Федеральное агентство связи (Россвязь).

Федеральное государственное бюджетное образовательное учреждение «Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра прикладной математики и кибернетики

КУРСОВОЙ ПРОЕКТ

По дисциплине «Структуры и алгоритмы обработки данных»

Вариант 1

Выполнил: Студент 2 курса группы ИП-811

Адов А.С.

Проверил: Доцент кафедры Янченко Е. В.

**Содержание**

[Содержание 2](#__RefHeading__4582_876082447)

[1. Постановка задачи 3](#__RefHeading__4584_876082447)

[2. Основные идеи и характеристики применяемых методов 4](#__RefHeading__4586_876082447)

[2.1. Метод сортировки 4](#__RefHeading__4588_876082447)

[2.2. Двоичный поиск 6](#__RefHeading__4590_876082447)

[2.3. Списки и очереди 7](#__RefHeading__4592_876082447)

[2.4. Вид дерева и поиск 9](#__RefHeading__4594_876082447)

[2.5. Метод кодирования 10](#__RefHeading__4596_876082447)

[3. Описание структур данных и использованных алгоритмов. 11](#__RefHeading__4598_876082447)

[3.1. Использованные структуры данных 11](#__RefHeading__4600_876082447)

[3.2. Особенности реализации алгоритмов 12](#__RefHeading__4602_876082447)

[4. Описание программы 13](#__RefHeading__4604_876082447)

[4.1. Основные переменные и структуры 13](#__RefHeading__4606_876082447)

[4.2. Описание подпрограмм 14](#__RefHeading__4608_876082447)

[5. Исходный текст программы 19](#__RefHeading__4610_876082447)

[6. Результаты 45](#__RefHeading__4612_876082447)

[7. Выводы 46](#__RefHeading__4614_876082447)

**1. Постановка задачи**

Хранящуюся в файле базу данных “Жизнь замечательных людей”(4000 записей) - загрузить динамически в оперативную память с формированием индексного массива как массива указателей, вывести на экран по 20 записей (строк) на странице с возможностью отказа от просмотра.

Упорядочить данные методом сортировки Уильямса-Флойда по фамилии замечательного человека. Упорядоченные данные вывести на экран.

Предусмотреть возможность быстрого поиска по первым трём буквам фамилии в упорядоченной базе, в результате которого из записей с одинаковым ключом формируется очередь, содержимое очереди выводится на экран.

Из записей очереди построить дерево поиска (АВЛ-дерево) по году издания, вывести на экран содержимое дерева и предусмотреть возможность поиска в дереве по запросу.

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

**2. Основные идеи и характеристики применяемых методов**

**2.1. Метод сортировки**

Метод сортировки Уильямса - Флойда основан на алгоритме построения пирамиды.

Последовательность **aL, aL+1, …, aR**называется пирамидой, если неравенство

ai≤ min (a2i, a2i+1)

выполняется для всех i, для которых хотя бы один из элементов a2i и a2i+1 существует.

Свойства пирамиды:

**1. Двустороннее усечение:**

Если последовательность aL, aL+1, ..., аR-1, aR–пирамида, то aL+1, ..., aR-1 тоже пирамида.

**2.** Если a1, a2, ..., an– пирамида, то а1 – минимальный элемент пирамиды.

**3.** Если a1, ..., an– произвольная последовательность, то an/2,.., an– пирамида.

Построение пирамиды:

Пусть aL+1, …, aR - пирамида, необходимо добавить элемент Х, чтобы получить новую пирамиду aL, …, aR.

Новый элемент **добавляем в начало**, расширяя последовательность влево.

Если aLудовлетворяет **условию пирамиды**, то пирамида построена.

Иначе найдутся такие a2L или a2L+1 , что **не будут удовлетворять** условию пирамиды.

Возьмем минимальный элемент из a2L и a2L+1, обозначим его за aj и обменяем с aL.

В результате получим a’L ≤ a2L и a’L ≤ a2L+1, что удовлетворяет **условию пирамиды**.

Идея сортировки:

**Первый этап.** Построение пирамиды из элементов массива.

В соответствии со **свойством3** правая часть массива уже пирамида. Будем добавлять по одному элементу слева, расширяя пирамиду, пока в нее не войдут все элементы массива.

**Второй этап.** Сортировка.

По **свойству 2** в пирамиде первый элемент минимальный. Производим двустороннее усечение пирамиды: уберем элементы а1 и аn. По **свойству1**a2, .., an-1 – пирамида. Поставим элемент а1 на последнее место, а элемент аn добавим к пирамиде a2,..,an-1. Отсекаем последний элемент и повторяем действия, пока пирамида не исчезнет.

Характеристики:

Оценим трудоемкость сортировки, используя уже известную оценку трудоемкости построения пирамиды:

C = 2 M = + 2

На первом этапе построение пирамиды производится n/2 раз, на втором этапе – n-1 раз.

Очевидно, трудоемкость пирамидальной сортировки имеет порядок O(nlog2n), , *n→∞*.

Количество операций сравнения и пересылки оценивается следующими неравенствами:

C < 2 n log2n + n + 2 M < n log2n + 6.5n - 4

Пирамидальная сортировка не устойчива.

Метод практически не зависит от исходной упорядоченности массива.

**2.2. Двоичный поиск**

Идея поиска:

Алгоритм двоичного поиска в упорядоченном массиве сводится к следующему. Берём средний элемент отсортированного массива и сравниваем с ключом X. Возможны три варианта:

1. Выбранный элемент равен X. Поиск завершён.
2. Выбранный элемент меньше X. Продолжаем поиск в правой половине массива.
3. Выбранный элемент больше X. Продолжаем поиск в левой половине массива.

Если в массиве несколько элементов с одинаковым ключом, алгоритм двоичного поиска, используемый в курсовом проекте, находит самый левый из них. Для поиска остальных элементов с заданным ключом требуется просмотреть массив только в одном направлении – вправо от найденного элемента.

