, который содержит в себе первые 5 частей курсового проекта и написан на языке C++.

В функции main() мы создаём указатель file на элемент типа FILE для открытия файла с целью дальнейшего чтения и загрузки хранящихся в нём записей в оперативную память компьютера. Также мы создаём константную целочисленную переменную N=4000, которая хранит в себе информацию о количестве записей в базе данных.

Полученную последовательность записей мы сохраняем в динамическом массиве Record, после чего создаём динамический индексный массив указателей, каждый элемент которого ссылается на соответствующий элемент массива записей.

В этой же функции мы создаём указатели на главнейшие элементы очереди, которая будет из найденный элементов по ключу поиска: голову списка Head\_List и хвост списка Tail\_List. Также мы инициализируем указатель на вершину Vertex будущего дерева оптимального поиска.

Дополнительно задаю переменную Size, которая будет равна длине полученной очереди из элементов поиска, чтобы из очереди элементов типа List создать динамический массив элементов типа Element.

В качестве буфера для загрузки записи из файла в массив используется глобальная переменная rec типа record.

Теперь рассмотрим код программы, который написан на языке C#, для кодирования базы данных кодом Хаффмана.

Из основных переменных можно выделить вещественную переменную sum, которая будет подсчитывать кумулятивную вероятностей встречаемости всех байтов и на их основе подстраивать их так, чтобы переменная была равна 1 (самой большой вероятности присвоить значение разности межу 1 и кумулятивной вероятности всех остальных байтов).

Также в функции main были созданы массивы из 257 элементов Bytes типа Byte\_Chance и Byte\_Repeat типа int, который хранит в себе каждое повторение определённых байтов и переменная Num\_of\_Bytes, которая подсчитывает общее количество байт в файле.

После чтения базы данных и сортировки байтов по убыванию их вероятностей, создаются динамические массивы из количества встреченных байтов (Exist) ‘+1’ L типа int для хранения длин кодовых слов, C типа string для хранения самих кодовых слов и P типа double для рабочего копирования вероятностей всех байтов. (В коде Хаффмана мы ведь суммируем вероятности и проще сделать резервную копию, с которой можно будет легко работать)

## Описание подпрограмм

Рассмотрим сначала код программы, который содержит в себе первые 5 частей курсового проекта и написан на языке C++.

Функция Copy\_Address() предназначена для по элементного присваивания указателям динамического индексного массива адресов элементов массива, состоящий из 4000 записей базы данных. В качестве параметров ей передаются указатель на индексный массив, указатель на массив записей и количество записей.

Функция more() отвечает за сравнение двух записей по условию упорядочивания (год издания и автор), которая возвращает единицу если первый элемент больше второго и ноль в противном случае. В качестве параметров ей передаются указатели на 2 записи типа record.

Функция more2() отвечает за сравнение двух записей по другому условию упорядочивания (количество страниц), которая возвращает единицу если первый элемент больше второго и ноль в противном случае. В качестве параметров ей передаются указатели на 2 записи типа record.

Функция more\_equal() отвечает за сравнение двух записей по условию упорядочивания (год издания и автор), которая возвращает единицу если первый элемент больше второго или равен ему и ноль в противном случае. В качестве параметров ей передаются указатели на 2 записи типа record.

Функция Heap() отвечает за создание пирамиды, подстановкой самого левого элемента в необходимую позицию уже созданной пирамиды. В качестве параметров ей передаются значения левой и правой границы индексного массива и сам индексный массив типа record. В перегруженной версии этой функции - индексный массив типа Element.

Функция Heap\_Sort() отвечает за пирамидальную сортировку, алгоритм которого было описан чуть ранее. В качестве параметров ей передаются индексный массив указателей типа Record и количество элементов в массиве. В перегруженной версии этой функции - индексный массив типа Element.

Функция ReadDataBase() отвечает за чтение базы данных и сохранение записей в динамической массиве. В качестве параметров передаются указатель на массив, количество записей и указатель на файл.

Функция WriteDataBase() отвечает за отображение в консоли полученного массива записей по 20 элементов или всего сразу. В качестве параметров передаются указатель на массив записей и количество записей.

Функция WriteSortDataBase() отвечает за отображение в консоли отсортированного индексного массива записей по 20 элементов или всего сразу. В качестве параметров передаются указатель на индексный массив указателей типа record и количество записей.

