Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовой проект

### по курсу

### «Структуры и алгоритмы обработки данных»

### Вариант -14

Выполнил: студент группы ИП-311

Николаев М.Ю.

Проверил: доцент кафедры ПМиК

Янченко Е.В.

Новосибирск 2024

### Содержание

[1. ПОСТАНОВКА ЗАДАЧИ 3](#_Toc20704)

[2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ 4](#_Toc20705)

[2.1. МЕТОД СОРТИРОВКИ 4](#_Toc20706)

[2.2 ДВОИЧНЫЙ ПОИСК 4](#_Toc20707)

[2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ 5](#_Toc20708)

[3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ………………………………5-6](#_Toc20709)

[4. ОПИСАНИЕ ПРОГРАММЫ 7](#_Toc20710)

[4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУР 8-9](#_Toc20711)

[4.2. ОПИСАНИЕ ПОДПРОГРАММ 9-11](#_Toc20712)

[5. ТЕКСТ ПРОГРАММЫ 12-29](#_Toc20713)

[6. РЕЗУЛЬТАТЫ 30-33](#_Toc20714)

[7. ВЫВОДЫ 33](#_Toc20715)

1. ПОСТАНОВКА ЗАДАЧИ

Хранящуюся в файле базу данных загрузить в оперативную память компьютера и построить индексный массив, упорядочивающий данные по ФИО и названию улицы, используя метод Хоара в качестве метода сортировки. Предусмотреть возможность поиска по ключу в упорядоченной базе, в результате которого из записей с одинаковым ключом формируется очередь, содержимое очереди выводится на экран. Из записей очереди построить дерево оптимального поиска (А1) по дате, и предусмотреть возможность поиска в дереве по запросу. Закодировать файл базы данных почти оптимальным кодом Шеннона, предварительно оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран.

Стpуктуpа записи:

- ФИО гражданина: текстовое поле 32 символа

формат <Фамилия>\_<Имя>\_<Отчество> Название улицы: текстовое поле 20 символов Номер дома: целое число Номер квартиры: целое число Дата поселения: текстовое поле 8 символов фоpмат дд-мм-гг

2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

2.1. МЕТОД СОРТИРОВКИ

Метод Хоара

Метод Хоара или метод быстрой сортировки заключается в следующем. Возьмём произвольный элемент массива х. Просматривая массив слева, найдём элемент ai ≥x. Просматривая массив справа, найдём aj ≤x. Поменяем местами ai и aj . Будем продолжать процесс просмотра и обмена, до тех пор, пока i не станет больше j. Тогда массив можно разбить на две части: в левой части все элементы не больше х, в правой части массива не меньше х. Затем к каждой части массива применяется тот же алгоритм. Очевидно, трудоёмкость метода существенно зависит от выбора элемента х, который влияет на разделение массива. Максимальные значения М и С для метода быстрой сортировки достигаются при сортировке упорядоченных массивов (в прямом и обратном порядке). Тогда в этом случае в одной части остаётся только один элемент (минимальный или максимальный), а во второй – все остальные элементы. Выражения для М и С имеют следующий вид M=3(n-1), C=(n^2+5n+4)/2 Таким образом, в случае упорядоченных массивов трудоёмкость сортировки имеет квадратичный порядок. Элемент am называется медианой для элементов aL…aR, если количество элементов меньших am равно количеству элементов больших am с точностью до одного элемента (если количество элементов нечётно). Асимптотические оценки для средних значений М и С имеют следующий вид С=О(nlogn), М=О(nlogn) при n → ∞. Метод Хоара неустойчив.

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.

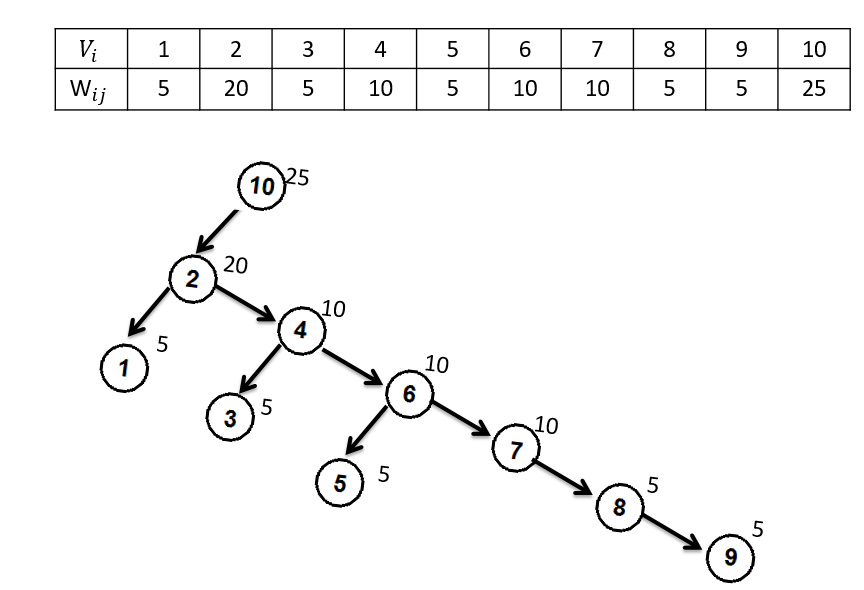
3. Выбранный элемент больше X. Продолжаем поиск в левой половине массива.

Двоичный поиск имеет две реализации. У первой версии алгоритма есть недостаток: на каждой итерации совершается два сравнения, но можно обойтись одним сравнением. Если в массиве несколько элементов с одинаковым ключом, то эта версия находит один из них. Поэтому в курсовом проекте использовалась вторая версия алгоритма. Если в массиве несколько элементов с одинаковым ключом, то эта версия алгоритма находит самый левый из них. Для поиска остальных элементов с заданным ключом требуется просмотреть массив только в одном направлении – вправо от найденного элемента. 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

Приближённый алгоритм дерева оптимального поиска (А1)

Как известно, точный алгоритм построения дерева оптимального поиска имеет квадратичную трудоёмкость. При больших объемах деревьев такие алгоритмы становятся неэффективными. Известны быстрые алгоритмы, строящие почти оптимальные деревья. Приближённый алгоритм дерева оптимального поиска (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.