Характеристики:

На каждой итерации поиска необходимо одно сравнение. Количество итераций не больше, чем . Таким образом, трудоёмкость двоичного поиска:

С=O(log*n*), *n → ∞*.

**2.3. Списки и очереди**

*Списком* называется последовательность однотипных элементов, связанных между собой указателями.

Поле **Next** может занимать произвольное место в структуре элементов списка. Однако, если оно является первым элементом структуры, то его адрес совпадает с адресом элемента списка, и это позволяет **оптимизировать** многие операции со списками.

*Очередь* – это один из двух видов списка, в котором новый элемент добавляется в конец последовательности, удаляется первый элемент последовательности.

Очередь реализует дисциплину обслуживания **FIFO** (**F**irst**I**nput, **F**irst**O**utput)

Основные операции с очередью:

- Добавление элемента в конец очереди (непустой):

- Добавление элемента в пустую очередь:

- Добавление элемента по адресу p в очередь:

- Исключение первого элемента из очереди

**2.4. Вид дерева и поиск**

**АВЛ-дерево** — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

АВЛ — аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса.

Идея построения:

Вначале **добавимновую вершину** в дерево, так же как **в случайное**, т.е. проходим по пути поиска до нужного места включения в качестве листовой вершины.

**Двигаясь назад по пути поиска,** будем искать вершину, в которой нарушился баланс, т.е. высота левого и правого поддерева стала отличаться больше, чем на единицу.

Если такая вершина найдена, то **изменим структуру дерева** для восстановления баланса.

При включении новой вершины её баланс равен нулю. При движении назад по пути поиска **показатель баланса для всех вершин пересчитывается**, причем не нужно просматривать все поддеревья, только путь поиска.

**Нарушение баланса возможно только в одной вершине и один поворот полностью восстанавливает структуру АВЛ-дерева.**

Балансировка выполняется с помощью поворотов дерева: **LL, LR, RL, RR**.

Характеристики:

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

log(*n*+1) ≤ hАВЛ(*n*) < 1,44 log(*n*+2) – 0,328 при *n→∞.*

Идея поиска:

Начиная с корневой вершины дерева, *сравниваем* **ключ поиска** с данными **в текущей вершине**.

Если **ключ** поиска **меньше**, то переходим в **левое поддерево**, если **ключ** поиска **больше**, то переходим в **правое поддерево**.

Действуем аналогично, **пока не будет найден элемент** с заданным ключом **или листовая вершина** дерева.

Если **достигнута листовая вершина**, то искомого элемента **нет в дереве**.

**2.5. Метод кодирования**

Характеристики:

Пусть имеется **дискретный вероятностный источник**, порождающий символы алфавита с вероятностями .

Основной характеристикой источника является **энтропия**, которая представляет собой

**среднее значение количества информации** в сообщении источника и определяется выражением (для двоичного случая):

Энтропия характеризует **меру неопределенности выбора** для данного источника.

Код называется алфавитным, если кодовые слова лексикографически упорядочены, т.е.

Хаффманом был предложен оптимальный код.

**Алгоритм построения оптимального кода Хаффмана**

1. Упорядочим символы исходного алфавита *А*={*a*1*,…,an*} по убыванию их вероятностей ***p1 ≥ p2 ≥ … ≥ pn***.

2. Если ***А* = {*a*1*,a2*}**, то ***a*1→0**, ***a*2→1**.

3. Если ***А* = {*a*1 *, … , aj , … , an*}** и известны коды **< *aj* → *bj* >**,

*j=1,…,n*, то для алфавита **{ *a1 , … aj /, aj //… , an* }**

с новыми символами ***aj /*** *и* ***aj //***вместо ***aj***, и вероятностями ***p(aj ) = p(aj /) + p(aj //)****,*

код символа ***aj***заменяется на коды *a****j /→ bj0****,* ***aj //→ bj1***.

**3. Описание структур данных и использованных алгоритмов.**

**3.1. Использованные структуры данных**

В курсовом проекте были использованы следующие структуры данных:

Простые структуры:

* Числовые
* Символьные
* Логические
* Указатели

Составные структуры:

* Массивы - структура данных, хранящая набор значений (элементов массива), идентифицируемых по индексу или набору индексов, принимающих целые (или приводимые к целым) значения из некоторого заданного непрерывного диапазона
* Очереди - список, операции чтения и добавления элементов в котором подвержены определенным правилам. При этом, при чтении элемента, он удаляется из очереди. Все операции проводятся по принципу «Первый пришел, первый вышел» (FIFO — first in, first out). Таким образом, для чтения в очереди доступна только голова, в то время как добавление проводится только в хвост
* Деревья - структура данных, представляющая собой древовидную структуру в виде набора связанных узлов.

**3.2. Особенности реализации алгоритмов**

В реализации алгоритма построения АВЛ – дерева можно выделить следующие особенности:

- Использование рекурсии, которая позволит хранить адреса всех пройденных вершин по пути поиска и автоматически в них возвращаться на обратном пути рекурсии.

- В каждую вершину дерева вводится дополнительный показатель баланса balance.

- Дополнительная глобальная переменная Height обозначающая, что произошел рост дерева.

В реализации алгоритма сортировки методом Уильямса – Флойда присутствуют следующие особенности:

- Отдельная процедура для построения пирамиды.

В реализации алгоритма кодирования методом Хаффмана выделю:

- Использование отдельных вспомогательных функций Up и Down.

