Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации  
СибГУТИ

Кафедра ПМиК

КУРСОВОЙ ПРОЕКТ

“Структуры и алгоритмы обработки данных”

ВАРИАНТ 140

Выполнил:

студент группы ИП-216

Русецкий А. С.

Проверил:

Старший преподаватель

Кафедры ПМиК

Солодов П.С

Новосибирск 2023

СОДЕРЖАНИЕ

1. ПОСТАНОВКА ЗАДАЧИ
2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ
   1. Метод сортировки
   2. Двоичный поиск
   3. Дерево и поиск по дереву
   4. Метод кодирования
3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ
4. ОПИСАНИЕ ПРОГРАММЫ
   1. Основные переменные и структуры
   2. Описание подпрограмм
5. ТЕКСТ ПРОГРАММЫ
6. РЕЗУЛЬТАТЫ
7. ВЫВОДЫ

1. ПОСТАНОВКА ЗАДАЧИ

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

Упорядочить данные **номеру отдела и ФИО**, используя **Цифровую сортировку (Digital Sort)**. Упорядоченные данные вывести на экран.

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

Из записей очереди построить **дерево оптимального поиска по номеру отдела**, вывести на экран содержимое дерева и предусмотреть возможность поиска в дереве по запросу.

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

База данных "Пpедпpиятие"

Стpуктуpа записи:

ФИО сотpудника: текстовое поле 30 символа

фоpмат <Фамилия>\_<Имя>\_<Отчество>

Hомеp отдела: целое число

Должность: текстовое поле 22 символа

Дата pождения: текстовое поле 10 символов

фоpмат дд-мм-гг

Пpимеp записи из БД:

Петpов\_Иван\_Иванович\_\_\_\_\_\_\_\_\_\_\_\_

130

начальник\_отдела\_\_\_\_\_\_

15-03-46

Ваpиант условий упоpядочения и ключи поиска (К):

C = 1 - по номеpу отдела и ФИО, К = номеp отдела;

Ключ в дереве – первые 3 буквы фамилии.

1. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

2.1. Метод сортировки

Цифровая сортировка (Digital Sort)

Цифровая сортировка является одним из методов сортировки последовательностей.

Пусть дана последовательность из S чисел, представленных в m – ичной системе счисления. Каждое число состоит из L цифр d1d2…dL, 0 ≤ di ≤ m – 1, i=1..L. Сначала числа из списка S распределяются по m очередям, причём номер очереди определяется последней цифрой каждого числа. Затем полученные очереди соединяются в список, для которого все действия повторяются, но распределение по очередям производится в соответствии со следующей цифрой и т.д.

Цифровой метод может успешно использоваться не только для сортировки чисел, но и для сортировки любой информации, представленной в памяти компьютера. Необходимо лишь рассматривать каждый байт ключа сортировки как цифру, принимающую значения от 0 до 255. Тогда для сортировки потребуется m=256 очередей. Для выделения каждого байта ключа сортировки можно использовать массив Digit, наложенный в памяти компьютера на поле элемента последовательности, по которому происходит сортировка. Приведем более детальный алгоритм цифровой сортировки.

Для цифровой сортировки М<const L(*m+n*). При фиксированных *m* и L М=O(*n*) при *n → ∞*, что значительно быстрее остальных рассмотренных методов. Однако если длина чисел L велика, то метод может проигрывать обычным методам сортировки. Кроме того, Метод применим только, если задача сортировки сводится к задаче упорядочивания чисел, что не всегда возможно.

Метод обеспечивает устойчивую сортировку.

2.2. Двоичный поиск

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

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.

Дадим верхнюю оценку трудоёмкости алгоритма двоичного поиска. На каждой итерации поиска необходимо два сравнение для первой версии, одно сравнение для второй версии. Количество итераций не больше, чем ⎡log2 n⎤ . Таким образом, трудоёмкость двоичного поиска в обоих случаях

С=O(log n), n → ∞.

2.3. Дерево и поиск по дереву

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 транслятора, который определяет, является ли каждое слово программы (идентификатор) служебным. Статистические измерения на сотнях транслируемых программ могут в этом случае дать точную информацию об относительных частотах появления в тексте отдельных ключей.

Припишем каждой вершине дерева Vi вес wi , пропорциональный частоте поиска этой вершины (например, если из каждых 100 операций поиска 15 операций приходятся на вершину V1, то w1=15). Сумма весов всех вершин дает вес дерева W. Каждая вершина Vi расположена на высоте hi , корень расположен на высоте 1. Высота вершины равна количеству операций сравнения, необходимых для поиска этой вершины. Определим средневзвешенную высоту дерева с n вершинами следующим образом: hср=(w1h1+w2h2+…+wnhn)/W. Дерево поиска, имеющее минимальную средневзвешенную высоту, называется деревом оптимального поиска (ДОП).

Алгоритм (А1) предлагает в качестве корня использовать вершину с наибольшим весом. Затем среди оставшихся вершин снова выбирается вершина с наибольшим весом и помещается в левое или правое поддерево в зависимости от ее значения, и т.д.

2.4. Метод кодирования

Алфавитный код Гилберта – Мура

Е. Н. Гилбертом и Э. Ф. Муром был предложен метод построения алфавитного кода, для которого Lcp < H( p1 ,...,pn ) + 2. Пусть символы алфавита некоторым образом упорядочены, например, a1≤a2≤…≤an. Код σ называется алфавитным, если кодовые слова лексикографически упорядочены, т.е. σ (a1) ≤σ (a2) ≤ ... ≤σ (an).

Процесс построения кода происходит следующим образом.

1. Вычислим величины Qi , i=1,n:

Q1=p1/2,

Q2=p1+p2/2,

Q3=p1+p2+p3/2,

…

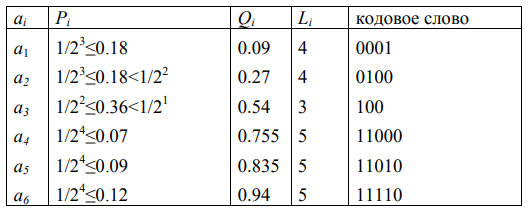
Qn=p1+p2+…+pn-1+pn/2.

2. Представим суммы Qi в двоичном виде.

3. В качестве кодовых слов возьмем ⎡− log pi⎤ +1 младших бит в двоичном представлении Qi , i =1,...,n.

**Пример.** Пусть дан алфавит A={a1, a2, a3, a4, a5, a6} с вероятностями p1=0.36, p2=0.18, p3=0.18, p4=0.12, p5=0.09, p6=0.07. Построенный код приведен в таблице 8.