**Вершины приближённого алгоритма дерева оптимального поиска (А1)**

2.4 МЕТОД КОДИРОВАНИЯ

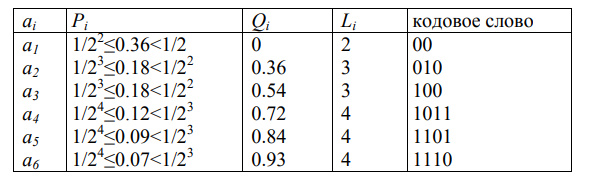
Код Шеннона

Код Шеннона позволяет построить почти оптимальный код с длинами кодовых слов Li < - log pi +1. Тогда Lcp. 1. Упорядочим символы исходного алфавита А={a1, a2,…,an} по убыванию их вероятностей: p1≥p2≥p3≥…≥pn. 2. Составим нарастающие суммы вероятностей Qi : Q0=0, Q1=p1, Q2=p1+p2, Q3=p1+p2+p3, … , Qn=1. 3. Представим Qi в двоичной системе счисления и возьмем в качестве кодового слова первые ⎡- log2pi⎤ знаков после запятой.

Для вероятностей, представленных в виде десятичных дробей, удобно определить длину кодового слова Li из соотношения



**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.



Вычислим среднюю длину кодового слова. Заметим, что она не превышает значение энтропии.

Lср= 0.36. 2+(0.18+0.18). 3+(0.12+0.09+0.07). 4=2.92< 2.37+1

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

1. З*агрузка и вывод базы данных*

Для загрузки базы данных в проекте реализована процедура readData(), которая считывает записи типа record из базы данных «Населённый пункт» testBase4.dat. Для вывода данных используется функция printAll(), которая отображает все записи базы данных на экране. Она также позволяет выводить записи по страницам, с возможностью просмотра 20 записей на странице, что реализовано через функцию printPage(). Пользователь может легко переходить по записям, переходя на следующую или предыдущую страницу, а также выбирать конкретную страницу.

2. *Вспомогательные функции и процедуры для сортировки данных*

Сортировка записей реализована через процедуру QuickSortV2(), которая использует алгоритм быстрых сортировок для упорядочивания данных по имени и адресу. Для доступа к записям используется индексный массив indexData, который облегчает процесс сортировки и поиска. Функция compareRecords() сравнивает записи, обеспечивая корректную сортировку сначала по ФИО, а затем по адресам. Если ФИО совпадают, выполняется дополнительное сравнение по адресу, что дает возможность создать структурированное и удобное представление данных.

3. *Особенности реализации бинарного поиска и построения очереди*

Поиск по отсортированным данным осуществляется через процедуру binSearch(), которая реализует бинарный поиск. Эта процедура принимает ключ и возвращает указатель на созданную очередь, содержащую найденные записи. Найденные записи добавляются в очередь с помощью функции addToQue(), а для вывода на экран используется функция printQue(). При реализации бинарного поиска была использована его вторая версия, так как в результате ее выполнения возвращается номер самого левого из найденных элементов, благодаря чему легко найти и вывести остальные элементы, лишь просмотрев оставшуюся правую часть массива, пока не встретится запись, не удовлетворяющая ключу поиска.

4. *Особенности построения дерева и его вывод на экран*

Построение дерева оптимального поиска A1 осуществляется в функции A1(), которая возвращает указатель на корень дерева. Внутри этой функции происходит инициализация очереди элементов, каждая запись из исходного массива данных преобразуется в элемент очереди и добавляется в массив. Данная структура данных сортируется с использованием модифицированного метода быстрой сортировки QuickSortV2Ubiv(), что позволяет расположить элементы в порядке их «веса». Добавление элементов в дерево выполняется с помощью функции addRST\_R. При добавлении элемента в дерево особое внимание уделяется тому, как обрабатываются элементы с одинаковыми ключами. В случае, если ключ элемента совпадает с ключом узла дерева (data->data->date), новый элемент добавляется в центральное поддерево (center), что позволяет сохранять все элементы с одинаковыми ключами в одном месте дерева. Процедура printTree() используется для вывода дерева на экран. Она реализует обход слева направо. Поиск элементов в дереве осуществляется с помощью функции search\_in\_tree(). Эта функция принимает указатель на корень дерева, ключ, по которому производится поиск, и индекс, который увеличивается при нахождении элементов. Для начала поиска функция сравнивает ключ с данными в корне дерева. Если они совпадают, осуществляется вывод всех элементов, находящихся в центральном поддереве. Если ключ поиска меньше, чем ключ корня, поиск рекурсивно продолжается в левом поддереве; если больше — в правом поддереве.

5*. Кодирование данных*

Кодирование базы данных с использованием алгоритма Шеннона начинается при выборе пользователем режима «mode == 5». Сначала происходит считывание базы данных. После считывания данных создается массив Symbols, каждый элемент которого представляет собой структуру, содержащую символ и его вероятность появления. Вероятность вычисляется как отношение количества появления каждого символа к общему количеству символов в базе данных. Основная часть кодирования происходит в функции ShannonCode Для формирования почти оптимального кода символы сортируются по вероятностям их появления. Это достигается с помощью алгоритма быстрой сортировки, что дает возможность быстро упорядочить массив и получить доступ к символам с высокой вероятностью. Для каждого символа определяется длина его кода. Функция calEntropy() вычисляет энтропию. В конце процесса кодирования результаты выводятся в формате таблицы, в которой представлены коды символов, их вероятности, длина кодов и прочие данные.