**4. Описание программы**

**4.1. Основные переменные и структуры**

Основные переменные:

* DB\_SIZE – количество элементов которые нужно считать из файла
* record \*note – указатель на массив считанных из файла записей
* record \*\*index - указатель на массив указателей на записи, считанные из файла
* queue \*find – указатель на очередь найденных по запросу элементов
* vertex \*root – указатель на корень АВЛ-дерева

Основные структуры:

* record

Структура записи из базы данных, состоящая из полей:

* char author[12] — автор
* char title[32] — ФИО замечательного человека
* char publishing[16] — издательство
* short int year — год издания
* short int page — количество страниц
* list\_entry

Структура элемента очереди, состоящая из полей:

* struct le\_s \*next – указатель на следующий элемент очереди
* record \*data – указатель на запись
* queue

Структура очереди, состоящая из полей:

* list\_entry \*head – указатель на головной элемент очереди
* list\_entry \*tail – указатель на хвостовой элемент очереди
* vertex

Структура элемента дерева, состоящая из полей:

* balance – баланс элемента дерева
* record \*data – данные элемента дерева, представленные в виде указателя на очередь, состоящую из записей
* struct tree \*left – указатель на левую вершину дерева
* struct tree \*right – указатель на правую вершину дерева

**4.2. Описание подпрограмм**

* void \*Read\_Data\_Base\_To\_Array(const char \*file\_name, record \*\*note)

Параметры:

const \*file\_name – имя файла

record \*\* note – адрес массива в который считываются записи

Функция открывает file в бинарном режиме, создает массив структур и считывает в данный массив структур DB\_SIZE структур из файла.

* void Print\_Data\_Base(record \*index[], int Number)

Параметры:

record \*\*index – массив указателей на структуры

Number– размер массива db

Функция выводит на экран 20 записей массива index.

* void Build\_Heap(record \*index[], short int L, short int R)

Параметры:

record \*\*index – массив указателей на структуры

short int L – левая граница для построения пирамиды

short int R – правая граница для построения пирамиды

Функция производит построение пирамиды на интервале массива index от L до R.

* void Heap\_Sort(record \*\*index)

Параметры:

record \*\*index – массив указателей на структуры

Функция производит сортировку массива index методом Уильямса-Флойда.

* bool Equally(char \*A, char \*X)

Возвращает bool значение. True, если запись A равна X ,false иначе.

Параметры:

char \*A – первая запись для сравнения

char \*X – вторая запись для сравнения

Функция проверяет равенство искомого ключа и найденной записи.

* void \*Print\_Queue(queue \*find,short int Number)

Параметры:

queue \*find – очередь которую нужно вывести на экран

short int Number – итератор для отображения порядкового номера записи в очереди

Функция отображает страницу вывода содержимого очереди.

* void \*Create\_Queue\_Found\_Note(record \*\*index, queue \*\*find,short int Number,char \*X)

Параметры:

record \*\* index – массив указателей на записи считанные из базы данных

queue \*\*find — очередь которая будет заполнена найденными элементами

short int Number — переменная содержащая номер первой записи из найенных

char \*X — введенный ключ поиска состоящий из первых трех букв фамилии замечательного человека

Функция создает очередь элементов удовлетворяющим поиску по ключу X

* short int Search\_in\_Data\_Base(record \*\*index,char \*X)

Возвращает номер первой записи удовлетворяющий ключу поиска X

Параметры:

record \*\*index – массив указателей на записи считанные из базы данных

char \*X - введенный ключ поиска состоящий из первых трех букв фамилии замечательного человека

Функция находит первую запись совпадающую с ключом поиска X

* int Less(record a[], record b[])

Возвращает 2 если первая сравниваемая запись больше второй по полю title, 1 если записи равны , 0 если первая сравниваемая запись по полю title меньше второй

Параметры:

record a[] - первая сравниваемая запись

record b[] - вторая сравниваемая запись

Функция осуществляет сравнение двух записей по полю title

* bool Less\_for\_Search(char \*A, char \*X)

Возвращает true если ключ поиска меньше или равен проверяемой записи, false если меньше

Параметры:

char \*A – поле title проверяемой записи

char \*X – ключ поиска состоящий из трех букв

* void Menu()

Функция выводит на экран текст поясняющий ввод какого значения приведет к какому действию.

* void \*Add\_in\_AVL(record \*d,vertex \*\*p)

Параметры:

record \*d – запись которая будет добавлена в значение вершины АВЛ дерева

vertex \*\*p — указатель на вершину АВЛ дерева в которую будет произведено добавление

Функция осуществляет добавление данных в АВЛ дерево

* void \*LL\_turn(vertex \*\*p)

Параметры:

vertex \*\*p – ссылка на указатель на вершину дерева, в которой нарушился баланс

Функция осуществляет LL поворот в АВЛ-дереве.

* void \*LL\_turn(vertex \*\*p)

Параметры:

vertex \*\*p – ссылка на указатель на вершину дерева, в которой нарушился баланс

Функция осуществляет RR поворот в АВЛ-дереве.

* void \*LR\_turn(vertex \*\*p)

Параметры:

vertex \*\*p– ссылка на указатель на вершину дерева, в которой нарушился баланс

Функция осуществляет LR поворот в АВЛ-дереве.

* void \*RL\_turn(vertex \*\*p)

Параметры:

vertex \*\*p – ссылка на указатель на вершину дерева, в которой нарушился баланс

Функция осуществляет RL поворот в АВЛ-дереве.