Таблица 8 Код Гилберта-Мура

  
 Рисунок 1 - Код Гилберта-Мура

Средняя длина кодового слова не превышает значения энтропии плюс 2. Действительно, Lср=4. 0.18+4. 0.18+3. 0.36+5. 0.07+5. 0.09+5. 0.12=3.92<2.37+2

1. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ
2. Интерфейс программы

Данный код реализует загрузку и вывод базы данных, представленной в бинарном файле "testBase2.dat". Сначала создается динамический массив указателей на структуру base, где каждый элемент Интерфейс программы реализован в бесконечном цикле с возможностью выхода из него, при нажатии клавиши «Esc». Выбор пункта меню реализован с помощью функции switch( ).

1. Особенности реализации бинарного поиска и построения очереди

База данных открывается в функции menu(). Считывается база данных в список структур в функции Read\_base(FILE \*fp, list \*rbase), где struct base – список, в котором хранится база данных. Считывание производится независимо от желания пользователя, в то время как большинство остальных функций он может выбрать посредствам меню. После считывания в список структур, файл закрывается.

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

1. Вспомогательные функции и процедуры для сортировки данных

База данных сортируется после запуска программы. Для сортировки базы данных используется процедура void digital\_sort(list \*&head, int sort). Она сортирует список сначала по полю должости, а потом по ФИО. Для быстрого доступа к отсортированной и неотсортированной базе данных, перед вызовом процедуры сортировки, делается копия списка, хранящего элементы базы данных, с помощью процедуры void copy\_base(list \*a, list \*b).

Особенности реализации бинарного поиска

Бинарный поиск по отсортированной базе данных осуществляется в процедуре void BSearch(list \*\*A, int Number). Результатом работы процедуры void BSearch(list \*\*A, int Number) является индекс найденного элемента в индексном массиве, удовлетворяющий условию поиска. При отсутствии элементов с заданным ключом, программа выводит сообщение о том, что таких элементов нет.

Вспомогательные функции и процедуры для построения дерева оптимального поиска (приближенный алгоритм А1).

Построение дерева осуществляется в процедуре void A1(int L, int R, list \*\*mas). Записи заносятся в дерево в процедуре void add\_vertex(vertex \*&p, list\* mas, int w). Обход дерева слева направо осуществляется в процедуре void LR\_print(vertex \*p, int& count). Поиск по дереву выполняется в процедуре void TreeSearch(vertex\* p, char\* name).

1. Кодирование данных

Кодирование данных начинается с процедуры void GillbertMoorCode( ), которая открывает файл базы данных для чтения, заполняет массив структур для алфавита кодовых слов всеми возможными символами, считывает из файла символы и считает их вероятности, закрывает файл, удаляет пустые символы, т.е. те, которые не встретились в файле и сортирует полученный алфавит по вероятностям. В процедуре считается длина кодового слова и само кодовое слово. В процедуре void CodePrint() осуществляется подсчет и вывод средней длины кодового слова, энтропии, а также выводит символы, их вероятности, длины кодовых слов и сами кодовые слова на монитор.

1. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

4.1. Основные программы и подпрограммы

struct base

{

char FIO[30];

short int Department;

char Position[22];

char Date[10];

};

Структура для хранения элемента базы данных. Всего таких элементов:

const int N = 4000 - размер базы данных.

struct list {

base \*data;

list \*next;

list \*prev;

};

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

struct queue {

list \*head;

list \*tail;

};

Структура, используемая для цифровой сортировки.

struct vertex {

record \*data;

vertex \*next;

int weight;

vertex \*left;

vertex \*right;

};

Структура, представляющая дерево оптимального поиска (А2).

int \*W – вес элементов, из которых строится дерево.

struct GM\_code {

unsigned char a;

float q;

int w[30];

int l;

}

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

GM\_code A[M]; - массив для всех символов

GM\_code B[alphabet\_num]; - массив для всех появляющихся символов

const int M = 256 – число символов в алфавите

const int alphabet\_num =81– количество элементов в итоговом алфавите, исключающем пустые символы

float entropy = (float)(0) - энтропия

float lgm = (float)(0) – средняя длина кодового слова

int sum =0 – счётчик всех символов в файле

1. ОПИСАНИЕ ПРОГРАММЫ
   1. Основные переменные

Структура record:

char fullname[32]; - полное имя (строка до 32 символов)

char street[18]; - название улицы (строка до 18 символов)

short int numberHouse; - номер дома (короткое целое число)

short int numberApartment; - номер квартиры (короткое целое число)

char dateSettle[10]; - дата заселения (строка до 10 символов)

const int N = 4000; - константа, определяющая размер массива и другие структуры данных

Структура node:

record\* data; - указатель на данные типа record

node\* next; - указатель на следующий элемент типа node

Структура MyQueue:

node\* head; - указатель на начало очереди

node\* tail; - указатель на конец очереди

Структура BTree:

vector<MyQueue> data; - вектор данных типа MyQueue

int balance; - баланс для бинарного дерева

BTree\* left; - указатель на левое поддерево

BTree\* right; - указатель на правое поддерево

int main() - главная функция программы

record\* locality - массив записей (база данных)

record indexArr - индексный массив для сортировки по улице и номеру дома

record indexArrStreet - индексный массив для сортировки по улице

MyQueue recordQ - очередь записей

BTree\* root - корень бинарного дерева

Дополнительные переменные в функции checkKeyMenu:

int currentPage - текущая страница при выводе записей

string targetKey - целевой ключ при поиске записей

char key - клавиша, введенная пользователем

bool flag - флаг для управления циклами

* 1. Описание подпрограмм

1. Структура «record»:

- Описывает структуру данных для хранения записей, таких как ФИО, улица, номер дома, номер квартиры и дата заселения.

2. Структура «node»:

- Представляет узел в связанном списке («MyQueue»). Содержит указатель на данные и указатель на следующий узел.

3. Структура «MyQueue»:

- Реализует очередь на основе связанного списка для хранения записей.

- Методы «push», «pop», «front», «empty» и «size» предоставляют функциональность добавления, удаления, получения элемента из головы, проверки на пустоту и получения размера очереди соответственно.

4. Функция «BinarySearch»:

- Реализует быстрый поиск по ключу в упорядоченном массиве записей («indexArr»).

- Использует бинарный поиск для нахождения всех записей с заданным ключом, а затем сохраняет их в очередь.