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ

глобальные переменные и константы:

const char\* base = "testBase4.dat"; Константа, содержащая имя файла с базой данных. int countRecords; Переменная типа int, хранящая количество записей в базе данных. const int n = 256; Константа типа int, задающая размер алфавита для кодирования. record\* pData = NULL; Указатель на массив структур типа record. record\*\* indexData = NULL; Указатель на массив указателей на структуры типа record, используемый для индексации записей. const int PAGE\_SIZE = 20; Константа, задающая размер страницы (количество записей, отображаемых на одной странице).

Структуры:

typedef struct record { структура, используемая для работы с базой данных «Населенный пункт» char name[32]; Поле для хранения имени (ФИО). char address[18]; Поле для хранения адреса (названия улицы) short int house\_number; Поле для хранения номера дома. short int kvartira\_number; Поле для хранения номера квартиры. char date[10]; Поле для хранения даты поселения в формате ДД-ММГГ } record;

Эта структура представляет собой элемент очереди, который содержит данные и указатель на следующий элемент.

struct queEl { record\* data; Указатель на данные типа `record`. queEl\* next = NULL; Указатель на следующий элемент в очереди. int wes; Переменная для хранения дополнительной информации (вес). };

Эта структура используется для представления самой очереди, хранящей элементы типа queEl.

struct que { queEl\* head = NULL; Указатель на начало очереди. queEl\* tail = NULL; Указатель на конец очереди. };

Эта структура представляет вершину в дереве, хранящую данные, а также указатели на левые, правые и центральные поддеревья.

struct vertex { record\* data; Указатель на данные типа `record`. vertex\* L = nullptr; Указатель на левое поддерево. vertex\* R = nullptr; Указатель на правое поддерево. vertex\* center = NULL; Указатель на центральное поддерево. int wes; Переменная для хранения дополнительной информации (вес). };

Эта структура предназначена для хранения информации о символах, используемых в процессе кодирования.

struct Symbol { char ch = 0; Символ, который кодируется. float chance = 0; Вероятность появления символа. };

Эта структура используется для хранения информации о кодах, созданных по алгоритму Шеннона.

struct Shannon { char symbol = 0; Кодируемый символ. float Pi = 0; Вероятность символа. float Qi = 0; Кумулятивная вероятность. unsigned short int L = 0; Длина кода для данного символа. char\* word = nullptr; Указатель на массив символов, представляющий код. };

4.2. ОПИСАНИЕ ПОДПРОГРАММ

Процедуры для обработки базы данных:

1. int readData() - эта функция отвечает за чтение данных из файла базы данных, который называется testBase4.dat. Она открывает файл в бинарном режиме и определяет количество записей, считав размер файла, деля его на размер структуры record. После этого данные считываются в динамически выделенный массив pData.

2. void printElement(const record\* p, int index) - эта функция выводит на экран отдельный элемент (запись) базы данных. Выводится индекс записи, имя, адрес, номер дома, номер квартиры и дата.

3. void printAll() - процедура отвечает за вывод всех записей из базы данных. Она вызывает printElement для каждой записи, используя индексированный массив indexData, чтобы вывести все элементы последовательно.

4. void printPage(int start) - эта функция выводит на экран записи, начиная с указанного индекса. Она учитывает размер "страницы" (количество записей, выведенных за один раз, определяемое константой PAGE\_SIZE), и выводит записи в заданном диапазоне. 5

5. void printDBPageByPage() - процедура реализует шаблон постраничного просмотра. Она выводит на экран записи по 20 элементов за раз, предоставляет возможность пользователю перемещаться между страницами, выбирать конкретную страницу или выходить в основное меню.

Функции и процедуры сортировки:

6. void QuickSortV2(record\* MyArray[], int L, int R) - эта функция реализует алгоритм быстрой сортировки для массива указателей на записи структуры record. Сортировка выполняется за среднее время O(nlogn). Происходит по полям ФИО и по названию улицы.

7. void createIndexMass(int size) - процедура создает индексный массив указателей на записи базы данных. Она принимает размер массива в качестве параметра и выделяет память для индекса indexData, в который заносятся адреса записей из массива pData. Это упрощает сортировку и поиск, так как вместо копирования данных используется работа с указателями.

8. void compareRecords(const record\* record1, const record\* record2) – это функция, которая сравнивает два экземпляра структуры record по строкам name(ФИО). Если имена не равны, функция возвращает результат этого сравнения, который указывает, какая запись "меньше" или "больше" по имени. Если имена равны, функция затем сравнивает адреса (address). Результат этого сравнения возвращается впоследствии

Функции и процедуры для поиска в отсортированной базе данных:

9. que\* binSearch(record\* indexData[], int size, char\* key) - эта функция реализует бинарный поиск в отсортированном массиве указателей на record. Она принимает массив записей и ключ для поиска, а также размер массива. Алгоритм осуществляет поиск, возвращая очередь (que), содержащую все записи, соответствующие ключу. Если соответствие найдено, первая запись помещается в очередь; функция продолжает добавлять все совпадающие записи, пока не достигнет конца массива.

Функции построения приближённого алгоритма дерева оптимального поиска (А1) и поиска в нем

10. vertex\* A1(record\* V[], int size) - эта функция отвечает за построение дерева оптимального поиска (А1). С помощью сортировки по убыванию весов, происходит добавление с помощью рекурсивной функцией addRST\_R для добавления элементов в дерево. Каждый элемент данных помещается в дерево.

11. void addRST\_R(vertex\*& p, queEl\* data, int& kolv) - эта рекурсивная функция добавляет новый элемент data в дерево поиска.

12. void search\_in\_tree(vertex\* root, char\* key, int& i) - эта функция осуществляет поиск по дереву оптимального поиска, принимая корень дерева и ключ для поиска.

13. void printTreeElement(vertex\* p, int& index) - эта функция предназначена для печати всех записей, находящихся в узле и его центральной части.