* void Print\_AVL(vertex \*p)

Параметры:

vertex \*\*p – ссылка на указатель на вершину дерева которая будет распечатана

Функция производит обход дерева слева направо для распечатывания данных из вершин

* void \*Find\_in\_AVL(short int X,vertex \*p)

Параметры:

short int X – ключ поиска в АВЛ дереве

vertex \*p – указатель на проверяемую вершину

Функция находит самую близкую к корню вершину данные которой по полю year совпадают с ключом поиска X

* void Print\_Found\_in\_AVL(vertex \*p,short int X)

Параметры:

vertex \*p -указатель на распечатываемую вершину

short int X -ключ поиска

Функция распечатывает при помощи обхода слева направо , каждый раз проверяя равенство данных из вершины по полю year с ключом поиска , начиная с вершины найденной при поиске в АВЛ дереве

* void Reading\_Symbols()

Функция осуществляет побайтовое считывание символов из базы данных и вывод на экран основных характеристик кодирования таких как энтропия

* void Fill\_Chance\_Meet\_of\_Symbols(char a, double \*\*A, int n)

Параметры:

char a – считанный символ из базы данных

double \*\*A -массив содержащий символы и частоты их встречаемости в базе данных

int n – переменная отражающая 256 символов

Функция добавляет в массив A символ если он ранее не встречался иначе увеличивает частоту встречаемости

* void Quick\_Sort\_For\_Chance(double \*\*A, int L, int R)

Параметры:

double \*\* A – массив содержащий символы и частоты их встречаемости в базе данных

int L – левая граница массива

int R – правая граница массива

Функция осуществляет быструю сортировку массива A по частоте встречаемости символов

* void Coding\_Symbols(double \*\*A, int n)

Параметры:

double \*\*A - массив содержащий символы и частоты их встречаемости в базе данных

int n – переменная отражающая 256 символов

Функция определяет вероятность для символов а также длины кодовых слов и распечатывает неравенство Крафта, символы и их вероятности, кодовые слова , и длины кодовых слов

* void Huffman(int n, double probability[], bool code\_word[][20], double length[])

Параметры:

int n – переменная отражающая 256 символов

double probability[] - массив вероятностей

bool code\_word[][20] — массив кодовых слов

double length[] - массив длин кодовых слов

Функция осуществляет оптимальное кодирование Хаффмана

* int Up(int n, double q, double probability[])

Функция возвращает переменную отражающую номер места куда вставлено число q

Параметры:

int n – переменная отражающая 256 символов

double probability[] - массив вероятностей

double q – сумма двух наименьших вероятностей

Функция находит в массиве вероятностей место куда вставить число q и вставляет его сдвигая вниз остальные элементы

* void Down(int n, int j, double length[], bool code\_word[][20])

Параметры:

int n – переменная отражающая 256 символов

double probability[] - массив вероятностей

bool code\_word[][20] — массив кодовых слов

Функция формирует кодовые слова для соответсвующих символов

**5. Исходный текст программы**

***task\_1\_1\_1\_1\_1.c***

#include <stdlib.h>

#include <stdio.h>

#include <stdbool.h>

#include <malloc.h>

#include <locale.h>

#include <string.h>

#include <windows.h>

#include <math.h>

#define DB\_SIZE 4000

typedef struct

{

char author[12];

char title[32];

char publishing[16];

short int year;

short int page;

} record;

typedef struct le\_s

{

struct le\_s \*next;

record \*data;

} list\_entry;

typedef struct

{

list\_entry \*head;

list\_entry \*tail;

} queue;

typedef struct tree

{

record \*data;

int balance;

struct tree \*left;

struct tree \*right;

} vertex;

bool height;

short int iterator1=0;

short int iterator2=0;

void Menu()

{

printf("1) Read Data Base in list \n");

printf("2) Sorting Data Base\n");

printf("3) Print Data Base\n");

printf("4) Create AVL-tree for Data Base\n");

printf("5) Find in Data Base\n");

printf("6) Coding elements by Huffman\n");

printf("8) Clear Display\n");

printf("0) Exit\n");

}

void Print\_Data\_Base(record \*index[], int Number)

{

while (1)

{

short int key = 1;

for (int i = Number; i < Number + 20; i++)

{

printf("\n%d) %s\t", i + 1, index[i]->author);

printf("%s\t", index[i]->title);

printf("%s\t", index[i]->publishing);

printf("%d\t", index[i]->year);

printf("%d", index[i]->page);

}

Number = Number + 20;

if (Number >= DB\_SIZE)

{

return;

}

printf("\n\nContinue-? Yes-1\n\n");

scanf("%hu", &key);

if (key != 1)

{

return;

}

}

printf("\n");

}

int Less(record a[], record b[])

{

short int f1 = 0, f2 = 0;

for (short int i = 0, j = 0; i < 32; i++, j++)

{

if (f1 == 2)

{

i--;

}

else

{

if (a->title[i] == ' ')

{

f1++;

}

}

if (f2 == 2)

{

j--;

}

else

{

if (b->title[j] == ' ')

{

f2++;

}

}

if (f1 == 2 && f2 == 2)

{

if (a->title[i] < b->title[j])

{

return 0;

}

else

{

if (a->title[i] > b->title[j])

{

return 2;

}

else

{

i++;

j++;

}

}

}

}

return 1;

}

void Build\_Heap(record \*index[], short int L, short int R)

{

short int i = L, j, z = 0;

record \*p = index[L];

while (1)

{

j = 2 \* i;

if (j > R)

{

break;

}

z = Less(index[j + 1], index[j]);

if (j < R && z == 2)

{

j = j + 1;

}

z = Less(p, index[j]);

if (z > 0)

{

break;

}

index[i] = index[j];

i = j;

}

index[i] = p;

}

void Heap\_Sort(record \*\*index)

{

short int L = (DB\_SIZE - 1) / 2, R = DB\_SIZE - 1;

record \*temp;

while (L >= 0)

{

Build\_Heap(index, L, DB\_SIZE - 1);

L--;

}

while (R > 0)

{

temp = index[0];

index[0] = index[R];

index[R] = temp;

R--;

Build\_Heap(index, 0, R);

}

}

void \*Read\_Data\_Base\_To\_Array(const char \*file\_name, record \*\*note)

{

\*note = (record \*)malloc(DB\_SIZE \* sizeof(record));

FILE \*file = fopen(file\_name, "rb");

fread(\*note, sizeof(record), DB\_SIZE, file);

fclose(file);