5. Функция «InsertInBTree»:

- Рекурсивно вставляет записи в B-дерево с учетом балансировки.

- Использует флаги «VR» и «HR» для обозначения необходимости вращения и балансировки.

6. Функция «InOrderTraversal»:

- Рекурсивно обходит B-дерево в порядке возрастания ключей и выводит записи.

7. Функции «heapify» и «heapSort»:

- Реализуют пирамидальную сортировку для массива записей по улице и номеру дома.

8. Функции «compareByStreet», «heapifyStreet» и «heapSortStreet»:

- Поддерживают сортировку записей по улице.

9. Функция «inputKey»:

- Запрашивает у пользователя ввод ключа для поиска.

10. Функция «printMenu»:

- Выводит меню пользователя в консоль.

11. Функции «printStartLine» и «printEndLine»:

- Выводят форматированные линии для красивого отображения записей.

12. Функции «printRecord», «printSortRecord» и «printQueue»:

- Выводят записи на экран в рамках текущей страницы с учетом пагинации.

13. Функции «printControlMenu» и «checkKey»:

- Выводят меню управления и обрабатывают действия пользователя при просмотре записей.

14. Функции «printTreeKey», «checkKeyTree» и «printControlMenuTree»:

- Отвечают за отображение и взаимодействие с B-деревом.

15. Функции «coding»:

- Реализует алгоритм Хаффмана для кодирования записей.

16. Функции «get\_char\_counts\_from\_file» и «calc\_probabilities»:

- Считывают количество символов из файла и вычисляют вероятности их появления.

17. Функции «up» и «down»:

- Используются в алгоритме Хаффмана для перемещения вверх и вниз в дереве кодирования.

18. Функция «huffman»:

- Реализует алгоритм Хаффмана.

19. Функция «checkKeyMenu»:

- Отвечает за взаимодействие пользователя с основным меню приложения.

1. ТЕКСТ ПРОГРАММЫ

#include <fstream>

#include <iostream>

#include <vector>

#include <cstring>

#include <cmath>

#include <algorithm>

#include <unordered\_map>

#include <stdio.h>

#include <conio.h>

#include <unistd.h>

using namespace std;

struct record {

char fullname[32];

char street[18];

short int numberHouse;

short int numberApartment;

char dateSettle[10];

};

const int N = 4000;

// Структура узла

struct node {

record\* data;

node\* next;

};

// Структура очереди

struct MyQueue {

node\* head;

node\* tail;

MyQueue() : head(nullptr), tail(nullptr) {}

void push(record\* data) {

node\* newNode = new node;

newNode->data = data;

newNode->next = nullptr;

if (!head) {

head = newNode;

tail = newNode;

} else {

tail->next = newNode;

tail = newNode;

}

}

bool empty() const {

return !head;

}

record\* front() const {

if (head) {

return head->data;

}

throw runtime\_error("Queue is empty");

}

void pop() {

if (head) {

node\* temp = head;

head = head->next;

delete temp;

}

}

int size() const {

int count = 0;

node\* current = head;

while (current) {

count++;

current = current->next;

}

return count;

}

};

// Быстрый поиск по ключу

MyQueue BinarySearch(record\*\* indexArr, const string& targetKey) {

MyQueue result; // Очередь для хранения записей с одинаковым ключом

int arrSize = 4000;

int left = 0;

int right = arrSize - 1;

while (left <= right) {

int mid = (left + right) / 2;

// Извлекаем ключ из фамилии

string key(indexArr[mid]->street, 3);

if (key == targetKey) {

// Если ключ совпадает, добавляем запись в очередь

result.push(indexArr[mid]);

// Поиск всех записей с таким же ключом влево

int leftIndex = mid - 1; // Изменено значение

while (leftIndex >= 0 && strncmp(indexArr[leftIndex]->street, targetKey.c\_str(), 3) == 0) {

result.push(indexArr[leftIndex]);

leftIndex--;

}

// Поиск всех записей с таким же ключом вправо

int rightIndex = mid + 1; // Изменено значение

while (rightIndex < arrSize && strncmp(indexArr[rightIndex]->street, targetKey.c\_str(), 3) == 0) {

result.push(indexArr[rightIndex]);

rightIndex++;

}

return result;

} else if (key < targetKey) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return result;

}

// Ввод ключа

string inputKey() {

string key;

cout << "Введите 3 буквы улицы : ";

cin >> key;

if (key.length() > 3) {

key = key.substr(0, 3);

}

return key;

}

struct BTree {

vector<MyQueue> data;

int balance;

BTree\* left;

BTree\* right;

};

void InsertInBTree(MyQueue& data, BTree\*& root, bool& VR, bool& HR)

{

if (root == nullptr) {

root = new BTree;

root->data.push\_back(data);

root->balance = 0;

root->left = nullptr;

root->right = nullptr;

VR = false;

HR = false;

} else {

if (data.front()->numberApartment < root->data.front().head->data->numberApartment) {

InsertInBTree(data, root->left, VR, HR);

if (VR) {

if (root->balance == 0) {

BTree\* q = root->left;

root->left = q->right;

q->right = root;

root = q;

q->balance = 1;

VR = false;

HR = true;

}

else {

root->balance = 0;

HR = true;

}

}

else {

HR = false;

}

}

else if (data.front()->numberApartment > root->data.front().head->data->numberApartment) {

InsertInBTree(data, root->right, VR, HR);

if (VR) {

root->balance = 1;

VR = false;

HR = true;

}

else if (HR) {

if (root->balance > 0) {

BTree\* q = root->right;

root->right = q->left;

q->left = root;

root->balance = 0;

q->balance = 0;

root->left = root;

root = q;

VR = true;

HR = false;

}

else {

HR = false;

}

}

else {

HR = false;

}

}

else {

root->data.push\_back(data);

}

}

}

void InOrderTraversal(BTree\* root) {

if (root == nullptr) {

return;

}

InOrderTraversal(root->left);

for (size\_t i = 0; i < root->data.size(); i++) {

cout << root->data[i].head->data->fullname

<< "\t"

<< root->data[i].head->data->street

<< "\t"

<< root->data[i].head->data->numberHouse

<< "\t\t"

<< root->data[i].head->data->numberApartment

<< "\t\t"

<< root->data[i].head->data->dateSettle

<< endl;

}

InOrderTraversal(root->right);

}

//Пирамидальная сортировка

void heapify(record\*\* indexArr, int size, int i)