14. void printTree(vertex\* root, int& index) – эта функция отвечает за печать всех элементов в дереве в порядке обхода.

15. void QuickSortV2Ubiv(queEl\*\* MyArray, int L, int R) – эта функция реализует алгоритм быстрой сортировки для сортировки весов по убыванию.

16. int compareWes(int wes1, int wes2) – эта функция предназначена для сравнения двух весов.

Процедуры и функции кодирования базы данных:

17. Shannon\* ShannonCode(Symbol\* Symbols, int n) - эта функция реализует алгоритм кода Шеннона. Функция сортирует символы по убыванию вероятности, затем для каждого символа рассчитывает длину кода и кодовое слово.

18. float calEntropy(Symbol\* A, int n) – эта функция рассчитывает энтропию источника данных.

19. float checkKraft(Shannon\* C, int n) – эта функция проверяет условие Крафта.

20. void printTable2(float entropy, float avgLength, float kraftCheck) - эта функция выводит на экран таблицу с расчетами, включающую значение Крафта, энтропию и среднюю длину кода.

21. void printTableSymbols(Shannon\* C, int n) – эта функция выводит на экран таблицу символов, их вероятностей и созданных кодов.

Основная программа:

В основной программе происходит бесконечный цикл, который вызывает функцию menu() для отображения доступных опций и получения выбора пользователя.

22. menu() – эта функция выводит список из 5 доступных действий и запрашивает у пользователя, что он может сделать.

23. mainMenu(int mode) – эта функция отвечает за обработку выбора пользователя. В зависимости от значения параметра mode, вызываются соответствующие функции.

5. ТЕКСТ ПРОГРАММЫ

#include <conio.h>

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <iostream>

#include <iomanip>

#include <fstream>

#include <string>

#include <unordered\_map>

#include <algorithm>

#include <cmath>

const char\* base = "testBase4.dat";

typedef struct record {

char name[32];

char address[18];

short int house\_number;

short int kvartira\_number;

char date[10];

} record;

struct queEl {

record\* data;

queEl\* next = NULL;

int wes;

};

struct que {

queEl\* head = NULL;

queEl\* tail = NULL;

};

struct vertex {

record\* data;

vertex\* L = nullptr;

vertex\* R = nullptr;

vertex\* center = NULL;

int wes;

};

struct Symbol {

char ch = 0;

float chance = 0;

};

struct Shannon {

char symbol = 0;

float Pi = 0;

float Qi = 0;

unsigned short int L = 0;

char\* word = nullptr;

};

using namespace std;

int countRecords;

const int n = 256;

record\* pData = NULL;

record\*\* indexData = NULL;

const int PAGE\_SIZE = 20;

void printElement(const record\* p, int index) {

printf("%6d ", index + 1);

printf("%32.32s ", p->name);

printf("St. ");

printf("%18.18s ", p->address);

printf("dom.");

printf("%02d ", p->house\_number);

printf("kv.");

printf("%03d ", p->kvartira\_number);

printf("%10.10s ", p->date);

printf("\n");

}

int readData() {

FILE\* f;

int count;

fopen\_s(&f, base, "rb");

if (f == NULL) {

printf("\nError: file base4.dat not found.");

printf("\nPlease check the file location.\n");

exit(1);

}

fseek(f, 0, SEEK\_END);

count = ftell(f) / sizeof(record);

if (count != 0) {

fseek(f, 0, SEEK\_SET);

pData = (record\*)malloc(sizeof(record) \* count);

fread(pData, sizeof(record), count, f);

}

fclose(f);

return count;

}

queEl\* initQueEl(record\* data) {

queEl\* newQueEl = new queEl;

newQueEl->data = data;

return newQueEl;

}

que\* initQue() {

que\* newQue = new que;

newQue->tail = (queEl\*)&(newQue->head);

return newQue;

}

void addToQue(que\* curQue, queEl\* rec) {

curQue->tail->next = rec;

curQue->tail = rec;

}

int searchCompare(record\* rec, char\* key) {

return strncmp(rec->name, key, 3);

}

int compareRecords(const record\* record1, const record\* record2) {

int cmp = strcmp(record1->name, record2->name);

if (cmp != 0) {

return cmp;

}

cmp = strcmp(record1->address, record2->address);

return cmp;

}

void QuickSortV2(record\* MyArray[], int L, int R)

{

while (L < R)

{

int i = L, j = R;

record x = \*indexData[L];

while (i < j)

{

while (compareRecords(indexData[i], &x) < 0) {

i++;

}

while (compareRecords(indexData[j], &x) > 0) {

j--;

}

if (i <= j)

{

record\* temp = indexData[i];

indexData[i] = indexData[j];

indexData[j] = temp;

i++;

j--;

}

}

if ((j - L) < (R - i))

{

QuickSortV2(MyArray, L, j);

L = i;

}

else

{

QuickSortV2(MyArray, i, R);

R = j;

}

}

}

int compareWes(int wes1, int wes2) {

return wes1 > wes2;

}

void QuickSortV2Ubiv(queEl\*\* MyArray, int L, int R)

{

while (L < R)

{

int i = L, j = R;

int x = MyArray[L]->wes;

while (i < j)

{

while (compareWes(MyArray[i]->wes, x)) {

i++;

}

while (compareWes(x, MyArray[j]->wes)) {

j--;

}

if (i <= j)

{

queEl\* temp = MyArray[i];

MyArray[i] = MyArray[j];

MyArray[j] = temp;

i++;

j--;

}

}

if ((j - L) < (R - i))

{

QuickSortV2Ubiv(MyArray, L, j);

L = i;

}

else

{

QuickSortV2Ubiv(MyArray, i, R);

R = j;