}

bool Less\_for\_Search(char \*A, char \*X)

{

short int f1 = 0;

for (short int i = 0; i < 32; i++)

{

if (A[i] == ' ')

{

f1++;

}

if (f1 == 2)

{

for (short int j = 0; j < 3; j++, i++)

{

if(X[j]<A[i+1])

{

return true;

}

else

{

if(X[j]>A[i+1])

{

return false;

}

}

}

return true;

}

}

}

bool Equally(char \*A, char \*X)

{

short int f1 = 0;

for (short int i = 0; i < 32; i++)

{

if (A[i] == ' ')

{

f1++;

}

if (f1 == 2)

{

for (short int j = 0; j < 3; j++, i++)

{

if (A[i+1] != X[j])

{

return false;

}

}

return true;

}

}

}

void \*Create\_Queue\_Found\_Note(record \*\*index, queue \*\*find,short int Number,char \*X)

{

bool continue\_queue\_flag=true;

list\_entry \*le =NULL;

le=(list\_entry\*) malloc (sizeof(list\_entry));

\*find=(queue\*) malloc (sizeof(queue));

le->data = index[Number];

le->next = NULL;

(\*find)->head=le;

(\*find)->tail=le;

Number++;

while(continue\_queue\_flag==true&&Number<DB\_SIZE)

{

if(Equally(index[Number]->title,X)==false)

{

continue\_queue\_flag=false;

break;

}

le=(list\_entry\*) malloc (sizeof(list\_entry));

le->data = index[Number];

le->next = NULL;

(\*find)->tail->next=le;

(\*find)->tail=le;

Number++;

}

}

short int Search\_in\_Data\_Base(record \*\*index,char \*X)

{

short int L = 0, R = DB\_SIZE-1, midle = 0;

while (L < R)

{

int midle =(L+R)/2;

if (Less\_for\_Search(index[midle]->title, X) == true)

{

R = midle;

}

else

{

L = midle+1;

}

}

if (Equally(index[R]->title, X) == true)

{

return R;

}

else

{

return -1;

}

}

void \*Print\_Queue(queue \*find,short int Number)

{

find->tail=find->head;

while(find->tail!=NULL)

{

printf("\n%d) %s\t", Number + 1, find->tail->data->author);

printf("%s\t", find->tail->data->title);

printf("%s\t", find->tail->data->publishing);

printf("%d\t", find->tail->data->year);

printf("%d", find->tail->data->page);

Number++;

find->tail=find->tail->next;

}

printf("\n");

}

void \*LL\_turn(vertex \*\*p)

{

vertex \*q;

q=(\*p)->left;

q->balance=0;

(\*p)->balance=0;

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

q->right=(\*p);

(\*p)=q;

}

void \*LR\_turn(vertex \*\*p)

{

vertex \*q,\*r;

q=(\*p)->left;

r=q->right;

if (r->balance<0)

{

(\*p)->balance=1;

}

else

{

(\*p)->balance=0;

}

if (r->balance>0)

{

q->balance=-1;

}

else q->balance=0;

r->balance=0;

(\*p)->left=r->right;

q->right=r->left;

r->left=q;

r->right=(\*p);

(\*p)=r;

}

void \*RR\_turn(vertex \*\*p)

{

vertex \*q;

q=(\*p)->right;

q->balance=0;

(\*p)->balance=0;

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

q->left=(\*p);

(\*p)=q;

}

void \*RL\_turn(vertex \*\*p)

{

vertex \*q,\*r;

q=(\*p)->right;

r=q->left;

if (r->balance>0)

{

(\*p)->balance=-1;

}

else

{

(\*p)->balance=0;

}

if (r->balance<0)

{

q->balance=1;

}

else

{

q->balance=0;

}

r->balance=0;

(\*p)->right=r->left;

q->left=r->right;

r->left=(\*p);

r->right=q;

(\*p)=r;

}

void Print\_AVL(vertex \*p)

{

if (p!=NULL)

{

Print\_AVL(p->left);

iterator1++;

printf("\n%hu) %s\t", iterator1 , p->data->author);

printf("%s\t", p->data->title);

printf("%s\t", p->data->publishing);

printf("%hu\t", p->data->year);

printf("%hu", p->data->page);

Print\_AVL(p->right);

}

}

void Print\_Found\_in\_AVL(vertex \*p,short int X)

{

if (p!=NULL)

{

Print\_Found\_in\_AVL(p->left,X);

if(p->data->year==X)

{

iterator2++;

printf("\n%d) %s\t", iterator2 , p->data->author);

printf("%s\t", p->data->title);

printf("%s\t", p->data->publishing);

printf("%d\t", p->data->year);

printf("%d", p->data->page);

}

Print\_Found\_in\_AVL(p->right,X);

}

}

void \*Find\_in\_AVL(short int X,vertex \*p)

{

if(p!=NULL)

{

if(X>p->data->year)

{

Find\_in\_AVL(X,p->right);

}

else

{

if(X<p->data->year)

{

Find\_in\_AVL(X,p->left);

}

else

{

Print\_Found\_in\_AVL(p,X);

}

}

}

else

{

printf("Need Notes not Found\n");

}

printf("\n");

}

void \*Add\_in\_AVL(record \*d,vertex \*\*p)

{

if ((\*p)==NULL)

{

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

(\*p)->data=d;

(\*p)->left=NULL;

(\*p)->right=NULL;

(\*p)->balance=0;

height=true;

}

else

{

if ((\*p)->data->year>d->year)

{

vertex \*v=NULL;

v=(\*p)->left;

Add\_in\_AVL(d,&v);

(\*p)->left=v;

if (height==true)

{

if ((\*p)->balance>0)

{

(\*p)->balance=0;

height=false;

}

else

{

if ((\*p)->balance==0)

{

(\*p)->balance=-1;