{

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < size && (strcmp(indexArr[left]->street, indexArr[largest]->street) > 0

|| (strcmp(indexArr[left]->street, indexArr[largest]->street) == 0

&& indexArr[left]->numberHouse > indexArr[largest]->numberHouse)))

largest = left;

if (right < size && (strcmp(indexArr[right]->street, indexArr[largest]->street) > 0

|| (strcmp(indexArr[right]->street, indexArr[largest]->street) == 0

&& indexArr[right]->numberHouse > indexArr[largest]->numberHouse)))

largest = right;

if (largest != i) {

swap(indexArr[i], indexArr[largest]);

heapify(indexArr, size, largest);

}

}

void heapSort(record\*\* indexArr, int size)

{

for (int i = size / 2 - 1; i >= 0; i--)

heapify(indexArr, size, i);

for (int i = size - 1; i >= 0; i--) {

swap(indexArr[0], indexArr[i]);

heapify(indexArr, i, 0);

}

}

// Функция для сравнения двух записей по улице

int compareByStreet(record\* a, record\* b) {

return strcmp(a->street, b->street);

}

// Пирамидальная сортировка для улицы

void heapifyStreet(record\*\* arr, int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && compareByStreet(arr[left], arr[largest]) > 0)

largest = left;

if (right < n && compareByStreet(arr[right], arr[largest]) > 0)

largest = right;

if (largest != i) {

swap(arr[i], arr[largest]);

heapifyStreet(arr, n, largest);

}

}

void heapSortStreet(record\*\* arr, int n) {

for (int i = n / 2 - 1; i >= 0; i--)

heapifyStreet(arr, n, i);

for (int i = n - 1; i > 0; i--) {

swap(arr[0], arr[i]);

heapifyStreet(arr, i, 0);

}

}

//Вывод меню

void printMenu() {

cout << "╔══════════════════════════════════════════════════════════════╗"

<< "\n"

<< "║ Меню ║"

<< "\n"

<< "╠══════════════════════════════════════════════════════════════╣"

<< "\n"

<< "║ 1. Посмотреть базу данных ║"

<< "\n"

<< "║ 2. Посмотреть отсортированную базу данных ║"

<< "\n"

<< "║ 3. Поиск по ключу ║"

<< "\n"

<< "║ 4. Вывод двоичного B-дерева поиска ║"

<< "\n"

<< "║ 5. Кодирование ║"

<< "\n"

<< "╚══════════════════════════════════════════════════════════════╝"

<< "\n\n"

<< ">>> ";

}

// Вывод начальной линии

void printStartLine()

{

cout << "╔═══════════════════════════════════════╦══════════════════╦═════════════════╦════════════════════╦════════════════════╗"

<< "\n"

<< "║\t\t "

<< "ФИО"

<< "\t\t ║ "

<< "Улица"

<< " ║ "

<< "Номер дома"

<< " ║ "

<< "Номер квартиры"

<< " ║ "

<< "Дата поселения ║"

<< endl;

}

// Вывод конецной линии

void printEndLine()

{

cout << "╚═══════════════════════════════════════╩══════════════════╩═════════════════╩════════════════════╩════════════════════╝" << "\n\n";

}

// Вывод базы данных

void printRecord(record\* locality, int& currentPage)

{

for (int i = currentPage \* 20; i < currentPage \* 20 + 20; i++) {

cout << "╠═══════════════════════════════════════╬══════════════════╬═════════════════╬════════════════════╬════════════════════╣"

<< "\n"

<< "║"

<< i + 1

<< ". "

<< locality[i].fullname

<< "\t║ "

<< locality[i].street

<< "║\t "

<< locality[i].numberHouse

<< "\t ║\t "

<< locality[i].numberApartment

<< "\t ║ "

<< locality[i].dateSettle

<< " ║"

<< endl;

}

}

// Вывод сортированной базы данных

void printSortRecord(record\*\* indexArr, int& currentPage)

{

for (int i = currentPage \* 20; i < currentPage \* 20 + 20; i++) {

cout << "╠═══════════════════════════════════════╬══════════════════╬═════════════════╬════════════════════╬════════════════════╣"

<< "\n"

<< "║"

<< i + 1

<< ". "

<< indexArr[i]->fullname

<< "\t║ "

<< indexArr[i]->street

<< "║\t "

<< indexArr[i]->numberHouse

<< "\t ║\t "

<< indexArr[i]->numberApartment

<< "\t ║ "

<< indexArr[i]->dateSettle

<< " ║"

<< endl;

}

}

// Вывод очереди

void printQueue(const MyQueue& result, int currentPage)

{

node\* currentNode = result.head;

int counter = 0;

int startIndex = currentPage \* 20;

// Пропускаем записи, чтобы начать с нужного индекса на текущей странице

while (counter < startIndex && currentNode) {

currentNode = currentNode->next;

counter++;

}

// Выводим до 20 записей на текущей странице

while (currentNode && counter < startIndex + 20) {

record\* recordPtr = currentNode->data;

cout << "╠═══════════════════════════════════════╬══════════════════╬═════════════════╬════════════════════╬════════════════════╣"

<< "\n"

<< "║"

<< counter + 1

<< ". "

<< recordPtr->fullname

<< "\t║ "

<< recordPtr->street

<< "║\t "

<< recordPtr->numberHouse

<< "\t ║\t "

<< recordPtr->numberApartment

<< "\t ║ "

<< recordPtr->dateSettle

<< " ║"

<< endl;

currentNode = currentNode->next;

counter++;

}

}

// Вывод меню управления

void printControlMenu()

{

cout << "<q> : Выйти из программы"

<< "\t\t "

<< "<h> : Предыдущая стрница"

<< "\t\t\t"

<< "<l> : Следующая страница"

<< "\n"

<< "<r> : Поиск записи"

<< "\t\t\t "

<< "<j> : Прошлые 10 записей"

<< "\t\t\t"

<< "<k> : Следующие 10 записей"

<< endl;

}

// Проверка клавиш управления

bool checkKey(int& currentPage, int totalRecords) {

char key = getch();

bool flag = true;

switch (key) {

case 'q':

system("cls");

flag = false;

break;

case 'h':

currentPage--;

if (currentPage < 0) currentPage = 0;

break;

case 'l':

if ((currentPage + 1) \* 20 < totalRecords) {

currentPage++;

}

break;

case 'r':

int numberPage;

cout << "Введите номер записи : ";

cin >> numberPage;

if (numberPage > 0 && numberPage <= totalRecords) {

currentPage = floor((numberPage - 1) / 20);