}

}

}

void QuickSortV2Shennon(Symbol\*& A, int R, int L, unsigned short int field, bool reverse) {

while (L < R) {

float x;

if (field == 0)

x = A[L].ch;

else if (field == 1)

x = A[L].chance;

int i = L;

int j = R;

while (i <= j) {

if (field == 0) {

if (reverse) {

while (A[i].ch > x)

i++;

while (A[j].ch < x)

j--;

}

else {

while (A[i].ch < x)

i++;

while (A[j].ch > x)

j--;

}

}

else if (field == 1) {

if (reverse) {

while (A[i].chance > x)

i++;

while (A[j].chance < x)

j--;

}

else {

while (A[i].chance < x)

i++;

while (A[j].chance > x)

j--;

}

}

if (i <= j) {

Symbol temp = A[i];

A[i] = A[j];

A[j] = temp;

i++;

j--;

}

}

if (j - L > R - i) {

QuickSortV2Shennon(A, R, i, field, reverse);

R = j;

}

else {

QuickSortV2Shennon(A, j, L, field, reverse);

L = i;

}

}

}

que\* binSearch(record\* indexData[], int size, char\* key) {

int L = 0, R = size - 1, m;

while (L < R) {

m = (L + R) / 2;

if (searchCompare(indexData[m], key) < 0) {

L = m + 1;

}

else {

R = m;

}

}

que\* newQue = initQue();

newQue->head = initQueEl(indexData[R]);

newQue->tail = newQue->head;

R++;

while (R < size && searchCompare(indexData[R], key) == 0) {

addToQue(newQue, initQueEl(indexData[R]));

R++;

}

return newQue;

}

int searchInTree(vertex\*& p, char\* r) {

return strncmp(p->data->date, r, 8);

}

void addRST\_R(vertex\*& p, queEl\* data, int& kolv)

{

if (!p)

{

p = new vertex;

p->data = data->data;

p->wes = data->wes;

kolv++;

}

else if ((searchInTree(p, data->data->date)) > 0)

{

addRST\_R(p->L, data, kolv);

}

else if (searchInTree(p, data->data->date) < 0)

{

addRST\_R(p->R, data, kolv);

}

else {

addRST\_R(p->center, data, kolv);

}

}

void printTreeElement(vertex\* p, int& index) {

vertex\* cursor = p;

while (cursor) {

index++;

printElement(cursor->data, index);

cursor = cursor->center;

}

}

void printTree(vertex\* root, int& index) {

if (!root)

return;

printTree(root->L, index);

printTreeElement(root, index);

printTree(root->R, index);

}

vertex\* A1(record\* V[], int size)

{

queEl\*\* arr = new queEl \* [size];

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

arr[i] = initQueEl(V[i]);

arr[i]->wes = rand() % 100;

}

QuickSortV2Ubiv(arr, 0, size - 1);

vertex\* root = nullptr;

int size1 = 0;

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

addRST\_R(root, arr[i], size1);

}

int index = -1;

printTree(root, index);

cout << "size of array: " << size << endl << "size of tree: " << size1 << endl;

return root;

}

void search\_in\_tree(vertex\* root, char\* key, int& i)

{

if (root)

{

if (searchInTree(root, key) < 0)

{

search\_in\_tree(root->R, key, i);

}

else if (searchInTree(root, key) > 0)

{

search\_in\_tree(root->L, key, i);

}

else if (searchInTree(root, key) == 0)

{

printTreeElement(root, i);

return;

}

}

}

Shannon\* ShannonCode(Symbol\* Symbols, int n) {

QuickSortV2Shennon(Symbols, n - 1, 0, 1, true);

Shannon\* C = new Shannon[n];

C[0].symbol = Symbols[0].ch;

C[0].Pi = Symbols[0].chance;

C[0].L = ceil(-log2(C[0].Pi));

C[0].Qi = 0;

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

C[i].symbol = Symbols[i].ch;

C[i].Pi = Symbols[i].chance;

C[i].L = ceil(-log2(C[i].Pi));

C[i].Qi = C[i - 1].Qi + C[i - 1].Pi;

}

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

float Qtemp = C[i].Qi;

C[i].word = new char[C[i].L];

for (int j = 0; j < C[i].L; j++) {

Qtemp = Qtemp \* 2;

C[i].word[j] = (char)(floor(Qtemp) + 48);

if (Qtemp >= 1) Qtemp = Qtemp - 1;

}

}

return C;

}

float calEntropy(Symbol\* A, int n) {

float result = 0;

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

result += (A[i].chance \* log2(A[i].chance));

}

return -result;

}

float calAVGLength(Shannon\* A, int n) {

float result = 0;

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

result += A[i].L \* A[i].Pi;

}

return result;

}

float checkKraft(Shannon\* C, int n) {

float kraftSum = 0;

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

kraftSum += pow(2, -C[i].L);

}

return kraftSum;

}

void printTable2(float entropy, float avgLength, float kraftCheck) {

cout << "\n-------------------------------------------------------------\n";

cout << "| Kraft | Entropy | Average lenght | Code word |\n";

cout << "-------------------------------------------------------------\n";

cout << "| " << setw(9) << fixed << setprecision(5) << kraftCheck

<< " | " << setw(10) << entropy

<< " | " << setw(14) << avgLength

<< " | " << setw(12) << avgLength - entropy << " |\n";

cout << "-------------------------------------------------------------\n";

}

void printTableSymbols(Shannon\* C, int n) {

cout << "----------------------------------------------------------\n";

cout << "| Alphavit | Symbol | Pi | Li | Code word |\n";

cout << "----------------------------------------------------------\n";

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

cout << "| " << setw(8) << i + 1 << " | " << setw(7) << C[i].symbol

<< " | " << setw(11) << fixed << setprecision(4) << C[i].Pi

<< " | " << setw(6) << C[i].L << " | ";

for (int j = 0; j < C[i].L; j++) {

cout << C[i].word[j];

}

cout << " |\n";

}

cout << "----------------------------------------------------------\n";

}

void Print\_tree(vertex\* p, int& i)