Функция BSearchDataBase() отвечает за двоичный поиск элемента в отсортированном индексном массиве элемента по ключу. В случаи нахождения возвращает индекс найденного элемента, в противном случае – ноль. В качестве параметров ей передаются индексный массив указателей на записи, количество записей и ключ поиска (год издания).

Функция BSearchAllDataBase() отвечает за создание очереди из всех элементов индексного массива указателей на записи, совпадающих по ключу поиска. В качестве параметров ей передаются индексный массив, количество записей, и указатели на голову и хвост будущей очереди, переданные по ссылке.

Функция PrintList() отвечает за отображение в консоли полученной очереди и возврат её длины. В качестве параметра ей передаётся указатель на голову очереди.

Функция Copy\_Array() отвечает за заполнение индексного массива типа Element элементами полученной очереди и присваивание весам записей определённых значений. В качестве параметров ей передаются указатель на голову списка и динамический массив типа Element.

Функция Add\_Vertex() отвечает за рекурсивное добавление вершины в дерево оптимального поиска по количеству страниц. В качестве параметров ей передаются добавляемый элемент типа Element и указатель на вершину, переданный по ссылке.

Функция A1\_Tree() отвечает за построение дерева оптимального поиска по приближенного алгоритму А1, который был описан чуть ранее. Функция возвращает в качестве результата работы указатель на корень дерева. В качестве параметров ей передаются указатель на голову списка и длина этого списка.

Функция Obhod() отвечает за рекурсивное отображение в консоли дерева поиска с помощью обхода слева-вниз направо по 20 записей или всех сразу. В качестве параметров ей передаются указатель на вершину и переменные отвечающие количество отображённых записей и вид отображения, переданные по ссылке.

Функция TreeSearch() отвечает за поиск первой вершины дерева, совпадающей по ключу, и в случае удачного нахождения возвращает указатель на вершину для дальнейшего отображения всех вершин по ключу. В качестве параметров ей передаётся указатель на корень дерева.

Функция Menu() отвечает за графическое отображение в консоли возможных вариантов работы с базой данных и возвращает числовое значение выбранного действия. В качестве параметров ей передаётся строка (массив символов) для отображения названия базы данных.

Теперь рассмотрим код программы, который написан на языке C#, для кодирования базы данных кодом Хаффмана.

В качестве параметров для данной версии функций Heap() и HeapSort() передаётся массив объектов класса Byte\_Chance.

Функция Up() отвечает за прямой ход рекурсии кода Хаффмана, а именно за сдвиг суммы последних двух вероятностей в массиве для восстановления обратной упорядоченности массива. Функция возвращает индекс элемента массива, в котором оказалась сумма. В качестве параметров ей передаются массив вероятностей, вещественное значение суммы и количество элементов в массиве.

Функция Down() отвечает за обратный ход рекурсии кода Хаффмана, а именно за увеличение кодовых слов для элементов суммы вероятностей. В качестве параметров ей передаются массив кодовых слов, массив их длин, индекс суммы в массиве и количество элементов в нём.

Функция Huffman() отвечает за построение кода Хаффмана, который был описан ранее. В качестве параметров ей передаются массивы вероятностей, кодовых слов, их длин и количество элементов массиве.

Функция N-Tropy() отвечает за вычисление энтропии источника по формуле

Где это вероятность встречаемости байта. В качестве параметров ей передаётся массив объектов класса Byte\_Chance.

Функция Average\_Len() отвечает за вычисления средней длины кодовых слов для сравнения с энтропией по формуле

Где это вероятность встречаемости байта, а длина кодового слова. В качестве параметров ей передаются массив объектов класса Byte\_Chance и массив длин кодовых слов.

# **Исходный текст программы**

1. Файл Kurs2-AlexNB.cpp с первыми 5 частями курсового проекта:

#include<stdio.h>

#include<stdlib.h>

struct record

{

char author[12];

char title[32];

char publisher[16];

short int year;

short int num\_of\_page;

}rec;

struct List

{

record\* Data;

List\* Next;

};

struct Element

{

record\* Data;

int W;

};

struct Vertex

{

Element\* Info;

Vertex\* Left;

Vertex\* Right;

Vertex\* Down;

};

**void Copy\_Address(struct record Records[], struct record\* Index\_Records[],int N)**

{

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

{

Index\_Records[i]=&Records[i];