} else {

cout << "Ошибка!" << endl;

sleep(1);

}

break;

case 'j':

currentPage -= 10;

if (currentPage < 0) currentPage = 0;

break;

case 'k':

currentPage += 10;

if (currentPage \* 20 >= totalRecords) {

currentPage = (totalRecords - 1) / 20;

}

break;

}

return flag;

}

short int numTree() {

system("cls");

short int num = 0;

cout << "Введите номер квартиры : ";

cin >> num;

return num;

}

void printTreeKey(BTree\* root, int num) {

if (root == nullptr) {

return;

}

printTreeKey(root->left, num);

for (size\_t i = 0; i < root->data.size(); i++) {

if (root->data[i].head->data->numberApartment == num) {

cout << root->data[i].head->data->fullname

<< "\t"

<< root->data[i].head->data->street

<< "\t"

<< root->data[i].head->data->numberHouse

<< "\t\t "

<< root->data[i].head->data->numberApartment

<< "\t\t "

<< root->data[i].head->data->dateSettle

<< endl;

}

}

printTreeKey(root->right, num);

}

bool checkKeyTree(BTree\* root) {

char key = getch();

bool flag = true;

short int num;

switch (key) {

case 'q' :

system("cls");

flag = false;

break;

case 'r' :

num = numTree();

system("cls");

printTreeKey(root, num);

system("pause");

break;

}

return flag;

}

void printControlMenuTree() {

cout << "<q> : Выйти из программы"

<< "\t"

<< "<r> : Поиск по ключу"

<< endl;

}

int up(int n, double q, double \*array, double chance[]) {

int i = 0, j = 0;

for (i = n - 2; i >= 1; i--) {

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

else {

j = i;

break;

}

if ((i - 1) == 0 && chance[i - 1] < q) {

j = 0;

break;

}

}

array[j] = q;

return j;

}

void down(int n, int j, int Length[], char c[][20]) {

char pref[20];

for (int i = 0; i < 20; i++) pref[i] = c[j][i];

int l = Length[j];

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

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

c[i][k] = c[i+1][k];

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

}

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

c[n-2][i] = pref[i];

c[n-1][i] = pref[i];

}

c[n-1][l] = '1';

c[n-2][l] = '0';

Length[n-1] = l + 1;

Length[n-2] = l + 1;

}

void huffman(int n, double \*array, int Length[], char c[][20], double chance[]) {

if (n == 2) {

c[0][0] = '0';

Length[0] = 1;

c[1][0] = '1';

Length[1] = 1;

} else {

double q = array[n - 2] + array[n - 1];

int j = up(n, q, array, chance);

huffman(n - 1, array, Length, c, chance);

down(n, j, Length, c);

}

}

unordered\_map<char, int> get\_char\_counts\_from\_file(const string &file\_name, int &file\_size, int n = N) {

ifstream file(file\_name, ios::binary);

if (!file.is\_open()) {

throw runtime\_error("Не удалось открыть файл!");

}

char ch\_arr[sizeof(record) \* n];

file.read((char \*) ch\_arr, sizeof(record) \* n);

file.close();

unordered\_map<char, int> counter\_map;

file\_size = 0;

for (auto ch : ch\_arr) {

counter\_map[ch]++;

file\_size++;

}

return counter\_map;

}

vector<pair<double, char>> calc\_probabilities(const unordered\_map<char, int> &counter\_map, int count) {

vector<pair<double, char>> probabilities;

probabilities.reserve(counter\_map.size());

for (auto i : counter\_map) {

probabilities.emplace\_back(make\_pair((double) i.second / count, i.first));

}

return probabilities;

}

void coding() {

int file\_size;

unordered\_map<char, int> counter\_map;

counter\_map = get\_char\_counts\_from\_file("testBase4.dat", file\_size);

auto probabilities = calc\_probabilities(counter\_map, file\_size);

counter\_map.clear();

sort(probabilities.begin(), probabilities.end(), greater<pair<double, char>>());

cout << "\nProbabil. Сhar\n";

for (auto i : probabilities) {

cout << fixed << i.first << " | " << i.second << '\n';

}

const int n = (int) probabilities.size();

char c[n][20];

int Length[n];

for (auto &i : Length) {

i = 0;

}

auto p = new double[n];

double chance\_l[n];

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

p[i] = probabilities[i].first;

chance\_l[i] = p[i];

}

huffman(n, chance\_l, Length, c, p);

cout << "\nКод Хаффмена:\n";

cout << "\nCh Prob Code\n";

double avg\_len = 0;

double entropy = 0;

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

avg\_len += Length[i] \* p[i];

entropy -= p[i] \* log2(p[i]);

printf("%c | %.5lf | ", probabilities[i].second, p[i]);

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

printf("%c", c[i][j]);

}

cout << '\n';

}

cout << "\n"

<< "Средняя длина = " << avg\_len << '\n'

<< "Энтропия = " << entropy << '\n'

<< "Средняя длина < энтропии + 1\n"

<< "N = " << n << "\n\n";

}

// Проверка клавиш меню

void checkKeyMenu(record\* locality, record\*\* indexArr, record\*\* indexArrStreet, int& currentPage, MyQueue& result, BTree\*& root)

{

printMenu();

string targetKey;

char key = getch();

bool flag = true;

currentPage = 0;

MyQueue tempQueue;

bool VR = false;

bool HR = false;

switch (key) {

case '1':

while(flag) {

system("cls");

printStartLine();

printRecord(locality, currentPage);

printEndLine();

printControlMenu();

flag = checkKey(currentPage, 4000);

}

break;

case '2':

while(flag) {

system("cls");

printStartLine();

printSortRecord(indexArr, currentPage);

printEndLine();

printControlMenu();

flag = checkKey(currentPage, 4000);

}

break;

case '3':

system("cls");

targetKey = inputKey();

while (flag) {

result = BinarySearch(indexArrStreet, targetKey);

if (result.empty()) {

std::cout << "Записи с ключом " << targetKey << " не найдены." << std::endl;

sleep(3);

break;

} else {

system("cls");

printStartLine();

printQueue(result, currentPage);

printEndLine();

printControlMenu();

flag = checkKey(currentPage, result.size());

}

}

break;

case '4':

if(!result.empty()) {

root = nullptr;

while(!(result.head == nullptr)) {

InsertInBTree(result, root, VR, HR);

result.head = result.head->next;

}

while(flag) {

system("cls");

InOrderTraversal(root);

cout << endl;

printControlMenuTree();

flag = checkKeyTree(root);