}

else

{

if ((\*p)->left->balance<0)

{

LL\_turn(p);

height=false;

}

else

{

LR\_turn(p);

height=false;

}

}

}

}

}

else

{

if ((\*p)->data->year<=d->year)

{

vertex \*v=NULL;

v=(\*p)->right;

Add\_in\_AVL(d,&v);

(\*p)->right=v;

if (height==true)

{

if ((\*p)->balance<0)

{

(\*p)->balance=0;

height=false;

}

}

else

{

if ((\*p)->balance==0)

{

(\*p)->balance=1;

}

else

{

if ((\*p)->right->balance>0)

{

RR\_turn(p);

height=false;

}

else

{

RL\_turn(p);

height=false;

}

}

}

}

}

}

}

void Fill\_Chance\_Meet\_of\_Symbols(char a, double \*\*A, int n)

{

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

{

if(A[i][0]==a)

{

A[i][1]++;

return;

}

else if(A[i][1]==0)

{

A[i][0]=a;

A[i][1]++;

return;

}

}

}

void Quick\_Sort\_For\_Chance(double \*\*A, int L, int R)

{

int j=R,i=L;

double X=A[(L+R)/2][1];

double t,m;

while(i<=j)

{

while(A[i][1]>X)

{

i++;

}

while(A[j][1]<X)

{

j--;

}

if(i <= j)

{

m=A[j][0];

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

A[i][0]=m;

t=A[j][1];

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

A[i][1]=t;

i++;

j--;

}

}

if(L<j)

{

Quick\_Sort\_For\_Chance(A,L,j);

}

if(i<R)

{

Quick\_Sort\_For\_Chance(A,i,R);

}

return;

}

void Down(int n, int j, double length[], bool code\_word[][20])

{

bool S[20]={0};

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

{

S[i]=code\_word[j][i];

}

int l=length[j];

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

{

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

{

code\_word[i][k]=code\_word[i+1][k];

}

length[i]=length[i+1];

}

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

{

code\_word[n-1][i]=S[i];

code\_word[n][i]=S[i];

}

code\_word[n-1][l]=0;

code\_word[n][l]=1;

length[n-1]=l+1;

length[n]=l+1;

}

int Up(int n, double q, double probability[])

{

int j=1;

for(int i=n-1;i>1;i--)

{

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

{

probability[i]=probability[i-1];

}

else

{

j=i;

break;

}

}

probability[j]=q;

return j;

}

void Huffman(int n, double probability[], bool code\_word[][20], double length[])

{

if(n==2)

{

code\_word[1][0]=0;

length[1]=1;

code\_word[2][0]=1;

length[2]=1;

}

else

{

double q=probability[n-1]+probability[n];

int j=Up(n,q,probability);

Huffman(n-1,probability,code\_word,length);

Down(n,j,length,code\_word);

}

}

void Coding\_Symbols(double \*\*A, int n)

{

double average\_code\_length=0,kraft\_nequality=0;

double probability[n+1],length[n+1];

for(short int i=0;i<n+1;i++)

{

probability[i]=0;

length[i]=0;

}

bool code\_word[n+1][20];

for(short int i=0;i<n+1;i++)

{

for(short int j=0;j<20;j++)

{

code\_word[i][j]=false;

}

}

Quick\_Sort\_For\_Chance(A,0,n-1);

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

{

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

}

Huffman(n,probability,code\_word,length);

printf("\nSymbol\tProbability\t\tCode word\tStake word length\n");

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

{

average\_code\_length+=A[i-1][1]\*length[i];

kraft\_nequality+=pow(2,(length[i]\*-1));

printf("%hu)%c\t%f\t\t",i,(char)A[i-1][0],A[i-1][1]);

for(int j=0;j<length[i];j++)

{

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

}

printf("\t\t%0.f\n",length[i]);

}

printf("Average length of code words: %f\n",average\_code\_length);

if(kraft\_nequality<=1)

{

printf("Kraft-McMillan's nequality: %f <=1\n",kraft\_nequality);

}

else

{

printf("Kraft-McMillan's nequality: %f >1\n",kraft\_nequality);

}

}

void Reading\_Symbols()

{

FILE \*file;

file=fopen("testBase1.dat","rb");

int n=257,sum=0;

double \*\*A,intr=0;

A=(double\*\*) malloc (n \*sizeof(double\*));

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

{

A[i] = (double\*) malloc (2 \* sizeof(double));

A[i][0]=0;

A[i][1]=0;

}

char a=0,digit[4]={0};

short int temp=0;

while(fread(&a, sizeof(char), 1, file)&&(!feof(file)))

{

Fill\_Chance\_Meet\_of\_Symbols(a,A,n);

}

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

{

sum+=A[i][1];

if(A[i][1]==0)

{

temp=i;

break;

}

}

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

{

free(A[i]);

}

n=temp;

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

{

printf("%c = %f \t--\t",(char)A[i][0],A[i][1]);

A[i][1]/=sum;

printf("%f\n",A[i][1]);

}

printf("\nNumber of characters read: %f\n",sum);

double g=0;

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

{

g+=A[i][1];

}

printf("Sum of the probability of characters : %f",g);

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

{

intr+=log(A[i][1])/log(2)\*A[i][1];

}

intr=intr\*-1;

Coding\_Symbols(A,n);

printf("Entropy: %f\n",intr);

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

{

free(A[i]);

}

free(A);

}

int main()

{

bool sort\_flag = false;

short int head\_for\_queue;

record \*note = NULL;

record \*\*index = NULL;

queue \*find = NULL;

vertex \*root=NULL;

while (1)

{

short int key = 0;

Menu();

scanf("%hu", &key);

switch (key)

{

case 1:

{

Read\_Data\_Base\_To\_Array("testBase1.dat", &note);

index = (record \*\*)malloc(DB\_SIZE \* sizeof(record));

for (short int i = 0; i < DB\_SIZE; i++)