}

}

**int more(struct record \*X, struct record \*Y)**

{

if(X->year>Y->year) return 1;

if(X->year<Y->year) return 0;

for(int i=0;i<12;i++)

{

if(X->author[i]>Y->author[i]) return 1;

if(X->author[i]<Y->author[i]) return 0;

}

return 0;

}

**int more2(struct record \*X, struct record \*Y)**

{

if(X->num\_of\_page>Y->num\_of\_page) return 1;

return 0;

}

**int more\_equal(struct record \*X, struct record \*Y)**

{

if(X->year>Y->year) return 1;

if(X->year<Y->year) return 0;

for(int i=0;i<12;i++)

{

if(X->author[i]>Y->author[i]) return 1;

if(X->author[i]<Y->author[i]) return 0;

}

return 1;

}

**void Heap(struct record\* Z[], int L, int R)**

{

struct record\* X=Z[L];

int i=L, j=0;

while(1)

{

j=2\*i;

if(j>R)

{

break;

}

if(j<R)

{

if(more\_equal(Z[j+1],Z[j])) j++;

}

if(more(X,Z[j]))

{

break;

}

Z[i]=Z[j];

i=j;

}

Z[i]=X;

}

**void HeapSort(struct record\* Z[], int N)**

{

int L=N/2, R=N;

struct record\* t;

while(L>0)

{

Heap(Z,L,R);

L--;

}

while(R>1)

{

t=Z[1];

Z[1]=Z[R];

Z[R]=t;

R--;

Heap(Z,1,R);

}

}

**void ReadDataBase(struct record Records[], int N, FILE \*file)**

{

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

{

fread(&rec,sizeof(rec),1,file);

Records[i]=rec;

}

}

**void WriteDataBase(struct record Records[], int N)**

{

int i=1, switcher=1;

struct record\* show;

printf("Жизнь замечательных людей!\n\n");

system("chcp 866>nul");

while(1)

{

for(int k=0;k<20;k++)

{

show=&Records[i];

printf("%-6i %s %s %s %i %i\n",i,show->author,show->title, show->publisher, show->year, show->num\_of\_page);

i++;

}

if(i<=N && switcher==1)

{

system("chcp 1251>nul");

printf("\nВы хотите продолжить?\n");

printf("0 -- Нет\n");

printf("1 -- Да, ещё 20 записей\n");

printf("2 -- Да, все оставшиеся записи\n");

scanf("%i",&switcher);

printf("\n");

system("chcp 866>nul");

if(switcher==0)

{

break;

}

}

if(i>4000) break;

}

printf("\n");

system("pause");

}

**void SortDataBase(struct record\* Index\_Records[], int N)**

{

HeapSort(Index\_Records,N);

}

**void WriteSortDataBase(struct record\* Index\_Records[],int N)**

{

int i=1, switcher=1;

struct record\* show;

printf("Жизнь замечательных людей!\n\n");

system("chcp 866>nul");

while(1)

{

for(int k=0;k<20;k++)

{

show=Index\_Records[i];

printf("%-6i %s %s %s %i %i\n",i,show->author,show->title, show->publisher, show->year, show->num\_of\_page);

i++;

}

if(i<=N && switcher==1)

{

system("chcp 1251>nul");

printf("\nВы хотите продолжить?\n");

printf("0 -- Нет\n");

printf("1 -- Да, ещё 20 записей\n");

printf("2 -- Да, все оставшиеся записи\n");

scanf("%i",&switcher);

printf("\n");

system("chcp 866>nul");

if(switcher==0)

{

break;

}

}

if(i>N) break;

}

printf("\n");

system("pause");

}

**int BSearchDataBase(struct record\* Index\_Records[], int N, int year)**

{

int L=1, R=N, m;

while(L<R)

{

m=(L+R)/2;

if(Index\_Records[m]->year<year)

{

L=m+1;

}

else

{

R=m;

}

}

if(Index\_Records[R]->year==year)

{

return R;

}

return 0;

}

**void BSearchAllDataBase(struct record\* Index\_Records[], int N, List\* &Head, List\* &Tail)**

{

int year, switcher=1;

printf("Пожалуйста, введите год, по которому необходимо провести поиск данных\n");

system("chcp 866>nul");

scanf("%i",&year);

printf("\n");

int j=BSearchDataBase(Index\_Records, N, year);

if(j==0)

{

return;