}

} else {

cout << "Очередь пустая! Сделайте сначала пункт 3!";

sleep(5);

}

break;

case '5':

while(flag) {

system("cls");

coding();

cout << "\n\n<q> : Выйти из программы" << endl;

if (getch() == 'q') {

system("cls");

flag = false;

}

}

break;

default :

exit(0);

break;

}

}

int main()

{

system("chcp 866 > nul");

system("cls");

int currentPage = 0;

record\* locality = new record[N];

ifstream fileDateBase("testBase4.dat", ios::binary);

if (!fileDateBase) {

cout << "Не удалось открыть файл!" << endl;

return 1;

}

for (int i = 0; !fileDateBase.read((char\*)&locality[i], sizeof(record)).eof(); i++);

// Индексный массив для сортировки по улице и номеру дома

record\*\* indexArr = new record\*[N];

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

indexArr[i] = &locality[i];

}

heapSort(indexArr, N);

// Индексный массив для сортировки по улице

record\*\* indexArrStreet = new record\*[N];

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

indexArrStreet[i] = &locality[i];

}

heapSortStreet(indexArrStreet, N);

MyQueue recordQ;

BTree\* root;

while(true) {

checkKeyMenu(locality, indexArr, indexArrStreet, currentPage, recordQ, root);

system("cls");

}

delete[] locality;

delete[] indexArr;

delete[] indexArrStreet;

fileDateBase.close();

return 0;