{

if (p)

{

Print\_tree(p->L, i);

printTreeElement(p, i);

Print\_tree(p->R, i);

i++;

}

}

void printAll() {

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

printElement(indexData[i], i);

}

}

void printPage(int start) {

int end = start + PAGE\_SIZE;

if (end > countRecords) end = countRecords;

for (int i = start; i < end; i++) {

printElement(indexData[i], i);

}

}

void printQue(que\* queToPrint) {

queEl\* cursor = queToPrint->head;

int ind = 0;

while (cursor) {

printElement(cursor->data, ind);

ind++;

cursor = cursor->next;

}

}

void printDBPageByPage() {

int totalPages = 200;

int currentPage = 0;

bool exitToMenu = false;

while (!exitToMenu) {

int start = currentPage \* PAGE\_SIZE;

int end = start + PAGE\_SIZE - 1;

if (end >= countRecords) end = countRecords - 1;

printPage(start);

printf("\n"

"1 - Next page\n"

"2 - Choose the page\n"

"3 - Previous page\n"

"4 - Sorting by FIO and name street\n"

"5 - Searching by FIO\n");

printf("0 - Exit to main menu\n");

int choice;

scanf\_s("%d", &choice);

if (choice == 0) {

exitToMenu = true;

}

else if (choice == 1 && currentPage < totalPages - 1) {

currentPage++;

}

else if (choice == 2) {

int page;

printf("Choose the page (1 - %d): ", totalPages);

scanf\_s("%d", &page);

if (page < 1 || page > totalPages) {

printf("Failed choice of the page.\n");

}

currentPage = page - 1;

}

else if (choice == 3 && currentPage > 0) {

currentPage--;

}

else if (choice == 4) {

QuickSortV2(indexData, start, end);

}

else if (choice == 5) {

printf("Input key to search by FIO:\n");

char\* key = new char[4];

scanf\_s("%s", key);

que\* findQue = binSearch(indexData, countRecords, key);

printQue(findQue);

delete[] key;

}

else {

continue;

}

}

}

void displayAll() {

printAll();

while (1) {

printf("\n1 - Sort by FIO and street\n0 - Exit\n");

int choice;

scanf\_s("%d", &choice);

if (choice == 0) return;

if (choice == 1) {

QuickSortV2(indexData, 0, countRecords - 1);

printAll();

break;

}

else {

continue;

}

}

while (1) {

printf("\n1 - Search by three symbols of surname\n0 - Exit\n");

int choice;

scanf\_s("%d", &choice);

if (choice == 0) return;

else if (choice == 1) {

printf("Input key to search:\n");

char\* key = new char[5];

scanf\_s("%s", key);

que\* findQue = binSearch(indexData, countRecords, key);

printQue(findQue);

break;

}

else {

continue;

}

}

}

vertex\* root = NULL;

void mainMenu(int mode) {

if (mode == 1) {

displayAll();

}

else if (mode == 2) {

printDBPageByPage();

}

else if (mode == 3) {

root = A1(indexData, countRecords);

}

else if (mode == 4) {

if (!root) {

cout << "no tree" << endl;

return;

}

char\* key = new char[5];

cout << endl << "input key: ";

scanf\_s("%s", key);

int index = 0;

search\_in\_tree(root, key, index);

}

else if (mode == 5) {

int windows866[n] = { 0 };

int totalNums = 0, Unique\_Element = 0;

char ch;

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

while (file.read(&ch, sizeof(ch))) {

totalNums++;

windows866[(unsigned char)ch]++;

}

file.close();

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

if (windows866[i] != 0) Unique\_Element++;

}

Symbol\* Symbols = new Symbol[Unique\_Element];

int temp = 0;

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

if (windows866[i] != 0) {

Symbols[temp].ch = (char)i;

Symbols[temp].chance = (float)windows866[i] / totalNums;

temp++;

}

}

float entropy = calEntropy(Symbols, Unique\_Element);

Shannon\* C = ShannonCode(Symbols, Unique\_Element);

float avgLength = calAVGLength(C, Unique\_Element);

float kraftCheck = checkKraft(C, Unique\_Element);

printTableSymbols(C, Unique\_Element);

printTable2(entropy, avgLength, kraftCheck);

}

}

int menu() {

int in;

printf("\n");

printf("1 - Display all records.\n");

printf("2 - Display records page by page (20 records per page).\n");

printf("3 - Create tree\n");

printf("4 - Search in tree\n");

printf("5 - Code\n");

printf("0 - Exit.\n");

scanf\_s("%d", &in);

printf("\n");

return in;

}

void createIndexMass(int size) {

indexData = new record \* [size];

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

indexData[i] = &(pData[i]);

}

}

int main() {

countRecords = readData();

createIndexMass(countRecords);

while (1) {

int mode = menu();

switch (mode) {

case 0:

printf("Program exit.\n");

exit(0);

break;

default:

mainMenu(mode);

break;