}

struct List\* p;

p = new List;

p->Data=Index\_Records[j];

Head=p;

Tail=p;

j++;

for(; j<=N && Index\_Records[j]->year==year;j++)

{

p = new List;

p->Data=Index\_Records[j];

Tail->Next=p;

Tail=p;

}

system("chcp 1251>nul");

printf("Жизнь замечательных людей!\n\n");

}

**int PrintList(List\* Head)**

{

List\* p=Head;

system("chcp 1251>nul");

if(p==NULL)

{

printf("Простите, но элементов с вышеуказанным годом не существует в базе данных\n");

system("pause");

return 0;

}

system("chcp 866>nul");

int i=1, switcher=1;

while(p!=NULL)

{

printf("%-6i %s %s %s %i %i\n",i,p->Data->author,p->Data->title, p->Data->publisher, p->Data->year, p->Data->num\_of\_page);

if((i%20==0)&&switcher==1)

{

system("chcp 1251>nul");

printf("\nВы хотите продолжить?\n");

printf("0 -- Нет\n");

printf("1 -- Да, ещё 20 записей\n");

printf("2 -- Да, все оставшиеся записи\n");

scanf("%i",&switcher);

printf("\n");

system("chcp 866>nul");

if(switcher==0)

{

break;

}

}

p=p->Next;

i++;

}

system("pause");

return i-1;

}

**void Copy\_Array(struct List\* &Head,struct Element\* Array\_Elements[])**

{

List\* p=Head;

int i=1;

while(p!=NULL)

{

Array\_Elements[i] = new Element();

Array\_Elements[i]->Data=p->Data;

Array\_Elements[i]->W=rand()%20+1;

p=p->Next;

i++;

}

}

**void Heap(struct Element\* Z[], int L, int R)**

{

struct Element\* X=Z[L];

int i=L, j=0;

while(1)

{

j=2\*i;

if(j>R)

{

break;

}

if(j<R)

{

if(Z[j+1]->W>=Z[j]->W) j++;

}

if(X->W>Z[j]->W)

{

break;

}

Z[i]=Z[j];

i=j;

}

Z[i]=X;

}

**void HeapSort(struct Element\* Z[], int N)**

{

int L=N/2, R=N;

struct Element\* t;

while(L>0)

{

Heap(Z,L,R);

L--;

}

while(R>1)

{

t=Z[1];

Z[1]=Z[R];

Z[R]=t;

R--;

Heap(Z,1,R);

}

}

**void Add\_Vertex(struct Vertex\* &v, struct Element\* X)**

{

if(v==NULL)

{

v=new Vertex();

v->Info=X;

v->Left=NULL;

v->Right=NULL;

v->Down=NULL;

}

else if(more2(X->Data,v->Info->Data)) Add\_Vertex(v->Right,X);

else if(more2(v->Info->Data,X->Data)) Add\_Vertex(v->Left,X);

else Add\_Vertex(v->Down,X);

}

**struct Vertex\* A1\_Tree(struct List\* Head, int N)**

{

if(N==0)

{

printf("Простите, но вы ещё не совершили поиск в пункте 3)\n");

return NULL;

}

struct Element\*\* Array\_Elements = new Element\*[N+1];

Copy\_Array(Head,Array\_Elements);

struct Vertex\* Root=NULL;

HeapSort(Array\_Elements,N);

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

{

Add\_Vertex(Root,Array\_Elements[i]);

}

return Root;

}

**void Obhod(Vertex \*v, int &L, int &switcher)**

{

if(v!=NULL && switcher)

{

Obhod(v->Left,L,switcher);

if(switcher)

{

printf("%-6i %s %s %s %i %i\n",L,v->Info->Data->author,v->Info->Data->title, v->Info->Data->publisher, v->Info->Data->year, v->Info->Data->num\_of\_page);

L++;

if(switcher==1 && L%20==1)

{

system("chcp 1251>nul");

printf("\nВы хотите продолжить?\n");

printf("0 -- Нет\n");

printf("1 -- Да, ещё 20 записей\n");

printf("2 -- Да, все оставшиеся записи\n");

scanf("%i",&switcher);

printf("\n");

system("chcp 866>nul");

}

Obhod(v->Down,L,switcher);

Obhod(v->Right,L,switcher);