}

1. РЕЗУЛЬТАТЫ

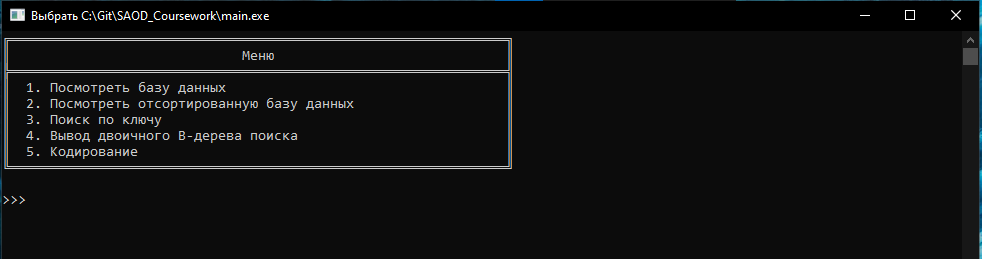


Рисунок 6 — Внешний вид меню

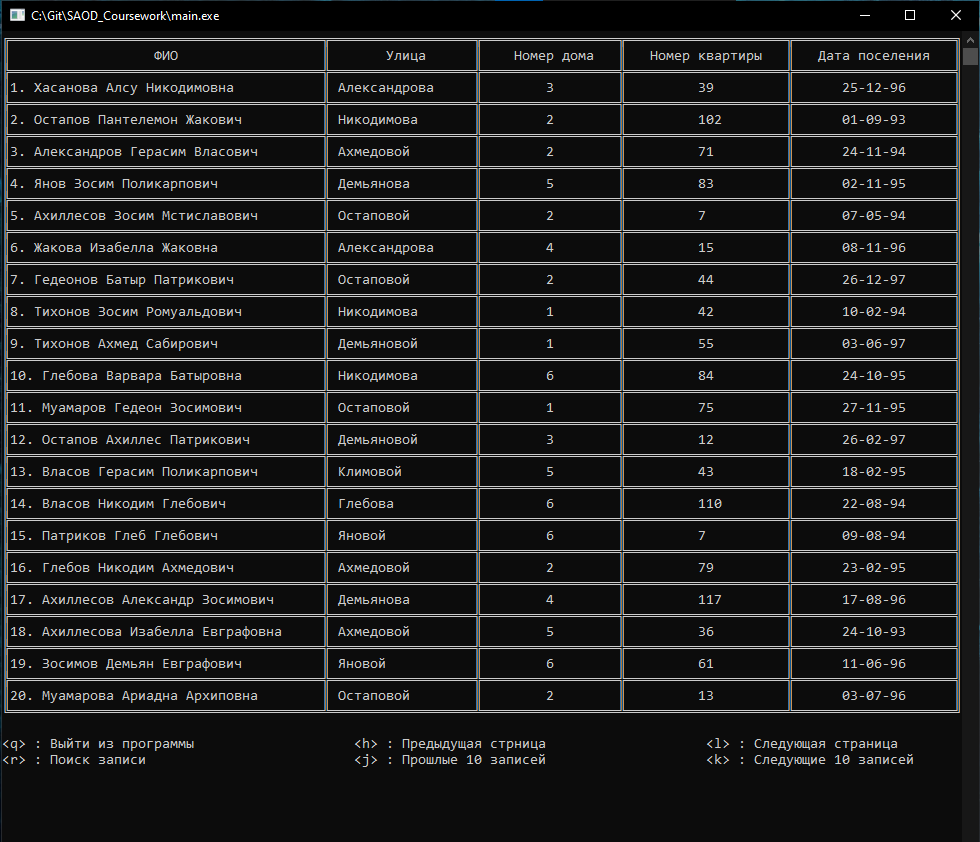


Рисунок 7 — Вывод базы данных

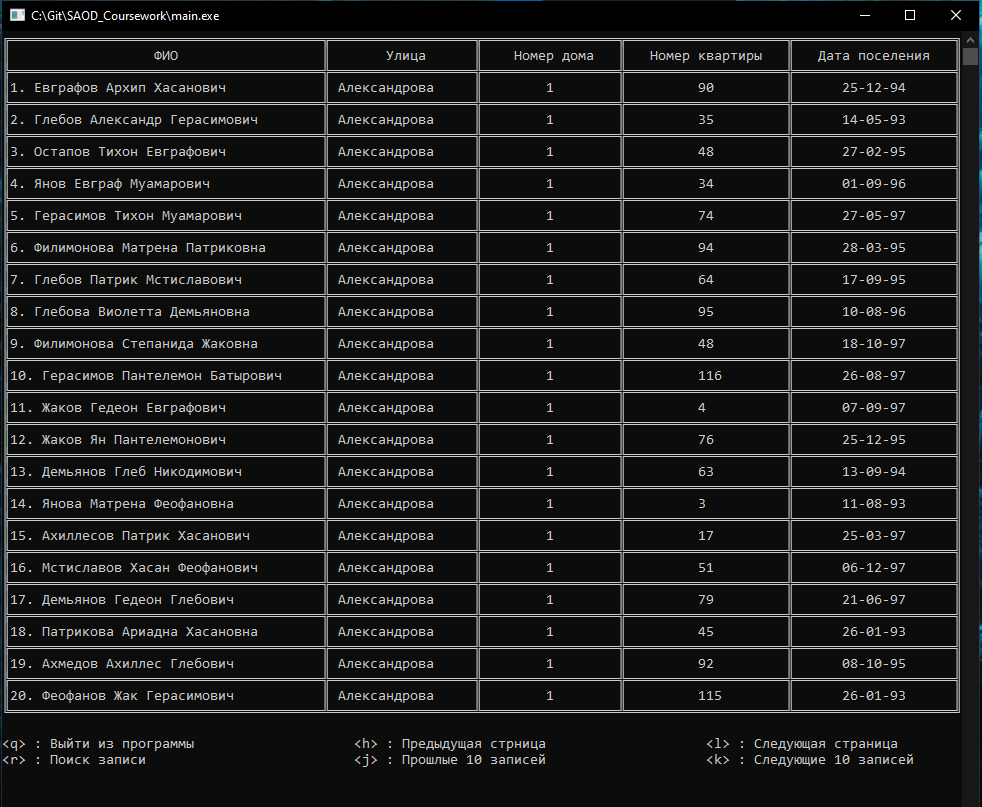


Рисунок 8 — Вывод отсортированноый базы данных

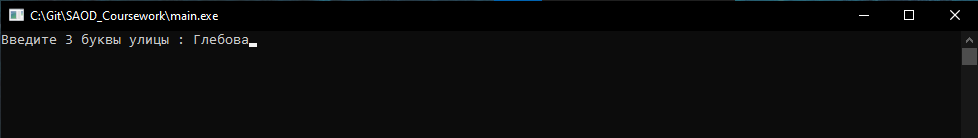


Рисунок 9 —Запрос первых 3 букв улицы для создания очереди

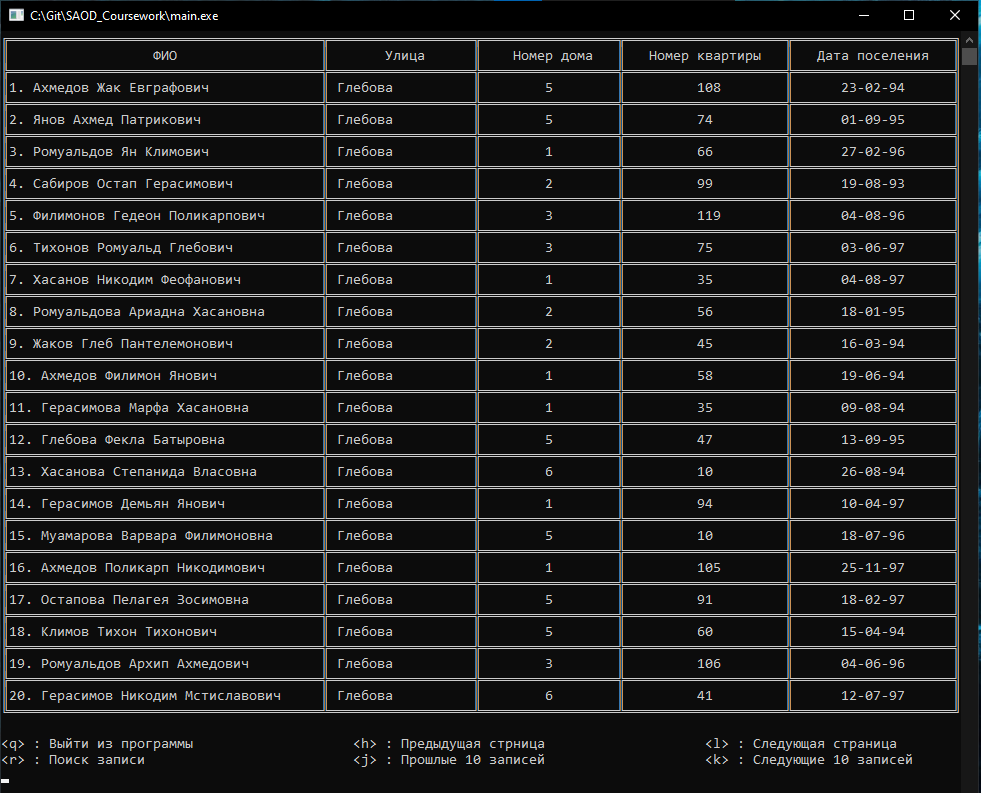


Рисунок 10 — Вывод очереди



Рисунок 11 — Вывод двоичного Б-дерева

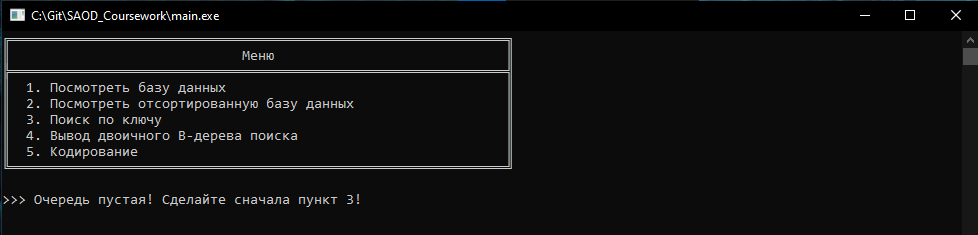


Рисунок 12 — Ошибка вывода двоичного Б-дерева из-за отсутствия очереди

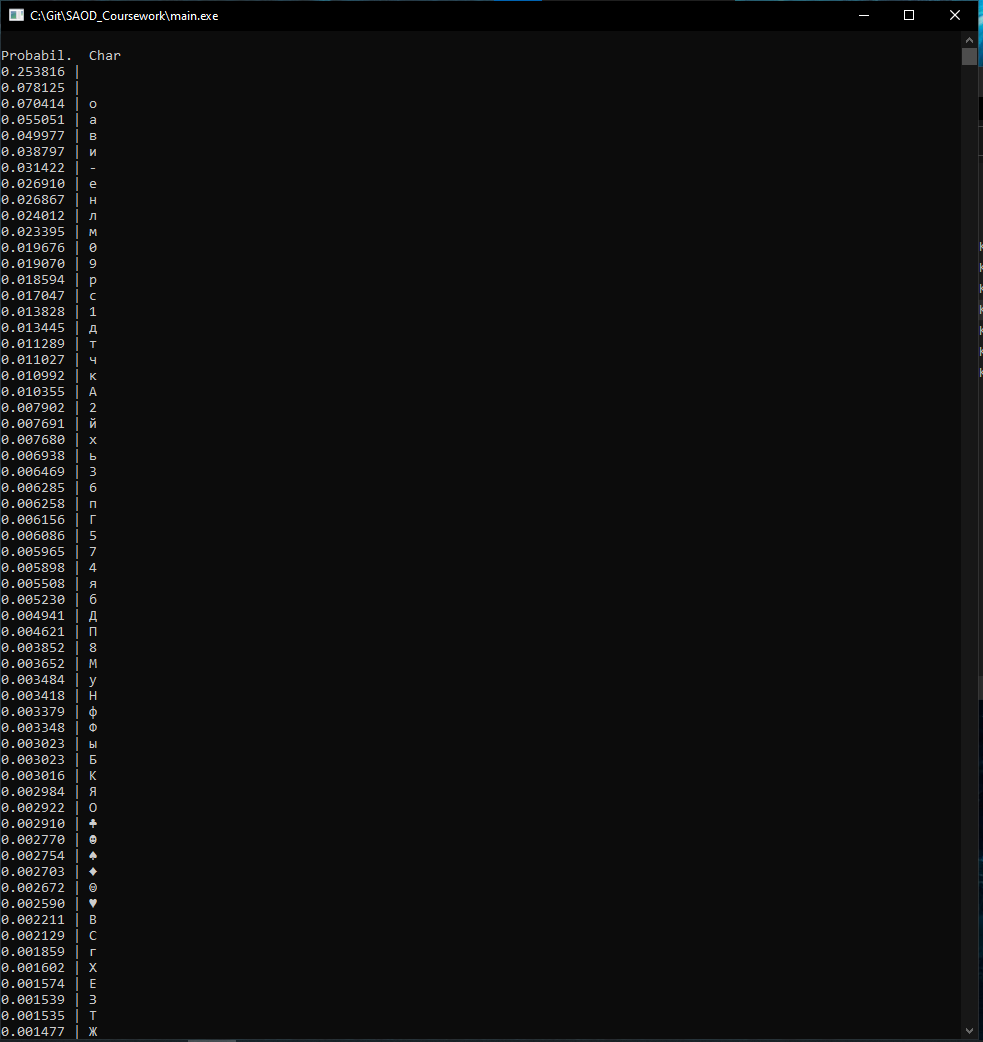


Рисунок 13 — Вывод вероятностей символа

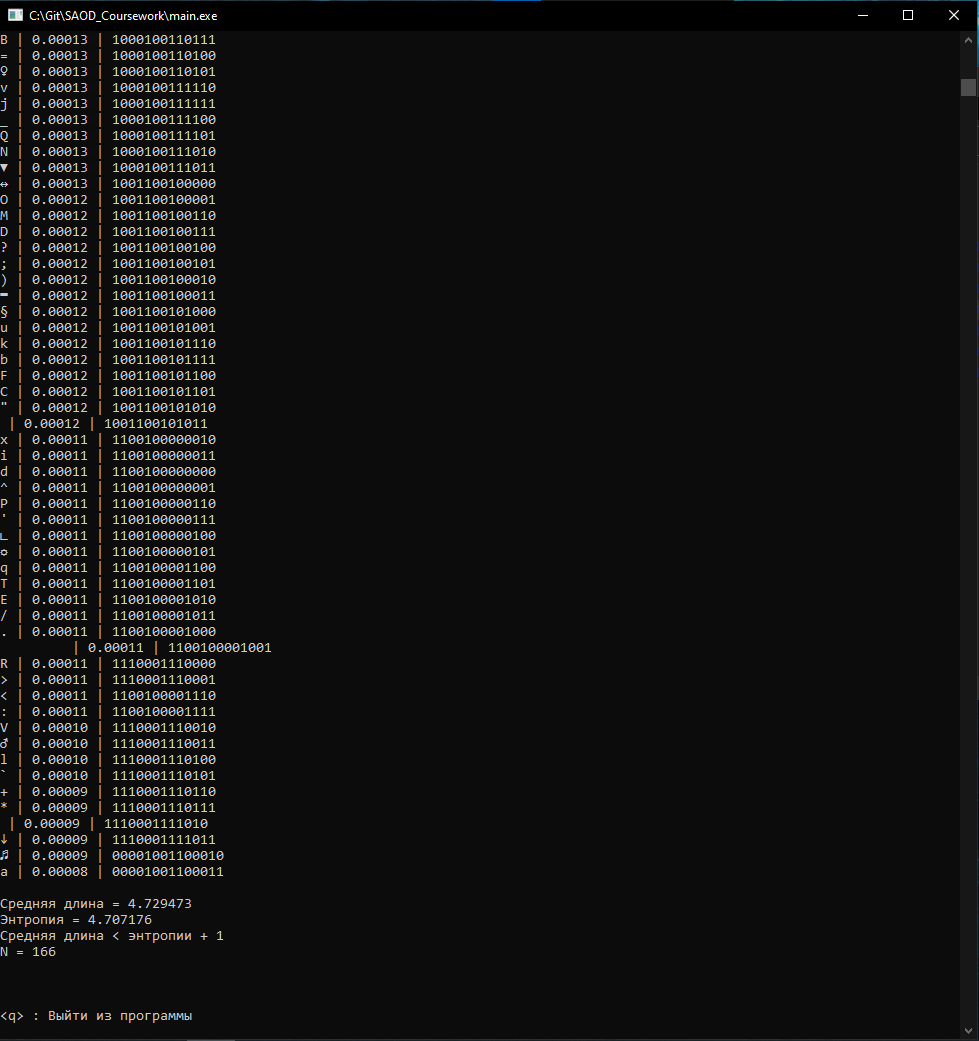


Рисунок 14 — Примеры кодовых слов для минимально вероятных символов в базе, а также вычисленная средняя длина кодового слова, энтропия источника и кол-во символов

1. ВЫВОДЫ

В ходе выполнения курсового проекта были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, двоичного поиска, создания очереди, построения двоичного бинарного дерева, поиска по дереву, кодирования данных.

Четкая структуризация кода и грамотно подобранные имена переменных, структур данных, функций и процедур способствуют удобочитаемости программы.

Все разработанные алгоритмы расширяют возможности работы с данными и способствуют улучшению эффективности анализа и обработки данных и представляют собой минимальный набор процедур для представления и обработки базы данных.