}

}

}

6. РЕЗУЛЬТАТЫ

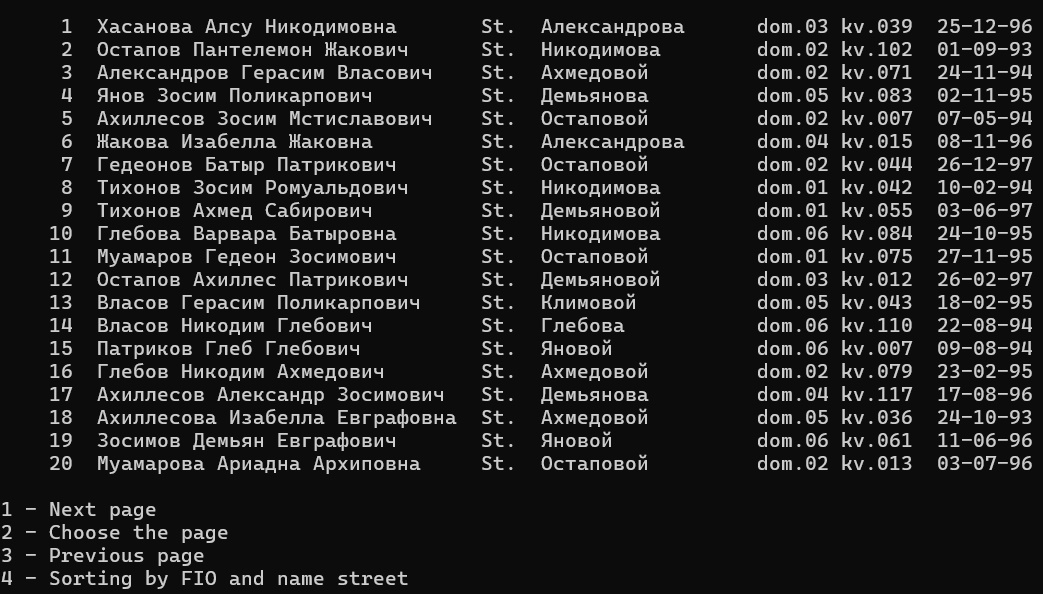


Рисунок 1. Неотсортированная база данных.

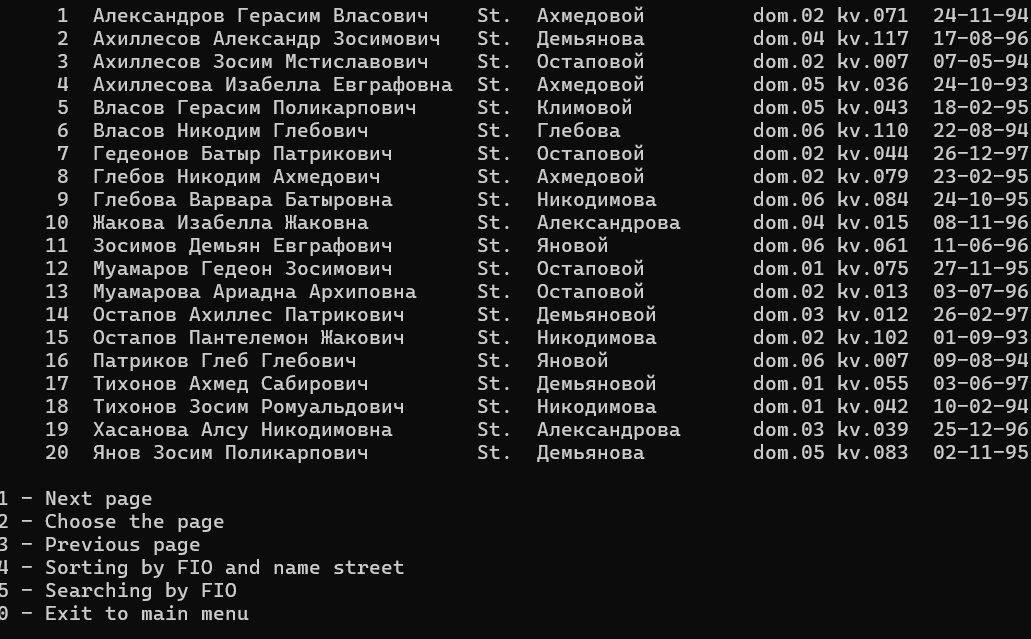


Рисунок 2. Отсортированная по ФИО и улице база данных.

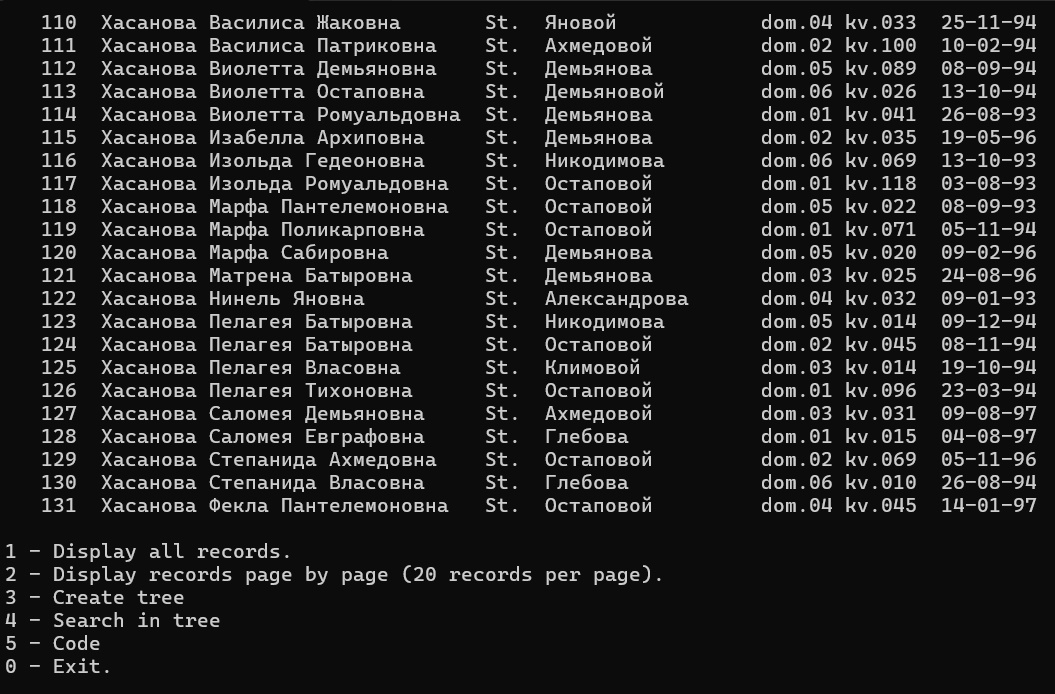


Рисунок 3. Очередь из записей, полученных в результате поиска (Фамилии начинаются на «Хас»).