}

}

}

**Vertex\* TreeSearch(Vertex \*Root)**

{

short int X;

printf("\nВведите кол-во страниц:\t");

scanf("%i",&X);

struct Vertex\* v=Root;

while(v!=NULL)

{

if(X>v->Info->Data->num\_of\_page) v=v->Right;

else if (v->Info->Data->num\_of\_page>X) v=v->Left;

else return v;

}

return NULL;

}

**int Menu(char path[])**

{

int switcher;

system("CLS");

system("chcp 1251>nul");

system("color 02");

printf("Пожалуйста выбирете что вы хотите сделать:\n\n");

printf("1) Просмотреть файл базы данных %s!\n",path);

printf("2) Просмотреть отсортированный файл базы данных %s по году и автору!\n",path);

printf("3) Произвести поиск всех данных в файле %s с указанным годом!\n",path);

printf("4) Просмотреть дерево оптимального поиска A1 базы данных %s!\n",path);

printf("5) Выйти из приложения!\n");

scanf("%i",&switcher);

system("color 07");

system("CLS");

return switcher;

}

**int main()**

{

FILE \*file;

char path[]="testBase1.dat";

file=fopen(path,"rb");

const int N=4000;

printf("%i",sizeof(record));

int L=1, switcher=1;

struct record \*Records = new record[N+1];

ReadDataBase(Records,N,file);

fclose(file);

struct record\* Index\_Records[N+1];

Copy\_Address(Records,Index\_Records,N);

SortDataBase(Index\_Records,N);

struct List \*Head\_List=NULL,\*Tail\_List=NULL;

Vertex \*Root;

int menu\_switcher;

int Size=0, Search;

do

{

menu\_switcher=Menu(path)%5;

switch(menu\_switcher)

{

case 1:

WriteDataBase(Records,N);

break;

case 2:

WriteSortDataBase(Index\_Records,N);

break;

case 3:

BSearchAllDataBase(Index\_Records,N,Head\_List,Tail\_List);

Size=PrintList(Head\_List);

break;

case 4:

Root=A1\_Tree(Head\_List,Size);

system("chcp 866>nul");

Obhod(Root,L,switcher);

system("chcp 1251>nul");

printf("\nВы хотите найти элемент в этом дереве 1/0:\n");

scanf("%i",&Search);

if(Search)

{

Vertex \*Temp=TreeSearch(Root);

if(Temp!=NULL)

{

printf("\nЭлемент с таким кол-вом страниц найден:\n");

system("chcp 866>nul");

while(Temp!=NULL)

{

printf("%s %s %s %i %i\n",Temp->Info->Data->author,Temp->Info->Data->title, Temp->Info->Data->publisher, Temp->Info->Data->year, Temp->Info->Data->num\_of\_page);

Temp=Temp->Down;

}

}

else

{

printf("\nЭлемент с таким кол-вом страниц НЕ найден:\n");

}

}

system("pause");

L=1;

switcher=1;

break;

}

}while(menu\_switcher);

delete []Records;

return 0;

}

1. Файл Huffman\_Kurs.cs с кодированием базы данных:

using System;

using System.IO;

class Byte\_Chance

{

public double Chance;

int value;

public int Num

{

get

{

return value;

}

}

public Byte\_Chance(int i)

{

value=i;

Chance=0;

}

public override string ToString()

{

Console.Write("{0, -4} {1,5: 0.00}",value,Chance);

return "";

//return ""+(char)value+value+":\t"+Chance;

}

}

class Program

{

static int Num\_of\_Bytes=0;

static int Exist=0;

**static void Heap(Byte\_Chance[] Z, int L, int R)**

{

Byte\_Chance X=Z[L];

int i=L, j;

while(true)

{

j=2\*i;

if(j>R) break;

if(j<R && Z[j+1].Chance<=Z[j].Chance) j++;

if(X.Chance<=Z[j].Chance) break;

Z[i]=Z[j];

i=j;

}

Z[i]=X;

}

**static void HeapSort(Byte\_Chance[] Z, int N)**

{

int L=N/2, R=N;

Byte\_Chance t;

while(L>0)

{

Heap(Z,L,N);

L--;

}

while(R>1)

{

t=Z[1];

Z[1]=Z[R];

Z[R]=t;

R--;

Heap(Z,1,R);

}

}

**static int Up(double[] P, double q, int N)**

{

int j=1;

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

{

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

else

{

j=i;

break;

}

}

P[j]=q;

return j;