{

index[i] = &note[i];

}

sort\_flag = false;

break;

}

case 2:

{

if ((index != NULL) && (sort\_flag == false))

{

Heap\_Sort(index);

sort\_flag = true;

}

else

{

printf("Need Read Data Base or Data Base Already Sorting\n");

}

break;

}

case 3:

{

if (note != NULL)

{

Print\_Data\_Base(index, 0);

}

else

{

printf("Data Base is not Read\n");

}

break;

}

case 4:

{

if(find==NULL)

{

printf("Need Search Note");

}

else

{

find->tail=find->head;

while(find->tail!=NULL)

{

Add\_in\_AVL(find->tail->data,&root);

find->tail=find->tail->next;

}

}

Print\_AVL(root);

printf("\nDo You Want to Search by Year around found notes? 1-Yes\n");

short int key\_search=0;

scanf("%hu",&key\_search);

if(key\_search==1)

{

short int exit\_key=1;

while(exit\_key==1)

{

printf("Write Need Year\n");

short int Xy=0;

scanf("%hu",&Xy);

Find\_in\_AVL(Xy,root);

printf("If You Want Continue - Press 1\n");

scanf("%hu",&exit\_key);

}

root=NULL;

iterator1=0;

iterator2=0;

}

break;

}

case 5:

{

if (sort\_flag == true)

{

find=NULL;

char X[3];

SetConsoleCP(866);

printf("Write What Note Need Find\n");

scanf("%s", &X);

short int first\_queue\_note=Search\_in\_Data\_Base(index ,X);

if(first\_queue\_note==-1)

{

printf("Need Note not Found\n");

break;

}

Create\_Queue\_Found\_Note(index,&find,first\_queue\_note,X);

Print\_Queue(find,first\_queue\_note);

}

else

{

printf("Need Sort Data Base\n");

}

break;

}

case 6:

{

Reading\_Symbols();

break;

}

case 8:

{

system("CLS");

break;

}

case 0:

{

return 0;