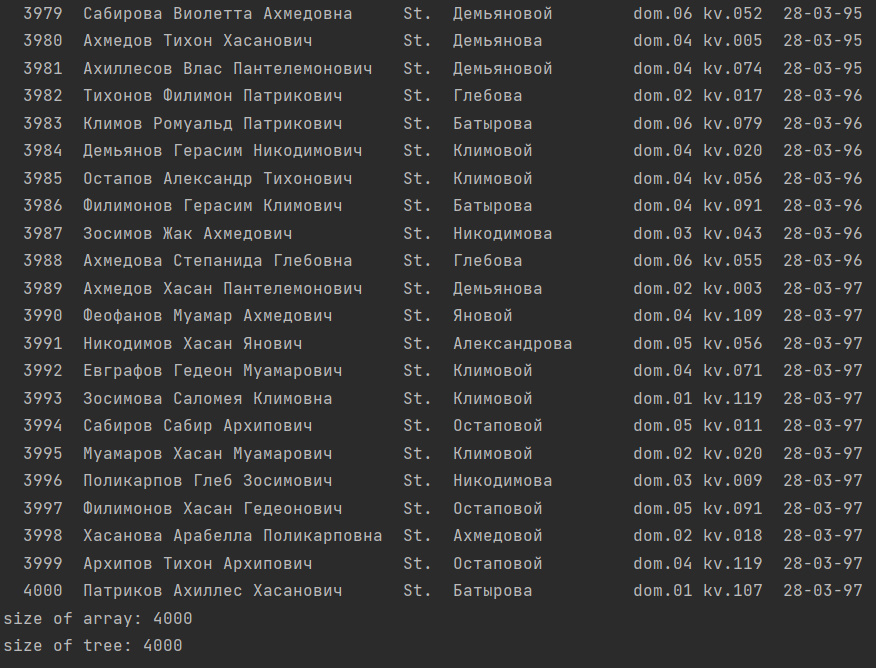


Рисунок 4. Дерево, ключ в дереве – дата.

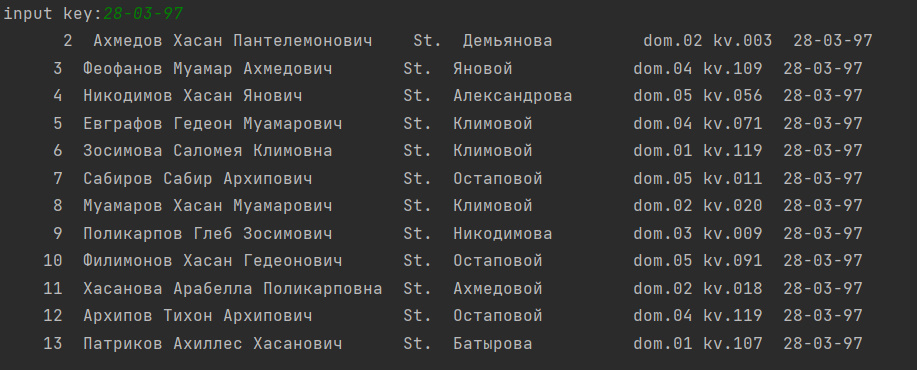


Рисунок 5. Поиск по дереву (элементы с одинаковым ключом).

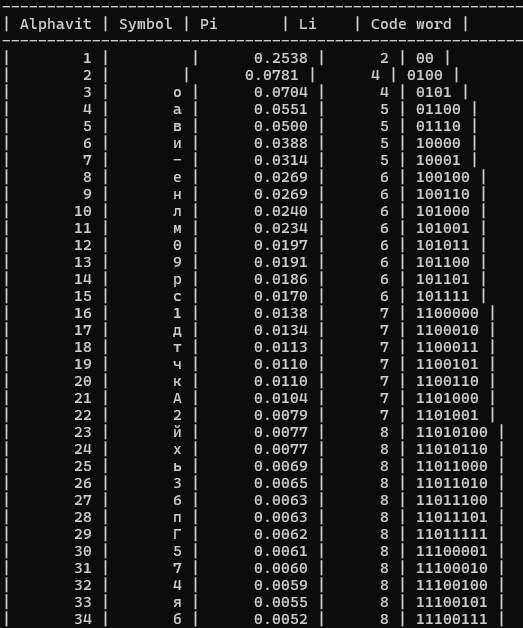


Рисунок 6. Примеры кодовых слов для наиболее вероятных символов в базе.

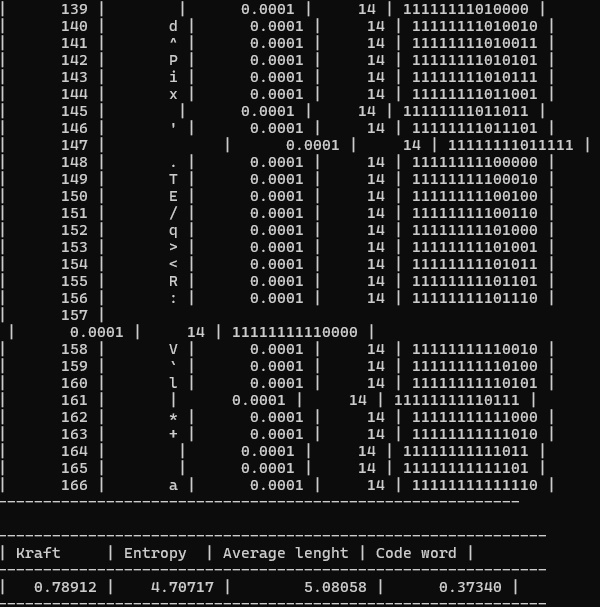


Рисунок 7. Примеры кодовых слов для минимально вероятных символов в базе, а также вычисленная средняя длина кодового слова и энтропия источника.

7. 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.