}

**static void Down(string[] C, int[] L, int j, int N)**

{

string S=new string(C[j].ToCharArray());

int LL=L[j];

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

{

C[i]=new string(C[i+1].ToCharArray());

L[i]=L[i+1];

}

C[N-1]=S+"0";

C[N]=S+"1";

L[N-1]=LL+1;

L[N]=LL+1;

}

**static void Huffman(double[] P, int[] L, string[] C, int N)**

{

if(N==2)

{

C[1]="0";

L[1]=1;

C[2]="1";

L[2]=1;

}

else

{

double q=P[N-1]+P[N];

int j=Up(P,q,N);

Huffman(P,L,C,N-1);

Down(C,L,j,N);

}

}

**static double N\_Tropy(Byte\_Chance[] Z)**

{

double res=0;

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

{

res-=Z[i].Chance\*Math.Log(Z[i].Chance)/Math.Log(2);

}

return res;

}

**static double Average\_Len(Byte\_Chance[] Z, int[] L)**

{

double res=0;

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

{

res+=Z[i].Chance\*L[i];

}

return res;

}

**public static void Main(string[] args)**

{

Console.WriteLine("Программа для кодирования представлена на языке C#");

FileStream fin;

try

{

fin=new FileStream("testBase1.dat",FileMode.Open);

}

catch

{

Console.WriteLine("\nК сожалению, файл не найден по пути поиска.\nПрограмма завершила свою работу!");

Console.ReadLine();

return;

}

int val;

double sum=0;

int[] Byte\_Repeat=new int[257];

Byte\_Chance[] Bytes =new Byte\_Chance[257];

for(int i=1;i<=256;i++) Bytes[i] = new Byte\_Chance(i-1);

while(true)

{

val=fin.ReadByte();

if(val==-1) break;

else

{

Num\_of\_Bytes++;

Byte\_Repeat[val+1]++;

}

}

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

{

if(Byte\_Repeat[i]>0) Exist++;

Bytes[i].Chance=(double) Byte\_Repeat[i]/Num\_of\_Bytes;

Console.WriteLine(Bytes[i]);

}

HeapSort(Bytes,256);

for(int i=2;i<=Exist;i++) sum+=Bytes[i].Chance;

Bytes[1].Chance=(double) 1-sum;

sum+=Bytes[1].Chance;

double[] P=new double[Exist+1];

string[] C=new string[Exist+1];

int[] L=new int[Exist+1];

for(int i=1;i<=Exist;i++) P[i]=Bytes[i].Chance;

Huffman(P,L,C,Exist);

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

{

Console.WriteLine("{0} {1} {2}",Bytes[i],C[i],L[i]);

}

Console.WriteLine("\nЭнтропия источника:\t"+N\_Tropy(Bytes));

Console.WriteLine("\nСредняя длина кодового слова:\t"+Average\_Len(Bytes,L)+"\n");

Console.WriteLine("Сумма вероятностей встречаемости байтов: "+sum);

Console.WriteLine("Количество встреченных значений байтов: "+Exist);