}

}

}

}

**6. Результаты**

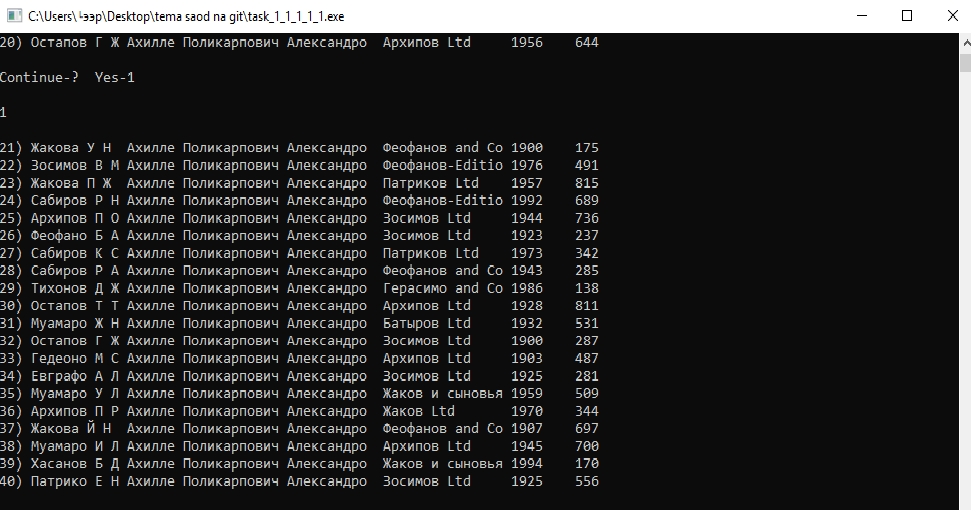


Рисунок 1. Отсортированные записи по фамилии замечательного человека

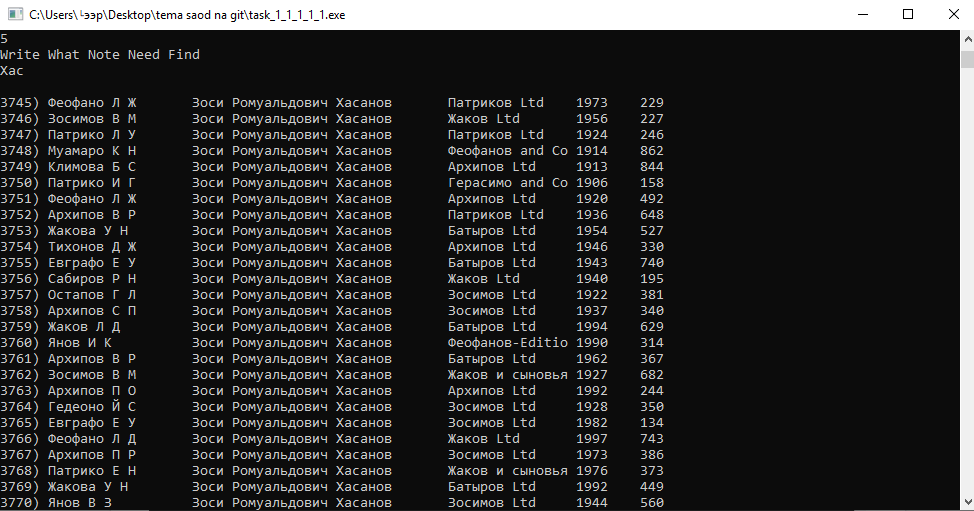


Рисунок 2. Бинарный поиск по ключу Хас

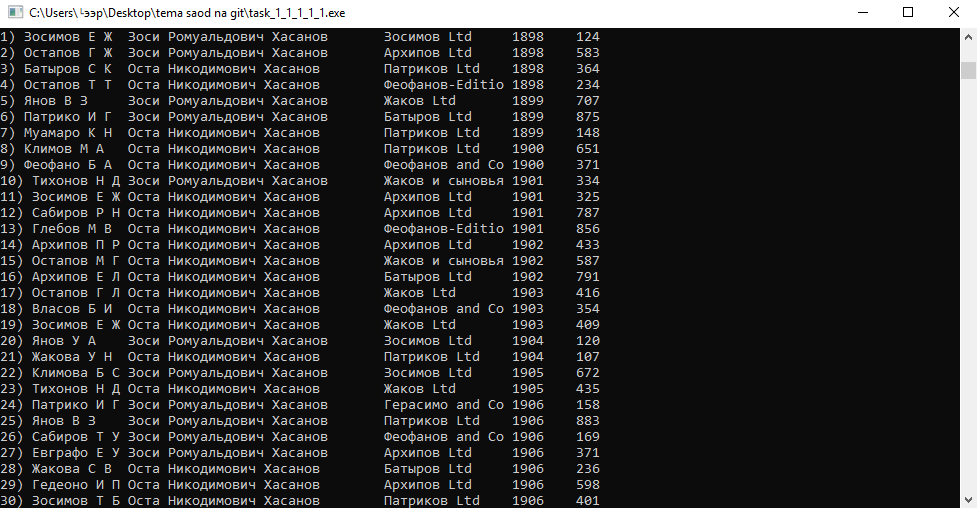


Рисунок 3. Обход построенного АВЛ дерева по году издания

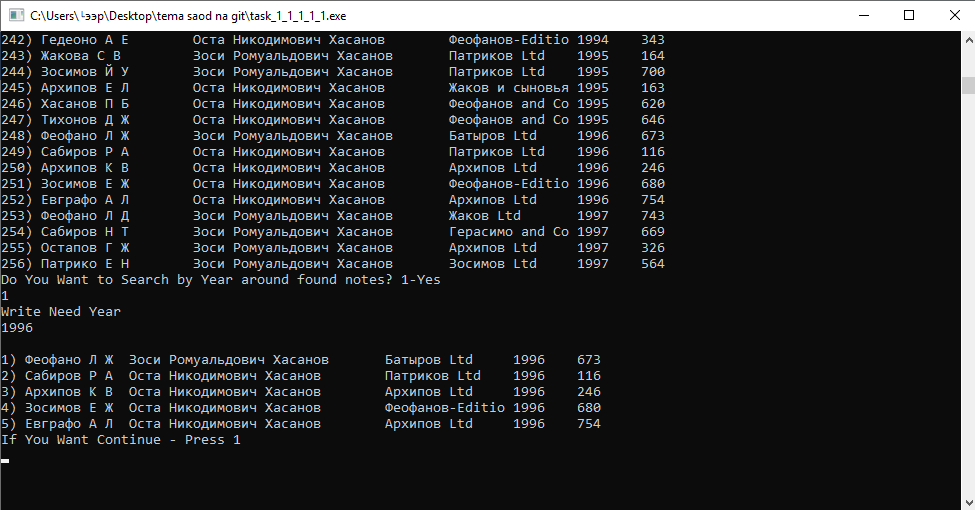


Рисунок 4. Поиск в АВЛ дереве по ключу 1996

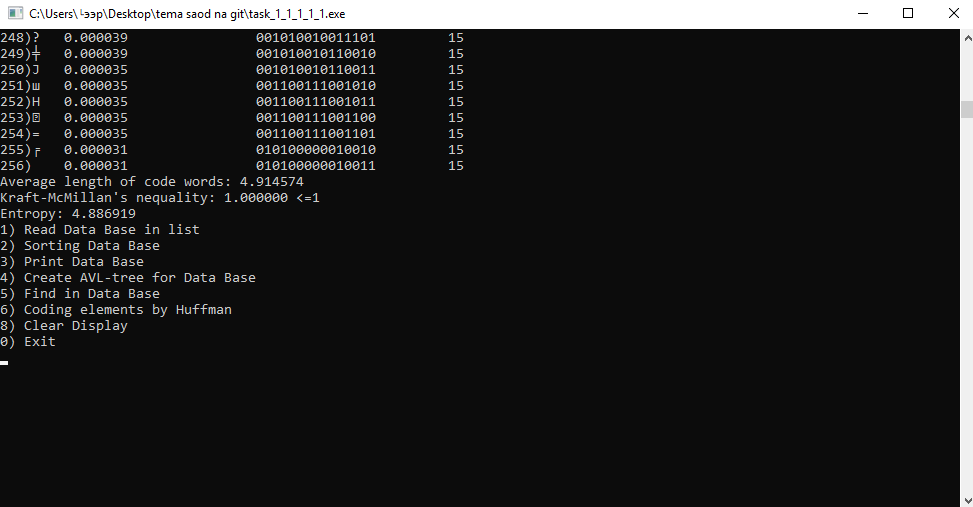


Рисунок 5. Энтропия, средняя длина кодового слова и неравенство Крафта для оптимального кода Хаффмана примененного к нашей базе данных

**7. Выводы**

В ходе выполнения данной работы была создана программа, предоставляющая функционал по обработке базы данных “Жизнь замечательных людей”.

А именно использование данной программы позволяет:

1. Вывести содержимое БД
2. Отсортировать записи методом Уильямса-Флойда по фамилиям замечательных людей
3. Выполнить двоичный поиск по первым трём буквам фамилии
4. Построить дерево поиска по ключу (АВЛ-дерево)
5. Выполнить поиск в дереве по году издания
6. Закодировать базу оптимальным кодом Хаффмана

По моему мнению, задача, поставленная в начале работы над курсовым проектом, была успешно выполнена.

Благодаря данному проекту я научился производить обработку объемных данных, а также, пройдя весь путь написания программы для курсового проекта, я познал все тонкости предоставленных алгоритмов.