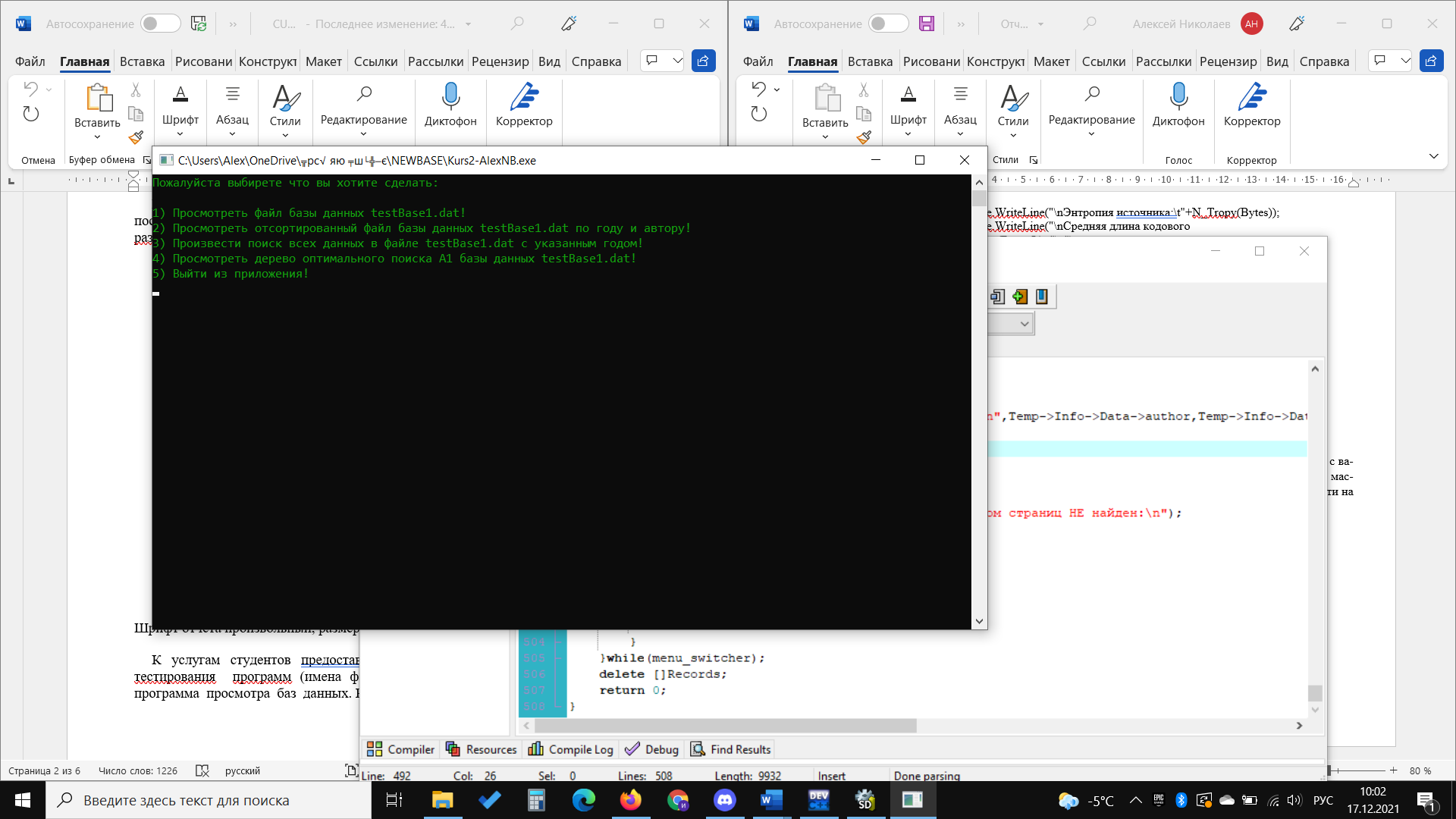
Console.WriteLine("Количество байтов в файле: "+Num\_of\_Bytes+"\n");

Console.ReadLine();

}

}

# **Результаты**



**Рисунок 1.** Меню выбора действий с базой данных

Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 2.** Начало просмотра не отсортированной базы данных

Изображение выглядит как текст, электроника, снимок экрана, компьютер

Автоматически созданное описание

**Рисунок 3.** Конец просмотра не отсортированной базы данных

Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 4.** Начало просмотра отсортированной базы данных

Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 5.** Начало результата поиска по году издания

Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 6.** Конец результата поиска по году издания

Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 7.** Начало обхода дерева из найденных элементов

Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 8.** Конец обхода дерева из найденных элементов из поиск по количеству страниц

Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 9.** Начало результата кода Хаффмана

Изображение выглядит как текст

Автоматически созданное описание

**Рисунок 10.** Конец результата кода Хаффмана и сравнение величин

# **Выводы**

В ходе выполнения данной работы была создана программа, предоставляющая функционал по обработке базы данных "Жизнь замечательных людей":

1) Просмотр исходной базы данных

2) Просмотр отсортированной базы данных

3) Поиск всех записей по ключу поиска

4) Построение и просмотр дерева оптимального поиска

5) Поиск записи по ключу в дереве поиска

6) Кодирование базы данных с помощью оптимального кода Хаффмана.

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки с помощью метода Вилльямса-Флойда, двоичный поиск, построение ДОП по приближённому алгоритму А1, поиск по дереву и кодирование базы данных с помощью оптимального кода Хаффмана.

В результате кодирования были получены данные, подтверждающие теоретические сведения. К таковым относятся: величины средней длины кодового слова и энтропии ( средняя длина кодового слова Lср не сильно больше энтропии H).

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.